

第1章 一种新的 Web 设计方法

1.1 为什么需要 Ajax 富客户端？

建造一个富客户端[2]毫无疑问要比设计一个网页复杂。付出这些额外的努力，动机何在？需要付出什么代价？而且……等一下，富客户端到底是什么？

富客户端的两个要点是：第一，它是“富”的；第二，它是“客户端”。

这好像是一句废话，别急，待我稍作解释。“富”是指客户端的交互模型，要有多样化的输入方式和符合直觉的及时反馈手段。说简单点儿，一个“富”的应用使用起来应该像是在使用现在的桌面应用一样，例如，就像是使用字处理软件（Word）或电子表格软件（Excel）。接下来，我们有必要仔细地考察一下所要涉及的各个方面。

1.1.1 比较用户体验

花几分钟使用一下你选中的应用（浏览器除外），记下它用到了哪些用户交互，然后马上回来。为了简短起见，我举一个电子表格的例子，但是，这里所涉及的要点是通用的，足以针对文本编辑器上的各种情形。

好，我们开始。先在电子表格中随便输入几个等式，注意到，可以以几种方式进行交互：编辑数据，用键盘和鼠标浏览数据，还可以使用鼠标拖拽来重新组织数据。

我做这些操作的时候，程序给了我反馈。移动鼠标的时候，光标改变了形状；当鼠标停在上面的时候，按钮变亮了；选中的文字也改变了颜色。窗口或者对话框被选中的时候，也和平常显得不一样了，等等（图 1-1）。这些就是所谓“富”的交互。当然了，仍然有一些有待改进的地方，但这是一个好的开始。

OK，电子表格就是一个富客户端程序了吗？当然不是。

在电子表格或者类似的桌面应用中，业务逻辑和数据模型是在一个封闭的环境中运行的。在这个环境中，它们彼此清晰地了解对方，并且可以互相访问，而环境之外的东西，对于它们来说是未知的（图 1-2）。那么客户端又是什么呢？它是与另一个独立的进程相互通信的程序，后者通常运行在服务器上。一般来说，服务器总是要比客户端大一些，能力强一些，配置更好一些，因为在服务器上通常

要存储浩如烟海的信息。客户端程序使得最终用户可以查看和修改这些信息，当多个客户端连接在同一个服务器上的时候，可以在它们之间共享这些信息。图 1-3 展示了一个简单的客户/服务器架构。

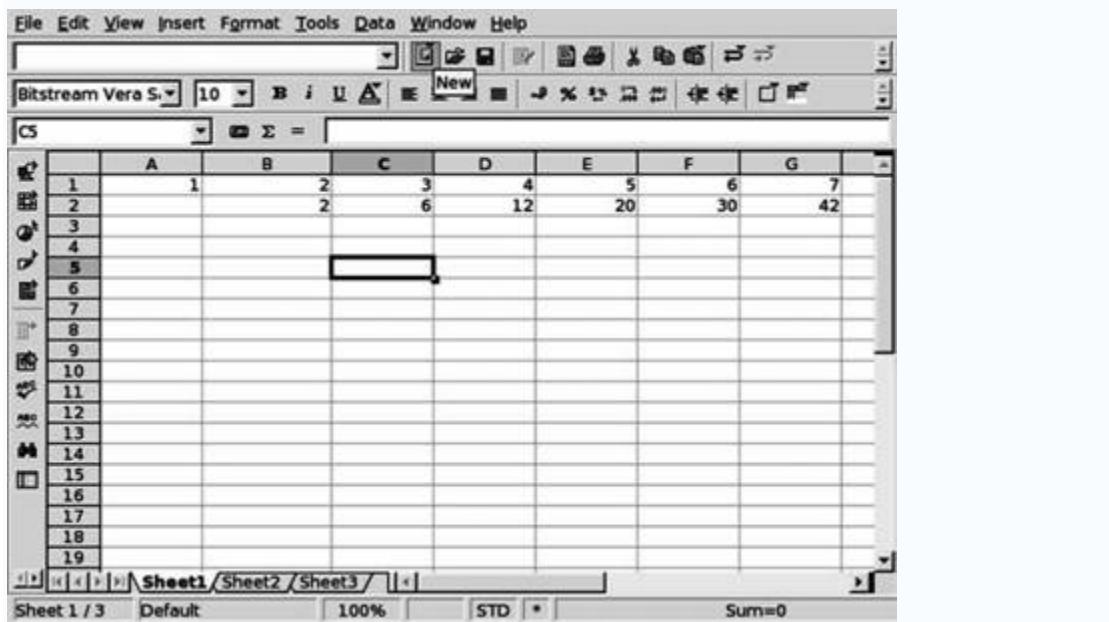


图 1-1 这个桌面电子表格应用展现了用户交互的众多可能性。被选中单元格的行列标题都是突出显示的；按钮在鼠标移上去的时候会显示提示信息；工具栏上排列着各种丰富的控件；单元格也可以交互地查看和编辑

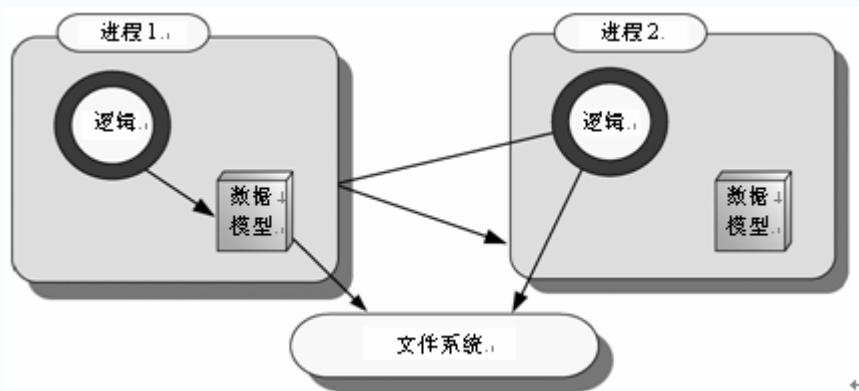


图 1-2 一个独立的桌面应用的架构示意图。应用运行在它自己的独立进程中，数据模型和程序逻辑能够彼此“看到”对方的存在。除了通过文件系统之外，同一个应用的第二个运行实例[3]没有办法访问到第一个运行实例的数据模型。通常来说，整个程序的状态都保存在单个文件中，当应用运行的时候，这个文件会被加锁以阻止其他的进程同时访问

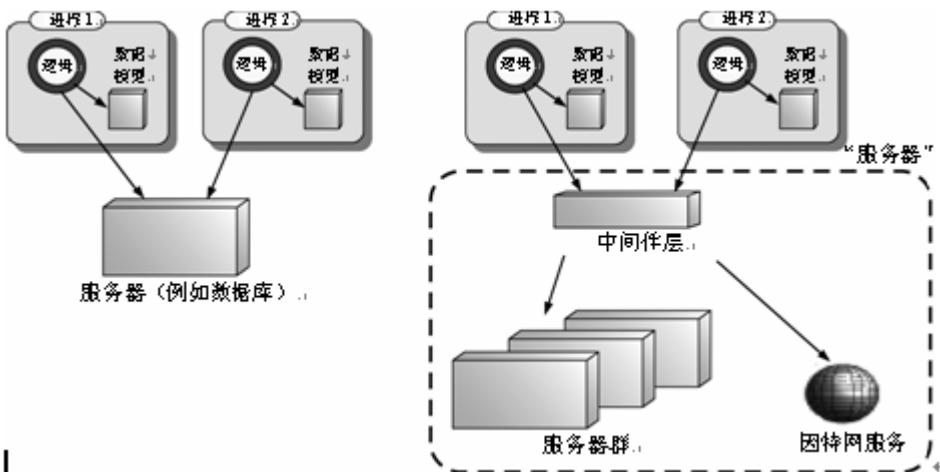


图 1-3 客户/服务器系统和 n 层架构的示意图。服务器提供了共享的数据模型，客户端与该数据模型交互。客户端同时还维护数据模型的一部分，以获得快速的访问，但是它将服务器端的模型当作业务领域对象的最终表示。多个客户端可以与同一个服务器交互，当然，这需要有合适的资源锁定机制和合理的对象（或者数据行）隔离措施作为保证。服务器可以是单进程的，就像在 20 世纪 90 年代早期和中期传统的客户/服务器模型中一样，也可以是由很多个中间件层或者多个 Web 服务等组成。在任何一种情况下，从客户端的角度来看，服务器都有一个单独的接入点，可以看作是一个黑盒

在现代的 n 层架构中，服务器往往要和更远的后端服务器（例如数据库）通信，因此被称作“中间件”的层同时扮演着客户端和服务器的角色。我们的 Ajax 应用位于这个链的一端，它仅仅是作为客户端，因此为讨论方便，我们可以把整个 n 层系统看作是一个标记为“服务器”的黑盒。

我的电子表格应用只需要管理它自己保存在内存或本地文件系统中的少量数据。如果架构设计良好的话，数据和它的表现形式的耦合可以非常松散，但是我不能通过网络来分割或者通过网络来共享它们。从这个意义上来说，电子表格应用不是一个客户端。

与之相对应的 Web 浏览器就是一个典型的客户端，它与 Web 服务器通信，请求需要的页面。浏览器有丰富的功能，用来管理用户的浏览行为，常见功能有回退按钮、历史列表和分页浏览多个文档等等。但是当我们把特定网站的 Web 页面看作是一个应用时，这些通用的浏览功能实际上和应用关系不大，充其量也就如电子表格和 Windows 的开始按钮或者窗口列表之间的关系。

我们来考察一下现代的Web应用。为了简单起见，我们选择了“地球人都知道”的在线书店Amazon.com（图1-4）。在浏览器中打开Amazon网站，因为在此之前我访问过，它会给我显示一个友好的问候、一些推荐书目，还有我的购买历史信息。

点击推荐书目中的任何一条，就会转到另外一个页面（此时，页面要刷新一下，在这几秒钟内我什么也看不到）。新页面是该书的相关信息：书评、二手书报价、同一作者的其他著作，以及以前我浏览过的其他书籍（图1-5）。

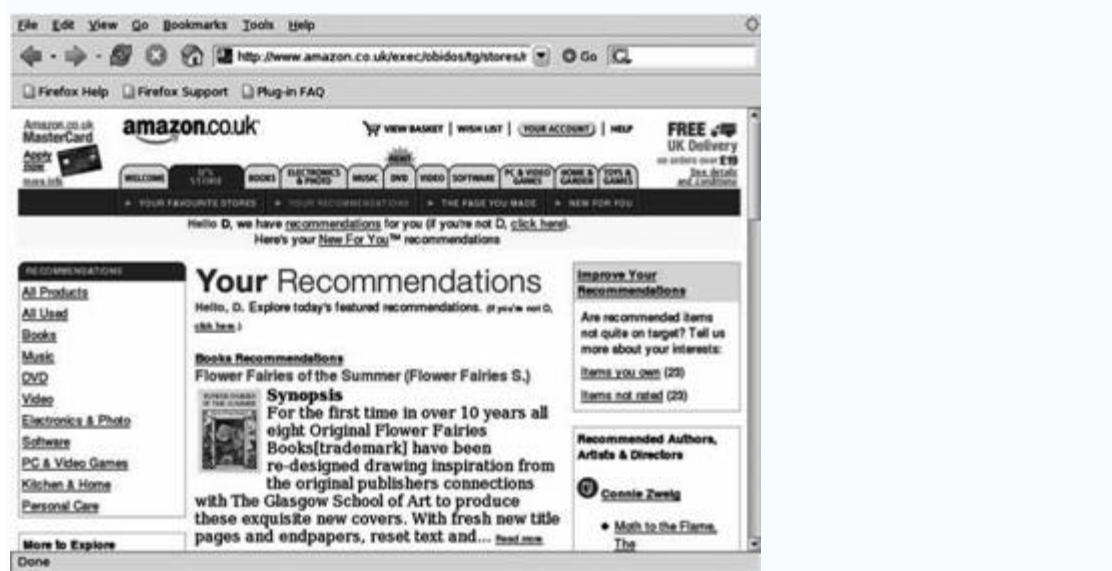


图1-4 Amazon.com的首页。系统记得我上一次的访问。导航链接除了通用模板文件，还有个性化信息

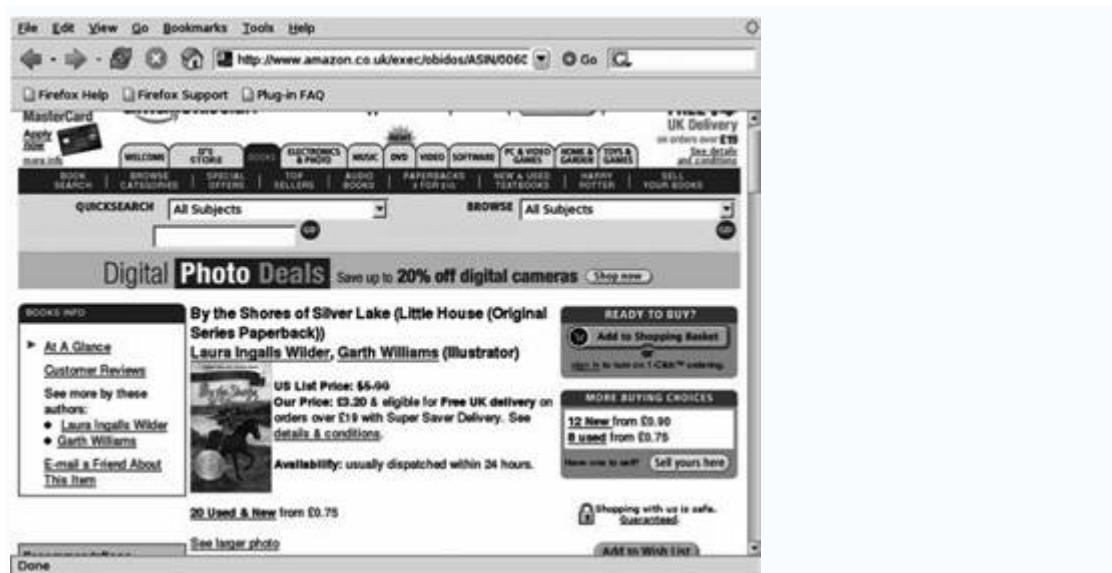


图 1-5 Amazon.com 书籍详细信息的页面。包括一大堆通用信息和个性化信息的超链接。

虽说其中的大量内容和图 1-4 中的一模一样，但是由于 Web 浏览器使用基于文档的操作，每次发送新页面都必须要重新发送这些内容

简而言之，呈现在我面前的是非常丰富的、关联度很高的信息。但是对我而言，交互的方式就是点击那些超链接，然后填写一些表格。假设我在键盘前面不小心睡着了，第二天才醒来，如果不刷新页面，我就没法知道《哈里·波特》系列的新书已经出版了，也不能将我的列表从一个页面带到另一个页面，我要是想同时看到更多一些东西也不行，因为我无法改变页面上局部内容区域的大小。

我似乎是在批评 Amazon 的界面，其实并非如此，我只是拿它来做个例子。事实上，在传统 Web 开发方式的桎梏下，他们已经做得非常棒了。但是比起电子表格来说，它所用的交互模型毫无疑问是太有限了。

为何现代的 Web 应用仍然有这么多的局限呢？造成目前的状况有一些合理的技术原因，我们现在就来考察一下。

1.1.2 网络延迟

因特网的宏伟蓝图是将这个世界上所有的计算机都连接起来，形成一个无比巨大的计算资源。如果能把本地调用和远程调用等同起来，那么无论是分析蛋白质的成分还是破解外太空的信号，使用者都无需考虑机器的物理位置，剩下来的只有愉快地计算。

但是非常不幸，本地调用和远程调用是完全不同的东西。在现有的技术水平之下，网络通信仍然是一件代价高昂的事情（也就是说，通常很慢，而且并不可靠）。在没有网络调用的情况下，不同的方法和函数以及它们所操作的数据都位于相同的本地内存中（图 1-6），向方法内传递数据并且获得方法的返回结果是非常直接的。

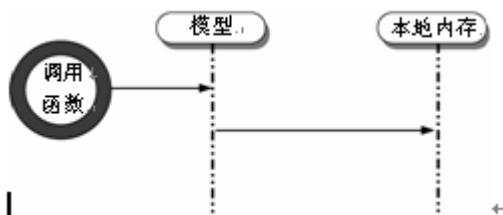


图 1-6 本地过程调用的顺序图。参与者很少，因为程序逻辑与数

据模型都保存在本地内存中，并且彼此可以直接访问

而在有远程调用的情况下，位于网络两端的通信双方为了发送和接收数据在底层需要进行大量计算（图 1-7）。比起数据在线路上的往返，这些计算需要消耗更多的时间。传输一段二进制的数据，中间要经过很多环节的编码和解码、错误校验、失败重发、数据包拆分和重组，数据最终转化为 0 和 1 表示的二进制信号，通过线路（或者无线连接）到达另外一方。

在本地，调用函数的调用请求被编码为一个对象，然后将这个对象序列化为一系列字节，最后使用应用层协议（通常是 HTTP）通过物理传输介质（例如铜缆、光纤或者无线电波）将其发送出去。

在远程机器上，对应用层协议解码，将获得的数据字节反序列化，创建一个请求对象的副本。然后对数据模型应用这个对象并生成一个响应对象。为了将响应对象传递给本地的调用函数，所有的序列化、反序列化以及传输层的操作都要反向再来一次。最后，响应对象被传递给本地的调用函数。

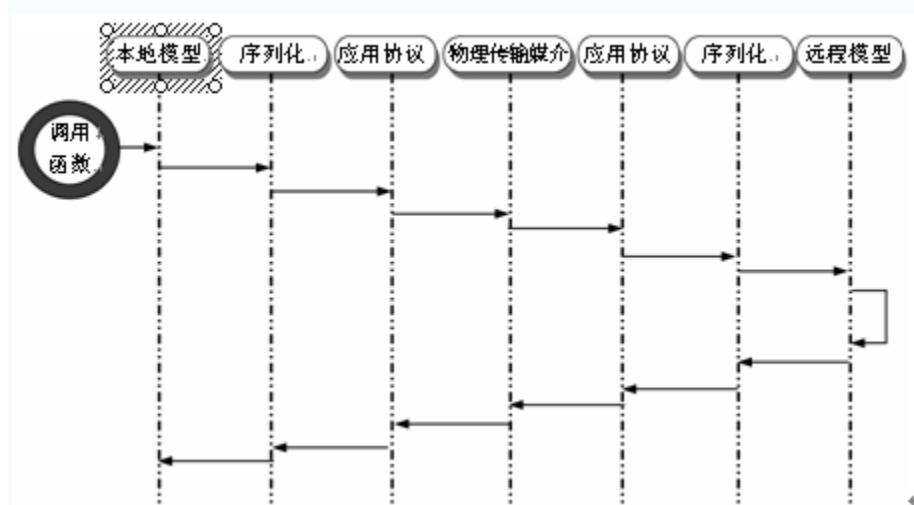


图 1-7 远程过程调用的顺序图。一台机器的程序逻辑尝试操作另外一台机器上的数据模型

这个交互过程很复杂吧，幸好，它是可以自动完成的。现代的编程环境如 Java 和 Microsoft 的.NET 框架都内置了这个能力。尽管如此，执行远程调用时，上述所有这些操作仍然会在内部执行。如果我们到处使用远程调用，性能势必会大受影响。

这也就是说，远程调用是不可能和本地调用一样有效率的。更糟糕的是，网络的不稳定更让这种效率损失捉摸不定，难以预计。相比之下，运行在本地内存之中的本地调用，在这一点上无疑要有优势得多。

等等，说了半天的远程调用，这和软件的可用性有关系吗？答案是，大有关系。

一个成功的计算机用户界面要能以最起码的水平模拟我们在真实世界中的体验。交互的基本规则之一就是，当我们推一下、刺一下或者捅一下某个东西的时候，它立刻就会响应。响应的时间只要稍微拖长一点，就会使人困惑，分散其注意力，把关注点从手头的任务转移到用户界面上。

远程调用横穿整个网络，需要执行大量的额外操作，它们往往会把系统拖慢，使用户察觉到延迟。在桌面应用中，只有当可用性设计做得非常糟糕的时候，才会出现这种令用户感觉充满 bug、反应迟钝的用户界面，但在网络应用中，什么都不做就能得到大量这样的界面。

因为网络延迟不可预测，这类界面问题往往都神出鬼没，对应用响应的测试也难以开展。换句话说，网络延迟是导致实际应用的交互性糟糕的一个普遍原因。

1.1.3 异步交互

用户界面的开发者对于网络延迟只能做最坏的假设。简单地说，就是要尽可能让用户界面与网络活动无关。天才的程序员们早已发明了一种确实有效而且久经考验的方案，来专门解决这一问题。先卖个关子，让我们到现实世界中走一趟。

在我每天早上必做的事中，很重要的一项是叫醒我的孩子去上学。我可以站在床边把他们折腾醒，催着他们起床穿衣，但这是一种很耗费时间的方法，总要耗费我很多宝贵的早间时光（图 1-8）。

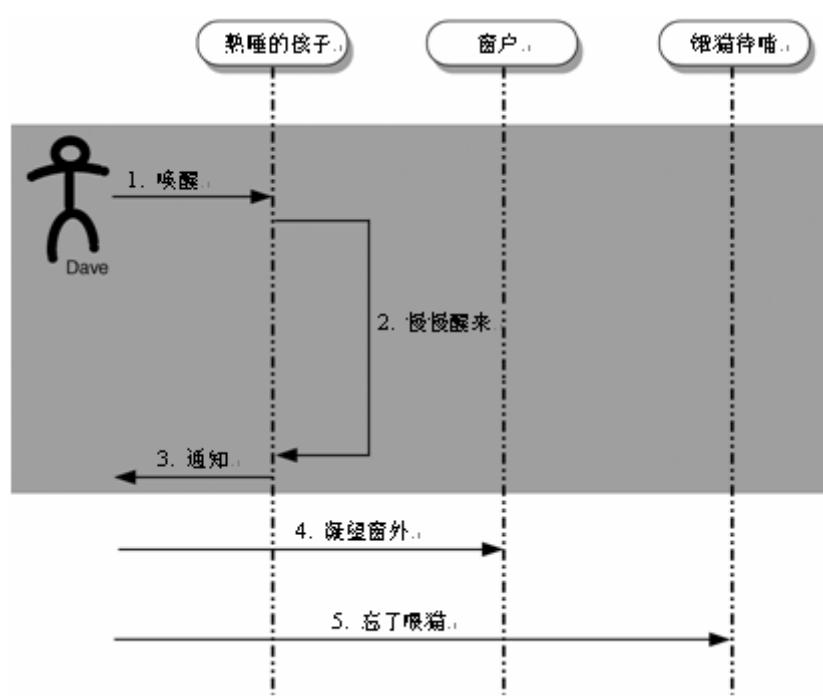


图 1-8 在我每日早晨必做的事中，以同步方式响应用户输入的顺序图。顺序图中纵向表示时间的流逝，其中的阴影区域表示了我被阻塞不能接受其他输入的时间长度

我要叫醒孩子，看看窗外，往往会忽略了喂猫。孩子们起来之后会问我要早餐。就像服务器端的进程一样，孩子们起床总是慢吞吞的。如果我遵循同步交互模式，就要等他们老半天。不过，只要他们嘟囔一句“我醒了”，我就可以先去干其他的事，需要时再回来看看他们。

按照计算机的术语，我需要做的就是为每个孩子在一个单独的线程中建立一个异步进程。开始之后，孩子们会在他们的线程里自己起床，我这个父线程没有必要同步傻等，他们完事后会通知我（往往还会问我想要吃的）。在他们醒来的过程中，我并不需要和他们交互，就当他们已经起来并自己穿好衣服了，因为我有理由相信他们很快会这么做的（图 1-9）。

对于任何用户界面来说，这是一种沿用已久的实践，即创建异步的线程，让它在后台处理那些需要计算很久的任务，这样用户可以继续做其他的事情。当启动这个线程的时候，有必要阻塞用户的操作[4]，但是在可接受的很短时间之后，阻塞就会解除。因为存在网络延迟，使用异步方式来处理任何耗时的远程调用是一种很好的实践。

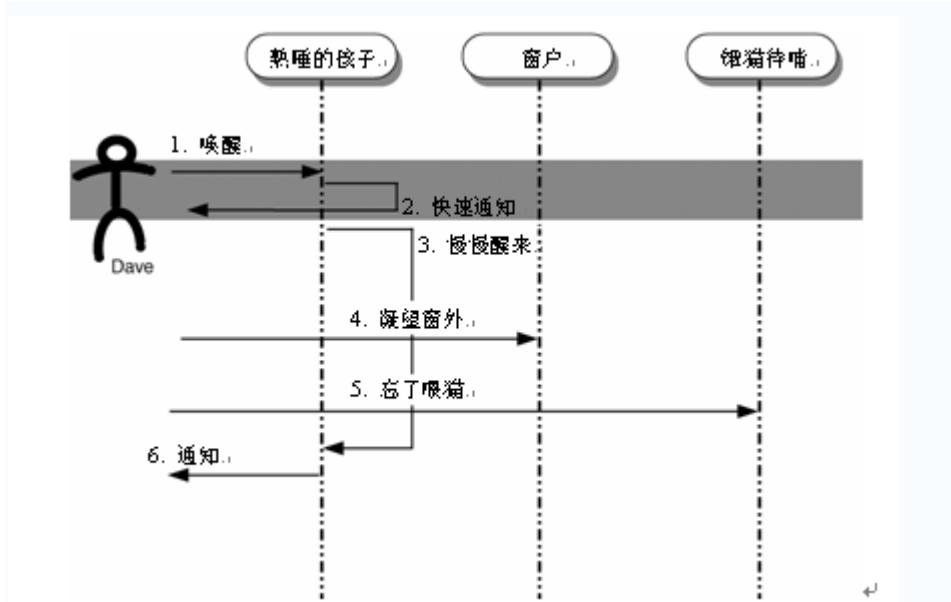


图 1-9 以异步方式响应用户输入的顺序图。如果遵循异步的输入模式，我可以让孩子们在醒来的时候通知我。在他们缓慢的起床过程中，我可以继续从事其他活动，这使得我被阻塞的时间大大缩短

实际上，网络延迟问题和相关的解决方案由来已久。在老的客户/服务器模式中，当设计不佳的客户端程序碰上了高负载的服务器时，用户界面就会出现让人难以忍受的延迟。即便是在如今的因特网时代，当切换页面时，如果浏览器半天出不来东西，那么这种糟糕的情况很可能就是因为网络延迟造成的。在现有技术条件下，我们暂时还没有办法消除网络延迟，但是至少有一个对策，那就是采用异步方式处理远程调用，不是吗？

糟糕的是，对于 Web 开发人员而言这样做存在一个难点：HTTP 协议是一个“请求—响应”模式的协议。也就是说，客户端请求一个文档，服务器要么返回这个文档，要么告诉客户端找不着文档或者让客户端去另外一个地方找，还可以告诉客户端可以使用它的本地缓存，诸如此类。总而言之，“请求—响应”模式的协议是一种单向的通信协议。客户端可以向服务器发起连接，但是服务器不可以向客户端发起连接。甚至，当客户端下次发起通信请求时，健忘的服务器都记不起来这个客户端是谁了（HTTP 是无连接的）。

多数 Web 开发者使用现代的编程语言，例如，Java、PHP 或者.NET，他们熟悉用户会话（user session）的概念，这其实是应用服务器对于不能保持连接状态的 HTTP 协议的一种补救措施。HTTP 就其最初的设计目的来说表现得非常好，采用一些巧妙的处理，它能够适应设计之初没有考虑的场合。但是我们的这个异步回调方案中的关键特征是，客户端会收到两次通知，一次是在线程创建的时候，另一次是在线程结束的时候。标准的 HTTP 和传统 Web 应用模型可不会提供这些。

像Amazon那样的传统Web应用，是建造在页面概念之上的。给用户显示一个文档，上面包括各种链接和表单，用户可进一步访问更多的文档。这种交互模式可以在很大的规模上支持复杂的数据集（就像Amazon和其他网站已经证明的那样），它所提供的用户体验也足以满足开展业务的需要。

十年来，这种交互模式在我们对因特网商业应用的看法上打下了深深的烙印。界面友好的所见即所得（WYSIWYG）Web制作工具使得站点更容易被理解为一堆页面。服务器端的Web框架使用状态图来对页面的转换建模。没有引入，传统的Web应用就这么一直牢牢地束缚在页面刷新操作之上，就好像这种刷新是理所当然而且无可避免的，从没有尝试过任何异步的处理方案。

当然，毫无疑问，传统的Web应用肯定不是一无是处的。毕竟Amazon在这种交互模式上创造了成功的商业应用。但是这种适用于Amazon的方式并不一定适用于所有的人。为什么这么说呢？要理解这一点，我们需要考察用户的使用模式（usage pattern）。

1.1.4 独占或瞬态的使用模式

泛泛地讨论自行车和 SUV（运动型轿车）孰优孰劣毫无意义。因为它们各自都有优点和缺点——舒适度、速度、油耗或者个人身份的象征等等。只有在特定的使用模式下讨论，这样的比较才有意义。例如，是要在上下班高峰时段穿越市中心，还是要带上一家老小去度假，或者只是要找个躲雨的地方。在类似这样的具体情况下，我们才能有的放矢地比较。对于用户界面，亦复如是。

软件可用性专家 Alan Cooper 写了很多有关使用模式的好文章，他定义了两种主要的使用方式：瞬态的（transient）和独占的（sovereign）[5]。瞬态应用可能每天都会偶尔使用一下，但是总是作为次要的活动，突发性地用上一会儿。与之形成鲜明对比的是独占应用，独占应用需要应付用户每天几个小时的持续使用。

很多应用天生就具备独占或者瞬态的性质。例如，写作用的字处理软件就是一个独占应用，可能还会用到几个瞬态的应用，比如文件管理（文件的开启和关闭窗口中常常会嵌入这个功能）、字典或者拼写检查工具（很多字处理程序也嵌入这些功能），还有与同事联络的电子邮件和聊天工具。而对于软件开发者，文本编辑器、调试器或者 IDE（集成开发环境）则是独占的。

独占应用常常使用得更频繁。要知道，一个良好的用户界面应该是不可见的。衡量使用频繁程度的一个好的标尺，是当这个用户界面发生明显的停顿时，它对于用户流程的影响。例如，从一个目录向另一个目录移动文件要发生 2 秒钟的停顿，我能愉快地接受；可是如果这两秒是发生在我正饱含激情地用绘画软件创作一幅作品时，或者是我正努力地调试一段很难对付的代码时，这肯定会让让我感觉十分不爽。

Amazon 是一个瞬态应用，eBay、Google 以及大多数大型的公众 Web 应用都是瞬态应用。自因特网诞生之日起，专家们就曾经预测传统的桌面应用面临 Web 应用的冲击。十年过去了，这些都还没有发生，为什么呢？基于 Web 页面的方案对于瞬态应用是足够了，但是对于独占应用却还远远不够。

1.1.5 忘掉 Web

现代 Web 浏览器和它最原始的出发点（从远程服务器上获得一个文档）相比已经完全不是一码事了，它们之间就像是瑞士军刀和新石器时代的狩猎工具一样，可谓是天壤之别。各种交互组件、脚本语言和插件，这些年来无法抑制地疯狂发展，近乎强制地一次又一次地创造着新的浏览体验。[可以到 www.webhistory.org/www.lists/wwwtalk.1993q1/0182.html 瞻仰一下浏览器的史前时代。1993 年的时候，Netscape

创立之前的 Marc Andreessen (Netscape 的创始人) 还在游说 Tim Berners-Lee (Web 的创始人, W3C 的领导者) 等人, 列举为 HTML 引入一个图片标签的好处]。

几年以前, 一些先行者就已经开始把 **JavaScript** 当作一种严肃的编程语言来对待。但就整体而言, 更多的人仍然把它和那些假模假样的警告框以及“点击猴子赢大奖”的广告一类的小把戏联系在一起。

浏览器大战导致 **JavaScript** 成了个被误解的、病态的孩子, **Ajax** 可以看作是他的康复中心[6]。只要适当引导, 然后给它配上合适的框架, **JavaScript** 就很有可能变成因特网的模范公民。它能真正增强 Web 应用的实用性, 而且不强迫用户安装额外的软件, 或者逼迫用户抛弃自己心爱的浏览器。得到广泛理解的成熟的工具可以帮助我们达成这一目标。本书后面会大量提到的设计模式就是这样一类工具。

推广和普及一项新技术, 既是技术事务, 也是社会行为。一旦技术已经成熟, 人们还需要领会应该如何去使用它。这一步骤常常是通过用它来做我们很熟悉的事情开始的。比如, 早年的自行车叫做“木马轮”或者“蹬行马”, 靠脚使劲蹬地的反作用力来前进。随着这一技术渐渐为大众所接受, 后来的发明者们会发明出这一技术新的使用方式, 给它加上踏板、刹车、链条齿轮以及充气轮胎。每一次的发明创造, 都使得自行车中马的影子越来越淡, 以至于彻底消失(图 1-10)。

相同的过程也发生在如今的 Web 开发领域。**Ajax** 背后的技术有能力将 Web 页面转换成某种完全不同的新东西。早期 **Ajax** 的使用尝试使得 Web 页面开始变得像“木马轮”一样不伦不类。要领悟 **Ajax** 的精髓, 我们就要忘掉 Web 的页面概念, 也就是说, 我们要打破这些年来所形成的经验。就在 **Ajax** 正式命名后的这几个月, 这样的事已经发生了不少。



图 1-10 现代自行车的发展

1.2 Ajax 的四个基本原则

我们用到的很多框架中都已经固化了基于页面的传统应用模式，同时这些应用模式也已经深深进入了我们的思想中。我们花几分钟来揭示出哪些核心概念是我们需要重新思考的，以及如何从 Ajax 的角度来重新思考。

1.2.1 浏览器中的是应用而不是内容

在传统的基于页面的 Web 应用中，浏览器扮演着哑终端[7]的角色。它对用户处于操作流程哪一阶段一无所知。这些信息全部都保存在服务器上，确切地说，就是在用户会话上。时至今日，服务器端的用户会话早已是司空见惯。如果你使用 Java 或者.NET 编程，服务器端的用户会话更是标准 API 的一部分——还有 Request、Response、MIME 类型——没有了它们简直不可想象。图 1-11 描绘了传统 Web 应用典型的生命周期。

当用户登录或者以其他方式初始化一个会话时，系统会创建几个服务器端的对象。例如，电子商务类型的网站需要创建表示购物车以及用户身份证明的对象。同时将浏览器站点的首页呈现给用户，这个 HTML 标记的数据流由模板文件以及特定于该用户的数据和内容（例如该用户最近浏览的商品列表）组成。

用户每次和服务器交互，都会获得另一个文档。在这个文档中，除了特定于该用户的数据以外，包含的其他模板文件和数据都是相同的。浏览器总是忠实地丢弃掉老的文档，显示新的文档，因为它是哑终端，而且也不知道还可以做些什么。

当用户选择退出或者关闭浏览器的时候，应用退出，会话消失。这个时候持久层会把用户下次登录后需要显示的信息存储起来。Ajax 则不同，它把一部分应用逻辑从服务器端移到了浏览器端，图 1-12 描绘了这一情况。

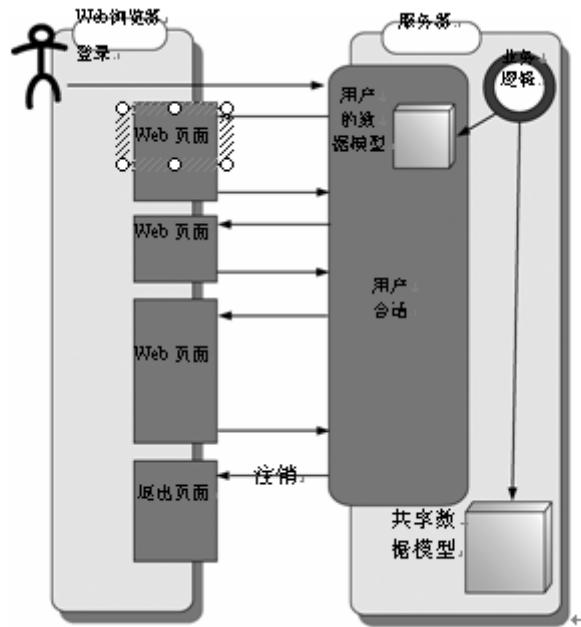


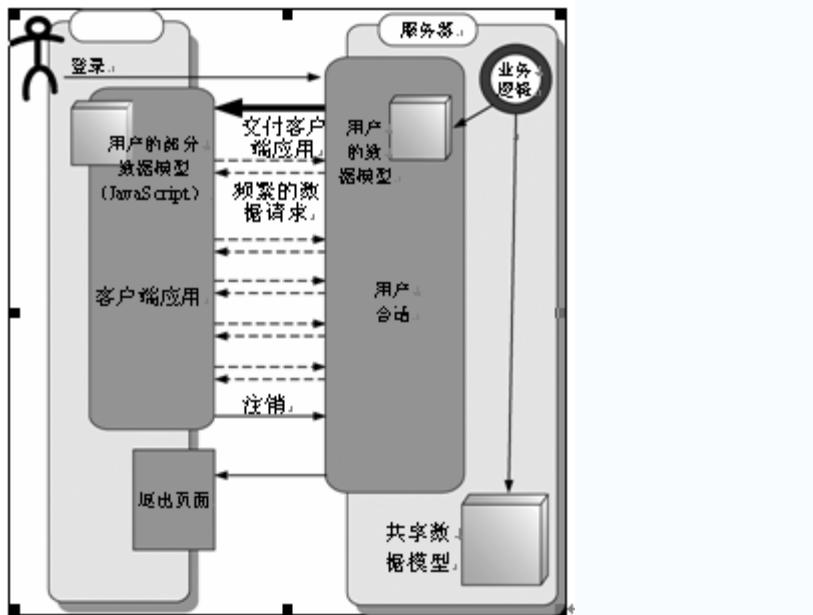
图 1-11 传统 Web 应用的生命周期。用户和应用会话的所有状态都保留在 Web 服务器上。

用户在会话中看到的是一系列的页面，每次页面切换都不可避免地要到服务器上走一个来回

图 1-12 Ajax 应用的生命周期。用户登录后，服务器交付一个客户端应用给浏览器。这个应用可以独立处理很多的用户交互，对于自己无法独立处理的交互，应用会以后台方式发送请求给服务器，而不会打断用户的操作流程

用户登录的时候，服务器交付给浏览器一个复杂得多的文档，其中包含大量的 JavaScript 代码。这个文档将会在整个会话的生命周期内与用户相伴。在这一过程中，随着用户与其交互，它的外观可能会发生相当大的变化。它知道如何响应用户的输入，能够决定对于这些请求，是自行处理还是传递给 Web 服务器（Web 服务器再去访问数据库或者其他资源），或者通过两者结合的方式进行处理。

因为这个文档在整个用户会话中都存在，所以它可以保存状态[8]。例如，购物车的内容可以保存在浏览器中而不是服务器的会话中。



1.2.2 服务器交付的是数据而不是内容

我们已经提到，在传统的 Web 应用中，服务器在每个步骤都需要把模板文件、内容和数据混合发送给浏览器。但实际上，当向购物车中添加一件物品的时候，服务器真正需要响应的仅仅是更新一下购物车中的价格。如图 1-13 所示，这只是整个文档中极小的一小部分。

基于 Ajax 的购物车可以向服务器发起一个异步请求来完成这件事，这样做显得更聪明。模板文件、导航列表和页面布局上的其他部分已经随着初始页面发送给了浏览器，服务器无需重发，以后每次只需要发送相关的数据就可以了。

Ajax 应用可以通过多种方式来做这件事情。例如，返回一段 JavaScript 代码、一段纯文本或者一小段 XML 文档。这些方式各自的优缺点，我们将留到第 5 章再详细考察。但是，毫无疑问的是，无论返回数据采用何种格式，这些方式所传输的数据量都要比传统的 Web 应用中一股脑返回一个大杂烩的方式少得多。

在 Ajax 应用中，网络的通信流量主要是集中在加载的前期，无论如何，用户登录后是需要一次性地将一个大而复杂的客户端交付给浏览器。但是在此之后，与服务器的通信则会有效率得多。对于瞬态应用来说，积累起来的通信流量要比以前的基于页面的 Web 应用少很多。与此同时，平均的交互次数则有所增加。整体而言，Ajax 应用的带宽消耗要比传统的 Web 应用低一些。

1.2.3 用户交互变得流畅而连续

浏览器提供了两种输入机制：超链接和 HTML 表单。

超链接可以在服务器上创建，并预加载指向动态服务器页面或者 servlet 的 CGI 参数。可以用图片或者 CSS（层叠样式表）来装饰超链接，并且当鼠标停在上面时还可以提供基本的反馈。经过合理设计，超链接可以变成一个很有想像力的 UI 组件。

表单则提供了桌面应用的一组基础 UI 组件：输入文本框、单选按钮和多选按钮，还有下拉列表。但仍然缺少很多有用的 UI 组件，例如，没有可用的树控件、可编辑的栅格、组合输入框等。表单像超链接一样，也指向服务器的一个 URL 地址。

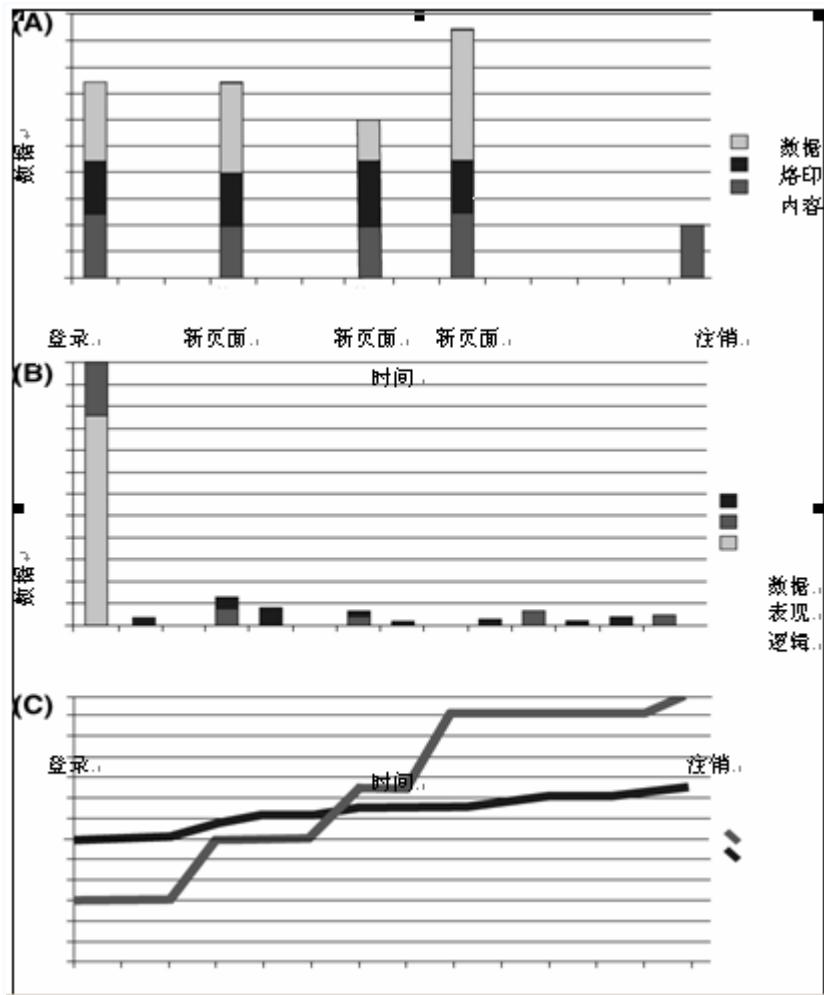


图 1-13 细分服务器发送的内容，(A)是传统的 Web 应用，(B)是 Ajax 应用。(C)表示随着应用使用时间的延长，累积的网络流量的增长情况

超链接和表单也可以指向 JavaScript 函数。这一技术通常用在将数据提交给服务器之前对表单输入进行简单的校验，如检验是否有空值，数值是否越界等等。这些 JavaScript 函数的生存期和页面本身是一致的，当页面提交之后，这些函数也就不存在了。

当一个页面已提交而下一个页面还没有显示出来的时候，用户实际上处于没人管的状态。老的页面还要显示一会儿，浏览器甚至还会允许用户点击一些链接。但这些点击可能会导致一些不可预料的结果，甚至破坏服务器端会话的状态。用户通常应该等到页面刷新完成，当然也可以选择在刷新完成之前就在新页面上做一些操作。例如，当页面只显示了一部分时从中选择一条裤子放进购物车不大可能会造成什么破坏（例如，不会修改顶级的服装分类：男装、女装、童装、配饰）。

我们继续看这个购物车的例子。**Ajax**的购物车是通过异步方式发送数据的，用户可以很快地把东西拖进来，就像点击一样快。只要客户端购物车的代码足够健壮，它可以很轻松地处理这样的负载，而用户则可以继续做他想做的事。

要知道，在服务器端并没有一个真正的购物车等着装东西，只有会话中的一个对象而已。购物的时候，用户并不想知道会话对象，购物车对于用户而言是一个较恰当的比喻，用现实世界中熟悉的概念来描述这里发生的事情。对于用户来说，如果强迫他们去理解计算机领域中的术语，只会让他们远离网站。等待页面的刷新，一下子就把用户从愉快的使用体验中拽了出来，让他感觉到自己所面对的只不过是一台冰冷的机器罢了（图 1-14）。使用**Ajax**来实现这些应用则可以避免这些令人不快的体验。当然了，在这个例子中的购物只是一个瞬态活动。考察一下其他的业务领域，例如，一个业务量很大的帮助中心或者一项复杂的工程任务，如果因为需要等待页面刷新而将工作流程打断几秒钟[9]，那肯定是不可接受的。

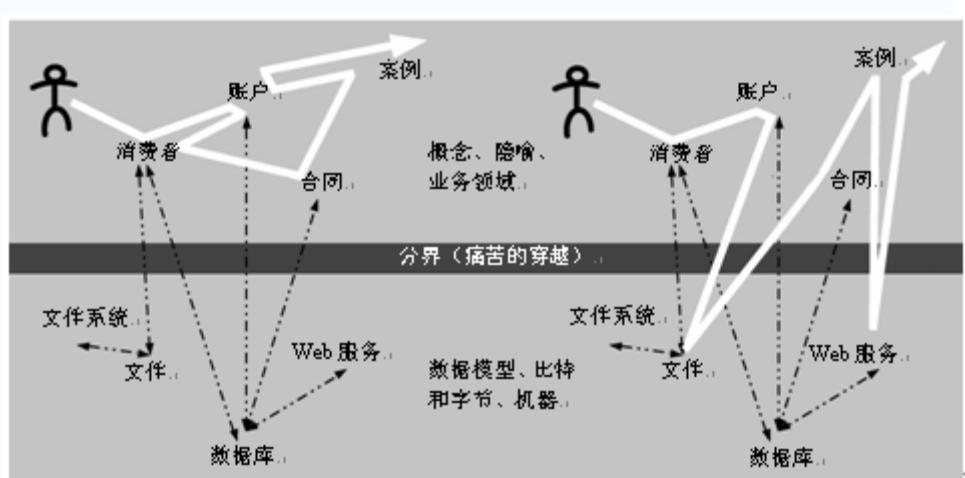


图 1-14 处理事件打断了用户的工作流程。用户要处理与业务相关的和与计算机相关的两种对象。这迫使用户频繁地在这两者之间切换，从而导致用户注意力分散，工作效率降低

Ajax 的另一个好处是，我们可以对丰富的用户操作事件进行捕获。类似于拖拽这样的复杂 UI 概念也不再是遥不可及的。这使得 Web 应用的 UI 体验可以全面提升到近乎与桌面应用的 UI 组件相媲美的高度。从可用性的角度来看，这很重要，不仅仅是因为它释放了我们的想象力，而且也是因为它可以将用户交互和服务器端的请求更加充分地混合起来。

在传统的 Web 应用中，与服务器交互需要点击超链接或者提交表单，然后等待页面的刷新，这打断了用户的工作流程。与之相对应的是，让服务器响应鼠标移动、拖拽或者键盘输入这样的用户事件，也就是说，服务器在用户身边为用户服务，而不是挡在用户前面，打断他的操作。[Google Suggest](http://www.google.com/webhp?complete=1) (www.google.com/webhp?complete=1) 就是这样一个简单的但是很有说服力的例子。当用户在搜索框键入一些字符的时候，应用从服务器取回与用户已键入字符串相似的搜索条目（根据全世界其他人的搜索），并且显示在输入框下方的下拉列表中。第 8 章将提供这类服务的一个简单实现。

1.2.4 有纪律的严肃编程

现在传统的 Web 应用有时候也会用到 JavaScript，不过主要是用来给页面添加一些花哨的东西。基于页面的模型使得这样的增强没有办法更进一步，限制了用户可以得到的更加理想的交互。这种类似于第 22 条军规的状况，使得 JavaScript 很不公平地获得了一种琐碎的、自由散漫的编程语言的名声，为那些严肃的开发者[10]所不屑。

为 Ajax 应用编程的情况则完全不同。提交给用户运行的应用将会一直运行直到用户关闭程序为止。不崩溃，不变慢，也没有内存泄漏之类的毛病。如果我们的产品定位于独占式应用的市场，这还意味着很多小时的密集使用。要达到这个目标，当然需要高性能的、可维护的代码，这与服务器端应用的要求是一致的。

相比之下，Ajax 的代码库会比传统的 Web 应用大很多。对代码库进行良好的组织是非常重要的。编写代码不再是单个开发者的职责，而是整个团队来参与。可维护性、分离关注点、共同的编程风格以及设计模式，这些都是需要考虑的问题。

从某个角度来看，Ajax 应用就是用户所使用的一块复杂的代码，它需要高效地与服务器进行通信。它显然来源于传统的基于页面的 Web 应用，但是它们之间的相似性也仅限于此，两者之间的差别就像是木马轮和现代自行车之间的差别。在脑海中要记得它们之间的这些差别，因为只有这样才能创造出真正引人注目的 Web 应用。

1.3 真实世界中的 Ajax 富客户端

理论已经说得够多的了，让我们再来看看一些真实世界中的应用，获得一些感性认识。Ajax 已经用于创建一些重要的应用，使用 Ajax 方法的好处是一目了然的。Ajax 现在仍然处在发展的早期，这么说吧，这就好比自行车发展到了还只有几个人装上踏板和实心橡胶轮胎的时代，刚有人开始研制盘式刹车器和变速齿轮。下面一节将会考察 Ajax 当前的状态，然后详细分析 Ajax 的一个卓越的早期应用，看看使用 Ajax 将得到什么回报。

1.3.1 现状

打造 Ajax 的版图，Google 比其他公司做得更多（和其他领域的开拓者一样，早在 Ajax 这个名字浮出水面之前，他们就做了很多工作）。在 2004 年初，他们就推出了 beta 版本的 GMail 服务。除了它阔绰的容量，GMail 最为人称道的就是它的用户界面。它允许用户一次打开多个电子邮件，并且，即便用户正在写邮件，

邮件列表也能够自动更新。与同期的大多数由因特网服务提供商（ISP）提供的Web邮件系统相比，这无疑是一个显著的进步。而与很多Web界面模仿Microsoft Outlook和Lotus Notes的企业邮件服务相比，GMail并没有依赖重量级的、容易出问题的ActiveX控件和Java applet，但在功能上却毫不逊色。这样做所带来的好处就是完全的跨平台，可以在任何平台[11]、任何地点使用GMail的服务，无需像企业邮件服务的用户那样需要预先在机器上安装一堆额外的软件[12]。

此后，在提供更加丰富的交互性方面，Google 走得更远。例如，当用户键入字符时，Google Suggest 可以为用户提供与输入字符相符的提示，帮助他们完成想要键入的搜索字符串；Google Maps 可以执行交互式的、可缩放的基于位置的搜索。与此同时，其他公司也纷纷开始试验使用这一技术，例如 Flickr 的在线照片共享系统，它现在已经是雅虎网站功能的一部分了。

到目前为止，我们这里讨论的应用都只算初步的尝试。它们仍然是瞬态应用，为偶尔的使用而设计。几个月来相关的技术框架显著增加，这可以看作是市场向独占式 Ajax 应用迁移的征兆。第 3 章将会考察其中的一些框架，而在本书的附录 C 中，我们将试图总结这一领域的当前状态。

这些证据表明，Ajax 正在赢得市场的青睐。我们开发者可以出于个人兴趣来玩一种新技术，但是像 Google 和雅虎这样的大型企业只有在看到了诱人的商业前景之后才会接受某种新技术。

我们已经概括了 Ajax 的很多理论上的优势，在下面一个小节，我们将剖析 Google Maps，看看这些理论是如何组合在一起建造真实的应用的。

1.3.2 Google Maps

Google Maps 是结合了地图浏览和搜索引擎的产物。初始状态，显示的是美国地图（图 1-15）。这个地图可以自由地通过文本来查询，并且可以精确地细化到街道地址或者像宾馆和餐馆这样的生活设施（图 1-16）。

查询功能类似传统的 Web 应用，需要刷新整个页面，但是地图本身是 Ajax 驱动的。在宾馆搜索的每一个链接上点击，将会在地图上立即弹出一个提示框，地图还可能会稍微滚动以适应这个提示框的位置。滚动地图本身可能是 Google Maps 最有意思的功能了，用户可以用鼠标拖拽整张地图。地图是由很多块小的图片拼接而成的，如果用户滚动得足够远，要显示出一些新的区域时，这些区域的图片将会异步地加载。这个延迟很明显，可以观察到起初它们是一些空白的区域，当它们被加载时，会一块一块地显示出来。但是在这个地图的更新过程中，用户还可以继续滚动，触发更多的更新。这些小块的地图在用户的会话过程中会被浏览器缓存起来，这使得当回到以前曾经访问过的地图时，显示的速度非常之快。

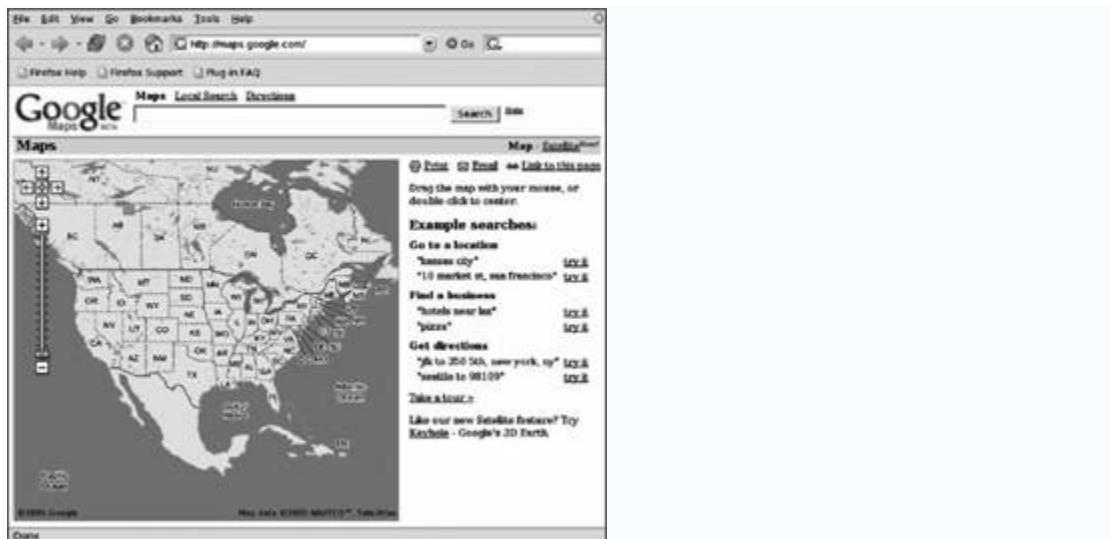


图 1-15 Google Maps 的首页提供了一个可以滚动和放大的美国地图，还有熟悉的 Google 搜索

条。要注意的是，缩放控件是在地图之上而不是在它的旁边，这使得用户的视线无需离开地图就可以进行缩放控制

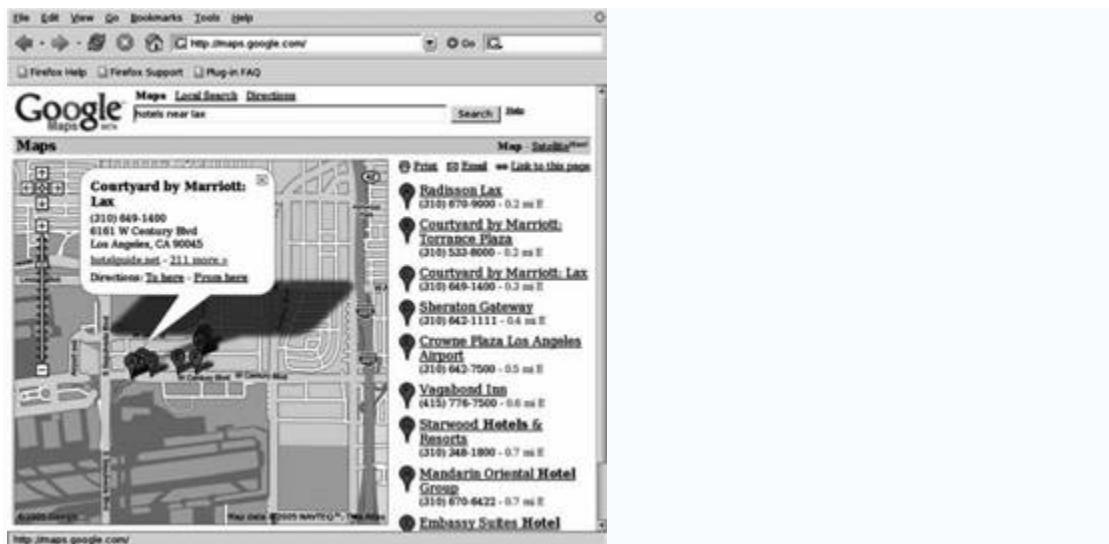


图 1-16 在 Google Maps 上进行宾馆搜索。注意，那些阴影和可爱的提示气球是使用传统 DHT

ML 技术创建的。Ajax 请求的使用使得这些功能更加动态也更加好用

回到我们关于可用性的讨论，你会发现有两个重要的东西浮现了出来。其一，触发下载新地图数据的操作不是点击一个特定的“取得更多的地图”的链接，而是用户的操作，也就是说，移动地图。用户的工作流程并没有被与服务器的通信所打断。其二，请求本身是异步的，这就意味着，当获取新数据的时候，相关的链接、缩放控件以及其他页面上的功能仍然都可以使用。

因特网上的地图服务并不是什么新东西。如果看看Ajax之前的典型的因特网地图网站，我们看到的是完全不同的交互模式。地图也明显地划分成很多个小块，缩放控件和导航链接可能会在地图的边缘。每次点击这些控件，都会引起整个页面的刷新，随后出现一个显示着不同地图区域的相似的页面。用户的操作总是被打断，在看到Google Maps之后，毫无疑问，用户会觉得Ajax之前的使用方式缓慢而又沉闷[13]。

转到服务器端，毫无疑问，所有的地图服务都有一个强大的地图系统作为支撑。每一小块的地图都是一个图片。当用户滚动地图的时候，Ajax之前的地图服务站点的Web服务器需要不断地刷新页面的模板；而在Google Maps上，一旦运行起来，服务器只需要提供必需的数据，而很多的图片已经被浏览器缓存起来了（是的，只要提供相同的URL，浏览器也能缓存页面中的图片。但是，采用这种方案，浏览器在检查缓存是否过期时仍然会造成不必要的服务器流量，而且比起以编程方式在内存中缓存图片的方案，也不是很可靠）[14]。作为Google所推出的一个最卓越的展示品[15]，节约带宽是必须要考虑的问题。

对于像Google这样的在线服务来说，易于使用是一个关键的特征，只有这样才能让用户用过之后再回来。而且页面点击数对于商业上的成败至关重要。而引入Ajax所提供的灵活的UI之后，Google的竞争使得传统的地图服务提供商们忧心如焚。当然，后端服务的质量也是需要比较的因素，但是当其他部分都是一样的时候[16]，Ajax形成了巨大的商业优势。

可以预期，像这样更为丰富的界面的公开展示会越来越多，变得更加普遍。作为一种很容易推销出去的技术，在接下来的几年里 Ajax 看来会有很光明的前途。然而，其他的富客户端技术也在谋求获得市场份额，虽然它们不在本书的讨论范围之列，但在结束我们的综述之前对它们做一下展望也是很重要的。

1.4 Ajax 的替代方案

市场需要基于 Web 的应用表现能力更加丰富，响应更加灵敏，Ajax 满足了这种需求，而且无需在客户端安装任何额外的软件。但是这一领域并非只有一个竞争者，在某些情况下，它甚至并不是最合适的选择。下面我们简要地描述一下主要的替代方案。

1.4.1 基于 Macromedia Flash 的方案

Macromedia的Flash是一种采用压缩的矢量图形格式来播放交互式电影的系统。Flash电影是一种流媒体格式，也就是说，可以一边下载一边播放，而不用等到媒体的所有字节全部都下载到本地之后再播放。F

lash电影是交互式的，它使用**ActionScript**来编程（**ActionScript**是**JavaScript**的一种近亲[17]）。它也提供了一些对输入表单**UI**组件的支持。**Flash**可以应用于从交互游戏到复杂商业应用的用户界面的广阔领域。**Flash**有非常棒的矢量图形支持，而在相对应的**Ajax**技术领域中，这部分则是完全空白的[18]。

Flash作为一种浏览器插件，已经存在了很长的时间。通常来说，依赖浏览器的插件并不是一个好主意，但是对于**Flash**则不然，主流浏览器的安装包中已经包含了它。而且，它也能跨越**Windows**、**Mac OS X**以及**Linux**三大主流的桌面操作系统平台（而且在**Linux**上的安装文件还要比其他平台小一些）。

如果你打算使用**Flash**来创建富客户端应用，可以从**Macromedia**的**Flex**和开源代码的**Laszlo**中进行选择，两者都是很有趣的技术，它们都提供了简化的服务器端框架用来生成基于**Flash**的用户界面，都在服务器端采用了**Java/J2EE**平台。另外，如果你还想要使用更低层次的功能来动态地创建**Flash**电影，一些工具包（例如**PHP**的**libswf**模块）可以为你提供这方面的核心功能。

1.4.2 Java Web Start 及其相关技术

Java Web Start是把基于**Java**的**Web**应用打包保存在**Web**服务器上的规约。通过一个在桌面电脑上运行的**Web Start**过程，可以自动完成应用的查找、下载和运行。在一个可以启动**Web Start**过程[19]的浏览器中，只需要点击一个超链接就可以无缝地访问这些应用。**Web Start**已经整合进了最近发布的**Java**运行环境，在**IE**和**Mozilla**浏览器中，安装过程会自动打开这个特性。

一旦**Web Start**应用下载完毕，它就被存储在文件系统的一个受控的“沙箱”之中，如果服务器上有了应用的新版本，它还能够自动更新。这使得当网络连接中断的时候，它仍然能够继续运行，而且因为它是存储在本地的，即使是一个几兆字节大小的重型应用，重新加载时也不会产生网络负载。应用本身包含数字签名[20]，用户可以选择打开全部访问权限，允许应用访问文件系统、网络端口以及其他本地资源。

传统**Web Start**应用的用户界面使用**Java**的**Swing UI**组件工具包来开发。对于**Swing**一直是褒贬不一，正反双方都有着强大的理由。除此之外，也可以使用建造**IBM**的**Eclipse**平台的**SWT UI**组件工具包来开发，只是需要多做一点额外的工作。

微软的**.NET**平台也提供了类似的功能，称为**No Touch Deployment**，它承诺会提供易于部署、丰富的用户界面以及安全性等类似的功能。

这两种技术的主要问题在于，它们都需要预先安装一个运行环境。当然，所有的富客户端都需要一个运行环境。对于Flash和Ajax来说，它们的运行环境是早已经部署好了，而且普遍存在。Ajax的运行环境就是浏览器本身。Java和.NET的运行环境则大大受限于它们目前的发行情况[21]，而对于一个公共的Web服务来说，这是不能依赖的[22]。

1.5 小结

我们讨论了瞬态应用与独占应用的差别及其各自的要求。瞬态应用也需要能够提供良好的用户体验，但是用户仅在自己原有工作流程之外偶尔使用一下，使用中的一点瑕疵是可以接受的。与此形成鲜明对比的是独占应用。它是为长时间的密集使用而设计的，其界面必须设计得近乎隐形，以免干扰用户集中于手头任务的注意力。

客户/服务器和相关的*n*层架构是采用合作方式的或者集中控制方式的应用的精髓所在，但在这个架构中，网络延迟是一个会严重影响用户工作效率的棘手问题。解决这一冲突的有效方案就是采用异步事件机制，相比之下，传统Web应用的请求—响应模式没有办法很好地解决这个问题。

我们为自己设定的目标是：通过Web浏览器交付具有良好可用性的独占应用，以满足提高用户的生产力和通过网络来共享数据两方面的需求，同时还要具备Web应用集中维护的优点。为了成功地实现这一目标，我们需要以一种完全不同的方式来思考Web页面和应用。我们发现，下面的这些要点是需要牢记在心的：

- 浏览器中的是应用，而不是内容。
- 服务器交付的是数据，而不是内容。
- 用户和应用的交互是连续的，大部分对于服务器的请求是隐式的而不是显式的。
- 代码库是巨大的、复杂的，而且是组织良好的，这个特点对于架构来说非常重要，需要认真对待。

在下一章我们会分解Ajax的技术要点，并开始动手开发一些代码。本书的剩余部分还将考察一些重要的设计原则，这将有助于我们实现设定的目标。

1.6 资源

想要更加深入地了解本章所提到的一些文章的内容，可以访问以下的 URL：

- Jesse James Garrett 在 2005 年 2 月 18 日的这篇文章中首次使用了 Ajax 这一名称：<http://www.adaptivepath.com/publications/essays/archives/000385.php>。
- Alan Cooper 对于独占和瞬态应用的解释可以在这个地方找到：http://www.cooper.com/articles/article_your_programmes_posture.htm。
- Google Maps 的地址。如果你生活在美国，就看这里：<http://maps.google.com>。

如果你生活在英国，则可以看这里：

<http://maps.google.co.uk>。

如果你生活在月球，那就看这里好了：

<http://moon.google.com>。

自行车的图片是从 Pedaling History 的网站上得到的：www.pedalinghistory.com。

第2章 Ajax 新手上路

2.1 Ajax 的关键元素

Ajax 不是单一的技术，而是四种技术的集合。表 2-1 简要介绍了这些技术，以及它们所扮演的角色。

表 2-1 Ajax 的关键元素

JavaScript	JavaScript 是通用的脚本语言，用来嵌入在某种应用之中。Web 浏览器中嵌入的 JavaScript 解释器允许通过程序与浏览器的很多内建功能进行交互。Ajax 应用程序是使用 JavaScript 编写的
CSS（层叠样式表）	CSS 为 Web 页面元素提供了一种可重用的可视化样式的定义方法。它提供了简单而又强大的方法，以一致的方式定义和使用可视化样式。在 Ajax 应用中，用户界面的样式可以通过 CSS 独立修改

(续)

DOM（文档对象模型）	DOM 以一组可以使用 JavaScript 操作的可编程对象展现出 Web 页面的结构。通过使用脚本修改 DOM，Ajax 应用程序可以在运行时改变用户界面，或者高效地重绘页面中的某个部分
-------------	---

XMLHttpRequest 对象

XMLHttpRequest 对象允许 Web 程序员从 Web 服务器以后台活动的方式获取数据。数据格式通常是 XML，但是也可以很好地支持任何基于文本的数据格式（XMLHttpRequest 这个名字取得实在是很不恰当）。尽管 XMLHttpRequest 对于完成这件工作来说是最为灵活和通用的工具，但还有其他方法也可以从服务器获取数据。我们在本章中会讨论所有的方法

在第 1 章中我们看到了 Ajax 如何为用户提供了复杂的、运转良好的应用，改善了用户的交互体验。JavaScript 就像胶水将各个部分粘合在一起，定义应用的工作流和业务逻辑。通过使用 JavaScript 操作 DOM 来改变和刷新用户界面，不断地重绘和重新组织显示给用户的数据，并且处理用户基于鼠标和键盘的交互。CSS 为应用提供了一致的外观，并且为以编程方式操作 DOM 提供了强大的捷径。XMLHttpRequest 对象（或者类似的机制）则用来与服务器进行异步通信，在用户工作时提交用户的请求并获取最新的数据。图 2-1 显示了这些技术在 Ajax 中是如何配合的。

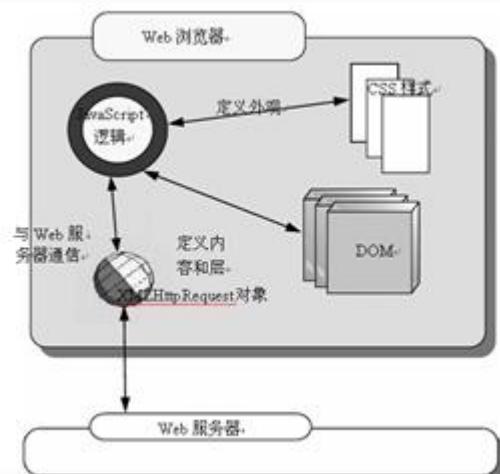


图 2-1 Ajax 的四个主要组件：JavaScript 定义了业务规则和程序流程。应用程序使用 XMLHttpRequest 对象（或类似的机制）以后台方式从服务器获得的数据，通过 DOM 和 CSS 来改变界面的外观

Ajax 的四种技术之中，CSS、DOM 和 JavaScript 这三个都不是新面孔，它们以前合在一起称作动态HTML，或者简称DHTML。DHTML是在 1997 年因特网大跃进时代放的一颗“卫星”，它从来也没有实现过自己最初的承诺（在这个行业早已见怪不怪了[\[1\]](#)）。DHTML可以为Web页面创造新奇古怪的、交互性很强的界面，但是它永远也无法克服需要完全刷新整个页面的问题。问题在于，如果没有和服务器通信的能力，空有漂亮的界面，还是无法实现一些真正有意义的功能。Ajax除了大量使用DHTML，还可以发送异步请求，这大大延长了Web页面的寿命。通过与服务器进行异步通信，无需打断用户正在界面上执行的操作，Ajax与其前任DHTML相比，为用户带来了真正的价值。

更加方便的是，所有这些技术都已经预先安装在绝大多数的现代 Web 浏览器之中，包括微软公司的 IE、Mozilla/Gecko 系列的浏览器（例如 Firefox、Mozilla Suite、Netscape Navigator 和 Camino）、Opera、苹果公司的 Safari，以及它的近亲 UNIX KDE 桌面系统里的 Konqueror。可惜的是，这些技术的实现细节在不同的浏览器之间，甚至在同一浏览器的不同版本之间存在着很多差异，这就是所谓的跨浏览器不兼容（cross-browser incompatibilities）问题。不过近 5 年来，这一状况得到了持续的改善，现在我们写代码时需要处理的此类问题已经非常少了。

所有现代的操作系统都会预先安装一个现代的浏览器。也就是说，这个星球上的绝大多数桌面电脑和笔记本电脑都已经为运行 Ajax 应用程序做好了准备。这正是大多数 Java 或 .NET 开发者所梦想的。（虽说在 PDA 和智能手机上也安装了浏览器，但是它们的功能通常已经被大幅裁减，无法支持 Ajax 所需的关键技术。即使它们能够支持这些技术，屏幕尺寸和输入方法方面的限制也是非常棘手的问题。因此就目前而言，Ajax 主要是一种用于桌面电脑和笔记本电脑上的技术。）

我们的做法是先分别考察每一种技术，然后再去考察它们如何相互配合。如果你已经是一个 Web 开发的老手，对这些技术已经烂熟于胸，那么可以直接跳到第 3 章，在那里我们将会考察如何使用设计模式来管理这些技术。

我们就从考察 JavaScript 来开始我们的探索之旅吧。

2.2 用 JavaScript 改善用户体验

JavaScript 毫无疑问是 Ajax 工具箱中的核心技术。Ajax 应用程序完全下载到客户端的内存中，由数据、表现和程序逻辑三者组成，JavaScript 就是用来实现逻辑的工具。JavaScript 是一种混合了多种编程思想的通用编程语言，提供了一个表面上与 C 系列语言相似的语法接口。

JavaScript 可以简短地描述为一种弱类型的、解释型的和通用的脚本语言。弱类型（loosely typed）意味着变量不需要明确声明为字符串、整数或者对象，同一个变量可以使用不同的类型来赋值。例如，下面的代码是合法的：

```
var x=3.1415926;  
x='pi';
```

变量 x 最初定义为一个数字类型的值，后来又赋给一个字符串类型的值。

解释型（interpreted）意味着不需要编译，源代码本身就可以直接执行。将源代码放在 Web 服务器上，通过因特网传输到用户的浏览器中，JavaScript 应用的部署就完成了。此外，在 JavaScript 中，甚至还可以在运行时对一小段代码进行求值。

```
var x=eval('7*5');
```

在这里，我们将表达式定义为一段文本，而不是两个数字加一个算术操作符。使用这段文本来调用 `eval()`，将会解释文本中包含的 JavaScript，并且返回表达式的值。在大多数场合，这样做执行效率不高；但是在某些场合，这样做是很有用的，因为它可以带来很大的灵活性。

通用（general purpose）意味着这种语言适用于大部分的算法和编程任务。JavaScript 语言核心支持数字、字符串、日期和时间、数组、用于处理文本的正则表达式，以及数学函数（例如三角运算、随机数生成等）。JavaScript 还支持定义结构化的对象，因此可以使用面向对象的设计原则来建造更加复杂的代码。

在浏览器环境中，通过 JavaScript 引擎可以访问浏览器的一些本地功能，例如 CSS、DOM、XMLHttpRequest 对象，这允许页面开发者通过编程方式在不同程度上控制页面的表现。尽管在浏览器中遇到的 JavaScript 环境和特定于浏览器的对象紧紧绑在一起，但是从底层来看，语言本身的语法却是一致的。

本书并不是一本详细介绍 JavaScript 基础知识的教程。在附录 B 中，我们近距离考察了 JavaScript 语言，讨论了 JavaScript 与 C 系列语言（包括名字上很相似的 Java 语言）的根本区别。JavaScript 的例子散布在本书的各个部分，已经有几本其他书籍专门介绍这门语言的基础知识（参见本章“资源”一节）。

在整个 Ajax 技术体系中，JavaScript 就像胶水一样将所有的部分黏合在一起，对 JavaScript 有基本的了解是编写 Ajax 应用的前提。只有熟练掌握和深刻理解了 JavaScript，你才能够释放出 Ajax 的全部潜力。

接下来的话题将会转向 CSS，它是 Web 页面中元素视觉样式的主宰。

2.3 用 CSS 定义应用的外观

CSS 是 Web 设计沿用已久的部分，无论是在传统的 Web 应用还是在 Ajax 应用中，CSS 都是一种频繁使用的技术。样式表提供了集中定义各种视觉样式的方法，并且可以非常方便地设置在页面的元素上。样式表可以定义一些明显的样式元素，例如颜色、边框、背景图片、透明度和大小等。此外，样式表还可以

定义元素相互之间的布局以及简单的用户交互功能，仅仅通过样式表就可以实现炫目的视觉效果。

在传统的 Web 应用中，样式表提供了一种很有用的方法，可以在某个地方定义在很多页面中重用的样式。在 Ajax 应用中，我们不再将应用思考为快速切换的一系列页面，但是样式表仍然是很有帮助的，它可以用最少的代码动态地为元素设置预先定义的外观。在本节中我们将看到一些基本的 CSS 例子。不过首先我们来考察一下 CSS 规则是如何定义的。

CSS 样式为一个文档定义显示规则，通常放在一个单独的文件中，由应用这些样式的 Web 页面来引用。当然，也可以在 Web 页面中定义样式规则，但这通常是一种很糟糕的做法。

一个样式规则由两部分组成：选择器（selector）和样式声明（style declaration）。选择器表明要为哪一个元素设置样式，样式声明则表明要应用哪些样式属性。例如我们想让文档中的一级标题（也就是 `h1` 标签）以红色显示，就可以这样定义 CSS 规则：

```
h1 { color: red }
```

这个例子中的选择器非常简单，文档中的所有 `h1` 标签都将应用这个样式。样式声明也很简单，仅仅设置了一个样式属性。在实际应用中，选择器和样式声明都要复杂得多。我们来分别考察一下它们的设置方法，就从选择器开始吧。

2.3.1 CSS 选择器

除了为某种类型的 HTML 标签设置样式，我们还可以将规则限制在一个指定的上下文内。有几种方法可以指定上下文：通过 HTML 标签类型、通过已声明类的类型或者通过元素的唯一 ID。

先看看标签类型的选择器。举例来说，如果我们只需要对位于 `div` 标签中的 `h1` 标签应用上面的样式规则，那么就可以将 CSS 规则修改为：

```
div h1 { color: red; }
```

这也称作“基于元素的选择器”，因为一个 DOM 元素是否应用这个样式取决于它的元素类型。我们也可以定义样式类（style class），通过与 HTML 标签类型无关的样式类来设置样式。例如，如果我们定义一个名为 `callout` 的样式类，将其显示在一个彩色框中，就可以这样写：

```
.callout { border: solid blue 1px; background-color: cyan }
```

要将一个样式类分配给一个元素，我们只需要简单地在 HTML 标签中声明一个类属性，例如：

一个元素可以分配多个样式类。假设我们还定义了一个名为`loud`的样式类：

```
.loud { color: orange }
```

可以在文档中同时应用这两个样式，就像这样：

And I'll appear as an unappealing mixture of both!

第三个元素会显示为蓝色边框、青色背景和橙色文字。通过组合CSS样式类，我们可以创造出令人愉快的、和谐的设计。

我们也可以混合使用样式类和基于元素的规则，来定义一个仅仅针对特定标签类型的样式类。例如：

```
span.highlight { background-color: yellow }
```

这个规则仅仅会应用在声明了值为`highlight`的类属性的标签上。其他的标签或者其他有`class='highlight'`属性设置的标签将不受影响。

我们甚至可以与父子关系的选择器联合使用，创建在非常特定的场合下使用的规则：

```
div.prose span.highlight { background-color: yellow }
```

这个规则仅会应用在设置了`prose`样式类的标签中嵌套的、设置了`highlight`样式类的标签。

I'll appear as a callout

I'll be bright orange

And I'll appear as a callout!

I'll appear as a normal bit of text

也可以为单个元素指定规则，它需要有唯一ID，这是通过在HTML中加上`id`属性来完成的。HTML中只会有一个使用这个ID的元素，所以，这样一类选择器通常只用来选择页面中的一个元素。例如，为了突出显示页面中的关闭按钮，我们可以这样定义样式：

```
#close { color: red }
```

在CSS中我们还可以定义基于伪选择器（pseudo-selectors）的样式。浏览器定义了一些有限的伪选择器，这里展示一些最有用的，例如：

```
*:first-letter {
```

```
    font-size: 500%;
```

```
    color: red;
```

```
float: left;  
}  
}
```

通过这个选择器，我们可以将任何元素的第一个字母以很大的粗体红色字体来显示。我们还可以将这个规则收紧一些，就像这样：

```
p.illuminated:first-letter {  
    font-size: 500%;  
    color: red;  
    float: left;  
}
```

这样的话，红色的效果只会应用在设置了`illuminated`样式类的`p`元素上。其他有用的伪选择器包括`first-line`和`hover`。`hover`可以改变当鼠标光标停在超链接上时，超链接的显示效果。例如，当鼠标光标停在链接上时，链接显示为黄色，我们可以编写下列规则：

```
a:hover{ color: yellow; }
```

CSS 选择器的基本内容就是这些了。在这些例子中，我们已经非正式地介绍了几种样式声明，现在我们就来仔细考察一下。

2.3.2 CSS 样式属性

HTML 页面中的每一个元素都可以通过很多方式来设置样式。对于那些最常见的元素（例如`h1`标签）来说，可能会有一大堆的样式应用在它们身上。我们来大致看一下。

对于元素的文本，可以设置它的颜色、字体大小、字体粗细、字体类型等样式。字体类型可以有多个选项，以便在客户端机器上并未安装希望使用的字体时，仍然能够通过使用其他的替代字体得到不错的显示效果。将一个段落的样式设置为灰色终端风格的文本，我们可以这样来定义：

```
.robotic{  
    font-size: 14pt;  
    font-family: courier new, courier, monospace;  
    font-weight: bold;  
    color: gray;
```

```
}
```

或者，使用更简捷的方式将所有字体元素合并在一起：

```
.robotic{  
    font: bold 14pt courier new, courier, monospace;  
    color: gray;  
}
```

在上面的两种样式声明中，多个样式属性都以“键—值”对的形式来书写，使用分号来分隔。

通过设置margin和padding属性，CSS可以定义元素的布局和尺寸（常常称作盒模型），而且，上下左右每一边的属性都可以单独设置：

```
.padded{ padding: 4px; }  
.eccentricPadded {  
    padding-bottom: 8px;  
    padding-top: 2px;  
    padding-left: 2px;  
    padding-right: 16px;  
    margin: 1px;  
}
```

元素的尺寸可以通过width和height属性来指定。元素的位置则可以指定为绝对的或者相对的。绝对定位的元素可以通过设置top和left属性指定位置，而相对定位的元素则依赖于页面其他部分的布局情况。

通过使用background-color属性可以设置背景颜色，还可以通过background-image属性来设置背景图片：

```
.titlebar{ background-image: url(images/topbar.png); }
```

通过设置visibility:hidden或display:none可以将元素隐藏起来。如果是相对定位，在第一种情况下，元素仍会保留它在页面上的位置；而在第二种情况下则不会，元素会消失得无影无踪。

以上内容讨论了使用 CSS 建造 Ajax 应用用户界面所需要的基本样式属性。在下面一小节，我们来看一个例子，实际使用一下 CSS。

2.3.3 简单的 CSS 例子

我们讨论了 CSS 的核心概念，现在来进行实战。CSS 可以用于创建优雅的图形设计，但是在 Ajax 应用中，我们常常更加关注创建模仿桌面 UI 组件的用户界面。作为 CSS 这种用法的一个简单例子，图 2-2 展示了一个使用 CSS 来设置样式的文件夹 UI 组件。

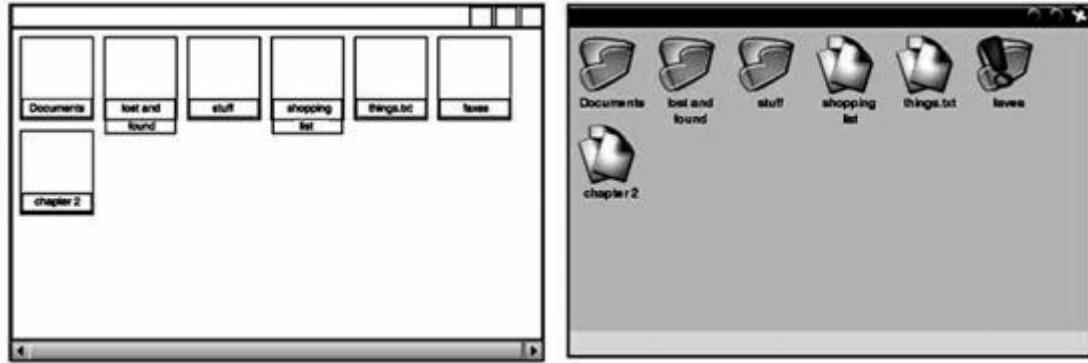


图 2-2 使用 CSS 为用户界面组件设置样式。两个截屏都是由同一个 HTML 生成的，只是采用了不同的样式表。左边的截屏所使用的样式表仅仅保留了定位元素占据的位置，而右边的截屏所使用的样式表则添加了一些装饰效果，例如彩色和图片

在图 2-2 右边的截屏中，CSS 对于创建窗口部件来说起到了两个作用。我们来逐个考察一下。

1. 使用 CSS 布局

CSS 完成的第一个工作是对这些元素进行定位。最外面的元素显示了整个的窗口，它使用绝对定位的方式：

```
div.window{  
    position: absolute;  
    overflow: auto;  
    margin: 8px;  
    padding: 0px;  
    width: 420px;  
    height: 280px;  
}
```

在内容区域中，使用 `float` 属性来设置图标的样式，使得它们在父元素的边界范围之内自动排列起来，并

且在需要的时候可以自动换行。

```
div.item{  
    position: relative;  
    height: 64px;  
    width: 56px;  
    float: left;  
    padding: 0px;  
    margin: 8px;  
}
```

嵌入在item元素内的itemName元素，通过将上边距设置为与图标相等的尺寸，将文本定位在图标的下方：

```
div.item div.itemName{  
    margin-top: 48px;  
    font: 10px verdana, arial, helvetica;  
    text-align: center;  
}
```

2. 使用 CSS 设置样式

CSS 执行的第二个工作是为元素加上视觉样式。文件夹中这些条目所使用的图片是通过类名来分配的，例如：

```
div.folder{  
    background:  
        transparent url(images/folder.png)  
        top left no-repeat;  
}  
  
div.file{  
    background:  
        transparent url(images/file.png)
```

```
    top left no-repeat;  
}  
  
div.special{  
  
background:  
    transparent url(images/folder_important.png)  
  
    top left no-repeat;  
}  

```

图标样式的background属性设置为：不重复显示图标，定位在元素的左上方，透明显示。（图 2-2 是在Firefox下的显示效果。IE对于透明.png图片的处理存在着很多bug，不过也有一些不太完善的变通解决方法。即将发布的IE 7 已经修正了这些bug。如果你需要跨浏览器显示透明图片，我们建议你使用.gif图片。）

个别的条目声明了两个样式类：通用的条目定义它们在容器中的布局，第二个指定它所用到的图标，例如：

样式中的所有图片都是使用 CSS 来设置的背景图片。标题栏的样式使用只有一个像素宽、与标题栏等高的图片在水平方向上重复来实现：

```
div.titlebar{  
  
background-color: #0066aa;  
  
background-image: url(images/titlebar_bg.png);  
  
background-repeat: repeat-x;  
  
...  
}
```

代码清单 2-1 是这个 UI 组件的完整 HTML。

代码清单 2-1 window.html

```

<html>
<head>
<link rel='stylesheet' type='text/css'
      href='window.css' />
</head>
<body>
<div class='window'> ← 顶级窗口元素
    <div class='titlebar'>
        <span class='titleButton' id='close'></span>
        <span class='titleButton' id='max'></span>
        <span class='titleButton' id='min'></span>
    </div>
    <div class='contents'>
        <div class='item folder'>
            <div class='itemName'>Documents</div>
        </div>
        <div class='item folder'>
            <div class='itemName'>lost and found</div>
        </div>
        <div class='item folder'>
            <div class='itemName'>stuff</div>
        </div>
        <div class='item file'>
            <div class='itemName'>shopping list</div>
        </div>
        <div class='item file'>
            <div class='itemName'>things.txt</div>
        </div>
        <div class='item special'>
            <div class='itemName'>faves</div>
        </div>
        <div class='item file'>
            <div class='itemName'>chapter 2</div>
        </div>
    </div>
</body>
</html>

```

HTML 标记定义了文档的结构，而不是外观。它同时也定义了文档中可以应用样式的位置，例如类名、唯一的 ID，甚至是标签类型本身。阅读这段 HTML，我们可以看到每个元素和其他元素之间的包含关系，而不是最终的视觉效果。编辑样式表可以在保持文档原有结构的情况下大幅度改变文档的外观，就像图 2-2 中展示的那样。UI 组件完整的样式表展示在代码清单 2-2 中。

代码清单 2-2 window.css

```

div.window{
    position: absolute;
    overflow: auto;
    background-color: #eeeeff;
    border: solid #0066aa 2px;
    margin: 0px;
    padding: 0px;
    width: 420px;
    height: 280px;
}
div.titlebar{
    background-color: #0066aa;
    background-image:
        url(images/titlebar_bg.png);
    background-repeat: repeat-x;
}

```

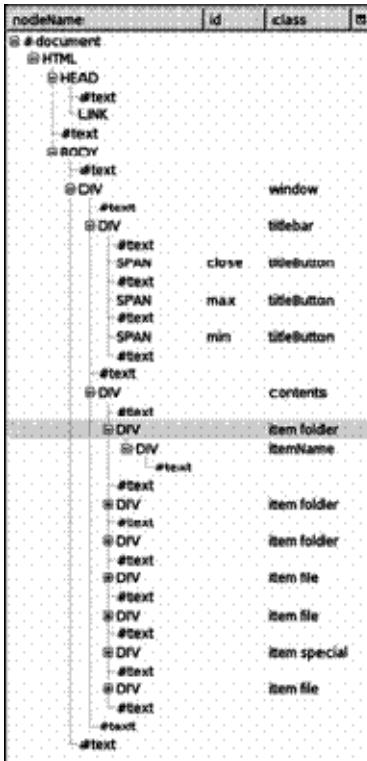
```

color:white;
border-bottom: solid black 1px;
width: 100%;
height: 16px;
overflow:hidden;
}
span.titleButton{
position: relative;
height: 16px;
width: 16px;
padding: 0px;
margin: 0px 1px; 0px 1px;
float:right;
}
③ 流动布局
span.titleButton#min{
background: transparent
url(images/min.png) top left no-repeat;
}
span.titleButton#max{
background: transparent
url(images/max.png) top left no-repeat;
}
span.titleButton#close{
background: transparent
url(images/close.png) top left no-repeat;
}
div.contents {
background-color: #e0e4e8;
overflow: auto;
padding: 2px;
height:240px;
}
div.item{
position : relative;
height : 64px;
width: 56px;
float: left;
}
color : #004488;
font-size: 18;
padding: 0px;
margin: 4px;
}
div.item div.itemName {
margin-top: 48px;
font: 10px verdana, arial, helvetica;
text-align: center;
}
④ 文本占位符
div.folder{
background: transparent
url(images/folder.png) top left no-repeat;
}
div.file{
background: transparent
url(images/file.png) top left no-repeat;
}
div.special{
background: transparent
url(images/folder_important.png)
top left no-repeat;
}

```

我们已经考察了在样式表中调整个别元素外观的一些技巧。在这里我们突出显示了更多的几个属性，目的是展示 CSS 可以应用于非常广泛的领域，包括设置以下内容：屏幕上的占位符①、元素的底纹②、元素的排列方式③和文字伴随图像的方式④。

CSS是Web开发者基础工具箱中的重要部分。正如我们在这里所展示的，它既适用于建造Ajax应用需要的用户界面，也适用于使用更加面向设计的方法来建造的静态手册风格（brochure-style）的网站。



2.4 用 DOM 组织视图

DOM 能够为 JavaScript 引擎公开文档（网页）。通过使用 DOM，可以采用编程方式操作文档的结构，如图 2-3 所示。当编写 Ajax 应用时，这是一种特别有用的能力。在传统 Web 应用中，我们通常使用来自服务器的新的 HTML 流来刷新整个页面，并通过提供新的 HTML 来重新定义用户界面。而在 Ajax 应用中，用户界面的更新主要是使用 DOM 来完成的。Web 页面中的 HTML 标签被组织成一个树状结构。树的根节点是标签，它代表这个文档。在它内部的标签代表文档的主体部分，是可见的文档结构的根节点。在文档主体之内，有表格、段落、列表以及其他类型的标签，每个标签之中还可能有其他标签。

图 2-3 DOM 以树结构表现一个 HTML 文档，每一个元素表现 HTML 标记中的一个标签

Web 页面的 DOM 表示也是一个树状结构，由元素或节点组成，节点还可能包含很多的子节点。JavaScript 引擎通过全局变量 `document` 公开当前 Web 页面的根节点，这个变量是所有 DOM 操作的起点。DOM 元素通过 W3C 规约来定义。它有一个父元素，没有或者有多个子元素，有任意多个作为关联数组来存储的属性（也就是说，使用 `width` 或 `style` 这样的文本形式的键，而不是使用数字索引）。图 2-3 展示了代码清单 2-2 中文档一个抽象结构，这个图是用 Mozilla DOM 检查器工具生成的（详见附录 A）。

DOM 中元素之间的关系可以看作是 HTML 清单的镜像。这种关系是双向的，修改 DOM 将会改变 HTML 标记，随之会反映在页面的显示上。

这提供了一种高层视图，使得我们对于 DOM 看起来像什么有一个直观的了解。在后续的小节中，我们将考察 DOM 为 JavaScript 解释器公开了哪些接口，并且学会如何使用它。

2.4.1 使用 JavaScript 操作 DOM

在任何应用中，我们都需要在用户的使用过程中改变用户界面，为用户执行的操作和完成的进度提供反馈。这些反馈包括修改单个元素的标签或颜色、弹出临时的对话框、使用一组全新的 UI 组件替换大部分的屏幕内容等等。到目前为止，最常见的方式就是，通过提供给浏览器一段声明式的 HTML 来构造 DOM 树（换句话说，也就是编写 HTML 页面）。

我们在代码清单 2-2 和图 2-3 中显示的文档有点太大和太复杂了，还是从小的步骤开始做 DOM 操作吧。假设我们要向用户显示友好的问候。当页面第一次加载的时候，我们并不知道他的名字，因此想要在稍后

修改页面的结构，以添加上用户的名字，通过以编程方式操作 DOM 节点可以做这件事。代码清单 2-3 展示了这个简单页面初始的 HTML 标记。

代码清单 2-3 Ajax 的“hello”页面

```
<html>
<head>
<link rel='stylesheet' type='text/css'
      href='hello.css' />           ←①到样式表的链接。
<script type='text/javascript'
       src='hello.js'></script>  ②到 JavaScript 的链接。
</head>
<body>
<p id='hello'>hello</p>
<div id='empty'></div>  ③空元素。
</body>
```

我们添加了两个到外部资源的引用：CSS①和包含 JavaScript 代码的文件②。我们还声明了带 ID 的空元素③，以后可以通过编程方式向其中添加更多的元素。

我们来看一下链接到的资源。通过修改字体和颜色，样式表为区分清单中不同类别的条目定义了一些简单的样式（代码清单 2-4）。

代码清单 2-4 hello.css

```
.declared{
  color: red;
  font-family: arial;
  font-weight: normal;
  font-size: 16px;
}
.programmed{
  color: blue;

  font-family: helvetica;
  font-weight: bold;
  font-size: 10px;
}
```

我们定义了两个样式来描述最初的 DOM 节点（样式的名称可以是任意的，我们取这两个名字是为了使例子容易理解，当然也可以给它们取名为 fred 和 jim）。这些样式类都没有在 HTML 中用到，但是我们将以编程方式将它们应用在元素上。代码清单 2-5 显示了伴随代码清单 2-4 的 Web 页面的 JavaScript。当加载文档时，我们将以编程方式为一个已有的节点设置样式，并且以编程方式创建一些新的 DOM 元素。

代码清单 2-5 hello.js

```

window.onload=function(){
    var hello=document.getElementById('hello');      ←通过 ID 找到元素。
    hello.className='declared';

    var empty=document.getElementById('empty');
    addNode(empty,'reader of');
    addNode(empty,'Ajax in Action!');

    var children=empty.childNodes;
    for (var i=0;i<children.length;i++){
        children[i].className='programmed';
    }

    empty.style.border='solid green 2px';           |直接为节点设置样式。
    empty.style.width='200px';
}

function addNode(el,text){
    var childEl=document.createElement("div");   ← 创建新元素
    el.appendChild(childEl);
    var txtNode=document.createTextNode(text);    ← 创建文本元素
    childEl.appendChild(txtNode);
}

```

JavaScript 代码比 HTML 或样式表更加复杂。代码的入口点是 `window.onload()` 函数，一旦整个页面加载完毕，浏览器就会调用这个函数。这个时候 DOM 树已经建造好，可以对它进行操作。代码清单 2-5 利用一些 DOM 操作方法来改变 DOM 节点的属性，显示和隐藏某些节点，甚至在运行时创建一些新节点。我们这里没有覆盖所有的 DOM 操作方法（可以查看“资源”一节获得所有的 DOM 操作方法）。在下面几个小节中，我们将深入讨论一些最有用的方法。

2.4.2 寻找 DOM 节点

用 JavaScript 操作 DOM 的第一件事就是找到要修改的元素。前面已经提到，我们开始只能得到根节点的一个引用，它保存在全局变量 `document` 中。DOM 中的每一个节点都是 `document` 的子节点（或孙节点、曾孙节点等等），但是要在大型的复杂文档中，一步一步地缓慢搜寻是件体力活。幸运的是，我们可以走一些捷径。最常用的方法就是给元素附加唯一的 ID。在代码清单 2-5 的 `onload()` 函数中，我们想要寻找两个元素：段落元素，我们为它设置样式；空的

标签，我们为它添加内容。如你所见，已经在 HTML 中为它们附加了唯一的 ID 属性，即：

和

任何一个 DOM 节点都可以分配一个 ID，用来在程序中通过一个函数调用获得这个节点的引用，无论它在文档中的什么位置：

```
var hello=document.getElementById('hello');
```

注意，这是 `Document` 对象的一个方法。在一个如上所述的简单情况中（以及在很多复杂的情况下），可以通过 `document` 访问当前的 `Document` 对象。如果你使用了 `Iframe`（我们将会在后面讨论），那么可能需要跟踪多个 `Document` 对象，并确定正在查询的是哪个 `Document` 对象。

在一些情况下，我们确实需要一步一步地搜索 DOM 树。因为 DOM 节点是以树形结构来组织的，每一个 DOM 节点都只有不多于一个的父节点，但是可以有任意多个子节点。可以通过 `parentNode` 和 `childNodes` 来访问它们。`parentNode` 返回另外一个 DOM 节点，而 `childNodes` 返回一个 JavaScript 节点数组，可以对其遍历，即：

```
var children=empty.childNodes;  
for (var i=0;i<CHILDREN.LENGTH;i++){< SPAN>  
...  
}
```

即便在文档中的某个节点上没有附加唯一 ID，我们仍有第三种方法可以方便地定位节点。那就是，使用 `getElementsByName()` 方法，基于 HTML 标签的类型搜索 DOM 节点。例如，`document.getElementsByTagName("UL")` 会返回一个包含文档中所有标签的数组。

这些方法对于操作那些我们几乎无法控制的文档[\[2\]](#)来说是很有用的。作为通常的规则，使用 `getElementById()` 要比使用 `getElementsByName()` 更加安全，因为前者对于文档结构和顺序的假设更少一些，这样文档结构可以独立于代码而变化。

2.4.3 创建 DOM 节点

除了重组已有的 DOM 节点，一些情况下还需要创建新的节点并把它们添加到文档中（例如，在运行时创建消息框）。DOM 的 JavaScript 实现也为我们提供了相应的方法。

我们再次考察一下代码清单 2-5 的例子代码。ID 为 '`empty`' 的 DOM 节点实际上是从空节点开始的。当加载这个页面的时候，我们为它动态创建了一些内容。`addNode()` 函数使用了标准的 `document.createElement()` 和 `document.createTextNode()` 方法。`createElement()` 可以用来创建任何 HTML 元素，带有一个标签类型参数，例如：

```
var childEl=document.createElement("div");  
  
createTextNode() 创建了一个代表一段文本的 DOM 节点，这样的节点通常嵌入在标题、div、段落以及列表条目等标签之中。  
  
var txtNode=document.createTextNode("some text");
```

DOM 标准将文本节点和代表 HTML 元素的节点区别开来，分别对待。不能直接对文本节点应用样式，因此也就少占用一些内存。不过，我们仍然可以通过包含这个文本节点的 DOM 元素来设置该节点所表示的文本的样式。

无论何种类型的节点一旦被创建，都必须将它附加到文档之上，然后才能在浏览器窗口中见到它。DOM 节点的 `appendChild()` 方法就是用来完成这个工作的：

```
el.appendChild(childEl);
```

`createElement()`、`createTextNode()` 和 `appendChild()` 这三个方法为我们提供了向一个文档中添加新的结构所需要的一切。完成了这个工作，我们通常还想要以一种适当的方式为新的结构设置样式。

我们来考察一下如何做这件事。

2.4.4 为文档增加样式

到目前为止，我们已经考察了使用 DOM 来操作文档的结构（一个元素如何被另外一个元素所包含，诸如此类）。这使得我们可以有效地改造在静态 HTML 中声明的结构。DOM 还提供了另外一类方法，允许以编程方式修改元素的样式和改造定义在样式表中的结构。

通过 DOM 操作，Web 页面上的每一个元素都可以拥有多种视觉样式，例如位置、高度和宽度、颜色、边框和空白。尽管分别修改每一个属性可以更加精细地控制元素的外观，但是这样做是很单调乏味的。幸运的是，Web 浏览器为我们所提供的 JavaScript 绑定除了提供底层接口以便精确操作之外，也允许通过 CSS 类来为元素设置一致的样式。我们来逐个考察一下。

1. `className` 属性

CSS 提供了一种简明的方式来将预先定义的、可重用的样式应用到文档中。当我们为在代码中创建的元素设置样式的时候，也可以通过设置 DOM 节点的 `className` 属性来利用 CSS。例如，下面一行为一个节点设置了通过 `declared` 类定义的显示规则：

```
hello.className='declared';
```

其中 `hello` 是到一个 DOM 节点的引用。这提供了一种容易和紧凑的方法，来为一个节点同时分配很多的 CSS 规则，并且可以通过样式表来管理复杂的样式。

2. `style` 属性

在其他一些情况下，我们想要细粒度地改变特定元素的样式，也许仅仅是作为已经通过 CSS 应用的样式的补充。

DOM 节点还包含了一个名为 `style` 的关联数组，其中包含了节点样式的全部细节。正如图 2-4 所示，DOM 节点的样式中通常包含有大量的条目。鲜为人知的是，为节点分配 `className` 也会改变 `style` 数组的值。

The screenshot shows the DOM Inspector interface with two main panes. The left pane, titled 'Document - DOM Nodes', displays a tree structure of nodes. A specific node under a 'SPAN' element is selected, highlighted with a gray background. The right pane, titled 'Object - Computed Style', lists the properties and their values for this selected node. The properties listed include: background-attachment (scroll), background-color (transparent), background-image (none), background-repeat (repeat), border-bottom-color (rgb(255, 0, 0)), border-bottom-style (none), border-bottom-width (0px), border-collapse (separate), border-left-color (rgb(255, 0, 0)), border-left-style (none), border-left-width (0px), border-right-color (rgb(255, 0, 0)), border-right-style (none), border-right-width (0px), border-top-color (rgb(255, 0, 0)), border-top-style (none), border-top-width (0px), bottom (auto), caption-side (top), clear (none), clip (auto), and color (rgb(255, 0, 0)). The 'Value' column for each property shows the computed value, often differing from the original CSS declaration.

图 2-4 在 DOM 检查器中检查 DOM 节点的 `style` 属性。大部分的值都不是由用户显式设置的，而是由呈现引擎自己设置的。注意旁边的滚动条：我们只看到了全部已计算样式列表的大约四分之一的内容

`style` 数组也可以直接操作。在为 `empty` 节点中的条目设置样式后，我们可以给它加一个边框：

```
empty.style.border="solid green 2px";  
empty.style.width="200px";
```

我们也可以很容易地仅仅声明一个 `box` 类，然后通过修改 `className` 属性来应用样式，但是这种方法在某些特定环境下用起来更加便捷，并且它还允许以编程方式构造字符串。例如，我们想要以像素的精度自由地改变元素的大小，如果为此预先定义从 1 到 800 像素每个宽度的样式，那显然是低效和笨拙的。

使用上面的方法，我们可以创建新的 DOM 元素，并且为它们设置样式。在内容处理技术的工具箱中，还有另外一个很有用的工具，它使用一种略微不同的方法来通过程序编写 Web 页面，这就是 `innerHTML` 属性，我们通过考察这个属性来结束本节。

2.4.5 捷径：使用 `innerHTML` 属性

到目前为止，我们所描述的方法通过使用 DOM API 提供了对页面结构进行低层次控制的能力。然而，对于创建一个文档来说，`createElement()` 和 `appendChild()` 提供的 API 相当冗长，它最适合于以下场合：文档遵循一种有规律的结构来创建，这种结构可以编码为一种算法。所有流行的 Web 浏览器的 DOM 元素都支持称作 `innerHTML` 的属性，允许以一种非常简单的方式来为元素分配任意的内容。`innerHTML` 是一个字符串，以 HTML 标记的形式表示一个节点中的内容。例如，我们可以使用 `innerHTML` 来重写 `appendChild()` 函数，就像这样：

```
function addListItemUsingInnerHTML(el, text){  
    el.innerHTML += "  
        "+text+"  
    ";  
}
```

元素和嵌入的文本节点可以使用单条语句来添加。另一个需要注意的地方是，这里是使用`+=` 操作来添加属性的，而不是直接赋值。如果使用 `innerHTML` 来删除一个节点，我们需要提取和解析这个字符串。`innerHTML` 更加简捷，适合于像这个例子一样相对简单的应用。如果应用程序需要大量修改一个节点，前面展示的操作 DOM 节点的方法提供了更好的机制。

到这里我们已经讲述了 JavaScript、CSS 和 DOM。它们在动态 HTML 这个名字首次出现时就在一起使用了。正如在本章前面的介绍中提到的，Ajax 使用了很多动态 HTML 的技术，但是 Ajax 令人耳目一新之处在于它向这个组合中加入了新的成分。在下面小节中，我们来考察一下 Ajax 与 DHTML 的区别所在——在用户使用的同时与服务器通信的能力。

2.5 使用 XML 技术异步加载数据

在使用应用的时候，特别是独占式应用，作为工作流的一部分，用户会持续地和应用进行交互。第 1 章中讨论了保持应用响应能力的重要性。如果执行一个长的后台任务使得界面上的一切东西都被锁住，那么就打断了用户的使用。我们讨论了使用异步方法调用的优点，它可以在执行这样的长任务时改善用户界面的响应能力。我们注意到，因为存在网络延迟，那些发到服务器的调用都应该被当作长任务来处理。在

基本的 HTTP 请求一响应模型下，不大可能做到这件事。传统的 Web 应用依赖于整个页面的重新加载，每一个发到服务器的调用会频繁地打断用户的使用。

我们不得不接受文档请求被阻塞直到服务器返回响应这样的现实。尽管如此，我们还是有很多办法，使得服务器请求对于用户来说看起来像是异步的，这样用户仍然可以继续工作。最早尝试提供这种后台通信能力的是使用 IFrame。后来，XMLHttpRequest 对象提供了更加清晰和强大的功能。我们在这里考察一下这两个技术。

2.5.1 IFrame

在第 4 版的 Netscape Navigator 和微软 IE 浏览器中，DHTML 为 Web 页面引入了灵活的、可编程的布局。作为老的 HTML 帧集的自然扩展，出现了 IFrame。I 代表的是内嵌（inline），意味着它是另外一个文档布局的一部分，而不是像在帧集中一样与之并列。一个 IFrame 表现为 DOM 树的一个元素，这意味着只要页面可见，我们就可以移动 IFrame、改变它的大小或者将它完全隐藏起来。关键的突破在于人们意识到可以设置一个 IFrame 的样式，使得它完全不可见。这使得不干扰用户体验，以后台方式获取数据成为了可能。突然之间，我们有了一种与服务器进行异步通信的机制，虽然这不过是一种 hack 式的临时解决方案。图 2-5 展示了这种方法背后的事件顺序。

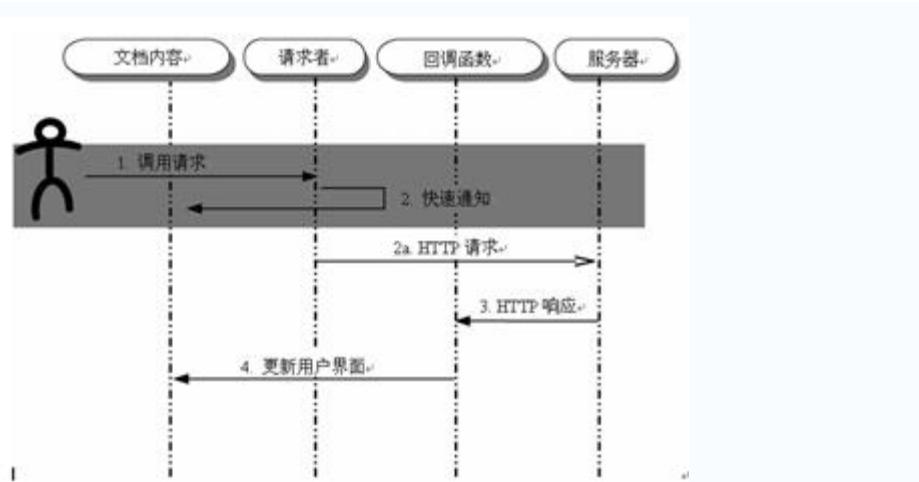


图 2-5 Web 页面中异步通信的事件顺序。用户操作触发了一个隐藏的请求对象

(一个 IFrame 或者 XMLHttpRequest 对象)向服务器发送一个异步调用。方法非常迅速地返回，只会将用户界面阻塞很短的时间（在图中用阴影区域的高度来表示）。在一个回调函数中解析服务器的响应，并据此更新用户界面

和其他 DOM 元素一样，IFrame 可以在页面的 HTML 中声明，也可以使用 `document.createElement()` 通过程序生成。在一种简单的情况下，我们仅仅想要使用一个不可见的 IFrame 来加载数据。我们可以将这个 IFrame 作为文档的一部分来声明，然后使用 `document.getElementById()` 获得一个它的引用，如代码清单 2-6 所示。

代码清单 2-6 使用 IFrame

```
<html>
<head>
<script type='text/javascript'>
window.onload=function(){
    var iframe=document.getElementById('dataFeed');
    var src='datafeeds/mydata.xml';
    loadDataAsynchronously(iframe,src);
}
function loadDataAsynchronously(iframe,src){
    //...do something amazing!
}
</script>
</head>
<body>
<!--
...some visible content here...
-->
<iframe
    id='dataFeed'
    style='height:0px;width:0px;=''
>
</iframe>
</body>
</html>
```

因为将它的宽度和高度都设置为 0 像素，这个 IFrame 实际上是不可见的。我们也可以将样式设置为 `display:none`，但是某些浏览器会据此进行优化，导致实际上不会去加载相关的文档！还要注意的是，我们需要等到文档加载完成之后才能访问 IFrame，因此我们在 `window.onload` 事件处理函数中调用 `getElementById()` 方法。另一种方法是通过程序按需生成 IFrame，如代码清单 2-7 所示。这种方法的额外好处是可以将所有与请求数据相关的代码放在一起，而不是需要在脚本和 HTML 之间保持 DOM 节点唯一 ID 的同步。

代码清单 2-7 创建 IFrame

```
function fetchData(){
    var iframe=document.createElement('iframe');
    iframe.className='hiddenDataFeed';
    document.body.appendChild(iframe);
    var src='datafeeds/mydata.xml';
    loadDataAsynchronously(iframe,src);
}
```

与前面的例子相似，仍然使用 `createElement()` 和 `appendChild()` 来修改 DOM。如果我们严格遵循这种方法，随着应用的持续运行，最终会创建出大量的 IFrame。我们要么必须在用完的时候销毁这些 IFrame，要么就得实现一个类似于对象池（pooling）的机制。

我们在第 3 章将要介绍到的设计模式能够帮助我们实现健壮的对象池、队列以及其他用来保证大型应用流畅运转的机制，稍后我们会更加深入地探讨这个话题。在此之前，我们将注意力转移到另外一套可以后台方式向服务器发送请求的技术。

2.5.2 XMLHttpRequest 对象

正如刚才看到的，我们可以使用 IFrame 后台方式请求数据，但是这从本质上来说不过是一种 hack 式的临时解决方案。最初引入 IFrame 的设计意图是在页面上显示可见的内容，这种用法歪曲了这个意图。在流行的 Web 浏览器的更新版本中，引入了专门为异步数据传输而设计的对象，我们将会看到，它比 IFrame 用起来要方便得多。

XmlDocument 和 XMLHttpRequest 对象并不是 Web 浏览器中 DOM 的标准扩展，它们只是碰巧得到了多数浏览器的支持。它们的设计目标很明确，就是用来后台方式获取数据，这使得发出异步调用的业务使用起来非常流畅。两个对象都是源自微软私有的 ActiveX 组件，可以在 IE 浏览器中作为 JavaScript 对象来访问。其他的浏览器则依照相似的功能和 API 调用实现了自己的原生对象。两个对象执行的功能很相似，不过 XMLHttpRequest 可以更加精细地对请求进行控制。在本书中，我们将主要使用 XMLHttpRequest，在这里我们简要介绍一下 XmlDocument，以便你了解这个对象与 XMLHttpRequest 有哪些不同。代码清单 2-8 展示了一段简单创建 XmlDocument 对象的代码。

代码清单 2-8 `getXmlDocument()` 函数

```
function getXMLDocument(){
    var xDoc=null;
    if (document.implementation
        && document.implementation.createDocument){
        xDoc=document.implementation
        .createDocument("", "", null);    // Mozilla/Safari
    }else if (typeof ActiveXObject != "undefined"){
        var msXmlAx=null;
        try{
            msXmlAx=new ActiveXObject
            ("Msxml2.DOMDocument");    // 较新版本的 IE 浏览器
        }catch (e){
            msXmlAx=new ActiveXObject
            ("Msxml.DOMDocument");    // 较老版本的 IE 浏览器
        }
        xDoc=msXmlAx;
    }
}
```

```
if (xDoc==null || typeof xDoc.load=="undefined"){
    xDoc=null;
}
return xDoc;
}
```

在大多数现代浏览器中，这个函数都能返回一个具有相同 API 的 XmlDocument 对象，尽管在不同的浏览器中创建文档的方式有很大不同。

这段代码检查文档对象是否支持创建一个原生的 XmlDocument 对象所需的 `implementation` 属性（在最近的 Mozilla 和 Safari 浏览器中都可以找到这个属性）。如果没有找到，它将测试浏览器是否支持 ActiveX 对象（只有微软的浏览器才能够支持），如果支持，它将尝试定位一个合适的对象。这段脚本优先使用较新一些的第二版 MSXML 库。

注意 检查浏览器的厂商和版本号信息，并且使用这些信息来开发用于不同浏览器的分支代码，是一种很常见的做法。在我们看来，这种做法容易导致错误，因为它无法预期浏览器的未来版本，并且还会将有能力执行这段脚本的浏览器排除在外。在 `getXmlDocument()` 函数中，我们没有尽力去猜测浏览器的版本，而是直接检查特定的对象是否可用。这个方法也称作**对象检测 (object detection)**，可以更容易地支持浏览器未来的版本，以及那些我们没有明确测试过的不常见的浏览器，通常这会使得代码更加健壮。

代码清单 2-9 的代码采用与前面的代码类似的方式获得 XMLHttpRequest 对象，不过略微简单一些。

代码清单 2-9 `getXmlHttpRequest()` 函数

```
function getXMLHttpRequest() {
    var xRequest=null;
    if (window.XMLHttpRequest) {
        xRequest=new XMLHttpRequest();    //—Mozilla/Safari
    }else if (typeof ActiveXObject != "undefined") {
        xRequest=new ActiveXObject
        ("Microsoft.XMLHTTP");    //—IE
    }
    return xRequest;
}
```

同样地，我们使用对象检测来测试是否支持原生的 XMLHttpRequest 对象，如果不支持，再测试是否支持 ActiveX 对象。在两者都不支持的浏览器中，我们简单地返回 null。如何更加优雅地处理失败的情况呢？这个问题我们留到第 6 章再来更加详细地探讨。

我们已经创建了向服务器发送请求的对象，下面我们用它来做什么呢？

2.5.3 向服务器发送请求

通过 XMLHttpRequest 对象向服务器发送请求是一件相当直接的事情。我们需要做的所有事情就是给它传递一个服务器页面的 URL，这个页面将生成数据。就像下面这样：

```
function sendRequest(url,params,HttpMethod){  
    if (!HttpMethod){  
        HttpMethod="POST";  
    }  
    var req=XMLHttpRequest();  
    if (req){  
        req.open(HttpMethod,url,true);  
        req.setRequestHeader  
            ("Content-Type",  
             "application/x-www-form-urlencoded");  
        req.send(params);  
    }  
}
```

XMLHttpRequest 支持大量的 HTTP 调用语义，包括用来动态生成页面的可选查询字符串参数（你可能已经知道这些 CGI 参数、Form 参数或者 ServletRequest 参数，取决于服务器端开发背景）。在考察请求对象如何支持这些功能之前，我们先来快速回顾一下 HTTP 的基础知识。

HTTP 快速入门

HTTP 对于因特网而言可谓是无处不在，以至于我们通常都对它熟视无睹。在编写传统的 Web 应用时，我们近距离接触 HTTP 协议的地方通常是定义一个超链接或者为一个表单设置 method 属性。而对于 Ajax 而言，我们可以深入那些协议的底层细节，这使得我们可以做一些惊人的事情。

浏览器和 Web 服务器之间的 HTTP 事务包括浏览器发起的一个请求和随后服务器返回的一个响应（其中也包括执行 Web 开发人员编写的聪明绝顶、激动人心的代码，这一点毫无疑问）。请求和响应本质上来说都是文本流，客户端和服务器将它们解释为首部信息和紧随其后的主体部分。你可以将首部信息想像为写在信封上的地址栏，而主体部分是信封中的信件。首部信息简单地指示接收方应该如何处理信件的内容。

一个 HTTP 请求主要由首部信息和可能包含一些数据或参数的主体部分组成。响应则通常包含返回页面的 HTML 标记。Mozilla 浏览器包含了一个很有用的工具，叫做 LiveHTTPHeaders（见本章“资源”一节

以及附录 A)。我们来查看一下在浏览器工作时，那些请求和响应的首部信息的内容。打开 Google 的首页，看看在底层发生了些什么事情。

我们发送的第一个请求包含有这样的首部信息：

GET / HTTP/1.1

Host: www.google.com

User-Agent: Mozilla/5.0

(Windows; U; Windows NT 5.0; en-US; rv:1.7)

Gecko/20040803 Firefox/0.9.3

Accept: text/xml,application/xml,

application/xhtml+xml,text/html;q=0.9,

text/plain;q=0.8,image/png,*/*;q=0.5

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7

Keep-Alive: 300

Connection: keep-alive

Cookie: PREF=ID=cabd38877dc0b6a1:TM=1116601572

:LM=1116601572:S=GD3SsQk3v0adtSBP

第一行告诉我们使用的是哪个 HTTP 方法。大部分 Web 开发者都很熟悉 GET，这是用来获取文档的；还有 POST，这是用来提交 HTML 表单的。万维网联盟（W3C）的规约还包括了一些其他的通用方法，包括 HEAD，用来获取一个文件的首部信息；PUT，用于向服务器上传文档；DELETE，用来删除服务器上的文档。后续的首部信息是用来进行沟通的，客户端告诉服务器它所能支持的内容类型、字符集等等。因为我之前曾经访问过 Google，它还发送了一个 cookie，这段简短的消息告诉 Google 我是谁。

下面显示的是响应的首部信息，也包含了大量的信息：

HTTP/1.x 302 Found

Location: http://www.google.co.uk/cxfer?c=PREF%3D:

TM%3D1116601572:S%3DzFxPsBpXhZzknVMF&prev=/

```
Set-Cookie: PREF=ID=cabd38877dc0b6a1:CR=1:TM=1116601572:  
LM=1116943140:S=fRfhD-u49xp9UE18;  
expires=Sun, 17-Jan-2038 19:14:07 GMT;  
path=/; domain=.google.com  
  
Content-Type: text/html  
  
Server: GWS/2.1  
  
Transfer-Encoding: chunked  
  
Content-Encoding: gzip  
  
Date: Tue, 24 May 2005 17:59:00 GMT  
  
Cache-Control: private, x-gzip-ok=""
```

第一行指示了响应的状态。302 响应意味着重定向到另外一个页面。服务器还为此次会话发回了另外一個 cookie。此外，还声明了响应的内容类型（也就是 MIME 类型）。这个重定向指令引起客户端发送了一个新的请求，随后得到了第二个响应，带有下列首部信息：

```
HTTP/1.x 200 OK  
  
Cache-Control: private  
  
Content-Type: text/html  
  
Content-Encoding: gzip  
  
Server: GWS/2.1  
  
Content-Length: 1196  
  
Date: Tue, 24 May 2005 17:59:00 GMT
```

状态码 200 表示成功，用以显示的 Google 首页附加在响应的主体部分。Content-type 告诉浏览器，内容的类型是 html。

sendRequest() 方法包含的第二个和第三个参数都是可选参数，大部分情况下都用不上。默认使用 POST 方法来获取资源，在请求的主体部分不需要传递任何参数。

清单中的代码对请求进行设置后，会立即将控制权返回给我们，与此同时网络和服务器则忙着执行它们自己的任务。这对于提高响应能力很有好处，但是我们要怎样才能知道请求完成了呢？

2.5.4 使用回调函数监视请求

处理异步通信的第二个部分是在代码中设置一个入口点，以便在调用结束的时候可以获取结果。这通常是通过分配一个回调函数来实现的，也就是说，在未来的某个不确定时刻，当结果返回的时候，将会执行这一段代码。我们在代码清单 2-9 中看到的 `window.onload` 函数就是一个回调函数。

回调函数非常适合用于大多数现代 UI 工具箱中的事件驱动的编程方法。按下键盘、点击鼠标等等，这些事件都将会在未来某个无法预测的时刻发生，程序员预见到了这一点，并且为这些事件写好了处理函数。在用 JavaScript 编写用户界面的事件处理代码时，我们将函数分配给 `onkeypress`、`onmouseover` 或者对象上的类似属性。在为服务器请求的回调函数编写代码时，我们见到过称作 `onload` 和 `onreadystatechange` 的类似属性。

IE 和 Mozilla 都支持 `onreadystatechange` 回调函数，所以我们可以放心地使用（Mozilla 还支持 `onload`，这更加直接一点，但是它不能像 `onreadystatechange` 那样给我们所需要的信息）。一个简单的回调处理函数展示在代码清单 2-10。

代码清单 2-10 使用回调处理函数

```
var READY_STATE_UNINITIALIZED=0;  
var READY_STATE_LOADING=1;  
var READY_STATE_LOADED=2;  
var READY_STATE_INTERACTIVE=3;  
var READY_STATE_COMPLETE=4;  
  
var req;  
  
function sendRequest(url,params,HttpMethod){  
    if (!HttpMethod){  
        HttpMethod="GET";  
    }  
    req=XMLHttpRequest();  
    if (req){  
        req.onreadystatechange=onReadyStateChange;
```

```

req.open(HttpMethod,url,true);

req.setRequestHeader
("Content-Type", "application/x-www-form-urlencoded");

req.send(params);

}

}

function onReadyStateChange(){

var ready=req.readyState;

var data=null;

if (ready==READY_STATE_COMPLETE){

data=req.responseText;

}else{

data="loading...["+ready+"]";

}

//... do something with the data...


}

}

```

首先，在发送请求之前，我们修改了 `sendRequest()` 函数，告诉请求对象使用哪个回调函数。其次，我们定义了这个处理函数，并且给它取了一个缺乏想象力的名字 `onReadyStateChange()`。

`readyState` 可以取一系列数值。为了使得代码易于阅读，我们在这里给每一个数值分配了一个有描述性的变量名。目前的代码只对数值 4 感兴趣，它表明请求已经完成。

要注意的是，我们将请求对象声明为一个全局变量。在这里，这有助于事情保持简单，以便更好地理解 XMLHttpRequest 对象的工作原理。但是，如果我们试图同时发送多个请求，这样就会带来一些麻烦。如何解决这个问题我们留到 3.1 节再讨论。现在将这些部分整合起来，从头到尾看看如何处理请求。

2.5.5 完整的生命周期

我们现在已经有了足够的信息，可以完成加载一个文档的全部生命周期，就如代码清单 2-11 中所示。我们初始化 XMLHttpRequest 对象，让它加载一个文档，然后使用回调处理函数来异步地对加载过程进行监视。在这个简单的例子中，我们定义了一个称作 `console` 的 DOM 节点，用来输出状态信息，以便得到下载过程的书面记录。

代码清单 2-11 完整的使用 XMLHttpRequest 加载文档的例子

2.6 Ajax 有何不同

CSS、DOM、异步请求和 JavaScript，这些就是 Ajax 所需要的全部组件。然而，即使将这些东西全部用上也并不意味着就是 Ajax，至少我们在本书中正在描述的场景是这样。

我们已经在第 1 章中讨论了传统 Web 应用和 Ajax 应用的区别，这里我们再回顾一下。在传统的 Web 应用中，用户的工作流通过服务器的代码来定义，用户从一个页面切换到另一个页面，正在进行的工作不时被重新加载整个页面而打断。重新加载页面时，用户无法继续工作。而在 Ajax 的应用中，工作流至少有一部分是在客户端应用中定义的，当用户正在工作的时候与服务器的通信悄悄地发生在后台。

在这两个极端之间，是广大的灰色地带。一个 Web 应用可能一方面遵循传统的方法交付一系列的离散页面，而与此同时，每个页面又可能巧妙地使用着 CSS、DOM、JavaScript 和异步请求来使用户与页面的交互更加流畅。当然，为了切换到下一个页面，这其中肯定免不了会有一些打断用户工作的页面刷新。也可能在工作流的某个位置，JavaScript 应用会显示一个类似于传统 Web 页面的弹出式窗口。Web 浏览器是一个灵活和宽容的环境，你可以在一个应用中混合使用 Ajax 和非 Ajax 的功能。

令 Ajax 显得与众不同的地方不是它所使用的技术本身，而是通过使用这些技术所带来的新的交互模型。我们所习惯的传统 Web 交互模型并不适合于独占式的应用，只有打破了这种交互模型，新的可能性才会慢慢浮现出来。

至少可以在两个层次上使用 Ajax，在传统的基于页面的方法和新型的 Ajax 方法之间有一些过渡方案。最简单的策略是开发基于 Ajax 的 UI 组件，它们大多是自包含的，使用少量的导入语句和脚本就可以添加到 Web 页面之中。股票行情、交互式日历、聊天窗口就是这样一类典型的 UI 组件。另外一个层次是将类似于应用的功能嵌入在类似于文档的 Web 页面中（图 2-6）。Google 的大部分 Ajax 应用（参见 1.3 节）都适

用于这种模型。例如，Google Suggest 的下拉列表和 Google Maps 的地图 UI 组件，它们都是以交互式组件的形式嵌入在页面之中。

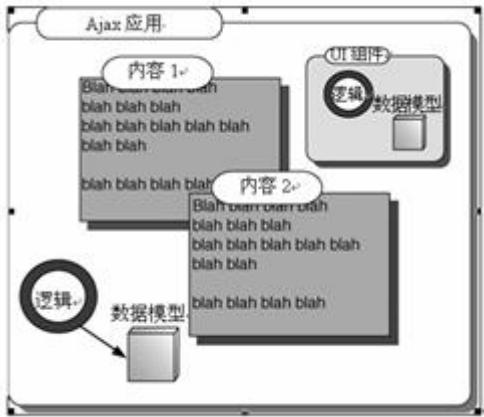


图 2-6 一个简单的 Ajax 应用工作起来仍然像是一个 Web 页面，交互功能嵌入在页面之中

如果想要以更加激进的方式应用 Ajax，我们还可以总结出另外一种模型，开发一个宿主应用，其中既可以容纳类似于应用的组件，也可以容纳类似于文档的组件（图 2-7）。这种方法更加类似于一个桌面应用、窗口管理器或者桌面环境。Google 的 GMail 适用于这个模型，个别的消息作为文档显示在交互式地类似于应用的架构中。

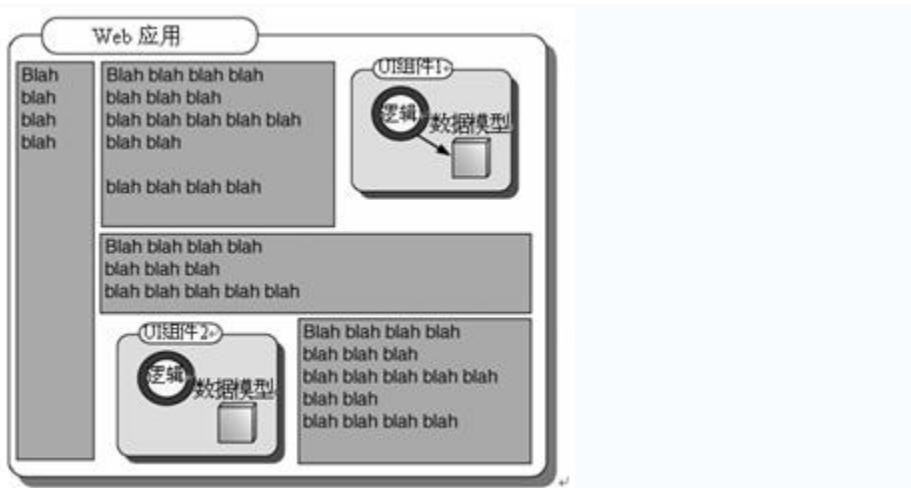


图 2-7 在一个更加复杂的 Ajax 应用中，整个应用就是一个交互式的系统，类似于文档的内容可以通过编程或者声明的方式来加载

在某种方式上，学习 Ajax 技术是很容易的部分。有趣的挑战在于学会在开发过程中如何很好地对它们进行综合应用。我们习惯于将 Web 应用看作是一组剧情故事板，我们使用一个预先确定的脚本让用户在这些

故事板中转来转去。而如果给Web应用加入一些类似于应用的功能，我们可以为用户提供对于业务领域问题更加细致的处理能力，这使得正在工作的用户感到更加自由。

为了从更大的灵活性中获得好处，必须要对我们的编程习惯多问几个为什么。HTML表单是用户输入信息的唯一方式吗？我们是不是要将所有的用户界面都声明为HTML？我们能不能在用户按下某个键或者移动鼠标时，与服务器进行联系，就像在传统的用户点击鼠标时所做的那样？在信息技术日新月异的今天，我们非常强调学习新的技能，但是放弃老的习惯至少也同样重要。

2.7 小结

在本章中，我们介绍了 Ajax 的四个技术基石。

JavaScript 是一种强大的通用编程语言，因为总是用来生成那些弹出窗口、行为古怪的回退按钮以及图片上的交替效果，因此有些声名狼藉。附录 B 包含了对这个语言一些特征的详细描述。但是通过这里的例子，你应该能够体会到，它确实可以真正地增强应用的可用性。

CSS 和 DOM 互为补充，为我们正在使用的用户界面提供了清晰的编程视图，同时保持了结构和视觉样式的分离。清晰的文档结构使得我们更容易以编程方式操作一个文档，保持职责分离对于开发大型 Ajax 应用非常重要，这一点我们在第 3 章和第 4 章中可以看到。

我们展示了如何使用 XMLHttpRequest 对象和更老的 XmlDocument 及 IFrame。当前围绕 Ajax 充斥着大量的市场炒作，将 XMLHttpRequest 赞誉为与服务器通信最时髦的方法。但是有些时候，IFrame 可以为我们提供恰好需要的一套不同的功能。对于两者都有所了解可以丰富你的工具箱。在本章中我们介绍了这些技术，也提供了一些例子。在第 5 章中，我们将对客户/服务器通信进行更加详细的讨论。

最后，我们考察了综合应用这些 Ajax 技术基石的方式，以便得到比将它们简单相加更好的效果。你既可以使用少量 Ajax，将一些吸引人的 UI 组件添加到静态页面中；也可以大胆地使用 Ajax 创建一个完整的用户界面，在其中包含一些静态内容。当然，后者需要你编写大量的 JavaScript 代码，而且代码还要能够长时间无故障运行。这需要我们用一种与以往不同的方式来编码，更多地考虑代码的可靠性、可维护性和灵活性。在下一章我们会考察这些问题，为大规模 Ajax 应用的代码库引入必需的秩序。

2.8 资源

想要更加深入地理解 CSS，我们推荐你访问 CSS Zen Garden (www.csszengarden.com)，这个网站仅靠使用不同的 CSS 就为自己创造出了丰富多彩的视觉效果。

Eric Meyer 也撰写了大量关于 CSS 的图书和文章，可以访问他的网站，位于 www.meyerweb.com/eric/css/。Blooberry (www.blooberry.com) 是另一处获取 CSS 信息的极好网站。

早期使用 IFrame 的 Ajax 解决方案在这里有详细的描述：<http://developer.apple.com/internet/webcontent/iframe.html>。

Mozilla 的扩展 LiveHttpHeaders 可以在这里找到：<http://livehttpheaders.mozdev.org/>。

Danny Goodman 关于 JavaScript 的著作是 DOM 编程方面的基本参考，其中非常详细地介绍了浏览器环境。*Dynamic HTML: The Definitive Reference* (O'Reilly, 2002 出版) 和 *JavaScript Bible* (John Wiley, 2004 出版)。

W3Schools 的网站上包含了一些关于 JavaScript 的交互式教程，非常适合喜欢边学边做的初学者 (www.w3schools.com/js/js_examples_3.asp)。

第3章 使 Ajax 秩序井然

3.1 从混沌到秩序

我们将要使用的主要工具是重构 (refactoring)。重构的目的不是增加新的功能，而是重写代码使得它们更加清晰。更加清晰的代码本身就是一个令人满意的结果，但是除此之外，它还可以带来其他引人注目的好处，应该能够吸引那些在底线上苦苦挣扎、如履薄冰的开发人员。

通常，如果代码经过了良好的重构，那么为它增加、修改或者去掉某些功能就会更容易。这是因为代码很容易理解。相反，如果代码没有经过良好的重构，经常会出现，即使每一件事情都符合了当前的需求，但是开发团队仍然对于代码为何能够工作缺乏信心[1]。

需求的变化总是要求在很短时间内就能实现，这已经是大多数专业编程工作中的一种常态。重构可以保持代码清晰，使它易于维护，允许你毫无畏惧地面对和实现需求的变化。

在第 2 章的例子中，当将 JavaScript、HTML 和样式表分别移到单独的文件中时，我们实际上已经在做一些基本的重构工作了。然而，得到的 JavaScript 代码大约有 120 行，底层功能（例如向服务器发起请求）和专门处理列表对象的代码混在一起。随着项目渐渐变大，这一单独的 JavaScript 文件（在这个例子中，也是一个单独的样式表文件）很快就会变得难以维护。我们正在追求的目标是，创建小的、易读的、易修改的代码块，用来解决特定的问题，也就是我们常说的职责分离 (separation of responsibilities)。

重构的另一个动机是识别出通用的解决方案，并且按照这种特定的模式来重新组织代码。这样做本身同样可以使得代码更加令人满意，而且还可以带来非常实际的好处。我们下面就来考虑一下这个问题。

3.1.1 模式：创造通用的词汇表

遵循成熟模式的代码，更有可能得到满意的结果，这仅仅是因为在此之前别人已经做过这件事。与之相关的很多问题都已经经过了其他人的深思熟虑，并且如我们希望的那样，已经解决了。如果我们运气好，可能其他人已经开发出了解决这类特定问题的可重用框架。

这种行事方式有时候也称作设计模式 (design pattern)。模式的概念诞生于 20 世纪 70 年代，当时用来描述建筑规划问题的解决方案，近十年来，软件开发领域借用了这概念。服务器端的 Java 开发有很强的设计模式文化，微软最近也开始在.NET 框架中大力推动设计模式的应用。这个术语总是令人产生一种难以亲近的学院派的味道。而且，为了令老板和客户听起来印象深刻，设计模式总是被人有意滥用。尽管如此，从根本上来说，设计模式仅仅是用来描述在软件设计中解决特定问题的一种可重复的方法。值得注意的是，设计模式为抽象的技术解决方案命名，使得它们更加便于讨论和容易理解。

设计模式之所以对重构如此重要，是因为它使我们能够简洁地描述意图达到的目标。“我们将这些执行用户操作的代码封装在一个对象中，这样就可以在需要的时候撤销这个操作”这样的说法，实在太拗口了，而且在重写代码的时候，要记住这样一个冗长的目标实在是有点困难。如果说“我们正在向代码中引入 Command 模式”，这种表述既更加准确，也更加便于讨论。

如果你是一位服务器端的 Java 开发者，或者是一位任何类型的系统架构师，可能会感到疑惑，我们说的这些有何新意？如果你是来自于 Web 设计或者新媒体世界[2]，可能会认为我们是有着某种控制欲怪癖的家伙，因为我们宁可画图也不去编写一行真正的代码。无论你来自哪个阵营，都可能在疑惑，这些玩意儿与 Ajax 有何关系？我们的简短回答是：“大有关系”。我们下面就来探讨一下忙于工作的 Ajax 程序员可以从重构中得到哪些好处。

3.1.2 重构与 Ajax

我们前面已经提到，Ajax 应用很可能使用更多的 JavaScript 代码，而这些代码倾向于在浏览器中运行更长的时间。

在传统的 Web 应用中，复杂的代码运行在服务器上。在这些地方，设计模式常常应用在 PHP、Java 或.NET 代码中。对于 Ajax 来说，我们需要考察如何在客户端代码中应用相同的技术。

甚至就连说 JavaScript 比 Java 和 C# 那样具有严格结构的语言更加需要良好的组织，都是大有争议的。不考虑它与 C 语言很相似的语法，JavaScript 其实更接近于 Ruby、Python，甚至是 Common Lisp 这样的语言，而不是 Java 或 C#。JavaScript 语言极具灵活性，有极大的空间允许开发者开发个人风格或者习惯用法。对于技艺纯熟的开发者来说，这是很棒的特征；而对于水平一般的开发者来说，它所能提供的安全保障实在是太少了。Java 和 C# 这样的企业语言为由水平一般的开发者组成并且人员变动频繁的团队提供了良好的支持，然而 JavaScript 却没有提供这样的支持。

创建混乱的、艰深难懂的 JavaScript 代码的危险相对来说要高得多，随着代码从简单的 Web 页面上的小技巧发展到 Ajax 应用，这些危险的真实性会逐渐显现，说不定什么时候就会跳出来狠狠咬你一口。出于这个原因，我强烈提倡在 Ajax 开发中使用重构，其必要性比在 Java 或 C# 这类“安全的”编程语言中要大得多，因为在这类语言中设计模式早已经枝繁叶茂了。

3.1.3 保持均衡

继续前进之前，有必要再强调的一点是，重构和设计模式不过只是工具，只应该应用在确实有用的地方。如果过度使用，将会导致所谓的“分析瘫痪”（paralysis by analysis）状态，为了增加设计的灵活性和架构的适应性，以便应对可能永远也不会出现的未来的需求，导致应用的实现被这些不确定的因素所阻碍，迟迟无法得到可以运行的应用。

设计模式专家 Erich Gamma 在最近的一次访谈中（参见本章“资源”一节）对这种情况作了很好的总结。他谈到曾经有位读者请他帮忙，这位读者在自己的项目应用中只用到了《设计模式》一书中所描述的 23 个设计模式中的 21 个。这就好比一个开发者试图在他写的所有代码中避免使用整数、字符串和数组一样，设计模式只在一些特定的情形中才是有用的。

Gamma 推荐将重构作为引入设计模式的最佳方式。第一次写代码的时候，先以最简单的方式来完成工作；然后当你遇到一些常见的问题时，引入模式来解决这些问题。如果你已经编写了大量的代码，或者负责维护大量别人编写的混乱代码，你可能已经体验过一种远离人群，孤独地沉浸在代码之中的生活。幸运的是，即便到了这个时刻，仍然有可能引入设计模式来改善代码的质量。在下面一小节，我们将拿第 2 章中开发的“粗糙但可用”的代码开刀，看看重构可以为它做些什么。

3.1.4 重构实战

重构听起来似乎是个好主意，但是头脑更加实际的开发者希望在花钱购买之前，至少要看到它确实会起作用。我们现在花一些时间，在上一章中那个包含了核心 Ajax 功能的例子（代码清单 2-11）中应用一些重构技术。为了翻新那段代码的结构，我们定义了一个 `sendRequest()` 函数，触发一个发到服务器的请求。`sendRequest()` 委派 `initHttpRequest()` 函数查找适当的 XMLHttpRequest 对象，并且以硬编码的方式为它分配了一个 `onReadyState()` 回调函数来处理服务器的响应。XMLHttpRequest 对象定义为全局变量，使得回调函数可以找到它的引用。回调函数随后询问请求对象的状态，并且生成一些调试信息。

代码清单 2-11 的代码做了我们需要它做的事情，但是有些难以重用。通常，当我们发送请求到服务器时，也想要解析服务器的响应，并且根据结果做一些与应用密切相关的事情。为了在当前代码中插入定制的业务逻辑，我们需要修改 `onReadyState()` 函数这一段。

全局变量的存在也是一个容易出问题的地方。如果想要同时向服务器发起多个调用，我们必须能够为每个调用指定不同的回调函数。如果我们正在获取一个需要更新的资源列表，同时又要丢弃另外一个资源列表，无论如何我们不能将它们搞混！

在面向对象编程中，这类问题的标准解决方案是将必需的功能封装在对象中。要实现这一点，JavaScript 语言支持的 OO 编码风格已经绰绰有余。我们将这个对象叫做 ContentLoader，因为它用来从服务器加载内容。那么，这个对象看起来应该是个什么样子呢？理想情况下，我们应该能够创建它，给它传递一个表示请求的发送地址的 URL。我们也应该能够给它传递两个到自定义回调函数的引用，一个在文档成功加载时调用，另一个在发生了错误时调用。对于这个对象的调用也许看起来像是这样：

```
var loader=new net.ContentLoader('mydata.xml',parseMyData);
```

其中 `parseMyData` 是文档成功加载时调用的回调函数。代码清单 3-1 展示了 ContentLoader 对象的实现代码。这其中涉及了一些新的概念，稍后我们就来讨论。

代码清单 3-1 ContentLoader 对象

```
var net=new Object();  
net.READY_STATE_UNINITIALIZED=0;  
net.READY_STATE_LOADING=1;  
net.READY_STATE_LOADED=2;  
net.READY_STATE_INTERACTIVE=3;  
net.READY_STATE_COMPLETE=4;  
net.ContentLoader=function(url,.onload,onerror){  
    this.url=url;  
    this.req=null;  
    this.onload=onload;  
    this.onerror=(onerror)?onerror:this.defaultError;  
    this.loadXMLDoc(url);  
}  
net.ContentLoader.prototype={  
  
    loadXMLDoc:function(url){  
        if (window.XMLHttpRequest){  
            this.req=new XMLHttpRequest();  
        } else if (window.ActiveXObject){  
            this.req=new ActiveXObject("Microsoft.XMLHTTP");  
        }  
        if (this.req){  
            try{  
                var loader=this;  
                this.req.onreadystatechange=function(){  
                    loader.onReadyState.call(loader);  
                }  
                this.req.open('GET',url,true);  
                this.req.send(null);  
            }catch (err){  
                this.onerror.call(this);  
            }  
        }  
        onReadyState:function(){  
            var req=this.req;  
            var ready=req.readyState;  
            if (ready==net.READY_STATE_COMPLETE){  
                var httpStatus=req.status;  
                if (httpStatus==200||httpStatus==0){  
                    this.onload.call(this);  
                }else{  
                    this.onerror.call(this);  
                }  
            }  
            defaultError:function(){  
                alert("error fetching data:  
                "+'\n'+readyState+" "+this.req.readyState  
                +" "+status+" "+this.req.status  
                +" "+headers+" "+this.req.getAllResponseHeaders());  
            }  
        }  
    }  
};
```

这段代码中值得注意的第一件事是我们定义了一个全局变量 `net`①，然后将所有其他的引用都附加在它的上面。这样做可以将变量名发生冲突的风险降到最小，并且使得所有与网络请求相关的代码都位于一个地方。

我们为对象提供了一个构造函数②。它有三个参数，只有前两个是必需的。第三个参数是错误处理函数，我们检查它是否为null值[3]，如果需要就进行默认的错误处理。可以给函数传递数量可变的参数，这种能力对于OO程序员来说可能感觉会很奇怪。另外，可以将函数当作头等的引用来传递。这种能力也容易

使人迷惑，不过，这些都是JavaScript的基本特征。我们在附录B中会更详细地讨论这些JavaScript语言的特征。

我们将代码清单 2-11 中的 `initXMLHttpRequest()`❸和 `sendRequest()`❹函数的很大一部分都移到了对象内部。我们也重新命名了这些函数以反映出它们稍微扩大的作用域，它现在称作 `loadXMLDoc`❺。我们使用同样的技术来查找 XMLHttpRequest 对象，并且初始化一个请求，不过这个对象的用户无需关注这一点。`onReadyState` 回调函数❻很大程度上也与代码清单 2-11 中的代码类似。我们将对调试控制台的调用替换成对 `onload` 和 `onerror` 函数的调用。这里的语法看起来有点奇怪，我们来更仔细地检查一下。`onload` 和 `onerror` 是 `Function` 对象，而 `Function.call()` 是这个对象的方法。`Function.call()` 的第一个参数成为这个函数的上下文，也就是说，在函数的内部可以使用 `this` 关键字来引用它。

编写一个回调处理函数，将它传递给 ContentLoader 是非常简单的。如果我们需要引用任何 Content Loader 的属性，例如 `XMLHttpRequest` 或者 `url`，可以简单地使用 `this` 来做到，例如：

```
function myCallBack() {
  alert(
    this.url
    +" loaded! Here's the content:\n\n"
    +this.req.responseText
  );
}
```

构建这样一个必需的“基础架构”（plumbing）需要理解一些 JavaScript 的怪癖，但是一旦这个对象写好了，最终用户不必再担心它。

这种状况通常是一次优秀重构的标记。我们将困难的代码隐藏在对象内部，对外只展示出容易使用的部分。最终用户减少了很多不必要的困难，与此同时，专家又可以在一个地方专心负责维护那些困难的代码。修改只需要做一次，就可以影响到整个代码库。

我们已经讨论了重构的基础知识，并且展示了在实际工作中它如何给我们带来好处。在下一节中，我们将考察 Ajax 编程中一些更加常见的问题，并且看看如何使用重构来解决这些问题。在这个过程中，我们将会发现一些有用的技巧，这些技巧在后续的章节中重用或者应用到你自己的项目中。

3.2 一些小型重构的案例研究

下面的几个小节致力于探讨 Ajax 开发中的一些问题，并且考察一些通用的解决方案。在每一个案例中，我们都会为你展示重构如何减轻了与这个问题相关的痛苦，然后我们会识别出可以在其他地方重用的这个方案中的元素。

为了保持《设计模式》这部著作的光荣传统，我们继续沿用它的经典表述方式，我们按照问题、技术解决方案，然后是相关的更大问题这样的顺序来讨论。

3.2.1 跨浏览器不一致性：Facade 和 Adapter 模式

如果你问任何一位 Web 开发者，他在工作中抱怨最多的是什么？无论他是程序员、设计师、图形艺术家或者其他的相关人员，如何能让他的作品在不同的浏览器上都能正确地显示，这个问题通常都会名列前茅。Web 世界技术标准林立，大部分浏览器厂商的实现都或多或少的与这些标准存在差异。有的时候，标准本身就很含糊，容易引起不同的解释；有的时候，浏览器厂商出于易用的目的，各自通过不一致的方式来扩展这些标准；还有的时候，仅仅是因为浏览器中残留着过时的 bug。

JavaScript 编程人员从很早以来就通过检查代码运行在哪个浏览器中，或者通过测试某个特定的对象是否存在来解决这个问题。我们来看一个简单的例子。

1. 使用 DOM 元素

我们在第 2 章中已经讨论过，Web 页面是通过 DOM 公开给 JavaScript 的，以一个树状结构表示，其元素对应 HTML 文档中的标签。当通过程序来维护一棵 DOM 树的时候，想要找到某个元素在页面上的位置是一种非常常见的需求。不幸的是，这些年来浏览器厂商提供了好几种非标准的方法，使得我们很难写出安全可靠的跨浏览器代码来完成这个任务。代码清单 3-2 是来自 Mike Foster 的 x 库（见 3.5 节）的一个函数简化版本。这个函数接受一个参数 e，以一种全面的方式发现这个 DOM 元素的左边界的位置。

代码清单 3-2 `getLeft()` 函数

```
function getLeft(e) {
    if(!!(e=xGetElementById(e))) {
        return 0;
    }
    var css=xDef(e.style);
    if (css && xStr(e.style.left)) {
        ix=parseInt(e.style.left);
        if(isNaN(ix)) ix=0;
    }else if(css && xDef(e.style.pixelLeft)) {
        ix=e.style.pixelLeft;
    }
    return ix;
}
```

通过在第 2 章中见到过的 style 数组，不同的浏览器提供了很多方法来确定节点的位置。W3C 的 CSS2 标准支持一个叫做 `style.left` 的属性，其定义为一个字符串，描述了值和单位，例如 `100px`。除了像素，它也支持其他的单位。而作为对照，`style.pixelLeft` 的值则是一个数字，它假设所有的值都是以像素为单位的。`pixelLeft` 只有微软的 IE 才支持。这里讨论的 `getLeft()` 方法首先检查浏览器是否支持 CSS，然后测试这两个值，首先尝试使用 W3C 的标准属性，如果没有找到这两个值，就会返回一个 0 作为缺省值。注意，我们没有明确地检查浏览器的名称或者版本，而是使用了在第 2 章中讨论过的更加健壮的对象检测技术。

编写这样功能的一些代码以适应不同的浏览器，确实是一件很乏味的事情。不过，一旦完成了，开发者就再也不必为这样的事情而烦恼，而可以专心来开发真正的应用。像经过良好测试的函数库（如 x 库），已经为我们完成了大部分困难的工作。通过这样一个可靠的适配器函数来发现 DOM 元素在页面上的位置，无疑将会大大加速 Ajax 用户界面的开发。

2. 发送请求到服务器

在第 2 章中，我们还提到过另一个类似的跨浏览器不兼容性问题。浏览器厂商提供了非标准机制来获得 XMLHttpRequest 对象，用于发送异步请求到服务器。当想要从服务器加载一个 XML 文档的时候，我们需要确定可以使用哪一个方法。

IE 只能通过访问 ActiveX 组件获得这个对象，而 Mozilla 和 Safari 则是以内建的原生对象的形式提供这个对象[4]，只有加载 XML 的代码本身需要知道这个差别。一旦 XMLHttpRequest 对象被返回给其余的代码，这个对象的行为在两种情况下都是相同的。调用它的代码既无须理解 ActiveX，也无须理解原生的对像子系统；它只需要理解 `net.ContentLoader()` 的构造函数就足够了。

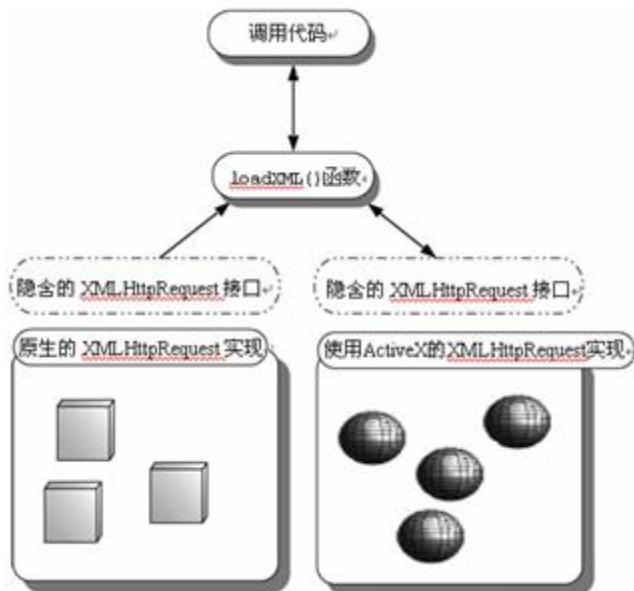
3. Façade 模式

无论是在 `getLeft()` 还是在 `new net.ContentLoader()` 中，完成对象检测的代码都显得丑陋而乏味。我们可以通过定义一个函数来隐藏这些代码，这使得其他代码更加容易阅读，并且将对象检测代码集中到了一个地方。这就是重构的一个基本原则——不要重复你自己，常常简写为 DRY (don't repeat yourself)。如果发现在某种边界情况下对象检测代码不能正常工作，那么只需要在一个地方修改，就可以将影响扩散到所有调用它的代码中，无论这些代码是用来发现 DOM 元素的左坐标、创建 XML 请求对象，或者是在做其他我们试图做的事情。

用设计模式的行话来说，我们正在使用一种称作 Façade 的模式。Façade 模式可以用来为一个服务或者一些功能的不同实现方式提供公共的访问点。例如，`XMLHttpRequest` 对象提供了有用的服务，只要它还能工作，应用就不会真正去关心它究竟是以什么方式实现的（图 3-1）。

图 3-1 Façade 模式的示意图，涉及跨浏览器的 `XMLHttpRequest` 对象。`loadXML()` 函数需要一个 `XMLHttpRequest` 对象，但是它并不关心实际的实现。底层实现可能需要提供相当复杂的 HTTP 请求的语义，但是两种实现在这里都做了简化，以提供了调用函数所需要的基本功能

在很多情况下，我们也想要简化对子系统的访问。例如，在获取 DOM 元素左坐标的情况下，CSS 规约提供了过多的选择，允许使用像素、点、em（屏幕字体尺寸）以及其他单位来指定



该值。代码清单 3-2 中的 `getLeft()` 函数以及布局系统全部都使用像素作为单位。以这种方式来简化这个子系统是 Façade 模式的另一种功能。

4. Adapter 模式

与 Façade 模式密切相关的是 Adapter 模式。在 Adapter 模式中，就像在微软和 Mozilla 的浏览器中得到一个 XMLHttpRequest 对象一样，我们是与两个提供相同功能的子系统共同工作。与前面为每个子系统构造一个新的 Façade 不同，我们为其中的一个子系统提供了一个额外的层，使得这个子系统展现出与另一个子系统相同的 API。这个层称作适配器层（adapter）。将在 3.5.1 节讨论的用于 Ajax 开发的 Sarissa XML 库使用了 Adapter 模式，使 IE 的 ActiveX 控件看起来像是与 Mozilla 内建的 XMLHttpRequest 对象一样。这两种方法都是有效的，都可以帮助我们将遗留的或第三方的代码（包括浏览器本身）集成到我们的 Ajax 项目中。

我们来继续研究下一个案例，考虑 JavaScript 的事件处理模型中的问题。

3.2.2 管理事件处理函数：Observer 模式

我们无法不使用基于事件的编程技术而编写大量的 Ajax 代码。JavaScript 的用户界面严重依赖于事件驱动，而在 Ajax 引入异步请求之后，应用程序需要处理的回调函数和事件更多了。在相对简单的应用中，我们可以使用单个函数来处理类似鼠标点击或者服务器端数据到达的事件。然而，随着应用的规模和复杂性日渐增长，我们可能想要通知几个不同的子系统，甚至需要提供一种机制，即对事件感兴趣的各方可以自行登记所需要的通知。我们通过一个例子来看看这里面有些什么问题。

1. 使用多个事件处理函数

当使用 JavaScript 对 DOM 节点进行脚本处理的时候，通常都需要定义一个 `window.onload` 函数，这个函数会在页面（以及其所对应的 DOM 树）全部加载完成的时候执行。一旦页面加载完成，页面上有一个 DOM 元素，用来显示从服务器获取的动态生成的数据。完成数据获取和显示的 JavaScript 代码需要一个到 DOM 节点的引用，于是它通过定义一个 `window.onload` 事件处理函数来得到这个引用。

```
window.onload=function(){  
    displayDiv=document.getElementById('display');  
}  
}
```

看起来还不错。假设我们现在想要增加另一个视觉效果，例如提供一个新闻提要警告框（如果你对实现这个功能感兴趣，参见第 13 章）。控制新闻提要显示的代码也需要在一开始就获取到某个 DOM 元素的引用，于是它也定义了一个 `window.onload` 事件处理函数：

```
window.onload=function(){  
    feedDiv=document.getElementById('feeds');  
}
```

我们分别在独立的页面中测试这两段代码，发现它们都可以正常工作。但是一旦将这两段代码放在一起，第二个 `window.onload` 函数就会覆盖第一个，导致来自服务器的数据无法显示，而是产生了 JavaScript 错误。问题就在于在 `window` 对象上只允许附加一个 `onload` 函数。

2. 组合事件处理函数的局限

第二个事件处理函数覆盖了第一个，通过将两者组合在单个的函数中可以解决这个问题：

```
window.onload=function(){  
    displayDiv=document.getElementById('display');  
    feedDiv=document.getElementById('feeds');  
}
```

对于我们当前的例子来说，这个方法是有效的，但是它导致本来毫不相干的数据显示与新闻提要阅读器的代码混杂在了一起。如果我们要处理的不是 2 个系统，而是 10 个或 20 个系统，它们都需要得到对几个 DOM 元素的引用，那么像这样的组合事件处理函数将会变得难以维护。从中插入和取出单独的组件将会变得非常困难而且极易出错，出现本章开头所描述的情况，没有任何人愿意去动这段一碰就坏的代码。我们可以通过为每个子系统定义一个加载函数来重构一下：

```
window.onload=function(){  
    getDisplayElements();  
    getFeedElements();  
}  
  
function getDisplayElements(){  
    displayDiv=document.getElementById('display');  
}
```

```
function getFeedElements(){  
    feedDiv=document.getElementById('feeds');  
}
```

这样写显得清晰一些，将组合 `window.onload()` 函数中每个子系统的代码缩减到一行，但这个组合函数仍然是设计中的一个薄弱环节，还是有可能带来麻烦。下面，我们考察这个问题的略微复杂、但扩展性更好的解决方案。

3. Observer 模式

一个操作应该是谁的职责？询问这个问题有些时候是很有帮助的。在这个组合函数的解决方案中，`window` 对象负责获得到 DOM 元素的引用，随后 `window` 对象必须知道当前页面中包括哪些子系统。理想情况下，每个子系统应该自己负责获取它们需要的引用。按照这种方式，如果它包括在页面中，就会获得自己需要的引用；如果没有包括在页面中，就不必做这件事情。

为了清晰地分离这个职责，可以允许这些系统通过传递一个函数来登记，从而在发生 `onload` 事件时得到通知，这些函数会在 `window.onload` 事件触发时调用。这里是一个简单的实现：

```
window.onloadListeners=new Array();  
  
window.addOnLoadListener(listener){  
    window.onloadListeners[window.onloadListeners.length]=listener;  
}  
  
}
```

窗口完全加载后，`window` 对象只需要遍历这个数组，并且依次调用每个方法：

```
window.onload=function(){  
    for(var i=0;i<window.onloadListeners.length;i++){  
        var func=window.onloadListeners[i];  
        func.call();  
    }  
}
```

如果每一个子系统都使用这种方法，我们就可以提供更清晰的方式来设置所有的子系统，而不必将它们混杂在一起。当然，只需要少量的恶意代码[\[5\]](#)就可以直接覆盖window.onload，使我们的努力功亏一篑。但是，我们必须负起照看代码库以避免此类问题发生的责任。

还有一点值得指出的是，新的W3C事件模型也实现了一个多事件处理函数的系统。我们之所以选择在老的JavaScript事件模型之上建造系统，是因为W3C的模型在不同浏览器中的实现不一致。第4章将会更深入的讨论这个问题。

在这里，将重构的设计模式叫做Observer模式。Observer模式定义了一个Observable对象，在我们这个例子中，它是内建的window对象，一组Observer或Listener可以将自己登记在这个对象上（图3-2）。

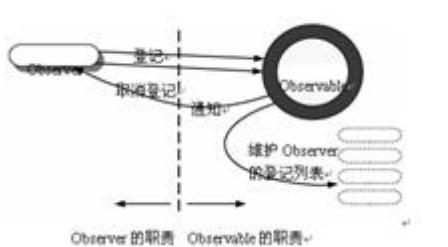


图3-2 Observer模式中的职责分离。希望得到事件通知的对象（即**Observer**）可以在消息源（**Observable**）上登记，也可以取消登记。当事件发生时消息源会通知所有已登记的对象

通过Observer模式，职责被适当地分配到了事件源和事件处理函数之间。处理函数负责它们自己的登记或取消登记；事件源则负责维护已登记各方的列表，并且在事件发生时通知它们。这个模式在事件驱动的用户界面编程领域拥有悠久的使用历史，我们在第4章深入讨论JavaScript事件的时候还会回到Observer模式上。正如我们将会看到的，Observer模式也可以独立于浏览器的鼠标和键盘事件的处理，使用在我们自己编写的对象之上。

现在，我们进入到下一个经常重复出现的问题，通过重构来解决它。

3.2.3 重用用户操作处理函数：Command模式

在大多数应用中，都是由用户来告诉（通过点击鼠标和按下键盘）应用程序要做些什么事情，然后应用程序照着做，这可能是一件显而易见的事情。在一个简单程序中，可能只会给用户提供一种方法来完成一个操作，但在更加复杂的界面中，我们常常想让用户通过几种途径来触发相同的操作。

1. 实现按钮 UI 组件

假设有一个 DOM 元素，通过设置样式使它看起来像是一个按钮 UI 组件。当按下它时会执行一个计算，然后使用计算的结果更新一个 HTML 表格。我们可以为这个 button 元素定义一个鼠标点击的事件处理函数，就像这样：

```
function buttonOnclkHandler(event){  
    var data=new Array();  
    data[0]=6;  
    data[1]=data[0]/3;  
    data[2]=data[0]*data[1]+7;  
    var newRow=createTableRow(dataTable);  
    for (var i=0;i<data.length;i++){  
        createTableCell(newRow,data[i]);  
    }  
}
```

在这里假定 dataTable 变量是一个到现有表格的引用，而 createTableRow() 和 createTableCell() 函数管理 DOM 处理的细节。这里真正有意思的是计算阶段，在实际应用中，这个阶段可能会包括上百行代码。我们将这个事件处理函数分配给 button 元素，像这样：

```
buttonDiv.onclick=buttonOnclkHandler;
```

2. 支持多种事件类型

假设我们现在为 Ajax 应用做压力测试。我们轮询服务器以获取更新的数据，如果某个特定的值被服务器更新了，需要重新执行计算，然后使用得到的数据更新另外一个表格。在这里，没有必要深入讨论重复轮询服务器的细节。假定有一个到 poller 对象的引用，在其内部使用了 XMLHttpRequest 对象，并且将

它的 `onreadystatechange` 处理函数设置为调用 `onload` 函数，当来自服务器的更新数据加载完成后会调用这个函数。我们可以将计算和显示的阶段抽象到帮助函数中，就像这样：

```
function buttonOnclickHandler(event){  
    var data=calculate();  
    showData(dataTable,data);  
}  
  
function ajaxOnloadHandler(){  
    var data=calculate();  
    showData(otherDataTable,data);  
}  
  
function calculate(){  
    var data=new Array();  
    data[0]=6;  
    data[1]=data[0]/3;  
    data[2]=data[0]*data[1]+7;  
    return data;  
}  
  
function showData(table,data){  
    var newRow=createTableRow(table);  
    for (var i=0;i<data.length;i++){  
        createTableCell(newRow,data[i]);  
    }  
}  
  
buttonDiv.onclick=buttonOnclickHandler;  
poller.onload=ajaxOnloadHandler;
```

我们看到，大量常用的功能已经抽象到了 `calculate()` 和 `showData()` 函数中，在 `onclick` 和 `onload` 处理函数中只有少量重复的代码。

现在已经将业务逻辑和用户界面更新很好地分离开来了。我们又发现了一个有用的可重复解决方案，称作 Command 模式。Command 对象定义了一些具有任意复杂性的活动，可以很容易地在代码之间传递，或者在 UI 元素之间交换。在面向对象语言的传统 Command 模式中，用户的交互都封装为 Command 对象，通常继承自一个基类或者实现一个接口。在这里我们使用一种略微不同的方法来解决相同的问题。因为在 JavaScript 中，函数本身就是头等对象，我们可以直接将它们当作 Command 对象来处理，与此同时仍然提供了相同的抽象级别。

将用户所做的事情都封装为 Command 对象可能看起来有点麻烦，但是这样做是有回报的。当所有的用户行为都封装在 Command 对象中时，我们就可以很容易地联合使用其他标准的功能。讨论最多的扩展是增加 `undo()` 方法，一旦完成了这个工作，就为在整个应用中提供通用的撤销（undo）功能奠定了良好的基础。在一个更加复杂的例子中，Command 在执行时可以被记录在一个栈中，用户可以通过撤销按钮来回退这个栈，从而将应用返回到以前的状态（图 3-3）。

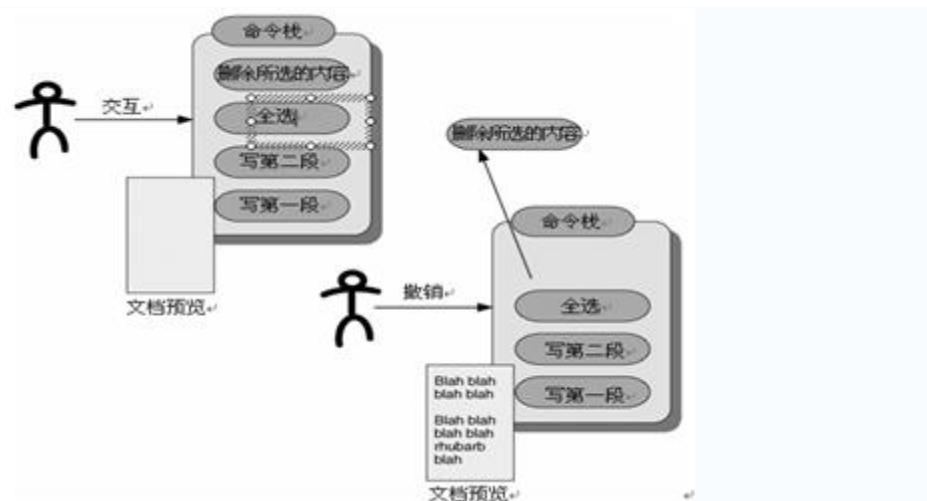


图 3-3 在字处理应用中，使用 Command 模式来实现通用的撤销栈。所有的用户交互都用 Command 对象来表示，可以同时支持执行和撤销操作

每个新的Command对象都放在栈的顶端，可以按顺序逐个回退。用户通过一系列写的操作创建了一个文档，然后选择整个文档时，一不小心点了删除按钮。当调用`undo`功能时，从栈中弹出最顶端的条目[6]，然后调用它的`undo()`方法，恢复被删除的文本。后续的撤销操作可能是取消文本的选择，等等。

当然，使用 Command 模式来创建一个撤销栈，还要确保这些执行和撤销操作的组合能够返回系统的初始状态，对于开发者来说这意味着一些额外的工作。提供完善的撤销功能可以使得产品显得与众不同，特

别是对于频繁或长时间使用的应用来说，这个功能更加重要。正如我们在第 1 章中讨论过的，这正是 Ajax 正在努力扩张的版图。

当我们需要在应用中跨越子系统的边界传递信息的时候，Command 对象也能派上用场。而网络正是这样的一个边界，在第 5 章中，讨论客户/服务器之间的交互时，还会再次谈到 Command 模式。

3.2.4 保持对资源的唯一引用：Singleton 模式

在一些场合，确保只能从一个地点访问某个特殊的资源是很重要的。这一点最好还是通过一个特殊的例子来解释，我们来看一下。

1. 简单的交易例子

假设 Ajax 应用可以操作股票市场的数据，允许我们在真实的市场上进行交易，执行 what-if 计算，并且通过网络与其他用户进行模拟交易。我们按照交通信号灯，为应用定义了 3 种模式。在实时模式（绿色模式）下，当股市开盘的时候，我们可以在真实的市场上买卖股票，并且使用保存的真实数据来执行 what-if 计算。当股市封盘的时候，我们恢复到分析模式（红色模式），仍然可以执行 what-if 计算，但是不能买卖。在模拟模式（黄色模式）下，我们可以执行所有绿色模式可以执行的操作，但是并不是与真实的股票市场交互，而是使用虚拟的数据。

客户端代码将这种变换表示为一个 JavaScript 对象，定义如下：

```
var MODE_RED=1;  
var MODE_AMBER=2;  
var MODE_GREEN=2;  
  
function TradingMode(){  
    this.mode=MODE_RED;  
}  
 
```

我们可以在代码中的很多地方查询和设置这个对象所表示的模式，还可以提供 `getMode()` 和 `setMode()` 函数，在其中检查一些条件是否满足，例如真实的市场是否已经开盘，但是在目前我们最好还是简单一些。

假设我们为用户提供了买和卖两个选项，在真正执行交易之前可以先计算出可能的盈利和损失。买和卖的操作依赖于操作的模式指向不同的 Web 服务：黄色模式指向内部的服务；绿色模式指向经纪人服务器上的服务；红色模式则不提供任何服务。类似地，分析基于所获取的当前和最近的价格数据：黄色模式使用模拟数据，绿色模式使用真实市场的数据。要知道该指向哪个数据来源，两者都需要查询一个如下定义的 TradingMode 对象（图 3-4）：

两个活动都指向相同的 TradingMode 对象是很有必要的。如果用户根据对真实市场数据所做的分析在模拟市场上进行买卖，那么她可能会输掉这次游戏；如果她根据对模拟市场数据所做的分析在真实市场上进行买卖，那么她可能很快就会丢掉饭碗！

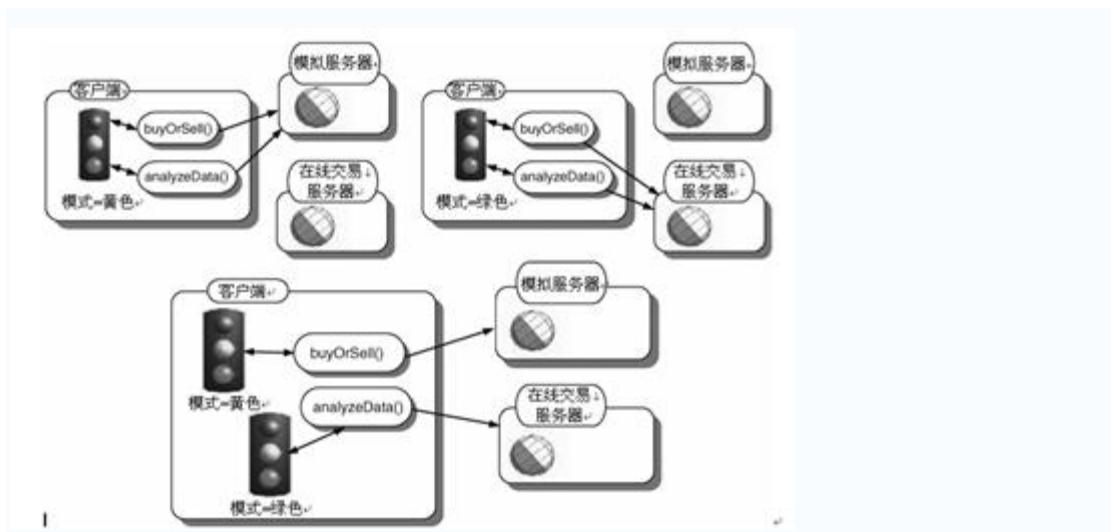
一个对象只有一个实例，有时也描述为一个单例（singleton）。我们先来考察一下在面向对象语言中是如何处理单例的，然后找到在 JavaScript 中使用它的策略。

2. Java 中的单例

在类似 Java 的语言中，实现单例的方法通常是隐藏对象的构造函数，并且提供一个 getter 方法，如代码清单 3-3 所示。

图 3-4 在 Ajax 交易应用的例子中，买/卖功能和分析功能都基于 TradingMode 对象的状态来确定是使用真实数据还是使用模拟数据。在黄色模式下访问模拟服务器，而在绿色模式下访问在线交易服务器。如果系统中有多于一个的 TradingMode 对象，系统将会出现状态不一致的情况

代码清单 3-3 Java 中 TradingMode 对象的单例实现



```
public class TradingMode{  
    private static TradingMode instance=null;  
    public int mode;  
    private TradingMode(){  
        mode=MODE_RED;  
    }  
    public static TradingMode getInstance(){  
        if (instance==null){  
            instance=new TradingMode();  
        }  
        return instance;  
    }  
    public void setMode(int mode){  
        ...  
    }  
}
```

基于 Java 的解决方案利用 private 和 public 的访问修饰符来强化单例的行为，下面的代码是无法编译的：

```
new TradingMode().setMode(MODE_AMBER);
```

因为构造函数不是公共的，而下面的代码则可以编译：

```
TradingMode.getInstance().setMode(MODE_AMBER);
```

这行代码保证了每次调用都得到相同的 TradingMode 对象。我们在这里使用了几种 JavaScript 所没有的语言特征，我们来看看如何解决这个问题。

3. JavaScript 中的单例

在 JavaScript 中，虽然没有内建的对于访问修饰符的支持，但是可以通过不提供构造函数的方式来“隐藏”构造函数。JavaScript 是基于原型的，构造函数是普通的 Function 对象（如果不理解这是什么意思，参见附录 B）。我们可以按照平常的方式来定义 TradingMode 对象：

```
function TradingMode(){  
    this.mode=MODE_RED;  
}  
  
TradingMode.prototype.setMode=function(){  
}
```

然后提供一个全局变量作为一个伪单例：

```
TradingMode.instance=new TradingMode();
```

但是这无法阻止恶意代码调用构造函数。另一方面，我们可以不使用原型，手工创建整个对象：

```
var TradingMode = new Object();

TradingMode.mode = MODE_RED;

TradingMode.setMode = function() {

    ...

}
```

也可以用更加简洁的方式来定义它：

```
var TradingMode = {

    mode:MODE_RED,

    setMode:function(){

        ...

    }

};
```

这两个例子都会生成相同的对象。前一种方式对于 Java 或 C# 程序员或许更加熟悉。我们展示后一种方式，是因为在 Prototype 库和这个库所衍生的框架中经常使用这种方式。

这种解决方案只在单独的脚本上下文范围内是有效的。如果脚本加载到另外一个 IFrame 中，它将使用自己的单例副本。我们可以明确指定通过最顶层的文档来访问单例对象（在 JavaScript 中，`top` 总是指向这个文档），如代码清单 3-4 所示。

代码清单 3-4 JavaScript 中 TradingMode 对象的单例实现

```
function getTradingMode(){
    if (!top.TradingMode){
        top.TradingMode=new Object();
        top.TradingMode.mode=MODE_RED;
        top.TradingMode.setMode=function(){
            ...
        }
    }
    return top.TradingMode;
}
```

这使得脚本可以安全地包括在多个 IFrame 中，同时保持单例对象的唯一性。（如果你计划支持跨多个顶层窗口的单例，你需要研究一下 `top.opener`。限于篇幅，我们将这个留给读者作为练习。）

在编写 UI 代码的时候，对单例的需要可能不是很强烈，但是在使用 JavaScript 编写业务逻辑代码的时候，它会变得非常有用。在传统的 Web 应用中，业务逻辑通常都位于服务器上，但是以 Ajax 的方式进行开发改变了这种状况，单例会变得很有用，因此有必要熟练掌握。

在实用的层面上，重构可以为我们做些什么？上面的例子为我们提供了一点初步的印象。到目前为止，我们所看到的例子都是相当简单的，即便如此，通过重构使代码更加清晰仍然帮助我们排除了代码中的一些薄弱环节，否则随着应用规模逐渐增大，这些问题可能会神出鬼没，使我们寝食难安。

我们讲述了几种设计模式，在下一节中将来考察一个大规模的服务器端模式，看看如何重构一些最初纠缠在一起的代码，使得它们变得更加清晰、更加灵活。

3.3 模型-视图-控制器

到目前为止，我们所考察的一些小模式在特定编码任务中是很有用的。还有一些用来组织整个应用的模式，有时候也称作架构模式（architectural pattern）。在本节中，我们将要考察一个能在很多方面帮助我们组织 Ajax 项目的架构模式，它可以使得代码更容易编写，也更容易维护。

模型—视图—控制器（MVC）描述的是将程序与用户交互的部分和完成其他繁重工作、科学计算或业务逻辑等等的部分很好分离的一种方式。

MVC 通常用于大规模的应用，覆盖应用的所有层次或者跨越多个层次。在本章中，我们引入了这个模式，并且展示如何在 Web 服务器上应用这一模式为 Ajax 应用提供数据。在第 4 章中，我们会更侧重于在客户端的 JavaScript 中应用这一模式。

MVC 模式识别出了系统中一个组件所实现的三种角色。模型表示应用的问题域，也就是要解决的问题。一个文字处理应用需要为文档建模；一个地图应用需要为点、栅格、等高线等建模。

视图是程序展现给用户的东西，例如输入表单、图片、文本或者 UI 组件。视图不一定是图形形式的。例如，在一个声音驱动的程序中，语音提示也是视图。

MVC 的黄金定律是视图和模型不应该相互通信。表面上看，这个要求似乎会导致程序的功能不全，但是这正是控制器的作用之所在。当用户按下一个按钮或者填写一个表单时，视图会通知控制器。控制器操作模型并且决定模型上的变化是否需要变化视图。如果需要，它就通知视图更新它自己（见图 3-5）。

图 3-5 模型—视图—控制器模式的主要组成部分。视图和模型不直接交互，始终通过控制器来进行。控制器可以看作是一个薄的边界层，允许模型和视图通过它来通信，它使得在代码库中实现清晰的职责分离，提高了代码的灵活性和可维护性

这样做的好处就在于，模型和视图可以保持松散耦合，也就是说，它们都不需要深入了解对方的内部实现。很显然，它们还是需要知道足够的事情才能完成工作，但是视图只需要对模型有一些大体上的了解就足够了。



我们来考虑一个库存管理的程序。控制器可以给视图提供一个函数，这个函数根据给定的类别 ID 返回该类别所有产品的列表，但是视图并不知道这个列表是从哪里来的。在这个程序的第 1 版中，用来生成列表数组的数据可能是保存在内存中或者保存在纯文本文件中。而在这个程序的第 2 版中，由于需要处理数量大得多的数据，在架构中加入了一台关系数据库服务器。模型上的这种变化意义重大，需要重写大量的代码。但是，只要控制器仍然能够提供与某个类别匹配的产品列表，对于视图的代码就不会造成任何影响。

类似地，开发视图代码的工程师也可以自由地修改代码，改善应用的可用性。只要遵循控制器所提供的接口上的基本约定，就无需担心会破坏模型中的隐含假设。通过将系统划分为子系统，MVC 提供了一种可靠的策略，使得小的代码变更[7]所引起的波动不至于扩散到整个代码库，并且使得负责每个子系统的团队可以在不影响其他团队的情况下，对需求变化做出迅速地响应。

MVC 模式通常以一种特殊的方式应用在传统的 Web 应用开发框架上[8]，用来提供组成用户界面的一系列连续的静态页面。当运行 Ajax 应用时，它需要从服务器获取数据，服务器端提供数据的机制与传统 Web 应用中相应的机制是类似的。Web 服务器风格的 MVC 对于 Ajax 而言仍是有益的。鉴于它得到了广泛的理解，我们就从这里开始，然后再去考察更加特定于 Ajax 的 MVC 使用方式。

如果你是一个 Web 开发框架的新手，下一节可以为你提供必需的信息，以便理解它们如何使 Ajax 应用更具扩展性或者更加健壮。另一方面，如果你对于 Web 层的工具[例如模板引擎、对象

关系映射（ORM）工具]或者开发框架（例如 Struts、Spring 或 Tapestry）已经非常熟悉，那么这里讨论的大部分内容对你已经不陌生了，可以跳过本节，直接阅读第 4 章，在那里我们会讨论一种非常不同的 MVC 使用方式。

3.4 Web 服务器端的 MVC

MVC 对于 Web 应用并不陌生，甚至是在本书中大力抨击的传统基于页面的 Web 应用中，它也是我们的老熟人了。分离视图和模型对于 Web 应用来说是很自然的，因为它们本来就分属不同的机器[9]。那么，Web 应用是否天生就符合 MVC 模式呢？换句话说，难道还有可能编写一个将视图和模型混杂在一起的 Web 应用吗？

很不幸，完全有可能出现这种情况，因为这太容易做到了，可能大多数 Web 开发者以前都这样做过，当然我们也未能幸免。

大多数 Web MVC 的提倡者都将生成的 HTML 页面以及生成页面的代码看作是视图，而不是将用户实际看到的页面所呈现的东西看作是视图。在一个为 JavaScript 客户端提供数据的 Ajax 应用中[10]，按照上述观点来看视图是通过 HTTP 响应返回给客户端的 XML 文档[11]。将生成文档的代码与执行业务逻辑的代码分离开，确实需要一些纪律才行。

3.4.1 不使用模式的 Ajax Web 服务器层

为了展示所讨论的内容，我们来为 Ajax 应用开发一个示例的 Web 服务器层。我们已经在第 2 章和 3.1.4 节看到了一些基础的客户端 Ajax 代码，第 4 章还会回到客户端的代码。现在，我们将注意力集中在 Web 服务器端发生的事情。我们开始以尽可能简单的方式来编码，随后逐渐重构到 MVC 模式，看看它能为应用提高应对变化的能力带来哪些好处。首先，我们来介绍一下这个应用。

我们有一个服装店的服装列表，这个列表存储在数据库中。我们想要查询这个数据库，然后将条目的列表展示给用户，每个条目[12]包括图片、标题、简短的描述以及价格。如果某种服装有几种颜色或尺寸，还允许用户来选择。图 3-6 展示了这个系统的主要组成部分，即数据库、表示产品的数据结构以及传输到 Ajax 客户端的 XML 文档，在这个文档中列出了匹配查询的所有产品。

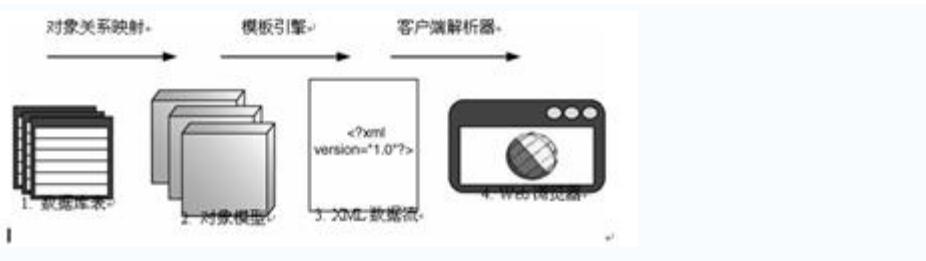


图 3-6 在在线商店的例子中，用来生成 XML 格式的产品数据的主要组成部分。在生成视图的过程中，从数据库中提取出一个结果集，使用它来组装表示每种服装的数据结构，然后以 XML 数据流的形式传输给客户端

假设用户刚刚进入商店，我们为他提供了男装、女装和童装几个选项。每个产品都属于这三种类别中的一种，通过数据库 Garments 表中的 Category 字段来标识。用来获得男装类的所有相关条目的简单 SQL 语句可能是这样：

```
SELECT * FROM garments WHERE CATEGORY = 'Menswear';
```

我们需要获取这个查询的结果，然后将它以 XML 的格式发送给 Ajax 应用。我们看看如何来做。

1. 为客户端生成 XML 数据

代码清单 3-5 展示了实现这个特定需求的迅速而粗糙[13]的解决方案。这个例子使用 PHP 和 MySQL 数据库，但是我们关注的重点是大体上的结构。如果换成 ASP、JSP 或者 Ruby 脚本，可能会得到结构类似的代码。

代码清单 3-5 迅速而随意地从数据库查询结果生成 XML 数据流

```
<?php
header("Content-type: application/xml");    // 告诉客户端返回是 XML
echo "<?xml version='1.0' encoding='UTF-8' ?>\n";
$db=mysql_connect("my_db_server", "mysql_user");
mysql_select_db("mydb", $db);
$sql="SELECT id,title,description,price,colors,sizes
    .FROM garments WHERE category='".$cat."'";
$result=mysql_query($sql,$db);
echo "<garments>\n";
while ($myrow = mysql_fetch_row($result)) {    // 遍历结果集
    printf("<garment id=\"%s\" title=\"%s\">\n",
        $myrow["id"],
        $myrow["title"],
        $myrow["description"],
        $myrow["price"]);
    if (!is_null($myrow["colors"])){
        ."<description>%s</description>\n<price>%s</price>\n",
        $myrow["colors"],
        $myrow["price"]);
    }
}
mysql_free_result($result);
mysql_close($db);
```

```
    echo "<colors>{$myrow['colors']}
```

代码清单 3-5 中的 PHP 页面可以生成类似于代码清单 3-6 的 XML 页面，在这个例子中，数据库里有两个匹配的产品。这里，代码进行了缩排以便于阅读。之所以选择 XML 作为客户端和服务器之间通信的媒介，是因为它通常都用于这个目的，并且第 2 章也已经提到如何使用 XML-`HttpRequest` 对象来处理服务器端生成的 XML 文档。第 5 章将会更加详细地探讨客户端和服务器通信的其他选项。

代码清单 3-6 代码清单 3-5 输出的简单的 XML

```
<garments>
  <garment id="SCK001" title="Golfers' Socks">
    <description>Garish diamond patterned socks. Real wool.
      Real itchy.</description>
    <price>$5.99</price>
    <colors>heather combo,hawaiian medley,wild turkey</colors>
  </garment>
  <garment id="HAT056" title="Deerstalker Cap">
    <description>Complete with big flappy bits.
      As worn by the great detective Sherlock Holmes.
      Pipe is model's own.</description>
    <price>$79.99</price>
    <sizes>S, M, L, XL, egghead</sizes>
  </garment>
</garments>
```

我们有了一个 Web 服务器端的应用，假设在前端有一个很好的 Ajax 应用来处理这个 XML。来展望一下未来。假设随着产品范围的扩大，要添加子类别（例如时装、休闲装、户外运动装），还要添加“按照季节搜索”的功能、实现关键字搜索、以及清除条目的链接。所有这些特征都可以通过类似的 XML 数据流来很好地支持。我们来考察一下如何重用当前的代码以便实现这些目标，以及在这个过程中将会遇到什么阻碍。

2. 可重用性问题

要重用当前的脚本有几个阻碍。首先，将 SQL 查询硬编码在了页面中。如果要按照类别或者关键字来搜索，就必须要修改 SQL 语句。随着不断加入更多的查询选项，最终的代码中会堆积大量 `if` 语句而变得丑陋不堪。

还有一个更糟糕的替代方案，那就是简单地接受 CGI 参数作为 WHERE 子句，即：

```
$sql="SELECT id,title,description,price,colors,sizes"  
."FROM garments  
  
WHERE ".$sqlWhere;
```

随后直接通过 URL 来调用，例如：

```
garments.php?sqlWhere=CATEGORY="Menswear"
```

这种方案进一步混淆了模型和视图，将原始的 SQL 暴露给表现层的代码。这为恶意的 SQL 注入攻击敞开了大门。虽然现代版本的 PHP 对于这类攻击有一些内建的防范措施，但是依赖它们是很愚蠢的。

其次，将 XML 的数据格式硬编码在了页面中，它被埋在了一大堆的 `printf` 和 `echo` 语句之中。我们有很多理由需要改变这个 XML 的数据格式。例如，在产品的售价旁边显示它的原价，用来说服某个可怜的傻瓜将积压的高尔夫球袜全部买走。

第三，使用数据库的结果集本身来生成 XML。一开始，这似乎是一个有效率的做法，但是它有两个潜在的问题。在生成 XML 的时候，需要保持数据库连接一直打开。这样在 `while()` 循环中，就不能做任何非常困难的事情，否则就有可能长时间占用连接，最终可能会成为系统的一个瓶颈。除此之外，只有当将数据库看作是一个扁平的数据结构时，这样做才是合理的。

3.4.2 重构领域模型

目前在 Garments 表中，使用逗号分隔的列表来保存颜色和尺寸，这是一种相当低效的方式。如果想要按照良好的关系模型来规范化数据，就需要使用一个独立的表来存储所有可用的颜色，并且使用一个关联表在服装和颜色之间建立连接（用数据库的术语来说，这叫做多对多的关系）。图 3-7 展示了这种多对多关系的用法。

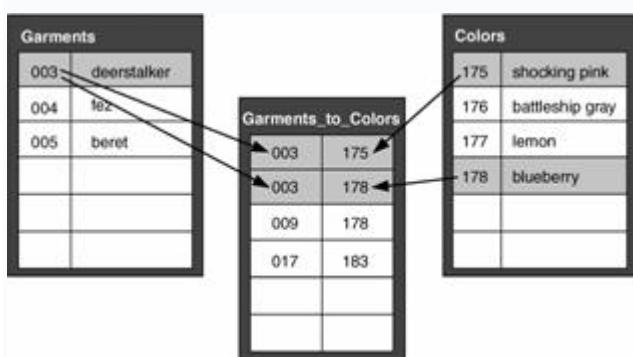


图 3-7 数据库模型中的多对多关系。Colors 表列出了服装的所有可用的颜色，而 Garments 表不再包含任何颜色信息

为了确定猎鹿帽 (deerstalker) 有哪些颜色，需要以 garment_id 作为外键来查找 Garments_to_Colors 表。关联的 color_id 字段指向 Colors 表的主键。我们可以看到，这种帽子有鲜粉红色 (shocking pink) 和蓝莓色 (blueberry)，却没有蓝灰色 (battleship gray)。反过来运行这个查询，可以用一种给定的颜色从 Garments_to_Colors 表中查找所有匹配的产品。

现在我们可以更好地使用数据库了，但是用来获取所有信息的 SQL 变得有些复杂。如果能将服装看作是包含颜色和尺寸的数组对象，而无需手工精心构造那些联接查询，那就太好了。

1. 对象关系映射工具

幸好我们已经有了做这件事的工具和库，就是“对象关系映射”（ORM）工具。ORM 工具可以自动完成数据库中的数据和内存中的对象之间的转换，为开发人员卸掉了编写原始 SQL 的重担。PHP 程序员可以看看 PEAR DB_DataObject、Easy PHP Data Object (EZPDO) 或者 Metastorage。Java 开发者得天独厚地拥有大量的选择，Hibernate (也移植到了.NET 上) 是时下流行的选择。ORM 工具可是一个巨大的话题，我们现在还是将它略过吧，否则说上三天三夜也说不完。

从 MVC 的角度来考察应用，我们会发现采用 ORM 带来了一个令人愉快的副作用，那就是我们开始有了一个真正的模型。现在我们能够编写生成 XML 的程序，与 Garment 对象通信，而让 ORM 工具去跟数据库打交道，不再与特定的数据库 API (以及它的怪癖) 绑定在一起了。代码清单 3-7 展示了使用 ORM 工具之后代码中发生的变化。

在这里，使用 PHP 来为商店例子定义业务对象 (即模型)，使用了 `Pear::DB_DataObject`，这要求类扩展 `DB_DataObject` 这个基类。不同的 ORM 工具做事的方式有所不同，但这里的要点是创建了一组对象，可以像平常的代码一样与它们通信，SQL 语句的复杂性被抽象到了 ORM 工具之中。

代码清单 3-7 服装商店的对象模型

```

require_once "DB/DataObject.php";
class GarmentColor extends DB_DataObject {
    var $id;
    var $garment_id;
    var $color_id;
}
class Color extends DB_DataObject {
    var $id;
    var $name;
}
class Garment extends DB_DataObject {
    var $id;
    var $title;
    var $description;
    var $price;
    var $colors;
    var $category;
    function getColors(){
        if (!isset($this->colors)){
            $linkObject=new GarmentColor();
            $linkObject->garment_id = $this->id;



---


            $linkObject->find();
            $colors=array();
            while ($linkObject->fetch()){
                $colorObject=new Color();
                $colorObject->id=$linkObject->color_id;
                $colorObject->find();
                while ($colorObject->fetch()){
                    $colors[] = clone($colorObject);
                }
            }
        }
        return $colors;
    }
}

```

除了主要的 Garment 对象，我们还定义了 Color 对象，以及可以返回所有可用颜色的 Garment 对象的方法。Size 对象也可以用类似的方式来实现，限于篇幅，在这里省略了。因为这个库不能直接支持多对多的关系，我们需要为连接表定义一个对象类型，并且在 `getColors()` 方法中遍历与这张表对应的对象模型以得到可用的颜色。虽然如此，这仍然是一个相当完整和易读的对象模型。我们来看看如何在页面中使用这个模型。

2. 使用修改后的模型

我们已经从更加清晰的数据结构中生成了一个数据模型，现在需要将这个模型用在 PHP 脚本中。代码清单 3-8 是使用基于 ORM 的对象修改主页面后的代码。

代码清单 3-8 修改后的页面，使用 ORM 与数据库通信

```

<?php
header("Content-type: application/xml");
echo "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n";
include "garment_business_objects.inc"
$garment=new Garment;
$garment->category = $_GET["cat"];
$number_of_rows = $garment->find();
echo "<garments>\n";
while ($garment->fetch()) {
    printf("<garment id=\"%s\" title=\"%s\">\n"
        . "<description>%s</description>\n<price>%s</price>\n",
        $garment->id,
        $garment->title,
        $garment->description,
        $garment->price);
    $colors=$garment->getColors();
    if (count($colors)>0){
        echo "<colors>\n";
        for($i=0;$i<count($colors);$i++){
            echo "<color>{$colors[$i]}</color>\n";
        }
        echo "</colors>\n";
    }
    echo "</garment>\n";
}
echo "</garments>\n";
?>

```

我们包含了对象模型的定义，然后按照这个对象模型来与数据库通信。我们没有专门构造一些 SQL，而是新建了一个空的 Garment 对象，然后使用搜索条件来部分组装它[14]。因为对象模型是从一个单独的文件包括进来的，所以可以在其他的搜索中重用它。XML 视图现在根据对象模型来生成。我们下一步要做的重构是将 XML 的格式从生成它的进程中分离出来。

3.4.3 从表现中分离内容

视图代码仍然与对象模型纠缠在一起，这是因为 XML 的格式和对象解析的代码绑在了一起。如果我们正在维护多个页面，希望能够只在一个地方修改 XML 的格式，然后应用到所有的地方。在要维护多种格式的更加复杂情况下，例如，一种格式用于为消费者提供概要和详细的列表，而另一种格式用于存货盘点的应用，我们希望每一种格式只定义一次，然后为它们提供一个集中的映射。

1. 基于模板的系统

解决这个问题的常见方法是使用模板语言。模板系统接受一个包含一些特殊标记符号的文档，这些标记符号充当占位符，在执行过程中由真正的变量值替换。PHP、ASP 和 JSP 本身就是这类的模板语言，它们是在 Web 页面中嵌入代码，而不是像 Java servlet 和传统的 CGI 脚本，在代码中嵌入内容。然而，它们将脚本语言的全部能力都公开给了页面，使得很容易将业务逻辑和表现混杂在一起。

与之相反，特定用途（purpose-built）的模板语言，例如 PHP Smarty 和 Apache Velocity（一种基于 Java 的系统，也移植到了.NET 上，称作 NVelocity），为代码提供了更加有限的能力，通常是有限的控制流程，例如简单的分支（例如 `if`）和循环（例如 `for`、`while`）结构。代码清单 3-9 展示了一个生成 XML 输出的 PHP Smarty 模板。

代码清单 3-9 生成 XML 输出的 PHP Smarty 模板

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

<garments>

{section name=garment loop=$garments}

<garment id="{{$garment.id}}" title="{{$garment.title}}>

    <description>{{$garment.description}}</description>

        <price>{{$garment.price}}</price>

    {if count($garment.getColors())>0}

        <colors>

            {section name=color loop=$garment.getColors()}

                <color>$color->name</color>

            {/section}

        </colors>

    {/if}

    </garment>

{/section}

</garments>
```

模板期望得到的输入是名为 `garments` 的数组，它包含了一些 `Garment` 对象。模板中的大部分内容都会原封不动地输出到结果文档中，但是花括号之中的部分会解释为指令，它们要么被看作是需要替换的变量名，要么被看作是简单的 `branch` 和 `loop` 语句。与代码清单 3-7 中的代码相比，在这个模板中，输出 XML 文档的结构显然更加易读。我们来看看如何在页面中使用这个模板。

2. 使用修改后的视图

我们已经将 XML 格式的定义从主页面中移到了 Smarty 模板中。这样的话，主页面只需要设置模板引擎然后传给它合适的数据就行了。代码清单 3-10 展示了需要做出的修改。

代码清单 3-10 使用 Smarty 来生成 XML

```
<?php  
header("Content-type: application/xml");  
include "garment_business_objects.inc";  
include "smarty.class.php";  
$garment=new DataObjects_Garment;  
$garment->category = $_GET["cat"];  
$number_of_rows = $garment->find();  
$smarty=new Smarty;  
$smarty->assign('garments',$garments);  
$smarty->display('garments_xml.tpl');  
?>
```

Smarty 使用起来非常简单，只需要遵循三个步骤就可以了。首先，创建一个 Smarty 引擎。然后，使用变量来组装它，在这个例子只有一个变量，但是其实可以增加任意多个变量。例如，如果用户的详细信息保存在会话中，就可以将它们传递进去，然后通过模板来显示一个个性化的问候信息。最后，调用 `display()` 方法，将模板文件的名称传递进去。

现在，从搜索结果的页面中分离视图已经达到了一种令人愉快的状态。XML 格式只需要定义一次，然后通过几行代码就可以使用这个格式。搜索结果的页面现在变得非常专注，只包含特定于它自身的信息，即组装搜索参数和定义输出格式。还记得前面所构思的“在线切换 XML 格式”的需求吗？使用 Smarty 来做

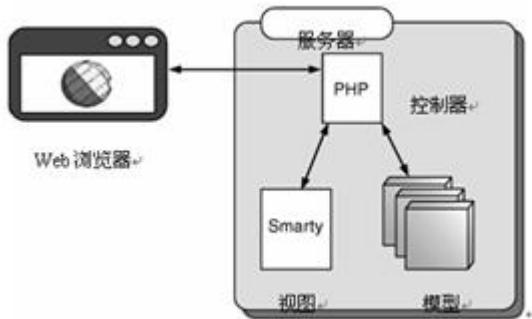
这件事情是很简单的，只需要再定义一个额外的格式就行了。如果我们想要追求更高程度的结构化，即在文档之中创建更小的变化，Smarty甚至还支持在模板之中包括其他的模板。

回顾一下最初对模型—视图—控制器模式的讨论，可以看到现在的实现已经相当令人满意了。图 3-8 形象地概括了目前所采用的方法。

图 3-8 MVC 应用在 Web 应用中一般方式。Web 页面或 servlet 扮演控制器的角色，它首先查询模型以得到相关的数据，然后将这些数据传递给模板文件（视图），视图再生成发送给用户的内容。注意，这里是只读的情况，如果要修改模型，事件的流程会略微不同，但是角色是一样的

模型是领域对象的集合，使用 ORM 工具自动持久化到数据库中。视图是定义 XML 格式的模板。控制器是“按照类别查询”的页面，以及定义的其他页面，这些页面将模型和视图粘合在了一起。

这就是 MVC 在 Web 应用中的传统使用方式。这里，我们在 Ajax 应用的 Web 服务器层使用



这种方式来为客户端提供 XML 文档，可以很容易地看到，它同样适用于那些提供 HTML 页面的传统 Web 应用。

依赖于所使用的技术，你可能会碰到这个模式的不同变种，但是它们的原理都是一样的。J2EE 的企业 beans 对模型和控制器加以抽象，使得它们可以位于不同的服务器上。.NET 中的“代码隐藏”(code-behind) 类将控制器的角色委派给了特定于页面的对象，然而类似 Struts 的框架定义了一个“前端控制器”，用来拦截和分发所有到应用的请求。类似 Apache Struts 的框架做这些事可谓是驾轻就熟，它将控制器的职责精练为仅仅负责在页面之间对于用户加以引导，我们也可以在单个页面的级别上实现相同的功能（在 Ajax 应用中，可以使用 JavaScript 来做这件事[15]）。但是在所有的情况下，映射基本上都是相同的，这就是 MVC 在 Web 应用领域中通常被理解成的样子。

使用 MVC 来描述 Web 架构是一种非常有用的方法，随着从传统的 Web 应用过渡到 Ajax 风格的应用，MVC 仍然会一如既往地支持我们。但是这里并不是在 Ajax 中用到 MVC 的唯一地方。在第 4 章中，我们将会考察这个模式的一个变种，可以使整个应用获得结构化设计的好处。在此之前，我们考察另外一种为 Ajax 应用引入秩序的方法。

除了对自己的代码进行重构，我们也经常使用第三方的框架和库来使代码更加合理。随着业界对于 Ajax 兴趣的增长，很多有用的框架浮现了出来，我们简要地介绍其中一些较为流行的框架以此结束本章。

3.5 第三方的库和框架

通过将完成细节工作的代码提取到通用的函数或者对象中，来减少代码库中重复代码的数量，是大多数重构的目的。顺着这个思路，我们可以将那些通用的功能包装为可以在不同项目中重用的库或者框架。这样做减少了项目中需要定制的代码数量，并且可以提高生产力。更进一步，因为库的代码已经在以前的项目中经过了测试，项目的质量也有望得到提高。

在本书中，我们会开发出一些可以在项目中重用的小 JavaScript 框架，包括第 4 章和第 5 章的 ObjectBrowser、第 5 章的 CommandQueue、第 6 章的通知框架、第 8 章的 StopWatch 性能分析工具以及附录 A 中的调试控制台。从第 9 章到第 13 章，我们在每章的末尾都会对教学的例子做重构，以便提供可以重用的组件。

当然，我们并不是这个游戏的唯一玩家，因特网上还有大量可用的 JavaScript 和 Ajax 的框架，其中较为成熟的一些框架已经经过了大量开发者的深入测试。

在本节中，我们将考察 Ajax 社区中的一些第三方库和框架。Ajax 框架这一领域目前处在非常活跃的发展阶段，所以无法详细讨论所有的竞争者，但是我们将尽力使你了解，目前已经有哪些类型的框架，以及如何通过使用它们来为自己的项目引入秩序。

3.5.1 跨浏览器库

正如 3.2.1 小节中提到的，对于编写 Ajax 应用，跨浏览器不一致性的问题近在咫尺。很多库都提供了一个令开发者可以在其上进行开发的通用 Façade，以此避开跨浏览器的不一致性问题，从而实现非常有

用的功能。一些库集中于一些特定的功能，而另外一些库则试图提供更加全面的编程环境。下面提到的库属于后一类库，当编写 Ajax 代码时，它们很有帮助。

1. x 库

x 库是一个用来编写 DHTML 应用的成熟、通用的库。第 1 版发布于 2001 年，取代了作者之前开发的 CBE (Cross Browser Extensions, 跨浏览器扩展) 库，而且使用了简单得多的编程风格。它提供了跨浏览器的函数，用来操作 DOM 元素、为 DOM 元素设置样式、处理浏览器的事件模型，还包括了支持动画和拖拽的一些开箱即用的库。它支持 IE 4 和最近版本的 Opera 和 Mozilla 浏览器。

x 库使用了一种简单的基于函数的编码风格，利用了 JavaScript 的可变参数列表和弱类型的特征。例如，`document.getElementById()` 方法只接受字符串作为输入，x 库将它包装为另外一个函数，既可以接受字符串也可以接受 DOM 元素。如果传入的是字符串，就将它作为元素的 ID 来确定 DOM 元素；但是如果传入的是 DOM 元素，就将它原封不动地返回。因此，可以调用 `xGetElementById()` 来保证参数通过 ID 确定为 DOM 节点，而不必测试这个参数是否是已经确定的 DOM 节点。在创建动态生成的代码时用 DOM 元素的文本 ID 代替 DOM 元素本身的能力特别有用，例如当传递一个字符串给 `setTimeout()` 方法或者回调函数的时候。

在操作 DOM 元素样式的函数中，也使用了类似的简练风格，相同的函数既可以用来获取属性的值，也可以用来设置属性的值。例如语句：

```
xWidth(myElement)
```

会返回 DOM 元素 `myElement` 的宽度，其中 `myElement` 既可以是 DOM 元素，也可以是 DOM 元素的 ID。通过像下面这样增加一个额外的参数：

```
xWidth(myElement,420)
```

我们设置了元素的宽度。因此，为了将一个元素的宽度设置为与另一个元素的宽度相同，可以这样写：

```
xWidth(secondElement,xWidth(firstElement))
```

x 库没有包含任何用于创建网络请求的代码。虽然如此，对于建造 Ajax 应用的用户界面，以及使得代码具有清晰、易懂风格，它仍然是一个很有用的库。

2. Sarissa

与 x 库相比，Sarissa 的目标更加明确，它主要关注于使用 JavaScript 来操作 XML。它可以支持 IE 的 MSXML ActiveX 组件（第 3 版以上）、Mozilla、Opera、Konqueror 和 Safari 的基本功能。不过只有一小部分浏览器可以支持一些较为高级的特征（例如 XPath 和 XSLT）。

对于 Ajax 开发者来说，它最重要的功能就是为 XMLHttpRequest 对象提供了跨浏览器的支持。在不提供原生的 XMLHttpRequest 对象的浏览器上（主要是 IE），Sarissa 使用 Adapter 模式创建了一个基于 JavaScript 的 XMLHttpRequest 对象，而不是创建一个自己的 Façade 对象。这个对象的内部实现还是要使用在第 2 章中描述过的 ActiveX 对象，但是对于开发者来说，他们只关注在引入了 Sarissa 之后，下面的代码就可以运行在任何浏览器中：

```
var xhr = new XMLHttpRequest();

xhr.open("GET", "myData.xml");

xhr.onreadystatechange = function(){

if(xhr.readyState == 4){

    alert(xhr.responseXML);

}

}

xhr.send(null);
```

将这段代码与代码清单 2-11 中的代码进行比较，我们会注意到这里的 API 调用与 Mozilla 和 Safari 浏览器中原生的 XMLHttpRequest 对象的 API 完全一样。

如同已经提到的那样，Sarissa 还为操作 XML 文档提供了很多通用的支持。例如，它能够完成 JavaScript 对象和 XML 之间的序列化。如果你的项目使用 XML 作为返回数据的标记语言，这些机制对于处理发到服务器端的 Ajax 请求所返回的 XML 文档是很有用的（我们会在第 5 章讨论这个问题以及它的替代方案）。

3. Prototype

Prototype 是为应用 JavaScript 编程开发的一个通用的帮助库 (helper library)，其重点在于扩展 JavaScript 语言本身，以便支持更加面向对象的编程风格。基于它所增加的这些语言特征，Prototype 的 JavaScript 代码有一种与众不同的编码风格。尽管 Prototype 本身的代码很难读懂，与 Java 和 C# 的编码风格大相径庭，但是使用 Prototype 或者在 Prototype 之上建造的库却是非常简洁明了的。Prototype 可

以看作是库的开发人员所使用的库。Ajax 应用的开发人员更喜欢使用建造在 Prototype 之上的库，而不是直接使用 Prototype 本身。在下面几个小节我们会考察几个这样的库。在此之前，简要地讨论一下 Prototype 的核心特征有助于介绍它的代码风格，并且对于后面讨论的 Scriptaculous、Rico 以及 Ruby on Rails，也会很有帮助。

Prototype 允许通过将父对象的所有属性和方法复制到子对象，将一个对象“扩展”为另一个对象。这个特征最好还是通过例子来展示。假设我们定义了一个父类 `Vehicle`:

```
function Vehicle(numWheels,maxSpeed){  
    this.numWheels=numWheels;  
    this.maxSpeed=maxSpeed;  
}
```

我们需要定义一个特定的实例用来代表一趟旅客列车。在子类中，我们还想要表示车厢的数量，并且提供一种机制来增加和减少车厢。使用普通的 JavaScript，可以这样写：

```
var passTrain=new Vehicle(24,100);  
  
passTrain.carriageCount=12;  
  
passTrain.addCarriage=function(){  
    this.carriageCount++;  
}  
  
passTrain.removeCarriage=function(){  
    this.carriageCount--;  
}
```

这段代码提供了 `passTrain` 对象所需要的功能。然而从设计的角度来考察这段代码，它对于将扩展的功能封装进一个一致的单元几乎没有什么帮助。在这里，Prototype 可以帮助我们将这些扩展的行为定义为一个对象，然后再来扩展这个基对象[16]。首先，我们将扩展的功能定义为一个对象：

```
function CarriagePuller(carriageCount){  
    this.carriageCount=carriageCount;  
    this.addCarriage=function(){
```

```
this.carriageCount++;

}

this.removeCarriage=function(){

    this.carriageCount--;

}

}
```

然后，可以将这两个对象的功能合并在一个对象之中，以获得所有需要的行为：

```
var parent=new Vehicle(24,100);

var extension=new CarriagePuller(12);

var passTrain=Object.extend(parent,extension);
```

注意，我们首先分别定义了父对象和扩展对象，然后将它们合并在一起。父子关系存在于这些实例之间，而不是存在于 `Vehicle` 和 `CarriagePuller` 类之间。准确地说，这并不是传统的面向对象的用法，但是它允许我们将与某个特定功能相关的代码放在一起（在这里就是与拖动车厢相关的功能），这样就可以很容易地重用这些代码。尽管在一个这样的小例子中似乎没有必要这样做，但是在大的项目中，以这种方式来封装某个功能是非常有用的。

Prototype 也以 Ajax 对象的形式提供了对于 Ajax 的支持，通过它可以获得一个跨浏览器的 `XMLHttpRequest` 对象。Ajax 对象可以通过 `Ajax.Request` 类别来扩展，该类型使用 `XMLHttpRequest` 向服务器发送请求，就像这样：

```
var req=new Ajax.Request('myData.xml');
```

这个构造函数使用了一种在很多基于 Prototype 的库中也能看到的风格。它接受一个关联数组作为可选的参数，允许根据需要设置各种广泛的选项。每个选项都提供有明智的默认值，所以只需要传进那些我们想要覆盖的对象就行了。对于 `Ajax.Request` 的构造函数来说，选项数组可以是 post 数据、请求参数、HTTP 方法以及定义的回调处理函数。一个更加自定义的 `Ajax.Request` 调用可能像下面这样：

```

var req=new Ajax.Request(
  'myData.xml',
  {
    method: 'get',
    parameters: { name:'dave',likes:'chocolate,rhubarb' },
    onLoaded: function(){ alert('loaded!'); },
    onComplete: function(){
      alert('done!\n\n'+req.transport.responseText);
    }
  }
);

```

这里的这个选项数组中传进来了 4 个参数。我们将 HTTP 方法设置为 `get`，因为 Prototype 默认使用 `post` 方法。因为使用了 HTTP 的 `get` 方法，参数数组会通过请求字符串来传递。如果使用的是 `post` 方法，参数会通过请求的主体部分来传递。`onLoaded` 和 `onComplete` 是回调事件处理函数，当底层 XMLHttpRequest 对象的 `readyState` 发生变化时将会被调用。`onComplete` 函数中的 `req.transport` 变量是到底层 XMLHttpRequest 对象的引用。

在 `Ajax.Request` 之上，Prototype 更进一步定义了对象的 `Ajax.Updater` 类型，用来获取服务器端生成的一段 JavaScript 脚本，然后执行它。这遵循了我们在第 5 章中描述的“以脚本为中心”的模式，已经超出了这里的讨论范围。

到这里，我们关于跨浏览器库的简要介绍就结束了。我们对这些库的选择在某种程度上有点武断，而且是不完整的。正如我们所提到的，目前这一领域正处在非常活跃的发展阶段，我们只能局限在那些较为流行或者更加成熟的库上。在下一小节，我们将考察一些 UI 组件的框架，它们建造于上面这些库和其他的一些库之上。

3.5.2 UI 组件和 UI 组件套件

到目前为止，我们所讨论的库为一些相当底层的功能提供了跨浏览器的支持，例如操作 DOM 元素和从服务器获取资源。有了这些工具，确实能够简化了构建用户界面和开发应用逻辑的工作，然而与对应的 Swing、MFC 或者 Qt 这些框架相比，我们仍然需要做大量繁重的工作。

供 Ajax 开发者使用的预制 UI 组件，甚至是完整的 UI 组件集，目前才刚刚开始浮出水面。在本小节中，我们来考察一下其中的几个成员，当然更多的是为本书增加一些特色，而不是做全面的综述。

1. Scriptaculous

Scriptaculous 库是建造在 Prototype（参见上一小节）之上的 UI 组件，它的当前版本提供了两部分主要的功能，不过目前针对它的开发很活跃，几个其他的特征已经列入了开发计划。

Scriptaculous 库的第一部分功能是它的 Effects (效果) 库，它定义了一些可以应用在 DOM 元素上的动画效果，用来改变 DOM 元素的大小、位置和透明度。这些效果可以很容易组合在一起使用，此外还有很多预定义的辅助效果，例如 `Puff()`，可以使一个元素变得越来越大、越来越透明，直到它完全从屏幕上淡出。另外一个有用的核心效果是 `Parallel()`，它能够同时执行多个效果。当需要迅速为 Ajax 用户界面添加各种视觉反馈时，Effects 库是很有用的，在第 6 章中将会看到这一点。

执行一个预先确定的效果非常简单，只需要调用它的构造函数，并且将目标 DOM 元素或者它的 ID 作为参数传递进来，例如：

```
new Effect.SlideDown(myDOMElement);
```

在各种效果之下是一个渐变对象的概念。这个对象接受两个参数，一个持续时间和一个当渐变过程结束时会调用的事件处理函数。它提供了一些基本的渐变类型，例如线性、正弦、钟摆和脉冲。创建一种自定义的效果，只需要将一些核心效果组合在一起，并且将适当的参数传递进来即可。创建自定义效果的详细讨论已经超出了简要介绍的范围。在第 6 章中，当开发一个通知系统的时候，还会用到 Scriptaculous 库的效果。

Scriptaculous 库的第二部分功能是它通过 `Sortable` 类提供了一个拖放库。这个类使用一个父 DOM 节点作为参数，使得它的所有子节点都可以进行拖放操作。传进构造函数的选项可以指定当节点被拖放时的回调处理函数、能够拖动的子元素类型，以及能够作为释放目标的元素列表（即接受用户通过鼠标拖动项的元素）。Effect 对象也可以作为选项传递进来，用来在开始拖动、拖动过程中和释放的时候执行这些效果。

2. Rico

Rico 和 Scriptaculous 一样也是基于 Prototyp 库的，它也提供了一些高度可定制的效果和拖放功能。除此之外，它还给出了一个 Behavior 对象的概念，也就是一段代码，可以应用在 DOM 树的一部分，为它增加交互功能。Rico 提供了少量示例的 Behavior，例如，Accordion (折叠) UI 组件，它可以将一组 DOM 元素嵌套在一个给定的空间内，每次展开其中的一个（这种风格的 UI 组件常常称作 outlook bar，在微软的 Outlook 中使用之后变得非常流行）。

我们来建造一个简单的 Rico Accordion UI 组件。一开始，我们需要一个父 DOM 元素，它的每个子节点都会成为一个折叠的面板。我们为每个面板定义一个 DIV 元素，其中还包含有另外两个 DIV，表示每个面板的标题和和主体部分：

```
<div id='myAccordion'>
<div>
  <div>Dictionary Definition</div>
  <div>
    <ul>
      <li><b>n.</b>A portable wind instrument with a small keyboard and free metal reeds that sound when air is forced past them by pleated bellows operated by the player.</li>
      <li><b>adj.</b>Having folds or bends like the bellows of an accordion: accordion pleats; accordion blinds.</li>
    </ul>
  </div>
</div>
<div>
  <div>A picture</div>
  <div>
    <img src='monkey-accordion.jpg'></img>
  </div>
</div>
</div>
```

第一个面板提供了 accordion[17] 这个词在字典中的定义；第二个面板包含了一个猴子玩手风琴的图片（参见图 3-9）。直接呈现上面这段 HTML，仅仅是将这两个元素上下显示。然而，我们给顶层的 DIV 元素分配了一个 ID 属性，使得可以将它的引用传递给 Accordion 对象，像这样来构造：

```
var outer=$('#myAccordion');

outer.style.width='320px';

new Rico.Accordion(
  outer,
  {
    panelHeight:400,
    expandedBg:'#909090',
    collapsedBg:'#404040',
  }
);
```

第一行看起来有点古怪。\$ 实际上是一个合法的 JavaScript 变量名，它指向核心 Prototype 库中的一个函数。\$() 函数以一种类似于上一小节中讨论的 x 库 xGetElementById() 函数的方式来确定 DOM 节点。我们将一个已确定的 DOM 元素的引用传递给 Accordion 对象的构造函数，并伴随一个符合源自 Prototype 库

的标准习惯的选项数组。这里仅仅提供了一些设置 Accordion UI 组件的视觉风格的选项，不过还可以在这里提供当面板打开或关闭时触发的回调处理函数。图 3-9 展现了使用 Accordion 对象来设置 DOM 元素样式之后的效果。Rico 的 Behavior 提供了一种简单的方式来从通常的标记创建可重用的 UI 组件，并且还将内容与交互分离开。在第 4 章，我们探讨将良好的设计原则应用在 JavaScript 用户界面开发中的话题。



图 3-9 Rico 框架的 Behaviors 库可以为简单的 DOM 节点设置样式，使得它变成交互的 UI 组件，这只需要将顶层节点的引用传给 Behavior 对象的构造函数就行。在这里，Accordion 对象应用在一组 DIV 元素（图中左边）上，以创建一个交互的菜单 UI 组件（图中右边），通过鼠标点击来打开和关闭单个面板

需要提到的最后一个 Rico 框架的功能，是它通过全局的 Rico AjaxEngine 对象为 Ajax 风格的服务器请求提供了非常棒的支持。AjaxEngine 所提供的不仅仅是 XMLHttpRequest 对象的一个跨浏览器的包装，它还定义了一种由很多<response>元素组成的 XML 响应格式。引擎会自动对这些响应进行解码，它内建了对于两种类型的响应的支持：直接更新 DOM 元素的响应类型和更新 JavaScript 对象的响应类型。在 5.5.3 节深入讨论客户/服务器交互的时候，我们还会更详细地考察一个类似的机制。现在，我们转到下一种类型的框架：一种跨越客户端与服务器的框架。

3.5.3 应用框架

到目前为止，我们所考察的框架都是完全在浏览器中执行的，可以作为静态的 JavaScript 文件从任何 Web 服务器提供。我们在这里将要介绍的最后一类框架位于服务器端，至少会动态地生成一些 JavaScript 代码或 HTML 标记。

在所讨论的框架中，这些框架是最复杂的，我们无法非常详细地讨论它们，而只能对其功能做一些简要的介绍。在第 5 章中，我们还会回到这个有关服务器端框架的话题。

1. DWR、JSON-RPC 和 SAJAX

我们首先一起考察一下这三个框架。尽管它们是使用不同的服务器端语言编写的，但是它们其实是一类共同的方法。SAJAX 可以与多种服务器端语言共同使用，包括 PHP、Python、Perl 和 Ruby。DWR（直接 Web 远程调用）是一个以类似方法实现的基于 Java 的框架，它能够将对象的方法暴露出来，而不是像 SAJAX 那样将独立的函数暴露出来。JSON-RPC（基于 JSON 的远程过程调用）在设计上也是相似的，它提供了对于服务器端的 JavaScript、Python、Ruby、Perl 和 Java 的支持。

它们都允许将定义在服务器端的对象上的方法直接暴露给 Ajax 请求。我们常常在这些服务器端的函数中执行那些必须要在服务器端执行的计算（例如，从数据库查询一个值），然后再将有用的结果返回给客户端。这些框架为从 Web 浏览器中访问这些函数或方法提供了方便的途径，并且也是将服务器端的领域模型暴露给 Web 浏览器代码的好方法。

我们来考察一个使用 SAJAX 的例子，其暴露了一个在服务器端用 PHP 定义的函数。这个例子非常简单，仅仅返回一个文本字符串，就像下面这样：

```
<?php  
  
function sayHello(name){  
  
    return("Hello! {$name} Ajax in Action!!!!");  
  
?>
```

为了将这个函数暴露给 JavaScript 层，我们只需要在 PHP 中引入 SAJAX 引擎，然后调用 `sajax_export` 函数：

```
<?php  
  
require('Sajax.php');  
  
sajax_init();  
  
sajax_export("sayHello");  
  
?>
```

当随后编写动态 Web 页面的时候，我们使用 SAJAX 来为这个暴露的函数生成一些 JavaScript 包装代码。生成的代码创建了一个本地的 JavaScript 函数，具有和服务器端函数完全相同的签名：

```
<script type='text/javascript'>
```

```
<?  
    sajax_show_javascript();  
?>  
...  
alert(sayHello("Dave"));  
...  
</script>
```

当我们在浏览器中调用 `sayHello("Dave")` 时，生成的 JavaScript 代码会向服务器发送一个 Ajax 请求，执行服务器端的函数，并且在 HTTP 响应中返回结果。然后它会解析 HTTP 响应，从中提出返回值，传给调用它的 JavaScript 代码。开发者无需接触到任何 Ajax 技术，所有的底层工作都通过 SAJAX 库以后台的方式来处理。

这三个框架提供了一个从服务器端函数和对象到客户端的 Ajax 调用的相当底层的映射。这些枯燥的任务可以自动地完成，但是它们确实也引入了将服务器端的逻辑过多暴露给因特网的危险。我们将在第 5 章中更加详细地讨论这个问题。

我们在本节中将要考察的其余框架则采用了更为复杂的方法，它们在服务器端根据声明的模型生成整个 UI 层。虽然在它们内部也使用了标准的 Ajax 技术，但是从本质上讲，这些框架其实是提供了它们自己的编程模型。带来的结果就是，使用这些框架来编程与编写普通的 Ajax 代码大不相同，我们在这里只做一些概括的介绍。

2. BackBase

BackBase 表现服务器 (BackBase Presentation Server) 提供了一套丰富的 UI 组件，这些 UI 组件可以与服务器端生成的、嵌入在 HTML 文档中的 XML 标签进行实时绑定。这里的原理类似于 Rich 的 Behavior 组件，只不过 BackBase 使用了自定义的 XHTML 标签集以标记 UI 组件，而 Behavior 使用的是标准 HTML 标签。

BackBase 为 Java 和 .NET 提供了服务器端的实现。它是一个商业化的产品，但是也提供了可以免费使用的社区版本。

3. Echo2

NextApp 的 Echo2 框架是一个基于 Java 的服务器引擎，它可以根据在服务器端声明的用户界面模型生成丰富的 UI 组件。一旦在浏览器中启动后，UI 组件就以一种相当自治的方式来运行，要么使用 JavaScript 在本地处理用户交互；要么使用一个类似于 Rico 中使用的请求队列批量地将请求发送回服务器。

Echo2 将自己提升为一个基于 Ajax 的解决方案[18]，开发者无需具备 HTML、JavaScript 或 CSS 的知识，除非你想要扩展现有组件集合。在大多数情况下，客户端应用的开发只需要使用 Java。Echo2 是一个开源项目，经 Mozilla 风格认证许可，可以用于商业应用。

4. Ruby on Rails

Ruby on Rails 是一个使用 Ruby 编程语言编写的 Web 开发框架。它将一些解决方案打包在一起，支持服务器端对象与数据库中的数据之间的映射[19]；还支持使用模板来生成内容，其风格非常类似于 3.4 节讨论过的服务器端 MVC。Ruby on Rails 宣称能够非常快速地开发简单的和中等规模的网站，因为它使用代码生成技术来生成大量通用的代码。它还试图用尽可能少的配置数量让应用跑起来。

在最近的版本中，Rails 通过 Prototype 库提供了强大的 Ajax 支持。Prototype 和 Ruby on Rails 是天生的一对，因为 Prototype 的 JavaScript 代码就是使用 Ruby 程序生成的，并且它们的编程风格也很相似。和 Echo2 一样，在 Rails 中使用 Ajax 也不要求开发者很了解 JavaScript 等 Ajax 技术，但是一个确实理解 JavaScript 的开发者，将能够以新的方式扩展 Rails 的 Ajax 支持。

到这里，我们对于第三方 Ajax 库和框架的介绍就结束了。正如我们已经提到的那样，这是一个目前正在迅速发展的领域，大部分在这里讨论的框架仍然处在活跃的开发阶段。

很多库和框架都有自己的编码习惯和风格。在为本书编写例子代码的过程中，我们尽力使读者体会到 Ajax 技术覆盖范围之广，而避免使用那些需要对某种特殊框架进行大量学习的技术。尽管如此，在本书的其余部分，你仍然会遇到一些在此讨论过的框架。

3.6 小结

在本章中，我们引入了重构的概念，将它作为一种改善代码的质量和灵活性的方法。我们的第一个重构是将 Ajax 技术体系中最为核心的 XMLHttpRequest 对象封装起来，成为一个简单的、可重用的对象。

我们考察了很多设计模式，可以应用它们来解决一些在进行 Ajax 开发时通常会遇到的问题。设计模式为我们提供了一个非正式的方法，通过它可以获得其他遇到过相同问题的程序员所积累的知识，并且帮助我们向着具体的目标进行重构。

Façade 模式和 Adapter 模式为掩盖不同实现之间的差异提供了很有用的方法。在 Ajax 中，这些模式特别有用，通过它们可以为跨浏览器不兼容性提供一个绝缘层，而这正是长期以来一直困扰 JavaScript 开发者的一个主要问题。

Observer 模式对于事件驱动的系统是一个很灵活的模式。在第 4 章中，当考察应用的 UI 层时，我们还会回到这个模式。通过联合使用 Observer 模式和 Command 模式，我们可以很好地封装用户的交互，从而有可能开发出健壮的框架，以处理用户的输入和提供撤销的功能。Command 模式还可以用于组织客户/服务器之间的交互，我们在第 5 章就会看到。

Singleton 模式为控制对于特定资源的访问提供了一种简单的方法。在 Ajax 中，我们可以有效地使用 Singleton 模式来控制对网络的访问，我们在第 5 章就会看到。

最后，我们引入了 MVC 模式，这是一个在 Web 应用中有很长使用历史的架构模式（至少从因特网的时间来看是这样）。我们讨论了 MVC 模式如何通过使用一个抽象的数据层和一个模板系统，来改善服务器端应用的灵活性。

服装店的例子也展示了设计模式和重构是如何协同工作的。第一次就写出具有完美设计的代码是困难的，但是重构类似代码清单 3-4 的丑陋但能用的代码，逐渐引入设计模式的好处，是完全可能的，最终的结果也非常令人满意。

最后，我们考察了一些第三方的库和框架，使用这些库和框架也可以为 Ajax 项目引入秩序。目前这个阶段，很多库和框架正在茁壮成长，从简单的跨浏览器的封装到完整的 UI 组件套件，再到跨越客户端和服务器的端到端解决方案。我们简要介绍了几个较为流行的框架，在后面的章节还会回到其中的一些框架上。

在后面的两章中，我们会将对于重构和设计模式的理解应用到 Ajax 客户端应用和客户/服务器的通信系统之中。这将有助于我们发展出一套词汇表，找到使得开发健壮的、多功能的 Web 应用更加容易的一些最佳实践。

3.7 资源

Martin Fowler（以及合著者Kent Beck、John Brant、William Opdyke和Don Roberts）写了开创性的重构指南：*Refactoring: Improving the Design of Existing Code*[20]（Addison-Wesley 1999 年出版）。

Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides（也称作GoF）写了影响深远的*Design Patterns*[21]（Addison-Wesley 1995 年出版）。

Gamma 后来成为了 Eclipse IDE/平台架构师，他在最近一次访谈中讨论了 Eclipse 和设计模式：www.artima.com/lejava/articles/gammadp.html。

Michael Mahemoff 最近建立了一个网站，对 Ajax 的设计模式进行分类：www.ajaxpatterns.org。

第4章 作为应用的页面

本章内容

- 组织复杂的用户界面代码
- 使用 JavaScript 实现模型—视图—控制器模式
- 为得到易维护的代码分离表现和逻辑
- 创建灵活的事件处理模式
- 直接从业务对象创建用户界面

第 1 章和第 2 章从可用性和技术的角度介绍了 Ajax 的基本原理，第 3 章简单谈到了通过重构和设计模式创建易维护（*maintainable*）代码的概念。在我们见过的例子中，这些方法看起来似乎是“杀鸡用牛刀”，但是随着我们更深入地探索 Ajax 编程，将会看到这些方法其实是不可缺少的。

在本章和下一章，我们讨论创建大型、可伸缩的 Ajax 客户端，以及达到这个目的所需要的架构原理。本章只考察客户端的代码，主要考察在第 3 章中讨论过的模型—视图—控制器（MVC）模式。我们在这个过程中也将遇到 Observer 和其他比较小的模式。第 5 章将考察客户端和服务器端之间的关系。

4.1 一种不同类型的 MVC

第 3 章介绍了将一个简单的服装店应用程序重构为符合 MVC 模式的例子。大部分 Web 开发者以前曾经遇到过这种 MVC：模型是服务器端的领域模型，视图是生成的发送给客户端的内容，控制器是一个 servlet 或一组定义应用的工作流页面。

MVC 最初来自于桌面应用开发，但在 Ajax 应用中有几个场合可以用它来很好地为我们服务。让我们来看看这些场合。

4.1.1 以不同的规模重复 MVC 模式

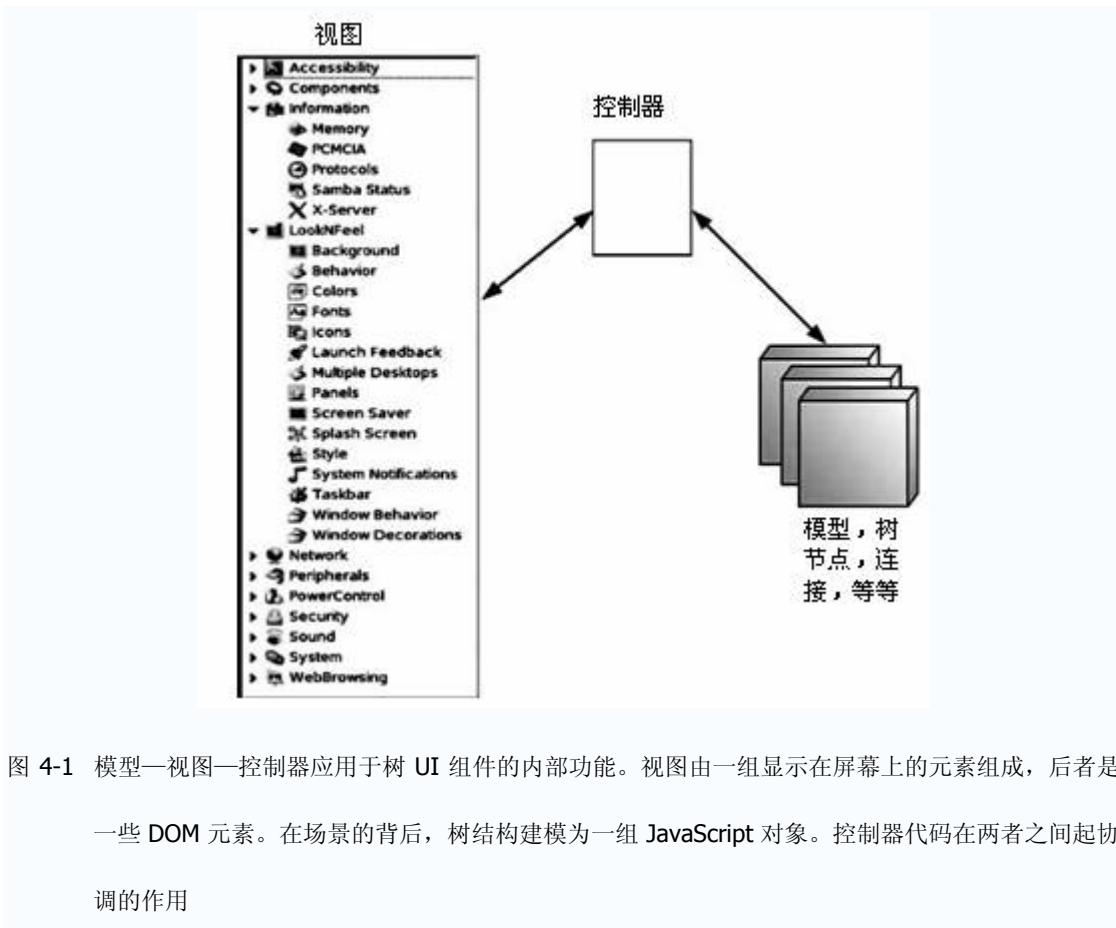
经典的 Web MVC 模式以粗粒度的规模描述完整的应用。生成的整个数据流是视图，整个 CGI 或 servlet 层是控制器，等等。

在桌面应用开发中，MVC 模式常常以粒度细得多的规模被应用，像按钮这样简单的 UI 组件也可以使用 MVC：

- 状态的内部表示（例如压下、放开、不活动）是模型。Ajax UI 组件通常实现为 JavaScript 对象。
- 显示在屏幕上、由 DOM（文档对象模型）节点组成的 UI 组件，在 Ajax 用户界面中（不同状态的修改、突出显示、工具提示）是视图。
- 将两者关联起来的内部代码是控制器。事件处理函数代码（即当用户按下按钮时，在更大的应用中发生的事情）也是控制器，但不是这个视图和模型的控制器。我们不久就会看到。

孤立的按钮只有很少的行为、状态或者可视的变化，所以在这里使用 MVC 的意义相当小。如果我们考察一个更加复杂的 UI 组件，例如一棵树或者一张表格，整个系统就足够复杂，可以从基于 MVC 的设计中得益甚多。

图 4-1 展示了将 MVC 应用于树 UI 组件。模型由树的节点组成，每个节点有一个子节点列表、一个打开/关闭的状态和一个到某些业务对象的引用，每个节点代表一个文件浏览器中的文件和目录。视图由图标和在 UI 组件画布上画的线组成。控制器处理用户事件，例如打开和关闭节点、显示弹出菜单、为特定的节点触发图形更新调用，允许视图增量地刷新自己。



这就是在熟悉的 Web 服务器场景之外应用 MVC 的一种方式。但是还不完整，让我们先将注意力集中于 Web 浏览器。

4.1.2 在浏览器端应用 MVC

我们在前面一直将注意力集中于应用中的小细节。现在可以扩大一下视野，考虑启动时交付在浏览器上的完整的 JavaScript 应用。这也按 MVC 模式进行结构化，由于清晰地分离了关注点，得到较大的优化。

在这个级别，模型由业务领域对象组成，视图是整个可编程处理的页面，控制器是将 UI 和领域对象相连接的代码中所有事件处理函数的组合。图 4-2 展示了这个级别的 MVC 操作。这可能是对于 Ajax 开发者最重要的 MVC 使用方式，因为它很自然地适应了 Ajax 富客户应用。我们将考察 MVC 模式的这种使用方法的细节，并在本章的剩余部分看看能从中吸取些什么。

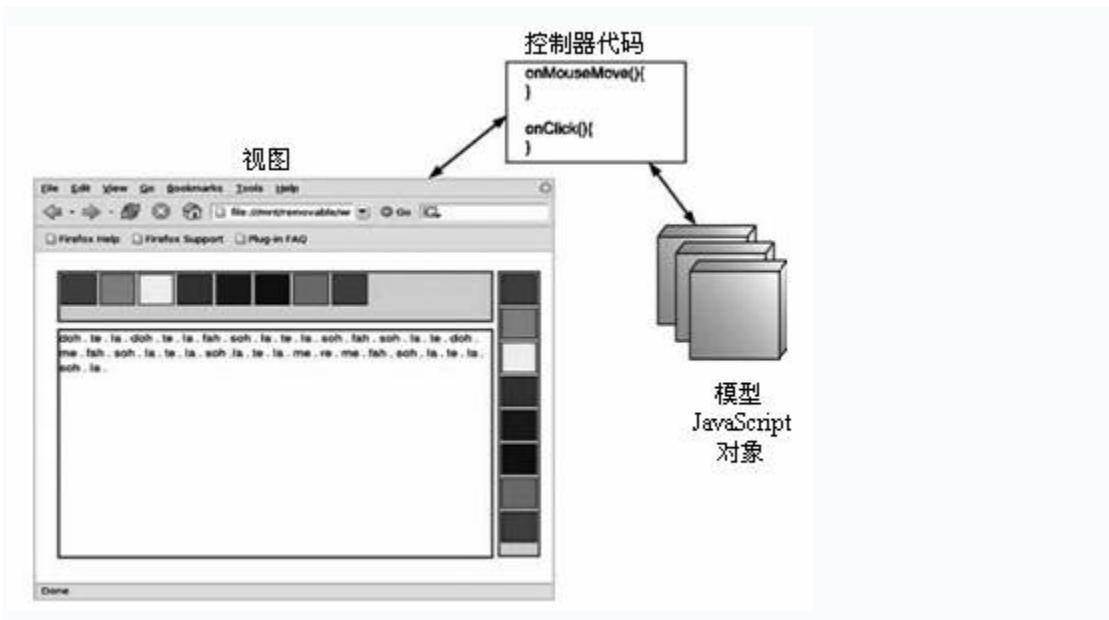


图 4-2 模型—视图—控制器整体应用于 Ajax 客户端应用。这个级别的控制器是将 UI 连接到 JavaScript 业务对象的代码

如果思考一下第 3 章讨论过的传统 Web MVC，你会知道在一个典型的 Ajax 应用中至少有 3 层，每一层在应用的生命周期中扮演不同的角色，它们都有助于开发出清晰、组织良好的代码。图 4-3 演示了这些不同规模的 MVC 模式如何嵌套在应用的架构中。

那么，当开发代码时这对我们意味着什么呢？在下面几节中，我们以更实际的观点来考察使用 MVC 定义 JavaScript 应用的结构，它将如何影响编写代码的方式，它的好处是什么？让我们开始考察一下视图。

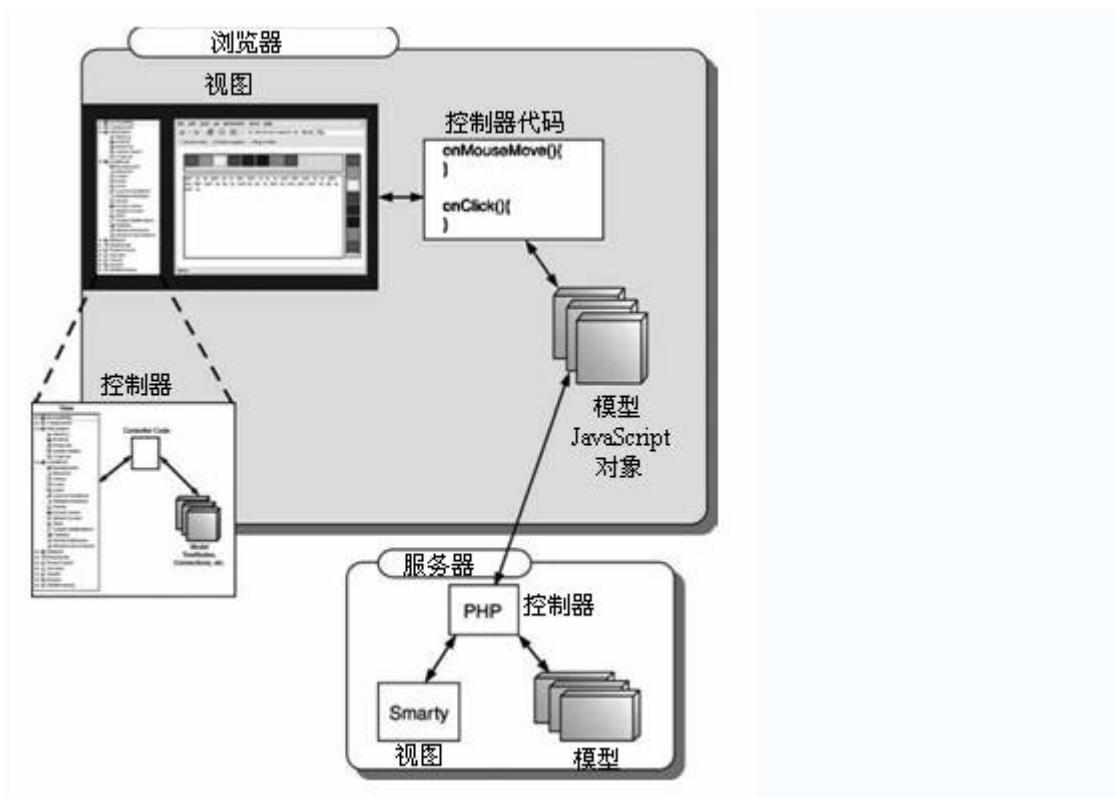


图 4-3 在嵌套的 MVC 架构中，模式以不同的规模重复自己。在最外层的级别，我们可以看到模式作为整体定义了应用的工作流，模型位于 Web 服务器端。在较小的规模，模式在客户端应用中重复；在更小的规模，模式在客户端应用的单个 UI 组件内部重复

4.2 Ajax 应用中的视图

从应用启动时交付在浏览器端的 JavaScript 应用的立场来看，视图是可视的页面，由 DOM 元素组成。这些 DOM 元素通过使用 HTML 标记呈现，或者采用编程方式处理。我们在第 2 章已经显示了如何采用编程方式处理 DOM。

遵照 MVC，视图有两个主要的责任：它必须为用户提供一个可视的界面，以便触发事件，事件用来与控制器对话；它也需要在模型改变时做出响应，更新自己，通常需要再次通过控制器进行通信。

如果应用由一个团队开发，视图可能会成为最有争议的领域。程序员、页面设计师和图形艺术家都会参与进来，特别是当我们探索 Ajax 界面中交互性作用域的时候。让设计师来写代码，或者让程序员介入应用的美学，通常都是坏主意。即使当你承担了双重角色，也应该将它们分离，以便在一段时间内集中处理一个方面。

在服务器 MVC 概览中，我们展示了代码和表现如何混淆在一起，并使用一个模版系统分离了它们。在浏览器端我们有什么选项呢？

第 3 章示范了如何将 Web 页面结构化，以便将 CSS、HTML 和 JavaScript 定义在分离的文件中。在页面部分，这种分离遵从 MVC：样式表是视图，HTML/DOM 是模型（一个 DOM）。尽管从现在的观点来看，页面的呈现是一个黑盒子，HTML 和 CSS 应该一起被看作是视图，但分离它们仍然是一个好主意。通过简单地将 JavaScript 分离出来并放在一个分离的文件中，我们可以使页面设计师和程序员相互隔离，不互相影响。你马上会看到，这仅仅是一个好的开始。

4.2.1 将逻辑从视图中分离

将所有的 JavaScript 编写在一个分离的文件中，是强化视图分离的良好开端。但是即使这样做，如果不注意，仍然可能使视图混入逻辑角色（即模型和控制器）。如果将 JavaScript 事件处理函数内嵌在页面中，例如：

```
<div class='importButton'  
onclick='importData("datafeed3.xml", mytextbox.value);'/>
```

那样就是将业务逻辑硬编码在视图中。什么是 datafeed3? mytextbox 的值和它有什么关系？为什么 importData() 有两个参数，它们的意思是什么？页面设计师不需要知道这些事情。

importData() 是一个业务逻辑函数。按照 MVC 的法则，视图和模型不应该直接通信，一种解决方案是使用额外的层来分离它们。如果将 DIV 标签重写为：

```
<div class='importButton' onclick='importFeedData()'>
```

并且将事件处理函数定义为：

```
function importFeedData(event){  
    importData("datafeed3.xml", mytextbox.value);  
}
```

那么参数就被封装在了 importFeedData() 函数中，而不是一个匿名的事件处理函数中。这允许我们在其他地方重用这个功能，同时分离关注点，保持代码的 DRY（冒着重复我自己的风险，DRY 的意思是“不重复你自己”）。

然而控制器仍然被嵌入在 HTML 中，这使得很难在一个大型应用中找到它。

为了保持控制器和视图分离，我们可以采用编程方式添加事件。除了使用嵌入的事件处理函数，我们还可以指定某种类型的记号，它随后会被代码获得。有几种方法来做这个记号。可以给元素附加唯一的 ID，以每个元素为基础指定事件处理函数。将 HTML 改写为：

```
<div class='importButton' id='dataFeedBtn'>
```

下面的代码作为 window.onload 回调的一部分来执行，例如：

```
var dfBtn=document.getElementById('dataFeedBtn');  
dfBtn.onclick=importFeedData;
```

如果希望对多个事件处理函数执行相同的操作，我们需要使用某种不唯一的记号，一种简单的方法是定义一个额外的 CSS 类。

1. 使用 CSS 间接添加事件

让我们来看一个简单的例子，在这里将鼠标事件绑定在虚拟的音乐键盘的键上。代码清单 4-1 定义了一个包含原始文档结构的简单页面。

代码清单 4-1 musical.html

```
<!DOCTYPE html  
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">  
  
<html>  
  <head>  
    <title>Keyboard</title>  
    <link rel='stylesheet' type='text/css' href='musical.css'/>  
    <script type='text/javascript' src='musical.js'></script>  
    <script type='text/javascript'>  
      window.onload=assignKeys;  
    </script>  
  </head>
```

```
<body>

    <div id='keyboard' class='musicalKeys'>
        <div class='do musicalButton'></div>
        <div class='re musicalButton'></div>
        <div class='mi musicalButton'></div>
        <div class='fa musicalButton'></div>
        <div class='so musicalButton'></div>
        <div class='la musicalButton'></div>
        <div class='ti musicalButton'></div>
        <div class='do musicalButton'></div>
    </div>
    <div id='console' class='console'>
    </div>
</body>
</html>
```

① “键盘”上的键

我们声明了页面以符合严格定义的 XHTML，仅仅为了显示这是可以做到的。我们为表示键盘的 keyboard 元素分配了唯一的 ID，但是没有为表示键的元素分配 ID。注意，每一个指定的键①都有两个样式。musicalButton 对于所有的键是通用的，另外一个单独的样式通过音符来区别它们。这些样式在样式表中单独定义（代码清单 4-2）。

代码清单 4-2 musical.css

```
.body{
    background-color: white;
}
.musicalKeys{
    background-color: #ffe0d0;
    border: solid maroon 2px;
    width: 536px;
    height: 68px;
    top: 24px;
    left: 24px;
```

```
    margin: 4px;
    position: absolute;
    overflow: auto;
}
.musicalButton{
    border: solid navy 1px;
    width: 60px;
    height: 60px;
    position: relative;
    margin: 2px;
    float: left;
}
.do{ background-color: red; }
.re{ background-color: orange; }
.mi{ background-color: yellow; }
.fa{ background-color: green; }
.so{ background-color: blue; }
.la{ background-color: indigo; }
.ti{ background-color: violet; }
div.console{
    font-family: arial, helvetica;
    font-size: 16px;
    color: navy;
    background-color: white;
    border: solid navy 2px;
    width: 536px;
    height: 320px;
    top: 106px;
    left: 24px;
    margin: 4px;
    position: absolute;
    overflow: auto;
}
```

样式 musicalButton 定义了每个键的通用属性，特定于音符的样式仅仅定义了每个键的颜色。注意，尽管顶级文档元素使用显式的像素精度来定位，但是通过 float 样式属性应用浏览器内建的布局引擎，可以将各个键分布在一条水平线上。

2. 绑定事件处理函数代码

JavaScript 文件（代码清单 4-3）采用编程方式将事件绑定到键上。

代码清单 4-3 musical.js

```

function assignKeys(){
    var keyboard=document.getElementById("keyboard");    ← 找到父 DIV
    var keys=keyboard.getElementsByTagName("div");        ← 枚举子节点
    if (keys){
        for(var i=0;i<keys.length;i++){
            var key=keys[i];
            var classes=(key.className).split(" ");
            if (classes && classes.length>=2
                && classes[1]=="musicalButton"){
                var note=classes[0];
                key.note=note;           ← 添加定制属性
                key.onmouseover=playNote;
            }
        }
    }
}

function playNote(event){
    var note=this.note;      ← 获得定制属性
    var console=document.getElementById("console");
    if (note && console){
        console.innerHTML+=note+" , ";
    }
}

```

window.onload 调用了 assignKeys() 函数（可以在这个文件中直接定义 window.onload，但是这限制了它的移植性）。通过唯一的 ID 来发现 keyboard 元素，然后使用 getElementsByTagName() 遍历访问其内部所有的 DIV 元素。这需要知道一些关于页面结构的知识，但是它允许页面设计师自由地在页面中将键盘 DIV 以希望的方式任意移动。

表示键的 DOM 元素返回一个单独的字符串作为 className 属性。我们使用内建的 String.split 函数将其转换为一个数组，并且检查元素是否是 musicalButton 类。之后读取样式的另一部分——它代表了键所演奏的音符——并且作为一个额外的属性附加在 DOM 节点上，这个属性可以被事件处理函数获得。

通过 Web 浏览器演奏音乐需要相当高的技巧，在这里，我们仅仅对键盘下的控制台进行了编程，用 innerHTML 已经足够了。图 4-4 显示了活动中的音乐键盘。这里我们实现了很好的角色分离，假设页面设计师去掉了页面上某个地方的键盘和控制台的 DIV 标签，只要页面包括了样式表和 JavaScript，应用程序仍然可以工作，偶然打破事件逻辑的风险是很小的。HTML 页面有效地成为了一个模版，我们向其中注入了变量和逻辑，这提供了一个保持逻辑与视图相分离的好方法。我们已经手工完成了这个例子，以此来示范工作机制的细节。在生产环境中，你可能喜欢使用几个解决了同样问题的第三方库。



图 4-4 运行于浏览器中的音乐键盘应用。顶部的彩色区域被映射到音符上，当鼠标在上面移动时，音符打印在下面的控制台区域

Rico 框架 (www.openrico.org) 有一个 Behavior 对象的概念，它以 DOM 树的特定部分为目标，为它们添加交互性。3.5.2 节曾简单地考察了 Rico Accordion 的行为。

类似的分离 HTML 标记和交互性的方法可以通过 Ben Nolan 的 Behaviour 库来实现（参见本章“资源”一节）。这个库允许基于 CSS 选择器规则将事件处理函数代码分配给 DOM 元素（见第 2 章）。在之前的例子中，assignKeys() 函数以 keyboard 作为 id 采用编程方式选择文档元素，然后使用 DOM 处理方法得到它直接包含的所有 DIV 元素。我们可以使用一个 CSS 选择器来表达：

```
#keyboard div
```

使用 CSS 选择器可以给所有的 keyboard 元素设置样式。使用 Behaviour.js 库，也可以用相同的方法应用事件处理函数，如下：

```
var myrules={  
  '#keyboard div' : function(key){  
    var classes=(key.className).split(" ");  
    if (classes && classes.length>=2  
    && classes[1]=='musicalButton'){  
      var note=classes[0];  
      key.note=note;  
      key.onmouseover=playNote;  
    }  
  }  
};  
Behaviour.register(myrules);
```

大部分逻辑与前面的例子是一样的，但是对 CSS 选择器的使用提供了一种采用编程方式定位 DOM 元素的简明的替代方法，特别是当需要立即添加几个行为的时候。

这种方法保持了逻辑与视图的分离，但是它也可能将视图和逻辑混在一起，下面我们将看到这一点。

4.2.2 保持视图与逻辑的分离

目前我们做到了页面设计师可以开发页面的外观，而不需要触动代码。然而，正如现在显示的，应用的一些功能（即键的顺序）仍然嵌入在 HTML 中。每一个键定义为一个独立的 DIV 标签，页面设计师可能会无意中删除一些内容。

如果键的顺序是业务领域功能，而不是页面设计问题——这一点也许会有争议——那么应该可以采用编程方式为组件生成一些 DOM，而不是在 HTML 中声明它。更进一步，我们可能想要在一个页面上有多个相同类型的组件。例如，如果不希望页面设计师修改键盘上键的顺序，我们可以简单地规定，为一个 DIV 标签分配 keyboard 类并且在初始化代码中找到它，然后采用编程方式添加键。代码清单 4-4 显示了为达到这个目的而修改过的 JavaScript。

代码清单 4-4 musical_dyn_keys.js

```
var notes=new Array("do","re","mi","fa","so","la","ti","do");
function assignKeys(){
    var candidates=document.getElementsByTagName("div");
    if (candidates){
        for(var i=0;i<candidates.length;i++){
            var candidate=candidates[i];
            if (candidate.className.indexOf('musicalKeys')>=0){
                makeKeyboard(candidate);
            }
        }
    }
}
function makeKeyboard(el){
    for(var i=0;i<notes.length;i++){
        var key=document.createElement("div");
        key.className=notes[i]+" musicalButton";
        key.note=notes[i];
        key.onmouseover=playNote;
        el.appendChild(key);
    }
}
function playNote(event){
    var note=this.note;
    var console=document.getElementById('console');
    if (note && console){
        console.innerHTML+=note+" . ";
    }
}
```

之前，我们在 HTML 中定义了键的顺序，现在将它定义为一个全局 JavaScript 数组。assignKeys() 方法检查文档中所有的顶级 DIV 标签，查看 className 是否包含了值 musicalKeys。如果包含了，尝试使用 makeKeyboard() 函数将 DIV 组装在一个工作键盘上。makeKeyboard() 简单地创建新的 DOM 节点，然后使

用与代码清单 4-4 相同的方式对它所遇到的已声明 DOM 节点进行处理。playNote() 回调处理函数的操作和以前完全一样。

因为我们将空的 DIV 与键盘控件组装在一起，所以添加另外一套键是很简单的，如代码清单 4-5 演示的那样。

代码清单 4-5 musical_dyn_keys.html

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html>
<title>Two Keyboards</title>
<head>
<link rel='stylesheet' type='text/css'
      href='musical_dyn_keys.css'/>
<script type='text/javascript'
      src='musical_dyn_keys.js'>
</script>
<script type='text/javascript'>
window.onload=assignKeys;
</script>
</head>
<body>
<div id='keyboard-top' class='toplonly musicalKeys'></div>
<div id='keyboard-side' class='sidebar musicalKeys'></div>
<div id='console' class='console'>
</div>
</body>
</html>
```

添加第二个键盘只需要一行操作。因为我们不希望它们一个摞在另一个的上面，我们将位置的样式从 musicalKeys 样式类中移到一个独立的类中。样式表的修改显示在代码清单 4-6 中。

代码清单 4-6 musical_dyn_keys.css 的修改

```
.musicalKeys{  ← 通用的键盘样式
    background-color: #ffe0d0;
    border: solid maroon 2px;
    position: absolute;
    overflow: auto;
    margin: 4px;
}
.toplong{  ← 键盘 1 的几何形状
    width: 536px;
    height: 68px;
    top: 24px;
    left: 24px;
}
.sidebar{  ← 键盘 2 的几何形状
    width: 48px;
    height: 400px;
    top: 24px;
    left: 570px;
}
```

musicalKeys 类定义了对于所有键盘通用的视觉样式。toplong 和 sidebar 简单地定义了每一个键盘的几何形状。

通过以这种方式重构键盘的例子，使得轻松地重用代码成为可能。然而，键盘设计部分定义在 JavaScript 中，即代码清单 4-4 中的 makeKeyboard() 函数内。正如图 4-5 所示，一个键盘是垂直布局，另外一个是水平布局。我们如何达到这种效果的呢？

makeKeyboard() 可以采用编程方式定位和放置每个按钮，轻松地计算 DIV 的尺寸。但是在这种情况下，我们必须要为一些琐事而烦心，例如需要确定 DIV 是垂直的还是水平的，并且需要编写自己的布局代码。对于一名熟悉 LayoutManager 对象内部机制的 Java GUI 程序员，这似乎是明显应该采取的方法。如果采取了这个方法，程序员将会就 UI 组件的外观与页面设计师发生分歧，继而产生麻烦。

就像它所显示的，makeKeyboard() 仅仅修改了文档的结构。键由浏览器自己的布局引擎来布局，通过样式表来控制——在这里是使用 float 样式属性。布局由页面设计师来控制是很重要的。逻辑和视图保持分离，给我们带来了和平共处。

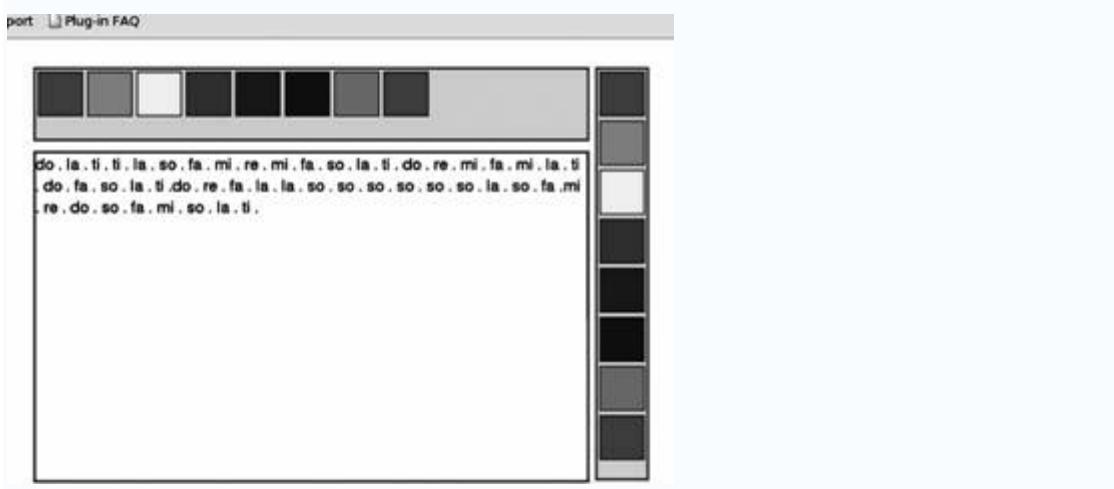


图 4-5 已修改的音乐键盘程序允许页面设计师指定多个键盘。使用基于 CSS 的样式设置和本地呈现引擎，

可以同时提供垂直布局和水平布局，而不需要在 JavaScript 中编写外部的布局代码

键盘是一个相当简单的 UI 组件。在一个更大的、更复杂的 UI 组件（例如树形表格）中，搞清楚如何强迫浏览器自己的呈现引擎做这个布局就更加困难了，并且在某些场合，以编程方式设置样式是不可避免的。然而，为了保持视图和逻辑分离，提出如何避免用 JavaScript 代码做布局，而用浏览器的呈现引擎做布局的问题总是值得的。浏览器的呈现引擎是一段高效、快速和经过良好测试的本地代码，有可能击败我们自己开发的任何 JavaScript 算法。

这些就是目前关于视图的内容。在下一节中，我们将探索 MVC 中控制器的角色，以及它如何在 Ajax 应用中与 JavaScript 事件处理函数相关联。

4.3 Ajax 应用中的控制器

MVC 中控制器的角色是作为模型和视图之间的仲裁者将两者解耦。在一个 GUI 应用（例如 Ajax 客户端应用）中，控制器层由事件处理函数组成。对于 Web 浏览器，技术随着时间而提高，现代浏览器支持两种不同的事件模型。传统的模型相当简单，并且正在被新的 W3C 事件处理规约所取代。然而，在写本书的时候，新的事件处理模型的实现在不同浏览器中是有差别的，并且会引起问题。这两种事件模型都将在讨论。

4.3.1 传统的 JavaScript 事件处理函

Web 浏览器中的 JavaScript 实现允许我们定义响应用户事件（通常是鼠标或者键盘事件）所执行的代码。在支持 Ajax 的现代浏览器中，这些事件处理函数可以被设置到大多数可视元素之上。我们可以使用事件处理函数将可视用户界面（即视图）与业务对象模型相连接。

传统的事件模型在 JavaScript 诞生的早期就存在了，它是相当简单和直接的。DOM 元素有几个预先定义的属性，可以赋值为回调函数。例如，为了附加一个在鼠标点击元素 myDomElement 时被调用的函数，我们可以这样写：

```
myDomElement.onclick=showAnimatedMonkey
```

myDomElement 是可以通过程序处理的任何 DOM 元素。showAnimatedMonkey 是一个函数，定义为：

```
function showAnimatedMonkey(){  
    //some skillfully executed code to display  
    //an engaging cartoon character here  
}
```

这只是一个普通的 JavaScript 函数。注意，当我们分配事件处理函数的时候，传递的是一个 Function 对象，而不是对那个对象的调用，因此它在函数名后面不包含圆括号。下面是一个常见的错误：

```
myDomElement.onclick=showAnimatedMonkey();
```

这对于不习惯将函数看作正常对象的程序员来说更加自然一些，但是它不会按照我们所设想的方式工作。函数将在进行赋值时被调用，而不是当点击DOM元素时才被调用。onclick 属性将被设置为函数返回的任何值。除非你编写一些非常巧妙的包含函数（involving function），可以返回对其他函数的引用，否则产生的效果可能不是你所希望的。正确的方法是：

```
myDomElement.onclick=showAnimatedMonkey;
```

这里向 DOM 元素传递了一个回调函数的引用，告诉它这是当点击节点时需要调用的函数。DOM 元素有很多此类的属性，事件处理函数可以附加在这些属性上。用于 GUI 的常用事件处理回调函数列举在表 4-1 中。类似的属性也可以在 Web 浏览器 JavaScript 的其他地方见到。我们已经遇到的 XMLHttpRequest.onreadystatechange 和 window.onload 也是由程序员赋值的事件处理函数。

表 4-1 DOM 中的常用 GUI 事件处理函数属性

属性	描述
onmouseover	当鼠标首次进入元素区域的时候触发
onmouseout	当鼠标离开元素区域的时候触发
onmousemove	任何时候, 当鼠标在元素区域中移动的时候触发(即频繁地触发)
onclick	当鼠标在元素区域内被点击的时候触发
onkeypress	当按下键, 且该元素有输入焦点时触发。全局键处理器可以附加在文档主体上
onfocus	可视元素获得了输入焦点
onblur	可视元素丧失了输入焦点

事件处理函数有一个不寻常的特征需要在这里提一下, 当编写面向对象的 JavaScript 时, 它最容易让人出错, 这也是在开发 Ajax 客户端时要严重依赖的特征。

我们已经得到了一个 DOM 元素的句柄, 分配了一个回调函数给 onclick 属性。当 DOM 元素收到鼠标点击事件时, 回调即被调用。然而, 函数上下文(即变量 this 所确定的值——参见附录 B, 可获得关于 JavaScript Function 对象的完整讨论)赋值为收到事件的 DOM 节点。根据函数最初是在什么地方声明以及如何声明的, 情况会有所不同, 这可能会把人搞糊涂。

让我们通过一个例子来研究这个问题。我们定义了一个表示按钮对象的类, 它有一个到 DOM 节点的引用、一个回调处理函数, 以及当点击按钮时显示出的一个值。当鼠标点击事件发生时, 按钮的任何实例都将以同样的方式响应, 因此我们定义回调处理函数作为按钮类的一个方法。这些说明对于初学者已经足够了, 下面让我们看看代码。这里是按钮类的构造函数。

```
function Button(value,domEl){
    this.domEl=domEl;
    this.value=value;
    this.domEl.onclick=this.clickHandler;
}
```

继续定义一个事件处理函数作为 Button 类的一部分。

```
Button.prototype.clickHandler=function(){
    alert(this.value);
}
```

这段代码看起来很直观，但是它并没有做我们希望它做的事情。警告框通常会返回消息 undefined，而不是传递到构造函数的 value 属性。为什么呢？当点击 DOM 元素时，函数 clickHandler 由浏览器调用，它设置函数上下文到 DOM 元素，而不是 Button 的 JavaScript 对象。于是，this.value 指向 DOM 元素的 value 属性，而不是 Button 对象的 value 属性。你永远也不可能通过查看事件处理函数的声明来发现这种情况，是不是？

我们可以通过向 DOM 元素传递 Button 对象的引用来解决这个问题，也就是，按下面的方法修改构造函数：

```
function Button(value,domEl){  
    this.domEl=domEl;  
    this.value=value;  
    this.domEl.buttonObj=this;  
    this.domEl.onclick=this.clickHandler;  
}  
DOM 元素仍然没有 value 属性，但是它有一个到 Button 对象的引用，可以从那里得到 value。通过对事件处理函数做如下修改，我们的工作就完成了：
```

```
Button.prototype.clickHandler=function(){  
    var buttonObj=this.buttonObj;  
    var value=(buttonObj && buttonObj.value) ?  
        buttonObj.value : "unknown value";  
    alert(value);  
}
```

DOM 节点引用 Button 对象，Button 对象引用它的 value 属性，这样事件处理函数就做了我们希望它做的事情。我们可以直接给 DOM 节点附加 value，不过附加一个指向整个后端对象的引用可以使这种模式容易地用于任意复杂的对象，顺便说一句，我们在这里已经实现了一个小的 MVC 模式，其中 DOM 元素作为后端对象模型的前端视图。

以上讨论的就是传统的事件模型。这种事件模型主要的缺点是每个元素只允许有一个事件处理函数。在第 3 章介绍的 Observer 模式中，我们注意到一个可观察的元素在给定时间可以有任意多个观察者附加在

上面。当为 Web 页面编写简单的脚本时，这可能不会成为一个严重的缺点，但是当迈向更加复杂的 Ajax 客户端的时候，我们开始感觉到了更多的限制。我们将在 4.3.3 节中更为密切地考察这个问题，现在来看看新近出现的事件模型。

4.3.2 W3C 事件模型

W3C 建议的更加灵活的事件模型要复杂一些，可以在一个 DOM 元素上附加任意数目的监听者。更进一步，如果行为发生在几个元素产生了重叠的文档区域，则每一个元素的事件处理函数都有机会调用并否决事件栈中更深的调用，这称为“吞没了”事件。规约建议事件栈总共遍历两次，第一次从最外面向最里面进行（从文档元素向下），第二次从最里面向最外面进行。实际上，不同的浏览器实现的是这种行为的不同子集。

在基于 Mozilla 的浏览器和 Safari 中，使用 `addEventListener()` 来附加事件回调，使用相应的 `removeEventListener()` 来删除。IE 提供了类似的函数：`attachEvent()` 和 `detachEvent()`。Mike Foster 的 `xEvent` 对象（`x` 库的一部分——参见本章“资源”一节）大胆地尝试在这些实现之上使用 Façade 模式（参见第 3 章）来提供丰富的跨浏览器事件模型。

这里有一个更深层的跨浏览器问题，用户定义的回调处理函数的调用方式有轻微的差异。在基于 Mozilla 的浏览器中，函数被调用，收到事件的 DOM 元素作为上下文对象，就像传统的事件模型一样。在 IE 中，函数上下文总是 `Window` 对象，因此不可能知道当前是哪个 DOM 元素调用了事件处理函数！甚至当有一个类似于 `xEvent` 层的时候，开发者在编写回调处理函数时还需要解决这些差异。

这里提到的最后问题是，没有任何一个实现提供了令人满意的方法来返回所有当前附加的监听器的列表。

因此，我建议不要使用新的事件模型。传统模型的主要缺点——缺少多个监听器的支持——可以通过使用设计模式来解决，我们在下面将要看到。

4.3.3 在 JavaScript 中实现灵活的事件模型...

由于新 W3C 事件模型之间的不兼容性，创造灵活的事件监听器框架的目标仍然没有达到。我们在第 3

章中描述了 Observer 模式，看起来很适合这个目标，它允许我们以一种灵活的方式向事件源添加和删除观察者。很清楚，W3C 感受到了同样的事情，因为修改过的事件模型使用了 Observer 模式，但是浏览器厂家所交付的却是不一致的且错误的实现。传统的事件模型比 Observer 模式要差很多，但是或许可以使用一些自己编写的代码来增强它。

1. 管理多个事件回调

在实现我们自己的解决方案之前，我们先通过一个简单的例子来了解问题所在。代码清单 4-7 显示了一个简单的Web页面，其中一个大的DIV区域以两种方式响应鼠标的移动事件。

代码清单 4-7 mousemat.html

```
<html>
<head>
<link rel='stylesheet' type='text/css' href='mousemat.css' />
<script type='text/javascript'>
var cursor=null;
window.onload=function(){
    var mat=document.getElementById('mousemat');
    mat.onmousemove=mouseObserver;
    cursor=document.getElementById('cursor');
}
function mouseObserver(event){
    var e=event || window.event;
    writeStatus(e);

    drawThumbnail(e);
}
function writeStatus(e){
    window.status=e.clientX+","+e.clientY;
}
function drawThumbnail(e){
    cursor.style.left=((e.clientX/5)-2)+"px";
    cursor.style.top=((e.clientY/5)-2)+"px";
}
</script>
</head>
<body>
<div class='mousemat' id='mousemat'></div>
<div class='thumbnail' id='thumbnail'>
    <div class='cursor' id='cursor'></div>
</div>
</body>
</html>
```

首先，它在writeStatus()函数中更新了浏览器的状态条，然后在drawThumbnail()函数中通过在旁边小的缩略视图区域中重新定位一个点，更新自己在这个区域的映像，以此来复制鼠标光标位置的移动。图 4-6 显示了活动中的页面。

这两个行为是彼此独立的，我们希望能够将这些行为和鼠标移动的其他响应进行交换，即使是在程序

运行时。

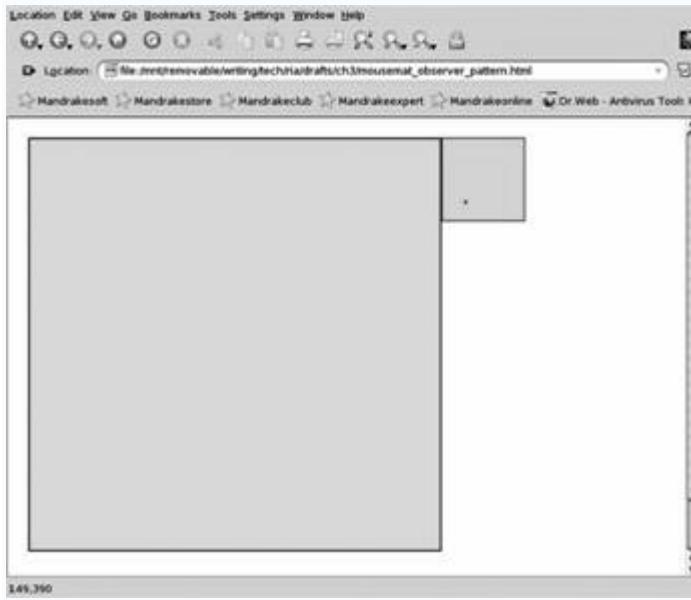


图 4-6 mousemat 程序在“虚拟 mousemat”主区域以两种方式追踪鼠标移动事件：以鼠标的坐标更新浏览器下方的状态条；在缩略视图上随着鼠标光标同步移动的点

mouseObserver()函数是事件监听器（顺便说一下，第一行执行了一点简单的跨浏览器魔法。与Mozilla、Opera或者Safari不同，IE不向回调处理函数传递任何参数，而是将Event对象保存在window.event中）。在这个例子中，我们在事件处理函数中依次调用writeStatus()和drawThumbnail()，将两种活动硬连接在一起。程序准确地完成了我们希望它做的事情，并且因为这只是一个小程序，mouseObserver()的代码还算清晰。在理想情况下，我们希望使用一种更加清晰的方式来将事件监听器连接在一起，以便可以扩展到更加复杂或动态的情况。

2. 用 JavaScript 实现观察者

建议的解决方案是定义一个通用的事件路由器对象，它为目标元素附加一个标准函数，作为一个事件回调，并且维护一个监听器函数的列表。这允许我们以下面的方式重写 mousemat 的初始化代码：

```
window.onload=function(){

    var mat=document.getElementById('mousemat');

    ...

    var mouseRouter=new jsEvent.EventRouter(mat,"onmousemove");

    mouseRouter.addListener(writeStatus);
```

```
mouseRouter.addListener(drawThumbnail);  
}  
}
```

我们定义了一个EventRouter对象，传入DOM元素并希望注册为参数的事件类型。然后向路由器对象增加监听器函数，路由器对象也支持removeListener()方法，这里我们不需要该方法。这个对象看起来很直接，但是我们如何实现它呢？

首先，我们为对象编写一个构造函数，这在JavaScript中仅仅是一个函数（附录B包含了JavaScript对象语法的初级教程。如果不明白下面的代码，可以参考该教程）。

```
jsEvent.EventRouter=function(el,eventType){  
    this.lsnrs=new Array();  
    this.el=el;  
    el.eventRouter=this;  
    el[eventType]=jsEvent.EventRouter.callback;  
}
```

我们定义了监听器函数的数组（最初它是空的），保存了一个到DOM元素的引用，并且使用3.5.1节描述的模式给DOM元素添加了一个到这个对象的引用。然后我们分配一个EventRouter类的静态函数，简单地称作callback，作为事件处理函数。记住在JavaScript中，方括号和点记号是等同的，这意味着：

```
el.onmouseover
```

和

```
el['onmouseover']
```

是相同的。为使用方便，我们将属性名称作为参数传递进来。这与Java或者.NET语言的反射是类似的。

然后，让我们看看callback：

```
jsEvent.EventRouter.callback=function(event){  
    var e=event || window.event;  
    var router=this.eventRouter;  
    router.notify(e)  
}
```

因为这是一个回调函数，函数上下文是触发事件的DOM节点，而不是路由器对象。我们使用前面提到的后端对象模式得到已经附加在 DOM 节点上的EventRouter的引用，然后调用路由器的notify()方法，将事件对象作为参数传递进来。

EventRouter 对象的完整代码如代码清单 4-8 所示。

代码清单 4-8 EventRouter.js

```
var jsEvent=new Array();
jsEvent.EventRouter=function(el,eventType){
    this.lsnrs=new Array();
    this.el=el;
    el.eventRouter=this;
    el[eventType]=jsEvent.EventRouter.callback;
}
jsEvent.EventRouter.prototype.addListener=function(lsnr){
    this.lsnrs.append(lsnr,true);
}
jsEvent.EventRouter.prototype.removeListener=function(lsnr){
    this.lsnrs.remove(lsnr);
}
jsEvent.EventRouter.prototype.notify=function(e){
    var lsnrs=this.lsnrs;
    for(var i=0;i<lsnrs.length;i++){
        var lsnr=lsnrs[i];
        lsnr.call(this,e);
    }
}
jsEvent.EventRouter.callback=function(event){
    var e=event || window.event;
    var router=this.eventRouter;
    router.notify(e)
}
```

注意，数组的一些方法不是标准的JavaScript，而是在我们扩展过的数组定义中定义的方法，附录B中讨论了这些方法。特别地，addListener()和removeListener()可以通过使用append()和remove()方法容易地实现。监听器函数使用Function.call()方法来调用，它的第一个参数是函数上下文，后续的参数（在这里是事件）传递给被调用的函数。

已修改的 mousemat 例子如代码清单 4-9 所示。

代码清单 4-9 已修改的mousemat.html，使用EventRouter

```

<html>
<head>
<link rel='stylesheet' type='text/css' href='mousemat.css' />
<script type='text/javascript' src='extras-array.js'></script>
<script type='text/javascript' src='eventRouter.js'></script>
<script type='text/javascript'>
var cursor=null;
window.onload=function(){
    var mat=document.getElementById('mousemat');
    cursor=document.getElementById('cursor');
    var mouseRouter=new jsEvent.EventRouter(mat,"onmousemove");
    mouseRouter.addListener(writeStatus);
    mouseRouter.addListener(drawThumbnail);
}
function writeStatus(e){
    window.status=e.clientX+"," +e.clientY
}
function drawThumbnail(e){
    cursor.style.left=((e.clientX/5)-2)+"px";
    cursor.style.top=((e.clientY/5)-2)+"px";
}
</script>
</head>
<body>
<div class='mousemat' id='mousemat'></div>

<div class='thumbnail' id='thumbnail'>
    <div class='cursor' id='cursor'></div>
</div>
</body>
</html>

```

内嵌的JavaScript是非常简单的。所需要做的只是创建EventRouter，传进监听器函数，并且为监听器提供实现。我们将此留给读者作为练习，包括使用选择框来动态添加和删除每个监听器。

在此我们讨论了 Ajax 应用中的控制器层，以及设计模式特别是 Observer 模式可以扮演的角色，即用来保持代码清晰且易于开发。在下一节中，我们将查看 MVC 模式的最后一部分——模型。

4.4 Ajax 应用中的模型

模型负责表示应用的业务领域，即应用涉及的真实世界主题，无论它是一家服装店、一件乐器，还是空间中点的集合。我们已经注意到，按照应用的规模来看DOM并不是模型。模型是使用JavaScript编写的一组代码。像大多数设计模式一样，MVC高度基于面向对象的思想。

JavaScript 并没有设计成一种面向对象语言，尽管用它来进行类似于面向对象的方式编程并不很困难。它确实可以通过 prototype 机制来定义一些与对象的类非常相似的东西，而且一些开发者已经为 JavaScript 实现了继承系统，附录 B 中将更多地讨论这些问题。就使用 JavaScript 实现 MVC 而言，我们已经

将这一模式修改为适应 JavaScript 的编码风格，例如，直接作为事件监听器传递 Function 对象。当定义模型时，使用 JavaScript 对象，并且尽量多地使用这种语言的面向对象开发方法，是很有意义的。在下一节，我们将展示如何做到这一点。

4.4.1 使用 JavaScript 为业务领域建模

当讨论视图的时候，我们非常依赖于 DOM。讨论控制器的时候，我们受到浏览器事件模型的限制。而编写模型的时候，我们几乎纯粹与 JavaScript 打交道，而与特定于浏览器的功能毫无关系。那些饱受浏览器的不兼容性和各种 bug 痛苦的人都会承认，这是一种舒适的状态。

让我们看一个简单的例子。在第 3 章，我们曾以从服务器生成数据的观点讨论了服装店应用。描述服装类型列表的数据，包括唯一 ID、名称、描述，还有价格、色彩、尺寸信息。现在让我们回到这个例子，考察当客户端收到数据时会发生什么。在其生命周期过程中，应用将会收到很多这样的数据流，并且需要将数据保存在内存中。如果你喜欢，可以将其看作是一个缓存——保存在客户端的数据可以很快显示，而不需要在用户请求数据的时候回到服务器去获取。这对于改善用户的工作流是有好处的，就像在第 1 章中讨论的那样。

我们可以定义一个简单的 JavaScript 对象，对应于服务器端定义的 garment 对象。代码清单 4-10 显示了一个典型的例子。

代码清单 4-10 Garment.js

```
var garments=new Array();

function Garment(id,title,description,price){

    this.id=id;

    garments[id]=this;

    this.title=title;

    this.description=description;

    this.price=price;

    this.colors=new Object();

    this.sizes=new Object();
```

```
}

Garment.prototype.addColor(color){

    this.colors.append(color,true);

}

Garment.prototype.addSize(size){

    this.sizes.append(size,true);

}
```

我们首先定义了一个全局数组，用来保存所有的服装（没错，全局变量是危险的。在生产环境中，我们应该使用一个名字空间对象，但是在这里为了清楚起见而省略了）。这是一个关联数组，服装的唯一 ID 作为键，保证我们在同一时间对于每种服装类型只有一个引用。在构造函数中，我们设置所有简单的属性，即那些不是数组的属性。我们将数组定义为空，并且提供简单的添加方法，这些方法使用增强的数组代码（参见附录 B）来避免重复。

没有默认提供获取或设置方法，也不像一个完整的面向对象语言做的那样，支持完全的访问控制——私有、受保护和公有变量及方法。有很多方法可以提供这个特征，这些方法在附录 B 中讨论，但是我自己偏向于保持简单的模型。

当解析 XML 数据流的时候，一开始创建一个空的 Garment 对象，然后逐个字段组装它，是很好的方法。敏锐的读者可能奇怪，为什么我们没有提供一个更简单的构造函数。实际上，JavaScript 函数的参数是可变的，当调用一个函数时，任何缺少值的参数会简单地初始化为 null。所以调用

```
var garment=new Garment(123);
```

将被看作与以下调用等同：

```
var garment=new Garment(123,null,null,null);
```

我们必须要传进 ID，因为在构造函数中要使用它以便在全局 garment 列表中放置新的对象。

4.4.2 与服务器交互

我们可以解析显示在代码清单 4-10 中的类型所对应的 XML 数据，来生成客户端应用中的 Garment 对象。我们已经在第 2 章中看到如何做这件事，还将在第 5 章中看到它的几种变化，所以在这里不会讲述所有的细节。XML 文档中包含了属性和标签内容。我们使用 attribute 属性和 getNamedItem() 函数读取属性的数据，并且使用 firstChild 和 data 属性读取标签的 body 文本，例如：

```
garment.description=descrTag.firstChild.data;
```

来解析一个 XML 片断，例如：

```
<description>Large tweedy hat looking  
like an unappealing strawberry  
</description>
```

注意一旦创建了 garment，只需要调用构造函数，就会将其自动添加到所有 garment 的数组中。从数组中删除 garment 也是相当直接的：

```
function unregisterGarment(id){  
    garments[id]=null;  
}
```

这样从全局注册表中删除了 garment 类型，但是不会连带地破坏任何已经创建的 Garment 实例。尽管如此，我们可以给 Garment 对象添加一个简单的验证测试：

```
Garment.prototype.isValid=function(){  
    return garments[this.id]!=null;  
}
```

我们现在通过每个阶段细粒度的、容易处理的对象，为从数据库到客户端传播数据定义了一条清晰的路径。让我们重新回顾一下这些阶段。首先，从数据库创建了一个服务器端对象模型。在 3.4.2 节，我们看到如何使用对象—关系映射（ORM）工具来做这些事情，这种工具提供了开箱即用的对象模型和数据库之间的双向交互。我们可以将数据读入对象，修改它，然后保存数据。

接下来，使用模板系统来从对象模型生成 XML 数据流。最后，解析这个数据流从而在 JavaScript 层创建对象模型。我们现在必须手工解析，而在不远的将来可能会看到类似于 ORM 的映射库的出现。

当然，在管理应用中，我们可能还希望编辑这些数据，即修改 JavaScript 模型，然后就这些修改与服务器端模型进行通信。这将迫使我们面对存在两个领域模型副本的问题，它们可能失去同步。

在传统的 Web 应用中，所有的智能都位于服务器端，所以无论使用什么语言，模型也位于那里。在 Ajax 应用中，我们希望将智能分布在客户端和服务器端，以便客户端可以在向服务器端发送调用请求之前针对自身做出一些决定。如果客户端仅仅做出非常简单的决定，我们可以以随手编写少量代码的方式来处理，但是不可能充分利用智能客户端的长处，系统仍然会反应迟钝。如果我们授权客户端针对自身做出更加重要的决定，那么它就需要知道一些关于业务领域的事情。从这个角度上看，它确实需要一个领域模型。

我们无法除去服务器端的领域模型，因为一些资源只能在服务器端获得，例如持久层的数据库连接，访问遗留系统等等。客户端领域模型必须与服务器端的领域模型配合工作，它需要承担什么工作呢？第 5 章将更为全面地解释客户/服务器之间的交互，以及如何清晰地与一个分离为两部分的领域模型共同工作。

到目前为止，我们以相互隔绝的方式考察了模型、视图和控制器。本章最后的主题是重新将模型和视图融合在一起。

4.5 从模型生成视图

通过在浏览器端引入 MVC，我们得到了 3 个需要关注的不同子系统。分离关注点可以得到更加清晰的代码，但是也可能得到大量的代码。对于设计模式的常见批评是，它们会将甚至是最简单的工作变成一个棘手的过程（EJB 开发者对此深有体会）。

多层应用的设计经常导致在几个层之间重复信息。我们知道 DRY 代码的重要性，解决这种重复的通常方法是在一处定义必需的信息，然后从这个定义自动生成不同的层。本节采用这种方法，介绍一种可以简化 MVC 实现的技术，将所有 3 层以一种简单的方式融合起来。特别地，我们将目标定在视图层上。

到目前为止，我们已经通过手工编码的方式考察了表现底层模型的视图。这使得我们可以很灵活地确定用户将能看到什么，但是有些时候，我们并不需要这种灵活性，手工为 UI 编码会变成一件冗长和重复的工作。一种替代方法是从底层的模型自动生成用户界面，或者至少自动生成用户界面的一部分。做这件事情有一些先例，例如 Smalltalk 语言环境和 Java/.NET Naked Objects 框架（参见“资源”一节），JavaScript 非常适合完成这类任务。让我们看看 JavaScript 反射在这件事情上可以做些什么，并且开发一个通用的 Object Browser 组件，它可以用来作为任何 JavaScript 对象的视图。

4.5.1 JavaScript 对象的反射

编写代码来处理对象的大多数时候，我们对于对象是什么以及对象可以做什么已经非常了解。然而在有些时候，我们需要以摸索的方式编码，在事先不了解的情况下检查对象。为领域模型对象生成用户界面就是这种情况。理想情况下，我们希望开发一个可重用的解决方案，可以同样地用于任何领域——金融、电子商务、科学可视化等等。本节就介绍这样一个可以在应用中使用的 JavaScript 库——ObjectViewer。为了让你体验一下活动中的 ObjectViewer，图 4-7 中使用 ObjectViewer 显示了一个复杂对象图中的几级。

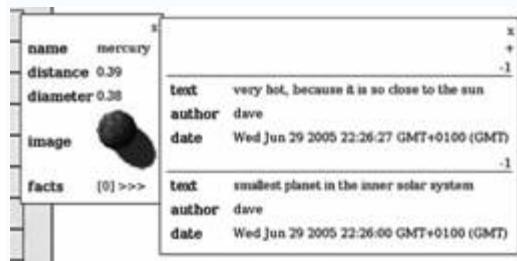


图 4-7 这里 ObjectViewer 用来显示行星系统的等级，每个等级包含一些信息的属性，以及一个保存在数组中的事实列表

所显示的对象代表水星。它是非常复杂的，包含一个图片的 URL，一个事实数组，还有一些简单字符串和数字。ObjectViewer 可以智能地处理所有这些情况，无需事先知道关于对象类型的任何特定事情。

检查对象、检测其属性和能力的过程称作反射（reflection）。熟悉 Java 或.NET 的读者应该熟悉这个术语，附录 B 将更加详细地讨论 JavaScript 的反射能力。这里简短地总结一下，JavaScript 对象可以被遍历，好像它是一个关联数组。为了打印出对象的所有属性，我们可以简单地编写这段代码如下：

```
var description="";
for (var i in MyObj){
    var property=MyObj[i];
    description+=i+" = "+property+"\n";
}
alert(description);
```

以一个警告来表现数据是相当原始的，不能与 UI 的其他部分很好地集成在一起。代码清单 4-11 代表 ObjectViewer 对象的核心代码。

代码清单 4-11 ObjectViewer 对象

```
objviewer.ObjectViewer=function(obj,div,isInline,addNew){
    styling.removeAllChildren(div);
    this.object=obj;
    this.mainDiv=div;
    this.mainDiv.viewer=this;

    this.isInline=isInline;
    this.addNew=addNew;
    var table=document.createElement("table");
    this.tbod=document.createElement("tbody");
    table.appendChild(this.tbod);
    this.fields=new Array();
    this.children=new Array();
    for (var i in this.object){
        this.fields[i]=new objviewer.PropertyViewer(
            this, i
        );
    }
    objviewer.PropertyViewer=function(objectViewer,name) {
        this.objectViewer=objectViewer;
        this.name=name;
        this.value=objectViewer.object[this.name];
        this.rowTr=document.createElement("tr");
        this.rowTr.className='objViewRow';
        this.valTd=document.createElement("td");
        this.valTd.className='objViewValue';
        this.valTd.viewer=this;
        this.rowTr.appendChild(this.valTd);
        var valDiv=this.renderSimple();
        this.valTd.appendChild(valDiv);

        viewer.tbod.appendChild(this.rowTr);
    }
    objviewer.PropertyViewer.prototype.renderSimple=function(){
        var valDiv=document.createElement("div");
        var valTxt=document.createTextNode(this.value);
        valDiv.appendChild(valTxt);
        if (this.spec.editable){
            valDiv.className+=" editable";
            valDiv.viewer=this;
            valDiv.onclick=objviewer.PropertyViewer.editSimpleProperty;
        }
        return valDiv;
    }
}
```

库包含两个对象：ObjectViewer，它对对象的成员进行遍历并装配 HTML 表格来显示数据；PropertyViewer，它作为表格的行呈现单个属性的名称和值。

这样虽然完成了基本的工作，但是遇到了几个问题。首先，它将遍历每一个属性。如果给对象的 prototype 添加帮助函数，我们可以看到它们。如果对 DOM 节点添加帮助函数，将看到所有的内建属性，并了解到 DOM 元素实际上是多么的重要。通常情况下，为选择将对象的哪些属性显示给用户，我们可以在将对象传递给对象呈现器之前，通过在对象上附加一个特殊的属性，即数组，来指定希望显示的对象属性。代码清单 4-12 演示了如何做这件事。

代码清单 4-12 使用 objViewSpec 属性

```
objviewer.ObjectViewer=function(obj,div,isInline,addNew){  
    styling.removeAllChildren(div);  
    this.object=obj;  
    this.spec=objviewer.getSpec(obj);  
    this.mainDiv=div;  
    this.mainDiv.viewer=this;  
    this.isInline=isInline;  
    this.addNew=addNew;  
    var table=document.createElement("table");  
    this.tbod=document.createElement("tbody");  
    table.appendChild(this.tbod);  
    this.fields=new Array();  
    this.children=new Array();  
    for (var i=0;i<this.spec.length;i++){  
        this.fields[i]=new objviewer.PropertyViewer(  
            this,this.spec[i]  
        );  
    }  
    objviewer.getSpec=function (obj){  
        return (obj.objViewSpec) ?  
            obj.objViewSpec :  
            objviewer.autoSpec(obj);  
    }  
    objviewer.autoSpec=function(obj){  
        var members=new Array();  
        for (var propName in obj){  
            var spec={name:propName};  
            members.append(spec);  
        }  
        return members;  
    }  
    objviewer.PropertyViewer=function(objectViewer,memberSpec){  
        this.objectViewer=objectViewer;  
        this.spec=memberSpec;  
        this.name=this.spec.name;  
        ...  
    }  
}
```

我们定义了属性 objViewSpec，ObjectViewer 构造函数将会查看每个对象。如果它不能找到这样的属性，就通过在 autoSpec() 函数中遍历这个对象来创建一个。objViewSpec 属性是一个数字数组，每一个元素是一个属性的查找表。就目前而言，我们只关注生成 name 属性。这个属性的 spec 传进 PropertyViewer 的构造函数，可以从 spec 中获得如何呈现自己的提示。

如果提供一个规范属性给要在 ObjectViewer 中检查的对象，我们就可以只显示那些我们认为相关的属性。

ObjectViewer 的第二个问题是，它不能很好地处理复杂的属性。如果对象、数组和函数附加到 string，则调用 `toString()` 方法。当属性是对象时，通常返回一些不具有描述性的东西，例如 [Object object]。当属性是 Function 对象时，返回函数的整个源代码。我们可以使用 `instanceof` 操作符来区分不同类型的属性。介绍这些之后，让我们看看如何来改善这个查看器。

4.5.2 处理数组和对象

处理数组和对象的一种方式是允许用户对每个属性使用分离的 ObjectViewer 对象访问数组和对象。可以选择几种表现方式，这里我们选择的是将子对象表现为弹出窗口，就像分级菜单一样。

为了达到这个目标，我们需要做两件事情。首先，需要在对象定义中添加 `type` 属性，并且定义所支持的类型：

```
objviewer.TYPE_SIMPLE="simple";
objviewer.TYPE_ARRAY="array";
objviewer.TYPE_FUNCTION="function";
objviewer.TYPE_IMAGE_URL="image url";
objviewer.TYPE_OBJECT="object";
```

我们修改了那些为自身并没有携带类型信息的对象生成 spec 信息的函数，如代码清单 4-13 所示。

代码清单 4-13 已修改的 `autoSpec()` 函数

```
objviewer.autoSpec=function(obj){
  var members=new Array();
  for (var propName in obj){
    var propValue=obj[name];
    var propType=objviewer.autoType(value);

    var spec={name:propName,type:propType};
```

```

        members.append(spec);
    }
    if (obj && obj.length>0){
        for(var i=0;i<obj.length;i++){
            var propName="array ["+i+"]";
            var propValue=obj[i];
            var propType=objviewer.ObjectViewer.autoType(value);
            var spec={name:propName,type:propType};
            members.append(spec);
        }
    }
    return members;
}
objviewer.autoType=function(value){
    var type=objviewer.TYPE_SIMPLE;
    if ((value instanceof Array)){
        type=objviewer.TYPE_ARRAY;
    }else if (value instanceof Function){
        type=objviewer.TYPE_FUNCTION;
    }else if (value instanceof Object){
        type=objviewer.TYPE_OBJECT;
    }
    return type;
}

```

注意，我们也添加了对按数字索引的数组的支持，它的元素不能通过 `for...in` 类型的循环来发现。

我们需要做的第二件事情是修改 `PropertyViewer` 以考虑不同的类型，并且相应地呈现它们，如代码

清单 4-14 所示。

代码清单 4-14 已修改的 `PropertyViewer` 构造函数

```

objviewer.PropertyViewer=function
(objectViewer,memberSpec	appendAtTop){
    this.objectViewer=objectViewer;
    this.spec=memberSpec;
    this.name=this.spec.name;
    this.type=this.spec.type;
    this.value=objectViewer.object[this.name];
    this.rowTr=document.createElement("tr");
    this.rowTr.className='objViewRow';
    var isComplexType=(this.type==objviewer.TYPE_ARRAY
        ||this.type==objviewer.TYPE_OBJECT);
    if ( !(isComplexType && this.objectViewer.isInline)
    ){
        this.nameTd=this.renderSideHeader();
        this.rowTr.appendChild(this.nameTd);
    }
    this.valTd=document.createElement("td");

```

```

        this.valTd.className='objViewValue';
        this.valTd.viewer=this;
        this.rowTr.appendChild(this.valTd);
        if (isComplexType){
            if (this.viewer.inline){
                this.valTd.colSpan=2;
                var nameDiv=this.renderTopHeader();
                this.valTd.appendChild(nameDiv);
                var valDiv=this.renderInlineObject();
                this.valTd.appendChild(valDiv);
            }else{
                var valDiv=this.renderPopoutObject();
                this.valTd.appendChild(valDiv);
            }
        }else if (this.type==objviewer.TYPE_IMAGE_URL){
            var valImg=this.renderImage();
            this.valTd.appendChild(valImg);
        }else if (this.type==objviewer.TYPE_SIMPLE){
            var valTxt=this.renderSimple();
            this.valTd.appendChild(valTxt);
        }
        if (appendAtTop){
            styling.insertAtTop(viewer.tbod,this.rowTr);
        }else{
            viewer.tbod.appendChild(this.rowTr);
        }
    }
}

```

为了适应不同类型的属性，我们定义了几种呈现方法，它们的具体实现太过详细，无法在这里完整地复制。整个 ObjectViewer 的源代码可以从本书的配套网站下载。

现在我们已经有了相当完整的方法来自动查看领域模型。为了使领域模型对象可视化，我们需要做的所有事情就是将 objViewSpec 属性分配给它们的原型。例如，图 4-7 中显示了为视图提供支持的 Planet 对象，在它的构造函数中有以下语句：

```

this.objViewSpec=[

    {name:"name",      type:"simple"},

    {name:"distance", type:"simple", editable:true},

    {name:"diameter", type:"simple", editable:true},

    {name:"image",     type:"image url"},

    {name:"facts",     type:"array", addNew:this.newFact, inline:true }

];

```

这个定义中的标注是 JavaScript 对象标注，称为 JSON。方括号代表数字数组，花括号代表关联数组或者对象（两者其实是相同的）。我们在附录 B 中将更加充分地讨论 JSON。

这里还有几个实体没有解释。addNew、inline 和 editable 是什么意思？它们的目的是通知视图，用户不仅可以查看还可以修改领域模型的这些部分，这就将控制器方面引入了系统。我们将在下一节考察这些内容。

4.5.3 添加控制器

能够查看领域模型当然很好，但是很多日常应用还要求我们修改它们——下载曲调、编辑文档、向购物篮添加条目等等。控制器的责任是在用户交互和领域模型之间进行协调，我们现在将这些功能添加到 ObjectViewer。

当点击文本值时，可以编辑简单的文本值，如果规范对象标记某些属性是可以编辑的。代码清单 4-15 为用来呈现简单文本属性的代码。

代码清单 4-15 renderSimple() 函数

```
objviewer.PropertyViewer.prototype.renderSimple=function(){
    var valDiv=document.createElement("div");
    var valTxt=document
        .createTextNode(this.value);    ← 显示只读值
    valDiv.appendChild(valTxt);
    if (this.spec.editable){    ① 如果可以编辑，添加交互功能
        valDiv.className+=" editable";
        valDiv.viewer=this;
        valDiv.onclick=objviewer.PropertyViewer.editSimpleProperty;
    }
    return valDiv;
}
objviewer.PropertyViewer.editSimpleProperty=function(e){    ② 开始编辑
    var viewer=this.viewer;
    if (viewer){
        viewer.edit();
    }
}
```

```

objviewer.PropertyViewer.prototype.edit=function(){
    if (this.type==objviewer.TYPE_SIMPLE){
        var editor=document.createElement("input");
        editor.value=this.value;
        document.body.appendChild(editor);
        var td=this.valTd;
        xLeft(editor,xLeft(td));
        xTop(editor,xTop(td));
        xWidth(editor,xWidth(td));
        xHeight(editor,xHeight(td));
        td.replaceChild(editor,td.firstChild); ③ 替换读/写视图
        editor.onblur=objviewer;
        PropertyViewer.editBlur; ④ 添加提交回调
        editor.viewer=this;
        editor.focus();
    }
}
objviewer.PropertyViewer
.editBlur=function(e){ ⑤ 结束编辑
    var viewer=this.viewer;
    if (viewer){
        viewer.commitEdit(this.value);
    }
}
objviewer.PropertyViewer.prototype.commitEdit=function(value){
    if (this.type==objviewer.TYPE_SIMPLE){
        this.value=value;
        var valDiv=this.renderSimple();
        var td=this.valTd;
        td.replaceChild(valDiv,td.firstChild);
        this.objectViewer
            .notifyChange(this); ⑥ 通知观察者
    }
}

```

编辑属性包括几个阶段。如果字段是可编辑的**①**，要给显示值的 DOM 元素分配 onclick 处理器，而且还要为可编辑的字段分配一个具体的 CSS 类名，使得它们在鼠标停在其上时改变颜色。毕竟，我们需要让用户意识到这个字段是可以编辑的。

`editSimpleProperty()`**②**是一个简单的事件处理函数，它从点击的 DOM 节点上得到 `PropertyViewer` 的引用，然后调用 `edit()` 方法。这种连接视图和控制器的方法和 4.3.1 节中提到的方法相似。我们检查属性类型是正确的，然后将只读标签替换为同样大小的、包含值的 HTML 表单文本输入**③**。我们还在这个文本区域附加了 `onblur` 处理函数**④**，它使用一个只读标签替换了可编辑区域**⑤**，并且更新了领域模型。

我们可以以这种方式处理领域模型，但是通常当更新模型时，我们常常希望采取一些其他操作。`ObjectViewer` 的 `notifyChange()` 方法**⑥**在 `commitEdit()` 函数中调用，在这里派上了用场。代码清单 4-16 完整地显示了这个函数。

代码清单 4-16 `ObjectViewer.notifyChange()`

```

objviewer.ObjectViewer.prototype
.notifyChange=function(propViewer){
  if (this.onchangeRouter){
    this.onchangeRouter.notify(propViewer);
  }
  if (this.parentObjViewer){
    this.parentObjViewer.notifyChange(propViewer);
  }
}
objviewer.ObjectViewer.prototype
.addChangeListener=function(lsnr){
  if (!this.onchangeRouter){
    this.onchangeRouter=new jsEvent.EventRouter(this,"onchange");
  }
  this.onchangeRouter.addListener(lsnr);
}
objviewer.ObjectViewer.prototype
.removeChangeListener=function(lsnr){
  if (this.onchangeRouter){
    this.onchangeRouter.removeListener(lsnr);
  }
}

```

使用 Observer 模式和 4.3.3 节中定义的 EventRouter 对象理想地解决了我们面临的问题——当领域模型发生修改时的，通知任意的进程。我们可以将 EventRouter 附加到可编辑字段的 onblur 事件上，但是一个复杂的模型可能包含很多可编辑字段，代码不应该看到 ObjectViewer 实现中如此具体的细节。

相反，我们在 ObjectViewer 对象本身定义自己的事件类型，即 onchange 事件，并且给它附加了 EventRouter。因为当细究对象和数组属性时，ObjectViewer 安排在树形结构中，递归地将 onchange 事件传递给父节点。通常这样就可以将监听器附加在根 ObjectViewer 上，这是我们在应用代码中所创建的，再将对象图下面几层模型属性的改动传播回来。

事件处理函数的一个简单例子是向浏览器状态条编写一条信息。行星模型的顶级对象是太阳系，所以可以写成：

```

var topview=new objviewer.ObjectViewer
(planets.solarSystem,mainDiv);

topview.addChangeListener(testListener);

```

这里 testListener 是一个事件处理函数，看起来类似于：

```

function testListener(propviewer){
  window.status=propviewer.name+" ["+propviewer.type+"] =
  "+propviewer.value;
}

```

当然，事实上当领域模型变化时，我们希望实现更多令人激动的事情，例如与服务器联系。下一章将考察联系服务器的方法，并且将 ObjectViewer 的应用推向深入。

4.6 小结

模型—视图—控制器模式是一个架构模式，它广泛应用于传统 Web 应用的服务器端代码。为了给客户端生成数据，我们显示了如何在 Ajax 应用中重用服务器端的这个模式。我们也应用这个模式来设计客户端自身的代码，并且通过使用这个模式来获得了许多深刻认识。

在考察视图子系统时，我们示范了如何有效地从逻辑中分离出表现，这样做带来了非常实用的好处，就是允许页面设计师和程序员的角色相分离。在代码库中保持责任明确，从而能够反映团队的组织结构和技能，可以显著地推进生产力。

在控制器代码中，我们考察了 Ajax 可以使用的不同事件模型，为了谨慎起见宁可选择老的事件模型。尽管它受限于每种事件类型的单个回调函数，但我们可以看到在标准 JavaScript 事件模型之上，如何实现 Observer 模式来开发灵活的、可以重新配置的事件处理函数层。

关于模型，我们提出了分布式多用户应用中更大的问题，第 5 章将就此进行更深入的探索。

关注模型、视图和控制器看起来工作量很大。在关于 ObjectViewer 例子的讨论中，我们考察了用自动方式来简化它们之间交互的方法，创建了能够为用户表现对象模型并允许它们与之交互的简单系统。

我们将在下一章继续利用设计模式来探索客户/服务器之间的交互。

4.7 资源

本章使用的 Behaviours 库可以在 <http://ripcode.cn.nz/behaviour/> 找到。Mike Foster 的 x 库可以在 www.cross-browser.com 找到。

从模型自动生成视图的技术其灵感来自于 Naked Objects 项目 (<http://www.nakedobjects.org>)。Naked Objects (John Wiley & Sons, 2002 年出版) 由 Richard Pawson 和 Robert Matthews 合著，随着代码的发展，现在有些过时了，但是这本书在开篇的几节中对手工编码的 MVC 提出了尖锐的批评。

在 ObjectViewer 中的行星图片由 Jim 的 Cool Icons 提供 (<http://snaught.com/JimsCoolIcons/>)，使用 POVRay 模型来建模，并用 NASA 的真实图片来进行纹理处理（据其网站上的说法）。

第5章 服务器的角色

本章内容

- 结合 Ajax 使用目前的 Web 框架
- 与服务器交换内容、脚本或数据
- 向服务器发送更新
- 将多个请求和响应打包成一个 HTTP 调用

本章继续并完成第 4 章开始的工作——使应用健壮和可扩展。我们从概念验证的阶段推进到一些可以在真实世界中使用的例子。第 4 章还检查了一些构造客户端代码，以达到目标的方式。本章将要考察服务器，特别是客户和服务器之间的通信。我们首先从大局开始考察，讨论服务器所应完成的功能。然后，我们继续描述在服务器端框架中通常使用的架构类型。当前，有很多正在使用的 Web 框架，特别是在 Java 领域中，并没有试图涉及每一个框架，而仅仅是确定通用的方法和从事 Web 应用开发的方式。大多数框架的设计目的是用来生成传统的 Web 应用，我们特别感兴趣的是了解它们如何适应 Ajax，并且存在着哪些挑战。

在考虑过大规模的模式之后，我们将研究客户和服务器间通信更微小的细节。第 2 章介绍了 XMLHttpRequest 对象和隐藏 IFrame 的基础知识。在考察从服务器更新客户端的各种模式，并且讨论使用 DOM 方法解析 XML 文档的替代方式时，我们还会回顾这些基础知识。在最后一节，通过提供一个客户端的请求队列和管理它们的服务器端进程来引入一个系统，在应用的生命周期中管理客户端和服务器端的通信流量。

那么让我们出发吧，先来了解 Ajax 中服务器的角色。

5.1 与服务器配合工作

在 Ajax 应用的生命周期中，服务器要实现两个角色，这两个角色截然不同。首先，它必须将应用交付到浏览器。迄今为止，我们一直假设内容的初始交付过程是一个相当静态的过程，即使用一系列.html、.css 和.js 文件编写 Web 应用，即使是一个非常基础的 Web 服务器也可以交付。这种方法没有什么问题——其实，也可以说问题很多——但是这并不是我们可以采用的唯一选项。在 5.3 节中讨论服务器端框架时，我们会考察其他的替代方法。

服务器的第二个角色是与客户端对话，处理查询并提供请求需要的数据。因为 HTTP 是唯一可用的传输机制，所以我们受限于任何会话都必须由客户首先发起，服务器只能响应。在第 4 章中，我们讨论

了 **Ajax** 应用有必要在客户端（为了更快地响应）和服务器端（为了访问资源，例如数据库）都维护一个领域模型。保持它们彼此之间的同步就成了一个主要的挑战，客户端没有能力独立解决这个问题。我们将在 5.5 节考察一些向服务器写数据的方法，并且基于在第 3 章中遇到的模式之一介绍这个问题的一个解决方案。

正如将在本章中看到的，我们可以以几种不同的方式来交付客户端应用，并且与客户端对话。一种方式比其他方式更好吗？有没有一些相互支持的特定组合？它们能混合使用并相匹配吗？这些不同的解决方案如何与遗留的服务器端框架和架构配合工作呢？为了回答这些问题，一个描述不同选项的列表是很有用的。这正是我们将要在本章中发展的内容。首先让我们考察一下在 Web 应用中建立服务器的方式，以及 Ajax 如何影响它。

5.2 编写服务器端代码

在传统的 Web 应用中，服务器端往往是一个相当复杂的地方，它通过应用控制和监视用户的工作流，并维护对话的状态。Web 应用使用某种特定的语言和一套约定来设计，这决定了它能做什么不能做什么。语言本身可能绑定在特定的架构、操作系统或者硬件之上。挑选一种编程语言需要做出的重大抉择，让我们讨论一下可以采用的选项。

5.2.1 流行的实现语言

服务器端编程被屈指可数的几种语言所统治。在因特网历史的短暂过程中，服务器端语言流行的变化是很显著的。这个领域现在的王者是 PHP、Java 和传统的 ASP, ASP.NET 和 Ruby 也越来越流行。毫无疑问，这些名字对于大多数读者来说已经很熟悉了，所以这里不会对它们详细解释。Ajax 主要是一种客户端技术，能够和这些语言中的任何一个进行互操作。某些使用 Ajax 的方式甚至相当忽视服务器端语言的重要性，使得可以很轻易地将 Ajax 应用从一种服务器平台转向另一种服务器平台。

对于 Ajax, Web 框架在很多场合比实现语言更加重要。Web 框架中有一些基本的假设，关于应用如何构造，以及关键的职责位于何处。大多数的框架都设计用来建造传统的 Web 应用，它们对于应用的生命周期的假设与 Ajax 应用的生命周期有很大的不同，这在某些地方会引起问题。我们将在下面一节考察服务器端的设计和框架，为了给这些讨论铺垫些基础，让我们先回顾一下基于 Web 的架构的基本原则。tier）。层通常代表了应用的一套特定的职责集合，但也可以描述为被特定的机器或进程物理隔离的子系统。这就将层与 MVC 中的角色区分开来。模型、视图和控制器都不是层，因为它们通常处于同一进程中。

5.2.2 N 层体系架构

分布式应用的一个核心概念是层（

早期的分布式系统由客户层和服务器层组成。客户层是一个桌面程序，通过使用网络套接字库与服务器通信。服务器层通常就是一个数据库服务器。

类似地，早期的 Web 系统由一个浏览器与一个 Web 服务器通信组成，Web 服务器是在网络上从（服务器的）文件系统发送文件（给客户端）的一个单片式的系统。

随着基于 Web 的应用变得越来越复杂，并且开始需要访问到数据库，客户/服务器的两层模型应用到了 Web 服务器上就创造出了三层模型，Web 服务器作为 Web 浏览器和数据库之间的媒介。这个模型在

后来的细化过程中，中间层出现了更进一步的分化，分离出了表现和业务两种角色，两者要么作为不同的进程，要么作为在同一进程中设计的、更加模块化的软件。

现代的Web应用通常有两个基本的层。业务层对业务领域建模，并且直接与数据库通信。表现层从业务层获得数据并且呈现给用户。在这样的设置中，浏览器是一个哑客户端¹。

Ajax 的引入可以看作是对客户层更加深入的开发，在Web服务器和客户之间将表现层传统的工作流和会话管理的职责分离开来（图 5-1）。

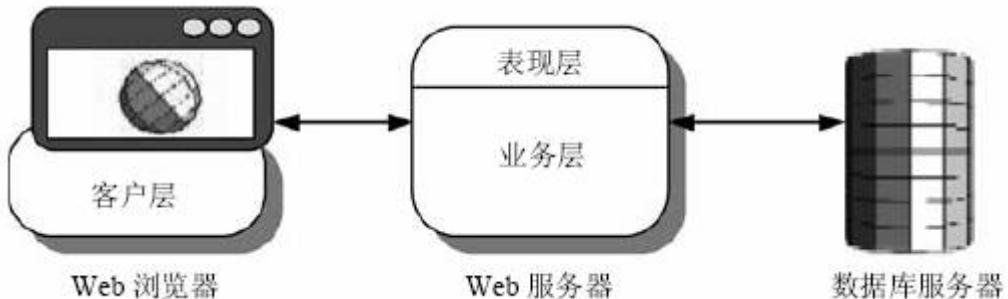


图 5-1 Ajax 应用将表现层的一些职责从服务器转移到了浏览器，在一个称为客户端层的新实体中服务器端表现层的角色可以极大地简化，对工作流的控制部分或者全部地转移到了新的层中，这个层使用 JavaScript 编写，位于浏览器中。

正如我们曾经讨论的，应用中新出现的这个层带来了新的可能性，同时也带来了潜在的更大的复杂性和混乱。显然，我们需要通过某种方式来加以管理。

5.2.3 维护客户端和服务端的领域模型

在 Ajax 应用中，我们仍然需要在服务器端为业务领域建模，因为它接近数据库和其他重大的集中式资源。然而，为了使客户端代码具有足够的响应性和智能，我们通常希望在浏览器至少维护一部分领域模型，这就引入了如何保持两个模型之间的同步这样一个有趣的问题。

增加额外的一层总会带来复杂性和通信上的负担。幸运的是，这个问题并不是一个全新的问题，类似的问题在 J2EE Web 开发中经常会遇到。例如，在对业务层和表现层有严格划分的应用中，存在着类似的问题。领域模型位于业务层中，由表现层来查询，然后表现层向浏览器传送生成的 Web 内容。这个问题在 J2EE 中通过使用“传输对象”来解决，这些对象是简单的 Java 对象，设计用来在相邻的层之间传递数据，并且向表现层展现领域模型的有限视图。

尽管如此，Ajax 也为带来了新的挑战。在 J2EE 中，表现层和业务层是以一种相同的语言编写的，并提供了远程过程调用的机制，这种情形在 Ajax 中通常不存在。例如，我们可以通过 Mozilla 的 Rhino 或者微软的 Jscript.Net 在服务器层使用 JavaScript，但是这种做法在目前相当地异端，并且我们仍然需要在两个 JavaScript 引擎之间进行通信。

在层间进行通信的基本需求是从服务器读数据和向服务器写数据。我们将在 5.3 节到 5.5 节中考察通信的细节。在结束关于架构问题的概览之前，我们将考察一下目前正在使用的服务器端架构的主要类别。我们特别感兴趣的是它们如何向表现层展现领域模型，以及这会给基于 Ajax 的设计带来哪些限制。

在最近一次非正式的调查（参见本章“资源”一节）中，单是 Java（公平地说，比起任何其他的服务器语言，Java 可能更受这些框架所累）就列出了超过 60 种表现层框架。大多数的框架仅仅在细节上有些差别，幸运的是，我们可以遵循以下几种架构模式中的某一种来描述表现层（无论是何种服务器语言）的特征。现在让我们来看一下。

5.3 大局观：通用的服务器端设计

3 大局观：通用的服务器端设计

服务器端的框架与所有的 **Ajax** 应用都有关系，如果选择了从服务器端模型生成客户端代码，那么它就更加重要的了。如果手工编写客户端代码，作为静态的 **HTML** 和 **JavaScript** 页面提交，那么框架就与应用的交付没有太大关系，但是应用所使用的数据仍然需要动态生成。而且，正如我们在前一节注意到的，服务器端框架通常都包含某种类型的领域模型，表现层框架位于模型和 **Ajax** 应用之间。为了能让应用流畅运行，我们需要使用这些框架。

有人不太友好地将 **Web** 应用服务器描述为开发者的竞技场。通过一系列的 **Web** 页面向用户提供连贯的工作流，同时又要与后端的系统（例如数据库服务器）相连接，这一问题从来就没有得到充分的解决。**Web** 上散落着营养不良、维护得很差的框架和工具，每月（如果不是每周）还会冒出很多新的项目。

幸运的是，我们可以从这一团无序的混沌中去芜存菁。它从这些良莠不齐的框架中提炼出精华，大概有四种主要的方法。下面依次讨论这些方法，看看它是如何适应 **Ajax** 模型的。

5.3.1 不使用框架进行简单的Web服务器编码...

最简单的框架就是根本不使用框架。编写一个 **Web** 应用，不使用框架来定义关键的工作流元素或协调对后端系统的访问，并不意味着完全的无序。很多的 **Web** 站点仍然以这样的方式来开发，每个页面产生自己的视图并执行自己的后端操作，可能通过某些帮助函数或对象共享库的协助，图 5-2 展示了这种编程模式。

如果我们假设客户端代码是手工编写的，修改这种方式的代码来支持 **Ajax** 是相当直接的。从服务器生成客户端的代码是个很大的话题，已经超出了本书涉及的范围。为了交付客户端应用，需要定义一个主页，其中包含任何需要的 **JavaScript** 文件、样式表、以及其他资源。为了提供数据，只需要将生成的 **HTML** 页面替换为 **XML** 或者我们选择的其他数据流（后面还会更多地讨论这个话题）。

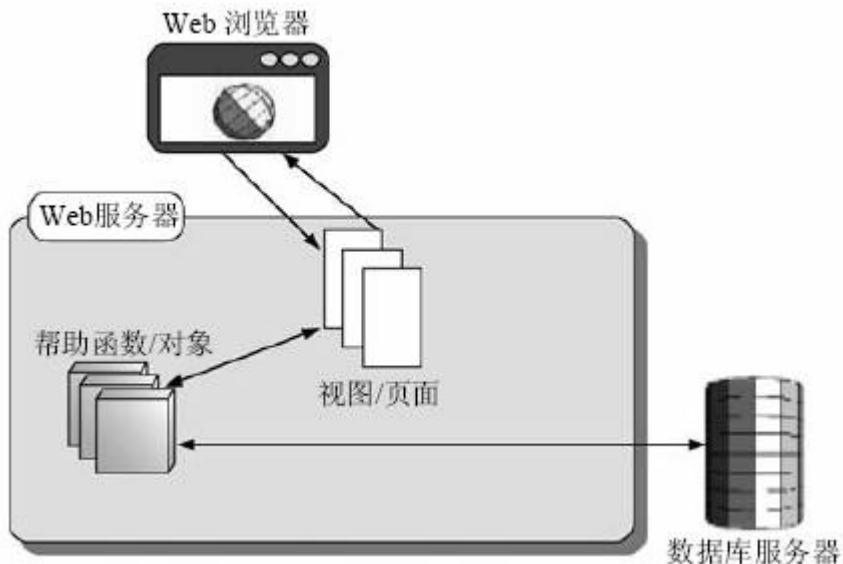


图 5-2 不采用框架的 Web 编程。每个页面、**servlet** 或者 **CGI** 脚本都维护自己的逻辑和表现层细节。帮助函数和（或）对象可以封装通用的低层功能，例如数据库访问

在传统的 Web 应用中，这种方式的主要缺点是文档间的链接在文档中分散得到处都是。也就是说，控制器的角色没有清晰地定义在一处。如果开发者需要在不同的屏幕重新执行相同的操作流程，在很多地方的超链接都需要修改。虽然可以通过把经常使用的链接内容（例如导航条）放进包含文件中，或是使用帮助函数以编程方式生成这些内容，来稍微改善一下这种状况，但是维护的成本仍然会随着应用变得更加复杂而急剧上升。

在 Ajax 应用中，这也许不至于引起什么问题，因为超链接和其他交叉引用通常不会像在 Web 页面中那样大量嵌入在数据中，但是页面之间的包含和前转（**forward**）指令仍然会引起问题。包含和前转在简单的 XML 文档中不是必需的，但我们在 5.5 节中看到，更大的应用可能会发送由多个子进程装配的复杂的结构化文档。早先一代的 Web 框架使用 **MVC** 来作为解决这些问题的良药，很多这样的框架仍然在使用，我们接下来就对它们做一些考察。

5.3.2 使用 Model2 工作流框架

Model2 设计模式是 **MVC** 的一个变种，其中控制器只有一个单独的入口点，并且在用户可能的工作流中只有一个单独的定义。应用到 Web 应用，这意味着一个单独的控制器页面或 **servlet** 负责路由大多数的请求，将请求传递到不同的后端服务，最后输出特定的视图。虽然有大量的 Java 和 PHP 框架遵循这个模式，但是 **Apache Struts** 可能是最有名的 Model2 框架了。图 5-3 展示了 Model2 Web 框架的结构。

那么我们如何将这个设计应用在与 Ajax 客户端对话的服务器应用上呢？**Model2** 与客户端应用的交付几乎没有关系，客户端应用的交付通常作为数据传输发生在启动阶段，对于所有通过了身份验证的用户都是完全相同的。中央控制器自身可以参与身份验证的过程，但是除了使用单个控制器端点，使用其他方式来交付客户端应用几乎没有什么优点。

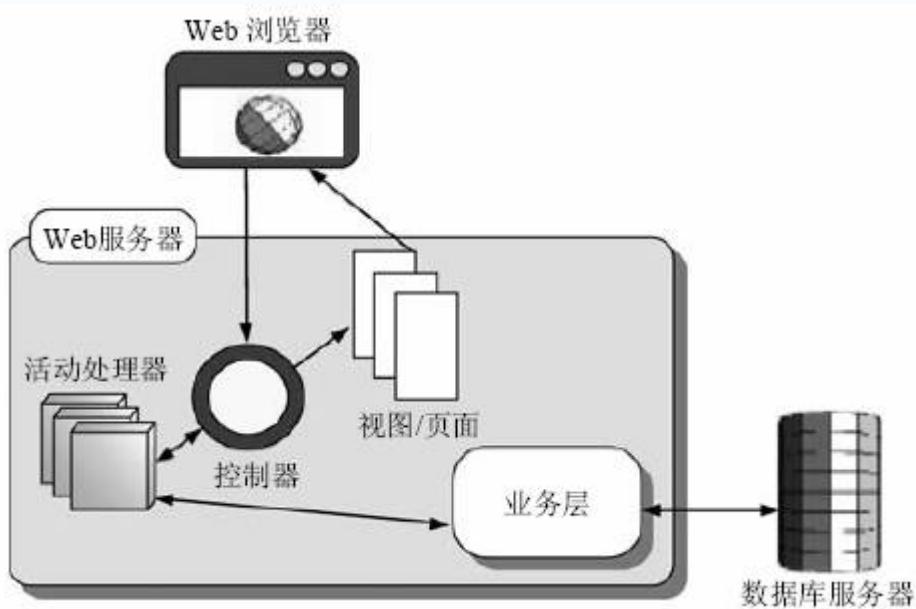


图 5-3 Model2 Web 框架。一个单独的控制器页面或 **servlet** 接收所有的请求，并用全部的用户工作流和交互来进行配置。请求被传递到一组用来做更具体处理的辅助类或函数中的一个，最后在发送输出到浏览器之前，被定向到一个视图组件（例如 **JSP** 或者 **PHP** 页面）

尽管如此，它提供了交付数据的有效解决方案。**Model2** 返回的视图本质上是独立于框架的，我们可以很轻松地将 **HTML** 替换为 **XML** 或者其他数据格式。控制器的一部分职责将会转移到客户端层，但是控制器的一些功能仍然需要通过服务器端映射来有效地表达。

传统 **Web** 应用中的 **Model2** 模式提供了一个好方法，可以在高层次的抽象上表达控制器的很多职责，但是视图的实现需要手工编写代码。**Web** 框架的后期发展也将尝试为视图提供更高层次的抽象，下面让我们来研究一下。

5.3.3 使用基于组件的框架

在为传统的 **Web** 应用编写 **HTML** 页面的时候，页面作者手边只有非常有限的一套预定义 **GUI** 组件，即 **HTML** 表单元素。它们的特征集近 10 年来几乎没有变化，与现代的 **GUI** 工具集相比，它们是非常基础和令人失望的。如果页面作者希望引入树控件或者可编辑的栅格、日历控件或者动态的分级菜单之类的，就需要借助于基础文档元素的低层编程。这跟开发者使用组件工具集（例如 **MFC**、**GTK+**、**Cocoa**、**Swing** 或 **QT**）来创建桌面 **GUI** 的抽象级相比，似乎是非常差的选择。

1. Web UI 组件

基于组件的框架的目标是通过提供服务器端组件工具集（其 **API** 类似于桌面 **GUI** 组件集的 **API**），来提高 **Web UI** 编程的抽象级别。当桌面 **UI** 组件呈现自身的时候，它们通常使用低层调用在一个图形上下文上绘图，来生成几何图元、位图或者类似的东西。当基于 **Web** 的 **UI** 组件呈现自身的时候，它们自动生成能在浏览器中提供相同功能的 **HTML** 和 **JavaScript** 流，以减轻可怜的编程者在低层编程的苦差事。图 5-4 展示了基于组件的 **Web** 框架的结构。

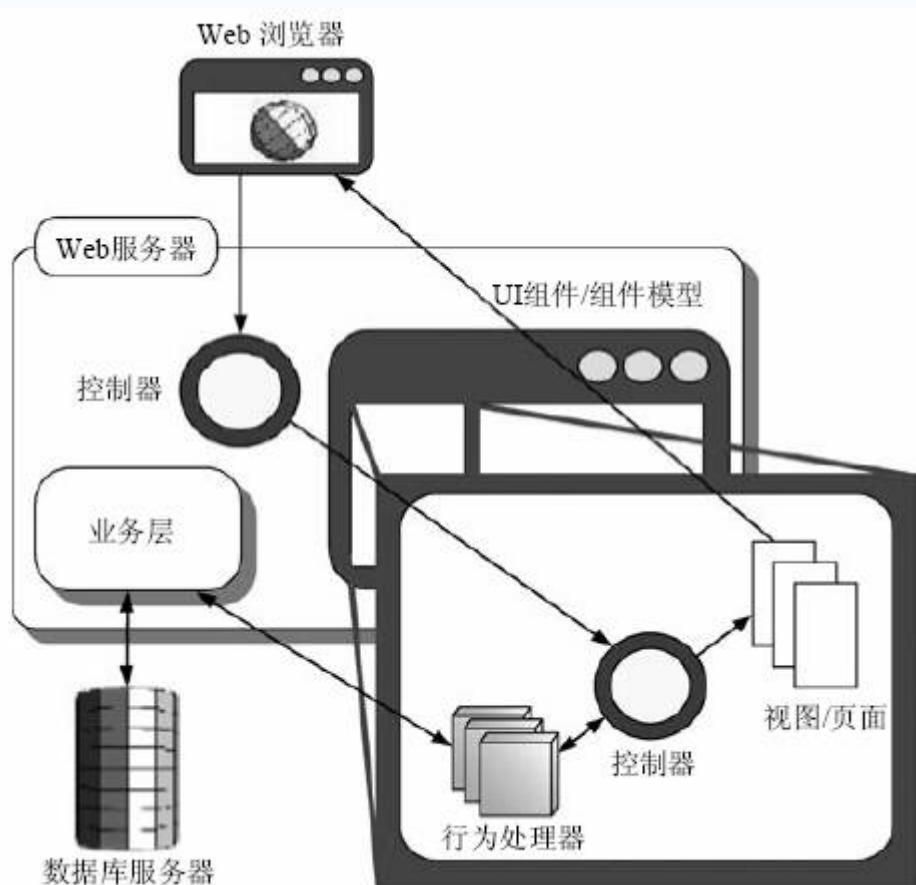


图 5-4 基于组件的框架的架构。应用被描述为一些组件的集合，它们通过向浏览器发送 HTML 和 JavaScript 流来呈现自己。除了搜集浏览器发送到单个组件的请求的较大的控制器和较大的领域模型外，每个组件还包含自己的小规模的模型、视图和控制器

很多基于组件的框架都使用桌面风格的隐喻来描述用户交互。也就是说，按钮组件应该有点击事件的处理函数，而文本域组件应该有 `valueChange` 处理函数，等等。在大多数框架中，通过每一次用户交互触发一个请求，大部分的事件处理都委派给了服务器。更加聪明的框架则设法在场景的背后完成这样的任务，但是在有些框架中每一次用户事件都会刷新整个页面。这明显导致了笨拙的用户体验，因为一个设计成 UI 组件集的应用，相对于使用 Model2 设计成页面集合的应用，通常会有大量细粒度的交互。

这些框架的重要设计目标是能够从一个单独的 UI 组件模型描述来呈现不同类型的用户界面。一些框架，例如，针对.NET 和 JSF (JavaServer Faces) 的 Windows 表单，已经能够做这件事了。

2. 与 Ajax 互操作

那么基于组件的框架如何适应 Ajax 呢？表面上看，两者都是从类似文档的界面向基于 UI 组件的界面转化的，所以出现重叠应该是件好事。如果理解 Ajax 的可插拔的呈现引擎能够开发出来，这种类型的框架会强大到足以生成客户端应用。这样做有相当大的吸引力，因为它可以避免对开发者进行再培训去学习 JavaScript 的复杂性，并且它通过普通的旧式 (plain-old) HTML 呈现系统，为较为陈旧的浏览器提供了一个替代品。

对于那些仅仅需要标准 UI 组件类型的应用，这样的解决方案会工作得很好。然而，仅有一定程度的灵活性是不够的。例如 Google Maps (参见第 1 章) 成功的很大原因就在于它定义了自己的 UI 组件集，从可滚动的地图到用于缩放的滑动条、可弹出的气球提示和地图上的大头针等等。试图使用桌面 UI 组件的标准集合来创建这些效果是非常困难的，即使最终创建出来了也可能不会令人满意。

也就是说，很多应用的确更容易适应传统范围内的 UI 组件类型，这种类型的框架可以很好地为它们服务。在灵活性和方便性之间取得平衡对于很多基于代码生成的解决方案是相通的，也很容易理解。

为了给 Ajax 应用提供完全的服务，框架也必须有能力提供必要的数据。这可能会有更多的问题，因为控制器紧紧地绑在了服务器层上，并且紧紧束缚在通过桌面隐喻来定义的机制上。有很好响应性的 Ajax 应用需要更多的自由来确定它自己的事件处理函数，而不是服务器事件模型看起来允许的那样。尽管如此，在这些框架的背后还是有股相当大的动力。无疑，伴随着 Ajax 越来越流行，更多的这类解决方案还会浮现出来。5.5.3 节将要引入的命令队列可能是向着 JSF 及其类似框架发展的一种方法，虽然它并不是为了这个目的设计的。目前从我的喜好来看，这些框架将客户端绑的有点太紧了。

观察这些框架如何在将来适应 Ajax 是一件很有趣的事情。Sun 的内部和一些 JSF 厂商对于提供支持 Ajax 的工具集已经产生了巨大的兴趣，.NET 表单已经可以支持一些类似于 Ajax 的功能，并且承诺在即将来临的 Atlas 工具集中提供更多的支持（参见本章“资源”一节以获得所有内容的 URL）。

这也提出了一个问题：如果 Web 框架专门为 Ajax 来设计，那么它看起来应该是个什么样子？今天，这类框架还没有出现，我们在这里阐述的 Web 框架终有一天会被认为是这类框架的先驱。

5.3.4 使用面向服务的体系架构

我们这里将要考察的最后一种框架是面向服务的架构（SOA）。在 SOA 中，服务（service）就是能够通过网络进行调用并作为响应返回结构化文档的东西。这里强调的是数据而不是内容，这对于 Ajax 是非常适合的。web service 是现在最常见的服务类型，它使用 XML 作为通信语言，这种方法在 Ajax 中也可以工作得很好。

注意

大写字母的 Web Service 通常是指使用 SOAP 作为传输协议的系统。广义的 web service（小写）包括任何运行在 HTTP 之上的远程数据交换系统，而不局限于只能使用 SOAP 甚至 XML 。XML-RPC 、 JSON-RPC，以及任何使用 XMLHttpRequest 对象开发的定制系统都是 web service，而不是 Web Service 。我们这一节讨论的是广义的 web service。

当使用 web service 来获取所需的数据时，Ajax 客户端达到高度的独立性，这有点类似于与邮件服务器通信的桌面邮件客户端。这是一种与基于组件工具集所能提供的不同类型的重用方式。在那里，客户一旦被定义后可以暴露给多个用户界面；而在这里，服务一旦被定义后可以由大量不相关的客户使用。很明显，SOA 和 Ajax 的结合是很强大的，我们将会看到个别框架逐渐进化，来生成和服务于 Ajax 应用。

1. 向 Ajax 暴露服务器端对象

已经出现的很多 SOA 和 web service 工具集使得以下开发方式成为可能：通过对对象方法和 web service 接口之间的一一映射，将使用 Java 、C# 或 PHP 编写的普通的旧式服务器端对象直接作为 web service 暴露。微软的 Visual Studio 工具支持这种方式，Apache 的 Axis for Java 也支持这种方式。许多 Ajax 工具集例如 DWR（用于 Java）和 SAJAX（用于 PHP 、.NET 、Python 以及几种其他语言）通过特定于 JavaScript 的客户端代码增强了这样的能力。

这些工具集非常有用，但是如果不小心的话，也有可能会被滥用。为了搞清楚如何正确使用这些工具，让我们来看一个使用 Java DWR 工具集的简单例子。我们定义了一个服务器端对象来代表一个人。

```
package com.manning.ajaxinaction;
```

```
public class Person{  
    private String name=null;  
  
    public Person() {  
    }  
  
    public String getName(){  
        return name;  
    }  
  
    public void setName(String name){  
        this.name=name;  
    }  
}
```

这个对象必须遵循基本的 **JavaBeans** 规约。也就是说，它必须提供一个公共的无参数构造函数，分别通过获取和设置方法来暴露想要读写的任何字段。接着我们通过编辑 **dwr.xml** 文件，告诉 DWR 把这个对象暴露给 **JavaScript** 层。

```
<dwr>
<init>
  <convert id="person" converter="bean"
    match="com.manning.ajaxinaction.Person"/>
</init>
<allow>
  <create creator="new" javascript="person">
    <param name="class" value="com.manning.ajaxinaction.Person">
  </create>
</allow>
</dwr>
```

在<init> 部分中，为 Person 类定义了一个转换器，类型为 bean；在<allow> 部分，定义了一个创建器，可以将对象实例暴露为一个名为 person 的 JavaScript 变量。Person 对象只有一个公共的方法 getName()，因此在 Ajax 客户端代码中可以这样写：

```
var name = person.getName();
```

然后从服务器异步取回 name 的值。

Person 对象只有一个方法，因此这就是我们已经暴露出来的全部内容了，对不对？很不幸，这是一个错误的假设。Person 类是由 `java.lang.Object` 继承而来的，也就继承了一些公共的方法，例如 `hashCode()` 和 `toString()`，这些也可以在服务器端进行调用。这些隐藏的特征并不是 DWR 特有的。例如 `JSONRPCBridge.registerObject()` 方法也能做相同的事。值得表扬的是，DWR 的确提供了一种机制来在它的 XML 配置文件中限制对于特定方法的访问。然而，默认的行为是暴露所有的内容。这个问题对于大多数基于反射的解决方案是与生俱来的。第 4 章里使用 `JavaScript` 反射的 `ObjectViewr` 工具的早期版本中也存在这样的问题。让我们看看怎么解决这个问题。

2. 有限的暴露

我们意外地将散列码暴露给了 Web，但是真的造成了破坏吗？在这个情况下也许没有，因为超类是 `java.lang.Object`，它不太可能被改变。但是在更加复杂的领域模型中，就有可能暴露自己超类中以后也许还想重构的实现细节。就在我们考虑这个问题的时候，一些聪明绝顶的开发者已经发现了我们无意中暴露的方法，并且使用在了他们的客户端代码中。因此当我们部署重构过的对象模型时，他们的客户端代码突然失效了。换句话说，我们没有能够充分地分离关注点。所以如果使用 DWR 或者 JSON-RPC 这样的工具集，就要十分小心地确定哪些对象将要作为 Ajax 接口来发布，最好能创建一个某种类型的 Façade 对象(图 5-5)。

在这种情况下使用 Façade 可以带来几个好处。首先，正如前面提到过的，它允许我们无需任何担心地重构服务器端模型。第二，它简化了客户端代码将要使用的公开发布的接口。与内部使用的代码相比，向其他团体发布的接口是昂贵的。除非我们事先详细地为这些接口编写文档，如果不写文档，我们就会被来自使用这些接口的人们的询问电话所淹没。

Façade 的另外一个好处就是允许我们与领域模型设计相互独立地定义服务的粒度级别。好的领域模型可能包含很多小的、精确的方法，因为在服务器端代码中需要这样的精确和控制。然而由于网络的延迟，应用于 Ajax 客户端的 web service 接口的需求大不相同。大量的小方法调用会毁掉客户端的可用性，如果（客户端应用）部署的数量足够多，（产生的负载）还会堵塞服务器甚至堵塞网络。

把这种情况想象成面对面地交谈和写信之间的区别（对于那些太年轻太新潮以至于不知道笔和纸为何物的新新人类来说，或者可以想象成即时通信工具交谈和写电子邮件之间的区别）。当我直接与你交谈的

时候，中间有许多小的信息交换，其中的一些可能仅仅是彼此之间的问候信息。但是当写信的时候，我会在信中描述我的健康状态、最近的休假、在家里做什么以及某天听说的一个笑话，把所有这些作为一份单独的文档来发送。

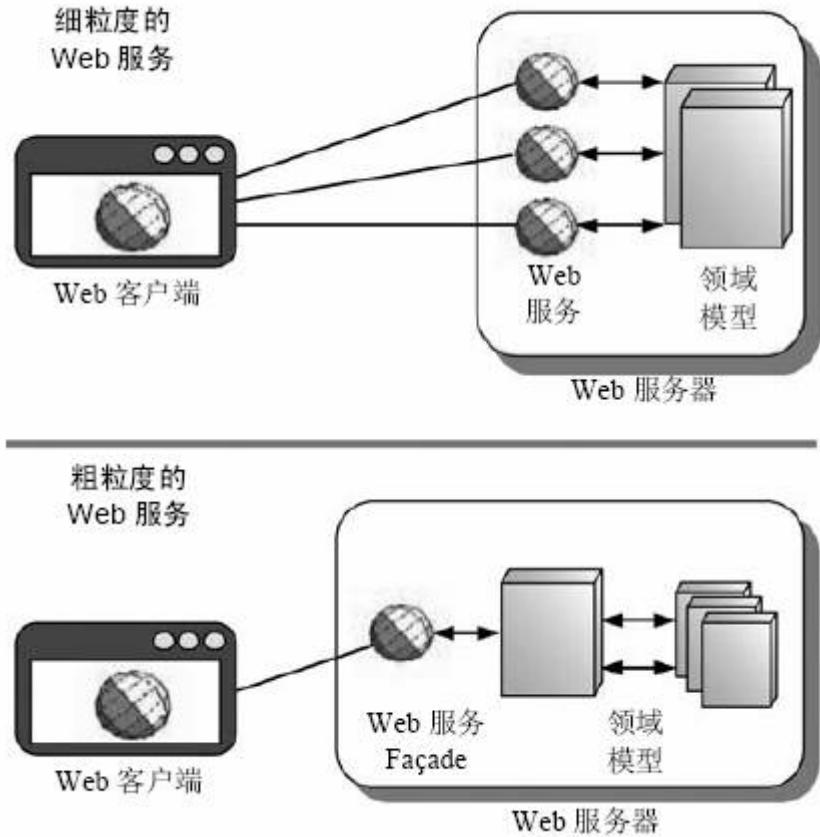


图 5-5 两个系统的对比，一个系统将所有的对象完全暴露为因特网服务给 Ajax 客户端使用；另一个系统使用 Façade 来暴露少量经过仔细挑选的功能。通过减少公开发布的方法的数量，可以重构领域模型，而不必担心客户端没有控制权的代码会失效

通过将跨网络的调用打包在更大的文档中，面向服务的架构可以更好地利用可用的网络资源。带宽相对于延迟来说通常是比较小问题。虽然它们也会因为在冗长的传输协议上（我们所熟悉和热爱的 HTTP。）对大块的 XML 数据格式进行标准化而引起问题，但是那不是我们需要在这里考虑的问题。如果我们考察一下 Ajax 可以使用的选项，就会发现在浏览器中为 HTTP 和 XML 技术提供了良好的本地支持，因此以文档为中心的方法对于分布式领域模型来说是很有意义的。

传统的文档（例如本书）由段落、标题、表格和图组成。同样的，对服务进行一次调用的文档也包含各种元素，例如查询、更新和通知。第 3 章中讨论的 Command 模式，为将文档结构化为一系列在客户端和服务器间传递的可以重做的操作提供了很好的基础。我们将在本章的稍后部分考察一个这样的实现。

这就结束了我们对于目前的服务器端架构的讨论。还没有哪种架构可以完美地适应 Ajax，但这没有什么可奇怪的，因为它们设计用来为相当不同类型的 Web 应用提供服务。大量将 Ajax 内建到现有的框架中的工作正在进行，2006 年应该是很有趣的一年。尽管如此，很多 Web 开发者还是要面对使 Ajax 与这些遗留系统¹配合工作的任务。这里对于各种方案的优点和缺点的概述应该为这些工作提供了起始点。

假设目前我们已经决定采用某种架构，并且开始了 Ajax 应用的开发工作。我们已经在第 4 章详细讨论了客户端应用的架构，并且在第 2 章提供了从服务器获得 XML 数据的例子。XML 很流行，但不是在客

户端和服务器端交换数据的唯一方式。在下节中，我们回顾一下客户和服务器之间通信的所有选项构成的完整内容。

5.4 细节：交换数据

我们已经考察了描述 Web 应用如何运转的大架构模式，前面的讨论显示出有很多可供选择的选项。我们强调客户端和服务器的领域模型之间通信的重要性，并且可能天真地假设一旦决定了采用某个框架，我们的设计选择就已经做出。在本节和下一节，我们会看到这其实和真相相距遥远。如果我们将精力集中在单一的数据交换上，我们有很多选项。致力于为 Ajax 数据交换开发一套模式语言，我们将对这些选项做一下分类。完成了这个工作，我们就能对在特定环境下使用什么技术作出更加明智的选择了。

交换纯粹的数据在传统 Web 应用中并没有类似的东西，所以模式语言在这个领域并没有得到很好的开发。我尝试通过定义一些自己的术语来填补这个空白。作为首先的切入点，我建议将用户交互分成四种类型：仅限于客户端的（client-only）、以内容为中心的（content-centric）、以脚本为中心的（script-centric）和以数据为中心的（data-centric）。仅限于客户端的交互是很简单的，我们在下一小节简单讨论一下，然后引入一个例子来仔细研究其余的三种类型。

5.4.1 仅限于客户端的交互

仅限于客户端的交互是指用户交互由已经加载到浏览器中的脚本来处理的交互。不需要任何 Web 服务器（老的表现层）的资源，这对于提高应用的响应速度和降低服务器的负载都有好处。这类交互适合于相对琐碎的计算，例如给消费者的订单上增加销售税或运输费用等等。总的说来，为了有效地使用这种方式，处理这类交互的客户端逻辑必须很少，并且在消费者交互的生命周期内保持不变。在运输的选项中，因为选项的数目在 2 到 5 之间，而不是有几千个（不像一个在线零售商的全部目录），并且运输价格也不可能每分钟都在变化（不像一台证券报价机或者先来先服务的订票系统），所以是很安全的。这种类型的交互在第 4 章对于客户端控制器的讨论中已经探索过了，所以我们这里不再赘述。

剩下的三种类型都涉及与后台进行通信，区别主要在于返回的内容不同。关键的区别以及各自的优缺点将在下面的几节中加以总结。

5.4.2 介绍行星浏览器的例子

在深入到不同的数据交换机制之前，先介绍一个简单的例子，以此来与我们的讨论挂钩。这个应用展现了太阳系行星的一组事实。主屏幕显示的是太阳系的理想化视图，每个行星有各自的图标。在服务器上已经记录了这些行星的不同事实，这些信息可以通过点击行星的图标显示在弹出窗口中（图 5-6）。我们在这里不是使用第 4 章的 ObjectViewer，但在本章的稍后将会回到它上面。

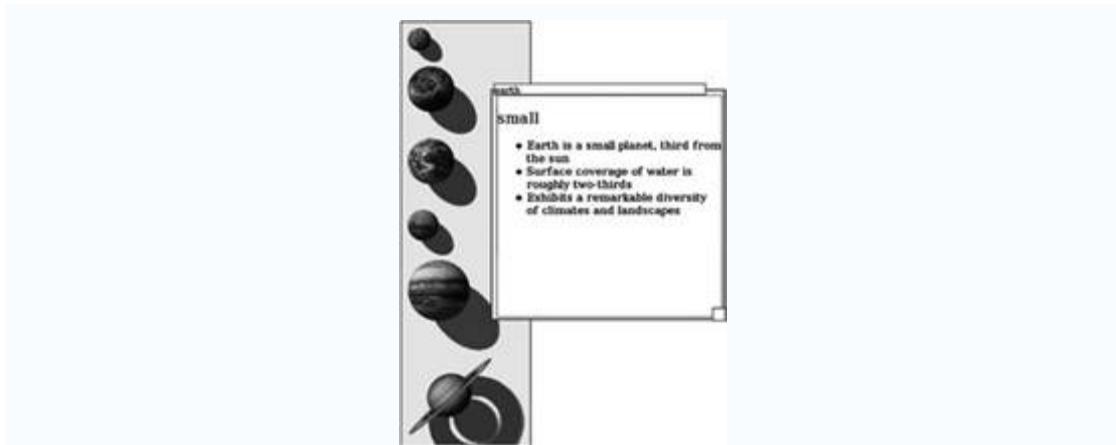


图 5-6 行星信息应用的屏幕截图，其中可以通过点击行星图标弹出描述每个行星的窗口

现在部分令我们感兴趣的难题是从服务器向浏览器交付数据，这些数据显示在弹出窗口中。我们将要考察每种交互类型中服务器发送的数据格式，但是不会涉及生成这些数据的细节，因为在第 3 章对 MVC 的讨论中已经提到过这些原理。代码清单 5-1 显示了客户端应用的架构，我们可以围绕它来探索不同的内容交付机制。

代码清单 5-1 popups.html

```
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
<title>Planet Browser</title>
<link rel=stylesheet type="text/css"
 href="main.css"/>
<link rel=stylesheet type="text/css"
 href="windows.css"/>
<link rel=stylesheet type="text/css"
 href="planets.css"/>

<script type="text/javascript"
 src="x/x_core.js"></script>
<script type="text/javascript"
 src="x/x_event.js"></script>
<script type="text/javascript"
 src="x/x_drag.js"></script>
<script type="text/javascript"
 src="windows.js"></script>
<script type="text/javascript"
 src="net.js"></script>
```

① 包括的 JavaScript 库

```

<script type="text/javascript">

window.onload=function(){
    var pbar=document.getElementById("planets");
    var children=pbar.getElementsByTagName("div");
    for(var i=0;i<children.length;i++){
        children[i].onclick=showInfo; ② 为图标分配事件处理函数
    }
}

</script>

</head>
<body>

<div class="planetbar" id="planets"> ③ 为行星添加硬编码的图标
<div class="planetbutton" id="mercury">
    
</div>
<div class="planetbutton" id="venus">
    
</div>
<div class="planetbutton" id="earth">
    
</div>
<div class="planetbutton" id="mars">
    
</div>
<div class="planetbutton" id="jupiter">
    
</div>
<div class="planetbutton" id="saturn">
    
</div>
<div class="planetbutton" id="uranus">
    
</div>
<div class="planetbutton" id="neptune">
    
</div>
<div class="planetbutton" id="pluto">
    
</div>
</div>
</body>
</html>

```

我们在文件中包含了一些 JavaScript 库①。net.js 使用第 2 章描述的 XMLHttpRequest 对象来处理底层的 HTTP 请求。window.js 定义了可拖放的窗口对象，可以用来作为弹出的窗口。除了构造函数的签名，我们在这里不关心窗口的实现细节：

`var MyWindow = new Window(bodyDiv,title,x,y,w,h);` 这里 bodyDiv 是添加进窗口体内的 DOM 元素，title 是在窗口标题栏上显示的字符串，x、y、w 和 h 描述窗口初始的几何形状。通过指定一个 DOM 元素作为参数，对于如何为窗口提供内容带来了相当大的灵活性。链接本书的可下载源代码中包含了 Window 对象的完整清单。

在 HTML 中，我们为每个行星简单地定义了一个 div 元素③，并在 window.onload 函数中使用标准的 DOM 树导航方法为每个 div 设定了一个 onclick 处理函数②。onclick 处理函数 showInfo() 并没有在这里

定义，因为我们将会在本章中提供多个不同的实现。现在我们开始考察当加载内容的时候，可以采取的不同操作。

5.4.3 从Web页面的角度思考：以内容为中心...

正如在第1章讨论马和自行车的时候提到的，向着Ajax前进的开始阶段与正在远离的传统Web应用是相似的。虽然内容为中心模式的交互仍然遵循传统的Web开发范例，但是它在Ajax应用中仍然占有席之地。

1. 概览

在以内容为中心模式的交互中，HTML内容仍然由服务器来生成，然后发送到嵌入在主要Web页面内的IFrame。在第2章我们已经讨论了IFrame，并且演示了如何在页面的HTML标记中定义它们或者通过编程方式生成它们。在后一种情况下，我们看到的是一种相当激进的动态风格的界面，比桌面应用更加类似于窗口管理器¹。图5-7描绘了内容为中心的架构的轮廓。

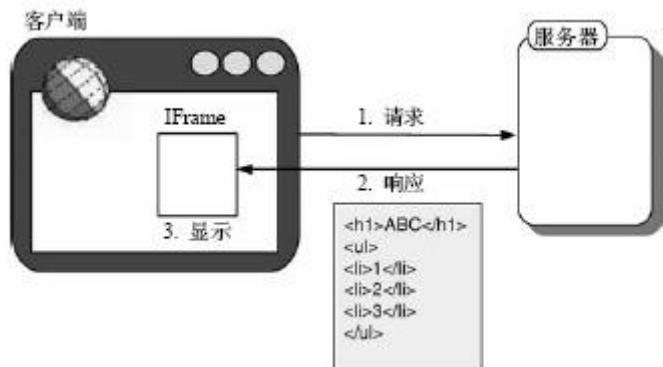


图5-7 Ajax应用中以内容为中心的架构。客户端创建了一个IFrame，并发起到服务器的请求以获得内容。内容由服务器表现层上的模型、视图、控制器生成并返回给IFrame。客户端层上不需要有业务领域模型

代码清单5-2使用内容为中心的方法，展示了行星信息应用中事件处理函数的实现。`showInfo()`就是代表行星的DOM元素的事件处理函数。在事件处理函数内部，`this`代表DOM元素，我们使用元素的`id`来确定要显示哪一个行星的信息。

代码清单5-2 ContentPopup.js

```

var offset=8;

function showInfo(event){
    var planet=this.id;
    var infoWin=new ContentPopup(
        "info_"+planet+".html",
        planet+"Popup",
        planet,offset,offset,320,320
    );
    offset+=32;
}

function ContentPopup(url,winEl,displayStr,x,y,w,h){

    var bod=document.createElement("div");
    document.body.appendChild(bod);

    this.iframe=document.createElement("iframe");
    this.iframe.className="winContents";
    this.iframe.src=url;
    bod.appendChild(this.iframe);

    this.win=new windows.Window(bod,displayStr,x,y,w,h);
}

```

我们定义了组成普通窗口对象之一的 `ContentPopup` 对象，创建了一个 `IFrame` 用作窗口体内的主要内容，并将给定的 URL 加载进来。在这个情况下，我们简单地将静态 HTML 文件的名字作为 URL。在更加复杂的带有动态生成数据的系统中，可能还要给 URL 加上查询字符串参数。在这个例子中，我们加载进 `IFrame` 的简单文件由服务器生成，显示在代码清单 5-3 中。没有什么值得注意的——我们可以像在传统 Web 应用中一样只使用普通的 HTML。

代码清单 5-3 `info_earth.html`

```

<html>
<head>
<link rel="stylesheet" type="text/css" href="../style.css"/>
</head>
<body class="info">
<div class="framedInfo" id="info">
<div class="title" id="infotitle">earth</div>
<div class="content" id="infocontent">
A small blue planet near the outer rim of the galaxy,
third planet out from a middle-sized sun.
</div>
</div>
</body>
</html>

```

在以内容为中心的模式中，客户层的代码负责放置 `IFrame` 和构造必需的 URL 来调用内容，它只需要对应用业务逻辑的有限了解。客户端和表现层间的耦合非常松散，因为大部分的职责仍然加载在服务器上。这种交互风格的好处是有大量可以备用的HTML 分散在Web 上。它在两种场景下会很有用：

从外部站点——可能是商业伙伴或者公共服务——合并内容，以及从一个应用显示遗留的内容。**HTML** 标记非常有效率，几乎没有必要将某种内容转换为应用风格的内容。帮助页面就是一个最好的例子。很多情况下，传统的**Web** 应用会使用一个弹出窗口，**Ajax** 应用更喜欢使用以内容为中心的代码片段¹，特别是考虑到最近很多浏览器都提供了弹出窗口拦截的功能。

所以，这种模式在有限的场合下很有用。在继续讨论之前，我们简短地回顾一下它的局限性。

2. 问题和局限

因为与传统**Web** 页面如此的相像，所以以内容为中心的交互也具有大多数老方法的局限性。内容文档在**IFrame** 中，与它所嵌入页面相隔离，这在某种程度上分割了屏幕真正的所有权。在布局方面，**IFrame** 给予文档强加了一个矩形窗口，虽然可以通过分配一个透明的背景来将它混合进父文档中。

在高度动态的应用中，使用这种机制来交付高度动态的子页面可能会让人感觉很有诱惑力，但是以这样的方式引入**IFrame** 可能会存在问题。每一个**IFrame** 都要维护自己的脚本上下文，在**IFrame** 和它的父窗口的脚本中相互对话所需要的重复代码数量是很可观的。如果再加上用来与其他**IFrame** 中的脚本通信的代码，这个问题就更严重了。很快，我们在考察以脚本为中心的模式时还会回到这个问题。

我们同样也会遭遇到很多传统**Web** 应用的可用性问题。首先，如果**IFrame** 的布局包含一些重要的模板标记²，我们还需要为每次的内容请求再次传送静态内容。其次，在数据刷新的时候，如果**IFrame** 重用来发送多次内容请求的话，虽然主文档不再会有闪烁感，但**IFrame** 仍然会有闪烁感。后一个问题是可以避免的，例如通过一点点额外的编码在帧的顶部显示一条正在加载的信息。

“以内容为中心”是**Ajax** 服务器请求技术词汇表中的第一个新术语。以内容为中心的方法在用途上是有限的，不过给它取一个名字仍然很好。还有很多场景下的问题不是以内容为中心的方法可以很容易解决的，例如更新某个**UI** 组件表面的一小部分、单个图标或表格的一行等等。执行这样修改的一种方法是（从服务器）发送**JavaScript** 代码，我们现在就来考察一下。

3. 变异

到目前为止，我们应用的以内容为中心的风格是：使用一个**IFrame** 来接收服务器生成的内容。另外一种替代方法也可以认为是以内容为中心的：（服务器）响应异步请求生成一段**HTML** 片断，然后将响应的内容赋值给当前文档的一个**DOM** 元素的**innerHTML** 属性³。我们将会在第 12 章**XSLT** 驱动的电话簿的例子中使用这个方法，因此这里不再给出完整的例子。

5.4.4 从插件的角度思考：以脚本为中心的…

当**JavaScript** 文件从**Web** 服务器发送到浏览器，并在浏览器里执行的时候，我们实际上是在做一件非常复杂的事情。如果正在发送的**JavaScript** 是由程序生成的，那么我们就是在创建更加复杂的系统了。传统上，客户/服务器程序之间通信交换的是数据。跨越网络通信交换可执行的移动代码带来了巨大的灵活性。企业级的网络语言（像**Java** 和**.NET** 语言栈）通过像**RMI**、**Jini** 和**.NET** 远程框架这样一些技术，刚刚得到了移动代码的可能性。而像我们这样的一些轻量级的**Web** 开发者做这件事情已经有很多年了！和通常一样，**Ajax** 让我们可以使用这种能力来做一些新鲜有趣的事情，下面让我们来看看都是些什么事情。

1. 概述

在传统的**Web** 应用中，一段**JavaScript** 代码和它相关联的**HTML** 是打包在一起交付的，并且脚本通常被编写来只与那个特定的页面一起工作。使用**Ajax**，可以彼此独立地加载脚本和页面，这带给我们一种可能性：依靠加载的脚本，使用多种不同方式修改特定的页面。组成客户层应用的代码可以在运行时有效地扩展。这样做既引入了机遇，也引入了问题，正如我们将会看到的。图 5-8 展示了以脚本为中心的应用的架构。

这种方法相比以内容为中心的解决方案，第一个优点是网络活动转移到了后台，消除了可见的闪烁感。

生成脚本的确切性质依赖于客户层本身所暴露出的钩子¹。对于大量的代码生成来说，成功的关键在于保持生成部分的简单，并且在可能的地方使用非生成的（静态的）库代码。这些库代码要么随着生成的代码一起传输，要么驻留在客户应用中。

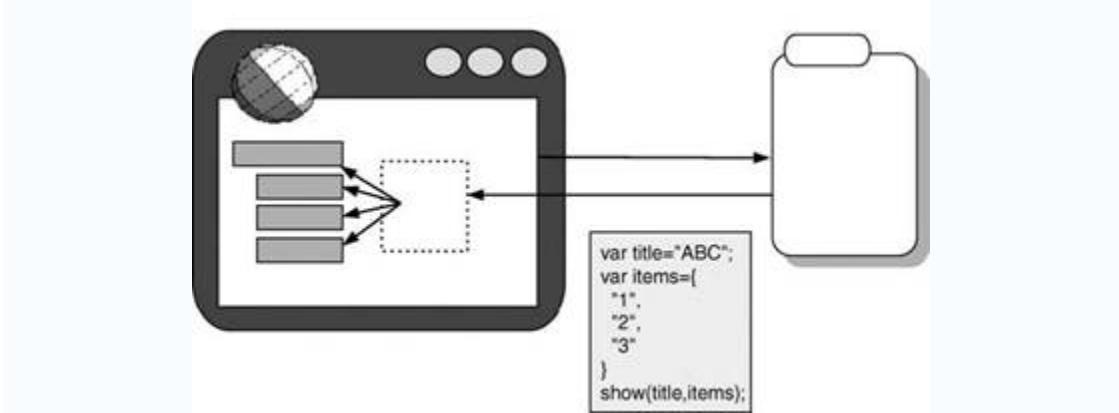


图 5-8 Ajax 应用中以脚本为中心的架构。客户端应用向服务器请求一段 JavaScript，然后解释执行。

客户端应用暴露一些入口点来与生成的脚本挂钩，并允许脚本对客户端进行操作

另一方面，这种模式导致了层间相对紧密的耦合。也就是说，服务器生成的代码需要了解客户端 API 调用的细节。两个问题浮现了出来。首先，服务器或客户端代码的修改会无意中使得另外一方的代码失效。好的模块化设计原则，通过提供良好定义的、文档齐全的 API——并且实现 Façade 模式，可以在某种程度上弥补这个缺陷。第二个问题是 JavaScript 流非常特定地为这个客户端而设计，所以不大可能在其他上下文环境中重用，例如 XML 流。然而，重用性并不是在所有的情况下都很重要。

我们再来看看那个行星信息应用的例子。代码清单 5-4 展示了一个简单的用来显示信息窗口的 API。

代码清单 5-4 showPopup()方法和支持代码

```
var offset=8;
function showPopup(name,description){
    var win=new ScriptIframePopup
        (name,description,offset,offset,320,320);
    offset+=32;
}
function ScriptIframePopup(name,description,x,y,w,h){
    var bod=document.createElement("div");
    document.body.appendChild(bod);
    this.contentDiv=document.createElement("div");
    this.contentDiv.className="winContents";
    this.contentDiv.innerHTML=description;
    bod.appendChild(this.contentDiv);
    this.win=new windows.Window(bod,name,x,y,w,h);
}
```

我们定义了函数 showPopup，它使用名称和描述作为参数来创建一个窗口对象。代码清单 5-5 显示的是调用这个函数的脚本例子。

代码清单 5-5 script_earch.js

```
var name='earth';
var description="A small blue planet near the outer rim of the galaxy,"
+"third planet out from a middle-sized sun.";

showPopup (name,description);
```

我们简单地定义了参数，然后对 API 进行调用。在场景的背后，需要从服务器加载这个脚本并且设法使浏览器执行它。可以使用两种完全不同的实现途径，让我们依次研究一下。

2. 加载脚本到 IFrame

如果使用 HTML 文档的<script> 标记来加载 JavaScript 脚本，当加载的时候，脚本会由解释器自动执行。在这方面，IFrame 和任何其他的文档都是一样。我们可以定义 showInfo() 方法来创建一个 IFrame，然后把脚本加载进去。

```
function showInfo(event) {
    var planet=this.id;
    var scriptUrl="script_"+planet+".html";
    var dataframe=document.getElementById('dataframe');
    if (!dataframe){
        dataframe=document.createElement("iframe");
        dataframe.className='dataframe';
        dataframe.id='dataframe';
        dataframe.src=scriptUrl;
        document.body.appendChild(dataframe);
    }else{
        dataframe.src=scriptUrl;
    }
}
```

我们使用的 DOM 操作方法应该已经是很熟悉的了。如果使用不可见的 IFrame 来加载脚本，就只需要将注意力集中在脚本自身的生成上，因为所有其他的交互是自动生成的。现在我们把例子脚本包括在一个 HTML 文档中，如代码清单 5-6 所示。

代码清单 5-6 script_earch.html

```
<html>
<head>
<script type='text/javascript' src='script_earth.js'>
</script>
</head>
<body>
</body>
</html>
```

当试图加载这段代码时，它并不能正常工作，因为 IFrame 创建了自己的 JavaScript 上下文，不能直接看到主文档中定义的 API。当在脚本中声明：

showPopup(name,description); 浏览器就在 IFrame 的上下文中寻找定义的 showPopup() 方法。在类似这样的简单的两个上下文情况下，为了引用顶层（top-level）的文档，可以给 API 调用加一个 top 前缀，即，

```
top.showPopup(name,description);
```

如果 IFrame 中还嵌套有 IFrame，或者想要在帧集中运行应用，事情就会变得更加复杂了。我们加载的脚本使用了一个函数化的方法。如果我们选择在 IFrame 脚本中实例化一个对象，就会遇到更大的复杂性。假设有一个 PlanetInfo.js 文件，它定义了一个 PlanetInfo 类型的对象，在脚本中这样调用：

```
var pinfo = new PlanetInfo(name,description); 为了在脚本中使用这个类型，可以通过添加额外的脚本标签来将 PlanetInfo.js 引入到 IFrame 的上下文中：
```

```
<script type='text/javascript' src='PlanetInfo.js'></script>
```

```
<script type='text/javascript'>
```

```
var pinfo=new PlanetInfo(name,description);
```

```
</script>
```

在`IFrame` 中创建的`PlanetInfo` 对象与在顶层帧¹创建的`PlanetInfo` 对象具有完全相同的行为，但是二者不具有相同的原型（`prototype`）。如果`IFrame` 后来无效了，但是顶层文档仍然保存着某个由`IFrame` 创建的对象的引用，对这个对象的方法的后续调用就会失败。此外，`instanceof` 操作符还会有违反直觉的行为，如表 5-1 所示。

表 5-1 跨不同帧的 `instanceof` 操作符的行为

对象创建位置	调用 <code>instanceof</code> 的位置	Obj <code>instanceof</code> 对象的值
顶层文档	顶层文档	true
顶层文档	<code>IFrame</code>	false
<code>IFrame</code>	顶层文档	false
<code>IFrame</code>	<code>IFrame</code>	true

将相同的对象定义引入到多个脚本上下文中乍看起来并不是那么简单的。我们可以提供一个工厂方法作为顶层文档的 API 的一部分来避免这样的情况，比如：

```
function createPlanetInfo(name,description){
```

```
    return new PlanetInfo(name,description);  
  
}
```

然后脚本就可以调用这个工厂方法，不再需要引用它

```
自己版本的 PlanetInfo 类型，即：
```

```
function evalScript(){  
  
    var script=this.req.responseText;  
  
    eval(script);  
  
}  
  
showInfo() 方法使用 XMLHttpRequest 对象（封装在 ContentLoader 类中）从服务器取回脚本，而不需要将脚本封装在 HTML 页面中。第二个函数 evalScript() 作为一个回调函数传到 ContentLoader 中，在其中可以读取 XMLHttpRequest 对象的 responseText 属性。整个脚本在当前的页面上下文中求值，而不是在 IFrame 的分离的上下文中求值。
```

我们现在可以添加以脚本为中心这个术语到模式语言中，并且注明它有两种实现：使用 **IFrame** 和使用 **eval()** 函数。让我们回过头来看看基于脚本的方法与基于内容的风格之间的比较。

4. 问题和局限

当直接从服务器加载脚本的时候，通常是传输更为简单的消息，这在某种程度上减小了带宽耗费。我们也在很大程度上将逻辑与表现解耦，直接带来的实用后果是，可视的变化不会像在以内容为中心的方法中那样被限制在屏幕的一个固定的矩形区域。

但是在不利方面，我们引入了客户端和服务器代码之间的紧密耦合。服务器发出的 JavaScript 不大可能在其他上下文中重用，我们需要为特定的 **Ajax** 客户端编写。此外，一旦发布，客户端提供的 API 相对来说就很难改变。

尽管如此，这是迈向正确方向的一步。**Ajax** 应用的行为开始更像是应用¹而不太像是文档²了。在下面将要讨论的客户—服务器通信的风格中，我们可以释放这里所引入的客户和服务器之间的紧耦合。

5.4.5 从应用的角度思考：以数据为中心的...

在刚才描述的以脚本为中心的方法中，应用的行为开始更像是传统的胖客户端（**thick client**），发到服务器的数据请求发生在后台，与用户界面解耦。尽管如此，脚本的内容仍然非常特定于基于浏览器的客户端。

1. 概述

在有些情况下，**Ajax** 客户端可能希望和其他类型的前端共享数据，例如 **Java**、**.NET** 智能客户端、蜂窝电话/PDA 客户端软件等等。在这种情况下，我们可能更喜欢使用一种中立的数据格

式而不是一套 JavaScript 指令。在以数据为中心的解决方案中，服务器提供纯数据流，由自己的客户端代码而不是 JavaScript 引擎来解析。图 5-9 展示了以数据为中心的解决方案的特征。

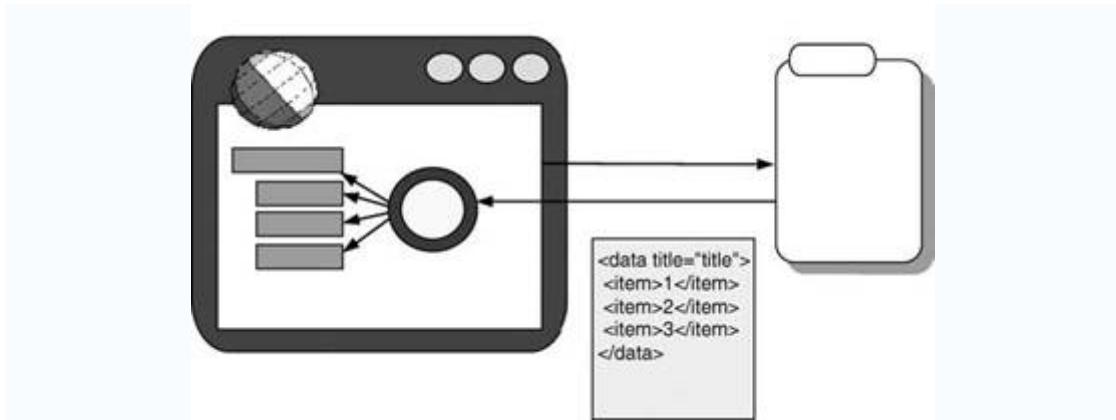


图 5-9 在以数据为中心的系统中，服务器返回原始数据流（这里是 XML），它在客户层解析，然后更新客户端层的模型和（或）用户界面

本书中的大多数例子都遵循以数据为中心的方法。最显而易见的数据格式是 XML，但是采用其他格式也是可能的，我们下面就会看到。

2. 使用 XML 数据

XML 在现代计算技术中是一种几乎无处不在的数据格式。Ajax 应用所处的 Web 浏览器环境，特别是 XMLHttpRequest 对象，对于处理 XML 提供了很好的本地支持。如果 XMLHttpRequest 接收到了一个 XML 内容类型，例如 application/xml 或 text/xml，它会将响应表现为一个 DOM，正如我们已经看到的一样。代码清单 5-7 显示了如何使用 XML 数据改写行星信息应用。

代码清单 5-7 DataXMLPopup.js

```
var offset=8;
function showPopup(name,description){
    var win=new DataPopup(name,description,offset,320,320);
    offset+=32;
}

function DataPopup(name,description,x,y,w,h){
    var bod=document.createElement("div");
    document.body.appendChild(bod);

    this.contentDiv=document.createElement("div");
    this.contentDiv.className="winContents";
    this.contentDiv.innerHTML=description;
```

```

bod.appendChild(this.contentDiv);

this.win=new windows.Window(bod,name,x,y,w,h);
}

function showInfo(event){
var planet=this.id;
var scriptUrl=planet+".xml";
new net.ContentLoader(scriptUrl,parseXML);
}

function parseXML(){
var name="";
var descrip="";
var xmlDoc=this.req.responseXML;
var elDocRoot=xmlDoc.getElementsByTagName("planet")[0];
if (elDocRoot){
    attrs=elDocRoot.attributes;
    name=attrs.getNamedItem("name").value;
    var ptype=attrs.getNamedItem("type").value;
    if (ptype){
        descrip+="

## "+ptype+"

";
    }

    descrip+="<ul>";
    for(var i=0;i<elDocRoot.childNodes.length;i++){
        elChild=elDocRoot.childNodes[i];
        if (elChild.nodeName=="info"){
            descrip+="<li>"+elChild.firstChild.data+"</li>\n";
        }
    }
    descrip+="</ul>";
} else{
    alert("no document");
}
top.showPopup(name,descrip);
}

```

`showInfo` 函数简单地打开一个封装在 `ContentLoader` 对象中的 `XMLHttpRequest` 对象，提供 `parseXML()` 方法作为回调函数。这个回调函数比起 5.6.3 节中的 `evalScript()` 方法稍微麻烦一些，因为我们需要在响应的 DOM 中导航，从中抽取出数据，然后再手工调用 `showPopup()` 方法。代码清单 5-8 显示的是服务器生成的 XML 响应的例子，以 XML 数据为中心的应用中可以使用这些数据。XML 的一个很大优点是有助于对信息进行结构化。我们在这里利用这个优点提供很多的`<info>`标签，然后在 `parseXML()`函数中将它们转换成一段 HTML 的无序列表。

代码清单 5-8 earth.xml

```

<planet name="earth" type="small">
    <info id="a" author="dave" date="26
    /05/04">
        Earth is a small planet, third from th
        e sun

    </info>
    <info id="b" author="dave" date="27/02/05">

```

```
Surface coverage of water is roughly two-thirds
</info>
<info id="c" author="dave" date="03/05/05">

Exhibits a remarkable diversity of climates and landscapes
</info>
</planet>
```

通过使用 **XML**，我们达到了对服务器层和客户层的更好分离。如果两端都能理解文档的格式，那么客户和服务器的代码就可以独立于对方而修改。然而，在前一节的以脚本为中心的解决方案中，**JavaScript** 解释器所做的工作也是很好的。下面的例子使用 **JSON**，结合了这两个领域最好的优点。让我们来看一下。

3. 使用 JSON 数据

XMLHttpRequest 对象能够接收任意基于文本的信息，由此可以证明它的名字取得并不恰当。一种将数据传输到Ajax 客户端的有用数据格式是**JSON**，一种表现普通**JavaScript** 对象图¹的紧凑方式。代码清单 5-9 显示了如何使用**JSON** 改写行星信息的例子。

代码清单 5-9 DataJSONPopup.js

```
function showInfo(event){
    var planet=this.id;
    var scriptUrl=planet+".json";
    new net.ContentLoader(scriptUrl,parseJSON);
}

function parseJSON(){
    var name="";
    var descrip="";
    var jsonTxt=net.req.responseText;
    var jsonObj=eval("(" + jsonTxt + ")");
    name=jsonObj.planet.name
    var ptype=jsonObj.planet.type;
    if (ptype){
        descrip+="<h2>" + ptype + "</h2>";
    }

    var infos=jsonObj.planet.info;
    descrip+="<ul>";
    for(var i in infos){
        descrip+="<li>" + infos[i] + "</li>\n";
    }
    descrip+="</ul>";

    top.showPopup(name,descrip);
}
```

我们再一次使用 **ContentLoader** 来获取数据，并且分配一个回调函数 **parseJSON()**。整个的响应文本是一条合法的 **JavaScript** 语句，因此我们可以通过简单地调用 **eval()** 函数来创建一个对象图：

```
var jsonObj = eval("("+jsonTxt+")");
```

注意，在求值之前，我们需要将整个表达式包在圆括号内，然后就可以直接通过名字来查询这个对象的属性，这带来了比 XML 的 DOM 操作方法更加简洁和可读性更好的代码。`showPopup()` 方法被省略了，与代码清单 5-7 中完全一样。

那么 JSON 实际上像是什么样呢？代码清单 5-10 显示了行星地球的数据，作为一段 JSON 字符串。

代码清单 5-10 earth.json

```
{"planet": [
    "name": "earth",
    "type": "small",
    "info": [
        "Earth is a small planet, third from the sun",
        "Surface coverage of water is roughly two-thirds",
        "Exhibits a remarkable diversity of climates and landscapes"
    ]
}]
```

花括号表示关联数组，方括号表示数字数组。每种类型的括号都可以嵌套另外一种。这里我们定义了一个名叫 `planet` 的对象，包含三个属性。`name` 和 `type` 属性都是简单的字符串，而 `info` 属性是一个数组。

JSON 没有 XML 那么通用，尽管它能够被任何 JavaScript 引擎使用，包括基于 Java 的 Mozilla Rhino 和微软的 JScript.NET。JSON—RPC 库包含了很多编程语言的 JSON 解析器（参见本章“资源”一节），也包含了用来将 JavaScript 对象转换成 JSON 字符串的“字符串转换器”（Stringifier），以便使用 JSON 作为双向通信的机制。如果在服务器和客户两端都有一个可用的 JavaScript 解释器，那么 JSON 绝对是一个可行的选项。JSON—RPC 项目已经为很多通用的服务器端语言开发了解析和生成 JSON 数据的库。

我们现在可以将以数据为中心（`data-centric`）添加到词汇表中，相比曾经流行的 XML，还应该注意到广泛的基于文本的数据格式的潜力。

4. 使用 XSLT

手工操作 DOM 树创建 HTML（如同在 5.7.3 节中所做的一样）的另外一种选择是使用 XSLT 转换来将 XML 自动转换为 XHTML。这是以数据为中心和以内容为中心两种方法的混合物。从服务器的角度来看，它是以数据为中心的，而从客户端的角度，它看起来更像是以内容为中心的。这种方法更迅速、更简单，但是也遭受了与以内容为中心的方法相同的局限性，即响应被完全解释为可视化标记，通常只会影响到可视用户界面的单个矩形区域。第 11 章会更加详细地讨论 XSLT。

5. 问题和局限

以数据为中心的方法的主要局限是它将解析数据的负担完全交给了客户端。因此客户层的代码会变得更加复杂，但是如果在大型的应用中大规模采用这种方法，付出的成本可以通过重用解析器代码或将一些功能抽象为库来弥补。

可以证明，我们在这里提出的三种方法形成了从传统的 Web 应用模型到桌面风格的胖客户端之间的一个系列。幸运的是，这三种模式并不是互相排斥的，甚至可以全部使用在同一个应用中。

当然，客户端/服务器的通信是双向的。我们将通过考察客户端如何将数据发送到服务器来完成这一章。

5.5 向服务器写数据

迄今为止，我们一直将注意力集中在对话的一端，即服务器告诉客户端什么正在进行。在大多数应用中，用户除了看到领域模型外还希望能够操纵它。在多用户环境中，我们也希望能够接收到其他用户所做的修改。

让我们先来考虑一下更新所做的修改的情况。从技术上来讲，提交数据有两种主要的机制：使用 **HTML 表单** 和使用 **XMLHttpRequest 对象**。让我们依次简要地看一下。

5.5.1 使用 HTML 表单

在传统的 Web 应用中，HTML 表单元素是用户输入数据的标准机制。表单元素可以在页面的 HTML 标记中声明：

```
<form method="POST" action="myFormHandlerURL.php">
```

```
    <input type="text" name="username"/>
```

```
    <input type="password" name="password"/>
```

```
    <input type="submit" value="login"/>
```

```
</form>
```

这在页面上呈现为两个空的文本输入框。如果在表单中输入值 `dave` 和 `letmein`，然后一个 HTTP POST 请求就会带着文本体 `username=dave&password=letmin` 发送到 `myFormHandler URL.php`。在大多数现代 Web 编程系统中，我们并不能直接看到这些编码后的表单数据，但名称一值对可以解码为一个关联数组或者“魔术”变量。

目前相当普通的一种做法是添加一小段 **JavaScript**，提交之前在本地验证表单内容。我们可以这样修改这个简单的表单：

```
<form id="myForm" method="POST" action
      =""
      onsubmit="validateForm(); return f
      alse;">
<input type="text" name="username
      "/>

<input type="password" name="password"/>

<input type="submit" value="login"/>

</form>
```

还可以为这个页面定义一个 JavaScript 的验证程序:

```
function validateForm(){

var form=document.getElementById('myForm');

var user=form.elements[0].value;

var pwd=form.elements[1].value;
```

```
if (user && user.length>0 && pwd && pwd.length>0){  
  
    form.action='myFormHandlerURL.php';  
  
    form.submit();  
  
}  
  
else{  
  
    alert("please fill in your credentials before logging in");  
  
}  
  
}
```

这个表单最初没有定义 `action` 属性。只有当表单中的值验证正确后，`action` 属性才会被真正的 URL 替代。也可以使用 JavaScript 来增强表单的功能，通过禁用提交按钮以防止多次提交，在将数据发送到网络前对密码加密等等。这些技术在其他地方都有很好的文档，在这里我们就不深究了。第 9 章和第 10 章包含了用 Ajax 增强过的 HTML 表单的更详细的例子。

我们也可以使用编程方式构造表单元素，并且以后台方式提交。如果将表单的风格设置为不可显示的，就可以以用户看不见的方式来做这件事，如代码清单 5-11 中所示。

代码清单 5-11 `submitData()`方法

```
function addParam(form,key,value){  
    var input=document.createElement("input");  
    input.name=key;  
    input.value=value;  
    form.appendChild(input);  
}  
function submitData(url,data){  
    var form=document.createElement("form");  
    form.action=url;  
    form.method="POST";  
    for (var i in data){  
        addParam(form,i,data[i]);  
    }  
    form.style.display="none";  
    document.body.appendChild(form);  
    form.submit();  
}
```

`submitData()` 创建了表单元素，并且遍历其数据，使用 `addParam()` 方法将数据添加到表单中。我们可以这样来调用它：

```
submitData(  
    "myFormHandlerURL.php",  
    {username:"dave",password:"letmein"}  
);
```

这个技术很简洁但是有个重大的缺点，就是没有简单的方法能够捕获服务器的响应。我们可以使用一个不可见的`IFrame` 指向这个表单¹，然后解析结果，但是这是相当麻烦的办法。幸运的是，我们可以通过使用`XMLHttpRequest` 对象来达到相同的效果。

5.5.2 使用`XMLHttpRequest`对象

我们已经在第 2 章和本章前面的部分看到过`XMLHttpRequest`对象的能力。从客户端代码的角度来看，读取和更新之间的区别是细微的。我们仅仅需要指定使用`POST`方法，并且传入表单的参数就可以了。

代码清单 5-12 显示的是在 3.1 节开发的`ContentLoader`对象的主要代码。我们已经对它进行过重构，允许将参数传递给请求，并且指定任意的`HTTP`方法²。

代码清单 5-12 `ContentLoader`对象

```

net.ContentLoader=function
(url,onload,onerror,method,params,contentType){ ❶ 额外的参数

  this.onload=onload;
  this.onerror=(onerror) ? onerror : this.defaultError;
  this.loadXMLDoc(url,method,params,contentType);
}

net.ContentLoader.prototype.loadXMLDoc
=function(url,method,params,contentType){
if (!method){
  method="GET";
}
if (!contentType && method=="POST"){
  contentType="application/x-www-form-urlencoded";
}
if (window.XMLHttpRequest){
  this.req=new XMLHttpRequest();
} else if (window.ActiveXObject){
  this.req=new ActiveXObject("Microsoft.XMLHTTP");
}
if (this.req){
  try{
    this.req.onreadystatechange=net.ContentLoader.onReadyState;
    this.req.open(method,url,true);           ← HTTP 方法
    if (contentType){                      ← 内容类型
      this.req.setRequestHeader("Content-Type", contentType);

    }
    this.req.send(params);     ← 请求的参数
  }catch (err){
    this.onerror.call(this);
  }
}
}
}

```

我们传入几个新的参数到构造函数中❶。其中只有 URL（与表单的 action 属性相对应）和 onload 处理函数是必需的，但是也可以指定 HTTP 的方法、请求的参数和内容类型。注意，如果通过 POST 方法提交数据的键一值对，内容类型必须设置为 application/x-www-form-urlencoded 。如果没有指定内容类型，将会自动地指定。HTTP 方法在 XMLHttpRequest 的 open() 方法中指定，而参数在 send() 方法中指定。因此，像这样的调用：

```
var loader=net.ContentLoader(
```

```
'myFormHandlerURL.php',
```

```
showResponse,  
  
null,  
  
'POST',  
  
'username=dave&password=letmein'  
  
);
```

会与代码清单 5-11 中基于表单的 `submitData()` 方法一样执行相同的请求。注意，使用在 URL 查询串中看到的 `form-encoded` 风格，参数作为字符串对象来传递，例如：

```
name=dave&job=book&work=Ajax_In+Action
```

以上这些内容涉及了向服务器提交数据的基本机制，无论是来自表单的基于文本的输入，还是来自其他行为，例如拖放或者鼠标移动。在下一节，我们将重拾第 4 章的 `ObjectViewer` 例子，学习如何以有条理的方式管理对领域模型的更新。

5.5.3 有效地管理用户的更新

第 4 章介绍了 `ObjectViewer`，这是一段通用的代码，用来浏览复杂的领域模型，并提供了一个简单的例子来观察行星的数据。每一个代表太阳系行星的对象都包含几个参数，我们将几个简单的文本属性——直径和到太阳的距离——标记为可编辑的。对系统中任何属性的修改都会被一个中央事件监听器函数所捕获，我们用这个函数向浏览器的状态条写一些调试信息（向状态条写信息的能力在最近版本的 `Mozilla Firefox` 中受到了限制。在附录 A 中，我们介绍了一个纯粹的 `JavaScript` 日志控制台，可以在本地状态条无

法使用时，用来向用户提供状态信息）。这种事件监听器机制也为将更新发送到服务器提供了一种理想捕获更新的方法。

假设我们有一个脚本 `updateDomainModel.jsp` 运行在服务器上，负责捕获下列信息：

更新的行星的唯一 ID；

更新的属性的名称；

分配给属性的值。

我们可以编写一个事件处理函数来向服务器触发所有的更新，像这样：

```
function updateServer(propviewer){ var planetObj=propviewer.viewer.object; var planetId=pla  
netObj.id; var propName=propviewer.name; var val=propviewer.value; net.ContentLoader(  
'updateDomainModel.jsp',  
someResponseHandler,  
null,  
'POST',  
'planetId='+encodeURI(planetId)  
+'&propertyName='+encodeURI(propName)  
+'&value='+encodeURI(val)  
);  
}
```

然后将它附加到 `ObjectViewer`：

```
myObjectViewer.addChangeListener(updateServer);
```

这些代码很容易编写，但是会导致大量到服务器的小段通信流量，这是很低效的，潜在地还会引起混乱。如果我们希望控制通信流量，可以捕获这些更新，在本地将它们排队，然后在空闲时间成批地将它们发送到服务器。一个使用 `JavaScript` 实现的简单的更新队列显示在代码清单 5-13 中。

代码清单 5-13 CommandQueue 对象

```
net.CommandQueue=function(id,url,freq){  
    this.id=id;  
    net.cmdQueues[id]=this;  
    this.url=url;  
    this.queued=new Array();  
    this.sent=new Array();  
    if (freq){  
        this.repeat(freq);  
    }  
}  
  
net.CommandQueue.prototype.addCommand=function(command){  
    if (this.isCommand(command)){  
        this.queue.append(command,true);  
    }  
}  
  
net.CommandQueue.prototype.addCommand=function(command){  
    if (this.isCommand(command)){  
        this.queue.append(command,true);  
    }  
}  
  
net.CommandQueue.prototype.fireRequest=function(){  
    if (this.queued.length==0){  
        return;  
    }  
    var data="data=";  
    for(var i=0;i<this.queued.length;i++){  
        var cmd=this.queued[i];  
        if (this.isCommand(cmd)){  
  
            data+=cmd.toRequestString();  
            this.sent[cmd.id]=cmd;  
        }  
    }  
    this.queued=new Array();  
    this.loader=new net.ContentLoader(  
        this.url,  
        net.CommandQueue.onload,net.CommandQueue.onerror,  
        "POST",data  
    );  
}  
  
net.CommandQueue.prototype.isCommand=function(obj){  
    return (  
        obj.implementsProp("id")  
        && obj.implementsFunc("toRequestString")  
        && obj.implementsFunc("parseResponse")  
    );  
}
```

```

net.CommandQueue.onload=function(loader){          ④ 解析服务器的响应
    var xmlDoc=net.req.responseXML;
    var elDocRoot=xmlDoc.getElementsByTagName("commands")[0];
    if (elDocRoot){
        for(i=0;i<elDocRoot.childNodes.length;i++){
            elChild=elDocRoot.childNodes[i];
            if (elChild.nodeName=="command"){
                var attrs=elChild.attributes;
                var id=attrs.getNamedItem("id").value;
                var command=net.commandQueue.sent[id];
                if (command){
                    command.parseResponse(elChild);
                }
            }
        }
    }
}

net.CommandQueue.onerror=function(loader){
    alert("problem sending the data to the server");
}

net.CommandQueue.prototype.repeat=function(freq){    ⑤ 轮询服务器
    this.unrepeat();
    if (freq>0){
        this.freq=freq;
        var cmd="net.cmdQueues["+this.id+"].fireRequest();";
        this.repeater=setInterval(cmd,freq*1000);
    }
}
net.CommandQueue.prototype.unrepeat=function(){      ⑥ 关闭轮询
    if (this.repeater){
        clearInterval(this.repeater);
    }
}

```

`CommandQueue` 对象（这样称呼它是因为它将 `Command` 对象排队——我们一会儿就会谈到）使用唯一 ID、服务器端脚本的 URL 以及可选的表示是否重复轮询的标志来初始化❶。如果不需要重复轮询，我们就需要经常采用手工方式来触发。两种操作模式都是有用的，所以在这里把两者都包括了。当队列向服务器发送一个请求的时候，它就会将队列中的所有命令转换成字符串，然后随着请求一起发送❷。

这个队列维护了两个数组。`queued` 是按数字索引的数组，新的更新都添加到这个数组上。`sent` 是关联数组，包含那些已经发送到服务器但还在等待回应的更新。在两个队列中的对象是 `Command` 对象，遵循通过 `isCommand()` 函数指定的接口❸，即：

- u 为自己提供唯一的 ID；
- u 将自己序列化，以便包含进发送到服务器的 POST 数据中（见❹）；
- u 解析来自服务器的响应（见❺）以决定结果是成功还是失败，以及它应该采取什么进一步的行动（如果有）。

使用 `implementsFunc()` 函数来检查是否遵守了这个契约。作为基类 `Object` 的方法，你也许会以为它是标准的 JavaScript，但是实际上我们在帮助库中是这样定义它的：

```

Object.prototype.implementsFunc=function(funcName){
    return this[funcName] && this[funcName] instanceof
    Function; } 附录 B 更详细地解释了 JavaScript 的 prot

```

`otype`。现在我们还是回到队列对象。队列的 `onload()` 方法❸ 期望服务器返回一个 XML 文档,该文档由包含在 `<commands>` 标签中的`<command>` 标签组成。最后, `repeat()` ❹ 和 `unrepeat()` ❺ 方法用来管理重复计时器对象, 它通过周期性轮询的方式向服务器发送更新。更新行星属性的 `Command` 对象显示在代码清单 5-14 中。

代码清单 5-14 `UpdatePropertyCommand` 对象

```
planets.commands.UpdatePropertyCommand=function(owner,field,value){
    this.id=this.owner.id+"_"+field;
    this.obj=owner;
    this.field=field;
    this.value=value;
}

planets.commands.UpdatePropertyCommand.toRequestString=function(){
    return {
        type:"updateProperty",
        id:this.id,
        planetId:this.owner.id,
        field:this.field,
        value:this.value
    }.simpleXmlify("command");
}

planets.commands.UpdatePropertyCommand.parseResponse=function(docEl){
    var attrs=docEl.attributes;
    var status=attrs.getNamedItem("status").value;
    if (status!="ok"){
        var reason=attrs.getNamedItem("message").value;
        alert("failed to update "
            +this.field+" to "+this.value
            +"\n\n"+reason);
    }
}
```

`Command` 对象简单地为命令和它封装的服务器所需的参数提供了唯一的 ID。`toRequest String()` 函数使用附加到 `Object` 类的原型上的自定义方法, 将其自身输出为一段 XML:

```
Object.prototype.simpleXmlify=function(tagname){
```

```
    var xml+"<"+tagname;

    for (i in this){

        if (!this[i] instanceof Function){
            xml+=" "+i+"='"+this[i]+"'";
        }

    }
    xml+="/>";
    return xml;
}
```

```
}
```

这将创建一个的简单的 XML 标签，像这样（这里为了清晰，手工格式化了一下）：

```
<command type='updateProperty'  
         id='001_diameter'  
         planetId='mercury'  
         field='diameter'  
         value='3'>
```

注意，唯一的 ID 仅仅由行星 ID 和属性名称组成。我们不能将同一个值的多份修改发送到服务器。如果在队列发送前确实几次编辑了一个属性，每个后来的值会覆盖先前的值。

依赖于轮询的频率和用户的繁忙程度，发送到服务器的 POST 数据将包含一个或多个这样的标签。服务器进程需要处理每个命令并将结果存储在类似的响应中。CommandQueue 的 onload() 方法会与发送队列中 Command 对象的响应的每一个标签相匹配，然后调用命令的 parseResponse() 方法。在这个例子中，我们仅仅需要寻找一个状态属性，因此响应看起来也许像是这样：

```
<commands>  
  
    <command id='001_diameter' status='ok'/>  
  
    <command id='003_albedo' status='failed' message='value out of range'/>  
  
    <command id='004_hairFTEL' status='failed' message='invalid property  
name' />  
  
</commands>
```

水星的直径已经更新，但是另外两个更新失败了，并且给出了每个失败的原因。我们已经通知用户出现的问题（相当基础的方式——使用 alert() 函数），然后他们可以采取补救的行动。

处理这些请求的服务器端组件需要能够将请求数据拆分成命令，并将每个命令分配给合适的处理对象进行处理。一旦每个命令都处理完毕，结果会写回到 HTTP 响应中。处理这个任务的一个简单的 Java Servlet 实现显示在代码清单 5-15 中。

代码清单 5-15 CommandServlet.java

```
public class CommandServlet extends HttpServlet {  
    private Map commandTypes=null;  
  
    public void init() throws ServletException {  
        ServletConfig config=getServletConfig();  
        commandTypes=new HashMap(); ① 启动时配置处理对象  
        boolean more=true;  
        for(int counter=1;more;counter++){  
            String typeName=config.getInitParameter("type"+counter);  
            String typeImpl=config.getInitParameter("impl"+counter);  
            if (typeName==null || typeImpl==null){  
                more=false;  
            }else{  
                try{  
                    Class cls=Class.forName(typeImpl);  
                    commandTypes.put(typeName,cls);  
                }catch (ClassNotFoundException clanfex){  
                    this.log(  
                        "couldn't resolve handler class name "  
                        +typeImpl);  
                }  
            }  
        }  
    }  
  
    protected void doPost(  
        HttpServletRequest req,  
        HttpServletResponse resp  
    ) throws IOException{  
        resp.setContentType("text/xml"); ② 处理请求  
        Reader reader=req.getReader();  
        Writer writer=resp.getWriter();  
        try{  
            SAXBuilder builder=new SAXBuilder(false);  
            Document doc=builder.build(reader); ③ 处理 XML 数据  
            Element root=doc.getRootElement();  
            if ("commands".equals(root.getName())){  
                for(Iterator iter=root.getChildren("command").iterator();  
                    iter.hasNext();){  
                    Element el=(Element)iter.next();  
                    String type=el.getAttributeValue("type");  
                    XMLCommandProcessor command=getCommand(type,writer);  
                }  
            }  
        }catch (Exception e){  
            e.printStackTrace();  
        }  
    }  
}
```

```

        if (command!=null){
            Element result=command.processXML(el);    ④ 委托给处理对象
            writer.write(result.toString());
        }
    }else{
        sendError(writer,
                  "incorrect document format - "
                  +"expected top-level command tag");
    }
}catch (JDOMException jdomex){
    sendError(writer,"unable to parse request document");
}
}

private XMLCommandProcessor getCommand
(String type,Writer writer)
throws IOException{          ⑤ 匹配处理对象和命令
    XMLCommandProcessor cmd=null;
    Class cls=(Class) (commandTypes.get(type));
    if (cls!=null){
        try{
            cmd=(XMLCommandProcessor) (cls.newInstance());
        }catch (ClassCastException castex){
            sendError(writer,
                      "class "+cls.getName()
                      +" is not a command");
        } catch (InstantiationException instex) {
            sendError(writer,
                      "not able to create class "+cls.getName());
        } catch (IllegalAccessException illex) {
            sendError(writer,
                      "not allowed to create class "+cls.getName());
        }
    }else{
        sendError(writer,"no command type registered for "+type);
    }
    return cmd;
}

private void sendError
(Writer writer,String message) throws IOException{
    writer.write("<error msg='"+message+"'/>");
    writer.flush();
}
}

```

这个 servlet 维护了一个 XMLCommandProcessor 对象的映射，这些对象是通过 Servlet Config 接口①来配置的。一个更为成熟的框架也许会自己提供 XML 配置文件。当处理一个接收到的 POST 请求时②，我们使用 JDOM 来解析 XML 数据③，然后遍历<command> 标签，将类型属性与 XMLCommandProcessor 进行匹配④。这个映射存有类的定义，在 getCommand() 方法中使用反射由该定义创建动态实例⑤。

XMLCommandProcessor 接口仅仅由单个方法组成：

```

public interface XMLCommandProcessor
{
    Element processXML(Element el);
}

```

这个接口依赖 JDOM 库，使用 Element 对象同时作为参数和返回类型¹，从而获得 XML 的方便的基于对象的表现。这个接口用来更新行星数据的简单实现显示在代码清单 5-16 中。

代码清单 5-16 PlanetUpdateCommandProcessor.java

```
public class PlanetUpdateCommandProcessor
    implements XMLCommandProcessor {

    public Element processXML(Element el) {
        Element result=new Element("command");
        String id=el.getAttributeValue("id");
        result.setAttribute("id",id);
        String status=null;
        String reason=null;
        String planetId=el.getAttributeValue("planetId");
        String field=el.getAttributeValue("field");
        String value=el.getAttributeValue("value");
        Planet planet=findPlanet(planetId); ❷ 访问领域模型
        if (planet==null){
            status="failed";
            reason="no planet found for id "+planetId;
        }else{
            Double numValue=new Double(value);
            Object[] args=new Object[]{ numValue };
            String method = "set"+field.substring(0,1).toUpperCase()
                +field.substring(1);
            Statement statement=new Statement(planet,method,args);
            try {
                statement.execute(); ❸ 更新领域模型
                status="ok";
            } catch (Exception e) {
                status="failed";
                reason="unable to set value "+value+" for field "+field;
            }
        }
        result.setAttribute("status",status);
        if (reason!=null){
            result.setAttribute("reason",reason);
        }
        return result;
    }

    private Planet findPlanet(String planetId) {
        // TODO use hibernate ❹ 为领域模型使用 ORM
        return null;
    }
}
```

我们不但使用 JDOM 来解析接收到的 XML，在这里也使用它来生成 XML，创建了一个根节点❶，其子节点在 `processXML()` 方法中使用编程方式来创建。一旦我们得到了唯一的 ID，就可以使用 `findPlanet()` 方法❷ 来访问服务器端的领域模型。为了简洁起见，`findPlanet()` 方法的实现在这里并没有给出——通常会使用像 Hibernate 的 ORM 工具以后台方式与数据库对话❹¹。

我们使用反射来更新领域模型❸，然后返回已经构建的 JDOM 对象，它将由 `servlet` 来序列化。

这些内容为基于队列的架构的完整生命周期提供了一个草图，这个架构将很多小的领域模型的更新组合成单个的 HTTP 事务。它将执行客户端和服务器的领域模型之间细粒度同步的能力和有效地管理服务器通信流量的需求结合了起来。正如在 5.3 节提到的，它可以为 JSF 和类似的框架提供一种解决方案，在这些框架中，用户界面的结构和交互模型被紧紧地绑在了服务器上。我们在这里仅仅提供了一种有效的方式来跨层修改领域模型。

这样就结束了本章对于客户/服务器通信技术和纵览 **Ajax** 应用中关键设计问题的旅程。沿着这条路，我们已经从起点开始发展出了一套 **Ajax** 服务器请求的模式语言，更好地理解了用来实现这些模式的可供选择的技术。

5.6 小结

我们通过考察在 **Ajax** 中应用服务器的关键角色（即交付客户端代码到浏览器，在它一旦运行后为客户端提供数据）来开始本章。我们考察了服务器端通常的实现语言，并且考察了目前服务器端框架的常用类型。这些框架主要是为传统的 **Web** 应用服务设计的，我们考虑了它们如何来适应 **Ajax**。服务器端框架的空间拥挤且变化迅速，我们按照通用的架构对它们进行分类，而不是考察特定的产品。这将范围缩小到了三个主要的方法：**Model2** 框架、基于组件的框架、面向服务的架构。**SOA** 似乎天生最容易适应 **Ajax**，虽然其他二者也可以在不同程度上成功地适应 **Ajax**。我们还通过引入 **Façade** 来考察了在 **SOA** 中如何更好地分离关注点。

深入到细节中，我们对使用三种方法从服务器取回数据进行了对比，将它们分别标记为以内容为中心、以脚本为中心和以数据为中心。它们形成了一个连续统一体，传统的 **Web** 应用严重倾向于以内容为中心的风格，而 **Ajax** 应用则倾向于以数据为中心的风格。在讨论以数据为中心的方法中，我们发现除了 **XML** 还可以使用其他的数据格式，并且考察了使用 **JSON** 作为向客户端传输数据的方法。

最后，我们描述了发送更新到服务器的两种方法：使用 **HTML** 表单和使用 **XMLHttpRequest** 对象。我们也通过使用客户端的 **Command** 对象队列，考虑了带宽管理的问题。这种技术通过减小了服务器的负担和网络的通信流量，获得了显著的性能提升。而且它与我们已经看到的 **SOA** 的最佳实践也是一致的，也就是从一种 **RPC** 风格的方法转向基于文档的通信策略。

本章总结了我们所覆盖到的 **Ajax** 核心技术。现在我们已经覆盖了所有的基础知识，并且接触到了相当多的高级话题。在接下来的三章，我们会回到可用性的主题，并且为了突出那些关键的问题（这些问题可以区分出聪明的编码和外行用户实际上想使用的方法），给这里所完成的技术巫术再添加一些光泽。

5.7 资源

我们在本章讨论过几个 **Web** 框架，它们的 URL 是：

Struts (<http://struts.apache.org>);

Tapestry (<http://jakarta.apache.org/tapestry/>);

JSF (<http://java.sun.com/j2ee/javaserverfaces/faq.html>)；**PHP-MVC** (www.phpmvc.net)。

单是 Java 就有超过 60 个 **Web** 框架，由 **Wicket** 项目的开发人员列出(<http://wicket.sourceforge.net/Introduction.html>)。

JSF 是涉及很多个别框架和产品的广泛分类。*JavaServer Faces in Action* (Manning, 2004) 的作者 Kito Mann 维护着 JSF 内容的权威门户站点：www.jsfcentral.com。Greg Murray 和 Sun 蓝图目录 (Sun's Blueprints catalog) 的同事在<https://bpcatalog.dev.java.net/nonav/ajax/jsf-ajax/frames.html> 讨论了 **Ajax** 和 **JSF**。**AjaxFaces** 是商业支持 **Ajax** 的 **JSF** 实现 (www.ajaxfaces.com)，Apache 的开源项目 **MyFaces** 也正在考察 **Ajax** (http://myfaces.apache.org/sandbox/inputSuggest_Ajax.html)。

在撰写本书的期间，微软的 **Atlas** 还处于开发阶段，但是预计在今年 (2005) 晚些时候会发布其早期版本。Scott Guthrie 是 **Atlas** 的项目经理。他的 blog 可以在这里找到：<http://weblogs.asp.net/scottgu/archive/2005/06/28/416185.aspx>。

你可以在 www.json-rpc.org/impl.xhtml 找到用于一系列编程语言的 **JSON-RPC** 库。

第6章 用户体验

本章内容

可用代码的关键特征

通用的通知功能

可重用的通知框架

在原处突出显示更新过的数据

1 章讨论的可用性是所有软件应用的重点。无论你将代码组织得多么好，或是应用的编写方法是何等高明，如果它的可用性不够，你就会给用户留下很坏的印象。这虽然有些不合理，但是生活就是这样。大多数人对爱因斯坦的印象是他不修边幅的外表和蓬乱的头发，而不是去理解他试图解开空间—时间本质所做的工作。抓住用户的第一印象，并且关注细节，这些真地很重要。

在第 2 章至第 5 章中，我们介绍了很多很酷的技术，并且使用 Ajax 完成了一些了不起的任务。本书后续章节的焦点将集中于如何灵活地、适应性极强地完成这些工作。到目前为止我们的例子都较为简单，只关注于一些技巧性的功能实现，现在我们需要回过头来重新评估所做的工作，将其转变成人们真正想使用的东西，而且这些东西他们可能每天要使用好几个小时。本章的一些主题将一步步地帮助你，使得你的 Ajax 应用可以真正满足真实世界的需要。

为了让用户感觉舒服，你能做的一件最重要的事就是时不时地以一致的方式通知他们后台到底发生了些什么。这些不是可用性的全部内容，但是我们在本章中将会重点讨论这些内容，看一看深入而一致的通知风格是如何使整个应用受益的。大多数的 Ajax 应用也需要在一些时候通知用户，所以我们希望这些已完成的组件也能用于你自己的项目中。

在本章中我们开发了几种方法，这些方法能够在不打扰用户工作的同时，让用户明白程序正在做什么。在介绍这些内容之前，先快速了解一下质量的含义以及如何获得高质量。

6.1 做正确的事：开发高质量的应用

可用性是 Ajax 特别关注的热点，因为 Web 应用的用户通常是薄情的。应用可以零成本地下载并且运行的后果就是，用户一旦不再喜欢这个应用，就会将它弃之如履，因为他们可以毫不费力地接着使用 Google 所提供的八百亿个网页中的下一个。复杂性的进一步影响是，使用 Ajax 我们可以看到两种不同的可用性传统结合到一起了，那就是桌面应用和网页。两者的正确结合是不容易的，如果结合得不好甚至会适得其反。

在第 1 章中，我们从用户的观点来看待可用性。在应用中他们需要什么？可以忍受什么？我们将这个问题反过来，问问自己为了达到可用性的目标，我们的代码需要具有什么样的质量。以此为起点，可以分析出为了使应用很好地工作，我们该如何做。下面的章节详细介绍了使应用具有高质量的一些关键特征。

6.1.1 响应性

最让计算机用户感到沮丧的是，计算机无法跟上他的思路，使他的工作不得不暂时中断。一些基本的设计错误会使用户无法跟踪他的工作，因为他所使用的笨拙的计算机硬件使他在操作所关注的领域模型时产生了思维间隙，比如说在写一个很长的配置文件时整个用户界面却被锁住了。

提到响应时，充分理解你的目标用户，并且了解其系统的典型配置是非常重要的。就写配置文件这种情况来说，如果开发者使用配备了高速 7200 转/分钟的 SATA 硬盘的工作站时速度是可以接受的，但是在将文件写到一个拥挤的网络上或是写到 U 盘上时他们却可能会有另外一种不同的体验。特别是在开发 Web 应用时，一些相似的错误经常会犯，例如，只是在回送接口（loopback interface），上对应用程序进行测试，即 Web 服务器与浏览器运行在同一台开发机器上。这并不能得出对网络延迟的有用的评估结果，所有的 Web 应用应该在一个实际的 LAN 或是 WAN 环境中进行测试，或是使用一个通信流生成工具来进行仿真测试。

除了网络方面的情况以外，客户端代码的性能也会对程序的响应性造成很大的影响。性能是一个大问题，我们将在第 7 章中进行深入的研究。

6.1.2 健壮性

应用如果能应付繁忙工作站上通常出现的情况就可以说是健壮的。但是它将如何处理网络消耗呢？如果一个糟糕的应用占用了 CPU 五分钟，你的应用程序还能继续运行吗？在我最近参加的一个项目中，我们为了测试应用程序的健壮性，在键盘上胡乱敲击了约 10 秒钟，并且用鼠标在页面上到处乱点。一种很残酷的测试，但是很有效，也很好玩。

这种测试能说明什么？至少，它能找出事件处理代码的不足之处。键盘的敲击、鼠标的移动以及诸如此类的事件需要很快地响应，因为这些事件很容易频繁地发生。而且，它们也能找出组件间无意中产生的依赖。在 GUI 中会出现一种特定的情况，例如一个模式对话框会阻塞对主应用程序的访问，而一个打开的菜单项会阻塞对模式对话框的访问。如果这种情况只在按精确的时间顺序打开对话框和菜单项时才会出现，那可能需要一个用户花上两个月的工作时间才会发现这个问题。可是一旦应用程序发布到一个有数千用户的环境中，那问题可能会在几小时内就会出现，而且仅仅根据现场报告将很难重现这种情况。事先确定问题，并且将其解决，这可以提高应用的整体健壮性。

除了应付随机敲击键盘这样的情况以外，健壮性还有更多的含义。观察一个非开发人员如何使用某个应用也是非常有益的。这种情况下，一个对程序完全不了解的人会对程序的可用性设计提供有用的信息，然而让一个对相关领域的知识甚至产品有深入了解的人通过测试驱动的方式开发一些新的功能也是有益的。当一个程序的作者自己运行这个程序时，他会“看见”隐藏在程序背后中的代码，并且潜意识地避免执行一些特定的操作组合，或是在特定条件下的某些操作。最终用户是不会了解到这些的，而且也不会有开发者坐在你的旁边（除非你们在做结对编程）。换一个人以非常规的方式完整地运行你的应用的工作流程会帮助你在早期建立起程序的健壮性。

6.1.3 一致性

我们介绍过，Ajax 的可用性模式（usability pattern）是混合了桌面应用与 Web 浏览器的使用习惯的，并且仍然在不断地进化着。一些 Ajax 工具，例如 Bindows、qooxdoo 和 Backbase，甚至提供刻意地设计成与桌面应用中的按钮、树和表格等外观相似的 UI 部件集。

随着Web的不断发展，Web管理员、可用性设计的大师和日常用户都在为如何提高Web应用的可用性而不断地寻求着更加理想的解决方案。在这种情况下，我们可以提供的最好的建议就是尽量使一切都保持一致。如果应用的一部分是以Web风格的单击来打开一个弹出窗口，而另一部分却需要在相似的图标上进行双击，那么你的用户很快就会糊涂了。如果你必须有一个“谈话猪”——来引导用户使用整个站点，那可千万别让它在半路上突然改变了语气、着装或是发型。

从代码库的观点来看，一致性与可重用性是息息相关的。如果你习惯于经常将一段功能代码从一个地方复制粘贴到另一个地方，例如，将一段找零请求按钮的呈现代码经多次复制后产生了 4 处拷贝，当需要修改按钮的呈现方式的时候，你只修改了 3 个地方，但是却漏掉了第 4 个地方，你的界面的一致性就会随着时间的增长而逐步恶化。如果只有一段按钮的呈现代码，并且每个人都使用它，那么应用的一致性就可以保持在较高的水平上。这个原则不仅仅适用于可视化的 UI 操作，也适用于界面中不大可视化的部分，例如网络超时或者对非法数据的响应等等。

6.1.4 简单性

最后，我们要强调一下简单性的重要。Ajax 使你可以做一些疯狂而有创造性的事情，有些可能还没有在 Web 页面上出现过。这些东西还没出现过的原因可能是使之成为可能的技术才刚刚出现，也有一种可能是有很好的理由不去这么做。一种菜单会像弹簧一样从页面中弹出，并且上下震荡直至渐渐减弱，这个功能开发起来很有趣，对偶尔路过、只用上五分钟后就会离开的用户来说，这个功能也许很好玩，但是对于每天要使用该程序好几个小时的用户来说，在一天的工作快结束时，这种功能可能就不会引起他的什么好感了。

在使用一个新的特征之前，总是有必要去问一下它是否确实能提高用户最终的使用体验。对于应用 Ajax 开发的很多情况来说，答案都是肯定的，开发者应该将精力集中在开发这些真正有益的功能上。

6.1.5 付诸实践

你的代码很有可能不具备我们刚才提到的所有特征。我的代码当然也没有全都具备这些特征。我们所提到的只是一些理念。如果你付出努力去实现这些理念，那么将来你在维护代码库时，就能得到丰厚的回报，并且在对已实现代码进行重构时可以使得这些质量继续保持下去。选择在什么地方投入精力是一种魔术，要想擅长于此道只有通过不断的实践。如果你对于重构不是很熟悉，可以先尝试一些小程序，并且逐步地取得突破。记住，重构是一个增量的过程，你可以小幅度地逐步改进代码的质量，而不用花几周的时间将代码大卸八块，抛弃很多原有的代码，重新编写。

在本章的最后，我们将了解可以在 Ajax 应用中实现的几个特征。本章很大的篇幅集中介绍了一些通知框架，介绍如何在计算、网络请求之类的事件发生时，使用户可以了解后台正在做些什么。通过显示给用户一些视觉信息告诉他们这些工作正在进行，我们就提高了应用程序的响应性。通过使用统一的框架来实现诸如此类的通知，我们就能使这些通知看起来是一致的，而且使用户很容易理解通知的内容，因为所有的东西在使用时看起来都差不多。

让我们考察一下几种不同的方法，这些方法都可以用于在某些应用事件发生时通知用户。

6.2 让用户知情

在 Ajax 应用中，我们经常需要来回穿梭于网络，从服务器上获取一些资源，然后将结果传递给一个回调函数，在这个函数中再对结果进行操作。如果我们是以同步的方式来处理发到服务器的请求，那么在用户界面中处理这些问题还比较简单。首先对请求进行初始化，然后整个用户界面将会被锁住并且停止响应，直至服务器将结果传送回来，页面自身会被更新并且开始响应用户的输入。这种方式对开发者是很有

利的，但是对于用户来说却是令人恶心的。当然，我们可以利用异步请求的机制，但是这也使得通过用户界面响应服务器的更新变得复杂多了。

6.2.1 处理自己请求的响应

让我们选一个现成的例子开始工作。在第 5 章中开发的行星信息应用可以让我们对行星的一些可编辑属性进行更新：行星的直径和与太阳的距离（参见 5.5 节）。这些更新被发送到服务器，随后服务器响应，告知更新是被接受了或是被拒绝了。通过 5.5.3 节中对命令队列概念的介绍，我们使每一个服务器响应都能对特定用户提交的几处更新进行处理。下面显示了一段 XML 响应的例子，该例子中带有一个成功的命令和一个失败的命令。

```
<commands>
  <command id='001_diameter' status='ok'/>
  <command id='003_albedo' status='failed'
    message='value out of range' />
</commands>
```

从用户的角度看，用户对属性进行编辑后，转而去处理其他的事情，例如编辑与行星相关的事实在列表，甚至会离开这个窗口。用户的注意力将不再停留在水星的直径上。与此同时，在后台程序中，更新了的数值被封装在一个 JavaScript 的 Command 对象中，该对象插入到一个待发消息队列中，并在随后被传输出去。Command 对象在被发送后，转移到“已发送”队列中，并且在响应返回时提取出来，同时提取的可能还有很多其他的更新。这时 Command 对象将负责对更新进行处理，并执行一些适当的操作。

让我们先回顾一下上次留下的代码，然后再对其进行重构。下面是在第 5 章中的 Command 对象中 parseResponse() 方法的实现。

```
planets.commands.UpdatePropertyCommand
  .parseResponse=function(docEl){
    var attrs=docEl.attributes;
    var status=attrs.getNamedItem("status").value;
    if (status!="ok"){
      var reason=attrs.getNamedItem("message").value;
      alert("failed to update "+this.field
        +" to "+this.value+"\n\n"+reason);
    }
}
```

这是很好的“概念验证”代码，适合于被重构为更加完善的实现。正如它所显示的，如果更新是成功的，便什么也不做。在数据传送到服务器之前本地的领域模型已经得到了更新，所以可以假设一切都可以与服务器端的领域模型保持同步。如果更新失败了，我们就弹出一条警告消息。使用警告消息对于开发者来说非常简单，但是却造成很差的可用性，这一点我们后面将会看到。

现在重新回到我们的用户那里，用户现在可能已经不再考虑水星的反射率问题了。突然用户看到弹出一个警告框提示“无法将反射率设置到 180，数据越界”，或者类似的通知。如果不从用户使用的上下文环境来考虑，这样做也许无伤大雅。我们还可以将错误消息改为更容易理解的形式：“无法更新水星的反射率，因为...”，但是这样做仍然会打断用户的工作流程，这也是我们在开始时决定采用异步消息处理的原因。

在实际的应用中，也许还会产生一些更为严重的问题。可编辑字段中可能会使用 `onblur` 事件来启动一个将数据送往服务器的过程。而 `onblur` 事件在文本输入字段每次失去焦点时都会发生，其中也包括突然弹出警告框的情况。这样，如果用户正在编辑另一个属性并且正在进行文本输入时，弹出一个警告框可能会导致未输入完成的数据被提交到服务器。这可能会把服务器的领域模型搞乱，或者是在相关的数据校验代码将其捕获后再弹出另外一个警告框。

因此我们需要一个比弹出警告框更为合理的解决方案。不久我们就会开发一个出来，但是首先还是进一步分析一下当我们在提交请求给服务器时，如果其他用户也同时在使用系统将会发生什么情况。

6.2.2 处理其他用户提交的更新

我们的行星信息应用程序允许多个用户同时登录，因此可以假设我们在编辑一些数据时，其他的用户可能也会做同样的工作。每个用户可能都会希望在其他用户进行一些更新操作时能或多或少地得到一些通知。大多数的 Ajax 应用都会处理多个浏览器同时操作一个领域模型的情况，因此这又是一个相对普遍的需求。

我们可以用以下的方式修改 XML 响应以及 Command 队列对象以适应这种情况。对于向服务器端模型的每次更新请求，我们都生成一个时间戳。我们对服务器端的处理更新的过程进行修改，使之也同时检查其他用户最近对领域模型所做的更新，并且将这些也附加到响应文档中，这样的文档可能会如下面所示：

```
<responses updateTime='1120512761877'>
```

```
<command id='001_diameter' status='ok'/>  
<command id='003_albedo' status='failed' message='value out of range' />  
<update planetId='002' fieldName='distance' value='0.76' user='jim' />  
</responses>
```

在<command>标签中的是在已发送队列中用 ID 进行标记的 Command 对象，与之并列的是一个<update>标签，在此例中显示的是金星距离太阳的距离已经被另一个用户 jam 设置为 0.76。我们还在顶级标签中加入了一个属性，关于它的用途我们稍后再介绍。

起初，我们的命令队列仅在有命令排队时才向服务器端发送请求。现在我们需要对其进行修改，使得为了获取更新，即使在队列为空时也向服务器端发送请求。这个功能的实现涉及好几处的代码。代码清单 6-1 显示的是已修改的 CommandQueue 对象，修改过的部分用粗体显示。

代码清单 6-1 CommandQueue 对象

```
net.cmdQueues=new Array();    ① 全局查找

net.CommandQueue=function(id,url,onUpdate,freq){
    this.id=id;
    net.cmdQueues[id]=this;
    this.url=url;
    this.queued=new Array();
    this.sent=new Array();
    this.onUpdate=onUpdate;
    if (freq){
        this.repeat(freq);
    }
    this.lastUpdateTime=0;
}

net.CommandQueue.prototype.fireRequest=function(){
    if (!this.onUpdate && this.queued.length==0){
        return;
    }
    var data="lastUpdate="+this.lastUpdateTime+"&data=";
    ← 时间戳请求
    for(var i=0;i<this.queued.length;i++){
        var cmd=this.queued[i];
        if (this.isCommand(cmd)){
            data+=cmd.toRequestString();
            this.sent[cmd.id]=cmd;
        }
    }
    this.queued=new Array();
    this.loader=new net.ContentLoader(
        this.url,
        net.CommandQueue.onload,net.CommandQueue.onerror,
        "POST",data
    );
}

net.CommandQueue.onload=function(loader){
    var xmlDoc=net.req.responseXML;
    var elDocRoot=xmlDoc.getElementsByTagName("responses")[0];
    var lastUpdate=elDocRoot.attributes.getNamedItem("updateTime");
    if (parseInt(lastUpdate)>this.lastUpdateTime){
        this.lastUpdateTime=lastUpdate;    ④ 已更新的时间戳
    }
    if (elDocRoot){
        for(i=0;i<elDocRoot.childNodes.length;i++){
            elChild=elDocRoot.childNodes[i];
        }
    }
}
```

```

        if (elChild.nodeName=="command"){
            var attrs=elChild.attributes;
            var id=attrs.getNamedItem("id").value;
            var command=net.commandQueue.sent[id];
            if (command){
                command.parseResponse(elChild);
            }
        }else if (elChild.nodeName=="update"){
            if (this.implementsFunc("onUpdate")){
                this.onUpdate.call(this,elChild);    ⑤ 已更新的处理函数
            }
        }
    }
}

net.CommandQueue.prototype.repeat=function(freq){    ⑥ 服务器轮询标志
    this.unrepeat();
    if (freq>0){
        this.freq=freq;
        var cmd="net.cmdQueues["+this.id+"].fireRequest();";
        this.repeater=setInterval(cmd,freq*1000);
    }
}

net.CommandQueue.prototype.unrepeat=function(){    ⑦ 切换轮询状态
    if (this.repeater){
        clearInterval(this.repeater);
    }
    this.repeater=null;
}

```

已更新的处理函数

在这里我们添加了不少新的功能，让我们逐一讨论一下。

首先，我们引入了一个对所有命令队列对象的全局查找表①。由于受到 setInterval() 方法的限制，这是一个有必要保留的坏味道，我们将在稍后讨论。命令队列的构造函数将唯一的 ID 作为参数，并以此 ID 为键值将自己注册到全局查找表中。

命令队列的构造函数还使用另外两个新的参数②。其中 onUpdate 是一个函数对象，用来处理在响应的 XML 文档中介绍过的<update>标签。而 freq 是一个数值，用来指示两次对服务器的更新请求之间相隔的秒数。如果设置了这个值，构造函数就会为 repeat() 函数初始化一个调用⑥，它利用 JavaScript 内置的 setInterval() 方法来周期性地执行一段代码。setInterval() 方法以及它的堂兄弟 setTimeout() 在 IE 中只接受字符串作为其参数，因此直接将变量的引用传入代码中执行是不可能的。我们在 repeat() 方法中使用全局查找表变量和队列的唯一 ID 来解决这一问题。我们同时也保存了一个重复的 interval 对象的引用，因此我们可以在 unrepeat() 方法中使用 clearInterval() 方法来中止它⑦。

在 fireRequest() 方法中，我们原先在待传输的命令队列为空时直接退出。现在这种做法已经修改为，如果 onUpdate 处理函数被设置后，我们仍将继续发送一个空的队列以取回正在服务器端等待发送的<update>标签。在传送自己编辑过的数据的同时，我们还发送一个时间戳，以告诉服务器我们上次收到更新的时间❸，这样它就可以找出需要传送给我们的更新通知了。时间戳存储为命令队列的一个属性，并且在初始化时设置为 0。

我们将时间戳以 UNIX 风格的日期格式传递，即以 1970 年 1 月 1 日为起点经过的毫秒数。这样选择时间戳的形式是出于通用性考虑的。如果选择一种更易读的日期形式，我们将会遇到本地化的问题，以及处理不同平台和不同语言的默认格式不同的问题，等等。正确处理本地化问题是 Ajax 应用的一个重要话题。因为如果应用程序是在公共的因特网或是大型组织的 WAN 上运行的话，它将要面对的是世界各地的用户。

在 onload() 函数中，我们加了一段代码来计算在响应返回时最近更新的时间戳❹，并且解析<update>标签❺。而 onUpdate 事件处理函数被调用时是将命令队列作为其上下文对象的，并且<update>标签的 DOM 元素是其唯一的参数。

在我们以行星系统作为领域模型的例子中，更新处理函数如代码清单 6-2 所示。

代码清单 6-2 updatePlanets() 函数

```
function updatePlanets(updateTag) {
    var attrs=updateTag.attributes;
    var planetId=attrs.getNamedItem("planetId").value;
    var planet=solarSystem.planets[planetId];
    if (planet){
        var fld=attrs.getNamedItem("fieldName").value;
        var val=attrs.getNamedItem("value").value;
        if (planet.fld){
            planet[fld]=val;
        }else{
            alert('unknown planet attribute '+fld);
        }
    }else{
        alert('unknown planet id '+planetId);
    }
}
```

<update>标签中的属性告诉了我们在 JavaScript 层内更新领域模型时需要的所有信息。当然，从服务器来的数据可能并不是正确的，如果它真的不正确，我们就需要采取一些行动。在这种情况下，我们就回到了在 6.2.1 节中讨论的用弹出的警告框通知发生了错误的问题。

我们在处理其他用户的更新时在命令队列对象中加入了一些更巧妙的代码，包括在客户端与 Web 层之间传递时间戳，以及加入一个可插拔的更新处理函数。最终我们转了一圈又回到了当修改或是异步更新发生时如何通知用户的问题上了。在下一节中，我们将关注如何以更好的方式将信息告知用户，并且抛弃讨厌的 alert() 函数。

6.3 为 Ajax 设计通知系统

迄今为止，我们所依赖的 alert() 函数是 JavaScript 在早期还较为简单时候遗留下的原始方案，那时候的网页还大多是静态的并且后台活动的数量非常少。我们无法用任何方法通过 CSS 来控制它的显示效果，并且对于产品级的通知需求来说，最好使用建立 Ajax 用户界面的其他技术来构造通知机制，这也为程序提供了更大的灵活性。

如果纵观所有的计算机系统，可以看出通知消息可以有各种形式和尺寸，它们对用户的影响也有很大的不同。改变鼠标的光标（如 Window 系统中的沙漏和 Mac 系统中的弹跳球），或者使用辅助的图标来指示文件夹中的文件或其他项的状态。这些做法有点过分炫耀，实际上这些简单的暗示只能提供很少的信息。状态栏可以提供关于后台事件稍微详细的信息，而完整的对话框则可以提供比前面的方式更加详细的信息。图 6-1 展示了一系列在 UNIX 平台 KDE 桌面系统中可以使用的通知方法。



图 6-1 用户界面中提供状态信息的各种方法：修改图标以反映特殊的特性（这里是访问权限），提供总体信息的状态栏以及模式对话框。此处显示的是 UNIX 平台 KDE 桌面系统，类似的方法在大多数图形界面中也可见

称作 lost+found 的文件夹是当前用户不能访问的，所以在其上有一个加锁的辅助图片。在主窗口下方的状态栏在不打扰用户的情况下给出了用户正在查看的文件夹内容的更多信息。最后，错误通知窗口在用户试图打开加锁的文件夹时会给出一个强制性的通知并且要求用户立即反应。

与这些通知方式相比，`alert()`的使用显得非常特别而且丑陋。为了满足对健壮性、一致性和简单性的要求，开发一个可以在所有程序中使用的向用户提供通知信息的框架是非常有益的。在下面的章节中我们就来做这件事。

6.3.1 对通知建模

首先，我们来决定通知信息看起来应该是什么样子。它会为用户显示一段文字说明，并且可能会有一个图标与之一起显示。

当我们向用户通知后台所发生的活动时，一些消息会比其他的消息更为紧急。与其每次都去判断一个特定的消息是否应该显示，我们不如定义一个通用的优先级，并为每个消息都进行指定。

通常来说，我们会告诉用户一些他们可知可不知的消息。但另一些消息可能会比较重要，它们需要一直显示在那里直到用户取消它们，而有些可能只需要显示一小段时间。如果消息没有用户干预就消失了，那它可能是因为响应了一个回调函数，例如它是用来告诉用户某些后台过程（例如一个网络的下载过程）正在进行，而用户在取消这个通知前这个过程就结束了。还有一些其他的情况，例如一个新的调用开始时，我们只希望让消息持续一小段时间，随后它会自行消失。

针对这些需求，代码清单 6-3 表示了一个 `Message` 对象，它提供了一个向用户显示消息的通用的后台方法。我们建立了这种通知消息的模型后，还可以用各种方式修饰它，随后你就会看到。

代码清单 6-3 `Message` 对象

```

var msg=new Object();

msg.PRIORITY_LOW= { id:1, lifetime:30, icon:"img/msg_lo.png" };
msg.PRIORITY_DEFAULT={ id:2, lifetime:60, icon:"img/msg_def.png" };
msg.PRIORITY_HIGH= { id:3, lifetime:-1, icon:"img/msg_hi.png" };

msg.messages=new Array();

msg.Message=function(id,message,priority,lifetime,icon){
  this.id=id;
  msg.messages[id]=this;
  this.message=message;
  this.priority=(priority) ? priority : msg.PRIORITY_DEFAULT.id;
  this.lifetime=(lifetime) ? lifetime : this.defaultLifetime();
  this.icon=(icon) ? icon : this.defaultIcon();
  if (this.lifetime>0){
    this.fader=setTimeout(
      "msg.messages['"+this.id+"'].clear()", 
      this.lifetime*1000
    );
  }
}
msg.Message.prototype.clear=function(){
  msg.messages[this.id]=null;
}

msg.Message.prototype.defaultLifetime=function(){
  if (this.priority<=msg.PRIORITY_LOW.id){
    return msg.PRIORITY_LOW.lifetime;
  }else if (this.priority==msg.PRIORITY_DEFAULT.id){
    return msg.PRIORITY_DEFAULT.lifetime;
  }else if (this.priority>=msg.PRIORITY_HIGH.id){
    return msg.PRIORITY_HIGH.lifetime;
  }
}

msg.Message.prototype.defaultIcon=function(){
  if (this.priority<=msg.PRIORITY_LOW.id){
    return msg.PRIORITY_LOW.icon;
  }else if (this.priority==msg.PRIORITY_DEFAULT.id){
    return msg.PRIORITY_DEFAULT.icon;
  }else if (this.priority>=msg.PRIORITY_HIGH.id){
    return msg.PRIORITY_HIGH.icon;
  }
}

```

我们为通知系统定义了一个全局名字空间（namespace）对象 msg，其中有一个相关的数组，在其中任何消息都可以使用唯一的 ID 来进行查找。ID 的产生机制取决于应用程序使用该框架的方式。

我们定义了三个常量分别表示三个优先级——低、默认和高，消息可以指定为其中的一种。每个优先级都定义了一个默认的图标和生命期（以秒为单位），这些都可以利用构造函数的参数进行重载。高优先级消息的生命期设为-1，表示该消息不会自己消失，而要明确地被用户或是回调函数来取消。

6.3.2 定义用户界面需求

根据 MVC，我们在这里需要为通知系统提供一个模型。为了使之可以操作，我们还需要定义一个视图。可视化地显示通知的内容有很多种可选的方式。例如，可以选择状态栏类的方式，通知信息在上面使用小图标来表示，如图 6-2 所示。



图 6-2 通知系统使用的状态栏用户界面，消息用它们自身的图标来表示

☒ 图标是系统中用来通知低优先级信息的标准图标。状态栏上的第三个消息对象提供了自己的图标，一个带阴影的球体，它重载了默认的☒图标。每个加入到此状态栏中的通知信息都可以利用一个工具提示框进行查看，如图 6-3 所示。



图 6-3 状态栏的消息可以通过将鼠标放在图标之上来看，这时候会出现一个工具提示框

这种机制设计成不让人厌烦的形式。状态栏只占用屏幕空间的一小部分，当一个新的图标加上后，它就告诉用户一个新的通知信息已经到达了。对于更为紧急的消息，我们可能希望消息能尽早地显示。到目前为止，我们只在状态栏中显示低优先级的消息；而默认或是高优先级的消息是先在一个弹出对话框中显示，之后由用户来将其取消，如图 6-4 所示。

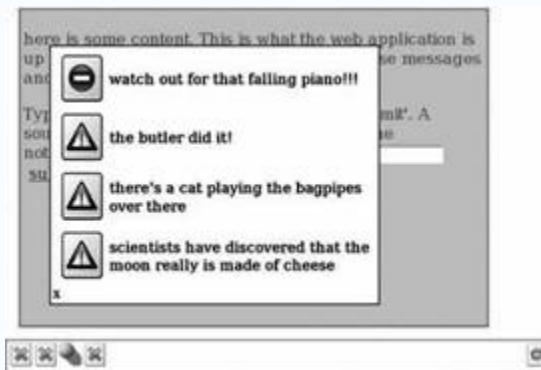


图 6-4 高优先级的消息显示在一个弹出对话框中，其中的消息按照优先级顺序来显示

对话框可以是模式的或是非模式的。如果使用模式对话框，我们采用一个半透明的层来阻塞用户使用用户界面的其余部分，直到用户将对话框取消。当对话框取消后，它用一个位于状态栏右侧的图标来表示。在下面两节中，我们将关注于如何实现这些功能。

6.4 实现通知框架

我们为用户界面定义了两个主要的部分：状态栏和弹出式对话框。让我们看看如何实现它们。一个完整的通知系统是比较复杂的，因此要将其分为几个阶段来描述。首先，我们要将 Message 对象加以完善，让它知道如何将自己显示在用户界面上，无论它是缩小在状态栏中还是完全显示的时候，也无论是以工具提示框的形式还是以弹出式对话框的形式。我们首先从状态栏组件的实现开始。

6.4.1 显示状态栏图标

状态栏需要将自己显示在屏幕上，而且还要在其中显示代表活动消息的图标。我们以一个图标为代表，来表示这些 Message 对象。这个 Message 有效地实现了一个小型的 MVC 模式，其显示功能代表的是视图，而它的交互功能充当一个控制器的角色。如果我们想为消息框架添加一个可选的显示机制时，将其设计成这种形式可能会带来一些问题。我们并不想这么做，而是想使一个应用中的所有通知信息都是一个样，这是出于一致性的考虑。代码清单 6-4 表示的是一个 Message 对象的显示方法。

显示消息

代码清单 6-4 带有用户界面的 Message 框架

用带有工具提示框的图标来显示

处理鼠标放在上面所触发的事件

```

msg.Message.prototype.render=function(el){          ①
  if (this.priority<=msg.PRIORITY_LOW.id){
    this.renderSmall(el);
  }else if (this.priority>=msg.PRIORITY_DEFAULT.id){
    this.renderFull(el);
  }
}

msg.Message.prototype.renderSmall=function(el){      ②
  this.icoTd=document.createElement("div");
  var ico=document.createElement("img");
  ico.src=this.icon;
  ico.className="msg_small_icon";
  this.icoTd.appendChild(ico);
  this.icoTd.messageObj=this;
  this.icoTd.onmouseover=msg.moverIconTooltip;
  this.icoTd.onmouseout=msg.moutIconTooltip;
  this.icoTd.onclick=msg.clickIconTooltip;

  el.appendChild(this.icoTd);
}

msg.moverIconTooltip=function(e){                  ③
  var event=e || window.event;
  var message=this.messageObj;
}

```

处理鼠标离开所触发的事

件

```

var popped=message.popped;
if (!popped){
  message.showPopup(event,false);
}
}

msg.moutIconTooltip=function(e){                  ④
  var message=this.messageObj;
  var popped=message.popped;
  var pinned=message.pinned;
  if (popped && !pinned){
    message.hidePopup();
  }
}

```

隐藏工具提示框

显示工具提示框

处理鼠标点击所触发的事

件

```

msg.clickIconTooltip=function(e){          ⑤
    var event=e || window.event;
    var message=this.messageObj;
    var popped=message.popped;
    var pinned=message.pinned;
    var expired=message.expired;
    if (popped && pinned){
        message.hidePopup();
        if (expired){
            message.unrender();
        }
    }else{
        message.showPopup(event,true);
    }
}
msg.Message.prototype.showPopup=function(event,pinned){ ⑥
    this.pinned=pinned;
    if (!this.popup){
        this.popup=document.createElement("div");
        this.popup.className='popup';
        this.renderFull(this.popup);
        document.body.appendChild(this.popup);
    }
    this.popup.style.display='block';
    var popX=event.clientX;
    var popY=event.clientY-xHeight(this.popup)-12;
    xMoveTo(this.popup,popX,popY);
    if (msg.popper && msg.popper!=this){
        msg.popper.hidePopup();
    }
    this.popped=true;
    msg.popper=this;
}
msg.Message.prototype.hidePopup=function(){  ←
    if (this.popped){
        if (this.popup){
            this.popup.style.display='none';
        }
        this.popped=false;
    }
}

```

我们介绍了一段顶层的显示 Message 对象的代码，以及它在状态栏中所显示的特定细节。首先我们来看看这些顶层代码。我们首先提供了一个 render() 方法①，它以一个 DOM 元素作为参数。根据消息的优先级，它会使用 renderSmall() ② 或 renderFull() 方法来处理。显示于状态栏中的消息总是低优先级的，它们被表示成一个图标并且当鼠标滑过时会显示相关的工具提示框。

renderSmall() 方法在 DOM 元素内部显示图标，并且为显示弹出式提示框提供事件处理函数。

本章主要是关于如何以更加专业的方式来完善 Ajax 应用，我们为显示图标而创建的工具提示框已经被完整地实现了，它包括三个事件处理函数。它将在鼠标移进图标时显示出来③，并且在它移出图标时消失④。如果图标被点击了，工具提示框就会固定住⑤，并且停留在原来的位置，直到图标被再次点击、消息过期或是选择了另一个工具提示框（在一个特定的时刻，只能看到一个工具提示框）。

6.4.2 显示详细的通知信息

用对话框显示的消息都是默认的或高优先级的，并且会同时显示出一个图标和旁边的一段消息（见图 6-4）。在这种形式的显示方法中我们仍然需要状态栏图标的工具提示框。当使用 `showPopup()` 方法显示工具提示框时，它将调用 `renderFull()` 方法显示消息的全部细节。该方法也可重用于将消息显示在对话框中。这种重用可以节省我们编写重复代码的时间，同时也增强了用户界面在视觉上的一致性。`renderFull()` 方法的实现如代码清单 6-5 所示。

代码清单 6-5 `renderFull()` 方法

```
msg.Message.prototype.renderFull=function(el){
    var inTable={el.tagName=="TBODY"};
    var topEl=null;
    this.row=document.createElement("tr");
    if (!inTable){
        topEl=document.createElement("table");
        var bod=document.createElement("tbody");
        topEl.appendChild(bod);
        bod.appendChild(this.row);
    }else{
        topEl=this.row;
    }

    var icoTd=document.createElement("td");
    icoTd.valign='center';
    this.row.appendChild(icoTd);
    var ico=document.createElement("img");
    ico.src=this.icon;
    icoTd.className="msg_large_icon";
    icoTd.appendChild(ico);

    var txtTd=document.createElement("td");
    txtTd.valign='top';

    txtTd.className="msg_text";
    this.row.appendChild(txtTd);
    txtTd.innerHTML=this.message;

    el.appendChild(topEl);
}
```

`renderSmall()` 方法为消息生成表格的一行。它会检查其所附加到的 DOM 元素中是否含有一个`<tbody>`标签，如果有就直接将自己加入其中，否则就创建必要的`<table>`或`<tbody>`标签。这样就可以将多个消息显示在主对话框中的同一个表格中，并且工具提示框对应的`<div>`标签也可以正确地显示。

请注意，这里使用 `innerHTML` 来将消息文本添加到用户界面中，而不是使用常用的 W3C DOM 方法。这样就可以在通知信息中使用 HTML 标记，使得消息具有更加丰富的表现形式，而不是只是产生一个文本节点。

6.4.3 集成

完成了以图标方式和完全方式来显示消息，我们就可以提供一个全面的 render() 方法用于各个消息的显示了。对话框和状态栏本身是由顶层的 render() 方法实现的，如代码清单 6-6 所示。

代码清单 6-6 msg.render() 函数

按照优先级将消息排序

保证状态栏存在

```
msg.render=function(msgbar){  
    if (!msgbar){  
        msgbar='msgbar';  
    }  
    msg.msgbarDiv=xGetElementById(msgbar);  
    if (!msg.msgbarDiv){  
        msg.msgbarDiv=msg.createBar(msgbar);  
    }  
    styling.removeAllChildren(msg.msgbarDiv);  
    var lows=new Array();  
    var meds=new Array();  
    var highs=new Array();  
    for (var i in msg.messages){  
        ②  
        var message=msg.messages[i];  
        if (message){  
            if (message.priority<=msg.PRIORITY_LOW.id){  
                lows.append (message);  
            }  
        }  
    }  
}
```

显示低优先级的消息

```
    }else if (message.priority==msg.PRIORITY_DEFAULT.id){  
        meds.append(message);  
    }else if (message.priority>=msg.PRIORITY_HIGH.id){  
        highs.append(message);  
    }  
}  
}  
for (var i=0;i<lows.length;i++){  
    ③  
    lows[i].render(msg.msgbarDiv);  
}
```

创建一个弹出对话框

创建一个状态栏

显示高优先级的消息

保证对话框存在

```

}
if (meds.length+highs.length>0){    ④
  msg.dialog=xGetElementById(msgbar+"_dialog");
  if (!msg.dialog){
    msg.dialog=msg.createDialog(
      msgbar+"_dialog",
      msg.msgbarDiv,
      (highs.length>0) );    ⑤
  }
  styling.removeAllChildren(msg.dialog.tbody);
  for (var i=0;i<highs.length;i++){
    highs[i].render(msg.dialog.tbody);
  }
  for (var i=0;i<meds.length;i++){
    meds[i].render(msg.dialog.tbody);
  }
  if (highs.length>0){
    msg.dialog.ico.src=msg.PRIORITY_HIGH.icon;
  }else{
    msg.dialog.ico.src=msg.PRIORITY_DEFAULT.icon;
  }
}
}

msg.createBar=function(id){    ⑥
  var msgbar=document.createElement("div");
  msgbar.className='msgbar';
  msgbar.id=id;
  var parentEl=document.body;
  parentEl.append(msgbar);
  return msgbar;
}

msg.createDialog=function(id,bar,isModal){    ⑦
  var dialog=document.createElement("div");
  dialog.className='dialog';
  dialog.id=id;
  var tbl=document.createElement("table");
  dialog.appendChild(tbl);
  dialog.tbody=document.createElement("tbody");
  tbl.appendChild(dialog.tbody);
}

```

如果需要添加一个模式层

```

var closeButton=document.createElement("div");
closeButton.dialog=dialog;
closeButton.onclick=msg.hideDialog;
var closeTxt=document.createTextNode("x");
closeButton.appendChild(closeTxt);
dialog.appendChild(closeButton);

if (isModal){    ⑧
  dialog.modalLayer=document.createElement("div");
  dialog.modalLayer.className='modal';
  dialog.modalLayer.appendChild(dialog);
  document.body.appendChild(dialog.modalLayer);
}

```

隐藏对话框

显示对话框

```

        }else{
            dialog.className+=' non-modal';
            document.body.appendChild(dialog);
        }

        dialog.ico=document.createElement("img");
        dialog.ico.className="msg_dialog_icon";
        dialog.ico.dialog=dialog;
        dialog.ico.onclick=msg.showDialog;
        bar.appendChild(dialog.ico);

        return dialog;
    }

    msg.hideDialog=function(e){  ←
        var dialog=(this.dialog) ? this.dialog : msg.dialog;
        if (dialog){
            if (dialog.modalLayer){
                dialog.modalLayer.style.display='none';
            }else{
                dialog.style.display='none';
            }
        }
    }

    msg.showDialog=function(e){  ←
        var dialog=(this.dialog) ? this.dialog : msg.dialog;
        if (dialog){
            if (dialog.modalLayer){
                dialog.modalLayer.style.display='block';
            }else{
                dialog.style.display='block';
            }
        }
    }
}

```

`render()`方法可以被多次地调用，它会检查一些常用的 UI 组件是否存在❶、❸，如果需要还会使用 `createDialog()` 方法❷ 和 `createBar()` 方法❶ 来创建它们。这些代码使用标准的 DOM 操作方法和事件处理函数（例如用来显示或隐藏对话框的那些方法）将 UI 组件组装起来。

为了显示所有的通知，系统先将它们按照优先级归类到三个临时数组中❷。随后低优先级的消息将显示在状态栏中❸，而其他消息显示在对话框中，优先级越高的消息越早处理❹。

为了实现一个模式对话框，我们只需要将一个可见的对话框嵌入到另一个占据了整个屏幕的 DIV 元素中，这会阻塞所有的鼠标事件，使它们无法被主用户界面接收到❺。这个模式 DIV 有一个背景图案，使用交替出现的白色和半透明像素来使得界面变成灰色，这样可以明确地表示出这个对话框是模式的。使用这种方式而不是使用 CSS 中的透明设置的原因是，以后我们还要使任何内嵌的元素（例如对话框本身）也变得透明。通知框架的这种效果使用代码清单 6-7 中所示的 CSS 文件来实现。

代码清单 6-7 msg.css

```
.msg_small_icon{
    height: 32px;
    width: 32px;
    position: relative;
    float: left;
}

.msg_dialog_icon{
    height: 32px;
    width: 32px;
    position: relative;
    float: right;
}

.msg_large_icon{
    height: 64px;
    width: 64px;
}

.msg_text{
    font-family: arial;
    font-weight: light;
    font-size: 14pt;
    color: blue;
}

.msgbar{
    position: relative;
    background-color: white;
    border: solid blue 1px;
    width: 100%;
    height: 38px;
    padding: 2px;
}

.dialog{
    position: absolute;
    background-color: white;
    border: solid blue 1px;
    width: 420px;
    top: 64px;
    left: 64px;
    padding: 4px;
}

.popup{
    position: absolute;
    background-color: white;
    border: solid blue 1px;
    padding: 4px;
}

.non-modal{

}

.modal{
    position: absolute;
    top: 0px;
    left: 0px;
    width: 100%;
    height: 100%;
    background-image: url(img/modal_overlay.gif);
}
```

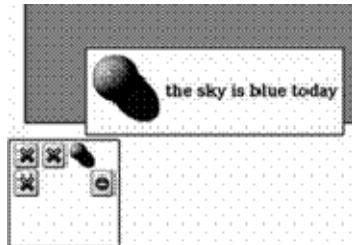


图 6-5 使用 CSS 的 float 属性允许一组图标自动适应不同形状的容器。我们在这里将状态栏的容器形状改变为一个正方形，图标和带阴影的球体图标自动改变它们的排列以适应新的区域，而启动关闭对话框的图标显示在区域的右侧

CSS 中的 float 属性在 msg_small_icon 和 msg_dialog_icon 中用来在状态栏中显示图标。其中 msg_small_icon 用来在工具提示框中显示低优先级的消息，并且将 float 属性设置为 left 使得它们向左对齐，而 msg_dialog_icon 将 float 属性设置为 right 使得启动关闭对话框的图标向右对齐。该框架可以将状态栏在各种形状各种大小的 DIV 元素中显示。浮动的元素将会使自己以合理的方式对齐，例如，如果需要的话，可以变成垂直对齐的状态栏，如图 6-5 所示。

最后，我们还需要修改一下用户界面中使用的 Message 对象。当一个个消息被创建后，它们有能力将自己加入到用户界面中，并且在消息过期后把自己从界面中移走。代码清单 6-8 显示了为实现这样的功能而需要做的一些修改。

代码清单 6-8 已修改的 Message 对象

```
var msg=new Object();

msg.PRIORITY_LOW= { id:1, lifetime:30, icon:"img/msg_lo.png" };
msg.PRIORITY_DEFAULT={ id:2, lifetime:60, icon:"img/msg_def.png" };
msg.PRIORITY_HIGH= { id:3, lifetime:-1, icon:"img/msg_hi.png" };

msg.messages=new Array();
msg.dialog=null;
msg.msgBarDiv=null;
msg.suppressRender=false;
msg.Message=function(id,message,priority,lifetime,icon){
    this.id=id;
    msg.messages[id]=this;
    this.message=message;
    this.priority=(priority) ? priority : msg.PRIORITY_DEFAULT.id;
    this.lifetime=(lifetime) ? lifetime : this.defaultLifetime();
    this.icon=(icon) ? icon : this.defaultIcon();
    if (this.lifetime>0){
        this.fader=setTimeout(
            function() {
                msg.messages[id].clear();
                msg.messages[id].lifetime=0;
            }, this.lifetime*1000
        );
    }
    if (!msg.suppressRender){ ①
        this.attachToBar();
    }
}

msg.Message.prototype.attachToBar=function(){ ②
    if (!msg.msgbarDiv){
        msg.render();
    }
}
```

```

}else if (this.priority==msg.PRIORITY_LOW.id){
    this.render(msg.msgbarDiv);
}else{
    if (!msg.dialog){
        msg.dialog=msg.createDialog(
            msg.msgbarDiv.id+"_dialog",
            msg.msgbarDiv,
            (this.priority==msg.PRIORITY_HIGH.id)
        );
    }
    this.render(msg.dialog.tbod);
    msg.showDialog();
}
}

msg.Message.prototype.clear=function(){
    msg.messages[this.id]=null;
    if (this.row){
        this.row.style.display='none';
        this.row.messageObj=null;
        this.row=null;
    }
    if (this.icoTd){
        this.icoTd.style.display='none';
        this.icoTd.messageObj=null;
        this.icoTd=null;
    }
}

```

我们希望这个框架可以让开发者很容易地使用，所以当一个消息创建后，我们自动将消息附加到用户界面上①。只需要简单地调用构造函数就可以自动显示新消息。根据消息的优先级，它会自己选择是显示在状态栏中还是在对话框中②。当不希望每个消息都自动显示时，比如说需要一次添加几条消息，我们可以加一个标志阻止其自动显示。在这种情况下，可以在产生了大量的消息后，手动地调用 `msg.render()` 方法。

同样的道理，当在 `clear()` 函数中清除一条消息时，我们自动地消除所有的用户界面元素，这样消息就会完全消失③。

现在我们已经有了一个实用的框架用于将消息显示给用户。我们可以用手动的方式触发它，也可以在其他可重用的代码组件中利用它。在下一节中，我们将展示如何将其与 ContentLoader 对象挂钩，用来报告网络下载过程完成的进度。

6.5 使用通知框架处理网络请求

第 5 章中介绍了作为网络通信流通用封装形式的 ContentLoader 对象。可以使用通知框架为我们自动提交的任何数据请求添加状态报告功能。让我们首先仔细研究一下需求。

当向服务器提交请求时，我们希望以低优先级的通知消息表示出这个过程正在进行中。为了使网络请求与其他低优先级的通知有所区别，我们希望使用一个不同的图标。在第 4 章和第 5 章中介绍的行星信息应用中有一个地球的图片，让我们就用这个作为图标吧。

当一次网络请求结束后，我们希望上述通知被清除掉，并且在一切正常时替换成另一个低优先级的通知，或是在有错误发生时替换为另一个中等优先级的通知。

为了实现上述功能，我们只需要在该请求的整个生命周期的适合时刻创建特定的 Message 对象，例如当请求发出时、请求结束时、错误发生时等等。已修改的 ContentLoader 对象的代码如代码清单 6-9 所示。

代码清单 6-9 带有通知功能的 ContentLoader 对象

```
net.ContentLoader=function( ... ){ ... };
net.ContentLoader.msgId=1;
net.ContentLoader.prototype={
    loadXMLDoc:function(url,method,params,contentType){
        if (!method){
            method="GET";
        }
        if (!contentType && method=="POST"){
            contentType='application/x-www-form-urlencoded';
        }
        if (window.XMLHttpRequest){
            this.req=new XMLHttpRequest();
        } else if (window.ActiveXObject){
            this.req=new ActiveXObject("Microsoft.XMLHTTP");
        }
        if (this.req){
            try{
                var loader=this;
                this.req.onreadystatechange=function(){
                    loader.onReadyState.call(loader);
                }
                this.req.open(method,url,true);
                if (contentType){
                    this.req.setRequestHeader('Content-Type', contentType);
                }
            }
            catch(e){
                alert("Error: "+e);
            }
        }
    }
}
```

通知请求已经发出

```
    this.notification=new msg.Message(
        "net00"+net.ContentLoader.msgId,
        "loading "+url,
        msg.PRIORITY_LOW.id,
        ①
    );
}
```

清除初始通知

通知发生了错误

```

        -1,
        "img/ball-earth.gif"
    );
    net.ContentLoader.msgId++;
    this.req.send(params);
}catch (err){
    this.onerror.call(this);
}
},
onReadyState:function(){
    var req=this.req;
    var ready=req.readyState;
    if (ready==net.READY_STATE_COMPLETE){
        var httpStatus=req.status;
        if (httpStatus==200 || httpStatus==0){
            this.onload.call(this);
            this.notification.clear(); ②
        }else{
            this.onerror.call(this);
        }
    }
},
defaultError:function(){ ③
    var msgTxt="error fetching data!"
    +"<ul><li>readyState:" +this.req.readyState
    +"<li>status: " +this.req.status
    +"<li>headers: " +this.req.getAllResponseHeaders()
    +"</ul>";
    if (this.notification){
        this.notification.clear();
    }
    this.notification=new msg.Message(
        "net_err00"+net.ContentLoader.msgId,
        msgTxt,msg.PRIORITY_DEFAULT.id
    );
    net.ContentLoader.msgId++;
}
};

```

当用 loadXMLDoc() 创建一个网络请求时，我们同时创建一个低优先级的通知并且为其附加一个到 ContentLoader 对象的引用①。我们将其生命周期设置为-1，因此通知不会自动地过期。

在 onReadyState() 方法中，如果一切正常，我们就清除通知②。当有错误发生时，我们调用 defaultError() 方法，它会产生一个自己的通知③。这个通知的消息使用 HTML 标记来表示，因而它可以创建表现能力比简单文本更加丰富的报告。

代码清单 6-10 展示了一个使用已修改的 ContentLoader 对象的例子。

代码清单 6-10 通知功能的例子页面

向服务器提交请求

通知资源被加载

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
<title>Notifications test</title>
<link rel=stylesheet type="text/css" href="msg.css"/>

<script type="text/javascript" src="x/x_core.js"></script>
<script type="text/javascript" src="extras-array.js"></script>
<script type="text/javascript" src="styling.js"></script>
<script type="text/javascript" src="msg.js"></script>
<script type="text/javascript" src="net_notify.js"></script>
<script type="text/javascript">

window.onload=function(){
    msg.render('msgbar');
}
var msgId=1;

function submitUrl(){
    var url=document.getElementById('urlbar').value;
    var loader=new net.ContentLoader(url,notifyLoaded); ①
}

function notifyLoaded(){
    var doneMsg=new msg.Message( ②
        "done00"+msgId,
        "loaded that resource you asked for: "+this.url,
        msg.PRIORITY_LOW.id
    );
    msgId++;
    msg.render('msgbar');
}

</script>
</head>

```

```

<body>
<div class='content'>
<p>here is some content. This is what the web
application is up to when not being bugged silly
by all these messages and notifications and stuff.
<p>Type in a URL in the box below (from the
same domain, see Chapter 7), and hit 'submit'.
A souped-up contentloader that understands the
notification system will be invoked.
<input id='urlbar' type='text'/>&nbsp;
<a href='javascript:submitUrl()'>submit</a>
</div>
<div id='msgbar' class='msgbar'></div>
</body>
</html>

```

这个页面（如图 6-6 和图 6-7 所示）呈现了一个简单的 HTML 表单，用户可以在其中键入 URL。点击了提交的链接后将试图去加载这个 URL ①，并且在加载成功后启动 notifyLoaded() 回调函数。这个 notifyLoaded() 函数实际上并不对返回的资源作什么处理，它只是通过创建另一个 Message 对象 ② 报告一下它已经获得了这些资源。

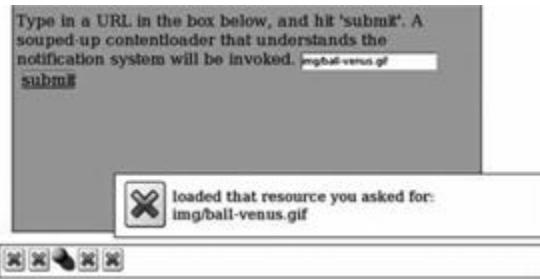


图 6-6 成功加载的资源将显示为状态栏中的一个工具提示框

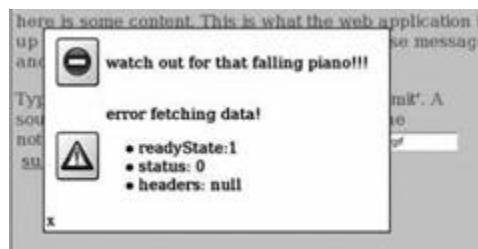


图 6-7 未成功的网络请求将会导致显示一个通知对话框（这里显示了两条消息，第二条消息是发生了网络错误的通知）

请注意，在一次成功的调用后所采取的行为并没有写进框架中，而只是提供了一个供用户完成的 onLoad 事件处理函数，这使得框架可以适用于各种不同的需求。在代码清单 6-9 中，我们以硬编码的方式来当错误发生时的情况编写需要采取的行动。在一个实际的应用中，并不是每一个调用失败都会严重到需要触发一个醒目的对话框。我们将此作为一个练习留给读者，请你们为 ContentLoader 对象添加一个参数，表示出当有错误发生时所需要给出通知的紧急程度（或是另外提供一个重载的 onError 事件处理函数，在其中可以采取温和一点的通知策略）。

至此这个通知框架的实现已经可以满足我们的需要了，并且还展示了它如何与已有的代码很好地配合工作。在下一节中，我们将考察另一种图形风格通知方法，它也可以很好地满足我们的需要。

6.6 表示数据的时效性

我们在前面几节中开发的通知框架提供了一系列用于显示系统活动信息的顶级组件。在某些情况下，我们可能希望显示一些与应用语境更为相关的信息，以在原来的位置表达出最近对某一段数据所做的更改。在这部分中，我们将扩展在第 4 章和第 5 章中开发的 ObjectViewer 的内容，使之可以提供关于最近数据更改的可视化通知。

6.6.1 定义简单的突出显示格式

让我们从一种简单的突出显示数据的方法开始，即使用反白显示的 (reverse video) 视觉效果。ObjectViewer 的用户界面总得来说比较苍白，主要使用蓝/白色调，因此深红色将会显得比较突出。我们首先要做的是定义一个代表最近更改过的数据的 CSS 类：

```
.new{
    background-color: #f0e0d0;
}
```

```
}
```

这里我们选择的是非常简单的样式。在一个更加完善的应用中，你可能希望采用更加复杂的样式。代码清单 6-11 显示了如何对 ObjectViewer 的代码进行修改，使得它可以将最近编辑的属性的样式设为 new，并且在一个特定的时间后自动地将这个样式去掉。

代码清单 6-11 带有最近编辑的通知样式的 ObjectViewer

```
objviewer.PropertyViewer.prototype.commitEdit=function(value){  
    if (this.type==objviewer.TYPE_SIMPLE){  
        this.value=value;  
    }
```

设置超时时长

将状态设置为 new

```
    var valDiv=this.renderSimple();  
    var td=this.valTd;  
    td.replaceChild(valDiv,td.firstChild);  
  
    this.viewer.notifyChange(this);  
    this.setStatus(objviewer.STATUS_NEW); ①  
}  
}  
  
objviewer.STATUS_NORMAL=1;  
objviewer.STATUS_NEW=2;  
  
objviewer.PropertyViewer.prototype.setStatus=function(status){  
    this.status=status;  
    if (this.fader){  
        clearTimeout(this.fader);  
    }  
    if (status==objviewer.STATUS_NORMAL){  
        this.valTd.className='objViewValue';  
    }else if (status==objviewer.STATUS_NEW){  
        this.valTd.className='objViewValue new';  
        var rnd="fade_"+new Date().getTime();  
        this.valTd.id=rnd;  
        this.valTd.fadee=this;  
        this.fader=setTimeout("objviewer.age('"+rnd+"')",5000); ②  
    }  
}
```

超时之后重置状态

```
objviewer.age=function(id){  
    var el=document.getElementById(id);  
    var viewer=el.fadee;  
    viewer.setStatus(objviewer.STATUS_NORMAL); ③  
    el.id="";  
    el.fadee=null;  
}
```

我们定义了两个状态：普通和新。当然也可以设置一个布尔值 isNew 来做这件事，但是我们选择这种方法是为了便于今后进一步扩充，例如为提交到服务器端的更新添加样式。我们在提交编辑过的数据时会

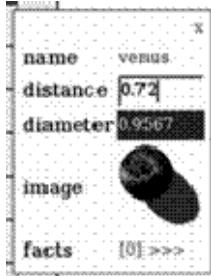


图 6-8 已修改的 ObjectViewer 显示了 `diameter` 属性最近编辑的值，采用彩色背景的样式。短暂的时间之后，其所代表的更新不再是最新的和需要注意的，这个样式将会消失。

调用 `setStatus()` 方法①。这样可以恰当地设置 CSS 类，并且在状态为“新”时可以设置一个计时器，在 5 秒后将状态设置为“普通”②（在实际应用中，我们可能需要将这个时间设置得更长，但是 5 秒对于测试和举例来说是合适的时间）。对象中包含一个对计时器的引用，如果在消息过期前状态发生了另一次改变，它会将计时器取消。

由于 JavaScript 的 `setTimeout()` 方法的局限性，我们为添加了样式的 DOM 节点设置了唯一的 ID，使得我们可以在计时器调用 `age()` 函数时再次找到这个节点③。这个 `age()` 函数也会取消掉 ID 和其他的临时引用。图 6-8 是一个显示了最近编辑值的 ObjectViewer。

用户的目光会被吸引到最近编辑的值，因为它的颜色与众不同。另一个吸引用户目光的办法是使用动画，在下一节中我们可以看到做这件事是多么简单。

6.6.2 用 Scriptaculous 效果库进行突出显示…

我们已经创建了一种简单的样式效果，主要是因为这种效果在静态的印刷书籍上易于显示。对于产品开发来说，我们推荐使用 Scriptaculous 库的 `Effect` 对象为在线通知添加动人的效果。在 3.5.2 节中简要地介绍了这个库，当时我们说过它可以只用一行代码的调用为 DOM 元素添加很多种不同的动画或渐变效果。

我们可以很容易地对代码清单 6-11 中的 `setStatus()` 方法进行改写以使用 Scriptaculous 库的 `Effect` 效果。例如使用 `Effects.Pulsate` 对象来为最近编辑的值添加跳动效果，这会使它们反复地淡入淡出，当然这会非常引人注目，但是不幸的是这种效果不能通过截屏的方式表达出来。代码清单 6-12 说明了要采用这种效果所需要对代码做的改动。

代码清单 6-12 使用 Scriptaculous 为最近编辑的值添加样式

```
objviewer.PropertyViewer.prototype.setStatus=function(status){
    this.status=status;
    if (this.effect){
        this.effect.cancel();
        this.effect=null;
    }
    if (status==objviewer.STATUS_NEW){
        this.effect=new Effect.Pulsate(
            this.valTd,
            {duration: 5.0}
        );
    }
}
```

这些 Effect 对象完成了一些琐碎的工作，我们不再需要去管理那些超时设置了。那个 age() 函数也可以被一起移走了。我们只需要引用 Pulsate 效果的构造函数，将一个引用传递给将要操作的 DOM 元素，如果需要重载一些选项的默认值，可以再传入一个关联数组。默认情况下，Pulsate 效果会持续 3 秒，我们将其改成了 5 秒，这样就与前面的例子一致了。这样做需要传递一个持续时间参数到选项数组中。

同样的样式技术也可以用于操作数据的其他事件中，例如从服务端发起的更新。为了避免不同效果间发生冲突，我们首先检查是否有某种效果正在运行，并且将其取消（在 1.1 版中，所有的 Scriptaculous 效果都会带有一个标准的 cancel() 函数）。

这一类的样式会在用户关注的位置为用户提供及时的回馈，而不像状态栏和对话框通知，它们更加适合于更为通用的信息。将它们组合在一起，这些视觉反馈可以极大地改善用户的使用体验。

6.7 小结

在本章中，我们考察了一系列改进 Ajax 应用的用户体验的话题。在开始阶段，我们定义了响应性、健壮性、一致性和简单性，并将它们作为高质量应用程序所应该具备的关键特征。

本章中大部分篇幅都在介绍如何在用户工作时提供反馈。在这个方面，我们分别实现了几种不同的视觉反馈机制，包括状态栏、弹出式对话框以及数据的突出显示。我们为添加这些功能所做的附加工作可以极大地丰富用户的使用体验，而将这些功能封装成可重用的框架又可以减轻开发者大量的负担。在完成了框架的开发之后，我们将几个先前实现过的代码作为例子，展示了如何简单方便地创建这些反馈。我们添加了状态栏通知，用来在向服务器发送请求时提供反馈；并且在用来查看太阳系行星数据的 ObjectBrowser 中突出显示最近更新的数据。

现在看来这些效果已经够眩了。在下面两章中，我们将考察在背后为应用的可用性提供支持的方面，也就是安全性和性能。

6.8 资源

Scriptaculous Effect 库可以在下面地址找到：<http://wiki.script.aculo.us/scriptaculous/list?category=Effects>。

例子中所使用的各种图标是从 David Vignoni 开发的 Nuvola 图标库中获得的（www.icon-king.com/）。

第7章 安全性与 Ajax

本章内容

JavaScript 安全模型

远程 Web 服务

保护因特网上的用户数据

保护 Ajax 数据流

安全性是因特网服务日益重要的关注点。Web 天生就是不安全的，为 Ajax 应用添加适当的安全性机制与为产品添加安全性机制大不相同。很明显，只要涉及用户的金钱，例如在线购物或者提供需要付款的服务，为应用提供适当的安全性就是必须考虑的基本要求。

安全性是一个大的话题，值得写一本专著。Ajax 应用遇到的很多安全问题和传统 Web 应用的都是一样的。出于这些原因，我们将讨论限定在那些对 Ajax 有特殊含义的与安全性相关的关注点上。首先，我们将考察在网络上传输可执行脚本的安全含义，以及浏览器厂商为了使这个过程安全而采取的步骤。我们也会考察在得到用户许可的情况下，放宽这些保护措施的步骤。其次，我们将考察当数据提交到服务器时如何保护用户的数据，让用户放心地使用 Ajax 服务。最后，我们将描述保护 Ajax 客户端所使用的数据服务的方式，以阻止网络上的外部实体非法使用这些数据服务。我们先来考察一下跨网络发送客户端应用的安全含义。

7.1 JavaScript 与浏览器安全性

当 Ajax 应用程序启动的时候，Web 服务器发送一组 JavaScript 指令给运行在另外一台机器上的 Web 浏览器，它们之间可能以前从未打过交道。浏览器继续执行这些指令。在允许 Web 浏览器做这件事上，Ajax 应用的用户对这个应用和它的作者表现出了很大的信任。浏览器厂商和标准团体意识到这样的信任并不总是恰当的，并且已经在适当的地方加入了保护，以免滥用。在本节中，我们将考察这些保护措施，以及

如何使用它们。然后，我们将讨论在什么情形下，这些限制是不恰当的，因此可以放宽。直接和第三方 Web 服务通信的能力就是这样一种情形，Ajax 开发者应该对此有着特殊的兴趣。

在深入讨论这个话题之前，我们先来定义一下移动代码（mobile code）的含义。电脑硬盘上的所有东西都不过只是一堆二进制数据。然而，我们可以将数据区分为纯粹的描述型数据和代表可执行的机器指令的数据。描述型数据本身什么也干不了，除非正在执行的某些过程使用它。在早期的客户/服务器应用中，客户端安装在用户的电脑上，就像任何其他的桌面应用一样。网上所有的数据流都是纯粹的描述型数据。然而，Ajax 应用的 JavaScript 代码是可执行代码。所以，与那些“死”的数据相比，它有潜力完成很多精彩的工作。它也可能会更加危险。只要代码是存放在一台机器上，其自身可以通过网络传输到另外一台机器上执行，我们就将这类代码描述为“移动”的。接收到移动代码的电脑需要判断是否应该信任代码的发送者（特别是来自公共因特网上的代码），以及它可以授权移动代码访问哪些系统资源？

7.1.1 引入来源服务器策略

我们刚才提到，当在 Web 浏览器中执行 JavaScript 代码的时候，用户允许与其素不相识的人编写的代码运行在自己的机器上。移动代码能够以这种方式自动跨越网络来运行，因此是一个潜在的安全隐患。为了解决移动代码的潜在危险，浏览器厂商在一个沙箱（sandbox）中执行 JavaScript 代码，沙箱是一个只能访问很少计算机资源的密闭环境。Ajax 应用不能读取或写入本地文件系统。大多数情况下，除了它自身所在的 Web 域，它也不能创建任何到其他 Web 域的网络连接。程序生成 IFrame 能够加载来自其他域的页面，并且运行页面中的代码，但是两个域的脚本不能互相交互。有时候这称作的“来源服务器”（server of origin）策略。

我们来做一个（非常）简单的例子。在第一个脚本文件中，我们定义了一个变量：

```
x=3;
```

在第二个脚本文件中，我们使用了那个变量：

```
alert(top.x+4);
```

第一个脚本包括在顶层文档中，它打开一个 IFrame，将包括第二个脚本的页面加载进来（图 7-1）。

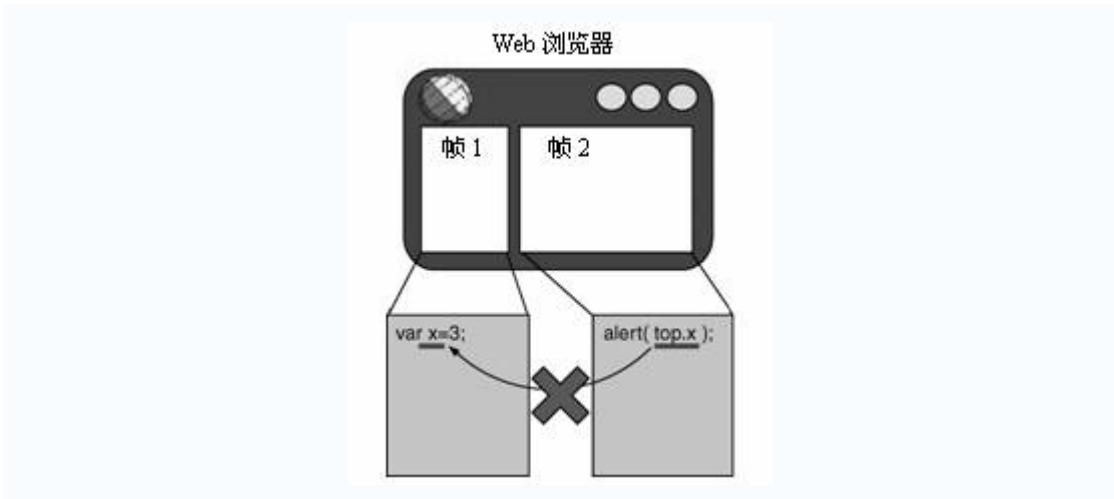


图 7-1 JavaScript 安全模型阻止了来自不同域的脚本互相交互

如果两个脚本是来自同一个域的话，会成功弹出一个警告框。如果不是的话，会抛出一个 JavaScript 错误，第二段脚本运行失败。

7.1.2 Ajax 的相关考虑

在第 5 章中讨论的以脚本为中心的交互中，从服务器上获取 JavaScript 代码，并且实时地执行。大多数情况下，客户端使用的是自己所在服务器上的代码，但是我们来考虑一下运行来自不同域的代码的情况，通常称为“脚本置换”（cross-scripting）。允许用户从任意网站下载脚本会带来很多潜在的危害，例如，使得第三方可以很方便地使用 DOM 操作方法来修改或者丑化你的网站。JavaScript 安全模型对于这样一类可能性提供了真正的保护。这个模型也阻止了这种情形，即恶意网站直接从你的网站下载 Ajax 客户端代码并且将其指向另外一个服务器，而你的用户并不知道他们正在和一个不同的后端进行通信。

在以数据为中心的交互中，服务器交付的是数据而不是可运行的代码，所以风险要稍微小一些。尽管如此，如果数据是来自第三方服务器的，那么数据中就可能包含有欺骗性的信息，当解析这些信息的时候会造成危害。例如，它可能会覆盖或删除重要信息，或者会造成服务器资源的大量消耗。

7.1.3 子域问题

最后，注意 Web 浏览器对于相同的域由什么组成只有相当有限的概念，可能会出现令人沮丧的错误。域仅仅是通过 URL 的首部来识别，而不去尝试判断相同的 IP 地址是否对应着两个域。表 7-1 展示了几个浏览器安全模型如何“思考”的例子。

表 7-1 跨浏览器安全策略的例子

URL	允许脚本置换	说 明
http://www.mysite.com/script1.js http://www.mysite.com/script2.js	是	和预期的一致！
http://www.mysite.com:8080/script1.js http://www.mysite.com/script2.js	否	端口号不匹配(script1 服务在 8080 端口)
http://www.mysite.com/script1.js https://www.mysite.com/script2.js	否	协议不匹配 (script2 使用了安全性 HTTP 协议)
http://www.mysite.com/script1.js http://192.168.0.1/script2.js	否	www.mysite.com 解析为 IP 地址 192.168.0.1，但是浏览器不会知道这个
http://www.mysite.com/script1.js http://scripts.mysite.com/script2.js	否	子域被视作为其他域
http://www.myisp.com/dave/script1.js http://www.myisp.com/eric/script2.js	是	尽管脚本来自不同人所有的网站，但是域是一样的
http://www.myisp.com/dave/script1.js http://www.mysite.com/script2.js	否	www.mysite.com 指向 www.myisp.com/dave，但是浏览器不会检查这个

在子域的情况下，通过设置 `document.domain` 属性，可以截掉域名所匹配的一部分内容。例如，在一个来自 `http://www.myisp.com/dave` 的脚本中，我们向在脚本中添加上一行：

```
document.domain='myisp.com';
```

这将允许这个脚本和来自子域 `http://dave.myisp.com/` 的脚本进行交互，假设那个脚本也设置了 `document.domain` 的值。然而，不可能将 `document.domain` 设置为任意值，例如 `www.google.com`。

7.1.4 跨浏览器安全性

如果不指出显著的跨浏览器的不一致性，我们的讨论还不够完善。IE的安全性是通过在一系列的“区域”（zone）之上设置或多或少受到限制的安全权限来实现的。默认情况下（至少对于 IE 6），在本地文

件系统上执行的文件有访问因特网上网站的权限，不需对用户进行提示。本地文件系统被认为是一个安全的区域。如果是从运行在本地的Web服务器上运行的话，同样的代码会触发一个安全性对话框（图 7-2）。



图 7-2 如果代码试图访问来自其自身的服务器之外的 Web 服务，IE 会显示一个安全性警

告对话框。如果用户同意完成这个交互，随后的交互将不会打断

编写复杂的 Ajax 应用，并且直接使用来自文件系统的虚拟数据来测试大部分功能是有可能的。在密集编码期间，将 Web 服务器从开发环境中取出，确实简化了开发环境的设置。然而，我们奉劝开发者，在测试访问因特网上的 Web 服务的代码时，除了在本地文件系统上测试，也要在本地 Web 服务器上测试。在 Mozilla 中，没有区域的概念，来自本地文件系统的 Web 应用和来自 Web 服务器的 Web 应用一样受到限制。然而，在 IE 中，在两种情况下代码运行在不同的安全性区域，表现出来的行为就大不相同。

以上总结了 Ajax 脚本必须接受的一些关键的限制。JavaScript 安全模型有点令人讨厌，但是它通常对我们是有利的。少了它，就无法提高公众对于 Ajax 所提供的丰富的因特网服务的信心，以至于 Ajax 只能用来做一些最微不足道的事情，而不能算作是切实可行的技术。

然而，也存在着一些合理的理由调用来自你所有域之外的域的脚本，例如当涉及 Web 服务的发布者的时候。我们将在下一节中看看对于这种情况，如何减轻安全性方面的顾虑。

7.2 使用远程服务进行通信

将安全性内建在 Web 浏览器中是明智之举，但是这样做也可能会令人沮丧。出于有效性的考虑，安全系统通常不信任任何人，但是某些时候出于合理的原因，你要访问在第三方服务器上的资源，并且你已经深思熟虑过做这件事的安全性含义。现在，我们理解了浏览器是如何应用安全性策略的，再来讨论一下放宽它的方法。我们将要考察的第一个方法需要开发额外的服务器端代码，第二个方法仅需要客户端就可以完成工作。

7.2.1 代理远程服务

因为“来源服务器”策略，Ajax 应用限制在从它自己的 Web 域获取全部数据。如果我们想要 Ajax 应用访问来自第三方网站的信息，一种解决方案是从我们自己的服务器而不是从客户端发送调用到远程服务器，然后再转送给客户端（图 7-3）。

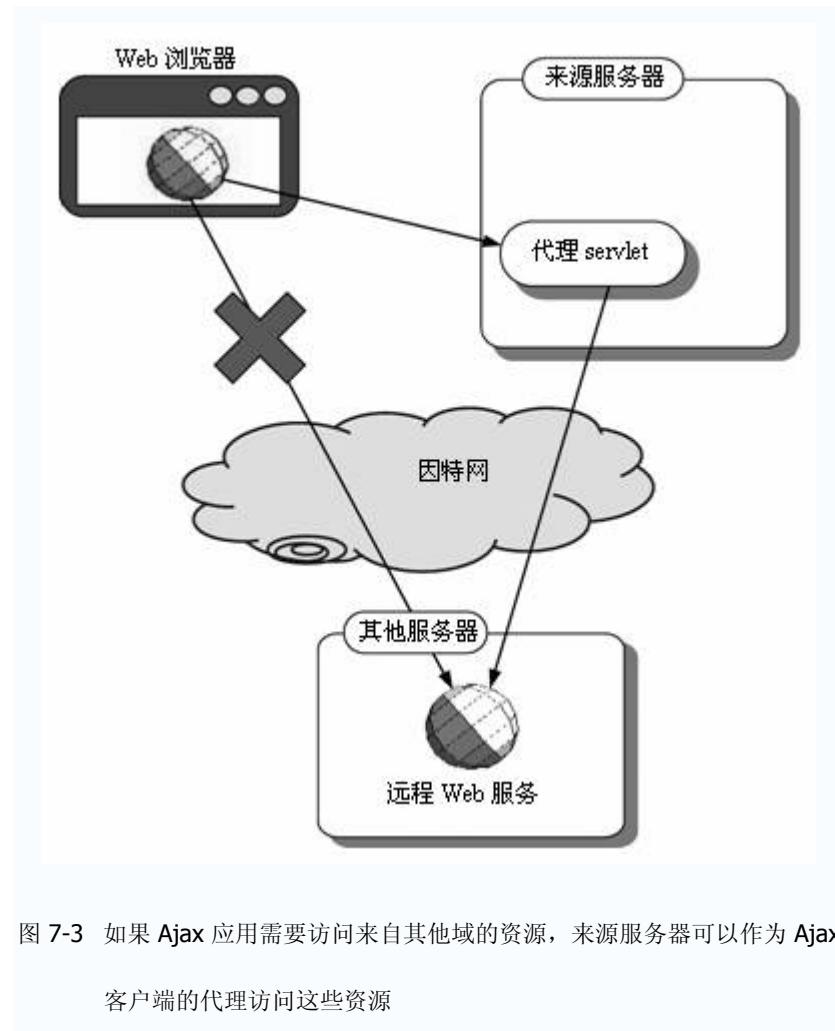


图 7-3 如果 Ajax 应用需要访问来自其他域的资源，来源服务器可以作为 Ajax 客户端的代理访问这些资源

按照这种方法，显示在浏览器中的数据来自本地服务器，这样就不会违反来源服务器策略。此外，所有的数据都受到服务器的详细审查，服务器有机会在将它们转送给客户端之前检查恶意的数据或代码。

不利的方面在于，这种方法增加了服务器的负载。我们将要考察的第二种解决方案是直接从浏览器访问第三方服务器。

7.2.2 使用 Web 服务

现在，很多机构都提供了供外部团体使用的 Web 服务，其中还包括了 JavaScript 客户端。Ajax 客户端想要直接与 Web 服务通信，来源服务器的安全策略在这里是个问题，但是 Ajax 客户端能够以编程方式请求权限来执行这些网络活动，从而克服这个问题。这个请求可以传递给用户，或者在得到用户的指示后，由浏览器记下来并且自动获得相应的权限。

在本节中，你将学会如何从 Ajax 客户端应用直接调用第三方的 Web 服务。IE 和 Mozilla Firefox 在处理这些请求上都有着自己的方式，我们来看看如何使用两者来满足我们的需求。

我们的例子程序将使用 SOAP（简单对象访问协议）来访问 Google 的一个 Web 服务。SOAP 是一个建造在 HTTP 之上的、基于 XML 的协议。SOAP 的基本原理是请求向服务器发送一个 XML 文档，其中描述了服务的参数；随后服务器响应一个 XML 文档，其中描述了结果。SOAP 发送的 XML 相当大，由首部信息和封装在“信封”中的内容组成。因为 SOAP 也使用了 XML，与 XMLHttpRequest 对象一起使用是很理想的。

Google 为它的搜索引擎提供了一个 SOAP 接口，用户可以在请求中发送搜索短语，随后得到一个列有结果页面的 XML 文档。这个 XML 响应和 Google 搜索结果页面中显示的数据非常相似，每个条目列有标题、片断、摘要和 URL。文档中也列出了这个短语的结果估计总数。

我们的应用是因特网时代的猜数字游戏，这是我们感兴趣的结果的估计数目。我们将要展示给用户的是一个简单的表单和一个随机生成的大数字（图 7-4）。用户必须输入一个短语，他们猜测将这个短语发送到 Google 之后，返回的搜索结果数目会在该随机数之上 1000 之内的范围内。

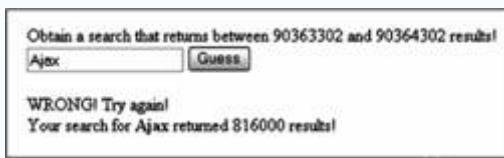


图 7-4 在一个简单的 Ajax 应用中以非常琐碎的方式使用 Google SOAP API。用户试图输

入一个短语，随后 Google 将会返回一个位于指定范围之内的结果的估计数目

我们将使用包装在第 3 章开发的 ContentLoader 对象中的 XMLHttpRequest 对象来访问 Google 的 SOAP 服务。我们最后在第 6 章中修改了这个对象，为它添加了一些通知能力。如果我们使用那个版本的 ContentLoad 来与 Google 通信，在 IE 中可以顺利完成任务，但是在 Mozilla 中则无法完成任务。我们来很快地浏览一下每一种浏览器的行为。

1. IE 和 Web 服务

我们已经提到，IE 的安全系统基于区域的概念。如果 Web 服务器提供了一个猜数字游戏的应用，即使它是运行在 localhost 上，默认情况下也被认为在某种程度上不大安全。当第一次使用 ContentLoader 访问 Google 的时候，我们收到一个像图 7-2 中描绘的通知消息。如果用户点击了 Yes，本次请求以及随后发送到这个服务器的任何请求，都将能够顺利完成。如果用户点击了 No，请求就取消了，并且调用 Content Loader 的错误处理函数。用户操作起来并不是很麻烦，而且还获得了适当级别的安全性。

记住，如果你正在本地文件系统上测试 Ajax 客户端的话，IE 会将这个应用看作是安全的，这样你就不会看到这个对话框。

Mozilla 浏览器，包括 Firefox，采取了更加严格的安全措施，因此处理起来更困难。我们下面来考察一下。

2. Mozilla 的 PrivilegeManager

Mozilla 浏览器的安全模型基于权限的概念。不同的活动，例如访问第三方的 Web 服务器、读写本地文件都视作是潜在不安全的。需要执行这些活动的应用代码必须请求相应的权限。权限由 netscape.security.PrivilegeManager 对象来分配。如果 Ajax 客户端想要访问 Google，它必须首先通过 PrivilegeManager 这一关。不幸的是，Firefox 可以配置为让 PrivilegeManager 不去监听代码发来的请求。对于来自 Web 服务器而不是来自本地文件系统的内容，这样的设置还是默认的设置。因此，下面的技术主要适合在 Internet 中使用。如果你正好处在这样一种情况，或者只是对 Firefox 如何工作有些好奇，请继续阅读下去。

为了请求一个权限，我们可以调用 enablePrivilege 方法。然后脚本会暂停，给用户显示一个对话框（图 7-5）。



图 7-5 在 Firefox 浏览器中请求额外的安全权限，会导致显示一个对话框，其中有标准化的警告信息

对话框解释了脚本将要做一些可能是不安全的事情。用户有机会给脚本授予或者拒绝授予相应的权限。在两种情况下，脚本随后都会恢复运行。如果授予了相应的权限，请求可以顺利完成；如果没有授予相应的权限，随后试图执行有权限要求的操作，通常都会导致一个脚本错误。

我们看到 IE 会自动记住用户的第一次决定，在出现第一次警告之后就不会再打扰用户了。Mozilla 授予的权限仅仅保持一段时间，与请求权限的函数的持续时间相同。除非用户点击了“记住我的决定”复选框，否则每次请求权限的时候，他们都会被对话框所打断（正如我们将会看到的，每个网络请求会弹出两次对话框）。在这里安全性和可用性似乎发生了争执。

IE 和 Mozilla 的其他不同之处在于，Mozilla 坚持在向用户显示对话框之前，必须在代码中提出明确的申请。我们再来看一下 ContentLoader 对象（见第 3、5、6 章），看看为了向 Google 提交请求，我们需要做些什么。修改过的代码中包含了向 PrivilegeManager 对象请求权限的代码，如代码清单 7-1 所示（我们也增加了自定义 HTTP 首部信息的能力，可以用来创建 SOAP 消息，下面我们会看到）。

代码清单 7-1 理解安全性的 ContentLoader 对象

```
net.ContentLoader=function(  
    url,onload,onerror,method,params,contentType,headers,secure  
) {  
    this.req=null;  
    this.onload=onload;  
    this.onerror=(onerror) ? onerror : this.defaultError;  
    this.secure=secure;  
    this.loadXMLDoc(url,method,params,contentType,headers);  
}
```

添加自定义的 HTTP 首部信息

申请发送请求的权限

```
net.ContentLoader.prototype={  
    loadXMLDoc:function(url,method,params,contentType,headers){  
        if (!method){  
            method="GET";  
        }  
        if (!contentType && method=="POST"){  
            contentType='application/x-www-form-urlencoded';  
        }  
        if (window.XMLHttpRequest){  
            this.req=new XMLHttpRequest();  
        } else if (window.ActiveXObject){  
            this.req=new ActiveXObject("Microsoft.XMLHTTP");  
        }  
        if (this.req){  
            try{  
                try{  
                    if (this.secure && netscape  
                        && netscape.security.PrivilegeManager.enablePrivilege) {  
                        netscape.security.PrivilegeManager  
                            .enablePrivilege('UniversalBrowserRead');  
                    }  
                } catch (err){}  
                this.req.open(method,url,true);  
                if (contentType){  
                    this.req.setRequestHeader('Content-Type', contentType);  
                }  
                if (headers){  
                    ②  
                    for (var h in headers){  
                        this.req.setRequestHeader(h,headers[h]);  
                    }  
                }  
                var loader=this;  
                this.req.onreadystatechange=function(){  
                    loader.onReadyState.call(loader);  
                }  
                this.req.send(params);  
            } catch (err){  
                this.onerror.call(this);  
            }  
        }  
    },  
  
    onReadyState:function(){  
        var req=this.req;  
        var ready=req.readyState;  
        if (ready==net.READY_STATE_COMPLETE){  
            var httpStatus=req.status;  
            if (httpStatus==200 || httpStatus==0){  
                try{  
                    if (this.secure && netscape  
                        && netscape.security.PrivilegeManager.enablePrivilege) {  
                        ①  
                    }  
                } catch (err){}  
            }  
        }  
    }  
},
```

申请解析响应的权限

```
        netscape.security.PrivilegeManager
            .enablePrivilege('UniversalBrowserRead');
    }
}catch (err){}
this.onload.call(this);
else{
    this.onerror.call(this);
}
},
defaultError:function(){
    alert("error fetching data!"
    +"\\n\\nreadyState:"+this.req.readyState
    +"\\nstatus: "+this.req.status
    +"\\nheaders: "+this.req.getAllResponseHeaders());
}
}
```

我们在构造函数上添加了两个新的参数。第一个参数是额外 HTTP 首部信息的数组❷，因为在构造 SOAP 请求期间，我们需要传入这些信息。第二个参数是个布尔型的标志，用来指示加载器在一些关键点上是否需要请求权限。

当我们向 `netscape.PrivilegeManager` 对象请求权限的时候，所授予的权限只能在当前函数的作用域内使用。所以，我们在两个点上都需要请求相应的权限：在向远程服务器发送请求的时候❶；在试图读取返回的响应的时候❷。我们在 `onReadyState` 函数的作用域中调用了自定义的 `onload` 处理函数，这样权限的持续时间就可以跨越任何自定义的逻辑。

IE 无法理解 `PrivilegeManager`，碰到这个对象时将会抛出异常。出于这个原因，我们简单地将引用这个对象的代码封装在 `try...catch` 块中，这样做可以捕获这个异常并且将它静悄悄地处理掉。当前面的代码在 IE 中运行的时候，它会进入 `try...catch` 块中的失败处理部分，自己恢复过来，继续运行而不会产生任何有害后果。在 Mozilla 中，可以与 `PrivilegeManager` 对象进行沟通，而不会抛出异常。

随后，我们来利用修改过的 `ContentLoader` 向 Google 发送一个请求。代码清单 7-2 展示了这个简单的猜数字游戏应用所需要的 HTML 文件。

代码清单 7-2 googleSoap.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
```

```

<head>
<title>Google Guessing</title>
<script type="text/javascript" src='net_secure.js'></script>
<script type="text/javascript" src='googleSoap.js'></script>
<script type="text/javascript">

    var googleKey=null;

    var guessRange = 1000;
    var intNum = Math.round(Math.random()
        * Math.pow(10,Math.round(Math.random()*8))) ;

    window.onload = function(){
        document.getElementById("spanNumber")
            .innerHTML = intNum + " and "
            + (intNum + guessRange);
    }

</script>
</head>
<body>
    <form name="Form1" onsubmit="submitGuess();return false;">
        Obtain a search that returns between &nbsp;
        <span id="spanNumber"></span>&nbsp; results!<br/>
        <input type="text" name="yourGuess" value="Ajax">
        <input type="submit" name="b1" value="Guess"/><br/><br/>
        <span id="spanResults"></span>
    </form>
    <hr/>
    <textarea rows='24' cols='100' id='details'></textarea>
</body>
</html>

```

我们在 HTML 中设置了表单元素，在这里计算一个合适的大随机数，也声明了一个变量 googleKey。这允许我们使用 Google SOAP API 的授权密钥（license key）。在这里没有包括一个合法的密钥，因为许可条款不允许。密钥是免费的，可以提供每天有限次数的搜索请求。通过一个简单过程，可以从 Google 在线获得（见本章“资源”一节中的 URL）。

3. 提交请求

大量的工作是通过 submitGuess() 函数完成的，提交表单的时候会调用它。它定义在包含的 JavaScript 文件中，接下来我们看一下。代码清单 7-3 展示了 JavaScript 的第一个部分，这段代码调用了 Google API。

代码清单 7-3 submitGuess() 函数

传递自定义的 HTTP
首部信息

创建 ContentLoader 对象

提供访问 Google API 的 URL

检查授权密钥

构造 SOAP 消息

```
function submitGuess(){
    if (!googleKey){ ①
        alert("You will need to get a license key "
            +"from Google,\n and insert it into "
            +"the script tag in the html file\n "
            +"before this example will run.");
        return null;
    }
    var myGuess=document.Form1.yourGuess.value;

    var strSoap='<?xml version="1.0" encoding="UTF-8"?>' ②
        +'\n\n<SOAP-ENV:Envelope'
        +' xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"'
        +' xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"'
        +' xmlns:xsd="http://www.w3.org/1999/XMLSchema">'
        +'<SOAP-ENV:Body><ns1:doGoogleSearch'
        +' xmlns:ns1="urn:GoogleSearch"'
        +' SOAP-ENV:encodingStyle='
        +' "http://schemas.xmlsoap.org/soap/encoding/">'
        +'<key xsi:type="xsd:string">' + googleKey + '</key>'
        +'<q xsi:type="xsd:string">' +myGuess+'</q>'
        +'<start xsi:type="xsd:int">0</start>'
        +'<maxResults xsi:type="xsd:int">1</maxResults>'
        +'<filter xsi:type="xsd:boolean">true</filter>'
        +'<restrict xsi:type="xsd:string"></restrict>'
        +'<safeSearch xsi:type="xsd:boolean">false</safeSearch>'
        +'<lr xsi:type="xsd:string"></lr>'
        +'<ie xsi:type="xsd:string">latin1</ie>'
        +'<oe xsi:type="xsd:string">latin1</oe>'
        +'</ns1:doGoogleSearch>'
        +'</SOAP-ENV:Body>'
        +'</SOAP-ENV:Envelope>';

    var loader=new net.ContentLoader( ③
        "http://api.google.com/search/beta2", ④
        parseGoogleResponse,
        googleErrorHandler,
        "POST",
        strSoap,
        "text/xml",
        {
            Man:"POST http://api.google.com/search/beta2 HTTP/1.1",
            MessageType:"CALL"
        },
        true
    );
}
```

在submitGuess()函数中做的第一件事情是检查是否有一个授权密钥，如果没有，就提醒用户①。当我们下载这个例子的代码时，授权序密钥将设置为null，所以如果想要使用密钥的话，需要从Google获得自己的密钥。

第二个任务是构造一个非常大的 SOAP 消息❷，包含了我们提交的短语和授权密钥的值。SOAP 在设计的时候就考虑了对于自动化的支持，像我们这样手工创建 XML 是很少见的。IE 和 Mozilla 都提供了特定于浏览器的对象，可以以更加简单的方式与 SOAP 交互。尽管如此，我们还是认为介绍如何手工构建 XML，以及查看 SOAP 的请求和响应的数据，对于读者是有意义的。

创建请求的 XML 文本之后，我们构造了 ContentLoader 对象❸，将 SOAP XML 作为 HTTP 主体部分的内容传进来，同时传进来的还有 Google API 的 URL❹ 以及自定义的 HTTP 首部信息❺。我们设置 content-type 为 text/xml。注意，这代表了请求的主体部分的 MIME 类型，而不是我们期待在响应中收到的 MIME 类型，尽管在这种情况下两者是一样的。最后一个参数，设置成了 true 值，指示我们应该从 PrivilegeManager 对象中请求权限。

4. 解析响应

ContentLoader 随后发送请求，如果得到了用户的授权，就会收到服务器返回的同样大块的 XML。这里有个响应的小例子，来自对术语“Ajax”的搜索：

```
<?xml version='1.0' encoding='utf-8'?>
<soap-env:envelope
  xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/*"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<soap-env:body>
  <ns1:dogooglesearchresponse xmlns:ns1="urn:googlesearch"
    soap-env:encodingstyle="http://schemas.xmlsoap.org/soap/encoding/">
    <return xsi:type="ns1:googlesearchresult">
      <directorycategories
        xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/*"
        xsi:type="ns2:Array"
        ns2:arraytype="ns1:directorycategory[1]">

      ...
      <estimateisexact xsi:type="xsd:boolean">false</estimateisexact>
      <estimatedtotalresultscount
        xsi:type="xsd:int">741000</estimatedtotalresultscount>
      ...
    </directorycategories>
  </ns1:dogooglesearchresponse>
</soap-env:body>
</soap-env:envelope>

<hostname xsi:type="xsd:string"></hostname>
<relatedinformationpresent xsi:type="xsd:boolean">true</
  relatedinformationpresent>
<snippet xsi:type="xsd:string">de officiële site van afc &lt;b&gt;ajax&lt;/b&gt;.</snippet>
<summary xsi:type="xsd:string">official club site, including roster,
```

```
history, wallpapers, and video clips.&lt;br&gt; [english/dutch]</summary>

<title xsi:type="xsd:string">

&lt;b&gt;ajax&lt;/b&gt;.nl – splashpagina

</title>

...


```

完整的 SOAP 响应太冗长，就不包括在这里了，我们显示了其中的三个片断。第一部分定义了一些传输的首部信息，说明了响应来自哪里等等。在主体部分中，我们找到了一对元素，描述了估计的结果数目——这条短语返回了 741 000 个结果，这不能认为是一个准确的数字。最后，我们可以看到第一个返回结果的一部分，描述的是到荷兰阿贾克斯足球队主页的链接。代码清单 7-4 展示了回调处理函数，用来解析这个响应。

代码清单 7-4 parseGoogleResponse() 函数

```
function parseGoogleResponse(){
    var doc=this.req.responseText.toLowerCase();
    document.getElementById('details').value=doc;
    var startTag='<estimatedtotalresultscount xsi:type="xsd:int">';
    var endTag='</estimatedtotalresultscount>';
    var spot1=doc.indexOf(startTag);
    var spot2=doc.indexOf(endTag);
    var strTotal1=doc.substring(spot1+startTag.length,spot2);
    var total1=parseInt(strTotal1);
    var strOut="";
    if(total1>=intNum && total1<=intNum+guessRange){
        strOut+="You guessed right!";
    }else{
        strOut+="WRONG! Try again!";
    }
    strOut+="  
Your search for <strong>" +
    +document.Form1.yourGuess.value +
    +"</strong> returned " + strTotal1 + " results!";
    document.getElementById("spanResults").innerHTML = strOut;
}
```

目前，我们不关心 SOAP 消息的结构，而只关心返回结果的估计数字。这个响应是一个有效的 XML，我们可以使用 XMLHttpRequest 对象的 responseXML 属性来解析它。然而，我们在这里选择了捷径，简单地使用字符串处理来抽取估计的结果数目。然后我们使用少量的 DOM 处理技术将结论显示给用户（他们猜测的怎么样）。出于教育目的，我们也将完整的 XML 响应显示在 textarea 元素中，以便于那些想要更加详细地查看 SOAP 数据的人查看。

5. 在 FireFox 中激活 PrivilegeManager

如前所述，PrivilegeManager 可以配置为不响应程序发出的请求。为了找出 Firefox 浏览器是否是以这种方式来配置的，在地址栏中输入 “about:config” ，显示出当前设置的偏好列表。使用过滤文本框来找到 signed.applets.codebase_principal_support。如果这个值是 true，那么上面的代码可以工作；如果不是，我们就无法访问 Google。

早期版本的 Mozilla 需要手工来编辑这些配置，然后完全重新启动浏览器。在 Firefox 中，双击偏好列表的相应行，会在 true 和 false 之间切换偏好的值。这样产生的变化会立即生效，不需要重新启动浏览器。如果偏好列表是在一个单独的标签页中打开的话，甚至也不需要刷新页面。

6. 为 Mozilla 客户端代码添加签名

因为 IE 绕过了 PrivilegeManager，所以应用程序在这个浏览器中运行地非常平滑。然而，在 Mozilla 中用户会两次面对看起来很吓人的对话框（假设浏览器配置为可以使用 PrivilegeManager），使得这样一类 Web 服务方法对于 Mozilla 用户来说更成问题了。通过选择“记住我的决定”复选框可以防止它再次出现（参见图 7-5），但是我们的开发者对此无法控制（并且这样做也是很正确的）。

这里有一种解决方案，但它要求应用程序以非常特定于 Mozilla 的方式来打包。Web 应用可以使用电子证书来签名。但是，为了添加签名，它们必须要以 JAR 文件的形式交付给 Mozilla 浏览器，也就是说，要将所有的脚本、HTML 页面、图片和位于同一地点的其他资源压缩成 zip 文件。JAR 文件使用 Thawte 和 VeriSign 等类似公司销售的各种证书来签名。已签名的 JAR 文件中的资源使用特殊的 URL 语法来引用，例如

```
jar:http://myserver/mySignedJar.jar!/path/to/someWebPage.html
```

当用户下载一个已签名的 Web 应用时，他们只会被询问一次，是否想要给应用授予任何它所申请的权限，用户需要做的就是这些。

Mozilla 提供了免费的、可下载的 JAR 文件签名工具。对于仅仅想要测试这个技术的用户来说，可以使用某些工具（例如 Sun 的 JDK 中的 keytool 工具）来生成未经鉴定的数字证书。然而，对于真正的部署，我们推荐使用来自公认的认证机构的证书。

签名的 JAR 文件是不可移植的，它们只能在 Mozilla 浏览器中起作用。出于这个原因，我们不再深究更多的细节。如果你对这种方法感兴趣的话，可以看看“资源”一节中的 URL。

到这里对使用 Ajax 与远程服务交互的讨论就结束了。到此为止，我们的应用可以在浏览器中运行，并且可以与它所在的服务器，甚至可能是与第三方的服务器交换数据。那些数据不大可能在你的机器上执行恶意的代码，但是它可能会产生另外一种类型的安全风险，特别是如果这些数据是机密的数据。在下一节中，我们将考察如何保护用户的数据不被窥探。

7.3 保护机密数据

用户所面对的 Web 浏览器并不是与服务器直接相连的。当数据提交给服务器时，它在找到服务器之前经过了很多因特网上的中间节点（例如路由器和代理服务器）。普通的 HTTP 数据是以明文来传输的，允许任何中间节点读取包中的数据。如我们将看到的，将数据暴露给任何对这些中间节点有控制权的人，都会危及数据的安全。

7.3.1 中间人

假设你刚刚编写了一个 Ajax 应用程序，该应用程序是要在因特网上发送财务明细信息，例如银行账号的数字和信用卡的明细。一个表现良好的路由器会不加修改地传输数据包，除了包头的路由信息之外不会查看任何信息。但是一个恶意的路由器（图 7-6）可能会读取传输的内容（也就是说，查找内容中的信用卡数字或者可以加入垃圾邮件列表的有效邮件地址）、修改路由信息（例如，将用户重定向到一个模仿其正在访问的网站的冒牌网站），甚至是修改数据的内容（例如，修改接收人从而将资金转移到他自己的账号）。

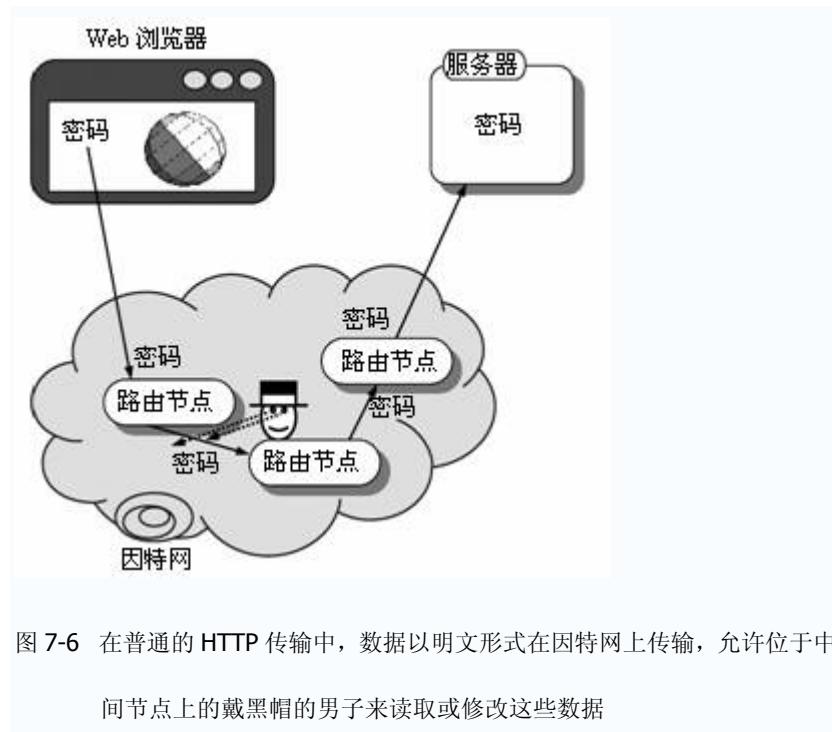


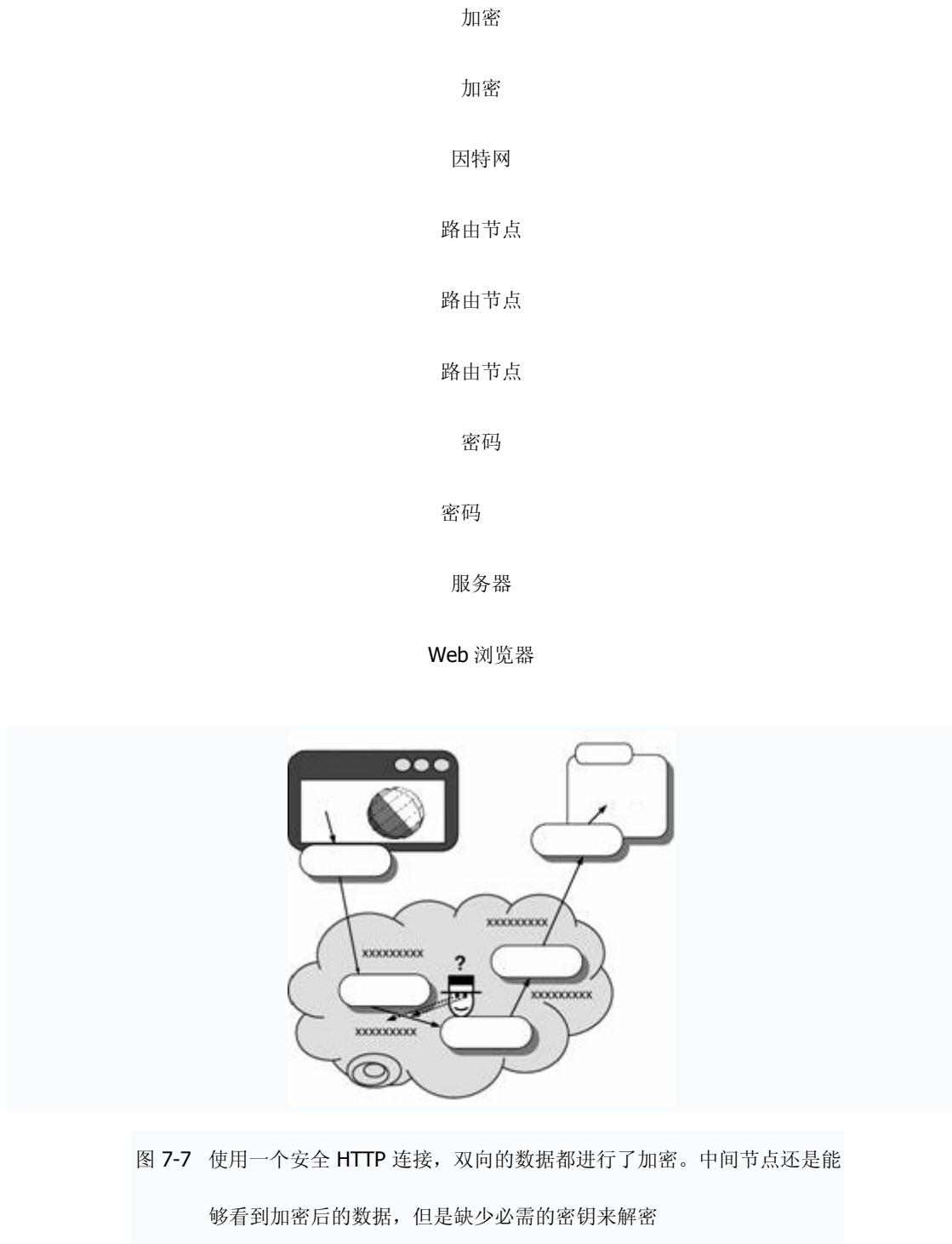
图 7-6 在普通的 HTTP 传输中，数据以明文形式在因特网上传输，允许位于中间节点上的戴黑帽的男子来读取或修改这些数据

Ajax 使用 HTTP，既用于传输客户端代码，也用于向服务器提交数据请求。我们考察过的所有通信方法——隐藏的 IFrame、HTML 表单、XMLHttpRequest 对象——在 HTTP 这个层面上都是一样的。对于任何基于 Web 的应用，试图干扰你的服务的恶意实体有几个可以下手的地方。利用这些弱点来攻击的方法称作“中间人”（man-in-the-middle）攻击。我们来考察一下保护自己免受攻击的几个措施。

7.3.2 使用安全 HTTP

如果要保护 Ajax 客户端和服务器之间的通信流量，你可以采用的最显而易见的措施就是使用一个安全连接来对通信流量进行加密。安全套接字层上的超文本传输协议（Hypertext Transfer Protocol over Secure Socket Layer，HTTPS）在明文 HTTP 周围增加了一层包装，使用公共—私有密钥对来加密双向传输的数据。中间人仍然可以看到数据包，但是因为内容加密了，他所能做的事情就不多了（图 7-7）。

HTTPS 要求在浏览器和服务器上都有对原生（native）代码的支持。现代的浏览器中都内建有对 HTTPS 的很好支持，此外大多数提供 Web 主机托管服务的公司都以合理的价格提供了安全连接。HTTPS 算起来是昂贵的，并且它传输的是二进制数据。JavaScript 在这里不是一个自然选择；就像我们不会试图使用 JavaScript 来重新实现 DOM、CSS 或者 HTTP 一样。最好将 HTTPS 看作是我们使用的一个服务，而不是为了我们自己的需要可以忽略和替换的东西。



我们来介绍几个关于 HTTPS 的警告。首先加密和解密带来了计算上的开销。在客户端，这不是一个问题，因为单个客户端只需要处理一个通信流量。然而，服务器上的额外负担对于一个大型网站来说可能是严重的。在传统的 Web 应用中，通常的做法是只在 HTTPS 上传输关键的资源，在普通的 HTTP 上发送一般的内容，例如图片和标记模板。在 Ajax 应用中，你需要理解这对于 JavaScript 安全模型的影响，它将 `http://` 和 `https://` 看作是截然不同的协议。

其次，使用 HTTPS 只能保护数据传输的安全，它本身并不是一种完善的安全解决方案。如果你使用 128 位 SSL 加密算法来安全地传输用户的信用卡明细信息，随后将信息保存在数据库中，这个数据库没有打过补丁并且已经被入侵者通过后门感染，那么数据还是很容易受到攻击。

尽管如此，HTTPS 仍然是在网络上传输敏感数据的推荐解决方案。然而，我们意识到使用它也有一些成本，可能不在小网站所有者的预算范围之内。对于那些有着更低安全需求的用户，我们接下来展示一个用于传输加密数据的普通 HTTP 机制。

7.3.3 在普通HTTP上使用JavaScript加密数...

假设你运营着一个小网站，它并不总是像例行公事一样传输需要安全连接的敏感数据。你允许用户登录进来，然而在以明文方式发送需要验证的密码时碰到了麻烦。

在这个场景中，JavaScript可以帮助你。首先，我们来概述一下解决方案，然后看看具体的实现。

1. 公钥和私钥

与其传输密码本身，倒不如传输密码的加密形式。加密算法从一个输入字符串生成一个随机的、但是可预言的输出。MD5 就是这类算法的一个例子，它有几个关键特征对于安全性是很有用的。第一，每次使用 MD5 加密一段数据总是会产生相同的结果。第二，两个不同的资源几乎不可能生成同样的 MD5 摘要。合在一起，这两个特征使得一个资源的 MD5 摘要（就是算法的输出结果）成为这个资源的一个相当好的指纹。第三个特征是 MD5 不容易被逆推。所以，MD5 摘要可以自由公开地传送，不存在恶意实体使用它来解密消息的风险。

例如，MD5 算法每次会从密码字符串“Ajax in action”生成摘要字符串“8bd04bbe6ad 2709075458c03b6ed6c5a”。我们可以在客户端加密这个字符串，将加密后的形式发送给服务器。然后服务器从数据库中获得用户的密码，使用同样的算法进行加密，并且比较两个字符串。如果匹配，服务器就允许我们登录。没有加密的密码不会通过因特网传输。

然而，我们不能为了登录而直接将MD5 摘要通过因特网传输。虽然恶意实体不可能计算出它是从“Ajax in action”生成的，但是它们很快会知道，那个特殊的摘要授权它们访问我们网站的帐户。

这就是公钥和私钥要解决的问题。与其只加密密码，倒不如加密密码和服务器提供的一串随机字符序列的组合。服务器在我们每次访问登录屏幕时提供一个不同的随机序列。随机序列通过因特网传输到客户端。

在客户端层，当用户输入密码的时候，我们在后面附加上随机字符串，然后对结果进行加密。服务器在本次登录尝试持续期间仍然记着自己以前生成的随机字符串。因此它可以获取到用户 id，从数据库中取出那个用户的正确密码，在后面附加上随机字符串，加密并且对结果进行比较。如果两个结果是匹配的，就允许用户登录；如果不匹配（说明我们输错了密码），就再次显示登录表单，但是这次带有一个不同的随机字符串。

假定服务器传输的字符串是“abcd”，“Ajax in actionabcd”的 MD5 摘要是“e992dc25b473842023f06a61f03f3787”。在下次请求中，服务器传输字符串“wxyz”，针对它会生成一个完全不同的摘要“3f2da3b3ee2795806793c56bf00a8b94”。恶意实体能够看到每一个随机字符串，将它们和加密的散列值进行匹配，但是没有办法从这些数据对中推断出密码。因此，除非它足够幸运，恰好以前看到过一个消息的随机字符串，否则它是无法截取到登录请求信息的。

这个随机字符串就是公钥。它对所有人都是可见的，可以被任意使用。我们的密码就是私钥。它的寿命很长，并且永远也不能让别人看到。

2. JavaScript 实现

实现这种解决方案需要在客户端和服务器端都有一个 MD5 生成器。在客户端，Paul Johnson 编写了一个可以免费获得的 JavaScript 生成函数库（参见“资源”一节）。使用他的代码只需要将这个库包括进来，并且调用一个简单的函数：

```
<script type='text/javascript' src='md5.js'></script>

<script type='text/javascript'>

var encrypted=str_md5('Ajax in action');

//now do something with it...

</script>
```

在服务器层，大多数流行的语言都有可用的 MD5 算法实现。PHP 从第 3 版开始有了内建的 `md5()` 函数。`java.security.MessageDigest` 类提供了 Java 加密算法的一个基本实现和很多通用算法的实现，包括 MD5 的实现。`.Net` 框架也提供了 `System.Security.Cryptography.MD5` 类。

这种技术的用处是有限的，因为服务器必须已经知道加密好的数据，以便进行比较。尽管它无法代替 HTTPS 作为一个全面的安全传输系统，但是它是提供安全登录能力并且不用求助于 HTTPS 的一种理想手段。

我们现在回顾一下讨论过的内容。来源服务器策略保护了用户的电脑免受恶意代码的攻击。通过 HTTPS 在客户端和服务器之间交换数据，保护了系统免受中间人攻击。在最后一节，我们考察一下攻击发生的第三个地点，服务器本身。你将学会如何保护自己的 Web 服务免受不受欢迎的访问者的入侵。

7.4 Ajax 数据流的访问策略

我们先来回顾一下标准的 Ajax 架构，以便识别出将要在本节讨论的这个架构中一些易受攻击的地方。客户端一旦运行在用户浏览器中，就可以使用 HTTP 向服务器发送请求。这些请求由 Web 服务器的进程（servlet、动态页面或者其他什么东西）来提供服务，服务器进程返回数据流给客户端，客户端再解析数据流。图 7-8 中概括了这种情况。

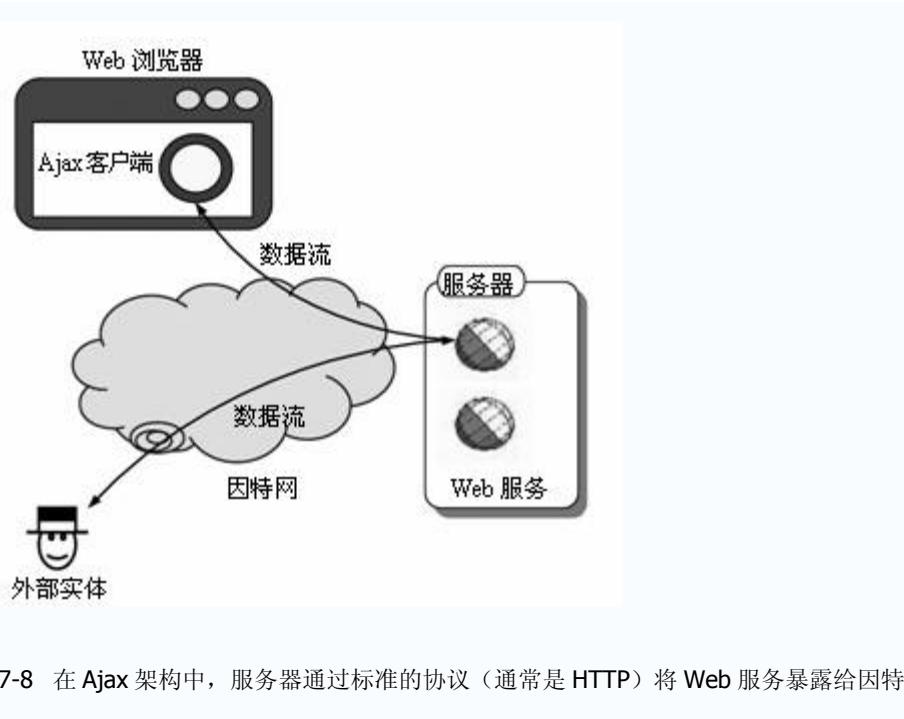


图 7-8 在 Ajax 架构中，服务器通过标准的协议（通常是 HTTP）将 Web 服务暴露给因特网。Ajax 客户端从服务器获取数据流。因为 Web 服务的天性是公开的，外部实体可以绕过客户端，直接来请求数据

外部实体可以访问 Web 服务或者页面，我们不用做任何额外的工作——这就是因特网工作的方式。我们可能还会鼓励外面的人以这种方式使用我们的 Web 服务，甚至可以发布一个 API，如同 eBay、Amazon、Google 以及其他网站已经做的那样。即使在这种情况下，我们还是需要时刻关注安全性。我们可以做两件事情，在随后两节中会加以讨论。首先，我们设计 Web 服务接口或者 API 可以采取这样的方式，即外部实体无法扰乱 Web 应用的用途——例如，订购商品而又不付帐。其次，我们考察一下限制特定的团体访问 Web 服务的技术。

7.4.1 设计安全的 Web 层

当设计 Web 应用的时候，我们的脑海中通常会有一个端到端的工作流程。例如，用户浏览商店中的货物，将采购的东西放进购物篮，然后进行结算。结算过程本身有一个经过良好定义的工作流，有送货地址、运输方式、支付方式，最后还需要订单确认。只要应用程序仍然掌握着控制权，我们就可以相信工作流是正确使用的。然而，如果外部实体开始直接调用 Web 服务的话，我们可能就有麻烦了。

1. 屏幕搜刮器与 Ajax

传统 Web 应用易于受到自动遍历这些工作流的“屏幕搜刮”（screen-scraping）程序的攻击，这些程序修改 HTTP 请求，就像是用户在表单录入后所产生的一样。屏幕搜刮器能够减少网站的广告收入，歪曲 Web 统计数据。更严重的是，通过使得原本意图只发生在人和应用之间的交互自动化，它们可以破坏应用的工作流程，使调用服务器的事件次序混乱，还可以通过重复提交使服务器进程超载。从安全性观点来看，它们通常都被认为是有问题的。

传统Web应用的页面中的数据经常埋葬在一堆模板HTML和装饰性的内容之中。在一个良好分解的（well-factored）Ajax应用中，发送给客户端的Web页面是简单得多的、结构良好的数据。在表现和逻辑之间分离关注点是很好的设计，但是这也使得屏幕搜刮器的工作更加简单了，因为从服务器返回的数据是用来解析的，而不是用来在浏览器中呈现的。屏幕搜刮程序往往是很脆弱的，网站的外观发生了变化就会引起它的崩溃。Ajax客户端在视觉上的改动不大可能改变客户端应用与服务器通信的底层Web服务的特征。为了保护应用程序的完整性，在设计用于客户端和服务器之间通信的高层API的结构时，我们需要考虑一下这些问题。通过API的设计来解决这些问题，我们不是指HTTP、SOAP或XML，而是指动态页面的URL和它们所接受的参数。

2. 举例：在线战舰游戏

为了展示 Web 服务 API 的设计是如何影响应用的安全性的。我们来考察一个简化了的例子。我们正在开发经典的棋盘游戏“战舰”（Battleship）的一个在线版本（见“资源”一节），该游戏使用 Ajax 客户端来玩，Ajax 客户端使用 Web 服务与服务器通信。我们想要确保这个游戏是无法作弊的，即使是在一个恶意的玩家对客户端做了手脚，不按顺序向服务器发送数据的情况下。

游戏的目标是让每个玩家去猜测其他人的船只的位置。游戏由两个阶段组成。第一个阶段，玩家将他们的每个棋子放在棋盘上。一旦完成之后，他们轮流去猜测棋盘上特定的方格，看看他们是否可以弄沉其

他人的船。在游戏的过程中，棋盘的主拷贝保存在服务器上，而每一个客户端也维护了一个代表自己棋盘的模型和一个其他人的棋盘的空白拷贝，随着他们的战舰被发现，这个空白拷贝会逐渐被填满（图 7-9）。



我们来看看设置阶段。首先，清空棋盘；然后将每个棋子放在棋盘上，直到全部棋子都放好。在设置过程中，有两种方式来设计客户端对服务器的服务调用。第一种是采用细粒度的方式，先清空棋盘，随后在指定位置放上指定的棋子。在设置阶段的过程中，会多次访问服务器，第一次是清空棋盘，以后是每次摆放一个棋子。表 7-2 描述了这个细粒度的设置 API。

表 7-2 战舰游戏设置阶段的细粒度 Web API

URL	参数	返回数据
clearBoard.do	userid	确认
positionShip.do	userid shiplength coordinates(x,y) format orientation (N, S, E 或 W)	确认或者错误

第二个设计是一种粗粒度的方式，只用一个服务调用来清空棋盘和摆放全部的棋子。在这种方式下，只在设置过程中访问一次服务器。表 7-3 描述了这种替代的 API。

表 7-3 战舰游戏设置阶段的粗粒度 Web API

URL	参数	返回数据
-----	----	------

URL	参 数	返回数据
setupBoard. do	userid coordinates array of (x, y, length, orientation) structs	确认或者错误

在第 5 章讨论 SOA 的时候，我们已经对比过这两种风格的服务架构。单个网络调用更有效率，在层之间提供了更好的解耦，而且也有助于保护我们的游戏。

在细粒度的方式下，客户端负责检查摆放的棋子的正确数目和类型，而服务器模型负责在设置过程后期校验系统的正确性。在粗粒度的方式下，这种检查也写入了服务调用的文档格式之中。

一旦设置完成，一个额外服务调用被定义，用来代表游戏的回合，一个玩家在这个回合中尝试猜测另一玩家的战舰位置。按照这个游戏的性质，必须有一个细粒度的服务调用来代表对一个方格的一次猜测，如表 7-4 所示。

表 7-4 战舰游戏阶段的 Web API（用于细粒度和粗粒度的设置风格）

URL	参 数	返回数据
guessPosition. do	userid coordinates(x, y)	“hit”、“miss”或“not your turn”，加上对其他玩家上次猜测的更新

在正确玩这个游戏的时候，用户双方可以以任何顺序放置棋子，然后依次调用 URL guessPosition. do。服务器会管理游戏的顺序，如果玩家试图不按顺序下棋，服务器会返回一个“not your run”的响应。

现在我们来戴上黑帽，试着 hack 这个游戏。我们编写了一个客户端，能够以任何它所喜欢的顺序调用 Web 服务 API。我们如何才能使得自己赢的机会更大呢？我们不能给自己增加额外的回合，因为服务器监视着这种行为——这是已发布的 API 的一部分。

一种可能的欺骗手段是在设置阶段结束后移动一个棋子。在细粒度的架构下，我们可以在游戏进行过程中尝试调用 positionShip. do。如果服务器代码写得很好，它会注意到这样做违反了规则，随后返回一个拒绝。然而，做这样的尝试我们并不会失去什么，它取决于服务器端的开发者可以预测到这些误用，并且编写了防范它们的代码。

另一方面，如果服务器使用的是粗粒度的 API，它就不可能移动个别的棋子，而不去清空整个棋盘。按照对你有利的方式对游戏做细微的调整是不可能的。

粗粒度的 API 限制了任何恶意黑客的灵活性，而又不必损害守法用户的可用性。在一个设计良好的服务器模型下，对细粒度的 API 的使用不应出现任何漏洞，但是存在潜在漏洞的入口点（entry point）的数量还是要高得多，而且检查这些存在安全缺陷的入口点的负担完全依靠服务器层的开发人员。

在 5.3.4 节中，我们建议使用 Façade 来简化面向服务的架构（SOA）所暴露的 API。从提高安全性的角度，我们推荐在这里也这样做，因为只向因特网暴露一组更加简单的入口点，也会更加容易管理。

设计可以对应用暴露给外部实体的 API 加以限制，但是我们仍然需要提供一些合法的 Ajax 客户端所使用的入口点。在下一小节，我们检查一下保护这些入口点的方法。

7.4.2 限制对 Web 数据的访问

在理想的世界里，我们允许 Ajax 客户端（或者可能是其他经授权的团体）访问我们的应用所提供的动态数据，并且阻止任何其他人的访问。利用一些富客户端技术，我们可以使用自定义的网络协议。但是 Ajax 应用只能通过 HTTP 来通信，因此无法这样做。正如前面讨论过的，安全 HTTP 可以让个人事务中的数据远离窥探的目光，但是它不能用于确定是谁在调用一个特定的 URL。

幸运的是，HTTP 是一个相当丰富的协议，XMLHttpRequest 对象在 HTTP 之上为我们提供了很好的细粒度的控制。当一个请求到达服务器的时候，我们可以访问一组 HTTP 首部信息，从中可以推断出请求的来源。

1. 过滤 HTTP 请求

为了提供具体的例子，我们将在这里使用 Java 代码。其他的服务器端技术也提供了类似的方法来实现我们所描述的技术。在 Java Web 应用规约中，我们可以定义类型为 javax.servlet.Filter 的对象，这些对象可以在某些特定的请求到达它们被处理的目的地之前拦截这些请求。Filter 的子类重载了 doFilter() 方法，可以在决定是允许通过还是将其发送到另一个目的地之前检查这些请求。代码清单 7-5 展示了一个简单的安全过滤器的代码，这个过滤器会检查请求，然后或是允许通过，或是转到一个错误页面。

代码清单 7-5 通用的 Java 安全过滤器

检查请求的合法性

转到拒绝 URL

配置拒绝 URL

```

public abstract class GenericSecurityFilter implements Filter {
    protected String rejectUrl=null;
    public void init(FilterConfig config)
        throws ServletException {
        rejectUrl=config.getInitParameter("rejectUrl"); ①
    }

    public void doFilter(
        ServletRequest request, ServletResponse response,
        FilterChain chain)
        throws IOException, ServletException {
        if (isValidRequest(request)){ ②
            chain.doFilter(request, response);
        }else if (rejectUrl!=null){ ③
            RequestDispatcher dispatcher
                =request.getRequestDispatcher(rejectUrl);
            dispatcher.forward(request, response);
        }
    }

    protected abstract boolean
        isValidRequest(ServletRequest request);

    public void destroy(){}
}

```

这个过滤器是个抽象类，定义了一个抽象方法 `isValidRequest()`。这个方法在传递一个结论之前检查接收到的请求对象。如果方法失败了②，就转到另外一个 URL③。这个 URL 是由 Web 应用的配置文件来定义的①，我们很快就会看到。

过滤器为我们定义具体的子类提供了相当大的灵活性。我们可以用不止一种安全性策略来改写它。

2. 使用 HTTP 会话

一个通用的方法是当用户登录时在 HTTP 会话中创建一个权标。处理后续的请求时，在执行任何其他操作之前检查会话中这个对象是否还存在。代码清单 7-6 展示了这种类型的一个简单的过滤器。

代码清单 7-6 会话的权标检查过滤器

```

public class SessionTokenSecurityFilter
extends GenericSecurityFilter {
    protected boolean isValidRequest(ServletRequest request) {
        boolean valid=false;
        HttpSession session=request.getSession();
        if (session!=null){
            UserToken token=(Token) session.getAttribute("userToken");
            if (token!=null){
                valid=true;
            }
        }
        return valid;
    }
}

```

这种技术通常用在传统的 Web 应用中，如果校验失败通常转到一个登录屏幕。在 Ajax 应用中，我们可以自由地以 XML、JSON 或者纯文本格式返回一个简单得多的响应，客户端收到这个响应后可以提示用户重新登录。在第 11 章，我们将讨论用于 Ajax Portal 应用的这类登录屏幕的更为完整的实现。

3. 使用加密的 HTTP 首部信息

另一个对请求做校验的通用策略是给 HTTP 请求加上额外的首部信息，并且在过滤器中检查这些首部信息。代码清单 7-7 展示了第二个过滤器的例子，它寻找特定的首部信息，然后将加密过的值和服务器持有的一个已知的密钥进行核对。

代码清单 7-7 HTTP 首部信息检查过滤器

配置首部信息名

```
public class SecretHeaderSecurityFilter  
extends GenericSecurityFilter {  
    private String headerName=null;  
    public void init(FilterConfig config) throws ServletException {  
        super.init(config);  
        headerName=config.getInitParameter("headerName"); ←
```

比较首部信息的值

得到首部信息的值

```
}  
  
protected boolean isValidRequest(ServletRequest request) {  
    boolean valid=true;  
    HttpServletRequest hrequest=(HttpServletRequest)request;  
    if (headerName!=null){  
        valid=false;  
        String headerVal=hrequest.getHeader(headerName); ①  
        Encrypter crypt=EncryptUtils.retrieve(hrequest);  
        if (crypt!=null){  
            valid=crypt.compare(headerVal); ②  
        }  
    }  
    return valid;  
}
```

当测试请求时，过滤器读取一个特定的首部信息名①，然后将它和一个保存在服务器会话中的加密过的值进行比较②。这个值是临时的，在每个特定的会话中都会随机生成，以便使得系统更难被攻破。Encrypter 类使用了 Apache Commons Codec 的类和 javax.security.Message-Digest 类来生成一个十六进制

编码的 MD5 值。完整的类的代码清单可以从本书的可下载代码中获得。用 Java 来得到十六进制编码的 MD5 值的原理如下所示：

```
MessageDigest digest=MessageDigest.getInstance("MD5");

byte[] data=privKey.getBytes();

digest.update(data);

byte[] raw=digest.digest(pubKey.getBytes());

byte[] b64=Base64.encodeBase64(raw);

return new String(b64);
```

在这里，privKey 和 pubKey 分别是指私钥和公钥。为了配置这个过滤器以便检查 path/Ajax/ data 下所有的 URL，我们在 Web 应用的 web.xml 配置文件中添加了下列过滤器定义：

```
<filter id='securityFilter_1'>
  <filter-name>HeaderChecker</filter-name>
  <filter-class>
    com.manning.ajaxinaction.web.SecretHeaderSecurityFilter
  </filter-class>
  <init-param id='securityFilter_1_param_1'>
    <param-name>rejectUrl</param-name>
    <param-value>/error/reject.do</param-value>
  </init-param>
  <init-param id='securityFilter_1_param_2'>
    <param-name>headerName</param-name>
    <param-value>secret-password</param-value>
  </init-param>
</filter>
```

这段定义配置了过滤器在检查 HTTP 首部信息 secret-password 的值之后，将拒绝的请求转向到 URL/error/reject.do。为了完成配置，我们定义了一个过滤器映射来匹配特定路径下的所有东西：

```
<filter-mapping>

  <filter-name>HeaderChecker</filter-name>

  <url-pattern>/ajax/data/*</url-pattern>

</filter-mapping>
```

在客户端，可以使用 Paul Johnson 的函数库（在本章前面讨论过）来生成 Base64 MD5 摘要。为了在 Ajax 客户端加上必需的 HTTP 首部信息，我们使用 setRequestHeader() 方法，概述如下：

```
function loadXml(url){

  var req=null;

  if (window.XMLHttpRequest){
```

```

req=new XMLHttpRequest();
} else if (window.ActiveXObject){
    req=new ActiveXObject("Microsoft.XMLHTTP");
if (req){
    req.onreadystatechange=onReadyState;
    req.open('GET',url,true);
    req.setRequestHeader('secret-password',getEncryptedKey());
    req.send(params);
}
}

```

这里的加密函数简单地定义为一个给定字符串的 Base64 MD5 摘要：

```

var key="password";
function getEncryptedKey(){
    return b64_md5(key);
}

```

这种解决方案仍然要求我们一开始就将变量 key 传递给 Ajax 客户端。我们可以在用户登录到应用时通过 HTTPS 发送会话的 key。实际使用中，key 是随机的，当然不会是像“password”这么简单的字符串。

这种特定的解决方案的力量在于，HTTP 首部信息不会被一个标准的超链接或者 HTML 表单修改。至少，要求使用一个以编程方式实现的 HTTP 客户端会难倒一些不够坚定的黑客。当然，随着 XMLHttpRequest 的使用变得更加普遍，如何修改网页请求中的 HTTP 首部信息的知识也会传播开来。以编程方式实现的 HTTP 客户端，例如 Apache 的 HttpClient 和 Perl LWP::UserAgent 很久以前就已经可以做到这件事了。

基本上，过滤器和类似的机制并不足以使得外部代理无法进入你的网站，但是它们可以为此增加一些难度。和其他开发者一样，邪恶的黑客同样只有有限的资源和时间，使用我们前面概述的几种方法来保护应用，当然也就阻止了偶尔对数据服务发起的攻击。

到这里我们对 Ajax 应用安全性的讨论就结束了。还有几个保护 Ajax 应用值得考虑的方面在这里没有涉及，因为它们很大程度上和传统的 Web 应用是一样的。一个好的认证和授权机制有助于基于角色和职责来控制对服务的访问。标准的 HTTP 首部信息可以用来校验调用者的起源，使其更难（但不是不可能）在官方渠道之外调用服务。对于保护 Ajax 应用有更深入的兴趣的读者，我们推荐阅读一些基于 Web 安全性的相关著作。

最后，记住安全并不是一个绝对的状态。永远没有绝对安全的东西。你可以期待的最好方式是在入侵者来临之前先行一步。在相应的地方使用 HTTPS、将需要暴露的基于 Web 的 API 最小化、明智地使用 HTTP 的请求检查，这些都是迈向这个方向的好步骤。

7.5 小结

在本章中，我们讨论了使用 Ajax 的安全含义。我们将注意力集中在 Ajax 和传统 Web 应用之间存在差异的安全问题。首先，我们考察了在 Web 浏览器中对 JavaScript 的使用加以控制的沙箱，以及阻止不同来源的代码之间进行交互的规则。我们看到了为了访问像 Google API 那样的第三方因特网服务，在征得用户的同意之后，如何放宽来源服务器策略。

其次，我们考察了当数据在客户端和服务器之间传递时保护数据的方法。HTTPS 是这里推荐的具有工业强度的解决方案，但是我们也展示了一个简单的基于 Ajax 的通过明文 HTTP 安全传输密码的方法。最后，我们看到由于服务器提供原始数据的方式，Ajax 具有哪些特殊的弱点。在一些案例中，这些弱点已经被证明是一个严重的威胁。我们考察了设计服务器架构的方式，以便将这样的风险最小化。我们也描述了在服务器端编程，使得从外部访问数据变得更加困难的方法。

我们在本章所解决的问题会有助于你加强 Ajax 应用，以便将它在实际中应用。在下一章，我们通过对性能问题的考察，来继续探讨这些关于严酷的现实的主题。

7.6 资源

Google Web 服务 API 的密钥可以从 <http://www.google.com/apis/> 获取。

Paul Johnston 的 JavaScript MD5 函数库可以在 <http://pajhome.org.uk/crypt/md5/md5src.html> 找到。对于想快速体验一下 MD5 的开发者，可以访问位于 www.fileformat.info/tool/hash.htm?text= ajax +in+action 的在线校验和生成器。

我们用 Apache Commons Codec 的 Java 类库在服务器端生成 Base64-MD5，它可以在 <http://jakarta.apache.org/commons/codec/> 下载。

在 7.1 节中，我们考察了签名的 JAR 文件，来为 Mozilla 浏览器创建安全的应用。这方面的官方介绍可以在 www.mozilla.org/projects/security/components/signed-scripts.html 找到。你可以在 <http://gamesmuseum.uwaterloo.ca/vexhibit/Whitehill/Battleship/> 找到有关战舰游戏的一些背景信息。

第8章 性能

在前面三章，我们已经理解了如何实现 Ajax 应用的健壮性和可靠性——能够适应实际的使用模式以及需求的变化。设计模式可以帮助我们对代码进行良好的组织，分离关注点的原则可以使代码保持低耦合，从而在不破坏原有实现的基础上快速响应需求的变化。

当然，应用要想真正有用，还需要能够以合理的速度运行，并且不给用户计算机上的其他程序带来令人难以忍受的停顿。目前为止，我们都是在理想环境中考虑问题：用户的工作站拥有无限的资源，而且 Web 浏览器知道如何高效地使用这些资源。本章将回归资源匮乏的现实世界，来考察一下应用的性能问题。我们还是离不开重构和设计模式的知识。对于可能遇到的性能问题，这些知识可以提供用于交流的词汇——还有宝贵的洞察力。

8.1 什么是性能？

计算机程序的性能（performance）取决于两个因素：运行速度的快慢和需要消耗的系统资源（最重要的是内存和 CPU）的多少，运行速度太慢的程序将会阻碍系统运行更多的任务。在现代多任务操作系统中，如果程序打断用户正在执行的其他活动，将会使用户倍感沮丧。这些都是相互关联的问题。对于可接受的执行速度或 CPU 使用量没有固定的标准，用户运行应用的感觉是最重要的。作为程序员，喜欢关注于应用逻辑，然而性能问题是无法回避的，我们必须保持关注，稍有疏忽，用户就会告诉我们。

如同国际象棋一样，计算机语言创造的独立世界是按照一套定义明确的规则运行的。在这套规则中，每个操作都是严格定义并且可以充分解释的。在这个舒适的按部就班的世界里，程序员很容易相信这套自成一体的规则可以完全描述我们工作而且赖以生存的系统。现代计算机语言采用虚拟机的趋势更加强了这种观念，我们只需要按照规范来编写代码，不需要考虑太多底层的细节。

这完全可以理解——但却大错特错了。现代的操作系统和软件是如此的复杂，以至于无法以这种纯粹的算术方式来理解，Web 浏览器也概莫能外。编写能够真正运行在真实机器上的代码，需要透过 W3C DOM 规约或 ECMA-262 规约的华丽外表，面对我们所熟知和热爱的浏览器中各种严重问题和漏洞。如果我们不能对这些底层软件足够重视，从一开始可能走错。

无论系统设计得多么优雅，如果应用程序需要花费几秒钟来响应按钮点击或者花费几分钟来处理一个表单，那么我们已经陷入了困境。类似地，如果某个时钟应用需要占用 20MB 的系统内存，用完之后只释放了 15MB，那么潜在的用户将会很快抛弃这个应用。

JavaScript 并不以高性能知名（也确实如此），它不可能像 C 语言那样高效率地执行数学计算。JavaScript 对象也不是轻量级的，特别是 DOM 元素耗费了大量的内存。Web 浏览器在许多方面也不尽完美，自身很容易造成内存泄漏。

JavaScript 代码的性能对于 Ajax 开发者来说尤为重要，因为我们正在大胆地探索其他 Web 程序员尚未涉足的领域。成熟 Ajax 应用的 JavaScript 代码量将远远超出传统 Web 应用的代码量。而且，因为不必频繁地刷新整个页面，甚至是完全不刷新，JavaScript 对象的寿命比通常在传统的 Web 应用中的要更长一些。

在接下来的两节中，我们将继续探讨性能的两大基石，即执行速度和内存使用量。本章最后是一个案例研究，针对开发者使用 Ajax 和 DOM 时所使用的模式，展示为这些模式命名并且理解它们的重要性

8.2 JavaScript 执行速度

我们生活在一个崇尚速度的世界里，在这个世界里事情恨不得都应该在昨天就已经完成（如果你不是生活在这样一个世界，给我发一张明信片或者最好是一张移民表格，让我也前去分享）。高速的代码如果能够完成同样的工作，当然比低速的代码更具竞争优势。作为代码的开发者，我们应该对于代码能够运行得有多快以及如何对它加以改进怀有足够的兴趣。

一般而言，程序的执行速度是指该程序最慢的子系统的执行速度。我们可以测定整个程序运行得有多快，但是这个数字对于我们没有多大意义。如果能够单独测定个别的子系统，将会很有用。详细测量代码执行速度的行为通常称作性能分析(profiling)。创建优秀代码的过程就好像创建优秀的艺术品，永无止境。伟大的、充满灵感的代码经常出现在一些有趣的新兴技术领域（另一方面，拙劣的垃圾代码也经常出现在这些有趣的领域）。通过对代码进行优化，总是可以压榨出更多一点的速度。这里的限制因素往往是时间而非技巧或灵活性。通过使用优秀的性能分析器，我们可以识别出代码的瓶颈所在，从而确定在哪里集中投入精力可以取得最佳的结果。另一方面，如果我们试图在编写代码的时候就对代码进行优化，结果可能会不尽理想，因为性能瓶颈很少出在开发者意料之中的位置。

在本节中，我们将检查测定应用代码运行时间的几种方法，并且在 JavaScript 中创建一个简单的性能分析工具，同时再实战演练一个真实的性能分析器。然后，继续考察一些简单的程序，并且通过性能分析器运行这些程序，看看如何以最佳的方式优化它们。

8.2.1 测定应用时间的艰难方式

用来测量时间的最简单的工具是系统时钟，JavaScript 通过 Date 对象提供对系统时钟的访问。如果不带参数实例化一个 Date 对象，该实例将告诉我们当前的时间。如果两个 Date 对象相减，将会得到它们之间相差的毫秒数。代码清单 8-1 概述了 Data 对象在定时事件中的用法。

代码清单 8-1 使用 Date 对象的定时代码

```
function myTimeConsumingFunction(){
    var beginning=new Date();
    ...
    //do something interesting and time-consuming!
    ...
    var ending=new Date();
    var duration=ending-beginning;
    alert("this function took "+duration
        +"ms to do something interesting!");
}
```

在这个例子函数中，我们在想要测量的代码块两端各自定义了一个 Date 对象，然后计算两者之差，得到这段代码运行的持续时间。在这个例子中，使用 alert() 语句通知我们测得的时间值，这是一种最简单的情况，不会打断测量的工作流程。通常的方法是收集这种类型的数据并写入到日志文件中，但是 JavaScript 的安全模型阻止我们访问本地文件系统。对于 Ajax 应用，可以使用的最佳方法是将性能分析数据作为一系列的对象保存在内存中，稍后将这些对象呈现为 DOM 节点以创建测量报告。

注意，在程序运行的过程中，为了避免对进行测量的系统产生干扰，我们希望性能分析代码尽可能地高速和简单。在程序运行过程中将变量写到内存要比创建额外的 DOM 节点快得多。

代码清单 8-2 定义了一个可以用来分析代码性能的简单的 stopwatch 库。当测试程序运行时，性能分析数据存储在内存中，然后呈现为报告。

代码清单 8-2 stopwatch.js

客户端代码的入口点

已注册的定时器数组

```
var stopwatch=new Object();
stopwatch.watches=new Array(); ←

stopwatch.getWatch=function(id,startNow){ ←
    var watch=stopwatch.watches[id];
    if (!watch){
        watch=new stopwatch.StopWatch(id);
    }
}
```

定时事件对象构造函数

对象构造函数

```

    if (startNow) {
        watch.start();
    }
    return watch;
}

stopwatch.StopWatch=function(id){  ←
    this.id=id;
    stopwatch.watches[id]=this;
    this.events=new Array();
    this.objViewSpec=[
        {name: "count", type: "simple"},
        {name: "total", type: "simple"},
        {name: "events", type: "array", inline:true}
    ];
}
stopwatch.StopWatch.prototype.start=function(){
    this.current=new TimedEvent();
}
stopwatch.StopWatch.prototype.stop=function(){
    if (this.current){
        this.current.stop();
        this.events.append(this.current);
        this.count++;
        this.total+=this.current.duration;
        this.current=null;
    }
}

stopwatch.TimedEvent=function(){  ←
    this.start=new Date();

```

性能分析报告生成函数

```

    this.objViewSpec=[
        {name: "start", type: "simple"},
        {name: "duration", type: "simple"}
    ];
}
stopwatch.TimedEvent.prototype.stop=function(){
    var stop=new Date();
    this.duration=stop-this.start;
}

stopwatch.report=function(div){  ←
    var realDiv=xGetElementById(div);
    var report=new objviewer.ObjectViewer(stopwatch.watches,realDiv);
}

```

stopwatch 系统由一个或者多个类别组成，每个类别一次可以定时一个活动事件，并且维护一个先前的定时事件列表。当客户端代码以给定的 ID 作为参数调用 stopwatch.start() 时，系统将为该类别创建一个新的 StopWatch 对象，或者重用已存在的对象。然后，客户端代码能够 start()（启动）和 stop()（停止）观察若干次。每次调用 stop() 时，生成一个 TimedEvent 对象，计算定时事件的开始时间和持续时长。如果多次启动 stopwatch 而中间没有停止，除了保留最后一次调用 start() 的时间，其他的时间将被废弃。

这个过程生成了一个 StopWatch 类别的对象图，每个类别包含了定时事件的历史记录，如图 8-1 所示。

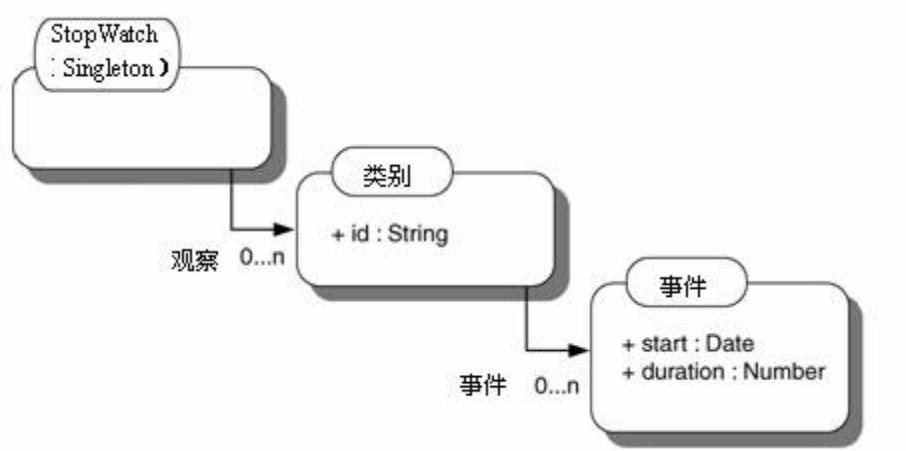


图 8-1 stopwatch 库中类的对象图。每个类别用一个包含该类别历史事件的对象来代表。

所有类别可以通过 `Singleton` 的 `stopwatch.watches` 来访问

当数据收集完成后，可以查询和显示整个对象图。这里的 `render()` 函数利用第 5 章使用的 `ObjectViewer` 库来自动呈现报告。还可以将数据以 CSV 格式输出，以便剪切和粘贴到文件中，我们将这个工作作为练习留给读者。

代码清单 8-3 展示如何将 stopwatch 代码应用于例子的“time consuming”函数。

代码清单 8-3 使用 stopwatch 库测定代码的运行时间

```

function myTimeConsumingFunction(){
    var watch=stopwatch.getWatch("my time consuming function",true);
    ...
    //do something interesting and time-consuming!
    ...
    watch.stop();
}

```

现在，stopwatch 代码能够以相对不唐突的方式添加到代码中。出于不同的测量目的，我们可以定义任意多个自己喜欢的类别。在这里，我们采用与函数相同的名称对类别进行命名。

在继续前进之前，我们将这个库应用到实例中。一个合适的例子是在第 4 章讨论 Observer 模式和 JavaScript 事件时用到的 mousemat 例子。这个例子包含两个观察鼠标在主 mousemat DOM 元素区域之上移动的进程，其中一个进程将鼠标当前坐标写到浏览器的状态栏，另一个进程在一个缩略图元素中描绘鼠标的光标位置。两个进程都提供了有用的信息，但是它们也带来了一些处理开销。我们想知道，哪一个进程占用了更多的处理时间。

使用 stopwatch 库，我们可以很轻松地向例子中添加性能分析功能。代码清单 8-4 展示了修改后的页面，包含了一个用于容纳性能分析报告的新 DIV 元素，并且在我们感兴趣的代码块中插入了一些 stopwatch 的 JavaScript 方法。

代码清单 8-4 带有性能分析的 mousemat.html

```
<html>

<head>
<link rel='stylesheet' type='text/css' href='mousemat.css' />
<link rel='stylesheet' type='text/css' href='objviewer.css' />
<script type='text/javascript' src='x/x_core.js'></script>
<script type='text/javascript' src='extras-array.js'></script>
<script type='text/javascript' src='styling.js'></script>
<script type='text/javascript' src='objviewer.js'></script>
<script type='text/javascript' src='stopwatch.js'></script>
<script type='text/javascript' src='eventRouter.js'></script>
<script type='text/javascript'>

var cursor=null;

window.onload=function(){
    var watch=stopwatch.getWatch("window onload",true);
    var mat=document.getElementById('mousemat');
    cursor=document.getElementById('cursor');

    var mouseRouter=new jsEvent.EventRouter(mat,"onmousemove");
    mouseRouter.addListener(writeStatus);
    mouseRouter.addListener(drawThumbnail);
    watch.stop();
}

function writeStatus(e){
    var watch=stopwatch.getWatch("write status",true);
    window.status=e.clientX+"-"+e.clientY;
    watch.stop();
}

function drawThumbnail(e){
    var watch=stopwatch.getWatch("draw thumbnail",true);
    cursor.style.left=((e.clientX/5)-2)+"px";
    cursor.style.top=((e.clientY/5)-2)+"px";
    watch.stop();
}
</script>
</head>

<body>

<div>
<a href='javascript:stopwatch.report("profiler")'>profile</a>
</div>

<div>
<div class='mousemat' id='mousemat'></div>
<div class='thumbnail' id='thumbnail'>
    <div class='cursor' id='cursor'></div>
</div>
<div class='profiler objViewBorder' id='profiler'></div>
</div>

</body>
</html>
```

我们定义了三个 stopwatch (一个 window.onload 事件和两个鼠标监听过程)，并且为这些 stopwatch 分配了有意义的名称，这些名称将在生成报告时使用。我们加载修改后的应用，然后快速地旋转鼠标。

当鼠标像以前那样在 mousemat 上移动时，性能分析器忙于收集数据，这些数据可以在任意时刻通过点击左上方的 profile 链接来检查。图 8-2 中显示了在鼠标移动上百次之后的浏览器中的应用，及其相关的性能分析报告。

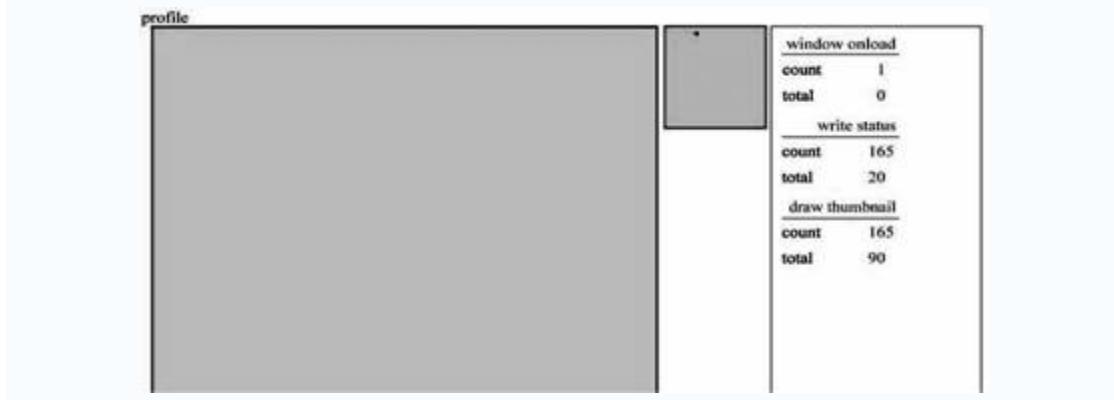


图 8-2 来自第 4 章的 mousemat 例子，它使用 JavaScript 性能分析器运行并使用活动的 stopwatch 生成报告。我们分析了 window.onload 事件的性能，响应鼠标移动时在缩略图中画下光标的位置，并且更新状态栏中鼠标的坐标。**count** 指示每个代码块的记录数，**total** 则为代码块所消耗的时间

在 Firefox 和 IE 浏览器中都可以看到，写状态栏所消耗的时间比描绘缩略图所消耗时间的四分之一还要少。

注意，window.onload 事件显示的执行时间为 0 毫秒，这要归功于 JavaScript Date 对象有限粒度。使用这个性能分析系统，我们完全工作在 JavaScript 解释器中，因此也带有了它所有的局限性。Mozilla 浏览器能够利用内建在浏览器中的原生性能分析器。我们下面来探讨一下。

8.2.2 使用 Venkman 性能分析器

Mozilla 系列的浏览器拥有一套丰富的插件扩展，其中发布历史较久而且比较著名的插件是 Venkman 调试器，可以用来对 JavaScript 代码做逐行单步调试。我们在附录 A 中讨论了 Venkman 的调试功能，现在来考察它的一个鲜为人知的能力，即用作代码性能分析工具。

在 Venkman 中对代码进行性能分析，只需打开感兴趣的页面，然后从浏览器的 Tool 菜单中打开调试器（这里假设已经安装了 Venkman 扩展，如果尚未安装，请参见附录 A）。在工具栏中有一个名为 Profile 的时钟模样的按钮（图 8-3），点击这个按钮后会在图标上添加一个绿色的勾号。

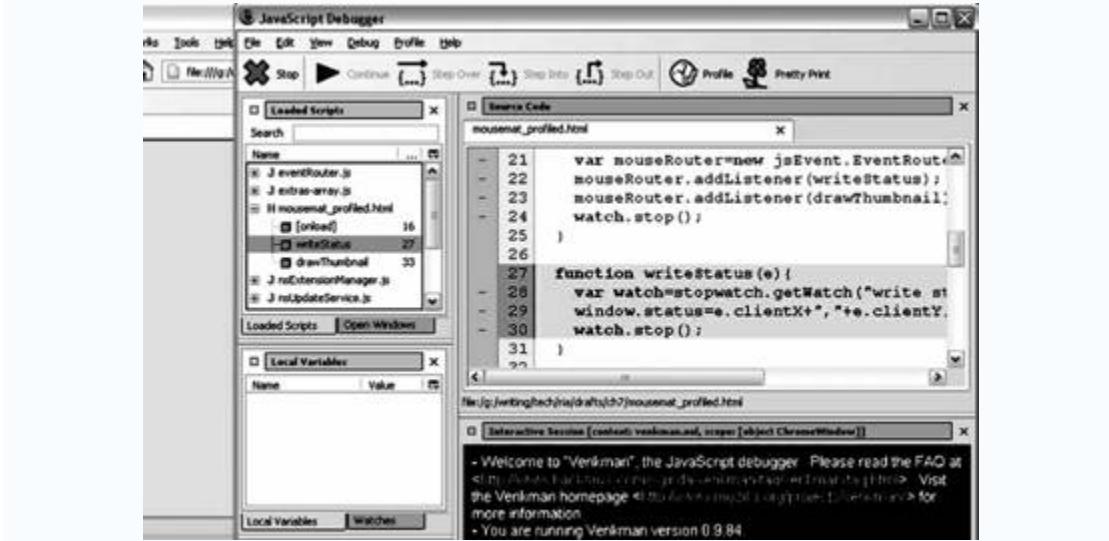


图 8-3 Mozilla 的 Venkman 调试器。选中 Profile 按钮后，所有加载脚本（显示在面板的左上方）执行所消耗的时间都会记录下来

现在，Venkman 开始细致地记录浏览器 JavaScript 引擎上发生的所有事情。在 mousemat 区域拖动鼠标几秒钟，然后再次点击调试器中的 Profile 按钮，停止性能分析。从调试器的菜单栏中选择 Profile→Save Profile Data As 选项，数据可以以多种格式来保存，包括 CSV（用于电子表格）、HTML 报告或者 XML 文件。

不幸的是，Venkman 往往会生成太多的数据而且会首先列出各种各样的 chrome://URL，这些是浏览器的内在组成部分或者是用 JavaScript 实现的插件，我们可以将它们完全忽略。除了 HTML 页面中那些主要的方法，我们使用的所有 JavaScript 库的所有函数都会被记录下来——包括上一节所开发的 stopwatch.js 性能分析器。图 8-4 显示了主 HTML 页面的 HTML 报告的相关部分。

Venkman 生成的结果与我们自己的 stopwatch 对象测定的时间差距不大——重写状态栏所消耗的时间大约是更新缩略图元素所消耗的时间的三分之一。

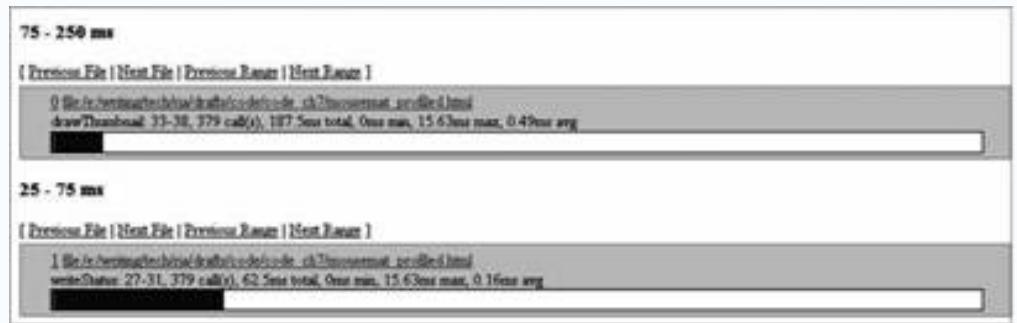


图 8-4 Venkman 生成的 HTML 性能分析报告的片断，显示出在例子页面中每个监听 mousemove 事件 DOM 元素上鼠标移动的方法的信息：被调用的次数、总共消耗的时间、最小消耗的时间和最大消耗的时间以及平均消耗的时间

Venkman 是一个很有用的性能分析工具，能够在完全不修改原有代码的情况下生成大量的数据。如果希望性能分析代码运行在不同的浏览器中，那么可以使用 stopwatch 库。在下一节，我们将考察几个例子的代码片断，演示一些可以提升代码速度的重构技巧。利用 stopwatch 库来测量可以从中获得的收益。

8.2.3 优化 Ajax 应用的执行速度

代码优化是一种“魔法”。为 Web 浏览器编写 JavaScript 通常的毛病是缺乏章法，因此，优化 Ajax 代码无疑是一个晦暗的话题。围绕这个话题有很多流行的说法，其中不少都说得很好。尽管如此，使用在 8.2.1 节开发的性能分析库，我们能够让很多怀疑论者大跌眼镜，并且对这些流行的说法进行检验。在本节中，我们将考察三种通常用于改进执行速度的策略，并且看看在实际应用中它们的适用范围。

1. 优化 for 循环

我们考察的第一个例子是一个相当常见的编程错误。这个错误并不局限于 JavaScript，但是在编写 Ajax 代码时肯定更容易产生。我们在例子中加入了一个长时间毫无意义的计算，仅仅是为了占用足够的时间以便测量出真实的差异。在这里，我们计算斐波那契 (Fibonacci) 序列，该序列中每个后续的值是前两个值的和。例如，假设使用两个 1 开始这个序列，我们会得到：

1, 1, 2, 3, 5, 8, ...

斐波那契序列的 JavaScript 计算函数如下所示：

```
function fibonacci(count){
```

```
var a=1;  
var b=1;  
for(var i=0;i<count;i++){  
    var total=a+b;  
    a=b;  
    b=total;  
}  
return b;  
}
```

对于这个序列，我们感兴趣的仅仅是它所耗费的计算时间。现在，假设我们想要计算所有从 1 到 n 的斐波那契序列中的值并且将它们加在一起。以下是完成这个任务的代码：

```
var total=0;  
for (var i=0;i<fibonacci(count);i++){  
    total+=i;  
}
```

顺便说一下，这是一个毫无意义的计算，但是在实际的程序中会经常遇到类似的情况，这时候需要检查一个值，而这个值在循环的每一次迭代中很难计算。上面的代码效率低，因为它在每次迭代中都要计算 fibonacci(count)，尽管每次得到的值都是相同的。for 循环的语法使得这段代码看起来不是很明显，导致这类错误很容易遗留在代码中。我们可以重新编写代码，仅仅计算一次 fibonacci()，即：

```
var total=0;  
  
var loopCounter=fibonacci(count);  
  
for (var i=0;i<loopCounter;i++){  
    total+=i;  
}
```

我们已经对代码进行了优化。但是，优化了多少？如果这是大型复杂代码的一部分，我们需要知道我们的努力是否值得。为了找到答案，我们将两个版本的代码和性能分析库都包括在一个 Web 页面中，并且将 stopwatch 附加在每个函数中。代码清单 8-5 显示了如何完成这个任务。

代码清单 8-5 对 for 循环进行性能分析

每次重新计算循环计数

```
<html>
<head>
<link rel='stylesheet' type='text/css' href='mousemat.css' />
<link rel='stylesheet' type='text/css' href='objviewer.css' />
<script type='text/javascript' src='x/x_core.js'></script>
<script type='text/javascript' src='extras-array.js'></script>
<script type='text/javascript' src='styling.js'></script>
<script type='text/javascript' src='objviewer.js'></script>
<script type='text/javascript' src='stopwatch.js'></script>
<script type='text/javascript' src='eventRouter.js'></script>
<script type='text/javascript'>

function slowLoop(count){
    var watch=stopwatch.getWatch("slow loop",true);
    var total=0;
    for (var i=0;i<fibonacci(count);i++){    ←—
        total+=i;
    }
    watch.stop();
    alert(total);
}


```

计算斐波那契序列

仅仅计算一次循环计数

```
function fastLoop(count){
    var watch=stopwatch.getWatch("fast loop",true);
    var total=0;
    var loopCounter=fibonacci(count);    ←—
    for (var i=0;i<loopCounter;i++){
        total+=i;
    }
    watch.stop();
    alert(total);
}

function fibonacci(count){    ←—
    var a=1;
    var b=1;
    for(var i=0;i<count;i++){
        var total=a+b;
        a=b;
        b=total;
    }
}
```

```

    return b;
}

function go(isFast){
    var count=parseInt(document.getElementById("count").value);
    if (count==NaN){
        alert("please enter a valid number");
    }else if (isFast){
        fastLoop(count);
    }else{
        slowLoop(count);
    }
}

</script>
</head>

<body>
<div>
<a href='javascript:stopwatch.report("profiler")'>profile</a>&ampnbsp
<input id='count' value='25' />&ampnbsp
<a href='javascript:go(true)'>fast loop</a>&ampnbsp
<a href='javascript:go(false)'>slow loop</a>
</div>

<div>
    <div class='profiler objViewBorder' id='profiler'></div>
</div>
</body>
</html>

```

函数 slowLoop() 和 fastLoop() 代表了前后两个版本的算法，并且都包装在 go() 函数中，这个函数将使用给定的计数值来调用某一版本的算法。在页面上提供了超链接，并且从相邻的 HTML 表单的文本框中传入一个计数值，用来执行每一个版本的循环。在测试机器上，我们发现使用 25 这个值可以得到一个合理的计算时间。第三个超链接用来呈现性能分析报告。表 8-1 中显示了简单测试的结果。

表 8-1 for 循环优化性能分析结果

算 法	执行时间 (ms)
最初的	3085
优化的	450

由表中的数据可以看出，将耗时很长的计算从 for 循环中提出确实产生了很大的效果。当然，在你自己的代码中，情况可能会有所不同。如果怀疑某段代码的效率，请对它进行性能分析！

在下一个例子中，我们考察一个特定于 Ajax 的问题：DOM 节点的创建。

2. 将 DOM 节点附加到文档

为了使用 Ajax 将某些内容呈现在浏览器窗口中，我们通常需要创建 DOM 节点，然后将它们附加到文档树上，可以附加到 document.body 节点或者其他节点上。DOM 节点一旦与文档建立了联系，该节点将立即被呈现，没有任何方式可以抑制这个特征。

在浏览器窗口中重新呈现文档需要重新计算不同的布局参数，因此可能需要付出相当昂贵的代价。如果我们在组装一个复杂的用户界面，可以创建所有的节点，将这些节点添加在一起，然后将组装好的结构添加到文档上。通过这种方式，页面的布局过程只需要执行一次。我们来考察一个简单的例子，这个例子创建了一个容器元素，我们将大量小的 DOM 节点随机放到容器中。在这个例子的描述中，我们首先谈到了容器节点，因此首先创建这个节点看起来是很自然的。以下是这段代码的开头部分：

```
var container=document.createElement("div");
container.className='mousemat';
var outermost=document.getElementById('top');
outermost.appendChild(container);
for(var i=0;i<count;i++){
    var node=document.createElement('div');
    node.className='cursor';
    node.style.position='absolute';
    node.style.left=(4+parseInt(Math.random()*492))+"px";
    node.style.top=(4+parseInt(Math.random()*492))+"px";
    container.appendChild(node);
}
```

outermost 元素是一个已经存在的 DOM 元素，我们将容器附加到该元素上，小的节点位于容器之中。因为首先附加容器然后再填充该容器，所以我们将要修改整个文档 count+1 次！很快地修改一点代码就可以纠正这个问题：

```
var container=document.createElement("div");
container.className='mousemat';
var outermost=document.getElementById('top');
for(var i=0;i<count;i++){
    var node=document.createElement('div');
    node.className='cursor';
```

```

node.style.position='absolute';

node.style.left=(4+parseInt(Math.random()*492))+"px";

node.style.top=(4+parseInt(Math.random()*492))+"px";

container.appendChild(node);

}

outermost.appendChild(container);

```

事实上，我们只需要移动一行代码的位置，就可以将对现有文档的修改次数降低为只有一次。代码清单 8-6 展示了测试页面的完整代码，该页面使用了 stopwatch 库来比较两个版本的函数。

代码清单 8-6 对 DOM 节点的创建进行性能分析

在开始时附加一个空容器

```

<html>

<head>
<link rel='stylesheet' type='text/css' href='mousemat.css' />
<link rel='stylesheet' type='text/css' href='objviewer.css' />
<script type='text/javascript' src='x/x_core.js'></script>
<script type='text/javascript' src='extras-array.js'></script>
<script type='text/javascript' src='styling.js'></script>
<script type='text/javascript' src='objviewer.js'></script>
<script type='text/javascript' src='stopwatch.js'></script>
<script type='text/javascript' src='eventRouter.js'></script>
<script type='text/javascript'>

var cursor=null;

function slowNodes(count){
    var watch=stopwatch.getWatch("slow nodes",true);
    var container=document.createElement("div");
    container.className='mousemat';
    var outermost=document.getElementById('top');
    outermost.appendChild(container);    ←
    for(var i=0;i<count;i++){
        var node=document.createElement('div');
        node.className='cursor';
        node.style.position='absolute';
        node.style.left=(4+parseInt(Math.random()*492))+"px";
        node.style.top=(4+parseInt(Math.random()*492))+"px";
        container.appendChild(node);
    }
    watch.stop();
}

function fastNodes(count){
    var watch=stopwatch.getWatch("fast nodes",true);
    var container=document.createElement("div");
    container.className='mousemat';

    var outermost=document.getElementById('top');
    for(var i=0;i<count;i++){
        var node=document.createElement('div');

```

在结束时附加一个填满的容器

```
node.className='cursor';
node.style.position='absolute';
node.style.left=(4+parseInt(Math.random()*492))+"px";
node.style.top=(4+parseInt(Math.random()*492))+"px";
container.appendChild(node);
}
outermost.appendChild(container);    ←
watch.stop();
}

function go(isFast){
  var count=parseInt(document.getElementById("count").value);
  if (count==NaN){
    alert("please enter a valid number");
  }else if (isFast){
    fastNodes(count);
  }else{
    slowNodes(count);
  }
}

</script>
</head>

<body>
<div>
<a href='javascript:stopwatch.report("profiler")'>profile</a>&ampnbsp;
<input id='count' value='640' />&ampnbsp;
<a href='javascript:go(true)'>fast loop</a>&ampnbsp;
<a href='javascript:go(false)'>slow loop</a>
</div>

<div id='top'>
  <div class='mousemat' id='mousemat'></div>
  <div class='profiler objViewBorder' id='profiler'></div>
</div>

</body>
</html>
```

同样，我们使用超链接来调用快速函数和慢速函数，超链接使用 HTML 表单字段的值作为参数来调用这两个函数。在这里，这个值指定了向容器中添加多少个大小的 DOM 节点，我们发现 640 是一个合理的值。简单测试的结果展示在表 8-2 中。

表 8-2 DOM 节点创建的性能分析结果

算 法	页面的布局次数	执行时间 (ms)
最初的	641	681
优化的	1	461

再一次，基于普遍认同的智慧来进行优化确实产生了立竿见影的效果。使用性能分析器，我们可以看到差别有多大，在这个特定的情况下，几乎节省了三分之一的执行时间。在一个不同的布局中，使用不同类型的节点，得到的数字可能会有所不同（注意，我们的例子只使用绝对定位的节点，因此布局引擎可以少做一些工作）。为了找出结论，性能分析器能够很容易地插入到你的代码中。

我们最后的例子考察一个JavaScript语言特征，并且通过比较不同的子系统找出系统的瓶颈所在。

3. 尽量减少点号操作符的使用

与很多其他语言一样，在JavaScript中可以通过使用“点号连接”来引用复杂对象层次中深处的变量。例如：

```
myGrandFather.clock.hands.minute
```

指的是我祖父时钟上的分钟指针。假设我们想要引用时钟上所有的三个指针，可以这样写：

```
var hourHand=myGrandFather.clock.hands.hour;  
var minuteHand=myGrandFather.clock.hands.minute;  
var secondHand=myGrandFather.clock.hands.second;
```

每次解释器遇到点号字符，它就会查找父变量的所有子变量。这里总共执行了9次这种类型的查找，其中很多次都是重复的。我们来改写这个例子：

```
var hands=myGrandFather.clock.hands;  
var hourHand=hands.hour;  
var minuteHand=hands.minute;  
var secondHand=hands.second;
```

现在我们只执行了5次查找，减少了解释器的重复工作。在编译型语言中，如Java或C#，编译器常常自动优化这些重复的代码。我不清楚JavaScript解释器是否能够进行这些优化（以及在哪个浏览器上可以），但是我能够使用stopwatch库来查明是否应该担心这个问题。

本节中的例子程序用来计算地球和月亮之间的万有引力。每个天体上都分配了很多的物理属性，例如质量、位置、速度和加速度，通过这些属性可以计算出它们之间的引力。为了很好地对点号操作符进行测试，将这些属性存储为一个复杂的对象图，就像下面这样：

```
var earth={  
    physics:{  
        mass:10,  
        pos:{ x:250,y:250 },  
        vel:{ x:0, y:0 },
```

```
acc:{ x:0, y:0 }

};

};
```

顶层对象 physics 可以认为是多余的，它的作用是增加需要解析的点号数量。

这个应用分两个阶段运行。首先，它为给定数量的时刻计算出一个仿真的场景，计算距离、引力、加速度和自从我们离开中学之后从未再看过一眼的其他类似的东西。它将每个时刻的位置数据存储在一个数组中，同时也会存储任何一个天体在运行中位置的最大值和最小值的估算值。

在第二阶段，我们使用这些数据并且使用 DOM 节点来绘制两个天体的轨道，使用最小值和最大值的数据来将画布缩放到适当的大小。在一个真实的应用中，更加常见的可能是随着仿真进程来绘制数据，这里我们划分成了两个阶段，以便分别来对计算阶段和呈现阶段进行性能分析。

我们又定义了两个版本的代码，一个效率低的和一个经过优化的。在效率低的代码中，我们使出了浑身解数来让代码使用尽可能多的点号。以下是代码的片断（不必过多理会方程式的含义！）：

```
var grav=(earth.physics.mass*moon.physics.mass)

/(dist*dist*gravF);

var xGrav=grav*(distX/dist);

var yGrav=grav*(distY/dist);
```

```
moon.physics.acc.x=-xGrav/(moon.physics.mass);

moon.physics.acc.y=-yGrav/(moon.physics.mass);

moon.physics.vel.x+=moon.physics.acc.x;

moon.physics.vel.y+=moon.physics.acc.y;

moon.physics.pos.x+=moon.physics.vel.x;

moon.physics.pos.y+=moon.physics.vel.y;
```

这是一段拙劣的代码——我们故意尽可能多地深度引用对象图，制造了冗长和缓慢的代码。这里当然存在着大量的改进空间！以下是来自优化过版本的相同代码：

```
var mp=moon.physics;
```

```
var mpa=mp.acc;  
var mpv=mp.vel;  
var mpp=mp.pos;  
var mpm=mp.mass;  
  
...  
  
var grav=(epm*mpm)/(dist*dist*gravF);  
var xGrav=grav*(distX/dist);  
var yGrav=grav*(distY/dist);  
  
mpa.x=-xGrav/(mpm);  
mpa.y=-yGrav/(mpm);  
mpv.x+=mpa.x;  
mpv.y+=mpa.y;  
mpp.x+=mpv.x;  
mpp.y+=mpv.y;
```

我们简单地在计算开始的位置将所有必需的引用保存在局部变量中，这样做可以使得代码的可读性更好，更重要的是，减少了解释器需要做的工作。代码清单 8-7 展示了允许两种算法并列进行性能分析的完整 WebGL 页面的代码。

代码清单 8-7 对变量确定进行性能分析

初始化星球的属性

```

<html>

<head>
<link rel='stylesheet' type='text/css' href='mousemat.css' />
<link rel='stylesheet' type='text/css' href='objviewer.css' />
<script type='text/javascript' src='x/x_core.js'></script>
<script type='text/javascript' src='extras-array.js'></script>
<script type='text/javascript' src='styling.js'></script>
<script type='text/javascript' src='objviewer.js'></script>
<script type='text/javascript' src='stopwatch.js'></script>
<script type='text/javascript' src='eventRouter.js'></script>
<script type='text/javascript'>
```

呈现轨道

选择计算类型

初始化星球的属性

```

    acc:{ x:0, y:0 }
  );
};

var earth={  ←
  physics:{
    mass:10,
    pos:{ x:250,y:250 },
    vel:{ x:0, y:0 },
    acc:{ x:0, y:0 }
  };
};

var gravF=100000;

function showOrbit(count,isFast){
  var data=(isFast) ? ①
    fastData(count) :
    slowData(count);
  var watch=stopwatch.getWatch("render",true);
  var canvas=document.②
    getElementById('canvas');
  var dx=data.max.x-data.min.x;
  var dy=data.max.y-data.min.y;
  var sx=(dx==0) ? 1 : 500/dx;
  var sy=(dy==0) ? 1 : 500/dy;
  var offx=data.min.x*sx;
  var offy=data.min.y*sy;
  for (var i=0;i<data.path.length;i+=10){
    var datum=data.path[i];
    var dpm=datum.moon;
    var dpe=datum.earth;

    var moonDiv=document.createElement("div");
    moonDiv.className='cursor';
  }
}
```

大量使用点号操作符

```

        moonDiv.style.position='absolute';
        moonDiv.style.left=parseInt((dpm.x*sx)-offx)+"px";
        moonDiv.style.top=parseInt((dpm.y*sx)-offy)+"px";
        canvas.appendChild(moonDiv);

        var earthDiv=document.createElement("div");
        earthDiv.className='cursor';
        earthDiv.style.position='absolute';
        earthDiv.style.left=parseInt((dpe.x*sx)-offx)+"px";
        earthDiv.style.top=parseInt((dpe.y*sx)-offy)+"px";
        canvas.appendChild(earthDiv);
    }
    watch.stop();
}

function slowData(count){          ③
    var watch=stopwatch.getWatch("slow orbit",true);

```

保守地使用点号操作符

```

var data={
    min:{x:0,y:0},
    max:{x:0,y:0},
    path:[]
};

...
}

watch.stop();
return data;
}

function fastData(count){          ④
    var watch=stopwatch.getWatch("fast orbit",true);
    var data={
        min:{x:0,y:0},
        max:{x:0,y:0},
        path:[]
    };
    ...
}

watch.stop();
return data;
}

function go(isFast){
    var count=parseInt(document.getElementById("count").value);
    if (count==NaN){
        alert("please enter a valid number");
    }else{
        showOrbit(count,isFast);
    }
}

</script>
</head>

```

```

<body>
<div>
<a href='javascript:stopwatch.report("profiler")'>profile</a>&nbsp;
<input id='count' value='640' />&nbsp;
<a href='javascript:go(true)'>fast loop</a>&nbsp;
<a href='javascript:go(false)'>slow loop</a>
</div>

<div id='top'>
  <div class='mousemat' id='canvas'>
</div>
  <div class='profiler objViewBorder' id='profiler'></div>
</div>

</body>
</html>

```

现在对结构应该是相当地熟悉了。函数 slowData()❸ 和 fastData()❹ 包含两个版本的计算阶段，用来生成数据结构❶。我已经从清单中忽略了全部的算法，因为它们会占据大量篇幅。样式的差别已经在早先展示的片断中描述过了，完整的代码清单可以在本书的下载代码中获得。程序为每个计算函数分配一个 stopwatch 对象，用于对整个计算步骤进行性能分析。这些函数通过 showOrbit() 函数来调用，该函数获得数据，然后创建计算得到的轨道的 DOM 表现❷。这个过程同样用第三个 stopwatch 进行性能分析。

用户界面的元素与前两个例子中相同，使用一个超链接来运行快速和慢速的计算，传入文本输入框的值作为参数。在这里，该值为仿真所运行的迭代数量。第三个超链接显示了性能分析数据。表 8-3 显示了运行了默认的 640 次迭代之后的结果。

表 8-3 变量确定的性能分析结果

算 法	执行时间(ms)
最初的计算	94
优化过的计算	57
呈现（平均值）	702

再一次，我们看到优化产生了重大的性能提升，减少了大于三分之一的执行时间。我们可以得到结论：关于变量确定和使用太多点号操作符的民间智慧是正确的。通过自己测量出来的结果可以令我们更加放心。

然而，当我们考察整个计算和呈现的流程时，优化后消耗的时间是 760 ms，与原始的 796 ms 进行对比，节约的时间百分比从 40% 降低到接近 5%。呈现子系统而非计算子系统，才是应用的瓶颈所在。可以断定，在这里对计算代码进行优化并没有产生巨大的回报。

这个例子展示了对代码进行性能分析的显著价值。优化之前需要知道两件事情：一段代码能够以何种特定的方式进行优化；以及这样的操作能够获得什么回报。在这里可以很容易推断出

DOM 操作消耗的时间大约是纯 JavaScript 计算的 8 倍，但是这个推断只在这个特定的例子中成立。你可能在其他很多场合中发现了相同的结果，但是不能单凭经验的推断，最好能够进行一些测量——并且最好在一组不同的机器和浏览器上进行测量。

现在，我们不再在性能分析和执行速度上花费更多的时间。通过已经运行的例子，你应该已经体会到对 Ajax 项目进行性能分析的好处。如果你的代码正在以令人满意的速度运行，这应该多少感谢一下性能分析。为了确保足够好的性能，我们还需要考察应用的内存使用量。在下一节中我们将探讨内存的使用量。

8.3 JavaScript 内存使用量

本节的主要目的是介绍在 Ajax 编程中关于内存管理的主题。其中的一些观念适用于任何的编程语言；而另一些则是 Ajax 所特有的，甚至是特定于 Web 浏览器的。

应用程序在运行过程中由操作系统来分配内存。理想情况下，程序将请求高效完成工作所需的足够内存，然后释放不再需要的内存。一个编写拙劣的应用程序可能在运行时消耗大量不必要的内存，或者在工作完成后没有释放内存。我们将程序使用的内存数量称作程序的内存使用量（memory footprint）。

随着我们从编写简单、瞬态的 Web 页面转到编写 Ajax 富客户端，内存管理的质量会对应用程序的响应性和稳定性产生巨大影响。使用基于模式的方法有助于产生稳定、易维护的代码，这些代码中潜在的内存泄漏问题会很容易发现和消除。

首先，我们来讨论一般定义上的内存管理概念。

8.3.1 避免内存泄漏

任何程序都有可能发生内存“泄漏”（即申请了系统内存并且在工作完成后没有释放），并且对于使用非托管语言（unmanaged languages）（如 C 语言）的开发者来说，内存的分配和释放是一个主要的关注点。JavaScript 是一种内存托管（memory-managed）的语言，垃圾回收过程能够帮助程序员自动地处理内存的分配和释放。该机制解决了大部分困扰的非托管代码的问题，但是，认为内存托管语言不会产生内存泄漏却是错误的。

垃圾回收进程尝试推断何时可以安全地回收不再使用的变量，通常是通过判定程序是否能够通过变量之间形成的引用网络到达该变量。当确信变量是不可达的，就在它上面标上可以回收的记号，并且在回收器的下一次清理中（可能在未来的任意时刻）释放相关的内存。在托管语言中产生内存泄漏非常简单：只需使用完变量而忘记解除引用。

我们来考虑一个简单的例子，其中定义了一个描述家庭宠物及其主人的对象模型。首先看看主人，以 Person 对象描述：

```
function Person(name){  
    this.name=name;  
    this.pets=new Array();  
}
```

一个主人可以养一只或者多只宠物。当主人得到了一只宠物，他告诉宠物现在自己是它的主人：

```
Person.prototype.addPet=function(pet){  
    this.pets[pet.name]=pet;  
    if (pet.assignOwner){  
        pet.assignOwner(this);  
    }  
}
```

类似的，当主人从他的宠物列表中删除了一只宠物，他告诉宠物自己不再是它的主人：

```
this.removePet=petName=function{  
    var orphan=this.pets[petName];  
    this.pets[petName]=null;  
    if (orphan.unassignOwner){  
        orphan.unassignOwner(this);  
    }  
}
```

主人在任何时刻都知道谁是他的宠物并且能够通过提供的 addPet() 和 removePet() 方法来管理宠物列表。

主人在领养或不再领养宠物时都会通知该宠物，这基于一个假设，即每个宠物都会遵守这个契约（在 JavaScript 中，这个契约是隐含的，可以在运行时检查是否遵守了契约）。

宠物多种多样，在这里定义了两种：猫和狗。它们的区别在于对待被领养的态度上，猫并不在意被谁所领养，而狗一生都会伴随领养它的主人（我为这个普遍的观点向动物世界道歉！）。

因此宠物猫的定义看起来像是这样：

```
function Cat(name){  
    this.name=name;  
}  
  
Cat.prototype.assignOwner=function(person){  
}  
  
Cat.prototype.unassignOwner=function(person){  
}
```

猫对于是否被领养并不感兴趣，因此仅仅提供了契约方法的空实现。

另一方面，我们可以将狗定义为忠实地记得它的主人是谁，即使被遗弃了仍然保持对主人的“引用”（一些狗确实如此！）：

```
function Dog(name){  
    this.name=name;  
}  
  
Dog.prototype.assignOwner=function(person){  
    this.owner=person;  
}  
  
Dog.prototype.unassignOwner=function(person){  
    this.owner=person;  
}
```

Cat 和 Dog 对象都是 Pet 的行为恶劣的实现。作为宠物，它们严格依照契约的文字来实现，但是却没有遵循契约的灵魂。在 Java 或 C# 的实现中，我们可以明确地定义 Pet 接口，但是那样仍然不能阻止实现违背契约的灵魂。在现实编程世界中，对象建模者花费了大量的时间防止出现接口的行为恶劣的实现，尽力封堵所有可能被利用的漏洞。

我们来将这个对象模型具体化。在下面的脚本中，我们创建了三个对象：

(1) jim, 人(Person)

(2) whiskers, 猫(Cat)

(3) fido, 狗(Dog)

首先，我们实例化一个人 (Person) (第 1 步)：

```
var jim=new Person("jim");
```

接下来，我们给了那个人一只宠物猫 (第 2 步)。在对 addPet() 的调用中以内嵌方式实例化 whiskers，因此对猫的特定引用的寿命仅仅与方法调用的时间一样长。然而，jim 同样带有一个到 whiskers 的引用，因此只要 jim 是可达的，该对象就是可达的，也就是说，直到我们在脚本的最后删除 jim：

```
jim.addPet(new Cat("whiskers"));
```

我们也给了 jim 一只宠物狗 (第 3 步)。作为一个全局变量来声明，fido 比 whiskers 稍微多一点优势：

```
var fido=new Dog("fido");
jim.addPet(fido);
```

有一天，jim 送掉了他的猫 (第 4 步)：

```
jim.removePet("whiskers");
```

后来，他又送掉了他的狗 (第 5 步)。也许他移民了？

```
jim.removePet("fido");
```

我们对 jim 失去了兴趣并且释放了对他的引用 (第 6 步)：

```
jim=null;
```

最后，我们又释放了对 fido 的引用 (第 7 步)：

```
fido=null;
```

在第6步和第7步之间，我们可能相信已经通过设置`jim`为`null`摆脱了他。事实上，他仍然被`fido`引用并且仍然可以通过代码`fido.owner`到达。垃圾回收器无法将他释放，只能留下他潜伏在JavaScript引擎的堆空间里，占用着宝贵的内存。直到第7步，当`fido`声明为`null`时，`jim`才变成不可达的，随后内存才能被释放。

在简单的脚本中，这是一个很小的、临时性的问题，但是这个例子展示了表面上很随意的决定对于垃圾回收过程所产生的影响。`fido`可能在删除`jim`后没有被直接删除，并且，如果它拥有记住多于一个前任主人的能力，在销毁之前可能会将大批Person对象密封在堆空间中，使其过着暗无天日的生活。如果我们选择以内嵌方式声明`fido`并且将那只猫声明为全局变量，将不存在任何这类的问题。为了评估`fido`行为的严重性，我们需要问自己以下的问题：

(1) 当它引用其他已删除的对象时，将会消耗多少内存？我们知道头脑简单的`fido`一次只能记住一个Person，但是尽管如此，Person可能还包含500个其他仅能通过他自身才能到达的对宠物猫的引用，因此额外的内存消耗可能无法估量。

(2) 额外的内存消耗将会保持多长时间？在这个简单的脚本中，答案是“不是很久”，但是我们可能稍后会在删除`jim`和删除`fido`之间添加额外的步骤。而且，JavaScript开发者总是以事件驱动的方式编程，因此，如果删除`jim`和删除`fido`发生在分离的事件处理函数中，我们将很难预言一个确定的答案。如果不去做某种类型的用例分析，甚至都无法给出一个概率性的答案。

任何一个问题都不像它们看起来那么容易回答。我们能够做的，就是在编写和修改代码时，对这类问题保持关注，并且执行测试以验证我们的假设是否正确。当编写代码的时候，我们就应该考虑应用的使用模式，而不只是在事后追悔莫及。

以上内容覆盖了内存管理的通用原则。在Ajax应用中，还有一些需要注意的特定问题，我们接下来就对它们加以讨论。

8.3.2 Ajax的特殊考虑因素

到目前为止，我们已经覆盖了大多数编程语言通用的内存管理知识。当开发Ajax应用时，正确理解使用量和可达性的概念是很重要的，但是还有一些问题是特定于Ajax的。使用Ajax，我们运行在一个托

管的环境中，即在一个暴露了一些原生功能而锁定了其他功能使我们无法访问的容器中。这或多或少改变了整个场景。

在第 4 章中，Ajax 应用分成三个概念上的子系统：模型、视图和控制器。模型通常由可以自己来定义和实例化的纯 JavaScript 对象组成。视图主要由 DOM 节点组成，DOM 节点就是浏览器暴露给 JavaScript 环境的原生对象。控制器将两者粘合在一起。在这一层我们需要特别关注内存管理的问题。

1. 打破循环引用

在 4.3.1 节，我们为事件处理引入了一种常用的模式，在这种模式中我们将领域模型对象（即模型子系统的一部分）附加到 DOM 节点（即视图的一部分）上。我们来回顾一下以前展示的这个例子。这里是代表一个按钮的领域模型对象的构造函数：

```
function Button(value,domEl){  
    this.domEl=domEl;  
    this.value=value;  
    this.domEl.buttonObj=this;  
    this.domEl.onclick=this.clickHandler;  
}
```

注意，这里创建了一个 DOM 元素 domEl 和领域对象本身之间的双向引用。下面是在构造函数中引用的事件处理函数：

```
Button.prototype.clickHandler=function(event){  
    var buttonObj=this.buttonObj;  
    var value=(buttonObj && buttonObj.value) ?  
        buttonObj.value : "unknown value";  
    alert(value);  
}
```

回忆一下，事件处理函数调用时，将 DOM 节点而不是 Button 对象作为该函数的上下文。我们需要一个从视图到模型的引用，以便与模型层交互。在这个例子中，我们读取它的 value 属性。在本书使用这个模式的其他例子中，我们调用了领域对象上的函数。

Button 类型的领域模型对象是可达的，只要其他任何可达的对象拥有一个到它的引用。类似的，DOM 元素将保持可达，只要其他任何可达的元素引用了它。对于 DOM 元素来说，只要它附加到了主文档树上，一个元素总是可达的，甚至在没有任何可编程的引用指向它时也如此。因此，只要 DOM 元素仍然是文档的一部分，Button 就不可能被垃圾回收，除非我们明确地打破 DOM 元素和 Button 对象之间的连接。

当脚本实现的领域模型对象与 DOM 进行交互时，有可能创建一个局部的 JavaScript 对象，通过 DOM 而不是任何定义的全局变量来保持它可达。为了确保对象不会不必要地被垃圾回收器保留，我们可以编写简单的清理（clean-up）函数（这在很多方面都像是倒退成 C++ 对象的析构函数了，尽管我们需要手工调用它们）。对于 Button 对象来说，我们可以编写下列代码：

```
Button.prototype.cleanUp=function(){
    this.domEl.buttonObj=null;
    this.domEl=null;
}
```

第一行删除了 DOM 节点到这个对象的引用。第二行删除这个对象到 DOM 节点的引用。这个过程并不会销毁该节点，仅仅是将这个节点的局部引用重置为 null 值。在这里是将 DOM 节点作为构造函数的参数传递给对象，因此删除它并非是我们的责任。不过在其他情况下，我们确实有这个责任，我们来看看如何处理。

2. 移除 DOM 元素

在进行 Ajax 开发的时候，特别是在使用大型的领域模型时，通常的做法是构造新的 DOM 节点，并且通过程序与文档树进行交互，而不是仅仅使用在页面初次加载时通过 HTML 声明创建的 DOM 节点。例如，第 4 章、第 5 章的 ObjectViewer 和第 6 章的通知框架都包含了一些领域模型对象，它们可以通过创建额外的 DOM 元素并且将它们附加到主文档的一部分上来呈现自己。这样强大的能力也带来了巨大的责任，对于每个通过程序创建的节点，我们不得不同样通过程序来将它们移除。

无论是 W3C DOM 还是流行的浏览器实现中都没有提供在 DOM 节点创建后销毁该节点的方法。销毁已创建 DOM 节点的最佳方法是将它从文档树上分离，随后希望浏览器的垃圾回收机制能够发现它。

我们来考察一个直接的例子。下列脚本展示了一个简单的弹出消息框，关闭消息框的时候使用 DOM 的 `document.getElementById()` 方法找到自己：

```
function Message(txt, timeout){  
    var box=document.createElement("div");  
    box.id="messagebox";  
    box.className="messagebox";  
    var txtNode=document.createTextNode(txt);  
    box.appendChild(txtNode);  
    setTimeout("removeBox('messagebox')",timeout);  
}  
  
function removeBox(id){
```

```
    var box=document.getElementById(id);  
    if (box){  
        box.style.display='none';  
    }  
}
```

当调用 `Message()` 时，创建了一个可见的消息框，并且设置了一个 JavaScript 定时器，在一段给定的时间后，调用另外一个函数来删除这个消息框。

变量 `box` 和 `txtNode` 都在局部创建，当函数 `Message()` 退出时离开它们的作用域，但是因为创建的文档节点已经附加在了 DOM 树上，它们依然是可达的。

当不再需要已创建的 DOM 节点时，`removeBox()` 函数完成将它们移除的工作。从技术的观点来看，我们有几种可能的选项来做这件事情。在上面的例子中，我们简单地通过从视图中隐藏消息框来将它删除。消息框虽然不可见了，但它仍然占据着内存，如果我们计划稍后重新显示它，这就不是一个问题。

一种替代方法是，我们可以修改 `remove()` 方法使 DOM 节点从主文档上脱离开，并且希望垃圾回收器能够很快发现它们。同样，我们实际上并没有销毁变量，而且它在内存中逗留的时间也超出了我们的控制范围。

```
function removeBox(id){  
    var box=document.getElementById(id);  
    if (box && box.parentNode){  
        box.parentNode.removeChild(box);  
    }  
}
```

我们为这里移除 GUI 元素提出两种模式，它们分别称作隐藏移除（Remove By Hiding）和分离移除（Remove By Detachment）。这里的 Message 对象没有事件处理函数——它仅仅是很快地显示和消失。如果我们在领域模型和 DOM 节点之间建立双向连接（就好像对 Button 对象所做的那样），并且使用分离移除模式，我们就需要明确地调用 cleanUp() 函数。

两种方法都有优点和缺点。对于我们而言，主要的判断因素是问问稍后我们是否需要重用 DOM 节点。对于通用的消息框，答案可能是“是的”，因此应该选择隐藏移除模式。对于更加特定的用途，例如复杂的树 UI 组件的节点，在完成操作后简单地销毁该节点通常更加简单，而不是保存大量对隐藏节点的引用。

如果选择使用隐藏移除模式，我们能够采用一种互补的方法来重用 DOM 节点。在这里，我们修改消息创建函数，首先检查一个已存在的节点，而仅在必需的时候才创建新的节点。我们可以重写 Message 对象的构造函数来提供这个功能：

```
function Message(txt, timeout){  
    var box=document.getElementById("messagebox");  
    var txtNode=document.createTextNode(txt);  
    if (box==null){  
        box=document.createElement("div");  
        box.id="messagebox";  
        box.className="messagebox";  
        box.style.display='block';  
        box.appendChild(txtNode);  
    }else{  
        var oldTxtNode=box.firstChild;
```

```
    box.replaceChild(txtNode,oldTxtNode);

}

setTimeout("removeBox('messagebox')",timeout);

}
```

现在我们可以比较创建 GUI 元素的两种模式，它们分别称作始终创建（Create Always）（最初的例子）和不存在时创建（Create If Not Exists）（上面修改后的版本）。因为检查的 ID 是硬编码的，所以同一时间只能显示一条 Message（在这里也许是合适的）。我们将一个领域模型对象附加到一个可重用的 DOM 节点上，该领域对象可以用于获取到 DOM 节点的初始引用，允许不存在时创建模式与对象的多个实例共存。

注意 当编写 Ajax 应用时，理解与 DOM 元素相关的内存管理问题和理解自己创建的常规变量的内存管理问题一样重要。我们也要考虑到管理 DOM 元素的托管本质，并且以不同的方式来处理它们。当混用 DOM 节点和普通变量时，建议使用清理代码，用来打破循环引用。

在下一小节，我们将更加深入地考察，当使用 IE 时 Ajax 程序员需要考虑的问题。

3. 针对 IE 的更多特殊考虑因素

每一种 Web 浏览器都实现有自己的垃圾回收器，一些实现与其他的实现有所不同。IE 浏览器确切的垃圾回收机制不大容易理解，但是，按照 comp.lang.Javascript 新闻组中达成的一致意见，当在 DOM 元素与普通 JavaScript 对象之间存在循环引用时，IE 在释放这类变量时存在特殊的困难。按照他们的建议，手工切断这类连接是一个好主意。

为了使用例子来描述这个问题，下面的代码定义了一个循环引用：

```
function MyObject(id){

    this.id=id;

    this.front=document.createElement("div");

    this.front.backingObj=this;

}
```

MyObject 是一个用户自定义的类型。每个实例将使用 this.front 来引用一个 DOM 节点，并且该 DOM 节点将使用 this.backingObj 来反向引用该 JavaScript 对象。

为了使对象在执行 finalize 操作时移除这个循环引用，我们可以提供这样的方法：

```
MyObject.prototype.finalize=function(){
    this.front.backingObj=null;
    this.front=null;
}
```

通过将两个引用都设置为 null 来打破循环引用。

一种替代方法是，以一种通用的方式来对 DOM 树进行清理，即遍历 DOM 树并基于名称、类型或其他条件移除引用。Richard Cornford 提供了一个这样的函数，专门用于处理附加在 DOM 元素上的事件处理函数（参见本章“资源”一节）。

我感觉像这种通用的解决方法应该仅仅是作为最后的手段，对于以 Ajax 富客户端为代表的大型文档树来说，这种方案的可伸缩性很差。采用一种基于模式的方法来组织代码库，允许程序员对特定的、需要清理的情况进行跟踪。

对于 IE 来说，第二点值得一提的是一个顶层的正式文档未记载的函数，称作 `ColletGarbage()`。在 IE 6 中有这个函数，也能够被调用，但是似乎是空的，什么都没做。我们在任务管理器（Task Manager）中从未发现使用它之后内存报告有什么变化。

现在我们理解了内存管理的问题，接下来探讨一下测量的实用性，并且将这些测量方法应用到真实的应用中。

8.4 考虑性能的设计

我们首先应该明确，性能是由良好的执行速度和可控制的内存使用量共同组成的。我们曾经提到设计模式能够帮助我们达到这些目标。

在本节中，我们看看如何在真实应用中测量内存的使用量，随后使用一个简单的例子来说明，使用设计模式如何能够帮助我们理解内存使用量的变化，这种变化可以在真实的工作代码中看到。

8.4.1 测量内存使用量

当测量执行速度的时候，我们既可以在 JavaScript 代码中使用 Date 对象来做这件事，也可以使用外部工具来做这件事。JavaScript 并没有提供任何内建的读取系统内存使用量的能力，因此我们只能依赖于外部工具。幸运的是，我们已经有了几种可供选择的工具。

有多种方法可以查看在应用程序执行过程中浏览器占用了多少内存。最简单的方法是使用操作系统上适当的系统工具来查看运行的进程。在 Windows 系统上，可以使用任务管理器；而在 Unix 系统上可以使用基于控制台的 top 命令。我们依次介绍一下每一种工具。

1. Windows 的任务管理器

很多版本的 Windows 上都包含有 Windows 任务管理器（图 8-5）（Windows 95 和 98 用户没有这么幸运），它提供了在操作系统中运行的所有进程及其资源使用情况的视图。当用户按下 Ctrl+Alt+Delete 组合键时，通常可以从显示给用户的菜单中调用它。在任务管理器界面中包含了几个选项卡，我们感兴趣的昰名为进程的选项卡。

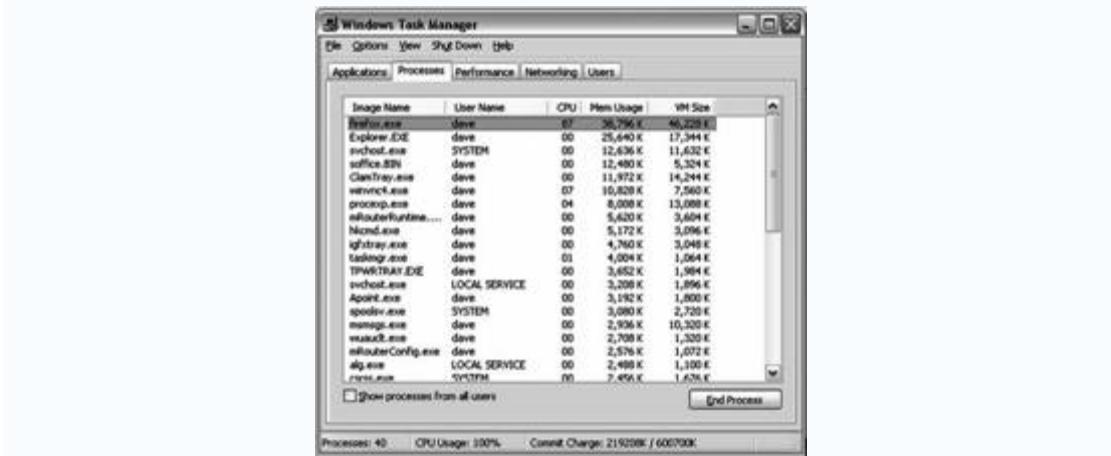


图 8-5 Windows 任务管理器显示了运行中的进程及其内存使用量。进程根据其内存使用量按降序排列

突出显示的行表明 Firefox 当前在计算机上使用了大约 38M 的内存。在默认状态下，内存使用 (Mem Usage) 一栏提供了应用所使用的活动内存的信息。在 Windows 的一些版本中，用户可以使用“查看→选择列”菜单栏添加附加的栏目（图 8-6）。

显示进程的虚拟内存大小 (Virtual Memory Size) 可能与显示进程的内存使用量 (Memory Usage) 同样有用。内存使用量代表分配给应用的活动内存，而虚拟内存大小代表已经写入到交换分区或文件中的

非活动内存。当最小化 Windows 应用时，内存使用量通常会明显下降，但是虚拟内存大小会或多或少地增加，表明应用仍然有可能在以后占用真实的系统资源。

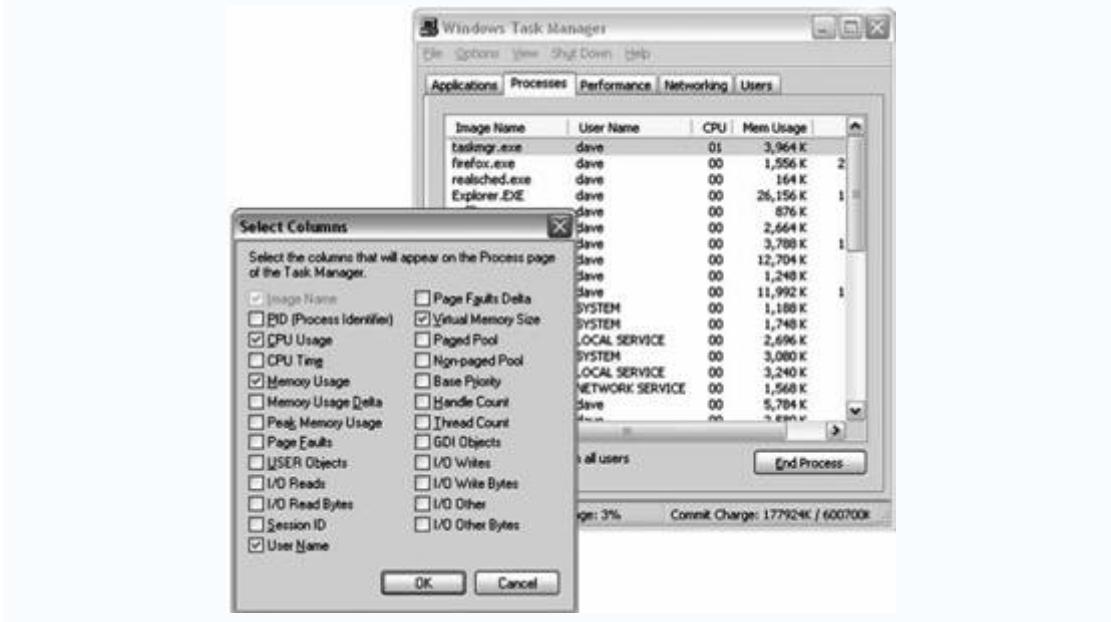


图 8-6 在任务管理器的进程标签中选择附加的栏目到视图中。虚拟内存大小显示了分配给进程的内存总数

2. Unix top

UNIX 系统（包含 Mac OS X）中基于控制台的应用 top 显示了一个和 Windows 任务管理器类似的视图（图 8-7）。

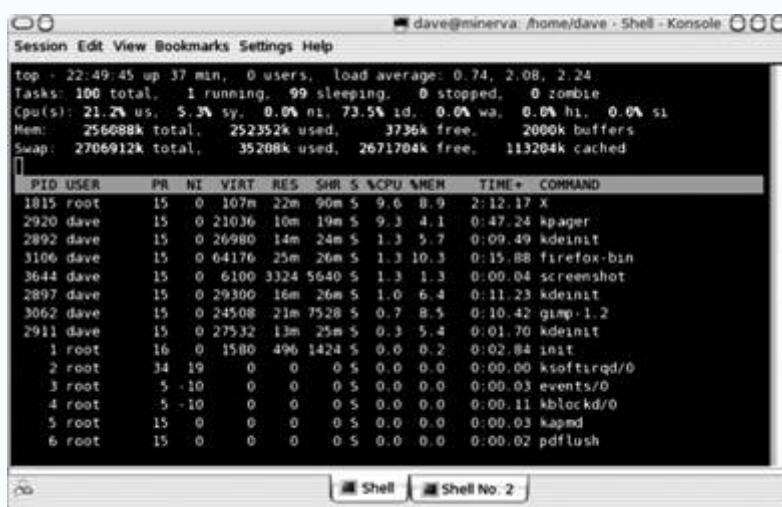


图 8-7 运行在控制台中的 UNIX top 命令，显示进程的内存和 CPU 的使用量

对于任务管理器来说，每一行代表一个活动的进程，包含了显示内存、CPU 使用量和其他统计信息的栏目。top 应用由键盘命令来驱动，在 man 格式或 info 格式的页面中，以及在因特网上都可以找到键盘命令的详细文档说明。篇幅有限，因此无法提供 top 的更全面的教程，或者来探讨它的图形界面等价工具，如在一些 UNIX/Linux 系统上可以找到的 GNOME System Manager。

3. 强大的工具

除了这些基本的工具以外，也可以用各种各样功能强大的工具跟踪内存的使用量，提供操作系统内部状态更加细致的视图。我们无法考察所有的这类工具，在这里仅仅简要介绍一下我们认为很有用的两个自由软件工具。

首先要介绍的是 Sysinternals.com 的 Process Explorer 工具（图 8-8）。这个工具也许可以描述为功能更加强大的任务管理器，它可以完成与任务管理器相同的工作，不仅如此，还可以显示单个进程的内存使用量和处理器使用量的细节，因此我们可以将 IE 或 Firefox 作为特定的目标来测量。

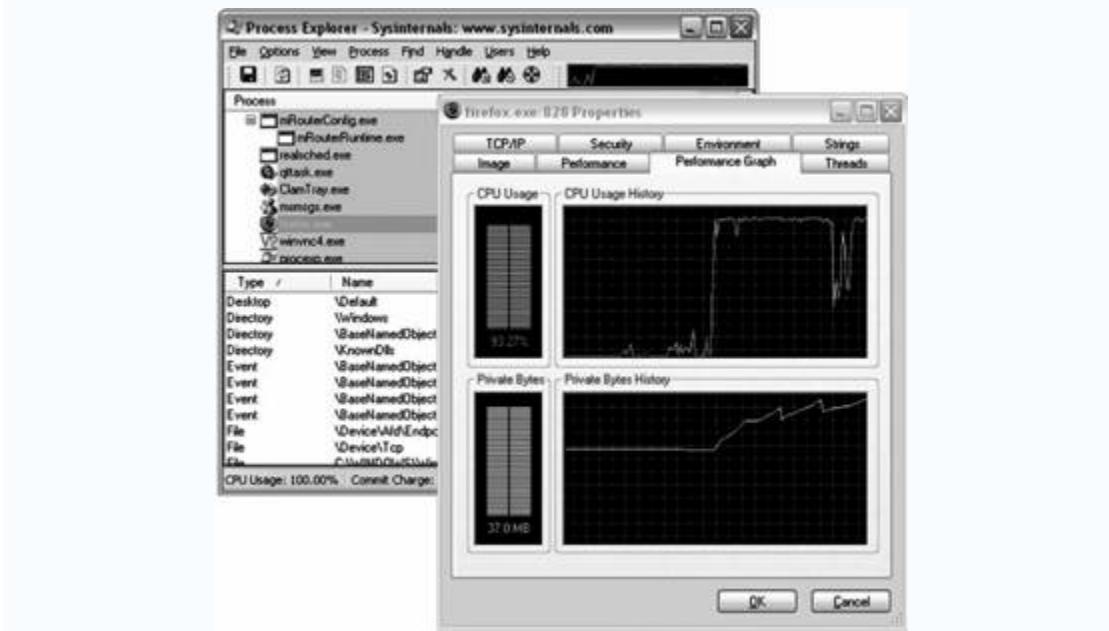


图 8-8 Process Explorer 以每个进程为基础提供了内存和处理器使用量的详细报告，在 Windows 机器上允许对浏览器资源使用情况进行更加精确的跟踪。这个窗口正在跟踪一个 Mozilla Firefox 实例，其中正在运行着 8.4.2 节中描述的压力测试

第二个要介绍的是 J. G. Webber 开发的 Drip（参见“资源”一节），用于 IE 的简单但是功能强大的内存管理报告器。它直接查询嵌入的 Web 浏览器，了解它所知道的 DOM 节点的信息，包括那些没有附加在文档树上的节点（图 8-9）。

	Refs	Tag	Id	Class
clickbox.html	1	DIV	box106	box1
clickbox.html	1	DIV	box2	box1
clickbox.html	1	DIV	box1	box1
clickbox.html	1	DIV	box3	box1
clickbox.html	1	DIV	box4	box1
clickbox.html	1	DIV	box5	box1
clickbox.html	1	DIV	box6	box1
clickbox.html	1	DIV	box63	box1
clickbox.html	1	DIV	box7	box1
clickbox.html	1	DIV	box8	box1
clickbox.html	1	DIV	box9	box1
clickbox.html	1	DIV	box64	box1
clickbox.html	1	DIV	box10	box1
clickbox.html	1	DIV	box107	box1
clickbox.html	1	DIV	box11	box1
clickbox.html	1	DIV	box12	box1
clickbox.html	1	DIV	box14	box1
clickbox.html	1	DIV	box108	box1
clickbox.html	1	DIV	box13	box1
clickbox.html	1	DIV	box109	box1
clickbox.html	1	DIV	box110	box1

图 8-9 Drip 工具可以查询 IE 的 DOM 树的内部状态的细节

然而，即使仅使用那些基本的工具，我们也可以发现 Ajax 应用运行过程中的大量状态信息。

到目前为止，我们通过一些代码片断分别考察了处理性能问题的单个模式和习惯用法。当编写适当规模的 Ajax 应用时，各种模式和习惯用法会在每个子系统中以惊人的方式相互影响。接下来的小节描述了一个案例，展示了理解模式之间如何相互组合的重要性。

8.4.2 简单示例

到目前为止，我们已经探讨了内存管理的理论，并且描述了当通过程序创建界面元素时可能有用的几个模式。在实际的 Ajax 应用中，我们会用到一些相互交互的模式。单个模式对性能有影响，模式之间交互的方式同样也会对性能有影响。在这里使用通用的词汇来描述代码正在做什么变得非常有价值了。阐明这个原则的最佳方式是通过例子，我们将在本节中介绍一个简单的例子，展示使用不同的模式组合对性能带来的影响。

在这个简单的测试程序中，我们能够重复创建和销毁小的 ClickBox UI 组件，取这个名字是因为它们是用户可以通过鼠标点击的小方框。UI 组件本身包含了有限的行为，通过下列代码来描述：

```
function ClickBox(container) {
    this.x=5+Math.floor(Math.random()*370);
    this.y=5+Math.floor(Math.random()*370);
    this.id="box"+container.boxes.length;
    this.state=0;
    this.render();
    container.add(this);
}
```

```

ClickBox.prototype.render=function(){
    this.body=null;
    if (this.body==null){
        this.body=document.createElement("div");
        this.body.id=this.id;
    }
    this.body.className='box1';
    this.body.style.left=this.x+"px";
    this.body.style.top=this.y+"px";
    this.body.onclick=function(){
        var clickbox=this.backingObj;
        clickbox.incrementState();
    }
}

ClickBox.prototype.incrementState=function(){
    if (this.state==0){
        this.body.className='box2';
    }else if (this.state==1){

        this.hide();
    }
    this.state++;
}

ClickBox.prototype.hide=function(){
    var bod=this.body;
    bod.className='box3';
}

```

首次呈现时，这些 ClickBox 显示为红色；点击一次后，它们变为蓝色；再点击一次会将它们从视图中移除。

如同前面讨论的，这种行为是通过在领域模型对象和 DOM 元素之间建立双向引用来实现的。

从编程的角度来看，每个 ClickBox 由唯一 ID、位置、内部状态的记录（即接收到了多少次点击）以及主体部分组成。主体部分是一个类型为 DIV 的 DOM 节点。DOM 节点在名为 backingObj 的变量中保存一个到后端对象的引用。

同时还定义了一个容纳 ClickBox 对象的 Container 类，其中保存了一个对象数组，以及它自己唯一的 ID。

```

function Container(id){
    this.id=id;
    this.body=document.getElementById(id);
    this.boxes=new Array();
}

Container.prototype.add=function(box){
    this.boxes[this.boxes.length]=box;
    this.body.appendChild(box.body);
}

```

```

Container.prototype.clear=function(){
    for(var i=0;i<this.boxes.length;i++){
        this.boxes[i].hide();
    }
    this.boxes=new Array();
    report("clear");
    newDOMs=0;
    reusedDOMs=0;
}

```

应用的截屏如图 8-10 所示。

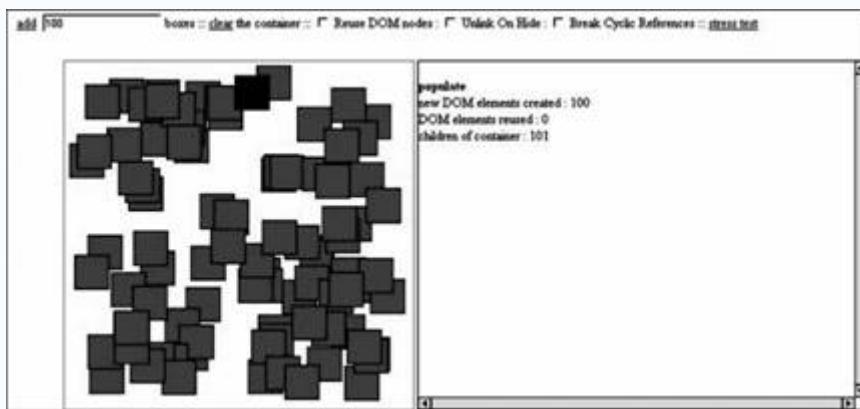


图 8-10 演示内存管理的应用，创建了 100 个 UI 组件后的景象。用户刚刚使用鼠标点击了其中的一个 UI 组件

右边的调试面板报告了系统在发生不同的用户事件（例如从容器中添加或者移除 UI 组件）之后的内部状态。

编写的代码允许应用在运行过程中，在创建和销毁 DOM 元素和循环引用的不同模式之间进行切换，用户可以在运行时通过选中或不选中页面上的 HTML 表单元素在它们之间进行选择。当激活容器内添加或移除 ClickBox 的链接时，用于实现用户界面的模式组合将会与选择框的状态相匹配。我们来检查一下每个选项以及相应的代码。

1. Reuse DOM Nodes 复选框

选中这个选项将确定 ClickBox UI 组件是否在创建自己时尝试找到一个已存在的 DOM 节点，以及仅仅将创建一个新的 DOM 节点作为最后的手段。这允许应用在始终创建模式和不存在时创建模式之间进行切换，我们在 8.3.2 节中讨论过这两个模式。修改后的呈现代码如下所示：

```

ClickBox.prototype.render=function(){
    this.body=null;
    if (reuseDOM){
        this.body=document.getElementById(this.id);
    }
    if (this.body==null){
        this.body=document.createElement("div");
        this.body.id=this.id;
        newDOMs++;
    }else{
        reusedDOMs++;
    }
    this.body.backingObj=this;
    this.body.className='box1';
    this.body.style.left=this.x+"px";
    this.body.style.top=this.y+"px";
    this.body.onclick=function(){
        var clickbox=this.backingObj;
        clickbox.incrementState();
    }
}

```

2. Unlink On Hide 复选框

当从容器中移除 ClickBox 时（无论是通过另一个点击还是通过调用 Container.clear()），这个选项可以决定是使用隐藏移除模式还是使用分离移除模式（参见 8.3.2 节）：

```

ClickBox.prototype.hide=function(){

    var bod=this.body;

    bod.className='box3';

    if (unlinkOnHide){

        bod.parentNode.removeChild(bod);

    }

    ...

}

```

3. Break Cyclic References 复选框

当移除 ClickBox UI 组件时，这个选项决定是否将 DOM 元素和后端对象之间的引用重置为 null，为使 IE 垃圾回收器满意，使用打破循环引用模式：

```

ClickBox.prototype.hide=function(){

    var bod=this.body;

    bod.className='box3';

```

```
if (unlinkOnHide){  
    bod.parentNode.removeChild(bod);  
}  
  
if (breakCyclics){  
    bod.backingObj=null;  
    this.body=null;  
}  
}
```

表单控件允许用户将 ClickBoxes 添加到容器中，随后清理容器。应用可以通过手工来驱动，但是在这里为了收集结果，我们也编写了一个压力测试函数用来模拟一些手工操作。这个函数运行自动的操作序列，下列操作序列被重复了 240 次：

- (1) 使用 populate() 函数添加 100 个 UI 组件到容器中。
- (2) 添加另外 100 个 UI 组件。
- (3) 清理容器。

stressTest 函数的代码如下所示：

```
function stressTest(){  
    for (var i=0;i<240;i++){  
        populate (100);  
        populate(100);  
        container.clear();  
    }  
    alert("done");  
}
```

注意，这里测试的功能与向容器元素中添加和移除节点有关，而与点击时单个 ClickBox 的行为无关。

这个测试故意设计得很简单。如果仅仅为了看到当发生变化时内存使用量是否会上升或下降，我们建议你为自己的应用开发更加简单的压力测试。设计测试脚本本身就是一门艺术，需要理解典型的使用模式，并且可能还需要理解不止一种类型的使用模式。

运行压力测试花了一分多钟，在此过程中浏览器没有响应用户的输入。如果增加迭代的次数，浏览器可能会崩溃；如果使用的迭代太少，内存使用量的变化可能就不会很显著。我们发现 240 次迭代对于运行测试的机器是一个合适的值。在你的机器上合适的值可能会相当地不同。

记录内存使用量的变化是相对简单的事情。我们在 Windows 操作系统上运行测试，任务管理器保持打开状态。我们观察 `iexplore.exe` 在加载测试页面后的内存占用量，然后观察在表明测试完成的警告框显示后的内存占用量。`top` 或者其他类似的工具能够用来在 UNIX 上进行测试（参见 8.4.1 节）。在每次运行测试后都要将浏览器完全关闭，以消除任何内存泄漏，确保每次运行都是从相同的基准开始的。

以上是测试方法的介绍，在下面的小节，我们来看看执行这些测试的结果。

8.4.3 结果：如何将内存使用量缩减 150 倍

按照 Windows 任务管理器的报告，在不同的模式组合下运行刚才描述的压力测试，产生了极其不同的内存消耗值，结果总结在表 8-4 中。

表 8-4 ClickBox 例子代码的基准测试结果

ID	重用 DOM 节点	移除隐藏	打破循环引用	最终的内存使用量 (IE)
A	N	N	N	166MB
B	N	N	Y	84.5MB
C	N	Y	N	428MB
D	Y	N	N	14.9MB
E	Y	N	Y	14.6MB
F	Y	Y	N	574MB
G	Y	Y	Y	14.2MB

压力测试程序运行相当寻常的 Windows 2000 Workstation (2.8GHz 处理器, 1GB 内存) 的 IE 6 中，使用了多种不同模式组合，得到了表 8-4 中显示的结果。所有例子的初始内存使用量大约都是 11.5MB，所有的内存使用量报告都来自任务管理器的进程标签中的内存使用栏（参见 8.4.1 节）。

当第一次面对真实数字的时候，我们注意到的第一件事情是应用占用了大量的内存。Ajax 经常被描述

为一种瘦客户端解决方案，但是如果我们将各种编码错误组合在一起，Ajax 应用完全有可能占用大量的内存！

关于结果的第二个重要观点是，设计模式的选择对于内存使用量产生了剧烈的影响。我们来详细看看结果。在对所有的 ClickBox UI 组件做完呈现和隐藏之后，有三种组合占用了不到 15MB 的内存。剩余组合的内存使用量向上攀升，从 80MB、160MB 到了令人惊愕的峰值 430MB 和 580MB。假设浏览器本身占用了 11.5MB 内存，额外的内存占用大小从 3.5MB 变化到 570MB——仅仅使用了不同的设计模式组合，就产生了超过 150 倍的差别。值得注意的是产生了如此巨大内存泄漏的浏览器仍然能够继续运行。

不能认定是哪一种特定的模式造成了这种结果，因为设计模式之间的交互相当复杂。例如，对运行 A、D 和 F 作比较，切换到重用 DOM 模式导致内存使用量大为减少（超过 90%），但是同时切换到移除隐藏模式却产生了三倍的增长！在这种特定的情况下，原因是很容易理解的——因为移除了 DOM 节点，通过调用 `document.getElementById()` 找不到可以重用的 DOM 节点。类似地，与基本的情况作比较（与运行 C 和 A 作比较），单独切换到移除隐藏模式增加了内存使用量。再看看运行 E 和 G，将移除隐藏模式这个贪婪的内存占用者排除在外——在正确的上下文中，它们所引起的内存使用量的增长是很小的。

有趣的是，并不存在明显的优胜者，三组大不相同的组合都只产生了很小的内存增长。所有这三种组合都使用了重用 DOM 节点模式，然而产生最大内存增长的组合同样也使用了这个模式。我们无法从中得出一个简单的结论，但是能够识别出哪些模式组合能够很好地协同工作，而哪些组合不行。如果我们理解了这些模式并且为它们命名，那么整个应用程序将始终一致地应用这些模式，而且得到可靠的性能也容易得多。如果我们不是使用一组固定的模式，而是很随意地为每个子系统的 DOM 生命周期编码，那么编写每一段新的代码都是在赌博，可能会引入大量的内存泄漏，也可能不会。

这个基准测试练习覆盖了开发在 Web 浏览器中能够长期正常运行的 DHTML 富客户端时涉及的各种问题，它能够找出会发生错误的位置，这些错误既包括一些通用的错误，也包括发生在本书其他地方讨论过的一些设计模式中的错误。

为了更好地解决内存问题，你必须在开发方法中为它们留有一席之地。经常问自己当向代码中引入变化时将对内存使用量产生什么影响，并且在实现这些变化的过程中经常测试内存使用量。

由于类似的内存问题总是以相同的模式重复出现，在这里为代码库应用一种基于模式的方法是很有帮助的。例如，后端对象在 DOM 节点和非 DOM 节点之间创建了循环引用；分离移除模式会干扰不存在时

创建模式。如果有意识地在设计中使用模式，我们就不大可能陷入这类问题。

编写和维护自动测试脚本，并且使用这些脚本来运行基准测试是很有帮助的。编写测试脚本可能是最难的部分，因为其中需要了解用户如何使用应用。你的应用可能拥有多种类型的用户，在这种情况下，你应该开发多个测试脚本而不是只开发单个平均化的测试脚本，这样的单个脚本无法代表任何一种类型的用户。对于任何类型的测试，代码一旦写好，不应该认为它们是一成不变的，而应该随着项目的发展进行积极的维护。

8.5 小结

任何计算机程序的性能都是执行速度和资源使用量的结合。对于 Ajax 应用来说，我们工作在一个高度托管环境中，远离操作系统和硬件，但是基于我们的编码方式，仍然有可能极大地影响应用的性能。

我们通过使用 JavaScript 库和原生的性能分析器工具（例如 Venkman 调试器）介绍了性能分析的实践。性能分析帮助我们了解系统的瓶颈所在，并且为我们能够测量到的变化提供了基准。通过比较代码变化前后的性能分析结果，我们可以评估代码变化对于应用的整体执行速度产生的影响。

我们也通过一些常见的不良实践，以及运行 DOM 或 IE 的一些特定问题之间的冲突，考察了内存管理的问题，并且展示了如何避免在代码中引入内存泄漏。我们也看到了如何使用 Windows 和 UNIX 操作系统上可用的工具来测量内存的消耗。

最后，基准测试的例子展示了这些细节问题会给代码带来的真实影响。设计模式是至关重要的，因为它们可以识别出哪里存在着巨大的内存使用量差异，以及如何对此加以管理。

性能是一个难以捉摸的指标——永远存在进一步优化的余地，并且我们应该在 Ajax 应用中采用注重实效的方法，以便获得“足够出色”的性能。本章应该已经为你提供了达到这一目标所需要的工具。

8.6 资源

在本章中，我们考察了一些有用的开发工具。

Drip, IE 内存泄漏检测器, 由 Joel Webber 开发。他的 blog 为 <http://jgwebber.blogspot.com/2005/05/drip-ie-leak-detector.html>, 但已经不可访问, 而 Drip 目前仍然可以从 www.outofhanwell.com/ileak/ 找到。

Venkman 性能分析器: www.svendtofte.com/code/learning_venkman/advanced.php#profiling。

Process Explorer: www.sysinternals.com。

关于 IE 内存泄漏的官方描述以及一些变通的解决方案, 可以在这里找到: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/IETechCol/dnwebgen/ie_leak_patterns.asp。Richard Cornford 所建议的解决方案可以通过在 Google Groups 中搜索“cornford javascript fix-CircleRefs()”找到——完整的 URL 太长了, 很难在这里给出。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真题解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资料下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引