

第 1 章

一种新的 Web 设计方法

本章内容

- 异步网络交互和使用模式
- Ajax 与传统 Web 应用的关键区别
- Ajax 的四个基本原则
- 真实世界中的 Ajax

理想的用户界面是无形的。需要的时候，召之即来，所有想要的功能一应俱全；不需要的时候，挥之即去，用户可以不受干扰地继续专注于手头的问题。然而，这样的用户界面总是可遇而不可求的。日复一日，用户只能顺从于那些远未尽如人意的用户界面，并且慢慢习惯了这些低劣的应用。直到某一天，有人给我们展示了一种更好的方法，这时候我们才意识到，过去使用的方法是多么地落伍。

当今的因特网界也正在认识到，用于显示文档内容的基本 Web 浏览器技术，已经无法胜任更高层面的任务了。

Ajax (Asynchronous JavaScript + XML，即异步 JavaScript + XML) 是一个相当新的名字，是由 Adaptive Path¹ 的咨询顾问 Jesse James Garrett 首先提出来的。Ajax 的一些部分，以前称作动态 HTML (Dynamic HTML) 和远程脚本 (remote scripting)。Ajax 的名字更加简洁，而且它容易让人联想起很多东西：洗衣粉（高露洁的 Ajax 牌洗衣粉）、荷兰的足球队（知名的阿贾克斯球队）、希腊英雄（疯狂的埃阿斯）等等。

不仅名字好听，无论是从技术还是从商业角度，Ajax 同样让人耳目一新。技术上，Ajax 极大地发掘了 Web 浏览器的潜力，开启了大量新的可能性。商业上，Google 和其他主要的参与者通过眼花缭乱的 Ajax 应用，让普通大众对于新一代的 Web 应用充满了期待。

越来越复杂的基于 Web 的新型服务，不断冲击着我们已经习惯了的传统 Web 应用。其实有一大堆技术可以提供表现能力更加丰富、更加智能或者在其他方面更好的客户端应用，但 Ajax 却只需要使用一些在绝大多数现代电脑上既有的技术，就能举重若轻地完成同样的任务。

也就是说，在 Ajax 中，采用的是一系列已有的甚至是老旧的技术，把它们重新锻造、延伸，

1. 它是一家业界知名的用户体验咨询公司，网址为 <http://www.adaptivepath.com>。——译者注

超越其原有的概念，让我们可以应付客户端程序所要面对的复杂情况。本书不仅会分别考察每一种技术，而且也会从整体上考察大型的Ajax项目。读完本书，你会对此有一个完整的理解。此外，书中还穿插介绍了很多Ajax设计模式。众所周知，设计模式有助于获取知识和经验，也帮助我们有效地与他人沟通。遵循设计模式所倡导的编程规律，我们可以更容易地建造出当需求变化时易于修改、便于扩充的应用。这些设计模式甚至会让你的工作更有乐趣。

1.1 为什么需要 Ajax 富客户端？

建造一个富客户端¹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，

1. Rich Client，译作富客户端，这里“富”的意思是“表现能力丰富的”，相关的术语还有 RIA（Rich Internet Application），译作富因特网应用。——译者注

4 第一部分 重新思考 Web 应用

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。

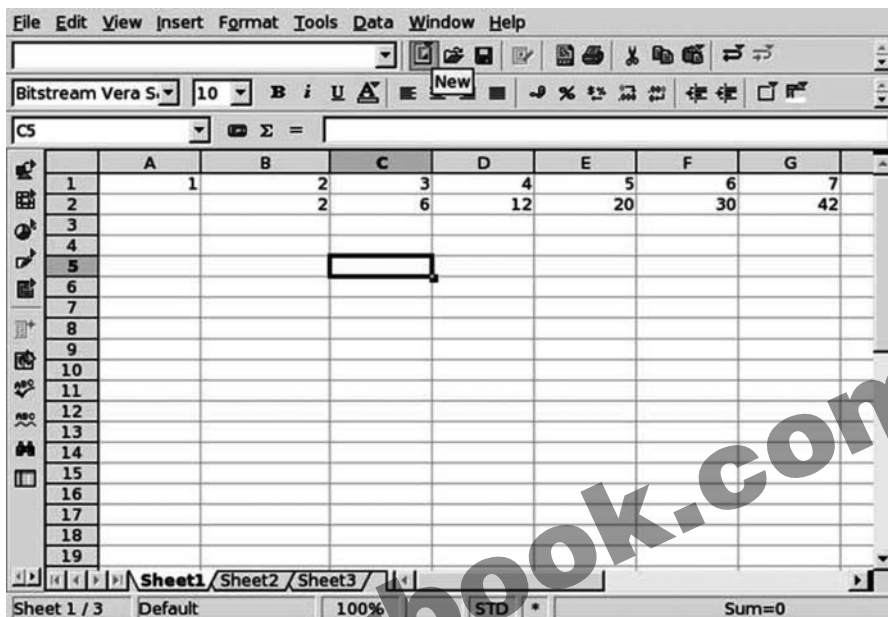


图 1-1 这个桌面电子表格应用展现了用户交互的众多可能性。被选中单元格的行列标题都是突出显示的；按钮在鼠标移上去的时候会显示提示信息；工具栏上排列着各种丰富的控件；单元格也可以交互地查看和编辑

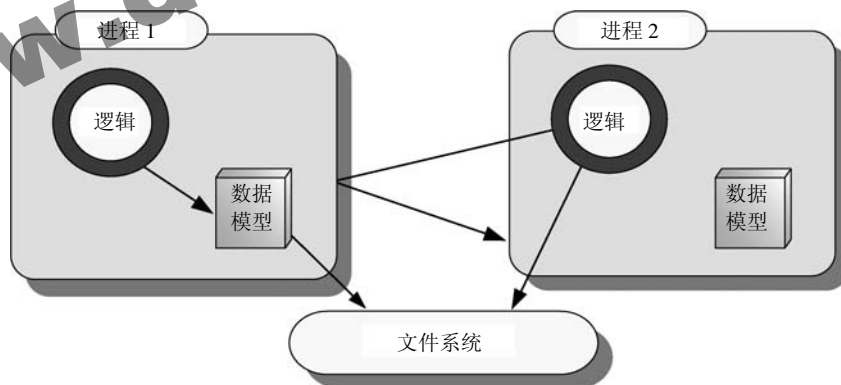


图 1-2 一个独立的桌面应用的架构示意图。应用运行在它自己的独立进程之中，数据模型和程序逻辑能够彼此“看到”对方的存在。除了通过文件系统之外，同一个应用的第二个运行实例¹没有办法访问到第一个运行实例的数据模型。通常来说，整个程序的状态都保存在单个文件中，当应用运行的时候，这个文件会被加锁以阻止其他的进程同时访问

1. 即进程。——译者注

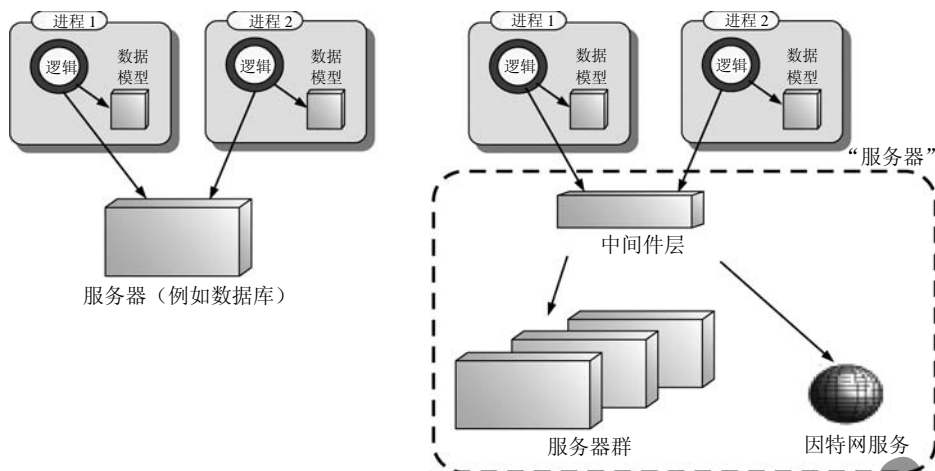


图 1-3 客户/服务器系统和 n 层架构的示意图。服务器提供了共享的数据模型，客户端与该数据模型交互。客户端同时还维护数据模型的一部分，以获得快速的访问，但是它将服务器端的模型当作业务领域对象的最终表示。多个客户端可以与同一个服务器交互，当然，这需要有合适的资源锁定机制和合理的对象（或者数据行）隔离措施作为保证。服务器可以是单进程的，就像在 20 世纪 90 年代早期和中期传统的客户/服务器模型中一样，也可以是由很多个中间件层或者多个 Web 服务等组成。在任何一种情况下，从客户端的角度来看，服务器都有一个单独的接入点，可以看作是一个黑盒

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中，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。

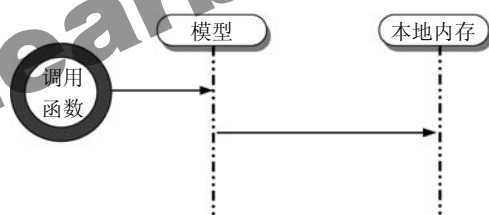


图 1-6 本地过程调用的顺序图。参与者很少，因为程序逻辑与数据模型都保存在本地内存中，并且彼此可以直接访问

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，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。

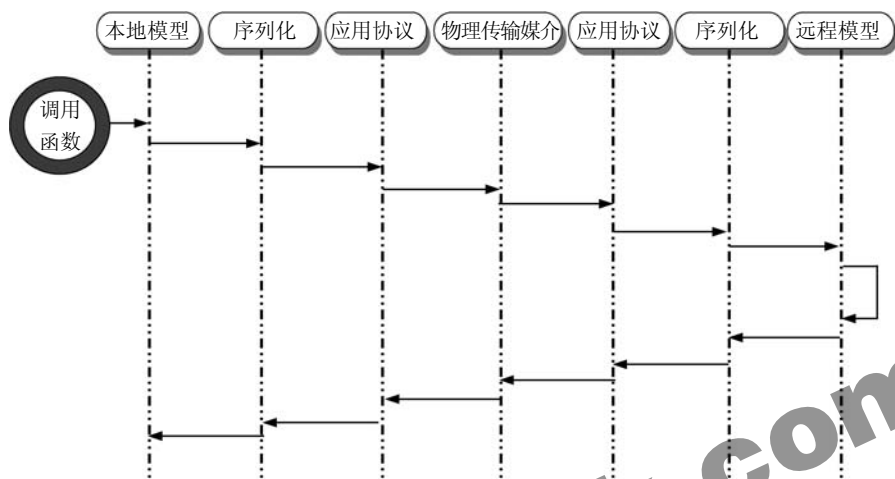


图 1-7 远程过程调用的顺序图。一台机器的程序逻辑尝试操作另外一台机器上的数据模型

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）。

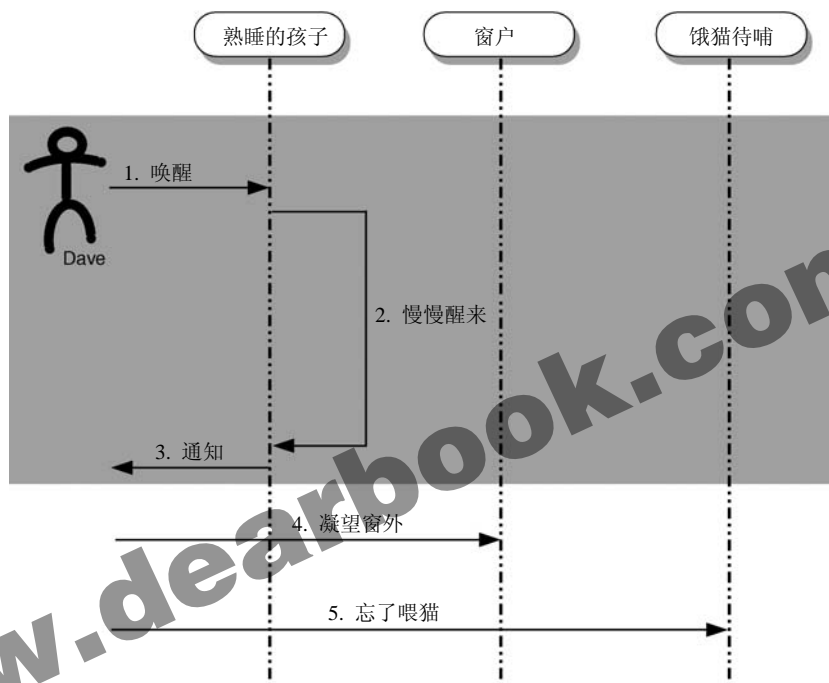


图 1-8 在我每日早晨必做的事中，以同步方式响应用户输入的顺序图。顺序图中纵向表示时间的流逝，其中的阴影区域表示了我被阻塞不能接受其他输入的时间长度

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¹，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。

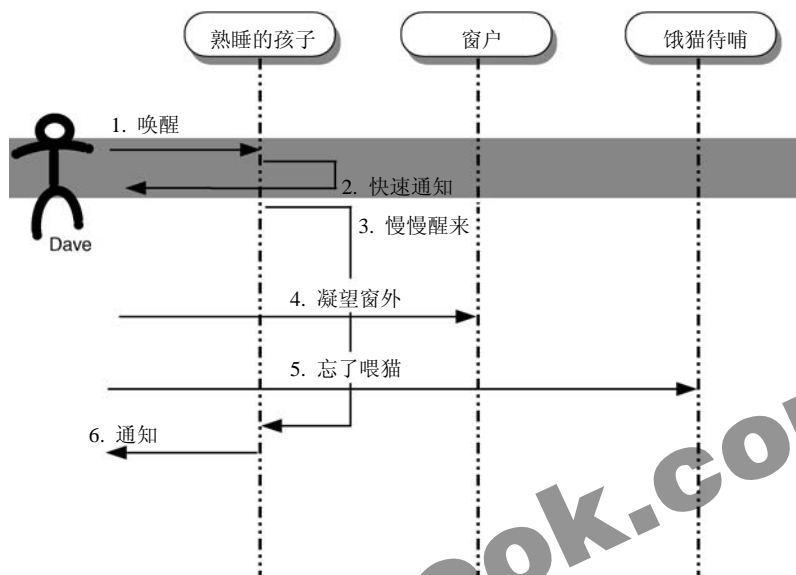


图 1-9 以异步方式响应用户输入的顺序图。如果遵循异步的输入模式，我可以让孩子们在醒来的时候通知我。在他们缓慢的起床过程中，我可以继续从事其他活动，这使得我被阻塞的时间大大缩短

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

1. 这么做的目的是确保线程被成功地创建和启动。——译者注

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）¹。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

1. 实际上，Alan Cooper 从行为的立场出发，总共定义了四种软件姿态（software posture），分别是：独占（sovereign）、瞬态（transient）、精灵（daemonic）和辅助（auxiliary）。后两者对于 Ajax 应用来说，意义尚不明确，所以此处并未提及。——译者注

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¹。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。

1. 即 Ajax 是使得 Web 恢复健康发展的良药。——译者注



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¹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。

1. 哑终端即指不具有智能的终端。——译者注

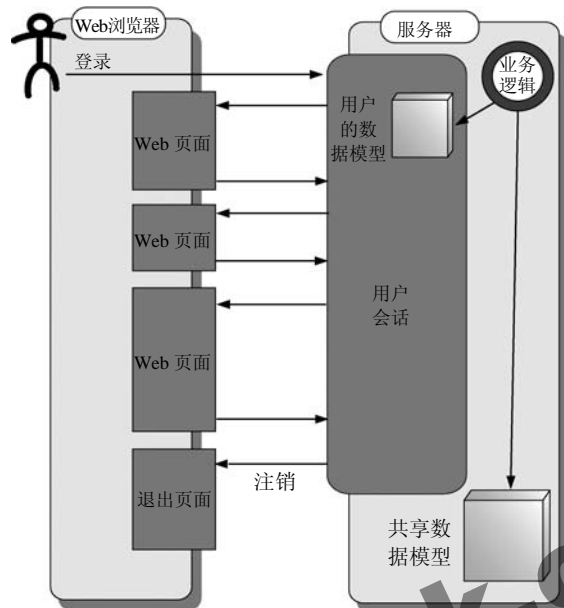


图 1-11 传统 Web 应用的生命周期。用户和应用会话的所有状态都保留在 Web 服务器上。用户在会话中看到的是一系列的页面，每次页面切换都不可避免地要到服务器上走一个来回

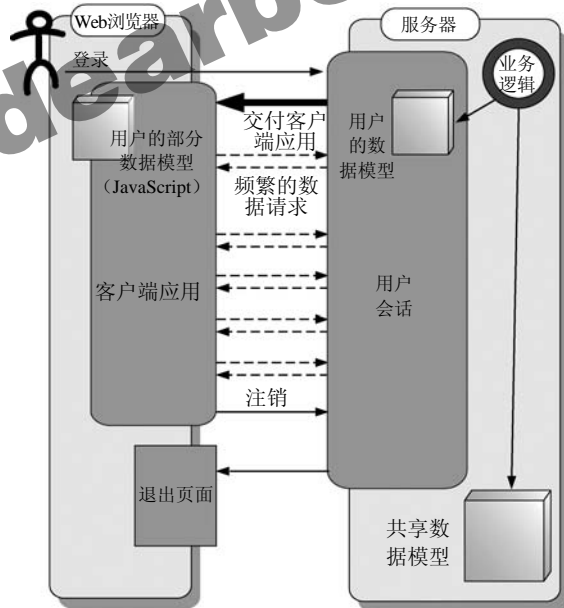


图 1-12 Ajax 应用的生命周期。用户登录后，服务器交付一个客户端应用给浏览器。这个应用可以独立处理很多的用户交互，对于自己无法独立处理的交互，应用会以后台方式发送请求给服务器，而不会打断用户的操作流程

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¹。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。

1. 即保存在本地内存中。——译者注

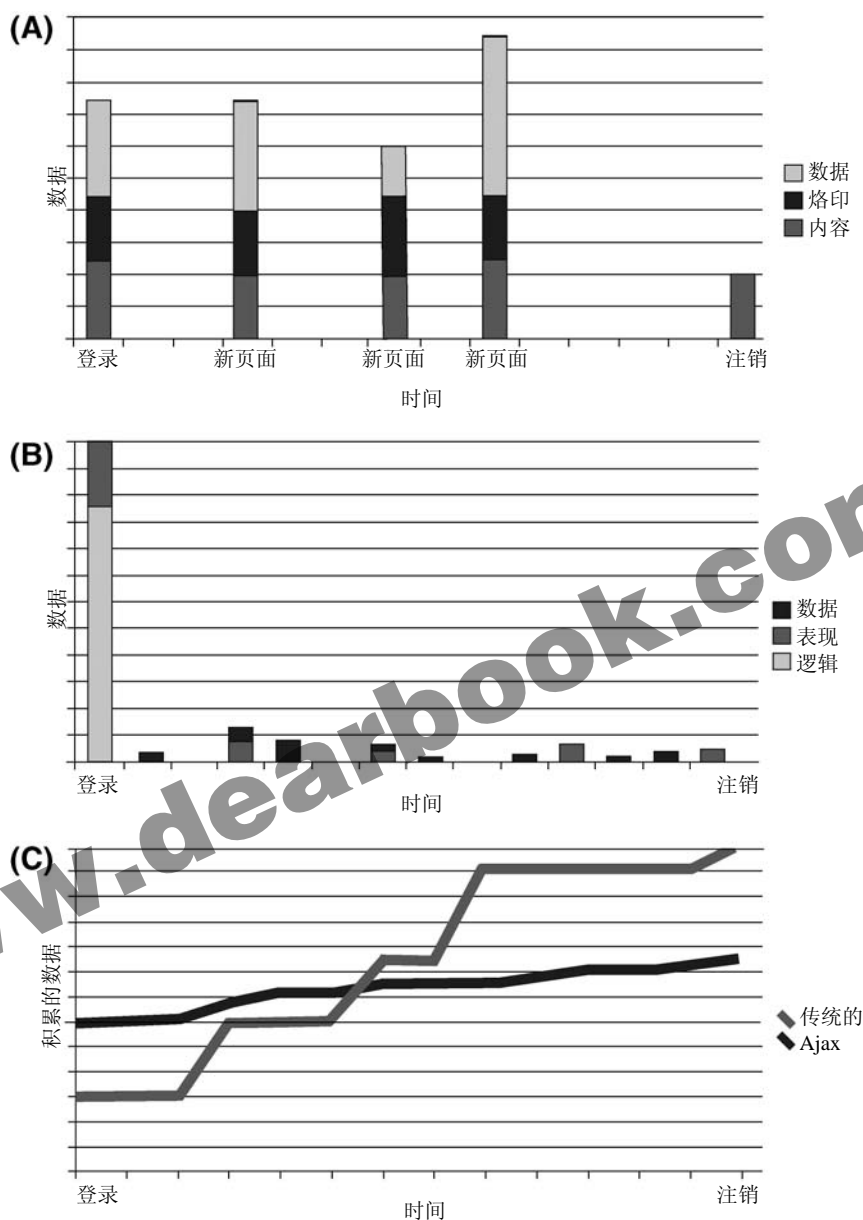


图 1-13 细分服务器发送的内容, (A)是传统的 Web 应用, (B)是 Ajax 应用。 (C)表示随着应用使用时间的延长, 累积的网络流量的增长情况

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¹，那肯定是不可接受的。

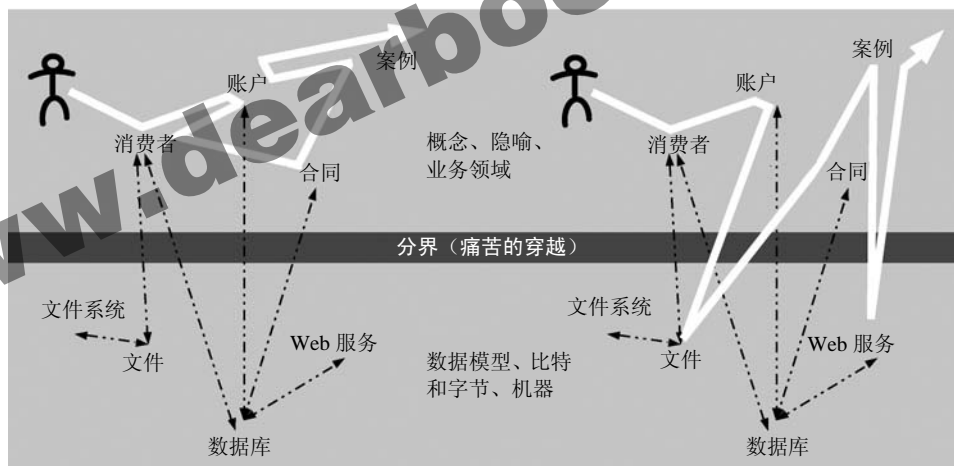


图 1-14 处理事件打断了用户的工作流程。用户要处理与业务相关的和与计算机相关的两种对象。这迫使用户频繁地在这两者之间切换，从而导致用户注意力分散，工作效率降低

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

1. 如果这种打断还是非常频繁的，所累积的效率损失将是巨大的。——译者注

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 (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¹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. 通常是指那些受过严格编程训练的开发者，特别是 J2EE 和 .NET 开发者，更是自觉得高人一等。——译者注

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, 即使用户正在写邮件, 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¹、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²。

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来建造真实的应用的。

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,

1. 有两方面的含义, 即任何的操作系统和任何的现代 GUI 浏览器。——译者注

2. 企业邮件服务的用户通常需要在客户端机器上安装一堆 ActiveX 控件, 然后才能使用这些服务, 而 ActiveX 的安全性受到了广泛的指责。——译者注

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。

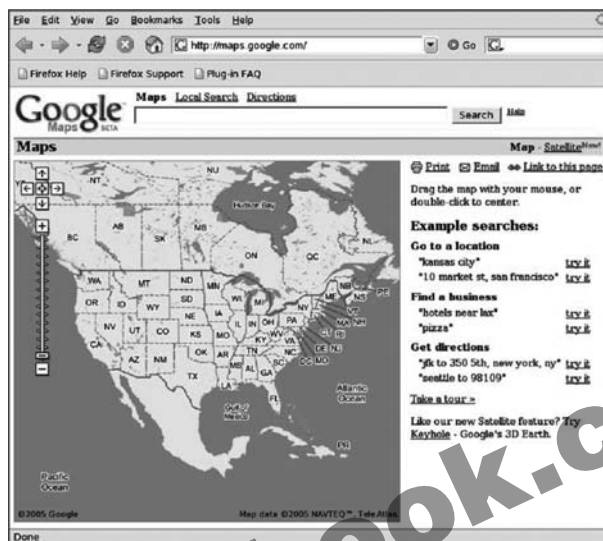


图 1-15 Google Maps 的首页提供了一个可以滚动和放大的美国地图，还有熟悉的 Google 搜索条。要注意的是，缩放控件是在地图之上而不是在它的旁边，这使得用户的视线无需离开地图就可以进行缩放控制。



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 DHTML 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¹。

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²）。作为 Google 所推出的一个最卓越的展示品³，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⁴，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

-
1. 作为对比，读者可以亲自去尝试一下微软公司所提供的的旧的地图服务 <http://terraserer.microsoft.com>。这个服务很有可能很快被废弃，因为微软公司推出了新的地图服务 <http://local.live.com>。新的地图服务，包括这个网站上的很多其他服务，都大量采用了 Ajax 的技术来建造。——译者注
 2. Google Maps 采用的方式是使用 JavaScript 动态创建 img 元素，以异步方式从服务器请求当前所需要的图片。浏览器直接缓存每一个图片，因此是最有效率的实现方式。——译者注
 3. 由此所带来的并发的访问量是极其巨大的。——译者注
 4. Google 同样有强大的后端地图数据的支持，甚至比很多竞争对手都要强。——译者注

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。Flash 电影是交互式的，它使用 ActionScript 来编程（ActionScript 是 JavaScript 的一种近亲¹）。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²。

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³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⁴，用户可以选择打开全部访问权限，允许应用访问文件系统、网络端口以及其他的本地资源。

1. ActionScript 和 JavaScript 都是标准化的 ECMAScript 所派生出的脚本语言，语法上几乎完全相同。——译者注

2. 套用一句歌词：不是我不明白，这世界变化快！仅仅过了不到半年，作者的这段话现在看来也过时了。目前最新的 Firefox 1.5 已经具有原生的 SVG 支持，而不需要用户额外安装任何插件，补充了 Ajax 技术领域的这片空白。

但是 IE 目前还没有类似的支持，Firefox 浏览器在 Ajax 技术发展的道路上扮演着开路先锋的角色。——译者注

3. 需要在浏览器上安装 Java 运行环境的插件。——译者注

4. 数字签名用来标识创建者的身份，以便有效地防止网络犯罪。——译者注

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¹，而对于一个公共的 Web 服务来说，这是不能依赖的²。

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. Java 和 .NET 的运行环境只在少量的机器上安装和部署。——译者注

2. 对于用于企业内部应用的 Intranet 应用，则是另外一回事，因此 Java Web Start 更多地还是应用于这类 Intranet 应用。——译者注

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/art_your_programs_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 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。

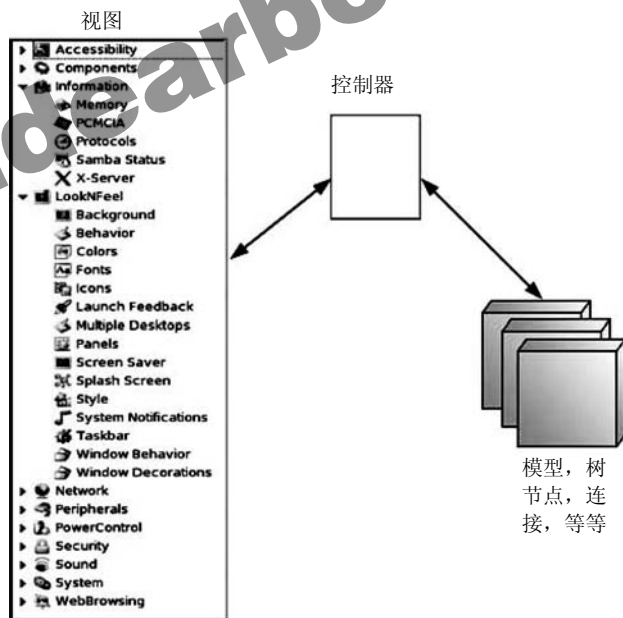


图 4-1 模型-视图-控制器应用于树 UI 组件的内部功能。视图由一组显示在屏幕上的元素组成，后者是一些 DOM 元素。在场景的背后，树结构建模为一组 JavaScript 对象。控制器代码在两者之间起协调的作用

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。这也可以按 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。

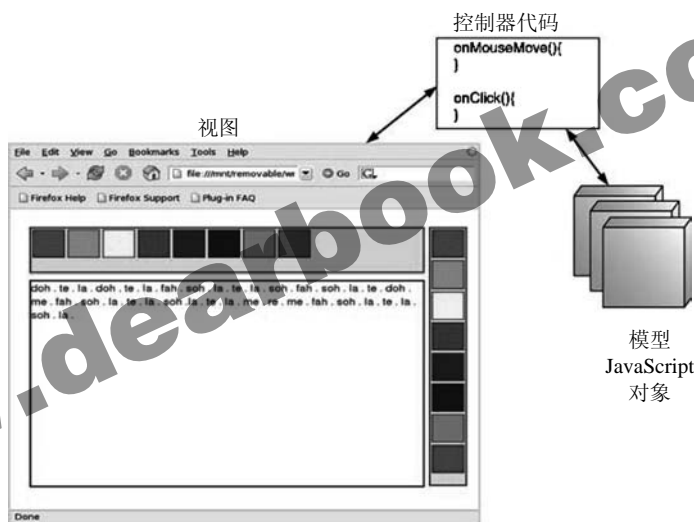


图 4-2 模型-视图-控制器整体应用于 Ajax 客户端应用。这个级别的控制器是将 UI 连接到 JavaScript 业务对象的代码

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。

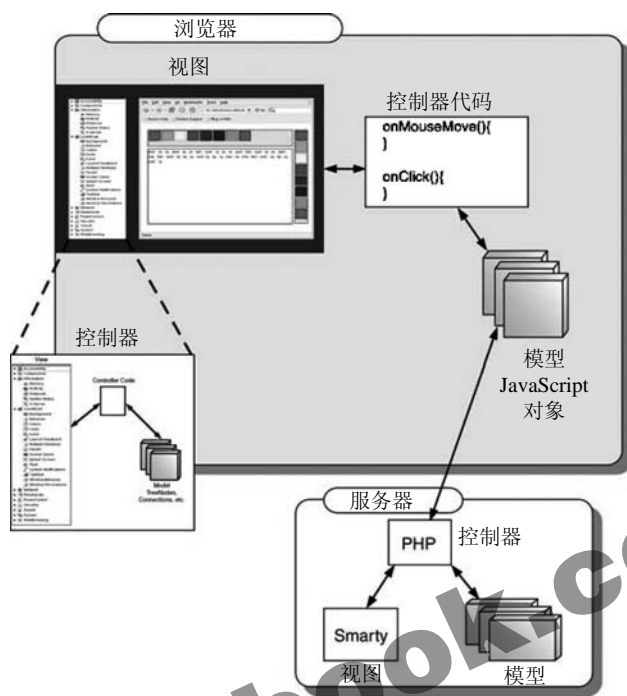


图 4-3 在嵌套的 MVC 架构中，模式以不同的规模重复自己。在最外层的级别，我们可以看到模式作为整体定义了应用的工作流，模型位于 Web 服务器端。在较小的规模，模式在客户端应用中重复；在更小的规模，模式在客户端应用的单个 UI 组件内部重复

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，但是这限制了它的可移植性）。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。

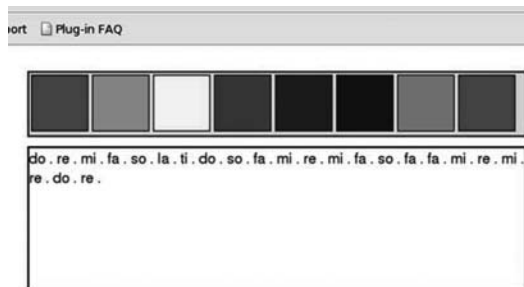


图 4-4 运行于浏览器中的音乐键盘应用。顶部的彩色区域被映射到音符上，当鼠标在上面移动时，音符打印在下面的控制台区域

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, 下面我们将会看到这一点。

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='toplong 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。

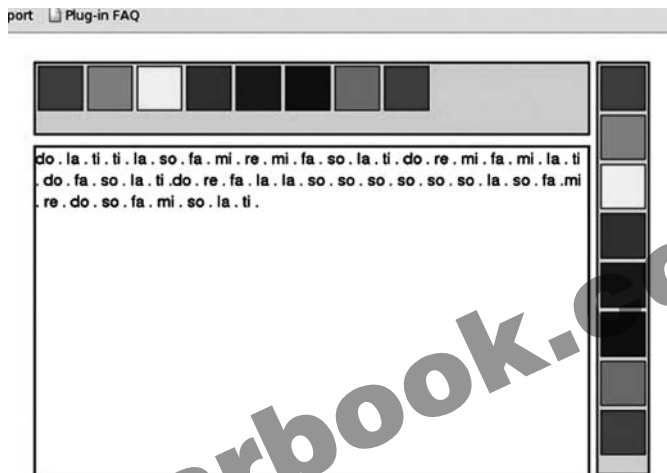


图4-5 已修改的音乐键盘程序允许页面设计师指定多个键盘。使用基于CSS的样式设置和本地呈现引擎，可以同时提供垂直布局和水平布局，而不需要在JavaScript中编写外部的布局代码

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 事件处理规范所取代。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这里讨论。

4.3.1 传统的 JavaScript 事件处理函数

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 |
|--------------------------|------------------------------------|
| <code>onmouseover</code> | 当鼠标首次进入元素区域的时候触发 |
| <code>onmouseout</code> | 当鼠标离开元素区域的时候触发 |
| <code>onmousemove</code> | 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） |
| <code>onclick</code> | 当鼠标在元素区域内被点击的时候触发 |
| <code>onkeypress</code> | 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 |
| <code>onfocus</code> | 可视元素获得了输入焦点 |
| <code>onblur</code> | 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个情况，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¹。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 `Facade` 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。

1. 实际上是指每个元素的每种事件只允许有一个事件处理函数。——译者注

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>
</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。

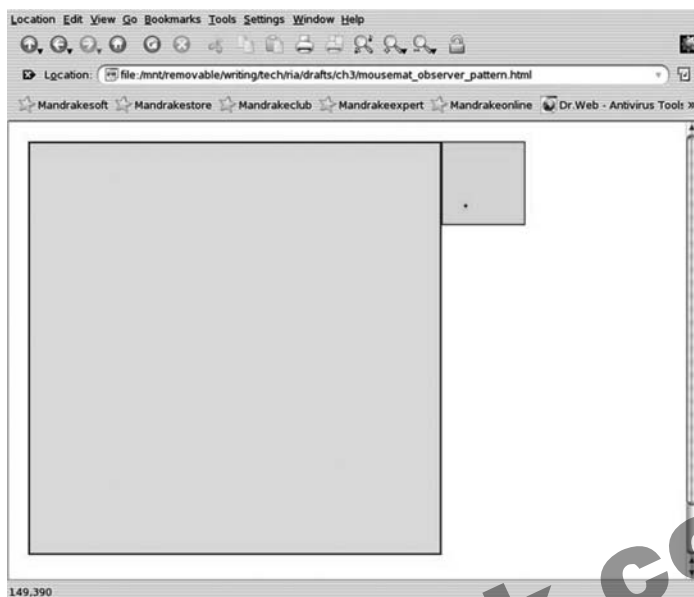


图 4-6 mousemat 程序在“虚拟 mousemat”主区域以两种方式追踪鼠标移动事件：以鼠标的坐标更新浏览器下方的状态条；在缩略视图上随着鼠标光标同步移动的点

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¹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);
}
```

1. 作者指的应该是 C#。——译者注

```

    }
    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>
</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。在下面一节，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 null¹。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

1. 实际上是一个特殊的值 undefined。但是在 JavaScript 中，很多时候 undefined 和 null 可以混用。——译者注

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，而在不久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，我们希望能将智能分布在客户端和服务端，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。

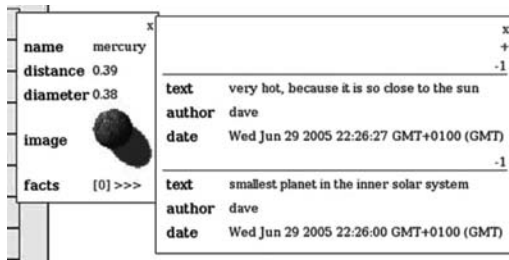


图 4-7 这里 ObjectViewer 用来显示行星系统的等级，每个等级包含一些信息的属性，以及一个保存在数组中的事实列表

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[propName];
        var propType=objviewer.autoType(propValue);
```

```

        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.isInline){
        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){    ❶ 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){    ❷ 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);    ❸ 替换读/写视图
        editor.onblur=objviewer.
            PropertyViewer.editBlur;    ❹ 添加提交回调
        editor.viewer=this;
        editor.focus();
    }
}
objviewer.PropertyViewer
    .editBlur=function(e){    ❺ 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); ❹ 通知观察者
    }
  }
}

```

编辑属性包括几个阶段。如果字段是可编辑的❶，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()❷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❸。我们还在这个文本区域附加了 onblur 处理函数❹，它使用一个只读标签替换了可编辑区域❺，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❻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://riptide.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）。