网站开发路线图

犀利开发 jQuery 内核详解与实践

- 第一部破解jQuery核心技术的先锋图书
- 生动诠释jQuery与JavaScript技术执行效率和实现方法之异同
- 使用通俗易懂的语言深度剖析jQuery框架设计模式和选择器实现原理
- 潜心研究jQuery多年，耗时近一年，国内著名原创IT作者又一力作

CD-ROM

- 本书实例素材和源代码
- CSS参考手册
- jQuery参考手册
- Ajax参考手册
- JavaScript参考手册
- HTML参考手册

朱印宏 编 著

清华大学出版社

网站开发线路图

犀利开发
——jQuery内核详解与实践

朱印宏　编　著

清华大学出版社
北京

大家网[www.topsage.com]

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

JavaScript 初学者及参考必备：

JavaScript 学习指南（第 2 版）

JavaScript: The Definitive Guide, 6th Edition (JavaScript 权威指南 第 6 版)PDF+epub

JavaScript 权威指南 (第 5 版) 中文版 | 英文版+随书源代码 | 第 4 版 英文版 |

JavaScript 高级程序设计 (第 2 版) 中文版

JavaScript Bible, 7th Edition (JavaScript 宝典 第 7 版)

JavaScript 宝典 (第 6 版) 中文版 | 英文版

JavaScript 入门经典（第 3 版）中文高清 PDF 下载

JavaScript 语言精粹 高清 PDF 中文版 | 英文 chm 版 | 英文 pdf 版

JavaScript 开发技术大全

JavaScript & DHTML Cookbook 中文版（第 2 版）

JavaScript 捷径教程

JavaScript 实战 (Practical JavaScript, DOM Scripting, and Ajax Projects) 中文版

JavaScript DOM 高级程序设计 (中文版高清 PDF 下载)

ppk 谈 JavaScript (中文高清 PDF)

JavaScript 设计模式 (Pro JavaScript Design Patterns) 中文版

JavaScript 模式

JavaScript 王者归来

Javascript Bible Golden Edition (JavaScript 金典)

JavaScript The Complete Reference (JavaScript 技术大全)

Advanced Javascript, 3rd Edition (JavaScript 高级编程)

JavaScript Examples Bible (JavaScript 实例宝典)

Wrox Beginning JavaScript (第三版)

Professional JavaScript for Web Developers

O'Reilly Head First Javascript

Pro Javascript RIA Techniques: Best Practices, Performance and Presentation

JScript 中文参考手册

Javascript 程序员字典

Special Edition Using JavaScript

Object Oriented JavaScript

O'Reilly JavaScript Patterns

JavaScript DOM 编程艺术第一版中英文 | 第二版英文

JavaScript 与 Jscript 从入门到精通

The.Pragmatic.Bookshelf 开发丛书-JavaScript 实用指南.Pragmatic.Guide.to.JavaScript.Christophe.Porteneuve.文字版

精通 JavaScript(图灵计算机科学丛书)

程序员常用 JavaScript 特效

ASP .NET 2.0 Demystified

w3school.Javascript 特效大全(上册)

High Performance JavaScript（中英文对照版）

CSS&javascript 动态网页设计与制作

JavaScript 使用手册.rar

JavaScript 网页特效范例宝典 | 源码

实用 JavaScript 网页特效编程百宝箱

JavaScript 客户端验证和页面特效制作

JavaScript 框架(JavaScript/Ajax Frameworks)：

jQuery 基础教程（第 2 版）中文高清 PDF 下载 | 英文版

jQuery 实战（jQuery in Action）中文高清 PDF 下载 | 英文版

锋利的 jQuery

jQuery 基础教程 中文高清 PDF 版

jQuery 攻略 (jQuery Recipes: A Problem Solution Approach) 中文 PDF | 英文版

15 天学会 jQuery (PDF 中文版)

O'Reilly jQuery Pocket Reference

jQuery Reference Guide

jQuery 1.4 Plugin Development Beginner's Guide

jQuery 1.2 API 全中文

jQuery 中英文对照手册

jQuery: Visual QuickStart Guide

Ext JS in Action

Prototype and Scriptaculous in Action

Prototype and script.aculo.us

精通 Dojo 中文版 PDF

Dojo: The Definitive Guide — Dojo 权威指南

Mastering Dojo: JavaScript and Ajax Tools for Great Web Experiences

Apress Practical Dojo Projects

Dojo: Using the Dojo JavaScript Library to Build Ajax Applications

Mastering Dojo: JavaScript and Ajax Tools for Great Web Experiences

Apress MooTools Essentials

Beginning Google Web Toolkit: From Novice to Professional

jQuery 开发视频教程 jQuery Projects: Creating an Interactive Photo Gallery

Pro Android Web Apps: Develop for Android using HTML5, CSS3 & JavaScript

Building.iPhone.Apps.with.HTML,CSS,and.JavaScript

AJAX (Asynchronous JavaScript and XML)：

AJAX 完全手册 (AJAX: The Complete Reference) 中文版 PDF 下载

Wiley AJAX Bible (Ajax 宝典)

MANNING AJAX In Action

Ajax 基础教程

XMLHttpRequest 中文参考手册

征服 Ajax Web 2.0 开发详解

Wrox Beginning Ajax

Wrox Professional Ajax, 2nd Edition (Ajax 高级编程)

Wrox Professional Rich Internet Applications AJAX and Beyond

O'Reilly Ajax: The Definitive Guide (Ajax 权威指南)

Beginning Javascript with DOM Scripting and Ajax 从入门到精通

Accelerated DOM Scripting with Ajax, APIs, and Libraries

Ajax Patterns and Best Practices

Head Rush Ajax

O'Reilly Ajax Hacks

O'Reilly Adding Ajax

Practical JavaScript DOM Scripting and Ajax Projects

Ajax - A New Approach to Web Applications

SEO: Search Engine Optimization Bible

AJAX 基础教程 AJAX Essential Training 视频教程系列

前　言

$表示美元符号，就是这个小符号让无数网页设计师和 Web 开发人员为之着迷，甚至折腰。实际上在 jQuery 诞生之前，已经有很多语言和框架在使用$符号了，如 Prototype 和 DWR 等。使用$符号代替 document.getElementById( )函数应该是 DOM 访问中最简捷的操作方法了。没错，jQuery 是后起之秀，是跟风者，但正是因为 jQuery，才让更多的读者、学者、设计师和开发人员把$符号铭记于心。

jQuery 确实有其无穷的魅力，我也是在这个魅力的潮流中开始研究 jQuery 并利用它进行开发的。从 2006 年到现在，不到 4 年的时间，jQuery 竟然能够聚焦全球亿万开发人员的眼球，实在是 Web 开发史上的一大奇迹。这又是为什么呢？

从 jQuery(JavaScript+Query)名称也可以看出，使用 CSS+XPath 选择器查询页面元素是该框架赖以起家的绝活。提及脚本选择器，最早可以追溯到 Dean Edwards 的 cssQuery 和 Simon Willison 的 getElementsBySelector，但是在早期的 Web 编程环境中，面对恶劣的浏览器环境和粗糙的 Web 应用，这些选择器的实现只能被当作内部实验，没有应用市场。而到了 2005 年，互联网已经进入 Web 2.0 和 Ajax 的新时代，1984 年出生的天才少年 John Resig 在先行者的启发下开始探索 JavaScript 选择器技术，并在一年后正式发布了 jQuery 1.0，迅速红遍全球，流行程度堪比摇滚巨星。

人都有一种惰性，或者说是习惯吧，用惯了 Java 作为开发语言，就不习惯.NET 的编程环境。同样，如果你习惯了 CSS 的用法及其选择网页标签的方式，再来学习 jQuery 的选择器你会倍感亲切，甚至于入迷。jQuery 完全仿制了 CSS 选择器的设计思路和用法，让广大的初学者和整日为获取网页节点而搔首的开发人员快速上手。

jQuery 灵巧、便捷，并能够按着人的思维去编写代码，这实在是一件很令人兴奋的事情。难怪但凡接触过 jQuery 的初学者都会对它爱不释手，也难怪 jQuery 在众多 JavaScript 流行框架中能快速杀出一条血路，并得到广大草根编程人员的喜爱。

很多读者和开发人员接触 jQuery 之后，都会身不由己地迷恋于 jQuery 的优雅和便捷，错误地认为它就是一种高效的 JavaScript 编程方法，于是在前端开发中毫无节制地使用 jQuery 匹配和操作网页元素。这种行为本无可厚非，但是我们不应忘记，jQuery 仅是 JavaScript 功能的外包装。因此在学习和使用 jQuery 的过程中，读者应该首先树立以下两种意识。

1. 执行效率是编写脚本的第一要务

jQuery 经过多个版本的磨炼，特别是新的独立开发的 Sizzle 选择器引擎，使 jQuery 匹配元素的速度达到了新的高度，可以毫不夸张地说 jQuery 已是业界第一。但是，无论 Sizzle 选

择器的速度如何优化，它都是在 JavaScript 原生方法的基础上进行打包的，这种打包过程虽然节省了用户编写代码的工作量，但是却增加了代码的执行速度。因此，在可能的条件下，建议读者不要完全抛弃 JavaScript 原生的选择器方法，适当混合使用 jQuery、DOM 和 JavaScript 能够提升程序的执行效率。例如，offset(获取页面上的各种尺寸和位置数据)、创建和插入 DOM 节点的方法(如 append, before)都是 JQuery 速度的瓶颈，在必要的情况下，读者完全不用它们，而直接使用 DOM 原生方法会更加高效。正是由于这个原因，本书在讲解 jQuery 选择器及其各种 DOM 元素的操作过程中，都会对比 JavaScript 设计相同效果或功能的实现方法，让读者能够在学习后辈技术的同时，不要忘记老一辈技术的实现途径。很多初学者由于习惯了 jQuery 开发方式，竟然不会使用 JavaScript 进行开发，这实在是件很不幸的事情。

2. 理解 jQuery 设计模式和工作机制是学习和提升 jQuery 开发水平的关键

jQuery 框架的源代码虽然仅有 4000 多行，但是它设计得精巧和复杂，非一般读者所能够读懂，但是即便如此，对于每一位初学者来说，我们都应该尽力理解框架中每一行代码的功能，以及各行代码之间的逻辑关系，这对于正确和高效地使用 jQuery 是至关重要的。同时当程序发生错误时，我们也能够通过浏览器简单的错误提示快速找到症结所在。正所谓知其然，也应知其所以然。本书在讲解 jQuery 框架的使用方法的同时，把更多的精力投入到 jQuery 框架的内部，揭秘 jQuery 框架的内部逻辑关系。相信读者在阅读本书的过程中，会对 jQuery 内核有更深入的认识。

关于本书

本书从前台开发人员的角度进行选材，主要研究 jQuery 框架的设计模式、实现机制和 jQuery 框架的一般用法、扩展应用及其实战演练。全书共分为 10 章，简单介绍如下。

第 1 章　jQuery 起步，重点介绍 JavaScript 及其框架发展概述，如何使用 jQuery，如何编写自己的第一个示例，以及 jQuery 框架的基本特性。

第 2 章　jQuery 解密技术，重点介绍 jQuery 框架的原型设计思路和模式，详细分解选择器接口和选择器引擎 Sizzle 的结构和工作机制，另外还详细讲解 jQuery 对象的数据结构和基本操作。

第 3 章　高效选择的技巧与原理，重点讲解 jQuery 的 CSS 选择器用法，以及这些选择器类型的实现方法。

第 4 章　文档对象的操作及其高效实践，重点讲解 jQuery 如何操作文档结构和元素，如何控制页面 CSS 的样式。同时，还就 JavaScript 实现方法进行详细讲解，以方便读者横向比较两者实现途径的异同，从而更加深刻地理解 jQuery 的操作机制。

第 5 章　事件封装机制与解析，重点讲解 jQuery 如何封装 JavaScript 的事件处理机制，并介绍如何使用 JavaScript 来定义 jQuery 的事件处理方法。

第 6 章　动画效果设计及其高效实践，重点讲解 jQuery 在设计动画方面的优势，以及如何使用 jQuery 设计常规动画方法，并就 JavaScript 实现相同动画效果进行讲解。

第 7 章　Ajax 异步通信高效实践，讲解 Ajax 异步通信的原理，并根据这个原理分解 jQuery 是如何实现这些异步通信的方法。

第 8 章　高效开发和使用插件，重点讲解如何扩展 jQuery 公共函数、jQuery 对象方法和 jQuery 选择器，并通过几个典型案例演示这些扩展方法。

第 9 章　jQuery 辅助工具，重点讲解 jQuery 公共函数工具，以及附加的缓存数据和数据队列处理，还介绍一些辅助工具的使用。

第 10 章　使用 jQuery 打造 Ajax 异步交互式动态网站，通过一个动态演示网站的架设帮助读者认识 jQuery 在网站开发中的角色。

读者对象

本书适合网页制作的初学者，广大网页制设计师和前端技术人员。

读者在阅读本书之前，应该初步掌握 HTML、CSS 和 JavaScript 语言的基础知识，特别是能够初步使用 JavaScript。

本书约定

(1) 考虑到版面限制，展示出来的代码仅包含 JavaScript 脚本和必要的结构代码。读者在学习测试时，应该把这些 JavaScript 脚本输入到网页。例如：

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("div").after("<p>最后一段文本</p>");
    $("div").before("<p>第一段文本</p>");
})
</script>

<div>
    <p>段落文本</p>
</div>
```

则应该按如下代码格式输入上面示例代码，并另存为 html 网页文件，然后在浏览器中测试即可。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title></title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("div").after("<p>最后一段文本</p>");
```

```
    $("div").before("<p>第一段文本</p>");
})
</script>
</head>
<body>
<div>
    <p>段落文本</p>
</div>
</body>
</html>
```

(2) 本书以 jQuery 1.3.2 版本为基础进行介绍和演示，能够兼容 jQuery 3.0 以后的任何版本。

(3) 在默认情况下，jQuery 1.3.2 库文件都会自动导入文档，如果没有特别说明，我们会在示例中省略以下命令。注：jquery-1.3.2.js 存放在 images 文件夹中。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
```

(4) 在默认情况下，我们都会使用 jQuery 的别名$来表示 jQuery 名字空间，同时直接把调用的函数放在$()函数中，该函数实际上是$("document").ready()方法的简写，它相当于 JavaScript 中的 window.onload =function () {}事件处理函数。例如：

```
<script type="text/javascript" >
$(function(){
    $("div").after("<p>最后一段文本</p>");
    $("div").before("<p>第一段文本</p>");
})
</script>
```

实际上上面的示例代码应该是下面的形式。

```
<script type="text/javascript" >
$("document").ready (function(){
    $("div").after("<p>最后一段文本</p>");
    $("div").before("<p>第一段文本</p>");
})
</script>
```

它相当于 JavaScript 中下面的示例代码，当然两者之间还存在细微的区别，详细讲解请参阅本书第 1、2 章中的讲解。

```
<script type="text/javascript" >
window.onload =function(){
    $("div").after("<p>最后一段文本</p>");
    $("div").before("<p>第一段文本</p>");
}
</script>
```

(5) 由于 jQuery 与 JavaScript 变量之间存在区别，默认情况下当定义 jQuery 对象变量时，应在该变量的前面附加一个$前缀，以便与 JavaScript 变量进行区分。例如：

```
<script type="text/javascript" >
var $div = $("div");  //jQuery 对象变量
```

```
var div = document.getElementsByTagName("div"); //JavaScript对象变量
</script>
```

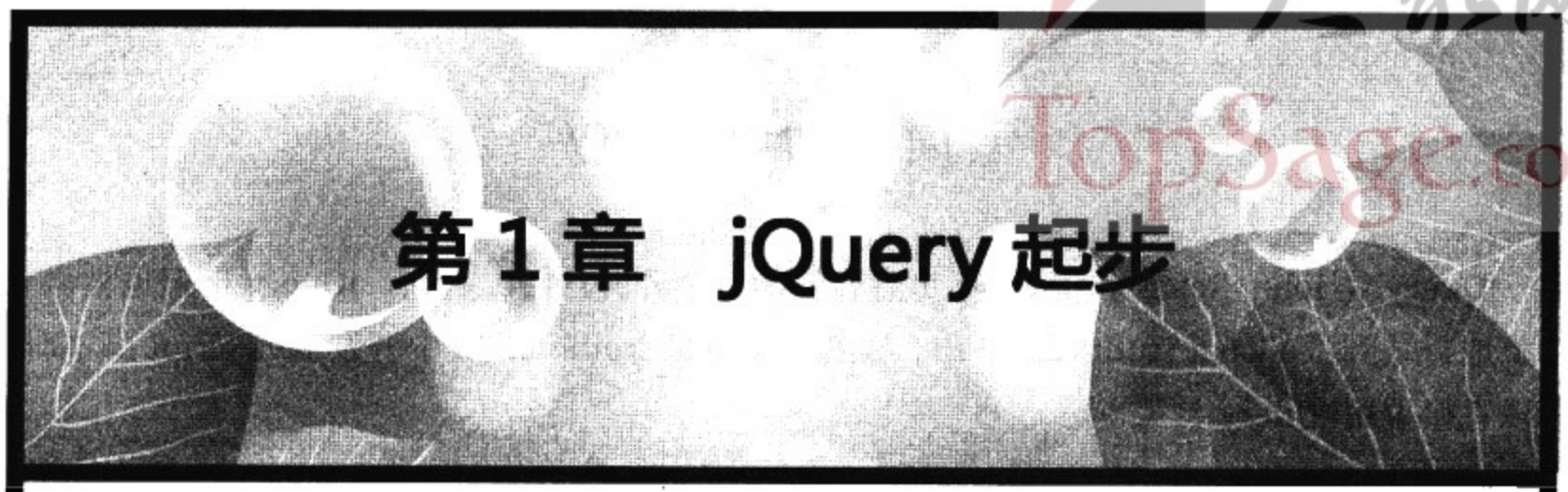

关于作者

本书由朱印宏编写，参与资料整理及编写的还有袁衍明、常才英、袁祚寿、袁衍明、张敏、袁江、田明学、唐荣华、毛荣辉、卢敬孝、刘玉凤、李坤伟、旷晓军、陈万林、陈锐、钱佩林、苏敬波、冉东林、杨龙贵、张炜、王慧明、涂怀清、卢国才、苏恢定、司成向、胡体清、陈宗亮、徐清银、周秀成、颜昌学、王幼平、冉原洲、李经键、胡厚成等，在此对大家的辛勤工作表示衷心的感谢。

由于作者水平有限，书中难免会有疏漏之处，恳请广大读者提出宝贵意见，也希望广大读者能够把个人的意见、建议或问题发送到 zhuyinhong@css8.cn，以便与作者进行交流。

编 者

目　录



第1章 jQuery起步

jQuery是继Prototype之后又一个优秀的JavaScript代码库(或JavaScript框架)。jQuery设计的宗旨是“Write Less，Do More”，即倡导写更少的代码，做更多的事情。

jQuery封装了JavaScript常用的功能代码，提供了一种简洁、快捷的JavaScript设计模式，优化了HTML文档操作、事件处理、动画设计和Ajax交互。可以说，jQuery改变了用户编写JavaScript代码的方式。由于jQuery能够兼容CSS 3，并且能兼容各种主流浏览器，如IE 6.0+、FF 1.5+、Safari 2.0+和Opera 9.0+等，因此它正被越来越多的开发人员喜爱和选用。

1.1 认识 jQuery

jQuery 是 John Resig 于 2006 年 1 月开发的一个开源项目。目前，jQuery 框架也被微软完整地封装到了 Visual Studio 2008 中，读者在 Visual Studio 2008 中开发 JavaScript 脚本时，可以直接调用 jQuery 框架。

jQuery 项目主要包括 jQuery Core(核心库)、jQuery UI(界面库)、Sizzle(CSS 选择器)和 QUnit.(测试套件)四部分，团队成员由核心库开发、UI 开发、插件开发人员，以及网站设计、运营和推广人员组成。团队核心人物主要是 John Resig、Brandon Aaron 和 Jorn Zaefferer。

- jQuery 框架官网：http://jquery.com。
- jQuery 项目组官网：http://jquery.org。
- John Resig 个人网站：http://ejohn.org。

John Resig 目前供职于 Mozilla 公司，专门负责开发 JavaScript 开发工具。这个小伙子出生于 1984 年 6 月 8 日，居住在美国波士顿，今年 26 岁，也就是说 John Resig 在 22 岁时就已经独立开发出了 jQuery 技术框架。先来欣赏一下这个白白的小胖子，如图 1.1 所示。

图 1.1 John Resig

1.1.1 JavaScript 及其库

JavaScript 最先是由 Netscape 公司开发的一种脚本语言，用来在网页中设计交互、动态的效果。随后，微软公司也推出了自己的 JavaScript 版本，并将其命名为 JScript。后来，ECMA 标准化组织推出了 ECMAScript 标准，并试图规范 JavaScript 语言的语法和用法。目前，我们所介绍、学习和使用的 JavaScript 版本都是以 ECMAScript 标准为基础的。

由于 JavaScript 版本之间的差异性，以及各大浏览器对于 JavaScript 和 DOM(Document Object Model，文档对象模型)解析的不统一，给开发人员带来了很多麻烦。正当 JavaScript 逐渐被开发人员弃用时，一种基于 JavaScript 语言的 Ajax 技术横空出世，重新唤起了人们对于 JavaScript 开发的热情。

为了简化 JavaScript 开发，一些 JavaScript 代码库诞生了。这些代码库封装了很多预定义

的对象和实用函数，能够简化开发人员的工作，提高代码的执行效率。JavaScript 代码库的诞生，标识了真正的 Web 2.0 应用开发的到来，为富客户端开发奠定了基础。

随着代码库的不断演化和升级，技术框架逐渐变得逻辑严密。所谓框架，就是指一套包含工具、函数库、约定，以及尝试从常用任务中抽象出可以复用的通用模块，其目标是使设计师和开发人员把重点放在任务项目所特有的方面，避免重复开发。通常地讲，框架就是最常用的 JavaScript 框架和 Web 应用框架，当然，还有 CSS 框架。下面分别介绍几个比较流行的 JavaScript 技术框架。

1. Dojo

Dojo(http://dojotoolkit.org)是一个强大的面向对象的 JavaScript 框架，主要由三大模块组成：Core、Dijit 和 DojoX。Core 提供 Ajax、events、packaging、CSS-based querying、animations 和 JSON 等相关操作 API。Dijit 是一个可更换皮肤，且基于模板的 Web UI 控件库。DojoX 包括一些创新/新颖的代码和控件：DateGrid、charts、离线应用和跨浏览器矢量绘图等，其官网如图 1.2 所示。

图 1.2　Dojo 官网

2. YUI

YUI(http://developer.yahoo.com/yui)是 Yahoo! User Interface (YUI) Librar 库的简称，它是一组采用 DOM Scripting、Dhtml 和 Ajax 等技术开发出的 Web UI 控件和工具，中文翻译就是 Yahoo 用户界面库，其官网如图 1.3 所示。

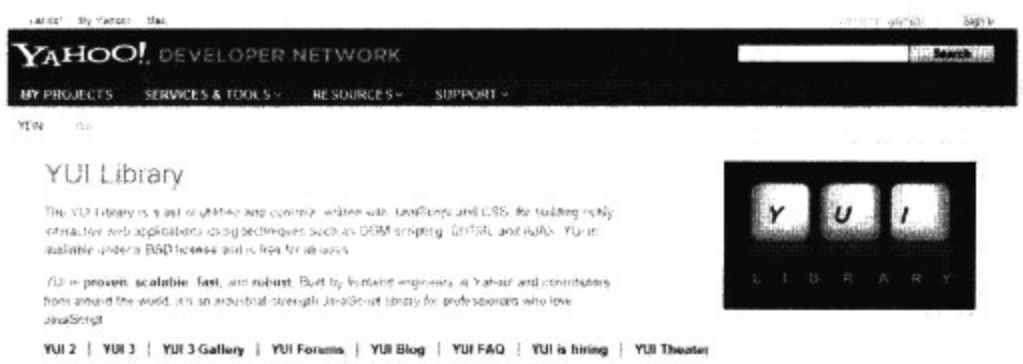


图 1.3　YUI 官网

- YUI 工具包利用 DOM 脚本来简化浏览器内的开发，使用 DHTML 和 AJAX 的特性开发所有的 Web 程序。
- YUI 控件库为页面提供一组具有高交互性的可视化元素。不需要请求服务器进行页面刷新。

3. jQuery

jQuery(http://jquery.com)是一个快速、简洁的 JavaScript 框架，可以简化查询 DOM 对象、处理事件、制作动画、处理 Ajax 交互过程。利用 jQuery 将改变 JavaScript 代码的编写方式。原先用 20 行代码才能完成的功能，jQuery 用 10 行就可以轻松搞定，其官网如图 1.4 所示。

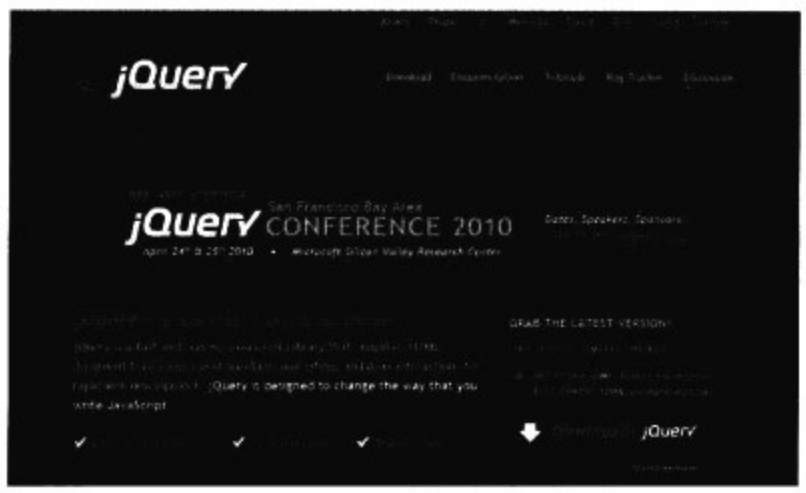


图 1.4 jQuery 官网

4. Mootools

Mootools(http://mootools.net)是一个简洁、模块化、面向对象的 JavaScript 框架。它能够更快、更简单地编写可扩展和兼容性强的 JavaScript 代码。Mootools 从 Prototype 中汲取了许多有益的设计理念，其语法与 Prototype 极其类似，但它提供的功能要比 Prototype 多，整体设计也比 Prototype 要相对完善，功能更强大，它增加了动画特效、拖放操作等功能。其官网如图 1.5 所示。

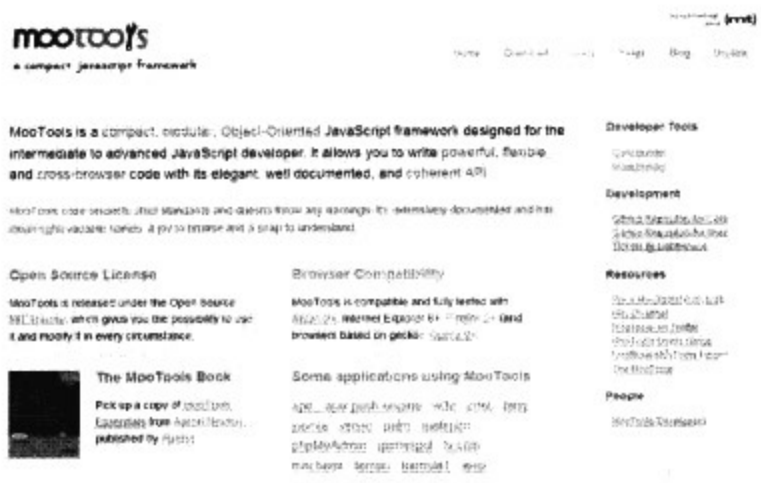


图 1.5 Mootools 官网

5. Prototype

Prototype(http://www.prototypejs.org)是一个易于使用、面向对象的 JavaScript 框架。它封装并简化和扩展一些在 Web 开发过程中常用到的 JavaScript 方法与 Ajax 交互处理过程，其官网如图 1.6 所示。

图 1.6　Prototype 官网

6. script.aculo.us

script.aculo.us(http://script.aculo.us)是一个易于使用，支持多种浏览器，且用于增强 Prototype 的 JavaScript 框架。script.aculo.us 包含动画框架、拖放、Ajax 控件、DOM 工具和单元测试等，其官网如图 1.7 所示。

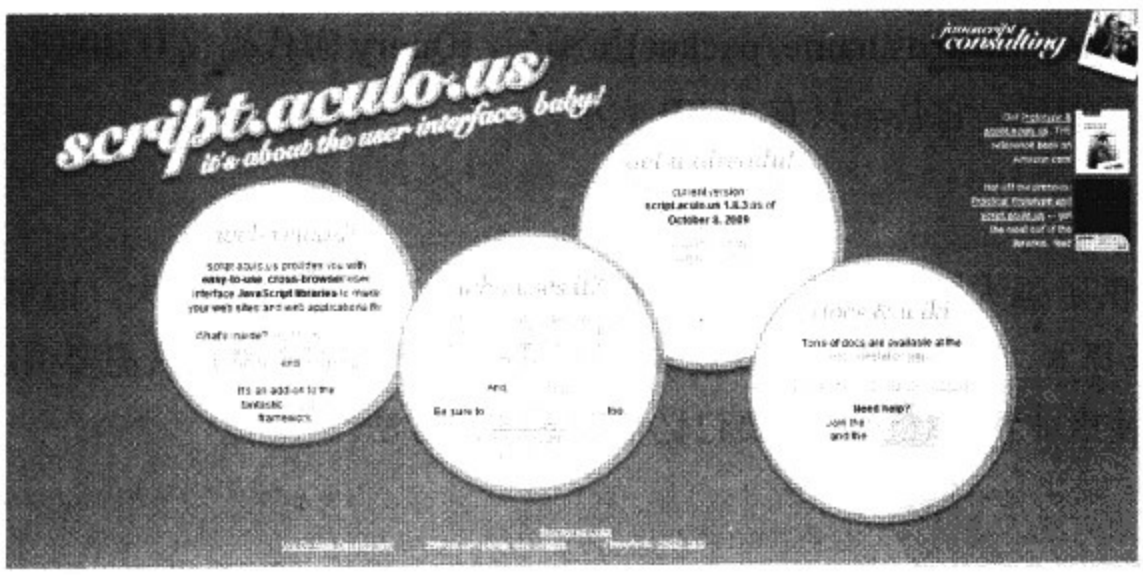


图 1.7　script.aculo.us 官网

7. ExtJS

ExtJS(http://www.extjs.com)是一个跨浏览器，用于开发 RIA(Rich Internet Application)应用的 JavaScript 框架，它提供高性能、可定制的 Web UI 控件库。该框架具有良好的设计、丰富的文档和可扩展的组件模型，其官网如图 1.8 所示。

图 1.8 ExtJS 官网

1.1.2 选用 jQuery 框架的理由

在众多流行的 JavaScript 框架中选用 jQuery，说明读者已经被 jQuery 的可爱和灵巧所吸引，但如果让你来描述选择 jQuery 的理由，你可能会无言表述，也可能会滔滔不绝。不过下面几点原因是必须要说的。当然，不同 JavaScript 技术框架都有其各自的优势和不足，同时每一种框架都有自己忠实的粉丝。在客观分析 jQuery 框架的优势时，也希望读者能够理性地汲取不同框架的精华，以促使你的 JavaScript 开发技术不断飞升。

1. 轻巧

轻巧是 jQuery 的先天优势，任何其他框架都无法与其相比。如果采用 Dean Edwards 的 Packer(http://dean.edwards.name/packer)压缩后，jQuery 源代码仅有 30KB 左右；如果服务器端启用 gzip 压缩后，它甚至只有 16KB 大小。

2. 方便

jQuery 提供了强大的 HTML 元素选择功能。Sizzle 引擎功能强大，能够支持 CSS 1 到 CSS 3 所有的选择器、XPath 选择器以及 jQuery 自定义选择器。用户只需要调用 jQuery()函数(或$())即可，使用非常方便，不需要记忆很多的访问方法。

3. 兼容

jQuery 能够在 IE 6.0+、FF 2+、Safari 2.0+和 Opera 9.0+下正常运行，解决了 JavaScript 在不同浏览器中的差异性。

4. 连写

jQuery 中最有特色的莫过于它的链式操作方式，即对发生在同一个 jQuery 对象上的一组动作，可直接连写而无需重复获取对象。这一点使 jQuery 的代码无比优雅。

5. 扩展

jQuery 提供了丰富的插件支持。jQuery 的易扩展性可以方便任何用户扩展 jQuery 的功能，因此也吸引了来自全球各地的开发者来共同编写 jQuery 的扩展插件。目前已经有超过几百种的官方插件支持。

6. 封装

jQuery 封装了 DOM 操作，并定义了大量的方法，使用户在编写 DOM 操作相关程序的时候能够得心应手，优雅地完成各种原本非常复杂的操作。jQuery 还封装了大量的事件处理函数，使得 jQuery 处理事件绑定的时候非常稳定。jQuery 将所有的 Ajax 操作封装到一个函数 $.ajax()里，使用户在处理 Ajax 的时候能够专心处理业务逻辑，而无需关心复杂的浏览器兼容性和 XMLHttpRequest 对象的创建和使用问题。

7. 封闭

jQuery 只建立一个名为 jQuery 的对象，其所有的方法都在这个对象之下。另外的一个别名$也是可以随时交出控制权的，不会污染其他的对象，也不会与其他 JavaScript 框架发生冲突。

8. 分离

jQuery 完全摆脱 JavaScript 的设计模式，可以允许用户在 jQuery 环境下自由开发程序，调用 jQuery()函数选择相关匹配元素，然后直接在 jQuery 对象上完成操作，而不用在 jQuery 和 JavaScript 两种设计模式中来回切换。

9. 完善

jQuery 框架能够后来者居上，主要得益于它的全新设计模式，并产生了滚雪球效应。网上到处充斥着 jQuery 相关的学习资料和开发应用案例，同时 jQuery 团队也提供了大量的帮助文档，这些都为 jQuery 框架的普及和推广奠定了基础。

1.2 jQuery 初步体验

jQuery 如此受欢迎，有很大一部分是因为它允许用户用最少的代码就能非常优雅地找出和操作 HTML 元素，且非常有效。下面我们就一起走入 jQuery 的世界，感受它的敏捷和强大。

1.2.1 安装 jQuery 库

学习和使用 jQuery 框架的第一步应该是安装 jQuery 代码库。

首先，进入 jQuery 官方网站(http://jquery.com)下载最新版本的 jQuery 源代码，如图 1.9 所示。目前最新版本是 jQuery 1.4.2。本书主要根据 jQuery 1.3.2 版本进行讲解，由于 jQuery 1.4.2 版本仅增加和完善了部分功能，所以读者使用 jQuery 1.4.2 版本也可以进行学习和上机测试。但是在讲解 jQuery 核心代码时，本书主要是以 1.3.2 版本为主的。

图 1.9　下载 jQuery 最新版本

在 jQuery 官网首页右侧，先选择要下载的 jQuery 框架类型，主要包括发布版 Production (24KB, Minified and Gzipped)和测试版 Development (155KB, Uncompressed Code)。

如果选择 Production(发布)选项，则可以下载代码压缩版本，此时 jQuery 框架源代码被压缩到了 24KB，下载的文件为 jquery-1.4.2.min.js；如果选择 Development(开发)选项，则可以下载包含注释的未被压缩的版本，大小为 155KB，下载的文件为 jquery-1.4.2.js。然后，单击 Download(jQuery)按钮即可。

如果要下载 jQuery 其他版本的源代码，可以访问下列网址进行下载。

- http://github.com/jquery/jquery
- http://code.google.com/p/jqueryjs
- http://docs.jquery.com/Downloading_jQuery

1.2.2　导入 jQuery 库

jQuery 框架不需要安装，读者只需在网页文档的头部导入 jQuery 框架源代码的文件即可。导入文件可以使用相对路径，也可以使用绝对路径，具体情况可根据读者存放 jQuery 库文件的位置而定。

导入 jQuery 库文件的代码如下所示(加粗代码)。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
```

```
TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
</head>
<body>
</body>
</html>
```

1.2.3 编写 jQuery 代码

当引入 jQuery 库文件之后，我们就可以进行 jQuery 开发了。例如，先让 jQuery 打个招呼，其代码如下。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    alert("Hello World!");
})
</script>
</head>
<body>
</body>
</html>
```

这样在浏览器中预览该示例文档时，可以看到在当前窗口中会弹出一个提示对话框，如图 1.10 所示。

图 1.10 第一个示例演示效果

首先，读者应该明确一个概念。在 jQuery 库中，$表示 jQuery 的别名。例如，$()等效于 jQuery()。在本书程序中，如果没有特别说明，$就表示 jQuery。

在做所有事情之前，如果要让 jQuery 读写和处理文档的 DOM，则必须在 DOM 载入之后再开始执行事件，所以应该使用 ready 事件作为处理 HTML 文档的开始。

```
$(document).ready(function() {
    //jQuery 代码
});
```

上面代码的语义是：匹配文档中的 document 节点，然后为该节点绑定 ready 事件处理函数。它类似于 JavaScript 的 window.onload 事件处理函数(如下所示)，不过 jQuery 的 ready 事件要先于 onload 事件被激活。

```
window.onload = function(){
    //JavaScript 代码
};
```

为了方便开发，jQuery 框架进一步简化了$(document).ready()方法的写法，直接使用$()方法来表示，其代码如下。

```
$( function() {
    //jQuery 代码
});
```

一般情况下，所有 jQuery 代码建议都包含在$()函数中，当然也可以不包含在$()函数中，这与 JavaScript 代码应该放在 window.onload 事件处理函数中的道理是一样的。

1.2.4 区分 jQuery 对象和 DOM 对象

第一次学习 jQuery 的读者很容易迷惑的就是什么是 jQuery 代码，什么是 JavaScript 代码。

实际上，jQuery 框架本身就是在 JavaScript 语言基础上进行包装的，因此 jQuery 代码本质上也是 JavaScript 代码，自然 jQuery 代码与 JavaScript 代码可以相互混合使用。读者也没有必要去区分每一行代码到底是 jQuery 代码，还是 JavaScript 代码。

但是，初学者必须明白 jQuery 对象和 DOM 对象。读者只要能够分辨清楚 jQuery 对象和 DOM 对象，以及它们各自适用的方法即可。

DOM 是 Document Object Model 的简写，中文翻译为文档对象模型。根据 W3C DOM 规范，DOM 是 HTML 与 XML 的应用编程接口(API)，DOM 将整个页面映射为一个由层次节点组成的文件。例如，将下面文档加载并用浏览器解析后，会自动生成一个映射文件，该文件以结构树的形式展示了当前文档的结构，如图 1.11 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>标准 DOM 示例</title>
    </head>
    <body>
        <h1>标准 DOM</h1>
```

```
        <p>这是一份简单的<strong>文档对象模型</strong></p>
        <ul>
            <li>D 表示文档，DOM 的物质基础</li>
            <li>O 表示对象，DOM 的思想基础</li>
            <li>M 表示模型，DOM 的方法基础</li>
        </ul>
    </body>
</html>
```

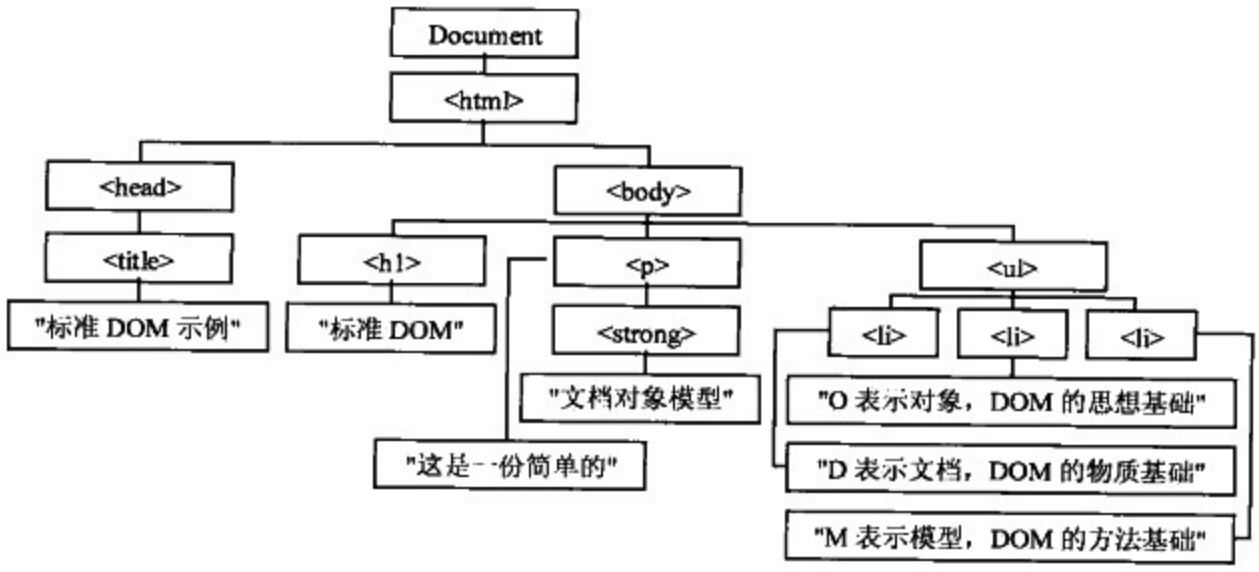


图 1.11　文档对象模型的树形结构

在这棵 DOM 树中，<li>、<li>、<li>标签都是<ul>标签的子节点，可以通过 document 对象的 getElementsByTagName()或者 getElementById()方法访问它们，也可以利用节点之间的关系，通过它们的关联指针快速进行相互访问。

上面所有的节点和元素都是 DOM 对象的组成部分，getElementsByTagName()和 getElementById()方法也是 DOM 模型提供的内置方法。所有这些就构成了 DOM 对象的基础。

jQuery 对象是通过 jQuery 框架包装 DOM 对象之后产生的一个新对象，从本质上分析它仅是 DOM 对象的集合，因此我们可以把 DOM 对象看作是一个独立的个体，而 jQuery 是多个 DOM 对象组成的数据集合。

jQuery 框架为 jQuery 对象定义了独立使用的方法和属性，它无法直接调用 DOM 对象的方法。相反，DOM 对象也无法直接调用 jQuery 对象的方法或属性。例如，下面的写法都是错误的。

```
$("#wrap").innerHTML ="嵌入文本";                    //jQuery 对象调用 DOM 属性
getElementById("wrap").html("嵌入文本") ;            //DOM 对象调用 jQuery 对象的方法
```

1.2.5　jQuery 对象和 DOM 对象的相互转换

jQuery 对象和 DOM 对象是可以相互转换的，因为它们所操作的对象都是 DOM 元素，只不过 jQuery 对象包含了多个 DOM 元素，而 DOM 对象本身就是一个 DOM 元素。简单地说，

jQuery 对象是 DOM 元素的数组，也称为类数组，而 DOM 对象就是单个的 DOM 元素。

1. 把 jQuery 对象转换为 DOM 对象

jQuery 对象不能够直接使用 DOM 对象的方法，但是如果用户需要时，就应该先把 jQuery 对象转换为 DOM 对象。转换的方法有以下两种。

第一，借助数组下标来读取 jQuery 对象集合中的某个 DOM 元素对象。例如，在下面的示例中，先使用 jQuery 匹配文档中所有的 li 元素，返回一个 jQuery 对象，然后通过数组下标的方式读取 jQuery 集合中第一个 DOM 元素，此时再返回的是 DOM 对象。这时就可以调用 DOM 属性 innerHTML 了。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var $li = $("li");      //返回 jQuery 对象
    var li =$li[0];         //返回 DOM 对象
    alert(li.innerHTML);
})
</script>

<ul>
    <li>列表 1</li>
    <li>列表 2</li>
    <li>列表 3</li>
</ul>
```

第二，借助 jQuery 对象的 get()方法。为 get()方法传递一个下标值，即可从 jQuery 对象中取出一个 DOM 对象元素。例如，上面示例可以改写为下面的方法。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var $li = $("li");      //返回 jQuery 对象
    var li =$li.get(0);     //返回 DOM 对象
    alert(li.innerHTML);
})
</script>
```

2. 把 DOM 对象转换为 jQuery 对象

对于 DOM 对象来说，直接把它传递给$()函数即可，jQuery 对象会自动把它包装为 jQuery 对象，然后它就可以自由调用 jQuery 定义的方法。例如，针对上面的示例，我们可以这样来设计：首先使用 DOM 的方法获取所有 li 元素。其代码如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
```

```
    var li = document.getElementsByTagName("li");    //获取所有 li 元素
    var $li = $(li[0]);    //把第一个 li 元素封装为 jQuery 对象
    alert($li.html());    //调用 jQuery 对象的方法
})
</script>

<ul>
    <li>列表 1</li>
    <li>列表 2</li>
    <li>列表 3</li>
</ul>
```

实际上，也可以把 DOM 元素数组传递给$()函数，jQuery 对象会自动把所有 DOM 元素包装在一个 jQuery 对象中。例如，上面示例还可以设计成下面的代码。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var li = document.getElementsByTagName("li");    //获取所有 li 元素
    var $li = $(li);        //把所有 li 元素封装为 jQuery 对象
    alert($li.html());      //调用 jQuery 对象的方法
})
</script>
```


注意：使用 document.getElementsByTagName()方法获取的 DOM 元素集合是一个真正的数组类型数据，而 jQuery 对象仅是一个类数组，不是真正意义上的数组类型数据。关于类数组和数组之间的区分，我们将在第 2 章中进行详细讲解。

1.2.6 ready 事件和 load 事件比较

在前面小节中曾经介绍过 jQuery 定义的 ready 事件和 JavaScript 默认的 load 事件。下面我们来比较这两个事件的区别。为了理解这两个事件的异同，读者应该先了解 HTML 文档加载的顺序。

DOM 文档加载是按顺序执行的，这与浏览器的渲染方式有关系。一般浏览器渲染操作的顺序大致按如下几个步骤来完成。

(1) 解析 HTML 结构。

(2) 加载外部脚本和样式表文件。

(3) 解析并执行脚本代码。

(4) 构造 HTML DOM 模型。

(5) 加载图片等外部文件。

(6) 页面加载完毕。

具体说明如下。

1. 执行时机

load 事件必须等到网页中所有内容全部加载完毕之后才被执行。如果一个页面中包含了大容量的多媒体文件，则就会出现这种情况：网页文档已经呈现出来，但由于网页数据还没有完全加载完毕，导致 load 事件不能够即时被触发。

开发人员习惯把页面初始化设置的脚本都放在 load 事件处理函数中，由于页面数据没有完全加载进来，导致网页文档呈现和脚本初始化配置不能够保持同步，从而影响了页面的可用性。

而 jQuery 的 ready 事件是在 DOM 结构绘制完毕之后就执行，也就是说它在外部文件加载之前就被执行了，这样就能够确保网页文档的呈现和脚本初始化设置保持同步。

总之，ready 事件先于 load 事件被激活，如果网页文档中没有加载外部文件，则它们的响应时间基本上是相同的。

2. 编写个数

JavaScript 的 load 事件只能够被编写一次，下面的写法是不允许的，此时它仅能够影响最后一次指定的事件处理函数。

```
<script type="text/javascript" >
window.onload = function(){
    alert("页面初始化 1");
}
window.onload = function(){
    alert("页面初始化 2");
}
window.onload = function(){
    alert("页面初始化 3");
}
</script>
```

如果希望这三个事件处理函数中的代码都被执行，则应该把它们包含在一个处理函数中，其代码如下。

```
<script type="text/javascript" >
var f1 = function(){
    alert("页面初始化 1");
}
var f2 = function(){
    alert("页面初始化 2");
}
var f3 = function(){
    alert("页面初始化 3");
}
```

```
window.onload = function(){
    f1();
    f2();
    f3();
}
</script>
```

而对于 jQuery 的 ready 事件类型来说，我们可以在同一个文档中多次定义。例如，针对上面示例，可以使用 jQuery 的 ready 事件类型来设计。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    alert("页面初始化 1");
});
$(function(){
    alert("页面初始化 2");
});
$(function(){
    alert("页面初始化 3");
});
</script>
```

这对于复杂页面中多次配置初始化程序非常重要，也方便了用户根据需要随时进行设计。

1.3 jQuery 核心特性

jQuery 框架的核心就是从 HTML 文档中匹配元素并对其执行操作。如果读者熟悉 CSS 技术，就会对 CSS 选择器的强大威力有所感触。CSS 选择器能够通过元素的特性或者文档位置来描述元素的样式，JavaScript 还无法内置 CSS 选择器的功能，它仅能提供两个有限的选择 DOM 元素的方法。现在 jQuery 通过封装 JavaScript 的原生方法，模拟了一套 CSS 选择器，甚至定义了完整的 XPath 语言的选择能力。这在一定程度上简化了 JavaScript 的操作。

jQuery 还为用户解决了跨浏览器的兼容问题，使 DOM 操作趋于统一。jQuery 把确保代码能够跨越所有的主要浏览器并以一致的方式进行工作摆在了高优先级的位置，许多让开发人员头疼的事件处理、样式设计等兼容操作问题变得轻松、方便许多。简单地说，jQuery 框架具有下面几个优点。

- 体积小，使用灵巧。
- 丰富的 DOM 选择器(CSS1~3、XPath)。
- 跨浏览器(IE6、FF、Safari、Opera)。
- 链式代码。
- 强大的事件、样式支持。

- 强大的 AJAX 功能。
- 易于扩展，插件丰富。

下面我们就 jQuery 框架的几个核心特性进行说明。

1.3.1 jQuery 构造函数

jQuery 把所有的操作都包装在一个 jQuery()函数中，形成了统一(也是惟一)的操作入口，这为 jQuery 操作降低了门槛。jQuery ()构造函数能够接收任意类型的数据，但是能够解析的参数包括下面 4 种类型。

1. jQuery(expression,context)

参数为一个表达式，该表达式可以是 ID、DOM 元素名、CSS 表达式和 XPath 表达式等，jQuery 将根据表达式匹配文档中的元素，然后把找到的元素包装到一个 jQuery 对象中返回。例如：

```
jQuery("div#wrap>p:first").addClass("red");
```

在表达式字符串中，div#wrap 表示 id 为 wrap 的 div 元素，先在该元素中匹配子元素 p，再筛选出第一个 p 元素。"div#wrap>p:first"是 CSS 表达式，如果使用 XPath 表达式表示，则应该为"div#wrap/p:first"；:first 是一个伪类，表示其中的第一个；addClass()为 jQuery 对象，用来添加 CSS 样式类的方法，相反操作的方法为 removeClass()。

2. jQuery(html)

参数表示一个 HTML 结构字符串，此时 jQuery 将创建一个对应结构的 HTML 文档片段。例如：

```
$('ul').append($('<li>new item</li>'));
```

其中$('
new item
')将其中的字符串转换为 DOM 对象，然后通过 append()方法加入到 ul 元素的最后。

3. jQuery(elements)

参数是一个 DOM 元素对象或者集合，此时 jQuery 将把 DOM 元素或集合中的 DOM 元素封装为 jQuery 对象。例如：

```
$(document).ready(function(){
    $('ul').css('color','red');
});
```

在上面示例中，jQuery 构造函数把 document 对象封装为一个 jQuery 对象，然后调用

ready()方法。ready 事件处理函数为 document 对象绑定一个事件，当页面初始化之后将 ul 的颜色设置为红色。

4. jQuery(fn)

参数是一个处理函数。由于$(document).ready()频繁使用，所以 jQuery 又使用$(fn)来代替，fn 表示处理函数。ready 处理函数在文档内容完全载入之后执行，因此不需要等待外部链接文件载入完成，响应要比 load 事件早。例如，下面代码就是 jQuery(fn)运用。

```
$(function(){
    $('ul').css('color','red');
});
```

1.3.2 jQuery 链式语法

jQuery 的代码是非常优雅的，也是非常灵巧的。它允许用户连续设计各种行为，从而实现按人的惯性思维进行快速开发。

例如，在下面这个示例中，仅写了两行脚本，就实现了复杂的页面交互效果，如图 1.12 所示。第一行代码使用 jQuery 构造器函数($())构建 4 个按钮，并把它们附加到文档中。第二行代码是一个连续的行为动作，分别选中这四个按钮并为它们绑定不同的事件处理函数。

```
<style type="text/css">
.panel {
    padding: 60px;
    background-color: red;
    color: #FFFFFF;
    font-size:50px;
    font-weight: bold;
    width: 600px;
    text-align:center;
}
</style>
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    //第一行代码，在文档中添加四个按钮
    $('<input type="button" value="click me"/><input type="button" value="triggle click
me"/><input     type="button"     value="detach     handler"/><input     type="button"
value="show/hide text"/>').appendTo($('body'));
    //第二行代码，分别选中四个按钮，并为它们绑定不同的事件处理函数
    $('input[type="button"]')
    .eq(0).click(function(){    //找到第一个按钮，并绑定 click 事件处理函数
        alert('you clicked me!');
    }).end().eq(1)    //返回所有按钮，再找到第二个
    .click(function(){    //为第二个按钮绑定 click 事件处理函数
        $('input[type="button"]:eq(0)').trigger('click');
```

```
    }).end().eq(2)      //返回所有按钮，再找到第三个
    .click(function(){     //为第三个按钮绑定 click 事件处理函数
       $('input[type="button"]:eq(0)').unbind('click');
    }).end().eq(3)      //返回所有按钮，再找到第四个
    .toggle(function(){     //为第四个按钮绑定 toggle 事件处理函数
       $('.panel').hide('slow');
    }, function(){
       $('.panel').show('slow');
    });
});
</script>

<div class="panel">welcome to jQuery!</div>
```

图 1.12　jQuery 链式语法示例效果图

在上面示例中，通过 end()方法取消当前的 jQuery 对象，返回前面的 jQuery 对象。这样当匹配某个按钮时，为其绑定事件处理函数，然后调用 end()方法，则又返回前面一个 jQuery 对象，即按钮集合。

链式代码已经成为 jQuery 非常流行的一个特点，在使用链条方式编写代码时，可能会用到 eq()、filter()的 jQuery 方法改变链式方法的对象，但是借助 jQuery 的 end() 方法又能够恢复或最初的 jQuery 对象，从而可以实现继续执行链式操作。注意，有几个 jQuery 的方法并不返回 jQuery 对象，所以链式操作就不能继续下去，如 get() 就不能像 eq() 那样使用。

链式语法是一种比较时尚的编程方法，但是在使用该种方法时，为了方便阅读，读者应该注意以下几个问题。

第一，如果在同一个 jQuery 对象上执行不超过 3 个方法，则可以在同一行内书写。例如，下面一行代码选择第一个按钮，修改它的名称，并为其附加一个类。

```
$('input[type="button"]').eq(0).val("修改按钮名称").addClass("red");
```

第二，如果在同一个 jQuery 对象上执行很多操作，则应该分行进行书写，这样方便阅读和修改。

第三，对于多个对象执行少量的操作，则可以为每一个对象书写一行代码。如果涉及子元素操作，可以考虑使用缩进方法进行设计，这样就能够区分层次。例如，针对上面示例，我们可以这样进行缩进显示。

```
$('input[type="button"]')
    .eq(0).click(function(){
        alert('you clicked me!');
    })
```

第四，如果对于多个对象执行很多连续的操作，则可以考虑结合上面几种方法进行设计。

1.3.3 jQuery 选择器

jQuery 之所以令人爱不释手，原因就在于其强大的选择器表达式令 DOM 操作优雅而艺术。jQuery 选择器支持 ID、tagName、CSS1~3 表达式和 XPath 表达式，详细说明请参阅 http://docs.jquery.com/Selectors 网站上的内容。jQuery 不仅模仿 CSS 和 XPath 选择器的用法和功能，还自定义了很多过滤方法，综合利用这些选择器，可以随心所欲地选择 HTML 结构中的任意元素。

所谓选择器(Selector)通俗地说就是一个表示特殊语义的字符串。只要把选择器字符串传递给 jQuery()构造函数，就能够选择不同的 DOM 对象，并且可以返回 jQuery 对象。jQuery 选择器支持 CSS3 选择器标准，读者可以在 W3C 官方网站(http://www.w3.org/TR/css3-selectors)上了解 CSS3 选择器的标准，本书将在第 3 章对其进行详细讲解。

jQuery 选择器按照功能主要分为选择和过滤，并允许配合使用，可以同时使用组合成一个选择器字符串。两者主要的区别如下。

(1) 过滤作用的选择器是利用指定条件从前面匹配的内容中筛选。过滤选择器也可以单独使用，此时表示从全部"*"元素中筛选。如：

```
$(":[title]")
```

等同于：

```
$("*:[title]")
```

(2) 选择功能的选择器则不会有默认的范围，因为它的作用是选择而不是过滤。

例如，在下面的示例中，$("input[type='button']")选择器可以匹配文档中 type 属性值为 button 的 input 元素，这个表达式是 CSS 表达式，然后为 button 添加 click 事件处理函数。示例演示效果如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
```

```
$(function(){
    $("input[type='button']").click(function(){
        var i = 0;
        $("input[type='text']").each(function(){
            i += parseInt($(this).val());
        });
        $('label').text(i);
    });
    $('input:lt(2)')
        .add('label')
        .css('border','none')
        .css('borderBottom','solid 1px navy')
        .css({'width':'30px'});
});
</script>

<input value="1" /> +
<input value="2" />
<input type="button" value="=" />
<label> </label>
```

在 click 事件处理函数中，$("input[type='text']")选择器能够匹配文档中所有输入框，然后调用 each()方法遍历所有匹配的文本框，利用$(this)选择器获取当前文本框，使用 val()读取当前文本框的值，再使用 JavaScript 的函数 parseInt()把获取的字符串类型的值转换为数值类型，相加之后作为文本信息添加到 label 元素中显示出来。

$('input:lt(2)')选择器能够匹配文档中的所有 input 元素，然后筛选出排在前面的两个 input 元素，其中的伪类:lt 表示序号小于某个值的意思。匹配到 input 元素之后，再添加 label 对象，合并成一个 jQuery 对象。然后通过链式语法连续调用三个 css()方法为文本框设置样式。如果一次需要设置多个 CSS 样式，也可以使用下面的方法来进行设计。

```
.css({
    'border':'none',
    'borderBottom':'solid 1px navy',
    'width':'30px'
});
```

如果只给 css()方法传递一个字符串参数，则表示读取样式值，如 css('color')就表示取得当前 jQuery 对象的样式属性 color 的值。而如果给它传递两个参数，则表示设置样式值。jQuery 对象定义了很多类似的方法，如 val()、text()、html()、click()、change()等。

1.3.4 jQuery 扩展性

个人认为，除了 jQuery 的选择器外，扩展性才是它受到用户热捧的主要原因。虽然说 jQuery 定义了庞杂的公共函数和 jQuery 方法，但没有哪个 JavaScript 框架能够预料所有人的

需求。甚至可以这样说，没有哪个框架应该设法将每个人需要的东西都预先准备好。这样做可能导致一堆大而笨重的、包含很少用到的功能的代码，这只会把事情搞砸。

jQuery 意识到了这个问题，并努力实现了扩展性，在核心库里仅仅定义了基础的方法和函数，但是特意留出了使得 jQuery 易于扩展的方法和接口。由于这个话题比较深奥，故我们把它放在最后进行讲解。本节就不再展开演示和说明。

第2章 jQuery技术解密

jQuery是一款非常优秀的JavaScript框架，与Prototype、YUI、Mootools等众多JavaScript框架齐名。不过jQuery框架为开发人员提供了一种精巧、便捷和高效的方法和编程思想，使其后来居上，不断成为广大JavaScript初学者首选的脚本库。本章将使用通俗易懂的语言，轻松流畅的思路来分析jQuery技术的核心及其设计原理和模型。

2.1 jQuery 框架设计概述

jQuery 1.3.2 版本的源代码共计有 4000 多行，虽然代码不多，但是由于它设计精巧，内部的结构关系和逻辑关系还是非常复杂，对于普通的开发人员来说，读懂 jQuery 源代码，并能够理清它的设计思路是件很困难的事情。如果开发人员仅仅知道如何使用 jQuery，却不理解 jQuery 的运行原理和内部机制，那么要想实现高级开发，将会遇到很多意想不到的系统级问题。

2.1.1 设计目标

jQuery 这个词可以分解为 JavaScript+Query，直接理解就是 JavaScript 查询的意思。jQuery 的核心目标是什么？

正如其名称一样，jQuery 的核心功能就是 JavaScript 查询，再通俗说就是选择 DOM 元素，并对选择的 DOM 元素进行操作，就这么简单。jQuery 框架仅仅实现两个动作：选择和操作。这也许就是它的理想，当然也是最实用的期望，为此，设计 jQuery 和理解 jQuery 都必须先回答下面 4 个问题。

- 选择什么？
- 如何选择？
- 怎么操作？
- 操作什么？

对于选择 DOM 元素，JavaScript 已经提供了以下两个方法。

```
document.getElementById();
document.getElementsByTagName();
```

但是它们的功能是有限的，同时它们的返回值也不同。getElementById()方法返回单个 DOM 元素，而 getElementsByTagName()方法则返回 DOM 元素集合。显然这两种方法还无法满足我们的开发需要。我们可能需要选择特定的 DOM 元素、指定范围的元素，或者特定属性的元素，所有这些需求都无法使用 JavaScript 提供的这两个方法来实现。

选择 DOM 元素最好的榜样应该算是 CSS 了，CSS 选择器提供了强大的选择文档元素的方式和规则。特别是 CSS3 版本中选择器更是非常强大。于是，jQuery 框架首先向 CSS 选择器学习。另一个学习的榜样就是 XPath。

XPath 是一门在 XML 文档中查找信息的语言。XPath 可用来在 XML 文档中对元素和属性进行遍历。XPath 是 W3C XSLT 标准的主要组成部分，并且 XQuery 和 XPointer 同时被构建

于 XPath 表达之上。因此，对 XPath 的理解是很多高级 XML 应用的基础。jQuery 把 XPath 语言封装进 JavaScript，当然是非常棒的选择。

另外，jQuery 也根据实际需要定义了很多选择器函数和选择器方法，这些共同组成了 jQuery 选择器，使其功力超强。

2.1.2 目标实现

选择 DOM 元素，仅是工作的第一步，因为这些元素集合不一定满足我们的要求，那么就要筛选，要过滤，要进行数组方面的相关操作。这就涉及类数组的处理。当最终选择出我们需要的 DOM 元素集合之后，最终的目标还是要操作这些元素。这些操作主要包括以下几个。

1. 属性操作

如 attribute、class 和 style 等都可以看作是属性操作，jQuery 也扩展了一些属性，如缓存数据、expando 的自定义属性。

2. 元素操作

如元素的创建、添加、补加、移动、复制和删除等。对于添加操作又可以分为追加、插入、前插、后插、内部前插和内部后插等。

3. 内容操作

在 JavaScript 中仅提供了 innerHTML 属性操作的 HTML 内容，也可以使用 childNodes 操作元素包含的子节点内容，或者使用 value 获取元素的值等。但是，jQuery 把这些孤零的方法和属性封装为不同的方法，并形成方便内容操作方法系列。

4. 样式操作

CSS 是 DOM 中重要的模块，它包括元素的 height、width、innerHeight、innerWidth、position、offset 和 display 等，还包括由这些基本样式演变出来的各种动画方法等。

动画时 CSS 的高级形态，无论什么动画效果，它都是基于时间的长短映射到元素 CSS 的变化属性值。一般都是采用 setInterval()方法间隔设置 CSS 样式，从而就形成了动画。

5. 事件操作

Event 也是 DOM 中的重要模块，它包括 addEvent、removeEvent 和 domready 等。

6. 通信操作

Ajax 是后来兴起的一种技术，也就是实现客户端和服务器端进行异步通信，借助 Ajax 可以为 DOM 文档动态添加内容，这些内容都通过简单的插件来实现。jQuery 提供了一种便捷的访问方式和使用途径。

如果用几个简单的字来概括 jQuery 的核心技术，那么就是：选择、操作和扩展。

选择和操作是 jQuery 框架的基础，而扩展却是 jQuery 框架不朽的灵魂。当然，对于一个优秀的框架来说，其所要考虑的问题并非这么单一，它还要考虑执行效率问题，考虑内存损耗问题，考虑可用性问题，考虑兼容性问题，考虑实用性问题等。所有这些问题，对于一个 JavaScript 框架开发者来说，都必须要面对。

不过，对于我们这些初学者以及试图理解 jQuery 框架的高级开发人员来说，理解上面所涉及的技术话题，已经足够了。

2.2 jQuery 原型技术分解

任何复杂的技术都是从最简单的问题开始的，如果你被 jQuery 几千行庞杂结构的源代码所困惑，那么建议你阅读本节内容，我们将探索 jQuery 是如何从最简单的问题开始，并逐步实现羽翼渐丰的演变过程，从 jQuery 核心技术的还原过程来理解 jQuery 框架的搭建原理。

2.2.1 起源——原型继承

用过 JavaScript 的读者都会明白，在 JavaScript 脚本中到处都是函数，函数可以归置代码段，把相对独立的功能封装在一个函数包中。函数也可以实现类，这个类是面向对象编程中最基本的概念，也是最高抽象，定义一个类就相当于制作了一个模型，然后借助这个模型复制无数的实例。

例如，下面的代码就可以定义最初的 jQuery 类，类名就是 jQuery，你可以把它视为一个函数，函数名是 jQuery。当然，你也可以把它视为一个对象，对象名是 jQuery。与其他面向对象的编程语言相比，JavaScript 对于这些概念的界定好像很随意，这降低了编程的门槛，反之也降低了 JavaScript 作为编程语言的层次。

```
<script language="javascript" type="text/javascript">
var jQuery = function(){
    //函数体
}
</script>
```

上面创建了一个空的函数，好像什么都不能够做，这个函数实际上就是所谓的构造函数。构造函数在面向对象语言中是类的一个特殊方法，用来创建类。在 JavaScript 中，你可以把任何函数都视为构造函数，这没有什么不可以的，这样不会伤害代码本身。

所有类都有最基本的功能，如继承、派生和重写等。JavaScript 很奇特，它通过为所有函数绑定一个 prototype 属性，由这个属性指向一个原型对象，原型对象中可以定义类的继承

属性和方法等。所以，对于上面的空类，可以继续扩展原型，其代码如下。

```
<script language="javascript" type="text/javascript">
var jQuery = function(){}
jQuery.prototype = {
   //扩展的原型对象
}
</script>
```

原型对象是 JavaScript 实现继承的基本机制。如果你觉得 jQuery.prototype 名称太长，没有关系，我们可以为其重新命名，如 fn，当然你可以随便命名。如果直接命名 fn，则表示该名称属于 Window 对象，即全局变量名。更安全的方法是为 jQuery 类定义一个公共属性 jQuery.fn，然后把 jQuery 的原型对象传递给这个公共属性，实现代码如下。

```
<script language="javascript" type="text/javascript">
jQuery.fn = jQuery.prototype = {
   //扩展的原型对象
}
</script>
```

这里的 jQuery.fn 相当于 jQuery.prototype 的别名，方便以后使用，它们指向同一个引用。因此若要调用 jQuery 的原型方法，直接使用 jQuery.fn 公共属性即可，不需要直接引用 jQuery.prototype，当然直接使用 jQuery.prototype 也是可以的。

既然原型对象可以使用别名，jQuery 类也可以起个别名，我们可以使用$符号来引用它，代码如下。

```
var $ = jQuery = function(){}
```

现在模仿 jQuery 框架源码，给它添加两个成员，一个是原型属性 jquery，一个是原型方法 size()，其代码如下。

```
<script language="javascript" type="text/javascript">
var $ = jQuery = function(){}
jQuery.fn = jQuery.prototype = {
   jquery: "1.3.2",          //原型属性
   size: function() {        //原型方法
      return this.length;
   }
}
</script>
```

2.2.2 生命——返回实例

2.2.1 节所列举的代码，相信很多读者都见过，甚至有可能也写过类似的代码，但是只有 John Resig 一个人把它发展成了 jQuery 这样不朽的框架。

当我们为 jQuery 添加了两个原型成员：jquery 属性和 size()方法之后，这个框架最基本的样子就孕育出来了。但是，该如何调用 jquery 属性和 size()方法呢？

也许，你可以采用如下方法调用：

```
<script language="javascript" type="text/javascript">
var my$ = new $();     //实例化
alert( my$.jquery );  //调用属性，返回"1.3.2"
alert( my$.size() );  //调用方法，返回 undefined
</script>
```

但是，jQuery 不是这样调用的。它模仿类似下面的方法进行调用。

```
$().jquery;
$().size();
```

也就是说，jQuery 没有使用 new 运算符将 jQuery 类实例化，而是直接调用 jQuery()函数，然后在这个函数后面直接调用 jQuery 的原型方法。这是怎么实现的呢？

如果你模仿 jQuery 框架的用法执行下面的代码，浏览器会显示编译错误。这说明上面这个案例代码还不是真正的 jQuery 技术原型。

```
alert($().jquery );
alert($().size() );
```

也就是说，我们应该把 jQuery 看做一个类，同时也应该把它视为一个普通函数，并让这个函数的返回值为 jQuery 类的实例。因此，下面这种结构模型才是正确的。

```
<script language="javascript" type="text/javascript">
var $ =jQuery = function(){
    return new jQuery();    //返回类的实例
}
jQuery.fn = jQuery.prototype = {
    jquery: "1.3.2",
    size: function() {
        return this.length;
    }
}
alert($().jquery );
alert($().size() );
</script>
```

但是，如果在浏览器中预览，则会提示如图 2.1 所示的错误。内存外溢，说明出现了死循环引用。

那么如何返回一个 jQuery 实例呢？

回忆一下，当使用 var my$ = new $();创建 jQuery 类的实例时，this 关键字就指向对象 my$，因此 my$实例对象就获得了 jQuery.prototype 包含的原型属性或方法，这些方法内的

this 关键字就会自动指向 my$实例对象。换句话说，this 关键字总是指向类的实例。

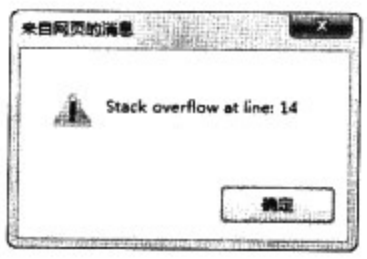


图 2.1　内存外溢错误

因此，我们可以这样尝试：在 jQuery 中使用一个工厂方法来创建一个实例，把这个方法放在 jQuery.prototype 原型对象中，然后在 jQuery()函数中返回这个原型方法的调用。代码如下所示。

```
<script language="javascript" type="text/javascript">
var $ =jQuery = function(){
    return jQuery.fn.init();    //调用原型方法 init()
}
jQuery.fn = jQuery.prototype = {
    init : function(){    //在初始化原型方法中返回实例的引用
        return this;
    },
    jquery: "1.3.2",
    size: function() {
        return this.length;
    }
}
alert( $().jquery );   //调用属性，返回"1.3.2"
alert( $().size() );    //调用方法，返回 undefined
</script>
```

2.2.3　学步——分隔作用域

我们已经初步实现了让 jQuery()函数能够返回 jQuery 类的实例，下面继续思考：init()方法返回的是 this 关键字，该关键字引用的是 jQuery 类的实例，如果在 init()函数中继续使用 this 关键字，也就是说，假设我们把 init()函数也视为一个构造器，则其中的 this 该如何理解和处理？

例如，在下面示例中，jQuery 原型对象中包含一个 length 属性，同时 init()从一个普通的函数转身变成了构造器，它也包含一个 length 属性和一个 test()方法。运行该示例，我们可以看到，this 关键字引用了 init()函数作用域所在的对象，此时它访问 length 属性时，返回 0。而 this 关键字也能够访问上一级对象 jQuery.fn 对象的作用域，所以$().jquery 返回"1.3.2"。但是调用$().size()方法时，返回的是 0，而不是 1。

```
<script language="javascript" type="text/javascript">
```

```
var $ =jQuery = function(){
    return jQuery.fn.init();
}
jQuery.fn = jQuery.prototype = {
    init : function(){
        this.length = 0;
        this.test = function(){
            return this.length;
        }
        return this;
    },
    jquery: "1.3.2",
    length: 1,
    size: function() {
        return this.length;
    }
}
alert( $().jquery );    //返回"1.3.2"
alert( $().test() );     //返回 0
alert( $().size() );     //返回 0
</script>
```

这种设计思路很容易破坏作用域的独立性，对于 jQuery 这样的框架来说，很可能会造成消极影响。因此，我们可以看到 jQuery 框架是通过下面的方式调用 init()初始化构造函数的。

```
<script language="javascript" type="text/javascript">
var $ =jQuery = function(){
    return new jQuery.fn.init();    //实例化 init 初始化类型，分隔作用域
}
</script>
```

这样就可以把 init()构造器中的 this 和 jQuery.fn 对象中的 this 关键字隔离开来，避免相互混淆。但是，这种方式也会带来另一个问题：无法访问 jQuery.fn 对象的属性或方法。例如，在下面的示例中，访问 jQuery.fn 原型对象的 jquery 属性和 size()方法时就会出现这个问题。

```
<script language="javascript" type="text/javascript">
var $ =jQuery = function(){
    return new jQuery.fn.init();
}
jQuery.fn = jQuery.prototype = {
    init : function(){
        this.length = 0;
        this.test = function(){
            return this.length;
        }
        return this;
    },
    jquery: "1.3.2",
    length: 1,
```

```
    size: function() {
        return this.length;
    }
}
alert( $().jquery );    //返回 undefined
alert( $().test() );    //返回 0
alert( $().size() );    //抛出异常
</script>
```

2.2.4 生长——跨域访问

如何做到既能够分隔初始化构造器函数与 jQuery 原型对象的作用域，又能够在返回实例中访问 jQuery 原型对象呢？

jQuery 框架巧妙地通过原型传递解决了这个问题，它把 jQuery.fn 传递给 jQuery.fn.init.prototype，也就是说用 jQuery 的原型对象覆盖 init 构造器的原型对象，从而实现跨域访问，其代码如下所示。

```
<script language="javascript" type="text/javascript">
var $ =jQuery = function(){
    return new jQuery.fn.init();
}
jQuery.fn = jQuery.prototype = {
    init : function(){
        this.length = 0;
        this.test = function(){
            return this.length;
        }
        return this;
    },
    jquery: "1.3.2",
    length: 1,
    size: function() {
        return this.length;
    }
}
jQuery.fn.init.prototype = jQuery.fn;     //使用 jQuery 的原型对象覆盖 init 的原型对象
alert( $().jquery );     //返回"1.3.2"
alert( $().test() );     //返回 0
alert( $().size() );     //返回 0
</script>
```

这是一招妙棋，new jQuery.fn.init()创建的新对象拥有 init 构造器的 prototype 原型对象的方法，通过改变 prototype 指针的指向，使其指向 jQuery 类的 prototype，这样创建出来的对象就继承了 jQuery.fn 原型对象定义的方法。

2.2.5 成熟——选择器

jQuery 返回的是 jQuery 对象，jQuery 对象是一个类数组的对象，本质上它就是一个对象，但是它拥有数组的长度和下标，却没有继承数组的方法。

很显然，上面几节的讲解都是建立在一种空理论基础上的，目的是希望读者能够理解 jQuery 框架的核心构建过程。下面，我们就尝试为 jQuery()函数传递一个参数，并让它返回一个 jQuery 对象。

jQuery()函数包含两个参数 selector 和 context，其中 selector 表示选择器，而 context 表示选择的内容范围，它表示一个 DOM 元素。为了简化操作，我们假设选择器的类型仅限定为标签选择器。实现的代码如下所示。

```
<div></div>
<div></div>
<div></div>
<script language="javascript" type="text/javascript">
var $ =jQuery = function(selector, context ){          //定义类
    return new jQuery.fn.init(selector, context );     //返回选择器的实例
}
jQuery.fn = jQuery.prototype = {          //jQuery 类的原型对象
    init : function(selector, context){ //定义选择器构造器
        selector = selector || document; //设置默认值为 document
        context = context || document;   //设置默认值为 document
        if ( selector.nodeType ) {       //如果选择符为节点对象
            this[0] = selector;          //把参数节点传递给实例对象的数组
            this.length = 1;             //并设置实例对象的 length 属性，定义包含的元素个数
            this.context = selector;     //设置实例的属性，返回选择范围
            return this;                 //返回当前实例
        }
        if ( typeof selector === "string" ) {//如果选择符是字符串
            var e = context.getElementsByTagName(selector);     //获取指定名称的元素
            for(var i = 0;i<e.length;i++){     //遍历元素集合，并把所有元素填入到当前实例数组中
                this[i] = e[i];
            }
            this.length = e.length;  //设置实例的 length 属性，即定义包含的元素个数
            this.context = context;  //设置实例的属性，返回选择范围
            return this;             //返回当前实例
        } else{
            this.length = 0;         //否则，设置实例的 length 属性值为 0
            this.context = context;      //设置实例的属性，返回选择范围
            return this;             //返回当前实例
        }
    },
    jquery: "1.3.2",
    size: function() {
        return this.length;
```

```
    }
}
jQuery.fn.init.prototype = jQuery.fn;
alert( $("div").size() );    //返回3
</script>
```

在上面示例中，$("div")基本拥有了 jQuery 框架中$("div")语法的功能，使用它可以选取页面中指定范围的 div 元素。同时，调用 size()方法可以返回 jQuery 对象集合的长度。

2.2.6 延续——迭代器

在 jQuery 框架中，jQuery 对象是一个很奇怪的概念，具有多重身份，所以很多初学者一听说 jQuery 对象就感觉很是不解，误以为它是 John Resig 制造的新概念。我们可以对 jQuery 对象进行如下分解。

第一，jQuery 对象是一个数据集合，它不是一个个体对象。因此，你无法直接使用 JavaScript 的方法来操作它。

第二，jQuery 对象实际上就是一个普通的对象，因为它是通过 new 运算符创建的一个新的实例对象。它可以继承原型方法或属性，同时也拥有 Object 类型的方法和属性。

第三，jQuery 对象包含数组特性，因为它赋值了数组元素，以数组结构存储返回的数据。我们可以以 JavaScript 的概念理解 jQuery 对象，例如下面的示例。

```
<script language="javascript" type="text/javascript">
var jquery = {                  //定义对象直接量
    name : "jQuery",            //以属性方式存储信息
    value : "1.3.2"
};
jquery[0] = "jQuery";           //以数组方式存储信息
jquery[1] = "1.3.2";
alert(jquery.name);             //返回"jQuery"
alert(jquery[0]);               //返回"jQuery"
</script>
```

上面的 jQuery 对象就是一个典型的 jQuery 对象，jQuery 对象的结构就是按这种形式设计的。可以说，jQuery 对象就是对象和数组的混合体，但是它不拥有数组的方法，因为它的数组结构是人为附加的，也就是说它不是 Array 类型数据，而是 Object 类型数据。

第四，jQuery 对象包含的数据都是 DOM 元素，是通过数组形式存储的，即通过 jQuery[n]形式获取。同时 jQuery 对象又定义了几个模仿 Array 基本特性的属性，如 length 等。

所以，jQuery 对象是不允许直接操作的，只有分别读取它包含的每一个 DOM 元素，才能够实现各种操作，如插入、删除、嵌套、赋值和读写 DOM 元素属性等。

那么如何实现直接操作 jQuery 对象中的 DOM 元素呢？

在实际应用中，我们可以看到类似下面的 jQuery 用法。

```
$("div").html()
```

也就是直接在 jQuery 对象上调用 html()，并实现操作 jQuery 包含的所有 DOM 元素。那么这个功能是怎么实现的呢？

jQuery 定义了一个工具函数 each()，利用这个工具可以遍历 jQuery 对象中所有的 DOM 元素，并把需要操作的内容封装到一个回调函数中，然后通过在每个 DOM 元素上调用这个回调函数即可。实现代码如下所示，演示效果如图 2.2 所示。

```
<div></div>
<div></div>
<div></div>
<script language="javascript" type="text/javascript">
var $ = jQuery = function(selector, context ){
    return new jQuery.fn.init(selector, context );
}
jQuery.fn = jQuery.prototype ={
    init : function(selector, context){
        //省略的初始化构造器主体代码，请参阅上节示例
    },
    //定义 jQuery 对象方法
    html : function(val){    //模仿 jQuery 框架中的 html()方法，为匹配的每一个 DOM 元素插入 html
代码
        jQuery.each(this, function(val){    //调用 jQuery.each()工具函数，为每一个 DOM 元素
执行回调函数
            this.innerHTML = val;
        }, val);
    }
}
jQuery.fn.init.prototype = jQuery.fn;
//扩展 jQuery 工具函数
jQuery.each = function( object, callback, args ){
    for(var i = 0; i<object.length; i++){
        callback.call(object[i],args);
    }
    return object;
}
//示例测试
$("div").html("测试代码");
</script>
```

在上面的示例中，通过先为自己的 jQuery 对象绑定 html()方法，然后利用 jQuery()选择器获取页面中所有的 div 元素，再调用 html()方法，为所有匹配的元素插入 HTML 源码。

注意，在上面的代码中，each()函数的当前作用对象是 jQuery 对象，故 this 指向当前 jQuery 对象，即 this 表示一个集合对象；而在 html()方法中，由于 each()函数是在指定 DOM 元素上

执行的，所以该函数内的 this 指针指向的是当前 DOM 元素对象，即 this 表示一个元素。

图 2.2 模拟 jQuery 对象的 html()方法

当然，上面示例所定义的 each()工具函数比较简单，适应能力比较有限。在 jQuery 框架中，它封装的 each()函数功能就强大很多，具体代码如下所示。

```
jQuery.extend({
    //参数说明：object 表示 jQuery 对象，callback 表示回调函数，args 回调函数的参数数组
    each: function( object, callback, args ) {
        var name, i = 0, length = object.length;
        if ( args ) {    //如果存在回调函数的参数数组
            if ( length === undefined ) { //如果 object 不是 jQuery 对象
                for ( name in object )     //遍历 object 的属性
                    if ( callback.apply( object[ name ], args ) === false )
                                           //在对象上调用回调函数
                        break;             //如果回调函数返回值为 false，则跳出循环
            } else                         //如果 object 是 jQuery 对象
                for ( ; i < length; )      //遍历 jQuery 对象数组
                    if ( callback.apply( object[ i++ ], args ) === false )
                                           //在对象上调用回调函数
                        break;             //如果回调函数返回值为 false，则跳出循环
        } else {
            if ( length === undefined ) { //如果 object 不是 jQuery 对象
                for ( name in object )     //遍历 object 的属性
                    if ( callback.call( object[ name ], name, object[ name ] ) === false )
                        break;             //如果回调函数返回值为 false，则跳出循环
            } else                         //如果 object 是 jQuery 对象
                for ( var value = object[0];     //遍历 jQuery 对象数组
                    i < length && callback.call( value, i, value ) !== false; value =
object[++i] ){}
        }
        return object;                     //返回 jQuery 对象
    }
})
```

同时，jQuery 框架定义的 html()方法如下所示。由于该方法包含的功能比较多，它不仅可以插入 HTML 源代码，还可以返回匹配元素包含的 HTML 源代码，故它使用了一个条件结构分别进行处理。首先，判断参数是否为空，如果为空，则表示获取匹配元素中第一个元素包含的 HTML 源代码，此时返回该 innerHTML 的值。如果不为空，则先清空匹配元素中每个

元素包含的内容，并使用 append()方法插入 HTML 源代码。

```
jQuery.fn = jQuery.prototype = {
    html: function( value ) {
        return value === undefined ?
            (this[0] ?
                this[0].innerHTML.replace(/ jQuery\d+="(?:\d+|null)"/g, "") :
                null) :
            this.empty().append( value );
    }
}
```

2.2.7 延续——功能扩展

根据一般设计习惯，如果要为 jQuery 或者 jQuery.prototype 添加函数或方法，可以直接通过点语法实现，或者在 jQuery.prototype 对象结构中增加一个属性即可。但是，如果分析 jQuery 框架的源代码，你会发现它是通过 extend()函数来实现功能扩展的。例如，下面两段代码都是 jQuery 框架通过 extend()函数来扩展功能的。

```
jQuery.extend({    //扩展工具函数
    noConflict: function( deep ) {},
    isFunction: function( obj ) {},
    isArray: function( obj ) {},
    isXMLDoc: function( elem ) {},
    globalEval: function( data ) {}
});
```

或者

```
jQuery.fn.extend({    //扩展 jQuery 对象方法
    show: function(speed,callback){},
    hide: function(speed,callback){},
    toggle: function( fn, fn2 ){},
    fadeTo: function(speed,to,callback){},
    animate: function( prop, speed, easing, callback ) {},
    stop: function(clearQueue, gotoEnd){}
});
```

这样做的好处是什么呢？

extend()函数能够方便用户快速扩展 jQuery 框架的功能，但是不会破坏 jQuery 框架的原型结构，从而避免后期人工手动添加工具函数或者方法时破坏 jQuery 框架的单纯性，同时也方便管理。如果不需要某个插件，只需要简单地删除即可，而不需要在 jQuery 框架源代码中去筛选和删除。

extend()函数的功能实现起来也很简单，它只是把指定对象的方法复制给 jQuery 对象或

者 jQuery.prototype 对象。例如，在下面的示例中，我们为 jQuery 类和原型定义了一个扩展功能的函数 extend()，该函数的功能很简单，它能够把指定参数对象包含的所有属性复制给 jQuery 或者 jQuery.prototype 对象，这样就可以在应用中随时调用它，并动态扩展 jQuery 对象的方法。

```
<script language="javascript" type="text/javascript">
var $ = jQuery = function(selector, context ){
    return new jQuery.fn.init(selector, context );
}
jQuery.fn = jQuery.prototype ={
    init : function(selector, context){
    }
}
jQuery.fn.init.prototype = jQuery.fn;
//jQuery 功能扩展函数
jQuery.extend = jQuery.fn.extend = function(obj) {
    for (var prop in obj) {
        this[prop] = obj[prop];
    }
    return this;
}
//扩展 jQuery 对象方法
jQuery.fn.extend({
    test : function(){
        alert("测试扩展功能");
    }
})
//测试代码
$("div").test();
</script>
```

在上面的示例中，先定义了一个功能扩展函数 extend()，然后为 jQuery.fn 原型对象调用 extend()函数，为其添加一个测试方法 test()。这样就可以在实践中应用，如$("div").test()。

jQuery 框架定义的 extend()函数的功能要强大很多，它不仅能够完成基本的功能扩展，还可以实现对象合并等功能，详细代码和解释如下所示。

```
jQuery.extend = jQuery.fn.extend = function() {
    // 定义复制操作的目标对象
    var target = arguments[0] || {}, i = 1, length = arguments.length, deep = false,
options;
    //获取是否深度复制处理
    if ( typeof target === "boolean" ) {
        deep = target;
        target = arguments[1] || {};
        // 跳出布尔值和目标对象
        i = 2;
    }
    // 如果第一个参数是字符串，则设置为空对象
```

```
if ( typeof target !== "object" && !jQuery.isFunction(target) )
    target = {};
//如果只有一个参数，表示把参数对象的方法复制给当前对象，则设置 target 为 this
if ( length == i ) {
    target = this;
    --i;
}
for ( ; i < length; i++ )    //遍历参数
    // 若参数值不为 null，则进行处理
    if ( (options = arguments[ i ]) != null )
        // Extend the base object
        for ( var name in options ) {    //遍历参数对象
            var src = target[ name ], copy = options[ name ];
            // 防止死循环访问
            if ( target === copy )
                continue;
            // 递归运算
            if ( deep && copy && typeof copy === "object" && !copy.nodeType )
                target[ name ] = jQuery.extend( deep,
                    // 不要复制原对象
                    src || ( copy.length != null ? [ ] : { } )
                , copy );
            // 不要传递未定义的值
            else if ( copy !== undefined )
                target[ name ] = copy;
        }
// 返回修改后的对象
    return target;
};
```

2.2.8 延续——参数处理

在很多时候，你会发现 jQuery 的方法都要求传递的参数为对象结构，例如：

```
$.ajax({
    type: "GET",
    url: "test.js",
    dataType: "script"
});
```

使用对象直接量作为参数进行传递，方便参数管理。当方法或者函数的参数长度不固定时，使用对象直接量作为参数存在很多优势。例如，对于下面的用法，ajax()函数就需要进行更加复杂的参数排查和过滤。

```
$.ajax("GET", "test.js","script");
```

如果 ajax()函数的参数长度是固定的，且是必须的，那么通过这种方式进行传递也就无所谓了，但是如果参数的个数和排序是动态的，那么使用$.ajax("GET", "test.js","script")这种

方法是无法处理的。而 jQuery 框架的很多方法都包含大量的参数，且都是可选的，位置也没有固定要求，所以使用对象直接量是惟一的解决方法。

使用对象直接量作为参数传递的载体，这里就涉及参数处理问题。如何解析并提出参数？如何处理参数和默认值？我们可以通过下面的方法来实现。

```
<script language="javascript" type="text/javascript">
var $ = jQuery = function(selector, context ){
    return new jQuery.fn.init(selector, context );
}
jQuery.fn = jQuery.prototype ={
    init : function(selector, context) {},
    setOptions : function(options) {
        this.options ={    //方法的默认值，可以扩展
            StartColor :    "#000",
            EndColor :       "#DDC",
            Background :    false,
            Step :           20,
            Speed :       10
        };
        jQuery.extend(this.options, options || {});    //如果传递参数，则覆盖原默认参数
    }
}
jQuery.fn.init.prototype = jQuery.fn;
jQuery.extend = jQuery.fn.extend = function(destination, source){    //重新定义 extend()
函数
    for (var property in source) {
        destination[property] = source[property];
    }
    return destination;
}
</script>
```

在上面的示例中，定义了一个原型方法 setOptions()，该方法能够对传递的参数对象进行处理，并覆盖默认值。这种用法在本书插件部分还将进行讲解，故这里就不再展开。

在 jQuery 框架中，extend()函数包含了所有功能，它既能够为当前对象扩展方法，也能够处理参数对象，并覆盖默认值。

2.2.9 涅槃——名字空间

现在，我们终于模拟出了 jQuery 框架的雏形，虽然它还比较稚嫩，经不起风雨，但至少能够保证读者理解 jQuery 框架构成的初期状态。不过对于一个成熟的框架来说，需要设计者考虑的问题还是很多的，其中最核心的问题就是名字空间冲突问题。

当一个页面中存在多个框架，或者自己写了很多 JavaScript 代码，我们是很难确保这些

代码不发生冲突的，因为任何人都无法确保自己非常熟悉 jQuery 框架中的每一行代码，所以难免会出现名字冲突，或者功能覆盖现象。为了解决这个问题，我们必须把 jQuery 封装在一个孤立的环境中，避免其他代码的干扰。

在详细讲解名字空间之前，我们先来温习两个 JavaScript 概念。首先，请看下面的代码。

```
var jQuery = function(){};
jQuery = function(){};
```

上面所示的代码是两种不同的写法，且都是合法的，但是它们的语义完全不同。第一行代码声明了一个变量，而第二行代码定义了 Window 对象的一个属性，也就是说它等同于下面的语句。

```
window.jQuery = function(){};
```

在全局作用域中，变量和 Window 对象的属性可以是相等的，也可以是互通的，但是当在其他环境中(如局部作用域中)，则它们是不相等的，也是无法互通的。

因此如果希望 jQuery 具有类似$.method();调用方式的能力，就需要将 jQuery 设置为 Window 对象的一个属性，所以你就会看到 jQuery 框架中是这样定义的。

```
var jQuery = window.jQuery = window.$ = function( selector, context ) {
    return new jQuery.fn.init( selector, context );
}
```

你可能看到过下面的函数用法。

```
(function(){
  alert("观察我什么时候出现");
})();
```

这是一个典型的匿名函数基本形式。为什么要用到匿名函数呢？

这时就要进入正题了，如果希望自己的 jQuery 框架与其他任何代码完全隔离开来，也就是说如果你想把 jQuery 装在一个封闭空间中，不希望暴露内部信息，也不希望别的代码随意访问，匿名函数就是一种最好的封闭方式。此时我们只需要提供接口，就可以方便地与外界进行联系。例如，在下面的示例中分别把 f1 函数放在一个匿名函数中，而把 f2 函数放在全局作用域中。可以发现，全局作用域中的 f2 函数可以允许访问，而匿名函数中的 f1 函数是禁止外界访问的。

```
<script language="javascript" type="text/javascript">
(function(){
    function f1(){
        return "f1()";
    };
})();
```

```
function f2(){
    return "f2()";
};
alert( f2() );     //返回"f2()";
alert( f1() );     //抛出异常，禁止访问
</script>
```

实际上，上面的匿名函数就是所谓的闭包，闭包是 JavaScript 函数中一个最核心的概念。

当然，$和 jQuery 名字并非是 jQuery 框架的专利，其他一些经典框架中也会用到$名字，也许读者也会定义自己的变量 jQuery。

在这之前我们需要让它与其他框架协同工作，这就带来一个问题，如果我们都使用$作为简写形式就会发生冲突，为此jQuery提供了一个noConfilit()方法，该方法能够实现禁止jQuery框架使用这两个名字。为了实现这样的目的，jQuery 在框架的最前面，先使用_$和_jQuery 临时变量寄存$和 jQuery 这两个变量的内容，当需要禁用 jQuery 框架的名字时，可以使用一个临时变量_$和_jQuery 恢复$和 jQuery 这两个变量的实际内容。实现代码如下。

```
(function(){
var
    window = this,
    undefined,
    _jQuery = window.jQuery,      //暂存 jQuery 变量内容
    _$ = window.$,      //暂存$变量内容
    jQuery = window.jQuery = window.$ = function( selector, context ) {
        return new jQuery.fn.init( selector, context );
    },
    quickExpr = /^[^<]*(<(.|\s)+>)[^>]*$|^#([\w-]+)$/,
    isSimple = /^.[^:#\[\.,]*$/;
jQuery.fn = jQuery.prototype = {
    init: function( selector, context ) {}
}
})()
```

至此，jQuery 框架的设计模式就初见端倪了，后面的工作就是根据应用需要或者功能需要，使用 extend()函数不断扩展 jQuery 框架的工具函数和 jQuery 对象的方法。

2.3 破解 jQuery 选择器接口

jQuery 选择器功能强大，但是用法简单，它仅仅提供了一个接口：jQuery()，也可以简写为$()。用法如此简单，但又具有如此强大的处理能力，使 jQuery 必然成为众人追捧的对象。

在上一节中，我们重点分析了 jQuery 框架的雏形，而对于选择器并没有深入分析，仅仅提供了一个简单的 DOM 元素选择作为演示，目的是方便读者理解该框架的架设思路和过程。

本节将重点研究 jQuery 选择器的设计思路、实现过程和工作原理。

2.3.1 简单但很复杂的黑洞

前面说到，jQuery 提供了惟一的接口(jQuery()或者$())使选择器与外界进行交流。那么这个对象是如何生成的呢？

jQuery 框架的基础是查询，即查询文档元素对象，因此我们可以认为 jQuery 对象就是一个选择器，并在此基础上构建和运行查询过滤器。

jQuery 查询的结果是获取 DOM 元素，这些查询到的 DOM 元素又是如何存储的呢？

根据前面的介绍，我们初步了解到它把查询的结果存储到 jQuery 对象内。由于查询的结果可能是单个元素，也可能是集合，因此，jQuery 对象内应该定义了一个集合。这个集合专门负责存放查询到的 DOM 元素。这正如 JavaScript 中的 Function 对象一样，其内部也构建了一个集合对象 Arguments，专门负责存储函数的参数。

但是，Function 对象和 Arguments 是两个相互独立的概念，仅通过 arguments 属性联系在一起。也就是说 Arguments 对象并非是 Function 对象的子对象，或者是它的内部组成部分。而 jQuery 对象与查询结果的数据集合就不同了，它是完全作为 jQuery 对象的一部分而存在的。

另外，jQuery 虽然仅提供了一个入口，但是它的构建并不只局限于从 DOM 文档树中查询到 DOM 元素，DOM 元素也有可能是从别的集合中转移过来的，或是从 HTML 片断生成的等。

例如，类似下面的代码在 jQuery 应用中经常会看到。

```
$("div.red").css("display","none");      //将 class 为 red 的 div 元素隐藏显示
var width = $("div .red").width();      //获取 div 元素下 class 为 red 的元素的宽度
var html = $(document.getElementById("wrap")).html();//获取 id 为 wrap 元素的 innerHTML 值
$("#wrap", document.forms[0]).css("color", "red");
//将在第一个 form 元素下 id 为 wrap 元素的字体颜色设置为红色
$("<div>hello,world</div>").appendTo("#wrap");
//将 HTML 字符串信息追加到 id 为 wrap 元素的末尾
```

在$()函数中可以包含选择符字符串、HTML 字符串、DOM 对象和数据集合等不同类型的参数。jQuery 是如何分辨这些参数是选择符字符串、HTML 字符串、DOM 对象或数据集合的呢？

为了方便读者理解这其中的奥妙，我们不妨把 jQuery 框架进行简化，先删除所有方法、函数以及逻辑代码，然后在 init()构造器中，使用 alert()方法获取 selector 参数的类型和信息，其代码如下。

```
<script language="javascript" type="text/javascript">
```

```
(function(){
    var window = this,
    jQuery = window.jQuery = window.$ = function(selector, context){
        return new jQuery.fn.init(selector, context);
    };
    jQuery.fn = jQuery.prototype = {
        init : function(selector, context) {
            alert(selector);
        }       .
    };
})();
window.onload = function() {
    $("div.red");    //获取"div.red"
    $("div .red");    //获取" div .red "
    $(document.getElementById("wrap")); //获取"[object]"
    $("#wrap", document.forms[0]);      //获取"#wrap"
    $("<div>hello,world</div>");        //获取"<div>hello,world</div>"
}
</script>
<div id="wrap"></div>
```

2.3.2 盘根错节的逻辑关系

根据 jQuery 官网提供的 API 文档可知，jQuery()提供了以下 4 种构建 jQuery 对象的方式。

- jQuery(expression, [context])
- jQuery(html, [ownerDocument])
- jQuery(elements)
- jQuery(callback)

其中 jQuery 可以使用$简写。上述四种构建 jQuery 对象的方式是经常用到的。从上述参数列表可以看出，其实 jQuery 的参数可以是任意元素。例如：

```
$("div > p");      //参数可以是字符串
$( $("div > p")); //参数可以是 jQuery 对象或者类数组(ArrayLike)的集合
$(document);       //参数可以是 DOM 元素
$();               // $(document)简写
$(function(){}); // $(document).ready()的简写
$([]);             // 参数可以是数组
$({});             // 参数可以是对象
$(1) ;             // 参数可以是数字，即把 1 存储 jQuery 对象的数据集合中
```

虽然说，在上面的示例中最后 4 行代码都可以被解析，但是这些参数数据是被存储到 ArrayLike(类数组)集合中的，而不是被转换为 DOM 元素。虽然语法不错，解析正常，但是它们无法完成实际应用，所以不建议传入非 DOM 元素的参数。

注意：jQuery 对象的方法都是针对 DOM 元素对象进行的操作，如果不清楚其使用的话，很有可能会导致错误。

下面我们就顺着 jQuery 框架的这个惟一入口，慢慢向里爬进，以窥视其中的秘密。当我们调用 jQuery()方法时，它没有被实例化，也就是说 jQuery 类型被抛弃了，我们仅仅把它作为一个普通函数来调用，此时该方法中的 this 关键字指向的是 Window 对象，而不是 jQuery 对象，请读者务必注意。

不过，当调用该方法时，会返回一个 jQuery.fn.init 类型的实例，同时，jQuery 又使用自己的原型对象覆盖了 jQuery.fn.init 类型的原型对象，所以就形成了一种错觉，很多初学者往往在这里栽了跟头。下面是 jQuery 框架中的核心代码(节选)。

```
jQuery = window.jQuery = window.$ = function( selector, context ) {
    return new jQuery.fn.init( selector, context );
}
jQuery.fn.init.prototype = jQuery.fn;
```

jQuery 对象不是通过 new jQuery 来继承其 prototype 中的方法的，而是通过 jQuery.fn.init 初始化构造器生成的。所以，为 jQuery.prototype 添加函数集也就失去了存在价值。虽然直接使用 new jQuery()也是允许的，但是由于该函数的返回值覆盖了 new jQuery()创建的实例对象，所以使用 new jQuery()来构建 jQuery 对象也是无法存活的。

总之，jQuery 对象其实就是 jQuery.fn.init 构造器创建的对象，而通过 jQuery.fn.init.prototype = jQuery.fn;途径，再使用 jQuery 的原型对象去覆盖 jQuery.fn 的原型对象，使得 jQuery 对象的原型方法也就被继承过来，从而形成了错综复杂但又井然有序的关系。

2.3.3 jQuery 构造器

jQuery.fn.init()负责对传入参数进行分析，然后生成并返回 jQuery 对象。jQuery.fn.init()构造器的第一个参数是必须的，如果为空，则默认为 document。

从本质上讲，使用 jQuery 选择器(即 jQuery.fn.init()构造器)构建 jQuery 对象，就是在 this 对象上附加 DOM 元素集合。附加的方式包括以下两类。

- 如果是单个 DOM 元素，可以直接把 DOM 元素作为数组元素传递给 this 对象，还可以通过 ID 从 DOM 文档中查询元素。
- 如果是多个 DOM 元素，则以集合形式附加，如 jQuery 对象、数组和对象等，此时可以通过 CSS 选择器匹配到所有 DOM 元素，然后过滤，最后构建类数组的数据结构。

而 CSS 选择器，则是通过 jQuery().find(selector)函数来完成的。通过 jQuery().find(selector)可以分析选择器字符串，并在 DOM 文档树中查找到符合语法的元素集合。这个函

数我们将在下面章节进行分析。该函数能够兼容 CSS1～CSS3 选择器。

下面就从 init()初始化构造器函数开始，来分析 jQuery 选择器是如何工作的。为了方便解释，我们先结合源代码进行讲解。

```
//jQuery 原型对象
//构造 jQuery 对象的入口
//所有 jQuery 对象方法都通过 jQuery 原型对象来继承
jQuery.fn = jQuery.prototype = {
    //jQuery 对象初始化构造器，相当于 jQuery 对象的类型，由该函数负责创建 jQuery 对象
    参数说明：selector：选择器的符号，可以是任意数据类型。考虑 DOM 元素操作需要，该参数应该是包含 DOM 元素
    的任何数据
    context：上下文，指定在文档 DOM 中哪个节点下开始进行查询，默认值为 document
    init: function( selector, context ) {
        selector = selector || document;     //确保 selector 参数存在，默认值为 document
        //第一种情况，处理选择符为 DOM 元素，此时将忽略上下文，即忽略第二个参数
        //例如，$(document.getElementById("wrap"))，jQuery (DOMElement)匹配 DOM 元素。
        //先使用 selector.nodeType 判断当 selector 为元素节点，将 length 设置为 1，
        //并且赋值给 context，实际上 context 作为 init 的第二个参数，
        //也意味着它的上下文节点就是 selector 该点，返回它的$(DOMElement)对象
        if (selector.nodeType) {     //存在 nodeType 属性，说明选择符是一个 DOM 元素
            this[0] = selector;     //直接把当前参数的 DOM 元素存入类数组中
            this.length = 1;      //设置类数组的长度，以方便遍历访问
            this.context = selector;     //设置上下文属性
            return this;     //返回 jQuery 对象，即类数组对象
        }
 //如果选择符参数为字符串，则进行处理
//例如，$("<div>hello,world</div>")，jQuery(html,[ownerDocument])匹配 HTML 字符串
 if (typeof selector == "string") {
    //使用 quickExpr 正则表达式匹配该选择符字符串，决定是处理 HTML 字符串，还是处理 ID 字符串
    // quickExpr = /^[^<]*(<(.|\s)+>)[^>]*$|^#([\w-]+)$/
    var match = quickExpr.exec(selector);
    // 验证匹配的信息，任何情况下都不是#id
    if (match && (match[1] || !context)) {
        //第二种情况，处理 HTML 字符串，类似$(html) -> $(array)
        if (match[1])
            selector = jQuery.clean([match[1]], context);
        //第三种情况，处理 ID 字符串，类似$("#id")
        else {
            var elem = document.getElementById(match[3]); //获取该元素确保元素存在
            处理在 IE 和 Opera 浏览器下根据 name，而不是 ID 返回元素
            if ( elem && elem.id != match[3] )
                return jQuery().find( selector ); // 默认调用 document.find()方法
            // 否则将把 elem 作为元素参数直接调用 jQuery()函数，返回 jQuery 对象
            var ret = jQuery( elem || [] );
            ret.context = document;     // 设置 jQuery 对象的上下文属性
            ret.selector = selector;     // 设置 jQuery 对象的上选择符属性
            return ret;     // 返回 jQuery 对象
        }
    } else
```

```
                //第四种情况，处理jQuery(expression, [context]),
                //例如，$("div .red")的表达式字符串
                return jQuery( context ).find( selector );
            } else if ( jQuery.isFunction( selector ) )
                //第五种情况，处理jQuery(callback)，即$(document).ready()的简写
                //例如， $(function(){ alert("hello,world");}) ,
                //或者$(document).ready(function(){ alert("hello,world");})
                return jQuery( document ).ready( selector );
               // 确保旧的选择符能够通过
            if ( selector.selector && selector.context ) {
                this.selector = selector.selector;
                this.context = selector.context;
            }
            // 第六种情况，处理类似$(elements)
            return this.setArray(jQuery.isArray( selector ) ?
                selector :
                jQuery.makeArray(selector));
        }
    //.....
    }
```

进一步分析 init()构造器函数的设计思路如下。

(1) 第一步，当第一个参数为 DOM 元素，则废除第二个参数，直接把 DOM 元素存储到 jQuery 对象的集合中，返回该 jQuery 对象。

(2) 第二步，如果第一个参数是字符串，则可能存在三种情况。

- 情况一，第一个参数是 HTML 标签字符串，第二个参数可选，则执行 selector = jQuery.clean([match[1]], context);，该语句能够把 HTML 字符串转换成 DOM 对象的数组，然后执行 Array 类型数组并返回 jQuery 对象。
- 情况二，第一个参数是#id 字符串，即类似$(id)，则先使用 document.getElementById()方法获取该元素，如果没有获得元素，则设置 selector = []，转到执行 Array 类型，并返回空集合的 jQuery 对象。如果获得元素，则构建 jQuery 对象并返回。这里，把#id 单独列出，是为了提高性能。
- 情况三，处理复杂的 CSS 选择符字符串，第二个参数是可选的。通过 return jQuery().find(selector);语句实现。该语句先执行 jQuery(context)，可以看出第二参数 context 可以是任意值，也可以是集合数据。然后调用 find(selector)找到 jQuery (context)上下文中所有的 DOM 元素，即这些元素都满足 selector 表达式，最后构建 jQuery 对象并返回。

(3) 第三步，如果第一个参数是函数，则第二个参数可选。它是$(document).ready(fn)形式的简写，return jQuery(document)[jQuery.fn.ready ? "ready" : "load"](selector)是其执行的代码。该语句先执行 jQuery(document)，再通过 new jQuery.fn.init()方式创建 jQuery 对象，此时元素为 document。再调用这个对象的 ready()方法，并返回当前的 jQuery 对象。

$(document).ready(fn)是实现 domReady 的 jQuery 对象的统一入口，可以通过$(fn)注册 domReady 的监听函数。所有的调用 jQuery 实现功能的代码都应该在 domReady 之后才能够运行。$(fn)是所有应用开发中的功能代码的入口，它支持任意多的$(fn)注册。

(4) 第四步，如果第一个参数是除 DOM 元素、函数和字符串之外的所有其他类型，也可以为空(如$())，而第二个参数可选。调用 return this.setArray(jQuery.makeArray(selector)); 进行处理时，它先是把第一个参数转换为数组。当然这个参数可以是类数组结构的集合，如 jQuery 对象、getElementsByTag 返回的 DOM 元素集合等，可支持$(this)。selector 还可能是单个任意对象，转换成标准的数组之后，执行 this.setArray 把这个数组中的元素全部存储到当前的 jQuery 对象集合中，并返回 jQuery 对象。

2.3.4 生成 DOM 元素

jQuery.fn.init()构造函数能够构建 jQuery 对象，并把匹配的 DOM 元素存储在 jQuery 对象内部集合中。jQuery.fn.init()构造函数可以接收单个的 DOM 元素，也可以接收 DOM 集合。如果接收的是字符串型 ID 值，则直接在文档中查找对应的 DOM 元素，并把它传递给 jQuery 对象；如果接收的是字符串型 HTML 片段，则需要把这个字符串片段生成 DOM 元素。下面我们将重点分析 jQuery 是如何把 HTML 片段生成 DOM 元素的。

在 2.3.3 节中，我们可以看到 jQuery.fn.init()构造器通过 jQuery.clean([match[1]], context); 语句实现把 HTML 片断生成 DOM 元素，jQuery.clean()是一个公共函数。源代码及其注释如下所示。

jQuery.clean()包含三个参数，其中 elems 和 context 可以支持多种形式的值。Elems 参数可以为数组、类数组、对象结构的形式。数组元素和对象属性可以混合使用。

对于数字类型参数，则会被转换为字符串型，除了字符串型外，其他的都放入返回的数组中，当然对于集合形式只需要读取集合中每个元素即可。

对于字符串型参数，则把它转换成 DOM 元素，再存入返回的数组中。转换的方式是，把 HTML 字符串片段赋值给创建的 div 元素的 innerHTML，这样就可以把 HTML 字符串片段挂到 DOM 文档树中，从而实现把字符串转换成 DOM 元素。

在转换过程中，应该考虑 HTML 语法约定，因为标签嵌套是有严格限制的，例如，<td>必须存在<tr>中。因此在执行转换前，还应该对 HTML 字符串进行预处理，即修正 HTML 标签不规范的用法，这也是 jQuery.clean()函数的一个重要工作。

```
//公共函数扩展
jQuery.extend({
    //把 HTML 字符串片段转换成 DOM 元素
    //参数说明：
    //elems 参数表示多个 HTML 字符串片段的数组
```

```
    // context 参数表示上下文
    // fragment 参数表示框架对象
    clean: function( elems, context, fragment ) {
        context = context || document;    //默认的上下文是 document
         //在 IE 中!context.createElement 是错误用法，因为它返回的是对象类型，而不是逻辑值，故通过返回类型进行判断
        if ( typeof context.createElement === "undefined" )
            //支持 context 为 jQuery 对象，并获取第一个元素
            context = context.ownerDocument || context[0] && context[0].ownerDocument || document;
        //如果近匹配一个标签，且没有指定框架参数，则直接创建 DOM 元素，并跳过后面的解析
        if ( !fragment && elems.length === 1 && typeof elems[0] === "string" ) {
            var match = /^<(\w+)\s*\/?>$/.exec(elems[0]);
            if ( match )
                return [ context.createElement( match[1] ) ];
        }
        var ret = [], scripts = [], div = context.createElement("div");
        //匹配每一个 HTML 字符串片段，并为每一个片段执行回调函数
        jQuery.each(elems, function(i, elem){
            if ( typeof elem === "number" )    //把数值转换为字符串的高效方法
                elem += '';
            if ( !elem )    //如果不存在元素，则返回，或者为''、undefined、false 等时也返回
                return;
            //把 HTML 字符串转换为 DOM 节点
            if ( typeof elem === "string" ) {
                //统一转换为 XHTML 严谨型文档的标签形式，如<div/>的形式修改为<div></div>
                //但是对于(abbr|br|col|img|input|link|meta|param|hr|area|embed)不修改
                //front=(<(\w+)[^>]*?)
                elem = elem.replace(/(<(\w+)[^>]*?)\/>/g, function(all, front, tag){
                    return
tag.match(/^(abbr|br|col|img|input|link|meta|param|hr|area|embed)$/i) ?
                        all :
                        front + "></" + tag + ">";
                });
                // 清除空格，否则 indexof 可能会出现不能正常工作
                var tags = elem.replace(/^\s+/, "").substring(0, 10).toLowerCase();
                //有些标签必须是有一些约束的，如<option>必须在<select></select>中间
                //下面代码大部分是对<table>中的子元素进行修正。数组中第一个元素为深度
                var wrap =
                    //约束<option>，<opt 在开始位置上(index=0)就返回&&运算符后面的数组
                    !tags.indexOf("<opt") &&
                    [ 1, "<select multiple='multiple'>", "</select>" ] ||
                    //<leg 必须在<fieldset>内部
                    !tags.indexOf("<leg") &&
                    [ 1, "<fieldset>", "</fieldset>" ] ||
                    //thead|tbody|tfoot|colg|cap 必须在<table>内部
                    tags.match(/^<(thead|tbody|tfoot|colg|cap)/) &&
                    [ 1, "<table>", "</table>" ] ||
                    !tags.indexOf("<tr") &&
                    [ 2, "<table><tbody>", "</tbody></table>" ] ||
                     //<tr 在<tbody>中间
                     //td 在 tr 中间
```

```
        //col 在<colgroup>中间
        (!tags.indexOf("<td") || !tags.indexOf("<th")) &&
        [ 3, "<table><tbody><tr>", "</tr></tbody></table>" ] ||
        !tags.indexOf("<col") &&
        [ 2, "<table><tbody></tbody><colgroup>", "</colgroup></table>" ] ||
        // IE can't serialize <link> and <script> tags normally
        !jQuery.support.htmlSerialize &&
        //IE 中 link script 不能串行化
        [ 1, "div<div>", "</div>" ] ||
        //默认不修正
        [ 0, "", "" ];
    // 包裹 HTML 之后，采用 innerHTML 转换成 DOM
    div.innerHTML = wrap[1] + elem + wrap[2];
    // 转到正确的深度，对于[1, "<table>","</table>"]，div=<table>
    while ( wrap[0]-- )
        div = div.lastChild;
    // fragments 去掉 IE 对<table>自动插入的<tbody>
    if ( !jQuery.support.tbody ) {
        //第一种情况：tags 以<table>开头但没有<tbody>。在 IE 生成的元素中可能会自动加<tbody>
        // 第二种情况：thead|tbody|tfoot|colg|cap 为 tags，那 wrap[1] == "<table>"
        var hasBody = /<tbody/i.test(elem),
            tbody = !tags.indexOf("<table") && !hasBody ?
                div.firstChild && div.firstChild.childNodes :
            // tbody 不一定是 tbody，也有可能是 thead 等
            wrap[1] == "<table>" && !hasBody ?
                div.childNodes :
                [];
                //除去<tbody>
                for ( var j = tbody.length - 1; j >= 0 ; --j )
                    if ( jQuery.nodeName( tbody[ j ], "tbody" ) && !tbody[ j ].childNodes.length )
                        tbody[ j ].parentNode.removeChild( tbody[ j ] );
                }
            //使用 innerHTML，IE 会去掉开头的空格节点，因此加上去掉的空格节点
            if ( !jQuery.support.leadingWhitespace && /^\s/.test( elem ) )
                div.insertBefore( context.createTextNode( elem.match(/^\s*/)[0] ),
div.firstChild );
            //elem 从字符转换成了数组
            elem = jQuery.makeArray( div.childNodes );
        }
        //如果是 DOM 元素，则推入数组，否则就合并数组
        if ( elem.nodeType )
            ret.push( elem );
        else
            ret = jQuery.merge( ret, elem );
    });
    //如果指定了第 3 个参数，即框架对象，则附加到框架对象上
    //这段是新增加的，用来处理 js 代码，同时也取消了 form 的处理
    if ( fragment ) {
        for ( var i = 0; ret[i]; i++ ) {
            if ( jQuery.nodeName( ret[i], "script" ) && (!ret[i].type || ret[i].type.toLowerCase()
=== "text/javascript") ) {
```

```
                    scripts.push( ret[i].parentNode ? ret[i].parentNode.removeChild( ret[i] ) :
ret[i] );
                } else {
                    if ( ret[i].nodeType === 1 )
                        ret.splice.apply( ret, [i + 1, 0].concat(jQuery.makeArray(ret[i].
getElementsByTagName ("script"))) );
                    fragment.appendChild( ret[i] );
                }
            }
            //返回脚本
            return scripts;
        }
        //返回 DOM 元素集合
        return ret;
    },
//……
}
```

上面这段代码实际上最后访问的是一个名为 ret 的数组，数组中的元素变为 DOM 元素的对象，而它的 innerHTML 正好就是刚才的 HTML 字符串。

2.3.5 引用 DOM 元素

jQuery()函数能够直接接受 HTML 字符串，并把它们转换为 DOM 结构，这是上一节中所讲解的利用 jQuery()函数生成 DOM 元素。当然，我们也可以看到 jQuery()函数还可以接收 DOM 元素、DOM 元素集合、HTML 标签或者 ID 值。下面我们就来分析 jQuery.fn.init()构造器是如何把这些类型的参数转换为 DOM 元素的。

对于 HTML 标签来说，它使用 document.getElementsByTagName()方法获取 DOM 元素集合。对于 ID 参数来说，它使用 document.getElementById()方法获取特定的 DOM 元素。另外，还可以使用 DOM 元素的 childNodes、firstChild、lastChild、nextSibling、parentNode 和 previousSibling 等属性引用 DOM 节点。

既然说 DOM 元素能够通过 lastChild、parentNode 等属性引用节点，jQuery 对象又是 DOM 元素的集合，因此，jQuery 就可以考虑通过整合 DOM 元素的这些属性来获得其集合中所有元素各自引用的节点。把这些间接引用的节点组合起来又构成了新的 DOM 元素集合。下面我们就来分析 jQuery 是如何引用节点的。

```
//通过 each()方法为 jQuery 对象映射一组引用 DOM 节点的方法
jQuery.each({
    parent: function(elem){return elem.parentNode;},          //引用父节点
    parents: function(elem){return jQuery.dir(elem,"parentNode");},     //引用所有父节点
    next: function(elem){return jQuery.nth(elem,2,"nextSibling");},     //引用相邻的下一
个 DOM 元素
    prev: function(elem){return jQuery.nth(elem,2,"previousSibling");},//引用相邻的上一
```

```
个 DOM 元素
    nextAll: function(elem){return jQuery.dir(elem,"nextSibling");},//引用所有后继 DOM
元素
    prevAll: function(elem){return jQuery.dir(elem,"previousSibling");},//引用所有前继
DOM 元素
    sibling: function(elem){return jQuery.sibling(elem.parentNode.firstChild,elem);},
//引用相邻 DOM 元素
    children: function(elem){return jQuery.sibling(elem.firstChild);}, //引用所有子元素
    //如果存在 iframe，则就是文档，或者所有子节点
    // elem.contentDocument|| elem.contentWindow.document iframe 的属性
    //http://developer.mozilla.org/en/docs/XUL:iframe
    contents: function(elem){return jQuery.nodeName(elem,"iframe")?elem.contentDocument||elem.
contentWindow.document:jQuery.makeArray(elem.childNodes);}
}, function(name, fn){
    //为 jQuery 对象注册同名方法
    jQuery.fn[ name ] = function( selector ) {
        //每个元素都执行同名方法
        var ret = jQuery.map( this, fn );
        //过滤元素集
        if ( selector && typeof selector == "string" )
            ret = jQuery.multiFilter( selector, ret );
        //返回构建的 jQuery 对象
        return this.pushStack( jQuery.unique( ret ), name, selector );
    };
});
```

在上面的代码中，为 jQuery 对象绑定了一组方法，这些方法能够引用不同位置的节点，主要包括父节点、子节点和兄弟节点三类。

- 对于父节点引用来说，可以引用当前 DOM 元素的父亲节点，也可以获得所有父级节点，包括祖先节点。
- 对于子节点引用来说，就是可以引用所有直接的子节点，但不包括不相邻的后代节点。
- 对于兄弟节点引用来说，可以引用包括当前 DOM 元素的前或后相邻节点，也包括前后相近的所有元素。

jQuery 通过 jQuery.each()公共函数把这个节点引用的方法注册到 jQuery.fn 原型对象中，即 jQuery 对象的同名方法中。因为 jQuery 对象的 DOM 元素是一个集合，所以就必须对集合中每个 DOM 元素执行相同的操作，也就是说为每个 DOM 元素调用属性包含的函数，如 parent: function(elem){return elem.parentNode;},中的 function(elem){return elem.parentNode;}。

当然，在引用的节点中，还包括很多重复的 DOM 元素，或者用户需要过滤的其他节点，这些操作都需要利用过滤函数进行过滤，关于这个话题将在 CSS 选择器一节中进行详细的讲解。

在上面定义的 jQuery 对象方法中，jQuery 也提供了几个公共函数：jQuery.dir()、

jQuery.nth()和 jQuery.sibling()来辅助完成引用 DOM 节点。下面我们来分析这几个公共函数的用法。

```
//从一个元素出发，检索某个方向上的所有元素
//如可以把元素的 parentNode、nextSibling、previousSibling、lastChild、firstChild 属性作为方向
//参数说明：
// elem 参数表示起点元素
// dir 参数表示元素的方向属性，如 parentNode、nextSibling、previousSibling、lastChild、firstChild
jQuery.dir = function( elem, dir ){
    var matched = [], cur = elem[dir];
    //逐级迭代访问，直到访问到 document 节点
    while ( cur && cur != document ) {
        if ( cur.nodeType == 1 )
            matched.push( cur );
        cur = cur[dir];    //向上一级传递节点
    }
    return matched;
};
```

dir 是 direction(方向)一词的缩写，表示朝一个方向一直迭代到尽头。例如，parentNode 能够把父节点作为当前节点，再取其父节点，通过这种方式可以迭代到 document 对象为止。另外，对于 nextSibling、previousSibling、lastChild 和 firstChild 元素属性都具有方向性，因此只要获取元素具有 dir(方向)特性的属性，就可以反复迭代读取。每循环一次都会把取到的元素保存起来。

所以说，dir()函数对于检索 DOM 文档树中呈放射线性排列的元素来说，是非常有用的。但是如果要检索在某个方向上的第几个元素，如检索偶数序号位置的元素，就需要使用 nth()函数。该函数的源代码如下所示。

```
//从一个元素出发，检索某个方向上的第几个元素。参数 Result 是第几个
//参数说明：
// cur 参数表示起点元素
// dir 参数表示元素的方向属性，如 parentNode、nextSibling、previousSibling、lastChild 和 firstChild
// result 参数表示级数，默认值为 1
// elem 参数是一个无用参数
jQuery.nth = function(cur, result, dir, elem){
    result = result || 1;
    var num = 0;
    for ( ; cur; cur = cur[dir] )
        if ( cur.nodeType == 1 && ++num == result )
            break;
    return cur;
};
```

jQuery.nth()函数与 jQuery.dir()函数在设计思路上是完全相同的，但是 jQuery.dir()函数不包含自身，而 jQuery.nth()函数可以包含自身，如果 jQuery.nth()函数的参数 result 等于 1，则返回自身元素。如果没有找到元素则返回空，如 undifined 或 null。

jQuery.sibling()函数就比较简单，但是没有上面两个方法有用。它实现了从一个元素(包括自身)检索所有后续相邻元素，然后再从这个后续的相邻元素排除一个指定元素。例如：

```
//从包含参数 n 的元素开始检索所有后续相邻元素，但不包含参数 elem 指定的元素
//参数说明：
// n 参数表示起点元素
// elem 参数排除元素
jQuery.sibling = function(n, elem){
    var r = [];
    for ( ; n; n = n.nextSibling ) {
        if ( n.nodeType == 1 && n != elem )
            r.push( n );
    }
    return r;
};
```

2.4 解析 jQuery 选择器引擎 Sizzle

jQuery 从 1.3 版本开始，使用了新的选择器引擎 Sizzle(官方网址 http://sizzlejs.com)。Sizzle 是 jQuery 作者 John Resig 开发的 DOM 选择器引擎(Dom Selector Engine)，速度号称业界第一。而且它有一个重要的特点就是 Sizzle 是完全独立于 jQuery 的，如果用户不想用 jQuery，还可以只用 Sizzle。

Sizzle 选择器引擎目前成为 jQuery 框架默认的选择器引擎，相比原来的 jQuery 引擎，速度有很大的提升，如图 2.3 所示的各种选择器执行效率的对比。

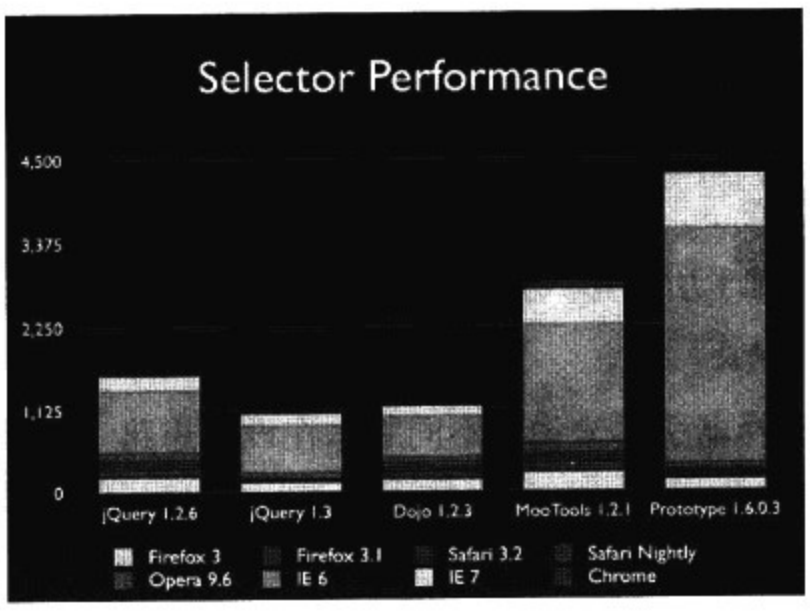


图 2.3 各种选择器执行效率的对比

2.4.1 回顾 CSS 的选择器

在解析 jQuery 选择器引擎 Sizzle 之前，我们不妨回顾一下 CSS 的选择器(CSS selector)。CSS 选择器可以分为三种基本类型：ID 选择器(#id)、Class 选择器(.class)和类型(type)选择器(p)。

另外，CSS 还支持高级选择器，如属性选择器(attribute)、伪类或伪对象选择器(Pseudo Classes)等。这些都是单一的选择器，可以在应用中把它们组合起来，形成组合选择器，如 div#id, div:last-child。组合型选择器又包括多种关系形式，如包含关系、并列关系、相邻关系和父子关系等。有关 CSS 选择器更详细的说明请参阅第 3 章的讲解。

2.4.2 解析 jQuery 选择器引擎的设计思路

尽管 jQuery 选择器引擎 Sizzle 非常复杂，功能也非常强大，但是它们都是建立在 JavaScript 已定义的方法或属性基础上来实现的。这主要包括元素的 getElementsByTagName()和 getElementById()方法，以及元素的 childNodes、firstChild、lastChild、nextSibling、parentNode 和 previousSibling 属性。借助这些方法和属性可以直接或间接地选择相匹配的 DOM 元素。

为了方便讲解，我们结合一个选择器进行说明。选择器代码如下所示。

```
$("div.red");
```

这是一个复合选择器，一般读者可以这样分析：在 DOM 文档树中找到 class 属性等于"red"的 div 元素。即一步到位，直接从 DOM 文档树中选择所需要的元素。实际上，如果根据选择器引擎的工作方式，可以把这个字符串拆分成两步走。

- 第一步，根据 document.getElementsByTagName()方法选择文档树中的 div 元素集合。
- 第二步，根据其 class 是否等于"red"进行判断，把不等于该值的元素从结果集中去掉。

不仅是这个选择器，对于所有形式的复合选择器，都可以根据这种思路来拆分选择器，然后分别完成每一部分的操作。

不过，对于 jQuery 选择器引擎来说，不同版本在设计思路上也存在一些细微的区别。例如，针对下面的选择器：

```
$("div  p");
```

早期的 jQuery 选择器是根据下面的步骤进行解析的。

- 第一步，根据 document.getElementsByTagName()方法选择文档树中的 div 元素集合。
- 第二步，迭代 div 元素集合，在所有的 div 元素中查找每个 div 元素下的 p 元素。
- 第三步，合并结果。

而 Sizzle 选择器适当调整了解析的顺序。

- 第一步，根据 document.getElementsByTagName()方法选择文档树中的 p 元素集合。
- 第二步，迭代 p 元素集合，在所有的 p 元素中查找每个 p 元素的父级元素。
- 第三步，检测父级元素。如果不是 div 元素，则遍历上一级元素；如果迭代到文档树的顶层，则排除该 p 元素；如果是 div 元素，则保存该 p 元素。

经过比较可以看到，Sizzle 选择器放弃了合并结果操作，直接在遍历过程中过滤元素，所以导致查找速度大幅提升。

初步把握了 jQuery 选择器的基本工作原理，那么我们就可以了解 jQuery 在匹配元素时是如何工作的。例如，对于$("div[id=value]");选择器，可以知道 JavaScript 先匹配 div 元素，然后再判断 div 元素的 id 属性是否等于 value，如果不等于就从结果集中删除掉。而对于$("div#value)");选择器也是一样，可以看到 div#value 与 div[id=value]是完全相同的操作。同样，div.class 也可以转换为属性选择符进行操作。

2.4.3 选择器和过滤器

根据上面的分析，我们可以把 CSS 选择器拆分为两大组成部分，或者说可以把选择器分为以下两类。

- 第一类，选择器(selector)。即根据给定的选择符，从 DOM 文档树找到相关的元素节点，并存储到结果集中。
- 第二类，过滤器(filter)。根据表达式的条件，在结果集中过滤元素。

于是，这里就有一个技术难题：哪些选择符适合选择器(selector)，哪些选择符适合过滤器(filter)?

由于可以直接使用 document.getElementsByTagName()方法进行选择，因此对于类型选择器来说，直接使用选择即可。

类型选择器相互之间还可以组成各种形式的复合选择器。例如：

*

E F

E~F

E+F

E>F

E/F

E

虽然这些特殊形式的类型选择器由多个选择器构成，但是它们都可以从文档中直接选择，故我们可以统一把它们归为选择器类型。而对于 ID 选择器和 Class 选择器来说：

- 如果它们在 selector 字符串的起始位置，那么它们也可以完成选择功能。例如 $("#id");、$(".calss");。
- 如果它们不在起始位置，那么就应该作为筛选器实现。例如$("div#id");、$("div.calss");。

实际上，由于 ID 选择器可以直接调用 JavaScript 的 document.getElementByID()方法进行选择。但是对于 Class 选择器来说，由于它无法直接进行选择，因此，我们可以把 ID 选择器视为选择器，而把 Class 选择器视为过滤器。

对于 Class 选择器来说，还可以把它转换为“*.class”形式。其中的“*”表示先选取范围内所有的元素，然后再进行过滤，这样就可以实现统一的编程接口。

对于属性选择器、伪类选择器来说，它们只能够附在其他选择器后面使用，如果单独使用，则可以通过在前面添加*标签，以便设计成统一的语法格式，以方便选择器引擎的工作。它们与 Class 选择器的工作方式是相同的，即都视为过滤器。

例如，对于下面这个复杂的选择器来说，包含了类型选择器、Class 选择器、ID 选择器、属性选择器和伪类选择器。

```
$("div.red:nth-child(odd)[title=bar]#wrap p");
```

jQuery 解析的步骤如下。

第一步，选择 DOM 文档树中所有的 p 元素，建立初步结果集。

第二步，在结果集中，选择父级元素为 div 的元素，形成新的结果集。

第三步，在新的结果集中筛选 class 属性为 red 的元素。

第四步，解析伪类选择器:nth-child(odd)，在结果集中筛选元素的子元素为偶数的元素。

第五步，解析属性选择器 [title=bar]，在结果集中筛选元素的 title 属性为 bar 的元素。

第六步，在结果集中筛选 id 等于 wrap 的元素。

2.4.4 Sizzle 引擎结构

jQuery 的 CSS 选择器引擎 Sizzle 共有 1000 多行代码，占据了 jQuery 框架四分之一的份额。这些代码被直接调用的匿名函数封装在一个独立的空间中，外界是无法访问的。通过下

面代码可以把引擎接口传递给 jQuery 空间下的四个公共函数。

```
jQuery.find = Sizzle;
jQuery.filter = Sizzle.filter;
jQuery.expr = Sizzle.selectors;
jQuery.expr[":"] = jQuery.expr.filters;
```

Sizzle 引擎在 jQuery 框架中的位置犹如咽喉，起到了核心作用，如图 2.4 所示。在下面的 jQuery 选择器逻辑流程图中，首先，对传入的选择符参数进行过滤，只有是表达式字符串时，才会进入 jQuery.fn.find()入口，然后进入 Sizzle 接口(jQuery.find())，在 Sizzle 构造器中分别调用 Sizzle.find()和 Sizzle.filter()函数完成选择和过滤操作。

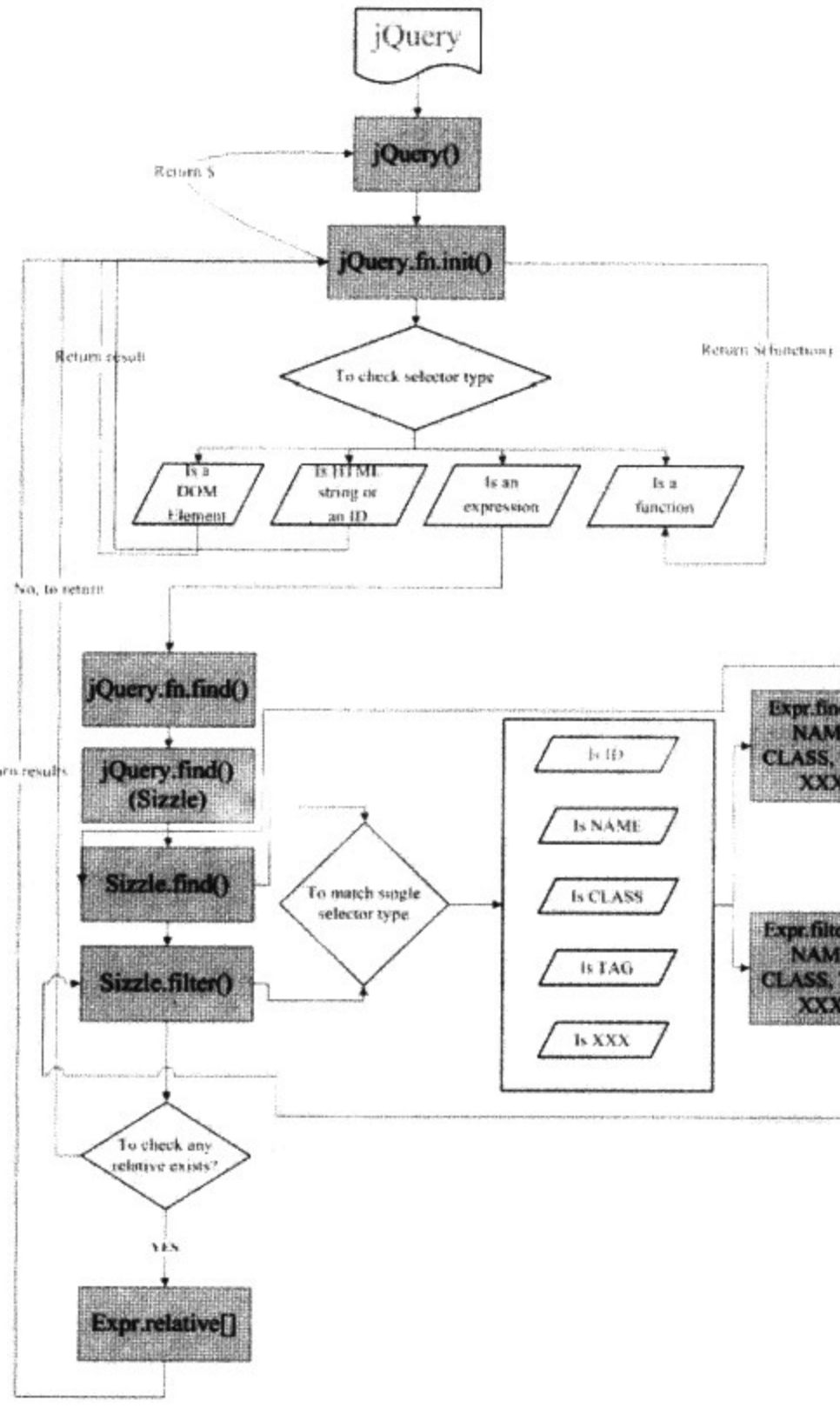


图 2.4 Sizzle 引擎在 jQuery 框架中的工作流程图

在选择(Sizzle.find())和过滤(Sizzle.filter())操作过程中，都会经过这样的处理流程：匹配简单的选择器类型(To match singleselector type)。

```
ID选择器
Name
Class选择器
类型选择器
……
```

对于选择器(Sizzle.find())来说，它会调用 Expr.find[type]，而对于过滤器(Sizzle.filter())来说，它会调用 Expr.filter[type]，最后分别返回 Sizzle.find()和 Sizzle.filter()。

在 Sizzle.filter()过程中，它还需要检测关系选择符是否存在(To check any relative exists?)。如果存在，则调用 Expr. relative[]，最后返回结果集。

下面对 Sizzle 引擎的主体结构进行简单的分析。

```
/*!
* Sizzle CSS Selector Engine - v0.9.3
 *  Copyright 2009, The Dojo Foundation
 *  Released under the MIT, BSD, and GPL Licenses.
 *  More information: http://sizzlejs.com/
 */
//把 Sizzle 引擎封装在一个独立的空间中
(function(){
//定义用于块识别器的正则表达式
var chunker = /((?:\((?:\([^()]+\)|[^()]+)+\)|\[(?:\[[^\]]*\]|['"][^'"]*['"]|[^\]'"]
+)+\]|\\.|[^ >+~,(\[\\]+)+|[>+~])(\s*,\s*)?/g,
    done = 0,
    toString = Object.prototype.toString;
// Sizzle 选择器引擎构造器函数
//参数说明:
// selector: 选择器字符串
// context: 上下文
// results: 结果集
// seed: 种子
var Sizzle = function(selector, context, results, seed) {
    //省略的函数体
};
// Sizzle 匹配函数
//参数说明:
// expr: 匹配表达式
// set: 条件设置选项
Sizzle.matches = function(expr, set){
    return Sizzle(expr, null, null, set);
};
// Sizzle 查询函数
//参数说明:
// expr: 查询表达式
// context: 上下文
```

```
// isXML: 检测函数
Sizzle.find = function(expr, context, isXML){
    //省略的函数体
};
// Sizzle 过滤函数
//参数说明:
// expr: 过滤表达式
// set: 条件设置选项
// inplace: 包含项
// not: 排除项
Sizzle.filter = function(expr, set, inplace, not){
    //省略的函数体
};
// Sizzle 表达式对象
//列举所用的各种匹配表达式
var Expr = Sizzle.selectors = {
    //省略成员属性
};
//省略其他辅助性工具函数和逻辑代码暴露的接口
jQuery.find = Sizzle;
jQuery.filter = Sizzle.filter;
jQuery.expr = Sizzle.selectors;
jQuery.expr[":"] = jQuery.expr.filters;
//定义的公共函数
Sizzle.selectors.filters.hidden = function(elem){};
Sizzle.selectors.filters.visible = function(elem){};
Sizzle.selectors.filters.animated = function(elem){};
jQuery.multiFilter = function( expr, elems, not ) {};
jQuery.dir = function( elem, dir ){};
jQuery.nth = function(cur, result, dir, elem){};
jQuery.sibling = function(n, elem){};
return;
window.Sizzle = Sizzle;
})();
```

通过上面的结构分析，可以看到 Sizzle 引擎主要包含一个构造器 Sizzle()，三个核心功能函数 matches()、find()和 filter()，以及一个表达式对象 selectors。下面分别对它们进行讲解。

2.4.5 Sizzle 构造器

在 jquery.fn.init()构造器函数中，通过调用 jQuery(context).find(selector)函数来解析并匹配 DOM 元素。jQuery.find()函数实际上是引用 Sizzle()函数，而 Sizzle()函数仅是 Sizzle 引擎的构造器，它主要调用 Sizzle.find()函数在 DOM 文档树中查找与 CSS 语法相匹配 DOM 的元素节点的集合。jQuery 名字中 Query 的意义就体现在这里。下面我们来分析一下 Sizzle 构造器函数。该函数是整个 Sizzle 引擎的入口。

```
var chunker = /((?:\((?:\([^()]+\)|[^()]+)+\)|\[(?:\[[^[\]]*\]|['"][^'"]*['"]|[^[\]'"]+)
```

```
+\]|\\.|[^ >+~,(\[\\]+)+|[>+~])(\s*,\s*)?/g,
    done = 0,
    toString = Object.prototype.toString;
/*
上边这个正则表达式比较复杂，通过逐层分解，我们可以看到它的作用
首先，可以把整个正则表达式切分为两大块
第一部分：((?:\((?:\([^()]+\)|[^()]+)+\)|\[(?:\[[^\[\]]*\]|[^\[\]]+)+\]|\\.|[^ >+~,
(\[]+)+|[>+~])
第二部分：(\s*,\s*)?
第一部分可以以|(表示或的意思)为分界符做进一步的切分。则切分后的正则表达式如下所示
/    //正则表达式起始符
 (    //第一个子表达式
    (?:\((?:\([^()]+\)    //匹配伪类选择符
    |[^()]+)+\)    //匹配函数式选择符
    |\[(?:\[[^\[\]]*\]    //匹配属性选择符
    |['"][^'"]*['"]    //匹配属性选择符
    |[^\[\]'"]+)+\]    //匹配属性选择符
    |\\.    //匹配 Class 选择符
    |[^ >+~,(\[\\]+)+    //匹配关系选择符
    |[>+~]    //匹配关系选择符
 )
 (\s*,\s*)    //匹配选择符组中间的分隔符
 ?    //非贪婪模式匹配
/    //正则表达式终止符
g    //正则表达式属性，全局匹配
/*
// Sizzle 选择器引擎构造器函数
//参数说明：
// selector：选择器字符串
// context：上下文
// results：结果集
// seed：种子
var Sizzle = function(selector, context, results, seed) {
    results = results || [];    //设置默认结果集为空
    context = context || document;    //设置上下文为 document 对象
    //如果上下文不是元素和文档对象，则返回空集合
    if ( context.nodeType !== 1 && context.nodeType !== 9 )
        return [];
    //如果选择器参数不存在，或者不为字符串类型，则返回默认结果集
    if ( !selector || typeof selector !== "string" ) {
        return results;
    }
    //初始化变量
    var parts = [], m, set, checkSet, check, mode, extra, prune = true;
    // 重置 chunker 正则表达式匹配的起始位置为开始位置
    chunker.lastIndex = 0;
    //匹配选择符字符串，如果存在则执行下面操作
    while ( (m = chunker.exec(selector)) !== null ) {
        parts.push( m[1] );    //存储匹配的字符串信息
        //如果是选择符组，则获取后半部分
        if ( m[2] ) {
```

```
            extra = RegExp.rightContext;
            break;
        }
    }
    //Expr 和 Sizzle.selector 指向同一个对象，这个选择器对象与老版本不一样，
    //放弃了 if else 语句判断，全部使用正则表达式进行处理，执行效率提高了很多，
    //具体代码参阅 Expr 对象。
    //POS:正则表达式/:(nth|eq|gt|lt|first|last|even|odd)(?:\((\d*)\))?(?=[^-]|$)/,
    //下面代码判断选择符字符串中是否存在特殊函数
    if ( parts.length > 1 && origPOS.exec( selector ) ) {
        //relative 是函数对象，封装了+>~函数，表示 typeof Expr.relative[ parts[0] ] ==Function
        if ( parts.length === 2 && Expr.relative[ parts[0] ] ) {
            set = posProcess( parts[0] + parts[1], context );
        } else {
            set = Expr.relative[ parts[0] ] ?
                [ context ] :
                Sizzle( parts.shift(), context );
            //如果匹配到>+~，过滤上下文，去掉选择器中符号，再分析选择器
            while ( parts.length ) {
                selector = parts.shift();
                if ( Expr.relative[ selector ] )
                    selector += parts.shift();
                set = posProcess( selector, set );
            }
        }
    } else {
        //执行种子操作
        var ret = seed ?
            { expr: parts.pop(), set: makeArray(seed) } :
            Sizzle.find( parts.pop(), parts.length === 1 && context.parentNode ?
context.parentNode : context, isXML(context) );
        set = Sizzle.filter( ret.expr, ret.set );
        if ( parts.length > 0 ) {
            checkSet = makeArray(set);
        } else {
            prune = false;
        }
        while ( parts.length ) {
            var cur = parts.pop(), pop = cur;
            if ( !Expr.relative[ cur ] ) {
                cur = "";
            } else {
                pop = parts.pop();
            }
            if ( pop == null ) {
                pop = context;
            }
            Expr.relative[ cur ]( checkSet, pop, isXML(context) );
        }
    }
    //设置默认值
```

```
    if ( !checkSet ) {
        checkSet = set;
    }
    //抛出异常
    if ( !checkSet ) {
        throw "Syntax error, unrecognized expression: " + (cur || selector);
    }
    //如果检测到选项 checkSet 是数组，则执行下面操作
    if ( toString.call(checkSet) === "[object Array]" ) {
        //如果 checkSet 没有包含 set 选项值
        if ( !prune ) {
            //则把 checkSet 推入到结果集中
            results.push.apply( results, checkSet );
        } else if ( context.nodeType === 1 ) {    //如果上下文是元素类型的对象
            for ( var i = 0; checkSet[i] != null; i++ ) {    //则遍历检测选项集，然后把 set
推入结果集中
                if ( checkSet[i] && (checkSet[i] === true || checkSet[i].nodeType === 1
&& contains(context, checkSet[i])) ) {
                    results.push( set[i] );
                }
            }
        } else {
            //则遍历检测选项集，然后把 set 推入结果集中
            for ( var i = 0; checkSet[i] != null; i++ ) {
                if ( checkSet[i] && checkSet[i].nodeType === 1 ) {
                    results.push( set[i] );
                }
            }
        }
    } else {
        //把结果集与检测选项组合并为数组
        makeArray( checkSet, results );
    }
    //处理组选择器中下一个选择器
    if ( extra ) {
        //把后半部分选择器字符串再次传递给构造器，执行下一次匹配处理
        Sizzle( extra, context, results, seed );
        //如果存在排序函数，则调用它对结果集进行排序
        if ( sortOrder ) {
            hasDuplicate = false;    //指示已经排序
            results.sort(sortOrder);    //排序结果集
            //如果没有排序，则对结果集进行重排
            if ( hasDuplicate ) {
                for ( var i = 1; i < results.length; i++ ) {
                    if ( results[i] === results[i-1] ) {
                        results.splice(i--, 1);
                    }
                }
            }
        }
    }
```

```
    //返回匹配的结果集
    return results;
};
```

2.4.6 Sizzle 选择器

在 jQuery 构造器一节中，我们介绍了 init()构造函数处理选择器字符串的第四种情况，截取代码如下所示。

```
} else
    //第四种情况，处理 jQuery(expression, [context]),
    //例如，$("div .red")的表达式字符串
    return jQuery( context ).find( selector );
```

在这里，jQuery 调用了 jQuery 对象的 find()方法来处理选择器字符串。find()方法的代码如下所示。

```
jQuery.fn = jQuery.prototype = {
    find: function( selector ) {
        //当表达式不包含“,”符号时
        if ( this.length === 1 ) {
            var ret = this.pushStack( [], "find", selector );
            ret.length = 0;
            jQuery.find( selector, this[0], ret );
            return ret;
        //当表达式包含“,”符号时
        } else {
            return this.pushStack( jQuery.unique(jQuery.map(this, function(elem){
                return jQuery.find( selector, elem );
            })), "find", selector );
        }
    }
}
```

首先，我们来分析一下 pushStack()方法。

```
jQuery.fn = jQuery.prototype = {
    // 获取多个元素的数组，并把它推入到堆栈中
    // 返回新的匹配元素的数据集合
    pushStack: function( elems, name, selector ) {
        //新建一个 jQuery 对象结构的匹配元素数据集合
        var ret = jQuery( elems );
        //将上个对象的引用推入栈中
        ret.prevObject = this;
        ret.context = this.context;
        //当关键字为 find 时，在原有 selector 的基础上，继续增加 selector
        //例如，$("div").find("p") 意思就是 $("div p")
        if ( name === "find" )
            ret.selector = this.selector + (this.selector ? " " : "") + selector;
```

```
        else if ( name )
            ret.selector = this.selector + "." + name + "(" + selector + ")";
        // 返回最新的 jQuery 对象
        return ret;
    }
}
```

注意：ret.prevObject = this;这个方法在$().andSelf()和$().end()中调用，对于筛选或查找后的元素，返回前一次元素状态是非常有用的。

接着，就调用 jQuery.find(selector, this[0], ret);。而 jQuery.find()函数是引用 Sizzle 对象的，首先看下面这几行代码。

```
jQuery.find = Sizzle;
jQuery.filter = Sizzle.filter;
jQuery.expr = Sizzle.selectors;
jQuery.expr[":"] = jQuery.expr.filters;
//……
return;
window.Sizzle = Sizzle;
```

在这里应用了 JavaScript 设计模式中的适配器模式，jQuery.find 调用的是 Sizzle 对象。Sizzle 对象的代码可以参阅上一节说明，其中有如下一段，它调用了 Sizzle.find()函数。

```
var ret = seed ?
    { expr: parts.pop(), set: makeArray(seed) } :
    Sizzle.find(  parts.pop(),  parts.length  ===  1  &&  context.parentNode  ?
context.parentNode : context, isXML(context) );
```

Sizzle.find 函数将对表达式字符串进行解析，最后返回一个 jQuery 对象。下面是 Sizzle.find()函数的详细代码，在这个函数中将选择匹配的元素。

```
Sizzle.find = function(expr, context, isXML){
    var set, match;
    //如果没有表达式，则返回空数组
    if ( !expr ) {
        return [];
    }
    //遍历 Expr.order 数组，匹配基本选择器类型
    // var Expr = Sizzle.selectors = {
    // order: [ "ID", "NAME", "TAG" ] }
    //这里将把复合选择器的字符串从左到右取最小单元的选择符进行分析操作
    //最小单元如#id、~F(+F,>F)、.class、[id='xx']、F、:last()等
    //分析操作完之后将分析过的字符串部分给删除，
    //然后循环分析接下来的剩余的部分，直到字符串为空
    for ( var i = 0, l = Expr.order.length; i < l; i++ ) {
        //获取基本选择器类型
        var type = Expr.order[i], match;
```

```
        //调用对应的正则表达式，匹配选择符表达式字符串
        //var Expr = Sizzle.selectors = {
        //    match: {
        //        ID: /#((?:[\w\u00c0-\uFFFF_-]|\\.)+)/,
        //        CLASS: /\.((?:[\w\u00c0-\uFFFF_-]|\\.)+)/,
        //        NAME: /\[name=['"]*((?:[\w\u00c0-\uFFFF_-]|\\.)+)['"]*\]/,
        //            ATTR: /\[\s*((?:[\w\u00c0-\uFFFF_-]|\\.)+)\s*(?:(\S?=)\s*(['"]*)
(.*?)\3|)\s*\]/,
        //        TAG: /^((?:[\w\u00c0-\uFFFF\*_-]|\\.)+)/,
        //        CHILD: /:(only|nth|last|first)-child(?:\((even|odd|[\dn+-]*)\))?/,
        //        POS: /:(nth|eq|gt|lt|first|last|even|odd)(?:\((\d*)\))?(?=[^-]|$)/,
        //          PSEUDO: /:((?:[\w\u00c0-\uFFFF_-]|\\.)+)(?:\((['"]*)((?:\([^\)]+\)
|[^\2\(\)]*)+)\2\))?/
        //    }
        //如果匹配到字符串，则进行处理
        if ( (match = Expr.match[ type ].exec( expr )) ) {
            //获取匹配位置左侧的字符串
            var left = RegExp.leftContext;
            //如果当前匹配字符前面不是反斜杠，则进行处理
            if ( left.substr( left.length - 1 ) !== "\\" ) {
                //清除双反斜杠
                match[1] = (match[1] || "").replace(/\\/g, "");
                //调用 Expr.find[ type ]()函数，选择匹配元素
                set = Expr.find[ type ]( match, context, isXML );
                //如果匹配成功，则删除选择符表达式中的当前选择符
                if ( set != null ) {
                    expr = expr.replace( Expr.match[ type ], "" );
                    break;
                }
            }
        }
    }
    //如果不存在结果集，则获取所有元素
    if ( !set ) {
        set = context.getElementsByTagName("*");
    }
    //返回结果集和表达式
    return {set: set, expr: expr};
};
```

2.4.7 Sizzle 过滤器

Sizzle 过滤器主要包含两部分：第一部分是过滤函数(jQuery.filter())，在该函数中将对需要过滤的表达式及其对应的表达式处理函数执行分析，并返回过滤后的 jQuery 对象；第二部分就是过滤表达式对象(Expr = Sizzle.selectors)，该对象包含了所有表达式处理的方法和匹配的正则表达式。

Sizzle 选择器先初步分析参数字符串，找出基本特征字符串，并根据这些特征字符在 Expr

对象中调用对应的正则表达式，然后进一步分析选择符字符串，同时根据进一步分解出来的特殊字符，在 Expr 对象中匹配到对应的选择器函数，最后根据这个选择器函数的返回值是否为 true，决定是否保留当前过滤的元素。被保留的元素就是筛选元素，它将被推进 jQuery 对象集合中。

jQuery.filter()函数完成分析属性([])、Pseudo(:)、Class (.)和 ID(#)的筛选功能，从给定的集合中筛选出满足上面四种筛选表达式的集合。针对 find()方法，jQuery.filter()函数的完成并不表明整个选择符的分析完成了，如果单独使用这个函数，表达式中就不应该含有查找的选择器表达式。筛选是根据[、:、#和.这四个符号来作为筛选器的分隔符的。Class 筛选器通过 classFilter 来完成，它还把 Pseudo 中的:not、:nth-child 单独从 Pseudo 类中提出来处理。对于[的属性筛选器，实现起来也很简单。除去这些，它还调用 jQuery.expr[m[1]];来处理 Pseudo 类。

Sizzle 过滤器的实现代码如下所示。

```
Sizzle.filter = function(expr, set, inplace, not){
    var old = expr, result = [], curLoop = set, match, anyFound,
        isXMLFilter = set && set[0] && isXML(set[0]);
    while ( expr && set.length ) {
        //Expr.filter 包含元素[CHILD,PSEUDO,ID,TAG,CLASS,ATTR,POS]，元素类型是 Function
        //match: {
        //    ID: /#((?:[\w\u00c0-\uFFFF_-]|\\.)+)/,
        //    CLASS: /\.((?:[\w\u00c0-\uFFFF_-]|\\.)+)/,
        //    NAME: /\[name=['"]*((?:[\w\u00c0-\uFFFF_-]|\\.)+)['"]*\]/,
        //    ATTR: /\[\s*((?:[\w\u00c0-\uFFFF_-]|\\.)+)\s*(?:(\S?=)\s*(['"]*) (.*?)\3|)\s*\]/,
        //    TAG: /^((?:[\w\u00c0-\uFFFF\*_-]|\\.)+)/,
        //    CHILD: /:(only|nth|last|first)-child(?:\((even|odd|[\dn+-]*)\))?/,
        //    POS: /:(nth|eq|gt|lt|first|last|even|odd)(?:\((\d*)\))?(?=[^-]|$)/,
        //    PSEUDO: /:((?:[\w\u00c0-\uFFFF_-]|\\.)+)(?:\((['"]*)((?:\([^\)]+\)|[^\2\
(\)]*)+)\2\))?/
        //}
        for ( var type in Expr.filter ) {
            //这里先判断下过滤器是不是符合 jQuery 选择器的语法
            //把正则匹配的结果赋值给 match 变量
 //当 type 为 PSEUDO 和 POS 时，进入该代码块
            if ( (match = Expr.match[ type ].exec( expr )) != null ) {
                var filter = Expr.filter[ type ], found, item;
                anyFound = false;
                if ( curLoop == result ) {
                    result = [];
                }
                //Expr.preFilter 和 Expr.filter 定义方法类似，只是元素的方法不一样
                //主要功能就是做过滤之前的过滤器处理，例如：
                //ID: function(match){
                //    return match[1].replace(/\\/g, "");
                //},
                //就是去掉\,
```

```
            //TAG: function(match, curLoop){
            //    for ( var i = 0; !curLoop[i]; i++ ){}
            //    return isXML(curLoop[i]) ? match[1] : match[1].toUpperCase();
            //},
            //判断标签是不是 XML 对象，如果不是返回标签大写
            if ( Expr.preFilter[ type ] ) {
                //进一步处理正则匹配的结果，由于 js 没有方法重载的概念，
                //虽然有些元素可能没有 5 个参数，js 编译器会自动忽略，而不会出错
                //例如，在 C 语言中会报错，js 不会
                //用到 5 个参数的只有 CLASS 过滤和 PSEUDO
                match = Expr.preFilter[ type ]( match, curLoop, inplace, result, not,
isXMLFilter );
                if ( !match ) {
                    anyFound = found = true;
                } else if ( match === true ) {
                    continue;
                }
            }
            //如果存在匹配
            if ( match ) {
                for ( var i = 0; (item = curLoop[i]) != null; i++ ) {
                    if ( item ) {
                        found = filter( item, match, i, curLoop );
                        var pass = not ^ !!found;
                        if ( inplace && found != null ) {
                            if ( pass ) {
                                anyFound = true;
                            } else {
                                curLoop[i] = false;
                            }
                        } else if ( pass ) {
                            result.push( item );
                            anyFound = true;
                        }
                    }
                }
            }
            //如果匹配到元素
            if ( found !== undefined ) {
                //如果没有指定上下文
                if ( !inplace ) {
                    curLoop = result;
                }
                expr = expr.replace( Expr.match[ type ], "" ); //清除空格
                //如果不匹配，则返回空数组
                if ( !anyFound ) {
                    return [];
                }
                break;
            }
        }
```

```
            }
            //处理不正确的表达式
            if ( expr == old ) {
                if ( anyFound == null ) {
                    throw "Syntax error, unrecognized expression: " + expr;
                } else {
                    break;
                }
            }
            //把表达式传递给 old 存储
            old = expr;
        }
        //返回过滤的集合
        return curLoop;
    };
```

2.4.8 jQuery 选择器应用优化

正确使用选择器引擎对于提高页面性能起了至关重要的作用。使用合适的选择器表达式可以提高性能、增强语义并简化逻辑。在传统用法中，最常用的简单选择器包括 ID 选择器、Class 选择器和类型标签选择器。其中 ID 选择器是速度最快的，这主要是因为它使用 JavaScript 的内置函数 getElementById()；其次是类型选择器，因为它使用 JavaScript 的内置函数 getElementsByTag()；速度最慢的是 Class 选择器，其需要通过解析 HTML 文档树，并且需要在浏览器内核外递归，这种递归遍历是无法被优化的。

就需求分析，CSS 的选择器是为了通过语义来渲染样式，而 jQuery 的选择器只是为了选出一类 DOMElement，执行逻辑操作。但是，在实际开发中，Class 选择器是使用频率最高的类型之一(如表 2.1 所示)。

表 2.1 jQuery 选择器使用频率列表

选 择 器	统计频率
#id	51.290%
.class	13.082%
tag	6.416%
tag.class	3.978%
#id tag	2.151%
tag#id	1.935%
#id:visible	1.577%
#id .class	1.434%
.class .class	1.183%
*	0.968%

续表

选 择 器	统计频率
#id tag.class	0.932%
#id:hidden	0.789%
tag[name=value]	0.645%
.class tag	0.573%
[name=value]	0.538%
tag tag	0.502%
#id #id	0.430%
#id tag tag	0.358%

Class 选择器在文档中使用频率靠前，这无疑会增加系统的负担，因为每使用一次 Class 选择器，整个文档就会被解析一遍，并遍历每个节点。因此，建议读者在使用 jQuery 选择器时，应该注意以下几个问题。

第一，多用 ID 选择器。

多用 ID 选择器，这是一个明智的选择。即使添加“在”ID 选择器，也可以从父级元素中添加一个 ID 选择器，这样就会缩短节点访问的路程。

第二，少直接使用 Class 选择器。

可以使用复合选择器，例如使用 tag.class 代替.class。文档的标签是有限的，但是类可以拓展标签的语义，那么大部分情况下，使用同一个类的标签也是相同的。

当然，应该摒除表达式中的冗余部分，对于不必要的复合表达式就应该进行简化。例如，对于#id2 #id1 或者 tag#id1 表达式，不妨直接使用#id1 即可，因为 ID 选择器是惟一的，执行速度最快。使用复合选择器，相反会增加负担。

第三，多用父子关系，少用嵌套关系。

例如，使用 parent>child 代替 parent child。因为">"是 child 选择器，只从子节点里匹配，不递归。而" "是后代选择器，递归匹配所有子节点及子节点的子节点，即后代节点。

第四，缓存 jQuery 对象。

如果选出结果不发生变化的话，不妨缓存 jQuery 对象，这样就可以提高系统性能。养成缓存 jQuery 对象的习惯可以让你在不经意间就能够完成主要的性能优化。

例如，下面的用法是低效的。

```
for (i = 0 ; i < 100 ; i ++ ) ... {
     var myList = $( ' .myList ' );
    myList.append(i);
}
```

而使用下面的方法先缓存 jQuery 对象，则执行效率就会大大提高。

```
var myList = $( ' .myList ' );
for (i = 0 ; i < 100 ; i ++ ) ... {
    myList.append(i);
}
```

2.5 类 数 组

类数组是 jQuery 框架的专有概念，它描述了 jQuery 对象的基本形态。我们曾经介绍过，jQuery 选择器能够匹配一个或多个 DOM 元素，并把这些元素打包到一个数据集中返回，然后提供众多操作这个数据集合的方法。那么类数组到底是一类什么数据？它的结构是什么样的？它又包含了那些基本特性？本节将就这些个问题专门介绍。

2.5.1 定义类数组

在 JavaScript 中，最高效的数据仓库就是数组了，也就是说数组是最佳数据集合结构。也许我们可以采用下面的方法让 jQuery 返回的数据都存储到数组中。

```
jQuery.fn.prototype=new Array();
```

然后，通过下面方法来实现数组继承机制。

```
jQuery.fn.prototype.constructor=jQuery
```

这样 jQuery 对象就继承了数组的所有特性，又可以在 jQuery 对象中进行数组的功能扩展。

但是，jQuery 并没有这样做，也就是说它抛弃了通过继承 Array 的途径来实现内部包含数据集合的功能，而是采用 Array-Like 对象结构来实现。

Array-Like(类数组)对象，实际上就是对象，但是它类似数组。如果从集合角度分析，对象和数组其实都是一家人，只不过一个是无序的，一个是有序的。而对于数组来说，它主要表现为带有元素下标和 length 属性。当数组元素增减时，length 属性会自动进行跟踪，反映这种变化。

例如，下面的代码是 jQuery.fn.init()构造函数中第一种实现，即当选择符为 DOM 元素时，jQuery 定义 jQuery 对象的过程。

```
if ( selector.nodeType ) {
    this[0] = selector;
    this.length = 1;
    this.context = selector;
```

```
    return this;
}
```

首先，它通过 this[0]来直接设定第一个位置的 DOM 元素，同时设定 this 的 length 属性值为 1。这里可以看到对象采用数组的 key/value(键/值)对方式存储在对象中。上面代码可以使用对象直接量表示为以下方式。

```
{
    0 : selector,
    length : 1
}
```

实际上，数组继承于对象。运算符[]与运算符{}所存储的数据都是相同的。不过，对于数组来说，它会把下标 index 作为对象属性的 key，把数组中的值作为对应的 value。

如果是多元素的 jQuery 对象，jQuery 会首先调用 jQuery.makeArray(selector)函数把集合(类数组)转换成数组。通过分析，数组和对象都可以采用 object[attr]语法格式取得 key 对应的 value。对于类数组来说，要实现下标访问对象，则必须要求其实现 length 属性，有了 length 的长度，那么就可以从 0~length-1 的 key 属性中取得对应的 value。

例如，jQuery 利用下面的函数将类数组对象转换为数组对象。类数组对象有 length 属性，其成员索引为 0～length-1。

```
//把类数组对象的元素全部推进数组对象
makeArray: function( array ) {
    var ret = [];
    //如果参数存在
    if( array != null ){
        var i = array.length;
        //单个元素，但window、string、function有length的属性，加其他的判断
        if( i == null || typeof array === "string" || jQuery.isFunction(array) ||
array.setInterval )
            ret[0] = array;
        else    //类数组的集合
            while( i )
                ret[--i] = array[i];
    }
    return ret;    //返回数组对象
}
```

下面这个函数能够把任何类数组对象转换为 jQuery 对象。

```
//把类数组对象的元素全部推进当前jQuery对象中
setArray: function( elems ) {
    //初始化长度
    this.length = 0;
    // push()方法会在原始length属性上递加其值
    Array.prototype.push.apply( this, elems );
```

```
        return this;
    }
```

当然添加元素时，会调用 Array.prototype.push 函数自动修改对象的 length 属性值。实际上，Array 的方法都可以自动改变 length 的值，从而在对象的 key/value 对中完成无序到有序或重新排序的工作。

setArray()函数只会改变当前 jQuery 对象的集合，它会清除这个对象集合中以前的元素。但是有的时候我们想保存原来集合中的元素，同时也想能就新传入的元素进行 jQuery 对象的操作。因此，jQuery 又定义了 pushStack()函数，新建一个 jQuery 对象的同时保存对原来对象的引用。这样就可以在需要时使用自己所要的对象了。

```
//把类数组对象的元素全部推进当前的 jQuery 对象
pushStack: function( elems, name, selector ) {
    //把参数 elems 封装进一个新的 jQuery 对象中，定义类数组
    var ret = jQuery( elems );
    // 把旧对象作为一个属性封装到 ret 中
    ret.prevObject = this;
    ret.context = this.context;
    if ( name === "find" )
        ret.selector = this.selector + (this.selector ? " " : "") + selector;
    else if ( name )
        ret.selector = this.selector + "." + name + "(" + selector + ")";
    //返回新的格式化的数据集
    return ret;
},
```

该函数返回的是新构建成的对象，它有着 jQuery 对象的全部功能，同时还可以通过 prevObject 来访问原来的老对象。

2.5.2 操作类数组

类数组的操作主要包括元素的定位、查找、复制和删除等。另外，还可以通过迭代器和映射器扩展对类数组的操作功能。注意，由于类数组的操作对象是集合，所以这与类数组包含的 DOM 元素操作是两个不同的概念。

1. 定位元素

jQuery 定义了 get()和 index()方法用来定位元素，它们是集合操作最基本的方法。另外，jQuery 还定义了 get(index)和 eq(index)方法，以读取指定位置的元素。get(index)方法和 eq(index)方法的主要区别如下。

- get(index)方法读取集合中的元素，它与直接通过[i]来读取元素的方法是完全相同的。

- eq(index)方法克隆集合中的元素，也就是说不修改数组元素。

get()方法的实现如下。

```
//获取 jQuery 对象的第几个 DOM 元素。无参数时表示全部的 DOM 元素
get: function( num ) {
    return num === undefined?
    // 返回全部 DOM 元素的数组
    Array.prototype.slice.call( this ):
    // 返回对应位置的 DOM 元素
    this[ num ];
}
```

eq()方法的实现如下。

```
//获取 jQuery 对象的第几个 DOM 元素。序号从 0 算起
eq : function(i) {
     //返回指定位置的元素
     return this.slice(i, +i + 1);
}
```

index()方法的实现如下。

```
//查找 elem 在 jQuery 对象的下标位置(index)
index: function( elem ) {
    return jQuery.inArray(
        //如果参数是 jQuery 对象，则判断 jQuery 参数对象中第一个元素在当前 jQuery 对象中的位置
        elem && elem.jquery ? elem[0] : elem
        , this );
}
```

在 index()方法中，调用 inArray()公共函数判断 elem 在当前类数组中的下标位置。index()方法支持的参数可以是 jQuery 对象或者 DOM 元素，而 inArray()函数的参数可以是任何类型的元素。inArray()函数的实现如下。

```
//获取指定元素在数组中的下标位置
inArray: function( elem, array ) {
    for ( var i = 0, length = array.length; i < length; i++ )
        if ( array[ i ] === elem )
            return i;
    //如果不存在指定元素，则返回-1
    return -1;
}
```

2. 复制元素

jQuery 模拟 Array 的 slice()方法也能实现元素的复制功能。其实现代码如下所示。

```
//模拟数组的 slice()方法
slice: function() {
```

```
    return this.pushStack( Array.prototype.slice.apply( this, arguments ),
        "slice", Array.prototype.slice.call(arguments).join(",") );
}
```

另外，它还模拟数组的 concat()方法定义了一个全局函数 merge()。其实现代码如下所示。

```
//模拟数组的 concat()方法
merge: function( first, second ) {
    // 因为 IE 和 Opera 浏览器会重写 length 属性，所以需要先存储 length 属性值
    var i = 0, elem, pos = first.length;
    //兼容 IE 浏览器
    if ( !jQuery.support.getAll ) {
        while ( (elem = second[ i++ ]) != null )
            if ( elem.nodeType != 8 )
                first[ pos++ ] = elem;
    } else
        while ( (elem = second[ i++ ]) != null )
            first[ pos++ ] = elem;
    return first;
},
```

3. 添加元素

jQuery 还为类数组定义了添加元素的方法，其实现代码如下所示。

```
add: function( selector ) {
    return this.pushStack( jQuery.unique( jQuery.merge(
        this.get(),
        typeof selector === "string" ?
        jQuery( selector ) :
        jQuery.makeArray( selector )
    )));
}
```

上面方法能够把与表达式匹配的元素添加到 jQuery 对象中，类数组的集合也可以被追加进来。

4. 过滤元素

使用 add()方法可以把其他元素增加到类数组中来，当然有的时候也需要过滤类数组中不需要的元素。jQuery 提供了 filter()和 not()方法来过滤元素。

filter()方法能够筛选出与指定表达式匹配的元素集合。可以通过该方法来筛选当前 jQuery 对象的元素，或者是使用逗号分隔的多个表达式。其实现代码如下。

```
filter: function( selector ) {
    return this.pushStack(
        jQuery.isFunction( selector ) &&
        jQuery.grep(this, function(elem, i){
            return selector.call( elem, i );
```

```
        }) ||
        jQuery.multiFilter( selector, jQuery.grep(this, function(elem){
            return elem.nodeType === 1;
    }) ), "filter", selector );
}
```

filter()方法是 jQuery.grep()和 jQuery.multiFilter()函数功能的综合，如果参数是函数的话，就采用 jQuery.grep()函数来完成，否则采用 jQuery.multiFilter()函数进行 selector 方式的过滤。

jQuery.grep()函数提供了以自定义函数回调的形式来过滤集合中不需要的元素，最后形成需要的数组，与 map()函数功能类似。

```
// 过滤 elems 中满足 callback 处理的所有元素，inv 参数表示相反操作
grep : function(elems, callback, inv) {
    var ret = [];
    for (var i = 0, length = elems.length;i < length; i++)
    if (!inv != !callback(elems[i], i))
        ret.push(elems[i]);
    return ret;
},
```

jQuery.multiFilter()函数与 jQuery.filter()函数的区别不大。multiFilter 支持采用符号分隔的 selector 多表达式方式。其实现代码如下。

```
jQuery.multiFilter = function( expr, elems, not ) {
    if ( not ) {
        expr = ":not(" + expr + ")";
    }
    return Sizzle.matches(expr, elems);
};
```

jQuery.multiFilter()函数可以作为筛选器，与 jQuery.filter()函数一样，selector 的多表达式也可以只是筛选器的组合，即以 .、#、:、[这四种符号做分隔的表达式。

not()方法也是根据 selector 来过滤不符合条件的元素，但是 not()方法是建立于 filter()方法基础之上的，执行效率会更高。其实现代码如下所示。

```
not: function( selector ) {
    if ( isSimple.test( selector ) )
        return this.pushStack( jQuery.multiFilter( selector, this, true ), "not",
selector );
    else
        selector = jQuery.multiFilter( selector, this );
    var isArrayLike = selector.length && selector[selector.length - 1] !== undefined
&& !selector.nodeType;
    return this.filter(function() {
        return isArrayLike ? jQuery.inArray( this, selector ) < 0 : this != selector;
    });
}
```

5. 映射元素

集合映射是非常实用的工具。jQuery 定义了 each()和 map()两个映射方法。each()方法是对集合中每个元素都执行回调函数，而 map()方法还能够收集每个回调函数的返回结果组成一个新的集合。

对于 each()方法来说，jQuery 实现代码如下所示。

```
each: function( callback, args ) {
    return jQuery.each( this, callback, args );
}
```

然后通过调用 jQuery.each()公共函数来实现元素的迭代操作。

```
//对 object 中的每个对象都执行 callback 函数
//参数 args 仅仅在内部使用
each: function( object, callback, args ) {
    var name, i = 0, length = object.length;
    if ( args ) {
        if ( length === undefined ) {
            for ( name in object )
                if ( callback.apply( object[ name ], args ) === false )
                    break;
                } else
                    for ( ; i < length; )
                        if ( callback.apply( object[ i++ ], args ) === false )
                            break;
    //不是类数组的 object，对每个属性都进行 callback 函数的调用
    } else {
        if ( length === undefined ) {
            for ( name in object )
                if ( callback.call( object[ name ], name, object[ name ] ) === false )
                    break;
            } else
                for ( var value = object[0]; i < length && callback.call( value, i,
value ) !== false; value = object[++i] ){}
    }
    return object;
}
```

这个公共函数支持第一个参数的类数组(数组)或对象。如果是数组就对每个元素进行 callback 操作；如果是对象，就对每个属性值进行 callback 操作。

callback 回调函数的语法格式如下。

```
callback:function(index,value)
```

参数 index 表示索引号，参数 value 表示数组中 index 对应的元素或对象的第 index 个处理的属性。如果使用参数 args，则 callback 回调函数的语法格式如下。

```
callback:function(args)
```

参数 args 是给回调函数设定的参数。而对于 jQuery 对象的 each()方法，它的第二个参数 args 是采用传入的 args 直接给 callback 设定参数，而不是默认集合中 index 和对应的元素，但执行的次数还是集合的 length 次。

map()方法能够将一组元素转换成其他数组，它是通过回调函数返回值组成的数组。其实现代码如下。

```
map: function( callback ) {
    return this.pushStack( jQuery.map(this, function(elem, i){
        return callback.call( elem, i, elem );
    }));
},
```

map()方法将一组元素转换成其他数组，然后根据这个数组构建新的 jQuery 对象。在该方法中，通过 jQuery.map(this, function(elem, i){}语句中 this 的 jQuery 对象集合的每个元素当作回调函数中 elem 的参数传到回调函数中。该回调函数又执行实例方法 map()中的 callback 函数。jQuery.map()通过代理的回调来取得转换而成的元素集合，然后采用 pushStack()方法把该集合的元素构建成新的 jQuery 对象并返回，同时保存原 jQuery 对象的引用。

map()公共函数的实现代码如下所示。

```
map: function( elems, callback ) {
    var ret = [];
    for ( var i = 0, length = elems.length; i < length; i++ ) {
    var value = callback( elems[ i ], i );
    if ( value != null )
        ret[ ret.length ] = value;
    }
    return ret.concat.apply( [], ret );
}
```

上面的函数能够返回对 elems 中每个元素都进行 callback 函数操作后返回值的集合。

第3章 高效选择的技巧与原理

文档操作的第一步就是要选择所控制的元素。其中，应该考虑以下两个问题。

- 是否精准选择元素？精准选择决定操作的成败，这就需要读者熟练各种选择器类型的使用。
- 是否高效选择元素？高效选择决定了代码的执行效率，不同的选择方式，会占用 JavaScript 引擎不同的计算时间，所以，应该理解不同选择器的工作原理。

本章将帮助读者了解选择器的来龙去脉，理解 jQuery 选择器不同类型的用法，以及这些用法背后的故事。本章的目的就是要让读者实现高效地选择文档元素。

3.1 选择器是什么

玩过 Photoshop 的读者，可能知道图像编辑的第一步是要选取范围，不选择的后果则是默认操作对象为当前图层的图像， Photoshop 的选择功能自然是非常强大的。

玩过 CSS 的读者，自然知道 CSS 选择器的功能很强大，欲精通 CSS，则必须先跨越选择器的关卡。学习 JavaScript 编程，也应该先学习选择文档元素。JavaScript 客户端仅提供了两个选择元素的方法(getElementById()和 getElementsByTagName())，这是不够用的，所以我们还要深入研究 JavaScript 选择器，学习 jQuery。

3.1.1 从 CSS 选择器说起

CSS 技术帮助网页理清并分离了结构和表现之间的瓜葛，我们想为页面中某个标签定义样式，就需要选择这个标签。如果知道标签的 ID 名或类名，也好办，但是在复杂的网页结构中，很多事情都是难以预料的。在 CSS 中，帮助我们解决这个棘手问题的角色就是 CSS 选择器了。

CSS 从 2.0 版本就开始不断扩张选择器的类型，到 CSS 3.0 版本时，已经定义了 44 种类型的选择器，分解如下。(更详细的信息请访问 W3C 专题页面)

http://www.w3.org/TR/CSS2/selector.html

也可以通过下面这个地址学习比较深入的 CSS 选择器用法。

http://www.456bereastreet.com/archive/200509/css_21_selectors_part_1

1. 基本选择器

基本选择器提供了选择 HTML 文档元素的基本方法，也是 CSS 最早支持的选择器类型，如表 3.1 所示。

表 3.1 基本选择器

选 择 器	说 明	示 例
E	标签选择器，选择所有 E 类型的元素	p { font-size:12px; }
.className	类选择器，选择所有 class 属性中包含 className 值的元素	.red { color:#f00; } p.red { color:#f00; }
#IDName	ID 选择器，选择元素的 id 属性值等于 IDName 的元素	#main { background:#ff0; } p#main { background:#ff0; }
*	通用选择器，选择指定范围内的任何元素	* { margin:0; padding:0; }

2. 组合元素选择器

组合元素选择器通过多个元素的组合，实现精确选择一个或多个元素，它们在 CSS 2.0 版本中获得了支持，但是 IE 7 以下版本浏览器没有提供完全支持，另外 E～F 邻后选择器在 CSS 3.0 版本中才获得支持。组合元素选择器如表 3.2 所示。

表 3.2　CSS 组合元素选择器

选 择 器	说　明	示　例
E,F	分组选择器，同时选择所有 E 元素或 F 元素，E 和 F 之间用逗号分隔	body, p { color:#f00; }
E F	后代元素选择器，选择所有属于 E 元素后代的 F 元素，E 和 F 之间用空格分隔	#main p { color:#f00; }
E > F	子元素选择器，选择所有 E 元素的子元素 F	#main > p { color:#f00; }
E + F	毗邻元素选择器，选择所有紧随 E 元素之后的同级元素 F	#main + p { color:#f00; }
E ~ F	邻后元素选择器，选择所有在 E 元素之后的同级 F 元素	#main ~ p { color:#f00; }

3. 属性选择器

从 CSS 2.1 版本开始支持属性选择器，在 CSS 3.0 版本中获得了扩展(如[att^="val"]、[att$="val"]和[att*="val"]，属性选择器如表 3.3 所示。IE 7 以下版本浏览器没有提供支持。

表 3.3　属性选择器

选 择 器	说　明	示　例
E[att]	选择所有设置 att 属性的 E 元素，不关注属性是否定义了值	p[class] { color:#f00; }
E[att=val]	选择所有 att 属性值等于"val"的 E 元素	p[class=red] { color:#f00; }
E[att~=val]	选择所有 att 属性值包含多个空格分隔的值，其中一个值等于"val"的 E 元素	p[class~=red] { color:#f00; }
E[att|=val]	选择所有 att 属性值包含多个连字符分隔的值，其中一个值以"val"开头的 E 元素	p[class|=red] { color:#f00; }
E[att^="val"]	选择所有 att 属性值以"val"开头的 E 元素	p[class^=red] { color:#f00; }
E[att$="val"]	选择所有 att 属性值以"val"结尾的 E 元素	p[class$=red] { color:#f00; }
E[att*="val"]	选择所有 att 属性值包含"val"的 E 元素	p[class*=red] { color:#f00; }

4. 伪类选择器

伪类选择器如表 3.4 所示。

表 3.4 伪类选择器

选 择 器	说 明	示 例
E:link	选择所有未被单击的 E 元素	p:link { color:#f00; }
E:visited	选择所有已被单击的 E 元素	p:visited { color:#f00; }
E:active	选择鼠标被按下时的 E 元素	p:active { color:#f00; }
E:hover	选择鼠标经过时的 E 元素	p:hover { color:#f00; }
E:focus	选择获得焦点的 E 元素	input[type=text]:focus { color:#f00; }
E:first-child	选择父元素的第一个子元素	p:first-child { color:#f00; }
E:lang(c)	选择 lang 属性等于 c 的 E 元素	q:lang(ch) { color:#f00; }

5. 伪对象选择器

伪对象选择器如表 3.5 所示。

表 3.5 伪对象选择器

选 择 器	说 明	示 例
E:first-line	选择 E 元素的第一行对象	p:first-line { color:#f00; }
E:first-letter	选择 E 元素的第一个字母	p p:first-letter { color:#f00; }
E:before	在 E 元素之前插入生成的内容	p:before { color:#f00; }
E:after	在 E 元素之后插入生成的内容	p:after { color:#f00; }

6. 用户界面选择器

用户界面选择器属于伪类选择器，用来标识表单元素的状态，也称为 UI 元素状态伪类。在 CSS 3.0 版本中开始被支持，由于其比较特殊，故单列出来，如表 3.6 所示。

表 3.6 用户界面选择器

选 择 器	说 明	示 例
E:enabled	选择表单中激活的 E 元素	input:enabled { color:#f00; }
E:disabled	选择表单中禁用的 E 元素	input:disabled { color:#f00; }
E:checked	选择表单中被选中的 E 元素，如 radio(单选框)或 checkbox(复选框)元素	input:checked { color:#f00; }
E:selection	选择表单中当前被选中的元素，如 radio(单选框)、checkbox(复选框)元素或 option(列表框中的选项)元素	input: selection { color:#f00; }

7. 结构性选择器

结构性选择器也属于伪类选择器，在 CSS 3.0 版本中开始被支持，由于其比较特殊，故单列出来，如表 3.7 所示。

表 3.7 结构性选择器

选 择 器	说 明	示 例
E:root	选择文档的根元素，对于 HTML 文档来说，就是 HTML 元素	p:root { font-size:12px; }
E:nth-child(n)	选择父元素的第 n 个子元素，第一个编号为 1	p:nth-child(1) { color:#f00; }
E:nth-last-child(n)	选择父元素的倒数第 n 个子元素，第一个编号为 1	p:nth-last-child(3) { color:#f00; }
E:nth-of-type(n)	与:nth-child()选择器作用类似，但是仅匹配使用同种类型标签的元素	p:nth-of-type(3) { color:#f00; }
E:nth-last-of-type(n)	与:nth-last-child()选择器作用类似，但是仅匹配使用同种类型标签的元素	p:nth-last-of-type(3) { color:#f00; }
E:last-child	选择父元素的最后一个子元素，等同于:nth-last-child(1)	p:last-child { color:#f00; }
E:first-of-type	选择父元素下使用同种标签的第一个子元素，等同于:nth-of-type(1)	p:first-of-type { color:#f00; }
E:last-of-type	选择父元素下使用同种标签的最后一个子元素，等同于:nth-last-of-type(1)	p:last-of-type { color:#f00; }
E:only-child	选择父元素下仅有的一个子元素，等同于:first-child:last-child 或 :nth-child(1):nth-last-child(1)	p:only-child { color:#f00; }
E:only-of-type	选择父元素下使用同种标签的惟一一个子元素，等同于:first-of-type:last-of-type 或 :nth-of-type(1):nth-last-of-type(1)	p:only-of-type { color:#f00; }
E:empty	选择一个不包含任何子元素的元素，其中文本节点也可以作为子元素而存在	p:empty { color:#f00; }

8. 其他选择器

在 CSS 3.0 版本中还定义了两个特殊的伪类：反选伪类和目标伪类，如表 3.8 所示。

表 3.8 其他选择器

选 择 器	说 明	示 例
E:not(s)	选择不符合当前选择器的任何元素	p:not { font-size:12px; }
E:target	选择文档中特定 ID 元素被单击后的效果	p:target { color:#f00; }

3.1.2 jQuery 盗了谁的版

jQuery 不是 JavaScript 选择器的缔造者，cssQuery 应该算上实际的开山者，而 jQuery 仅模仿于 cssQuery，并在超越中获得了成功。

伟大始于卑微，jQuery 也是如此。John Resig(jQuery 作者)不是神，读者不要迷信，他仅是一位 JavaScript 代码的爱好者，大概他无意间看到 cssQuery 代码的精巧和使用的灵巧，才萌生了模仿与超越的念头和动力。下面我们就来领略一下 cssQuery 的真容。

cssQuery 是 Dean Edwards 写的一段 JavaScript 选择器函数，它模仿 CSS 选择器的用法和习惯，实现在脚本中快速、方便地查找文档元素。cssQuery 具有强大的跨浏览器兼容能力，支持 CSS1、CSS2 版本的所有选择器语法和用法，对于 CSS3 版本中部分选择器也提供支持。

cssQuery 可兼容的浏览器及其版本如下所示。

- IE 5+ (Windows)
- IE 5.2 (Mac)
- Firefox/Mozilla 1.6+
- Opera 7+
- Netscape 6+
- Safari 1.2

简单概括，就是当前所有主流浏览器也被纳入了兼容的范围之内。详细信息请参阅下面提供的链接。

- cssQuery 官方网址：http://dean.edwards.name/my/cssQuery
- cssQuery 函数源代码下载地址：http://dean.edwards.name/download/#cssQuery.js
- cssQuery 函数测试页面：http://dean.edwards.name/my/cssQuery/test.html#

3.1.3 认识 cssQuery 选择器

由于 jQuery 与 cssQuery 一脉相承，且 jQuery 选择器源于 cssQuery，因此掌握了 cssQuery 的使用，也就等于部分驾驭了 jQuery。cssQuery 支持的 CSS 选择器类型如表 3.9 所示。

表 3.9 cssQuery 支持的 CSS 选择器类型

选 择 器	说 明
*	选择所有类型的元素
E	选择指定类型的 E 元素
E F	选择 E 类型元素包含的所有 F 类型元素
E > F	选择 E 类型元素包含的所有 F 类型子元素
E + F	选择 E 类型元素之后相邻的 F 类型同级元素
E ~ F	选择 E 类型元素之后的 F 类型同级元素
E.warning	选择类名(即 class 属性值)为 warning 的 E 类型元素
E#myid	选择 ID 为 myid 的 E 类型元素
E:link	选择具有超链接的 E 类型伪元素

续表

选 择 器	说 明
E:first-child	选择父元素的第一个子元素，等同于:nth-child(1)
E:last-child	选择父元素的最后一个子元素，等同于:nth-last-child(1)
E:nth-child(n)	选择父元素的第 n 个子元素
E:nth-last-child(n)	选择父元素的倒数第 n 个子元素
E:only-child	选择父元素仅有的一个子元素
E:root	选择文档的根元素，在 HTML 中根元素永远是 HTML
E:lang(fr)	选择 lang 属性等于 fr 的 E 元素
E:target	选择相关 URL 指向的 E 元素，例如 div#top:target，在浏览器地址栏的 URL 后面输入#top，可以选择 ID 为 top 的 div 元素
E:enabled	选择表单中可用的 E 元素
E:disabled	选择表单中不可用的 E 元素
E:checked	选择文档的根元素
E:contains("foo")	选择 E 类型元素中包含"foo"文本的元素
E:not(s)	选择不符合当前选择器的任何元素
E[foo]	选择设置了 foo 属性的 E 元素
E[foo="bar"]	选择设置了 foo 属性，且属性值等于"bar"字符的 E 元素
E[foo~="bar"]	选择设置了 foo 属性，且属性值包含多个以空格进行分隔的值，其中一个值等于"bar"字符的 E 元素
E[foo|="bar"]	选择设置了 foo 属性，且属性值包含多个以连字符进行分隔的值，其中一个值以"bar"字符开头的 E 元素
E[foo^="bar"]	选择设置了 foo 属性，且属性值以"bar"字符开头的 E 元素
E[foo$="bar"]	选择设置了 foo 属性，且属性值以"bar"字符结尾的 E 元素
E[foo*="bar"]	选择设置了 foo 属性，且属性值包含"bar"字符的 E 元素

3.1.4 使用 cssQuery 选择器

使用 cssQuery 选择器之前，请到官方网站下载 cssQuery()函数的源代码(http://dean.edwards.name/download/#cssQuery.js)，解压之后在页面中导入 cssQuery-p.js 文件，这是一个压缩文件，在 src 目录下可以研究 cssQuery 选择器的源代码。

```
<script src="cssQuery-p.js" type="text/javascript"></script>
```

然后，我们在页面中构建一个简单的结构，如下所示。

```
<div id="test">
    <p class="p1">
        <span>文本</span>
        <a href="#" title="测试超链接">超链接</a>
```

```
    </p>
</div>
```

最后，就可以在脚本中引用 cssQuery()函数，以实现精确选择页面中的元素。cssQuery()函数的用法如下。

```
elements = cssQuery(selector [, from]);
```

该函数的返回值为一个元素数组，如果没有匹配的元素，则返回一个空数组。

参数 selector 为字符串型的 CSS 选择器；参数 from 可选，表示 document、element 或者元素数组，用来指定选择器筛选的范围。

如果我们希望选择 span 元素，则可以构建如下的选择器。

```
div#test .p1 span
```

选择 span 元素，该元素包含在 ID 为 test 的 div 元素下的类名为 p1 的元素中。当然，你也可以构建其他形式的选择器，这就要看具体的文档结构了。同理，如果选择 a 元素，则可以构建如下的选择器。

```
a[href][title]
```

此选择器选择包含 href 和 title 属性的 a 元素。

构建好选择器字符串，以及 cssQuery()函数之后，我们就可以轻松控制所选择元素的样式或脚本行为了。例如，使用脚本控制标签包含的文本显示为红色，超链接文本的字体颜色为绿色，实现的完整代码如下。

```
<html>
<head>
<script src="cssQuery-p.js" type="text/javascript"></script>
<script type="text/javascript">
window.onload = function(){    //页面初始化事件函数
    var selector = "div#test .p1 span"; //定义选择器字符串
    var elements = cssQuery(selector); //调用 cssQuery()函数，获选匹配元素
    elements[0].style.color = "red";    //设置选择的第 1 个元素的字体颜色为红色
    selector = "a[href][title]";        //定义选择器字符串
    elements = cssQuery(selector);      //调用 cssQuery()函数，获选匹配元素
    elements[0].style.color = "green";  //设置选择的第 1 个元素的字体颜色为绿色
}
</script>
</head>
<body>
<div id="test">
    <p class="p1">
        <span>文本</span>
        <a href="#" title="测试超链接">超链接</a>
    </p>
```

```
</div>
</body>
</html>
```

3.1.5 初步接触 jQuery 选择器

jQuery 选择器完全继承 CSS 选择器的语法规则和使用习惯，如果读者熟悉了 CSS 选择器的用法，那么也就基本上掌握了 jQuery 选择器的用法。

使用 jQuery 选择器可以选择文档对象模型(DOM)中的特定元素，即获取特定元素的引用句柄，然后就可以为其附加各种行为，或者更改选择元素的样式等。熟练使用选择器是学习 jQuery 的基础，jQuery 的所有行为和命令也都建立在选择元素的基础之上。下面看一个示例。

例 1：使用 JavaScript 的传统方法来设计一个斑马线效果的隔行换色表格，代码如下。

```
<html>
<head>
<script type="text/javascript" >
window.onload = function(){    //页面初始化处理函数
    var tds = document.getElementsByTagName("td");    //获取表格行引用数组
    for(var i = 0; i < tds.length; i ++ ){    //遍历表格行
        if(i % 2 != 0 ) tds[i].style.background = "yellow";    //设置偶数行背景色为黄色
    }
}
</script>
</head>
<body>
<table width="100%">
    <tr><td>1</td></tr>
    <tr><td>2</td></tr>
    <tr><td>3</td></tr>
    <tr><td>4</td></tr>
    <tr><td>5</td></tr>
    <tr><td>6</td></tr>
</table>
</body>
</html>
```

我们可以看到，上面的示例使用了 6 行脚本代码完成了一个简单的隔行换色的表格效果。如果使用 jQuery 来实现相同的效果，则只需要 3 行代码。

例 2：使用 jQuery 方法设计一个斑马线效果的隔行换色表格，代码如下。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){    //页面初始化处理函数
    $("tr:odd").css("background","yellow");//选择奇数行，并设置背景颜色样式为黄色
});
```

```
</script>
```

在使用 jQuery 之前，不要忘记使用<script>标签导入外部的 jQuery 文件，本书选用的版本为 jquery-1.3.2.js。

美元符号($)在 jQuery 中具有广泛的语义，它是 jQuery 的别称(或者简写)，当$()函数包含一个函数时，它的功能等同于 window.onload 事件处理函数。而当$()函数包含一个字符串时，该字符串表示选择字符串，此时$()函数就表示一个选择器，它能够根据选择字符串的类型和形式决定所要选择的文档元素。

css()函数是一个 jQuery 对象方法，它能够为所有选择的元素附加样式，其中第一个参数表示 CSS 属性名字符串，而第二个字符串表示 CSS 属性值(全写格式)。

上面的代码表示，在文档初始化加载之后，执行选择奇数行表格行，并为所选择的表格行附加样式，定义它们的背景色为黄色。

很明显，使用 jQuery 实现相同的功能要灵便很多，仅需要 3 行代码。jQuery 选择器的写法与 CSS 选择器的写法基本相同，但是它们的作用对象和效果却是不同的。CSS 是为选择的元素添加表现层的样式，而 jQuery 是为选择的元素添加行为层的动态样式。当然，jQuery 选择器为我们解决了以下两个难题。

第一，支持 CSS1、CSS2、CSS3 不同版本的所有选择器。而很多浏览器并没有完全支持 CSS3 版本的选择器。

第二，支持不同类型的主流浏览器。这在实际开发中是件令人很头疼的事情，现在如果使用 jQuery 选择文档元素，就不用考虑这么多烦人的兼容性问题了。

简单说，jQuery 选择器具有如下优势。

- 简化代码书写。这个就不多说了，你可以从上面示例代码中感受到它的简便。
- 完善的支持。支持 CSS 不同版本选择器，同时也支持不同类型浏览器。
- 完善的处理机制。jQuery 选择器的核心依然依靠 JavaScript 的 getElementById()和 getElementsByTagName()方法，但是它封装了这两个方法，既简洁，又避免了易错问题。

jQuery 选择器返回的永远是一个数组对象，但是若没有找到指定的元素，则会返回一个空的数组对象。如果使用 getElementById()和 getElementsByTagName()方法，就容易抛出异常。因此，判断一个 jQuery 对象是否存在，不能够使用如下的语句。

```
if($("tr")){
    // code
}
```

而应该使用数组长度来判断，语句如下。

```
if($("tr").length > 0){
```

```
    // code
}
```

jQuery 选择器分为简单选择器、包含选择器、筛选选择器、内容和属性过滤选择器，以及表单专用选择器等，在下面章节中我们将详细讲解每类选择器，并比较 JavaScript 与 jQuery 的用法和执行效率，分析不同类型 jQuery 选择器的工作机制。

3.2 简单选择器

简单选择器主要包括 5 种类型，这与 CSS 基本选择器类型相一致，如表 3.10 所示。

表 3.10 jQuery 简单选择器类型

选 择 器	说 明	返 回 值
#id	根据指定的 ID 值选择一个元素	单个元素
element	根据指定的元素类型名称选择该类型的所有元素	同类型集合元素
.class	根据指定的类名选择所有同类元素	集合元素
*	选择限定范围的所有元素	所有元素的集合
selector1,selector2,selectorN	分别选择选择器组中每个选择器匹配的元素，然后合并返回所有元素	集合元素

3.2.1 选择指定 ID 元素

1. JavaScript 实现方法

JavaScript 提供了原生方法实现在 DOM 中选择指定 ID 值的元素，用法如下。

```
var element = document.getElementById("id");
```

其中，getElementById()方法的返回值为所选择元素的对象引用，document 是 Window 对象的属性，它引用 Document 对象，参数值为字符串型 ID 值，该值在 HTML 文档标签中通过 id 特性设置。例如，在下面的示例中，选择文档中的 ID 值为“div1”的元素，并设置它的背景色为红色。

```
<script type="text/javascript" >
window.onload = function(){    //页面初始化函数
    var e = document.getElementById("div1");    //选择 ID 等于"div1"的元素
    e.style.background = "red";    //控制选择元素的背景色
}
</script>
<div id="div1">测试盒子</div>
```

2. jQuery 实现方法

jQuery 简化了操作，如果要实现上述相同的功能，则可以这样来设计。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
window.onload = function(){
    var e = $("#div1");
    e[0].style.background = "red";
}
</script>
<div id="div1">测试盒子</div>
```

其中$("#div1")中的"#div1"表示 ID 选择器，jQuery 构造器能够根据这个选择器，准确定位到 DOM 中该元素的位置，并返回包含该元素引用的数组。

3. 执行效率比较分析

从本质上分析，JavaScript 与 jQuery 在选择 ID 元素时是异曲同工的，jQuery 只不过是包装了 getElementById()方法。但是，从执行效率来分析，两者的差距还是很大的，我们不妨做一个简单的试验。

首先，使用循环语句生成 1000 个 span 元素，并按顺序设置每个 span 元素的 ID 值，实现代码如下。

```
<script type="text/javascript" >
window.onload = function(){
    for(var i=0;i<1000;i++){
        var span = document.createElement("span");          //创建 span 元素
        span.setAttribute("id","span" + i);      //设置 id 属性值
        document.getElementsByTagName("body")[0].appendChild(span);     //把新建元素附加
到文档中
    }
}
</script>
```

然后，使用 JavaScript 原生方法循环获取这些创建的 span 元素，并把它们推入到数组中存储，实现代码如下。

```
<script type="text/javascript" >
window.onload = function(){
    //循环生成 span 元素
    var a = [];    //定义数组
    var t1 = new Date();    //获取当前时间
    for(var j=0;j<1000;j++){
        var b = document.getElementById("span"+j);    //获取每一个 span 元素
        a.push(b);    //把选择的 span 元素推入数组
    }
```

```
    var t2 = new Date();    //获取当前时间
    alert("执行时间 = "+ (t2-t1) + " 毫秒");    //提示执行时间长度
}
</script>
```

最后，在浏览器中预览，你会发现上面代码的执行时间大约为 593 毫秒(如图 3.1 所示)，当然不同系统、不同类型浏览器以及每一次执行所花费的时间都会略有差异。

但是，如果你使用 jQuery 来选择该 ID 元素，其实现代码如下。在浏览器中预览，你会惊讶地发现，它所花费的时间竟然几乎是 JavaScript 原生方法的一倍(如图 3.2 所示)。

图 3.1 使用 JavaScript 原生方法选择所花费的时间

图 3.2 使用 jQuery 方法选择所花费的时间

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
window.onload = function(){
    //循环生成 span 元素，代码如上
    var a = [];    //定义数组
    var t1 = new Date();    //获取当前时间
    for(var j=0;j<1000;j++){
        var b = $("#span"+j);    //获取每一个 span 元素
        a.push(b);    //把选择的 span 元素推入数组
    }
    var t2 = new Date();    //获取当前时间
    alert("执行时间 = "+ (t2-t1) + " 毫秒");    //提示执行时间长度
}
</script>
```

由于 jQuery 需要对参数字符串进行分析，并匹配出所传递的参数值是 ID 值，然后才能调用 getElementById()方法获取该 ID 元素，所以花费的时间一定会成倍地增长。

因此，在不是很必须的前提下，可以考虑使用如下方法来实现快速选择 ID 元素。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
window.onload = function(){
    //循环生成 span 元素，代码如上
    var a = [];    //定义数组
    var t1 = new Date();    //获取当前时间
    var $ = document.getElementById;    //把 JavaScript 原生方法传递给变量$
    for(var j=0;j<1000;j++){
        var b = $("#span"+j);    //获取每一个 span 元素
```

```
        a.push(b);     //把选择的 span 元素推入数组
    }
    $ = jQuery;      //恢复变量$的默认值
    var t2 = new Date();     //获取当前时间
    alert("执行时间 = "+ (t2-t1) + " 毫秒");     //提示执行时间长度
}
</script>
```

在上面的示例中，通过临时改变变量$引用的对象，以实现使用原生方法直接选择 ID 元素，然后再恢复变量$的默认值。当然，你也可以直接使用 Document 对象的 getElementById() 方法来获取该 ID 元素。

4. jQuery 方法应用解析

在 ID 选择器中，如果选择器包含特殊字符，则可以在 jQuery 中使用两个斜杠对特殊字符进行转义。例如：

```
<div id="a.b">div1</div>
<div id="a:b">div2</div>
<div id="[div]">div3</div>
```

在上面三个 div 元素中，它们的 id 特性值都包含了特殊的字符，如果不进行处理，jQuery 在解析时会因误解而不能够达到目的。此时，你可以使用如下方法来实现准确选择。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("#a\\.b").css("color","red");
    $("#a\\:b").css("color","red");
    $("#\\[div\\]").css("color","red");
})
</script>
```

但是，如果我们直接使用 JavaScript 的原生方法 getElementById()就不用顾虑这个问题，例如，上面示例的代码可以改写为以下形式。

```
<script type="text/javascript" >
$(function(){
    document.getElementById("a.b").style.color = "red";
    document.getElementById("a:b").style.color = "red";
    document.getElementById("[div]").style.color = "red";
})
</script>
```

这是为什么呢？

原来，在执行 jQuery()函数时，jQuery 使用到正则表达式来匹配参数值，并判断当前参数是否为 ID 值，代码如下。

```
ID: /#((?:[\w\u00c0-\uFFFF_-]|\\.)+)/
```

而正则表达式对于特殊字符是敏感的，要避免正则表达式被误解，就要考虑进行字符转义，在正则表达式字符串中一般都通过双斜杠来转义特殊字符。

3.2.2 选择指定类型元素

1. JavaScript 实现方法

为了实现选择指定类型的元素，JavaScript 也提供了一个原生方法，用来在 DOM 中选择指定类型的元素，用法如下。

```
var elements = document.getElementsByTagName("tagName");
```

其中，getElementsByTagName()方法的返回值为所选择类型元素集合的数组对象引用，document 是 Window 对象的属性，它引用 Document 对象，参数值为字符串型 HTML 标签名称。例如，在下面的示例中，HTML 文档中包含 3 个 div 类型的元素。

```
<div>div1</div>
<div>div2</div>
<div>div3</div>
```

使用 JavaScript 原生方法 getElementsByTagName()选择文档中标签名称为“div”的所有元素，并设置它们的前景色为红色。其实现代码如下。

```
<script type="text/javascript" >
window.onload = function(){    //页面初始化函数
    var divs = document.getElementsByTagName("div");    //返回 div 元素集合
    for(var i=0;i<divs.length;i++){    //遍历 div 元素集合
        divs[i].style.color = "red";    //设置 div 元素的前景色为红色
    }
}
</script>
```

2. jQuery 实现方法

在 jQuery 中，与 ID 选择器不同，类型选择器的字符串不需要附加标识前缀(#)。因此，如果要实现上述相同的功能，则可以这样来设计。

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("div").css("color","red");
})
</script>
```

此时$("div")与 document.getElementsByTagName("div")的运行结果是一样的，都返回一个元素集合对象。所以，你还可以混合使用它们，代码如下。

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
window.onload = function(){    //以 JavaScript 方法初始化页面处理函数
    var divs = $("div");    //以 jQuery 方法选择所有 div 元素
    for(var i=0;i<divs.length;i++){    //以 JavaScript 方法遍历返回的 jQuery 结果对象
        divs[i].style.color = "red";
    }
}
</script>
```

其实，这个也没有什么奇怪的，jQuery 技术本身就是在 JavaScript 基础上进行的代码封装，它们都遵循相同的语法规则。

3. 执行效率比较分析

与 ID 选择器一样，jQuery 的类型选择器也存在效率低下问题，而且这个问题还比较严重，为了说明这个问题，我们可以看一个示例。

使用 JavaScript 动态生成 10000 个 span 元素，然后使用 getElementsByTagName()方法选择所有 span 元素，则在 IE 7 浏览器中所用时间大约为 16 毫秒(如图 3.3 所示)，多次执行可能仅需要 0 毫秒，如果在 Firefox 中仅需要 0 毫秒。其对应的代码如下。

```
<script type="text/javascript">
window.onload = function(){
    var body = document.getElementsByTagName("body")[0];    //选择 body 元素
    for(var i=0;i<10000;i++){    //循环生成 span 元素
        var span = document.createElement("span");    //创建 span 元素
        body.appendChild(span);    //把 span 元素附加到 body 元素内子元素末尾
    }
    var t1 = new Date();
    var s = document.getElementsByTagName("span");    //获取所有 span 元素
    var t2 = new Date();
    alert("执行时间 = "+ (t2-t1) + " 毫秒");
}
</script>
```

下面使用 jQuery 选择动态生成的 10000 个 span 元素，则在 IE 7 浏览器中所用时间可能需要 156 毫秒(如图 3.4 所示)，初次执行可能需要 200 多毫秒，而在 Firefox 中竟然需要 300 多毫秒。其对应的代码如下。

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
window.onload = function(){
    //此处为动态生成的 10000 个 span 元素，代码如上
    var t1 = new Date();
```

```
    var s = $("span");    //获取所有 span 元素
    var t2 = new Date();
    alert("执行时间 = "+ (t2-t1) + " 毫秒");
}
</script>
```

图 3.3 使用 JavaScript 原生方法选择所花费的时间

图 3.4 使用 jQuery 方法选择所花费的时间

出现如此巨大的执行效率落差，不仅仅是因为 jQuery 需要使用正则表达式匹配选择器类型，过滤出的参数值为标签名字符串，另外，由于 jQuery()函数需要对第一个参数执行多路判断(即多分支条件判断)，而对于标签字符串的判断位于队列的后面，不像 ID 选择器是第一个就被匹配判断(即第一条件分支)，所以就耗费掉大量的宝贵时间。

从执行效率的角度考虑，读者更应该积极考虑多使用 JavaScript 原生的 getElementsByTagName() 方法来选择同类型的元素。即使在复杂的 jQuery 编程环境中，嵌入使用 getElements ByTagName()方法要比直接使用$()方法高效，虽然$()方法的这种简写让你感觉非常惬意。

3.2.3 选择指定类元素

1. JavaScript 实现方法

JavaScript 没有内置的类选择方法，读者可以为其扩展一个方法，实现代码如下所示。

```
document.getElementsByClassName = function(className) {
    var el = [],
        _el = document.getElementsByTagName('*');     //获取所有元素
    for (var i=0; i<_el.length; i++ ) {    //遍历元素集合
      if (_el[i].className == className ) {    //获取相同类名的元素
         el[el.length] = _el[i];
       }
   }
   return el;
}
```

例如，在下面的示例中，HTML 文档中包含 3 个 div 类型的元素。

```
<div class="red">div1</div>
<div>div2</div>
```

```
<div class="red">div3</div>
```

然后使用自定义类选择器方法 getElementsByClassName()选择文档中类名为“red”的所有元素，并设置它们的前景色为红色，实现代码如下。

```
<script type="text/javascript" >
window.onload = function(){    //页面初始化函数
    var red = document.getElementsByClassName("red");    //返回 div 元素集合
    for(var i=0, l=red.length; i< l; i++){    //遍历 div 元素集合
        red[i].style.color = "red";    //设置 div 元素的前景色为红色
    }
}
</script>
```

2. jQuery 实现方法

在 jQuery 中，类选择器的字符串需要附加标识前缀(.)。因此，如果要实现上述相同的功能，则可以进行如下的设计。

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $(".red").css("color","red");
})
</script>
```

此时$(".red")与 document.getElementsByClassName("red")的运行结果是一样的，都返回一个元素集合对象。

3. 执行效率比较分析

除了 JavaScript 原生方法外，jQuery 所提供的 DOM 访问方法也是高效的。下面我们做一个比较。使用 JavaScript 动态生成 5000 个 span 元素，然后使用 getElementsByClassName()方法选择所有 span 元素，则在 IE 8 浏览器中所用时间大约为 3685 毫秒(如图 3.5 所示)。其对应的代码如下。

```
<script type="text/javascript">
window.onload = function(){
    var body = document.getElementsByTagName("body")[0];    //选择 body 元素
    for(var i=0;i<5000;i++){    //循环生成 span 元素
        var span = document.createElement("span");    //创建 span 元素
        span.setAttribute("class","red");    //设置类
        body.appendChild(span);    //把 span 元素附加到 body 元素内子元素的末尾
    }
    var t1 = new Date();
    var $ = document.getElementsByClassName;
    for(var j=0;j<500;j++){
        var b = $("red");
```

```
        a.push(b);
    }
    var t2 = new Date();
    alert("执行时间 = "+ (t2-t1) + " 毫秒");
}
</script>
```

下面使用 jQuery 选择动态生成的 500 个 span 元素，则在 IE 8 浏览器中所用时间可能需要 1348 毫秒(如图 3.6 所示)。其对应的代码如下。

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    for(var i=0;i<500;i++){
        var span = document.createElement("span");
        span.setAttribute("class","red");
        document.getElementsByTagName("body")[0].appendChild(span);
    }
    var a = [];
    var t1 = new Date();
    for(var j=0;j<500;j++){
        var b = $(".red");
        a.push(b);
    }
    var t2 = new Date();
    alert("执行时间 = "+ (t2-t1) + " 毫秒");
})
</script>
```

图 3.5 使用 JavaScript 原生方法选择所花费的时间

图 3.6 使用 jQuery 方法选择所花费的时间

从执行效率的角度考虑，使用自定义的 getElementsByClassName()方法不如使用 jQuery 选择器，因为作为 jQuery 技术框架，它的代码经过了优化处理，执行速度要比自定义的方法快。

3.2.4 选择所有元素及其优化

jQuery 定义了*选择器，该选择器能够匹配指定上下文中的所有元素。例如，下面的示例将选择文档中 body 元素下包含的所有元素。

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("body *").css("color","red");
})
</script>

<div>DIV</div>
<span>SPAN</span>
<p>P</p>
```

JavaScript 也提供了类似的匹配方法，读者可以使用 document.getElementsByTagName("*")。因此，我们可以使用下面的方法来模拟上面示例的实现效果。

```
<script type="text/javascript" >
$(function(){
    var all = document.getElementsByTagName("*");
    for(var i=0; i<all.length; i++){
        all[i].style.color = "red";
    }
})
</script>

<div>DIV</div>
<span>SPAN</span>
<p>P</p>
```

当然，更高效的方法是把 JavaScript 原生方法和 jQuery 迭代操作相结合，这样可以提高代码执行效率，也不会多写很多代码。实现方法是：先使用 JavaScript 原生方法获取页面中所有元素，然后把这个 DOM 元素集合传递给 jQuery()函数，再把 JavaScript 数组集合封装为 jQuery 对象的类数组集合，最后借助 jQuery 的 css()方法快速定义样式，从而提高整个程序的执行速度。其实现代码如下所示。

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var all = document.getElementsByTagName("*");
    $(all).css("color","red");
})
</script>
```

3.2.5 选择多组元素及其实现

jQuery 支持 CSS 的分组选择器，通过这种方式可以扩大选择器的选择范围，同时增强 jQuery 选择器引擎的应用能力。选择多组元素可以通过逗号分隔符来分隔多个不同的选择器，这些选择器可以是任意类型的，也可以是复合选择器。

例如，下面的示例分别使用了不同类型的选择器匹配指定元素，然后通过选择器分组方式把它们组合在一起，从而简化代码的书写。

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("h2, #wrap, span.red, [title='text']").css("color","red");
})
</script>

<h2>H2</h2>
<div id="wrap">DIV</div>
<span class="red">SPAN</span>
<p title="text">P</p>
```

分组选择器的实现思路是这样的。

首先，在 Sizzle()构造器函数中，获取选择器字符串，并通过下面的正则表达式模式进行匹配。

```
var chunker = /((?:\((?:\([^()]+\)|[^()]+)+\)|\[(?:\[[^\]]*\]|['"][^'"]*['"]|[^\]
'"]+)+\]|\\.|[^ >+~,(\[\\]+)+|[>+~])(\s*,\s*)?/g,
```

然后把该正则表达式的下标位置恢复到初始化位置。用选择器字符串中的逗号作为分隔符，把选择器字符串分开，再分别推入到 parts 数组中。

最后，通过条件语句分别判断 parts 数组的长度，如果长度大于 1，则重复调用 Sizzle()函数，并分析第一个逗号后面的选择器字符串，依此类推。主要实现代码如下所示。

```
var Sizzle = function(selector, context, results, seed) {
    //……
    chunker.lastIndex = 0;
    //用逗号为分隔符，分开选择器字符串
    while ( (m = chunker.exec(selector)) !== null ) {
        parts.push( m[1] );
        if ( m[2] ) {
            extra = RegExp.rightContext;
            break;
        }
    }
    //处理多组选择器
    if ( parts.length > 1 && origPOS.exec( selector ) ) {
```

```
            //……
            //重复调用 Sizzle()，迭代操作多组选择器的字符串
            Sizzle( parts.shift(), context );
            //……
        //处理单个选择器
        } else {
            //……
                Sizzle.find( parts.pop(), parts.length === 1 && context.parentNode ?
context.parentNode : context, isXML(context) );
            set = Sizzle.filter( ret.expr, ret.set );
            //……
        }
        //……
        return results;
};
```

3.3 关系选择器

关系选择器就是根据 HTML 文档结构中节点之间的包含或并列关系，决定匹配元素的一种方法。jQuery 模仿 CSS 的关系过滤模式定义了 4 个层级选择器，同时还根据包含关系，自定义了 4 个子元素选择器。

3.3.1 层级选择器

层级选择器能够根据元素之间的关系进行匹配操作，主要包括选择器、子选择器、相邻选择器和兄弟选择器。详细说明如表 3.11 所示。

表 3.11 层级选择器

选 择 器	说 明
ancestor descendant	在给定的祖先元素下匹配所有的后代元素。ancestor 表示任何有效选择器，descendant 表示用以匹配元素的选择器，并且它是第一个选择器的后代元素。 例如，$("form input")可以匹配表单下所有的 input 元素
parent > child	在给定的父元素下匹配所有的子元素。parent 表示任何有效的选择器，child 表示用以匹配元素的选择器，并且它是第一个选择器的子元素。 例如，$("form > input")可以匹配表单下所有的子级 input 元素
prev + next	匹配所有紧接在 prev 元素后的 next 元素。prev 表示任何有效选择器，next 表示一个有效选择器并且紧接着第一个选择器。 例如，$("label + input")可以匹配所有跟在 label 后面的 input 元素
prev ~ siblings	匹配 prev 元素之后的所有 siblings 元素。prev 表示任何有效选择器，siblings 表示一个选择器，并且它作为第一个选择器的同辈。 例如，$("form ~ input")可以匹配所有与表单同辈的 input 元素

例如，在下面的示例中，利用 jQuery 定义的层级选择器可以方便地控制 HTML 文档中各级元素的样式。虽然这些结构没有定义 id 或 class 属性，但是这并不影响用户方便、精确地控制文档样式。本示例的演示效果如图 3.7 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("div").css("border", "solid 1px red");    //控制文档中的所有 div 元素
    $("div>div").css("margin", "2em");    //控制 div 元素包含的 div 子元素，实际上它与 div div
包含选择器所匹配的元素是相同的
    $("div div").css("background", "#ff0");        //控制最外层 div 元素包含的所有 div 元素
    $("div div div").css("background", "#f0f");    //控制第三层及其以内的 div 元素
    $("div + p").css("margin", "2em");    //控制 div 相邻的 p 元素
    $("div:eq(1) ~ p").css("background", "blue");    //控制 div 后面并列的所有 p 元素
})
</script>

<div>一级 div 元素
    <div>二级 div 元素
        <div>
            三级 div 元素
        </div>
        <p>段落文本 11</p>
        <p>段落文本 12</p>
    </div>
    <p>段落文本 21</p>
    <p>段落文本 22</p>
</div>
<p>段落文本 31</p>
<p>段落文本 32</p>
```

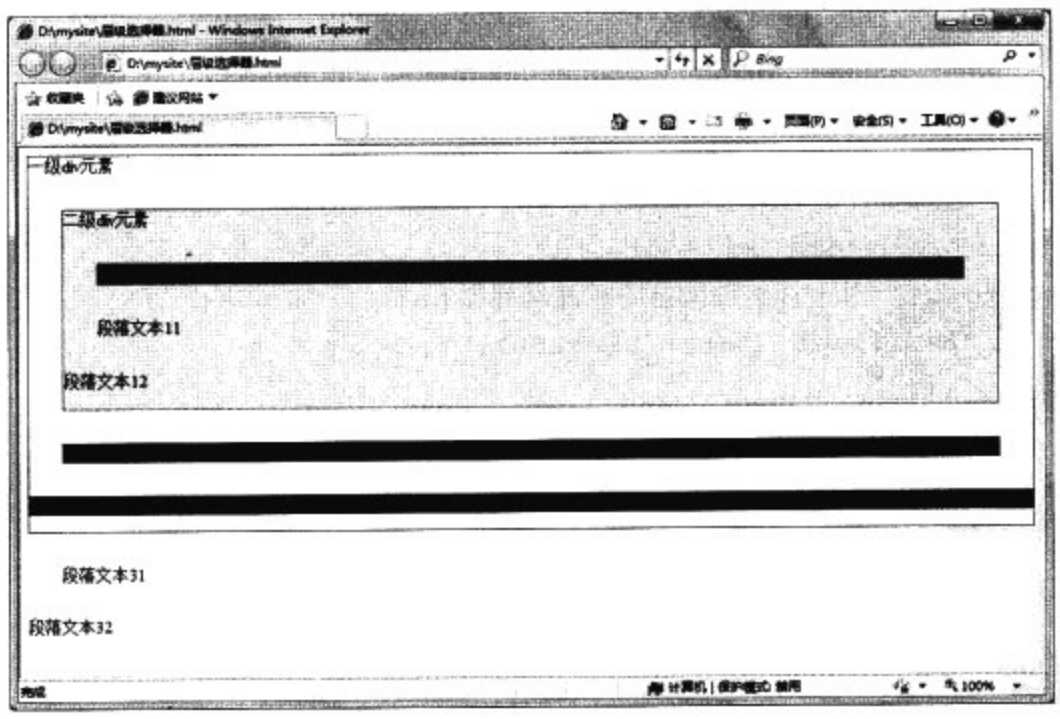


图 3.7　层级选择器的演示效果

3.3.2 层级选择器的实现方法

jQuery在Expr. relative对象中定义了4个层级选择器函数，然后在Sizzle()接口函数中直接调用这些函数来匹配对应的选择器类型，并根据这些选择器表达式调用Sizzle.filter()过滤函数，筛选指定关系的元素，并把这些匹配元素封装到jQuery对象中返回。其实现代码如下。

```
var Expr = Sizzle.selectors = {
    relative: {
        //匹配prev + next选择器
        "+": function(checkSet, part, isXML){
            //处理标签字符串
            var isPartStr = typeof part === "string",
                isTag = isPartStr && !/\W/.test(part),
                isPartStrNotTag = isPartStr && !isTag;
            //把标签转换为大写形式
            if ( isTag && !isXML ) {
                part = part.toUpperCase();
            }
            //遍历检测结果集
            for ( var i = 0, l = checkSet.length, elem; i < l; i++ ) {
                //获得elem的前一个节点
                if ( (elem == checkSet[i]) ) {
                    ///返回已选元素的上一个同属节点(同级节点中的上一个)
                    //当elem的节点类型不为元素节点的时候，继续得到elem的前一个节点，否则循环结束
                    while ( (elem = elem.previousSibling) && elem.nodeType !== 1 ) {}
                    //筛选符合指定标签的相邻元素
                    ///处理返回同级DOM对象
                    checkSet[i] = isPartStrNotTag || elem && elem.nodeName === part ?
                        elem || false :
                        elem === part;
                }
            }
        }
        //如果不是标签，则调用Sizzle.filter()函数筛选结果集
        if ( isPartStrNotTag ) {
            //在checkSet以后的元素中过滤选择器part
            Sizzle.filter( part, checkSet, true );
        }
    },
    //匹配parent > child选择器
    ">": function(checkSet, part, isXML){
        //part是选择器
        var isPartStr = typeof part === "string";
        //当part为单词字符时，如$("form > input")，part为"form"
        if ( isPartStr && !/\W/.test(part) ) {
            part = isXML ? part : part.toUpperCase();
            //遍历检测结果集
            for ( var i = 0, l = checkSet.length; i < l; i++ ) {
```

```
            var elem = checkSet[i];
            if ( elem ) {
                //获得 elem 的父节点
                var parent = elem.parentNode;
                //如果父节点名称为 part 值时，则在 checkSet[i]上赋值父节点，否则赋值 false
                checkSet[i] = parent.nodeName === part ? parent : false;
            }
        }
    //当 part 为非单词字符时，如$(".red > input")，part 为“.red”
    } else {
        //遍历检测结果集
        for ( var i = 0, l = checkSet.length; i < l; i++ ) {
            var elem = checkSet[i];
            if ( elem ) {
                checkSet[i] = isPartStr ?
                    elem.parentNode :
                    elem.parentNode === part;
            }
        }
        //如果不是标签，则调用 Sizzle.filter()函数筛选结果集
        if ( isPartStr ) {
            Sizzle.filter( part, checkSet, true );
        }
    }
},
//匹配 ancestor descendant 选择器
"": function(checkSet, part, isXML){
    var doneName = done++, checkFn = dirCheck;
    if ( !part.match(/\W/) ) {
        var nodeCheck = part = isXML ? part : part.toUpperCase();
        checkFn = dirNodeCheck;
    }
    checkFn("parentNode", part, doneName, checkSet, nodeCheck, isXML);
},
//匹配 prev ~ siblings 选择器
"~": function(checkSet, part, isXML){
    var doneName = done++, checkFn = dirCheck;
    if ( typeof part === "string" && !part.match(/\W/) ) {
        var nodeCheck = part = isXML ? part : part.toUpperCase();
        checkFn = dirNodeCheck;
    }
    checkFn("previousSibling", part, doneName, checkSet, nodeCheck, isXML);
}
    }
}
```

在匹配 **ancestor descendant** 选择器和 **prev ~ siblings** 选择器时，用到了两个内部函数，它们分别是 **dirNodeCheck()**和 **dirCheck()**函数，其代码如下所示。

```
//检测节点型操作符函数
```

```
function dirNodeCheck( dir, cur, doneName, checkSet, nodeCheck, isXML ) {
    var sibDir = dir == "previousSibling" && !isXML;
    //遍历结果集
    for ( var i = 0, l = checkSet.length; i < l; i++ ) {
        var elem = checkSet[i];
        if ( elem ) {    //如果结果集中存在该节点
            //如果节点为元素，且操作符为"previousSibling"
            if ( sibDir && elem.nodeType === 1 ){
                elem.sizcache = doneName;
                elem.sizset = i;
            }
            elem = elem[dir];
            var match = false;
            while ( elem ) {
                if ( elem.sizcache === doneName ) {
                    match = checkSet[elem.sizset];
                    break;
                }
                if ( elem.nodeType === 1 && !isXML ){
                    elem.sizcache = doneName;
                    elem.sizset = i;
                }
                if ( elem.nodeName === cur ) {
                    match = elem;
                    break;
                }
                elem = elem[dir];
            }
            checkSet[i] = match;
        }
    }
}
//检测操作符函数
function dirCheck( dir, cur, doneName, checkSet, nodeCheck, isXML ) {
    var sibDir = dir == "previousSibling" && !isXML;
    for ( var i = 0, l = checkSet.length; i < l; i++ ) {
        var elem = checkSet[i];
        if ( elem ) {
            if ( sibDir && elem.nodeType === 1 ) {
                elem.sizcache = doneName;
                elem.sizset = i;
            }
            elem = elem[dir];
            var match = false;
            while ( elem ) {
                if ( elem.sizcache === doneName ) {
                    match = checkSet[elem.sizset];
                    break;
                }
                if ( elem.nodeType === 1 ) {
                    if ( !isXML ) {
                        elem.sizcache = doneName;
                        elem.sizset = i;
```

```
                }
                if ( typeof cur !== "string" ) {
                    if ( elem === cur ) {
                        match = true;
                        break;
                    }
                } else if ( Sizzle.filter( cur, [elem] ).length > 0 ) {
                    match = elem;
                    break;
                }
            }
            //由于这里 dir 为 previousSibling，所以这里利用循环不断得到 elem 的前一个节点，
            //并且赋值 checkSet 数组
            elem = elem[dir];
        }
        checkSet[i] = match;
    }
  }
}
```

3.3.3 子元素选择器

子元素选择器就是通过当前匹配的元素选择该元素包含的特定子元素。子元素选择器主要包括 4 种类型，说明如表 3.12 所示。

表 3.12 子元素选择器

选 择 器	说 明
:nth-child	匹配其父元素下的第 N 个子或奇偶元素
:first-child	匹配第一个子元素。 :first 选择器只匹配一个元素，而:first-child 选择符将为每个父元素匹配一个子元素
:last-child	匹配最后一个子元素。 :last 只匹配一个元素，而:last-child 选择符将为每个父元素匹配一个子元素
:only-child	如果某个元素是父元素中惟一的子元素，将会被匹配；如果父元素中含有其他元素，将不会被匹配

注意：3.4.1 节将介绍:eq(index)选择器，该选择器只能够匹配一个元素，而:nth-child 能够为每一个父元素匹配子元素。:nth-child 是从 1 开始的，而:eq()是从 0 算起的。下面的表达式都是可以使用的。

```
:nth-child(even)        //匹配偶数位元素
:nth-child(odd)         //匹配奇数位元素
:nth-child(3n)          //匹配第 3 个及其后面间隔 3 的每个元素
:nth-child(2)           //匹配第 2 个元素
:nth-child(3n+1)        //匹配第 1 个及其后面间隔 3 的每个元素
:nth-child(3n+2)        //匹配第 2 个及其后面间隔 3 的每个元素
```

例如，下面的示例分别利用子元素选择器匹配不同位置上的 li 元素，并为其设计不同的样式，演示效果如图 3.8 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("ul li:first-child").css("color", "red");
    $("ul li:last-child").css("color", "blue");
    $("ul li:nth-child(1)").css("background", "#ff6");
    $("ul li:nth-child(2n)").css("background", "#6ff");
})
</script>

<ul>
    <li>列表 1</li>
    <li>列表 2</li>
    <li>列表 3</li>
    <li>列表 4</li>
    <li>列表 5</li>
    <li>列表 6</li>
</ul>
```

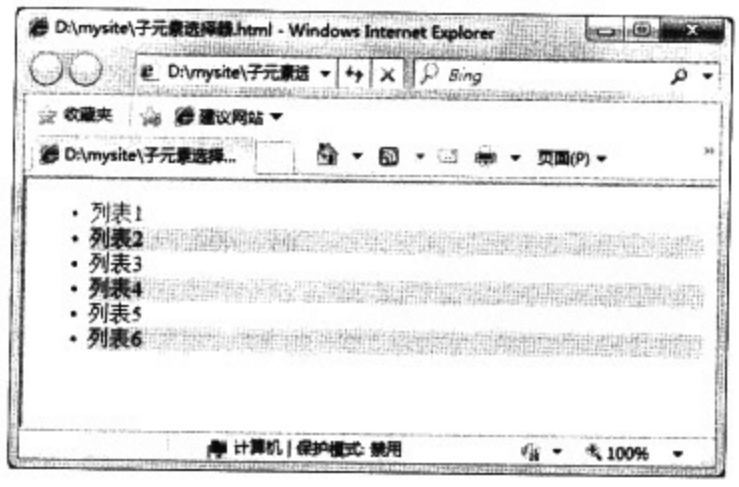


图 3.8 子元素选择器的演示效果

3.3.4 子元素选择器的实现方法

首先，jQuery 在 Expr.match 对象中定义了 CHILD 正则表达式匹配模式/:(only|nth|last|first)-child(?:\((even|odd|[\dn+-]*)\))?/，利用该模式寻找选择器表达式中的子元素类型，实现代码如下。

```
var Expr = Sizzle.selectors = {
    order: [ "ID", "NAME", "TAG" ],
    match: {
        ID: /#((?:[\w\u00c0-\uFFFF_-]|\\.)+)/,
        CLASS: /\.((?:[\w\u00c0-\uFFFF_-]|\\.)+)/,
        NAME: /\[name=['"]*((?:[\w\u00c0-\uFFFF_-]|\\.)+)['"]*\]/,
```

```
        ATTR: /\[\s*((?:[\w\u00c0-\uFFFF_-]|\\.)+)\s*(?:(\S?=)\s*(['"]*)(.*?)\3|)\s*\]/,
        TAG: /^((?:[\w\u00c0-\uFFFF\*_-]|\\.)+)/,
        CHILD: /:(only|nth|last|first)-child(?:\((even|odd|[\dn+-]*)\))?/,
        POS: /:(nth|eq|gt|lt|first|last|even|odd)(?:\((\d*)\))?(?=[^-]|$)/,
        PSEUDO: /:((?:[\w\u00c0-\uFFFF_-]|\\.)+)(?:\((['"]*)((?:\([^\)]+\)|[^\2\(\)]*)+)\2\))?/
    }
}
```

然后，在 Sizzle.filter()过滤器函数中调用该正则表达式匹配到子元素选择器的特征字符，并调用 Expr.preFilter 对象中包含的 CHILD 方法，代码如下所示。最后把处理所得的匹配元素封装到 jQuery 对象中返回。

```
if ( Expr.preFilter[ type ] ) {
    match = Expr.preFilter[ type ]( match, curLoop, inplace, result, not, isXMLFilter );
    if ( !match ) {
        anyFound = found = true;
    } else if ( match === true ) {
        continue;
    }
}
```

CHILD()方法位于 Expr.preFilter 对象中，详细代码如下所示。该代码的详细讲解请读者参阅第 8.1 节。

```
var Expr = Sizzle.selectors = {
    preFilter: {
        CHILD: function(match){
            if ( match[1] == "nth" ) {
                // parse equations like 'even', 'odd', '5', '2n', '3n+2', '4n-1', '-n+6'
                var test = /(-?)(\d*)n((?:\+|-)?\d*)/.exec(
                    match[2] == "even" && "2n" || match[2] == "odd" && "2n+1" ||
                    !/\D/.test( match[2] ) && "0n+" + match[2] || match[2]);
                // calculate the numbers (first)n+(last) including if they are negative
                match[2] = (test[1] + (test[2] || 1)) - 0;
                match[3] = test[3] - 0;
            }
            // TODO: Move to normal caching system
            match[0] = done++;
            return match;
        },
    }
}
```

3.4 过滤选择器

过滤选择器主要是通过特定的过滤表达式来筛选特殊需求的 DOM 元素，过滤选择器的语

法形式与 CSS 的伪类选择器的语法格式相同，即以冒号作为前缀标识符。根据需求的不同，过滤选择器又可以分为定位过滤器、内容过滤器和可见过滤器。下面对它们分别进行介绍。

3.4.1 定位过滤器

定位过滤器主要是根据编号和排位筛选特定位置上的元素，或者过滤掉特定元素。定位过滤器详细说明如表 3.13 所示。

表 3.13 定位过滤器

选 择 器	说 明
:first	匹配找到的第一个元素。例如，$("tr:first")表示匹配表格的第一行
:last	匹配找到的最后一个元素。例如，$("tr:last")表示匹配表格的最后一行
:not	去除所有与给定选择器匹配的元素。注意，在 jQuery 1.3 中，已经支持复杂选择器了，如:not(div a)和:not(div,a)。例如，$("input:not(:checked)")可以匹配所有未选中的 input 元素
:even	匹配所有索引值为偶数的元素，从 0 开始计数。例如，$("tr:even")可以匹配表格的 1、3、5 行(即索引值 0、2、4...)
:odd	匹配所有索引值为奇数的元素，从 0 开始计数。例如，$("tr:odd")可以匹配表格的 2、4、6 行(即索引值 1、3、5...)
:eq	匹配一个给定索引值的元素，从 0 开始计数。例如，$("tr:eq(0)")可以匹配第一行表格行
:gt	匹配所有大于给定索引值的元素，从 0 开始计数。例如，$("tr:gt(0)")可以匹配第二行及其后面行
:lt	匹配所有小于给定索引值的元素。例如，$("tr:gt(1)")可以匹配第 1 行及其后面行
:header	匹配如 h1、h2、h3 之类的标题元素
:animated	匹配所有正在执行动画效果的元素

例如，在下面这个示例中，我们分别借助基本过滤器，为表格中不同行设置不同的显示样式，演示效果如图 3.9 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("tr:first").css("color", "red");          //设置第一行字体为红色
    $("tr:eq(0)").css("font-size", "20px");  //设置第一行字体大小为 20 像素
    $("tr:last").css("color", "blue");          //设置最后一行字体为蓝色
    $("tr:even").css("background", "#ffd");  //设置偶数行背景色
    $("tr:odd").css("background", "#dff");   //设置奇数行背景色
    $("tr:gt(3)").css("font-size", "12px");  //设置从第 5 行开始所有行的字体大小
    $("tr:lt(4)").css("font-size", "14px");  //设置从第 1 行到第 4 行字体大小
})
</script>
```

```
<table>
    <tr>
        <th>选择器</th>
        <th>说明</th>
    </tr>
    <tr>
        <td>:first</td>
        <td>匹配找到的第一个元素。例如，$(&quot;tr:first&quot;)表示匹配表格的第一行 </td>
    </tr>
    <tr>
        <td>:last</td>
        <td>匹配找到的最后一个元素。例如，$(&quot;tr:last&quot;)表示匹配表格的最后一行 </td>
    </tr>
    <tr>
        <td>:not</td>
        <td>去除所有与给定选择器匹配的元素。注意，在jQuery 1.3 中，已经支持复杂选择器了，如:not(div a)
和:not(div,a)。例如,$(&quot;input:not(:checked)&quot;)可以匹配所有未选中的 input 元素 </td>
    </tr>
    <tr>
        <td>:even</td>
        <td>匹配所有索引值为偶数的元素，从 0 开始计数。例如，$(&quot;tr:even&quot;)可以匹配表格
的1、3、5行(即索引值0、2、4...)
</td>
    </tr>
    <tr>
        <td>:odd</td>
        <td>匹配所有索引值为奇数的元素，从0开始计数。例如，$(&quot;tr:odd&quot;)可以匹配表格的
2、4、6行(即索引值1、3、5...)
</td>
    </tr>
    <tr>
        <td>:eq</td>
        <td>匹配一个给定索引值的元素，从0开始计数。例如，$(&quot;tr:eq(0)&quot;)可以匹配第一行
表格行
</td>
    </tr>
    <tr>
        <td>:gt</td>
        <td>匹配所有大于给定索引值的元素，从0开始计数。例如，$(&quot;tr:gt(0)&quot;)可以匹配第
二行及其后面行
    </td>
    </tr>
    <tr>
        <td>:lt</td>
        <td>匹配所有小于给定索引值的元素。例如，$(&quot;tr:gt(1)&quot;)可以匹配第1行及其后面行
</td>
    </tr>
    <tr>
        <td>:header</td>
        <td>匹配如 h1、h2、h3之类的标题元素
</td>
    </tr>
    <tr>
```

```
        <td>:animated</td>
        <td>匹配所有正在执行动画效果的元素
</td>
    </tr>
</table>
```

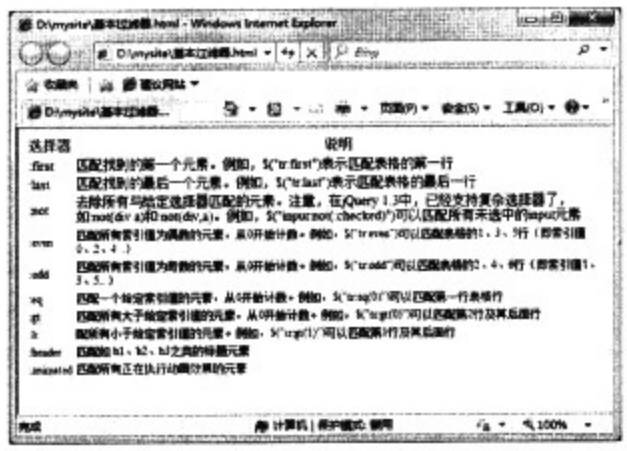


图 3.9　定位过滤器的演示效果

3.4.2　定位过滤器的实现方法

jQuery 在 Expr.setFilters 对象中收集了各种定位过滤器的表达式算法，代码如下所示。

```
var Expr = Sizzle.selectors = {
    setFilters: {
        first: function(elem, i){      //:first 选择器，如果是第一个元素，则返回 true
            return i === 0;
        },
        last: function(elem, i, match, array){ //:last 选择器，如果是最后一个元素，则返回 true
            return i === array.length - 1;
        },
        even: function(elem, i){      //:even 选择器，如果下标值是 2 的倍数，则返回 true
            return i % 2 === 0;
        },
        odd: function(elem, i){      //:odd 选择器，如果下标值不是 2 的倍数，则返回 true
            return i % 2 === 1;
        },
        lt: function(elem, i, match){      //:lt 选择器，如果下标值小于 0，则返回 true
            return i < match[3] - 0;
        },
        gt: function(elem, i, match){      //:gt 选择器，如果下标值大于 0，则返回 true
            return i > match[3] - 0;
        },
        nth: function(elem, i, match){      //:nth 选择器，如果下标值等于某个值，则返回 true
            return match[3] - 0 == i;
        },
        eq: function(elem, i, match){      //:eq 选择器，如果下标值等于某个值，则返回 true
            return match[3] - 0 == i;
        }
    }
}
```

然后在 Expr.filter 对象的 POS()函数中调用该对象集合，根据所设置的定位过滤器表达式，调用 filter()函数匹配对应的元素，并返回 jQuery 对象。

```
var Expr = Sizzle.selectors = {
    filter: {
        POS: function(elem, match, i, array){
            var name = match[2], filter = Expr.setFilters[ name ];
            if ( filter ) {
                return filter( elem, i, match, array );
            }
        }
    }
}
```

3.4.3　内容过滤器

内容过滤器主要根据匹配元素所包含的子元素或者文本内容进行过滤。具体主要包括 4 种内容过滤器，说明如表 3.14 所示。

表 3.14　内容过滤器

选 择 器	说　明
:contains	匹配包含给定文本的元素。例如，$("div:contains('图片')")匹配所有包含"图片"的 div 元素
:empty	匹配所有不包含子元素或者文本的空元素
:has	匹配含有选择器所匹配的元素的元素。例如，$("div:has(p)")匹配所有包含 p 元素的 div 元素
:parent	匹配含有子元素或者文本的元素

例如，在下面这个示例中，分别借助内容过滤器选择文档中特定的内容元素，然后对其进行控制，演示效果如图 3.10 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("li:empty").text("空内容");    //匹配空 li 元素
    $("div ul:parent").css("background", "#ff1");    //匹配 div 包含的 ul 元素中含有的子元素
或者文本的元素
    $("h2:contains('标题')").css("color", "red");    //标题元素中包含'标题'文本内容的
    $("p:has(span)").css("color", "blue");    //匹配可以包含 span 元素的 p 元素
})
</script>

<div>
    <h2>标题</h2>
    <p>段落文本 1</p>
    <p><span>段落文本 2</span></p>
    <ul>
        <li></li>
        <li></li>
```

```
    </ul>
</div>
```

图 3.10　内容过滤器的演示效果

3.4.4　内容过滤器的实现方法

jQuery 在 Expr.filters 对象中收集了各种内容过滤器的表达式算法，代码如下所示。

```
var Expr = Sizzle.selectors = {
    filters: {
        parent: function(elem){    //:parent 选择器，如果存在第一个子元素，则返回 true
            return !!elem.firstChild;
        },
        empty: function(elem){  //:empty 选择器，如果不存在第一个子元素，则返回 true
            return !elem.firstChild;
        },
        has: function(elem, i, match){ //:has 选择器，调用 Sizzle()函数检测指定的表达式所匹配的元素是否存在，如果存在，则返回 true
            return !!Sizzle( match[3], elem ).length;
        }
    }
}
```

其中当!elem.firstChild(即 elem 元素)不包含子节点或者文本元素时，empty 返回真；当!!elem.firstChild(即 elem 元素)包含子节点或者文本元素时，parent 返回真；在:has 选择器中，match[3]为紧跟在 has 后面含有的元素，如$("div:has(p)")中的 p；!!Sizzle(match[3], elem).length 获得 match[3]元素中包含在 elem 中的元素个数，如果个数大于 1，则 has 返回真。

然后在 Expr.filter 对象的 PSEUDO()函数中调用该对象集合，根据所设置的定位过滤器表达式，调用 filter()函数匹配对应的元素，并返回 jQuery 对象。其实现代码如下。

```
var Expr = Sizzle.selectors = {
    filter: {
        PSEUDO: function(elem, match, i, array){
            var name = match[1], filter = Expr.filters[ name ];
            if ( filter ) {    //匹配:parent、:empty 和:has 选择器
                return filter( elem, i, match, array );
```

```
            } else if ( name === "contains" ) {    //匹配: contains 选择器
                // textContext 在 FF 下和 innerText 在 IE 下的属性是等效的,
                // match[3]得到的是紧跟在 contains 后面包含的字符串,
                //当 elem 元素的文本内容包含 contains 包含的关键字时, 返回 true
                return (elem.textContent || elem.innerText || "").indexOf(match[3]) >= 0;
            } else if ( name === "not" ) {
                var not = match[3];
                for ( var i = 0, l = not.length; i < l; i++ ) {
                    if ( not[i] === elem ) {
                        return false;
                    }
                }
                return true;
            }
        }
    }
}
```

上面代码的关键在于 (elem.textContent || elem.innerText || "").indexOf(match[3]) >= 0; 的布尔值判断，当返回真时，elem 元素为匹配的元素。

3.4.5 可见过滤器

可见过滤器就是根据元素的可见或者隐藏来进行匹配，详细说明如表 3.15 所示。

表 3.15 可见过滤器

选 择 器	说 明
:hidden	匹配所有不可见元素，或者 type 为 hidden 的元素
:visible	匹配所有的可见元素

例如，在下面的示例中分别设置奇数位 p 和偶数位 p 的字体颜色，如果奇数位 p 元素被隐藏，则通过 p:hidden 过滤器匹配它们，并把它们显示出来。

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("p:odd").hide();     //隐藏奇数位 p 元素
    $("p:odd").css("color", "red");    //设置奇数位 p 元素的字体颜色为红色
    $("p:visible").css("color", "blue");    //设置偶数位 p 元素的字体颜色为蓝色
    $("p:hidden").show();    //显示奇数位 p 元素
})
</script>

<p>段落文本 1</p>
<p>段落文本 2</p>
<p>段落文本 3</p>
```

```
<p>段落文本 4</p>
```


3.4.6 可见选择器的实现方法

jQuery 专门为:hidden、:visible 和:animated 三个伪选择器定制了三个独立的公共函数，然后在 filter()中调用这些函数来匹配特殊的过滤器。具体代码如下所示。

```
Sizzle.selectors.filters.hidden = function(elem){
    return elem.offsetWidth === 0 || elem.offsetHeight === 0;        //匹配:hidden 选择器
};
Sizzle.selectors.filters.visible = function(elem){
    return elem.offsetWidth > 0 || elem.offsetHeight > 0;    //匹配:visible 选择器
};
Sizzle.selectors.filters.animated = function(elem){  //匹配:animated 选择器
    return jQuery.grep(jQuery.timers, function(fn){
        return elem === fn.elem;
    }).length;
};
```

当 elem 元素的 CSS 属性 offsetWidth 为 0，或者 offsetHeight 为 0 时，hidden 返回真；当 elem 元素的 CSS 属性 offsetWidth 不为 0，或者 offsetHeight 不为 0 时，visible 返回真。jQuery 通过布尔值来判断元素是否显示。

3.5 属性选择器

属性过滤器主要是把元素的属性及其值作为过滤的条件，来匹配对应的 DOM 元素。用好属性过滤器，能够增强读者对于 HTML 文档的控制能力。下面我们将重点介绍属性过滤器的使用，以及 jQuery 是如何实现属性选择器的设计。

3.5.1 使用属性选择器

属性选择器一般都是以中括号作为起止分界符的，如[attribute]。为了便于与其他选择器进行区分，jQuery 定义了 7 类属性选择器，具体说明如表 3.16 所示。

表 3.16 属性选择器

选 择 器	说 明
[attribute]	匹配包含给定属性的元素。 注意，在 jQuery 1.3 版本中，前导的@符号已经被废除，如果想要兼容最新版本，只需要简单去掉@符号即可。例如，$("div[id]")表示查找所有含有 id 属性的 div 元素

续表

选 择 器	说 明
[attribute=value]	匹配属性等于特定值的元素。属性值的引号在大多数情况下是可选的，如果属性值中包含"]"时，需要加引号用以避免冲突。 例如，$("input[name='text']")表示查找所有 name 属性值是'text'的 input 元素
[attribute!=value]	匹配所有不含有指定的属性，或者属性不等于特定值的元素。该选择器等价于:not([attr=value]) 要匹配含有特定属性但不等于特定值的元素，则可以使用[attr]:not([attr=value])。 例如，$("input[name!='text']") 表示查找所有 name 属性值不是'text'的 input 元素
[attribute^=value]	匹配给定的属性是以某些值开始的元素。 例如，$("input[name^='text']") 表示匹配所有 name 属性值以'text'开始的 input 元素
[attribute$=value]	匹配给定的属性是以某些值结束的元素。 例如，$("input[name$='text']")表示匹配所有 name 属性值以'text'结束的 input 元素
[attribute*=value]	匹配给定的属性是以包含某些值的元素。 例如，$("input[name*='text']")表示匹配所有 name 属性值包含'text'字符串的 input 元素
[selector1][selector2][selectorN]	复合属性选择器，需要在同时满足多个条件时使用。 例如，$("input[name*='text'] [id]")表示匹配所有 name 属性值包含'text'字符串，且包含了 id 属性的 input 元素

下面看一个示例，在这个示例中将根据超链接文件的类型，分别为不同类型的文件添加类型文件图标。页面初始化前的效果如图 3.11 所示，执行脚本之后，则显示效果如图 3.12 所示。

```
<style type="text/css">
a img {
    border:none;
}
</style>
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    var a1 = $("a[href$='.pdf']");
    a1.html(a1.attr("href") + " <img src='images/pdf.gif' />");
    var a2 = $("a[href$='.rar']");
    a2.html(a2.attr("href") + " <img src='images/rar.gif' />");
    var a3 = $("a[href$='.jpg'],a[href$='.bmp'],a[href$='.gif'],a[href$='.png']");
```

```
        a3.html(a3.attr("href") + " <img src='images/jpg.gif' />");
        var a4 = $("a[href^='http:']");
        a4.html(a4.attr("href") + " <img src='images/html.gif' />");
    })
    </script>

    <a href="1.pdf">超链接</a><br />
    <a href="2.rar">超链接</a><br />
    <a href="3.jpg">超链接</a><br />
    <a href="4.bmp">超链接</a><br />
    <a href="5.gif">超链接</a><br />
    <a href="6.png">超链接</a><br />
```

然后，通过 Expr. preFilter[ATTR]函数预处理属性选择器字符串中的特殊格式，并返回正则表达式匹配结果的数组。其实现代码如下所示。

```
var Expr = Sizzle.selectors = {
    preFilter: {
        ATTR: function(match, curLoop, inplace, result, not, isXML){
            var name = match[1].replace(/\\/g, "");
            //映射 class 和 for 属性
            if ( !isXML && Expr.attrMap[name] ) {
                match[1] = Expr.attrMap[name];
            }
            //为兄弟选择器添加空格
            if ( match[2] === "~=" ) {
                match[4] = " " + match[4] + " ";
            }
            //返回匹配的数组
            return match;
        }
    }
}
```

最后，通过 Sizzle.filter 方法，获得 ATTR 的正则匹配，然后在过滤器中调用 Expr.filter[ATTR]函数匹配属性选择器字符串的基本特征，以中括号字符为起止分界符。如果匹配对应的字符串，则返回 true，即选择当前元素，否则返回 false，表示放弃选择当前元素。具体实现代码如下所示。

```
var Expr = Sizzle.selectors = {
    filter: {
        ATTR: function(elem, match){
            var name = match[1],
                result = Expr.attrHandle[ name ] ?
                    Expr.attrHandle[ name ]( elem ) :
                    elem[ name ] != null ?
                        elem[ name ] :
                        elem.getAttribute( name ),
                value = result + "",
                type = match[2],
                check = match[4];
            //返回匹配结果，以布尔值表示，true 表示选择，false 表示不选择
            return result == null ?
                type === "!=" :
                type === "=" ?
                value === check :
                type === "*=" ?
                value.indexOf(check) >= 0 :
                type === "~=" ?
                (" " + value + " ").indexOf(check) >= 0 :
                !check ?
                value && result !== false :
```

```
                type === "!=" ?
                value != check :
                type === "^=" ?
                value.indexOf(check) === 0 :
                type === "$=" ?
                value.substr(value.length - check.length) === check :
                type === "|=" ?
                value === check || value.substr(0, check.length + 1) === check + "-" :
                false;
        }
    }
}
```

其中 value 相当于“newsletter”，check 相当于“new”。

各种属性选择器的说明如下。

- “!=”和“=”判断 value === check; 的布尔值，即 value 是否等于 check。
- “^=”取 value.index(check) === 0; 的布尔值，即 check 的字符串是否在 value 的开头。
- “$=”取 value.substr(value.length - check.length) === check; 的布尔值，即 check 的字符串是否在 value 的末尾。
- “*=”取 value.index of(check) >= 0; 的布尔值，即 value 包含 check 字符串为真。

3.6 表单选择器

表单是页面中使用频率较高的元素之一，但是很多表单域都是使用 input 元素来定义的，为了方便用户灵活地操作表单，jQuery 专门定义了表单选择器，使用表单选择器可以方便地获取表单中某类表单域对象。

3.6.1 基本表单选择器

jQuery 定义了一组伪类选择器，利用它们可以获取页面中的表单类型元素，具体说明如表 3.17 所示。

表 3.17 基本表单选择器

选 择 器	说 明
:input	匹配所有 input、textarea、select 和 button 元素
:text	匹配所有单行文本框
:password	匹配所有密码框
:radio	匹配所有单选按钮
:checkbox	匹配所有复选框

续表

选 择 器	说 明
:submit	匹配所有提交按钮
:image	匹配所有图像域
:reset	匹配所有重置按钮
:button	匹配所有按钮
:file	匹配所有文件域
:hidden	匹配所有不可见元素，或者 type 为 hidden 的元素

下面我们就结合一个表单案例，演示如何使用表单选择器控制实现交互操作。表单的 HTML 结构代码如下。根据该 HTML 结构代码，生成的页面效果如图 3.13 所示。

```
<form id="test" action="" method="get">
    <input name="" type="text" value="文本域"><br />
    <input name="" type="password" value="密码域"><br />
    <input name="" type="checkbox" value="复选框">复选框<br />
    <input name="" type="radio" value="单选按钮">单选按钮<br />
    <input name="" type="image" value="图像域" src="images/pink.png"S><br />
    <input name="" type="file" value="文件域"><br />
    <input name="" type="hidden" value="隐藏域"><br />
    <input name="" type="button" value="普通按钮"><br />
    <input name="" type="submit" value="提交按钮"><br />
    <input name="" type="reset" value="重置按钮"><br />
</form>
```

然后，使用表单选择器快速选择这些表单域，并修改它们的 value 属性值，对应的代码如下，显示效果如图 3.14 所示。

图 3.13　设计初的表单效果

图 3.14　修改后的表单效果

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
$(function(){
    $("#test :text").val("修改后的文本域");
    $("#test :password").val("修改后的密码域");
```

```
    $("#test :checkbox").val("修改后的复选框");
    $("#test :radio").val("修改后的单选按钮");
    $("#test :image").val("修改后的图像域");
    $("#test :file").val("修改后的文件域");
    $("#test :hidden").val("修改后的隐藏域");
    $("#test :button").val("修改后的普通按钮");
    $("#test :submit").val("修改后的提交按钮");
    $("#test :reset").val("修改后的重置按钮");
})
</script>
```

3.6.2 高级表单选择器

除了简单的表单域选择器外，jQuery 还根据表单域中特有的属性定义了 4 个表单属性选择器，这些选择器与基本表单选择器不同，它们可以选择任何类型的表单域，因为它们主要根据表单属性来进行选择。具体说明如表 3.18 所示。

表 3.18 高级表单选择器

选 择 器	说 明
:enabled	匹配所有可用元素
:disabled	匹配所有不可用元素
:checked	匹配所有选中的被选中元素(复选框、单选框等，不包括 select 中的 option)
:selected	匹配所有选中的 option 元素

下面我们就结合一个表单案例，演示如何使用表单的属性选择器控制实现交互操作。表单的 HTML 结构代码如下。根据该 HTML 结构代码，生成的页面效果如图 3.15 所示。

```
<form id="test" action="" method="get">
    <input name="" type="text" disabled="disabled" value="文本域"><br />
    <input name="" type="text" disabled="disabled"  value="文本域"><br />
    <input name="" type="text"  value="文本域"><br />
    <input name="" type="checkbox" checked="checked" value="复选框">复选框<br />
    <input name="" type="radio" value="单选按钮">单选按钮<br />
    <select name="">
        <option value="1">1</option>
        <option value="1">2</option>
        <option value="1" selected="selected">3</option>
    </select>
</form>
```

然后，使用表单属性选择器快速选择这些表单域，并对表单域实施控制，对应的代码如下，显示效果如图 3.16 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript" >
```

```
$(function(){
    $("#test :disabled").val("不可用");
    $("#test :enabled").val("可用");
    $("#test :checked").removeAttr("checked");
    $("#test :selected").removeAttr("selected");
})
</script>
```

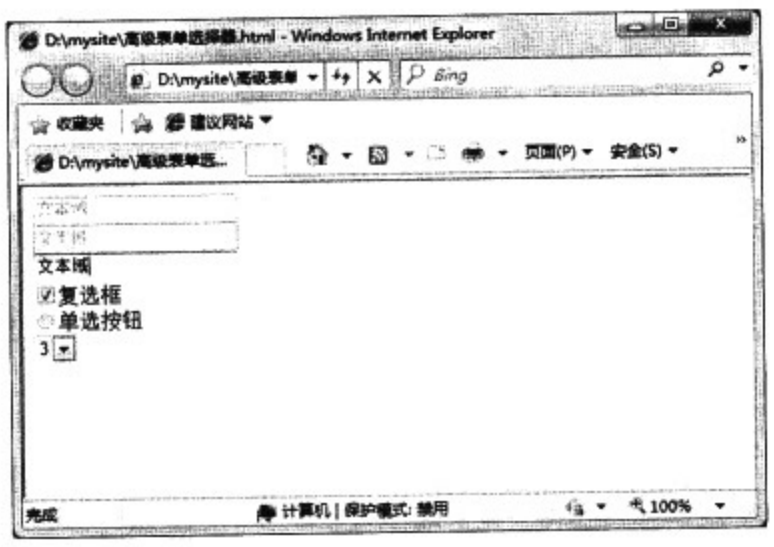


图 3.15 设计初的表单效果

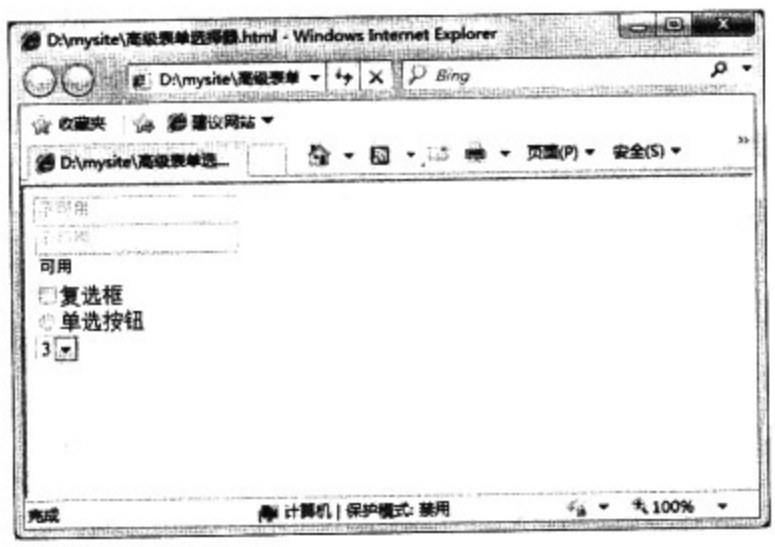


图 3.16 修改后的表单效果

3.6.3 表单选择器的实现方法

jQuery 在 Sizzle.selectors 表达式对象中定义了 filters 子对象，该子对象是一个表达式过滤函数集，主要负责过滤表单域中各种特征域。如果匹配的元素符合该特征，则返回 true，否则返回 false。

首先，jQuery 将调用下面正则表达式匹配到表单选择器字符，并通过匹配到的选择器特征字符找到它们的核心代码。

PSEUDO: /:((?:[\w\u00c0-\uFFFF_-]|\\.)+)(?:\((['"]*)((?:\([^\)]+\)|[^\2\(\)]*)+)\2\))?/

详细代码如下所示。

```
var Expr = Sizzle.selectors = {
    filters: {
        enabled: function(elem){    //: enabled选择器
            return elem.disabled === false && elem.type !== "hidden";
        },
        disabled: function(elem){    //: disabled选择器
            return elem.disabled === true;
        },
        checked: function(elem){    //: checked选择器
            return elem.checked === true;
        },
        selected: function(elem){    //: selected选择器
            // Accessing this property makes selected-by-default
```

```
        // options in Safari work properly
        elem.parentNode.selectedIndex;
        return elem.selected === true;
    },
    parent: function(elem){     //:parent选择器
        return !!elem.firstChild;
    },
    empty: function(elem){     //: empty选择器
        return !elem.firstChild;
    },
    has: function(elem, i, match){     //:has选择器
        return !!Sizzle( match[3], elem ).length;
    },
    header: function(elem){     //: header选择器
        return /h\d/i.test( elem.nodeName );
    },
    text: function(elem){     //: text选择器
        return "text" === elem.type;
    },
    radio: function(elem){     //: radio选择器
        return "radio" === elem.type;
    },
    checkbox: function(elem){     //: checkbox选择器
        return "checkbox" === elem.type;
    },
    file: function(elem){     //: file选择器
        return "file" === elem.type;
    },
    password: function(elem){     //: password选择器
        return "password" === elem.type;
    },
    submit: function(elem){     //: submit选择器
        return "submit" === elem.type;
    },
    image: function(elem){     //:image选择器
        return "image" === elem.type;
    },
    reset: function(elem){     //:reset选择器
        return "reset" === elem.type;
    },
    button: function(elem){     //: button选择器
        return "button" === elem.type || elem.nodeName.toUpperCase() === "BUTTON";
    },
    input: function(elem){     //: input选择器
        return /input|select|textarea|button/i.test(elem.nodeName);
    }
  }
}
```

elem.type 获得元素的 **type** 属性，当元素 **type** 属性等于相应的值时，返回相应的布尔值。

如果为真，最后返回匹配的 jQuery 对象。其中，enabled 对应 elem 的 disabled 属性为 false，并且 elem 的 type 属性为 hidden；disabled 对应 elem 的 disabled 属性为 true；checked 对应 elem 的 checked 属性为 true；selected 对应 elem 的 selected 属性为 true。

然后，jQuery 在 Sizzle.selectors 表达式对象中定义了 filter 子对象，该子对象如果匹配的选择器类型是伪类选择器，则调用该对象集合包含的 PSEUDO()方法。PSEUDO()方法首先调用 Expr.filters 集合对象，查询选择器的名称，并根据选择器的名称确定所要执行的操作。详细代码如下所示。

```
var Expr = Sizzle.selectors = {
    filter: {
        PSEUDO: function(elem, match, i, array){
            var name = match[1], filter = Expr.filters[ name ];
            //如果是表单类型选择器，则返回调用该表单类型选择器对应的
            //Expr.filters[ name ]()函数的值，返回 true 或者 false
            /最后 Sizzle 选择器和过滤器再根据这个返回值，逐一筛选匹配的表单域元素
            if ( filter ) {
                return filter( elem, i, match, array );
            } else if ( name === "contains" ) {
                return (elem.textContent || elem.innerText || "").indexOf(match[3]) >= 0;
            } else if ( name === "not" ) {
                var not = match[3];
                for ( var i = 0, l = not.length; i < l; i++ ) {
                    if ( not[i] === elem ) {
                        return false;
                    }
                }
                return true;
            }
        }
    }
}
```

第4章 文档对象的操作及其高效实践

简单设想一下，当用户单击某个皮肤切换按钮时，页面也许在瞬间会发生翻天覆地的变化。这其中 JavaScript 起到决定性的作用。对于初学者来说，可能存在这样的疑问：JavaScript 是在单击时才执行全部任务，还是早已做好潜伏，只等单击瞬间扣动扳机？要想弄清楚上面的问题，读者就需要了解 JavaScript 的解析机制，理解文档对象模型。

浏览器中都包含一个 JavaScript 引擎，它相当于解析 JavaScript 代码的发动机。这个 JavaScript 引擎在浏览器显示页面之前，也就是在加载网页内容时，会做一个热身运动，即所谓的预处理，把 JavaScript 代码简单处理一下，检索所有的对象和变量，以便后期能够快速处理。同时，浏览器还包含一个渲染引擎，专门负责解析 HTML 结构和 CSS 样式。渲染引擎能够根据一定的标准把 HTML 结构的所有节点检索出来，定义对应的对象和属性与之映射。这样在后期处理时，JavaScript 就不需要慌慌张张到文档中寻找对应的标签，饥不择食地进行处理，而是直接通过文档映射的对象快速操作，最后把处理的结果反射给文档，再呈现出来。

4.1 DOM 标准

DOM 是 Document Object Model 短语的缩写，中文翻译为文档对象模型。根据 W3C DOM 规范(http://www.w3.org/DOM/)，DOM 是一种与浏览器、平台和语言无关的接口，使用该接口可以访问页面其他的标准组件。

4.1.1 分解 DOM

通俗地说，DOM 提供了一套标准，方便不同类型的浏览器以及不同脚本的语言轻松访问页面中所有标准组件。例如，在 DOM 标准化以前，IE 与其他浏览器访问页面对象的方法是不同的，JavaScript 与 JScript 脚本语言的语法、词法标准也存在差异，导致 Web 设计师为了实现同一个功能需要为不同浏览器设计不同的实现方法。现在有了 DOM 作为统一的接口，访问文档内容、操作文档对象的方法就可以统一了，Web 设计师不需要再顾虑用户所使用的浏览器类型。DOM 缩写词可以分解如下。

- D 是 Document 一词的简写，表示文档，它是 DOM 的表现层。
  虽是表现，但是基础，若没有文档，DOM 也是无源之水，当创建文档并在浏览器中显示网页时，DOM 在幕后悄然而生，它将根据文档结构创建文档对象。
- O 是 Object 一词的简写，表示对象，它是 DOM 的逻辑层。
  现实中的对象是有形的，而 DOM 中的对象是无形的。任何概念都可以被视为对象，更准确地讲，它是一种逻辑对象，是用来完成特定任务的概念，相当于抽象的称谓。
- M 是 Model 一词的简写，表示模型，它是 DOM 的交互层。
  模型比较模糊，如果说是地图、索引、列表等会更准确地理解。简单地说，Model 就是一种约定俗成的规则，用来描述文档的结构，方便访问和交互。

4.1.2 HTML DOM

DOM 是一套对文档内容进行抽象和概念化的方法。在早期的 JavaScript 版本中就已经提供了初级的 DOM，它向程序员提供了对 Web 文档的某些实际内容进行查询和操作的方法。例如，图像和表单等是网页中的主要内容，程序员也很容易明白这些名词概念，于是 JavaScript 预定义了 images 和 forms 等关键字，用来指代这些网页对象，于是早期的程序员就可以在 JavaScript 脚本中访问页面中的图像和表单，实现代码如下。

```
document.images[2]      //访问文档中第 2 个图像
document.forms["login"]     //访问文档中名称为 login(name 属性值)的表单
```

现在人们把这种初级方法统称为 0 级 DOM(DOM Level0)，或者称为 HTML DOM(网页文档对象模型)。在还没有形成统一的标准之前，0 级 DOM 大行其道，常用于访问和操作网页内图像、链接、表单和多媒体对象等。

Netscape 公司和微软公司在各自的浏览器中很早就支持 0 级 DOM，但是方法略有差异。1997 年 6 月，Netscape 公司发布了 Navigator 4 版本浏览器，4 个月后微软发布了 IE 4 版本浏览器。这两种浏览器都对早期版本进行了全面升级，增强了 DOM 功能，可以允许 JavaScript 借助 DOM 完成更多的任务。在这个时候，网页设计人员开始熟悉了 DHTML。

DHTML 是 dynamic HTML 短语的缩写，中文翻译为动态 HTML。实际上 DHTML 并不是一项新技术，它仅是 HTML、CSS 和 JavaScript 三种技术的混合物，其意图是如下。

- 使用 HTML 设计网页内容和结构。
- 使用 CSS 设计网页显示样式。
- 使用 JavaScript 动态控制网页显示样式。

利用 DHTML，可以轻松设计各种复杂的动画效果，但是真正把这些技术捆绑在一起的是 DOM。不过，Netscape 和微软的浏览器支持 DOM 的标准不同，这使得设计师为了实现相同的目标，需要为不同浏览器设计不同的实现方法，这自然增大了开发的难度和兼容成本。

4.1.3 DOM Core

浏览器的不兼容性给 Web 开发带来了一定的难度和不确定性。例如，Netscape 公司支持的 DOM 标准使用 layer 元素来表示层，并利用它来访问页面元素。

```
document.layers["e"]  //在Netscape Navigator浏览器下，访问页面中id为e的元素
```

而微软公司支持的 DOM 标准却使用 all 来表示文档中元素集合，要访问页面中相同元素，则应该是：

```
document.all["e"]     //在IE浏览器下访问页面中id为e的元素
```

当然，这两种在当时为主流的浏览器，各自支持的 DOM 在细节方面的差异可不止这点。DHTML 概念的推出给人们带来了希望，同时也让设计师们倍感艰辛。

这时，W3C 组织识势而为，于 1998 年 10 月推出了一份标准 DOM，即所谓的 1 级 DOM(DOM Level1)。W3C 对 DOM 的定义如下。

一个与系统平台和编程语言无关的接口，程序和脚本可以通过这个接口动态地对文档的内容、结构和样式进行访问和修改。

W3C 推出的这种标准化 DOM 得到了所有浏览器厂商的支持和响应，因为作为一种 API(应用编程接口)，作为一种统一接口。

W3C 推出的标准化 DOM，在独立性和适用范围等方面，都超出了先前的 0 级 DOM，为 Web 标准化设计奠定了基础。

W3C DOM 标准包含很多模块，每个模块下面又包含很多子模块。其中用来操纵标记文档(如 HTML 和 XML 等)的模块被称为核心模块，即所谓的 DOM Core。

DOM Core 统一了访问网页文档的方法，并为不同类型的节点对象提供了统一的操作方法和属性。JavaScript 中的 getElementById()、getElementsByTagName()、getAttribute()和 setAttribute()等方法都是 DOM Core 模块的组成部分，具体作用如下。

- getElementById()，利用标记的 id 属性值访问标记元素。
- getElementsByTagName()，利用标记的名称访问所有同名标记元素。
- getAttribute()，获取指定元素的属性值。
- setAttribute()，为指定元素的属性设置属性值。

如果使用 DOM Core 标准获取网页中 id 值为 e 的元素，则可以使用如下代码实现。

```
document.getElementById("e")    //在 DOM Core 标准下，访问页面中 id 为 e 的元素
```

关于这些技术细节和应用技巧，我们将在其他章节中不断渗透讲解。

4.1.4 DOM 文档树

jQuery 作为一种 JavaScript 库，继承并优化了 JavaScript 访问 DOM 对象的特性，使开发人员能更加方便地操作 DOM 对象。为了能够更详细地讲解 DOM 操作，我们需要构建一份网页文档，并提炼出它的文档结构，以 DOM 树形模式演示出来。HTML 结构如下。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>DOM 文档对象模型</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<body>
<h1>DOM 文档对象模型</h1>
<p>DOM 是 Document Object Model 短语的缩写，中文翻译为文档对象模型，根据 W3C DOM 规范
(http://www.w3.org/DOM/ )，DOM 是一种与浏览器、平台、语言无关的接口，使用该接口可以访问页面其他
的标准组件。</p>
<ul>
    <li>D 是 Document 一词的简写，表示文档，它是 DOM 的表现层。</li>
    <li>O 是 Object 一词的简写，表示对象，它是 DOM 的逻辑层。</li>
    <li>M 是 Model 一词的简写，表示模型，它是 DOM 的交互层。</li>
</ul>
</body>
</html>
```

每一份网页文档都可以使用 DOM 来进行描述，每一份 DOM 都会把网页看作是一棵节点树。浏览器在渲染文档时，会自动在内存中构建一份完整的 DOM 树形结构(或者说是树形结构列表)，如图 4.1 所示。

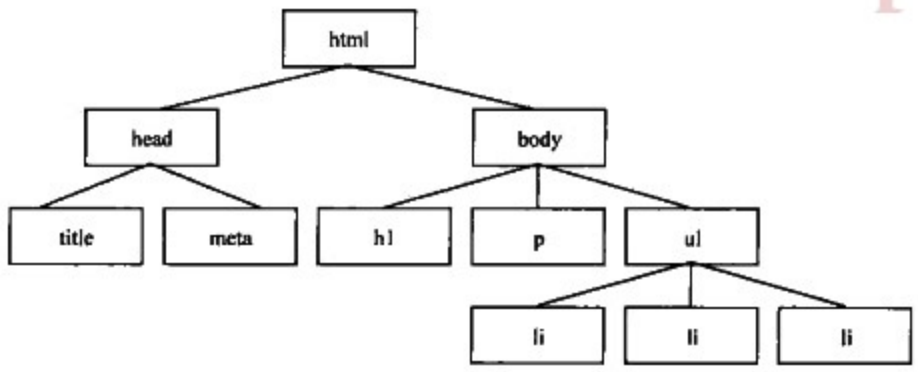


图 4.1 DOM 树形结构

而浏览器在窗口中所显示的网页内容如图 4.2 所示。

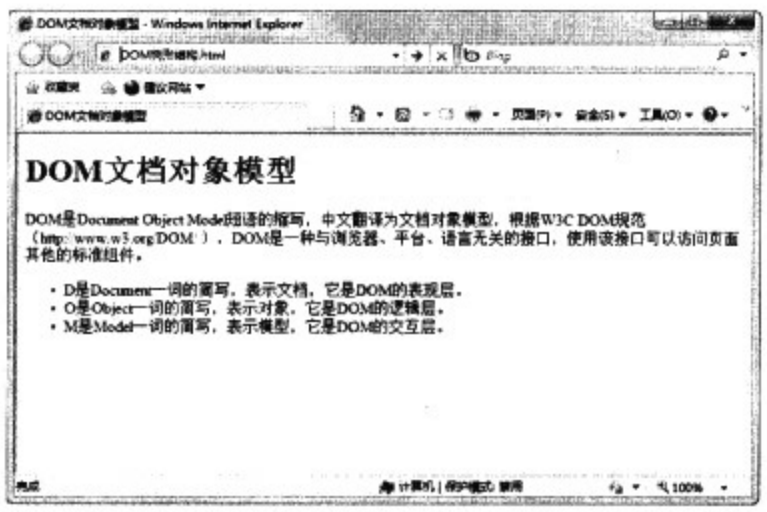


图 4.2 网页文档显示出来的效果

在下面的章节中，我们将利用这个文档结构讲解使用 jQuery 和 JavaScript 操作文档的各种方法。

4.2 创 建 节 点

节点是 DOM 结构的基础。根据 DOM 规范，节点是一个很宽泛的概念，范围包含元素、属性、文本、文档和注释等，而不仅仅指代元素或属性，网页文档中所有内容都可以归为某一类节点。但是在实际开发中，要创建动态内容，主要操作的节点包括元素、属性和文本。

4.2.1 创建元素

例如，创建一个 h1 元素，并把它作为 body 元素的子节点添加到 DOM 节点树中。

1. jQuery 实现

首先，使用 jQuery 工厂函数$()创建一个 h1 对象。格式如下。

```
$(html)
```

该函数能够根据参数 html 所传递的 HTML 标记字符串，创建一个 DOM 对象，并将该对象包装为 jQuery 对象返回。

注意：参数字符串必须符合严谨型 XHTML 结构要求，标记应该包含起始标签和结束标签。如果没有结束标签，则应该添加闭合标记，即在起始标签中添加斜线。例如，下面的字符串参数都是合法的。

```
"<h1></h1>" //合法的参数字符串
"<h1 />"    //合法的参数字符串
```

而下面的字符串参数都是非法的。

```
"<h1>"  //非法的参数字符串
"</h1>" //非法的参数字符串
```

然后，将创建的元素插入到文档中。动态创建的元素不会自动添加到文档中，需要使用其他方法把它添加到文档中。例如，可以使用 jQuery 的 append()方法把创建的 h1 元素添加到文档 body 元素节点下。

最后，使用 jQuery 实现的完整代码如下。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){//页面初始化函数
    var $h1 = $("<h1></h1>");   //创建 h1 对象集合
    $("body").append($h1); //把创建的 h1 集合内的对象添加到文档中
})
</script>
```

在浏览器中运行代码后，新创建的 h1 元素被添加到文档中，由于该元素没有包含任何文本，所以看不到任何显示效果。

2. JavaScript 实现

JavaScript 实现的思路与 jQuery 基本相同，即先创建 h1 元素对象，然后添加到文档中。使用 JavaScript 实现的完整代码如下。

```
<script type="text/javascript" >
window.onload = function(){    //页面初始化函数
    var h1 = document.createElement("h1");   //创建 h1 对象
    document.body.appendChild(h1);  //把创建的 h1 对象添加到 DOM 文档树中
}
```

```
</script>
```

createElement()是 Document 对象的方法，它能够根据参数传递的字符串创建元素对象，并返回新创建的元素对象。返回对象仅在 JavaScript 上下文中有效，如果要把它添加到文档中，还需要调用 Element 对象的 appendChild()方法实现，因此在调用 appendChild()方法之前，读者应获取添加位置的父元素，本例为 body 元素。

3. 实现方法比较

从代码输入的角度分析，jQuery 和 JavaScript 实现都用了两行代码，但 jQuery 稍显简便，仅用到复合函数$()和 jQuery 的 append()方法。而 JavaScript 需要调用 Document 对象的 createElement()方法和 Element 对象的 appendChild()方法，用户需要考虑的技术细节要多，且输入稍显麻烦。

从执行效率角度分析，两者差距明显，JavaScript 要比 jQuery 快 10 倍以上，在 IE 8 中差距会拉大到 30 倍以上，在其他主流浏览器中的执行效率差距就更大了。

4.2.2 创建文本

文本节点无法独立存在，必须包裹在元素节点内部，因此在创建文本之前，应确保新建或者选择元素节点。下面继续以上一节新创建的 h1 元素为例，在 h1 元素中添加文本。

1. jQuery 实现

使用 jQuery 实现的完整代码如下。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){//页面初始化函数
    var $h1 = $("<h1>DOM 文档对象模型</h1>");  //创建 h1 对象集合，并添加文本节点
    $("body").append($h1); //把创建的 h1 集合内的对象添加到文档中
})
</script>
```

从上面示例的代码中可以看到，创建文本节点就是在创建元素节点时，直接把文本字符串添加到元素标记字符串之中，然后再使用 append()等方法把它们添加到 DOM 文档结构树中。注意，无论 HTML 代码有多么复杂，我们都可以使用相同的方法进行创建。

2. JavaScript 实现

使用 JavaScript 实现的完整代码如下。

```
<script type="text/javascript" >
window.onload = function(){    //页面初始化函数
    var h1 = document.createElement("h1");   //创建 h1 对象
```

```
    var txt = document.createTextNode("DOM 文档对象模型");//创建文本对象
    h1.appendChild(txt);    //把文本节点增加到标题元素节点中
    document.body.appendChild(h1);  //把创建的 h1 对象添加到 DOM 文档树中
}
</script>
```

DOM 在 Document 对象中定义了 createTextNode()方法，调用该方法可以创建文本节点。其语法格式如下。

```
var txt = document.createTextNode("text");
```

该方法包含一个参数，即新建文本节点所包含的文本字符串。参数中不能够包含任何 HTML 标签，否则 JavaScript 会把这些标签作为字符串进行显示。最后返回新创建的文本节点。

新创建的文本节点不会自动增加到 DOM 结构树中，如果要把创建的文本节点添加到新建或指定的元素内，需要使用 appendChild()方法实现。在本例中，先把新创建的文本节点添加到新创建的 h1 元素节点下，最后再添加到文档根节点下。

3. 实现方法比较

从代码输入的角度分析，JavaScript 实现比较麻烦，用户需要分别创建元素节点和文本节点，然后再一步步把文本节点添加到元素节点中，最后才能够添加到 DOM 结构树中。而 jQuery 经过包装之后，与 jQuery 创建元素节点操作相同，仅需要两步操作即可快速实现。

从执行效率角度分析，两者差距进一步拉大，在 JavaScript 执行速度最快的 Safari 浏览器中，JavaScript 实现要比 jQuery 实现快 80 倍以上，这实在令人惊叹。在执行速度最慢的 IE 浏览器中，两者差距也在 10 倍以上。

4.2.3 创建属性

在 DOM 规范中，属性节点比较特殊，用户无法通过 Node 对象提供的方法遍历或者定位属性节点，必须使用 Element 对象定义的特定方法来创建和访问属性节点。

jQuery 创建属性节点与创建文本节点类似，但是 JavaScript 使用 setAttribute()方法实现创建属性节点。下面继续以前面小节中创建的 h1 元素为例，为 h1 元素添加 title 和 class 属性。

1. jQuery 实现

使用 jQuery 实现的完整代码如下。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){//页面初始化函数
    var $h1 = $("<h1 title='一级标题' class='red'>DOM 文档对象模型</h1>");
```

```
    //创建 h1 对象集合，设置两个属性，并添加文本节点
    $("body").append($h1); //把创建的 h1 集合内的对象添加到文档中
})
</script>
```

从上面示例的代码中可以看到，不管是创建元素节点、文本节点，还是创建属性节点，都可以模仿 HTML 语法格式编写 HTML 代码字符串，然后把这个字符串作为参数传递给$()函数即可，这大大降低了 DOM 操作的难度。正因为如此简单，jQuery 才引人注目，并获得大批粉丝。

2. JavaScript 实现

使用 JavaScript 实现的完整代码如下。

```
<script type="text/javascript" >
window.onload = function(){    //页面初始化函数
    var h1 = document.createElement("h1");   //创建 h1 对象
    var txt = document.createTextNode("DOM 文档对象模型");   // 创建文本对象
    h1.setAttribute("title","一级标题"); //为 h1 对象创建属性节点 title，并设置属性值
    h1.setAttribute("class","red"); //为 h1 对象创建属性节点 title，并设置属性值
    h1.appendChild(txt);    //把文本节点增加到标题元素节点中
    document.body.appendChild(h1);  //把创建的 h1 对象添加到 DOM 文档树中
}
</script>
```

DOM 在 Element 对象中定义了 setAttribute()方法，调用该方法可以创建属性节点，并设置属性节点包含的值。其语法格式如下。

```
e.setAttribute(name,value);
```

其中参数 e 表示指定的元素对象，参数 name 和 value 分别表示属性名称和属性值。属性名称和属性值必须以字符串的形式进行传递。如果元素中存在指定的属性，它的值将被刷新；如果不存在，则 setAttribute()方法将为元素创建该属性并赋值。

setAttribute()方法也可以为已经存在的元素创建属性节点。例如，在上面的示例中，我们可以把调用 setAttribute()方法的代码行放置在最后，代码如下。

```
<script type="text/javascript" >
window.onload = function(){    //页面初始化函数
    var h1 = document.createElement("h1");   //创建 h1 对象
    var txt = document.createTextNode("DOM 文档对象模型");   // 创建文本对象
    h1.appendChild(txt);    //把文本节点增加到标题元素节点中
    document.body.appendChild(h1);  //把创建的 h1 对象添加到 DOM 文档树中
    h1.setAttribute("title","一级标题"); //为 h1 对象创建属性节点 title，并设置属性值
    h1.setAttribute("class","red"); //为 h1 对象创建属性节点 title，并设置属性值
}
</script>
```

运行代码，在浏览器中依然看到同样的效果。由于创建的属性不能直接显示出来，初学者可能疑惑如何知道当前文档在预览时是否已添加了属性？大家不妨使用 FireFox 浏览器下的 Firebug 工具查看文档的结构，如图 4.3 所示。Firebug 工具能够查看文档中 JavaScript 动态生成的 HTML、CSS 结构和样式，是开发人员不可缺少的工具。

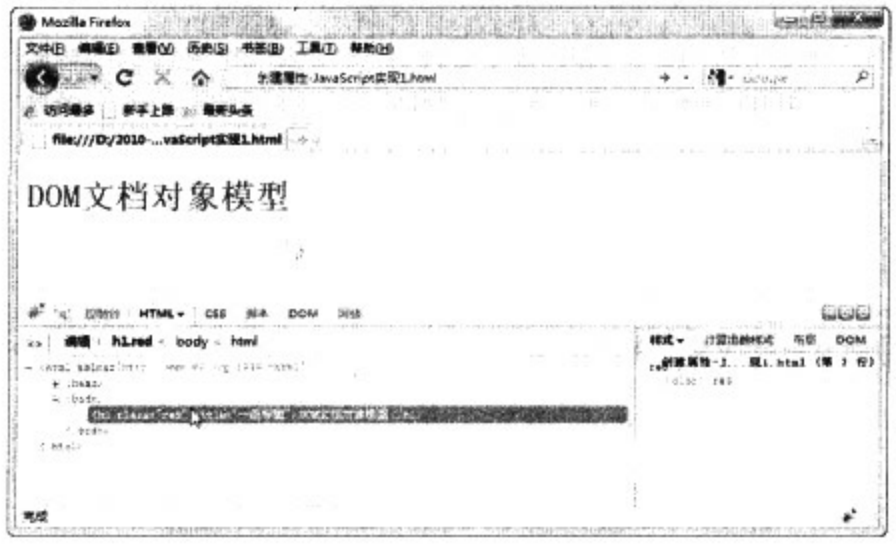


图 4.3　使用 Firebug 工具查看动态生成的属性节点

3. 实现方法比较

从代码输入的角度分析，JavaScript 实现很麻烦，用户需要单独为元素设置属性，而 jQuery 能够直接把元素、文本和属性混合在一起，根据 HTML 语法格式编写成字符串，再传递给$()函数即可。

从执行效率角度分析，JavaScript 实现与 jQuery 实现的效率差距也非常大。在 JavaScript 执行速度最快的 Safari 浏览器中循环执行 1000 次，JavaScript 实现耗时为十几毫秒，而 jQuery 实现耗时为五百多毫秒。不同环境、版本和每次执行时间可能略有误差，但是差距基本保持在 40 倍左右。在执行速度最慢的 IE 浏览器中进行同比测试，JavaScript 实现耗时为 300~400 毫秒，而 jQuery 实现耗时为 3500 多毫秒，可见两者差距也在 10 倍左右。

由此可见，jQuery 以一种简易的方法代替繁琐的操作简化了 Web 开发的难度和门槛，当然由于 jQuery 仅是在 JavaScript 基础上外部代码封装，所以执行速度并没有得到优化，相反却拖延了代码的执行效率。在不必要的前提条件下，建议读者尽量多使用 JavaScript 直接操作 DOM。

4.3　插 入 元 素

在实际开发中，动态创建元素很少被使用，而直接把元素插入文档会显得更加方便快捷。jQuery 和 JavaScript 都提供了很多插入元素的方法，本节将进行详细讲解并进行比较。

4.3.1 jQuery 实现

jQuery 提供了众多插入元素的方法，这些方法从不同角度和用途进行设计，极大地方便了 Web 设计师的操作，使开发者的思维和开发速度提升到一个全新的境界。

1. 在节点内部插入内容

在节点内部插入内容的方法共有 4 种，具体说明如表 4.1 所示。

表 4.1 在节点内部插入内容

方 法	说 明
append()	向每个匹配的元素内部追加内容
appendTo()	把所有匹配的元素追加到另一个指定的元素集合中。实际上，该方法颠倒了 append()的用法。例如，$(A).append(B)与$(B).appendTo(A)是等价的
prepend()	向每个匹配的元素内部前置内容
prependTo()	把所有匹配的元素前置到另一个指定的元素集合中。实际上，该方法颠倒了 prepend()的用法。例如，$(A).prepend (B)与$(B). prependTo (A)是等价的

append()方法能够把参数指定的内容插入到指定的节点中，并返回一个 jQuery 对象。参数可以是 DOM 元素对象、jQuery 对象或者是字符串。下面的示例演示了使用 append()方法分别把 DOM 元素对象、jQuery 对象和字符串添加到 ul 元素下面，演示效果如图 4.4 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){//页面初始化函数
    var $ul = $("<ul></ul>");   //创建包含 ul 元素的 jQuery 对象
    var o = $("body").append($ul);  //把创建的 ul 元素附加到 body 元素下，并返回包含 body 元素
的 jQuery 对象
    var li = document.createElement("li");   //使用 JavaScript 方法创建 li 元素对象
    $ul.append(li);    //用 jQuery 方法 append()把 DOM 的元素对象添加到 ul 元素下
    $ul.append($("<li></li>"));//用 jQuery 方法 append()把 jQuery 对象添加到 ul 元素下
    $ul.append("<li></li>");    //用 jQuery 方法 append()把字符串作为元素对象添加到 ul 元素下
    alert(o[0].nodeName);  //返回字符串"BODY"
})
</script>
```

prepend()方法与 append()方法的参数、返回值和用法基本相同，惟一的区别是 prepend()方法添加的内容被放在指定元素内的第一个子元素位置，而 append()方法是把添加的内容放在最后一个元素位置。请看下面的示例，演示效果如图 4.5 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
```

```
    $("div").append("<p>最后一段文本</p>");     //在 div 元素下的第一个子节点位置插入段落文本
    $("div").prepend("<p>第一段文本</p>");      //在 div 元素下的最后一个子节点位置插入段落文本
})
</script>

<div>
    <p>段落文本</p>
</div>
```

图 4.4　append()方法演示

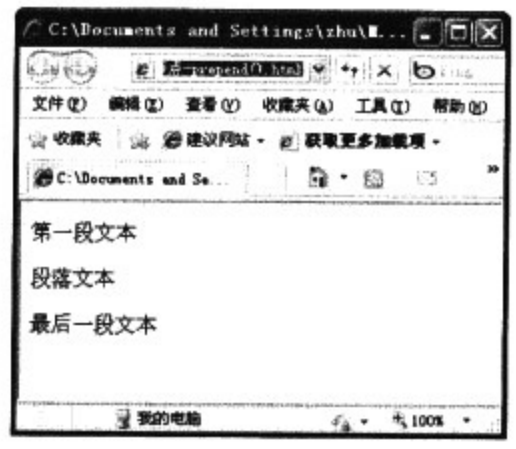


图 4.5　prepend()方法演示

appendTo()和 prependTo()方法的设计思维比较怪异，打破了 DOM 设计中一贯的从右到左的操作方式，不过这种思维形式更符合人的一般思维，所以很受初学者追捧。我们说把 a 插入到 b 中，按照 DOM 逻辑，则应该是 b←a，而现在 jQuery 还可以通过 a→b 形式实现，是不是更符合人的惯性思维？例如，把上面示例的代码可以进行如下修改，但运行结果都是相同的。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("<p>最后一段文本</p>").appendTo("div");  //把段落文本插到 div 元素的第一个子节点位置
    $("<p>第一段文本</p>").prependTo("div");   //把段落文本插到 div 元素的最后一个子节点位置
})
</script>

<div>
    <p>段落文本</p>
</div>
```

2. 在节点外部插入内容

在节点外部插入内容的方法也有 4 种，具体说明如表 4.2 所示。

表 4.2　在节点外部插入内容

方　法	说　明
after()	在每个匹配的元素之后插入内容
before()	在每个匹配的元素之前插入内容

续表

方 法	说 明
insertAfter()	把所有匹配的元素插入到另一个指定的元素集合的后面
insertBefore()	把所有匹配的元素插入到另一个指定的元素集合的前面

举一反三，如果明白了在节点内部插入内容的 4 个方法的用法，那么上表 4 个方法的用法也就很简单了。它们的参数、返回值和用法与节点内插入内容的方法基本相似且一一对应。例如，在下面的示例中，虽然在浏览器中的演示效果与图 4.5 所示的没有差异，但是它们的结构却完全不同，本示例的 HTML 结构如图 4.6 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("div").after("<p>最后一段文本</p>");      //在 div 元素后面插入段落文本
    $("div").before("<p>第一段文本</p>");       //在 div 元素前面插入段落文本
})
</script>

<div>
    <p>段落文本</p>
</div>
```

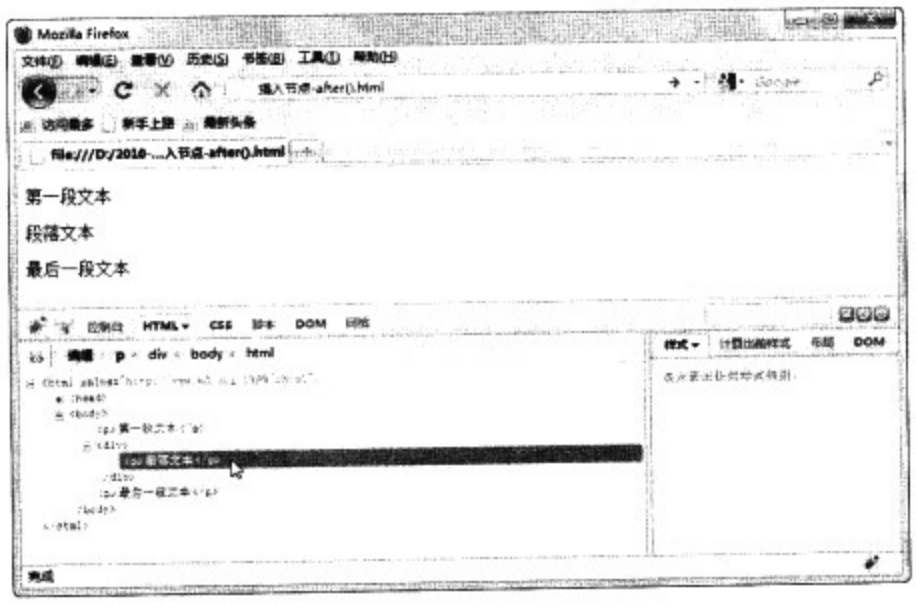


图 4.6 在元素外部插入内容的演示效果

同理，如果使用 insertAfter()和 insertBefore()方法实现上面的示例效果，则可以反向进行操作，代码如下。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("<p>最后一段文本</p>").insertAfter("div");    //把段落文本插入到 div 元素后面
    $("<p>第一段文本</p>").insertBefore("div");     //把段落文本插入到 div 元素前面
})
</script>
```

3. 插入内容的破坏性

在 jQuery 1.3.2 中，appendTo()、prependTo()、insertBefore()和 insertAfter()方法具有破坏性操作特性。也就是说，如果选择已存在内容，并把它们插入到指定对象中时，则原位置的内容将被删除。例如，下面的示例中将把原 div 元素中包含的段落文本选中并移到 div 元素的后面，演示效果如图 4.7 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").insertAfter("div"); //把选中的段落文本插入到 div 元素后面
})
</script>

<div>
    <p>段落文本</p>
</div>
```

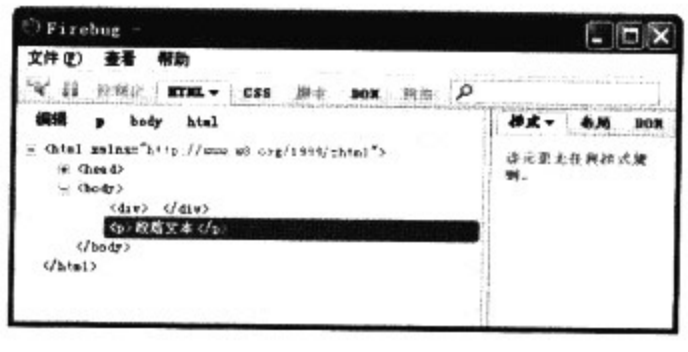


图 4.7 jQuery 插入方法的破坏性演示效果

4.3.2 JavaScript 实现

JavaScript 提供了两个方法实现在节点内部插入元素，其中 appendChild()方法与 jQuery 的 append()方法相对应，insertBefore()方法与 jQuery 的 prepend()方法相对应。

appendChild()方法的用法在前面已经介绍过，它是把参数元素插入到指定节点内的尾部。例如，下面示例演示了如何把一个 h1 元素添加到 div 元素内部的尾部，演示效果如图 4.8 所示。

```
<script type="text/javascript" >
window.onload = function(){
    var div = document.getElementsByTagName("div")[0];    //选择文档中第一个 div 元素
    var h1 = document.createElement("h1");   //创建 h1 元素对象
    div.appendChild(h1);    //把 h1 添加到 div 元素内部的尾部
}
</script>

<div>
    <p>段落文本</p>
```

```
</div>
```

insertBefore()方法的用法比较特殊，它可以把一个指定的节点插入到给定元素中。这与appendChild()方法略有不同，insertBefore()方法所插入的节点位于指定元素的指定子节点的前面，而不是放置在最后面。因此该方法可以包含两个参数，语法格式如下。

```
var o = e.insertBefore(new_e, target_node);
```

返回值 o 表示新添加的子节点，参数 new_e 表示新添加的子节点。而参数 target_node 表示元素 e 包含的一个子节点。如果指定了 target_node 参数，则表示在 target_node 子节点前面增加一个指定子节点对象；如果 target_node 参数值为 null，则插入的子节点被放置在最后面，此时它等同于 appendChild()方法。

使用 insertBefore ()方法可以把指定元素精确插入到文档结构中的任意位置。例如，下面示例的用法与使用 jQuery 的 prepend()方法实现是相同的，演示效果如图 4.9 所示。

```
<script type="text/javascript" >
window.onload = function(){
    var div = document.getElementsByTagName("div")[0];     //选择文档中第一个 div 元素
    var h1 = document.createElement("h1");   //创建 h1 元素对象
    var o = div.insertBefore(h1,div.firstChild);       //把 h1 添加到 div 元素内部第一个子节点的前面
}
</script>
<div>
    <p>段落文本</p>
</div>
```

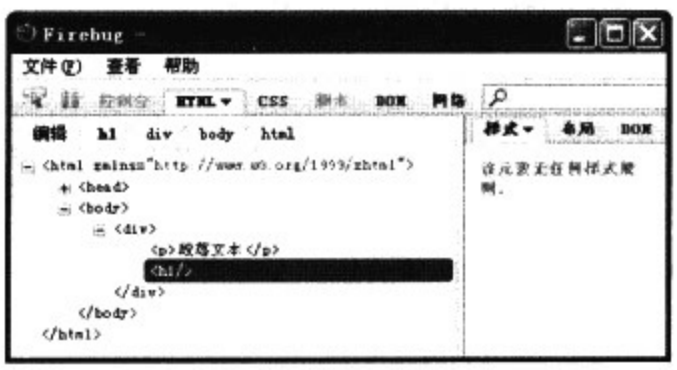


图 4.8 appendChild()方法应用

图 4.9 insertBefore()方法应用

4.3.3 自定义 JavaScript 扩展 DOM 功能函数

JavaScript 没有像 jQuery 那样提供更多的方法以方便 Web 开发人员灵活编写程序。在我们准备为 JavaScript 扩展这些方法之前，应先制造一个“车轮”，以便在后面进行“造车”。

首先，大家应该明白一个道理：DOM 与 JavaScript 拥有不同的作用域。简单说，在

JavaScript 脚本中定义的函数，DOM 节点对象是无法访问的。也正因为如此，下面的函数对于元素对象来说是无法访问的。

```
<script type="text/javascript" >
var appendTo = function(e){     //直接定义插入节点的扩展方法
    e.appendChild(this);
    return this;
}
//在试验中调用自定义方法
window.onload = function(){
    var div = document.getElementsByTagName("div")[0];
    var h1 = document.createElement("h1");
    h1.appendTo(div);  //模仿 jQuery 的 appendTo()用法调用自定义方法
}
</script>

<div>
    <p>段落文本</p>
</div>
```

其次，不同浏览器对于 DOM 标准的支持存在差异。对于符合 DOM 标准的浏览器来说，都支持 HTMLElement 类型，DOM 文档结构中所有元素都继承这个类型，如果为 HTMLElement 类型的原型对象添加方法，则 HTML DOM 文档中所有元素都会继承该方法。因此，针对符合 DOM 标准的浏览器来说，我们可以通过下面方法来扩展 JavaScript 的插入节点方法。

```
<script type="text/javascript" >
HTMLElement.prototype.appendTo = function(e){     //为 HTMLElement 类型扩展原型方法
    e.appendChild(this);
    return this;
}
//在试验中调用自定义原型方法
window.onload = function(){
    var div = document.getElementsByTagName("div")[0];
    var h1 = document.createElement("h1");
    h1.appendTo(div);  //模仿 jQuery 的 appendTo()用法调用原型方法
}
</script>

<div>
    <p>段落文本</p>
</div>
```

注意，在函数体内，应该通过关键字 this 来指向当前调用方法的元素对象，而不用从参数变量中获取当前元素。

遗憾的是 IE 浏览器不支持 HTMLElement 类型，禁止通过 JavaScript 脚本来访问它，因此上述方法在 IE 浏览器中是无法通过测试的。

因此，从兼容性角度考虑，我们可以分别为 IE 和非 IE 浏览器设计解决方案。对于非 IE 浏览器来说，采用上述方法即可，而对于 IE 浏览器来说，则可以考虑使用借壳上市的策略为 DOM 节点绑定自定义方法。虽然这种方法略显笨拙，但不失为最有效的方法。完整解决方案如下。

```
//制造车轮，该函数可以为 DOM 扩展名称为 name 的方法 fn
var DOMextend = function(name, fn){
    if( ! document.all)     //兼容非 IE 浏览器
        eval("HTMLElement.prototype." + name + " = fn");
    else{ //兼容 IE 浏览器
        var _createElement = document.createElement;
        document.createElement = function(tag){   //为 createElement()方法绑定自定义函数
            var _elem = _createElement(tag);
            eval("_elem." + name + " = fn");
            return _elem;
        }
        var _getElementById = document.getElementById;
        document.getElementById = function(id){   //为 getElementById()方法绑定自定义函数
            var _elem = _getElementById(id);
            eval("_elem." + name + " = fn");
            return _elem;
        }
        var _getElementsByTagName = document.getElementsByTagName;
        document.getElementsByTagName = function(tag){     //为 getElementsByTagName()
方法绑定自定义函数
            var _arr = _getElementsByTagName(tag);
            for(var _elem = 0; _elem < _arr.length; _elem ++ )
            eval("_arr[_elem]." + name + " = fn");
            return _arr;
        }
    }
};
```

对于上述解决方案的代码分解如下。

首先，根据 Document 对象的 all 属性来判断浏览器的类型，因为只有 IE 支持该属性。

然后，对于非 IE 浏览器来说，都会支持 DOM 标准模型，因此可以直接使用 HTMLElement.prototype 为文档节点扩展原型方法，然后文档内所有元素都会继承。对于 IE 浏览器来说，虽然不能够设计原型对象，但是我们可以使用一个技巧来间接实现方法扩展。因为，JavaScript 只能够通过 Document 对象的 getElementById()或 getElementsByTagName()方法，以及借助 Document 对象的 createElement()方法访问已存在或新建的元素节点。如果在访问元素节点时，顺便将要继承的方法绑定到这些被访问的元素对象上，即可实现继承自定义方法的问题。

如何把指定的方法绑定到 Document 对象的 getElementById()、getElementsByTagName()和 createElement()方法上呢？

由于 JavaScript 允许用户重写默认方法，所以我们可以稍稍修改 Document 对象的 getElementById()、getElementsByTagName()和 createElement()方法。为了避免在方法重写过程中可能会破坏原方法的功能，故在重写前先存储原方法的内容，代码如下。

```
var _createElement = document.createElement;
```

然后，重写方法。在重写过程中先执行原方法的代码。

```
var _elem = _createElement(tag);
```

接着，为当前元素绑定用户自定义方法。

```
eval("_elem." + name + " = fn");
```

最后，返回原方法执行的值，这样既不破坏调用方法的功能，同时又为当前元素绑定了一个方法。

4.3.4 使用 JavaScript 自定义 appendTo()和 prependTo()方法

打造好“车轮”之后，“造车”工作就比较轻松了。利用 DOMextend()造车函数可以快速定义 appendTo()和 prependTo()方法，并把它们绑定到每个元素对象上，间接实现方法继承问题。

为 DOM Element 类型对象扩展 appendTo()方法的代码如下。

```
DOMextend("appendTo",function(e){   //为 DOM Element 类型对象绑定 appendTo()方法
    var _this = this;  //存储当前元素对象
    e.appendChild(_this);  //把当前元素添加到参数元素内部的尾部
    return _this; //返回当前元素
})
```

为 DOM Element 类型对象扩展 prependTo()方法的代码如下。

```
DOMextend("prependTo",function(e){ //为 DOM Element 类型对象绑定 prependTo()方法
    var _this = this;  //存储当前元素对象
    e.insertBefore(_this,e.firstChild); //把当前元素添加到参数元素内部的首部
    return _this; //返回当前元素
})
```

下面尝试调用使用 JavaScript 自定义的 appendTo()和 prependTo()方法，演示后的文档结构发生了变化，如图 4.10 所示。

```
<script type="text/javascript" >
//省略了上述自定义的 DOMextend()函数，以及两次调用该函数制造两个方法
window.onload = function(){     //页面初始化
    var div = document.getElementsByTagName("div")[0];     //访问页面中第一个 div 元素
    var h1 = document.createElement("h1");   //创建 h1 元素
```

```
    var h2 = document.createElement("h2");   //创建 h2 元素
    h1.prependTo(div);      //把 h1 添加到 p 元素前面
    h2.appendTo(div);  //把 h2 添加到 p 元素后面
}
</script>

<div>
    <p>段落文本</p>
</div>
```

4.3.5 使用 JavaScript 自定义 after()和 before()方法

虽然 JavaScript 不支持 jQuery 的 after()和 before()方法，但是可以很容易实现，这里主要用到 Element 对象的 parentNode 和 nextSibling 属性。其中 parentNode 属性引用父节点，而 nextSibling 属性引用相邻的下一个节点。

例如，下面的示例就是利用 parentNode 和 nextSibling 属性轻松实现了 jQuery 的 after()和 before()方法。在浏览器预览时，可以看到插入节点的位置，如图 4.11 所示。

```
<script type="text/javascript" >
window.onload = function(){    //页面初始化
    var div = document.getElementsByTagName("div")[0];    //访问页面中第一个 div 元素
    var h1 = document.createElement("h1");   //创建 h1 元素
    var h2 = document.createElement("h2");   //创建 h2 元素
    div.parentNode.insertBefore(h1,div);      //把 h1 添加到 div 元素前面
    div.parentNode.insertBefore(h2,div.nextSibling); //把 h2 添加到 div 元素的后面
}
</script>

<div>
    <p>段落文本</p>
</div>
```

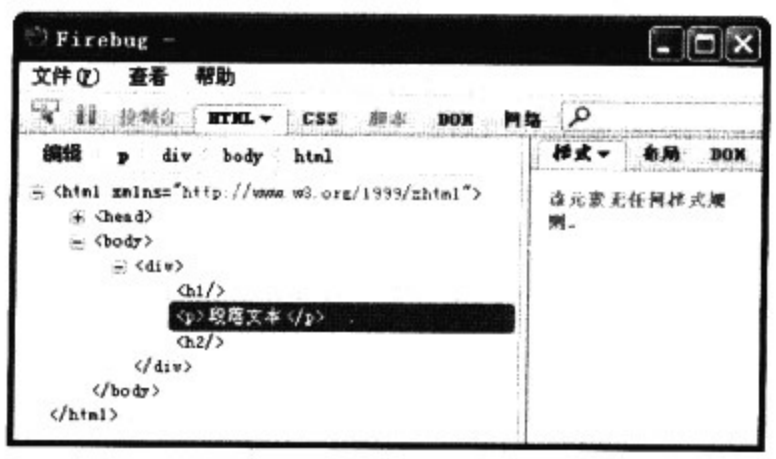


图 4.10 使用 JavaScript 模仿 jQuery 在节点内部插入内容的方法

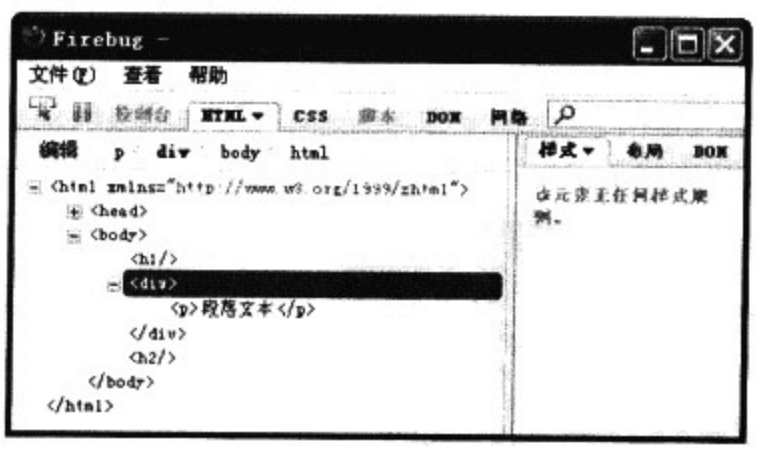


图 4.11 使用 JavaScript 模仿 jQuery 在节点外部插入内容的方法

4.3.6 使用 JavaScript 自定义 insertAfter()和 insertBefore()方法

对于 jQuery 的 insertAfter()和 insertBefore()方法来说，借助第 4.3.3 节定义的扩展功能函数 DOMextend()，也可以轻松实现。考虑到 jQuery 的 insertBefore()方法与 DOM 默认的 insertBefore()方法同名，故在使用 JavaScript 自定义 jQuery 的 insertAfter()和 insertBefore()方法时，全部以小写形式命名。

为 DOM Element 类型对象扩展 insertafter ()方法的代码如下。

```
DOMextend("insertafter",function(e){    //为 DOM Element 类型对象绑定 insertafter()方法
    var _this = this;  //存储当前元素对象
    e.parentNode.insertBefore(_this,e.nextSibling);   //把当前元素插入到参数元素的前面
    return _this; //返回当前元素
})
```

为 DOM Element 类型对象扩展 insertbefore ()方法的代码如下。

```
DOMextend("insertbefore",function(e){   //为 DOM Element 类型对象绑定 insertbefore()方法
    var _this = this;  //存储当前元素对象
    e.parentNode.insertBefore(_this,e); //把当前元素插入到参数元素的后面
    return _this; //返回当前元素
})
```

然后调用使用 JavaScript 自定义的 insertafter()和 insertbefore()方法，代码如下。

```
<script type="text/javascript" >
//省略了上述自定义的 DOMextend()函数，以及两次调用该函数制造两个方法
window.onload = function(){    //页面初始化
    var div = document.getElementsByTagName("div")[0];    //访问页面中第一个 div 元素
    var h1 = document.createElement("h1");   //创建 h1 元素
    var h2 = document.createElement("h2");   //创建 h2 元素
    h1.insertbefore(div);  //把 h1 添加到 div 元素前面
    h2.appendTo(div);  //把 h2 添加到 div 元素后面
}
</script>

<div>
    <p>段落文本</p>
</div>
```

4.4 删除元素

jQuery 支持两种删除元素的方法：remove()和 empty()，而 DOM 仅定义了与 jQuery 的 remove()方法对应的 removeChild()方法。

4.4.1 jQuery 实现

jQuery 的 remove()和 empty()方法说明如表 4.3 所示。

表 4.3 jQuery 删除元素的方法

方 法	说 明
remove()	从 DOM 中删除所有匹配的元素
empty()	删除匹配的元素集合中所有的子节点

由于 remove()方法能够删除匹配的元素，并返回这个被删除的元素，因此在特定条件下该方法的功能可以使用 jQuery 的 appendTo()、prependTo()、insertBefore()或 insertAfter()方法进行模拟。例如，下面的示例将父元素 div 的子元素 p 移出，然后插入到父元素 div 的后面，执行之后的 HTML 结构如图 4.12 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    var $p = $("p").remove();   //删除段落文本
    $p.insertAfter("div"); //把删除的段落文本添加到 div 元素后面
})
</script>
<div>
    <p>段落文本</p>
</div>
```

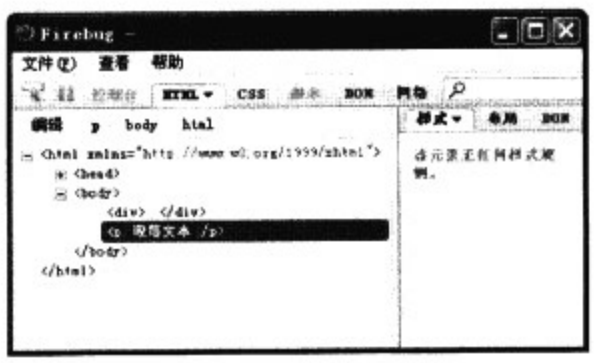


图 4.12 使用 jQuery 移动 HTML 结构

如果使用 insertAfter()方法，则可以把上面的两步操作合并为一步，代码如下。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").insertAfter("div"); //直接把段落文本移动到 div 元素后面
})
</script>
```

当然，remove()方法的主要功能是删除指定节点以及包含的子节点；empty()方法不是删除节点，而是清空元素包含的内容。在用法上，两者都相似，但是执行结果略有区别。

4.4.2 JavaScript 实现

DOM 内置了 removeChild()方法，该方法可以删除指定的节点及其包含的所有子节点，并返回这些删除的内容。因此，jQuery 的 remove()实际上就是简单包装了 DOM 的 removeChild()方法。例如，针对上节中使用 jQuery 演示的示例，我们完全可以使用 JavaScript 实现。

```
<script type="text/javascript" >
window.onload = function(){
    var div = document.getElementsByTagName("div")[0];     //选择 div 元素
    var p = document.getElementsByTagName("p")[0];     //选择 p 元素
    var p1 = div.removeChild(p);     //移出 p 元素
    div.parentNode.insertBefore(p1,div.nextSibling); //把移出的 p 元素附加到 div 元素后面
}
</script>

<div>
    <p>段落文本</p>
</div>
```

由于 DOM 的 insertBefore()与 appendChild()方法都具有破坏性，当使用文档中的现有元素进行操作时，会先删除原位置上的元素。因此对于下面两行代码：

```
var p1 = div.removeChild(p);    //移出 p 元素
div.parentNode.insertBefore(p1,div.nextSibling);      //把移出的 p 元素附加到 div 元素后面
```

可以合并为：

```
div.parentNode.insertBefore(p,div.nextSibling); //直接使用 insertBefore()移动 p 元素
```

4.4.3 使用 JavaScript 自定义 empty()方法

DOM 不支持 jQuery 的 empty()方法，不过使用 JavaScript 却很容易实现。使用 JavaScript 自定义 empty()方法的完整代码，以及调用该方法的用法如下。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
//……
//为 DOM 元素节点类型绑定 empty()方法
DOMextend("empty", function(){
    var _this = this;  //存储当前父元素
    var a = [];    //声明数组变量
```

```
    for(var i = 0, c = _this.childNodes, l = c.length;i < l;i++ ){//循环删除所有子节点
        a.push(_this.removeChild(c[0]));      //把删除的子节点存储到临时数组中
    }
    return a;      //返回被删除的子节点数组
})
window.onload = function(){     //页面初始化
    var div = document.getElementsByTagName("div")[0];     //访问页面中第一个 div 元素
    div.empty();   //调用自定义的 empty()方法清空 div 元素的内容
}
</script>

<div>
    <p>段落文本</p>
</div>
```


4.5 复制元素

jQuery 定义了 clone()方法用来复制节点，与之对应 DOM 定义了 cloneNode()方法实现相同的操作功能。

4.5.1 jQuery 实现

jQuery 的 clone()方法能够复制匹配的 DOM 元素并且选中这些复制的副本。例如，在下面的示例中，当单击段落文本时，页面将复制 div 元素及其包含的所有属性和子节点，然后将它们添加到 div 元素的后面，演示效果如图 4.13 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){//页面初始化函数
    $("div").click(function(){  //为匹配的 div 元素绑定单击事件
        var $div = $("div").clone();//复制匹配的 div 元素
        $div.insertAfter("div");      //把复制的元素添加到当前 div 元素的后面
    })
})
</script>

<div class="red" title="no" ondblclick="alert('ok')">
    <p>段落文本</p>
</div>
```

但是复制的 div 元素不拥有鼠标单击事件，而拥有 div 元素自身携带的鼠标双击事件。如果为 clone()方法传递 true 参数，则可以使复制的 div 元素也拥有单击事件，也就是说当单击复制 div 元素时，它会继续进行复制操作，连续单击会使复制的 div 元素成倍增加，如图 4.14 所示。

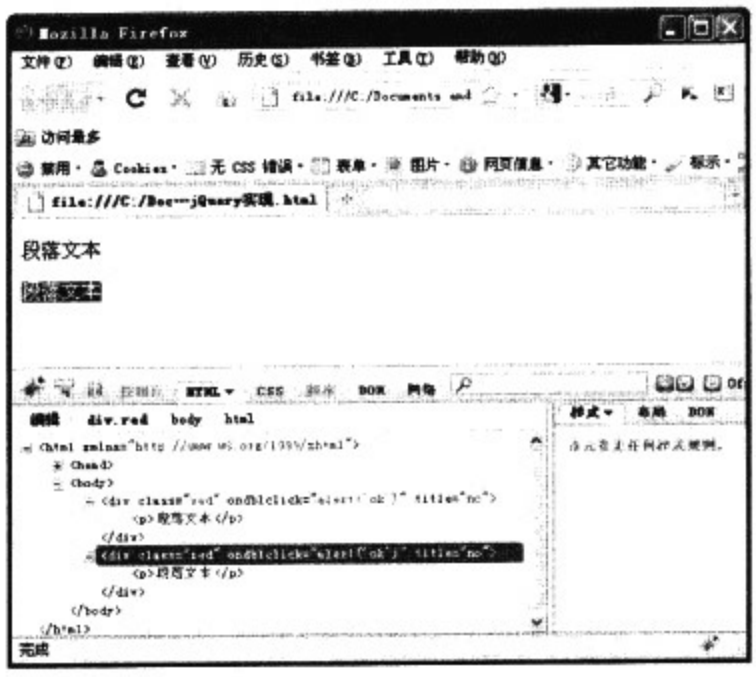


图 4.13　复制元素但不复制所有的事件处理函数

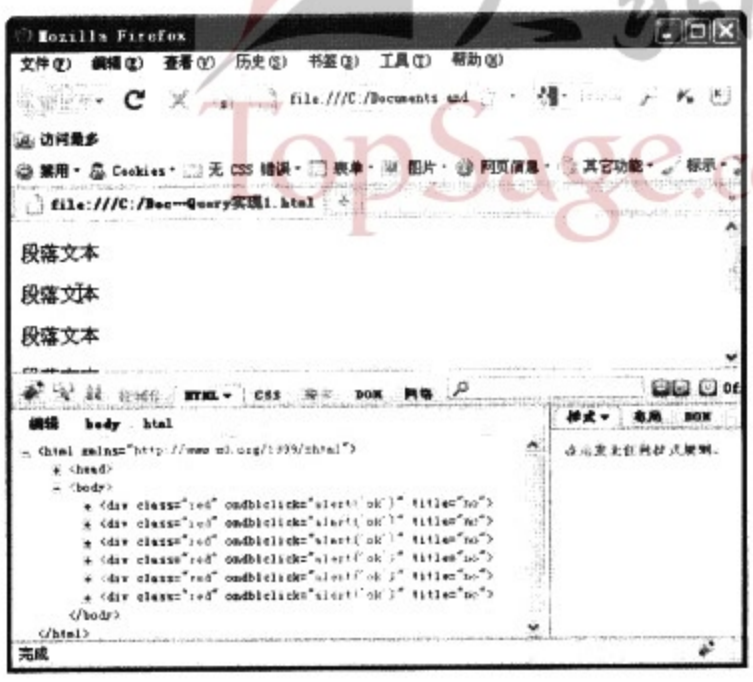


图 4.14　复制元素并复制所有的事件处理函数

4.5.2　JavaScript 实现

DOM 预定义了 cloneNode()方法，可以创建指定节点的副本。该方法包含一个参数，取值包括 true 和 false，用来设置被复制的节点是否包括原节点的所有属性和子节点。例如，下面示例将使用 cloneNode()方法来复制 div 元素及其所有属性和子节点，演示效果如图 4.13 所示。

```
<script type="text/javascript" >
window.onload = function(){     //页面初始化函数
    var div = document.getElementsByTagName("div")[0];     //选择 div 元素
    div.onclick = function(){   //绑定单击事件函数
        var div1 = div.cloneNode(true);  //复制 div 元素
        div.parentNode.insertBefore(div1,div.nextSibling)
                                                    //将复制的 div 元素添加到 div 元素后面
    }
}
</script>

<div class="red" title="no" ondblclick="alert('ok')">
    <p>段落文本</p>
</div>
```

4.6　替 换 元 素

jQuery 定义了 replaceWith()方法和 replaceAll()方法来替换节点，与之对应 DOM 定义了 replaceChild()方法实现相同的操作功能，不过它们的用法迥然不同。

4.6.1 jQuery 实现

replaceWith()方法能够将所有匹配的元素替换成指定的 HTML 或 DOM 元素，replaceAll()方法与之功能相同，但是操作相反。例如，在下面的示例中，将使用 replaceWith()方法把所有 p 元素替换为 div 元素，演示效果如图 4.15 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){//页面初始化函数
    $("p").replaceWith("<div>盒子</div>");    //替换所有 p 元素为 div 元素
})
</script>

<p>段落 1</p>
<p>段落 2</p>
<p>段落 3</p>
```

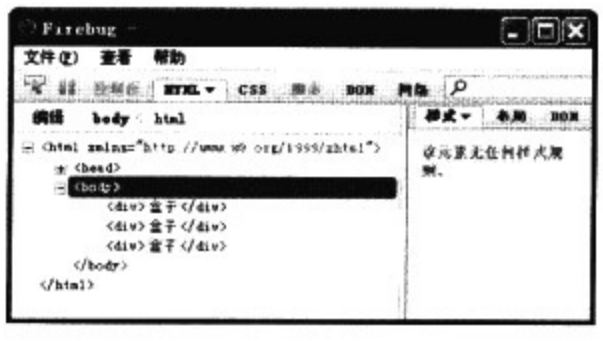


图 4.15 替换元素

如果使用 replaceAll()方法替换上面示例中的代码，则可以这样进行修改。

```
$(function(){
    $("<div>盒子</div>").replaceAll("p");
})
```

由此可见，replaceWith()方法与 replaceAll()方法的操作思路是相反的。

4.6.2 JavaScript 实现

DOM 预定义了 replaceChild()方法实现节点替换，该方法包含两个参数，第一个参数指定替换的节点，第二个参数指定被替换的节点。例如，针对上一节的示例，使用 JavaScript 实现的代码如下。

```
<script type="text/javascript" >
window.onload = function(){    //页面初始化函数
    var p = document.getElementsByTagName("p");  //选择所有 p 元素
    var div = document.createElement("div"); //创建 div 元素
    div.innerHTML = "盒子"; //为 div 元素添加文本内容
```

```
    for(var i=0,l = p.length;i< l;i++){ //遍历 p 元素集合
        var div1 = div.cloneNode(true);  //复制 div 元素
        p[0].parentNode.replaceChild(div1,p[0]); //使用复制的 div 元素替换掉 p 集合中的第一个 p 元素
    }
}
</script>

<p>段落 1</p>
<p>段落 2</p>
<p>段落 3</p>
```

4.6.3 使用 JavaScript 自定义 replaceWith()和 replaceAll()方法

DOM 提供的 replaceChild()方法在用法上与 jQuery 定义的 replaceWith()和 replaceAll()方法存在很大不同。replaceChild()方法需要知道元素的父节点，虽然使用 parentNode 属性可以定位父节点，但是很多初学者并不习惯 replaceChild()方法的用法。

使用 JavaScript 模仿 jQuery 自定义 replaceWith()和 replaceAll()方法也很简单，完整代码及调用方法如下。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
//……
//为 DOM 元素节点类型绑定 replaceWith()和 replaceAll()方法
DOMextend("replaceWith",function(e){
    var _this = this;
    _this.parentNode.replaceChild(e,_this);
    return _this;
})
DOMextend("replaceAll",function(e){
    var _this = this;
    e.parentNode.replaceChild(_this,e);
    return _this;
})
window.onload = function(){     //页面初始化函数
    var p = document.getElementsByTagName("p");  //选择所有 p 元素
    var div = document.createElement("div"); //创建 div 元素
    div.innerHTML = "盒子"; //为 div 元素添加文本内容
    for(var i=0,l = p.length;i< l;i++){ //遍历 p 元素集合
        var div1 = div.cloneNode(true);  //复制 div 元素
        p[0].replaceWith(div1);;     //调用 replaceWith()方法,用复制的 div 替换掉 p 集合中的第
一个 p 元素
    }
}
</script>

<p>段落 1</p>
<p>段落 2</p>
<p>段落 3</p>
```

4.7 包裹元素

开发中经常需要将某个结构元素进行包裹。当需要在结构中插入额外的标记时，这种包裹操作不会破坏文档原有的结构和语义。jQuery 定义了 wrap()系列方法用来包裹元素，而 DOM 并不支持这种操作。

4.7.1 jQuery 实现

jQuery 定义了 3 种包裹元素的方法：wrap()、wrapAll()和 wrapInner()方法。这些方法主要的区别就在于包裹的形式不同。下面我们用一个示例进行演示，代码如下。

先使用 wrap()方法为 p 元素包裹 span 元素，则包裹后的 HTML 文档结构如图 4.16 所示。如果把示例源代码中的 wrap()方法替换为 wrapAll()方法，则包裹后的 HTML 文档结构如图 4.17 所示。如果把示例源代码中的 wrapAll()方法替换为 wrapInner()方法，则包裹后的 HTML 文档结构如图 4.18 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").wrap("<span class='wrap'></span>");
})
</script>

<p>段落 1</p>
<p>段落 2</p>
<p>段落 3</p>
```

图 4.16 wrap()方法包裹后的结构

图 4.17 wrapAll()方法包裹后的结构

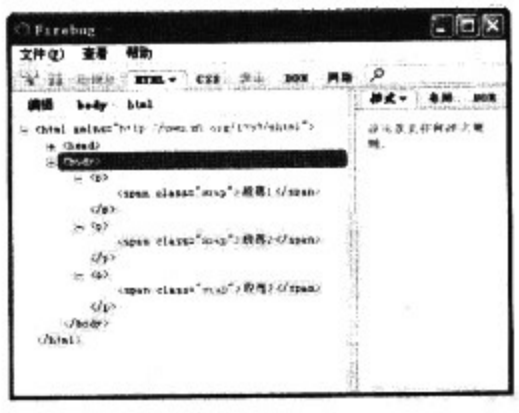


图 4.18 wrapInner()方法包裹后的结构

wrap()、wrapAll()和 wrapInner()方法可以接受 HTML 字符串或者 DOM 元素对象，它们可以是单个元素，也可以是嵌套的多个元素。如果不是嵌套的多个元素，而是多个并列的元素，则 jQuery 会把它们分开并分别进行包裹操作。

4.7.2 使用 JavaScript 自定义 wrap()、wrapAll()和 wrapInner()方法

DOM 没有提供包裹元素的操作方法，但是我们也可以自定义 wrap()、wrapAll()和 wrapInner()方法。

定义 wrap()方法及其用法如下，演示效果如图 4.19 所示。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("wrap",function(e){
    var _this = this;
    _this.parentNode.insertBefore(e,_this); //在当前被包裹元素前面插入包裹元素
    e.appendChild(_this);  //然后把被包裹元素移动到包裹元素中
    return _this;
})
window.onload = function(){    //页面初始化函数
    var p = document.getElementsByTagName("p");  //选择所有 p 元素
    for(var i=0, l=p.length; i< l; i++){     //遍历 p 元素集合
        var span = document.createElement("span");     //创建 span 元素
        p[i].wrap(span);     //调用 wrap()方法包裹每个 p 元素
    }
}
</script>

<p>段落 1</p>
<p>段落 2</p>
<p>段落 3</p>
```

如果我们按下面方式调用自定义的 wrap()方法，则会模仿出 wrapAll()方法的执行效果，如图 4.20 所示。也就是说，把包裹元素放在 for 循环体的外面进行创建即可。

```
window.onload = function(){    //页面初始化函数
    var p = document.getElementsByTagName("p");  //选择所有 p 元素
    var span = document.createElement("span");    //创建 span 元素
    for(var i=0, l=p.length; i< l; i++){     //遍历 p 元素集合
        p[i].wrap(span);     //调用 wrap()方法包裹每个 p 元素
    }
}
```

定义 wrapInner()方法的代码如下，调用该方法为 p 元素嵌套包裹元素，则执行后的 HTML 结构效果如图 4.21 所示。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("wrap",function(e){
    var _this = this;
    _this.parentNode.insertBefore(e,_this); //在当前元素前面插入包裹元素
    while(_this.firstChild){    //遍历当前元素，把所有子节点转移到包裹元素中
```

```
        e.appendChild(_this.firstChild);
    }
    _this.appendChild(e);  //再把包裹元素转移到当前元素内部
    return _this;
})
window.onload = function(){    //页面初始化函数
    var p = document.getElementsByTagName("p");  //选择所有 p 元素
    for(var i=0, l=p.length; i< l; i++){     //遍历 p 元素集合
        var span = document.createElement("span");     //创建 span 元素
        p[i].wrapInner(span);    //调用 wrapInner()方法包裹每个 p 元素
    }
}
</script>

<p>段落<b>1</b></p>
<p>段落 2</p>
<p>段落 3</p>
```

图 4.19　自定义 wrap()方法用法

图 4.20　自定义 wrap()方法特殊用法

图 4.21　自定义 wrapInner()方法用法

4.8　操 作 属 性

jQuery 和 DOM 都提供了元素属性的基本操作方法，掌握这些方法可以满足 Web 开发中普通的应用需求。

4.8.1　设 置 属 性

jQuery 使用 attr()方法设置元素的属性，而 DOM 定义了 setAttribute()方法来设置元素的属性。为了兼容 XML 文档，DOM 提供了更多属性设置的方法，由于它们在 HTML 文档中不经常使用，故这里也不再具体讲解。

1. jQuery 实现

使用 jQuery 的 attr()方法设置元素属性时，需要为该方法设置两个参数值，第一个参数指定属性名，第二个参数指定属性值。也可以为第二个参数传递一个函数，由函数返回的结

果充当属性值。例如，下面的示例就为 p 元素设置了 title 属性，属性值为“段落文本”。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").attr("title","段落文本");
})
</script>

<p>段落文本</p>
```

2. JavaScript 实现

DOM 定义的 setAttribute()方法与 jQuery 定义的 attr()方法在用法上基本相同。例如，针对上面的示例，直接使用 JavaScript 进行设置，则代码如下。

```
<script type="text/javascript" >
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    p.setAttribute("title","段落文本");
}
</script>

<p>段落文本</p>
```

3. 执行效率比较

通过 1000 次循环试验比较，发现 jQuery 的 attr()方法要比 DOM 的 setAttribute()方法慢十倍左右。因此，从执行效率的角度分析，如果文档中需要动态设置的属性比较多，则建议考虑选用 DOM 的 setAttribute()方法。

不过从用法角度考虑，jQuery 的 attr()方法稍显灵活，可以批量设置属性。例如：

```
$("img").attr({ src: "test.jpg", alt: "Test Image" });
```

而 DOM 的 setAttribute()方法一次只能够设置一个属性。

4.8.2 获取属性

jQuery 仍然使用 attr()方法获取元素的属性值，而 DOM 重新定义了 getAttribute()方法来获取元素的属性值。

1. jQuery 实现

当为 jQuery 的 attr()方法设置一个参数时，则表示读取参数指定名称的属性值。如果没有相应属性，则返回 undefined。例如，下面的示例就是读取 p 元素的 title 属性，属性值为

“段落文本”。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    alert($("p").attr("title"));     //弹出"段落文本"提示信息
})
</script>

<p title="段落文本">段落文本</p>
```

2. JavaScript 实现

DOM 定义的 getAttribute()方法也仅有一个参数，使用该参数可以指定要获取的属性。例如，针对上面的示例，直接使用 JavaScript 进行获取，则代码如下。

```
<script type="text/javascript" >
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    alert(p.getAttribute("title")); //弹出"段落文本"提示信息
}
</script>

<p title="段落文本">段落文本</p>
```

4.8.3 删除属性

jQuery 定义了 removeAttr()方法删除指定的元素属性，DOM 也定义了 removeAttribute()方法删除指定的元素属性。

1. jQuery 实现

jQuery 的 removeAttr()方法仅需要指定属性名即可。例如：

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").removeAttr("title"); //删除 p 元素的 title 属性，并返回 jQuery 对象，包含 p 元素
})
</script>

<p title="段落文本">段落文本</p>
```

2. JavaScript 实现

如果使用 JavaScript 直接删除元素的属性，则代码如下。

```
<script type="text/javascript" >
window.onload = function(){
```

```
    var p = document.getElementsByTagName("p")[0];
    p.removeAttribute("title"); //删除 p 元素的 title 属性
}
</script>

<p title="段落文本">段落文本</p>
```

4.9 操作类样式

在设计动态样式时，经常需要与元素的 class 属性打交道，该属性可以为元素定义类样式。既然作为元素的属性，当然可以使用 jQuery 的 attr()方法或者 DOM 的 setAttribute()和 getAttribute()方法设置和读取元素的类样式。不过，jQuery 为了方便设计师操作，又单独定义了几个与类样式相关的操作方法，详细讲解如下。

4.9.1 追加样式

jQuery 定义了 addClass()方法专门负责为元素追加样式。为了方便演示，我们先定义以下几个类样式。

```
<style type="text/css">
.red {/* 红色字体类样式 */
    color:red;
}
.bold {/* 加粗字体类样式 */
    font-weight:bold;
}
.italic {/* 斜体字体类样式 */
    font-style:italic;
}
</style>
```

1. jQuery 实现

addClass()方法用法简单，仅需要指定类名即可，但是在调用前应先定义类样式，否则在预览时会看不到效果。例如，在下面的示例中，设计 3 个按钮，当单击这些按钮时，将分别为 p 元素追加类样式，演示效果如图 4.22 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){//页面初始化
    $("input").eq(0).click(function(){   //为第一个按钮注册单击事件
        $("p").addClass("red"); //为 p 元素追加 red 类样式
    })
    $("input").eq(1).click(function(){   //为第二个按钮注册单击事件
        $("p").addClass("bold");      //为 p 元素追加 bold 类样式
    })
```

```
    $("input").eq(2).click(function(){   //为第三个按钮注册单击事件
        $("p").addClass("italic");   //为p元素追加italic类样式
    })
})
</script>
</head>
<body>

<p>段落文本</p>
<input type="button" value="红色"  />
<input type="button" value="加粗"  />
<input type="button" value="斜体"  />
```

2. JavaScript 实现

DOM 虽然没有定义 addClass()方法，但是结合 setAttribute()和 getAttribute()方法，我们也可以模拟出 jQuery 的 addClass()方法。例如，针对上面的示例，使用 JavaScript 直接设计，则完整的 JavaScript 代码如下。

```
<script type="text/javascript" >
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    var input = document.getElementsByTagName("input");
    input[0].onclick = function(){  //为第一个按钮注册单击事件
        var a = p.getAttribute("class"); //获取p元素的class属性值
        if(a){      //如果存在类样式，则进行追加操作
            p.setAttribute("class", a + " red");
        }else{ //如果不存在类样式，则直接设置类样式
            p.setAttribute("class", "red");
        }
    }
    input[1].onclick = function(){  //为第二个按钮注册单击事件
        var a = p.getAttribute("class"); //获取p元素的class属性值
        if(a){      //如果存在类样式，则进行追加操作
            p.setAttribute("class", a + " bold");
        }else{ //如果不存在类样式，则直接设置类样式
            p.setAttribute("class", "bold");
        }
    }
    input[2].onclick = function(){  //为第三个按钮注册单击事件
        var a = p.getAttribute("class"); //获取p元素的class属性值
        if(a){      //如果存在类样式，则进行追加操作
            p.setAttribute("class", a + " italic");
        }else{ //如果不存在类样式，则直接设置类样式
            p.setAttribute("class", "italic");
        }
    }
}
</script>
```

4.9.2 移出样式

如果要清空或者重设元素的 class 属性值，只需用 jQuery 的 attr()方法或者 DOM 的 setAttribute()方法即可。但是如果要移出复合类样式中的某个样式，就需要另想办法了。为

了解决这个问题，jQuery 单独定义了 removeClass()方法。

1. jQuery 实现

removeClass()方法能够移出指定类名的样式，如果要删除多个类样式，则可以把多个类样式名组合成一个字符串，并用空格键分隔，然后作为参数传递给 removeClass()方法即可。如果要删除所有类样式，则可以不传递参数。例如，我们继续沿用上节示例的 HTML 结构，然后添加几个移出样式的按钮，以便执行不同的操作，演示效果如图 4.23 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){//页面初始化
    //……省略前 3 个按钮注册事件的函数
   $("input").eq(3).click(function(){    //为第四个按钮注册单击事件
        $("p").removeClass("red");   //移出红色类样式
    })
    $("input").eq(4).click(function(){   //为第五个按钮注册单击事件
        $("p").removeClass("bold red");  //移出红色和加粗类样式
    })
    $("input").eq(5).click(function(){   //为第六个按钮注册单击事件
        $("p").removeClass();    //移出所有类样式
    })
})
</script>

<p>段落文本</p>
<input type="button" value="红色"  />
<input type="button" value="加粗"  />
<input type="button" value="斜体"  /><br />
<input type="button" value="移出红色类样式"  /><br />
<input type="button" value="移出红色和加粗类样式"  /><br />
<input type="button" value="移出全部类样式"  />
```

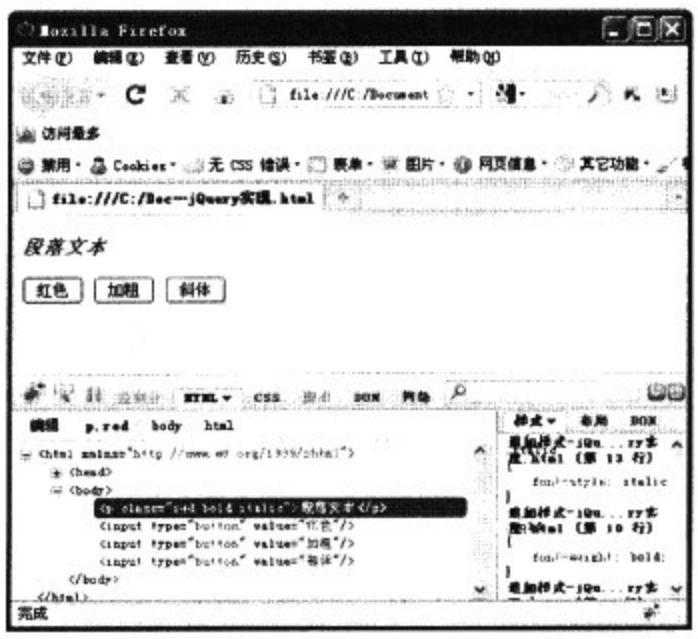


图 4.22　应用 jQuery 的 addClass()方法

图 4.23　应用 jQuery 的 removeClass()方法

2. JavaScript 实现

使用 JavaScript 自定义的 removeClass()方法也很简单，主要利用正则表达式匹配指定的

类名，然后把它替换为空字符串即可。使用 JavaScript 自定义 removeClass()方法的详细代码，以及调用 removeClass()方法的示例如下。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("removeClass",function(s){    //自定义 removeClass()方法，并绑定到 HTML 元素身上
    var _this = this;
    if(!s)    //如果没有传递参数，则清除所有类样式
        this.setAttribute("class","");
    else {
        var a = s.split(" ");    //如果传递参数，则把参数字符串按空格键劈开为数组
        var attr = _this.getAttribute("class");    //获取元素的类样式
        for(var i=0; i<a.length; i++){    //遍历参数字符串中的类名
            attr = attr.replace(a[i],"");      //与当前元素的类样式字符串进行匹配，并把匹配字符
串替换为空格
        }
        _this.setAttribute("class",attr);      //使用匹配后的字符串重新设置当前元素的 class 属
性值
    }
    return _this;
})
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    var input = document.getElementsByTagName("input");
    //……省略前 3 个按钮注册事件的函数
    input[3].onclick = function(){  //为第四个按钮注册单击事件
        p.removeClass("red");    //清除红色类样式
    }
    input[4].onclick = function(){  //为第五个按钮注册单击事件
        p.removeClass("red bold");    //清除红色和加粗类样式
    }
    input[5].onclick = function(){  //为第六个按钮注册单击事件
        p.removeClass();    //清除所有类样式
    }
}
</script>

<!-- 省略了 HTML 结构，请参阅上节示例显示 -->
```

4.9.3 切换样式

样式切换在鼠标动态操作中非常实用，在 Web 开发中诸如折叠、开关、伸缩和 Tab 切换等动态效果都需要用到交互切换。jQuery 为此定义了 toggleClass()方法，该方法可以开/关指定的类样式，从而实现切换样式的设计目标。

1. jQuery 实现

jQuery 的 toggleClass()方法包含两个参数，第一个参数指定作为开关的类样式名称。第

二个参数用于决定元素是否打开类样式，该参数为可选。如果没有设置第二个参数，则 toggleClass()方法根据指定元素是否存在来决定参数设置的样式，如果存在，则清除该类样式，如果不存在，则追加该类样式，实现动态切换效果。例如，定义一个 hidden 类样式，该样式中包含隐藏元素显示的声明，然后调用 toggleClass()方法，并把 hidden 类样式传递给它，就可以实现显示/隐藏的动态切换效果，如图 4.24 所示，实现代码如下。

```
<style type="text/css">
.hidden {
    display:none;
}
</style>
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("input").eq(0).click(function(){
        $("p").toggleClass("hidden");     //切换显示或隐藏 p 元素
    })
})
</script>

<input type="button" value="切换样式"  />
<p><img src="images/bg1.jpg" height="200" /></p>
```

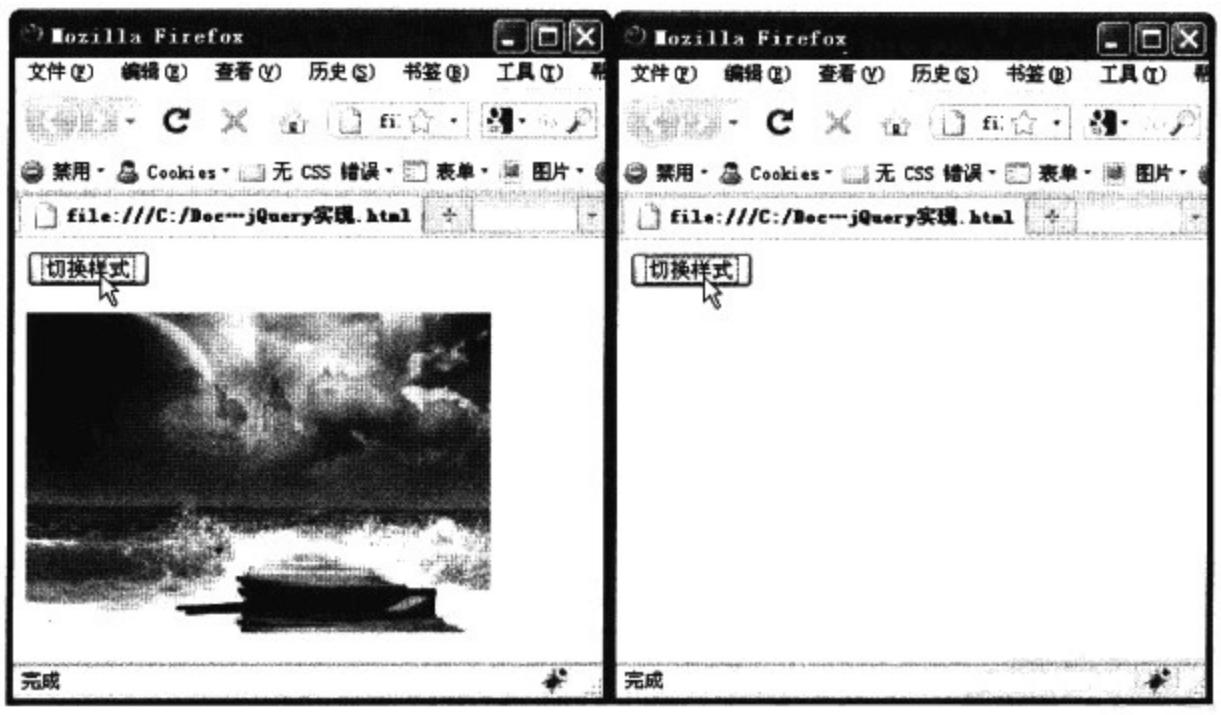


图 4.24　动态切换样式效果

再如，下面的示例设置第二个参数为一个表达式，设计每单击三下才追加上一次 hidden 类样式。

```
var n = 0;
$("input").eq(0).click(function(){
    $("p").toggleClass("hidden", n++ % 3 == 0);
})
```

2. JavaScript 实现

使用 JavaScript 自定义 toggleClass()方法稍显麻烦，这里需要考虑三个条件：第一是判断是否存在第二个参数，以及第二个参数的值；第二是判断元素是否已经设置了 class 属性；第三是判断元素设置了 class 属性后，class 属性值是否包含指定类样式。根据上面三个条件，来决定是设置类样式，还是追加类样式，还是清除类样式。使用 JavaScript 自定义 toggleClass() 方法的完整代码及其调用方法如下。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("toggleClass",function(){     //自定义 toggleClass()方法，并绑定到 HTML 元素身上
    var _this = this;
    var attr = _this.getAttribute("class");  //获取当前元素的类样式
    if(!attr){     //如果不存在类样式，则设置指示变量的值为-2
        var t = -2;
    else  //如果存在类样式，匹配指定的参数
        var t = attr.search(arguments[0]);
    if(arguments[1] == true ){  //如果第二个参数值为 true
        if(t==-2) //如果不存在类样式，则设置指定的类样式
            _this.setAttribute("class",arguments[0]);
        if(t==-1) //如果 class 属性值中不存在指定的类样式，则追加类样式
            _this.setAttribute("class",attr + " " + arguments[0]);
    }else if(arguments[1] == false){     //如果第二个参数值为 false，则清除类指定的类样式
        attr = attr.replace(arguments[0],"");
        _this.setAttribute("class",attr);
    }else{     //如果没有设置第二个参数
        if(t>-1){  //如果 class 属性存在指定的类样式，则清除该样式
            attr = attr.replace(arguments[0],"");
            _this.setAttribute("class","red");
        }else if(t==-2)      //如果没有设置 class 属性，则设置指定的类样式
            _this.setAttribute("class",arguments[0]);
        else if(t==-1) //如果设置了 class 属性，但没包含指定类样式，则追加该样式
            _this.setAttribute("class",attr + " " + arguments[0]);
    }
    return _this;
})
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    var input = document.getElementsByTagName("input");
    input.onclick = function(){
        p.toggleClass("hidden");     //调用自定义的 toggleClass()方法
    }
}
</script>

<!-- 省略了 HTML 结构，请参阅上节示例显示 -->
```

4.9.4 判断样式

1. jQuery 实现

jQuery 定义了 hasClass()方法用来判断元素是否包含指定的类样式。例如：

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    alert($("p").hasClass("red"));  //返回 true
})
</script>

<p class="red">段落文本</p>
```

hasClass()方法实际上是 is()方法的再包装，jQuery 为了方便用户使用，重新定义了 hasClass()专门用来判断指定类样式是否存在。其中$("p").hasClass("red")，可以改写为 $("p").is(".red")。

2. JavaScript 实现

DOM 定义了 hasAttribute()方法，使用该方法可以判断元素是否设置了指定属性，但是它无法判断 class 属性中是否已经包含了指定的类样式。为此，我们可以进行简单扩展，详细代码如下。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("hasClass",function(p){   //自定义 hasClass()方法，并绑定到 HTML 元素身上
    var _this = this;
    if(!_this.hasAttribute("class")) return false;      //如果没有 class 属性,则直接返回 false
    var attr = _this.getAttribute("class");  //获取 class 属性值
    if(attr == p || attr.search(p+" ")>-1 || attr.search(" "+p)>-1 ) return true;
     //如果属性值等于参数值，或者匹配到部分样式为参数样式，则返回 true
    return false;       //如果都不是，则返回 false
})
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    alert(p.hasClass("red1"));  //调用自定义的 hasClass()方法
}
</script>

<p class="red">段落文本</p>
```

4.10 操作 HTML、文本和值

如果直接通过 DOM 结构树来操作文档，有时候会感觉很麻烦。而若把 HTML 文档结构

视为字符串，并以字符串的形式进行操作，会感觉很多问题能够迎刃而解，不用再考虑节点对象和节点之间的关系了。另外，对于文本节点来说，直接把它视为字符串进行操作，更符合一般人的思维习惯。所幸的是，jQuery 和 DOM 都提供了这方面的解决方法。

4.10.1 读写 HTML 字符串

1. jQuery 实现

jQuery 定义了 html()方法，该方法可以以字符串形式读写 HTML 文档结构。当 html()方法不包含参数时，表示以字符串形式读取指定节点下的所有 HTML 结构；当 html()方法包含参数时，表示向指定节点下写入 HTML 结构字符串，同时覆盖该节点原来包含的所有内容。

例如，下面的示例先读取 div 元素及其所有子节点，然后把这些 HTML 结构插入到 p 元素内，演示效果如图 4.25 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    var s = $("div").html();
    $("p").html(s);
})
</script>

<div>
    <h1>标题</h1>
    <p>段落文本</p>
</div>
```

2. JavaScript 实现

jQuery 的 html()方法实际上是对 DOM 中元素对象的 innerHTML 属性包装。该属性可以以字符串形式读写元素包含的 HTML 结构。例如，针对上面的示例，使用 JavaScript 直接设计，则代码如下。

```
<script type="text/javascript" >
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    var div = document.getElementsByTagName("div")[0];
    p.innerHTML = div.innerHTML;
}
</script>

<!-- 省略了 HTML 结构，请参阅上面示例显示 -->
```

4.10.2 读写文本内容

1. jQuery 实现

jQuery 定义了 text()方法用来读写指定元素下包含的文本内容，这些文本内容主要是指文本节点包含的数据。当 text()方法不包含参数时，表示以字符串形式读取指定节点下的所有文本内容；当 text()方法包含参数时，表示向指定节点下写入文本字符串，同时会覆盖该节点原来包含的所有文本内容。

例如，下面的示例先读取 div 元素及其所有子节点包含的文本内容，然后把这些文本内容插入到 p 元素内，演示效果如图 4.26 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    var s = $("div").text();
    $("p").text(s);
}
</script>

<div>
    <h1>标题</h1>
    <p>段落文本</p>
</div>
```

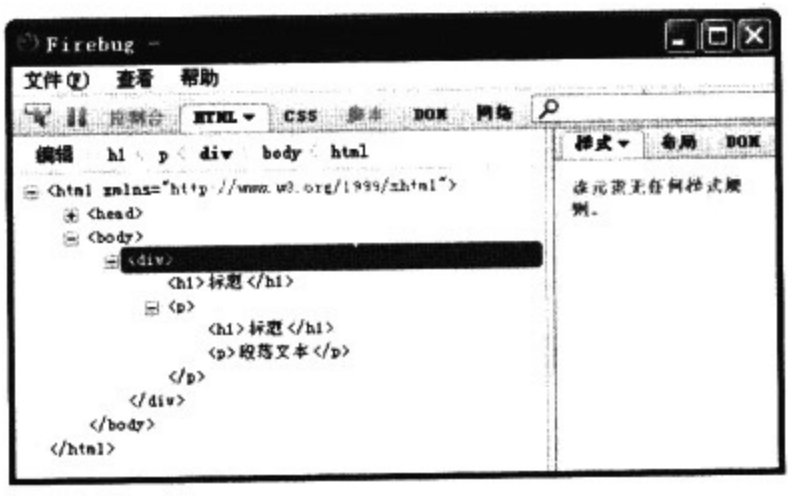


图 4.25 动态读写 HTML 字符串

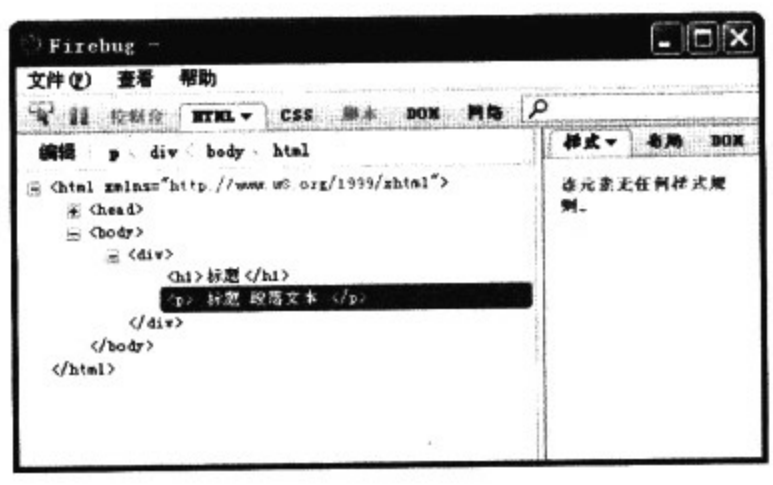


图 4.26 动态读写文本内容

2. JavaScript 实现

jQuery 的 text()方法与 DOM 的 innerText 属性功能相同，遗憾的是 innerText 属性存在兼容性问题，很多标准浏览器不支持该属性。不过，使用 JavaScript 可以自定义 text()方法。完整的 text()自定义方法和调用用法如下，调用示例的演示效果如图 4.26 所示。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
```

```
DOMextend("text",function(s){  //自定义 text()方法，并绑定到 HTML 元素身上
    var _this = this;
    if(s!=undefined){  //如果传递文本内容的参数，则以 HTML 字符串形式写入
        _this.innerHTML = s;
        return _this;
    }
    return sum (_this);     //否则，调用 sum()函数，汇总所有子节点包含的文本内容
    function sum (e){  //自定义内部函数 sum()，迭代指定节点下的所有子节点，并把节点包含的所有文本
内容连接起来，然后返回
        var son = e.childNodes; //获取子节点集合
        var string = "";     //声明临时字符串变量
        for(var i = 0; i < son.length; i ++ ){     //遍历子节点
            if(son[i].nodeType == 3){      //如果节点类型为文本节点，则连接文本节点的文本数据
                string += son[i].data;
            }
            if(son[i].nodeType == 1){      //如果节点类型为元素节点，则调用当前函数，迭代汇总子
节点包含的文本节点内容
                string += arguments.callee(son[i]);
            }
        }
        return string; //返回经过迭代计算后的连接字符串
    }
})
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    var div = document.getElementsByTagName("div")[0];
    p.text(div.text());     //调用 text()方法读取 div 元素包含的文本内容，然后插入到 p 元素内部
}
</script>

<!-- 省略了 HTML 结构，请参阅上面示例显示 -->
```

4.10.3 读写表单值

值是一类特殊的文本字符串，主要是指表单元素中 value 属性设置的值。

1. jQuery 实现

jQuery 定义了 val()方法用来读写指定表单元素包含的值。当 val()方法不包含参数时，调用此方法表示将读取指定表单元素的值；当 val()方法包含参数时，调用此方法表示向指定表单元素写入值。

例如，下面的示例演示了当文本框获取焦点时，清空默认的提示文本信息，准备用户输入值，而当离开文本框后，如果文本框没有输入信息，则重新显示默认的值，如图 4.27 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
```

```
    $("input").focus(function(){
        if($(this).val() == "请输入文本") $(this).val("");
    })
    $("input").blur(function(){
        if($(this).val() == "") $(this).val("请输入文本");
    })
})
</script>

<form action="" method="get">
    <input type="text" value="请输入文本" />
</form>
```

图 4.27　动态控制文本框的值

val()方法在读写单选按钮、复选框、下拉菜单和列表框的值时，比较实用且操作速度比较快。对于 val()方法来说，可以传递一个参数设置表单的显示值。由于下拉菜单和列表框显示为每个选项的文本，而不是 value 属性值，故通过设置选项的显示值，可以决定应显示的项目。不过对于其他表单元素来说，必须指定 value 属性值方才有效。如果为元素指定多个值，则可以以数组的形式进行传递参数。

例如，在下面的示例中，单击第一个按钮可以使用 val()方法读取各个表单的值，单击第二个按钮可以设置表格表单的值。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("button").eq(0).click(function(){  //单击第一个按钮使用 val()方法读取各个表单的值
        alert($("#s1").val() + $("#s2").val() +$("input").val()+ $(":radio").val());
    })
    $("button").eq(1).click(function(){  //单击第二个按钮设置表格表单的值
        $("#s1").val("单选 2");
        $("#s2").val(["多选 2", "多选 3"]);
        $("input").val(["6", "8"]);
    })
})
</script>

<form action="" method="get">
    <select id="s1">
      <option value="1" selected="selected">单选 1</option>
```

```
    <option value="2">单选 2</option>
  </select>
  <select id="s2" size="3" multiple="multiple">
    <option value="3" selected="selected">多选 1</option>
    <option value="4">多选 2</option>
    <option value="5" selected="selected">多选 3</option>
  </select>
  <input type="checkbox" value="6"/>复选框 1
  <input type="checkbox" value="7" checked="checked"/>复选框 2<br />
  <input type="radio" value="8"/>单选按钮 1
  <input type="radio" value="9" checked="checked"/>单选按钮 2<br /><br />
  <button>显示各个表单对象的值</button>
  <button>设置各个表单对象的值</button>
</form>
```

2. JavaScript 实现

jQuery 定义的 val()方法实际上就是包装了 DOM 的 value 属性。不过，val()方法能够根据下拉菜单或列表框的选项显示文本，决定选中的选项，同时，它能够根据单选按钮和复选框的 value 属性值，设置被勾选的选项。当然，我们也可以使用 JavaScript 自定义 val()方法，限于篇幅这里就不再演示。

4.11 操作样式表

在动态网页设计中，样式表是让很多初学者害怕的技术难题。CSS 技术的不兼容性和用法不一，给开发者带来了很多麻烦。从传统设计角度考虑，借助 DOM 的 style 属性，可以读写元素的行内样式，但是它无法访问外部 CSS 样式表。如果使用标准的 DOM 方法访问外部样式表，又存在浏览器兼容性的尴尬。

jQuery 为设计师解决了这些难题，它把所有样式表操作的难题都包容到几个实用方法中，从而在设计时只需简单调用这几个方法，即可解决 Web 开发中很多界面设计难题。当然，直接使用 JavaScript 也能够解决这些，如果读者感兴趣，且愿意优化代码、提高代码执行效率，建议你不妨亲自动手写自己的解决方法。

4.11.1 通用 CSS 样式读写方法

CSS 存在三种形式：行内样式、文档内部样式和文档外部样式。行内样式以元素属性的形式存在，使用 style 属性即可读写，而文档内部样式和文档外部样式统一被视为外部样式，这些外部样式只能够通过 DOM 的 StyleSheets、CSS 和 CSS2 模块提供的对象、方法和属性进行访问和操作。

1. jQuery 实现

jQuery 定义了 css()方法，使用该方法能够读取指定的样式，也能够为元素设置 CSS 样式。例如，下面的示例使用 css()方法分别读取 CSS 的 color 和 font-weight 样式值。但是，即便 jQuery 为我们提供了方便的访问 CSS 样式的途径，但是由于浏览器之间的解析差异化，依然会给开发带来诸多潜在的不便，如 IE 对于颜色值的解析以及对于 font-weight 样式值的解析(如图 4.28 所示)与 Firefox 的解析结果(如图 4.29 所示)是不同的，如果要再比较其他主流浏览器之间的解析结果，可能会有更多答案。

jQuery 也无法解决这样的细节差异，即使解决起来，也会非常繁琐。所以读者应该在开发中避免利用读取的 CSS 样式值作为某种执行的条件。

```
<style type="text/css">
.red {
    color:red;
}
</style>
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").html("color=" +$("p").css("color") + "<br />font-weight=" + $("p").css
("font-weight"));
})
</script>

<p class="red" style="font-weight:bold">段落文本</p>
```

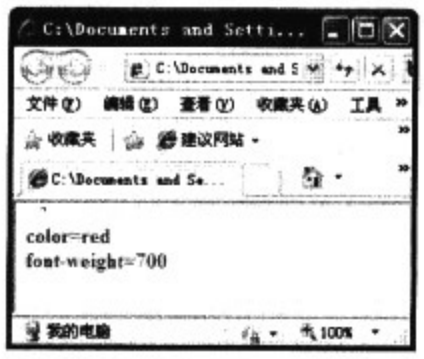


图 4.28 IE 读取 CSS 样式的值

图 4.29 Firefox 读取 CSS 样式的值

通过上面的示例还可以看到，css()方法能够读取指定元素的所有 CSS 样式，不管它是行内样式、内部样式或外部样式。例如，分别用两种传递参数的方法为 p 元素定义 CSS 样式，演示效果如图 4.30 所示。

注意：当使用对象结构传递样式参数时，样式名称不要加引号，但是样式值应该加引号。

下面再来介绍如何使用 jQuery 的 css()方法为元素定义样式，实现代码如下。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
```

```
<script type="text/javascript" >
$(function(){
    $("p").css("font-style","italic");  //设置单个样式
    $("p").css({color:"red", fontWeight:"bold"});     //以对象结构的形式传递多个样式
})
</script>

<p>段落文本</p>
```

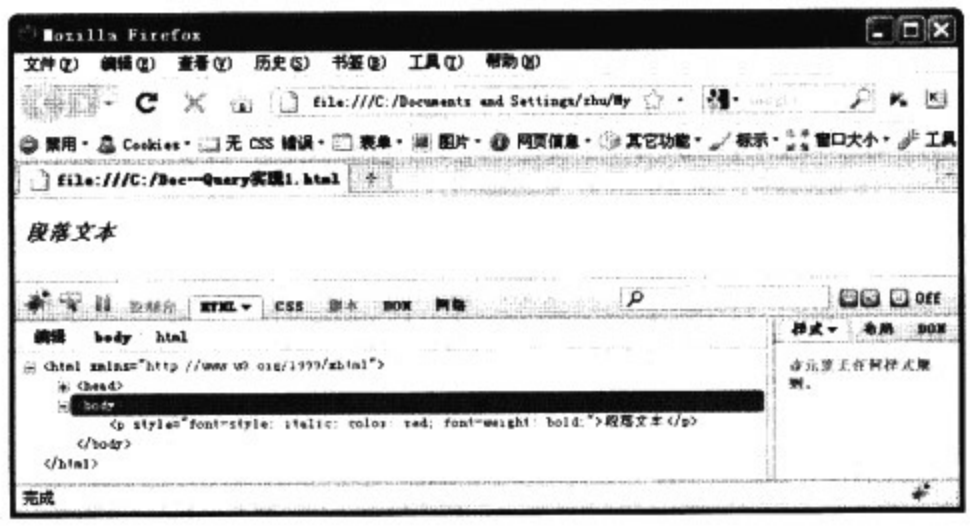


图 4.30 动态设置的 CSS 样式

通过上面的演示效果图，也可以看到写入的 CSS 样式是以行内样式而存在的。

2. JavaScript 实现(读写行内样式)

为了方便读写 CSS 样式，0 级 DOM 定义了 Style 对象，并允许 Element 对象通过 style 属性进行访问。Style 对象包含大量 CSS 样式属性，这些属性与 CSS 属性一一对应，但是它们的名称略有不同。详细说明如下。

- 对于独立单词的 CSS 样式来说，Style 对象以同名来表示对应的 CSS 脚本属性。例如，使用 style.color 可以访问 color 样式。
- 对于复合词的 CSS 样式来说，由于 CSS 样式使用连字符连接多个单词(如 border-right)，但是在 JavaScript 中连字符被约定为减号运算符，如果不进行处理 JavaScript 会误以为是表达式进行运算。因此 Style 对象的属性名与 CSS 属性名是不同的。如果对应的 CSS 属性名包含一个或多个连字符，则 Style 对象就会删除这些连字符，并以驼峰命名法重命令脚本属性名。例如，对于 border-right 样式属性来说，在脚本中就必须使用 borderRight 脚本属性来访问 border-right 样式属性。
- 由于 float 是 Java 及其他语言中的关键字，JavaScript 虽然没有把它作为关键字，但作为保留字禁止用户使用。因此，Style 对象没有直接与 float 样式属性对应的脚本属性名。为了解决这个问题，Style 对象在 float 样式属性名前增加了 css 前缀，使用 cssFloat 属性名来表示与 float 样式属性对应的脚本属性。
- Style 对象约定所有 CSS 脚本属性值都是字符串，因此在 JavaScript 中为脚本属性赋值时必须加上引号，以字符串数据类型进行传递。

- 在 CSS 中，样式声明的尾部会添加分号，但是在脚本属性中分号就不能作为属性值的一部分被引用，因为分号是 JavaScript 的语法组成部分。
- 当为脚本属性赋值时，必须包含值和完整的单位，若省略单位，则所设置的脚本样式无效。
- 在 JavaScript 中可以使用变量动态设置脚本样式的属性值，但是变量的值最后以字符串形式存在。另外不要忽略了属性值的单位。

下面看一个示例，该示例是使用 JavaScript 来模仿上面的示例进行设计的。在预览时，会发现使用 Style 对象只能够读取行内样式，效果如图 4.31 所示。

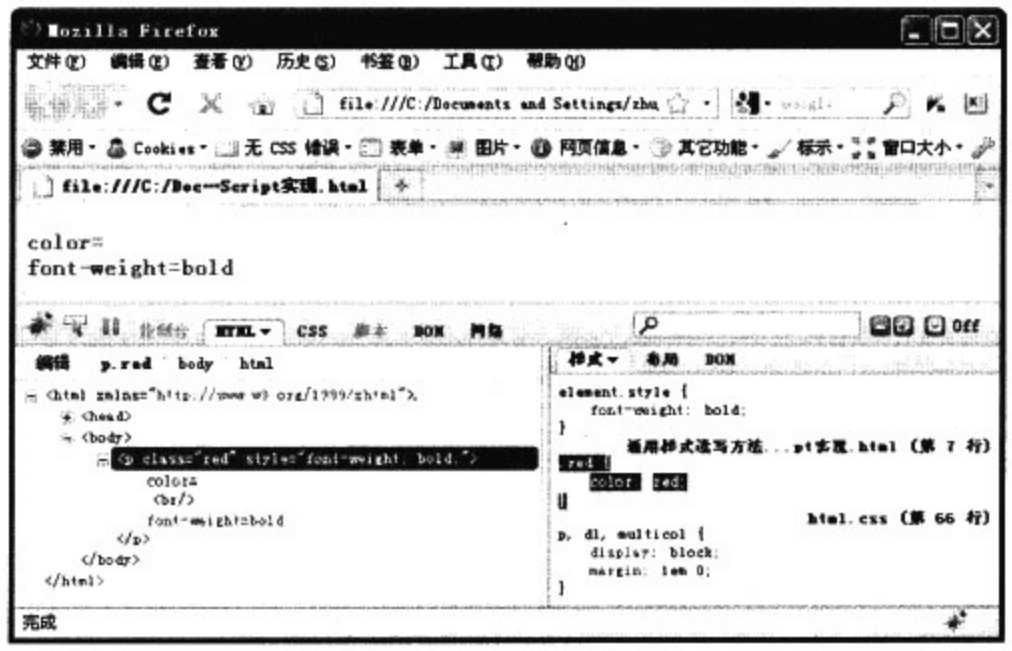


图 4.31 使用 JavaScript 读取 CSS 样式值

```
<style type="text/css">
.red {
    color:red;
}
</style>
<script type="text/javascript" >
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    p.innerHTML = "color=" + p.style.color +  "<br />font-weight=" + p.style.fontWeight
}
</script>

<p class="red" style="font-weight:bold">段落文本</p>
```

下面介绍如何使用 DOM 的 Style 对象为元素定义样式，实现代码如下，演示效果如图 4.32 所示。

```
<script type="text/javascript" >
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    p.style.color = "red";//定义颜色样式
```

```
    p.style.fontWeight = "bold";    //定义粗体样式
}
</script>

<p>段落文本</p>
```

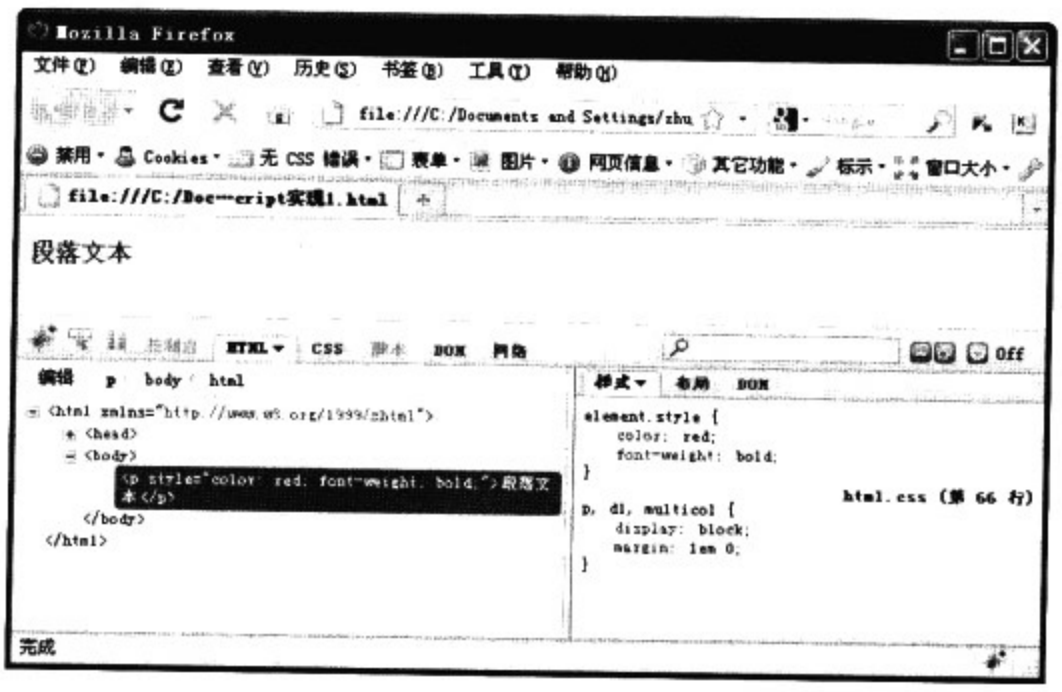


图 4.32　使用 JavaScript 定义 CSS 样式

3. JavaScript 实现(读写样式表)

使用 Style 对象可以获取指定元素的行内样式，但是无法获取由 style 元素定义的内部样式表，以及使用 link 元素或@import 命令导入的外部样式表。要读写 CSS 样式表，则可以使用 Document 对象的 styleSheets 集合实现。styleSheets 集合包含了文档中所有样式表的引用。例如，所有<style>标签定义的内部样式表，以及使用 link 元素或@import 命令导入的外部样式表。

DOM 还为每个样式表定义了一个 cssRules 集合，用来包含指定样式表中所有的规则。但是 IE 浏览器不支持 cssRules 集合，而是定义 rules 集合来支持相同的操作。如果需要同时支持 IE 和 Firefox 等主流浏览器，可以使用下面的代码进行兼容。

```
var cssRules = document.styleSheets[0].cssRules || document.styleSheets[0].rules;
```

上面代码先判断浏览器是否支持 cssRules 集合，如果支持则调用 cssRules 集合，否则使用 rules 集合。

cssRules 集合和 rules 集合都包含 style 属性，用来访问 Style 对象，并通过 Style 对象包含的样式脚本属性，读写具体的样式。

例如，在下面的示例中，文档中仅包含一个样式表，故要访问它可以使用 document.styleSheets[0]，而该样式表中仅包含一个样式，故可以先使用 cssRules[0]进行访问，然后借助 cssRules 对象的 style 属性访问 Style 对象，并读取 color 样式属性的值。示例代码如下，演示效果如图 4.33 所示。

```
<style type="text/css">
.red {
    color:red;
}
</style>
<script type="text/javascript" >
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    var cssRules = document.styleSheets[0].cssRules || document.styleSheets[0].rules;
    //访问样式
    alert(cssRules[0].style.color);     //访问样式属性值，弹出"red"字符串
}
</script>

<p class="red" style="font-weight:bold">段落文本</p>
```

在样式表中写入样式也很简单，例如，以上面示例为基础，在当前样式中写入一个加粗样式，实现代码如下，演示效果如图 4.34 所示。

```
<style type="text/css">
.red {
    color:red;
}
</style>
<script type="text/javascript" >
window.onload = function(){
    var p = document.getElementsByTagName("p")[0];
    var cssRules = document.styleSheets[0].cssRules || document.styleSheets[0].rules;
    cssRules[0].style.fontWeight = "bold";       //写入样式
}
</script>

<p class="red" style="">段落文本</p>
```

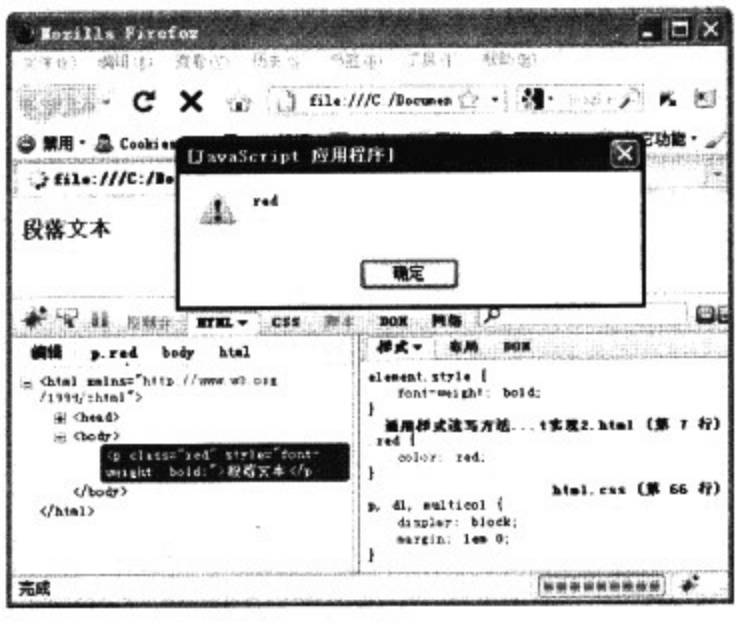


图 4.33　使用 JavaScript 读取样式表中的样式值

图 4.34　使用 JavaScript 在样式表中写入样式

读者也可以利用现有样式表，创建新的样式，并定义各种属性。例如，在下面的示例中为当前样式表添加一个标签样式，并定义背景色为蓝色，字体颜色为白色，实现代码如下，演示效果如图 4.35 所示。由于样式的优先级缘故，文本仍然会显示为红色。

```
<style type="text/css">
.red {
    color:red;
}
</style>
<script type="text/javascript" >
window.onload = function(){
    var n = document.styleSheets[0].length; //获取当前样式表中包含样式的个数
    if(document.styleSheets[0].insertRule){      //兼容非 IE 浏览器
        document.styleSheets[0].insertRule("p{background-color:blue;color:#fff;}",
n); //插入样式
    }else{    //兼容 IE 浏览器
        document.styleSheets[0].addRule("P", "background-color:blue;color:#fff;",n);
    //添加建样式
    }
}
</script>

<p class="red">段落文本</p>
```

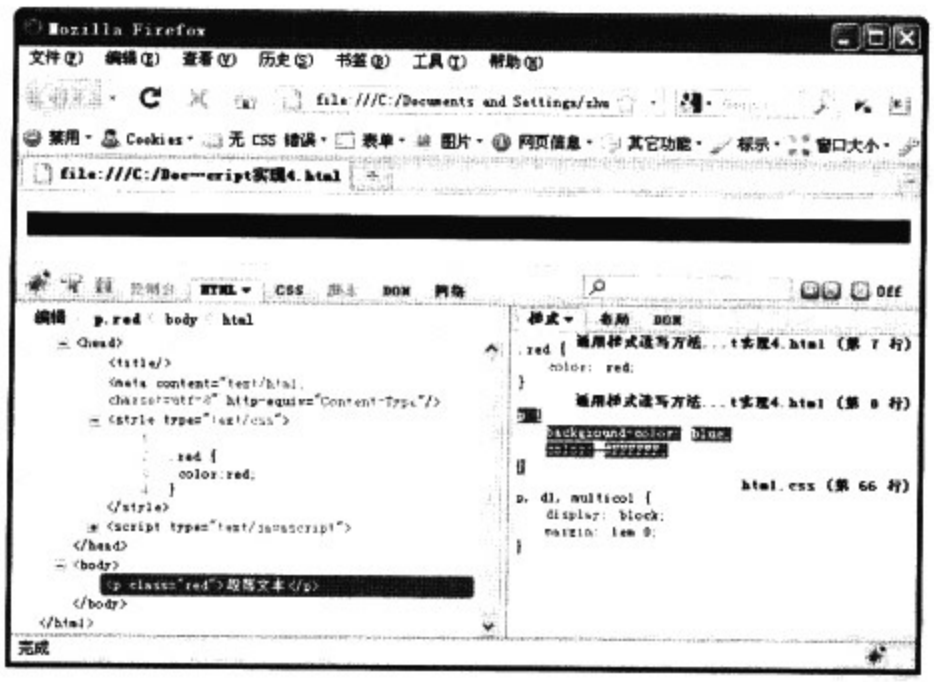


图 4.35　动态添加的样式效果

addRule()方法是 IE 浏览器的专用方法，它能够为样式表增加一个样式，语法如下。

```
styleSheet.addRule(selector,style ,index))
```

其中 styleSheet 表示样式表索引对象，参数 selector 表示样式的选择符，必须以字符串的形式传递；style 表示声明样式的字符串，格式与 CSS 样式的格式相同；参数 index 表示索引号，设置新建样式在样式表中的索引位置，默认为-1，表示位于样式表的末尾，该参数可

以不设置。

insertRule()方法是非 IE 浏览器的专用方法，它也能够为样式表增加一个样式，语法如下。

```
styleSheet.insertRule(rule ,index))
```

其中参数 rule 是一个完整的 CSS 样式字符串，包括选择符和声明两部分，格式必须与 CSS 样式的格式相同；参数 index 与 addRule()方法中的 index 参数作用相同，但是该参数必须设置，当值为 0 时表示放置在样式表的末尾。

4.11.2 绝对偏移位置

所谓绝对偏移位置就是指定元素距离浏览器窗口左上角的偏移距离。

1. jQuery 实现

jQuery 定义了 offset()方法，该方法能够获取匹配元素在当前窗口的相对偏移。该方法没有参数，返回值为一个对象，包含两个属性：top 和 left 属性，分别存储匹配元素的顶部偏移和左侧偏移。注意，该方法仅对可见元素有效。

例如，下面的示例演示了如何获取三个 div 元素的绝对偏移位置，演示效果如图 4.36 所示。为了方便比较和观察，在样式表中定义页边距为 0，并定义 div 元素的大小固定，且边框都为 10 像素宽。

```
<style type="text/css">
body {//清除页边距
    padding:0;
    margin:0;
}
div {//统一 div 元素的显示样式
    height:60px;
    width:200px;
    border:solid 10px red;
}
</style>
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    var o1 = $("div").eq(0).offset();    //获取第一个 div 元素的偏移信息
    $("div").eq(0).html( "left: " + o1.left + "<br />top: " + o1.top );     //显示信息
    var o2 = $("div").eq(1).offset();    //获取第二个 div 元素的偏移信息
    $("div").eq(1).html( "left: " + o2.left + "<br />top: " + o2.top );     //显示信息
    var o3 = $("div").eq(2).offset();    //获取第三个 div 元素的偏移信息
    $("div").eq(2).html( "left: " + o3.left + "<br />top: " + o3.top );     //显示信息
})
</script>
```

```
<div>盒子 1</div>
<div style="float:left">盒子 2</div>
<div style="float:left">盒子 3</div>
```

2. JavaScript 实现

我们继续以上面的示例为基础，演示如何直接使用 JavaScript 获取元素的绝对偏移位置，并使用 JavaScript 自定义 offset()方法。代码如下，演示效果与图 4.36 所示相同。

```
<style type="text/css">
//省略了两个样式，请参阅上面示例的内部样式表
</style>
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("offset",function(){ //自定义 offset()方法，并绑定到 HTML 元素上
    var _this = this
    var left = 0, top = 0; //声明并初始化坐标变量
    while(_this.offsetParent){ //如果存在 offsetParent 属性，则获取并定位元素的偏移坐标
        left += _this.offsetLeft;    //叠加 x 轴偏移距离
        top += _this.offsetTop; //叠加 y 轴偏移距离
        _this = _this.offsetParent; //迭代计算
    }
    return {  //返回一个对象，该对象包含当前元素的偏移坐标
        "left" : left,
        "top" : top
    };
})
window.onload = function(){
    var div = document.getElementsByTagName("div")[0];    //获取第一个 div 元素
    var o = div.offset();  //获取偏移信息
    div.innerHTML = "left: " + o.left + "<br />top: " + o.top ;    //显示偏移信息
    var div = document.getElementsByTagName("div")[1];    //获取第二个 div 元素
    var o = div.offset();  //获取偏移信息
    div.innerHTML = "left: " + o.left + "<br />top: " + o.top ;    //显示偏移信息
    var div = document.getElementsByTagName("div")[2];    //获取第三个 div 元素
    var o = div.offset();  //获取偏移信息
    div.innerHTML = "left: " + o.left + "<br />top: " + o.top ;    //显示偏移信息
}
</script>

<div>盒子 1</div>
<div style="float:left">盒子 2</div>
<div style="float:left">盒子 3</div>
```

在上面的解决方案中，读者需要明白元素的相对偏移问题。DOM 约定任何元素都拥有 offsetLeft 和 offsetTop 属性，它们描述了元素的最近偏移位置。

但是不同浏览器定义元素的偏移参照对象不同。例如，IE 总是以父元素为参照对象进行偏移，而非 IE 浏览器会以最近非静态定位元素为参照对象进行偏移。

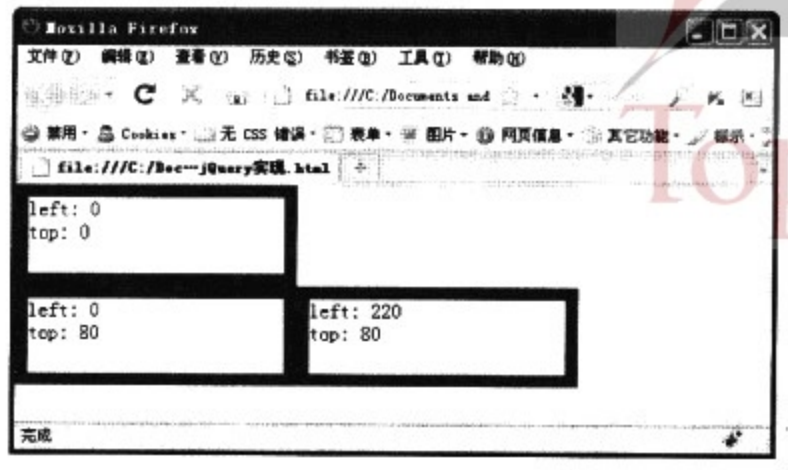


图 4.36　获取元素的绝对偏移位置

尽管元素偏移定位存在兼容性问题，但是所有浏览器都支持 offsetParent 属性，由于 offsetParent 属性总能够自动识别当前元素偏移的参照对象，所以不用担心 offsetParent 在不同浏览器中具体指代什么元素。因此，我们可以不考虑浏览器兼容性问题，通过迭代方法计算当前元素距离窗口左上角的偏移距离，而不去关心各个浏览器需要经过几次偏移定位，才能够到达窗口左上角。

4.11.3　相对偏移位置

所谓相对偏移位置就是指定元素距离最近父级定位元素左上角的偏移距离。这里读者首先需要弄明白以下两个概念。

第一，定位元素就是被定义了相对、绝对或固定定位的元素，即设置了 CSS 的 position 属性值为 absolute、fixed 或 relative 属性值的元素。

第二，所谓父元素是指与当前元素相邻的上一级元素，而最近的父级元素不一定是与当前元素相邻，也可能距离很远。如果当前元素的上级元素 position 属性值都没有被定义为 absolute、fixed 或 relative，则当前元素的最近父级定位元素就应该是 body 元素了，此时相对偏移位置与绝对偏移位置是相同的。

1. jQuery 实现

jQuery 定义了 position()方法，使用该方法可以获取匹配元素的相对偏移位置。该方法的用法与 offset()方法相同，都返回一个包含两个属性(即 top 和 left)的对象。注意，为精确计算结果，请在补白、边框和填充属性上使用像素单位，该方法只对可见元素有效。

例如，在下面的示例中，分别定义两个 div 元素，一个直接放在文档中，另一个包裹在被定义了相对定位的盒子中，同时设置这个盒子向右浮动。最后使用 position()方法读取这两个 div 元素的相对偏移位置，则效果如图 4.37 所示。

```
<style type="text/css">
body {
```

```
    padding:0;
    margin:0;
}
div {
    height:60px;
    width:200px;
    border:solid 10px red;
}
</style>
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    var o1 = $("div").eq(0).position(); //获取元素的相对偏移位置
    $("div").eq(0).html( "left: " + o1.left + "<br />top: " + o1.top );//显示相对偏移位置
    var o2 = $("div").eq(2).position(); //获取元素的相对偏移位置
    $("div").eq(2).html( "left: " + o2.left + "<br />top: " + o2.top );//显示相对偏移位置
})
</script>

<div>盒子 1</div>
<div style="position:relative; float:right; width:300px; height:100px; border-color:blue;">
    <div>盒子 2</div>
</div>
```

在上面的示例中，如果取消定义盒子元素的行内样式 position:relative;，则可以看到两个 div 元素都会以窗口左上角为坐标原点进行定位测量，演示效果如图 4.38 所示。

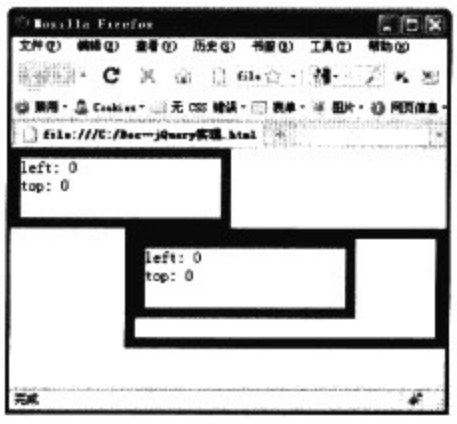


图 4.37 为盒子元素定义相对定位后的效果

图 4.38 没有为盒子元素定义相对定位后的效果

2. JavaScript 实现

我们继续以上面的示例为基础，演示如何直接使用 JavaScript 获取元素的相对偏移位置，并使用 JavaScript 自定义 offset()方法。代码如下，演示效果与图 4.34 和图 4.35 所示相同。

```
<style type="text/css">
//省略了两个样式，请参阅上面示例的内部样式表
</style>
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
```

```
DOMextend("position",function(){    //自定义position()方法，并绑定到HTML元素上
    var _this = this
    if(_this.parentNode == _this.offsetParent){  //如果父元素就是定位元素
        var px = parseInt(getStyle(_this.parentNode,"borderLeftWidth")) || 0;
     //获取父元素的左侧边框宽度
        var py = parseInt(getStyle(_this.parentNode,"borderTopWidth")) || 0;
     //获取父元素的顶部边框宽度
        var x = _this.offsetLeft;     //获取当前元素的x轴相对位置
        var y = _this.offsetTop;      //获取当前元素的y轴相对位置
        if(document.all){   //兼容IE浏览器
              x = _this.offsetLeft - px;      //减去父元素边框的宽度
              y = _this.offsetTop - py;       //减去父元素边框的宽度
        }
    }
    else{ //如果定位元素不是父元素
        var o = offset(_this);          //获取当前元素的绝对偏移信息
        var p = offset(_this.offsetParent);  //获取最近定位元素的绝对偏移信息
        var x = o.left - p.left;        //计算当前元素距离定位元素的x轴偏移距离
        var y = o.top - p.top;          //计算当前元素距离定位元素的y轴偏移距离
    }
    return {  //返回相对偏移信息
        "left" : x,
        "top" : y
    };
    function offset(_this){     //获取指定元素的绝对偏移信息函数，请参阅上节讲解
        var left = 0, top = 0;
        while(_this.offsetParent){
            left += _this.offsetLeft;
            top += _this.offsetTop;
            _this = _this.offsetParent;
        }
        return {
            "left" : left,
            "top" : top
        };
    }
    function getStyle(e,n){     //获取元素指定样式的属性值函数。其中参数e表示具体的元素，n表示要获取元素的样式脚本属性名，如"borderColor"
        if(e.style[n]){     //如果指定样式属性在Style对象中存在，说明已显式定义该样式，则直接返回这个样式的属性值
            return e.style[n];
        }
        else if(e.currentStyle){     //否则，如果是IE浏览器，则调用私有对象currentStyle读取当前样式的属性值
            return e.currentStyle[n];
        }
        else if(document.defaultView && document.defaultView.getComputedStyle){ // 如果是支持DOM标准的浏览器，则利用DOM定义的方法读取当前样式的属性值
            n = n.replace(/([A-Z])/g,"-$1");  //把脚本样式属性名转换为CSS样式属性名
            n = n.toLowerCase(); //转换为小写格式
            var s = document.defaultView.getComputedStyle(e,null);  //获取当前元素的样式
```

```
属性对象
            if(s)   //如果当前元素的样式属性对象存在
                return s.getPropertyValue(n);       //则获取属性值
        }
        else   //如果都不支持，则返回null
            return null;
    }
})
window.onload = function(){
    var div = document.getElementsByTagName("div")[0];     //获取div元素
    var o = div.position();      //获取相对偏移信息
    div.innerHTML = "left: " + o.left + "<br />top: " + o.top ;     //显示相对偏移信息
    var div = document.getElementsByTagName("div")[2];     //获取div元素
    var o = div.position();      //获取相对偏移信息
    div.innerHTML = "left: " + o.left + "<br />top: " + o.top ;     //显示相对偏移信息
}
</script>

<div>盒子1</div>
<div style=" position:relative; float:right; width:300px; height:100px; border- color:blue;">
    <div>盒子2</div>
</div>
```

使用 JavaScript 实现获取指定元素的相对偏移位置稍显麻烦，但是其设计思路比较简单，设计的核心就是利用 offsetParent 属性获取最近的父级定位元素，然后判断该元素的位置。如果它是父元素，则可以直接读取当前元素的 offsetLeft 和 offsetTop 属性值；如果不是父元素，则可以将获取的当前元素的绝对偏移位置减去定位元素的绝对偏移位置，即可获得当前元素距离定位元素的偏移距离。

因此，在本案例的解决方法中需要用到上一节讲解的 offset()方法来获取当前元素和定位元素的绝对偏移位置。同时，由于不同浏览器对于 offsetLeft 和 offsetTop 属性值的解析方式不同，当直接调用该属性读取相对偏移位置时，可能存在误差。

其中，IE 浏览器会加上父元素的边框，因此还必须使用 offsetLeft 和 offsetTop 属性值减去父元素的边框值。如果当元素定义了 border 样式后，我们可以直接读取该样式值，但是如果元素没有显示定义父元素的边框，或者通过脚本动态设置了元素的边框，就无法读取这个值，此时就必须根据不同浏览器提供的私有解决方法分别进行读取，为了方便操作，我们把这个操作单独定义为 getStyle()函数，这样在需要时直接调用即可。

4.11.4 扩展 DOM 操作函数

在上一节中介绍了 getStyle()自定义方法，这个方法比较实用，它能够获取任何形式元素指定样式的属性值，不管该元素是显式为定义了样式，还是显示为默认样式，或者显示为动态设置的样式，调用该方法都能够获取指定样式的属性值。为了方便后期开发，不妨把这个

方法绑定到 DOM 元素上，这样就可以随时调用，而不再重复复制该函数。绑定代码如下。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("getStyle",function(n){  //自定义 getStyle()方法，并绑定到 HTML 元素上。其中参数 n
表示样式的脚本属性名
    var _this = this
    if(_this.style[n]){
        return _this.style[n];
    }
    else if(_this.currentStyle){
        return _this.currentStyle[n];
    }

else if(document.defaultView && document.defaultView.getComputedStyle)
        {
        n = n.replace(/([A-Z])/g,"-$1");
        n = n.toLowerCase();
        var s = document.defaultView.getComputedStyle(_this,null);
        if(s)
            return s.getPropertyValue(n);
    }
    else
        return null;
})
</script>
```

使用 getStyle()方法获取的样式属性值为字符串，值中包含单位，有时返回值还可能包含 auto 默认值，显然这样的返回值不适合 JavaScript 运算。所以，我们再为 DOM 元素扩展一个 fromStyle()方法，以便把 getStyle()方法获取的值转换为可以参与运算的数字。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
//省略了自定义 getStyle()函数，请参阅第上一段代码内容
DOMextend("fromStyle",function(w, p){  //自定义 fromStyle()方法，并绑定到 HTML 元素上。其中
参数 w 表示元素的样式属性值，该值通过 getStyle()方法获取，参数 p 表示将当前元素百分比转换为小数的值，
以便在上级元素中计算出当前元素的尺寸
    var _this = this
    var p = arguments[2];   //获取百分比转换后的小数值
    if( ! p) p = 1;     //如果不存在，则默认其为 1
    if(/px/.test(w) && parseInt(w) ) return parseInt(parseInt(w) * p);      //如果元素
尺寸的值为具体的像素值，则直接转换为数字，并乘以百分比值，并返回该值
    else if(/\%/.test(w) && parseInt(w)){    //如果元素宽度值为百分比值
        var b = parseInt(w) / 100;   //则把该值转换为小数值
        if((p != 1) && p) b *= p;    //如果子元素的尺寸也是百分比，则乘以转换后的小数值
        _this = _this.parentNode;    //获取父元素的引用指针
        if(_this.tagName == "BODY") throw new Error("文档结构无尺寸，请使用其他方法获取尺寸.");
     //如果父元素是 body 元素，则抛出异常
        w = _this.getStyle("width");      //调用 getStyle()方法，获取父元素的宽度值
        return arguments.callee(_this, w, b);      //回调函数，把上面的值作为参数进行传递，实
```

```
现迭代计算
    }
    else if(/auto/.test(w)){    //如果元素宽度值为默认值
        var b = 1;       //定义百分比值为 1
        if((p != 1) && p) b *= p;    //如果子元素的尺寸是百分比，则乘以转换后的小数值
        _this = _this.parentNode;    //获取父元素的引用指针
        if(_this.tagName == "BODY") throw new Error("文档结构无尺寸，请使用其他方法获取尺寸.");
    //如果父元素是 body 元素，则抛出异常
        w = _this.getStyle("width"); //调用 getStyle()方法，获取父元素的宽度值
        return arguments.callee(_this, w , b);    //回调函数，实现迭代计算
    }
    else //如果 getStyle()函数返回值包含其他单位，则抛出异常，如 em、ex 等，由于它们不经常使用，故
不再进行处理
        throw new Error("元素或其父元素的尺寸定义了特殊的单位.");
})
</script>
```

通过上面的方法，如果元素设置的样式值为 auto 或者百分比，则也能够自动计算出元素的实际值。因为 auto 总是等于父元素对应样式的属性值，但是只能通过人工计算才能够获取。同理百分比取值是也是根据父元素对应样式的属性值进行计算的。如果元素处于多层嵌套的结构中，当各个层次元素的取值单位都不确定时，只能够通过迭代方式进行计算，从而最终获取元素的实际属性值。

如果元素被隐藏显示，也就是说设置样式属性 display 的值为 none 时，则使用 getStyle()方法获取的样式会出现误差。为了解决这个问题，我们可以临时设置元素将其显示出来，当获取它的样式值之后，再恢复它隐藏显示。为了完整类似这样的临时性动态设置样式的设计需要，下面再为 DOM 元素扩展两个方法，其中 setCSS()方法能够批量为元素设置多个样式，并保存和返回元素修改前的样式设置。而 resetCSS()方法能够恢复临时修改元素样式之后，恢复它的默认显示效果。如果配合使用 setCSS()和 resetCSS()方法，可以在程序中随时修改元素的显示样式，然后再恢复修改前的状态，从而可以方便设计复杂的动画效果，实现代码如下所示。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("setCSS",function(o){ {   //自定义 setCSS()方法，并绑定到 HTML 元素上。其中参数 o 表
示要设置的样式对象，可包含多个名值对
    var _this = this
    var a = {};    //声明临时对象
    for(var i in o){   //遍历参数对象
        a[i] = _this.style[i]; //先备份元素被修改前的样式
        _this.style[i] = o[i];  //修改样式
    }
    return a;      //返回备份样式
})
DOMextend("resetCSS",function(o){  //自定义 resetCSS()方法，并绑定到 HTML 元素上。其中参数 o
表示要还原的样式对象，可包含多个名值对，该对象可以为 setCSS()方法的返回值
```

```
    var _this = this
    for(var i in o){    //遍历参数对象
        _this.style[i] = o[i];   //恢复默认样式
    }
})
</script>
```

4.11.5 元素的宽和高

在 Web 开发中，经常需要控制元素的大小。使用 CSS 技术可以直接设置大小，但是如果没有显式定义元素的宽和高，获取元素大小就比较麻烦。

1. jQuery 实现

jQuery 定义了 width()和 height()方法，利用这两个方法可以很轻松读写元素的大小。下面这个示例演示了如何读取元素的大小，以及如何动态设置元素的大小，演示效果如图 4.39 所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("div").html("height(原)=" + $("div").height() + "<br />width(原)=" + $("div").width());
     //获取元素设置前的宽和高
    $("div").height(140px);      //重设元素高度为 140px
    $("div").width("30em");      //重设元素宽度为 30em
    $("div").html( $("div").html() + "<br />height(现)=" + $("div").height() + "<br />width(现)="
+ $("div").width());   //获取元素设置前的宽和高
})
</script>

<div style="border:solid 10px red;">盒子</div>
```

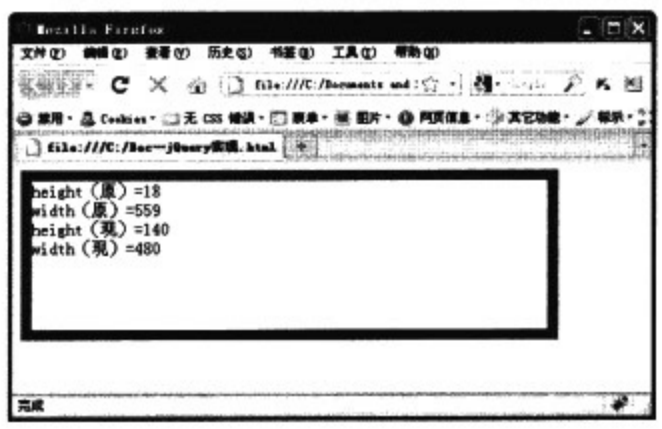


图 4.39 读写元素的宽和高

width()和 height()方法在没有传递参数时，表示读取元素的宽度和高度，返回值的单位为像素。它们可以通过传递参数来设置元素的宽和高，如果直接传递一个数值，则默认单位为 px。也可以以字符串形式传递值和单位。

2. JavaScript 实现

使用 JavaScript 直接实现 jQuery 的 height()和 width()方法的功能，需要借助 4.11.4 节讲解的4个DOM扩展函数。由于使用JavaScript可以直接设置元素的宽和高，所以使用JavaScript自定义的 height()和 width()方法只需要能够获取元素的宽和高即可，详细代码如下，演示效果与图 4.39 所示相同。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
//省略了自定义 DOM 扩展方法 getStyle()、fromStyle()、setCSS()和 resetCSS()，请参阅第 4.11.4 节内容
DOMextend("width",function(){  //自定义 width()方法，并绑定到 HTML 元素上
    var _this = this
    if(_this.getStyle("display") != "none") return _this.offsetWidth ||
    _this.fromStyle(_this.getStyle("width"));     //如果元素没有隐藏，则读取它的宽度
    var r = _this.setCSS({ //如果元素隐藏，则调用 setCSS()方法显示元素，并备份样式
        display:"",
        position:"absolute",
        visibility:"hidden"
    });
    var w = _this.offsetWidth || _this.fromStyle(_this.getStyle("width")); //获取元素
的宽度
    _this.resetCSS(r);      //恢复元素的隐藏显示
    return w; //返回元素的宽度
})
DOMextend("height",function(){ //自定义 height()方法，并绑定到 HTML 元素上
    var  this = this
    if(_this.getStyle("display") != "none") return _this.offsetHeight ||
_this.fromStyle(_this.getStyle("height"));  //如果元素没有隐藏，则读取它的高度
    var r = _this.setCSS({ //如果元素隐藏，则调用 setCSS()方法显示元素，并备份样式
        display:"",
        position:"absolute",
        visibility:"hidden"
    });
    var h = _this.offsetHeight || _this.fromStyle(_this.getStyle("height"));//获取元素
的高度
    _this.resetCSS(r);      //恢复元素的隐藏显示
    return h;      //返回元素的高度
})
window.onload = function(){
    var div = document.getElementsByTagName("div")[0];     //获取元素
    div.innerHTML = "height(原)=" + div.height() + "<br />width(原)=" + div.width();
     //显示元素的宽度和高度
    div.style.height = "140px"; //重设元素高度
    div.style.width = "30em";   //重设元素宽度
    div.innerHTML += "<br />height(现)=" + div.height() + "<br />width(现)=" + div.width();
         //显示元素的宽度和高度
}
</script>

<div style="border:solid 10px red;">盒子</div>
```

3. jQuery 实现的其他宽和高的方法

除了 height()和 width()方法，jQuery 还定义了 innerHeight()、innerWidth()、outerHeight()和 outerWidth()方法，这些方法实际上是在 height()和 width()方法基础上，计算了元素的边框或补白。其中 outerHeight()和 outerWidth()方法能够返回元素的总宽和总高(包括宽高、补白和边框宽度)，innerHeight()和 innerWidth()方法能够返回元素的内容宽度和高度(包括宽高和补白)。

例如，下面的示例分别演示了如何计算元素的总宽、总高、内容宽度和内容高度，演示效果如图 4.40 所示。

```
<style type="text/css">
div {
    width:200px;
    height:50px;
    margin:50px;
    padding:50px;
    border:solid 50px red;
}
</style>
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("div").html("innerHeight=" + $("div").innerHeight() + "<br />innerWidth=" +
$("div").innerWidth());
    $("div").html( $("div").html() + "<br />outerHeight=" + $("div").outerHeight() +
"<br />outerWidth=" + $("div").outerWidth());
})
</script>

<div>盒子</div>
```

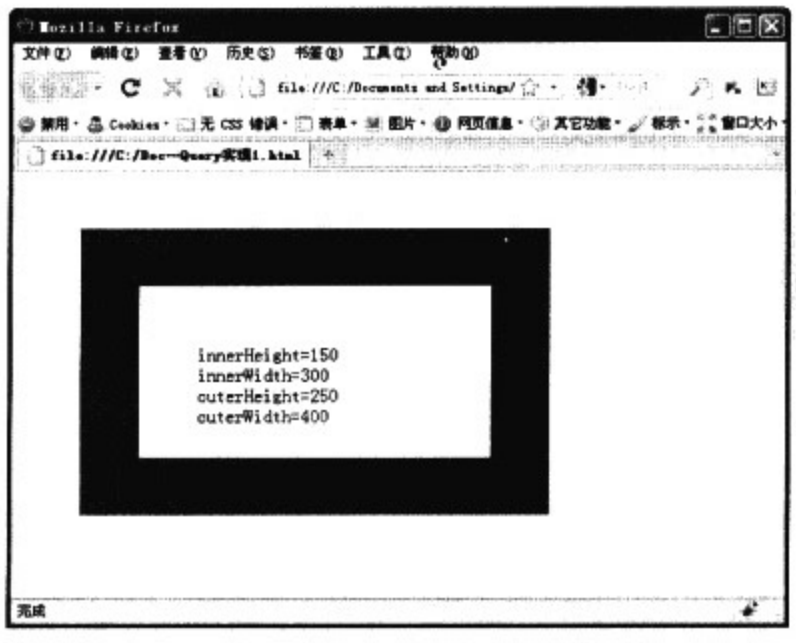


图 4.40　获取元素的总宽、总高、内容宽度和内容高度

4.12 元素遍历操作

DOM为Node类型定义了childNodes、parentNode、nextSibling、previousSibling、firstChild和lastChild属性，这些属性为节点之间相互访问提供了支持。开发人员正是利用这些指针属性，从而可以自由地在文档结构内进行访问。

由于DOM提供的这些指针属性是针对节点而言的，但是节点类型又有很多种，并且在实际开发中开发人员更多的是需要遍历元素，而不是遍历文本或注释等类型节点。因此，直接使用这些属性有时会感到不便。jQuery考虑到开发人员的实际需求，对这些属性进行了包装，以方便用户快速访问。

4.12.1 jQuery实现的元素遍历方法

jQuery定义了children()、next()、prev()和parent()四个基本元素遍历方法，使用它们可以轻松访问文档中任何元素。其中children()方法获取当前元素包含的所有子元素，next()方法获取当前元素相邻的下一个同级元素，prev()方法获取当前元素相邻的上一个同级元素，parent()方法获取当前元素的父元素。不过这些方法的返回值都是jQuery对象，而不是DOM集合或对象。

例如，在下面的示例中，借助jQuery定义的基本指针函数，从body元素开始，沿着DOM结构树，一步步访问到li元素，并修改文档中三个li元素包含的文本内容，演示效果如图4.41所示。

```
<script src="jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    var $body = $("body"); //获取body元素
    var li = $body.children().eq(2).children()[0];//利用 children()方法，遍历到第一个 li
元素
    $(li).text("第1项").next().text("第2项").next().text("第3项");//利用next()方法，遍
历li元素，并修改每个li元素的文本内容
})
</script>

<h1>DOM文档对象模型</h1>
<p>DOM 是 Document Object Model 短语的缩写，中文翻译为文档对象模型，根据 W3C DOM 规范
(http://www.w3.org/DOM/ )，DOM是一种与浏览器、平台、语言无关的接口，使用该接口可以访问页面其他
的标准组件。</p>
<ul>
    <li>D是Document一词的简写，表示文档，它是DOM的表现层。</li>
    <li>O是Object一词的简写，表示对象，它是DOM的逻辑层。</li>
```

```
    <li>M 是 Model 一词的简写，表示模型，它是 DOM 的交互层。</li>
</ul>
```

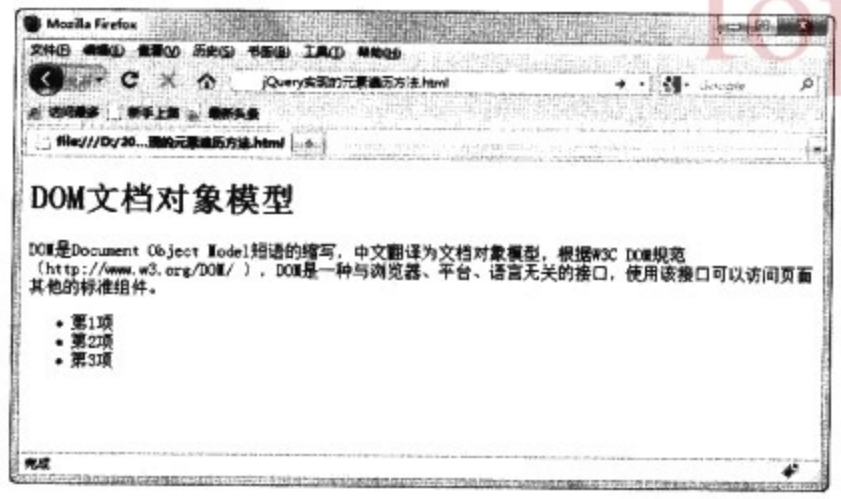


图 4.41　使用 jQuery 遍历指针方法

children()方法返回的是一个集合，因此既可以使用 jQuery 的方法进行筛选，也可以使用数组方法直接访问其中的元素。如果使用数组方法访问时，就需要使用$()方法把访问得到的元素对象转换为 jQuery 对象，否则无法调用 jQuery 的方法。

4.12.2　JavaScript 实现的元素遍历方法

DOM 预定义的节点指针属性是不区分类型的，为了方便 DOM 元素遍历操作需要，可以使用 JavaScript 自定义，说明如下。

1. 自定义 getChildren()方法

由于 children 是 DOM 默认的关键字，因此这里自定义的 getChildren()方法，模仿 jQuery 中的 children()方法，实现代码如下。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("getChildren",function(){     //获取元素节点的子元素集合
    var _this = this;
    var a = _this.childNodes;   //获取当前元素的所有子节点
    var b = [];
    for(var i=0; i<a.length; i++){  //遍历子节点，筛选出所有元素节点
        if(a[i].nodeType == 1) b.push(a[i]);
    }
    return b;       //返回存储子元素的集合
})
window.onload = function(){
    var ul = document.getElementsByTagName("ul")[0];
    var o = ul.getChildren();   //调用 getChildren()方法，获取所有子元素集合
    alert(o.length);   //返回 3
}
</script>

<ul>
```

```
    <li>D 是 Document 一词的简写，表示文档，它是 DOM 的表现层。</li>
    <li>O 是 Object 一词的简写，表示对象，它是 DOM 的逻辑层。</li>
    <li>M 是 Model 一词的简写，表示模型，它是 DOM 的交互层。</li>
</ul>
```

2. 自定义 next()方法

同理我们可以通过节点类型来筛选出下一个元素节点，从而使用 JavaScript 自定义 next()方法，实现代码如下。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("next",function(){    //获取相邻的下一个元素节点
    var _this = this.nextSibling;
    while (_this && _this.nodeType != 1){//如果节点类型不等于 1，说明它不是元素节点，继续遍历
        _this = _this.nextSibling;
    }
    return _this;
})
window.onload = function(){
    var h1= document.getElementsByTagName("h1")[0];
    var o = h1.next(); //调用 next()方法，获取下一个相邻元素
    alert(o.nodeName);      //返回 P
}
</script>

<h1>一级标题</h1>
<p>段落文本</p>
```

3. 自定义 prev()方法

prev()方法与 next()方法访问方向正好相反，可以筛选出上一个元素节点，自定义 prev()方法的代码如下。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("prev",function(){    //获取相邻的上一个元素节点
    var _this = this.previousSibling;
    while (_this && _this.nodeType != 1){//如果节点类型不等于 1，说明它不是元素节点，继续遍历
        _this = _this.previousSibling;
    }
    return _this;
})
window.onload = function(){
    var h1= document.getElementsByTagName("h1")[0];
    var o = h1.next(); //调用 next()方法，获取上一个相邻元素
    alert(o.nodeName);      //返回 h1
}
</script>

<h1>一级标题</h1>
<p>段落文本</p>
```

4. 自定义 first()和 last()方法

虽然 jQuery 没有封装 DOM 的 firstChild 和 lastChild 属性，但是我们仍有必要自定义访问第一个子元素和最后一个子元素的方法。jQuery 没有定义这两个方法，是因为它提供了更多更灵活的筛选元素的方法。

例如，下面的示例自定义了 first()和 last()方法，然后调用它们，并修改第一个子元素和最后一个子元素包裹的内容，演示效果如图 4.42 所示。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("first",function(){  //获取第一个子元素节点
    var _this = this.firstChild;
    while (_this && _this.nodeType != 1){    //如果第一个节点等于 1，说明它不是元素节点
        _this = _this.nextSibling;   //则向下遍历节点
    }
    return _this;
})
DOMextend("last",function(){   //获取最后一个子元素节点
    var _this = this.lastChild;
    while (_this && _this.nodeType != 1){    //如果最后一个节点等于 1，说明它不是元素节点
        _this = _this.previousSibling;   //则向上遍历节点
    }
    return _this;
})
window.onload = function(){
    var ul= document.getElementsByTagName("ul")[0];
    ul.first().innerHTML = "第一个子元素"; //调用 first()方法，为第一个子元素插入 HTML 文本
    ul.last().innerHTML = "最后一个子元素"; //调用 lastt()方法，为最后一个子元素插入 HTML 文本
}
</script>

<ul>
    <li>D 是 Document 一词的简写，表示文档，它是 DOM 的表现层。</li>
    <li>O 是 Object 一词的简写，表示对象，它是 DOM 的逻辑层。</li>
    <li>M 是 Model 一词的简写，表示模型，它是 DOM 的交互层。</li>
</ul>
```

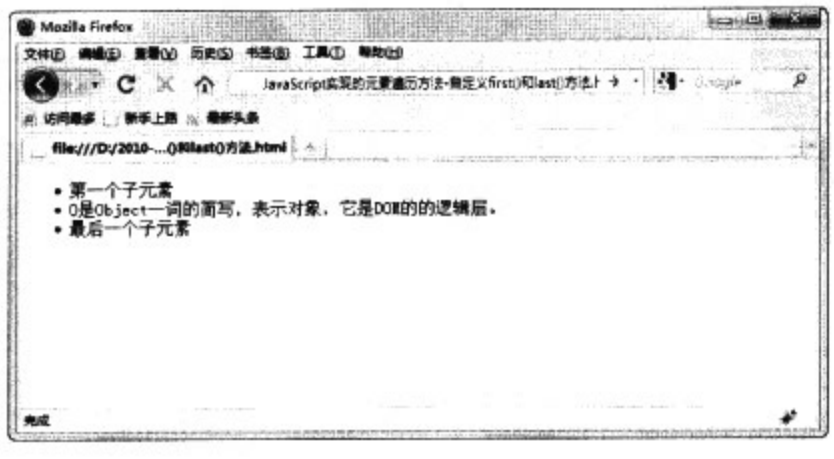


图 4.42　访问元素的第一个和最后一个子元素

第5章 事件封装机制与解析

在自然状态下，JavaScript 脚本在浏览器加载 DOM 文档结构的过程中会自动执行，如同流水一样顺势而下、一泻万里。但是为了实现用户与 Web 的互动，设计者并不希望所有 JavaScript 脚本在加载的过程中就被执行，而是希望设置一个恰当时机，由用户自己来决定 JavaScript 脚本何时执行。于是就产生了触发脚本执行的事件，而事件又可能包括必然事件和偶然事件。页面加载完毕和页面将要关闭，可以看作是必然要发生的事件，而单击按钮、输入文本就属于偶然事件了。偶然事件有可能发生，也有可能不发生，这个就看用户的需要了。

JavaScript 内置了一组与用户进行交互的事件，以及把事件与要处理的 JavaScript 脚本进行绑定、管理和操作的方法。jQuery 增加并扩展了 JavaScript 的事件处理机制，它不仅提供了更加优雅的事件处理语法，而且极大地增强了事件处理能力。

5.1 事件模型

事件模型(即所谓的事件驱动模式或方式)是面向对象程序设计的一个核心概念，它以消息为基础，以事件来驱动(Message based，Event driven)。当文档发生了特定事情时，浏览器会自动生成一个事件对象(Event)，以响应该事件，并执行对应的 JavaScript 脚本。

在事件处理标准化之前，Netscape 公司率先在 Navigator 浏览器中引入事件模型，后来的各种事件模型都是在此基础进化演变而来的。

5.1.1 0 级事件模型

由于是在 Navigator 浏览器中最早引入事件驱动模型的，因此人们习惯上称之为 0 级事件模型。所有现代浏览器都支持这种模型，它是应用最广泛、影响最深远的事件处理模型。由于这种模型历史渊源，基本上所有的浏览器都支持它。

在 0 级事件模型中，事件处理程序先被定义为函数实例，然后绑定到 DOM 元素的事件属性上面，从而实现事件注册。例如，绑定函数到 onclick 属性上，用来处理 click(鼠标单击)事件；绑定函数到 onmouseover 属性上，用来处理 mouseover(鼠标移过)事件等。例如，在下面的示例中先定义了一个匿名函数，然后把它赋值给按钮的 onclick 属性，实现注册事件。

```
<script type="text/javascript" >
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0];
    btn.onclick = function(){   //绑定事件处理函数，为按钮注册鼠标单击事件
        alert(this.nodeName);
    }
}
</script>

<input type="button" value="0 级事件模型" />
```

0 级事件模型允许把特定的事件处理函数的函数体直接赋值给 DOM 元素的 HTML 特性，从而简化了事件注册的程序。例如，针对上面的示例可以简写为下面一行代码。

```
<input type="button"  onclick="alert(this.nodeName);" value="0 级事件模型" />
```

5.1.2 事件模型中的 Event 对象

浏览器在事件被触发时，会自动创建一个 Event 对象，Event 对象实际上是 Event 类型实例，在默认状态下它会被作为参数传递给了事件处理函数。不过 IE 浏览器把 Event 视为独立

的对象，通过 Window 对象的 event 属性进行访问。为了能够兼容 IE 与非 IE 浏览器共享 Event 对象，我们可以使用如下代码进行兼容。

```
var event = event || window.event; //兼容不同类型浏览器的访问 Event 对象方法
```

Event 对象包含了属性和方法，利用这些属性可以动态存储与当前事件相关的信息，如响应事件的类型、按下的鼠标键、按下的键盘键和光标指针的位置等。下面的示例演示了 Event 对象在事件处理中扮演的角色。如果单击按钮，则会在按钮上提示当前单击的事件类型；如果双击，则会显示双击事件类型，如图 5.1 所示。

```
<script type="text/javascript" >
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0];
    btn.onclick = function(event){  //绑定鼠标单击事件处理函数
        var event = event || window.event;    //兼容不同类型浏览器
        btn.value = event.type; //获取当前事件的类型
    }
    btn.ondblclick = function(event){    //绑定鼠标双击事件处理函数
        var event = event || window.event;    //兼容不同类型浏览器
        btn.value = event.type; //获取当前事件的类型
    }
}
</script>

<input type="button" value="Event 对象" />
```

图 5.1 Event 对象在事件中的作用

注意：IE 和 DOM 标准浏览器虽然都使用同名的 Event 来存储事件的动态信息，但是它们的属性和方法存在差异，即使是同名的属性，返回的属性值也可能不同。例如，IE 使用 srcElement 属性记录当前事件作用的元素，而 DOM 标准使用 target 属性记录当前事件作用的元素。IE 和 DOM 标准都支持 button 属性，但是 IE 的 button 属性返回值为 1、2、4，而 DOM 标准的 button 属性返回值为 0、2、1，这些值分别表示鼠标的左、右和中键。关于这个技术话题请参阅 JavaScript 的基础类教程。

5.1.3 事件模型中的冒泡现象

在 DOM 文档结构树中，每个元素并非都孤立的存在，上下级元素之间必然存在嵌套关系，这就给 Web 的事件处理带来很多麻烦。当触发 DOM 树上某个元素的事件时，浏览器的事件处理机制会检查在那个元素上是否已经建立了特定的事件处理程序。如果是，则调用该事件处理程序，但是事情远远没有结束。

在目标元素获取机会处理事件之后，事件模型会检查目标元素的父元素，检测父元素是否也注册了同类型的事件，如果是，还要调用父元素的事件处理程序。在这之后，事件模型还会继续检测上一级元素，以此逐层检索，持续不停直到 DOM 树的顶部。这个事件检测的过程如同水中向上传播的气泡，故有人形象地把这个事件处理过程称之为事件冒泡，如图 5.2 所示。

图 5.2 事件冒泡示意图

例如，在下面这个示例中分别为层层嵌套的多个 div 元素注册 mouseover 和 mouseout 事件。在 mouseover 的事件处理程序中动态设置当前 div 元素的边框颜色为红色，在 mouseout 的事件处理程序中动态设置当前 div 元素的边框颜色为白色。这样当鼠标在页面中移动时，可以看到 div 元素的边框呈波浪形变色或者显隐，演示效果如图 5.3 所示。

```
<style type="text/css">
div {
    margin:12px 10px;
    border:solid 2px blue;
}
</style>
<script type="text/javascript" >
window.onload = function(){
    var div = document.getElementsByTagName('div');   //获取所有 div 元素
    for (var i = 0; i < div.length; ++i){    //遍历 div 元素
        div[i].onmouseover = (function(i){    //依次为每个 div 元素注册鼠标经过事件
            return function(){   //以闭包形式存储动态变量 i 的值，以便定位 div
                div[i].style.borderColor = 'red';  //定义边框的颜色样式为红色
            }
        })(i);      //向闭包内传递变量 i 的值
    }
```

```
    for (var i = 0; i < div.length; ++i){     //遍历 div 元素
        div[i].onmouseout = (function(i){     //依次为每个 div 元素注册鼠标移出事件
            return function(){    //以闭包形式存储动态变量 i 的值，以便定位 div
                div[i].style.borderColor = 'white';     //定义边框的颜色样式为白色
            }
        })(i);          //向闭包内传递变量 i 的值
    }
}
</script>

<div>
    <div>
        <div>
            <div>
                <div>冒泡型事件</div>
            </div>
        </div>
    </div>
</div>
```

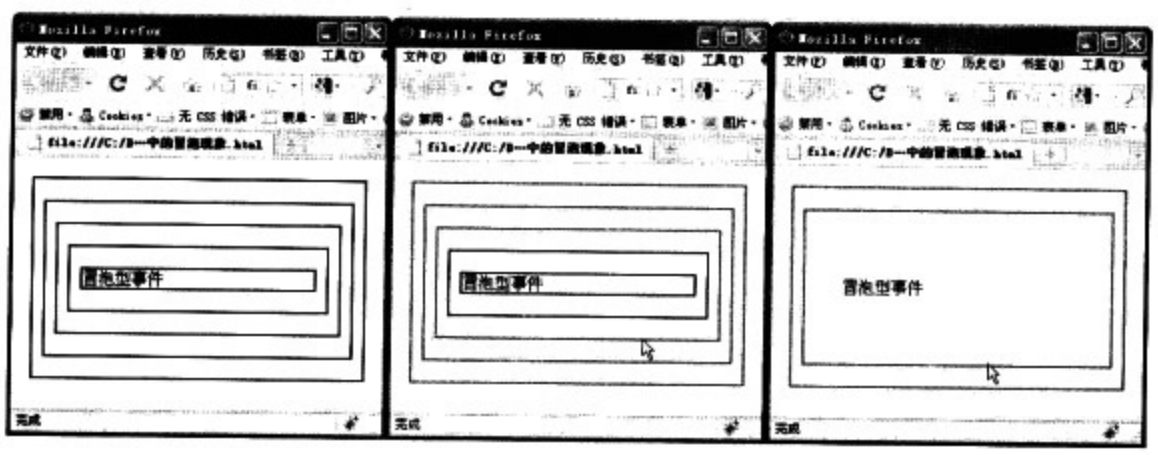


图 5.3 事件冒泡动画演示效果

注意：实际上事件冒泡仅是事件流的一种类型，另外还有捕获型事件流，演示示意图如图 5.4 所示。从 DOM 标准角度分析，事件流一般可以分为三个阶段：捕捉阶段、目标阶段和冒泡阶段。标准事件流一般都会包括这三个阶段，演示示意图如图 5.5 所示。

图 5.4 捕获型事件流示意图

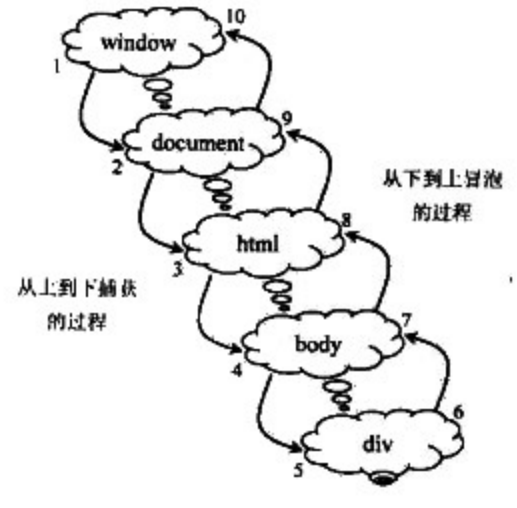


图 5.5 标准事件流示意图

不同类型的浏览器在响应事件时，因为事件流的类型不同，在复杂程序设计中往往会呈现不同的设计效果。同时，对于不同类型浏览器及其版本，事件流所能影响的 DOM 树顶不同，例如：

- IE 5.5 及其以下版本： div → body → document
- IE 6.0 及其以上版本： div → body → html → document
- Mozilla 1.0 及其以上版本： div → body → html → document → window

5.1.4 事件流控制与默认事件动作

不管是什么类型的浏览器，都会为 Event 对象预定操纵事件流的方法。例如，在 DOM 标准事件模型中，使用 Event 对象的 stopPropagation()方法可以中止事件继续向上级层次传播；而对于 IE 浏览器来说，则可以使用 Event 对象的 cancelBubble 属性来阻止，只要设置该属性值为 true 即可。有关事件流的更详细控制操作将在后面章节中进行详细讲解。

一些事件往往被预设了特定的动作，也就是说它包含特定的语义。例如，当单击 a 元素时会自动进行导航；而单击提交按钮时，会自动提交表单；单击重设按钮时，会自动清除表单域的输入值等。如果要控制这些特定的动作，我们可以在这些事件处理函数中返回 false 即可，从而取消事件的默认动作。例如，在下面的示例中，当单击超链接之后，超链接默认的动作将失效，仅弹出一个提示对话框，显示超链接地址。

```
<script type="text/javascript" >
window.onload = function(){
    var a = document.getElementsByTagName("a")[0];
    a.onclick = function(){
        alert(a.getAttribute("href"));
        return false;
    }
}
</script>

<a href="http://www.baidu.com/">百度一下</a>
```

5.1.5 2 级 DOM 标准事件模型

0 级事件模型使用起来虽然很方便，但是存在致命的缺陷：元素属性被用来存储事件处理函数的引用。所以每个元素对于任何特定事件类型，每次只能够注册一个事件处理程序。如果独立管理事件流会显得非常笨拙，甚至无能为力。

W3C 在 DOM 2.0 版本中正式制订了标准化的事件处理模型，并于 2000 年 11 月实行。2 级事件模型得到了所有现代主流浏览器的主持。不过 IE 浏览器还不能够完全支持，仅能支持

2 级事件模型中的基本模块功能。

1. 注册事件

2 级事件模型为 DOM 中的 Element 类型元素定义了 addEventListener()方法，使用这个方法注册事件，而放弃使用 0 级事件模型中为元素的事件属性指定事件处理函数的方法。

addEventListener()方法的格式如下。

```
addEventListener(type, function, useCapture);
```

在该方法中，第一个参数表示要绑定的事件类型。事件类型与事件属性不同，它没有前缀 on。例如，对于事件属性 onclick 来说，对应事件类型为 click。

第二个参数表示调用的事件处理函数，该函数自带一个默认参数引用 Event 对象，以方便传递事件发生时操作相关的信息。

第三个参数为一个布尔值。如果为 true，则在事件传播的捕捉阶段触发响应；如果该参数值为 false，则在事件传播的冒泡阶段触发响应。

例如，在下面的示例中为按钮注册一个鼠标单击事件类型，然后在事件处理函数中获取当前事件的 Event 对象，并显示当前事件的类型。

```
<script type="text/javascript" >
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0];  //获取按钮元素
    btn.addEventListener("click", function(event){    //注册鼠标单击事件，并设置事件流为捕获型事件
        var event = event || window.event;    //兼容 Event 对象
        btn.value = event.type; //显示当前事件的类型
    },true);
}
</script>

<input type="button" value="Event 对象" />
```

2. 事件传播

在 2 级事件模型中，一旦事件被触发，事件流首先从 DOM 树顶部(文档节点)向下传播，直到目标节点，然后再从目标节点向上传播到 DOM 树顶。从上到下的过程被称为捕获阶段，从下到上的过程被称为冒泡阶段。

addEventListener()方法的第三个参数可以设置事件响应的阶段。如果参数值为 true，则事件只在捕获阶段被触发，可以标志为捕获型处理程序；如果参数值为 false，则事件只在冒泡阶段被触发，可以标志为冒泡型处理程序。

例如，在下面的示例中利用循环体结构分别为按钮元素及其所有父级节点注册一个捕获型鼠标单击类事件处理函数。在浏览器中预览，当单击按钮时，可以看到事件触发的先后过程：首先触发的是#document 节点，然后是 HTML 节点、BODY 节点，最后才是 INPUT 节点，如图 5.6 所示。

```
<script type="text/javascript" >
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0];  //获取按钮
    var p = document.getElementsByTagName("p")[0];    //选择第一个 p 元素
    var i = 1;     //声明并初始化一个临时变量
    do{   //使用 do 循环结构逐层注册鼠标单击事件
        btn.addEventListener("click", function(){ //注册鼠标单击事件
            p.innerHTML += "<br />(" + i++ + ")  " + this.nodeName;
    },true);  //动态跟踪当前响应节点的名称
        btn = btn.parentNode;    //访问上一级父元素
    } while(btn); //设置循环条件：如果存在父节点
}
</script>

<input type="button" value="Event 对象" />
<p>捕获型事件流传播过程：</p>
```

下面修改 addEventListener()方法的第三个参数，设置参数值为 false，即注册事件为冒泡型处理程序。然后在浏览器中预览，当单击按钮时，可以看到事件触发的先后过程：首先触发的是 INPUT 节点，然后是 BODY 节点、HTML 节点，最后才是#document 节点，如图 5.7 所示。

```
<script type="text/javascript" >
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0];  //获取按钮
    var p = document.getElementsByTagName("p")[0];        //选择第一个 p 元素
    var i = 1;     //声明并初始化一个临时变量
    do{   //使用 do 循环结构逐层注册鼠标单击事件
        btn.addEventListener("click", function(){ //注册鼠标单击事件
            p.innerHTML += "<br />(" + i++ + ")  " + this.nodeName;
    },false);      //动态跟踪当前响应节点的名称
        btn = btn.parentNode;    //访问上一级父元素
    } while(btn); //设置循环条件：如果存在父节点
}
</script>

<input type="button" value="Event 对象" />
<p>冒泡型事件流传播过程：</p>
```

图 5.6 捕获型事件流传播过程

图 5.7 冒泡型事件流传播过程

3. 销毁事件

2 级事件模型还为 DOM 中的 Element 类型元素定义了 removeEventListener()方法，通过这个方法可以随时销毁已经注册的事件，以节省系统资源。该方法的用法与 addEventListener()方法相同。例如，针对上面的示例，可以在事件处理函数中为当前对象添加 removeEventListener()方法，销毁刚刚注册的鼠标单击事件，避免事件处理函数被多次触发。其中 arguments.callee 引用当前匿名的事件处理函数。

```
<script type="text/javascript" >
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0];  //获取按钮
    var p = document.getElementsByTagName("p")[0];         //选择第一个 p 元素
    var i = 1;     //声明并初始化一个临时变量
    do{   //使用 do 循环结构逐层注册鼠标单击事件
        btn.addEventListener("click", function(){ //注册鼠标单击事件
            p.innerHTML += "<br />(" + i++ + ")  " + this.nodeName;
    },false);      //动态跟踪当前响应节点的名称
            this.removeEventListener("click",arguments.callee,false);    //注销当前鼠标
单击事件
        btn = btn.parentNode;    //访问上一级父元素
    } while(btn); //设置循环条件：如果存在父节点
}
</script>

<input type="button" value="Event 对象" />
<p>冒泡型事件流传播过程：</p>
```

注意：removeEventListener()方法的第三个参数值与 addEventListener()方法的第三个参数值保持一致。也就是说，如果使用 addEventListener()方法将事件处理函数绑定到捕获阶段，则必须在 removeEventListener()方法中指明捕获阶段，只有这样才能够正确地删除事件处理函数。如果绑定在冒泡阶段的事件处理函数，而在捕获阶段来删除它，虽然这样做不会引发错误，但是销毁操作是无效的。

5.1.6 IE 事件模型

IE 7 及其以下版本浏览器都不支持 DOM 标准的 2 级事件模型，而在 IE 8 版本中才开始支持，不过 IE 浏览器从早期版本开始就提供了一套与 2 级事件模型非常相似的模型，虽然用法略有区别，但功能基本相同。

1. 注册事件

IE 浏览器为 DOM 中的 Element 类型元素定义了 attachEvent()方法，通过这个方法可以注册事件，该方法包含两个参数，语法格式如下。

```
attachEvent(type, function);
```

其中第一个参数用来设置元素的事件属性，如 onclick，而不是 2 级事件模型中的 click 事件类型；第二个参数与 addEventListener()方法中的第二个参数相同，都是指事件处理函数。

如果完全兼容 IE 与非 IE 浏览器，则在注册事件时可以借助 if 条件语句分别进行注册。例如，针对上一节中注册事件的示例，可以按如下方法进行修改，以实现在不同浏览器下都可以正常注册。

```
<script type="text/javascript" >
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0];
    var p = document.getElementsByTagName("p")[0];
    var i = 1;
    do{
        if(btn.addEventListener)     //如果支持 addEventListener()方法，则调用该方法
            btn.addEventListener("click", function(){
                p.innerHTML += "<br />(" + i++ + ") " + this.nodeName;
            },false);
        else{ //否则，调用 attachEvent()方法在 IE 下注册事件
            btn.attachEvent("onclick", (function(btn){
                return function(){    //返回闭包函数，从而动态锁定响应事件的当前对象
                    p.innerHTML += "<br />(" + i++ + ") " + btn.nodeName;
                }
            })(btn));    //直接调用函数，以便向内部传递当前绑定的元素对象
        }
        btn = btn.parentNode;
    } while(btn);
}
</script>

<input type="button" value="Event 对象" />
<p>事件流传播过程：</p>
```

当使用 attachEvent()方法注册事件时，在事件处理函数中的 this 指针总是指向 Window

对象；但是在 0 级事件模型中，事件处理函数中的 this 指针总是指向当前注册事件的对象；而在 DOM 2 级事件模型中，addEventListener()方法中的 this 都会指向触发当前事件的对象。为了解决 attachEvent()方法无法定位当前事件对象的问题，可以通过返回闭包函数的形式，在 attachEvent()方法中的第二个参数中直接调用函数，而不是引用函数，然后通过返回闭包函数以实现传递事件处理函数，这样在调用函数时，把当前事件对象传递给闭包函数，从而动态锁定事件对象。本例分别在 Firefox 和 IE 浏览器下预览，则效果如图 5.8 和图 5.9 所示，执行效果完全相同。

图 5.8 在 Firefox 浏览器下的浏览效果

图 5.9 在 IE 浏览器下的浏览效果

2. 事件传播

IE 事件模型不支持事件流中的捕获阶段，但是支持冒泡型事件传播方式。也就是说，在 IE 浏览器中，事件流总是从目标对象向上逐层传递到 DOM 树顶。在 2 级事件模型中，事件冒泡只适用于原始事件和输入类事件，如鼠标事件和键盘事件，不支持高级语义事件。IE 的事件冒泡传播与 2 级事件模型是相同的，但是它们中止冒泡的方式不同。

2 级事件模型通过 stopPropagegation()方法可以中止事件流的冒泡，而 IE 事件模型设计通过 Event 对象的 cancelBubble 属性来控制，如果要中止事件流冒泡，则可以如下定义。

```
window.event.cancelBubble = true;
```

注意： cancelBubble 属性只适用于当前事件，当新事件发生时，它将被赋予给新的 Event 对象，此时 cancelBubble 属性值被还原为默认值 false。

3. 注销事件

IE 浏览器支持使用 detachEvent()方法注销事件，该方法也包含有两个参数：第一个参数为事件属性名(如 onclick)，第二个参数为事件处理函数。例如，针对上面的示例，设计当按钮被单击一次之后，使用 detachEvent()方法中止当前对象绑定的鼠标单击事件。这样按钮就只能够被单击一次，单击一次之后自动失效，代码如下。

```
<script type="text/javascript" >
window.onload = function(){
    var btn = document.getElementsByTagName("input")[0];
    var p = document.getElementsByTagName("p")[0];
    var i = 1;
    do{
        if(btn.addEventListener)      //如果支持 addEventListener()方法，则调用该方法
            btn.addEventListener("click", function(){
                p.innerHTML += "<br />(" + i++ + ") " + this.nodeName;
                this.removeEventListener("click",arguments.callee,false);
            },false);
        else{ //否则，调用 attachEvent()方法在 IE 下注册事件
            btn.attachEvent("onclick", (function(btn){
                return function(){     //返回闭包函数，从而动态锁定响应事件的当前对象
                    p.innerHTML += "<br />(" + i++ + ") " + btn.nodeName;
                    btn.detachEvent("onclick", arguments.callee);     //注销 IE 事件
                }
            })(btn));    //直接调用函数，以便向内部传递当前绑定的元素对象
        }
        btn = btn.parentNode;
    } while(btn);
}
</script>

<input type="button" value="Event 对象" />
<p>事件流传播过程：</p>
```

5.2 jQuery 事件模型

为了能够更好地兼容不同类型的浏览器，jQuery 在 JavaScript 的基础上，进一步封装了不同类型的事件模型，从而形成了一种功能更强大、用法更优雅的“jQuery 事件模型”。jQuery 事件模型体现如下特征。

- 统一了事件处理中的各种方法。
- 允许在每个元素上为每个事件类型建立多个处理程序。
- 采用 2 级事件模型中标准的事件类型名称。
- 统一了 Event 对象的传递方法，并对 Event 对象的常用属性和方法进行规范。
- 为事件管理和操作提供统一的方法。

考虑到 IE 浏览器不支持事件流中的捕获型阶段，且开发者很少使用此阶段，所以 jQuery 事件模型也没有支持事件流中的捕获型阶段。除了这一点区别外，jQuery 事件模型的功能与 2 级事件模型基本相似。

5.2.1 绑定事件

绑定事件有以下几种方法。

1. 使用 bind()方法绑定

jQuery 定义了 bind()方法作为统一的接口，用来为每一个匹配元素绑定事件处理程序。其基本语法如下。

```
bind(type, [data], fn)
```

其中参数 type 表示事件类型，与 2 级事件模型中的 addEventListener()方法的第一个参数相同；参数 fn 表示事件处理函数；参数 data 比较特殊，是可选参数，它可以作为 event.data 属性值传递给事件对象的额外数据对象。例如，下面的示例分别为文档中的 p 元素绑定单击事件，这样当单击段落文本时会提示显示当前段落文本。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").bind("click",function(){
        alert($(this).text());
    });
})
</script>

<p>段落 1</p>
<p>段落 2</p>
<p>段落 3</p>
```

如果希望向事件处理函数传递更多的信息，则可以把这些信息封装在一个对象结构中，然后把这个对象作为 bind()方法的第二个参数，从而实现事件外与事件内之间进行数据通信。例如，在上面示例的基础上向其传递两个值"A"和"B"，则先使用对象结构将这两个值进行封装，然后作为参数传递给 bind()方法。在事件处理函数中可以通过 Event 对象的 data 属性来访问这个对象，进而访问该对象内包含的数据，代码如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").bind("click",{a:"A",b:"B"},function(event){
        $(this).text(event.data.a + event.data.b);
    });
})
</script>

<p>段落 1</p>
<p>段落 2</p>
```

```
<p>段落 3</p>
```

如果既想取消元素特定事件类型默认的行为，又想阻止事件冒泡，设置事件处理函数返回值为 false 即可。示例代码如下。

```
$("p").bind("click",{a:"A",b:"B"},function(event){
    $(this).text(event.data.a + event.data.b);
    return false;
});
```

也可以通过使用 preventDefault()方法只取消默认的行为。示例代码如下。

```
$("p").bind("click",{a:"A",b:"B"},function(event){
    $(this).text(event.data.a + event.data.b);
    event.preventDefault();
});
```

还可以通过使用 stopPropagation()方法只阻止一个事件冒泡。示例代码如下。

```
$("p").bind("click",{a:"A",b:"B"},function(event){
    $(this).text(event.data.a + event.data.b);
    event.stopPropagation();
});
```

2. 使用快捷方法绑定

除了 bind()方法外，jQuery 还定义了 20 个快捷方法为特定的事件类型绑定事件处理程序，这些方法与 2 级事件模型中的事件类型一一对应，名称完全相同，如表 5.1 所示。

表 5.1 绑定特定事件类型的方法

blur()	focus()	mousedown()	resize()
change()	keydown()	mousemove()	scroll()
click()	keypress()	mouseout()	select()
dblclick()	keyup()	mouseover()	submit()
error()	load()	mouseup()	unload()

例如，对于下面使用 bind()方法绑定的事件：

```
$("p").bind("click",function(){
    alert($(this).text());
});
```

可以直接使用 click()方法绑定：

```
$("p").click(function(){
    alert($(this).text());
});
```

注意：当使用这些快捷方法时，无法向 event.data 属性传递额外的数据。如果不为这些方法传递事件处理函数而直接调用它们，则会触发已绑定在这些对象上的对应事件，包括默认的动作。

3. 使用 one()绑定

one()方法是 bind()方法的一个特例，由它绑定的事件在执行一次响应之后就会失效。它的用法与 bind()完全相同。例如，对于下面使用 one()方法绑定的鼠标单击事件，它只能够响应一次，当第二次单击段落文本时就不再响应。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").one("click",function(){
        alert($(this).text());
    });
})
</script>

<p>段落 1</p>
<p>段落 2</p>
<p>段落 3</p>
```

这个方法的设计思路是在事件处理函数内部增加了注销当前事件的代码，读者可以参阅第 5.1.6 节中注销事件的示例。

5.2.2 注销事件

交互型事件的生命周期往往与页面的生命周期是相同的，但是很多交互事件只是在特定的时期或者条件下有效，超过了时效期，就应该把它注销掉，以节省系统空间。one()方法在执行一次任务之后，能够自动删除事件，而 bind()方法和其他快捷方法需要手工清除。

jQuery 定义了 unbind()方法，该方法与 bind()方法是反向操作，能够从每一个匹配的元素中删除绑定的事件。如果没有指定参数，则删除所有绑定的事件，包括使用 bind()方法注册的自定义事件。例如，在下面的示例中，分别为 p 元素绑定 click、mouseover、mouseout 和 dblclick 事件类型，在 dblclick 事件类型的事件处理函数中调用 unbind()。这样在没有双击段落文本之前，鼠标移过、移出和单击都会触发响应，一旦双击段落文本，则所有类型的事件都被注销，鼠标移过、移出和单击动作就不再效应。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
```

```
    $("p").dblclick(function(){ //注册双击事件
        $("p").unbind();    //注册注销所有事件
    });
    $("p").click(f);   //注册单击事件
    $("p").mouseover(f);    //注册鼠标移过事件
    $("p").mouseout(f);     //注册鼠标移出事件
    function f(event){      //事件处理函数
        this.innerHTML  = "事件类型 = " + event.type;
    }
})
</script>

<p>百变文本</p>
```

如果提供了事件类型作为参数，则只删除该类型的绑定事件。例如，下面的代码将只注销 mouseover 事件类型，而其他类型的事件依然有效。

```
$("p").dblclick(function(){
    $("p").unbind("mouseover");
});
```

如果把在绑定时传递的处理函数作为第二个参数，则只有这个特定的事件处理函数会被删除。例如，在下面的示例中分别为 p 注册鼠标移过事件，并绑定两个事件处理函数，这样当鼠标经过段落文本时，会分别调用这两个事件处理函数。但是单击段落文本时，将移出其中一个事件处理函数，则再次移过段落文本时，将只有一个事件处理函数被调用。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").click(function(){    //注册单击事件
        $("p").unbind("mouseover", e);    //注销鼠标移过事件中 e()事件处理函数
    });
    $("p").mouseover(f);    //注册鼠标移过事件，绑定 f()事件处理函数
    $("p").mouseover(e);    //注册鼠标移过事件，绑定 e()事件处理函数
    function f(){
        $(this).text("第一个单击事件")
    }
    function e(){
        $(this).text("第二个单击事件")
    }
})
</script>

<p>段落文本</p>
```

5.2.3 jQuery 事件模型中的 Event 对象

当使用 bind()、one()或者其他快捷方法注册事件时，Event 对象实例将作为第一个参数

传递给事件处理函数，这与 2 级事件模型是完全相同的，但是 jQuery 统一了 IE 事件模型和 2 级事件模型中 Event 对象属性和方法的用法，使其完全符合 DOM 标准事件模型的规范。除统一了 Event 对象的生僻属性外，jQuery 还为我们修正了在 Web 开发中可能遇到的浏览器兼容性问题，使我们不再为了浏览器兼容问题而烦恼。如表 5.2 所示是 jQuery 的 Event 对象可以使用的属性和方法。

表 5.2 jQuery 的 Event 对象属性和方法

属性/方法	说 明
type	获取事件的类型，如 click、mouseover 等。返回值为事件类型的名称，该名称与注册事件处理函数时使用的名称相同
target	发生事件的节点。一般利用该属性来获取当前被激活事件的具体对象
relatedTarget	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将要进入的那个节点
altKey	表示在声明鼠标事件时，是否按下了【Alt】键。如果返回值为 true，则表示按下
ctrlKey	表示在声明鼠标事件时，是否按下了【Ctrl】键。如果返回值为 true，则表示按下
shiftKey	表示在声明鼠标事件时，是否按下了【Shift】键。如果返回值为 true，则表示按下
metaKey	表示在声明鼠标事件时，是否按下了【Meta】键。如果返回值为 true，则表示按下
which	当在声明 mousedown、mouseup 和 click 事件时，显示鼠标键的状态值，也就是说哪个鼠标键改变了状态。返回值为 1，表示按下左键；返回值为 2，表示按下中键；返回值为 3，表示按下右键
which	当在声明 keydown 和 keypress 事件时，显示触发事件的键盘键的数字编码
pageX	对于鼠标事件来说，指定光标指针相对于页面原点的水平坐标
pageY	对于鼠标事件来说，指定光标指针相对于页面原点的垂直坐标
screenX	对于鼠标事件来说，指定光标指针相对于屏幕原点的水平坐标
screenY	对于鼠标事件来说，指定光标指针相对于屏幕原点的垂直坐标
data	存储事件处理函数第二个参数所传递的额外数据
preventDefault()	取消可能引起任何语义操作的事件，如元素特定事件类型的默认动作
stopPropagation()	防止事件沿着 DOM 树向上传播

5.2.4 jQuery 事件触发

事件都在特定条件下发生，自然不同类型的事件触发的时机是无法预测的。你无法知道用户何时单击按钮提交表单，或者何时输入文本。但是在很多情况下，开发人员需要在脚本中控制事件触发的时机。例如，设计一个弹出广告，虽然广告画面提供了允许用户关闭广告的按钮，但是我们也应该设计一个条件，控制广告在显示 3 秒钟之后自动关闭。

也许读者可以把事件处理函数定义为独立的窗口函数(顶层函数)，以便直接通过名称调

用它，而不需要特定的事件交互。但是如果允许直接调用事件的处理函数，则会简化程序的设计，更为重要的是可以方便操作。在传统表单设计中，表单域元素都拥有 focus()和 blur()方法，调用它们将会直接调用对应的 focus 和 blur 事件处理函数，使文本域获取焦点或者失去焦点。

jQuery 定义在脚本控制下自动触发事件处理函数的一系列方法，其中最常用的是 trigger()方法。语法如下。

```
trigger(type, [data])
```

其中第一个参数 type 表示事件类型，以字符串形式传递；第二个参数 data 是可选参数，利用该参数可以向调用的事件处理函数传递额外的数据。例如，在下面的示例中，本应该在用户单击时才能够触发的事件处理程序，现在利用 trigger()方法，定义鼠标移过事件处理函数，从而当鼠标移过段落文本时，会自动触发鼠标单击事件。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").click(function(){
        $("p").text("鼠标单击事件");
    });
    $("p").mouseover(function(){
        $("p").trigger("click");      //调用 trigger()方法直接触发 click 事件
    });
})
</script>

<p>段落文本</p>
```

trigger()方法也会触发同名的浏览器默认行为。例如，如果用 trigger()触发一个 submit' 事件类型，则同样会导致浏览器提交表单。如果要阻止这种默认行为，则可以在事件处理函数中设置返回值为 false。

所有触发的事件都会冒泡到 DOM 树顶。例如，如果在 p 元素上触发一个事件，它首先会在这个元素上触发，然后向上冒泡，直到触发 Document 对象为止。这个事件对象有一个 target 属性指向最开始触发这个事件的元素。读者可以用 stopPropagation()方法来阻止事件冒泡，或者在事件处理函数中返回 false 即可。

triggerHandler()方法对 trigger()方法进行补充，该方法的行为表现与 trigger()方法类似，用法也相同，但是存在以下三个主要区别。

- triggerHandler()方法不会触发浏览器默认事件。
- triggerHandler()方法只触发 jQuery 对象集合中第一个元素的事件处理函数。
- triggerHandler()方法返回的是事件处理函数的返回值，而不是 jQuery 对象。如果最

开始的 jQuery 对象集合为空，则这个方法返回 undefined。

除了 trigger()和 triggerHandler()方法外，jQuery 还为大部分事件类型提供了快捷触发的方法，如表 5.3 所示。

表 5.3 jQuery 定义的快捷触发事件的方法

blur()	dblclick()	keydown()	select()
change()	error()	keypress()	submit()
click()	focus()	keyup()	

这些方法没有参数，直接引用能够自动触发引用元素绑定的对应事件处理程序。例如，针对上面的示例，也可以直接使用 click()方法替代 trigger("click")方法。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").click(function(){
        $("p").text("鼠标单击事件");
    });
    $("p").mouseover(function(){
        $("p").click();//调用 click()方法快速触发 click 事件
    });
})
</script>

<p>段落文本</p>
```

5.2.5 jQuery 事件切换

jQuery 定义了两个事件切换的合成方法：hover()和 toggle()方法。事件切换在 Web 开发中经常会用到，如样式交互、行为交互等。在第 4 章中曾经介绍 toggleClass()方法，它能够显示/隐藏指定的类样式，实现样式动态切换，而 hover()和 toggle()方法能够实现行为交互。

1. 使用 toggle()切换

toggle()方法能够为 click 事件类型绑定两个事件处理函数，并确保每次单击后依次调用不同的函数。它与直接为 click 事件绑定两个函数的功能不同。

toggle()方法可以包含多个函数参数。如果单击了一个匹配的元素，则触发指定的第一个函数，当再次单击同一元素时，则触发指定的第二个函数，如果有更多函数，则再次触发，直到最后一个。随后的每次单击都重复对这几个函数的轮番调用。例如，下面的示例为 input 元素注册了一个 toggle 合成事件，这样每次单击按钮时，将会循环轮流调用参数中指定的事件处理函数。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("input").toggle(
    function(){
        this.value = "第一次单击";
    },
    function(){
        this.value = "第二次单击";
    },
    function(){
        this.value = "第三次单击";
    })
})
</script>

<input type="button" value="持续单击可以切换事件" />
```

这个方法比 toggleClass()方法更加实用，它可以根据元素被单击的次数切换元素的启用状态。例如，切换显示元素的不透明度。对于 toggleClass()方法来说，可以使用 unbind("click")来删除它。

2. 使用 hover()切换

hover()方法可以模仿悬停事件，即鼠标移动到一个对象上面及移出这个对象的方法。这是一个自定义的方法，它为频繁使用的任务提供了一种保持在其中的状态。

hover()方法包含两个参数，其中第一个参数表示鼠标移到元素上要触发的函数，第二个参数表示鼠标移出元素要触发的函数。例如，在下面的示例中为按钮绑定 hover 合成事件，这样当鼠标移过按钮时，会触发指定的第一个函数；当鼠标移出这个元素时，会触发指定的第二个函数。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("input").hover(
    function(){
        this.value = "鼠标移过";
    },
    function(){
        this.value = "鼠标已移出";
    })
})
</script>

<input type="button" value="鼠标切换事件"  />
```

mouseout 事件存在一个很严重的错误：如果鼠标移到当前元素包含的子元素上时，将会

触发当前元素的 mouseout 和 mouseover 事件。这种错误性解释严重影响了设计人员开发各类悬停处理程序，如导航菜单。

例如，在下面的示例中为 div 元素绑定 mouseover 和 mouseout 事件处理程序，当鼠标进入 div 元素时将会触发 mouseover 事件，而当鼠标移到 span 元素上时，虽然鼠标并没有离开 div 元素，但是将会触发 mouseout 和 mouseover 事件。如果鼠标在 div 元素内部移动，就可能会不断触发 mouseout 和 mouseover 事件，产生不断闪烁的事件触发现象，演示效果如图 5.10 所示。

```
<style type="text/css">
div {
    width:300px;
    height:180px;
    background:red;
    padding:20px;
}
span {
    float:right;
    width:120px;
    height:80px;
    background:blue;
    color:white;
    font-weight:bold;
}
</style>
<script type="text/javascript" >
window.onload = function(){
    var div = document.getElementsByTagName("div")[0];
    var p = document.getElementsByTagName("p")[0];
    var span = document.getElementsByTagName("span")[0];
    if(div.addEventListener){   //兼容非 IE
        div.addEventListener("mouseover",over,false); //注册 mouseover 事件
        div.addEventListener("mouseout",out,false);   //注册 mouseout 事件
    }else{    //兼容 IE
        div.attachEvent("onmouseover",over); //注册 mouseover 事件
        div.attachEvent("onmouseout",out);   //注册 mouseout 事件
    }
    function over(event){  //事件处理函数
        var event = event || window.event;   //兼容 Event 对象
        p.innerHTML += event.type + "<br />";
    }
    function out(event){   //事件处理函数
        var event = event || window.event;   //兼容 Event 对象
        p.innerHTML += event.type + "<br />";
    }
}
</script>
```

```
<div>
    <span></span>
</div>
<p></p>
```

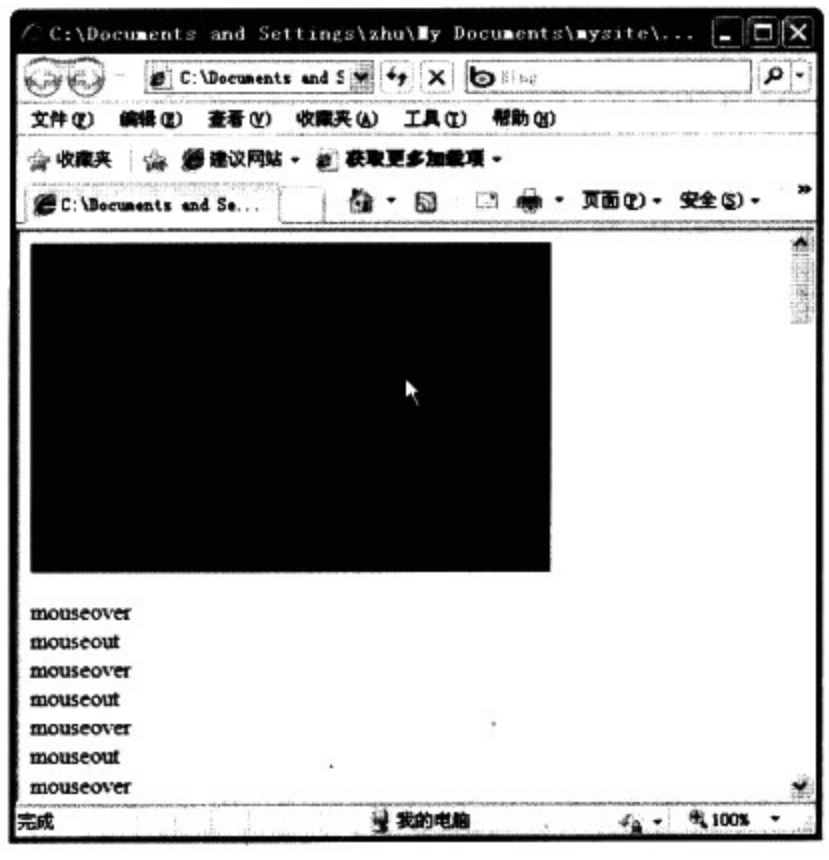


图 5.10 mouseout 事件存在错误

由于 bind()、mouseover()和 mouseout()方法都是直接在原事件基础上进行包装的，因此使用 jQuery 的 bind()、mouseover()和 mouseout()方法绑定时也会存在上述问题。而 hover()方法修正了这个错误，它会对鼠标是否仍然处在特定元素中进行检测，如果是，则会继续保持悬停状态，而不触发移出事件。例如，针对上面的示例，使用 hover()来实现相同的设计效果，当鼠标进入 div 元素，并在 div 元素内部移动时，只会触发一次 mouseover 事件，演示效果如图 5.11 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("div").hover(    //绑定 hover()合成事件
        function(event){    //注册 mouseover 事件处理函数
            $("p").append(event.type + "<br />");
        },
        function(event){    //注册 mouseout 事件处理函数
            $("p").append(event.type + "<br />");
        }
    )
})
</script>
```

图 5.11 hover()方法修正了 mouseout 事件存在的错误

5.2.6 jQuery 事件委派

在传统编程中，委派不是一个方法或者任何类型的函数。实际上，它与函数指针的定义很相似，委派是一种引用方法的类型。一旦为委派分配了方法，委派将与该方法具有完全相同的行为。委派方法的使用可以像其他任何方法一样，它具有参数和返回值。

在 jQuery 1.3 版本中新增了事件委派的方法 live()。该方法能够给所有当前以及未来将会匹配的元素绑定一个事件处理函数(如 click 事件)，也能绑定自定义事件。live()方法的用法与 bind()方法相同，第一个参数设置事件类型，第二个参数设置事件处理函数。

例如，在下面的示例中，为 p 元素委派一个 click 事件，这样在后面动态生成的 p 元素也会拥有这个 click 事件。因此，单击页面中任何一个存在的 p 元素，都会调用事先委派的事件处理函数。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").live("click", function(){     //委派事件
        $(this).after("<p>段落文本</p>");
    });
})
</script>

<p>段落文本</p>
```

但是如果使用 bind()方法为当前 p 元素绑定 click 事件，则在后面动态生成的 p 元素就不会拥有这个 click 事件，只有先前存在的 p 元素绑定了 click 事件，代码如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").bind("click", function(){     //绑定事件
        $(this).after("<p>段落文本</p>");
    });
})
</script>

<p>段落文本</p>
```

live()方法与 bind()方法还有一个区别，那就是 live()方法一次只能绑定一个事件，而 bind()方法可以绑定多个事件。

注意：jQuery 1.3 版本仅支持 click、dblclick、mousedown、mouseup、mousemove、mouseover、mouseout、keydown、keypress 和 keyup 事件类型，不支持 blur、focus、mouseenter、mouseleave、change 和 submit 等事件类型。目前 live 事件只能支持使用选择器选择的元素，如 $("li a").live(...)，但是不支持类似于$("a", someElement).live(...) 或者 $("a").parent().live(...)。

live 事件冒泡的行为与传统的方式不同，因此也不能完全支持 stopPropagation()或者 stopImmediatePropagation()阻止冒泡，但部分支持。如果内外元素都用 live 事件绑定，则可以通过 return false 来阻止冒泡。如果外部父元素是普通事件，而内部子元素是 live 事件，则无法通过 return false 来阻止冒泡。

如果要移除 live()绑定的事件，可以用 die()方法来实现。例如，在下面的示例中，当单击段落文本后，会自动解除事件委派，则第二行文本就不再拥有鼠标单击事件，但是第一个 div 元素依然保持单击事件。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").live("click", function(){     //委派事件
        $(this).after("<p>段落文本</p>");
        $("p").die("click");     //解除委派
    });
})
</script>

<p>段落文本</p>
```

die()方法与 live()方法是两个相反的操作。如果 die()方法不带参数，则所有绑定的 live 事件都会被移除；如果设置 type 参数，那么会移除对应的 live 事件；如果同时指定了第二个

参数，则只移出指定事件的处理函数。

5.2.7 jQuery 事件命名空间

jQuery 支持事件命名空间，以方便事件管理。例如，在下面的示例中，为 div 元素绑定多个事件类型，然后使用命名空间进行规范，从而方便管理。所谓事件命名空间，就是在事件类型后面以点语法附加一个别名，以便引用事件，如"click.a"，其中" a"就是 click 当前事件类型的别名，即事件命名空间。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("div").bind("click.a", function(){     //绑定 click 事件
        $("body").append("<p>click 事件</p>");
    });
    $("div").bind("dblclick.a", function(){  //绑定 dblclick 事件
        $("body").append("<p>dblclick 事件</p>");
    });
    $("div").bind("mouseover.a", function(){ //绑定 mouseover 事件
        $("body").append("<p>mouseover 事件</p>");
    });
    $("div").bind("mouseout.a", function(){  //绑定 mouseout 事件
        $("body").append("<p>mouseout 事件</p>");
    });
})
</script>

<div>jQuery 命名空间</div>
```

若在所绑定的事件类型后面附加命名空间，这样在删除事件时，就可以直接指定命名空间。例如，调用下面一行代码就可以把上面示例中绑定的事件全部删除。

```
$("div").unbind(".a");
```

同样，如果为相同的事件类型设置不同的命名空间，如果仅删除某一个事件处理程序，则只需要指定命名空间即可。例如，在下面的示例中如果直接单击段落文本，会触发命名空间为 a 的 click 事件和命名空间为 b 的 click 事件，当单击按钮之后，则删除命名空间为 a 的事件类型，则再次单击段落文本时，就只能触发命名空间为 b 的 click 事件。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("div").bind("click.a", function(){
        $("body").append("<p>click.a 事件</p>");
    });
    $("div").bind("click.b", function(){
```

```
        $("body").append("<p>click.b 事件</p>");
    });
    $("input").click(function(){
        $("div").unbind(".a");  //注销命名空间为 a 的事件
    });
})
</script>

<div>jQuery 命名空间</div>
<input type="button" value="删除事件"  />
```

另外，在 trigger()方法中，如果事件类型后面附加感叹号，则表示触发不包含命名空间的特定事件类型。例如，在下面的示例中，当单击按钮时，将会触发没有命名空间的 click 事件。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("div").bind("click", function(){
        $("body").append("<p>click 事件</p>");
    });
    $("div").bind("click.b", function(){
        $("body").append("<p>click.b 事件</p>");
    });
    $("input").click(function(){
        $("div").trigger("click!");  //注意 click 类型后面的感叹号
    });
})
</script>

<div>jQuery 命名空间</div>
<input type="button" value="删除事件"  />
```

5.2.8 jQuery 的多事件绑定

jQuery 最大优势就是提供了多种灵巧的用法，方便设计师的开发。对于在同一个对象上绑定多个事件来说，jQuery 也提供了很多种方法，这些方法适用于不同的开发环境以及习惯用法，以方便设计师加快开发速度。

例如，在下面的示例中，为当前 div 元素绑定了两个 click 事件，当单击 div 元素时，分别会触发这两个绑定的事件处理函数。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("div").bind("click", function(){  //绑定 click 事件 1
        $("body").append("<p>click 事件 1</p>");
    });
    $("div").bind("click", function(){  //绑定 click 事件 2
```

```
        $("body").append("<p>click.b 事件 2</p>");
    });
})
</script>

<div>jQuery 命名空间</div>
```

对于为同一个对象绑定的多个事件，可以以连写的形式串在一起，代码如下所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("div").bind("click", function(){
        $("body").append("<p>click 事件 1</p>");
    }).bind("click", function(){
        $("body").append("<p>click 事件 2</p>");
    });
})
</script>

<div>jQuery 多事件绑定</div>
```

使用 jQuery 定义的 bind()方法，可以为元素一次绑定多个事件类型。例如，下面的示例在同一个 bind()方法中同时绑定了 mouseover 和 mouseout 事件类型，代码如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("div").bind("mouseover mouseout", function(event){  //同时绑定多个事件类型
        $("body").append(event.type + "<br />");
    });
})
</script>

<div>jQuery 多事件绑定</div>
```

在上面的示例中，当光标移过 div 元素时，会触发 mouseover 事件，调用绑定的事件处理函数；而当光标移出 div 元素时，将再次触发 mouseout 事件，并再次调用该函数。上面的代码可以拆分为如下形式。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("div").bind("mouseover", function(event){        //绑定 mouseover 事件
        $("body").append(event.type + "<br />");
    });
    $("div").bind("mouseout", function(event){         //绑定 mouseout 事件
        $("body").append(event.type + "<br />");
    });
```

```
})
</script>

<div>jQuery 多事件绑定</div>
```

5.2.9 jQuery 自定义事件

jQuery 支持自定义事件，所有自定义事件都可以通过 jQuery()方法触发。例如，在下面的示例中自定义了一个 delay 事件类型，并把它绑定到 input 元素对象上。然后在按钮单击事件中触发自定义事件，以实现延迟响应的设计效果。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("input").bind("delay", function(event){     //自定义并绑定 delay 事件类型
        setTimeout(function(){   //延迟响应
            alert(event.type);
        },1000);
    });
    $("input").click(function(){     //绑定 click 事件
        $("input").trigger("delay"); //触发自定义事件
    });
})
</script>

<input  type="button" value="jQuery 自定义事件" />
```

实际上，自定义事件不是真正意思上的事件，读者可以把它理解为自定义函数，触发自定义事件就相当于调用自定义函数。由于自定义事件拥有事件类型的很多特性，因此自定义事件在开发中拥有特殊的用途。

5.3 jQuery 页面初始化

当浏览器在页面加载完毕之后，在 JavaScript 原生代码中，通常会使用 window.onload 方法触发 load 事件类型，这个事件类型也称为页面初始化响应事件，当页面加载完毕之后自动触发。一般利用这个事件为页面进行初始化处理工作。

5.3.1 使用 jQuery 的 ready()方法

jQuery 定义了 ready()方法封装了 JavaScript 原生的 window.onload 方法。ready()方法表示当 DOM 载入就绪，并可以查询和被操纵时，能够自动执行的函数。它是 jQuery 事件模型

中最重要的一个函数，极大地提高了 Web 应用程序的响应速度。

例如，在下面的示例中分别为三个 div 元素绑定 ready 事件，在浏览器中预览，则可以看到绑定的三个事件都在文档加载完毕后被集中触发。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$("div").eq(0).ready(function(){
    alert(1);
});
$("div").eq(1).ready(function(){
    alert(2);
});
$("div").eq(2).ready(function(){
    alert(3);
});
</script>

<div>模块 1</div>
<div>模块 2</div>
<div>模块 3</div>
```

因此，ready()方法纯粹是 window.load 事件的替代方法。通过使用这个方法，可以在 DOM 载入就绪并能够读取并操纵时立即调用所绑定的函数，实际上 JavaScript 中的绝大部分函数都需要在那一刻执行。

ready()方法一般按如下方式进行调用。

```
$(document).ready(function(){
    //页面初始化后执行的函数体代码
});
```

对于上面的语法格式可以简写为以下格式。

```
jQuery(function($) {
    //页面初始化后执行的函数体代码
});
```

或者

```
$(function() {
    //页面初始化后执行的函数体代码
});
```

在上面的格式中，我们可以看到 ready()方法包含一个参数，该参数为一个事件处理函数。同时，事件处理函数也包含一个参数，该参数引用 jQuery 函数，并实现把 jQuery 函数传递到 ready 事件处理函数内。因此，读者也可以给这个参数起一个别名，以方便在函数体内部进行引用，同时不再担心命名冲突而放心使用这个别名。

例如，在下面的示例中为 jQuery 起一个别名 me，即设置 ready 事件处理函数的参数名为 me，则在页面初始化处理函数中就可以使用 me 来代替 jQuery 函数。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(me){
    me("div").text("jQuery 函数别名");     //这里的别名 me 指代 jQuery 函数
});
</script>

<div></div>
```

当然读者也可以省略这个参数，在默认状态下 jQuery 会使用$或者同名 jQuery 别名来指代 jQuery 函数，代码如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("div").text("jQuery 函数别名");
    jQuery("div").text("jQuery 函数别名");
});
</script>

<div></div>
```

jQuery 允许在文档中无限次使用 ready 事件，其中注册的事件处理函数会按照代码中的先后顺序依次执行。但是，一旦使用 jQuery 事件模型中的 ready 事件初始化页面，就不能够使用 JavaScript 原生的 load 事件类型了，否则就会发生冲突，而不能够触发 ready 事件。

5.3.2 ready 事件的触发时机

jQuery 的 ready 事件与 JavaScript 的 load 事件具有相同的功能，但是它们在触发时机方面还存在以下细微的区别。

- JavaScript 的 load 事件是在文档内容完全加载完毕后才被触发，这个文档内容包括页面中所有节点以及与节点关联的文件。这时 JavaScript 才可以访问网页中任何元素和内容。这种情况对于编写功能性的代码非常有利，因为无需考虑加载的次序。
- jQuery 的 ready 事件是在 DOM 完全就绪时就可以被触发，此时文档中所有元素都是可以访问的，但是与文档关联的文件可能还没有下载完毕。通俗说，就是浏览器下载并完成了解析 HTML 的 DOM 树结构后，代码就可以运行。

例如，对于一个大型图库网站来说，为页面中所有显示的图像绑定一个初始化设置的脚本。如果使用 JavaScript 原生的 load 事件来设计，那么用户在使用这个页面之前，必须等待

页面中所有图像下载完毕才能够实现。而在 load 事件等待图像加载过程中，如果行为还未添加到那些已经加载的图像上，则此时如果用户操作它们，可能会导致很多意想不到的尴尬。而使用 jQuery 的 ready 事件，则在 DOM 树结构解析之后，就立即触发页面初始化事件，从而避免使用 load 事件所带来的尴尬。

但是，由于 jQuery 的 ready 事件过早的触发，虽然 DOM 树结构已经解析完毕，但是很多元素的属性未必生效。例如，很多图像还没有加载完毕，导致这些图像的属性无效，如图像的高度和宽度。要解决这个问题，可以使用 jQuery 的 load 事件进行触发，该事件等效于 JavaScript 的 load 事件。

```
$(window).load(function(){
    //页面初始化后执行的函数体代码
})
```

等效于：

```
window.onload = function(){
    //页面初始化后执行的函数体代码
}
```

5.3.3 初始化事件的多次调用

JavaScript 的 load 事件存在一个很严重的缺陷：就是它不允许多次调用。例如，在下面的示例中分两次调用 load 事件，但是当网页加载完毕后，JavaScript 仅触发了第二个 load 事件调用。

```
<script type="text/javascript" >
window.onload = function(){
    alert("一次调用 load 事件");
}
window.onload = function(){
    alert("二次调用 load 事件");
}
</script>
```

实际上，第一次事件调用已经被第二个调用覆盖。要解决两次调用之间的冲突问题，则可以把两个页面初始化函数放在同一个 load 事件中。例如，针对上面的示例可以按如下方式进行修改。

```
<script type="text/javascript" >
window.onload = function(){
     (function(){
        alert("一次调用 load 事件");
    })();
     (function(){
```

```
        alert("二次调用 load 事件");
    })()
}
</script>
```

上面的代码直接在 load 事件的处理函数中定义和调用两个匿名函数。当然也可以把这两个匿名函数改为函数声明方式定义，然后在 load 事件处理函数中调用。

即便这样，通过间接的方式解决 load 事件多次调用的问题，但是 load 事件仍然存在很多局限。例如，在多个 JavaScript 文件中，可能每个 JavaScript 文件都会用到 window.load()方法，在这种情况下使用上面的方法是无法解决的，同时也无法保证按顺序执行多个注册的函数。

而 jQuery 的 ready 事件能够很好地解决这个问题，在同一个文档中可以进行多次调用。例如，针对上面的示例，可以使用如下方法轻松解决，即便在不同的 JavaScript 文件中，都可以无限制多次调用 ready 事件。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    alert("一次调用 load 事件")
});
$(function(){
    alert("二次调用 load 事件")
});
</script>
```

5.3.4 使用 JavaScript 自定义 addLoadEvent()方法

针对页面初始化处理问题，jQuery 提供了灵活和方便的解决方案，但是 jQuery 的 ready 事件与 JavaScript 的 load 事件存在天然的冲突。如果读者在页面中对于 jQuery 的 ready 事件使用需求不是那么强烈，不妨自定义一个 addLoadEvent()方法来解决 window.onload 方法注册事件存在的缺陷。addLoadEvent()方法的代码如下。

```
<script type="text/javascript" >
function addLoadEvent(func){
    var oldOnload = window.onload;  //把 window.onload 事件处理函数的值存入变量 oldOnload
    if (typeof window.onload != 'function'){      //如果 window.onload 事件还没有绑定任何事件处理函数，则为其添加新的事件处理函数
        window.onload = func;
    }else{     //如果 window.onload 事件已绑定事件处理函数，则重新绑定事件处理函数，在事件处理函数中先执行原来已绑定的事件处理函数，然后调用新添加的事件处理函数
        window.onload = function(){
            oldOnload();
            func();
        }
```

```
    }
}
</script>
```

虽然说，window.onload 只能够被赋值一次，也就是说 load 事件只能绑定一个事件处理函数。但是我们可以设置 load 的事件处理函数为一个管道，并借助这个管道无限次地调用 load 事件。addLoadEvent()方法的工作流程如下。

(1) 先把现有的 window.onload 事件处理函数的值存入变量 oldOnload。

(2) 如果在这个处理函数上还没有绑定任何函数，则为其添加新的函数。

(3) 如果在这个处理函数已经绑定了一些函数，就把函数追回到现有函数的尾部。

浏览器在加载 HTML 文档内容时，在默认状态下是自上而下地执行 JavaScript 代码，如果要改变 load 事件处理函数的执行顺序，可以利用 addLoadEvent()方法改变调用顺序。

在开发过程中，如果需要给 load 事件绑定多个函数，但又不确定 load 事件是否已经绑定了函数，则使用 addLoadEvent()方法就能够很轻松地解决这个问题。例如，借助 addLoadEvent()方法，我们可以轻松多次调用 load 事件，代码如下。

```
<script type="text/javascript" >
//省略 addLoadEvent()方法的声明
addLoadEvent(function(){
    alert("一次调用 load 事件");
});
addLoadEvent(function(){
    alert("二次调用 load 事件");
});
addLoadEvent(function(){
    alert("三次调用 load 事件");
});
</script>
```

5.4 使用 JavaScript 自定义 jQuery 事件方法

jQuery 确实给开发者提供了极大的方便。尽管$(document).ready 非常有用，ready 事件可以在页面渲染时，其他元素还没下载完成就执行，但是如果当页面总是处于载入中的状态，也许通过 load 事件可以减少页面载入时 CPU 的使用率，这在一些特效的应用中，如拖放、视觉特效、动画和大文件预载，表现明显。

所以，当你陶醉于乱用 jQuery 方法时，请时刻保持警惕，并铭记效率是代码的生命。如果可以保证你的技术和精力，请尽力使用 JavaScript 原生的方法，这将会确保你的程序变得更加高效。

5.4.1 JavaScript 与 jQuery 的执行效率比较

原生的 JavaScript 代码总会比使用诸如 jQuery 代码库的写法执行效率更高。选择器是一种非常低效的用法，同样 jQuery 事件模型的运行速度也要比原生的事件模型慢好几倍。

下面做一个简单的试验，为同一个 div 元素注册 1000 次 click 事件，比较 JavaScript 和 jQuery 的运行速度。其中 JavaScript 的示例代码如下。

```
<script type="text/javascript" >
window.onload = function(){
    var div = document.getElementsByTagName("div")[0] ;          //获取 div 元素
    var t1, t2;    //声明临时变量
    if(div.addEventListener) {  //兼容非 IE 浏览器
        t1 = new Date();     //开始计时
        for(var i = 0; i <1000; i++){
            div.addEventListener("click", function(){}, false);//注册事件
        }
        t2 = new Date();     //结束计时
    }else{     //兼容 IE 浏览器
        t1 = new Date();     //开始计时
        for(var i = 0; i <1000; i++){
            div.attachEvent("onclick",function(){});              //注册事件
        }
        t2 = new Date();     //结束计时
    }
    alert("执行时间 = "+ (t2-t1) + " 毫秒");                      //返回执行时间
}
</script>

<div></div>
```

然后使用 jQuery 为 div 元素注册 1000 次 click 事件，为了避免其他代码的影响，这里把 jQuery 注册方法放在 JavaScript 的 load 事件中进行测试，代码如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
window.onload = function(){
    var $div = $("div");    //通过缓存提前获取 div 元素，避免 jQuery 选择器对于循环测试的影响
    var t1 = new Date();    //开始计时
    for(var i = 0; i <1000; i++){
        $div.bind("click", function(){});     //调用 bind()方法绑定事件
    }
    var t2 = new Date();    //结束计时
    alert("执行时间 = "+ (t2-t1) + " 毫秒");    //返回执行时间
}
</script>

<div></div>
```

分别在不同浏览器中进行测试，可以看到 JavaScript 要比 jQuery 快 3 倍到 10 多倍不等。其中在 IE 和 Safari 浏览器中可以看到两者运行速度差距在 10 倍左右，在 Chrome 浏览器中两者运行速度差距最小，但是差距也在 3 倍左右。当然，每一次执行的情况会略有不同，下面分别在 Firefox 3.5 浏览器中运行，则 JavaScript 方法的运行速度如图 5.12 所示，而 jQuery 方法的运行速度如图 5.13 所示。

图 5.12 JavaScript 执行时间

图 5.13 jQuery 执行时间

总之，对于页面初始化这样类型的事件，建议读者采用 JavaScript 的 load 方法会更妙，同时，使用 load 方法注册初始化事件不用担心浏览器的兼容性问题，况且 jQuery 方法和 JavaScript 方法可以混合使用，只要读者注意区分 jQuery 对象和 JavaScript 对象的相互转换即可。而对于其他交互型事件，如果要在页面中频繁使用，也建议采用 JavaScript 原生的方法。当然，对于浏览器兼容性存在很大差异的鼠标事件来说，采用 jQuery 提供的事件解决方案会更方便。

5.4.2 自定义 ready()方法

jQuery 的 ready 事件实际上是对 IE 的 readystatechange 事件和 DOM 的 DOMContentLoaded 事件进行封装，这两个事件都是在 DOM 树结构下载并解析完毕后触发。使用 JavaScript 自定义 ready()方法的源代码如下。

```
<script type="text/javascript" >
var $ = ready = window.ready = function(fn){
    if ( document.addEventListener ) {   //兼容非 IE
        document.addEventListener( "DOMContentLoaded", function(){
     //绑定 DOMContentLoaded 事件处理函数
            document.removeEventListener( "DOMContentLoaded", arguments.callee, false );
     //绑定 DOMContentLoaded 事件后，立即注销该事件，避免反复触发
            fn();   //调用参数函数
        }, false );
    } else if( document.attachEvent ) { //兼容 IE
        document.attachEvent("onreadystatechange", function(){
     //绑定 readystatechange 事件
            if ( document.readyState === "complete" ) {     //如果当前文档结构加载完毕
```

```
                document.detachEvent( "onreadystatechange", arguments.callee );  // 则
注销 readystatechange 事件，避免反复触发
                fn();     //调用参数函数
            }
        });
    }
}

$(function(){//测试使用 JavaScript 自定义的 ready 事件
    alert("ok");
})
</script>
```

在正常情况下，JavaScript 的 load 事件在所有页面元素(包括图片、脚本等)都下载完毕后才会触发。 除了 IE 浏览器，其他浏览器还支持 DOMContentLoaded 事件。当 DOM 内容下载完毕，即当 DOM 树加载完毕后，就会立刻触发 DOMContentLoaded 事件。因为多数时候只是获取 DOM 节点，如果使用 window.onload 进行响应，就必须等到 image、flash 和 iframe 等内容都加载完毕才会执行，其实在这些外部文件载入之前，页面中的 DOM 树早已载入完毕，可以调用了，所以，使用 load 事件响应会浪费大量的时间等待 DOM 以外的内容加载。

针对 IE 浏览器，有各种模拟 DOMContentLoaded 事件的办法。目前被广泛采纳的方案是判断 document 是否可以滚动(doScroll)。一旦可以滚动，就意味着 DOM Content 已经加载完毕。不过也可以使用 IE 的私有事件 readystatechange 来进行响应。

在使用 ready 事件时，应该注意以下两个陷阱。

- 在 setTimeout ()方法里给 window.onload 添加事件处理函数。这是不可靠的，无法保证添加的处理函数会被执行。事件处理函数是否执行取决于 setTimeout ()方法的延时和页面内容的加载时间。例如：

```
setTimeout(function() {
    onLoad(function() {
        alert('文档已经加载');     //不一定被执行
    });
}, 20);
```

- 在动态加载 onload 事件时，给 DOMContentLoaded 添加事件处理函数也是不可靠的，除了 IE 浏览器，其他浏览器下根本不会执行。

5.4.3 自定义 bind()方法

jQuery 的 bind()方法中包含一个附加数据参数，因此使用 JavaScript 自定义 bind()方法时就需要考虑如何把用户的数据传递到 Event 对象中去。为了实现这个问题，我们需要在自定义 bind()方法中调用事件处理函数，然后在调用过程中把用户附加的数据传递进去，最后再绑定到事件注册方法中。实现的详细代码如下。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("bind", function(type,data,fn){    //为 HTML 元素自定义事件绑定方法，其中参数 type
表示事件类型，参数 data 表示用户要附加的数据，参数 fn 表示绑定的事件处理函数
    var _this = this;
    if(_this.addEventListener) //兼容非 IE 浏览器
        _this.addEventListener(type,function(event){  //注册事件
            event.datas = data;  //把用户附加的数据集绑定到 Event 对象上
            fn(event);  //执行用户传递的事件处理函数
        },false);
    else{ //兼容 IE 浏览器
        _this.attachEvent("on"+type, function(){  //注册事件
            var event = window.event;
            event.datas = data;  //把用户附加的数据集绑定到 event 属性上
            fn(event);  //执行用户传递的事件处理函数
        });
    }
    return _this; //返回当前元素
})
//测试 bind()方法
window.onload = function(){     //页面初始化
    var btn = document.getElementsByTagName("input")[0];
    var i = 1;
    btn.bind("click",{a:"this is a",b:"this is b"},function(event){
     //绑定鼠标单击事件
        btn.value = event.datas.a + i++; //获取用户传递的数据
    })
}
</script>
<input type="button" value="自定义 bind()方法"  />
```

在自定义 bind()方法中，第一个参数为事件类型，由于 IE 浏览器的 attachEvent()方法的第一个参数为事件属性，所以需要在事件类型前面附加一个"on"前缀，用户附加的数据以对象结构进行传递，可以通过 Event 对象的自定义属性 datas 进行引用。由于在 IE 浏览器中，window.event 的 data 属性表示一个备份关键字，所以本方法中定义 datas 属性名作为读取 Eveent 对象的自定义属性，来引用用户附加的数据。

在上面的示例中，当单击按钮时，将触发绑定的鼠标单击事件，并调用绑定的事件处理函数，读取并显示用户在绑定事件时传递的附加数据，演示效果如图 5.14 所示。

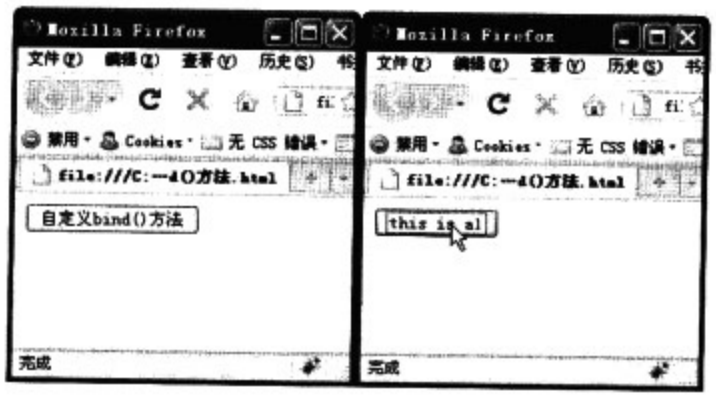


图 5.14 自定义 bind()及其应用效果

5.4.4 自定义 one()方法

jQuery 的 one()方法实际上是 bind()方法的简单封装，在事件注册函数中添加注销事件的行为，从而在用户单击一次之后，该事件即被注销，从而不再响应第二次操作。使用 JavaScript 自定义 one()方法实现的详细代码如下。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("one", function(type,data,fn){     //为 HTML 元素自定义事件绑定方法，其中参数 type
表示事件类型，参数 data 表示用户要附加的数据，参数 fn 表示绑定的事件处理函数
    var _this = this;
    if(_this.addEventListener) //兼容非 IE 浏览器
        _this.addEventListener(type,function(event){  //注册事件
            _this.removeEventListener(type,arguments.callee,false); //注销事件
            event.datas = data;  //把用户附加的数据集绑定到 Event 对象上
            fn(event);  //执行用户传递的事件处理函数
        },false);
    else{ //兼容 IE 浏览器
        _this.attachEvent("on"+type, function(){  //注册事件
            _this.detachEvent("on"+type, arguments.callee);     //注销事件
            var event = window.event;
            event.datas = data;  //把用户附加的数据集绑定到 event 属性上
            fn(event);  //执行用户传递的事件处理函数
        });
    }
    return _this; //返回当前元素
})
//测试 one()方法
window.onload = function(){     //页面初始化
    var btn = document.getElementsByTagName("input")[0];
    var i = 1;
    btn.one("click",{a:"this is a",b:"this is b"},function(event){ //绑定鼠标单击事件
        btn.value = event.datas.a + i++; //获取用户传递的数据
    })
}
</script>

<input type="button" value="自定义 bind()方法"  />
```

由于 one()方法仅是在 bind()方法基础上添加了注销行为，故不再进行详细讲解。在调用 one()方法绑定事件时，该事件响应一次之后即被销毁，所以就不存在第二次响应问题。

第6章 动画效果设计及其高效实践

JavaScript 程序本身是无法实现特效的，它必须借助 CSS 技术来实现动画效果。读者可以先在样式表中定义好类样式，然后再通过 jQuery 或 JavaScript 来添加或移出这些类样式。一般而言，这都是 HTML 应用 CSS 的首选方式，当然我们也可以通过脚本样式来动态控制文档的结构，甚至设计出各种复杂的动画效果。本章将多角度分析各类动画的设计和实现途径，并比较相同效果在 jQuery 和 JavaScript 中的实现方法。

6.1 直接显示和隐藏

最简单的动画效果也许就是元素的显示和隐藏了。在 jQuery 中，使用 show()方法可以显示元素，使用 hide()方法可以隐藏元素。如果把 show()和 hide()方法配合起来，就可以设计最基本的显隐动画。

6.1.1 jQuery 实现显隐效果

show()和 hide()方法是 jQuery 设计显隐动画效果的基础。这两个方法用法很简单，jQuery 通过将 style 属性的 display 值设置为 none 来使元素隐藏。如果元素已经隐藏，则保持隐藏状态，并返回这些元素。

如果在初始化时，将 style 属性的 display 值设置为 none 使元素隐藏，则调用 jQuery 的 show()方法之后，元素的 style 属性的 display 属性声明将被清空；如果在初始化时，使用 jQuery 的 hide()方法隐藏元素，在调用 jQuery 的 show()方法之后，元素的 style 属性的 display 属性将显示元素的默认值。

下面的示例将演示 jQuery 的 show()和 hide()方法的应用和影响，结构变化效果如图 6.1 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){
    $("p").hide().hide();
    $("div").hide().show();
    $("span").eq(0).hide();
    $("span")[1].style.display = "none";
    $("span").show();
})
</script>

<p>P 元素</p>
<div>DIV 元素</div>
<span>SPAN 元素 1</span>
<span>SPAN 元素 2</span>
<span style="display:none;">SPAN 元素 3</span>
```

除了简单的显示和隐藏功能外，show()和 hide()方法还可以设置参数，以优雅的动画显示所有匹配的元素，并在显示完成后可选地触发一个回调函数。例如，在下面的示例中，调用 show()和 hide()方法，并设置显隐过程为 1000 毫秒，同时在显隐动画播放完毕之后，调用第二个参数回调函数，弹出一个提示对话框。

图 6.1 jQuery 的 show()和 hide()方法应用的结构变化效果图

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
$( function(){
    var t = false;
    $( "input" ).click( function(){
        if( t ){
            $( "div" ).show(1000,function(){
                alert("显示 DIV 元素");
            });
            $( "input" ).val("隐藏元素");
            t = false;
        }
        else{
            $( "div" ).hide(1000,function(){
                alert("隐藏 DIV 元素");
            });
            $( "input" ).val("显示元素");
            t = true;
        }
    });
})
</script>

<input type="button" value="隐藏元素" />
<div>DIV 元素</div>
```

这两个方法的第一个参数表示动画时长的毫秒数值，也可以设置预定义的字符串("slow"、"normal"、"fast")，用来表示动画的缓慢、正常和快速效果。在 jQuery 1.3 版本中，padding 和 margin 样式属性也会有动画，而且效果更流畅。第二个参数是一个可选参数，表示在动画演示完毕后要调用的回调函数。

6.1.2 JavaScript 实现显隐效果

对于简单的显隐控制，使用 JavaScript 可以轻松实现，但是如果设计元素的显隐过程以动画方式显示，则应该考虑渐隐和渐显效果设计。关于这个技术话题，将在后面的小节中进行详细讲解。

使用 JavaScript 自定义 show()和 hide()方法的代码如下。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("show",function(){    //自定义 show()方法，显示元素
    var _this = this;
    _this.style.display = "";   //恢复元素的默认显示状态
    return _this;
})
DOMextend("hide",function(){    //自定义 show()方法，隐藏元素
    var _this = this;
    _this.style.display = "none";   //隐藏元素的显示状态
    return _this;
})
//应用自定义方法
window.onload = function(){
    var t = false;//初始化临时变量
    var input = document.getElementsByTagName("input")[0];     //获取按钮元素
    var div = document.getElementsByTagName("div")[0];     //获取 div 元素
    input.onclick = function(){//绑定单击事件处理函数
        if( t ){
            div.show(); //调用自定义 show()方法，显示元素
            input.value = "隐藏元素";
            t = false;
        }
        else{
            div.hide(); //调用自定义 hide()方法，隐藏元素
            input.value = "显示元素";
            t = true;
        }
    };
}
</script>

<input type="button" value="隐藏元素" />
<div>DIV 元素</div>
```

在上面的示例中，使用 JavaScript 自定义 show()和 hide()方法，并把它们绑定到 HTML 元素上，这样可以在当前元素中调用 show()和 hide()方法，实现显示和隐藏元素。在本示例中，利用自定义 show()和 hide()方法能模拟出上节示例的效果。

6.1.3 折叠效果

折叠是网页设计中经常用到的效果，实现起来也很简单。但是为了技术的规范性和适应性，本节将对 JavaScript 代码进行简单的封装，实现在相同的结构和类样式下，都可以获得相同的折叠效果，演示效果如图 6.2 所示。

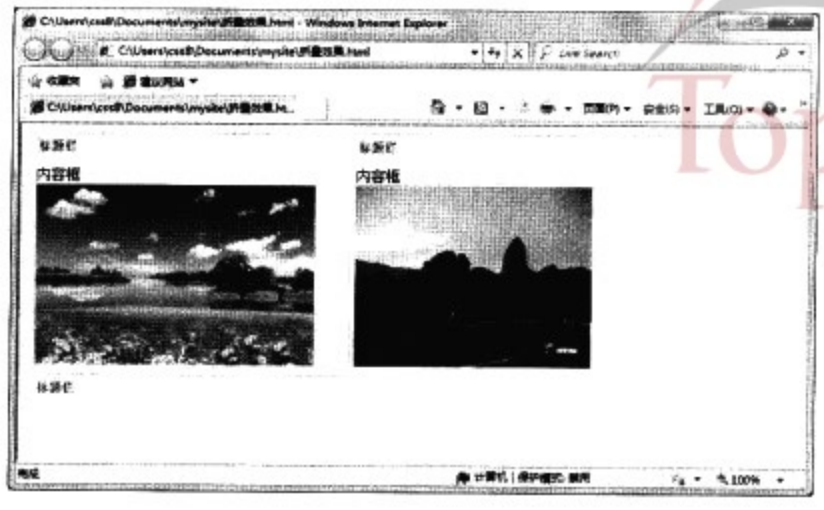


图 6.2 折叠效果

首先定制折叠面板的 HTML 结构，这里选用 dl、dt 和 dd 三个元素配合使用，既符合语义性，也方便管理。并设想只要文档中包含 collapse 类样式的 dl 元素，并确保只包含一个 dt 和 dd 子元素，都可以拥有相同的折叠效果，代码如下所示。

```
<dl class="collapse">
    <dt>标题栏</dt>
    <dd>内容框<br /><img src="images/bg2.jpg" height="200" /></dd>
</dl>
```

关于该模块结构的样式设计就不再说明，读者可以自由设计，或者参考本书光盘示例源代码。

1. JavaScript 实现

在 JavaScript 内建的核心中，Document 对象及 Element 对象只能用以下三个方式来获取元素。

- getElementById('id')
- getElementsByName('name')
- getElementsByTagName('tag')

这些方法分别根据元素的 id、name 和 tag 来获取元素。在 Web 文档中，id 具有惟一性，所以 getElementById(id)方法获取的对象也是惟一的，可以直接使用；而其他两个方法则会返回一个根据具有该属性的元素在文档中出现顺序排列的数组，使用时必须指定数组编号，如 array[0]表示第 1 个元素。

但是在设计网页时，经常会用到 class 属性，而 JavaScript 没有定义相应的方法以获取同类元素。所以，在实现折叠效果之前，我们先为 Document 对象预定义一个 getElementsBy ClassName()方法，以方便获取指定类元素。

```
<script type="text/javascript">
document.getElementsByClassName = function(className) {  //扩展 Document 对象方法,获取同
类元素
```

```
    var el = [],  //定义临时数组
    _el = document.getElementsByTagName('*');     //获取文档中所有元素
    for (var i=0; i<_el.length; i++ ) {           //遍历文档元素
      if (_el[i].className == className ) { //如果元素的类名与指定参数相同，则传入数组存储
        el[el.length] = _el[i];
      }
    }
    return el; //返回存储同类元素的数组
}
</script>
```

这个方法称不上优雅，因为需要遍历文档，并检测元素是否包含相同的类名，最后返回符合条件的元素数组。不过目前还没有更好的解决方案。

下面就可以利用这个 **getElementsByClassName()**方法，以及前一节讲解的 **show()**和 **hide()**方法来设计折叠动画了。详细代码如下。

```
<script type="text/javascript" >
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
//省略了自定义 show()方法，请参阅第 6.1.2 节内容
//省略了自定义 hide()方法，请参阅第 6.1.2 节内容
window.onload = function(){    //页面初始化
    var t = [];    //定义空数组
    var dl = document.getElementsByClassName("collapse");     //获取类名为"collapse"的所有元素
    for(var i=0; i<dl.length;i++){  //遍历 collapse 类中所有元素
        var dt = dl[i].getElementsByTagName("dt")[0]; //获取 collapse 类元素下的 dt 元素
        var dd = dl[i].getElementsByTagName("dd")[0]; //获取 collapse 类元素下的 dd 元素
        t[i] = false;  //设置折叠初始状态
        dt.onclick = (function(i,dd){    //绑定单击事件处理函数
            return function(){   //返回一个闭包函数，闭包能够存储传递进来的动态参数值
                if( t[i] ){
                    dd.show();    //显示元素
                    t[i] = false;
                }else{
                    dd.hide();    //隐藏元素
                    t[i] = true;
                }
            }
        })(i,dd);  //向当前执行函数中传递参数
    }
}
</script>
```

当执行上面脚本后，JavaScript 会自动为文档中所有的 collapse 类元素绑定折叠效果。当单击 collapse 类元素包含的 dt 子元素时，则会自动显隐 dd 元素。演示效果可以参阅图 6.2。

2. jQuery 实现

继续以上面示例的 HTML 结构为基础，尝试使用 jQuery 来实现折叠效果。由于 jQuery 已经封装了 getElementsByClassName()、show()和 hide()方法，所以可以直接调用。实现折叠效果的详细代码如下所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){//页面初始化处理函数
    var t = [];    //定义空数组
    var dt = $("dl.collapse dt");    //获取类名为 collapse 的 dl 元素包含的所有 dt 子元素
    var dd = $("dl.collapse dd");    //获取类名为 collapse 的 dl 元素包含的所有 dd 子元素
    dt.each(function(i){    //遍历所有的 dt 元素，并向函数传递遍历序号
        t[i] = false;   //设置折叠初始状态
        $(dt[i]).click((function(i,dd){  //为当前 dt 元素绑定 click 事件处理函数
            return function(){    //返回一个闭包函数，闭包能够存储传递进来的动态参数值
                if( t[i]){
                    $(dd).show(); //显示元素
                    t[i]  = false;
                }else{
                    $(dd).hide(); //隐藏元素
                    t[i]  = true;
                }
            }
        })(i,dd[i]));   //向当前执行函数中传递参数
    })
})
</script>
```

使用 jQuery 设计的思路与 JavaScript 设计思路完全相同，不过 jQuery 已经封装了 getElementsByClassName()、show()和 hide()方法，所以就会节省很多代码。同时 jQuery 使用 each()方法封装了 for 循环结构，实现快捷遍历文档节点。each()方法包含一个默认的参数，该参数可以传递遍历过程中元素的序列位置，以方便动态跟踪每个元素。

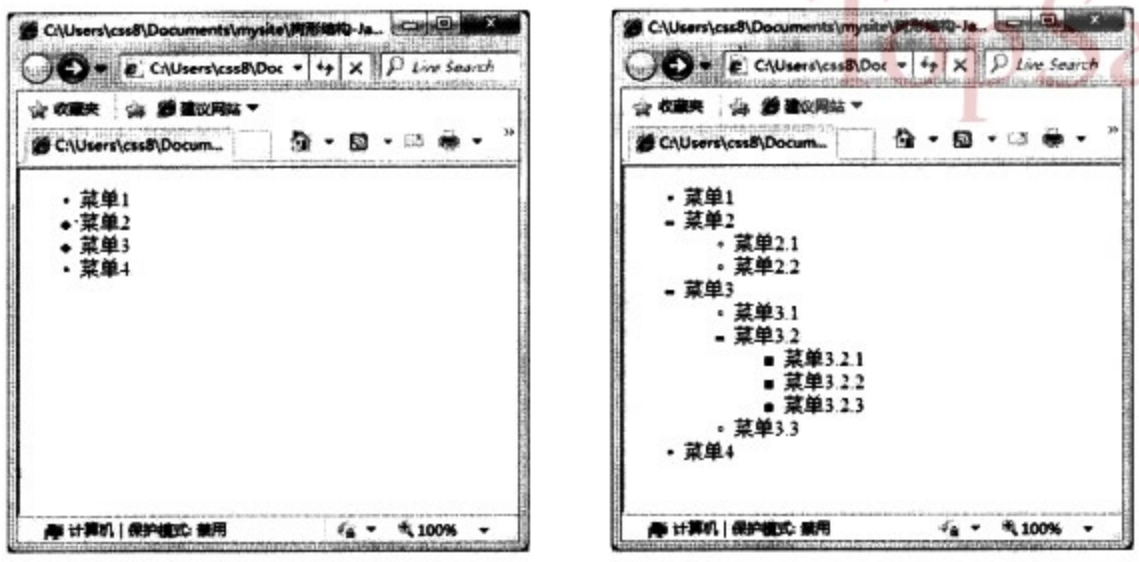
考虑到在元素遍历的过程中，动态定位元素比较困难，这里使用了闭包函数存储元素的序列位置，由于在闭包中无法访问闭包函数外的对象，故还需要向其传递当前要操作的元素对象。

在多层嵌套的结构中，应注意大括号和小括号的使用，要避免缺少使用小括号运算符，如$(dt[i]).click((function(i,dd){ //.....})(i,dd[i]));。

6.1.4 树形结构

文件系统通常以层次结构列表形式显示，在层次结构列表里文件夹包含的内容相互嵌套，以便表示各种复杂的包含关系，这种类似树形结构的设计效果，在网页中经常见到，如目录

导航，效果如图 6.3 所示。另外，在网页中看到的多级菜单也是一种经典的树形结构。



(a)　　(b)

图 6.3　树形结构

树形结构的第一步，就是要设计一个好的 HTML 结构。从语义性角度考虑，选择列表结构是最恰当的，当然读者也可以使用 div 和 span 元素实现相同的显示效果。列表结构在多层嵌套时，会自动显示出多层结构的关系，即使不使用 CSS 进行样式设计，仍然可以让人一目了然。图 6.3(b)树形结构的 HTML 代码如下所示。

```
<ul class="tree">
    <li>菜单 1</li>
    <li>菜单 2
        <ul>
            <li>菜单 2.1</li>
            <li>菜单 2.2</li>
        </ul>
    </li>
    <li>菜单 3
        <ul>
            <li>菜单 3.1</li>
            <li>菜单 3.2
                <ul>
                    <li>菜单 3.2.1</li>
                    <li>菜单 3.2.2</li>
                    <li>菜单 3.2.3</li>
                </ul>
            </li>
            <li>菜单 3.3</li>
        </ul>
    </li>
    <li>菜单 4</li>
</ul>
```

整个树形结构包含在 ul 容器中，每一个 li 元素作为一个选项进行呈现，不同层次的结构分别以 ul 子元素进行包裹，从而实现层层嵌套的关系。

1. JavaScript 实现

有了完善的树形结构，下一步是考虑如何使用 JavaScript 脚本进行控制。树形结构的动画效果设计思路如下。

先获取树形结构中所有的 li 元素，因为 li 元素代表一个选项，不管该选项处于什么层次位置都可以被选取。然后，遍历 li 元素集合，在遍历过程中检测当前 li 元素是否包含 ul 元素。如果包含 ul 元素，则设置临时标识变量 b 为 true，否则设置变量 b 为 false。

- 如果 b 为 true，则设置当前 li 元素的样式(如鼠标样式和列表项目符号)，并获取 li 元素包含的第一个 ul 元素，并隐藏该 ul 元素。然后为当前 li 元素绑定 click 事件处理函数。在该事件处理函数中，根据 ul 元素是否显示为条件，分别隐藏或者显示 ul 元素，同时动态修改当前 li 元素的样式。
- 如果 b 为 false，则设置当前 li 元素的样式为默认状态。

为了防止单击当前 li 元素的子元素时触发 click 事件，应该检测当前单击的元素是否为 li 元素。为此，可以使用 Event 对象的 target(兼容非 IE 浏览器)或 srcElement(兼容 IE 浏览器)属性进行判断。完整的 JavaScript 脚本代码如下。

```
<script type="text/javascript">
window.onload = function(){    //页面初始化处理函数
   var li = document.getElementsByTagName("li");     //获取页面中所有 li 元素
   var t =[];     //定义临时数组
   for(var i = 0; i < li.length; i ++ ){    //遍历数组
      var child = li[i].childNodes;     //获取当前 li 元素包含的所有子节点
      var b = false; //定义临时变量，并初始化为 false
      for(var j=0; j<child.length;j++){//遍历当前 li 元素包含的节点，并检测是否包含 ul 元素
         if(child[j].nodeType == 1 && child[j].nodeName.toLowerCase() == "ul")
            b = true;     //如果 li 元素包含 ul 元素，则设置 b 为 true
      }
      if(b){ //如果 li 元素包含 ul 元素
         li[i].style.cursor = 'pointer';          //定义当前 li 元素的鼠标指针样式为手形
         li[i].style.listStyleImage = 'url(images/+.gif)';   //修改当前 li 元素的选项列
表图标形状
         var ul =  li[i].getElementsByTagName("ul")[0]; //获取第一个 ul 子元素
         ul.style.display = "none";     //隐藏第一个 ul 元素
         t[i] = true;     //设置当前序号位置的数组元素的值为 true
         li[i].onclick = (function(o,li,i){     //绑定 click 单击事件处理函数
            return function(e){   //返回闭包函数
               if(li == e.target || li == window.event.srcElement ){
//如果当前元素就是事件触发的目标对象，则允许执行。这样做的目的是防止单击当前 li 元素的子元素时，也触
发 cilck 事件
                  if( t[i]){      //如果当前数组元素值为 true
                     o.style.display = "";     //恢复显示 ul 元素
                     li.style.listStyleImage =  'url(images/-.gif)';
//修改 li 元素项目列表符号
                     t[i] = false;          //切换当前数组元素值为 false
```

```
                    }
                    else{       //如果当前数组元素值为 false
                        o.style.display = "none";     //隐藏显示 ul 元素
                        li.style.listStyleImage =  'url(images/+.gif)';     //修改 li
元素项目列表符号
                        t[i] = true;     //切换当前数组元素值为 true
                    }
                }
                if ( e && e.stopPropagation )   //兼容非 IE 浏览器
                    e.stopPropagation();     //阻止事件传播
                else //兼容 IE 浏览器
                    window.event.cancelBubble = true;     //阻止事件传播
                return false; //避免触发默认事件
            }
        })(ul,li[i],i); //调用当前函数，并传递当前 li 元素及其包含的第一个 ul 元素，以及当前
li 元素位置的序号
    }
    else{  //如果 li 元素不包含 ul 元素
        li[i].style.cursor = 'default';   //恢复 li 元素的鼠标默认样式
        li[i].style.listStyleImage = 'none';  //恢复 li 元素的默认列表项目符号
    }
  }
}
</script>
```

2. jQuery 实现

根据 JavaScript 的设计思路，下面尝试使用 jQuery 来实现相同的设计效果。详细代码如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$( function(){    //页面初始化处理函数
    $( 'li:has(ul)' ).click( function( event ){ //如果 li 元素包含 ul 元素，则绑定 click 事件
        if ( this == event.target ) {     //如果当前 li 元素就是事件触发的目标对象
            if ( $( this ).children().is( ':hidden' ) ) {  //如果当前 li 元素的子元素隐藏，
则修改 li 元素的项目列表符号，并显示所有子元素
                $( this ).css( 'list-style-image', 'url(images/-.gif)' ).children().show();
            }
            else {  //否则修改 li 元素的项目列表符号，并隐藏所有子元素
                $( this ).css( 'list-style-image', 'url(images/+.gif)' ).children().hide();
            }
        }
        return false;
    }).css( { //设置包含 ul 子元素的 li 元素的样式
        cursor : 'pointer',      //设置鼠标样式为手形
        'list-style-image' : 'url(images/+.gif)'  //设置项目列表符号为减号样式
    }).children().hide();  //隐藏当前 li 元素的所有子元素
    $( 'li:not(:has(ul))' ).css( {  //如果 li 元素没有包含 ul 元素，则设置它的样式
        cursor : 'default',      //恢复默认的鼠标样式
        'list-style-image' : 'none'  //恢复默认的项目列表符号
    });
```

```
});
</script>
```


6.1.5 Tab 选项卡

如果说树形结构体现的是一种多层次结构关系，那么 Tab 选项卡体现的就是索引结构关系。通过 Tab 索引可以快速定位到相应的模块选项。Tab 选项卡在分类组织方面功能显著。在 Web 开发中，这种以 Tab 选项卡形式设计的页面或者模块比较常见，如图 6.4 所示。

图 6.4 Tab 选项卡

Tab 选项卡的结构通常按二叉型进行设计，框 1(分支一)负责组织 Tab 标题内容，而框 2(分支二)负责组织每个 Tab 选项卡对应的显示内容。在设计时，为了方便控制，应确保 Tab 标题序列与内容序列一一对应，这样可以方便程序进行控制。本案例的 HTML 结构如下所示。

```
<div class="tab">
    <ul> <!-- Tab选项卡标题框 -->
        <li>Tab1</li>
        <li>Tab2</li>
        <li>Tab3</li>
    </ul>
    <ol> <!-- Tab选项卡内容框 -->
        <li><img src="images/bg2.jpg" width="450" /></li>
        <li><img src="images/bg3.jpg" width="450" /></li>
        <li><img src="images/bg4.jpg" width="450" /></li>
    </ol>
</div>
```

1. JavaScript 实现

Tab 选项卡的功能设计思路是这样的：先使用 CSS 设计 4 对类样式，分别用来控制标题栏和内容框的显隐样式。使用 JavaScript 设计在默认状态下标题栏和内容框的类样式，然后通过遍历方式为每个标题栏绑定 mouseover 事件处理函数，设计当鼠标经过标题栏时，隐藏

所有内容框，修改所有标题的类样式，并显示该标题栏的样式和现实所对应的内容框。

使用 JavaScript 实现 Tab 选项卡功能的完整代码如下。

```
<script type="text/javascript">
//省略了为 Document 对象扩展的 getElementsByClassName()方法，详细讲解请参阅第 6.1.3 节内容
window.onload = function(){
    var tab = document.getElementsByClassName("tab")[0];  //获取 Tab 选项卡的外框
    var ul = tab.getElementsByTagName("ul")[0];  //获取 Tab 选项卡标题栏的外框
    var ol = tab.getElementsByTagName("ol")[0];  //获取 Tab 选项卡内容框的外框
    var uli = ul.getElementsByTagName("li"); //获取所有标题栏选项
    var oli = ol.getElementsByTagName("li"); //获取所有内容选项
    for(var i=0; i<uli.length; i++){     //遍历标题栏选项
        uli[i].className = "normal"; //设置所有标题栏选项的类样式为普通样式
    }
    for(var i=0; i<oli.length; i++){     //遍历内容框选项
        oli[i].className = "none";   //设置所有内容框选项的类样式为隐藏样式
    }
    uli[0].className = "hover"; //设置第一个标题栏选项为凸起显示效果
    oli[0].className = "show"; //显示第一个标题栏对应的内容框选项
    var addEvent=function(e, fn) {  //自定义绑定 mouseover 事件函数
        if(document.addEventListener){    //兼容非 IE 浏览器
            return e.addEventListener("mouseover", fn, false);
        }
        else if(document.attachEvent){    //兼容 IE 浏览器
            return e.attachEvent("onmouseover", fn);
        }
    };
    for(var j = 0; j < uli.length; j ++ ){   //遍历标题栏选项
        (function(j,uli,oli){    //调用匿名函数
            addEvent(uli[j], function(){ //为当前标题栏选项元素绑定 mouseover 事件
                for(var n = 0; n < uli.length; n ++ ){ //遍历标题栏选项
                    uli[n].className = "normal";    //恢复所有标题栏选项为普通显示状态
                    oli[n].className = "none"; //隐藏所有内容框选项
                }
                uli[j].className = "hover";    //设置当前标题栏为凸起效果
                oli[j].className = "show";     //显示当前标题栏对应的内容框选项
            });
        })(j,uli,oli); //把当前序号、标题栏选项数组和内容框选项数组传递进去
    }
}
</script>
```

2. jQuery 实现

根据 JavaScript 的设计思路，使用 jQuery 实现相同的设计效果，编写的代码会非常简洁，如下所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
```

```
$( function(){    //页面初始化事件处理函数
    var $uli = $(".tab ul li");//获取所有标题栏选项元素
    var $oli = $(".tab ol li");//获取所有内容框选项元素
    $uli.addClass("normal");    //为所有标题栏选项元素添加普通类样式
    $oli.addClass("none"); //为所有内容框选项元素添加隐藏类样式
    $uli[0].className = "hover";    //初始化第一个标题栏选项显示为凸起效果
    $oli[0].className = "show";//初始化第一个内容框选项为显示效果
    $uli.each(function(n){ //遍历所有标题栏选项
        $(this).mouseover(function(){    //为每个选项绑定 mouseover 事件处理函数
            $uli.removeClass().addClass("normal");    //移出所有标题栏选项的类样式，恢复普通显示状态
            $(this).removeClass().addClass("hover");    //移出所有类样式，为当前标题栏选项元
素设置凸起显示状态
            $oli.removeClass().addClass("none");//移出所有内容框选项的类样式，恢复隐藏显示状态
            $($oli[n]).removeClass().addClass("show"); //移出所有类样式，为当前内容框选项元
素设置显示状态

        })
    })
});
</script>
```

6.1.6 显隐切换

通过上面示例的演示，我们可以看到，显示和隐藏是一对密不可分的动画形式。隐藏元素之后，一般都需要显示元素，反之亦然。在如此频繁的操作中，需要把 show()方法和 hide()方法与 if 条件结构结合起来，并定义一个显隐指示变量，是不是很麻烦？jQuery 把这种繁琐的操作进行封装，定义了 toggle()方法，下面就来进行详细讲解。

1. jQuery 实现

jQuery 定义的 toggle()方法能够切换元素的可见状态。如果元素是可见的，将会把它切换为隐藏状态；如果元素是隐藏的，则把它切换为可见状态。例如，在下面的示例中，单击按钮可以切换显示或隐藏 div 元素。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript" >
$(function(){//页面初始化函数
    $("input").click(function(){    //为按钮绑定 click 事件处理函数
        $("div").toggle();    //为 div 元素调用 toggle()方法
    })
})
</script>

<input type="button" value="显隐元素" />
<div>DIV 元素</div>
```

toggle()方法还可以接受多个参数。如果传入 true 或者 false 参数值，则可以设置元素显

示或者隐藏，功能类似于 show()和 hide()方法。如果参数值为 true，则功能类似调用 show()方法来显示匹配的元素，如果参数值为 false，则调用 hide()来隐藏匹配的元素。

如果传入一个数值或者一个预定义的字符串，如"slow"、"normal"或者"fast"，则表示在显隐切换时，以指定的速度动态显示匹配的显隐过程。

除了可以指定动画显隐的速度外，还可以为第二个参数指定一个回调函数，以备在动画演示完毕之后，调用该函数，以完成额外的任务。

2. JavaScript 实现

使用 JavaScript 直接定义 toggle()方法，需要借助 getStyle()方法(请参阅 4.11.4 节的讲解)。首先，使用 getStyle()方法获取当前元素的 display 属性值，根据元素的显示属性，决定是显示还是隐藏当前元素。实现的完整代码如下。

```
<script type="text/javascript">
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
//省略了自定义 getStyle()方法，请参阅第 4.11.4 节内容
DOMextend("toggle",function(){ //自定义显隐切换方法
    var _this = this;  //获取当前元素
    var t = false;//定义切换变量
    var d = _this.getStyle("display");     //调用 getStyle()方法，获取当前元素的 display 样式属性
    if(d && d=="none") t = true;     //如果当前元素被隐藏，则设置切换变量值为 true
    if( t ){  //如果切换变量为 true，则显示当前元素
        _this.style.display = "";
        t = false;
    }
    else{ //如果切换变量为 false，则隐藏当前元素
        _this.style.display = "none";
        t = true;
    }
    return _this; //返回当前元素
})
//应用案例
window.onload = function(){     //页面初始化事件函数
    var t = false;
    var input = document.getElementsByTagName("input")[0];     //获取按钮元素
    var div = document.getElementsByTagName("div")[0];     //获取 div 元素
    input.onclick = function(){//注册 click 事件
      div.toggle();    //调用 toggle()方法，显示或隐藏元素
    };
}
</script>

<input type="button" value="显隐元素" />
<div style="display:none;">DIV 元素</div>
```

在上面的示例中，先借助 DOMextend()函数，为每个元素绑定一个自定义 toggle()方法，

该方法能够模拟 jQuery 的 toggle()方法，实现动态切换显示或隐藏当前元素。

6.2 滑动显示和隐藏

动感是现代网页设计中一个重要的时尚元素，动感网页的直接体现就是各种炫目的下拉菜单、幻灯片等。设计动感效果，首先应该理解运动的速度问题。根据物理学基本知识，运动包括匀速运动和变速运动。设计匀速运动效果只需要使用 JavaScript 动态控制均匀移动元素的位置即可，而变速运动就需要掌握一些简单的算法。

jQuery 提供了简单的滑动方法，以及自定义动画函数。借助 jQuery 的这些功能，我们能够完成各种网页动画的设计要求。

6.2.1 jQuery 实现的滑动显隐效果

slideDown()和 slideUp()方法是 jQuery 定义的两个滑动方法，它们分别是向下滑动和向上滑动，相当于缓慢舒展和缓慢收缩元素对象。如果灵活配合 slideDown()和 slideUp()方法，可以设计出很多奇妙、动感的滑动效果。

例如，在下面这个示例中，对于被隐藏的 div 元素，当单击按钮时，将为其调用 slideDown()方法，缓慢舒展显示 div 元素。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
       $("div").slideDown();
    })
})
</script>

<input type="button" value="滑动效果" />
<div style="display:none;">
   <img src="/mysite/images/bg5.jpg" height="200" />
</div>
```

注意：slideDown()方法仅适用于被隐藏的元素，如果为已显示的元素调用 slideDown()方法，是看不到效果的。而 slideUp()方法正好相反，它可以把显示的元素缓慢地隐藏起来。slideDown()和 slideUp()方法正像卷帘，slideDown()方法能够缓慢地展开帘子，而 slideUp()方法能够缓慢地收缩帘子。通俗描述，slideDown()方法作用于隐藏元素，而 slideUp()方法作用于显示元素，两者功能和效果截然相反。

slideDown()和 slideUp()方法可以包含两个可选的参数，第一个参数设置滑动的速度，可以设置预定义字符串，如"slow"、"normal"和"fast"，或者传递一个数值，表示动画时长的毫秒数。第二个可选参数表示一个回调函数，当动画完成之后，将调用该回调函数。

例如，在下面的示例中，当单击按钮时，隐藏的 div 元素会自动调用 slideDown()方法，缓慢展开 div 元素，动画执行完毕后，再调用 slideDown()方法包含的回调函数，在该回调函数中为 div 元素调用 slideUp()方法，最后缓慢地收缩 div 元素。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){//页面初始化函数
    $("input").click(function(){     //绑定 click 事件处理函数
        $("div").slideDown(1000,function(){   //调用 slideDown()方法
            $("div").slideUp("slow",function(){    //在回调函数中调用 slideUp()方法
                alert("div 元素重新被隐藏起来！");
            });
        });
    });
})
</script>

<input type="button" value="滑动效果" />
<div style="display:none;">
    <img src="/mysite/images/bg5.jpg" height="200" />
</div>
```

6.2.2 JavaScript 实现的滑动显示效果

使用 JavaScript 实现滑动效果之前，读者不妨回忆一下：在第 4 章中曾经讲解过 offset()、width()和 height()方法，利用这些方法可以获取当前元素的绝对坐标值和大小。而要实现滑动显示效果，我们就需要调用这些自定义方法，利用它们获取元素的坐标值和尺寸，只有这样才可以设计出滑动效果。具体设计思路如下。

(1) 先使用 getStyle()方法获取元素的 display 样式属性值，判断元素是否为隐藏状态，若不是则返回，不允许执行滑动显示。

(2) 借助 setCSS()和 resetCSS()方法设置元素 display 样式属性值为默认值(即非隐藏)，visibility 属性值为 hidden(即隐藏显示)。这时就可以利用 offset()方法获取元素的坐标值，以及元素的宽和高的尺寸值。

(3) 复制当前元素，并清除该元素包含的子节点，把这个复制的元素作为盒子，将当前元素移动到该盒子中，并删除原元素。

(4) 设置盒子元素的 overflow 样式属性为 hidden，display 为显示状态，宽度等于元素的宽度，高度为 0，这样就可以通过逐步增大盒子元素的高度，来逐步显示盒子包含的元素，

从而设计出元素逐步滑动显示的效果。

(5) 当元素完全显示之后，则清除循环执行命令，并把这个临时的盒子元素删除掉，同时移出该盒子包含的当前元素，把它移到与盒子相邻的位置，这样删除了临时盒子元素之后，又恢复了文档最初的结构。

(6) 最后，调用回调函数，整个动画演示过程结束，演示效果 6.5 所示。

图 6.5 自定义 JavaScript 滑动方法

整个方法的代码如下。

```
<script type="text/javascript">
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
//省略了自定义 getStyle()方法，请参阅第 4.11.4 节内容
//省略了自定义 offset()方法，请参阅第 4.11.2 节内容
//省略了自定义 height()和 width()方法，请参阅第 4.11.5 节内容
DOMextend("slideDown",function(time, fn){    //自定义向下滑动方法，该方法包含两个参数，第一个参数 time 表示滑动的时间，以毫秒为单位，第二个参数 fn 表示滑动结束之后触发的回调函数
    var _this = this;
    var isShow = _this.getStyle("display"); //获取当前元素的 display 属性值
    if(isShow != "none") , //如果元素非隐藏显示，则返回不执行下面的代码
        return;
    //备份当前元素的 display 和 visibility 样式，并定义 display 为显示，而 visibility 为不可见
    var oldcss = _this.setCSS({
        display : "",
        visibility : "hidden"
    })
    var x = _this.offset().left;     //获取当前元素的 x 轴偏移坐标值
    var y = _this.offset().top;      //获取当前元素的 y 轴偏移坐标值
    var height = _this.height();     //获取当前元素的高度
    var width = _this.width();       //获取当前元素的宽度
    //恢复元素的 display 和 visibility 默认样式
    _this.resetCSS(oldcss);
    _this.style.display = "";   //设置 display 属性值为默认值，即显示元素
    //自定义盒子元素，用来包裹当前元素
    var box = _this.cloneNode(true);     //克隆当前元素
    for(var i=0; i < box.childNodes.length; i++){     //清除克隆元素包含的所有子节点
        box.removeChild( box.childNodes[i]);
```

```
}
_this.parentNode.insertBefore(box, _this);    //把克隆元素插入到当前元素的前面
_this = _this.parentNode.removeChild(_this); //移出当前元素
box.appendChild(_this);      //把当前元素嵌入到盒子元素中
//定义盒子元素的 overflow 属性值为 hidden，这样就可以避免显示当前元素
box.style.overflow = "hidden";
box.style.display = "";      //设置盒子元素的 display 属性值为显示
box.style.height = 0;  //设置盒子元素的高度为 0
box.style.width = width;     //设置盒子元素的宽度为当前元素的宽度
//定义间隔调用函数
var step = 5; //默认设置每隔 5 毫秒调用一次函数
var stepheight = step*height/time;  //每次调用函数时，设置盒子高度增长值
var curstep = 0;   //默认设置起步值为 0
//周期调用函数
var interval = setInterval(function(){   //设计定时器
    if( curstep  >= height){//如果当盒子高度超过元素的高度时，停止循环执行函数
        clearInterval(interval); //清除循环调用函数
        _this = _this.parentNode.removeChild(_this);    // 移出当前元素
        box.parentNode.insertBefore(_this, box);   //并把它插入到盒子元素前面
        box.parentNode.removeChild(box);           //移出盒子元素
        fn();   //调用回调函数
    }else{ //如果盒子高度没有超过元素高度，则循环执行下面语句
        curstep += stepheight;    //递加高度
        box.style.height  =  curstep + "px";//设置盒子高度，通过逐步增加高度显示出当前元素
    }
},step);
})
</script>
```

下面就来利用自定义的 slideDown()方法设计一个向下滑动的动画效果，代码如下，演示效果与图 6.5 所示相同。

```
<script type="text/javascript">
window.onload = function(){
   var input = document.getElementsByTagName("input")[0];
    var div = document.getElementsByTagName("div")[0];
    input.onclick = function(){
        div.slideDown(500,function(){     //调用 slideDown()方法
            alert("ok");
        });
    };
}
</script>

<input type="button" value="显隐元素" />
<div style="display:none; ">
    <img src="/mysite/images/bg5.jpg" height="200" />
</div>
```

6.2.3 JavaScript 实现的滑动隐藏效果

同理，如果自定义 slideUp()方法，可以反向思维，先定义盒子高度等于元素高度，然后逐步收缩盒子高度，以动态遮盖元素的显示，从而模拟出元素向上缓慢收缩的动态效果。最后，当盒子高度收缩为 0 时，则隐藏当前元素，并删除盒子元素即可。实现的完整代码如下。

```
<script type="text/javascript">
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
//省略了自定义 getStyle()方法，请参阅第 4.11.4 节内容
//省略了自定义 offset()方法，请参阅第 4.11.2 节内容
//省略了自定义 height()和 width()方法，请参阅第 4.11.5 节内容
DOMextend("slideUp",function(time, fn){      //自定义向上滑动方法，该方法包含两个参数，第一个参数 time 表示滑动的时间，以毫秒为单位，第二个参数 fn 表示滑动结束之后触发的回调函数
    var _this = this;
    var isShow = _this.getStyle("display"); //获取当前元素的 display 属性值
    if(isShow == "none")    //如果当前元素隐藏显示，则返回不执行下面代码
        return;
    var x = _this.offset().left;     //获取当前元素的 x 轴偏移坐标值
    var y = _this.offset().top;      //获取当前元素的 y 轴偏移坐标值
    var height = _this.height();     //获取当前元素的高度
    var width = _this.width(); //获取当前元素的宽度
    //自定义盒子元素，用来包裹当前元素
    var box = _this.cloneNode(true);     //克隆当前元素
    for(var i=0; i < box.childNodes.length; i++){     //清除克隆元素包含的所有子节点
        box.removeChild( box.childNodes[i]);
    }
    _this.parentNode.insertBefore(box, _this);    //把克隆元素插入到当前元素的前面
    _this = _this.parentNode.removeChild(_this); //移出当前元素
    box.appendChild(_this);      //把当前元素嵌入到盒子元素中
    //定义盒子元素的 overflow 属性值为 hidden，这样就可以避免显示当前元素
    box.style.overflow = "hidden";
    box.style.display = "";      //设置盒子元素的 display 属性值为显示
    box.style.height = height; //设置盒子的高度为当前元素的高度
    box.style.width = width;    //设置盒子的宽度为当前元素的宽度
    //定义间隔调用函数
    var step = 5; //默认设置每隔 5 毫秒调用一次函数
    var stepheight = step*height/time;  //每次调用函数时，盒子高度的增长值
    var curstep = height;  //设置起步值为元素的默认高度
    //周期调用函数
    var interval = setInterval(function(){   //设计定时器
        if( curstep <= 0){  //如果当盒子高度小于等于 0 时，停止循环执行函数
            clearInterval(interval); //清除循环调用函数
            _this.style.display = "none"; //隐藏当前元素
            _this = _this.parentNode.removeChild(_this);    // 移出当前元素
            box.parentNode.insertBefore(_this, box);   //并把它插入到盒子元素前面
            box.parentNode.removeChild(box);  //移出盒子元素
            fn();   //调用回调函数
        }else{ //如果盒子高度没有为 0，则循环执行下面语句
```

```
            curstep -= stepheight;    //递减高度
            box.style.height  =  curstep + "px";//设置盒子高度，通过逐步减去高度隐藏当前元素
        }
    },step);
})
</script>
```

同样，调用自定义的 sildeUp()方法可以把显示的元素向上滑动隐藏起来。例如，下面的示例将逐步向上滑动隐藏 div 元素。

```
<script type="text/javascript">
window.onload = function(){
    var input = document.getElementsByTagName("input")[0];
    var div = document.getElementsByTagName("div")[0];
    input.onclick = function(){
        div.slideUp(500,function(){ //调用 slideUp()方法
            alert("ok");
        });
    };
}
</script>

<input type="button" value="显隐元素" />
<div>
    <img src="/mysite/images/bg5.jpg" height="200" />
</div>
```

6.2.4 jQuery 设计的滑动显隐切换

与 toggle()方法的功能相似，jQuery 为滑动效果也设计了一个切换滑动的方法—slideToggle()方法。slideToggle()方法的用法与 slideDown()和 slideUp()方法的用法相同，但是它综合了 slideDown()和 slideUp()方法的动画效果，可以在滑动中切换显示或隐藏元素。

slideToggle()方法也包含两个参数，第一个参数表示设置滑动时间，第二个参数表示滑动后调用的回调函数，实现代码如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
        $("div").slideToggle(); //滑动显示或隐藏元素
    })
})
</script>

<input type="button" value="滑动效果" />
<div style="display:none;"><img src="/mysite/images/bg5.jpg" height="200" /></div>
```

6.3 渐隐和渐显

渐隐和渐显是通过不透明度的变化来实现匹配元素的淡入和淡出效果。与滑动显示和隐藏相比，淡入和淡出动画只调整元素的不透明度，也就是说元素的高度和宽度不会发生变化。因此这类动画的效果比较含蓄，但是能够给页面添加无穷的魅力。

jQuery 为元素的渐隐和渐显定义了 3 个方法：fadeIn()、fadeOut()和 fadeTo()方法。本节将具体讲述这 3 个方法，并将讲解如何使用 JavaScript 直接实现渐隐和渐显的动画效果。

6.3.1 jQuery 实现的渐隐渐显效果

fadeIn()方法能够实现所有匹配元素的淡入效果，并在动画完成后可选地触发一个回调函数。而 fadeOut()方法正好相反，它能够实现所有匹配元素的淡出效果。

例如，下面的示例设置当单击按钮时，使隐藏的 div 元素逐渐显示出来，这个逐渐显示的过程持续了 2000 毫秒。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
       $("div").fadeIn(2000);
    })
})
</script>

<input type="button" value="渐隐渐显效果" />
<div style="display:none;"><img src="images/bg5.jpg" height="200" /></div>
```

fadeIn()和 fadeOut()方法与 slideDown()和 slideUp()方法的用法是完全相同的，它们都可以包含两个可选参数。第一个参数表示动画持续的时间，以毫秒为单位，另外还可以使用预定义字符串"slow"、"normal"和"fast"，使用这些特殊的字符串可以设置动画以慢速、正常速度和快速进行演示；第二个参数表示回调函数，该参数为可选参数，用来在动画演示完毕之后被调用。例如，在下面的示例中，当单击按钮之后调用 div 元素的 fadeIn()方法，逐步显示隐藏的元素，当显示完成之后，再调用回调函数，把 div 元素逐渐隐藏起来。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){     //绑定 click 事件，渐显 div 元素
       $("div").fadeIn(2000,function(){  //回调函数
           $("div").fadeOut(2000);       //渐隐 div 元素
```

```
        });
    })
})
</script>

<input type="button" value="渐隐渐显效果" />
<div style="display:none;"><img src="images/bg5.jpg" height="200" /></div>
```

注意：与 slideDown()和 slideUp()方法的用法相同，fadeIn()方法只能够作用于被隐藏的元素，而 fadeOut()方法只能够作用于显示的元素。

fadeTo()方法能够把所有匹配元素的不透明度以渐进方式调整到指定的不透明度，并在动画完成后可选地触发一个回调函数。该方法包含三个参数：第一个参数用来指定动画演示的时间；第二个参数表示元素要调整到的不透明度，取值范围在 0～1 之间；第三个参数为可选的回调函数。

例如，下面的示例将把图像逐步调整到不透明度为 0.4 的显示效果，显示效果如图 6.6 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
      $("div").fadeTo(2000,0.4);
    })
})
</script>

<input type="button" value="渐隐渐显效果" />
<div><img src="images/bg5.jpg" height="200" /></div>
```

图 6.6　使用 jQuery 的 fadeTo()方法实现的半透明效果

注意：fadeTo()方法仅能够作用于显示的元素，对于被隐藏的元素来说是无效的。

6.3.2 JavaScript 实现的渐显效果

实现渐隐渐显效果的技术核心是各个浏览器支持的不透明度特效，但是由于不同浏览器支持的不透明度特效方法不同，导致我们在设计渐隐渐显效果时，首先应该考虑浏览器的兼容性问题。

IE 浏览器支持 filters 滤镜集，而非 IE 浏览器支持 style.opacity 属性。另外 IE 的 opacity 属性值范围为 0~100，其中 0 表示完全透明，100 表示不透明。而 style.opacity 属性的取值范围是 0~1，其中 0 表示完全透明，1 表示不透明。为了兼容不同的浏览器，我们要先为元素扩展一个方法，用来设置元素的不透明度。

```
<script type="text/javascript">
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
DOMextend("setOpacity",function(n){//设置元素不透明度，参数 n 表示不透明度，取值范围为 0 到 100
    var _this = this
    var n = parseFloat(n); //把参数值转换为浮点数
    if(n && (n>100) || !n) //如果参数值大于 100，则设置其值为 100
        n=100;
    if(n && (n<0))      //如果参数值小于 0，则设置其值为 0
        n =0;
    if (_this.filters){     //兼容 IE 浏览器
        _this.style.filter = "alpha(opacity=" + n + ")";
    }
    else{ //兼容非 IE 浏览器
        _this.style.opacity = n / 100;//在非 IE 浏览器内设置它的不透明度
    }
})
</script>
```

自定义 setOpacity()方法能够为元素设置一个不透明度并显示出来。有了这个方法，我们就可以很方便地模仿 jQuery 的 fadeIn()和 fadeOut()方法。

其中使用 JavaScript 直接定义的 fadeIn()方法的完整代码如下。

```
<script type="text/javascript">
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
//省略了自定义 getStyle()方法，请参阅第 4.11.4 节内容
//省略了自定义 setOpacity()方法，请参阅第 6.3.2 节内容
DOMextend("fadeIn",function(time, fn){  //自定义逐渐显示的动画方法
    var _this = this;
    var isShow = _this.getStyle("display");  //调用 getStyle()方法，判断当前元素是否隐藏
    if(isShow != "none")    //如果没有隐藏则返回
        return;
    var t = 5;      //设置定时器的步长为 5 毫秒
```

```
var step = t*100/time; //计算定时器每步增加的不透明度值
var i = 0;     //不透明度初始值
//为当前元素包裹一个克隆元素，以防止元素在演示过程中突然闪现的问题
var box = _this.cloneNode(true);     //克隆当前元素
for(var i=0; i < box.childNodes.length; i++){     //清除所有子节点
    box.removeChild( box.childNodes[i]);
}
_this.parentNode.insertBefore(box, _this);    //把克隆元素插入到当前元素前面
_this = _this.parentNode.removeChild(_this); //删除当前元素
box.appendChild(_this);     //把当前元素添加到盒子元素中
box.style.display = "";     //显示盒子元素
box.setOpacity(0); //设置盒子元素不透明度为 0
box.style.visibility = "hidden";     //设置盒子的元素不可见
_this.style.display = "";   //显示当前元素，由于盒子的元素不显示，故也看不到
var interval = setInterval(function(){   //设计定时器
    box.style.visibility = "visible";     //设置盒子元素可见
    box.setOpacity(i);  //设置盒子元素的不透明度
    i += step ;          //递加步长
    if(i >= 100){   //如果步长大于或等于 100，则退出定时器
        clearTimeout(interval);   //清除定时器
        //删除临时包裹的盒子元素，恢复元素当初的位置和结构
        _this = _this.parentNode.removeChild(_this);    //移出当前元素
        box.parentNode.insertBefore(_this, box);   //把当前元素插入到克隆盒子的前面
        box.parentNode.removeChild(box);  //移出克隆的盒子
        if(fn)  //如果存在回调函数参数
            fn();     //则调用该回调函数
    }
    }, t);
})
```

下面我们就应用这个使用 JavaScript 直接定义的 fadeIn()方法，示例代码如下。

```
<script type="text/javascript">
window.onload = function(){
    var input = document.getElementsByTagName("input")[0];
    var div = document.getElementsByTagName("div")[0];
    input.onclick = function(){
        div.fadeIn(2000,function(){
            alert("ok");
        });
    };
}
</script>

<input type="button" value="显隐元素" />
<div style="display:none;"><img src="/mysite/images/bg5.jpg" height="200" /></div>
```

上面示例的显示效果与 jQuery 定义的 fadeIn()方法实现的效果完全相同，当单击按钮时，图片逐步显现，演示效果不再显示。

6.3.3 JavaScript 实现的渐隐效果

相对而言，使用 JavaScript 直接定义 fadeOut()方法要简单一些，因为它不需要考虑元素的闪现问题，元素在最初状态是显示的，当逐步淡出之后，再设置元素隐藏显示即可。

使用 JavaScript 直接实现 jQuery 的 fadeOut()方法代码如下。

```
<script type="text/javascript">
//省略了自定义 DOMextend()函数，请参阅第 4.3.3 节内容
//省略了自定义 getStyle()方法，请参阅第 4.11.4 节内容
//省略了自定义 setOpacity()方法，请参阅第 6.3.2 节内容
DOMextend("fadeOut",function(time, fn){ //自定义逐渐隐藏的动画方法
    var _this = this;
    var isShow = _this.getStyle("display");
    if(isShow == "none")    //如果元素隐藏显示，则返回
        return;
    var t = 5;     //设置定时器的间隔时间为 5 毫秒
    var step = t*100/time; //计算步长
    var i = 100;   //定义不透明度初始值为 100
    var interval = setInterval(function(){      //定义定时器
        _this.setOpacity(i);     //设置当前元素的不透明度
        i -= step ;     //递减不透明度值
        if(i <= 0 ){    //如果不透明度值小于等于 0，则退出定时器
            clearTimeout(interval);  //清除定时器
            _this.style.display = "none"; //隐藏当前元素
            if(fn){}     //如果存在回调函数参数，则调用该回调函数
                fn();
        }
    }, t);
})
```

下面再使用 JavaScript 直接定义的 fadeOut()方法淡出元素，示例代码如下。

```
<script type="text/javascript">
window.onload = function(){
    var input = document.getElementsByTagName("input")[0];
    var div = document.getElementsByTagName("div")[0];
    input.onclick = function(){
        div.fadeOut(500,function(){
            alert("ok");
        });
    };
}
</script>

<input type="button" value="显隐元素" />
<div><img src="/mysite/images/bg5.jpg" height="200" /></div>
```

6.4 自定义动画

对于 Web 设计来说，动画形式主要包括三种：位置变化、形状变化和显示变化。位置变化主要通过元素的坐标值来控制。对于网页元素来说，形状变化主要是大小变化，这种形式主要依靠宽和高进行控制。而显示变化主要通过显示和隐藏属性进行控制，在前面几节中已经进行了详细讲解。

jQuery 自定义了 animate()方法，这个方法功能强大，用法灵活，可以自定义各种形式的动画效果。本节将重点讲解 animate()方法的使用，同时探索如何使用 JavaScript 直接定义各种类型的运动动画效果。

6.4.1 jQuery 自定义动画

jQuery 定义的 animate()方法可以用于创建自定义动画。该方法的关键就在于指定动画的形式，以及动画结果样式属性的对象。例如，设计当单击按钮时，图像的大小被放大到原始大小，实现代码如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
        $("img").animate({
            width: "100%",
            height: "100%"
        }, 1000 );
    })
})
</script>

<input type="button" value="自定义动画" />
<div><img src="images/bg5.jpg" height="200" /></div>
```

animate()方法包含 4 个参数：第一个参数是一组包含作为动画属性和终值的样式属性和及其值的集合。形式类似下面代码。

```
{
    width: "90%",
    height: "100%",
    fontSize: "10em",
    borderWidth: 10
}
```

这个集合对象中每个属性都表示一个可以变化的样式属性，如 height、top 和 opacity 等。注意，所有指定的属性必须采用驼峰命名形式，如 marginLeft，而不是 margin-left。属性的值表示这个样式属性到多少时动画结束。

如果属性值是一个数值，样式属性就会从当前的值渐变到指定的值。如果属性值是“hide”、“show”或“toggle”等特定字符串值，则会为该属性调用默认的动画形式。例如，下面的示例在一个动画中同时应用 4 种类型的效果，放大文本大小，扩大元素宽和高，同时多次单击，可以在高度和不透明度之间来回切换显示 div 元素。当然，读者可以添加更多的动画样式，以设计复杂的动态效果。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
        $("div").animate({
            width: "200%",
            height: "200%",
            fontSize: "5em",
            height: 'toggle',
            opacity: 'toggle'
        }, 1000 );
    })
})
</script>

<input type="button" value="自定义动画" />
<div>自定义动画</div>
```

第二个参数表示动画持续的时间，以毫秒为单位，也可以设置预定义字符串，如"slow"、"normal"和"fast"。在 jQuery 1.3 版本中，如果第二个参数设置为 0，则表示直接完成动画，而在以前的版本中则会执行默认动画。

第三个参数表示要使用的擦除效果的名称，这是一个可选参数，要使用该参数，需要插件支持。默认 jQuery 提供"linear" 和"swing"特效。

第四个参数表示回调函数，表示在动画演示完毕之后，将要调用的函数。

例如，下面的示例可以使 div 向左右平滑移动，代码如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").eq(0).click(function(){
        $("div").animate({
            left: "-100px"
        }, 1000)
    })
    $("input").eq(1).click(function(){
```

```
        $("div").animate({
            left: "+100px"
        }, 1000)
    })
})
</script>

<input type="button" value="向左运动" /><input type="button" value="向右运动" />
<div style="position:absolute;left:200px; border:solid 1px red;">自定义动画</div>
```

注意，要想使 div 元素能够自由移动，必须设置它的定位方式为绝对定位、相对定位或者固定定位，如果是静态定位，则移动动画是无效的。

同时，移动的动画总是在默认位置作为参照物的。例如，在上面的示例中，已经定义 div 元素 left:200px，如果在 animate()方法中设置 left: "+100px"，则 div 元素并不是向右移动，而是向左移动 100 像素。对于 left: "-100px"移动动画来说，则会在现在固定位置基础上，向左移动 300 像素，如图 6.7 所示。

默认定位在 200px 位置(参考点是窗口左边框)　　向右移动 100px(参考点是窗口左边框)　　向左移动 100px

图 6.7　使用 jQuery 的 fadeTo()方法实现的半透明效果

animate()方法的功能是很强大的，我们可以把第二个及其后面的所有参数都放置在一个对象中，在这个集合对象中包含动画选项的值，然后把这个对象作为第二个参数传递给 animate()方法。该参数可以包含下面多个选项。

- duration：该选项指定动画演示的持续时间，与在 animate()方法中直接传递时间的作用是相同的。duration 选项也可以包含 3 个预定义的字符串，如"slow"、"normal"和"fast"。
- easing：该选项接受要使用的擦除效果的名称，需要插件支持，默认值为"swing"。
- complete：该选项指定动画完成时执行的函数。
- step：该选项表示动画演示之后的回调值。
- queue：该选项表示是否将使此动画不进入动画队列，默认值为 true。

例如，在下面的示例中设置了一个动画队列，其中设置第 1 个动画不在队列中运行，此时可以看到第 1 个动画的字体变大和第 2 个动画的元素高度增加是同步进行的。当这两个动画同步进行完成之后，然后才触发第 3 个动画。在第 3 个动画中，设置 div 元素的最终不透

明度为 0，则经过 2000 毫秒的淡出演示过程之后，该 div 元素消失。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
        $("div").animate(   //第 1 个动画
            {height:"120%"},
            {duration: 5000, queue: false}
        ).animate({     //第 2 个动画，将与第 1 个动画并列进行
            fontSize: "10em"
        },1000).animate({   //第 3 个动画
            opacity: 0
        }, 2000);
    })
})
</script>

<input type="button" value="自定义动画" />
<div style="border:solid 1px red;">自定义动画</div>
```

6.4.2 使用 jQuery 停止动画

jQuery 定义了 stop()方法，该方法可以随时停止所有在指定元素上正在运行的动画。例如，在下面的示例中，单击第 1 个按钮可以运行动画，随时单击第 2 个按钮可以停止动画的演示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    $("input").eq(0).click(function(){
        $("div").animate({
            fontSize : "10em"
        }, 8000);
    });
    $("input").eq(1).click(function(){
        $("div").stop();
    })
})
</script>

<input type="button" value="自定义动画" /><input type="button" value="停止动画" />
<div style="border:solid 1px red;">自定义动画</div>
```

stop()方法包含两个可选的参数，第一个参数表示布尔值，如果设置为 true，则清空队列，立即结束所有动画；如果设置为 false，则如果动画队列中有等待执行的动画，会立即执行队列后面的动画。

第二个参数也是一个布尔值，如果设置为 true，则会让当前正在执行的动画立即完成，并且重设 show 和 hide 的原始样式，调用回调函数等。

6.4.3 使用 jQuery 关闭动画

jQuery 除定义了 stop()方法外，还定义了 off 属性，如果把这个属性设置为 true，可以立即关闭所有动画，所有效果会立即执行完毕。例如，在下面的示例中，首先调用 jQuery.fx 空间下的属性 off，设置该属性值为 true，即关闭当前页面中所有的 jQuery 动画，因此下面按钮所绑定的 jQuery 动画也是无效的，当单击按钮时，会直接显示 animate()方法的第一个参数设置的最终样式效果。

```
<script src="images/jquery-1.3.2.js" type="text/javascript"></script>
<script type="text/javascript">
$(function(){
    jQuery.fx.off = true;
    $("input").click(function(){
        $("div").animate({
            fontSize : "10em"
        }, 8000);
    });
})
</script>
<input type="button" value="自定义动画" />
<div style="border:solid 1px red;">自定义动画</div>
```

关闭 jQuery 动画，对于配置比较低的电脑，或者当遇到了可访问性问题时，是非常有帮助的。如果要重新开启所有动画，只需要设置 jQuery.fx.off 属性值为 false 即可。

6.4.4 使用 JavaScript 实现滚动动画

滚动动画的设计方法有很多种，最简单的方法是通过元素的坐标进行移动，即通过动态改变 left 和 top 样式值实现控制。本例将介绍另一种方法：动态改变滚动轴的大小，从而设计出动态移动的滚动效果，演示效果如图 6.8 所示。

首先，设计 HTML 结构。在这个滚动动画模块中，主要由三层结构嵌套而成，第一层是最外围的 wrap 层，该层负责设计模块的样式(如边框)，以及控制按钮定位，因此该层结构主要使用了定位技术来实现。

第二层是 box 结构层，该层负责设计滚动动画模块的大小，该框宽度应该小于内部包含的内容宽度，这样才能够使 box 出现滚动条，然后通过 overflow:hidden;样式声明，隐藏该盒子的滚动条。下面就可以利用 JavaScript 动态控制滚动条的偏移距离进而实现滚动效果。

图 6.8 JavaScript 实现的滚动动画效果

第三层包括类名为 allwidth 的 div 层和 scroll 层，其中类名为 allwidth 的 div 层的宽度预设为 2000px，从而强迫 box 层出现滚动条。scroll 层作为内容包含框而存在。

在内容包含框中设计所有超链接为浮动显示，这样就可以实现并行显示。考虑到篇幅，本模块的 CSS 样式就不再显示，读者可以参阅本书光盘示例。

```
<div id="wrap">
    <span id="left">向左滚动</span>
    <div id="box">
        <div class="allwidth">
            <div id="scroll">
                <a style="float:left" href="?#1"> <img src="images/bg3.jpg" width="150px"
height="150px;"/> </a>
                <a style="float:left" href="?#2"> <img src="images/bg4.jpg" width="150px"
height="150px;"/> </a>
                <a style="float:left" href="?#3"> <img src="images/bg5.jpg" width="150px"
height="150px;"/> </a>
                <a style="float:left" href="?#4"> <img src="images/bg6.jpg" width="150px"
height="150px;"/> </a>
                <a style="float:left" href="?#5"> <img src="images/bg7.jpg" width="150px"
height="150px;"/> </a>
            </div>
        </div>
    </div>
    <span id="right">向右滚动</span>
</div>
```

完成结构和样式设计之后，下面我们重点讲解如何使用 JavaScript 来实现滚动动画功能。首先，定义两个功能函数，其中$()函数负责选择指定 ID 名的元素，addEventHandler()函数负责为指定对象绑定事件处理函数。详细代码如下。

```
//获取指定 ID 元素
var $ = function (id) {
    return "string" == typeof id ? document.getElementById(id) : id;    //如果指定参数
值为字符串，则获取该 ID 名的元素，否则直接返回该对象
```

```
};
//为特定对象绑定事件处理函数
//参数说明：oTarget 表示绑定元素，sEventType 表示事件类型，fnHandler 表示事件处理函数
var addEventHandler = function (oTarget, sEventType, fnHandler) {
    if (oTarget.addEventListener) { //兼容 DOM 2 事件处理模型
        oTarget.addEventListener(sEventType, fnHandler, false);
    } else if (oTarget.attachEvent) {    //兼容 IE 事件模式
        oTarget.attachEvent("on" + sEventType, fnHandler);
    } else {  //兼容 DOM 0 级事件模型
        oTarget["on" + sEventType] = fnHandler;
    }
};
```

下面这个函数是一个类结构，首先在该函数中通过 apply()方法调用类的原型方法 init()，这样在类实例化过程中，init()方法会自动被调用，并绑定到当前实例上，从而进行初始化设置。在初始化过程中，对用户传递的 5 个参数进行设置，然后调用原型方法 scrolling()进行滚动显示。

```
//定义类型函数，在类型函数中直接调用该类型的原型方法 init()
var Scroller = function(){
    this.init.apply(this, arguments);    //在调用原型方法时，会把类型函数的参数传递给它
}
//定义类型的原型方法
Scroller.prototype = {
    init: function(box, scroller, left, right, options) { //初始化方法，参数分别表示滚动模
块的外框，滚动模块的内框，以及左右控制按钮和参数对象
        var _this = this, box = $(box), scroller = $(scroller);
        this.boxWidth = box.offsetWidth; //获取外框宽度
        this.scroWidth = scroller.offsetWidth;    //获取内框宽度
        if(this.scroWidth <= this.boxWidth)  //如果内框宽度小于外框宽度，则返回
            return;
        box.style.overflow = "hidden";   //设置外框样式
        scroller.appendChild(scroller.cloneNode(true));    //复制并添加滚动内容
        this.box = box;
        this.scroll = true;
        this.setOptions(options);    //调用 setOptions()方法初始化参数选项
        this.side = 1; //设置滚动方向，值为 1 时表示向左，为-1 时表示向右
        switch (this.options.Side) {
            case "right" :
                this.side = -1;
                break;
            case "left" :
            default :
                this.side = 1;
        }
        //为控制按钮和内框绑定事件处理函数
        addEventHandler(scroller , "mouseover", function() { _this.scroll = false; });
        addEventHandler(scroller , "mouseout", function() { _this.scroll = true; });
        if(left) { addEventHandler($(left), "click", function() { _this.side = 1; }); }
```

```
        if(right) { addEventHandler($(right), "click", function() { _this.side = -1; }); }
        this.scrolling();
    },
    //设置默认属性
    setOptions: function(options) {
        this.options = {    //默认值
            Step: 1,    //每次变化的 px 量
            Time: 10,    //速度
            Side: "left"    //默认滚动方向
        };
        var options = options || {};
        for (var property in options) {
            this.options[property] = options[property];
        }
    },
    //滚动处理函数
    scrolling: function() {
        if (this.scroll) {  //如果允许滚动
            var iscroll = this.box.scrollLeft, thisWidth = this.scroWidth;
            if(this.side > 0){   //向左滚动
                if(iscroll >= (thisWidth * 2 - this.boxWidth)){ iscroll -= thisWidth; }
            } else {    //向右滚动
                if(iscroll <= 0){ iscroll += thisWidth; }
            }
            this.box.scrollLeft = iscroll + this.options.Step * this.side;   //
scrollLeft 超过 1400 会自动变回 1400，注意长度
        }
        var _this = this; window.setTimeout(function(){   //设置定时器
            _this.scrolling();  //调用滚动函数
        }, this.options.Time);
    }
};
```

最后，在页面初始化事件处理函数中，将该类型函数实例化，代码如下。

```
window.onload = function(){
    new Scroller("box","scroll", "left", "right");
}
```

第7章 Ajax异步通信高效实践

Ajax 是 Asynchronous JavaScript and XML 的缩写，中文翻译为“异步 JavaScript 和 XML”。它不是原创技术，而是利用 JavaScript 语言和 XML 数据实现客户端与服务器端进行异步通信的一种方法。Ajax 主要用到下面几种技术实现异步通信。

- 基于标准的 XHTML 结构和 CSS 样式。
- 通过 DOM(Document Object Model)实现动态显示和交互。
- 通过 XML 和 XSLT 进行数据交换和处理。
- 使用 XMLHttpRequest 插件进行异步通信。
- 使用 JavaScript 实施逻辑控制，以便整合以上所有的技术。

7.1 Ajax 应用准备

jQuery 对 Ajax 应用进行了完全封装，并提供了众多简便、易用的方法，读者不再需要从零起步书写所有代码，也不需要为浏览器兼容问题而焦头烂额。当然，为了更加深入理解 Ajax 应用的技术背景，本章仍然结合 jQuery 和 JavaScript 两种技术实现的途径，让读者更加全面地认识异步通信技术的实质。

7.1.1 Ajax 应用利弊分析

任何技术和概念的诞生，都存在一定的必然性，也符合一定的社会发展要求。Ajax 概念也是如此，首先它给 Web 开发提供了方便，并带来了很多益处。

1. 获得了主流浏览器的主持

这使得异步通信技术获得快速发展，并能够普及到 Web 应用的每个角落。由于浏览器都内置了 Ajax 组件，这样用户就不用担心所开发的 Web 应用是否被支持，只要浏览器允许执行 JavaScript 程序即可。

2. 提升了用户体验

Ajax 概念的提出就是为了解决传统通信给用户访问带来的不便，优秀的用户体验正是 Ajax 获得广大开发者和用户认可的根本原因。浏览者不用刷新页面就可以快速更新页面显示信息，这在传统 Web 开发中是很难实现的。正是因为异步通信技术的普及，才诞生了众多 Web 2.0 类型的网络应用。

3. 提高了 Web 应用的性能

与传统的页面刷新请求不同，Ajax 实现局部数据请求和更新功能，使得客户端与服务器端交互的数据量大大降低，节省了大量带宽，同时请求响应的速度也变得更加迅速。

当然，由于主流浏览器对于异步通信技术支持标准的各异，给开发带来了诸多不便，当然使用 jQuery 技术框架，这个问题可以忽略不计。由于异步通信把所有交互都集成到一个页面中实现，因此也会带来很多潜在的问题，这些问题对于传统应用习惯来说，会感到很不方便。例如：

- 浏览器提供的历史记录功能可能会变得无效，前进和后退按钮无法翻阅以前浏览的信息。
- Ajax 对于搜索引擎来说是不友好的，用户都希望页面能够很好地被搜索引擎抓取，但是这种完全脚本化的页面呈现的不友好姿态，是很难被搜索引擎优先索引的。当

然，这个问题会随着异步通信技术的进一步普及，以及各大搜索引擎技术改进的发展，将会变得不是主要问题。

- 异步通信完全是以一种潜在方式实现的，如何能够更好地跟踪和调试也是一个不小的挑战。当然，随着这类开发工具的不断推出，这个问题也将得到解决。

7.1.2 安装虚拟服务器

如果希望学习和研究 Ajax 应用，读者应该在本地计算机中安装虚拟的服务器，因为异步通信技术必须在客户端和服务器端才能够实现。下面我们介绍 ASP 和 PHP 虚拟服务器的安装方法。同时，本章以及全书的 Ajax 应用都将在 ASP 服务器技术环境下进行讲解。

1. 安装 PHP 虚拟服务器

要安装 PHP 虚拟服务器，可以使用 AppServ 工具包，下载地址为 http://www.appservnetwork.com。AppServ 工具包包含 Apache、Apache Monitor、PHP、MySQL、PHP-Nuk 和 phpMyAdmin 软件。

AppServ 工具包的安装比较简单，下载该软件后，根据提示轻松单击 Next 按钮，按要求输入网址、电子邮箱和密码等用户信息即可。端口默认为 80，当然这些信息可以在后期进行随时修改。

当 AppServ 工具包安装好之后，就可以把 PHP 文件放置在 AppServ\www \目录下，然后在浏览器地址栏中输入 http://loaclhost 地址，最后在相应的文件夹中选定页面文件，即可运行 Ajax 应用示例。

2. 安装 ASP 虚拟服务器

由于 ASP 虚拟服务器与 Windows 操作系统捆绑在一起，所以读者不需要按住任何软件，只需要在系统控制面板中启动管理工具，然后打开 Internet 信息服务(IIS)管理器即可，在其中定义虚拟站点，或者对 IIS 服务器进行设置。

如果读者在控制面板的工具管理窗口中没有看到 IIS 管理器，则可以单击拆卸程序按钮，在打开的 Windows 组件管理选项中安装 IIS 服务组件即可。

当 IIS 组件包安装好之后，就可以把 ASP 文件放置在 inetpub\wwwroot 目录下，然后在浏览器地址栏中输入 http://loaclhost 地址，最后在相应的文件夹中选定页面文件，即可运行 Ajax 应用示例。

7.2 Ajax 应用的第一个示例

Ajax 应用的核心是 XMLHttpRequest 对象，它是实现异步通信的技术关键。XMLHttpRequest 对象最早是在 IE 5.0 版本浏览器的 ActiveX 组件中被引入的，之后其他主流

浏览器也内置了 XMLHttpRequest 对象，虽然各大浏览器实现的方式不同，但是它们都保留了相同的属性、方法和用法。

7.2.1 jQuery 实现

在系统、深入学习 Ajax 应用之前，我们先看一个简单的示例。设想在客户端单击按钮之后，向服务器端发出一个异步请求，然后把服务器端的响应显示出来。

要实现这样的设想，我们先在虚拟服务器端建立一个服务器文件(test.asp)，在这个文件中输入下面的字符串，如图 7.1 所示。

```
Hello World
```

然后在客户端的静态网页文件中，输入下面的 jQuery 代码。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
        $.get("test.asp", function(data){
            alert("来自服务器端的问候: " + data);
        });
    });
})
</script>

<input type="button" value="向服务器发出异步请求" />
```

然后，打开网页浏览器，在地址栏中输入该文件的地址：

```
http://localhost/第一个示例-jQueryt 实现.html
```

在客户端向服务器端请求浏览静态页面，然后在该文档中单击“向服务器发出异步请求”按钮，则会显示如图 7.2 所示的提示信息。

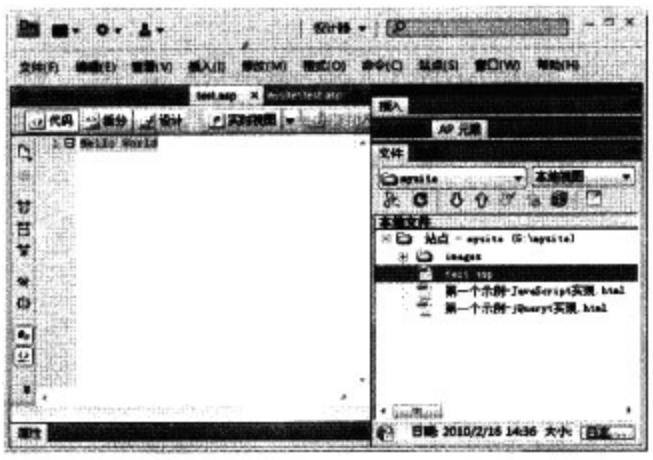


图 7.1 设计 ASP 服务器端文件内容

图 7.2 异步通信后响应效果

在这个示例中，jQuery 的公共方法 get()能够以异步方式向服务器发出一个简单的请求，然后把服务器端的响应信息存储在 get()方法的回调函数参数中，从而完成一次异步通信的过程。在客户端通过读取回调函数的参数，就可以获取服务器端的响应信息。

get()方法可以包含四个参数，第一个参数表示请求的服务器端地址，第二个参数及其后面所有参数都是可选参数，其中第二个参数可以是请求的信息，第三个参数表示回调函数，第四个参数表示响应信息的类型。在上面的示例中，我们简化了 get()方法的使用，仅传递了服务器端的 URL，以及回调函数参数。

7.2.2 JavaScript 实现

若要直接使用 JavaScript 实现异步通信，则应该先定义.XMLHttpRequest 对象，由于不同浏览器定义 XMLHttpRequest 对象的方式不同，因此必须考虑浏览器的兼容性问题。定义 XMLHttpRequest 对象的详细代码如下。

```
<script type="text/javascript">
//定义 XMLHttpRequest 对象
if (window.XMLHttpRequest){     //兼容 Mozilla、Safari 等非 IE 浏览器
    var xmlhttprequest = new XMLHttpRequest();
}
else if (window.ActiveXObject){     //兼容 IE 浏览器
    try{
        var xmlhttprequest = new ActiveXObject("Msxml2.XMLHTTP");
    }
    catch (e){
        try{
            xmlhttprequest = new ActiveXObject("Microsoft.XMLHTTP");
        }
        catch (e)
        { }
    }
}
</script>
```

在上面的代码中，使用条件语句分别判断当前浏览器的类型，并根据不同类型浏览器决定定义对象的方法。对于非 IE 浏览器来说，一般都支持 XMLHttpRequest 对象，可以使用 new XMLHttpRequest()方法定义 XMLHttpRequest 对象。由于 IE 浏览器支持 ActiveXObject 组件实现 Ajax 技术，只有使用 new ActiveXObject("Msxml2.XMLHTTP")或 new ActiveXObject ("Microsoft.XMLHTTP")方法定义 XMLHttpRequest 对象。

在下面示例中，演示了如何使用 XMLHttpRequest 对象如何与服务器通信。

```
<script type="text/javascript">
//省略定义 XMLHttpRequest 对象，请参阅上面代码
window.onload = function(){     //页面初始化处理函数
```

```
    var input = document.getElementsByTagName("input")[0];          //获取按钮
    input.onclick = function(){ //绑定 click 事件类型
        xmlhttprequest.open("GET","test.asp", false); //调用 XMLHttpRequest 对象的 open()
方法，打开与服务器之间的同步通信连接
        xmlhttprequest.send(null);   //向服务器发送请求
        alert("来自服务器端的问候: " + xmlhttprequest.responseText);     //显示服务器的影响信息
    }
}
</script>

<input type="button" value="向服务器发出异步请求" />
```

XMLHttpRequest 对象包含很多方法和属性，虽然不同浏览器的定义方式不同，但是它们都包含相同的方法和属性，且用法和参数基本相似，这为开发人员提供了便利。

在上面的示例中，先调用 open()方法打开一个请求，然后调用 send()方法发送请求的信息，如果服务器响应了请求，则会把响应信息发送给 XMLHttpRequest 对象的 responseText 属性。最后，直接读取 XMLHttpRequest 对象的 responseText 属性值即可。

7.3 从 JavaScript 角度分析 XMLHttpRequest 对象

XMLHttpRequest 对象是 Ajax 异步通信的技术核心，各主流浏览器通过组建嵌入的方式提供支持。要深入理解 Ajax，应该先熟悉 XMLHttpRequest 对象，了解该对象包含的方法和属性，以及它们的基本用法。在此基础上，再理解 jQuery 封装的 Ajax 应用，就会轻松许多。

7.3.1 XMLHttpRequest 对象成员和用法

XMLHttpRequest 对象共包含 8 个基本属性和 6 个基本方法，这些属性和方法负责完成异步通信的全部工作，具体说明如表 7.1 和表 7.2 所示。

表 7.1 XMLHttpRequest 对象属性

属 性	说 明
onreadystatechange	指定当 readyState 属性改变时的事件处理句柄
readyState	返回当前请求的状态
status	返回当前请求的 HTTP 状态码
statusText	返回当前请求的响应行状态
responseBody	返回正文信息
responseStream	以文本流的形式返回响应信息
responseText	以字符串的形式返回响应信息
responseXML	以 XML 数据的形式返回响应信息

表 7.2 XMLHttpRequest 对象方法

方 法	说 明
open()	创建一个新的 HTTP 请求，并指定此请求的方法、URL，以及验证信息(用户名/密码)
send()	发送请求到 HTTP 服务器并接收回应
getAllResponseHeaders()	获取响应的所有 HTTP 头信息
getResponseHeader()	从响应信息中获取指定的 HTTP 头信息
setRequestHeader()	单独指定请求的某个 HTTP 头信息
abort()	取消当前请求

使用 XMLHttpRequest 对象实现异步通信一般需要下面几个步骤。

- 定义 XMLHttpRequest 对象实例。
- 调用 open()方法建立与服务器端的连接。
- 注册 onreadystatechange 事件处理函数，以便接收和处理从服务器端响应的信息。
- 调用 send()方法发送请求。

7.3.2 建立异步连接

定义了 XMLHttpRequest 对象之后，调用 open()方法可以建立异步连接。open()方法的用法如下。

```
xmlhttprequest.open(Method, Url, Async, User, Password);
```

该方法包含 5 个参数，其中前两个参数是必须的。

- xmlhttprequest 表示 XMLHttpRequest 对象实例。
- 参数 Method 表示 HTTP 方法，如 POST、GET、PUT 和 PROPFIND，方法的名称不区分大小写。
- 参数 Url 表示请求的地址，可以是绝对地址，也可以是相对地址。
- 参数 Async 为可选选项，设置是否为异步通信，默认为 true，表示可以异步；而取值为 false 时，表示必须同步通信。
- 参数 User 和 Password 表示请求的文件需要服务器进行验证，如果未指定，当服务器需要验证时，会弹出验证窗口要求进行验证。

例如：

```
xmlhttprequest.open("GET","http:// localhost/mysite/test.asp", false);
```

在上面的代码中，利用 open()方法为 XMLHttpRequest 对象建立了一个与 test.asp 文件的

连接。此时 open()方法包含以下 3 个参数。

- 第一个参数设置发送 HTTP 请求为 GET 方法，即以查询字符串的形式传输数据(把数据附加在 URL 后面)，如果要上传大量数据，则应该使用 POST 方法。
- 第二个参数用来设置要打开的服务器文件(该文件可以是任意类型的文件)，这里要打开的是服务器端 mysite 站点内 server.asp 文件。
- 第三个参数用来指定不要异步请求。所谓异步就是发出请求之后，不需要等待服务器端的响应，用户可以继续执行其他操作。默认为 true，表示异步请求。

7.3.3 发送请求

当建立连接之后，就可以使用 send()方法发送请求到 HTTP 服务器端，并接收服务器的响应。send()方法只有一个参数，该参数可以传递客户端发送给服务器端的请求数据。如果不传递信息，则可设置参数值为 null。

send()方法虽然能够接收服务器端的响应，但是要把接收的数据读取出来，还需用到 responseBody、responseStream、responseText 或者 responseXML 属性。例如，在客户端的页面中输入下面脚本。

```
<script type="text/javascript">
//省略定义 XMLHttpRequest 对象，请参阅 7.2.2 节内容
window.onload = function(){
    var input = document.getElementsByTagName("input")[0];
    input.onclick = function(){
        xmlhttprequest.open("GET","test1.asp?name=css8", false);    //建立连接
        xmlhttprequest.send(null);   //发送请求
        alert(xmlhttprequest.responseText);   //提示服务器端响应信息
    }
}
</script>

<input type="button" value="向服务器发出异步请求" />
```

在上面脚本中，可以在 open()方法的第二个 URL 参数值末尾附加查询字符串，以便向服务器端传递请求的数据，然后再在服务器端的 test1.asp 文件中接收这个值，具体代码如下。

```
<%@LANGUAGE="JAVASCRIPT" CODEPAGE="65001"%>
<%
var name = Request.QueryString("name");
if(name && name == "css8"){     //如果存在该参数，且参数值等于"css8"
    Response.Write(name + "是合法的用户名。");
}
else{     //否则提示其他信息
    Response.Write(name + "非法的用户名");
}
%>
```

上面的脚本是以 JavaScript 脚本形式进行书写的，Request 和 Response 作为 ASP 的两个基本对象而存在，它们分别用来接收请求信息和发送响应信息。

首先，使用 Request 对象的 QueryString 集合对象读取请求的 URL 中的查询字符串信息，如果存在 name 参数值，且该参数值等于"css8"，则响应给客户端一种提示信息，否则就响应另一种信息。响应信息通过 Response 对象的 write()方法实现。

然后，客户端可以借助 XMLHttpRequest 对象的 responseText 属性读取服务器端响应信息的文本字符串，最后把这些信息以提示的形式显示出来，如图 7.3 所示。

图 7.3 GET 请求和响应

7.3.4 发送 GET 请求

GET 方法就是通过查询字符串的方式来传递请求信息。GET 请求的参数通过问号(?)前缀附加在 URL 的末尾，参数是以连字符(&)连接的一个或多个名/值对。每个名称和值都必须在编码后才能用在 URL 中，读者可以在 JavaScript 中使用 encodeURIComponent()方法进行编码，服务器端在接收这些数据时也必须使用 decodeURIComponent()方法进行解码。

当使用 XMLHttpRequest 对象发送一个 GET 请求时，只需将包含所有参数的 URL 传入 open()方法，同时设置第一个参数值为“GET”即可。这样服务器就能够自动在 URL 后面的查询字符串中接收到客户端传递过来的信息。使用 GET 请求比较简单，也非常方便。由于 URL 最大长度为 2048 字符(2KB)，因此它适合传递一些简单的参数信息，而不易传输大容量或受保护的数据。

演示示例可以参阅上一节示例，当然读者可以在 GET 请求中发送更多的参数信息。例如，在下面连接请求中，共传递了 3 个参数值：参数变量 name 的值等于 css8，参数变量 pass 的值等于 123456，参数变量 age 的值等于 1。

```
xmlhttprequest.open("GET","test1.asp?name=css8&pass=123456&age=1", false); //建立连接
```

7.3.5 发送 POST 请求

POST 请求方式与 GET 请求方式截然不同。POST 请求支持发送任意格式、任意长度的数据，而不仅仅限于名/值对字符串。对于传递二进制的文件、大容量信息、安全信息或 XML 格式数据时，使用 POST 方式比较高效。

一般来说，POST 请求用于在表单中输入数据后的提交过程。与 GET 请求相似，POST 请求的参数也必须进行编码，并用连字符(&)进行分隔，这些参数在发送 POST 请求时，不会被附加到 URL 的末尾，而是作为 send()方法的参数进行传递，然后被送到服务器端。

例如，对于"test1.asp?name=css8&pass=123456&age=1"查询字符串，可以作为参数传递给 send()方法，代码如下所示。

```
send("name=css8&pass=123456&age=1");
```

如果使用 POST 方式模仿前面示例效果实现把"name=css8"参数信息传递给服务器，则可以使用如下代码。

```
<script type="text/javascript">
//省略定义 XMLHttpRequest 对象，请参阅 7.2.2 节内容
window.onload = function(){
    var input = document.getElementsByTagName("input")[0];
    input.onclick = function(){
        xmlhttprequest.open("POST","test2.asp", false);    //建立连接

xmlhttprequest.setRequestHeader('Content-type','application/x-www-form-urlencoded')
;
        xmlhttprequest.send("name=css8");     //发送请求
        alert(xmlhttprequest.responseText);   //提示服务器端响应信息
    }
}
</script>

<input type="button" value="向服务器发出异步请求" />
```

由于使用 POST 方式进行请求，需要对部分代码进行以下修改。

- 在 open()方法中设置第一个参数为 POST，表明当前请求连接的是 POST 方式。
- 调用 setRequestHeader()方法，设置请求的消息头，指定请求内容类型为"application/x-www-form-urlencoded"。

"application/x-www-form-urlencoded"类型表示传递的是表单值，一般使用 POST 发送请求时都必须设置该选项，否则服务器会无法识别传递过来的数据。setRequestHeader()方法的语法格式如下。

```
xmlhttprequest.setRequestHeader("Header-name", "value");
```

为了方便服务器能够识别当前请求为 Ajax 异步请求，一般设置头部信息中 User-Agent 首部为 XMLHTTP，以便于服务器端能够辨别出 XMLHttpRequest 异步请求和其他客户端的普通请求，语法格式如下。

```
xmlhttprequest.setRequestHeader("User-Agent", "XMLHTTP");
```

这样就可以在服务器端编写脚本，分别为现代浏览器和不支持 JavaScript 的浏览器呈现不同的文档，以提高可访问性。如果使用 POST 方法传递数据，就必须设置另一个头部信息，代码如下。

```
xmlhttprequest.setRequestHeader("Content-type ", " application/x-www-form-urlencoded ");
```

用于发送 POST 请求的数据类型(Content-Type)通常是 application/x-www-form- urlencoded，这意味着读者还可以以 text/xml 或 application/xml 类型给服务器直接发送 XML 数据，甚至以 application/json 类型发送 JavaScript 对象数据。例如，下面的示例将向服务器端发送 XML 类型的数据，而不是简单的名/值对参数。

```
xmlhttprequest.send("<bookstore><book id='1'>书名</book>< /bookstore >")
```

读者可以访问 http://www.w3.org/Protocols/HTTP/HTRQ_Headers.html 网页了解 HTTP 请求头信息的总览表。

- 在 send()方法中传递参数值，该值是一个或多个“名/值”对，多个“名/值”对之间使用“&”分隔符进行分隔。这样在 ASP 服务器端就可以利用 Request.Form()方法捕获所传递过来的值。

在“名/值”对中，“名”可以为表单域的名称(与表单域相对应)，“值”可以是固定的值，也可以是一个变量。如果是变量，可以把表单域内包含的值直接传递给变量，再由变量负责把数据传递给服务器端。

- 必须把 open()方法中的第三个参数值设置为 false，即关闭异步通信。如果不需要通过 send()方法传递数据，则只要传递 null 作为参数值即可。

最后，还需要对服务器端的请求文件进行修改，使用 Request.Form()方式获取客户端传递的参数值，具体实现代码如下。

```
<%@LANGUAGE="JAVASCRIPT" CODEPAGE="65001"%>
<%
var name = Request.Form("name");
if(name && name == "css8"){     //如果存在该参数，且参数值等于"css8"
   Response.Write(name + "是合法的用户名。");
}
else{     //否则提示其他信息
   Response.Write(name + "非法的用户名");
```

```
}
%>
```

注意：针对 GET 和 POST 方式，服务器端获取数据的方法也是不同的，不同的服务器技术可能会略有区别，上面的示例主要根据 ASP 技术进行演示。

7.3.6 跟踪响应状态

异步通信实际上就是不需要刷新的暗部通信方式，这种方式虽然存在很多优势，但是却无法让人直观了解通信的过程和状态。为了避免客户端被动的等待，XMLHttpRequest 对象定义了 readyState 属性，该属性可以动态跟踪异步通信的状态。readyState 属性值说明如表 7.3 所示。

表 7.3 readyState 属性值列表

属 性 值	说 明
0	未初始化。表示对象已经建立，但是尚未初始化，尚未调用 open()方法
1	初始化。表示对象已经建立，尚未调用 send()方法
2	发送数据。表示 send()方法已经调用，但是当前的状态及 HTTP 头未知
3	数据传送中。表示已经接收部分数据，因为响应及 HTTP 头不全，这时通过 responseBody 和 responseText 获取部分数据会出现错误
4	完成。表示数据接收完毕，此时可以通过 responseBody 和 responseText 获取完整的响应数据

在异步通信过程中，一旦请求和响应的状态发生变化，也就是说当 readyState 属性值发生变化时，就会触发 onreadystatechange 事件处理函数。

readystatechange 是一种特殊的事件类型，它与 HTTP 传输紧密相关。每当 HTTP 请求状态改变时，服务器都会回调客户端的 onreadystatechange 事件处理函数。如果 readyState 属性返回值为 4，则说明异步请求和响应完毕，那么就可以放心地读取返回的数据了。

虽然通过 readyState 属性可以确定响应是否完成，但是还存在另一个问题：如果服务器完成响应请求，但是报告了一个错误，此时如果读取响应信息，也容易发生错误。为了防止此类意外，应坚持 HTTP 通信状态。只有当状态码为 200 时，才表示服务器正确完成了响应处理。在 XMLHttpRequest 对象中可以借助 status 属性获取当前的 HTTP 状态码。

例如，定义一个回调函数，检测服务器是否已正确完成响应，然后决定是否读取响应信息。实现的代码如下所示。

```
function response(){
   if(xmlhttprequest.readyState == 4){
      if (xmlhttprequest.status == 200 || xmlhttprequest.status == 0){
```

```
            alert(xmlhttprequest.responseText);
        }
    }
}
```

然后，绑定 onreadystatechange 事件处理函数(以上节示例为基础进行演示操作)，代码如下。

```
<script type="text/javascript">
//省略定义 XMLHttpRequest 对象，请参阅 7.2.2 节内容
window.onload = function(){
    var input = document.getElementsByTagName("input")[0];
    input.onclick = function(){
        xmlhttprequest.open("POST","test2.asp", false);    //建立连接

xmlhttprequest.setRequestHeader('Content-type','application/x-www-form-urlencoded')
;
        xmlhttprequest.onreadystatechange = response;      //绑定状态变化事件处理函数
        xmlhttprequest.send("name=css8");     //发送请求
    }
}
function response(){
    //省略代码，请参阅上面演示代码
}
</script>

<input type="button" value="向服务器发出异步请求" />
```

readystatechange 事件类型不会创建事件对象 Event，所以读者不要在事件处理函数中传递 Event 对象。另外，onreadystatechange 属性应该放置在调用 send()方法之前。也就是说，在 XMLHttpRequest 对象发送请求之前必须绑定 onreadystatechange 事件处理函数。

7.3.7 获取响应信息

XMLHttpRequest 对象定义了 responseText、responseBody、responseStream 和 responseXML 属性，分别用来存储服务器端不同的响应格式，具体说明如表 7.4 所示。

表 7.4 获取响应信息方式

属 性	说 明
responseBody	将响应信息正文以 Unsigned Byte 数组形式返回
responseStream	以 ADO Stream 对象的形式返回响应信息
responseText	将响应信息作为字符串返回
responseXML	将响应信息格式化为 XML 文档格式返回

服务器端响应数据一般为文本格式(responseText)或 XM 格式(responseText)，对于二进

制数据流可以使用 responseStream 属性存取。获取响应信息之后，还需要使用 JavaScript 脚本把它们转换为需要的形式进行显示。

XMLHttpRequest 对象允许从服务器端响应任意格式的数据。但在实际应用中，大多数的 Web 开发人员一般都将格式约定为 XML、HTML、JSON 或其他某种纯文本格式。读者不妨参考下面几条原则，来决定使用哪种响应格式。

- 如果是响应 HTML 结构的数据，选择 HTML 格式会比较省事，所要编写的脚本也会很少。此时使用 responseText 属性以字符串格式读取。
- 如果是团队协作开发，且项目庞杂，选择 XML 格式会更容易协调，因为所有成员都比较熟悉和了解这种格式。此时使用 responseXML 属性以 XML 格式读取。
- 如果是检索复杂的响应数据，且结构复杂，那么选择 JSON 格式会比较轻松。此时使用 responseText 属性以字符串格式读取。

例如，下面的示例是在上节示例基础上，演示如何获取 XML 格式的数据，并进行解析。

```
<script type="text/javascript">
//省略定义 XMLHttpRequest 对象，请参阅 7.2.2 节内容
window.onload = function(){
    var input = document.getElementsByTagName("input")[0];
    input.onclick = function(){
        xmlhttprequest.open("GET","test3.asp", false);      //建立连接
        xmlhttprequest.onreadystatechange = response;       //绑定状态变化事件处理函数
        xmlhttprequest.send(null);   //发送请求
    }
}
function response(){
    if(xmlhttprequest.readyState == 4){
        if (xmlhttprequest.status == 200 || xmlhttprequest.status == 0){
            var data = xmlhttprequest.responseXML;//获取 XML 格式的数据
            //利用 DOM 方法解析 XML 数据结构
            var node = data.getElementsByTagName("data")[0];    //获取第一个 data 节点
            var text = node.firstChild.data;  //获取该节点包含的文本节点中的数据
            alert(text);     //返回字符串"响应数据"
        }
    }
}
</script>

<input type="button" value="异步信息交互" />
```

在服务器端，设计使用 ASP 脚本生成 XML 文档(test3.asp)，代码如下。

```
<%@LANGUAGE="JAVASCRIPT" CODEPAGE="65001"%>
<?xml version="1.0" encoding="utf-8"?>
<%
Response.ContentType = "text/xml"
Response.Write("<data>响应数据</data>")
%>
```

对于 XML 文档结构来说，第一行必须是<?xml version="1.0" encoding="utf-8"?>，该行命令表示输出的数据为 XML 格式文档，同时标识了 XML 文档的版本和字符编码。为了能够兼容 IE 和 FF 等浏览器，允许不同浏览器都可以识别 XML 文档，还应该为响应信息定义 XML 文本类型。最后根据 XML 语法规范编写文档的信息结构。当然，也可以直接向服务器端的 XML 格式文件发出请求，此时可以返回该文档的数据。

如果以 JSON 格式解析响应数据，我们可以按如下方法来设计，示例的完整代码如下。

```
<script type="text/javascript">
//省略定义 XMLHttpRequest 对象，请参阅 7.2.2 节内容
window.onload = function(){
    var input = document.getElementsByTagName("input")[0];
    input.onclick = function(){
        xmlhttprequest.open("GET","test4.asp", false);      //建立连接
        xmlhttprequest.onreadystatechange = response;       //绑定状态变化事件处理函数
        xmlhttprequest.send(null);   //发送请求
    }
}
function response(){
    if(xmlhttprequest.readyState == 4){
        if (xmlhttprequest.status == 200 || xmlhttprequest.status == 0){
            var data =  xmlhttprequest.responseText;    //获取文本格式的数据
            //调用 JavaScript 的 eval()方法进行解析
            var obj = eval("(" + data + ")"); //把 JavaScript 字符串转换为 JavaScript 对象
            var text = obj.data;       //获取对象的属性值
            alert(text);      //返回字符串"响应数据"
        }
    }
}
</script>

<input type="button" value="异步信息交互" />
```

在服务器端设计请求的文件(test4.asp)内容如下。

```
{
data:"响应数据"
}
```

7.4 从 jQuery 角度分析 XMLHttpRequest 对象

jQuery 封装了 Ajax 的交互过程，读者不需要考虑 XMLHttpRequest 对象的兼容性问题，以及使用 XMLHttpRequest 建立连接、发送请求、发送方式、接收方式等技术细节。利用 jQuery 定义的几个简单方法，即可轻松实现客户端与服务器端异步通信问题，从而帮助开发人员从

繁琐的技术细节中解放出来，专心于业务层开发工作。

7.4.1 使用 GET 方式请求

jQuery 定义了 get()方法，专门负责通过远程 HTTP GET 请求方式载入信息。该方法是一个简单的 GET 请求功能，以取代复杂的 ajax()方法。

get()方法包含 4 个参数，说明如下，其中第一个参数为必须设置项，后面三个参数为可选参数。

- 第一个参数表示要请求页面的 URL 地址。
- 第二个参数表示一个对象结构的名/值对列表。
- 第三个参数表示异步交互成功之后调用的回调函数。回调函数的参数值为服务器端响应的信息。
- 第四个参数表示服务器端响应信息返回的内容格式，如 XML、HTML、Script、JSON 和 Text，或者_default。

例如，在下面这个示例中，使用 get()方法向服务器端的 test1.asp 文件发出一个请求，并把一组数据传递给该文件，然后在回调函数中读取并显示服务器端响应的信息。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){     //绑定 click 事件
          $.get("test1.asp",{  //向 test1.asp 文件发出请求
              name : "css8",     //发送的请求信息
              pass : 123456,
              age : 1
          },function(data){     //回调函数
              alert(data); //显示响应信息
        });
    });
})
</script>

<input type="button" value="jQuery 实现的异步请求" />
```

get()方法能够在请求成功时调用回调函数。如果需要在出错时执行函数，则必须使用 $.ajax()方法。

读者可以把 get()方法的第二个参数所传递的数据，以查询字符串的形式附加在第一个参数 URL 后面。例如，针对上面的 get()方法用法，还可以这样书写。

```
$.get("test1.asp?name=css8&pass=123456&age=1",function(data){//回调函数
      alert(data);    //显示响应信息
});
```

如果回忆上一节使用 JavaScript 直接编写的 GET 方式所建立的异步通信连接，相信读者能够理解这种方式。

jQuery 还定义了两个专用方法：getJSON()和 getScript()方法。这两个方法的功能和用法与 get()是完全相同的，不过 getJSON()方法能够请求载入 JSON 数据，getScript()方法能够请求载入 JavaScript 文件。

这两个方法与 get()方法的用法基本相同，但是仅支持 get()方法的前三个参数，不需要设置第四个参数，即指定响应数据的类型，因为方法本身已经说明了接收的信息类型。例如，在服务器端文件(test5.asp)中输入下面的响应信息。

```
[
    {name:"zhu",pass:"123456",age:"1"},
    {name:"zhang",pass:"abcdef",age:"2"},
    {name:"zhao",pass:"opqrst",age:"3"}
]
```

上面的信息以 JSON 格式进行编写，整个数据包含在一个数组中，每个数组元素是一个对象，对象中包含三个属性，分别是 name、pass 和 age。

然后，在客户端的 jQuery 脚本中，使用 getJSON()方法请求服务器端文件(test5.asp)，并把响应信息解析为数据表格形式显示，如图 7.4 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
        $.getJSON("test5.asp",function(data){ //使用 getJSON()方法发送请求并接收 JSON 格式数据
                var data  = data; //获取响应数据
                var str = "<table border=1 width=100%>";    //定义字符串临时变量
                str += "<tr>";
                for(var name in data[0]){ //遍历响应数据中的第一个数组元素对象
                    str += "<th>" + name + "</th>"; //获取并显示元素对象的属性名
                }
                str += "</tr>";
                for(var i=0; i<data.length; i++){  //遍历响应数据中的数组元素
                    str += "<tr>";
                    for(var name in data[i]){  //遍历数组元素中的每个属性成员
                        str += "<td>" + data[i][name]  + "</td>";    //获取并显示元素对象的属性值
                    }
                    str += "</tr>";
                }
                str += "<table>";
                $("div").html(str);   //把临时字符串以 HTML 格式嵌入到 div 元素中显示
        });
    });
})
```

```
</script>

<input type="button" value="jQuery 实现的异步请求" />
<div></div>
```

图 7.4 使用 getJSON()方法获取并解析 JSON 格式数据

使用 getScript()方法还能够异步请求并导入外部 JavaScript 文件，具体示例就不再演示。

7.4.2 使用 POST 方式请求

jQuery 定义了 post()方法，专门负责通过远程 HTTP POST 请求方式载入信息。该方法是一个简单的 POST 请求功能，以取代复杂的方法。

post()方法包含 4 个参数，与 get()方法相似，其中第一个参数为必须设置的参数，后面三个参数为可选参数。

- 第一个参数表示要请求页面的 URL 地址。
- 第二个参数表示一个对象结构的名/值对列表。
- 第三个参数表示异步交互成功之后调用的回调函数。回调函数的参数值为服务器端响应的信息。
- 第四个参数表示服务器端响应信息返回的内容格式，如 XML、HTML、Script、JSON 和 Text，或者_default。

例如，在下面这个示例中，使用 post()方法向服务器端的 test2.asp 文件发出一个请求，并把一组数据传递给该文件，然后在回调函数中读取并显示服务器端响应的信息。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){     //绑定 click 事件
          $.post("test2.asp",{ //向 test2.asp 文件发出请求
              name : "css8",     //发送的请求信息
              pass : 123456,
              age : 1
          },function(data){     //回调函数
              alert(data); //显示响应信息
       });
```

```
    });
})
</script>

<input type="button" value="jQuery实现的异步请求" />
```

通过上面的示例可以看到post()方法与get()方法在用法上是完全相同的，数据传递和接收响应信息的方式都相同，惟一的区别是请求方式不同。具体选用哪个方法，主要根据客户端所要传递的数据容量和格式，同时还应该考虑服务器端接收数据的处理方式。

不管是get()方法，还是post()方法，它们都是一种简单的请求方式，而对于特殊的数据请求和响应处理，应该选择$.ajax()方法，ajax()方法的参数比较多且复杂，能够处理各类特殊的异步交互行为。关于这个问题请参阅下一节内容。

7.4.3 使用ajax()方法请求

ajax()方法是jQuery实现Ajax的底层方法，也就是说它是get()、post()等方法的基础，使用该方法可以完成通过HTTP请求加载远程数据。由于ajax()方法的参数较为复杂，在没有特殊需求时，使用高级方法(如get()、post()等)即可。

ajax()方法只有一个参数，即一个列表结构的对象，包含各配置及回调函数信息，详细参数选项可以参阅表7.5。例如，加载JavaScript文件，则可以使用下面的参数选项。

```
$.ajax({
    type: "GET",        //请求方式
    url: "test.js",     //请求文件的URL
    dataType: "script" //响应的数据类型
});
```

如果把客户端的数据传递给服务器端，并获取服务器的响应信息，则可以使用类似下面的参数选项。

```
$.ajax({
    type: "POST",                          //请求方式
    url: "test.asp",                       //请求文件的URL
    data: "name=John&location=Boston",     //传递给服务器的数据
    success: function(data){               //异步通信成功后的回调函数
        alert(data);                       //显示服务器的响应信息
    }
});
```

如果要加载HTML页面，则可以使用下面的参数选项。

```
$.ajax({
    url: "test.html",           //请求文件的URL
    cache: false,               //禁止缓存
```

```
    success: function(html){    //异步通信成功后的回调函数
        $("#box").append(html); //把 HTML 片段附加到当前文档的盒子中
    }
});
```

如果希望以同步方式加载数据，则可以使用下面的选项设置。使用同步方式加载数据时，其他用户的操作将被锁定。

```
var html = $.ajax({
    url: "test.asp",    //请求文件的 URL
    async: false        //同步请求
 }).
```

表 7.5 ajax()方法参数选项列表

参 数	数据类型	说 明
async	Boolean	设置是否异步请求。默认为 true，即所有请求均为异步请求。如果需要发送同步请求，设置为 false 即可。注意，同步请求将锁住浏览器，用户其他操作必须等待请求完成才可以执行
beforeSend	Function	发送请求前可修改 XMLHttpRequest 对象的函数，如添加自定义 HTTP 头。XMLHttpRequest 对象是惟一的参数。 该函数如果返回 false，可以取消本次 Ajax 请求
cache	Boolean	设置缓存。默认值为 true，dataType 为 script 时，默认为 false。设置为 false 将不会从浏览器缓存中加载请求信息
complete	Function	请求完成后回调函数 (请求成功或失败时均调用)。该函数包含两个参数：XMLHttpRequest 对象和一个描述成功请求类型的字符串
contentType	String	发送信息至服务器时的内容编码类型。默认为"application/x-www-form-urlencoded"
data	Object、String	发送到服务器的数据。将自动转换为请求字符串格式，必须为 Key/Value 格式。GET 请求中将附加在 URL 后。查看 processData 选项说明以禁止此自动转换。如果为数组，jQuery 将自动为不同值对应同一个名称。如将{foo:["bar1", "bar2"]} 转换为 '&foo=bar1&foo=bar2'
dataFilter	Function	给 Ajax 返回的原始数据进行预处理的函数。提供 data 和 type 两个参数：data 是 Ajax 返回的原始数据，type 是调用 jQuery.ajax 时提供的 dataType 参数。函数返回的值将由 jQuery 进一步处理
dataType	String	预期服务器返回的数据类型。如果不指定，jQuery 将自动根据 HTTP 包含的 MIME 信息返回 responseXML 或 responseText，并作为回调函数参数传递，可用值包括以下几个。 "xml": 返回 XML 文档，可用 jQuery 处理。 "html": 返回纯文本 HTML 信息；包含的 script 标签会在插入 dom 时执行。 "script": 返回纯文本 JavaScript 代码。不会自动缓存结果，除非设置了"cache"参数。注意：在远程请求时(不在同一个域下)，所有 POST 请求都将转为 GET 请求(因为将使用 DOM 的 script 标签来加载)。 "json": 返回 JSON 数据。 "jsonp": JSONP 格式。使用 JSONP 形式调用函数时，如"myurl?callback=?"，jQuery 将自动替换?为正确的函数名，以执行回调函数。 "text": 返回纯文本字符串

续表

参　数	数据类型	说　明
error	Function	请求失败时调用函数。该函数包含三个参数：XMLHttpRequest 对象、错误信息和(可选)捕获的错误对象。如果发生了错误，错误信息(第二个参数)除了可能是 null 之外，还可能是"timeout"、"error"、"notmodified"和"parsererror"
global	Boolean	是否触发全局 Ajax 事件，默认值为 true。设置为 false 将不会触发全局 Ajax 事件，如 ajaxStart 或 ajaxStop 可用于控制不同的 Ajax 事件
ifModified	Boolean	仅在服务器数据改变时获取新数据，默认值为 false。使用 HTTP 包含的 Last-Modified 头信息进行判断
jsonp	String	在一个 jsonp 请求中重写回调函数的名字。这个值用来替代在"callback=?"这种 GET 或 POST 请求中 URL 参数里的"callback"部分，比如{jsonp:'onJsonPLoad'}会导致将"onJsonPLoad=?"传给服务器
password	String	用于响应 HTTP 访问认证请求的密码
processData	Boolean	发送的数据将被转换为对象(技术上讲并非字符串) 以配合默认内容类型 "application/x-www-form-urlencoded"。默认值为 true，如果要发送 DOM 树信息或其他不希望被转换的信息，请设置为 false
scriptCharset	String	只有当 dataType 为"jsonp"或"script"，并且 type 是"GET"时才会用于强制修改 charset。通常本地和远程的内容编码不同时使用
success	Function	请求成功后的回调函数。函数的参数由服务器返回，并根据 dataType 参数返回进行处理后的数据；描述状态的字符串
timeout	Number	设置请求超时时间(毫秒)。此设置将覆盖全局设置
type	String	设置请求方式，如"POST"或"GET"， 默认为"GET"。其他 HTTP 请求方法，如 PUT 和 DELETE 也可以使用，但仅部分浏览器支持
url	String	发送请求的地址，默认为当前页面地址
username	String	用于响应 HTTP 访问认证请求的用户名
xhr	Function	需要返回一个 XMLHttpRequest 对象。在 IE 下默认是 ActiveXObject，而其他情况下是 XMLHttpRequest。用于重写或者提供一个增强的 XMLHttpRequest 对象

如果设置了 dataType 选项，应确保服务器返回正确的 MIME 信息，例如，XML 返回"text/xml"。如果设置 dataType 为"script"，则在请求时，如果请求文件与当前文件不在同一个域名中，所有 POST 请求都被转换为 GET 请求，因为 jQuery 将使用 DOM 的 script 标签来加载响应信息。

7.4.4 跟踪响应状态

jQuery 在 XMLHttpRequest 对象定义的 readyState 属性基础上，对异步交互中服务器响

应状态进行封装，提供了 6 个响应事件，以便进一步细化对整个请求响应过程的跟踪，具体说明如表 7.6 所示。

表 7.6 jQuery 封装的响应状态事件

事 件	说 明
ajaxStart()	Ajax 请求开始时进行响应
ajaxSend()	Ajax 请求发送前进行响应
ajaxComplete()	Ajax 请求完成时进行响应
ajaxSuccess()	Ajax 请求成功时进行响应
ajaxStop()	Ajax 请求结束时进行响应
ajaxError()	Ajax 请求发生错误时进行响应

例如，在下面的示例中，为当前异步请求绑定 6 个 jQuery 定义的 Ajax 事件，在浏览器中预览，则可以看到浏览器根据请求和响应的过程，逐步提示过程进展。首先，响应的是 ajaxStart 和 ajaxSend 事件，然后是 ajaxSuccess 事件，最后是 ajaxComplete 和 ajaxStop 事件，如图 7.5 所示。如果请求失败，则中间会响应 ajaxError 事件。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
        $.ajax({
            type: "POST",
            url: "test2.asp",
            data: "name=css8"
        });
        $("div").ajaxStart(function(){
            alert("Ajax 请求开始");
        })
        $("div").ajaxSend(function(){
            alert("Ajax 请求将要发送");
        })
        $("div").ajaxComplete(function(){
            alert("Ajax 请求完成");
        })
        $("div").ajaxSuccess(function(){
            alert("Ajax 请求成功");
        })
        $("div").ajaxStop(function(){
            alert("Ajax 请求结束");
        })
        $("div").ajaxError(function(){
            alert("Ajax 请求发生错误");
        })
    });
```

```
})
</script>

<input type="button" value="jQuery 实现的异步请求" />
```

图 7.5 jQuery 的 Ajax 事件响应过程

在这些事件中大部分都会包含几个默认参数。例如，ajaxSuccess、ajaxSend 和 ajaxComplete 都包含 event、request 和 settings 参数，其中 event 表示事件类型，request 表示请求信息，settings 表示设置的选项信息。ajaxError 事件还包含 4 个默认参数：event、XMLHttpRequest、ajaxOptions 和 thrownError，其中前 3 个参数与上面几个事件方法的参数基本相同，最后一个参数表示抛出的错误。

7.4.5 载入网页文件

遵循 Ajax 异步交互的设计原则，jQuery 定义了可以加载网页文档的方法：load()方法。该方法与 getScript()方法的功能相似，都是加载外部文件，但是它们的用法完全不同。load()方法能够把加载的网页文件附加到指定的网页标签中。

例如，新建一个简单的网页文件(test.html)，实现代码如下。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>静态网页文件</title>
</head>
<body>
<table width="100%" border="1">
    <tr>
        <th>name</th>
        <th>pass</th>
        <th>age</th>
    </tr>
    <tr>
        <td>zhu</td>
        <td>123</td>
```

```
        <td>1</td>
    </tr>
    <tr>
        <td>zhang</td>
        <td>456</td>
        <td>2</td>
    </tr>
    <tr>
        <td>wang</td>
        <td>789</td>
        <td>3</td>
    </tr>
</table>
</body>
</html>
```

然后，在另一个页面中输入下面的 jQuery 脚本。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){
        $("div").load("test.html");
    });
})
</script>
<input type="button" value="jQuery 实现的异步请求" />
<div></div>
```

这样当在浏览器中预览时，打击“jQuery 实现的异步请求”按钮后，则会把请求的 test.html 文件中的数据表格加载到当前页面的 div 元素中，如图 7.6 所示。

图 7.6 使用 jQuery 的 load()方法载入外部文件

使用 ajax()方法可以替换 load()方法，因为 load()方法是以 ajax()方法作为底层来实现的。例如，针对上面的示例，可以使用下面的 jQuery 代码进行替换。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
```

```
    $("input").click(function(){
        var str = ($.ajax({ //调用 ajax()方法，返回 XMLHttpRequest 对象
            url :  "test.html",   //载入的 URl
            async: false      //禁止异步载入
        })).responseText;    //获取 XMLHttpRequest 对象中包含的服务器响应信息
        $("div").html(str);       //把载入的网页内容附加到 div 元素内
    });
})
</script>

<input type="button" value="jQuery 实现的异步请求" />
<div></div>
```

7.4.6 预设 Ajax 选项

对于频繁与服务器进行交互的页面来说，每一次交互都要设置很多选项，这种操作很繁琐，也很容易出错。为此，jQuery 定义了 ajaxSetup()方法，该方法能够预设异步交互中的通用选项，从而减轻设置选项的频繁。

ajaxSetup()方法仅包含一个参数选项的列表对象，这与 ajax()方法的参数选项设置是相同的。在该方法中设置的选项，可以实现全局共享，从而在具体交互中只需要设置个性化参数即可。

例如，在下面的示例中，先使用 ajaxSetup()方法把本页面中异步交互的公共选项进行预设，包括请求的服务器端文件、禁止触发全局 Ajax 事件、请求方式、响应数据类型和响应成功之后的回调函数。这样在不同按钮上绑定异步请求时，只需要设置需要发送请求的信息即可。

在服务器端的请求文件(test6.asp)中输入下面的代码。

```
<%@LANGUAGE="JAVASCRIPT" CODEPAGE="65001"%>
<%
var name = Request.Form("name");
if(name){
    Response.Write("接受到请求信息：" + name);
}
else{
    Response.Write("没有接受到请求信息！");
}
%>
```

这样当单击不同按钮时，会弹出不同的响应信息，这些信息都是从客户端接收到的请求信息，如图 7.7 所示。完整设计代码如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
```

```
    $.ajaxSetup({ //预设公共选项
        url: "test6.asp",   //请求的 URL
        global: false, //禁止触发全局 Ajax 事件
        type: "POST",   //请求方式
        dataType: "text",   //响应数据的类型
        success : function(data){    //响应成功之后的回调函数
            alert(data);
        }
    });
    $("input").eq(0).click(function(){   //为按钮 1 绑定异步请求
        $.ajax({
            data : "name=zhu"    //发送请求的信息
        });
    });
    $("input").eq(1).click(function(){   //为按钮 2 绑定异步请求
        $.ajax({
            data : "name=wang"   //发送请求的信息
        });
    });
    $("input").eq(2).click(function(){   //为按钮 3 绑定异步请求
        $.ajax({
            data : "name=zhang"  //发送请求的信息
        });
    });
})
</script>

<input type="button" value="异步请求 1" />
<input type="button" value="异步请求 2" />
<input type="button" value="异步请求 3" />
```

图 7.7　使用 jQuery 的 load()方法载入外部文件

7.4.7　预处理请求的字符串

在 Ajax 异步通信过程中，客户端所发送的请求字符串格式必须是由“&”字符连接的多个名/值对，例如“user=zhu&sex=man&grade=2”。而当使用表单发送请求时，发送请求的

信息并非按此格式进行传递，用户需要手工编写发送信息的字符串格式。为了减轻开发人员不必要的劳动量，jQuery 特意定义了 serialize()方法，该方法能够帮助用户按名/值对的字符串格式快速整理，并返回合法的请求字符串。

例如，在下面这个复杂表单中，用户需要传递的表单值是比较多的，如果逐项获取并组织为请求字符串，就稍显繁琐。

```
<form action="#" method="post">
    姓名：<input  type="text" name="user" /><br />
    性别：
    <input type="radio" name="sex" value="man" checked="checked" />男
    <input type="radio" name="sex" value="woman" />女<br />
    年级：
    <select name="grade">
        <option value="1">一</option>
        <option value="2">二</option>
        <option value="3">三</option>
    </select><br />
    科目：
    <select name="kemu" size="6" multiple="multiple">
        <option value="yuwen">语文</option>
        <option value="shuxue">数学</option>
        <option value="waiyu">外语</option>
        <option value="wuli">物理</option>
        <option value="huaxue">化学</option>
        <option value="jisuanji">计算机</option>
    </select><br />
    兴趣：
    <input type="checkbox" name="love" value="yundong" />运动
    <input type="checkbox" name="love" value="wenyi" />文艺
    <input type="checkbox" name="love" value="yinyue" />音乐
    <input type="checkbox" name="love" value="meishu" />美术
    <input type="checkbox" name="love" value="youxi" />游戏<br />
    <input type="submit" value="提交" id="submit" />
</form>
```

如果在发送请求之前，调用 serialize()方法，就可以轻松解决合法格式请求字符串的设计。实现代码如下所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $("#submit").click(function(){
        $("p").html($("form").serialize());  //获取和格式化表单的请求字符串信息，并显示出来
        return false;  //禁止提交表单
    });
})
</script>
```

在浏览器中预览，然后单击提交按钮，则可以看到规整的请求字符串，如图 7.8 所示。

图 7.8　预处理请求的字符串

除了 serialize()方法外，jQuery 还定义了 serializeArray()方法，该方法能够返回指定表单域值的 JSON 结构的对象。注意，该方法返回的是 JSON 对象，而非 JSON 字符串。JSON 对象是由一个对象数组组成的，其中每个对象包含一个或两个名/值对：name 参数和 value 参数(如果 value 不为空的话)。

例如，针对上面的表单结构，我们可以设计如下的 jQuery 代码，获取用户传递的请求值，并把这个 JSON 结构的对象解析为 HTML 字符串显示出来，如图 7.9 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $("#submit").click(function(){
        var array = $("form").serializeArray();   //注意，不能够直接在 form 元素上调用该方法
        var array = $("input, select, :radio").serializeArray();   //在表单域对象上调用
serializeArray()方法，返回包含传递表单域和值的 JSON 对象
        var str = "[ <br />"
        for(var i = 0; i<array.length; i++){      //遍历数组格式的 JSON 对象
            str += "    {"
            for(var name in array[i]){            //遍历数组元素对象
                str += name + ":" + array[i][name]  + ","    //组合为 JSON 格式字符串
            }
            str = str.substring(0,str.length-1);  //清除掉最后一个字符
            str += "},<br />";
        }
        str = str.substring(0,str.length-7);      //清除掉最后 7 个字符
        str += "<br />]";
        $("p").html(str);                         //显示返回的 JSON 结构字符串
        return false;
    });
})
    </script>
```

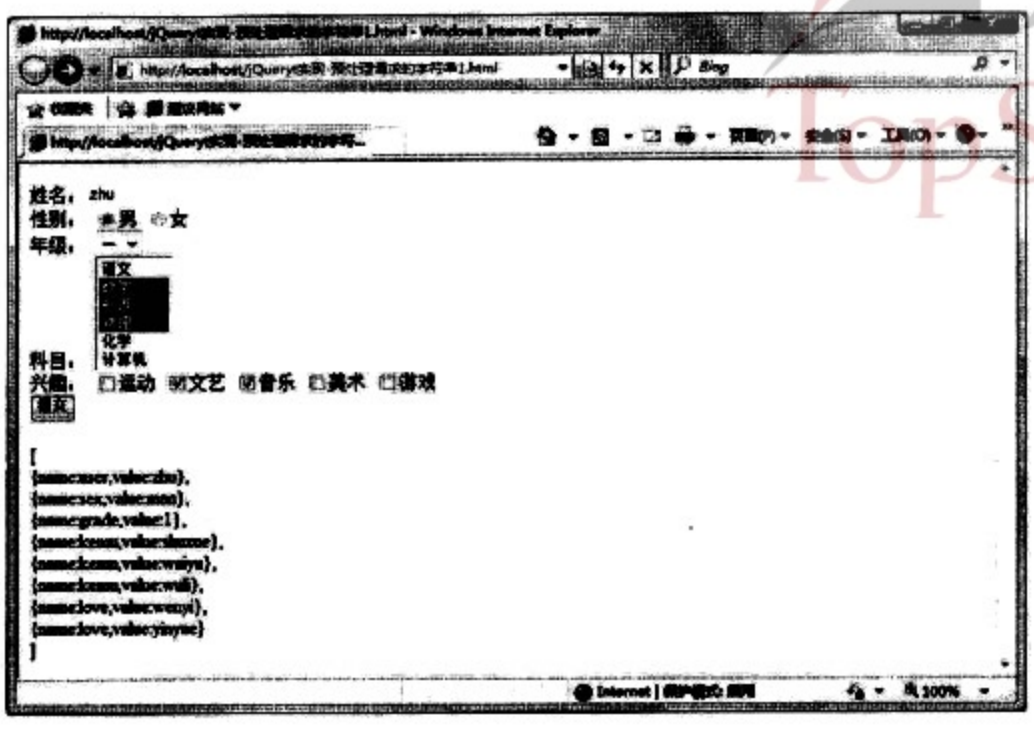


图 7.9 把请求的值转换为 JSON 对象结构

第8章 高效开发和使用插件

jQuery 的流行在很大程度上归功于其对插件的支持。插件也就是功能扩展的意思，jQuery 允许任何开发人员超越最初的库函数创建并扩展 jQuery 函数。这种开发性框架设计思路催生了无数实用型的插件，jQuery 几乎能够提供 Web 应用程序内所需的任何一种函数。

jQuery 的易扩展性吸引了越来越多的开发者和业余爱好者去研究、设计和使用 jQuery 插件。目前，全球有超过上千种不同应用需要的插件。使用这些插件能够帮助开发人员提升开发速度，节约劳动成本。最权威的插件可以访问 jQuery 官方网站(http://plugins.jquery.com)获取。本章将重点讲解如何创建自己的 jQuery 插件，并就网络上比较经典的几个插件设计原理和使用进行介绍。

8.1 创建 jQuery 插件

网络上流传着成百上千的第三方插件，这些插件虽然能够增强我们的编程体验，但是很难满足所有设计需要，特别是个性化开发需求。如果自己编写的代码可以重用，或者供其他用户参考，很自然任何人都希望把这些代码进行封装，打包为一个新插件。这个实现过程并不困难，只要读者认真阅读本节内容即可轻松实现。

8.1.1 jQuery 插件的类型

jQuery 插件主要分为三种类型，说明如下。

1. jQuery 方法

这种类型的插件是把一些常用或者重复使用的功能定义为函数，然后绑定到 jQuery 对象上，从而成为 jQuery 对象的一个扩展方法。

目前，大部分 jQuery 插件都是这种类型的插件，由于这种插件是将对象方法封装起来，在 jQuery 选择器获取 jQuery 对象过程中进行操作，从而发挥 jQuery 强大的选择器优势。有很多 jQuery 内部方法，也是在 jQuery 脚本内部通过这种形式插入到 jQuery 框架中的，如 parent()、appendTo()和 addClass()等方法。

2. 全局函数

也可以把自定义的功能函数独立附加到 jQuery 命名空间下，从而作为 jQuery 作用域下的一个公共函数使用。例如，jQuery 的 ajax()方法就是利用这种途径内部定义的全局函数。

由于全局函数没有被绑定到 jQuery 对象上，故不能够在选择器获取的 jQuery 对象上调用。需要通过 jQuery.fn()或者$.fn()方式进行引用。

3. 选择器

jQuery 提供了强大的选择器，当然在个性化开发中，读者也可能会感觉到这些选择器不够用，或者使用不是很方便。这个时候，我们就可以考虑自定义选择器，以满足特定环境下的选择元素需要。

8.1.2 解析 jQuery 插件机制

为了方便用户创建插件，jQuery 自定义了 jQuery.extend()和 jQuery.fn.extend()方法。其中 jQuery.extend()方法能够创建全局函数或者选择器，而 jQuery.fn.extend()方法能够创建

jQuery 对象方法。

例如，下面的示例将在 jQuery 命名空间上创建两个公共函数。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery.extend({    //扩展 jQuery 的公共函数
    minValue : function(a,b){     //比较两个参数值，返回最小值
        return a<b?a:b;
    },
    maxValue : function(a,b){     //比较两个参数值，返回最大值
        return a<b?b:a;
    }
})
</script>
```

然后就可以在页面中调用这两个公共函数，示例代码如下所示。在下面这个示例中，当单击按钮后，浏览器会弹出提示对话框，要求输入两个值，然后提示两个值的大小。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
//省略 jQuery.minValue()和 jQuery.maxValue()方法创建代码，请参阅上面代码
$(function(){
    $("input").click(function(){
        var a = prompt("请输入一个数值？");
        var b = prompt("请再输入一个数值？");
        var c = jQuery.minValue(a,b);
        var d = jQuery.maxValue(a,b);
        alert("你输入的最大值是：" + d + "\n 你输入的最小值是：" + c);
    });
})
</script>

<input type="button" value="jQuery 插件扩展测试" />
```

jQuery.extend()和 jQuery.fn.extend()方法都包含一个参数，该参数仅接受名/值对结构的对象，其中名表示函数或方法名，而值表示函数体。关于如何创建 jQuery 插件的问题，我们将在下面小节中进行详细讲解。

jQuery.extend()方法除了可以创建插件外，还可以用来扩展 jQuery 对象。例如，在下面的示例中，调用 jQuery.extend()方法把对象 a 和 b 合并为一个新的对象，并返回合并对象将其赋值给变量 c。

在合并操作中，如果存在同名属性，则后面参数对象的属性值会覆盖前面参数对象的属性值，在下面的示例中当把对象 a 和 b 合并为 c 之后，则合并后的对象如图 8.1 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
var a = {    //对象直接量
```

```
    name : "zhu",
    pass : 123
}
var b = {    //对象直接量
    name : "wang",
    pass : 456,
    age : 1
}
var c = jQuery.extend(a,b);     //合并对象 a 和 b
$(function(){
    for(var name in c){     //遍历对象 c，显示合并后的对象 c 的具体属性和值
        $("div").html($("div").html()  + "<br />"+ name + ":" +  c[name]);
    }
})
</script>

<div></div>
```

图 8.1　合并后的对象

在实际开发中，常常使用 jQuery.extend()方法为插件方法传递系列选项结构的参数。

```
function fn(options){
    var options = jQuery.extend({    //默认参数选项列表
        name1 : value1,
        name2 : value2,
        name3 : value3
    }, options);    //使用函数的参数覆盖或合并到默认参数选项列表中
    //函数体
}
```

这样当在调用该方法时，如果想传递新的参数值，就会覆盖掉默认的参数选项值，或者向函数参数添加新的属性和值；如果没有传递参数，则保持并使用默认值。例如，在下面几个函数调用中，分别传入新值，或者添加新参数，或者保持默认值。

```
fn({name1 : value2, name2 : value3, name3 : value1});    //覆盖新值
fn({name4 : value4, name5 : value5 });    //添加新选项
fn();    //保持默认参数值
```

jQuery.extend()方法的对象合并机制，比传统的逐个检测参数不仅灵活且简洁，使用命名参数添加新选项也不会影响已编写的代码风格，让代码变得更加直观明白。

8.1.3 创建 jQuery 全局函数

jQuery 内置的很多方法都是通过全局函数实现的。所谓全局函数，就是 jQuery 对象的方法，实际上就是位于 jQuery 命名空间内部的函数。

有人把这类函数称为实用工具函数，这些函数有一个共同特征，就是不直接操作 DOM 元素，而是操作 JavaScript 的非元素对象，或者执行其他非对象的特定操作，如 jQuery 的 each()函数和 noConflict()函数。

回忆一下上一章讲解过的 ajax()方法，它就是一个典型的 jQuery 全局函数，$.ajax()所做的一切都可以通过调用名称为 ajax()的全局函数来实现。但是，这种方式会带来函数冲突问题，如果把函数放置在 jQuery 命名空间内，就会降低这种冲突，只要在 jQuery 命名空间内注意不要与 jQuery 其他方法冲突即可。

在上一节中曾经介绍了使用 jQuery.extend()方法可以扩展 jQuery 对象的全局函数。当然，我们也可以使用下面的方法快速定义 jQuery 全局函数。例如，针对上一节示例，我们也可以按如下方法进行编写。

```
<script src="images/jquery-1.3.2.js"
```

```
        return a<b?b:a;
    }
}
</script>
```

尽管我们仍然可以把这些函数当成全局函数来看待，但是从技术层面分析，它们现在都是 jQuery 全局函数的方法，因此在调用这些函数时，请求方式也会发生变化，调用函数的代码如下。

```
var c = jQuery.css8.minValue(a,b);
var d = jQuery.css8.maxValue(a,b);
```

这样就可以轻松避免与别的插件发生冲突。注意，即使页面中包含了 jQuery 框架文件，但是考虑到安全性，不建议以一种简写的方式(即使用$代替 jQuery)进行书写，我们应该在编写的插件中始终使用 jQuery 来调用 jQuery 方法。

8.1.4 使用 jQuery.fn 对象创建 jQuery 对象方法

除了全局函数外，jQuery 中的大多数功能都是通过 jQuery 对象的方法提供的，这些对象方法对于 DOM 操作来说非常方便，这也是很多用户喜欢并选用 jQuery 框架的原因。

创建全局函数只需要通过为 jQuery 对象添加属性即可，而创建 jQuery 对象方法也可以通过为 jQuery.fn 对象添加属性来实现。实际上 jQuery.fn 对象就是 jQuery.prototype 原型对象的别名，使用别名更方便引用。例如，下面这个函数就是一个简单的 jQuery 对象方法，当调用该方法时，将会弹出一个提示对话框。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery.fn.test = function(){
    alert("这是 jQuery 对象方法！");
}
</script>
```

然后，就可以在任何 jQuery 对象中调用该方法。在下面的应用示例中，如果单击页面中的“jQuery 插件扩展测试”按钮，即可弹出一个提示对话框，提示"这是 jQuery 对象方法！"。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $("input").click(function(){    //绑定 click 事件
        $(this).test();    //在当前的 jQuery 对象上调用 test()方法
    });
})
</script>
```

```
<input type="button" value="jQuery 插件扩展测试" />
```

由于这里并没有在任何地方匹配 DOM 节点，所以编写全局函数也是可以的。但是，在使用 jQuery 对象方法时，函数体内的 this 关键字总是引用当前 jQuery 对象，因此我们可以对上面的方法进行重写，实现动态提示信息。代码如下。

```
jQuery.fn.test = function(){
    alert(this[0].nodeName);     //提示当前 jQuery 对象的 DOM 节点名称
}
```

这样，当单击按钮时，就会弹出当前元素的节点名称，如图 8.2 所示。

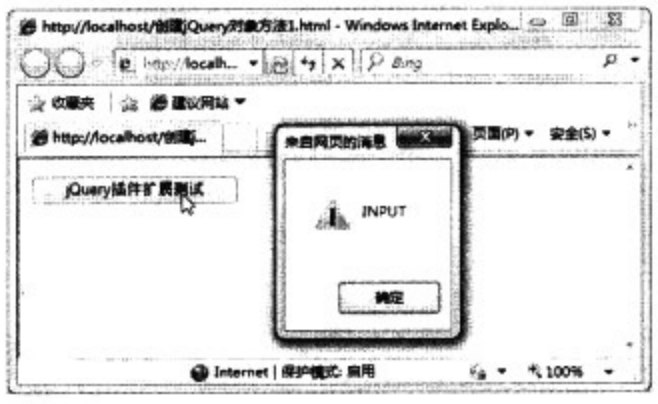


图 8.2　弹出当前对象的节点名称

在上面的示例中，可以看到由于 jQuery 选择器返回的是一个数组类型的 DOM 节点集合，this 指针就指向当前这个集合，故要显示当前元素的节点名称，就必须在 this 指针后面指定当前元素的序号。

试想，如果 jQuery 选择器匹配多个元素，我们该如何准确指定当前元素对象呢？

要解决这个问题，我们不妨在当前 jQuery 对象方法的环境中调用 each()方法，通过隐式迭代的方式，让 this 指针依次引用每一个匹配的 DOM 元素对象。这样也能够保持插件与 jQuery 内置方法保持一致。例如，针对上面的示例做进一步的修改。

```
jQuery.fn.test = function(){
    this.each(function(){     //遍历所有匹配的元素，此处的 this 表示对象集合，即 jQuery 对象
        alert(this.nodeName);     //显示当前元素的节点名称，此处的 this 表示元素对象
    });
}
```

然后，在调用该方法时，就不用担心 jQuery 选择器所匹配的元素有多少了。例如，在下面的示例中，当单击不同类型的元素，会提示当前元素的节点名称，如图 8.3 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery.fn.test = function(){     //定义的 jQuery 对象方法
    this.each(function(){     //遍历 jQuery 对象
        alert(this.nodeName);     //提示当前元素的节点名称
    });
```

```
}
$(function(){
    $("body *").click(function(){     //选择 body 元素下所有元素
        $(this).test();     //为当前元素调用 test()方法，提示当前 DOM 元素对象的节点名称
    });
})
</script>

<input type="button" value="jQuery 插件扩展测试" />
<div>div 元素</div>
<p>p 元素</p>
<span>span 元素</span>
```

图 8.3　显示当前元素的节点名称

这样就可以实现根据前面选择器所匹配元素的不同，令所定义的 test()方法给出不同的提示信息。

用惯 jQuery 的用户可能习惯于连写行为，也就是说在调用一个方法之后，紧跟着调用另一个方法，如此连写不断，形成一个珍珠链，而且编写灵活、方便，也符合人的思维习惯。例如：

```
$(this).test().hide().height();
```

要实现类似的行为连写功能，就应该在每个插件方法中返回一个 jQuery 对象，除非方法需要明确返回值。返回的 jQuery 对象通常就是 this 所引用的对象。如果使用 each()方法迭代 this，则可以直接返回迭代的结果。例如，针对上面的示例做如下进一步的修改。

```
jQuery.fn.test = function(){
    return this.each(function(){     //返回迭代的 jQuery 对象
        alert(this.nodeName);
    });
}
```

然后，我们就可以在应用示例中连写行为了。例如，在下面的示例中，先弹出提示节点名称的信息，然后使用当前节点名称改写当前元素内包含的信息，最后再慢慢隐藏该元素，如图 8.4 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
```

```
<script type="text/javascript">
$(function(){
    $("body *").click(function(){
        $(this).test().html(this.nodeName).hide(4000);    //行为连写
    });
})
</script>

<div>div 元素</div>
<p>p 元素</p>
<span>span 元素</span>
```

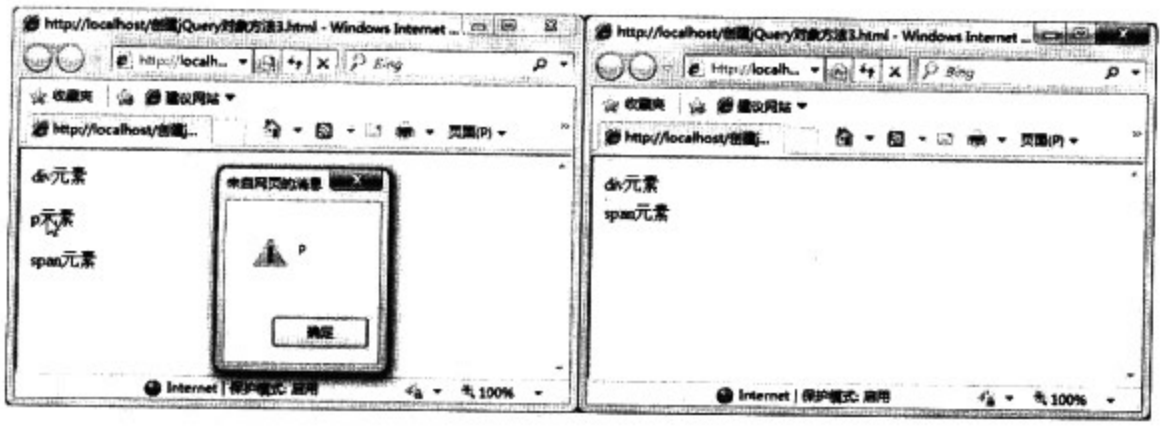


图 8.4 jQuery 方法连写演示效果

8.1.5 使用 extend()方法创建 jQuery 对象方法

jQuery.extend()方法能够创建全局函数，而 jQuery.fn.extend()方法可以创建 jQuery 对象方法。如果读者明白了上一节使用 jQuery.fn 对象属性的方法创建 jQuery 对象方法，那么使用 extend()方法创建 jQuery 对象就比较容易理解了。

例如，针对上节中介绍的示例，我们可以调用 jQuery.fn.extend()方法来创建 jQuery 对象方法。具体代码如下。

```
jQuery.fn.extend({
    test : function(){
        return this.each(function(){
            alert(this.nodeName);
        });
    }
})
```

jQuery.fn.extend()方法仅包含一个参数，该参数是一个对象直接量，是以名/值对形式组成的多个属性，名称表示方法名称，而值表示函数体。因此，在这个对象直接量中可以附加多个属性，为 jQuery 对象同时定义多个方法。

针对上面定义的 test()方法，同样可以在 jQuery 选择器中直接调用。演示效果与图 8.3 相同。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $("body *").click(function(){
        $(this).test();    //调用 jQuery 对象方法
    });
})
</script>

<div>div 元素</div>
<p>p 元素</p>
<span>span 元素</span>
```

8.1.6 创建自定义选择器

jQuery 提供了强大的选择和过滤功能，它完全演绎了 CSS 3.0 选择器的语法规则，使开发者根据 CSS 选择器的使用习惯，以一种惯性思维快速找到所要匹配的 DOM 节点。当然，再强大的工具都会存在缺陷或者不足。对于 jQuery 选择器来说，道理是相同的，所幸的是 jQuery 允许开发人员扩展选择器功能，用户可以根据个人开发需要创建个性化的选择器，以满足特殊选择操作。为了能够轻松创建自己的选择器，我们应该对 jQuery 选择器的工作机制要有所了解。

首先，jQuery 选择器会使用一组正则表达式来分析选择符。然后，针对所解析出来的每一个选择符执行一个函数，这个函数被称为选择器函数。最后，根据这个选择器函数的返回值是否为 true，决定是否保留当前元素，这样就可以找到所要匹配的元素节点。例如，针对下面这个基本选择器，可以选择所有匹配的 p 元素的前面两个元素。

```
$("p:lt(2)")
```

当 jQuery 解析这个选择器时，首先找出当前范围内所有的 p 元素，然后隐式遍历这些 p 元素，并逐个将这些 p 元素作为参数，连同参数 2 都传递给 lt()函数。lt()函数的代码如下。

```
lt: function(elem, i, match){
    return i < match[3] - 0;
},
```

该函数包含三个参数，说明如下。

第一个参数表示当前遍历的 DOM 元素对象。

第二个参数表示当前 DOM 元素对象在所有匹配元素中的索引位置，从 0 开始。

第三个参数表示正则表达式执行匹配后返回的数组对象。相当于 match()方法返回的子表达式所匹配的信息。其正则表达式直接量如下所示。

```
match: {
```

```
    POS: /:(nth|eq|gt|lt|first|last|even|odd)(?:\((\d*)\))?(?=[^-]|$)/
}
```

结合上面的示例代码，match 参数数组的元素组成说明如下。

- match[0]：表示“:lt(2)”部分字符串。
- match[1]：表示选择器引导符，即表示“:”字符。
- match[2]：表示选择器函数，即表示“lt”字符串。
- match[3]：表示选择器函数中的序号参数，即表示“1”字符，它非常有用，在编写选择器函数时将会用到。
- match[4]：表示选择器函数中的特殊参数。在本例中没有体现，例如"p:lt(a(b))"选择器，match[4]就匹配“(b)”字符串部分。

明白了 jQuery 选择器的设计思路，下面我们就模仿这种设计思路创建自定义的选择器。针对上面示例的$("p:lt(2)")选择器，jQuery 还提供了:gt 和:eq 选择器，但是没有定义:ge(大于等于)和:le(小于等于)等选择器。为此，我们将自定义:ge 和:le 选择器。

首先，模仿上面的方法，设计选择器函数，代码如下。

```
le: function(elem, i, match){
    return i < match[3] - 0 || i == match[3] - 0;
},
ge: function(elem, i, match){
    return i > match[3] - 0 || i == match[3] - 0;
},
```

在上面两个函数中，le()通过比较参数 i 的值与匹配元素的序号(match[3])的大小相等关系，决定返回 true 或者 false。通过 match[3] - 0 表达式，强制转换 match[3]值为数值型数据。而 ge()函数正好相反。

然后，把上面的选择器函数添加到 jQuery 选择器对象上即可。在 jQuery 框架中，jQuery.expr[":"]表示 jQuery 选择器对象的别名，它等于 jQuery.expr.filters，又等于 Sizzle.selectors.filters。实现的代码如下。

```
jQuery.expr[":"].le = function(elem, i, match){    //自定义小于等于选择器
    return i < match[3] - 0 || i == match[3] - 0;
}
jQuery.expr[":"].ge = function(elem, i, match){    //自定义大于等于选择器
    return i > match[3] - 0 || i == match[3] - 0;
}
```

最后，我们尝试利用自定义选择器来选择元素并定义样式。在下面的示例中，选择序号等于或者小于 2 的 li 元素，设置它们的字体颜色为红色。选择序号等于或者大于 2 的 li 元素，设置它们的背景颜色为浅灰色，演示效果如图 8.5 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
```

```
<script type="text/javascript">
jQuery.expr[":"].le = function(elem, i, match){
    return i < match[3] - 0 || i == match[3] - 0;
}
jQuery.expr[":"].ge = function(elem, i, match){
    return i > match[3] - 0 || i == match[3] - 0;
}
$(function(){
    $("li:le(2)").css("color","red");    //设置小于等于 2 的 li 元素字体颜色为红色
    $("li:ge(2)").css("background","#ddd");    //设置大于等于 2 的 li 元素背景颜色为灰色
})
</script>

<ul>
    <li>选项 1</li>
    <li>选项 2</li>
    <li>选项 3</li>
    <li>选项 4</li>
     <li>选项 5</li>
</ul>
```

图 8.5 使用自定义选择器选择元素并设置样式

当然，读者也可以使用 jQuery.extend()方法来扩展 jQuery.expr[":"]对象的方法。具体实现的代码如下。

```
jQuery.extend(jQuery.expr[":"],{
    le : function(elem, i, match){
        return i < match[3] - 0 || i == match[3] - 0;
    },
    ge : function(elem, i, match){
        return i > match[3] - 0 || i == match[3] - 0;
    }
})
```

8.1.7 优化 jQuery 默认选择器

除了自定义选择器外，我们也可以对 jQuery 默认选择器进行重写或者优化。重写的方法是直接覆盖该方法。例如，针对:nth-child 选择器来说，它能够匹配指定位置的元素，或者选择奇偶元素。下面的示例可以把所有匹配的 li 元素中第 2 元素设置为红色字体样式。

```
$(function(){
    $("li:nth-child(2)").css("color","red");
})
```

在 CSS3 规范中，:nth-child()伪类选择器的功能是非常强大的，它不仅能够接受整数参数，还可以接受 an+b 形式的任何表达式，如果某项位置的序号等于这个表达式的值，或者等于 n 在任何整数值下计算出的值，这个项就匹配，如 3n+2 表达式，就会匹配 2、5、8 等位置上的元素。下面就根据这个计算原理优化:nth-child 选择器的功能。

要覆盖 nth-child()方法，可以通过以下两种途径实现。

第一，直接覆盖，结构代码如下所示。

```
jQuery.expr[":"].nth-child = function(elem, match){
    //函数体
}
```

但是，考虑到“nth-child”字符串中的减号是运算符，故建议直接使用第二种方法进行定义。

第二，调用 jQuery.extend()方法重写，结构代码如下所示。

```
jQuery.extend(jQuery.expr[":"],{
    "nth-child" : function(elem, match){
        //函数体
    }
})
```

在 jQuery 框架代码中，找到:nth-child()选择器的正则表达式直接量如下所示。

```
match: {
    CHILD: /:(only|nth|last|first)-child(?:\((even|odd|[\dn+-]*)\))?/,
}
```

通过观察分析，我们看到对于:a(b(c))格式的伪类选择符，match 数组中的元素分别表示如下意思。

- match[0]：表示“:a(b(c))”部分字符串。
- match[1]：表示选择器引导符，即表示“:”字符。
- match[2]：表示选择器函数名，即表示“a”字符串。
- match[3]：表示选择器函数参数，即表示“b(c)”字符串。
- match[4]：表示选择器函数中的特殊参数，即表示“(c)”字符串。

然后，在 jQuery 框架代码中，找到 nth-child()函数体的代码如下所示。

```
CHILD: function(elem, match){
    var type = match[1], node = elem;
    switch (type) {
```

```
            case 'nth':
                var first = match[2], last = match[3];
                if ( first == 1 && last == 0 ) {
                    return true;
                }
                var doneName = match[0],
                parent = elem.parentNode;
                if ( parent && (parent.sizcache !== doneName || !elem.nodeIndex) ) {
                    var count = 0;
                    for ( node = parent.firstChild; node; node = node.nextSibling ) {
                        if ( node.nodeType === 1 ) {
                            node.nodeIndex = ++count;
                        }
                    }
                    parent.sizcache = doneName;
                }
                var diff = elem.nodeIndex - last;
                if ( first == 0 ) {
                    return diff == 0;
                } else {
                    return ( diff % first == 0 && diff / first >= 0 );
                }
        }
    },
```

根据上面函数体的设计思路，nth-child()函数的重写代码如下所示。

```
jQuery.extend(jQuery.expr[":"],{
    "nth-child" : function(elem, match){    //按 an+b 公式匹配元素，参数 elem 表示遍历元素，
match 表示匹配返回的数组
        var index = $(elem).parent().children().index(elem) + 1;    //重新计算序号，起始
位置为 1，而不是 0
        var num = match[3].match(/((\d+)n)?\+(\d+)?/);    //使用 match()方法匹配 a(b(c))
格式中的“b(c)”字符串
        if(num[2] == undefined){    //如果 an+b 公式中，a 未知，则直接返回 b 作为序号
            return index == num[3];
        }
        else if(num[3] == undefined){    //如果 an+b 公式中，b 未知，则直接返回 0 作为序号
            num[3] = 0;
        }
        return (index-num[3]) % num[2] == 0;    //最后根据公式测试当前元素是否匹配
    }
})
```

在上面这个函数中，首先从同辈中找到当前节点的索引，由于 CSS3 选择器规定:nth-child()伪选择器的参数序号从 1 开始，故需要重新计算 index 序号值。

然后，根据 an+b 公式，我们反向推导，an+b = y，则 n=(y−b)/a，也就是说某个位置的元素，它的位置序号加 1 之后(即在上面公式中表示变量 y)，如果参与公式计算后，所得到数

值为一个整数，则说明该元素可以匹配。

例如，针对 3n+2 公式来说，当 n 等于 0 时，它可以匹配第 2 个元素；当 n 等于 1 时，它可以匹配第 5 个元素；当 n 等于 2 时，它可以匹配第 8 个元素，依此类推。

覆盖了 jQuery 的:nth-child()选择器之后，下面我们来进行测试。示例代码如下所示，该示例将匹配 3n+2 公式中的所有元素，凡是匹配元素的字体颜色将显示为红色，演示效果如图 8.6 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery.extend(jQuery.expr[":"],{
    "nth-child" : function(elem, match){    //按 an+b 公式匹配元素，参数 elem 表示遍历元素，
match 表示匹配返回的数组
        //省略函数体，请参阅上面代码
    }
})
$(function(){
    $("li:nth-child(3n+2)").css("color","red");    //匹配 3n+2 位置上的元素
})
</script>

<ul>
    <li>选项 1</li>
    <li>选项 2</li>
    <li>选项 3</li>
    <li>选项 4</li>
    <li>选项 5</li>
    <li>选项 6</li>
    <li>选项 7</li>
    <li>选项 8</li>
</ul>
```

图 8.6 使用优化后的:nth-child()选择器选择元素

8.1.8 封装 jQuery 插件

上面几节就 jQuery 插件的创建方法进行详细讲解，一般对外发布的自定义插件都应该进

行封装，封装的插件还应该符合规范，只有这样所创建的插件才具有推广价值，并得到其他用户的喜爱。

封装 jQuery 插件的第一步是定义一个独立域，代码如下所示。

```
(function($){
    //自定义插件代码
})(jQuery)      //封装插件
```

确定创建插件类型，选择创建方式。例如，创建一个设置元素字体颜色的插件，则应该创建 jQuery 对象方法。考虑到 jQuery 提供了插件扩展方法 extend()，调用该方法定义插件会更为规范。代码如下。

```
(function($){
    $.extend($.fn,{     //jQuery 对象方法扩展
        //函数列表
    })
})(jQuery)     //封装插件
```

一般插件都会接受参数，用来控制插件的行为，根据 jQuery 设计习惯，我们可以把所有参数以列表形式封装在选项对象中进行传递。例如，对于设置元素字体颜色的插件，应该允许用户设置字体颜色，同时还应考虑如果用户没有设置颜色，则应确保使用默认色进行设置。实现代码如下所示。

```
(function($){
    $.extend($.fn,{     //jQuery 对象方法扩展
        color : function(options){     //自定义插件名称
            var options = $.extend({     //参数选项对象处理
                bcolor : "white",     //背景色默认值
                fcolor: "black"     //前景色默认值
            },options);
            //函数体
        }
    })
})(jQuery);     //封装插件
```

最后，设计插件自定义功能代码，如下所示。

```
(function($){
    $.extend($.fn,{
        color : function(options){     //自定义插件名称
            var options = $.extend({     //参数选项对象处理
                bcolor : "white",     //背景色默认值
                fcolor : "black"     //前景色默认值
            },options);
            return this.each(function(){     //返回匹配的 jQuery 对象
                $(this).css("color", options.fcolor);     //遍历设置每个 DOM 元素的字体颜色
                $(this).css("backgroundColor", options.bcolor);     //遍历设置每个 DOM 元素
```

```
//的背景颜色
            })
        }
    })
})(jQuery);     //封装插件
```

完成插件封装之后，我们不妨来测试一下自定义的 color()方法。代码如下，演示效果如图 8.7 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
//省略插件定义，请参阅上面代码
$(function(){    //页面初始化
    $("h1").color({    //设置标题的前景色和背景色
        bcolor : "#eea",
        fcolor : "red"
    });
})
</script>

<h1>标题文本</h1>
```

图 8.7　封装 jQuery 插件

8.1.9　优化 jQuery 插件——开放公共参数

优秀的 jQuery 插件，应该以开放性的姿态满足不同个性化的设计要求，同时还应做好封闭性，避免外界有意或无意的破坏。

首先，可以考虑开放插件的默认设置，这对于插件使用者来说，会更容易使用较少的代码覆盖和修改插件。

继续以上面的代码为例进行说明，把其中的参数默认值作为$.fn.color 对象的属性单独进行设计，然后借助 jQuery.extend()方法覆盖原来参数选项即可。

```
(function($){
    $.extend($.fn,{
        color : function(options){
            var options = $.extend({}, $.fn.color.defaults, options);    //覆盖原来参数
            return this.each(function(){
```

```
            $(this).css("color", options.fcolor);
            $(this).css("backgroundColor", options.bcolor);
        })
    }
})
$.fn.color.defaults = {    //独立设置$.fn.color 对象的默认参数值
    bcolor : "white",
    fcolor : "black"
}
})(jQuery);
```

在 color()函数中，jQuery.extend()方法能够使用参数 options 覆盖默认的 defaults 属性值，如果没有设置 options 参数值，则使用 defaults 属性值。同时，由于 defaults 属性是单独定义的，故我们可以在页面中预设前景色和背景色，然后就可以多次调用 color()方法，示例代码如下，演示效果如图 8.8 所示。通过这种开发插件默认参数的做法，用户不再需要重复定义参数，这样就可以节省开发时间。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
//省略插件定义，请参阅上面代码
$(function(){
    $.fn.color.defaults = {    //预设默认的前景色和背景色
        bcolor : "#eea",
        fcolor : "red"
    }
    $("h1").color();    //为标题 1 设置默认色
    $("p").color({bcolor:"#fff"});    //为段落文本设置默认色，同时覆盖背景色为白色
    $("div").color();    //为盒子设置默认色
})
</script>

<h1>标题文本</h1>
<p>段落文本</p>
<div>盒子</div>
```

图 8.8　开发 jQuery 插件的默认参数设置

8.1.10 优化 jQuery 插件——开放部分功能

用过 Cycle 插件的读者可能会知道，它是一个滑动显示插件，支持很多内部变换功能，如滚动、滑动和渐变消失等。实际上，在封装插件时，我们无法把所有功能都封装进去，也没有办法定义滑动变化上每一种类型的变化效果。但是 Cycle 插件通过开放部分功能，允许用户重写 transitions 对象，这样就可以添加自定义变化效果，从而使该插件满足不同用户的不同需求。

Cycle 插件是这样开放部分功能的，代码如下。

```
$.fn.cycle.transitions = {
    //扩展方法
};
```

这个技巧就可以允许其他用户定义和传递参数到 Cycle 插件内部。

例如，继续以上一节的示例为基础，我们为其添加一个格式化的扩展功能，这样用户在设置颜色的同时，还可以根据需要适当进行格式化功能设计，如加粗、斜体、放大等功能操作。扩展的 color()插件代码如下所示。

```
(function($){
    $.extend($.fn,{
        color : function(options){
            var options = $.extend({}, $.fn.color.defaults, options);    //覆盖原来参数
            return this.each(function(){
                $(this).css("color", options.fcolor);
                $(this).css("backgroundColor", options.bcolor);
                var _html = $(this).html();        //获取当前元素包含的 HTML 字符串
                _html = $.fn.color.format(_html);     //调用格式化功能函数对其进行格式化
                $(this).html(_html);              //使用格式化的 HTML 字符串重写当前元素内容
            })
        }
    })
    $.fn.color.defaults = {    //独立设置$.fn.color 对象的默认参数值
        bcolor : "white",
        fcolor : "black"
    }
    $.fn.color.format = function(str){    //开放的功能函数
        return str;
    };
})(jQuery);
```

在上面的示例中，通过开放的方式定义了一个 format()功能函数，在这个功能函数中默认没有进行格式化设置，然后在 color()函数体内利用这个开放性功能函数格式化当前元素内的 HTML 字符串。

例如，下面的示例调用了 color()插件，同时在调用时分别扩展了它的格式化功能，演示效果如图 8.9 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
//省略插件定义，请参阅上面代码
$(function(){
    $.fn.color.defaults = {    //预设默认的前景色和背景色
        bcolor : "#eea",
        fcolor : "red"
    }
    $.fn.color.format = function(str){    //扩展 color()插件的功能，使内部文本加粗显示
        return "<strong>" + str + "</strong>";
    }
    $("h1").color();
    $("p").color({bcolor:"#fff"});
    $.fn.color.format = function(str){    //扩展 color()插件的功能，使内部文本放大显示
        return "<span style='font-size:30px;'>" + str + "</span>";
    }
    $("div").color();
})
</script>

<h1>标题文本</h1>
<p>段落文本</p>
<div>盒子</div>
```

图 8.9　开放的 color()插件

上述技巧让用户能够传递自己的功能设置，以覆盖插件默认的功能，从而方便了其他用户以当前插件为基础进一步去扩写插件。

8.1.11　优化 jQuery 插件——保留插件隐私

优秀的插件，不仅仅要追求开放性，还应该留意插件的隐私性，对于不该暴露的部分，如果不注意保护，很容易被外界入侵，从而破坏插件的功能。因此，在设计插件时必须考虑

插件实现中不应该暴露的部分。一旦被暴露，就需要铭记任何对于参数或者语义的改动也许会破坏向后的兼容性。如果不能确定不应该暴露的特定函数，那么就必须考虑如何进行保护的问题。

若插件包含很多函数，在设计时我们希望这么多函数不搅乱命名空间，也不会被完全暴露，惟一的方法就是使用闭包。为了创建闭包，可以将整个插件封装在一个函数中。

继续以上节示例进行讲解，为了验证用户在调用 color()方法时所传递的参数是否合法，我们不妨在插件中定义一个参数验证函数，但是该验证函数是不允许外界侵入或者访问的，此时我们可以借助闭包把它隐藏起来，只允许在插件内部进行访问。实现的代码如下。

```
(function($){
    $.extend($.fn,{
        color : function(options){
            if(!filter(options))    //调用隐私方法验证参数，不合法则返回
                return this;
            var options = $.extend({}, $.fn.color.defaults, options);
            return this.each(function(){
                $(this).css("color", options.fcolor);
                $(this).css("backgroundColor", options.bcolor);
                var _html = $(this).html();          //获取当前元素包含有 HTML 字符串
                _html = $.fn.color.format(_html);    //调用格式化功能函数对其进行格式化
                $(this).html(_html);                 //使用格式化的 HTML 字符串重写当前元素内容
            })
        }
    });
    $.fn.color.defaults = {//省略函数体代码};
    $.fn.color.format = function(str){ //省略函数体代码};
    function filter(options){    //定义隐私函数，外界无法访问
        //如果参数不存在，或者存在且为对象，则返回 true，否则返回 false
        return !options || (options && typeof options === "object")?true : false;
    }
})(jQuery);
```

这样对于下面的非法参数设置，则会忽略该方法的调用，但是不会抛出异常。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
//省略插件定义，请参阅上面代码
$(function(){
    $("p").color("#fff");
})
</script>

<p>段落文本</p>
```

8.1.12 优化 jQuery 插件——非破坏性操作

在特定情况下，jQuery 对象方法可能会修改 jQuery 对象匹配的 DOM 元素，这时就有可能破坏方法返回值的一致性。为了遵循 jQuery 框架的核心设计理念，我们应该时刻警惕任何修改 jQuery 对象的操作。

例如，定义一个 jQuery 对象方法 parent()，用来获取 jQuery 匹配的所有 DOM 元素的父元素。实现代码如下。

```
(function($){
    $.extend($.fn,{
        parent : function(options){    //扩展 jQuery 对象方法，获取所有匹配元素的父元素
            var arr = [];
            $.each(this, function(index, value){   //遍历匹配的 DOM 元素
                arr.push(value.parentNode);        //把匹配元素的父元素推入临时数组
            });
            arr = $.unique(arr);           //在临时数组中过滤重复的元素
            return this.setArray(arr);     //把变量 arr 打包为数组类型返回
        }
    })
})(jQuery);
```

在上面的 jQuery 对象方法中，通过遍历所有匹配元素，获取每个 DOM 元素的父元素，并把这些父元素存储到一个临时数组中，通过过滤、打包再返回。

下面我们就用这个新方法为所有 p 元素的父元素添加一个边框，示例代码如下所示，演示效果如图 8.10 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
//省略 jQuery 对象 parent()方法定义
$(function(){
    var $p = $("p");    //获取所有 p 元素，并存储到变量$p 中
    $p.parent().css("border","solid 1px red");    //调用 parent()方法获取 p 元素的父元素，并设置它们的边框样式为 1 像素宽的红色实线
})
</script>

<div style="width:400px;height:200px;">大盒子
    <p>段落文本 1</p>
    <div style="width:200px;height:100px;">小盒子
        <p>段落文本 2</p>
    </div>
</div>
```

如果在设置了父元素的边框后，我们希望把 jQuery 对象匹配的所有元素都隐藏起来，则

可以添加下面的代码，则在浏览器中预览就会发现 div 元素也被隐藏起来了，如图 8.11 所示。

```
$(function(){
    var $p = $("p");    //获取所有 p 元素，并存储到变量$p 中
    $p.parent().css("border","solid 1px red");
    $p.hide();    //隐藏所有 p 元素，即当前 jQuery 对象
})
```

也就是说，在上面代码中**$p** 变量已经被修改，它不再指向当前 **jQuery** 对象，而是指向 jQuery 对象匹配元素的父元素，因此在为**$p** 调用 **hide()**方法时，就会隐藏 div 元素，而不是 p 元素。

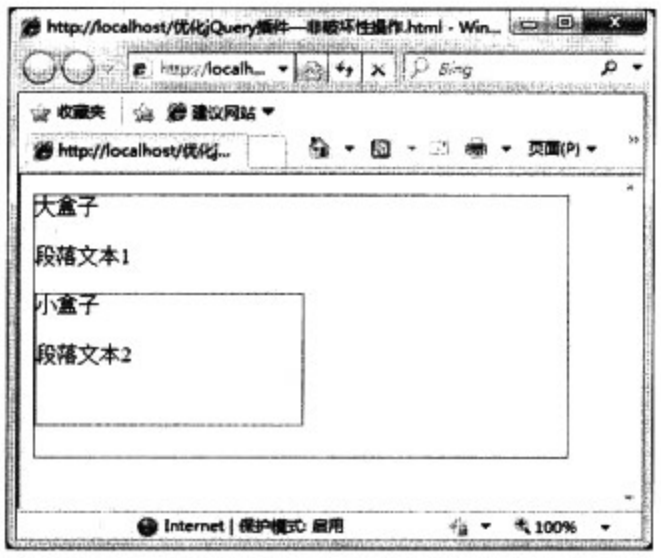


图 8.10　调用 parent()自定义方法

图 8.11　div 元素被隐藏起来

上面示例仅仅是破坏性操作的一种表现，如果要避免此类隐性修改 jQuery 对象的行为，建议采用非破坏性操作。例如，在本例中我们可以使用 **pushStack()**方法创建一个新的 jQuery 对象，而不是修改 this 所引用的 jQuery 对象，这样可以避免破坏性操作行为，同时 pushStack()方法还允许调用 **end()**方法操作新创建的 jQuery 对象方法。把上面示例的 jQuery 对象方法进行优化，代码如下所示。

```
(function($){
    $.extend($.fn,{
        parent : function(options){ //扩展 jQuery 对象方法，获取所有匹配元素的父元素
            var arr = [];
            $.each(this, function(index, value){ //遍历匹配的 DOM 元素
                arr.push(value.parentNode);        //把匹配元素的父元素推入临时数组
            });
            arr = $.unique(arr);                    //在临时数组中过滤重复的元素
            return this.pushStack(arr);      //返回新创建的 jQuery 对象，而不是修改后的当前
jQuery 对象
        }
    })
})(jQuery);
```

这时，如果继续执行上面的演示示例操作，则可以看到 **div** 元素边框样式被定义为红色

实现了，同时也隐藏了其包含的 p 元素，演示效果如图 8.12 所示。

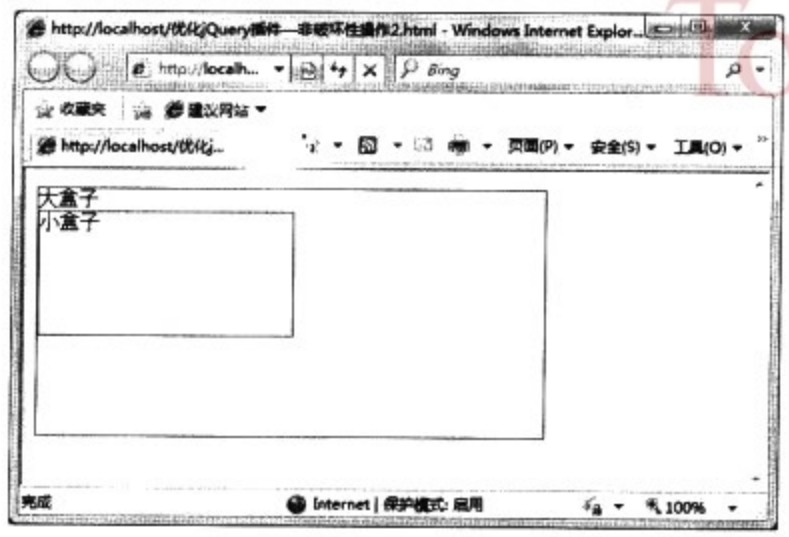


图 8.12　使用非破坏性的 parent()方法效果

针对上面的代码，我们就可以采用连续行为进行编写了，代码如下所示。

```
$(function(){
    var $p = $("p");
    $p.parent().css("border","solid 1px red").end().hide();
})
```

其中 end()方法能够恢复被破坏的 jQuery 对象，也就是说 parent()方法返回的是当前元素的父元素的集合，现在调用 end()方法之后，又恢复到最初的当前元素集合，此时可以继续调用方法作用于原来的 jQuery 对象上了。

在 jQuery 1.0 版本中，与 children()之类的 DOM 遍历方法都具有破坏性，从 jQuery 1.1 版本之后，这个错误才得以修正。

8.1.13　优化 jQuery 插件——添加事件日志

在传统开发中，软件都包含有事件日志，这样就可以在事件发生时或发生后进行跟踪。在 JavaScript 程序调试中，我们常常使用 alert()方法来跟踪进程，但是这种做法影响了程序的正常流程，不符合频繁、实时显示事件信息。我们可以模仿其他软件中的调试台 log()函数，借助这个函数将事件日志信息输出到独立的日志文件中，从而避免中断页面交互进程。

首先，我们为 jQuery 对象添加一个全局函数 log()。在这个函数中，将把发生的事件信息写入事件日志包含框中。实现代码如下。

```
jQuery.log = function(msg){
    var html = jQuery('<div class="log"></div>').text(msg);
    jQuery(".logbox").append(html);
}
```

然后，在事件中调用该日志方法，从而实时跟踪事件发生时的事件，实现代码如下，演

示效果如图 8.13 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
//省略 jQuery.log()函数的定义，请参阅上面的代码
$(function(){
    $("input").click(function(event){
            var e = event.type;
            $.log(e);
    });
    $("input").mouseover(function(event){
            var e = event.type;
            $.log(e);
    });
    $("input").mouseout(function(event){
            var e = event.type;
            $.log(e);
    });
})
</script>

<input type="button" value="提交按钮" />
<div class="logbox"></div>
```

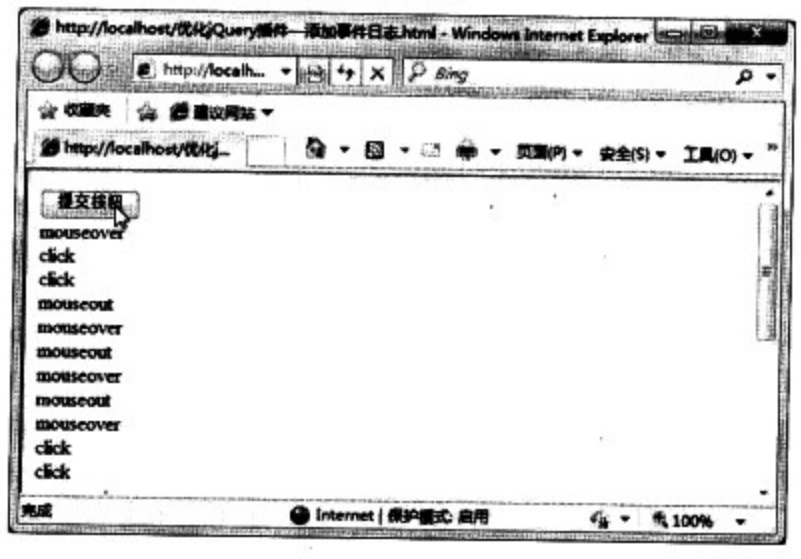


图 8.13 jQuery 事件日志跟踪

但是，在一个页面中往往会包含很多事件示例，如果分别进行记录，会非常不方便，为此可以定义一个对象方法，而不是使用全局函数。代码如下所示。

```
(function($){
   $.extend($.fn,{
      log : function(msg){
         var html = jQuery('<div class="log"></div>').text(msg);
         return this.each(function(){
            jQuery(".logbox").append(html);
         });
      }
   })
```

```
})(jQuery);
```

然后在实例中调用该日志方法，代码如下，演示效果如图 8.14 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
//省略 log()对象方法的定义，请参阅前面的代码
$(function(){
    $("h1").click(function(event){
            var e = event.type;
            $(this).log(this.nodeName.toLowerCase() + "." + e);
    });
    $("p").mouseover(function(event){
            var e = event.type;
            $(this).log(this.nodeName.toLowerCase() + "." + e);
    });
    $("input").mouseout(function(event){
            var e = event.type;
            $(this).log(this.nodeName.toLowerCase() + "." + e);
    });
})
</script>

<h1>标题文本</h1>
<input type="button" value="提交按钮" />
<p>段落文本</p>
<div class="logbox"></div>
```

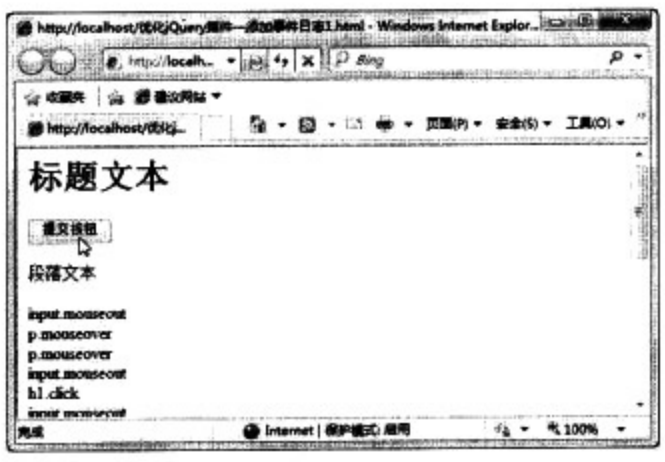


图 8.14　改进 jQuery 事件日志跟踪

我们还可以进一步改善 log()日志方法的灵活度，使其自动搜索最近显示日志信息的元素，通过利用该方法的语境，我们可以在遍历 DOM 元素时找到距离最近的日志元素。实现代码如下。

```
(function($){
    $.extend($.fn,{
        log : function(msg){
            return this.each(function(){
                var $this = $(this);                    //获取当前元素
                while($this.length){                    //如果存在当前元素
```

```
                var $logbox = $this.find(".logbox");  //在当前元素内搜索是否存在日志元素
                if($logbox.length){          //如果存在日志元素
                    var html = jQuery('<div class="log"></div>').text(msg);
                    $logbox.append(html);
                    break;                   //跳出检索
                }
                $this =  $this.parent();     //检索上一级匹配元素
            }
        });
    }
  })
})(jQuery);
```

最后，我们还可以改善参数的处理机制，考虑到log()方法只能够简单地接受字符串型信息，如果要向log()方法传递更多的信息，就会变得无能为力。遵循jQuery框架方法的一贯设计思想，我们可以考虑允许用户以对象列表的形式向log()方法传递更多甚至无限制的信息。因此，我们还需对log()方法中的参数处理机制进行改善。如果用户向其传入对象类型的参数，则直接调用它，将会显示“[object object]”的字符串，显然这并不是我们所希望的日志信息。

```
(function($){
   $.extend($.fn,{
      log : function(msg){
         if(typeof msg == "object"){          //如果参数为对象类型，则解析该对象包含的信息
            var str = "{ ";
            $.each(msg, function(name, value){ //遍历对象成员
               str += name + " : " + value + ", ";
            });
            str = str.substring(0,str.length-2);    //清除最后一个成员的逗号
            str += " }";
            msg = str;                       //把解析的对象信息返给参数变量
         }
         return this.each(function(){
            var $this = $(this);                    //获取当前元素
            while($this.length){                 //如果存在当前元素
               var $logbox = $this.find(".logbox");  //在当前元素内搜索是否存在日志元素
               if($logbox.length){               //如果存在日志元素
                  var html = jQuery('<div class="log"></div>').text(msg);
                  $logbox.append(html);
                  break;                         //跳出检索
               }
               $this =  $this.parent();     //检索上一级匹配元素
            }
         });
      }
   })
})(jQuery);
```

这样，我们就可以在log()方法中传入更多的信息，当然也可以直接传入字符串信息，应

用示例如下所示，演示效果如图 8.15 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
//省略的 log()方法，请参阅前面的代码
$(function(){
    $("h1").mouseout(function(event){
        $(this).log({
            nodeName : this.nodeName.toLowerCase(),
            eventType : event.type
        });
    });
    $("p").mouseover(function(event){
        $(this).log({
            nodeName : this.nodeName.toLowerCase(),
            eventType : event.type
        });
    });
    $("input").click(function(event){
        var e = event.type;
        $(this).log(this.nodeName.toLowerCase() + "." + e);
    });
})
</script>

<h1>标题文本</h1>
<input type="button" value="提交按钮" />
<p>段落文本</p>
<div class="logbox"></div>
```

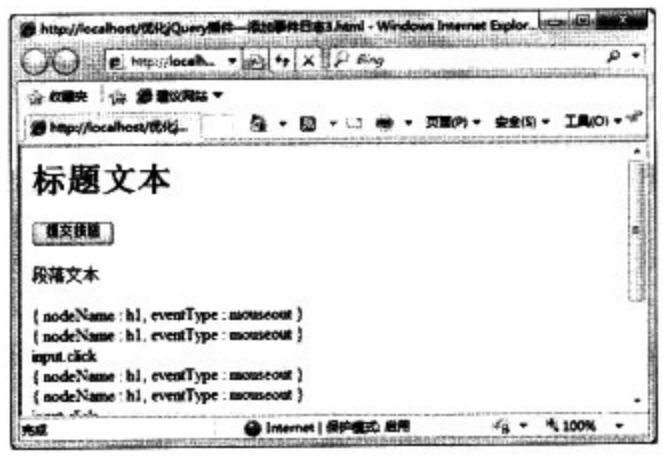


图 8.15　完善 log()日志方法的参数处理

8.1.14　创建 jQuery 插件应注意的问题

由于插件结构简单，创建比较容易，对于初学者来说它简单易学，对于高手来说它很灵活，因此网上共享的 jQuery 插件数量急速上升。另外，jQuery 允许使用多种风格创建插件，如何遵守公共规则，是很多初学者必须注意的问题。

jQuery 开发团队制定了通用规则，为插件用户创建了一个通用而可信的环境。因此，建议读者在创建插件中遵守这些规则，确保自定义的插件与其他代码和平相处，并获得广大用户的认可。

创建 jQuery 插件应注意以下几个问题。

1. 命名规则

如果希望用户查看文件时能立即知道这是一个什么插件，统一文件名称是必须的。我们可以遵循这样的命名规则：jquery. plug-in_name.js。

其中 plug-in_name 表示插件的名称，在这个文件中，所有全局函数都应该包含在名为 plug-in_name 的对象中。如果插件只有一个函数，则可以考虑使用 jQuery. plug-in_name()形式命名。

插件中的对象方法可以灵活命名，但是应保持相同的命名风格。如果定义多个方法，建议在方法名前添加插件名前缀，以保持清晰。不建议使用过于简短的名称，或者语义含糊的缩写名，或者公共方法名，如 set()、get()等，这样很容易与外界的方法混淆。

2. 基本思想

所有新方法都附加到 jQuery.fn 对象上，所有新功能都附加到 jQuery 对象上。这是实际编码过程中最重要的规则。

3. 方法内的 this 关键字

在插件方法中，this 关键字用于引用 jQuery 对象，这有利于插件编写，它让所有插件在引用 this 时，知道从 jQuery 接收到哪个对象。所有 jQuery 方法都是在一个 jQuery 对象的环境中调用的，因此函数体中 this 关键字总是指向该函数的上下文，即 this 此时是一个包含多个 DOM 元素的数组。

4. 迭代匹配元素

使用 this.each()迭代匹配的元素，这是一种可靠而有效地迭代对象的方式。

出于性能和稳定性考虑，推荐所有的方法都使用它迭代匹配的元素。无论 jQurey 对象实际匹配的元素有多少，所有方法都必须以适当方式运行，一般来说，应该调用 this.each()方法来迭代所有匹配的元素，然后再依次操作每个 DOM 元素。

注意：在 this.each()方法体内，this 关键字就不再引用 jQuery 对象，而是引用当前匹配的 DOM 元素对象。

5. 方法返回值

所有方法都应该返回一个值，除了特定需求方法外，所有方法都必须返回 jQuery 对象。

除了需要方法返回计算值或者某个特定对象等，一般方法都应该返回当前上下文环境中的 jQuery 对象，即 this 关键字引用的数组。通过这种方式，可以保持 jQuery 框架内方法的连续行为，即方法连缀写法。jQuery 方法的顺序链非常著名，如果编写打破链条的插件，它就会给用户开发带来诸多不方便。

如果匹配的对象集合被修改，则应该通过调用 pushStack()方法创建新的 jQuery 对象，并返回这个新对象，如果返回值不是 jQuery 对象，则应该明确说明。

6. 方便压缩

插件中定义的所有方法或函数，在末尾都必须加上分号(即;)，以方便代码压缩。压缩 JavaScript 文件是最佳实践，若大于最小值会让自定义插件很快就被用户抛弃。

7. jQuery 和$有区别

在插件代码中总是使用“jQuery”，而不是“$”。$并不总是等于 jQuery，这个很重要，如果用户使用 var JQ = jQuery.noConflict();函数更改 jQuery 别名，那么就会引发错误，另外其他 JavaScript 框架也可能使用$别名。

在复杂的插件中，如果全部使用“jQuery”代替“$”，又会让人难以接受这种复杂的写法，为了解决这个问题，建议使用如下插件模式。

```
(function($){
    //在插件包中使用$代替 jQuery
})(jQuery);
```

这个包装函数接受一个参数，该参数传递的是 jQuery 全局对象，由于参数被命名为$，因此在函数体内就可以安全使用$别名，而不用担心命名冲突。

上述这些规则在插件代码中都必须遵守，如果不遵守这些插件规则，那么自己开发的插件就得不到广泛应用和推广。因此，遵守这些规则非常重要，它不仅能保证插件代码的统一性，还能增加插件的成功几率。

8.2 创建 jQuery 插件实战

插件并不神秘，设计一个好的插件需要智慧，当你掌握了 jQuery 插件创建的一般方法之后，下面你所面临的挑战不是如何创建插件，而是如何巧做插件。当然，插件并不是巨大、功能庞杂才是好东西。好的插件应该能切实解决开发人员在实践中遇到的挠头小事儿，也许稍稍迈下台阶就会越过，如果你的插件能够帮助这些挠头的开发者提高工作效率，应该说这样的插件才是最棒的。

所以，本节列举了几个小巧的 jQuery 插件，来帮助读者理解插件的创建过程，以及如何

去设计好的、实用的 jQuery 插件。

8.2.1 简化式插件设计

在 jQuery 框架中，我们可以看到很多功能相近，但用法繁简不一的方法。例如，ajax()方法能够解决所有异步通信问题，但是 jQuery 在 ajax()方法的基础又定义了 load()、get()、getJSON 和 getScript()等方法，这就是一种基于特定方法上的简化式插件设计方法。类似的还有 bind()方法，以及作者为每一种事件类型单独定义的绑定方法，如 click()、mouseover()等方法。

如果发现某段代码需要多次重复使用，或者需要多次重复调用某个方法，不妨考虑在这个方法基础上创建一种简写形式，或者挑选出对项目开发有用的方法，并省略掉 jQuery 中那些无关或者繁琐的方法。这样就可以提高工作效率，优化代码结构。

animate()方法是 jQuery 动画的基础，很多方法都是从该方法延伸出来的。在设计动画时，我们经常会把滑动显示和隐藏与渐显和渐隐动画混合在一起设计。下面这个示例就是直接使用 animate()方法进行设计。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    $("input").eq(0).click(function(){
        $("div").animate({    //淡出收起
            height :"hide",
            opacity :"hide"
        },"slow");
    })
    $("input").eq(1).click(function(){
        $("div").animate({    //淡入展开
            height :"show",
            opacity :"show"
        },"slow");
    })
})
</script>

<input type="button" value="渐隐收起" /><input type="button" value="渐显展开" />
<div  style="height:200px;  width:300px;  background-color:blue;  border:solid  1px
red;"></div>
```

现在把上面这个简单的动画演示功能封装起来，定义为 jQuery 对象方法。为了与 jQuery 默认的简写动画方法保持一致，故设计这两个方法的参数分别为自定义速度和回调函数，详细代码如下所示。

```
(function($){
```

```
    $.extend($.fn,{
        showIn : function(speed, fn){
            return this.animate({
                height :"show",
                opacity :"show"
            },speed,fn);
        },
        hideOut : function(speed, fn){
            return this.animate({
                height :"hide",
                opacity :"hide"
            },speed,fn);
        }
    })
})(jQuery);
```

在上面的代码中，this 关键字引用当前上下文的 jQuery 对象，故可以直接调用 jQuery 自定义的 animate()动画方法。然后，就可以直接为 div 元素调用 showIn()和 hideOut()方法。详细代码如下所示。

```
$(function(){
    $("input").eq(0).click(function(){
        $("div").hideOut(4000);
    })
    $("input").eq(1).click(function(){
        $("div").showIn(4000);
    })
})
```

8.2.2 定宽输出插件

在狭窄的边栏中经常会遇到显示超长的列表选项，遇到这类问题时，不是文本挤出或者撑开边栏框，就是换行显示，从而破坏边栏选项之间的均匀显示。解决此类问题的方法是：定义定宽输出函数，把多出的字符替换为省略号，或者直接截取掉。下面来定义一个 jQuery 全局函数 fixedWidth()，解决定宽输出问题。

技术难点：字符包括单字节和双字节，定宽是根据字符数确定，还是根据字节数确定？

如果根据字符串的 length 属性直接进行定宽输出，也就是说根据字符数处理，可能会存在英文字符和汉字排列长短不一的问题。应该说以单字节为单位进行宽度计算会更为严谨，能够适应不同类型字符的处理。而 JavaScript 没有提供直接计算字符串的字节长度。所以，在本插件中，需要定义两个内部函数，用来计算字符串的字节数，并用字节数来截取字符串。

参数：

(1) str：字符串类型，将被格式化为等宽的原字符串。

(2) length：数值类型，设置等宽字符串的长度，单位是字节，而不是字符。注意，英文字符为单字节，而汉语字符为双字节字符。

(3) char：字符串类型，可选参数，默认值为“..”，是指定截取字符串之后，添加的后缀字符串。

返回值：返回被格式化的等宽字符串。

在定宽输出函数 fixedWidth()中，包含两个私有函数：substringB()和 lengthB()，说明如下。

1) substringB()

以指定的字节数截取字符串。

参数：

- str：字符串类型，将被截取的字符串。
- length：数值类型，截取字符串的长度，单位是字节，而不是字符。注意，英文字符为单字节，而汉语字符为双字节字符。

返回值：返回指定字节数的截取字符串，从原字符串第一个字符开始截取。

2) lengthB()

返回值：返回指定字符串的字节长度。

参数：

str：字符串类型，被计算的字符串。

返回值：返回指定字符串的字节数。

该全局函数的详细代码如下。

```
(function($){
   $.extend($,{
      fixedWidth : function(str, length, char){
         str = str.toString();    //把参数转换为字符串
         if(!char) char = "...";    //如果没有设置 char 参数，则设置默认值
         var num = length - lengthB(str);     //获取字符串的字节数与指定长度的差值
         if(num < 0 ){     //如果超过指定宽度
            str = substringB(str, length - lengthB(char) ) + char ;     //按字节数截
取字符串，并附加后缀，因为要添加后缀，故需要去除后缀字符的字节数
         }
         return str;     //返回等宽字符串
            //按字节数截取字符串
         function substringB(str, length){
            var num = 0, len = str.length, tenp = "";
            if( len ){     //如果存在字符串
               for( var i = 0; i < len; i ++ ){     //遍历字符串
                  if(num > length) break;     //如果超过指定宽度，则跳出循环
```

```
                if(str.charCodeAt( i ) > 255 ){      //如果是双字节字符
                    num += 2;      //则增加两个字节数
                    tenp += str.charAt( i );      //叠加字符
                }else{
                    num ++ ;                    //则增加一个字节数
                    tenp += str.charAt( i );      //叠加字符
                }
            }
            return tenp;      //返回截取的字符串
        }else{     //如果不存在字符串，则返回 null
            return null;
        }
    }
    //获取字符串的字节数
    function lengthB(str){
        var num = 0, len = str.length;
        if( len ){      //如果存在字符串
            for( var i = 0; i < len; i ++ ){      //遍历字符串
                if(str.charCodeAt( i ) > 255 ){      //如果是双字节字符
                    num += 2;      //则增加两个字节数
                }else{
                    num ++ ;      //否则增加一个字节数
                }
            }
            return num;      //返回字符串的字节数
        }else{     //如果不存在字符串，则返回 0
            return 0;
                }
            }
        }
    })
})(jQuery);
```

下面就来调用 fixedWidth()全局函数设计一个等宽的新闻列表栏目。该新闻栏宽度固定为 240 像素，每条新闻列表单行显示。在没有调用 fixedWidth()函数处理之前，则整个栏目显示效果如图 8.16 所示。可以看到，新闻列表变得参差不齐，影响新闻浏览效果。如果在脚本中使用 fixedWidth()函数把每条新闻先格式化之后再显示，则整个栏目看起来更加符合浏览习惯，如图 8.17 所示。调用代码如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
//省略 jQuery.fixedWidth()工具函数，请参见上面的代码
$(function(){
    var $a = $("li a");
    $a.each(function(){     //遍历每条新闻
        var str = $(this).text();     //获取新闻信息
        str = jQuery.fixedWidth(str, 30);     //定宽截取，设置为 30 个字节
        $(this).text(str);     //输出显示等宽字符串
    });
```

```
})
</script>

<ul style="width:240px; border:solid 1px red; padding-left:1.5em;">
    <li><a href="#">广西人社厅：2010年广西公务员考试考题泄露 18:45</a></li>
    <li><a href="#">广东今年普通高校招生录取首次实施平行志愿 09:25</a></li>
    <li><a href="#">教师绩效改革标准制定难 09:31</a></li>
    <li><a href="#">Top US Military Officer: Let Gays Serve Openly 11:20</a></li>
    <li><a href="#">Some US Students Learn Mandarin With China's Help13:43</a></li>
    <li><a href="#">A Rough Road for Toyota 17:14</a></li>
</ul>
```

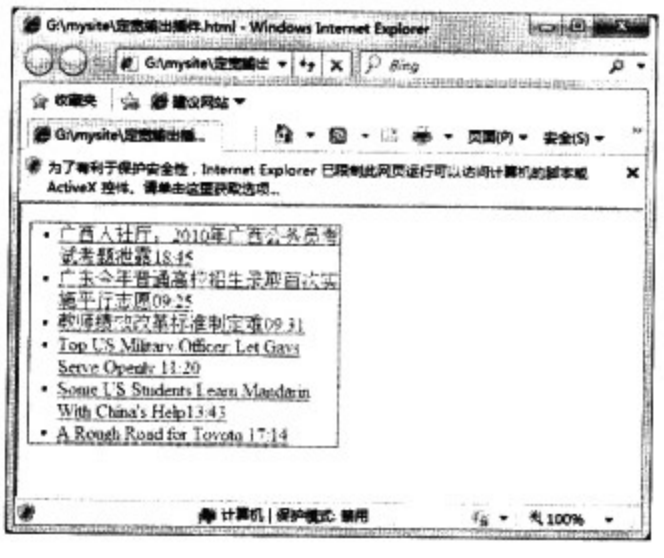


图 8.16　非等宽输出效果

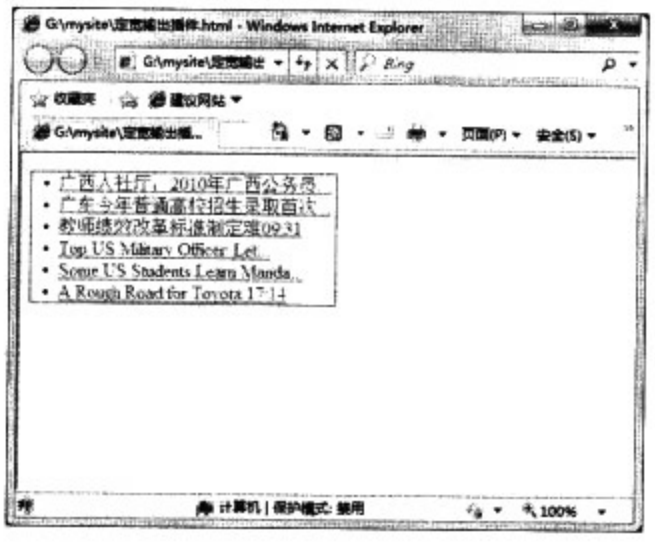


图 8.17　等宽输出效果

8.2.3　Tab 选项卡插件

Tab 选项卡是常用的 UI 组件，在页面中经常见到这种 Web 应用形式。虽然网上有很多类似 jQuery 的插件，但是功能强弱不同，且未必适合个人设计需要，在 jQuery 插件设计中，除了一些经典的插件，只有自己设计的，才最适合自己。而且当遇到需求变化时，也方便自己去修改。

1. 功能描述

本 Tab 选项卡插件具有强大的功能，主要功能说明如下。

- Tab 选项个数可以任意设置。
- 可禁用指定 Tab 选项。
- 可以绑定 Tab 事件触发函数。
- 可以定制 Tab 切换动画显示效果。
- 支持三种回调函数，方便 Ajax 异步交换数据，包括 callBackStartEvent、callBackHideEvent 和 callBackShowEvent，分别在单击 Tab 选项时立即执行、当显示的 Tab 选项的内容完全隐藏时执行、当单击 Tab 选项后内容完全显示时执行，回调函数默认包含一个当前 Tab 选项标签序号的参数。

2. 参数

本插件在初始化时可以设置参数，参数为一个散列结构的对象，该对象包含下面这些选项列表。

- tabList：字符串类型，指定 Tab 选项卡标题包含框的类名。
- contentList：字符串类型，指定 Tab 选项卡内容框的类名。
- tabActiveClass：字符串类型，指定激活选项卡的类名。
- tabDisableClass：字符串类型，指定要禁用选项卡的禁用类名。
- eventType：字符串类型，指定触发事件类型，包括 click 和 mouseover 两种类型。
- showType：字符串类型，设置/显示方式，"show "表示直接显示，"fade "表示渐变显示，"slide "表示滑动显示。
- showSpeed：数值类型，表示/显示速度，单位为毫秒。
- callBackStartEvent：回调函数，在单击时触发。
- callBackHideEvent：回调函数，在隐藏时触发。
- callBackShowEvent：回调函数，在显示时触发。

3. 返回值

本插件是一个类型插件，需要初始化才能够使用，故没有返回值。

4. 成员描述

(1) 本类型插件包含一个初始化配置函数，说明如下。

init()：类型初始化方法。该方法在类型实例化过程中被调用，用来初始化设置 Tab 选项。

(2) 本类型插件包含 3 个公共方法，说明如下。

- setDisable()：该方法是一个公共方法，可以设置锁定 Tab 选项。包含一个参数，用来传递锁定的 Tab 选项的序号。
- setEnable()：该方法是一个公共方法，可以设置解锁 Tab 选项。包含一个参数，用来传递解锁的 Tab 选项的序号。
- triggleTab()：该方法是一个公共方法，可以触发选中指定的 Tab 选项。包含一个参数，用来传递被选中的 Tab 选项序号。

(3) 在本类型插件内部还包含一个私有方法，说明如下。

showContent()：该方法是一个私有方法，可以设置要显示的选项内容。包含两个参数，第一个参数用来设置要显示的选项序号，第二个参数传递参数选项对象。

(4) 本类型插件包含一个公共属性，说明如下。

defaults：该属性是一个列表结构的对象，可以设置该插件的默认选项值。

(5) 本类型插件包含两个私有属性，说明如下。

- **disableArr**：该属性是一个数组，用来记录哪些选项卡被禁用。
- **opts**：该属性是一个列表结构的对象，可以设置当前调用插件的默认选项值。

```
(function($) {
    var isShow = false;     //初始化显示变量
    $.fn.tab = function(options) {  //类型结构
        this.opts = $.extend({},$.fn.tab.defaults, options);   //设置类型的默认参数选项
        this._init();  //调用初始化配置方法
        this.disableArr=[]; //本地私有属性，存储禁用选项
    };
    $.fn.tab.prototype={    //类型原型对象
        _init:function(){   //初始化配置方法
            var _this = this;     //获取当前对象
            if($(_this.opts.tabList).length>0){    //如果存在选项
                $(_this.opts.tabList).each(function(index){ //遍历选项
                    $(this).bind(_this.opts.eventType,function(){     //为选项卡注册事件
                        if($.inArray(index,_this.disableArr)==-1&&(!isShow)&&$(this).
attr("class").indexOf(_this.opts.tabActiveClass)==-1){
                            //判断是否禁用，是否效果还在执行中，是否在当前选中的选项上
                            if(_this.opts.callBackStartEvent){ //调用回调函数
                                _this.opts.callBackStartEvent(index);
                            }
                            $(_this.opts.tabList).removeClass(_this.opts.tabActiveClass);
                            //移出所有激活选项卡
                            $(this).addClass(_this.opts.tabActiveClass);
                            //设置当前选项为激活状态
                            showContent(index,_this.opts);
                            //调用 showContent()私有函数，显示内容
                        }
                    });
                });
            }
        },
        //公共方法：禁用函数
        setDisable:function(index){  //选项序列号
            var _this = this;
            if($.inArray(index,this.disableArr)==-1){  //如果当前选项没有被禁用
                this.disableArr.push(index);           //记录当前选项序号
                $(_this.opts.tabList).eq(index).addClass(_this.opts.tabDisableClass);
//禁用选项
            }
        },
```

```
        //公共方法：解禁函数
        setEnable:function(index){
            var _this = this;
            var i =$.inArray(index,this.disableArr);   //判断当前选项是否被禁用
            if(i>-1){          //如果禁用，则解禁，并清除记录数组中的序号
                this.disableArr.splice(i,1);
                $(_this.opts.tabList).eq(index).removeClass(_this.opts.tabDisableClass);
            }
        },
        //公共方法：触发函数，根据指定序号，可以选中该项
        triggleTab:function(index){
            $(this.opts.tabList).eq(index).trigger(this.opts.eventType);
        }
    }
    //公共属性，设置插件的默认选项，详细说明请参阅上面参数说明
    $.fn.tab.defaults = {
        tabList:".tab_list li",
        contentList:".tab_content",
        tabActiveClass:"active",
        tabDisableClass:"disable",
        eventType:"click",
        showType:"show",
        showSpeed:200,
        callBackStartEvent:null,
        callBackHideEvent:null,
        callBackShowEvent:null
    };
    //内部函数，显示选项内容
    function showContent(index,opts){
        isShow = true;
        var _this = this;
        switch(opts.showType){  //根据显示类型，确定显示动画
            case "show":     //直接显示
                $(opts.contentList+":visible").hide();
                if(opts.callBackHideEvent){    //调用单击回调函数
                    opts.callBackHideEvent(index);
                }
                $(opts.contentList).eq(index).show();
                if(opts.callBackShowEvent){    //调用显示回调函数
                    opts.callBackShowEvent(index);
                }
                isShow =false;
                break;
            case "fade":     //渐变显示
                $(opts.contentList+":visible").fadeOut(opts.showSpeed,function(){
```

```
        //渐隐所有选项
            if(opts.callBackHideEvent){      //调用隐藏回调函数
                opts.callBackHideEvent(index);
            }
            $(opts.contentList).eq(index).fadeIn(function(){ //渐显当前选项
                if(opts.callBackShowEvent){ //调用显示回调函数
                    opts.callBackShowEvent(index);
                }
                isShow =false;
            });
        });
        break;
    case "slide":    //滑动显示
        $(opts.contentList+":visible").slideUp(opts.showSpeed,function(){
        //滑动隐藏所有选项
            if(opts.callBackHideEvent){      //调用隐藏回调函数
                opts.callBackHideEvent(index);
            }
            $(opts.contentList).eq(index).slideDown(function(){//滑动显示当前选项
                if(opts.callBackShowEvent){ //调用显示回调函数
                    opts.callBackShowEvent(index);
                }
                isShow =false;
            });
        });
        break;
    }
  };
})(jQuery);
```

下面我们就利用该插件设计一个 Tab 选项卡，代码如下。在该示例中，设置选项卡的切换事件为鼠标移过，显示类型为渐隐显示，效果如图 8.18 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
//省略$.fn.tab()类函数，参见上面的代码。
$(function(){
    //实例化$.fn.tab()类型
    var tab = new $.fn.tab({    //初始化设置参数
        tabList:".tab_list li",
        contentList:".tab_content",
        eventType:"mouseover",
        showType:"fade"
    });
})
```

```
</script>

<div class="tab_box">
    <ul class="tab_list">
        <li class="active">tab1</li>
        <li>tab2</li>
        <li>tab3</li>
    </ul>
    <div class="sub_box">
        <div class="tab_content">
            <p><img src="images/bg5.jpg" height="300" /></p>
        </div>
        <div class="tab_content" style="display:none">
            <p><img src="images/bg6.jpg" height="300" /></p>
        </div>
        <div class="tab_content" style="display:none">
            <p><img src="images/bg7.jpg" height="300" /></p>
        </div>
    </div>
</div>
```

下面设计一个更加复杂的选项卡切换效果。当单击选项标题栏时，将在单击后、隐藏行为后和显示当前选项行为后调用回调函数。同时，以滑动动画效果显示选项卡的切换过程，演示效果如图 8.19 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
//省略$.fn.tab()类函数，参见上面的代码
$(function(){
    var tab = new $.fn.tab({
        tabList:".tab_list li",
        contentList:".tab_content",
        eventType:"click",
        showType:"slide",
        callBackStartEvent:function(index){
            alert("单击了第 "+ (index +1) + " 个选项" );
        },
        callBackHideEvent:function(index){
            alert("隐藏了所有选项");
        },
        callBackShowEvent:function(index){
            alert("显示了当前选项");
        }
    });
})
</script>
```

图 8.18　鼠标移过切换选项卡效果

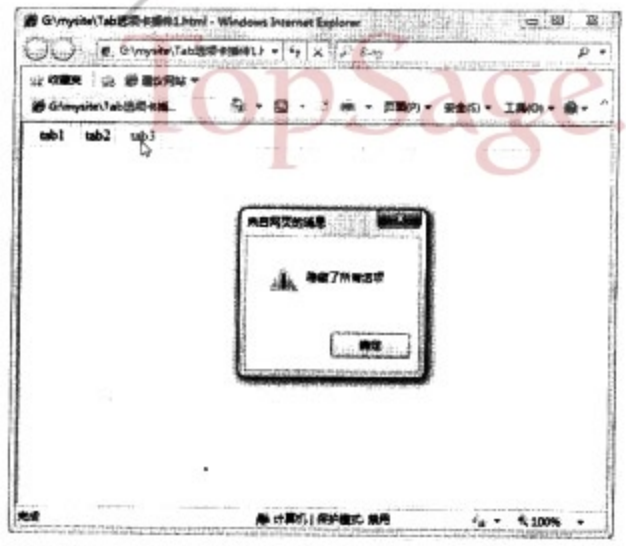


图 8.19　复杂动画切换过程

8.3　jQuery UI 插件应用

jQuery 官网提供了众多插件，读者可以访问 http://plugins.jquery.com 网站下载各类插件，如图 8.20 所示。其中插件数量还在不断增加。当然，你也可以把自己开发的插件上传到 jQuery 官网，同时对自己喜爱的插件进行评级和评价。

jQuery 对所有插件进行的分类如下所示，通过这些分类读者可以了解插件开发和应用的主题范围，并能够根据实际开发需要，有选择地下载这些插件。注意，分类列表中的数字表示该类插件的数目，截止到 2010 年 3 月 1 日前。

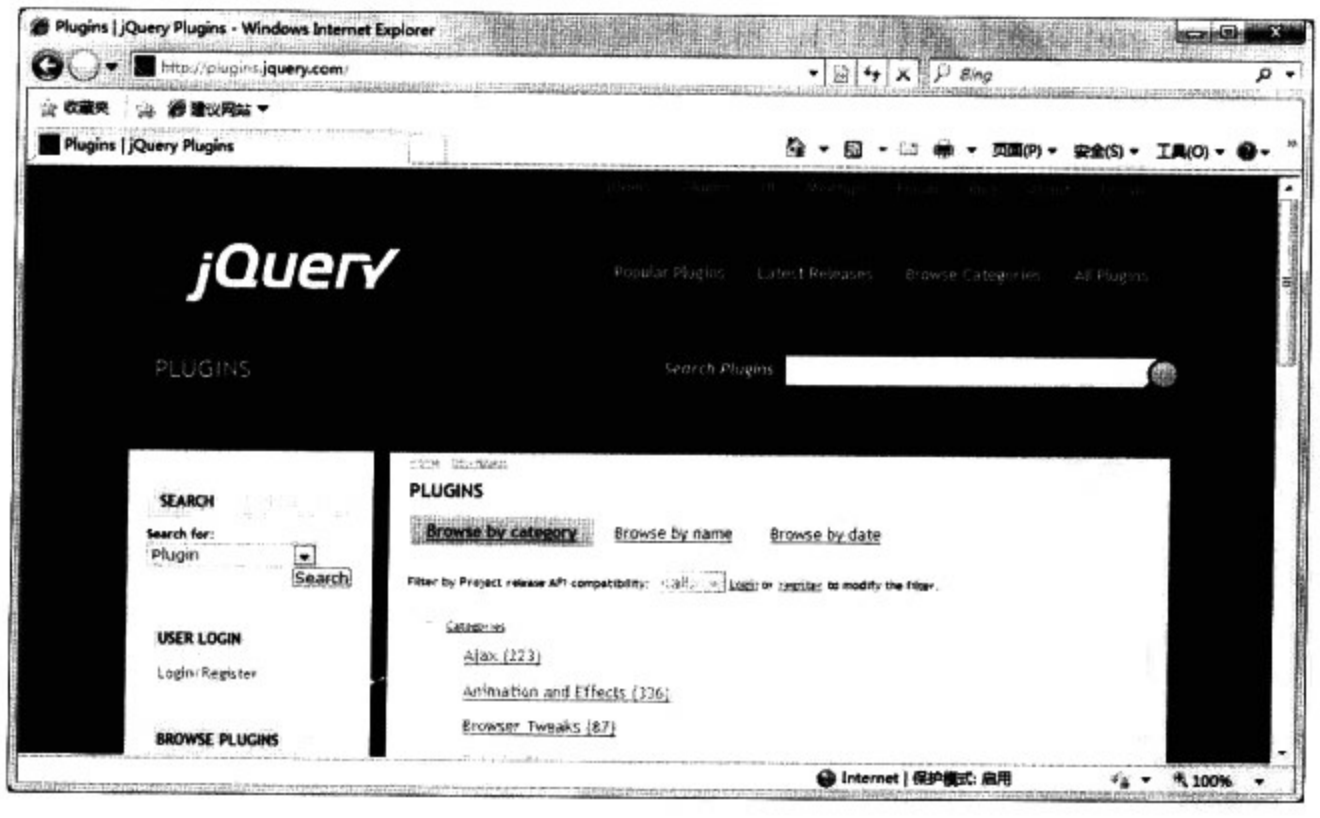


图 8.20　jQuery 官网插件

- Ajax (223)：异步通信插件。
- Animation and Effects (336)：动画和特效插件。
- Browser Tweaks (87)：浏览器系统调整插件。
- Data (163)：数据处理插件。
- DOM (165)：文档对象模型相关插件。
- Drag-and-Drop (38)：拖放插件。
- Events (143)：事件插件。
- Forms (397)：表单插件。
- Integration (118)：整合插件，即功能综合性质的插件。
- JavaScript (162)：JavaScript 核心功能插件。
- jQuery Extensions (249)：jQuery 扩展功能插件。
- Layout (204)：布局插件。
- Media (145)：媒体插件。
- Menus (114)：菜单插件。
- Metaplugin (27)：元插件，即所谓的插件的插件，通俗说就是用来制作插件的插件。
- Navigation (170)：导航器插件。
- Tables (82)：数据表格插件。
- User Interface (698)：用户界面插件，或者说是窗口或面板插件。
- Utilities (366)：公用工具插件。
- Widgets (263)：小部件插件，小的应用程序。
- Windows and Overlays (115)：窗口和遮盖插件。

8.3.1 如何使用外部插件

使用外部插件比较简单，但是如果稍不留神，也会弄错。

首先，在网上下载第三方插件文件，并把该文件包含在当前文档的头部区域，且确保其位于 jQuery 核心源文件的后面。

例如，下面是一个数据表格拖动效果的插件。当按下表格行之后，可以拖动其在表格中的位置，演示效果如图 8.21 和图 8.22 所示。

本插件在 jQuery 官网上的下载地址：

http://plugins.jquery.com/project/TableDnD

本插件作者的官方网址：

http://www.isocra.com/2008/02/table-drag-and-drop-jquery-plugin

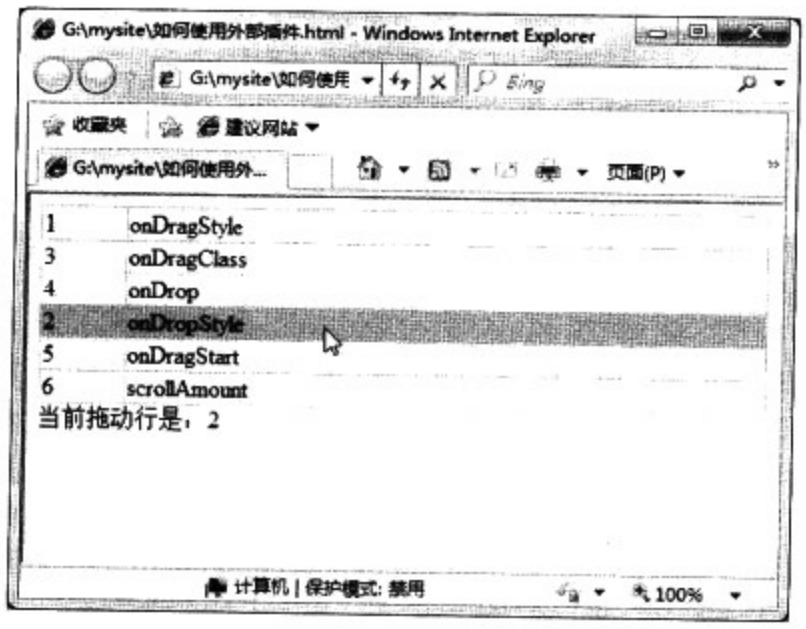


图 8.21　拖动表格行

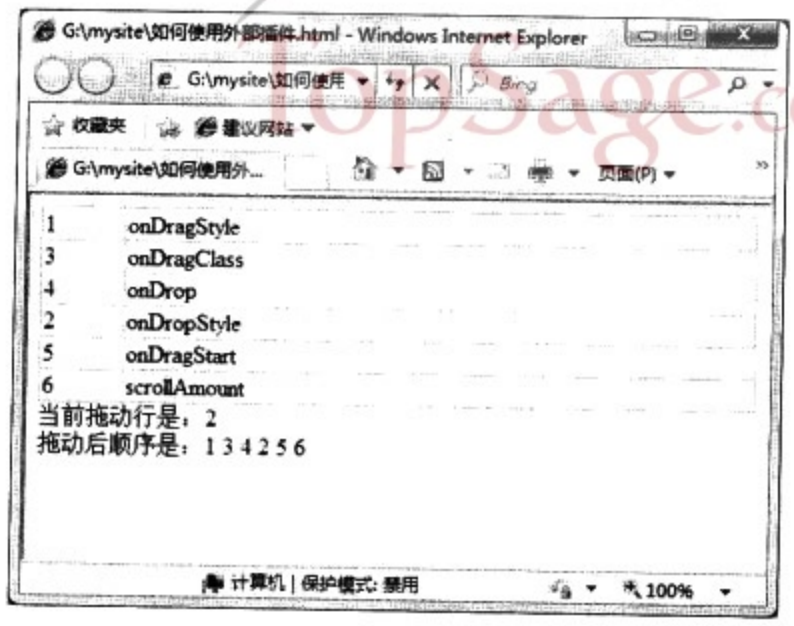


图 8.22　拖动后的表格行效果

下载并解压该插件，然后在当前页面中导入该文件。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title></title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script src="images/jquery.tablednd_0_5.js" type="text/javascript" ></script>
</head>
```

然后，在当前文档中使用该插件创建应用实例，或者调用该插件的扩展方法。一般可以放置在页面初始化事件处理函数中。在应用该插件之前，应该在文档中插入一个数据表格，并把数据表格的 id 传递给 jQuery 选择器，然后在选择器上面调用 tableDnD()插件函数。

每一种插件根据功能大小，都会包含一些参数说明，或者使用帮助文档。读者可以在插件的脚本文件顶部的注释说明进行查阅，也可以在插件的官方网址上进行查询。例如，本表格拖曳插件可以参阅 http://www.isocra.com/2008/02/table-drag-and-drop-jquery-plugin 网址了解其参数及其应用案例。

在下面的示例中，我们设置 tableDnD()插件函数包含 3 个插件选项参数，说明如下所示。

```
<script type="text/javascript">
$(function(){//页面初始化处理函数
    $("#table1").tableDnD({      //调用表格拖动函数
        onDragClass: "drag",     //设置拖动表格行的类样式
        onDrop: function(table, row) {    //设置放下拖动表格行后触发的回调函数，默认包含两个参
数，第一个参数传递当前表格对象，第二个参数传递当前行对象
            var rows = table.tBodies[0].rows;
            var debugStr = "当前拖动行是："+row.id+"<br />拖动后顺序是：";
            for (var i=0; i<rows.length; i++) {
```

```
                debugStr += rows[i].id+" ";
            }
            $("#debugArea").html(debugStr);
        },
        onDragStart: function(table, row) {   //设置拖动表格行后触发的回调函数，默认包含两个参
数，第一个参数传递当前表格对象，第二个参数传递当前行对象。
            $("#debugArea").html("当前拖动行是："+row.id);
        }
    });
})
</script>

<table id="table1">
    <tr id="1">
        <td>1</td>
        <td>onDragStyle</td>
    </tr>
    <tr id="2">
        <td>2</td>
        <td>onDropStyle</td>
    </tr>
    <tr id="3">
        <td>3</td>
        <td>onDragClass</td>
    </tr>
    <tr id="4">
        <td>4</td>
        <td>onDrop</td>
    </tr>
    <tr id="5">
        <td>5</td>
        <td>onDragStart</td>
    </tr>
    <tr id="6">
        <td>6</td>
        <td>scrollAmount</td>
    </tr>
</table>
<div id="debugArea"></div>
```

很多插件都具有很强的灵活性，并提供一些供用户设置的可选参数，用来改变插件的行为和属性，这样我们就可以按照特殊需要自定义插件来使用，也可以简单保持其默认的行为。例如，在上面表格拖曳的插件中，我们可以直接调用 tableDnD()函数。语句格式如下。

```
$("#table1").tableDnD()    //调用表格拖动函数
```

8.3.2 认识 UI 插件

UI 是 User Interface 一词的简称，翻译为用户界面，也称为人机界面，是指用户与某些系统进行交互的方法集合。这些系统不仅是电脑程序，也包括某种特定的机器、设备和复杂的工具等。

jQuery UI 源自一个 Interface 插件，它是由 Stefan Petre 创作的，最初的 Interface 插件只支持 jQuery 1.1.2 版本，其版本号为 1.2。后来在 Paul Bakaus 的参与和领导下，对 Interface 插件的源代码进行重构，并统一和规范了 API 接口和文档，将其更名为 jQuery UI 1.5。从此，jQuery UI 确定了官方插件的地位，官方访问地址是 http://jqueryui.com。

jQuery UI 是未来 jQuery 技术框架发展的趋势，也是未来互联网客户端发展的方向。jQuery UI 包含三部分：交互、部件和效果。

1) 交互

包括与鼠标的交互，与键盘的交互，如拖放、缩放、选择和排序等基本交互行为。Web 部件中很多行为都是基于这些基本交互行为来设计的。交互操作需要导入 jQuery UI 核心库 ui.core.js 文件。

2) 部件

包含各种界面风格和形式，如导航、对话框、提示、面板、侧栏、滑块、树结构、日历、拾色器、放大镜、标签、自动完成、进度条、微调控制器、历史、布局、栅格、菜单、工具提示、工具栏和上传组件等。部件需要导入 jQuery UI 核心库 ui.core.js 文件。

3) 效果

包含各种动画效果，需要导入效果库 effects.core.js 文件。

jQuery UI 插件下载地址为 http://jqueryui.com，单击页面右侧的稳定版本 1.7.2(即 Stable (1.7.2: jQuery 1.3+))。网页上还有最新版本 1.82，但是它仅支持 jQuery 1.4+，目前为测试版本，如图 8.23 所示。下载完毕，解压 jquery-ui-1.7.2.custom.rar 文件，就可以在 jquery-ui-1.7.2.custom 文件中看到欢迎页面 index.html 和三个子目录(js、CSS 和 development-bundle)。

打开开发包(development-bundle 文件夹)，可以看到 UI 插件的示例(demos)、文档(docs)、主题(themes)和核心库文件(ui)和扩展(external)文件夹。

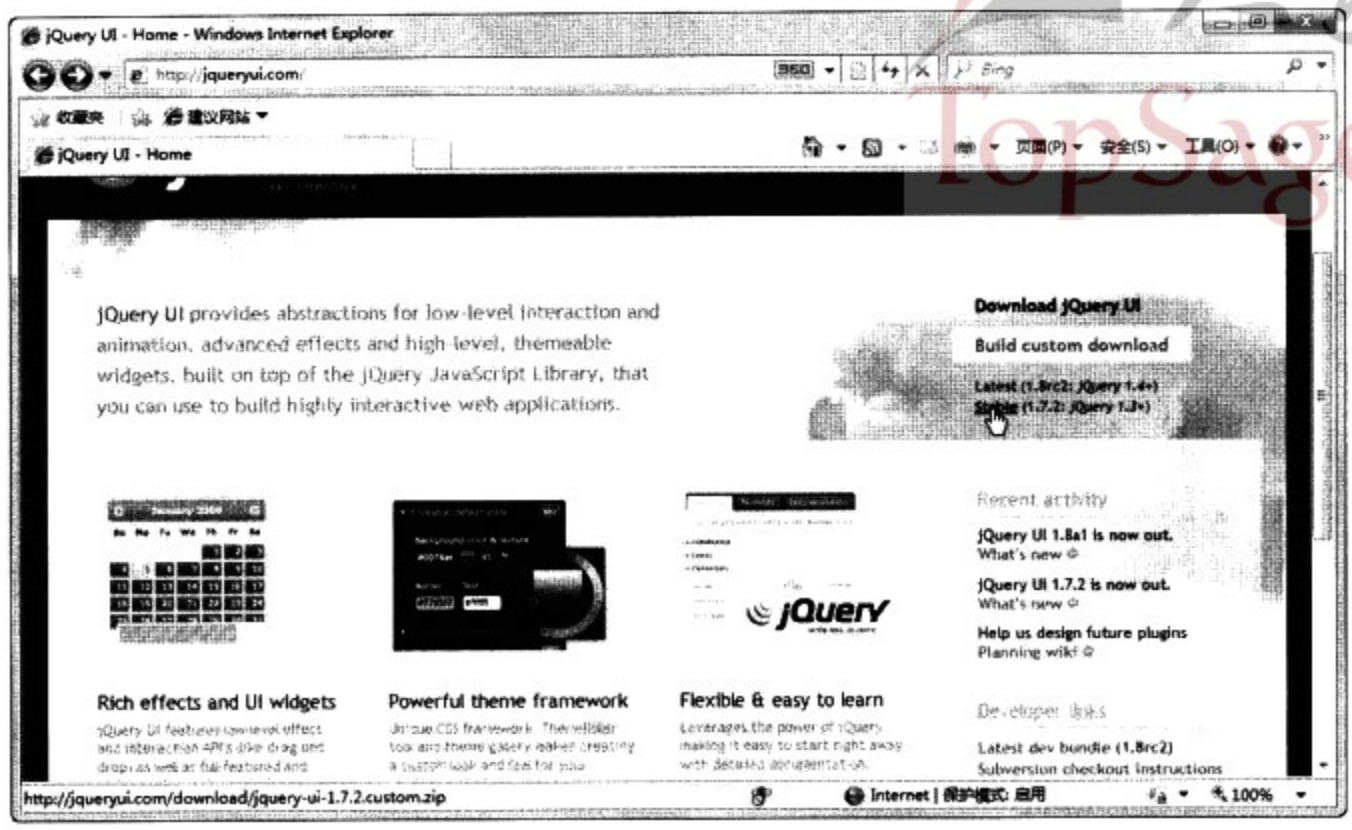


图 8.23　jQuery UI 插件官方下载页面

如果读者觉得这个 UI 插件包比较麻烦，可以直接单击右上角的 Build custom download 超链接(如图 8.23 所示)，打开自定义组件下载(http://jqueryui.com/download)，如图 8.24 所示。在左侧选择需要的组件，包括 UI Core(UI 核心库)、Interactions(交互)、Widgets(部件)、Effects(效果)。然后，在右侧 Theme(主题)中选择一种组件的样式主题，在右侧下面的 Version(版本)中选择框架的版本。最后单击 Download 按钮，即可按需下载对应的组件。

图 8.24　自定义下载 jQuery UI 插件

8.3.3 调整大小

先定制组件所需要的库。在 http://jqueryui.com/download 页面中(如图 8.24 所示)单击 Deselect all components 按钮，取消对所有组件的选择。在页面右侧的 Version(版本)区域勾选 1.7.2 版本，在页面左侧勾选 Resizable 复选框(位于 Interactions 交互区域)，此时页面会自动选中 Core 核心库。也可以在页面右侧的 Theme 区域选择一种主题。单击 Download 按钮下载即可。

把 jquery-ui-1.7.2.custom.rar 压缩文件解压在 jquery-ui-1.7.2.custom 文件夹中，并放置在当前文件的目录下。打开 jquery-ui-1.7.2.custom\development-bundle 目录，可以看到 UI 插件的示例(demos)、文档(docs)、主题(themes)和核心库文件(ui)和扩展(external)文件夹。

在使用之前，建议浏览 docs 子目录下的说明文档(resizable.html)，然后在 demos 子目录下测试不同应用示例。通过学习和模仿就能够快速掌握该插件的用法。

例如，新建文档，在当前头部区域导入核心库文件和样式表文件，代码如下。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title></title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<link type="text/css" href="jquery-ui-1.7.2.custom/development-bundle/themes/base/
ui.all.css" rel="Stylesheet" />
<script type="text/javascript" src="jquery-ui-1.7.2.custom/development-bundle/jquery-1.3.2.js">
</script>
<script type="text/javascript" src="jquery-ui-1.7.2.custom/development-bundle/ui/ui.core.js">
</script>
<script type="text/javascript" src="jquery-ui-1.7.2.custom/development-bundle/ui/
ui.resizable.js"> </script>
</head>
```

其中 jquery-1.3.2.js 为 jQuery 框架核心源代码，也可以导入压缩文件(jquery-ui-1.7.2.custom/js/ jquery-1.3.2.min.js)。ui.core.js 表示 UI 插件的核心库源代码，而 ui.resizable.js 表示改变大小插件的源文件。在 jquery-ui-1.7.2.custom/development-bundle/ui/minified 目录下可以导入压缩的文件。ui.all.css 表示主题样式文件。

下面就可以在文档中调用该插件了，示例代码如下所示，演示效果如图 8.25 所示。

```
<script type="text/javascript">
$(function(){
    $("#resizable").resizable();
})
</script>
<style type="text/css">
```

```
#resizable { width: 300px; height: 150px; padding: 0.5em; }
#resizable h2 { text-align: center; margin: 0; font-size:1.2em; }
</style>

<div id="resizable"   class="ui-widget-content">
    <h2  class="ui-widget-header">面板标题</h2>
</div>
```

该插件包含多个选项，读者可以参阅 docs 子目录下的文档说明，例如，我们为该 resizable() 函数添加一个参数，设计在拖曳过程中能够显示拖曳的重影，演示效果如图 8.26 所示。

```
<script type="text/javascript">
$(function(){
    $("#resizable").resizable({
        ghost: true
    });
})
</script>
```

图 8.25　调整元素大小

图 8.26　显示拖曳的重影

8.3.4　日期选择器

首先，定制组件所需要的库。在 http://jqueryui.com/download 页面中(如图 8.24 所示)单击 Deselect all components 按钮，取消对所有组件的选择。在页面右侧的 Version(版本)区域勾选 1.7.2 版本，在页面左侧勾选 Datepicker 复选框(位于 Widgets 部件区域)，也可以在页面右侧的 Theme 区域选择一种主题。单击 Download 按钮下载即可。

把 jquery-ui-1.7.2.custom.rar 压缩文件解压在 jquery-ui-1.7.2.custom 文件夹中，并放置在当前文件的目录下。如果本示例与上一节示例在同一个网站中，可以打开

jquery-ui-1.7.2.custom\development-bundle 目录，复制 themes 和 ui 文件夹到网站中上一节示例相同目录下，覆盖网站中的 themes 和 ui 文件夹。此时，如果打开 ui 文件夹，可以看到在原文件基础上，又添加了 ui.datepicker.js 文件和 i18n 文件夹。在 themes 文件夹内可以看到新添加的主题文件夹 sunny。

例如，新建文档，在当前头部区域导入核心库文件和样式表文件，代码如下。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title></title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<link  type="text/css"  href="jquery-ui-1.7.2.custom/development-bundle/themes/
sunny/ui.all.css" rel="Stylesheet" />
<script type="text/javascript"src="jquery-ui-1.7.2.custom/development-bundle/
jquery-1.3.2.js"></script>
<script type="text/javascript" src="jquery-ui-1.7.2.custom/development-bundle/ui/
ui.core.js"> </script>
<script type="text/javascript" src="jquery-ui-1.7.2.custom/development-bundle/ui/
ui.datepicker.js"></script>
</head>
```

实际上，如果仔细观察，可以看到本示例导入的库文件中，只是稍稍改变了 ui.datepicker.js，其他都没有变化。在主题样式表中，修改了主题文件夹名称，把 base 主题改为了 sunny 主题。

然后，就可以在页面初始化事件处理函数中调用 datepicker()函数，代码如下所示，演示效果如图 8.27 所示。

```
<script type="text/javascript">
$(function(){
    $('#datepicker').datepicker();
})
</script>

<p>设置日期: <input type="text" id="datepicker"></p>
```

通过添加设置选项，在日期选择器中添加自由选择年份和日期的下拉菜单，则代码如下，演示效果如图 8.28 所示。

```
<script type="text/javascript">
$(function(){
    $('#datepicker').datepicker({
        changeMonth: true,
        changeYear: true
    });
})
</script>
```

图 8.27　日期选择器

图 8.28　可选择日期和年份的日期选择器

第9章 jQuery辅助工具

jQuery 技术的核心是选择器，jQuery 框架的优势也在于它所设计的类似 CSS 语法规则的选择器。利用 jQuery 可以优化和改善开发人员设计 DOM 交互的灵活性和依赖性。当然，jQuery 为了开发需要也定义了很多实用性很强的工具函数，或者 jQuery 辅助函数。

本章将重点研究这些工具和辅助函数的应用，以及如何使用 JavaScript 来直接实现这些功能，以方便读者更深刻地认识和理解这些工具函数及其使用。

9.1 检测浏览器特性

jQuery 是基于跨浏览器的技术框架，也就是说当我们在使用 jQuery 代码时，不用为了兼容不同浏览器而烦恼。很多开发人员钟情于各种类型的 JavaScript 技术框架或者技术库，其中一个重要原因就是这些框架或者库都是建立在跨浏览器基础上的，都封装了浏览器兼容方案。

另外，jQuery 也定义了几个直接检测浏览器相关信息的工具函数，使用它们可以快速确定用户端浏览器的相关信息。

9.1.1 jQuery 检测浏览器的类型

检测浏览器的类型，主要根据 Navigator 对象的 userAgent 属性来实现，即通过引用 Window 对象的 navigator 属性来读取，语法如下。

```
var browser = navigator.userAgent;
```

jQuery 定义了 browser 对象，通过该对象可以获取当前浏览器的类型，浏览器对象检测技术与此对象属性共同使用可提供可靠的浏览器检测支持。

读者可以通过遍历或者检测属性值，确定当前浏览器的类型。例如，在下面的示例中，通过遍历对象属性，获取每个属性值，并确定当前浏览器的类型，演示效果如图 9.1 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    var browser = $.browser;
    var temp = ""
    for(var name in browser){
        if(browser[name] == true)
            temp += name + " = " + browser[name] + ", <strong>当前浏览器是 " + name  + "
</strong><br />";
        else
            temp += name + " = " +  browser[name] + "<br />";
    }
    $("div").html(temp)
})
</script>

<div></div>
```

browser 对象包含 5 个属性，其中用来检测浏览器类型的属性名分别为：safari、opera、

msie、mozilla。我们可以直接调用这些属性来检测当前浏览器是否为特定类型浏览器。这些属性在 DOM 树加载完成前即有效，因此可用于为特定浏览器设置 ready 事件。

图 9.1 获取浏览器的类型

例如，下面的代码可以分别为不同浏览器编写不同的页面初始化配置函数。访问方式可以通过点号运算符直接调用属性，也可以作为名称下标进行访问。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
if($.browser.msie){
    $(function(){
        alert("IE 浏览器专用页面初始化函数！");
    })
}
else if($.browser.safari){
    $(function(){
        alert("Safari 浏览器专用页面初始化函数！");
    })
}
else if($.browser["opera"]){
    $(function(){
        alert("Opera 浏览器专用页面初始化函数！");
    })
}
else if($.browser["mozilla"]){
    $(function(){
        alert("Firefox 浏览器专用页面初始化函数！");
    })
}
</script>
```

由于这种检测浏览器的方式缺乏灵活性，与 jQuery 技术框架的灵巧性相违背，因此在 jQuery 1.3 版本中不建议使用。当然，调用该对象是有效的。

9.1.2 JavaScript 检测浏览器的类型

jQuery 通过下面的代码计算当前浏览器的类型。

```
//获取用户代理字符串信息，并把字符串全部转换为小写形式
var userAgent = navigator.userAgent.toLowerCase();
//定义 jQuery 全局对象，该对象包含 5 个公共属性
jQuery.browser = {
    //匹配并过滤出字符串中的版本号
    version: (userAgent.match( /.+(?:rv|it|ra|ie)[\/: ]([\d.]+)/ ) || [0,'0'])[1],
    safari: /webkit/.test( userAgent ),    //匹配 webkit 关键字
    opera: /opera/.test( userAgent ),    //匹配 opera 关键字
    msie: /msie/.test( userAgent ) && !/opera/.test( userAgent ),    //匹配 msie 关键字
    mozilla: /mozilla/.test( userAgent ) && !/(compatible|webkit)/.test( userAgent )
    //匹配 mozilla 关键字
};
```

读者可以在上面代码的基础上对其进行封装和完善。例如，在下面的示例中，定义了browser()全局函数，通过向该函数传递一个字符串，以测试当前浏览器是否是指定类型的浏览器。

```
<script type="text/javascript">
//参数：str 是一个字符串类型的参数，包含下面几个固定值。
ie：匹配 IE 浏览器
op：匹配 Opera 浏览器
sa：匹配 Safari 浏览器
ch：匹配 Chrome 浏览器
ff ：匹配 Firefox 浏览器
//返回值：布尔值，true 表示是特定类型的浏览器，false 表示不是特定类型的浏览器
function browser(str){
    var userAgent = navigator.userAgent.toLowerCase();
    switch (str){
        case "ie":
            return /msie/.test( userAgent ) && !/opera/.test( userAgent );
        case "ff":
            return /mozilla/.test( userAgent ) && !/(compatible|webkit)/.test( userAgent );
        case "sa":
            return /webkit/.test( userAgent );
        case "op":
            return /opera/.test( userAgent );
        case "ch":
            return /chrome/.test( userAgent );
        default:
            return false;
    }
}
//测试当前浏览器类型，使用一个长表达式语句来模拟多条件的判断结构
window.onload = function(){
    browser("ie") && (useragent = "当前浏览器是 IE 浏览器") ||
    browser("ff") && (useragent = "当前浏览器是 Firefox 浏览器") ||
    browser("sa") && (useragent = "当前浏览器是 safari 浏览器") ||
    browser("op") && (useragent = "当前浏览器是 Opera 浏览器") ||
    browser("ch") && (useragent = "当前浏览器是 Chrome 浏览器");
    alert(useragent);
```

```
}
</script>
```

9.1.3 更灵巧的浏览器检测方法

浏览器检测的方法有许多种，如字符串检测法和特征检测法。所谓字符串检测法就是根据 navigator.userAgent 属性返回值进行检测，读者可以参阅上面两节内容的讲解。但是，jQuery 从 1.3 版本开始就不再支持使用这种方法，原因就是它使用起来比较麻烦，与 jQuery 技术框架的灵巧特色相违背。

现在，在 JavaScript 脚本和一些技术框架中，都喜欢使用特征检测法来判断浏览器的类型。特征检测法就是根据浏览器是否支持特定功能来决定操作的方式。

特征检测法是一种非精确的检测方法，也是最安全的检测途径。因为准确检测浏览器的类型和型号是一件很困难或者说很麻烦的事情，而且很容易存在误差。如果读者不关心浏览器的身份，仅仅在意浏览器的执行能力，那么使用特征检测法就完全可以满足需要。例如：

```
if(document.getElementsByName){          //如果存在，则使用该方法获取 a 元素
    var a = document.getElementsByName("a");
}
else(document.getElementsByTagName){     //如果存在，则使用该方法获取 a 元素
    var a = document.getElementsByTagName("a");
}
```

当使用对象、方法或属性时，可以先检测当前浏览器是否支持它。在逻辑表达式中，如果浏览器支持，则会返回该对象、属性或方法，这时 JavaScript 就会强制把这些对象或成员转换为 true；如果不支持，则会返回 undefined，JavaScript 会自动把它转换为布尔值 false。

如果要检查方法或函数，应注意不要附加小括号运算符。否则 JavaScript 解释器会调用该方法或函数，同时如果指定函数或方法不存在，就会产生编译错误。

9.1.4 检测浏览器的版本号

1. jQuery 实现

在 jQuery 中可以借助 jQuery.browser.version 属性来获取浏览器的版本号。例如，下面的代码可以返回当前浏览器的版本号，返回值是字符串类型。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    alert( $.browser.version );
})
</script>
```

2. JavaScript 实现

如果使用 JavaScript 实现，则可以使用下面的函数。

```
<script type="text/javascript">
function getVersion(){    //返回浏览器的版本号
    var userAgent = navigator.userAgent.toLowerCase();
    return  (userAgent.match( /.+(?:rv|it|ra|ie)[\/: ]([\d.]+)/ ) || [0,'0'])[1];
}
window.onload = function(){
    alert(getVersion());
}
</script>
```

9.1.5 检测浏览器的盒模型

所谓盒模型，就是指浏览器解析元素的方法，它是 CSS 布局中的一个基本概念。盒模型规定了如何决定元素(与其内边距和边框间距)的内容大小，外边距虽然也是盒模型的一部分，但是它不参与确定内容的大小。

除了 IE 浏览器外，其他浏览器都只支持 W3C 标准的盒模型，而 IE 浏览器能够根据页面模式(严格模式或者怪异模式)有选择地使用不同类型的盒模型。如果页面顶部声明了文档类型(DOCTYPE)，则 IE 也会采用严格模式，即 W3C 标准的盒模型解析元素；如果文档中没有包含文档类型，或者包含了无法识别的文档类型声明，则 IE 会以怪异模式显示，并按 IE 传统的盒模型来解析元素。

总之，IE 的传统盒模型与 W3C 盒模型主要区别在于如何解释元素 width 和 height 属性的计算标准。请记住以下两条原则。

- IE 传统盒模型：width 和 height 属性包含内边距和边框宽度。
- W3C 盒模型：width 和 height 属性不包含内边距和边框宽度。

1. jQuery 实现

通过调用 jQuery.boxModel 属性可以确定浏览器在解析当前文档时是否支持 W3C 标准的盒模型。如果返回值为 true，则表示支持，否则表示不支持，即支持 IE 的怪异模式。例如，下面代码可以使你了解该属性的应用。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    alert( $.boxModel  &&  "支持 W3C 标准盒模型"  ||  "支持 IE 的怪异解析模式"  );
})
</script>
```

此属性在 jQuery 1.3 版本中不建议使用，如果要检测当前页面中浏览器是否使用标准盒模型渲染页面，建议使用 jQuery.support.boxModel 属性来代替。

2. JavaScript 实现

如果直接使用 JavaScript 实现检测浏览器在解析当前文档时所采用的解析模型，可以利用脚本在当前文档中创建一个 div 元素，设置 div 元素的左侧补白和边框宽度为 1 像素，然后调用该元素的 offsetWidth 属性获取 div 元素的可见宽度，如果它的值等于 2 像素，则说明是按 W3C 标准进行解析的，否则就是按 IE 的怪异模型进行解析的。实现的详细代码如下所示。

```
<script type="text/javascript">
function isBoxModel(){
    var div = document.createElement("div");
    div.style.width = div.style.paddingLeft = "1px";
    document.body.appendChild( div );
    var width =  div.offsetWidth;
    div.style.display = 'none';
    document.body.removeChild( div )
    return width === 2;
}
window.onload = function(){
    alert( $.boxModel  &&  "支持 W3C 标准盒模型"  ||  "支持 IE 的怪异解析模式"  );
}
</script>
```

9.1.6　浏览器特性综合测试

从 1.3 版本开始，jQuery 重新设计了浏览器特性检测方法，把所有相关属性都集中到 support 对象中，这样就方便管理和使用。当然，用户也可以自由增加自己的属性。在 support 对象中，很多属性是很低级的，所以很难确保在日后版本升级中总是保持有效，但这些功能主要用于插件和内核开发。support 对象包含的属性及其测试内容说明如表 9.1 所示。

表 9.1　jQuery.support 对象包含属性说明

属　性	说　明
boxModel	如果浏览器解析当前文档是支持 W3C 标准盒模型的，则返回 true。如果是支持 IE 6 和 IE 7 的怪异模式则返回 false。在页面初始化之前，则返回 null
cssFloat	如果浏览器使用 cssFloat 属性来访问 CSS 的 float 样式值，则返回 true，否则返回 false。在 IE 浏览器中会返回 false，因为它使用 styleFloat 属性来访问 CSS 的 float 样式值
hrefNormalized	如果浏览器从 getAttribute("href")返回的是原封不动的结果，则返回 true，否则返回 false。在 IE 浏览器中会返回 false，因为它对返回的结果进行了格式化处理

续表

属　性	说　明
htmlSerialize	如果浏览器通过 innerHTML 插入 a 元素的时候，会自动序列化这些超链接，则返回 true，否则返回 false。目前在 IE 浏览器中会返回 false
leadingWhitespace	如果浏览器在使用 innerHTML 时，保持前导空白字符，则返回 true，否则返回 false。目前在 IE6～8 版本浏览器中会返回 false
noCloneEvent	如果浏览器在克隆元素的时候不会连同事件处理函数一起复制，则返回 true，目前在 IE 中返回 false
objectAll	如果在某个元素对象上执行 getElementsByTagName("*")时会返回所有子孙元素，则为 true，目前在 IE 7 中为 false
opacity	如果浏览器能适当解释透明度样式属性，则返回 true，由于 IE 浏览器使用 alpha 滤镜实现，故返回 false
scriptEval	使用 appendChild()或 createTextNode() 方法插入脚本代码时，浏览器是否执行脚本，目前在 IE 中不能够执行，故返回 false，而其他浏览器执行 IE 使用 text 方法插入脚本代码可以执行
style	如果 getAttribute("style")返回元素的行内样式，则为 true。由于 IE 使用 cssText 返回元素的行内样式，故返回 false
tbody	如果浏览器允许 table 元素不包含 tbody 元素，则返回 true。目前在 IE 中会返回 false，它会自动插入缺失的 tbody 元素

所有这些支持的属性值都通过特性检测来实现，而不是用任何浏览器检测。下面是一些非常棒的资源用于解释这些特性检测是如何工作的。

- http://peter.michaux.ca/articles/feature-detection-state-of-the-art-browser-scripting
- http://yura.thinkweb2.com/cft
- http://www.jibbering.com/faq/faq_notes/not_browser_detect.html

9.2　字符串处理

jQuery 扩展了字符串处理方法，定义了 trim()和 param()方法。其中 trim()方法用于修剪字符串，而 param()方法能够把数组或对象转换为字符串序列。

9.2.1　修剪字符串

JavaScript 定义了很多字符串处理方法，但是竟然没有考虑提供如何修剪字符串的便捷方法，而这些方法在实际开发中是非常实用的。例如，在处理表单提交的数据时，经常需要清理掉字符串前后的空白，由于这些空白对于人来说是不可见的，但是程序却能够识别，所以

很容易产生低级错误。

1. jQuery 实现

jQuery 定义了 trim()函数能够清理字符串前后的空白。trim()是一个全局函数，可以直接使用 jQuery 对象进行调用，该方法包含一个字符串型的参数，即将被修剪的字符串，返回修剪后的字符串。例如，下面的示例演示了字符串在被 jQuery 的 trim()方法修剪前后的字符串长度变化。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
var  str = "   去掉字符串起始和结尾的空格   ";
alert(str.length);    //返回 19
str = jQuery.trim(str);
alert(str.length);    //返回 13
</script>
```

2. JavaScript 实现

jQuery 定义的 trim()函数，使用 JavaScript 实现也很简单，只需要借助正则表达式即可。例如，下面示例定义了一个 trim()全局函数，并调用该函数修剪字符串。

```
<script type="text/javascript">
function trim(text){
    return (text || "").replace( /^\s+|\s+$/g, "" );
}
window.onload = function(){
    var  str = "   去掉字符串起始和结尾的空格   ";
    alert(str.length);    //返回 19
    str = trim(str);
    alert(str.length);    //返回 13
}
</script>
```

9.2.2 序列化字符串

jQuery 的 param()函数能够将表单元素数组或者对象序列化，它是 serialize()方法的基础(请参阅 Ajax 一章相关内容)。所谓序列化，就是指数组或者 jQuery 对象按照名/值(name/value)对格式进行序列化，而 JavaScript 普通对象按照 key/value 对格式进行序列化。

1. jQuery 实现

例如，在下面的示例中 param()函数能够把列表结构的对象 obj 转换为字符串类型的名/值对字符串，返回字符串“width=400&height=300”。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
```

```
$(function(){
    var option = {
        width:400,
        height:300
    };
    var str = jQuery.param( option );
    $("#result").text(str);
})
</script>

<div id="result"></div>
```

2. JavaScript 实现

jQuery 在实现 param()函数功能时，考虑的因素比较多，例如参数是否是 jQuery 类型对象，故设计的函数代码比较复杂。实际上我们可以使用下面的普通函数来实现相同的功能。

```
<script type="text/javascript">
function param( a ){    //实现 jQuery 中的 param()函数功能
    var s = [ ];
    function add( key, value ){    //定义私有函数，把名值对连接为字符串，并传递给数组
        s[ s.length ] = encodeURIComponent(key) + '=' + encodeURIComponent(value);
    };
    for( var j in a ){    //遍历对象，连接所有属性
        add( j, a[j] );
    };
    return s.join("&").replace(/%20/g, "+");    //连接数组中的元素，并把空格替换为加号符号
}
window.onload = function(){
    var option = {
        width:400,
        height:300
    };
    var str = param( option );
    var div = document.getElementsByTagName("div")[0];
    div.innerHTML = str;
}
</script>

<div></div>
```

9.3 数 组 处 理

与字符串一样，数组在 JavaScript 开发中一样重要，使用频繁且必不可少，善于使用数组能够提高代码的执行效率。虽然 Array 数组对象包含了众多强大的方法，但是 jQuery 仍然根据实际开发需要对其进行扩展。

9.3.1 检测数组

由于数组和对象都是散列式列表结构，它们都可以存储大量数据，开发人员喜欢使用数组或者对象来进行数据周转，但是数组和对象的操作方法各异，因此在开发中如何快速了解当前值是数组还是对象就非常重要。

1. jQuery 实现

isArray()函数是 jQuery 定义的负责检测对象是否为数组的专用工具。该工具用法简单，而且比较实用，它可以快速判断指定对象是否为数组，以方便程序进行处理。

例如，下面的示例将检测变量 a 是否为数组，如果是数组则执行特定的代码。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    var a = [
        {width:400},
        {height:300}
    ];
    if(jQuery.isArray( a ))
        alert("变量 a 是数组");
})
</script>
```

2. JavaScript 实现

jQuery 定义的 isArray()函数实现起来很简单，我们直接调用该对象的 toString()方法即可，考虑到参数对象的 toString()方法可能会被重写，因此，直接调用 Object 对象的原型方法即可。使用 JavaScript 实现的代码如下所示。

```
<script type="text/javascript">
function isArray( obj ){
    return Object.prototype.toString.call(obj) === "[object Array]";
}
window.onload = function(){
    var a = [
        {width:400},
        {height:300}
    ];
    if(isArray( a ))
        alert("变量 a 是数组");
}
</script>
```

3. 关于 isFunction()函数

isFunction()函数是 jQuery 定义的用来检测指定对象是否为函数类型的函数。该函数与

isArray()函数用法相同，其实现的 JavaScript 代码如下。

```
function isFunction( obj ){
    return Object.prototype.toString.call(obj) === "[object Function]";
}
```

在 jQuery 1.3 版本以后，在 IE 浏览器里，浏览器提供的函数，如'alert 和'getAttribute(DOM 元素方法)将被认为不是函数。

9.3.2 遍历数组或集合对象

对数组或者集合对象进行迭代操作，是开发中的家常便饭。JavaScript 提供了 for 和 for in 语句完成类似的任务。但是这种繁琐的操作有时候会让人觉得很头疼。jQuery 简化了这种操作，把所有工作都交付给 each()函数来实现。

1. jQuery 实现

each()函数是 jQuery 通用迭代工具，可用于遍历数组或者集合对象。语法格式如下。

```
jQuery.each(object, [callback])
```

参数 object 表示要遍历的集合对象，callback 表示回调函数，该函数将在遍历每个成员时触发。回调函数包含两个默认参数，第一个参数为对象成员或数组的索引，第二个参数为对应变量或内容。

例如，在下面的示例中，调用 jQuery.each()函数遍历数组 a，然后在遍历过程中，逐一提示该数组元素的下标值和元素值，演示效果如图 9.2 和图 9.3 所示。

图 9.2 访问第一个元素

图 9.3 访问第二个元素

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    var a = [
        {width:400},
        {height:300}
```

```
    ];
    jQuery.each(a,function(name,value){
        alert("当前成员的名称：" + name + " = " + value);
    })
})
</script>
```

如果中途需要退出 each()循环，则可以在回调函数中返回 false，其他返回值将被忽略。例如，在上面示例的基础上，在 each()函数中添加一个条件语句，如果数组下标超过 0，则退出循环，即当显示如图 9.2 所示的提示之后，就退出循环。具体代码如下所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    var a = [
        {width:400},
        {height:300}
    ];
    jQuery.each(a,function(name,value){
        if(name>0) return false;    //仅执行一次循环即退出
        alert("当前成员的名称：" + name + " = " + value);
    })
})
</script>
```

jQuery 的 each()函数与 jQuery 对象的 each()方法功能相同，但是用法不同，另外 each()函数可用于遍历任何对象。

2. JavaScript 实现

如果遍历对象集合，可以使用 for in 语句实现；如果遍历数组集合，则可以使用 for 语句实现。在下面的示例中，直接使用 JavaScript 实现 each()函数。

```
<script type="text/javascript">
function each( object, callback, args ) {
        var name, i = 0, length = object.length;
        if ( args ) {    //如果存在第 3 个参数
            if ( length === undefined ) {    //如果是对象集合
                for ( name in object )
                    if ( callback.apply( object[ name ], args ) === false )
                        break;    //如果回调函数返回值为 false，则跳出循环
            } else    //如果是数组
                for ( ; i < length; )
                    if ( callback.apply( object[ i++ ], args ) === false )
                        break;    //如果回调函数返回值为 false，则跳出循环
        } else {    //如果不存在第 3 个参数
            if ( length === undefined ) {    //如果是对象集合
                for ( name in object )
```

```
                if ( callback.call( object[ name ], name, object[ name ] ) === false )
                    break;     //如果回调函数返回值为 false，则跳出循环
            } else    //如果是数组
                for ( var value = object[0];
                    i < length && callback.call( value, i, value ) !== false; value =
object[++i] ){}
        };
        return object;
};
window.onload = function(){
    var a = [
        {width:400},
        {height:300}
    ];
    each(a, function(name,value){
        if(name>0) return false;
        alert("当前成员的名称：" + name + " = " + value);
    });
}
</script>
```

9.3.3 转换为数组

散列表结构的数据可能是数组类型，也可能是对象类型。由于数组和对象类型拥有不同的操作方法，特别是数组对象，JavaScript 为其定义了众多强大的处理方法。因此，在 DOM 中经常需要把列表结构的数据转换为数组。

例如，使用 jQuery 获取文档中所有 li 元素，则返回的应该是一个类似数组结构的对象，但是如果直接为其调用 reverse()数组方法，则会显示编译错误，如图 9.4 所示。因为，$("li") 返回的是一个类数组结构的对象，而不是数组类型数据。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    var arr = $("li");
    $("ul").html(arr.reverse());
})
</script>

<ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ul>
```

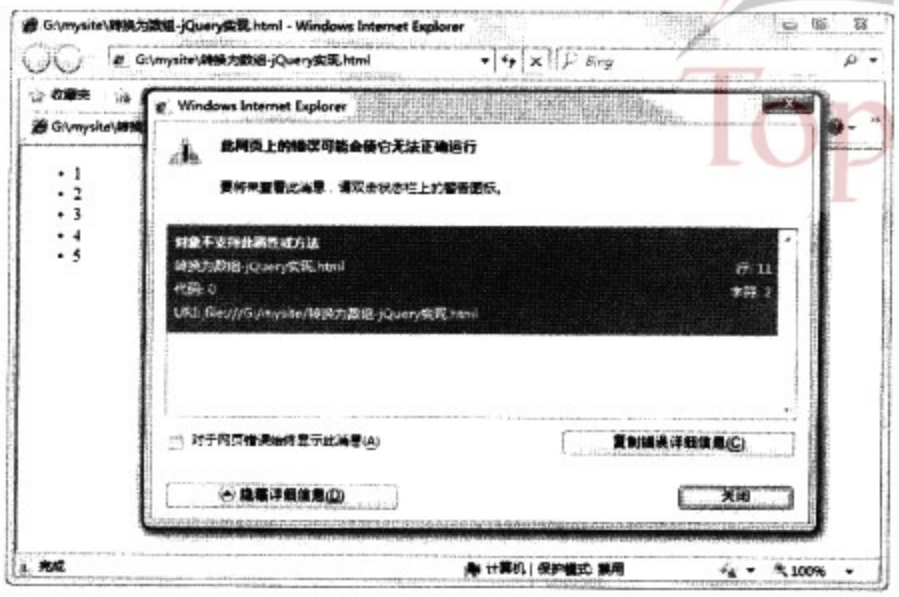


图 9.4 错误的调用方法

1. jQuery 实现

jQuery 的 makeArray()函数能够把这些类数组结构的对象转换为数组对象。所谓类数组对象，就是对象也拥有 length 属性，其成员索引从 0 到 length-1。但是这些对象不能够调用数组方法。

例如，针对上面的示例，可以先使用 makeArray()函数把类数组对象转换为数组对象，然后再为其调用 reverse()方法。这时就可以看到页面中的列表结构被颠倒过来。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    var arr = jQuery.makeArray($("li"));    //转换为数组
    $("ul").html(arr.reverse());     //再调用 reverse()方法
})
</script>
```

2. JavaScript 实现

如果直接使用 document.getElementsByTagName("li")获取 li 元素集合，则返回的也是一个类数组结构的对象，该对象也无法直接调用数组方法。使用 JavaScript 直接定义 makeArray()函数的代码如下所示。然后就可以利用该函数把 document.getElementsByTagName("li")获取的 li 元素集合转换为数组，从而实现调用数组排序方法。

```
<script type="text/javascript">
function makeArray( array ) {    //模拟 jQuery 定义的转换数组函数 makeArray()
    var ret = [];
    if( array != null ){    //如果参数存在
        var i = array.length;
        //如果参数对象不支持 length 属性，或者仅是字符串类型，或者是一个函数对象，或者是一个 DOM 元素对象
        if( i == null || typeof array === "string" || isFunction(array) ||
array.setInterval )
            ret[0] = array;    //则把它作为一个元素直接传递返回的数组
```

```
        else    //否则，则遍历对象成员，把每个成员作为数组元素返回
            while( i )
                ret[--i] = array[i];
    }
    return ret;
    function isFunction( obj ){
        return Object.prototype.toString.call(obj) === "[object Function]";
    }
}
window.onload = function(){
    var arr = document.getElementsByTagName("li");
    var ul = document.getElementsByTagName("ul")[0];
    ul.innerHTMl = arr.reverse();
}
</script>
```

9.3.4 过滤数组

1. jQuery 实现

过滤数组是一个很有意思的功能，jQuery 定义了 grep()函数，该函数能够根据过滤函数过滤掉数组中不符合条件的元素。

grep()函数包含三个参数，语法格式如下。

```
jQuery.grep(array, callback, [invert])
```

参数 array 表示要过滤的数组，callback 表示过滤函数。如果过滤函数返回 true，则保留元素，如果过滤函数返回 false，则可以删除元素。

过滤函数将遍历并处理数组中的每个元素。该函数包含两个参数，第一个参数表示当前元素，第二个参数表示元素的索引值。过滤函数应返回一个布尔值，如果为 true，则表示当前元素保留，如果为 false，则表示当前元素被删除。另外，此函数可设置为一个字符串，当设置为字符串时，将视为“lambda-form”(缩写形式)，其中 a 代表数组元素，i 代表元素索引值。如“a>0”代表“function(a){ return a > 0; }”。

grep()函数的第三个参数 invert 是一个可选的布尔值，如果为 false 或者没有设置，则函数返回数组中由过滤函数返回 true 的元素；如果该参数为 true，则返回过滤函数中返回 false 的元素集。

例如，在下面这个示例中，使用 grep()函数筛选出大于等于 5 的数组元素，并返回一个新数组。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    var arr = [1,2,3,4,5,6,7,8,9,0];
```

```
    arr = jQuery.grep(arr, function(value, index){
            return value >= 5;
    });
    alert(arr);    //返回"5,6,7,8,9"
})
</script>
```

反过来，如果过滤掉大于等于 5 的数组元素，则可设置第三个参数值为 true。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    var arr = [1,2,3,4,5,6,7,8,9,0];
    arr = jQuery.grep(arr, function(value, index){
            return value >= 5;
    }, true);
    alert(arr);    //返回"1,2,3,4,0"
})
</script>
```

2. JavaScript 实现

jQuery 定义的 grep()函数用法比较复杂，但是使用 JavaScript 直接定义该函数的方法却很简单，详细代码如下所示。

```
<script type="text/javascript">
function grep( elems, callback, inv ) {//模拟 jQuery 的 grep()函数功能，直接定义 grep()函数
    var ret = [];
    for ( var i = 0, length = elems.length; i < length; i++ )    //遍历数组
        if ( !inv != !callback( elems[ i ], i ) )    //如果符合条件，则存储该元素
            ret.push( elems[ i ] );
    return ret;    //返回筛选后的新数组
}
window.onload = function(){
    var arr = [1,2,3,4,5,6,7,8,9,0];
    arr = grep(arr, function(value, index){
            return value >= 5;
    }, true);
    alert(arr);    //返回"1,2,3,4,0"
}
</script>
```

9.3.5 映射数组

jQuery 定义了一个映射数组的函数 map()，该函数拥有 grep()函数的过滤功能，同时还可以把当前数组根据处理函数处理后，映射为新的数组，甚至可以在映射过程中放大数组。

1. jQuery 实现

map()函数的用法与 grep()函数基本相似，包含两个参数：第一个参数表示被映射的数组，

第二个参数表示数组元素处理转换函数。其语法格式如下所示。

```
jQuery.map(array, callback)
```

作为第二个参数的转换函数会为每个数组元素调用，而且会给这个转换函数传递一个表示被转换的元素作为第一参数，元素的序号作为第二个参数被传递给转换函数。转换函数可以返回转换后的值。

如果转换函数返回值为 null，则表示删除数组中对应的项目；如果转换函数返回值为一个包含值的数组，则表示将扩展原来的数组。例如，下面的示例将把数组 arr 中的元素放大一倍之后，映射到一个新的数组中。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    var arr = [1,2,3,4];
    arr = jQuery.map(arr, function(elem){
            return elem * 2;
    });
    alert(arr);    //返回"2,4,6,8"
})
</script>
```

如果要修改转换函数，设置放大之后小于 5 的元素值，则返回 null，即过滤掉数组中 1 和 2 两个元素。实现代码如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    var arr = [1,2,3,4];
    arr = jQuery.map(arr, function(elem){
            return elem * 2 > 5? elem * 2 : null;
    });
    alert(arr);    //返回"6,8"
})
</script>
```

如果在转换函数中，设置返回值为数组，则可以在映射数组中扩大数组的长度。实现代码如下。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    var arr = [1,2,3,4];
    arr = jQuery.map(arr, function(elem){
            return [elem,elem * 2];
    });
    alert(arr);    //返回"1,2, 2,4, 3,6, 4,8"
```

```
})
</script>
```

2. JavaScript 实现

使用 JavaScript 直接定义 map()函数的代码如下。

```
<script type="text/javascript">
function map( elems, callback ) {    //模仿 jQuery 的 map()函数功能，映射数组
    var ret = [];
    for ( var i = 0, length = elems.length; i < length; i++ ) {    //遍历数组
        var value = callback( elems[ i ], i );    //执行回调转换函数，并获取函数返回值
        if ( value != null )    //如果返回值不为 null，则把该元素存储到临时数组中
            ret[ ret.length ] = value;
    }
    return ret.concat.apply( [], ret );    //把临时数组连接到一个返回的空数组中，从而实现把数
组类型的元素分开，存储到返回数组中
}
//调用新定义的 map()函数
window.onload = function(){
    var arr = [1,2,3,4];
    arr = map(arr, function(elem){
        return [elem,elem * 2];
    });
    alert(arr);
}
</script>
```

9.3.6 合并数组

1. jQuery 实现

jQuery 定义了一个合并数组的函数 merge()，该函数能够把两个参数数组合并为一个新数组并返回。

merge()函数用法很简单，只需要向其传递两个数组参数即可。返回的结果会修改第一个参数数组的内容，也就是说第一个参数数组的元素后面被连接了第二个参数数组的元素。

例如，在下面的示例中，调用 merge()函数把数组 arr1 和 arr2 合并在一起，并把合并后的数组传递给 arr1，同时返回合并后的新数组。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    var arr1 = [1,2,3,["a", "b", "c"]];
    var arr2 = [4,5,6,[7,8,9]];
    arr3 = jQuery.merge(arr1, arr2);
    alert(arr1);    //返回数组[1,2,3,["a", "b", "c"],4,5,6,[7,8,9] ]
    alert(arr1.length);    //返回 8
```

```
    alert(arr3);     //返回数组[1,2,3,["a", "b", "c"],4,5,6,[7,8,9] ]
    alert(arr3.length);     //返回 8
})
</script>
```

2. JavaScript 实现

使用 JavaScript 直接定义 merge()函数的代码如下。

```
<script type="text/javascript">
function merge( first, second ) {    //模仿 jQuery 的 merge()函数功能，合并数组
    var i = first.length, j = 0;
    if ( typeof second.length === "number" ) {    //如果第二个参数为数组
        for ( var l = second.length; j < l; j++ ) {    //遍历参数数组 2
            first[ i++ ] = second[ j ];    //逐一把数组 2 中的元素添加到参数数组 1 中
        }
    } else {    //如果第二个参数为类数组的对象
        while ( second[j] !== undefined ) {    //遍历该对象
            first[ i++ ] = second[ j++ ];    //逐一把对象中的成员添加到参数数组 1 中
        }
    }
    first.length = i;    //重设数组 1 的 length 属性值
    return first;    //返回合并后的数组
}
//应用 merge()函数的示例
window.onload = function(){
    var arr1 = [1,2,3,["a", "b", "c"]];
    var arr2 = [4,5,6,[7,8,9]];
    arr3 = merge(arr1, arr2);
    alert(arr1);
    alert(arr1.length);
    alert(arr3);
    alert(arr3.length);
}
</script>
```

9.3.7 删除数组中的重复项

在 DOM 操作中，如果合并两个 jQuery 对象，可能会存在重复的 DOM 元素对象。为此，jQuery 专门定义了 unique()函数，该函数可以把重复的 DOM 元素删除掉。

考虑到 JavaScript 数组中可能会存在相同数值的元素，因此 jQuery 把该函数的功能限制在只能处理删除 DOM 元素数组，而不能处理字符串或者数字数组。

unique()函数用法简单，它能够把传递进来的参数数组进行过滤，并删除重复的 DOM 对象元素。例如，在下面的示例中，变量 arr1 存储了 3 个 DOM 元素，而 arr2 存储了 2 个 DOM 元素，合并之后，其中两个 DOM 元素是重复的，调用 unique()函数之后，则删除了这两个重复的选项，从而使合并后的数组中仅包含 3 个 DOM 对象。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
    var $arr1 = $("#u1 li");
    var $arr2 = $(".red");
    var $arr3 = jQuery.merge($arr1, $arr2);
    var $arr4 =  jQuery.unique($arr3);
    alert($arr1.length);    //返回 3
    alert($arr3.length);     //返回 3
    alert($arr4.length);     //返回 3
})
</script>

<ul id="u1">
    <li>1</li>
    <li>2</li>
    <li class="red">3</li>
</ul>
<ul id="u2">
    <li class="red">4</li>
    <li>5</li>
    <li>6</li>
</ul>
```

9.4 多库共存

所谓多库共存，就是多个 JavaScript 技术库被引入到同一个页面后，如何确保它们不发生名字冲突。例如，jQuery 定义$符号代表 jQuery 对象，而 Prototype 技术库也引用了$名字空间。如果把它们直接导入到同一个文档中，就可能会引发名字空间的混乱。为此，jQuery 提供了多库共存的技术解决途径。

9.4.1 解决$名字冲突

jQuery 定义了 noConflict()函数工具，调用该工具可以把变量$的控制权交给第一次实现它的库或者代码。为了方便理解这个工具的应用，我们先看下面这个示例。

```
<script type="text/javascript">
var $ = function(){
    alert("其他库别名");
}
</script>
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
```

```
    alert("jQuery库别名");
})
</script>
```

在这个示例中，先于 jQuery 库之前命名一个$变量，为该变量定义一个简单的函数。然后导入 jQuery 库，再调用$()函数，则可以看到浏览器根据最后导入的 jQuery 库的名字空间来执行$()函数，如图 9.5 所示。

如果希望执行 jQuery 库前面的$()或者其他库名字空间中的$()函数，则只需要在导入 jQuery 库后的脚本中调用 jQuery.noConflict()函数即可。例如，针对上面的示例，可以按如下方法来设计，则在浏览器中浏览时，可以看到最先定义的$()函数有效，如图 9.6 所示。

```
<script type="text/javascript">
var $ = function(){
    alert("其他库别名");
}
</script>
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery.noConflict();    //恢复早先定义的$名字空间
$(function(){
    alert("jQuery库别名");
})
</script>
```

图 9.5　最后导入的库覆盖前面的库

图 9.6　最先导入的库覆盖后面的库

通过这种方式可以确保 jQuery 不会与其他库的$对象发生冲突，在运行 jQuery.noConflict()函数之后，就只能使用 jQuery 变量访问 jQuery 对象。例如，在要用到$()的地方，就必须换成 jQuery()。

jQuery.noConflict()函数必须在导入 jQuery 库之后，并且在导入另一个导致冲突的库之前使用。当然也应当在其他冲突的库被使用之前使用，除非 jQuery 是最后一个导入的。

分析 noConflict()函数的源代码(如下所示)，可以看到 noConflict()函数实际上是把备份的$变量进行恢复，恢复到最初的状态。

```
noConflict: function( deep ) {
    window.$ = _$;
        if ( deep )
            window.jQuery = _jQuery;
    return jQuery;
},
```

9.4.2 解决 jQuery 名字冲突

如果 jQuery 名字空间也发生了冲突，可以使用 jQuery.noConflict(deep)函数进行解决，它是上一节介绍的 noConflict()函数的高级版本，当参数 deep 为 true 时，该函数能够把$和 jQuery 的控制权都交还给原来的库，因此将完全重新定义 jQuery。

继续看下面的示例，在这个示例中没有调用 jQuery.noConflict(deep)函数，因此最后执行的依然是 jQuery 框架的名字空间。

```
<script type="text/javascript">
var jQuery = function(){
   alert("其他库名");
}
</script>
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery(function(){
   alert("jQuery 库名");
})
</script>
```

现在，调用 jQuery.noConflict()函数，并向其传递一个 true 参数，则 jQuery 会使用内部变量_jQuery 恢复 jQuery 库之前的最初功能。

例如，在下面的示例中，定义全局变量 jQuerySelf 暂存 jQuery 名字空间，并通过 jQuery.noConflict(true);函数恢复 jQuery 最初的名字空间语义。所以，在下面示例中将看到如何避免库冲突，同时又能够实现库之间相安无事，可以在同一个文档中交叉使用。

```
<script type="text/javascript">
var jQuery = function(){
   alert("其他库名");
}
</script>
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
var jQuerySelf = jQuery.noConflict(true);
jQuery(function(){    //将执行其他库名字空间
   alert("jQuery 库名");
})
jQuerySelf(function(){    //将执行 jQuery 库名字空间
```

```
    alert("jQuery库名");
})
</script>
```

jQuery.noConflict(deep)函数的源代码可以参阅上节介绍的jQuery.noConflict()函数，其实现原理很简单，即如果参数值为true，则使用临时变量_jQuery恢复它的最初功能。

9.5 数据缓存

提及缓存，读者可能会联想到客户端浏览器中的缓存，或者服务器端的缓存。客户端缓存是存在浏览者电脑硬盘上的，即存在浏览器临时文件夹中，而服务器缓存是存在服务器内存上的，当然在一些高级应用场合也有专门的缓存服务器，甚至有利用数据库进行缓存的实现。

jQuery在1.2.3版本中开始加入缓存功能，并通过data()方法来实现。实际上Prototype与Mootools库也有缓存，目的都是用来辅助事件系统，用来缓存其中生成的数据，而非缓存普通函数中上一次计算的结果。Prototype利用它的Hash类，Mootools利用内部的uuid来实现缓存。

一般来说为页面元素设置uuid非常有用，要查找元素时，可以避免重复查找，也可以用于与事件回调函数相绑定。由于uuid目前只有IE支持，即所谓的uniqueID，格式为ms_id\d+，后面的数字叫做uniqueNumber。jQuery也模拟uniqueNumber，而且它的缓存系统非常复杂，支持缓存单个数据和一组数据。

9.5.1 jQuery数据缓存的作用

在jQuery的API帮助文档中，jQuery这样描述数据缓存的作用：用于在一个元素上存取数据而避免了循环引用的风险。

为了理解这句话，我们不妨先看一个示例。在下面这个示例中，数据对象被循环引用，如果数据对象的容量很大，且在文档中多次引用，就会造成系统资源的紧张。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery(function(){
    //被引用的数据
    var userInfo = [{
        "name" : "张三",
        "age" : 12,
        "grade" :1
    },{
        "name" : "李四",
```

```
        "age" : 13,
        "grade" : 2
    }];
    //绑定事件，调用方法读取数据
    $("input").eq(0).click(function(){
        showInfo("张三")
    });
    $("input").eq(1).click(function(){
        showInfo("李四")
    });
    function getData(name){                    //根据 name 字段名检索数据
        for (var i in userInfo){               //遍历数据对象
            if (userInfo[i].name == name){     //过滤数据
                return userInfo[i];
                break;
            }
        }
    }
    function showInfo(name){     //显示数据
        var info = getData(name);
        alert('姓名:' + info.name + '\n' + '年龄:' + info.age + '\n' + '年级:' + info.grade);
    }
})
</script>

<input type="button" value="显示张三的资料" />
<input type="button" value="显示李四的资料" />
```

要优化循环，降低引用的风险，就要重新设计数据结构。在这里我们重写了 userInfo 的 JSON 结构，使 name 与对象 key 直接对应。

```
var userInfo = {
    "张三":{
        "name" : "张三",
        "age" : 12,
        "grade" :1
    },
    "李四":{
        "name" : "李四",
        "age" : 13,
        "grade" : 2
    }
};
```

这样我们就可以直接读取 name 对应的数据，而不需要重复引用数据对象，并进行迭代操作。实现代码如下所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery(function(){
    var userInfo = {
```

```
        //省略，请参阅上面代码
    };
    $("input").eq(0).click(function(){
        showInfo("张三")
    });
    $("input").eq(1).click(function(){
        showInfo("李四")
    });
    function showInfo(name){
       var info = userInfo[name];
       alert('姓名:' + info.name + '\n' + '年龄:' + info.age + '\n' + '年级:' + info.grade);
    }
})
</script>

<input type="button" value="显示张三的资料" />
<input type="button" value="显示李四的资料" />
```

jQuery 正是根据上面示例的简单原理来设计 jQuery 数据缓存系统的。有关 jQuery 数据缓存的实现原理请参阅 9.5.5 节内容。

9.5.2 定义缓存数据

使用 data(name, value)方法可以为 jQuery 对象定义缓存数据。这些缓存数据被存放在匹配的 DOM 元素集合中所有 DOM 元素，同时返回缓存数据的 value。

例如，在下面示例中分别为导航列表中的 li 元素定义缓存数据，即列表选项的类型为 menu，同时为新闻列表中的 li 元素定义缓存数据，即列表选项的类型为 news。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery(function(){
    $("#menu li").data("type","menu");
    $("#news li").data("type","news");
})
</script>

<ul id="menu">
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>
<ul id="news">
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>
```

如果 jQuery 集合指向多个元素，则函数将为所有元素定义缓存数据。data(name，value)

函数在 DOM 元素上可以存放任何格式的数据，而不仅仅是字符串。

9.5.3 获取缓存数据

jQuery 的 data()方法不仅可以定义缓存数据，同时还可以读取 DOM 元素的缓存数据。此时，只需要一个参数即可，该参数指定缓存数据的名称。

例如，针对上面的示例，我们可以分别获取 li 元素列表中的数据，并根据 type 缓存数据的值，分别显示不同的信息，演示效果如图 9.7 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery(function(){
    $("#menu li").data("type","menu");
    $("#news li").data("type","news");
    $("li").each(function(index){
        if($(this).data("type") == "menu"){
            $(this).text("导航菜单" + (index + 1))
        }
        else if($(this).data("type") == "news"){
            $(this).text("新闻列表" + (index + 1))
        }
    });
})
</script>

<ul id="menu">
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>
<ul id="news">
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>
```

如果读取的缓存数据不存在，则返回的值为 undefined；如果 jQuery 集合指向多个元素，则将只返回第一个元素的对应缓存数据。

该函数可以用于在一个元素上存取数据，从而避免了循环引用的风险。jQuery.data 是 jQuery 1.2.3 版的新功能。读者可以在很多地方使用这个函数，jQuery UI 经常调用该函数。

图 9.7 获取缓存数据在程序中的应用

9.5.4 删除缓存数据

removeData()函数能够删除指定名称的缓存数据，并返回对应的 jQuery 对象。例如，在下面的示例中将删除导航列表内 li 元素的 type 缓存数据。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery(function(){
    $("#menu li").data("type","menu");
    $("#news li").data("type","news");
    $("li").each(function(index){
        if($(this).data("type") == "menu"){
            $(this).removeData("type");
        }
        else if($(this).data("type") == "news"){
            $(this).text("新闻列表" + (index + 1))
        }
    });
})
</script>
```

9.5.5 jQuery 数据缓存的 JavaScript 实现原理

jQuery 数据缓存的实现方法其实很简单，如果打开 jQuery 技术框架源代码，搜索“data: function(elem, name, data)”关键词，可以找到下面一段代码。

```
var expando = "jQuery" + now(), uuid = 0, windowData = {};
jQuery.extend({
    cache: {},
    data: function( elem, name, data ) {
        elem = elem == window ?    //如果元素为 window 对象，则设置 elem 为 windowData 对象
            windowData :
            elem;
```

```
        var id = elem[ expando ];    //为当前元素定义一个数据属性，并传递给 id 变量
        if ( !id )    //如果不存在 id，则重赋一个动态值
            id = elem[ expando ] = ++uuid;
        //如果缓存数据对象中未存在特定数据的属性，则重设该属性
        if ( name && !jQuery.cache[ id ] )
            jQuery.cache[ id ] = {};
        //如果存在数据值参数，则把它存储到数据缓存对象
        if ( data !== undefined )
            jQuery.cache[ id ][ name ] = data;
        //如果存在属性名，则返回数据缓存对象中的对应属性，否则返回 id 值
        return name ?
            jQuery.cache[ id ][ name ] :
            id;
    },
```

jQuery 首先声明 cache 缓存对象，初始化该对象为空。实际上，它的结构如下所示。

```
cache = {
    "uuid1":{
        "name1":value1,
        "name2":value2
    },
    "uuid2":{
        "name1":value1,
        "name2":value2
    }
}
```

每个 uuid 对应一个 elem 缓存数据，每个缓存对象可以是由多个 name/value 对组成的，而 value 可以是任何数据类型。例如，可以为 elem 存一个 JSON 数据片段。

```
$(elem).data('JSON':{
    "name":"张三",
    "age":12
})
```

expando 作为 elem 的一个新加属性，是为了防止与用户自定义的变量产生冲突，这里采用可变后缀的方式，从而最大限度地避开名字冲突。

下面我们使用 JavaScript 来模拟 jQuery 的数据缓存实现方法，并通过示例可以看到该数据缓存的作用原理。实际上，jQuery 通过一个公共数据缓存对象来存储不同 DOM 元素的数据，并通过一定方法区分不同元素的数据。为了方便元素访问自身的数据，jQuery 利用该元素及一定的算法得到一个复合属性名，并利用这个复合属性名从公共缓存对象中读取自己的数据。在下面的示例中，先直接通过 JavaScript 方式定义一个 data()函数，然后根据传进的元素对象，把特定名/值对数据添加到 cache 对象中。读写时，通过一定的算法区分不同元素的缓存数据，避免重复引用和迭代操作所造成的系统资源紧张，下面示例的演示效果如图 9.8 所示。

```
<script type="text/javascript">
var expando = "jQuery" + now(),    //获取随机扩展数
uuid = 0,     //uuid起始值
windowData = {};    //特殊顶级对象
cache = {},    //公共缓存对象
function data( elem, name, data ) {    //DOM元素的缓存数据读写函数
    elem = elem == window ?
        windowData :
        elem;
    var id = elem[ expando ];
    if ( !id )
        id = elem[ expando ] = ++uuid;
    if ( name && !cache[ id ] )
        cache[ id ] = {};
    if ( data !== undefined )
        cache[ id ][ name ] = data;
    return name ?
        cache[ id ][ name ] :
        id;
}
function now(){    //以当前时间为基础获取一个随机数
    return +new Date;
}
window.onload = function(){
    var div1 = document.getElementById('div1');
    var div2 = document.getElementById('div2');
    data(div1, "tag", "div");     //为div1添加缓存数据
    data(div1, "id", "div1");    //为div1添加缓存数据
    div1.innerHTML = "div1.tag = " + data(div1, "tag") + "<br />" + "div1.id = " + data(div1,
"id")
    var div2 = document.getElementById('div2');
    div2.innerHTML = "div2.tag = " + data(div2, "tag") + "<br />" + "div2.id = " + data(div2,
"id")
}
</script>

<div id="div1">盒子 1</div>
<div id="div2">盒子 2</div>
```

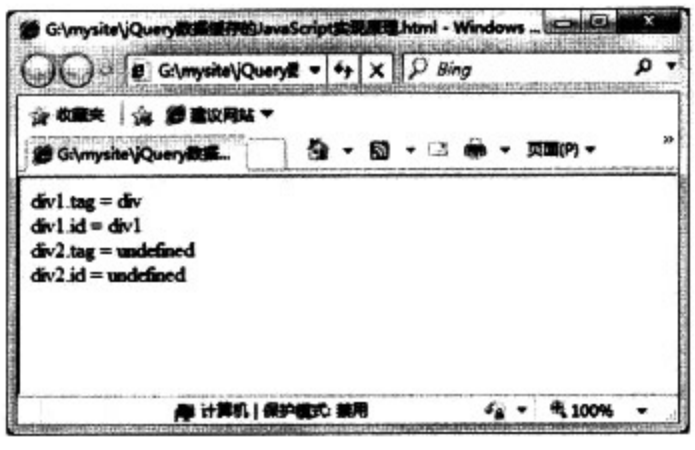


图 9.8　JavaScript 实现的数据缓存效果

为了方便jQuery对象的操作，jQuery又把全局函数data()绑定到jQuery.fn原型对象中，从而实现在jQuery对象上直接调用data()方法。绑定代码如下所示。

```
jQuery.fn.extend({
    data: function( key, value ){
        var parts = key.split(".");
        parts[1] = parts[1] ? "." + parts[1] : "";
        if ( value === undefined ) {
            var data = this.triggerHandler("getData" + parts[1] + "!", [parts[0]]);
            if ( data === undefined && this.length )
                data = jQuery.data( this[0], key );
            return data === undefined && parts[1] ?
                this.data( parts[0] ) :
                data;
        } else
            return this.trigger("setData" + parts[1] + "!", [parts[0], value]).each
(function(){
                jQuery.data( this, key, value );
            });
    },
    })
```

9.5.6 jQuery数据缓存的使用规范

由于jQuery缓存对象是全局对象，因此在Ajax应用中，由于页面很少被刷新，缓存对象将会一直存在。随着调用data()函数操作次数增多，或者因使用不当，会使得cache对象急剧膨胀，最终影响程序的性能。

所以在使用jQuery数据缓存功能时，应及时清理缓存对象。jQuery提供了removeData()函数帮助用户手动清除数据。根据jQuery框架的运行机制，下面几种情况不需要手动清除数据缓存。

- 对elem执行remove()操作，jQuery会自动清除对象可能存在的缓存。
- 对elem执行empty()操作，如果当前elem子元素存在数据缓存，jQuery也会清除子对象可能存在的数据缓存，因为jQuery的empty()操作的实现其实是循环调用remove()删除子元素。
- jQuery复制节点的clone()方法不会复制data缓存，也就是说jQuery不会在全局缓存对象中分配一个新节点存放新复制的elem缓存。

jQuery在clone()方法中把可能存在的缓存指向的属性(即elem的expando属性)替换成空。如果直接复制这个属性，就会导致原elem和新复制的elem都指向一个数据缓存，中间的互操作都将会影响到两个elem的缓存变量。

把数据缓存一起复制有时候也是很有用的。例如，在拖动操作中，当单击源目标elem节

点时就会复制一个显示为半透明的 elem 副本拖动，并把 data 缓存复制到拖动层中，等到拖动结束，就可能取到当前拖动的 elem 相关信息。现在 jQuery 方法没有给我们提供这样的处理，不过我们在复制源目标的 data 时，可以把这些 data 都重新设置到复制出来的 elem 中，这样在执行 data(name, value)方法时，jQuery 会在全局缓存对象中为我们开辟新空间。实现代码如下。

```
if (typeof($.data(currentElement)) == 'number') {
    var elemData = $.cache[$.data(currentElement)];
    for (var k in elemData) {
        dragingDiv.data(k, elemData[k]);
    }
}
```

在上面代码中，$.data(elem,name,data)包含三个参数，如果只有一个 elem 参数，这个方法将返回它的缓存 key(即 uuid)，利用这个 key 就可以得到整个缓存对象，然后把对象的数据都复制到新的对象中。第二个参数表示数据的名称，第三个参数表示数据的内容。

9.6 数 据 队 列

队列是一种特殊的线性列表结构，它只允许在表的前端(front)进行删除操作，而在表的后端(rear)进行插入操作。允许插入操作的一端被称为队尾，允许删除操作的一端被称为队头，队列中没有元素时，称为空队列。在队列这种数据结构中，最先插入的元素必定最先被删除，反之最后插入的元素将最后被删除，因此队列又称为“先进先出”(FIFO—first in first out)的线性表。

jQuery 支持数据队列，并通过定义 queue()方法实现对队列的完整操作。这对于一系列需要按次序执行的函数特别有用。例如，animate 动画、Ajax 异步请求和交互以及 timeout 等需要一定时间的函数。

实际上，jQuery 把队列看作是 elem 对象的数据缓存工具。但是它与 data()函数实现的数据缓存在很大差异，因为队列中存储的是将要被执行的一连串的动作函数。

9.6.1 添 加 队 列

jQuery 定义了 queue()方法，该方法能够把函数加入队列，这里的队列通常是一个函数数组。当为同一个元素上设计连续动画时，如多次执行 animate()方法。jQuery 会自动将其加入名为 fx 的函数队列中。但是，如果需要对于多个元素依次执行动画，就必须借助 queue()方法手动设置队列。

具体地讲，queue()方法能够在匹配元素的队列最后添加一个函数，并调用该函数。其用

法如下。

```
queue(name, callback)
```

第一个参数 name 表示队列的名称，可以省略，默认值为 fx；第二个参数 callback 表示添加到队列末尾的回调函数。调用该方法之后，将在匹配元素的队列末尾添加一个函数，并执行该函数，最后返回当前 jQuery 对象。

例如，在下面的示例中，在按钮的 click 事件中定义了 6 个动作，其中第 3 个和第 5 个动作是通过 queue()方法手动添加到队列中的。

但是，由于 queue()方法是在队列末尾添加一个函数，则在该行后面的动作都将被忽视。所以读者会看到，当在浏览器中预览时，小方块滑动到最右侧后面，调用末尾添加的队列函数之后，就停止了响应，如图 9.9 所示。

```
<style type="text/css">
.bg {
    background:blue;
}
div{
    position:absolute;
    width:50px; height:50px;
    background:red; left:0; top:50px;
    display:none;
}
</style>
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery(function(){
    var  $div = $("div");
    $("input").click(function(){
          $div.slideDown("slow");
          $div.animate({left:'+=400'},2000);
          $div.queue(function(){    //在队列的末尾添加一个函数
              $(this).addClass("bg");    //则调用该回调函数之后，动画将停止
          });
          $div.animate({left:'-=400'},2000);
          $div.queue(function(){
              $(this).removeClass("bg");
          });
          $div.slideUp("slow");
    });
})
</script>

<input type="button" value="动画演示" />
<div></div>
```

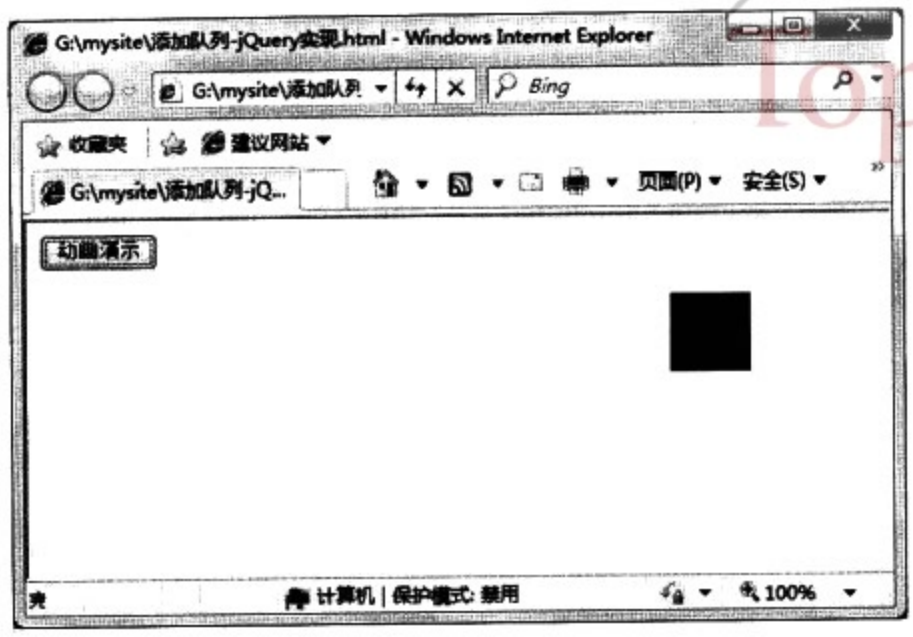


图 9.9 使用 queue()方法在匹配元素队列末尾添加一个响应函数

9.6.2 获取队列

当为匹配的元素添加队列之后，可以使用 queue(name)方法获取对该队列的引用。这里的队列实际上就是一个函数数组，并能够自动连续执行。参数 name 表示队列名称，一般默认为 fx。

例如，在下面的示例中获取 div 元素默认的 fx 队列，并查询该队列中包含多少函数成员。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery(function(){
    var  $div = $("div");
    $("input").click(function(){
         $div.slideDown("slow");
         $div.animate({left:'+=400'},2000);
         $div.animate({left:'-=400'},2000);
         $div.slideUp("slow");
         var x = $div.queue();    //获取 div 元素默认的队列 fx
         alert(x.length);    //显示 fx 队列包含 4 个函数成员
    });
})
</script>

<input type="button" value="动画演示" />
<div></div>
```

如果匹配的元素不止一个，则返回指向第一个匹配元素的队列，即返回第一个元素包含的函数数组。

9.6.3 替换队列

一个队列执行完毕之后，可以使用另一个队列进行替换，具体实现方法是在 queue()方法的第二个参数中传递一个队列，将匹配元素的队列使用一个新的队列来代替，即使用新的函数数组代替现在已执行的函数数组。

例如，在下面的示例中，分别为 div 元素设计两个动画序列，其中第一个为默认的 fx 动画序列，它直接被绑定在第一个按钮的 click 事件处理函数中，该动画序列包含四个动作函数，按顺序作用于 div 元素，分别为慢速显示、慢速前进、慢速后退和慢速隐藏元素。第二个动画序列通过 queue()方法定义，序列名称为 fa，该序列中包含四个动作函数，按顺序作用于 div 元素，分别为快速显示、快速前进、快速后退和快速隐藏元素。然后使用 queue()方法获取名称为 fa 的动画序列，并调用 queue()方法使用 fa 动画序列替换 fx 动画序列，演示效果如图 9.10 所示。

```
<style type="text/css">
.bg {
    background:blue;
}
div{
    position:absolute;
    width:50px; height:50px;
    background:red; left:0; top:50px;
    display:none;
}
</style>
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
    <script type="text/javascript">
jQuery(function(){
    var  $div = $("div");
    $("input").eq(0).click(function(){    //默认的第一个动画序列，慢速动画
        $div.slideDown("slow");
        $div.animate({left:'+=400'},4000);
        $div.animate({left:'-=400'},4000);
        $div.slideUp("slow");
    });
    $div.queue("fa",function(){    //自定义动画序列，快速动画
        $div.slideDown("fast");
        $div.animate({left:'+=400'},200),
        $div.animate({left:'-=400'},200)
        $div.slideUp("fast ");
    });
    var fa = $div.queue("fa");    //获取对自定义动画序列的引用
    $("input").eq(1).click(function(){
         $div.queue("fx",fa);    //使用 fa 动画序列覆盖默认的 fx 动画序列
    });
```

```
})
</script>

<input type="button" value="执行慢速动画队列" />
<input type="button" value="替换慢速动画，执行快速动画队列" />
<div></div>
```

图 9.10　替换队列函数

注意，在动画序列执行过程中，并不是立即进行替换，而是等到当前正在执行的动作完成之后，才停止正在执行的 fx 序列，并继续执行第二个 fa 动画序列。

在 queue(name, queue)方法中，如果第二个参数是一个空数组，则将会清除原来的动画序列。例如，下面的代码将清空匹配的 div 元素的默认动画序列。

```
$("div").queue("fx", []);
```

9.6.4　删除队列函数

dequeue()方法能够删除指定队列中最顶部的函数，并执行这个队列函数。实际上，dequeue()方法是将函数数组中的第一个函数取出来，并执行这个函数。那么当再次执行 dequeue()方法时，得到的就是另一个函数了，如果不执行 dequeue()方法，则队列中的下一个函数将永远不会执行。

dequeue()方法包含一个参数，用来指定队列的名称，默认为 fx。例如，在下面的示例中(样式表代码请参阅上面示例)，使用 dequeue()方法结束自定义队列函数，并使队列继续进行下去。这样动画将会连续播放，直到最后一个函数被执行为止。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery(function(){
    var  $div = $("div");
    $("input").click(function(){
         $div.slideDown("slow");
```

```
            $div.animate({left:'+=400'},2000);
            $div.queue(function(){
                $(this).addClass("bg");
                $(this).dequeue();
            });
            $div.animate({left:'-=400'},2000);
            $div.queue(function(){
                $(this).removeClass("bg");
                $(this).dequeue();    //删除最顶部的函数，并继续执行队列
            });
            $div.slideUp("slow");
        });
    })
    </script>

    <input type="button" value="动画演示" />
    <div></div>
```

9.7 内 核 工 具

jQuery 是一个类数组结构的对象，但是它不是数组，我们可以把它视为散列表结构的数据集合，更准确的讲是一个 DOM 元素集合。为了方便访问这个数据集合，jQuery 定义了一套工具，使用这些工具可以模拟数组的访问方式，同时方便用户遍历 jQuery 对象，以便对其中的 DOM 元素进行操作。

9.7.1 遍历 jQuery 对象

jQuery 为 jQuery 对象定义了 each()方法，实现对 jQuery 数据集合进行遍历，并能够以每一个匹配的元素作为上下文来执行一个函数。该方法的语法格式如下。

```
each(callback)
```

其中参数 callback 表示一个可执行的回调函数，并在每个匹配的元素上执行。这意味着，每次执行传递进来的函数时，函数中的 this 关键字总是指向一个不同的 DOM 元素(每次都是一个不同的匹配元素)。

例如，在下面示例中使用 jQuery 获取文档中所有的 li 元素，然后为这个 jQuery 对象调用 each()方法，在参数回调函数中，使用回调函数的参数 index 重写当前元素包含的内容，则可以看到每个元素在 jQuery 对象集合中的序号，如图 9.11 所示。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
jQuery(function(){
```

```
    $("li").each(function(index){
        this.innerHTML = index;
    })
})
</script>

<ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>
```

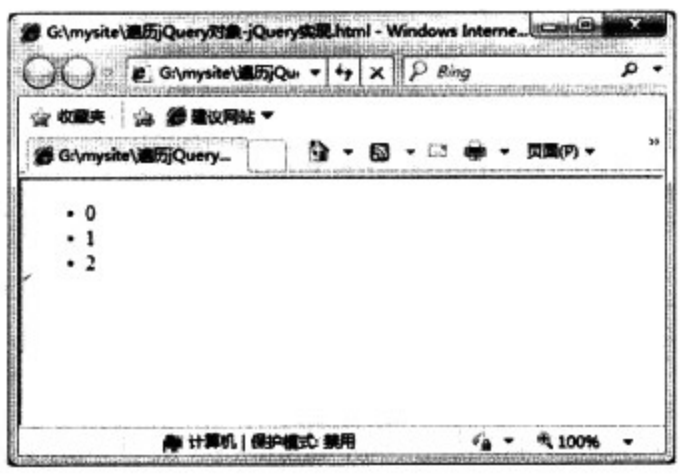


图 9.11　遍历列表结构选项

如果希望中途停止 each()方法的迭代操作，则可以在 callback 回调函数中设置返回值为 false，这样将自动停止循环，如同在普通循环中使用 break 语句一样。如果返回值为 true，将跳至下一个循环，如同在普通的循环中使用 continue 语句。

9.7.2　遍历 jQuery 对象的 JavaScript 实现

each()方法是 jQuery 框架的核心工具，该方法的实现很简单，jQuery 首先在 jQuery 名字空间下定义了一个全局函数 each()。代码如下所示。

```
each : function( object, callback, args ){
    var name, i = 0, length = object.length;
    if ( args ){    //如果存在第三个参数
        if ( length === undefined ) {    //如果对象集合不是数组类型
            for ( name in object )    //遍历对象
                if ( callback.apply( object[ name ], args ) === false )    //调用回调函数
                    break;
        }
        else    //如果对象集合是数组类型
            for ( ; i < length; )    //遍历数组
                if ( callback.apply( object[ i ++ ], args ) === false )  //调用回调函数
                    break;
    }
    else{    //如果不存在第三个参数
        if ( length === undefined ){    //如果对象集合不是数组类型
```

```
            for ( name in object )     //遍历对象
                if ( callback.call( object[ name ], name, object[ name ] ) === false )
//调用回调函数
                    break;
        }
        else    //如果对象集合是数组类型
            for ( var value = object[0];     //遍历数组
                i < length && callback.call( value, i, value ) !== false; //调用回调函数
                value = object[ ++ i] ) {
        }
    }
    return object;     //返回 jQuery 对象
},
```

然后在 jQuery.fn 对象上绑定原型方法。

```
each: function( callback, args ) {
    return jQuery.each( this, callback, args );
},
```

9.7.3 获取 jQuery 对象的元素个数

jQuery 模仿数组的 length 属性，为 jQuery 对象都定义了 length 属性，该属性能够反馈当前 jQuery 对象包含的 DOM 元素的个数。例如，在下面的示例中 jQuery 对象的 length 属性值为 3，即包含 3 个 li 元素。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
   alert($("li").length);
})
</script>

<ul>
   <li></li>
   <li></li>
   <li></li>
</ul>
```

jQuery 在 length 属性基础上还封装了 size()方法，该方法的返回值与 length 属性值是完全相同的，其封装代码如下所示。

```
jQuery.fn = jQuery.prototype = {
   size: function() {
      return this.length;
   }
}
```

9.7.4 获取选择器和选择范围

从 1.3 版本开始，jQuery 新增了 selector 和 context 属性，其中 selector 属性能够返回传给 jQuery()的原始选择器，而 context 属性能够返回传给 jQuery()的原始的 DOM 节点内容，即 jQuery()函数的第二个参数，如果没有指定，默认指向当前的文档对象(document)。

简单地说，selector 和 context 属性能够返回你用什么选择器来找到这个元素的。这两个属性对插件开发人员很有用，可以用于精确检测选择器的查询情况。

例如，在下面的示例中，对于$("li",ul)对象来说，它的 selector 属性值等于"li"，而 context 属性值等于 DOM 元素对象，即节点名称为 UL 的元素。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
   var ul = $("ul")[0];
   alert($("li",ul).selector);
   alert($("li",ul).context.nodeName);
})
</script>

<ul>
   <li></li>
   <li></li>
</ul>
```

9.7.5 获取 jQuery 对象的元素

jQuery 为了方便 jQuery 对象与 DOM 集合之间相互转换而定义了 get()方法，该方法能够把 jQuery 对象转换为 DOM 元素集合，即把 jQuery 集合对象转换为真正意义上的数组，以方便操作。

例如，在下面这个示例中，调用 get()方法把 jQuery 转换为 DOM 数组集合，然后调用 reverse()方法，颠倒数组中的元素排序，最后再把这个集合插入到 ul 元素中。在浏览器中可以看到这个选项列表的顺序发生了倒置。

```
<script src="images/jquery-1.3.2.js" type="text/javascript" ></script>
<script type="text/javascript">
$(function(){
        var $li = $("li"); //获取 jQuery 对象
        var li = $li.get();     //转换为 DOM 集合
        li.reverse(); //调用数组方法，颠倒数组元素顺序
        $("ul").html(li);  //重叠 ul 的选项列表结构
```

```
})
</script>

<ul>
   <li>1</li>
   <li>2</li>
   <li>3</li>
</ul>
```

get()方法还可以包含一个 index 参数，该参数可以接受一个自然数，表示从 jQuery 对象中取得其中一个匹配的元素。index 表示 jQuery 对象内的元素下标位置，即取得第几个匹配的元素。这能够让你选择一个实际的 DOM 元素并对它进行直接操作，而不是通过 jQuery 方法或者函数。

实际上，get(index)方法与 jQuery 使用对象下标读取其中的元素对象是等同的，例如：

```
$(this).get(3)
```

与

```
$(this)[3]
```

上面两种用法的结果是完全相同的。

如果希望获取指定元素在 jQuery 对象中的位置，可以使用 index()方法来获取。该方法包含一个 DOM 元素对象，并根据这个元素搜索与之匹配的元素，并返回相应元素的索引值。如果找到了匹配的元素，则从 0 开始返回，如果没有找到匹配的元素，则返回-1。

第10章 使用jQuery打造Ajax异步交互式动态网站

本章是一个综合案例，将从零起步讲解如何创建一个含有相册展示的网站，并使用 jQuery 改善图片展示的交互性，提升用户体验。我们将通过本章案例的实践操作来帮助读者认识 jQuery 在网站开发中的作用，同时我们还将介绍开发一个网站的基本流程和方法，其中涉及大量的 HTML 和 CSS 知识。作为一位优秀的网页设计师来说，HTML、CSS 和 JavaScript 都是最基本的技术，只有充分把这些技术结合起来，才能够自如地开发动态网站。

10.1 案例背景介绍

这是一个电子相册网站，网站能够分类展示用户的数码照片。本案例能够根据需求允许用户对照片进行分类，并根据分类显示照片。在浏览照片过程中，示例提供了缩微图、小图和原图三种视图效果。

整个示例以 HTML+CSS+JavaScript+jQuery+XML 技术混合进行开发，遵循结构、表现、逻辑和数据完全分离的原则进行设计。结构层由 HTML 负责，在结构内不包含其他层代码；表现层完全独立，并实现表现层皮肤定制功能；CSS 代码兼容 IE 6、IE 7、IE 8 和 FF 等主流浏览器，符合 Web 标准设计的规范；逻辑层使用 JavaScript+jQuery 技术配合进行开发，充分发挥各自优势，以实现最优化代码编辑原则，其中数据的导入由 JavaScript 负责，数据解析由 jQuery 负责。

示例所要呈现的数据完全独立，并以 XML 格式进行存储，数据容量可以自由增减，不受程序和页面结构的限制。本示例不需要后台服务器技术的支持，因此对于广大初学者来说，可以在本地或远程上自由进行调试和运行。本案例主要包括下面几个交互功能。

- 照片按类展示，方便浏览者浏览，如图 10.1 所示。
- 照片视图包括缩微图、小图和灯箱视图，以提高照片浏览的效果，如图 10.2 所示。

图 10.1 缩微图浏览效果

图 10.2 灯箱浏览效果

- 图片分类可以任意定制，分类设置和显示不受页面结构限制，如图 10.3 所示。
- 可以定制相册皮肤，实现用户自己决定相册的肤色，如图 10.4 所示。

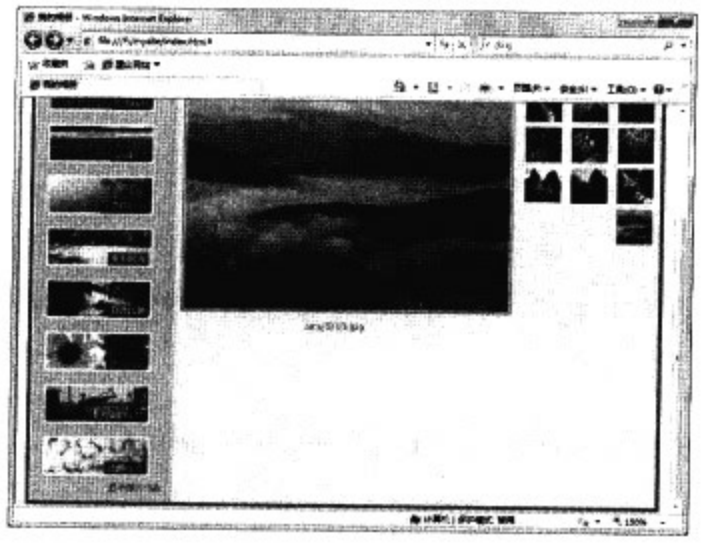


图 10.3 分类浏览

图 10.4 皮肤定制

10.2 网站设计思路

整个示例的设计思路可以按如下几个部分进行讲解。

- 结构部分：由 HTML 负责。结构代码存放在 index.html 文件中，负责构建页面的基本骨架。
- 布局部分：由 CSS 负责。页面样式代码存放在 images/javasript.js、images/colour_blue.css(蓝色皮肤)、images/colour_ green.css(绿色皮肤)、images/colour_orange.css(橙色皮肤)、images/colour_pink.css(粉红色皮肤)和images/colour_purple.css(紫色皮肤)中。
- 脚本部分：由 JavaScript 和 jQuery 负责。脚本代码存放在 images/javasript.js 中，负责异步数据读写、动态交互设计。本示例使用的 jQuery 框架版本为 jquery-1.3.2，可兼容使用 jquery-1.4.2。但是早期 jQuery 版本的语法可能会略有不同。
- 数据部分：由 XML 负责。存放在 pics 文件夹中。左侧分类导航信息存放在 pics/class.xml 文件中，每个子文件夹表示一类照片，文件夹的名称可以自由设置。

10.3 结构设计

本网站遵循 Web 2.0 的设计风格，因此页面仅包括 index.html 一个文件。该文件相当于一个展示容器，所有图片都将在这个容器中展示。该文档的结构代码及其说明请参阅下面代码。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/
xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

```
<title>我的相册</title>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<!-- 导入页面样式 -->
<link rel="stylesheet" type="text/css" href="images/style.css" />
<!-- 导入默认的皮肤样式主题 -->
<link rel="stylesheet" type="text/css" href="images/colour_pink.css" />
<!-- 导入 jQuery 框架源代码 -->
<script language="javascript" type="text/javascript" src="images/jquery-1.3.2.js">
</script>
<!-- 导入页面脚本 -->
<script language="javascript" type="text/javascript" src="images/javasript.js">
</script>
</head>
<body>
<!-- 页面包含框 -->
<div id="wrap">
    <!-- 标题栏结构 -->
    <div id="logo">
        <h1>我的相册</h1>    <!-- 主标题 -->
        <h2 id="links">联系主人<a href="#">Email</a><br />    <!-- 副标题 -->
            访问主人<a href="#">Blog</a></h2>
    </div>
    <!-- 导航栏结构 -->
    <div id="nav">
        <!-- 导航栏选项列表 -->
        <ul>
            <li><a class="selected" href="#">相册</a></li>
            <li><a href="#">简介</a></li>
            <li><a href="#">留言</a></li>
        </ul>
        <!-- 皮肤控制按钮 -->
        <div id="colours">
            <a href="#"><img src="images/pink.png" alt="粉红色皮肤" /></a>
            <a href="#"><img src="images/blue.png" alt="蓝色皮肤" /></a>
            <a href="#"><img src="images/green.png" alt="绿色皮肤" /></a>
            <a href="#"><img src="images/purple.png" alt="紫色皮肤" /></a>
            <a href="#"><img src="images/orange.png" alt="橙色皮肤" /></a>
        </div>
    </div>
    <!-- 图片包含框 -->
    <div id="main">
        <!-- 左侧分类栏 -->
        <div id="side_menu"></div>
        <!-- 右侧浏览框 -->
        <div id="content">
            <!-- 图片标题 -->
            <h3><span class="title">相册 1</span><span class="sub">[鼠标移过缩微图可以放大
浏览]</span></h3>
            <!-- 大图显示区域 -->
            <div id="gallery"><span class="big_pic" id="big_pic" title="单击灯箱可以放大
浏览"><img src="pics/1/6.jpg" alt="默认图片" /></span><span class="big_title">pics/1/
```

```
6.jpg</span>
                <!--  右侧缩微图包含框  -->
                <span id="thumbs"></span>
            </div>
        </div>
    </div>
</div>
<!--  灯箱  -->
<div id="lightbox">
    <div><img src="pics/1/6.jpg" /></div>
    <div><input name="" type="button" value=" 关闭 " id="idBoxCancel" /></div>
</div>
</body>
</html>
```

10.4 样式设计

网站结构仅是一个框架，还需要使用 CSS 让其看起来更加漂亮，更方便浏览。好的样式设计既要保证页面看起来很漂亮、富有个性，同时还应该确保页面在不同浏览器下都能够呈现相同或者相似的浏览效果，即所谓的浏览器兼容性。本案例的 CSS 样式能够确保在 IE 家族和 FF 等主流浏览器中都可以呈现相近的浏览效果。

本示例的 CSS 样式被分为两个部分：第一部分是基本样式，这部分样式存放在 style.css 文件中，详细说明见第 10.4.1 节。第二部分是皮肤样式，这部分样式分别存放在 colour_blue.css (蓝色主题)、colour_ green.css(绿色主题)、colour_orange.css(橙色主题)、colour_pink.css(粉红色主题)和 colour_purple.css(紫色主题)文件中。然后借助 JavaScript 脚本动态控制所要导入的主题样式表文件。

10.4.1 基本样式

基本样式主要包括默认样式(即标签样式)、结构布局和类样式，详细说明如下所示。

```
/* 统一标签默认样式
这样做的好处就是方便整个页面样式的统一，
同时省却了在文档其他部位重复设置这些样式
*/
body {                          /*文档默认属性*/
    font-family: verdana, arial, sans-serif;    /*统一页面字体类型*/
    padding: 0;                 /*清除页边距*/
    margin: 0;                  /*清除页边距*/
    font-size:.75em;            /*统一页面字体大小*/
    text-align:center;          /*默认文本居中显示*/
    color: #656565;             /*默认字体中性灰色显示*/
}
```

```
p {                                /*清除段落文本默认的上下间距*/
    margin: 0;
}
img {                              /*清除图片在附加超链接时所呈现的粗边框*/
    border:none;
}
a {                                /*清除超链接默认的下划线样式*/
    text-decoration:none;
}
ul, ol, dl, dt, dd, li {           /*统一列表结构中的样式*/
    margin: 0;                     /*清除缩进*/
    padding: 0;                    /*清除缩进*/
    list-style: none;              /*清除项目符号*/
}
/* 下面的样式代码实现页面布局
主要包括：基本结构布局、标题栏样式、导航布局、图片框和灯箱样式
*/
/* 基本框架
----------------------------------------*/
#wrap {                            /*定制页面包含框*/
    width:880px;                   /*固定宽度显示*/
    margin:0 auto;                 /*居中显示*/
    text-align:left;               /*恢复文本左对齐*/
}
#logo {/*固定标题栏高度和样式*/
    height: 60px;
    border-top:6px solid;          /*镶嵌一条顶部线条*/
}
#nav {/*导航栏样式*/
    height: 38px;                  /*固定高度显示*/
    border-bottom: 6px solid;      /*镶嵌一条底部线条*/
    clear:both;                    /* 兼容 FF，避免并列显示 */
}
#main {                            /*图片框控制*/
    overflow:auto;                 /*允许自由伸展*/
    border: 6px solid;             /*镶边*/
    border-top:none;               /*清除顶部镶边*/
    background:url(bg.gif) repeat-y center;     /*设计伪列布局，模拟 2 栏布局效果*/
    min-height:438px;              /*规定最低高度，避免浏览框伸缩*/
    zoom:1;                        /* 兼容 IE 6，不能够撑开高度 */
}
html>/**/body #main  {             /* 兼容 FF，边框样式覆盖问题 */
    border-width:6px;
    border-style:solid;
}
* html #main {                     /* 兼容 IE 6，不支持最小高度 */
    height:438px;
}
/* 网页标题
----------------------------------------*/
#logo h1 {                         /*标题样式*/
```

```
    font-size: 300%;              /*字体大小*/
    padding: 4px 0 0 20px;        /*定位*/
    float: left;                  /*向左浮动显示*/
    filter:alpha(opacity=40);     /*半透明显示，兼容 IE*/
    -moz-opacity:0.4;             /*半透明显示，兼容 FF*/
    opacity:0.4;                  /*半透明显示，兼容其他标准浏览器*/
    margin:0;                     /* 兼容 FF，避免上下间距太大 */
}
#links {                          /*副标题样式*/
    margin: 0 9px 0 0;            /*控制外边距*/
    font-size: 90%;               /*字体大小*/
    text-align: right;            /*右对齐显示*/
    padding: 20px 10px 0 0;       /*固定位置*/
    text-transform: uppercase;    /*大写样式*/
    font-weight:normal;           /*清除加粗显示样式*/
}
#links a, #links a:hover {        /*副标题的超链接样式*/
    padding: 0 0 2px 0;           /*定位*/
}
/* 页面导航栏目
----------------------------------------*/
#nav li {                         /*导航栏选项样式*/
    float: left;
}
#nav li a {                       /*导航栏选项超链接样式*/
    display: block;               /*块状显示，以方便设置样式*/
    float:left;                   /*兼容 IE 6，块状元素的宽度为 100%，会占据一行显示 */
    height: 32px;                 /*固定高度*/
    line-height:32px;             /*垂直居中处理*/
    text-decoration: none;        /*清除下划线*/
    padding: 6px 16px 0 16px;     /*定位*/
    border-right: 1px solid;      /*增加分隔线*/
    font-weight: bold;            /*加粗显示*/
}
#colours {                        /*皮肤控制按钮框样式*/
    text-align: right;
    padding: 16px 16px 0 0;
}
/* 左侧分类
----------------------------------------*/
#side_menu {                                  /*左侧栏的外框样式*/
    padding: 22px 12px 12px 12px;             /*定位*/
    float: left;                              /*向左浮动*/
    width: 170px;                             /*固定宽度*/
    margin-top:2em;                           /*增加与顶部的距离*/
    text-align:right;                         /*文本右对齐*/
}
#side_menu div {                              /*设置左侧分类显示框的样式*/
    padding: 4px;                             /*调整空隙*/
    margin:2px auto;                        /*居中显示，并增加上下分类的间距*/
    float:left;                             /*浮动显示*/
```

```
    position:relative;          /*相对定位，以方便内部文本绝对定位时作为参考*/
}
#side_menu a {                  /*分类超链接的样式*/
    float: left;                /*向左浮动*/
    border:solid 2px #bbb;      /*为分类栏目镶边*/
}
#side_menu a:hover {            /*设计超链接的动态样式*/
    border-color:#efefef;       /*鼠标经过时动态显示边框颜色*/
}
#side_menu img {                /*设计分类图像的边框样式*/
    border:solid 3px #efefef;
}
#side_menu span {{              /*设计左侧分类栏中的标题文本样式*/
    position:absolute;          /*绝对定位，这样可以任意控制其位置*/
    right:12px;                 /*偏右侧 12 像素靠齐*/
    bottom:10px;                /*偏底部 10 像素靠齐*/
    padding:2px 4px;            /*调整空隙，使文本居中显示*/
    color:#eee;                 /*字体颜色为白色*/
    background:#444;            /*背景色为深灰色*/
    filter:alpha(opacity=60);   /*增加透明效果，兼容 IE*/
    -moz-opacity:0.6;           /*增加透明效果，兼容 FF*/
    opacity:0.6;                /*增加透明效果，兼容其他浏览器*/
}
#side_menu p {                  /*设计分类图像的样式*/
    cursor:pointer;             /*鼠标经过时手形显示*/
    display:inline;             /*以行内显示，方便居中显示*/
}
/* 右侧图像浏览区域
----------------------------------------*/
#content {                      /*图像预览框样式*/
    float: left;                /*向左浮动*/
    padding:12px 8px 6px 8px;   /*调整显示位置*/
}
html>/**/body #content  {
    float: right;               /* 兼容 FF，调整浮动方向 */
    width:648px;                /* 兼容 FF，不能够自动伸展宽度 */
}
#content h3 {                   /*图像预览框标题样式*/
    font-weight:normal;         /* 清除默认的加粗显示样式 */
    padding: 6px;               /* 通过内边距调整显示位置 */
    margin: 0 0 6px 0;          /* 通过外边距调整底部的空隙 */
    border-bottom: 1px solid;   /* 增加一条下划线效果 */
    font-size: 180%;            /* 适当放大字体显示 */
}
.sub {                          /* 设置标题右侧的说明性文字的样式*/
    font-size: 12px;            /* 固定字体大小 */
    padding-left: 12px;         /* 增加与左侧标题的距离 */
}
#gallery {                      /* 图像浏览框样式 */
    position: relative;         /* 相对定位，以方便内部缩微图定位 */
    padding-top: 8px;           /* 调整顶部的空隙 */
```

```
    width:100%;                    /* 兼容 IE，避免出现滚动条 */
}
#thumbs {                          /* 缩微图浏览框样式 */
    width: 205px;                  /* 固定宽度 */
    float: right;                  /* 浮动到右侧显示 */
}
#thumbs a {                        /* 设置缩微图超链接样式 */
    float: right;                  /* 向右浮动 */
    margin: 1px 0 3px 10px;        /* 增加缩微图之间的间距 */
    width: 50px;                   /* 固定宽度 */
    height: 50px;                  /* 固定高度 */
    border: 2px solid #FFF;        /* 增加边框 */
    position:relative;             /* 相对定位，以方便内部缩微图定位 */
}
#thumbs a img {                    /* 设置缩微图样式 */
    width: 50px;                   /* 固定宽度 */
    height: 50px;                  /* 固定高度 */
}
#thumbs a:hover img {              /* 设置鼠标经过时缩微图样式，通过这种方式可以模拟 JavaScript 脚
本设计图像放大和缩小的动态效果 */
    width: 56px;                   /* 适当放大宽度 */
    height: 56px;                  /* 适当放大高度 */
    position:absolute;             /* 绝对定位，以便使放大的缩微图能够居中显示 */
    left:-5px;                     /* 向左侧外部移动 5 个像素 */
    top:-5px;                      /* 向顶部外侧移动 5 个像素 */
    border:2px solid #fff;         /* 改变边框样式，产生动态效果 */
}
.big_pic {                         /* 设置放大图像显示框的样式 */
    position: absolute;            /* 绝对定位，以方便控制其显示位置 */
    right: 196px;                  /* 距离右侧的偏移位置 */
    top: 8px;                      /* 距离顶部的偏移位置 */
}
.big_pic img {                     /* 设置放大图像显示样式 */
    width: 450px;                  /* 固定宽度显示 */
    height: 320px;                 /* 固定高度显示 */
    padding: 2px;                  /* 增加 2 个像素的空隙 */
    border: 2px solid #8A8A8A;     /* 设计边框样式 */
}
.big_title {                       /* 设置放大图像的注释文本样式 */
    position: absolute;            /* 绝对定位，以便精确控制其显示位置 */
    right: 400px;                  /* 距离右侧的位置 */
    top: 348px;                    /* 距离顶部的位置 */
}
#lightbox {                        /* 隐藏灯箱包含框显示 */
    display:none;
}
.hidden {                          /* 隐藏类样式 */
    display:none;
}
```

10.4.2 主题皮肤样式

皮肤样式根据主题分别放在不同主题的样式表文件中，包括 colour_blue.css(蓝色主题)、colour_ green.css(绿色主题)、colour_orange.css(橙色主题)、colour_pink.css(粉红色主题)和colour_purple.css(紫色主题)。这些主题文件的样式基本相似，用来设置页面结构中的边框颜色、字体颜色和背景颜色，下面以默认的 colour_pink.css(粉红色主题)皮肤样式进行说明。

```
#main {/* 图像浏览框的边框颜色 */
    border: #DF368F;
}
#nav li a {/* 导航栏超链接的字体颜色和背景色 */
    color: #EEE;
    border-color: #A8A8A8;
}
#logo {/* 标题顶部镶边线颜色 */
    background: transparent;
    border-color: #DF368F;
}
#logo h1 {/* 标题字体的颜色 */
    color: #DF368F;
}
#content h3 {/* 图像浏览框标题颜色和底边线颜色 */
    background: transparent;
    color: #8A8A8A;
    border-color: #DF368F;
}
#links a, #content a {/* 设置文档中超链接的字体颜色 */
    background: transparent;
    color: #DF368F;
}
#nav {/* 设置导航栏背景色、字体颜色和边框颜色 */
    background: #BE7B9E;
    color: #EEE;
    border-color: #DF368F;
}
#nav li a:hover, #nav li a.selected, #nav li a.selected:hover {/* 导航栏选项超链接字体颜
色、背景颜色和边框颜色 */
    background: #DF368F;
    color: #EEE;
    border-color: #A8A8A8;
}
.title {/* 图像标题颜色 */
    color:#DF368F;
}
```

10.5 网站脚本设计

在开始设计之前，我们先理清整个页面需要设计的脚本功能。

- 主题样式动态控制
- 数据异步载入
- 分类信息显示和动态响应
- 缩微图显示和动态响应
- 灯箱广告

围绕这些功能，下面我们就来逐步进行分解。

10.5.1 主题样式动态控制

主题样式的设计核心就是动态导入 CSS 样式表文件。具体设计思路是：页面初始化之后获取皮肤切换按钮，然后使用 for 语句遍历每个按钮，并分别为它们绑定 click 事件。在 click 事件处理函数中，先读取当前按钮包含的信息。利用 JavaScript 脚本截取图像名称字符串，如 purple.png 中的“purple”字符串。

使用 getAttribute("href")方法获取 link 元素的 href 属性信息，通过动态改变 link 元素的 href 属性值实现动态载入主题样式表文件的目的。详细代码及其说明如下。

```
//页面初始化
window.onload = function(){
    // 获取<div id="colours">包含框中所有 a 元素的引用指针
     var color = document.getElementById("colours").getElementsByTagName("a");
    // 获取页面中第二个 link 元素的引用指针
     var linkcss = document.getElementsByTagName("link")[1];
     // 遍历所有 a 元素(皮肤控制按钮)
     for(var i=0 ;i<color.length; i++){
        //为每一个 a 元素绑定鼠标单击事件处理函数
        color[i].onclick = (function(i){
            //为了能够在循环体内正确向处理函数传递循环变量值,
            //这里定义了一个闭包结构，并在闭包结构中设置了一个返回函数,
            //因为 onclick 属性赋值必须为函数体
            return function(){
                //获取当前 a 元素包含的 img 元素的引用指针
                var img = color[i].getElementsByTagName("img");
                //获取当前 a 元素包含图像的 src 属性值
                var src =img[0].getAttribute("src");
                //获取 URL 中名称前面的斜杠位置
                var a = src.lastIndexOf("/");
```

```
                //获取 URL 中扩展名前的点号位置
                var b = src.lastIndexOf(".");
                //利用上面两个序号值截取图像名称字符串
                src = src.substring(a+1,b);
                //获取 link 元素的 href 属性值，
                //并利用正则表达式技术将截取的图像名称字符串替换掉 href 属性值原来的值。
                //例如，假设 href 属性值为“images/colour_pink.css”，
                //而图像名称字符串为“orange”，
                //则所要替换而得到的新 href 属性值为“images/colour_orange.css”，
                //并把这个新值保存在变量 newcss 中
                var newcss = linkcss.getAttribute("href").replace(/(\w+)\_(\w+)(\.css)/,
"$1_"+src+"$3 ");
                //设置 link 导入的外部样式表文件的 href 属性值为 newcss，
                //从而实现动态改变导入的外部样式表文件，最终实现动态换肤功能
                linkcss.setAttribute("href",newcss);
            }
        })(i);
    }
}
```

10.5.2 导入外部数据

首先，读者应该在 index.html 文件头部位置导入 jQuery 技术框架文件。代码如下。

```
<script  language="javascript"  type="text/javascript"  src="images/jquery.  js">
</script>
```

然后，再导入本示例的脚本文件 javasript.js。代码如下。

```
<script  language="javascript"  type="text/javascript"  src="images/javasript.js">
</script>
```

JavaScript 语言本身是不能够读写 XML 文件的，必须使用 XML DOM 控件来实现，目前主流浏览器都支持该控件，但是我们要考虑兼容性问题，以确保本案例在 IE 和 FF 等主流浏览器中能正常使用。实现代码如下。

```
//初始化 XML DOM 控件，并加载指定的 XML 文件
//参数 xmlpath 表示外部 XML 文件的路径
function loadXML(xmlpath){
    //声明并初始化变量
    var xmlDoc=null;
    //兼容 IE 浏览器
    if (window.ActiveXObject){
       xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
    //兼容 FF 浏览器
    }else if (document.implementation && document.implementation. createDocument){
       xmlDoc=document.implementation.createDocument("","",null);
```

```
        //否则提示错误
        }else{
            alert('你的浏览器暂时不支持 XML DOM 控件');
        }
        //禁止异步通信
        xmlDoc.async=false;
        //加载数据
        xmlDoc.load(xmlpath);
        //返回加载的数据
        return xmlDoc;
    }
```

上面的函数能够根据不同浏览器类型采用不同方法实例化控件，最后调用该实例对象的load()方法加载参数中指定的XML文件。

10.5.3 分类导航设计

分类导航栏位于页面的左侧，该栏显示的分类信息是从pics/class.xml文件中动态读取的。class.xml文件负责存储分类信息，XML数据格式如下。

```
<?xml version="1.0" encoding="utf-8"?>
<pics url="pics">
    <folder name="1" class="class.jpg">热带风情</folder>
    <folder name="2" class="class.jpg">高山风光</folder>
    <folder name="3" class="class.jpg">异国风情</folder>
    <folder name="4" class="class.jpg">冬日风光</folder>
    <folder name="5" class="class.jpg">野外风景</folder>
    <folder name="6" class="class.jpg">花儿灿烂</folder>
    <folder name="7" class="class.jpg">充实的乡村</folder>
    <folder name="8" class="class.jpg">天高云淡</folder>
</pics>
```

其中pics表示根节点，folder是子节点，folder中包含每个分类文件夹(照片分类)的信息，当前节点的name属性表示分类文件夹的名称，class属性表示分类导航的图标，folder节点所包含的信息为当前分类的标题。该文件可以允许自由增加folder子节点的数目，以实现自由增减相册分类。在设计分类导航栏的显示功能时，读者应该考虑下面几个问题。

- 页面初始化时自动显示默认的导航数据。
- 动态绑定事件处理函数，当在导航栏中单击某类图标时，缩微图包含框中的信息能够自动更新，从而实现信息联动显示效果。
- 动态显示分类信息的数目，初步实现固定条目和所有条目两种显示方式。

为了实现上述功能，我们可以定义两个功能函数，详细说明如下所示。

- 定义initMenu()函数显示导航图。该函数能够根据所指定的xmlpath和more参数决

定所要显示的导航图信息以及信息记录数。

```
//显示导航图
//参数说明
//xmlpath 参数表示外部 XML 文件
//more 参数表示是否显示全部数据
function initMenu(xmlpath,more){
    //调用 loadXML()函数加载 XML 文件
    var oxml=loadXML(xmlpath);
    //清空分类包含框内信息
     $("#side_menu").empty();
    //遍历加载的 XML 文档中的 folder 节点
    $(oxml).find("pics > folder").each(function(i){
    //如果显示记录数大于四条，且参数 more 为 false，则跳出遍历结构
         if(i>4&&more!=true){return false;};
        var temp_str;
        //利用从节点读取的数据组合 HTML 结构字符串
        temp_str= "<div><a href='#' title='"+$(this).attr("name")+"'><img src='pics/"+
$(this).attr("name")+"/"+$(this).attr("class")+"' alt='"+$(this).text()+"' /></a><span>"+
$(this).text()+"</span></div>";
        //把该字符串应用到导航包含框中
        $(temp_str).appendTo("#side_menu");
    });
    //如果记录数大于五条，则可以考虑提供如下操作选项，否则不提供
     if($(oxml).find("pics > folder").length>5){
    //如果参数 more 为 false，则在底部插入“全部分类”按钮，
    //并在该按钮中绑定本函数调用，并传递参数 more 的值为 true
        if(more!=true){
            temp_str="<p onclick='initMenu(\"pics/class.xml\",true);'>全部分类</p>";
            //把超链接结构插入到包含框导航信息的底部
            $(temp_str).appendTo("#side_menu");
        }
        //如果参数 more 为 true，则在底部插入“显示部分分类”按钮，
        //并在该按钮中绑定本函数调用，传递参数 more 的值为 false
        if(more==true){
            temp_str="<p  onclick='initMenu(\"pics/class.xml\",false);'>显示部分分类
</p>";
            $(temp_str).appendTo("#side_menu");
        }
    }
    //最后调用该函数为所有导航信息项绑定鼠标单击事件处理函数
    bindMenuEvent();
}
```

在上面的函数中，先调用 loadXML()函数加载指定的 XML 文件，同时使用 jQuery 方法清除分类导航包含框内的所有导航信息。然后利用 jQuery 获取导入的 XML 文件中 folder 节点集合，并使用 each()方法遍历该集合。

在遍历过程中，判断参数 more 是否为 true，如果为 true，则说明仅显示前五条记录项。

利用 jQuery 方法获取当前节点所包含的信息，并组成一个 HTML 结构的字符串，字符串中包含节点所要传递的数据。最后把这个字符串插入到分类导航包含框中，从而实现动态绑定分类信息的目的。

为了方便用户操作，在分类导航的末尾插入一个文本按钮，用来决定是否显示所有数据。在函数的结尾调用 bindMenuEvent()函数为导航图像绑定 click 事件处理函数。

- bindMenuEvent()函数为导航图标绑定事件。在 initMenu()函数体末尾调用 bindMenuEvent()函数，以便为分类信息绑定 click 事件处理函数。

```
//为导航图标绑定事件
function bindMenuEvent(){
    //遍历分类导航超链接
    $("#side_menu a").each(function(i){
        //为该超链接绑定鼠标单击事件处理函数
        $("#side_menu a")[i].onclick = (function(i){
            //返回闭包函数
            return function(){
                //获取超链接中的 title 属性
                var url = $($("#side_menu a")[i]).attr("title");
                //清空缩微图包含框中的信息
                $("#thumbs").empty();
                //再次调用该函数，使用新的分类目录信息初始化显示缩微图
                initThumbs("pics.xml","pics/"+url+"/");
            };
        })(i);
    });
}
```

在上面的函数中，使用 jQuery 对象的 each()方法遍历分类导航包含框中的每个导航图标，并分别为它们绑定 click 事件处理函数。在该事件处理函数中，先获取导航图标包裹的超链接元素 a 的 title 属性值，该属性中包含了每个分类相册的目录地址。然后清空缩微图包含框，再次调用 initThumbs()函数，使用当前分类的目录地址再次初始化显示缩微图。

最后，为了实现页面初始化时能够显示分类导航信息，不要忘记在页面初始化事件中调用 initMenu()函数。代码如下。

```
window.onload = function(){
    //默认的导航图标
    initMenu("pics/class.xml");
}
```

10.5.4 缩微图显示

与分类导航信息显示一样，缩微图的显示也包含这样两个功能：动态导入数据和动态绑定 click 事件。但是缩微图还包括当单击缩微图时，显示大图的操作。因此，我们可以先定义

三个功能函数：显示缩微图函数(initThumbs())、为缩微图绑定事件函数(bindThumbsEvent())和显示大图函数(showBigImg())。

(1) initThumbs()负责显示缩微图。代码和说明如下所示。

```
//显示缩微图
//参数说明
//xmlpath 表示要加载的 XML 文件的名称
//url 表示要加载的 XML 文件的路径
//function initThumbs(xmlpath,url){
    //加载 XML 数据
    var oxml=loadXML(url+xmlpath);
    //遍历加载数据中的 file 节点
    $(oxml).find("pics file").each(function(){
        var temp_str;
        //获取 file 节点中的属性和信息，组成缩微图 HTML 结构代码的字符串
        temp_str= "<a href='#'><img src='pics/"+$(this).parent().attr("name")+"/t"+
$(this).text()+"' title='"+$(this).text()+"' alt='"+$(this).text()+"' /></a>";
        //获取 file 节点中的 class 属性值，改写大图浏览区标题
        $(".title").text($(this).parent().attr("class"));
        //把组成的缩微图 HTML 结构代码应用到缩微图包含框中
        $(temp_str).appendTo("#thumbs");
    });
    //为每个缩微图绑定事件处理函数
    bindThumbsEvent();
}
```

在上面的函数中，先调用 loadXML()函数加载参数中指定的目录和文件(pics.xml)。pics.xml(pics/子目录/pics.xml)负责存储指定分类的照片信息，XML 的数据格式如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<pics url="pics">
    <folder name="4" class="冬日风光">
        <file>1.jpg</file>
        <file>2.jpg</file>
        <file>3.jpg</file>
        ……
        <file>9.jpg</file>
        <file>10.jpg</file>
    </folder>
</pics>
```

上述 XML 数据结构是以 pics 为根节点，子节点 folder 表示当前分类信息(当前目录)，其中 name 属性表示目录的名称，class 属性表示当前分类的标题。注意，本文件 XML 结构与 class.xml 文件结构中的节点和属性名称所表示的语义是不同的。

在 folder 节点下包含很多个 file 节点，它们分别表示每个图片文件的信息，file 节点包含的文本为每个照片的完整名称。我们还可以根据需要为 file 节点增加各种属性，如图片大小

和说明等。

在 pics 相片目录下的每个子文件夹中都包含一个同名的 pics.xml 文件，文件中存储着当前子目录所有照片的信息。同时每个子目录中还应保存增加了前缀(t)的同名缩微图，图为 jpg 格式，大小为 50×50 像素左右。

然后，使用 jQuery 遍历 XML 文件中的 file 节点，获取该节点中的相关属性和信息，并利用这些信息组合一个缩微图的 HTML 结构字符串，最后把该字符串应用到缩微图包含框中。

最后，利用分类导航中的说明信息改写大图浏览区中的标题文本。当遍历完成之后，再调用 bindThumbsEvent()函数为每个缩微图绑定事件处理函数。

(2) bindThumbsEvent()函数负责为缩微图绑定 click 事件。代码和说明如下所示。

```
//为缩微图绑定事件
function bindThumbsEvent(){
    //遍历缩微图中的超链接
    $("#thumbs a").each(function(i){
        //为超链接绑定鼠标单击事件处理函数
        $("#thumbs a")[i].onclick = (function(i){
            //返回闭包函数
            return function(){
                //获取缩微图的 src 属性值
                var url = $($("#thumbs img")[i]).attr("src");
                //清空大图包含框
                $(".big_pic").empty();
                //利用缩微图的 src 属性值作为参数重新显示大图
                showBigImg(url);
            };
        })(i);
    });
}
```

(3) showBigImg()函数负责显示大图。代码和说明如下所示。

```
//显示大图
function showBigImg(url){
    //获取缩微图中“/t”的位置
    var a = url.lastIndexOf("/t");
    //截取“/t”位置前的字符串
    b = url.substring(0,a);
    //截取“/t”位置后的字符串
    c = url.substring(a+2);
    var temp_str;
    //设置大图的 src 属性值字符串
    temp_str= "<img src='"+b+"/"+c+"' alt='"+c+"' />";
    //把大图应用到预览框中
    $(temp_str).appendTo(".big_pic");
}
```

上面的函数设计大图预览功能，当用户单击缩微图时，会调用该缩微图被绑定的事件处理函数，并把缩微图的 src 属性值作为参数传递给 showBigImg()函数，从而实现显示大图的效果。

10.5.5 灯箱广告

灯箱广告(Lightbox)是 JavaScript 应用中的一类特殊效果，简单地说就是锁定屏幕禁止任何操作，仅显示特定的对话框让用户操作。灯箱特效在图片浏览中经常使用，它通过设计一个覆盖层覆盖屏幕，然后设置覆盖层的样式为半透明显示，并显示在最上层，在该层上面再显示可以操作的工作层。如果单击关闭灯箱按钮，则清除覆盖层，并隐藏操作层，恢复电脑屏幕原来的显示效果。

首先，定义一个构造函数 lightBox()，模仿 jQuery 构造器设计方法，在构造函数中调用它的原型方法 init() 创建一个显示层和一个覆盖层，并为 lightBox 构造器定义两个原型方法 show()(显示灯箱)和 close()(关闭灯箱)。详细代码如下所示。

```
//灯箱构造器
var lightBox = function(){
//调用该构造器的原型方法 init()，初始化灯箱效果
    this.init.apply(this,arguments);
}
//灯箱构造器原型对象
lightBox.prototype = {
    //灯箱构造器初始化方法
    init : function(id) {
        //显示层
        if(!id && !(typeof id === "string")) return false;
        this.box = document.getElementById(id);   //获取灯箱框
        this.box.style.zIndex = 10001;   //设置覆盖的 z 轴坐标，确保位于上面
        this.box.style.position = "absolute";//绝对定位显示
        this.box.style.display = "none"; //初始化为隐藏显示
        //覆盖层
        this.lay = document.body.insertBefore(document.createElement("div"), document.
body.childNodes[0]);  //创建一个 div 元素
        this.lay.style.display = "none"; //初始化为隐藏显示
        this.lay.style.backgroundColor = "#000";  //设置背景色为黑色
        this.lay.style.zIndex = 10000;   //设置覆盖的 z 轴坐标，确保位于显示层的下面
        this.lay.style.position = "fixed";    //以固定定位显示
        this.lay.style.left = 0;     //x 轴坐标
        this.lay.style.top = 0; //y 轴坐标
        this.lay.style.width = "100%";   //宽度，覆盖整个屏幕
        this.lay.style.height = "100%";  //高度，覆盖整个屏幕
        if(document.all){   //设置覆盖层透明度，兼容 IE
            this.lay.style.filter = "alpha(opacity:60)";
        }
        else{  //设置覆盖层透明度，兼容其他浏览器
```

```
            this.lay.style.opacity = 0.6;
        }
    },
    //显示灯箱
    show: function(options){
        this.lay.style.display = "block";     //显示灯箱覆盖层
        this.box.style.display = "block";     //显示灯箱
        var top = document.documentElement.scrollTop - this.box.offsetHeight / 2;
     //居中定位
        var left =  document.documentElement.scrollLeft - this.box.offsetWidth / 2;
     //居中定位
        //预防图像过大，影响显示效果
        //如果图像高度不是很大，则可以考虑居中显示
        if(top>-300){
            this.box.style.marginTop = document.documentElement.scrollTop - this.box.
offsetHeight / 2 + "px";
            this.box.style.top = "50%";
        }
        //否则不再居中显示
        else{
            this.box.style.top =  "50px";
        }
        //如果图像宽度不是很大，则可以考虑居中显示
        if(left>-512){
            this.box.style.marginLeft = document.documentElement.scrollLeft - this.box.
offsetWidth / 2 + "px";
            this.box.style.left = "50%";
        }
        //否则不再居中显示
        else{
            this.box.style.left =  "20px";
        }
    },
    //关闭灯箱
    close: function() {
        this.box.style.display = "none"; //隐藏灯箱
        this.lay.style.display = "none"; //隐藏覆盖层
    }
};
```

建造一个简单的构造器之后，下面再定义一个功能函数，在这个函数中创建构造器的实例，然后获取文档中的灯箱包含框，并把它传递给灯箱构造器。同时为灯箱包含框中的关闭按钮绑定 click 事件，在该事件处理函数中调用灯箱构造器实例中的 close()方法；同时为图像预览绑定 click 事件，在该事件处理函数中调用灯箱构造器实例的 open()方法，显示灯箱效果，并把要显示的大图传递给灯箱包含框。实现代码如下所示。

```
//灯箱广告调用函数
function lightbox(){
    var box = new lightBox("lightbox");
```

```
    $("#idBoxCancel").click(function(){ box.close(); })
    $("#big_pic").click(function(){
            var url = $(".big_pic img").attr("src");
            $("#lightbox img").attr("src",url);
            box.show();
    })
    $("#big_pic").css("cursor","pointer");
}
```

最后，在页面初始化处理函数中调用灯箱广告函数。代码如下。

```
window.onload = function(){
    //初始化设置
    initMenu("pics/class.xml");
    initThumbs("pics.xml","pics/1/");
    //调用灯箱特效
    lightbox();
}
```

内 容 简 介

本书循序渐进地讲解了jQuery高效开发的方法和技巧，内容包括jQuery框架的设计模式和思路、Sizzle选择器的构成和工作机制、DOM文档操作、事件处理、动画设计、Ajax异步通信、插件扩展和辅助工具等。

执行效率是JavaScript脚本的第一要务，本书在详细讲解jQuery基础知识和技巧用法的同时，重点讲解了如何提高jQuery工作效率，如何混合使用jQuery和JavaScript进行高效开发。此外，本书还深入剖析了jQuery框架的设计模式和选择器的工作机制。全书理论结合实践，通过大量生动的示例帮助读者快速上手。

本书适合Web开发人员阅读和参考，同时也适合广大网页制作和设计爱好者阅读和学习。

图书在版编目(CIP)数据
犀利开发——jQuery内核详解与实践/朱印宏编著. —北京：清华大学出版社，2010.8
(网站开发路线图)
ISBN 978-7-302-23111-0

Ⅰ. ①犀… Ⅱ. ①朱… Ⅲ. ①JAVA语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2010)第144391号

责任编辑：应　勤　张　瑜
封面设计：杨玉兰
责任校对：王　晖
责任印制：孟凡玉
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：北京密云胶印厂
装 订 者：三河市新茂装订有限公司
经　　销：全国新华书店
开　　本：185×260　**印　张**：26.25　**字　数**：620千字
附光盘1张
版　　次：2010年8月第1版　　**印　　次**：2010年8月第1次印刷
印　　数：1～4000
定　　价：49.00元

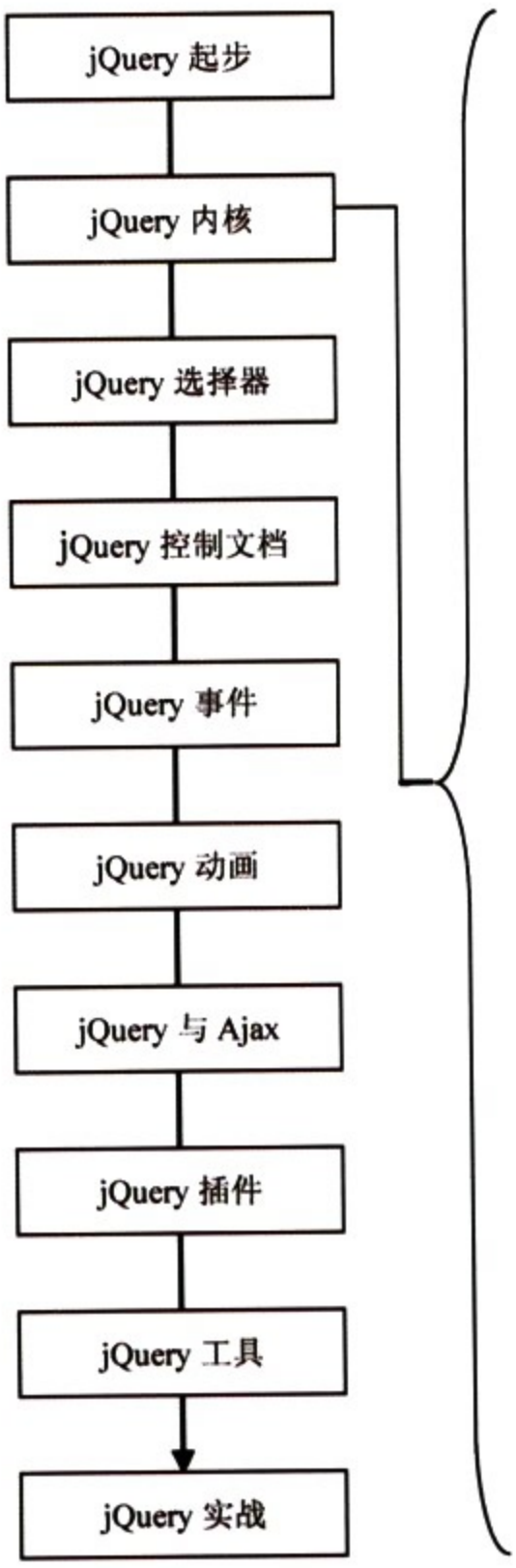
产品编号：034795-01

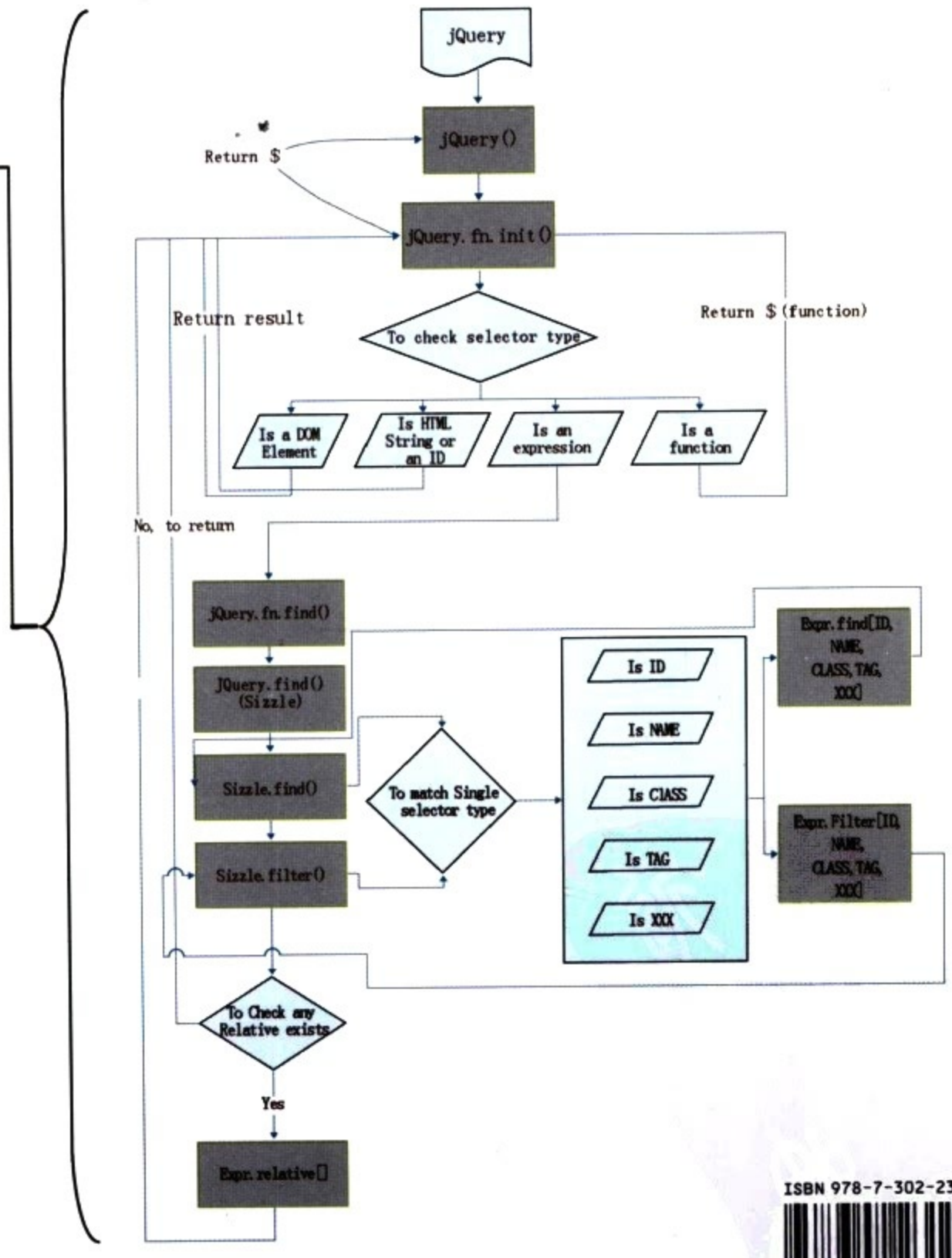
阿基米德说，给我一个支点，我能把地球撬起来。

jQuery说，给你一个方法，你能让互联网炫目富有。

顾名思义，jQuery融合了JavaScript+Query（查询）技术，汲取CSS+Xpath设计模式，让生涩的JavaScript脚本变得敏捷而又犀利。掌握jQuery技术，就能够在JavaScript、DOM、CSS、Event、DHTML、Ajax、Form等技术之间游刃有余，让恼人的Web开发变得轻松而又生动。

本书从破解jQuery技术内核为切入点，由点入面，然后探析jQuery主要功能实现和开发，并比较jQuery技术与原生的JavaScript技术在实现方面的异同，及其执行效率之差异，真正帮助读者知其然，又能知其所以然。最后，通过一个综合案例让读者在实践中快速掌握jQuery开发。





ISBN 978-7-302-23111-0
9 787302 231110 >
定价：49.00元（附光盘1张）