

Aj ax 高级程序设计



第 1 章 什么是 Aj ax

- [1.1 Aj ax 的诞生](#)
- [1.2 Web 的演化过程](#)
- [1.3 真正的 Aj ax](#)
- [1.4 Aj ax 原则](#)
- [1.5 Aj ax 后面的技术](#)
- [1.6 谁在使用 Aj ax?](#)
- [1.7 混乱与争议](#)
- [1.8 小结](#)

第 2 章 Aj ax 基础

- [2.1 HTTP 基础](#)
- [2.2 Aj ax 通讯技术](#)
 - [2.2.2 XMLHttpRequest 请求](#)
- [2.3 进一步考虑](#)
- [2.4 小结](#)

第 3 章 Aj ax 模式

- [3.1 通信控制模式](#)
 - [3.1.3 提交节流](#)
 - [3.1.4 表单增量验证的实例](#)

- [3.1.5 字段增量验证实例](#)
- [3.1.6 定期刷新](#)
- [3.1.7 新评论提示实例](#)
- [3.1.8 多阶段下载](#)
- [3.1.9 附加信息链接实例](#)
- [3.2 失效处理模式](#)
- [3.2.1 取消待处理的请求](#)
- [3.2.2 重试](#)
- [3.3 小结](#)

第4章 XML、Xpath 和 XSLT

- [4.1 浏览器对 XML 的支持](#)
- [4.1.1 IE 中的 XML DOM](#)
- [4.1.2 Firefox 中的 XML DOM](#)
- [4.1.3 跨浏览器兼容的 XML](#)
- [4.1.4 基本的 XML 实例](#)
- [4.2 浏览器对 Xpath 的支持](#)
- [4.2.1 XPath 概述](#)
- [4.2.2 IE 中的 XPath](#)
- [4.2.3 使用命名空间](#)
- [4.2.4 Firefox 中的 XPath](#)
- [4.2.5 使用命名空间解析器](#)
- [4.2.6 跨浏览器兼容的 XPath](#)
- [4.3 浏览器对 XSLT 的支持](#)
- [4.3.1 XSLT 概述](#)
- [4.3.2 IE 中的 XSLT](#)
- [4.3.3 Firefox 中的 XSLT](#)
- [4.3.4 跨浏览器兼容 XSLT](#)
- [4.3.5 重访“最佳选择”](#)

- [4.4 小结](#)

第5章 基于 RSS/Atom 的 Syndication

- [5.1 RSS](#)
 - [5.1.1 RSS 0.91](#)
 - [5.1.2 RSS 1.0](#)
 - [5.1.3 RSS 2.0](#)
- [5.2 Atom](#)
- [5.3 FooReader.NET](#)
 - [5.3.1 客户端组件](#)
 - [5.3.2 服务器端组件](#)
 - [5.3.3 将客户端和服务端连接起来](#)
- [5.4 安装](#)
- [5.5 测试](#)
- [5.6 小结](#)

第6章 Web 服务

- [6.1 相关技术](#)
 - [6.1.1 SOAP](#)
 - [6.1.2 WSDL](#)
 - [6.1.3 REST](#)
- [6.2 .NET 连接](#)
- [6.3 设计决策](#)
- [6.4 创建 Windows 平台的 Web 服务](#)
 - [6.4.1 系统需求](#)
 - [6.4.2 配置 IIS](#)
 - [6.4.3 编写 Web 服务](#)
 - [6.4.4 创建程序集](#)

- [6.5 Web 服务和 Ajax](#)
- [6.5.1 创建测试工具](#)
- [6.5.2 IE 使用的方法](#)
- [6.5.3 Mozilla 使用的方法](#)
- [6.5.4 通用方法](#)
- [6.6 跨域的 Web 服务](#)
- [6.6.1 Google Web API 服务](#)
- [6.6.2 创建代理](#)
- [6.7 小结](#)

什么是 Ajax

1.1 Ajax 的诞生

- 2005 年 2 月，Adaptive Path 公司的 Jesse James Garrett 在网上发表了一篇名为《Ajax: 一种 Web 应用程序开发的新方法》的文章（现在还可以在 www.adaptivepath.com/publications/essays/archives/000385.php 看到）。在这篇文章中，Garrett 阐述了他为什么认为 Web 应用程序正在填平与传统桌面应用程序之间的鸿沟。他引用了一些新的技术，并以几个 Google 的项目作为例子，说明了如何将传统的、基于桌面应用程序的用户交互模型应用到 Web 上。然后他说出了两句引起人们大量兴趣、兴奋和争论的话：

- Google Suggest 和 Google Maps 就是这种新型 Web 应用程序的两个例子，在 Adaptive Path 公司里，我们将这种理念称为 Ajax。这是 Asynchronous（异步）JavaScript + XML 的简写，它预示着 Web 可能将发生一次根本性的变革。

- 从此之后，关于 Ajax 的文章、示例代码以及争议有如潮水一般充斥于整个因特网上。开发人员在 Blog 上谈到它，技术杂志关注它，而许多公司则在产品中应用它。但要理解到底什么是 Ajax，还必须先了解促使其产生的一些 Web 技术的演化过程。

1.2 Web 的演化过程

- 当 Tim Berners-Lee 在 1990 年首次提出 World Wide Web（万维网）时，其概念是相当简单的：使用超文本和 URI（统一资源标识符）来创建一个关联信息的网，它能

够链接来自世界各个地方的各种学术文献，使人们可以立即访问所引用的素材。的确，第一版本的 HTML（超文本标记语言）对于格式化和链接之外的事情关注得很少，它并不适用于构建交互性强的软件，只是一个用来共享最新的各种文字和图表信息的平台。Web 就是从这样的静态页面开始发展的。

- 随着 Web 的发展，商业界很快就发现了它在向大众发布产品及服务信息等应用上所具有的优势。紧接下来的新一代 Web 则着眼于提高信息的格式化和显示能力，而 HTML 也随之发展，以满足这些需求和这些新的媒体意识强烈的用户期望。很快，一家名为 Netscape 的小公司将推动 Web 的发展进程迈出更迅速的一步。

- 1.2.1 JavaScript

- Netscape Navigator 是第一个成功的主流 Web 浏览器，同样也使 Web 技术得以快速发展。但是，Netscape 在标准出台之前（就像现在微软在 IE 的开发中忽视现有标准而遭到批评一样）就开发新技术或对原有技术进行扩展的做法，却经常遭到标准化组织的批评。JavaScript 就是这种技术中的一个。

- JavaScript 原名为 LiveScript，是 Netscape 公司的 Brendan Eich 开发的，包含于 Navigator 浏览器 2.0 版本（发布于 1995 年）之中。开发人员第一次能够控制页面与用户之间的交互。对于诸如数据验证这样的简单任务，不再需要持续地在服务器和客户端之间往返，只需在浏览器中就可以实现。对于大部分因特网用户都还是通过 28.8Kbit/s 的调制解调器实现连接的时代而言，这一能力是十分重要的，因为那时向服务器发送每个请求就像是一个等候游戏。使用户等候响应的次数尽可能地小，这是朝着 Ajax 方法发展的第一个重要步骤。

- 1.2.2 帧

- HTML 的最初版本将每个文档都看作是独立的，直到 HTML 4.0 版，帧（frame）还没有正式引入。帧的理念是使一个网页能够分成几个独立的文档，由于 Netscape 在 HTML 4.0 还没有完成时就实现了该功能，因而引发了争议。Netscape Navigator 2.0 是第一个同时支持帧和 JavaScript 的浏览器。这是 Ajax 演化进程中的一个重要步骤。

- 20 世纪 90 年代末，当微软和 Netscape 之间的浏览器之战爆发之后，JavaScript 和帧都被纳入了正式的标准之中。随着它们功能的不断增加，富于创新的开发人员开始尝试将它们集成在一起使用。由于帧表示的是一个完全独立的对服务器的请求，JavaScript 能够控制帧及其内容的能力，使实现某些令人兴奋的效果成为了可能。

- 1.2.3 隐藏帧技术

- 当开发人员掌握操作帧的方法之后，也就为客户端—服务器通信引入了一种有效的工具。隐藏帧技术是指配置一个帧集（frameset），使其中一个帧的宽度或高度为 0 像素，其唯一的功能就是用来初始化与服务器的通信。隐藏帧可以包含一个 HTML 表单，表单中包含一些特定的字段，这些字段能够通过 JavaScript 实现动态填充，并将其发回到服务器端。当返回该帧时，将会调用另一个 JavaScript 函数，以提示数据已经返回了。隐藏帧技术是 Web 应用系统中第一个异步请求/响应模型。

- 然而这只是第一个 Ajax 通信模型，另一个技术进展已经不远了。

- 1.2.4 动态 HTML 和 DOM

- 直到 1996 年左右，Web 的主流还是静态页面。尽管 JavaScript 和隐藏帧技术能使用户交互更具活力，但除了重载页面之外仍然没有改变页面内容显示的方法。紧接着，IE 4.0 发布了。

- 在此时，IE 引入了动态 HTML（DHTML）技术，凭借该技术 IE 成功地赶上了市场的领导者 Netscape Navigator，甚至还占了上风。尽管还在开发阶段，但 DHTML 还是代表了从静态网页向前迈出的重要一步，它使得开发人员能够通过 JavaScript 来修改已载入页面的任何部分。随着 CSS（层叠样式表）的出现，DHTML 使 Web 开发重现活力，尽管早期微软和 Netscape 所遵循的路线有很大的不同。开发社区感到振奋是可以理解的，因为将 DHTML 和隐藏帧技术组合在一起，就意味着可以随时根据服务器的信息来更新页面的任何部分，这才是 Web 开发的一次真正的范型转变（paradigm shift）。

- 但 DHTML 却从来没有被纳入标准，尽管微软的影响随着 DOM（文档对象模型）成为标准工作的中心变得更加强大。与 DHTML 只追求修改网页的某个片段不同，DOM 有一个更雄心勃勃的目标：为整个网页提供一个标准结构，对该结构的操作将使修改 DHTML 风格的页面成为可能。这是向 Ajax 方法发展的下一个重要步骤。

- 1.2.5 iframe

- 尽管隐藏帧技术流行得令人难以置信，但它仍然存在一个不足：必须提前计划，为可预见的隐藏帧设置帧集。在 1997 年，<iframe/>元素作为 HTML 4.0 官方标准的一部分引入，这是 Web 进化过程中另一个重要步骤。

- 开发人员可以在页面的任何地方放置 iframe，而不必定义帧集。它可以使开发人员把前面提到的帧集完全抛之脑后，只需简单地在页面中放置隐形的 iframe（通过使用 CSS），以完成客户端到服务器的通信。当 DOM 最后在 IE 5.0 和 Netscape 6.0 中实现时，还能够动态地在运行时创建 iframe，也就是意味着 JavaScript 函数能够创建 iframe，

发出请求，获取响应，并且在页面中不需任何额外的 HTML 元素。这就是新一代的隐藏帧技术：隐藏 i frame 技术。

- 1.2.6 XMLHttpRequest

- 微软的浏览器开发人员肯定了解到隐藏帧技术和新的隐藏 i frame 技术的广为流行，因为他们决定向开发人员提供一个实现客户端—服务器交互的更好的工具。这个名为 XMLHttpRequest 的工具是在 2001 年以 ActiveX 对象的形式引入的。

- 微软的 JavaScript 扩展可以用来创建 ActiveX 控件这种微软专有的程序对象。当微软通过一个名为 MSXML 的库来提供 XML 支持时，就引入了 XMLHttpRequest 对象。虽然名字中有 XML，但是这个对象更像是操作 XML 数据的另一种方法。实际上，它更像是一个能够在 JavaScript 中进行控制的特定 HTTP 请求。开发人员可以像处理其他从服务器端返回的数据一样，访问其 HTTP 状态代码和首部信息。这些数据可能以 XML 格式组织，也可能预格式化为 HTML、序列化为 JavaScript 对象或者采用开发人员预想的其他格式。现在可以独立于页面的载入/重载周期，使用纯 JavaScript 通过程序访问服务器，而不再需要使用隐藏帧或隐藏 i frame 技术。XMLHttpRequest 对象对 IE 开发人员而言有着巨大的影响。

- 随着使用者不断增加，开源项目 Mozilla 的开发人员也开始移植 XMLHttpRequest。为了避免使用 ActiveX，Mozilla 开发者将 XMLHttpRequest 对象的主要方法和属性都复制到自己浏览器的 XMLHttpRequest 对象中。随着所有主流浏览器都对某种形式的 XMLHttpRequest 提供支持，Ajax 风格的界面开发迅速流行起来，诸如 Opera、Safari 的边缘浏览器软件也不得不支持某种形式的 XMLHttpRequest（均模仿 Mozilla，选择在浏览器中实现 XMLHttpRequest 对象）。

• 1.3 真正的 Ajax

- 尽管在 Garrett 的文章最后加了一些经常被问到的问题，但对于“Ajax 到底是什么”仍然存在一些争议。简单地说，Ajax 只不过是一种 Web 交互的方法。这种方法只是在客户端和服务器间传输少量的信息，从而给用户提供更及时的体验。

- 在传统的 Web 应用程序模型中，浏览器本身负责初始化向服务器的请求，以及处理来自服务器的响应，而 Ajax 模型不同，它提供了一个中间层（Garrett 称之为 Ajax 引擎）来处理这种通信。Ajax 引擎（Ajax engine）实际上只是一个 JavaScript 对象或函数，只有当信息必须从服务器上获得的时候才调用它。与传统的模型不同，不再需要为其他资源（诸如其他网页）提供链接，而是当需要调度和执行这些请求时，向 Ajax

引擎发出一个函数调用。这些请求都是异步完成的，也就意味着不必等收到响应之后就可以继续执行后续的代码。

- 服务器（传统模式中，它是提供 HTML、图像、CSS 或 JavaScript）将配置为向 Ajax 引擎返回其可用的数据，这些数据可以是纯文本、XML 或者需要的任何格式，唯一的要求就是 Ajax 引擎能够理解和翻译这种数据。

- 当 Ajax 引擎收到服务器响应时，将会触发一些操作，通常是完成数据解析，以及基于其所提供的数据对用户界面做一些修改。由于这个过程中传送的信息比传统的 Web 应用程序模型少得多，因此用户界面的更新速度将更快，用户也就能够更快地进行他们的工作。图 1-1 是在 Garrett 文章中原图的基础上进行修改的，它说明了传统 Web 应用程序模型和 Ajax 模型之间的区别。

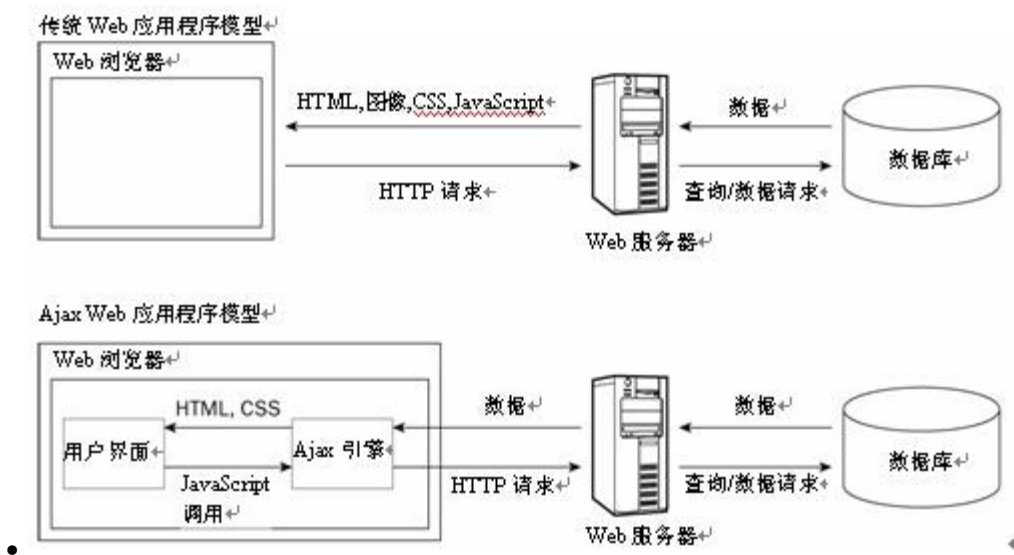


图 1-1

1.4 Ajax 原则

- 作为一种新的 Web 应用程序模型，Ajax 仍处于幼年时期。不过，一些 Web 开发人员却已将这种新的开发方法视为一个挑战。其挑战在于定义什么样的应用程序是好的 Ajax Web 应用程序，什么样的是不好的或平庸的。软件开发及可用性专家 Michael Mahemoff (<http://mahemoff.com>) 指出了一个好的 Ajax 应用程序应遵循的如下关键原则，它很有价值：

- q 尽量减少通信量：Aj ax 应用程序向服务器发送的信息量及从服务器接收的信息量应尽可能地少。简单地说，Aj ax 应尽量减少客户端和服务端之间的通信流量。确保 Aj ax 应用程序不发送和接收不需要的信息，以增强其可靠性。

- q 不意外：Aj ax 应用程序通常会引入与传统 Web 应用程序不同的用户交互模式。与 Web 标准的“点击—等待”模型相反，一些 Aj ax 应用程序将使用诸如拖放、双击等其他用户界面风格。不管选择什么样的用户交互模型，一定要确保用户知道下一步该如何操作。

- q 遵循常规：不要在发明用户不熟悉的交互模型上浪费时间。直接参考传统的 Web 应用程序和桌面应用程序，这样可以使学习更快捷。

- q 无干扰：避免采用不必要的干扰性页面元素（诸如循环式动画、闪烁的页面部分）。这些小伎俩将会使用户无法专心于所要完成的工作。

- q 可访问性：考虑谁是主要用户、谁是次要用户，他们通常喜欢如何访问 Aj ax 应用程序。不要闭门造车，将没有预料到的新用户关在门外。你的用户是否会使用老版本的浏览器或特定的软件？确保及早地了解这些并制定相应的计划。

- q 避免下载整个页面：当最初的页面下载之后，所有与服务器的通信都将由 Aj ax 引擎管理。不要一些地方通过 Aj ax 来完成少量数据的下载，而在另外的地方却重新下载整个页面，这将对用户体验造成破坏。

- q 用户第一：以用户为本设计 Aj ax 应用程序比其他任何东西都重要。尽量使常见的使用场景易于实现，而不要过于追求引人注目或很酷的效果。

- 以上这些原则的共同出发点都是可用性。Aj ax 最根本的是要提高用户的 Web 体验，其后面的技术只是完成这一目标的手段而已。只要坚持上述原则，完全可以确信你的 Aj ax 应用程序是有效且可用的。

• 1.5 Aj ax 背后的技术

- Garrett 的文章中提到了几个他认为是 Aj ax 解决方案组成部分的技术。它们包括：

- ☆ HTML/XHTML：主要的内容表示语言；
- ☆ CSS：为 XHTML 提供文本格式定义；
- ☆ DOM：对已载入的页面进行动态更新；
- ☆ XML：数据交换格式；
- ☆ XSLT：将 XML 转换为 XHTML（用 CSS 修饰其样式）；

- ☆ XMLHttp: 主要的通信代理;
- ☆ JavaScript: 用来编写 Ajax 引擎的脚本语言。
- 实际上, 在 Ajax 解决方案中这些技术都是可用的, 不过只有三种是必需的:

HTML/XHTML、DOM 以及 JavaScript。XHTML 显然是显示信息所必需的, 而 DOM 则是为了在不重新载入 XHTML 页面的前提下修改部分内容所必需的, 最后的 JavaScript 则是初始化客户端—服务器通信、操作 DOM 来更新网页所必需的。列表中的其他技术则对于微调 Ajax 解决方案很有用, 但不是必需的。

- 在 Garrett 的文章中忽略了一个很重要的组件——必要的服务器端处理逻辑。前面列出的所有技术都与客户端的 Ajax 引擎直接相关, 但如果没有一个稳定、响应及时的服务器来向引擎发送内容, 也就不会有 Ajax 的存在。为了实现这一目标, 可以使用你所选择的应用服务器。不管你将服务器端组件编写为 PHP 页面、Java servlet 还是 .NET 组件, 都只需要确保向 Ajax 引擎发送的数据格式是正确的。

- 本书中的例子尽可能使用多种服务器端技术, 以便为你提供在不同服务器上实现 Ajax 交互的足够信息。

• 1.6 谁在使用 Ajax

- 有许多商业性网站已经使用 Ajax 技术来改进其用户体验。这些网站和传统的产品手册式的网站相比, 更像是一个 Web 应用, 因为它不再仅用来显示信息, 而是通过访问它来实现一个特定的目标。下面就是一些知名的、运转良好的、使用 Ajax 的 Web 应用程序。

• 1.6.1 Google Suggest

- 当开发人员讨论 Ajax 时, 引用的第一个例子往往是 Google Suggest

(www.google.com/webhp?complete=1), 其界面是 Google 主界面的一个简单克隆, 有一个突出的文本框用来输入搜索关键字。当你在这个文本框中输入内容时, 所有可能相匹配的内容都将显示出来。当你输入时, Google Suggest 会从服务器上获取一些提示, 以下拉列表的形式将你可能感兴趣的搜索关键字都显示出来。而且对于显示出的每个提示都将列出可能匹配的结果总数, 以帮助你做出选择 (参见图 1-2)。



• 图 1-2

• 这个简单的客户端—服务器交互的功能很强大、有效，并且不会让用户感到厌烦。其界面所能做出的反应超出了原来对于一个 Web 应用程序的认识和预期；不管你输入有多快它都将做出相应的更新，就像桌面软件中的自动填充功能一样，可以通过上下箭头来在提示列表中选择任何一项。尽管它仍然还是 beta 版，不过可以肯定这个方法将会应用于 Google 的主页面上。

• 1.6.2 Gmail

• Google 的免费电子邮件服务 Gmail 已被当作 Ajax 时代客户端—服务器交互的奇迹而广为宣传。当你第一次登录 Gmail 时，应用程序所使用的某一个 iframe 将会载入用户界面引擎，以后所有与服务器交互的请求都将由这个用户界面引擎通过 XMLHttpRequest 对象来完成。往返传输的数据将是 JavaScript 代码，浏览器下载之后能够快速执行。这些请求作为对用户界面引擎的指令，指示需要在屏幕上更新的内容。

• 另外，Gmail 应用程序使用几个帧和 iframe 来管理和缓存较大的用户界面变化。如果用帧，要使 Gmail 能够正确地应对后退和前进按钮是一件极其复杂的事，这也是使用帧（或 iframe）或结合 XMLHttpRequest 的好处之一。

• Gmail 最大的胜利在于其可用性。如图 1-3 所示的用户界面相当简单、毫不杂乱，与用户之间的交互和与服务器之间的通信都显得自然、无缝。Google 再次使用 Ajax 来对原本简单的概念进行改进，提供了一种特殊的用户体验。

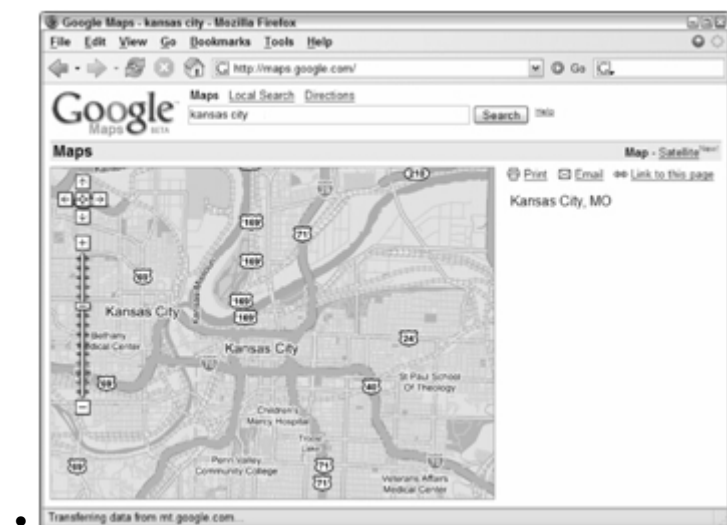


• 图 1-3

• 1.6.3 Google Maps

- Google 最后一个引领潮流的 Ajax Web 应用程序是 Google Maps

(<http://maps.google.com>)。为了与原来已经地位稳固的地图应用网站竞争，Google Maps 通过 Ajax 彻底避免了对主页面的重载（参见图 1-4）。



• 图 1-4

• 与其他地图应用网站不同，Google Maps 可以让你朝不同方向拖动地图。对于 JavaScript 开发人员而言，这些实现拖动效果的代码并没有什么新东西，不过，地图的分块拼接和看似无限制的滚动效果则另当别论。地图被分解成一组图像，它们组合在一起就构成了连续的图像。用来显示地图的图像数量是有限的，如果每次用户移动地图时创建新的图像，那很快会造成内存问题。因此，应将同样的图像反复用于显示地图的不同片段。

• 客户端—服务器通信将通过一个隐藏的 i frame 来完成。只要你搜索或请求一个新的方向，该信息将在该 i frame 中提交并接收响应。返回的数据将以 XML 格式表示，并传给一个 JavaScript 函数（Ajax 引擎）来处理。接着，这个 XML 将以不同的方式使用：一些用来调用正确的地图图像，一些使用 XSLT 转换成 HTML 并显示在主窗体上。其结果就是展示出了一个前景光明的、复杂的 Ajax 应用程序。

• 1.6.4 A9

• Amazon.com 是世界著名的在线商城，几乎销售所有商品。当其发布搜索引擎时，并未引起太大声势和注意。A9 (www.a9.com) 的引入中显示了大大增强的搜索能力，它允许你同时搜索不同类型的信息。它通过 Google 来搜索网站和图像，还可以在 Amazon.com 上搜索图书，在 IMDb（因特网电影数据库）上搜索电影。而且还可以搜索在 2005 年中期发布的 Answers.com 以及黄页和维基百科（Wikipedia）的内容。

• 让 A9 与众不同的是其用户界面的工作方式。当你执行一个搜索时，不同类型的结果将显示在页面上的不同区域中（参见图 1-5）。

• 在搜索结果页面中，你可以使用同一个条件执行其他搜索。当选中与要搜索的类型相应的复选框时，搜索将通过组合隐藏帧技术和 XMLHttpRequest 在后台执行。用户界面将为额外的搜索结果腾出位置，一旦从服务器接收到搜索结果就将其载入到页面中。结果是一个响应迅速的搜索结果页面，在你想搜索不同类型信息时，该页面根本无需重新载入。



• 图 1-5

• 1.6.5 Yahoo! News

• 网站 Yahoo! News (<http://news.yahoo.com>) 也在 2005 年引入了新的设计。新设计最主要的特性是一个令人感兴趣的功能增强：当你将鼠标移到一个特定的标题上时，

将会弹出一个大方框，里面显示出消息的摘要，而且可能还包括一个相关的图片（参见图 1-6）。



图 1-6

- 图片和摘要信息是使用 XMLHttpRequest 从服务器上获得的，然后动态地插入到页面上。

这是一个展示 Ajax 如何用于增强 Web 页面的绝佳例子。Yahoo! News 网站并没有将 Ajax 作为最主要的使用模型，当浏览器没有 Ajax 支持时仍然是可用的；Ajax 函数只是用来在有浏览器支持时增强用户体验的及时响应性。在其之下的是语义正确的 HTML 页面，甚至在不支持 CSS 的浏览器上也能够做出合乎逻辑的布局。

1.6.6 Bitflux Blog

• Bitflux Blog (<http://blog.bitflux.ch/>) 是另外一个将 Ajax 只用于功能增强的好例子，它的主要特性是一个称为 LiveSearch 的技术。LiveSearch 和网站上的搜索框协同工作。当你在搜索框中输入信息时，一组可能的搜索结果就会立即显示在搜索框下面（参见图 1-7）。



图 1-7

- 这些搜索结果是通过 XMLHttp 获取的,并以 HTML 字符串返回,然后插入到页面中。

同样你也可以采用原来的方式来完成搜索:在文本框中输入文字,然后按回车。

LiveSearch 的 Ajax 功能只是为了增加整个网站的功能,但并非所有的搜索都必须使用它。

• 1.7 混淆与争议

- 尽管 Ajax 这一术语是如此流行,但也遇到了许多反对和争议。在 Ajax 浮出水面之前,某些人认为它是一种在 Web 发展道路上的离经叛道。关于语义化 HTML (semantic HTML) 设计、可访问性以及内容与表现分离等提议早已被 Web 开发人员广为接受,而某些人认为 Ajax 的流行已经将这一趋势推向幕后。这些反对者认为, Ajax 鼓励在 JavaScript 中创建表现,因此将像早期的服务器端脚本一样,必会使其陷入混乱。大多数人认为如果越来越多开发人员使用 Ajax 解决方案,则可访问性会受到很大考验。

- 其他人还花了许多时间来分析 Garrett 的文章,并对该文章中所做出的假设进行了批驳。例如,该文章中提到不断使用 XML 和 XMLHttp 是 Ajax 模型的核心,但在他所列出的许多例子中却并没有使用它们。在 Gmail 和 Google Maps 中,这两个技术都没有使用;Google Suggest 也只是使用了 XMLHttp,并使用 JavaScript 数组而非 XML 来进行数据交换。批评者同时也指出在该文章中关于 Ajax 的技术解释容易使人误解,所引用的几项技术不但并非是必需的(诸如 XML 和 XMLHttp),而且还是在许多场合下不可能使用的(例如 XSLT)。

- 针对 Ajax 和 Garrett 在 Adaptive Path 公司网站上发布的文章,还存在另一个很大的质疑,即认为它只是拿新瓶装旧酒。在 Netscape Navigator 2.0 中就可以实现这种类型的数据获取,不过它是直到 2001 年~2002 年才突显出来的,特别是当 Apple Developer Connection 网站上发表了一篇名为《基于 IFRAME 的远程脚本》(位于 <http://developer.apple.com/internet/web-content/iframe.html>) 的文章之后。这篇文章被广泛认为是关于 Ajax 网络方法的第一篇主流文章。当然,术语远程脚本(remote scripting)的确没有像 Ajax 那样获得持久的广泛关注。

- 还有人嘲笑 Ajax 这一术语和 Garrett 的文章,认为与其说是一种创新,还不如说是 Garrett 就职的 Adaptive Path 公司所做的市场噱头而已。为已经存在的技术创造一个名字是狡猾的,显然有着不良意图。虽然围绕着 Ajax 有着这样或那样的争议,但这

个名字非常上口，能使开发人员很快熟悉，并进而想更多地了解它，最终能够以最好的方式来使用它。

• 1.8 小结

- 本章介绍了 Ajax 的基本知识。简单地说，它就是异步 JavaScript+XML，术语 Ajax 是 Jesse James Garrett 发表在 Adaptive Path 公司网站上的一篇文章中首创的。该文章将 Ajax 介绍为一种新的 Web 应用程序用户交互模型，它将不再需要重载整个页面。

- 本章还探究了 Web 技术的演化过程，以及使 Ajax 发展至今的各种技术之间的关系。Ajax 的存在应归功于将 JavaScript 和帧引入到 Web 浏览器之中，以及使用 JavaScript 使异步数据获取成为可能的 Netscape Navigator 2.0。贯穿新 Web 技术的演化过程，许多诸如隐藏帧技术的 Ajax 方法被开发出来，而 iframe 和 XMLHttpRequest 的引入才真正推动了 Ajax 的开发进程。

- 尽管 Ajax 能够用来实现许多事情，但应该用来增加用户的体验而不是实现很酷的效果。本章还讨论了几个 Ajax 原则，所有的原则都可以归结为，在开发 Web 应用程序过程中用户的需求是至高无上的。

- 本章讨论了几个大家熟知的 Ajax 应用程序，包括 Google Suggest、Gmail、Google Maps、Yahoo! News 以及 Bitflux Blog。

- 最后，本章还谈到了关于 Ajax、Garrett 的文章，以及 Web 上有关 Ajax 的争议。有些人认为 Ajax 的流行将会降低可访问性，还有人质疑 Garrett 写这篇著名文章的最初动机。与所有的方法一样，Ajax 最好是用来对设计良好的原有 Web 应用程序进行合乎逻辑的增强。

• 2.1 HTTP 基础

- 要很好地领会 Ajax 技术的关键是了解超文本传输协议（HTTP），该协议用来传输网页、图像以及因特网上在浏览器与服务器间传输的其他类型文件。只要你在浏览器上输入一个 URL，最前面的 http:// 就表示使用 HTTP 来访问指定位置的信息。（大部分浏览器还支持其他一些不同的协议，其中 FTP 就是一个典型例子。）

- 注意：本节中只涉及 HTTP 协议，这是 Ajax 开发人员关心的方面，它可作为 HTTP 的参考手册或指南。

- HTTP 由两部分组成：请求和响应。当你在 Web 浏览器中输入一个 URL 时，浏览器将根据你的要求创建并发送请求，该请求包含所输入的 URL 以及一些与浏览器本身相关

的信息。当服务器收到这个请求时将返回一个响应，该响应包括与该请求相关的信息以及位于指定 URL（如果有的话）的数据。直到浏览器解析该响应并显示出网页（或其他资源）为止。

- 2.1.1 HTTP 请求

- HTTP 请求的格式如下所示：

- <request-line>

- <headers>

- <blank line>

- [<request-body>]

- 在 HTTP 请求中，第一行必须是一个请求行（request line），用来说明请求类型、要访问的资源以及使用的 HTTP 版本。紧接着是一个首部（header）小节，用来说明服务器要使用的附加信息。在首部之后是一个空行，再此之后可以添加任意的其他数据[称之为主体（body）]。

- 在 HTTP 中，定义了大量的请求类型，不过 Ajax 开发人员关心的只有 GET 请求和 POST 请求。只要在 Web 浏览器上输入一个 URL，浏览器就将基于该 URL 向服务器发送一个 GET 请求，以告诉服务器获取并返回什么资源。对于 www.wrox.com 的 GET 请求如下所示：

- GET / HTTP/1.1

- Host: www.wrox.com

- User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)

- Gecko/20050225 Firefox/1.0.1

- Connection: Keep-Alive

- 请求行的第一部分说明了该请求是 GET 请求。该行的第二部分是一个斜杠（/），用来说明请求的是该域名的根目录。该行的最后一部分说明使用的是 HTTP 1.1 版本（另一个可选项是 1.0）。那么请求发到哪里去呢？这就是第二行的内容。

- 第 2 行是请求的第一个首部，HOST。首部 HOST 将指出请求的目的地。结合 HOST 和上一行中的斜杠（/），可以通知服务器请求的是 www.wrox.com/（HTTP 1.1 才需要使用首部 HOST，而原来的 1.0 版本则不需要使用）。第三行中包含的是首部 User-Agent，服务器端和客户端脚本都能够访问它，它是浏览器类型检测逻辑的重要基础。该信息由你使用的浏览器来定义（在本例中是 Firefox 1.0.1），并且在每个请求中将自动发送。

最后一行是首部 Connection，通常将浏览器操作设置为 Keep-Alive（当然也可以设置为其他值，但这已经超出了本书讨论的范围）。注意，在最后一个首部之后有一个空行。即使不存在请求主体，这个空行也是必需的。

- 如果要获取一个诸如 <http://www.wrox.com/books> 的 www.wrox.com 域内的页面，那么该请求可能类似于：

- GET /books/ HTTP/1.1
- Host: www.wrox.com
- User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)
- Gecko/20050225 Firefox/1.0.1
- Connection: Keep-Alive
- 注意只有第一行的内容发生了变化，它只包含 URL 中 www.wrox.com 后面的部分。
- 要发送 GET 请求的参数，则必须将这些额外的信息附在 URL 本身的后面。其格式

类似于：

- URL ? name1=value1&name2=value2&...&nameN=valueN
- 该信息称之为查询字符串 (query string)，它将会复制在 HTTP 请求的请求行中，

如下所示：

- GET /books/?name=Professional Ajax HTTP/1.1
- Host: www.wrox.com
- User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)
- Gecko/20050225 Firefox/1.0.1
- Connection: Keep-Alive

- 注意，为了将文本“Professional Ajax”作为 URL 的参数，需要编码处理其内容，将空格替换成%20，这称为 URL 编码 (URL encoding)，常用于 HTTP 的许多地方 (JavaScript 提供了内建的函数来处理 URL 编码和解码，这些将在本章中的后续部分中说明)。“名称—值” (name—value) 对用 & 隔开。绝大部分的服务器端技术能够自动对请求主体进行解码，并为这些值的访问提供一些逻辑方式。当然，如何使用这些数据还是由服务器决定的。

- 浏览器发送的首部，通常比本节中所讨论的要多得多。为了简单起见，这里的例子尽可能简短。

• 另一方面，POST 请求在请求主体中为服务器提供了一些附加的信息。通常，当填写一个在线表单并提交它时，这些填入的数据将以 POST 请求的方式发送给服务器。

• 以下就是一个典型的 POST 请求：

• POST / HTTP/1.1

• Host: www.wrox.com

• User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)

• Gecko/20050225 Firefox/1.0.1

• Content-Type: application/x-www-form-urlencoded

• Content-Length: 40

• Connection: Keep-Alive

• name=Professional%20Ajax&publisher=Wiley

• 从上面可以发现，POST 请求和 GET 请求之间有一些区别。首先，请求行开始处的 GET 改为了 POST，以表示不同的请求类型。你会发现首部 Host 和 User-Agent 仍然存在，在后面有两个新行。其中首部 Content-Type 说明了请求主体的内容是如何编码的。浏览器始终以 application/x-www-form-urlencoded 的格式编码来传送数据，这是针对简单 URL 编码的 MIME 类型。首部 Content-Length 说明了请求主体的字节数。在首部 Connection 后是一个空行，再后面就是请求主体。与大多数浏览器的 POST 请求一样，这是以简单的“名称—值”对的形式给出的，其中 name 是 Professional Ajax，publisher 是 Wiley。你可以以同样的格式来组织 URL 的查询字符串参数。

• 正如前面所提到的，还有其他的 HTTP 请求类型，它们遵从的基本格式与 GET 请求和 POST 请求相同。下一步我们来看看服务器将对 HTTP 请求发送什么响应。

• 2.1.2 HTTP 响应

• 如下所示，HTTP 响应的格式与请求的格式十分类似：

• <status-line>

• <headers>

• <blank line>

• [<response-body>]

• 正如你所见，在响应中唯一真正的区别在于第一行中用状态信息代替了请求信息。状态行（status line）通过提供一个状态码来说明所请求的资源情况。以下就是一个 HTTP 响应的例子：

- HTTP/1.1 200 OK
- Date: Sat, 31 Dec 2005 23:59:59 GMT
- Content-Type: text/html; charset=ISO-8859-1
- Content-Length: 122
- <html>
- <head>
- <title>Wrox Homepage</title>
- </head>
- <body>
- <!-- body goes here -->
- </body>
- </html>

• 在本例中，状态行给出的 HTTP 状态代码是 200，以及消息 OK。状态行始终包含的是状态码和相应的简短消息，以避免混乱。最常用的状态码有：

- ◆ 200 (OK): 找到了该资源，并且一切正常。
- ◆ 304 (NOT MODIFIED): 该资源在上次请求之后没有任何修改。这通常用于浏览器的缓存机制。
- ◆ 401 (UNAUTHORIZED): 客户端无权访问该资源。这通常会使得浏览器要求用户输入用户名和密码，以登录到服务器。
- ◆ 403 (FORBIDDEN): 客户端未能获得授权。这通常是在 401 之后输入了不正确的用户名或密码。
- ◆ 404 (NOT FOUND): 在指定的位置不存在所申请的资源。

• 在状态行之后是一些首部。通常，服务器会返回一个名为 Data 的首部，用来说明响应生成的日期和时间（服务器通常还会返回一些关于其自身的信息，尽管并非是必需的）。接下来的两个首部大家应该熟悉，就是与 POST 请求中一样的 Content-Type 和 Content-Length。在本例中，首部 Content-Type 指定了 MIME 类型 HTML（text/html），其编码类型是 ISO-8859-1（这是针对美国英语资源的编码标准）。响应主体所包含的就是所请求资源的 HTML 源文件（尽管还可能包含纯文本或其他资源类型的二进制数据）。浏览器将把这些数据显示给用户。

- 注意，这里并没有指明针对该响应的请求类型，不过这对于服务器并不重要。客户端知道每种类型的请求将返回什么类型的数据，并决定如何使用这些数据。

• 2.2 Ajax 通信技术

- 你已经了解了 HTTP 通信的基本方式，现在该是了解网页中实现这种通信的细节的时候了。正如你所了解的，当在网上冲浪时，将在浏览器和服务器之间存在大量的请求。最初，所有的这种请求都是在用户做出需要这一步骤的明显操作时发生的。Ajax 技术将开发人员从等待用户做出这样的操作中解放出来，允许他在任何时间创建一个对服务器的调用。

- 正如在第 1 章所讨论的，Ajax 通信支持许多不同的技术。每一种技术都有自己的优点和缺点，因此了解什么情况使用哪一种技术是很重要的。

• 2.2.1 隐藏帧技术

- 随着 HTML 帧的引入，隐藏帧（hidden frame）技术也应运而生了。该技术后面的基本想法是创建一个帧集，其中包含用于客户端—服务器通信的隐藏帧。可以通过将帧的宽度或高度设置为 0 像素来隐藏一个帧，以使其不显示。尽管一些早期的浏览器（诸如 Netscape 4）不能够完全隐藏帧，经常会留下一些明显的帧边框，但该技术还是广泛地为开发人员所采用。

• 1. 模式

- 隐藏帧技术遵循一种特定的四步模式（参见图 2-1）。第一步总是从一个与用户交互的 Web 页面中的可见帧开始的。显然，用户并不知道隐藏帧的存在（在现代浏览器中，它是不显示的），以通常的形式与网页进行交互。在某些时间，用户执行了一个需要从服务器获取额外数据的操作。当这个操作发生时，第一步就发生了：产生一个对隐藏帧的 JavaScript 函数调用。这个调用可以简单地将隐藏帧重定向到另一个页面，或者复杂地传送表单数据。不管这个函数有多复杂，其结果都是产生第 2 步：向服务器发送一个请求。

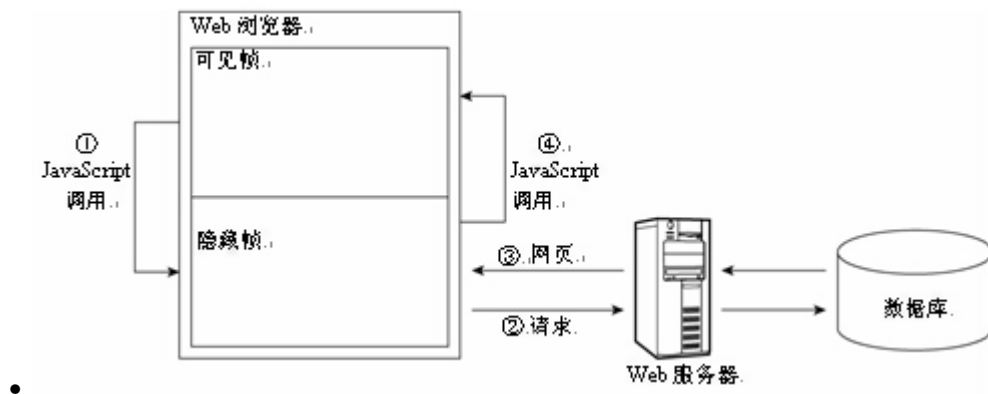


图 2-1

- 该模式中的第 3 步是从服务器上接收一个响应。由于处理的是帧，因此该响应必然是另一个网页。该网页必须包含从服务器返回的所请求的数据，同时一些 JavaScript 将把这些数据传给可见的帧。通常，这是通过在返回的网页中分配一个 onload 事件处理函数（event handler）做到的，该网页在其全部载入之后调用可见帧中的函数（这就是第 4 步）。当数据位于可见帧中后，该帧就可以决定如何处理这些数据了。

- 2. 隐藏帧的 GET 请求

- 我们已经阐述了隐藏帧技术的基本原理，现在将更深入地研究它。对于任何一种新技术，最好的方法就是通过具体的实例来学习。在该实例中，将创建一个简单的查询页面，客户服务代表通过该页面可以查询客户的信息。由于这是本书的第一个例子，因此它十分的简单：用户输入客户 ID，然后接收与该客户相关的信息。由于该功能通常需要数据库支持，因此还必须做一些服务器端的开发。该例子使用的是 PHP——这是一种优秀的开源服务端语言，还将使用到 MySQL（在从 www.mysql.org 下载）——这是一种与 PHP 结合得很好的开源数据库。

- 尽管本例确定为使用 MySQL，但只需少量的修改就可以在其他数据库上运行。

- 首先，在实现客户资料查询之前，你必须有一个包含该信息的数据库表。可以使用以下 SQL 脚本来创建一个客户表：

- CREATE TABLE `Customers` (
- `CustomerId` int(11) NOT NULL auto_increment,
- `Name` varchar(255) NOT NULL default '',
- `Address` varchar(255) NOT NULL default '',

- ``City` varchar(255) NOT NULL default ''`,
- ``State` varchar(255) NOT NULL default ''`,
- ``Zip` varchar(255) NOT NULL default ''`,
- ``Phone` varchar(255) NOT NULL default ''`,
- ``E-mail` varchar(255) NOT NULL default ''`,
- `PRIMARY KEY (`CustomerId`)`
- `) TYPE=MyISAM COMMENT='Sample Customer Data';`
- 在这张数据库表中最重要字段是 `CustomerId`，我们将通过它来查询客户信息。
- 你可以在 www.wrox.com 下载这个脚本以及一些测试数据。
- 当建好数据库表后，就可以将精力转到 HTML 代码上了。要使用隐藏帧技术，首先

必须创建一个 HTML 帧集，例如：

- `<frameset rows="100%,0" frameborder="0">`
- `<frame name="displayFrame" src="display.htm" noresize="noresize" />`
- `<frame name="hiddenFrame" src="about:blank" noresize="noresize" />`
- `</frameset>`
- 这部分代码中最重要的是 `<frameset/>` 元素的 `rows` 属性。通过将其设置为 `100%,0`，浏览器就知道不显示名为 `hiddenFrame` 的第二个帧了。紧接着，将 `frameborder` 属性设置为 `0` 则是确保每个帧都没有可见的边框。在帧集声明中最后一个重要的步骤是为每个帧设置 `noresize` 属性，使得用户不可能在不经意间调整帧的大小而发现隐藏帧，隐藏帧的内容永远不会成为可显示的用户界面的一部分。

• 接下来要处理的是一个请求和显示客户信息的页面。这是一个相对简单的页面，由一个用来输入客户 ID 的文本框，一个执行请求的按钮，以及用来显示查询到的客户信息的 `<div>` 元素所组成：

- `<p>Enter customer ID number to retrieve information:</p>`
- `<p>Customer ID: <input type="text" id="txtCustomerId" value="" /></p>`
- `<p><input type="button" value="Get Customer Info"`
- `onclick="requestCustomerInfo()" /></p>`
- `<div id="divCustomerInfo"></div>`
- 注意，按钮调用的是名为 `requestCustomerInfo()` 的函数，该函数将负责与隐藏

帧交互以获取数据。它将获取文本框中的值，将其添加到 `getcustomerdata.php` 的查询

字符串上，以 `getcustomerdata.php?id=23` 的格式创建一个 URL。然后将这个 URL 指派给隐藏帧，以下就是这个函数的代码：

- `function requestCustomerInfo() {`
- `var sId = document.getElementById("txtCustomerId").value;`
- `top.frames["hiddenFrame"].location = "getcustomerdata.php?id=" + sId;`
- `}`

• 该函数的第一步是从文本框中获取客户标识号（"txtCustomerId"）。这是将文本框的 ID `txtCustomerId` 作为参数，调用 `document.getElementById()` 函数，并获取返回的 `value` 属性（`value` 属性保存了文本框中的文本内容）来实现的。然后，将这个 ID 添加到字符串 `getcustomerdata.php?id=` 之后生成完整的 URL。第二行代码则是创建此 URL 并将其赋给隐藏帧。为了获得对隐藏帧的引用，首先要使用 `top` 对象来获取浏览器的顶级窗口（`topmost window`）。该对象拥有一个 `frames` 数组，在其中可以找到这个隐藏帧。由于每个帧都是一个窗口对象，因此可以将其位置设置为预期的 URL。

• 这是发出请求所需的所有信息。注意，由于这是一个 GET 请求（通过一个查询字符串传递信息），因此是很简单的。（很快，你将看到如何使用隐藏帧技术来执行一个 POST 请求。）

• 除了 `requestCustomerInfo()` 函数之外，还需要另一个在查询后显示客户信息的函数。当数据返回时，隐藏帧将调用这个 `displayCustomerInfo()` 函数，其唯一的参数是包含要显示的客户数据的字符串：

- `function displayCustomerInfo(sText) {`
- `var divCustomerInfo = document.getElementById("divCustomerInfo");`
- `divCustomerInfo.innerHTML = sText;`
- `}`

• 在这个函数中，第一行代码将查询对用于数据显示的 `<div/>` 元素的引用。第二行代码将把包含客户信息的字符串（`sText`）的值赋给 `<div/>` 元素的 `innerHTML` 属性。使用 `innerHTML` 属性，可以将 HTML 嵌入到格式化的字符串中。这将由主显示页面的代码来完成。现在我们将创建服务器端程序逻辑。

- `getcustomerdata.php` 中的基本代码是在基本的 HTML 页面上添加两处 PHP 代码：
- `<html>`
- `<head>`

- <title>Get Customer Data</title>
- <?php
- //php 代码
- ?>
- </head>
- <body>
- <div id="divInfoToReturn"><?php echo \$sInfo ?></div>
- </body>
- </html>

• 在该页面中，第一个 PHP 代码块将包括查询客户数据的逻辑（很快将讨论到）。

而第二个 PHP 代码块则负责将包含客户数据的 \$sInfo 变量的值输出到 <div/> 元素中。从这个 <div/> 元素中，你可以读取该数据并将数据传送给显示帧。为此，需要创建在页面完全载入后调用的 JavaScript 函数。

- window.onload = function () {
- var divInfoToReturn = document.getElementById("divInfoToReturn");
- top.frames["displayFrame"].displayCustomerInfo(divInfoToReturn.innerHTML);
- };
- 该函数将直接赋给 window.onload 事件处理函数中。它首先获取对包含客户信息的 <div/> 元素的引用，然后使用数组 top.frames 访问显示帧，并调用前面定义的 displayCustomerInfo() 函数，将其传给 <div/> 元素的 innerHTML 属性。这就是所有与发送该信息相关的 JavaScript。但首先如何获取这些信息呢？需要一些 PHP 代码来从数据库查询信息。

• 在 PHP 代码中的第一步是定义所有需要的数据块。在本例中，这些数据块包括用来查询的客户 ID、返回信息的 \$sInfo 变量，以及访问数据库所需要的信息（数据库服务器、数据库名、用户名、密码以及 SQL 查询字符串）：

- <?php
- \$sID = \$_GET["id"];
- \$sInfo = "";
- \$sDBServer = "your.databaseserver";

- `$sDBName = "your_db_name";`
- `$sDBUsername = "your_db_username";`
- `$sDBPassword = "your_db_password";`
- `$sQuery = "Select * from Customers where CustomerId=".$sID;`
- `//更多代码`
- `?>`

• 这段代码首先从查询字符串中获取 `id` 参数。为了便于获取，PHP 将所有的查询字符串参数组织于 `$_GET` 数组中。这个 `id` 存储在 `$sID` 中，它将用来创建存储于 `$sQuery` 中的 SQL 查询字符串。在此还将创建 `$sInfo` 变量，并将其设置为空字符串。在这段代码中的所有其他变量，都包含了指定特定数据库配置的信息，根据你自己的实现环境将其替换为正确的值。

• 获取了用户的输入，做好了连接数据库的基本准备，下一步就是创建数据库连接，执行查询，返回结果。如果存在一个指定 ID 的客户，`$sInfo` 将填入包含所有数据的 HTML 字符串，包括对电子邮件地址创建一个链接，如果客户 ID 是无效的，那么 `$sInfo` 将填入错误消息，以传给显示帧：

- `<?php`
- `$sID = $_GET["id"];`
- `$sInfo = "";`
- `$sDBServer = "your.databaseserver.server";`
- `$sDBName = "your_db_name";`
- `$sDBUsername = "your_db_username";`
- `$sDBPassword = "your_db_password";`
- `$sQuery = "Select * from Customers where CustomerId=".$sID;`
- `$oLink = mysql_connect($sDBServer, $sDBUsername, $sDBPassword);`
- `@mysql_select_db($sDBName) or $sInfo="Unable to open database";`
- `if($oResult = mysql_query($sQuery) and mysql_num_rows($oResult) > 0) {`
- `$aValues = mysql_fetch_array($oResult, MYSQL_ASSOC);`
- `$sInfo = $aValues['Name']. "
". $aValues['Address']. "
".`
- `$aValues['City']. "
". $aValues['State']. "
".`
- `$aValues['Zip']. "

Phone: ". $aValues['Phone']. "
".`

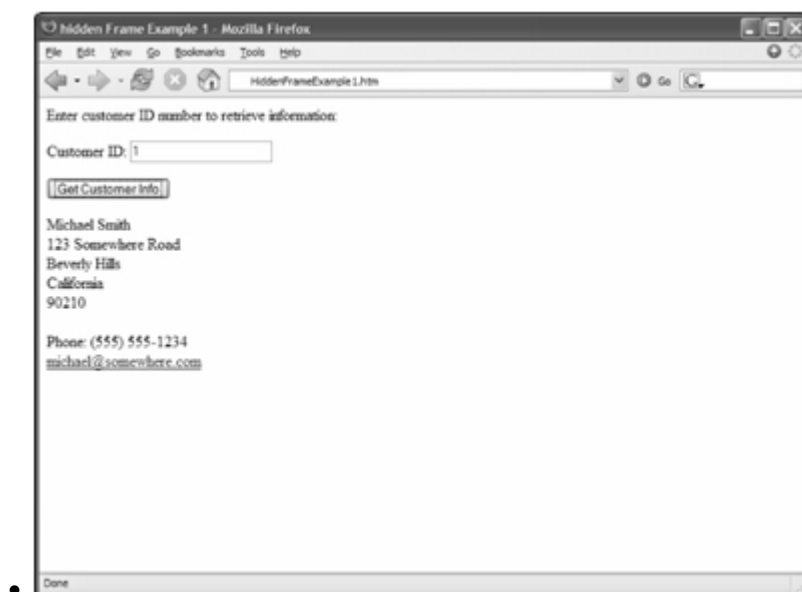
- "\".
- \$aValues['E-mail'].\"\";
- } else {
- \$sInfo = "Customer with ID \$sID doesn't exist.\";
- }
- mysql_close(\$oLink);
- ?>

• 突出显示的头两行代码用来完成从 PHP 到 MySQL 数据库的连接。紧接着，调用 `mysql_query()` 函数来执行 SQL 查询。如果函数返回结果，并且该结果至少包括一行，那么程序将获取该信息，并将其存入变量 `$sInfo` 中；否则，`$sInfo` 将填入一个错误消息。最后两行则负责释放数据库连接。

• 关于更复杂的 PHP 和 MySQL 编程的阐述已超出了本书讨论的范围。如果你想了解更多这方面的知识，可以阅读 *Beginning PHP and MySQL 5: From Novice to Professional* [1]（Apress 出版社，ISBN 1-59059-552-1）。

• 现在当 `$sInfo` 输出到 `<div>` 元素时，它将包含正确的信息。`onload` 事件处理函数将读取这些数据，然后将其发送到显示帧上。如果查询到客户，其相应的信息将会显示出来，如图 2-2 所示。

• 另一方面，如果客户不存在，则会在屏幕的相同位置显示错误消息。无论如何，客户服务代表都将获得一个很好的用户体验。你的第一个 Ajax 程序也就完成了。



• 图 2-2

- 3. 隐藏帧的 POST 请求

- 前面的例子使用 GET 请求来从数据库中获取信息。由于客户 ID 能够以查询字符串的形式添加到 URL 中，因此十分简单。但如果需要发送 POST 请求该怎么办呢？它也可以使用隐藏帧技术，不过需要一些额外的工作。

- POST 请求通常是用于向服务器发送数据的场合，而与 GET 请求仅从服务器上获取数据不同。尽管 GET 请求可以通过查询字符串来向服务器发送额外的数据，但一些浏览器最多只能处理 512KB 以内的查询字符串信息。对于 POST 请求而言，则可以发送 2GB 的信息，能够良好地满足绝大多数的应用。

- 从传统意义上说，只能够通过将表单的 method 属性设置为 post 来发送 POST 请求。然后，包含在表单中的数据就会通过 POST 请求发送到 action 属性中指定的 URL 上。更复杂的问题是当表单提交之后，将会从当前页跳转到一个新的 URL 上，这与 Ajax 的目的是背道而驰的。但万幸的是，可以通过表单中一个不太知名的 target 属性来简单实现。

- <form/>元素的 target 属性的功能从某种意义上说与<a/>元素的 target 属性的功能类似：用来指定跳转的目的 URL。通过设置表单元素的 target 属性，可以有效地使得在其他帧或窗口（在本例中是隐藏帧）中显示出表单的提交结果之后，表单页面仍然保持不变。

- 首先重新定义一个帧集。与上一个例子唯一不同的是可见帧包含了用来输入客户数据的表单：

- <frameset rows="100%,0" frameborder="0">
- <frame name="displayFrame" src="entry.htm" noresize="noresize" />
- <frame name="hiddenFrame" src="about:blank" noresize="noresize" />
- </frameset>

- 输入表单的内容包含在一个<form/>元素中，而且针对保存在数据库中的每个字段都有一个相应的文本框（除了自动生成的客户 ID 之外）。同样也有一个<div/>元素，用来显示与客户端—服务器通信相关的状态信息：

- <form method="post" action="SaveCustomer.php" target="hiddenFrame">
- <p>Enter customer information to be saved:</p>
- <p>Customer Name: <input type="text" name="txtName" value="" />

- Address: <input type="text" name="txtAddress" value="" />

- City: `<input type="text" name="txtCity" value="" />
`
- State: `<input type="text" name="txtState" value="" />
`
- Zip Code: `<input type="text" name="txtZipCode" value="" />
`
- Phone: `<input type="text" name="txtPhone" value="" />
`
- E-mail: `<input type="text" name="txtEmail" value="" /></p>`
- `<p><input type="submit" value="Save Customer Info" /></p>`
- `</form>`
- `<div id="divStatus"></div>`
- 注意, `<form/>` 元素的 `target` 属性也设置为 `hiddenFrame`, 因此当用户点击该按钮时, 提交的结果将显示在隐藏帧中。

• 在本例中, 主页面中只需要一个 JavaScript 函数: `saveResult()`。当隐藏帧返回客户数据保存结果时, 将调用该函数:

- `function saveResult(sMessage) {`
- `var divStatus = document.getElementById("divStatus");`
- `divStatus.innerHTML = "Request completed: " + sMessage;`
- `}`
- 隐藏帧的职责是向该函数传递一个消息, 该消息将显示给用户。它可能是信息已保存的确认信息, 或者是说明为什么保存失败的错误信息。

• 接下来处理 POST 请求的文件是 `SavaCustomer.php`。与前一个例子一样, 该页面也是由简单的 HTML 页面加上一些 PHP 和 JavaScript 代码组成的。其中 PHP 代码用来从请求中收集信息, 然后将其保存到数据库中。由于这是一个 POST 请求, 因此 `$_POST` 数组中包含了提交的所有信息:

- `<?php`
- `$sName = $_POST["txtName"];`
- `$sAddress = $_POST["txtAddress"];`
- `$sCity = $_POST["txtCity"];`
- `$sState = $_POST["txtState"];`
- `$sZipCode = $_POST["txtZipCode"];`
- `$sPhone = $_POST["txtPhone"];`
- `$sEmail = $_POST["txtEmail"];`

- `$sStatus = "";`
- `$sDBServer = "your.database.server";`
- `$sDBName = "your_db_name";`
- `$sDBUsername = "your_db_username";`
- `$sDBPassword = "your_db_password";`
- `$sSQL = "Insert into`

`Customers(Name, Address, Ci ty, State, Zi p, Phone, `E-mai l`) "`.

- `" values (' $sName' , ' $sAddress' , ' $sCi ty' , ' $sState' , ' $sZi pCode' "`
- `", ' $sPhone' , ' $sEmail')";`
- `//更多代码`
- `?>`

这个代码片段将获取与客户相关的所有 POST 信息；此外，还定义了一个状态消息（`$sStatus`）以及所需的数据库信息（与上一个例子相同）。这里的 SQL 语句是一个 INSERT 语句，它将获取的信息添加到数据库中。

- 执行这个 SQL 语句的代码与上一个例子十分类似：

- `<?php`
- `$sName = $_POST["txtName"];`
- `$sAddress = $_POST["txtAddress"];`
- `$sCi ty = $_POST["txtCi ty"];`
- `$sState = $_POST["txtState"];`
- `$sZi pCode = $_POST["txtZi pCode"];`
- `$sPhone = $_POST["txtPhone"];`
- `$sEmail = $_POST["txtEmail"];`
- `$sStatus = "";`
- `$sDBServer = "your.database.server";`
- `$sDBName = "your_db_name";`
- `$sDBUsername = "your_db_username";`
- `$sDBPassword = "your_db_password";`
- `$sSQL = "Insert into`

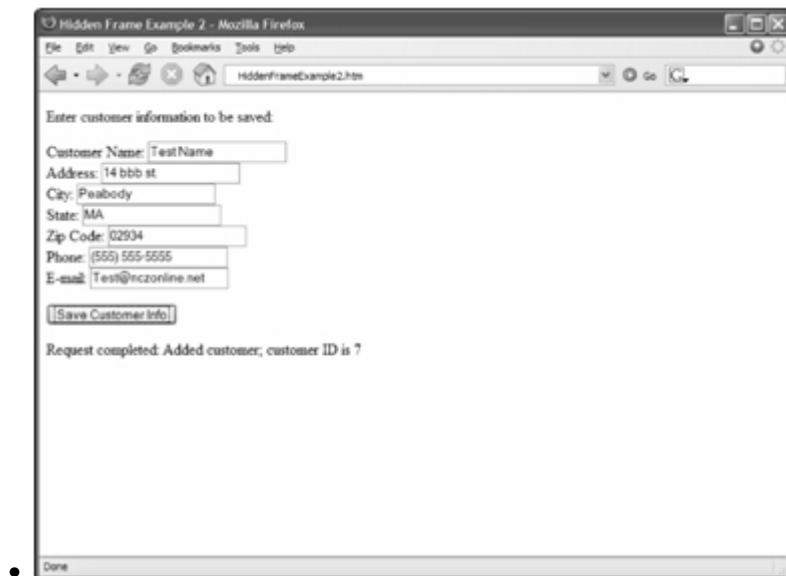
`Customers(Name, Address, Ci ty, State, Zi p, Phone, `E-mai l`) "`.

- " values ('\$sName', '\$sAddress', '\$sCity', '\$sState', '\$sZipCode' "
- ", '\$sPhone', '\$sEmail')";
- \$oLink = mysql_connect(\$sDBServer, \$sDBUsername, \$sDBPassword);
- @mysql_select_db(\$sDBName) or \$sStatus = "Unable to open database";
- if(\$oResult = mysql_query(\$sSQL)) {
- \$sStatus = "Added customer; customer ID is ".mysql_insert_id();
- } else {
- \$sStatus = "An error occurred while inserting; customer not saved.";
- }
- mysql_close(\$oLink);
- ?>

在此，mysql_query()函数的结果只是一个表示语句执行成功的指示器。如果执行成功，\$sStatus 变量中将填入一个消息，表明保存已经成功，并返回为该数据指定的客户 ID。mysql_insert_id()函数始终返回在最新的 INSERT 语句返回值的基础上自动递增的值。如果因为某些原因，该语句没有成功执行，\$sStatus 变量将填入一个错误消息。

- \$sStatus 变量将输出到一个在载入窗口时运行的 JavaScript 函数中：
- <script type="text/javascript">
- window.onload = function () {
- top.frames["displayFrame"].saveResult("<?php echo \$sStatus ?>");
- }
- </script>

这段代码调用了 saveResult()函数，该函数定义于显示帧中，传入的参数值是 PHP 变量\$sStatus。由于该变量包含一个字符串，因此必须将 PHP 的 echo 语句放在引号中。当执行该函数时，假设客户数据已保存，则输入表单页面看起来如图 2-3 所示。



• 图 2-3

• 当执行这段代码之后，你还可以自由地使用同样的表单向数据库中添加更多客户，因为它不再消失。

• 4. 隐藏 iFrame

• 新一代的客户端—服务器通信模式幕后所采用的是 i frame，它是在 HTML 4.0 中引入的。i frame 与帧基本是相同的，唯一的区别是 i frame 可以放在一个未设置帧集的 HTML 页面中，可以使页面中的任意部分成为一个帧。i frame 技术可以在未预先设置帧集的页面中使用，能够更好地适应于功能的逐渐添加。i frame 甚至还可以使用 JavaScript 在运行时创建，为了简单起见，语义化 HTML（semantic HTML）支持使浏览器将 Ajax 功能看作是一个有益的增强（这将在稍后讨论）。由于可以用与普通帧相同的方法使用和访问 i frame，因此它们都是 Ajax 通信的理想选择。

• 发挥 i frame 的优势有两种方法。最简单的方法是在页面中简单地嵌入 i frame，并像隐藏帧那样用来发出请求。为此，第一个例子中的显示页面将修改为：

- <p>Enter customer ID number to retrieve information:</p>
- <p>Customer ID: <input type="text" id="txtCustomerId" value="" /></p>
- <p><input type="button" value="Get Customer Info"
- onclick="requestCustomerInfo()" /></p>
- <div id="divCustomerInfo"></div>
- <i frame src="about:blank" name="hiddenFrame" width="0" height="0"
- frameborder="0"></i frame>

• 注意，这个 `iframe` 中的 `width`、`height` 和 `frameborder` 属性都设置成了 0，这可将其从视线中隐去。由于 `iframe` 的名字仍是 `hiddenFrame`，所以这个页面的 JavaScript 代码可以如前一样正常工作。不过，对于 `GetCustomerData.php` 页面还需要做一些小的修改。在该页面中的 JavaScript 函数先前是在名为 `displayFrame` 的帧中查找 `displayCustomerInfo()` 函数。如果你使用该技术，又不存在该名字的帧，则必须修改代码，用 `parent` 来代替它：

- `window.onload = function () {`
- `var divInfoToReturn = document.getElementById("divInfoToReturn");`
- `parent.displayCustomerInfo(divInfoToReturn.innerHTML);`
- `};`

• 现在这个例子能够和本章中的第一例子一样正常工作了。

• 第二种使用隐藏 `iframe` 的方法是通过 JavaScript 动态地创建它们。由于并非所有浏览器实现 `iframe` 的方法都是一样的，所以需要一些技巧，使得它有助于一步步地创建隐藏的 `iframe`。

• 第一步很简单，使用 `document.createElement()` 方法创建 `iframe` 并赋予必要的属性：

- `function createIFrame() {`
- `var olFrameElement = document.createElement("iframe");`
- `olFrameElement.width=0;`
- `olFrameElement.height=0;`
- `olFrameElement.frameBorder=0;`
- `olFrameElement.name = "hiddenFrame";`
- `olFrameElement.id = "hiddenFrame";`
- `document.body.appendChild(olFrameElement);`
- `//更多代码`
- `}`

• 本段代码的最后一行很重要，因为它将 `iframe` 添加到 `document` 结构中；没有添加到 `document` 中的 `iframe` 是无法执行请求的。另外注意，该 `iframe` 的 `name` 和 `id` 属性都是设置为 `hiddenFrame`。这是必要的，因为有些浏览器是通过 `name` 属性访问新的帧，而有些则是通过 `id` 属性新的帧。

• 紧接着定义一个全局变量，用来保存对该帧对象的引用。注意，针对 i frame 元素的这个帧对象并非是从 createElement() 函数返回的。要获得该对象，必须从帧集合中获取。以下就是即将保存在全局变量中的内容：

- var oIFrame = null;
- function createIFrame() {
- var oIFrameElement = document.createElement("i frame");
- oIFrameElement.width=0;
- oIFrameElement.height=0;
- oIFrameElement.frameBorder=0;
- oIFrameElement.name = "hiddenFrame";
- oIFrameElement.id = "hiddenFrame";
- document.body.appendChild(oIFrameElement);
- oIFrame = frames["hiddenFrame"];
- }
- 如果你将这些代码放到前面的 i frame 例子中,那么需要对 requestCustomerInfo()

函数进行如下修改：

- function requestCustomerInfo() {
- if (!oIFrame) {
- createIFrame();
- setTimeout(requestCustomerInfo, 10);
- return;
- }
- var sId = document.getElementById("txtCustomerId").value;
- oIFrame.location = "GetCustomerData.php?id=" + sId;
- }

• 基于这些修改，该函数将会检查 oIFrame 是否为空。如果为空，则调用 createFrame()，并会为该函数的调用设置 10ms 的超时时间。这是很必要的，因为只有 IE 浏览器能够立即识别插入的 i frame，大部分其他浏览器需要花几毫秒来识别它，以允许通过它发送请求。当再次执行该函数时，将执行代码的其余部分，其中最后一行已经修改为对 OIFrame 对象的引用。

• 尽管该技术能够很容易地应用于 GET 请求，但 POST 请求却完全不同。只有一部分浏览器允许你设置表单的 target 属性来动态创建 iframe；但 IE 并不是其中的一种。因此，要使用隐藏 iframe 技术来发送 POST 请求还需要一些技巧。

• 5. 隐藏 iframe 的 POST 请求

• 要使用隐藏 iframe 来完成 POST 请求，其方法是在隐藏帧中载入一个包含表单的页面，用数据填充该表单，然后再提交该表单。当这个可见的表单（你实际输入数据的那个）提交时，必须取消这次提交而将信息转发给隐藏帧。为此，必须定义一个函数，用来处理 iframe 的创建以及隐藏表单的载入：

```
• function checkIFrame() {  
• if (!oIFrame) {  
• createIFrame();  
• }  
• setTimeout(function () {  
• oIFrame.location = "ProxyForm.htm";  
• }, 10);  
• }
```

• 这个名为 checkIFrame() 的函数首先检查隐藏的 iframe 是否已经创建。如果没有，则调用 createIFrame()。然后，在将 iframe 的地址设置为 ProxyForm.htm（这是一个隐藏表单页面）之前，为其设置一个超时值。由于该函数调用需要花一些时间，而重要的是该页面每次加载时都将提交该表单。

• ProxyForm.htm 文件很简单，只包括很少的 JavaScript，用来提示主页面已经装载完成：

```
• <html>  
• <head>  
• <title>Proxy Form</title>  
• <script type="text/javascript">  
• window.onload = function () {  
• parent.formReady();  
• }  
• </script>
```

- `</head>`
- `<body>`
- `<form method="post"></form>`
- `</body>`
- `</html>`

正如你所见，该页面的主体只包含一个空的表单，而标题中只包含一个 `onload` 事件处理函数。当载入该页面时，页面将通过调用 `parent.formReady()` 来使主页面知道它已经做好接收请求的准备。而 `formReady()` 函数则是包含在主页面本身中的，类似于：

- `function formReady() {`
- `var oHiddenForm = oIFrame.document.forms[0];`
- `var oForm = document.forms[0];`
- `for (var i=0 ; i < oForm.elements.length; i++) {`
- `var oHidden = oIFrame.document.createElement("input");`
- `oHidden.type = "hidden";`
- `oHidden.name = oForm.elements[i].name;`
- `oHidden.value = oForm.elements[i].value;`
- `oHiddenForm.appendChild(oHidden);`
- `}`
- `oHiddenForm.action = oForm.action;`
- `oHiddenForm.submit();`
- `};`

在该函数中的第一步是获取对隐藏 `iFrame` 中表单的引用，可以通过访问该帧的 `document.forms` 集合来获取。由于在该页面中只有一个表单，因此可以安全地从该集合中获得第一个表单（即索引值为 0），并将其存储于 `oHiddenForm` 中。然后，将对主页面表单的引用存于 `oForm` 中。紧接着，一个 `for` 循环对主页面中该表单的各元素进行遍历（使用 `elements` 集合）。对于表单中的每一个元素，都将在隐藏帧（注意，必须使用 `oIFrame.document.createElement()` 而不只是 `document.createElement()`）中创建一个隐藏的输入元素。这个隐藏的输入元素拥有与该表单元素相同的名字和值，然后使用 `appendChild()` 函数将其添加到隐藏的表单中。

- 当所有的表单元素都添加完后,隐藏的表单还将设置与主页面表单相同的 `action`。通过从表单中读取 `action` 来取代硬编码,就可以在任何页面中使用 `formReady()`。该函数的最后一步是提交这个隐藏的表单。

- 剩下的最后一件事就是确保主页面的表单不以通常的方式提交自己。要达到这一目标,只需在 `onsubmit` 事件处理函数中调用 `checkIframe()` 并返回 `false`:

- `<form method="post" action="SaveCustomer.php"`
- `onsubmit="checkIframe();return false">`
- `<p>Enter customer information to be saved:</p>`
- `<p>Customer Name: <input type="text" name="txtName" value="" />
`
- `Address: <input type="text" name="txtAddress" value="" />
`
- `City: <input type="text" name="txtCity" value="" />
`
- `State: <input type="text" name="txtState" value="" />
`
- `Zip Code: <input type="text" name="txtZipCode" value="" />
`
- `Phone: <input type="text" name="txtPhone" value="" />
`
- `E-mail: <input type="text" name="txtEmail" value="" /></p>`
- `<p><input type="submit" value="Save Customer Info" /></p>`
- `</form>`
- `<div id="divStatus"></div>`

- 通过以这种方式返回 `false`,可以阻止表单的默认行为(将自己提交到服务器)。通过调用 `checkIframe()` 方法来启动隐藏 `iframe` 中表单的提交进程。

- 当这一任务完成后,就可以像使用隐藏帧 `POST` 请求的例子一样使用本例;页面 `SavaCustomer.php` 负责处理数据,并当完成时调用主页面中的 `savaResult()` 函数。

• 注意,本节中的例子是为了使其聚焦于与 `Ajax` 技术相关的问题上,因而进行了简化。如果在实际的 `Web` 应用程序中使用,还需要提供更多的用户反馈,诸如在发出请求时屏蔽该表单的输入等。

• 6. 隐藏帧技术的优点和缺点

- 现在,你已经对使用隐藏帧所实现的强大功能有所了解,我们将讨论它的实用性。正如前面所说的,该技术已经存在多年,并且仍然在许多 `Ajax` 应用中使用。

- 使用隐藏帧的一个最大理由之一是它可以维护浏览器的历史,使用户仍然能够使用浏览器上的后退和前进按钮。浏览器由于并不知道隐藏帧实际上被隐藏了,但对于其

所发出的请求仍然是记录在案的。然而，Ajax 应用程序的主页面却没有修改，在隐藏帧中的修改意味着后退和前进按钮将依据该隐藏帧的访问历史而非主页面而变化。这也是为什么 Gmail 和 Google Maps 仍然使用该技术的理由。

- 注意，iframe 并非一直会存储浏览器的历史记录。尽管 IE 始终会存储 iframe 的历史记录，但 Firefox 只对使用 HTML 定义（也就是不包括使用 JavaScript 动态创建）的 iframe 保存历史记录。Safari 从不为 iframe 保存历史记录，不管它们是如何包含在该页面中的。

- 隐藏帧技术不利的一面是，对其背后发生的事了解甚少。它完全依赖于返回的正确页面。本节的例子都存在相同的问题：如果隐藏帧的页面载入失败，并不会向用户提示出错消息；主页面将继续等待直到调用适当的 JavaScript 函数。必须通过设置一个较长周期（可能是 5 分钟）的超时时间，然后如果页面仍然没有成功载入则显示一条消息，以给用户一个安慰。但这一切都只是一个变通方法，最主要的问题是，对于后台发生的 HTTP 请求缺乏充足的信息。幸运的，我们还有其他选择。

• 2.2.2 XMLHttpRequest 请求

- 当微软的 IE 5.0 引入了一个基本的 XML 支持时，同时引入了一个名为 MSXML 的 ActiveX 库（将在第 4 章更详细地讨论）。该库所提供的 XMLHttpRequest 对象很快变得十分流行。

- 通过 XMLHttpRequest 对象，开发人员可以在应用程序的任何地方初始化 HTTP 请求。这些请求将以 XML 格式返回，因而 XMLHttpRequest 对象提供了一种以 XML 文档的格式访问信息的简单途径。由于 XMLHttpRequest 是一个 ActiveX 控件，因此不仅能够在网页上使用，还可以在任何 Windows 桌面应用程序中使用；但是在网页上的流行速度要比在桌面应用程序上的流行速度快得多。

- 随着 XMLHttpRequest 对象的流行，Mozilla 在其浏览器（诸如 Firefox）中也实现了 XMLHttpRequest 的功能。此后不久，Safari（1.2 版）和 Opera（7.6 版）浏览器也复制了 Mozilla 的实现。现在，四种浏览器都在不同程度上实现了对 XMLHttpRequest 支持。（Safari 和 Opera 的实现还不完善，对于 GET 和 POST 之外的请求还不支持。）

• 1. 创建 XMLHttpRequest 对象

• 使用 XMLHttpRequest 对象的第一步显然是创建一个对象实例。由于微软将其实现为一个 ActiveX 控件，因此必须在 JavaScript 中使用其专有的 ActiveXObject 类，并传入 XMLHttpRequest 控件的签名：

- `var oXmlHttp = new ActiveXObject("Microsoft.XMLHttp");`

• 这行代码创建了第一个版本（即 IE 5.0 中发布的）的 XMLHttpRequest 对象。问题是随着 MSXML 库后续版本的发布，也发布了 XMLHttpRequest 的几个新版本。每个新版本都更加稳定、速度更快，因此确保在用户机器中使用最新的可用版本。这些签名包括：

- `© Microsoft.XMLHttp`

- `© MSXML2.XMLHttp`

- `© MSXML2.XMLHttp.3.0`

- `© MSXML2.XMLHttp.4.0`

- `© MSXML2.XMLHttp.5.0`

• 不幸的是，确定使用的最好版本的唯一方法是必须试着逐个创建它们。由于这是一个 ActiveX 控件，创建对象时发生的所有问题都会抛出一个异常，也就意味着你必须将其包含在 try...catch 程序块中。该函数最终的结果类似于：

- `function createXMLHttp() {`
- `var aVersions = ["MSXML2.XMLHttp.5.0",`
- `"MSXML2.XMLHttp.4.0", "MSXML2.XMLHttp.3.0",`
- `"MSXML2.XMLHttp", "Microsoft.XMLHttp"`
- `];`
- `for (var i = 0; i < aVersions.length; i++) {`
- `try {`
- `var oXmlHttp = new ActiveXObject(aVersions[i]);`
- `return oXmlHttp;`
- `} catch (oError) {`
- `//不处理`
- `}`
- `}`
- `throw new Error("MSXML is not installed.");`
- `}`

• createXMLHttpRequest() 函数中存储了一个 XMLHttpRequest 签名的数组，其中最新的签名位于第一个。该函数将遍历这个数组，尝试基于每个签名来创建 XMLHttpRequest 对象。如果创建失败，catch 语句将避免 JavaScript 错误而使程序停止执行；然后再尝试下一个签名。当创建一个对象之后，将返回该对象。如果函数执行完，还没有成功创建 XMLHttpRequest 对象，将抛出一个错误以表示创建失败。

• 幸运的是，在其他浏览器中创建一个 XMLHttpRequest 对象是很简单的。Mozilla Firefox、Safari 和 Opera 使用的代码都相同：

• var oXMLHttpRequest = new XMLHttpRequest();

• 当然，拥有一种创建 XMLHttpRequest 对象跨浏览器的方法是很有帮助的。你可以将前面定义的 createXMLHttpRequest() 函数做如下修改从而创建这样的函数：

```
function createXMLHttpRequest() {  
    if (typeof XMLHttpRequest != "undefined") {  
        return new XMLHttpRequest();  
    } else if (window.ActiveXObject) {  
        var aVersions = [ "MSXML2.XMLHttp.5.0",  
            "MSXML2.XMLHttp.4.0", "MSXML2.XMLHttp.3.0",  
            "MSXML2.XMLHttp", "Microsoft.XMLHttp"  
        ];  
        for (var i = 0; i < aVersions.length; i++) {  
            try {  
                var oXMLHttpRequest = new ActiveXObject(aVersions[i]);  
                return oXMLHttpRequest;  
            } catch (oError) {  
                //Do nothing  
            }  
        }  
        throw new Error("XMLHttpRequest object could not be created.");  
    }  
}
```


- 现在该函数首先检查是否定义了 XMLHttpRequest 类（通过 typeof 操作符）。如果已经存在 XMLHttpRequest，则用它来创建 XMLHttpRequest 对象；否则，检查 ActiveXObject 类是否存在，如果存在则采用与创建 IE 的 XMLHttpRequest 相同的方法进行处理。如果这些检查都失败，则抛出一个错误。

- 创建跨浏览器兼容的 XMLHttpRequest 对象的另一种方法是使用已编写好跨浏览器代码的库。本书的两位作者开发的 zXml 库就是这样的库，它可以在

www.nczonline.net/downloads/中下载。该库为创建 XMLHttpRequest 对象定义了一个函数：

- `var oXmlHttp = zXmlHttp.createRequest();`

- 本书将使用 createRequest() 函数以及 zXml 库本身来帮助 Ajax 技术实现跨浏览器处理。

• 2. 使用 XMLHttpRequest

- 在创建 XMLHttpRequest 对象之后，就可以开始在 JavaScript 中发起 HTTP 请求了。第一步是调用 open() 方法来初始化该对象。该方法可以接受以下参数：

- ◎ 请求类型（request type）：说明所发送的请求类型的字符串——通常是 GET 或 POST（这也是现在所有浏览器都支持的类型）；

- ◎ URL：说明发送请求的目标 URL 的字符串；

- ◎ async：布尔值，用来说明请求是否为异步模式。

- 最后一个参数 async 是很重要的，因为它是用来控制 JavaScript 如何执行该请求。当设置为 true 时，将以异步模式发送该请求，JavaScript 代码将继续执行而不再等待响应，且必须使用一个事件处理函数来监控请求的响应。如果将 async 设置为 false，则将以同步模式发送该请求，JavaScript 将等待接收到响应后再继续执行剩余代码。这意味着如果响应时间很长，则用户在浏览器收到响应之前是无法与其交互的。基于这个原因，Ajax 应用程序开发的最佳实践是，使用异步请求来实现数据获取，使用同步请求来实现与服务器之间发送和接收简单的消息。

- 如果要以异步模式向 info.txt 发出请求，则最先要做的事是：

- `var oXmlHttp = zXmlHttp.createRequest();`

- `oXmlHttp.open("get", "info.txt", true);`

- 注意，这个例子中的第一个参数是请求类型，虽然请求类型通常用大写字母定义，但在这里并不区分大小写。

• 紧接下来，需要定义一个 `onreadystatechange` 事件处理函数。`XMLHttpRequest` 对象有一个名为 `readyState` 的属性，该属性从请求发送到接收响应期间会发生变化。`readyState` 共有 5 种可能的取值：

- `0` (`uninitialized`, 未初始化): 对象已经创建，但还没有调用 `open()` 方法；
- `1` (`loading`, 载入中): `open()` 方法已经调用，但请求还没有发送；
- `2` (`loaded`, 已载入): 请求已经发送；
- `3` (`interactive`, 交互中): 已经接收到部分响应；
- `4` (`complete`, 完成): 所有数据都已经收到，连接已经关闭。

• 每当 `readyState` 属性的值发生变化时，就将触发 `readystatechange` 事件，并调用 `onreadystatechange` 事件处理函数。由于浏览器的实现不同，要实现跨浏览器开发，则可靠的 `readyState` 的值是 0、1 和 4。在大部分情况下，当返回请求时只需检查值是否为 4。

- `var oXmlHttp = XMLHttpRequest.createRequest();`
- `oXmlHttp.open("get", "info.txt", true);`
- `oXmlHttp.onreadystatechange = function () {`
- `if (oXmlHttp.readyState == 4) {`
- `alert("Got response.");`
- `}`
- `};`

• 最后一步是调用 `send()` 方法，它将完成请求的发送。该方法只接受一个参数，即表示请求主体的字符串。如果请求不包含主体（应该记得，GET 请求就不包含），则必须传入 `null`：

- `var oXmlHttp = XMLHttpRequest.createRequest();`
- `oXmlHttp.open("get", "info.txt", true);`
- `oXmlHttp.onreadystatechange = function () {`
- `if (oXmlHttp.readyState == 4) {`
- `alert("Got response.");`
- `}`
- `};`
- `oXmlHttp.send(null);`

- 好了！请求发送并接收到响应时将显示出一个警告框（alert）。但仅将请求已经接收到的信息显示出来并没有太大用处。XMLHttpRequest 强大之处主要体现在可以访问返回的数据、响应的状态以及响应的首部。

- 要获得从请求返回的数据，可以使用 responseText 或 responseXML 属性。responseText 属性返回一个包含响应主体的字符串，而 responseXML 属性则返回一个 XML 文档对象，它仅当返回的内容类型为 text/xml 时才使用。（XML 文档对象将在第 4 章详细说明。）因此，要获取 info.txt 中的文本，调用应为：

- var sData = oXmlHttp.responseText;
- 注意，仅当文件找到了并且没有发生错误，才能够返回 info.txt 中的文本。假设 info.txt 不存在，那么 responseText 将包含服务器的 404 消息。幸运的是，我们还有办法判断是否出现了错误。

- status 属性中包含了响应中的 HTTP 状态码，而 statusText 则包含对该状态的文字描述（诸如“OK”或“Not Found”）。使用这两个属性，就可以确保接收的数据是你实际需要的数据，或者告诉用户为什么查找不到所要的数据：

- if (oXmlHttp.status == 200) {
- alert("Data returned is: " + oXmlHttp.responseText;
- } else {
- alert("An error occurred: " + oXmlHttp.statusText;
- }
- 通常，总是确保响应的状态是 200，这说明请求已经成功地完成了。即使服务器端发生了错误，readyState 属性的值仍然会设置为 4，因此只检查它是不够的。在本例中，只有当 status 的值是 200 时才显示 responseText 属性的内容；否则将显示错误消息。

- 在 Opera 中并没有实现 statusText 属性，并且有时在其他浏览器中会返回一个错误的描述。因此永远不要只依赖于 statusText 来判断是否出现错误。

- 正如前面所提到的，你还可以访问响应的首部。使用 getResponseHeader() 方法，并传入需要获得的首部名称，就可以获得一个特定的首部值。最有用的响应首部是 Content-Type，它将告诉你所发送数据的格式：

- var sContentType = oXmlHttp.getResponseHeader("Content-Type");
- if (sContentType == "text/xml") {

- `alert("XML content received.");`
- `} else if (sContentType == "text/plain") {`
- `alert("Plain text content received.");`
- `} else {`
- `alert("Unexpected content received.");`
- `}`

• 这个代码片段用来检查响应的内容类型，并通过一个警告框来显示返回的数据类型。通常，从服务器可能只接收 XML 数据（内容类型为 `text/xml`）或纯文本（内容类型为 `text/plain`），因为只有这两种内容类型可以使用 JavaScript 更容易地处理。

• 如果你想看到从服务器中返回的所有首部，则可以使用 `getAllResponseHeaders()` 方法，它将简单地返回一个包含所有首部的字符串。在这个字符串中每个首部可以用一个换行字符（在 JavaScript 中表示为 `\n`），或一个回车加换行字符（在 JavaScript 中表示为 `\r\n`）隔开，因此可以像下面这样来处理每个首部：

- `var sHeaders = oXmlHttp.getAllResponseHeaders();`
- `var aHeaders = sHeaders.split(/\r?\n/);`
- `for (var i=0; i < aHeaders.length; i++) {`
- `alert(aHeaders[i]);`
- `}`

• 在这个例子中，我们使用 JavaScript 中字符串类型的 `split()` 方法，并传入一个正则表达式（通过回车/换行或换行来匹配），将每个首部字符串分解出来，存到一个首部数组中。现在你就可以遍历所有的首部并按自己的想法来处理。注意，在 `aHeaders` 数组中每个字符串的格式都是 `headername: headervalue`（首部名：首部值）。

• 在发送之前，我们还能够设置请求的首部。你可以在发送时说明数据的内容类型，或者需要发送一些服务器在处理该请求时需要的一些额外数据。要实现这一目标，可以在调用 `send()` 之前使用 `SetRequestHeader()` 方法：

- `var oXmlHttp = zXmlHttp.createRequest();`
- `oXmlHttp.open("get", "info.txt", true);`
- `oXmlHttp.onreadystatechange = function () {`
- `if (oXmlHttp.readyState == 4) {`
- `alert("Got response.");`

- }
- };
- oXmlHttp.setRequestHeader("myheader", "myvalue");
- oXmlHttp.send(null);

• 在这段代码中，在发送之前将在请求中添加了一个名为 myheader 的首部。该首部将以 myheader:myvalue 的默认格式进行添加。

• 到现在为止，你已经知道如何实现在大多数情况都能很好应用的异步请求了。发送同步请求则意味着无需指定 onreadystatechange 事件处理函数，因为在 send() 方法返回时将接收到其响应。因此，你可能像这样进行处理：

- var oXmlHttp = XMLHttpRequest.createRequest();
- oXmlHttp.open("get", "info.txt", false);
- oXmlHttp.send(null);
- if (oXmlHttp.status == 200) {
- alert("Data returned is: " + oXmlHttp.responseText;
- } else {
- alert("An error occurred: " + oXmlHttp.statusText;
- }

• 用同步模式来发送该请求（将 open() 方法的第三个参数设置为 false），可以使你在调用 send() 方法之后马上对其响应进行处理。这对于想让用户交互等待响应，或希望只接收很少的数据（例如，小于 1KB）的应用场景是很有用的。而对于通常的数据量或较大的数据量而言，最好还是使用异步调用。

• 3. XMLHttpRequest 的 GET 请求

• 现在我们来考虑用隐藏帧发送 GET 请求的例子，看看如何使用 XMLHttpRequest 来改进它。第一个要修改的是 GetCustomerData.php，必须将其从返回一个 HTML 页面改为返回一个 HTML 片段。现在整个文件就显得更加简练了：

- <?php
- header("Content-Type: text/plain");
- \$SID = \$_GET["id"];
- \$SInfo = "";
- \$SDBServer = "your.databaseserver";

- `$sDBName = "your_db_name";`
- `$sDBUsername = "your_db_username";`
- `$sDBPassword = "your_db_password";`
- `$sQuery = "Select * from Customers where CustomerId=".$sID;`
- `$oLink = mysql_connect($sDBServer, $sDBUsername, $sDBPassword);`
- `@mysql_select_db($sDBName) or $sInfo="Unable to open database";`
- `if($oResult = mysql_query($sQuery) and mysql_num_rows($oResult) > 0) {`
- `$aValues = mysql_fetch_array($oResult, MYSQL_ASSOC);`
- `$sInfo = $aValues['Name'] . "
". $aValues['Address'] . "
".`
- `$aValues['City'] . "
". $aValues['State'] . "
".`
- `$aValues['Zip'] . "

Phone: ". $aValues['Phone'] . "
".`
- `"".`
- `$aValues['E-mail'] . "";`
- `} else {`
- `$sInfo = "Customer with ID $sID doesn't exist.";`
- `}`
- `mysql_close($oLink);`
- `echo $sInfo;`
- `?>`

正如你所见，在该页面中没有可见的 HTML 或 JavaScript 调用。所有的主要逻辑还是相同的，但添加了两行新的 PHP 代码。第一处在开始位置，使用 `header()` 函数来设置页面的内容类型。即便该页面将返回一个 HTML 片段，但最好还是将内容类型设置为 `text/plain`，因为它不是一个完整的 HTML 页面（因此可能不是有效的 HTML）。对于向浏览器发送非 HTML 格式数据的页面，都应该设置为该内容类型。第二处则是接近最后面，使用 `echo` 命令将变量 `$sInfo` 的值输出到流。

- 在 HTML 主页面中，其基本的设置是：
- `<p>Enter customer ID number to retrieve information:</p>`
- `<p>Customer ID: <input type="text" id="txtCustomerId" value="" /></p>`
- `<p><input type="button" value="Get Customer Info"`
- `onclick="requestCustomerInfo()" /></p>`

- `<div id="divCustomerInfo"></div>`

- `requestCustomerInfo()` 函数先前创建的是隐藏 `iframe`，但现在必须改为使用

XMLHttpRequest:

- `function requestCustomerInfo() {`
- `var sId = document.getElementById("txtCustomerId").value;`
- `var oXmlHttp = new XMLHttpRequest();`
- `oXmlHttp.open("get", "GetCustomerData.php?id=" + sId, true);`
- `oXmlHttp.onreadystatechange = function () {`
- `if (oXmlHttp.readyState == 4) {`
- `if (oXmlHttp.status == 200) {`
- `displayCustomerInfo(oXmlHttp.responseText);`
- `} else {`
- `displayCustomerInfo("An error occurred: " + oXmlHttp.statusText);`
- `}`
- `}`
- `};`
- `oXmlHttp.send(null);`
- `}`

- 可以看到这个函数以相同的方法开始，都是先获取用户输入的 ID。然后，使用 `XMLHttpRequest` 库创建一个 `XMLHttpRequest` 对象。接着，调用 `open()` 方法，指定以异步模式发出对 `GetCustomerData.php`（并且将上面提到的 ID 附加到查询字符串后）的 GET 请求。紧接着将任务交给事件处理函数，由它来检查 `readyState` 的值是否为 4，以及该请求的 `status`。如果请求成功（`status` 的值是 200），那么将调用 `displayCustomerInfo()` 函数，并将响应主体（通过 `responseText` 访问）作为参数传入。如果发生了错误（`status` 的值不是 200），那么将其错误信息作为参数传给 `displayCustomerInfo()` 函数。

- 与隐藏帧/`iframe` 的例子相比有几处不同。首先，不需要主页面之外的 JavaScript 代码。这很重要，因为任何时候将代码保存在两个地方，总是可能出现不兼容的情况；在基于帧的例子中，依赖于分别位于显示页面和隐藏帧中的脚本之间的通信。通过将 `GetCustomerInfo.php` 改为只返回你感兴趣的数据，就可以消除在这些位置之间调用 JavaScript 的潜在问题。第二个不同是，当请求执行出现问题时很容易获知。在前一个

例子中，当请求处理过程出现错误时，没有一个机制来标识和响应服务器端错误。使用 XMLHttp，所有的服务器端错误都将展现给开发人员，可以给用户提供一个有意义的错误反馈。针对页面中的 HTTP 请求，XMLHttp 是比隐藏帧技术更优秀的解决方案。

- 4. XMLHttp 的 POST 请求

- 现在你已经知道 XMLHttp 是如何处理 GET 请求了，接下来看看如何处理 POST 请求。首先，必须以修改 GetCustomerInfo.php 的方式对 SavaCustomer.php 做出同样的修改，也就意味着需要去除无关的 HTML 和 JavaScript，添加内容类型信息，并将其输出到文本中：

- <?php
- header("Content-Type: text/plain");
- \$sName = \$_POST["txtName"];
- \$sAddress = \$_POST["txtAddress"];
- \$sCity = \$_POST["txtCity"];
- \$sState = \$_POST["txtState"];
- \$sZipCode = \$_POST["txtZipCode"];
- \$sPhone = \$_POST["txtPhone"];
- \$sEmail = \$_POST["txtEmail"];
- \$sStatus = "";
- \$sDBServer = "your.database.server";
- \$sDBName = "your_db_name";
- \$sDBUsername = "your_db_username";
- \$sDBPassword = "your_db_password";
- \$sSQL = "Insert into

Customers(Name, Address, City, State, Zip, Phone, `E-mail`) ".

- " values (' \$sName', ' \$sAddress', ' \$sCity', ' \$sState', ' \$sZipCode' ".
- ", ' \$sPhone', ' \$sEmail')";
- \$oLink = mysql_connect(\$sDBServer, \$sDBUsername, \$sDBPassword);
- @mysql_select_db(\$sDBName) or \$sStatus = "Unable to open database";
- if(\$oResult = mysql_query(\$sSQL)) {
- \$sStatus = "Added customer; customer ID is ".mysql_insert_id();

- } else {
- \$sStatus = "An error occurred while inserting; customer not saved.";
- }
- mysql_close(\$oLink);
- echo \$sStatus;
- ?>
- 上面就是 SavaCustomer.php 完整的代码。注意，调用函数 header() 是用来设置

内容类型，而 echo 命令则是用来输出变量 \$sStatus 的值。

- 在主页面中，只是设置一个让用户输入新客户信息的简单表单，如下所示：
- <form method="post" action="SaveCustomer.php"
- onsubmit="sendRequest(); return false">
- <p>Enter customer information to be saved:</p>
- <p>Customer Name: <input type="text" name="txtName" value="" />

- Address: <input type="text" name="txtAddress" value="" />

- City: <input type="text" name="txtCity" value="" />

- State: <input type="text" name="txtState" value="" />

- Zip Code: <input type="text" name="txtZipCode" value="" />

- Phone: <input type="text" name="txtPhone" value="" />

- E-mail: <input type="text" name="txtEmail" value="" /></p>
- <p><input type="submit" value="Save Customer Info" /></p>
- </form>
- <div id="divStatus"></div>

• 你会发现 onsubmit 事件处理函数现在改为调用 sendRequest() 函数（尽管事件处理函数仍然返回 false，以阻止实际的表单提交）。该方法首先为 POST 请求组装数据，然后创建一个 XMLHttpRequest 对象来发送这个请求。该数据必须按下列的查询字符串格式来发送：

- name1=value1&name2=value2&name3=value3
- 每个参数的名字和值都必须转化为 URL 编码格式，以避免在传输过程中丢失数据。

JavaScript 提供了一个名为 encodeURIComponent() 的内建函数，通过它可以完成这个

编码转换。为了创建这个字符串，需要遍历表单的所有字段，提取名字和值并对其进行编码转换。这些将交由 `getRequestBody()` 函数来处理：

```
• function getRequestBody(oForm) {  
•   var aParams = new Array();  
•   for (var i=0 ; i < oForm.elements.length; i++) {  
•     var sParam = encodeURIComponent(oForm.elements[i].name);  
•     sParam += "=";  
•     sParam += encodeURIComponent(oForm.elements[i].value);  
•     aParams.push(sParam);  
•   }  
•   return aParams.join("&");  
• }
```

• 该函数假定将一个表单的引用作为参数传入。它将创建一个数组（`aParams`）来存储每个名字—值数据对。然后遍历表单的每个元素，针对每个元素生成一个字符串并存储到 `sParam` 变量中，再将该变量添加到数组中。这样做可以避免多次字符串连接操作，这种操作在一些浏览器中会降低代码的执行速度。最后一步则是调用数组的 `join()` 方法，并将符号 `&` 作为参数传入，这实际上是通过符号 `&` 将所有的名字—值对组合起来，生成格式正确的一个字符串。

• 字符串连接操作对于大多数浏览器而言都是一个代价很高的操作，这是因为字符串是不可变的，即一旦创建就不能够改变其值。因此要连接两个字符串，首先要创建一个新的字符串，然后将两个字符串的内容复制到其中。重复这种操作过程，将会使服务器的处理速度下降。正是因为这个原因，最好只对较小的字符串进行连接操作，而对于较长的字符串则使用 `array` 的 `join()` 方法。

```
• sendRequest()函数将调用 getRequestBody()并配置请求：  
• function sendRequest() {  
•   var oForm = document.forms[0];  
•   var sBody = getRequestBody(oForm);  
•   var oXmlHttp = XMLHttpRequest.createRequest();  
•   oXmlHttp.open("post", oForm.action, true);
```

- `oXmlHttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");`
- `oXmlHttp.onreadystatechange = function () {`
- `if (oXmlHttp.readyState == 4) {`
- `if (oXmlHttp.status == 200) {`
- `saveResult(oXmlHttp.responseText);`
- `} else {`
- `saveResult("An error occurred: " + oXmlHttp.statusText);`
- `}`
- `}`
- `};`
- `oXmlHttp.send(sBody);`
- `}`

与前面的例子一样，该函数的第一步也是获取表单的引用并将其存到一个变量（`oForm`）中。然后，将生成的请求主体存到变量 `sBody` 中。接下来则是创建和配置 `XMLHttp` 对象。注意，现在 `open()` 函数的第一个参数不是 `get` 而是 `post`，第二个参数变成了 `oForm.action`（该脚本可以用在多个页面）。你可能还会注意到已经对请求首部进行了设置。当表单从浏览器传送到服务器时，将会把请求的内容类型设置为 `application/x-www-form-urlencoded`。大多数服务器端语言都需要这种编码格式，才能够按照它对收到的 `POST` 数据进行正确解析，因此对其进行设置是很重要的。

而 `onreadystatechange` 事件处理函数则与 `GET` 请求例子中的十分类似；唯一的变化是将调用的函数从 `displayCustomerInfo()` 改为 `saveResult()`。最后一行是很重要的，它将 `sBody` 变量的内容作为字符串传给 `send()` 函数，这样它将成为请求主体的一部分。这有效地模拟了浏览器的操作，因此所有的服务器端程序逻辑将能够按预期意愿完成工作。

• 5. XMLHttp 的优点和缺点

显然，你能够感受到使用 `XMLHttp` 替代隐藏帧技术来实现客户端—服务器通信的优点。编写的代码很清晰，而且代码的意图也比使用隐藏帧中大量的回调函数更易于理解。不仅可以访问请求和响应首部，还能够访问 `HTTP` 状态码，这使你可以判断出请求处理是否成功。

- 不利的一面是，它不像隐藏帧技术，当发出调用时并没有浏览器的历史记录保存下来。浏览器的后退和前进按钮并没有和 XMLHttpRequest 请求绑定在一起，因此将会使其失去效用。正是因为这个原因，许多 Ajax 应用程序将 XMLHttpRequest 和隐藏帧技术结合使用，以生成一个更加可用的用户界面。

- 另一个缺点只体现在 IE 上，它要求必须启用 ActiveX 控件。如果用户将你的页面设置为特定的安全区域^[11]，该区域禁用 ActiveX 控件，这将使得你无法访问 XMLHttpRequest 对象。在这种情况下，可能只能使用隐藏帧技术。

- ^[11]. 即 IE 菜单的“工具”à“Internet 选项”à“安全”设置中的安全区域，包括 Internet、本地 Intranet、受信任站点及受限制的站点四种。——译者注

• 2.3 进一步考虑

- 无论你决定使用隐藏帧还是 XMLHttpRequest，在构建 Ajax 应用程序时还有一些事情必须考虑。将 JavaScript 的角色延伸到服务器端程序逻辑中，能够带来一些强大的功能，但是也存在一些 Web 开发人员必须警惕的缺陷。

• 2.3.1 同源策略

- 由于 Web 浏览器是运行在用户的计算机上，浏览器厂商采取了一些安全限制技术，以防止恶意的代码对用户的机器造成破坏。对于 JavaScript 领域而言，最重要的安全限制是同源策略（same origin policy），它用来决定服务器能够相互通信的页面。

- 通常源（origin）是指诸如 www.wrox.com 的单个域，它可以通过单个协议访问，通常是 HTTP。同源策略规定只有同源的页面才能访问、下载，以及与来自该源的资源进行交互（使用 JavaScript）。对于隐藏帧技术而言，这要求所有的帧都是从相同的源装载的，因此可以使用 JavaScript 来通信。如果你尝试在帧中装载来自于其他源的页面，就不能够与这个页面进行交互或者访问其任何脚本。同源策略的意图是防止恶意程序员在合法的网页之外获取你的信息。

- 同源策略对于 XMLHttpRequest 的工作也是有影响的。使用 XMLHttpRequest 就不能访问与运行该代码的页面不同源的资源。这就意味着，默认情况下是不能够在 open() 方法中使用以 http://开头的 URL，只能使用同一域名中的绝对 URL 或相对 URL。如果你需要访问一个位于不同源中的 URL，就必须创建一个服务器端代理来处理这个通信（参见图 2-4）。

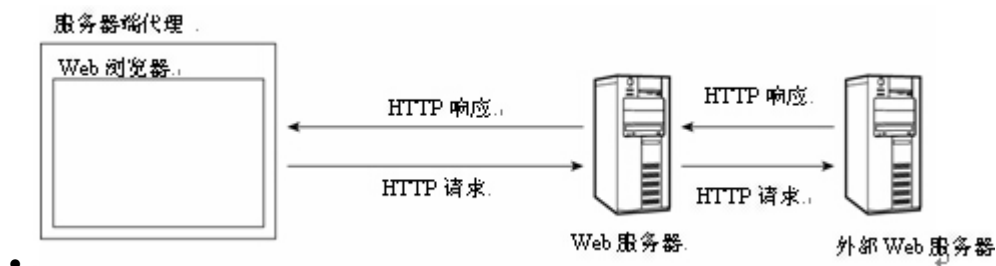


图 2-4

- 使用一个服务器端代理，浏览器可以将请求发给 Web 服务器。而 Web 服务器则与非本域中的另一台 Web 服务器联系，以请求适当的信息。当你的 Web 服务器收到响应时，就会把响应转发给浏览器。其结果就是外部数据可以进行无缝传输。在本书的后面部分中将会使用服务器端代理。

- IE 并未提供一个显式的同源策略，而是依赖于其自己的安全区域来决定能够访问什么。属于 Internet 安全区域的那些页面通常遵循的规则与同源策略相似，而属于受信任站点区域的页面则会免受该策略的限制。

• 2.3.2 缓存控制

- 只要处理对相同页面的重复调用，就必须考虑浏览器的缓存机制。在暗地里，Web 浏览器会将已经下载和显示的网站存到缓存中，以提高其访问速度。对于经常访问的网站而言，这样会使访问速度大幅提高，但对于频繁更新的页面也会带来问题。如果你发出几个 Ajax 调用，必须意识到缓存机制可能会带来问题。

- 解决缓存机制带来的问题的最好方法是在服务器往浏览器发送的数据前面加上一个表示 no-cache 的首部，它的设置方法是：

- Cache-Control: no-cache

- 这将告诉浏览器，对于来自该 URL 中的数据不做缓存处理。浏览器将总是从服务器调用一个新的版本，而不是从其缓存中找一个已保存的版本。

• 2.4 小结

- 本章介绍了几种针对客户端—服务器通信的 Ajax 技术。我们从 HTTP 入门开始，研究了 HTTP 请求和响应。在此学到了 HTTP 消息的格式以及 GET 请求和 POST 请求之间的区别，并且还引入了首部和消息主体的概念。

- 你学到的第一个 Ajax 技术是隐藏帧技术，它使用一个宽度或高度为 0 的帧，使其有效地对用户隐藏。该技术使用 JavaScript 调用隐藏帧来实现客户端—服务器通信，并学到了如何使用隐藏帧技术来发送 GET 和 POST 请求。

- 紧接着，讲述了如何用隐藏的 i frame 来代替隐藏帧技术。由于 i frame 可以使用 JavaScript 动态地创建，在现代浏览器中或许是一种更理想的初始化客户端—服务器通信的方法。尽管 i frame 使得页面设计时更灵活，但其使用技术和隐藏帧是一样的。

- 本章还介绍了如何通过 XMLHttpRequest 来实现客户端—服务器通信。IE、Mozilla Firefox、Safari 及 Opera 都以某种形式对 XMLHttpRequest 对象提供了支持，但各有所不同，检查这些不同还需要一些额外的代码。我们还解释了以异步或同步模式发送请求的区别，并使你掌握了如何使用 XMLHttpRequest 来发送 GET 和 POST 请求。同时，还介绍了如何结合使用请求、响应首部以及 HTTP 状态码来更好地处理请求。

- 最后，讨论了初始化客户端—服务器时要考虑的其他问题。你了解了同源策略以及它与其他服务器通信将会产生什么影响。同时还讨论了不同的安全限制，并简要介绍了服务器端代理。另外，还阐述了在创建 Ajax 函数时对缓存进行控制的重要性。

• 3.1 通信控制模式

- 从第 2 章中，你已经掌握了如何用 JavaScript 与服务器进行通信。现实的问题是：启动并连续向服务器发送请求的最佳方法是什么？有些情况下，最好是从服务器预载入一些信息，以便能够快速响应用户的操作；而在另外一些情况下，可能想在不同的时间间隔内，向服务器发送数据或从服务器接收数据。或许所有的东西并非必须立刻下载，而可能按一个特定的顺序来下载。对于客户端与服务器的通信，Ajax 提供了精细的控制，能够实现预想的行为。

• 3.1.1 预先获取

- 在传统的 Web 解决方案中，应用程序并没有下一步要做什么的想法。页面中包含一些链接，每一个链接都指向网站的不同部分。这也可以称为“按需获取”（fetch on demand），根据用户的操作，服务器就可以准确地获知需要获取的数据。虽然从一开始 Web 就是以这种范式定义的，但这对于“开始—结束”的用户交互模式而言则有不利的一面，但有了 Ajax 的帮助，就可以改变这一局面。

- 预先获取（Predictive Fetch）模式的概念很简单，实现却不容易：Ajax 应用程序必须猜测用户下一步要做什么，然后获取相应的数据。最理想的情况是，最好总是知

道用户下一步将要做什么，并且确得下一步的数据当需要时就已经准备好了。但是在现实中，判断用户的后续操作却只是一个根据你的意愿的猜谜游戏。

- 也有一些使用场景是更容易预测用户下一步操作的。假设你正在阅读一篇分为三页的在线文章。其潜在的逻辑就是如果你对阅读第一页感兴趣，那么就会对阅读第二页和第三页感兴趣。因此如果第一页已经载入了几秒钟（通过超时来判断是很简单的），那么在后台下载第二页就是很安全的。同样，如果第二页载入了几秒钟，同样的逻辑可以推测读者会继续阅读第三页。当这些额外的数据已经下载并且缓存在客户端，那么用户就几乎可以在点击“下一页”按钮后立即看到下一页的内容。

- 另一个简单的使用场景是撰写电子邮件的过程。大多数情况下，你所撰写的电子邮件将发送给一个你认识的人，因此可以假定该人的信息已经存在于你的地址簿中。要想提供帮助，可以在后台预先载入你的地址簿并提供建议。该方法在许多基于 Web 的电子邮件系统中经常采用，包括 Gmail 和 AOLWebmail。关键还是“合理假设”准则。预测并预载入与用户可能的下一步相关的信息，可以使应用程序更轻快、反应更迅速；使用 Ajax 获取与任何可能的下一步相关的信息，那么很快会使服务器超负荷运转，使浏览器陷入额外的处理中。经验表明，只有从逻辑上确认该信息是用户下一步请求必需的，才预先获取它。

- 3.1.2 页面预载入实例

- 正如前面所提到的，预先获取模式的最简单也是最多的应用是预载入在线文章的后续页。随着 Weblog（或简称为 blog）的出现，好像所有的人都迷上了发表文章，都在自己的网站写作。在线阅读长文章对眼睛来说是件费力的事，因此大多数网站都会将其分成多个页。这样有利于阅读，但会使得内容的载入时间更长，因为要对每个新页面都需要处理其格式、菜单以及原始页中的广告。而预先获取模式则可以在读者还在阅读第一页时就在客户端和服务器载入下一页面中的文本信息。

- 最开始，我们需要一个处理页面预装载的服务器端逻辑。下面的 ArticleExample.php 文件中就包含了在线显示文章的代码：

- ```
<?php
$page = 1;
$dataOnly = false;
if (isset($_GET["page"])) {
 $page = (int) $_GET["page"];
```

- }
- if (isset(\$\_GET["data"]) && \$\_GET["dataonly"] == "true") {
- \$dataOnly = true;
- }
- if (!\$dataOnly) {
- ?>
- <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
- "<http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd>">
- <html xmlns="<http://www.w3.org/1999/xhtml>" xml:lang="en" lang="en">
- <head>
- <title>Article Example</title>
- <script type="text/javascript" src="xml.js"></script>
- <script type="text/javascript" src="Article.js"></script>
- <link rel="stylesheet" type="text/css" href="Article.css" />
- </head>
- <body>
- <h1>Article Title</h1>
- <div id="divLoadArea" style="display: none"></div>
- <?php
- \$output = "<p>Page ";
- for (\$i=1; \$i < 4; \$i++) {
- \$output .= "<a href='\"ArticleExample.php?page=\$i\"' id='\"aPage\$i\"'";
- if (\$i==\$page) {
- \$output .= "class='\"current\"'";
- }
- \$output .= ">\$i</a> ";
- }
- echo \$output;
- }
- if (\$page==1) {



- echo \$page1Text;
- } else if (\$page == 2) {
- echo \$page2Text;
- } else if (\$page == 3) {
- echo \$page3Text;
- }
- if (!\$dataOnly) {
- ?>
- </body>
- </html>
- <?php
- }
- ?>

• 默认情况下，该文件将显示文章内容的第一页。如果指定了查询字符串参数 `page` 的值，例如 `page=2`，那么将显示文章的指定页。当查询字符串中包含 `dataonly=true` 时，则页面只输出一个包含文章指定页内容的 `<div>` 元素，与 `page` 参数结合使用，可以获取文章中的任意一页。

• 在该页面中的 HTML 为文章标题预留了位置，同时还包括一个用来载入额外页面的 `<div>` 元素。这个 `<div>` 元素首先将 `display` 属性设置为 `none`，以确保不会显示意外的内容。紧接其后的 PHP 代码所包含的程序逻辑就是负责输出该文章的所有页码。在本例中，有三页内容，因此在顶部输出了三个链接（如图 3-1 所示）。

• 当前页码的 CSS 类指定为 `current`，因而用户知道现在查看的是哪一个页面。该类在 `Article.css` 中定义为：



• 图 3-1

- a.current {
- color: black;
- font-weight: bold;
- text-decoration: none;
- }

• 当读者阅读一个指定页时，指向该页的链接将变成黑色、粗体，并且去掉了下划线，这样就清晰地标识出了其当前所阅读的页。默认情况下，这些链接所调用的是相同的页，只不过修改了查询字符串中的 page 参数的值；大多数网站都是采用这种方法来处理多页文章的。但是，使用预先获取模式可以改进用户体验，提高访问的速度。

- 在本例中，实现预先获取模式还需要设置一些 JavaScript 全局变量：
- var oXmlHttp = null; //XMLHttp 对象
- var iPageCount = 3; //总页数
- var iCurPage = -1; //当前显示页
- var iWaitBeforeLoad = 5000; //载入新页前等待的时间（单位为 ms）
- var iNextPageToLoad = -1; //要下载的下一页

• 第一个变量是全局的 XMLHttp 对象，它用来发送所有请求以获得更多的信息。第二个参数是 iPageCount，它用来存储该文章的总页数（这里采用了硬性编码，但实际应用中它应该是生成的）。变量 iCurPage 则用来存储当前显示给用户的页码。接下来的两个变量直接处理数据预载入：iWaitBeforeLoad 表示在载入下一页之前等待的毫秒数，

而 `iNextPageToLoad` 则表示当指定时间结束应该载入的页码。在这个例子中, 5 秒钟(5000 毫秒) 之后就会载入新页, 这个时间足够让读者阅读文章的前几句内容, 并判断是否有兴趣阅读剩下的内容。如果读者在 5 秒钟之内就离开了, 说明他对文章的其他部分没有兴趣。

• 要完成这个处理, 首先需要有一个函数来确定将要获取的指定页面的 URL。这个 `getURLForPage()` 函数有一个参数, 用来指定你想要获取的页码。然后, 将提取当前 URL 并将 `page` 参数附在最后:

- `function getURLForPage(iPage) {`
- `var sNewUrl = location.href;`
- `if (location.search.length > 0) {`
- `sNewUrl = sNewUrl.substring(0, sNewUrl.indexOf("?"))`
- `}`
- `sNewUrl += "?page=" + iPage;`
- `return sNewUrl;`
- `}`

• 该函数首先从 `location.href` 中抽取 URL, 但它是针对该页面的完整 URL, 可能包括其中的查询字符串。接着测试这个 URL, 通过 `location.search` 的值是否大于 0 (`location.search` 返回的只是查询字符串, 如果有一个指定, 则将包含? 符号) 来判断是否有查询字符串。如果有查询字符串, 则使用 `substring()` 方法将其去除。然后将 `page` 参数附到 URL 后面并将其返回。这个函数将在许多不同的地方应用。

• 接下来的函数是 `showPage()`, 你或许猜得到, 它负责显示文章的下一页:

- `function showPage(sPage) {`
- `var divPage = document.getElementById("divPage" + sPage);`
- `if (divPage) {`
- `for (var i=0; i < iPageCount; i++) {`
- `var iPageNum = i+1;`
- `var divOtherPage = document.getElementById("divPage" + iPageNum);`
- `var aOtherLink = document.getElementById("aPage" + iPageNum);`
- `if (divOtherPage && sPage != iPageNum) {`
- `divOtherPage.style.display = "none";`

- `aOtherLink.className = "";`
- `}`
- `}`
- `divPage.style.display = "block";`
- `document.getElementById("aPage" + sPage).className = "current";`
- `} else {`
- `location.href = getURLForPage(parseInt(sPage));`
- `}`
- `}`

该函数首先检查指定页是否已经载入了一个`<div>`元素，`<div>`元素将以 `divPage` 加上页号进行命名（例如，`divPage1` 为第一页，`divPage2` 为第二页，等等）。如果这个`<div>`元素已经存在，那么说明该页已经预先获取了，因此只需切换当前显示的页即可。我们将遍历所有的页，然后将除了 `sPage` 参数中指定的页之外的其他页都隐藏。同时，将每个页的链接的 CSS 类指定为空字符串。然后，将当前页面的`<div>`元素的 `display` 属性设置为 `block` 以显示它，然后再将该页面链接的 CSS 类设置为 `current`。

如果`<div>`元素不存在，那么将通过获取 URL（使用前面定义的 `getURLForPage()` 函数）和赋给 `location.href`，以传统方式访问下一页。当用户在 5 秒钟之内点击页面的连接，其用户体验就与传统的 Web 范式相同了。

`loadNextPage()` 函数用来在后台载入每个新的页面。该函数确保执行对有效页的请求，同时保证按顺序并在指定的时间间隔获取页面：

- `function loadNextPage() {`
- `if (iNextPageToLoad <= iPageCount) {`
- `if (!oXmlHttp) {`
- `oXmlHttp = zXmlHttp.createRequest();`
- `} else if (oXmlHttp.readyState != 0) {`
- `oXmlHttp.abort();`
- `}`
- `oXmlHttp.open("get", getURLForPage(iNextPageToLoad)`
- `+ "&dataonly=true", true);`
- `oXmlHttp.onreadystatechange = function () {`

- //更多代码
- };
- oXmlHttp.send(null);
- }
- }

• 该函数首先通过与 iPageCount 进行比较, 以确保存在 iNextPageToLoad 中的页码是有效的。通过了这个检查后, 下一步就是检查全局 XMLHttpRequest 对象是否已经创建。如果没有, 则使用 XMLHttpRequest 库的 createRequest() 方法来创建。如果已经实例化, 则检查 readyState 属性确保其值为 0。如果 readyState 不为 0, 那么必须调用 abort() 方法来重新设置 XMLHttpRequest 对象。

• 接下来, 调用 open() 方法, 指明请求将为异步的 GET 请求。使用 getURLForPage() 函数来获取该 URL, 然后附上字符串 "&dataonly=true", 确保只返回页面的文本信息。设置完这些信息后, 则转到 onreadystatechange 事件处理函数。

• 在本例中, onreadystatechange 事件处理函数负责获取文章的文本内容, 同时创建相应的 DOM 结构以显示它:

- function loadNextPage() {
- if (iNextPageToLoad <= iPageCount) {
- if (!oXmlHttp) {
- oXmlHttp = XMLHttpRequest.createRequest();
- } else if (oXmlHttp.readyState != 0) {
- oXmlHttp.abort();
- }
- oXmlHttp.open("get", getURLForPage(iNextPageToLoad)
- + "&dataonly=true", true);
- oXmlHttp.onreadystatechange = function () {
- if (oXmlHttp.readyState == 4) {
- if (oXmlHttp.status == 200) {
- var divLoadArea = document.getElementById("divLoadArea");
- divLoadArea.innerHTML = oXmlHttp.responseText;
- var divNewPage = document.getElementById("divPage"

- + iNextPageToLoad);
- divNewPage.style.display = "none";
- document.body.appendChild(divNewPage);
- divLoadArea.innerHTML = "";
- iNextPageToLoad++;
- setTimeout(loadNextPage, iWaitBeforeLoad);
- }
- }
- };
- oXmlHttp.send(null);
- }
- }

正如上一章中所阐述的,将检查 `readyState` 属性什么时候等于 4,以及检查 `status` 属性确保没有错误。当传入这两个条件后,就会开始实际的处理。首先,获取载入区域的 `<div/>` 元素的引用,并将其存于 `divLoadArea` 变量中。然后,将请求的 `responseText` 赋给载入区域的 `innerHTML` 属性。由于返回的文本信息是一个 HTML 片段,将其解析后将创建相应的 DOM 对象。接下来,获取包含下一页内容(将 `divPage` 加上 `iNextPageToLoad` 就可以获取其 ID)的 `<div/>` 元素的引用,以及将 `<div/>` 元素的 `display` 属性设置为 0,以确保其移出载入区域时隐藏起来。紧接的后一句将 `divNewPage` 附加到文档的主体,为了便于使用将其放到正式的视图区域。然后将载入区域的 `innerHTML` 属性设置为空字符串,为载入下一页做好准备。在此之后,当指定的时间间隔过后 `iNextPageToLoad` 变量的值将加 1,并且超时时间值也将重新设置为原值,以便再次调用该函数。该函数将每 5 秒钟执行一次,直到所有的页面都载入为止。

• 由于该页面没有 JavaScript 也能工作,因此在确定浏览器支持 XMLHttpRequest 后再附加这些代码。幸运的是,在 `zXml` 库中, `zXmlHttp` 对象提供了一个名为 `isSupported()` 的函数,它可以用来完成这种检查:

- window.onload = function () {
- if (zXmlHttp.isSupported()) {
- //在此开始编写 Ajax 代码
- }

- };

- 这个代码块之中是预先获取模式的代码,这样就可以确保浏览器不支持 XMLHttpRequest,也不会因功能不完整的代码而对可用性产生负面影响。

- 在对文章设置预先获取模式时的第一步是决定用户当前阅读的是哪一页。要实现这一目标,必须检查 URL 的查询字符串,看是否指定了 page 参数。如果有,则可以从其中获取页码;否则就可以假设页码为 1(默认值):

- window.onload = function () {
- if (zXmlHttp.isSupported()) {
- if (location.href.indexOf("page=") > -1) {
- var sQueryString = location.search.substring(1);
- iCurPage =
- parseInt(sQueryString.substring(sQueryString.indexOf("=")+1));
- } else {
- iCurPage = 1;
- }
- iNextPageToLoad = iCurPage+1;
- //更多代码
- }
- };

- 在这段代码中,将测试页面的 URL(通过 location.href 可访问)是否指定了 page=。如果有,则使用 location.search(它只返回查询字符串,它的最前面是一个 "?",可以通过调用 substring(1)去除这个符号)来获得其查询字符串。紧接着的一行则是用来获取查询字符串中 "=" 号之后的内容(也就是页码),然后使用 parseInt()将其转成整型,并将结果存入 iCurPage 变量中。另外,如果在查询字符串中没有指定 page 参数,则可以假设它是第一页,因此将 iCurPage 变量值赋为 1。这段代码的最后一行,则是将 iNextPageToLoad 变量的值设置为当前页码加 1,确保不会重新载入已经存在的数据。

- 下一步则是重载处理页面链接的函数。记住,默认情况下这些链接将指向相同的页面,只是查询字符串设置为将要显示的页码。如果支持 XMLHttpRequest,就需要重载其行为并替换为调用 Ajax 功能的函数:

- window.onload = function () {

- if (zXmlHttp.isSupported()) {
- if (location.href.indexOf("page=") > -1) {
- var sQueryString = location.search.substring(1);
- iCurPage =

parseInt(sQueryString.substring(sQueryString.indexOf("=")+1));

- } else {
- iCurPage = 1;
- }
- iNextPageToLoad = iCurPage+1;
- var colLinks = document.getElementsByTagName("a");
- for (var i=0; i < colLinks.length; i++) {
- if (colLinks[i].id.indexOf("aPage") == 0) {
- colLinks[i].onclick = function (oEvent) {
- var sPage = this.id.substring(5);
- showPage(sPage);
- if (oEvent) {
- oEvent.preventDefault();
- } else {
- window.event.returnValue = false;
- }
- }
- }
- }
- }
- setTimeout(loadNextPage, iWaitBeforeLoad);
- }
- };

• 在这里，将使用 `getElementsByTagName()` 来获取链接（<a/>元素）的集合。如果该链接拥有一个以 `aPage` 开头的 ID，那么这是一个页面链接并且需要指定；这将使用 `indexOf()` 并检查其值是否为 0 来确定。其值为 0 将表明 `aPage` 是该字符串的第一部分。接下来，将为链接指定一个 `onclick` 事件处理函数。在这个事件处理函数中，页码将使



用链接的 ID（通过 `this.id` 可访问）来获取，并使用 `substring()` 来返回 `aPage` 后的信息。然后，这个值将传给本节前面定义的 `showPage()` 函数，它将显示相应的页。在此之后，还必须考虑如何去除链接的默认行为，即指向一个新页。由于这在 IE 和 DOM 事件模型中存在一些区别，因此需要一个 `if` 语句来确定采用的是什么样的操作。如果已经向函数传入了 `event` 对象（参数 `oEvent`），那么就说明是一个兼容 DOM 的浏览器，可以通过调用 `preventDefault()` 方法来屏蔽默认的行为。但如果 `oEvent` 的值为 `null`，那么意味着浏览器是 IE，需要通过 `window.event` 来访问这个 `event` 对象。这时就需要通过将 `returnValue` 属性设置为 `false` 来屏蔽默认行为，这是 IE 所采用的方法。

- 当恰当地处理了这些链接后，则为启动 `loadNextPage()` 创建了一个超时值。5 秒钟之后将第一次调用该函数，而后每过 5 秒钟再自动执行一次。
- 当你自己测试这个功能时，应在不同的时间段点击页面链接。如果在 5 秒钟之内点击，会看到页面转到了一个新的 URL，其查询字符串发生了变化。下一次，你在约 10 秒钟后点击页面链接，你会发现文字信息发生了变化，但 URL 没变（同时会发现其响应速度明显比转向一个 URL 要快得多）。

### • 3.1.3 提交节流

- 预先获取是一种从服务器获取数据的模式；Ajax 解决方案的另一面是向服务器发送数据。由于想避免页面刷新，这对于发送用户数据时显得更为重要。在传统的网站或 Web 应用程序中，每次点击都会向服务器发送一个请求，因此服务器总是知道客户端的行为。而在 Ajax 模型中，用户与网站或应用程序交互时，并非每一次点击都会产生请求。

- 一种解决方案是向传统的 Web 解决方案一样，在用户每次操作时都向服务器发送数据。因此，当用户输入一个字母时，该字母就会立即发给服务器。在输入每个字母时都将重复这个处理过程。这种方法所存在的问题是，在很短的时间内可能创建大量的请求，这不仅可能引起服务器端的问题，也可能在每个请求发出和处理的过程中使得用户界面变得很慢。提交节流（Submission Throttling）设计模式则是解决这个问题的另一种方法。

- 使用提交节流模式，可以将要发送到服务器端的数据存入客户端的缓存中，然后在预定的时间一次性发送数据。大名鼎鼎的 Google Suggest 就是这样的一个例子。它并没有在输入每个字符时发送请求，而是等待一个特定的时间后再将文本框中所有当前

的字符一次性发送到服务器。从输入到发送的处理过程进行了精细的调整，使用户根本感觉不到延迟。提交节流在一定程度上提高了 Google Suggest 的速度。

- 提交节流通常要么在网站或应用程序第一次装载时，要么当出现一个特定的用户操作时开始。紧接着，将调用一个客户端的函数来对数据进行缓冲处理。时常检查用户的状态看其是否处于空闲 (idle) 状态 (以避免与用户界面冲突)。如果用户仍然在操作，那么继续收集数据。当用户空闲了，则说明他已不再执行操作，就该决定是否发送数据了。确定的方法取决于具体的使用场景；或者希望收集到的数据达到一定数量才发送，或者每当用户空闲时就发送。在数据发送之后，应用程序通常将继续收集数据，直到收到了服务器的响应或者一些用户操作中中止了数据收集的过程。图 3-2 描述了这个过程。

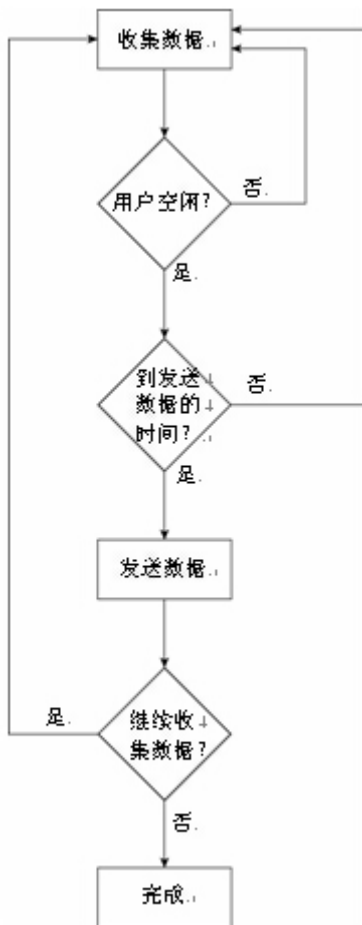


图 3-2

- 提交节流模式一般从不用于关键任务的数据。如果数据必须在一个特定的时间范围内传送到服务器，最好还是采用传统的表单，以确保信息准确及时的传送。

- 

### 3.1.4 表单增量验证实例

- 如上所述，提交节流模式可以通过不同的用户交互来实现。当使用表单时，增量上传用户输入的数据有时是很用的。最常见的使用场景是对用户在表单上填写的数据进行实时验证，而非等到最后再来检查所有的错误。在这种情况下，你最有可能使用表单中每个元素的 `onchange` 事件处理函数来决定什么时候来上传其数据。

- 当 `<select/>` 元素选择了另一个选项，其他控件的值改变并失去焦点时，都会启动 `change` 事件。例如，如果你在一个文本框中输入一些字符，然后点击屏幕的其他地方（使得文本框失去了焦点），那么就将启动 `change` 事件，同时将调用 `onchange` 事件处理函数。如果你再次点击这个文本框，然后再点其他地方（或按下 `Tab` 键），那么该文本框会失去焦点但不会启动 `change` 事件，因为其内容没有变化。对提交节流模式而言，使用这个事件处理函数可以避免产生无关的请求。

- 通常，表单的验证要优先于提交。一开始表单的提交按钮是禁用的（`disable`），只有填入表单的所有字段已经通过服务器验证，才将其启用（`enable`）。例如，假设你访问的网站中有一个需要注册才能够使用的功能时。这可能是一个购物网站，需要注册后才能购买商品，或者是一个只允许注册用户访问消息公告（`message board`）的网站。当创建一个新账号时，可能要求提供以下信息：

- ◎ 不重复的用户名；
- ◎ 有效的电子邮件地址；
- ◎ 填写的生日必须是有效的日期。
- 当然，不同的使用场景需要的数据类型是不同的，但这些内容为大部分应用程序提供了一个良好的开始。

- 创建这种交互的第一步是定义一个用来收集这些信息的 HTML 表单。该表单可以独立使用，即使在不支持 `Ajax` 调用的情况下也一样可以使用：

- `<form method="post" action="Success.php">`

- `<table>`

- <tr>
- <td><label for="txtFirstName">First Name</label></td>
- <td><input type="text" id="txtFirstName" name="txtFirstName" /></td>
- </tr>
- <tr>
- <td><label for="txtLastName">Last Name</label></td>
- <td><input type="text" id="txtLastName" name="txtLastName" /></td>
- </tr>
- <tr>
- <td><label for="txtEmail">E-mail</label></td>
- <td><input type="text" id="txtEmail" name="txtEmail" /></td>
- </tr>
- <tr>
- <td><label for="txtUsername">Username</label></td>
- <td><input type="text" id="txtUsername" name="txtUsername" /></td>
- </tr>
- <tr>
- <td><label for="txtBirthday">Birthday</label></td>
- <td><input type="text" id="txtBirthday" name="txtBirthday" />
- (m/d/yyyy)</td>
- </tr>
- <tr>
- <td><label for="selGender">Gender</label></td>
- <td><select id="selGender"

```
• name="sel Gender"><option>Male</option></option>Female</option></select></td>
```

```
• </tr>
```

```
• </table>
```

```
• <input type="submit" id="btnSignUp" value="Sign Up!" />
```

```
• </form>
```

• 在这个表单中需要注意一些事件。首先，并非所有的字段都需要使用 Ajax 调用进行验证。例如名字、姓和性别字段（以组合框形式表示）都不需要验证，而其他字段，包括电子邮件、用户名和生日字段都需要使用 Ajax 来进行验证。其次，你会发生所有的这些字段对应的文本框后面都包含一个隐藏的图像，该图像仅在验证失败时使用。该图像在开始时是隐藏的，如果浏览器不支持 Ajax 功能则永远都看不到它们。在该表单中完全没有 JavaScript，所有的函数和事件处理函数都定义在独立的文件中。

• 我们将通过一个名为 `validateField()` 的函数来验证每个字段。由于每个字段都使用相同的验证技术（向服务器发出调用并等待响应），因此是可行的。唯一的区别是什么样类型的数据需要验证，以及在验证失效时显示哪一个图像。

• 服务器端程序存放在一个名为 `ValidateForm.php` 的文件中。该文件预设为通过查询字符串来获取名字—值数据对，其中名字是要验证其值的控件名字，而值则是该控件的值。根据控件的名称，该页面将对值进行相应的验证。然后，以下述格式返回一个简单的字符串：

```
• <true|false>||<error message>
```

• 该字符串的第一部分用来表示该值是否有效（`true` 表示有效，`false` 表示无效）。在两个管道符（`|`）之后的第二部分则是一个错误消息，只当值无效时才提供。以下是一些可能返回的字符串的实例：

```
• true||
```

```
• false||Invalid date.
```

• 第一行表示这是一个有效值；第二行则表示这是一个无效值。

• 这里使用的是纯文本格式的消息，而在本书的后面部分将会讲述使用其他数据格式来实现该方法的方法，诸如 XML 和 JSON。

• 用来实现验证功能的代码如下所示：

```
• <?php
```

- \$valid = "false";
- \$message = "An unknown error occurred.";
- if (isset(\$\_GET["txtUsername"])) {
- //载入用户名数组
- \$usernames = array();
- \$usernames[] = "SuperBlue";
- \$usernames[] = "Ninja123";
- \$usernames[] = "Daisy1724";
- \$usernames[] = "NatPack";
- //检查用户名
- if (in\_array(\$\_GET["txtUsername"], \$usernames)) {
- \$message = "This username already exists. Please choose another.";
- } else if (strlen(\$\_GET["txtUsername"]) < 8) {
- \$message = "Username must be at least 8 characters long.";
- } else {
- \$valid = "true";
- \$message = "";
- }
- } else if (isset(\$\_GET["txtBirthday"])) {
- \$date = strtotime(\$\_GET["txtBirthday"]);
- if (\$date < 0) {
- \$message = "This is not a valid date.";
- } else {
- \$valid = "true";
- \$message = "";
- }
- } else if (isset(\$\_GET["txtEmail"])) {
- if(!ereg(
- ""^[\_a-z0-9-]+(\.[\_a-z0-9-]+)\*@[a-z0-9-]+(\.[a-z0-9-]+)\*(\. [a-z]{2,3})\$

",

- `$_GET["txtEmail"]))) {`
- `$message = "This e-mail address is not valid";`
- `} else {`
- `$valid = "true";`
- `$message = "";`
- `}`
- `}`
- `echo "$valid||$message"; ?>`

- 在这个文件中，第一步是确定哪些字段要验证。这可过使用 `isset()` 函数检查 `$_GET` 数组是否有值来完成。如果特定字段有值，那么就开始验证。对于用户名而言，则先检查其值是否已经存储于用户名数组中，然后再检查它是否满足至少 8 个字符的要求。生日则直接传给 PHP 内建的 `strtotime()` 函数，它将把任何 U.S. 格式的日期字符串转成 UNIX 时戳（即从 1970 年 1 月 1 日到该日期的总秒数）。如果存在错误，则该函数将返回 -1，说明传入的字符串不是一个有效的日期。电子邮件地址则通过正则表达式来检查，确保其格式正确，这个正则表达式是由 John Coggeshall 设计的（参见他的文章“基于 PHP4 验证电子邮件”，可以在 [www.zend.com/zend/spotlight/ev12apr.php](http://www.zend.com/zend/spotlight/ev12apr.php) 中获得）。

- 注意在本例中，用户名是存储在一个简单的数组中，并以硬性编码的方式写在页面中。而在实际的实现中，用户名应该存储在一个数据库中，我们可以通过查询数据库来决定该用户名是否存在。

- `$valid` 和 `$message` 变量分别初始化为 `false` 和 `An unknown error occurred`，这样就可以确保如果该文件产生错误（例如传入了一个未认可的字段名），仍然能够返回表示否定的验证结果。但是如果验证通过，则需要将这两个变量设置为相应的值（即将 `$valid` 赋值为 `true`，`$message` 赋值为空字符串）。而如果验证失效，则只需要设置 `$message` 变量的值，因为 `$valid` 的值已经是 `false` 了。在该页面中的最后一步是以前面提到的格式输出这个字符串。

- 接下来，必须创建执行该验证的 JavaScript。只要知道哪个字段需要验证，就可以只通过一个 `validateField()` 函数来验证每个字段。为了消除跨浏览器兼容问题，还需要一些简单的处理：

- `function validateField(oEvent) {`
- `oEvent = oEvent || window.event;`

- `var txtField = oEvent.target || oEvent.srcElement;`
- `//更多代码`
- `}`

• 在这个函数中的前两行代码用来处理 IE 和 DOM 兼容浏览器(诸如 Mozilla Firefox、Opera 和 Safari) 之间事件模型的不同。对于 DOM 兼容 (DOM-compliant) 浏览器, 是向每个事件处理函数传递一个 event 对象, 引发这个事件的控件存储于该 event 对象的 target 属性中。而在 IE 中, event 对象是 window 对象的一个属性, 因此, 该函数中的第一行则用来为 oEvent 变量赋予正确的值。当对一个对象和一个 null 对象进行逻辑或 (|) 运算时, 将返回一个非空值。如果你使用 IE, 那么 oEvent 将为 null; 因此将把 window.event 的值赋给 oEvent。如果你使用的是 DOM 兼容的浏览器, 则 oEvent 将把自身的值再赋给自己。第二行代码对引发该事件的控件做相同操作, 因为在 IE 中该控件存放在 srcElement 属性中。在这两行代码执行后, 引发该事件的控件将存储在 txtField 变量中。下一步就该使用 XMLHttpRequest 来创建这个 HTTP 请求了:

- `function validateField(oEvent) {`
- `oEvent = oEvent || window.event;`
- `var txtField = oEvent.target || oEvent.srcElement;`
- `var oXmlHttp = new XMLHttpRequest();`
- `oXmlHttp.open("get", "ValidateForm.php?" + txtField.name + "="`
- `+ encodeURIComponent(txtField.value), true);`
- `oXmlHttp.onreadystatechange = function () {`
- `//更多代码`
- `};`
- `oXmlHttp.send(null);`
- `}`

• 与第 2 章一样, 你可以使用 XMLHttpRequest 库来获得跨浏览器兼容的 XMLHttpRequest 支持。XMLHttpRequest 对象将被创建并存储在 oXmlHttp 对象中。接下来, 使用 open() 来启动一个 GET 请求的连接。注意, ValidateForm.php 的查询字符串是由字段名称、等号以及字段的值组合而成的 (并且还将使用 encodeURIComponent() 对 URL 进行编码)。另外还要注意这是一个异步请求。对于这个应用而言, 这是十分重要的, 因为你并不想在对某个字段进行验证时对用户输入表单中其他信息的操作造成影响。紧记, 使用 XMLHttpRequest 对象发出同步请求



将在其执行过程中冻结用户界面（包括输入和点击操作）。该函数的最后一部分是处理来自服务器的响应：

```
• function validateField(oEvent) {
• oEvent = oEvent || window.event;
• var txtField = oEvent.target || oEvent.srcElement;
• var oXmlHttp = new XMLHttpRequest();
• oXmlHttp.open("get", "ValidateForm.php?" + txtField.name + "="
• + encodeURIComponent(txtField.value), true);
• oXmlHttp.onreadystatechange = function () {
• if (oXmlHttp.readyState == 4) {
• if (oXmlHttp.status == 200) {
• var arrInfo = oXmlHttp.responseText.split("|");
• var imgError = document.getElementById("img"
• + txtField.id.substring(3) + "Error");
• var btnSignUp = document.getElementById("btnSignUp");
• if (!eval(arrInfo[0])) {
• imgError.title = arrInfo[1];
• imgError.style.display = "";
• txtField.valid = false;
• } else {
• imgError.style.display = "none";
• txtField.valid = true;
• }
• btnSignUp.disabled = !isFormValid();
• } else {
• alert("An error occurred while trying to contact the server.");
• }
• }
• };
• oXmlHttp.send(null);
```

- }

- 当确认 `readyState` 和 `status` 的值都是正确的后，则用 JavaScript 的 `split()` 方法将 `responseText` 的值分解到一个字符串数组 (`arrInfo`) 中。`arrInfo` 的第一组值将是 PHP 变量 `$valid` 的值；第二组值则是 PHP 变量 `$message` 的值。同时也引用相应的错误图像并且返回 “Sign Up”（注册）按钮。错误图像可以通过分析字段名获得，移除最前面的 “txt”（使用 `substring()`），在前面加上 “img”，后面加上 “Error”（因此对于字段 “txtBirthday” 而言，其错误图像名将构造为 “imgBirthdayError”）。

- `arrInfo[0]` 的值必须传给 `eval()` 函数，以获得真正的布尔值。（紧记，这时它只是一个字符串：true 或 false。）如果该值是 false，那么错误图像的 `title` 属性将赋值为 `arrInfo[1]` 中存放的错误消息，并且将显示出错误图像，而为文本框定制的 `valid` 属性将设置为 false（这在稍后将用到）。如果一个值是无效的，那么将显示错误图像，当用户把鼠标移到图像上时，将显示出错误消息（参见图 3-3）。但如果该值是有效的，则图像仍然是隐藏的，而定制的 `valid` 属性也将设置为 true。

- 图 3-3

- 你也会发现在该函数中使用了 “Sign Up” 按钮。如果该表单中仍然存在无效数据，那么 “Sign Up” 按钮将会是禁用的。要实现这一功能，需要调用一个名为 `isFormValid()` 函数。如果该函数返回 false，那么将把 “Sign Up” 按钮的 `disable` 属性设置为 true，从而禁用它。`isFormValid()` 函数只是简单遍历表单中的每个字段，检查其 `valid` 属性。

- ```
function isFormValid() {  
    var frmMain = document.forms[0];  
    var blnValid = true;  
    for (var i=0; i < frmMain.elements.length; i++) {  
        if (typeof frmMain.elements[i].valid == "boolean") {  
            blnValid = blnValid && frmMain.elements[i].valid;  
        }  
    }  
    return blnValid;  
}
```

- 对于表单中的每个元素，首先将检查 `valid` 属性是否存在，这可以使用 `typeof` 操作符来实现。如果该属性存在将会返回一个布尔值。由于有些字段是不需要验证的（因此也不需设置定制的 `valid` 属性），因此只检查认为需要验证的字段。

- 该脚本的最后一部分是为文本框设置事件处理函数。这将在表单载入完成后，并且只有当具备 XMLHttpRequest 支持时（由于这里执行的是基于 Ajax 的验证）才进行：

- `//如果 Ajax 可用，则禁用提交按钮，并指定事件处理函数`
- `window.onload = function () {`
- `if (XMLHttpRequest.prototype.isSupported()) {`
- `var btnSignUp = document.getElementById("btnSignUp");`
- `var txtUsername = document.getElementById("txtUsername");`
- `var txtBirthday = document.getElementById("txtBirthday");`
- `var txtEmail = document.getElementById("txtEmail");`
- `btnSignUp.disabled = true;`
- `txtUsername.onchange = validateField;`
- `txtBirthday.onchange = validateField;`
- `txtEmail.onchange = validateField;`
- `txtUsername.valid = false;`
- `txtBirthday.valid = false;`
- `txtEmail.valid = false;`
- `}`
- `};`

- 这个 `onload` 事件处理函数将为每个文本框指派一个 `onchange` 事件处理函数，同时会将定制的 `valid` 属性初始化为 `false`。另外，将“Sign Up”按钮禁用可以避免提交无效的数据。但要注意，只有当具有 XMLHttpRequest 支持时才能禁用该按钮；否则它将成为一个普通的 Web 表单，而当整个表单要提交时都无法完成验证。

- 当你调入该例子页面时，在三个要验证的文本字段中修改了值并移到其他字段时，将会向服务器发出一个验证请求。使用提交节流模式的用户体验是很流畅，而且即使禁用了 JavaScript 或不支持 XMLHttpRequest，该表单的功能仍然是可用的。

- 即使使用了这种类型的验证，但实际上在整个表单提交时还需要对所有数据再进行一次验证。紧记，如果用户禁用了 JavaScript，在对这些数据操作前还要确保它们是有有效的。

3.1.5 字段增量验证实例

- 上面的例子是在字段的值发生改变时对每个字段进行验证，而提交节流设计模式的另一种流行用法是在修改某个字段过程中周期性地提交该字段。Bitflux LiveSearch 和 Google Suggest 都使用了这种提交节流模式，数据将随着用户的输入周期性地发送到服务器。在这些例子中，提交用来在服务器上实现搜索；而同样的方法也可以用来实现在用户输入时进行单个字段的验证。

- 假设在某个网站的注册过程中，并非要求你填入整个表单，而是需要你首先选择一个用户名（或许是多步注册过程中的第一步）。在这种情况下，你必须确保使用的是一个不存在的用户名。或许不希望等到提交整个表单时再验证，而是周期性地向服务器提交验证申请，确保在输入一个有效的用户名之前不提交整个表单。

- 注意，该例子只是为了演示。如果你想在实际的产品开发环境中使用这里描述的技术，必须提防木马程序（spam bot）使用该功能来获取用户名和密码。

- 针对该例子的表单很简单，只需要一个文本框和一个“Next”（下一步）按钮：
- `<form method="post" action="Success.php">`
- `<table>`
- `<tr>`
- `<td><label for="txtUsername">Username</label></td>`
- `<td><input type="text" id="txtUsername" name="txtUsername" />`
- `</td>`
- `</tr>`
- `</table>`
- `<input type="submit" id="btnNext" value="Next" />`
- `</form>`
- 可以看到它与前一个例子保持了相同的格式，包括一个隐藏的错误图像。接下来，只需对上一个例子中的 `validateField()` 函数进行少量修改就可以在此使用了：

- var oXmlHttp = null;
- var iTimeoutId = null;
- function validateField(oEvent) {
- oEvent = oEvent || window.event;
- var txtField = oEvent.target || oEvent.srcElement;
- var btnNext = document.getElementById("btnNext");
- btnNext.disabled = true;
- if (iTimeoutId != null) {
- clearTimeout(iTimeoutId);
- iTimeoutId = null;
- }
- if (!oXmlHttp) {
- oXmlHttp = new XMLHttpRequest();
- } else if (oXmlHttp.readyState != 0) {
- oXmlHttp.abort();
- }
- oXmlHttp.open("get", "ValidateForm.php?" + txtField.name + "="
- + encodeURIComponent(txtField.value), true);
- oXmlHttp.onreadystatechange = function () {
- if (oXmlHttp.readyState == 4) {
- if (oXmlHttp.status == 200) {
- var arrInfo = oXmlHttp.responseText.split("|");
- var imgError = document.getElementById("img"
- + txtField.id.substring(3) + "Error");
- if (!eval(arrInfo[0])) {
- imgError.title = arrInfo[1];
- imgError.style.display = "";
- txtField.valid = false;
- } else {
- imgError.style.display = "none";

- `txtField.valid = true;`
- `}`
- `btnNext.disabled = !txtField.valid;`
- `} else {`
- `alert("An error occurred while trying to contact the server.");`
- `}`
- `}`
- `};`
- `iTimeoutId = setTimeout(function () {`
- `oXmlHttp.send(null);`
- `}, 500);`
- `};`

• 在这个修改后的函数中首先要注意的是两个全局变量：`oXmlHttp` 和 `iTimeoutId`。首先，`oXmlHttp` 用来保存对多次使用的 `XMLHttp` 对象的全局引用（而在前一个例子中，该对象只使用了一次）；其次，`iTimeoutId` 用来保存延迟发送请求的超时时间标识符。在这个函数中，第一处更新是将“Next”按钮设置为立即禁用。这一步很重要，因为当调用该函数时不会立即发送请求。接下来一处更新则是判断超时时间值是否为 `null`，如果不是则将其清空，这是避免连续发送太多的请求（如果有待处理请求，则取消它）。

• 接下来则检查全局变量 `oXmlHttp` 对象是否为 `null`。如果是，则创建一个新的 `XMLHttp` 对象，并将其赋给该全局变量。如果已经有了 `XMLHttp` 对象，那么则检查它的 `readyState` 属性是否准备发送一个请求。正如上一章提到的，当调用 `open()` 方法时，则 `readyState` 的值将从 0 变为 1；因此，只要 `readyState` 的值不是 0 就说明请求已经开始了，因此在尝试发送一个新请求前，必须调用 `abort()` 方法取消前一个请求。可以发现用来验证的 `ValidateForm.php` 页面是相同的。

• 在 `onreadystatechange` 事件处理函数中，只有一行新代码基于用户名的有效性来修改“Next”按钮的禁用状态。在该函数的结尾处，调用了 `setTimeout()` 函数用来将请求的发送延迟半秒（500 毫秒）。该调用返回的标识符将保存到 `iTimeoutId` 中，因此在下一次调用该函数时可以取消这次请求。以这种形式使用 JavaScript 的超时功能，可以确保用户至少在半秒钟内没有输入任何新的值。如果用户输入很快，则该超时时间将被不断清空，请求也将被取消。只有当用户暂停输入时，该请求才会最终发送出去。

• 现在只剩下设置事件处理函数了。由于该方法是根据用户的输入来上传信息的，因此你不能够再只依赖于 `onchange` 事件处理函数了（尽管仍然需要）。在本例中，需要使用 `onkeyup` 事件处理函数，它将在每次按下键并放开键时调用。

```
• window.onload = function () {  
• if (zXmlHttp.isSupported()) {  
• var btnNext = document.getElementById("btnNext");  
• var txtUsername = document.getElementById("txtUsername");  
• btnNext.disabled = true;  
• txtUsername.onkeyup = validateField;  
• txtUsername.onchange = validateField;  
• txtUsername.valid = false;  
• }  
• };
```

• 同样，这与上一个例子也很类似，只需要修改按钮的名称（现在是 `btnNext`）以及将 `validateField()` 赋给 `onkeyup` 事件处理函数。当用户输入时，将会对用户名称进行验证。每当输入有效的用户名，“Next”按钮都将启用。每当发出一个请求时，该按钮将首先禁用以适应特殊情况。这种特殊情况可能出现在当用户输入一个有效的用户名后继续输入，这些后输入的额外字符使得用户名变成了无效的用户名，而你并不希望提交这个无效的数据。

• 尽管字段增量验证是一个很好的功能，但由于它会带来大量的请求因此应该降低使用。除非配置服务器时考虑了这些增量的请求，那么上述方法是不错的选择。

• 3.1.6 定期刷新

• 定期刷新（Periodic Refresh）设计模式描述了一种在指定时间间隔内检查服务器是否有新消息的过程。该方法也称为轮询（polling），要求浏览器知道什么时候向服务器发送另一个请求。

• 在 Web 应用中，该模式有多种不同的使用方法：

• © ESPN 使用定期刷新模式来自动更新在线的记分牌。例如，<http://sports.espn.go.com/nfl/scoreboard> 上的 NFL 记分牌，可以随着 NFL 比赛的进行实时显示比分和图表。使用 `XMLHttp` 对象和少量的 `Flash`，当有新信息时该页面就会自己不断更新。

- ©Gmail (<http://gmail.google.com>)在收到新邮件时，使用定期刷新模式来通知用户。当你在阅读电子邮件或执行其他操作时，Gmail 会不断地检查服务器，看是否收到了新邮件，除非有新邮件否则就不会通知用户。当收到新邮件时将会在 Inbox（收件箱）菜单项的后面括号中显示收到的新邮件数量。

- ©XHTML Live Chat (www.plasticshore.com/projects/chat/) 使用定期刷新模式来实现一个基于简单 Web 技术的聊天室。每隔几秒钟，该聊天室就会检查服务器上是否有新的信息，以实现文本信息的自动更新。如果有新消息，则会更新页面以反映其变化，从而创建一个传统的聊天室体验。

- © Magnetic Ajax 演示程序(www.broken-notebook.com/magnetic/)创建了在线版本的 magnetic poetry^[1]的体验（使用能够重组成为句子的一个个单词）。完整版本将会在每几秒钟重新排列一次，因此如果有其他人在重新排序句子，则你会马上看到。

- 显然，使用定期刷新模式还有许多不同的改进用户体验的方法，但其基本的意图都是一样的：提示用户信息已更新。

- ^[1]. 这是指 Magnetic Poetry 公司开发的 Magnetic Poetry Kit，它是利用 Java 语言编写的，将屏幕当作一块白板，其上散布着一堆像磁块一样的文字，用户可以任意排列文字，组成词句。——译者注

• 3.1.7 新评论提示实例

- 从 2005 年初开始，网上的 blog 就引入了新评论提示（New Comment Notifier）这一个新功能。新评论提示实际上就是意味着：当新的评论添加进来后，它将会提醒用户。它可能以在页面上显示简单文本信息的形式，或通过视图外的幻灯片动画性信息的形式来实现，但其基本想法是一样的。在这个例子中，定期刷新模式通过检查包含评论的数据库表，以寻找最新的评论。

- 假设你定义了一个如下所示的 MySQL 数据库表：

- CREATE TABLE `BlogComments` (

- `CommentId` INT NOT NULL AUTO_INCREMENT ,

- `BlogEntryId` INT NOT NULL ,

- `Name` VARCHAR(100) NOT NULL ,

- `Message` VARCHAR(255) NOT NULL ,

- `Date` DATETIME NOT NULL ,

- PRIMARY KEY (`CommentId`)

-) COMMENT = 'Blog Comments';

- 将运行下面的 SQL 查询语句:

- select CommentId, Name, LEFT(Message, 50)

- from BlogComments order by Date desc

- limit 0,1

- 该查询语句将返回最新评论的 ID (自动生成的)、评论者名称, 以及消息内容中前 50 个字符 (使用 LEFT() 函数)。这 50 个字符将作为实际评论的预览 (由于内容比较长, 因此你可能不想获得整条消息)。

- 执行该查询的页面称为 CheckComments.php, 它将以下列格式输出一个字符串:

- <comment ID>||<name>||<message>

- 对于该格式, 只需使用 JavaScript 的 Array.split() 方法就可以简单地获取信息的各个独立部分。如果没有评论或出现错误, 那么评论的 ID 将为 -1, 而字符串的其他部分将为空白字符。以下是 CheckComments.php 的完整代码清单:

- <?php

- header("Cache-control: No-Cache");

- header("Pragma: No-Cache");

- //数据库信息

- \$sDBServer = "your.database.server";

- \$sDBName = "your_db_name";

- \$sDBUsername = "your_db_username";

- \$sDBPassword = "your_db_password";

- //创建 SQL 查询字符串

- \$sSQL = "select CommentId, Name, LEFT(Message, 50) as ShortMessage from

- BlogComments order by Date desc limit 0,1";

- \$oLink = mysql_connect(\$sDBServer, \$sDBUsername, \$sDBPassword);

- @mysql_select_db(\$sDBName) or die("-1|| || ");

- if(\$oResult = mysql_query(\$sSQL) and mysql_num_rows(\$oResult) > 0) {

- \$aValues = mysql_fetch_array(\$oResult, MYSQL_ASSOC);

- echo \$aValues['CommentId']. "||". \$aValues['Name']. "||".

- \$aValues[' ShortMessage'];
- } else {
- echo "-1|| || ";
- }
- mysql_free_result(\$oResult);
- mysql_close(\$oLink);
- ?>

• 这个文件中最重要的部分或许就是最开始的两个 header 语句。通过将 Cachecontrol 和 Pragma 的值设置为 No-Cache，可以告诉浏览器将一直从服务器上而非客户端的缓存中获取该文件。如果没有这个设置，有些浏览器就可能重复地返回相同的信息，使该功能完全失效。该文件中剩余部分的代码看起来很熟悉，它与前面利用 MySQL 数据库调用的例子一样，使用了相同的算法。

• 也可以在每次针对该文件的请求时修改查询字符串来避开缓存问题。通常的做法是取一个时间戳附加到查询字符串中，这样就可以使浏览器认为你是要从服务器上获取一个新的版本。

- 接下来，JavaScript 就可以调用这个文件了。最开始，还再次需要一些全局变量：
- var oXmlHttp = null; //XMLHttp 对象
- var iInterval = 1000; //检查的时间间隔（单位为毫秒）
- var iLastCommentId = -1; //上次接收的评论的 ID
- var divNotification = null; //显示提示信息的层

• 与通常一样，第一个全局变量是名为 oXmlHttp 的 XMLHttp 对象，它将在所有请求中使用。第二个全局变量是 iInterval，它为每次检查新评论设置一个间隔时间，单位是毫秒。在本例中设置为 1000 毫秒，也就是 1 秒，当然这是可以根据自己的需要设置的。接下来，iLastCommentId 变量用来将最新评论的 ID 存储在数据库中，通过将该值与最新接收到的评论的 ID 相比较，从而判断是否有新的评论加到了数据库中。最后一个变量是 divNotification，它保存一个到<div/>元素的引用，该元素用来向用户显示新评论提示。

• 当检查到有新的评论时，则将与新评论相关的信息填入到 divNotification 中，包括评论者名称、消息的摘要以及查看完整评论的链接。如果还没有创建<div/>元素，则必须先创建该元素然后赋予相应的信息：

- `function showNotification(sName, sMessage) {`
- `if (!divNotification) {`
- `divNotification = document.createElement("div");`
- `divNotification.className = "notification";`
- `document.body.appendChild(divNotification);`
- `}`
- `divNotification.innerHTML = "New Comment
" + sName`
- `+ " says: " + sMessage + "...
<a href=\"ViewComment.php?id=\""`
- `+ iLastCommentId + "\">View";`
- `divNotification.style.top = document.body.scrollTop + "px";`
- `divNotification.style.left = document.body.scrollLeft + "px";`
- `divNotification.style.display = "block";`
- `setTimeout(function () {`
- `divNotification.style.display = "none";`
- `}, 5000);`
- `}`

正如你所见，`showNotification()` 函数将接受两个参数：名称和消息。但是，在使用这些信息之前，必须确保 `divNotification` 的值不是 `null`。必要的话，还将创建新的 `<div/>` 元素，并在其添加到文档主体之前将其 CSS 类设置为 `notification`。在此之后，使用 `innerHTML` 属性来设置提示用的 HTML 片段，内容就是粗体的“New Comment”，后面是名字、消息以及查看该消息的链接。该链接将指向 `ViewComment.php`，并将 `iLastCommentId` 的值作为查询字符串中参数 `id` 的值，它指向的就是要查看的评论。然后，使用 `document.body` 的 `scrollTop` 和 `scrollLeft` 属性来设置提示的位置。这将确保提示信息会出现在页面的左上角，而不管滚动条的位置（如果你滚动到了右下角）。接着，将 `display` 属性设置为 `block`，使该提示可见。

- 该函数的最后一部分是设置一个超时时间，在 5 秒（5000 毫秒）之后隐去这个提示。在屏幕上保留这种提示信息并不是一个好主意，除非你为其做了特殊的设计；否则可能会遮盖重要的信息。

- 在本例中，CSS 类 `notification` 的定义为：

- `div.notification {`

- border: 1px solid red;
- padding: 10px;
- background-color: white;
- position: absolute;
- display: none;
- top: 0px;
- left: 0px;
- }

• 这将创建一个红边白底的文本框。当然，可以根据你的网站或应用程序的风格来设置这里的格式。这个例子中最重要的部分是将 position 设置为 absolute 以及将 display 设置为 none。通过设置这些属性可以确保当把<div/>元素添加到该页面时，不会中断正常的页面流程或移动原来的元素，其结果是生成如图 3-4 所示的提示区域。



• 图 3-4

• 让我们回到 JavaScript 代码。完成大部分工作的函数是 checkComments()，它负责检查服务器上的信息并完成更新。其代码与前面的例子十分类似：

- function checkComments() {
- if (!oXmlHttp) {
- oXmlHttp = new XMLHttpRequest();
- } else if (oXmlHttp.readyState != 0) {
- oXmlHttp.abort();

- }
- oXmlHttp.open("get", "CheckComments.php", true);
- oXmlHttp.onreadystatechange = function () {
- if (oXmlHttp.readyState == 4) {
- if (oXmlHttp.status == 200) {
- var aData = oXmlHttp.responseText.split("||");
- if (aData[0] != iLastCommentId) {
- if (iLastCommentId != -1) {
- showNotification(aData[1], aData[2]);
- }
- iLastCommentId = aData[0];
- }
- setTimeout(checkComments, iInterval);
- }
- }
- };
- oXmlHttp.send(null);
- }

• 该函数创建了一个 XMLHttpRequest 对象，然后以异步模式调用 CheckComments.php。代码中重要的部分已经突出显示出来了（其余的代码与前面例子中的代码基本相同）。在本节中，使用 split() 方法将 responseText 分解到数组中。该数组中的第一个值 aData[0]，就是最后添加的评论的 ID。如果与最后保存的评论的 ID 不相等，就需要提示用户了。另外，如果最后的评论的 ID 的值为 -1，则说明没有接收到评论的 ID，因此也就不必进行提示了。如果最后的评论的值不是 -1，说明至少接收到一个评论的 ID，因为它与从服务器接收到的不同，所以将显示一些提示信息。然后，把新的 ID 赋给 iLastCommentId，供以后使用。在事件处理函数的最后一步是为 checkComments() 设置另一个超时时间，用来检查更多的评论。

• 这个过程最后一步是在页面载入时调用 checkComments() 函数。它将从数据库中获取最近评论的 ID，但不显示相应的提示信息（因为 iLastCommentId 的初值等于 -1）。当下一次调用 checkComments() 时，基于存储在 iLastCommentId 中的值检查从数

数据库中获取的 ID 来决定是否显示提示信息。与前面一样，该功能只有当浏览器支持 XMLHttpRequest 时才能启动：

- `window.onload = function () {`
- `if (zXmlHttp.isSupported()) {`
- `checkComments();`
- `}`
- `};`
- 这就是创建定期刷新解决方案的全部工作。只需要记住，在想实现类似该功能的

页面中包含必要的 JavaScript 和 CSS 文件。

• 本例的文件可以从 www.wrox.com 下载。同时下载的文件是用于测试的、可添加和浏览评论的其他页面。

• 3.1.8 多阶段下载

• Web 上的一个永久性问题就是页面下载的速度。当每个用户使用的都是 56Kbps 的调制解调器时，Web 设计者必须关注其页面的“重量”（即页面的总字节数）。随着家庭宽带网的流行，许多网站已经升级，包含了多媒体信息、更多的图片以及更多的内容。这种方法，在为用户提供更多信息的同时，也使得下载速度越慢，而且每个东西都好像以随机的顺序在载入。幸运的是，针对这个问题存在一种 Ajax 解决方案。

• 多阶段下载 (Multi-Stage Download) 是一个载入页面的 Ajax 模式。在完成之后，页面会开始下载其他将显示在页面上的组件。如果用户在所有组件下载完成之前离开页面，则它们将不再重要。但是如果用户要在该页面停留更长一段时间（或许在阅读一篇文章），那么将在后台载入其他功能，以便该功能在用户需要时可用。这对于开发人员而言，最主要的优点是可以决定什么时候下载什么。

• 这是一个比较新的 Ajax 模式，它因微软的 start.com 而流行。当你第一次访问 start.com 时，它只是一个中间有一个搜索文本框的简单页面。但在幕后，开始发出一系列的请求以向页面中填入更多的内容。几秒钟后，随着来自不同地方的内容显示出来，页面也变得生动起来。

• 多阶段下载模式虽然很好，但也存在一个缺点：对于不支持 Ajax 技术的浏览器，页面就只有最简单的表单可以使用。也就是，没有额外的下载，所有基本的功能也必须能够正常工作。处理这类问题的典型方法是提供优雅的降级 (degradation) 方案，也

就是对于支持 Ajax 技术的浏览器可以获得更丰富的用户界面，而对于不支持的浏览器则将获得一个简单的、不加渲染的用户界面。如果希望搜索引擎能够找到你的网站，那么这就更加重要；由于这些 bot 程序不支持 JavaScript，它们仅能依靠页面中的 HTML 来确定网站的值。

3.1.9 附加信息链接实例

- 当在线阅读一篇文章时，通常需要提供与进一步阅读该主题的附加信息链接。这里的关键问题是：主要内容是什么？显然文章内容是页面中的主要内容，因此在最初载入该页面时必须先下载这部分信息。而附加信息链接不太重要，因此应该后一步载入。这个例子将说明如何创建这种解决方案。

- 首先，要对一个显示文章的页面进行布局。对于本例而言，下面是一个很简单的布局：

- `<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"`
- `"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">`
- `<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">`
- `<head>`
- `<title>Article Example</title>`
- `<script type="text/javascript" src="xml.js"></script>`
- `<script type="text/javascript" src="Article.js"></script>`
- `<link rel="stylesheet" type="text/css" href="Article.css" />`
- `</head>`
- `<body>`
- `<h1>Article Title</h1>`
- `<div id="divAdditionalLinks"></div>`
- `<div id="divPage1">`
- `<!-- article content here -->`
- `</div>`
- `</body>`
- `</html>`

• 这段 HTML 代码最重要的部分是 ID 为 `divAdditionalLinks` 的 `<div/>` 元素。它是用来放置用于下载该文章的附加链接的容器。默认情况下，其格式是右对齐、不可见：

- `#divAdditionalLinks {`
- `float: right;`
- `padding: 10px;`
- `border: 1px solid navy;`
- `background-color: #cccccc;`
- `display: none;`
- `}`

• 由于将 CSS 的 `display` 属性设置为 `none` 非常重要，这使得该空的 `<div/>` 元素不会占用页面的空间。如果没有这个设置，将会在文章的右边看到一个很小的空方框。

• 与前一个例子不同，下载的内容只是包含链接和标题的文本文件中的纯文本。这个 `AdditionalLinks.txt` 文件包含一些简单的 HTML 代码：

- `<h4>Additional Information</h4>`
- ``
- `Wrox`
- `NCZOnline`
- `XWeb`
- ``

• 这个文件可以使用服务端程序逻辑动态地创建，但对于该实例的用途，静态内容就足够用了。

• 完成工作的 JavaScript 很简单，也与本章前几个例子十分相似：

- `function downloadLinks() {`
- `var oXmlHttp = XMLHttpRequest.createRequest();`
- `oXmlHttp.open("get", "AdditionalLinks.txt", true);`
- `oXmlHttp.onreadystatechange = function () {`
- `if (oXmlHttp.readyState == 4) {`
- `if (oXmlHttp.status == 200) {`
- `var divAdditionalLinks =`
- `document.getElementById("divAdditionalLinks");`

- `divAdditionalLinks.innerHTML = oXmlHttp.responseText;`
- `divAdditionalLinks.style.display = "block";`
- `}`
- `}`
- `}`
- `oXmlHttp.send(null);`
- `}`
- `window.onload = function () {`
- `if (zXmlHttp.isSupported()) {`
- `downloadLinks();`
- `}`
- `};`
- 完成该任务的函数是 `downloadLinks()`，只有当浏览器支持 `XMLHttp`，并且页面已经完全装载之后才调用该函数。在 `downloadLinks()` 函数中的代码也是前面一直使用的标准 `XMLHttp` 算法。当从 `AdditionalLinks.txt` 中获取内容后，将使用 `innerHTML` 属性将内容放置到占位符 `<div>` 中。最后一步则是将 `<div>` 元素的 `display` 属性设置为 `block`，以使其结果可见。最终的结果如图 3-5 所示。



• 图 3-5

- 如果浏览器不支持 `XMLHttp`，包含附加链接的区域将永远不会显示，因此第一段将会像其他段落一样展开。

- 这个技术可以在同一个页面的多个部分中多次使用，显然也不会限于在最初的页面载入，完成后每次只能更新一个部分。你可以为每个请求创建一个新的 XMLHttpRequest 对象，然后相继地发送各个请求，也可以按顺序地发送——等收到前一个请求的响应后再发送下一个请求。这些选择完全取决于你及其预期的功能。

• 3.2 失效处理模式

- 前面几节在提到处理向服务器发送或从服务器接收的数据时，都假设服务器端的每件事都能够按计划进行：接收到请求，做出必要的修改，然后将相应的响应发给客户端。但如果服务器端出错会发生什么呢？或者更糟糕一些，请求没有发送到服务器又会怎样呢？当你开发 Ajax 应用程序时，必须预先对这些问题进行考虑，并决定遇到这些问题时应用程序要怎么做。

• 3.2.1 取消待处理的请求

- 如果在服务器端发生错误，也就意味着返回内容的状态值不是 200，则必须决定如何处理。假设遇到的是“file is not found”（文件未找到，404）或“internal server error occurred”（内部服务错误，500），由于这些错误只有管理员能够修复，因此在几分钟内重试是没有效果的。处理这种情况的最简单方法是取消所有待处理的请求。你可以在代码中的某处设置一个标志，说明“不要再发送请求了”。这显然对于使定期刷新模式的解决方案有很大的影响。

- 对于新评论提示的例子，可以通过修改来考虑该问题。这是一个 Ajax 解决方法为用户提供增值，而并非关注页面的例子。如果请求失败，不必向用户发出警告；你可以简单地取消后续的请求来避免更多的错误。要实现这一功能，必须添加一个全局变量，用来说明是否允许发送请求：

- `var oXmlHttp = null;`
- `var iInterval = 1000;`
- `var iLastCommentId = -1;`
- `var divNotification = null;`
- `var blnRequestsEnabled = true;`

- 现在，在发送请求之前必须先检查变量 `blnRequestsEnabled` 的值。这可以通过将 `checkComments()` 函数的主体封装到一个 `if` 语句中来实现：

- `function checkComments() {`

- if (blnRequestsEnabled) {
- if (!oXmlHttp) {
- oXmlHttp = zXmlHttp.createRequest();
- } else if (oXmlHttp.readyState != 0) {
- oXmlHttp.abort();
- }
- oXmlHttp.open("get", "CheckComments.php", true);
- oXmlHttp.onreadystatechange = function () {
- if (oXmlHttp.readyState == 4) {
- if (oXmlHttp.status == 200) {
- var aData = oXmlHttp.responseText.split("||");
- if (aData[0] != iLastCommentId) {
- if (iLastCommentId != -1) {
- showNotification(aData[1], aData[2]);
- }
- iLastCommentId = aData[0];
- }
- setTimeout(checkComments, iInterval);
- }
- }
- };
- oXmlHttp.send(null);
- }
- }

• 但工作还没有全部完成，还必须检查可能遇到的两种不同类型的错误：给出了状态码的服务器错误；服务器不可达错误（例如服务器宕机或因特网连接失效）。

• 首先，将最外层的 if 语句中的内容封装到 try...catch 程序块中。当服务器不可达时，不同的浏览器响应的的时间不同，但都会抛出错误。将这些请求封装在 try...catch 中，可以确保捕获其抛出的任何错误，在那里可以将 blnRequestsEnabled 设置为 false。接下来，对于服务器错误而言，可以在状态不等于 200 时抛出一个自定义的错误。这时

try...catch 程序块将捕获这个错误，其效果与服务器不可达的错误一样（将 `blnRequestsEnabled` 设置为 `false`）：

```
• function checkComments() {  
•   if (blnRequestsEnabled) {  
•     try {  
•       if (!oXmlHttp) {  
•         oXmlHttp = zXmlHttp.createRequest();  
•       } else if (oXmlHttp.readyState != 0) {  
•         oXmlHttp.abort();  
•       }  
•       oXmlHttp.open("get", "CheckComments.php", true);  
•       oXmlHttp.onreadystatechange = function () {  
•         if (oXmlHttp.readyState == 4) {  
•           if (oXmlHttp.status == 200) {  
•             var aData = oXmlHttp.responseText.split("||");  
•             if (aData[0] != iLastCommentId) {  
•               if (iLastCommentId != -1) {  
•                 showNotification(aData[1], aData[2]);  
•               }  
•               iLastCommentId = aData[0];  
•             }  
•             setTimeout(checkComments, iInterval);  
•           } else {  
•             throw new Error("An error occurred.");  
•           }  
•         }  
•       };  
•       oXmlHttp.send(null);  
•     } catch (oException) {  
•       blnRequestsEnabled = false;  
•     }  
•   }  
• }
```

- }
- }
- }

• 现在，不管出现哪种错误，都将抛出一个异常，并且变量 `blnRequestEnable` 的值将设置为 `false`，如果再次调用 `checkComments()`，将有效地取消后续的请求。

• 你可能还会发现，只有当状态为 200 时，才对另一个请求创建超时时间，它是为了防止遇到其他状态的请求。这对于服务器错误而言可以正常工作，但对于通信错误则无能为力。因此遇到这类错误时，通常最好有多种处理错误的方法。

• 3.2.2 重试

• 处理错误的另一种选择是静静地重试，其重试可以设置为一个时间段，也可能设置为特定的次数。再次强调，除非 Ajax 功能对于用户体验而言很关键，否则不要在出现错误时提示用户。处理这类问题的最好方法是在幕后进行，直到问题解决。

• 为了解释重试（Try Again）模式，我们来回顾一下多阶段下载的例子。在该例子中，将下载额外的链接并将其显示在文章的边上。如果在该请求期间遇到错误，那么大多数浏览器将弹出一个错误消息。用户不知道出了什么错，或什么导致了错误，因此显示这种消息根本就是烦人？更好的方法是，在放弃请求之前，在幕后多尝试几次信息的下载。

- 要记录尝试失败的次数，还需要有一个全局变量：

- `var iFailed = 0;`

• 变量 `iFailed` 最开始的值为 0，每次请求失败，它的值都将加 1。因此，当 `iFailed` 超过了一个特定的值时，可以取消这个请求，因为它显然是无法工作的。例如，如果你想在取消所有待处理请求之前尝试 10 次，那么程序类似于：

- `function downloadLinks() {`
- `var oXmlHttp = new XMLHttpRequest();`
- `if (iFailed < 10) {`
- `try {`
- `oXmlHttp.open("get", "AdditionalLinks.txt", true);`
- `oXmlHttp.onreadystatechange = function () {`
- `if (oXmlHttp.readyState == 4) {`

```

• if (oXmlHttp.status == 200) {
•   var divAdditionalLinks =
•   document.getElementById("divAdditionalLinks");
•   divAdditionalLinks.innerHTML = oXmlHttp.responseText;
•   divAdditionalLinks.style.display = "block";
• } else {
•   throw new Error("An error occurred.");
• }
• }
• }
• }
• oXmlHttp.send(null);
• } catch (oException) {
•   iFailed++;
•   downloadLinks();
• }
• }
• }

```

• 这段代码的结构与前面的例子很相似。使用 `try...catch` 代码块来捕获通信过程中可能遇到的错误，并且当 `status` 不是 200 时抛出一个自定义的错误。主要的区别在于，当捕获错误时变量 `iFailed` 的值自动增 1，并且还会再次调用 `downloadLinks()`。当变量 `iFailed` 的值小于 10 时（意味着失败的次数小于 10），会启动另一个请求来尝试下载。

• 通常，重试模式仅适用于每个请求只计划发送一次的场景，就像多阶段下载的例子一样。如果你尝试在诸如定期刷新等周期性发起请求的场景中使用该模式，最终会导致打开的请求数量不断增多，占用的内存也越来越大。

• 3.3 小结

• 在本章中，你了解了应用于 Ajax 解决方案中的不同设计模式。首先了解是如何使用预先获取模式，通过预先载入用户可能将要使用的信息来改善用户的体验。我们学会

了使用预先获取模式创建了一个例子，它在确定用户将阅读整篇文章时，在几秒钟后预先载入该文章的页面。

- 紧接着，讲述了提交节流模式，它是一种向服务器间断地传送增量数据，而不是一次性提交所有数据的方法。我们了解了如何使用该模式完成表单的数据验证，同时还讨论了它的兄弟定期刷新模式，它用来周期性地从服务器接收信息。使用定期刷新模式构建了一个例子，用于在一个 blog 或留言板中有一个新的评论时显示提示信息。

- 本章还介绍了多阶段下载模式，它是一种在页面已经载入之后继续下载额外信息的方法。这能够加快页面最初下载的时间，并可以根据情况以你认为合适的方法来控制后续请求的发送频率与顺序。

- 最后一节讨论的是失效处理模式，它用来处理客户端—服务器通信中的错误。我们了解到其中可能存在两种错误：服务器错误（诸如 404，文件没找到）或通信错误（无法连接服务器），并讲述了两种处理这类错误的模式：取消待处理的请求和重试。

- 随着 XML 技术的流行，Web 开发人员希望将此技术同时应用于 Web 服务器端和客户端，而不仅仅像过去那样只在 Web 服务器端才提供 XML 功能。从 IE 5.0 和 Mozilla 1.0 开始，微软和 Mozilla 在其各自的浏览器中通过 JavaScript 提供了对 XML 的支持。最近，Apple 和 Opera 也在他们的浏览器加入 XML 的部分支持，但还没有像微软和 Mozilla 提供的那么全面。浏览器制造商继续提供基于新特性的 XML 支持，这就为 Web 开发人员提供了强大的工具，而这些工具过去只存在于服务器端。

- 在本章中，你将学习到如何在 XML DOM 对象中载入和操作 XML 文档，使用 XPath 技术选择符合特定条件的 XML 节点，以及使用 XSLT 技术将 XML 文档转化为 HTML。

- 4.1 浏览器对 XML 的支持

- 今天大多数浏览器都支持 XML，但是能够像 IE、Mozilla Firefox 这样对 XML 及其相关技术支持得如此全面的却为数不多。尽管其他浏览器正在迎头赶上，如 Safari 和 Opera 现在就支持 XML 文档的浏览，但是这些支持要么功能不全面，要么存在错误。由于其他浏览器存在这样那样的问题，所以本章就以 IE 和 Firefox 上的实现为主进行阐述。

- 4.1.1 IE 中的 XML DOM

- 当微软在 IE 5.0 中第一次加入对 XML 支持时，他们只是在 MSXML ActiveX 库(最初是为了在 IE 4.0 中解析 Active Channels 的组件)中实现 XML 的功能。最初的版本并没有打算公开使用，然而随着开发人员逐渐了解这个组件并尝试使用时，微软才意识到这个库的重要性，很快就在 IE 4.01 中发布了 MSXML 完全升级版本。

- MSXML 最初还只是 IE 的一个组件。直到 2001 年，微软发布了 MSXML 3.0，这是一个通过其公司网站独立发布的产品。在 2001 年晚些时候，微软又发布了 MSXML 4.0，并且将其更名为微软 XML 核心服务组件。MSXML 从最初一个基本的、无校验功能的 XML 解析器，逐渐发展成一个功能强大的组件，能够校验 XML 文档，进行 XSL 转化，支持命名空间、XML 的简单 API (SAX)，以及 W3C XPath 和 XML Schema 标准，并且每个新版本都在性能上有一定的提升。

- 为了在 JavaScript 中创建 ActiveX 对象，微软实现一个新的 ActiveXObject 类，该类用来实例化 ActiveX 对象。ActiveXObject 类的构造函数包含一个字符串参数，该参数表示要创建的 ActiveX 对象的版本，在此指的就是 XML 文档的版本。第一个 XML DOM ActiveX 对象名为 Microsoft.Xml Dom，其创建方法如下所示：

- `var oXml Dom = new ActiveXObject("Microsoft.Xml Dom");`

- 这个新创建的 XML DOM 对象与其他 DOM 对象一样，可以用来遍历 DOM 树，操作 DOM 节点。

- 到本书截稿为止，MSXML DOM 文档共有五个不同的版本，分别是：

- `Microsoft.Xml Dom;`

- `MSXML2.DOMDocument;`

- `MSXML2.DOMDocument.3.0;`

- `MSXML2.DOMDocument.4.0;`

- `MSXML2.DOMDocument.5.0`。

MSXML 是基于 ActiveX 的实现，因此只能够在 Windows 平台上使用。在 Mac 平台上的 IE 5 是不提供 XML DOM 支持的。

因为存在五个不同版本，而你总是会使用最新版，所以使用一个函数来判断浏览器所使用的版本是相当有用的。这样就可以确保使用最新的 XML 支持，获取最佳的性能。下面的函数 `createDocument()` 将使你能够创建正确的 MSXML DOM 文档。

```
function createDocument() {
    var aVersions = [ "MSXML2.DOMDocument.5.0",
        "MSXML2.DOMDocument.4.0", "MSXML2.DOMDocument.3.0",
        "MSXML2.DOMDocument", "Microsoft.XmlDom" ];
    for (var i = 0; i < aVersions.length; i++) {
        try {
            var oXmlDom = new
ActiveXObject(aVersions[i]);
            return oXmlDom;
        } catch (oError) {
            // 不做任何处理
        }
    }
    throw new Error("MSXML is not installed.");
}
```

该函数遍历存放 MSXML DOM 文档的版本号的 `aVersions` 数组，从最新版本 `MSXML2.DOMDocument.5.0` 开始尝试创建 DOM 文档。如果成功创建对象，那么返回该对象且退出 `createDocument()`；否则 `try...catch` 语句将捕获

所抛出的异常，并继续下一次循环，尝试下一个版本。如果 **MSXML DOM** 文档创建失败，那么抛出异常，说明 **MSXML** 未安装。由于该函数不是一个类，所以用法与其他函数类似，都将返回一个值：

- `var oXml Dom = createDocument();`
- 使用 `createDocument()` 函数将确保程序使用最新的 **DOM** 文档。当创建了 **XML** 文档后，下一步就是载入 **XML** 数据。

- 1. 在 IE 中载入 XML 数据

- **MSXML** 支持两种载入 **XML** 的方法：`load()` 和 `loadXML()`。`load()` 方法从 **Web** 的指定位置载入一个 **XML** 文件。与 **XMLHttpRequest** 一样，`load()` 方法可以以同步或异步两种模式载入数据。默认情况下，`load()` 方法采用异步模式；如果要采用同步模式，那么必须将 **MSXML** 对象的 `async` 属性设置为 `false`，代码如下：

- `oXml Dom.async = false;`
- 当采用异步模式时，**MSXML** 对象公开了 `readyState` 属性，该属性和 **XMLHttpRequest** 的 `readyState` 属性一样，包含五种状态。
- 此外，**DOM** 文档支持 `onreadystatechange` 事件处理函数，可以监控 `readyState` 属性。因为异步模式是默认选项，因此将 `async` 属性设置为 `true` 是可选的：

- `oXml Dom.async = true;`
-
- `oXml Dom.onreadystatechange = function () {`
- `if (oXml Dom.readyState == 4) {`
- //当 document 完全载入后，进行某些操作
- `}`
- `};`
-
- `oXml Dom.load("myxml.xml");`

- 本示例中，将把虚构的、名为 myxml.xml 的 XML 文档载入到 XML DOM 文档中。当 readyState 值为 4 时，说明文档已经完全载入，则执行 if 语句中的代码。

- 第二种载入 XML 数据的方法是 loadXML()，该方法与 load() 方法的主要区别在于从字符串载入 XML，而不是根据指定的文件名载入 XML。该字符串必须包含正确格式的 XML，如下所示：

- var sXml = "<root><person><name>Jeremy McPeak</name></person></root>";
-
- oXml Dom. loadXML(sXml);
- 在此，oXml Dom 文档将载入 sXml 变量中包含的 XML 数据。loadXML() 方法不需要像 load() 方法那样检查 readyState 属性，也不需要设置 async 属性，因为该方法并不涉及服务器请求。

- 2. 在 IE 中遍历 XML DOM 文档

- XML DOM 文档的遍历与 HTML DOM 的遍历非常类似，因为它们都是节点层次的结构。节点树的最顶部是 documentElement 属性，包含文档的根元素。使用表 4-1 中所列出的属性，可以访问文档中任何元素或属性。

• 表 4-1 XML DOM 属性

| 属 性 | 描 述 |
|---------------|----------------------|
| attributes | 包含当前节点属性的数组 |
| childNodes | 包含子节点数组 |
| firstChild | 指向当前节点的第一个子节点 |
| lastChild | 指向当前节点的最后一个子节点 |
| nextSibling | 返回当前节点的下一个邻居节点 |
| nodeName | 返回当前节点的名字 |
| nodeType | 指定当前节点的 XML DOM 节点类型 |
| nodeValue | 包含当前节点的文本 |
| ownerDocument | 返回文档的根元素 |
| parentNode | 指向当前节点的父节点 |

| | |
|------------------------------|--------------------------------------|
| <code>previousSibling</code> | 返回当前节点的前一个邻居节点 |
| <code>text</code> | 返回当前节点的内容或当前节点及其子节点的文本（只有 IE 才支持的属性） |
| <code>xml</code> | 以字符串返回当前节点及其子节点的 XML（只有 IE 才支持的属性） |

- 遍历 DOM 文档并获取数据，是一个很直观的过程。让我们看看下面的 XML 文档：

- `<?xml version="1.0" encoding="utf-8"?>`
-
- `<books>`
- `<book isbn="0471777781">Professional Ajax</book>`
- `<book isbn="0764579088">Professional JavaScript for Web Developers</book>`
- `<book isbn="0764557599">Professional C#</book>`
- `<book isbn="1861002025">Professional Visual Basic 6 Databases</book>`
- `</books>`

- 这是一个简单的 XML 文档，包含一个根元素 `<books/>` 以及四个子元素 `<book/>`。以该文档为例，我们可以研究 DOM 的细节。DOM 树是基于节点之间的关系构造的。一个节点可能包含其他节点或者子节点。另一个节点可能与其他节点拥有相同的父节点，我们称之为邻居节点。

- 如果要获取文档中第一个 `<book/>` 元素，那么只需简单通过访问 `firstChild` 属性就可以达到目的：

- `var oRoot = oXmlDom.documentElement;`
-
- `var oFirstBook = oRoot.firstChild;`
- 将 `documentElement` 赋给变量 `oRoot`，可以节省程序空间和输入的内容，尽管这并不是必需的。使用 `firstChild` 属性可以引用根元素 `<books/>` 的第一个子元素 `<books/>` 的引用，并将其赋值给变量 `oFirstBook`。
- 使用 `childNodes` 集合也可以达到相同的目的：

- `var oFirstBook2 = oRoot.childNodes[0];`

• 选择 `childNodes` 集合中的第一项将返回根节点的第一个子节点。因为 `childNodes` 是 JavaScript 中的 `NodeList` 类型，所以使用 `length` 属性可以得到子节点的数量，如下：

- `var iChildren = oRoot.childNodes.length;`
- 本示例中，因为文档元素有四个子节点，所以 `iChildren` 值为 4。
- 正如前面所述，节点可以有子节点，也就意味着它可以有父节点。

通过 `parentNode` 属性可以选择当前节点的父节点：

- `var oParent = oFirstBook.parentNode;`
- 在本小节前面已经提到变量 `oFirstBook`，不过很快，它现在已经是文档中第一个 `<book/>` 元素，所以其 `parentNode` 属性就是指 DOM 的 `documentElement` 属性，也就是 `<books/>` 元素。

• 如果当前节点是 `book` 元素，那么如何选择另一个 `book` 元素呢？因为 `<book/>` 元素有共同的父节点，所以它们互为邻居关系。通过 `nextSibling` 和 `previousSibling` 属性可以选择当前节点的临近节点。`nextSibling` 属性指向下一个邻居，而 `previousSibling` 属性指向前一个邻居：

- `var oSecondBook = oFirstBook.nextSibling;`
-
- `oFirstBook2 = oSecondBook.previousSibling;`

• 这段代码引用第二个 `<book/>` 元素，并将其赋值给 `oSecondBook`。通过 `oSecondBook` 邻居节点对变量 `oFirstBook2` 重新赋值，`oFirstBook2` 的值不变。如果节点没有下一个邻居节点，那么 `nextSibling` 为 `null`。对于 `previousSibling` 也是同样的，如果当前节点没有前一个邻居节点，那么 `previousSibling` 也为 `null`。

• 现在我们知道了如何遍历文档结构，接下来要了解的是如何从树的节点获取数据。例如，使用 `text` 属性可以得到包含第三个 `<book/>` 元素的文本，代码如下：

- `var sText = oRoot.childNodes[2].text;`

• `text` 属性（微软特有的属性）可以得到该节点包含的所有文本节点，该属性相当有用。如果没有 `text` 属性，访问文本节点必须：

- `var sText = oRoot.childNodes[2].firstChild.nodeValue;`

• 这段代码与前面使用 `text` 属性的代码一样得到同样的结果。类似上一个例子，使用 `childNodes` 集合引用第三个 `<book/>` 元素，而使用 `firstChild` 指向 `<book/>` 元素的文本节点，因为文本节点在 **DOM** 中仍是一个节点。使用 `nodeValue` 属性获取当前节点的值，就可以获取文本。

• 这两个示例所产生的结果是相同的，然而使用 `text` 属性和使用文本节点的 `nodeValue` 属性之间存在一个主要的区别。`text` 属性将得到包含当前元素及其子节点的所有文本节点的值，而 `nodeValue` 属性只能得到当前节点的值。它虽然是个有用的属性，但可能会返回比预期值更多的内容。例如，假设我们将 **XML** 文档修改成：

- `<?xml version="1.0" encoding="utf-8"?>`

-

- `<books>`

-

- `<book isbn="0471777781">`

- `<title>Professional Ajax</title>`

- `<author>Nicholas C. Zakas, Jeremy McPeak, Joe Fawcett</author>`

- `</book>`

• `<book isbn="0764579088">Professional JavaScript for Web Developers</book>`

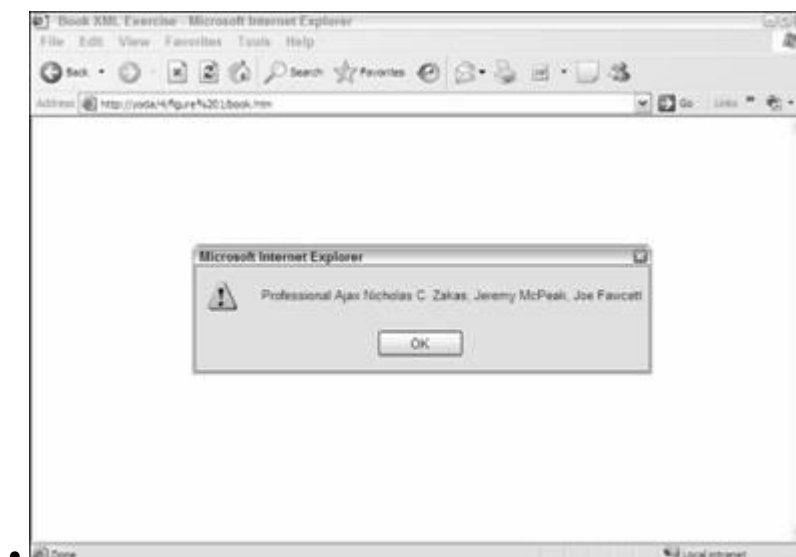
- `<book isbn="0764557599">Professional C#</book>`

• `<book isbn="1861002025">Professional Visual Basic 6 Databases</book>`

- `</books>`

- 新的 XML 文档在第一个<book/>元素中添加了两个新的子节点：
<title/>元素（书名），<author/>元素（作者）。我们再一次使用 text 属性：

- `alert(oFirstChild.text);`
- 代码中没有其他新的内容，我们可以看看图 4-1 中所显示的结果。



• 图 4-1

- 请注意，这时我们将获得<title/>和<author/>元素的文本节点，并将其连接在一起。这就是 text 与 nodeValue 的不同之处。nodeValue 属性只能得到当前节点的值，而 text 属性则将得到包含当前节点及其子节点的所有文本节点。

- MSXML 还提供其他一些获取特定节点或数值的方法，最常用的方法是 `getAttribute()` 和 `getElementsByTagName()`。

- `getAttribute()` 方法将接受一个包含属性名称的字符串型参数，并返回属性值。如果指定的属性不存在，那么返回的值为 `null`。我们还将使用本小节前面提到的那个 XML 文档，请看下列代码：

- `var sAttribute = oFirstChild.getAttribute("isbn");`
-
- `alert(sAttribute);`

- 这段代码获取第一个<book/>元素的 isbn 属性值，并将其赋值给变量 sAttribute，然后使用 Alert() 方法显示该值。

- getElementByTagName() 方法根据其参数所指定的名字，返回子元素的 NodeList。该方法只搜索给定的节点中的元素，所以返回的 NodeList 不包含任何外部元素。例如：

- var cBooks = oRoot.getElementsByTagName("book");

-

- alert(cBooks.length);

- 这段代码获取文档中所有的<book/>元素，并将返回的 NodeList 赋值给变量 cBooks。对于前面那个 XML 文档例子而言，警告框将显示找到的四个<book/>元素。如果要获取所有子节点，那么必须用 “*” 作为 getElementByTagName() 方法的参数，其代码如下所示：

- var cElements = oRoot.getElementsByTagName("*");

- 因为前面的 XML 文档例子中只包含<book/>元素，所以这段代码的结果与上一个示例相同。

- 3. 在 IE 中获取 XML 数据

- 要获取 XML 数据只需使用一个属性，即 xml。该属性将对当前节点的 XML 数据进行序列化。序列化 (serialization) 是将对象转换成简单的可存储或可传输格式的过程。xml 属性将 XML 转换成字符串形式，包括完整的标签名称、属性和文本：

- var sXml = oRoot.xml;

- alert(sXml);

- 这段代码从文档元素开始序列化 XML 数据，并将其作为参数传递给 alert() 方法。下面就是部分已序列化的 XML：

- <books><book isbn="0471777781">Professional Ajax</book></books>

- 已序列化的数据可以载入到另一个 XML DOM 对象，发送到服务器，或者传给另一个页面。通过 xml 属性返回的已序列化 XML 数据，取决于当前节点。如果是在 documentElement 节点使用 xml 属性，那么将返回整个文档的 XML 数据；如果只是在<book/>元素上使用它，那么将返回该<book/>元素所包含的 XML 数据。

- xml 属性是只读属性。如果希望往文档中添加元素，那么必须使用 DOM 方法来实现。

- 4. 在 IE 中操作 DOM

- 现在为止，我们已经学习如何遍历 DOM，从 DOM 中提取信息，将 XML 转换成字符串格式。接下来学习的是如何在 DOM 中添加、删除和替换节点。

- 1 创建节点

- 使用 DOM 方法可以创建多种不同的节点。第一种就是用 createElement()方法创建的元素。向该方法传入一个参数，指明要创建的元素标签名称，并返回一个对 XMLDOMElement 的引用：

- var oNewBook = oXml Dom.createElement("book");
-
- oXml Dom.documentElement.appendChild(oNewBook);

- 这段代码创建一个新的<book/>元素，并通过 appendChild()方法把它添加到 documentElement 中。appendChild()方法添加由其参数指定的新元素，并且将其作为最后一个子节点。但在该例子中，添加到该文档中的是一个空的<book/>元素，因而还需要为该元素添加一些文本：

- var oNewBook = oXml Dom.createElement("book");
-
-
- var oNewBookText =

oXml Dom.createTextNode("Professional .NET 2.0 Generics");

-
- `oNewBook.appendChild(oNewBookText);`
-
- `oXml Dom. documentElement.appendChild(oNewBook);`

这段代码通过 `createTextNode()` 方法创建一个文本节点，并通过 `appendChild()` 方法把它添加到新创建的 `<book/>` 元素中。`createTextNode()` 方法只有一个字符串参数，用来指定文本节点的值。

- 现在已经通过程序创建了新的 `<book/>` 元素，为其提供了一个文本节点，并将它添加到文档中。对于这个新元素而言，还需要像其他邻居节点一样，为其设置 `isbn` 属性。这很简单，只要通过 `setAttribute()` 方法就可以创建属性，该方法适用于所有元素节点。

- `var oNewBook = oXml Dom.createElement("book");`
-
- `var oNewBookText =`
`oXml Dom.createTextNode("Professional .NET 2.0 Generics");`
- `oNewBook.appendChild(oNewBookText);`
-
- `oNewBook.setAttribute("isbn", "0764559885");`
-
- `oXml Dom. documentElement.appendChild(oNewBook);`

- 上面这段代码中，新添加的一行是用来创建 `isbn` 属性的，并将其值赋为 0764559885。`setAttribute()` 方法有两个参数：第一个参数是属性名，第二个参数则是赋给该属性的值。对于向元素添加属性，IE 还提供其他一些方法，不过它们实际上并不比 `setAttribute()` 更好用，而且还需要更多的编码。

- I 删除、替换和插入节点

- 如果能够往文档中添加节点，那么同样意味着可以删除节点。

`removeChild()`方法正是用来实现该功能的。该方法包含一个参数：要删除的节点。例如，要从文档中删除第一个`<book/>`元素，则可以使用以下代码：

- `var oRemovedChild = oRoot.removeChild(oRoot.firstChild);`

• `removeChild()`方法返回被删除的子节点，因而 `oRemovedChild` 变量将指向已删除的`<book/>`元素。当拥有对旧节点的引用时，就可以将其放置在文档的任何地方。

• 如果想用 `oRemovedChild` 指向的元素来替换第三个`<book/>`元素，那么可以通过 `replaceChild()`方法来实现，该方法返回被替换的节点：

- `var oReplacedChild = oRoot.replaceChild(oRemovedChild, oRoot.childNodes[2]);`

• `replaceChild()`方法接受两个参数：新添加的节点和将被替换的节点。在这段代码中，将用 `oRemovedChild` 变量引用的节点替换第三个`<book/>`元素，而被替换节点的引用将存在 `oReplacedChild` 变量中。

• 由于 `oReplacedChild` 变量是被替换节点的引用，因而可以容易地将其插入到文档中。使用 `appendChild()`方法可以将其添加到子节点列表的最后，也可以使用 `insertBefore()`方法将该节点插入到某个节点之前：

- `oRoot.insertBefore(oReplacedChild, oRoot.lastChild);`

• 这段代码将之前被替换的节点插入到最后一个`<book/>`元素的前面。`lastChild`属性的用法与 `firstChild`选择第一个子节点非常相似，通过该属性可以获取最后一个子节点。`insertBefore()`方法接受两个参数：要插入的节点和表示插入点的节点（插入点在该节点之前）。该方法也将返回插入节点的值，但上述例子中并不需要。

• 如你所见，**DOM** 是一个相当强大的接口，通过它可以实现数据的获取、删除和添加等操作。

- 5. 在 IE 中处理错误

- 在 XML 数据的载入过程中,可能会由于不同的原因而抛出错误。例如,外部的 XML 文件找不到,或者 XML 的格式不正确。为了处理这些情况,MSXML 提供了一个包含错误信息的 parseError 对象。对于每个由 MSXML 创建的 XML DOM 文档对象而言,该对象都是其所属的属性值之一。

- 我们可以通过 parseError 对象公开的与整数 0 进行比较的 errorCode 属性来检查错误。如果 errorCode 不等于 0,则表示有错误发生。下面的例子故意设计出现一个错误。

- ```
var sXml = "<root><person><name>Jeremy
McPeak</name></root>";
```

- ```
var oXml Dom = createDocument();
```
- ```
oXml Dom. loadXML(sXml);
```
- ```
if (oXml Dom. parseError. errorCode != 0) {
```
- ```
 alert("An Error Occurred: " +
```
- ```
oXml Dom. parseError. reason);
```
- ```
} else {
```
- ```
    //当 XML 载入成功后的操作
```
- ```
}
```

- 大家会注意到,在突出显示的代码行中,<person>元素是不完整的(没有相应的</person>标签)。由于要载入的 XML 的格式不正确,因此将产生一个错误。然后 errorCode 与 0 进行比较,如果不相等(在本例中就不相等),那么将显示发生错误的警告。要实现该功能,可以使用 parseError 对象的 reason 属性来获取错误出现的原因。

- parseError 对象提供了以下属性,能够帮助你更好地了解错误:
- **q** errorCode: 错误代码(长整型);
- **q** filePos: 在文件中发生错误的位置(长整型);
- **q** line: 包含错误的代码行的行号(长整型);

- `q linePos`: 在特定行中发生错误的位置（长整型）；
- `q reason`: 错误的原因（字符串型）；
- `q srcText`: 发生错误的代码行内容（字符串型）；
- `q url`: XML 文档的 URL（字符串型）。

• 尽管这些属性提供了每种错误的信息，但是应该使用哪个属性，则取决于你的需要。

• **`errorCode`** 属性可以是正数也可以是负数，只有当 **`errorCode`** 为 0 时才表示没有错误发生。

#### • 4.1.2 Firefox 中的 XML DOM

• 现在让我们看看 Firefox 中的 XML DOM 实现，Firefox 的开发人员采用更为标准的方法，将其作为 JavaScript 实现的一部分。Mozilla 确保所有基于 Gecko 的浏览器的所有平台都支持 XML DOM。

• Firefox 中创建一个 XML DOM，需要调用 `document.implementation` 对象的 `createDocument()` 方法。该方法接受三个参数：第一个参数是包含文档所使用的命名空间 URI 的字符串；第二个参数是包含文档根元素名称的字符串；第三个参数是要创建的文档类型（也称为 `doctype`）。如果要创建空的 DOM 文档，则代码如下所示：

```
var oXmlDom = document.implementation.createDocument("", null);
```

• 前两个参数是空字符串，第三个参数为 `null`，这样可以确保生成一个彻底的空文档。事实上，现在 Firefox 中并不提供针对文档类型的 JavaScript 支持，所以第三个参数总是为 `null`。如果要创建包含文档元素的 XML DOM，那么可以在第二个参数中指定标签名称：

```
var oXmlDom = document.implementation.createDocument("", "books", null);
```

- 这段代码创建了一个 XML DOM，其 documentElement 是<books/>。

如果要创建包含指定命名空间的 DOM，可以在第一个参数中指定命名空间 URI：

- `var oXmlDom =`

`document.implementation.createDocument("http://www.site1.com",`

- `"books", null);`

• 当在 `createDocument()` 方法中指定命名空间时，Firefox 会自动附上前缀 `a0` 以表示命名空间 URI：

- `<a0:books xmlns:a0="http://www.site1.com" />`

• 接着，你可以通过程序来填充 XML 文档，不过在一般情况下，还需要在空的 XML DOM 对象中载入现有的 XML 文档。

- 1. 在 Firefox 中载入 XML 数据

• 在 Firefox 中，将 XML 载入 XML DOM 的方法和微软采用的方法大致相同，只存在一个显著区别：Firefox 只支持 `load()` 方法。因此，在这两种浏览器中载入外部 XML 数据的代码是相同的：

- `oXmlDom.load("books.xml");`

• 与微软的 IE 一样，Firefox 同样实现了 `async` 属性，该属性的行为也与其一致：将 `async` 设置为 `false`，表示以同步模式载入文档；否则，以异步模式载入文档。

• Firefox 的 XML DOM 实现和微软的 XML DOM 实现还存在另一个不同，即 Firefox 不支持 `readyState` 属性及 `onreadystatechange` 事件处理函数。在 Firefox 中，支持 `load` 事件和 `onload` 事件处理函数。在文档完全载入后将触发 `load` 事件：

- `oXmlDom.load("books.xml");`

- `oXmlDom.onload = function () {`

- `//文档完全载入后的操作`

- `};`

- 正如前面所说,在 Firefox 的 XML DOM 实现中,并没有 loadXML() 方法,不过通过 Firefox 中的 DOMParser 类可以模拟 loadXML() 的行为。该类有一个名为 parseFromString() 的方法,用来载入字符串并解析成文档:

- ```
var sXml = "<root><person><name>Jeremy  
McPeak</name></person></root>";
```

- ```
var oParser = new DOMParser();
```

- ```
var oXml Dom = oParser.parseFromString(sXml,"text/xml");
```

- 在这段代码中,创建了一个 XML 字符串,并作为参数传递给 DOMParser 的 parseFromString() 方法。parseFromString() 方法的两个参数分别是 XML 字符串和数据的内容类型(一般设置为 text/xml)。parseFromString() 方法返回 XML DOM 对象,因此这里得到的 oXml Dom 与第一个例子相同。

- 2. 在 Firefox 中获取 XML 数据

- 尽管存在这样那样的不同,但 IE 和 Firefox 中用于获取文档中 XML 数据的大多数属性和方法是一致的。正如在 IE 中,可以使用 documentElement 属性来获取文档的根元素,例如:

- ```
var oRoot = oXml Dom.documentElement;
```

- Firefox 同样支持 W3C 标准属性,包括 childNodes、firstChild、lastChild、nextSibling、nodeName、nodeType、nodeValue、ownerDocument、parentNode 和 previousSibling。不幸的是,对于微软专有的 text 和 xml 属性,Firefox 并不支持,不过可以利用其他方法来模拟该属性的行为。

- 大家应该还记得, text 属性返回了当前节点的内容,或者是当前节点及其子节点的内容。这不仅仅返回当前节点的文本,还有所有子节点的文本,因此要模拟该功能实现是十分容易的。下面这个简单的函数就能够完成该功能,该函数唯一的参数是一个节点:

- ```
function getText(oNode) {
```

- ```
 var sText = "";
```

- ```
    for (var i = 0; i < oNode.childNodes.length; i++) {
```

- if (oNode.childNodes[i].hasChildNodes()) {
- sText += getText(oNode.childNodes[i]);
- } else {
- sText += oNode.childNodes[i].nodeValue;
- }
- }
- return sText;
- }

- 在 `getText()` 函数中，`sText` 变量用来保存获取的所有文本。接着对 `oNode` 的子节点使用 `for` 循环进行遍历，检查每个子节点是否包含子节点。如果有子节点，那么就将其 `childNodes` 传给 `getText()` 函数，并进行同样的处理；如果没有子节点，那么将当前节点的 `nodeValue` 加到字符串中（对文本节点而言，这只是文本字符串）。处理了所有子节点后，该函数返回变量 `sText`。

- IE 中的 `xml` 属性将存放对当前节点包含的所有 XML 进行序列化的结果。在 Firefox 中，提供了一个名为 `XMLSerializer` 对象来完成这一功能。该对象提供一个使用 JavaScript 可访问的 `serializeToString()` 方法，使用该方法可以对 XML 数据进行序列化。

- function serializeXml (oNode) {
- var oSerializer = new XMLSerializer();
- return oSerializer.serializeToString(oNode);
- }

- `serializeXml()` 函数以 XML 节点作为参数，创建一个 `XMLSerializer` 对象，并将该节点传给 `serializeToString()` 方法。该方法将向调用者返回 XML 数据的字符串表示。

- 对于节点操作的 DOM 方法，Firefox 与 IE 大致相同。参见“在 IE 中操作 DOM”小节。

- 3. 在 Firefox 中处理错误

- Firefox 与 IE 的错误处理并不一样。当 IE 遇到错误时，它会填充 `parseError` 对象；而当 Firefox 遇到错误时，它会将包含错误信息的 XML 文档载入到 XML DOM 文档中。看下面的这个例子：

- ```
var sXml = "<root><person><name>Jeremy
McPeak</name></root>";
```
- ```
var oParser = new DOMParser();
```
- ```
var oXmlDom = oParser.parseFromString(sXml, "text/xml");
```
- ```
if (oXmlDom.documentElement.tagName != "parsererror") {
```
- ```
 //没有错误发生，进行所需操作
```
- ```
} else {
```
- ```
 alert("An Error Occurred");
```
- ```
}
```

- 在突出显示的代码行中，你会发现其中将产生一个错误：XML 字符串格式不正确（因为 `<person>` 元素不完整，没有相应的 `</person>` 元素）。当载入错误的 XML 时，XML DOM 对象将会载入一个 `documentElement` 为 `<parsererror/>` 的错误文档。我们可以通过检查 `documentElement` 的 `tagName` 属性来很容易地确定是否发生错误。如果 `tagName` 属性不是 `parsererror`，就可以确定没有发生任何错误。

- 在本例中，可能会生成如下所示的错误文档：

- ```
<parsererror
xmlns="http://www.mozilla.org/newlayout/xml/parsererror.xml">XML
```

- Parsing Error: mismatched tag. Expected: `</person>`.
- Location: <http://yoda/fooreader/test.htm>
- Line Number 1, Column

43: `<sourcetext><root><person><name>Jeremy`

- `McPeak</name></root>`

- -----^</sourcetext>  
></parsererror>

- 所有的错误信息都包含在错误文档的文本中。如果要通过程序使用这些错误信息，那么首先就要对其进行解析。最简单的方法是使用一个稍长的正则表达式：

- `var reError = />([\s\S]*?)Location: ([\s\S]*?)Line Number (\d+), Column`

- `(\d+):<sourcetext>([\s\S]*?)(?:\-*\^)/;`

- 该正则表达式将错误文档分为五个部分：错误消息、发生错误的文件名、行号、该行中发生错误的位置，以及发生错误的源代码。使用正则表达式对象的 `test()` 方法可以使用这些信息：

- `if (oXml Dom. firstChild. tagName != "parsererror") {`

- `//没有错误发生，进行所需操作`

- `} else {`

- `var oXml Serializer = new XmlSerializer();`

- `var sXml Error =`

- `oXml Serializer. serializeToString(oXml Dom);`

- `var reError = />([\s\S]*?)Location: ([\s\S]*?)Line Number (\d+), Column`

- `(\d+):<sourcetext>([\s\S]*?)(?:\-*\^)/;`

- `reError. test(sXml Error);`

- 正则表达式捕获到的第一部分数据是错误消息，第二部分是文件名，第三部分是行号，第四部分是行内位置，第五部分是源码。你可以使用这些解析后的信息来创建自定义的错误消息：

- `var str = "An error occurred!!\n" +`

- `"Description: " + RegExp. $1 + "\n" +`

- `"File: " + RegExp. $2 + "\n" +`

- `"Line: " + RegExp. $3 + "\n" +`

- `"Line Position: " + RegExp.$4 + "\n" +`
- `"Source Code: " + RegExp.$5;`
- `alert(str);`
- 如果发生错误, 那么 `alert()` 方法会以易于阅读的格式在警告框中来显示相关的错误信息。

#### 4.1.3 跨浏览器兼容的 XML

• 在一个 Ajax 应用程序及大多数 JavaScript 代码中, 必须考虑浏览器之间的差异问题。当在 IE 和 Firefox 浏览器中使用基于 XML 的解决方案时, 有两种选择: 基于所用的浏览器编写正确的代码, 以创建自己的函数; 或者使用现成的库。在大多数情况下, 使用现成的库是最简单的方法, 例如第 2 章中介绍的 zXml 库。zXml 不仅支持 XMLHttpRequest, 还提供了 XML 操作的通用接口。

- 例如, 如果要创建 XML DOM 文档, 就可以使用

`zXml Dom. createDocument()`:

- `var oXml Dom = zXml Dom. createDocument();`
- 对于不同的浏览器, 这行代码都能够创建 DOM 文档, 无需为不同的浏览器提供不同的版本。此外, zXml 还为标准的 Firefox DOM 文档添加了一些 IE 中实现的功能。

• 其中最重要的一个改进是提供对 `readyState` 属性和 `onreadystatechange` 事件处理函数的支持。原先对于 Firefox, 需要使用特定的 `onload` 事件处理函数, 但现在则可以使用一个无需检测浏览器的代码, 例如:

- `oXml Dom. onreadystatechange = function () {`
- `if (oXml Dom. readyState == 4) {`
- `//当文档成功载入后, 完成某些操作`
- `}`
- `};`

- zXml 库还为 Firefox 中的所有节点添加了原先没有的 xml 和 text 属性支持。原先需要使用 XMLSerializer 或者单独的函数来获取这些值，现在的做法和 IE 完全一样：

- var oRoot = oXml Dom.documentElement;
- 
- var sFirstChildText = oRoot.firstChild.text;
- 
- var sXml = oRoot.xml;
- zXml 还为 Firefox DOM 文档提供了 loadXML() 方法，避免使用

DOMParser 对象。

- var oXml Dom2 = zXml Dom.createDocument();
- 
- oXml Dom2.loadXML(sXml);
- 最后，zXml 库还为 Firefox 的实现添加 parseError 对象。该对象与 IE 中相应的 parseError 对象非常类似。主要的区别在于 errorCode 属性，当发生错误时该属性设置为非零值。因此不要使用该属性来查找特定的错误，只可以用来判断是否出错。其他属性和 IE 基本一致：

- if (oXml Dom.parseError.errorCode != 0) {
- var str = "An error occurred!!\n" +
- "Description: " + oXml Dom.parseError.reason + "\n"
- +
- "File: " + oXml Dom.parseError.url + "\n" +
- "Line: " + oXml Dom.parseError.line + "\n" +
- "Line Position: " + oXml Dom.parseError.linePos +
- "\n" +
- "Source Code: " + oXml Dom.parseError.srcText;
- alert(str);
- } else {

- `//成功载入的操作`
- `}`

当然你也不是一定要使用跨浏览器兼容的 XML 库，但这样的库确实相当有用。下面的小节就将使用 zXml 库来完成一个例子的开发。

#### • 4.1.4 基本的 XML 实例

XML 是一种语义描述语言。一般来说，包含在给定 XML 文档中的元素描述该文档的数据，因此对于静态信息或是不常改变的信息，这是一种不错的数据存储方法。

假设有一个在线书店，它有一个保存在 XML 文档 `books.xml` 中的“最佳选择”列表。你需要将这些信息显示给用户，但是又不希望使用服务器组件来实现，因此只能采用基于 JavaScript 的解决方案。使用 zXml 库就可以编写这样的 JavaScript 解决方案，载入、解析 XML 文件，并使用 DOM 方法在 Web 页面中显示这些信息。

- `books.xml` 文件中包含下列 XML 数据：

```

• <?xml version="1.0" encoding="utf-8"?>
•
• <bookList>
• <book isbn="0471777781">
• <title>Professional Ajax</title>
• <author>Nicholas C. Zakas, Jeremy McPeak, Joe
Fawcett</author>
• <publisher>Wrox</publisher>
• </book>
• <book isbn="0764579088">
• <title>Professional JavaScript for Web
Developers</title>
• <author>Nicholas C. Zakas</author>
• <publisher>Wrox</publisher>

```

- `</book>`
- `<book isbn="0764557599">`
- `<title>Professional C#</title>`
- `<author>Simon Robinson, et al</author>`
- `<publisher>Wrox</publisher>`
- `</book>`
- `<book isbn="1861006314">`
- `<title>GDI+ Programming: Creating Custom Controls`

Using C#</title>

- `<author>Eric White</author>`
- `<publisher>Wrox</publisher>`
- `</book>`
- `<book isbn="1861002025">`
- `<title>Professional Visual Basic 6`

Databases</title>

- `<author>Charles Williams</author>`
- `<publisher>Wrox</publisher>`
- `</book>`
- `</bookList>`

正如你所见，文档元素<bookList/>中包含一系列的<book/>元素，每个<book/>元素包含一本特定书籍的信息。

# 1. 载入 XML 数据

实现的第一步是创建 XML DOM 文档，并载入 XML 数据。因为 books.xml 是以异步模式载入的，因此需要设置 onreadystatechange 事件处理函数：

- `var oXml Dom = zXml Dom.createDocument();`
- `oXml Dom.onreadystatechange = function () {`
- `if (oXml Dom.readyState == 4) {`
-

- }
- };

• 当 readystatechange 事件触发并调用相应的事件处理函数时，将对 readyState 属性进行检查，如果该属性的值为 4，就表明文档已完全载入，可以使用 DOM 了。

• 接下来一步是检查错误，因为即使文档已经完全载入，但并不表示一切正常：

- var oXml Dom = zXml Dom.createDocument();
- oXml Dom.onreadystatechange = function () {
- if (oXml Dom.readyState == 4) {
- if (oXml Dom.parseError.errorCode == 0) {
- parseBookInfo(oXml Dom);
- } else {
- var str = "An error occurred!!\n" +
- "Description: " +
- oXml Dom.parseError.reason + "\n" +
- "File: " + oXml Dom.parseError.url + "\n" +
- "Line: " + oXml Dom.parseError.line + "\n"
- +
- "Line Position: " +
- oXml Dom.parseError.linePos + "\n" +
- "Source Code: " +
- oXml Dom.parseError.srcText;
- alert(str);
- }
- }
- };

- 如果没有错误发生，那么 XML DOM 文档将作为参数传给 `parseBookInfo()` 函数，该函数负责解析这个书籍列表。如果发生了错误，那么将通过 `alert()` 方法在警告框中来显示 `parseError` 对象中的错误信息。

- 当完成了 `onreadystatechange` 事件处理函数的编写之后，就将使用 `load()` 方法载入 XML 数据：

- `oXml Dom. load("books.xml");`
- 现在 XML 文档已经载入。接下来则是对 XML 数据进行解析。

- 2. 解析书籍列表

- `parseBookInfo()` 函数负责对 DOM 文档进行解析。该函数接受一个参数，即 DOM 文档自身：

- `function parseBookInfo(oXml Dom) {`
- `var oRoot = oXml Dom. documentElement;`
- `var oFragment = document.createDocumentFragment();`

- `oRoot` 变量将赋值为 XML 文档的 `documentElement`。这样做仅仅是为了方便，毕竟输入 `oRoot` 要比输入 `oXml Dom. documentElement` 简单、迅速得多。你还需要创建一个文档片断。`parseBookInfo()` 函数将生成许多 HTML 元素，并将其转换成 HTML DOM 载入浏览器。往 HTML DOM 中添加每个元素是一个大开销的过程，因为显示这些变化会花费不少时间。所以我们采用的方法是，先将每个元素添加到文档片断，当所有 HTML 元素创建后，再将文档片断添加到文档中。这样做只需要对 HTML DOM 进行一次更新，从而可以更快地呈现。

- 你知道，文档元素的子节点只有 `<book/>` 元素，因此可以通过 `childNodes` 集合来遍历该文档：

- `var aBooks = oRoot.getElementsByTagName("book");`
- 
- `for (var i = 0; i < aBooks.length; i++) {`
- `var slsbn = aBooks[i].getAttribute("isbn");`



- `var sAuthor, sTitle, sPublisher;`
- 在 for 循环中,实际的解析工作才开始。首先,通过 `getAttribute()` 方法获取<book/>元素的 `isbn` 属性,并保存在变量 `sIsbn` 中。该值用来向用户显示书籍封面及其实际的 **ISBN** 值。这里还声明了变量 `sAuthor`、`sTitle` 和 `sPublisher`,它们分别用来保存<author/>、<title/>和<publisher/>元素的值。
- 接下来必须获取书籍的相关数据,这可以通过多种不同的方法来实现。例如可以使用 `childNodes` 集合并遍历子节点,但这个例子中采用了另一种不同的方法。它通过一个使用 `firstChild` 和 `nextSibling` 属性的 `do...while` 循环来达到相同的目的:

```

• var oCurrentChild = aBooks[i].firstChild;

•

• do {
• switch (oCurrentChild.tagName) {
• case "title":
• sTitle = oCurrentChild.text;
• break;
• case "author":
• sAuthor = oCurrentChild.text;
• break;
• case "publisher":
• sPublisher = oCurrentChild.text;
• break;
• default:
• break;
• }
• oCurrentChild = oCurrentChild.nextSibling;
• } while (oCurrentChild = oCurrentChild.nextSibling);

```

- 第一行代码将 `oCurrentChild` 变量的值设置为当前 `<book/>` 元素的第一个子节点（记住，这将发生在 `for` 循环中）。子节点的 `tagName` 属性用在 `switch` 语句中来判断如何进行数据的处理。`switch` 语句将根据当前节点的相应 `tagName` 来将其值赋给相应的变量。为了获取这个数据，可以使用节点的 `text` 属性，该属性可以获取元素内的所有文本节点。使用 `nextSibling` 属性，将当前节点的下一个节点赋给变量 `oCurrentChild`。如果下一个邻居存在，那么继续循环，否则 `oCurrentChild` 设为 `null` 并退出循环。

- 当所有数据变量包含所需数据时，就可以开始生成 **HTML** 元素来显示数据。通过程序创建的 **HTML** 元素结构可能如下所示：

- `<div class="bookContainer">`
- ``
- `<div class="bookContent">`
- `<h3>Professional Ajax</h3>`
- `Written by: Nicholas C. Zakas, Jeremy McPeak, Joe Fawcett<br />`
- `ISBN #0471777781`
- `<div class="bookPublisher">Published by Wrox</div>`
- `</div>`
- `</div>`

- 为了提高列表的可读性，可以通过 `<div/>` 元素来实现交替背景色的效果。奇数行的书籍信息的背景色为浅灰，由类名为 `bookContainer-odd`（**CSS**）类定义；偶数行的书籍信息的背景色为白色，由 `bookContainer`（**CSS**）类定义。

- 通过 **DOM** 方法来生成这些代码，虽然很简单但也十分冗长。第一步是通过 `createElement()` 的 **DOM** 方法来创建容器 `<div/>`、`<img/>` 和内容 `<div/>` 元素。

- `var divContainer = document.createElement("div");`
- `var imgBookCover = document.createElement("img");`
- `var divContent = document.createElement("div");`
- 
- `var sOdd = (i % 2) ? "" : "-odd";`
- `divContainer.className = "bookContainer" + sOdd;`

在创建元素的同时，将为其指定相应的 CSS 类。通过使用取余操作符 (%) 判断当前书籍是位于奇数行还是偶数行。根据不同情况对 `sOdd` 变量赋予相应的值，偶数是空字符串，奇数则是“-odd”，并用在 `className` 的赋值上。

接下来对书籍封面图像的属性进行赋值。这些 PNG 图像使用 ISBN 号作为文件名：

- `imgBookCover.src = "images/" + sIsbn + ".png";`
- `imgBookCover.className = "bookCover";`
- `divContainer.appendChild(imgBookCover);`

在这里，将对该图像的 `src` 和 `className` 属性进行赋值，并将其添加到 `divContainer` 中。完成图像处理后，就可以添加内容。首先要添加的信息是书籍标题，这是一个 3 级标题元素 `<h3/>`。该元素也通过 `createElement()` 方法创建。

- `var h3Title = document.createElement("h3");`
- `h3Title.appendChild(document.createTextNode(sTitle));`
- `divContent.appendChild(h3Title);`

如果要创建包含标题的文本节点，那么要使用 `createTextNode()` 方法，并将其添加到 `<h3/>` 元素中，然后将完整标题添加到 `divContent` 中。

接下来添加作者和 ISBN 的信息。这两部分信息都是文本节点，且除了 `divContent` 不存在父元素。不过在这两个文本节点间还将加入一个换行元素。

- `divContent.appendChild(document.createTextNode("Written by: " + sAuthor));`
- `divContent.appendChild(document.createElement("br"));`
- `divContent.appendChild(document.createTextNode("ISBN: #" + slsbn));`

这段代码将创建这些信息。首先，将包含作者信息的文本节点添加到 `divContent` 中，接着创建并添加换行元素 `<br/>`，最后添加包含 ISBN 信息的文本节点。

- 最后添加的是出版社信息：
- `var divPublisher = document.createElement("div");`
- `divPublisher.className = "bookPublisher";`
- `divPublisher.appendChild(document.createTextNode("Published by: " + sPublisher));`
- `divContent.appendChild(divPublisher);`

出版社信息在 `<div/>` 元素中显示。`<div/>` 元素创建后，其 `className` 设为 “bookPublisher”，包含出版社名称的文本节点添加到该元素中。这样就完成了 `divPublisher` 元素，可以将其添加到 `divContent` 中。

到此为止，所有的数据操作就全部完成了。但是，`divContent` 还没有指定 CSS 类名，而且必须添加到 `divContainer` 中，而 `divContainer` 必须依次地添加到文档片段中。下面三行代码将完成这个功能：

- `divContent.className = "bookContent";`
- `divContainer.appendChild(divContent);`
- `oFragment.appendChild(divContainer);`
- 最后一步是，当通过下述代码遍历了所有的书籍节点之后，则将文档片段添加到页面主体中。
- `document.body.appendChild(oFragment);`

- 这段代码并没有真正地添加文档片段本身，实际上只是添加文档片段的所有子节点，并立刻将其转换成 **HTML DOM**。有了这最后一行代码，`parseBookInfo()` 函数也就完成了。

- **3. 整合**

- 该 **Web** 页面的主体部分完全是由 **JavaScript** 生成的。正因如此，所以必须在文档载入后才能执行元素的创建和代码的插入。请记住，在载入 `books.xml` 后才调用 `parseBookInfo()`，所以创建 **XML DOM** 对象的代码必须在页面载入时执行。创建一个名为 `init()` 的函数，用来放置创建 **XML DOM** 对象的代码。

- ```
function init() {  
    var oXml Dom = zXml Dom.createDocument();  
    oXml Dom.onreadystatechange = function () {  
        if (oXml Dom.readyState == 4) {  
            if (oXml Dom.parseError.errorCode == 0) {  
                parseBookInfo(oXml Dom);  
            } else {  
                alert("An Error Occurred: " +  
oXml Dom.parseError.reason);  
            }  
        }  
    };  
    oXml Dom.load("book.xml");  
}
```

- `init()` 函数用来处理 `window.onload` 事件。这有助于确保 **JavaScript** 生成的元素添加到页面中，而不引起任何错误。

- 这个小应用程序就在一个 **HTML** 文档中。所需的只是两个 `<script/>` 元素，一个为 **CSS** 提供的 `<link/>` 元素，以及为 `onload` 事件处理函数赋值。

- `<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"`
 - `"http://www.w3.org/TR/xhtml11/DTD`
- `D/xhtml11.dtd">`
- - `<html xmlns="http://www.w3.org/1999/xhtml" >`
 - `<head>`
 - `<title>Book XML Exercise</title>`
 - `<link rel="stylesheet" type="text/css"`
- `href="books.css" />`
- `<script type="text/javascript"`
- `src="zxml.js"></script>`
- `<script type="text/javascript"`
- `src="books.js"></script>`
- `</head>`
 - `<body onload="init()"`
 -
 - `</body>`
 - `</html>`
- 当运行这段代码时，将看到如图 4-2 所示的结果。



- 4.2 浏览器对 XPath 的支持

- 随着 XML 的愈发流行,直接访问特定数据的需求也越来越紧迫。1999 年 7 月, XPath (XML Path 语言) 首次在 XSL (可扩展样式表语言) 规约中作为一种在 XML 文档中查找任意节点的解决方案被提出。XPath 采用的是非 XML 语法, 它非常类似于文件系统路径的语法。该语言由位置路径、表达式, 以及一些有助于获取特定数据的函数等所组成。

- 4.2.1 XPath 概述

- XPath 表达式由两部分所组成: 上下文节点和选择模式。上下文节点是从选择模式开始的上下文。根据前一节的 books.xml, 考虑下列 XPath 表达式:

- book/author

- 如果在根节点 (上下文) 处执行该表达式, 那么会返回所有 <author/> 节点。因为 <book/> 元素是文档元素的子节点, 并包含一个 <author/> 元素。该表达式没有特别指定, 所以返回所有 <author/> 元素。

- 如果只想得到特定 ISBN 的 <book/> 元素, 该怎么样呢? XPath 表达式应该是这样:

- book[@i sbn='0471777781']

- 表达式中 book 部分描述获取的元素。方括号里的是元素匹配的条件。@i sbn 部分表示 i sbn 属性 (@ 是属性的简写)。所以, 该表达式的含意是 “查找 i sbn 属性为 '0471777781' 的作者元素”。

- XPath 表达式也可以非常复杂。请看下面这个表达式:

- book[author[contains(text(),'McPeak')]]

- 该表达式的含意是, “查找其作者元素的文本中包含字符串 'McPeak' 的作者元素”。因为这是一个更加复杂的表达式, 将其从外至内地分解有助于阅读。移去所有条件, 可以得到表达式:

- book[...]

- 首先，因为这是最外层的元素，因此可以得知该表达式将返回 <book/> 元素；接下来是查询条件。在方括号中第一组信息是 <author/> 元素：

- `author[...]`

- 现在可以得知查找的是包含 <author/> 子元素的作者元素。不过，还需要继续对 <author/> 元素的子节点进行检查，因为表达式还没有结束：

- `contains(text(), 'McPeak')`

- `contains()` 函数包含两个参数，且如果第一个字符串参数包含第二个字符串参数时返回 `true`。`text()` 函数根据当前上下文返回所有文本，所以 <author/> 元素的文本内容作为第一个参数传入 `contains()` 函数中。第二个传入 `contains()` 的参数是查找的文本，本示例中是 'McPeak'。

- 请注意，**`contains()`** 函数与所有其他 XPath 函数一样，都是区分大小写的。

- 结果的节点集合是一个 <book/> 元素，因为只有一个书籍元素的作者名字为 **McPeak**。

- 如你所见，XPath 是一门有用的语言，它使得在 XML 数据中查找特定节点的工作变得简单许多。因此毫不奇怪，微软和 Mozilla 都在各自的浏览器中实现了供客户端使用的 XPath。

- 4.2.2 IE 中的 XPath

- 微软在 MSXML 3.0 及以后的版本中实现 XPath。如果使用 Windows XP 的操作系统，或者安装 IE 6.0 以上的版本，那么就可以使用 XPath。如果不是，那么你需要下载并安装最新的 MSXML 包。

- 微软通过两种方法来实现基于 XPath 表达式的节点选择。第一种是 `selectSingleNode()` 方法，它将返回匹配表达式的第一个节点。例如：

- ```
var oFirstAuthor =
oXmlDom.documentElement.selectSingleNode("book/author");
```



- 这段代码返回是 documentElement 的上下文中<book/>元素子节点的一个<author/>元素。对于本例而言，其结果是下面这个节点：

- <author>Nicholas C. Zakas, Jeremy McPeak, Joe Fawcett</author>

- 在微软 XPath 实现中，第二种基于 XPath 表达式的节点选择方法是 selectNode()。该方法将返回一个 NodeList，它是所有与该 XPath 表达式匹配的节点集合：

- var cAuthors =  
oXmlDom.documentElement.selectNodes("book/author");

- 你或许会猜到，这段代码将返回 documentElement 的上下文中<book/>元素的所有<author/>元素。如果文档中没有相匹配的元素，那么还是会返回一个 NodeList，只是长度为 0。因此在使用 NodeList 前，最好先判断其长度是否大于 0：

- var cAuthors =  
oXmlDom.documentElement.selectNodes("book/author");

- • if (cAuthors.length > 0) {  
•     //进行操作  
• }

- 4.2.3 使用命名空间

- XML 中的 X 表示可扩展（eXtensible）。在一个 XML 文档中没有预先定义的元素。开发人员可以在任意给定的 XML 文档中创建各种元素。XML 的可扩展性是 XML 之所以这么流行的原因之一，不过其本身存在一个问题：命名冲突。例如，请分析下列 XML 文档：

- <?xml version="1.0" encoding="utf-8"?>  
•

- <addresses>
- <address>
- <number>12345</number>
- <street>Your Street</street>
- <city>Your City</city>
- <state>Your State</state>
- <country>USA</country>
- </address>
- </addresses>

该文档中并没有什么特别之处，只是描述一个美国地址。但是如果增加了下面这些行：

- <?xml version="1.0" encoding="utf-8"?>
- 
- <addresses>
- <address>
- <number>12345</number>
- <street>Your Street</street>
- <city>Your City</city>
- <state>Your State</state>
- <country>USA</country>
- </address>
- 
- <address>
- <ip>127.0.0.1</ip>
- <hostname>local host</hostname>
- </address>
- </addresses>

现在，这个文档就描述了两类地址：邮寄地址和计算机网络地址。这些地址都是合法地址，以不同方法进行处理，因为这些<address/>元素包

含完全不同的子元素。但是 XML 处理器却无法区分这两类<address/>元素之间的不同之处，所以需要你自己完成。对于这种情况，命名空间就是解决方案。

- 命名空间由两部分组成：命名空间 URI 和前缀。命名空间 URI 定义命名空间。一般来说，命名空间 URI 是一个网站 URL，因为不同网站地址必须唯一的。前缀则是 XML 文档中命名空间的局部名称。命名空间中的每个标签名称都使用命名空间前缀。命名空间声明的语法格式如下所示：

- `xmlns:namespace-prefix="namespaceURI"`

- `xmlns` 关键字将告诉 XML 解析器，这是一个命名空间声明。

`namespace-prefix` 则是该命名空间下所有元素使用的局部名称，而 `namespaceURI` 是前缀所表示的统一资源定位符。

- 命名空间必须在 XML 文档中使用命名空间之前进行声明。在这个例子中，在根元素中包含了对命名空间的声明。

- `<?xml version="1.0" encoding="utf-8"?>`

- 

- `<addresses xmlns:mail="http://www.wrox.com/mail"`

- `xmlns:comp="http://www.wrox.com/computer">`

- 

- `<mail:address>`

- `<mail:number>12345</mail:number>`

- `<mail:street>Your Street</mail:street>`

- `<mail:city>Your City</mail:city>`

- `<mail:state>Your State</mail:state>`

- `<mail:country>USA</mail:country>`

- `</mail:address>`

- `<comp:address>`

- `<comp:ip>127.0.0.1</comp:ip>`

- `<comp:hostname>localhost</comp:hostname>`
- `</comp:address>`
- `</addresses>`

最新的 XML 文档定义了两个命名空间：前缀为 mail 的命名空间表示邮寄地址；前缀为 comp 的命名空间表示计算机网络地址。可能你已经注意到前缀的用法。每个与地址类型相关的元素都与对应的命名空间关联，所以每个与邮寄地址相关的元素包含前缀 mail，而每个与计算机网络地址相关的元素包含前缀 comp。

- 使用命名空间就可以避免命名冲突，现在 XML 处理器就能够区分这两种地址类型之间的区别。

- 当使用 selectSingleNode() 和 selectNodes() 方法时，命名空间会使 XPath 增加一点点复杂度。请看看下面这个修改过的 books.xml 文件：

- `<?xml version="1.0" encoding="utf-8"?>`
- `<bookList xmlns="http://site1.com"`  
`xmlns:pub="http://site2.com">`
- 
- `<book isbn="0471777781">`
- `<title>Professional Ajax</title>`
- `<author>Nicholas C. Zakas, Jeremy McPeak, Joe`  
`Fawcett</author>`
- 
- `<pub:name>Wrox</pub:name>`
- 
- `</book>`
- `<book isbn="0764579088">`
- `<title>Professional JavaScript for Web`  
`Developers</title>`
- `<author>Nicholas C. Zakas</author>`
-

- `<pub: name>Wrox</pub: name>`
- 
- `</book>`
- `<book isbn="0764557599">`
- `<title>Professional C#</title>`
- `<author>Simon Robinson, et al</author>`
- 
- `<pub: name>Wrox</pub: name>`
- 
- `</book>`
- `<book isbn="1861006314">`
- `<title>GDI+ Programming: Creating Custom Controls`

Using C#</title>

- `<author>Eric White</author>`
- 
- `<pub: name>Wrox</pub: name>`
- 
- `</book>`
- `<book isbn="1861002025">`
- `<title>Professional Visual Basic 6`

Databases</title>

- `<author>Charles Williams</author>`
- 
- `<pub: name>Wrox</pub: name>`
- 
- `</book>`
- `</bookList>`

- 最新修改过的文档使用两个命名空间：默认命名空间

xmlns="<http://site1.com>"; pub 命名空间 xmlns:pub="<http://site2.com>"。默认命

名空间不带前缀，因此文档中所有没有前缀的元素都使用默认命名空间。请注意，`<publisher/>`元素被`<pub:name/>`元素所替换。

- 当对包含命名空间的 XML 文档进行处理时，必须声明这些命名空间才能使用 XPath 表达式。MSXML DOM 文档公开了 `setProperty()` 方法，该方法用来设置对象的第二级属性。特定属性 `SelectionNamespaces` 必须以命名空间别名对任何默认或外部命名空间进行设置。除了使用 `setProperty()` 方法外，命名空间的声明与在 XML 文档是一样的：

- ```
var sNameSpace = "xmlns:na='http://site1.com'
xmlns:pub='http://site2.com'";
```
- `oXml Dom.setProperty("SelectionNamespaces", sNameSpace);`
- 命名空间 `na` 和 `pub` 表示 XML 文档中使用的命名空间。请注意，命名空间前缀 `na` 定义为默认命名空间。当通过 XPath 选择节点时，MSXML 并不能识别默认命名空间，所以默认命名空间声明前缀别名是必需的。现在完成对 `SelectionNamespaces` 属性的设置，那么开始在文档中选择节点：

- ```
var oRoot = oXml Dom.documentElement;
```
- ```
var sXPath = "na:book/pub:name";
```
- ```
var cPublishers = oRoot.selectNodes(sXPath);
```
- 
- ```
if (cPublishers.length > 0) {
```
- ```
 alert(cPublishers.length + " <pub:name/> elements
```
- found with " + sXPath);
- ```
}
```
- XPath 表达式使用 `SelectionNamespaces` 属性中定义的命名空间，并选择所有 `<pub:name/>` 元素。本示例返回包含五个元素的 `NodeList`，接着就可以使用这个 `NodeList` 了。

-

- 4.2.4 Firefox 中的 XPath

- Firefox 的 XPath 实现遵循 DOM 标准，不过与 IE 中的实现差别较大。Firefox 的实现版本允许 XPath 表达式以相同方式在 HTML 和 XML 文档中运行。这里最主要的对象有两个：XPathEvaluator 和 XPathResult。

- XPathEvaluator 类使用 evaluate() 方法对给定的 XPath 表达式进行求值，evaluate() 方法包含五个参数：需要计算的 XPath 表达式字符串，表达式执行的上下文节点，命名空间解析器（处理表达式中命名空间的函数），结果类型（允许 10 种不同的结果类型）以及包含结果的 XPathResult 对象（如果参数为 null，那么返回新的 XPathResult 对象）。

- 在继续深入讨论之前，理解 evaluate() 方法返回的不同结果类型是很重要的。它们分别是：

- **q** XPathResult.ANY_TYPE，返回不确定类型。该方法返回的类型由表达式计算的结果决定。

- **q** XPathResult.ANY_UNORDERED_NODE_TYPE，返回通过 singleNodeValue 属性访问的某个节点的节点集合，如果没有匹配的节点，那么返回 null。返回的节点集合不一定按出现的顺序排列。

- **q** XPathResult.BOOLEAN_TYPE，返回布尔值。

- **q** XPathResult.FIRST_ORDERED_NODE_TYPE，返回某个节点的节点集合。该节点使用 XPathResult 类的 singleNodeValue 属性访问。返回的节点是文档中第一个出现的节点。

- **q** XPathResult.NUMBER_TYPE，返回数字值。

- **q** XPathResult.ORDERED_NODE_ITERATOR_TYPE，返回文档顺序的节点集合（使用 iterateNext() 方法遍历）。因此，可以容易地访问集合中每个独立的节点。

- **q** XPathResult.ORDERED_NODE_SNAPSHOT_TYPE，返回文档顺序的节点集合（结果集合的快照）。任何对文档中节点的修改都不影响结果。

- **q** XPathResult.STRING_TYPE，返回字符串值。

- **q** XPathResult.UNORDERED_NODE_ITERATOR_TYPE，返回可以遍历的节点集合，然而，节点的顺序与其在文档中出现的顺序不一定一致。

- **q** XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE，返回无序快照节点集合。任何对文档中节点的修改都不影响结果。

- 最常用的结果类型是 `XPathResult.ORDERED_NODE_ITERATOR_TYPE`:
- `var oEvaluator = new XPathEvaluator();`
- `var sXPath = "book/author";`
- `var oResult = oEvaluator.evaluate(sXPath, oXmlDom.documentElement, null,`
- `XPathResult.ORDERED_NODE_ITERATOR_TYPE, null);`
-
- `var aNodes = new Array;`
-
- `if (oResult != null) {`
- `var oElement;`
- `while (oElement = oResult.iterateNext()) {`
- `aNodes.push(oElement);`
- `}`
- `}`

• 这段代码创建了一个 `XPathEvaluator` 对象，该对象用于计算文档根节点上下文中的 XPath 表达式 `book/author`。因为结果类型是 `ORDERED_NODE_ITERATOR_TYPE`，所以将返回一个可以使用 `iterateNext()` 方法进行遍历的节点集合。

• `iterateNext()` 方法类似于 DOM 节点的 `nextSibling` 属性，在结果集合中选择下一个节点，当到达结果集合结尾时返回 `null`。该函数可以在 `while` 循环中使用，只要 `oElement` 不为 `null`，那么就可以通过 `push()` 方法将其添加到 `aNodes` 数组中。填充数组的方法与 IE 中的方法类似，因此，可以容易地在 `for` 循环中使用或访问每个数组元素。

• 4.2.5 使用命名空间解析器

• 在 `evaluate()` 方法的语法中，你将看到一个对命名空间解析器的引用。命名空间解析器（`namespace resolver`）是一个将 XPath 表达式中命名空间前缀解析成命名空间 URI 的函数。该命名空间解析器函数可以由你自行命名，但是要求传入一个字符串参数（需要解析的前缀）。

• 解析器对参数所提供的前缀进行检查，并返回相关的命名空间 URI。为了在 IE 示例中使用命名空间 URI，可以编写下列的解析器：

- function nsResolver(sPrefix) {
- switch (sPrefix) {
- case "na":
- return "<http://site1.com>";
- break;
- case "pub":
- return "<http://site2.com>";
- break;
- default:
- return null;
- break;
- }
- }

• 有了上面的解析器，那么就可以在 IE 命名空间示例中对修改后的 books.xml 文档使用下列 XPath 表达式：

- var sXPath = "na:book/pub:name";
- var oEvaluator = new XPathEvaluator();
-
- var oResult =

oEvaluator.evaluate(sXPath,oXmlDom.documentElement,nsResolver,

- XPathResult.ORDERED_NODE_ITERATOR_TYPE, null);
- var aNodes = new Array;
- if (oResult != null) {
- var oElement;
- while (oElement = oResult.iterateNext()) {
- aNodes.push(oElement);
- }
- }

- 该示例与上一段计算表达式的代码类似。不过，请注意对 `evaluate()` 方法的补充：之前编写的 `nsResolver()` 函数的指针，作为参数传入 `evaluate()` 方法来处理 XPath 表达式的命名空间。其余部分应该相当熟悉了。使用 `XPathResult` 类的 `iterateNext()` 方法遍历返回结果 `NodeList`，并转换成数组。

- 正如你所见，Firefox 的 XPath 实现与微软提供的方法有很大的不同，因此使用一个跨浏览器兼容的库会使得执行 XPath 计算更加简单。

• 4.2.6 跨浏览器兼容的 XPath

- 本书作者开发的 `zXml` 库通过一个公共接口提供了跨浏览器兼容的 XPath 功能。负责提供 XPath 功能的对象是 `zXPath`，该对象包含两个方法。

- 第一个方法是 `selectSingleNode()`。该方法与 IE 方法同名，返回与模式匹配的第一个节点。与 IE 实现不同的是，该方法接受三个参数：上下文节点、XPath 表达式字符串以及包含命名空间声明的字符串。命名空间字符串的格式如下所示：

- `"xml ns:na='http://site1.com' xml ns:pub='http://site2.com'`

- `xml ns:ns='http://site3.com' "`

- 如果文档没有命名空间，那么 `selectSingleNode()` 方法只需要前两个参数。

- `selectSingleNode()` 方法所返回的结果是所选中的 XML 节点，或者没有查到匹配项时将为 `null`。如果浏览器不支持 XPath，那么会抛出错误，声明浏览器没有安装 XPath 引擎。下面这个例子将基于文档元素来执行 XPath 表达式：

- `var oRoot = oXmlDom.documentElement;`

- `var oNode = zXPath.selectSingleNode(oRoot, "book/author", null);`

-

- `if (oNode) {`

- `alert(oNode.xml);`

- `}`

- 该例子从文档根节点的上下文中查找 `<book/>` 元素中的第一个 `<author/>` 元素。如果找到，就在警告框中显示序列化后的 XML 数据。

• XPath 的第二个方法是 `selectNodes()`，该方法与 IE 的 `selectNodes()` 方法相当类似，而且都是返回节点集合。该方法的语法与上面的 `selectSingleNode()` 方法很接近，同样的参数，同样的命名空间规则，并且当浏览器没有安装 XPath 引擎时，也将抛出同样的错误提示。下面的例子演示了 `selectNodes()` 方法的使用：

```
• var sNamespace = "xmlns:na='http://site1.com'
xmlns:pub='http://site2.com' ";

• var oRoot = oXmlDom.documentElement;

• var sXPath = "na:book/pub:name";

• var oNodes = zXPath.selectNodes(oRoot, sXPath, sNamespace);

•

• if (oNodes.length > 0) {

•     alert(oNodes.length);

• }
```

• 这个例子与 `selectSingleNode()` 的例子相当类似，都是查找文档中包含命名空间的所有作者元素。如果结果集合个数大于 0，那么显示结果的个数。

• XPath 是一个强大的工具，可以在 XML 文档中遍历并选择特定的节点，尽管设计者最初并没有考虑将其作为一个独立的工具。相反，XPath 只是为 XSLT（可扩展样式表语言转换）所创建。

• 4.3 浏览器对 XSLT 的支持

• XSL 是一组为 XML 数据转换设计的语言。XSL 包含三种主要语言：XSLT（可扩展样式表语言转换），用来将 XML 文档转换成另一种 XML 文档的语言；前一小节中介绍的 XPath；XSL-FO（XSL 格式化对象），用来描述如何呈现已转换数据的语言。因为现在尚没有浏览器支持 XSL-FO，因而所有的转换都必须使用 XSLT 来完成。

• 4.3.1 XSLT 概述

• XSLT 是一门基于 XML 的语言，它的设计目的是用来将一个 XML 文档转换成另一种格式的数据。这样的定义也许会让人觉得 XSLT 并不是一种相当有用的技术，但事

实并非如此。XSLT 最流行的用途是将 XML 文档转换成 HTML 文档，这也正是本节所要介绍的内容。

- XSLT 文档和 XML 文档相比，没有什么特别之处，所以它们与所有 XML 文档一样，遵循相同的规则：它们必须包含 XML 声明、必须有唯一的根元素以及必须有正确的格式。

- 在此，我们将再次以 books.xml 为例。该文件的内容可以使用 XSLT 将其转换成 HTML，而无需手动地将其构造成 DOM 结构。首先我们需要一个 XSLT 文档 books.xsl，其中包含 XML 声明和根元素：

- `<?xml version="1.0" encoding="UTF-8" ?>`
-
- `<xsl:stylesheet version="1.0"`
`xmlns:xsl="http://www.w3.org/1999/XSL/Transform">`
- `<xsl:output method="html" omit-xml-declaration="yes" indent="yes" />`
-
- `</xsl:stylesheet>`

- XSLT 文档的文档元素是 `<xsl:stylesheet/>`。在该元素中指定 XSL 版本，并声明 xsl 命名空间。这些必要的信息决定 XSLT 处理器的行为；否则将会抛出错误。xsl 前缀相当重要，因为所有 XSL 指令都可以清晰地与文档中其他代码分离开，不论是从视觉上分离还是在逻辑上分离。

- `<xsl:output/>` 元素定义结果输出的格式。该示例中，通过忽略 XML 声明且缩进元素，将结果转换成 HTML 数据。可以根据需要，将格式指定为普通文本、XML 或者 HTML 数据。

- 与应用程序一样，转换必须有入口点。XSLT 是一门基于模板的语言，处理器根据模板匹配规则对 XML 文档进行操作。该示例中，第一个匹配元素是 XML 文档的根节点。该工作使用 `<xsl:template>` 指令完成。指令（directive）通知处理器执行特定的函数。`<xsl:template/>` 指令创建一个模板，当 match 属性中的模式匹配时，使用该模板：

- `<?xml version="1.0" encoding="UTF-8" ?>`

- <xsl:stylesheet version="1.0"

xmlns:xsl="<http://www.w3.org/1999/XSL/Transform>">

- <xsl:template match="/">
- <html>
- <head>
- <link rel="stylesheet" type="text/css" href="books.css" />
- <title>XSL Transformations</title>
- </head>
- <body>
- <xsl:apply-templates />
- </body>
- </html>
- </xsl:template>
- </xsl:stylesheet>

• match 属性的值是一个 XPath 表达式，用来选择正确的 XML 节点。在本例中，就是 books.xml 的根元素（XPath 表达式/总是选择文档元素）。在模板内部包含 HTML 元素，这些元素是转换输出的一部分。<body/>元素内部，可以找到另一个 XSL 指令。

<xsl:apply-templates/>元素通知处理器开始解析文档元素上下文中的所有模板，并进行下一个模板的处理。

- <?xml version="1.0" encoding="UTF-8" ?>
-
- <xsl:stylesheet version="1.0"

xmlns:xsl="<http://www.w3.org/1999/XSL/Transform>">

- <xsl:output method="html" omit-xml-declaration="yes" indent="yes" />
-
- <xsl:template match="/">
-
- <html>
- <head>

- `<link rel="stylesheet" type="text/css" href="books.css" />`
- `<title>XSL Transformations</title>`
- `</head>`
- `<body>`
- `<xsl:apply-templates />`
- `</body>`
- `</html>`
-
- `</xsl:template>`
-
- `<xsl:template match="book">`
- `<div class="bookContainer">`
- `<xsl:variable name="varIsbn" select="@isbn" />`
- `<xsl:variable name="varTitle" select="title" />`
- ``
- `<div class="bookContent">`
- `<h3><xsl:value-of select="$varTitle" /></h3>`
- `Written by: <xsl:value-of select="author" />
`
- `ISBN #<xsl:value-of select="$varIsbn" />`
- `<div class="bookPublisher"><xsl:value-of select="publisher"`
- `/></div>`
- `</div>`
- `</div>`
- `</xsl:template>`
-
- `</xsl:stylesheet>`

• 新的模板匹配所有<book/>元素，所以当处理器到达 XML 文档中每个<book/>时，就会应用该模板。该模板中前两个 XSL 指令是<xsl:variable/>，该指令的作用是定义变量。

- XSL 中的变量主要在 XPath 表达式或属性中使用（其中不破坏 XML 语法时不能使用元素）。<xsl:variable/>元素有两个属性：name 和 select。name 属性就是用来设置变量的名称。select 属性指定一个 XPath 表达式，并在变量中存储匹配值。最初声明后，变量就可以通过\$符号进行引用（所以定义为 varIsbn 的变量可以通过\$varIsbn 引用）。

- 第一个变量\$varIsbn，赋值为<book/>元素的 isbn 属性值。第二个变量\$varTitle，赋值为<title/>元素的值。这两部分信息将用在 HTML元素的属性中。为了在属性中输出变量，可以给变量名称加上大括号：

-

- 如果不加大括号，那么输出就是字符串"\$varTitle"和"\$varIsbn"。

• 在 XSL 指令属性中使用诸如 **select** 的变量，是该规则的一种例外情况。在这些属性中使用大括号将会出现错误，并导致文档转换失败。

- 在这个例子中，余下的 XSL 指令是<xsl:value-of/>元素。这些元素根据 select 属性获取匹配变量或节点的值。select 属性与<xsl:variable/>中的 select 属性作用一样：都是 XPath 表达式，并选出匹配表达式的节点或变量。该模板中<xsl:value-of/>的第一个实例引用\$varTitle 变量（注意没有大括号），所以使用该变量的值。接下来，使用<author/>元素的值，\$varTitle 和<publisher/>也一样。

- 为了在浏览器中对 XML 文档进行转换，必须指定一个样式表。在 books.xml 中，在 XML 声明部分后添加下面的代码行：

- <?xml-stylesheet type="text/xsl" href="books.xsl"?>

- 这将通知 XML 处理器对该文档应用 books.xsl 样式表。在 Web 浏览器中查看这个修改后的 XML 文档将不再显示 XML 结构，而将显示转换成 HTML 后的结果。然而，使用该指令对 JavaScript 并不起作用，如果想达到这样的功能，必须使用一些特定的对象。

• 4.3.2 IE 中的 XSLT

- IE 中有两种方法来完成 XML 文档的转换，这两种方法都必需使用 MSXML。从 MSXML 3.0 版开始，MSXML 对 XSLT1.0 提供完整的支持。如果没有 Windows XP 或

IE 6, 那么应当升级。你可以在 <http://msdn.microsoft.com/XML/XMLDownloads/>中找到并下载最新的 MSXML。

- 首先,使用最简单的方法将 XML 和 XSLT 文档载入到各自的 XML DOM 对象中:

- `var oXmlDom = zXmlDom.createDocument();`

- `var oXslDom = zXmlDom.createDocument();`

-

- `oXmlDom.async = false;`

- `oXslDom.async = false;`

-

- `oXmlDom.load("books.xml");`

- `oXslDom.load("books.xsl");`

- 当载入文档后,就可以调用 `transformNode()` 方法进行转换:

- `var sResults = oXmlDom.transformNode(oXslDom);`

- `transformNode()` 方法的参数为 XML DOM 对象 (本示例为 XSL 文档), 并以字符串形式返回转换后的数据。但是不一定要在文档对象处调用 `transformNode()` 方法, 也可以在 XML 文档的任何元素处调用该方法:

- `var sResults =`

`oXmlDom.documentElement.firstChild.transformNode(oXslDom);`

- `transformNode()` 方法只会从调用的元素开始转换该元素及其子元素。示例中转换了第一个 `<book/>` 元素, 如图 4-3 所示。这是因为对没有子元素的元素调用 `transformNode()` 方法只会转换一个节点。



• 图 4-3

• IE 中第二个转换方法稍微有些复杂，但提供更多的控制权和特性。该过程需要创建多个 MSXML 库中的对象。这个稍长过程的第一步是创建一个线程安全的 XML DOM 对象，并将 XSL 样式表载入该对象：

- `var oXml Dom = zXml Dom. createDocument ();`
- `oXml Dom. async = false;`
- `oXml Dom. load ("books. xml ");`
-
- `var oXsl Dom = new ActiveXObject ("Msxml 2. FreeThreadedDOMDocument. 3. 0");`
- `oXsl Dom. async = false;`
- `oXsl Dom. load ("books. xsl ");`

• FreeThreadedDOMDocument 类是另一个 ActiveX 类，且是 MSXML 库中的一部分。必须使用 FreeThreadedDOMDocument 类来创建 XSLTemplate 对象，如该示例所示（下一个示例将显示 XSLTemplate 对象的创建）。在 MSXML 的早期版本中，对每个 transformNode() 方法的调用都需要对 XSL 样式表进行重新编译，这在一定程度上会降低转换的速度。如果基于 FreeThreadedDOMDocument，编译后的样式表会放入缓存供以后使用，直到从内存中移出。

- XML DOM 对象创建后，还需要创建另一个 ActiveX 对象，即 XSL 模板对象：

- `var oXsl Template = new ActiveXObject ("Msxml 2. XSLTemplate. 3. 0");`

- `oXslTemplate.stylesheet = oXslDom;`

• XSLTemplate 类用于缓存 XSL 样式表并创建 XSLProcessor，所以当模板创建后，XSL 文档赋值给 XSLTemplate 类的 stylesheet 属性，实现缓存功能并载入 XSL 样式表。

• 该过程的下一步是创建 XSLProcessor，通过调用 XSLTemplate 类的 `createProcessor()` 方法创建：

- `var oXslProcessor = oXslTemplate.createProcessor();`

- `oXslProcessor.input = oXmlDom;`

• 创建处理器后，其 `input` 属性将赋值为 `oXmlDom`，也就是包含待转换 XML 文档的 XML DOM 对象。现在，处理器所需一切都准备就绪，所以余下工作就是实际的转换和获取输出：

- `oXslProcessor.transform();`

- `document.body.innerHTML = oXslProcessor.output;`

• 与 `transformNode()` 不一样，`transform()` 方法并不以字符串形式返回结果。为了得到转换的输出，需要使用 XSLProcessor 对象的 `output` 属性。其整个过程比 `transformNode()` 方法需要更多的代码，并得到相同的结果。为什么要使用这样的过程呢？

• MSXML 提供一些额外的可以用于转换的方法。第一个方法是 `addObject()`。该方法往样式表中添加一个 JavaScript 对象，可以在转换后的文档中调用方法和 `output` 属性值。考虑下面的对象：

- `var oBook = {`
- `propertyOne : "My Current Books",`
- `methodOne : function () {`
- `alert("Welcome to my Book List");`
- `return "";`
- `}`
- `};`

• 如果想要在转换中使用该信息，该怎么样呢？使用 `addObject()` 方法，就可以将该信息传到 XSLT 样式表，这里需要传入两个参数：`oBook` 对象和命名空间 URI。所以，

要将"http://my-object"的命名空间 URI 添加到该对象，可以按下述代码所示的方法来完成：

- `var oXml Dom = zXml Dom. createDocument();`
- `oXml Dom. async = false;`
- `oXml Dom. load("books. xml ");`
-
- `var oXsl Dom = new ActiveXObject("Msxml 2. FreeThreadingDOMDocument. 3. 0");`
- `oXsl Dom. async = false;`
- `oXsl Dom. load("books. xsl ");`
-
- `var oXsl Template = new ActiveXObject("Msxml 2. XSLTemplate. 3. 0");`
- `oXsl Template. stylesheet = oXsl Dom;`
-
- `var oXsl Processor = oXsl Template. createProcessor();`
- `oXsl Processor. input = oXml Dom;`
-
- `oXsl Processor. addObject(oBook, "http://my-object");`
-
- `oXsl Processor. transform();`
- `document. body. innerHTML = oXsl Processor. output;`

• oBook 对象将传到 XSLProcessor 中，意味着 XSLT 样式表可以使用该对象。现在必须对 XSL 文档进行修改，才能够查找该对象并使用其信息。第一件事是添加新的命名空间到根元素<xsl:stylesheet/>中。该命名空间将匹配 addObject()中的命名空间：

- `<xsl:stylesheet version="1.0"`

`xmlns:xsl="http://www.w3.org/1999/XSL/Transform"`

-
- `xmlns:bookObj="http://my-object">`

• 前缀 bookObj 可以用来访问该信息。现在命名空间和前缀都准备好了，应当往文档中添加一些<xsl:value-of/>元素来获取对象成员：

- `<xsl:template match="/">`
-
- `<html>`
- `<head>`
- `<link rel="stylesheet" type="text/css" href="books.css" />`
- `</head>`
- `<body>`
-
- `<xsl:value-of select="bookObj:methodOne()" />`
- `<div align="center">`
- `<xsl:value-of select="bookObj:get-propertyOne()" />`
- `</div>`
- `<xsl:apply-templates />`
- `</body>`
- `</html>`
-
- `</xsl:template>`

• 请记住，`<xsl:value-of/>`XSL 指令获取元素或对象的值。第一个`<xsl:value-of/>`指令获取（或调用）`methodOne()`，该方法发送一个欢迎用户到该页面的信息。第二个`<xsl:value-of/>`指令与第一个非常相似，该指令获取 `oBook` 对象的 `propertyOne` 属性的值。当在浏览器中显示转换后的输出时，用户会在页面顶部看到“My Current Books”的字句。

• 当在转换中使用对象时，所有属性和方法都必须返回 **XSLProcessor** 能够理解的值：字符串、数字以及布尔值。返回其他值将会在执行转换时抛出 JavaScript 错误。

• XSLProcessor 另一个有用的特性是 `addParameter()` 方法。与将对象进行转换不同，该方法的参数是 XSLT 标准部分。参数传入 XSL 样式表，并可以作为变量使用。为了指定参数，并传入名称和值，如下所示：

- `var oXslProcessor = oXslTemplate.createProcessor();`

- `oXslProcessor.input = oXmlDom;`
- `oXslProcessor.addParameter("message", "My Book List");`

• 这段代码将往 XSLProcessor 中添加"message"参数。当执行 XSL 转换时，处理器使用参数的值，"My Book List"，并将其放置在相应的位置。XSL 中的参数使用 `<xsl:param/>` 指令：

- `<xsl:param name="message" />`
- 请注意，name 属性匹配传入 `addParameter()` 的名称。该参数接收使用之前介绍的变量语法获取的值 "My Book List"：

- `<xsl:value-of select="$message" />`
- 本示例中，通过 `<xsl:value-of/>` 指令得到参数值。更新后的 XSL 样式表如下所示：
- `<xsl:stylesheet version="1.0"`

`xmlns:xsl="http://www.w3.org/1999/XSL/Transform">`

- `<xsl:param name="message" />`
- `<xsl:template match="/">`
- `<html>`
- `<head>`
- `<link rel="stylesheet" type="text/css" href="books.css" />`
- `</head>`
- `<body>`
- `<xsl:value-of select="$message" />`
- `<xsl:apply-templates />`
- `</body>`
- `</html>`
- `</xsl:template>`

• 更新后的样式表增加两行新的代码。第一行代码是 `<xsl:param/>` 指令，第二行代码则是获取参数值的 `<xsl:value-of/>` 指令。参数声明可以在 XSL 文档的任何位置出现。这段代码在文档顶部进行参数声明，但是并不限制只能在文档顶部出现。

- 使用 XSLProcessor 的最后一个特性是它的速度。XSLProcessor 编译 XSL 样式表，所以后续的转换可以使用相同的样式表结果，转换速度更快（与使用 transformNodes() 相比）。如果要这样做，那么需要使用 XSLProcessor 对象的 reset() 方法。该方法清空 input 和 output 属性，但没有清空 stylesheet 属性。这就让处理器为下次基于相同样式表的转换做好了准备。

• 4.3.3 Firefox 中的 XSLT

- 与 XML 和 XPath 一样，Firefox 的 XSLT 实现与 IE 中的实现存在许多不同之处。Firefox 实现一个 XSLTProcessor 类来执行转换，不过相似之处也就仅此而已。

- Firefox 中执行转换的第一步是将 XML 和 XSLT 文档载入到 DOM 对象中：

- var oXmlDom = zXmlDom.createDocument();

- var oXslDom = zXmlDom.createDocument();

-

- oXmlDom.async = false;

- oXslDom.async = false;

-

- oXmlDom.load("books.xml");

- oXslDom.load("books.xsl");

- XSLTProcessor 类公开了 importStylesheet() 方法，该方法的参数为包含 XSLT 文档的 XML DOM 对象：

- var oXsltProcessor = new XSLTProcessor();

- oXsltProcessor.importStylesheet(oXslDom);

- 最后调用转换方法。这里有两类方法：transformToDocument() 和 transformToFragment()。transformToDocument() 方法的参数为 XML DOM 对象，并返回包含转换的新 XML DOM 文档。通常情况下，将以下述的形式来使用它：

- var oNewDom = oXsltProcessor.transformToDocument(oXmlDom);

- 作为结果返回的 DOM 对象与其他 XML DOM 对象的用法类似。可以根据 XPath 选择特定节点，基于属性和方法遍历节点结构，甚至在另一个转换中使用它。

- `transformToFragment()`方法返回一个文档片段，恰如其名，附加到另一个 DOM 文档上。该方法需要两个参数：第一个参数是要转换的 XML DOM 对象，第二个参数是计划附加结果的 DOM 文档：

- `var oFragment = oXsltProcessor.transformToFragment(oXmlDom, document);`
- `document.body.appendChild(oFragment);`

- 在示例中，返回结果的文档片段附加到文档对象的主体中。请注意，结果片段可以附加到作为参数传入 `transformToFragment()` 方法的 DOM 对象内的任意节点。

- 但是如果想像微软的 `transformNode()` 方法实现一样，返回的字符串作为转换结果，该怎么样呢？那么可以使用之前介绍的 `XMLSerializer` 类。只需要将转换结果作为参数传到 `serializeToString()` 方法中：

- `var oSerializer = new XMLSerializer();`
- `var str = oSerializer.serializeToString(oNewDom);`

- 如果使用 `zXml` 库，则只需简单使用 `xml` 属性即可：

- `var str = oFragment.xml;`

- `XSLTProcessor` 类还可以设置传入 XSL 样式表的参数。`setParameter()` 方法可以很容易地完成该功能。该方法接受三个参数：命名空间 URI、参数名称以及参数值。例如：

- `oXsltProcessor.importStylesheet(oXslDom);`
-
- `oXsltProcessor.setParameter(null, "message", "My Book List");`
-

- `var oNewDom = oXsltProcessor.transformToDocument(oXmlDom);`

- 在该示例中，赋给参数 `message` 的值为 `"My Book List"`。第一个参数命名空间 URI 传入的值为 `null`，那么样式表中的参数就不需要指定前缀和相应的命名空间 URI：

- `<xsl:param name="message" />`

- `setParameter()` 方法必须在调用 `transformToDocument()` 或 `transformToFragment()` 前调用，否则转换中就不能使用所设置的参数值。

- 4.3.4 跨浏览器的 XSLT

- 在前面几小节中你已经看到，通过 `zXml` 库可以很容易地实现跨平台的 XML 数据处理。现在，我们还将使用该库来进行 XSLT 转换。`zXml` 库中只有一个 XSLT 的方法：`transformToText()`。`transformToText()` 方法返回转换的文本，接受两个参数：需要转换的 XML 文档和 XSL 文档：

- `var sResult = zXslt.transformToText(oXmlDom, oXslDom);`
- 根据方法名称可以得知，它返回的结果是一个字符串，然后将转换结果（`sResult`）添加到 HTML 文档：

- `var oDiv = document.getElementById("transformedData");`
- `oDiv.innerHTML = sResult;`
- 这也许是 `zXml` 库中最简单的对象。

- 4.3.5 重访“最佳选择”

- 让我们再来看看前面提到的在线书店的运行情况。客户们很喜欢已经实现的“最佳选择”功能，但是近来收到一些反馈，他们希望能够看到上周的“最佳选择”。我们还是决定使用 Ajax 来解决该问题。

- 浏览器使用 `XMLHttpRequest` 获取书籍列表，并将请求的 `responseText` 载入到 XML DOM 对象中。样式表载入到自己的 XML DOM 对象中，书籍列表的 XML 数据转换成 HTML，并将转换后的 HTML 元素写到页面中。为了提高实用性，在右上角提供一个链接切换列表。

- 该解决方案的第一步是通过 `XMLHttpRequest` 获取 XML 文件。这是代码的开始部分，也是该应用程序的入口点，所以将这部分代码封装到 `init()` 函数中：

- `function init(sFilename) {`
- `var oReq = XMLHttpRequest.createRequest();`
- `oReq.onreadystatechange = function () {`
- `if (oReq.readyState == 4) {`
- `// 仅当"OK"`
- `if (oReq.status == 200) {`

- transformXml (oReq.responseText);
- }
- }
- };
- oReq.open("GET", sFilename, true);
- oReq.send();
- }

• init()函数接受一个参数：需要载入的 XML 文件名。为了能够兼容各种浏览器，将使用 zXml 创建 XMLHttpRequest 对象。这是一个异步请求，所以必须使用 onreadystatechange 事件处理函数检查 readyState 属性。当请求正常返回时，responseText 作为参数传入 transformXml () 函数中：

- function transformXml (sResponseText) {
- var oXml Dom = zXml Dom.createDocument();
- oXml Dom.async = false;
- oXml Dom.loadXML(sResponseText);
-
- var oXsl Dom = zXml Dom.createDocument();
- oXsl Dom.async = false;
- oXsl Dom.load("books.xsl");
-
- var str = zXslt.transformToText(oXml Dom, oXsl Dom);
- document.getElementById("divBookList").innerHTML = str;
- }

• 调用 transformXml () 函数使用 loadXML() 方法将传入的响应文本载入 XML DOM 对象中。XSL 样式表也同样载入到一个 XML DOM 对象中，这两个 XML DOM 对象都作为参数传入 zXml 库的 transformToText() 方法中。转换结果是一个字符串，通过 innerHTML 属性可以将其添加到文档的元素中。该函数的作用就是显示本周的书籍列表。

• 现在已经完成一部分代码，不过还欠缺列表切换功能。为了更容易完成该功能，需要编写另一个函数，但是首先，当用户点击链接时，应用程序要能够知道应该载入哪

个列表。这可以通过布尔型变量 `blsThisWeek` 很容易地完成处理。当载入本周书籍列表时，`blsThisWeek` 为 `true`，否则为 `false`。因为本周列表已经载入，所以现在 `blsThisWeek` 变量设为 `true`：

- `var blsThisWeek = true;`
- 用户切换书籍列表的链接使用 `onclick` 事件，所以下面的函数将处理该事件：
- `function changeList() {`
- `var aChanger = document.getElementById("aChanger");`
-
- `if (blsThisWeek) {`
- `aChanger.innerHTML = "This Week's Picks";`
- `init("lastweekbooks.xml");`
- `blsThisWeek = false;`
- `} else {`
- `aChanger.innerHTML = "Last Week's Picks";`
- `init("thisweekbooks.xml");`
- `blsThisWeek = true;`
- `}`
- `return false;`
- `}`

在这段代码中，将通过 `getElementById()` 方法来获取链接 (`aChanger`)。`blsThisWeek` 变量的值将被检查。根据变量的值，向 `init()` 函数传入正确的文件名，载入正确的书籍列表。这就完成了获取新的列表，转换数据，并写到页面中。注意到链接文本也随之改变，这就为用户下次点击链接的操作提供提示。`blsThisWeek` 变量也改变，所以当下次用户点击链接时，载入正确的列表。最后，该函数将返回 `false`。由于该函数是一个链接的事件处理函数，返回其他值将导致该链接被“点击”，从而转到了应用程序中的其他位置中。

- 最后，可以使用 **HTML** 来完成这个迷你的应用程序，以下就整个 **HTML** 文档：

- `<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"`

- `"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">`
- `<html xmlns="http://www.w3.org/1999/xhtml" >`
- `<head>`
- `<title>Book XML Exercise</title>`
- `<link rel="stylesheet" type="text/css" href="books.css" />`
- `<script type="text/javascript" src="zxml.js"></script>`
- `<script type="text/javascript">`
- `function init(sFilename) {`
- `var oReq = zXmlHttp.createRequest();`
- `oReq.onreadystatechange = function () {`
- `if (oReq.readyState == 4) {`
- `// 当且仅当"OK"`
- `if (oReq.status == 200) {`
- `transformXml(oReq.responseText);`
- `}`
- `}`
- `};`
- `oReq.open("GET", sFilename, true);`
- `oReq.send();`
- `}`
-
- `function transformXml(sResponseText) {`
- `var oXmlDom = zXmlDom.createDocument();`
- `oXmlDom.async = false;`
- `oXmlDom.loadXML(sResponseText);`
-
- `var oXslDom = zXmlDom.createDocument();`
- `oXslDom.async = false;`
- `oXslDom.load("books.xsl");`

-
- var str = xslt.transformToText(oXmlDom, oXslDom);
- document.getElementById("divBookList").innerHTML = str;
- }
-
- var blsThisWeek = true;
-
- function changeList() {
- var aChanger = document.getElementById("aChanger");
- if (blsThisWeek) {
- aChanger.innerHTML = "This Week's Picks";
- init("lastweekbooks.xml");
- blsThisWeek = false;
- } else {
- aChanger.innerHTML = "Last Week's Picks";
- init("thisweekbooks.xml");
- blsThisWeek = true;
- }
- return false;
- }
- </script>
- </head>
- <body onload="init('thisweekbooks.xml')">
- Last Week's
Picks
- <div id="divBookList"></div>
- </body>
- </html>

- 要运行这个应用程序，必须将其部署在 Web 服务器上，因为它使用了 XMLHttp。

对于这个例子而言，任何 Web 服务器都可以正常工作。只需要将 HTML 文件、zXml 库和 CSS 文件放到 Web 服务器的 booklists 目录中。接着打开浏览器，输入链接 `http://localhost/booklists/book.htm` 即可。

• 4.4 小结

- 本章介绍了如何在 IE 和 Firefox 中创建并遍历 XML DOM 对象，并了解了这两种浏览器实现的不同之处。再次使用了跨浏览器兼容的 XML 库 zXml，通过唯一的接口可以容易地创建、遍历并操作 XML DOM 对象。我们还学习到如何使用 JavaScript 载入 XML 数据，并输出到页面中。

- 在第二部分中，简要地介绍了 XPath 这门为 XML 文档提供的强大语言，了解到 IE 和 Firefox 支持 XPath 和命名空间的方法，这两种浏览器的实现存在较大的差异。为了消除差异，引入了 zXml 库中的 zXPath 对象，该对象为浏览器提供唯一接口，使得选择所需的节点变得更加容易。

- 最后介绍了 XSLT 转换，以及如何使用 MSXML 和 Firefox 的 XSLTProcessor 类完成转换。尽管这两个接口存在一些相同之处，但仍推荐使用 zXml 库中的另一个跨浏览器兼容的 zXslt 对象来消除差异，通过调用该对象一个方法，就能够在两种浏览器中完成 XSLT 转换。

- XML 的引入掀开了信息共享的新纪元。先前，实现数据共享是困难的。许多公司都拥有专有的传输协议和数据格式，而且没有公开。在网站使用 HTML 之外的协议来实现数据传输是一个前所未闻、难以接受的想法。但在 1998 年，微软在 IE 4.0 中引入一个名为 Active Channels（活动通道）的新功能后，事情发生了改变。在微软开发的通道定义格式（Channel Definition Format, CDF）的基础上，网站可以通过活动通道将数据传输（或聚合）到设置为活动桌面（Active Desktop）的用户桌面上。但使用活动通道的问题是，它对于平常的用户支持不够。每个人都可以创建一个通道，但业内缺乏创建和管理 CDF 文件的简单工具。活动通道的主要用户——大型媒体公司——向用户推播大量的广告信息，因而大大提高了通道所需的带宽总量。另外，在使用活动通道方面的需求和可认知的价值都不大。当整个聚合（syndication）概念似乎伴随着活动通道一起消亡，且 CDF 也没有成为万维网联盟的推荐标准时，RSS 出现了。

• 5.1 RSS

- 1999 年 3 月，Netscape 公司发布了 My Netscape 门户，用户在这一个地方就可以访问所有的新闻。其概念很简单：从许多新闻源获取信息，然后将它们显示在 My Netscape 上。为了实现这个想法，Netscape 通信公司的 Dan Libby 开发了 RDF 网站摘要（RDF Site Summary，RSS），这是一种基于资源描述框架（Resource Description Framework）的 XML 数据格式。它后来也就成为了大家熟知的 RSS 0.9。

- 在 RSS 0.9 出现后不久，Userland 软件公司的 Dave Winer 与 Libby 开始探讨 RSS 0.9 格式。Winer 也为其网站开发了一个名为 ScriptingNews 的 XML 格式，他认为该 XML 格式与 RSS 0.9 整合在一起并做一些简化，可以获得一个更好、复用性高的格式规范。1999 年 7 月，Libby 发布了一个新的原型，丰富网站摘要（Rich Site Summary），其缩写仍然是 RSS，也就是 RSS 0.91。然后，My Netscape 开始使用 RSS 0.91 并一直延用到 2001 年，直到需要对外部 RSS 提要提供支持时才放弃。Netscape 很快就对 RSS 失去了兴趣，RSS 成了无主孤儿。这也是 RSS 格式分裂成两个不同版本的原因。

- 一些开发人员和其他感兴趣的合作伙伴通过一个邮件列表组织起来，决定继续开发 RSS。该组织名为 RSS-DEV（<http://groups.yahoo.com/group/rss-dev/>），于 2000 年 12 月完成了一个新的版本 RSS 1.0。RSS 1.0 是基于最初的 RDF 网站摘要（RSS 0.9）的，并且通过将原来 0.9 版本模块化来扩展它。这些模块是任何人都可以创建的命名空间，不需要修改规范就可以添加新的功能。另外，要注意 RSS 1.0 是 RSS 0.9 的后代，而与 RSS 0.91 无关，这点很重要。

- 与此同时，Winer 也声明他是 RSS 的所有者，将继续开发他自己的版本，并发布了 RSS 2.0（全名改成了 Really Simple Syndication）。这个新的 RSS 格式则是基于 RSS 0.91 的，也就是由 Winer 和 Libby 一起开发的版本。RSS 2.0 强调的是格式的简单性。当 Winer 加盟哈佛大学后，它将 RSS 2.0 的所有权交给了因特网协会哈佛大学 Berkman 中心，该中心现在通过网址在 <http://blogs.law.harvard.edu/tech/rss> 管理和发布这一规范。现在 RSS 2.0 已经成了广泛应用的 RSS 格式。

- 现在，术语 RSS 包括三个不同版本的 RSS 格式：RSS 0.91、RSS 1.0 以及 RSS 2.0。

- 5.1.1 RSS 0.91

• RSS 0.91 是基于 DTD (Document Type Declarations, 文档类型声明) 的, 可能是 RSS 2.0 发布之前最流行的 RSS 版本。一些统计数据显示, 在 2001 年 RSS 0.91 占据了聚合市场 52% 的份额, 并在 RSS 2.0 发布之前还在稳定地上升。它的流行得益于其简单的特性。现在还在使用 0.91 版的提要已经不多了, 不过 RSS 2.0 的流行应归功于 RSS 0.91 的成功。

• RSS 0.91 的 DTD 共列举了 24 个元素 (比 RSS 0.9 多 14 个), 人和电脑都很容易阅读。让我们来看一个 0.91 的例子:

```
• <?xml version="1.0" encoding="UTF-8" ?>
•
• <!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN"
•   "http://my.netscape.com/publish/formats/rss-0.91.dtd">
•
• <rss version="0.91">
•   <channel>
•     <title>My Revenge</title>
•     <link>http://si thboys.com</link>
•     <description>Dedicated to having our revenge</description>
•     <item>
•       <title>At last!</title>
•       <link>http://si thboys.com/atlast.htm</link>
•       <description>
•         At last we will reveal ourselves to the Jedi. At last we
will have
•         our revenge.
•       </description>
•     </item>
•   </channel>
• </rss>
```

- RSS 0.91 (2.0 也是一样) 的特有特征是所有数据都包含在<channel />元素中。所有定义网站的定义信息以及<item/>元素都包含在<channel />中。这与下一个 RSS 版本 (RSS 1.0) 形成了鲜明对比。

- 5.1.2 RSS 1.0

- RSS 1.0 违背了 0.91 标准的格式, 它遵从的是 0.9 版的 RDF 格式。RSS 1.0 要比上一个版本冗长得多, 不过它的扩展性使其成为吸引人的格式, 特别对于那些基于 RDF 应用程序的开发人员而言。

- 虽然 RSS 1.0 和 RSS 0.91 有许多类同之处, 但是结构上却不同, 只是与 RSS 0.9 相似。

- <?xml version="1.0"?>
-
- <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns="http://purl.org/rss/1.0/">
-
- <channel rdf:about="http://si thboys.com/about.htm">
- <title>My Revenge</title>
- <link> http://si thboys.com</link>
- <description>
- Dedicated to having our revenge
- </description>
- <image rdf:resource="http://si thboys.com/logo.jpg" />
- <items>
- <rdf:Seq>
- <rdf:li resource="http://si thboys.com/atlast.htm" />
-
- </rdf:Seq>
- </items>
- <textinput rdf:resource="http://si thboys.com/search/" />

- `</channel>`
-
- `<image rdf:about="http://si thboys.com/logo.jpg">`
- `<title>The Logo of the Si th</title>`
- `<link>http://si thboys.com/</link>`
- `<url>http://si thboys.com/logo.jpg</url>`
- `</image>`
-
- `<item rdf:about="http://si thboys.com/atlast.htm">`
- `<title>At last!</title>`
- `<link>http://si thboys.com/atlast.htm</link>`
- `<description>`
- `At last we will reveal ourselves to the Jedi. At last we`

will have

- `our revenge.`
- `</description>`
- `</item>`
- `</rdf:RDF>`

- 注意，`<item/>`元素是在`<channel />`元素外面的。在`<channel />`元素中的`<items/>`元素则包含一组`<rdf:Seq/>`元素的值，该元素包含的是在`<channel />`元素之外的引用。正如你所见，它要比 RSS 0.91 更复杂，虽然 RSS 1.0 也获得了一些拥护者，但与其他格式的流行程度是无法相比较的。

- 与 RSS 0.91 不同，RSS 1.0 不是基于 DTD 的，所以在该文档中 DTD 不是必要的。

• 5.1.3 RSS 2.0

- RSS 2.0 几乎与 RSS 0.91 同出一辙。2.0 版引入了许多新的元素，诸如下面例子中的`<author/>`元素，同时与 RSS 1.0 一样支持模块化的扩展。由于从 RSS 0.91 中继承了简单性，同时拥有了与 RSS 1.0 类似的可扩展性，因此 RSS 2.0 成为现在最常用的 RSS 格式一点都不令人惊讶。

- 下面就是一个基本的 RSS 2.0 文档的例子：

- `<?xml version="1.0" encoding="UTF-8" ?>`

-

- `<rss version="2.0">`

- `<channel>`

- `<title>My Revenge</title>`

- `<description>Dedicated to having our revenge</description>`

- `<link>http://si thboys.com</link>`

- `<item>`

- `<title>At last!</title>`

- `<link>http://si thboys.com/atlast.htm</link>`

- `<author>DarthMaul@si thboys.com</author>`

- `<description>`

- `At last we will reveal ourselves to the Jedi . At last we will`

have

- `our revenge.`

- `</description>`

- `</item>`

- `</channel>`

- `</rss>`

• RSS 2.0 的普及使其成为大家争相支持的格式。在本章中的所有应用程序都只支持一种 RSS 格式，即 RSS 2.0。

• 5.2 Atom

• Atom 是一种刚登上聚合舞台的新技术。Atom 从出现开始，就受到了相当多的关注和应用。与 RSS 不同，Atom 是一个严格的规范。RSS 规范存在的一个问题是，在其元素中缺乏告诉开发人员如何处理 HTML 标记的信息。Atom 规范则解决了这个问题，并为开发人员指定了严格的、必须遵从的规则，还提供了许多新的功能，可以让开发人员选择

元素的内容类型,为某个特定元素指定处理方法的特殊属性。毫不奇怪,基于这些功能,Google 和 Movable Type 等使用 Atom 的网站都显示出强大的一面。

• 总体而言,Atom 还是有些类似 RSS 的,除了元素名不同之外,其文档结构也有一些区别:

- <?xml version="1.0" encoding="iso-8859-1"?>
-
- <feed version="0.3" xmlns="http://www.w3.org/2005/Atom" xml:lang="en">
- <title>My Revenge</title>
-
- <link rel="alternate" type="text/html" href="http://si thboys.com" />
- <modified>2005-06-30T15:51:21-06:00</modified>
- <tagline>Dedicated to having our revenge</tagline>
- <id>tag: si thboys.com</id>
- <copyright>Copyright (c) 2005</copyright>
- <entry>
- <title>At last!</title>
- <link rel="alternate" type="text/html" href="
- <http://si thboys.com/atlast.htm> />
- <modified>2005-06-30T15:51:21-06:00</modified>
- <issued>2005-06-30T15:51:21-06:00</issued>
- <id>tag: si thboys.com/atlast</id>
- <author>
- <name>Darth Maul</name>
- </author>
- <content type="text/html" xml:lang="en"
- xml:base="http://si thboys.com">
- At last we will reveal ourselves to the Jedi. At last we will
- have
- our revenge.

- `</content>`
- `</entry>`
- `</feed>`

• 5.3 FooReader.NET

• 基于 RSS 和 Atom 的聚合主要关注的是信息的共享。为了实现查看来自几个不同地方的信息的功能，应用程序将调用一个聚合器（aggregator）来将不同的提要收集到同一个地方。聚合器可以使得从 Web 上收集最新的消息变得更加容易和快捷（和每天访问几个 Web 网站相比要简单得多）。本章的剩余部分将聚焦于如何设计和开发这样的聚合器。

• FooReader.NET 是一个基于 Web 的、.NET 环境下的 RSS/Atom 聚合器，它是在 ForgetFoo 的基于 ColdFusion 的 FooReader (<http://reader.forgetfoo.com>) 的基础上移植的。包括最普遍的电子邮件应用程序在内的许多传统应用程序都能够填补聚合器的空白，那为什么还要构建一个基于 Web 的 RSS/Atom 聚合器呢？主要有以下几个原因：

- **q Web 是跨平台的。**构建一个基于 Web 的聚合器可以确保任何现代浏览器都能够访问它。
- **q Web 处于中心位置。**传统的聚合器中存在问题，由于聚合器是安装在计算机中的，因此会在多个地方维护数据。如果想在办公室和家里都能够阅读聚合提要，就必须在每台计算机上安装一个聚合器，并载入相应的提要。一个基于 Web 的聚合器则解决了这个问题，因为任何对提要列表做出的修改都与用户的位置无关。

• 下一小节将说明 FooReader.NET 是如何使用 Ajax 创建的。正如其他 Web 应用程序一样，这里也包括两个主要组件：客户端组件和服务器端组件。

• 5.3.1 客户端组件

• 正如前面所说的，Ajax 解决方案的客户端组件负责向用户显示数据，管理与服务器的通信。对于 FooReader.NET 而言，管理所有用户体验需要几个客户端组件。

- **q JavaScript 的 XParser 类，**当接收到数据时负责获取信息并解析它。
- **q 将用户与数据绑定的用户界面。**由于用户界面实际上就是一个网页，因此使用的是 Web 浏览器中常见的 HTML、CSS 和 JavaScript 技术。

- `q` 负责 XParser 接收数据，并在用户界面中显示出来的 JavaScript 代码。

• 尽管 JavaScript 没有正式的类定义，但拥有等价的程序逻辑。为了便于理解，本会将创建对象的函数作为类。

• 1. XParser

• FooReader.NET 的第一个组件是 XParser，它是一个 JavaScript 类，用来将 RSS 和 Atom 提要解析为 Web 应用程序中更易于使用的 JavaScript 对象。XParser 的主要目标是为开发人员提供一个接口，通过该接口可以更方便地访问大部分重要元素。它不仅可以减少代码行数（暂不考虑下载时间），而且还将减少客户端的多余工作。

• 以类为中心的 .NET 框架首先将其纳入设计，XParser 类由三个类组成：XParseItem、XParserElement 和 XParserAttribute。

• 从 XParser 中最简单的类开始，XParserAttribute 用来表示元素的属性。由于属性中只有一部分数据（属性值）是需要的，因此它也是 XParserAttribute 的唯一属性。

```
• function XParserAttribute(oNode) {
  •   this.value = oNode.nodeValue;
  • }
```

• XParserAttribute 类的构造函数有一个参数，即 DOM 的属性节点。根据这个节点，就可以通过它的 `nodeValue` 属性获得属性值，然后将其存到 `value` 属性中。很简单，是吗？

• XParserElement 类是一个 XML 元素，负责访问和获取元素的值和属性。该类的构造函数有两个参数：XML 元素的节点和该节点的值。

```
• function XParserElement(oNode, sValue) {
  •   this.node = oNode || false;
  •   this.value = sValue || (this.node && this.node.text) || false;
```

• 在最前面几行，类的四个属性中的两个，将按照参数的属性值来设置它的值。注意，在赋值语句中 OR (`|`) 操作符的用法。

• 在对 `node` 进行赋值的语句中，使用 OR 操作符的效果和一个三元操作符是相同的：
`this.node = (oNode)?oNode:false;`。而对于 `value` 进行赋值的语句中，则更类似于如下所示的 `if...else` 程序块：

- if (sValue) {
- this.value = sValue;
- } else if (this.node && this.node.text) {
- this.value = this.node.text;
- } else {
- this.value = false;
- }

• 而在简洁版本中，节省了 if...else 程序块中一半的字符，去除多余的字符总是好的。

• 在 Firefox 的 DOM 中，对于元素没有 text 属性。为了实现该功能，XParser 可以使用前面章中介绍的 zXml 库，它对 Firefox 的 DOM 进行了扩展，使其包含了 text 和 xml 属性。

• 接下来几行代码则用来生成 attributes 集合，一个 XParserAttribute 对象的集合。

- if (this.node) {
- this.attributes = [];
- var oAtts = this.node.attributes;
- for (var i = 0; i < oAtts.length; i++) {
- this.attributes[i] = new XParserAttribute(oAtts[i]);
- this.attributes[oAtts[i].nodeName] = new
- XParserAttribute(oAtts[i]);
- }
- } else {
- this.attributes = false;
- }
- this.isNull = (!this.node && !this.value && !this.attributes);

• 首先检查节点是否存在（为一个不存在的节点创建属性集合是没有意义的）。然后创建一个数组，再通过 DOM 的 attributes 属性来收集元素的属性。使用一个 for 循环，为每个属性节点创建一个 XParserAttribute 对象；一种方法是使用整型键，另一种方法

则是使用从 `nodeName` 属性中获取的属性名称作为键。当知道属性名称后就可以通过它来访问特定的属性。

- 如果元素的节点不存在，则 `attributes` 将设置为 `false`（就像不能够除以 0 一样，也不能够从一个不存在的节点上获取属性）。最后，将对 `isNull` 进行赋值。这个 `isNull` 属性可以用来检查返回的对象是否为 `null`。如果 `node`、`value` 和 `attributes` 属性的值均为 `false`，那么 `XParserElement` 对象将被看作 `null`。

- `XParserItem` 类用来表示一个 `<rss:item/>` 或 `<atom:entry/>` 元素（从现在开始，这些元素将用来表示一个简单的条目）。条目（`item`）由几个元素组成，因而显然需要一些解析处理。`XParserItem` 的构造函数需要一个名为 `itemNode` 的参数，用来表示条目的 DOM 节点。

- ```
function XParserItem(itemNode) {
 this.title=this.link=this.author=this.description=this.date =
 new XParserElement();
```

- 这个类中的第一行代码很重要。尽管 RSS 和 Atom 都是标准化的，但并不意味着所有的人都要遵循这个标准。RSS 提要可能会在一个或所有条目中略去 `<author/>` 元素，而 Atom 提要则可能忽略 `<content/>` 标签，而倾向于 `<summary/>` 标签。正是由于存在这些差异，因此为 `XParserItem` 属性设置一个默认值是很重要的，这可以避免出现错误。其默认值是一个空的（`null`）`XParserElement`。现在我们开始解析子元素。

- ```
for (var i = 0; i < itemNode.childNodes.length; i++) {  
    var oNode = itemNode.childNodes[i];  
    if (oNode.nodeType == 1) {  
        switch (oNode.tagName.toLowerCase()) {
```

- 先将通过一个 `for` 循环来遍历其子元素，然后检查节点的类型。如果 `nodeType` 为 1，则说明当前节点是一个元素。检查节点的类型似乎像一个意料之外的处理，但这是必需的。`Mozilla` 会将元素之间的空白也当作子节点；因此对 `nodeType` 进行检查可以避免当将该节点发送给 `XParserElement` 时出错。在证实了当前节点是一个元素之后，在 `switch...case` 程序块中将使用节点的标签名（`tag name`），并且条目的属性也将根据该标签名进行赋值；

- //共享的标签
- case "title":
 - this.title = new XElement(oNode);
- break;
- case "link":
 - if (oNode.getAttribute("href")) {
 - this.link = new

XParserElement(oNode, oNode.getAttribute("href"));

- } else {
 - this.link = new XElement(oNode);
- }
- break;
- case "author":
 - this.author = new XElement(oNode);
- break;

• 尽管在许多方面存在不同,但 RSS 和 Atom 还是有许多相同的元素。在这段代码中,相近的元素包括<title/>、<link/>和<author/>。在<link/>元素中只有一个区别:在 Atom 提要中认为其值存在于 href 属性中,而 RSS 的值就是元素的值。

- //RSS 标签
- case "description":
 - this.description = new XElement(oNode);
- break;
- case "pubdate":
 - this.date = new XElement(oNode);
- break;

• 在此之后的共享元素是 RSS 规范特有的元素,它用来匹配<description/>和<pubdate/>元素,并将其传给 XElement。

- //Atom 标签
- case "content":

- `this.description = new XParserElement(oNode);`

- `break;`

- `case "issued":`

- `this.date = new XParserElement(oNode);`

- `break;`

• 这段代码用来匹配 Atom 规范中特定的<content/>和<i ssued/>元素。最后检查的元素是一个扩展（extension）：

- `//扩展`

- `case "dc:date":`

- `this.date = new XParserElement(oNode);`

- `break;`

- `default:`

- `break;`

• 该段代码用来检查<dc:date/>元素，它是 Dublin 核心扩展（Dublin Core extension，<http://dublincore.org/documents/dcmi-terms>）的一部分。该扩展应用十分广泛，检查和使用它的值是一个好办法。

• 如果因为某种原因，提要中不包含上面这段 switch 程序块中指定的元素，那么该元素将被设置为默认值，即一个空的 XParserElement 对象（其值来自于 XParserItem 类的第一行）。由于 XParser 是一个 JavaScript 类，因此并非每一个元素都必须解析。但是，使用 switch 能够很容易添加其他元素（如果需要）。

• XParser 类是一个主要类，它封装了前面讨论的所有类。它的构造函数要求一个参数 sFileName，以及一个可选的 blsXml 参数。为了获得最大的灵活性，XParser 设计为要么自己发出一个 XMLHttpRequest 请求，要么将一个外部请求的 responseText 传给构造函数。如果 blsXml 的值是 false 或 undefined，那么 sFileName 将当作一个 URL 处理，同时 XParser 将自己发出请求；否则，sFileName 将当作一个 XML 字符串处理，并将其载入到一个 XML DOM 对象中。

- `var oThis = this;`

- this.title=this.link=this.description=this.generator=this.modified=

- this.author = new XElement();

- this.onload = null;

- 使一个变量的值包含对象的引用，似乎还不熟悉，不过当稍后涉及 XMLHttpRequest 对象的 onreadystatechange() 事件处理函数时，该技术就逐渐常用起来。第 2 行代码的用途和 XElement 类的第一行代码很接近。它设置了对象的属性，使得重要的元素有一个默认值。接下来的代码则声明了 onload 事件处理函数，当提要完全载入后将启动它。

- XElement 的 load() 方法将在收到 XML 数据时调用。当 XML 传给构造函数时，就会立即调用 load() 方法，并且将 fileName 中的 XML 数据传给该方法。

- if (isXml) {

- this.load(fileName);

- }

- 但时，当传给构造函数的是一个 URL（即 isXml 的值是 false 或 undefined）时，那么 XElement 将自己发出请求。

- else {

- var oReq = XMLHttpRequest.createRequest();

- oReq.onreadystatechange = function () {

- if (oReq.readyState == 4) {

- //仅当"OK"时

- if (oReq.status == 200) {

- this.load(oReq.responseText);

- }

- }

- };

- oReq.open("GET", fileName, true);

- oReq.send(null);

- }

- 注意，这段代码中的 `onreadystatechange()` 事件中，使用了本书前面已经使用过的 `zXmlHttp` 跨浏览器兼容的工厂方法，当请求成功时将检查请求的状态，并将 `responseText` 传给 `load()` 方法。这是由变量 `oThis` 来实现的。如果 `this.load()` 不是一个函数，那么当使用 `this.load(oReq.responseText)` 时将会抛出一个错误。由于关键字 `this` 是位于 `onreadystatechange` 事件处理函数之中，它是对事件处理函数的引用，而非对 `XMLParser` 对象的引用，因此将抛出这个特殊的错误。

- 最后，将通过 `send()` 方法发送该请求。

- 只有当要解析的 XML 数据准备好后，才会调用 `load()` 方法。它需要传入一个要解析的 XML 数据的 `sXml` 参数。

- `XMLParser.prototype.load = function (sXml) {`

- `var oXmlDom = zXmlDom.createDocument();`

- `oXmlDom.loadXML(sXml);`

- 在这段代码中，将把 XML 数据载入到一个 `XMLDOM` 对象中，这个对象是通过第 4 章中介绍的跨浏览器的 XML DOM 工厂创建的。现在该 DOM 已经准备好了，就可以开始解析了。

- `this.root = oXmlDom.documentElement;`

- 第一步是解析文件的简单属性。通过使用表示 XML 文档元素引用的 `root` 属性，可以确定将解析哪种类型的提要。

- `this.isRss = (this.root.tagName.toLowerCase() == "rss");`

- `if (this.isRss && parseInt(this.root.getAttribute("version")) < 2) {`

- `throw new Error("RSS version is less than 2");`

- `}`

- `this.isAtom = (this.root.tagName.toLowerCase() == "feed");`

- `this.type = (this.isRss)?"RSS":"Atom";`

- 为了确定提要的类型，需要检查文档的根元素。如果其标签名是 `rss`，则说明使用的是某个版本的 RSS。如果文档的根元素是 `feed`，则说明是一个 `Atom` 提要。根据提要的类型将布尔属性 `isRss` 和 `isAtom` 的值进行相应的设置。最后，根据提要的类型设置 `type` 属性。这个属性是用来向用户显示的。

- 如果提要的类型是 RSS, 那么检查提要的版本是很重要的。XParser 是基于 RSS 2. x 编写的解析器, 如果版本小于 2, 那么将抛出一个异常并且停止所有的解析工作。

- RSS 和 Atom 都有包含提要自身信息的元素, 但却位于文档的不同位置。在 RSS 中, 这些信息也包含在<channel />元素中, 该元素包含了提要中所有的其他元素。而 Atom 只使用根元素来封装这个信息。这些相似之处可以使提要的解析更简单:

- ```
var oChannel =
(this.isRss)?this.root.getElementsByTagName("channel")[0]:this.root;
```

- 如果提要的类型是 RSS, 那么 oChannel 变量的值则设置为<channel />元素; 否则设置为提要的文档元素。从这个相同的语句开始, 就将对 non-item 进行解析:

- ```
for (var i = 0; i < oChannel.childNodes.length; i++) {  
    var oNode = oChannel.childNodes[i];  
    if (oNode.nodeType == 1) {  
  
        用一个 for 循环来遍历 channel 的所有子节点。同样, 还是要再次检查当前节点的  
nodeType 属性。如果确认是一个元素, 则继续解析。与 XParseItem 相同, 也将使用一个  
switch 程序块。
```

- ```
switch (oNode.tagName.toLowerCase()) {
 //共享标签
 case "title":
 this.title = new XParserElement(oNode);
 break;
 case "link":
 if (this.isAtom) {
 if (oNode.getAttribute("href")) {
 this.link = new
XParserElement(oNode,oNode.getAttribute("href"));
 }
 } else {
 this.link = new XParserElement(oNode);
```

- }
- break;
- case "copyright":
- this.copyright = new XParserElement(oNode);
- break;
- case "generator":
- this.generator = new XParserElement(oNode);
- break;

• RSS 和 Atom 类似，在其规范的这部分中元素都很少，这使得整个过程更简单（同时代码量增加得也不多）。在这些元素中最主要的不同是<link/>元素（和 XParserItem 类似）。Atom 的提要包含一个<link/>元素，但要使用的是它的属性值。使用 `getAttribute()` 方法，获取 href 属性的值，该值同样也包含在 XParserElement 构造函数中。紧下来则是 RSS 特有的元素。

- //RSS 标签
- case "description":
- this.description = new XParserElement(oNode);
- break;
- case "lastbuilddate":
- this.modified = new XParserElement(oNode);
- break;
- case "managingeditor":
- this.author = new XParserElement(oNode);
- break;

• 而以下则是 Atom 特有的元素。

- //Atom 标签
- case "tagline":
- this.description = new XParserElement(oNode);
- break;
- case "modified":

- `this.modified = new XElement(oNode);`
- `break;`
- `case "author":`
- `this.author = new XElement(oNode);`
- `break;`
- `default:`
- `break;`

• 现在已经完成了对提要信息元素的解析，接下来将创建并设置 `items` 数组。顾名思义，`items` 数组就是 `XParserItem` 对象的集合。`<rss:item/>`和`<atom:entry>`元素将逐个发送给 `XParserItem` 构造函数中：

- `var oltems = null;`
- `if (this.isRss) {`
- `oltems = oChannel.getElementsByTagName("item");`
- `} else {`
- `try {`
- `oXmlDom.setProperty('SelectionLanguage', 'XPath');`
- `oXmlDom.setProperty("SelectionNamespaces",`
- `"xmlns:atom='http://www.w3.org/2005/Atom'");`
- `oltems = oXmlDom.selectNodes("/atom:feed/atom:entry");`
- `} catch (oError) {`
- `oltems = oChannel.getElementsByTagName("entry");`
- `}`
- `}`

• 从微软的 XML DOM 4.0 版本之后，`getElementsByTagName()` 方法就有了一些变化。在 3.0 及以下版本中，当使用 `getElementByTagName()` 时将忽略全名 (qualified name)，因此要获取任何元素，只需简单地将标签名传给该方法即可。

• 在 4.0 以后的版本中，要在一个默认的命名空间中选择一个元素，需要使用 `selectNode()` 方法，并传入一个 XPath 表达式。而用来为 XML DOM 解析器设置额外的属性的 `setProperty()` 方法，将用来设置命名空间以使 `selectNode()` 方法可以用来选择元素。

在前面例子的 try...catch 程序块中可以看到它的使用。新的 selectNodes() 方法是可靠的，如果它失败了则使用 getElementsByTagName() 来获取元素。

- SelectNodes() 方法是 MSXML 的一部分，仅限于在 IE 中使用，而基于 Mozilla 的浏览器则继续使用带默认命名空间的 getElementsByTagName() 来获取元素。

- 当获取了 <rss:item/> 和 <atom:entry> 元素之后，每个节点都将传给 XParserItem 类的构造函数，以创建 items 数组：

- for (var i = 0; i < oltems.length; i++) {
  - this.items[i] = new XParserItem(oltems[i]);
  - }

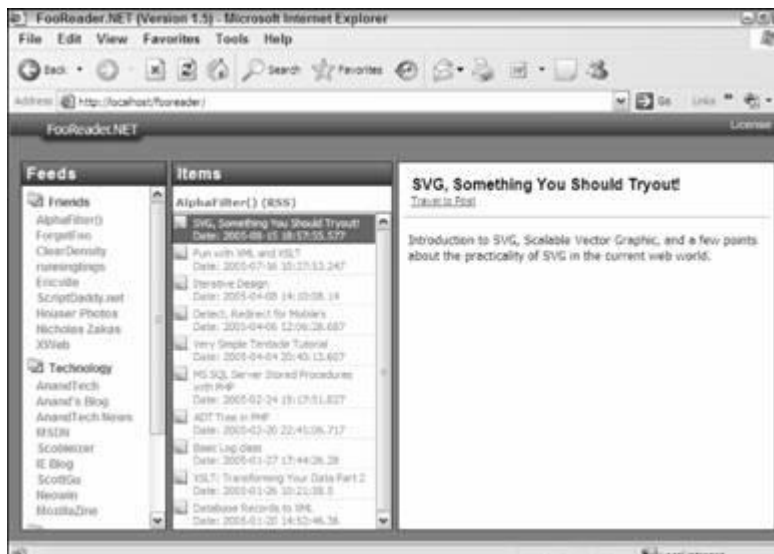
- 在这里，所有的重要元素都已经解析出来，并且包含其相应的属性。剩下的最后一件事就是启动 onload 事件：

- if (typeof this.onload == "function") {
  - this.onload();
  - }

- 当第一次声明 onload 时，曾赋值为 null。因此，检查 onload 函数的类型是很重要的。如果它是一个函数（意味着该事件现在已经有了解决函数），那么就将调用 onload() 方法。我们将会在本章的后一部分看到该事件的使用。

## • 2. 用户体验

- 对于任何应用程序而言，用户界面或许是最重要的。如果用户不能够使用应用程序，那么应用程序就没有存在的理由。FooReader.NET 的设计是使用户易于使用和理解。实际上，它的用户界面与微软的 Outlook 2003 十分相似。界面中有三个窗格，前两个的宽度是固定的，第三个则是可动的（参见图 5-1）。



• 图 5-1

• 第一个窗格称为提要窗格（feeds pane），以用户可以点击的链接的形式显示不同的提要。在提要窗格中有一个 id 为 di vFeedLi st 的<di v/>元素，用来将提要列表写到文档中。由于该窗格的内容是动态构成的，因此只有这个包含元素是静态编写的：

- <di v i d="di vFeedsPane">
- <di v cl ass="paneheader">Feeds</di v>
- <di v i d="di vFeedLi st"></di v>
- </di v>

• 当用户点击某个提要时，提要的项将在中间的条目窗格（i tems pane）中显示出来。这个窗格中包含两个用来显示信息的元素。第一个<di v/>元素的 i d 是 di vI ewi ngI tem。该元素用来向用户显示两个内容：当前阅读的是哪个提要，该提要的文档类型是什么（RSS 和 Atom）。第二个元素则是另一个<di v/>元素，其 i d 为 di vI temLi st。该元素包含一组 RSS 的<i tem/>元素或 Atom 的<entry/>元素。与提要窗格相同，在该页面中静态写入的只有这些包含元素：

- <di v i d="di vI temPane">
- <di v cl ass="paneheader">I tems</di v>
- <di v i d="di vI ewi ngI tem"></di v>
- <di v i d="di vI temLi st"></di v>
- </di v>



• 如果用户单击某个条目时，将把该条目的内容载入到最后的阅读窗格（reading pane）上。该窗格中包含三个显示条目信息的元素。第一个元素的id是divMessageTitle，用来显示RSS或Atom提要的<title>元素。第二个元素的id是aMessageLink，它的href属性是动态变化的。最后一个元素是divMessageBody，用来显示<rss:description/>和<atom:content/>元素的内容：

- <div id="divReadingPane">
- <div class="contentcontainer">
- <div class="messageheader">
- <div id="divMessageTitle"></div>
- <a href="" id="aMessageLink" title="Click to goto posting." target="\_new">Travel to Post</a>
- </div>
- <div id="divMessageBody"></div>
- </div>
- </div>

### • 3. 可用性

• 还有一些可用性问题必须考虑。第一，用户希望 Web 应用程序的功能能够与其他应用程序相似。与该应用程序界面很相似的 Outlook 2003 中，双击条目窗格中的某个条目将会在一个新窗口中弹出特定的电子邮件消息。由于 FooReader.NET 是基于这个模型的，因此双击一个条目也应该弹出一个新窗口，显示出特定的 blog 或文章信息。要实现该功能，应把条目的 onclick 事件赋给 doubleClick() 事件处理函数（稍候你将看到赋值方法）：

- function doubleClick() {
- var oltem = oFeed.items[this.getAttribute("frFeedItem")];
- var oWindow = window.open(oltem.link.value);
- }

- 其次，用户必须知道应用程序什么时候发送请求。只有当提要载入时才将向服务器发送请求，因此需要通过一些可视化的方法来提示用户所发生的事情。为了达到该目的，当提要发送请求时将显示一个“载入提示”，而当提要载入完成后就将其隐藏。图 5-2 展示了运行中的这种用户界面提示。

- 这个“载入提示”将通过一个名为 toggleLoadingDiv() 的 JavaScript 函数来控制。该函数有一个布尔型参数，用来确定是否显示该提示：

- function toggleLoadingDiv(bShow) {
- var oToggleDiv = document.getElementById("LoadingDiv");
- oToggleDiv.style.display = (bShow)?"block":"none";
- }



• 图 5-2

### • 5.3.2 服务器端组件

- 在理想情况下，诸如 FooReader.NET 的简单应用程序，严格意义上是客户端应用。JavaScript 能够通过 XMLHttpRequest 来访问域以获取 XML 提要，也就不需要对服务器端组件发出任何调用。但由于 IE 和 Firefox 的安全约束，它是无法在不同的域中获取数据的，因此就必须有服务器端组件的支持。

#### • 1. 可能的范型

- 在 FooReader.NET 应用程序中，服务器端的任务是根据客户端的需求获取远程主机的 XML 提要。遵照这个模型，服务器端有两种可行的设计途径，每种都有其赞成者，也有其反对者。

- 第一种模式是缓存提要的架构。服务器端程序将担当一个服务，在一个确定的时间周期内获取一个提要列表，然后存入缓存，当客户端请求时将缓存中的提要提供给客户端。这种选择显然能够节省带宽，但也存在使读者无法获得最新提要的风险。显示当前、最新的提要需要更多的用户操作，这违背了 Ajax 的思想。

- 第二种方法是按需传送的架构。当用户请求时，服务器再获取其指定的提要。这种方法将占用更多的带宽，但可以确保读者收到的是最新的信息。此外，这种设计符合 Ajax 概念并且是用户所预期的。

- 2. 实现

- FooReader.NET 使用的是按需传送模型，不同的是当其获取提要时仍然对其进行缓存处理。这个缓存的版本只在因无法连接远程主机而无法获取最新提要时使用。这可以确保用户至少能够看到某些信息，虽然是稍旧一些的数据。

- 由于服务器的职责只是获取远程主机的提要，并对其进行缓存处理，因此通过一个 ASP.NET 的页面就可以处理这些操作。这个页面命名为 xml.aspx，并且有个 code-behind（后台代码）文件，其中包含大量的 ASP.NET 代码。

- Code-behind 是在 ASP.NET 平台开发 Web 页面的方法。与内嵌编程模型不同，它不是将服务器端代码和 HTML 标记混合在一起（诸如 PHP 和 ASP），code-behind 可以将所有的程序逻辑从 HTML 代码分离出来，存放在单独的类文件中。其结果是使 HTML 代码和选择的 .NET 编程语言实现清晰的分离。

- 服务器端的入口是 Page\_Load 事件处理函数，在该事件处理函数中将调用名为 StartFooReader() 的方法。而代码则包含在 StartFooReader() 中。

- 该项目选择的语言是 C#，它是专为 .NET 框架开发的新语言。

- 1 设置首部

- 对于这个应用程序而言，还必须设置几个首部（header）。在 ASP.NET 中设置首部是一件简单的事：

- `Response.ContentType = "text/xml";`
- `Response.CacheControl = "No-cache";`

• 可以通过 `Response` 对象来设置首部，该对象封装了 HTTP 的响应信息。对于应用程序的操作而言，是必须设置 MIME 内容类型的。基于 Mozilla 的浏览器在默认情况下是不会以 XML 格式载入 XML 文件的，除非将 MIME 设置为 XML 文档，也就是将内容类型设置为 "text/xml"。

• 确保对 `XMLHttpRequest` 获取的 XML 数据不做缓存处理也是很重要的。IE 默认会对 `XMLHttpRequest` 获取的所有数据进行缓存处理，除非显式地设置首部 `CacheControl`。如果没有设置该首部，IE 在浏览器的缓存区满之前仍然会对其进行缓存处理。

#### • 1 获取远程主机的数据

• 为了确定要显示的提要，`FooReader.NET` 使用了一个名为 `feeds.xml` 的专有 XML 文档，它包含针对用户请求的所有可用提要列表。该文件包含一组 `<link/>` 元素，这组元素通过 `<section/>` 元素分成若干个部分。每个 `<link/>` 元素均有一个和 HTML 中的 `id` 属性类似的 `filename` 属性，而且它必须是唯一的，是 `<link/>` 元素的标识符：

- `<?xml version="1.0" encoding="utf-8"?>`
- `<feeds>`
- `<section name="News">`
- `<link name="Yahoo! Top Stories" filename="yahoo_topstories"`
- `href="http://rss.news.yahoo.com/rss/topstories" />`
- `</section>`
- `</feeds>`

• 该例子展现了一个基本的提要列表。一个典型的列表可以根据需要包含任意个 `<section/>` 和 `<link/>` 元素。注意，`<section/>` 元素只能够作为根元素的子节点，而 `<link/>` 元素则只能够包含于 `<section/>` 元素中。

• `<section/>` 和 `<link/>` 元素的 `name` 属性只是在提要窗格中显示给用户，在其他操作中都不使用。

- 当请求一个提要时，`filename` 属性的值将赋给查询字符串中的变量 `xml`。
- `xml.aspx?xml=filename`

- 在 ASP.NET 中，Request 对象包含一个名为 QueryString 的 NameValueCollection（名—值集合）。使用该集合，可以提取查询字符串中 xml 变量的值：

- if (Request.QueryString["xml"] != null)
- {
- string xml = Request.QueryString["xml"];
- FeedsFile feedsFile = new FeedsFile(Server.MapPath("feeds.xml"));
- 在第一行代码中，将检查查询字符串中是否包含 xml 变量。如果存在，就将其值

赋给 xml 变量，然后实例化一个 FeedsFile 对象。

- FeedsFile 类包含一个名为 GetLinkByFileName 的方法，该方法将返回一个 FeedsFileLink 对象，并包含一个特定的<link/>元素的信息。字符串变量 fileName 则用来保存稍后还将使用的提要路径信息。

- FeedsFileLink link = feedsFile.GetLinkByFileName(xml);
- string fileName = string.Format(
- @"{0}\xml\{1}.xml", Server.MapPath(String.Empty), link.FileName);

- 顾名思义，String.Format() 就是用来对字符串进行格式化的。传给该方法的第一个参数是表示格式的字符串。该字符串通常包含称为格式项（类似于 {0}、{1} 和 {2} 等）的字符。这些格式项将根据传入的相应参数来替换成具体的内容。在上面例子中，{0} 将替换为 Server.MapPath(String.Empty) 返回的字符串。

- 而在字符串之前的 @ 操作符则是用来告诉编译器不要处理转义字符串。在上面的例子中，如果将字符串写为 "{0}\\xml\\{1}.xml"，也将得到相同的结果。

- 在 .NET 框架中提供了一个 HttpRequest 类，它包含于 System.Net 命名空间中，用来向远程 Web 主机请求数据。.NET 框架中实现该任务的还包括其他一些类（诸如一般化的 WebRequest 类），不过 HttpRequest 类提供的是特定于 HTTP 的功能，因此对于这个应用程序而言是最适用的。要创建一个 HttpRequest 对象，需要调用 WebRequest 类的 Create() 方法，并需将其强制转型为 HttpRequest：

- HttpRequest getFeed = (HttpRequest) WebRequest.Create(link.Url);
- getFeed.UserAgent = "FooReader.NET (<http://reader.wdonline.com>)";

- `string feedXml = String.Empty;`

- `HttpRequest` 中有一个特定于 HTTP 的成员，它提供了 `UserAgent` 属性。虽然并非一定要使用这个属性，但它能够精确地说明它访问的远程服务器。许多传统的聚合器都有自己的用户代理 (user agent) 字符串，`FooReader.NET` 也如此。然后创建了 `feedXml` 字符串，现在只是一个空串。该变量稍后将用来存放远程提要的内容。下一步则是发送请求，并获取远程主机的响应：

- `//获取响应`

- `using (HttpWebResponse responseFeed = (HttpWebResponse)`

`getFeed.GetResponse())`

- {

- `//创建流阅读器 (Stream reader)`

- `using (StreamReader reader = new`

- `StreamReader(responseFeed.GetResponseStream()))`

- {

- `//读取内容`

- `feedXml = reader.ReadToEnd();`

- }

- }

- 当调用 `HttpRequest` 类的 `GetResponse()` 方法时，将创建一个 `HttpWebResponse` 对象。然后通过 `HttpWebResponse` 类的成员方法 `GetResponseStream()` 来创建一个 `StreamReader` 对象，用该对象来读取这个服务器响应。该流将把服务器的应用“读”到前面声明的 `feedXml` 变量中。由于在创建 `HttpWebResponse` 和 `StreamReader` 对象时，使用了 `using` 语句，因此在用完后会自己处理而不需要手动结束。

- I 缓存提要

- 尽管并不一定需要对提要进行缓存处理，但是万一无法连接远程服务器，这样做还是有好处的。`FooReader.NET` 是一个新闻阅读器，因此总是希望用户能够阅读到一些内容，即使版本稍有些旧。但只有一种情况下，才使用缓存的版本向用户提供服务：当到远程主机的因特网连接或路由失效而无法连接远程主机时。

- StreamWriter 就可以很好地完成这个任务，默认情况下它使用的编码格式是 UTF-8:

```
• using (StreamWriter strmWriter = new StreamWriter(fileName))

• {

• strmWriter.Write(feedXml);

• }
```

• 在前面创建的 fileName 变量现在将传给 StreamWriter 的构造函数，这将创建一个文件或者覆盖一个现有的同名文件。然后使用 Write() 方法将 feedXml 中包含的 XML 数据写到该文件中。

• 这样提要就存入缓存了。现在还需使用 Response.Write() 方法将 XML 数据写到页面中。

```
• Response.Write(feedXml);
```

- I 错误处理

• 有两个地方可能发生主要错误。第一个地方是在查询字符串中没有找到 xml 变量。没有这个变量，应用程序就无法知道在 feeds.xml 中要定位哪个提要，于是使用 GetLocalFile() 方法（稍后讨论）载入错误文档并将其写到页面中，该错误文档是一个静态 RSS 文档。

```
• if (Request.QueryString["xml"] != null)

• {

• //在此编码

• }

• else

• {

• Response.Write(GetLocalFile(Server.MapPath("error.xml")));

• }
```

• 由于这个错误文档是一个 RSS 文档，阅读器将像显示提要一样显示该错误消息。

• 第二个出错的地方是，当无法连接远程主机时会抛出一个错误。为了捕获该错误，必须将请求和响应的相关代码放在 try...catch 程序块中。

- try
- {
- //此处为处理 Web 请求的代码
- }
- catch (WebException webEx)
- {
- string fileToUse =

File.Exists(fileName)?fileName:Server.MapPath("error.xml");

- Response.Write(GetLocalFile(fileToUse));
- }

- 只捕获 WebException 将使得其他错误不能处理。ASP.NET 错误报告是很丰富的。

如果抛出一个 UnauthorizedAccessException（由于要进行缓存操作，因此可能在 try...catch 程序块中发生），则说明权限设置不正确，其错误信息能够有效地帮助你找到一个解决方案。

• 接下来看一下使用缓存文件的地方。当捕获到错误时，应用程序需要知道要把什么文件写到页面中：是缓存的提要还是错误文档。它首先使用 File.Exists() 来检查是否有一个缓存版本。如果缓存文件存在，则通过 GetLocalFile() 来获取结果文件并写到页面上。GetLocalFile() 方法是一个简单的文件读取函数，前面已经用过几次了。它首先将打开指定的文件，然后读取内容，将其内容作为一个字符串返回：

- public string GetLocalFile(string path)
- {
- string contents = string.Empty;
- using (StreamReader file = File.OpenText(path))
- {
- contents = file.ReadToEnd();
- }
- return contents;
- }

- 由于这个应用程序有大量的文件读取的操作，因此它的确是一个很实用的函数。



- I 解析提要文件

- `FeedFile` 类打开一个提要文件，并将其转为一个 `Xml Document` 类，使用该类就可以通过 DOM 来获取任何一个元素了。

- `private Xml Document _doc;`
- `public FeedFile(string path)`
- {
- `_doc = new Xml Document();`
- `_doc.Load(path);`
- }

- `FeedFile` 类公开一个 `GetLinkByFileName()` 方法。该方法只有一个 `fileName` 参数，且返回一个 `FeedFileLink` 对象。

- `public FeedFileLink GetLinkByFileName(string fileName)`
- {
- `string pathToNode =`
- `String.Format("/feeds/section/link[@filename='{0}']", fileName);`
- `XmlNode linkNode = _doc.SelectSingleNode(pathToNode);`
- `return new FeedFileLink(linkNode);`
- }

- `Xml Document` 类的 `SelectSingleNode()` 方法将根据传入的 XPath 表达式来选择一个特定的 `XmlNode`。在这个例子中，XPath 表达式是使用 `String.Format()` 来格式化的，它通过指定 `filename` 属性来选择一个特定的 `<link/>` 元素。然后将选择出来的 `XmlNode` 传给 `FeedFileLink` 类的构造函数，并返回该对象引用。

- `FeedFileLink` 将获取一个 `XmlNode`，然后获取该节点所必须的信息，并将其值赋给其私有成员。

- `private string _name;`
- `private string _filename;`
- `private Uri _uri;`
-

- public FeedsFileLink(XmlNode myNode)
- {
- \_name = myNode.Attributes["name"].Value;
- \_filename = myNode.Attributes["filename"].Value;
- \_uri = new Uri(myNode.Attributes["href"].Value);
- }

• 其中\_name 保存的是节点的 name 属性值，\_filename 则是 filename 属性的值，并且将创建一个 System.Uri 对象并将其赋给\_uri。

• 是的，可以用 string 来代替 Uri 对象。在 .NET 中，大多数包含 Uri 对象的类和方法同时也重载了一个接受 string 型的类或方法，但 Uri 对象更好一些。

• 注意前面提到的成员都是私有的。如果没有访问方法（accessor），则在类之外是无法访问私有成员的。在 C# 中访问方法能够对私有成员进行读、写或读写操作：

- public string Name
- {
- get
- {
- return \_name;
- }
- }
- 
- public string FileName
- {
- get
- {
- return \_filename;
- }
- }
- public Uri Uri
- {

- get
- {
- return \_uri;
- }
- }

• 在 `FeedsFileLink` 中的访问方法只提供了对私有成员的只读访问。这些特定的访问方法称为获取方法（getter），它将返回私有成员的值。这可以确保类中的数据不会被错误地修改或破坏。

### • 5.3.3 连接客户端和服务端

• 粘合 `FooReader.NET` 的是 JavaScript 代码，通过 `XParser` 来获取数据，通过 `FeedsFile` 类将其发送到用户界面并显示出来。这些代码分别包含于 `feedsfileparser.js` 和 `fooreader.js`。

#### • 1. 解析提要文件——客户端样式

• 在 `feedsfileparser.js` 中的 `FeedsFile` 类，与其服务器端的版本十分接近，它负责将 `feeds.xml` 解析成 JavaScript 对象。

• `FeedsFileElement` 类用来表示 `feeds.xml` 中的一个元素。由于在文档中仅使用两种不同的元素，因此该类很简单。它只有一个参数：元素的节点。

- function `FeedsFileElement(oNode)` {
- if (`oNode.tagName.toLowerCase() == "section"`) {
- `this.links = []`;
- var `linkNodes = oNode.getElementsByTagName("link")`;
- for (var `i = 0`; `i < linkNodes.length`; `i++`) {
- `this.links[i] = new FeedsFileElement(linkNodes[i])`;
- }
- }

• 如果 `aNode` 是一个 `<section/>` 元素，那么将创建一个包含 `<link/>` 元素的数组。而 `links` 数组的内容是通过将 `<link/>` 元素传给 `FeedsFileElement` 的构造函数来生成的。当

aNode 是一个<link/>元素，则使用 getAttribute()方法将<link/>元素的 filename 属性值赋给该类的 fileName 属性：

- else {
- this.fileName = oNode.getAttribute("filename");
- }
- this.name = oNode.getAttribute("name");

• 由于<section/>和<link/>元素都有 name 属性，在所有程序逻辑完成后将把 name 属性的值赋予该类的 name 属性。

• 提要文件是通过第 4 章中介绍的跨浏览器兼容的 XML 库载入的。用 XML DOM 来代替 XMLHttpRequest 有许多原因。首先，feed.xml 是一个静态的 XML 文件。使用 IE 中的 XMLHttpRequest 需要更多后台代码来设置不缓存的首部；否则 IE 将一直载入 feeds.xml 的缓存版本，即使该文件已经发生了变化。

- function FeedsFile() {
- var oThis = this;
- this.sections = [];
- this.onload = null;
- 
- var oXmlDom = zXmlDom.createDocument();
- oXmlDom.load("feeds.xml");

• 尽管提要文件是通过 XML DOM 来载入的，但你仍然可以使用 XML DOM 对象的异步能力（默认行为）。如果同步地载入文件，那么应用程序会等到整个文档完全载入后才会继续，这些使得用户会感觉应用程序冻结了。监控载入状态仍然使用

onreadystatechange 事件：

- oXmlDom.onreadystatechange = function () {
- if (oXmlDom.readyState == 4) {
- var oSections =

oXmlDom.documentElement.getElementsByTagName("section");

-

- for (var i = 0; i < oSections.length; i++) {
- oThis.sections[i] = new FeedsFileElement(oSections[i]);
- }
- }
- 
- oXmlDom = null;
- 
- if (typeof oThis.onload == "function") {
- oThis.onload();
- }
- };

• 我们将通过 DOM 中的 `getElementsByName()` 方法来获取 `<section/>` 元素，然后通过 `FeedsFileElement` 对象来构成一个 `sections` 数组。当所有的 DOM 操作都完成后，通过把 `oXmlDom` 设置为 `null`，就可以从内存中释放文档，并且启动 `onload` 事件。

• `FeedsFile` 类中有一个名为 `getLinkByFileName()` 的方法（应该还记得 C# 版本）。该方法在 `links` 数组中搜索每个 `section`，然后返回一个与文件名匹配的链接（`link`）对象。再强调一下，文件名是唯一的（也必须唯一），它是链接的标识符：

- this.getLinkByFileName = function (sFileName) {
- for (var i = 0; i < this.sections.length; i++) {
- var section = this.sections[i];
- for (var j = 0; j < section.links.length; j++) {
- var link = section.links[j];
- if (sFileName.toLowerCase() == link.fileName.toLowerCase())
- {
- return link;
- }
- }
- }
- }
- alert("Cannot find the specified feed information.");

- `return this.sections[0].link[0];`
- `};`

• 如果没有找到匹配的，则通过 `alert()` 方法显示一个错误消息，并返回提要列表中的第一个 `<link/>` 元素。

## • 2. 绘制用户界面元素

• 在 `fooreader.js` 中的代码负责生成向用户显示数据所需的元素。提要窗格中的提要列表，以及条目窗格中的条目列表都是基于接收到的数据动态生成的。这些动态内容都是通过 DOM 方法和 `innerHTML` 生成的。在页面及新提要载入之前，就需要预先通过 DOM 将所需的元素创建好。

• 消息窗格中的内容则通过 `innerHTML` 显示。这是必需的，因为许多提要包含 CDATA 小节，它包含了 HTML 格式的内容；使用 DOM 方法来显示这些文本可能生成一些混乱的结果（例如，希望加粗的文本看起来就成了 `<b>text goes here</b>`）。

### • 1 `init()`

• 客户端组件的入口是 `init()` 函数，它负责填充提要窗格和载入第一个提要。对于提要窗格而言，清晰、易于使用的布局最关键，而最后的无序列表是用来格式化数据的。

- `var divFeedList = document.getElementById("divFeedList");`
- `var oFragment = document.createDocumentFragment();`
- 
- `for (var i = 0; i < oFeedsFile.sections.length; i++) {`
- `var oSection = oFeedsFile.sections[i];`

• 提要列表的目标元素是通过 `getElementById()` 获取的，并存在 `divFeedList` 中。创建提要列表并不断更新页面，会降低用户界面的性能（从而使用户感到苦恼）。为了纠正这个问题，提要列表一开始只添加到一个文档片段中。当所有的操作都完成后，再把这个片段添加到文档上（包括该片段所有的子节点）。

• 正如前面所说的，在用户界面中定义一个小节是很容易也很直观的。先用一个图标，然后放上小节的名称就是一种很好的方法：

- `var olcon = document.createElement("img");`
- `olcon.src = "img/category_olcon.gif";`

- `olcon.border = "0";`
- `olcon.alt = "";`
- 
- `var oSpan = document.createElement("span");`
- `oSpan.appendChild(document.createTextNode(section.name));`
- `oSpan.className = "navheading";`

• DOM 的 `createElement()` 方法可以用来创建图标、图像以及小节的名称，然后将其封装到一个 `<span/>` 元素中。当小节的标题创建完成后，就可以添加到文档片段中；并且还将创建一个无序的列表，它将包含小节的提要：

- `oFragment.appendChild(olcon);`
- `oFragment.appendChild(oSpan);`
- `var oUl = document.createElement("UL");`
- `oUl.className = "navlist";`
- 每个提要将显示为一个链接，因此将创建 `<a/>` 和 `<li/>` 元素：
- `for (var j = 0; j < section.links.length; j++) {`
- `var oLink = oSection.links[j];`
- `var oLi = document.createElement("li");`
- 
- `var oA = document.createElement("a");`
- `oA.appendChild(document.createTextNode(oLink.name));`
- `oA.href = "#";`
- `oA.onclick = loadFeed;`
- `oA.className = "navlinks";`
- `oA.title="Load " + oLink.name;`
- `oA.setAttribute("fileName", link.fileName);`
- 
- `oLi.appendChild(oA);`
- `oUl.appendChild(oLi);`
- }

• 元素有一个名为 `fileName` 的属性，它包含与提要相关的文件名。当所有的链接和 `section` 都创建后，这个无序的列表将添加到文档片段中：

- `oFragment.appendChild(oUI);`
- 然后这个文档片段将添加到 `divFeedList` 中，而第一个提要将载入到用户界面中。
- `divFeedList.appendChild(oFragment);`
- `loadFeed(feedsFile.sections[0].links[0].fileName);`

• 到此为止，整个应用程序也就全部载入完毕了。提要列表全部生成了，但条目窗格还是空的。控制权现在转给了 `loadFeed()`。

#### • `loadFeed()`

• `loadFeed()` 函数负责载入一个 `XParser` 对象中的特定提要，然后将其填充到条目窗格中。该函数也与提要窗格中链接的 `onclick` 事件处理函数作用相同。它需要传入一个名为 `sFileName` 的参数，该参数与 `feed.xml` 文件中的 `<link/>` 元素的 `filename` 属性相对应。如果由于触发了一个事件而启动了 `loadFeed()`，那么 `sFileName` 的值将改为前面提及的 `fileName` 属性的值：

- `function loadFeed(sFileName) {`
- `sFileName = (typeof sFileName == "string")?sFileName:`
- `this.getAttribute("fileName");`
- 首先必须找到位于用户界面中的 `divItemList` 和 `divViewingItem` 两个元素，它们是分别用来显示条目列表和提要信息的：

- `var divItemList = document.getElementById("divItemList");`
- `var divViewingItem = document.getElementById("divViewingItem");`

• 其中条目列表是封装在 `divItemList` 中，而 `divViewingItem` 则包含了提要的名称和类型。

- `var sName = feedsFile.getLinkByFileName(sFileName).name;`
- `toggleLoadingDiv(true);`

• 提要的名称是通过 `feedsFile` 对象的 `getListByFileName()` 方法获得的。然后将调用 `toggleLoadingDiv()` 函数来显示进度提示，以告知用户应用程序正在处理请求。



- 条目列表中仅有由 DOM 方法创建的<a/>元素。但首先必须将已有的条目删除掉，换成新的列表。这可以通过 while 循环来实现它：

- while (divItemLi st.hasChildNodes()) {
- divItemLi st.removeChild(divItemLi st.lastChild);
- }

- 由于 divItemLi st 唯一的子节点就是条目，因此可以放心地删除所有的子节点。当所有旧的条目都删除后，就可以创建 XParser 对象了：

- sFileName = "xml.aspx?xml=" + sFileName;
- oFeed = new XParser(sFileName);
- sFileName 变量的值将修改为服务器端应用程序的 URL，并将其作为参数传给

XParser 的构造函数。

- 执行一个 XMLHttpRequest 请求需要花一些时间，如果没有考虑这个时间就可能抛出错误。为了解决这个问题，XParser 公开了一个名为 onload 的事件，当 XMLHttpRequest 请求返回一个 OK 状态（状态码为 200）时将启动它。在 FooReader.NET 中，XParser 的 onload 事件用来将条目写到 divItemLi st 中，并且在阅读窗格中载入第一个条目，并隐藏用户界面上的进度提示。

- oFeed.onload = function () {
- var oFragment = document.createDocumentFragment();
- divViewingI tem.innerHTML = sTitle + " (" + oFeed.type + ")";

- 与 init()函数一样，在此也将创建一个文档片段，以减少用户界面的载入时间。正如前面提到的，divViewI tem将通过 innerHTML 属性来显示提要的名称和类型。这种类型的用户反馈是创建一个良好的 Ajax 应用程序的关键。尽管这些信息对于应用程序的功能而言并不是必要的，但它使得用户能够了解自己在做什么。

- 为了简单起见，使用了 innerHTML 属性。在这个场景中，与 DOM 方法相比，可以通过 innerHTML 以更少的字符和更小的空间来修改元素的内容。innerText 属性很适用，但是 Firefox 不支持该属性。

• 现在应该创建条目了。条目是一个块网络的元素，其中包含两个元素：一个用来存放条目的标题，而另一个则用来存放条目的日期。与提要窗格中的列表类似，表示条目的元素也是使用 DOM 的 createElement() 方法创建的。

- for (var i = 0; i < oFeed.items.length; i++) {
- var oItem = oFeed.items[i];
- var oA = document.createElement("a");
- oA.className = "itemLink";
- oA.href = "#";
- oA.onclick = itemClick;
- oA.ondblclick = doubleClick;
- oA.id = "oA" + i;
- oA.setAttribute("frFeedItem", i);

• 第一个创建的是元素，其属性都是标准的：class 属性设置为 CSS 类 itemLink；href 属性设置为“#”；onclick 属性设置为 itemClick，当用户双击当该元素时，将启动 ondblclick 事件，并交由 doubleClick() 函数处理；id 属性则设置为字符串 oAx 的值，其中 x 是条目的编号。

- 关于 itemClick() 函数的介绍参见下一小节。

- 然后将创建第一个元素。这个元素将包含条目的标题：

- var divTitle = document.createElement("div");
- divTitle.className = "itemheadline";
- divTitle.innerHTML = oItem.title.value;

• 你会发现再次使用 innerHTML 代替了 createTextNode()。这是因为在标题（可以是 HTML 实体或元素）中有些使用了 HTML 格式。以下则创建另一个元素来存放日期：

- var divDate = document.createElement("div");
- divDate.className = "itemdate";
- divDate.appendChild(document.createTextNode("Date: " + oItem.date.value));

- 为存放日期而创建的元素与为存放标题而创建的元素相类似，唯一的区别是它使用 `createTextNode()` 函数来向元素中添加文本信息（可以这么做是因为日期中不会包含任何 HTML 格式）。

- 两个 `<div>` 元素将依次添加到 `<a>` 元素中，然后再将 `<a>` 元素添加到文档片段中：

- `oA.appendChild(divTitle);`
- `oA.appendChild(divDate);`
- `oFragment.appendChild(oA);`

- 当退出循环并创建了所有的条目之后，文档片段（准确地说，就是条目列表）也将添加到文档中去：

- `divItemList.appendChild(oFragment);`

- 然后将调用 `itemClick()` 函数来将第一个条目载入到阅读窗格中，完成后将隐藏用户界面上的进度提示：

- `itemClick(0);`
- `toggleLoadingDiv();`

- 现在已经完成了填充条目窗格的工作，唯一剩下的是将第一个条目载入到阅读窗格中，这通过调用 `itemClick()` 函数实现。

- `itemClick()`

- 当用户点击条目窗格的链接时，将调用 `itemClick` 函数（与 `loadFeed()` 类似，`itemClick()` 也是一个事件处理函数），它需要一个名为 `iItem` 的参数，该参数是一个表示要显示的条目数量的整数。该函数还通过修改背景色来使用户最后点击的链接与其他链接区别开来。通过将元素的 CSS 类修改为 `itemlink-selected` 来实现背景色的修改。基于 CSS 的不同样式可以很容易地针对应用程序进行定制。

- 为了使用户界面正常工作，可以定义一个跟踪已选择条目的全局变量。该变量名为 `oSelected`，其值为对选中元素的引用：

- `iItem = (typeof iItem == "number") ? iItem : this.getAttribute("frFeedItem");`

- `var oEl = document.getElementById("oA"+iItem);`

- `if (oSelected != oEl) {`
- `if (oSelected) oSelected.className = "itemlink";`
- `oEl.className += "-selected";`
- `oSelected = oEl;`

• 它将通过 `getElementById()` 函数来获取对所点击的链接的对象引用，并将其赋给变量 `oEl`。然后比较变量 `oEl` 和 `oSelected`，来判断这是一个新链接还是一个点击过的链接。如果这两个对象是相同的引用，则什么都不处理，因为点击的链接是当前选中的链接。如果其引用是不匹配的，则 `oSelected` 的 CSS 类将重新设置为 `itemlink`，而新链接的 CSS 类则设置为 `itemlink-selected`，并将 `oSelected` 设置为新链接的引用。

- 为了在阅读窗格中显示条目，还必须获取几个元素的引用。
- `var divTitle = document.getElementById("divMessageTitle");`
- `var aLink = document.getElementById("aMessageLink");`
- `var divBody = document.getElementById("divMessageBody");`

• 顾名思义，`divTitle` 元素就是用来显示条目标题的。`aLink` 元素用来提供一个链接，在用户点击后将载入实际的新闻条目。`divBody` 元素用来显示 `<rss: description/>` 或 `<atom: content/>` 元素的信息：

- `var oltem = oFeed.items[iItem];`
- `divTitle.innerHTML = oltem.title.value;`
- `aLink.href = oltem.link.value;`
- `divBody.innerHTML = oltem.description.value;`

• 然后将条目对象赋值给 `oltem`，并用该对象从 `oFeed` 中获取数据。条目的标题显示到 `divTitle` 中，链接的 `href` 属性也将根据新值进行修改。通过 `innerHTML` 在 `divBody` 中显示条目的描述信息。使用 `divBody` 的 `innerHTML` 可以确保以可读性较高的形式显示给用户。诸如 `&amp;` 的 HTML 实体将会正确地显示为 `"&"`。

## • 5.4 安装

• 通常，Web 应用程序的设置和使用是很简单的，但是.NET 的 Web 应用程序在安装时需要一个额外的步骤。

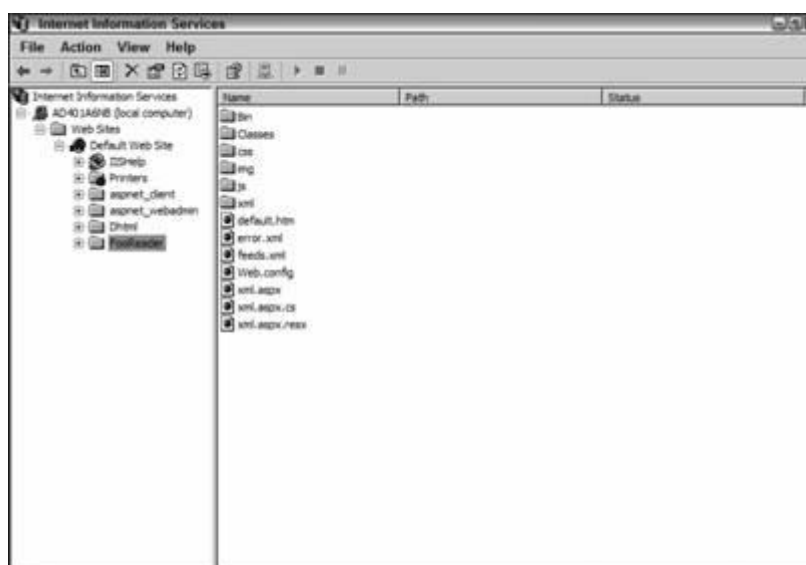
- 安装 FooReader.NET 的第一个要求是保证机器上已经安装了 IIS。IIS 是微软的 Web 服务器，仅在 Windows 2000 Professional、Windows 2000 Server、Windows XP Professional 和 Windows Server 2003 中可用。安装 IIS 需要有 Windows 安装光盘，并通过控制面板的“添加删除程序”中的“添加/删除 Windows 组件”来安装（参见图 5-3）。



• 图 5-3

- 为了运行 FooReader.NET 还需要安装 .NET 框架，这需要 1.1 或以上版本。对于 Window 98 及以后版本的用户可以在 <http://msdn.microsoft.com/netframework/downloads/updates/default.aspx> 免费下载。

- 当 IIS 和 .NET 框架都安装后，接着在 IIS 的 wwwroot 目录（位于 c:\inetpub\）中创建一个名为 FooReader 的目录，然后将 FooReader.NET 的所有文件都移到这个新建的 FooReader 目录下。当文件都放在 FooReader 目录下，还需要通过 IIS 管理控制器（参见图 5-4）向 IIS 注册这个应用程序。在计算机的控制面板（开始 → 控制面板）中双击“管理工具”，然后双击“Internet 信息服务”图标。



• 图 5-4

• 在 IIS 控制台中，你会看到一个文件和目录列表。这些都包含在 IIS 的根目录下。在左边的窗格中，定位在 FooReader 目录。在该目录上点击鼠标右键，然后在弹出菜单上选择“属性”，就可以看到 FooReader 目录的 Web 属性（如图 5-5 所示）。



• 图 5-5

• 在“目录”（Directory）选项卡中，会看到“应用程序设置”（Application Setting）部分。点击“创建”（Create）按钮，则属性窗口的内容将可见（参见图 5-6）。

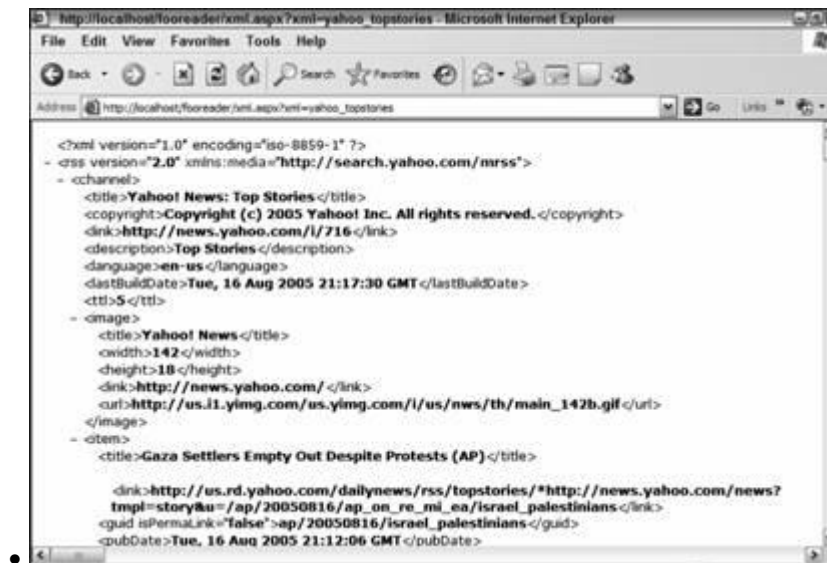


• 图 5-6

- 点击“确定”（OK）。IIS 现在就会把 FooReader 目录当作一个应用程序，并使其单独运行。到此，FooReader.NET 应用程序就安装完成了。

## • 5.5 测试

- 在部署任何 Web 应用程序之前，对安装进行测试总是一个好主意。为了测试的需要，feeds.xml 中只包含一个提要：Yahoo!News。
- 打开浏览器，访问 [http://localhost/fooreader/xml.aspx?xml=yahoo\\_topstories](http://localhost/fooreader/xml.aspx?xml=yahoo_topstories)。
- 这个测试用来确保服务器端组件能够正确地获取外部的新闻提要。如果一切工作正常，那么就会在浏览器中显示出一个 XML 提要（参见图 5-7）。
- 如果因某些原因使你看到了 ASP.NET 错误，则错误消息会告诉你该如何处理。最常见的错误是访问拒绝（Access Denied）错误，这时请对指定的 ASP.NET 用户账户（在 Windows 2003 下是 NETWORK SERVICE）的权限进行正确地修改。
- 如果你无权为 Web 服务器设置权限，例如当你从供应商那儿租用了一个 Web 空间，则可以通过打开 impersonation 来解决访问拒绝错误。impersonation 的设置位于 web.config 文件中。
- web.config 文件是一个针对 Web 应用程序、基于 XML 的配置文件。web.config 的根元素是<configuration/>，它通常紧跟着<system.web/>元素，你可以在<system.web/>元素后添加：



• 图 5-7

- `<identity impersonate="true"/>`
- 记住，这个解决方案只适用于非常情况，并且也依赖于 Web 服务器的设置。但是对于一个租用的服务器空间而言，它可能解决未授权访问的错误。
- 当应用程序通过测试并被证实可以工作，就可以编辑 feeds.xml 文件，使其包含你所希望的所有提要。

## • 5.6 小结

- 在本章中，探讨了在线聚合的历史，包括 RSS 和 Atom 这两种在聚合领域占统治地位的 XML 格式，同时也阐述了诸如 FooReader.NET 的聚合器是如何使用新闻提要更为简单的。
- 本章研究了基于 Web 的 RSS/Atom 聚合器 FooReader.NET 的创建与实现。首先从客户端组件开始，讨论如何创建 XParser，它是一个 JavaScript 中的 RSS/Atom 解析器，为基于 RSS 和 Atom 的 Web 应用程序开发人员提供了一个简单的使用接口，同时也讨论了 XParser 如何使用在微软的 IE 和基于 Mozilla 的浏览器中的 XML DOM。
- Web 应用程序客户端组件是正确向用户显示数据所必需的，服务器端组件则用来获取必要的信息。使用 C# 和 .NET 框架，可以了解如何获得远程主机上的 XML 提要，如何对其进行缓存处理，如何将它们输出到页面上。也可了解到如何设置 HTTP 的首部，使得浏览器知道期望什么，远程服务器知道向它请求什么。



- 最后，讨论了如何在 IIS 中设置一个应用程序，并保证其安装正确。

• 2000 年之后 XML 开始得到广泛使用，商业公司、开发人员及相关人员都在寻找一种使用它的新方法。XML 已经达成了将内容与表示层分离的目标，但如何才能够充分利用这个功能呢？答案就是以 Web 服务。

• Web 服务为应用程序和服务器之间的数据交换提供了一种途径。为了实现这种通信，Web 服务利用因特网在使用程序（consumer，使用这些数据的应用程序）和提供程序（provider，包含这些数据的服务器）之间传递由 XML 数据组成的消息。这与诸如 CORBA、DCOM 及 RMI 等传统的分布式计算模型不同，它们只是通过网络连接执行远程方法调用。与它们相比，Web 服务的主要不同在于传输的数据是 XML 文本而不是二进制格式。

• Web 服务的目标是为任何应用程序按需提供软件组件。不管是 Web 应用程序还是传统的桌面应用程序，都可以使用相同的服务来完成相同的任务。

## • 6.1 相关技术

• Web 服务不是一个单一的技术或平台，实际上，它们是一组协议、语言和数据格式的混合体。尽管现在有许多不同的平台都提供了自己的 Web 服务支持，但它们的基本组成部分仍然是一致的。

### • 6.1.1 SOAP

• SOAP 是由基于 XML 的语言和一些实现数据传递的公共协议组成的。SOAP 规范定义了一种复杂的语言，其中包括大量元素和属性，可以描述绝大多数类型的数据。这些信息可以通过许多协议来传输，不过最常见的方法还是与其他 Web 通信一样通过 HTTP 进行传送。

- SOAP 是简单对象访问协议（Simple Object Access Protocol）的首字母缩写。

- 可以通过两种方法来应用 SOAP：远程过程调用（RPC）风格和文档风格。

#### • 1. RPC 风格的 SOAP

• RPC 风格的 SOAP 将 Web 服务当成包含一个或多个方法的对象（在许多情况下，可以使用一个本地类建立与数据库的通信）。服务的请求包含调用的方法名和需要传入的

参数。被调用的方法在服务器上执行，并返回包含该方法的返回值的 XML 响应（如果有返回值的话）或者错误消息。假设有一个 Web 服务，它提供了一些诸如加、减、乘、除的简单算术运算操作。每个方法都包含两个数，并返回计算的结果。对加法操作的 RPC 风格请求如下所示：

- `<?xml version="1.0" encoding=" utf-8" ?>`
- `<soap:Envelope xmlns:soap=" http://schemas.xmlsoap.org/soap/envelope/`
- `soap:encodingStyle="`  
`http://schemas.xmlsoap.org/soap/encoding/">`
- `<soap:Body>`
- `<w:add xmlns:w=" http://www.wrox.com/services/math">`
- `<w:op1>4.5</w:op1>`
- `<w:op2>5.4</w:op2>`
- `</w:add>`
- `</soap:Body>`
- `</soap:Envelope>`

在处理重要的 XML 文档时，例如跨公司和客户共享的文档，那么命名空间就有了用武之地。命名空间在 SOAP 中相当重要，因为需要生成这些文档并在不同的系统中浏览。这个例子中，SOAP 命名空间的定义是 <http://schemas.xmlsoap.org/soap/envelope/>，它是基于 SOAP 1.1 的，你可以根据所使用的版本进行修改。SOAP 1.2 的命名空间是 <http://www.w3.org/2003/05/soap-envelope>。

• `<w:add/>` 元素说明被调用方法的名称（add）以及该本例中包含的其他命名空间 <http://www.wrox.com/services/math>。对于被调用服务的命名空间是由开发人员定义的。`soap:encodingStyle` 属性指向用来说明请求中的数据如何编码的 URI。编码风格有很多种，诸如 XML schema 中使用的类型系统。

• `<soap:Header/>` 元素是可选项，它用来包含一些诸如安全认证的附加信息。如果使用该元素，那么就应该放在 `<soap:Body/>` 之前。

• 如果加法操作的请求成功执行，那么响应消息类似于：

- `<?xml version="1.0" encoding=" utf-8" ?>`

- <soap:Envelope xmlns:soap=" <http://schemas.xmlsoap.org/soap/envelope/>"
- soap:encodingStyle="

<http://schemas.xmlsoap.org/soap/encoding/>">

- <soap:Body>
- <w:addResponse xmlns:w=" <http://www.wrox.com/services/math>">
- <w:addResult>9.9</w:addResult>
- </w:addResponse>
- </soap:Body>
- </soap:Envelope>

• 正如你所见，响应的格式与请求格式非常类似。它提供结果的标准方式是创建一个新元素，该元素的名称是方法名称后面加上字 Response。在本示例中，该元素是 <w:addResponse/>，与请求中的<w:add/>元素具有相同的命名空间。返回的实际结果则保存在<w:addResult/>元素中。Web 服务开发人员可以自行定义所有这些元素的名称。

• 如果服务器在处理 SOAP 请求时出现错误，那么将会返回<soap:Fault>元素。例如，如果示例中第一个操作数是字母而不是数字，那么将会接收到以下响应：

- <?xml version="1.0" encoding=" utf-8" ?>
- <soap:Envelope xmlns:soap=" <http://schemas.xmlsoap.org/soap/envelope/>">
- <soap:Body>
- <soap:Fault>
- <faultcode>soap:Client</faultcode>
- <faultstring>Server was unable to read request.
- Input string was not in a correct format.
- There is an error in XML document (4, 13).
- </faultstring>
- <detail/>
- </soap:Fault>
- </soap:Body>
- </soap:Envelope>

- 在响应中，<soap:Fault>元素有且只有一个，它对所遇到问题提供了一些信息。最有价值的信息包含在<faultcode/>中。该元素的可选值不多，最常见的是 soap:Server 和 soap:Client。错误代码 soap:Server 表示是服务器出现问题，诸如服务器无法连接数据库。在这种情况下，如果再次发送信息则程序可能会成功执行。如果错误代码是 soap:Client，那么通常是表示消息的格式错误，因此必须进行修改才能保证程序成功执行。

- 在<faultstring/>元素中存放着可读性更高的错误消息，它包含与应用程序相关的错误细节。如果引起 Web 服务产生错误的主要原因是诸如数据库的间接系统，那么该错误信息就会在可选的<faultactor/>元素中返回（在上一个例子中，并没有该元素）。

## • 2. 文档风格的 SOAP

- 文档风格的 SOAP 依赖于 XML schema 来说明请求和响应的格式。该风格越来越流行，并有人预测这种风格最终会取代 RPC 风格。在 <http://msdn.microsoft.com/library/en-us/dnsoap/html/argsoape.asp> 中，详细和清晰地解释了人们不愿意使用 RPC 风格 SOAP 编码的原因。

- 文档风格的请求看起来与 RPC 风格的请求并没有什么不同。例如，对于上一小节中 RPC 请求的例子，如果简单地移除 soap:encodingStyle 属性，那么就是一个有效的文档风格请求。这两种风格的不同之处在于，RPC 请求始终遵循相同的模式，即方法名位于其参数元素所在的外围元素中；而文档风格的请求则没有这样的约束。下面是一个与 RPC 请求完全不同的例子：

- <?xml version="1.0" encoding="utf-8" ?>
- <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
- <soap:Body>
- <w:add xmlns:w="http://www.wrox.com/services/math" op1="4.5" op2="5.4" />
- </soap:Body>
- </soap:Envelope>
- 请注意，突出显示的代码行位于一个元素中，包含了方法名称（add）和两个操作数（op1 和 op2）。这样的结构是不可能应用于 RPC 风格请求中的。文档风格之所以能有

这样的灵活性，主要是因为 XML schema 的存在。Web 服务可以使用该 XML schema 来校验请求的结构，并正确地自由使用请求中的信息。响应的基本规则与请求一致，它们可以与 RPC 风格相似，也可以完全不同，这也是基于 XML Schema 实现的。

- 使用 Visual Studio .NET 创建的 Web 服务，默认采用的是文档风格（尽管可以通过底层提供的不同属性来修改）。

- 现在你可能非常想了解 XML schema 保存在何处，并且该 schema 如何同时应用于客户和服务。该问题的答案就是 WSDL。

- 6.1.2 WSDL

- WSDL（Web 服务描述语言）是另一个基于 XML 的语言，该语言用来描述特定 Web 服务的用法，或者如何调用一个特定的 Web 服务。其规约描述了一种半透明、灵活的语言，对重用提供了最大的支持，狂热的拥护者大都会乐于手动创建这种的文档。一般情况下，使用工具来创建初始的 WSDL 文件，并根据需要手动调整。

- 下面的 WSDL 文件中描述了一个简单的算术服务，该服务提供一个加法操作（稍后建立）：

- `<?xml version="1.0" encoding="utf-8"?>`
- `<wscdl:definitions`
- `xmlns:http="http://schemas.xmlsoap.org/wscdl/http/"`
- `xmlns:soap="http://schemas.xmlsoap.org/wscdl/soap/"`
- `xmlns:s="http://www.w3.org/2001/XMLSchema"`
- `xmlns:tns="http://www.wrox.com/services/math"`
- `xmlns:wscdl="http://schemas.xmlsoap.org/wscdl/"`
- `targetNamespace="http://www.wrox.com/services/math">`
- `<wscdl:types>`
- `<s:schema elementFormDefault="qualified"`
- `targetNamespace="http://www.wrox.com/services/math">`
- `<s:element name="add">`
- `<s:complexType>`
- `<s:sequence>`

- `<s:element minOccurs="1" maxOccurs="1" name=" op1" type="s:float" />`
- `<s:element minOccurs="1" maxOccurs="1" name=" op2" type="s:float" />`
- `</s:sequence>`
- `</s:complexType>`
- `</s:element>`
- `<s:element name=" addResponse">`
- `<s:complexType>`
- `<s:sequence>`
- `<s:element minOccurs="1" maxOccurs="1" name=" addResult"`
- `type=" s:float" />`
- `</s:sequence>`
- `</s:complexType>`
- `</s:element>`
- `</s:schema>`
- `</wsdl:types>`
- `<wsdl:message name=" addSoapIn">`
- `<wsdl:part name=" parameters" element=" tns:add" />`
- `</wsdl:message>`
- `<wsdl:message name=" addSoapOut">`
- `<wsdl:part name=" parameters" element=" tns:addResponse" />`
- `</wsdl:message>`
- `<wsdl:portType name=" MathSoap">`
- `<wsdl:operation name=" add">`
- `<wsdl:documentation>`
- `Returns the sum of two floats as a float`
- `</wsdl:documentation>`
- `<wsdl:input message=" tns:addSoapIn" />`
- `<wsdl:output message=" tns:addSoapOut" />`

- `</wsdl:operation>`
- `</wsdl:portType>`
- `<wsdl:binding name=" MathSoap" type=" tns:MathSoap">`
- `<soap:binding transport=" http://schemas.xmlsoap.org/soap/http`
- `style=" document" />`
- `<wsdl:operation name=" add">`
- `<soap:operation soapAction="`

`http://www.wrox.com/services/math/add`

- `style=" document" />`
- `<wsdl:input>`
- `<soap:body use=" literal " />`
- `</wsdl:input>`
- `<wsdl:output>`
- `<soap:body use=" literal " />`
- `</wsdl:output>`
- `</wsdl:operation>`
- `</wsdl:binding>`
- `<wsdl:service name=" Math">`
- `<wsdl:documentation>`
- `Contains a number of simple arithmetical functions`
- `</wsdl:documentation>`
- `<wsdl:port name=" MathSoap" binding=" tns:MathSoap">`
- `<soap:address location=" http://localhost/Math/Math.asmx " />`
- `</wsdl:port>`
- `</wsdl:service>`
- `</wsdl:definitions>`

• 请记住，为了简单起见，这个 WSDL 文件只描述了一个实现加法功能的基本服务，将要实现的其他三个方法在此并未包含。虽然这个 WSDL 文件相当冗长和复杂，但是你应该理解每一小段代码的含义。

- 文档元素<wsdl:definitions/>包含正文内容,并允许在该处声明不同的命名空间。

下一个元素<wsdl:types/>包含 Web 服务所使用 XML schema。该元素的里面是<s:schema/>,描述了在请求或响应的<soap:Body/>中可能出现的所有元素的格式。

- schema 中的第一个元素是<add/>。因为<s:schema/>元素的属性

elementFormDefault 设置为 qualified,所以假设<add/>处于 targetNamespace 属性所指定的命名空间 [http:// www.wrox.com/services/math](http://www.wrox.com/services/math) 中。接着,声明<add/>元素以包含一组两个元素: <op1/>和<op2/>。这两个元素的 minOccurs 和 maxOccurs 的属性均设置为 1,这意味着它们只能出现一次。这两个元素还有着相同的 type 属性 s:float,这是 XML schema 的内建类型之一。

- 在 [www.w3.org/TR/xmlschema-0/#CreatDt](http://www.w3.org/TR/xmlschema-0/#CreatDt) 中,可以找到完整的 XML schema 数据类型的列表。如果 Web 服务需要更加复杂的类型,那么可以通过将基本类型聚合、约束来实现复杂类型。

- schema 的下一个元素是另一个<s:element/>,该元素描述了<addResponse/>。定义这个元素,使其有一个包含操作结果(同样定义为 s:float 类型)的子元素<addResult/>。这是内置的 XML schema 中最后一个元素。

- 现在回到 WSDL 文件的主体部分,这是一个简短的部分,描述了两个<wsdl:message/>元素: addSoapIn 和 addSoapOut。每个元素都包含一个<wsdl:part/>元素,<wsdl:part/>元素指定所使用的 XML schema 中的元素。这两个元素分别对应 add 和 addResponse 元素。该部分声明每个消息的格式。

- 下面的一段中,<wsdl:portType/>用来将<wsdl:message>元素分组到各个操作中。每个操作可以当成是可工作的独立单元,因此,由<wsdl:input>、<wsdl:output>和可选的<wsdl:fault>元素所组成。之前的示例包含一个<wsdl:portType>元素,并描述名为 add 的<wsdl:operation/>元素。<wsdl:input>和<wsdl:output>子元素的消息属性分别指向之前定义的<wsdl:message>元素。这段代码还有一个<wsdl:documentation/>元素,它包含对方法的描述,提高用户友好性(在本章的后面部分,将说明该信息源于何处)。

- 在端口类型的后面是<wsdl:binding/>代码段。每个绑定对(binding pair)将操作和与服务通信所用的协议组成一组。WSDL 规约中有三种绑定描述:SOAP、HTTP GET/POST 和 MIME。



- 本章主要介绍 SOAP 绑定。HTTP GET/POST 绑定用来处理 URL 的构建 (GET 请求)，以及表单数据的编码 (POST 请求)。MIME 绑定允许对部分消息 (通常是输出) 以不同的 mime 类型表示。这就意味着响应的一部分可以是 XML 格式，而另一部分则可以是 HTML 格式。

- 在 [www.w3.org/TR/2002/WD-wsdl12-bindings-20020709/](http://www.w3.org/TR/2002/WD-wsdl12-bindings-20020709/) 中，你可以了解关于这些绑定的更详细信息。

- 首先，绑定的 name 设为 MathSoap，type 属性指向 <wsdl:portType> 代码段中定义的 MathSoap 端口类型。其次，<soap:binding/> 元素使用 transport 属性来说明服务基于 HTTP。<wsdl:operation/> 元素只是简单地定义方法名称 add。<soap:operation/> 元素包含 soapAction 属性，该属性要和 style 属性一起添加到 HTTP 请求的首部，style 属性指定了 SOAP 消息是文档风格的，而不是 RPC 风格。

- 基于文档风格的消息和基于 RPC 风格的消息之间最大的区别在于，文档风格将消息作为 <soap:body> 中的一个元素进行发送，<soap:body> 并不限定结构，接收者和发送者对使用内嵌的 schema 达成一致。而 RPC 风格的消息则是有一个以调用方法命名的元素。每个方法接受的参数都对应一个元素。

- <soap:operation/> 元素包含两个子元素 <wsdl:input> 和 <wsdl:output>，这两个元素用来进一步描述请求和响应的格式。在本示例中，<soap:body> 将 use 属性定义为 literal。实际情况中，这是基于文档风格服务的唯一选项；基于 RPC 风格的服务还可以选择编码，在这种情况下，<soap:body> 可以对参数类型进一步准确指定具体的编码类型。

- 该文档的最后部分是 <wsdl:service/>，用来处理如何从客户端调用该服务。它包含一个易于阅读的服务描述，以及一个 <wsdl:port/> 元素，在该元素中引用了文档上一节中绑定的 MathSoap。在该小节中，最重要的元素或许是 <soap:address/>，它包含了含有访问该服务所需的 URL 的重要 location 属性。

- <wsdl:service name=" Math">
- <wsdl:documentation>
- Contains a number of simple arithmetical functions
- </wsdl:documentation>
- <wsdl:port name=" MathSoap" binding=" tns:MathSoap">

- `<soap:address location=" http://localhost/Math/Math.asmx" />`
- `</wsdl:port>`
- `</wsdl:service>`

• 对新的开发人员来说，了解一个完整的 WSDL 文件是件令人畏惧的事情，但是幸好我们不必手工编写 WSDL 文件。事实上，本节的示例文件是由 .NET 的 Web 服务自动创建的，该服务将检查底层代码，并自动生成所需的 XML。

• XML schema 是一个庞大的主题，超出了本书的讨论范畴。如果了解更多关于 XML schema 的知识，可以参考 *Beginning XML* 第 3 版 (Wiley 出版，ISBN 0-7645-7077-3)，或者从 [www.w3schools.com/schema/default.asp](http://www.w3schools.com/schema/default.asp) 获取 Web 教程。

### • 6.1.3 REST

• 具像状态传输 (Representational State Transfer)，通常简称为 REST，描述了一种使用现有 HTTP 协议传输数据的方式。尽管 REST 广泛用于 Web 服务，但是 REST 还可以用于任何基于 HTTP 请求和响应的系统。对于 Web 服务，REST 允许以指定格式调用 URL，并返回指定格式的数据。该数据可以包含如何获取更多数据的信息。在 Web 服务应用中，数据通常以 XML 的格式返回。

• 例如，假设 Wrox 想要为其他人提供一种获取所有作者列表的方法。REST 风格的 Web 服务使用简单的 URL 对数据进行访问。Wrox Book 服务使用下面链接获取作者列表：

- <http://www.wrox.com/services/authors/>

• 该服务可能返回一个包含作者信息的 XML，并提供访问每个作者的详细信息的方式，例如：

- `<?xml version="1.0" encoding=" utf-8" ?>`
- `<authors xmlns:xlink="http://www.w3.org/1999/xlink"`
- `xmlns="http://www.wrox.com/services/authors-books"`
- `xlink:href="http://www.wrox.com/services/authors/">`
- `<author forenames="Michael" surname="Kay"`
- `xlink:href="http://www.wrox.com/services/authors/kaym"`
- `id=" kaym"/>`
- `<author forenames="Joe" surname="Fawcett"`

- `xlink:href="http://www.wrox.com/services/authors/fawcettj"`
- `id=" fawcettj"/>`
- `<author forenames="Jeremy" surname=" McPeak"`
- `xlink:href="http://www.wrox.com/services/authors/mcpeakj"`
- `id=" mcpeakj"/>`
- `<author forename=" Ni cholas" surname=" Zakas"`
- `xlink:href="http://www.wrox.com/services/authors/zakasn"`
- `id=" zakasn"/>`
- `<!--`
- `More authors`
- `-->`
- `</authors>`

• 该 XML 有许多值得注意的地方。首先，声明默认命名空间为 <http://www.wrox.com/services/authors-books>，那么任何没有前缀的元素，例如`<authors/>`，都假定属于该命名空间。这就意味着`<authors/>`元素与其他元素虽然名称相同，但是可以根据不同的命名空间加以区别。命名空间 URI <http://www.wrox.com/services/authors-books> 只作为简单的唯一字符串使用，这并不保证该 Web 位置有实际的资源可以访问。关键在于，这是统一资源标识符 URI，只是一个标识符，而不是统一资源定位符 URL，URL 才表示该位置有可访问的资源。

• 其次，请注意 <http://www.w3.org/1999/xlink> 命名空间的 `href` 属性的使用。尽管该属性不是基本元素，但是大多数 REST 风格的 Web 服务已经将该符号标准化（该符号在 HTML 是个标准的超链接）。

• Xlink 是一种链接文档的方式，与 HTML 的超链接有所区别。Xlink 允许双向依赖，所以只要声明链接如何生效，那么文档之间就可以互相访问。例如，手动声明，自动声明或者某段预先设定的时间后。XPointer 是 Xlink 的“兄弟”，它可以指向文档内的某个指定段落，其功能比 HTML 页面中链接强大许多。

• 尽管 W3C 同时推荐这两种方法，但是它们仍然没有得到广泛的使用。更多详细信息请参见 [www.w3.org/XML/Linking](http://www.w3.org/XML/Linking)。

• 如果在网站或应用程序中使用，REST 服务返回的 XML 将在客户端或服务器端转换成更为客户接受的格式（例如 HTML）。转换后的结果类似于：

```
• <html>
• <head>
• <title>Wrox Authors</title>
• </head>
• <body>
• Michael Kay
• Joe
Fawcett
• Jeremy
McPeak
• Nicholas
Zakas
• </body>
• </html>
```

• 用户可以根据每个链接获取单个作者数据，返回的 XML 类似于：

```
• <?xml version="1.0" encoding="utf-8" ?>
• <author xmlns:xlink="http://www.w3.org/1999/xlink"
• xmlns="http://www.wrox.com/services/authors-books"
• xlink:href="http://www.wrox.com/services/authors/fawcettj"
• id="fawcettj" forenames="Joe" surname="Fawcett">
• <books>
• <book
• xlink:href="http://www.wrox.com/services/books/0764570773"
• isbn="0764570773" title="Beginning XML"/>
• <book
• xlink:href="http://www.wrox.com/services/books/0471777781"
• isbn="0471777781" title="Professional Ajax"/>
```

- </books>
- </author>
- 同样可以看到，这些元素在命名空间

<http://www.wrox.com/services/authors-books> 中，可以通过 `xlink:href` 属性获取更多信息。该数据的 HTML 表现方式可能是：

- <html>
- <head>
- <title>Author Details</title>
- </head>
- <body>
- <p>Details for
- <a href="http://www.wrox.com/services/authors/fawcettj">Joe

Fawcett</a></p>

- <p>Books</p>
- <a href="http://www.wrox.com/services/books/0764570773">Beginni ng

XML</a>

- <a

href="http://www.wrox.com/services/books/0471777781">Professional Ajax</a>

- </body>
- </html>

- 如果客户期望得到 *Professional Ajax* 一书的更多信息，那么他可以在响应中获取下

述 XML 数据：

- <?xml version="1.0" encoding=" utf-8" ?>
- <book xmlns:xlink="http://www.w3.org/1999/xlink"
- xmlns="http://www.wrox.com/services/authors-books"
- xlink:href="http://www.wrox.com/services/books/0471777781"
- isbn="0471777781">
- <genre>Web Programmi ng</genre>
- <title>Professional AJAX</title>

- <description>How to take advantage of asynchronous JavaScript
- and XML to give your web pages a rich UI.</description>
- <authors>
- <author forenames=" Nicholas" surname=" Zakas"
- xlink:href="<http://www.wrox.com/services/authors/zakasn>"
- id=" zakasn" />
- <author forenames=" Jeremy" surname=" McPeak"
- xlink:href="<http://www.wrox.com/services/authors/mcpeakj>"
- id=" mcpeakj" />
- <author forenames=" Joe" surname=" Fawcett"
- xlink:href="<http://www.wrox.com/services/authors/fawcettj>"
- id=" fawcettj" />
- </authors>
- </book>

• REST 风格的 Web 服务相当简单易懂，并遵循着同样的模式。例如，可以使用 <http://www.wrox.com/services/authors/> 得到完整的作者列表，通过简单的修改（在链接最后加上作者 ID），就可以得到单个作者的信息，例如

<http://www.wrox.com/services/authors/fawcettj>。

• Web 服务可以通过多种方式来实现。包括通过静态网页，以及更常用的诸如 ASP、JSP 或 PHP 等服务器端处理技术，从数据库获取数据并返回合适的 XML 结构。在这个例子中，URL 将会通过服务器映射到一个用来获取数据的特定于应用程序的方法上，例如调用数据库的存储过程。

• 在 [www.network.world.com/ee/2003/ee/rest.html](http://www.network.world.com/ee/2003/ee/rest.html) 中，可以了解到更多关于 REST 风格服务（也称为 RESTful 服务）的信息。

## • 6.2 .NET 连接

• 在所有的成员中，微软是将 SOAP 引入 Web 服务的先锋。当微软把 SOAP 作为一种数据交换方式向 IBM 推荐时，IBM 毫不犹豫地加入了，促使 SOAP 演化成了 WSDL。在微软和 IBM 的强力合作下，吸引了越来越多大公司加入，其中包括 Oracle、Sun 和 HP 等。

随着标准的制定，也就揭开了 Web 服务的新纪元。但是还有一个遗漏：没有一个用来创建 Web 服务的工具。于是就诞生了 .NET。

- 微软在 2000 年发布了 .NET 框架，其目标是提供一个平台无关的框架来与 Java 竞争。由于 .NET 从一开始就主动内建了强有力的 XML 支持，同时也使得能够为基于 SOAP 和 WSDL 的 Web 服务提供良好的支持。使用 .NET，为现有应用程序的 Web 服务封装提供了简单的方式，同时还可以使用 Web 服务来输出 .NET 类。

- 在开发 Web 服务时，必须决定如何与 SOAP 和 WSDL 进行交互。虽然 .NET 为开发人员提供了一种隐藏底层细节的工具，但是如果必要，开发人员也是可以自行对细节进行修改的。.NET 框架的 2005 版更加充分地利用 XML 和 Web 服务。

## • 6.3 设计决策

- 尽管 .NET 框架使得 Web 服务的开发更加容易，但是并不意味着这是唯一创建 Web 服务的方法。如同其他程序设计，还需要对设计和开发等进行决策。记住，Web 服务提供了一种平台无关的数据请求和接收方式，所以 Web 服务的使用者不需要知道实现细节。不幸的是，互相操作时有一些注意事项：

- **q** 并不是所有平台都支持相同的数据类型。例如，许多 Web 服务会返回 ADO.NET 数据集。非 .NET 系统就无法识别该数据类型。同样，数组也存在问题，因为它们的表现方式可能存在不同。

- **q** 一些 Web 服务允许在请求中遗漏首部或增加额外首部。当 Web 服务使用者没有发送出全部正确的首部时，那么就会产生问题，特别是需要可靠服务时。

- 为解决这些及其相关的问题，成立了 Web 服务互操作组织（Web Services Interoperability Organization）。在网站 [www.ws-i.org/](http://www.ws-i.org/) 上可以了解该组织的目的、定位以及一致性推荐。

- 创建 Web 服务时，首先要决定所使用的平台。如果选择 Windows，那么基本要选用 IIS 作为 Web 服务器。可以使用 ASP.NET 来创建 Web 服务，或者旧版本 IIS 支持的 ASP（尽管困难得多）。本章中的例子采用的是 ASP.NET 技术。

- 如果使用 UNIX 或者 Linux，那么就可能使用 JSP 或者 PHP，这两种技术都有开源的 Web 服务器。使用这些需要分别用 Java 或者 PHP 来编写代码以创建 Web 服务。

- Axis 项目 (<http://ws.apache.org/axis/>) 为 Java 和 C++ 提供开发工具。
- 对于 PHP 也有许多可选项, 例如 PhpXMLRPC (<http://phpxmlrpc.sourceforge.net/>) 和 Pear SOAP (<http://pear.php.net/package/SOAP>)。

• 当选定开发语言后, 就需要决定谁能访问 Web 服务。只有特定的应用程序访问, 还是允许公开访问? 如果是后者, 那么需要考虑之前提到的互操作问题; 如果是前者, 那么可以充分利用客户端或服务端所提供的一些特性的优点。

• 在创建了 Web 服务后, 接下来就是使用它。任何调用 Web 服务的应用程序被看作是使用者。通常, 使用 Web 服务都将遵循一个特定的模式: 创建请求、发送该请求、然后针对收到的响应进行操作。处理这些步骤的方法由使用者可以访问的功能决定。

## • 6.4 创建 Windows 平台的 Web 服务

• 现在应该是告别规约和理论, 真正动手创建一个简单 Web 服务的时候了。本节描述的 Web 服务使用文档风格的 SOAP 请求和响应来实现之前在 WSDL 文件中所描述的 Math 服务。请注意, 在这个过程中采用了微软提供的免费工具, 它比使用 Visual Studio .NET 而言需要更多的工作量。然而, 这些额外的工作量将有助于更加深刻地理解 Web 服务, 以后的开发工作中如果自动生成 Web 服务出了某些问题, 你就不会手足无措了。

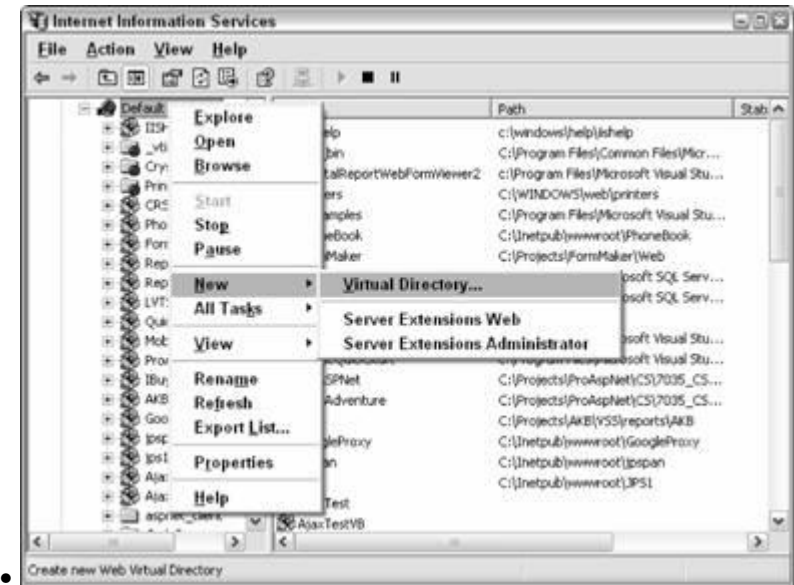
### • 6.4.1 系统需求

- 要创建这个 Web 服务, 至少需要满足以下三个需求:
- **q** 运行 IIS5 或更高版本的 Windows 机器。所有 XP 之后的 Professional 版本, 或者 Windows 2000 之后的 Server 版本都符合需求。
- **q** 必须在运行 IIS 的机器上安装 .NET 框架。还需要在开发机器上安装 .NET 软件开发包 (SDK)。本示例假设在运行 IIS 的机器上进行开发 (.NET 框架和 SDK 都可以从 <http://msdn.microsoft.com/netframework/downloads/updates/default.aspx> 下载)。
- **q** 用来编写代码的文本编辑器。可以采用所有 Windows 操作系统自带的记事本 (Notepad) 程序, 对于这个例子来说已经足够 (尽管高级的开发环境支持语法突出显示等功能)。

### • 6.4.2 配置 IIS



• 创建 Web 服务的第一个任务是为该服务创建 home 目录。点击开始菜单 **管理** 工具，并选择 Internet 信息服务（或者，点击开始 **运行**，并输入 %SystemRoot%\System32\inetmgr\ iis.msc，回车）。展开左侧的树结构，显示出默认网站节点，然后点击右键，选择新建 **虚拟目录**，如图 6-1 所示。这时会弹出虚拟目录创建向导，在此选择客户端可见的 Web 服务名称（如图 6-2）。



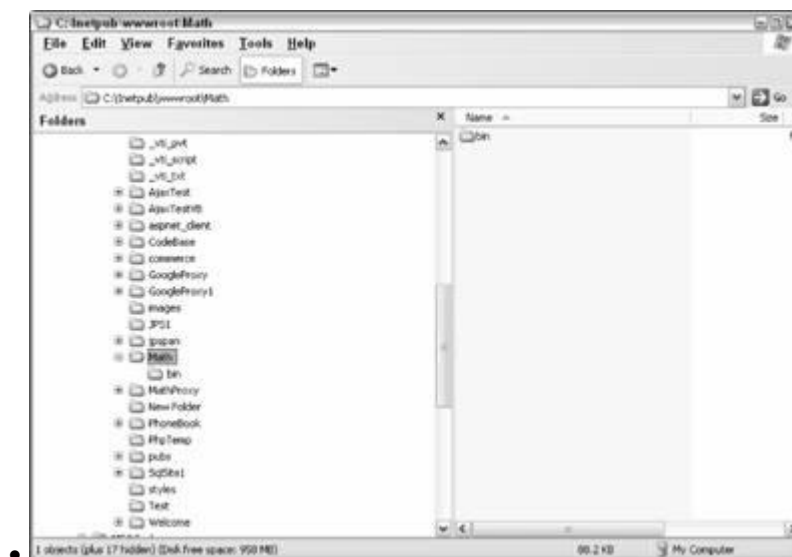
• 图 6-1



• 图 6-2

• 将该目录命名为 **Math**，并点击下一步。下一个界面中，浏览标准的 IIS 目录 C:\InetPub\wwwroot。在该目录下直接创建新的目录，同样命名为 **Math**。余下界面和向

导都选择默认项。完成后,使用 Windows 资源浏览器在 Math 目录下创建一个子目录 bin,用来存放 Web 服务的 DLL 文件。现在的目录结构如图 6-3 所示。



### • 6.4.3 编写 Web 服务

- 这里创建的 Web 服务相当简单。Web 服务名称为 Math, 并实现了四种基本算术运算: 加法、减法、乘法和除法。这四种运算中的每个都包含两个浮点型参数, 并返回浮点型结果。这个类通过 C# 语言编写, 并通过 ASP.NET 发布该 Web 服务。

- 在文本编辑器中创建一个新的文件, 并添加下列三行:

- using System;
- using System.Web;
- using System.Web.Services;

- 这段代码并没有增加任何功能, 只是用来避免输入带路径的类全称的不便。因为该 Web 服务将使用这些命名空间中的一些类, 所以需要引用这些命名空间以节省空间。

- 紧接着, 创建名为 Wrox.Services 的命名空间, 以及 Math 类, 该类是对 System.Web.Services.WebService 的继承:

- namespace Wrox.Services
- {
- [WebService (Description = "Contains a number of simple arithmetical functions",

- `Namespace = "http://www.wrox.com/services/math")]`
- 
- `public class Math : System.Web.Services.WebService`
- `{`
- `//类代码`
- `}`
- `}`

• 关键字 `namespace` 与 XML 中命名空间的用法相同。这意味着 `Math` 类的全路径是 `Wrox.Services.Math`。命名空间定义里的第一个属性是 `WebService`，表示下面代码行的类是一个 Web 服务。通过这种做法可以为该类增加额外的功能，例如生成 WSDL 文件。`WebService` 属性中还包含一个 `Description` 参数（同样包含在 WSDL 文件中）。

• 紧接着是类名 `Math`，从基类 `System.Web.Services.WebService` 中继承。从该类中继承就意味着可以不必费心编写 Web 服务细节代码，在基类中已经实现了该功能。只需把注意力集中在 Web 服务的发布方法即可。

• 定义 Web 服务使用的方法与编写平常的方法一样简单，只需添加上特殊的 `WebMethod` 属性标识即可：

- `[WebMethod(Description = "Returns the sum of two floats as a float")]`
- `public float add(float op1, float op2)`
- `{`
- `return op1 + op2;`
- `}`

• 再强调一次，这段代码非常简单。（还有什么会比加法运算更简单吗？）所有带有 `WebMethod` 属性前缀的方法都被当成 Web 服务的一部分。`Description` 参数成为所生成的 WSDL 文件的一部分。尽管可以在 Web 服务中编写许多方法，但是本示例的代码只包含四种算术运算方法：

- `using System;`
- `using System.Web;`
- `using System.Web.Services;`

- 
- namespace Wrox.Services
- {
  - [WebService (Description = "Contains a number of simple arithmetical functions",
    - Namespace = "<http://www.wrox.com/services/math>")]
    - public class Math : System.Web.Services.WebService
    - {
      - 
      - [WebMethod(Description = "Returns the sum of two floats as a float")]
        - public float add(float op1, float op2)
        - {
          - return op1 + op2;
          - 
          - }
          - 
          - [WebMethod(Description = "Returns the difference of two floats as a float")]
            - public float subtract(float op1, float op2)
            - {
              - return op1 - op2;
              - }
              - 
              - [WebMethod(Description = "Returns the product of two floats as a float")]
                - public float multiply(float op1, float op2)
                - {
                  - return op1 \* op2;
                  - }

- 
- `[WebMethod(Description = "Returns the quotient of two floats as a float")]`
- `public float divide(float op1, float op2)`
- `{`
- `return op1 / op2;`
- `}`
- `}`
- `}`

- 将该文件保存在 `Math` 目录中，并命名为 `Math.asmx.cs`。

- 创建另一个文本文件，并输入：

- `<%@WebService Language=" c#" Codebehind=" Math.asmx.cs" Class="Wrox.Services.Math" %>`

• 这是使用刚才创建的 `Math` 类的 ASP.NET 文件。`@WebService` 指令表示这是一个 Web 服务。其他属性的含意非常显而易见：`Language` 表示代码所使用的语言；`Codebehind` 表示代码所在的文件名称；`Class` 表示所使用的类的全路径。同样将该文件保存在 `Math` 目录中，并命名为 `Math.asmx`。

#### • 6.4.4 创建程序集

- 创建完这两个文件后，现在开始下一个步骤：将源代码编译成程序集（DLL）。这个步骤需要使用 .NET SDK 包含的 C# 编译器。该编译器位于 `Windows` 目录 `Microsoft.Net\Framework\ <version number>` 文件夹中（例如，`C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\`）。

- 编译并调试代码的最简单方法是创建一个批处理文件。创建另一个文本文件，并输入（可以将下面的命令放在一行）：

- `C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\csc.exe /r:System.dll`
- `/r:System.Web.dll`
- `/r:System.Web.Services.dll /t:library /out:bin\Math.dll Math.asmx.cs`

- 根据实际情况修改 csc.exe 的相应路径，并将该文件保存到 Math 目录中，命名为 Make-Service.bat。

- 如果使用的编辑器是记事本，那么在使用“另存为”对话框要特别注意：确保文件类型框选择为所有文件，或者将文件名用双引号括起来。否则记事本会给文件加上.txt 的后缀名。

- 接下来就是编译 DLL。点击开始运行，然后输入 cmd，进入命令提示符窗口。输入 cd\inetpub\wwwroot\Math，从而转到 Math 目录。最后，运行批处理文件：

- C:\inetpub\wwwroot\Math\MakeService.bat

- 如果一切顺利，那么可以看到编译器显示版本和版权信息，之后是一个空白行，这就表示编译成功（如果出错，那么控制台会显示出错信息。可以根据提示的错误信息纠正语法和拼写错误）。

- 假设 DLL 编译成功，现在就可以准备测试 Web 服务了。.NET Web 服务一个有趣之处在于，整个测试是自动创建的。打开 Web 浏览器，浏览 <http://localhost/Math/math.asmx>，所看到的页面与图 6-4 大致相同。



• 图 6-4

- 你现在可以对这四个方法进行测试，或者通过服务描述符的链接来查看所生成的 WSDL 文件。该 WSDL 文件与之前的示例大致相同，唯一的不同是该 WSDL 文件包含了所有四种方法的入口。

- 另外一种查看 WSDL 文件的方法是，在 Web 服务 URL 中添加?WSDL，例如 `http://localhost/Math/math.asmx?WSDL`。

- 由于你已经对 add 方法很熟悉了，因此可以试试 divide 方法。点击上一个页面的链接后将显示图 6-5 所示的页面。



• 图 6-5

- divide 标题下面是使用 WebMethod 属性的描述，再下面是一个小的测试表单。如果输入两个数字，例如 22 和 7，那么就能得到图 6-6 中所显示的结果。



• 图 6-6

- 在产品环境中，可以删除 Math.asmx.cs 文件，在运行时并不需要该文件。Math.asmx 会将所有请求直接传给 DLL。

- 该测试并没有在请求和响应中使用 SOAP，事实上只是通过 POST 请求来传入两个操作数。这个过程的处理细节显示在 divide 页面底部，如图 6-5 所示。现在已经定义好 Web 服务，那么接下来就是使用 Ajax 调用该服务。

## • 6.5 Web 服务与 Ajax

- 我们现在已经对 Web 服务有了基本的了解，并且还实际动手创建一个 Web 服务，接下来要了解 Web 服务如何与 Ajax 协作。Web 服务是 Ajax 应用获取信息的另一种渠道。在上一章中，介绍了如何使用 RSS 和 Atom 来向用户显示信息，这与使用 Web 服务非常相似。最主要的区别在于，使用 Web 服务，不仅可以获取信息，还可以发送数据到可操作的服务器并接收返回结果。

- 只要使用主流的浏览器，就可以通过 JavaScript 在 Web 页面中使用 Web 服务。IE5.0 或更高版本，以及 Mozilla 系列浏览器（Firefox），都包含使用 Web 服务的一部分功能。

### • 6.5.1 创建测试工具

- 首先需要创建一个测试工具（Test Harness），对浏览器中调用 Web 服务的不同方法进行测试。这个测试工具相当简单：一个用来选择所要执行的算术运算操作的下拉选择框，输入两个操作数的文本框，以及调用 Web 服务的按钮。当页面完全载入时才激活这些控件。这些控件下方是另外一个用来显示结果的文本框，以及显示请求和响应数据的两个文本区域。

- `<html>`
- `<head>`
- `<title>Web Service Test Harness</title>`
- `<script type=" text/javascript">`
- 
- `var SERVICE_URL = "http://localhost/Math/Math.aspx";`
- `var SOAP_ACTION_BASE = "http://www.wrox.com/services/math";`
- 
- `function setUIEnabled(bEnabled)`



```

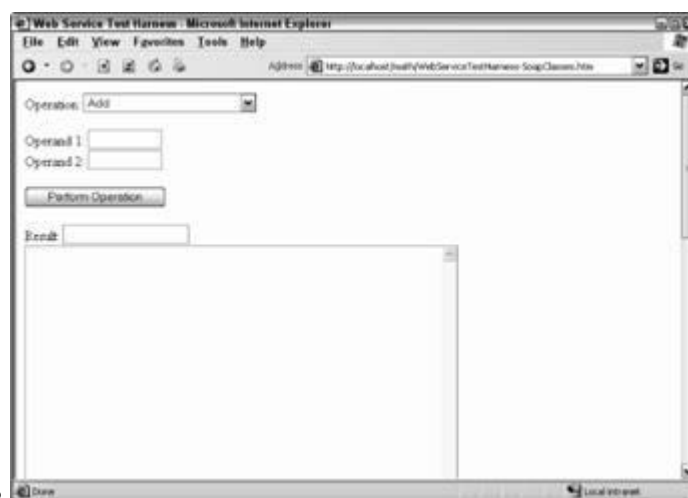
• {
• var oButton = document.getElementById("cmdRequest");
• oButton.disabled = !bEnabled;
• var oList = document.getElementById("lstMethods");
• oList.disabled = !bEnabled
• }
•
• function performOperation()
• {
• var oList = document.getElementById("lstMethods");
• var sMethod = oList.options[oList.selectedIndex].value;
• var sOp1 = document.getElementById("txtOp1").value;
• var sOp2 = document.getElementById("txtOp2").value;
•
• // 清空消息面板
• document.getElementById("txtRequest").value = "";
• document.getElementById("txtResponse").value = "";
• document.getElementById("txtResult").value = "";
• performSpecificOperation(sMethod, sOp1, sOp2);
• }
• </script>
• </head>
• <body onload=" setUIEnabled(true)">
• Operation: <select id=" lstMethods" style=" width: 200px" disabled="
di sabled">
• <option value=" add" selected=" selected">Add</option>
• <option value=" subtract">Subtract</option>
• <option value=" multiply">Multiply</option>
• <option value=" divide">Divide</option>
• </select>

```

- `<br/><br/>`
- Operand 1: `<input type=" text" id=" txtOp1" size="10"/><br/>`
- Operand 2: `<input type=" text" id=" txtOp2" size="10"/><br/><br/>`
- `<input type=" button" id=" cmdRequest"`
- `value=" Perform Operation"`
- `onclick=" performOperation();" di sabl ed=" di sabl ed"/>`
- `<br/><br/>`
- Result: `<input type=" text" size="20" id=" txtResult">`
- `<br/>`
- `<textarea rows="30" col s="60" id=" txtRequest"></textarea>`
- `<textarea rows="30" col s="60" id=" txtResponse"></textarea>`
- `</body>`
- `</html >`

• `setUIEnabled()` 函数用来激活或禁止测试工具的用户界面。这将确保每次只发送一个请求。这里还定义了两个常量，`SERVICE_URL` 和 `SOAP_ACTION_BASE`，它们包含 Web 服务的 URL 和调用 Web 服务所需的 SOAP 动作的标题。按钮调用 `performOperation()` 函数，获取相应的数据，并在调用 `performSpecfi cOperation()` 函数前清除文本框的内容。

`performSpecfi c- Operation()` 方法必须根据使用的特殊测试定义，用来执行 Web 服务的调用（包含在一个 JavaScript 文件中）。根据于你选择的个人浏览器，该页面大致如图 6-7 所示。



• 图 6-7

## • 6.5.2 IE 使用的方法

• 在开发人员对 Web 服务发生兴趣后，微软针对 IE 浏览器开发了一套 Web 服务行为。该行为允许重新定义现有 HTML 元素的功能、属性和方法，或者创建一个完整的替代者。它的优点在于将功能封装到一个后缀名为 .htc 的独立文件中。虽然该行为所承诺的功能让人失望，但是 Web 服务行为却成为一个对 Web 开发人员非常有用的组件。在 <http://msdn.microsoft.com/library/default.asp?url=/workshop/author/webservice/web-service.asp> 中可以下载到该 Web 服务行为。

msdn.microsoft.com/library/default.asp?url=/workshop/author/webservice/web-service.asp 中可以下载到该 Web 服务行为。

• 本章所涉及的文件或代码，都包含在 [www.wrox.com](http://www.wrox.com) 网站中本书配套的下载包中。

• 往元素中添加行为有多种方法。最直接的方法是使用元素的 CSS style 属性。在 <div/> 中添加 Web 服务行为，代码如下：

• <div id="divServiceHandler" style="behavior: url(web-service.htc);"></div>

• 假设该行为与 Web 页面处于服务器上相同的目录中。

• 为了展示如何使用该行为，我们创建一个新版本的测试装置，并在 <body/> 元素下插入一行新代码（即突出显示的行），例如：

• <body onload="setUIEnabled(true);">

• <div id="divServiceHandler" style="behavior: url(web-service.htc);"></div>

• Operation: <select id="lstMethods" style="width: 200px" name="lstMethods"

• disabled="disabled">

• 接下来，则是使用 Web 服务行为定义相关的 performSpecificOperation() 方法。该方法接受三个参数：方法名和两个操作数。其代码如下：

• var iCallId = 0;

• function performSpecificOperation(sMethod, sOp1, sOp2)

• {

- `var oServiceHandler = document.getElementById("divServiceHandler");`
- `if (!oServiceHandler.Math)`
- `{`
- `oServiceHandler.useService(SERVICE_URL + "?WSDL", "Math");`
- `}`
- `iCallId =`

```
oServiceHandler.Math.calculateService(handleResponseFromBehavior,
```

- `sMethod, s0p1, s0p2);`
- `}`

- `iCallId` 变量初始化为 0。尽管测试中没有用到该变量，但是该变量用来跟踪多个并发调用。然后，将包含行为的 `<div>` 元素的引用保存在 `oServiceHandler` 中。接下来通过测试检查 `Math` 属性是否存在，来判断是否曾经用到该行为。如果不存在，那么必须将 WSDL 文件的 URL 和服务标识符名称传入 `useService()` 方法，完成对行为的设置。需要服务标识符的原因是允许行为每次使用多个 Web 服务。接着传入回调函数 (`handleResponse-FromBehavior()`)、方法名称以及两个参数，以执行 `callService()` 方法。

- 一旦接收到响应，那么就会调用回调函数 `handleResponseFromBehavior()`：
- `function handleResponseFromBehavior(oResult)`
- `{`
- `var oResponseOutput = document.getElementById("txtResponse");`
- `if (oResult.error)`
- `{`
- `var sErrorMessage = oResult.errorDetail.code`
- `+ "\n" + oResult.errorDetail.string;`
- `alert("An error occurred:\n"`
- `+ sErrorMessage`
- `+ "See message pane for SOAP fault.");`
- `oResponseOutput.value = oResult.errorDetail.raw.xml;`
- `}`

- else
- {
- var oResultOutput = document.getElementById("txtResult");
- oResultOutput.value = oResult.value;
- oResponseOutput.value = oResult.raw.xml;
- }
- }

• 回调函数的参数是一个 oResult 对象，该对象包含调用的细节。如果 error 属性不为 0，那么就会显示相关的 SOAP 错误信息；否则就将返回值 oResult.value 显示在页面上。

• performSpecificOperation() 和 handleResponseFromBehavior() 函数可以放在一个外部 JavaScript 文件中，并在测试工具页面中使用 <script/> 元素包含该文件，例如：

```

• <script type=" text/javascript" src="
WebServiceExampleBehavior.js"></script>

```

• 综上所述，Web 服务行为的使用是相当简单的。所有工作都在后台行为中完成，尽管 webservice.htc 文件是一个相当大的脚本文件(51KB)，但是也提供了一些相当有用的功能。

• 如果要想了解行为是如何工作的，那么请使用文本编辑器查看 webservice.htc 文件。不过，该文件并不是教程，且包含将近 2300 行 JavaScript 代码

### • 6.5.3 Mozilla 使用的方法

• 诸如 Firefox、Netscape 等基于 Mozilla 的现代浏览器，都在它们的 JavaScript 实现中内建了一些高级 SOAP 类。这些浏览器试图将 SOAP 调用的基本策略封装成为易于使用的类。与上个例子一样，首先必须定义 performSpecificOperation() 函数：

- function performSpecificOperation(sMethod, sOp1, sOp2)
- {

- `var oSoapCall = new SOAPCall();`
- `oSoapCall.transportURI = SERVICE_URL;`
- `oSoapCall.actionURI = SOAP_ACTION_BASE + "/" + sMethod;`
- `var aParams = [];`
- `var oParam = new SOAPParameter(sOp1, "op1");`
- `oParam.namespaceURI = SOAP_ACTION_BASE;`
- `aParams.push(oParam);`
- `oParam = new SOAPParameter(sOp2, "op2");`
- `oParam.namespaceURI = SOAP_ACTION_BASE;`
- `aParams.push(oParam);`
- `oSoapCall.encode(0, sMethod, SOAP_ACTION_BASE, 0, null, aParams.length,`  
`aParams);`
- `var oSerializer = new XMLSerializer();`
- `document.getElementById("txtRequest").value =`  
`oSerializer.serializeToString(oSoapCall.envelope);`
- `setEnabled(false);`
- 
- 
- `//其他代码`

•该段脚本使用一些内建类，首先是 SOAPCall。这个类与 IE 的 Web 服务行为类似，都是封装 Web 服务的功能。创建 SOAPCall 实例后，为其设置两个属性：指向 Web 服务的位置的 transportURI；指定 SOAP 操作和方法名称的 actionURL。

•接着，使用 SOAPParameter 构造函数（传入要创建的参数名称和值）创建两个参数。每个参数的命名空间 URI 设置成 WSDL schema 段落中 targetNamespace 的值。理论上并不需要这么做，但是 Mozilla SOAP 类是基于 RPC 风格的，而这里的 Web 服务是基于文档风格的，因此需要这个额外的步骤。这两个参数都存入 aParams 数组。encode()函数准备好调用需要的所有数据。该 Web 服务调用共需要 7 个参数。第一个参数是所使用 SOAP 的版本号，该参数设置为 0，除非需要特别指定版本。第二

个参数是所使用的方法名称,第三个参数是WSDL文件scehma部分的targetNamespace。接下来的一个参数是调用中所需额外标题的个数(本示例为0),然后是这些标题所组成的数组(这里设置为null)。最后两个参数分别是发送的SOAPParameter对象的个数和实际的参数。

•紧接着就将发送实际的请求,所要完成的工作是使用asyncInvoke()方法,例如:

```
• function performSpecificOperation(sMethod, sOp1, sOp2)
• {
• var oSoapCall = new SOAPCall();
• oSoapCall.transportURI = SERVICE_URL;
• oSoapCall.actionURI = SOAP_ACTION_BASE + "/" + sMethod;
• var aParams = [];
• var oParam = new SOAPParameter(sOp1, "op1");
• oParam.namespaceURI = SOAP_ACTION_BASE;
• aParams.push(oParam);
• oParam = new SOAPParameter(sOp2, "op2");
• oParam.namespaceURI = SOAP_ACTION_BASE;
• aParams.push(oParam); oSoapCall.encode(0, sMethod, SOAP_ACTION_BASE, 0,
null,
• aParams.length, aParams);
• document.getElementById("txtRequest").value =
• oSerializer.serializeToString(oSoapCall.envelope);
• setUIEnabled(false);
•
• oSoapCall.asyncInvoke(
• function (oResponse, oCall, iError)
• {
• var oResult = handleResponse(oResponse, oCall,
iError);
```

- `showSoapResults(oResult);`
- `}`
- `);`
- `}`

• `asyncInvoke()` 方法只接受一个参数：回调函数 `handleResponse()`。SOAP 调用向该函数传入三个参数：一个 `SOAPResponse` 对象、原始 SOAP 调用的指针（用来跟踪多个实例），以及错误代码。当调用返回时，这些值都传入 `handleResponse` 函数进行处理：

- `function handleResponse(oResponse, oCall, iError)`
- `{`
- `setEnabled(true);`
- `if (iError != 0)`
- `{`
- `alert("Unrecognized error.");`
- `return false;`
- `}`
- `else`
- `{`
- `var oSerializer = new XMLSerializer();`
- `document.getElementById("txtResponse").value =`
- `oSerializer.serializeToString(oResponse.envelope);`
- `var oFault = oResponse.fault;`
- `if (oFault != null)`
- `{`
- `var sName = oFault.faultCode;`
- `var sSummary = oFault.faultString;`
- `alert("An error occurred:\n" + sSummary`
- `+ "\n" + sName`
- `+ "\nSee message pane for SOAP fault");`



- return false;
- }
- else
- {
- return oResponse;
- }
- }
- }

- 如果错误代码不为 0，那么就表示发生未知错误。这种情况非常少见，大多数情况下通过响应对象的 `fault` 属性返回错误。

- 现在用到另一个内建类 `XMLSerializer`。该类获取一个 XML 节点，并将其转换成字符串或流。本示例中得到一个字符串，并显示在右边的文本区域中。

- 如果 `oResponse.fault` 不为 `null`，那么就表示发生 SOAP 错误，所以生成错误信息，显示给用户并停止后续的操作。如果调用成功，那么将响应对象作为函数的返回值，并由 `showSoapResults()` 函数显示结果：

- function showSoapResults(oResult)
- {
- 
- if (!oResult) return;
- document.getElementById("txtResult").value =
- oResult.body.firstChild.firstChild.firstChild.data;
- }

- 在检查 `oResult` 是否有效后，使用 DOM 得到 `<methodResult>` 元素的值。

- `SoapResponse` 类中有一个 `getParameters` 的方法，理论上，可以使用该方法以更优雅的方式获取参数。但是，该方法在基于文档模式的调用下无效。于是就需要使用更基本方法检查 `soap:Body` 的结构。

#### 6.5.4 通用方法

- 对于大多数浏览器来说，使用 Web 服务的唯一方法是使用 XMLHttp。因为 IE、Firefox、Opera 和 Safari 都对 XMLHttp 提供基本的支持，所以这也是浏览器通用的最佳选择。不幸的是，为了保持跨浏览器的一致性，要付出的代价不小。需要手动创建 SOAP 请求并将其发送到服务器，还要自己解析返回结果并观察错误。

- 这其中的两大主角是，微软提供的 XmlHttpRequest 类和基于 Mozilla 浏览器提供的 XMLHttpRequest 类。它们都包含类似的方法和属性，尽管微软 XmlHttpRequest 类的实现在标准指定之前，而后来发布的 Mozilla 版本则更多基于 W3C 标准。这些类的本质都是允许一个 HTTP 请求访问一个 Web 地址。目的地址并不是必须返回 XML（事实上可以获取任何内容）并且可以包含传送数据。对于 SOAP 调用，一般的方法是，以原始<soap:Envelope>作为负载发送 POST 请求。

- 下面的例子再一次调用了测试工具。这时，可以使用 zXml 库来创建 XMLHttp 对象，并自行创建完整的 SOAP 调用。该库使用一系列技术来检查当前所使用的浏览器是否支持 XML 类。zXml 库封装这些 XML 类，并添加了一些额外的方法和属性，这样它们就可以以几乎一致的方式使用。调用 getRequest() 函数可以生成 SOAP 请求字符串：

- function getRequest(sMethod, sOp1, sOp2)
- {
- var sRequest = "<soap:Envelope xmlns:xsi=\""
- + "http://www.w3.org/2001/XMLSchema-instance\" "
- + "xmlns:xsd=\""http://www.w3.org/2001/XMLSchema\" "
- + "xmlns:soap=\""
- <http://schemas.xmlsoap.org/soap/envelope/>">\n"
- + "<soap:Body>\n"
- + "<" + sMethod + "xmlns=\"" + SOAP\_ACTION\_BASE + "\">\n"
- + "<op1>" + sOp1 + "</op1>\n"
- + "<op2>" + sOp2 + "</op2>\n"
- + "</" + sMethod + ">\n"
- + "</soap:Body>\n"
- + "</soap:Envelope>\n";

- return sRequest;
- }

• `getRequest()` 函数相当的简单易懂。该函数只是简单地根据正确的格式构造 SOAP 字符串（正确的格式可以使用在 Math 服务创建描述中的 .NET 测试工具看到）。  
`getRequest()` 函数返回完整的 SOAP 字符串，该字符串通过 `performSpecificOperation()` 函数用于创建 SOAP 请求：

- function performSpecificOperation(sMethod, sOp1, sOp2) {
- oXmlHttp = zXmlHttp.createRequest();
- setUIEnabled(false);
- var sRequest = getRequest(sMethod, sOp1, sOp2);
- var sSoapActionHeader = SOAP\_ACTION\_BASE + "/" + sMethod;
- oXmlHttp.open("POST", SERVICE\_URL, true);
- oXmlHttp.onreadystatechange = handleResponse;
- oXmlHttp.setRequestHeader("Content-Type", "text/xml");
- oXmlHttp.setRequestHeader("SOAPAction", sSoapActionHeader);
- oXmlHttp.send(sRequest);
- document.getElementById("txtRequest").value = sRequest;
- }

• 首先，调用 `zXmlHttp` 库的 `createRequest()` 方法来创建 `XMLHttpRequest` 请求。如前所描述，这是一个 `ActiveX` 类（基于 IE 浏览器）或 `XMLHttpRequest`（基于 Mozilla 浏览器）的实例。该对象的 `open` 方法尝试对请求进行初始化。第一个参数表示该请求是一个包含数据的 POST 请求，第二个参数表示服务的 URL 地址，第三个参数是一个布尔型参数，表示该请求是否为异步模式或该代码是否在调用后应该等待响应。

- 请求的 `onreadystatechange` 属性表示当请求的状态改变时所调用的函数。

• `performSpecificOperation()` 函数向 HTML 请求添加两个标题。第一个标题表示内容类型是 `text/xml`；第二个标题则添加 `SOAPAction` 标题。该值可以从 .NET 测试工具页面阅读，或者查看 WSDL 文件中相关的 `<soap: operation/>` 元素的 `soapAction` 属性。一旦发送了请求，原始的 XML 将显示在左侧的文本框中。当请求的状态改变时，那么会调用 `handleResponse()` 函数：

```

• function handleResponse()
• {
• if (oXmlHttp.readyState == 4)
• {
• setUIEnabled(true);
• var oResponseOutput = document.getElementById("txtResponse");
• var oResultOutput = document.getElementById("txtResult");
• var oXmlResponse = oXmlHttp.responseXML;
• var sHeaders = oXmlHttp.getAllResponseHeaders();
• if (oXmlHttp.status != 200 || !oXmlResponse.xml)
• {
• alert("Error accessing Web service.\n"
• + oXmlHttp.statusText
• + "\nSee response pane for further details.");
• var sResponse = (oXmlResponse.xml ? oXmlResponse.xml :
oXmlResponseText);
• oResponseOutput.value = sHeaders + sResponse;
• return;
• }
• oResponseOutput.value = sHeaders + oXmlResponse.xml;
• var sResult =
•
• oXmlResponse.documentElement.firstChild.firstChild.firstChild.firstChild.data;
• oResultOutput.value = sResult;
• }
• }

```

• handleResponse()函数负责回应请求中状态的变化。当 readyState 属性值等于 4，那么就不再需要其他处理，只需要检查是否返回有效的结果。

- 如果 `oXmlHttp.status` 不等于 200，或者 `responseXML` 属性为空，那么就会产生错误并向用户显示错误消息。如果该错误是 SOAP 错误，那么同样在消息框中显示错误消息。如果不是 SOAP 错误，那么就显示 `responseText` 的值。如果调用成功，那么就将原始的 XML 显示在右侧文本框中。

- 假设 XML 响应有效，那么通过以下方法可以获取实际的结果：使用 XSLT、使用 DOM 或者文本解析。不幸的是，这些方法并不适用于所有浏览器。虽然遍历 DOM 树的方法效率不高，但是该方法的优点在于对各种 XML 文档都适用。

## • 6.6 跨域的 Web 服务

- 到目前为止，所调用的 Web 服务与所访问的页面处于同一域。这样就避免了跨域脚本的问题（例如跨站点脚本或者 XSS）。如本书前面所讨论的，该问题的原因在于，调用外部网站具有安全风险。如果 Web 服务与调用页面处于同一域，浏览器就允许 SOAP 请求，但是如果调用 Google 或者 Amazon.com 的 Web 服务呢？

- 如果要调用外部 Web 服务，那么就需要使用服务器端代理(server-side proxy)，该代理运行在 Web 服务器上并为客户端执行调用。代理将从 Web 服务接收到的信息返回到客户端。这样的设置就如同 Web 代理服务器，将不同的网络连接起来。在这种模型中，所有请求都发送到中心服务器，中心服务器获取网页，并将页面发送给各个请求者。

### • 6.6.1 Google Web API 服务




- Google 通过 Web 服务提供许多可以调用的方法，这些服务包括获取页面缓存副本的方法，根据给定的查询条件获取更多明显的结果。用来证明服务器端代理的方法是 `doSpellingSuggestion`，该方法包含一个条件并返回 Google 认为的结果。如果条件中的短语拼写错误或者太模糊，那么就返回一个空的字符串。

- Google Web API 服务的开发工具可以从 [www.google.com/apis/index.html](http://www.google.com/apis/index.html) 处下载。该开发工具包含 Web 服务的 SOAP 和 WSDL 标准，以及 C#、Visual Basic .NET 和 Java 的示例。

- 访问该 Web 服务还必须在 Google 注册并获取安全码。关于开发研究和商业应用还有相关的规则。

## • 6.6.2 创建代理

• 下载完 Google 文档和获得安全码以后，需要在本地服务器上创建一个服务，该服务接收用户的调用，并将这些调用转向 Google。基本服务的创建方法与之前介绍的 Math 服务的创建方法一样。

• 首先打开 IIS 管理工具（开始  管理工具  Internet 信息服务）。展开左侧的树结构，在默认 Web 网站节点上点击右键，选择新建  虚拟目录。在虚拟目录的创建向导中的别名字段处填入新的目录名 GoogleProxy，并点击下一步。下一个界面中，浏览标准的 IIS 目录 C:\InetPub\wwwroot，并创建一个新的文件夹，同样命名为 GoogleProxy。向导的余下界面都选择默认项，并使用 Windows 资源管理器在 GoogleProxy 目录下创建新的文件夹 bin。

• 接下来，打开文本编辑器并创建如下文件：

- <%@ WebService Language=" c#" %>
- Codebehind=" GoogleProxy.asmx.cs" Class="

Wrox.Services.GoogleProxyService" %>

• 将该文件命名为 GoogleProxy.asmx 并保存在 GoogleProxy 目录中。

• 现在创建最主要的文件 GoogleProxy.asmx.cs：

- using System;
- using System.Web;
- using System.Web.Services;
- using GoogleService;
- 
- namespace Wrox.Services
- {
- [WebService (Description = "Enables calls to the Google API",
- Namespace = "<http://www.wrox.com/services/googleProxy>")]
- public class GoogleProxyService : System.Web.Services.WebService
- {
- readonly string GoogleKey = "EwVqJPJQFHL4i nHoIQMEP9j ExTpcf/KG";

- 
- [WebMethod(
- Description = "Returns Google spelling suggestion for a given phrase." )]

```

 • public string doSpellingSuggestion(string Phrase)
 • {
 • GoogleSearchService s = new GoogleSearchService();
 • s.Url = "http://api.google.com/search/beta2";
 • string suggestion = "";
 • try
 • {
 • suggestion = s.doSpellingSuggestion(GoogleKey, Phrase);
 • }
 • catch(Exception Ex)
 • {
 • throw Ex;
 • }
 • if (suggestion == null) suggestion = "No suggestion found.";
 • return suggestion;
 • }
 • }
 • }

```

- 请记住在 GoogleKey 变量中输入 Google 安全码的值,并将该文件保存在相同的位置。

- 这段代码本身相同简单易懂。GoogleSearchService 完成所有实际工作。

doSpelling- Suggestion 方法创建 GoogleSearchService 类的一个实例。接着设置该服务的 URL。尽管不一定需要该步骤,但是我们发现这有助于 Web 服务 URL 的修改。

在实际环境中, URL 可以从配置文件中获取,这样可以容易地在服务器间切换。

- 现在调用 doSpellingSuggestion 方法，传入 Google 安全码和 Phrase 参数。这是使用服务器端代理的又一个好处：将敏感信息与客户端的浏览器隔离开，例如 Google 安全码。

- 如果抛出异常，那么会被再次抛出并以 SOAP 错误的形式返回。如果返回的 suggestion 为空值 null，那么返回一个合适的字符串；否则，直接将 suggestion 传回客户端。

- 现在创建了一个与 Google 交互的类。将 GoogleSearch.wsdl 文件拷贝到 GoogleProxy 目录下，接着打开命令提示符，转到 GoogleProxy 目录，并运行如下命令（可以忽略关于 schema 遗漏的警告）：

- WSDL /namespace:GoogleService /out:GoogleSearchService.cs  
GoogleSearch.wsdl

- WSDL 工具读取 GoogleSearch.wsdl 文件，并创建与该服务通信的类的源代码。如 WSDL 命名的第一个参数，该类位于 GoogleService 命名空间内。该源代码需要通过使用 C#编译器编译成一个 DLL 文件。在命令提示符下输入以下命令，或者运行名为 MakeGoogleServiceDLL.bat 的批处理文件。

- WSDL.exe 可能安装在不同目录下，这取决于机器上微软的其他组件。在安装 Visual Studio .NET 的机器上，该文件可能位于 C:\Program Files\Microsoft Visual Studio.NET 2003\SDK\v1.1\Bin 目录，不过需要在机器上查找该文件的位置。

- C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\csc.exe /r:System.dll  
/r:System.Web.dll /r:System.Web.Services.dll /t:library  
/out:bin\GoogleSearchService.dll GoogleSearchService.cs  
/r: 参数通知编译器为目标 DLL 文件 GoogleSearchService 提供支持类需要哪些 DLL 文件。

- 最后一步是编译 GoogleProxy 类本身：

- C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\csc.exe /r:System.dll  
/r:System.Web.dll /r:System.Web.Services.dll  
/r:bin\GoogleSearchService.dll  
/t:library /out:bin\GoogleProxy.dll GoogleProxy.asmx.cs



- 注意，该命令需要引用刚创建的 GoogleSearchService.dll 文件。
- 现在可以在浏览器中输入下列 URL 测试 Web 服务了：
- <http://localhost/GoogleProxy/GoogleProxy.asmx>
- 这时可以看到标准的欢迎界面，如图 6-8 所示。



• 图 6-8

- 点击 doSpellingSuggestion 链接，可以使用内建的测试工具测试方法，如图 6-9 所示。



• 图 6-9

- 点击 Invoke 按钮，可以看到返回的 XML，如图 6-10 所示。



• 图 6-10

## • 6.7 小结

- 本章介绍了 Web 服务的概念：一种基于因特网的数据交换架构，同时也介绍了 Web 服务的演化以及相关技术，例如 SOAP、WSDL 和 REST，并讨论了 SOAP 和 REST 服务之间的异同。

- 接着，阐述了使用 ASP.NET 和 C# 如何创建 Web 服务。这部分包括了 .NET SDK 的下载，内建 Web 服务创建和管理工具的使用。你还可了解如何使用所生成的 .NET 测试工具检查并测试 Web 服务。

- 接下来，分别为 IE 和 Mozilla 创建了测试工具，使用不同的技术来调用 Web 服务。介绍了 IE 的 Web 服务行为和 Mozilla 的高级 SOAP 类，最后将创建的测试工具改成跨浏览器通用的方法，使用 XMLHttp 发送和接收 SOAP 消息。

- 最后，讨论了 Web 服务的跨域问题，以及如何使用服务器端代理来避免这样的问题。

- 在本章中，客户端和服务器端之间使用 SOAP 规约进行参数传递。尽管这是一种健壮灵活的方法，但是这种方法在整个过程中添加许多工作，使得客户端还必须能够处理 XML 文档，这带来了一定的复杂性。下一章将展示一种更简单的、非正式的数据传递方法 JSON (JavaScript Object Notation)。