目　录

第一部分　Ajax 基础

第二部分 Ajax 最佳实践

Part 1 第一部分

Ajax 基础

本书将带领你进入Ajax网络应用的新世界。全书特别注重实例，提供了大量可重用的实用示例，所展示的技巧极富实践性，可直接运用于你自己的应用中。为了准备好这次令人兴奋的旅程，整个第一部分会作为其后第二部分各个章节的一个强化预备课程。

第1章论述了Ajax与那些以往惯用的技术的不同之处，并由此展望了全书内容。我们首先讨论了如何用Ajax支持浏览器，以及如何在JavaScript代码里处理异步响应。我们也会看一看Prototype，作为一个非常流行的JavaScript程序库，在整本书中你会一次又一次地看到它的身影。

第2章研究了Ajax请求所能产生的各种响应格式，包括：纯文本、HTML、JSON（JavaScript Object Notation，JavaScript对象记法）、XML，乃至SOAP文档。

第3章深入探讨了每个严肃的Ajax开发者都需要吃透的高级JavaScript技巧。我们研究了JavaScript的对象和函数，解释了如何使用它们来创建自己的JavaScript类，从而利用面向对象技术对Ajax所需的不断增长的客户端代码规模进行控制。你会明白JavaScript的函数是一个比你想象的更为丰富多样的概念。

第4章纵览了各种提供Ajax编程支持的JavaScript库。我们更为深入地考察了元老级选手Prototype库，还考察了全能选手Dojo工具包，以及jQuery——这个Ajax竞技场上初来乍到却令人兴奋的选手。本章最后介绍了DWR，它借助Ajax提供了类似RPC（Remote Procedure Calling，远程过程调用）的能力，实质上就是将Ajax作为一种传输机制来加以利用。

第1章 拥抱Ajax

1

本章内容

- 是什么让Ajax与众不同
- XMLHttpRequest的基本用法
- 运用程序库简化Ajax编程

Ajax在最近一年里[①]迅猛发展。在写这段文字的时候，Ajax正式年龄已经是一岁半了，不过其底层技术已经存在了许多年，只是没有统一的名字来表述它们。Ajax的故事我们已经耳熟能详：故事开始于微软的Outlook的Web版[②]所加入的一个ActiveX控件——现在我们叫它XMLHttpRequest；到2005年2月，Jesse James Garrett创造了术语*Ajax*；然后围绕着Google的Suggest、GMail和Maps应用，人们对这些技术突然爆发出极大的热情。

正如现代社会中的小孩一样，Ajax的成长之路也充满艰辛。比起一年半以前，今天的它看上去已经有了很大不同。无论是技术自身、技术交流所需的语汇，还是用来解决实际问题的工具，都已经发展成熟。对此我们会在1.4节中稍作论述，而到第4章时，我们还会进一步深入探索令Ajax编程更加方便的新一代框架和程序库。

不过最大的变化，则是我们对“Ajax能做什么”的理解已经随着Ajax的成熟而扩展了。开发者正在向自己提出一系列新的问题——超出那些基本的“如何做”的知识，而是转向更为深入和更为广阔的领域，例如：如何管理异步通信？Ajax对应用程序的架构（architecture）有什么影响？甚至还有，Ajax对于我们的商业模式（business model）来说到底意味着什么？

整个开发社区已经热忱地拥抱着Ajax，并在那些最出色的创新项目中以崭新而有趣的方式运用着Ajax。Google已经向我们证明，对于像在线地图和Web邮箱（webmail）那样的“已经解决的问题”，通过运用Ajax，仍然可以做出许多革命性的创新。最近对于“混搭（mash-up）”（在单个页面中混入多个网站的内容）的关注也和Ajax具有天然的联系。

我们已经把Ajax运用到了实际应用和真实环境中，这一路走来也积累了不少实践经验。而我们写作本书的目的也正是为了记录下这些Ajax实践经验，并且超越那些用于概念验证（proof-of-concept）的基础代码，去看一看在现实世界中到底哪些是可行的，哪些是不可行的。正因为如此，

---

① 原书大约写于2006年9月到2007年2月之间。——译者注

② 指微软的Outlook Web Access，这项产品可以让用户通过浏览器以Web方式访问微软Exchange邮件服务器上的电子邮箱。——译者注

我们会特别关注那些切合今天的Ajax的、更为深入和更为广阔的问题。

1.1 Ajax，一种颠覆性的技术

Ajax是一种颠覆性技术（disruptive technology）[①]。它的出现，颠覆了构建和部署在线应用的常规方法，改变着人们认识和使用Web应用的方式。

狭义地说，Ajax不过是向服务器发送异步请求。通过异步（asynchronous），我们让请求在后台运转，而不影响用户界面。当我们编写一个Ajax应用时，只有很小一部分时间花在创建异步的服务器调用和处理服务器响应上，其余时间里，我们在使用既有的技术，如CSS（Cascading Style Sheets，级联样式表）、DOM（Document Object Model，文档对象模型）以及浏览器事件模型。简而言之，我们所使用的正是被统称为动态HTML（Dynamic HTML）的技术杂烩，它两年前差不多还死水一潭，只是被用来呈现形形色色的导航菜单和那些人见人爱的广告弹出窗口。而在杂烩中加入异步HTTP通信的能力，居然使这些技术重获新生，又有了用武之地。为什么Ajax的这么一点功用能有这样巨大的影响呢？下面我们就会看到，答案是出奇地简单。

1.1.1 重新定义用户的工作流程

要理解Ajax对Web开发的影响，关键就在于用户的工作流程。所谓工作流程（workflow），是指用户和应用交互的方式，广义地说，就是用户对应用的体验。我们通常按照浏览器和服务器之间的分工协作来讨论Web应用，但是这些只不过是实现工作的手段而已，真正重要的工作其实在用户的头脑中进行。一个好的应用能够对用户的工作模式加以支持，从而提升用户的生产力；反过来，如果应用按着自身的技术限制来支配用户的工作模式，则会削弱用户的生产力。图1-1展示了在Ajax出现之前的典型的Web应用的工作流程。

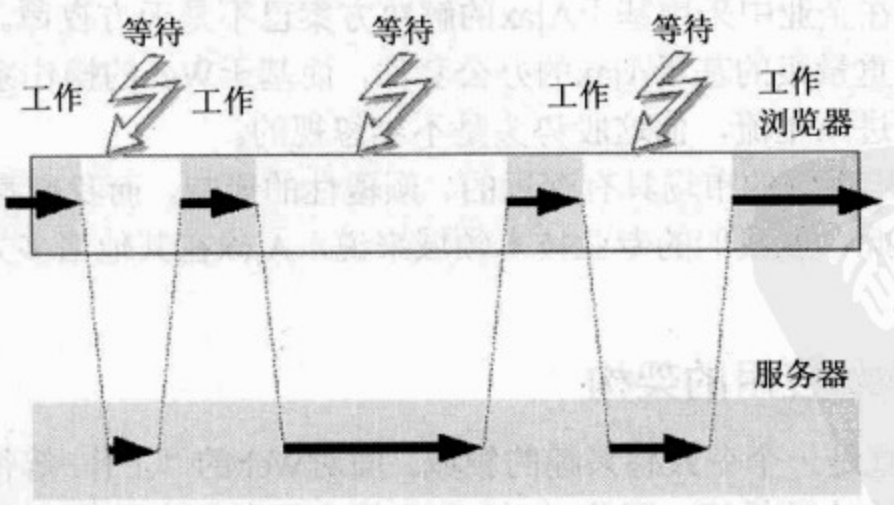


图1-1 传统Web应用中的用户交互模式是“工作-等待”

这一工作流程遵循“工作-等待”模式。也就是说，在任何一点上，浏览器端应用要么是在向用户呈现信息，要么是在等待服务器返回响应。从用户观点来看，体验是断断续续的。在等待期间，

① 英文原文为“disruptive technology”，“颠覆性技术”。1995年克莱顿·克里斯坦森（Clayton M. Christensen）在《哈佛商业评论》上首次提出这一概念。——译者注

用户无法与Web应用进行交互——除了或许能看到一些即将被服务器响应所替换的页面内容。

从优使性（usability）的角度来说，这很有问题。每一次进入等待周期，都会中断用户的思路。而且还必须频繁地进入等待周期——因为大多数Web应用需要频繁地联络服务器。显然，这一模式不适合任何需要解决复杂问题的领域——这实在可惜，因为基于浏览器的应用其实具有很多优势。比如，DHTML[①]提供了舒适迅捷的用户界面的所有要素，而最显著的优势可能是Web应用非常易于部署和维护，因为无需在客户端机器上进行任何安装操作。

在图1-2中，我们展示了Ajax怎样为用户改变工作流程。其中，应用依旧向服务器发送相同的请求，不同的是使用Ajax来发送。这使得用户界面在服务器运转的同时仍能保持活力，这就消除了对用户的注意力的频繁干扰。

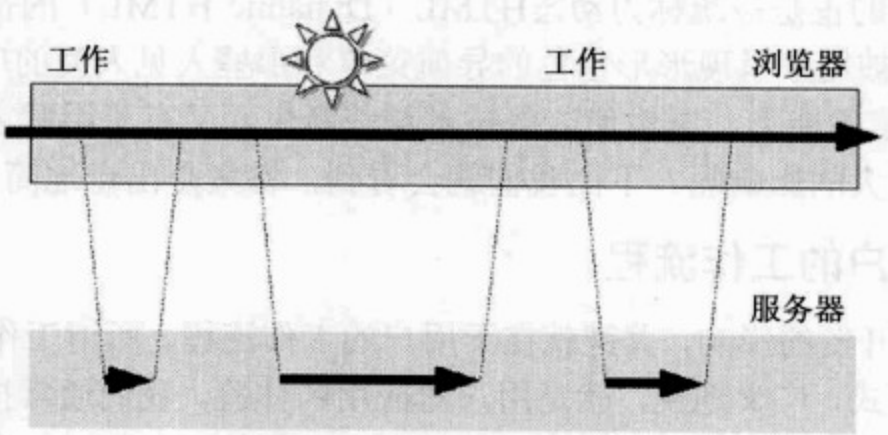


图1-2　Ajax应用中的交互模式是“工作-工作”

从商业视角来说，它的重要性是不言而喻的。Ajax开启了一个庞大的基于浏览器的业务线（line-of-business）应用[②]的新市场，所以它不仅正在颠覆Web开发的原有格局，也在冲击着胖客户端和桌面应用世界。在企业中采用基于Ajax的解决方案已不是天方夜谭。过去的一年中，在因特网上已经出现了一些重量级的基于Ajax的办公套件，而基于Web的操作系统也正在开发中。尽管它们之中还没有一个进入主流，但这股势头是不容忽视的。

可以说，Ajax对应用系统的市场具有深远的、颠覆性的影响，而我们身为开发者也同样面临着挑战和机遇。不但如此，就我们的专业技术领域来说，Ajax在其他诸多方面也都是颠覆性的。下一节将对此进行讨论。

1.1.2　重新定义 Web 应用的架构

Web应用的架构一直是一个令人感兴趣的领域。面对Web的“工作-等待”的特质，维持适当的用户工作流程是一个不小的挑战，因此对于Web应用在服务器上的组织方式，一直有着持续不断的革新。尽管如此，也已经形成一些惯例，比如要对表现层和业务层进行职责划分，也要建立一个可持久化的领域模型。图1-3是这种设计的示意图，以及Ajax对它的影响。

① 即前面提到的Dynamic HTML，动态HTML。——译者注

② 原文是“line-of-business apps”，常译作“行业应用”或“企业应用”。 line-of-business即业务线，指服务于特定业务需求的一组高度相关的产物。业务线经常直接对应于企业在战略层面的业务单位。常见的业务线应用如供应链管理、呼叫中心、会计结算等。传统上，这类应用通常是桌面应用或基于胖客户端的C/S应用。——译者注

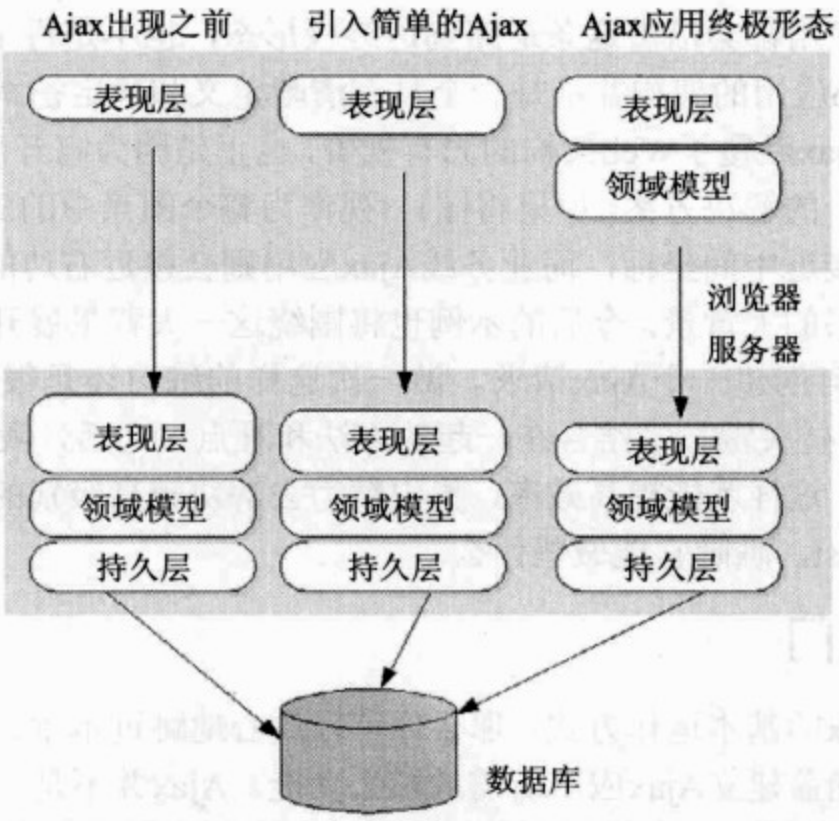


图1-3 Web应用的多层架构，以及Ajax对这种设计的影响

图左面一栏是Ajax出现之前的架构，所有的操作都发生在服务器上，浏览器只是一个哑终端，仅仅接受预先处理好的HTML内容。

中间一栏展示的是向应用中引入一些相对简单的Ajax之后所取得的效果。虽然说工作流程的所有方面仍旧由服务器进行控制，但是现在可由超链接和表单请求HTML内容片段，只对屏幕上的部分内容进行更新，而不是进行全屏刷新。服务器响应会由JavaScript代码处理，代码会读取响应并据此对DOM进行重组。我们可以看到浏览器的表现层开始变厚，因为我们添加了JavaScript来安排服务器返回的内容。通常我们也会看到，服务器的表现层开始变薄，因为我们生成的响应更简单、更专注，不再是每次都组装完整的页面。

一个被良好分解的（well-factored）传统Web应用也会以模块化的方式生成响应，所以引入一些Ajax的特性并不需要重写整个应用。当然，长久以来的服务器请求和响应的模式可能要进行改造，并且我们已在浏览器上引入一个用JavaScript编写的新的表现层。沿着这样一条Ajax之路循序渐进，我们就不会让开发团队陷入技术断层，但是我们会让他们重新思考，并逐步采用一些新的技术。

当我们更深入地运用Ajax，或是向Ajax所开启的新一代的业务线Web应用迈进时，就会有更多架构层次发生改变。在图1-3的右面一栏，我们描绘了基于Ajax的应用的终极形态，其浏览器中的JavaScript代码将会十分复杂以至于其本身也可进行分层。在这种情况下，客户端表现层掌控着用户的工作流程。同时，客户端代码还维持着主要领域实体的一个局部模型，JavaScript表现层会与它们进行通信，而不是直接访问服务器。

在服务器端，我们会发现表现层精简了许多。它主要的职责将是在领域模型之上提供一个粗粒度的（coarser-grained）门面（façade）。门面定义了应用的主要用例，而通过HTTP接口传送数据所需的数据编组（marshal）和解组（unmarshal）过程也是由它控制的。至于流程的控制和内容的视觉表现，则大多已经转移到了客户端JavaScript层。

并不是每一个Ajax应用都会沿着这条道路到达终极形态，也不是每个应用都适合这样发展。我们一开始就说过，Web应用的架构并不是一个具有清晰定义和确定答案的问题，仍然存在很大的创新空间。之所以说Ajax颠覆了Web架构的固有视野，也正是因为它开启了一个新的创新方向，而不仅仅是提供一个单一的解决方案。如果将图1-3视作为整个图景中的三个坐标，那么经过Ajax增强的老系统就会比较接近中间坐标，而业务线Ajax应用则会接近右边的坐标。

我们已经设定了本书的大背景，今后的示例也将围绕这一大背景展开。现在让我们进入第一堂编码训练课，看看如何创建一个Ajax请求。做一次这样的练习还是很有用的，它标示出了将JavaScript和HTTP这样一些关键技术结合在一起的方法和难点。之后，我们会将较低层级的功用包装起来放到程序库中，这样就能提高效率，集中精力去解决较高级别的问题。当然，首先让我们看一下XMLHttpRequest，瞧瞧它能做些什么。

1.2　Ajax快速入门

如果你已经了解Ajax的基本运作方式，那么你可以放心地跳过本章。如果你不太熟悉或想稍作复习，下面的内容会涵盖建立Ajax应用所需的核心功能。Ajax并不是一个专门的产品，浏览器中也并没有专门的Ajax函数集。正如你将看到的那样，Ajax无非就是将名为XMLHttpRequest的特殊JavaScript对象与JavaScript事件和动态HTML（即DHTML，也叫做DOM操作，DOM manipulation）技术结合起来使用。在本节中，我们会把XMLHttpRequest对象拉出来遛一遛，并晒一晒它的基本特性。

1.2.1　XMLHttpRequest简介

在编写传统的Web应用时，我们使用HTTP协议在浏览器和服务器间进行通信。用户交互的主要方式是超链接和HTML表单，两者都会在浏览器中引发HTTP请求。两者的功能也都只限于将响应内容自动装入当前页面或页面的一个窗格（frame）中。确切地说，它们的设计目标就是通过Web取回文档内容。

当我们开始处理更加复杂的客户端应用时，我们可能需要取回数据而非文档内容，或者可能需要取回更细粒度（finer-grained）的内容以插入到当前页面中。XMLHttpRequest对象（以下简写为XHR）的出现正是为了解决这一问题，它让我们能更好地通过程序来控制HTTP请求。

我们在上一节中论述过，XHR对象允许我们以编程方式向服务器发送请求和接收响应，而不是让浏览器将响应自动呈现为一个新的页面。从客户端代码的角度来看，为了达成这一目标我们需要完成几个步骤，如图1-4所示。

首先，我们需要创建一个XHR对象❶。然后，我们向它提供发送请求所需的信息❷。最后，我们对返回的响应进行处理❸。在发送请求和接收响应之间，服务器也有一些工作要做，当然，为此我们也要写一些代码，如PHP、Java，或.NET下的某种语言，或是由当前环境所决定的某种语言。在此我们主要关心客户端代码，不过在服务器端处理一个简单Ajax请求所涉及的工作机理与Ajax出现之前的Web编程并没有太大差异。稍后，在一些更复杂的示例中我们会给出服务器端代码。但现在，我们只需说明客户端是如何工作的。在完成这些步骤时，我们还会不时回顾图1-4。

我们要做的第一步就是拿到一个XHR对象。

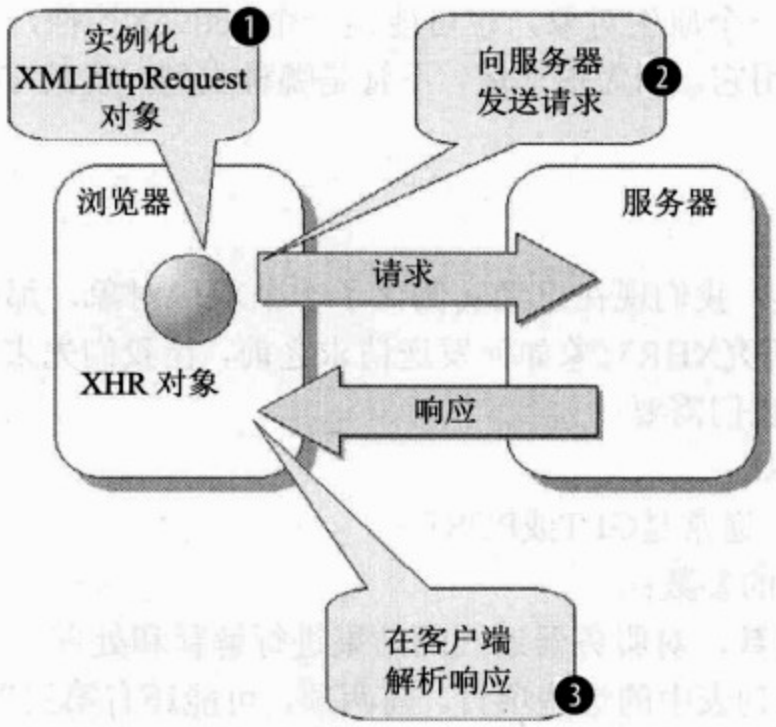


图1-4 使用XHR对象创建一个Ajax请求的关键步骤

1.2.2 实例化 XMLHttpRequest

四个主要的现代浏览器家族，即Internet Explorer、Firefox/Mozilla/Netscape、Safari以及Opera，都内建了XHR对象。要使用它，你可以创建XHR对象的一个实例，设置参数以建立你想要发送的请求，通知它发送请求，然后处理结果。代码清单1-1所列出的是第一个步骤的跨浏览器的代码示例，即实例化一个XHR对象。

代码清单1-1 实例化一个XHR对象

```
var xhr;
if (document.XMLHttpRequest) {                    检查XHR对象
   xhr = new XMLHttpRequest();                    创建本机对象
} else {
  xhr = new ActiveXObject("Microsoft.XMLHTTP");   创建ActiveX控件
} else {
  alert("cannot use Ajax");
}
```

这段代码有点复杂，原因是浏览器之间不兼容（对于Web开发老手来说，这是意料之中的事）。Internet Explorer（直到第7版之前）并没有原生（native）的XHR对象。相反，它将XHR实现为一个ActiveX对象。因此，如果用户关闭了“安全ActiveX”的脚本编程许可[①]，就无法运行支持Ajax的应用。（在就此向微软发难之前，请记住正是他们在90年代末期发明了XHR对象，并将其作为XML解析模块加入到IE中。其他主要浏览器直到最近才加入了对XHR的支持。）

我们也需要检测出那些不支持任何一种XHR对象的旧浏览器，并发布某种消息，声明应用无法在这个浏览器上运行。通过`if()`语句我们会沿着某一个代码分支执行直到最终获得一个XHR

① IE的Internet Options→Security Settings→ActiveX controls and plug-ins中的Script ActiveX controls marked safe for scripting。——译者注

对象的引用。它可能是一个原生对象，也可能是一个ActiveX控件，不过只要我们有了某种类型的XHR，就可以开始使用它。因为很幸运，不管是哪种类型，就其方法和属性来说几乎是完全一致的。

1.2.3　发送请求

回过来看一下图1-4。我们现在已经实例化了一个XHR对象，那么接下来我们就要迈出第二步：发送请求。在开始研究XHR对象如何发送请求之前，让我们先来看看建立一个对服务器的调用所需要的基本信息。我们需要：

- 服务器资源的URL；
- HTTP请求类型，通常是GET或POST；
- 服务器资源所需的参数；
- 一个JavaScript函数，对服务器返回的结果进行解释和处理。

OK，让我们过一下列表中的这些项目。前两项，可能还有第三项，要在调用open()方法的时候作为参数传入。该方法会初始化一个指向URL的连接。这个方法有三种重载形式：

```
open(http_method, url)
open(http_method, url, asynchronous)
open(http_method, url, asynchronous, userid, password)
```

HTTP方法几乎总是GET或POST，但也可以是其他有效的HTTP方法，比如PUT、DELETE、HEAD等，只要服务器支持这些方法。如果asynchronous为真（true），那么请求会在后台运行，这允许用户在XHR请求处理的过程中执行其他工作。如果为假（false），那么请求将会是同步的，用户的操作会被阻塞直到请求结束，就像传统的“工作-等待”模式的Web应用那样。

第三个参数反映了XHR作为通用ActiveX控件所继承的遗产。在某些应用中可能有创建同步请求的需要，但是JavaScript解释器本质上是单线程的，同步请求会阻塞浏览器上所有的用户交互，直到响应返回。所以在Ajax应用中，你应该始终采用异步请求。

userid和password用于连接需要用户名和密码的服务器。它只适用于HTTP验证（例如不能用于NT域验证），密码会以纯文本方式发送出去，因此应慎重使用，除非是运转在基于HTTPS的安全socket之上。

所以，要打开一个URL连接，我们可以这样写：

```
xhr.open('GET', 'servlets/ajax/getItem?id=321', true);
```

因为使用的是HTTP的GET方法，所以我们把参数以查询字符串的形式放在URL中传递给服务器。如果使用POST，我们就要在请求的主体（body）里传递它们，过一会儿我们就会看到了。

启用XHR对象的第二步是指派一个回调处理器函数来接收响应。眼下我们不用担心函数要做些什么，直接指派它就可以了。例如：

```
xhr.onreadystatechange = parseResponse;
```

注意，我们传递的是一个函数对象的引用。我们此刻并没有调用这个函数——因为函数后面并没有圆括号——而是告知XHR对象在响应返回时调用这个函数。指派回调函数同设置UI事件处理器——如DOM元素上的onclick和onmouseover——是完全相同的。

第三步就是调用send()方法。send()执行服务器调用，并能用于发送那些不是由URL指定的附加数据。send()需要的唯一参数就是要在请求主体中发送的附加数据。通常，只有POST请求带有主体，所以对于GET请求，我们只需传入一个空字符串：

```
xhr.send('');
```

这样就搞定了！现在请求已经送往服务器，客户端代码可以清闲一阵，直到响应返回，而这是我们下一节要解决的问题。

1.2.4 处理响应

我们已经向服务器发起了请求，并且我们可以暂时假设服务器会做好它的工作并返回响应给我们。回顾图1-4，我们接下去要做的就是在响应可用时接收和拆解它。在上一节中，我们就已经为这个时刻做了准备——给XHR对象指派了一个回调处理器函数。而在本节中，我们会看到回调时的情形。

你或许以为XHR只是在响应到达后发出通知，这样想也情有可原，不过实际上它在响应的生命周期的数个阶段都会发出通知。偶尔这些信息也会非常有用，但通常来说，它只会使你分心。

我们在上一节中所指派的回调处理器的名称为onreadystatechange。在XHR对象经历的每种就绪状态（ready state）上，回调函数都至少会被调用一次。在onreadystatechange函数中，需要手动检查readyState属性，以此确定当前请求处于生命周期的哪个阶段，以及是否可以开始对最终结果进行处理。任何时候readyState总会是下列预定义值之一：

值	状态	描述
0	Uninitialized（未初始化）	尚未调用open()
1	Loading（正在加载）	已经执行open()
2	Loaded（已加载）	已经执行send()
3	Interactive（正在交互）	服务器返回了一个数据块
4	Complete（完成）	请求完成，服务器数据发送完毕

基本上你只需检测readyState == 4，也就是查看请求是否完成。一个典型的回调函数如下：

```
xhr.onreadystatechange = function(){
  var ready = xhr.readyState;
  if (ready == 4){
    parseCompletedResponse(xhr);
  }
};
```

在该函数中，我们检查readyState属性的值是否为4，如果是4，就表明响应已经完整到达并可以解析了，我们会将它转给一个解析函数。而在解析响应时，我们首先要知道的就是请求是否成功。

status属性包含了请求的HTTP状态。一个有效的HTTP GET或POST在所请求的URL得到正确处理后通常会返回200。当URL不存在时，则会返回404。一般说来，任何200到299之间结果代码都表示成功，而其他的代码则表示失败或指示浏览器的进一步动作。通过结合readyState和status，你可以检测出请求是否成功完成。所以可以如下修改我们的函数：

```
xhr.onreadystatechange = function(){
  var ready = xhr.readyState;
  if (ready == 4) {
    var status = xhr.status;
    if (status >= 200 && status < 300) {
      parseCompletedResponse(xhr);
    } else {
      parseErroredResponse(xhr);
    }
  }
};
```

假设我们的响应已经成功返回。下面就可通过两个属性获取响应：responseText和responseXML。responseText会把响应呈现为一个普通字符串，而responseXML会把响应呈现为一个经过解析的XML文档。我们会在本章和下一章中通过示例更详细地了解它们。

至此，我们已经将一个普通的Ajax请求和响应的生命周期完整地过了一遍。这一路上，我们干了许多埋管布线的活儿（plumbing）——取得对象、装配请求，还有解析响应。好消息是，在演示过一遍这些底层细节之后，我们不会在本书中再次看到这些细节了，因为已经有许多好的框架和程序库能够代我们完成这些令人生厌的工作。

不过，了解XHR的工作原理还是很有用的。因为它可以帮助我们理解Ajax程序库中的XHR包装器能做什么和不能做什么。最后，在转向更高级的方式之前，我们将先介绍XHR对象上的其他方法和属性。

1.2.5　XMLHttpRequest的其他方法和属性

XHR还有一些不太常用的方法。一般情况下，你不需要了解它们，但它们对于一些特定的任务来说很有用。

1. abort()

abort()用于在可行情况下中止当前请求。这只是在客户端中止——只要调用了send()方法，服务器就会收到HTTP请求并处理结果。不过浏览器会忽略结果并停止处理。

2. setRequestHeader(header, value)

该函数设置HTTP请求的首部（header）。它经常被用来设置请求主体的内容类型（content type）。也可以使用任何有效的HTTP首部和值。你可能为许多不同的目的使用这个函数。例如，把请求的MIME类型设为x-www-form-urlencoded，就可以模仿一个HTML表单的POST过程：

```
xhr.setRequestHeader(
  'Content-type',
  'application/x-www-form-urlencoded'
);
```

通过Ajax，我们不仅可以向服务器POST键值对，也可以发送XML数据，此时，我们需要告诉服务器，我们在发送XML：

```
xhr.setRequestHeader(
  'Content-type',
  'application/xml; charset=UTF-8'
);
xhr.send("<data source='ajax in practice'>hello world</data>");
```

在第2章中我们会更加详细地讨论如何在Ajax中使用XML。与设置请求首部类似，也有一些方法可供我们在处理响应时使用。

3. getResponseHeader(header)/getAllResponseHeaders()

一个HTTP请求通常包含许多首部，每个首部都是一个键值对。XHR对象可用getAllResponseHeaders()列出所有首部名，也可以用getResponseHeader()读取特定首部的值，该函数所需参数就是首部的名称。例如，要检测服务器是否是微软的IIS服务器，可以这样：

```
if (xhr.getResponseHeader("Server")
  .indexOf("Microsoft-IIS") != -1 ) {
    alert("The server is a Microsoft IIS server.");
}
```

现在我们已经介绍完了XHR对象的详细用法，可以转而去关注一些更有趣的问题。在本章以后的内容中，我们会介绍Prototype库中为简化Ajax所设计的助手（helper）对象，并通过一系列的示例来探索怎样使用我们新得到的力量来向服务器发起异步请求。

1.3 使用框架简化 Ajax

在上一节中，我们了解了使用XHR对象创建Ajax请求的基础知识。现在我们已经有能力直接从服务器读取数据，而无需对整个页面进行刷新，但我们发现自己并不了解要怎样与服务器沟通，服务器又能提供何种类型的响应。

HTTP协议定下的基本规则相当宽松。任何一次通信都起始于客户端的请求，完成于服务器的响应，并且通信的这两个部分都必须基于文本。除此之外，差不多怎样都行。在本章我们就会看一下通过Ajax传递的数据所能采用的各种不同形式，并开始考虑怎样使用Ajax来构建应用中的B/S（浏览器/服务器）通信。

其次，我们还会在本章中介绍一些程序库和框架代码，它们能让开发者轻松许多。要直接使用原始的（raw）XHR对象，我们得编写大量的代码去搞定许多谜一般的跨浏览器问题，还要手工编织HTTP请求和响应的所有底层细节。要学习HTTP的工作原理，这或许有所助益，但是在编写产品级代码时，我们通常会全神贯注于有关应用状态和应用逻辑的更高层次问题，而不想去操心底层管线设施（plumbing）的运转细节。

幸运的是，已经有不少出色的Ajax框架和程序库可为我所用了。我们会在第4章中更正式地展示这些框架，现在我们只是在行文中捎带着介绍一下。

在本章的示例中，我们主要关注数据在客户端和服务器之间的传递方式。因此我们在本章中会一直使用“Hello World”型的示例。示例的后端使用JavaServer Pages（JSP），并已在Tomcat Web服务器上进行了测试。我们在本书的代码下载中为所有示例提供了一个.war文件[①]，它为各个示例提供了一个起始页面，如图1-5所示。

闲言少叙，让我们进入Ajax通信的第一个示例吧。

尽管我们在上一节中看到创建Ajax请求相当简单明了，但还是要做不少簿记工作

① 本书网站上的代码下载并没有提供war包，还是需要手动部署的。

（bookkeeping）。当向客户端交付一个真实的Web应用时，服务器的异步调用只是一种达到目的的手段。每次与领域模型沟通时还得操心一堆readyState、HTTP首部以及经过URL编码的查询字符串，实在太烦人，也让人分心。

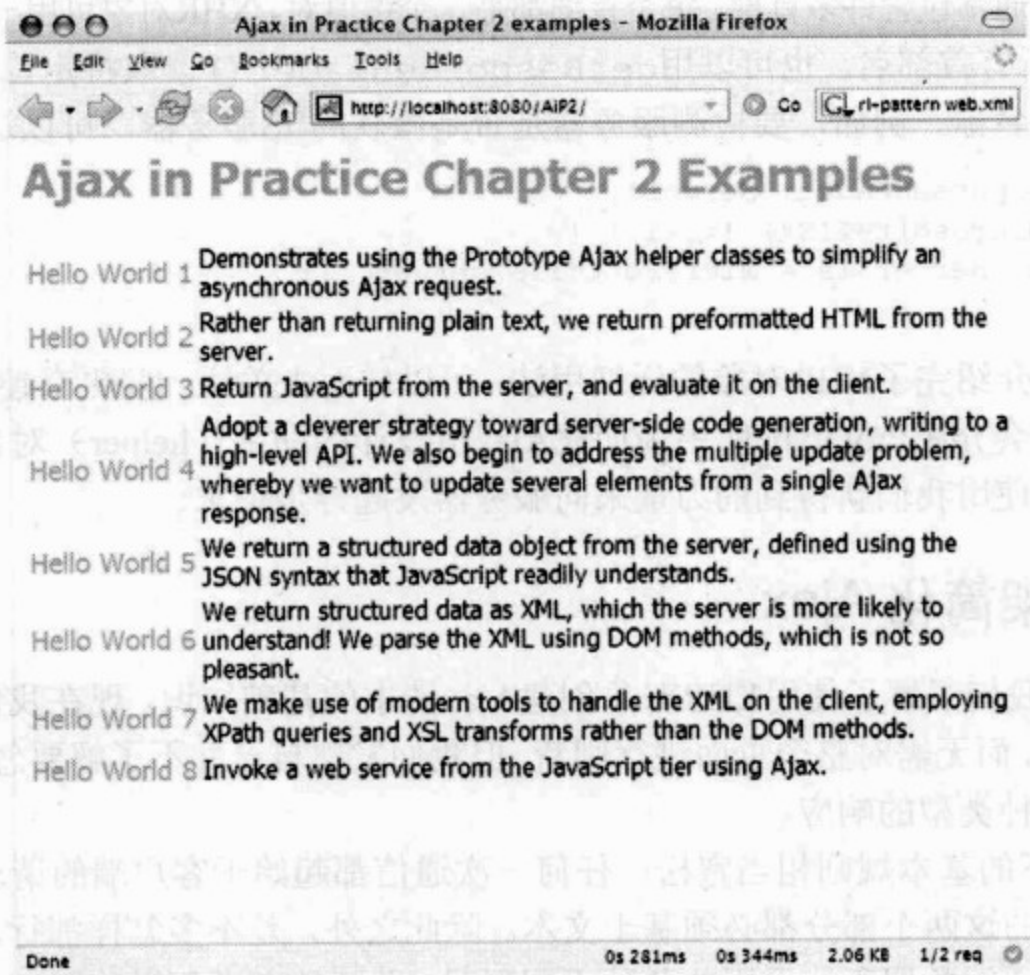


图1-5　第1章和第2章中的示例的起始页面

在软件开发过程中，当我们理解了一个特定任务所包括的众多问题，就应该把我们的解决方案封装入一个助手对象或一组函数中，这样我们就可以把注意力集中到更高层次的事务上。我们希望能够用尽可能简洁的代码来建立常规的Ajax请求并处理响应，但在需要的时候仍然有能力调整细节选项（fine-tuning）。

我们可以自己写一套助手程序库去封装XHR对象，或者也可以直接使用一个第三方库，这样就不用亲自做那些体力活了。在本节中，我们会看一下Prototype库中的Ajax助手类，瞧瞧它们是怎样帮我们简化Ajax的。我们就从创建请求开始吧。

1.3.1　用 Prototype 库中的 Ajax.Request 对象创建请求

当我们按1.2节的示例向服务器发送一个Ajax请求时，要面临一堆琐碎的问题。首先，我们得以一种跨浏览器兼容的方式创建XHR对象。其次，我们得调用XHR对象上的多个方法，向它提供URL、HTTP方法、用于POST的主体，还要设置其他的HTTP首部。要搞定这些，就需要掌握足够的关于HTTP协议的知识。了解底层原理当然总是件好事，但是不应该强迫我们在每次访问服务器时都去考虑这些。

一个良好的包装对象，比如Prototype库中的Ajax.Request，能自动处理跨浏览器的问题。它

也允许我们只传入那些必需的信息，其他没有显式设定的参数会被自动补上合适的默认值。

1. 问题

为了使用XHR对象，我们需要关注许多低层次的细节，例如以跨浏览器的方式获得对象，以及在响应到达期间对readyState的变化作出反应。

2. 方案

使用框架来简化Ajax请求的创建过程。让我们从一个很简单的应用程序开始，图1-6显示了它的用户界面。

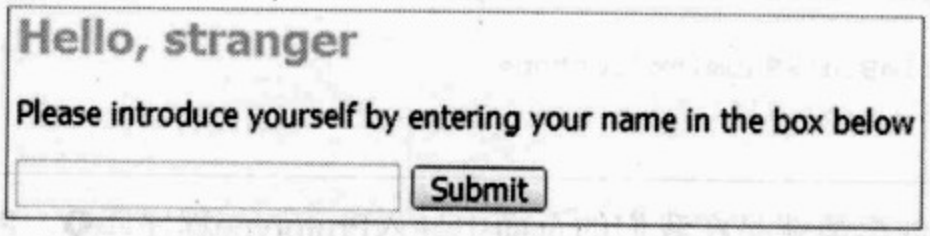


图1-6 Hello World应用程序第一版的用户界面

我们提供了一个文本输入框和一个提交按钮。当点击提交按钮后，文本输入框中的文字就会发送到服务器，然后短句“Hello, stranger”会换成服务器所返回的名字。在这个示例里，服务器并没有对名字做任何处理——直接原样返回——不过这里我们只关心怎样在客户端处理这个响应。服务器其实可以做任何处理。我们首先来看一看客户端代码。

3. 客户端代码

代码清单1-2列出了本示例的客户端代码。

代码清单1-2 hello1.html

```
<html>
<head>
<title>Hello Ajax version 1</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; font-size: 1.5em; }
</style>
<script type='text/javascript'
  src='prototype.js'> </script>
<script type='text/javascript'>
window.onload=function() {
  document.getElementById('helloBtn')
  .onclick = function(){
    var name = document.getElementById('helloTxt')
      .value;
    new Ajax.Request(
      "hello1.jsp?name = "+encodeURI(name),
      {
        method:"get",
        onComplete:function(xhr){
          document.getElementById('helloTitle')
          .innerHTML = xhr.responseText;
        }
```

❶ 包含Prototype库

❷ 创建Ajax Request对象

❸ 提供URL

❹ 提供可选择的参数

```
        }
      );
    };
};
</script>
</head>
<body>
<h1 id='helloTitle'>Hello, stranger</h1>
<p>Please introduce yourself by entering
your name in the box below</p>
<input type='text' size='24' id='helloTxt'>
</input> 
<button id='helloBtn'>Submit</button>
</body>
</html>
```

我们要做的第一件事情就是在我们的页面中导入Prototype程序库❶。Prototype库只有一个文件，所以我们也只要写一行代码。

在window.onload事件处理器中，我们引用了按钮对象，并为它指派了一个单击事件的处理器。在事件处理器中，我们从文本输入框中读取名字❷，然后创建了一个Ajax.Request对象。这个对象需要两个参数。第一个参数是服务器资源的URL❸。第二个参数是一个自由形态（free-form）的JavaScript对象，它包含了一组任意的额外配置选项❹。这些选项可以包括HTTP动词[①]、首部、请求主体、多种回调选项以及其他一些特性。这里，我们只传递了两个选项。其一是将HTTP方法设为GET，因为Ajax.Request默认设为POST；其二是提供了一个回调函数。

回顾1.2节，每当readyState改变时XHR对象上的回调函数就会被触发。当Ajax.Request在内部创建一个XHR对象时，它定义了一个内部回调来处理这些细节。而作为程序库的最终用户，我们就可以设定更高层次的回调，比如onComplete，它更符合我们目前的需求。在一些相对特殊的情况下，我们希望在readyState发生其他变化时也能获得通知，对此我们可以额外设定其他的回调函数以捕获这些变化。

我们的回调函数极其简单，就是修改一下页面中的标题文字。如图1-7所示，用户输入了自己的名字，点击了提交按钮，随后标题就被修改了。

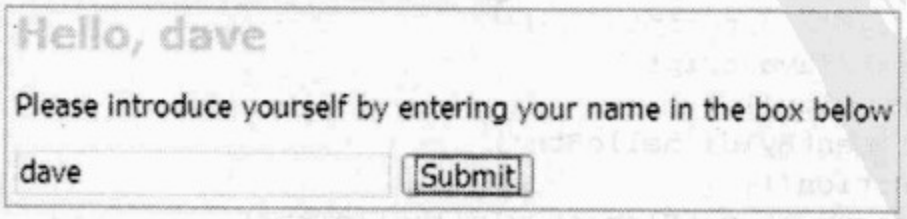


图1-7　Hello World第一版对Ajax响应进行处理之后

以上是本示例所需的客户端代码，下面我们会看一看服务器端代码。

4．服务器端代码

这个示例的服务器端实现非常简单，代码清单1-3列出了这些代码。

① 原文“HTTP verbs”，HTTP动词是指HTTP方法（HTTP methods）。——译者注

代码清单1-3 hello1.jsp

```
<jsp:directive.page contentType="text/plain"/>
<%
String name = request.getParameter("name");
%>
Hello, <%=name%>
```

像我们之前提过的，这个示例的服务器端代码会尽可能地简单，因为我们希望把注意力放在客户端代码上。你可以想象这个简单JSP背后运行着一个复杂的多层应用——但原理还是一样的。

5. 讨论

Prototype库所提供的Ajax.Request类让我们能轻松创建Ajax请求。我们所要提供的就是一个URL、HTTP方法和一个回调函数。在内部，Ajax.Request解决了如何创建XHR对象的问题，并会在调整请求特性时补齐相关参数。它也在相当程度上简化了回调函数的语义，允许我们指定一个函数，只在响应完成后才会被调用一次，这样在这个函数中就只需要处理应用逻辑。

除了Ajax助手类之外，Prototype库还提供了许多其他特性。在本示例中，我们有意避免使用这些特性，因为我们只是把Ajax.Request作为一个设计良好的包装对象的示例。其他流行的Ajax库中也有类似的包装器，也同样采用简单直接的调用语义。仅举三例：Dojo有dojo.io.Request类；MochiKit有MochiKit.Async，JQuery则有`$.ajax()`函数。取决于你所选用的程序库，Ajax包装器的实际功能会有所不同，但应该都会让你感到满意，因为每一个都会帮你省下不少时间。

在本例中，我们用于处理响应的回调函数非常简单。在下一个示例中，我们会看到怎样才能基于Ajax产生更加丰富的内容。

1.3.2 简化 Ajax 响应

在上一个示例中，我们用innerHTML属性把服务器内容直接写入到页面的DOM元素中。服务器传给我们的响应是一段简单的文本。然而，在许多情况下，我们希望服务器能传送给我们更加复杂的信息，这些信息经常会有自己的内部结构。在第2章中，我们会研究在服务器和客户端之间传递结构化信息的各种方法。

结构化数据是绝大多数Web应用的核心。在服务器端，数据通常存储在关系数据库中。在客户端，数据则显示为某种报表或用户界面。在数据库和用户界面之间，数据可能要按照应用逻辑进行处理和运算。传统的做法是全部在服务器上完成，客户端只是一个哑终端。而有了Ajax，我们既能够在客户端上运行应用逻辑，也能够在服务器上运行逻辑，甚至能在两端都运行。这里有许多转变亟待探索。

我们将从哑终端的方法开始。服务器从应用数据中产生一个丰富的报表，并发给客户端显示。客户端不需要知道这些信息的含义，只需要知道如何显示它们。我们会让本示例的响应稍微活泼一点，在标题下面多返回一小块内容（就本例来说完全微不足道！）。图1-8显示的是当示例把整个响应插入页面后的效果。

1. 问题

服务器发送给我们一个关于应用状态的丰富的可视化报表（而不是一段简单文本），这些信息需要加入到用户界面中。

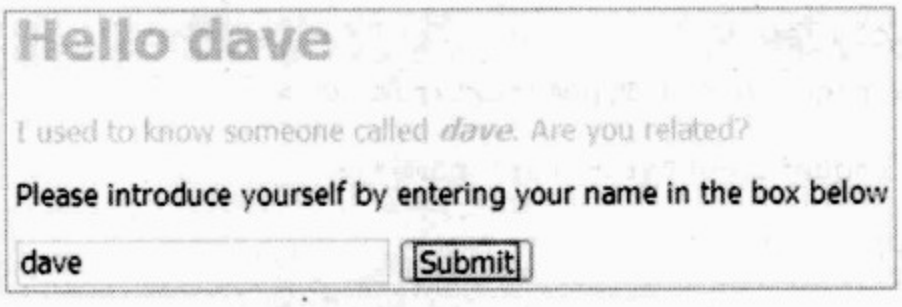


图1-8　在Ajax响应中返回丰富HTML的结果

2. 方案

这个问题的解决方案相当直接，并不需要编写多少JavaScript代码。客户端代码的改动非常小，实际上，我们仍可以用innerHTML属性把响应粘贴到文档中。代码清单1-4列出了示例2的客户端代码。

代码清单1-4　hello2.html

```
<html>
<head>
<title>Hello Ajax version 2</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
</style>
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript'>
window.onload=function(){
  $('helloBtn').onclick = function(){
    var name = $('helloTxt').value;
    new Ajax.Request(
      "hello2.jsp?name = "+encodeURI(name),
      {
            method:"get",
            onComplete:function(xhr){
              $('helloTitle').innerHTML = xhr.responseText;
            }
          }
        );
      };
    };
    </script>
    </head>
    <body>
    <div id='helloTitle'>
    <h1>Hello, stranger</h1>
    </div>
    <p>Please introduce yourself by entering your name in the box below</p>
    <input type='text' size='24' id='helloTxt'></input>
     
    <button id='helloBtn'>Submit</button
    </body>
    </html>
```

要做的主要改变在服务器上。代码清单1-5列出了修改后的JSP代码。

代码清单1-5 hello2.jsp

```
<jsp:directive.page contentType="text/html"/>
<%
String name=request.getParameter("name");
%>
<h1>Hello <%=name%></h1>
<p>I used to know someone called <b><i><%=name%></i></b>. Are you
related?</p>
```

为规范起见，我们将响应的MIME类型设置成了text/html。严格上来讲，这并不是必需的，只是对用途做一个声明。之后你会看到，有些Ajax库会比较在意响应的MIME类型，所以最好养成这个习惯。

响应的主体根据输入的参数生成一小块内容。在本例中，我们就只是把用户所提供的名字复述一遍，但这只是为了让我们的示例的后端保持简单。在正式产品中，我们就会有一些真正的代码——前端控制器、领域对象、数据库等——来获取请求的参数然后生成响应。但仅就Ajax的工作机理而言，效用是一样的，都是向客户端返回一个HTML片断。

3. 讨论

在服务器上生成HTML片段是Ajax web应用开发中的可行策略，也可以满足所有的需要。它提供了一条从传统的Web应用和框架逐步迁移到Ajax的清晰路线，传统上也是由服务器生成HTML并发送给浏览器的。

我们使用innerHTML属性将新的HTML片段插入到现有页面中。在大多数情况下，这种方式是有效的，它完全清除一个元素的原有的内容，并替之以新的内容。如果我们想要更细致的控制，也可以用一些其他的DOM方法，比如insertAdjacentHTML()和createContextual-Fragment()，但它们无法跨浏览器使用。Prototype库提供了跨浏览器的包装器，将这些方法包装为Insertion对象，其他程序库也提供了类似特性。

在使用innerHTML时，你还需要搞清几个挺“搞”的问题[①]。首先，浏览器会忽略用innerHTML包含到页面中的<script>标签。其次，除了单元格（即<TD>标签），其他HTML表格元素的innerHTML属性都是只读的。

最后需要指出的是，Prototype库提供了一个Ajax.Request的专门子类，它会自动把响应设置到指定DOM元素的innerHTML属性，用它完成这类任务更轻松。如果你正在使用Prototype库，并且要从服务器发送HTML片段，那么我们强烈建议你去看一下Ajax.Updater，但是我们并不想在这里用它，因为它隐藏了太多我们想要研究的底层机理，而且如果我们的讨论聚焦在单独一个程序库上，就太局限了。

---

① 原文为gotcha(s)，是(I've) got you的连读，意思是“骗到你了”，比如合同中的陷阱条款就可以叫做gotcha(s)。在编程领域，gotcha(s)常指系统、程序或程序语言中违反直觉或者容易造成错误的特性。这些特性并非bug，只是很容易因误用而产生完全不想要和不合理的结果。——译者注

1.4　总结

本章伊始我们对Ajax当前的发展做了简要论述。这项技术正在快速成熟，同时也正在发生着一些有趣的变化。就整个趋势而言，我们注意到了两点。首先，对于Ajax的讨论，已经从如何创建一个Ajax请求转向了如何发挥Ajax的能力。人们的关注范围已扩展到Web应用架构、商业模式，还有Ajax所支持的新的可能性。

第二点值得注意的就是已经出现了一批成熟的Ajax框架和程序库。随着它们的兴起，徒手埋管布线的日子正令人欣喜地渐渐远去。并且，诸如Prototype、Dojo、Rico、DWR……这些程序库中都包含了大量的最佳实践。

从以上这两点展开，就形成了本书。我们将探讨围绕着Ajax开发涌现出来的更高层次的问题；我们也将向你展示，怎样通过合理运用各种工具包，使得我们在应对这些问题时更容易一些。这些内容结合到一起，给予你实践知识，让你能将Ajax成功运用到真实环境中。

不过我们也不能全然忽略管线设施及其底层细节。所以在本章第二节我们大致看了一下XHR对象的工作机理，然后我们以Prototype库的Ajax.Request为例，展示了如何通过包装对象来大大简化编程。在最后一个示例中，我们首次涉及应用设计的更高层次，提出了服务器以何种数据进行响应的问题。一种最简单的方法就是从服务器取回HTML片段并编织到现有页面中。

这一方法的确能满足需要，并且以相对简单的JavaScript就让我们的应用获得不少Ajax的优点。然而，对于复杂操作来说，那些关键的决策需要往返访问服务器，这样就不可能在客户端提供即时响应（responsiveness）。为此，我们需要把一些智能（intelligence）[①]移到客户端上去。在下一章中，我们会看一看如何使用JavaScript、JSON和XML来达到这个目标。

① intelligence，智能，简单来说就是指对问题作出决策和处理问题的能力，常用词汇如人工智能、商业智能等。此处基本上就是指应用逻辑。——译者注

第2章 Ajax的通信方式

本章内容

- Ajax的主要通信方式
- 通过JavaScript和JSON来使用Ajax
- 运用XML、XPath和Web服务

Ajax是一个熔炉，它把各种不同的应用设计方法和各式各样的通信方式融汇在一起。在第1章中，我们了解了Ajax核心要素XHR对象的工作机理，也看到了怎样把那些细节包装到助手对象。我们无须分神动手编写管线设施代码，可以专注于更加令人感兴趣的问题——如何建立服务器和客户端之间的通信。我们现在就可以看一看Ajax通信的不同类别，并教你如何使用这些形式各异的Ajax通信方式。

我们会继续使用在第1章中引入的Hello World示例和“问题/方案”格式。我们还会继续使用框架来帮我们处理Ajax底层的相关问题，让我们能去关注更加令人感兴趣的问题。许多示例会继续使用Prototype库的Ajax.Request对象，但我们也会看一下Sarissa库和一个Web服务（web services）客户端工具包。首先让我们看一看怎样以JavaScript作为通信媒介。

2.1 在服务器端生成 JavaScript

当服务器通过Ajax请求返回HTML时，我们可以在运行时生成复杂的用户界面，但它们在很大程度上还是静态的。与应用的任何一次重要交互都需要和服务器进行进一步的通信。在多数情况下，这没有什么问题，但有些时候，不仅要交付内容，还要把行为交付给客户端。客户端行为由JavaScript驱动，所以一种可行的方法就是由服务器生成JavaScript。

2.1.1 执行由服务器生成的代码

在处理服务器生成的HTML时，使用`innerHTML`就行了。而在处理服务器生成的JavaScript时，类似地，我们可以使用`eval()`方法。JavaScript是一种解释型语言，并且任何一个文本片段都可以拿来进行求值运算，也就是当作代码来执行。在下面这个示例中，我们会看到如何将`eval()`作为整个Ajax处理流水线上的一环。

在本章中，我们会继续我们的Hello World应用。在第一个示例中我们会再次根据响应来修改标题文字，如图2-1所示。

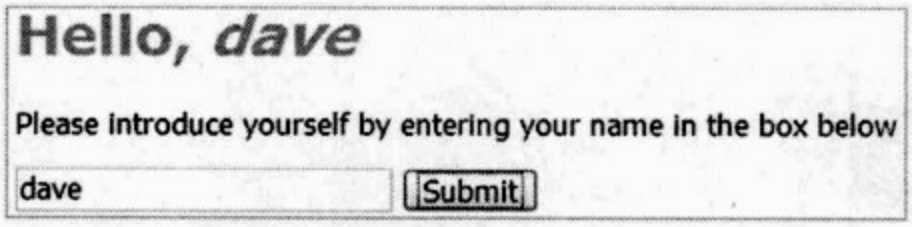


图2-1　执行服务器所生成代码后的结果

1. 问题

服务器通过Ajax请求返回JavaScript代码。当我们接收到代码时，就需要运行它。

2. 方案

使用eval()几乎和使用innerHTML一样简单。代码清单2-1展示了Hello World应用的第三个版本。

代码清单2-1　hello3.html

```
<html>
<head>
<title>Hello Ajax version 3</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
</style>
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript'>
window.onload=function(){
  $('helloBtn').onclick=function(){
    var name=$('helloTxt').value;
    new Ajax.Request(
      "hello3.jsp?name="+encodeURI(name),
      {
        method:"get",
        onComplete:function(xhr){
          eval(xhr.responseText);
        }
      }
    );
  };
};
</script>
</head>
<body>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
 in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
 
<button id='helloBtn'>Submit</button>
</body>
</html>
```

❶ 对响应进行eval

比较一下之前的方案和代码清单2-1，你会发现我们只需稍做修改。我们仍旧可以通过XHR对象的responseText属性来读取响应的主体，然后直接传递给eval()❶。

当响应到达的时候，我们会修改页面中的标题块。为此，我们需要执行DOM操作（DOM manipulation）。这些方法调用可以直接在服务器上生成，如代码清单2-2所示。

代码清单2-2 hello3.jsp

```
<jsp:directive.page contentType="text/plain"/>
<%
String name = request.getParameter("name");
%>
document.getElementById('helloTitle').innerHTML =
  "<h1>Hello, <b><i>"+name+"</i></b></h1>";
```

通常来说，设置MIME类型是个好习惯，但是在这里我们切换为text/plain，因为Prototype库太聪明了，会识别出text/javascript的MIME类型[①]并会自动帮我们执行代码。这里我们要手动执行以演示一般原理，所以就不去秀Prototype库的高级功能了！

3. 讨论

在这个示例中，我们演示了将JavaScript从服务器传送到客户端的基本原理，但是这一过程中也有一些问题。我们会在下一个示例中解决这些问题，但是，首先让我们对这些问题进行分析。

第一个问题是，客户端代码和服务器代码之间形成了非常紧密的耦合。JSP代码需要知道所装填的DOM元素的id属性。如果我们修改了用户界面上的HTML，那么我们也需要去修改服务器代码。在像这样的小型示例中，那并不算是很大的负担，但是随着应用规模的增长，它很快就会变成噩梦。

第二，虽然有了方案，用在这里却不得其所，甚至还不如直接使用innerHTML，这样会更加优雅和简单。如果响应只是用来更新页面上的单独一个元素，那么就没有必要使用这种方法。

在下个示例中我们会解决这两点问题，了解如何减少层次间的耦合，以及如何同时更新多个元素。

2.1.2 遵循代码生成的实践准则

当在服务器端生成JavaScript时，我们正在实践代码生成（code generation）。代码生成技术本身就是个令人感兴趣的话题，有一套自己的惯例和准则，其中一条基本准则就是总是尽可能针对最高的层面来生成代码。

在下一个示例中，我们会把Hello World应用弄得稍微复杂一点，演示如何更好地运用代码生成技术来应对这种情况。

1. 问题

在服务器上编写低层面的JavaScript会导致在服务器和客户端代码之间产生不可接受的紧耦合，这会令我们的应用难以扩展并愈加脆弱。

---

① 根据RFC4329，JavaScript的正式MIME类型是application/javascript或application/ecmascript，Prototype库自1.5.0 rc2开始支持正式的MIME类型。——译者注

同时，除了显示页面的当前访问者，我们也希望能列出之前的访问者。服务器还会按照访问者名字的长短进行分类，我们要为长名字和短名字分别准备一个列表（同样地，这是一个真实业务逻辑的替代品，因为我们希望本章的服务器代码能保持简单）。

我们还会在数据到达时弹出一个提示（alert）信息。图2-2展示了Hello World应用修改后的UI。

每当表单提交到服务器之后，最近一位访问者的名字就会和以前一样显示在标题文字中。我们还会在左边的矩形区域中保持一个不断更新的访问者列表。图2-3就是运行中的Hello World应用的第四版，可以看出已经有若干个有趣的访问者[①]来过了。

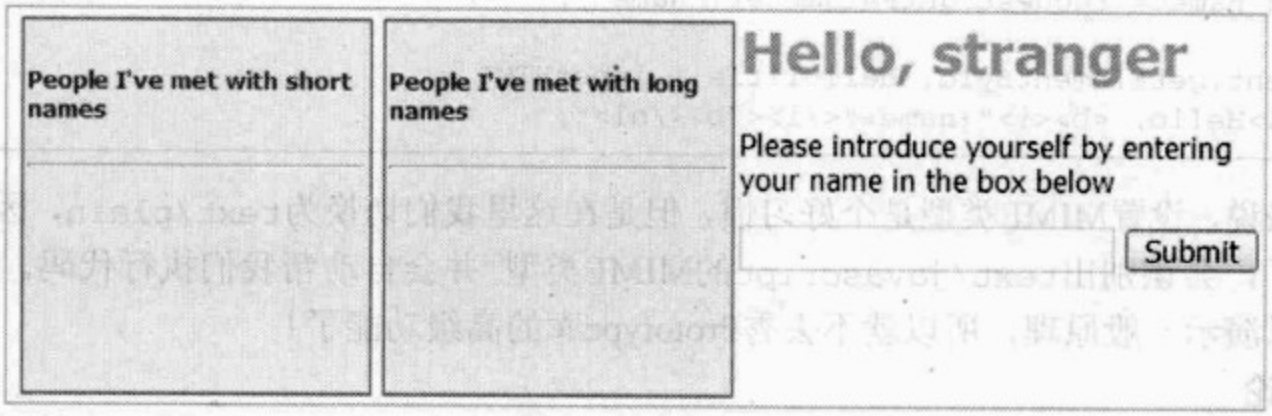


图2-2　Hello World第四版中扩展后的UI，在表单旁边是之前访问者的列表

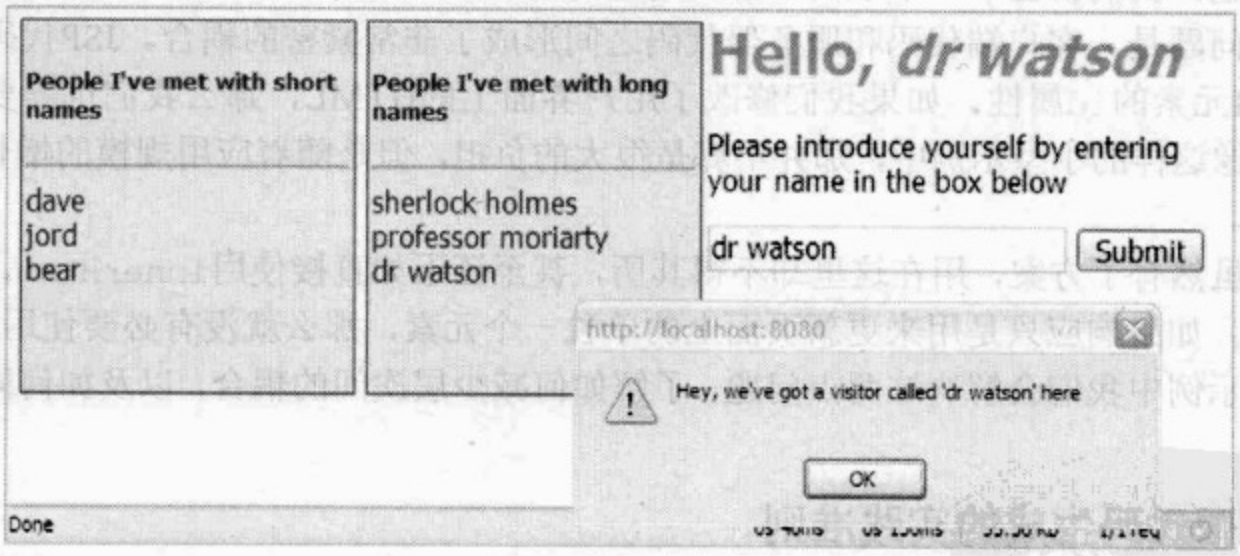


图2-3　Hello World第四版经过几次访问之后的样子。这里我们可以看到修改过的标题、左边更新过的列表以及一个提示信息。所有这些都源自于一个由服务器所生成的调用

2. 方案

当服务器返回响应时，我们要用新的信息来更新客户端。服务器发送给我们的代码只是数据的载体，所以我们会简化服务器生成的JavaScript，直接调用`updateName()`函数，并把数据作为参数传进去。

在客户端，我们需要写一段JavaScript代码来定义`updateName()`函数，如代码清单2-3所示。`updateName()`会负责处理所有的需求。

① 左边短名字这一栏是本书的三位主要作者，右边长名字的一栏则是福尔摩斯侦探小说中的人物：歇洛克·福尔摩斯、莫利亚蒂教授和华生医生。——译者注

代码清单2-3 hello4.html

```
<html>
<head>
<title>Hello Ajax version 4</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
        <script type='text/javascript' src='prototype.js'> </script>
        <script type='text/javascript'>
        window.onload=function(){
          $('helloBtn').onclick = function(){
            var name=$('helloTxt').value;
            new Ajax.Request(
              "hello4.jsp?name = "+encodeURI(name),
            {
              method:"get",
              onComplete:function(xhr){
                eval(xhr.responseText);      <-- 对响应进行求值
              }
            }
          );
        };
};

function updateName(name,isLong){      <-- 定义API
  $('helloTitle').innerHTML=
    "<h1>Hello, <b><i>"+name+"</i></b></h1>";
  var listDivId=(isLong)
    ? 'longNames' : 'shortNames';
  $(listDivId).innerHTML+=name+"<br/>";
  alert("Hey, we've got a visitor called '"
    +name+"' here");
}

</script>
</head>
<body>

<div id='shortNames' class='sidebar'>
<h5>People I've met with short names</h5><hr/>
</div>
<div id='longNames' class='sidebar'>
```

```
<h5>People I've met with long names</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
 in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
 
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

虽然需求变复杂了，但服务器代码却变简单了。代码清单2-4列出了为我们应用的第四个版本提供数据服务的JSP代码。

代码清单2-4　hello4.jsp

```
<jsp:directive.page contentType="text/plain"/>
<%
String name = request.getParameter("name");
boolean isLong = (name.length() > 8);
%>
updateName("<%= name %>",<%= isLong %>);
```

不仅这个JSP代码的长度缩短了，而且概念的数量也减少了。它只涉及应由服务器处理的业务逻辑，并通过高层面的API与客户端进行沟通。

3．讨论

通过本示例，我们跨越了一个重要的门槛。要通过一个单独的Ajax调用来刷新UI的多个区域，示例2所使用的简单的innerHTML方法就不再胜任了。在本例中，虽然这样的需求稍有斧凿之嫌，但是在许多真实应用中，确实存在同时进行多个更新的需求。例如：

- 在购物车应用中，添加一个新的商品后会在购物清单中增加一行，并更新总价，可能还要更新运费、发货日期等。
- 在更新数据表格中的一行后，可能需要更新总计项、分页信息，等等。
- 常见的双面板布局，在左侧的一览表中选择某一项之后会在右侧展开细目，两者是联动的。

我们通过定义JavaScript的API并生成调用该API的代码，解决了同时进行多个更新的问题。本例中只定义了一个API方法，并只调用了它一次，但是一个更复杂的应用可能要提供许多API，并生成很多行脚本。只要我们坚持在高层面的概念上进行通信，而不是陷入像DOM元素的ID和方法之类的细节，那么即使应用越来越复杂，这样的策略也始终可行。

除了在服务器上生成API调用，另一种选择是在服务器上生成原始数据，然后传给客户端来解析。这开辟了一个广阔的领域，本章后续部分就将对这一领域展开探索。

2.2 JSON 简介

回顾我们的Ajax探索之旅，在第1章中，所有处理都是在服务器上进行的，向浏览器发送的是预先转换好的HTML。在2.1节中，我们看到也可以用JavaScript作为一种在HTTP响应中承载数据的媒介。这里最关键的一点是，我们能够同时更新屏幕的多个部分了。同时，我们也能够让客户端和服务器端代码之间的耦合度维持在一个比较低的水平上。

朝着这个方向更进一步，我们可以分离各个层次的职责，服务器端只处理业务逻辑，客户端只处理应用的工作流程。这种设计类似于胖客户端架构，但是没有在客户PC机上安装维护客户端的麻烦。

在此类设计中，服务器会向客户端发送数据——可以是很复杂的结构化数据。正如本章伊始所说，对于数据采取什么形式，我们有大量的选择。目前来说，两个最有力的竞争者是JavaScript对象记法（JSON）和XML。下面我们就从JSON开始探索以数据为中心（data-centric）的Ajax。

JSON 一分钟快速入门

在深入示例之前，让我们快速过一下JSON。JSON是一个轻量级的数据交换格式，各种不同的服务器端技术以及JavaScript本身都可以很容易地生成和解析JSON。作为一个完整的数据交换格式，它可以在活对象[①]和用于交换的格式之间进行双向转译，如图2-4所示。

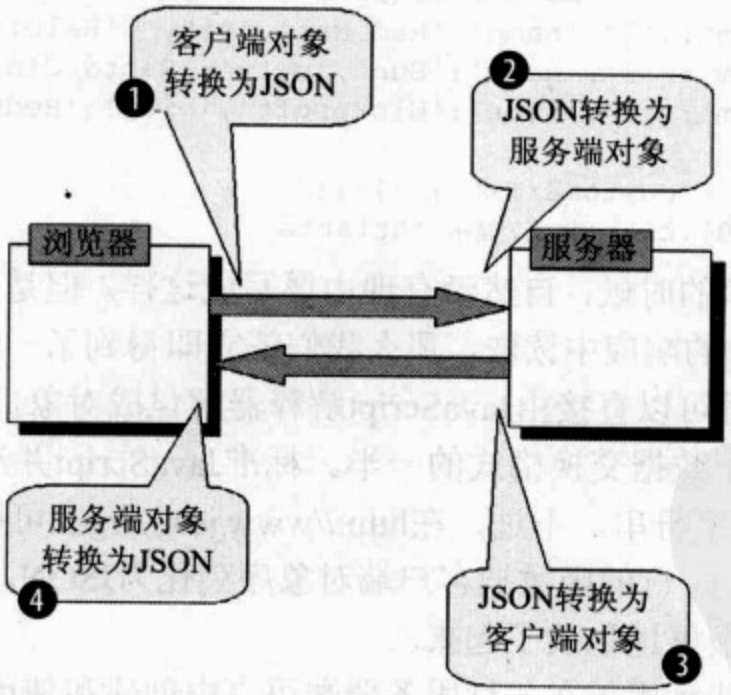


图2-4 JSON——一种双向往返的数据交换格式

JSON一半是由JavaScript语言规范免费提供，另一半则由第三方库提供。这听上去有点不太寻常，那么我们具体解释一下。

首先，让我们看一看JSON的定义是什么样子的。下面的示例定义了一个变量customers，

① 所谓活对象（live object），在这里大体指存在于内存中并能与程序中其他对象互通消息的对象，而对象在序列化或转换成数据交换格式之后，独立于程序而存在，也不可能接收其他对象发送的消息，自然就不是“活”的了。——译者注

其带有一个数组属性details。数组的每个元素是一个顾客（customer）对象的属性集合。每个顾客对象有三个属性：num、name和city。

```
var customers = { "details": [
                {"num": "1","name":"JBoss","city":"Atlanta"},
                {"num": "2","name":"Red Hat","city":"Raleigh"},
                {"num": "3","name":"Sun","city":"Santa Clara"},
                {"num": "4","name":"Microsoft","city":"Redmond"}
            ]
        } ;
```

我们用JSON语法定义了这个有点复杂的变量。与此同时，这也是一段标准的JavaScript代码。花括号用来定义JavaScript对象（类似于关联数组，associative array），方括号用来定义JavaScript数组。如果你想训练一下你的JavaScript语言基础技能，我们在第3章中进行了深入探讨。

变量定义好之后，我们就可以简单地通过标准JavaScript语法来读取它的值：

```
alert (customers.details[2].name);
```

这将会在提示框里显示字符串“Sun”。到目前为止，我们所做的一切只是使用一些标准的JavaScript语法，并称之为JSON。

我们也可以创建一个字符串变量，然后通过eval()执行它并生成对象变量：

```
var customerTxt = "{ 'details': [ "  +
                "{'num': '1','name':'JBoss','city':'Atlanta'}, " +
                " {'num': '2','name':'Red Hat','city':'Raleigh'}, " +
                " {'num': '3','name':'Sun','city':'Santa Clara'}, " +
                " {'num': '4','name':'Microsoft','city':'Redmond'}" +
              "] }" ;
var cust = eval ('(' + customerTxt + ')');
alert (cust.details[0].city); //显示 'Atlanta'
```

当我们直接声明字符串的时候，自然没有理由要写成这样。但是，如果是以另一种方式读取字符串，即从一个Ajax请求的响应中读取，那么我们就立即得到了一种极好的数据格式。它可以表达复杂的数据结构，并且可以直接由JavaScript解释器解包成对象。

就这点而言，我们有了数据交换格式的一半。标准JavaScript并没有提供任何方式来把一个JavaScript对象转换成JSON字符串。不过，在http://www.json.org上可以找到第三方程序库，允许我们使用一个叫做stringify()的函数把客户端对象序列化为JSON。JSON库还提供一个parse()方法，它把eval()的使用很好地包装了起来。

在json.org上，还可以找到用于在各种服务器端语言中创建和使用JSON的程序库。有了这些工具，就能够在Web应用的整个用户会话过程中以JSON为交换格式，在客户端和服务器之间来回传送结构化数据。

现在让我们回到Hello World示例，看一下客户端是如何处理JSON响应的。

2.2.1　在服务器上生成 JSON

对于JSON我们要花上不少篇幅，所以让我们把它分成两个部分。首先我们会看看如何能直接用浏览器内建的功能来解析JSON数据，并把上一个示例中的一般性的JavaScript响应换成一个JSON对象的定义。

1. 问题

我们希望服务器能以结构化数据的形式响应我们的请求，并由客户端来决定如何呈现这些数据。

2. 方案

继续Hello World的主题，现在它将返回更丰富的个人信息，而不仅仅只是名字。

- 访问者名字的首字母会在服务器上通过字符串操作运算出来。
- 访问者的喜好列表。
- 他们最喜欢的菜谱以关联数组进行编码。

图2-5展示了应用在接收到一个响应之后的情况。

同上一个示例一样，后端只是一个摆设，事实上，它对所有的名字都返回同样的数据（除了首字母以外）。从虚设的数据到一个真实的数据库只是一步之遥，但是我们不想因为引入太多特定于Java的后端特性而造成误解，因为客户端代码实际上可以与任何服务器端技术进行沟通。

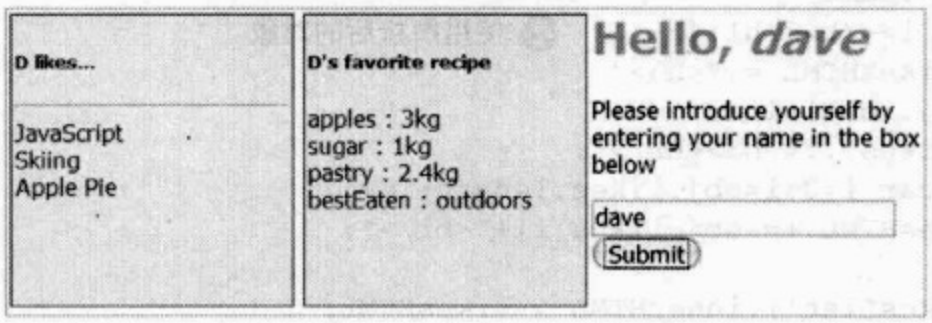


图2-5 使用JSON的Hello Wolrd应用所显示的丰富数据

好了，我们下面首先采用一种简单的方式，并且只利用JavaScript本身就具备的JSON解析能力。代码清单2-5列出了我们的客户端代码。

代码清单2-5 hello5.html

```
<html>
<head>
<title>Hello Ajax version 5</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript'>
window.onload=function(){
  $('helloBtn').onclick = function(){
```

```
var name=$('helloTxt').value;
new Ajax.Request(
  "hello5.jsp?name = "+encodeURI(name),
  {
    method:"get",
    onComplete:function(xhr){
      var responseObj = eval("("+xhr.responseText+")");
      update(responseObj);
    }
  }
        );
      };
    };

    function update(obj){
      $('helloTitle').innerHTML = "<h1>Hello, <b><i>"
        +obj.name
        +"</i></b></h1>";
      var likesHTML = "<h5>"
        +obj.initial
        +"likes...</h5><hr/>";
      for (var i=0;i<obj.likes.length;i++){
        likesHTML += obj.likes[i]+"<br/>";
      }
      $('likesList').innerHTML = likesHTML;
      var recipeHTML = "<h5>"
        +obj.initial
        +"'s favorite recipe</h5>";
      for (key in obj.ingredients){
        recipeHTML += key
         +" : "
         +obj.ingredients[key]
         +"<br/>";
      }
      $('ingrList').innerHTML=recipeHTML;
    }

</script>
</head>
<body>

<div id='likesList' class='sidebar'>
<h5>Likes</h5><hr/>
</div>
<div id='ingrList' class='sidebar'>
<h5>Ingredients</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
   in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
```

❶ 解析JSON响应

❷ 使用解析后的对象

```
 
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

这样一种使用JSON的方式是相当简单的。我们用eval()来解析JSON响应❶，注意，在解析它之前应在字符串两端加上一对圆括号[①]。之后很自然地就可以在update()❷方法中使用解析后的对象，因为它也就是一个JavaScript对象嘛！

下面让我们稍稍看一下目前所需的服务器端代码。代码清单2-6列出了JSP文件的第五个版本[②]。

代码清单2-6 hello5.jsp

```
<jsp:directive.page contentType="application/javascript"/>
<%
String name=request.getParameter("name");
%>
{
  name: "<%=name%>",
  initial: "<%=name.substring(0,1).toUpperCase()%>",
  likes: [ "JavaScript", "Skiing", "Apple Pie" ],
  ingredients: {
    apples: "3kg",
    sugar: "1kg",
    pastry: "2.4kg",
    bestEaten: "outdoors"
  }
}
```

正如之前所说，我们在这里生成的绝大多数都是虚设的数据。我们感兴趣的是JSON字符串的创建，这里我们是直接手写代码，把变量值插入到适当的位置。

3. 讨论

我们通过这个示例说明了在客户端上解析JSON是极其简单的，仅此就很有价值了。不过，回顾图2-4你会发现在客户端和服务器之间的整个四步往返过程中，我们只涉及了其中的一个步骤：从JSON到客户端对象的转换。对于与本示例差不多的小型应用来说，这已经足够了，但是在更大型的应用中，或者需要处理更复杂的数据时，我们希望序列化（serialization）和反序列化（deserialization）的所有方面都能自动处理掉，这样在服务器上我们就只需专注于业务逻辑，在客户端上我们就只需专注于页面呈现。因此我们下面会有另外一个示例，在该示例中我们完成了客户端和服务器之间的整个往返过程。

2.2.2 使用 JSON 往返传输数据

当我们编写客户端回调时，JSON是我们的最爱，因为它让一切变得简单。不过，我们仍然

① 圆括号把代码限定为一个表达式。比如eval("{x:100}")与eval("({x:100})")的含义是不一样的。——译者注

② 严格来说，这段代码所产生的结果是不符合JSON规范的，因为按照JSON规范，属性名必须用双引号包裹起来。——译者注

要通过一个合乎规格的HTTP查询字符串把请求数据传给服务器，并手动构造JSON响应。如果我们可以用JSON来管理浏览器和服务器之间的所有通信，那么就可以省下大量额外的编码工作。

为了抵达幸福的彼岸，我们需要一些第三方程序库。好了，让我们开始编码，瞧瞧我们能有多幸福吧。

1. 问题

我们想让应用层和HTTP之间的所有接口都采用JSON，这样，客户端代码就可以纯粹用JavaScript的对象来编写，服务器代码也可以纯粹用Java（或PHP、.NET以及任何其他语言）的对象来编写。

2. 方案

可以用图2-4作为对照表，瞧瞧我们的设计与其相比存在哪些差距。在浏览器上，我们已经处理了第四步，即从响应文本到JavaScript对象的转换。所以我们还需要考虑客户端上从对象到JSON的转换。为此，我们需要用到www.json.org上的json.js程序库[①]。代码清单2-7展示了它的工作方式。

代码清单2-7 hello5a.html

```
<html>
<head>
<title>Hello Ajax version 5a</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript' src='json.js'> </script>
<script type='text/javascript'>
window.onload = function(){
  $('helloBtn').onclick = function(){
    var name = $('helloTxt').value;
    new Ajax.Request(
      "hello5a.jsp",
      {
        postBody:JSON.stringify({name:name}),
        onComplete:function(xhr){
```

❶ 导入json库

❷ 将对象转换到JSON

① 注意，应该是http://www.json.org/json2.js，而不是http://www.json.org/json.js。后者是一个旧版本，已经被前者取代。——译者注

```
            var responseObj = JSON.parse(xhr.responseText);
            update(responseObj);
          }
        }
      );
    };
};

function update(obj){
  $('helloTitle').innerHTML = "<h1>Hello, <b><i>"+obj.name+"</i></b></h1:
  var likesHTML = "<h5>"+obj.initial+" likes...</h5><hr/>";
  for (var i=0;i<obj.likes.length;i++){
    likesHTML+=obj.likes[i]+"<br/>";
  }
  $('likesList').innerHTML=likesHTML;
  var recipeHTML="<h5>"+obj.initial+"'s favorite recipe</h5>";
  for (key in obj.ingredients){
    recipeHTML+=key+" : "+obj.ingredients[key]+"<br/>";
  }
  $('ingrList').innerHTML=recipeHTML;
}

</script>
</head>
<body>

<div id='likesList' class='sidebar'>
<h5>Likes</h5><hr/>
</div>
<div id='ingrList' class='sidebar'>
<h5>Ingredients</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
   in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
 
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

❸ 将JSON转换到对象

我们首先要导入json.js程序库❶。之后，我们就可以使用JSON.parse()方法❸简化响应处理代码。更重要的是，我们可以重新考虑组装请求的方式。

迄今为止，我们一直是向服务器发送GET请求，并通过查询字符串传递数据。对于一般的数据请求来说这种方式是不错的，但是当我们想要更新信息或向服务器发送一个更复杂的请求时，最好是换用POST请求。POST请求除了有一组请求首部之外，还有一个请求主体，我们可以把任何想要的文本装填到请求主体中。这里我们就会用JSON。

我们仍使用Prototype库来发送请求，现在Ajax.Request构造器的选项增加了postBody属性，

它的值是调用JSON.stringify()的结果❷。stringify()接受一个JavaScript对象作为参数，对其进行递归遍历并输出为JSON。这样，我们的POST请求主体不会像HTML表单所发送的请求主体那样包含经过URL编码的键值对，而是会包含一个JSON字符串，就像这样：

```
{ name: 'dave' }
```

当然，对于这样一小段简单数据而言，这可能是大材小用了，但是通过这种方式，我们就能够传递有可能非常复杂的数据。

现在我们已经了解了方案的客户端部分，接下去该轮到服务器部分了。在这些例子中，我们在服务器上使用Java，但Java完全不知道JSON是什么，大多数服务器端语言也都不知道。为了让服务器端理解我们所发送的请求，我们需要引入第三方库。

无论使用哪种服务器端技术，你几乎都能在www.json.org（页面底部）找到适用的JSON库。我们选择Json-lib，它基于Doug Crockford最初的JSON的Java库。

Json-lib有相当多的工作要做。JSON以一种非常具有弹性的方式对结构化数据进行编码，而Java是一种强类型语言，所以两者有些不匹配。虽然如此，我们仍旧可以顺利完成任务。代码清单2-8列出的细节并不算太复杂[①]。

代码清单2-8　hello5a.jsp

```
<jsp:directive.page
  contentType="application/javascript"
  import="java.util.*,net.sf.json.*"
/>                                                    ❶ 导入JSON类
<%
String json=request.getReader().readLine();           ❷ 读取POST请求主体
      JSONObject jsonObj=new JSONObject(json);        ❸ 解析JSON字符串
      String name=(String)(jsonObj.get("name"));      ❹ 读取解析后的对象

      jsonObj.put("initial",
        name.substring(0,1).toUpperCase());           ❺ 添加新的值

      String[] likes=new String[]
        { "JavaScript", "Skiing", "Apple Pie" };
      jsonObj.put("likes",likes);

      Map ingredients=new HashMap();
      ingredients.put("apples","3kg");
      ingredients.put("sugar","1kg");
      ingredients.put("pastry","2.4kg");
      ingredients.put("bestEaten","outdoors");
      jsonObj.put("ingredients",ingredients);         ❻ 以JSON的形式输出对象
      %><%=jsonObj%>
```

为了在项目中使用Json-lib类，我们需要导入net.sf.json包，正如在<jsp:directive.page>标签中所做的那样❶，然后是代码。

① 这个jsp中的contentType的值，最好设置为application/json（由RFC4627规定），以表示其为一个JSON响应，而不是一个普通的JavaScript响应。——译者注

我们面临的首个挑战是对POST请求主体进行解码。Java Servlet API像许多Web技术一样，对HTML表单发送的POST请求进行了优化设计。而对于JSON请求的主体来说，我们就不能用`HttpServletRequest.getParameter()`了，而是需要通过`java.io.Reader`❷从请求中读取JSON字符串。对于其他技术来说，也有类似的功能。如果你使用PHP，那么就用`$HTTP_RAW_POST_DATA`变量。如果你使用.NET库，那么就需要从HttpRequest对象获得一个`InputStream`，这与我们在这里使用Java的方式差不多。

现在回到Java上来。一旦获得了JSON字符串，我们就可以把它解析成一个对象❸。因为在弱类型的JSON和强类型的Java之间是脱节的，所以Json-lib库定义了一个JSONObject类来表示解析后的JSON对象。我们可以通过`get()`方法❹来读取它，并从请求中取出名字。

现在我们已经反序列化了传进来的JSON对象，我们要对它进行操作，然后再发送回客户端。JSONObject类可以接受简单的变量类型，比如字符串、数组，还有Java的映射表（Map，也就是关联数组）❺，并把这些额外的数据加入到对象中。当修改了对象之后，我们就可以再次进行序列化❻，随后通过响应传回到浏览器。

简而言之，从客户端发送来一个请求，在服务器上修改它，并把它返回给客户端——就是这样！

3. 讨论

这是到目前为止最复杂的示例，它演示了一种通过基于文本的HTTP协议在网络上往返传输结构化对象的通信方式。借助程序库的帮助，客户端和服务器都可以理解JSON语法，所以，我们就不必自己编写代码来解析JSON。然而正如我们所注意到的，JSON源于弱类型脚本语言，所以仍旧需要一些转译过程。像这样一个系统的目标，就是能够对跨越网络的领域对象进行序列化和反序列化。如果我们的域对象是用Java（或C#）编写的，那么在传递给JSON序列化器之前，我们仍需要手动把它们转译成一般的散列表和数组。整个数据往返过程中最不优雅的代码就在JSP中，在那里我们要为JSONObject装配数据映射表和数据列表。这个问题在Java中特别突出，与Ruby或PHP相比，Java尤其缺少定义关联数组的紧凑语法。

4. 从方括号到尖括号

还有另一种基于文本的格式，客户端和服务器都能理解，那就是XML。绝大多数服务器端语言对XML都有非常好的支持，所以我们可能会发现在服务器上使用XML要比使用JSON更简单。下一节我们会研究XML和Ajax，看看是否确实如此。

2.3 在 Ajax 中使用 XML 和 XSLT

对于表达结构化数据而言，XML是一种非常成熟的技术。在绝大多数编程语言中，或者是作为语言的核心部分，或者是通过久经考验的库或扩展，XML都得到了良好的支持。在本章余下的部分，我们会看看各种XML技术是如何处理XML的，以此完成我们对基本的Ajax通信方式的考察。

用于Ajax请求的XMLHttpRequest对象已经内建了对XML的支持。迄今为止，我们一直是从HTTP响应体中提取出文本并进行解析。Web浏览器中的JavaScript并没有一个通用的XML解析器

可用[①]，但是在下面的示例中我们会看到，XHR对象可以帮我们解析XML响应。

2.3.1　解析服务器生成的XML

到目前为止，在Hello World应用的各个版本中，我们让服务器生成过HTML，生成过JavaScript，也生成过JSON响应。所有这些格式都主要是针对浏览器设计的，而不是针对服务器设计的。相反，XML在各种技术中常常用于服务器进程间的通信，简单如RSS种子（feed），复杂如XML-RPC和SOAP之类的Web服务协议。如果需要把领域对象中的信息传递到客户端，许多服务器端技术也提供了将对象按照XML格式进行序列化和反序列化的功能。

在这些场合，从服务器的角度来说，我们可能会发现以XML传输数据是很方便的。但是要真正用起来，我们也需要在客户端处理XML。我们将先看看XHR对象对XML的内建支持。与JSON一样，XML也是一个用于交换结构化数据的格式。要对两者进行比较，最好的办法就是给它们设定相同的任务，所以与上一节一样，我们要列出用户的喜好列表和最喜欢的菜谱，如图2-6所示。

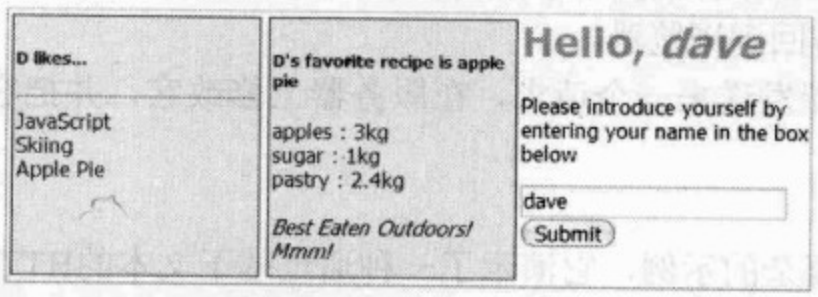


图2-6　解析了XML响应后的Hello World示例的第六版

1．问题

服务器发送XML结构化数据。我们需要在客户端解析这个数据。

2．方案

在客户端处理XML的第一步是使用XHR把响应解析成一个XML文档。第二步则是使用W3C的文档对象模型（DOM）标准来读取（也可能是改写）解析后的XML文档。在JavaScript编程中，我们已经一直在使用DOM进行HTML网页编程了。所以，好消息是我们可以运用这些已有的技巧来处理由Ajax所递送的XML文档。代码清单2-9列出了第六版Hello World应用的完整代码。

代码清单2-9　hello6.html

```
<html>
<head>
<title>Hello Ajax version 6</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
```

① 2004年6月所发布的E4X标准（ECMA-357，ECMAScript for XML）对JavaScript进行了扩展，增加了原生XML支持，但是目前只有使用SpiderMonkey引擎的浏览器如FireFox支持，并且该标准本身尚存在一些互操作性问题。——译者注

```
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript'>
window.onload = function(){
  $('helloBtn').onclick = function(){
    var name=$('helloTxt').value;
    new Ajax.Request(
      "hello6.jsp?name="+encodeURI(name),
      {
        method:"get",
        onComplete:function(xhr){
          var responseDoc = xhr.responseXML;
          update(responseDoc);
        }
      }
    );
  };
};

function update(doc){
  var personNode = doc
    .getElementsByTagName('person')[0];
  var initial = personNode
    .getAttribute('initial');
  var nameNode = personNode
    .getElementsByTagName('name')[0];
  var name = nameNode.firstChild.data;
  var likesNode = personNode
    .getElementsByTagName('likes')[0];
  var likesList = likesNode
    .getElementsByTagName('item');
  var likes = [];
  for (var i=0;i<likesList.length;i++){
    var itemNode = likesList[i];
    likes[i] = itemNode
      .firstChild.data;
  }
  var recipeNode = personNode
    .getElementsByTagName('recipe')[0];
  var recipeNameNode = recipeNode
     .getElementsByTagName('name')[0];
  var recipeName = recipeNameNode.firstChild.data;
  var recipeSuggestNode = recipeNode
    .getElementsByTagName('serving-suggestion')[0];
  var recipeSuggest = recipeSuggestNode.firstChild.data;
  var ingredientsList = recipeNode
    .getElementsByTagName('ingredient');
```

❶ 读取XML响应

❷ 使用DOM提取数据

```
var ingredients = {};
for(var i=0;i<ingredientsList.length;i++){
  var ingredientNode = ingredientsList[i];
  var qty = ingredientNode.getAttribute("qty");
  var iname = ingredientNode.firstChild.data;
  ingredients[iname] = qty;
}

$('helloTitle').innerHTML =
  "<h1>Hello, <b><i>"
  +name
  +"</i></b></h1>";
var likesHTML = '<h5>'
  +initial
  +' likes...</h5><hr/>';
for (var i=0;i<likes.length;i++){
  likesHTML += likes[i]+"<br/>";
}
$('likesList').innerHTML = likesHTML;
var recipeHTML = "<h5>"
  +initial
  +"'s favorite recipe is "
  +recipeName
  +"</h5>";
for (key in ingredients){
  recipeHTML += key+" : "
    +ingredients[key]
    +"<br/>";
  }
  recipeHTML+="<br/><i>"
    +recipeSuggest
    +"</i>";
  $('ingrList').innerHTML=recipeHTML;
}

</script>
</head>
    <body>

    <div id='likesList' class='sidebar'>
    <h5>Likes</h5><hr/>
    </div>
    <div id='ingrList' class='sidebar'>
    <h5>Ingredients</h5><hr/>
    </div>
    <div>
    <div id='helloTitle'>
    <h1>Hello, stranger</h1>
    </div>
    <p>Please introduce yourself by entering your name
       in the box below</p>
    <input type='text' size='24' id='helloTxt'></input>
     
    <button id='helloBtn'>Submit</button>
```

❷ 使用DOM提取数据

❸ 装配HTML

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

第一步显然是最简单的。直接从以前的responseText换成读取responseXML属性❶，我们就可以获得一个XML文档对象。然后，我们重写了update()函数以接受这个XML对象。此处我们所要做的就是从XML对象中读取每个数据元素❷，然后把数据转换为HTML呈现出来❸。

在实践中，这些步骤都不困难，就是相当冗长。使用DOM方法和属性，例如getElementsByTagName()、getAttribute()、firstChild等，就可以获得想要的数据，只是我们需要一级一级向下钻探。这些属性和方法与我们在处理HTML文档时所用到的属性和方法是一致的，所以这里就不一一介绍了。如果你用DOM操作过HTML，应该对这些用法已经很熟悉了；如果你没怎么用过，在网上也有关于这些方法的大量信息可供参考。

当提取出了所需的数据之后，我们就可以直接装配HTML内容用于更新UI。

我们之前已经提到过不少适合由服务器生成XML的场景，在本示例中，为了保持简单并且避免过于深入那些仅适用于单一编程语言的技术，我们直接在JSP中手工生成了XML。为完整起见，代码清单2-10还是列出了该JSP。

代码清单2-10 hello6.jsp

```
<jsp:directive.page contentType="text/xml"/>
<%
String name=request.getParameter("name");
%>
<person initial="<%=name.substring(0,1).toUpperCase()%>">
 <name><![CDATA[<%=name%>]]></name>
 <likes>
  <item>JavaScript</item>
  <item>Skiing</item>
  <item>Apple Pie</item>
 </likes>
 <recipe>
  <name>apple pie</name>
  <ingredient qty="3kg">apples</ingredient>
  <ingredient qty="1kg">sugar</ingredient>
  <ingredient qty="2.4kg">pastry</ingredient>
  <serving-suggestion>
   <![CDATA[Best Eaten Outdoors! Mmm!]]>
  </serving-suggestion>
 </recipe>
</person>
```

注意，这里我们把响应的contentType设为text/xml。之前我们设置contentType主要是演示一种好的习惯，而在本例中确是基于实践上的原因。如果不将MIME类型设为xml类型（text/xml或application/xml），那么XHR对象的responseXML属性可能无法正确装入数据。

JSP的其余部分就没什么花头了。对于一个老练的Java和XML程序员来说，直接以文本方式输出XML，恐怕是太小儿科了。更健壮（robust）的解决方案会使用如JDOM这样的程序库通过编程来生成XML文档，我们也鼓励读者在实践中采用这样的方案。不过，我们在这里还是保持

简单——虽然可能简单过头了——以凸显我们的目标，而不至过于深入特定于Java的程序库。毕竟，本书的主要目的是讲授客户端技术，至于是选择Java还是选择PHP、Ruby或.NET，其实并不重要。

所以，回到代码上来，我们只不过是创建了一个XML文档的模板，大部分是虚设的数据，同时也加入了少数动态的值。下面让我们总结一下从这个示例所得到的经验。

3．讨论

第一次与XML和Ajax亲密接触，心情有点复杂。最初还算不错，XHR对象有对XML的专门支持。然而，用DOM手动遍历XML响应确实挺冗长，而且不怎么有趣。这样的编码经历足以让许多开发者放弃XML转向JSON。

DOM是一种语言无关的（language-independent）标准，除了我们熟悉的JavaScript/Web浏览器上的实现，在Java、PHP、C++和.NET中也都有实现。如果观察一下XML在Web浏览器之外是如何使用的，就可以发现，DOM用的并不多，因为有另一些更称手的XML处理技术。幸运的是，这些技术也同样可以在浏览器中使用。我们将在下一节中看到，如何利用这些技术让Ajax和XML更好地协同工作。

2.3.2　用XSLT和XPath来更好地处理XML

我们在上一个示例中所看到的XML处理技术展示了在XHR对象的所有实现中都直接可用的基本功能。但是直接处理DOM令人不爽，特别是你可能已经习惯于其他语言中更加现代的XML处理技术。其中最常见的就是XPath查询和XSLT转换。

XPath是一种用于从XML文档中提取数据的语言。在清单2-9中，我们必须在文档中一个节点一个节点地向下钻探。而使用XPath，只要一行代码就可以遍历多个节点。XSLT则是一种基于XML的模板语言，让我们能更方便地从XML文档生成各种格式的内容，例如HTML，并且能更好地分离逻辑和表现。XSLT样式表（也叫模板——不要与级联样式表混淆起来）内部使用XPath将数据绑定到表现上。

好消息是，许多浏览器尤其是Firefox和IE都支持XSLT转换和XPath查询。而且它们直接暴露给JavaScript引擎的本机（native）对象，所以性能很好。Safari还没有本机XSLT处理器，所以，如果必须支持（非Firefox的）Mac用户，这就不是一个好选择了[①]。

在下面的示例中，我们将给Prototype库放个大假，改请Sarissa程序库来演示如何利用跨浏览器的XSLT和XPath简化基于XML的Hello World示例。

1．问题

用DOM来处理Ajax的XML响应既费时间又很麻烦。我们希望使用现代XML技术让Ajax和XML的开发变得更容易。

---

① Safari 3将开始支持XSLT转换的JavaScript编程接口。此外，准确地说，Safari 2也有XSLT处理器，因为可以通过XML文档中的<?xml-stylesheet?>处理指令进行XSLT转换，只是Safari 3之前的版本没有提供像XSLTProcessor那样的JavaScript编程接口。不过从理论上来说，可以结合使用<?xml-stylesheet?>处理指令、iframe以及data协议来模拟XSLTProcessor的功能。此外，你也可以使用一些纯粹用JavaScript实现的XSLT库，如后文提到的AJAXSLT，尽管其效率会比本机实现差许多。——译者注

2. 方案

XPath和XSLT可以帮你简化工作。IE和Firefox都支持这些技术，但是方式却大相径庭。对于大多数跨浏览器兼容性问题，最好的解决方法就是采用一个第三方程序库，给我们的代码提供一个统一的前端接口。对于本示例，我们选择了Sarissa（http://sarissa.sf.net），它所提供的跨浏览器包装器涵盖了在浏览器中处理XML的方方面面。代码清单2-11展示了这个以XSLT和XPath驱动的应用所需的客户端代码。

代码清单2-11 hello7.html

```
<html>
<head>
<title>Hello Ajax version 7</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
<script type='text/javascript'
  src='sarissa.js'> </script>
<script type='text/javascript'
  src='sarissa_ieemu_xpath.js'> </script>
<script type='text/javascript'
  src='sarissa_dhtml.js'> </script>
<script type='text/javascript'>

var xslDoc=null;

window.onload=function(){

  xslDoc=Sarissa.getDomDocument();
  xslDoc.load("recipe.xsl");

  document.getElementById('helloBtn')
  .onclick = function(){
    var name = document.getElementById('helloTxt').value;
    var xhr = new XMLHttpRequest();
    xhr.open("GET",
      "hello7.jsp?name="
      +encodeURI(name),true);
    xhr.onreadystatechange = function(){
      if (xhr.readyState == 4){
        update(xhr.responseXML);
      }
    };
```

❶ 导入Sarissa程序库

❷ 装载XSLT样式表

❸ 创建XHR对象

❹ 指派回调函数

```
    xhr.send("");
  };
};

function update(doc){
  var initial = doc.selectSingleNode(
    '/person/@initial'
  ).value;
  var name = doc.selectSingleNode(
    '/person/name/text()'
  ).nodeValue;
  document.getElementById('helloTitle')
   .innerHTML = "<h1>Hello, <b><i>"
     +name+"</i></b></h1>";

  var likesList = doc
    .selectNodes('/person/likes/item');
  var likes = [];
  for (var i=0;i<likesList.length;i++){
    var itemNode = likesList[i];
    likes[i]=itemNode
      .firstChild.data;
  }
  var likesHTML='<h5>'
    +initial+' likes...</h5><hr/>';
  for (var i=0;i<likes.length;i++){
    likesHTML += likes[i]+"<br/>";
  }
  document.getElementById('likesList')
    .innerHTML = likesHTML;

  var personNode = doc.selectSingleNode('/person');

  var xsltproc = new XSLTProcessor();
  xsltproc.importStylesheet(xslDoc);
  Sarissa.updateContentFromNode(
    personNode,
    document.getElementById('ingrList'),
    xsltproc
  );
}
  </script>
  </head>
  <body>

  <div id='likesList' class='sidebar'>
  <h5>Likes</h5><hr/>
  </div>
  <div id='ingrList' class='sidebar'>
  <h5>Ingredients</h5><hr/>
  </div>
  <div>
  <div id='helloTitle'>
  <h1>Hello, stranger</h1>
  </div>
```

❺ 选择单个节点

❻ 选择多个节点

❼ 调用XSLT转换

```
<p>Please introduce yourself by entering your name
   in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
 
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

我们首先需要导入Sarissa程序库❶。除了导入核心库，我们也要导入支持库，它为Firefox提供了IE风格的XPath接口。我们还导入了一个助手程序库[①]，它提供了将XSLT生成的内容插入到网页中的便捷方法。

使用XSLT来生成内容需要从服务器读取两个XML文档，一个是样式表（即模板），一个是数据。像之前一样，我们会按需读取数据，但是样式表可以在应用加载时预先加载❷。这里我们用的是DomDocument对象而非XHR，它是Sarissa给我们提供的跨浏览器的包装对象。

我们使用一个XHR对象来加载XML数据。因为在这个示例中我们不会使用Prototype库，所以需要手动创建XHR对象❸和指派回调函数❹。不过可以看到这里的代码比在第1章中的代码要简单许多，因为我们可以直接访问原生XHR对象，即使是在IE浏览器上。在内部，Sarissa会进行对象检测，如果找不到原生的XHR对象，就会帮我们创建一个，该XHR对象会“暗地里”创建一个ActiveX控件并使用它。

当我们有了XHR对象后，就可以传一个DOM对象给update()方法了。在使用DOM时，这个部分曾是我们的麻烦源头。而使用XPath，我们用单独的一行代码就可以穿越DOM节点的多个层次。例如，XPath查询：

```
/person/name/text()
```

选取文档顶端的<person>标签下面直接嵌套的<name>标签的内部文本。XPath这个主题很大，我们在这里无法深入展开。如果你是XPath和XSLT的新手， http://zvon.org是一个不错的起点。DOM节点上的selectSingleNode()❺和selectNodes()❻方法通常只能在IE中使用，但是我们加载的第二个Sarissa库文件提供了Firefox/Mozilla下的实现。我们用XPath提取名字和喜好列表的数据，然后手工装配那些区域的HTML内容，因为它们还是挺简单的。菜谱部分就比较复杂，所以该是XSLT出马的时候了。

最后一步就是执行XSLT转换❼。XSLT处理器（XSLTProcessor）对象在Mozilla中是原生的，而在IE下则由Sarissa提供。我们传给它一个样式表的引用，然后调用updateContentFromNode()方法。这个助手方法会让数据（即personNode）通过XSLT处理器，然后将所得到的HTML写入一个指定的DOM节点（即ingrList）。

当然，为此我们还需要提供一个XSLT样式表，其代码列于代码清单2-12中。

代码清单2-12 recipe.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
```

① 自Sarissa 0.9.9开始，助手库被并入了核心库。——译者注

```
    Transform">
<xsl:output method="xml"/>

 <xsl:template match="/person">
  <div>
  <h5><xsl:value-of select='@initial'/>'s
   favorite recipe is
  <xsl:value-of select='recipe/name'/></h5>
  <p><xsl:apply-templates select="recipe/ingredient" /></p>
  <p><i><xsl:value-of select='recipe/serving-suggestion'/></i></p>
  </div>
 </xsl:template>

 <xsl:template match="ingredient">
  <xsl:value-of select='@qty'/> : <xsl:value-of select='.'/><br/>
 </xsl:template>

</xsl:stylesheet>
```

我们的XSLT样式表相当简单。它是普通的XHTML标记以及带有xsl前缀的特殊标签的混合体。xsl前缀表示XSLT的命名空间，该命名空间下的标签会被作为处理指令。<xsl:template>标签指定了当节点与match属性中的XPath查询正好匹配时所要输出的各块内容。<xsl:value-of>输出所匹配的节点中的数据使用的也是XPath表达式。<xsl:apply-templates>标签把节点转给其他模板进行下一步处理。在本例中，菜谱的每个原料（ingredient）节点都会由第一个模板转给第二个模板，从而生成一个列表。

我们在这里还是没有足够篇幅来对XSLT样式表规则进行全面讲解。如果你希望了解更多，我们建议你访问http://zvon.org。

最后，在服务器端，这个示例所用的JSP与上一个示例是完全一样的，即代码清单2-10，所有的改动都在客户端代码那边。

3. 讨论

XSLT和XPath确实简化了客户端的XML处理代码。在更复杂的应用中，考虑到编码效率，XSLT和XPath比DOM更容易扩展[①]。我们建议，如果考虑在Ajax中使用XML，就应该研究一下这些技术。

在JSON一节，我们讨论了在客户端和服务器之间往返传输结构化数据的思路。Sarissa推广了这一方法，采用XML而非JSON作为交换格式，因为它也跨浏览器支持XML对象的序列化。我们之前已经提到过，几乎所有服务器端技术都支持以XML格式进行序列化和反序列化。我们这里就不举完整的示例了，但其原理和JSON是类似的。当在Java中使用JSON时，我们注意到需要不少手工工作来构建JSON响应，因为在弱类型的JSON和强类型的Java之间存在不匹配。Java和XML之间的转换也存在同样的问题，但是这一领域更为人所熟知，并且已经有不少开箱即用的（out-of-the-box）解决方案，比如Castor和Apache XMLBeans。

① 与使用DOM相比，使用XSLT和XPath编码所需投入的精力要小许多，而随着应用规模的增长，前者的优势会越来越显著。——译者注

我们所介绍的Sarissa提供了上述技术的一站式解决方案，但它并不是唯一的选择。如果你只需要XPath查询，那么mozXPath.js程序库（http://km0ti0n.blunted.co.uk/mozXPath.xap[①]）是一个轻量级的替代品，它也支持Opera浏览器[②]。另外，如果你喜欢XSLT并且需要让它跑在Safari上，那么你可以试一下Google的AJAXSLT，一个完全以JavaScript实现的XSLT引擎（http://goog-ajaxslt.sf.net[③]）。不过请注意，比起IE和Mozilla中的本机引擎，AJAXSLT很慢，而且其XSLT实现并不完整[④]。所以在编写样式表时，你需要留意AJAXSLT的限制，并尽量保持短小。

我们对Ajax技术的回顾接近尾声了。在最后一节中，我们将探索另一种利用XML和基于SOAP协议的Web服务的互联网技术，并且看看Ajax如何能与其对接。

2.4 在 Web 服务中使用 Ajax

在本节中，我们会看到如何通过SOAP协议调用运行在远程服务器上的Web服务。归根到底，Web服务不过就是把XML数据传来传去。XHR对象最适合这样的任务了，因此通过SOAP调用远程方法也不像看上去那么吓人。

IE和Mozilla浏览器都有原生对象可以用来调用Web服务。不幸的是，这些对象没法移植到其他浏览器中，开发者必须自己写个框架来选择适当的对象。微软有若干个页面介绍其浏览器端SOAP，网址是 http://msdn2.microsoft.com/en-us/library/ms531032.aspx 。微软的实现是基于JavaScript和VBScript的[⑤]。Mozilla在http://www.mozilla.org/projects/webservices/上对其Web服务作了说明，更详细的信息可以参考http://developer.mozilla.org/en/docs/SOAP_in_Gecko-based_Browsers[⑥]。这些浏览器端SOAP需要通过在各自浏览器中构造相应的原生对象来访问。

幸运的是，还有另一条路。我们不是编写高层API以利用IE或Mozilla的对象，而是创建我们自己的程序库，使用XMLHttpRequest来交换XML，并自行解析和生成SOAP消息。这样，在不支持微软和Mozilla的SOAP API的浏览器上也能运行我们的代码。正好有IBM的同志创建了一个这样的程序库，叫做ws-wsajax。你可以在http://www.ibm.com/developerworks/webservices/library/ws-wsajax/中找到它[⑦]，本节后续部分会使用这个程序库。

我们为本示例简化了UI，去掉了菜谱。传入访问者的名字后，会返回一个映射表（map），有三个条目：名字、首字母以及喜好列表。图2-7是本示例的UI。

本节假定你已经对SOAP和SOAP-RPC有所了解。同样地，有不少书可以参考，也有许多不

---

① 该程序库的当前地址应为http://km0.la/js/mozxpath/mozxpath.js。——译者注

② 只支持Opera 9+。——译者注

③ 该项目的主页已经迁往http://code.google.com/p/ajaxslt/。——译者注

④ 例如不支持XPath函数`local-name()`和`namespace-uri()`，不支持`xsl:import`、`xsl:include`、`xsl:apply-imports`、`xsl:number`、`xsl:key`等。——译者注

⑤ 微软的实现是一个HTC文件，其中的脚本是用JavaScript写的，但其运行时会动态生成一些VBScript代码。——译者注

⑥ 该文档说明了Mozilla的底层SOAP API，此外http://developer.mozilla.org/en/docs/Accessing_Web_Services_in_Mozilla_Using_WSDL_Proxying说明了如何直接使用WSDL。——译者注

⑦ 这其实是IBM的developerWorks网站上的一篇文章，内有代码下载链接。该文有中文版，网址为http://www.ibm.com/developerworks/cn/webservices/ws-wsajax/。——译者注

错的在线教程详细讨论了这一主题。

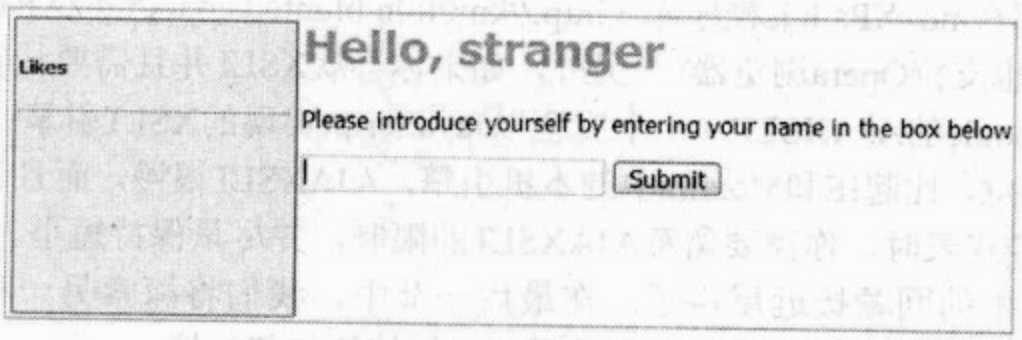


图2-7　第8版Hello World示例。在这里我们简化了表现层，从UI中去掉了recipe元素

1. 问题

你需要从Web浏览器执行SOAP-RPC，并将所得到的SOAP响应显示为HTML。

2. 方案

在本节中，我们会用IBM的SOAP工具包编写一个小型的客户端来访问我们的Hello World SOAP服务，该服务是用Apache的Axis框架（http://ws.apache.org/axis/）编写的。让我们首先定义我们的Web服务。Axis可以轻松建立Web服务的原型，只要把Java类以特殊的文件扩展名.jws保存即可。像JSP一样，会有一个专门的servlet即AxisServlet对.jws文件按需编译，虽然这对于产品级用途来说还不够健壮，但对于我们简单的演示用途来说已经绰绰有余了。代码清单2-13列出了Hello World服务所需的简单的.jws文件。

代码清单2-13　HelloWorld.jws

```
import java.util.Map;
import java.util.HashMap;

/**
 * class to list headers sent in request as a string array
 */
public class HelloWorld {

    public Map getInfo(String name) {
      String initial=name.substring(0,1).toUpperCase();
      String[] likes=new String[]
  { "JavaScript", "Skiing", "Apple Pie" };
      Map result=new HashMap();
      result.put("name",name);
      result.put("initial",initial);
          result.put("likes",likes);
          return result;
        }

    }
```

这个类只有一个方法，这个方法会被映射为一个SOAP-PRC函数。该函数接受一个字符串类型的参数，返回一个关联数组（即Java中的映射表）。

用浏览器访问HelloWorld.jws会得到一个MSDL（Web Service Description Language，Web

服务描述语言）的文件。SOAP客户端，例如我们所用的IBM库可以根据该文件构建客户端存根（stub）[①]，让我们能调用这个服务。代码清单2-14列出了根据这个类生成的WSDL。

代码清单2-14 WSDL for HelloWorld.jws

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://localhost:8080/AiP2/HelloWorld.jws"
  xmlns:apachesoap="http://xml.apache.org/xml-soap"
  xmlns:impl="http://localhost:8080/AiP2/HelloWorld.jws"
  xmlns:intf="http://localhost:8080/AiP2/HelloWorld.jws"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->
 <wsdl:types>
  <schema
    targetNamespace="http://xml.apache.org/xml-soap"
    xmlns="http://www.w3.org/2001/XMLSchema">
   <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
   <complexType name="mapItem">
    <sequence>
     <element name="key" nillable="true" type="xsd:anyType"/>
     <element name="value" nillable="true" type="xsd:anyType"/>
    </sequence>
   </complexType>
   <complexType name="Map">
    <sequence>
     <element maxOccurs="unbounded" minOccurs="0"
       name="item" type="apachesoap:mapItem"/>
    </sequence>
   </complexType>
  </schema>
 </wsdl:types>
       <wsdl:message name="getInfoResponse">
          <wsdl:part name="getInfoReturn" type="apachesoap:Map"/>
       </wsdl:message>
       <wsdl:message name="getInfoRequest">
          <wsdl:part name="name" type="xsd:string"/>
       </wsdl:message>
       <wsdl:portType name="HelloWorld">
          <wsdl:operation name="getInfo" parameterOrder="name">
             <wsdl:input message="impl:getInfoRequest"
                name="getInfoRequest"/>
             <wsdl:output message="impl:getInfoResponse"
                name="getInfoResponse"/>
```

① 存根（stub），是分布式计算中的概念，大体是指一段客户端或服务器端的代码，客户端存根负责将本地调用的参数转换为远程调用的参数，将远程调用的结果转换回本地调用的结果，服务器端存根则执行相反的转换。存根将分布式调用的通讯细节封装了起来，从而简化了客户端或服务器端程序的开发。——译者注

```
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HelloWorldSoapBinding" type="impl:HelloWorld">
        <wsdlsoap:binding style="rpc"
            transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="getInfo">
            <wsdlsoap:operation soapAction=""/>
            <wsdl:input name="getInfoRequest">
                <wsdlsoap:body
                    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                    namespace="http://DefaultNamespace" use="encoded"/>
            </wsdl:input>
            <wsdl:output name="getInfoResponse">
                <wsdlsoap:body
                    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                    namespace="http://localhost:8080/AiP2/HelloWorld.jws"
                    use="encoded"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="HelloWorldService">
        <wsdl:port binding="impl:HelloWorldSoapBinding"
            name="HelloWorld">
            <wsdlsoap:address
                location="http://localhost:8080/AiP2/HelloWorld.jws"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

这个WSDL包含了针对每个RPC调用的参数类型和返回类型的详细信息，包含了函数绑定信息，还包含了其他一些客户端和服务器所需的用以指明数据交换特性的细节。幸运的是，这个WSDL由Axis自动生成，并由IBM工具包使用，我们并不需要了解其中任何一行代码。

现在让我们回到客户端代码。代码清单2-15列出了第8版Hello World应用的完整代码。

代码清单2-15　hello8.html

```
<html>
<head>
<title>Hello Ajax version 8</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
```

```
</style>
<script type='text/javascript'
  src='prototype_v131.js'> </script>
<script type='text/javascript' src='ws.js'> </script>
<script type='text/javascript'>

window.onload=function(){
  $('helloBtn').onclick = function(){
    var name=$('helloTxt').value;
    var wsNamespace = '../axis/HelloWorld.jws';
    var wsCall = new WS.Call(wsNamespace);
    var rpcFunction = new
      WS.QName('getInfo',wsNamespace);
    wsCall.invoke_rpc(
      rpcFunction,
      [{name:'name',value:name}],
      null,
      function(call,envelope){
        var soapBody = envelope.get_body();
        var soapMap = soapBody
          .get_all_children()[1].asElement();
        var itemNodes = soapMap
          .getElementsByTagName('item');
        var initial = "";
        var likes = [];
for (var i=0;i<itemNodes.length;i++){
  var itemNode = itemNodes[i];
  var key = itemNode
    .getElementsByTagName('key')[0]
              .firstChild.data;
            if (key == 'initial'){
              initial = itemNode
                .getElementsByTagName('value')[0]
                .firstChild.data;
            }else if (key == 'likes'){
              var likeNodes = itemNode
                .getElementsByTagName('value')[0]
                .getElementsByTagName('value');
              for (var j=0;j<likeNodes.length;j++){
                likes[likes.length] = likeNodes[j]
                  .firstChild.data;
              }
            }
          }
          update(initial,likes);
        }
      );
    };
  };

  function update(initial,likes){
    var content = "<h5>"+initial
      +" likes...</h5><hr/>";
```

❶ 导入Prototype库

❷ 导入IBM WS程序库

❸ 根据WSDL创建客户端

❹ 引用PRC函数

❺ 传入PRC参数

❻ 定义回调函数

❼ 更新UI

```
    for (var i=0;i<likes.length;i++){
      content += likes[i]+"<br/>";
    }
    $('likesList').innerHTML = content;
}

</script>
</head>
<body>

<div id='likesList' class='sidebar'>
<h5>Likes</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
   in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
 
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

这里有许多内容，所以让我们一行一行看。首先，我们需要导入IBM库❷。由于这个库是建立在Prototype库之上的，所以我们也要导入Prototype。又因为它用到的是Prototype的一个老版本（v1.3.1），所以我们修改了这个版本的Prototype库的文件名，以免与我们的其他示例相混淆❶。

要使用Web服务，我们首先需要引用WSDL，并把它“喂”给WS.Call对象❸。然后我们提取出一个特定函数的引用，它是一个WS.QName对象❹。我们可以调用这个对象，以JavaScript对象作为输入参数（此处我们用一行JSON来定义）❺，并提供一个回调函数对响应进行解析❻。解析响应需要遍历大量节点。我们在这里处理的是SOAP节点而不是DOM节点，但SOAP节点可以转换成DOM节点。为了简明起见，这里我们并没有使用XPath。但如果要处理更大型的SOAP响应，当然就应该考虑XPath了。从响应中提取出数据后，我们照例将数据传给update()函数❼。同样地，这里我们也选择了最简单的方式，当然，你完全可以对SOAP响应使用XSLT转换。

3. 讨论

我们已经展示了在Ajax中使用SOAP是可行的，当然前提是SOAP服务来自于Ajax客户端所能访问的服务器，即符合浏览器的同源安全限制。如果你的后端系统已采用了SOAP，那么这是重用现有资源的有效途径。然而，我们倾向于认为，如果用作零起点（green-field）[①]开发的架构，并且没有外部实体的互操作性需求，SOAP是过于复杂了。

---

① 所谓green-field是指项目不存在任何既有的约束，就好像从一片空地开始建造，而老系统升级或要考虑与遗留系统对接就不是green-field。——译者注

IBM的SOAP工具包令调用服务变得非常轻松，但是解析响应就不那么方便了。SOAP-RPC的响应通常包含了很多命名空间，解码起来很是复杂。文档/文本式（document/literal-style）[①]的SOAP绑定提供的响应更简单，如果要在产品中使用这个工具包，可能最好采用这种风格的SOAP绑定。

像往常一样，这里是给程序员的忠告。如果你需要一个快速解决方案，SOAP可能不是好方法。然而，如果你正在创建一个大型应用，可以预见到将来会需要许多更新和扩展，同时还要整合你单位中的诸多系统，而且你也有时间和技能，那么浏览器端SOAP可能会帮上你。

2.5 总结

在第1章结束前，我们了解了如何发起一个Ajax请求，以及如何通过第三方程序库简化这一过程。自那时起，我们学习了许多新的内容，我们的关注点也不再是仅仅能发起请求那么简单，而是上升到了要怎样构造整个应用生命周期中的客户端和服务器之间的会话。

在本章中我们已经看过了好几种不同的技术，并评估了每种技术的长处和短处。我们首先看了在服务器上生成JavaScript代码的方式，明白了所生成的代码应该针对高层次的API编写，以防止客户端和服务器代码过分混乱。

接下来我们看了几种在客户端和服务器之间传送结构化数据的不同方式，先是JSON，然后是XML。对于每一种方式，我们首先了解了在接收到服务器数据之后如何对它进行解析，然后进一步考量了客户端和服务器之间完整的数据往返过程。通过往返传输数据，并由程序库代码在两端进行序列化和反序列化，我们就可以从低层面的管线设施中腾出手来，专心编写业务代码。

对比JSON和XML，JSON对客户端更有亲和力。而客户端XML则让我们头痛不已，但在采用XPath和XSLT之后，情况有显著改善。JSON和XML这两项技术之间并没有绝对的赢家，选择哪一种仍然只是个人爱好的问题，或者，如果要集成于遗留系统中，则取决于该系统本身更适应JSON还是XML。

在下一章中，我们会研究JavaScript，它是将整个Ajax应用连成一体的程序粘合剂。我们将讨论JavaScript编程思想的最新进展，并了解它们是如何帮助你编写出具有更好结构的Ajax应用程序代码的。而在本书第一部分的最后，我们会对一些流行的Ajax框架做一些讨论。

---

① 所谓SOAP绑定，是指服务如何转换为SOAP消息协议。wsdlsoap:binding元素上的style属性定义了绑定的风格，可以是RPC风格的，也可以是文档风格的。wsdlsoap:body元素上的use属性定义了SOAP主体的使用方式，主体中的数据可以经过编码，也可以直接使用文本。采用文档风格并且直接使用文本的，就是所谓文档/文本式（document/literal），而清单2-14则采用了RPC/编码式（RPC/encoded）。——译者注

第3章 面向对象的JavaScript与Prototype库

本章内容

- 使用JavaScript语言中的基本类型
- 编写有效的面向对象的JavaScript
- 运用Prototype程序库

像大多数Web开发者那样，你最初接触JavaScript大概是为一些相当基本的Web页面编写简短的脚本。比方说创建图像翻转效果（image rollover），或在表单提交前对输入字段进行一些简单的验证。

不过，既然你正在阅读本书，相信你的Web应用开发技能应该已经远远超过了这些入门内容——而你的Web应用页面所需的脚本也早已不是靠入门水准就可以对付得了的。

就现代Web应用的特点来看，其代码量和复杂度不断增长（无论是在服务器端还是在客户端）。而当你引入Ajax，即使是不大的应用，其客户端代码的复杂度和代码规模也会有显著跃升。

这就产生了一个紧迫的需求，JavaScript代码需要“长大成人”，要跟Java、C++和C#等服务器端语言编写的代码一样，得到同等程度的关注和维护。组织我们的客户端代码时，怀有像对服务器代码一样谨慎的态度，不仅能帮助我们对代码的创建过程保持头脑清醒，也有助于代码的可读性、可重用性、可测试性、可扩展性以及可维护性。

一种流行的组织代码的方法体系就是面向对象（object orientation）。我们之前提到的服务器端的那些语言都是面向对象的。JavaScript也是一种面向对象的（OO）语言，不过缺乏一些其他面向对象语言所具备的OO概念和特性。但是这并不意味着我们不能从其他语言的经验和概念中获益。JavaScript或许没有Java或C++所拥有的OO卖点，但是它也有一些独特的特性是其他语言所没有的。在本章中，你会学习这些特性，并了解如何通过充分发挥这些特性来合理运用面向对象的概念和技巧，从而将秩序引入到你的客户端JavaScript中。

学究们也许会认为我们在本章中所讨论的并不是真正的面向对象的代码，而只是带有面向对象风格的代码（object-orientation-influenced）[①]。不管怎样，我们还是把它叫做面向对象的JavaScript，只要我们都能理解它的实质含义就行了。

我们首先会看一下JavaScript中的一些很独特的方面，没有它们就不会有面向对象的

① 更常见的说法是JavaScript是一种基于对象的（object-based）语言。——译者注

JavaScript代码。并且我们要学习如何用好这些方面以运用OO技巧和概念来更好地组织代码。然后我们会介绍一个自由可用的JavaScript程序库Prototype，看看它如何帮助我们写出更好的JavaScript，特别是更好的面向对象的JavaScript的。

3.1 面向对象的 JavaScript

顾名思义，对象这一概念是任何一种面向对象语言的核心。JavaScript也不例外，不过对于熟悉更传统的OO语言如Java或C++[①]的人来说，它的方式看上去相当诡异甚或完全不可思议。

要理解怎样才能最好地使用JavaScript的OO特性，不仅要理解JavaScript对象，更要彻底掌握JavaScript函数。事实上，JavaScript语言中的函数相当独特和有趣，要运用JavaScript语言特性来驾驭复杂代码，函数正是关键所在。

因此，在本节中我们既要考察JavaScript对象的概念也要深入探索JavaScript函数。在对它们的特性和操作进行概述之后，我们会展示如何把这些概念结合起来，构成我们所称的面向对象的JavaScript的基础。

3.1.1 对象的基本原理

要在脑海里想象一个JavaScript对象，第一步就是要清除你由其他语言习得的关于Object类——通常是构建所有其他对象的基本单元的先入为主的观念。尽管JavaScript中Object类也是构建其他对象的基本单元，但在其他方面就大相径庭了。

或许想象JavaScript对象的最佳方式是把它看作键值对的无序集合，类似于其他语言中的映射表（Map）或字典（Dictionary）的概念。键值对被称为属性（property），由一个标识属性的键（key）也即属性名，与一个属性具有的值（value）构成。

你可能会认为属性与其他语言中的成员变量差不多，不过JavaScript语言并不需要对类的成员进行声明。要创建对象属性，在运行时直接给属性赋值就可以了。而且，这些属性的数据类型是动态的而不是预先声明好的。给属性赋的值是什么数据类型，属性就是什么数据类型，并且完全可以在它的生命周期中改变数据类型。

让我们看一个简单的示例。假设我们要把所收藏的音乐CD管理起来。一个表示CD的对象会有一些属性来记录它的标题、歌手或乐队，对了，还有存放它的架子或抽屉的编号。你知道，我们有太多的CD啦！

当然，一张CD还有不少其他特征，我们也可以为它们创建属性，不过作为示例，上面这三个已经足够了。要建立起这样一个对象，其代码大体如代码清单3-1所示。

代码清单3-1 创建并填装一个CD对象

```
var aCD = new Object();
aCD.title = 'The Lovin\' Spoonful Greatest Hits';
aCD.artist = 'The Lovin\' Spoonful';
aCD.location = 3;
```

① 以下所提到的“传统OO语言”或“其他语言”也几乎都是指C++和Java，而不包括被称为OO鼻祖的Smalltalk，或2.0版本以后的C#。——译者注

在代码清单3-1中，我们做的第一件事就是在Object构造器上调用new运算符，创建了一个Object类的实例。尽管看上去再寻常不过了，但这里还是有些很重要的微妙之处，对于理解new运算符来说这至关重要。

以前你可能不会把new看成运算符（operator），但是它确实是一种运算符。其运算元（operand）是一个函数，该函数会成为new所创建的对象的构造器（constructor）。

在对函数进行深入研究之后我们会进一步讨论构造器，就目前而言，我们只需了解通过new创建对象也要用到一个作为运算元的构造器函数，在本例中就是JavaScript语言预定义好的Object类，它产生一个空的对象实例，也就是一个没有属性的对象[①]。

在我们的这个示例中，这个空的对象被赋给叫做aCD的变量，这样我们可以在之后的三行代码中接着设置它的属性。

注意，我们并不需要像在其他语言中那样预先声明对象可以接受哪些属性。在JavaScript中，对尚不存在的属性进行赋值操作，就会使对象上产生这些属性。

如我们的示例所展示的，属性通常可用“点”（句点）运算符来引用，但也可以使用更一般化的“属性访问器（property accessor）”来引用。使用该运算符，location属性的赋值操作可以这样写：

```
aCD['location'] = 3;
```

这一语句与使用点号的aCD.location = 3是完全等价的。

如果属性名符合标识符的命名格式，就可以使用点号。如果属性名不符合格式，例如包含空格字符，就必须使用方括号：

```
ACD['the location of the CD'] = 3;
```

一般来说，为了可读性考虑，我们在命名对象属性时通常是遵循标识符格式的[②]。

现在我们有了一个对象的实例，它包含三个属性，描述了一张CD。这当然挺好，但是你可以看到，它不太可伸缩。我们花了四行代码来创建和装填这个对象，如果要输入许多CD——记得吗，我们跟你说过这是很大一笔收藏呢——我们需要很多行这样的代码。

就算我们不在乎代码的量，也应该注意到，这样子下去出错概率会变得很高。因为我们在创建CD对象时要显式地在每个实例上设置属性，所以任意一组赋值操作中的一个小小打字错误都可能演变成一个极难发现的大麻烦。记住，赋值操作会直接创建属性。想象一下，我们不小心把location打成了lcoation：

```
anotherCD.lcoation = 213;
```

赋值操作不会发生任何错误，但是在后面某一行代码处，我们会搞不懂实例怎么没了location属性。

我们真正需要的是建立一个CD实例的构造器（constructor），能在内部一致地处理这些赋值操作，这就是面向对象中的封装（encapsulation）概念——它使用起来大体如下：

```
var aCD = new CD('The Very Best of the Rascals', 'The Rascals', 6);
```

① 虽然可以读取到toString、valueOf、constructor等属性，但是这些属性其实不是该对象本身的属性，而是其原型对象的属性。——译者注

② 这样就可以使用点号——因为使用点号，代码更紧凑，更容易阅读。——译者注

这样，由单独的代码块（构造器）负责进行属性赋值，从而消除了在重复代码块中出现打字错误的可能。

不过在讨论如何将函数用作构造器之前，我们首先需要理解在JavaScript中函数本身是如何工作的。

3.1.2 函数是一等公民

乍一看，JavaScript函数与传统OO语言中的方法（method）差不多，但真正了解之后我们就会发现两者有许多地方是截然不同的。

这些差异大多基于这样的事实：JavaScript的函数是语言中“第一等级的（first-class）”[①]对象。所谓“第一等级”并不是高人一等（例如在飞行中享受更宽敞的头等舱座席），而是指函数与语言中的其他对象类型有着同等的地位和一样的特性。比如可以在运行时创建新的函数，可以通过不带函数名的字面量（literal）[②]直接创建出无名的函数对象，可以通过变量来引用函数，函数可以用作其他函数的参数，也可以是其他函数返回的结果，总而言之，可以像其他数据一样对待函数[③]。

下面让我们首先看一下如何声明和调用函数。

1. 声明和调用函数

要调用一个函数，当然先要建立它。最简单的就是声明一个函数，其语法看上去可能很熟悉：

```
function doSomething(value) {
  alert("I'm doing something with " + value);
}
```

但是函数并不一定要有名字。可以通过函数字面量（function literal）来创建函数，有时我们称之为无名函数（anonymous function）。

```
function(value) {
  alert("I'm doing something with " + value);
}
```

这挺有意思的，不过（至少在这个示例中）不那么有用，因为我们没有办法实际调用这一函数。一个无法调用的函数能有什么用呢？

但是记住，函数作为一等公民跟任何其他JavaScript对象一样，可以被赋给变量。有了这样一个引用，函数就可以通过该引用来调用。不仅是存于变量中的函数引用，还有作为函数参数被传入的引用，以及存于对象实例的属性中的函数也是如此。这让我们可以做许多有意思的事情。考虑下列代码：

---

① 英国计算机学家克里斯托弗·斯特雷奇（Christopher Strachey）于1960年代中期以“作为一等公民（first-class citizen）的函数”首次提出了这一术语。——译者注

② 字面量（literal）是指程序源代码中用来表示固定的值的符号序列。例如在大多数语言中，引号包围的字符序列即为字符串字面量（string literal），表示一个特定的字符串值。——译者注

③ 相反地，在C++、Java等语言中，函数（方法）就不是一等公民，而是二等公民。在这些语言中，函数（方法）不是一种类型（type），而是一种附属于类型的语言构造。通常，函数（方法）在编译时就已经确定，而无法在运行时创建，函数（方法）本身也是无法直接充当其他函数（方法）的参数或结果的。当然，利用语言特性或类库，比如C中的函数指针、C++的各种模板库中的Functor、Java中的无名类（anonymous class）、C#中的委托（delegate），等等，也可以在某些方面获得作为一等公民的函数所具有的性质。——译者注

```
var doSomething = function(value) {
  alert("I'm doing something with " + value);
}
```

现在，就可以使用代码doSomething('some value')了，就好像我们声明了以该名字命名的函数。

除了提供了另一种有趣的语法选择之外，通过引用来调用函数的能力在某些场合是非常有用的。考虑这段代码：

```
function saySomething(text) { alert('value: ' + text); }

function doSomething(value,onComplete) {
  // does something with value
  onComplete(value);
}

doSomething(213,saySomething);
```

第一个函数接受一个值并根据它产生一个提示（alert），第二个函数则就所传入的值执行一些操作。令人感兴趣的是，第二个函数允许调用者通过onComplete参数传入一个回调函数（callback function）的引用，以此定制在处理完成后发出怎样的通知。

回调函数的引用被传递给处理函数，然后通过引用，回调函数会被调用，这样就允许函数的调用者（caller）而不是函数本身来决定处理完成之后的行为。

“那有什么了不起的呢？”你也许会这样想：“我可以在处理函数返回之后做任何想做的事情，所以干嘛要多此一举？”对于一个同步的代码段来说这样想或许没错，但是如果我们考虑异步的场景，比如各种输入事件和Ajax，那么能够通过引用来回调函数就变得很有意义并很有必要了。

此外，这段代码示例的重点原本是展示如何以函数引用作为其他函数的参数。我们也可以用函数引用作为对象实例的属性值（property value）。如：

```
var o = new Object();
o.doSomething = function() { alert('Yo!'); }
```

然后可以这样调用函数：

```
o.doSomething();
```

当对象实例的属性是一个函数时，此函数就被称为对象的方法——这个概念可不像你想的那样简单。存储着函数引用的对象属性不单单是提供了一个存储引用的地方，还在方法和引用它的对象之间建立了一种关联。

这里就引出了函数上下文的概念。让我们来看看它到底是什么。

2. 理解函数上下文

所有函数都是在一个JavaScript对象的上下文（context）中执行的，即使你从未意识到它的存在。这一对象就称为函数上下文（function context），在函数体内可以通过保留字this使用它，你可能在C++和Java这些语言中就很熟悉this了。

当通过对象属性中所保存的对函数的引用（由此这一函数就成了对象的方法）来调用函数时，此函数中由this所引用的上下文对象就是拥有该方法的对象。而直接以函数名来调用的函数，它没有被存储到某个对象的属性中，从而不是任何对象的方法，但它调用时也是有上下文对象的，

那就是页面的窗口（window）对象[①]。

有一点很重要，函数上下文是函数的一次调用（invocation）所具有的属性，而不是函数本身的属性。让我们考虑代码清单3-2中的代码。

代码清单3-2 观察函数的上下文

```
function xyz() {                          ❶ 创建有名函数
  alert(this.handle);
}

                                          ❷ 创建对象并赋以函数
var o = new Object();
o.methodXyz = xyz;

window.handle = "I'm the window";         ❸ 创建handle属性
o.handle = "I'm o";

xyz();                                    ❹ 调用函数两次
o.methodXyz();
```

在这段代码中，我们首先创建了一个有名函数xyz❶，它会发出一个提示，显示出this所引用的某个对象上handle属性的值——换句话说，就是函数上下文对象的handle属性的值。注意，handle不是对象的内建属性，我们会创建这个属性以便区分各个对象。

接着，我们新建一个对象并赋给变量o❷。在这个对象上我们建立了名为methodXyz的属性以保存函数xyz的引用。然后在页面窗口对象上创建了一个名为handle的属性，在o所引用的对象上也创建了一个handle属性❸。这让我们在任何时刻都能很方便地确认引用的是哪个对象。

然后我们调用这个函数两次❹：一次直接通过xyz的函数名，一次则通过对象o上的methodXyz属性。代码执行后会产生两次提示，如图3-1所示。

两次提示依次显示，清楚地证明了尽管每次调用的都是同一个函数，但每次函数调用的函数上下文取决于调用函数的方式。

当通过对象属性来引用函数时，函数就成了对象的方法，而函数上下文就是对象本身。这一事实正是面向对象的JavaScript的基石。

不过，有些时候JavaScript解释器会把我们通常所认定的上下文对象换成另一个对象。这是咋回事呢？

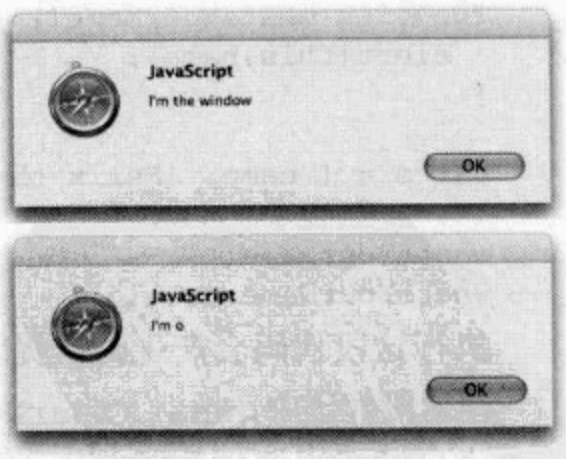


图3-1 同一个函数，不同的上下文

3. 当陌生人拿着拴绳

我们在上一节中看到，函数的上下文对象取决于函数如何调用。当函数成为对象的方法并通过该对象来调用时，函数的上下文就是这个对象。

我们会在许多示例中看到，有时候我们希望使用类方法作为事件处理器——例如用作按钮的onclick处理器。然后事情就变得有点绕了。你知道，当函数作为事件处理器被调用时，函数的

① 准确地说是JavaScript语言中的全局（global）对象，只是在浏览器中按惯例会将窗口（window）对象作为JavaScript的全局对象。——译者注

上下文会被设为触发该事件的那个元素，尽管函数已经是另一个对象的方法了。

是不是有点晕？

这也难怪。还是让我们做一个不太恰当但是可能会有帮助的类比吧。

假设你有只宠物蜥蜴（iguana）[①]，它代表一个函数。蜥蜴，在大自然造物中并不算最聪明，它们需要一种方法来找到主人，也就是你，你代表了函数的上下文对象。蜥蜴和你之间的这一连接就是一根拴绳，它代表了this变量。

当蜥蜴需要你的时候，比如说要吃东西了，它就可以顺着绳子找到你并得到一把健康美味的蒲公英嫩叶。但是，如果我们假设蜥蜴（函数）作为处理器被调用，这种情况下，上下文对象就变为了发起事件的对象——也就是说，绳子到了一个陌生人手里。当饥肠辘辘的蜥蜴沿着绳子找到这家伙时，却得不到食物。（好嘛，你会去喂一只陌生的蜥蜴吗？）

这就是我们要在示例中处理的问题。一种解决方法是在元素上设置属性，这样我们就可以由元素上的引用回到“正确的”对象。而在本章后面我们还会看到一个巧妙的方法，可以强制JavaScript解释器服从我们对于处理器上下文的意愿。

即使还没有掌握这些方法，只要是由我们主动发起调用，也还是能够控制以哪个对象作为函数调用的上下文的。虽然不适用事件处理器的情况（因为不是由我们发起调用），不过还是让我们了解一下，当事情由我们掌控时，怎样显式地指定用哪个对象作为函数上下文。

4. 设置函数上下文

每个函数（也即JavaScript内建的Function类的一个实例）都有一个名为call()的方法（记住，作为一等公民，函数和所有其他对象一样，也可以有属性和方法）。

当函数被“正常”调用时，如我们之前所述，函数上下文对象是由解释器决定的。

现在来看下面这一小段代码：

```
function whatsYourName() {
  alert(this.name);
}

var o = { name: 'Felix the Cat' };

whatsYourName();
whatsYourName.call(o);
```

执行这段代码会产生两次提示，如图3-2。

在这段代码中，whatsYourName()函数发出一个提示，参数是函数上下文对象上的name属性。

当直接调用该函数时，提示信息显示的是窗口的name属性，因为该函数调用的上下文对象正是窗口对象。所显示的字符串可能类似“file://localhost”，取决于你的浏览器以及载入页面的方式[②]。

---

① iguana，美洲绿鬣蜥，一种草食性大型蜥蜴，是最常见的蜥蜴饲养品种之一。——译者注

② window.name属性即窗口（window）或窗格（frame）的名字。可以指定超链接和表单的目标（target）为某个窗口/窗格的名字，这样所取回的文档就会显示在该窗口/窗格内。使用window.open()方法打开新窗口时可以通过方法的参数设定新窗口的名字，<frame>标签上的属性则可以指定窗格的名字，通常也可以用脚本动态地设置或改变window.name的值。其他时候，window.name的值一般会是一个空字符串。但是Safari在3.0版之前，其window.name属性的设计与其他浏览器有些不同，本书的截图正是取自Safari浏览器，所以我们会看到“file://localhost”这样奇怪的结果。——译者注

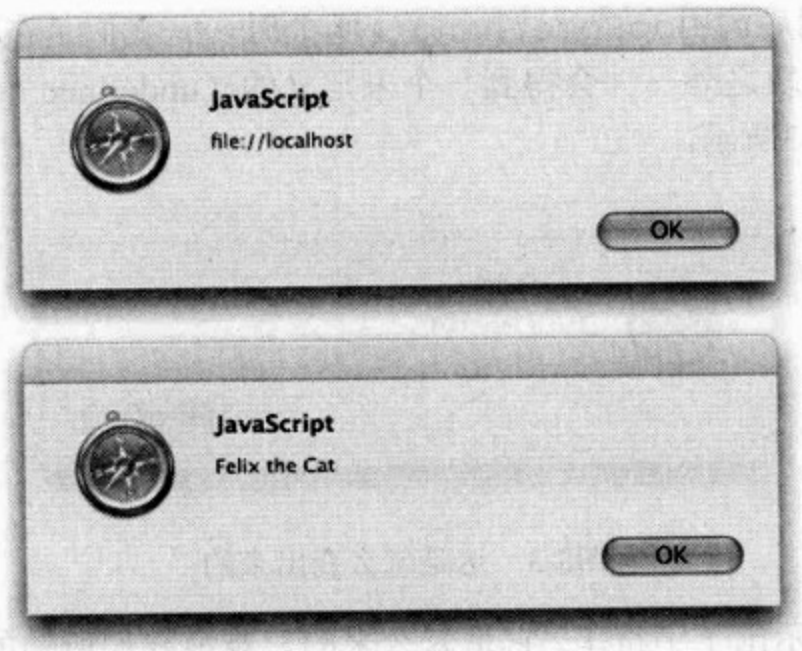


图3-2 如果需要，我们可以控制函数上下文

我们也定义了一个具有name属性的对象o（使用JSON记法）。当使用call()方法来调用函数并以对象o为参数时，显然，o被用作了函数调用的上下文。

掌握了这一点，我们就可以继续有关对象的讨论了，但是之前，我们还需要理解另一个有关函数的概念：闭包（closure）。让我们来看看它到底是什么。

5. 打开闭包看一看

如果你来自于Java或C++世界，那你可能从没有接触过闭包这个概念，因为在那些语言中并没有对应的概念。闭包这个概念很难意会，所以我们直接举例子，请看代码清单3-3：

代码清单3-3 创建闭包

```
var o = new Object();
o.setup = function() {
  var someText = 'This is some text';
  this.doSomething = function() {
    alert(someText);
  };
};
o.setup();
o.doSomething();
```

在代码清单3-3中，我们创建了一个新的对象并赋给了o，然后创建了setup()函数，用于对它进行初始化。（当然这一工作最好通过构造器来完成，我们稍后会讨论它。）

在setup()中，我们创建了一个叫做someText的局部变量，并给它设了一个字符串值。然后我们创建了一个叫做doSomething()的方法，它简单地发出一个提示消息，显示出someText的值。之后我们就调用setup()方法对对象进行初始化，然后调用doSomething()方法。

仔细查看这个方法的代码我们会发现一个问题：代码引用了someText变量，它是该函数所在代码块中的局部变量，而根据JavaScript语言的规则，someText变量在代码块结束后就超出作用域（scope）了。

因此，我们猜想当稍后调用doSomething()函数时——我们是在顶层调用函数，明显已经在定义someText的代码块之外——会得到一个未定义的（undefined）引用。但是真的执行时，我们会看到提示，如图3-3所示。

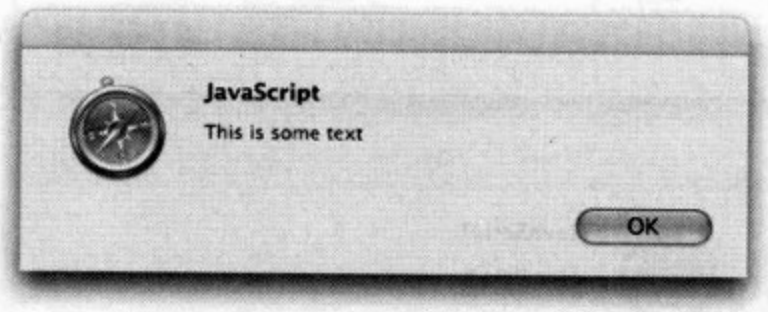


图3-3　这是怎么变出来的

奇怪！提示信息中显示出了字符串。它是怎么来的？难道我们所知的JavaScript的作用域规则有问题？

其实，当我们调用o.doSomething()时someText变量确实已经超出了作用域。如果我们在o.doSomething()后面再加上一句alert(someText)，就会发现JavaScript解释器确实会发出“someText未定义”的错误。然而，这个方法怎么就能工作呢？

这是因为，当JavaScript解释器创建一个函数（也就是一个Function类型的对象实例）时，它也为这个函数创建了一个闭包，闭包不仅包含函数本身，还包含函数创建时的作用域环境（the environment that is in scope）。

所以，函数在被调用时也能访问所有在函数声明时处于作用域中的变量。

这一概念很好很强大，但也易于混淆，需谨慎使用。随意乱用可能炮制出非常糟糕的代码。不过后面我们将会看到，如何借助闭包创造出面向对象的优雅代码。

另一个注意事项是：闭包创建时并不包含当前执行函数的函数上下文（即this引用）[①]。当深入探讨如何定义JavaScript类时，我们就会看到由此所引起的问题。

好了，我们对函数又多了一分了解了，现在我们可以来看看创建自定义JavaScript类所涉及的更多细节了，比如构造器。

3.1.3　对象构造器和方法

我们已经了解了什么是JavaScript对象，对于JavaScript的函数也有了更好的理解，下面我们准备看看如何通过JavaScript类运用对象和函数来创建出组织良好的（well-organized）JavaScript对象。

首先，让我们看一看new运算符是如何作用于函数并将函数转换为对象构造器的。

---

① 或者毋宁说，你无法在函数中访问它，因为函数中的this已经用来表示它自己的函数上下文。前面介绍函数上下文时提到过，普通的函数调用，即直接以函数名调用，而不是作为方法调用，实际上没有指定函数上下文，而是使用默认的全局对象（在浏览器中就是窗口对象）作为函数上下文。值得指出的是，这种默认设计对于开发者来说几乎毫无作用。一般来说，我们在一个类的方法中使用普通函数调用时，往往是希望利用它的闭包特性，并且很可能需要在闭包中访问表示对象实例的this引用，然而我们现在无法直接访问它，因为它被闭包自己的this所遮蔽了。正在制定中的ECMAScript第四版规范，即未来的JavaScript 2.0，可能会改变当前这一饱受诟病的设计。在新规范中，普通函数调用中的this不再指向默认的全局对象，而是其外层函数的this。——译者注

1. 定义构造器

考虑以下代码：

```
function Something(p1,p2,p3) {
  this.param1 = p1;
  this.param2 = p2;
  this.param3 = p3;
}
```

这个函数相当简单，它读取所传入的参数并将它们保存于当前的上下文对象中。熟悉Java或C++的面向对象编程的人一眼就能看出这是构造器常见的模式。不过，这究竟是不是构造器呢？

答案：是也不是——或者更精确地说，视情况而定。

我们想象一下，在定义了这个函数之后，如果我们这样调用它：

```
Something(1,2,3);
```

会发生什么呢？

当函数执行时，它在上下文对象上创建三个属性。在这个调用中，由于函数是直接从顶层代码调用的，所以上下文对象就是当前页面的窗口对象。

这好像没什么特别之处，是吧？

不过下面这个就厉害了，我们不是直接调用函数，而是给函数用上new运算符，如

```
new Something(1,2,3);
```

new运算符创建一个新的空Object实例，然后调用作为运算元的函数，并将新创建的这个对象作为这一函数调用过程的上下文。这样，当函数执行时，this引用的就是这个新对象，函数也就变成了这个对象的构造器。

这样就妥了！现在，当Something()执行时，它会在这个新的对象实例上创建三个属性，正如一个构造器所应做的那样。

很不错，现在我们知道如何声明构造器并使用它们来初始化新创建的对象了。但是除了将对象的初始化过程封装起来之外，我们还有什么收获呢？

不要忽视这种封装所带给我们的好处——代码清单3-1中的示例已经证明了这一点——不过为了让我们的对象变得真正有用，仅仅一个构造器是不够的，它还需要有方法（method），让它能以一种面向对象的方式运作。

之前我们已经看到，将对象属性设为函数就可以创建方法，下面我们就研究一下如何使用这一机制来进一步定义我们的对象。

2. 添加方法

我们继续本章前面的CD示例，为其定义一个构造器，如代码清单3-4所示。

代码清单3-4 CD对象的构造器

```
function CD(title,artist,location) {
  this.title = title;
  this.artist = artist;
  this.location = location;
}
```

假设我们要添加一个方法，这个方法会报告CD所在的位置。记住，函数要变成对象方法，必须通过对象的属性来引用，这样在调用时，它的函数上下文就会变成那个对象实例。所以我们可以这样：

```
var aCD = new CD('Afterburner','ZZ Top',17);
aCD.whereIsIt = function() {
  alert( 'The CD is on shelf ' + this.location);
};
```

但是，这并不能真正解决问题！如果我们有很多CD（如果只有一张，我们何必劳师动众添加方法呢？），就得为每个实例都执行一遍，比如像下面这样（附了一个数组来保存所有实例）。

```
var myCDs = new Array();
var aCD = new CD('Afterburner','ZZ Top',17);
aCD.whereIsIt = function() {
  alert( 'The CD is on shelf ' + this.location);
}
myCDs.push( aCD );
aCD = new CD('Mirage','Fleetwood Mac',7);
aCD.whereIsIt = function() {
  alert( 'The CD is on shelf ' + this.location);
}
myCDs.push( aCD );
aCD = new CD(''Please,'Pet Shop Boys',23);
aCD.whereIsIt = function() {
  alert( 'The CD is on shelf ' + this.location);
}
```

很无聊，而且还乱糟糟的是吧？

当然，我们可以把相同的代码提取为单一的函数，并将每个实例的属性设为这个函数。然而，这样虽然减少了打字错误的可能，但是并不能提高伸缩性。因此，正如将建立属性的代码提取到构造器中，我们也希望把方法的创建过程封装起来。

一种方式是将方法创建过程置于构造器中，如代码清单3-5所示。

代码清单3-5　CD对象的构造器

```
function CD(title,artist,location) {
  this.title = title;
  this.artist = artist;
  this.location = location;
  this.whereIsIt = function() {
    alert( 'The CD is on shelf ' + this.location);
  };
}
```

这达到了封装方法创建过程的目的，但是有一个缺点，所构造的每个对象都会有一个函数拷贝。这有一点点浪费，因为不像title属性是每个实例都各自不同，所有CD实例的whereIsIt()函数都是相同的。

如果有办法能让类的所有实例都引用一个函数实例作为它们的方法，就完美了。这正是JavaScript构造器的prototype机制的用武之地。

让我们一探究竟吧。

3. 定义原型

当用new运算符创建出对象实例时，对象初始时是空的，也就是说它不包含任何属性。然而，后面其实还有一些过程。

当我们引用一个对象属性时，JavaScript解释器会在这个对象实例中按照我们所引用的属性名来查找这个属性。这也正是我们所期望的。

不过你可能还不知道，它并不仅止于此。如果在对象本身找不到这样的属性，解释器会查看对象的构造器，从而进行最后的尝试。构造器本身有一个属性叫做prototype，解释器最后查找的正是prototype所引用的对象。如果构造器的prototype包含一个符合我们所引用的名字的属性，就会返回这个属性，而不是最终返回undefined。

对于共享定义来说，这个机制惊人地有效，一个对象类型的多个实例可以共享属性和方法的定义，而不必把值复制到每个实例。

就这点而言，你可能觉得这听上去与C++和Java等语言中类一级（class-level）的声明[①]非常相似，从较高的层面上看，你这样想是没错的。不过，如果深挖下去，这种相似性很快就不复存在了，所以你可能最好把原型（prototyping）机制看成一种求取默认值（defaulting）的机制，而不是类一级的声明。

将我们的方法作为CD对象的prototype方法来定义是很简单的，如代码清单3-6所示。

代码清单3-6 创建闭包

```
function CD(title,artist,location) {
  this.title = title;
  this.artist = artist;
  this.location = location;
}

CD.prototype.whereIsIt = function() {
  alert( 'The CD is on shelf ' + this.location);
}
```

现在，创建出的任何一个CD实例都会自动“继承”构造器的prototype中所定义的所有属性。注意，“继承”是带引号的，因为这并非真正的面向对象概念中的继承机制，这只是一种求取默认值的机制。

除了这一区别，原型的用途也不只限于方法。任何属性，如果你要定义默认值，都可以在prototype中指定。例如考虑下面这句代码：

```
CD.protoype.LOCATION_PREFIX = 'The CD is on shelf ';
```

这实际上创建了一个“类常量”，可以在CD实例的任何方法中引用，包括whereIsIt()方法：

```
CD.prototype.whereIsIt = function() {
  alert( this.LOCATION_PREFIX + this.location);
}
```

① 比如类或接口声明。——译者注

如果一个特殊的实例希望用其他前缀（prefix），它可以定义一个具有相同名字的属性。因为解释器总是会先在对象的实例中寻找，所以任何这样的声明都会覆盖prototype中的声明。需要注意的一点是：一旦在对象实例中定义了一个属性，即使再把这个属性设置成null，它仍然会覆写原型中的定义[①]。

内建的JavaScript对象也是基于prototype的，所以你可以按需要修改或扩展它们。

尽管这类操作应当谨慎从事——毕竟你在改动语言的固有类型——不过，如果你觉得内建类型缺乏某些有用的方法，那就加上一些便利方法好了，因为通常来说，这样做并不会有什么问题。

一个例子是字符串对象上的“修剪（trim）”方法。修剪字符串以剪除尾部和头部的空白字符是一项我们可能频繁调用的功能——比如，在对用户输入进行有效性验证时，可能要忽略或去掉尾部的空白字符。不知为什么，虽然这是一个非常有用的方法并且其他语言的String类大都提供了这个方法，但是JavaScript的String类却没有用于修剪字符串的方法。

那么我们来加一个吧。

我们知道，给一个类的prototype加一个方法很容易，只需在prototype上加一个函数字面量。所以给String类加入trim()方法非常简单，如代码清单3-7所示。

代码清单3-7　让String学会怎么修剪字符串

```
String.prototype.trim = function() {
  var matches = this.match(/^[ \t\n\r]+/);
  var prefixLength = (matches == null) ? 0 : matches[0].length;
  matches = this.match(/[ \t\r\n]+$/);
  var suffixLength = (matches == null) ? 0 : matches[0].length;
  return this.slice(prefixLength, this.length - suffixLength);
}
```

String类的这个新的方法使用JavaScript的正则表达式（regular expression）匹配功能来计算字符串头部和尾部的空白字符的数量，然后使用String的slice()方法来截取和返回修剪过的字符串值[②]。

注意，与其他String的方法一样，这个方法并不会修改这个String本身的值[③]，而是将修改后的值作为方法的返回结果。当对对象进行扩展时，无论它是否是内建对象，在添加新的功能时最好始终遵循该对象既有的API风格。

按照OO的术语，我们之前在自定义类或内建类如String上所创建的这些方法叫做实例方法（instance method），因为他们可以通过类的实例来访问，并且以类的实例作为它们的函数上下文。我们也可以定义另外一种方法。

---

① 原文是“it will forever override any prototype declaration”，欠妥。实际上我们可以使用delete语句来删除对象上的属性，这样就会再次使用prototype上的声明。设成null不奏效，因为null也是一个值，将一个对象的属性设成null，甚至设成undefined，并不会删除这个属性。——译者注

② 这个实现有点太过复杂了，实际上我们可以写得更简单一点，Prototype程序库中相同功能的函数叫做strip，实现是这样的：return this.replace(/^\s+/, '').replace(/\s+$/, '');。——译者注

③ 实际上在各种语言里，字符串对象通常都是所谓值对象或不可变（immutable）对象，JavaScript也是如此。所以根本没有办法去修改它的值，你只能修改变量，让其指向另一个字符串对象。——译者注

4. 创建类方法

我们已经看到，将函数添加为构造器的prototype上的属性就可以创建实例方法，JavaScript还允许我们创建其他OO语言中所称的类方法（class method）。要创建类方法，我们不是把方法加到构造器的prototype上，而是加到构造器本身。

例如，下面两个语句就大相径庭：

```
Object.prototype.sayHi = function() { alert('Hi!'); }

Object.sayHello = function() { alert('Hello!'); }
```

第一个语句创建一个名为sayHi()的方法，对于每一个Object类型的实例，都可以由实例对象访问到这个方法（所有对象其实都是Object实例，所以所有对象上都有了sayHi()方法）。第二个语句直接在Object构造器上创建了名为sayHello()的类方法。

要调用sayHi()方法，就需要一个对象的实例，这样我们就可以调用它的方法：

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

而要调用sayHello()这个类方法，我们只需要用拥有这个方法的构造器作为前缀：

```
Object.sayHello();
```

“大家快看！没有实例哟！”

如果方法的代码不需要用到函数上下文，那最好采用这种类型的方法。实际上，在其他OO语言中，类方法中完全没有this的概念，试图使用this通常会产生编译错误。不过，在JavaScript中，所有函数都有函数上下文，类方法也一样。

类方法没有对象实例作为函数上下文，那么又是哪个对象在充当这一角色呢？如果你猜是窗口对象，你可以得个鼓励奖，不过可拿不到胜利雪茄。对于类方法来说，实际是构造器函数本身充当了函数上下文。以一个函数作为函数上下文的想法可能让你感到惊讶，不过，首先请记住，在JavaScript中函数也是一种对象，其他对象能充当的角色函数也能担当。其次，就类方法的情况来说，构造器就是上下文对象这个事实实属无关紧要。所以你大可忽略JavaScript的类方法有一个this引用这件事情，如果这让你比较舒坦的话。

现在你已了解如何运用JavaScript的语言机制来创建JavaScript类和其他面向对象的构造[①]，让我们将理论付诸实践，编写一个类吧。

3.1.4 编写 JavaScript 类：一个按钮

你已了解了创建面向对象的JavaScript类所需的语言设施，下面我们更进一步，实际编写一个简单的类。

我们在本节中所要编写的这个类是用来给HTML按钮元素的实例锦上添花的。因为，虽然<button>元素是一个很乘手的HTML控件，但它在状态变化时没能给予用户足够的视觉反馈[②]。

---

① 指实例方法、类方法等。——译者注

② 这一说法有些奇怪，因为现代主流浏览器提供的<button>实现默认具有下面列出的所有视觉反馈，除了Safari中的按钮没有“待命”指示之外（因为按照Mac OS X的Aqua风格，这类按钮就是不带hover效果的）。所以，我们姑且认为原作者只是举例心切吧。——译者注

我们编写的这个类则会在发生某个特定的状态变化时变更按钮的视觉表现，比如：

- 当鼠标指针进入按钮区域时，按钮就“随时待命”。
- 当鼠标指针离开按钮区域时，则“解除待命”。
- 当鼠标按键按下时，就“按住”按钮。
- 当鼠标按键释放时，则变为“没按住”按钮。
- 当按钮启用或禁用时，也有对应反馈。

在发生上述状态变化时，改变按钮上的CSS样式类[①]（不要与OO中的类混淆起来），就可以给予用户超乎原有实现的视觉反馈。

我们就叫这个类Button好了，并且按照惯例，我们将定义该类的JavaScript文件命名为：Button.js。

下面让我们预先规划一下这个类所应具备的功能。

- 这个类会根据id识别一个已有的HTML<button>实例，并对其进行改造。
  - 这个类在构造实例的时候可带有一些选项，包括：
  - 按钮初始时是启用还是禁用；
  - 在各种状态变化时所应用的样式类名；
  - 点击按钮时所要执行的函数；
- 对于未指定的选项，构造器会提供适当的默认值。

按照这个思路，我们开始编写这个类的构造器。

1. 按钮的构造器

回想我们之前的讨论，JavaScript类的构造器其实就是一个普普通通日常可见的函数。但是，一个构造器函数如果被用作new运算符的运算元就不同了。这样的函数调用会用一个新分配的空Object实例作为上下文。

按照惯例，构造器是以其所建立的类来命名的[②]，对于我们的示例来说就是Button，构造器的职责是将对象置于初始并有效的状态。

为此，我们的按钮构造器需要接收一些信息来完成对实例的设置。我们至少得提供所要改造的按钮的id。然后根据我们前面所列出的目标，也可以传给构造器各种可选信息，包括样式类名以及点击按钮时所要执行的回调函数。

最简单的方式是直接在构造器的参数列表中列上每个可能的参数。在调用构造器时，对于我们不想包含的可选参数，我们可以传入null。

这样一个构造器的签名将采用如下形式：

```
function Button(elementName, disabled, onClickCallback,
                enabledStyleClass, disabledStyleClass,
                armedStyleClass, pressedStyleClass)
```

① 指HTML元素上的class属性。值得指出的是，所谓CSS样式类只是一个惯用的称呼。从HTML语言的用意来说，应该使用class属性来表示该元素所具有的更细化的语义，而不是纯粹的样式种类。例如<p>元素表示一个段落（paragraph），那么可以用<p class="note">表示这个段落是一段注解，而不应该用<p class="small-text">去表示段落是采用小号文字样式。——译者注

② 也就是直接以类名作为构造器函数的名字。——译者注

参数列表的第一个参数是必需的（元素名），接着是六个可选参数。

从构造器代码的角度来看，这并不难处理，只需为值为null的参数提供默认值即可。

但是从调用者的角度来看呢？

参数值与参数名是按顺序一一对应的，函数签名如果有大量的可选参数，对于函数的用户来说，就是一个不太友好的API。考虑这样一次构造器调用，调用者希望初始状态以及所有的样式类名（除了“按下（pressed）”样式之外）都使用默认值：

```
new Button('myButton',null,doSomething,null,null,null,
           'pressedButton');
```

看看那些讨厌的null。

构造器签名如果定义成这样，使用我们这个类的用户就不得不仔细数数null的个数，确保它们匹配我们那长长的参数列表中相应的参数。而且，如果我们决定扩展这个类加入更多功能，并且因此包含更多可选参数，事情会变得更加糟糕！

很明显，定义带有大量参数的函数签名——特别是其中许多参数是可选的——是一个很不友好和不可扩展的方案。为了避免这样丑陋的设计，有一种技巧迅速得到了人们的青睐，我们将用它来提供可选参数的组合，即hash[①]，有时也叫无名对象（anonymous object）。

hash就是一个对象，它被作为参数传入函数，它的属性则充当了函数的可选参数。因为是通过单个选项传入的，所以函数的参数数量就仅限于必需参数外加一个可选参数。并且因为属性都有名字，所以就不再需要按特定顺序书写可选参数，也不需要为省略的参数留下位置。

这一技巧为众多流行的JavaScript代码库所广泛采用(我们会在后面及下一章中对其中一些代码库做更深入的探索)，它帮助我们把前面那种尾大不掉、不堪卒读的构造器调用简化成：

```
new Button('myButton',
           {
             onClick: doSomething,
             pressedClassName: 'pressedButton'
           });
```

这样不仅消灭了那些null，代码也更容易阅读，因为每个可选参数都有明确的名字，不再需要记住参数的次序，不再需要数数来搞清楚哪个是哪个了。

这给构造器代码稍微增加了一点负担，不过这种负担并不大，而且如果要有负担的话，最好把负担抽取出来放入构造器中，而不是把负担丢给所有每一行要调用构造器的代码，毕竟构造器只需要写一次。

好了，行胜于言，开始编码！使用我们刚才描述的用hash作为选项表[②]的技巧，构造器的初步轮廓就应是下面这样：

```
function Button(elementName, options) {
  //TODO: fill this in!
}
```

现在我们给构造器填上代码。在列出代码清单3-8之后，我们会对代码的工作方式进行详细

① 这里的hash也就是关联数组（associative array），也有称为映射表（map）或字典（dictionary）的。之所以被称为hash，是因为它通常用散列表（hash table）这种数据结构来实现。——译者注

② 为行文方便，以下将options hash直接译为“选项表”。——译者注

描述。

代码清单3-8　Button类上的构造器

```
function Button(elementName, options) {
  this.element = document.getElementById(elementName);
  if (!this.element) throw new Error(elementName + ' not found');
  this.element.button = this;
  this.options = options || {};
  if (options) {
    this.options.enabled = options.enabled || true;
    this.options.onClick = options.onClick || function() {};
    this.options.enabledClassName =
      options.enabledClassName || this.CLASS_DEFAULT_CLASS_ENABLED;
    this.options.disabledClassName =
      options.disabledClassName || this.CLASS_DEFAULT_CLASS_DISABLED;
    this.options.armedClassName =
      options.armedClassName || this.CLASS_DEFAULT_CLASS_ARMED;
    this.options.pressedClassName =
      options.pressedClassName || this.CLASS_DEFAULT_CLASS_PRESSED;
    }
    var instance = this;
    this.element.onclick = function() {
      if (instance.options.enabled) {
        instance.options.onClick.call(instance);
      }
    }
    this.element.onmouseover = this.onArm;
    this.element.onmouseout = this.onDisarm;
    this.element.onmousedown = this.onPress;
    this.element.onmouseup = this.onRelease;
    if (this.options.enabled) {
      this.enable();
    }
    else {
      this.disable();
    }
}
```

❶ 确定HTML按钮

❷ 详细设定选项表

❸ 改造HTML按钮

❹ 改变按钮的视觉状态

❺ 将按钮置于所设定的状态

虽然这个构造器看上去有点长，但它做的事情其实相当简单——不过其中的一些新手法可能暂时有点令人困惑，因为我们在本章之前尚未见过它们。

我们在构造器中最先做的一件事情就是找到所要改造的HTML<button>元素❶。通过document.getElementById()函数，我们获得了HTML DOM中该元素的引用，并将其保存到上下文对象的名为element的属性中。记住，new运算符已经为我们创建了一个全新的Object实例并且将其设为构造器的上下文对象。我们不必操心对象的分配，当构造器被调用时，新对象已经就绪，可以通过this引用来访问了。

我们将元素的DOM引用保存于属性中，这样只要有Button对象的引用就可以顺利找到<button>元素。如果我们不能找到具有给定id的元素，就会抛出一个错误信息，显示在浏览器的JavaScript控制台（console）上。这样，对于调用者这边的错误（例如拼写错了元素的id），我

们会得到一个明确的错误信息，而不是在以后的某行代码处遇到难以理解的“对象没有属性”（或类似的莫名其妙）的错误。

我们已经可以从Button对象实例引用到所要改造的<button>元素，我们也希望能够由给定的<button>元素找到Button对象实例。为此，我们给<button>元素添加了一个名为button的属性来保存到Button实例的引用。

这个小花招意味着无论我们是有DOM元素的引用还是对象实例的引用，总可以找到对应的另一个引用，稍后它就能派上用场。

处理完elementName参数之后，让我们把注意力转到选项表上❷。我们希望在类的所有方法中都能访问这个hash对象所包含的选项信息，所以我们把它保存到实例对象的options属性中。

稍等！这个hash之所以叫做“选项”，当然是因为它是可选的！如果调用者根本没提供hash对象会怎么样呢？诚然我们可以板起脸，坚持要求调用者在不打算传入选项时也提供一个空的hash对象，但是这实在是很不友好。所以，我们会在构造器中处理这种可能性。

我们会使用下面这个语句，如果你以前没有见过这样的构造，可能会觉得有点怪异：

```
this.options = options || {};
```

对象之间的或（or）运算？那不是只用于Boolean表达式吗？

其实，这个紧凑好用的运算符实质上表示“如果第一个运算元存在，就用它，否则就用第二个。”之所以能这样用，是因为不像Java等语言，在JavaScript中，需要Boolean的地方可以使用任何表达式，不管是Boolean还是其他类型。每个运算元会按照强制转换规则转换为Boolean[①]。

对于我们这个或运算，因为null和undefined会被转换为false，所以，如果第一个运算元是null或undefined，就会对或运算表达式的第二个运算元进行求值，以此作为整个表达式的运算结果。所以上面这行代码以非常简洁的记法对名为options的属性进行了赋值，如果存在options参数，就设为它，否则就设为一个空对象。

如果选项表中没有指定enabled选项，options会指向一个空对象，否则指向一个由调用者预先装填的对象，其中包括了他所指定的选项。我们下面需要进一步给那些未指定的选项设置适当的默认值。这也是我们的目标之一，记得吧？

所以，我们运用刚才学到的“或运算”技巧来逐一检测每个可能的选项。对于每一个选项，我们检查是否已经设置过值，有，就用它；没有，就设置一个默认值。

我们检测的第一个选项很简单：

```
this.options.enabled = options.enabled || true;
```

如果选项hash中没有指定enabled选项，默认设为true。

类似地，下一个语句为onClick处理函数提供了一个默认的回调函数：

```
this.options.onClick = options.onClick || function() {};
```

这里，因为我们想不出什么行为比较合适，所以如果调用者没有提供回调，我们就默认设置一个什么都不干的函数。

---

① 精确地说，ECMAScript语言规范定义了一系列繁琐的规则来决定是否要进行转换，以及什么时候进行转换，比如或运算的第二个运算元其实就不需要进行类型转换。——译者注

其余选项如果没有指定CSS样式类名，也会设为默认的类名。你或许打算用包含类名的字符串作为默认值，这样确实也行，但是我们的做法稍有不同。

设定CSS样式类名的第一个语句是：

```
this.options.enabledClassName =
  options.enabledClassName || this.CLASS_DEFAULT_CLASS_ENABLED;
```

语句的结构和前面是一样的，但是不再是直接用各种类型的字面量（literal）作为默认值了，我们看到它引用了一个我们尚不了解的东西：this.CLASS_DEFAULT_CLASS_ENABLED。因为是通过this来引用的，所以我们知道它会是类定义的一部分。但是那到底是什么呢？

这其实是我们要在类的prototype上建立的属性，它们将包含用于定义默认样式类名的字符串字面量。根据长期以来的实践，我们不是直接把字符串字面量写死在构造器中，而是将这类字面量提取为“类成员”，这有两个目的：

- 将字符串字面量与代码隔离开来，使得程序员更容易找到这些字符串字面量；相反，如果将它们嵌于代码中就可能很难找。
- 如果类的方法需要多次引用这些字符串值，所有的引用最终都会指向一个单一的字符串实例。这样，字符串字面量中的小小打字错误就不会演变成难以找到的bug，并且在开发过程中如果要变更字面量的值，也只需要到一个地方进行修改。

遵循前述的实践惯例，我们在命名这些引用时全部用大写字母，这是C++和Java中的“类常量”的命名约定。在JavaScript中，其实没有所谓常量（constant），但是通过这一全大写的记法，我们清楚地表明，这些值是要作为常量来使用的。

在这些语句执行完成后，我们就得到了一个完整的选项属性，它合并了调用者提供的选项和类所提供的默认值。自此以后，我们有幸不必区分这两者了。通过先期装配好选项表，我们将全部选项配置都集中于一处，这样我们就能在其余代码中方便地使用这些选项配置。

接着，我们将注意力转向对HTML <button>的实际改造❸，首先从onclick事件处理器开始：

```
var instance = this;
this.element.onclick = function() {
  if (instance.options.enabled) {
    instance.options.onClick.call(instance);
  }
};
```

在上面这一小段代码中，我们指派了一个内嵌函数（inline function）[①]作为<button>元素的onclick事件处理器。

在这个函数中，我们在执行单击处理函数[②]之前需要先检查按钮是否启用。（事实上，我们可以不用检查，因为对于禁用的按钮来说，这个函数应该不会被调用，但是你知道人们常说要未雨绸缪！）

---

① 在C++领域inline function有特指含义，常译作“内联函数”（裘宗燕译为“在线函数”），但在JavaScript中并无特指，大体上等同于嵌套函数（nested function）。——译者注

② 指Button对象上的options.onClick()函数。——译者注

这里有一个小问题。当事件处理器被激活时，它的上下文对象是引起事件的那个元素，此处就是<button>元素，而不是我们的Button实例。对此，我们可以使用放在元素上的自引用[①]（在后面的方法里我们就会这样做），也可以利用函数定义时的闭包。

既然我们会在其他方法中利用自引用，那么这儿我们就用一下闭包，没有其他原因，只是演示一下闭包的用法。（在实际编程中你可能最好选择一种方式并一以贯之，不过为了演示，我们就不管那么多了。）

回想闭包的工作原理，我们指派给<button>元素的onclick处理器应该能访问在函数声明所在的作用域内定义的变量。然而this引用不会被包括在闭包中，可我们要访问的正是它！为了解决这个小问题，我们将this的值赋给一个名为instance的局部变量。因为局部变量处于函数声明时所在的作用域内，所以instance的值，也就是Button实例的引用的一个副本，作为闭包的一部分，肯定可以访问得到。

在单击事件处理得到解决之后，让我们把注意力转向其他需要捕获的鼠标事件，通过这些事件我们将变更<button>元素的视觉状态❹。与onclick不同，我们不是指派一个无名的内联函数（及其闭包）作为元素的事件处理器，而是指派了方法的引用，稍后我们将在prototype中定义这些方法。这让我们的构造器看上去更加整洁，并允许我们把大段代码切分成更小的代码块，但是我们失去了通过闭包引用Button对象的实例的能力。

```
this.element.onmouseover = this.onArm;
this.element.onmouseout = this.onDisarm;
this.element.onmousedown = this.onPress;
this.element.onmouseup = this.onRelease;
```

稍后在研究这些函数的实现时，我们会看到，之前存储在<button>元素上的自引用此时就可以发挥作用了。

最后，我们要确保<button>元素和Button实例均已置于正确的初始状态❺。根据选项表上所定义的状态，我们将从启用按钮和禁用按钮这两个方法中选择一个并进行调用。稍后我们会在代码清单3-11中看到它们的具体实现。

现在，还记得那些其实并非真正常量的“类常量”吗？下面我们就来定义它们。

2. 类一级的成员变量

在关于构造器的讨论中，我们提到了一种良好的实践，即将各种字面量（字符串字面量或其他字面量）从函数内部的代码中提取出来并作为类的成员。这样，在整个代码中可以始终一致地引用单一的字面量实例。在Button类中，我们声明了一系列字符串成员，作为在状态改变时要设到<button>元素上的各种CSS类名的默认值。这些字符串成员和默认值列于代码清单3-9中。

代码清单3-9 用于CSS样式类名的类一级成员

```
Button.prototype.CLASS_DEFAULT_CLASS_ENABLED = 'buttonEnabled';
Button.prototype.CLASS_DEFAULT_CLASS_ARMED = 'buttonArmed';
Button.prototype.CLASS_DEFAULT_CLASS_DISABLED = 'buttonDisabled';
Button.prototype.CLASS_DEFAULT_CLASS_PRESSED = 'buttonPressed';
```

① 这里的“自引用（self-reference）”是指<button>元素上的button属性，它指回到Button的实例。——译者注

通过使用全部大写的名字——这是许多语言如Java、C、C++中常量的命名约定——我们指明了这些成员应该当作常量来使用（尽管JavaScript并没有常量的概念）。而通过对字符串字面量进行集中管理，我们就确保了对成员常量的多个引用最后都指向同一个字符串字面量。

这当然远胜于将大量字符串字面量到处写死在代码里。引用成员时的打字错误会产生“未定义（undefined）”错误，这相对容易调试，而字符串字面量里的打字错误则直接导致元素不能正确工作，我们就只好打哑谜，直到辨认出字面量里的拼写失误。就这点来说，类一级的成员既可以用于需要在多个Button实例中共享的读写型（read-write）变量，也可以用于像我们这里定义的只读型（read-only）伪常量。

下面我们将面对的是为<button>元素的鼠标事件所指派的处理器函数。让我们迎难而上吧！

3. 鼠标事件处理器

要在鼠标事件发生时变更<button>元素的视觉状态，简单地切换不同的CSS样式类就可以了。Button类的用户可以在选项中设定样式类名，也可以使用我们所提供的默认类名。

你可能记得，我们将鼠标事件处理器设置成了对Button类的方法的引用。举例来说，当鼠标指针移动到按钮区域时，我们希望呈现与待命状态相关联的样式类。

在构造器中我们是这样指派这个事件处理器的：

```
this.element.onmouseover = this.onArm;
```

现在，我们来定义该方法（及其同类方法），如代码清单3-10所示。

代码清单3-10　鼠标事件处理

```
Button.prototype.onArm = function() {
  if (this.button.options.enabled) {
    this.className = this.button.options.armedClassName;
  }
}

Button.prototype.onDisarm = function() {
  if (this.button.options.enabled) {
    this.className = this.button.options.enabledClassName;
  }
}

Button.prototype.onPress = function() {
  if (this.button.options.enabled) {
    this.className = this.button.options.pressedClassName;
  }
}

Button.prototype.onRelease = function() {
  if (this.button.options.enabled) {
      this.className = this.button.options.enabledClassName;
    }
  }
```

注意，尽管这些事件处理器本是Button的方法，但是当它们被触发时，其上下文对象将是触发它们的元素——对于这里来说就是<button>。然而，我们需要访问Button实例以便查找存于

实例中的选项。

我们可以通过定义一个内联方法并使用闭包来解决这个问题（之前onclick处理器就是如此），不过在构造实例时我们已经深谋远虑地将Button实例的引用置于<button>元素的一个属性之中（我们很有创意地命名为button），所以我们已经有所需的引用了。于是，在每个事件处理器中，我们首先检查enabled选项，确定按钮是否是“活的”，如果是，我们就把适当的样式类应用到元素的className属性。

定义这个类的最后一个步骤是让调用者可以控制按钮状态是启用还是禁用。

4. 按钮状态相关的方法

用户能够将按钮初始置于启用或者禁用状态。然而，如果一个被禁用的按钮要是不能在某个时候被启用，那它就永远派不上用场了，所以我们需要给我们的类添加一些方法，以便让调用者在实例构造完成之后也能对状态进行控制。

为此我们要定义三个方法：

- 允许调用者获得当前状态。
- 允许调用者将按钮置于启用状态。
- 允许调用者将按钮置于禁用状态。

这些方法的具体实现列于代码清单3-11中。

代码清单3-11 状态相关方法的具体实现

```
Button.prototype.isEnabled = function() {
  return this.options.enabled;
}

                                          ❶ 将按钮置于可用状态
Button.prototype.enable = function() {
  this.options.enabled = true;
  this.element.disabled = false;
  this.element.className = this.options.enabledClassName;
}

    Button.prototype.disable = function() {      ❷ 禁用按钮
      this.options.enabled = false;
      this.element.disabled = true;
      this.element.className = this.options.disabledClassName;
    }
```

这些方法比前面那几个方法要稍微简单一点。因为它们只是类的方法（而不会被当作事件处理器来使用），所以我们不必去担心闭包或事件的上下文对象，它们的上下文对象就是Button实例，正如我们所期望的一样。

isEnabled()方法返回enabled选项属性的值，这样，类的用户就能检测到按钮的当前状态。在其余方法中，我们会小心确保该属性始终如实反映按钮元素的当前状态。

让我们实际考察一下将按钮置于启用状态的那个方法❶。它首先将enabled选项属性设为true。然后通过将<button>元素的disabled属性设为false（所有HTML表单元素都使用这种逆向逻辑设置来确定控件的启用状态）启用了<button>元素本身。最后，元素上的CSS样式类设为了按钮启用状态所对应的样式类名。

禁用按钮的方法❷遵循同样的步骤，只是所设的值不同。

有了上述这些方法，我们的Button类就完工了。至少目前我们还没有想到要加什么其他东西！

在创建这个小小的类的过程中，我们运用了许多前几节介绍的OO技术：对象属性、上下文对象、闭包，还有prototype属性。用这些知识武装起来之后，我们完全有能力定义任何其他的对象类，并通过它们更好地组织代码。

不过在筹划未来之前，我们需要先对我们的代码进行测试，以确保它能够运行！

5. 测试Button类

为了测试Button类，我们将创建一个简单的HTML页面来实际检验一下类的功能。如果一切顺利，在测试页面中点击改造后的按钮，我们就会看到提示，如图3-4所示，显示出所点击的按钮元素的ID。代码清单3-12列出了用于测试的代码。

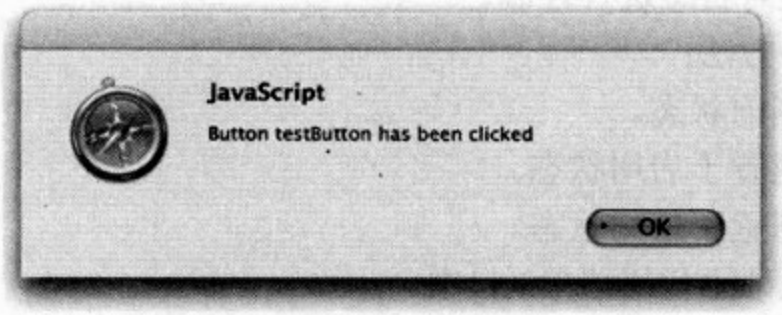


图3-4　得到我们所希望的按钮了没有？

代码清单3-12　对Button类进行测试

```
<html>
  <head>
    <title>Button Test</title>
    <script type="text/javascript" src="Button.js"> </script>
    <script type="text/javascript">
      window.onload = function() {
        window.testButton = new Button(
          'testButton',
          {
           onClick: onClicked
          });
      };

      function onClicked() {
        alert('Button ' + this.element.id + ' has been clicked');
      }

      function toggleButtonState() {
        if (window.testButton.isEnabled()) {
          window.testButton.disable();
        }
        else {
          window.testButton.enable();
        }
      }
    </script>
    <style type="text/css">
```

❶ 引用Button.js文件

❷ 创建按钮实例

❸ 触发按钮

❹ 检测按钮状态

❺ 设置视觉样式

```
#testButton {
  padding: 3px 6px;
  font-size: 1.1em;
  border-width: 3px;
}
.buttonEnabled {
  background-color: maroon;
  border-color: maroon;
  color: white;
  border-style: outset;
}
  .buttonDisabled {
    background-color: #999999;
    border-color: #999999;
    color: white;
    border-style: outset;
  }
  .buttonArmed {
    background-color: maroon;
    border-color: maroon;
    color: orange;
    border-style: outset;
  }
  .buttonPressed {
    background-color: #660000;
    border-color: maroon;
    color: orange;
    border-style: inset;
  }
  </style>
</head>

<body>

  <button type="button" id="testButton">Click me!</button>

  <div style="margin-top:16px">
    <input type="checkbox" onclick="toggleButtonState();">
    Disable button
  </div>

</body>

</html>
```

❻ 定义<button>元素

❼ 触发“启用/禁用”函数

注意，这个页面不可能构成对这个类的完整测试。作为练习，你可以考虑以它为样板，实施一系列更全面的测试。虽然只是样板，但是它代表了我们所预想的Button类最普遍的使用方式。

在这个页面中，我们引用了一个.js文件来导入Button类❶。这样Button构造器及其原型就被引入了页面的作用域中。注意，每个需要使用Button类的页面我们都必须先导入它。

在页面的onload事件处理器中❷，我们创建了Button的一个实例，并将其保存于窗口对象的名为testButton的属性中。这样，整个页面的其余部分也都能访问这个Button实例。假如我们直

接使用var，其作用域就仅限于onload处理器。我们也可以在事件处理器之外声明一个以window为作用域的var，然后在事件处理器内进行赋值。不过，既然一个语句就可以做到，为什么还要把声明分在两处？

在对Button的构造器的调用中，我们指定了id字符串"testButton"，它标识了在页面body中所定义的一个普通的<button>元素❻。

这个测试中，我们所提供的唯一选项就是一个名为onClicked的函数，当点击按钮时就会触发它。

一切都很简单，而这正是我们想要的。如果用户想更周到一些，可以指定其他的选项。不过这个示例或许代表了大多数Button实例创建时的情形，你可以看到这对于我们的类的用户来说易如反掌。

我们的测试页面中的下一个主角是onClicked()函数❸，用户点击<button>元素时它就会被触发。在这个函数中——真实环境或许会做一些更有意义的事情——我们发出一个提示，证实函数的上下文已被设为了正确的对象。

引用this.element.id证明了这个Button实例就是上下文对象。this引用应该会指向这个Button实例，其element属性包含了<button>元素的引用。

上面测试了点击按钮的功能，我们也想操练一下类的状态相关的函数。所以最后我们写了一个函数❹来检测按钮所处的状态，使用了我们先前提供的便捷方法，并将按钮置于相反的状态。稍后我们会看到如何从一个页面的元素触发该函数。

这个页面主要是为了展示能给予我们的按钮所需要的视觉反馈的CSS样式❺——归根到底，那正是Button类的意义所在。这些样式让按钮在启用状态但是未激活时具有一个outset的外观；在禁用时具有“灰暗”外观，当进入待命时具有高亮外观（将文本改为橘红色）；当按下时则具有inset外观。这些状态如图3-5所示，尽管在静态截屏中（特别是转换为灰阶图像之后）不是很令人兴奋，但至少你有概念了。

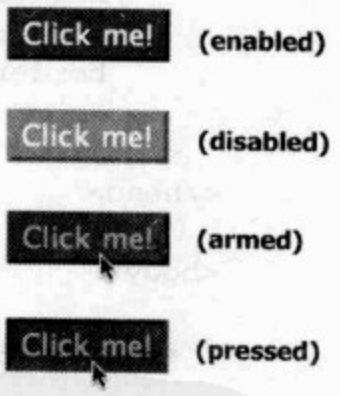


图3-5　按钮的视觉状态

在HTML页的主体中，我们定义了两个控件元素，首先是一个将要给被扩充的按钮❻。注意，只需要设置按钮的id属性。剩余的过程，包括处理函数和类名等，都是有Button类提供的。这是多么轻松啊。

第二个控件元素是一个复选框❼，它用来触发先前定义的“激活/失效”函数❹。

很明显，接下去应该做的一系列测试就是检测是否可以正确地用自己给定的样式来覆盖默认的CSS类名，以及可以正确地把按钮初始化为不可用状态。

负面的测试也应该一起进行，这样可以保证在传递错误的值时，类的行为还是合理的。作为一个额外的练习，你需要考虑如何使类更加健壮。例如，为了简短，我们还没有检查过传递过去的元素是否是一个真正按钮的情况。那么你还可以填补其他哪些漏洞呢。

我们已经了解了怎样在JavaScript中运用面向对象准则来更好地组织代码，下面让我们看看一个流行的JavaScript库，它能让上述过程变得更加简单。

3.2　Prototype 程序库

在本章的第一部分中，你已经看到了如何用语言的内建设施来扩展JavaScript。JavaScript基

于原型的继承特性使我们不仅能创建和扩展我们自己的类，而且还可以改进语言内建的类。

程序库作者当然不会忽略这样的特性，他们利用这些特性给JavaScript提供了许多有用的扩展。在本节中，我们就来研究这样一个程序库，它在Web开发社区中已经非常流行了，这就是Prototype(它正是根据JavaScript的原型机制来命名的)。

Prototype为JavaScript提供了大量极其有用的扩展和补充，它使Web应用开发者，尤其是那些需要编写大量DHTML(例如Ajax应用)的开发者轻松了许多。甚至，JavaScript创作者可以用到的某些其他更高层面的流行程序库[①]本身就是建于Prototype库之上的。

在本节中，我们没有足够的时间和篇幅来涵盖Prototype提供的所有特性。事实上，那可以成为单独一本书的主题。所以我们将集中于一些最有用的特性，然后把重点放在那些能帮助我们使用面向对象方法有效组织代码的特性上。在下一章中，我们会重访Prototype，届时主要会关注Prototype库的Ajax特性。

要使用Prototype，唯一所需的就是一个`prototype.js`文件，它可以从Prototype的站点http://prototype.conio.net上下载到。把它导入你的页面中，你就可以开始使用它了！

让我们先来看一下Prototype的一些便捷特性，几乎所有JavaScript代码都可能用得上它们。

3.2.1 常用的函数和扩展

尽管我们这里的主要目的是关注Prototype的对象创建和操作机制，但有一些Prototype的一般特性非常有用，而且非常漂亮，值得我们首先研究一下。

我们先来看一个非常方便的助手函数，一旦你习惯了它，就再也无法想象没有它的日子了。

1. 获取DOM元素的引用

无论我们的目的是查看一下元素上的属性，以某种方式添加或修改元素的属性值；抑或是像本章第一部分中那样对元素进行强化改造，都会频繁地在页面中获取DOM元素的引用。

这个过程通常需要根据给定的id值来查找DOM元素，所以我们会经常写这样的代码：

```
document.getElementbyId("someElementId")
```

这并不难掌握，但是相当罗嗦。Prototype为这个非常常用的操作定义了一个缩略形式的函数——`$()`函数。(没错，函数的名字就是一个美元符号。)

使用这个函数，通过id来获得元素引用就变成了：

```
$("someElementId")
```

这比浏览器原生的版本要简洁得多。必须承认，第一次用这个函数来编写代码，我们可能需要一个适应过程。但是只要你稍稍使用一段时间，就会对这种初看有点奇异的记法安之若素了，而且你会发现，在没法使用Prototype时，你将会是多么想念它。

与原生版本相比，`$()`函数不仅记法紧凑，还有一个有意为之的细微差异：如果传给函数的值已经是一个DOM引用(而不是一个id字符串)，那么会直接返回该引用。这样，我们在使用`$()`时既可以传入一个id字符串，也可以传入一个DOM引用。实际上，我们根本不必去操心我们所持有的变量中所保存的到底是一个DOM引用还是一个id字符串。这听上去好像也没什么大不了

① 如script.aculo.us(http://script.aculo.us/)，它是一个基于Prototype的UI库。——译者注

的，但无论是编写类还是编写一般的JavaScript代码，它都能给我们带来极大的便利。

除了获取元素引用，Prototype还提供了其他的便捷方法。

2. 读取表单控件的值

Prototype提供的另一个便捷函数是$F()，将表单控件元素的id作为参数传入，$F()就会返回该控件元素的值。

比如说，有一个文本框控件id为someTextControl，那么就可以这样获取它的值：

```
$F("someTextControl")
```

需要特别注意的是，这个函数通过id来定位控件元素，而并不是通过它们的名字。表单控件元素既有id属性也有name属性，如果你不理解它们的差别，就很容易被它们搞糊涂。考虑下面这个控件元素：

```
<input type="text" id="someId" name="someName" value="whatever"/>
```

在JavaScript代码中，可以通过两种方式来引用这个元素——通过name（假设这个元素处于一个名为someForm的表单中）：

```
document.someForm.someName
```

也可以通过id:

```
document.getElementById("someId")
```

这两个属性的不同在于：id属性给HTML DOM中的元素指定一个标识符，这个标识符在整个页面中必须是唯一的；而name属性则指定了与控件相联系的请求参数名，请求参数是表单提交时所创建的请求的一部分，多个控件可共用同一个请求参数名。

id属性适用于HTML DOM中的所有元素，而name属性仅仅适用于控件元素，控件元素所具有的值在表单提交过程中将成为请求的参数。

请记住，$F()函数需要的是元素的id，而不是元素的name。此外，按照典型的Prototype风格，$F()函数也可以接受一个DOM控件元素的引用而不是id。

3.2.2 对数组的扩展

很难想象程序可以没有数组。尽管也有其他的构造（construct）可用于数据的有序列表，但对于大多数编程语言来说，数组仍是使用最普遍的，而且通常是语言本身固有的构造。

JavaScript也不例外。Array类提供了许多有用的方法，可以处理页面数据的重复元素。但是，正如人们所说，没有最好，只有更好。

Prototype扩展了JavaScript的Array类，提供了一些很棒的特性，在应用中我们会发现它们非常有用。不过我们这里所看的也只是冰山一角，你可以亲自深入探索Prototype的代码或是互联网上的资源来了解更多的细节。

1. $A()函数

从技术上来说它并不是Array对象的扩展，但Prototype的$A()函数可以把其他构造转换成Array类的实例，这很有用。特别是它能把一个XML文档的NodeList转换成数组，这对Ajax程序员来说尤其有用。因为我们不仅可以轻松的遍历它，而且还可以利用我们即将探索的那些

Prototype数组扩展来操作它。

例如，我们假设有一个名为xmlDoc的变量引用了一个XML文档：

```
var arrayOfNodes = $A(xmlDoc.getElementsByTagName('xyz'));
```

上面的代码会创建一个数组，包含XML文档中所有的<xyz>元素，并将这个数组赋给arrayOfNodes。

现在让我们来看看Prototype是怎样让JavaScript的Array类超越其原有定义，变成一个更加有用的构造的。

2. Enumerable类及其方法

Prototype库对Array类本身进行了增强，增加了一些有趣的方法，比如shift()（允许你将一个数组当成堆栈来使用）[①]以及compact()（返回一个原有数组的副本，但移除了其中所有未定义项[②]）。但是，随着本书示例的展开，我们会发现，真正令我们感兴趣的是Prototype用Prototype所定义的名为Enumerable的类对Array进行了“扩展”。

这里，我们并不是在严格意义上使用“扩展（extension）”这一术语，我们知道JavaScript并没有真正的面向对象的基于类的继承和扩展，但是Prototype让我们能极好地模仿这种能力。（我们将在3.2.5节中对它进行讨论。）

Enumerable类具有许多特别有用的方法，用来对Enumerable实例（也包括所有的数组）中的元素进行迭代[③]，其中最简单的就是each()方法。

each()方法接受一个函数的引用作为参数，它会按照顺序为Enumerable实例中的每个元素都调用一次所传入的函数。这一用于迭代的函数应该符合如下接口：

```
function iteratorFunction(element,index);
```

其中，element是数组迭代过程中的当前元素，而index是该元素在数组中的索引。代码清单3-13给出了一个简短的示例。

代码清单3-13 在一个Array对象上使用Enumerable.each()方法

```
var myArray = [ 1, 2, 3, 4 ];
myArray.each(showMe);

function showMe(element,index) {
  document.write('<p>[' + index + '] ' + element + '</p>');
}
```

迭代myArray的每个元素

会为每个元素调用

这个简单示例可能无法体现出each方法的实用价值——直接使用一个for循环也一样方便，甚至可能更清晰——然而，当用于更加复杂的任务时，它的效用就会显现出来。

将一个算法中的复杂代码提取出来纳入一个单独的函数中，所获收益不可低估——特别是函

① 原作者有误，shift()方法是Array类本来就有的方法。只是较老的JavaScript引擎，如JScript 5.6之前版本可能没有这个方法或实现上有缺陷。Prototype 1.5.0开始去掉了对shift()方法的不必要的扩展。此外，严格地说，shift()方法对应的是队列而非堆栈。——译者注

② 实际上是筛除了所有值等于null（包括undefined）的项。——译者注

③ 迭代（iteration）就是指遍历每个元素。——译者注

数如果具有良好命名的话。提取之后，复杂的代码就变得清晰有序，因此花时间来正确构造这样的函数是完全值得的。

在后面的章节中，我们会看到这一模式是如何帮助我们让各个函数更加简短和易于把握的。这一模式也减少了处理数组元素时寻址元素所需的符号数量。

Enumerable类上还有许多这样的迭代函数，利用它们，我们能以一种可定制的方式遍历整个数组，还可以轻松根据各种条件筛选数组元素来得到一个新的数组。我们强烈建议你探索一下这些函数。

数组，如前所述，是用于存储有序列表数据的基本编程元素。我们在本章中还研究过另一种语言构造hash，下面就让我们看一看Prototype为它所提供的正式化版本：Hash类。

3.2.3　Hash 类

在本章第一部分中我们了解到，对象的属性可以动态地创建和赋值，因此我们可以把Object类的实例看成一种特别的“关联数组（associative array）”，类似于其他语言中的Map或Hash。

Prototype库更进一步，它正式定义了一个Hash类，不仅提供了键值关联，而且还拥有诸如keys()、values()、merge()等有用的方法。它还定义了一个在Ajax代码中极其有用的方法：toQueryString()方法。

这个方法对hash所包含的名值对进行格式化，形成对应的HTTP查询字符串。它可以与$H函数配合使用，$H()函数可根据任意一个JavaScript对象的属性和值创建一个Hash类的实例。在稍后的实例中，我们会使用toQueryString()来生成Ajax请求的目标URL，这样既方便，又避免了自行构造URL的出错可能。

代码清单3-14是一个使用这一方法的小示例，它会输出这样一个结果：

```
a=1&b=2&c=3
```

蛮不错的！更棒的是，这个方法会对查询字符串参数的名称或值中包含的保留字符进行必要的URL编码。

代码清单3-14　使用Hash.toQueryString()方法

```
var o = new Object();
o.a = 1;
o.b = 2;
o.c = 3;
document.write($H(o).toQueryString());
```

创建对象，给属性赋值

格式化并输出查询字符串

如果示例用到更复杂的值和特殊符号，可能会得到这样的输出：

```
param%201=1%261&param%202=2%3D2&param%203=3%2B3
```

对应的代码是：

```
var queryString = $H(
  {
    'param 1': '1&1',
    'param 2': '2=2',
```

```
      'param 3': '3+3'
    }
).toQueryString();
document.write(queryString);
```

注意，在这个示例中，每个参数名都包含一个空格字符，所以需要进行编码，而参数值同样包含需要被编码的字符（&、=和+）。

在输出结果中，这些特殊字符都已经被自动替换成了对应的编码。这样贴心的设计想必你一定挺满意的吧？

3.2.4 给函数绑定上下文对象

回想我们在3.1.2节中对函数的讨论，函数的上下文对象就是函数执行时this所引用的对象。通常，这个对象可能是当前页面的window对象（对于所有顶层函数来说就是如此），或者是函数所属的对象实例——如果函数作为对象上的方法被调用。

但是，如果对象方法是作为回调函数在发生事件时（比如一次鼠标点击）被调用，this所指向的将是产生事件的元素。在这种情况下，我们要使用一些JavaScript技法来确保我们可以访问到方法所属的对象实例。

我们可以采用的一种手段是，使用Function类的call()方法来调用函数而不是直接调用。call()方法需要指定的第一个参数就是函数的上下文对象。

举例来说，如果我们希望所调用方法的this变量指向某个对象，我们就可以使用函数的call()方法，并把要变成函数中的this的对象作为第一个参数：

```
var a = someFunction.call(someObject);
```

通过使用call()方法，在someFunction函数内部，this引用就会指向someObject。

如果是由我们来调用函数，这样当然是可以的。但是，如果并非由我们来控制函数的调用时机，例如，当一个函数引用被传给另一个对象作为回调函数使用时，我们就无法用call()方法了。在Ajax程序中肯定会遇到这种场景，比如我们需要将函数引用传给XMLHttpRequest或其他程序库的代码作为回调通知。

我们无法再控制这些函数被怎样调用，然而，我们需要把this变量绑定到一个特定的对象，而不是通常所绑定的那个对象。

为此，Prototype库对Function类进行了扩展，加入了一个bind()方法，我们可以用它将一个对象预先绑定到一个函数引用，这样之后调用函数时它的this变量就会指向那个对象。

一份代码胜过千言万语，来看看代码清单3-15的例子：

代码清单3-15 把一个Object实例预先绑定到一个函数上

```
window.x = 1;                          ❶ 给window对象做个标记
var o = { x: 2 };                      ❷ 任意创建一个对象，并做上标记

function doSomething(callback) {       ❸ 定义未经绑定的回调
  callback();
}
```

```
function callback() {    ◁—❹ 定义绑定后的回调
  alert(this.x);
}

doSomething(callback);
doSomething(callback.bind(o));
```

在这个示例中，我们给window对象打上一个x属性作为标记❶，然后任意创建一个对象，也用x属性做了一个标记，但值不同❷。这样，我们就可以轻松识别引用所指向的是哪个对象。

然后，我们定义了一个名为doSomething()的处理函数，它的唯一参数是一个回调函数的引用❸。为了简明起见，这个函数除了调用所传入的回调函数之外并没有做任何其他事情，但是在现实世界的实例中，一个库函数可能会先进行一些处理，完成之后再调用回调函数。

回调函数本身❹只是显示一个提示，告知我们this引用的那个对象的x属性的值。

为了说明未经绑定的函数和绑定后的函数之间的差别，我们首先调用doSometing()函数，传入回调函数的普通引用。当显示提示时，我们会看到值是1，这表明在回调函数调用时，this变量指向的是window对象。

然后我们再次调用doSomething()，这次我们使用Prototype所提供的bind()扩展将对象o绑定到回调函数上。当出现提示时，我们会看到值是2，这表明这次this引用的是对象o。

当把处理函数传给XHR的方法或其他库函数时，这种预定义绑定会变得极其有用，它让我们能够控制处理函数被调用时的上下文对象（this所引用的对象）。

如果回调函数是用于响应DOM事件（比如一个鼠标点击）并需要访问触发函数的事件，可以使用Prototype所提供的一个名为bindAsEventListener()的方法。这个方法与bind()类似，其参数对象也将被用作回调函数的上下文。而且当调用回调函数时，事件对象会被作为参数传入回调函数。在后面的几章中，我们会看到一些例子使用了这个方法。

在新学习了Prototype库的所有这些实用知识之后，让我们来看看Prototype是如何帮助我们编写面向对象的JavaScript的。

3.2.5　面向对象的 Prototype

在本章的第一部分中我们已经看到了，在JavaScript中应用面向对象的概念能够帮助我们更好地组织代码并提高重用性。在这一节中，我们将研究Prototype怎样协助我们编写面向对象的JavaScript类，从而让这一过程变得更加轻松。

我们首先来看看Prototype提供的名为Class的类。

1．用Prototype创建类

Prototype的Class类[①]是一个生成对象构造器的快捷工具。与之前构造JavaScript类的原始方式相比，它的记法更简洁，而且可能也更一致。

你可能还记得3.1.4节中的Button示例，一个JavaScript类的定义包括一个构造函数，然后是一段声明语句，为构造器的prototype属性设置成员变量和方法。有些人认为这样的记法有点不太一致，初始化代码要放入一个用作构造器的普通函数声明中，而方法代码却要置于prototype上所

① 注意，Prototype 1.6.0会对Class进行大幅改造，引入新的继承机制。——译者注

设的函数中。

对于这些希望将代码以更一致的方式聚集到一处的人来说，Class的create()方法（也是它的唯一方法[①]）就很合胃口，它生成一个构造器，然后构造器的功能可以放到prototype中声明。当构造器被调用时，它需要类中定义有一个名为initialize()的方法并把控制权移交给它。让我们来看看代码清单3-16中的示例。

代码清单3-16 用Prototype的方式构造一个类

```
Something = Class.create();

Something.prototype.initialize = function(p1,p2,p3) {
  /* constructor code goes here */
};

Something.prototype.someMethod = function() {
  /* method code goes here */
};
```

在代码清单3-16中，我们看到构造对象实例的代码委托给了一个名为initialize()的方法。它可以接受任意个参数，并且正像其他方法一样，它是在类的prototype中声明的。许多开发者更喜欢以一致的方式在prototype中声明所有代码。

为了使代码看起来更加紧凑，许多开发者还用JSON记法来聚集prototype属性，如代码清单3-17所示。

代码清单3-17 用JSON的方式来聚合属性

```
Something = Class.create();

Something.prototype = {

  initialize: function(p1,p2,p3) {
    /* constructor code goes here */
  },

  someMethod: function() {
    /* method code goes here */
  }

}
```

尽管记法有所不同，但代码清单3-17列出的代码完全等价于代码清单3-16中的代码。许多开发者更喜欢以这种方式给类的prototype设置全部的成员变量和方法。

使用哪种记法来声明JavaScript类取决于你的个人偏好。不管你如何声明类——使用原始的JavaScript，借助Prototype的Class，用或不用JSON符号——都可以使用其余的Prototype对象扩展。并且不管哪一种声明方式，都是按照相同的方式用new运算符来创建类的实例的：

```
var anInstance = new Something(1,2,3);
```

① 从Prototype 1.6.0开始，Class类上将增加一些方法，实现新的继承机制。——译者注

Prototype有许多扩展，让我们在对对象做各种处理时游刃有余。下面让我们快速浏览其中一些扩展。

2. 用Prototype合并对象

不知你是否注意到，我们在本章的第一部分已经接触过了一个合并对象的实例。但是，在涉及它之前，让我们先看看Prototype是如何合并对象的，合并（merge）的真正含义又是什么。

在Prototype中，对象合并这一概念本质上是对两个对象中的所有属性进行合并。这是由Object构造器上的名为extend()的类方法所实现的。

奇怪，为什么不将其命名为merge()呢？这个原因过一会儿就清楚了。

extend()方法是具破坏性的，作为合并的结果，所传入的两个参数之一会被修改。这个方法的签名是：

```
Object.extend(object1,object2)
```

extend()会把在object2中找到的所有属性都拷贝到object1中。结果是，object1既包含了初始就有的属性，也获得了object2中的所有属性。

如果两个对象拥有相同名字的属性，那么object2属性的值将覆盖object1属性的值，即object2的属性优先。当合并完成后，extend()函数将返回object1的引用。

如果用Prototype重写我们的Button类，就会发现这个方法对处理选项hash很有用，在3.2.6节中我们会展示这一改进。除了合并对象实例，extend()方法还有一个更基本的用途——这也是它为什么叫这个名字的原因。

3. 用Prototype扩展类

在Java和C++这类面向对象语言中，可以通过继承来创建类的层级结构，子类会继承父类的成员和方法。JavaScript并不具备这样的继承能力，但是Object.extend()类方法提供给我们一种与继承相当类似的机制。

记住，JavaScript类是由构造器及其prototype中的属性所组成的。虽然我们不能令一个JavaScript类魔法般地从其他类继承些什么，但是，如果我们用Object.extend()方法将父类对象合并到子类的prototype对象中，构成一个新的由这两个对象组合而成的prototype，会怎么样呢？

晕头转向了？

让我们通过一个示例来检验一下我们的想法吧。还记得本章开始时的那个CD示例吗？我们创建了一个小型对象来保存我们收藏的某个CD的信息。我们（通过属性）记录了CD的标题、歌手，以及碟片存放的位置（架子的编号）。

不过，实际上，我们不仅是音乐发烧友，而且还是电影发烧友！所以我们希望示例也能包括DVD。这个想法令我们着迷。

我们知道，CD和DVD有许多共同点，但是它们也各自拥有一些独特之处。CD的“歌手”概念对DVD来说没有什么意义，对于DVD，我们也许想记录电影的导演，而这对CD来说也是没有意义的。但是，它们都有标题以及收藏位置。

下面，我们首先创建一个能描述两种碟片的共有特征的类。使用Prototype，我们的代码如代

码清单3-18所示。

代码清单3-18 定义Disc“父类”

```
Disc = Class.create();                          ❶ 创建Prototype风格的构造器

Disc.prototype = {

  initialize: function(title,location,type) {   ❷ 定义用于构造器的initialize方法
    this._initializeDisc(title,location,type);
  },

  _initializeDisc: function(title,location,type) {
    this.title = title;
    this.location = location;
    this.type = type;
  },

  whereIsIt: function() {
    return 'The ' + this.type + ' titled ' + this.title +
           ' is on shelf ' + this.location;
  }

}
```

像之前所看到的那样，我们首先创建构造器（以Prototype的方式）❶，然后在prototype中定义成员。因为我们使用了Prototype库的Class.create()机制，所以需要定义一个initialize()方法❷，构造器会在构造对象时调用它来对实例进行初始化设置。在这个方法中我们做了一些有点奇怪的事情。

initialize()方法仅仅只是转而调用另一个叫做_initializeDisc()的方法，让它来设置实例。为什么要这么做呢？

首先，_initializeDisc()方法的名字以下划线开头是一种命名约定，表示这个方法是在内部使用的，外部代码在使用这个类时不应该去调用这个内部方法。JavaScript没有私有（private）或保护（protected）成员的概念，所以我们通过这种命名方式来说明我们的**意图**，即位于这个类（或整个类继承树）以外的代码应该忽略这个方法，虽然事实上我们并没有办法来**强制**这一点。

但是，究竟为什么要委托给一个内部方法呢？

因为，我们的意图是把这个类作为还未编写的CD类和DVD类的父类。而当建立CD类和DVD类时，它们都会有一个自己的initialize()方法，这会覆盖父类所定义的initialize()方法。为了确保能够从子类中执行父类的初始化过程，我们就把初始化代码提取到_initializeDisc()方法中，它不会被子类覆盖，这样子类就仍可以调用它。对于任何一个要被其他类扩展的父类，我们都需要做这样的两层初始化。

你也许会认为_initializeDisc()这个名字有点罗嗦，或者觉得在这个局部初始化器的名字中加上类名实属多余，但是子类也可能用作其他类的父类。这种情况下如果我们用_initialize()这样简单的名字，那么又会陷入到同样的问题中，也就是，这个方法可能会被继承链中的其他类给覆盖掉。通过使用类名作为局部初始化器的名字的一部分，可以让每个类的本地初始化器都具

有唯一命名，这样就不会被子类给覆盖了。①

好了，现在我们已经知道了如何正确地编写一个父类，接下来我们看看如何编写子类：CD和DVD。代码清单3-19列出了CD子类的代码。

代码清单3-19　编写CD子类

```
CD = Class.create();

CD.prototype = Object.extend(
  new Disc(),
  {
    initialize: function(title,artist,location) {
      this._initializeDisc(title,location,'CD');
      this.artist = artist;
    }
  }
);
```

❶ 用Prototype的 Class.create()方法定义CD类

❷ 合并Disc类

❸ 调用“继承的”初始化器

代码清单3-19非常之短，但是仔细研究之下，却并不简单。其中有许多不同寻常之处。

我们首先按照通常的方式使用Prototype的Class.create()方法❶。但随后定义类的prototype时，事情就变得相当有趣了。

通常，在创建一个类的prototype时，我们会将类的prototype赋值为一个对象，这个对象的属性即为类的成员和方法。但是在这里，我们不是直接将prototype设为这个对象，而是把该对象作为Object.extend()方法的第二个参数，将其与一个Disc类的新实例进行了合并❷。

那会发生什么呢？

回忆一下Object.extend()，它把在第二个参数对象中找到的所有属性加到第一个参数对象中，然后返回这个对象。

所以是这样：我们用来定义CD的成员（在本例中，就只有initialize()这一个方法）的那个hash对象被合并到Disc的一个新实例中，并且这个Disc实例成为了CD的prototype对象。结果，CD的原型包含了所有Disc的成员，以及所有CD的成员。因为CD是Object.extend()的第二个参数，所以若有相同名字的属性，最后只会留下CD的属性值。

事实上，这里并没有什么继承。但是通过将CD的属性与Disc的属性合并，达到了继承的效果：看上去，CD不仅仅定义了自己的属性，而且还继承了Disc的属性。

这个示例的另一个令人感兴趣的地方是在CD的初始化器中，我们调用了继承自Disc类的局部初始化器来设置公共属性❸。

DVD子类的声明与CD子类十分相似，请看代码清单3-20。

---

① 从作者的这两段解说我们可以看出，使用Prototype来构造类继承其实并不方便，因为extend()方法原本就不是为完整的类继承机制而设计的，它模仿的是Ruby语言中的extend的语义。困难的根源在于缺乏对super调用的支持，因此不得不为每一层级的类都做一个独立的初始化器，以便子类可以调用父类的初始化器（相当于super调用）。Prototype 1.6.0将引入全新的Class设计，在每个子类方法中，如果第一个参数名为$super，就可以在该方法中通过$super来调用对应的父类方法。这样就不再需要写出类似_initializeDisc()这样难看的代码了。——译者注

代码清单3-20 编写DVD子类

```
DVD = Class.create();

DVD.prototype = Object.extend(
  new Disc(),
  {
    initialize: function(title,director,location) {
      this._initializeDisc(title,location,'DVD');
      this.director = director;
    }
  }
);
```

我们应该做一些基本的测试吧？代码清单3-21列出了一个简单的测试页面，它的输出结果如图3-6所示。

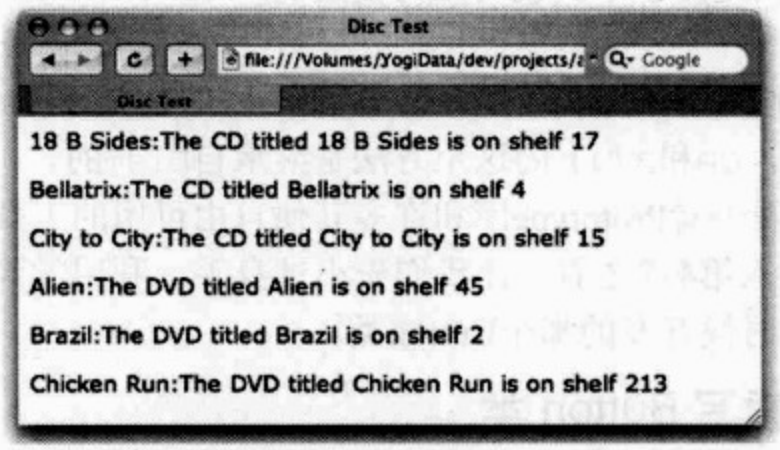


图3-6 碟片在哪里

代码清单3-21 测试继承层级

```
<html>
  <head>
    <title>Disc Test</title>
    <script type="text/javascript" src="../scripts/prototype.js"> </script>
    <script type="text/javascript" src="Disc.js"> </script>
    <script type="text/javascript" src="CD.js"> </script>
    <script type="text/javascript" src="DVD.js"> </script>
    <script type="text/javascript">
      var myCollection = [
       new CD('18 B Sides','Moby',17),
       new CD('Bellatrix','Jorgen Skogmo',4),
       new CD('City to City','Jerry Rafferty',15),
       new DVD('Alien','Ridley Scott',45),
       new DVD('Brazil','Terry Gilliam',2),
       new DVD('Chicken Run','Nick Park',213)
       /* and on and on ... */
      ];
    </script>
  </head>

  <body>
```

```
    <script type="text/javascript">
      myCollection.each(
        function(disc) {
          document.write(
            '<p>' + disc.title + ':' + disc.whereIsIt() + '</p>'
          );
        }
      );
    </script>

  </body>

</html>
```

当然了，如果真的要记录我们的收藏，我们大概会把它们存储在数据库里而不是一个HTML页面中！但在这里，我们只是为了举例说明如何使用Object.extend()来模拟类继承的概念，所以姑且如此。

当页面显示时，我们希望看到我们所收藏的碟片的标题和位置，如图3-6所示。注意whereIsIt()方法的使用，CD和DVD上的这个方法是继承自Disc的。

在下一章中，我们将看一看Prototype库和许多其他自由可用的工具是如何帮助我们轻松编写Ajax代码的。不过，在进入第4章之前，让我们先小试身手，利用学到的Prototype的各项功能来重写一下我们在本章早些时候开发的那个Button类。

3.2.6　用 Prototype 重写 Button 类

Prototype赐予了我们魔术手法般的技巧[①]来声明JavaScript对象类，现在让我们利用这些新能力来重新实现3.1.4节中的Button类。

重写的版本有很多代码和以前是相似的，但是也有不少重要的变化，请看代码清单3-22。

代码清单3-22　重访Button类

```
Button = Class.create();    ❶ 声明构造器

Button.prototype = {    ❷ 声明类的原型

  initialize: function(element, options) {    ❸ 接受构造器的参数
    this.element = $(element);
    if (!this.element) throw new Error(element + ' not found');
    this.options = Object.extend(
      {
        enabled: true,
        onClick: function() {},
        enabledClassName: this.CLASS_DEFAULT_CLASS_ENABLED,
        disabledClassName: this.CLASS_DEFAULT_CLASS_DISABLED,
        armedClassName: this.CLASS_DEFAULT_CLASS_ARMED,
        pressedClassName: this.CLASS_DEFAULT_CLASS_PRESSED
```

① 如果说这里所展现的是魔术手法（sleight-of-hand），那Prototype 1.6.0提供的就是超级符咒（super invocation）。——译者注

```
      },
      options
    );
    this.element.onclick = this.onclick.bind(this);
    this.element.onmouseover = this.onArm.bind(this);
    this.element.onmouseout = this.onDisarm.bind(this);
    this.element.onmousedown = this.onPress.bind(this);
    this.element.onmouseup = this.onRelease.bind(this);
    if (this.options.enabled) {
      this.enable();
    }
    else {
      this.disable();
    }
  },

  CLASS_DEFAULT_CLASS_ENABLED: 'buttonEnabled',
  CLASS_DEFAULT_CLASS_ARMED: 'buttonArmed',
  CLASS_DEFAULT_CLASS_DISABLED: 'buttonDisabled',
  CLASS_DEFAULT_CLASS_PRESSED: 'buttonPressed',

  onclick: function() {
    if (this.options.enabled) {
      this.options.onClick.call(this);
    }
  },

  onArm: function() {
    if (this.options.enabled) {
      this.element.className = this.options.armedClassName;
    }
  },

  onDisarm: function() {
      if (this.options.enabled) {
        this.element.className = this.options.enabledClassName;
      }
    },

    onPress: function() {
      if (this.options.enabled) {
        this.element.className = this.options.pressedClassName;
      }
    },

    onRelease: function() {
      if (this.options.enabled) {
        this.element.className = this.options.enabledClassName;
      }
    },

    isEnabled: function() {
      return this.options.enabled;
```

```
  },

  enable: function() {
    this.options.enabled = true;
    this.element.disabled = false;
    this.element.className = this.options.enabledClassName;
  },

  disable: function() {
    this.options.enabled = false;
    this.element.disabled = true;
    this.element.className = this.options.disabledClassName;
  }

}
```

重写版本与原来实现有一些显著不同之处。首先，我们用了Class.create()来声明构造器❶。如前所述，Prototype这样一种声明构造器的方式，将实际的设置代码转移到一个初始化方法中，使得类的代码具有一种一致性，这可能正是你想要的效果。不然，你可以继续使用原来固有的记法。不管用什么记法，都可以使用Prototype的其他设施。

接下来，类的prototype是用JSON记法声明的❷。JSON记法和Prototype库并没有必然联系，你可以选择使用它，也可以选择不使用。不过这一记法似乎成了许多使用Prototype的开发者的首选，事情也因此变得有趣。

因为我们使用了Prototype的构造器机制，所以我们必须创建一个initialize()方法来接受构造器的参数。于是我们就创建了它❸，并且在方法的第一行把element参数赋给element成员。因为我们使用的是Prototype的$()函数，所以element参数既可以是一个元素id也可以是元素的一个引用。

在原来的类中，我们可以通过做一些类型检查和条件赋值来加入这种灵活性，而这里，只要使用$()就可以免费获得。相应地，我们也将参数名从原先的elementName改为了更一般的element。

与原先的实现不同，我们并没有在<button>元素上创建属性来指回Button实例。这并不是我们冥顽不灵，而是已经无此必要了。一会儿我们会看到为什么。

在设置选项时，Prototype的表现确实相当出色。原来我们需要逐一装填options的成员，检查调用者是否已设置了这个选项，如果没有就使用默认值。在新的实现中，我们利用Object.extend()的能力，将调用者提供的选项对象和我们预先装填好默认值的对象进行合并。

比较一下代码清单3-8中原来的记法和代码清单3-22中的记法。显然，后者的方式更加清晰。可用的选项集与它们的默认值各自有序、一目了然。

接着，我们给<button>元素指派事件处理器。使用Prototype给Function对象增加的bind()扩展，我们快速轻松地建立了每个处理器，并且当它们被调用时，函数上下文将是Button实例而不是<button>元素。

在我们的原有实现中，我们要么依靠闭包来获得Button实例的引用，要么依靠加在<button>元素上的属性。而在这里两者都不需要，这也是我们为什么不再给元素加上属性的原因。

构造器中最后两行代码和原先相同，即确保按钮处于正确的初始状态。

实现的其余部分简单易懂。值得注意的是，在建立事件处理器时（多亏使用了bind()），所有处理器都可以认为，当自己被触发时，上下文对象就是Button实例（而不是<button>元素）。

3.3 总结

本章介绍了大量的内容。

我们集中讨论了怎样使用面向对象的概念来组织JavaScript代码，尽管JavaScript语言本身缺乏一些其他OO语言普遍具备的面向对象设施。我们学习了如何创建JavaScript类及其成员和方法，由此，JavaScript那些面向对象的兄弟语言在编程时所具备的优点，JavaScript也全部都获得了。将代码组织成一个一个的类，不仅让代码变得更有组织，而且也有利于复用，并让代码更易扩展和更好维护。

然后，我们介绍了Prototype代码库。你可以看到Prototype提供了众多有用的函数，几乎任何JavaScript程序代码都能从中获益。我们接着研究了Prototype库是如何帮助我们更轻松且更清晰地编写面向对象的JavaScript的。我们还看到了如何合并对象，以及如何使用这一能力在一个并没有类继承概念的语言中模拟类继承。

不过记住，我们在这里只是浅尝辄止。半个章节所能涵盖的仅仅是Prototype的很少一部分内容。例如，Prototype库包含了一些工具，让我们能轻松编写事件处理代码——这通常是一个很艰巨的任务，因为事件处理涉及许多特定于浏览器的棘手问题。Prototype还有许多内容有待探索。在下一章中，我们会对Prototype了解更多，特别是它的Ajax工具。

虽然Prototype很快成为了最流行的JavaScript程序库之一，但它并非是一枝独秀。在下一章中，我们将探索多种自由可用的程序库（也包括Prototype在内），看看在编写Ajax应用时，它们都能给予我们哪些具体的帮助。

第4章 Ajax开源工具集

本章内容

- 选择一个开源工具包
- 使用Dojo工具包发起Ajax请求
- 使用Prototype发起Ajax请求
- 使用jQuery发起Ajax请求
- 使用DWR调用服务器端的Java方法

我们时常得到他人的帮助。有些帮助需以金钱换取，有些则是无偿的。

互联网上有众多的开源工具，这些工具的作者将他们的劳动成果免费提供给他人。无论背后的动机是为了获得他人的赞誉、充实自己的履历、作为其他服务的免费广告、充当骄傲的资本，抑或是纯粹的利他主义，我们都心怀感激。

不过，这并不意味着我们可以信手拈来。

任何人只要有一个FTP客户端，就能把任何东西放到网络上。所以我们必须细细筛选，挑出能为我们所用的程序库和工具。在进行选择时，我们必须权衡各种因素，不过有一个很好的参考指标，那就是看看已有多少成功项目采用了这些工具和程序库。

在本章中，我们将纵览一批开源工具包，它们能使我们这些Ajax Web开发者的生活变得轻松一点。这些开源库包括Dojo工具包、Prototype库（这次，我们着重看它的Ajax功能）、jQuery以及DWR。你会在本书第二部分的实例里发现所有这些程序库的身影，Prototype用得尤其广泛，其他程序库也会在许多示例中用到。

注意，本章中的描述和示例并不是列举工具包各项特性的完整教程，也不会对工具包的功能做全面考察，而是会聚焦于浏览器与服务器间的Ajax异步通信问题，研究一下各个工具包都是如何简化这一过程的。

至于各个工具包提供的其他特性，还是留给读者去自行探索——这样的探索总是充满乐趣和惊喜的。

4.1 Dojo 工具包

Dojo是由Dojo基金会发布和维护的开源JavaScript程序库。像许多其他JavaScript工具包一样，它的目标是让DHTML编程，尤其是动画效果那样常见却复杂的任务变得更加轻松。

你可以从http://dojotoolkit.org/downloads/下载Dojo。它被打包成了众多版本，每个版本包含了满足不同需求的特定模块组合。我们后面的代码主要用到I/O模块，所以任何包含I/O包的版本都可以满足我们的需要[①]。

4.1.1 用 Dojo 进行异步请求

下载之后，只需把dojo.js文件放到Web应用的合适位置。为简单起见，在本节中，我们将把它和我们的示例放在相同的文件夹下。

针对异步请求，Dojo借助dojo.io.bind()函数[②]对发起和响应Ajax请求所需的步骤进行了简化。调用该函数时，我们需要传入一个JavaScript对象，其属性将作为函数的参数，这与我们在上一章中讨论过的选项表（options hash）类似。假如以前你没有接触过这种方式，可能会觉得它有点不合常规。但是这一方式确有显著优点，并且在JavaScript程序员特别是工具包作者中越来越流行。使用这一技巧，我们就不必考虑参数的顺序，也可以轻松处理可选参数，并且因为每个参数属性都有名字，所以调用语句具有很强的可读性。

1. 问题

我们要创建一个页面，能根据姓名列表查找与每个姓名对应的电话号码。我们不希望因提交表单而引起页面刷新。

2. 方案

为了解决这个问题，我们将使用Dojo I/O的bind函数向服务器发出异步调用，查找并返回与指定的姓名字符串相对应的电话号码。

首先，我们需要导入Dojo库。在页面的head元素中，加入

```
<script type="text/javascript" src="dojo.js"></script>
```

然后，为简单起见，我们将把姓名列表写死在代码中（在后面的问题/方案中，我们将看到如何动态获取姓名列表）。

在页面的body部分，编写以下的代码：

```
<form name="lookupForm" onsubmit="return false;">
  <select name="who" onchange="lookup();">
    <option value="JOHN">John</option>
    <option value="MARY">Mary</option>
    <option value="BILL">Bill</option>
  </select>
</form>
```

❶ 声明form元素

❷ 硬编码select控件所包含的选项

我们声明了一个表单元素❶，它所附带的onsubmit事件处理器会阻止表单提交给服务器。我们

---

① 很不幸，Dojo经常改头换面，不仅新版本与旧版本的API经常不兼容，而且会发生一些概念和策略上的彻底变化。本书所提到的Dojo预先提供不同的打包版本，比如0.4.3有Ajax Edition、Widget Edition、Storage Edition等数种预打包版本，现在就已经不再提供了（但用户可以通过Dojo的build工具进行定制打包）。——译者注

② dojo.io.bind()可以使用不同类型的传输机制（transport），比如XHR或IFrame，但是在Dojo 0.9之后，这个API被认为过于抽象而被取消，其功能被分解到许多独立的函数，例如dojo.xhrGet()、dojo.xhrPost()、dojo.io.iframe.send()等，参数形式仍然是使用hash，但具体参数有少许变化。请读者自行阅读dojo的文档。——译者注

会自行处理与服务器的通信，所以就不需要提交表单，至少在本示例中是这样的。我们还声明了一个带有onchange事件处理器的select元素，当选择了列表中的某一项时该处理器将查找电话号码。

本例中，控件中的选项是写死在页面中的。显然，在现实世界的编程中，需要读取服务器上的联系人数据库中的姓名，并动态地创建选项列表。这里我们可以暂时略过此节，或者想象这个页面就是由某种服务器端机制比如JSP或PHP生成的，并且服务器已经处理了这个方面的问题。

用户选择某个选项时，lookup()函数作为元素的onchange事件处理器被调用。此处，我们使用了Dojo来为我们发起Ajax调用：

```
function lookup() {
  dojo.io.bind(
    {
      url: 'phone.jsp?who='+document.lookupForm.who.value,
      mimetype: 'text/plain',
      load:
        function(type, data, req) {
          document.getElementById('displayArea').innerHTML = data;
        }
    }
  );
}
```

在这个函数中，我们调用了dojo.io.bind()函数并传入处理请求所需的信息。它是一个JSON格式的JavaScript对象，其中包含的特定属性将充当函数的参数。

url属性指定了所要调用的服务器端资源的URL。在本例中，我们已经定义了一个简单的JSP文件来处理请求。这个JSP文件简单至极，它利用强大的JSP表达式语言（JSP Expression Language）读取who这个请求参数并把它加上一个电话号码前缀，这样就创建出一个电话号码。代码就只有一行：

```
555.555.${param.who}
```

不过我们不妨想象这个JSP（或者是Servlet，或者是PHP脚本，又或者是其他服务器端资源）会执行数据库查询或其他过程来获取电话号码。

默认情况下，dojo.io.bind()函数在给服务器发送请求时，将使用HTTP的GET方法。我们也可改用POST方法，这需要添加一个值为POST的method属性，并通过content属性提供请求参数。

mimetype属性被设为text/plain，这表明响应将是纯文本的。load属性指定了在请求正常完成后所调用的处理器函数。因为这个示例中的处理器函数非常简短，所以我们直接把它内嵌于参数对象中。也可以在页面的其他地方定义函数，并在这里引用它。如果需要注册一个在事件出现问题时调用的处理器，可以使用error属性。

load处理函数被调用时会被传入三个参数：

- 第一个是被调用处理器的类型。在本例中将始终是load。这个参数使我们可以将一个处理器函数用于多个事件类型[①]。
- 第二个参数是响应数据。在本例中就是所生成的电话号码。

① 也就是说，在函数内可以通过type参数来了解当前事件类型，从而做出相应的处理。——译者注

- 第三个参数是XMLHttpRequest对象本身的引用[①]。这样，如果有需要，就可以查询XHR对象，获取请求状态的详情。

load处理器函数所完成的工作相当简单：取得名为displayArea的元素，对其innerHTML属性进行赋值，从而将所获得的响应数据插入该元素中。

displayArea元素被定义为一个初始时并无内容的span元素。

```
<div>
  Phone # is: <span id="displayArea"></span>
</div>
```

以上总结了页面所需的各个部分。浏览器显示效果如图4-1所示。代码清单4-1则列出了这个相当具有斯巴达风格[②]的页面的完整代码。

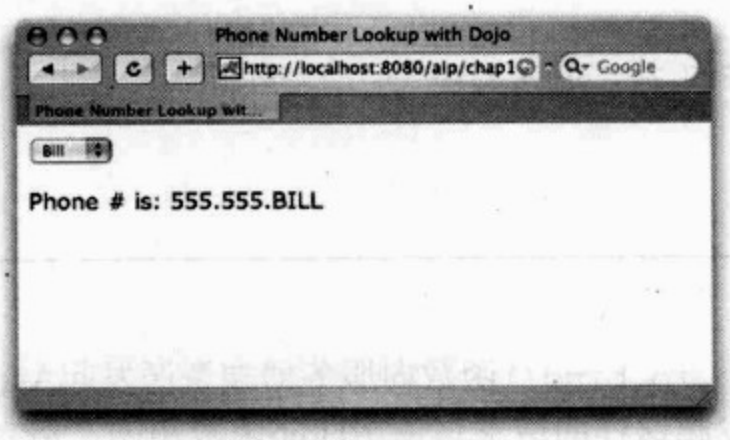


图4-1 比尔先生的电话

代码清单4-1 用Dojo查找电话号码

```
<html>
  <head>
    <title>Phone Number Lookup with Dojo</title>
    <script type="text/javascript" src="dojo.js"></script>
    <script type="text/javascript">
      function lookup() {
        dojo.io.bind(
          {
            url: 'phone.jsp?who='+document.lookupForm.who.value,
            mimetype: 'text/plain',
            load:
              function(type, data, req) {
                document.getElementById('displayArea').
                  innerHTML=data;
```

① 因为dojo.io.bind()的底层传输机制可能有多种，所以在使用非XHR机制时，第三个参数当然就不再是XHR对象了。实际上Dojo在这一点上相当混乱，不同的传输机制下，第三个参数并没有一定规律，有时甚至没有第三个参数，而有时还会有第四个参数，表示你传给dojo.io.bind()的原始参数。当然，如前所述，dojo.io.bind()这个API已经是过去时了，这种混乱也随之而去了，但是新API中的load等处理器的参数也发生了变化，具体请读者自行查阅Dojo的最新文档。——译者注

② 大约是指不加修饰的意思，以勇武著称的斯巴达人日常穿着很简单，尤其男孩在成年以前会被编入少年队，一年到头只能穿一件外衣。——译者注

```
                }
            }
        );
    }
</script>
</head>

<body>
  <form name="lookupForm" onsubmit="return false;">
    <select name="who" onchange="lookup();">
      <option value="JOHN">John</option>
      <option value="MARY">Mary</option>
      <option value="BILL">Bill</option>
    </select>
  </form>
  <div>
    Phone # is: <span id="displayArea"></span>
  </div>
</body>

</html>
```

3. 讨论

本节介绍了如何用dojo.io.bind()函数向服务器端资源发起Ajax请求。

直接使用XHR对象来发起这样的请求与使用Dojo函数相比，后者所需的JavaScript代码量当然少得多。然而，这并非只是几行代码，使用像Dojo这样的工具，有助于减少页面中的“管线设施”代码，聚焦于页面功能的核心代码则保持不变。减少管线代码，特别是对于一个大型复杂的页面来说，可以有效地降低页面的**复杂度**，即使我们从总体比例看代码行数只是稍稍下降了一点。

4.1.2　用 Dojo 自动对表单进行编组

上一节展示了如何轻松使用Dojo的dojo.io.bind()函数创建服务器回调并异步获取数据。不过这个示例有一个问题，为了构造GET操作所需的URL，我们必须建一个查询字符串，并把它加到URL后面。通常，最好避免手工建立查询字符串，因为这样有不少隐患，比如：

- 语法问题，你有多少次把“?”写成了“&”，或者反过来把“&”写成了“?”？
- 参数名和参数值编码错误（或者压根没有编码）。

如果我们不想被迫卷入这些问题，那么就得找其他办法来建立查询字符串。

1. 问题

我们希望不必再为Dojo的bind()函数的URL参数加上查询字符串。我们想：“如果能把包含select元素的表单直接‘提交’给异步请求，就不用自行读取控件的值来建立查询字符串了。”

Dojo恰好提供了这样的能力。

2. 方案

dojo.io.bind()函数的参数对象接受一个叫做formNode的属性，通过它我们可以指定一个表单元素的DOM节点，该表单中的控件将作为请求的参数传递给异步请求。这允许我们通过Ajax请求“提交”表单，尽管我们知道表单并没有被真正提交。

这需要对上一节的代码进行少许修改。首先，将bind()函数调用修改为：

```
dojoio.bind(
  {
    url: 'phone.jsp',     ❶ 看，没有参数！
    mimetype: 'text/plain',
    load:
      function(type, data, req) {
        document.getElementById('displayArea').innerHTML = data;
      },
    formNode: document.lookupForm     ❷ 指定要提交的表单！
  }
);
```

注意，在设定url属性值时，我们不再需要为URL建立和附加查询字符串❶，而且我们添加了一个formNode属性❷，指定了包含select元素的表单。bind()函数会自动对指定表单中的控件元素的值进行编组（marshal），并将它们作为参数传递给异步请求。在代码清单4-2中列出了修改后的代码，改动的部分以粗体显示。

代码清单4-2 用Dojo完成异步表单的任务提交

```
<html>
  <head>
    <title>Phone Number Lookup with Dojo</title>
    <script type="text/javascript" src="dojo.js"></script>
    <script type="text/javascript">
      function lookup() {
        dojo.io.bind(
          {
            url: 'phone.jsp',
            mimetype: 'text/plain',
            load:
              function(type, data, req) {
                document.getElementById('displayArea').innerHTML=data;
              },
            formNode: document.lookupForm
          }
        );
      }
    </script>
  </head>

  <body>
    <form name="lookupForm" onsubmit="return false;">
      <select name="who" onchange="lookup();">
        <option value="JOHN">John</option>
        <option value="MARY">Mary</option>
        <option value="BILL">Bill</option>
      </select>
    </form>
    <div>
      Phone # is: <span id="displayArea"/>
    </div>
```

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

3. 讨论

这个示例向我们展示了，如何轻松地将表单提交给异步请求，而不必麻烦地根据表单控件值来建立查询字符串。只要传递一个引用给Dojo函数，它就能确保为我们搞定所有细节。

采集表单值以及构建URL查询字符串虽然并非什么尖端技术，但确实是一个相当散乱和繁重的任务，很容易出现低级错误。Dojo代我们处理了这一累人的任务。这个示例说明，有些特性也许不能显著减少代码行数，但仍然可以显著降低页面代码的复杂度。

4.2 Prototype

我们在上一章中已经介绍过了Prototype，这个JavaScript工具包的目标就是让DHTML编程者的生活变得轻松。Prototype非常流行，不仅自身实用，而且也被用作许多其他工具包或框架的基础构件，比如Scriptaculous、Ruby on Rails、Rico等。

要使用Prototype，只需要prototype.js文件，它可以从Prototype的网站http://prototype.conio.net/上下载。像Dojo工具包一样，Prototype也提供了广泛的DHTML特性。如果你跳过了第3章中Prototype的部分，那在继续阅读之前可以回过去重新读一下该部分内容。我们之后的示例代码中会使用一些更有用的Prototype扩展。

4.2.1 Prototype 中的异步请求

和Dojo工具集一样，Prototype提供了一些简便的方法来通过Ajax发起异步请求。我们首先来看一下Prototype如何发起一个基本请求。

1. 问题

我们在一个订单页面遇到了一个常见的问题，即根据一个下拉列表控件（<select>元素）的选择结果动态填充另一个下拉列表控件的内容。本例中，我们的下拉列表列出了我们现有的T恤的颜色种类。我们需要根据所选的颜色动态地查阅存货数据库并填充尺寸下拉列表来显示该颜色T恤实际有货的那些尺寸种类。

2. 方案

首先，我们得导入Prototype库。假设我们把prototype.js文件放在和HTML页面相同的目录下，那样引用起来会比较简单一点：

```
<script type="text/javascript" src="prototype.js"></script>
```

现在来建立表单。为简单起见，只包含两个下拉列表元素和一个提交按钮。显然，一个实际的订购表单肯定需要许多其他控件。

```
<form action="/submitOrder" name="tshirtForm">
  <label>T-shirt color:</label>
  <select name="color" id="color" onchange="updateSizes();">
    <option value="">Select color</option>
    <option value="cardinal">Cardinal</option>
    <option value="ecru">Ecru</option>
```

❶ 定义onchange事件处理器

```
      <option value="hunter">Hunter</option>
      <option value="azure">Azure</option>
    </select>
    <label>Size:</label>
    <select name="size" id="size" disabled="disabled">
      <option value="">Select size</option>
    </select>
    <input type="submit"/>
  </form>
```

❷ 创建尺寸下拉列表

这个表单有几点值得注意。首先，表单的所有控件元素既有id属性也有name属性，而且它们的值是相同的。这样我们既可以通过ID也可以通过名字来引用这些控件元素，本来这两种标识符各有各的JavaScript命名空间，容易混淆，现在就不必担心了，因为两者的值是相同的，所以不管哪种方式你总能引用到同一个控件。我们还在颜色下拉列表上定义了一个onchange事件处理函数❶，当用户从列表中选择一个颜色时我们就可以有所动作。尺寸下拉列表❷除了“Select size”的指示以外没有任何内容，并且初始处于禁用状态，因为没有填充值之前，用户点击它并没有什么意义。

当用户从可选颜色列表中选择一个颜色时，就会调用updateSizes()函数：

```
function updateSizes() {
  if ($F('color')=='') return;
  new Ajax.Request('getSizes.jsp?color=' + $F('color'),
    {
      method: 'get',
      onSuccess: populateSizes,
      onFailure: function(r) {
        throw new Error( 'Fetch sizes failed: '
          + r.statusText );
      }
    }
  );
}
```

❶ 获得颜色下拉列表被选中的值

❷ 抛出错误

这个事件处理器首先使用$F()函数来获得颜色下拉列表中被选中的值❶，并且在没有颜色被选中时退出（用户可能点击“Select color”项，它只是一个辅助标签）。为了确保尺寸元素处于一个已知状态，我们可以更严格一些，但目前我们主要关注Ajax请求。

如果通过了检查，就用Prototype的Ajax.Request对象来创建一个异步请求。触发请求的方式是，构造一个Ajax.Request的新实例，并传递两个参数：请求的URL以及一个hash对象——它包含的属性指定了请求的选项（我们在第3章中就用过这一技巧，而且Dojo工具包也采用这种方式）。

URL指向一个JSP文件，我们把所选的颜色值传给它，它则会模拟数据库查询库存。在选项参数对象中，我们通过method属性指定了HTTP方法为GET（Prototype要求这个值是小写的[①]），并通过onSuccess和onFailure提供了请求成功处理器和和请求失败处理器的函数引用。

请求失败处理器会被传入一个XHR实例的引用，我们会据此抛出一个描述失败状态的错误❷。请求成功处理器则指向populateSizes()函数。

① 早期Prototype有这样的设计缺陷，1.5.0 rc2开始已经没有这个限制了。——译者注

为了简化客户端代码（通常应该尽量让服务器端代码处理复杂问题以便简化客户端代码），JSP会返回一个JSON字符串作为响应，它包含可用尺寸的JavaScript数组。典型的响应大致是这样的：

```
['Small','Medium','Large','XL','XXL']
```

下面是JSP代码，其中使用了JSTL（JSP标准标签库，JSP Stanardard Tag Library）的core标签库：

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<c:choose>
  <c:when test="${param.color =='azure'}">
    ['Small','Medium','XXXL']
  </c:when>
  <c:when test="${param.color =='cardinal'}">
    ['Medium','Large','XL']
  </c:when>
  <c:when test="${param.color =='ecru'}">
    ['Small','Medium','Large','XL','XXL','XXXL']
  </c:when>
  <c:when test="${param.color =='hunter'}">
    ['Small','Medium','Large','XL','XXL']
  </c:when>
</c:choose>
```

当然，在现实世界中，这应该是一个Servlet或其他服务器端资源，且会确切执行一个数据库查询，而不是返回硬编码的值。

当JSP返回响应时，Prototype的Ajax.Request对象会调用popluateSize()函数（当然，是在假定一切运行正常的前提下），我们是这样定义这个函数的：

```
function populateSizes(r) {
  eval('var sizes=' + r.responseText);
  var sizeElement = $('size');
  while (sizeElement.options.length > 1) {
    sizeElement.remove(1);
  }
  for (var n = 0; n < sizes.length; n++) {
    sizeElement.add(
      new Option(sizes[n],sizes[n]),document.all ? 0 : null
    );
  }
  sizeElement.disabled = false;
}
```

❶ 获得下拉列表元素的引用

清空下拉列表元素

❷ 添加选项

事件处理器会被传入一个XHR实例的引用，首先要做的事是获得响应的结果。通过使用JavaScript的eval()函数，我们对JSON的响应文本进行求值运算，然后把它赋值给一个变量以便之后使用。在求值运算后，这个变量会包含一个Javascript字符串数组的引用，其中存储了JSP所返回的对应于所选颜色的尺寸值。

我们要把这些值加入到尺寸下拉列表中，所以我们先获得这个下拉列表元素的引用❶，然后清空它，移除除了第一个（包含Select Size辅助标签的选项）以外的其他所有选项❷。之后我们遍历每个返回的尺寸值，把一个包含该尺寸的新选项添加到尺寸下拉列表中。

在调用select元素的add()方法时❷，第二个参数需要做下解释：

```
document.all ? 0 : null
```

在W3C规范中，select元素的add()方法的第二个参数指定了一个现有选项的索引，表示新选项插入到该选项之前，或者可以指定null，表示新选项插入到选项列表的尾部。

然而，IE却用0代替null来指明新选项应该放到尾部，所以我们做一个浏览器检测以便提供适当的值。通常我们建议使用**对象检测**而不是浏览器检测来做出特定于客户端的选择，不过在这个例子里并没有一个对象可用于检测从而提供适当的值[①]。

最后，在添加了所有尺寸之后，就可以启用尺寸下拉列表了。在浏览器中，选择颜色之前我们所看到的页面如图4-2所示。选择颜色之后则如图4-3所示。页面的完整代码列于代码清单4-3中。

图4-2 选择颜色之前的页面

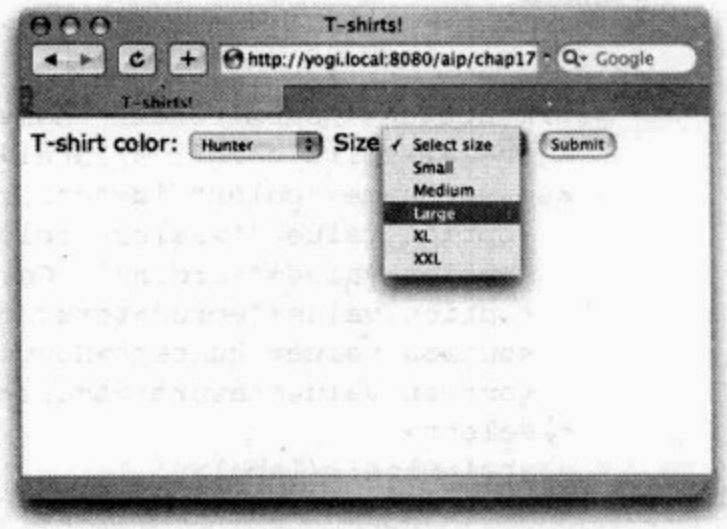


图4-3 选择颜色之后的页面

代码清单4-3 用Prototype进行动态查找

```
<html>
  <head>
    <title>T-shirts!</title>
    <script type="text/javascript" src="prototype.js"></script>
    <script>
      function updateSizes() {
        if ($F('color')=='') return;
        new Ajax.Request('getSizes.jsp?color=' + $F('color'),
          {
            method: 'get',
            onSuccess: populateSizes,
            onFailure: function(r) {
              throw new Error( 'Updates sizes failed: ' +
                               r.statusText );
            }
          }
        );
      }
```

① 所以，这里的代码是通过检测document.all这一IE特有的字段来选择参数的。这其实暗示了对象检测法的一个根本缺陷，也就是即使有相同名字的方法，也不能确保他们的行为是跨浏览器一致的。所以与作者建议的相反，我个人更喜欢使用明确的浏览器检测，当然这类问题是见仁见智的。——译者注

```
    function populateSizes(r) {
      eval('var sizes=' + r.responseText);
      var sizeElement = $('size');
      while (sizeElement.options.length > 1) sizeElement.remove(1);
      for (var n = 0; n < sizes.length; n++) {
        sizeElement.add(
          new Option(sizes[n],sizes[n]), document.all ? 0 : null
        );
      }
      sizeElement.disabled = false;
    }
  </script>
</head>

<body>
  <form action="/submitOrder" name="tshirtForm">
    <label>T-shirt color: </label>
    <select name="color" id="color" onchange="updateSizes();">
      <option value="">Select color</option>
      <option value="cardinal">Cardinal</option>
      <option value="ecru">Ecru</option>
      <option value="hunter">Hunter</option>
      <option value="azure">Azure</option>
    </select>
    <label>Size:</label>
    <select name="size" id="size" disabled="disabled">
      <option value="">Select size</option>
    </select>
    <input type="submit"/>
  </form>
</body>
</html>
```

3. 讨论

和Dojo工具包一样，Prototype会处理发送请求的细节以及状态改变时的对应回调，这让我们能以一种（相比直接使用XHR对象）更简单的方式来发起异步Ajax请求。初始化和处理请求所必需的代码会被抽象为简单的构造器调用，而我们的重点则能集中到处理过程上。

Prototype在这个部分所提供的抽象虽然级别不高，在这样简单的例子里也可能体现不出什么特别价值，但是在现实世界的更加复杂的页面中，保持代码整洁就成为了提高页面可维护性和可扩展性的关键因素。

4.2.2　用 Prototype 进行自动更新

在上一个问题中，从服务器返回来的数据需要进行一些处理然后才能显示。字符串数组的值需要被转换成option元素并组装到一个select元素中。

不过更常见的情形是，从服务器返回的数据无需处理直接显示。对此，Prototype提供了Ajax.Updater类，进一步简化了这个过程。

1. 问题

当页面发生事件，比如点击了一个按钮，我们希望从服务器获得日期和时间，并在页面上显示出来。

2. 方案

如往常那样，先从导入Prototype工具包开始：

```
<script type="text/javascript" src="prototype.js"></script>
```

我们将在按下按钮后触发日期和时间信息的更新操作，所以得定义一个按钮：

```
<button type="button" onclick="update();">Click me!</button>
```

再定义一个初始为空的用于显示日期和时间的容器：

```
<span id="timeDisplay"></span>
```

按钮的onclick事件触发update()函数，在该函数中我们使用Ajax.Updater对象：

```
function update() {
  new Ajax.Updater('timeDisplay','date.jsp',
    {
      method: 'get'
    }
  );
}
```

同Ajax.Request对象的用法一样，我们先创建一个Ajax.Updater对象实例，并通过构造器参数传入相关信息。

第一个参数指定了放置响应文本的元素id，第二个参数指定了从哪个URL获得响应。在本例中，服务器端资源是一个名为data.jsp的JSP页面，它使用了JSTL的国际化格式标签库来格式化并返回当前时间：

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<jsp:useBean id="now" class="java.util.Date"/>
<fmt:formatDate value="${now}" pattern="MMMM dd, yyyy hh:mm aa"/>
```

第三个参数是包含请求选项的JavaScript对象。这里我们仅仅指定了HTTP方法为GET。点击按钮后的显示结果如图4-4所示。页面的完整代码则列于清单4-4中。

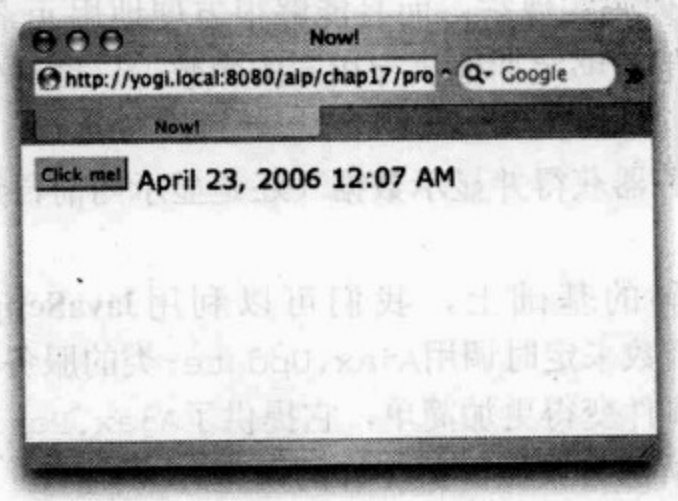


图4-4 用Prototype来查询时间

代码清单4-4　与时俱进

```
<html>
  <head>
    <title>Now!</title>
    <script type="text/javascript" src="prototype.js"></script>
    <script>
      function update() {
        new Ajax.Updater( 'timeDisplay', 'date.jsp',
          {
            method: 'get'
          }
        );
      }
    </script>
  </head>

  <body>
    <button type="button" onclick="update();">Click me!</button>
    <span id="timeDisplay"></span>
  </body>
</html>
```

3. 讨论

在许多支持Ajax的页面中最为常见的操作就是从一个Ajax请求中获得数据来更新一个HTML元素。通过Ajax.Updater对象，Prototype将执行这一常见任务所需的代码减至最少。

它特别适用于请求那些简单地返回已格式化的HTML代码的服务器端资源，比如JSP页面、PHP脚本或静态HTML文件。让服务器端资源来执行数据的格式化，可以大大减少页面中的处理过程，否则我们必须获取原始数据，将它格式化为HTML字符串以便用于innerHTML，或者用DOM的API来动态建立所需的HTML元素。

4.2.3　用 Prototype 进行定期更新

在上一个问题中，我们使用Prototype的Ajax.Updater对象，使得来自服务器的预先格式化好的响应数据能自动地更新到一个HTML元素容器中，我们所举的例子是显示当前日期和时间。

我们已经能用最少的代码实现它，而且能够很方便地用元素的id来指定用于显示的目标元素。不过有时候我们还希望能够定期地执行相同的函数，以便确保尽可能显示出最新的数据。

1. 问题

我们希望定期地从服务器获得并显示数据（还是显示当前日期和时间）。

2. 方案

在上一个方案的代码的基础上，我们可以利用JavaScript的window.setTimeout()或window.setInterval()函数来定时调用Ajax.Updater类的服务，这样就可以轻松地达到目标。但是，Prototype还可以让事件变得更加简单，它提供了Ajax.PeriodicalUpdater类。使用这个类，我们只需要对上一个解决方案做一些少许的调整。

我们调用一个稍有变化的后端处理资源——data2.jsp。与之前类似，它也在响应中返回当

前日期和时间。但这次我们在时间值中包含秒数，这样就可以检测出以秒为单位的更新：

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<jsp:useBean id="now" class="java.util.Date"/>
<fmt:formatDate value="${now}" pattern="MMMM dd, yyyy hh:mm:ss aa"/>
```

对update()函数的改动包括使用Ajax.PeriodicalUpdater类，以及指定进行自动更新的时间间隔：

```
function update() {
  new Ajax.PeriodicalUpdater('timeDisplay','date2.jsp',
    {
      method: 'get',
      frequency: 5
    }
  );
}
```

正如你看到的，这里仅仅有两处改动。首先创建一个Ajax.PeriodicalUpdater的实例，所请求的URL指向新的JSP页面。其次，我们在参数对象中引入了一个frequency属性，它指定了两次更新间隔的秒数。

当在浏览器中显示这个页面时，它看上去与之前的方案别无二致，区别在于当你点击了按钮显示日期之后，它会每隔5秒钟自动更新一次。这个页面的完整代码列于清单4-5中，较之上一个方案修改过的部分以粗体显示。

代码清单4-5 永远与时俱进

```
<html>
  <head>
    <title>Right now!</title>
    <script type="text/javascript" src="prototype.js"></script>
    <script>
      function update() {
        new Ajax.PeriodicalUpdater( 'timeDisplay', 'date2.jsp',
          {
            method: 'get',
            frequency: 5
          }
        );
      }
    </script>
  </head>

  <body>
    <button type="button" onclick="update();">Click me!</button>
    <span id="timeDisplay"/>
  </body>

</html>
```

3. 讨论

我们没费多大工夫就改进了之前的示例，现在它能按照我们需要的频率自动保持显示服务器

提供的最新数据。Prototype的Ajax.PeriodicalUpdater类让我们可以轻松排定自动更新计划，而不必编写任何调度逻辑和timeout处理函数。

在定时触发自动更新时，必须注意不要超过服务器的负载能力。尽管乍看之下每一秒都触发一次更新来保证显示最新的时间是个不错的主意，但是想象一下成千上万的访问者都在看这个页面——会有大量数据更新请求持续不断地“捶打”我们的服务器。就算请求和生成的响应通常都非常小（在这个示例中也确实如此），但是所谓“积羽沉舟，群轻折轴”，服务器完全可能被大量请求所冲垮。

所以我们要进行权衡：哪些内容需要自动更新、更新的间隔、预期的网站并发访问量、更新响应的大小、生成响应所需的处理过程、服务器配置的负载处理能力，等等，所有这些都需要通盘考量。最好，更新间隔写在一个配置文件中，而不是写死在页面中，这样网站管理员就可以随时调整这个值而无需改写页面。

4.3　jQuery

jQuery，自称为“新式”JavaScript库，相比我们在本章接触过的其他工具包，它的编程理念稍有不同。jQuery力图改变JavaScript的编程方式，并且jQuery的哲学确实对我们开发脚本的方式产生了巨大的冲击。

在http://jquery.com/上可以找到jQuery的下载以及文档。你可以下载未经压缩的程序库（代码可供阅读），也可以下载一个较小的经过压缩的文件（不具可读性）。

不管是哪个，在用到jQuery的页面中，都需要导入jQuery脚本文件。根据本节的目的，我们将使用没有压缩过的（可读的）版本，把它放在和示例页面相同文件夹下（以便于导入），并命名为jquery.js。

4.3.1　jQuery基础

在深入研究如何用jQuery发送Ajax请求之前，让我们先来看看一些基本概念。要理解jQuery的工作方式，首先就需要掌握这些概念。

本节并非是jQuery的完整教程——这需要多得多的篇幅——但是本节应该能让你对jQuery工作方式背后的哲学有个大致概念。

1．jQuery包装器

我们看到过的其他程序库，比如Prototype，其工作方式是通过引入新的类以及扩展内建的JavaScript类来增加我们页面上脚本的功能。例如在第3章中我们可以看到Prototype是如何扩展Object、Function以及Array类的。

jQuery则采取不同的方式。

jQuery不是对类进行扩展，而是提供了一个新类，其名字就是jQuery，它作为其他对象的**包装器**，为那些对象提供扩展操作。包装器对象的概念对于面向对象程序的资深开发者来说并不陌生。这一模式常用于**适配器**，它提供了一个与对象原本接口不同的接口来操作对象。

在jQuery中，大多数操作是这样进行的：用jQuery包装器包装一组元素，然后调用包装器的方法对经过包装的元素进行操作。为了让包含jQuery包装器的表达式和语句更加紧凑，jQuery类

被映射为$。不要把它和Prototype的$()混为一谈，因为它的目的完全不同。

jQuery对象可以包装许多不同的对象类型，并且包装的对象不同，它所提供的功能也各有不同。例如，我们可以包装一段HTML：

```
$("<p>What's cooking?</p>")
```

这由HTML构建了一个DOM片段，我们可以使用jQuery的方法来操作它。例如，如果我们想要把这个片段加到文档的末尾，可以这样：

```
$("<p>What's cooking?</p>").appendTo("body");
```

Ajax开发者经常需要生成新的DOM元素，采用这样便捷简短的方式就可以实现添加元素的功能，jQuery的优点是显而易见的。

除了添加新的DOM元素外，我们还经常需要操作页面中已有的元素。jQuery包装器也可以包装已有的元素，只需传给$()一个字符串，$()包装器支持许多种方式来标识出需要包装的项目：CSS选择器、XPath表达式以及元素名。我们将在示例代码中大量使用CSS选择器。比如：

```
$("div")
```

这将对文档中的所有 元素进行包装以便操作。另一个示例是：

```
$("#someId")
```

这会对id为someId的DOM元素进行包装以便操作。还有第三个示例：

```
$(".someClass")
```

这将包装所有具有someClass类名的元素。

jQuery的作者非常聪明地运用了CSS选择器和XPath来标识目标元素，而不是发明某些jQuery专用语法去强迫用户接受。作为网页开发者，我们对这些机制已经相当熟悉，这让我们可以更容易接受和使用jQuery来标识所需操作的元素。

还可以包装其他项目，如元素和函数。稍后我们将看到这样的例子。

2. 串连jQuery操作

jQuery还很明智地允许我们在一个单独的表达式中把许多操作串连起来。绝大多数jQuery包装器方法会返回一个jQuery包装器对象自身的引用，这样当我们需要对所包装的对象执行多个操作时，就能够在一个单独的表达式上不断追加操作。

考虑这种情况，我们想要给一个元素（它的id是something）增加一个CSS类，并且将这个元素显示出来（假设最初它是隐藏的）。我们不用这样：

```
$('#something').addClass('someClass');
$('#something').show();
```

而可以这样写：

```
$('#something').addClass('someClass').show();
```

3. 在文档就绪时执行代码

在页面中，我们往往需要执行一些初始化代码，以便在用户与应用交互前先做好准备工作。通常我们使用窗口对象的onload事件处理器来做初始化。它保证在执行onload代码前页面已加

载完毕，由此保证了DOM元素已经存在，从而可以进行DOM操作了。

但是onload有一个问题，就是它不仅会等待文档主体加载完毕，还会等待图像等资源加载完毕。如果这些图像不在浏览器的缓存中，就必须从服务器上去拉回来，这样初始化代码运行的那个时间点就远远迟于文档本身加载完成的那个时间点，而当时已经完全可以执行初始化代码了。

jQuery帮我们解决了这一问题，它引入了“文档就绪处理器”的概念。这一机制令函数在文档加载完成之时执行，而不必等待图片及onload事件处理器。

要使用这一机制，我们需要对文档元素进行包装，并调用包装过的文档对象上的ready()方法：

```
$(document).ready(function);
```

一旦DOM就绪并可供操作，传给ready()的函数就会执行。注意，当你同时使用ready机制和页面上的onload事件处理器时，两者都会执行，且ready事件处理器会先于onload事件处理器被触发。

ready()处理器有一个简略记法，即直接用jQuery的包装器对一个函数进行包装。下面的代码片段：

```
$(function);
```

与之前那段声明ready()处理器的代码是等价的。

4. 同时使用jQuery和Prototype

Prototype非常流行，而jQuery也迅速走红。因此，网页创作者完全有可能想在一个页面中同时使用这两个库。

总的来说，jQuery遵循着最佳实践的准则，避免污染全局命名空间(global namespace)——例如将工具函数这类构造放到jQuery命名空间中而不是直接放在全局命名空间中。不过有一个地方存在冲突，就是我们之前提到过的，使用了$这一全局名称。

jQuery，作为JavaScript程序库世界中的一个模范公民，已经预先考虑到了这个问题。当在一个页面中同时使用Prototype和jQuery时，只要在两个程序库加载完成后调用jQuery的jQuery.noConflict()工具函数，就会使得$名字的功能恢复成Prototype的定义。

通过jQuery命名空间，你仍可以使用jQuery的功能，你也可以定义简短的别名。对于将jQuery与Prototype结合使用的情况，jQuery文档建议使用如下的别名：

```
var $j = jQuery;
```

好了，我们已经学习了足够的预备知识了！

在本节后续的解决方案中我们将看到jQuery的更多用途。即便如此，我们也只是稍微触及到了jQuery的皮毛。如果你在学习这些解决方案后发现自己已被jQuery的能力所深深吸引，那么我们强烈建议你访问http://docs.jquery.com/，阅读全面的在线文档并了解jQuery所提供的其他能力。

4.3.2　用 jQuery 进行异步加载

jQuery提供了不少方法来发送Ajax请求。有些是简单且实用的高层方法，用于初始化Ajax请求，并执行大多数一般的请求任务。其他一些方法是底层的，它们提供了对Ajax请求各方面

的控制。

在本节的解决方案中，我们将使用具有代表性的一些方法。首先来处理最普遍的Ajax交互：从服务器获得动态内容。

1. 问题

让我们来想象一个电子冰箱——一个假想的高科技电冰箱，它不仅可以保存食品，而且还提供一个网络接口能够让软件用来与其通信和交互。

电子冰箱用来记录其存货的虚拟技术并不重要。它可以是条形码扫描，RFID（无线电频率识别）标签，或者是其他虚构的技术。作为网页的作者，关心的是拥有一个服务器组件，能够让我们发出请求来获得食物状态的信息。

网页的焦点将是呈现电子冰箱中的食物列表。通过点击列表中的某一个食物，可以显示出更加详细的信息。

针对这个问题，我们将假设网页已经预先载入了食物的列表，它是通过某一种服务器端模板机制生成的。在下一节中，我们将看到一种从服务器动态获取列表的技术。

2. 解决方案

一开始，为了在页面中使用jQuery，就必须导入jQuery库：

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

假设已经由服务器端机制生成了电子冰箱内的食物，并在一个select元素中陈列出来：

```
<form>
  <select id="itemsControl" name="items" size="10">
    <option value="1">Milk</option>
    <option value="2">Cole Slaw</option>
    <option value="3">BBQ Sauce</option>
    <option value="4">Lunch Meat</option>
    <option value="5">Mustard</option>
    <option value="6">Hot Sauce</option>
    <option value="7">Cheese</option>
    <option value="8">Iced Tea</option>
  </select>
</form>
```

为了达到本示例的目的，我们仅列出8个食物。一般的冰箱通常会包含更多的食物，但是，我们都知道速食主义者的冰箱内的食物总是很稀少的。

服务器（或许可以叫做“电子冰箱驱动”）分配给食物一个标识号，它被用来唯一标识每个食物。在这里，简单地用一个连续整型值。这个标识值将被设置为每个<options>的value，用来标明食物。

虽然我们知道需要select控件对用户的输入做出反应，但可以注意到，在创建<select>元素标记中并没有声明任何处理函数。这就需要提出jQuery设计背后的另一个体系。

jQuery的一个目标是让网页作者能够轻松地把脚本从文档标记中分离出来，这和CSS允许我们把表现从文档标记中分离出来的方法很像。诚然，我们可以不用jQuery来做到这些事，毕竟jQuery是用JavaScript编写的，而且不是所有无法做到的事它都去帮我们做了。但是，它还是为我们做了不少事，并且以轻松地从文档标记中分离脚本为目标来设计。所以，与其直接在<select>

元素标记上增加一个onchange事件处理，还不如用借助jQuery的帮助，在脚本控制下增加这个事件处理。

在文档就绪前，我们无法操作页面上的DOM元素，所以在页面首部的<script>元素中，我们将用一个之前讨论过的jQuery ready()处理函数。在这个处理函数内，我们将用jQuery的方法来给元素增加一个change事件处理。请看下面的代码片段：

```
$(document).ready(function(){
   $('#itemsControl').change(showItemInfo);
});
```

在ready()处理函数内，我们通过传递itemsControl这个id创建了一个封装了<select>元素的jQuery实例。之后，用了jQuery的change()方法，它把自己的参数分配给封装的元素，作为它的change事件处理。

这里，我们已经声明了一个showItemInfo()函数。其中，我们将为列表中选中的食物创建Ajax请求。

```
function showItemInfo() {
   $('div#itemData').load(
     'fetchItemData.jsp',
     {itemId: $(this).val()}
   );
}
```

❶ 封装元素，并调用load方法

❷ 给出服务器端资源

❸ 获得食物的id并把它作为参数传递

jQuery提供大量不同的方法来向服务器发送Ajax请求。这个解决方案的目的是希望从服务器得到一个预先格式化好的HTML片段（包含了食物的数据）。并且把它加载到一个元素中，即一个id是itemData的 元素。jQuery的load()❶方法可以完美地达到这个要求。

这个方法将从由第一个参数提供的URL处取得一个响应，并且把它插入到封装的DOM元素内。这个函数的第二个参数允许我们传递一个对象，它的属性将作为请求的参数。第三个参数可以用来指定一个在请求完成后执行的回调函数。

首先，我们封装一个DOM元素❶，它通过CSS选择器div#itemData来定位，是一个用于把加载的食物数据放入其中的空 元素。之后，用load()方法，提供一个JSP页面的URL❷，它将获取由itemId请求参数指定的食物数据，这个参数是方法的第二个参数提供的❸。

那个参数的值必须是用于在<select>元素中用户点击的那个option的值。因为<select>元素被设置为chanage句柄的函数上下文，所以可以通过this引用来访问它。我们封装这个引用，并用jQuery的val()方法来获得当前被选中的控件的值❸。

因为我想要做的就只是给DOM中加载食物数据，所以我们并不需要一个回调函数，省略了load()方法的第三个参数。

这就是全部了。

jQuery拥有许多常用的过程，它们实现了大量的代码，并且允许我们用很少的几行代码来执行它们。这个处理函数调用的JSP页面用了itemId请求参数的值来获取相应食物的信息，并把它格式成HTML显示在页面上。

图4-5展示了在选择电冰箱内的食物后显示的完成页面。其源代码在清单4-6中完整地列了出来。

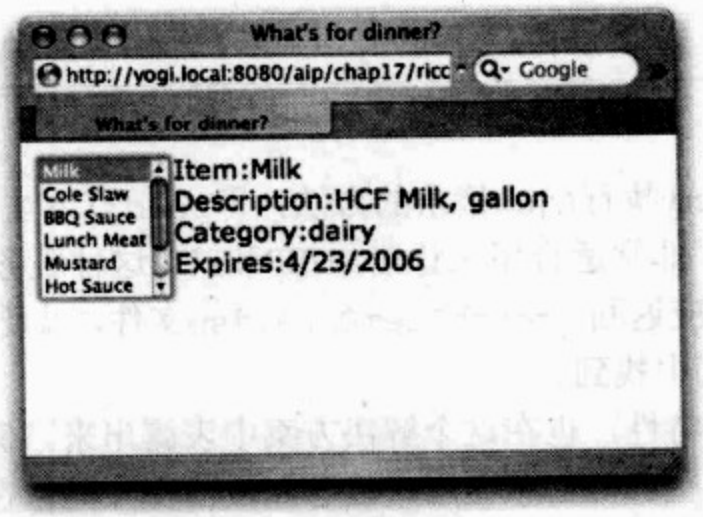


图4-5 要牛奶吗

代码清单4-6 用jQuery来决定晚饭吃什么

```
<html>
  <head>
    <title>What's for dinner?</title>
    <script type="text/javascript" src="jquery.js"></script>
    <script type="text/javascript">
      $(document).ready(function(){
        $('#itemsControl').change(showItemInfo);
      });
      function showItemInfo() {
        $('div#itemData').load(
          'fetchItemData.jsp',
          {itemId: $(this).val()}
        );
      }
    </script>
    <style type="text/css">
      form,#itemData {
      float: left;
      }
    </style>
  </head>

  <body>
    <form>
      <select id="itemsControl" name="items" size="10">
        <option value="1">Milk</option>
        <option value="2">Cole Slaw</option>
        <option value="3">BBQ Sauce</option>
        <option value="4">Lunch Meat</option>
        <option value="5">Mustard</option>
        <option value="6">Hot Sauce</option>
        <option value="7">Cheese</option>
        <option value="8">Iced Tea</option>
      </select>
    </form>

    <div id="itemData"></div>
```

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

3. 讨论

本节为我们介绍了jQuery执行Ajax请求的方法，即load()方法。

jQuery的load()方法，非常适合用来让诸如JSP和PHP这种服务器端的模板语言格式化一小段数据，并把HTML作为响应返回。fetchItemData.jsp文件，以及仿造电子冰箱功能的Java类，可以在本章供下载的源代码中找到。

其他一些重要的jQuery特性，也在这个解决方案中表露出来。例如，我们用了一个ready()句柄来触发代码的执行，这些代码必须在用户能够和页面交互前但在整个DOM构建完成后执行。

我们还看到了val()方法，它返回一个封装的input元素的值。如果有多于一个元素被封装，那么这个方法将返回第一个匹配的元素的值。

在这个解决方案中，我们假设电子冰箱的原始列表是由任何一种服务器端语言通过生成页面而给出，比如，一个JSP模板。这是Web应用的一种普遍做法，但是，为了探索jQuery的Ajax能力，在下一个问题中，就需要在页面加载中动态获取那个列表。

4.3.3　用 jQuery 获取动态数据

在上一节中，我们介绍了jQuery的load()方法，它可以非常轻松地执行获取HTML片段并把它加载到一个DOM元素中的任务。虽然这个方法非常好用，但是有时我们却想要获得对Ajax请求过程的更多控制权，或者想从服务器获得数据（而不是预格式化好的HTML）。

在本节中，我们将探索jQuery在Ajax领域提供的更多的东西。

1. 问题

我们希望扩展之前的代码，使它从页面初始化的异步请求中获得电子冰箱内的食物列表。

2. 解决方案

回顾下上一个解决方案，可以发现为了动态地加载选择框的选项，我们需要做的修改就是从<select>元素中移除<options>元素，并且在ready()句柄中增加代码来获取并加载这些食物。但是，在着手之前，我们先来修改编写showItemInfo()处理函数的方式，这样可以来找个借口继续探索jQuery的性能。除了使用jQuery封装器的load()的方法，这次我们将使用jQuery的另外一个有用的函数：$.get();

不过，先等一下。那个函数里面的句点在这里是干嘛用的？这不像是我们一直在用的$()封装器。

事实上，jQuery不仅仅提供了我们一直用着很不错的那个封装器类，而且还提供了大量实用的方法，许多是作为$封装器类的类方法实现的。

假如$.functionName()这样的符号看上去很怪，那么可以想象不使用$别名的表达式：

```
jQuery.get();
```

好，这样看起来就比较熟悉了。$.get()函数是作为一个类方法定义的，也就是jQuery封装器函数的一个函数属性（如果类方法的概念还是让你很头痛，那么最好能回顾一下3.1.3节）。

虽然，我们知道了它们是作为jQuery封装器类的类方法，但是jQuery却把这些方法定义为实

用的函数，并且还让它们同jQuery的术语保持一致。这就是我们将要在本节中应用它们的方式。

$.get()函数接受的参数和load()方法一样：请求的URL，一个请求参数的hash，以及一个在请求完成后执行的回调函数。在使用这个函数时，因为并没有封装的对象来自动地把响应填入其中，所以最好是明确指定可选参数的回调函数。这是在请求完成后处理事务的主要手段。

还必须注意的是，如果没有传递请求参数，还可以指定回调函数作为函数的第二个参数。

在内部，jQuery用了一些JavaScript的技巧来保证取得的是正确参数。

下面是用这个实用的函数重写的showItemInfo()：

```
function showItemInfo() {
  $.get('fetchItemData.jsp',
        {itemId: $(this).val()},
        function(data) {
          $('#itemData').empty().append(data);
        }
  );
}
```

除了使用$.get()函数外，对之前的代码所做的另一个修改是增加了一个回调函数作为第三个参数。我们可以用它来把返回的HTML插入到itemData元素中。

于是，我们用另外两个封装器方法：empty()，它用来清除封装的DOM元素；append()，它用来给封装的元素增加用data参数传递给回调函数的HTML片段。

现在，我们已经准备好要在文档加载后给<options>填入从服务器获得的数据。这里，我们将采用JavaScript hash对象的格式，从服务器获得原始数据。虽然我们可以返回XML数据，但是为了使JavaScript代码便于处理，将采用JSON。

jQuery再次用一个实用的函数帮了我们的大忙，它非常适合这次的任务：$.getJSON()。这个函数接受已经非常熟悉的三个参数：一个URL，一个请求参数的hash，以及一个回调函数。

用$.getJSON()所带来的优势是，回调函数调用时将获得已经执行过的JSON结构体。我们不再需要去对返回的响应做任何的执行。多么方便啊！

用这个实用的方法，需要在文档的ready()句柄中加入下面这一行：

```
$.getJSON('fetchItemList.jsp',loadItems);
```

一个fetchItemList.jsp的JSP页面被作为URL，并且提供了loadItems()（接下去会看到它的定义）函数作为回调。注意，因为我们不需要传递任何请求参数，所以就可以简单地忽略对象hash，并提供回调函数，把它作为第二个参数。

LoadItems()函数定义如下：

```
function loadItems(itemList) {
  if (!itemList) return;
  for(var n = 0; n < itemList.length; n++) {
    $('#itemsControl').get(0).add(
      new Option(itemList[n].name,itemList[n].id),
                 document.all ? 0 : null
    );
  }
}
```

❶ 调用回调函数时使用执行过的JSON结构

定位Select元素

❷ 增加新的option

回忆一下，$.getJSON()函数在调用回调函数时，将传递已经执行过的JSON响应❶。在这个解决方案中，fetchItemList.jsp返回的响应包括：

```
[
{id:'3',name:'BBQ Sauce'},
{id:'5',name:'Mustard'},
{id:'7',name:'Cheese'},
{id:'2',name:'Cole Slaw'},
{id:'4',name:'Lunch Meat'},
{id:'8',name:'Iced Tea'},
{id:'6',name:'Hot Sauce'},
{id:'1',name:'Milk'}
]
```

当调用回调函数时，请求字符串已经被转换成一个JavaScript对象数组，其中每个对象都包含了一个id和一个name属性。每个对象还将被用来构造一个新的<options>元素，并用一种熟悉的方式把它加入到select控件中❷，这在4.2.1节的解决方案中已经出现过。

为了给<select>元素增加选项，我们需要这个控件DOM元素的引用。我们可以用document.getElementById()或$()，但是，我们还是选择了jQuery的方式，即get()封装器方法：

```
$('#itemsControl').get(0)
```

当没有传递任何参数时，这个方法将返回所有匹配CSS选择器的元素数组。如果，只想要其中的一个，那么就可以指定一个从0开始的索引作为参数。这里，因为使用了一个id，所以就知道只有一个单独匹配符合这个选择器，因此，就指定0号索引来返回第一个匹配的元素。

代码清单4-7列出了整个页面的代码，其中对上个解决方案进行修改的地方够用高亮显示了出来。

代码清单4-7　用jQuery来获得更多晚餐的食物

```
<html>
  <head>
    <title>What's for dinner?</title>
    <script type="text/javascript" src="jquery.js"></script>
    <script type="text/javascript">
      $(document).ready(function(){
        $.getJSON('fetchItemList.jsp',loadItems);
        $('#itemsControl').change(showItemInfo);
      });

      function loadItems(itemList) {
        if (!itemList) return;
        for(var n = 0; n < itemList.length; n++) {
          $('#itemsControl').get(0).add(
            new Option(itemList[n].name,itemList[n].id),
                       document.all ? 0 : null
          );
        }
      }

      function showItemInfo() {
        $.get('fetchItemData.jsp',
              {itemId: $(this).val()},
```

```
                function(data) {
                  $('#itemData').empty().append(data);
                }
            );
        }
    </script>
  </head>

  <body>
    <form style="float:left">
      <select id="itemsControl" name="items" size="10">
      </select>
    </form>
    <div id="itemData" style="float:left"></div>
  </body>
</html>
```

3. 讨论

这一节，让我们领略了jQuery在DOM操作和遍历以及Ajax请求领域上更多出色的性能。我们了解了jQuery的$.get()函数，它使得我们能够轻松地用HTTP GET方法来发送Ajax请求。另一个有着同样函数签名的$.post()通信函数，可以同样轻松地通过Ajax提交POST请求。因为两个函数都使用同样的参数签名——清晰的请求参数hash。这样，就可以简单地在我们喜欢的任何一种HTTP方法之间进行切换，不必为请求参数应该如何编码的细节而烦恼。

另外一个Ajax实用函数$.getJSON()可以让我们极其轻松地使用服务器端格式化并返回的JSON符号。在调用这个操作的回调函数时，JSON字符串已经被执行过了，这就让我们能够从处理JavaScript eval()函数的不可知行为中解脱出来。

在我们希望能够获得更多控制权且能够清楚Ajax请求的情况下，jQuery提供了一个叫做$.ajax()的通用实用函数。在线文档对如何使用底层的实用函数提供了更多细节。

我们还看到了一些强大的DOM操作封装器方法，比如get()，empty()，val()以及append()，这些都让作为Ajax网页开发者的我们能够轻松地操作页面的DOM。

在这里，几乎不可能完整展现jQuery性能的深度。例如，比如无法探索effects的API，它提供了fading、sliding、flashing、hovering等众多特效。甚至拥有自己提供动画的能力。我们鼓励你访问http://jquery.com/来获得jQuery的更多信息，以及获知它是如何帮助编写强大的Ajax应用的。

此外，高级的开发者可能会对jQuery的插件程序API感兴趣。这个API是jQuery众多强大性能中的一个，因为它可以让任何人都可以立刻扩展这个工具集。更多信息请详见http://docs.jquery.com/Plugins/Authoring。

现在，有些事物已经完全不同了。让我们开始研究第四个框架，它将用一种全新的角度来处理异步请求的问题。

4.4 DWR

DWR的全称是Direct Web Remoting，它采用Ajax作为传输机制，可以执行从客户端JavaScript到服务器端Java代码的远程过程调用。本章之前所介绍的所有程序库，都是一种提交请求给服务

器端资源然后接受响应的模式，但是DWR的模式就有些不同了。

远程过程调用（RPC）机制允许本地代码调用存在于远程服务器的对象上的方法，就好象远程对象就在本地一样。通常，RPC的工作方式（在这里我们掩盖了许多不那么重要的细节）是通过创建一个本地代理接口模仿远程方法的签名（通常称为**存根**[stub]）。本地代码调用本地接口，本地系统上的RPC代理汇集输入数据的一个RPC代理，并且执行必要的网络处理来传递这些信息给在远端系统上运行着的副本。

远端代理把数据解包成合适的格式，并调用真正的远端方法。当方法返回时，任何返回数据再次汇集，并发送回本地代理，并轮到它来返回从代理到远端方法原始调用者的控制权。

从本地调用代码的角度来看，事实上，在代理机制背后进行数据的汇集以及网络通信都是隐藏的。同样地，远端方法也不知道自己被远端调用了。

DWR实际不是一个纯正的RPC实现，主要有两个原因：

- JavaScript中使用的代理方法签名并不和远端的Java方法一致。
- 调用不是同步的。与其他Ajax机制类似，所有远端方法的调用都是异步的，并且处理函数会在远端方法完成后被调用。

但是，在任何对于是否是纯正实现的学术争论之外，DWR为那些宁愿考虑方法调用而不愿设计传统HTTP请求响应周期的人提供了一种聪明的方式。

在http://getahead.ltd.uk/dwr/上可以下载到DWR的JAR文件。下载后，把它放到Web应用的WEB-INF/lib文件夹下。

用 DWR 来进行远程调用

DWR是作为一个servlet来实现的，它用来处理所有客户端的请求。大多数RPC实现需要使用一个预处理程序来生成客户端和服务器端的存根。另一方面，DWR通过一个看起来像普通JavaScript文件引用的servlet引用来动态地生成客户端存根。

但在开始之前，需要进行一些设置：必须在Web应用的部署文件（web.xml）中声明并映射DWR的servlet。

可以按照下面这样在部署文件中增加一个<servlet>元素来声明servlet：

```
<servlet>
  <servlet-name>DwrServlet</servlet-name>
  <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
  <init-param>
     <param-name>debug</param-name>
     <param-value>true</param-value>
  </init-param>
</servlet>
```

那个debug初始参数是可选的，但是在开发过程中是非常有用的。在实际部署中就不要包括它了，因为这样将允许访问者试图获取本不该共享的代码信息。稍后，你将看到这个参数的作用。

这个servlet被映射到一个URL上。

```
<servlet-mapping>
  <servlet-name>DwrServlet</servlet-name>
  <url-pattern>/dwrserver/*</url-pattern>
</servlet-mapping>
```

它映射所有在Web应用中以/dwrserver开始的URL到这个DWR servlet上。

还剩下一个设置步骤，但这取决于我们到底要如何使用DWR。所以，先来看一个特定的示例。

1. 问题

在包含客户信息表单的页面上，假如客户的名字在数据库中是唯一的，那么我们想要自动地为客户填充地址信息。当然，我们希望能够异步地执行，而并不需要重新加载页面。

2. 解决方案

第一件要做的是新建一个Java类。我们可以根据给定的姓名来查询以便获得用户信息。显然，这样一个类的真实实现需要执行一个数据库查询，但是为了这个示例的目的，我们将仅在API背后硬编码一个虚假的实现。它可以假定，我们当前只拥有单独一个叫做Bill Moody的客户。

```
public class CustomerFactory {

  public Customer findByName(String firstName,String lastName) {
    if ("Bill".equalsIgnoreCase(firstName) &&
        "Moody".equalsIgnoreCase(lastName)) {
      return new Customer("Bill", "Moody", "123 Nowhere Lane",
                          "Austin", "TX", "USA", "78701");
    } else {
      return null;
    }
  }
}
```

findByName()方法返回查找到的客户的一个实例，或者在没有找到的情况下返回null。在一个真实的实现中，当发现有多个客户拥有同一个名字，即不足以来唯一地识别一个特定客户时，它也会返回null。

为了在DWR引擎中定义这个类，我们在WEB-INF文件夹中创建一个名称为dwr.xml的XML文件。

```
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr10.dtd">

<dwr>
  <allow>
    <convert converter="bean" match="org.bibeault.*"/>

    <create creator="new" javascript="CustomerFactory">
      <param name="class"
             value="org.bibeault.aip.dwr.CustomerFactory"/>
    </create>
  </allow>
</dwr>
```

❶ 指定bean类

❷ 定义JavaScript类

对于在这个文件中需要进行的所有可能设置，都应该参考DWR文档。但本质上来说，是要告诉DWR，我们想要它来转换包中的那个bean类（比如由findByName()方法返回的那个）❶。之后，定义需要远端访问的类，连同构造的方法以及作为客户端存根的JavaScript对象名字❷。

现在有趣的事情发生了！

启动Web应用，然后不加任何相对路径来访问DWR的servlet，假设Web应用的项目路径为/aip.chap4，那么URL就该是http://localhost:8080/aip.chap4/dwrserver/。因为我们通过servlet中的初始(init)参数启用了debug模式，所以DWR显示一个测试页面来动态地显示关于DWR环境的一些有用的信息，就像图4-6显示的那样。

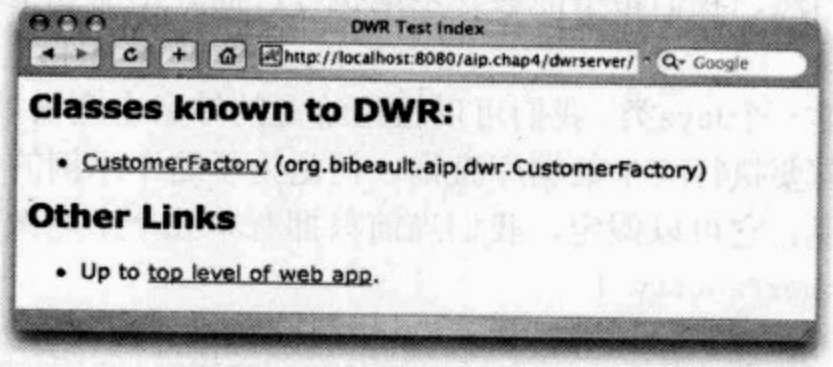


图4-6　DWR测试页面显示：已知的类列表

这个页面显示了所有映射后的类。点击CustomerFactory链接来显示一些关于这个类的有用信息，就像图4-7展示的那样。

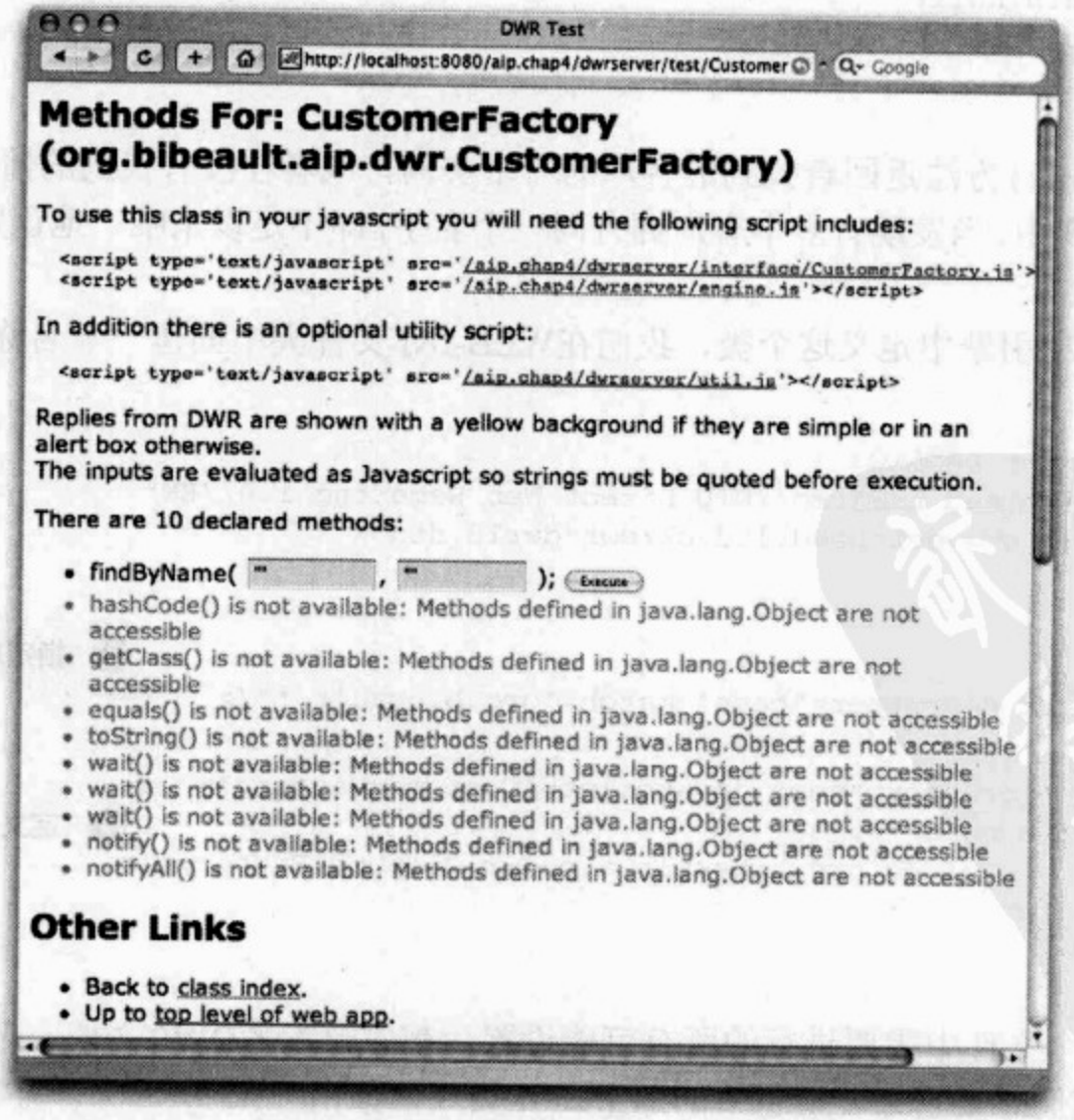


图4-7　DWR测试页面显示：类信息

这个页面不仅显示了为了使用这个类而应该包含的<script>元素，还显示了已经声明的方法，甚至还允许我们用简单的数据来测试它们。在开发期间，这种特性的作用是不言而喻的。

如果观察一下<script>元素的URL，会发现即使它们看起来像是JavaScript文件的引用，但实际上它们是在web.xml中声明的DWR servlet的调用。.js文件并不真正存在于文件系统中，而是通过引用由DWR servlet动态生成。

如果我们点击CustomerFactory.js文件的那个链接，那么就会看到图4-8所显示的那样，也就是为映射类中的方法所创建的本地存根。

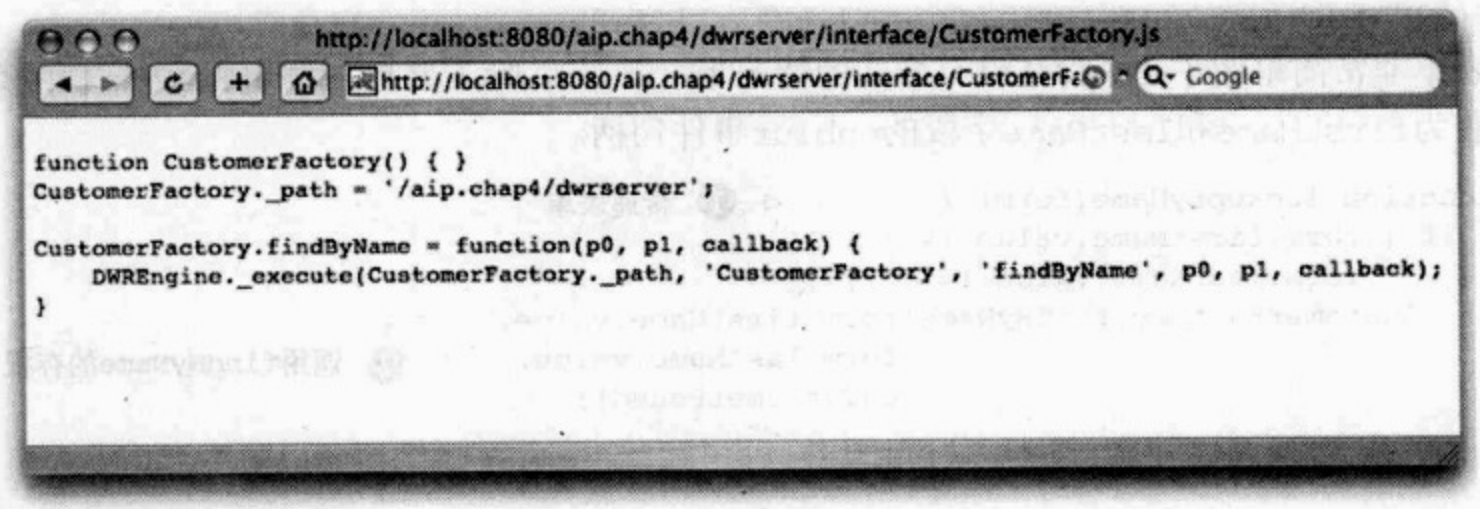


图4-8 DWR动态生成JS纯根

那么，现在已经准备好要真正编写页面了。剪切并粘贴图4-7显示的来自DWR生成的那个页面上的script元素，把它门加入到页面的<head>元素中：

```
<script type='text/javascript'
  src='/aip.chap4/dwrserver/interface/CustomerFactory.js'></script>
<script type='text/javascript'
  src='/aip.chap4/dwrserver/engine.js'></script>
```

用于捕获客户数据的表单的编码如下：

```
<form name="customerForm" action="/doSomething">
  <div>
    <label>First name:</label>
    <input type="text" name="firstName"
      onblur="lookupByName(this.form);"/>
    <label>Last name:</label>
    <input type="text" name="lastName"
      onblur="lookupByName(this.form);"/>
  </div>
  <div>
    <label>Address:</label>
    <input type="text" name="address"/>
  </div>
  <div>
    <label>City:</label>
    <input type="text" name="city"/>
    <label>State/Province:</label>
```

```
      <input type="text" name="state"/>
    </div>
    <div>
      <label>Postal Code:</label>
      <input type="text" name="postalCode"/>
      <label>Country:</label>
      <input type="text" name="country"/>
    </div>
    <div>
      <input type="submit" value="OK"/>
    </div>
  </form>
```

表单非常简单明了（尽管在没有样式的情况下，很丑），除了要把lookupByName()函数的调用设置为firstName和lastName字段的onblur事件句柄。

```
function lookupByName(form) {                    <-❶ 传递表单
  if ((form.firstName.value != '') &&
      (form.lastName.value != '')) {
    CustomerFactory.findByName(form.firstName.value,
                               form.lastName.value,
                               onCustomerFound);
  }
}
```

❷ 调用findNyName的存根

在这个处理函数里，传递表单❶，并检查firstName和lastName是否已经全部填入。是的话，远端findByName()方法的存根就会被调用❷。

注意，这个存根很像远端方法，但是并不完全一样。首先，调用是异步的，因此这个函数并没有返回值。其次，方法的签名中加入了一个额外的参数来指定在异步调用结束后需要调用的回调函数。其中，将把远端方法的返回值作为这个函数单独的参数传递进去。

回调函数的代码如下：

```
function onCustomerFound(customer) {
  if (customer != null) {
    var form = document.customerForm;
    form.address.value = customer.address;
    form.city.value = customer.city;
    form.state.value = customer.state;
    form.postalCode.value = customer.postalCode;
    form.country.value = customer.country;
  }
}
```

调用回调函数将得到远端方法的返回值。我们简单地对其进行检查以保证它不会是null（回忆一下，当客户并不能唯一被识别时，远端方法将返回null）。如果不是null，那么我们用传递过来的对象的值来填充表单。

DWR已经汇集了来自Customer类的数据，并创建一个拥有同在Customer类中定义的每个JavaBean属性一致的JavaScript对象。在浏览器中，浏览这个示例，并输入客户的名字，结果显示在图4-9中。代码清单4-8列出了完成后的整个页面。

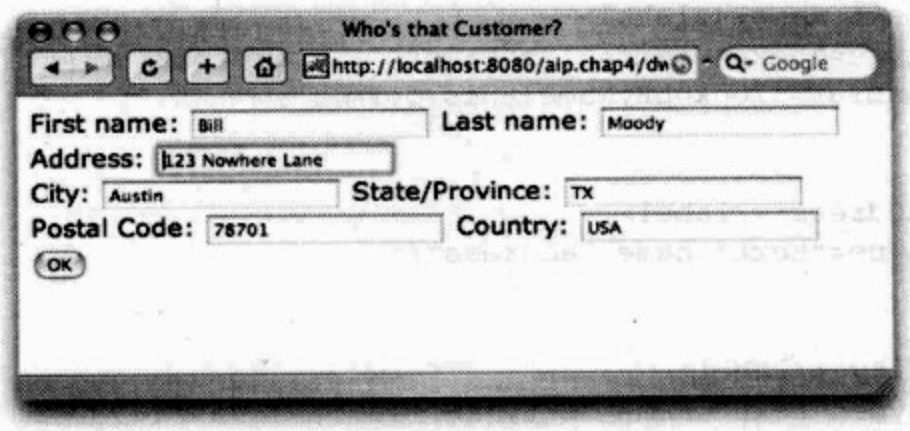


图4-9 Bill Moody的页面

代码清单4-8 用DWR进行远程调用

```
<html>
  <head>
    <title>Who's that Customer?</title>
    <script type='text/javascript'
      src='/aip.chap4/dwrserver/interface/CustomerFactory.js'>
    </script>
    <script type='text/javascript'
      src='/aip.chap4/dwrserver/engine.js'></script>
    <script>
      function lookupByName(form) {
        if ((form.firstName.value != '') &&
            (form.lastName.value != '')) {
          CustomerFactory.findByName(form.firstName.value,
                                     form.lastName.value,
                                     onCustomerFound);
        }
      }

      function onCustomerFound( customer ) {
        if (customer != null) {
          var form = document.customerForm;
          form.address.value = customer.address;
          form.city.value = customer.city;
          form.state.value = customer.state;
          form.postalCode.value = customer.postalCode;
          form.country.value = customer.country;
        }
      }

    </script>
  </head>

  <body>
    <form name="customerForm" action="/doSomething">
      <div>
        <label>First name:</label>
        <input type="text" name="firstName"
               onblur="lookupByName(this.form);"/>
```

```
      <label>Last name:</label>
      <input type="text" name="lastName"
             onblur="lookupByName(this.form);"/>
    </div>
    <div>
      <label>Address:</label>
      <input type="text" name="address"/>
    </div>
    <div>
      <label>City:</label>
      <input type="text" name="city"/>
      <label>State/Province:</label>
      <input type="text" name="state"/>
    </div>
    <div>
      <label>Postal Code:</label>
      <input type="text" name="postalCode"/>
      <label>Country:</label>
      <input type="text" name="country"/>
    </div>
    <div>
      <input type="submit" value="OK"/>
    </div>
  </form>
</body>
</html>
```

3. 讨论

本节使我们领略了DWR工具集的少部分能力。与其他看过的Ajax工具集不同，DWR抽象了请求响应周期，而提供了一个类RPC的方法来调用服务器上的Java函数。

虽然我们的示例很简单，但也能预想到针对各种客户端任务，如何来使用这样的能力，而并不仅限于数据的查询。想象一下，例如，当执行由服务器端提供的字段数据验证时，这些能力的价值。

显然，DWR仅仅只是对支持servlet引擎的Java Web引用起作用。使用其他服务器端机制的开发者或页面设计者只能使用坚持传统请求响应周期的工具集。

4.5 总结

在本章中，我们介绍了许多开源的程序库，它们让我们可以在Web应用中轻松运用Ajax，或通过Ajax在客户端浏览器和服务器之间进行异步通信。

其中一些程序库，如Dojo工具包和Prototype，对Ajax调用做了一层薄薄的包装，使得代码更容易编写和维护。另一些程序库，如jQuery，则提供了丰富的方法，让我们能够轻松执行一些最常用的Ajax交互。还有一些，如DWR，通过使用Ajax作为传输机制提供了一个与典型的HTTP请求响应周期不同的模式。这些程序库只是互联网上众多自由可用的程序库中的几个代表。

基于时间、空间以及其他实际条件的考量，我们在此无法研究更多的程序库。如果你还意犹未尽，这里列出了其他一些程序库可供探索：

- Scriptaculous（http://script.aculo.us/），这是另一个基于Prototype的JavaScript库，和Ruby on Rails项目有着紧密联系。
- Rico（http://openrico.org/），一个用于创建富因特网应用的JavaScript库，其中包括了对拖放、Ajax和电影效果[①]的支持。
- Echo2（http://www.nextapp.com/）[②]，一个Web应用开发平台，目标是让Web应用也能具有如桌面客户端那样丰富的功能。
- Sarissa（http://sarissa.sourceforge.net/），一个以XML和XSLT为中心的程序库，为各个浏览器原生的XML API提供了跨浏览器的包装器。
- Sajax（www.modernmethod.com/sajax/），一个JavaScript Ajax框架，用于使用PHP、Perl或Python所开发的Web应用[③]。
- ThinkCAP JX（www.clearnova.com/），一个RAD开发工具，可用于开发带有Ajax特性的业务应用。

……还有很多。这个列表只是冰山一角。

所以不要局限于这里，开阔你的视野吧。这是一个令人兴奋、发展迅速的领域，到本书付梓之时，可能会有更多新的令人兴奋的工具包[④]已在影响着Web了。

① 指像淡入淡出等电影中常用的过场切换效果。——译者注

② 已推出新一代的Echo3，请访问http://echo.nextapp.com/site/echo3。——译者注

③ 在概念上，Sajax类似于DWR，只是它不是专门针对Java的，而是用于许多其他服务器语言，如ASP、Cold Fusion、Io、Lua、Perl、PHP、Python以及Ruby等。——译者注

④ 确实如此，比如MooTools、Ext等就是例证。——译者注

Part 2 第二部分

Ajax 最佳实践

第二部分由9章组成，每一章都会选取一个对于Ajax程序来说至关重要的Web开发领域进行深入探讨。

首先是第5章，该章对浏览器的事件处理机制做了深入的介绍。我们将讨论各种事件处理模型，并说明如何为各种事件类型建立事件处理器。我们还列出了浏览器的兼容问题以及能简化跨浏览器编程的方法。

第6章，利用前一章学到的事件处理知识，我们开发出了对表单数据输入值进行有效性验证的方法。我们分析了一个有效性验证框架，你还将学习如何接管表单的提交操作以避免刷新整个页面。

第7章的主题是应用的内容导航。菜单（menu）、树（tree）、可折叠控件（accordion control）、选项卡（tab）和工具栏（toolbar），均在我们的考察之列。通过本章的示例代码，我们还瞅了一下 OpenRico 和 qooxdoo 库。

有些用户执意点击浏览器的“后退”和“刷新”按钮控件，第8章介绍如何处理这些棘手问题。我们描述了隐藏这些控件的技巧，同时也介绍了在不隐藏它们时的应对策略。

第9章专注于为Web应用添加拖放功能。借助Scriptaculous库的支持，我们可以用拖放来对列表进行内容分类，然后研究了一个简单的拖放式购物车。本章最后介绍了ICEfaces库。

第10章我们考察了优使性（usability）[①]，尤其是常与Ajax应用相关的那些问题。我们提出了一些应对延时问题的策略，并讨论了如何缓解用户的挫折感。

第11章介绍了客户端状态维护，数据缓存及预取，以及有关状态管理的其他主题。

第12章考察了众多生机勃勃的Web服务开放API，例如Yahoo! Maps、Yahoo! Geocoding、Yahoo! Traffic、Google搜索服务以及Flickr图片服务，等等。我们设计了一种方法以规避讨厌的浏览器跨域安全限制，然后你会学习如何通过Ajax发起RESTful请求，访问这些激动人心的服务。

最后，第13章结合运用所有这些内容，利用Yahoo!和Flickr的开放API，创建了一个完整的“混搭式”Web应用。

---

① “usability”一词也常翻译为“易用性”。——编者注

第5章 事件处理

5

本章内容

- 浏览器事件处理模型
- 常用事件类型
- 如何简化事件处理
- 事件处理的实际应用

HTML应用常呆滞而乏味，幸好这样的时代现已终结，因为Ajax让我们能创造出高度互动的Web应用。对用户操作，无论是点击按钮、输入文本，还是简单地动动鼠标，它们都能流畅响应。在各种图形用户界面的历史上，用户操作通常被转译为事件，浏览器世界也是如此。当用户与网页进行交互，网页的DOM树内部就会激发事件。如果文档元素上设有与这些事件对应的事件处理器，那么一旦事件发生，事件处理器就会被调用。Ajax应用要跑起来，全依靠这些事件和事件处理器，可以说，它们是每个Ajax应用的生命线。

进入正题之前，让我们先看看如何给网页加上一个简单的事件处理器。在下面这段代码中，我们可以注意到<img>元素有一个onclick属性，这个属性定义了一个事件处理器，用户在<img>上点击鼠标，浏览器就会调用该事件处理器。

```
<html>
  <body>
    <img src="image.jpg" id="anImage" onclick="alert('Woof!');"/>
  </body>
</html>
```

在浏览器中加载这个示例，你会看到，当把鼠标移动到图像上并点击一下，就会弹出一个提示框，显示消息“Woof!”，如图5-1所示。

这个小例子说明，给DOM元素指派事件处理器还是相当简单的。

在接下来的整个章节中，我们将研究有关事件处理的几个重要问题。首先考察在现有的各种事件模型下定义事件处理器的不同方式。我们会看到，在不同浏览器平台上定义事件处理器都有哪些差异，然后看一看如何能让这一过程变得跨浏览器通用。我们也将了解事件处理器在被调用时可以得到哪些与事件相关的信息。我们将讨论事件冒泡（event bubbling）和事件捕获（event capturing）的概念，它们指明了事件是如何在DOM中传播的。我们还会好好看一下那些常用的事件类型。最后，我们会准备几个现实世界的例子，来演示如何在实际应用中运用这些理论知识。

图5-1 狗吠汪汪汪

5.1 事件处理模型

之前我们已经看到，声明一个简单的事件处理器有多简单，或许你会认为编写事件处理器就是小菜一碟：在事件处理器属性里写点脚本代码，然后在事件发生时浏览器就会执行代码。还有比这更简单的吗？不过，如果真是如此简单，我们何必要写上整整一章呢？

在现实世界里，为了在Web应用中使用事件，我们需要对付三种不同的事件处理模型：

- 基本事件模型，也被非正式地称为DOM Level 0事件模型，简单直接且跨平台。
- DOM Level 2事件模型，它更灵活，遵从标准的浏览器如Firefox、Mozilla和Safari等支持它，但是IE不支持。
- Internet Explorer事件模型，功能上类似于DOM Level 2事件模型，但顾名思义，是IE浏览器专有的。

我们先看一下在基本模型下如何注册和编写事件处理器，然后再看两种高级模型。

5.1.1 基本的事件处理注册

本章一开始就以一个小例子展示了基本模型或叫做DOM Level 0模型的用法。这是最古老的事件处理方式，具有不错的平台无关性（尽管不是完全无关）。它很好地满足了基本的事件处理需求。并且我们会看到，它也并没有完全被高级事件模型取代，通常会与高级模型结合使用。

在基本模型下有两种方式可以指派事件处理器：

- HTML标记内嵌法，用到HTML元素上的相关事件属性（attribute）。
- 脚本操作法，用到DOM元素对象上的相关属性（property）。

还记得前面小示例中的<img>元素吗？

```
<img src="image.jpg" id="anImage" onclick="alert('Woof!');"/>
```

这就是使用标记内嵌法的一个例子。

以onclick属性值为函数体的匿名函数，即是单击（click）事件的处理器。这样的方式很简单直接，但也有缺点。

构建Web应用的最佳设计实践就是要分离应用程序的外观（HTML）和行为（JavaScript）。内嵌法定义事件处理器的方式违背了这一准则。因此一般来说，应限制或避免使用内嵌式的事件处理器声明。

相比之下，更好的办法是通过脚本操作来给DOM元素附加事件处理器。近年来，浏览器的DOM逐步趋向标准化，JavaScript开发者对它也更加熟悉了，因此脚本操作法已越来越普及。所有的DOM元素都有一些对象属性，表示该元素上可能激发的事件，例如onclick、onkeyup、onchange等。

下面我们就把之前的示例代码重构为一个完整的HTML文档，并改以脚本编程方式设定图像上的onclick事件处理器，如代码清单5-1所示。

代码清单5-1　通过脚本指派事件处理器

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {                                    ❶ 声明页面onload事件处理器
        document.getElementById('anImage').onclick = function() {
          alert('Woof!');
        }
      };
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>                               ❷ 声明脚本无关的图像元素
  </body>
</html>
```

如果你已经从www.manning.com/crane2下载了本章对应的源代码，可以在文件chap5/listing-5.1.html找到该HTML文档。

这个示例和我们前面的示例在功能上是一样的，但其代码显得更为老练。通过将脚本从<body>❷元素中提取出来并放入<head>下的<script>元素，我们把行为从外观中分离了出来。注意，我们把原来的代码放到了另一事件处理器函数：页面的onload事件处理器❶。

虽然看上去是更多的代码也只实现了与之前示例相同的功能，但这种方法不仅改善了页面结构，而且提供了更好的灵活性。

灵活性的一个重要方面是控制何时安置（establish）和移除事件处理器。标记内嵌法限定了事件处理器将在页面加载时安置①，并且在页面整个存续期间一直存在，无法移除。而脚本操作法让我们可以在任何需要的时候安置事件处理器。在代码清单5-1的示例中，我们选择在页面加载时安置事件处理器，不过我们也可以等到某些其他事件发生时再进行安置。此外，只需将事件属性设置为null，我们就可以在任何时候移除事件处理器，这是标记内嵌法所办不到的。

在这个示例中，我们使用了无名的函数源文本（anonymous function literal）来创建事件处理

① 准确来说，浏览器会一边加载HTML文档一边将其解析为DOM树。当一个带有事件属性的元素标记解析完成，并在DOM树中产生了对应的元素和属性对象后，就已经建立起了事件处理器。——译者注

器——毕竟，若非必须，何必单独创建一个有名函数？不过有一点很重要，如果要指派一个有名函数为事件处理器，记得不要在函数名后画蛇添足地加上括号。我们要将属性值设定为函数的引用，而不是函数的调用结果！例如，下面的代码会调用sayWoof()方法，而不会将其设为事件处理器。你可不要犯这样的低级错误哦。

```
element.onclick = sayWoof(); //错误!

element.onclick = sayWoof; //正确!
```

虽然DOM Level 0事件模型还算灵活，但也有很大的局限性。比如，不太容易让多个JavaScript方法串联响应单个事件。

那么，怎样给一个事件注册两个处理函数呢？先来看一个有点幼稚的尝试，我们把前面的示例修改一下，给<img>元素的onclick属性上加上两个JavaScript事件处理器。见代码清单5-2（源代码中的chap5/listing-5.2.html文件），新添加的代码以粗体显示。

代码清单5-2 试图指派两个事件处理器

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').onclick = function() {
          alert('Woof!');
        }
        document.getElementById('anImage').onclick = function() {
          alert('Woof again!');
        }
      };
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
  </body>
</html>
```

运行这段代码，我们会看到只有一个提示“Woof again!”，显然只有第二个处理器被调用了。这是意料之中的。onclick不过是<img>元素上的一个属性，像其他属性一样，如果对其多次赋值，后面的值定会覆盖前一次的值。

这就提出了这样一个问题：到底可不可能让多个函数响应单个事件？就DOM Level 0事件模型来说，要给同一个事件注册多个事件处理器，靠直接对元素的事件属性赋值肯定是行不通的。当然，我们可以把多个函数的代码合并成一个函数，或者可以写一个函数依次调用其他函数。但这样的方式过于呆板，无法应对复杂情况。如果没有其他方法可用，那比较成熟的方案是采用观察者（Observer）模式（也称作发布/订阅模式，Publisher/Subscriber），我们注册一个处理器作为观察者，其他函数则注册为订阅者。

所幸，我们并非必须求助于这样高深的技巧。使用浏览器高级事件处理模型我们就能注册多个事件处理器——尽管其方式在不同浏览器中并不一致。让我们看看怎么做吧。

5.1.2　高级事件处理

在理想世界里，为一种浏览器写的代码也能完美地运行在其他浏览器上，可惜我们生活在现实世界里。当进入到高级事件模型时，我们就需要应对浏览器的分歧。一方面，是W3C（World Wide Web Consortium，万维网联盟）的方式，另一方面，则是微软的方式。让我们先看看标准的W3C方式。

对于遵从DOM Level 2事件模型[①]的浏览器来说，每个DOM元素上都有叫做addEventListener()的方法，可以调用它来给元素添加事件处理器。该方法有三个参数：第一个参数是字符串，用来声明事件的类型，第二个是所要执行的事件处理器函数（也称作监听器，listener），最后一个是布尔值，表示是否启用事件捕获。我们会在讨论事件传播机制时解释最后一个参数，此刻我们暂时设其为false。

事件类型参数是一个字符串，包含所要观察的事件类型的名字。事件属性名去掉前缀on即为事件类型名，例如，click和mouseover。

让我们按W3C方式改写清单5-2的示例代码，将基本方式（设置元素上的onclick属性）改为调用addEventListener()方法，结果如代码清单5-3（修改部分以粗体显示）所示。

代码清单5-3　按W3C方式添加事件处理器

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').addEventListener(
          'click',
          function() { alert('Woof!'); },
          false);
        document.getElementById('anImage').addEventListener(
          'click',
          function() { alert('Woof again!'); },
          false);
      };
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
  </body>
</html>
```

当浏览器加载此页面后，点击图片，两个提示框都显示出来了，这里并不需要借助“容器函数”[②]来串联两个事件处理器。注意上述代码在Internet Explorer下无法工作，本节稍后我们将看到IE如何以其专有的方式实现高级事件处理。

另一点需要注意的是，如果像我们示例中展示的那样，在某个元素上为同一事件安置多个事

① 该W3C标准位于http://www.w3.org/TR/DOM-Level-2-Events/。——译者注

② 即前文提到的container function，构造一个特定函数用来调用多个其他函数。——译者注

件处理器，DOM Level 2事件模型并不保证这些事件处理器始终按某种特定顺序执行。虽然通过测试可以观察到事件处理器是按照安置时的顺序逐一调用的，但是并不保证总是如此，所以让代码依赖于安置顺序是不明智的[①]。

如果要移除元素上的事件处理器，可以调用DOM元素上的removeEventListener()方法。

微软专有的事件附加（attach）方法，在概念上也是类似的，但是实现上不同。它使用一个名为attachEvent()的方法为DOM元素安置事件处理器。该方法有两个参数：事件名称和所要执行的事件处理器。与addEventListener()方法所用的事件类型不同，attachEvent()方法需要的是包含on前缀的事件属性名称。

在掌握了这些知识之后，让我们再次修改示例代码。我们将在代码中加入检测机制，并使用当前浏览器所支持的方法。更新后的代码如代码清单5-4所示（在本章源代码中可以找到），变更部分同样以粗体显示。

代码清单5-4 以另一种方式添加事件处理器

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        if (document.getElementById('anImage').attachEvent) {
          document.getElementById('anImage').attachEvent(
            'onclick',
            function() { alert('Woof!'); });
          document.getElementById('anImage').attachEvent(
            'onclick',
            function() { alert('Woof again!'); });
        }
        else {
          document.getElementById('anImage').addEventListener(
            'click',
            function() { alert('Woof!'); },
            false);
          document.getElementById('anImage').addEventListener(
            'click',
            function() { alert('Woof again!'); },
            false);
        }
      }
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
  </body>
</html>
```

onload事件处理器中的首行代码检测应该使用哪个方法来添加事件。注意这种测试被称为对象检测（object detection）。它并不测试是哪一种浏览器，而是检测元素上是否存在attachEvent()

① 新的DOM3事件模型规范（目前仍是草案）对此做了修改，明确了事件处理器将按照注册的顺序执行。——译者注

方法。如果该方法存在，我们就使用它，否则我们就使用W3C标准方法。

当在浏览器中显示这个页面时，只要浏览器支持其中一种机制，代码就保证能运行。不过在Internet Explorer中，当我们点击图片，就会注意到情况有些不同，两个提示以相反的次序显示！不过也有可能不是[①]。之前已经说过，在使用DOM Level 2事件模型时，我们无法确知它们的显示次序。同样地，attachEvent()方法的定义[②]也明确声明，在同一元素上附加的同一事件类型的多个事件处理器会以任意次序触发。

以上我们探索了如何跨不同浏览器注册事件处理器。你看到了内嵌法的便利，但也了解了其缺点。DOM Level 0模型的事件处理器的注册方式可以跨多种浏览器通用，但是缺乏一种自动串联多个事件处理器函数的方法。然后我们展示了一种更高级的附加事件处理器的方式，使用DOM Level 2或Internet Explorer事件模型。这种方式很灵活，允许我们动态附加、卸除（detach）和串联事件处理器，不过因为有跨浏览器的问题，我们必须通过对象检测来调用当前浏览器所支持的方法。幸运的是，一些框架提供了统一的事件处理接口，这让我们可以更容易地编写能在框架支持的所有浏览器中通用的代码。在5.3节中，我们将看到如何使用Prototype库来达成这一目标。

在引入框架之前，我们还是得先打好事件处理的基础。在下面几节中，我们会详细介绍在事件处理器中如何访问事件的相关信息，以及事件如何在DOM树中传播。

5.2　Event对象与事件传播机制

要处理浏览器中的事件，我们还需理解另外两个重要概念：Event对象和事件传播方式。Event对象，也就是Event类的实例，通过它可以获取事件的相关信息。事件传播机制则定义了一个事件传递到各个事件观察者的顺序。我们首先来搞定Event对象。

5.2.1　Event对象

当触发某个事件时，会产生一个Event类的实例，它包含描述事件的若干属性。在事件处理器中，我们通常会访问该Event对象并读取想要的属性，例如事件发生于哪一个HTML元素，或者（在鼠标事件中）用户点击的是哪个鼠标键。像事件处理的其他领域一样，在事件处理器中访问事件对象实例，各浏览器所采用的方式不尽相同。

对于遵从标准的浏览器来说，浏览器在执行事件处理器时，会将该Event对象实例作为第一个参数传入事件处理器函数。在Internet Explorer中，则可通过窗口对象的event属性来读取该对象实例（也可将event看作一个全局变量）。

下面我们研究一下怎么处理Event对象。注意，因为老是弹出提示框有点烦人，所以我们改为将诊断信息输出到图片下方的 元素中，见代码清单5-5。

代码清单5-5　获取事件对象实例

```
<html>
  <head>
```

① 根据实际测试来看，在两个事件处理器的情况下似乎总是相反顺序，当有更多事件处理器时则会按照一种奇特的顺序，然而究竟按照怎样的规则排序，恐怕只有IE的开发团队才知道。——译者注

② 该文档位于：http://msdn2.microsoft.com/en-us/library/ms536343(VS.85).aspx。——译者注

```
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').onclick =
          function(event) {
            if (!event) event =          ❶ 获取event对象实例
              window.event;
            var target =                 ❷ 获取事件目标元素的引用
              event.target ? event.target : event.srcElement;
            document.getElementById('info').innerHTML +=
              'I woof at ' + target.id + '!<br/>';
          }
      }
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
    <div id="info"></div>
  </body>
</html>
```

在这一示例中，我们首先检查事件处理器函数的event参数（event这个名字起得挺聪明的）是否已定义（遵从标准的浏览器会定义它），如果未定义，就拷贝窗口对象❶上的event属性，因为IE把Event对象放在那里。由此，我们确保得到一个Event实例的引用。

然后，我们要获得目标元素（target element）❷的引用，也就是事件是为哪个元素所产生的。同样地，由于IE与标准浏览器对Event类的定义不同，我们需要为特定浏览器作特定处理。

我们检查标准的target属性，如果没有定义，就使用IE专有的srcElement属性。

这真是烦人！看起来几乎事件处理的每个步骤都需要用上两种不同的做法才能让IE和支持W3C标准浏览器都能运行。

是的，事实确实如此。不过不用怕，我们已经有了解决之道。不过，首先让我们搞清楚事件传播是怎么回事。

5.2.2 事件的传播

到目前为止，我们重点介绍了直接定义于触发事件的元素之上的事件处理器，似乎只有这样的事件处理器才有意义。实际上并非如此，事件不仅传递到目标元素，也会传递到目标元素在DOM树中的所有祖先节点（ancestor）。本节，我们将看到事件是如何在DOM树中传播的，并学习怎样改变在传播途中调用哪个事件处理器——乃至如何控制一个事件的传播。

我们将首先讨论遵循DOM Level 2事件模型的浏览器中的事件是如何传播的，然后考察Internet Explorer，它所支持的特性只相当于DOM Level 2模型的一个子集。

在遵从标准的支持DOM Level 2事件模型的浏览器中，一个事件被触发后，会包含三个处理阶段（phase）。这三个阶段按先后顺序分别叫做捕获（capture）阶段、目标（target）阶段和冒泡（bubble）阶段。

捕获阶段，事件从文档节点[①]向下遍历DOM树直到目标元素。事件经过一个元素时，每一个为该种事件类型而安置到该元素上的事件处理器都会被调用——前提是该事件处理器属于捕获处理器（capture handler）。还记得我们之前略过的addEventListener()方法的第三个参数吗？如果将该参数设为true，事件处理器就会被注册为捕获处理器；如果设为false，事件处理器则被注册为冒泡处理器（bubble handler）[②]，到目前为止，我们一直是设为false的。每个事件处理器要么是捕获处理器要么是冒泡处理器，但不可能两者皆是。

事件自上向下遍历并激活所有对应的捕获处理器，一旦到达目标元素，事件传播就进入到目标阶段。在该阶段，安置于目标元素本身的相关事件处理器将被触发。如果目标元素上同时安置了捕获处理器和冒泡处理器，所有这些事件处理器都会被调用[③]。

然后事件将反向传播，从目标元素向上沿着DOM树“冒泡”至文档节点[④]，沿途经过的元素上所有为该种事件类型而安置的冒泡处理器都会被触发。这就是冒泡阶段。

一图胜千言。假设我们对示例程序的body稍作修改，将<img>元素嵌于两层 元素之中，如下：

```
<div id="level1">
  <div id="level2">
    <img src="image.jpg" id="anImage"/>
  </div>
</div>
```

当我们点击图片，点击（click）事件在DOM树中的传播路径就如图5-2所示：

现在让我们来实际考察一下，考虑代码清单5-6中的代码。

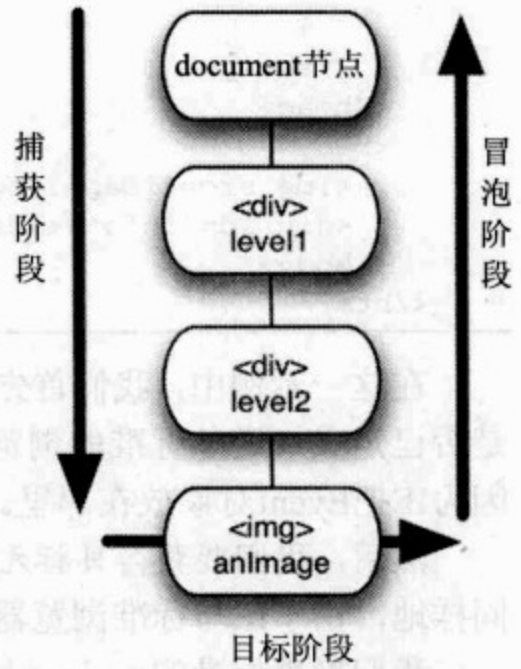


图5-2　上下遍历DOM树

代码清单5-6　创建捕获型和冒泡型事件处理器

```
<html>
  <head>
    <title>Events!</title>
```

① 原文是“...from the document root element...”，有误。捕获阶段通常从树的顶端，即文档节点开始，然后才是文档根元素。此外，对于浏览器来说，在事件的捕获阶段，事件通常会首先发送到Window对象。正在制定中的HTML5规范会对此作出明确规定。——译者注

② 如果要咬文嚼字的话，冒泡处理器应该称作“非捕获处理器”或者“冒泡兼目标处理器”，因为它也可执行于目标阶段。——译者注

③ 这一点存在问题，比如Opera浏览器就不会调用捕获处理器。有些人认为DOM Level 2事件规范在这一点上存有歧义，但如果仔细分析，可以确定DOM 2规范的意思确实是不应在目标阶段调用捕获处理器，W3C发布的测试套件（test suite）也测试了这一点，DOM 3事件规范的草案也再次明确了这一点。然而基于种种原因，所有版本的Gecko引擎（Firefox等浏览器）和最近版本的WebKit引擎（Safari等浏览器）都会在目标阶段调用包括捕获处理器在内的所有事件处理器，并且有些人认为应该据此修改DOM规范。这一问题在短期内可能不会有确定的结果。无论如何，我们建议在使用捕获处理器时应注意避免这一不确定行为的影响，此问题可参考http://hax.javaeye.com/blog/162718。——译者注

④ 原文“...root element...”有误。冒泡阶段最后会抵达树的顶端，即文档节点。此外，对于浏览器来说，事件在送达文档节点后还会被发送到Window对象。正在制定中的HTML5规范会对此作出明确规定。——译者注

```
<script type="text/javascript">
  window.onload = function() {
    document.getElementById('anImage').addEventListener(
      'click', react, false);
    document.getElementById('level1').addEventListener(
      'click', react, true);
    document.getElementById('level2').addEventListener(
      'click', react, false);
  }

  function react(event) {
    document.getElementById('info').innerHTML +=
      'I woof at ' + event.currentTarget.id + '!<br/>';
  }
</script>
 </head>
 <body>
   <div id="level1">
     <div id="level2">
       <img src="image.jpg" id="anImage"/>
     </div>
   </div>
   <div id="info"></div>
 </body>
</html>
```

❶ 挂接事件处理器

❷ 定义事件处理函数

❸ 定义嵌套元素

如前所述，我们对此示例的body❸进行了修改，把<img>元素嵌入到两层 元素中。

在onload事件处理器❶中，我们安置了三个事件处理器：<img>元素上一个，嵌套的 元素上也各有一个。注意，id为level1的元素上所安置的事件处理器，因为其注册时的第三个参数被设为true，所以是一个捕获处理器。

所有事件处理器都指定到同一个函数，react()❷，该函数返回一个消息，内容包含传给函数的event实例上的currentTarget属性的值。currentTarget属性与target属性不同，target表示触发事件的元素，而currentTarget表示事件传播过程中当前正在处理事件的元素，换言之，事件处理器正是安置在该元素之上。

在看图5-3之前，请先试着猜一下事件处理器的调用顺序。怎么样，你是不是已经答出来了？

在遵从标准的浏览器（记住，我们目前所用的代码并不适合Internet Explorer）中打开该示例，点击图片，我们就可以看到如图5-3所示的结果。

输出的次序完全符合我们的预期。level1元素上是捕获处理器，其余则是冒泡处理器。在捕获阶段，level1的处理器被触发，输出消息；目标阶段，<img>元素上的事件处理器被触发；最后是冒泡阶段，level2上的事件处理器被调用。

Internet Explorer仅支持目标和冒泡阶段，而不支持捕获阶段。为IE而修改本示例时，我们需要把对addEventListener()的方法调用改为attachEvent()，也需要像之前一样对事件处理器函数做一定的修改。糟糕的是，Internet Explorer中的Event对象没有任何一个属性是与currentTarget对应的。

如果你要针对IE编写代码，并且确实需要在冒泡阶段获取当前目标元素的引用，那么你就需

要暗渡陈仓去取得事件处理器所对应的元素的引用。我们可以采用的手段之一就是使用Prototype库的bind()机制，将事件处理器的函数上下文对象（即this引用）强制绑定为要安置该处理器的那个元素，代码如下：

```
Event.observe('someId','click',someHandler.bind($('someId')));
```

然后，在该事件处理器中添加一行代码：

```
if (!event.currentTarget) event.currentTarget = this;
```

该代码会检查出currentTarget未定义的情况，并给Event对象实例加上上下文对象的引用，这样处理器的后续代码就能够以跨浏览器一致的方式读取currentTarget。尽管有些繁琐，但如果你确实需要能够跨浏览器读取这一信息，费些周折还是值得的。

图5-3　本例捕捉型和冒泡型事件处理器的运行结果

1. 停止事件传播

有些时候，你可能想阻止某个事件继续传播。比如，当你确定需要对某个事件进行的处理已经全部完成，并且该事件如果继续传播，就会触发一些你并不想执行的事件处理器，这个时候你就需要中止事件传播。

在遵从标准的浏览器中，可在事件处理器内部调用Event类的stopPropagation()方法来阻止当前事件继续传播。在IE中，则需将Event实例的cancelBubble属性设为true。通过设置属性而不是调用方法来执行停止事件传播的操作，这看起来实在古怪，不过IE就是这样定义的。

2. 阻止默认操作

有一些事件，即所谓语义事件（semantic event），在浏览器中会触发某种默认的操作——例如提交表单时，或者点击链接点（anchor）元素时，都会触发某种默认操作[①]。

在DOM Level 0模型中，事件处理器可返回false来取消默认操作。在DOM Level 2中，Event类的preventDefault()方法具有相同的作用，所以可以调用该方法来阻止默认操作的发生。例如，如果submit事件处理器进行有效性验证时发现有一个或多个表单字段无效，就可以调用

① 前者是组织和编码表单数据并提交到指定的URL，后者则是打开超链接所指向的资源。——译者注

preventDefault()方法来阻止表单提交。在IE中，将Event实例的returnValue属性设为false可以阻止浏览器执行默认操作。

要处理所有这些浏览器差异，实在是令人头痛心烦外加恶心。很幸运，并非只有我们这么认为，所以那些编写JavaScript库的家伙来帮我们搞定了。让我们看看目前非常流行的Prototype库是怎样简化事件处理并使我们脱离苦海的。

5.3 使用 Prototype 进行事件处理

一些JavaScript程序库通过抽象出一般的事件处理接口，抹平了浏览器之间的差异，这样，定义一个事件处理器就变得简单了，Prototype库就是如此。在第3章和第4章中我们考察了它如何帮助我们编写面向对象的JavaScript和发起Ajax请求，现在让我们看看它对事件处理所做的简单而有用的抽象。

Prototype定义了Event命名空间，包含了众多有用的方法，最重要的两个方法是observe()和stopObserving()。前者用于将事件处理器附加到元素上，而后者则用于从元素上移除事件处理器。

我们选取代码清单5-6中的示例，将其改为使用Prototype库，结果如代码清单5-7所示。

代码清单5-7 Prototype方式的事件处理器

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('anImage', 'click', react, false);
        Event.observe('level1', 'click', react, true);
        Event.observe('level2', 'click', react, false);
      }

      function react(event) {
        $('info').innerHTML +=
          'I woof at ' + Event.element(event).id + '!<br/>';
      }
    </script>
  </head>
  <body>
    <div id="level1">
      <div id="level2">
        <img src="image.jpg" id="anImage"/>
      </div>
    </div>
    <div id="info"></div>
  </body>
</html>
```

❶ 定义事件处理器

❷ 声明事件处理函数

有了Prototype，果然不一样！不仅能用上Prototype所提供的乘手的$()函数，而且还能让我们的示例跨浏览器兼容，同时所需的代码量也减少了。

在onload事件处理器❶中，我们使用Event.observe()方法安置事件处理器，这一方法是跨浏览器的。我们仍可以指定，对于兼容W3C标准的浏览器，该事件处理器应作为捕获处理器还是冒泡处理器。而在IE浏览器下，这种区别将会被忽略掉。

在事件处理器❷中，我们使用Event.element()方法获得目标元素的引用，而不必考虑是哪一种浏览器。

注意，并不是浏览器在事件处理上的所有差异Prototype库都百分之百地抽象出了统一接口。例如，如果要获得currentTarget属性的值，我们需要直接读取，但我们必须保证在IE中运行时不去用它①。不过，Prototype对那些最普遍的事件处理需求进行了抽象，并给出了一套 API。

Prototype 的事件 API

本节快速过一下Prototype的Event命名空间下的API，并简要描述一下每个方法。

首先，方法

```
Event.observe(element,eventType,handler,useCapture)
```

为所传入的element元素安置一个指定类型的事件处理器。参数useCapture可省略并默认为false，在IE浏览器中该参数会被忽略。

接下来，方法

```
Event.stopObserving(element,eventType,handler,useCapture)
```

用于移除一个事件处理器。移除处理器时的参数应与安置该处理器时所用参数完全一致。

方法

```
Event.unloadCache()
```②

移除所有通过observe()方法安置的事件处理器，并释放所有引用，以便被垃圾回收器处理。这对于IE浏览器非常重要，因为IE的事件处理存在一个极严重的内存泄露问题③。

好消息是，对于IE，Prototype会在页面卸载（unload）时自动调用该方法。

下一个方法

```
Event.element(event)
```

返回所传入的事件的目标元素。

方法

```
Event.findElement(event,tagName)
```④

---

① Prototype 1.6.0重写了Event部分，现在你可以在事件处理器中使用this引用来得到处理器的所属元素，虽然它还是没有加上currentTarget属性，但至少在事件处理器函数内部，this引用与currentTarget是等价的。——译者注

② 该方法不应由使用者调用。Prototype 1.6.0已取消此方法，对应功能改由内部方法destroyCache完成。——译者注

③ 准确地说，此问题并不限于事件处理，只要存在JavaScript原生对象与DOM对象的循环引用，就会导致JavaScript引擎的垃圾回收器与微软COM对象引用计数器互相等待对方断开引用而无法释放对象，于是形成了内存泄漏。由于在第3章中所提到的JavaScript的闭包特性会让函数隐含持有外层变量的引用（如果这些变量中有一个引用了某个DOM节点，则函数也就间接引用了该DOM节点），而事件处理器函数必然将被挂接到DOM节点上（即DOM节点内部必然会持有该JavaScript函数的引用），所以实践中几乎无法避免循环引用。——译者注

④ Prototype 1.6.0将此方法扩展为Event. findElement(event, expression)，即不限于按标签名查找而可以按照一个CSS表达式进行匹配。——译者注

从所传入的事件的目标元素开始向上遍历其祖先元素，返回其中第一个具有指定标签名的元素[①]。例如，将tagName参数设为字符串“div”，就可以查找离目标元素最近[②]的 上级元素。

方法

```
Event.pointerX(event)
```

返回一个鼠标事件的相对页面的水平位置，方法

```
Event.pointerY(event)
```

返回一个鼠标事件的相对页面的垂直位置。

方法

如果鼠标事件中用户点击的是鼠标主键（primary mouse button）[③]，

```
Event.isLeftClick(event)
```

会返回true。

最后，方法

```
Event.stop(event)
```

停止事件传播并且取消与此事件关联的默认操作。

好了，这些方法应该会使编写事件处理代码变得简单许多了。现在让我们把目光转向那些我们时常会遇到的事件类型。

5.4 事件类型

对于Web应用来说，我们关心的大多数事件都是用户使用鼠标或键盘与应用进行交互而产生的。这些事件在DOM树中激发，以响应用户操作，如加载页面、点击按钮、移动鼠标、拖动鼠标、键盘输入或某种会卸载页面的操作。正如我们所见，可以为这些事件编写事件处理器，这样我们的应用就可以对用户的操作作出反应。本节，我们将深入了解常用的事件类型——就从鼠标事件开始吧。

5.4.1 鼠标事件

Web应用中最常遇到的鼠标事件包括mouseup、mousedown、click、dbclick和mousemove。当用户点击某个元素时，会激发三个鼠标事件：mousedown、mouseup和click。让我们通过代码清单5-8中的代码亲自观察一下这个过程。

代码清单5-8 一次点击产生的鼠标事件

```
<html>
  <head>
    <title>Mouse events!</title>
```

① 原文是“returns the nearest ancestor of the target element for the passed event that has the passed tag name”，不太准确，实际上，如果目标元素符合条件，应该会直接返回目标元素。——译者注

② 同上，如果目标元素是一个div元素，则直接返回目标元素。——译者注

③ 其实isLeftClick这个名字不太好，虽然多数情况下鼠标主键通常就是左键，但是操作系统可能会允许用户将其他键设为主键，比如左撇子可能会把鼠标右键设为主键。而且有些系统的鼠标只有一个键，自然没有左右之分。——译者注

```
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">                    ❶ 挂接鼠标事件处理器
      window.onload = function() {
        Event.observe('anImage', 'click', react);
        Event.observe('anImage', 'mousedown', react);
        Event.observe('anImage', 'mouseup', react);
      }

                                          ❷ 设置事件信息
      function react(event) {
        $('info').innerHTML +=
          'I bark for ' + event.type +
          ' at (' + Event.pointerX(event) + ','+
          Event.pointerY(event) + ')!<br/>';
      }
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
    <div id="info"></div>
  </body>
</html>
```

在这段代码中，我们在<img>元素上为click、mouseup和mousedown安置了事件处理器❶。当点击图片时，事件处理器函数❷检查event实例并输出事件类型以及点击时鼠标指针相对于页面的坐标。我们会在浏览器中看到如图5-4所示的结果。

从输出的结果中我们可以看出，当元素被点击时，mousedown事件首先激发，紧接着是mouseup，最后是click。作为练习，你可以加上mousemove和dbclick事件处理器，并看看这些事件的发送过程和互相之间的关联。

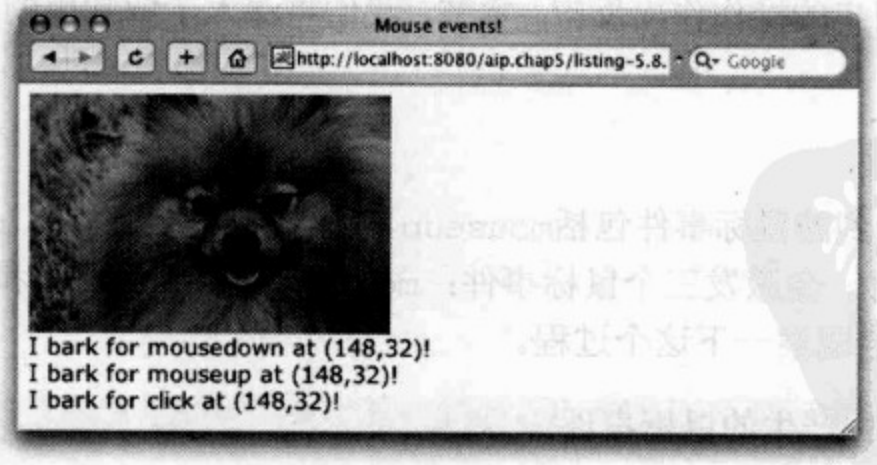


图5-4　鼠标事件的响应模式

5.4.2　键盘事件

常用的键盘事件包括keyup、keydown、blur和focus[①]。keyup和keydown事件与mouseup

① 当然blur和focus不能算是键盘事件，只是因为常由tab键切换而导致焦点变更，所以作者姑且将它们列于此处。——译者注

和mousedown事件类似，在键盘上按下某个键时激发keydown事件，释放按键时则激发keyup事件。

当一个DOM元素获得或失去焦点时会触发focus和blur事件。在任何页面中，同一时刻只有一个DOM元素可以拥有焦点。焦点可以以编程方式来改变，也可能因用户操作而改变。当用户用tab键跳离一个字段时，就会激发blur事件，随后则是获得焦点的元素的focus事件。用户也可以通过点击一个可拥有焦点的元素[①]改变当前焦点。

我们来看一个blur和focus事件的示例，请看代码清单5-9中的代码。

代码清单5-9 blur然后focus然后blur然后focus然后……

```
<html>
  <head>
    <title>Blur and Focus</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('nameField', 'blur', react);
        Event.observe('nameField', 'focus', react);
        Event.observe('breedField', 'blur', react);
        Event.observe('breedField', 'focus', react);
        Event.observe('dobField', 'blur', react);
        Event.observe('dobField', 'focus', react);
        $('nameField').focus();
      }
      function react(event) {
        $('info').innerHTML +=
          Event.element(event).id + ' ' +
          event.type + '<br/>';
      }
    </script>
  </head>
  <body>
    <form name="infoForm">
      <div>
        <label>Dog's name:</label>
        <input type="text" id="nameField"/>
      </div>
      <div>
        <label>Breed:</label>
        <input type="text" id="breedField"/>
      </div>
      <div>
        <label>Date of birth:</label>
        <input type="text" id="dobField"/>
      </div>
      <div>
        <input type="submit" id="submitButton"/>
```

❶ 在页面load时挂接事件处理器

❷ 让第一个字段拥有焦点

❸ 处理blur和focus事件

❹ 包含可设焦点元素的表单

① 通常只有表单控件、链接点（a）元素以及body本身可以拥有焦点，但是IE存在设计缺陷，导致几乎所有的元素都有可能获得焦点。——译者注

```
        </div>
      </form>
      <div id="info"></div>
    </body>
  </html>
```

示例的结构与之前大同小异，不过我们做了一些改动，以把焦点从鼠标事件（主要是click）转移到键盘事件。

修改后页面的body包含一个<form>元素❹，其中定义了三个文本框。在onload事件处理器中❶，我们为每一个文本框都安置了focus和blur事件处理器。为清晰起见，这些处理器是一个一个添加的。作为一个练习，你能否将代码改写成给表单中的所有文本框装备事件处理器而不必逐一列出？

在onload处理器的结尾，我们通过脚本将焦点设到了表单的第一个字段上❷。这很有意义（除了对用户更友好之外），因为在页面加载时，它向我们显示出，第一个字段的focus处理器会被触发。这说明，焦点无论是脚本设置的，还是通过用户的活动设置的，都会触发focus事件。

但并非所有事件都是如此。例如，通过脚本控制来提交表单，就不会触发表单元素上的submit事件。

我们还对react()❸事件处理器函数稍微做了些改动，让它在事件类型之前也显示出目标元素的名字。

当页面最初加载到浏览器时，显示结果如图5-5上半部分所示。可以看到，focus事件处理器已经被调用了一次，因为我们在onload处理器中将焦点设到了nameField元素之上。

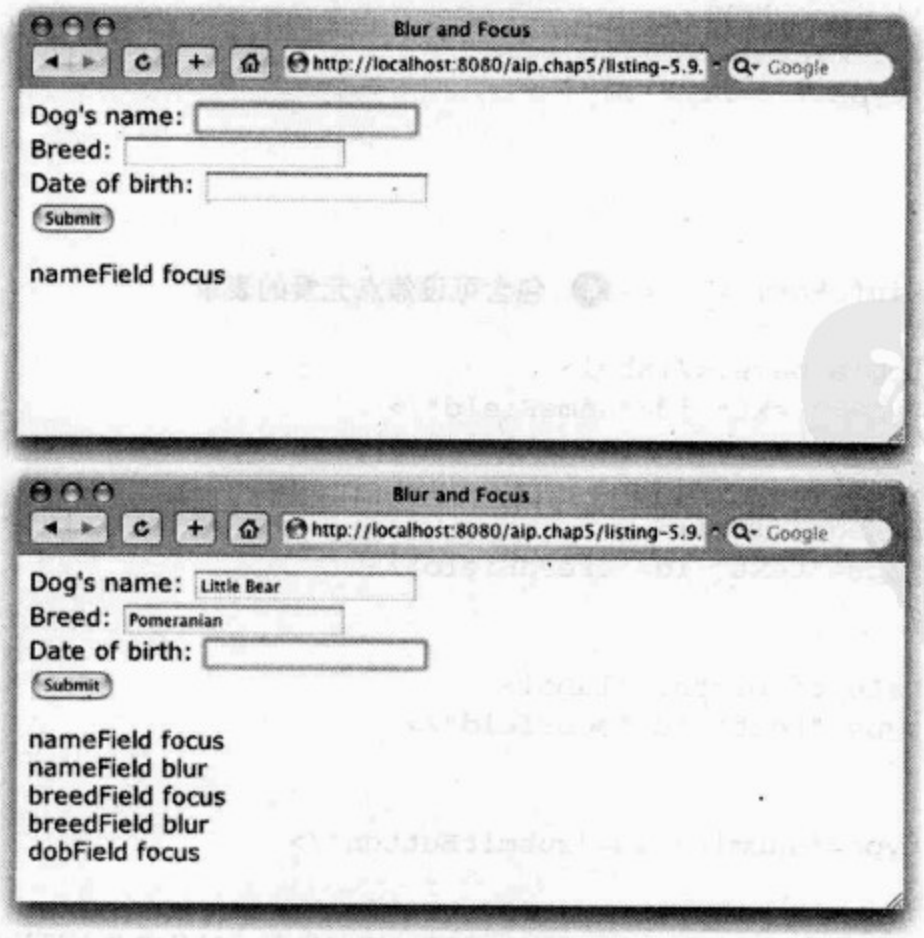


图5-5　获得焦点和丢失焦点

在填写数据和按tab键跳转到dobField元素的过程中，我们可以看到每次按下tab键跳离某个字段时，我们所离开的元素上的blur事件处理器就会被调用，然后按照tab次序下一个元素获得焦点（在第10章我们将详细了解tab次序）并触发focus事件处理器。

请将代码清单5-9中的示例代码复制一份，并给这些字段添加一些其他键盘事件的处理器。观察一下，当你在字段键入内容时，这些事件处理器是如何触发的。

5.4.3 change 事件

我们已经看了如何使用blur事件处理器来得到用户离开某个元素的通知。我们可能还需要了解，一个DOM元素在失去焦点时元素的值是否也发生了改变——比如，我们希望只在字段值改变时才执行有效性验证，而不是每次失去焦点时都检查一遍。有一些特定类型的元素，例如text、textarea、select和file，如果失去焦点并且在拥有焦点期间内容发生了改变，就会在DOM中激发一个change事件。

为了实际演示一下，我们将修改之前的示例，给文本框加上change事件处理器。改后的代码列于代码清单5-10中，粗体显示的是针对代码清单5-9的增改。

代码清单5-10 获取改变的内容

```
<html>
  <head>
  <title>Ch-ch-changes</title>
  <script type="text/javascript" src="prototype-1.5.1.js">
  </script>
  <script type="text/javascript">
    window.onload = function() {
      Event.observe('nameField', 'blur', react);
      Event.observe('nameField', 'focus', react);
      Event.observe('nameField', 'change', react);
      Event.observe('breedField', 'blur', react);
      Event.observe('breedField', 'focus', react);
      Event.observe('breedField', 'change', react);
      Event.observe('dobField', 'blur', react);
      Event.observe('dobField', 'focus', react);
      Event.observe('dobField', 'change', react);
      $('nameField').focus();
    }
    function react(event) {
      $('info').innerHTML +=
        Event.element(event).id + ' ' +
        event.type + '<br/>';
    }
  </script>
</head>
<body>
  <form name="infoForm">
    <div>
      <label>Dog's name:</label>
      <input type="text" id="nameField"/>
    </div>
```

```
        <div>
          <label>Breed:</label>
          <input type="text" id="breedField"/>
        </div>
        <div>
          <label>Date of birth:</label>
          <input type="text" id="dobField"/>
        </div>
        <div>
          <input type="submit" id="submitButton"/>
        </div>
      </form>
      <div id="info"></div>
    </body>
  </html>
```

对HTML文档做了小小改动之后，我们在表单文本框字段发生变更时就可以得到通知。

如果我们在浏览器中加载该页面，在第一个字段输入一些文本，然后按tab跳至第二个字段，不在第二个字段输入任何内容就继续跳至第三个字段，我们将看到如图5-6的结果。如你所见，在狗名字段[①]的blur事件之前触发了一个change事件，用户的输入导致它的内容发生了改变，但品种字段[②]就没有触发change事件，因为它的内容没有改变。

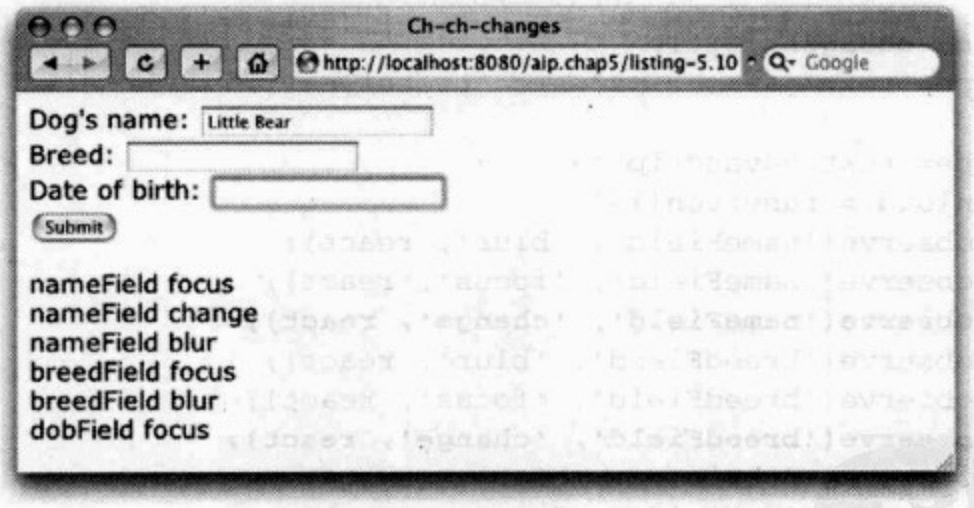


图5-6　发生了什么改变?

5.4.4　页面事件

我们已经介绍了用户与一个已载入的（loaded）页面中的元素进行交互所激发的各种事件，但浏览器也能激发一些代表页面级别活动的事件。它们被称为**页面事件**（page events），当文档加载、卸载、改变窗口大小或进行页面滚动（scroll）时，就会产生页面事件。尽管这些事件看似特殊，但像其他事件一样也可以被捕获。我们可以在页面的<body>元素上设置事件处理器，或通过窗口对象指派事件处理器。

在本章介绍过的每个示例中，我们都看到了load事件的实际运用，我们使用load事件来声

① 即“Dog's name”所对应的字段，也即id为nameField的input元素。——译者注

② 即“Breed”所对应的字段，也即id为breedField的input元素。——译者注

明所要演示的其他事件处理器。现在让我们加入unload和onberforunload事件的例子，如代码清单5-11所示。

代码清单5-11 处理页面事件

```
<html>
  <head>
    <title>Page Events</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {          ❶ 页面加载完毕后弹出警示窗口
        alert('Loaded!');
        window.onunload = function() {      ❷ 页面卸载完毕后弹出警示窗口
          alert('Unloaded!');
        }
        window.onbeforeunload =             ❸ 提供退出选择
          function() {
            return 'Leaving so soon?';
          }
        }
      }
    </script>
  </head>
  <body>
    <a href="listing-5.11.html">Do it again!</a>
  </body>
</html>
```

由于我们要进行页面本身的加载和卸载，采用页面上的输出来观察运行状况就不太靠谱了，因此我们再次回到了提示对话框。在onload事件处理器中，我们在页面加载后❶发出一个提示，然后继续安置unload和beforeunload事件处理器。

在onunload事件处理器中❷，我们简单的发出另一个提示来报告事件已被触发。不过onbereunload事件处理器就比较有趣了。

在onunload事件处理器中，我们除了对页面卸载的既成事实作出被动反应外，就没什么事情可干了。但在onbeforeunload事件处理器中，我们可以对页面是否卸载产生实际影响。如果我们的onbeforeunload事件处理器❸返回一个值，浏览器就会显示一个对话框，询问用户是否卸载页面。该对话框的文本会包含从事件处理器返回的值。

当我们在浏览器中载入这个示例，会得到一个已经有点烦人的提示，指出页面已加载完毕。点击页面上的链接会直接重新显示相同的页面。点击后，我们看到浏览器触发了onbeforeunloaded事件处理器，处理器的返回结果显示于对话框中，如图5-7所示。

这一技巧显然很有用，当用户尚未完成操作就试图离开页面时，我们可以进行询问，这样就能确保用户不会无意中丢失数据。如果用户点击取消按钮，页面跳转就会被取消，卸载操作也就不会发生；如果用户点击确定按钮，卸载操作就会执行，用户在页面重新加载之前会收到提示，报告unload事件处理器已被调用。

关于load事件的一个题外话：我们也会看到一些页面把<script>元素放到页面末尾附近，

以便在页面加载时执行脚本代码。该手段和load事件的区别在于，load事件直到页面完全加载完毕之后才会触发，这包括所有的外部元素如脚本文件、级联样式表以及图片[①]。

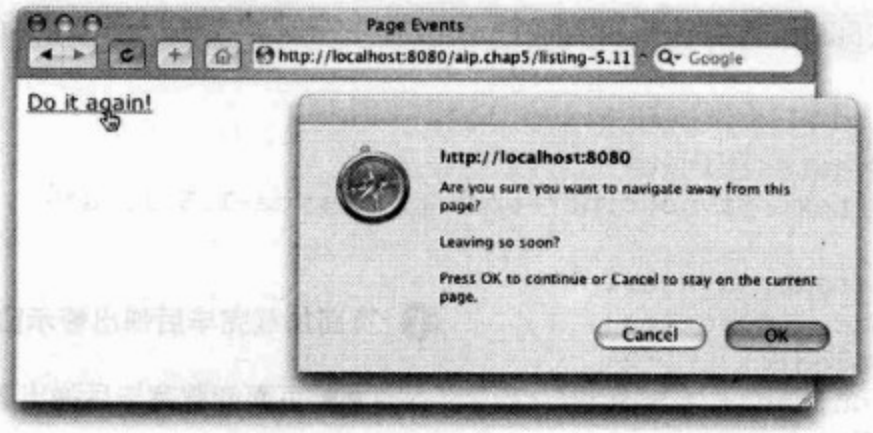


图5-7　退出之前让我们聊聊

以上我们完成了对事件处理以及常见事件类型的介绍。显然，我们无法讲到所有可能在网页中触发的事件类型——这样一个完整的介绍可能要占用好多章节——不过目前这些信息已经足够帮助你理解如何进行事件处理，以及如何应付当前Web应用中最常用的事件类型了。

现在我们已经有了不少关于事件处理和事件类型的知识，下面让我们来看几个示例，看看怎样将理论用于实践。

5.5　事件处理实践

本节示例需要服务器端资源提供服务支持才能运行。为了方便读者，本章位于www.manning.com/crane2的示例代码已经打包成一个完整可运行的Web应用。

如果你的系统中已经运行有servlet容器，只需创建一个叫做aip.chap5的应用环境（application context），并将其文档基准(document base)指向所下载的代码包中的chap5目录。

如果你没有运行servlet引擎，也不用担心。在下载包的chap4目录中有一个PDF文档介绍了如何下载和配置Tomcat，包括如何建立应用上下文。

在浏览器中打开本章的示例时，请确保是通过Web服务器访问这些网页，而不是以本地文件形式打开这些HTML页面。例如，要加载代码清单5-12中的示例，你应使用如下地址：

```
http://localhost:8080/aip.chap5/listing-5.12.html
```

当然，这假定你的servlet容器运行在默认的8080端口上。如果你修改了端口，就需要对上述URL进行相应的调整。

5.5.1　在服务端验证文本字段

在我们真正掌握了如何在一个DOM元素上附加change和blur事件处理器的相关知识后，利

① 所以，对于许多需求来说load事件发生得太迟了。对此，Firefox等浏览器支持一个称作DOMContentLoaded的扩展事件，在整个HTML DOM加载和解析完成之后就会触发该事件而不必等待其他资源载入，这非常类似于在页面底端加入<script>。Prototype 1.6.0对此事件提供了跨浏览器支持，用法请参考http://www.prototypejs.org/api/document/observe。——译者注

用这些处理函数在客户端校验用户输入的数据是否合法相当容易实现。简单的客户端校验很容易实现，但有时业务需求规定数据可能需要使用服务端特有的信息才能完成校验。或者校验太复杂无法使用JavaScript实现，也许校验所需的信息量庞大难以传输到页面以便供客户端使用。

传统Web应用的常用策略就是在页面做简单的校验，表单提交之后在服务端执行更为复杂的校验。但随着Ajax的出现，我们不必再让用户忍受这种令人发狂的校验方式。要想让服务端协助完成校验，我们需要在客户端发生特定事件时向服务器发生一个请求，服务端校验数据并返回一个相应消息。

我们拥有所要解决的问题的全部信息。我们知道可以在一个文本框上附加事件并检测任何改变，我们还明白可以使用该事件触发一个到服务器的Ajax请求。服务器处理请求的资源可以校验该数据并在数据无效时返回一个错误消息。

注意本节示例的用途在于演示实际应用如何使用事件处理，而不是展示一个成熟或复杂的校验框架。这个主题将在本书稍后的第6章和第10章继续讨论。

1. 问题

当文本字段的值改变时，使用服务端的资源对它们进行校验。

2. 解决方案

我们已经明白如何对一个文本输入框指定其事件处理器，上述问题的解决方案也是如此，没有区别。问题是，我们是跟踪blur事件还是change事件？

答案取决于数据和执行校验的种类。由于我们打算采用服务端协助的方式完成校验操作，无论什么时候进行校验都需要在客户端和服务端往返，因此我们应该确保不去生成一些没有用的校验请求。

如果我们从开始就知道某个数据合法，可以限制只跟踪change事件。毕竟，我们没有必要去校验已经知道的合法数据。但更为普遍的情况是，字段的开始部分可能是未知数据（甚至是空），或许我们需要跟踪blur事件以便每次访问该字段时它已经通过校验。

为某字段创建一个支持校验的事件处理器很简单，如下所示：

```
Event.observe('fieldId','blur',validationFunction);
```

代码清单5-12显示了一个含有小型表单的网页。该表单由美国地址、城市、州和邮政编码组成。我们的业务需求规定邮政编码和地址必须匹配。这需要访问一个可接受外部校验请求的服务端的API即USPS（United States Postal Service，美国邮政服务），这应当在服务端的代码上通过实现。请看我们如何在页面对此进行处理。

代码清单5-12 校验邮政编码

```
<html>
  <head>
    <title>I Need Validation</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">                    ❶ 创建事件处理器
      window.onload = function() {
        Event.observe('zipCodeField','blur',validateZipCode);
        $('addressField').focus();
```

```
    }

    function validateZipCode(event) {    ❷ 发起验证请求
      new Ajax.Request(
        '/aip.chap5/validateZipCode',
        {
          method: 'get',
          parameters: $('infoForm').serialize(true),
          onSuccess: function (transport) {
            if (transport.responseText.length != 0)
              alert(transport.responseText);
          }
        }
      );
    }
  </script>
</head>
<body>
  <form id="infoForm">    ❸ 创建数据输入表单
    <div>
      <label>Address:</label>
      <input type="text" id="addressField" name="address"/>
    </div>
    <div>
      <label>City:</label>
      <input type="text" id="cityField" name="city"/>
      <label>State:</label>
      <input type="text" id="stateField" name="state"/>
      <label>Zip Code:</label>
      <input type="text" id="zipCodeField" name="zipCode"/>
    </div>
    <div>
      <input type="submit" id="submitButton"/>
    </div>
  </form>
  <div id="info"></div>
</body>
</html>
```

本页面主要处理三个活动：建立事件处理器❶，通过发起到服务端资源的校验请求❷响应blur事件，建立用户可填充的表单❸项目。

在页面的onload事件处理器❶中，我们为blur事件创建了validatateZipCode方法，他将在用户离开的邮政编码字段的任意时刻被调用。该函数❷触发一个由Prototype协助完成的Ajax请求访问服务端名为validateZipCode的服务。此刻你会看到，该资源就是一个Java Servlet，它实现一些简单响应以模拟实际的邮政编码校验过程。

Prototype为<form>元素添加了便于使用的serialize()方法，利用此方法我们给服务端资源传递表单的所有字段和值。

服务端校验资源这样处理校验：如果一切正常，返回空响应；校验失败，则返回错误消息。所以在创建Ajax请求的onSuccess事件处理器中，我们检测响应的文本，在某个字段没有通过校

验就弹出一个警告框。记住，我们在稍后章节将介绍更为复杂的校验。

在浏览加载该页面（确认使用Web服务器URL访问，而不是从文件菜单打开某文件）并填充它们。注意，当你离开邮政编码字段时，会弹出一个警告框显示校验失败消息，如图5-8所示。

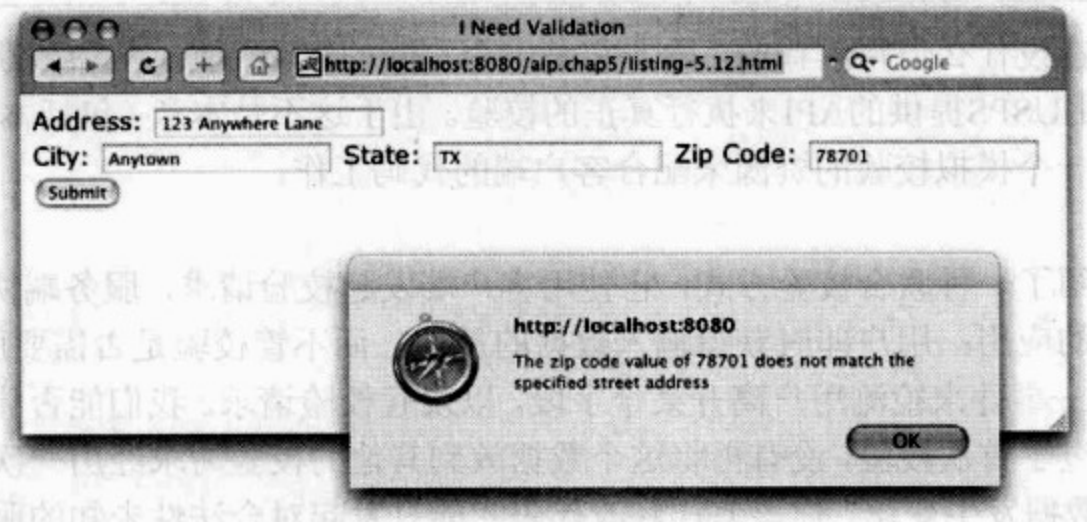


图5-8 邮政编码校验错误

事实上，你会发现你输入的每个邮政编码都会校验失败并弹出一个提示框，除非你猜对了唯一正确的邮政编码值“01826”。当然，这是因为服务端用于校验的servlet不是真地连接USPS数据库去执行实际校验。图5-13展示了模拟校验操作的servlet代码。

代码清单5-13 在邮政编码的校验中模拟真实校验

```
package org.aip.chap5;

import java.io.IOException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Smoke-and-mirrors validator servlet for listing 5.12. The
 * zip code must be non-blank and equal to "01826" to be
 * considered valid.
 */
public class ZipCodeValidatorServlet extends HttpServlet {

  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
      throws IOException {
    StringBuilder result = new StringBuilder ();
    String zipCodeValue = request.getParameter("zipCode");
    if (zipCodeValue.length() == 0) {
      result.append("The zip code field cannot be blank");
    }
    else if (!zipCodeValue.equals("01826")) {
      result
        .append("The zip code value of ")
        .append(zipCodeValue)
```

```
            .append(" does not match the specified street address");
        }
        response.getWriter().write(result.toString());
    }
}
```

上述代码实在没什么值得解释的。除非它是一个实际的校验资源，从参数获取表单的所有字段和值，然后利用USPS提供的API来执行真正的校验。由于这不是本章（包括本书）关注的重点，因此我们只提供一个模拟校验的资源来配合客户端的代码工作。

3. 讨论

本节我们介绍了一种联合校验方式，它使用客户端发起校验请求，服务端协助完成校验。它使我们创建这样的应用：用户即时获得输入数据的反馈，而不管校验是否需要服务端资源。

我们使用blur事件来检测用户离开某个字段，以发起校验请求。我们能否让这个过程更加智能？一旦数据经过了首次校验，没有再将这个数据放到其他的校验请求经历一次客户服务端的往返过程，除非该数据发生变化。你该怎样修改代码才能只发起对合法性未知的服务端校验请求？

上述联合校验方式使用客户端和服务端协助完成校验，它是对我们创建Web应用程序的工具箱的一种强大补充。这种校验方式能防止表单提交之后被告之提交数据中有错误数据，这会使许多用户产生挫折感。因此，你应该尽一切办法实现类似的校验，但你不要依赖它。

任何人都可以随意看到客户端代码并访问网页，一些居心叵测的人很容易对代码进行逆向工程然后提交他们的恶意数据，完全绕开任何客户端的校验框架，不管框架设计多么巧妙。数据之所以合法，总是在表单提交的基础上执行服务端校验，而不管之前已经进行多少次校验。你可以在表单最后提交时刻调用这些用于校验的代码（例如本例介绍过的代码），如客户端发起校验请求、服务端协助校验等。

说到表单提交，现在或许是时候使用新的表单提交方式了，无需重新加载整个页面的负担。接下来我们介绍它。

5.5.2　无需页面重新加载的表单元素提交方式

目前绝大多数接受用户输入的Web页面，在用户填写完页面数据项后，仍然使用传统技术将表单提交到服务端。在今天我们使用Ajax创建富因特网应用的环境下，这种强制进行整页面刷新的方式显然不合时宜。

1. 问题

我们想在没有整页面重新加载的情况下提交一个表单到服务器。

2. 解决方案

尽管已经有了方案，但它还几乎完全不值一提。事实上，我们几乎在上个例子中完成了这个工作。要提交该表单，我们将利用上个例子用到的相同技术，在上个例子中我们把表单的元素发送到服务端进行校验。

这个解决方案或许既毫无意义又很熟悉，但有一些细微差别使得它值得考虑。我们将采用前一个示例的代码，移除校验检查部分（因此我们得以专注于表单提交这个主题），并且改写部分代码以拦截表单提交过程，以便在Ajax控制下提交表单，而不是采用正常的表单提交方式，如代

码清单5-14所示。

代码清单5-14 拦截表单提交过程

```
<html>
  <head>
    <title>Submit!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('infoForm','submit',submitMe);
        $('addressField').focus();
      }

      function submitMe(event) {
        new Ajax.Request(
          '/aip.chap5/handleSubmission',
          {
            method: 'post',
            parameters: $('infoForm').serialize(true),
            onSuccess: function (transport) {
              $('info').innerHTML = transport.responseText;
            }
          }
        );
        Event.stop(event);
      }
    </script>
  </head>

  <body>
    <form id="infoForm"
          action="/aip.chap5/shouldNotActivate">
      <div>
        <label>Address:</label>
        <input type="text" id="addressField" name="address"/>
      </div>
      <div>
        <label>City:</label>
        <input type="text" id="cityField" name="city"/>
        <label>State:</label>
        <input type="text" id="stateField" name="state"/>
        <label>Zip Code:</label>
        <input type="text" id="zipCodeField" name="zipCode"/>
      </div>
      <div>
        <input type="submit" id="submitButton"/>
      </div>
    </form>
    <div id="info"></div>
  </body>
</html>
```

❶ 挂接submit事件处理器

❷ 使用Ajax控制表单提交

❸ 指定常规的提交操作

该页面改动虽小，但意义重大。首先，在window对象的onload处理器❶中，我们给表单元素添加了submit事件处理器，它将在表单提交❷时调用submitMe()方法。

我们过会再解释该方法，先来看看我们对<form>元素❸所做的改动。我们对它的action属性指定了一个不存在的资源。这样做，我们马上能明白表单永远不会通过使用这个默认的属性提交。当服务器报告无法找到默认的提交资源时，浏览器将显示一个准确无误的错误页面。

当submit事件触发时，submitMe()方法被调用，它发起一个我们在前面示例中看到过的Ajax请求。但在这个例子中，我们指定HTTP方法为POST而不是GET。这个繁重工作已经由Prototype的serialize()方法完成。

服务端处理该请求的资源是一个servlet，它收集请求的参数并格式化一个包含HTML代码片段的响应。该片段由参数中的名称和值组成（由于这个操作和本主题没有特别密切的关系，这里不作深入讨论。如果你对此感兴趣，可以从可下载的代码org.aip.chap5.ParamaterInspectorServlet中看到源代码）。该响应主体显示在页面中的info元素上。

最后，以下代码执行：

```
Event.stop(event);
```

这个Prototype的方法停止事件的继续传播并取消该事件的默认操作，对本例来说就是表单提交。没有这行代码，表单将继续提交至表页面中表单元素action属性指定的资源。

3. 讨论

尽管这个例子没有涵盖很多新东西，但它指出了一些重要概念，例如使用submit事件屏蔽表单的默认提交。我们使用一个事件处理器和Prototype事件方法来达到该目的，如果你想屏蔽所有的表单提交操作行为，可以在表单的声明处使用DOM Level 0级别的事件处理器来返回false：

```
<form id="my Form" action="whatever" onsubmit="return false;">
```

在我们的示例中，我们对Prototype的serialize()方法相当依赖。该方法将表单上包含的元素名称及值进行散列化，并组装成一个查询字符串或把这些参数进行哈希算法处理的对象。

由于我们对该方法的参数指定了true，因此它将返回一个hash对象，这是Prototype 1.5的默认特性。

当该页面加载后，输入数据，然后点击Submit按钮（或按下回车键），显示结果如图5-9所示。

这显得相当容易，但如果我们吹毛求疵呢？

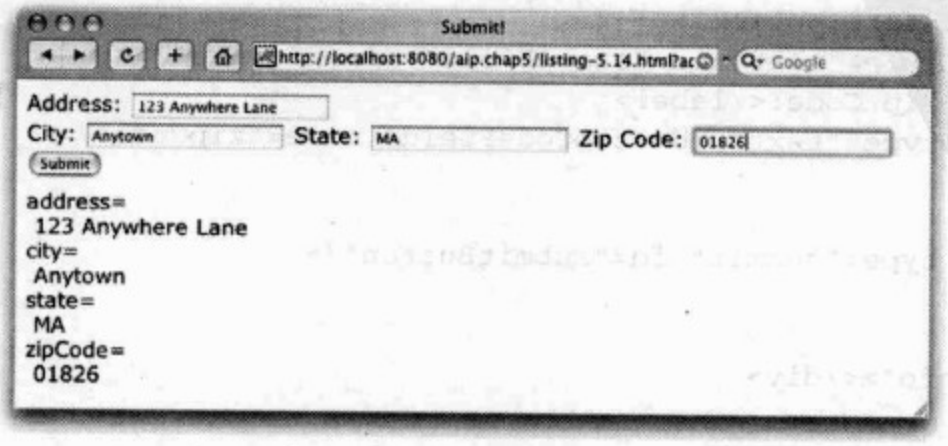


图5-9　在屏蔽默认提交方式的情况下提交表单

5.5.3 只提交发生改变的元素

前一示例向我们展示了如何控制表单提交过程和使用事件处理把表单数据重新组织到Ajax请求。Prototype的serialize()方法替我们完成了这些近乎琐碎的工作，它收集所有的表单数据并将其发送到服务器。

但如果我们不希望把所有的表单数据发送到服务端，又该如何？假设我们只想把发生改变的元素数据发送出去，又该如何？更有甚者，我们为何要把没有发生变化的数据放入请求？使用表单元素的change事件，我们可以了解元素何时发生了改变。不过，怎样才能最好地跟踪这些信息以便在需要向服务器发生数据的时候使用它们？

我们可以采用初级的解决方案，使用一个全局变量存储该信息。但这样做既不雅观，同时也会使多表单页面产生一些严重问题，并且这是一种非面向对象的解决途径。

我们可以采用精巧的解决方案，通过给元素添加一个自定义属性将该信息存储在元素自身上。如下所示：

```
element.hasChanged = true;
```

需要提交表单时，我们可以遍历所有元素，查找已设置该属性的元素以收集数据。

或者更好，我们还能让它更加智能（和偷懒相比，这个说法好听得多）和高效，我们已经实现了关键代码，代码清单5-15展示了该方法。

代码清单5-15 只发送发生改变的数据

```
<html>
  <head>
    <title>Submit, or not!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('infoForm','submit',submitMe);
        Event.observe('infoForm','change',
                      markChanged);
        $('addressField').focus();
      }

      function markChanged(event) {
        Event.element(event).addClassName('changedField');
      }

      function submitMe(event) {
        var changedElements = $$('.changedField');
        if (changedElements.length > 0 ) {
          var parameters = {};
          changedElements.each(
            function(element) {
              parameters[element.name] = element.value;
              element.removeClassName('changedField');
            }
          );
```

❶ 将change处理器安置到表单中

❷ 把目标元素标记为已改变

❸ 收集发生改变的元素

```
            new Ajax.Request(
              '/aip.chap5/handleSubmission',
              {
                method: 'post',
                parameters: parameters,
                onSuccess: function (transport) {
                  $('info').innerHTML = transport.responseText;
                }
              }
            );
          }
          Event.stop(event);
        }
      </script>
    </head>

    <body>
      <form id="infoForm" action="/aip.chap5/shouldNotActivate">
        <div>
          <label>Address:</label>
          <input type="text" id="addressField" name="address"/>
        </div>
        <div>
          <label>City:</label>
          <input type="text" id="cityField" name="city"/>
          <label>State:</label>
          <input type="text" id="stateField" name="state"/>
          <label>Zip Code:</label>
          <input type="text" id="zipCodeField" name="zipCode"/>
        </div>
        <div>
          <input type="submit" id="submitButton"/>
        </div>
      </form>
      <div id="info"></div>
    </body>
  </html>
```

在本例中，我们对代码清单5-14中的代码做了一些微小但意义非凡的修改。在onload事件处理器中，我们为表单❶创建了一个change事件处理器。我们可以遍历所有元素并为每个元素添加事件处理器，不过表单即将在冒泡阶段接收事件通知，有什么值得担忧?

任意时刻的表单元素发生改变都会调用markChanged()处理方法❷，该方法能获得改变事件的目标元素并为它添加一个名为changeField的级联样式表Class。

啊？层叠样式表和跟踪记录表单元素改变有何关系？等到我们在submitMe()事件处理器中检查改变情况时一切就真相大白了。

在该方法❸中，我们用到了轻巧的Prototype库的$$()方法。把级联样式表选择器作为参数传入该方法，该函数返回所有和选择器匹配的元素数组。由于我们指定字符串“.changeField”作为参数传入该方法，所有带有该级联样式表Class名称的元素将以一个数组形式返回。

如果返回的数组为空，我们就简单地跳过这段请求提交代码。否则，我们遍历所有元素，并

创建一个使用名/值对表示的Hash对象以便从数组元素中收集所有改变的元素。该Hash对象稍后将作为参数设置到Ajax请求对象中。

由于数据已经提交完毕，并且这些元素不再被认为是发生改变的元素，我们从这些元素中删除名称为changeField的级联样式表Class，再次执行上述过程仍然有效。

讨论

本示例的实现如代码清单5-14所示。其中，发往服务器的Ajax请求包含的参数仅限于那些值发生改变的元素，并完全忽略那些没有任何改变的元素。

我们使用表单中的change事件处理器来捕捉所有表单元素的改变，该技巧灵活地利用了事件传播的冒泡阶段。而且我们介绍了一个辨别元素是否改变的技巧，这就是利用添加层层叠样式表Class名称和Prototype库的$$()函数。

在浏览器显示该示例时，仅有City和State字段发生了改变，见图5-10所示。

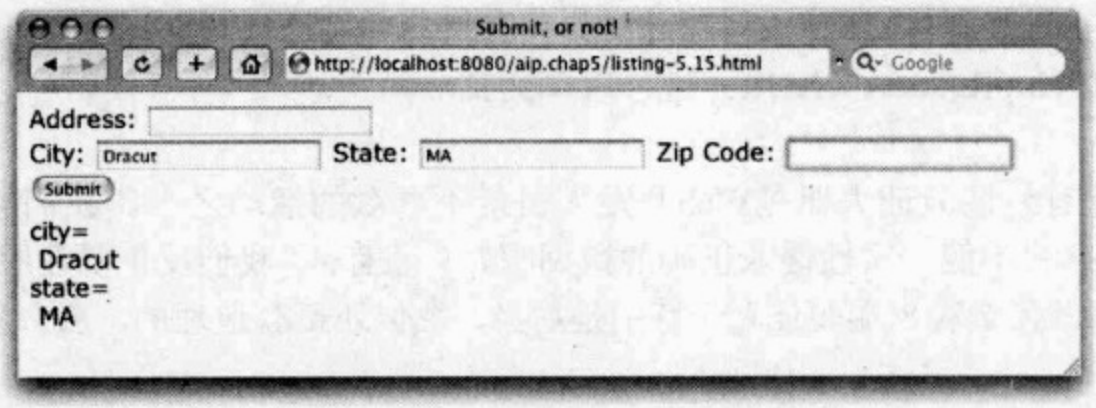


图5-10 只提交发生改变的元素

5.6 总结

在本章中，我们介绍了一些可以增强Web应用交互性的有趣而且有效的技巧。我们了解了给DOM元素添加事件处理器的各种方式，领略了Prototype程序库是如何极大地简化了添加和编写事件处理器的过程。我们介绍了所有的主要事件类型，并考察了许多代码片段，它们演示了怎样在Web应用中使用这些事件。我们还看了一些表单验证和表单提交的示例，下一章中将会对此进行更深入的介绍。

第6章 表单验证与提交

本章内容

- 客户端字段验证
- 客户端跨字段验证
- 用POST方法发送HTTP请求
- 使用XMLHttpRequest（XHR）进行表单提交

对输入进行有效性验证大概是Web开发人员最不喜欢的活动之一。我们经常听到这样的抱怨："为什么用户就不能一次性键入正确的数据呢？"还有："我们反正要在服务器上进行验证，干嘛要自寻烦恼做什么客户端验证？"特别是后者，貌似还挺有道理的。那么我们为什么要进行输入验证呢？

这是因为，它创造了更好的用户体验：

- 将表单上的错误数据立即反馈给用户。
- 减少服务器资源占用（网络流量和服务器运行周期），令用户界面运行更流畅。

输入验证要和HTML表单提交配合使用：首先验证数据是否有效，然后将它提交到服务器。本章第二部分，我们将讨论使用Ajax请求实现HTML表单处理和提交。自行实现POST请求其实要进行低层次的数据处理，你要直接与服务器通讯，而不是由浏览器代劳。这个过程非常容易出错，不过如果正确实现了它，就能消除那些讨厌的浏览器页面刷新，令你的用户界面更为迅速。

6.1 客户端表单验证

什么是表单验证？简单地说，就是确认用户在表单中输入的数据是否有效。有效这个概念有点模糊，不过基本上是指表单中的数据要符合某些规则。

表单验证早已有之，最初多在服务端实现，后来转移到客户端，这使得用户能更快得到反馈。许多服务端框架比如Struts都提供了表单验证功能，但遗憾的是，对客户端验证的支持始终不够。

本节中，我们将看一看怎样亲手创建一个可扩展的客户端验证框架。我们会循序渐进，先是一些简单的验证功能，然后是"即输即验"式验证以及跨字段验证。

6.1.1 在客户端进行验证

在客户端进行验证可能非常……唔，可能不太有趣。显然，如果能创建一个程序库，然后每

个页面都重用这个程序库，而不是每次去重新编写一遍验证代码，那就好了。没错，这正是我们下面所要看到的。

注意，我们将使用面向对象的JavaScript，以便提高程序的可维护性和可扩展性。如果你需要复习一下这部分内容，请参考本书第3章。

1. 问题

你需要一个可重用的验证框架，有了它，你就不必重复编写客户端验证规则了。框架应该是可扩展的，并且易于维护和使用。

2. 解决方案

我们将使用Prototype库来协助编写验证框架。为什么使用Prototype？因为我们在第3章和第4章已经看到，使用一个好的JavaScript库，我们能更轻松地实现更加面向对象的验证框架，这有助于代码重用。大家都知道面向对象的代码易于重用！我们将使用Prototype库，但使用jQuery或Dojo工具包也能运用同样的概念。

请看图6-1，瞧瞧我们的验证框架有何本领。用户在填写一个需要数字值的字段时，显然是一不小心多打了一个“r”。现在，我们来看看这个页面的HTML代码和脚本，请看代码清单6-1。

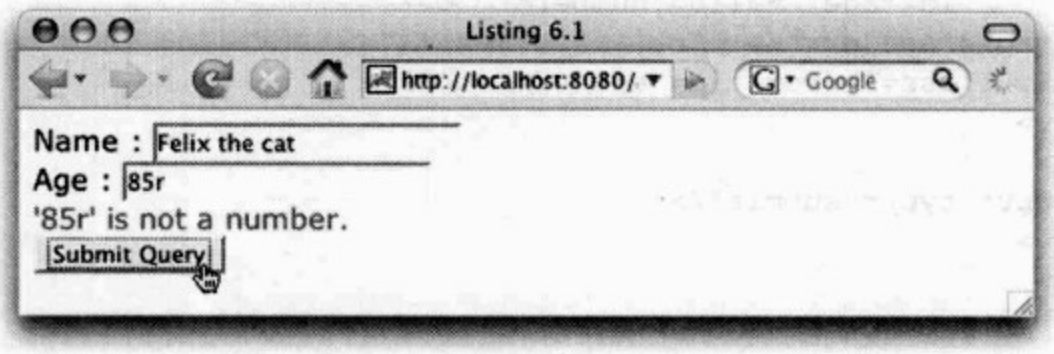


图6-1 有效性验证的实例

代码清单6-1 验证框架的HTML代码

```
<html>
  <head>
    <title>Listing 6.1</title>
    <script type="text/javascript"
            src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript"
            src="listing.6.2.js">
    </script>
    <script type="text/javascript">
      var framework = new ValidatorFramework();
      window.onload = function() {
        Event.observe('testForm','submit',
          function(event) {
            if (!framework.validateForm(event.target))
              Event.stop(event);
          }
        );
```

使用Prototype

包含验证框架

调用框架

```
    };
  </script>
  <title>Validation FrameWork</title>
  <style type="text/css">
    div.error {
      color: red;
    }
  </style>
</head>
<body>
  <form id="testForm"
        method="post"
        action="/aip.  chap6/requestInspector">
    <div> Name :
      <input name="name" type="text"
             id="name" valid="all"
             error="name_err"/>
      <div class="error" id="name_err"></div>
    </div>
    <div> Age :
      <input name="age" type="text"
             id="age" valid="number"
       error="age_err"/>
      <div class="error" id="age_err"></div>
    </div>
      <div>
        <input type="submit"/>
      </div>
  </form>
 </body>
</html>
```

将input标记为需要验证

使用数值验证器

代码非常简洁。我们通过input标签上的两个属性标出了希望进行验证的字段：valid属性告知验证器采用哪种验证规则；error属性告知验证器在哪里输出错误消息。

这里我们为HTML元素添加了自定义属性，这个技巧虽然很有用但也很麻烦，因为各种浏览器对此的处理方式不尽相同。IE和Safari会将这些元素标记上的自定义属性（attribute）作为DOM元素节点的对象属性（property）[①]，但Firefox和其他基于Gecko的浏览器并不这样处理。不过所有主流浏览器都允许通过元素的getAttribute()方法获取属性值。现在来看看我们的验证框架所包含的各个类（代码清单6-2）。

代码清单6-2　有效性验证框架

```
var Validator = Class.create();

Validator.prototype = {
  type: "all",

  initialize: function(validators) {
    validators[this.type] = this;
```

❶ 注册特定类型的有效性验证器

① Safari 3的行为也与FireFox一致，不会把自定义属性（attribute）转化为对象属性（property）。——译者注

```
  },

  doValidate: function(input) {
    return "";
  },

  validate: function(input, errordiv) {
    errorMsg = this.doValidate(input);
    errordiv.innerHTML = errorMsg;
    return (errorMsg.length == 0);
  }
}

var NumberValidator = Class.create();

Object.extend(NumberValidator.prototype,
              Validator.prototype);

Object.extend(NumberValidator.prototype, {
    type: "number",

    doValidate: function(input) {
      var numberpattern=/(^\d+$)|(^\d+\.\d+$)/;
      if (numberpattern.test(input)) {
        return "";
      } else {
        return "'" + input + "' is not a number." ;
      }
    }
});

var ValidatorFramework = Class.create();

ValidatorFramework.prototype =
{
  validators: 0,

  validateForm: function(form) {
    var retval = true;
    for(i = 0; i < form.length; i++) {
      currentInput = form[i];
      type = currentInput.getAttribute("valid");
      errorDivName = currentInput.getAttribute("error");
      if(type == null || errorDivName == null) {
        continue;
      } else {
        valid = this.validate(
          type, currentInput.value, $(errorDivName));
        if(!valid) {
          retval = false;
        }
      }
    }
    return retval;
```

❷ 定义有待子类改写的抽象方法

❸ 调用doValidate()得到输出

❹ 用于所有的数值字段

❺ 覆写doValidate()方法

❻ 定义有效性验证器的映射表

❼ 用适当的验证器进行有效性验证

```
  },

  validate: function(type, input, errordiv) {
    return this.validators[type].
      validate(input, errordiv);
  },

  initialize: function() {
    this.validators = new Object();
    new Validator(this.validators);
    new NumberValidator(this.validators);
  }

}
```

❽ 选择适当的验证器

❾ 设置映射表，注册验证器

3. 讨论

来讨论一下清单6-2中的代码。我们首先讨论Validator（有效性验证器）类。该类将作为我们所要实现的所有验证类的基类。我们把该类的验证类型设置为all，它在这里并没有什么意义，不过Validator类的子类将为它设置一个有意义的值。

ValidatorFramework类会用到这个类型属性。当它读取<input>元素的valid属性时，会与已注册的Validator对象的类型进行匹配来决定执行哪个Validator实现。在Validator的initialize()方法❶中，每个新的Validator对象都会对自身进行注册，通知ValidatorFramework实例建立从验证类型到验证器的映射。Validator类的核心是doValidate()方法❷，我们在该方法中评估用户输入是否有效。如果无效，doValidate()方法返回验证错误信息，否则返回一个空字符串表示验证成功。doValidate()方法由validate()方法❸调用，后者会刷新用于输出错误消息的 元素的内容，显示出validate()方法返回的验证错误消息，并根据是否有错误返回true或false。

我们已经明白了Validator基类如何工作，下面将看到如何（使用我们在第3章中学到的技巧）来实现自己的验证子类。我们从简单的NumberValidator类开始，它将改写doValidate()方法❺，利用正则表达式来验证所有的number类型的字段❹。你可以看到，创建自己的验证类非常简单：建立Validator的子类、设置适当的验证类型、创建一个相应的doValidate()方法。

没有框架来驱动，我们的Validator类就无法发挥效用，这就是引入ValidatorFramework类的原因。该类包含一个叫做validators的hash（映射表），包含所注册的所有Validator对象❻，这一过程是在这些Validator对象的initialize()方法❾中完成的。当我们从用户界面中调用框架的validateForm()方法时，方法会遍历指定表单中的所有<input>标签，寻找对应的验证器❼，对输入进行验证❽。

如果我们运行该示例并输入无效数据，会有什么结果呢？你可以在前面的图6-1中看到，85r的确不是一个数字。我们的验证框架确实有效！

建立上面这样的有效性验证框架并不困难，特别是我们采用了面向对象的方式。你会发现这样的框架易于维护，同时也易于扩展——例如，增加新的验证规则。通过给错误信息 指定CSS类，你能在整个应用中保持一致的视觉效果（look and feel）。

关于客户端验证，我们有个告诫：它并不保证安全！之前我们已经提到过该问题，但还是值

得再次重申一遍。即使已经在客户端对数据进行了有效性验证，你始终必须在服务端再次进行验证。心怀不轨的用户能轻易仿造HTTP POST请求，绕过你精心搭建的验证框架，从而提交他们所希望的任意数据。

你或许可以再看一下第5章，回忆一下如何创建由服务器协助的有效性验证。这种混合式验证能减少客户端代码并缓解了对服务器端Java和客户端JavaScript进行同步维护[①]的棘手问题。

我们刚才实现的验证框架还有一个缺点，用户直到提交表单时才知道输入数据是否正确。更加友好的方式或许是在用户输入错误数据时立即加以提醒。我们下一节的例子就能向用户即时反馈他们的输入是否有效。

6.1.2 即时验证

你很可能已经见过一些Web应用会在你键入数据的同时验证输入是否有效并提供即时反馈。这节省了用户的时间并减少了挫折感，因为用户在点击表单提交按钮之前已知道输入是有效的。本例扩展了我们的验证框架以支持这一特性。

1. 问题

你希望在用户键入数据时给予他们关于输入数据有效性的即时反馈。

2. 方案

用户在键入数据时能看到自己输入的数据是否已有错误，而不是完整填写完表单之后在提交表单时才发现输入错误——这样的方式当然很棒。

整本书我们都在大谈代码重用。本书作者不希望被别人指责为“全是帽子，没有牛群”（见http://en.wiktionary.org/wiki/all_hat_and_no_cattle）[②]，所以我们将重用我们创建的验证框架来实现即时输入验证。事实上，我们所做的唯一改变就是为表单添加一个keyup事件处理器。首先我们来看图6-2，当用户在数字字段中输入一个r之后，将立即显示出错误消息。

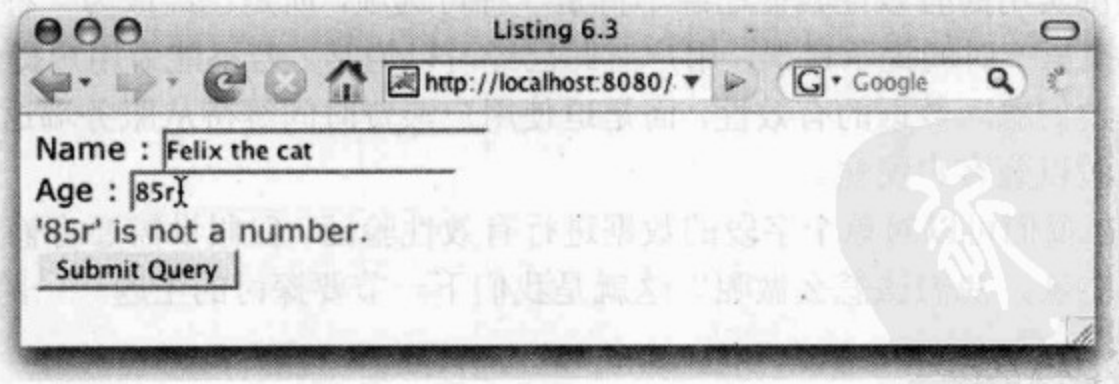


图6-2 输入数据时的即时反馈

代码清单6-3只展示了onload事件处理器，而没有重复列出整个清单6-1的全部代码，新加代码以粗体显示。

① 这是指要确保服务器端和客户端的有效性验证逻辑的一致性。归根到底，要得到最好的用户体验，这个问题是无法回避的。——译者注

② 美国西部得克萨斯州谚语，只有牛仔帽而没有牛群的牛仔，相当于中国人所说的“干打雷不下雨”、“夸夸其谈”、“光说不练”、“说一套做一套”等。最近这个词常用于乔治·布什身上，因为他正好是得克萨斯人。——译者注

代码清单6-3　即输即验式的表单验证

```
window.onload = function() {
  Event.observe('testForm','submit',function(event) {
    if (!framework.validateForm(event.target))
      Event.stop(event);
  });
  Event.observe('testForm','keyup',function(event) {
    framework.validateForm(event.target.form);
  });
};
```

3. 讨论

本节的解决方案并没有对上一节的方案做非常大的改动。然而，现在当我们输入一个数字时就会得到一个即时反馈。

但它是怎么工作的呢？你可能自言自语：“onkeyup事件处理器绑定到了<form>，而不是绑定到我正在输入数据的那个<input>标签！”你观察得很仔细。但是记住，在浏览器的事件模型中，事件会在DOM树上传播。因此，如果某个<input>元素上触发了keyup事件，它将在冒泡阶段向上冒泡到包含那个<input>的<form>元素，其onkeyup事件处理器就会被调用并运行我们的验证器。所以我们不需要费劲地为每个<input>元素都添加onkeyup事件处理器。

该解决方案有个麻烦问题：当我们正在姓名字段中输入内容时，年龄字段还是空白的，所以会导致显示出一个错误消息。你该怎样修改我们的验证框架来消除这个讨厌的问题呢？（提示：考虑使用onfocus和onblur事件处理器，根据字段是否获得焦点来启用和禁用即时验证。）

另一个问题是，每次keyup事件触发时会对整个表单执行验证，这是没有必要的。毕竟，我们在一个时刻只能改变一个字段的内容。你该怎样修改或扩展验证框架来允许对个别字段进行验证呢？还有，如何用它来帮助我们解决刚才提到的问题？

为用户提供实用的有效性验证已经不再是一种可选项，而是已经成为一种必需品。大量易用的Web应用会在第一时间提示错误，用户对此已经习以为常。若不能为用户提供一种方式在提交数据前就确保他们输入数据的有效性，而是迫使用户浪费时间等待从服务端返回验证结果，就必将导致用户不满以至客户流失。

好了，现在我们可以对单个字段的数据进行有效性验证。但假设需要将输入的数据结合其他字段数据进行验证，我们该怎么做呢？这就是我们下一节要探讨的主题——跨字段验证。

6.1.3　跨字段验证

跨字段验证不太关心单个元素值的数据格式是否正确（是数字？是电子邮件地址？），而是关注于两个或更多输入字段的值是否符合业务规则对这些数据在相互关系上的要求。例如，设想有一个表单需要用户输入某个活动的起始日期和结束日期。显然，起始日期必须早于结束日期。我们的普通验证对象能检查用户所输入的是否是有效的日期值，如果有效，我们将运行跨字段验证器来确保起始日期和结束日期在逻辑上也是正确的，也就是说起始日期早于结束日期。

1. 问题

你需要验证多个字段之间的关联是否是有效的。

2. 方案

让我们着手开发跨字段的验证机制。我们将在前面开发的基础上进行扩展，添加跨字段验证的能力。首先来编写HTML以建立我们的表单和跨字段验证类（代码清单6-4）。

代码清单6-4 跨字段验证示例的HTML

```
<html>
  <head>
    <title>Listing 6.4</title>
    <script type="text/javascript"
            src="prototype-1.5.1.js"></script>
    <script type="text/javascript"
            src="listing.6.5.js"></script>
    <script type="text/javascript">
      var framework = new ValidatorFramework();
      var xref1;
      var xref2;

      window.onload = function() {
        Event.observe('testForm','submit',function(event) {
          if (!framework.validateForm(event.target))
            Event.stop(event);
        });
        Event.observe('testForm','keyup',function(event) {
          framework.validateForm(event.target.form);
        });
        xref1 =                                          跨字段验证第一组日期
          new DateRangeCrossValidator(
            framework,
            new Array($('start'),$('end')),$('startend_err'));
        xref2 =                                          跨字段验证第二组日期
          new DateRangeCrossValidator(
            framework,
            new Array($('start2'),$('end2')),$('startend_err2'));
      };
    </script>
    <style type="text/css">
      div.error {
        color: red;
      }
    </style>
  </head>
  <body>
    <form id="testForm"
          method="post"
          action="/aip.chap6/requestInspector">
      <div id="startend_err"                             指定用于显示错误消息的<div>
           class="error"></div>
      <div> Start Date :
        <input name="start" type="text"                  定义起始日期
               id="start" valid="date"
               error="start_err"/>
```

```
    <div class="error" id="start_err">
    </div>
  </div>
  <div> End Date :
    <input name="end" type="text"
           id="end" valid="date"
           error="end_err"/>
    <div class="error" id="end_err"></div>
  </div>

  <div id="startend_err2"
       class="error"></div>
  <div> Start Date :
    <input name="start2" type="text" id="start2" valid="date"
           error="start_err2"/>
    <div class="error" id="start_err2"></div> </div>
  <div> End Date :
    <input name="end2" type="text" id="end2" valid="date"
           error="end_err2"/>
    <div class="error" id="end_err2"></div>
  </div>
  <div>
    <input type="submit"/>
  </div>
</form>
  </body>
</html>
```

指定另一个用于显示错误消息的

定义结束日期

定义第二组日期

基本上，我们为每个<input>元素定义了一个与之对应的用于显示错误消息的 元素，也为每个跨字段验证组定义了一个错误 用来显示该组的错误消息。然后我们构造了跨字段验证器并把需要校验的<input>元素的引用传给它们。注意观察图6-3和图6-4，然后研究代码（代码清单6-5）是如何实现这些功能的。为了实现跨字段验证，我们对验证框架做了一些改动。

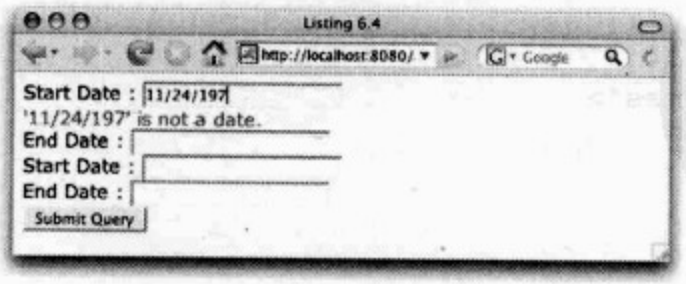


图6-3　验证单个数据字段

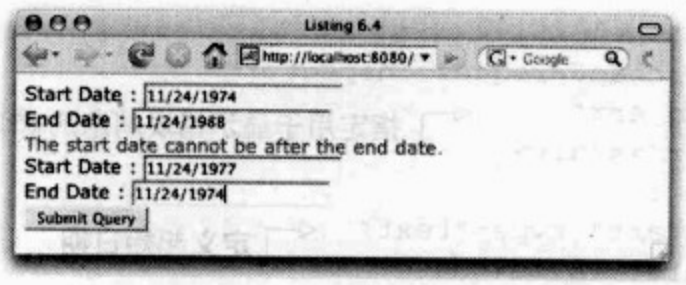


图6-4　跨字段验证的实例

代码清单6-5 跨字段验证框架

```
var Validator = Class.create();

Validator.prototype = {
  type: "all",

  initialize: function(validators) {
    validators[this.type] = this;
  },
  doValidate: function(input) {
    return "";
  },

  validate: function(input, errordiv) {
    errorMsg = this.doValidate(input);
    errordiv.innerHTML = errorMsg;
    return (errorMsg.length == 0);
  }
}

var NumberValidator = Class.create();

Object.extend(NumberValidator.prototype,
              Validator.prototype);

Object.extend(NumberValidator.prototype, {
    type: "number",

     doValidate: function(input) {
      var numberpattern=/(^\d+$)|(^\d+\.\d+$)/;
      if (numberpattern.test(input)) {
        return "";
      } else {
        return "'" + input + "' is not a number." ;
      }
    }
});

var DateValidator = Class.create();
Object.extend(DateValidator.prototype, Validator.prototype);
Object.extend(DateValidator.prototype, {
  type: "date",

   doValidate: function(input) {
    var value = Date.parse(input);
    if(value <= 0) {
      return "'" + input + "' is not a date.";
    } else {
      return "";
    }
  }
});
```

❶ 创建一个新的类

❷ 将输入的字符串解析为date对象

```
var ValidatorFramework = Class.create();
ValidatorFramework.prototype =
{
  validators: 0,
  crossValidators: 0,    ← 保存所有的跨字段验证器

  validateForm: function(form) {
    var retval = true;
    for(i = 0; i < form.length; i++) {
      currentInput = form[i];
      type = currentInput.getAttribute("valid");
      errorDivName = currentInput.getAttribute("error");
      if(type == null || errorDivName == null) {
        continue;
      } else {
        valid = this.validate(type, currentInput.value,
          $(errorDivName));
        if(!valid) {
          retval = false;
        }
      }
    }
    for(i = 0;                    ← ❸ 遍历所有验证器
        i < this.crossValidators.length; i++) {
      this.crossValidators[i].clearErrors();
    }
    if (retval) {                 ← ❹ 检查验证器的完成状态
      for(i = 0; i < this.crossValidators.length; i++)  {
        valid = this.crossValidators[i].validate();
        if(!valid) {
          retval = false;
        }
      }
    }
    return retval;
  },

  validate: function(type, input, errordiv) {
    var validator = this.validators[type];
    if(!validator) {
      alert("No validator for type '" + type + "'.");
      return "";
    }
    return validator.validate(input, errordiv);
  },

  initialize: function() {
    this.validators = new Array();
    this.crossValidators = new Array();

    new Validator(this.validators);
    new NumberValidator(this.validators);
```

```
    new DateValidator(this.validators);
  }

}

var CrossValidator = Class.create();                    ❺ 创建CrossValidator类
Object.extend(CrossValidator.prototype, {
    type: "none",
    crossError: 0,
    crossInputs: 0,

    initialize: function(framework,                     a. 定义initialize()函数
                         p_crossInputs,
                         p_crossError) {
      framework.crossValidators.push(this);             b. 向框架注册验证器
      this.crossError = p_crossError;
      this.crossInputs = p_crossInputs;
    },

    validate: function() {
      errorMsg = this.doValidate(                       c. 验证并设置错误消息
        this.crossInputs);
      this.crossError.innerHTML = errorMsg;
      return (errorMsg.length == 0);
    },

    clearErrors: function() {                           d. 清除之前产生的错误消息
      this.crossError.innerHTML = "";
    }
});

var DateRangeCrossValidator =                           ❻ 扩展CrossValidator类
  Class.create();
Object.extend(DateRangeCrossValidator.prototype,
              CrossValidator.prototype);
Object.extend(DateRangeCrossValidator.prototype, {

  doValidate: function(inputs) {
    var startDate = Date.parse(inputs[0].value);
    var endDate = Date.parse(inputs[1].value);
    if (startDate > endDate) {
      return "The start date cannot be after the end date.";
    } else {
      return "";
    }
  }

});
```

3. 讨论

来看一下我们在代码清单6-5中做了什么。我们需要一个Validator实现来处理日期型数据，所以我们添加了一个验证类❶并实现正确的doValidate()方法❷。我们将使用JavaScript内置的

Date类来判定输入是否有效。注意，该函数不能用于1970年1月1日之前的日期[①]。如果需要更大的灵活性，你或许可以考虑使用正则表达式。

我们并不需要对验证框架做很多改动。不过注意，如果不是所有输入都有效，我们不会进行跨字段验证，但是我们仍需要清除任何由上一次跨字段验证所产生的错误消息❸。如果不这样做，用户可能在输入数据时看到一些无意义的错误消息。一旦所有的常规字段验证器都验证通过❹，我们就执行一次跨字段有效性验证。

因为现在要实现跨字段的有效性验证，所以我们创建一个跨字段验证器的基类❺。该类和普通Validator类很相像，它的initialize()方法现在接受3个参数：与验证器相关联的验证框架的引用，要进行验证的<input>元素数组（此处次序很重要，查看你使用的验证器以确定次序），以及用于显示错误消息的 元素的引用。

此外，我们还创建了跨字段验证器的子类❻DateRangeCrossValidator类。该类将检查两个日期值是否确实符合时间先后顺序。这个验证器接受两个参数，首先是开始日期<input>字段的引用，其次是结束日期<input>字段的引用。

至此，你已经拥有一个可重用并可扩展的跨字段验证框架！请回顾一下图6-3和图6-4，看看它在浏览器中的表现如何。你可以看到，只有所有<input>字段都输入正确才会进行跨字段有效性验证，所以即使第二组开始/结束日期有错误，也不会显示出错误信息。当我们修正了第一个日期问题（记住，我们的DateValidator类不能处理早于1970年的日期）之后，我们应该能看到如图6-4的结果。第一组有效，第二组则无效。

跨字段验证当然并不局限于开始日期和结束日期。本人真切地记得有个应用成功地使用了跨字段验证技术。这个应用是为电力贸易而设计的，因此必须对发电设备的可用容量了如指掌。在电力设备的并行可用性、给定时间段内的发电量以及整个设备的计划发电总量之间，存在多重的相关性。我们只有一个相关性简表，但概念是很清晰的。在实施一个电能计划之前，必须先确保计划的有效性。光用户界面的绘制就要花上好一会儿，还有大量的数据要提交到服务器端。不用说，要花很长的时间才能完成验证。为了加快检查电能计划的速度，我们开发了一个大型验证框架，它可在几秒之内完成整个计划的有效性验证。而老的方法可能要花上两分钟。可见跨字段验证器确实是个好东西。

目前，我们已经完成了数据验证并准备好把数据送上服务器。但我们会怎样发送它呢？告诉你一个好消息，下一节将介绍数据传输问题！

6.2　投递数据

在上一章中，我们看到了怎样使用Prototype以Ajax方式来投递（post）表单。本节，我们则会深入研究如何通过XMLHttpRequest对象来自行模拟表单的投递过程。我们可以使用XHR对象

① 原作者的代码逻辑有些奇怪。Date完全可以用于1970年以前的日期，对于早于1970年的时间，Date.parse()返回一个负数。所以代码中判定value小于等于0即为日期无效当然也是说不通的，而且恰好不能捕获到真正无效的日期输入——Date.parse()如果解析失败会返回NaN，而NaN与任何值进行比较都返回false，结果就是所有解析失败的值都被认为是有效的日期！合理的代码应为：if (isNaN(value))...。——译者注

来发起HTTP请求并指定请求的类型，所以只要组织好通过XHR传递给服务器的数据，就能模拟表单的提交。

在展示如何发起Ajax POST请求后，我们将看看Prototype之外的另一种选择——jQuery，了解如何用它发起POST请求。

6.2.1 POST 请求剖析

POST到底长啥样？问得好！下面我们就来看一看。首先我们需要一个表单（代码清单6-6）。

代码清单6-6 一个简单的表单

```
<html>
  <body>
    <form method="post" action="http://localhost:2020/xyz">
      <input type="text" name="input1"/>
      <input type="text" name="input2"/>
      <input type="text" name="input3"/>
      <input type="submit"/>
    </form>
  </body>
</html>
```

好了，现在我们有了一个表单。该表单的action属性看起来有些让人疑惑。谁在端口2020侦听？答案是……我们自己！通过一个叫做Netcat（http://netcat.sourceforge.net）的工具，我们可以侦听任意端口，并将该端口传输进来的数据打印到命令行输出上。有了Netcat，我们就能轻松监测网络数据了。

要用Netcat在端口2020上侦听，我们输入下面这行命令：

```
netcat -l -p2020
```

让我们看看对于图6-5那样简单的表单，客户端会向服务器投递些什么。

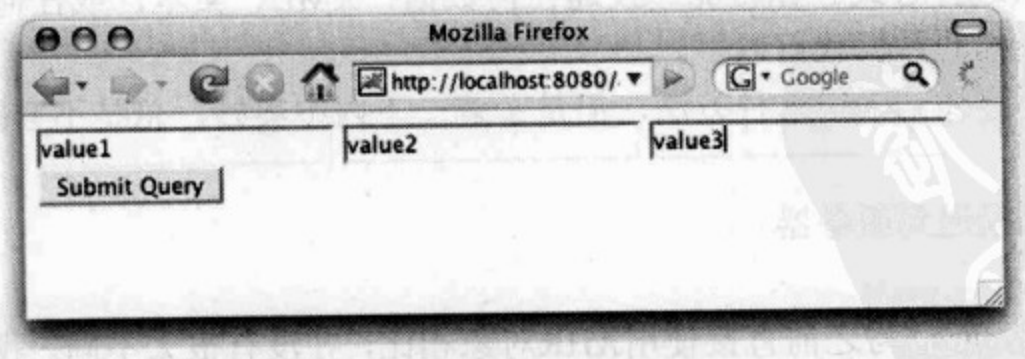


图6-5 一个简单的表单

当我们提交表单时，发送到（我们正通过Netcat侦听的）端口2020上的数据大体如下（根据浏览器及其设置，结果会稍有不同）：

```
POST /xyz HTTP/1.1
Host: localhost:2020
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X;
```

```
    ➥ en-US; rv:1.8.1.2) Gecko/20070219 Firefox/2.0.0.2
Accept: text/xml,application/xml,application/xhtml+xml,
  ➥ text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 32

input1=value1&input2=value2&input3=value3
```

这里无疑有许多有趣的信息。不过就模拟表单投递而言，我们只需对其中几个部分加以说明。

第一行必不可少，它包括请求的方法（此处是POST）、请求的资源以及使用的HTTP协议。之后是一系列HTTP首部，它们由冒号分隔的键/值对构成。其中只有Host首部是HTTP协议1.1要求必须发送的。其他首部信息是可选的，但强烈推荐使用。

HTTP头部之后有一个空行，它后面跟随的是POST请求的主体。对一般的内容类型（content type）来说，在表单提交时，请求主体是由经过URL编码的表单数据构成的。

它看起来和GET请求的查询字符串有些类似，不是吗？事实上，GET的查询字符串和POST的请求主体遵循相同的格式。

那么，我们该如何利用XHR对象来投递表单？这个问题将在下节详细讲解。注意我们的第一个方案并不是用来解决表单投递问题的，它的目的是向你展示如何投递任意类型的数据，稍后我们会对表单投递做专门讨论。

6.2.2　将数据投递到服务器

之前我们已对POST请求进行了分析，本节我们将应用从中学到的知识，学习如何通过XHR对象自行向服务器发起POST请求。当你需要将数据提交给服务器但是又不希望强制浏览器刷新页面时就可以采用这种方式。你将能够投递任何数据：XML、文本，或任何其他类型的数据。你只需适当地构造你的数据就可以了。

虽然你大概不会经常采用这种方法，但是了解一下内中奥妙，总是不会错的。

1．问题

你需要把数据投递到服务器。

2．方案

方案相当简洁，而且与之前直接使用XHR对象相比，并没有很大不同。我们简单地获得一个XHR对象，设置其相关属性，执行到某个URL的POST，并传送数据。注意，我们不是在模拟HTML表单的数据投递方式，那是我们下一个解决方案的议题。本方案可用于提交任何数据到服务器。

代码清单6-7列出了我们自行模拟POST请求的示例代码。注意，该代码在IE 6浏览器中无法工作。我们知道，IE 6使用ActiveX控件创建XHR对象。之前我们早就讨论过如何以跨浏览器的方式创建XHR对象，所以这里为了将注意力聚焦于数据的投递机制，我们就简单地假设该示例是运行在Firefox、Safari和IE 7中。

当我们在浏览器中加载该示例时，会看到两个提示框接连弹出。首先我们看到如图6-6所示的客户端通知，在我们点击OK按钮后，第二个提示框，如图6-7所示，向我们显示从服务器收到一条消息。

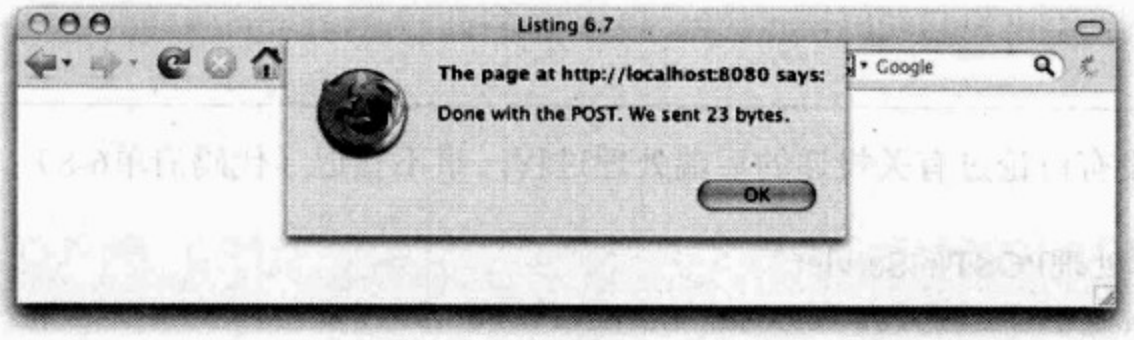


图6-6 客户端通知

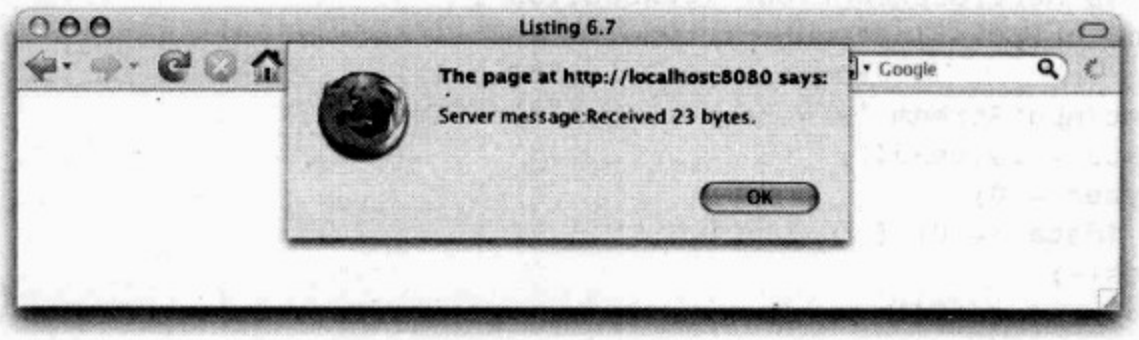


图6-7 服务端通知

代码清单6-7 投递数据到服务器

```
<html>
  <head>
    <title>Listing 6.7</title>
    <script type="text/javascript">
      window.onload = function () {
        var data = "This is just some data.";
        var url = "/aip.chap6/postServlet";
        var xhr = new XMLHttpRequest();          ← 创建XHR实例
        xhr.onreadystatechange = function () {
          if (xhr.readyState == 4 && xhr.status == 200) {
            alert('Done with the POST. We sent ' + data.length +
                  ' bytes.');
            alert('Server message:' + xhr.responseText);
          }
          else if (xhr.readyState == 4) {
            alert('Error posting. Server status: ' + xhr.status);
          }
        };
        xhr.open('POST',url,true);               ← 指定使用POST方法
        xhr.setRequestHeader(
          "Content-Type", "whatever");           ← 指定数据的内容类型
        xhr.setRequestHeader(
          "Content-Length",data.length);         ← 指定内容长度
        xhr.setRequestHeader(
```

```
            "Connection", "close");   <-- 指示是单个请求
          xhr.send(data);
        };
      </script>
    </head>
    <body>
    </body>
  </html>
```

我们还真没有讨论过有关投递的后端处理过程，事不宜迟（代码清单6-8）。

代码清单6-8　处理POST的Servlet

```
public class PostServlet extends HttpServlet {
  protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
      throws ServletException, IOException {
    System.out.println("Content type: " +
                       request.getContentType());
    ServletInputStream is = request.getInputStream();
    int data = is.read();
    int bytes = 0;
    while (data >= 0) {
      bytes++;
      data = is.read();
    }
    response.getWriter().write("Received " + bytes + " bytes.");
  }
}
```

这个简单的servlet仅仅接收请求的主体并输出已接收数据的字节长度作为响应。

3. 讨论

数据投递看上去相当简单。在xhr.open()方法中，我们不再使用HTTP GET方法，而是直接指定为POST。可以把内容类型设置为任何我们想要的类型，比如要投递XML文件，你可以将该值设置为text/xml。我们还需要设定Content-Length请求首部。尽管HTTP协议没有强制要求，但这非常重要！如果值设置得过小，你可能丢失数据，因为服务器可能认为数据已经发送完毕而事实上还没有。同理，如果值设得过大，服务器将会挂起，等待接收更多数据。在正确设定了所要发送数据的长度后，我们使用xhr.send()方法来发送实际数据。

这个方式看似简单，但功能相当强大。你可以用它发送任意类型的数据到服务器，例如XML或JSON。因为服务器同样可以回送数据到客户端，所以你可以使用该机制来作为一种远程过程调用：投递一些数据，服务器对其进行处理，返回数据给客户端，客户端处理数据并刷新UI。IBM的JavaScript SOAP/web services库正是这样进行SOAP消息交换的（详见http://www-128.ibm.com/developerworks/web/library/ws-wsajax/）[①]。

现在你已了解了如何完成到服务器的投递。接下来，我们将聚焦于表单的投递。

① 本书第2章中也介绍过它。——译者注

6.2.3 将表单数据投递到服务器

既然你已明白了POST请求的背后机理，那就让我们来看看如何使用Ajax把表单投递到服务器。我们在5.5.2节已经介绍过一个类似示例(在没有页面提交的情况下投递表单元素)，但那个示例关注的是事件处理，本节将探究POST本身并测试更多种类的表单元素。

另外，前面章节中的示例使用Prototype库处理请求，而本节将采用在第4章介绍过的另一个库：jQuery。

1. 问题

你需要模拟HTML表单投递。

2. 方案

首先，我们将创建一个表单，它包含多种控件元素，以测试POST请求是否能成功地将所有数据提交到服务器。未输入任何数据之前的空白表单如图6-8所示。

然后，我们输入各种数据，表单的“正常”提交将转由Ajax控制。代码清单6-9列出了代码。如果你需要温习jQuery，现在就可以先回顾一下4.3节。

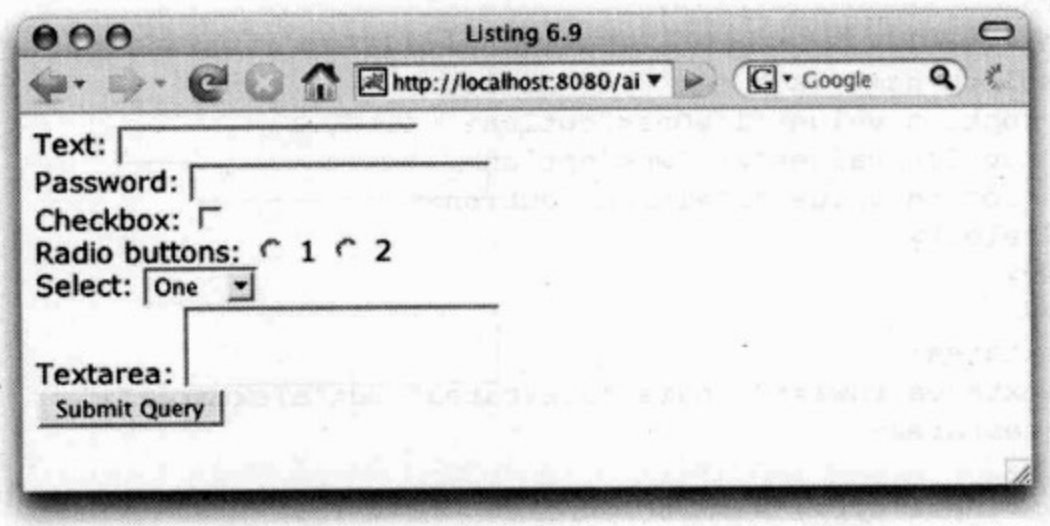


图6-8 POST测试表单

代码清单6-9 利用jQuery将表单的POST请求重定问

```
<html>
  <head>
    <title>Listing 6.9</title>
    <script type="text/javascript"
            src="jquery.js"></script>
    <script type="text/javascript"
            src="jquery.form.js"></script>
    <script type="text/javascript">
      $(document).ready(function() {
        $('#testForm').ajaxForm({
          type: 'POST',
          target: '#results'
        });
      });
    </script>
  </head>
```

❶ 导入jQuery程序库

❷ 导入jQuery表单插件

❸ 为Ajax提交对表单进行预处理

```
<body>
  <form id="testForm"
        action="/aip.chap6/requestInspector">
    <div>
      Text:
      <input type="text" id="aTextField" name="aTextField"/>
    </div>
    <div>
      Password:
      <input type="password" id="aPassword" name="aPassword"/>
    </div>
    <div>
      Checkbox:
      <input type="checkbox" id="aCheckbox" name="aCheckbox"/>
    </div>
    <div>
      Radio buttons:
      <input type="radio" name="aRadioGroup" id="aRadio1"/> 1
      <input type="radio" name="aRadioGroup" id="aRadio2"/> 2
    </div>
    <div>
      Select:
      <select name="aSelect" id="aSelect">
        <option value="1">One</option>
        <option value="2">Two</option>
        <option value="3">Three</option>
      </select>
    </div>
    <div>
      Textarea:
      <textarea rows="2" name="aTextarea" id="aTextarea">
      </textarea>
    </div>
    <div><input type="submit"/></div>
  </form>
  <div id="results"></div>
</body>
</html>
```

❹ 用于服务器响应的容器

代码看上去非常简单。如你所见，使用jQuery❶以及它的表单插件❷，完成表单投递易如反掌——简直像是用了电玩秘技！

ajaxForm()方法并没有提交表单，而是对表单进行预处理，以便当该表单的submit事件最终触发时在Ajax控制下提交表单。虽然没有深入该插件代码，但你可以根据从本书中学到的知识，想象一下要达到该目的需要采取哪些步骤。请思考一下，如何将submit事件、事件处理机制以及上个方案中XHR的经验融汇贯通，实现这一功能。

传入ajaxForm()方法❸的选项表指定了请求类型为POST，还指定了target，即用于显示响应结果的DOM元素。此处的target选项指定了一个空 元素❹，定义于页面底部。注意，按照“jQuery风格”，我们使用CSS选择器来确定目标元素。

本示例所用到的服务端资源是ParameterInspectorServlet类，我们在第5章的一些示例中曾用到过它。所以这里略过该类的细节，只需记得该类会收集所提交的参数并按照一定格式显示出来。

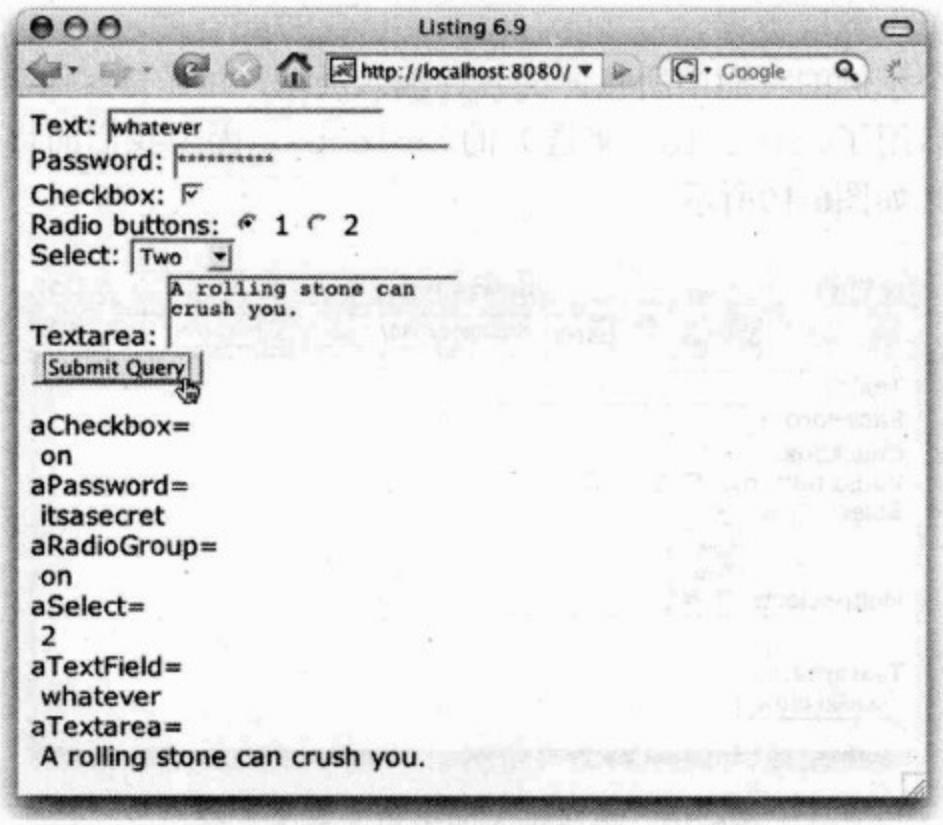


图6-9 拦截表单提交

在表单上填写一些数据然后提交它，我们的页面显示结果如图6-9所示。

3. 讨论

你已经看到了，通过使用“秘技”和使用jQuery及其表单插件，我们就能确保拦截表单提交并转由Ajax投递到服务器，而且这只需寥寥数行代码。

从用户角度看，以这种方式模拟表单投递具有很大的优点。浏览器不必刷新，不必从服务器获取用户界面（很可能它没有改变），这节省了大量时间。从开发者角度看，由于不再需要重新生成用户界面并将页面发送到客户端，就节约了服务器运行周期和带宽。当然，你需要开发客户端代码来实现这一新的运转方式，现在我们不再强制刷新浏览器了，所以这些代码的主要任务是保持用户界面状态连贯有序。

6.2.4 检测表单数据变化

为了减小网络数据流量和数据库访问，有时候我们有必要只把客户端表单中确实发生改变的部分发送到服务器。在5.5.3节中，我们阐述过该问题，着重介绍了如何使用事件处理来实现它，但我们略过了处理实际表单元素的细节。在本节中，我们更深入的介绍了如何以一种智能的方式处理表单元素。我们还会使用jQuery，希望它能给你提供关于事件处理和DOM操作的另一种视角。

1. 问题

你想要检测表单数据是否发生改变，并只把发生变化的数据发送到服务器。

2. 方案

我们解决该问题的方式与5.5.3节相当类似。我们在<form>元素上设置一个onchange事件处理器，它对那些值发生改变的控件添加一个fieldChanged类名，我们还安置了一个onsubmit事件处理器用于拦截表单提交。但是这一回，我们不会再回避处理表单元素的差异，而是特别关注

怎样从所要提交的表单中抓取数据。

我们将沿用上一个示例的页面中所包罗的各种输入类型，并补充其他需要的内容。我们将添加一个新控件，一个启用了multiple（多选）的<select>。因为我们的目标就是最好增加一些复杂性，该页面显示时如图6-10所示。

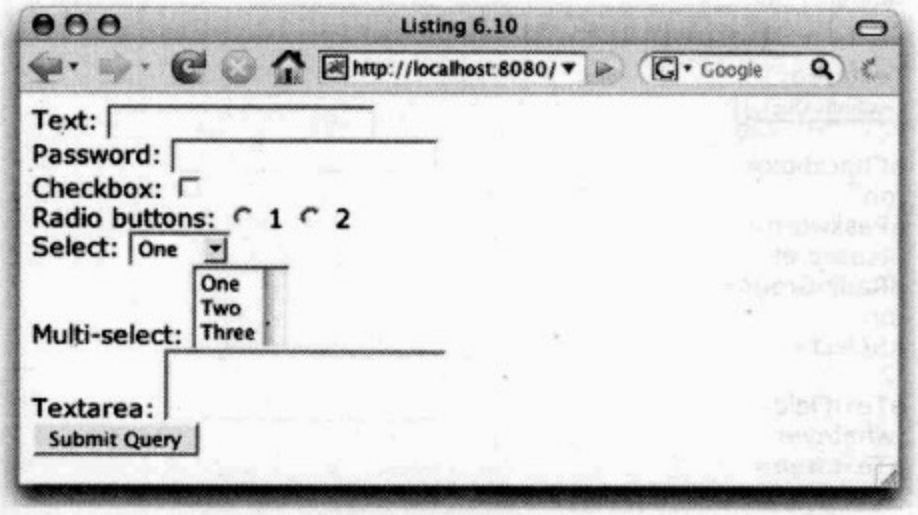


图6-10　等待录入的表单元素大汇聚

现在来看该页面的代码（代码清单6-10）。我们添加了好些脚本以便只提交改变过的元素。

代码清单6-10　只提交发生改变的元素值

```
<html>
  <head>
    <title>Listing 6.10</title>
    <script type="text/javascript"
            src="jquery.js"></script>
    <script type="text/javascript">
      $(document).ready(function() {
        $('form').bind("change",
          function(event) {
            $(event.target).addClass('fieldChanged');
          }
        );
        $('form').bind("submit",
          function(event) {
            submitForm();
            return false;
          }
        )
        trackCheckboxes();
      });

      function trackCheckboxes() {
        $('input[@type=checkbox]').each(function() {

  $(this).bind('change',function() {return false;});
          var hidden = document.createElement('input');
          hidden.type = 'hidden';
          hidden.name = this.name;
```

❶ 给表单绑定change事件处理器

❷ 给表单绑定submit事件处理器

❸ 跟踪复选框的变化

```
    this.name = '_' + this.name;
    this.hidden = hidden;
    this.form.appendChild(hidden);
    $(this).bind('click', function() {
      var onOff = $(this).attr('checked') ? 'on' : 'off';
      this.hidden.value = onOff;
      $(this.hidden).addClass('fieldChanged');
    });
  });
}

function submitForm() {          ❹ 处理表单提交
  var params = {};
  $('#testForm .fieldChanged').each(function() {
    if (this.disabled) return;
    if (this.name.length==0) return;
    if ((this.type=='radio' || this.type=='checkbox') &&
      !this.checked) return;
  if (this.type=='reset') return;
  if (this.type=='multiple' ||
      this.type=='select-multiple') {
    for(n = 0; n < this.length; n++) {
      if (this[n].selected)
        addParam(params,this.name,this[n].value);
    }
    else {
      addParam(params,this.name,this.value);
    }
  });
  $.post(          ❺ 投递到服务器
    $('#testForm').get(0).action,
    params,
    function(data) {
      $('#results').empty().append(data);
    }
  );
  $('.fieldChanged')          ❻ 复位状态
    .removeClass('fieldChanged');
}

function addParam(params,name,value){          ❼ 收集参数和值
  if (!params[name]) params[name] = new Array();
  params[name].push(value);
}
</script>
<style>
.fieldChanged {
  border: 1px solid red;
}
</style>
</head>

<body>
```

```
<form id="testForm"
      action="/aip.chap6/requestInspector">
  <div>
    Text:
    <input type="text" id="aTextField" name="aTextField"/>
  </div>
  <div>
    Password:
    <input type="password" id="aPassword" name="aPassword"/>
  </div>
  <div>
    Checkbox:
    <input type="checkbox" id="aCheckbox" name="aCheckbox"/>
  </div>
    <div>
      Radio buttons:
      <input type="radio" name="aRadioGroup" id="aRadio1"/> 1
      <input type="radio" name="aRadioGroup" id="aRadio2"/> 2
    </div>
    <div>
      Select:
      <select name="aSelect" id="aSelect">
        <option value="1">One</option>
        <option value="2">Two</option>
        <option value="3">Three</option>
      </select>
    </div>
    <div>
      Multi-select:
      <select name="aMultiSelect" id="aMultiSelect"
              multiple="multiple" rows="3">
        <option value="1">One</option>
        <option value="2">Two</option>
        <option value="3">Three</option>
      </select>
    </div>
    <div>
      Textarea:
      <textarea rows="2" name="aTextarea" id="aTextarea">
      </textarea>
    </div>
    <div><input type="submit"/></div>
  </form>
  <div id="results"></div>
 </body>
</html>
```

在jQuery库所提供的文档ready()处理器中，我们给<form>绑定了一个onchange事件处理器，<form>包含的任何一个元素上激发了change事件❶都会触发该事件处理器。处理器会给字段添加fieldChanged类名，以此标记字段发生了改变。在我们的示例页面中，我们添加了一个CSS规则，为这些发生改变的元素画上红色边框以便于诊断调试。它让我们在调试代码时可以一眼看出某个元素是否应用了fieldChanged类。你最终发布的代码中或许可以去掉这些代码（除非

你的需求要求告知用户哪些字段已经改动过）。

我们还为表单元素添加了onsubmit事件处理器❷，这样我们就可以中止常规的表单提交流程，改由我们自己处理。

我们在5.5.3节中已经见过类似的处理，但这次我们采用的是jQuery而不是Prototype来安置事件处理器。

在ready()处理器的末尾，我们调用了一个名为trackCheckboxes()❸的函数，它对所有的复选框输入元素做了一些特殊处理。这里讨论它还为时尚早，等我们了解了示例的其他部分后再讨论它。

大部分有趣的事情发生在<form>元素上的onsubmit事件处理器，也就是submitForm()函数中❹。在该函数中，我们遍历所有标记为fieldChanged的元素并在params这个hash对象中建立参数列表以备提交。不过注意，我们相当挑剔！

经过一番筛选，我们剔除了所有不应该被提交的元素。它们包括被禁用的字段、没有名字（name为空）的字段、未选中的单选框或复选框控件元素，还有所有重置（reset）元素（按照W3C规则，决不应该提交reset）。

然后我们收集那些通过筛选测试的元素的值。但这可不仅仅是简单的名值对问题。我们不仅需要处理多选元素，还需谨记多个控件可能使用同一个名字。这意味着对于每一个参数名，可能会有多个值对应。addParam()函数❼通过创建数组存储多个参数值来解决该问题。

最后，在收集了所有发生改变的元素值之后，我们使用jQuery提供的$.post()函数将请求投递回服务器❺并给所有标记过的元素移除fieldChanged类名以去掉标记❻。

作为整个过程的最后一个步骤，我们确保onsubmit事件处理器返回false以避免<form>元素继续执行“常规”的表单提交过程。在本示例页面的表单上输入有效数据并点击提交按钮之后，该页面的显示结果大概如图6-11所示。

好了，现在是时候来讨论棘手的复选框控件了！

检测复选框的变化是一个特殊的挑战，因为相对于其他输入元素，复选框相当独特，没被选中的元素不会作为HTTP请求的一部分被提交。

也就是说，不做特殊设计，我们将无法报告复选框的状态已经从选中变到了未选中[①]。因此，我们将巧妙地为这些顽皮的控件采取一种特殊设计！这正是trackCheckboxes()❸函数的目的。像之前所提到的，我们在ready()处理器的末尾调用了该函数。

该函数为跟踪复选框的变化所采取的策略是引入一个隐藏（hidden）字段来代替实际的复选框。该隐藏字段根据复选框的状态保持一个on或off值。这样，当复选框的状态发生改变，包括从选中变成未选中时，可以将对应的隐藏字段的值提交给服务器。

服务器代码显然也需要跟着做一些改动。正常情况下，未选中的复选框元素是不会被作为参数提交给服务器的。而采用了这个策略之后，未选中的复选框元素将传送off值。

为了实现这一策略，trackCheckboxes()函数遍历每一个复选框元素（注意jQuery是如何

① 因为一个复选框的值本身是不会变化的，变化的只是你发送（选中）还是不发送（未选中）。注意，作者的解决方案是采用一个取巧的办法，但是这是以限制复选框的用法为前提的，具体请看下一条译注。——译者注

简化这一过程的），为元素植入onchange事件处理器，创建一个新的隐藏元素并使用复选框的原有名字，而复选框元素本身的名字则被改掉(相当随意的加上一个下划线做前缀)。复选框元素上还添加了hidden属性以便引用对应的隐藏元素。最后，在该复选框上还安置了一个onclick事件处理器，确保可以通过隐藏元素的值来跟踪实际复选框的状态。

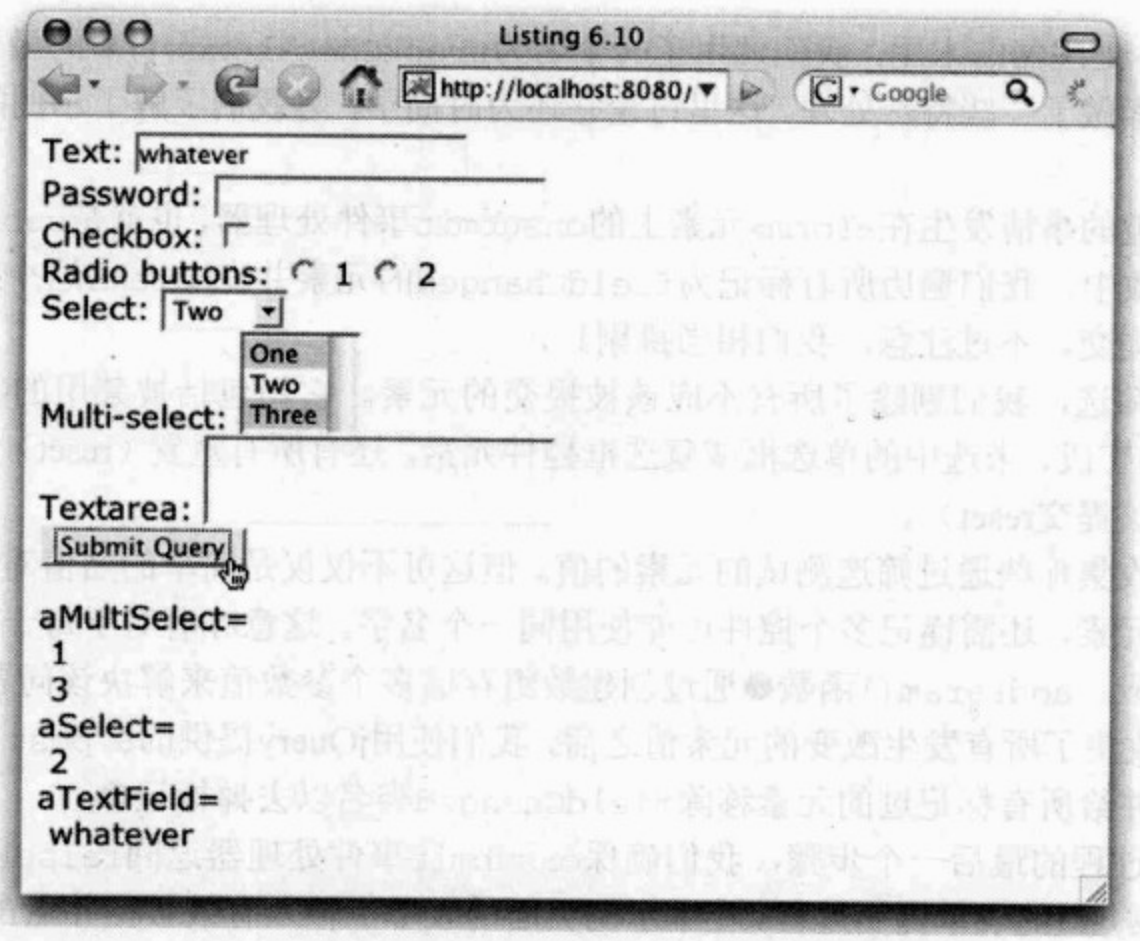


图6-11　仅提交发生改变的元素的值

尽管所有这些看起来可能有点错综复杂，不过它并不像最初看上去那么难以理解，采用这一方式，我们就能可靠地跟踪复选框的变化[①]，就像对其他输入类型所做的一样。

3. 讨论

在本解决方案中，我们深入研究了如何通过Ajax POST请求只把表单中发生改变的元素提交到服务器。这一次我们对细节更加留意，例如避免发送被禁用的控件的值。我们还充分利用jQuery来解决众多凡尘俗事，例如实现事件绑定、添加或移除类名以及搜索DOM元素。

---

① 需要指出的是这种方式有一定的局限性，不适用多个复选框使用同一个名字的情况。原本一个名字可以对应若干个不同的值，现在传递的值不是on就是off，无法区分到底是哪个选项被选中了。在这里的示例中，实际上是把一个复选框当成一个简单的是非选项来使用，这非常符合checkbox的英文字面义。有趣的是，中文译为“复选框”实质上说明了<input type=checkbox>的另一个特质：复选框并非提供一个简单的名值对（一个名字对应一个字符串值），而是一种特殊的名值对，即名字所对应的值是一个字符串集合。单个复选框的变化实际使得这个名字所对应的字符串集合也发生了变化，因而你需要提交改变后的字符串集合，也即所有选中的复选框的值的集合。从逻辑上说，复选框其实和允许多选的<select>类似，所以你可以参考代码中对多选<select>的处理。结论是，最稳妥和简便的方法其实是严格按照表单提交的“常规”方式，每次都遍历所有的复选框，将所有选中的复选框的值发送给服务器。——译者注

6.3 总结

本章介绍了一些Ajax数据处理的相关概念：在投递数据之前进行有效性验证和跨字段有效性验证，以及打包所有数据并通过自定义POST请求发送到服务器。这些技巧可以为你的应用带来实质性的速度提升。即时输入验证将为用户省去烦恼，不必再等待服务器告诉他们数据有误。Ajax表单POST将为用户省下时间，不必再等待服务器重新生成和刷新页面。

记住，自行发送POST请求有一些特别注意事项，比如对数据进行编码，以及为请求主体的内容长度和内容类型设置相应的首部。你应该对代码进行全面细致的调试，因为这很容易出错。我们强烈建议你使用诸如Prototype或jQuery等程序库。事实上，别人已经把复杂问题替你解决了，你只需善加利用。我们的一贯信条就是代码重用。

第7章 内容导航

本章内容

- 内容导航原理
- 桌面应用和Web应用的影响
- Tab控件、窗口控件和树控件
- JavaScript的适度降级

我们可以将大多数网站简单地归结为：服务器上的巨量信息存储库，以及一套信息导航机制。经由导航，用户可获取网站信息的一个子集以便与之交互。在图片和视频共享网站中，需要导航的信息是图片集或其他媒体的集合；在Web邮箱应用中，需要导航的是邮件；在网上商店（e-commerce store）中，商品目录需要导航……无论网站的性质和内容是什么，基本技术问题是相同的：服务器包含大量内容，用户必须能对这些内容加以筛选，以便找到他们想要的内容。便捷的导航是在同类服务中取得竞争优势的重要砝码，也是任何基于Web的应用都必须处理的要素。

关于内容导航和Web的主题，已经有许多书进行了探讨，也已发展出了各种成功策略，并广泛运用在许多Web网站中。在本章中我们会简略地介绍这些策略，不过这些只是本章主题的背景知识。本书讲的是Ajax及其对Web的颠覆性影响，所以我们会来看一看Ajax到底是如何改变了基于Web的内容导航。下面我们首先概览一下内容导航问题，探索一下导航策略的关键要素。

7.1 网站导航原理

前面已经说过，许多Web站点的一个基本问题就是服务器存储着大量信息。网站越成功，问题就越大。比如你的网上商店可能拥有数十万种商品。用户无法直接处理如此大量的信息，他们需要查看数据的不同子集。从技术层面上讲，这向我们提出了两个问题：第一，需要一种机制能生成符合每个用户需求的数据子集；第二，需要一种方式能让用户通过与网站的交互表达出他们的需求，而且要尽量简单。我们依次讨论这两个问题。

7.1.1 大海捞“针”

第一个问题是帮助用户找到正确的数据。这主要是一个后端任务，而本书是一本主要讲前端技术的书，但为了理解网站导航原理，仍然值得在此问题上花点功夫。概括而言，有两种不同的

信息组织方式："方以类聚"（classification）和"物以群分"（categorization）。"聚"和"分"这两种方式最好通过示例来解释，所以我们选了几个众所周知的例子。

Yahoo!对世界上最超级庞杂的数据集——也就是整个公众互联网——采用"分"（categorization）的方式。信息按照一个层级分类进行组织，每一份信息都恰好属于一个"分"类（category）：购物、运动、财经、音乐，或者其他某个分类。简单的"分"法可以只包含一级分类，但Yahoo!采用了一套层级分类方案。也就是说，每个类还可进一步分成子类，这样就形成树状结构，只需从树的顶端向下展开几层，就可以分门别类地管理大量信息。图7-1演示了"物以群分"的原理。

"聚"和"分"不同，一个数据项只能属于一个"分"类，但可以归属多个"聚"类。如果我们把Yahoo!看成是对互联网的一种"分"法，那么Google就是一种"聚"（classification）法。每个搜索词就是一个"聚"类，而若干互相无关的搜索词也可能会返回同一个Web页面。对于Google来说，当它通过爬网软件（spidering software）[①]和网页排序（page-ranking）算法对文档进行索引时，会自动生成"聚"类。另一些情况下，可能由网站管理员手工生成"聚"类。最近，允许访问者对页面内容进行"聚"类——通常被称为打标签（tagging）——的做法也流行了起来，它提供了一种强大并可伸缩（scalable）[②]的网站内容组织机制。图7-2 描述了"方以类聚"的原理。

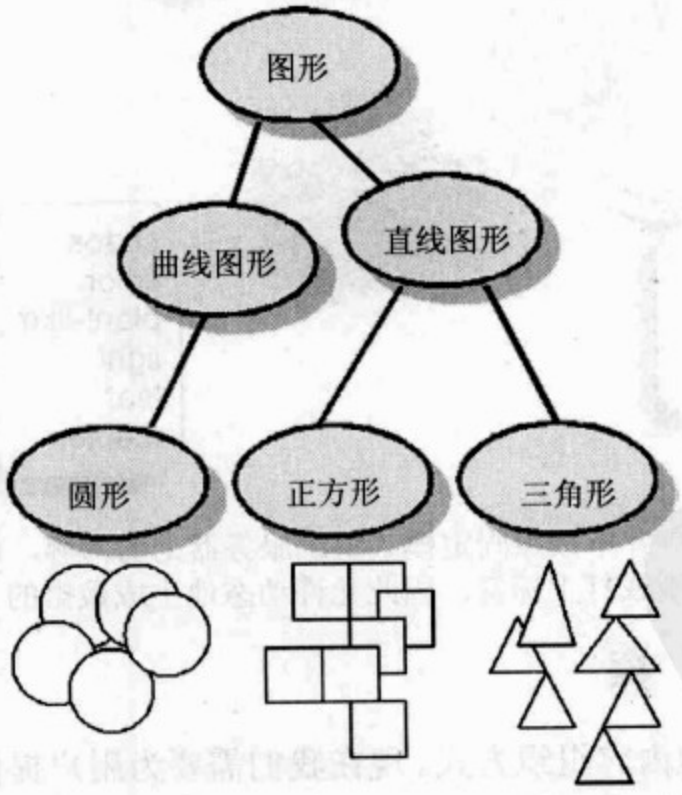


图7-1 "物以群分"即每个信息元素在任何层级上都可确切地划分到一个类别

"分"和"聚"各有所长。"分"的方式没有"聚"的方式灵活，对超大型的数据集进行"分"类最终会陷入到三种困境之一：要么每个层级上的分类太多，要么子类层级太深，要么层级树的每个叶子节点中的信息项太多。尽管如此，"分"的方式为用户提供了一种可靠的、相对更明确

① 即web spider，也称web crawler或web robot，指按照一定规律自动浏览Web的程序或脚本。搜索引擎通过爬网软件来获得最新的数据以便进行索引。——译者注

② "可伸缩"在这里指可以适用于不同规模的网站。——译者注

的途径，用户可以有迹可循按部就班地找到他们想要的内容。在线商店如果只是简单地给用户呈上一个搜索框，而没有线索来指示食品、衣服、电脑配件等是否有售，将可能失去许多客户。在实践中，大多数网站会结合使用这两种技术。以Yahoo!为例，它在每个分类下也提供了搜索功能，而Google其实也有几种顶层分类，如网页、图片和视频搜索。

我们已经研究了信息组织的原理，下面让我们把注意力转向用户，看看怎样为他们提供一种方式来驾驭我们或“分”或“聚”的体系。

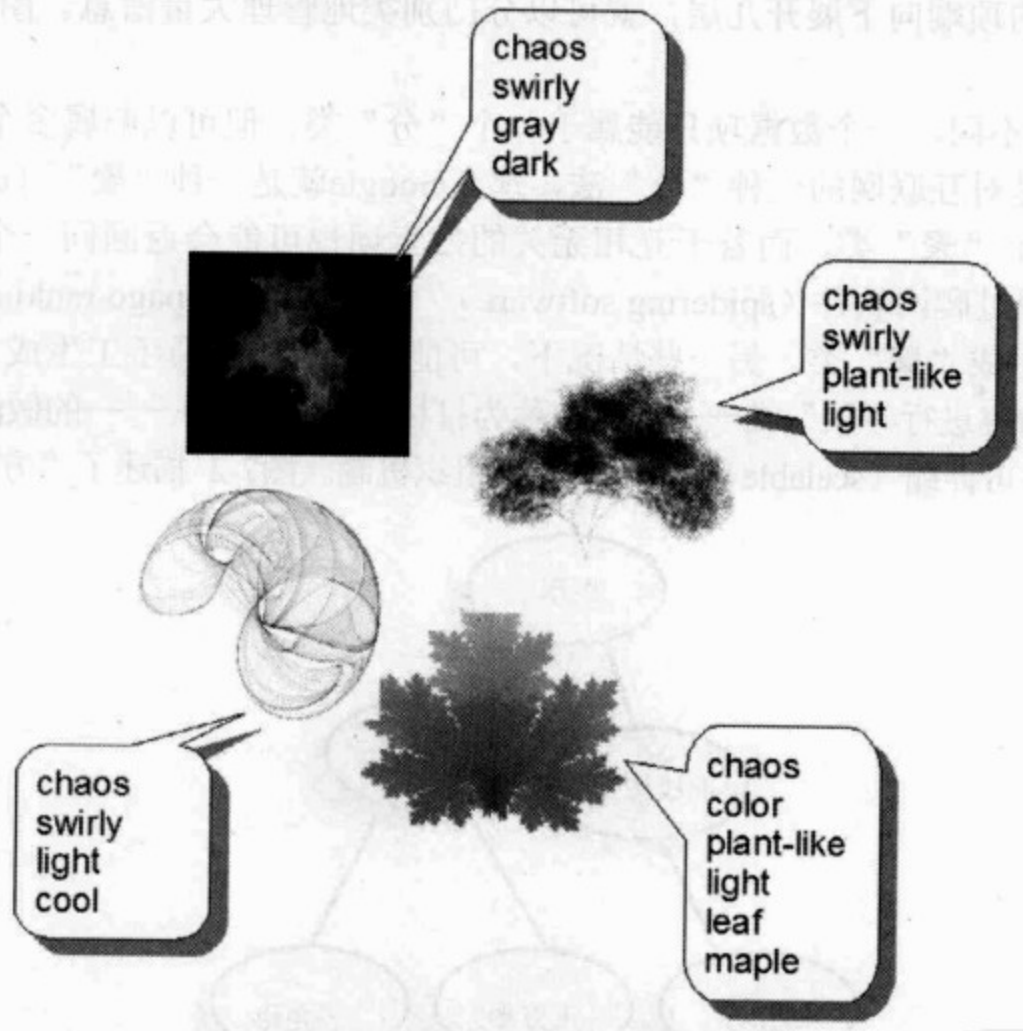


图7-2　“方以类聚”并不按照固定模式组织服务器上的内容，而是将每个元素编入索引或打上标签，因此允许动态地生成数据的子集

7.1.2　创造更好的“针”探

我们已建立了网站背后的内容组织方式。现在我们需要为用户提供一种方式来使用我们经过内容组织的系统并找到他们想要的数据。

“分”类系统在Web上有许多用户界面机制。流行的方案包括侧边栏（sidebar）、导航栏（navigation bar）和当前路径（breadcrumb trail）[①]。Yahoo!在网站顶层使用了一个简单的侧边栏（见图7-3）。

如果从Web转向桌面应用，我们还可以看到许多控件正是被设计用来处理分类的，比如菜单

① 当前路径导航被形象地称为breadcrumb trail即面包屑踪迹，典故出自格林童话中的《糖果屋》（Hänsel und Gretel）：一对兄妹被继母带入森林遗弃，沿途丢下面包屑以便循迹返回。——译者注

（menu）、下拉列表（drop-down list）、Tab控件和树（tree）控件。其中一些已经被Web网站采纳，最显著的就是层级菜单（hierarchical menu）。事实上，早在Ajax出现之前，动态HTML（DHTML）的最成功和广泛的用途之一就是交互式的下拉菜单条，它能在有限的空间内列出许多导航选项。图7-4显示了一个在线商店页面，它使用了众多从Web和桌面借鉴而来的导航方式。

图7-3 Yahoo网站采用了一个简单的侧边栏展现顶级分类

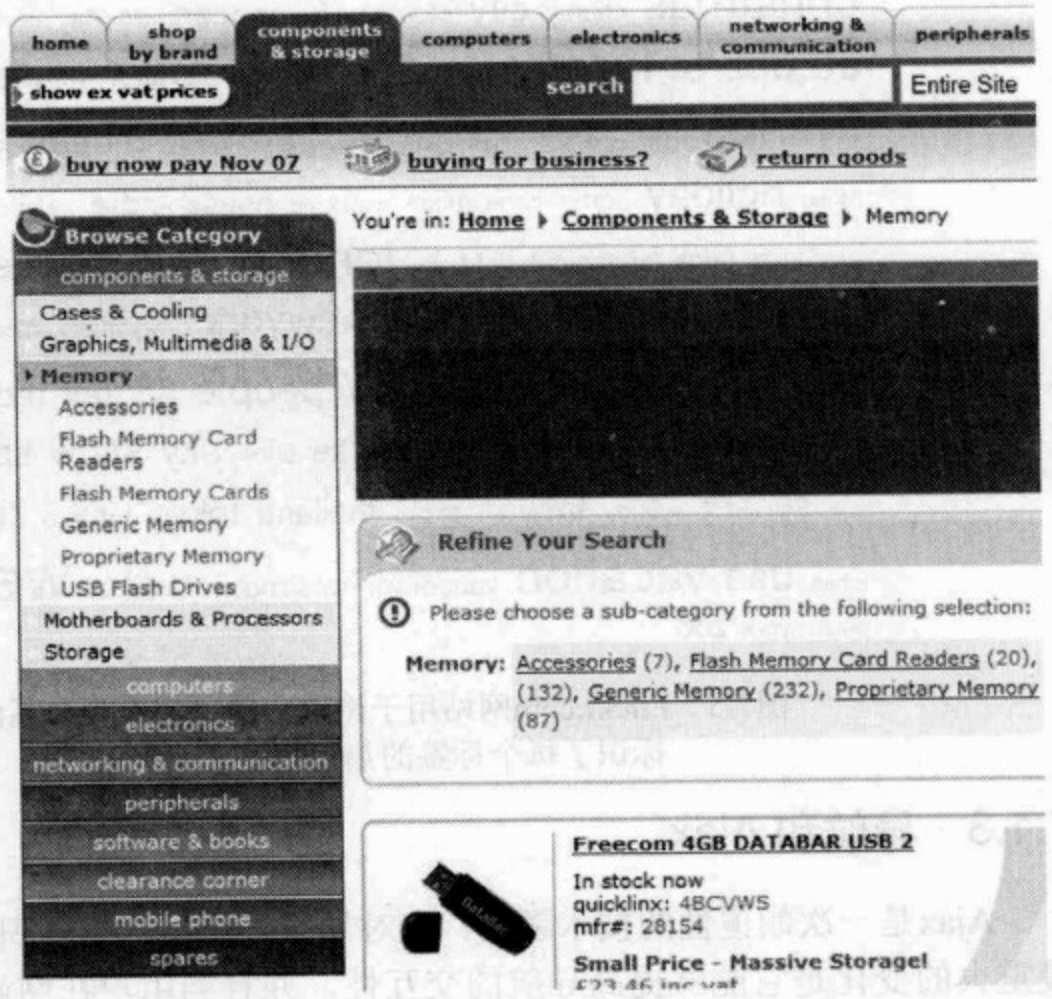


图7-4 为了帮助用户在复杂的分类体系中导航，在线商店dabs.com运用了多种UI控件，包括一个树控件（左侧）、一组Tab（顶部）和一个当前路径导航条（Tab的下方）

另一方面，如果转而关注“聚”类方式（classification），我们会发现，到目前为止最通用的内容导航用户界面正是不怎么起眼的搜索框，Google极简主义的主页就是最好写照。也可以给搜索加上一些限定字段，比如日期范围，或者与分类方式配合使用，例如Google的搜索大类。标签（tagging）的出现也带来了新一代的UI模式，例如Flickr所采用的热门标签机制（见图7-5）。

所有这些导航体系都能用传统的超链接和HTML表单来实现，也许还可以采用一点儿动态HTML来锦上添花。正如本章开始时所说的，到目前为止的讨论只是为本章主题——Ajax是如何改变Web导航领域的——设定一个背景。下一节，我们就会看到Ajax提供了哪些新的机制来帮助用户进行网站内容的导航。

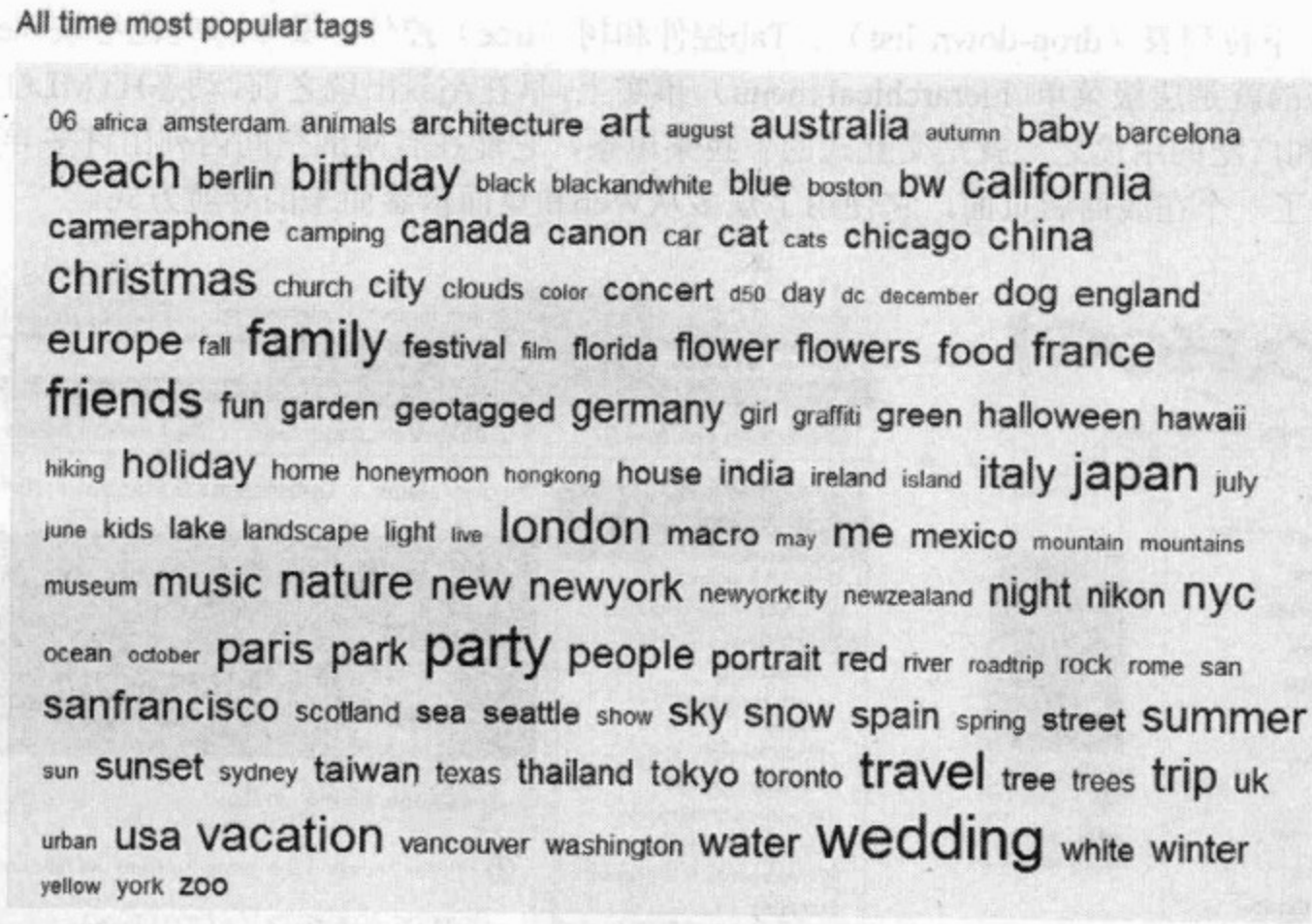


图7-5 Flickr.com网站用于浏览它的“聚”类体系的新式UI，字体大小标识了每个标签的热门程度

7.1.3 导航和 Ajax

Ajax是一次颠覆性的技术融合，它对导航领域也产生了积极的影响。毫无疑问，Ajax带来的最重大的变化是它能够提高导航的交互性，并且当用户在网站中浏览时能向用户提供更好的反馈。Ajax出现之前，无论导航控件是由静态HTML还是DHTML构成，只有改变当前窗口的URL或者窗格（frame或iframe）的URL才能真正跳转到一个新的位置。而有了Ajax，我们可以从服务器异步请求信息并以一种增量方式更新页面。对于简单的应用，Ajax提供了一种更加高效的更新屏幕指定区域的方式。在更复杂的案例中，我们可以对若干界面元素进行局部更新，创造一种非常类似桌面应用的用户体验。

如果把导航看成一个两步骤处理过程，第一步是用户与导航控件的交互，第二步是获取新内容或新数据，就能明白，Ajax改变了第二步的处理，而DHTML则无法涉及这一步骤的处理过程。DHTML可以改善控件的交互性，达到类似桌面应用的外观效果，但内容获取方式还是局限于在屏幕上呈现全新内容，因此还是像一个网站而不是一个应用。

目前来说，Ajax应用中的导航设计就是要对网站用户体验和桌面应用用户体验进行某种平衡。在体验的一端，可以创建视觉效果与桌面应用类似的网页，并且完全采用桌面风格的GUI惯例（GUI convention）。在体验的另一端，可以利用Ajax来更新页面的局部内容，并更多使用Web应用的风格。在这两端之间则是综合了Web风格导航和桌面风格导航所可能创造出来的各种新类型的应用。

我们将在本章的后续部分探索这些有趣的可能性。下面让我们首先看一下传统的基于Web的导航方法。

7.2 传统的 Web 导航

我们已经说过，一个Ajax应用可以同时引入传统Web应用和桌面应用两者的惯例。如果一开始是一个传统的Web站点，最简单的转变方式就是保持基于Web的导航惯例，但把超链接和表单替换成对服务器的异步请求。当我们处理一个传统Web应用，更新部分功能来逐步引入Ajax时，这也是一个不错的起点。

下面我们先来看一个导航菜单的例子，看一下启用Ajax支持涉及哪些内容。

7.2.1 一个简单的导航菜单

让我们看看在Ajax风格的应用中一个简单的导航菜单是什么样子的。就像我们之前所提到的，不能再用传统的超链接了，我们需要更先进一点儿。

1. 问题

你正在把一个标准网站移植为一个单页面Web应用。你的导航方案需要改为通过Ajax调用来链接到内容。

2. 解决方案

对于本例（效果如图7-6所示），我们使用了一个简单的纵向列表菜单。当菜单项被选中时，应用会从服务器获取所需内容并动态加载至页面的内容显示区域。我们的第一步是创建函数`invokeLink(url)`，它从服务器请求图片内容并插入到DOM中；然后，需要把每个菜单链接改成调用`invokeLink()`函数，就这么两步即可。请看该示例的（代码清单7-1）。

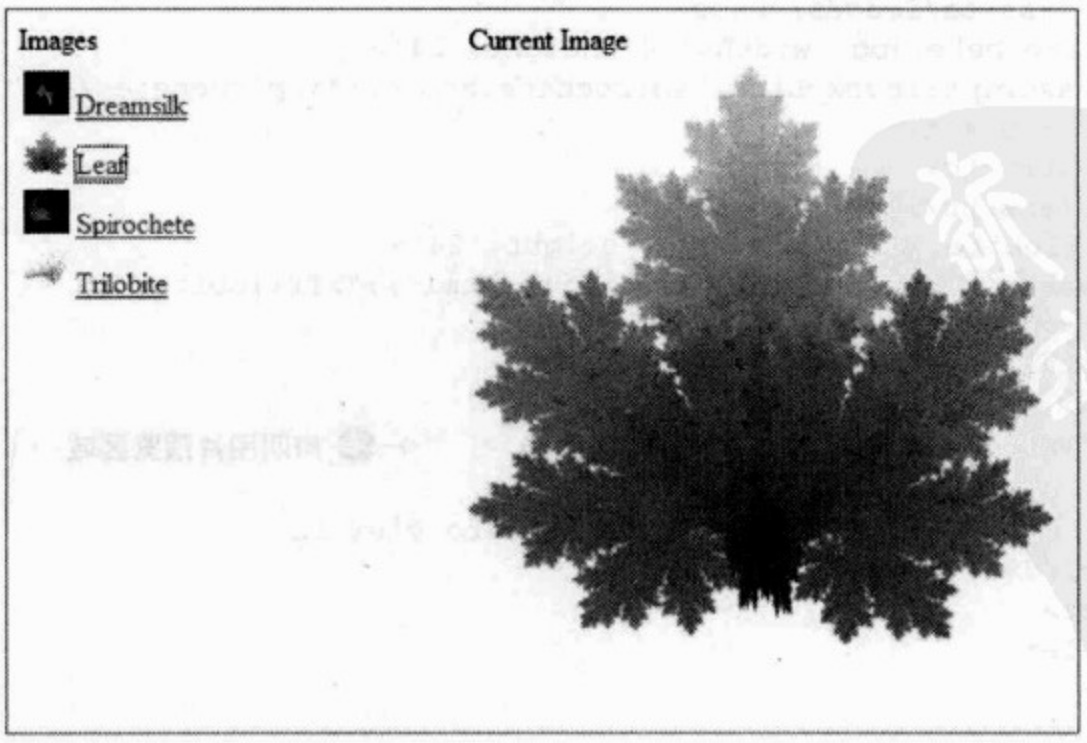


图7-6 一个简单的导航菜单

代码清单7-1　我们的导航菜单

```
<html>
  <head>
    <title>Chaotic Images</title>
      <script type='text/javascript'
        src='../assets/js/jQuery.js'></script>      ❶ 导入jQuery程序库
      <script type='text/javascript'>
function invokeLink(url){
  $('#content_area').load(url);      ❷ 发起Ajax调用
}
    </script>
  </head>
  <body>
    <table width='100%'>
      <tr>
        <td width='25%'>Images</td>
        <td width='75%'>Current Image</td>
      </tr>
      <tr>
        <td valign='TOP'>      ❸ 声明默认菜单选项
          <table>
            <tr><td>
<img src='../assets/images/chaos/
  ➥ ifs/dreamsilk.jpg' width='24' height='24'>
<a href="javascript:invokeLink('dreamsilk.html');">Dreamsilk</a>
            </td></tr>
            <tr><td>
<img src='../assets/images/chaos/ifs/leaf.jpg' width='24' height='24'>
<a href="javascript:invokeLink('leaf.html');">Leaf</a>
            </td></tr>
            <tr><td>
<img src='../assets/images/chaos/
  ➥ ifs/spirochete.jpg' width='24' height='24'>
<a href="javascript:invokeLink('spirochete.html');">Spirochete</a>
            </td></tr>
            <tr><td>
<img src='../assets/images/chaos/
  ➥ ifs/trilobite.jpg' width='24' height='24'>
<a href="javascript:invokeLink('trilobite.html');">Trilobite</a>
            </td></tr>
          </table>
        </td>
        <td valign='TOP' id='content_area'>      ❹ 声明图片预览区域
          <br/><br/>
          Click on an image to the left to view it.
        </td>
      </tr>
    </table>
  </body>
</html>
```

3. 讨论

我们不想浪费时间手工编码Ajax请求，所以使用了程序库来处理底层细节。在本例中，我们选用了jQuery，在添加我们自己的脚本之前把该库加载到浏览器❶。有了jQuery的帮助，`invokeLink()`方法简单得不值一提❷。jQuery的`$()`函数与Prototype库类似，也是用来选择DOM元素的。在jQuery中，`$()`接受一个CSS选择器作为参数。我们根据在HTML中所定义的元素id❹选定目标节点。jQuery为经`$()`函数所得到的DOM节点添加了`load()`方法，该方法会帮我们创建一个Ajax请求并将响应内容填入节点。载入内容区域的HTML文件包含一些HTML代码片段，即指向正确的图片的`<img>`标签。这些HTML代码片段会替换掉页面的内容区域，浏览器会自动显示出图片。这些HTML代码片段并不需要对应到实际的服务器文件，而是可以由服务器生成，这样就可以实现更加动态的用户界面。

余下的工作就是把`invokeLink()`函数挂接到我们的UI。本例中，我们在定义菜单的HTML中使用了超链接，也就是锚点（anchor）标签，但是URL是JavaScript协议而非HTTP ❸。这种方法可以奏效，不过有些太过原始，我们不得不在文本中定义回调方法，而不像真正的代码可以直接引用程序中其他地方定义的变量。我们将在下一个示例中看看如何利用编程方式将事件挂接到UI。

现在我们已经了解了使用Ajax请求从服务器直接请求内容是多么简单。这仅仅是一个非常初级的、最简单的示例。让我们继续前进，研究一个复杂一点儿的，也是当前Web应用中被大量使用的技术：下拉式菜单。

7.2.2 DHTML 菜单

下拉式菜单是DHTML技术中为数不多的成功案例之一，如果在Google上搜索一下，你会发现在互联网上有各种各样的下拉菜单可供所谓的Web设计者选用和定制。DHTML菜单通常使用JavaScript，有时也用CSS来控制菜单控件的交互，然后菜单项相关联的操作则由一个简单的超链接负责。

本节，我们将会选一个比较不错的DHTML菜单作为示例，并使用上一节的`invokeLink()`函数来改进它，创建出一个简单的、交互式的、支持Ajax的菜单。

1. 问题

你正在改进网站的Ajax特性并且碰到一个DHTML菜单需要启用Ajax支持。另一方面，你可能有了一个新需求，就是要在分类中保存Ajax操作。不管如何，你将发现这个控件非常有用！

2. 解决方案

用作我们示例起点的菜单控件是基于Nick Rigby在“横向展开式的下拉菜单（Drop-down Menus, Horizontal Style）”一文（http://alistapart.com/articles/horizdropdowns）中所描述的技巧。Nick描述了如何创建一个层级式DHTML菜单，其布局和交互主要是使用CSS完成而不是用JavaScript，只是为了弥补在Internet Explorer浏览器中CSS实现的某些不足才加入了少量的JavaScript代码。

在上一个示例中，我们用到的混沌图片都属于迭代函数系统（iterated function system，IFS）[①]的范畴。在本例中，我们添加了两个图片，它们属于另一种类型：Lindenmeyer系统，或称作L系统（L-system）[②]。我们使用Nick所描述的DHTML菜单的实现方法，把我们的链接组织为DHTML式的层级下拉菜单。我们还利用jQuery将菜单的JavaScript代码改写得更加简洁优雅。图7-7显示了该菜单的实际效果，所需的则列于代码清单7-2中。

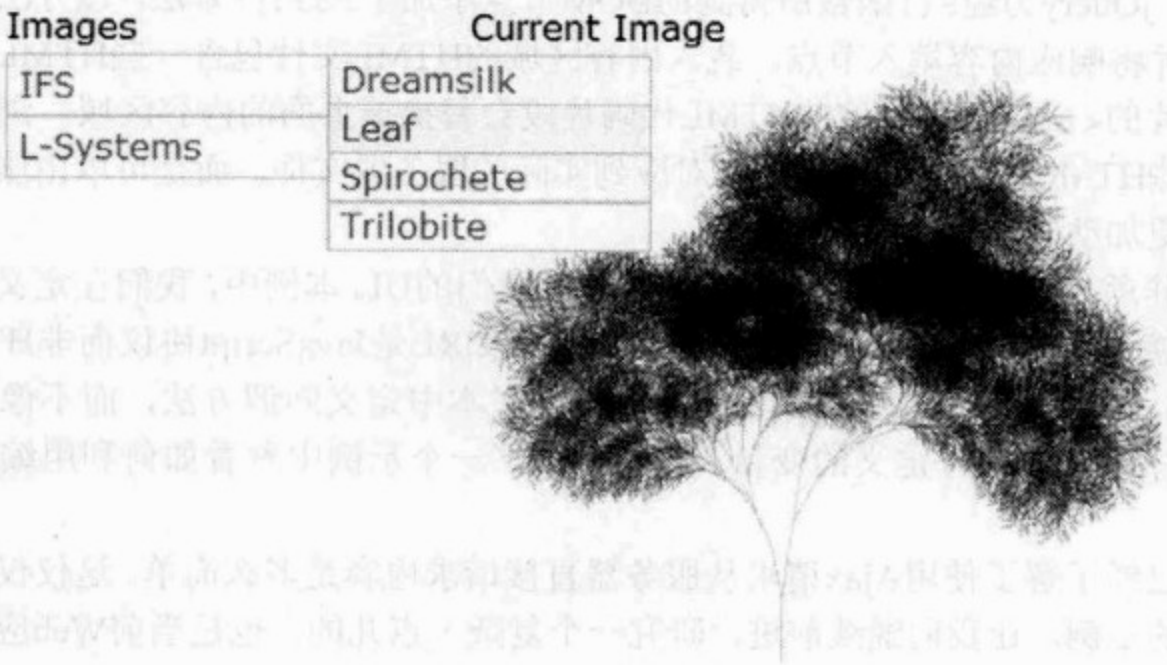


图7-7　经过Ajax增强后的简易DHTML菜单

代码清单7-2　DHTML导航菜单

```
<html>
<head>
<title>Chaotic Images</title>
<link rel="stylesheet" type="text/css"          ❶ 导入CSS样式表
  href="menu.css">
<script type='text/javascript'
src='../assets/js/jQuery.js'></script>          ❷ 导入jQuery程序库
<script type='text/javascript'>
  function startList() {
    if ($.browser.msie) {                       ❸ 进行浏览器检测
      $("#nav > li").each(
        function(index, node){                  ❹ 通过编程方式加上hover时的样式
          $(node).hover(
            function() {
              $(this).addClass("over");
            },
            function() {
              $(this).removeClass("over");
            }
```

① IFS可以用来产生分形（fractal），图案构造都是自相似的。——译者注

② L系统常用于植物生长的建模，也能用来产生分形。——译者注

```
            );
          }
        );
      }
    }

    function invokeLink(url){
      $('#content_area').load(url);
    }

    $(startList);
  </script>
  </head>
  <body>

    <table width='100%'>
      <tr>
        <td width='25%'>Images</td>
        <td width='75%'>Current Image</td>
      </tr>
      <tr>
        <td valign='TOP'>
          <ul id="nav">
            <li><a href="#">IFS</a>
              <ul>
                <li><a
href="javascript:invokeLink('dreamsilk.html');">Dreamsilk</a></li>
                <li><a href="javascript:
                  ➥ invokeLink('leaf.html');">Leaf</a></li>
                <li><a href="javascript:
                  ➥ invokeLink('spirochete.html');">Spirochete</a></li>
                <li><a href="javascript:
                  ➥ invokeLink('trilobite.html');">Trilobite</a></li>
              </ul>
            </li>
            <li><a href="#">L-Systems</a>
              <ul>
                <li><a href="javascript:invokeLink('bush.html');">Bush
                    </a></li>
                <li><a href="javascript:invokeLink('weed.html');">Weed
                    </a></li>
              </ul>
            </li>

          </ul>
        </td>
        <td valign='TOP' id='content_area'>
          <br/><br/>
          Click on an image to the left to view it.
        </td>
      </tr>
    </table>
  </body>
  </html>
```

❺ 使用Ajax读取内容

❻ 声明菜单目录

要运行该示例，我们需要导入CSS样式表❶以及jQuery库❷。在Firefox浏览器中，只需CSS菜单就能运转，但对于IE浏览器，我们需要添加一些JavaScript代码。这里我们没有列出CSS样式表，你可从www.manning.com/crane2下载本章的源代码，示例中包括该CSS样式表。因为我们使用的就是Nick Rigby的原始代码，并没有做什么改动，所以这里就不再赘述。我们并不关心DHTML如何工作，关注的是如何增加Ajax功能。

不过在考虑添加Ajax之前，让我们先看看jQuery能为我们做点什么。在startList()函数中，我们用到了好几个值得注意的jQuery特性。首先，比起手工检测user-agent字符串，jQuery提供的浏览器检测机制当然更加清晰❸。

其次，我们使用了$()并传入CSS选择器作为参数。这次选择器将匹配多个DOM元素，$()将返回一个数组的包装对象。我们可以使用each()方法遍历该数组。each()方法接受一个迭代函数作为参数（该函数将应用到数组的每个元素上）。我们通过迭代函数添加mouseover和mouseout事件处理器以改变这些元素的样式❹。jQuery为DOM元素附加的hover()方法使这一目标变得异常简单①。我们也使用了addClass()和removeClass()方法，它们能更细致地控制元素的CSS类。

最后，我们再次调用$()，此次参数是startList()函数。当以函数为参数调用$()时，jQuery会将该函数绑定为window.onload的事件侦听器，当页面的DOM树加载完成时就会调用该方法。我们可以直接写成：

```
window.onload=startList;
```

但是使用$()方法的优势在于允许我们添加多个侦听器。对于本例这种规模的示例来说，这不算什么，但对于大型的模块化的系统来说就非常有用。

菜单呼之欲出了。为了增加Ajax支持，我们可以再次直接使用由jQuery驱动的invokeLink()函数❺并给菜单节点加上超链接。我们将菜单声明为一组HTML无序列表元素❻，CSS或JavaScript代码会改变它们的外观。好了，当页面加载后，我们的菜单就开始工作了。

本小节中我们看过了Web风格的导航。下一节，我们将探索导航体验的另一端：桌面应用的用户界面之路。

7.3 借鉴桌面应用的导航设施

早在Web发展初期，桌面应用开发人员就有了处理内容导航的丰富经验并开发了许多用于组织可视化内容的组件。在前一章介绍过的DHTML菜单的视觉效果从桌面应用中借鉴了很多内容，但总体而言，桌面应用和Web应用有部分内容直接重叠。原因之一是传统Web应用曾经依赖整个页面刷新模型，而桌面程序的用户界面在基于增量更新的基础上演化。与诸如树、表格和工具栏等复杂界面元素交互时，每次交互一般只改变整个程序屏幕的部分内容。

Ajax的出现改变了这种状况。通过和服务器通信实现内容增量更新，Ajax创建了和桌面应用程序更加吻合的导航技术，并使得从视觉效果上和桌面应用相近的Web应用开始浮现。

---

① 对于IE浏览器，jQuery的hover方法实际上会使用IE专有的onmouseenter事件和onmouseleave事件，而不是mouseover事件和mouseout事件。——译者注

我们在本章开始时指出，Ajax开发的一个极端是用户界面酷似桌面应用，但这个结论显然无法适用所有基于Web的应用。在桌面应用和传统Web模型之间，有很多有趣的导航实现可能存在，而我们在本章稍后将回到妥协方案。不过现在，让我们考虑桌面应用世界有什么可在导航技术方面加以借鉴。

7.3.1 使用 qooxdoo 库实现 Tab 组件

在前一个示例中，我们使用CSS和少量JavaScript为已声明的页面主体(body)部分的一段HTML代码添加了一些功能。qooxdoo组件库从根本上采取了不同的实现途径来表现Web用户界面，并且它通过提供面向对象的JavaScript API创建、维护用户界面组件，从而帮助你避免Web用户界面开发过程中的缺陷。如果你曾经用过支持GTK、AWT、Swing或其他瘦客户端开发的面向对象GUI工具，qooxdoo库将使你感到很亲切。

1. 问题

你的应用包含由若干零散内容片段组成的几组信息，但你希望在一个时刻仅显示一组数据。这些内容片段可以是很多内容：导航组件、表单等。

2. 解决方案

关于qooxdoo库，首先要了解的是程序初始化在`window.application.main()`方法中发生。可以提供一个匿名实现方法，如代码清单7-3中所示，结果如图7-8所示。

在创建Tab的过程中，首先需要创建一个QxTabView实例，然后设置left、tab、width和height。之后，我们可以创建四个QxTabViewButton（用于切换Tab）实例，再把它们添加到Tab实例边框容器（bar container），然后将第一个Tab设置为选中状态。接下来，我们需要创建实际的QxTabViewPage示例并将它们添加到Tab组件实例的面板(pane)。然后为我们的图片创建QxImage实例并把它们添加到相应的页面。最后，我们把标签组件示例添加到客户端文档即qooxdoo库的顶级容器。qooxdoo库的所有一切都存在于客户端文档中。

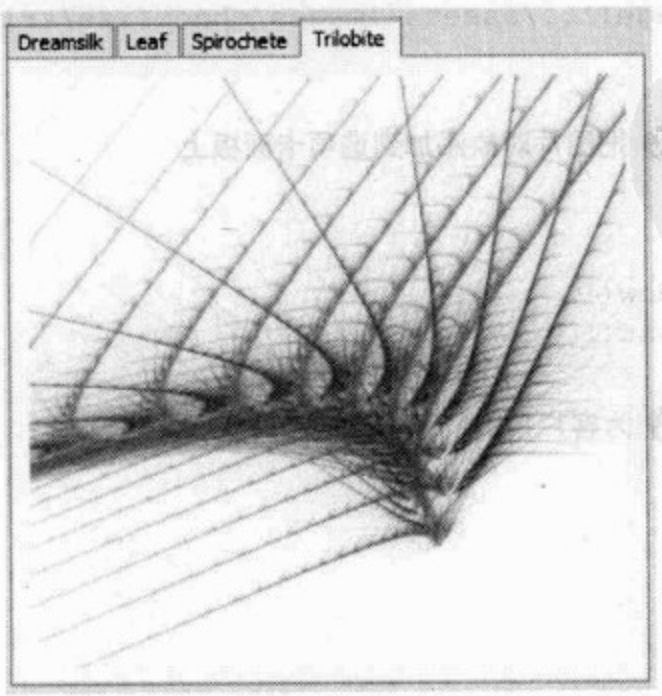


图7-8 qooxdoo库tab组件

代码清单7-3　使用qooxdooTab组件

```
<html>
  <head>
    <title>Chaotic Images</title>
    <script src='../assets/js/qooxdoo/include.js'
            type='text/javascript'></script>

    <script LANGUAGE='JavaScript'>
      <!--
  window.application.main = function()
  {
    var tf1 = new QxTabView();
    tf1.set({ left: 20, top: 48, width: 342, height: 362 });

    var t1 = new QxTabViewButton('Dreamsilk');
    var t2 = new QxTabViewButton('Leaf');
    var t3 = new QxTabViewButton('Spirochete');
    var t4 = new QxTabViewButton('Trilobite');

    t1.setChecked(true);

    tf1.getBar().add(t1, t2, t3, t4);

    var p1 = new QxTabViewPage(t1);
    var p2 = new QxTabViewPage(t2);
    var p3 = new QxTabViewPage(t3);
    var p4 = new QxTabViewPage(t4);

    tf1.getPane().add(p1, p2, p3, p4);

    var i1 = new QxImage('../assets/images/
      chaos/ifs/dreamsilk.jpg');
    var i2 = new QxImage('../assets/images/chaos/ifs/leaf.jpg');
    var i3 = new QxImage('../assets/images/chaos/ifs/spirochete.jpg');
    var i4 = new QxImage('../assets/images/chaos/ifs/trilobite.jpg');

    p1.add(i1);
    p2.add(i2);
    p3.add(i3);
    p4.add(i4);

    this.getClientWindow()
      .getClientDocument()
      .add(tf1);
  };
      //-->
    </script>
  </head>
  <body>

  </body>
</HTML>
```

❶ 声明新的选项卡

❷ 创建选项卡按钮

❸ 为选项卡栏添加按钮

❹ 创建选项卡面板

❺ 把选项卡面板添加到选项卡对象的视图面板

❻ 创建图片对象

❼ 把图片对象添加到选项卡面板上

❽ 为客户端文档添加选项卡对象

3.讨论

在代码清单7-3中，我们首先声明了一个新的Tab组件对象❶。此后，我们新建了4个选项视图按钮对象，并把第一个设置为选中❷。我们将这些新创建的选项视图按钮添加到Tab视图的边栏❸。我们还需要新建4个Tab视图页面对象❹，它们将拥有该Tab的实际内容，并把它们添加到Tab视图面板❺。我们还没有提供内容。为实现上述目的，新建4个图片对象❻并把它们添加到相应的页面❼。最后，为了显示Tab视图对象，我们把它添加到客户端文档❽。

我们第一个介绍qooxdoo库的API只是让你感觉一下在qooxdoo中设置组件多么简单以及它与编写传统Web应用有多大的区别。非常少的JavaScrip代码、无HTML（注意body标签内部完全空白），对我们大有裨益。使用qooxdoo库的一大主要诟病如同其巨大优势一样一直有争论，这就是：它完全放弃了Web页面的设计/布局模型。美工设计人员不需要进入所见即所得（WYSIWYG）的编辑器并设计用户界面布局。

Tab面板让我们体验qooxdoo的功能，但我们强调以更加传统的开始方式解决问题，因为看了本章稍后介绍的Rico库的Accordion组件后大家就会明白。在下一示例中，我们将对特征更为明显的桌面应用开发模式加强关注，来看一下我们能把这种开发类型推动到什么程度。

7.3.2 qooxdoo 工具栏和窗口

现在，我们再次使用qooxdoo库并研究它提供的窗口和工具栏功能。通过把屏幕区域划分成窗口，它提供了灵活可选择的Tab面板组件，并且如果用户愿意，用户可以浏览几个并排的资源。

1. 问题

为用户提供更丰富的用户界面控制，例如提供多文档界面怎样对页面的不同区域进行空间分配。

2. 解决方案

我们可以通过为用户提供工具栏，使他们像使用Window控件一样加载所有资源来满足这种需求。使用qooxdoo库，工具栏和窗口控件已经是现成的，而我们只需重组它们即可。图7-9显示了最终结果。

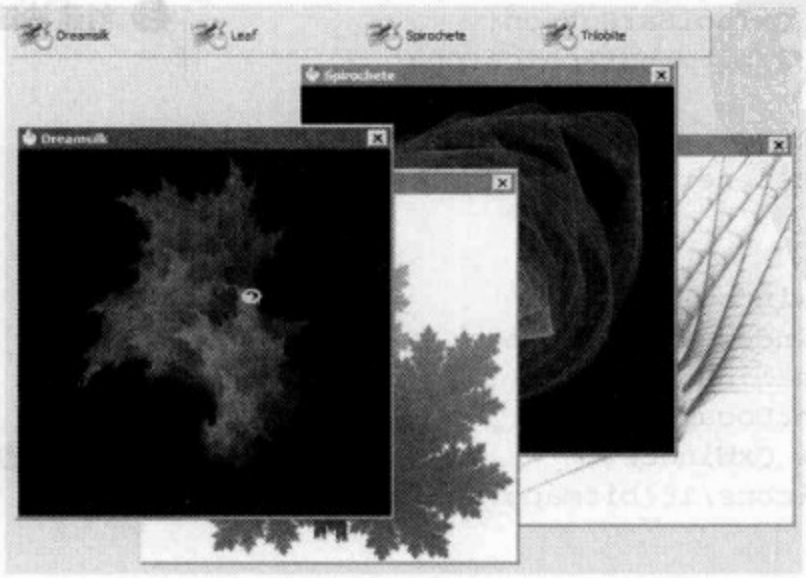


图7-9 qooxdoo窗口和工具栏

实现该目标不是特别难，因为qooxdoo已经为我们实现了大部分繁重的工作。代码清单7-4显

示了所需代码。

代码清单7-4　qooxdoo工具栏和窗口示例

```
<html>
  <head>
    <title>Chaotic Images</title>
    <script src='../assets/js/qooxdoo/include.js' type='text/javascript'></
   script>

    <script type='text/javascript'>
      <!--
window.application.main = function(){

  var d  = this.getClientWindow()
    .getClientDocument();
  var tb = new QxToolBar();      ❶ 创建工具栏对象
  tb.set(
   {top : 20, left : 20, width : 602}
  );

  createLaunchButton(tb, 'Dreamsilk',
    '../assets/images/chaos/ifs/dreamsilk.jpg');
  createLaunchButton(tb, 'Leaf',
    '../assets/images/chaos/ifs/leaf.jpg');
  createLaunchButton(tb, 'Spirochete' ,
    '../assets/images/chaos/ifs/spirochete.jpg');
  createLaunchButton(tb, 'Trilobite',
    '../assets/images/chaos/ifs/trilobite.jpg');

  d.add(tb);
};

var windowCount=0;

function createLaunchButton(toolbar,title,image){
  var button = new QxToolBarButton(
    title, 'icons/32/bitmapgraphics.png'
  );
  button.setWidth(150);
  button.addEventListener(    ❹ 添加事件处理器
    'execute',
    function(){
      if (!button.window){
        var d  = window.application
          .getClientWindow()
          .getClientDocument();
        var win=new QxWindow(
          title, 'icons/16/bitmapgraphics.png'
        );

        win.setSpace(
          20+(48*(windowCount+1)), 320,
```

❷ 为工具栏添加按钮

❸ 创建按钮对象

❺ 据要求创建window对象

```
            20+(48*(windowCount+1)), 320
          );
          win.set(
            {showMinimize : false,
             showMaximize : false,
             resizeable : false}
          );
          win.add(new QxImage(image));

          d.add(win);

          button.window=win;
          windowCount++;
        }
        if (button.window.isSeeable()){
          button.window.close();
        }else{
          button.window.open();
        }
      }
    );

    toolbar.add(button);
}
```

❻ 隐藏或显示窗口对象

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

像以前一样，我们的HTML页面在body标签内不包含任何HTML标记，因为qooxdoo库将为我们产生DOM元素。我们像往常一样在window.application.main()方法中创建、装配组件。

要创建工具栏，我们需要调用工具栏对应的构造函数❶，然后把各个按钮添加到其内部❷。添加这些按钮需要一些额外的操作步骤，因此我们将它们提出放到一个叫做createLanuch-Button()的辅助函数中。

在该函数内部，我们创建qooxdoo按钮对象❸，然后在按钮上添加事件处理器❹。当按钮被点击时，希望它能够在显示或隐藏窗口之间切换❻，但首先要确保该窗口已经创建。可以通过为每个按钮指派一个新属性window，该属性要么为null，要么是一个qooxdoo支持的window对象。因此，在隐藏或显示窗口之前，我们首先检查该属性是否设置，如果没有，我们就按照需求创建一个qooxdoo支持的window对象❺。这部分代码只在第一次点击按钮时执行。

注意，我们定义该事件处理器时采用了内嵌的匿名方法，允许我们在按钮对象上创建闭包。

3. 讨论

在领略浏览器窗口实现多文档用户界面的强大功能之前，本例是个简单的入门。如果你是纯粹的Web开发人员并且从未接触过面向对象的传统的瘦客户端的图形用户界面设计，qooxdoo或许看似愚蠢和不合常规。不过，采用这种方式抽象了HTML技术应用的全部缺点并给你提供了整

洁、简明的面向对象的API，对Web开发人员来说这是一种相当强大的工具。

当采用这种用户界面的应用时，还需要考虑你的客户会有何反应。多文档用户界面为用户提供了更多的页面布局控制，但也向他们提出了更高的要求。用户没必要去更多地控制用户界面的所有方面，用户的需求是什么，你的判断此时很关键。在第1章中，我们讨论过连续性应用与那些设计成偶尔使用的应用程序的区别。我们建议目前这种富客户端的用户界面适用于连续性应用，用户愿意投入时间和努力来设置页面布局，但不适用于偶尔使用的应用例如购物车或在线词典应用。

如果使用得当，那么诸如qooxdoo之类的框架的价值无法估量。在结束qooxdoo介绍之前，我们还将介绍最后一个例子，这次利用它实现最复杂的用户界面组件之一：树。

7.3.3　qooxdoo 树组件

树组件是常用的比较复杂的但具备强大功能的导航组件之一。虽然诸如Tab控件和菜单等简单控件已经在Web应用中被采用，但树组件还没有普遍地被大家欣然接受，或许是因为采用树组件所付出的成本与瞬态应用[①]极不匹配。

尽管Ajax为开发业务线Web应用拓展了空间，我们可以预计对诸如树组件之类的更高级组件的需求会增长。qooxdoo凭借其眼光准确地瞄准了该领域，它为我们提供了现成的树组件。在本例中，我们将看到如何让树组件运行起来。

1. 问题

我们想给用户提供一种数据展现方式，它们被分成很多种类和子种类，而不必同时超负荷地加载成百上千种数据。

2. 解决方案

使用树组件！由于树控件的复杂性，我们不想陷入细节实现的困境，例如监测节点事件处理器、描绘节点之间的连线等。qooxdoo库提供的树控件允许我们关注手边的业务：组织数据并展现给用户。对本例来说，我们已经添加了三类分型图像，它们属于被称之为“混沌吸引子（Strange Attractors，又称‘奇异吸引子’）”的一类分型图像。二维或三维混沌模式（chaotic patterns）又对此类图像进行了二次分类。图7-10显示了该应用程序的外观，不过由于当前打印机技术的限制，

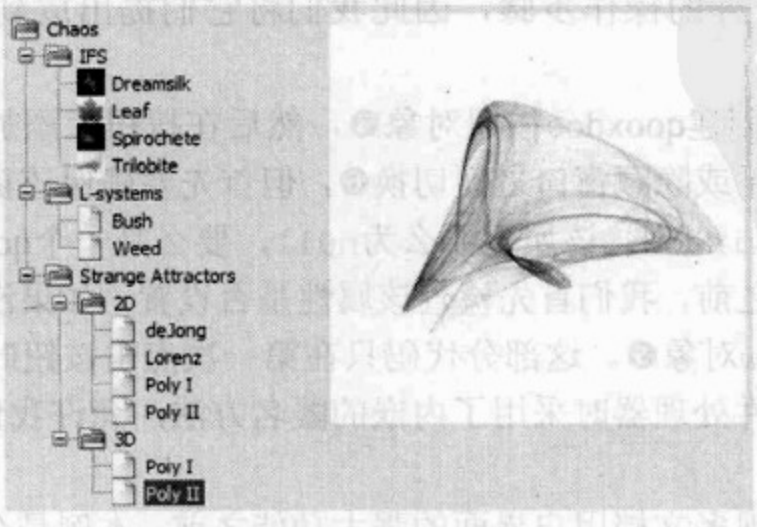


图7-10　qooxdoo树控件

① 原文是casual-use application，实际上这里指的是瞬态应用，详见《Ajax实战》第11页的说明。——译者注

你不得不通过自己下载并运行示例来观看3D动画图片。代码清单7-5显示了如何使用该树控件。

代码清单7-5 使用qooxdoo树控件

```
<html>
  <head>
    <title>Chaotic Images - Tree Navigation</title>
    <script src='../assets/js/qooxdoo/include.js'
      type='text/javascript'></script>

  <script type='text/javascript'>
    <!--
window.application.main = function()
{
  var d = this.getClientWindow()
    .getClientDocument();

  var panel = new QxImage(                    ❶ 创建预览面板
    '../assets/images/chaos/ifs/spirochete.jpg'
  );
  panel.set({
    left: 204, top: 48,
    width: 320, height: 320
  });
d.add(panel);

var addNode=function(parent,title){           ❷ 添加普通节点
  var node=new QxTreeFolder(title);
  if (parent){
    parent.addToFolder(node);
  }
  return node;
};

var addLeafNode=function(                     ❸ 添加叶子节点
  parent,title,image,hasThumbnail
){
  var iconUrl=(hasThumbnail) ?
    image+'16.jpg' : null;
  var mainUrl=(hasThumbnail) ?
    image+'.jpg' : image;
  var leaf=new QxTreeFile(title,iconUrl);
  leaf.addEventListener(                      ❹ 添加事件处理器
    "click",

    function(e){
      panel.set({ source:mainUrl });
    }
  );
  parent.addToFolder(leaf);
  return leaf;
};
```

❺ 创建树的根节点

```
var tree = new QxTree('Chaos');
tree.useTreeLines=true;
tree.set({
  left: 20, top: 48,
  width: 180, height: 320
});
```

❻ 开始为根节点添加子节点

```
var nodeIFS=addNode(tree,'IFS');
addLeafNode(
  nodeIFS,'Dreamsilk',
  '../assets/images/chaos/ifs/dreamsilk',
  true
);

addLeafNode(nodeIFS,'Leaf',
  '../assets/images/chaos/ifs/leaf',true
);
addLeafNode(
  nodeIFS,'Spirochete',
  '../assets/images/chaos/ifs/spirochete',
  true
);
addLeafNode(nodeIFS,'Trilobite',
  '../assets/images/chaos/ifs/trilobite',true
);

var nodeLS=addNode(tree,'L-systems');
addLeafNode(nodeLS,'Bush',
  '../assets/images/chaos/ls/bush.jpg'
);
addLeafNode(nodeLS,'Weed',
  '../assets/images/chaos/ls/weed.jpg'
);

var nodeSA=addNode(tree,'Strange Attractors');
var nodeSA_2D=addNode(nodeSA,'2D');
nodeSA.addToFolder(nodeSA_2D);
addLeafNode(nodeSA_2D,'deJong',
  '../assets/images/chaos/sa/sa2d/deJong.jpg'
);
addLeafNode(nodeSA_2D,'Lorenz',
  '../assets/images/chaos/sa/sa2d/lorenzII.jpg'
);
addLeafNode(nodeSA_2D,'Poly I',
  '../assets/images/chaos/sa/sa2d/quad.jpg'
);
addLeafNode(nodeSA_2D,'Poly II',
  '../assets/images/chaos/sa/sa2d/quad2.jpg'
);

var nodeSA_3D=addNode(nodeSA,'3D');
addLeafNode(nodeSA_3D,'Poly I',
```

```
        '../assets/images/chaos/sa/sa3d/KRTY_240.gif'
      );
      addLeafNode(nodeSA_3D,'Poly II',
        '../assets/images/chaos/sa/sa3d/MMDW_240.gif'
      );

      d.add(tree);
    };
        //-->
      </script>
    </head>
    <body>

    </body>
  </html>
```

本例的代码量比前一个例子要稍微多点，但大部分都是循环调用辅助方法添加树节点。大部分功能在前半部分实现，因此我们来看看它的实现。

目前，从window.application.main()方法开始使用qooxdoo，大家应该不再对此感到陌生。我们需要为该应用创建两个组件：树控件和右侧的预览面板。该预览面板用于显示相应的图片。预览面板的初始化过程很简单❶，创建该对象的同时我们随意加载了一张图片作为开始。

创建树组件期间，需要重复执行一些操作，因此我们定义了两个辅助函数以使代码尽可能保持简洁。qooxdoo树组件按节点是否包含子元素把节点区分为非叶子节点和叶子节点。因此，我们分别为它们提供辅助函数，addNode()函数提供了一种添加非叶子节点或按“文件夹”（qooxdoo的说法）的机制❷；addLeafNode()方法也提供了一种添加叶子节点或“文件”的方式❸。非叶子节点的打开或关闭事件被qooxdoo库自动处理，但我们需要为叶子节点添加自己的事件处理器❹。当点击该叶子节点时，我们希望它对应的图片显示到预览面板上。我们还提供了替换叶子节点图标的参数，为叶子节点提供缩略图以替代树组件的标准图标。你可以在图7-10中看到我们为树组件的前4个节点提供了缩略图，以演示如何对该组件进行客户化开发。

现在，我们已实现了辅助方法，可以准备开始创建树组件了。首先，我们声明树的根节点❺，然后开始使用辅助方法❻，直到用完图片集合中的所有图片。就这些——我们的树控件已经完全可以操作并能将图片显示到预览面板上。

3. 讨论

相比Tab或工具栏，树控件相当复杂，而且它需要更多的编码才能实现，但这更大程度上应归咎于日益复杂的数据模型，目前我们已经增加了额外的数据分类。一旦把辅助方法放到恰当的位置，我们就不至于偏离主题：支持低级用户界面组件，还有大部分代码只是描述领域模型（domain model）。

本例中，树组件的内容很少，刚好适合手工编码。在数据种类更多的情况下，我们或许会利用Ajax按需获取子类数据，限于篇幅，我们在此不作进一步讨论。

至此，我们结束了对桌面应用中导航技术的尝试。在7.2节中，我们介绍了传统的Web风格的导航实现，可以看出那些例子和本节展示的例子有着天壤之别。我们填补了基于瞬态应

用和业务线应用[①]中间地带的部分鸿沟，前者倾向于Web风格的导航系统，后者倾向于采用更复杂的桌面应用风格的导航技术。尽管如此，两者之间的模糊地带并非遥不可及，并且可能存在某种有趣的折中方案。本章结尾，我们将介绍混合导航模式，它结合了桌面应用和Web应用的最佳特性。

7.4　桌面应用和 Web 应用的折中

Ajax出现之前，Web应用局限于缺乏用户关注[②]的场合，例如购物/商业网站、搜索、在线词典和查找。用户可能在一天内经过多次思考，但通常只是暂时使用一次，并且把它作为其他更为复杂的任务的补充手段。复杂任务本身会通过桌面应用或某个瘦客户端应用完成。

程序所需的用户交互设施很简单，显然是用户界面控件。当Ajax进入由瘦客户端占据的领域时，它可以采用桌面应用使用的更加复杂（以及要求更为特殊和苛刻的）的导航技术，例如我们在7.3节中展示的那些范例。不过，很多开发者和设计人员不愿意放弃Web应用中的导航技术，并试图将两者的优势结合。本节，我们将介绍几个采用了这种妥协方案的示例。

7.4.1　OpenRico 库的 Accordion 控件

OpenRico提供的Accordion控件很实用，它减少了界面的冗余信息并能有效地利用有限的空间。有了它，用户可以只展示一小部分内容，但仍可以快速切换到相关信息项目。

或许你曾经听说过Accordion控件的另一个名字：Outlook工具条。它由一个面板和若干工具条组成。内容面板显示当前选中的内容。其他几个工具条拥有标题并保留可选择的内容片段。你可以立即在图7-11看个究竟。当你点击工具条访问其包含的内容时，OpenRico库以渐进式动画效果隐藏旧的信息并显示新选择的信息。相当酷！

1．问题

你的应用程序包含一组若干个分离的内容组成，而你想在同一时刻只显示一个片段。这些信息片段可以是其他组件：导航组件、表单，等等。

2．解决方案

OpenRico提供了一个叫做Accordion的DHTML组件。该组件将几个独立信息片段组合到一起并提供一种受欢迎的视觉效果，在任意给定时刻仅显示一个信息片段。该组件易于使用，但需要在HTML表单代码中添加少量标签。我们将在代码清单7-6中向你展示如何实现。首先，看一下如图7-11所示的结果。

首先，你需要定义一个 容器元素。在本例中，我们将它命名为imageAccordionDiv。在该容器元素内部，我们接下来需要添加子容器，用于表示即将添加到该组件的每个信息片段组。这些子容器通过后缀panel识别，并且依次包含两个 元素。每个子容器的首个 元素

① 原文为line-of-business application，本书已有注解，详见1.1节。——译者注

② 原文为user's attention，这里是指在人机交互方面，相对桌面应用而言，Web应用缺乏对用户的关注。——译者注

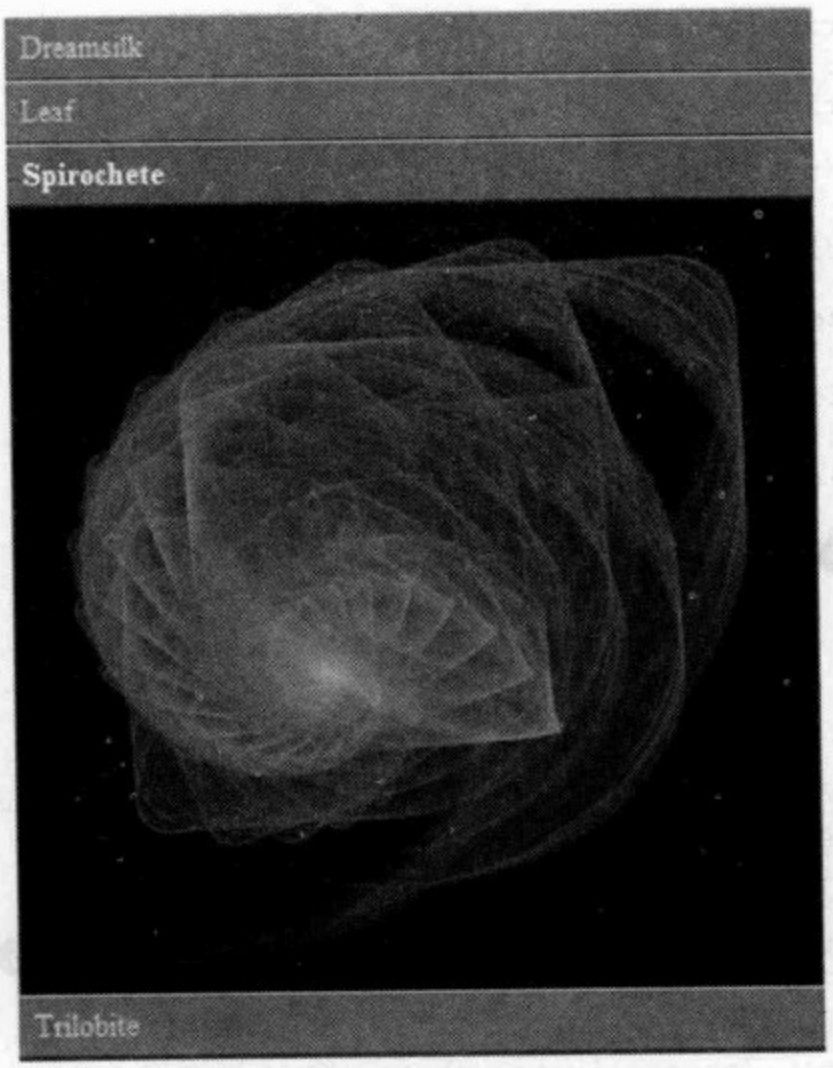


图7-11 OpenRico库提供的Accordion组件

用于表示标题工具条，它不同于当前显示的其他信息组标题，它只是为用户提供可视化的操作提示。标题工具条 通过后缀Header识别。每个子容器的第二个 将用于容纳给定的内容。这些 可以通过后缀Content识别。

一旦用于Accordion组件的HTML元素定义完毕，我们就可以通过必需的JavaScript初始化该组件，其代码位于initialize()方法中。Accordion对象的构造函数接受容器元素 以及一组配置作为参数。本例中我们只设置panelHeight属性，但还有更多的参数可用。

代码清单7-6中有一处代码格外值得注意：在Accordion构造函数中，我们使用目前已经熟知的$()方法获取容器 的引用。我们在早前的例子中看到过jQuery库$()的用法，但在本例中，我们使用的是Prototype，Rico基于它实现。

代码清单7-6 使用OpenRico库提供的Accordion组件

```
<html>
  <head>
    <title>Chaotic Images</title>
    <link href='../assets/css/rico.css'
          media='all' rel='Stylesheet' type='text/css' >

    <script src='../assets/js/prototype.js'
```

```
type='text/javascript'></script>
    <script src='../assets/js/rico.js'
type='text/javascript'></script>

    <script type='text/javaScript'>
      <!--
window.onload=initialize;

function initialize()
{
  new Rico.Accordion(
    $('imageAccordionDiv'),
    {panelHeight:320}
  );
}
      //-->
    </script>
  </head>
  <body>

    <div id='imageAccordionDiv' style='width:322px;overflow:hidden'>
      <div id='dreamsilkPanel'>
        <div id='dreamsilkHeader'
          class='accordionTabTitleBar'>
          Dreamsilk
        </div>
        <div id='dreamsilkContent'>
          <img src='../assets/images/chaos/ifs/dreamsilk.jpg'>
        </div>
      </div>
      <div id='leafPanel'>
        <div id='leafHeader' class='accordionTabTitleBar'>
          Leaf
        </div>
        <div id='leafContent'>
          <img src='../assets/images/chaos/ifs/leaf.jpg'>
        </div>
      </div>
      <div id='spirochetePanel'>
        <div id='spirocheteHeader' class='accordionTabTitleBar'>
          Spirochete
        </div>
        <div id='spirocheteContent'>
          <img src='../assets/images/chaos/ifs/spirochete.jpg'>
        </div>
      </div>
      <div id='trilobitePanel'>
        <div id='trilobiteHeader' class='accordionTabTitleBar'>
          Trilobite
        </div>
        <div id='trilobiteContent'>
          <img src='../assets/images/chaos/ifs/trilobite.jpg'>
        </div>
      </div>
```

❶ 创建可折叠对象

❷ 声明主容器

❸ 声明可折叠面板

❹ 为每个元素声明标题

❺ 为每个元素声明内容

```
    </div>

  </body>
</html>
```

本例的核心代码是initialize()方法。该方法简单地利用名称为imageAccordionDiv的 内容创建了一个新的Accordion组件，还指定该组件内容面板的高度为320像素❶。

其余代码依赖于你希望在该Accordion组件显示的那些内容的正确设置。你会需要一个用于整个Accordion组件的容器 ❷，我们将其声明为imageAccordionDiv。注意，该名字在代码前面的Accordion构造方法中❶作为参数被传入。然后，我们需要在该 内部声明若干个 元素，它们用于在Accordion组件内部显示。对每个要显示的元素，你需要声明以下HTML元素：

- 用于Accorion面板的容器 元素❸；
- 用于显示标题工具条的 元素，采用名称为accordionTabTitleBar的级联样式表装饰❹；
- 用于显示实际内容 ❺。

Accordion对象负责管理所有其他的事情!

3. 讨论

使用Accordion控件为Web应用增添某些动态效果是一种不错的方式。当然，它也有缺点。每个内容块必须占用和其他块相同的空间。如果不想这么做，那么开发人员必须提供相应的级联样式表以支持在每个内容 元素内部滚动。

有件有趣的事情值得注意，当把Accordion组件和7.3节中的例子进行对比时，我们发现OpenRico和qooxdoo采取了完全不同的方式来解决创建DHTML组件所遇到的问题。对OpenRico库来说，它要求开发人员先声明和布局HTML标签，然后负责其余的所有工作。可是，由于qooxdoo库的目标是创建完整的GUI库，因此它强调完全采用JavaScript API，而不需要在此之前声明HTML标签，起码它们由核心框架元素负责管理。结果是，或者你需要编写少量的JavaScript代码但需要事先完成大量HTML的声明布局工作，或者编写大量的JavaScript代码和少量HTML声明，看你如何取舍。

7.4.2 创建HTML友好[①]的树控件

在开发交互性导航控件方面，Rico库Accordion控件展现出一种有趣的方式，即我们通过声明普通HTML来构成组件的结构原型，然后使用JavaScript创建组件的交互操作。这与qooxdoo的实现方式完全不同。在qooxdoo中，完全使用JavaScript创建组件，所有的DOM元素都是通过框架自动生成的。我们曾提到，在导航以及应用程序的观感方面，传统Web和桌面应用存在着值得关注的模糊地带。在此，我们开始探索这个中间地带。

有个问题自然会被提及，即大家熟知的预先声明HTML和级联样式表的方案是否可以和纯JavaScript交互性方案相结合，并达到两全其美的效果。甚至，我们是否可以这样创建页面，

① 原文HTML-friendly，指的是结合HTML和JavaScript库，避免有些用户界面组件在关闭JavaScript的情况下失效。——译者注

它仍然能实现功能——虽然交互性差些——此时完全关闭使用JavaScript？在下一个例子中，我们打算为树控件实现此功能，它无可辩驳地成为迄今为止我们见过的最复杂、交互性最棒的树组件。

1. 问题

我们的数据有大量的种类和子种类，它适合用树控件展现。然而，应用应该能为更广大的用户服务，因此我们需要保证，在有些浏览器关闭JavaScript的情况下，它仍可使用。我们不想为应用保留两个完全独立的代码库，Ajax版和非Ajax版。因此，我们需要找到一种办法使同一个设计能适应不同类型的用户环境。

2. 解决方案

我们面对的是一种非常苛刻的需求！如果明白了非JavaScript版本的应用和JavaScript版的应用在功能上并不是一致的，那么我们就可以实现它。

在上一个例子中我们注意到，Rico库Accordion组件增添了在页面上声明HTML标签这一功能。这种实现方式符合目前的要求，因为我们需要纯粹（unadorned）的HTML提供基础功能。那么，它的特征应该是什么样的呢？

我们为该HTML程序选择了一种简单的交互模型，树组件上的每个叶子节点拥有一个直接到图片的链接。预览完某个图片，用户可以使用后退按钮返回到树组件。这并不是最佳的交互模式，但它能工作。图7-12显示了使用该模式和树交互的两种场景。

默认情况下，整棵树的内容完全展开显示所有节点，并且不能通过点击非叶子节点收缩。预览面板不使用该模式，因此它初始状态是隐藏的。点击一个叶子节点将导致浏览器显示全尺寸图片，用户可以通过点击后退按钮从此处返回到树。代码清单7-7显示了该树控件的HTML代码。

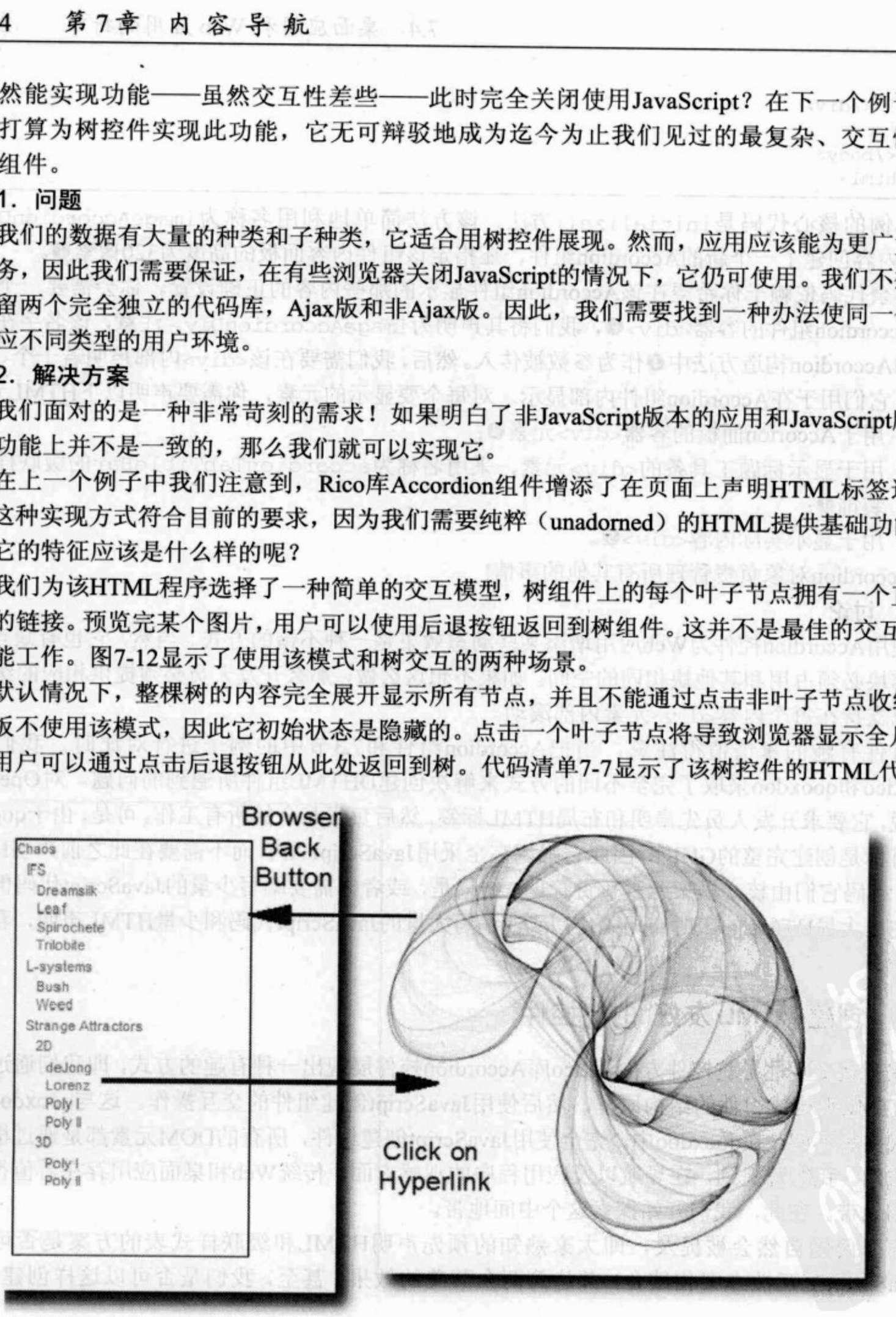


图7-12　与屏蔽JavaScript功能的树控件交互

代码清单7-7 树控件的HTML代码

```
<html>
  <head>
    <title>A Poem Lovely As A Tree</title>
    <link rel="stylesheet" type="text/css" href="main.css">
    <script type='text/javascript'
       src='scripts/prototype.js'></script>
    <script type='text/javascript'
       src='scripts/tree.js'></script>
    <script type='text/javascript'>
      window.onload=function(){
        initTree();
      };
    </script>
  </head>
  <body>
    <div class="pane" id="tree">
      <div class="nodeHeader" id="head_0">Chaos</div>
      <div class="nodeChildren" id="child_0">
        <div class="nodeHeader" id="head_1">IFS</div>
        <div class="nodeChildren" id="child_1">
          <a href='../assets/images/chaos/ifs/dreamsilk.jpg'>
            <div class="nodeHeader leaf"
              id="head_2">Dreamsilk</div>
          </a>
          <a href='../assets/images/chaos/ifs/leaf.jpg'>
            <div class="nodeHeader leaf"
              id="head_3">Leaf</div>
          </a>
          <a href='../assets/images/chaos/ifs/spirochete.jpg'>
            <div class="nodeHeader leaf"
               id="head_4">Spirochete</div>
          </a>
          <a href='../assets/images/chaos/ifs/trilobite.jpg'>
            <div class="nodeHeader leaf"
              id="head_4">Trilobite</div>
          </a>
        </div>
        <div class="nodeHeader" id="head_5">L-systems</div>
        <div class="nodeChildren" id="child_5">
          <a href='../assets/images/chaos/ls/bush.jpg'>
            <div class="nodeHeader leaf" id="head_6">Bush</div>
          </a>
          <a href='../assets/images/chaos/ls/weed.jpg'>
            <div class="nodeHeader leaf" title="ls/weed.jpg"
              id="head_7">Weed</div>
          </a>
        </div>
        <div class="nodeHeader" id="head_8">Strange Attractors</div>
        <div class="nodeChildren" id="child_8">
          <div class="nodeHeader" id="head_8a">2D</div>
          <div class="nodeChildren" id="child_8a">
            <a href='../assets/images/chaos/sa/sa2d/deJong.jpg'>
```

❶ 导入相关Java Script文件

❷ 开始声明树节点

```
            <div class="nodeHeader leaf" id="head_9">deJong</div>
          </a>
          <a href='../assets/images/chaos/sa/sa2d/lorenzII.jpg'>
            <div class="nodeHeader leaf" id="head_10">Lorenz</div>
          </a>
          <a href='../assets/images/chaos/sa/sa2d/quad.jpg'>
            <div class="nodeHeader leaf" id="head_11">Poly I</div>
          </a>
          <a href='../assets/images/chaos/sa/sa2d/quad2.jpg'>
            <div class="nodeHeader leaf" id="head_12">Poly II</div>
          </a>
        </div>
        <div class="nodeHeader" id="head_8b">3D</div>
        <div class="nodeChildren" id="child_8b">
          <a href='../assets/images/chaos/sa/sa3d/KRTY_240.gif'>
            <div class="nodeHeader leaf" id="head_13">Poly I</div>
          </a>
          <a href='../assets/images/chaos/sa/sa3d/MMDW_240.gif'>
            <div class="nodeHeader leaf" id="head_14">Poly II</div>
          </a>
        </div>
      </div>
    </div>
  </div>

  <div class='pane' id='preview' stylle='display:none;'>
    <img id='preview_img' src='../assets/images/chaos/ifs/
spirochete.jpg'></img>
  </div>
 </body>
</html>
```

❸ 声明并隐藏预览面板

首先，我们在该页面中引入所需的JavaScript代码库❶。因为打算在本例中使用Prototype，因此我们已经把它移至一个单独的文件：tree.js。在代码规模已经不小的演示程序中，这样做无疑是件好事。不过在本例中，这样做还有特殊的用途，因为我们期望为有些禁止JavaScript的用户提供支持。我们不希望在HTML中添加太多即将被忽略的内嵌脚本，这会浪费用户的网络带宽。

与使用qooxdoo树控件的示例一样（见代码清单7-5），在创建树对象时，我们有大量重复代码,但这一次我们采取普通HTML表示❷。我们在利用HTML文档和树型对象有类似结构这一事实，并且页面上嵌套元素遵循我们的树控件的结构。每个树节点由一个级联样式表class名为nodeHeader的 组成，它包含该节点的标题文字。如果是叶子节点，那些元素还拥有另外一个后缀名为leaf的级联样式表class，并且该 外围被一个锚点（anchor）标签包裹成一个链接。非叶子节点没有链接，但包含第二个 元素，该元素的级联样式表class名为nodeChildren。该 元素和nodeHeader是兄弟元素。所有的子节点被完全包含在nodeChildren元素内部，当我们简单地显示或隐藏子容器实现交互功能后，它将允许我们展开或收缩该节点。

该方案还几乎无偿为树控件提供了基础的结构布局，如果我们添加一点CSS来保证树左侧的每个子容器都有一个明显的缩进值，就能实现每个树层次的缩进。

最后，我们声明预览面板❸。我们只能在支持JavaScript的版本中使用它，因此，默认情况下，我们把该元素的style属性设置为“display: none”来隐藏它自己，直到使用程序显示它为止。

这样，我们就满足了少数不愿支持JavaScript的用户。对我们来说，该控件的外观将是什么样？图7-13显示了结果。

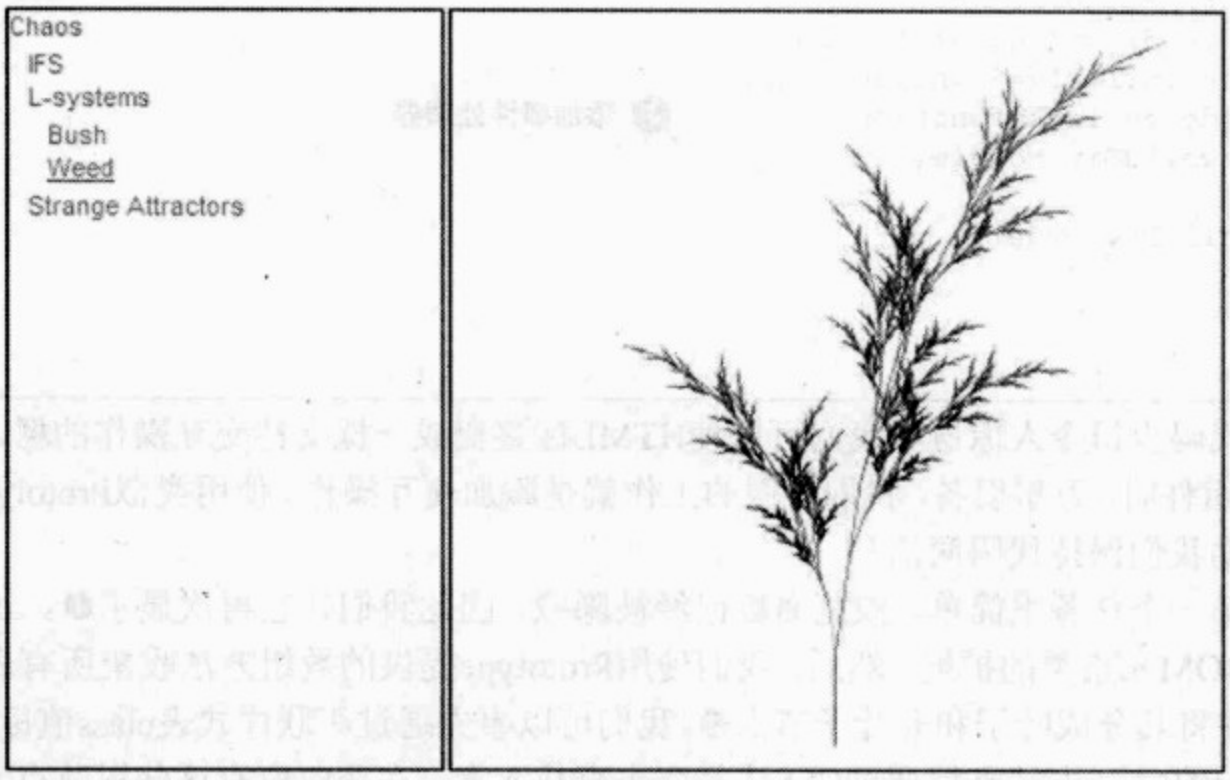


图7-13 支持JavaScript的树控件

在支持JavaScript的情况下，该树控件开始将显示为收缩状态，只有根节点和预览面板可见。在截图中，我们打开了几个已选中节点并点击了某个叶子节点，与该节点相关的图片目前显示在预览面板上。我们该如何实现该功能？代码清单7-8显示了tree.js的内容，我们用它改写了树节点的交互规则。

代码清单7-8 tree.js

```
function initTree(){
  $('preview').show();
  var allNodes=$$('.nodeHeader');
  var partitioned=allNodes.partition(
    function(node){
      return node.hasClassName('leaf');
    }
  );
  var leafNodes=partitioned[0];
  leafNodes.each(
    function(node){
      var anchor=node.parentNode;
      var imgsrc=anchor.href;
      anchor.href='#';
      node.onclick=function(){
        $('preview_img').src=imgsrc;
```

❶ 显示预览面板

❷ 分离叶子节点和非叶子节点

❸ 用onclick事件取代超链接元素

```
        };
      }
    );
    var nonLeafNodes=partitioned[1];
    nonLeafNodes.each(
      function(node){
        var childDivId=node.id
          .replace(/head/,"child");
        var childDiv=$(childDivId);
        node.onclick=function(){
          childDiv.toggle();
        }
        childDiv.hide();
      }
    );
  }
```

❹ 添加事件处理器

使用的代码少得令人惊讶，我们可以使HTML标签变成一棵支持交互操作的树。在使用Rico库Accordion组件时，万事俱备，我们要做的工作就是添加交互操作。使用类似Prototype之类的库，当然也能帮助我们保持代码简洁！

我们的第一个任务很简单。预览面板已经被隐藏，因此我们让它再次显示❶。show()方法是Prototype对DOM元素类的扩展。然后，我们使用Prototype提供的数组方法收集所有表示树节点的HTML元素并将其分成叶子和非叶子节点❷。我们可以事先通过级联样式表class值标识这些元素，而且Prototype的$$()方法支持通过CSS选择器来查找元素。在获取所有这些树节点元素之后，我们利用partition()方法返回一个包含两个元素的数组对象，该方法来自接口优雅的Prototype库的Array类。该数组中的所有元素本身也是数组。第1个数组元素包含所有通过指定的测试的树节点，第2个包括所有未通过该测试的树节点。该测试被定义为一个方法对象并作为参数传入partition()方法。该测试方法简单地再次检查节点的级联样式表的class来判断它是否是叶子节点。

然后，我们可以遍历所有叶子节点并对其进行修改以便用户使用支持JavaScript版本的树控件。我们首先需要做的事情就是让超链接元素的图片链接功能失效。这些超链接元素是我们为不支持JavaScript的用户着想而添加的。做完这个，我们为节点添加一个简单的由程序控制的事件处理器❸。

最后，我们遍历非叶子节点。在HTML版本的应用中，这些节点不支持交互操作，因此我们在此简单地添加该功能，辨别出容纳所有子节点的容器元素并在点击该节点的标题时切换其可见性❹。

3. 讨论

我们这里介绍的树控件没有7.3.3节中介绍的qooxdoo组件那么复杂，但对于几小时的工作量来说，它表现还不错。根据你的喜好和你的应用的类型，相比完全采用桌面开发风格的qooxdoo库而言，结构更扁平、界面风格更接近Web的方式或许更合适。当然，这两种方式可以共存。

我们在此实践的技术——允许Web应用在禁用JavaScript的情况下仍继续提供功能——经常

被称作适度降级（graceful degradation）[①]。如果你想支持更广范围的用户，这可能是一种成功的策略，而且不会承担太多的额外付出。从一开始就强迫进行内容和行为的分离，从而我们得以通过添加少量相关的代码而实现完整的交互性。

至此，我们将终止对导航技术和组件的研究，本章也就此结束。我们将在下章继续介绍用户的工作流，到时我们会解释完全支持历史记录机制的Ajax应用的工作方式。

7.5 总结

在思考Ajax应用的内容导航方式时，天空才是极限。而且已经有大量组件可自由选择，比如qooxdoo和OpenRico库中的各种控件。

我们正处于一个非常有趣的阶段，对于Web导航的思考同时汇集了来自传统Web设计领域的概念（如我们在7.2节中所见）以及桌面应用和胖客户端领域的概念（如我们在7.3节中所见）。拜Ajax的颠覆性本质所赐，这两者正在相互影响。也正是这种颠覆性本质让业务线应用也被纳入到Web应用的领地。

在7.4节中，我们研究了如何综合当前这两个领域的思想。与此同时，我们也涉及了如何从工作流程逻辑（workflow logic）上分离设计和内容的问题，还有怎样让不能或不愿使用JavaScript的用户也能使用导航。

① 适度降级（graceful degradation）指的是指电脑、机器、电子系统或网络本身在大部分已经毁坏或无效的情况下还能保持有限功能的能力。此处指的是在没有JavaScript的情况下，用户仍能访问Web应用并与之交互，只是界面没有原来那么酷。——译者注

第8章 处理后退、刷新和撤销

本章内容

- 禁用浏览器的导航功能
- Really Simple History框架
- 处理撤销（undo）操作

为Web应用设计动态内容的一个主要难点是最终用户应有能力刷新页面和任意导航历史记录。浏览器的导航设备对浏览静态内容很有用处，但在使用动态Web应用时，它引发了一些公认的尚未解决的复杂问题。例如，只须轻轻点击浏览器的后退按钮或按F5键刷新整个页面，客户端和服务器之间的状态就失去了同步。如果你在应用中使用了诸如可拖动内容之类的DHTML技术，客户端的布局状态将被破坏。上述问题甚至更多问题在支持Ajax的单页面应用中都存在。

在本章中，我们将研究几个技巧以阻止最终用户访问浏览器的历史记录导航设备或执行刷新页面操作。这些技巧包括打开一个移除了所有工具栏的新浏览器窗口、禁止使用快捷键访问历史记录和禁止右键弹出上下文菜单（content menu）。我们还将介绍一些能和这些浏览器功能相结合的技术，例如，在hash对象中存储以URL形式表示的应用程序状态；使用RSH（Really Simple History）框架为单页面Ajax应用轻松添加书签和历史记录导航功能，以及实现你自己的撤销栈。

8.1 禁止用户访问浏览器的导航控件

禁用浏览器导航控件的访问权，需要采取三个措施：必须禁止用户访问具备导航功能的各种浏览器工具栏；必须限制任何支持导航功能的快捷键；必须禁止显示上下文菜单。有一点很重要，用户不可能因为你强行减弱他们的用户体验而感到愉悦，这意味着你不得不为他们提供一种在应用中轻松进行导航的机制。让我们来看看如何最佳地处理这些问题。

8.1.1 移除浏览器导航工具栏

要移除浏览器的地址栏和导航栏，必须通过JavaScript编程打开一个新窗口。不可能在现有的、已打开的浏览器窗口上添加或移除工具栏。受此限制，你只能创建一个Launchpad页面，从该页面派生一个包含你的应用的新窗口。用于创建新窗口的JavaScript API非常简单和直接：

```
window.open(URL, name, options, replace);
```

为定制要打开的窗口，window.open()方法提供了大量的可定制选项[①]。关于用户定制选项，表8-1和8-2提供了详细而深入的介绍。

表8-1 window.open()方法的参数

参 数	类 型	描 述
URL	String	指定你想显示的页面的地址。如果不想一开始就加载某个页面，可以传入空字符串（如果你希望通过脚本为当前窗口动态生成页面内容，这有助于你达到该目的）
name	String	指定新建窗口的name属性。窗口的name属性可用作该窗口的引用，它和在frameset中指定frame是一个道理。例如，表单中的链接 <a target='thewindow' href='thepage.html'>将在名为thewindow的窗口中显示thepage.html。如果引用窗口已存在，那么，window.open()将不再打开新窗口，而是在该窗口显示内容
options	String	可选项。为新窗口指定可用的选项值。该参数可以包含一个或多个以逗号分隔的key=value对。对布尔型选项合法的值是yes、no、1或0。如果希望所有布尔型选项的默认值为false，你可以不设置它们
replace	Boolean	可选项。如果是true，新页面地址将替换浏览器历史记录中的当前页面（地址）。某些游览器或许不支持该参数

表8-2 window.open()支持的常用选项

选 项	类 型	描 述	默 认 值
width	Integer	窗口宽度(像素)	和父窗口相同
height	Integer	窗口高度(像素)	和父窗口相同
left	Integer	窗口左上顶点的x坐标	Auto
Top	Integer	窗口左上顶点的y坐标	Auto
scrollbars	Boolean (yes, no, 1 或 0)	确定滚动条是否有效	Yes
resizable	Boolean (yes, no, 1 或 0)	确定窗口是否可拉伸	Yes
toolbar	Boolean (yes, no, 1 或 0)	确定是否应该显示工具栏	Yes
location	Boolean (yes, no, 1 或 0)	确定是否应该显示地址栏	Yes
directories	Boolean (yes, no, 1 或 0)	确定是否应该显示链接栏	Yes
status	Boolean (yes, no, 1 或 0)	确定是否应该显示状态栏	Yes
menubar	Boolean (yes, no, 1 或 0)	确定是否应该显示菜单栏	Yes

该方法的常见用法是创建不含任何工具栏的窗口，实际上就是打开一个未加装饰的简单窗口。用于创建不含工具栏窗口的函数的一般写法大概如下：

```
function openWithoutToolbars(URL, windowName) {
    window.open(
        URL,
        windowName,
        'status=1,scrollbars=1,resizable=1',
        true
    );
}
```

至此，我们就移除了工具栏上的可见按钮，但是用户仍然可以通过使用快捷键或右键上下文

① 原文是customization options，此处是指window.open()方法支持的可选参数。——译者注

菜单实现同样的功能。我们再来看看如何删除它们。

8.1.2 捕捉快捷键

捕捉快捷键操作涉及在文档对象层次上添加事件处理器以拦截相应快捷键（我们在第5章已经讨论过JavaScript事件模型）。控制浏览器导航工具栏和当前已加载页面状态的常见快捷键有8个，如表8-3所示。

表8-3　用于历史记录导航的键盘快捷键

快捷键	描述
Backspace	在历史记录中后退
Alt/Option+左箭头键	在历史记录中后退
Alt/Option+右箭头键	在历史记录中前进
Ctrl/Command+左箭头键	在历史记录中后退
Ctrl/Command+右箭头键	在历史记录中前进
F5	刷新窗口
Ctrl/Command+R	刷新窗口
Ctrl/Command+H	显示历史记录
Alt/Option+Home	到主页

要监测这些按键事件，必须在文档对象上挂接一个keydown事件处理器（如果使用Mozilla/Firefox浏览器，我们还必须添加keypress事件处理器）以检查用户按下哪个键，以及用于监测组合键的任何修饰键，并阻止事件在DOM树中继续传播。此外，由于退格键也是8个快捷键之一，我们需要做一项特殊处理，以便当用户在获得焦点的文本框和输入框按下退格键时允许该事件继续执行。

在8.14节中，我们将简单展示一个使用该技术的示例。现在，继续介绍下一个特性：上下文菜单。

8.1.3 禁止右键弹出上下文菜单

上下文菜单是指当用户在浏览器窗口的内容区域点击鼠标右键时弹出的独立菜单。上下文菜单包含一些导航功能，因此我们也要禁止该菜单的出现。多数浏览器的新版本提供了一个叫做oncontextmenu的事件，在最终用户点击鼠标右键时，该事件触发。要让该菜单的功能失效，只需简单地在文档对象上注册oncontextmenu事件处理器，在处理器中阻止该事件的传播。

好了，在我们的措施清单中提到的各种技巧都已介绍完毕。让我们把新学到的知识付诸实践并查看一个实际的例子。

8.1.4 阻止用户导航历史记录或刷新页面

开发Ajax应用，有些场合必须阻止用户访问浏览器的导航工具栏或页面刷新。因为Ajax应用

的用户界面频繁更新，页面刷新或后退操作将破坏当前应用的状态，还会导致用户操作丢失，而不是把他们带到历史记录中的上一次访问页面。

1. 问题

你开发了一个单页面的Ajax应用，并且需要移除用户导航浏览器历史记录或刷新页面的能力。

2. 解决方案

提供一种机制来打开移除了所有工具栏的新浏览器窗口是创建所有导航功能均被禁用的浏览器窗口的第一步。你可以在图8-1看到页面显示效果。我们将创建一个Launchpad页面（代码清单8-1），它作为注册页面（registration page）[①]的起点。该页面不允许最终用户在历史记录中导航、浏览其他页面、或刷新/重新加载注册页面的内容。

代码清单8-1 Launchpad页面

```
<html>
  <head>
    <title>Application Launchpad</title>
    <script type='text/javascript'>

function openWithoutToolbars(URL, windowName, width, height) {
    window.open(
        URL,
        windowName,
        'status=1,scrollbars=1,resizable=1'+
        (width  ? ',width='+width   : '')+
        (height ? ',height='+height : ''),
        true
    );
}

function openRegistration() {
    openWithoutToolbars(
        './registration.html',
        'REGISTRATION',
        480, 580
    );
}

    </script>
  </head>
  <body>
    <a href='javascript:openRegistration();'>
      Launch Registration
    </a>
  </body>
</html>
```

一组用于移除工具栏的options参数

① 即registration.html页面。

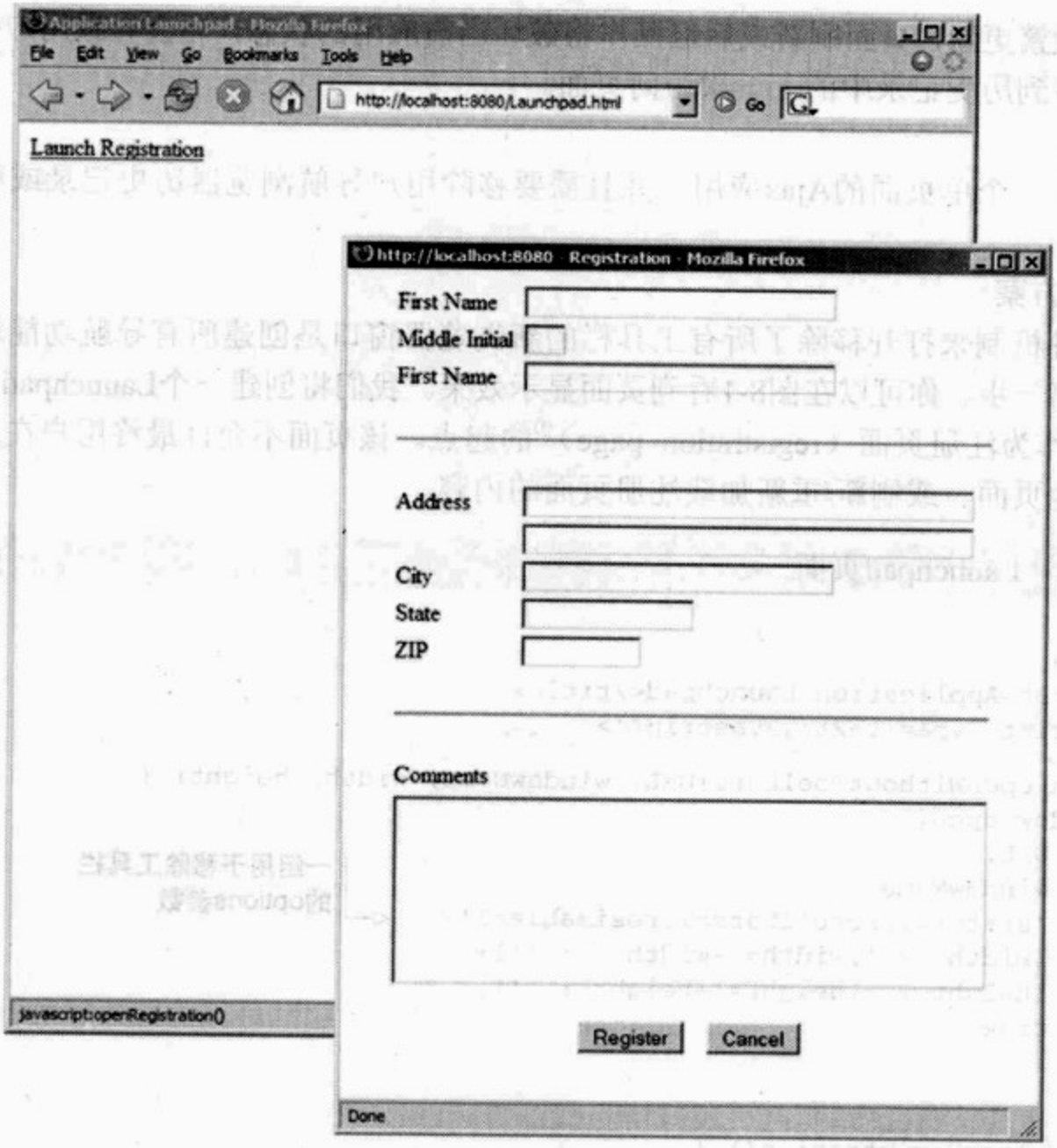


图8-1　没有工具栏的注册页面

第二步（如代码清单8-2所示），禁用所有可用于在新打开的窗口内进行历史记录导航和页面刷新的快捷键。为了保证尽量使用符合“试用并已经过测试”条件的第三方工具库，我们将在该示例中使用Prototype库提供的跨浏览器事件注册机制，因此需要在页面中引用prototype.js。Prototype库为我们提供了两个关键方法：Event.observe()方法用于在对象上注册事件，Event.stop()方法用于阻止事件继续传播。

代码清单8-2　Launchpad应用中的JavaScript代码

```
var isHistoryShortcutDisabled = false;

Event.observe(                          ❶ 禁用上下文菜单
  document,
  'contextmenu',
  function(event){
    Event.stop(event);
    return false;
  }
```

```
);
Event.observe(document,'keypress',
  checkHistoryShortcutDisabled);
Event.observe(document,'keydown',
  disableHistoryShortcuts);

function disableHistoryShortcuts(event) {
    var targetTag = Event.element(event).tagName;
    var isTextInput = (
      (targetTag == 'TEXTAREA')
      || (targetTag == 'INPUT')
    );

    var keyCode = event.which || event.keyCode;

    if (( keyCode == 116) ||
        ((keyCode ==   8) && (!isTextInput))    ||
        ((keyCode ==  36) && event.altKey)      ||
        ((keyCode ==  37) && event.altKey)  ||
        ((keyCode ==  39) && event.altKey)  ||
        ((keyCode ==  37) && event.ctrlKey) ||
        ((keyCode ==  39) && event.ctrlKey) ||
        ((keyCode ==  82) && event.ctrlKey) ||
        ((keyCode ==  72) && event.ctrlKey)) {
        isHistoryShortcutDisabled = true;
        Event.stop(event);
        return false;
    }
}
function checkHistoryShortcutDisabled(event){
    if (isHistoryShortcutDisabled) {
        isHistoryShortcutDisabled=false;
        Event.stop(event);
        return false;
    }
}
```

❷ 禁用快捷键

❸ 禁用导航快捷键

❹ 在Mozilla浏览器中禁用keypress事件

首先，我们禁用了上下文菜单❶。在回调函数中要做的事情就是停止事件传播，因此我们定义了一个匿名方法。捕获keypress❷需要多费一点脑筋，所以对回调函数进行了单独定义。

主要的回调函数是注册到keydown事件上的diableHistoryShortcuts()。在该函数中，需要我们辨别出那些要捕捉的导航快捷键❸并禁止这些快捷键事件传播。这要求我们能将其他的非导航快捷键区别出来，比如正在输入文本的字段，或者确实是其他响应按键事件的任何可输入字段，例如下拉列表。我们还需要在Mozilla浏览器❹中跟踪keypress事件。无论如何，我们都要注册该事件[①]的回调函数，在IE浏览器中，它不会产生副作用。最后，我们努力换来的结果是，我们将拥有一个既没有导航功能也没有页面刷新功能的窗口。

3. 讨论

我们刚看到一个完整的方案，用于删除最终用户导航历史记录和刷新页面的能力。尽管该方

① 即前文说的keypress事件。——译者注

案非常有效，但可能导致用户对你的应用感到不满。没有人喜欢别人限制自己的自由，某些最终用户会觉得你只是剥夺了他们在Web应用中进行浏览和导航的权利，这些操作对他们来说已经习以为常。如果你打算采用这种技术，请记住这一点。

无论用户是否需要，我们必须拦截并改造浏览器原有的导航功能，重新赋予用户访问它们的权利。如果你覆写了本机浏览器的既有导航控件，提供一种替代性的导航能力将变得极其重要。如果不给用户提供替代方案乃至书签功能，他们大概不会喜欢使用你的应用。此刻，Google公司的GMail和Map应用示例跃入我们的脑海。尽管Google这些应用拥有动态客户端界面，它们依然允许用户使用浏览器的既有导航控件在应用中（甚至制作标签）后退或前进。Google只是暗中把这些控件的功能变成他们自己的功能。接下来，我们将向你具体展示如何实现它。

8.2　与浏览器导航控件协作

如果你希望应用能提供丰富且限制较少的用户体验（两者将使最终用户更加高兴），你只好支持浏览器的历史记录导航和页面刷新能力。这可能是一项让人畏缩不前的艰巨任务。在用户刷新页面、点击后退按钮或转到一个完全不同的页面，然后又退回到应用的情况下，该如何维护应用的状态呢？你可以使用几个技巧来保存单页面应用的状态，无论是页面刷新还是实现理所应当提供的书签功能所引发的状态。因此，当用户点击后退或前进按钮时，不至于对应用的表现感到异常惊讶（应用重新回到某些默认状态甚至更糟）。结果是，用户可以单步回滚他们的操作记录。

8.2.1　使用 JavaScript 内建的 history 对象

借助history对象，JavaScript提供了可通过编程控制的历史记录导航。使用该对象，你能模拟浏览器后退和前进按钮、在动态Web应用中提供链接以返回之前的页面（即使当前页面有多个操作入口）或强制浏览器永远显示浏览记录的最后一个页面。把下列代码添加到应用的所有页面中可以实现该功能：

```
window.onload = function() {history.go(1);}
```

不过，这是一种极不优雅的编程手法，它很可能招致Web开发社区对你进行指责。history对象的属性和方法分别在表8-4和表8-5中展示。

表8-4　history对象的属性

属　性	描　述
length	该history对象中包含的浏览项目个数

表8-5　history对象的方法

函　数	描　述
back	加载历史记录列表中的前一个URL
forward	加载历史记录列表中的下一个URL
go	转到历史记录列表中指定的URL。参数where可以是一个整型值或字符串，如果使用整型值作为参数，转到相对于当前文档的指定位置处的URL。例如，-1后退一个页面，1前进一个页面。如果参数是字符串，转到和该字符串局部匹配或完全匹配的URL

8.2.2 使用 Hash 对象实现书签

Hash对象是URL的一部分，位于URL的尾端，并被预先设计为以一个#符号作为开始。在Web网站的典型用法中，Hash对象用于指明浏览器应当访问的某个已命名的锚标签。注意，要更新浏览器的当前位置（location）并且不导致页面完全重新加载，唯一途径就是使用Hash对象。通过一个简单、短小的正则表达式就可以访问该对象的值。

```
var hash = window.location.href.replace(/(.*)(#)(.*)/,'$3');
```

该hash对象的值适用于各种情形，但它最好用于捕捉应用程序的状态快照。你可以用它直接存储少量应用状态，或者把它作为主键去访问更大规模的、更复杂的状态快照。例如，如果你有一个气象服务应用，它能显示多个地区的天气情况，你可以允许最终用户在选中他们要查看的某个位置之后添加书签。这样的话，当他们下次访问该应用时，就不必选择地区了。

1. 问题

你需要在Web应用中设置一个参考基点，以便最终用户能够在稍后浏览时添加书签。

2. 解决方案

我们从本节一个非常简单的例子开始，其中的JavaScript代码将监视当前页面URL的hash值的变化情况并相应改变表单内容。该示例的用户界面相当简洁，如图8-2所示，实现代码见代码清单8-3。

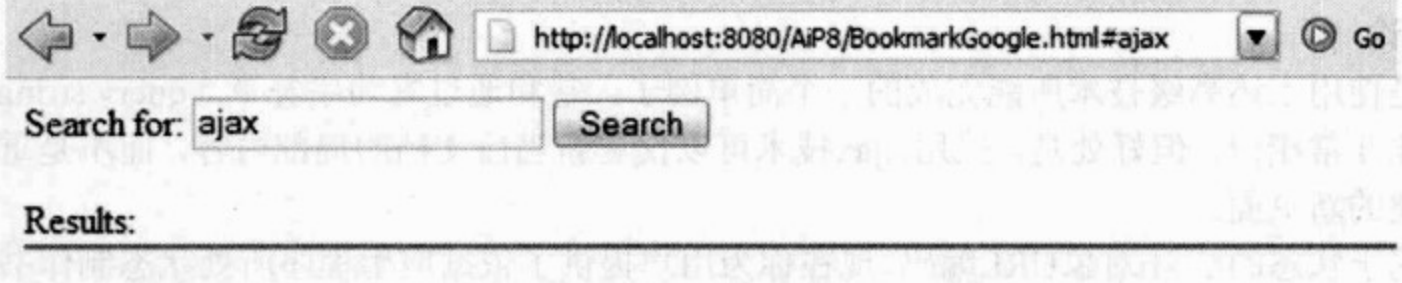


图8-2 简单的书签示例。文本输入框的内容将设置到URL末尾的hash对象

代码清单8-3 处理Hash对象的JavaScript代码

```
var ajaxRequest;
var currentHash;

setInterval('checkHash()', 250);

function checkHash() {
    var newHash = getHash();

    if (newHash && (newHash != currentHash)) {
        currentHash = newHash;
        getBookmark();
    }
}

function getHash() {
    if (window.location.href.indexOf('#') > -1) {
        return window.location.href.replace(/(.*)(#)(.*)/,'$3');
```

❶ 设置检查时间间隔

❷ 检测当前视图状态

❸ 发起到服务器的请求

❹ 从href提取用于表示状态的hash对象

```
        } else {
            return null;
        }
    }

function getBookmark() {                    ❺ 创建Ajax请求
    new Ajax.Request(
      '/servlet/Bookmarks?bookmarkId='+currentHash,
      { method: 'get',
        onComplete: function(xhr){
          eval(xhr.responseText);           ❻ 对返回结果执行eval
        }
      }
    );
}
```

代码清单8-3中的代码检测window.location.href属性❹的值，如果某个状态存在，我们还将校验标签网页是否不等于当前浏览页面❷。该方法叫做checkHash()，它将每隔0.25秒钟执行一次❶。如果发现一个新的URL出现，它❸将调用getBookmark()，该方法将使用一个XMLHttpRequest对象(Ajax.Request对它进行了整洁的封装)与服务器进行交互❺，它将该hash值作为书签ID传入函数。该方法中的回调函数将对服务器返回的数据进行eval操作并重现书签状态❻。

3. 讨论

本例是使用上述高级技术所能完成的一个简单例子。它和通过查询字符串（query string）传递数值相比非常相似，但好处是，采用Ajax技术可以仅更新当前文档的局部内容，而不是重新请求一个完整的新页面。

采用基于状态的hash对象URL编码，现在你为用户提供了依据应用程序所处状态制作书签的功能。或许你会对第11章的内容感兴趣，该章讨论客户端状态管理。你应当有能力在客户端存储应用程序的状态而不是将其存储在服务端。正如我们讨论的那样，状态在一个hash对象上进行了索引。当某个用户定位到书签定制过的hash对象，可以从客户端的状态缓存而不是服务器恢复对应的状态。你不必担心状态维护影响到负荷已经很重的服务器，它完全把状态维护的责任推给客户端实现。

既然我们已经有能力让用户在动态Web应用中制作书签，让我们继续前进介绍另一个棘手问题：维护动态Web页面的历史记录。RSH（Really Simple History）框架此刻要为我们展示实现途径，而且它使用我们刚才讨论过的URL状态散列（URL hashing）技术。

8.2.3　RSH 框架介绍

继续探索和浏览器的历史记录及状态有关的问题。我们知道，很多框架支持在Ajax应用中操作历史记录。其中，由Brad Neuberg创建的RSH（Really Simple History）框架凭借极佳的易用性和无以伦比的功能独占鳌头。你可以到http://codinginparadise.org访问它。RSH提供了存储客户与Web应用交互过程产生的多个历史记录事件的能力。每个事件与一个保存在文档的URL相联系。

当用户点击浏览器的后退和前进按钮时，RSH使用该Hash对象的值获取与其相关的事件并调用任何注册到该事件的事件处理器。

RSH提供了大量的功能，但目前我们将关注基础知识。RSH实现管理历史记录状态的机制是dhtmlHistory对象。该对象提供了四个主要方法（表8-6），包括自身初始化、注册事件监听方法、获得当前位置和增加历史记录事件。

表8-6 dhtmlHistory对象的方法

函 数	描 述
initialize	初始化dhtmlHistory对象本身。应该在window.onload中调用
getCurrentLocation	返回以字符串表示的当前页面的位置
addListener	添加一个历史记录事件处理器以处理历史记录改变事件
add	存储一个历史记录事件。当页面URL发生改变并包含以该位置值表示的Hash对象时，历史记录事件监听方法被触发，传入的参数是location和event的数据。add()接受两个参数。第一个是事件作为主键存储在URL上的对象。第二个是和事件有关的数据

下面来看一些使用RSH的实际例子。

8.2.4 使用 RSH 框架维护客户端状态

在本节及下节的示例中，我们将使用RSH对7.4.2节中介绍过的树组件进行增强。操作该树组件的时候，页面状态改变了多次，但浏览器的历史记录维护系统没有捕捉过一次状态改变。这会使得我们的用户感到手足无措。下一个示例，我们将用RSH实现一个客户端状态管理系统，它能很好地与浏览器的后退和前进按钮协同工作。

1. 问题

你的应用包含一些编程可控的状态迁移，用户可能认为它们是一些“新页面”。因此他们期望历史记录按钮能支持这些变化，否则他们会感到失落。你需要为你的单页面应用提供支持用户操作的状态管理机制。你只需维护用户与客户端交互过程中的状态，因此只编写实现客户端状态管理的代码即可。

2. 解决方案

在本例中，我们采用第7章实现的、基于HTML的树组件作为树型可视组件，如图8-3所示。

RSH是一个灵活的框架，因为你将目睹它的这一特性。首要任务是确定应用中有哪些状态重要得足以让我们觉得应该把它们保存到浏览器的历史记录。我们还决定在示例中同时包含节点展开/收缩的状态变化，在点击一个叶子节点时预览区域图片的改变也包括在内。因此，那些事件的任意一次事件触发，我们都需要添加一个历史记录条目并保存与之相关的足够信息，或者是当用户通过点击浏览器的后退或前进按钮引发调用历史记录时，我们需要恢复应用的对应状态。

在第7章中，我们已经为树组件编写了处理节点展开/收缩操作的函数及改变预览图片的代码。我们将在本示例中定义一个新的辅助函数，以便在那些事件触发时通过调用dhtml-History.add()函数记录当前状态。我们还会在dhtmlHistory对象上注册一个侦听函数，它用来将树组件的状态恢复到前一个记录值。代码清单8-4包含了支持历史记录维护与操作功能的树

组件的全部JavaScript源代码。上一版本的树组件（见代码清单7-8）没有的新加代码以粗体表示。

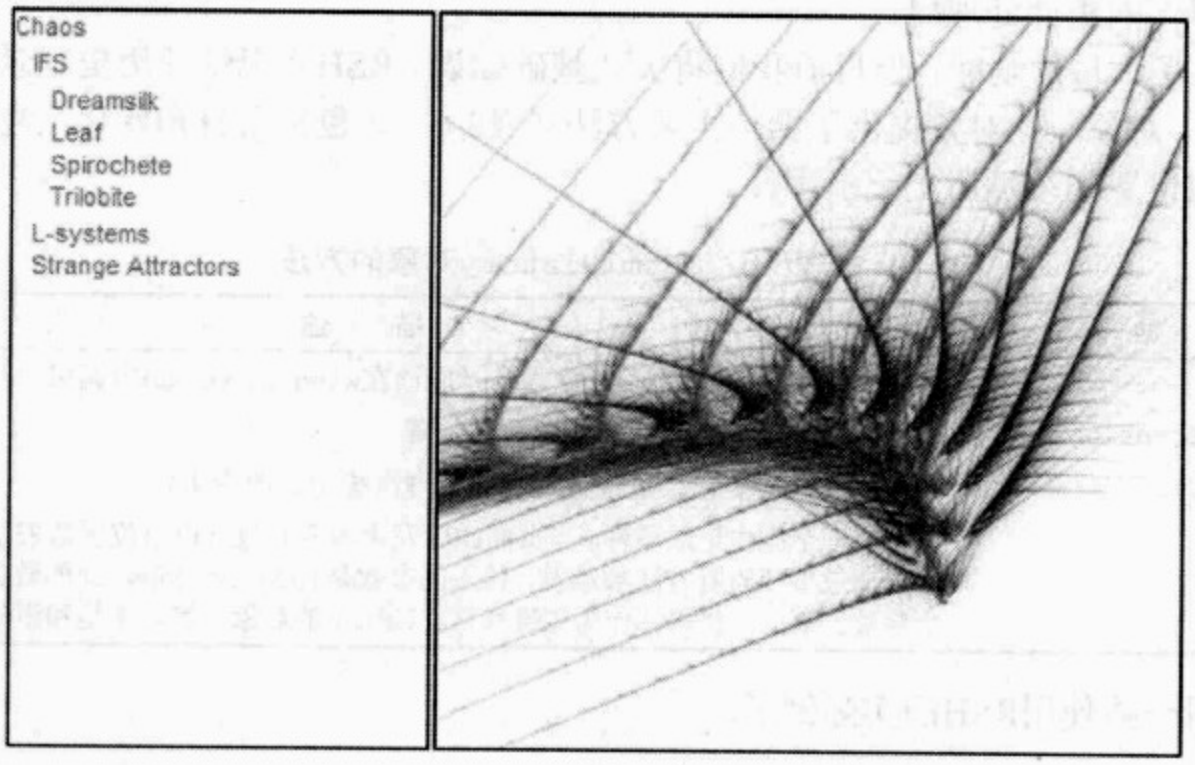


图8-3　第7章的树组件。本章我们为该示例添加后退按钮的支持

代码清单8-4　增加了历史记录的tree.js

```
var historyCount=0;

function initTree(){
var leafNodes=null;
var nonLeafNodes=null;

  $('preview').show();
  var allNodes=$$('.nodeHeader');
  var partitioned=allNodes.partition(
    function(node){
      return node.hasClassName('leaf');
    }
  );
  leafNodes=partitioned[0];
  leafNodes.each(
    function(node){
      var anchor=node.parentNode;
      var anchorParent=anchor.parentNode;
      var imgsrc=anchor.href;
      anchor.removeChild(node);
      anchorParent.replaceChild(node,anchor);
      node.onclick=function(){
        $('preview_img').src=imgsrc;
        addHistory();
      }
    }
  );
```

❶ 注册预览图像改变

```
    nonLeafNodes=partitioned[1];
    nonLeafNodes.each(
      function(node){
        var childDivId=node.id.replace(/head/,"child");
        var childDiv=$(childDivId);
        node.onclick=function(){
          childDiv.toggle();
          addHistory();
        }
        childDiv.hide();
      }
    );
    dhtmlHistory.initialize();
    dhtmlHistory.addListener(
      function(location,data){
        if (data){
          setTreeState(data);
        }
      }
    );
    if (!dhtmlHistory.getCurrentLocation()){
      addHistory();
    }
}
```

❷ 注册节点展开/收缩

❸ 历史记录对象初始化

❹ 添加事件监听函数

❺ 记录最初的历史状态

```
function addHistory(){
  dhtmlHistory.add(
    'history'+historyCount,
    getTreeState()
  );
  historyCount++;
}

function getTreeState(){
  var treeState={
    nodes:{},
    image:$('preview_img').src
  };
  nonLeafNodes.each(
    function(node){
      var childDivId=node.id.replace(/head/,"child");
      var childDiv=$(childDivId);
      var isOpen=childDiv.visible();
      treeState.nodes[childDivId]=isOpen;
    }
  );
  return treeState;
}

function setTreeState(state){
  for (node in state.nodes){
    var nodeDiv=$(node);
```

❻ 记录历史输入

```
        if (nodeDiv){
          if (state.nodes[node]){
            nodeDiv.show();
          }else{
            nodeDiv.hide();
          }
        }
      }
      $('preview_img').src=state.image;
    }
```

我们对应用所做的改动是增加dhtmlHistory对象的生命周期管理，如同表8-6简略描述的那样。首先，我们需要初始化该对象❸。做完这个工作之后，我们立即注册了一个侦听方法❹，它将在用户点击后退或前进按钮时，由dhtmlHistory对象调用。然后我们记录初始状态并将它作为历史记录的第一个项目❺。我们还修改了树组件上的事件处理器，添加了用于响应改变预览图片❶或者展开/收缩树节点❷时的代码。为保持代码的整洁，我们提供了一个可在树组件上任意应用的辅助方法addHistory()❻。

每一个历史记录项目的名字必须唯一。我们在此采用一种较为直接的方式，每次添加历史记录项目时，在一个全局计数器上简单递增。

示例应用的状态由每个节点在树上的状态（即它是打开还是关闭）和显示在图片预览区域的图片组成。我们提供了两个辅助方法getTreeState()和setTreeState()来分别读写状态。这样就保证了和dhtmlHistory对象交互的简单性还有和树组件内部逻辑相分离。

注意，我们在代码清单7-8中通过在href属性上设置纯hash符号关闭了锚节点，如下

```
anchor.href='#';
```

在这里，我们不能这么实现，因为RSH使用URL中的hash时另有目的。因此，我们只好用一些DOM操作的小技巧以实现在不丢失数据的情况下完全移除链接。这有点像旧舞台魔术的表演伎俩，在不伤及盘子或玻璃杯的情况下突然抽离桌布，但没有演砸魔术的危险！

3. 讨论

现在，图片浏览程序应该可以操作程序的状态了。尝试展开并收缩树上的多个节点并浏览这些图片。现在，使用后退和前进按钮来操纵你的历史记录。

我们提供了后退和前进按钮的简单再实现，用户将为此而高兴。RSH框架使得更改此类浏览器行为变得易于实现，它只是通过URL的hash对象及其和每个操作hash相关的一些附加数据来实现状态维护。RSH允许我们按照应用程序的要求制作或简单或复杂的hash信息。

本例向我们展示了如何在客户端维护状态，该状态将在单一会话期间有效。在其他情况下，我们或许想把应用程序状态保存到用户会话或跨机器保存。在这种情况下，我们将需要把状态保存到服务器端的会话对象。我们将在下一节实现这个功能，看看对我们的实现机制做了多少改动。

8.2.5　使用 RSH 在服务端维护应用程序状态

现在，你已经明白如何使用RSH维护客户端状态，让我们来看看怎样利用它在服务器端维护应用程序的状态。

1. 问题

你需要在用户会话间存储状态，并需要把客户端状态存储到服务器。

2. 解决方案

要演示如何使用RSH服务器维护应用程序状态，我们将要求RSH框架仅仅把事件的位置/主键的信息存储并将实际状态的持久化处理转移到服务端实现，它将存储在服务端的会话中。（如果我们想要更健壮的解决方案，我们可以把状态持久化保存到文件或数据库中，但此刻我们希望保持服务器示例的简单性。）因此，我们需要Ajax。当存储历史记录状态时，我们将客户端的状态连同客户端的标识符一起发送到服务器。同样，当一个浏览历史事件通过后退按钮触发时，一个包含位置/主键信息的Ajax请求将发送到服务器，服务器将返回一个数据块描述客户端的浏览历史状态。

首先，来看看客户端所需的改动。代码清单8-5展示了详细代码和代码清单8-4不同的代码以粗体显示。

代码清单8-5 增加了在服务端维护历史记录功能的tree.js

```
var leafNodes=null;
var nonLeafNodes=null;
var historyCount=0;

function initTree(){
$('preview').show();
var allNodes=$$('.nodeHeader');
var partitioned=allNodes.partition(
  function(node){
    return node.hasClassName('leaf');
  }
);
leafNodes=partitioned[0];
leafNodes.each(
  function(node){
    var anchor=node.parentNode;
    var anchorParent=anchor.parentNode;
    var imgsrc=anchor.href;
    anchor.removeChild(node);
    anchorParent.replaceChild(node,anchor);
    node.onclick=function(){
      $('preview_img').src=imgsrc;
      addHistory();
    }
  }
);
nonLeafNodes=partitioned[1];
nonLeafNodes.each(
  function(node){
    var childDivId=node.id.replace(/head/,"child");
    var childDiv=$(childDivId);
    node.onclick=function(){
      childDiv.toggle();
```

```
            addHistory();
          }
          childDiv.hide();
        }
      );

      dhtmlHistory.initialize();
      dhtmlHistory.addListener(
        function(location,key){
          if (key){
            fetchTreeState(key);
          }
        }
      );
      if (!dhtmlHistory.getCurrentLocation()){
        addHistory();
      }
    }

    function addHistory(){
      var data=getTreeState();
      var key='history'+historyCount;
      historyCount++;
      new Ajax.Request(
        "jsp/treeState.jsp",
        {
          method: "post",
          parameters: $H({
            key:key,
            data:JSON.stringify(data)
          }).toQueryString(),
          onComplete:function(response){
            var responseObj=
              JSON.parse(response.responseText);
            if (responseObj &&
              responseObj.status=="ok"){
              dhtmlHistory.add(key,key);
            }
          }
        }
      );
    }

    function getTreeState(){
      var treeState={
        nodes:{},
        image:$('preview_img').src
      };
      nonLeafNodes.each(
        function(node){
          var childDivId=node.id.replace(/head/,"child");
          var childDiv=$(childDivId);
          var isOpen=childDiv.visible();
```

❶ 发送Ajax请求

❷ 把状态按JSON格式进行编码

❸ 解析JSON格式的响应

```
        treeState.nodes[childDivId]=isOpen;
      }
    );
    return treeState;
}

function fetchTreeState(key){
  new Ajax.Request(
    "jsp/treeState.jsp",
    {
      method:"get",
      parameters: $H({ key:key }).toQueryString(),
      onComplete:function(response){
        var responseObj=
          JSON.parse(response.responseText);
        if (responseObj){
          updateTreeState(responseObj);
        }
      }
    }
  );
}

function updateTreeState(state){
  for (node in state.nodes){
    var nodeDiv=$(node);
    if (nodeDiv){
      if (state.nodes[node]){
        nodeDiv.show();
      }else{
        nodeDiv.hide();
      }
    }
  }
  $('preview_img').src=state.image;
}
```

❹ 从服务器获取历史状态

❺ 更新客户端状态

我们使用Ajax（使用Prototype库的Ajax.Request）来记录浏览历史的每个条目❶。在上一个示例中，我们把状态转化成文本型的JavaScript对象，因此发送数据到服务器之前使用JSON对数据进行编码似乎是一个很自然的选择❷。我们使用第2章讨论过的json.js库❸。Ajax请求的响应同样也是JSON格式❹。注意我们还需要在客户端存储历史记录，可以通过上一个示例介绍过的辅助函数addHistory()实现。我们可以在发送Ajax请求的同时做这个操作，但我们选择推迟该操作直到响应从服务器返回并告诉我们历史记录已经被存储。这样的话，我们能确保当我们调用历史记录时有数据可用。

为响应后退按钮重现应用程序的历史状态时，我们需要再次与服务器通讯❹。从服务器获取状态响应之后，我们可将其包含的数据处理成JSON格式并更新客户端以恢复到以前的状态❺。（为了更好地反映其角色，我们已经将setTreeState()函数名改为updateTreeState()，但代码还和原来一样。）

这样一来，我们就可以来回向服务器发送数据，但当数据到达服务器时我们该怎么做呢？代码清单8-6显示的简单JSP页面用于将我们的历史记录数据存储到用户会话中。

代码清单8-6 treeState.jsp

```
<jsp:directive.page
  contentType="text/json"
  import="java.util.*"
/>
<%
String key=request.getParameter("key");
String data=request.getParameter("data");
if (data==null){
  %><%=session.getAttribute(key)%><%
}else{
  session.setAttribute(key,data);
  %>{ "status" : "ok" }<%
}
%>
```

我们承诺保持服务器端的代码简单，而且确实做到了这一点！在服务器端，对存储中的历史记录状态我们不再做任何操作，这样的话我们就不用为对收到的数据进行JSON格式解码而烦恼，只需要在会话中以字符串存储它们即可。就像在本例开头指出的那样，我们因为简单而选用会话。如果希望用户能够跨会话或跨机器获取他们的历史记录状态，我们就需要采用文件或数据库作为更持久的存储介质。

3. 讨论

该示例在运行时，应该可以表现出和上一个客户端示例相同的行为，除了状态数据的持久化机制和状态重现逻辑目前在服务器实现。该技术的可能用法不胜枚举，它为你提供了又一个强大的工具。

和客户端状态持久化相比，在服务器端实现状态持久化有好多优点，也有不少缺点。优点之一是用户现在不必再束缚在本地浏览器。他们可以在离开应用程序的某个时刻后，从任意一个客户端登录然后恢复之前会话过程中所有可用的历史状态。如果该应用程序某个地方除了错误或它的表现有些异常，该应用程序故障修复工作现在变简单了。开发人员可以审查当前的用户历史记录显示状态与期望行为的细微差别。用户会话记录可以后退，对用户行为模式进行更广泛的分析，目前是可行的。

当然，缺点之一就是为了给用户提供这个功能，需要在服务器端编写很多代码并多费一些脑筋。因为我们现在开始创建客户端到服务器端的数据往返传输，响应时间和带宽的问题显露出来了，到服务器的缓慢链接可能给试图在应用中前后导航的用户的操作体验带来负面影响。从用户隐私角度出发，如果我们着手记录如此大量的用户行为模式信息，我们将陷入隐私保密的泥沼不能自拔。我们的大量精力将被牵涉到如何保护这些已收集的用户数据。

和处理应用程序状态及历史记录很接近的一个主题是实现某种撤销机制。我们将在下一节研究如何给一个应用程序添加撤销功能。

8.3 处理撤销操作

浏览器提供了一些支持撤销/恢复（undo/redo）操作的功能，但它们在很大程度上限于表单编辑。如果你希望允许用户使用普通富DHTML控件支持撤销/恢复操作，你应当自己实现这些功能。它并非最初看起来的那么困难。实现一个基本的撤销/恢复系统需要做四件事情：

- 记录用户操作，在这些操作发生时将它们放入一个类似栈的数据结构中。
- 提供遍历栈元素的功能，包括前进和后退（对该功能而言，我们的撤销系统不是一个真正的栈，因为用户操作从栈中弹出之后还会继续保留，以便用户恢复该操作）。
- 为用户提供恢复他们过去操作的能力。
- 维护撤销栈的当前位置。

实现一个撤销栈，有两件重要的事情值得注意。首先，对每一个存储在栈中的操作，你需要追踪记录该操作的前后状态。例如发生了一个可撤销的操作，在一个文本字段内有字符输入，在该文本字段中新输入字符之前的字符串和新输入的字符都应该被存储到栈。这样的话，可以后向遍历栈（撤销）同样也可以前向遍历（恢复）栈。其次，如果一个操作被加入到栈中间的某个位置，插入操作插入位置之后的所有操作将被删除。例如：

(1) 一个用户完成了10个可撤销的操作。

(2) 之后用户撤销了其中的5个操作。

(3) 然后用户执行了另一个可撤销操作[①]。

(4) 目前撤销栈中包含6个操作，原有的1-5个操作被保留，位于6号位置的新操作同样也被保留。原有的6-10号操作不再有效，并因此被删除了。

8.3.1 何时提供可撤销功能

判断何时为应用程序提供撤销功能取决于很多因素，其中包括应用类型、期望的用户群体以及机器性能。通常，仅在用户操作引发的数据改变规模小且可管理的情况下才应该提供撤销功能。应尽量避免为复杂的操作实现撤销，它们不容易实现，因为这将伴随在客户端和服务器之间往返传递大量的数据。

8.3.2 实现一个可撤销/恢复操作栈

就本例而言，我们将创建一个撤销栈（undoStack）对象。作为最主要的全局变量，它将包含一个索引和一个以数组表示的操作栈。我们将提供一个事件处理器来检测Ctrl+Z和Ctrl+Y快捷键何时被按下，此外还提供撤销和恢复按钮或其他UI机制以操作该栈。我们刚才描述的栈对象将负责实现可撤销和恢复栈的基本功能。由于撤销或恢复用户操作发生时，程序如何工作取决于应用的种类，所以我们还没有谈及这个主题。我们从上一个使用RSH库的示例获得启示，本例将提供回调机制以支持用户自定义应用程序状态如何响应撤销和恢复操作。那么来看看这个可撤销栈是如何实现的。

① 这里指的是执行一个新的可撤销操作。

1. 问题

你需要为Ajax应用提供一种浏览器未必支持的用户操作撤销/恢复机制，或者你需要将浏览器的撤销/恢复功能替换成你自己定制的版本。

2. 解决方案

我们将从一个简单示例开始。为拥有几个文本输入字段的该示例添加可撤销栈。实现过程有两个要点。首先，我们要实现前面讨论过的一般性的撤销栈。其次，我们需要把文本输入过程和撤销栈进行挂钩。让我们看看撤销栈对象本身，代码清单8-7显示了全部代码。

代码清单8-7　undo.js

```
var undoStack={
    curIdx: 0,
    stack: [],
    undoHandler: null,
    doDiffCheck: true,
    actionPerformed:false,
    init: function(theUndoHandler){                    ❶ 栈初始化
        Event.observe(document, 'keydown',
          this.checkUndo);
        Event.observe(document, 'keypress',
          this.postCheckUndo);
        this.undoHandler=theUndoHandler;
    },
    checkUndo: function(event){                        ❷ 处理撤销按键
        var keyCode = event.which || event.keyCode;
        if((keyCode ==  90) && event.ctrlKey){
        undoStack.undo();
        undoStack.actionPerformed=true;
        Event.stop(event);
        return false;
    }else if((keyCode ==  89) && event.ctrlKey){
        undoStack.redo();
        undoStack.actionPerformed=true;
        Event.stop(event);
        return false;
    }
},
postCheckUndo: function(event){
    if(undoStack.actionPerformed){
        undoStack.actionPerformed=false;
        Event.stop(event);
        return false;
    }
},
seek: function(index){
    if(index >= 0 &&
      index < this.stack.length){
        this.curIdx=index;
    }
},
```

```
add: function(theType,theValue,
  ignoreDiffCheck){
    var action =this
      .getNewUndoAction(theType,theValue);
    var success=false;
    var differs=(
      this.doDiffCheck && !ignoreDiffCheck)?
        (this.checkAction(action)):
        true;
    if(differs){
        this.stack[this.curIdx++]=action;
        this.stack.length=this.curIdx;
        success=true;
    }
    var stateAction=this.getNewUndoAction();
    stateAction.canUndo=true;
    stateAction.canRedo=false;
    this.undoHandler(stateAction);
    return success;
},
checkAction: function(action){
    var latest = this.stack[this.stack.length-1];
    return (latest) ?
        (!action.equals(latest)) :
        true;
},
undo: function(){
    var action=null;
    if(this.curIdx > 0){
        action=this.stack[--this.curIdx];
    }else{
        action=this.getNewUndoAction();
    }
    action.canUndo=this.curIdx > 0;
    action.canRedo=this.curIdx < this.stack.length;
    this.undoHandler(action,true);
},
redo: function(){
    var action=null;
    if(this.curIdx < this.stack.length){
        action=this.stack[this.curIdx++];
    }else{
        action=this.getNewUndoAction();
    }
    action.canUndo=this.curIdx > 0;
    action.canRedo=this.curIdx < this.stack.length;
    this.undoHandler(action,false);
},
getNewUndoAction: function(theType,theValue){
    return {
        type:   theType,
        value:  theValue,
```

❸ 为栈添加动作

❹ 指定撤销动作

❺ 指定恢复动作

❻ 创建撤销动作对象

```
            canUndo:false,
            canRedo:false,
            equals: function(action){
              return (
                JSON.stringify(this.type)  ==
                  JSON.stringify(action.type) &&
                JSON.stringify(this.value) ==
                  JSON.stringify(action.value)
              );
            }
          };
      },
  };
```

首先，我们通过在文档（document）对象上注册键盘事件处理器进行栈初始化❶操作。和本章上一个示例相同，我们需要把keypress事件绑定到指定的处理器上，同时也绑定keydown事件，以便在Mozilla浏览器中终止事件传播。主要的按键处理函数是checkUndo()❷，它能在所有的浏览器中触发，并检测Ctrl+Z和Ctrl+Y组合键的发生。

init()函数还需要一个回调函数引用，就是该函数最初传入的参数。一旦init()函数初始化完毕，我们就可以向撤销栈❸添加项目了。在添加项目时，我们提供了一个默认选项参数来过滤恢复操作。在多数情况下，如果新事件和前一个事件相同，我们就不会再次把它添加到栈，尽管这与我们将要在8.3.3节看到的同样都是异常情况。

当撤销❹或恢复❺事件发生时，我们调用该回调函数，在该函数中传入了一个简单的撤销操作对象，它包含应用程序的状态信息并指定了执行撤销还是恢复操作。undo()和redo()两者还各自执行了同样的索引检测代码以确认栈中是否存在前驱或后继操作。如果栈为空，将通过getNewAction()返回一个空操作对象，表示不存在可能的操作。

接下来，要查看最后一个注意点是操作对象本身❻的结构。可撤销操作对象封装了执行撤销或恢复操作的必要数据，同时还包含两个布尔型变量，它们对确定该操作是否拥有可撤销或恢复操作很有用。它还提供了一个函数用于评估对象本身和另一个对象是否等价，add()函数利用该函数来确认栈中的两个相邻项目不重复同一状态。

既然我们已经实现了该撤销栈，我们来创建一个简单的撤销处理函数以使页面上的文本输入字段实现输入可撤销。在该页面的HTML代码中，我们定义了两个文本输入字段，其id分别是foo和baz。代码清单8-8显示的是监测文本输入操作并添加到撤销栈所必须的JavaScript代码。

代码清单8-8　撤销处理器示例

```
window.onload = function(){
    addTextEventListener('foo');
    addTextEventListener('baz');
    undoStack.init(demoHandler);
}

function addTextEventListener(element){
    Event.observe(
        element,
        'keydown',
```

❶ 启用文本输入功能

❷ 捕获keydown事件

```
            startTextUndo
        );
        Event.observe(
            element,
            'keyup',
            endTextUndo
        );
    }
function startTextUndo(event){
    var target   =Event.element(event);
    var keyCode  =event.which || event.keyCode;
    if((keyCode != 16 && keyCode != 17 && keyCode != 18) &&
     !((keyCode ==  90) && event.ctrlKey) &&
     !((keyCode ==  89) && event.ctrlKey)){
        target.beforeUndoVal=target.value;
    }
}

function endTextUndo(event){
    var target   =Event.element(event);
    var keyCode  =event.which || event.keyCode;

    if((keyCode != 16 && keyCode != 17 && keyCode != 18) &&
      !((keyCode ==  90) && event.ctrlKey) &&
      !((keyCode ==  89) && event.ctrlKey)){
        undoStack.add(
            'TEXT',
            {   elementRef:target.id,
                prevValue :target.beforeUndoVal,
                newValue  :target.value
            }
        );
    }
}

function demoHandler(action,undo){
    if(action){
        if(action.type == 'TEXT'){
            var el=$(action.value.elementRef);
            el.value=(undo) ?
              action.value.prevValue :
              action.value.newValue;
        }

        $('undoButton').disabled=!action.canUndo;
        $('redoButton').disabled=!action.canRedo;
    }}
```

❸ 捕获keypress事件

❹ 更新上一次文本值

❺ 为栈添加动作

❻ 更新输入文本

❼ 更新撤销按钮的状态

首先，通过将keydown❷和keyup❸事件处理函数绑定到它们自身，我们为该页面的两个文本输入框❶都启用了文本输入监测功能。我们需要捕捉两个文本框内的输入，因此我们可以记录keydown❹之前的键盘输入值，然后一旦该次输入按键❺再次弹出就把当前这次操作添加到撤销

栈中。注意，我们过滤掉了某些按键——Shift、Ctrl和Alt——还有Ctrl+Z以及Ctrl+Y组合键，因为我们不想把这些键盘敲击事件添加到撤销栈中。

撤销操作包含一个操作类型的标识符——在该示例中我们使用了TEXT作为标签——而value对象必须包含执行该操作所需的全部状态。在该例中，我们记录了文本输入框的当前及之前内容，还有一个用于更新状态到该文本输入框元素的引用。我们的undo()处理函数方法将会传入撤销对象。一旦我们建立了正确的操作类型，就可以依据是撤销还是恢复操作来设置相应的文本输入框的内容❻，然后我们设置界面上的撤销和恢复按钮的相应状态(恢复和禁止)❼。

3. 讨论

这仅是撤销栈的一般实现思路。尽管它重新实现了浏览器的撤销功能，但没有增加新特性。定制每个撤销操作都需要定义一个操作处理函数来负责设置及取消数据。我们将在下一节研究如何实现常见操作的撤销。

从用户体验的角度上讲，为你的Web应用程序提供撤销功能可带来真正的价值。它有助于在常见的非浏览器应用程序和Web应用程序之间搭建起一座桥梁，跨越因操作体验差距在两者间形成的鸿沟。用户已经习惯了与操作系统固有应用程序和瘦客户端应用交互过程中学到的众多术语，如果在你的应用中重用和利用这些概念，将会降低用户掌握此类应用的学习曲线。的确，一些Web应用程序已经是实现了撤销功能，但通常它们限制在服务端实现。将该功能推到客户端将使Web应用程序获得更快的执行速度。这种服务器到客户端的功能转移将减少用户在使用Web应用程序时的挫折感。

通过前面的几个示例，我们演练了如何在客户端和服务器两端维护浏览器的历史记录。当然，我们同样可以将撤销功能应用到客户端和服务器。下一个示例，我们将检查如何在服务端实现处理撤销功能。

8.3.3　扩展撤销栈以支持更复杂的用户操作

现在，我们已经看到了一个实现撤销栈的简单示例，我们将再接再厉来处理更加复杂的例子。让我们往下进行吧，同时在服务器端引入Ajax和撤销功能。

1. 问题

你需要为用户操作定制撤销功能，这远比简单文本输入复杂。和客户端一样，重设应用程序的状态将需要服务器的输入。

2. 解决方案

我们在本例将重用之前实现的基本撤销栈对象，并为一个简单的图片编辑处理应用提供撤销功能。图片操作处理将在服务器发生，服务器把前一个状态下的图片快照、操作对象的引用和其他相关操作数据存储。图片编辑器的用户界面如图8-4所示。当前状态下的图片在页面中间，左侧是可用的操作列表，右侧是撤销堆栈对象的形象显示。

首先我们需要定义图片编辑器应用中的可用操作类型。这些操作将采用唯一整型值作为标识并将被服务端的代码引用。接下来要展示的例子的可用的操作列表如下所示：

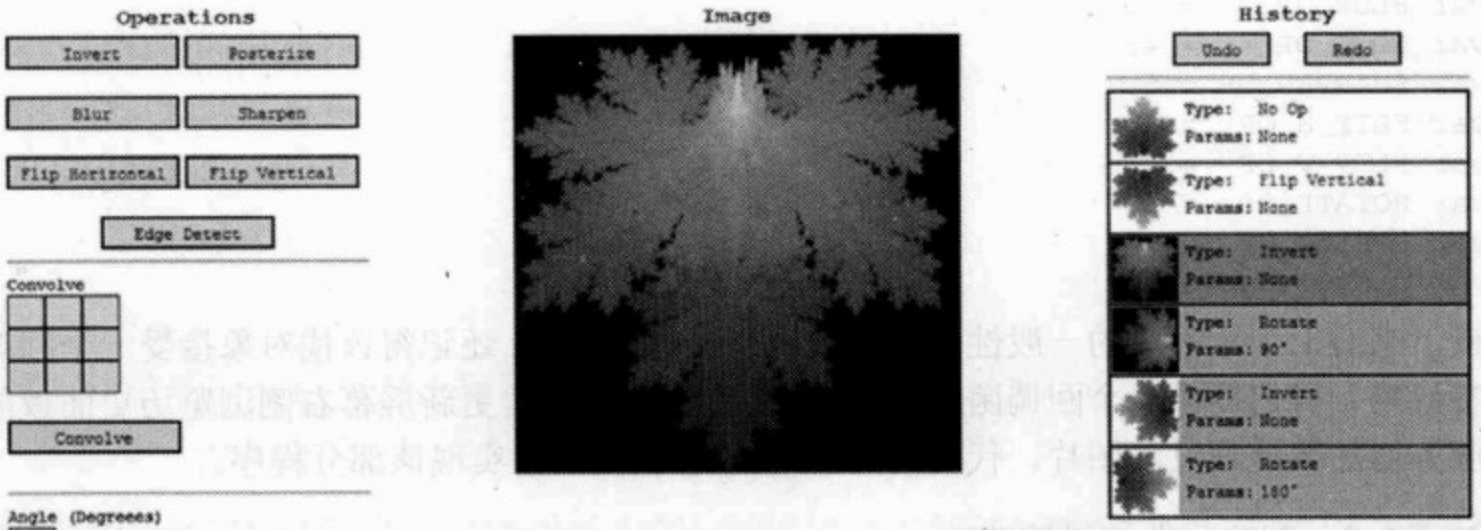


图8-4 一个功能完整的图像编辑器，包含一组图像复制工具、图像预览工具，还有一个可视化的撤销栈展现器

- NO_OP（无操作）——用于初始状态。
- INVERT——对图像的颜色值采取反色处理[①]。
- POSTER （色调分离）——减少图像中的色阶（color level）数值。
- BLUR——使用3x3卷积矩阵进行图像模糊[②]。
- EDGE——采用高强度对比的图像区域检测。
- SHARPEN——减弱图像的模糊度。
- FLIP_H （水平翻转）——按Y轴翻转。
- FLIP_V （垂直翻转）——按X轴翻转。
- ROTATE——图片旋转（以度数）。
- STRING——在指定的坐标处添加文本字符串。
- CONVOLVE——允许用自定义的3x3卷积矩阵对图像进行卷积处理。

本示例需要编写大量客户端代码，因此我们先来分析介绍这部分内容。首先定义图片编辑处理操作的常量值，如下所示：

```
var NO_OP       = 0;
var INVERT_OP   = 1;
var POSTER_OP   = 2;
```

① 即变为互补色，如白变黑。——译者注

② 即图像锐化。——译者注

```
var BLUR_OP     = 3;
var EDGE_OP     = 4;
var SHARPEN_OP  = 5;
var FLIP_H_OP   = 6;
var FLIP_V_OP   = 7;
var ROTATE_OP   = 8;
var STRING_OP   = 9;
var CONVOLVE_OP=10;
```

接下来对上个示例中的一般性的撤销栈对象进行初始化。还记得该栈对象接受一个回调函数作为参数吗？我们提供一个回调函数作为参数，该函数将负责更新屏幕右侧浏览历史面板的数据和从服务器加载处理过的图片。代码清单8-9显示了我们如何实现该部分程序。

代码清单8-9　创建复杂的撤销队列

```
window.onload = initialize;

function initialize() {
    undoStack.init(imageEditorHandler);
    undoStack.doPrevActionCheck=false;
    new Ajax.Request(
      'ImageEditor?timestamp='+
        (new Date().getTime())+'&init=yes'
        +'&src='+encodeURI($('editingImage').src),
      { method: 'get',
        onSuccess: addActionCallback
      }
    );}

function imageEditorHandler(action,undo) {
    if (action != null) {
        if (action.type != null) {
            var table = $('historyPalette');
            var selected=-1;

            for (var i = 0; i < table.rows.length; i++) {
                if(table.rows[i].selected) {
                    selected = i;
                    break;
                }
            }

            if (selected > -1) {
                if(undo){
                    table.rows[selected]
                      .className ='paletteDisabled';
                    table.rows[selected].selected  =false;
                    table.rows[selected].dimmed    =true;

                }else{
                    // set the next action
                    table.rows[selected]
                      .className ='paletteNormal';
                    table.rows[selected].selected  =false;
```

❶ 创建初始化函数

❷ 撤销栈初始化

❸ 忽略动作等价性检查

❹ 服务器端撤销栈初始化

❺ 定义图片编辑器的撤销函数

❻ 处理历史动作面板状态

```
                table.rows[selected].dimmed    =false;
            }

            var new_i=(undo) ? selected-1 : selected+1;
            table.rows[new_i].className='paletteSelected';
            table.rows[new_i].selected =true;
            table.rows[new_i].dimmed   =false;

            undoStack.seek(new_i);
            new Ajax.Request(
              'ImageEditor?timestamp='+
                (new Date().getTime())+'&seek='+(new_i),
              { method: 'get',
                onSuccess: seekCallback
              }
            );
        }
    }

    $('undoButton').disabled=!action.canUndo;
    $('redoButton').disabled=!action.canRedo;
  }
}
function seekCallback(xhr){
  $('editingImage').src=xhr.responseText;
}
```

❼ 把任意可撤销的动作恢复原状

❽ 定义Ajax回调方法

代码清单8-9新添加的initialize()函数❶需要实现几个功能。首先，我们必须初始化客户端的撤销栈❷并传入自定义的回调处理函数引用。其次，我们需要通知栈对象忽略添加新撤销操作对象执行等价性检查[①]❸。该设置允许同一类型的图片操作同时执行两次——例如，多次对图片进行模糊处理。最后，我们需要在服务端初始化撤销栈并设置应用程序的初始状态❹。

图片编辑处理函数❺比前一个示例略显复杂。当一个实际操作执行时，我们不仅需要处理撤销/恢复，还不得不处理历史记录面板的状态❻。此外，由于初步介绍了浏览历史面板，我们需要在没有遍历新选中的操作和前一个选中操作之间的情况下，能够激活任意条目❼。

当转到一个新条目时，我们希望执行一个Ajax请求，使服务器更新服务端的撤销栈状态。我们定义了一个用于响应Ajax请求的回调函数，该函数简单地更新主要图片资源❽。尽管这是简单的回调函数，通常可以将它内嵌到其他函数主体，但我们还是在此单独定义了一个函数以便在其他地方重用。

代码中还有另一个预定义的回调函数，在初始化撤销栈函数内发起Ajax请求过程中我们用到了它。我们将简要介绍addActionCallback()函数，它更棘手。

我们继续前进，然后介绍表单左侧的控件支持的几个操作背后的JavaScript代码。所有这些操作遵循类似的操作流程，散列化任意必需的参数，然后发起一个Ajax请求到服务器，由服务器执行图片操作。创建Ajax.Request的函数addAction()，如代码清单8-10所示。

① 与栈中前一个操作对象做等价性检测。——译者注

代码清单8-10　addAction()函数

```
function addAction(op){
    var action     =null;
    var paramString='';

    switch(op){
        case ROTATE_OP:{
            action={
            rotAngle: $F('rotAngle')
        };
        paramString='&params='+action.rotAngle;
        break;
    }
    case STRING_OP:{
        action={
            string: $F('string'),
            color:  $F('color'),
            locX:   $F('locX'),
            locY:   $F('locY')
        };
        paramString=
            '&params='+
            encodeURI(
            action.string.replace(/,/g,'%%')+','+
            action.color+','+
            action.locX+','+
            action.locY);
        break;
    }
    case CONVOLVE_OP:{
        action=[];
        for(var i=0;i<9;i++){
            action[i]=$F(
                'c'+
                Math.floor(i/3)+
                '_'+
                (i%3)
            );
        }
        paramString='&params='+encodeURI(action);
    }
    case INVERT_OP:
    case POSTER_OP:
    case BLUR_OP:
    case EDGE_OP:
    case SHARPEN_OP:
    case FLIP_H_OP:
    case FLIP_V_OP:
    case NO_OP:
    default:{break;}
}

undoStack.add(op,action);
```

```
new Ajax.Request(
  'ImageEditor?timestamp='+
    (new Date().getTime())+'&action='+
    op+paramString,
  { method: 'get',
      onSuccess: addActionCallback
    }
  );
}
```

该函数的第一部分是一个case语句，它根据给定的操作类型提取所有相关的参数并拼装到一个叫做paramString的变量。注意有些操作不需要任何参数，例如invert和flip等。其他操作则相反，诸如convolve和rotate等操作需要若干参数。装配完参数，我们更新客户端撤销栈，然后发起一个请求到服务器，连同操作类型和所有参数都传入到该请求中。这将更新服务端的撤销栈。

此外，该函数中的Ajax请求用到了addActionCallback()函数的引用。我们在代码清单8-9初始化撤销栈时看到过该函数引用。来看看该函数的实现（代码清单8-11）。

代码清单8-11 addActionCallback()函数

```
function addActionCallback(xhr){
    var table   =$('historyPalette');
    var content =xhr.responseText.split('<!-- BREAK -->');
    var found   =false;
    var toDelete=new Array();

    var editingImgSrc   =content[0];
    var newRowProperties=JSON.parse(content[1]);
    var eventCalls      =content[2];

    $('editingImage').src=editingImgSrc;

    for(var i=0;i<table.rows.length;i++){
        if(newRowProperties.id == table.rows[i].id){
            found=true;
        }

        if(found){
            toDelete[toDelete.length]=i;
        }else{
            table.rows[i].className='paletteNormal';
            table.rows[i].dimmed   =false;
            table.rows[i].selected =false;
        }
    }

    for(var i=toDelete.length-1;i>=0;i--){
        table.deleteRow(toDelete[i]);
    }
    var newRow     =table.insertRow(table.rows.length);
    var imageCell  =newRow.insertCell(0);
    var contentCell=newRow.insertCell(1);
```

```
        newRow.id          =newRowProperties.id;
        newRow.className='paletteSelected';
        newRow.selected =true;
        newRow.dimmed    =false;

        imageCell.className=newRowProperties.paletteImageClass;
        imageCell.innerHTML=newRowProperties.paletteImageHTML;

        contentCell.className=newRowProperties.paletteContentClass;
        contentCell.innerHTML=newRowProperties.paletteContentHTML;

        eval(eventCalls);
    }
```

此处我们定义了用于从服务器返回响应的一个自定义格式，它采用一个分割定界字符串[①]将响应分割为三部分：预览图片的URL、要在浏览历史面板显示的新项目的CSS属性定义和要调用的一组JavaScript事件。我们把响应分割为上述三部分，然后更新相应的预览图片和浏览历史面板。

此刻我们已经规定了客户端撤销栈以及它与服务端栈通信的全部工作过程。余下的值得关心的JavaScript代码只是为浏览历史面板元素添加一些行为，如代码清单8-12所示。

代码清单8-12　为浏览历史面板添加行为

```
function historyPaletteMouseOver(event) {
    var tr = findTarget(Event.element(event));
    var id = tr.id;
    tr.className = 'paletteHighlight';
}

function historyPaletteMouseOut(event) {
    var tr = findTarget(Event.element(event));
    var id = tr.id;
    if (tr.dimmed) {
        tr.className = 'paletteDisabled';
    } else if (tr.selected) {
        tr.className = 'paletteSelected';
    } else {
        tr.className = 'paletteNormal';
    }
}
function historyPaletteMouseDown(event) {
    var tr = findTarget(Event.element(event));
    var id = tr.id;
    tr.className = 'paletteSelected';
}

function historyPaletteMouseClick(event) {
    var table  = $('historyPalette');
    var tr     =findTarget(Event.element(event));
    var id     =tr.id;
```

❶ 把触发mouseover事件的历史面板记录高亮显示

❷ 把触发mouseout事件的历史面板记录高亮显示移除

❸ 把触发mousedown事件的历史面板记录选中

❹ 把选中记录的后续动作记录状态设置为禁用

① 即代码清单中的字符串<!-- BREAK -->。——译者注

```
var disable=false;
for(var i = 0; i < table.rows.length; i++) {
    var className = 'paletteNormal';
    table.rows[i].dimmed   = false;
    table.rows[i].selected = false;
    if (disable) {
        table.rows[i].dimmed = true;
        className = 'paletteDisabled';
    } else if (tr.id == table.rows[i].id) {
        table.rows[i].selected = true;
        className = 'paletteSelected';
        disable = true;
    }

    table.rows[i].className = className;
}

var idx = parseInt(id.replace(/action_/,''));

undoStack.seek(idx);
seekRequest=sendGETRequest(
    '/servlet/ImageEditor?timestamp='+
    (new Date().getTime())+'&seek='+
    idx,seekCallback
);
}
function findTarget(element) {
    var parent = null;

    while (parent == null) {
        if (element.id.indexOf('action_') == 0) {
            parent = element;
        } else {
            element = element.parentNode;
        }
    }

    return parent;
}
```

❺ 激活指定的撤销

浏览历史面板需要几个事件处理其观感（look and feel）以及激活指定的撤销操作❺。我们需要以下事件处理函数：

- onMouseOver 高亮显示历史记录面板的项目❶。
- onMouseOut 取消高亮历史记录面板的项目❷。
- onMouseDown 选择历史记录面板的项目❸。
- onClick 选择历史记录面板的项目，激活撤销栈并禁止后面的项目❹。

3. 讨论

本例提供了功能更加丰富的撤销栈。可以利用它创建你自己的操作处理函数为用户刚完成的操作提供撤销功能。现在，你拥有了一个附带历史记录面板的撤销/重复的基础框架。

在服务器维护一个撤销数据，你遇到的新问题是垃圾回收。什么时候删除用户的撤销信息是安全的？基于你要提供的应用程序有何优点以及你希望向用户提供哪些功能，你必须对上述问题做出决断。方案之一是当用户会话过期就简单地将撤销信息设置为失效。实现起来也不困难，因为很多Web应用框架提供回调函数在会话过期时调用。当然，本例中将不允许用户在某个浏览器停用撤销栈而在另一个浏览器恢复该功能。如果你希望为用户提供此类功能，那么必须在服务器创建工作流机制并关联到用户。一旦这一切准备就绪，用户可以在他们停止工作流机制的任意位置将其恢复。如果你打算实现一个面向工作流的撤销栈，那么你应该仔细斟酌何时将用户的撤销栈清除并将撤销栈的状态告知用户。

关于这一点，你还需要判断你愿意维护多少历史记录。在我们这个图片编辑处理程序中，如果不能限制用户撤销栈的规模，用户就可能滥用服务器服务并耗尽存储空间。只需简单地通过申请传输大量图片（或许通过一段自动化脚本实现），它们就可以通过创建大量撤销栈将服务器资源耗尽。

8.4　总结

本章介绍了怎样实现剥夺用户访问浏览器导航工具栏和页面刷新的权利。我们还介绍了通过使用Hash对象和Really Simple History框架，如何使用浏览器的历史记录和页面刷新功能。如果只是禁止用户使用这些功能相当简单，但这样的解决方案最终也会让用户放弃使用你的应用程序。更明智的解决方案是支持这些功能并重新实现客户端历史记录导航方案，以便为最终用户提供一种既丰富又有用的操作体验。

我们还实现了一个简单的撤销栈方案以帮助开发者创建用户可撤销的所有操作，而不仅仅是一味前进。这为你的应用增加了灵活性，同时也为用户提供后退功能以便出错时或改变主意时使用该功能。

第9章 拖 放 9

本章内容

- 拖放的基础知识
- 拖放列表[①]
- 使用ICEfaces框架实现拖放

我们都熟悉桌面应用程序中的“拖放”，事实上，拖放是让人机交互过程变得真实可见的关键因素（“我的资料不仅仅是一串二进制的0、1——我拥有的是可触摸、可移动的文件和目录。它们是真实的”）。资源浏览器大概是人们最熟悉的支持拖放的应用程序。它呈献给用户的是带图标的文件和目录，用户可以打开目录并在一个窗口中显示内容——作为开发人员，我们最感兴趣的是——可以从某个位置拖动图标（通过按住鼠标左键并移动鼠标）并放到另一个位置（通过松开鼠标键）。这是一种表达“对这个和那个组件做某种操作”的重要方式。其他的用户界面技术（例如点击按钮或选择菜单）通常只能帮你实现“对这个组件做某种操作”，如图9-1所示。

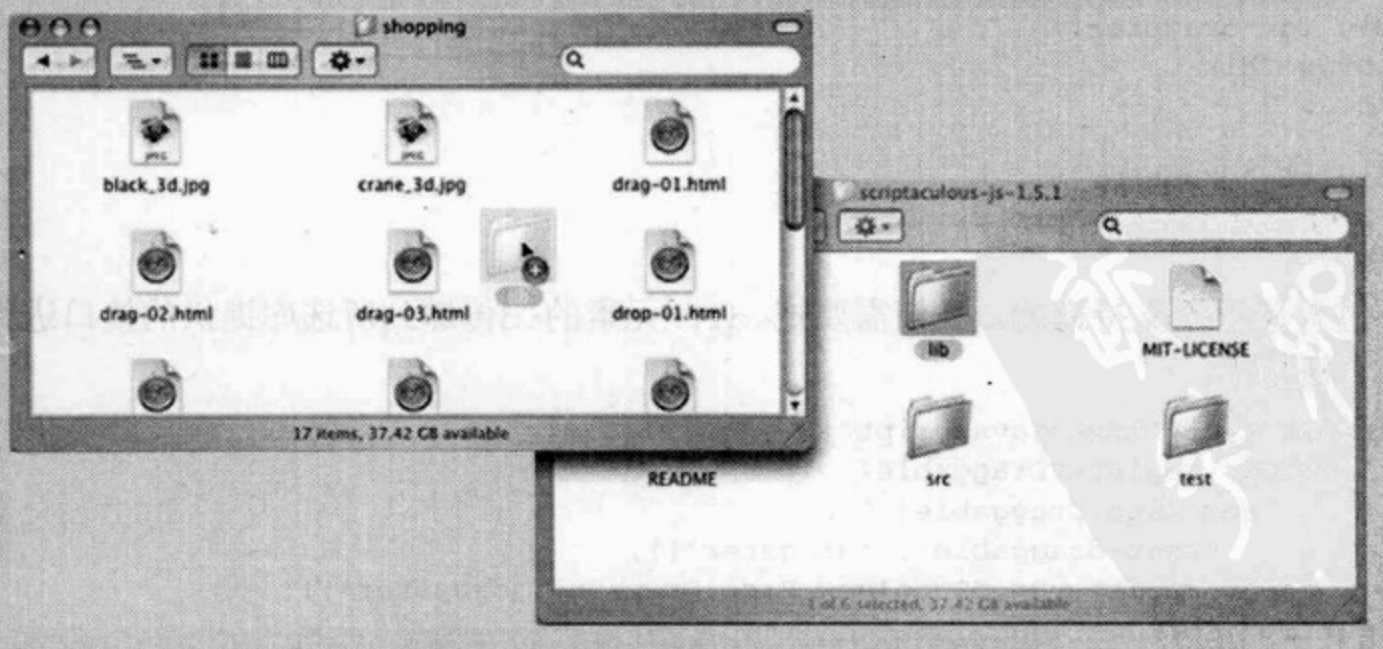


图9-1 桌面应用程序中的拖放

① 原文是drag and drop lists，此处译为支持拖放操作的列表，本章将通过一个示例介绍它。

我们刚才根据原始鼠标事件对拖放进行了解释。不过，对桌面应用开发人员而言，拖放以何种方式出现并不重要。它通常以一种数据转移操作形式出现，是对拖放事件以及数据在对象间移动等概念的抽象。也就是说，拖放通常和拷贝粘贴紧紧结合在一起。在本章中，我们不会偏离拖放主题而扯太远，但我们也会发现，为了舒适地创建一些非常有趣的应用，拖动和停放事件已被充分抽象化。

关于拖放，有个事实令人兴奋，即尽管早在桌面图形界面产生初期拖放就存在了，但大量应用程序仍然没有用好它。换句话说，本章将要学习的技术，为我们构建真正超越同类桌面应用的Ajax应用提供了条件。如你所料，要让浏览器应用支持拖放，离不开JavaScript语言。幸运的是，有许多好心人已经实现了这类JavaScript库。让我们继续前进并了解有哪些支持拖放的JavaScript框架可以利用。

9.1　支持拖放的 JavaScript 框架

很多JavaScript开源框架都实现了拖放，但更重要的是选择一个维护积极的库（已被大家确认能在所有主流的浏览器中可靠运行），并拥有Ajax应用易用的API。其中两个比较流行的框架是Rico（http://openrico.org）和Scriptaculous （http://script.aculo.us/）。

Rico可用于创建富因特网应用，而且它提供了完整的Ajax支持、拖放管理和一个炫目的效果库（effects library）。Scriptaculous同样提供了耀眼的视觉效果和拖放API。注意，我们无法实现在其他程序和浏览器之间进行对象拖放，所有可拖放对象都被严格限制在浏览器窗口内活动，如同这些强大的库无法脱离浏览器环境运行一样。

这两个库的API的用法极为相似。二者都采用简单的HTML 元素充当拖动和停放对象。我们仅需在HTML代码中编写如下片段：

```
<div id="dragster">
  Drag This
</div>

<div id="dropster">
  Drop Something Here
</div>
```

不过对象还不支持拖动。我们需要把 元素的id传递到所选库提供的接口进行注册。对Rico，大概如下：

```
<script type="text/javascript">
    dndMgr.registerDraggable(
        new Rico.Draggable(
          "test-draggable", "dragster"));
    dndMgr.registerDropZone(new Rico.Dropzone("dropster"));
</script>
```

而使用Scriptaculous，写法如下：

```
<script type="text/javascript">
    new Draggable("dragster")
    Droppables.add("dropster");
</script>
```

显然，这两种库都比较易用。由于代码可以毫不费力地从其中一个库迁移到另一个库，且在浏览器兼容方面Scriptaculous库表现略好，因此本章示例将采用后者。接下来，我们将了解如何在Ajax应用中把已创建的对象关联到拖动对象（能被拖动的对象）和停放对象（允许拖动对象停放的区域）。

9.2 Ajax 应用中的拖放

显然，我们不打算从头实现支持拖放功能的JavaScript库。通过利用现有的库，例如Scriptaculous，我们就能确保应用中的拖放不仅工作正常而且还可以在多种浏览器之间移植。对我们来说，困难已经简单地变成如何把拖放融入Ajax应用。Ajax出现之前，我们能做的就是为用户提供一些花哨的功能：用户可以在浏览器内四处拖动某个对象并为这种新奇的功能感到愉悦。但用户绝对没有能力让其他用户分享自己的操作结果，或者改变来自服务器的数据。为了使应用更加真实，需要将拖放事件和到服务器的Ajax调用进行关联。来看一下如何为一个使用Scriptaculous库实现的购物车示例添加这些功能。

9.2.1 支持拖放的 Ajax 购物车示例

通过采用拖放，我们能给用户提供一个直观的购物车。当用户在用户界面上看到自己喜欢的商品，可以把它们拖动到购物车以便购买。图9-2显示了三个可被购买的商品：两本计算机书籍和一块石头。如果用户不能购买某个特殊商品（或者是因为钱不够了，再或者是因为在这个示例中我们只允许用户购买书籍而非石头），购物车可以禁止停放这些商品，并将它返回至初始位置。

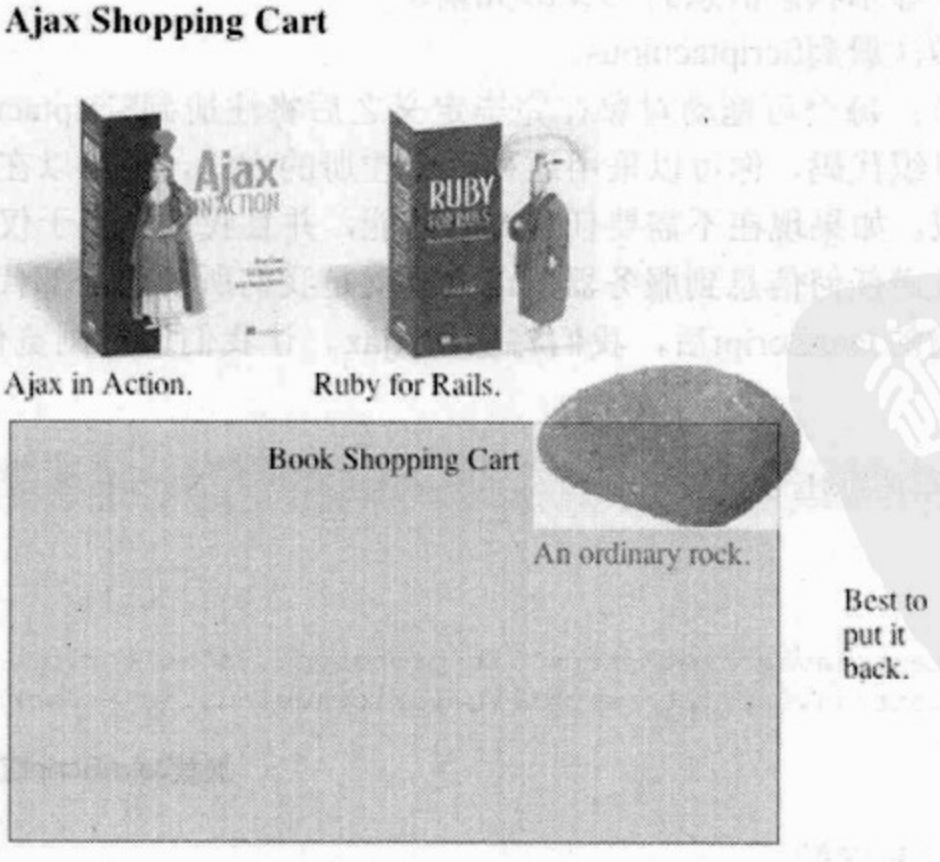


图9-2 支持拖放的购物车

概念不难理解，不过还有几个用户界面事件需要采用Ajax得以实现：onHover、onDrop和revert。但在该购物车应用中仅有一种类型的对象可以被拖动（可被购买的商品），并且仅有一个地方用于停放这些对象，上述几个事件含义如下：

- onHover——用户或许正在考虑（购买）该商品，我们最好鼓励用户购买它。它也可以表明从购物车移除的某个商品。
- onDrop——用户已选中某个要购买的商品。
- revert——在我们的购物车应用中，该回调函数用于表示某个对象应该被放到某个位置还是应该立刻退回到原来位置。当我们拒绝接受某个商品放入购物车时，该函数返回true。

现在，我们已经定义了可拖动区域（书籍和石头），可停放区域（仅为购物车）及有关事件（onHover、onDrop和revert）。让我们看看具体实现过程。

1. 问题

你要让浏览器中的拖放事件和服务端进行通讯。

2. 解决方案

为理解本示例，我们先来看一段HTML代码，一旦理解了这段代码，其中JavaScript代码的职责就会清晰了。在下面的HTML代码中，我们会发现几个重要部分：

- 加载JavaScript库。
- 样式表定义。
- 定义可拖动和可停放区域 元素。
- 定义一个用于显示状态信息的元素。
- 把可停放区域注册到Scriptaculous。

要留意一个细节：每个可拖动对象在完毕定义之后将注册到Scriptaculous库。如何注册它取决于你喜欢怎样组织代码，你可以采用这种逐个注册的方式，也可以在某个代码块中一次注册所有的可拖动区域。如果现在不需要任何Ajax功能，并且我们满足于仅允许用户在浏览器内四处拖动对象而不发送任何信息到服务器，这大概就是我们所要的全部代码。或许你已有所察觉，得到本示例所需的JavaScript后，我们将应用Ajax。让我们通过浏览代码清单9-1来开始这段旅程吧。

代码清单9-1　购物车HTML

```
<html>
<head>

<script type="text/javascript" src="lib/prototype.js"></script>
<script type="text/javascript" src="lib/scriptaculous.js"></script>

</head>

<style type="text/css">
  div.carthoverclass {
    border:1px solid blue;
  }
```

加载JavaScript工具库

指定购物车边框样式表为绿色高亮

```
  div.cart  {
    z-index:100;
    text-align:center;
    height:200px;
    padding:10px;
    background-color:#abf;
  }
</style>

<body>
  <h3>Ajax Shopping Cart</h3>

  <table><tr>

    <td>
      <div alt="Product1" id="product_1"
          itemid="01" style="z-index:500" />
        <img alt="Product1" src="crane_3d.jpg" />
        <br />
        Ajax in Action.
      </div>
      <script type="text/javascript">
        new Draggable('product_1',
            {revert:handleRevert});
      </script>
    </td>
    <td>
      <div alt="Product2" id="product_2"
          itemid="02" style="z-index:500" />
        <img alt="Product2" src="black_3d.jpg" />
        <br />
        Ruby for Rails.
      </div>
      <script type="text/javascript">
      new Draggable('product_2', {revert:handleRevert});
      </script>
    </td>
    <td>
      <div alt="Product3" id="product_3"
          itemid="03" style="z-index:500" />
        <img alt="Product3" src="rock.jpg" />
        <br />
        An ordinary rock.
      </div>
      <script type="text/javascript">
        new Draggable('product_3', {revert:handleRevert});
      </script>
    </td>

</tr></table>

<table><tr>
```

指定购物车的样式表

定义首个可拖动商品

❶ 注册首个可拖动商品

定义第二个可拖动商品

定义第三个可拖动商品

```
    <td width="400px">
      <div id="cart" class="cart" >
        Shopping Cart
      </div>

    </td>
    <td width="25">
    </td>

    <td width="50">
      <span id="cartinfo"></span>
    </td>

    </tr></table>

<script type="text/javascript">
  cartinfoDiv = $("cartinfo");
  Droppables.add('cart',
       { hoverclass:'carthoverclass',
         onHover:
         function(dragged, dropon, event) {
                handleHover(dragged, dropon,
                        event, cartinfoDiv);
         },


         onDrop: :
         function(dragged, dropon, event) {
                handleDrop(dragged, dropon,
                        event, cartinfoDiv);
    }
        }
  )
</script>
</body>
</html>
```

定义可停放区域

定义状态消息区域

❷ 注册购物车可停放区域

把可拖动元素 的id传递给Scriptaculous❶接口并注册一个revert回调函数，这样我们就可以控制某个商品是留在购物车内还是立即返回到初始位置。我们也可以把可停放购物车 ❷的id传递给Scriptaculous并注册onDrop和onHover匿名回调函数，这样我们就可以把状态信息区域的id传入这些回调函数。

如你所见，可拖动和可停放的 元素能包含任意内容——文本、图像等。因此，我们要做的事情就是添加相应代码以创建你想要看到的对象。一旦创建了相应的 元素，我们需要把它们作为可拖动或可停放对象注册到Scriptaculous库，这将实现在用户用鼠标拖动这些可拖动对象时，Scriptaculous库显示适当的动画。当然，我们也注册了很多事件回调函数，它们或用于在某个可拖动对象滑过某可停放对象区域时触发onHover、或用于当可拖动对象停放到可拖动区域时触发onDrop。我们还注册了一个revert回调函数，该函数返回true或false，表示该物品应该立即返回其初始位置还是放入购物车内。我们在这些回调函数内做什么？Ajax。代码中，有两个重要的JavaScript函数支持Ajax：其一，获取最新的状态信息，这取决于用户如何拖动对象（例

如当用户把商品放入购物车时，向他们表示感谢）；其二，获取为可拖动对象定义的revert状态信息（不能购买石头，因此当拖动它到购物车时必须将它返回其初始位置）。其他的JavaScript代码[①]用于确保本例中的用户界面事件能被正确处理——事实证明它是用于处理hover事件的一个诡计。图9-2显示了HTML实现的最终用户界面，代码清单9-2包含这些HTML代码。

代码清单9-2 购物车JavaScript

```
<script type="text/javascript">
function AjaxHTML(url, target)  {
    AjaxOperation(url, function(req) {replaceHTML(req, target);});
}
function AjaxOperation(url, func, asynch) {
   var req;
   if (window.XMLHttpRequest) {
       req = new XMLHttpRequest();
   } else if (window.ActiveXObject) {
       req = new ActiveXObject("Microsoft.XMLHTTP");
   }
   if (req) {
       req.onreadystatechange =
           function (){onReady(req, func);};
       req.open("GET", url, true);
       req.send("");
   }
}

function onReady(req, func) {
   if (req.readyState == 4) {
       if (req.status == 200) {
           func(req);
       }
   }
}

function replaceHTML(req, target)  {
    var content = req.responseText;
    target.innerHTML = content;
}

var revertFlag = true;
function checkRevert(req)  {
    var content = req.responseText;
    revertFlag = (1 == (content - 0));
}

var lastHover;
function handleHover(dragged, dropon, event, target)  {
    itemid = dragged.getAttribute("itemid");
```

通过Ajax更新HTML页面

❶

❷

等待异步响应完成

使用Ajax响应更新HTML页面

❸

① 这里指的是handleHover函数。——译者注

```
    if (lastHover == itemid)  {
        return;
    }
    lastHover = itemid;
    AjaxOperation("revert-" + itemid + ".txt",
            function(req) {checkRevert(req)});
    AjaxHTML("hover-" + itemid + ".html", target);
}

function handleDrop(dragged, dropon, event, target)  {
    lastHover = null;
    itemid = dragged.getAttribute("itemid");
    AjaxHTML("drop-" + itemid + ".html", target);
}

function handleRevert(dragged)  {
    return revertFlag;
}
</script>
```

❹ ❺ ❻

通过Ajax异步显示hover状态

通过Ajax异步显示可停放区域状态

返回revert标志供Scriptaculous库使用

3. 讨论

既然已经看过了代码，那我们就讨论一些层次稍高的而且更有趣的问题。由于需要快速连续地执行多个Ajax请求，所以我们要声明多个不同的请求对象❶以支持在这些对象上创建回调函数❷。这能防止不同请求对象相互重写。还需注意的是revert标志的编码规则，0代表false,1代表true。此处❸，我们从Ajax响应中提取结果并转化成JavaScript语言中的Boolean型对象。还需谨记的是，每次鼠标移动都会产生一个hover事件，所以，除了保留首个事件之外，我们过滤所有❹hover事件。执行完过滤，在hover消息返回之前以同步方式❺获取revert标志的值。之所以这样做是因为能使两个请求不互相干扰。最后，我们重设lastHover的值❻。这将保证handleHover()函数在下次调用时能够执行整个函数。之所以这样做，是因为即使已经把某个商品放入购物车内，但我们可能还会再次拖动它。

在代码清单9-2中，大部分JavaScript代码只是简单地对拖放等事件做出响应并通过Ajax从服务器获取页面更新。我们认为，只有handleHover()函数看起来有点复杂。第一个问题是不仅仅在hover事件首次出现时用该函数，而且包括此后可拖动对象在可停放对象上方的悬浮过程中的每次鼠标移动。你希望把每次鼠标移动事件都通过网络发送到服务器，这种可能性极小，而实际上并不可行，因此我们添加了一段代码只处理首次发生的hover事件。它的任务是跟踪当前可拖动对象的hover事件，假如可拖动对象和上次的对象是同一个，程序会立即从handleHover()函数跳出不再进行后续处理。

第二个问题是我们需要在handleHover()函数中执行两个操作：一个操作是取得和商品相关的消息并显示在购物车右侧的信息栏以鼓励用户购买它；另一个操作是检测商品是否无法放入购物车内。在简单的Ajax程序中，通常只使用一个单例的、全局的请求对象。但在本购物车示例中，我们要保证正确处理获取revertFlag变量以及悬浮消息。解决方案就是把XMLHttpRequest对象作为一个参数传递到相应的回调函数，这样每个不同的请求就能够单独处理。现实世界中的网上购物应用通常拥有三个以上的物品供用户选择，而且它们可能是动态生成并加载到应用中的。幸

好，只需将下列代码包含到每个商品的声明部分：

```
<script type="text/javascript">
  new Draggable('product_1', {revert:handleRevert});
</script>
```

9.2.2 拖放列表中的数据操纵

可拖放对象基础的、原生的“拖-放”概念非常强大，因此不难想象用它能创建多少种各式各样的应用。例如，假设你有一些商品如水果和蔬菜，你想把它们组织到两个列表：按类型（水果或菜）和按优先级（按它们在列表中的次序）。使用一个拖放组件比较合理，所有的物品都是可拖动的（因此你可以到处移动它们），两个列表都支持停放。当你把某个物品从一个列表拖动到另一个列表，该物品会从原列表删除并添加到新列表。当你在列表内部拖动物品时，其他物品将会腾出位置以便该物品停放到某个特定位置。实现上述功能，看似确实需要在onHover事件处理器中编写大量复杂JavaScript代码。幸好，Scriptaculous提供了支持拖放对象复制功能的内建列表控件。我们要做的就是在HTML代码中使用list标签（即<li>）元素创建两个列表，然后告诉Scriptaculous这些商品各自属于哪个列表以及当列表发生改变时该怎么做。当列表变化时，我们可以借机发起Ajax请求以调用服务端的程序，如图9-3所示。

对本例而言，在用户首次将水果或蔬菜放错位置时，我们将通知他们。由于需要传递与此相关的复杂数据结构到服务器（两个列表的次序），我们将采用DWR来处理数据散列化（data marshaling）问题。结合使用Scriptaculous和DWR，我们能够利用这两个功能强悍的框架以集中精力编写服务端Java代码。因为你将会明白，判断水果还是蔬菜是最为复杂的部分。

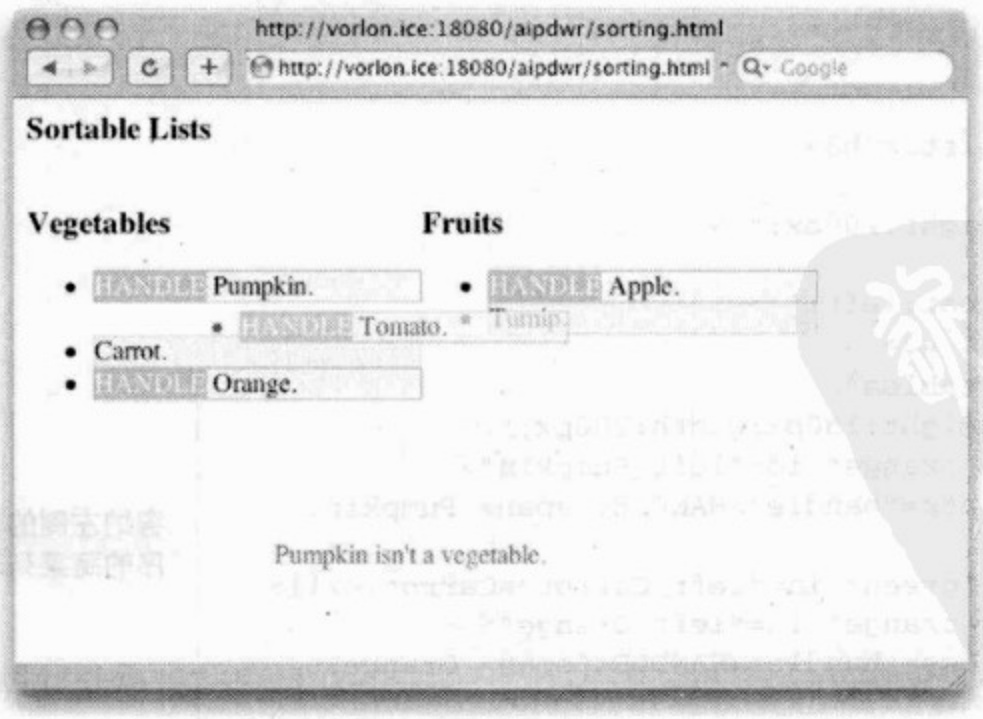


图9-3 蔬菜水果种类验证

1. 问题

你希望提供一个数据列表组件，可以通过拖放操作修改数据。

2. 解决方案

和其他Ajax应用一样，我们的代码将分为3个主要部分：HTML、JavaScript和服务器端代码。由于我们采用DWR和Scriptaculous，程序中的HTML和JavaScript将会变得相当简单，本质上是对象注册和事件处理代码，而应用代码将用Java编写。让我们从HTML代码（代码清单9-3）开始介绍如何创建两个列表。

代码清单9-3　HTML代码

```
<html>
<head>
<style type="text/css">
li.green {
  background-color: #ECF3E1;
  border:1px solid #C5DEA1;
  cursor: move;
}

li.orange {
  border:1px solid #E8A400;
  background-color: #FFF4D8;
}

span.handle {
  background-color: #E8A400;
  color:white;
  cursor: move;
}
</style>
</head>
<body>

<h3>Sortable Lists</h3>

<div style="height:200px;" >

<div style="float:left;" >
<h3>Vegetables</h3>
  <ul id="vegetables"
      style="height:150px;width:200px;">
    <li class="orange" id="left_Pumpkin">
      <span class="handle">HANDLE</span> Pumpkin.
    </li>
    <li class="green" id="left_Carrot">Carrot.</li>
    <li class="orange" id="left_Orange">
      <span class="handle">HANDLE</span> Orange.
    </li>
  </ul>
</div>

<div style="float:left;" >
  <h3>Fruits</h3>
```

指定蔬菜列表的样式表

指定水果列表的样式表

指定水果列表的处理样式表

容纳左侧的已排序的蔬菜列表

```
  <ul id="fruits" style="height:150px;width:200px;" >
    <li class="orange" id="right_Apple">
      <span class="handle">HANDLE</span> Apple.
    </li>
    <li class="orange" id="right_Tomato">
      <span class="handle">HANDLE</span> Tomato.</li>
    <li class="green" id="right_Turnip">Turnip.</li>
  </ul>
</div>

<br>

<div style="clear: both; margin-left: 150px;" >
  <span id="plant-error"
        style="color: red;" ></span>
</div>

</div>

<script type="text/javascript">
  Sortable.create("vegetables",
     {dropOnEmpty:true,
     containment:["vegetables","fruits"],
     constraint:false,
     onUpdate:handleUpdate});
  Sortable.create("fruits",
     {dropOnEmpty:true,handle:'handle',
     containment:["vegetables","fruits"],
     constraint:false,
     onUpdate:handleUpdate});
</script>

</body>
```

容纳右侧的已排序的水果列表

指定应用的消息区域

注册左侧列表

注册右侧列表

上述代码为我们实现了两个商品列表和一个用于显示服务端信息的区域，还有使商品得以拖动停放的Scriptaculous拖动对象注册。接下来，我们需要采用DWR库的一个层来粘合客户端和服务器端的Java代码。我们只需引入必需的DWR库并实现和DWR协作的Scriptaculous回调函数（代码清单9-4）。

代码清单9-4 JavaScript代码

```
<head>
<script type="text/javascript" src="lib/prototype.js">
  </script>
<script type="text/javascript" src="lib/scriptaculous.js">
  </script>
<script type="text/javascript" src="dwr/engine.js"></script>
<script type="text/javascript" src="dwr/util.js"></script>
<script type="text/javascript" src="dwr/interface/Demo.js"></script>

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

加载Scriptaculous库

加载DWR基础库

加载DWR应用库

```
function handleUpdate(list)  {
    Demo.checkPlant(
            Sortable.serialize("vegetables")
            + "&" + Sortable.serialize("fruits"),
             showPlantMessage);
}

function showPlantMessage(message)  {
    DWRUtil.setValue("plant-error", message);
}
</script>
</head>
```

❶ 对应用中发生改变的列表内容进行编码

❷ 显示应用信息

有两个列表，每个列表都可以改变，因此❶任何改变发生时，我们都将两个列表的当前状态发送到服务端。服务端使一个消息响应我们的列表更新❷，通过使用DWRUtil.setValue()，我们把该消息插入到页面中的相应位置。

还可以看到，在DWR的脚本函数中，我们只需实现一个服务端方法：基于两个列表的内容进行检测，在发生错误时返回一个字符串（代码清单9-5）。

代码清单9-5　Java代码

```
package aip;

import java.util.HashMap;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
public class Demo  {

    Map plants;

    public Demo()  {
        initPlantData();
    }

    private void initPlantData()  {
        plants = new HashMap();
        List fruits = new ArrayList();
        List vegetables = new ArrayList();
        fruits.add("Apple");
        fruits.add("Pumpkin");
        fruits.add("Orange");
        fruits.add("Tomato");

        vegetables.add("Carrot");
        vegetables.add("Turnip");

        plants.put("vegetables", vegetables);
        plants.put("fruits", fruits);
    }

    public String checkPlant(String listUpdate)  {
```

初始化水果/蔬菜存储系统

```
        String[] items = listUpdate.split("&");          | 把编码的列表
        for (int i = 0; i < items.length; i++)  {        | 分割成项目
            if ("".equals(items[i]))  {
                continue;
            }
            String[] pair =                              | 把项目分割
                items[i].split("\\[\\]=");               | 成列表/条目
            List plant = (ArrayList) plants.get(pair[0]);
            if (!plant.contains(pair[1]))  {
                return pair[1] + " isn't a " +           |
                pair[0].substring(                       | 汇报水果/蔬菜
                    0, pair[0].length() - 1) +           | 的鉴别结论
                ".";                                     |
            }
        }
        return "";
    }
}
```

让我们检查checkPlant()方法的更细节的内容，因为处理植物类别的代码量很少，而用于对Scriptaculous序列化后的列表参数进行解码的代码则很多。在Scriptaculous回调函数的handleUpdate()函数中，我们把第一个列表和第二个列表先后序列化并通过DWR发送到服务端。参数listUpdate看起来如下：

```
vegetables[]=Pumpkin&vegetables[]=Carrot&vegetables[]=Orange
    &fruits[]=Apple&fruits[]=Tomato&fruits[]=Turnip
```

每个商品条目以&分隔（就像HTML表单数据一样），单个条目编码如下：

```
listname[]=id
```

因此，为了对它进行解码，我们先把整个串在&处分割成若干个条目，然后再将每个条目以[]=分割为列表名称及其id。函数checkPlant()的其余部分用于在内存（本例没有采用真正的数据库，此处指的是利用Hash表存储数据）中查找首个错误。找到错误时，该函数产生一个基于实际数据的错误消息。

3. 讨论

使用Scriptaculous可以轻松开发支持列表的Ajax应用。不过，Scriptaculous和桌面应用的API不同，本例中列表发生更新时，不会传给客户端一个对象。相反，我们收到的是以id数组方式存储的列表当前状态。正因如此，给列表中的各个元素分配有意义的id很重要，而不要简单地按照它在列表中的位置进行命名。更确切的说，在我们的应用中，列表元素id必须唯一。实现id唯一的方式有很多种，你可能想利用数据库的主键或者一个对象的hash码。在请求到达服务端时，任何能够被唯一识别的值都可以作为id(但应适当简短，因为所有列表元素的id都将发送到服务端)。

尽管Scriptaculous令人惊叹，但它不是我们工具箱中的唯一工具。让我们来看看另一个流行的Ajax框架——ICEfaces。使用它，你可以实现支持拖放功能的Ajax应用程序。

9.2.3 使用 ICEfaces 创建 Ajax 购物车

ICEfaces是一个用于开发Ajax、同时也是标准的JSF（JavaServer Faces）应用的开源工具

（http://www.icefaces.org）。ICEfaces的显著特点是它的开发方法学和Ajax推送（Push）应用发起的更新的原生能力。（Ajax推送与“Comet”或者“反向Ajax（Reverse Ajax）”很接近，它用于在Web应用中实现消息通知或多用户协作。）开发一个ICEfaces应用就是在开发一个标准JSF应用。无须修改代码它就能变成一个Ajax应用，并因此而保留JSF非常强调分离MVC（Model-View-Controller）的好处。要从“服务端发起更新”中获益，仅需调用一个ICEfaces API方法就能搞定。当需要更新页面时，服务端只需调用`render()`，ICEfaces框架判定是否需要更新并将所需的局部页面更新发送到浏览器。

要想了解如何使用ICEfaces开发Ajax应用，我们将用它来实现之前用JavaScript和Scriptaculous完成的支持拖放的购物车示例。这是一个简单的应用：用户可以拖动三个物品到购物车，应用将更新相应信息。该信息取决于拖动对象和是否拖到购物车。

1．问题

使用ICEfaces框架实现支持拖放的购物车应用。

2．解决方案

图9-4展示了Ajax版本的购物车页面。如果你觉得它似曾相识，那就对了。ICEfaces将利用Scriptaculous提供的拖放功能，开发人员仅被JSF提供的组件模型吸引，而Scriptaculous的JavaScript接口被抽象并隐藏到组件模型。

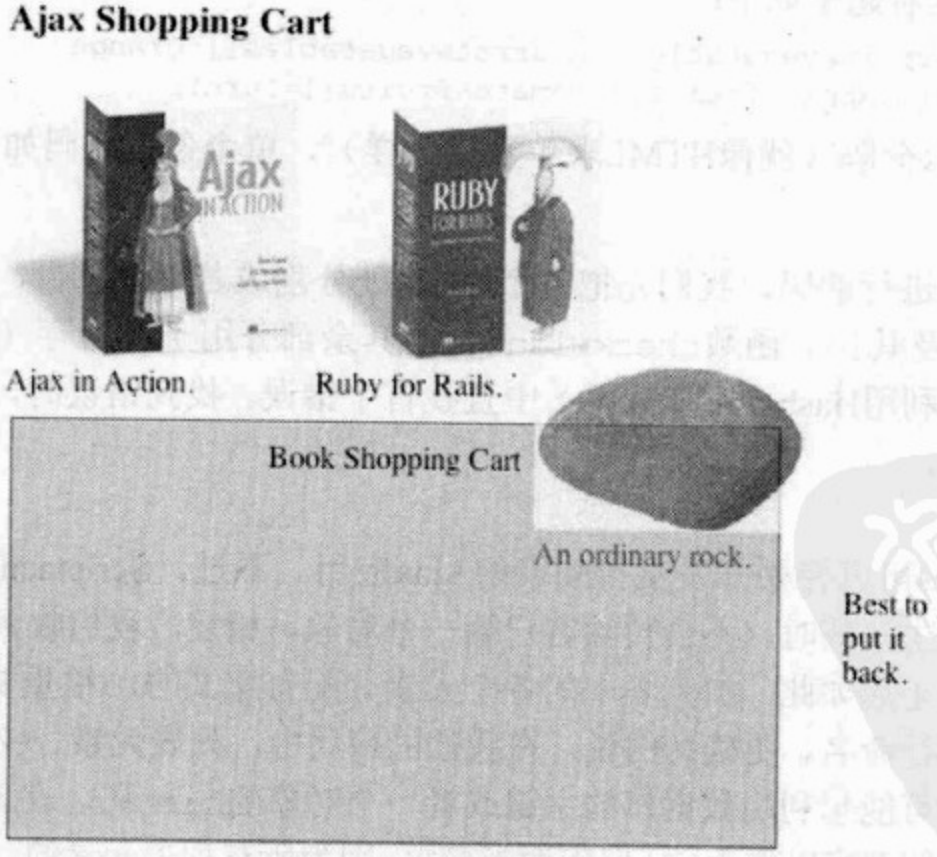


图9-4　使用ICEfaces创建支持拖放的购物车

该实现将分成3个主要部分：一个用户界面声明文件（在代码清单9-6中以JSP文档显示，但与HTML标记极为类似），两个浅显易懂的JavaBean和一个包含注入应用数据的XML配置文件。让我们深入研究界面声明文件中的标记，我们会看到少量的JSF配置和CSS定义，但我们主要把

注意力放到可拖动对象的配置上。值得关注的其他部分如何显示购物车消息，它很简单但容易被忽视，因为它只不过是一个绑定到JavaBean的文本输出框组件。ICEfaces库为我们实现了所有Ajax功能，例如当某个物品被停放到购物车并且消息改变，框架能检测到局部页面更新并自动执行。我们只需指定页面中的各个部分如何绑定到模型数据。

代码清单9-6 ICEfaces JSF

```
<f:view xmlns:f="http://java.sun.com/jsf/core"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:ice="http://www.icesoft.com/icefaces/component">

 <html>
 <head>
    <style type="text/css">
      div.cart  {
        z-index:100;
        text-align:center;
        height:200px;
        padding:10px;
        background-color:#abf;
      }
    </style>
 </head>
 <body>

  <h3>Ajax Shopping Cart</h3>

  <ice:form>

  <table><tr>

   <td>
     <ice:panelGroup
         style="z-index:500;cursor:move;"
         draggable="true"
         dragListener=
           "#{dndBean.dragListener}"
         dragMask=
           "dragging,drag_cancel,hover_end"
         dragOptions=
           "#{craneItem.dragOptions}"
         dragValue="#{craneItem}" >
     <img alt="Product1" src="crane_3d.jpg" />
     <br/>
       Ajax in Action.
   </ice:panelGroup>

</td>

<td>
  <ice:panelGroup
```

声明JSF标签命名空间

包含HTML标记和CSS样式表

指定包含输入组件的表单

声明可拖动书籍的容器

指定是否可拖动

1

2

3

4

指定拖动书籍的HTML元素

关闭可拖动书籍容器

```
        style="z-index:500; cursor:move;"
        draggable="true"
        dragListener=
          "#{dndBean.dragListener}"
        dragMask=
          "dragging,drag_cancel,hover_end"
        dragOptions=
          "#{blackItem.dragOptions}"
        dragValue="#{blackItem}">
      <img alt="Product2" src="black_3d.jpg" />
      <br/>
        Ruby for Rails.
    </ice:panelGroup>
  </td>

  <td>
    <ice:panelGroup
        style="z-index:500; cursor:move;"
        draggable="true"
        dragListener=
          "#{dndBean.dragListener}"
        dragMask=
          "dragging,drag_cancel,hover_end"
        dragOptions=
          "#{rockItem.dragOptions}"
        dragValue="#{rockItem}">
      <img alt="Product3" src="rock.jpg" />
      <br/>
        An ordinary rock.
    </ice:panelGroup>
  </td>

</tr></table>

<table><tr>

  <td width="400px">
    <ice:panelGroup style="z-index:0;"
        dropTarget="true">
      <div id="cart" class="cart" >
          Book Shopping Cart
      </div>
      </ice:panelGroup>
    </td>

    <td width="25">
    </td>

    <td width="50">
      <ice:outputText
          value="#{dndBean.dragMessage}"/>
    </td>
```

指定第2本可拖动的书籍

指定可拖动石头

指定购物车停放区域

定义停放目标

从JavaBean获取消息并设置到文本域

```
    </tr></table>

   </ice:form>
  </body>
  </html>
</f:view>
```

名为dndBean❶的JavaBean用于对我们的购物车应用进行建模。它有一个监听器方法叫做dragListerner()，绑定表达式通知JSF，当该可拖动对象发生一些有趣操作时触发此方法。

我们并不是对拖放过程中的所有事件都感兴趣。ICEfaces在服务端处理这些事件，而且我们不希望把拖动过程的每次鼠标移动发给服务器❷，因此我们使用拖放掩码（Drag Mask）屏蔽不感兴趣的事件。

我们希望不同的商品有不同的拖动行为（例如当它停放到购物车时是否应该被恢复到其初始位置）。通过为商品JavaBean的dragOptions属性绑定表达式❸，我们可以对各个物品设置不同的行为。

dragItem是一个与可拖动对象关联的实际JavaBean对象，我们的dragListerner()函数可以在拖动发生时操作该对象。这令我们的Java应用程序与应用程序级别❹有意配置的对象协同工作。

你能在页面声明中发现Ajax吗？这是有意不让人们看见——ICEfaces的目标之一就是提供对Ajax的透明支持。开发人员集中精力在应用上：页面上有哪些组件以及它们如何绑定到动态数据模型；组件如何通过网络使用XHR对象来更新等底层概念被ICEfaces框架进行了抽象处理。

我们已经创建了一个页面，它声明了给用户显示的各个组件。现在我们需要实现这个应用的可执行部分，该部分完全使用JavaBean在服务端实现。让我们开始最简单的模型部分：购物车商品。ShoppingItem（代码清单9-7）类完全面向数据，它仅是一个能容纳商品信息的JavaBean。在一个更为完整的应用中，该类还会包括诸如价格、重量、可用性等信息。但对于一个简单的应用，它只包含一条在用户把物品拖动到某处时要显示的消息，还有拖动行为的一些选项（以支持石头可以跳回到初始位置而不是停放在购物车内）。

代码清单9-7 ShoppingItem

```
package aip;

import java.io.Serializable;

public class ShoppingItem implements Serializable  {

    String hoverMessage = "";

    public void setHoverMessage(String message) {
        this.hoverMessage = message;
    }

    public String getHoverMessage() {
        return this.hoverMessage;
    }
```

设置/获取hover消息

```
    String dropMessage = "";

    public void setDropMessage(String message) {
        this.dropMessage = message;
    }

    public String getDropMessage(){
        return this.dropMessage;
    }

    String dragOptions = "";

    public void setDragOptions(String options) {
        this.dragOptions = options;
    }

    public String getDragOptions() {
        return this.dragOptions;
    }

}
```

设置/获取停放消息

设置/获取拖动选项

ShoppingItem可以描述在我们这个简单示例中的商品，这很容易理解，但不同的商品如何被分别创建以及如何让它们变成可配置的？一个值得注意的技巧是利用Managed Bean的依赖注射或控制反转能力。本例中（代码清单9-8），我们将使用少数Managed Bean来实例化并指定本应用中的商品，然后将它们注入到来自客户端的用户会话中（此简单示例中的这些Managed Bean不存在有意义的依赖）。

文件清单9-8　ICEfaces JSF 配置文件

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">

<faces-config>

    <managed-bean>
      <managed-bean-name>dndBean</managed-bean-name>
      <managed-bean-class>aip.DragDropBean</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

    <managed-bean>
      <managed-bean-name>craneItem</managed-bean-name>
      <managed-bean-class>aip.ShoppingItem</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
        <managed-property>
            <property-name>hoverMessage</property-name>
            <value>Good choice.</value>
        </managed-property>
```

声明JSF配置文件

在session中声明拖放Bean

为第一本书命名

指定session的范围

❶ 声明商品的hover消息

```
        <managed-property>
            <property-name>dropMessage</property-name>
            <value> Thank you for your purchase.</value>
        </managed-property>
        <managed-property>
            <property-name>dragOptions</property-name>
            <value></value>
        </managed-property>
    </managed-bean>

    <managed-bean>
      <managed-bean-name>blackItem</managed-bean-name>
      <managed-bean-class>aip.ShoppingItem</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
        <managed-property>
            <property-name>hoverMessage</property-name>
            <value>Excellent selection.</value>
        </managed-property>
        <managed-property>
            <property-name>dropMessage</property-name>
            <value>You are sure to enjoy.</value>
        </managed-property>
        <managed-property>
            <property-name>dragOptions</property-name>
            <value></value>
        </managed-property>
    </managed-bean>

    <managed-bean>
      <managed-bean-name>rockItem</managed-bean-name>
      <managed-bean-class>aip.ShoppingItem</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
        <managed-property>
            <property-name>hoverMessage</property-name>
            <value>Put it back.</value>
        </managed-property>
        <managed-property>
            <property-name>dropMessage</property-name>
            <value>You really don't want this one.</value>
        </managed-property>
        <managed-property>
            <property-name>dragOptions</property-name>
            <value>revert</value>
        </managed-property>
    </managed-bean>

</faces-config>
```

声明商品的停放消息

声明商品的拖放选项（默认）

声明第二本

声明第三个商品（石头）

声明商品的拖放选项（返回原位置）

属性hoverMessage直接对应ShoppingItem对象的hoverMessage域。JSF查找一个名为setHoverMessage()的函数❶，并在以我们在配置文件中设定的值为参数实例化的对象上调用该方法。

以上文件提供了一个用户界面和一些数据，但我们的应用实际上还不能完成什么功能。我们还没有任何用于用户事件处理的函数。我们的应用该做什么？当用户拖动一个物品，状态信息的更新取决于用户做了什么。正如我们在页面文件中看到的一样，我们的JavaBean通过dragListerner回调函数发现该对象被拖动了。该信息依靠绑定到DragDropBean的文本输出组件实现更新（ICEfaces确保它在需要时在页面内更新）。让我们看看这个对象有哪些函数以及如何实现（代码清单9-9）。

代码清单9-9　DragDropBean

```
package aip;

import com.icesoft.faces.component
       .dragdrop.DragEvent;                          使用ICEFaces库的拖放事件
import com.icesoft.faces.component.ext.HtmlPanelGroup;

public class DragDropBean  {

    private String dragMessage = "";

    public void setDragMessage(String message)  {
        this.dragMessage = message;                  ❶ 设置/获取dragMessage
    }

    public String getDragMessage()  {
        return this.dragMessage;
    }

    public void dragListener(                        处理拖动事件
            DragEvent dragEvent)  {
        ShoppingItem item = (ShoppingItem)           提取购物商品对象
          ((HtmlPanelGroup)
                dragEvent.getComponent())
                      .getDragValue();

        if (null != item) {
          if ( dragEvent.getEventType()
                == dragEvent.HOVER_START )  {        ❷ 如果商品在购物车上方悬浮显示hover消息
            this.dragMessage =
              item.getHoverMessage();
          } else if ( dragEvent.getEventType()
                == dragEvent.DROPPED )  {            ❸ 如果商品已放入购物车显示停放消息
            this.dragMessage =
              item.getDropMessage();
          }
        }

    }
}
```

setDragMessage()方法❶或许永远不会被调用，但它在语法上还是包含了属性的setter方法。如果是一个鼠标hover事件，我们将从这个物品对象上提取hover消息。该商品对象是一个

ShoppingItem对象，当它在购物车❷上方悬浮时，它知道应当显示什么信息。如果该对象已经被停放❸，我们可以提取其停放信息。

3. 讨论

按照MVC的理论，我们要关注降低模型与视图的耦合程度。拖放事件不应该直接触发购物车旁边的消息以切换到某个确定值，而是由应用程序接收该事件并导致该程序内部状态发生变化。如何设计内部状态改变页面视图，对设计人员来说可以是他想要的任何方式，例如在购物车旁边或者弹出对话框中显示一条消息。不过重要的是设计人员不必知道拖动某个物品到购物车会改变内部状态。状态改变属于业务逻辑范畴，把它放到应用层代码中实现让人感觉更舒服。

9.3 总结

本章我们介绍了在Ajax应用中使用拖放的三个示例。首先，我们学习了如何使任意对象支持拖放功能；其次，我们了解了一个精巧的、特别适用于数据复制的拖放列表；最后，我们考察了如何使用ICEfaces框架创建支持拖放的Ajax应用。后两者区别在于使用Scriptaulous库，列表中的物品会在需要时自动挪出空间以及Scriptaculous提供了方便的条目序列化函数（它不仅考虑列表中已放置的物品，还考虑物品在列表的位置的问题）。总之，本章得出的重要经验就是JavaScript库（例如Scriptaculous和ICEfaces）提供了高度抽象但易于集成Ajax的拖放事件模型。而现在的任务就是使用这些技巧去创造比桌面用户预想的还要丰富的令人惊叹的浏览器应用。

第10章 对用户友好一点

本章内容

- 处理网络延迟问题
- 提供上下文相关帮助
- 错误检测和报告

你是想让用户沮丧的咒骂，还是会心地微笑？

这像是一个很愚蠢的问题（毕竟，谁会故意虐待他们的用户呢？），但实际上我们很容易就会搞出一些一点儿也不友好的用户界面。特别在Web上更是如此，因为在Web上，我们被局限在HTML规范所规定的那几种界面控件中[①]。

事实上，Web应用所面对的优使性的挑战与桌面程序面对的挑战大相径庭，尽管如此，那些适用于桌面应用的准则仍然可以适用于有如此之多限制的Web应用。本章就会研究这样的三个准则：

- 提供反馈——用户若搞不清楚状况，就会灰心丧气。让用户了解下面会发生什么，可算是优使性最重要的准则之一。
- 主动帮助——预先备好用户需要了解的信息，将大大减少用户傻盯着屏幕不知道干什么的情形。
- 符合直觉——确保用户不会迷失或者无法确定下面该干什么。最要紧的是，不要做让用户意料不到或感到怪异的事情。

当然，还有许多其他的优使性原则，但是上述这些方面是Web应用最常失足之处，本章就会探讨这些主题。你会看到如何运用Ajax技术来清除这些常见的绊脚石，使用户获得良好的用户体验。

在本章中，我们将以开发者的身份投身于一个基于社区共享菜谱的Web应用并探索这些主题。根据所谓“Web 2.0”的精神，在这个网站上，社区成员提供大量表现为菜谱形式的内容，并与其他成员分享这些内容。而我们所要做的正是提供这样一个平台。

由于这是一个基于社区的应用，优使性就成为最重要的考量。如果社区成员在尝试我们的系统时屡遭挫折，就很难为社区贡献内容。而对社区应用来说，内容就是一切。同时，作为社区系统，很可能会有大量用户频繁地使用。我们需要保证，用户荷载量对社区成员使用我们应用时的体验不会有很大的影响。

下面就让我们深入进去，看看我们所面对的优使性挑战，以及如何用Ajax来应对这些挑战。

---

① 正在制定的HTML5规范会增加许多新的表单控件并对现有的控件进行扩展。——译者注

10.1 与延迟作斗争

想象一个用户一边狂敲键盘一边对着屏幕大喊："我的数据在哪儿？"

Web上的延迟是无可避免的。无论是由于网络速度特慢、服务器性能不足、代码效率低下，还是数据库操作过多，有时候我们的用户不得不苦苦等待。这常令人沮丧，不仅是因为浪费时间，而且用户会纳闷："操作真的有起作用，网站真的在运转吗？"

这一延迟，即从请求提交给服务器到浏览器收到响应得到结果所间隔的时间，就称为等待时间（latency），它是Web应用最主要的优使性问题之一。

10.1.1 以反馈来应对等待

Ajax可以帮助我们减少等待时间，相比非Ajax的Web应用所必须的传统而笨重的整页刷新而言，Ajax让我们的应用可以有更小的请求和响应。然而，在降低了应用运转时的等待时间的同时，这也会增加用户的疑惑。

怎么会这样？

Web浏览器用户习惯于浏览器所提供的视觉反馈线索——通常是状态栏里某种形式的状态条，以及工具栏区域中的"旋转图标（throbber）"或其他指示。使用Ajax常常绕过了这些反馈机制。所以尽管我们的Ajax应用运转的等待时间缩短了，但因为不能完全消除延迟，我们的用户就会陷于没有反馈的延迟中——尽管延迟变短了。

1. 问题

因为我们不可能完全消除Web应用的等待时间，所以我们希望那些可能要花上一点时间的操作在运转时给用户提供某种视觉反馈。

2. 解决方案

让我们考虑菜谱共享应用的搜索页面，在这个页面上用户可以通过标题关键字来搜索菜谱。记住这是一个网络社区，所以我们预计（或者说热切地希望）网站数据库会有大量的由网站会员贡献的内容。这意味着完成一个搜索操作可能要花上一些时间。

既然我们会通过Ajax请求来得到搜索结果，浏览器的视觉提示可能不起作用，或者不足以给用户以适当的反馈。我们需要让用户了解到我们的应用程序正为他们努力工作。有许多方式来提供这样的反馈，这里我们会通过两段代码来演示两种不同的方式：一个是显示页面一级的加载指示（loading banner），另一个是通过局部动画来给予反馈提示。

- 一些设置细节

本章中的解决方案需要由服务器端资源服务来提供应用所需的后端行为。显然这里并没有包括一个真正的Web应用后端，而是由一些Servlet和Java类充当"魔法烟雾和镜子"，对我们来说这样就可以了。

本章的源代码可从www.manning.com/crane2下载，可以阅读readme文件来了解如何配置Java Web应用。听上去有点吓人？甭担心，所有代码已经被组装为一个完整独立的Web应用，比你想的还简单。

因为这些例子的目的是要说明客户端的优使性（usability）技巧，而不是要研究社区网站的

服务器端运作，所以除了后端操作的调用接口之外，我们不会过多关注后端运作。

如果你对后端组件确实有兴趣，本章示例也提供了源代码，实现了一个轻量级的采用命令模式（Command pattern）的前端控制器。如果有兴趣，你可以看readme文件或是直接阅读源代码来了解细节。

从客户端代码的视角来看，所有后端提供的服务都可通过如下形式的URL来访问：

```
/aip.chap10/command/WhatToDo
```

在这个URL中，/aip.chap10是为本章代码所设置的Java Web应用的context路径（如果你遵照readme文件中的指令），/command部分是servlet路径，用于调用（在应用程序的部署描述符（deployment descriptor）中定义的）前端控制器，/WhatToDo是我们希望服务器为我们执行的特殊命令。

从Ajax页面创作者的角度来说，我们不必操心这些细节。我们唯一需要知道的就是如果我们想执行SearchForRecipes（搜索菜谱）命令，对应的URL应该是：

```
/aip.chap10/command/SearchForRecipes
```

至于如何调用正确的命令，服务器端的代码会去处理这些细节。

有了这些背后的支持，我们就可以开始构建搜索页面了，至少对展示我们的示例是足够了。

- 方式1：显示页面指示

代码清单10-1是我们的第一个方案的HTML部分，如你所料，它非常简单。本章下载包中的solution-10.1.1文件夹下的solution-10.1.1a.html文件即为该HTML文件。

代码清单10-1　页面指示“正在处理中”的HTML部分

```
<html>
  <head>
    <!--script and CSS will go here-->
  </head>
  <body>
    <fieldset>
      <legend>Find Recipes</legend>

      <form name="searchForm"
            onsubmit="performSearch();return false;">

        <div>
          <label>Where recipe title contains:</label>
          <input type="text" name="searchTerms"
                 id="searchTermsField"/>
        </div>

        <div>
          <input type="submit" value="Search!"/>
        </div>
      </form>

      <div id="resultsContainer"></div>
```

❶ 声明了一对fieldset和legend元素

❷ 定义表单

❸ 声明表单所包含的控件

❹ 创建用于存放结果的容器

```
    </fieldset>

    <div id="processingNotice" style="display:none" >
      Searching! Please wait...
    </div>
  </body>
</html>
```

❺ 声明初始时隐藏的页面指示

<fieldset>和<legend>元素组合❶提供了良好的样式，其中的表单❷[①]包含了我们的控件和一个用于显示结果的 ❹。

表单控件❸包括一个标签（label）、一个用于输入搜索词的文本字段和一个提交按钮。

另一个值得注意的是表单上的onsubmit事件处理器。我们耍了点小花招，调用函数来执行我们的“提交操作”，然后返回false以防止表单本身像原来那样提交到服务器，因为我们会在performSearch()函数中自行发送Ajax调用到服务器。

在<fieldset>之外，我们定义了一个 元素，包含了文本“Searching! Please wait...（正在搜索！请稍等……）”，该元素最初是隐藏的❺。由于我们会对这个元素进行绝对定位，所以它在HTML中的位置是无所谓的，不过习惯上我们会把这类元素放在所有其他元素之后。

下面的样式表中定义了该元素的绝对定位规则，以及其他具有非常醒目效果的样式规则，使得该元素“无法被忽略”：

```
#processingNotice {
  position: absolute;
  top: 0px;
  right: 0px;
  background-color: red;
  color: white;
  font-size: 1.2em;
  width: 320px;
  text-align: center;
  padding: 4px;
}
```

上述样式把元素定位到页面的右上角，同时加大了字体，使用了醒目的红色背景。在搜索运转期间，页面看起来就像图10-1。

代码清单10-2显示的是在<head>元素中定义的页面脚本部分。

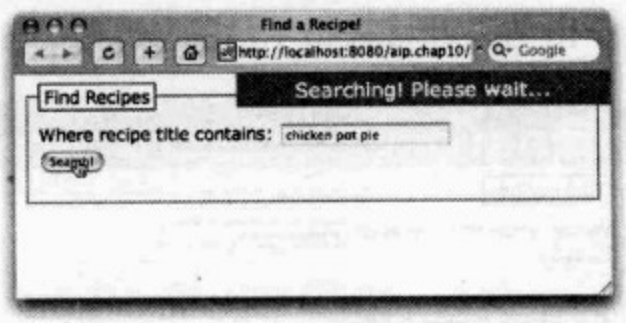


图10-1 我正在搜索

① 原作者存在一个疏忽，fieldset元素表示表单内的分组，所以通常应该是form包含fieldset，而不是fieldset包含form。

代码清单10-2　为页面上的“正在处理中”的指示所准备的脚本

```
<script type="text/javascript">
  function performSearch() {
    $('resultsContainer').innerHTML = '';
    new Ajax.Request(
      '/aip.chap10/command/SearchForRecipes',
      {
        onSuccess: showResults,
        onFailure: showResults,
        parameters: $('searchForm').serialize(true)
      }
    );
    Element.show('processingNotice');
  }

  function showResults(request) {
    Element.hide('processingNotice');
    $('resultsContainer').innerHTML = request.responseText;
  }
</script>
```

❶ 清空结果

❷ 开始搜索

❸ 显示指示

❹ 隐藏指示并显示结果

函数performSearch()被用于表单上的onsubmit事件处理器，它执行三个操作。首先清空之前的搜索结果❶——在一个页面上可以进行多次搜索，所以我们要清除已有的结果。

然后发起搜索❷——创建一个Ajax请求，调用由(如之前所描述的)后端所提供的SearchForRecipes命令，并传入搜索词字段的值。为方便起见，无论你输入了什么搜索词，我们的“魔法烟雾和镜子”般的服务器代码都完全忽略，直接返回“Chicken Pot Pie（鸡肉派）”的搜索结果——这当然不符合现实，但对于我们的目的来说已经足够了。

发送请求之后，名为processingNotice的隐藏指示就会显示出来❸，让用户了解到尽管看上去什么都没有显示，但其实服务器正忙着搜索香甜的鸡肉派的菜谱。

处理器onSuccess和onFailure被设为同一个函数：showResults()。这个函数❹会把指示隐藏起来，并在名为resultContainer的 元素中显示结果。万一发生错误，服务器格式化后的错误信息也会作为结果显示出来。这不是显示错误信息的最好方法，但是目前先这样（我们会在后面的例子中处理故障信息的显示问题）。

这里的要点是，无论成功（success）或失败（failure），当请求完成时，指示都需要被隐藏起来。

图10-2展示的就是页面显示出结果并移去指示后的效果。

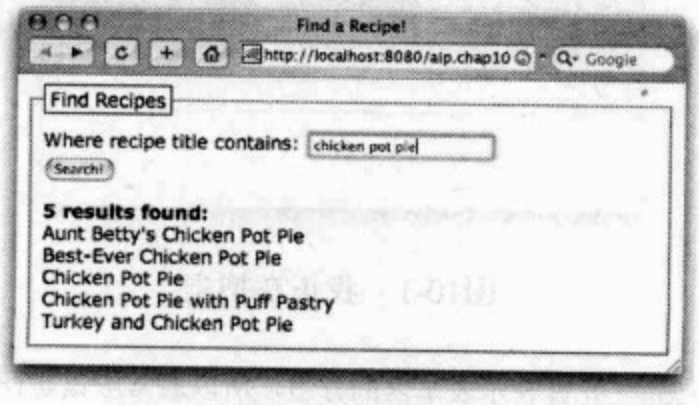


图10-2　搜索完毕

显然，在真实应用中，我们会把结果组织得更好一点，并会返回关于这些菜谱的更多信息：诸如菜谱的作者、提交时间、社区对菜谱的评分等。我们也会把菜名链接到实际的菜谱，不过这里所做的已经足以说明我们关于用户反馈的观点了。

这个例子的完整代码可以从本章源代码/solution-10.1/solution-10.1a.html中获得。

有人可能会认为，把指示放在页面顶部会令人分心，因为它使用户的关注焦点离开了页面的主要元素。我们下面会修改代码，使用另一种更局部化的形式。

- *方式2：局部动画*

在页面右上角闪现指示，特别是明亮的红色指示，肯定可以引起用户的注意。这很好，我们希望他们注意到反馈信息。但是很多时候，特别是在一个较长的或较为复杂的页面上，让用户的视线离开操作区域，会使用户分心，甚至成为一种严重干扰。

对于这种情况，我们希望反馈只出现在用户正在操作的局部区域。那么，就让我们对原来的页面做点修改吧。

这一方式只需要对我们的原有方案的源代码做少许修改。首先把包含指示的 挪下位置。原来的方式中，我们对元素进行绝对定位，并把它放到整个HTML的最后。在这种方式中，指示将按正常流程与其他元素一起顺序显示，所以我们把它移到结果容器之前。这样，当搜索正在运转时，页面看上去就是图10-3所展现的样子。这个例子的HTML文件是solution-10.1.1b.html。

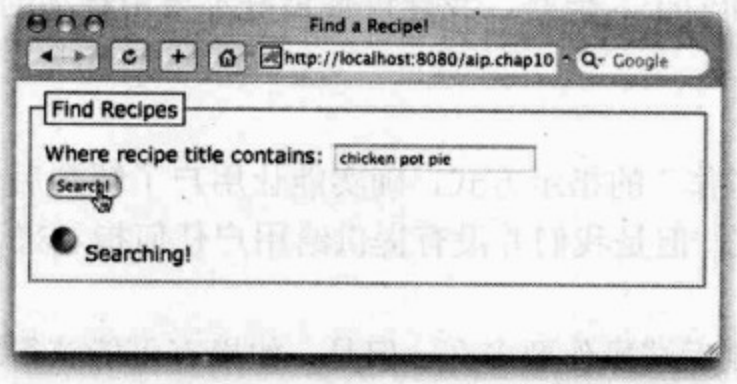


图10-3 在搜索时不停旋转

代码清单10-3显示了这一部分的标记。

代码清单10-3 将指示放在新的位置

```
      ...
      <div>
        <input type="submit" value="Search!" id="submitButton"/>
      </div>
    </form>

    <div id="processingNotice" style="display:none" >
      <img src="whirlybug.gif"/>Searching!
    </div>

    <div id="resultsContainer"></div>

  </fieldset>
```

我们删除了这一元素的CSS样式（尽管可以注意到元素最初仍是隐藏的），对文字稍做修改，并且添加了一个小动画。该GIF图像是一个原地旋转的图标，给人以正在运转的印象。

上面这些就是所有必要的改动。

3. 讨论

在本节中，我们研究了两种当Ajax操作正在运转时给予用户反馈的方式。使用Ajax能减少操作的等待时间，因为只需要从服务器取回结果而不需要读取整个页面，但是考虑到大数据库和重负载，整个操作可能仍然花上不少时间。

在搜索命令的服务器端代码中，通过人为拖延，操作会消耗5秒时间（第一次运行由于Servlet引擎要进行初始化可能会更长）。这样，我们就可以看到反馈显示出来的效果。

第一种方式我们采用了无法忽略的页面级别的指示，而第二种方式我们在原本位置加入了旋转动画。无论哪一种方式，都能让用户了解到系统正在为他们服务，而不会疑惑到底有没有在运转。这也能避免用户因不耐烦或搞不清状况重复点击按钮而让原本忙碌的数据库负担更重。

告诫　小心动画，因为很容易就走向极端。像我们的旋转小球，或者有点催眠的理发店旋转招牌那样不太惹眼的效果就足够了。你不需要让一只手臂不停地戳着页面，不需要一个建筑工人挖地不止，也不需要任何其他已被用滥颇为烦人的动画，Web上已经充斥了太多这样的东西了。记住“保持低调（keep it subtle）”的法则。符合站点主题的效果可能挺合适——或许我们的菜谱网站可以采用旋转的叉子和勺子。但是，我建议你最好不要用舞动的香蕉。

10.1.2　显示进度

上一节的两种“正在工作”的指示方式，确实能让用户了解到后台正在工作。虽然知道事情正在进行中能让用户放宽心，但是我们并没有提供给用户任何指示来了解某一时刻事情进行到了哪一步。

许多时候我们很难预测后端操作要多久。但是，如果有可能获得这一信息，那么通过进度条（progress bar）让用户了解操作预期完成时间，就是一个很好的给予用户反馈的手段。

1. 问题

我们希望能给用户比“事情正在进行中”的指示更为丰富的反馈，给予用户当前操作所处进度的指示。

2. 解决方案

在本节中我们会创建一个进度条，显示服务器端操作完成状况的百分比。这里我们只考虑客户端。如何控制服务器端操作，使其能报告进度信息，已经超出了本节的范围。事实上，这可能需要另一本服务器端线程方面的书的一整章内容。这里我们简单地假设服务器能够给予我们任何想要的信息。

上一个方案中，显示指示所需的代码和元素是直接包括在页面的标记和代码中的。因为进度条会更加复杂一些，所以我们会把它从页面中提炼出来放到一个可重用的JavaScript类中。

进入正题。

- `ProgressBar`类

把进度条的代码提取出来放到它自己的JavaScript类中，不仅令使用它的页面更加简洁，而且

确保它可以很容易地在任何其他页面中使用，甚至在一个页面中使用多次。

我们会用Prototype库的方式来创建类，所以对于ProgressBar.js，我们会从常见的类声明开始：

```
ProgressBar = Class.create();
```

类的prototype属性上是声明的其他部分，其中包括构造器所需的初始化器，请看代码清单10-4。

代码清单10-4 ProgressBar类的初始化器

```
initialize: function(parent,options) {
  this.parentElement = $(parent);
  this.options = Object.extend(
    {
      className: 'progressBar',
      color: 'red',
      interval: 2500
    },
    options
  );
  this.parentElement.innerHTML = '';
  this.parentElement.style.display = 'none';
  this.barContainer = document.createElement('div');
  this.barContainer.className = this.options.className;
  this.barContainer.style.position = 'relative';
  this.bar = document.createElement('div');
  this.bar.style.position = 'absolute';
  this.bar.style.height = '16px';
  this.bar.style.width = '0%';
  this.bar.style.backgroundColor = this.options.color;
  this.barContainer.appendChild(this.bar);
  this.parentElement.appendChild(this.barContainer);
},
```

❶ 声明初始化器，其带有与构造器一样的参数列表

❷ 将传入的options（选项）与默认值进行合并

❸ 清空父元素

❹ 创建进度条元素

方法的签名❶指定了两个参数：一个用于确定进度条所在的父元素，另一个是选项表。按照Prototype库的典型风格，父元素可以通过引用或id来确定。

用户传入的选项会与我们设定的一组默认值❷进行合并。选项包括：

- className，应用于进度条的样式类名。如果省略，会以progressBar作为默认值。
- color，进度条的填充色，默认为红色。
- interval，不断获取完成状况百分比所需的服务器轮询时间间隔，默认是2.5秒。

我们确保父元素是空的❸并且是隐藏的，然后在其中创建进度条元素❹。注意，我们用innerHTML属性来清空父元素（因为这很简单），用DOM API来创建元素（以避免在字符串中创建标记）。

我们在这里所做的DOM操作创建出与下面等价的标记：

```
<div class="nameFromOptions" style="position:relative;">
  <div style="position:absolute;height:16px;width:0%;
              background-color:fromOptions;">
```

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

外层的 用作进度条本身，其外观可由选项中的CSS类名来控制。内层的 是用选项中的颜色填充的滑动条，指示操作所完成的百分比。其宽度初始时设为0%。

一旦页面中创建一个进度条的实例，一切准备就绪只等启动。为此我们提供了start()方法。当然，如果页面代码可以启动进度条，也需要能够停止它。这两个方法列于代码清单10-5中。

代码清单10-5　ProgressBar的启动和停止

```
start: function() {
  this.bar.style.width = '0%';
  Element.show(this.parentElement);
  this.timer =
    setInterval(this._tick.bind(this),this.options.interval);
},

stop: function() {
  clearInterval(this.timer);
  Element.hide(this.parentElement);
},
```

❶ 启动进度条

❷ 停止进度条

在start()方法中，我们把显示百分比的 的宽度设为0%，以防止不是第一次启动进度条❶，然后把我们之前在构造器中隐藏起来的它的父元素给显示出来，以展现进度条。

然后我们使用setInterval()函数启动一个定时器来轮询服务器，以获得正在运行的操作的完成百分比。我们将一个内部方法作为回调函数，并将调用间隔时间设为选项表中的最终值。

我们把定时器的句柄（handle）保存在叫做timer的实例成员中。之所以需要保存，是因为我们之后能停止定时器的唯一方法要用到这个句柄。另一个值得注意的地方是，我们把回调函数绑定到了对象示例上。如果不这样做，回调函数的上下文对象就会是window而不是我们的ProgressBar的实例。

方法stop()通过对所保存的定时器句柄调用clearInterval()来取消定时器，并隐藏进度条❷。

当在start()中安上定时器之后，_tick()回调函数会按照指定的时间间隔定期调用，直到我们停止它。回调函数的责任是向服务器询问操作的完成状况并相应调整进度条的显示结果。tick()方法列于代码清单10-6中。

代码清单10-6　tick、tick、tick

```
_tick: function() {
  var self = this;
  new Ajax.Request(
    '/aip.chap10/command/GetProgress',
    {
      onSuccess: function(request) {
        self.bar.style.width = request.responseText + '%';
      }
    }
  );
}
```

由start()方法创建的定时器，其每一个tick[①]都产生一次对_tick()方法的调用。_tick()方法名的前缀下划线是一个命名惯例，表明这个方法是仅供类自身使用的内部实现细节，外部代码不应调用它。

_tick()方法中用叫做self的变量保存对实例的引用，因为我们要能从之后创建的闭包中引用该实例。记住，闭包不会包含this[②]。

然后，_tick()方法发起了一个针对GetProgress命令的Ajax请求，向服务器询问操作完成的百分比。在实现中，服务器或许会需要一个任务ID或其他的标识，表示要测量哪一个操作。不过我们只是为示例模拟一个命令，所以就不为真实世界的服务器的需要做任何假设了[③]。

这个请求的success处理器是一个闭包，它从响应中获取完成状态的百分比，然后据此调整进度条的内层 的宽度。

以上就是ProgressBar的实现，下面我们看看如何使用它。

● *在页面中使用ProgressBar*

让我们对10.1.1节中的第二种方式的代码稍作修改，使用一个ProgressBar的示例来代替简单动画。因为我们已把创建和操作进度条的代码提炼到它自己的类中，这样的修改很是简单。

一旦我们完成修改，页面显示并进行搜索时的样子看上去就应和图10-4一样。

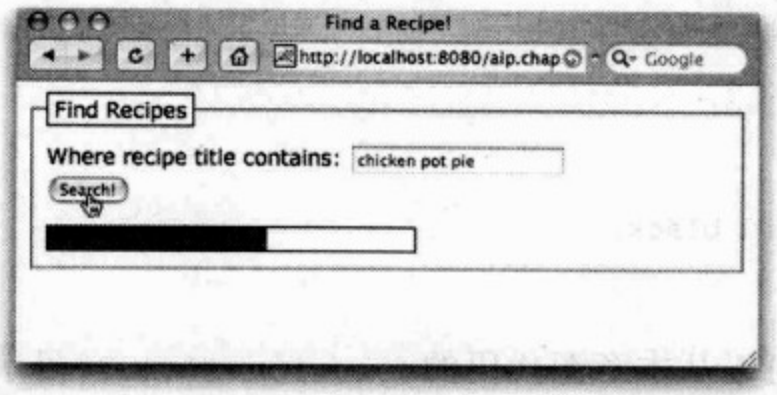


图10-4 进行中

我们首先需要导入类：

```
<script type="text/javascript" src="ProgressBar.js"></script>
```

然后，在performSearch()函数中，在请求创建后，我们把显示指示的代码换成下面的创建和启动ProgressBar实例的代码。

```
if (!window.progressBar) {
  window.progressBar = new ProgressBar(
    'progressBarContainer',
    {
      className: 'progressBar',
      color: 'blue',
```

① 定时器每过一个interval就产生一个tick。——译者注

② 指闭包中的this，并不是闭包外部的this。——译者注

③ 作者的意思是，在真实情况下GetProgress也许需要带一个参数如GetProgess?job=1，但是这个例子没有必要加入这样的假设。——译者注

```
        interval: 1000
      }
    );
  }
  window.progressBar.start();
```

这个代码片段检查我们是否已经创建了进度条，如果不存在，就实例化一个每秒进行一次轮询的蓝色进度条。在确保存在进度条之后，我们就会启动它。

修改showResults()方法以停止进度条：

```
function showResults(request) {
  window.progressBar.stop();
  $('resultsContainer').innerHTML = request.responseText;
}
```

我们把body中的指示替换为放置进度条的容器：

```
...

<div id="progressBarContainer"></div>

<div id="resultsContainer"></div>
```

最后，我们为进度条创建样式表：

```
<style type="text/css">
  .progressBar {
    position: relative;
    width: 256px;
    height: 16px;
    border: 1px solid black;
  }
</style>
```

本方案的完整代码可以从本章源代码的solution-10.1.2/目录下的solution-10.1.2.html和ProgressBar.js文件中获得。

3. 讨论

在本节中，我们通过让用户了解他们在一个要运行较长时间的过程中所处的进度，提供了一种能给予用户更精确反馈的方法。如果可以从服务器得到完成状况的准确百分比的话，这一方法就比前一节中的静态反馈更胜一筹。

如果你从服务器得到的信息无法合理地反映实际情况，那保持少一点信息总还是比不正确的信息要好一点。在某些情况下，完成百分比可能是一个估算值，但足够接近真实数值，所以仍然是可用的。在这种情形下，最好总是对完成状况进行较为保守的估算而不是过于乐观的估算。

当进度早于预期完成时，用户会喜出望外，但是如果进度僵在100%却不完成，用户就会认为我们的应用很是愚钝。

我们的ProgressBar类现在已经蛮不错了，但还是有改善空间的，你可以自己练练手：

- 在进度条中间显示百分比数字。
- 允许包含标注或其他文字。
- 给予页面创作者对内层 样式的更多控制。

迄今为止，我们所研究的方案都聚焦于如何让用户了解当前的状况。下面让我们看一下通过

预先准备好用户的所需来帮助用户的方法。

10.1.3 Ajax 请求超时

按其本身来说，XHR请求是没有超时时限的。而网络延迟极有可能会导致请求要花上很长时间才得到响应。那么我们怎样才能够检测并停止那些失控的请求？

1. 问题

我们希望能中止那些耗时超过我们指定期限的Ajax请求。

2. 解决方案

XHR缺乏指定一段时间后超时的特性。但是它有abort()方法，我们只要能找到办法在适当时候调用它，就可以中止哪些已经死在途中的请求。

这里，window对象上有两个方法可以帮到我们——window.setTimeout()和window.setInterval()。这两个方法相当类似，前者安放一个一次性的定时器，而后者安放一个不断重复的定时器。我们只希望在指定超时时限后一次性中止请求，所以我们使用setTimeout()方法。

我们的步骤是，在发起请求后启动一个定时器，如果定时器在请求完成之前就到期，就取消请求。

这听上去相当简单明了，但是要确保在我们需要的时候可以访问所需的信息，却有一些微妙之处需要考量。

让我们重温一下10.1.1节中的方案，在第二种方式里，当一个长时间的请求在服务器端处理时，我们会显示一个旋转小球。让我们研究一下如何修改这段代码，使得如果请求没有在设定的时间周期内（假定为3秒）完成，就中止连接。

为此我们会在请求产生后立即启动一个定时器。调用setTimeout()一下就可以了，不过我们也需要确保当能在请求在超时时限以内就完成时，能取消这个定时器。因此我们需要把setTimeout()返回的句柄保存在某个地方，这样就可以在请求成功所对应的处理器中访问它。

还有，我们不想采用全局变量来保存它。

代码清单10-7中的代码列出了从之前例子改写的performSearch()和showResults()函数，变更和新增的部分以粗体显示。这个修改后的HTML页面可以从文件/solution-10.1.3/solution-10.1.3.html中获得。

代码清单10-7 改写后的“正在工作”的示例

```
function performSearch() {
  $('resultsContainer').innerHTML = '';
  var request = new Ajax.Request(                    ❶ 创建请求并记录其引用
    '/aip.chap10/command/SearchForRecipes',
    {
      onSuccess: function() { showResults(request); },   ❷ 使用闭包
      onFailure: function() { showResults(request); },
      parameters:
        $H({
          terms: $F('searchTermsField')
```

```
        }).toQueryString()
    }
  );
  request.timer = setTimeout(                    ❸ 设置定时器
    function() {
      request.transport.abort();
      Element.hide('processingNotice');
      $('resultsContainer').innerHTML = 'Request timed out!'
    },
    3000
  );
  Element.show('processingNotice');
}

function showResults(request) {                  ❹ 引用请求
  clearTimeout(request.timer);                   ❺ 取消定时器
  Element.hide('processingNotice');
  $('resultsContainer').innerHTML =
    request.transport.responseText;
}
```

为了使用超时定时器，我们对代码做了四处修改。其中新增的主要代码是在向服务器发起Ajax请求之后立即调用setTimeout()❸，并将返回的定时器句柄保存于请求对象的名为timer的属性中。这样我们就不需要用全局变量来保存定时器句柄，不过也意味着需要增加一个局部变量❶来保存我们所创建的Ajax.Request实例。由于局部变量不会产生全局变量所引发的问题，所以这一改变非常值得。

定时器上的回调函数通过请求的引用找到传输器(transport)实例(实际上就是XHR对象)，调用它的abort()方法来取消请求。然后将指示隐藏起来并在结果区域中显示消息告知用户请求超时了。

以上就是请求过程超出可接受时限的情况，那么没有超时的情况呢？当处理过程在定时器到期前就已经结束时，我们就需要取消定时器，以免回调函数在实际已经完成的情况下却错误地告知用户请求超时了。

在这种情况下，我们的问题是onSuccess处理器可以得到XHR实例的引用，但是不能访问到持有定时器句柄的Ajax.Request实例，而我们需要用它来取消定时器。

我们的第一个念头可能就是把定时器句柄存到传输器（XHR）实例上而不是Ajax.Request实例上。如果我们能这样做，就可以确保在所有需要的地方（定时器回调函数和onSuccess处理器）都能访问定时器句柄。实际上，这一方式在Safari和Firefox等以本地JavaScript对象方式实现XHR的浏览器中是行之有效的。但是IE 6的XHR是一个ActiveX对象，而ActiveX对象上是不允许创建属性的。

唉！

因为我们不得不把定时器句柄附加到Ajax.Request实例上，我们需要找到一种办法能让onSuccess处理器可以获得Ajax.Request实例的引用。

为此，我们修改了原先方案的代码，把对showResult()函数的直接引用：

```
onSuccess: showResults,
onFailure: showResults,
```

替换为一个内嵌函数❷，该函数的闭包内包含了对请求的引用，这样我们就可以把请求实例传递给onSuccess处理器。

```
onSuccess: function() { showResults(request); },
onFailure: function() { showResults(request); },
```

这也意味着，showResults()函数所接收的参数将是Ajax.Request的实例，而不是原先的XHR实例。这使得我们在该函数内❹能像访问XHR实例一样访问所保存的定时器句柄。

这样，我们就可以在showResults()函数中取消定时器❺。我们也调整了引用，通过传输器获得响应的文本值。

3. 讨论

通过本方案，我们现在有能力取消那些超过我们允许时限的请求，而且在做到这一点的同时，也没有让全局变量搞乱页面。

这个例子的代码非常轻量级，所以我们没有把它像进度条代码那样提炼到一个独立的类。尽管代价可能并不算太大，但是要能够很轻松地从一个页面直接复制到另一个页面，代码还是多了点。

通过让Ajax.Request的实例捎带上定时器句柄，我们其实已经向在Ajax.Request中集成该功能的方向跨出了一小步。所以，一个令人感兴趣的代码组织方式就是把该功能完全集成到请求类中。

我们在第3章中学习了如何使用Prototype库所提供的机制来扩展类，你能利用这一知识来扩展Ajax.Request从而得到一个加入了超时特性的新类吗？

想象一下这个类，或许可以叫做Ajax.TimedRequest，我们应该可以像这样调用它：

```
new Ajax.TimedRequest(
  '/aip.chap10/command/SearchForRecipes',
  {
    onSuccess: showResults,
    onFailure: showResults,
    onTimeout: reportTimeout,
    timeoutPeriod: 3000,
    parameters:
      $H({
        terms: $F('searchTermsField')
      }).toQueryString()
  }
);
```

这样很酷吧！所以为什么不试着自己做一下呢？

下面我们把注意力转向与等待时间相关的另一个问题：毫无耐心的用户。

10.1.4 处理多次点击

要杀死一个服务器，没有什么方式能比同时用大量请求捶打它更快了。作为Ajax开发者，我们总是担心这一点，尽管我们发送的请求更小了，但我们很可能制造出比传统的Web应用多得多

的请求。

在本章中我们已经看过一些方案，通过让用户随时了解状况，明白系统正在为他们工作，来试着劝阻用户不要急躁和反复提交多次无用的请求。虽然如此，你知道总有一些用户即使已经知道状况，仍会在屏幕前叫嚷着："快一点"，并不断地点击。

而且我们要应对的不仅仅是没有耐心的用户。在桌面应用中，用户习惯用双击来激活项目[①]。必须承认，就像我们的Web应用一样，桌面应用中的按钮多数也是用单击调用的，但是有些习惯很难改掉，用户群体中，总有一定比率的用户会残忍地双击我们的按钮。

1. 问题

用户可能毫无耐心、心不在焉、手笨眼拙或三者皆有可能。我们需要一种方法能在操作进行时防止对同一个操作提交多次请求。

2. 解决方案

好消息是我们可以使用一个相对简单的技巧防患这些用户于未然：如果当时我们已经在处理对按钮的第一次点击，只要从源头上禁止点击——禁止按钮。

让我们再一次从10.1.1的"旋转小球"方案开始，使它能防止多次点击。图10-5显示了搜索正在进行时的最终页面。

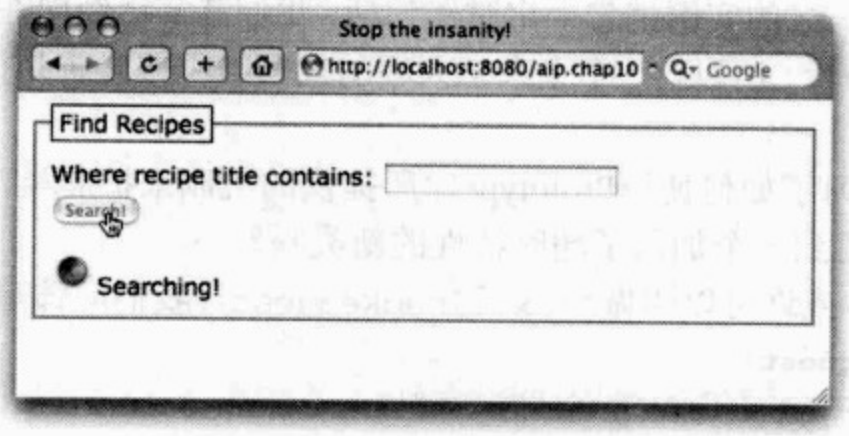


图10-5　试着双击！试验一下！

代码清单10-8中的代码（可从文件/solution-10.6/solution-10.6.html中获得）列出了原来方案修改后的代码片段，改动和新增部分以粗体显示。

代码清单10-8　停止疯狂的点击！

```
function performSearch(button) {
  button.disabled = true;                          ❶ 禁用按钮
  $('resultsContainer').innerHTML = '';
  var request = new Ajax.Request(
    '/aip.chap10/command/SearchForRecipes',
    {
      onSuccess: function() { showResults(request); },
      onFailure: function() { showResults(request); },
                                                   ❷ 使用闭包传递实例
```

① 例如在Windows桌面上双击图标来启动程序或打开文档。——译者注

```
        parameters: $('searchForm').serialize(true)
        trigger: button
      }
    );
    Element.show('processingNotice');
}

function showResults(request) {
    request.options.trigger.disabled = false;
    Element.hide('processingNotice');
    $('resultsContainer').innerHTML =
      request.transport.responseText;
}

...

<form id="searchForm" onsubmit="return false">

...

<input type="button" value="Search!"
       onclick="performSearch(this) "/>
```

❸ 把button加入选项

❹ 启用按钮

❺ 禁止表单提交

❻ 将按钮传递给处理器

变化实际上始于页面底部，在那里我们修改了按钮元素❻。我们不再用submit类型的input，而是用button类型的input，这样按钮就不再触发表单提交动作。我们用onclick事件处理器来触发performSearch()函数，并传入按钮的引用。

我们之前的方案就可以用这个技巧，现在更是必须这样做。因为我们已经给按钮加上onclick事件处理器，如果不替换submit，点击按钮就会产生竞态：onclick事件处理器和表单提交两者都被启动。

因为现在直接由按钮来触发onclick事件处理器，表单的onsubmit事件处理器❺就被改为直接返回false，从而防止表单意外提交（如果用户在文本框中按回车，有可能会出现这样的意外）。

onclick事件处理器在调用函数performSearch()时，也传入了触发事件的按钮的引用。在没有耐性的用户有机会再次点击该按钮之前，performSearch()函数就会立刻禁用引用所指向的按钮❶。

假如这是一个传统Web应用，我们就只需要做这些改动。当响应返回时页面会发生刷新，确保我们会再次得到可用的按钮。但是这是一个Ajax应用，我们不打算就此刷新页面，所以需要确保当可以再次发起搜索的时候，重新启用按钮。

在请求的onSuccess处理器中❷，我们想重新启用按钮，但是我们没有按钮的引用。我们也不希望把按钮写死在代码中，因为可能有多个不同的按钮都可以发起搜索动作[①]。

如果你记得我们最近的一个方案，也有类似的保存定时器句柄的需求，我们是把它直接附加在Ajax.Request实例上，然后用闭包来确保onSuccess处理器可以访问该实例。这里我们也会用类似的手法，不过比前面更巧妙一点。

在超时时限的方案中，在请求实例创建之后，我们将计时器句柄直接附加到实例之上。在本方

① 另一种方式是把showResults()函数放到performSearch()方法中，作为闭包，可以直接访问传入performSearch()的button。——译者注

案中，我们则把指向按钮的引用记录在传递给Ajax.Request的选项表里，其属性名为trigger❸。

那是什么？我们怎么能这样胡来呢？Ajax.Request并不知道任何叫做trigger的选项啊。

其实Ajax.Request类处理其选项的方式与我们在本章中创建的类中处理选项表的方式是大体相同的，都会把传入的对象与包含默认值的内部对象进行归并。归并之后得到的结果对象包含两个对象所有属性的并集。所以，我们在传入Ajax.Request的选项表上放置的任何属性都会被传递到最终的选项对象里。

既然Ajax.Request并不知道这些额外的选项是干嘛的，就不会访问它们，甚至不会知道它们的存在。但是我们可以在任何可以访问到请求实例的地方引用它们。

所以，当onSuccess处理器showResults()被调用并将请求的引用作为参数传入时，我们就可以使用藏在选项表中的按钮引用来重新启用按钮❹。

瞧吧！没有全局变量，没有那种写死的假定性的代码。

3. 讨论

现在我们已经找到了方法，防止在操作已经运行的情况下因多次点击发起同样的操作，那么我们的服务器是不是就高枕无忧了呢？也不是。我们仍然需要确保服务器代码能应付多个请求，特别是对于发起重复请求可以有灾难性结果的情况，例如数据库增删记录的操作。

记住，虽说我们的用户可能既没耐性又笨拙，但同时他们也可能过分聪明和存心不良。服务器代码必须总是保持警戒。所以像本节中所描述的这类技巧并不会让我们在写服务器代码的时候有机会偷懒，但它可以帮助我们保持网络流量不至于因一些偶发情况而升高。

我们也探索了如何通过传递给Ajax.Request的选项表携带触发请求的按钮引用，而避免使用全局变量，或是把引用写死在代码中。作为练习，你可以回到超时时限的方案，试着使用这一技巧来改进它。

10.2　预防和检测输入错误

用户受挫的另一个原因是数据输入错误。

用户在输入数据的时候免不了犯错。从打字失误，到确实搞不清要输入什么，什么都可能引发错误。本节我们会考察一些方案以应对这些输入错误。

我们首先会看一种方法，能让我们预先准备好用户所需的帮助，确保他们在输入任何内容之前先理解到底要输入什么。然后我们会看看，在输入发生错误的时候，怎样以尽可能有用的方式来提示用户。

10.2.1　主动显示上下文帮助

多数帮助系统会在用户就某个操作或主题请求帮助之后显示对应的帮助信息，也就是说，用户能获得帮助——如果用户知道怎么获得帮助的话。然而，有些应用程序的帮助藏得太深以至于用户从来就不知道它们的存在。

另一些应用的帮助系统可能根本没有提供上下文相关帮助。它们不是给用户提供与当前所从事任务相关的帮助信息，而是把一份列着所有可用帮助主题的庞大清单扔给用户，让他们自己搜寻和猜测哪些主题可能包含有用的信息。

在本方案中我们会考虑一种方法，以一种主动式的、上下文相关的方式检索和显示与表单控件相关联的帮助信息。

1. 问题

我们想在用户访问表单的每个控件时检索和显示出与该控件相关联的帮助文本。帮助应该是主动的，并且不应该干扰用户填写表单的正常流程。

2. 解决方案

像前面的方案一样，我们不是把帮助系统的代码直接写在页面里，而是把它提炼出来放入一个JavaScript类。这一方式不仅降低了页面代码的复杂程度，而且也让我们得到一个可重用的组件，能在许多其他页面中反复使用。

我们的这个类可以叫做HelpConveyer，它会对指定表单所包含的所有元素进行改造，使得每当元素获得焦点时，就联络服务器，获取与该元素相关联的帮助文本。

- HelpConveyer类

像往常一样，我们使用Prototype库提供的机制来声明我们的JavaScript类：

```
HelpConveyer = Class.create();
```

这个类只包含一个初始化器，定义于prototype属性之上，如代码清单10-9所示。

代码清单10-9 HelpConveyer类的初始化器

```
initialize: function(targetElement,form,url,paramName) {          ❶ 接受初始化参数
  this.target = targetElement;
  this.form = form;
  this.url = url;
  this.paramName = paramName;
  var conveyer = this;
  $A(this.form.elements).each(                                    ❷ 遍历表单元素
    function(control) {
      if (control.name != undefined && control.name != '') {
        control.onfocus = function() {                            ❸ 指定onfocus
          var paramHash = {};                                       事件处理器
          paramHash[conveyer.paramName] = control.name;
          new Ajax.Updater(                                       ❹ 读取帮助文本
            conveyer.target,
            conveyer.url,
            {
              method: 'get',
              parameters: paramHash
            }
          );
        };
      }
    }
  );
}
```

这一由构造器调用的初始化器需要四个参数❶：用于显示帮助文本的目标元素、待改造元素所属表单的引用、查询帮助文本时调用的URL以及用于传递要检索的帮助主题的URL参数名。按

照我们的目的[①]，可以用表单字段名作为帮助主题。

在将四个参数值保存于实例成员后，我们创建了一个名为conveyer的局部变量，并将当前实例的引用存于其中。至此，你应该可以认出这是后面要使用闭包的信号。

我们使用Prototype库的each()方法遍历表单的所有元素❷。如果某个元素的name属性不为空，就指定一个函数（及其闭包）作为该元素的onfocus事件处理器❸。

这个内嵌的事件处理器函数会创建一个名为paramHash的对象，其中包含了帮助主题参数，在创建Ajax请求时它会被转换为查询字符串。

然后，我们用Prototype库的Ajax.Updater服务❹从服务器读取帮助文本并将其填入目标元素。

一个小小的初始化器就能得到很大的回报——现在表单的每个元素都能读取并显示出它自己的帮助信息了。那么让我们看看如何在页面中使用这个类。

● 使用HelpConveyer类

因为我们已经把所有困难工作都抽取到了HelpConveyer类中，所以在页面中使用起来是很简单的，只需要以适当的参数生成一个类的实例就可以了。

考虑一下我们的菜谱分享应用中供会员输入或编辑菜谱的页面，你可以想象一下，有不少字段需要填写。表单的初始设计包括下列字段：

- 标题
- 来源
- 最少份数
- 最多份数
- 配料清单
- 烹饪步骤

我们的页面显示出来就像图10-6所示。其中最后两个字段比较令人感兴趣，它们会用多行文本框。对于配料清单，多行文本框里输入的每一行都表示一种配料。对于烹饪步骤，多行文本框中以空行分割的每个段落都表示一个步骤。

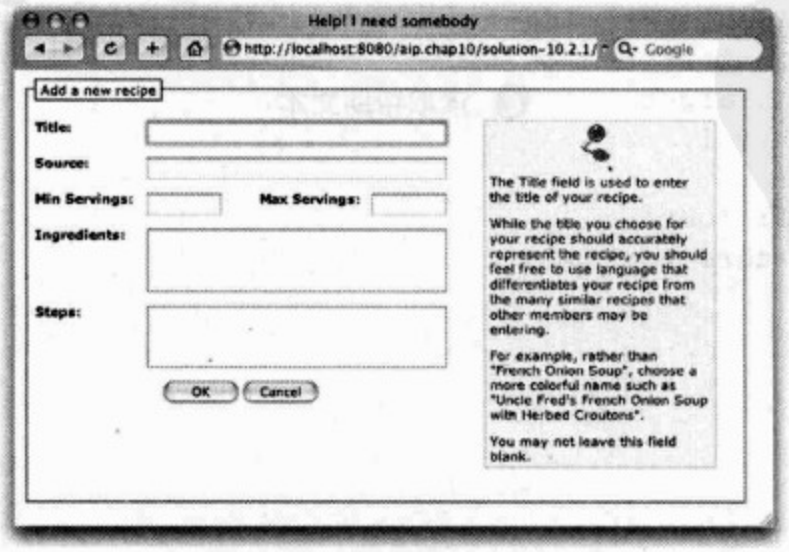


图10-6　带有帮助的菜谱表单

① 即获得与表单字段相关的帮助文本。

这对于会员来说并不是显而易见的，所以我们的主动式帮助系统对于这个表单来说应该会很有用，表单代码见代码清单10-10。

代码清单10-10 菜谱表单

```
<div id="formControls">
  <form name="recipeForm">
    <div class="line">
      <label>Title:</label>
      <input type="text" name="title" id="titleField"/>
    </div>

    <div class="line">
      <label>Source:</label>
      <input type="text" name="source" id="sourceField"/>
    </div>

    <div class="line">
      <label>Min Servings:</label>
      <input type="text" name="minServings"
             id="minServingsField" />
      <label id="maxServingsLabel" >Max Servings:</label>
      <input type="text" name="maxServings"
             id="maxServingsField"/>
    </div>

    <div class="line">
      <label>Ingredients:</label>
      <textarea name="ingredients" id="ingredientsField"
                rows="4"></textarea>
    </div>

    <div class="line">
      <label>Steps:</label>
      <textarea name="steps" id="stepsField"
                rows="4"></textarea>
    </div>

    <div align="center" id="buttonBar">
      <input type="button" value="OK" name="okButton"/>
      <input type="button" value="Cancel" name="cancelButton"/>
    </div>

  </form>
</div>
```

表单的每一行即为一个 元素围起的区域，每个 包含一个<label>元素和一个<input>元素，只有一处，两个<input>元素及其标签（<label>）被并入了一行之中。我们也会用CSS规则（定义于页面首部）对表单进行合适的布局。

我们还需要一个地方显示与每个字段相关联的帮助文本。如代码清单10-11所示，我们使用一系列嵌套的 元素界定该区域，帮助文本会被插入到最内层的 。之所以使用嵌套，

只是为了便于应用一些有趣的样式。

代码清单10-11　帮助文本显示区域

```
<div id="helpContainer">
  <div id="helpSticky">
    <div id="helpDisplay">
    </div>
  </div>
</div>
```

表单和帮助文本容器并排于页面之上——表单在左，帮助在右——这样用户在填写表单的同时也可以看到帮助文本。我们也希望帮助文本区域能醒目一点，所以我们用了一点巧妙的样式以及一个简单的GIF图像，使得它看上去就像一个以图钉钉在页面上的黄色小贴纸。（我说不出为什么你要用图钉把注释钉在那里，不过它看上去确实蛮酷。）

我们就不详细描述创造出这一效果的CSS了，不过你可以从本章源代码的/solution-10.3/solution-10.3.html文件中获得那些细节。

注意，与标题字段相关联的帮助文本已经显示了出来，因为页面加载时标题字段就获得了焦点。我们来看看怎么做吧。代码清单10-12显示了页面的onload事件处理器。

代码清单10-12　在onload中装载帮助信息

```
window.onload = function() {
  new HelpConveyer(                          ❶ 创建HelpConveyer的实例
    'helpDisplay',
    document.recipeForm,
    '/aip.chap10/command/GetHelp',
    'topic'
  );
  document.recipeForm.title.focus();         ❷ 让第一个字段获得焦点
}
```

可以看到，我们完全用不着做太多工作。首先我们创建一个HelpConveyer的实例❶，确定显示帮助文本的目标区域、菜谱的表单、返回帮助文本的URL以及URL资源所需的帮助主题的参数名。

接着我们让表单的第一个字段（标题）获得焦点❷，之后HelpConveyer类中所写好的代码便会帮我们做好所有剩下的事情！

当我们切换到表单的不同字段时，帮助文本会自动更新为与该字段关联的帮助文本。图10-7展现了切换到配料清单字段后的页面。

这真是个对用户非常友好的方式，不是吗？

3．讨论

本节中我们提出了一种方法，以主动方式为用户提供上下文相关帮助。

用户不需要用额外操作来请求帮助，也不需要去操心帮助是否可用以及如何获得它。而且这个方法一点儿也不唐突，用户完全不会因为要查看帮助而被迫打断文本输入流程。

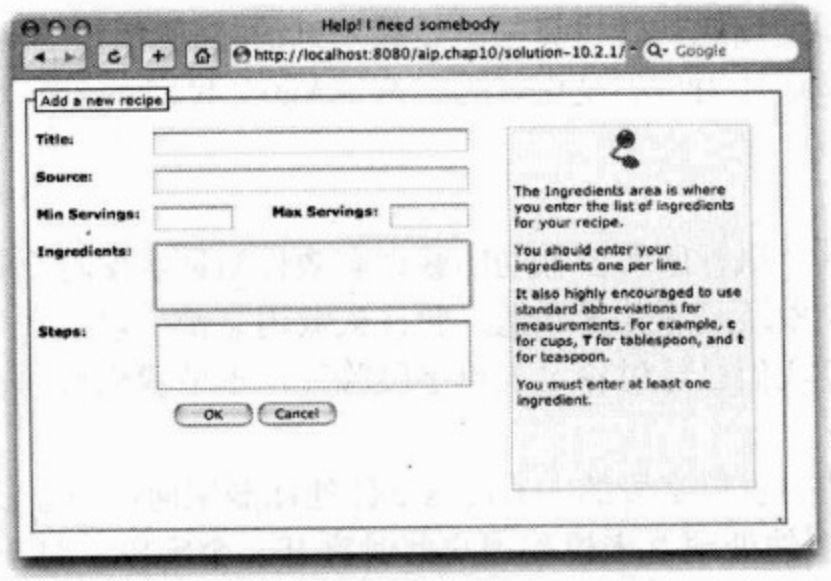


图10-7 输入配方清单

本方案中，我们使用的命令URL将接受一个帮助主题（值为字段名称）作为参数。当前的实现是很简单的，直接在本地目录中以主题名称寻找一个HTML文件，然后返回该文件。

我们所提出的方法已经很不错了，不过总是有改进的空间。

在输入过四五个菜单之后，我们的会员大概就会是填写表单的老手了，这些帮助文本也可能变得没什么用处，反而有点让人分心。我们可以考虑实现某种方法，允许会员在成为使用系统的行家之后关闭帮助。

本方案中我们也做了一个简单的假设：每个帮助主题用字段名就可以唯一确定。然而在稍具规模的真实应用中这就很可能导致命名冲突。所以最好对主题的索引方式加以改进，或许把表单名和字段名组合在一起就可以了。

通过提供主动式帮助，我们减少了会员在填写表单时可能发生的错误，当然我们不能指望完全没有错误。下面让我们看一下如何检查错误。

10.2.2 对表单输入项进行有效性验证

假定用户都是笨蛋肯定不是什么好主意，假定他们不会犯错误同样也是个馊主意。无论是因为不明白应该做什么（前一节的上下文帮助机制应该会大大减少这方面的因素），还是因为注意力不集中，或是由于一时疏忽，甚至仅仅是患有“肥大手指综合症”，都有可能造成表单字段的输入无效。

在传统Web应用中，可以在客户端对表单输入的数据做一点有限的有效性验证，而在服务器端则应该始终对数据进行有效性验证（无论是否执行了客户端验证）。

此类应用中的客户端有效性验证只能执行简单的检查，这些检查不需要用到那些只有在服务器上才可用的上下文和信息。举例来说，可以用简单的JavaScript来校验必填字段是否还空着没填，或是数值字段是否包含有效的值。但是其他一些有效性检查，诸如检查邮政编码是否和地址匹配，信用卡卡号是否有效，或是需要用到客户端没有的信息，就必须于表单提交之后在服务器上执行验证。

这类验证过程往往让用户痛苦不堪，除非返回给用户的表单没有丢失任何数据，输入错误也

能被清晰地标识出来。即便如此，这也算不上是最友好的方式。

一有输入错误就立即通知用户不是更好吗？有了Ajax，我们不必等待表单提交也可以执行服务器端检查。

1．问题

我们想对表单字段逐一执行服务器辅助的数据有效性验证。我们希望方案是可重用的，并且希望无论检查是在客户端做还是服务器端做，验证失败消息都能以一种一致的方式呈现给用户。

之前在第5章和第6章我们已经讨论过表单字段验证，本节我们再一次涉及这一重要的主题。

2．解决方案

在上一节中，我们使用表单字段的onfocus事件处理器来向用户显示主动式帮助，类似地，我们也可以使用onblur事件处理器来检测用户何时离开一个字段。对于只能在服务器上执行有效性验证的字段，我们会建立一种机制来处理有效性验证和错误报告。

按字段逐一编写Ajax事件处理器函数是很简单，但是不够聪明。我们希望解决方案是可重用的，我们不想老是把同样的代码从一个页面剪切和粘帖到另一个页面。

我们要更进一步，不过你最好有心理准备，这会是本章中最为复杂的方案之一。而且就算花了很多精力，我们也还无法得到像我们最终希望的那样健壮的（robust）有效性验证框架，但是我们将会得到一个构建框架的坚实基础。

系好安全带，我们出发了。

- FieldValidator类

我们会定义一个叫做FiledValidator（字段有效性验证器）的类来满足一组相当苛刻的需求：

- 对每一个需要进行有效性验证的字段都会创建一个FieldValidator的实例，它能以一致的方式处理验证错误。
- 有效性验证器通过插入式的“校验器（verifier）”工作，校验器可以是客户端的操作也可以是服务器端的操作。用户不会知道到底验证是在哪里进行的。
- 需要提供一组常用的客户端校验器。页面创作者也可以插入自定义的校验器。
- 当有验证错误，字段的外观会改变，验证失败消息会呈现给用户。
- 当更正了字段并通过验证后，字段的外观会恢复成原来的样子，验证失败消息也会清除。
- 这些变化是实时的，用户一离开字段就会触发。

唔！这笔单子可真不小。让我们从定义构造器开始吧，像平常一样：

```
FieldValidator = Class.create();
```

代码清单10-13是这个类的初始化器。

代码清单10-13　FieldValidator的初始化器

```
initialize: function(field,verifier,options) {
  this.field = $(field);
  this.verifier = verifier;
  this.options = Object.extend(
    {
      errorContainer: 'errorContainer',
```

❶ 声明初始化器

❷ 将选项与默认值进行归并

```
        errorClassName: 'fieldInError',
        paramName: 'value'
      },
      options
    );
    this.errorContainer = $(this.options.errorContainer);
    this.field.validator = this;
    this.field.onblur = function() {
      this.validator.validate();
    }
  },
```

❸ 指向错误消息容器的引用

❹ 指向字段有效性验证器的引用

❺ 在失去焦点的时候进行有效性验证

初始化器接受三个参数❶：要装备有效性验证器的元素的ID或引用，用于对字段数据进行有效性验证的校验器以及选项表。

校验器可以是执行客户端有效性验证的JavaScript函数，也可以是执行服务器端有效性验证的资源URL。如果是JavaScript函数，那么会在适当的时候调用，并会把字段的引用作为参数传递给它。如果字段通过了有效性验证，函数应该返回null；如果验证失败，则应该返回一个消息字符串，解释失败的原因。

FieldValidator类内建有两个客户端校验器，提供了两种最常用的有效性检查。我们稍后会讨论它们，现在让我们继续看初始化方法。

如果指定了服务器端有效性验证的资源URL作为校验器，进行有效性验证时就会实例化一个指向该资源的Ajax请求，并把要验证的字段值作为参数传递给该资源，参数名字则由选项表中的选项确定。

在将字段引用和校验器保存到实例变量之后，页面创作者所传入的选项会与默认值进行归并❷。所支持的选项包括：

- errorContainer，用于显示有效性验证失败信息的容器，可以是ID或引用。
- errorClassName，有效性验证失败时应用于字段的样式类名。
- paramName，将字段值传给服务器端的有效性验证资源时所使用的请求参数名。

用于显示错误信息的容器确定之后，指向该容器的引用会存储在成员变量中❸，便于之后使用。

字段元素本身也会添加一个指向该字段的有效性验证器[①]的引用❹[②]，当用户切换焦点离开这个字段时，字段的onblur事件处理器函数❺就会通过这个引用来调用validate()方法。一旦初始化完成，字段就准备妥帖，无论何时触发onblur事件，处理器都会执行有效性验证。

前面我们提到过内建的校验器。代码清单10-14显示了FieldValidator类中所提供的这两个校验器。

代码清单10-14 内建校验器

```
FieldValidator.verifier.NotBlank = function(field) {
  if ($F(field) == '') {
```

① 也就是this所指向的对象实例。——译者注

② 这构成了一个双向引用：有效性验证器上的field属性指向字段元素，而字段元素上的validator属性指向有效性验证器。——译者注

```
      return 'The ' + field.name + ' field cannot be blank';
    }
    else {
      return null;
    }
  }

  FieldValidator.verifier.IsNumeric = function(field) {
    if ($F(field) == '' || isNaN(new Number($F(field)))) {
      return 'The ' + field.name + ' field must be numeric';
    }
    else {
      return null;
    }
  }
```

第一个校验器会在字段值为空白时报告验证失败，第二个则会确认字段值是否是数字。

注意，这些函数并没有被作为类的prototype属性的一部分来声明。从下面这个创建验证器实例的样例代码中可以看到，引用内建校验器函数并不需要FieldValidator的实例。

```
new FieldValidator('someField',FieldValidator.verifier.NotBlank);
```

一旦装备有验证器的字段上触发了onblur事件处理器，validate()方法就会被调用。这一方法担负着类的主要工作，代码清单10-15是该方法的定义。

代码清单10-15　担负着主要工作的validate()方法

```
validate: function() {
  this.clearError();                                          ❶ 清除错误信息
  if (this.verifier instanceof Function) {                    ❷ 由JavaScript函数进行校验
    var message = this.verifier(this.field);
    if (message != null) this.markInError(message);
  }
  else {                                                      ❸ 发起Ajax请求
    var validator = this;
    var paramHash = {};
    paramHash[this.options.paramName] = $F(this.field);
    new Ajax.Request(
      this.verifier,
      {
        parameters: $H(paramHash).toQueryString(),
        method: 'get',
        onSuccess: function(transport) {
          if (transport.responseText != '') {
            validator.markInError(transport.responseText);
          }
        }
      }
    );
  }
},
```

方法首先调用clearError()来清除错误信息❶。如果字段上一次验证失败，就可以回复到未失败的状态，过会儿我们会详细看一下。

接着validate()方法根据校验器的种类决定执行哪一种操作。如果校验器是JavaScript函数❷，就调用它，返回的若不是null（表示有错误），就调用方法markInError()把字段置于错误状态。

如果校验器是URL，就发起一个针对该URL的Ajax请求❸，并把字段值传给它。若请求得到的响应不是空字符串（表示有错误），也会把字段置于错误状态。

无论是客户端检查还是服务器端检查，只要字段有效性验证失败，就都会调用代码清单10-16所列出的markInError()方法。这样就确保了不管是在哪里进行检查，验证失败消息总能以一致的方式呈现给用户。

代码清单10-16 标记无效字段

```
markInError: function(message) {
  Element.addClassName(this.field,
                       this.options.errorClassName);
  this.errorMessageElement = document.createElement('div');
  this.errorMessageElement.appendChild(
    document.createTextNode(message));
  this.errorContainer.appendChild(
    this.errorMessageElement);
  Element.show(this.errorContainer);
},
```

方法markInError()执行两个操作。首先，在字段的样式类名列表中添加一个（在选项中所设定的）表示验证失败的样式类名，这会改变字段的外观，这样页面创作者就可以决定字段在有效性验证失败后应该如何显示。

其次，将一个包含有效性验证失败信息的 元素加入到为此用途所准备的消息容器里。容器最初是隐藏的，所以也要确保把容器显现出来。

我们的类的最后一个方法，如代码清单10-17所示，清除了字段的错误状态。它删除了字段上的验证失败的样式类，也从消息容器中去掉了错误消息。

代码清单10-17 清空失败字段

```
clearError: function() {
  Element.removeClassName(this.field,
                          this.options.errorClassName);
  if (this.errorMessageElement) {
    this.errorMessageElement.parentNode
      .removeChild(this.errorMessageElement);
  }
}
```

这个类是个很好的例子，说明我们为什么先得把代码组织成类的形式。要是这类代码在我们要用到的每一个页面上都放一份的话，那就得让人抓狂了！

你可能已经注意到贯穿本章乃至本书其他部分中所有方案的主题了。那就是：“别让页面粘乎乎！”所谓“粘乎乎”，也就是对不必要的复杂性（unnecessary complexity）的最准确和科学的

称呼。

事实上，下面我们就可以看到我们的字段有效性验证器是怎样避免让页面“粘乎乎”的。

● 使用FieldValidator类

让我们想象一下菜谱共享社区应用中的注册表单，当网站访问者想成为网站会员时就要填写它。这个表单跟我们上一个方案中用于输入菜单的表单差不多，也包括了上下文帮助的功能，不过这个表单只有三个字段：会员名、电子邮件地址、年龄。

头两个字段是必填的，其中电子邮件字段的值必须是格式有效的电子邮件地址。年龄字段是选填的，如果要填写，也必须是数字。

可以在本章源代码中的/solution-10.4/solution-10.4.html文件中获得这个例子的HTML页。当你没有填写会员名字段，填写了虚假的电子邮件地址，并切换到年龄字段时，页面看上去如图10-8所示。（注意，本方案的实际代码中，字段错误时的样式是和错误消息框一样的浅黄色背景，但是在灰度屏幕截取中无法清楚地显现出来，所以我们为这个截图使用了更加尖锐的红色。）

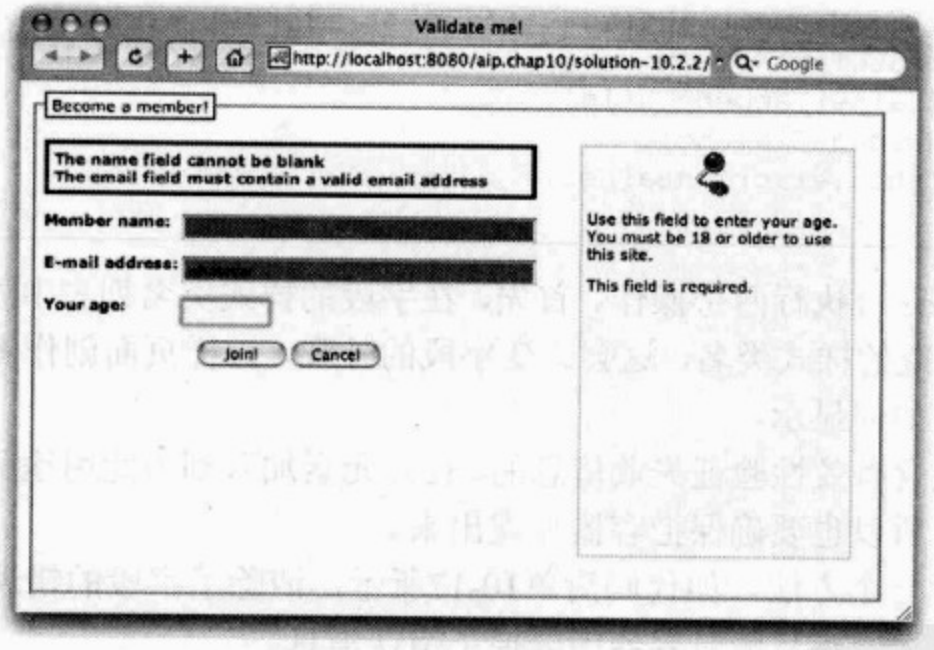


图10-8　不要忘记填写名字

为了使用FieldValidator类来对这些字段的输入进行有效性验证，我们给页面加上了onload事件处理器，如代码清单10-18所示。

代码清单10-18　为会员注册表单装备上有效性验证功能

```
window.onload = function() {
  new FieldValidator('nameField',
                     FieldValidator.verifier.NotBlank);
  new FieldValidator('emailField',
                     '/aip.chap10/command/VerifyEmail');
  new FieldValidator('ageField',
                     FieldValidator.verifier.IsNumeric);
  new HelpConveyer(
    'helpDisplay',
```

```
        document.memberForm,
        '/aip.chap10/command/GetHelp',
        'topic'
    );
    document.memberForm.name.focus();
}
```

在这个事件处理器中，我们为表单中的每个字段都创建了一个有效性验证器的实例。会员名和年龄字段使用的是内建的客户端校验器，而电子邮件字段则使用服务器端资源来检查电子邮件地址的格式。

这样就行了。事件处理器的其余部分和上一个方案中的一样，创建了一个HelpConveyer类的新实例，然后随着页面加载让会员名字段获得焦点。

还有一个额外的任务是增加用于显示有效性验证失败消息的容器。我们在表单的顶部这样定义它：

```
<div id="errorContainer" style="display:none;"></div>
```

我们为它设定了以下样式，确保不会注意不到它：

```
#errorContainer {
  border: 3px outset maroon;
  background-color: #ffffcc;
  padding: 4px;
  color: maroon;
  font-weight: bold;
  margin-bottom: 12px;
}
```

3. 讨论

我们在本节开始时说过，要为字段有效性验证框架打下了良好基础，我们已经基本做到了，但是还有许多有待改善之处。首先，我们最好有能力检测出是否有任何一个字段处于错误状态，这样我们就能阻止表单提交，直到所有字段都验证通过。

我们也可以加入更多的内建验证器。比如说验证数字的取值范围。

也有这样的意见，把错误消息组合到一块地方（我们使用了表单的顶部）对于较长的表单来说不太适合。所以可以考虑一下如何能调整这个类，让验证消息显示在相关字段的旁边。

不过，迄今为止，最大的缺点可能是这个类以及上一个方案中的HelpConveyer，要独占onblur和onfoucs事件处理器，页面创作者无法为其他目的使用这两个事件处理器。为简单起见，使用DOM Level 0事件处理模型直接导致了这一后果。你可以利用在第5章所累积的DOM Level 2事件模型的知识，修改这些类，以去掉这个限制。

我们解决了诸多棘手问题，就是为了确保无论是在客户端进行检查还是在服务器端进行检查，任何有效性验证失败都能以一致的方式报告给用户。这很好，但是未必总能奏效。如果服务器端验证会有很长的等待时间，那用户就会意识到差别。特别是如果用户已经切换到好几个字段之后，前面的字段才忽然显示出错误状态，那就非但没有帮助反而让人心烦意乱。所以，在网络等待时间较长不适用这类步进式验证的场合，最好只进行客户端检查，而留待表单提交之后再进行服务器端检查。好了，我们已经学习了如何消除源自于数据输入问题的混乱，下面让我们看看

如何能把其他一些混乱源头也消灭于萌芽状态。

10.3　维护焦点和分层顺序

我们早已说过，几乎没有什么比让用户限于迷惑之中更能打击用户了。借助已掌握的Ajax利器，我们可以给予用户动态的用户界面，而在本节中，我们就会研究一些方法，避免让用户在面对这类动态用户界面时产生不必要的迷惑和挫折。

10.3.1　维护焦点顺序

在“点击”(point-and-click)的世界中，焦点管理这个话题可能并不那么令人感兴趣。但是在Ajax世界中，用户界面可以非常动态，因此焦点管理对于应用程序的优使性来说就变得至关重要。

假定基于某种特别理由，表单字段或其他用户界面元素需要打乱顺序，隐藏显现，甚至由用户控制移动。在这些场合，input元素的Tab次序（按下Tab键访问元素的先后顺序）可能会乱七八糟。

而当Tab顺序不再符合直觉，我们的应用程序就会变得相当难以使用。特别是对于那些因为生理上的限制或偏好需使用键盘而非鼠标的用户来说，就更是如此。

1．问题

我们希望用户界面元素的Tab次序始终符合直觉，即使这些元素经常移来移去。

2．解决方案

在本节中，我们将使用有点不合常规的用户界面元素：带有图片背景的 元素。通常，我们的界面元素会是文本框之类的控件，但是文本框的样子大同小异，不太容易区分，所以为直观起见，我们会使用更具特征的界面元素，这样我们很容易就能看到移动效果。

不幸的是，这将会是本章中唯一不能在Safari中正确执行的示例，因为Safari当前不允许 元素获得焦点。[①]

我们的方案页面上总共会有六个元素顺序排列，每一个元素显示不同的图像，这样我们一眼就能分辨。这些元素可以重新排列次序，而我们会看看要在脚本中做些什么才能让重新排列之后的元素的Tab次序仍然保持符合直觉。一开始，我们的页面看上去就像图10-9所示。

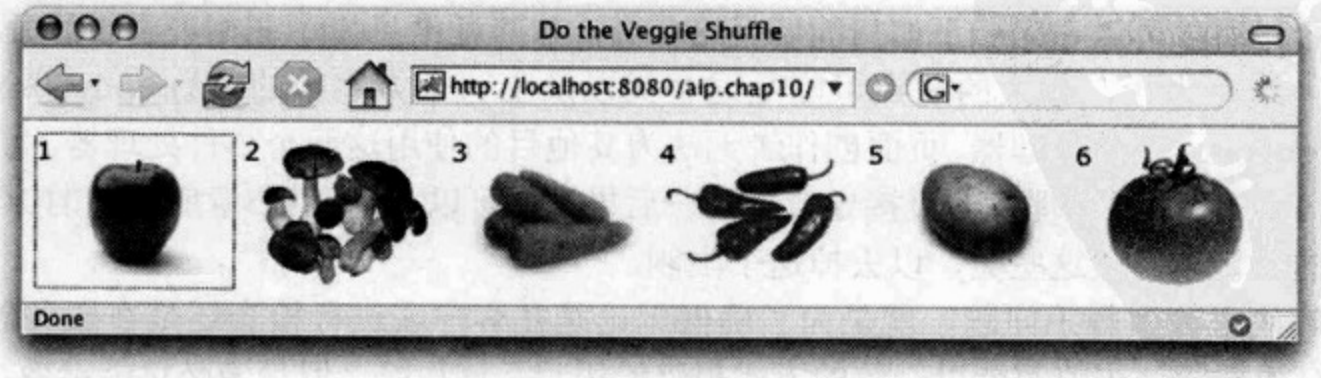


图10-9　排列整齐的美味果蔬

① 本示例所用技巧是有争议的。实际上这个示例在Opera中也不能正确执行（我们测试的是Opera 9.23 for Windows），因为Opera虽然允许 获得焦点，但是并不支持 上的tabindex（你通常只能通过鼠标点击来让 获得焦点），并且 上也不会有像虚线框那样的视觉指示。——译者注

每一个果蔬都有一个图片，左上角则有一个数字显示出它的Tab次序。当页面加载的时候，每项的Tab次序从左到右定为1到6。最初焦点在苹果上（我们可以从浏览器在元素周围勾画的虚线框了解到这一点），如果连续按Tab键，我们会看到蘑菇，接着是胡萝卜，再接着是辣椒，逐个获得焦点，这跟我们预料的一样（更重要的是，跟我们的用户预料的一样）。

我们也对每一项都做了处置，当它获得焦点时就可以用左右方向键来修改它的次序。按左方向键会让项目跟它左边那项交换位置，按右方向键则让项目跟它右边那项交换位置。在耍了一阵之后，我们可能最终会得到类似图10-10所显示的样子。

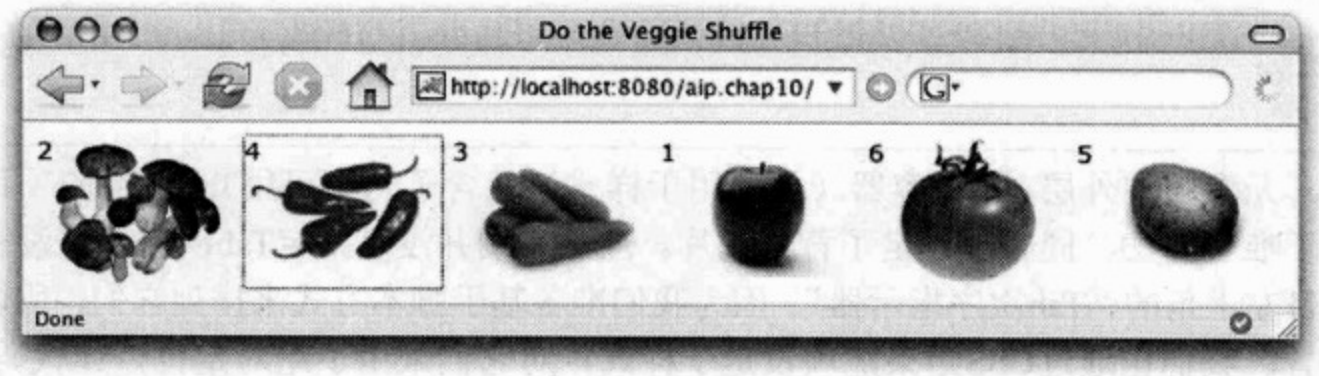


图10-10 什锦果蔬大杂拌儿

现在我们把所有项的次序完全打乱了，Tab次序因此也变得一团糟！当焦点在苹果上的时候，如果连续按Tab键，会把我们带回到蘑菇上，接着跳过辣椒到了胡萝卜上，接着又回到了辣椒上，接着到了土豆上……真是一头雾水啊！

这说明我们必须在移动项目位置的同时也改变它们的Tab次序，使得Tab次序保持一个规则而合乎直觉的顺序。所以在果蔬的顺序打乱之后，我们真正想得到的是像图10-11那样的效果。其Tab次序与果蔬项的物理顺序相匹配，这样当我们按下Tab键进到下一项（或者按Shift+Tab回到上一项）的时候就不会得到出乎意料的结果了。

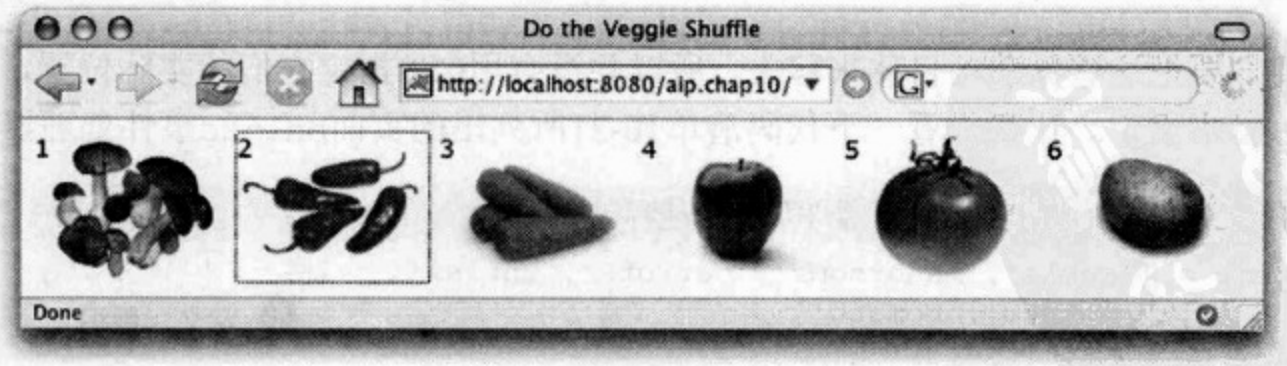


图10-11 什锦果蔬杂拌却有序

那么应该怎么做呢？怎样操作Tab次序的呢？让我们看代码。从果蔬项目的元素本身开始最为简单，所以我们先研究一下页面的<body>元素，如代码清单10-19所示。

代码清单10-19 创建果蔬项目

```
<body>
  <div id="container">
```

```
        <div id="apple"
             style="background-image:url('apple.jpg');"></div>
        <div id="mushrooms"
             style="background-image:url('mushrooms.jpg');"></div>
        <div id="carrots"
             style="background-image:url('carrots.jpg');"></div>
        <div id="chiles"
             style="background-image:url('chiles.jpg');"></div>
        <div id="potato"
             style="background-image:url('potato.jpg');"></div>
        <div id="tomato"
             style="background-image:url('tomato.jpg');"></div>
      </div>
    </body>
```

<body>元素中的外层 容器（主要用于样式）包含了六个可用Tab切换的项目。每一项上都指定了唯一的id，同时也设定了背景图片。注意我们并没有指定Tab次序，也没有设定显示于每个项目左上角的“Tab次序指示器”。因为我们准备基于脚本方式来排列它们，所以会留到代码中去设置。我们也通过CSS给果蔬项设定了样式，对它们采用了绝对定位，如代码清单10-20所示。

代码清单10-20　果蔬的样式

```
<style type="text/css">
  #container {
    position: relative;
  }
  #container div {
    position: absolute;
    width: 110px;
    height: 86px;
  }
</style>
```

因为我们要把这些果蔬项目移来移去，所以并没有用CSS指定它们的实际位置，而是会在页面加载时以脚本设定。让我们看一下代码清单10-21所列出的页面onload事件处理器吧。

代码清单10-21　onload事件处理器和相关内容

```
var items = ['apple','mushrooms','carrots','chiles',
             'potato','tomato'];                          ❶ 定义id数组

window.onload = function() {                              ❷ 对项目进行排列
  arrange();
  items.each(
    function(item,index) {
      $(item).onkeydown = move.bind($(item));             ❸ 设定onkeydown事件处理器
    }
  );
  $(items[0]).focus();
}
```

```
function arrange() {
  items.each(
    function(image,index) {
      $(image).style.left = (116 * index) + 'px';
      $(image).tabIndex = index + 1;
      $(image).innerHTML = $(image).tabIndex;
    }
  );
}
```

❹ 按顺序安排每个项目

我们首先建立一个包括所有果蔬项id的数组❶。注意，这并不是最好的设计决策。我们这个页面是以演示焦点管理为主，所以权且采用它，然而，在数组中重复列出<body>元素中的项目是一个很差的设计，因为它们很容易与另外一方不再同步。更好的方式是从项目列表产生数组，或者从数组产生项目列表。

页面的onload事件处理器会立刻调用名为arrange()的方法来排列项目❷，过一会儿我们会看到它是如何运作的。

接着onload事件处理器遍历所有果蔬项目，给每一项都设定了onkeydown事件处理器❸。这个事件处理器负责检测方向键并交换项目的位置。最后，在onload事件处理器的末尾，我们让第一个果蔬项获得焦点。

对于arrange()函数，我们不仅会在页面加载时调用，也会在每次改变项目次序的时候调用它。它遍历项目id数组❹，对每一项都执行下面三项任务：

- 指定项目的位置。每个项目是110像素宽，那么将位置设为116的倍数，就可以让项目之间留下6像素的空间。
- 按照数组里id的顺序逐一指定Tab次序。由于Tab次序从1开始而数组索引从0开始，所以在计算Tab次序的值时需要给数组索引加1。
- 将项目的tabIndex属性值写入该项目的innerHTML，这样就能在项目的左上角显示出Tab索引值。

在页面加载的时候每个项目上都设定了按键事件处理器，这个处理器函数如代码清单10-22所示。

代码清单10-22 处理按键

```
function move(event) {
  if (!event) event = window.event;
  if (event.keyCode == 37) {
    moveItem(this,-1)
  }
  else if (event.keyCode == 39) {
    moveItem(this,+1)
  }
}
```

当一个项目获得焦点时按下键盘就会调用此事件处理器函数，事件相关的信息会通过参数传递给处理器——至少遵循标准的浏览器是这样的。Internet Explorer 6不遵从标准，对于事件处理有它自己的一套。为此，我们检查是否提供了参数，如果没有，就从window实例中取得事件对

象，因为IE 6坚持要把它放在那里。

然后检查所按按键的键值代码（key code），如果是左方向键（键值代码为37），就调用名为moveItem()的函数，传递给它的参数包括函数上下文（this）和指定项目如何移动的数值——对于左方向键来说，就是回退一格（向左移动）。

如果键值代码对应的是右方向键（39），项目就要向前移动一格（向右移动）。

注意，这里的函数上下文对象就是项目本身。这是因为我们在设定事件处理器的时候，已经预先使用了Prototype库的bind()函数。如果不进行绑定，函数上下文就会是window。

移动项目的工作是在moveItem()方法中执行的，如代码清单10-23所示。

代码清单10-23　移动项目

```
function moveItem(item,by) {
  var oldIndex = item.tabIndex - 1;
  var newIndex = (oldIndex + by + items.length) % items.length;
  items[oldIndex] = items[newIndex];
  items[newIndex] = item;
  arrange();
  item.blur();
  item.focus();
}
```

代码清单10-23中的moveItem()方法并不会在物理上移动传给它的项目，而是改变项目在items数组中的位置。记住，arrange()函数会根据items数组来处理位置和Tab索引，所以这里我们所要做的只是按照我们希望的方式重新排列数组，而由arrange()来帮我们做其他的那些体力活。

这个函数根据参数by的值来交换项目的位置。注意对取模运算的运用，这能让列表两端的项目绕回到另一端。一旦数组调整完毕，我们就调用arrange()，它会在物理上重新设定项目的位置并给它们设定规则的Tab索引。

这个函数最后执行的任务看上去有点奇怪。有人可能认为浏览器应该能自行搞定，但是它们通常并不行。所以我们主要就是通过让项目失去焦点然后又获得焦点来强制唤醒浏览器，找回方向感。

3. 讨论

这些果蔬的图片并不能代替实际的用户界面元素，但是在本方案中，它们可以让我们清楚地看到可获得焦点的项目在页面上重新排列时的情形。

我们看到，重新排列项目时如果不重新设定Tab次序，就会导致应用程序出现让人混乱失措的行为。我们也看到了重新调整元素的Tab次序所必需的脚本代码的典型示例。

尽管这个示例本质上并没有使用Ajax，但是它的经验对于Ajax Web应用来说非常重要，因为在富用户界面设计中经常会用到动态变化的视觉元素。

10.3.2　管理堆叠顺序

我们早晚会有一个Web应用程序需要能将一些元素分层放置于其他元素之上。无论是要进行

拖放操作（drag and drop），编写包含多重虚拟页面（virtual page，指在同一个网页中模拟多个页面效果）的向导，创建浮动的模态对话框，还是用JavaScript编写我们自己的窗口管理器，都可以归结为一件事情：管理堆叠顺序。在HTML页面中，堆叠顺序由z-index的值决定，z-index定义了元素的“地位高低”。z-index的值越大，其层次就越接近“顶端”。因此z-index为100的元素会被绘制于z-index为99的元素之上。当元素的z-index相同时，则依照它们在HTML文档中的声明语句的先后顺序进行堆叠。

正像焦点管理一样，处理堆叠顺序对于创建着富用户界面的Ajax开发者来说，虽然不是Ajax的直接主题，但也是很重要的一课。

1. 问题

我们想学习如何在富Web应用页面中控制元素的堆叠顺序。

2. 解决方案

我们将学习如何通过页面元素的z-index来管理元素的堆叠顺序，并借用蔬菜抽屉里的果蔬当作道具。让我们创建一个和上一个方案很接近的页面，但是这次果蔬不是排成一行，而是垒成一叠。当然我们的关注点也从Tab次序转向了z-index。页面最初如图10-12所示。

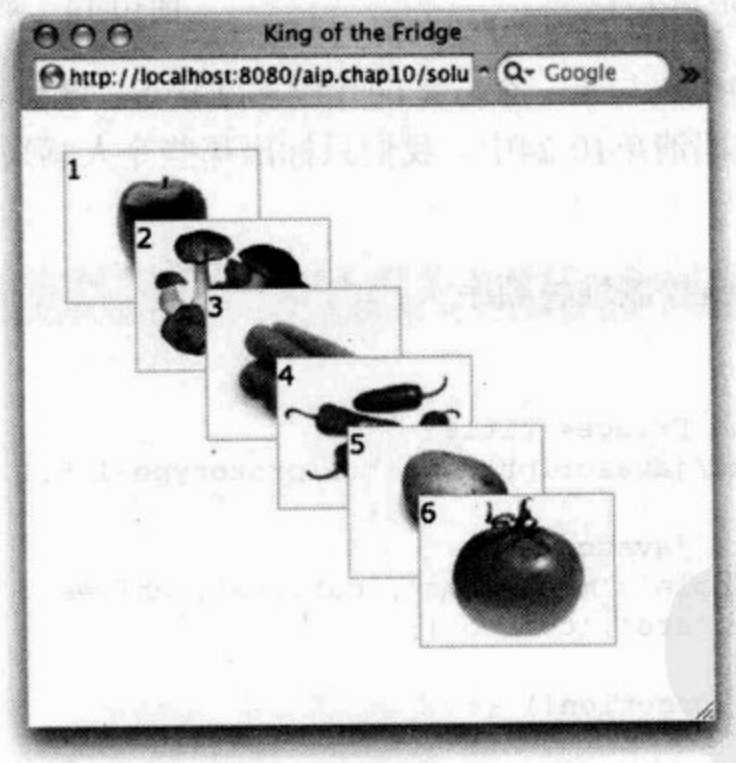


图10-12 番茄是果蔬保鲜盒中的老大

我们不仅把项目改成层叠排列，也给它们增加了边框（可更清晰地显示出它们各自所占区域），而每个项目左上角的数字表示其z-index。如我们所见，z-index越大，就绘制在越“顶端”。

我们也给每个项目加上了onclick事件处理器，这样一旦点击某个项目，就会把它的z-index改为6，使其成为最顶端的元素，同时也重新调整其他元素的z-index。

比如在点击胡萝卜（最初z-index为3）后，页面就变成了图10-13的样子。

如果我们接着点击苹果，那么z-index以及项目的堆叠顺序就会再次重新排列，结果变成了图10-14的样子。

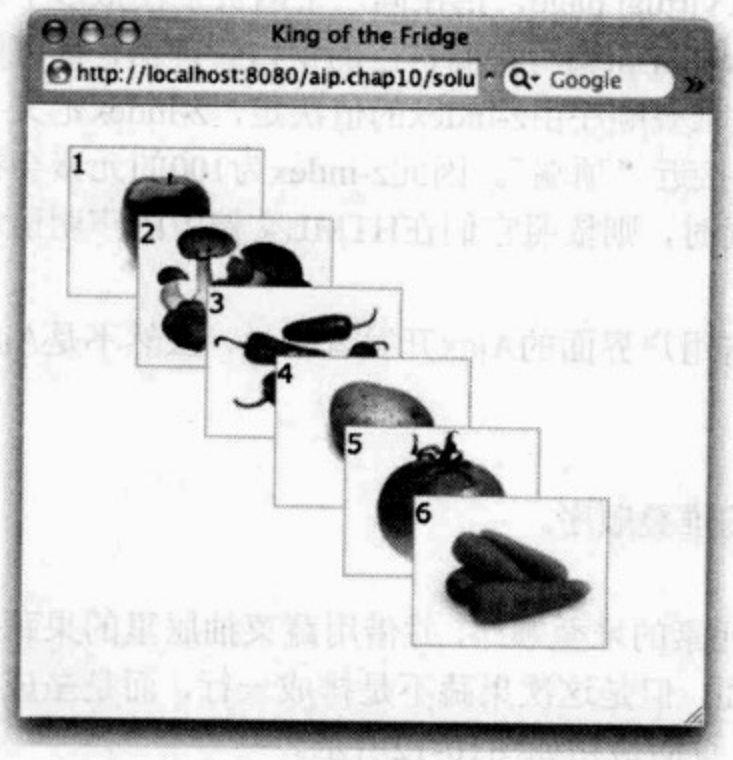


图10-13　胡萝卜成老大了

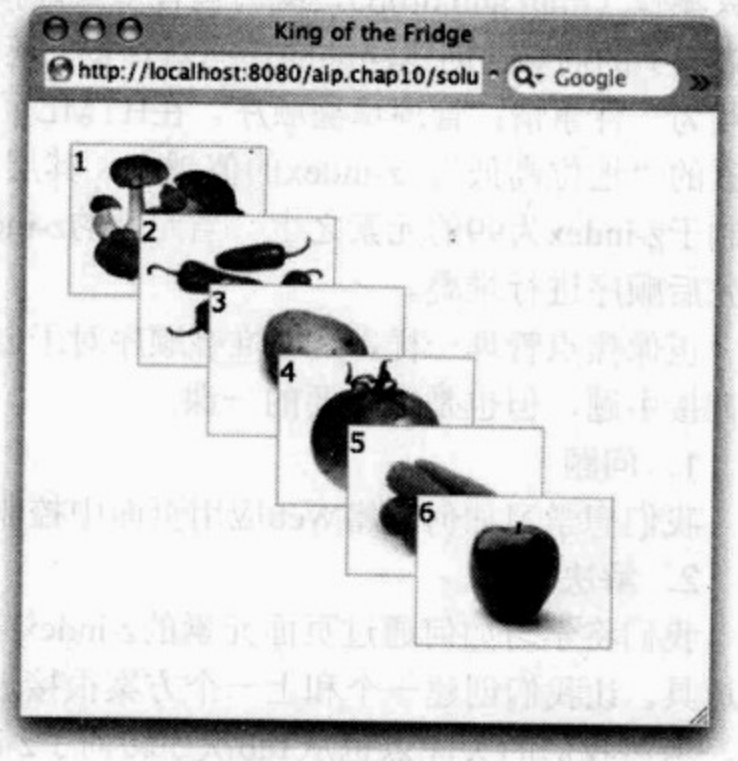


图10-14　苹果宣称它才是老大

完成所有这一切所需的代码与上一个方案的代码非常接近，既然如此，我们就不再逐段分析了。页面的完整代码列于代码清单10-24中，我们只标出那些令人感兴趣的与上一节中的Tab索引例子有所变化的部分。

代码清单10-24　通过z-index控制堆叠顺序

```
<html>
  <head>
    <title>King of the Fridge</title>
    <script type="text/javascript" src="../prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      var items = ['apple','mushrooms','carrots','chiles',
                   'potato','tomato'];

      window.onload = function() {
        arrange();
        items.each(
          function(item,index) {
            $(item).onclick = raiseItem.bind($(item));
          }
        );
        $(items[0]).focus();
      };

      function arrange() {
        items.each(
          function(image,index) {
            $(image).style.left = (16 + (40 * index)) + 'px';
            $(image).style.top = (16 + (40 * index)) + 'px';
```

❶ 鼠标点击时对z-index进行调整

❷ 设定每个项目的位置

```
            $(image).style.zIndex = index + 1;
            $(image).innerHTML = $(image).style.zIndex;
          }
        );
      }

      function raiseItem() {                    ❸ 将项目提升到顶端
        var itemIndex = this.style.zIndex - 1;
        var newItems = [];
        items.each(
          function(item,index) {
            if (index != itemIndex) newItems.push(item);
          }
        );
        newItems.push(this);
        items = newItems;
        arrange();
      }
    </script>
    <link rel="stylesheet" type="text/css" href="../styles.css"/>
    <style type="text/css">
      #container {
        position: relative;
      }
      #container div {
        position: absolute;
        width: 110px;
        height: 86px;                           ❹ 添加边框
        border: 1px silver solid;
      }
    </style>
  </head>

  <body>
    <div id="container">
      <div id="apple"
           style="background-image:url('apple.jpg');"></div>
      <div id="mushrooms"
           style="background-image:url('mushrooms.jpg');"></div>
      <div id="carrots"
           style="background-image:url('carrots.jpg');"></div>
      <div id="chiles"
           style="background-image:url('chiles.jpg');"></div>
      <div id="potato"
           style="background-image:url('potato.jpg');"></div>
      <div id="tomato"
           style="background-image:url('tomato.jpg');"></div>
    </div>
  </body>

</html>
```

本例中的onload事件处理器给每个项目设定了onclick事件处理器(而不再是键盘事件处理

器），这样在点击鼠标的时候就会调整z-index❶。

arrange()函数仍然用于设定项目的位置❷，只不过不是排成一行，而是垒成一叠。它根据项目在items数组中的顺序重新设定其z-index（通过对style.zIndex属性的赋值操作）。注意写入项目内容中的数字也是它的z-index。

函数raiseItem()作为项目上的onclick事件处理器，会对所有项目重新排序。它会创建一个新数组，在加入了所有其他项目之后，把被点击的项目放在数组的末尾❸。这个新数组会取代旧的，然后arrange()函数会被调用，执行那些体力活。最终结果就是所有项目会依照它们新的堆叠顺序重新定位和绘制。

最后一个可注意到的变化就是添加了样式来绘制环绕元素的边框❹。

3. 讨论

本方案大量借鉴了10.3.1节的方案，展示了如何通过管理项目的z-index来改变它们的堆叠顺序。概念再简单不过了——z-index越大，堆叠顺序就越高，至少理论上应该如此。然而IE 6在涉及<select>元素的时候，就显露出其丑陋的一面，破坏了我们的完美设计。

假定我们在页面最后加入一个<select>元素，它也会与其他项目交叠，我们将其z-index设为4。由于它与辣椒具有相同的z-index又是在其后面定义的，所以我们会预期它会绘制于辣椒上方，而处于土豆和番茄下方，因为它们的z-index值大于4。

实际上，在大多数表现良好的浏览器中就是这样的。如图10-15的左半部分所示，页面在Safari中能正确地显示。另一方面，IE 6总是把<select>绘制在最顶端，而不管它们的z-index值是多少，如图的右半部分所示。在IE 6中的<iframe>元素也会表现出同样的情况①。

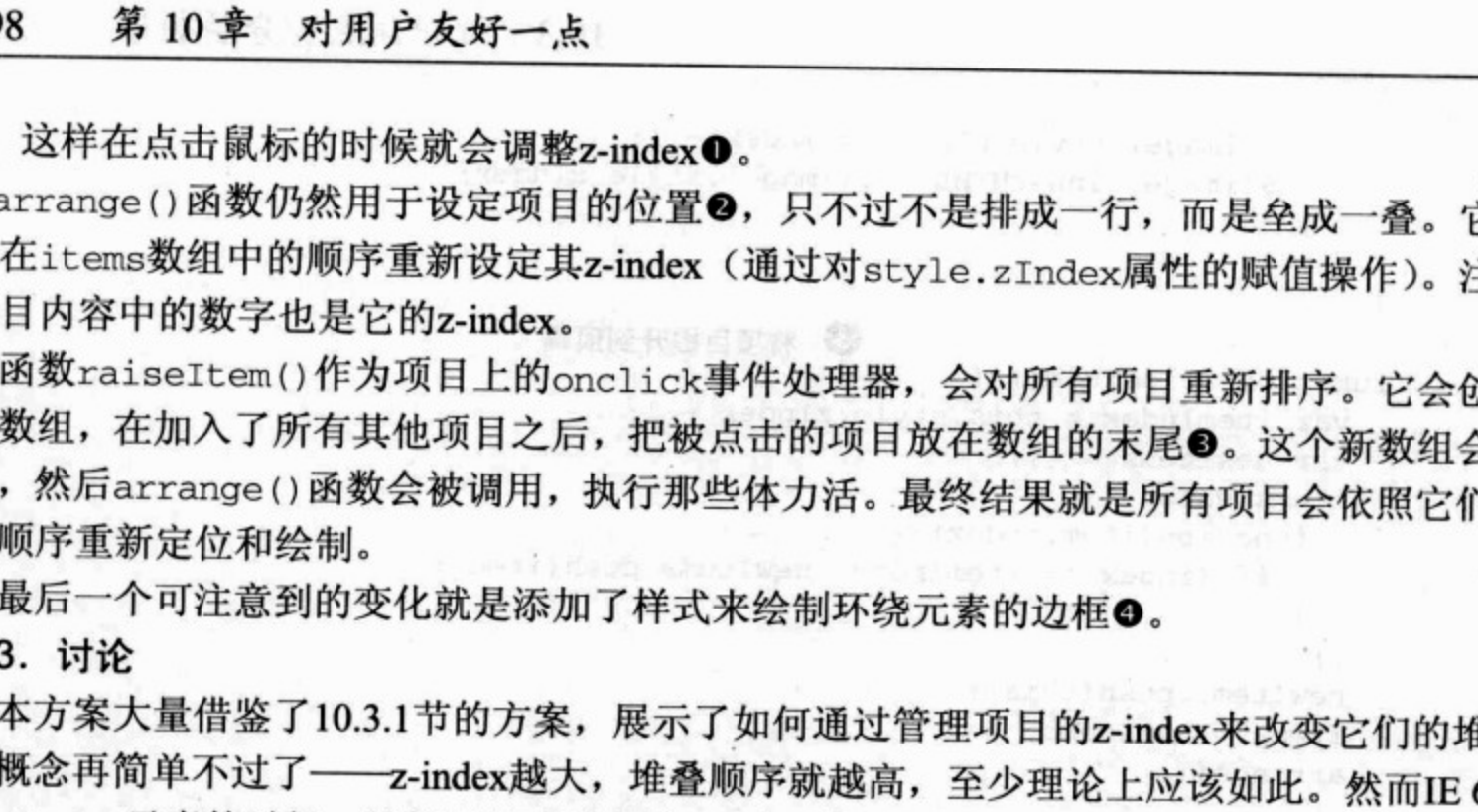


图10-15　唔……这可能是个问题

① 原作者有误，IE 5.5之后的<iframe>元素不再有这种问题。——译者注

除了调整页面设计不让<select>之类的元素与其他元素交叠之外，还有几个对策可以规避IE 6的这一缺陷。一个常见的方法是当要在select元素上方显示其他内容时把select元素隐藏起来（利用CSS的display属性）。另一个对策是把更高层的元素显示于<iframe>元素中[①]。

这些方法都不能很令人满意，特别是后者有严重的侵入性，因为同属一个逻辑页面的一部分需要被强制分离到另一个HTML文档中才能通过<iframe>载入。[②]

幸运的是，Internet Explorer 7不再有这一特殊问题了。

10.4 总结

通过本章的讨论我们认识到，像通常在桌面应用中那样，在Web应用中也应该给予优使性充分关注，特别是随着Ajax进入开发者的工具箱，Web应用也正在变得越来越复杂和丰富。

只要我们有心，从小处着手也可以改善优使性。让用户随时掌握状况，让界面变得符合直觉，不做让用户意想不到的事情，这样就能“赢得战役”。

我们已经看到不少方法，仅仅利用了一些Ajax所支持的简单技巧，就能让我们达成这些目标。由于采用了面向对象的方式来组织这些技巧，所以使用这些技巧所需要引入页面的额外代码也都很清晰，不会是无法维护的一团乱麻。

让我们的用户称心如意而不是灰心丧气，是每一个Web应用开发者的目标。借助Ajax，我们能比以往更接近这一目标。

---

① 原作者理解有误，见下一条注解。——译者注

② 原作者理解有误，通常在IE 5.5以上使用<iframe>用来解决<select>覆盖普通元素的问题，是指在元素层和<select>之间插入一个<iframe>作为遮罩层，该<iframe>与元素层大小一致，而z-index小于元素层但大于<select>。可参考http://dotnetjunkies.com/WebLog/jking/archive/2003/07/21/488.aspx。这一方式之所以能奏效是因为IE5.5以上的<iframe>以一种非常怪异的方式处理zindex，具体请参考http://support.microsoft.com/kb/q177378/。——译者注

第11章 状态管理和缓存

11

本章内容

- 客户端状态的维持
- 服务器数据的预取
- 在客户端本地存储数据

相比上一代Web应用，Ajax应用往往更多地基于客户端。过去，主要由服务器来操控状态和数据——使用基于服务器的会话数据，在服务器端进行数据处理及展示。而现在，随着富客户端的出现，越来越多的数据转由客户端管理。比如说Google Mail，这个应用程序中的页面布局和程序流程完全取决于客户端，服务器端主要充当信息存储库，而由客户端来维持和展现数据。

正是由于现在数据的操控更多涉及客户端，我们就更需要注意从旧的方法（应用程序和数据操作主要在服务器端）到新兴的Ajax方法（应用程序和数据操作主要在客户端）所发生的一些变化，最主要的就是数据载入方式的变化。以前，服务器会从数据库取得所有数据，转换呈现[①]为HTML，然后发送到客户端。现在，各部分数据可以由客户端分开载入，数据的转换呈现也在客户端完成。这导致产生许多个小的请求响应周期，而不是单独一个大的请求-转换呈现-响应的周期。

这里存在一个有点让人担心的问题，那就是从数据库载入数据时所可能出现的延迟。非Ajax应用通常会在服务器端对数据进行缓存，后续的数据请求可以直接以服务器上的缓存数据作为回应。服务器端的数据缓存策略可以做到很智能，因为大部分应用程序在服务器上，服务器了解客户端下一步可以做什么。而现在，Ajax应用程序的服务器组件变得越来越无状态，大部分应用程序驻留于客户端，很难实现智能的服务器端缓存策略，因为我们不知道客户端下一步会干什么。因此，要减少从服务器获取数据所固有的延迟，客户端缓存就变得越来越重要。

如你所料，当处理客户端数据的时候，会出现许多问题，在处理Ajax应用程序时，所有这些都需要重新考虑。

作为一位Web开发者，你要考虑：

- 安全性
- 数据一致性

① 此处render包含转换和呈现双重含义，呈现数据的手段是转换成HTML，而转换的目的则是为了呈现数据。译者水平有限，用单一的中文词汇恐难正确表达，因此勉强将一个英文词转换为两个中文词的复合形式。——译者注

- 性能

性能是使用客户端状态管理的最主要原因，所以我们会着重关注如何优化我们的应用程序以提高速度。

11.1 客户端状态的维持

在研究Web应用程序时，关于在客户端维持用户状态，你可能会碰到两类不同的问题。第一个问题涉及如何在用户使用应用程序的同时维持用户状态，这一问题主要研究应用中如何跨越多个页面保持数据。第二个问题涉及如何跨越多个用户会话维持用户状态。

第一个问题比较容易解决，可以在窗格集合（frameset）中维持一个数据窗格（data frame），如代码清单11-1所示。简单来说，你的应用运行在两个窗格里。其中一个窗格是可见的，占据整个浏览器，用户在这个窗格中与应用程序进行交互。另一个窗格仅用作数据存储库，是不可见的。为什么需要数据窗格？难道不能用JavaScript全局变量来保存数据吗？是的，你不能。因为每当离开当前页面时，你在该页面上声明的变量就消失了。当你返回到该页面，你的所有JavaScript变量再一次空空如也。因此我们要用数据窗格，它能在应用程序的多次页面跳转间保持数据。更精确地说，数据窗格中维持着一些JavaScript全局变量指向我们的数据。

让我们看一个例子，它展示了我们上面所讨论的行为。首先是<frameset>声明（代码清单11-1），接着是窗格的内容（代码清单11-2和代码清单11-3），下面我们会对它们作进一步讨论。

代码清单11-1 窗格集合的声明

```
<html>
<frameset cols="0%, 100%">
  <frame noresize src="data_frame.html" name="data" />
  <frame src="content_frame.html" name="content" />
</frameset>
</html>
```

代码清单11-2 数据窗格

```
<html>
<script type="text/javascript" language="javascript">
 var foo = 'foo';

 function getfoo() {
   foo = foo + '1';
   return foo ;
 }
</script>
</html>
```

代码清单11-3 内容窗格

```
<html>
<body>

<a href="content_frame2.html">foo</a>
```

```
<script type="text/javascript" language="javascript">
  alert(parent.data.getfoo());
</script>
</body>

</html>
```

内容窗格中的链接点（anchor）标签（即<a>标签）指向content_frame2.html，其内容与content_frame.html是一样的，只是它的链接点标签是指回到content_frame.html。现在当我们点击链接，在应用中进行页面跳转（就是简单地在内容窗格1和2之间切换），我们会看到提示框不断出现，并表明“foo”不断被追加上越来越多个“1”。

这显然是个很简单的例子，但由此可以创造出一些有趣的行为。一个马上可以想到的例子便是简单而强大的“向导(wizard)”概念。在一个向导应用中，用户需要通过多个页面来进行某些设置。跟随向导在不同页面跳转的同时，你所设置的值也必须被记录下来。通常的方法是在用户每次页面跳转时都把数据发送到服务器，然后以某种方式记录这些数据（在J2EE世界中，通常会存于用户的会话中）。当用户完成了向导的最后一页后，所有数据必须被合并成某种事务（transaction）。而使用数据窗格法，用户跟随向导的每一步所输入的值都会保存在数据窗格中，只在最后一步才一起发送到服务器。这把维持向导状态的复杂性从服务器端转移到了客户端。

也可以不用数据窗格法，比方说你的应用驻留于单一页面，并通过管理客户端页面布局来操控内容的动态变更，如果是这样，你就可以直接用JavaScript全局变量来保存数据。另外，数据窗格法的一个缺点是：在页面刷新后，数据不可能继续存留于JavaScript变量中。如果你的用户老是喜欢按刷新按钮，那么他们就会老是丢掉状态。你还需要留意安全问题：如果正在使用数据窗格法，你要注意，浏览器已经打开的所有窗口都有机会访问这些窗格中存储的数据。对于存心不良的人来说，他们可以制作出间谍网页来搜罗数据窗格所保存的数据。一旦得逞，这些数据就可以被送回他们指定的服务器。

我们之前提到的第二个问题涉及用户退出应用和关闭浏览器时如何维持用户状态。这可以在服务器端处理，把所有用户状态数据保存在数据库中。这也可以在客户端处理，把数据以某种方式存留于用户的电脑上。运行于客户端的JavaScript应用程序只有一种浏览器内置方法[①]能把数据存留在浏览器端：cookie。当然，我们也可以采用ActiveX或Java applet来处理客户端的数据持久化（这些插件程序可以访问客户端的文件系统），但那要求用户安装插件，违背了轻量级客户端应用程序的观念。我们鼓励读者探索这些方法，并会研究这样一种重量级客户端持久化机制——AMASS（Ajax Massive Storage System，Ajax数据块存储系统）。我们展示AMASS的唯一原因是，它利用了非常普及的Flash插件（根据AMASS作者的说法，95%的机器都具备Flash）来进行存储。在11.3节中，我们会更深入地探索如何用cookie和AMASS来实现状态持久化。

如果允许以非跨浏览器兼容的方法在客户端存储大量数据，那我们可以关注一下Internet Explorer的客户端持久化机制。你可以从这儿找到很多微软的文档：http://msdn.microsoft.com/

① 正在制定中的HTML5规范草案会加入一些新的浏览器端存储机制。——译者注

library/default.asp?url=/workshop/author/persistence/overview.asp。如果你打算使用微软的客户端存储机制，请注意它的容量有限，每个域名下只有大约640KB的空间，而在每个页面里也不能超过64KB。

11.2 服务器数据缓存

迟缓的用户界面保证会让用户恼怒。那么让用户界面如此迟缓的罪魁祸首又是什么呢？那便是向服务器请求数据并从服务器取回数据所导致的延迟。因此，要提供一个爽利的客户端界面，我们必须能很快地获得要显示的数据。在服务器方面有一些方法（诸如调优查询或缓存常用数据），但还是要承受请求-响应的开销。唯一能除去这一开销的方法就是把数据保存在客户端。本节就会对Ajax应用程序的这个方面进行探索——服务器数据缓存。

我们会看一下应该怎样在客户端保存服务器数据。当然，我们仍必须首先获得要保存的数据，因此我们也会研究如何使用服务器数据预取来进一步提速。

你也许想知道如何缓存从服务器收到的数据。这个问题的答案很大程度上取决于你的应用程序的类型。存储数据的一些可能形式如：

- JavaScript对象；
- 多重数组（multiple arrays）；
- XML DOM树，可被插入或移出客户端DOM；
- 在这儿写下你喜欢的其他机制。

如果你的数据是高度静态的，那么一个不错的方法或许是预先将它们转换成客户端的DOM树，然后插入或移除。如果你要对数据做一些修改并要遍历它们，多重数组是合适的方法。而JavaScript对象可以被装配到一个复杂的客户端缓存框架中，框架能截听(intercept)到上述对象上发生的事件并把这些事件分派到一些已注册的监听器(它们可以根据事件的性质来采取适当的动作)。所以你可能已经得出结论，存储的形式取决于你的应用的特性。

在本章中，我们会以JavaScript对象的形式存储数据，并使用面向对象的JavaScript编程方式。面向对象的JavaScript在Ajax世界正变得越来越流行（有了卓越的Prototype库，面向对象的JavaScript也变得很轻松），因此开发者也应该更熟悉它了。

第一个例子会构建一个简单的对象存储机制来反映从数据库查询检索到的记录。然后第二个例子会扩展第一个例子，加上预取机制。为了展示这些功能，我们会开发一个简单的顾客名录的应用，让我们可以分页浏览一个顾客的列表。开始编程吧！

11.2.1 Java类的数据的交换

在这个问题里，我们会研究一个简单的客户端对象存储机制，来存储从数据库检索到的记录。我们会从一个小小的只具备简单功能的网页开始。它能向服务器查询顾客信息列表，然后存储在客户端。它也能给数据分页。它不会预取，也不会向服务器检查数据是否过期，我们稍后再研究那些情况。

1. 问题

你需要把服务器端数据缓存于客户端，以此提高界面的速度。

2. 解决方案

让我们首先为顾客开发一个Java类（见代码清单11-4）。最终结果看上去应该和图11-1差不多。这个类应该具有一些简单的属性，例如姓名和顾客ID号。ID号很简单，就是该顾客在我们用来模拟数据库的顾客数组中的索引值。如果我们使用真实的数据库，它可以是一个UUID（universally unique identifier，通用唯一标识符）或其他标识字段。为了与Java类中的数据进行数据交换，我们会使用JSON库来把数据序列化为JavaScript。

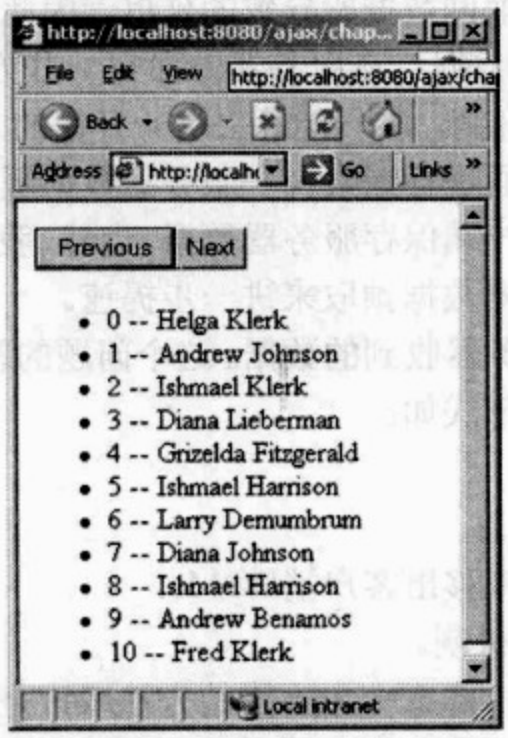


图11-1　在客户端缓存顾客数据

代码清单11-4　Customer类

```
public class Customer
{
    private String firstName;
    private String lastName;
    private int customerID;

    public Customer(String firstName, String lastName, int customerID) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.customerID = customerID;
    }

    public JSONObject toJSON() throws JSONException {          ❶将Java对象转换为JSON
        JSONObject obj = new JSONObject();
        obj.put("firstName", firstName);
        obj.put("lastName", lastName);
        obj.put("customerID", customerID);
        return obj;
    }
}
```

正如你看到的，我们用JSON的Java类库（可在www.json.org获得）来把Java对象转换为对应的JSON表达❶。

有了顾客数据的JSON表达，我们还需要一个类来存储所有顾客并能对他们进行查询（见代码清单11-5）。这个类也要帮我们创建一些所需的模拟数据。

代码清单11-5 CustomerManager类

```
public class CustomerManager
{

  private static CustomerManager instance = new CustomerManager();

  public static CustomerManager getInstance() {
    return instance;
  }

  private CustomerManager() {
    this.customers = new ArrayList<Customer>();
    createCustomers(100);
  }

  private List<Customer> customers;
  private void createCustomers (int total) {
    Random random = new Random();

    String[] firstNames = { "Andrew", "Benjamin", "Chris",
        "Diana", "Elaine", "Fred", "Grizelda", "Helga",
        "Ishmael", "Julia", "Kevin", "Larry", "Mallory" };
    String[] lastNames = { "Andersen", "Benamos", "Costa",
        "Demumbrum", "Evans", "Fitzgerald", "Glen",
        "Harrison", "Ibrahim", "Johnson", "Klerk",
        "Lieberman", "Murakami" };

    for (int id = 0; id < total; id++) {
      String firstName = firstNames[random
          .nextInt(firstNames.length)];
      String lastName = lastNames[random
          .nextInt(lastNames.length)];
      Customer customer = new Customer(firstName, lastName,
          id);
      this.customers.add(id, customer);
    }
  }
  public List<Customer> getCustomers() {
    return this.customers;
  }

}
```

创建顾客数据库

随机生成一些顾客数据填充数据库

存储顾客数据库

查询数据库中的顾客数据

有了顾客的JSON表达和某种顾客数据库，我们还需要一个方法来把数据库中所存储的顾客返回到客户端所运行的JavaScript中。代码清单11-6的CustomerServlet就是干这个的。

代码清单11-6　CustomerServlet类

```
public class CustomerServlet extends HttpServlet
{

  CustomerManager cm = CustomerManager.getInstance();

  @Override
  protected void doGet(HttpServletRequest req,
      HttpServletResponse res) throws ServletException,
      IOException {
    int start = Integer.parseInt(req.getParameter("start"));
    int pageSize = Integer.parseInt(req
        .getParameter("pageSize"));

    List<Customer> customers = cm.getCustomers();

    JSONArray ja = new JSONArray();
    for (int current = start; current < start + pageSize
        && !(current >= customers.size()); current++) {
      try {
        ja.put(customers.get(current).toJSON());
      } catch (JSONException e) {
        e.printStackTrace();
        throw new ServletException(e);
      }

      res.getWriter ().write(ja.toString());
    }
}
```

获取customer manager

指定顾客的起始索引

指定返回顾客的数量

❶ 创建新的JSON数组

❷ 存储顾客

将JSON表达输出到客户端

注意，我们使用JSONArray❶来保存一个JavaScript表达，它表示我们要返回的所有顾客。然后我们把每位顾客的JSON表达放入数组❷，之后把数组返回给客户端。

所有的服务器端组件都就绪了，我们的应用也差不多成形了。剩下的就是写一些JavaScript代码来向服务器查询顾客并将结果显示在页面上。我们会使用Prototype库并以面向对象的JavaScript方式写一个CustomerManager类（代码清单11-7），处理顾客的读取、缓存和显示。

代码清单11-7　客户端的CustomerManager

```
var CustomerManager = Class.create();

CustomerManager.prototype =
{
  customerData : new Array(),

  drawCustomerDIV : function(start, pageSize, div, cached) {
    displayString = '<ul>';

    for (i = start;
        i < pageSize + start && i < this.customerData.length;
        i++) {
```

❶ 保存顾客数据

```
    customer = this.customerData[i];
    displayString += '<li>';
    displayString += customer.customerID;

    if (cached) {
      displayString += ' (cached) ';
    }
    displayString += ' -- ';
    displayString += customer.firstName;
    displayString += ' ';
    displayString += customer.lastName;

    displayString += '</li>';
  }

  displayString += '</ul>';

  div.innerHTML = displayString;
},

cacheCustomerData : function (response) {
  responseArray = response.responseText;
  currentCustomerData =
    eval('(' + responseArray + ')');

  for (i = 0; i < currentCustomerData.length; i++) {
    customerID =
      currentCustomerData[i].customerID;
    this.customerData[customerID] =
      currentCustomerData[i];
  }
},

getCustomerData : function (start, pageSize, div) {
  if (this.customerData.length > start) {
    this.drawCustomerDIV(start,
      ➥ pageSize, div, true);
  } else {
    manager = this;
    options = {
      method: 'get',
      parameters: 'start=' + start +
        '&pageSize=' + pageSize,
      onSuccess: function(response) {
        manager.cacheCustomerData(response);
        manager.drawCustomerDIV(start, pageSize, div, false);
        },
      onFailure: function(r) {
        alert('Server Status: ' + response.status + ' - ' +
        response.statusText);
      }
    };
```

❷ 显示缓存标志

❸ 显示顾客数据

❹ 反序列化读取到的顾客数据

❺ 获得顾客的ID

❻ 缓存顾客数据

❼ 显示顾客数据

❽ 设置自身引用以便回调

❾ 向服务器传递正确的参数

```
        new Ajax.Request('/ajax/servlet/Customers', options);

      }
    },

    initialize : function() {}
  }
```

让我们仔细观察CustomerManager类。首先我们创建了一个数组用来缓存顾客数据❶。这个数组在后续代码中会频繁用到。

其次，我们有drawCustomerDIV()方法，它取出顾客缓存数据并在我们指定的 中创建一个列表。变量displayString会保存我们拼装出的顾客列表的HTML代码。当拼装完成，我们就可以把 的内容设为displayString变量所包含的HTML代码。参数cached的值若为true，意味着我们不是从服务器取得要显示的顾客，而是完全依靠customerData数组里的缓存数据。如果是这样，我们会加上标注，表示这些顾客是取自缓存❷。方法的最后，我们将用于显示顾客的 的内容❸设为我们通过循环拼装出来的列表。

然后，我们有cacheCustomerData()方法，只有在从服务器读取到顾客之后，它才会被调用。它会根据从服务器端接收到的JSON数组来创建JavaScript对象，并适当地装填到customerData数组中。具体来说，它获得我们从服务器接收到的JSON文本，并据此创建一个JavaScript数组❹。然后，对数组的每个顾客对象读取顾客ID❺并存储到以ID为索引的缓存中❻。

最后，我们创建了getCustomerData()函数，它会决定所请求的顾客数据的范围是否在缓存中。如果在缓存中❼，函数会简单地输出顾客数据。如果不在缓存中，函数会向服务器请求顾客对象，进行缓存，然后呈现。我们也用manager保存当前对象的引用❽，因为后面的onSuccess事件处理函数中需要引用manager变量而不是this变量。因为当这个回调函数被调用时，this变量并不会指向CustomerManager对象而是指向一个Ajax.Request对象。因此，我们需要以这种方式来调用manager。为了使用Prototype的Ajax.Request对象来创建一个异步的XHR对象，我们装填了一个选项对象来给Ajax.Request传递一些选项。在选项中，start参数❾指示所请求的数据范围在数据库中的起始索引，pageSize参数指示返回顾客的数量。如果请求成功，我们会读取响应，对返回的顾客进行缓存并呈现他们。我们所执行的最后一个动作是对数据Servlet发出请求，路径/ajax/servlet/Customers会定位到CustomerServlet。

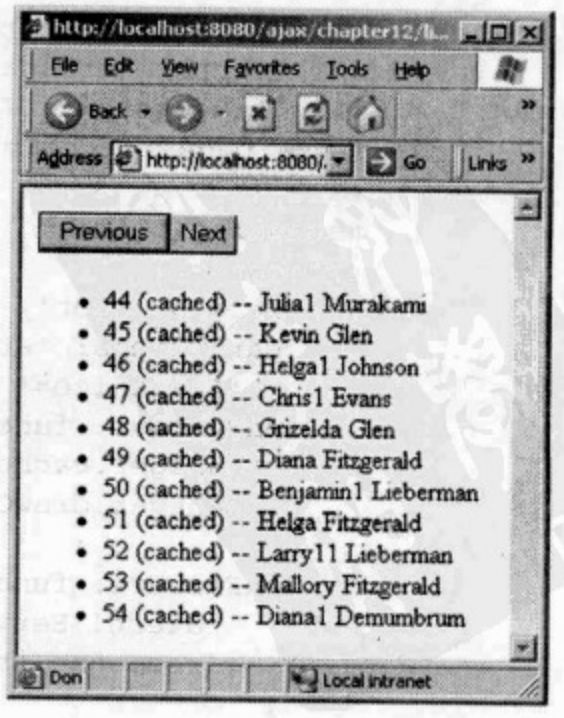


图11-2　显示已缓存的顾客

● 显示顾客数据

有了服务器端代码和客户端代码，我们差不多大功告成了。如图11-2所示，cached标注说明当前所显示的顾客是取自内部顾客缓存而不是服务器。

下面是一段HTML（代码清单11-8），除了显示顾客数据外，还带有两个JavaScript函数可让我们前后翻页。

代码清单11-8 分页显示顾客的HTML

```
<html>
<head>
<script type="text/javascript" src="../json.js"></script>
  <script type="text/javascript" src="../prototype-1.4.0.js"></script>
  <script type="text/javascript" src="listings.js"></script>
  </head>
<body>

  <button onclick="previous();">Previous</button>
  <button onclick="next();">Next</button>

  <div id="customers"></div>                    <-- <div>用于显示顾客数据

  <script type="text/javascript">
  var totalCustomers = 100;
  var currentCustomerIndex = 0;
  var pageSize = 11;

  var manager = new CustomerManager();          <-- 移至上一页顾客列表
  function previous() {
    currentCustomerIndex -= pageSize ;
    if (currentCustomerIndex < 0) {
      currentCustomerIndex = 0;
    }
    manager.getCustomerData(currentCustomerIndex, pageSize,   <-- 在<div>中显示顾客数据
    $('customers'));
  }

  function next() {                             <-- 移至下一页客户列表
    currentCustomerIndex += pageSize;
    if(currentCustomerIndex >= totalCustomers) {
      currentCustomerIndex = totalCustomers - 1;
    }
   manager.getCustomerData(currentCustomerIndex, pageSize, $('customers'));
  }

  manager.getCustomerData(currentCustomerIndex, pageSize, $('customers'));
  </script>
</body>
</html>
```

代码的快速注解：我们需要了解查询一共返回了多少顾客——在本例中是所有的顾客。我们在代码中硬性设定了totalCustomers变量的值，从而去掉了读取顾客总数的代码。如果是在现实世界里，需要编程取得可供显示的顾客总数并赋值给totalCustomers变量。

3. 讨论

我们现在已经完成所有需要的组件。让我们通过Next按钮向后跳过几页，然后按Previous按钮回到之前的列表。因为我们已经缓存了那些顾客，页面应会告诉我们所列出的顾客是取自于缓存。

本节用一个简单的例子展示了复杂的客户端行为。在现实世界中，就不那么简单了。像我们前面提到的，我们把可供显示的顾客总数写死在代码中，从而省略了许多客户端和服务器端代码。

在现实世界中，为了在执行查询时取得顾客的数量，我们需要写上不少代码。

把之前请求过的结果缓存起来，无论对于用户还是应用提供者来说都可获得很大的性能提升。用户享受到了显著的性能提升，因为不必再为已经装载过的数据往返访问服务器。应用提供者也可以节省下服务器周期和带宽，因为他们不必承受那些已经给客户端发送过的数据的额外请求。

我们还能进一步优化用户体验。在下面的示例中，我们会扩展缓存行为，把用户可以跳转到的下一个页面也载入进来。按这一方法，缓存会马上预热，翻页根本不花时间。

11.2.2　预取

为了提供响应更为迅速的用户界面，我们需要探讨一下预取这个主题。基本的想法是，我们知道用户下一步要看什么数据——在一开始他们只有一个选择就是转到下一页。所以我们不是等待用户点击Next按钮后再载入数据，而是用户还没有动作之前我们就装载它、缓存它。由于行为是异步的，预取可在幕后进行，客户端不会出现浏览器抓取未读数据时所出现的迟滞。

1. 问题

你需要从服务器预取数据以提高客户端的速度。

2. 解决方案

我们会重用上一个例子的大多数代码，并在少数地方做一些改动。服务器端代码一点儿也不用改。它只需供给我们顾客数据，相比之前也没有发生任何变化。我们需要改动的地方是客户端的CustomerManager对象。现在不再是直接发送一个请求到服务器，可能发送两个（当我们第一次装载页面时需要读取当前页面和下一页面），也可能是一个（当点击Next时，需要读取再下一个页面），结果应该类似图11-3。让我们看一下修改过的CustomerManager的JavaScript代码（代码清单11-9）。

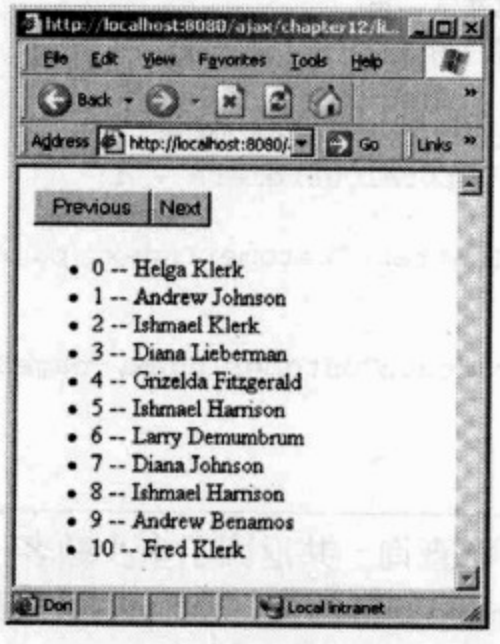


图11-3　最初页面上未经预取的顾客

代码清单11-9　客户端的JavaScript

```
drawCustomerDIV : function(start, pageSize, div, cached) {
    if (div == null) {
      return;
    }
```

❶ 防止无穷递归

```
    displayString = '<ul>';
    for (i = start;
         i < pageSize + start && i < this.customerData.length;
         i++) {
      customer = this.customerData[i];
      displayString += '<li>';

      displayString += customer.customerID;
      if (cached) {
        displayString += ' (cached) ';
      }
      displayString += ' -- ';
      displayString += customer.firstName;
      displayString += ' ';
      displayString += customer.lastName;

      displayString += '</li>';
    }

    displayString += '</ul>';
    div.innerHTML = displayString;
  }

getCustomerData : function (start, pageSize, div) {
  if (this.customerData.length > start) {
    this.drawCustomerDIV(start, pageSize, div, true);
    if(div != null) {
      this.getCustomerData(start + pageSize, pageSize, null);
    }
  } else {
    manager = this;
    options = {
        method: 'get',
        parameters: 'start=' + start + '&pageSize=' + pageSize,
        onSuccess: function(response) {
          manager.cacheCustomerData(response);
          manager.drawCustomerDIV(start, pageSize, div, false);
          if (div != null) {
            manager.getCustomerData(
              ➥ start + pageSize, pageSize, null);
          }
        },
        onFailure: function(r) {
            alert('Server Status: ' + response.status + ' - ' +
response.statusText);
        }
    };

    new Ajax.Request('/ajax/servlet/Customers', options);

  }
}
```

❷ 测试是否是UI调用的

❸ 测试是否是UI调用的

❹ 获得下一页数据

3. 讨论

对于drawCustomerDIV()函数❶，我们只做了一处修改：如果 为null，什么都不做直接退出函数。在后面的函数里你会看到为什么要做这样的修改。

对于getCustomerData()函数，我们做了不少修改。如果 不为null❷（意味着是从UI直接调用），我们就获取下一页顾客数据，并把 设为null。这是因为我们不希望显示下一页，并且我们要避免产生死循环。同样地，如果 不为null❸，我们就读取下一页数据❹。我们传入null作为 的值，因为要避免递归调用，并且不需要呈现预取的页面。

有了这些小改动，我们现在就可以对数据进行预取了。我们期待第一页不会显示cached标注（图11-3），而后续的页面则应该会显示出cached标注。

现在让我们翻到下一页（图11-4），它会告诉我们使用了顾客数据缓存。实际上，我们确实知道这些顾客是从缓存而不是从服务器读取的。

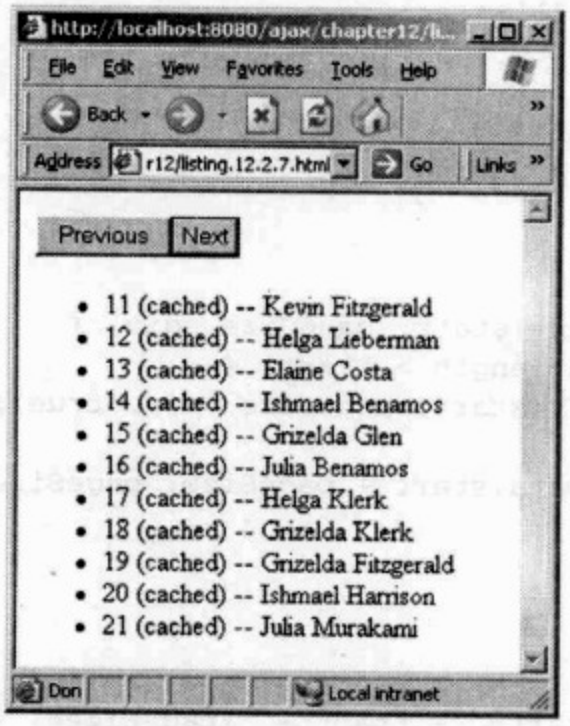


图11-4　浏览器显示出经过预取的顾客及cached标注

为预取数据对CustomerManager所做的修改非常简单：我们只做了三处小修改，就可以在用户查阅当前数据的同时，暗中预取下一页的顾客数据。

这里浮现出一个有趣的问题：如果用户翻页速率比我们预取的速率更快会发生什么呢？在上面这个解决方案中，仍能显示出正确的数据：当我们跳转到下一页时，会发现在缓存中没有所需的顾客数据（因为它们还在读取中），于是我们会发出另一个请求从服务器获取那些顾客的数据。这会导致我们为相同数据发出多份请求——这一情况当然不能令人满意。如果我们使用Java（或任何其他具有互斥锁[mutex]的编程语言），我们只需要实现一对带锁的reader-writer，其中reader在数据不可用时会等待互斥锁，直到writer把数据准备好后唤醒reader。不幸的是，JavaScript并没有这样的同步机制。当然有一些迂回的解决方法，其中一种可行的方案就是实现一种基于定时器和队列排程的模拟线程机制：

- 把向前翻页和向后翻页的功能实现为两个命令对象，并置于命令队列中。命令可以检查

所需数据是否准备好了，如果准备好了，我们就刷新UI，否则就失败。

- 我们使用setInterval()或setTimeout()方法来定时调用一个命令对象执行器。它会查看命令对象队列并执行队列最顶端的命令对象。如果执行成功，我们就把该对象从执行队列中删除。如果失败，我们就不管它，直到再次进入执行环路。

也可以在服务器端解决这个问题。服务器可以检测出相同的查询，并把任何重复的查询放入等待队列中。当最初的查询返回结果时，可以把同一个结果传给所有相同的查询。然而，这个方法仍然会在服务器端和客户端之间交换重复的数据。如你所见，这不是一个很容易解决的问题。是否值得引入额外的复杂性？这取决于应用的具体情况。

11.3 客户端状态的持久化

上一节研究了暂存数据，所谓暂存（transient）数据，就是在浏览器重启和页面刷新后无法在客户端保存下来的数据。本节我们会探索如何能把数据长久地存储在客户端机器上。

我们之前提到过，有两种方法可在客户端进行本地数据存储：通过cookie来保存数据，或通过可读写本地文件系统的浏览器插件。下面我们会通过示例来研究这两种机制。

11.3.1 以 JSON 形式存储和取回用户状态

这个例子会向你展示如何把用户状态所对应的JSON表达保存到cookie中以及如何从cookie中读取出来。我们把用户状态存储在一个JavaScript对象树中，并用JSON库来对数据进行序列化和反序列化。你可以在www.json.org了解到更多关于JSON的信息。在www.json.org/js.html可以获得适用于JavaScript语言的JSON库。为了存储和读取cookie，我们使用Webmonkey的cookie函数库，可从http://www.webmonkey.com/webmonkey/reference/javascript_code_library/wm_ckie_lib/获得。你知道，cookie函数库多如满天繁星[①]，所以不用把它当成唯一选择。

1. 问题

你需要通过跨越多个浏览器会话的cookie存储和取回数据。

2. 解决方案

让我们从一段JavaScript和HTML（代码清单11-10）开始讨论我们的解决方案。它展示了如何结合JSON数据和cookie。这段代码没有什么复杂之处，只是一个简单的脚本，它从一个小巧的IP地址向导获得一些信息，存储到浏览器cookie中。

代码清单11-10 使用JSON和cookies对状态对象进行序列化和存储

```
<html>
<head>
<script type="text/javascript" src="../json.js"></script>
<script type="text/javascript" src="../cookies.js"></script>
</head>

<script type="text/javascript">
function getState() {
```

获得用户状态的表达

① 原作者给出的数值是：大约12.5个billion，也就是125亿。——译者注

```
    mainState = new Object();
    var wizardState = new Object();

    mainState.wizard = wizardState;

    wizardState.ip = '192.168.1.6';
    wizardState.nm = '255.255.255.0';
    wizardState.gw = '192.168.1.1';

    mainState.userName = 'json';
    mainState.password = 'nosj15';
    mainState.style = 'bluesteel';

    return mainState;
}

function storeState(state) {                    <-- 存储状态
    serialized = state.toJSONString();          <-- 转换为JSON字符串
    WM_setCookie('state', serialized, 24*100);  <-- 设为100小时后过期
}


function testStateMechanism()
{
    state = getState();                         <-- 获得当前状态
    storeState(state);                          <-- 将状态存入cookie
}

testStateMechanism();                           <-- 进行测试
</script>
</html>
```

运行代码清单11-10中的代码时，我们就把当前状态存储到了cookie中。然后关闭浏览器，重新启动，运行代码清单11-11，我们就得到图11-5所展现的结果。

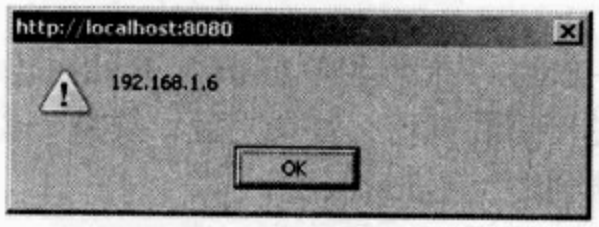


图11-5　alert框显示了我们取回的状态

代码清单11-11　使用JSON和cookie对状态对象进行反序列化

```
<html>
<head>
<script type="text/javascript" src="../json.js"></script>
<script type="text/javascript" src="../cookies.js"></script>
</head>

<script type="text/javascript">
function retrieveState() {                      <-- 从cookie读取序列化数据
    serialized = WM_readCookie('state');
```

```
  return serialized.parseJSON();  <-- 返回反序列化后的数据
}

function testStateMechanism()
{
  state2 = retrieveState();
  alert(state2.wizard.ip);
}

testStateMechanism();
</script>
</html>
```

3. 讨论

使用cookie来跨浏览器会话存储数据也存在一些限制(在RFC 2109中规定)。首先，每个cookie只能存4KB数据。如果你想保存像客户关系管理数据库那样的内容，那就实在不够用。其次，每个域名下只能有20个cookie，所以数据总量被限制在80KB内。

同时要谨记，一个域名下的所有cookie中的所有数据都会被发送到服务器。所以你必须做到：一旦从cookie中读取了数据，就在客户端删除这个cookie；否则，每个客户端请求都会来回发送这些用于状态管理的cookie中的内容。这肯定会拖慢你的应用。

最后，一个需要注意的安全问题是：用户可以很容易地修改cookie的内容。所以你必须小心不要在cookie里存储敏感数据，例如用户名、密码、登录状态、角色、组，等等。有两方面的安全考量：

- 同一台机器上的用户可以获取这些数据，窃取密码和其他敏感信息，也可以简单地复制一下cookie来假扮合法用户。
- 如果像角色或组这样的信息保存在cookie里，用户就可以通过篡改这些数据，让自己获得不应有的权限。

因此，所有与登录和安全相关的内容决不应保存在客户端[①]。

11.3.2 通过 AMASS 保存 JSON 字符串

当你希望在客户端保存超过80KB的信息时，cookie就完全不适用了。那么Web开发者怎么办？AMASS（http://codinginparadise.org/projects/storage/）的出现[②]拯救了我们。它利用Flash插件在客户端存储大量的数据，这些数据会保存在本地文件中。在不需要用户许可的情况下，你可以

① 作者这里稍嫌绝对化，我们常会在安全性和便利性之间进行权衡。实际上，现在大量Web应用都使用cookie记录登录状态，但是会提示用户不应在公共机器上开启这一特性，并且在执行敏感操作的时候（例如修改密码）可能会要求再次验证身份。——译者注

② 目前AMASS已经不再继续后续开发，AMASS的作者Brad Neuberg为Dojo开发了存储模块（dojo.storage或dojox.storage），以取代AMASS。Dojo的存储体系基于灵活的storage provider，通过统一的API对底层存储机制做了抽象。Dojo 0.4.3所带的provider，分别基于Flash(即与AMASS相同)、WHAT WG Storage(即正在制定中的HTML5草案所新增的浏览器端存储机制，Firefox 2已支持该特性）和本地文件（如果Web应用是从本地加载，会利用ActiveX、 XPCOM或Java Applet)。Dojo 1.0.x的存储模块正在重构中，目标将进一步加入基于 AIR的若干种存储机制以及Google Gears的存储机制。——译者注

存储总共100KB数据。要保存更多数据，用户就需要授权Flash允许其存储这样大量的数据。AMASS有点像哈希表——数据以特定的键值存储。你可以保存简单的文本字符串，也可以保存JavaScript对象，只要它们可以转换成字符串表示。

1. 问题

你需要在客户端存储和取回大量的数据。

2. 解决方案

让我们用AMASS来重新实现前面的例子（代码清单11-12）。

代码清单11-12　运用AMASS

```
<html>
<head>
<script type="text/javascript" src="../json.js"></script>
<script type="text/javascript" src="x_core.js"></script>
<script type="text/javascript" src="x_dom.js"></script>
<script type="text/javascript" src="x_event.js"></script>
<script type="text/javascript" src="storage.js"></script>
<script type="text/javascript" language="javascript">
storage.onload(initialize);

function initialize(){
  alert('store is initialized');
}

function storeState(state) {
  storage.putString('state', state.toJSONString(), statusHandler);
}

function testStateMechanism() {
  state2 = retrieveState();
  alert(state2.wizard.gw);
}

function retrieveState() {
  return storage.getString('state').parseJSON();
}

function persistState() {
  state = getState();
  storeState(state);
}

function getState() {
  mainState = new Object();
  wizardState = new Object();

  mainState.wizard = wizardState;

  wizardState.ip = '192.168.1.6';
  wizardState.nm = '255.255.255.0';
```

注册事件处理器

将对象编码为JSON并进行存储

取回状态并打印所包含的信息

取回存储的状态转换为对象

读取应用状态并进行持久化

以嵌套对象来创建模拟状态

```
  wizardState.gw = '192.168.1.1';

  mainState.userName = 'json';
  mainState.password = 'nosj15';
  mainState.style = 'bluesteel';

  return mainState;
}

function statusHandler(status) {
  if (status != Storage.SUCCESS) {
    alert(status);
  }
}
   </script>
   </head>
   <body>
    <button onclick="persistState();">
       Store State</button>
    <button onclick="testStateMechanism();">
       Test Stored State</button>
   </body>
</html>
```

如果不成功就发出提示

3. 讨论

我们前面说过，当我们在页面中跳转或刷新浏览器的时候，会丧失所有内存中的状态。为了测试AMASS，展示其功能，我们会载入示例页面，存储状态，刷新页面，然后检查状态。让我们试试看吧。

首先映入眼帘的是一个提示框（图11-6），告诉我们存储系统已经完成初始化。这是由我们传给storage.onLoad()的回调函数产生的。接着我们会看到应用程序的主页面，其上有我们用于存储和取回状态的按钮（图11-7）。

让我们点击Store State按钮，然后刷新页面。这会产生另外一个提示框，通知我们存储系统准备就绪。关闭提示框之后，我们点击Test Stored State按钮。这应该会弹出另一个提示框（图11-8），告诉我们持久化状态的信息。

图11-6 AMASS初始化消息

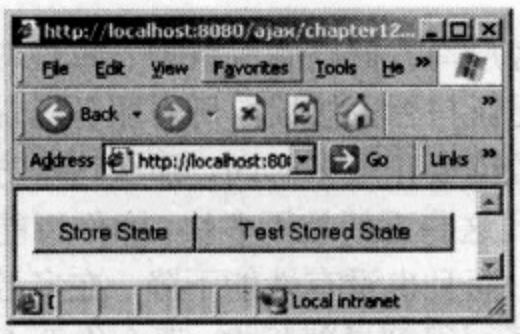


图11-7 存储测试

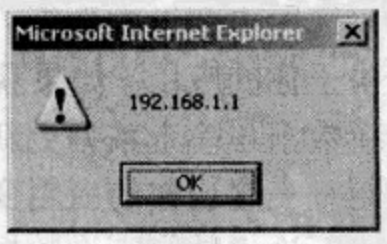


图11-8 存储测试成功

实际情形也确实如此。提示框所显示的值（192.168.1.1）就是我们在向导状态（wizardState）上设置的变量gw的值。在我们的测试函数中，我们取回状态（通过retrieveState()），得到向导对象（wizard），接着得到向导对象上的gw的值，并在提示框中显示出来。如你所见，通过结

合AMASS与JSON，我们可以把整个对象树存起来，并在需要的时候重新载入。

你可能对我们在之前的JavaScript代码中使用的storage变量是从哪里来的感到疑惑。实际上它是由storage.js这个JavaScript库声明的。当我们的页面导入了AMASS JavaScript库，你可以在你的JavaScript函数中的任何地方引用它。AMASS也有另一些函数可能对你有用。我们使用了putString()方法来存储对象，但AMASS也有一个put()方法可以接受任何对象而不限于字符串。不过在测试中我们发现，手动进行JSON编码然后用putString()，比直接把未编码的JavaScript对象传给put()方法要更快。同样地，getString()也有一个对应的get()，可将以put()方法存储的对象反序列化。但正如JSON + putString()快于常规的put()，JSON + getString()也快于常规的get()。AMASS也有一个叫做hasKey()的函数，可以返回true或false表示所指定的键是否有对应关联的值。不过显然还缺一个delete()函数！当我们需要移除一个键，就以null为参数调用put()，好在这样也能奏效。

像cookie一样，AMASS把数据存储在文件里。也像cookie一样，用一个文本编辑器或十六进制编辑器，花上点工夫，就能轻易篡改这些文件。因此，我们前面提到的使用cookie的安全考量同样适用于使用AMASS进行持久化存储的情况——不要用AMASS存储敏感信息。

当你需要在客户端持久存储大量信息时，AMASS是极有用的工具。例如，Web邮件应用可以在客户端存储读过的消息，避免重新读取已经下载过的消息。使用客户端持久化的一个缺点是，软件升级不像之前那样简单了。如果应用程序完全在服务器端，你可以选用任何方式来操作数据。现在有大量的数据存在客户端上，你就失去了这一优势，而是面临一个艰巨任务——写一个客户端程序来升级存储在客户端的数据。程序员请记住这一告诫！

● *对缓存数据的脏数据检查*

在之前的例子中，我们未作讨论的一个内容是对比来自服务器的数据，对客户端持有的数据进行脏数据检查（dirty-checking）。你会遇到这样的情形，服务器数据已经更新，而客户端还缓存着老数据的副本。怎么办？你可以实现复杂的缓存算法来询问服务器是否有数据已经过时并通知服务器把数据更新发送给客户端。要提供这样的功能，服务器端和客户端需要为所有可在客户端缓存的数据保持某种时间戳（timestamp）或版本号。这样我们可以将时间戳或版本号信息与客户端持有的同一信息进行比对。客户端必须向服务器端传送它所持有的时间戳信息，并接收改动过的数据以便更新。

然而，当你对比服务器来检查客户端数据以确保数据一致性时，是基于这样一个假设——检查并只读取更新过的数据，比不管更新与否直接读取所有服务器端数据要更快。这一假设是否有效取决于你的应用。如果获得最新数据要花很多时间，那么这个方法对你是奏效的。如果直接访问最新数据与检查数据更新并只取回这些更新差不多快，那你可能最好不要去实现复杂的版本检查机制——这让你的应用变得简单，而且也没有性能下降。在多数情况下，直接读取新数据会更快——无论如何你查询数据总是为了要得到最新版本，那么你倒不如避免编写复杂的客户端和服务器端代码的麻烦，因为你很可能不会获得任何性能提升。如果你的应用必须总是显示最新最好的数据，那你可能最好不要在客户端做任何缓存。另一方面，如果读取数据要花很长时间并且及时性并不是很重要，那最好在浏览器端做缓存并可以定时刷新缓存。通过XHR，刷新可以在后台发生而无需用户知晓。

11.4 总结

通过本章，你学习了如何存储暂存数据（通常是更新很频繁的数据库数据）和持久数据（例如，用户的设置和应用的状态）。对每种数据我们都展示了两种方法：对于暂存数据来说，你可以对服务器数据进行缓存，或者使用一个包含预取的类似方法；对于持久数据来说，你可以在cookie中保存数据（少量数据）或者使用AMASS库（大量数据）。

你可以也应该在自己的应用程序中使用数据缓存。通过减少网络往返和服务器存取，它能使应用变得更快。使用数据缓存也大大减少了服务器的负载（无论是应用服务器还是数据库服务器）。

在使用客户端缓存时也要注意安全问题。如果用户要使用一些敏感数据，如信用卡号和个人社会安全编号（social security numbers）[①]，你可能需要仔细检查一下所使用的缓存技术。如果你的用户的电脑失窃，数据就可能泄露出去——留下你来解释这一切[②]。开放Web应用安全项目（www.owasp.org/index.php/Guide_Table_of_Contents）就Web应用安全这一主题讨论了相当多的内容，你会发现那里的信息非常有用。

① 美国的个人社会安全编号（SSN）相当于中国的身份证。——译者注

② 美国有较为完善的保护个人隐私的法律和法案，诸如SSN等个人信息的泄漏有可能遭到起诉。——译者注

第12章 开放式Web API和Ajax

12

本章内容

- 使用开放式Web API
- 创建跨服务器代理
- 使用Yahoo!地图、Geocoding及Traffic API
- 使用Google搜索API
- 使用Flickr照片API

有些时候，你其实不必去做所有的事情。

作为Web开发人员，为了交付我们承担的某些具体应用和项目，我们已经习惯于编写大部分代码。但是，如果停下来思考一下，你就会发现，除了自己编写代码，我们还依赖不计其数的辅助软件（support software），这些软件为我们完成了大量功能。从发布网页的Web应用服务器到各种浏览器及其辅助软件、还有运行着服务器和客户端浏览器程序的操作系统。我们在实现自己的应用时，所有的这些程序或应用作为一种基础设施（supporting framework）不可或缺。

除了这些基础技术（enabling technology）[①]之外，还有数不胜数的框架和工具库可以帮助我们为应用添加特性。其中，**开放式API**是最令人激动的领域之一。它们是一些知名（或不太知名）网站提供的API，你可以将这些技术集成到你自己的应用中。

假设你希望给某个应用增加地图，或是Web搜索，抑或是照片维护和共享功能。我们自己实现任意一个特性，都将是艰巨无比的任务。但是，值得高兴的是，有人已经实现了这些应用并愿意把它们的功能服务接口开放给我们。

对应用开发人员来说，这是一种莫大的好处。对我们开放API显示了他们的慷慨，接受这种馈赠，我们就能有效地利用他人的劳动成果。没错，有人可能将这种屡见不鲜的“慷慨”视为一种商业策略，以促使开发人员对非免费的高级服务感兴趣。不过，只要这些服务提供商预先声明[②]并且不存在欺诈，它就是一种合法的营销行为。

本章中，我们将提供一些代码实例，它们将Yahoo!地图、Geocoding及Traffic、Google Web搜索和Flickr照片等特性集成到支持Ajax的Web应用中。在实现该方案时，我们将研究一个结合多

① 主要指基础技术和平台，如前文提到的操作系统、应用服务器和浏览器等各种软件。——译者注

② 这里指的是服务在功能和费用上的差别。——译者注

种技术的出色技巧，以便在利用Ajax构建的Web页面中集成开放式Web API。

好吧，让我们继续进行深入研究并从中汲取乐趣！

12.1 Yahoo!开发者网络

Yahoo！（http://developer.yahoo.com/）开发者网络提供了大量的Web服务，包括旅游、购物和招聘等。要使用如此众多的服务，需要从Yahoo！获取一个应用识别码。它可以是你选取的某个字符串（很像用户名），调用任何Yahoo！Web服务API都使用它作为请求参数，以识别是哪个应用在调用这些服务。

Yahoo！开发者网络上有很多文档链接指导你获取应用识别码。申请应用识别码的页面URL是http://api.search.yahoo.com/webservices/register_application/。注意，要访问该页面，你必须登录到你的Yahoo！账号。

Yahoo！服务器能确保注册期间所有的应用识别码都是唯一的。为了保证它的唯一性并且容易记住，你可以采用类似Java语言的包命名规范。比如说，你拥有自己的域名：yourlastname.org。在你所有的Java项目中，包名都会以org.yourlastname.projectname开始，其中*projectname*是为某个具体项目选取的名称。同样，如果要在项目中使用Yahoo！Web服务，那你可以注册一个名为org.yourlastname.projectname的应用识别码。这通常能保证识别码的唯一性，除非有人强行使用你的包命名方式。

一旦注册应用识别码成功，你就可以立即使用Yahoo！Web服务了。

12.1.1 Yahoo!地图

假设你和你的朋友们是勇敢的飓风追逐者。每年春天风暴来临之际，你们都扛起设备，满怀憧憬地离开家门，希望自己拍摄的录像带能在天气频道播出。

1. 问题

你的朋友们拥有高端的全球定位系统（GPS）设备，它能实时显示地图。唉，你的设备有点简陋，但还可靠，它只能提供经、纬度坐标。你没有多余资金来更换更先进的设备，不过有一台通过手机连接互联网的笔记本电脑。你还拥有编程技能，并且你掌握了Ajax！

因此，就让你的朋友们去玩他们的GPS系统吧。我们将构建一个自己的地图应用系统！

2. 解决方案

Yahoo！地图（http://developer.yahoo.com/maps/index.html）服务提供了多种可选API，不过我们仍将选用该服务的Ajax API，以便利用我们的JavaScript和DHTML知识。

和其他Yahoo！服务（稍后我们会在示例中看到）不同，Yahoo！Maps服务的Ajax API将Ajax功能封装并提供内建的JavaScript API。因此，我们只需导入Yahoo！Maps服务API并引用它提供的对象和函数，而不是自己实现Ajax调用。

首先，我们必须从Yahoo！服务引用其提供的API。在文档<head>标签内，我们添加如下元素导入脚本：

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

```
  src="http://api.maps.yahoo.com/ajaxymap?v=2.0&appid=YahooDemo">
</script>
```

我们还要通过以下脚本导入Prototype：

```
<script
  type="text/javascript" src="prototype-1.5.1.js ">
</script>
```

我们从GPS设备获得（当前位置的）纬度和经度，并输入到应用页面的表单中。在表单（不把该表单提交到任何服务器）输入数据之后，我们希望显示一张以该坐标为中心的地图。

先创建一个简单的表单，如下所示（是否美化它，随你便）：

```
<form name="mapForm" onsubmit="showMap();return false;">
  <div>
    Latitude: <input type="text" id="latitude"/>
    Longitude: <input type="text" id="longitude"/>
    <input type="submit"/>
  <div>
</form>
```

注意，该表单的onsubmit事件处理器触发一个JavaScript函数调用，并通过返回false来有效阻止表单提交。

除这个表单外，我们还需要在某个区域显示地图本身。Yahoo！Maps服务API希望我们给它传递一个元素，以便在该元素上将“画”出地图。因此我们在页面的适当位置上添加一个地图容器元素，该元素初始状态为空白。

```
<div id="theMap" style="width:600px;height:480px;"></div>
```

在文档加载后，我们希望创建并初始化服务API的核心对象：map[①]。之所以在页面的onload事件处理器中实现，是因为在此之前地图容器元素已经存在：

```
var map;
window.onload = function() {
  map = new YMap($('theMap'));
  map.addPanControl();
  map.addZoomLong();
};
```

上述代码创建了一个可访问的[②]YMap实例，它将在传入元素上绘制地图。我们还给地图添加两个可选控件：方向指示盘控件和缩放控件。

当表单提交（因此调用showMap()函数）后，我们创建一个YGeoPoint实例并将其作为参数传入YMap对象的drawZoomAndCenter()函数，然后命令YMap实例在以指定坐标为中心的区域绘制一张地图。showMap()函数如下所示：

```
function showMap() {  var zoomLevel = 4;
  var latitude = $F('latitude');
  var longitude = $F('longitude');
  var point = new YGeoPoint(latitude,longitude);
```

① 即YMap实例。——译者注

② 原文是reusable，这里指的不是一般意义上的重用，更准确的说是在应用中可访问。——译者注

```
    map.drawZoomAndCenter(point,zoomLevel);
}
```

我们随意选定缩放级别4，它看似是个不错的常规初始值。一旦地图加载后，我们可以调节缩放级别，因为地图上已经设置了缩放控件。不过，如果愿意，我们可以在表单上添加另一个字段，以便设置不同于4的初始缩放级别。

假设我们从GPS设备上读出了纬度和经度值分别是30.27和-97.74，并把它们输入到我们的应用中，我们就可以看到如图12-1所示的地图。代码清单12-1列出了完整的代码，可以从www.manning.com/crane2下载源代码，并在本章部分找到它们。

代码清单12-1 Yahoo!地图页面

```
<html>
  <head>
    <title>Where Am I?</title>
    <script
      type="text/javascript"
      src="http://api.maps.yahoo.com/ajaxymap?v=2.0&appid=YahooDemo">
    </script>
    <script type="text/javascript" src="prototype-1.5.1.js"></script>
    <script type="text/javascript">

      var map;

      window.onload = function() {
        map = new YMap($('theMap'));
        map.addPanControl();
        map.addZoomLong();
      }

      function showMap() {
        var zoomLevel = 4;
        var latitude = $F('latitude');
        var longitude = $F('longitude');
        var point = new YGeoPoint(latitude,longitude);
        map.drawZoomAndCenter(point,zoomLevel);
      }
    </script>
  </head>

  <body>

    <div>
      <form name="mapForm" onsubmit="showMap();return false;">
        Latitude: <input type="text" id="latitude"/>
        Longitude: <input type="text" id="longitude"/>
        <input type="submit"/>
      </form>
    </div>

    <div id="theMap" style="width:600px;height:480px"></div>

  </body>

</html>
```

图12-1　现在我们知道自己在哪

3. 讨论

这个应用是不是非常酷？谁还需要那些花哨的GPS设备？

仅仅通过观察页面代码，我们根本无法了解，近似于服务器通信机制的所有Ajax功能隐藏在Yahoo！地图服务的JavaScript API背后。不过，代码明显在某处连接了一些服务器并获取了用于显示地图的数据。

还要注意，本例并不需要我们刚刚得到的Yahoo！开发者网络的应用识别码。不过我们马上就会看到一个需要识别码的示例。

这个简单页面适合显示指定经度和纬度的地图。不过，有时候或许我们希望应用支持输入传统的街道名称。Yahoo！Maps的YMap API对绘制指定街道名称的地图无计可施，但我们不能就此罢手。Yahoo！还提供了Geocoding API。

12.1.2　跨服务器代理

Yahoo！Geocoding REST API（http://developer.yahoo.com/maps/rest/V1/geocode.html）支持测量指定地址位置的经度和纬度。与上节讨论的Yahoo！Maps API不同，Geocoding API是一种**表述性状态转移**（REST）接口。

REST是一种简单HTTP接口。通过它，我们可以在URL末尾添加适当的请求参数并接收一个由XML、JSON、HTML乃至纯文本组成的响应，所以不再需要构造诸如SOAP（Simple Object Access Protocol，简单对象访问协议）那样的额外层。

太虚幻了!这能简化什么？我们继续前进，在Web页面上编写一个Ajax请求，通过给它设置必

需的请求参数来访问某个特定的URL以检验这种接口的能力，我们希望能收到包含各种奇妙信息的响应。

没想到浏览器阻拦我们这样做，并且警告我们不允许执行跨浏览器的脚本！

怎么回事？

1. 问题

原来，我们遇到了Ajax安全沙箱（Ajax security sandbox）问题。为安全起见，浏览器只允许Ajax访问首次给该浏览器发送页面的那台服务器。

可恶！很明显它阻碍了我们的计划。如果不能运行跨浏览器的脚本，通过Ajax访问REST API的可能性就极其渺茫了。

果真如此吗？别忘了，我们也不是等闲之辈。

大家知道，我们可以随意创建访问我们自己的服务器的Ajax请求。而且，一旦请求到达服务器，就不必再为安全沙箱的问题担忧，它可以向我们希望的任何服务器发起请求。

因此，我们确实需要一个代理，它运行在我们自己的服务器上。作为一个代理，它帮我们实现将请求发送到提供REST API的远程服务器，并将该服务器传回的请求结果转发给我们。

2. 解决方案

我们将研究一个基于Java的解决方案，它适应于运行在servlet容器上的所有Web应用。本解决方案完全可以使用其他的服务端技术实现，例如PHP、以Perl或cURL[①]编写的非常古老的CGI脚本，甚至采用wget[②]编写简单的shell脚本。

设计思想是创建一个代理，我们可以向它发起请求，而由代理把这些请求转发到其他的服务器、接收远程服务器的响应，并最终将响应回复给我们。代理通常是一种有用的工具，但对我们而言，最为重要的是它绕过了Ajax安全沙箱的限制。

来看几张示意图，它们将帮助我们理解其中的奥秘。考虑图12-2。

步骤(1)，我们的本地服务器booboo接收到来自客户端的一个请求（A），然后返回HTML页面，客户端浏览器加载并显示该页面。在步骤(2)中，由于页面触发某些事件，一个访问远程服务器yogi的Ajax请求（B）产生了，该远程服务器提供了一些我们想要的Web服务。

但是，这不可能实现！浏览器知道我们的页面是从booboo提供的，它阻止任何试图发送到其他服务器的请求。不过，“条条大路通罗马”。现在考虑图12-3。

在这个新场景中，浏览器发送请求到服务器并接收响应（A），然后加载页面，和之前步骤(1)的过程是一样的。不过现在，当我们想发送Ajax请求到yogi服务器，以访问部署在该服务器上的Web服务时，我们可以发送该请求（B）到服务器booboo。它是我们最初访问的本地服务器，如步骤(2)所述。代理servlet将重新发起到目标服务器的请求，该请求还包含服务器标识信息和所访问服务的描述信息。

步骤(3)，代理servlet发送请求（C）至远端yogi服务器，访问指定的Web服务。因为离开了浏览器，我们摆脱了Ajax安全沙箱的限制范围——后面的路就平坦了，远程服务器返回一个对应请

① 一个利用URL语法在命令行下工作的文件传输工具，支持文件的上传和下载，所以是综合传输工具，但按传统，习惯称cURL为下载工具——http://curl.haxx.se/。——译者注

② 和cURL类似的一个工具。——译者注

求的响应。

最后，步骤(4)，服务器booboo上的代理将从yogi服务器接收到的响应作为它自身的请求响应返回到客户端（B）。

Ajax安全沙箱对此毫不知情。

那么，来看看我们采取了哪些编程技巧以绕开安全沙箱的限制。我们可以把所有的代码都放到一个servlet中实现，但考虑再三我们才意识到，由于我们希望该实现能在多种环境下通用——命令行程序或后台程序，例如，甚至是Swing应用——因此我们应当创建一个可重用的代理组件。

那就开始吧。我们将创建一个UI无关类，它可以用于任何Java程序以通过HTTP协议从远程服务器获取内容。

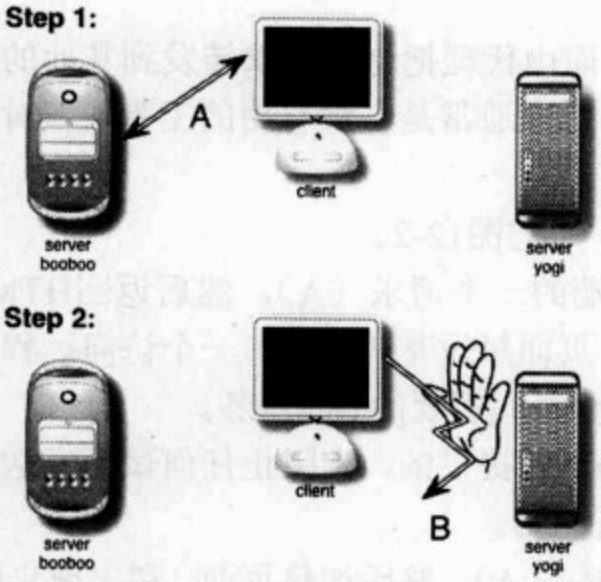


图12-2　Ajax安全沙箱的限制

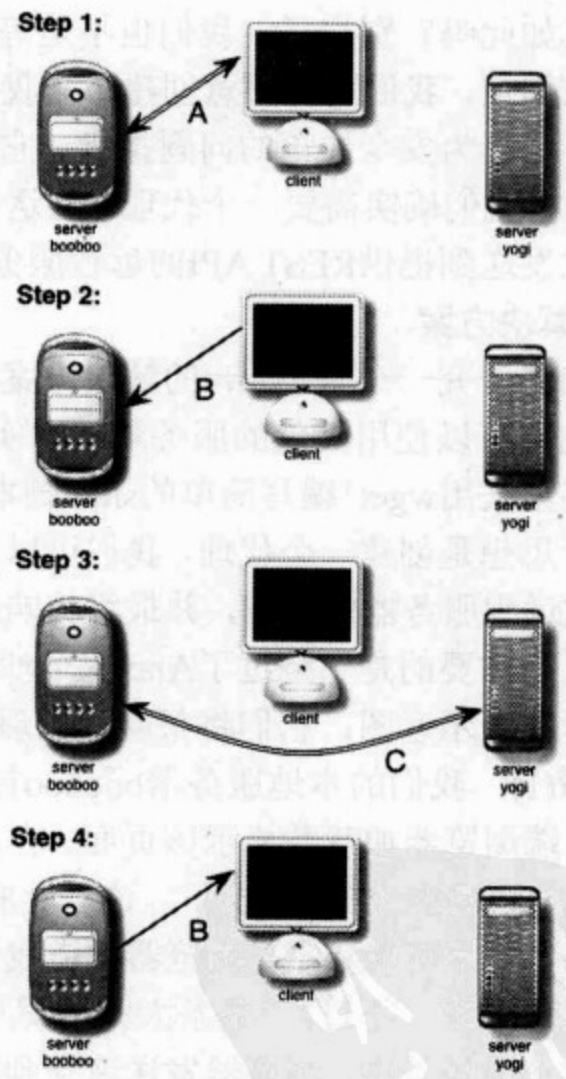


图12-3　我们绕开Ajax安全沙箱的限制

● 内容获取器

此刻，我们面临一个经常遇到的问题，即如何在“自建、购买或者借用”之间进行选择。是利用Java的网络功能包（Java networking packages）自己实现，还是利用他人已完成工作的核心部分？

在该内容获取器的实现中，我们决定采用该领域内其他程序员已完成的工作成果。毕竟，本章内容不就是在利用别人慷慨提供的代码吗？

在Apache Jakarta项目中有一个叫做HttpClient的组件，我们将使用它来实现内容获取器。HttpClient是开源组件，它能轻松模拟HTTP客户端的行为，它因此而得名。

可以在http://jakarta.apache.org/commons/httpclient/浏览该项目信息或下载。该组件所需的JAR文件已经作为本章可下载源代码的一部分。

因为使用了该组件，ContentGrabber类变得相当简单。代码清单12-2显示了它的实现代码。

代码清单12-2 ContentGrabber类

```
package org.bibeault.rest;

import java.util.*;
import org.apache.commons.httpclient.*;
import org.apache.commons.httpclient.methods.*;

public class ContentGrabber {

  private String url;
  private String content;
  private String contentType;
  private Integer contentLength;

  public ContentGrabber(String url,
                        Map<String,String[]> parameters) {
    this.url = url;
    try {
      HttpMethod method = new GetMethod( url );
      List<NameValuePair> params = new ArrayList<NameValuePair>();
      for (Map.Entry<String,String[]> entry : parameters.entrySet()) {
        for (String value : entry.getValue()) {
          params.add(new NameValuePair(entry.getKey(), value));
        }
      }
      method.setQueryString(
        params.toArray(new NameValuePair[params.size()]));
      new HttpClient().executeMethod(method);
      this.content =
        method.getResponseBodyAsString();
      Header contentTypeHeader =
        method.getResponseHeader("content-type");
      Header contentLengthHeader =
        method.getResponseHeader("content-length");
      if (contentTypeHeader != null)
        this.contentType = contentTypeHeader.getValue();
      if (contentLengthHeader != null)
        this.contentLength =
          Integer.parseInt(contentLengthHeader.getValue());
      method.releaseConnection();
    }
    catch (Exception e) {
      e.printStackTrace();
      throw new RuntimeException("Error obtaining content from " +
          this.url + ": " + e, e);
    }
  }
}
```

❶ 导入HttpClient相关类

❷ 声明实例变量

❸ 定义构造函数

❹ 创建执行GET方法的HttpClient实例

❺ 执行GET方法

❻ 获取响应

```
    public String getUrl() { return this.url; }
    public String getContent() { return this.content; }
    public String getContentType() { return this.contentType; }
    public Integer getContentLength() { return this.contentLength; }

    public static void main(String[] args) {
      Map<String, String[]> params = new HashMap<String, String[]>();
      params.put("appid", new String[] {"org.bibeault.aip"});
      params.put("location", new String[] {"78701"});
      System.out.println(
        new ContentGrabber
          ("http://api.local.yahoo.com/MapsService/V1/geocode",
                        params)
          .getContent());
    }

}
```

❼ 定义url属性访问器

从命令行测试Content-Grabber类 ❽

和其他的Java类相似，在该类的起始部分导入需要的外部类❶。我们导入了HttpClient类，还有java.util包中的所有类。

我们声明了许多变量❷以保存输入URL、还有执行HTTP GET方法后的返回结果：主体内容、内容类型和内容长度。由于不是所有GET响应的首部都包含内容类型和长度，我们必须稳妥地处理该问题。所有的contentType和contentLength变量都被初始化为null，并且在没有返回相应的首部时仍保留null值。注意，我们将contentLength变量定义为Integer类型，而不是int，这样一来，我们可以检测它是否是一个null值。

在ContentGrabber构造函数中❸，根据传入的参数即URL和可选查询参数集合，创建并执行GET请求。注意，我们使用Java 5的范型来表示构造函数的传入参数map，它用字符串变量作为主键，用字符串数组作为值。在老版本的JDK中，可以不使用范型（但会导致失去范型所提供的类型安全）。

在构造函数中执行GET请求，设计是否合理，的确值得商榷。完全可能存在这样的情况，在ContentGrabber构造函数和访问可用响应属性之间存在延迟。该类可以重构为迟加载模式，当任何属性访问器被调用时就执行对象检测过程。但是，这不是需求问题，因此目前我们尽量保持代码简单。

该类的主要操作就是创建一个GetMethod实例，它表示我们希望执行的某个HTTP方法❹。传入参数URL并使用它创建GetMethod实例，接下来，在该实例的Map参数转换为需要的NameValuePair数组完毕之后，设置该示例的查询参数。

当GetMethod实例准备好执行时，重新创建一个HttpClient实例并用于执行该GetMethod实例的方法❺。检查响应❻并且将它的主体内容赋给内容变量。如果返回内容类型和首部长度，它们的值也被存储在相应的变量contentType和contentLength中。

该类定义了很多属性访问器，它们用于本类的调用者获取响应结果❼。

最后，该类包含一个main()方法❽，它用于对本类的方法进行一个简单的测试。调用者不能调用此方法。

试试看，把该类加载到IDE开发环境，或在命令行中编译并运行它。无论哪种情况，你都应该能看到从远程服务返回的一个XML文件的打印输出。

既然掌握了获取远程服务内容的方法，让我们来把它添加到工作示例中。

● 跨服务器代理servlet

在实现ContentGrabber类之后，我们准备创建真正的Ajax请求代理类。由于我们已经在内容截取器中类中完成了所有的繁重的编码工作，因此代理servlet实际上变得极其简单，如代码清单12-3所示。

代码清单12-3 CrossServerProxy类

```
package org.bibeault.rest;

import javax.servlet.ServletException;
import javax.servlet.http.*;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

public class CrossServerProxy extends HttpServlet {

  public static final String KEY_SERVICE_URL = ".serviceUrl.";

  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
          throws ServletException, IOException {
    String serviceUrl =
      request.getParameter(KEY_SERVICE_URL);
    if (serviceUrl == null) {
     throw new ServletException
                    ("the " + KEY_SERVICE_URL +
                     " parameter must be provided");
    }
    Map parameters = new HashMap();
    parameters.putAll(request.getParameterMap());
    parameters.remove(KEY_SERVICE_URL);
    ContentGrabber grabber =
      new ContentGrabber(serviceUrl, parameters);
    if (grabber.getContentType() != null)
       response.setContentType(grabber.getContextType());
    if (grabber.getContentLength() != null)
      response.setContentLength(grabber.getContentLength());
    response.getWriter ()
      .print(grabber.getContent());
  }
}
```

❶ 支持GET请求

❷ 获取URL服务参数

❸ 如果找不到所需服务参数就抛出异常

❹ 从拷贝的映射表中删除服务URL

❺ 构造内容获取器实例

❻ 设置内容类型和长度

❼ 中转代理响应的内容

该servlet支持GET请求❶。要支持POST方法也很简单，在doPost()方法中简单调用doGet()方法并传递其请求和响应参数就可以实现。

必须为该servlet提供一个参数❷，它用于表示代理请求要求访问的URL。注意，我们为这个参数起了一个不寻常的名字，它的开头和末尾都包含一个句号。为什么要这样处理?

由于无法预先得知代理请求有那些参数，因此我们把所有到达的请求添加到代理类的请求队

列中。也就是说，除了我们用于表示基本URL的个别参数之外，所有请求参数都加入到队列。

这就是我们为何要把它的名字格式搞得如此古怪的原因。也许这种用法极为罕见，但是我们感兴趣的某个Web服务的请求参数定义中也包含句号的可能性也微乎其微。这么做可能导致JavaScript代码无法引用HTML文档DOM树中的元素。与此同时，我们也无法使用点符号，应该使用不存在引用问题的普通方括号。因此，使用句号命名保留参数并不可靠，它可能污染我们要使用的任何Web服务的参数命名空间。

之后，我们检查是否已经提供必需的服务URL参数❸，并且针对参数遗漏的情况做一些处理。

将参数的副本传递给代理servlet，用来传递给内容获取器❹。首先，复制请求参数映射表，然后移除服务URL的参数。

为什么要复制参数列表？为什么不直接用`getParameterMap()`方法返回的Map实例呢？记住，Map是一个接口，并且在这个接口中`remove()`是备选方法。我们并不知道从`getParameterMap()`方法返回的Map是否实现了`remove()`方法。并且事实上，在Tomcat 5.5环境下，返回的Map并没有实现该方法。

通过把返回的Map复制给已实现`remove()`的类实例，这里也就是HashMap，我们就可以确保成功调用`remove()`方法。

然后，内容获取器实例就构造完毕❺，并且我们为它指定了代理URL及其参数。它触发一个跨服务器访问代理URL调用。一旦完成，代理请求响应的内容类型和长度就会被存储在我们自己的响应中❻。

最后，代理响应的内容作为我们自己的响应的内容返回❼。

3. 讨论

介绍完该解决方案之后，至少对基于Java语言的Web应用来说，阻止我们在页面使用Ajax发起Web服务调用的安全沙箱问题已经不复存在。正如我们提到的那样，可以使用其他的服务端技术来实现类似的解决方案。

为了保证示例紧扣当前主题，本章介绍的其他大部分示例代码几乎没有多少错误检查。在真正的产品代码中，必须加入更多的错误检查以便增强代码的健壮性。

我们创建了一个内容获取器类，它允许我们通过指定URL和一个请求参数集合来实现从其他站点获取内容。该类非常依赖Jakarta的HttpClient项目提供的工具库。如果你准备自己实现类似的库，请研究java.net.URL和java.net.URLConnection等类并作为起点。

要明白，自己实现类似HttpClient的工具可能很复杂。假设你想使用的Web服务需要处理cookie，那该怎么办？动手实现任何代码库都将是个复杂的问题。要扩展内容获取器或在这个以及其他类似领域中增加功能，HttpClient已经包含所需的工具库。

跨服务器请求代理servlet利用内容获取器作为页面上Ajax请求的代理人，把它添加到应用中很简单。显然，要把它放到应用的类路径下。还需要使用两个简单的元素在部署描述文件（`web.xml`）中添加声明和映射关系。首先，声明servlet本身：

```
<servlet>
  <servlet-name>CrossServerProxy</servlet-name>
  <servlet-class>org.bibeault.rest.CrossServerProxy</servlet-class>
```

```
    <load-on-startup>4</load-on-startup>
  </servlet>
```

之后声明servlet的URL映射关系：

```
<servlet-mapping>
  <servlet-name>CrossServerProxy</servlet-name>
  <url-pattern>/proxy</url-pattern>
</servlet-mapping>
```

这里，我们决定使用/proxy作为servlet的路径。下一节，我们将好好利用这个代理。

12.1.3 Yahoo! Maps Geocoding

借助上节创建的代理类，现在我们能够调用远程服务器上的服务了。我们准备强化地图解决方案的功能，无论给出经度纬度坐标还是某个街道，都给大家显示一张地图。

1. 问题

正如前面指出的那样，要显示地图，Yahoo! Maps API需要经度纬度坐标，因此如果允许使用地理位置作为输入，我们需要将地理位置转换成其相应的经度纬度坐标。

2. 解决方案

我们实现该解决方案的途径是Yahoo! Maps Geocoding REST（http://developer.yahoo.com/maps/rest/V1/geocode.html）。之前我们介绍过Yahoo!Maps API，它根据经度纬度坐标显示地图，与此不同的是，Geocoding API基于REST协议。我们将使用合适的参数发起请求访问提供Geocoding服务的URL，而响应将是一个包含请求信息的XML文档。该服务的URL地址是http://api.local.yahoo.com/MapsService/V1/geocode。

注意，在本书的写作期间，该服务仅限于在美国地区使用。请参考上面提及的网页了解有关该服务支持地区的最新信息。

我们将发送两个必需的参数。它们是在Yahoo！注册的应用程序识别码以及我们希望转换的地理位置字符串。该服务还接受其他参数，但我们将关注这两个参数。

一个服务请求示例或许如下所示：

```
http://api.local.yahoo.com/MapsService/V1/geocode?
  appid=your.yahoo.app.id&location=78701
```

在浏览器的地址栏输入上述地址并查看有何结果。请确认你输入了正确的Yahoo！应用识别码。注意，你可以提供尽可能少的信息如邮政编码，但仍然可以获取到有意义的结果。

掌握了这些知识后，我们开始修改地图应用页面。首先，我们需要添加一个新的字段以显示地理位置。由于希望区别于现有的经纬度表单，因此我们打算单独提供一个表单。在现有的mapForm表单前面，我们添加以下内容：

```
<div>
  <form name="geocodeForm"
        onsubmit="findLocation();return false;">
    Location: <input type="text" id="locationField"
                     name="location" style="width:200px;"/>
    <input type="submit"/>
  </form>
</div>
```

上述代码为页面添加了一个大文本字段，它能容纳字数较多的文本，我们可以在该字段上输入地理位置。

还有，因为我们计划发起Ajax调用，所以通过在onsubmit事件处理器中返回false，我们可以阻止表单自动提交到服务器。调用findLocation函数将触发请求Geocoding服务：

```
function findLocation() {
  new Ajax.Request(
    '/aip.chap12/proxy',
    {
      method: 'get',
      onSuccess: onCoordsObtained,
      parameters: {
        '.serviceUrl.':
         'http://api.local.yahoo.com/MapsService/V1/geocode',
        appid: 'org.bibeault.aip',
        location: $F('locationField')
      }
    }
  );
}
```

❶ Ajax包含可选参数method

❷ 指定Ajax请求的参数

在该函数中，我们发起了一个访问Geocoding API的Ajax请求，该请求中频繁使用Prototype库为我们提供的接口。说到由单语句组成的函数，它无疑颇为接近！让我们从各个角度进行解释。

传入到Ajax请求的URL相当简单：/aip.chap12/proxy。切记，我们无法直接访问Yahoo! API，因此将通过我们的服务代理servlet路由请求。该URL假设Web应用已经映射到上下文路径/aip.chap12，并且servlet路径/proxy负责对发送到代理servlet的请求进行路由。

对Ajax请求的可选参数❶，我们将它指定为HTTP协议的GET方法，并命令它在请求成功时调用一个名为onCoordsObtained()的回调事件处理器。然后，我们设置要传递到请求的query参数。

由于在本例中使用Prototype 1.5.1，因此我们可以把一个简单的Hash对象❷作为参数传入，该对象将来需转换成一个查询字符串。（如果你使用的是老版本的Prototype，建议研究并尝试新版Prototype提供的Hash对象声明功能，它能帮助你摆脱手工方式创建查询字符串的繁琐过程。）

来查看我们要指定的请求参数。参数.serviceUrl.通知代理servlet客户在请求哪个服务，参数appid指定了应用在Yahoo！开发者网络上注册的应用识别码（本例中，我们为这些示例注册为同一个应用识别码），还有参数location将用户在位置字段输入的内容传入。注意，参数服务URL的命名规则有点古怪，由于包含点符号，必须使用引号把它声明为合法的字符串。

如果请求调用一切顺利，我们指定的回调事件处理器onCoordsObtained()将被调用。其实现如下：

```
function onCoordsObtained(request) {
  var xml = request.responseXML;
  document.mapForm.latitude.value =
    xml.getElementsByTagName('Latitude').item(0).firstChild.data;
  document.mapForm.longitude.value =
    xml.getElementsByTagName('Longitude').item(0).firstChild.data;
  showMap();
}
```

在成功调用该Ajax请求的事件处理器中，我们传递了通过Prototype创建的一个XHR对象实

例。当Geocoding服务请求成功返回一个简单XML文档表示的响应时，我们获得该文档并处理它以收集服务请求的处理结果。

简单指定位置为78701的Geocoding请求，它返回的XML文档大概如下：

```
<ResultSet xsi:schemaLocation="urn:yahoo:maps
http://api.local.yahoo.com/MapsService/V1/GeocodeResponse.xsd">
  <Result precision="zip">
    <Latitude>30.271</Latitude>
    <Longitude>-97.741</Longitude>
    <Address/>
    <City>AUSTIN</City>
    <State>TX</State>
    <Zip>78701</Zip>
    <Country>US</Country>
  </Result>
</ResultSet>
```

对本例而言，我们唯一关心的文档字段是<Latitude>和<Longitude>。我们的函数使用XML DOM API定位这些字段并将经度和纬度保存到mapForm的相应字段中。

最后，我们的事件处理器调用已声明的showMap()函数以显示指定坐标的地图。

3. 讨论

通过使用Yahoo! Maps Geocoding REST API，我们不仅能输入经度纬度，此外还可以输入位置信息。好处是，我们能把返回的经度、纬度值显示到对应的表单字段中，还可以显示正确的地图。

代码还可以写得更健壮。例如，如果服务无法确定输入位置的经纬度坐标，它返回一个失败响应，而实际代码则忽略了这一点。虽然它还不至于让我们大发雷霆，它或许能在错误发生时以更加智能的方式通知客户。图12-4显示了修改后的页面的显示结果。

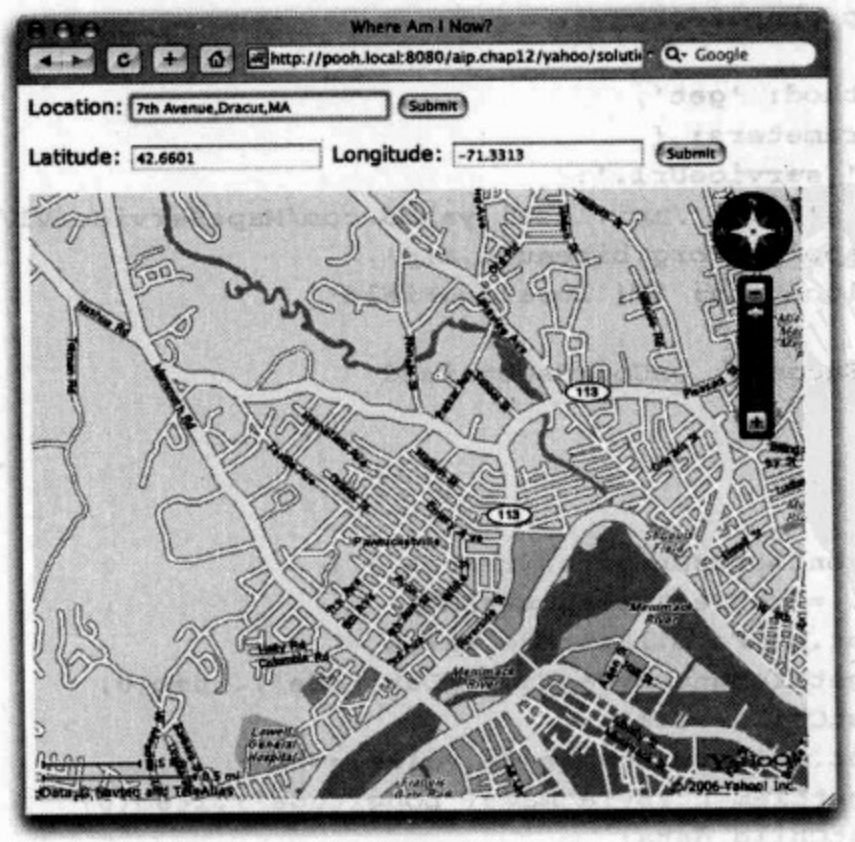


图12-4 改进后的页面增加了地址栏

可以在可下载源代码的本章部分找到扩展后的页面的完整代码，如代码清单12-4所示（更改和新增的部分以粗体字表示）。

代码清单12-4　使用Geocoding显示Yahoo! 地图

```
<html>
  <head>
    <title>Where Am I Now?</title>
    <script type="text/javascript"
      src="http://api.maps.yahoo.com/ajaxymap?v=2.0&appid=YahooDemo">
    </script>
    <script type="text/javascript" src="prototype-1.5.1.js"></script>
    <script type="text/javascript">
      var map;

      window.onload = function() {
        map = new YMap($('theMap'));
        map.addPanControl();
        map.addZoomLong();
      };

      function showMap() {
        var zoomLevel = 4;
        var latitude = $F('latitude');
        var longitude = $F('longitude');
        var point = new YGeoPoint(latitude,longitude);
        map.drawZoomAndCenter(point,zoomLevel);
      }

      function findLocation() {
        new Ajax.Request(
          '/aip.chap12/proxy',
          {
            method: 'get',
            parameters: {
              '.serviceUrl.':
                'http://api.local.yahoo.com/MapsService/V1/geocode',
              appid: 'org.bibeault.aip',
              location: $F('locationField')
            },
            onSuccess: onCoordsObtained
          }
        );
      }

      function onCoordsObtained(request) {
        var xml = request.responseXML;
        document.mapForm.latitude.value =
          xml.getElementsByTagName('Latitude').item(0)
          .firstChild.data;
        document.mapForm.longitude.value =
          xml.getElementsByTagName('Longitude').item(0)
          .firstChild.data;
        showMap();
```

```
    }
  </script>
</head>
<body>
  <div>
    <form name="geocodeForm"
          onsubmit="findLocation();return false;">
      Location: <input type="text" id="locationField"
                       name="location" style="width:200px;"/>
      <input type="submit"/>
    </form>
  </div>

  <div>
    <form name="mapForm" onsubmit="showMap();return false;">
      Latitude: <input type="text" name="latitude"/>
      Longitude: <input type="text" name="longitude"/>
      <input type="submit"/>
    </form>
  </div>
  <div id="theMap" style="width:600px;height:480px;"></div>
</body>
</html>
```

12.1.4 Yahoo!交通

我们是勇敢的户外摄影师，因此有必要关心旅途中的路况和交通情况（前一节已经介绍过Map的妙用），因此在展示某些不可思议的风暴视频片段之前，我们想为我们的Yahoo！地图应用页面增加一些功能。

1．问题

我们想了解使用上节创建的页面所显示地图区域内的交通状况。

2．解决方案

Yahoo! Traffic API（http://developer.yahoo.com/traffic/index.html）很适合解决该问题（假设我们在跟踪拍摄美国境内的风暴）。和Yahoo! Maps Geocoding API相似，Traffic API 也是基于REST协议的服务接口，因此我们采用与前一节中的解决方案相同的多种技术把地址转换成经度和纬度坐标。该服务的URL是http://api.local.yahoo.com/MapsService/V1/trafficData。

我们将再次把在Yahoo！注册应用程序识别键和我们感兴趣的地区的交通状况的经度纬度作为必需的参数发送到服务器。和Geocoding API相似，交通服务接受多个其他参数，但实现我们的目的所需的全部参数就是当前地图中的经度和纬度。

我们打算这样实现页面，一旦地图绘制完毕，我们就可以查询该地区有关的任何交通状况。因此，我们需要在地图显示区域下部添加一个按钮：

```
<div>
  <form name="trafficForm" onsubmit="showTraffic();return false;">
      <input type="submit" id="trafficButton"
             value="Show Traffic Alerts" disabled="disabled"/>
  </form>
</div>
```

注意，因为不希望该按钮在地图显示之后可用，我们先把它禁止。在showMap()函数的末尾，我们添加下列代码，使它在地图显示完毕后可用：

```
$('trafficButton').disabled = false;
```

在可点击的traffic按钮之上，showTraffic()方法执行了一个似曾相识的操作：

```
function showTraffic() {
  new Ajax.Request(
    '/aip.chap12/proxy',
    {
      method: 'get',
      parameters: {
        '.serviceUrl.':
          'http://api.local.yahoo.com/MapsService/V1/trafficData',
        appid: 'org.bibeault.aip',
        latitude: $F('latitudeField'),
        longitude: $F('longitudeField')
      },
      onSuccess: onTrafficObtained
    }
  );
}
```

该函数使用和访问Geocoding API相同的方式对Traffic API发起请求，不同之处在于使用的具体参数：提供访问Traffic API服务的URL，将经度和纬度作为查询参数传递，还有指定一个不同的回调函数。

该请求的回调函数和我们为访问Geocoding服务而实现的回调函数大不相同。在后者的实现中，我们只是解析响应返回值并将它们设置到表单中对应的经度纬度字段，如图12-5所示。

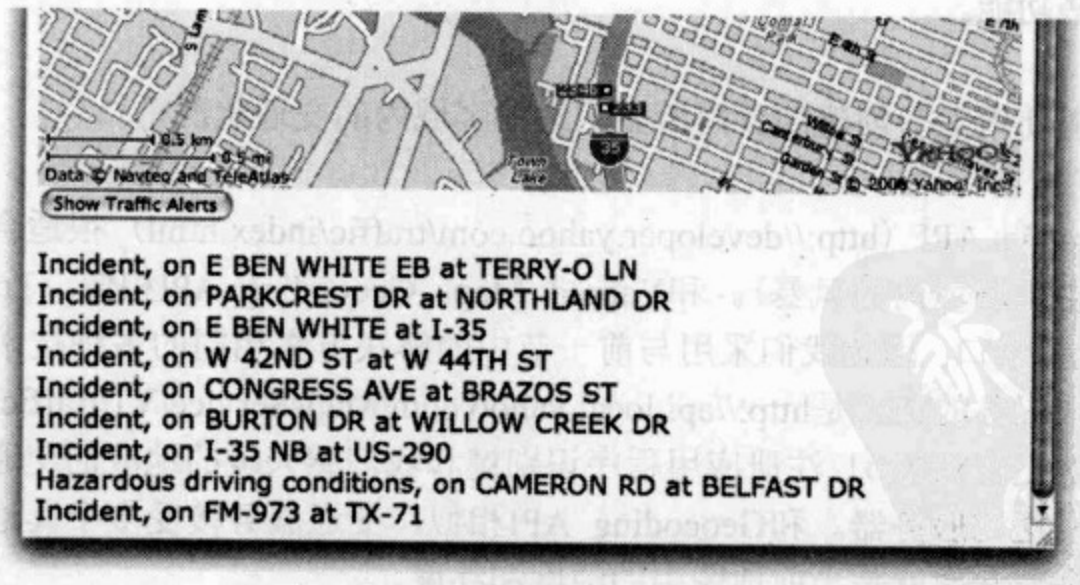


图12-5　增加交通状况区域

对交通状况响应来说，我们想要显示的是页面上指定地区的所有交通事故状况，而不是表单元素。实现该目的有两种常见方法：

- 我们可以使用DOM操作API创建所需的元素节点以显示我们的数据。
- 我们可以创建一个可扩展的字符串缓存并用它设置某个元素的innerHTML属性。

在12.3.1节中，我们将看到第一个技巧，因此这里我们将使用innerHTML方式。添加完这些

功能后，界面显示结果如图12-5所示。

无论哪种情况，我们都需要一个放置新元素的HTML元素，因此在该页面主体内的traffic按钮下方添加：

```
<div id="trafficAlerts"></div>
```

然后，我们实现如下回调函数：

```
function onTrafficObtained(request) {
  var xml = request.responseXML;
  var results =
    $A(xml.getElementsByTagName('Result'));
  $('trafficAlerts').innerHTML = '';
  if (results.length > 0) {
    results.each(
      function(result) {
        $('trafficAlerts').innerHTML +=
          result.getElementsByTagName('Title').item(0)
          .firstChild.data +
          '<br/>';
      }
    )
  }
  else {
    $('trafficAlerts').innerHTML = 'No incidents reported';
  }
}
```

❶ 获取所有的<Results>子节点元素

❷ 读取<Title>子节点信息，拼装信息

该服务的响应是XML格式，它由一个包含多个<Result>元素的<ResultSet>元素组成。每个<Result>元素包含一个已报道的交通事故信息。<Result>元素的子元素描述交通事故的各种信息，甚至包括地理位置坐标。不过由于我们只对事故的大致信息感兴趣，因此我们将提取每个<Result>的<Title>子元素的内容。

下面是一个Traffic API请求的响应文档，这里显示的内容有省略：

```
<ResultSet xsi:schemaLocation="urn:yahoo:maps
http://api.local.yahoo.com/MapsService/V1/TrafficDataResponse.xsd">
  <LastUpdateDate>1144350730</LastUpdateDate>

  <Result type="incident">
        <Title>Incident, on E OLTORF ST at BENJAMIN ST</Title>
        <Description>COLLISION PRIVATE PROPERTY J</Description>
        <Latitude>30.231270</Latitude>
        <Longitude>-97.734753</Longitude>
        <Direction>N/A</Direction>
        <Severity>3</Severity>
        <ReportDate>1144350420</ReportDate>
        <UpdateDate>1144350705</UpdateDate>
        <EndDate>1144352505</EndDate>
  </Result>

/* more Result elements removed to save space */

</ResultSet>
```

该回调函数执行从响应文档❶获取所有<Result>操作（使用Prototype库的$A()函数很容易将节点列表转换为数组）、从<Result>节点❷查找<Title>元素，并将这些元素的内容格式化以设置页面上名为trafficAlerts的 元素。

将输出进行装饰使之更加好看，这或许是你早就想做的事情。

3. 讨论

在Yahoo! Traffic REST API的帮助下，我们为该地图应用增加了查询所在区域的交通状况的功能，这能让我们更加有效地工作——在风暴袭来之前这尤为重要。

本解决方案使用和前面的Geocoding类似的实现机制，但我们使用一个 元素的innerHTML属性动态地直接把文本添加到页面，而不是将返回信息填充到表单元素上。尽管该机制非常直接和简洁，但它并不总是最佳选择。

我们的页面产生的标记非常简单：文本字符串以
标签分割。如果我们决定试图添加更复杂的标记来修饰信息的风格和外观，我们会很快发现创建和维护以文本字符串组成的标签不是一件愉快的事情。

通过客户端控件在页面动态添加元素的另外一个机制是使用DOM API，可以直接把元素添加到页面HTML DOM树上。和使用innerHTML机制相比，尽管这会引入更多代码，但这种方式实现并不费劲，而且和使用字符串标记相比，其代码无疑更为清晰。我们将在下一节介绍该技巧。本示例的完整代码如代码清单12-5所示，新增和修改部分以粗体显示，在可下载源代码的本章部分能找到该代码。

代码清单12-5　修改页面增加交通状况提示功能

```
<html>
  <head>
    <title>What's Going On?</title>
    <script type="text/javascript"
    src="http://api.maps.yahoo.com/ajaxymap?v=2.0&appid=YahooDemo">
    </script>
    <script type="text/javascript" src="prototype-1.5.1.js"></script>
    <script type="text/javascript">
      var map;

      window.onload = function() {
        map = new YMap($('theMap'));
        map.addPanControl();
        map.addZoomLong();
      };

      function showMap() {
        var zoomLevel = 4;
        var latitude = $F('latitudeField');
        var longitude = $F('longitudeField');
        var point = new YGeoPoint(latitude,longitude);
  map.drawZoomAndCenter(point,zoomLevel);
  $('trafficButton').disabled = false;
}
```

```
function findLocation() {
  new Ajax.Request(
    '/aip.chap12proxy',
    {
      method: 'get',
      parameters: {
       '.serviceUrl.':
        'http://api.local.yahoo.com/MapsService/V1/geocode',
       appid: 'org.bibeault.aip',
       location: $F('locationField')
      },
      onSuccess: onCoordsObtained
    }
  );
}

function onCoordsObtained(request) {
  var xml = request.responseXML;
  document.mapForm.latitude.value =
    xml.getElementsByTagName('Latitude').item(0).
    firstChild.data;
  document.mapForm.longitude.value =
    xml.getElementsByTagName('Longitude').item(0).
    firstChild.data;
  showMap();
}

function showTraffic() {
  new Ajax.Request(
    '/aip.chap12/proxy',
    {
      method: 'get',
      parameters: {
       '.serviceUrl.':
         'http://api.local.yahoo.com/' +
         'MapsService/V1/trafficData',
       appid: 'org.bibeault.aip',
       latitude: $F('latitudeField'),
       longitude: $F('longitudeField')
      },
      onSuccess: onTrafficObtained
    }
  );
}

function onTrafficObtained(request) {
  var xml = request.responseXML;
  var results = $A(xml.getElementsByTagName('Result'));
       $('trafficAlerts').innerHTML = '';
       if (results.length > 0) {
         results.each(
           function(result) {
             $('trafficAlerts').innerHTML +=
               result.getElementsByTagName
                 ➥ ('Title').item(0).firstChild.data +
```

```
                '<br/>';
            }
          )
        }
        else {
          $('trafficAlerts').innerHTML = 'No incidents reported';
        }
      }
    </script>
</head>
<body>
  <div>
    <form name="geoForm" onsubmit="findLocation();return false;">
      Location: <input type="text" id="locationField"
                       name="location" style="width:200px;"/>
      <input type="submit"/>
    </form>
  </div>

  <div>
    <form name="mapForm" onsubmit="showMap();return false;">
      Latitude: <input type="text" id="latitudeField"
                       name="latitude"/>
      Longitude: <input type="text" id="longitudeField"
                        name="longitude"/>
      <input type="submit"/>
    </form>
  </div>

  <div id="theMap" style="width:600px;height:480px;"></div>

  <div>
    <form name="trafficForm"
          onsubmit="showTraffic();return false;">
        <input type="submit" id="trafficButton"
               value="Show Traffic Alerts" disabled="disabled"/>
    </form>
  </div>

  <div id="trafficAlerts"></div>

</body>

</html>
```

接下来，我们将不再介绍Yahoo！API的其余部分，还是留给你自己单独研究吧。你还能为我们的地图应用页面添加哪些比较酷的特性？

现在让我们把注意力转到另一个著名的Web服务提供商。

12.2 Google 搜索 API

说到搜索，恐怕没有人会不知道Google这个出众的Web搜索引擎。本节我们将看一看如何使用Google提供的公开API来给我们的页面加上搜索功能。

Google 搜索

很多人都维护着博客或其他网站，其中包含不少记叙文字。在这些文本段落中，我们常常会引入一些对于读者来说可能不大熟悉的术语和概念。

假设我们再一次扮演风暴追逐者的角色，并正在写有关气象状况的报道。其中某些文字段落，我们可能使用到了诸如乳状积雨云（mammatus）、后侧面下曳气流（rear-flank downdraft）、干线（dry line）、中气旋（mesocyclone）或中尺度对流（mesoscale convection）之类的术语。气象爱好者或许熟悉这些概念，但对大多数人来说，这些可不是日常概念。

1. 问题

在博客或其他在线文本页面中提供搜索功能，以帮助读者即时搜索他们不熟悉的术语。我们应该在自己的页面中实现搜索功能，而不是强迫用户打开一个新浏览器窗口，并且我们当然不希望他们离开我们的页面。我们努力工作是为了留住用户，可不是把他们送走！

2. 解决方案

要给页面添加搜索功能，首先要拥有执行搜索的工具。创建自己的搜索引擎显然不太可能，因此我们将利用Java版的Google搜索API来为我们（或者说为我们的用户）执行搜索。

和Yahoo!一样，Google也提供了一些其他API，包括一个老派的SOAP协议API（传闻Google公司已放弃了基于SOAP协议的Web服务体系，而全面转向了RESTful的Web服务体系，因为它具有更好的伸缩性并且更加易于使用）。不过为展示API类型的多样性，我们在本节将采用Java版的API。

一旦我们有了用来搜索的Java类，就可以轻松创建一个Servlet，页面中可以以它为Ajax请求的目标，并取回我们所希望了解的术语的搜索结果。

- 简单的搜索引擎

Google搜索API出奇地简单。只需关注4个类：`GoogleSearch`、`GoogleSearchResult`、`GoogleSearchResultElement`，还有异常类`GoogleSearchFault`，出现问题时会抛出该异常。

不过开始之前，我们必须先得到这些Google搜索Java类，还必须注册一个Google许可证码，这和我们使用Yahoo!服务的方式类似。你可以选择要注册的Yahoo！应用识别码，与此不同，Google将代你生成并为你分配注册码。你可以在www.google.com/apis/找到下载开发包和获取注册码的说明。

在开发包中，你将发现一个名为`googleapi.jar`的文件。请将该JAR文件放入Web应用的`WEB-INF/lib`文件夹下，然后确保它在编译文件的类路径下。

简单设置妥当，我们就准备就绪并编写一个简易的Java类，它使用Google搜索引擎执行搜索任务。当我们执行搜索，假设输入的搜索内容是*outflow boundary*，其查询结果如下所示（为节省空间，有删节）

```
title: <b>Outflow</b> Technologies
url: http://www.outflow.net/
snippet: Offers web design, web marketing, web hosting, and
e-commerce services.

title: <b>Outflow</b>: winds flowing outward from thunderstorms
url: http://ww2010.atmos.uiuc.edu/(Gh)/guides/mtr/svr/dngr/oflow.
rxml
```

```
snippet: <b>Outflow</b>. winds flowing outward from thunderstorms.
 Thunderstorm winds also cause<br>  widespread damage and occasion
al fatalities. Thunderstorm &quot;straight-line&quot; <b>...</b>

title: <b>Outflow</b> Phenomena: downbursts
url: http://ww2010.atmos.uiuc.edu/(Gh)/guides/mtr/svr/comp/out/
home.rxml
snippet: <b>Outflow</b> Phenomena. downbursts. This section is on
visual identification of<br>  macrobursts, microbursts, gust front
s and other <b>outflow</b> phenomena. <b>...</b>

... more ...
```

令人遗憾的是，第一个显示结果和搜索内容有点不匹配，但其他结果符合搜索目标。

创建SimpleGoogleSearch类，如代码清单12-6所示。

代码清单12-6　SimpleGoogleSearch类

```
package org.bibeault.aip.search.google;

import com.google.soap.search.*;

public class SimpleGoogleSearch {

    private GoogleSearch googleSearch;

    public SimpleGoogleSearch(String clientKey) {
        this.googleSearch = new GoogleSearch();
        this.googleSearch.setKey(clientKey);
    }

    public GoogleSearchResultElement[]
        search(String searchTerm)
            throws GoogleSearchFault {
        this.googleSearch.setQueryString(searchTerm);
        GoogleSearchResult googleSearchResult =
            this.googleSearch.doSearch();
        return googleSearchResult.getResultElements();
    }
}
```

❶ 导入Google搜索API

❷ 记录Google搜索引擎

❸ 构建简单的搜索类

❹ 执行搜索请求

并不需要你想象中的那么多代码，是吧？

导入Google搜索API库中的类❶之后，我们声明了一个保存Google搜索引擎示例的变量❷，它是SimpleGoogleSearch类的属性。简单搜索类的构造函数❸并接受Google注册码(从Google网站获得)参数。该注册码由一串随机字符组成，它难于记忆（至少对人类来说），你在登录Google服务网站的时候可以生成它。在构造函数内部，我们创建并保持一个Google搜索引擎类实例，还把传入的注册码传给它。

类实例构造完毕即可执行搜索。该类之所以这样设计，是因为可以使用同一个Google搜索类实例执行多次搜索。所以，我们在构造函数中将Google搜索类实例化并用它随时执行搜索。

通过给search()函数❹传递字符串参数并执行该函数实现搜索，这和在Google网站页面上输入搜索字符串没什么区别。将搜索条目参数设置到搜索引擎并调用其方法doSearch()，该函数

将返回搜索结果的容器对象——一个GoogleSearchResult类实例，它包含搜索结果对象。当结果返回时，存储搜索结果数组的容器对象被获得，搜索结果是GoogleSearchResultElement类的实例。出现任何错误，GoogleSearchFault类的实例都将被抛出。

和内容截取器一样，我们添加了main()函数，它可以用于对类本身执行有限程度的测试，如下所示：

```
public static void main(String[] args) throws Exception {
    if (args.length == 0)
      throw new Exception("A search term must be provided");
    SimpleGoogleSearch searcher =
      new SimpleGoogleSearch("aApewexQFHItVSrlTMDk2iglRbhB+6AR");
    GoogleSearchResultElement[] results = searcher.search(args[0]);
    for (GoogleSearchResultElement result : results) {
      System.out.println("title: " + result.getTitle());
      System.out.println("url: " + result.getURL());
      System.out.println("snippet: " + result.getSnippet() + '\n');
    }
  }
```

继续，将前面所说的程序编译并运行。命令行的首个参数作为搜索条目。

当然，GoogleSearch类拥有选项，远比简单搜索类展示的要多。例如，你可以指定返回的结果数组的起始位置和偏移值，以便在较大的返回结果数组中实现分页，甚至设置语言限制。详细信息请研究GoogleSearch类的API。

- 简单的搜索servlet

既然我们已经拥有了SimpleGoogleSearch类，它能执行搜索必需的绝大部分任务。编写一个Servlet去获取搜索结果并把它作为响应（在我们来说针对的是Ajax请求）返回，应该不难实现。

实际上，我们发现servlet搜索功能的实现近乎繁琐。例如，谁来调用servlet的大部分代码，还有我们的开发时间，以及值得关注的返回数据的格式。

在本章前一节的解决方案中，客户端Ajax调用的响应是XML文档，该文档的内容来自于我们部署的服务器API。不过，在本例中，我们自己动手生成响应，它们来自于搜索结果的GoogleSearchResultElement实例。

我们可以创建XML文档并把它序列化为响应流，仅在客户端去解析它。不过在本例中，由于我们清楚搜索请求是在客户端以JavaScript代码发起的，因此我们需要把搜索结果进行解析。我们将使用JSON对响应进行格式化，客户端处理用JSON格式的响应要比XML数据容易。

值得讨论的是，使用JSON的局限是servlet只能用于JavaScript应用场合。我们清楚它用于JavaScript，而且该限制对我们来说影响不大。如果我们想要搜索servlet能更加广泛地重用，我们可以重新考虑此设计。

响应是一个由多个搜索结果组成的数组。每个搜索结果有三个字段：标题，URL和文本片段。以JSON规范表示一个由三条假想数据组成的响应，大概如下：

```
[
 { title: 'Title 1', url: 'http://url1/', snippet: 'Snippet 1' },
 { title: 'Title 2', url: 'http://url2/', snippet: 'Snippet 2' },
 { title: 'Title 3', url: 'http://url3/', snippet: 'Snippet 3' }
]
```

一组方括号表示数组，数组中的每个对象元素以花括号相分隔，而且标签/值对表示对象中的属性及其对应的值。有了这些内容以及简单搜索Java类，在代码清单12-7中显示搜索servlet的具体实现。

代码清单12-7　用于搜索的Servlet

```
package org.bibeault.aip.search.google;

import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;
import com.google.soap.search.*;

public class SimpleGoogleSearchServlet extends HttpServlet {

  public static final String
    KEY_SEARCH_TERM = "term";                    ❶ 期望的搜索条件

  protected void doGet(
    HttpServletRequest request,                  ❷ 获取搜索条件
    HttpServletResponse response)
        throws ServletException, IOException {
    String searchTerm = request.getParameter(KEY_SEARCH_TERM);
    SimpleGoogleSearch searcher =
      new SimpleGoogleSearch("aApewexQFHItVSrlTMDk2iglRbhB+6AR");
    try {
      GoogleSearchResultElement[] results =
        searcher.search(searchTerm);
      StringBuilder responseBody = new StringBuilder();
      responseBody.append('[');
      for (GoogleSearchResultElement result : results)
        appendResultAsJSON(responseBody, result);
      responseBody.append(']');
      response.setContentType("text/plain");
      response.setContentLength(responseBody.length());
      response.getWriter().print(responseBody.toString());
    } catch (GoogleSearchFault e) {
      e.printStackTrace();
      throw new ServletException("Search error: " + e, e);
    }
  }

  private void appendResultAsJSON(                ❸ 把响应按JSON规范格式化
    StringBuilder responseBody,
    GoogleSearchResultElement result) {
    responseBody
      .append('{')
      .append("title:'")
      .append(escapeQuotes(result.getTitle())).append("',")
      .append("snippet:'")
      .append(escapeQuotes(result.getSnippet())).append("',")
      .append("url:'").append(escapeQuotes(result.getURL()))
      .append("'")
      .append("},");
  }
```

```
    private String escapeQuotes(String text) {
      return text.replaceAll("'", "\'");
    }

}
```

我们实现的servlet只接受一个名为term的请求参数，它用于获取搜索条件❶。收到GET请求❷（POST也可以轻松支持）之后，term参数就取到了。（如果能添加一些错误检查就更好了。）

使用Google注册码创建SimpleGoogleSearch类的实例。一般来说，类似这种注册码之类的代码永远不应该以硬编码的形式处理。应该通过外部资源来提供——或许是一个属性文件，或者作为部署描述上下文参数配置。

使用该搜索引擎Java类实例，可以执行搜索并获取搜索结果。

要把响应格式化成JSON数据格式并发送到客户端，我们创建了一个StringBuilder类实例，在appendResultsAsJSON()函数❸的帮助下生成响应的主体。注意，另一个方法escapaQuotes()用于确保返回的文本结果中的单引号符合JSON语法。

我们把响应的MIME类型设置为普通文本，长度设为JSON响应主体的长度。最后，JSON数据以响应输出流返回到客户端。

在我们的Web应用的web.xml部署描述文件中，添加servlet及其映射关系，结果如下：

```
<servlet>
  <servlet-name>SimpleGoogleSearchServlet</servlet-name>
  <servlet-class>
    org.bibeault.aip.search.google.SimpleGoogleSearchServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>SimpleGoogleSearchServlet</servlet-name>
  <url-pattern>/search</url-pattern>
</servlet-mapping>
```

到此，我们可以开始编写页面来使用新实现的搜索servlet。

● *为搜索改造页面*

回到我们最初的设计意图，对文本片段添加搜索功能以便用户能够通过单击操作轻松搜索自己不熟悉的术语或概念。我们可以这样实现，这些文本片段以锚点标签出现在文本中，并使用JavaScript来发起Ajax搜索请求。不过，它将导致我们难以使用不同的样式来区分这种“自动术语搜索”和“真正的”超链接。另外，这样也将导致页面的代码变得略显冗长。

我们这儿采用一种更为取巧的方法，借用<abbr>标签——它是一个不太常用的HTML/XHTML标签，原本用于在HTML页面中标识出缩写。经过大量煞有介事的合理化论证，我们认定这样使用标签是正当的，因为术语就是搜索结果的“缩写”[①]。不过说实在的，如果劫持<abbr>标签让你不爽，你可以使用标签并应用一个样式类。这样也同样能工作，只是更啰嗦一点。

研究下列文本片段，它来自于采用<abbr>标签的页面。

① 作者在这里是半开玩笑的，这样使用abbr标签是不合理的。我个人建议使用标签。——译者注

```
<p>
  I find that the accumulation of
  <abbr onclick="searchFor(this);">stratocumulus</abbr> and
  absence of <abbr onclick="searchFor(this);">solar
  radiation</abbr> presages an increased rate of
  <abbr onclick="searchFor(this);">condensation</abbr>. We may
  expect vertical <abbr onclick="searchFor(this);">precipitation
  </abbr> of moisture at any time.
</p>
```

注意，我们已经把所有术语嵌入<abbr>标签，这些标签的onclick事件处理器调用JavaScript函数，该函数将<abbr>元素作为参数。这非常简单，但有点凌乱，不过为我们为搜索的每个术语添加相同onclick代码有些多余。

那么，通过编写JavaScript类，让它来自动处理原本由页面实现的所有内容。我们在本书已经看到过大量的JavaScript类，第3章已经深入地介绍过如何创建类。因此，在代码清单12-8中展现其代码之后，我们将没用的代码省略只保留本类的重要部分。

代码清单12-8　SearchInstrumenter类

```
SearchInstrumenter = Class.create();

SearchInstrumenter.prototype = {

  DEFAULT_ELEMENT_TYPE: 'abbr',
  DEFAULT_RESULTS_CONTAINER: 'resultsContainer',
  SEARCH_URL: '/aip.chap12/search',

  initialize: function(options) {
    this.options = Object.extend(
      {
        elementType: this.DEFAULT_ELEMENT_TYPE,
        resultsContainer: this.DEFAULT_RESULTS_CONTAINER
      },
      options
    );
    var self = this;
    $$ (this.options.elementType)
      .each(
        function(element) {
          element.onclick = function() {
            self.doSearch(element.innerHTML);
          }
        }
      );
  },

  doSearch: function(term) {
    new Ajax.Request(
      this.SEARCH_URL,
      {
        method: 'get',
        parameters: {term: term},
        onSuccess: this.showResults.bind(this),
```

❶ 为每个特定类型的标签构造实例

❷ 提供默认的参数

❸ 为每个元素指派onclick事件处理器

❹ 通过点击按钮执行搜索请求

```
        onFailure: this.showError.bind(this)
      }
    );
    $(this.options.resultsContainer).innerHTML =
      'Searching for ' + term + '...';
  },

  showResults: function(request) {
    var jsonResponse = request.responseText;
    eval ('(results='+jsonResponse+')');
    $('resultsContainer').innerHTML = '';
    results.each(
      function(result) {
        $('resultsContainer').innerHTML +=
          '<p>Title: <b>' + result.title + '</b><br/>' +
          'Summary: ' + result.summary + '<br/>' +
          'URL: ' + result.url +'</p>';
      }
    );
  },

  showError: function(request) {
    $(this.options.resultsContainer).innerHTML =
      request.responseText;
  }
}
```

❺ 获取并显示响应的结果

❻ 显示错误消息

该类的构造函数❶将为页面中的每个特定类型的标签创建实例。它接受一个参数options，该参数允许我们覆盖类提供的默认值。该类❷的options属性是elementType和resultsContainer，它们分别拥有默认值abbr和resultsContainer。

一旦确定要创建的元素类型，该类型的所有实例就能够找到并为每个实例❸的onclick处理器指派方法。这就可以避免我们通过手工为每个可搜索的元素添加事件处理器，而在此之前显示的示例文本中要求我们这么做。

每个已定位元素的onclick事件处理器指派了本类声明的doSearch()方法。注意，该函数使用闭包保证类实例和元素引用在事件触发时可用。（如果我说的这些听起来冗长乏味，请参考第3章关于闭包的内容。）

doSearch()方法用作按钮onclick处理器❹，使用它向搜索servlet发起Ajax请求，我们在前一节的解决方案中已经介绍过这部分内容。值得特别注意的是使用对象Hash传递请求参数，并且将成功和失败处理器设置为本类的其他方法。

该类的onSuccess事件处理器是showResults()❺，它从XHR实例获取响应文本。因为你将回忆起来，搜索servlet返回包含搜索结果JSON对象。我们对结果执行eval操作并将结果进行迭代操作，将包含数据的HTML页面格式化并将其设置到结果容器的内容。

该类的最后部分是错误处理函数showError()❻。它从请求实例获取错误消息并将其设置到结果容器上，以便大家可以看到。

- 测试搜索组件

最后，我们已经准备好编写一个页面，它使用我们创建好的搜索Java类。可以在可下载源代码的本章部分找到该页面，如代码清单12-9所示。这个普通页面将搜索结果放到一个文本区域中，如图12-6所示。

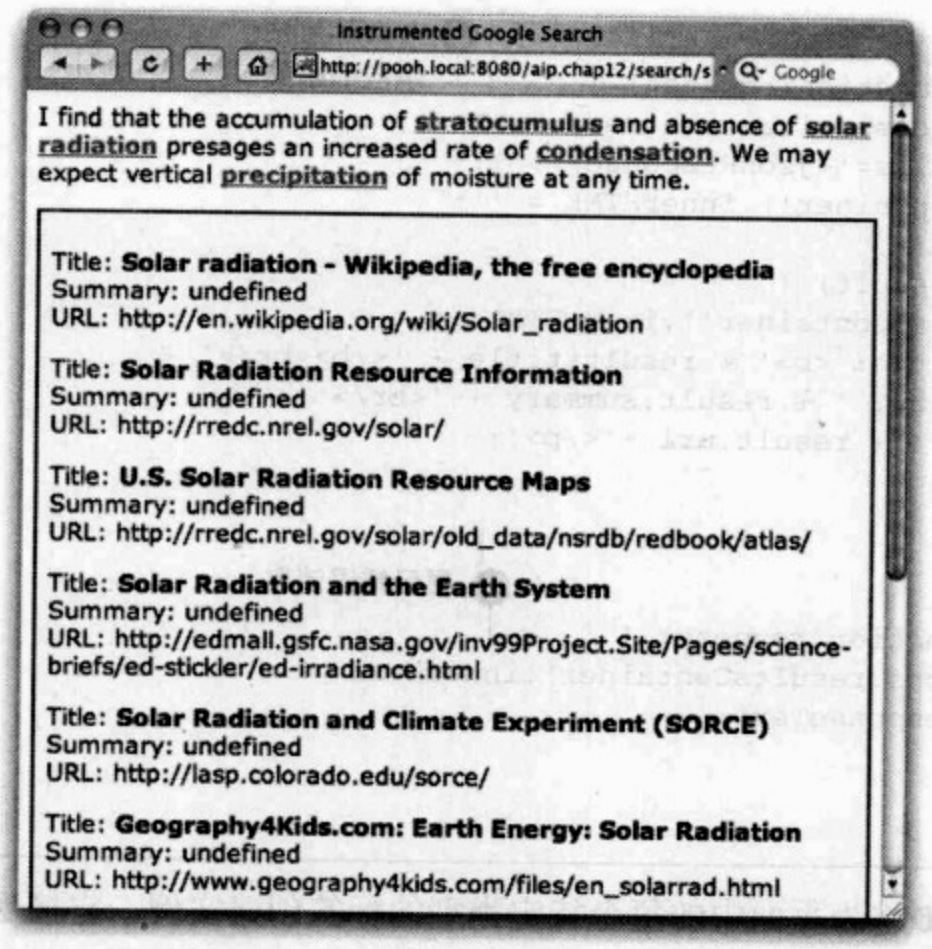


图12-6　搜索“solar radiation”的结果

代码清单12-9　什么是stratocumulus

```
<html>
  <head>
    <title>Instrumented Google Search</title>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="SearchInstrumenter.js">
    </script>
    <script type="text/javascript">
      window.onload = function () {
        new SearchInstrumenter();
      };
    </script>
    <style type="text/css">
      abbr {
        color: green;
        font-weight: bold;
        text-decoration: underline;
        cursor: pointer;
      }
      #resultsContainer {
        border: 2px ridge maroon;
```

```
        background-color: #ffffcc;
        padding: 8px;
      }
    </style>
  </head>

  <body>
    <div>
      <p>
        I find that the accumulation of <abbr>stratocumulus</abbr>
        and absence of <abbr>solar radiation</abbr> presages an
        increased rate of <abbr>condensation</abbr>. We may expect
        vertical <abbr>precipitation</abbr> of moisture at any
        time.
      </p>
    </div>
    <div id="resultsContainer"></div>
  </body>
</html>
```

这个页面简单而带有欺骗性，它用到了我们在本节介绍过的所有技术。在该页面上，我们使用<abbr>标签标识出了一些术语，该页面允许访问者通过简单点击实现自动执行搜索。

使用少量CSS和JavaScript代码，我们可以实现更具创造性的功能。例如，我们可以将搜索结果在原术语附近以浮动 方式显示。或者我们可以将结果元素扩展以便嵌入文本，直至不再显示。具体实现什么样取决于你的想象力。

3. 讨论

通过本解决方案，我们实现了发起任意异步搜索的功能。我们设计了一种机制从HTML页面直接获取搜索项，并且我们研究了以不同于XML的格式返回动态数据给页面的方法。

你不赞同使用JSON而不是XML返回数据，因为这可能限制我们的搜索服务只能应用在实现JSON的环境。不过，对使用servlet的应用场合，HTML页面之外的其他部分并不一定使用后台搜索到的数据。况且，如果需要，我们还可以编写一个JSON-to-XML适配器。

和本章中的其他示例一样，本例可以应用更多的错误检查和恢复机制。事实上，在测试本例的过程中，我们发现相当数量的Google搜索请求遇到了“服务暂时不可用”的错误。你该如何改进搜索组件以处理该问题呢？

哦，还有搜索结果中包含的URL？它们大概是一些超链接，你不这么认为吗？

12.3 Flickr 图片分享

Flickr网站专注于图片分享。

无论你是铁杆的气象摄影师、摄影爱好者还是溺爱孩子的老人，你所拍摄的大量气势恢宏的超级单体雷暴、得克萨斯州春季里的野花，还有孩子们极其可爱的照片，全部都可以拿来放到网站分享。

Flickr（www.flickr.com）是互联网上发展最迅速的图片分享网站之一。让我们感到高兴的是它为开发人员提供了丰富的格式多样的API。这些API支持Java、.NET、Delphi，还有Ruby和Perl，

还支持基于SOAP、XML-RPC和REST的访问请求。和以前一样，我们将关注REST API，因为它是最适合用于支持Ajax的页面的技术之一。

和其他Web API一样，使用Flickr接口需要注册并获取一个API识别码，它用于识别每个请求的发起者。该识别码由随机生成的字母字符串和数字组成，访问www.flickr.com/services/api/misc.api.keys.html页面可以轻松获取。

一旦获取到识别码，我们就可以使用它发起到Flickr REST API的请求了。所有访问Flickr的REST请求都使用相同的URL——www.flickr.com/services/rest/。

每个请求必须提供两个查询参数：api-key和method，前者向系统提供表示我们身份的识别码，后者指明我们希望执行的服务函数。不要把Flickr方法和类方法混淆；Flickr方法只是传递到Flickr服务API的文本字符串，用以标识要执行的操作。取决于要执行的方法，它们可能有多个参数——有些是必须的，有些是可选的。

Flickr区分公共和私有图片集的概念。两种图片都可以通过Web API访问，但毫无疑问，访问私有图片集需要采用更为复杂的认证协议。对于我们当前的实现意图来说，我们将专注公共图片集访问以避免这些不必要的麻烦。

12.3.1　Flickr 用户内部标识

本节我们将设计一个页面，它将显示某个朋友的公开图片缩略图。点击每个缩略图，其尺寸将变大。

你或许会问："为什么这么麻烦呢？"毕竟我们可以使用硬编码的图片标记指向其图片URL，难道不是吗？

当然可以。Flickr不会关心你是否在自己的页面中嵌入以其网站作为宿主的图片引用。但是，在静态页面中采用硬编码的方式引用外部图片资源，这些资源有可能在我们不知情的状态下发生改变，原有的引用不就失效了吗？难道我们需要经常检查这些引用是否存在并在其改变时作相应调整？多麻烦啊！

相反，通过依赖Flickr提供的API，我们可以实时获取当前的图片资源，并且利用我们的Ajax和DHTML技能，动态创建页面，它们永远都不会过期。

1．问题

我们想构建一个页面，它显示某好友公开图片集的活动缩略图。我们知道这位好友在Flickr的用户名，不过只要我们查看用于返回好友公开图片列表的方法，我们就会发现一个问题。该方法不需要用户名，而需要他（她）的NSID——在多数Flickr公开方法中区别目标用户的内部标识符。

不要把这个标识符和API识别码混淆。后者用于区别谁在发起请求，前者标识请求的目标客户。

因此，在介绍创建好友图片缩略图页面之前，我们需要快速掌握如何通过用户名获取其NSID。

2．解决方案

图12-7显示了本方案查询特定用户名的最终结果。

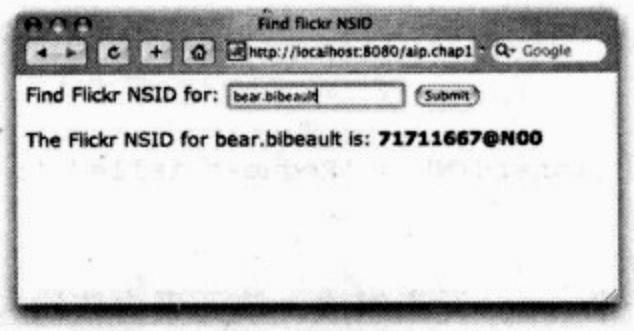


图12-7 查找NSID

下列Flickr方法名用于获取指定用户名的NSID：

```
flickr.people.findByUsername
```

所有的Flickr方法名以字符串flickr开头，然后跟随一个类别名称（people、contacts、groups或其他），最后请求特定的服务。

由于要根据指定的用户名获取其NSID，我们首先在页面中创建一个简单的表单以容纳用户名文本字符串，并触发和获取NSID请求：

```
<form name="queryForm" onsubmit="findNSID();return false;">
  Find Flickr NSID for:
  <input type="text" name="username"/>
  <input type="submit"/>
</form>
```

当用户输入一个用户名并点击提交按钮时，findNSID()方法被调用：

```
function findNSID() {
  new Ajax.Request(
    '/aip.chap12/proxy',
    {
      method: 'get',
      parameters: {
        '.serviceUrl.':
          'http://www.flickr.com/services/rest/',
        api_key: '78eaa5287f3f0b37dfba77ef40c7df03',
        method: 'flickr.people.findByUsername',
        username: $F('username')
      },
      onSuccess: onResultObtained
    }
  );
  $('resultContainer').innerHTML = '';
}
```

此刻，你应该会发现该函数的结构你很熟悉。要发起Ajax请求，我们需要为服务端代理中转对象指定我们的Flickr API识别码、要调用的Flickr方法（不要和Prototype提供的HTTP方法混淆），还有要查找的用户名。

该请求的onSuccess事件处理器是：

```
function onResultObtained(request) {
  var xml = request.responseXML;
  if (xml.getElementsByTagName('rsp')[0].getAttribute('stat')
```

```
            == 'ok') {
        showResults(xml);
      }
      else {
        $('resultContainer').innerHTML = 'Request failed.';
      }
    }
```

所有请求的响应文档由一个<rsp>元素组成，该元素内依次包含子元素信息。<rsp>元素有一个名为stat的属性，根据请求的成功与否，它的值被设置为ok或fail。

一个失败的请求大概是这样的：

```
<rsp stat="fail">
  <err code="112" msg="Method &quot;&quot; not found" />
</rsp>
```

我们指定了事件处理器检测该属性以确定请求成功与否。如果检测到失败，名为resultContainer的 元素将设置简单的错误消息（使用寥寥数行JavaScript代码，我们就可以从响应文档中轻易获取特定错误信息）。如果请求成功，请求的响应文档将传入名为showResults()函数并进行处理。

在前一个示例中，为了显示请求结果，我们使用非正规的innerHTML属性动态设置事先定义的元素内容。虽然这种方法能够实现显示简单文本或极其简单的标记，但是它的扩展性不好。不过，要生成标记就变得更加复杂或啰嗦了，创建文本字符串格式的标记变得相当麻烦和笨重——更不要提潜在的维护性难题。

因此，在showResults()函数中，我们将使用一种新方式来动态创建请求结果元素的内容：DOM操作。

我们使用DOM API查询传入函数的XML文档内容，但我们还可以使用相同的DOM API函数来操作我们所需的HTML文档的DOM树。

对Flickr服务的响应，findByUsername方法能处理的正确格式如下：

```
<rsp stat="ok">
  <user id="71711667@N00" nsid="71711667@N00">
    <username>bear.bibeault</username>
  </user>
</rsp>
```

这样的话，我们可以这样实现showResults()函数：

```
function showResults(xml) {
  var nsid = xml.getElementsByTagName('user')[0]
    .getAttribute('nsid');
  var p = document.createElement('p');
  p.appendChild(document.createTextNode('The Flickr NSID for ' +
                document.queryForm.username.value + ' is: '));
  var span = document.createElement('span');
  span.style.fontWeight = 'bold';
  span.appendChild(document.createTextNode(nsid));
  p.appendChild(span);
  $('resultContainer').appendChild(p);
}
```

大概你还记得，我们把请求响应的XML文档传入该函数，并且我们知道之前确实已经检查过请求是否已成功。

在找到用户名对应的NSID并将其记录下来以备后用之后，我们已经准备开始在文档中创建新HTML元素。首先，创建一个新段落元素（<p>）。我们想在该元素上放置某些文本，因此我们创建一些文本并将其作为该段落元素的子元素。

NSID以粗体字出现，因此我们将其嵌入到一个采用粗体类型的元素中。粗体元素作为段落元素的子元素，并且最后我们把该段落元素以子元素添加到resultsContainer元素。

3. 讨论

在这个极为简单的示例中，我们拥有了从Flickr获取任意指定用户名的NSID的功能。该NSID对于发起到其他任何Flickr API的调用是必须的，用于识别目标客户。

对每一个用户名，其对应的NSID是固定不变的，一旦获得就可以记录下来并直接使用。我们将在下个解决方案中看到这种用法。

我们还介绍了一种新的方式实现将动态结果显示到结果元素，即通过DOM操作，而不是创建字符串格式的标签并通过innerHTML属性设置到结果元素。

和innerHTML属性实现机制相比，尽管这种技巧最初看似有些过于复杂或者繁琐，但最终的好处是它便于维护并且更加健壮。超越innerHTML方案的一个优势是我们不会理会引号和其他符号问题。使用文本字符串创建标签，这些问题的出现将不可避免。

12.3.2 Flickr 图片和缩略图

由于我们已经拥有目标客户的NSID，实现最有趣的部分因此也已准备就绪：从Flickr获取图片。

1. 问题

我们要创建一个页面，它显示某个朋友公开的图片集的缩略图。由于已经有了他在Flickr的NSID，接下来没什么难点了。

我们希望单击某张缩略图后会显示它的大尺寸版本。并且，我们希望在整个实现过程中除了目标用户的NSID之外不要有其他任何硬编码出现。

2. 解决方案

获取指定NSID的公开图片集列表的Flickr方法名称：

```
flickr.people.findPublicPhotos
```

此方法除了必须的方法名称和API识别码参数之外，该方法还需一个user_id参数，它用于提供目标客户的NSID。其他的可选参数影响返回数据的类型和数量，但目前我们将保持其简单性。

该方法的成功响应示例如下：

```
<rsp stat="ok">
  <photos page="1" pages="1" perpage="100" total="3">
    <photo id="128217127" owner="71711667@N00" secret="09e814e0b0"
```

```
        server="51" title="DSC01660" ispublic="1" isfriend="0"
        isfamily="0" />
      <photo id="128217125" owner="71711667@N00" secret="ef6ad6886d"
        server="40" title="DSC01476" ispublic="1" isfriend="0"
        isfamily="0" />
      <photo id="128216010" owner="71711667@N00" secret="ff13ad56b9"
        server="46" title="DSC01118" ispublic="1" isfriend="0"
        isfamily="0" />
    </photos>
</rsp>
```

无论你是否相信，上述内容给出了我们创建到Flickr服务器上的图片的URL所需的全部信息。每张来自Flickr的图片的URL其格式均如下：

```
http://static.flickr.com/{server}/{id}_{secret}{suffix}.jpg
```

每张图片的server，secret和id值直接来自于响应XML文档中的<photo>元素属性，其中suffix后缀指明了图片的尺寸，如下所示：

- _s: 75×75像素方形（75-by-75-pixel square）；
- _t: 最长100像素（100 pixels on longest side）；
- _m: 最长240像素（240 pixels on longest side）；
- (none): 最长500像素（500 pixels on longest side）；
- _b: 最长1024像素（1024 pixels on longest side）；
- _o: 原始尺寸（original image）。

如果我们希望为上述响应的第一个图片创建其缩略图URL，替换server、id、secret和相应的suffix后缀，将生成一个URL：

```
http://static.flickr.com/51/128217127_09e814e0b0_t.jpg
```

使用该URL作为图片元素的来源，为该图片创建缩略图。通过简单地改变URL的后缀，例如改为_b，我们能获得同一图片的更大尺寸的URL。

利用这些新学到的知识，我们来实现页面。首先，我们希望在页面显示后自动加载图片缩略图。因此在页面首部的<script>元素中，我们的实现如下：

```
window.onload = function() {
  new Ajax.Request(
    '/aip.chap12/proxy',
    {
      method: 'get',
      parameters: {
        '.serviceUrl.':
          'http://www.flickr.com/services/rest/',
        api_key: '78eaa5287f3f0b37dfba77ef40c7df03',
        method: 'flickr.people.getPublicPhotos',
        user_id: '71711667@N00'
      },
    onSuccess: onInfoObtained
    }
  );
}
```

该事件处理器发起Ajax请求希望使用findPublicPhotos方法。如果成功，处理该请求的函数：

```
function onInfoObtained(request) {
  var xml = request.responseXML;
  if (xml.getElementsByTagName('rsp')[0].
          getAttribute('stat') == 'ok') {
    showThumbnails(xml);
  }
}
```

上述方法和我们之前介绍的onSuccess事件处理器类似，它检查Flickr方法的状态并且在一切正常的情况下调用showThumbnails()函数，参数响应XML文档。注意，可能有一些未知错误发生，这时用户看到的是空白页面。显然，代码还有改进的余地。

showThumbnails()方法负责实现非常有趣的缩略图功能，代码如下：

```
function showThumbnails(xml) {
  var photos = $A(xml.getElementsByTagName('photo'));
  photos.each(
    function(photo) {
      var baseUrl = 'http://static.flickr.com/' +
                    photo.getAttribute('server') + '/' +
                    photo.getAttribute('id') + '_' +
                    photo.getAttribute('secret');
      var thumbUrl = baseUrl + '_t.jpg';
      var photoUrl = baseUrl + '.jpg';
      var thumb = document.createElement('img');
      thumb.src = thumbUrl;
      thumb.style.cursor = 'pointer';
      thumb.onclick = showPhoto;
      thumb.photoUrl = photoUrl;
      $('thumbnailsContainer').appendChild(thumb);
    }
  );
}
```

该函数调用时传入的参数是成功XML响应文档，其中包含从Flickr获取的<photo>元素列表。在循环处理每个<photo>元素时，利用<photo>元素提供的信息创建了一个base URL（不包含图片尺寸后缀）。然后，通过在base URL后面添加_t后缀和空后缀，分别创建缩略图和实际大小的图片。

然后使用DOM API创建用于显示缩略图的<img>元素。当鼠标在图片元素上滑过时，鼠标图标被设置为pointer（微型手指针）。这样一来，用户就会明白缩略图可以点击，并且该图片的onclick处理器设置为showPhoto()方法。

在获取图片之后，我们在图片元素上直接动态添加名称为photoUrl的属性，它用于显示中型尺寸的图片。记住，JavaScript允许我们为任意的对象动态地添加自己的属性。即使是HTML DOM树中的对象也可以采用该技巧，非常方便。

最后，将新创建的<img>元素添加到 容器作为其子元素，该容器元素的ID是：thumbnailsContainer。

事件先后触发的结果是缩略图在页面显示后就加载了，并且每个缩略图都支持并等待用户点击以显示更大的图片。点击操作在showPhoto()方法中完成，我们为每个缩略图元素的onclick事件处理器声明了该方法：

```
function showPhoto() {
  var photo = $('photoElement');
  if (photo == null) {
    photo = document.createElement('img');
    photo.id = 'photoElement';
    $('photoContainer').appendChild(photo);
  }
  photo.src = this.photoUrl;
}
```

当页面第一次加载时，我们不希望显示不完整的图片，所以最初我们甚至没有为图片创建<img>元素。相反，我们在元素上首次检测它不存在时再临时创建<img>元素。因此，在showPhoto()方法中，我们试图找到ID为photoElement的元素，如果它不存在，就创建它。

一旦我们拥有该元素的引用，不管是否是新创建的，我们将<img>元素的src属性设置为photoUrl，该图片（图12-8）URL是我们在showThumbnails()方法中为缩略图元素生成的。之所以能够在事件处理器中使用上下文对象（通过this使用对象引用）是因为触发事件的对象实际就是缩略图元素。

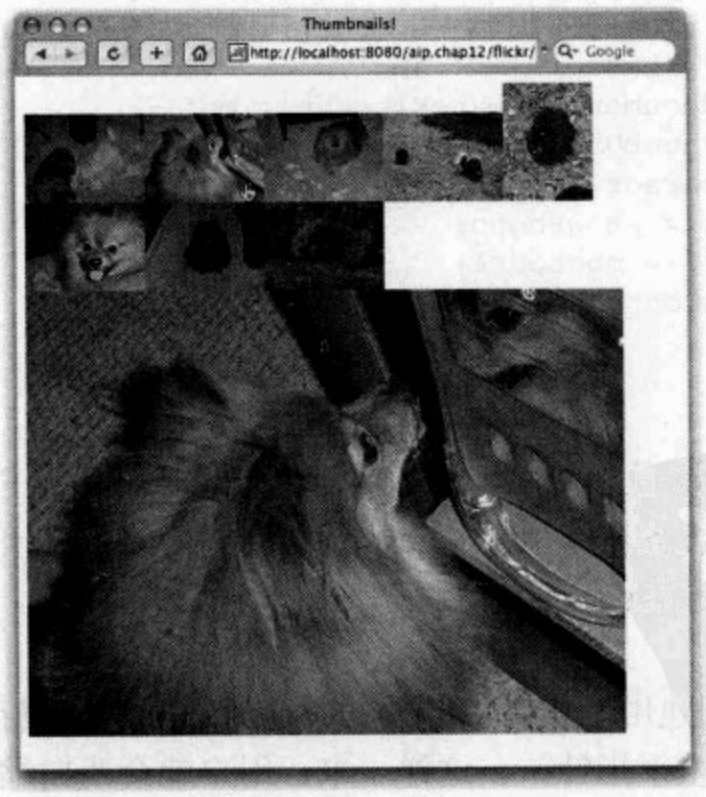


图12-8　屏幕上的小狗的主人是谁

最后，我们准备好编写HTML页面的主体部分。但由于我们将大量内容动态生成，页面极其简单：

```
<body>
  <div id="thumbnailsContainer"></div>
  <div id="photoContainer"></div>
</body>
```

页面（在页面加载后并且单击某个缩略图）的显示结果如图12-8所示。

3. 讨论

本节我们学习了如何使用Flickr REST API动态访问某用户的公开图片及缩略图——访问我们自己的或其他已知用户名的用户的图片。

显然，我们所创建的页面可以使用某些样式，还有在初始阶段希望添加但并未实现的可用性。不过，可以在很短的时间内，使用合适的CSS及少量JavaScript代码就能改进页面。检查缩略图元素是否存在的代码也需要完善。

本解决方案中，我们将希望访问的用户NSID硬编码。对特定的用户来说，其NSID从不更改，因此对那些我们已知的用户及其NSID而言，这么实现就很安全。如果我们不希望预先查找那些NSID值，可以将这个示例和前一个示例整合成一个方案：提供Ajax请求，按用户名查找NSID，然后在该请求的成功处理器中提交第二个请求，从而获取已知NSID的公共图片列表。

可以（并且应该能够）增强该页面，利用Flichr的`getPublicPhotos`方法提供的可选参数实现大型的图片列表分页显示。

我们还领略了一些JavaScript的小技巧：在HTML页面的DOM元素上增加属性以便稍后引用。对使用JavaScript操纵DOM元素和创建动态页面来说，这是非常有利的工具。

12.4 稍等！据说，还有很多……

本章介绍了Yahoo！、Google和Flickr提供的Web服务，并花了一些精力通过Ajax访问这些服务以实现动态Web页面。但是，我们看到的这些接口只是冰山一角！

互联网上公开提供的API少则几十个，多则数以百计。

12.4.1 Amazon服务

Amazon提供了许多实用的服务，使用它，人们可以创建自己的Web应用、网上商店或由Amazon及其伙伴提供的有大量商品的门户。和其他服务一样，Amazon提供了REST API，这使得在支持Ajax的应用中可以方便地使用该API。

你可以在www.amazon.com/gp/aws/landing.html上找到关于注册和使用接口的详细内容。

12.4.2 eBay服务

想创建下一个基于eBay的卓越应用吗？

eBay为eBay拍卖活动提供了非常丰富而完整的API，包括Java、PHP、SOAP，当然还有一个REST API。

REST API看起来有点限制，它只能搜索eBay网站提供的列表，至少在本书写作时是如此。不过如果我们希望开发高级的eBay应用，也可以使用其他的API来为支持Ajax的页面创建服务端代理。

eBay API开发人员可以在http://developer.ebay.com/找到编程方面的有关信息。

12.4.3 MapQuest

不太喜欢使用Yahoo！或Google的地图服务？请研究一下MapQuest OpenAPI：www.mapquest.

com/features/main.adp?page=developer_tools_oapi。

12.4.4　NOAA/国家气象服务

对天气感兴趣吗？NOAA提供了基于SOAP协议的气象预报、天气和预警服务API。请查看：www.nws.noaa.gov/forecasts/xml/。

12.4.5　更多 Web 服务接口

在Web上搜索一下，你会找到非常多的Web服务API，其中的一些接口，你未必了解怎样使用它。ProgrammableWeb网站的列表www.programmableweb.com/apilist是一个不错的起点。

祝大家研究得开心！

12.5　总结

本章我们进展神速。我们考察了开发人员可用的多个Web服务，同时还研究了许多技巧，它们几乎正是处理所有可用服务所必需的。

我们学习了如何使用JavaScript API和如何将服务端API整合到Web应用中。也许最重要的是，我们学会了怎样绕开Ajax安全沙箱，以便向第三方服务器上的Web服务发起请求。

尽管掌握了这些知识，但事实上并不意味着我们利用Web服务的工作结束了，有很多网站大方地为我们提供了免费的Web服务。

我们并不仅限于使用某一种Web服务。在下一章中，我们将介绍如何将多个不同的服务提供者聚合到单个Web应用中。

第13章 使用Ajax进行混搭

本章内容

- “mashup”到底是什么？
- 决定数据格式
- 解读从服务器载入的XML数据
- 开放API大合唱

“搅和一下（mashing it up）”并不是说我们要到厨房去拌土豆泥，也不是说我们要到五光十色的舞池里参加潮流热舞。对于Web应用来说，一个**mashup**[①]就是一个融合了来自多个源头的内容的网页。“混搭”也许是一个很新的术语，但并不是一个很新的概念。然而，随着Ajax的出现，加上越来越多可用的开放API（见第12章），使现在比以往更容易创建混搭式应用。

在本章中，我们会利用在第3章和第4章研究过的并且在全书中也经常使用的Prototype脚本库，结合我们在第12章中获得的开放API的知识，使用Yahoo! Maps和Flickr Photo Services 的开放API来创建一个我们自己的混搭式应用。

如果你还没有读过第3章和第12章，那最好在继续学习本章之前先回到前面读一下这两章。特别是那些讨论Prototype、Yahoo! Maps和Flickr的小节，你应该好好阅读一下。

13.1 Trip-o-matic 应用简介

旅行的乐趣有一半，有时甚至大半，是因为将来可以吹嘘一番。通常，我们一边吹嘘还会一边展示满满一大相册的照片，这些照片（取决于拿着镜头的那位的水准）好得可以引人入胜比得上专业摄影展，差得也可能索然无味连焦距都没调准。

无论是哪一种情形，互联网的威力让任何人都有能力把他们的照片推向更大的观众群体，而不仅限于向家人和朋友炫耀，当你拉出幻灯机和屏幕的时候，他们别无选择只好坐在那里。

13.1.1 应用的目的

有一件事情可以让旅行照片不至于乏善可陈，那就是给照片加上**情境**。为了帮助传达情境，我们会编写一个Web应用，就叫它**Trip-o-matic**[②]吧。

---

① 译为“混搭”或“混搭式应用”。——译者注

② trip意为旅行，-o-matic是一个常见后缀，表示这是一种自动完成某种功能的产品或服务。——译者注

这里你可能会说："把照片存放到Flickr账户里不就好了吗？"

没错，我们是可以这样做，但是要传达我们的情境，这还不够好。当然，我们可以在照片后面附上注释，告诉访问者这个照片是在哪里拍摄的，比方说"这是在得州首府奥斯汀拍的照片"，"这是在俄克拉荷马城拍的"，"这是在堪萨斯州府托皮卡拍的"，"这是在内布拉斯加州府林肯市拍的"，以及"哦瞧！这是总统山！"

但是我们要做得更好：直接显示地图！

13.1.2　应用概览和需求

Trip-o-matic应用虽然相当简单，但足以帮我们为更复杂的混搭式应用打下基础。这个应用程序就只包含一个页面，其内容完全由数据文件驱动，而数据文件包含着我们已完成的某次旅行的相关信息。因为不管显示哪次旅行都是用的同一个页面，所以不能把特定于某次旅行的信息写死在页面或者应用程序的代码中。所有的旅行信息都会存放在数据文件中，由页面读取。

这可能会让你想起那些采用JSP或PHP之类的服务器端模板技术的应用程序。但是我们的应用程序是完全由客户端代码构成的，唯一的例外是我们需要使用在12.1.2节中所开发的提供跨服务器代理服务的Servlet。这个代理会允许我们避开Ajax安全沙箱的限制，向Yahoo! 和Flickr Photo Services站点发起跨服务器请求。

要"喂"给应用程序的数据文件包含了我们对一次旅行需要了解的所有信息。旅行的名称、旅程描述，还有用于访问我们的Yahoo!和Flickr账户的API Key的相关信息，这些也是数据的一部分。

旅行信息也会包含一个列表，列出旅行路线上的景点（**points of interest**）[①]。对于每个景点，我们都会有一个简短的名字、一段较长的描述、景点的位置以及如何取回与景点相关联的Flickr照片的信息。

当页面显示时，应用程序会读取旅行信息并列出所有的景点名字。如果访问者点击了这个列表中的某一项，就会显示出这个景点区域的地图。点击地图会显示与景点相关联的照片的缩略图，点击缩略图则会显示对应的原始大小的照片。

没有比增加交互性更有助于提高趣味的了！

与第12章中那些简短的示例不同，这次我们会为应用程序页面添加一定数量的样式来达到基本水准的优使性。不过正像那些示例一样，用户界面专家的建议无疑会让应用受益匪浅，最好也可以请优秀的视觉设计师出手相助。

同时，为了简短起见（你会感谢我的），我们没有考虑错误检查。如果你对这个示例真的很有兴趣，那你要做的第一件事情就应该是补上事实上不存在的错误处理。

总而言之，这个示例足以作为"概念验证（proof of concept）"——要让这样一个应用上路运行（一语双关[②]），我们需要用到许多技术，而这个示例包含了所有这些技术！

13.2　Trip-o-matic 的数据文件

在开始写代码之前，我们必须先设计出要"喂"给页面的数据文件。数据文件存放于服务器，

① POI（point of interest）常指名胜古迹，但也可以泛指对用户有用的地点，例如餐馆、酒店等。——译者注

② 原文"to get such an application on the road"，也可以表示"要在旅行中得到这样一个应用"。——译者注

当页面加载时，会通过Ajax请求传回到客户端，然后客户端代码会解读[①]文件并创建JavaScript结构体来表达其中所包含的信息。

至于加载哪一个数据文件，可以通过URL上的查询参数来告知应用程序。

13.2.1 我们应该采用什么格式

正如我们在前面几章所看到的，Ajax请求的响应可以采用好几种不同的形式，纯文本、HTML片段、JSON记录或XML文档。那么对于我们的旅行数据来说，哪种格式最合适呢？

旅行数据显然是一组结构化信息，因此首先可以排除纯文本和HTML，前者不能表达数据的结构，后者则是显示格式而不是数据格式[②]。这样，我们还剩下JSON和XML可以选择。

JSON是一种很不错的记法，因为对于客户程序来说它很容易解读，只要调用一下`eval()`函数就可以把JSON响应转换成对应的JavaScript结构体。JSON也很容易用程序生成。但是像我们的旅行数据文件这样的需要手工编码大型数据集的情形，JSON就未必是最佳选择了。

JSON记法很紧凑，使用一些简单的字符进行定界，比如用方括号和花括号来表示数组和结构体。嵌套的JSON数据很快就变成天书，看上去就像充满线路噪声的数据流（或者是你养的蜥蜴在键盘上散步）。这样的数据不仅很难以目力解析，而且在修订和追加的时候，也很容易在无意中产生错误。

另一方面，要在客户端JavaScript代码中解读XML，就需要做更多工作，但是XML很适合手工编码，因为有更详尽的标记，人们在检视文档中的数据结构时就较为轻松，这使得数据易于创建、维护和修订。

这样，我们所面对的选择就是：

- JSON易于解读，但是不易手工处理。
- XML更难解读，但是易于手工处理。

解读数据的代码我们只会写一遍，而旅行数据文件可能要写好多个，所以为了让自己轻松一点，我们选择XML，它简化了执行次数更多的任务。

这倒不是为了偷懒，只是为了做得更聪明！

此外，我们构建应用的方式相当灵活，如果我们的选择不太明智，大可收回这一决定[③]。

13.2.2 旅行数据格式

一个旅行XML文档会包含一次旅行的相关信息。对于每次旅行，我们需要的标量[④]数据包括下列内容：

- 旅行的题目。
- Flickr的API key（参见12.3节）。

---

① 本章中所说的解读（digest）是指将数据流转换为编程语言可操作的数据对象的过程。通常在实现上是“解析并读取”，故译作“解读”，也与汉语中“解读”的本意暗合。——译者注

② HTML的本意其实也并非一种显示格式，但即便如此，其语义也并不适合表示我们的旅行数据。——译者注

③ 在下一节中会具体讨论这一点。——译者注

④ 所谓标量是指只能存储单一值的变量或数据字段。通常我们认为数字、字符串等是标量，而数组、结构体等不是标量。——译者注

- 旅行照片所属Flickr账号的NSID（参见12.3.1节）。
- 景点列表部分的标题。
- 对旅行的描述。

我们的XML文档格式的根元素是<trip>元素，元素上的属性指定了前面的四个标量数据。由于描述数据可能会很长并且包含特殊字符，甚至干脆就是HTML标记，所以我们会把它放在独立的子元素<description>里。如果需要，<description>元素中的文本可以包含CDATA部分，这样就可以直接嵌入HTML标记而不会导致XML解析问题。

一个典型的<trip>元素和它的<description>元素可能是这样的：

```
<trip title="The Trip Title"
      flickrNSID="97545223@N00"
      flickrKey="78eca5287f3f05397dfba77ef40c7df53"
      poiTitle="Where we went">
  <description>
    <![CDATA[
      This is a descriptive comment for the trip complete with
      <b>some HTML markup</b>.
    ]]>
  </description>
    ... <!-- other child nodes go here -->
</trip>
```

除了<description>元素外，<trip>元素还包含了一组子元素，每一个都定义了旅程中的一个景点。每个景点上的信息（在<poi>元素中）包括：

- 景点简短的名字（用于页面上的景点列表）。
- 景点所处纬度。
- 景点所处经度。
- 景点照片所属Flickr图片集的ID。
- 对景点的描述或者附注。

除了景点的描述信息以外，其他部分都由<poi>元素上的属性指定。因为景点的描述信息也可能包含HTML标记，所以我们用<poi>元素内的文本和CDATA部分来表示景点的描述信息。

一个典型的<poi>元素可能是这样的

```
<poi name="Austin, TX"
     latitude="30.266748"
     longitude="-97.74176"
     photoSetId="1151001">
  <![CDATA[
    We started in Austin, TX on May 23, 2006. It was a
    <i>gorgeous</i> and sunny day. Little did we know what
    was in store for us...
  ]]>
</poi>
```

如果没有用GPS(全球定位系统)记录下旅程中每个景点的经度和纬度，可以使用我们在12.1.3节中研究过的Yahoo! Geocoding API将地点字符串转换为坐标。

下一节，让我们看看如何设置Flickr照片集，这样Trip-o-matic应用程序就可以通过photo-

SetId属性读取相应的照片集ID。

13.2.3 设置 Flickr 照片集

Flickr照片服务在短时间内不可思议地流行开来，其中一个原因就是它用起来真的很方便。注册账号简单快速（如果你已经有一个Yahoo!账号，那就直接用Yahoo!账号吧），然后你可以立马开始使用Web界面上传照片。如果你想再方便一些，可以下载他们的本地上传工具（让你很容易就能上传一批照片），Windows上和Mac OS X上都可用。也有集成到iPhoto或Windows资源管理器的插件，能让上传照片变得更简单。甚至还能通过Email来上传照片。实在是太方便了！

我们在12.3节中所完成的示例会取得已上传到目标账户中的所有公开照片的信息，显然这不是我们在这里想要的。我们并不想被无数影像所淹没，而只想得到已上传照片的一个子集：我们在某一个景点所拍摄的那些照片。

Flickr提供了一种组织图片的方式，正好可为我们所用，它允许我们创建一些**照片集**（photo set），并把任何已上传图像加入这个照片集。把照片加入一个照片集并不会对照片的使用产生任何影响，照片也可以再加入到其他照片集，所以我们可以放心地使用照片集功能而不必担心对Flickr上的照片的其他用途产生人为约束。

创建和管理照片集与Flickr的其他部分一样简单。直接点击Organize页签（tab）进入Organizer界面，就可以创建和管理你的照片集了。当你上传了旅行照片后，请为旅行数据文件中所列出的每个景点创建一个照片集，然后加入合适的照片。

在我们的应用程序中用来获得照片集信息的Flickr方法要求我们用照片集**ID**而不是名字来指定照片集。这个ID没有在Flickr网站上列出来（至少我们没能找到方法显示它），但是Flickr API有一个方法flickr.photosets.getList，可以让我们获取我们的所有照片集的信息，包括它们的ID。

我们其实不用每次都查询ID，所以没有必要写代码来获取这个信息，只要用和下面一样的URL访问Flickr服务：

```
http://www.flickr.com/services/rest/?method=flickr.photosets.getList
&api_key=78eaa5287f3f0b37dfb47def40c7df13&user_id=95355920@N00
```

然后从结果中找出要使用的照片集的ID。当然，在上面这个URL里，你得换上你自己的API key和Flickr账户的NSID。

有了这些，我们终于可以开始写代码啦！

13.3 TripomaticDigester 类

我们手上有Prototype工具包听凭调遣，我们前后看过许多高级JavaScript的例子，我们准备好了Flickr照片集，我们也已经决定了旅行数据的格式。那么让我们开始干正事吧！

我们要面对的首个任务就是解读我们在前一节中定义的旅行数据文件。我们的脚本要获取XML文档，解读它，并创建一个JavaScript结构体来持有旅行数据。但是我们不会把这些脚本直接塞到页面里去，而是创建一个JavaScript对象来处理这一任务。

实际上，我们会将这一任务与Trip-o-matic应用的其余部分完全分离开来。为什么要这样做呢？

把数据解读代码与应用逻辑分离开来，不仅能让代码更有序，并且通过把这一职责从其他应用代码中完全抽取出来，我们解耦（decouple）了数据读取任务与数据处理任务，这让我们在将来任何时候都能重新考虑数据文件格式。假设某个时候我们想改变XML格式，甚至判定XML是个错误的选择而决定回到JSON路线，我们也只需要去修改解读器[①]，而不会影响到其他部分。这种程度的抽象，作为关注点分离（separation of concerns）概念的小小例证，对于用服务器端语言编写的代码来说，这是人所共知的准则（de rigueur），我们的客户端JavaScript自然也应遵从。

我们的解读器的类叫做TripomaticDigester，它会在TripomaticDigester.js文件中定义。下面，让我们研究这个类的具体实现的各个部分。

13.3.1　依赖性检查

你是否曾经从浏览器的JavaScript引擎中获得一些像密语一样的错误消息，最终却发现只是因为缺少了几个脚本？我们都曾浪费大量时间一边抓耳挠腮一边拼命想找出到底我们的JavaScript代码出了啥毛病，然后发现是忘记导入代码所依赖的某些.js文件，恨不能以头抢地。

我们的解读器以及这个应用的其他JavaScript代码都严重依赖Prototype库。要是在导入我们的解读器类之前没有预先载入Prototype库，自然会出现错误。要避免得到浏览器那典型的密语式的错误消息，我们可以执行一个显式的检查，然后发送我们自己的——但愿能更加清晰的错误消息。

所以，在解读器的脚本文件的顶部我们这样写：

```
if (typeofPrototype=='undefined') {
  throw new Error(
    "Prototype must be in scope to use Trip-O-Matic");
}
```

Prototype库定义了一个名为Prototype的对象，我们检查一下它是否存在。如果没有定义它，我们就扔出一个JavaScript错误，它会显示在JavaScript控制台上（或者显示在一个弹出窗口中——这取决于所使用的浏览器），清楚地告知我们所留下的疏漏。

如果再考究一点，Prototype对象上有一个属性声明了Prototype库的版本号，我们也可以检查一下这个版本号。

如果你需要对其他JavaScript库的依赖性进行检查，也可以参考这个简单的技巧。

13.3.2　TripomaticDigester 的构造器

TripomaticDigester构造器的职责是取得旅行数据的资源URL，解读该资源所指定的XML文档，存储解读后的数据以备检索，并在任务完成之后通知调用者。完成它一定能算是个壮举，不过我们可以每次只做一件事情，并且把一些任务委托给那些负责具体实现的函数，这样我们就会看到它并不像听起来那么难。

因为我们是使用Prototype库来创建我们的JavaScript类，所以TripomaticDigester的构造器由两部分组成，首先创建TripomaticDigester类，然后在类的prototype属性上定义一个初始化函数，如代码清单13-1所示。注意这个清单并没有包括类的全部代码，我们一次只看类的一个部分就好。

---

① 即执行数据解读任务的类。——译者注

代码清单13-1 TripomaticDigester构造器的组成部分

```
TripomaticDigester = Class.create();

TripomaticDigester.prototype = {

  initialize: function(dataUrl,onLoadHandler) {
    this.dataUrl = dataUrl;
    this.onLoadHandler = onLoadHandler;
    new Ajax.Request(
      this.dataUrl,
      {
        onSuccess: this.onDigest.bind(this),
        onFailure: function() {
        throw new Error('failed to load ' + this.dataUrl);
      }
    }
  );
},
```

❶ 保存初始化参数

❷ 通过Ajax读取数据文件的内容

❸ 将onSuccess事件处理器与当前实例绑定

这个代码看上去不赖，不过实话说，它也没干多少事。方法initialize()要接收的参数包括旅行数据资源的URL和一个函数引用，后者在旅行文件加载完成时会被调用。在保存了参数值之后❶，构造器会发送一个Ajax请求来读取指定URL的内容❷。

注意，在给请求指定onSuccess事件处理器时❸，我们使用了Prototype的bind()方法来确保onDigest()方法的上下文对象就是当前这个TripomaticDigester类的实例。

如果你需要回顾绑定上下文对象是怎么回事以及它与宠物蜥蜴之间的关系，请参考3.1.2节。

直到调用onDigest()回调方法的时候，这才开始真正的工作。

13.3.3 解读旅行数据

如果提供给构造器的URL是有效的，并且也没出其他问题，那么当从服务器接收到XML文档时，就会接着调用onDigest()方法。方法的实现代码如清单13-2所示。

代码清单13-2 TripomaticDigester.onDigest()方法

```
onDigest: function(request) {
  var xmlDoc = request.responseXML;
  var tripElement = xmlDoc.childNodes.item(0);
  if (tripElement.nodeName != 'trip')
    throw new Error( 'root element must be <trip>' );
  this.title = tripElement.getAttribute('title');
  this.flickrNSID = tripElement.getAttribute('flickrNSID');
  this.flickrKey = tripElement.getAttribute('flickrKey');
  this.poiTitle = tripElement.getAttribute('poiTitle');
  this.description = '';
  var self = this;
  var descriptionElements =
    tripElement.getElementsByTagName('description');
  $A(descriptionElements).each(function(descriptionNode) {
   self.description += self.collectText(descriptionNode);
  });
```

```
    this.points = new Array();
    var poiElements = tripElement.getElementsByTagName('poi');
        $A(poiElements).each(this.loadPoint.bind(this));
        this.onLoadHandler(this);
     },
```

这个函数执行的任务相当重要，但也很简单明了，就是解析并读取XML。从request参数（XMLHttpRequest类的一个实例）上可以得到XML文档对象。我们读取它的第一个（也是唯一一个）子节点，也就是文档的根元素，然后检查一下它是否是我们所要的<trip>节点。

如果检查成功，我们就读取元素的属性并保存到解读器的实例变量中。然后收集所有的<description>子元素的文本内容，这需要调用负责具体实现的collectText()方法（我们过一会儿再描述它）。注意，我们使用了Prototype库的$A()函数把NodeList（节点列表）转换为Array，然后调用数组上的each()方法来收集所有<description>节点的内容。

等一等！难道有很多节点吗？不是只有一个？

没错。我们会比较宽松，允许用户把对旅行的描述分散到多个元素中——如果他想要这么做的话。当然我们可以苛刻一点，只允许一个<description>元素——但是为什么不友善一点呢？①

然后要收集所有的<poi>子元素，这一步骤同样是利用Prototype库的$A()和each()函数，针对每个<poi>元素都调用loadPoint()方法（注意，要先绑定到解读器对象实例上）。这是一个很好的例子，它向我们展示了，借助Prototype的Enumerable类的each()方法，我们很容易就能把对数组元素的复杂处理过程分离到一个单独函数中，从而令我们的代码更易于管理。

方法loadPoint()负责解读<poi>元素，你会在下一节进一步了解这个方法。

此前客户代码通过构造器注册了一个处理器，最后我们就调用这个函数并把解读器实例作为参数传递给它。这样，对于处理器来说就无需通过全局变量或其他外部方式来挂接解读器。

现在让我们转到真正令人感兴趣的数据，也就是所谓兴趣点（景点）②了！

13.3.4　加载经典信息

在代码清单13-2中的onDigest()方法中，我们对<trip>元素下的每个<poi>元素都调用了名为loadPoint()的方法。这个方法绑定了解读器实例，其职责是从传给它的元素中采集信息，并据此创建用于描述每个景点的JavaScript对象。其实现代码列于代码清单13-3。

代码清单13-3　TripomaticDigester.loadPoint()方法

```
loadPoint: function(poiElement,index) {
    this.points.push({
      name: poiElement.getAttribute('name'),
      latitude: poiElement.getAttribute('latitude'),
      longitude: poiElement.getAttribute('longitude'),
      photoSetId: poiElement.getAttribute('photoSetId'),
```

① 允许多个<description>元素是一个有弹性的设计，例如可以使用xml:lang属性表示几种不同语言的描述；再比方说，描述可能分几次写成，可能是几个人写的，我们可以把它们记录成多个description元素。——译者注

② 原文为points of interest，意译为“景点”，直译就是“兴趣点”。——译者注

```
        description: this.collectText(poiElement)
      });
    },
```

根据所传入的元素的属性来创建JavaScript对象只是琐事一桩：在元素上调用DOM的getAttribute()方法得到每个属性值，赋值给JavaScript对象上对应名字的属性。这个新创建的对象实例然后会被追加（push）到points数组实例变量以备日后之用。

我们使用collectText()方法从<trip>元素下的<description>子元素中取得描述数据，同样地，<poi>元素的描述数据也是通过collectText()方法来收集的。

好吧，那鬼家伙[①]到底长啥样呢？

13.3.5 收集元素的文本内容

从我们前面在解读器类中创建的那些方法中可以看到，读取XML文档中的元素属性值是很简单的，获取某个元素的子元素也难不了多少。但是，要从元素体中采集文本数据，就有一点挑战了。

不像元素属性或者子元素，并没有一个DOM方法能让你直接取得元素体中包含的所有文本[②]。如果你思考一下，也很容易理解其中的原因。在我们的文档中可能有这样一个XML文档片段：

```
<description>
  The quick young cub jumped over the lazy bear.
</description>
```

这看上去相当简单，并且在绝大多数XML解析器中，<description>元素体的内容会是单独一个文本节点。但是如果像下面这样的呢？

```
<description>
  The quick young cub <!-- was he that quick? --> jumped
  over the lazy bear.
</description>
```

这个片段里加入了注释节点，这导致XML解析器会把这段文字分成（至少）两个文本节点，当中有一个注释节点。

下面这段代码又如何呢？

```
<description>
  <![CDATA[
   The <strong>quick</strong> young cub jumped over the
  <i>lazy</i> bear.
  ]]>
</description>
```

在这个片段中，元素体中的文本是以CDATA片段（CDATA section）的形式插入的，这样它

---

① 指collectText()方法。——译者注

② 实际上DOM Level 3增加了textContent属性，可以直接取得元素中的所有文本，多数浏览器都已经支持，当然IE除外。但是，IE的XML DOM专有的text属性或IE的HTML DOM专有的innerText属性与textContent是基本等价的。——译者注

就能包含那些会在文档中引起语法错误的特殊字符——本例中就是一些HTML标记。CDATA信息有自己的节点类型。

要如何处理这些情况，取决于应用程序的需要。对于我们的例子来说，我们会收集所有的文本和CDATA节点（忽略注释），把它们拼接成一整个文本块，作为节点的内容。这适用于旅行描述，也适用于景点元素。

最后，我们创建collectText()方法，其具体实现代码列于代码清单13-4中[①]。

代码清单13-4　TripomaticDigester.collectText()方法

```
collectText: function(element) {
  var text = '';
  $A(element.childNodes).each(
    function(child) {
      if ((child.nodeName == '#text') ||
          (child.nodeName == '#cdata-section')) {
        text += child.data;
      }
    }
  );
  return text.strip();
}
```

这个方法接受一个元素作为参数，方法会遍历该元素的所有子节点以寻找文本和CDATA节点，也就是节点名称为#text和#cdata-section的那些节点。我们把所有的这样的节点拼接到一个文本变量中，去除头尾空白（通过Prototype库的String.strip()方法）后作为方法的返回值。

我们已经完成了独立的解读器类。再强调一次，在单独的类中定义这个过程，能使数据读取与应用程序的其他部分解耦合。这给了我们很大的自由，能对解读过程的各个细节包括数据格式进行修改，而不必担心会让应用程序出现错误。

13.4　Tripomatic应用类

现在我们已经解决了数据解读的问题，让我们把注意力转向到应用本身。我们当然可以直接编写一个页面并嵌入JavaScript代码来实现我们的应用，但是我们要做得更聪明一些。在整本书中我们一直高举面向对象的主旋律旗帜，所以我们将建立一个JavaScript类来实现整个应用。

这个应用头绪众多。我们有旅行信息、景点、地图，还有缩略图和照片需要创建、管理和操作。不过，只要每次前进一小步，并且结合运用面向对象编程、JavaScript语言以及Prototype库所给予我们的力量，我们就能最终达成目标。

本应用需要完成的任务如下所示。

- 创建DOM元素用于显示应用的内容。这包括：
  - 旅行的题目；

① 注意，这个函数并不能处理嵌套节点中包含文本的情形。——译者注

- 旅行的描述；
- 景点，每个景点都是可点击的；
- 景点列表的标题；
- 所点击的景点的地图；
- 在景点所摄照片的缩略图；
- 缩略图所对应的全尺寸的照片。

- 指派事件处理器，作用是：
  - 点击景点后使用Yahoo! Maps显示出对应的地图；
  - 点击地图后使用Flickr服务读取并显示出与该地图的景点相关的照片集所对应的缩略图；
  - 点击缩略图后再次使用Flickr服务显示出该缩略图所对应的全尺寸照片。

所有这些也需要通过面向对象的方式完成，而不能出现全局变量或任何硬性编码元素ID。我们首先来建立Tripomatic类及其构造器。

13.4.1 Tripomatic 类和构造器

我们先创建一个名为Tripomatic.js的文件，加入与之前TripomaticDigester类相同的依赖性检查。我们也再次使用Prototype的创建类的方式，建立了Tripomatic类的骨架，代码列于代码清单13-5中[①]。这个代码清单当然远非类的完整定义，不过我们会在本节中逐渐完善它，直到得到一个完整可运行的应用。本节后续代码清单通常不会重复列出已经列出过的代码，而是会着重描述新添加的代码。

代码清单13-5 Tripomatic类的骨架

```
if (!Prototype) {
  throw new Error(
    "Prototype must be in scope to use Trip-O-Matic");
}

if (!TripomaticDigester) {
  throw new Error(
    "TripomaticDigester must be in scope to use Trip-O-Matic");
}

Tripomatic = Class.create();

Tripomatic.prototype = {
  /* instance methods will go here */
}
```

和往常一样，以Prototype方式所定义的类，其实际的构造代码会被置于名称为initialize()的方法中。在本例中，方法的参数分别是包含旅行信息的数据文件的URL、用于放置应用内容的DOM元素，以及用于传递可选信息的选项表。该方法的代码列于代码清单13-6中。

① 与之前的代码类似，这里的依赖性检查代码也存在错误，正确方式是使用typeof来检测一个变量是否存在。——译者注

代码清单13-6　Tripomatic.initialize()方法

```
initialize: function(dataUrl,container,options) {
  this.container = $(container);
  this.options = Object.extend(
    {
      enablePanAndZoom: false
    }, options
  );
  this.createContent();
  this.digester = new TripomaticDigester(
    dataUrl,
    this.onDataLoaded.bind(this));
  this.map = new YMap(this.mapContainerElement);
  if (this.options.enablePanAndZoom) {
    this.map.addPanControl();
    this.map.addZoomLong();
  }
},
```

这个方法看上去挺简单，因为它只是把所有困难的工作委派给其他方法。参数container存到一个实例变量中，而参数options则与默认选项合并。注意，这个类只定义了一个选项enablePanAndZoom，它指定了是否给Yahoo! Maps的地图加入Pan和Zoom控件[①]。

不过，如果我们只有一个选项，干嘛要用一个对象hash作为选项表呢？答案就是可扩展性。像这种复杂的应用，很有可能会需要加入更多的选项。如果我们现在不使用选项表，那么等到将来为了容纳额外选项而将参数改为hash时，我们就需要改变构造器的签名，所有已经在页面中使用该类的那些既有代码就都会被破坏。如果未雨绸缪，我们就能在将来添加选项而不更改构造器签名，因此也就避免了让我们这个类的使用者去进行不必要的修改。

方法接着调用了名称为createContent()方法来执行具体工作。该方法会创建显示本应用的数据内容所需的所有元素，后面我们会深入研究它。

在创建了内容元素之后，会使用所传入的dataUrl创建TripomaticDigester类的实例，并保存起来以备后用。当解读器完成解读之后，就会调用名为onDataLoaded()的方法。注意，我们将函数的上下文（this）绑定到了这个回调函数，于是onDataLoaded()在被调用时也会以这个Tripomatic实例作为函数上下文。

最后，我们在容器中创建和初始化了Yahoo! Maps地图，你不难想到这个容器是在createContent()中创建的（事实上也的确如此，稍后我们就会看到）。

至此，应用已完成初始化并准备好进行用户交互。不过，有很多细节隐藏在createContent()和onDataLoaded()方法背后。让我们看一看这些方法都为我们做了些什么。

13.4.2　创建内容元素

现在我们涉及了“创建还是挂接”的抉择。处理应用的内容有两种方式。一种是我们在第3章的Button类的例子中已经使用过的方式，由使用者定义HTML元素，我们的类则在初始化时挂

① Pan是罗盘控件，可控制地图视点朝不同方向移动；Zoom是缩放控件，可控制地图缩放的比例。——译者注

接到这些元素上。

这种方式适合于HTML非常简单或在自行创建时有太多变化需要操心的情况。但是，对于需求相当复杂但是可预先确定的层级结构的情形，更好的方式可能是由脚本来动态创建元素。

对于Trip-o-matic应用，我们就会采用后一种方式。这个步骤是在构造器所调用的createContent()方法中完成的。

createContent()方法的职责是在参数所指定的容器元素内创建DOM层级结构，该DOM结构等价于下面所列的HTML片段：

```
<h1></h1>
<h2></h2>
<div class="tripomaticPoiContainer">
  <h3></h3>
  <ul></ul>
</div>
<div class="tripomaticMapContainer"></div>
<div class="tripomaticPoiDescription"></div>
<div class="tripomaticThumbsContainer"></div>
<div class="tripomaticPhotoContainer"></div>
```

我们的旅行的各种数据内容就会插入到这些元素中。

<h1>和<h2>元素分别用于显示旅行的标题和描述。 元素包含<h3>和空的<ul>元素用来显示景点列表。数据文件中的<trip>元素的poiTitle属性字符串将会被置于<h3>元素中，而景点项则会被填入<ul>元素。

每个 元素都各有一个不同的CSS类名（所有类名都以tripomatic作为前缀，这样就不太会与用户的CSS类名发生冲突），不仅帮助标识出了每个元素的用途，也让使用者可以为所有元素定制样式。当你为使用者创建元素时，这一点是很重要的。所以我们必须确保每个元素都能够通过CSS选择器来选定。如果无法以CSS选择器选定，用户就无法定制它的样式。这里的其他非 的元素不需要特别加上类名，因为它们在这个HTML结构中恰好都是唯一的，所以可以使用元素名CSS选择器来选定。

通常应该避免把字符串写死在代码中，所以我们要把表示类名的字符串从代码中提取出来放入类一级的引用中。正如我们在第3章所探讨过的，我们可以通过类的属性赋值来模拟类一级的“常量”。当然，我们知道它们不是真的常量，不过我们把它当作常量来对待。这些常量赋值操作处于Tripomatic.js文件中的类定义和原型定义之间，代码列于代码清单13-7。

代码清单13-7 类名“常量”

```
Tripomatic = Class.create();

Tripomatic.CLASS_POI_CONTAINER = 'tripomaticPoiContainer';
Tripomatic.CLASS_MAP_CONTAINER = 'tripomaticMapContainer';
Tripomatic.CLASS_POI_DESCRIPTION = 'tripomaticPoiDescription';
Tripomatic.CLASS_THUMBS_CONTAINER = 'tripomaticThumbsContainer';
Tripomatic.CLASS_PHOTO_CONTAINER = 'tripomaticPhotoContainer';

Tripomatic.prototype = {
```

看过这些常量，我们就可以接着研究创建DOM元素的createContent()方法（代码清单13-8）。

代码清单13-8　Tripomatic.createContent()方法

```
createContent: function() {
  this.container.innerHTML = '';
  this.tripTitleElement = document.createElement('h1');
  this.tripDescriptionElement = document.createElement('h2');
  this.poiContainerElement = document.createElement('div');
  this.poiTitleElement = document.createElement('h3');
  this.poiListElement = document.createElement('ul');
  this.mapContainerElement = document.createElement('div');
  this.poiDescriptionElement = document.createElement('div');
  this.thumbsContainerElement = document.createElement('div');
  this.photoContainerElement = document.createElement('div');
  this.container.appendChild(this.tripTitleElement);
  this.container.appendChild(this.tripDescriptionElement);
  this.container.appendChild(this.poiContainerElement);
  this.container.appendChild(this.mapContainerElement);
  this.container.appendChild(this.poiDescriptionElement);
  this.container.appendChild(this.thumbsContainerElement);
  this.container.appendChild(this.photoContainerElement);
  this.poiContainerElement.appendChild(this.poiTitleElement);
  this.poiContainerElement.appendChild(this.poiListElement);
  this.poiContainerElement.className =
    Tripomatic.CLASS_POI_CONTAINER;
  this.mapContainerElement.className =
    Tripomatic.CLASS_MAP_CONTAINER;
  this.poiDescriptionElement.className =
    Tripomatic.CLASS_POI_DESCRIPTION;
  this.thumbsContainerElement.className =
    Tripomatic.CLASS_THUMBS_CONTAINER;
  this.photoContainerElement.className =
    Tripomatic.CLASS_PHOTO_CONTAINER;
},
```

Tripomatic.createContent()可能有点冗长，不过还是相当简单直接的。首先，页面创作者传入构造器的容器元素会被清空，然后应用所需的各个元素被创建并存于实例的属性中。

接着，createContent()方法将元素挂接组合，形成本节之前所描绘的HTML层级结构，并作为子元素插入到页面创作者的容器中。最后，给所创建的元素加上CSS类名。

好了，我们干得不赖。不过这只是创建了一组空元素，还没有任何实际的旅行数据。下面让我们看一下如何获取和填充数据。

13.4.3　填充旅行数据

如果你还记得，在我们的initialize()方法中有一个步骤就是创建TripomaticDigester类的实例。我们传给它的参数有旅行数据文件的URL，还有名为onDataLoaded()的方法的引用，该方法会在解读旅行数据后被调用。

我们会在创建内容元素后创建解读器实例，这样当onDataLoaded()被调用时，我们就可以

安全地假设DOM层级结构已经组装完成了。那么让我们看一看这个方法的代码（代码清单13-9）。

代码清单13-9 Tripomatic.onDataLoaded()方法

```
onDataLoaded: function(digester) {
  this.tripTitleElement.innerHTML = digester.title;
  this.tripDescriptionElement.innerHTML = digester.description;
  this.poiTitleElement.innerHTML = digester.poiTitle;
  digester.points.each(this.makePointOfInterest.bind(this));
  this.showPoint(digester.points[0]);
},
```

解读器会将自身的引用传入Tripomatic.onDataLoaded()方法，我们使用该引用来获取解读后的数据。不过就算解读器没有提供这一便利条件，我们仍有办法获取数据，因为这个方法已经绑定到当前的Tripomatic实例，所以我们可以访问保存在实例属性中的解读器引用。啊哈，这就是面向对象的乐趣！

显示旅行的题目和描述只需将（我们在createContent()方法中建立的）相应内容元素的innerHTML属性设为相应的值即可。然后，我们在景点列表上使用Prototype提供的each()方法，并指定makePointOfInterest()方法作为迭代函数。

在将控制交给页面访问者之前（记住，所有这些操作会在页面加载时发生），我们调用名为showPoint()的方法，并传入景点列表中第一个景点的引用。这样列表中第一个景点的信息就会显示在页面上，而当访问者点击任何一个景点的名称时，也会调用该方法来显示所点击的景点的信息。我们会在13.4.4节中了解它的具体实现。

对于解读器加载进来的每个景点，都会调用作为迭代函数的makePointOfInterest()方法。这个方法承担的任务是在景点列表中创建景点目录项，访问者可以点击它来加载该景点的地图。方法的具体实现列于代码清单13-10中。

代码清单13-10 Tripomatic.makePointOfInterest()方法

```
makePointOfInterest: function(point) {
  var pointItem = document.createElement('li');
  pointItem.appendChild(document.createTextNode(point.name));
  pointItem.onclick = this.onPoint.bindAsEventListener(this);
  pointItem.point = point;
  this.poiListElement.appendChild(pointItem);
},
```

Tripomatic.makePointOfInterest()方法使用DOM操作API创建一个<li>元素来作为景点目录项。其click事件的事件处理器被设为onPoint()方法，并作为事件监听器绑定了当前实例。这样事件处理器可以很容易获得该项景点的信息。我们同时也将景点保存在<li>元素的名为point的属性中。然后所创建的<li>元素会被追加到它的父元素，也就是我们在初始化时所创建的<ul>元素之下。

在最后一个景点上调用了这个方法之后，页面加载时所要进行的一系列步骤也就执行完毕了。如果使用位于www.manning.com/crane2的本章源代码下载包中所提供的样例XML数据文件，访问者将看到如图13-1所示的结果。

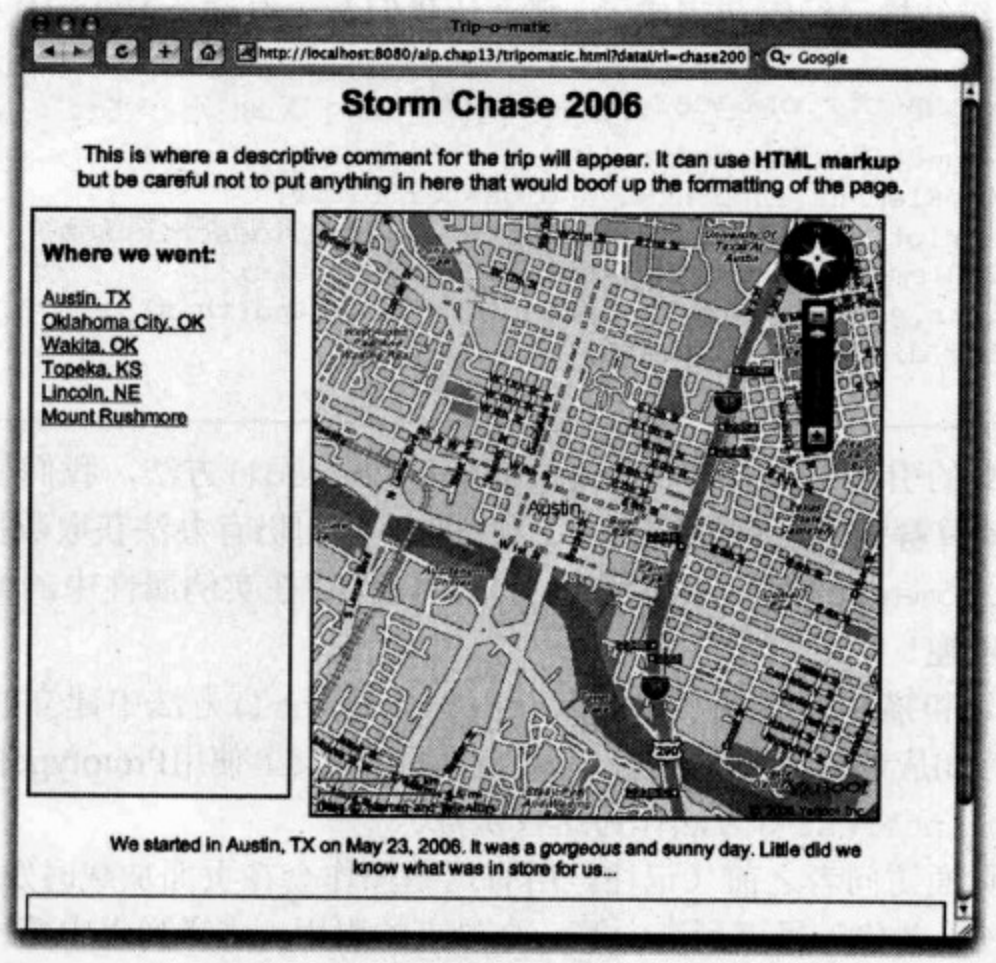


图13-1　我们在奥斯汀！

我们可以在图13-1中看到旅行的标题、描述、景点列表以及第一个景点的地图。页面底部空白区域将会加载一组缩略图，点击之后则会显示出全尺寸的照片。

不过地图是怎么来的呢？

13.4.4　显示地图

因为我们在景点目录项上设置了事件处理器，所以访问者点击某个景点名称时会触发onPoint()方法。作为事件处理器，这个方法被调用时会有一个Event实例作为参数，由该Event实例我们可以确定点击的是哪一个<li>元素。你应该记得，我们之前将景点信息保存在<li>元素的名为point的属性中，这样在本方法中就能轻松访问。事件处理器方法的代码列于代码清单13-11。

代码清单13-11　Tripomatic.onPoint()事件处理器方法

```
onPoint: function(event) {
  this.showPoint(Event.element(event).point);
},
```

这个处理器的唯一操作就是定位事件目标元素上的景点信息，并将其传入showPoint()方法。showPoint()方法绘制景点所对应的地图。在用户点击景点列表的某一项时会调用该方法，还有，你可能记得，在数据初始加载时也会调用该方法。让我们看一看它的具体实现——代码清单13-12。

代码清单13-12 Tripomatic.showPoint()方法

```
showPoint: function(point) {
  this.currentPoint = point;
  var geoPoint = new YGeoPoint(point.latitude,
                               point.longitude);
  this.map.drawZoomAndCenter(geoPoint,4);
  this.mapContainerElement.onclick =
    this.showThumbnails.bindAsEventListener(this);
  this.mapContainerElement.point = point;
  this.poiDescriptionElement.innerHTML = point.description;
  this.thumbsContainerElement.innerHTML = '';
  this.photoContainerElement.innerHTML = '';
},
```

在将传入该方法的景点记录为当前景点之后，就像我们在12.1.1节中所研究过的示例一样，方法创建了YGeoPoint实例并使用它来显示以景点位置为中心的YMap。

然后我们在地图元素上安置了单击事件处理器，我们将名为showThumbnails()的方法绑定为事件处理器。我们稍后就会看看这个方法。

接着我们显示出景点的描述，并将缩略图和照片容器清空以移除与之前所显示的景点相关的图像。

现在我们就可以坐下来耐心等着访问者点击地图了。

13.4.5 加载缩略图

如果访问者最终发现点击地图会发生有趣的事情——记住，我们已经声明过本应用并非优使性的范本——那些有直觉力的人将得到回报[①]。当点击了地图后，我们要读取保存于Flickr账号中的缩略图信息，这些缩略图是通过照片集的ID与当前显示地图的景点关联在一起的。

我们在12.3节中已经试过发起Flickr请求，所以这里所要完成的代码看上去会相当熟悉，尽管运用Flickr的REST API时我们将使用一个不同的Flickr方法。

我们为每个景点建立了对应的Flickr照片集，并在旅行数据文件中记录了这些照片集的ID。所以我们不是使用flickr.people.getPublicPhotos这个Flickr方法来取回所有公开照片，而是使用flickr.photosets.getPhotos方法将检索范围限制在以ID指定的照片集内。图13-2显示了在点击地图后页面底部会显示相关的缩略图。响应点击执行这一操作的事件处理器的代码列于代码清单13-13中。

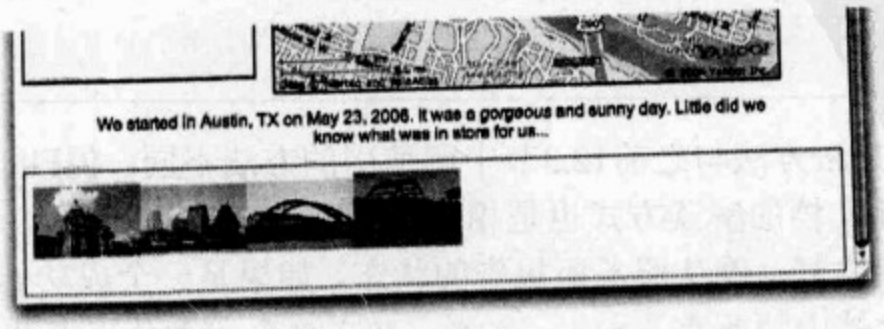


图13-2 奥斯汀照片的缩略图

① 作者的意思是此处的优使性实际上并不好，没有“直觉力”，恐怕发现不了点击地图会显示缩略图。——译者注

代码清单13-13　Tripomatic.showThumbnails()方法

```
showThumbnails: function(event) {
    new Ajax.Request(
      '/aip.chap13/proxy',
      {
        onSuccess: this.onPhotosetList.bind(this),
        method: 'get',
        parameters: {
          '.serviceUrl.':
            'http://www.flickr.com/services/rest/',
          api_key: this.digester.flickrKey,
          method: 'flickr.photosets.getPhotos',
          photoset_id: this.currentPoint.photoSetId
        }
      }
    );
  },
```

与我们所发起的许多其他Ajax请求一样，这个方法也是通过我们的服务器端代理来联络Flickr服务的。12.1.2节研究了这个代理的目的以及所执行的操作。

我们事先已经将当前所载入的景点信息保存于名为currentPoint的实例属性中。这样，事件处理器就可以方便地使用这个属性来引用当前景点的photoSetId属性，而无需借助全局变量来保存引用或景点数组的索引。

对于我们发起的请求，Flickr服务会返回一个XML文档作为响应。请求的onSuccess事件处理器（代码清单13-14）被指定为onPhotosetList()方法，负责解读返回给我们的这个XML文档。我们来看一看它的具体实现。

代码清单13-14　解读照片集文档

```
onPhotosetList: function(xhr) {
    var doc = xhr.responseXML;
    var status =
      doc.getElementsByTagName('rsp')[0].getAttribute('stat');
    if (status == 'ok') {
      this.thumbsContainerElement.innerHTML = '';
      var photos = doc.getElementsByTagName('photo');
      $A(photos).each(this.makeThumbnail.bind(this));
    } else {
      throw new Error('getPhotos request failed');
    }
  },
```

尽管我们调用的Flickr方法与之前12.3节中所使用的方法不同，但Flickr服务返回的XML文档的格式是相同的。因此文档的解读方式也是相同的。

我们首先获得响应文档，确认服务所报告的状态。如果是一个成功的响应，我们会清除先前的缩略图，取得响应文档中的所有<photo>元素，并在每个元素节点上调用makeThumbnail()方法作为迭代函数。makeThumbnail()方法的代码列于代码清单13-15中，创建缩略图图像的重要

工作就在这个方法中完成。（注意如果是一个失败的响应，我们会抛出一个Error实例。在实际应用中，我们可能要从响应文档中提取出错误信息，而不是直接使用整个响应文本来构造Error实例[①]，不过为简短起见，我们在这里省去了这个步骤。你可以自行练习如何改进这里的代码，提供更好的错误检测和恢复。）

代码清单13-15 Tripomatic.makeThumbnail()方法

```
makeThumbnail: function(photo,index) {
    var baseUrl = 'http://static.flickr.com/' +
                  photo.getAttribute('server') + '/' +
                  photo.getAttribute('id') + '_' +
                  photo.getAttribute('secret');
    var thumbUrl = baseUrl + '_t.jpg';
    var photoUrl = baseUrl + '.jpg';
    var thumb = document.createElement('img');
    thumb.src = thumbUrl;
    thumb.style.cursor = 'pointer';
    thumb.onclick = this.showPhoto.bindAsEventListener(this);
    thumb.photoUrl = photoUrl;
    this.thumbsContainerElement.appendChild(thumb);
},
```

❶ 构造照片的URL

❷ 存储URL以备后用

你会发现这些重要工作并不太复杂。就像在12.3.2节中所做的，我们构造URL以引用位于Flickr网站上的缩略图，然后创建HTML<img>元素指向该URL。

当我们生成缩略图的URL时，我们也生成了点击缩略图时所要显示的照片的全尺寸版本的URL❶，并将它保存在缩略图<img>元素的属性中❷。

稍后，在showPhoto()方法——也就是我们为缩略图<img>元素设置的onclick事件处理器中，我们会创建用于全尺寸照片的<img>元素，届时就会用到之前所记录的URL。

13.4.6 显示照片

我们做的工作还真不少，不过最后我们终于准备好要显示照片了！如图13-3所示，传说中的照片终于出现在页面的底部。

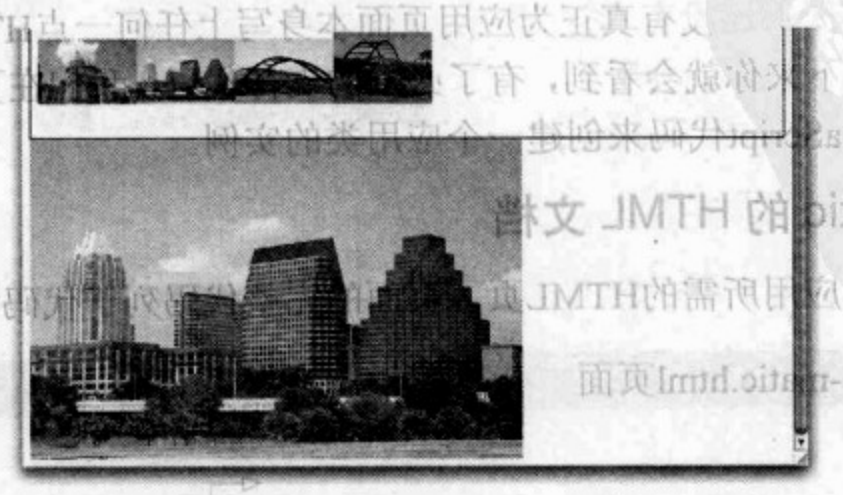


图13-3 奥斯汀的城市景色

① 代码清单中其实也没有使用响应文本，而是直接使用了一个字符串作为错误消息。——译者注

在代码清单13-15中的makeThumbnail()方法中，我们指定了名为showPhoto()的方法作为每个缩略图图像元素的单击事件处理器。这个方法（代码清单13-16）与我们在12.3.2节中建立的对应功能的函数十分类似。

代码清单13-16　Tripomatic.showPhoto()方法

```
showPhoto: function(event) {
  if (this.photoElement == null) {
    this.photoElement = document.createElement('img');
    this.photoContainerElement.appendChild(this.photoElement)
  }
  this.photoElement.src = Event.element(event).photoUrl;
}
```

这个函数是缩略图图像的单击事件处理器，所以当函数被调用时，缩略图的<img>元素会是事件的目标元素。因为我们明智地将点击缩略图所要显示的照片的URL预存在该元素的属性中，所以函数的工作就变得相当简单：它检查用于显示照片的<img>元素是否已经创建，如果没有就创建它。在定位到已有的元素或创建新元素之后，将元素的src属性设为所要显示的照片。

这里你可能会问为什么不在createContent()方法中创建用于全尺寸照片的<img>元素？为什么要等到这个方法再来创建？

这是因为，在执行createContent()方法时，我们还没有照片URL可供显示。事实上，直到页面的访问者点击地图，然后点击某个缩略图之后，我们才会得到URL。如果我们先创建一个没有src属性或src为空的<img>元素，许多浏览器会显示出一个“无法显示图像”的图标，直到我们换上一个有效的src属性值，而我们并不希望在页面上显示出这样的东西。

有了showPhoto()方法后，我们的应用类终于算完成了。现在让我们把它放入HTML页面中。

13.5　Trip-o-matic 应用页面

直到此刻，你可能一直在催促，“我们是不是能开始编写页面了啊？”您的愿望，我们倾听。虽然看上去我们忙了半天却还没有真正为应用页面本身写上任何一点HTML，但是所有这些辛苦工作都是有回报的，接下来你就会看到，有了必须的支持代码，我们在页面上所要做到就只是写一点HTML、CSS和JavaScript代码来创建一个应用类的实例。

13.5.1　Trip-o-matic 的 HTML 文档

下面，让我们建立应用所需的HTML页，页面的完整代码列于代码清单13-17。

代码清单13-17　trip-o-matic.html页面

```
<html>
  <head>
    <title>Trip-o-matic</title>
    <link rel="stylesheet" href="styles.css"/>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
```

❶ 导入样式表、脚本库和类

```
  <script type="text/javascript" src=
    "http://api.maps.yahoo.com/ajaxymap?v=2.0&appid=YahooDemo">
  </script>
  <script type="text/javascript" src="TripomaticDigester.js">
  </script>
  <script type="text/javascript" src="Tripomatic.js">
  </script>
  <script type="text/javascript">
    window.onload = function() {
      var dataUrl = document.URL.toQueryParams().dataUrl;
      if (dataUrl == null)
        throw new Error('the dataUrl parameter must be set');
      new Tripomatic(
        dataUrl,
        'tripomatic',
        {
          enablePanAndZoom: true
        });
    };
  </script>
</head>

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

❷ 创建应用的实例

❸ 声明应用的元素所需要的容器

元素<head>❶中导入了CSS样式表，提供了我们要应用到页面上的基本样式，同时也导入了我们需要的多个外部JavaScript文件：Prototype程序库、Yahoo! Maps程序库、解读器类以及Tripomatic类。

<head>也包括了一个定义页面onload事件处理器的脚本元素。在这个处理器函数中，我们借助Prototype库提供的toQueryParams()字符串方法读取dataUrl查询参数得到数据文件的URL。如果没有dataUrl参数，我们会抛出错误报告。

然后我们创建一个Tripomatic类的实例❷，参数包括数据文件的URL、容器元素的引用❸以及选项表。

在页面的<body>中，我们创建了一个空的 元素作为容器，应用所创建的元素将被置于其中。

就是这些了。我们前面所完成的那些代码将会处理剩下的事情。看到没有，页面创作者有多轻松啊！

经过漫长的旅程，我们终于完成了应用页面了！不过在我们开始庆祝之前，下一小节会简要讨论一下页面头部所链接的styles.css文件的内容。如果你有兴趣了解页面布局的设计，可以看看这个部分。

13.5.2 样式之旅

我们所使用的样式文件styles.css绝算不上出众。它的目标只是让页面具有最低限度的优

使性，并且避免影响我们对于页面核心功能的讨论。

本示例的关注点并不是样式和优使性，而是着重于探讨混搭式应用的技术机理。你完全可以在现有styles.css文件以及页面HTML代码的基础上进行大幅改进，让应用具有更好的样式和优使性。

这个样式表还说明了一件重要的事情，就是通过对所要创建的DOM元素进行适当地设计，我们确保了这些元素可以通过CSS选择器进行选定。这样，页面创作者对于元素如何显示和布局就可以控制自如。

styles.css文件列于代码清单13-18。

代码清单13-18　基本的样式表

```
body {
  font-family: Arial,Helvetica,sans-serif;
  padding: 8px;
  margin: 0px;
}

#tripomatic h1,h2 {
  text-align: center;
}

#tripomatic h1 {
  font-size: 1.8em;
}

#tripomatic h2 {
  font-size: 1.1em;
  font-weight: normal;
  padding: 0px 32px;
}

.tripomaticPoiContainer {
  float: left;
  border: 2px ridge maroon;
  background-color: #ffffcc;
  padding: 8px;
  width: 200px;
  height: 480px;
  overflow: auto;
}

.tripomaticPoiContainer li {
  cursor: pointer;
  list-style-type: none;
  margin-bottom: 2px;
  color: maroon;
  text-decoration: underline;
}

.tripomaticPoiContainer ul {
```

```
    margin: 0px;
    padding: 0px;
  }
  .tripomaticMapContainer {
    float: left;
    width: 440px;
    height: 480px;
    border: 2px ridge maroon;
    padding: 8px;
    margin-left: 16px;
    cursor: pointer;
  }

  .tripomaticPoiDescription {
    clear: both;
    margin: 0px 32px;
    text-align: center;
    padding: 16px 32px;
  }

  .tripomaticThumbsContainer {
    clear: both;
    border: 1px solid black;
    padding: 8px;
    height: 100px;
    overflow: auto;
    left: 8px;
    right: 8px;
  }
```

13.6 总结

在本章中我们看到，混搭并非是一堆汽车撞到一起的结果，而是一种将多个源头获取的内容组合在一起的Web页面或应用。我们所完成的小示例，将Yahoo! Maps和Flickr Photo Services的内容组合到一起，它证明了Ajax与开放API结合之后的强大威力，能让我们很容易创造出这样的混搭式应用。

我们创建的应用也是一个自包含的JavaScript类，不需要在使用它的HTML页面里加入额外的代码（当然，创建类的实例所必需的代码除外）。所以我们创建的既是一个应用也是一个组件（component）。事实上，因为在这个应用/组件中我们没有使用任何全局变量或全局的元素ID，所以甚至可以在一个页面中容纳它的多个实例。尽管对于我们的这个特定组件来说，多个实例没有什么意义，但是你可以在自己的组件中应用本章中所使用的技巧，这样就可以在每个页面中多次使用同一个组件。

本示例结合运用了本书前后所探讨过的大量技巧。我们信奉面向对象技术、发挥Prototype库的功能、发起一个又一个Ajax请求、通过跨服务器代理调用可用的开放API，并且对大量的事件进行处理。

此外，你再次看到了Prototype程序库中的各个类和函数是如何让你的JavaScript代码更加简

练、更加模块化、更加有条理的。一个例子就是我们在Trip-o-matic应用中大量地使用了Prototype给JavaScript数组所添加的each()方法。将数组元素的处理过程从传统的for循环中移出并模块化为以单个数组元素为参数的迭代函数，不仅将元素的处理代码分离到了一个简洁明了易于理解的函数中，而且也有助于代码重用——循环以外的代码也可以传入适当类型的对象来直接调用迭代函数。

我们再次使用了Yahoo! Maps API来显示地图。Yahoo! Map API还有许多其他功能，利用这些功能，你甚至可以让你的地图应用变得更具有交互性。

你还进一步学习了Flickr的照片服务，特别是如何使用照片集来对照片进行分组。不过我们的讨论又一次仅仅只是触及了所有可用功能的冰山一角。那些想使用交互式照片的站点和应用可以利用Flickr完成许多功能，远远超出我们这里探索过的范围。

所以，请看看在Web上有哪些内容可以通过开放API为你所用。肯定有一个组合能引起你的共鸣，激发你的创意——创造一个你自己的混搭吧！

最新插播新闻！①

最近有一个叫做Yahoo! Pipes（http://pipes.yahoo.com）的新的Yahoo!服务上线了，它可算是说明混搭威力的最精华（至少到目前为止）的例子！它带有一个基于Web的可视化的混搭生成器，可以通过拖放来组合程序原语、默认数据源（Google Base、Yahoo!，等等）以及许多其他东西。试试看它吧！

① 当然读者看到本书译本时肯定不能算新闻了。类似Yahoo! Pipes的服务现在还有Google混搭编辑器（Google Mashup Editor）、微软的Popfly等。——译者注

TURING 图灵程序设计丛书 Web开发系列

Ajax in Practice

Ajax实战

实例详解

[英] Dave Crane
[美] Bear Bibeault 等著
[美] Jord Sonneveld

贺师俊 许超 金鑫 等译
贺师俊 审校

人民邮电出版社
北京

TURING 图灵程序设计丛书 Web开发系列

MANNING

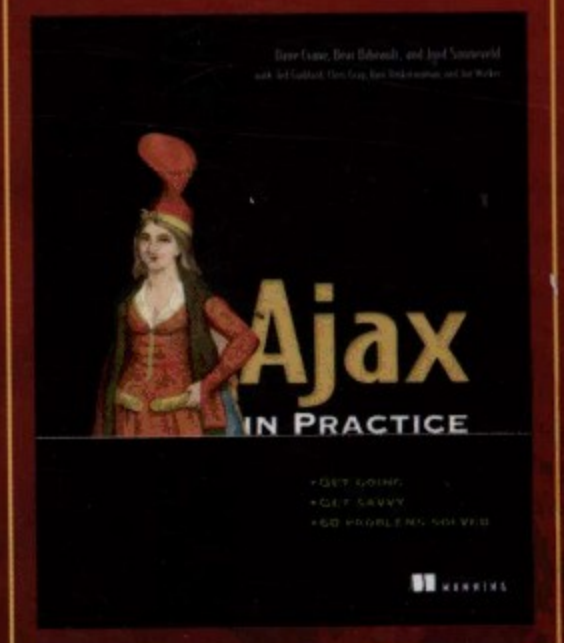
《Ajax实战》三部曲
王者归来

Ajax in Practice

Ajax实战

实例详解

[英] Dave Crane
[美] Bear Bibeault 等著
[美] Jord Sonneveld
贺师俊 许超 金犊 等译
贺师俊 审校



- 七位世界级Web专家巨献
- 引领你进入Ajax 2.0时代
- 大量Ajax/JavaScript核心技巧和最佳实践

人民邮电出版社
POSTS & TELECOM PRESS

Ajax in Practice

Ajax实战——实例详解

“通过本书中的示例，我掌握了大量关于Ajax的核心技巧和高级JavaScript技术。绝对必读的Ajax著作！”

——Javaranch.com

“我太喜欢这本书了。具体、实用，可以立刻付诸实践。我强烈推荐！”

——Amazon.com

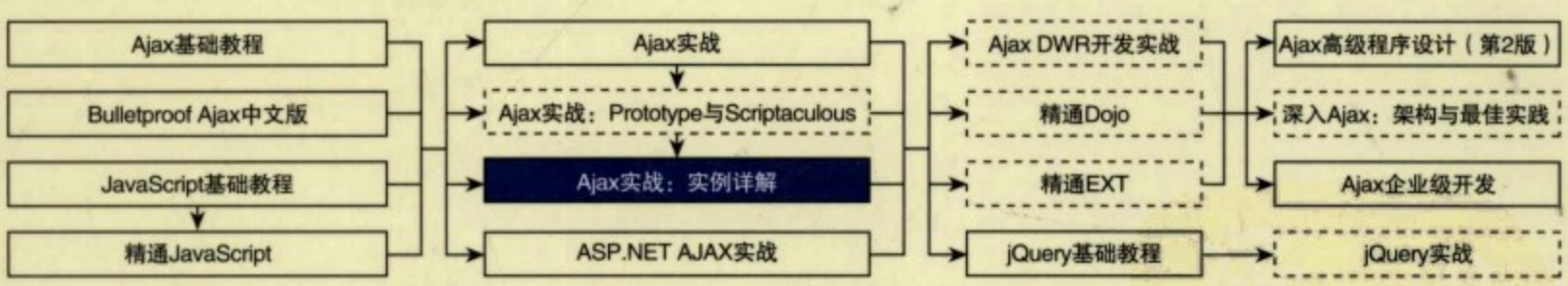
Ajax赋予了Web程序员开创未来的潜力，但是，Ajax应用程序的复杂性和开发难度也大大增加——如此空前的挑战，我们应该如何应对？

本书是Ajax大师Dave Crane继巨著《Ajax实战》之后，与数位顶尖高手联袂推出的又一部Ajax力作。书中直指Ajax/JavaScript应用程序开发中急需解决的各种具体问题，通过大量经典的可重用代码，以Cookbook的形式深入讨论和分析了Ajax/JavaScript开发涉及的最关键的实战技术，包括JSON格式和Dojo、Prototype、DWR、jQuery等框架，还有事件处理、表单验证、内容导航、状态管理、拖放等任务的实现，以及各种Open API的应用。同时，手把手教会读者如何将各种技术运用到实际应用中，从而创建强大的应用解决方案。

本书作者均为Ajax和Web开发领域的世界级专家，有丰富的Ajax实战经验，阵容极尽豪华。

Dave Crane是著名的Ajax权威，领衔撰写了《Ajax实战》三部曲。诸位合著者中，**Bear Bibeault**是著名技术社区JavaRanch的核心人物之一，也是名著《jQuery实战》的第一作者；**Jord Sonneveld**是Google的资深Web工程师；**Chris Gray**、**Ram Venkataraman**和**Ted Goddard**分别是ClearNova、JBoss和IceFaces等著名开源技术公司的资深Web工程师；**Joe Walker**则是DWR框架之父。

图灵Ajax/JavaScript类图书阅读路线图



MANNING

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线： (010)88593802
反馈/投稿/推荐信箱： contact@turingbook.com

分类建议 计算机/网络开发/程序设计

人民邮电出版社网址 www.ptpress.com.cn

ISBN 978-7-115-18765-9

ISBN 978-7-115-18765-9/TP

定价：59.00 元

版 权 声 明

前　言

Web始终都是催生创新的温床，在它不长的历史里，以某项创造为基础进行重造和重用，以致在某些方面远远超出原始发明者意图的例子比比皆是。例如，CGI取代了一种基于网络的文档检索协议，同时又提供了从后台数据库中取得数据并（根据实时请求）动态生成文档的能力；HTTP首部被用来在无状态协议之上提供持续的用户会话，这使预订系统和电子商务等有状态应用成为可能；在核心协议上创建的加密层，给那些网上商店的顾客和业务应用的用户以信心。

这些突破性的技术永久地改变了人们使用Web的方式。时至今日，诸如服务器端页面、用户会话以及SSL等技术只是构建Web应用的日常构件，并成为所有Web开发人员工具箱里的必备，以至于我们认为这是理所当然的。然而，Web创新的步伐仍然没有停止，几乎每周都会有新的Web框架出现。

Ajax是近年来Web开发领域最大的突破性技术之一。先前的所有创新对Web用户界面（点击、发送请求、响应、重绘页面）的基本模式没有多大影响，直到XMLHttpRequest（XHR）对象1999年悄然出现在IE 5中时，这一状况才结束。该对象的使用最初是为了增强Outlook Web Access客户端程序，并未引起太大的关注。

2005年，当Google举起Ajax的旗帜开发邮件（Gmail）、地图和Suggest等应用时，人们才开始猛然醒悟，关注起Ajax来。来自Adaptive Path公司的Jesse James Garrett提出了“Ajax”这一术语，树起一面大旗，人们云集旗下讨论Ajax究竟是什么，可以用它做什么。

Ajax好像只是在等待一个名字，一旦有了，一系列令人兴奋的活动接踵而至，而人们也开始研究Ajax的技术本质。Ajax引入了一种创建Web应用的全新的方式。尽管这也导致有许多新的问题亟待解决，但随着Web开发社区不断突破新的极限，过去两年爆发了新一轮创新热潮。

沿着这种发展路线，Ajax的基础（例如XMLHttpRequest对象）将重复服务器端页面、用户会话以及SSL的道路。处于集体无意识的Web开发社区犹如神助，一下子明白了Ajax技术的根本，并转向如何解决使用中产生的更广泛的问题。

为解决这些问题，我们决定撰写本书。我们希望本书能帮助熟练的和不那么熟练的Web开发者完全掌握Ajax技术并成功创建其自己的Ajax应用。它可以看作是第二代Ajax图书：第一代Ajax图书介绍Ajax是什么，而第二代Ajax图书介绍可以用它做什么以及如何做。

本书从Manning出版社联系Steve Benfield并希望他成为第二代Ajax图书的主编开始启动，可以看作是Dave Crane所著畅销书《Ajax实战》的后续之作。后来，Steve因故不再担任主编，于是Jord Sonneveld、Bear Bibeault和Dave Crane携手为你奉献了本书。

当你看完前言，我们就完成了自己的任务，因此可以坐下来分享几杯早该享用的饮料。我们希望你在阅读本书时能获得和我们写作本书时一样多的乐趣！

关于本书

Ajax席卷了Web开发社区，它使Web开发人员得以创建以客户为中心的富因特网应用。不过它也给这些应用带来了新的复杂性和多样性。本书抓住Ajax的核心内容，提供了大量实践性技巧和可重用的代码，以帮助开发者解决创建Ajax解决方案时遇到的具体问题。

简要介绍Ajax之后，本书将带领读者领略几十个易于使用的、以解决方案为重点的示例。读者可以学会如何实现富客户端用户界面，包括拖拽实践性策略、有效导航、事件处理、表单项验证、状态管理、如何选择Ajax库、访问第三方的开放WebAPI等实用策略。

与传统的“cookbook”类图书不同，本书提供对各个技巧的深入讨论并演示如何将这些独立的组件连接起来，以创建强大的应用解决方案。本书结尾一章是令人愉快的“混搭(mashup)”，之所以选择该示例，是因为它有趣、好玩，更重要的是它很实用。

本书将帮助你：

- 超越Ajax本身并学习如何让Ajax运行起来；
- 掌握许多用户界面设计和站点导航的技巧；
- 动手实践专业级的可重用的Ajax代码以解决实际问题。

读者对象

本书针对那些希望借助Ajax技术创建最佳富用户界面应用的Web开发人员。

Ajax初学者会发现入门的前两章对快速了解异步请求的知识有帮助，但本书面向的主要读者是开发人员，他们应当至少有基本的Web应用开发背景并能使用基础的JavaScript语法实现一些客户端特效。

在前所未有的富客户端用户界面应用里，客户端代码数量迅速增多，因此这部分代码应和服务端代码同样得到重视。本书介绍了一些高级的JavaScript技巧帮助你组织客户端代码并有效使用Ajax。

如果你不仅对使用新技术扩展自身编码能力感兴趣，同时也关注如何应用编程技巧和模式来最好地利用这些技术，我们认为本书能满足你的这些需求。

无论你是老练的客户端开发人员，还是刚开始创建拥有富用户界面的新手，我们都希望本书对你能有所帮助。

阅读路线图

本书分为两部分。第一部分：“Ajax基础”，包含4章导读性的内容，以确保你在学习本书第二部分时已经消化掌握了这些技巧。第二部分“Ajax最佳实践”，每章都讲解了客户端编程的各种实践性主题。它们或强调直接使用Ajax，或强调在支持Ajax的应用中运行良好的实践和原则。

第1章深入探讨Ajax与其他技术的区别，并介绍为何有如此多的内容需要学习。本章提供了一个快速教程介绍如何跨浏览器使用Ajax以及如何处理到来的响应。最后介绍Prototype库如何确

保整个过程更加流畅。

第2章讨论了Ajax通信的各种方式，包括JSON、XML和XSLT。我们还研究了Ajax与SOAP Web服务的结合使用。

第3章介绍了怎样使用面向对象的JavaScript来控制典型的Ajax应用都具备的客户端源代码增长问题。我们介绍的主要概念有对象构造、函数是一等对象、函数是类方法、函数上下文以及闭包等，并在面向对象技术的背景下加以介绍。最后介绍了如何使用Prototype库帮助我们轻松定义JavaScript类。

第4章继续讨论支持Ajax的JavaScript库并进一步讲解Prototype、Dojo工具箱、jQuery和DWR库，虽然不可能完全地介绍这些库具备的各种特性，但我们特别介绍了它们给Ajax带来的变化。我们还会在接下来的章节的多个代码示例中看到这些库的实践用法。

第5章讲解事件处理，介绍了多种事件模型并特别强调了跨浏览器问题，并介绍了使用Prototype库以减轻跨浏览器带来的痛苦。还讨论了Ajax应用程序中最常用的事件类型。

第6章详细研究了表单数据项验证及其与上一章介绍的事件处理的联系。本章示例采用Prototype和jQuery库以获得最大好处。这些示例演示了如何截取表单提交（以前通常会引起整个页面刷新的操作）并把它重定向为不甚唐突的Ajax请求。

第7章讨论内容导航。讲解了创建简单的菜单，然后进入更加复杂的导航辅助设施如树视图、accordion控件、Tab视图和工具条等内容。我们还在本章给出了支持这些功能的OpenRico库和qooxdoo库的相关代码。

第8章关注用户在浏览器中点击后退和刷新按钮导致的问题。我们会从两个角度介绍：如何避免用户出现这些问题以及如何支持后退和刷新操作。本章还会介绍如何为应用程序添加一个简单的撤销功能。

第9章讨论拖放。我们将研究拖放操作的原理并讨论支持拖放的JavaScript库。我们会介绍如何使用Scriptaculous实现支持项目复制的列表，并用Scriptaculous和ICEfaces实现一个简单的购物车。

第10章讨论关于可用性的考虑，并介绍Ajax怎样帮助我们解决或至少减轻网络延时的问题。讨论了通过主动提供由服务器端协助完成的帮助减少用户的挫折感，并再次回顾了表单数据验证。我们还解释了在富用户界面中如何处理多控件Tab键次序以及多层控件次序问题。

第11章介绍状态管理。我们将探索如何维护客户状态、缓存数据、预加载数据和如何持久化客户状态。我们还讨论了使用AMASS库持久化大量数据。

第12章探讨第三方开放的API的用法。我们学习如何避免令人畏惧的“Ajax安全沙箱”使Ajax请求到达远程服务器，然后使用该技巧访问第三方开放的API，例如Yahoo!Maps、Geocoding和Traffic、Google搜索引擎以及Flickr照片服务等。

第13章以一个完整的混搭式应用结束，它使用了上一章介绍的第三方开放API以及本书介绍的各种技巧来创建一个完整的且可以运行的混搭式应用。

代码约定

所有源代码清单或正文中的源代码都使用一种等宽字体（例如`like this`）以区别于普通

文本。正文中的方法和函数名、对象属性、XML元素都用此类字体显示。

多数情况下，对源代码重新编排了格式。为适应本书的页面宽度，我们增加了换行并调整了缩进。在少数情况下，这样做还不够，源代码清单里还包括续行记号。另外，还从源代码清单中去掉了许多注释。

许多源代码清单都伴有代码注解，对重要概念加以说明。在一些情况下，我们加了编号，对应到代码后面的注解文字。

代码下载

本书所有示例的源代码都可以从图灵公司网站上本书页面下载下载。也可以从http://www.manning.com/crane2或http://www.manning.com/AjaxinPractice下载。

作者在线

购买本书也就意味着你可以自由访问一个由Manning出版社运营的私有网上论坛。你可以在该论坛上对本书发表评论、提一些技术问题并从作者或其他读者那里获得帮助。要访问此论坛并订阅其内容，请访问http://www.manning.com/crane2或http://www.manning.com/AjaxinPractice。该网页提供了注册之后如何访问论坛的说明，也介绍了你可以获得什么帮助，并且公布了论坛的规则。

Manning出版社对读者承诺提供这样一个场所，让读者与读者之间以及读者与作者之间可以进行有意义的对话。但对作者的参与程度并不做承诺，因为作者的参与和贡献完全是自愿的（而且是无偿的）。我们建议你向作者提一些有挑战性的问题，以免作者失去兴趣。

只要本书不绝版，“作者在线”论坛和以前讨论的存档就可以从出版社的网站上访问。

封面图片

本书封面上的插图是一位苏丹女眷，苏丹家族的一位女性成员，苏丹的妻子和母亲可能都叫这个名字。插图来自一本土耳其奥斯曼帝国的服饰画册，由伦敦Old Bond街的William Miller于1802年1月1日出版。画册的扉页已经丢失，因此我们很难推断准确的创作时间。此书的目录同时使用英语和法语表示插图，每张图片都有创作它的两位艺术家的名字，他们一定会为其能美化两百年后的一本计算机编程图书的封面而感到惊讶。

Manning出版社的一个编辑在位于曼哈顿西26街“Garage”的古董跳蚤市场买到了这本画册。卖主是住在土耳其安卡拉的一个美国人，交易时间是在那天他准备收摊的时候。这位编辑没带够买这本画册所需的现金，并且卖主礼貌地拒绝了他使用信用卡和支票。卖主当天晚上要飞回安卡拉，看起来好像没什么希望了。该怎么解决呢？最后通过握手来约定的老式的君子协议解决了。卖主提议通过银行转账付款，编辑在纸上抄下了收款银行的信息，随后画册就到他手里了。不用说，第二天我们就把款付给了卖主。我们感谢这位陌生人能如此信任我们的同事。这让我们回忆起了那个很久以前的美好时代。

来自奥斯曼帝国画册的图片，就像在丛书封面上出现的其他插图一样，将两个世纪之前的服饰习俗的丰富性和多样性带到我们身边。这些图片令我们回忆起那个时代的隔绝和距离，以及除

了这个通信极度发达的时代之外的其他每一个历史时代。

从那以后，服饰习俗所代表的不同区域间的差异已经改变了，那个时期的丰富多彩的服饰文化也逐渐褪色。如今已经很难区分两个不同地区的居民了。也许我们应该尽量乐观地看待这些变化，文化上、视觉上的差异已经转变为更加多样化的个人生活上的差异，或者更加多样化和更加有趣的智力上和技术生活上的差异。

我们使用来自这本画册的图片作为本书的封面，这为我们带来了两个世纪之前的丰富的、多样性的区域生活。Manning出版社使用这种方式来赞美计算机行业中的创新性和主动性，当然，还包括其中的乐趣。

Dave、Bear和Jord

致　谢

一本书的致谢部分总是会包含长长的名单，因为没有这些人的共同努力，是不可能有你手中的这本书的。身为作者，我们对此最有体会！尽管在键盘上度过的漫漫长夜常让人有独行侠的错觉，但我们明白，我们并不是一个人在战斗。

这一路上的每一步，Manning出版社的出版人和编辑们都与我们并肩前行，以确保本书能做到尽善尽美。我们要感谢他们的鼓励，感谢他们对质量的坚持以及对细节的关注。我们在此感谢出版人Marjan Bace和我们的编辑Mike Stephens，还有许多人工作在幕后，在此一并谢过，他们是：Karen Tegtmayer、Howard Jones、Liz Welch、Dottie Marsico、Katie Tennant、Mary Piergies、Gabriel Dobrescu、Ron Tomich以及Olivia DiFeterici。

在本书的写作和成书过程中，我们的审稿人贡献良多，小到代码中的错误、行文中的别字，大到各个章节的编排和组织，都给予了我们极有价值的反馈。手稿历经多次审阅，而每经一次审阅，本书就更上一层楼。我们衷心感谢他们能在百忙之中抽出时间阅读我们的手稿，他们是：Curt Christianson、Anil Radhakrishna、Robert W. Anderson、Srinivas Nallapati、Ernest Friedman-Hill、Jeff Cunningham、Christopher Haupt、Bas Vodde、Bill Fly、Ryan Lowe、Aleksey Nudelman、Lucas Carlson、Derek Lakin、Jonas Trindler、Eric Pascarello、Joel Webber、Jonathon Esterhazy以及Benjamin Gorlick。

我们要特别感谢本书的技术编辑Valentin Crettaz，是他检查了代码，并一遍又一遍地阅读这些章节，直至定稿。我们非常感激他的付出。

最后，要感谢Ted Goddard、Chris Gray、Ram Venkataraman以及Joe Walker，作为各自领域的专家，他们参与编写了本书的相关主题，能与他们合作是我们的荣幸。

Dave Crane

我要感谢Historic Futures公司的同事们：Simon Warrick、Tim Wilson、Susannah Ellis、Simon Crossley、Rob Levine和Miles Wilson，感谢他们对我这次写作的支持。感谢Wendy、Nic、Graeme以及Skillsmatter.com团队——还有我那些富有天分的学生们——在他们的帮助下，本书的写作思路才得以成形。最后，我要感谢我的家人Chia、Ben还有Sophie，没有你们的包容，我不可能并行写出两部书稿[①]；感谢我的老爸老妈；还要感谢我的几只小猫，在每个写作的深夜，你们都充

① 在写作本书的同时，Dave Crane和Bear Bibeault也参与了*Prototype and Scriptaculous in Action*一书的编写、该书中文版即将由人民邮电出版社出版。——译者注

分地利用了我的笔记本电脑排放出来的暖气。

Bear Bibeault

有太多的人需要致以谢意。我要感谢我在javaranch.com的所有朋友与伙伴，包括（但不限于）Ernest Friedman-Hill、Ben Souther、Max Habibi、Mark Herschberg、Kathy Sierra……感谢你们在我表现出写作兴趣时给予我热情的鼓励。

我要特别感谢javaranch.com的老板Paul Wheaton创建了这么棒的网站并给予员工充分的信任。感谢Eric Pascarello将我推荐给Manning。

我要感谢我的小狗Gizmo、Cozmo和Little Bear，在我写作这些文字和代码时，他们躺在我的脚边陪伴着我。特别是Cozmo，在我打字时它也踩着笔记本键盘“贡献”了不少字母。我要替编辑们说声感谢上帝。

我还要感谢我的伙伴Jay，谢谢你陪伴我度过同时赶两本书的那些日子，忍受我对Internet Explorer和Word的所有咆哮，并且自始至终给予我鼓励和支持。

Jord Sonneveld

我要特别感谢我的合著者Dave和Bear，他们在写作的后期承担了大量工作。如果没有他们的辛苦努力，这本书就无法面世。

感谢我的爸爸、妈妈、爷爷、奶奶，谢谢你们给我买了第一台电脑，谢谢你们在我埋首著书时给予我的所有支持。

最后，我要感谢我的猫咪，是它们把我的UPS搞短路；还有Mallory，我那了不起的研究UNIX的女友，谢谢你。