第 1 部分: Ajax 简介

Ajax 由 HTML、JavaScript™ 技术、DHTML 和 DOM 组成，这一杰出的方法可以将笨拙的 Web 界面转化成交互性的 Ajax 应用程序。本系列的作者是一位 Ajax 专家，他演示了这些技术如何协同工作 —— 从总体概述到细节的讨论 —— 使高效的 Web 开发成为现实。他还揭开了 Ajax 核心概念的神秘面纱，包括 XMLHttpRequest 对象。

五年前，如果不知道 XML，您就是一只无人重视的丑小鸭。十八个月前，Ruby 成了关注的中心，不知道 Ruby 的程序员只能坐冷板凳了。今天，如果想跟上最新的技术时尚，那您的目标就是 Ajax。

但是，Ajax *不仅仅* 是一种时尚，它是一种构建网站的强大方法，而且不像学习一种全新的语言那样困难。

但在详细探讨 Ajax 是什么之前，先让我们花几分钟了解 Ajax *做* 什么。目前，编写应用程序时有两种基本的选择：

- 桌面应用程序
- Web 应用程序

请访问 Ajax 技术资源中心，这是有关 Ajax 编程模型信息的一站式中心，包括很多文档、教程、论坛、blog、wiki 和新闻。任何新信息都能在这里找到。

两者是类似的，桌面应用程序通常以 CD 为介质（有时候可从网站下载）并完全安装到您的计算机上。桌面应用程序可能使用互联网下载更新，但运行这些应用程序的代码在桌面计算机上。Web 应用程序运行在某处的 Web 服务器上 —— 毫不奇怪，要通过 Web 浏览器访问这种应用程序。

不过，比这些应用程序的运行代码放在何处更重要的是，应用程序如何运转以及如何与其进行交互。桌面应用程序一般很快（就在您的计算机上运行，不用等待互联网连接），具有漂亮的用户界面（通常和操作系统有关）和非凡的动态性。可以单击、选择、输入、打开菜单和子菜单、到处巡游，基本上不需要等待。另一方面，Web 应用程序是最新的潮流，它们提供了在桌面上不能实现的服务（比如 Amazon.com 和 eBay）。但是，伴随着 Web 的强大而出现的是等待，等待服务器响应，等待屏幕刷新，等待请求返回和生成新的页面。

显然这样说过于简略了，但基本的概念就是如此。您可能已经猜到，Ajax 尝试建立桌面应用程序的功能和交互性，与不断更新的 Web 应用程序之间的桥梁。可以使用像桌面应用程序中常见的动态用户界面和漂亮的控件，不过是在 Web 应用程序中。

还等什么呢？我们来看看 Ajax 如何将笨拙的 Web 界面转化成能迅速响应的 Ajax 应用程序吧。

老技术，新技巧

在谈到 Ajax 时，实际上涉及到多种技术，要灵活地运用它必须深入了解这些不同的技术（本系列的头几篇文章将分别讨论这些技术）。好消息是您可能已经非常熟悉其中的大部分技术，更好的是这些技术都很容易学习，并不像完整的编程语言（如 Java 或 Ruby）那样困难。

下面是 Ajax 应用程序所用到的基本技术：

- HTML 用于建立 Web 表单并确定应用程序其他部分使用的字段。
- JavaScript 代码是运行 Ajax 应用程序的核心代码，帮助改进与服务器应用程序的通信。
- DHTML 或 Dynamic HTML，用于动态更新表单。我们将使用 `div`、`span` 和其他动态 HTML 元素来标记 HTML。

Ajax 的定义

顺便说一下，Ajax 是 Asynchronous JavaScript and XML（以及 DHTML 等）的缩写。这个短语是 Adaptive Path 的 Jesse James Garrett 发明的（请参阅 参考资料），按照 Jesse 的解释，这*不是* 个首字母缩写词。

- 文档对象模型 DOM 用于（通过 JavaScript 代码）处理 HTML 结构和（某些情况下）服务器返回的 XML。

我们来进一步分析这些技术的职责。以后的文章中我将深入讨论这些技术，目前只要熟悉这些组件和技术就可以了。对这些代码越熟悉，就越容易从对这些技术的零散了解转变到真正把握这些技术（同时也真正打开了 Web 应用程序开发的大门）。

XMLHttpRequest 对象

要了解的一个对象可能对您来说也是最陌生的，即 `XMLHttpRequest`。这是一个 JavaScript 对象，创建该对象很简单，如清单 1 所示。

清单 1. 创建新的 XMLHttpRequest 对象

```
<script language="javascript" type="text/javascript">
var xmlHttp = new XMLHttpRequest();
</script>
```

下一期文章中将进一步讨论这个对象，现在要知道这是处理所有服务器通信的对象。继续阅读之前，先停下来想一想：通过 `XMLHttpRequest` 对象与服务器进行对话的是 *JavaScript* 技术。这不是一般的应用程序流，这恰恰是 Ajax 的强大功能的来源。

在一般的 Web 应用程序中，用户填写表单字段并单击 *Submit* 按钮。然后整个表单发送到服务器，服务器将它转发给处理表单的脚本（通常是 PHP 或 Java，也可能是 CGI 进程或者类似的东西），脚本执行完成后再发送回全新的页面。该页面可能是带有已经填充某些数据的新表单的 HTML，也可能是确认页面，或者是具有根据原来表单中输入数据选择的某些选项的页面。当然，在服务器上的脚本或程序处理和返回新表单时用户必须等待。屏幕变成一片空白，等到服务器返回数据后再重新绘制。这就是交互性差的原因，用户得不到立即反馈，因此感觉不同于桌面应用程序。

Ajax 基本上就是把 JavaScript 技术和 `XMLHttpRequest` 对象放在 Web 表单和服务器*之间*。当用户填写表单时，数据发送给一些 JavaScript 代码而*不是* 直接发送给服务器。相反，JavaScript 代码捕获表单数据并向服务器发送请求。同时用户屏幕上的表单也不会闪烁、消失或延迟。换句话说，JavaScript 代码在幕后发送请求，用户甚至不知道请求的发出。更好的是，请求是异步发送的，就是说 JavaScript 代码（和用户）不用等待服务器的响应。因此用户可以继续输入数据、滚动屏幕和使用应用程序。

然后，服务器将数据返回 JavaScript 代码（仍然在 Web 表单中），后者决定如何处理这些数据。它可以迅速更新表单数据，让人感觉应用程序是立即完成的，表单没有提交或刷新而用户得到了新数据。

JavaScript 代码甚至可以对收到的数据执行某种计算，再发送另一个请求，完全不需要用户干预！这就是 `XMLHttpRequest` 的强大之处。它可以根据需要自行与服务器进行交互，用户甚至可以完全不知道幕后发生的一切。结果就是类似于桌面应用程序的动态、快速响应、高交互性的体验，但是背后又拥有互联网的全部强大力量。

加入一些 JavaScript

得到 `XMLHttpRequest` 的句柄后，其他的 JavaScript 代码就非常简单了。事实上，我们将使用 JavaScript 代码完成非常基本的任务：

- 获取表单数据：JavaScript 代码很容易从 HTML 表单中抽取数据并发送到服务器。
- 修改表单上的数据：更新表单也很简单，从设置字段值到迅速替换图像。
- 解析 HTML 和 XML：使用 JavaScript 代码操纵 DOM（请参阅 下一节），处理 HTML 表单服务器返回的 XML 数据的结构。

对于前两点，需要非常熟悉 `getElementById()` 方法，如 清单 2 所示。

清单 2. 用 JavaScript 代码捕获和设置字段值

```
// Get the value of the "phone" field and stuff it in a variable called phone
var phone = document.getElementById("phone").value;
// Set some values on a form using an array called response
document.getElementById("order").value = response[0];
document.getElementById("address").value = response[1];
```

这里没有特别需要注意的地方，真是好极了！您应该认识到这里并没有非常复杂的东西。只要掌握了 `XMLHttpRequest`，Ajax 应用程序的其他部分就是如 清单 2 所示的简单 JavaScript 代码了，混合有少量的 HTML。同时，还要用一点儿 DOM，我们就来看看吧。

以 DOM 结束

最后还有 DOM，即文档对象模型。可能对有些读者来说 DOM 有点儿令人生畏，HTML 设计者很少使用它，即使 JavaScript 程序员也不大用到它，除非要完成某项高端编程任务。大量使用 DOM 的*是* 复杂的 Java 和 C/C++ 程序，这可能就是 DOM 被认为难以学习的原因。

幸运的是，在 JavaScript 技术中使用 DOM 很容易，也非常直观。现在，按照常规也许应该说明如何使用 DOM，或者至少要给出一些示例代码，但这样做也可能误导您。即使不理会 DOM，仍然能深入地探讨 Ajax，这也是我准备采用的方法。以后的文章将再次讨论 DOM，现在只要知道可能需要 DOM 就可以了。当需要在 JavaScript 代码和服务器之间传递 XML 和改变 HTML 表单的时候，我们再深入研究 DOM。没有它也能做一些有趣的工作，因此现在就把 DOM 放到一边吧。

回页首

获取 Request 对象

有了上面的基础知识后，我们来看看一些具体的例子。`XMLHttpRequest` 是 Ajax 应用程序的核心，而且对很多读者来说可能还比较陌生，我们就从这里开始吧。从 清单 1 可以看出，创建和使用这个对象非常简单，不是吗？等一等。

还记得几年前的那些讨厌的浏览器战争吗？没有一样东西在不同的浏览器上得到同样的结果。不管您是否相信，这些战争仍然在继续，虽然规模较小。但令人奇怪的是，`XMLHttpRequest` 成了这场战争的牺牲品之一。因此获得 `XMLHttpRequest` 对象可能需要采用不同的方法。下面我将详细地进行解释。

使用 Microsoft 浏览器

Microsoft 浏览器 Internet Explorer 使用 MSXML 解析器处理 XML（可以通过 参考资料 进一步了解 MSXML）。因此如果编写的 Ajax 应用程序要和 Internet Explorer 打交道，那么必须用一种特殊的方式创建对象。

但并不是这么简单。根据 Internet Explorer 中安装的 JavaScript 技术版本不同，MSXML 实际上有两种不同的版本，因此必须对这两种情况分别编写代码。请参阅 清单 3，其中的代码在 Microsoft 浏览器上创建了一个 `XMLHttpRequest`。

清单 3. 在 Microsoft 浏览器上创建 XMLHttpRequest 对象

```
var xmlHttp = false;
try {
  xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");
} catch (e) {
  try {
    xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
  } catch (e2) {
    xmlHttp = false;
  }
}
```

您对这些代码可能还不完全理解，但没有关系。当本系列文章结束的时候，您将对 JavaScript 编程、错误处理、条件编译等有更深的了解。现在只要牢牢记住其中的两行代码：

`xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");`

和

`xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");`。

这两行代码基本上就是尝试使用一个版本的 MSXML 创建对象，如果失败则使用另一个版本创建该对象。不错吧？如果都不成功，则将 `xmlHttp` 变量设为 false，告诉您的代码出现了问题。如果出现这种情况，可能是因为安装了非 Microsoft 浏览器，需要使用不同的代码。

处理 Mozilla 和非 Microsoft 浏览器

如果选择的浏览器不是 Internet Explorer，或者为非 Microsoft 浏览器编写代码，就需要使用不同的代码。事实上就是 清单 1 所示的一行简单代码：

`var xmlHttp = new XMLHttpRequest object;`。

这行简单得多的代码在 Mozilla、Firefox、Safari、Opera 以及基本上所有以任何形式或方式支持 Ajax 的非 Microsoft 浏览器中，创建了 `XMLHttpRequest` 对象。

结合起来

关键是要支持*所有* 浏览器。谁愿意编写一个只能用于 Internet Explorer 或者非 Microsoft 浏览器的应用程序呢？或者更糟，要编写一个应用程序两次？当然不！因此代码要同时支持 Internet Explorer 和非 Microsoft 浏览器。清单 4 显示了这样的代码。

清单 4. 以支持多种浏览器的方式创建 XMLHttpRequest 对象

```
/* Create a new XMLHttpRequest object to talk to the Web server */
var xmlHttp = false;
```

```
/*@cc_on @*/
/*@if (@_jscript_version >= 5)
try {
  xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");
} catch (e) {
  try {
    xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
  } catch (e2) {
    xmlHttp = false;
  }
}
@end @*/
if (!xmlHttp && typeof XMLHttpRequest != 'undefined') {
  xmlHttp = new XMLHttpRequest();
}
```

现在先不管那些注释掉的奇怪符号，如 `@cc_on`，这是特殊的 JavaScript 编译器命令，将在下一期针对 `XMLHttpRequest` 的文章中详细讨论。这段代码的核心分为三步：

1. 建立一个变量 `xmlHttp` 来引用即将创建的 `XMLHttpRequest` 对象。
2. 尝试在 Microsoft 浏览器中创建该对象：
    - 尝试使用 `Msxml2.XMLHTTP` 对象创建它。
    - 如果失败，再尝试 `Microsoft.XMLHTTP` 对象。
3. 如果仍然没有建立 `xmlHttp`，则以非 Microsoft 的方式创建该对象。

最后，`xmlHttp` 应该引用一个有效的 `XMLHttpRequest` 对象，无论运行什么样的浏览器。

关于安全性的一点说明

安全性如何呢？现在浏览器允许用户提高他们的安全等级，关闭 JavaScript 技术，禁用浏览器中的任何选项。在这种情况下，代码无论如何都不会工作。此时必须适当地处理问题，这需要单独的一篇文章来讨论，要放到以后了（这个系列够长了吧？不用担心，读完之前也许您就掌握了）。现在要编写一段健壮但不够完美的代码，对于掌握 Ajax 来说就很好了。以后我们还将讨论更多的细节。

回页首

Ajax 世界中的请求/响应

现在我们介绍了 Ajax，对 `XMLHttpRequest` 对象以及如何创建它也有了基本的了解。如果阅读得很仔细，您可能已经知道与服务器上的 Web 应用程序打交道的是 JavaScript 技术，而不是直接提交给那个应用程序的 HTML 表单。

还缺少什么呢？到底如何使用 `XMLHttpRequest`。因为这段代码非常重要，您编写的*每个* Ajax 应用程序都要以某种形式使用它，先看看 Ajax 的基本请求/响应模型是什么样吧。

发出请求

您已经有了一个崭新的 XMLHttpRequest 对象，现在让它干点活儿吧。首先需要一个 Web 页面能够调用的 JavaScript 方法（比如当用户输入文本或者从菜单中选择一项时）。接下来就是在所有 Ajax 应用程序中基本都雷同的流程：

1. 从 Web 表单中获取需要的数据。
2. 建立要连接的 URL。
3. 打开到服务器的连接。
4. 设置服务器在完成后要运行的函数。
5. 发送请求。

清单 5 中的示例 Ajax 方法就是按照这个顺序组织的：

清单 5. 发出 Ajax 请求

```
function callServer() {
  // Get the city and state from the web form
  var city = document.getElementById("city").value;
  var state = document.getElementById("state").value;
  // Only go on if there are values for both fields
  if ((city == null) || (city == "")) return;
  if ((state == null) || (state == "")) return;
  // Build the URL to connect to
  var url = "/scripts/getZipCode.php?city=" + escape(city) + "&state=" +
escape(state);
  // Open a connection to the server
  xmlHttp.open("GET", url, true);
  // Setup a function for the server to run when it's done
  xmlHttp.onreadystatechange = updatePage;
  // Send the request
  xmlHttp.send(null);
}
```

其中大部分代码意义都很明确。开始的代码使用基本 JavaScript 代码获取几个表单字段的值。然后设置一个 PHP 脚本作为链接的目标。要注意脚本 URL 的指定方式，city 和 state（来自表单）使用简单的 GET 参数附加在 URL 之后。

然后打开一个连接，这是您第一次看到使用 XMLHttpRequest。其中指定了连接方法（GET）和要连接的 URL。最后一个参数如果设为 true，那么将请求一个异步连接（这就是 Ajax 的由来）。如果使用 false，那么代码发出请求后将等待服务器返回的响应。如果设为 true，当服务器在后台处理请求的时候用户仍然可以使用表单（甚至调用其他 JavaScript 方法）。

xmlHttp（要记住，这是 XMLHttpRequest 对象实例）的 onreadystatechange 属性可以告诉服务器在运行*完成* 后（可能要用五分钟或者五个小时）做什么。因为代码没有等待服务器，必须让服务器知道怎么做以便您能作出响应。在这个示例中，如果服务器处理完了请求，一个特殊的名为 updatePage() 的方法将被触发。

最后，使用值 `null` 调用 `send()`。因为已经在请求 URL 中添加了要发送给服务器的数据（city 和 state），所以请求中不需要发送任何数据。这样就发出了请求，服务器按照您的要求工作。

如果没有发现任何新鲜的东西，您应该体会到这是多么简单明了！除了牢牢记住 Ajax 的异步特性外，这些内容都相当简单。应该感激 Ajax 使您能够专心编写漂亮的应用程序和界面，而不用担心复杂的 HTTP 请求/响应代码。

清单 5 中的代码说明了 Ajax 的易用性。数据是简单的文本，可以作为请求 URL 的一部分。用 GET 而不是更复杂的 POST 发送请求。没有 XML 和要添加的内容头部，请求体中没有要发送的数据；换句话说，这就是 Ajax 的乌托邦。

不用担心，随着本系列文章的展开，事情会变得越来越复杂。您将看到如何发送 POST 请求、如何设置请求头部和内容类型、如何在消息中编码 XML、如何增加请求的安全性，可以做的工作还有很多！暂时先不用管那些难点，掌握好基本的东西就行了，很快我们就会建立一整套的 Ajax 工具库。

处理响应

现在要面对服务器的响应了。现在只要知道两点：

- 什么也不要做，直到 `xmlHttp.readyState` 属性的值等于 4。
- 服务器将把响应填充到 `xmlHttp.responseText` 属性中。

其中的第一点，即就绪状态，将在下一篇文章中详细讨论，您将进一步了解 HTTP 请求的阶段，可能比您设想的还多。现在只要检查一个特定的值（4）就可以了（下一期文章中还有更多的值要介绍）。第二点，使用 `xmlHttp.responseText` 属性获得服务器的响应，这很简单。清单 6 中的示例方法可供服务器根据清单 5 中发送的数据调用。

清单 6. 处理服务器响应

```
function updatePage() {
  if (xmlHttp.readyState == 4) {
    var response = xmlHttp.responseText;
    document.getElementById("zipCode").value = response;
  }
}
```

这些代码同样既不难也不复杂。它等待服务器调用，如果是就绪状态，则使用服务器返回的值（这里是用户输入的城市和州的 ZIP 编码）设置另一个表单字段的值。于是包含 ZIP 编码的 `zipCode` 字段突然出现了，而用户*没有按任何按钮*！这就是前面所说的桌面应用程序的感觉。快速响应、动态感受等等，这些都只因为有了小小的一段 Ajax 代码。

细心的读者可能注意到 `zipCode` 是一个普通的文本字段。一旦服务器返回 ZIP 编码，`updatePage()` 方法就用城市/州的 ZIP 编码设置那个字段的值，用户就可以改写该值。这样做有两个原因：保持例子简单，说明有时候可能*希望* 用户能够修改服务器返回的数据。要记住这两点，它们对于好的用户界面设计来说很重要。

回页首

连接 Web 表单

还有什么呢？实际上没有多少了。一个 JavaScript 方法捕捉用户输入表单的信息并将其发送到服务器，另一个 JavaScript 方法监听和处理响应，并在响应返回时设置字段的值。所有这些实际上都依赖于*调用*第一个 JavaScript 方法，它启动了整个过程。最明显的办法是在 HTML 表单中增加一个按钮，但这是 2001 年的办法，您不这样认为吗？还是像 清单 7 这样利用 JavaScript 技术吧。

清单 7. 启动一个 Ajax 过程

```
<form>
 <p>City: <input type="text" name="city" id="city" size="25"
      onChange="callServer();" /></p>
 <p>State: <input type="text" name="state" id="state" size="25"
      onChange="callServer();" /></p>
 <p>Zip Code: <input type="text" name="zipCode" id="city" size="5" /></p>
</form>
```

如果感觉这像是一段相当普通的代码，那就对了，正是如此！当用户在 city 或 state 字段中输入新的值时，`callServer()` 方法就被触发，于是 Ajax 开始运行了。有点儿明白怎么回事了吧？好，就是如此！

回页首

结束语

现在您可能已经准备开始编写第一个 Ajax 应用程序了，至少也希望认真读一下 参考资料 中的那些文章了吧？但可以首先从这些应用程序如何工作的基本概念开始，对 `XMLHttpRequest` 对象有基本的了解。在下一期文章中，您将掌握这个对象，学会如何处理 JavaScript 和服务器的通信、如何使用 HTML 表单以及如何获得 DOM 句柄。

现在先花点儿时间考虑考虑 Ajax 应用程序有多么强大。设想一下，当单击按钮、输入一个字段、从组合框中选择一个选项或者用鼠标在屏幕上拖动时，Web 表单能够立刻作出响应会是什么情形。想一想*异步*究竟意味着什么，想一想 JavaScript 代码运行而且*不等待* 服务器对它的请求作出响应。会遇到什么样的问题？会进入什么样的领域？考虑到这种新的方法，编程的时候应如何改变表单的设计？

如果在这些问题上花一点儿时间，与简单地剪切/粘贴某些代码到您根本不理解的应用程序中相比，收益会更多。在下一期文章中，我们将把这些概念付诸实践，详细介绍使应用程序按照这种方式工作所需要的代码。因此，现在先享受一下 Ajax 所带来的可能性吧。

第 2 部分：使用 JavaScript 和 Ajax 发出异步请求

多数 Web 应用程序都使用请求/响应模型从服务器上获得完整的 HTML 页面。常常是点击一个按钮，等待服务器响应，再点击另一个按钮，然后再等待，这样一个反复的过程。有了 Ajax 和 XMLHttpRequest 对象，就可以使用不必让用户等待服务器响应的请求/响应模型了。本文中，Brett McLaughlin 介绍了如何创建能够适应不同浏览器的 XMLHttpRequest 实例，建立和发送请求，并响应服务器。

本系列的上一期文章（请参阅 参考资料 中的链接），我们介绍了 Ajax 应用程序，考察了推动 Ajax 应

用程序的基本概念。其中的核心是很多您可能已经了解的技术：JavaScript、HTML 和 XHTML、一点动态 HTML 以及 DOM（文档对象模型）。本文将放大其中的一点，把目光放到具体的 Ajax 细节上。

本文中，您将开始接触最基本和基础性的有关 Ajax 的全部对象和编程方法：`XMLHttpRequest` 对象。该对象实际上仅仅是一个跨越所有 Ajax 应用程序的公共线程，您可能已经预料到，只有彻底理解该对象才能充分发挥编程的潜力。事实上，有时您会发现，要正确地使用 `XMLHttpRequest`，显然*不能* 使用 `XMLHttpRequest`。这到底是怎么回事呢？

Web 2.0 一瞥

在深入研究代码之前首先看看最近的观点 —— 一定要十分清楚 Web 2.0 这个概念。听到 Web 2.0 这个词的时候，应该首先问一问“Web 1.0 是什么？”虽然很少听人提到 Web 1.0，实际上它指的就是具有完全不同的请求和响应模型的传统 Web。比如，到 Amazon.com 网站上点击一个按钮或者输入搜索项。就会对服务器发送一个请求，然后响应再返回到浏览器。该请求不仅仅是图书和书目列表，而是另一个完整的 HTML 页面。因此当 Web 浏览器用新的 HTML 页面重绘时，可能会看到闪烁或抖动。事实上，通过看到的每个新页面可以清晰地看到请求和响应。

Web 2.0（在很大程度上）消除了这种看得见的往复交互。比如访问 Google Maps 或 Flickr 这样的站点（到这些支持 Web 2.0 和 Ajax 站点的链接请参阅 参考资料）。比如在 Google Maps 上，您可以拖动地图，放大和缩小，只有很少的重绘操作。当然这里仍然有请求和响应，只不过都*藏到了幕后*。作为用户，体验更加舒适，感觉很像桌面应用程序。这种新的感受和范型就是当有人提到 Web 2.0 时您所体会到的。

需要关心的是如何使这些新的交互成为可能。显然，仍然需要发出请求和接收响应，但正是针对每次请求/响应交互的 HTML 重绘造成了缓慢、笨拙的 Web 交互的感受。因此很清楚，我们需要一种方法使发送的请求和接收的响应*只* 包含需要的数据而不是整个 HTML 页面。惟一需要获得整个新 HTML 页面的时候就是希望用户*看到* 新页面的时候。

但多数交互都是在已有页面上增加细节、修改主体文本或者覆盖原有数据。这些情况下，Ajax 和 Web 2.0 方法允许在*不* 更新整个 HTML 页面的情况下发送和接收数据。对于那些经常上网的人，这种能力可以让您的应用程序感觉更快、响应更及时，让他们不时地光顾您的网站。

回页首

XMLHttpRequest 简介

要真正实现这种绚丽的奇迹，必须非常熟悉一个 JavaScript 对象，即 `XMLHttpRequest`。这个小小的对象实际上已经在几种浏览器中存在一段时间了，它是本专栏今后几个月中要介绍的 Web 2.0、Ajax 和大部分其他内容的核心。为了让您快速地大体了解它，下面给出将要用于该对象的*很少的几个* 方法和属性。

- `open()`：建立到服务器的新请求。
- `send()`：向服务器发送请求。

- `abort()`：退出当前请求。
- `readyState`：提供当前 HTML 的就绪状态。
- `responseText`：服务器返回的请求响应文本。

如果不了解这些（或者其中的*任何*一个），您也不用担心，后面几篇文章中我们将介绍每个方法和属性。现在*应该*了解的是，明确用 `XMLHttpRequest` 做什么。要注意这些方法和属性都与发送请求及处理响应有关。事实上，如果看到 `XMLHttpRequest` 的所有方法和属性，就会发现它们*都*与非常简单的请求/响应模型有关。显然，我们不会遇到特别新的 GUI 对象或者创建用户交互的某种超极神秘的方法，我们将使用非常简单的请求和非常简单的响应。听起来似乎没有多少吸引力，但是用好该对象可以彻底改变您的应用程序。

简单的 new

首先需要创建一个新变量并赋给它一个 `XMLHttpRequest` 对象实例。这在 JavaScript 中很简单，只要对该对象名使用 `new` 关键字即可，如 清单 1 所示。

清单 1. 创建新的 XMLHttpRequest 对象

```
<script language="javascript" type="text/javascript">
var request = new XMLHttpRequest();
</script>
```

不难吧？记住，JavaScript 不要求指定变量类型，因此不需要像 清单 2 那样做（在 Java 语言中可能需要这样）。

清单 2. 创建 XMLHttpRequest 的 Java 伪代码

```
XMLHttpRequest request = new XMLHttpRequest();
```

因此在 JavaScript 中用 `var` 创建一个变量，给它一个名字（如 “request”），然后赋给它一个新的 `XMLHttpRequest` 实例。此后就可以在函数中使用该对象了。

错误处理

在实际上各种事情都可能出错，而上面的代码没有提供任何错误处理。较好的办法是创建该对象，并在出现问题时优雅地退出。比如，任何较早的浏览器（不论您是否相信，仍然有人在使用老版本的 Netscape Navigator）都不支持 `XMLHttpRequest`，您需要让这些用户知道有些地方出了问题。清单 3 说明如何创建该对象，以便在出现问题的时候发出 JavaScript 警告。

清单 3. 创建具有错误处理能力的 XMLHttpRequest

```
<script language="javascript" type="text/javascript">
var request = false;
try {
```

```
  request = new XMLHttpRequest();
} catch (failed) {
  request = false;
}
if (!request)
  alert("Error initializing XMLHttpRequest!");
</script>
```

一定要理解这些步骤：

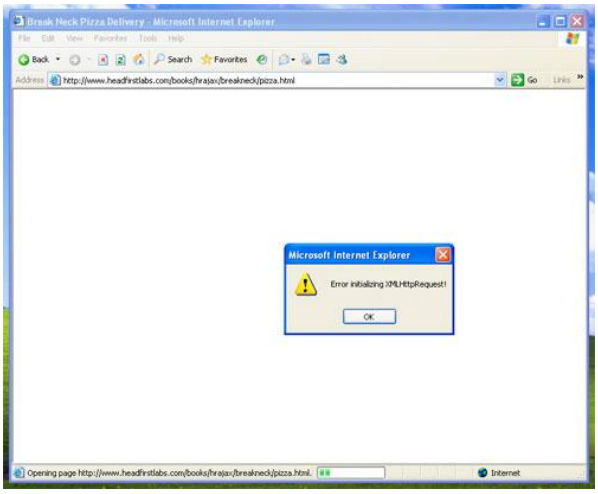
1. 创建一个新变量 `request` 并赋值 false。后面将使用 false 作为判定条件，它表示还没有创建 `XMLHttpRequest` 对象。
2. 增加 try/catch 块：
    1. 尝试创建 `XMLHttpRequest` 对象。
    2. 如果失败（`catch (failed)`）则保证 `request` 的值仍然为 false。
3. 检查 `request` 是否仍为 false（如果一切正常就不会是 false）。
4. 如果出现问题（`request` 是 false）则使用 JavaScript 警告通知用户出现了问题。

代码非常简单，对大多数 JavaScript 和 Web 开发人员来说，真正理解它要比读写代码花更长的时间。现在已经得到了一段带有错误检查的 `XMLHttpRequest` 对象创建代码，还可以告诉您哪儿出了问题。

应付 Microsoft

看起来似乎一切良好，至少在用 Internet Explorer 试验这些代码之前是这样的。如果这样试验的话，就会看到 图 1 所示的糟糕情形。

图 1. Internet Explorer 报告错误



Microsoft 参与了吗？

关于 Ajax 和 Microsoft 对该领域不断增长的兴趣和参与已经有很多文章进行了介绍。事实上，据说 Microsoft 最新版本的 Internet Explorer —— version 7.0，将

显然有什么地方不对劲，而 Internet Explorer 很难说是一种过时的浏览器，因为全世界有 70% 在使用 Internet Explorer。换句话说，如果不支持 Microsoft 和 Internet Explorer 就不会受到 Web 世界的欢迎！因此我们需要采用不同的方法处理 Microsoft 浏览器。经验证发现 Microsoft 支持 Ajax，但是其 XMLHttpRequest 版本有不同的称呼。事实上，它将其称为*几种*不同的东西。如果使用较新版本的 Internet Explorer，则需要使用对象 Msxml2.XMLHTTP，而较老版本的 Internet Explorer 则使用 Microsoft.XMLHTTP。我们需要支持这两种对象类型（同时还要支持非 Microsoft 浏览器）。请看看 清单 4，它在前述代码的基础上增加了对 Microsoft 的支持。

> 在 2006 年下半年推出 —— 将开始直接支持 XMLHttpRequest，让您使用 new 关键字代替所有的 Msxml2.XMLHTTP 创建代码。但不要太激动，仍然需要支持旧的浏览器，因此跨浏览器代码不会很快消失。

清单 4. 增加对 Microsoft 浏览器的支持

```
<script language="javascript" type="text/javascript">
var request = false;
try {
  request = new XMLHttpRequest();
} catch (trymicrosoft) {
  try {
    request = new ActiveXObject("Msxml2.XMLHTTP");
  } catch (othermicrosoft) {
    try {
      request = new ActiveXObject("Microsoft.XMLHTTP");
    } catch (failed) {
      request = false;
    }
  }
}
if (!request)
  alert("Error initializing XMLHttpRequest!");
</script>
```

很容易被这些花括号迷住了眼睛，因此下面分别介绍每一步：

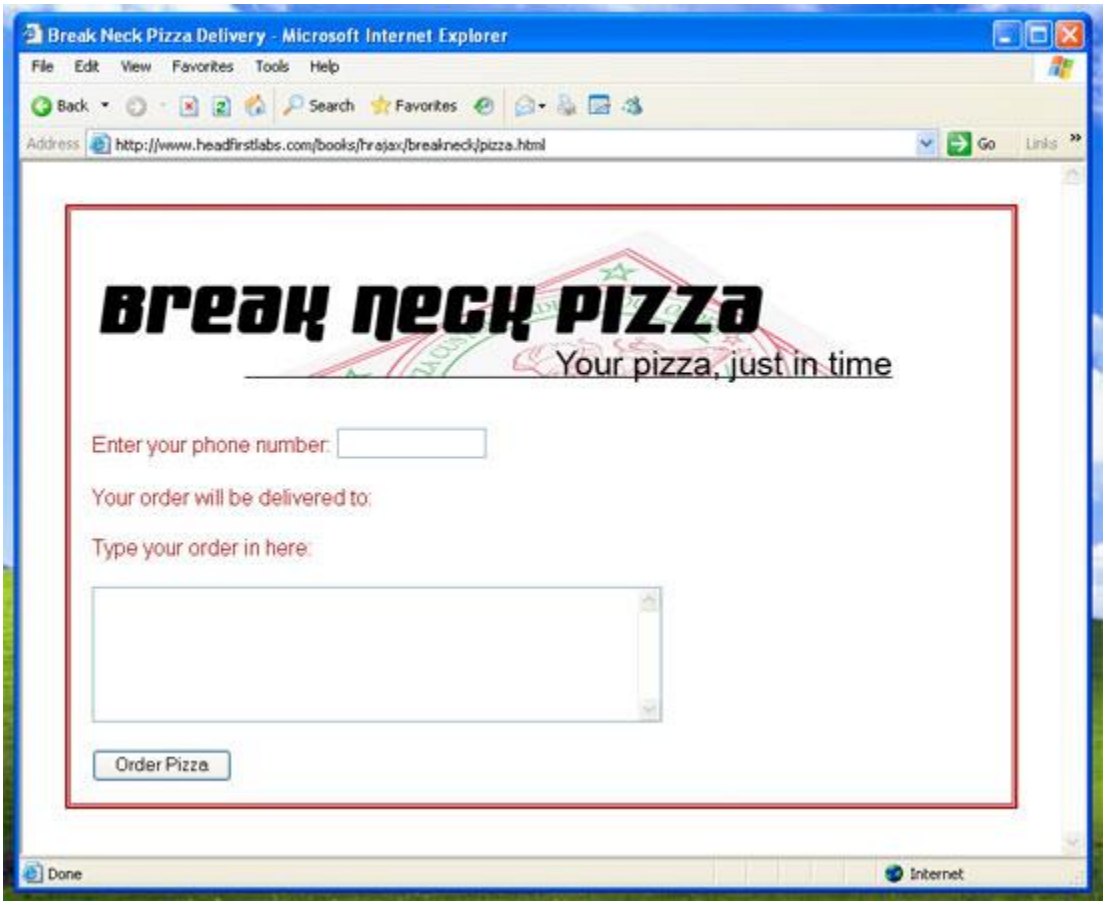
1. 创建一个新变量 request 并赋值 false。使用 false 作为判断条件，它表示还没有创建 XMLHttpRequest 对象。
2. 增加 try/catch 块：
    1. 尝试创建 XMLHttpRequest 对象。
    2. 如果失败（catch (trymicrosoft)）：
        1. 尝试使用较新版本的 Microsoft 浏览器创建 Microsoft 兼容的对象（Msxml2.XMLHTTP）。
        2. 如果失败（catch (othermicrosoft)）尝试使用较老版本的 Microsoft 浏览器创建 Microsoft 兼容的对象（Microsoft.XMLHTTP）。

3. 如果失败（`catch (failed)`）则保证 `request` 的值仍然为 false。

3. 检查 `request` 是否仍然为 false（如果一切顺利就不会是 false）。
4. 如果出现问题（`request` 是 false）则使用 JavaScript 警告通知用户出现了问题。

这样修改代码之后再使用 Internet Explorer 试验，就应该看到已经创建的表单（没有错误消息）。我实验的结果如 图 2 所示。

图 2. Internet Explorer 正常工作



静态与动态

再看一看清单 1、3 和 4，注意，所有这些代码都直接嵌套在 `script` 标记中。像这种不放到方法或函数体中的 JavaScript 代码称为*静态 JavaScript*。就是说代码是在页面显示给用户之前的某个时候运行。（虽然根据规范不能完全*精确地* 知道这些代码何时运行对浏览器有什么影响，但是可以保证这些代码在用户能够与页面交互之前运行。）这也是多数 Ajax 程序员创建 `XMLHttpRequest` 对象的一般方式。

就是说，也可以像 清单 5 那样将这些代码放在一个方法中。

清单 5. 将 XMLHttpRequest 创建代码移动到方法中

```
<script language="javascript" type="text/javascript">
var request;
function createRequest() {
```

```
  try {
    request = new XMLHttpRequest();
  } catch (trymicrosoft) {
    try {
      request = new ActiveXObject("Msxml2.XMLHTTP");
    } catch (othermicrosoft) {
      try {
        request = new ActiveXObject("Microsoft.XMLHTTP");
      } catch (failed) {
        request = false;
      }
    }
  }
  if (!request)
    alert("Error initializing XMLHttpRequest!");
}
</script>
```

如果按照这种方式编写代码，那么在处理 Ajax 之前需要调用该方法。因此还需要 清单 6 这样的代码。

清单 6. 使用 XMLHttpRequest 的创建方法

```
<script language="javascript" type="text/javascript">
var request;
function createRequest() {
  try {
    request = new XMLHttpRequest();
  } catch (trymicrosoft) {
    try {
      request = new ActiveXObject("Msxml2.XMLHTTP");
    } catch (othermicrosoft) {
      try {
        request = new ActiveXObject("Microsoft.XMLHTTP");
      } catch (failed) {
        request = false;
      }
    }
  }
  if (!request)
    alert("Error initializing XMLHttpRequest!");
}
function getCustomerInfo() {
```

```
  createRequest();
  // Do something with the request variable
}
</script>
```

此代码惟一的问题是推迟了错误通知，这也是多数 Ajax 程序员不采用这一方法的原因。假设一个复杂的表单有 10 或 15 个字段、选择框等，当用户在第 14 个字段（按照表单顺序从上到下）输入文本时要激活某些 Ajax 代码。这时候运行 `getCustomerInfo()` 尝试创建一个 `XMLHttpRequest` 对象，但（对于本例来说）失败了。然后向用户显示一条警告，明确地告诉他们不能使用该应用程序。但用户已经花费了很多时间在表单中输入数据！这是非常令人讨厌的，而讨厌显然不会吸引用户再次访问您的网站。
如果使用静态 JavaScript，用户在点击页面的时候很快就会看到错误信息。这样也很烦人，是不是？可能令用户错误地认为您的 Web 应用程序不能在他的浏览器上运行。不过，当然要比他们花费了 10 分钟输入信息之后再显示同样的错误要好。因此，我建议编写静态的代码，让用户尽可能早地发现问题。

回页首

用 XMLHttpRequest 发送请求

得到请求对象之后就可以进入请求/响应循环了。记住，`XMLHttpRequest` 惟一的目的是让您发送请求和接收响应。其他一切都是 JavaScript、CSS 或页面中其他代码的工作：改变用户界面、切换图像、解释服务器返回的数据。准备好 `XMLHttpRequest` 之后，就可以向服务器发送请求了。

欢迎使用沙箱

Ajax 采用一种沙箱安全模型。因此，Ajax 代码（具体来说就是 `XMLHttpRequest` 对象）只能对所在的同一个域发送请求。以后的文章中将进一步介绍安全和 Ajax，现在只要知道在本地机器上运行的代码只能对本地机器上的服务器端脚本发送请求。如果让 Ajax 代码在 www.breakneckpizza.com 上运行，则必须 www.breakneck.com 中运行的脚本发送请求。

设置服务器 URL

首先要确定连接的服务器的 URL。这并不是 Ajax 的特殊要求，但仍然是建立连接所必需的，显然现在您应该知道如何构造 URL 了。多数应用程序中都会结合一些静态数据和用户处理的表单中的数据来构造该 URL。比如，清单 7 中的 JavaScript 代码获取电话号码字段的值并用其构造 URL。

清单 7. 建立请求 URL

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

```
  var request = false;
  try {
    request = new XMLHttpRequest();
  } catch (trymicrosoft) {
    try {
      request = new ActiveXObject("Msxml2.XMLHTTP");
    } catch (othermicrosoft) {
      try {
        request = new ActiveXObject("Microsoft.XMLHTTP");
      } catch (failed) {
        request = false;
      }
    }
  }
  if (!request)
    alert("Error initializing XMLHttpRequest!");
  function getCustomerInfo() {
    var phone = document.getElementById("phone").value;
    var url = "/cgi-local/lookupCustomer.php?phone=" + escape(phone);
  }
</script>
```

这里没有难懂的地方。首先，代码创建了一个新变量 `phone`，并把 ID 为“phone”的表单字段的值赋给它。清单 8 展示了这个表单的 XHTML，其中可以看到 `phone` 字段及其 `id` 属性。

清单 8. Break Neck Pizza 表单

```
<body>
 <p><img src="breakneck-logo_4c.gif" alt="Break Neck Pizza" /></p>
 <form action="POST">
  <p>Enter your phone number:
   <input type="text" size="14" name="phone" id="phone"
         onChange="getCustomerInfo();" />
  </p>
  <p>Your order will be delivered to:</p>
  <div id="address"></div>
  <p>Type your order in here:</p>
  <p><textarea name="order" rows="6" cols="50" id="order"></textarea></p>
  <p><input type="submit" value="Order Pizza" id="submit" /></p>
 </form>
</body>
```

还要注意，当用户输入电话号码或者改变电话号码时，将触发 清单 8 所示的 getCustomerInfo() 方法。该方法取得电话号码并构造存储在 url 变量中的 URL 字符串。记住，由于 Ajax 代码是沙箱型的，因而只能连接到同一个域，实际上 URL 中不需要域名。该例中的脚本名为 /cgi-local/lookupCustomer.php。最后，电话号码作为 GET 参数附加到该脚本中："phone=" + escape(phone)。

如果以前没用见过 escape() 方法，它用于转义不能用明文正确发送的任何字符。比如，电话号码中的空格将被转换成字符 %20，从而能够在 URL 中传递这些字符。

可以根据需要添加任意多个参数。比如，如果需要增加另一个参数，只需要将其附加到 URL 中并用“与”（&）字符分开 [第一个参数用问号（?）和脚本名分开]。

打开请求

有了要连接的 URL 后就可以配置请求了。可以用 XMLHttpRequest 对象的 open() 方法来完成。该方法有五个参数：

- *request-type*：发送请求的类型。典型的值是 GET 或 POST，但也可以发送 HEAD 请求。
- *url*：要连接的 URL。
- *asynch*：如果希望使用异步连接则为 true，否则为 false。该参数是可选的，默认为 true。
- *username*：如果需要身份验证，则可以在此指定用户名。该可选参数没有默认值。
- *password*：如果需要身份验证，则可以在此指定口令。该可选参数没有默认值。

open() 是打开吗？

Internet 开发人员对 open() 方法到底做什么没有达成一致。但它实际上并*不是* 打开一个请求。如果监控 XHTML/Ajax 页面及其连接脚本之间的网络和数据传递，当调用 open() 方法时将看不到任何通信。不清楚为何选用了这个名字，但显然不是一个好的选择。

通常使用其中的前三个参数。事实上，即使需要异步连接，也应该指定第三个参数为“true”。这是默认值，但坚持明确指定请求是异步的还是同步的更容易理解。

将这些结合起来，通常会得到 清单 9 所示的一行代码。

清单 9. 打开请求

```
function getCustomerInfo() {
  var phone = document.getElementById("phone").value;
  var url = "/cgi-local/lookupCustomer.php?phone=" + escape(phone);
  request.open("GET", url, true);
}
```

一旦设置好了 URL，其他就简单了。多数请求使用 GET 就够了（后面的文章中将看到需要使用 POST 的情况），再加上 URL，这就是使用 open() 方法需要的全部内容了。

挑战异步性

本系列的后面一篇文章中，我将用很多时间编写和使用异步代码，但是您应该明白为什么 open() 的最后一个参数这么重要。在一般的请求/响应模型中，比如 Web 1.0，客户机（浏览器或者本地机器上运行的代码）向服务器发出请求。该请求是同步的，换句话说，客户机等待服务器的响应。当客户机等待的时候，至少会用某种形式通知您在等待：

- 沙漏（特别是 Windows 上）。

- 旋转的皮球（通常在 Mac 机器上）。
- 应用程序基本上冻结了，然后过一段时间光标变化了。

这正是 Web 应用程序让人感到笨拙或缓慢的原因 —— 缺乏真正的交互性。按下按钮时，应用程序实际上变得不能使用，直到刚刚触发的请求得到响应。如果请求需要大量服务器处理，那么等待的时间可能很长（至少在这个多处理器、DSL 没有等待的世界中是如此）。
而异步请求 *不* 等待服务器响应。发送请求后应用程序继续运行。用户仍然可以在 Web 表单中输入数据，甚至离开表单。没有旋转的皮球或者沙漏，应用程序也没有明显的冻结。服务器悄悄地响应请求，完成后告诉原来的请求者工作已经结束（具体的办法很快就会看到）。结果是，应用程序感觉 *不* 那么迟钝或者缓慢，而是响应迅速、交互性强，感觉快多了。这仅仅是 Web 2.0 的一部分，但它是很重要的一部分。所有老套的 GUI 组件和 Web 设计范型都不能克服缓慢、同步的请求/响应模型。

发送请求

一旦用 `open()` 配置好之后，就可以发送请求了。幸运的是，发送请求的方法的名称要比 `open()` 适当，它就是 `send()`。
`send()` 只有一个参数，就是要发送的内容。但是在考虑这个方法之前，回想一下前面已经通过 URL 本身发送过数据了：

```
var url = "/cgi-local/lookupCustomer.php?phone=" + escape(phone);
```

虽然可以使用 `send()` 发送数据，但也能通过 URL 本身发送数据。事实上，`GET` 请求（在典型的 Ajax 应用中大约占 80%）中，用 URL 发送数据要容易得多。如果需要发送安全信息或 XML，可能要考虑使用 `send()` 发送内容（本系列的后续文章中将讨论安全数据和 XML 消息）。如果不需要通过 `send()` 传递数据，则只要传递 `null` 作为该方法的参数即可。因此您会发现在本文中的例子中只需要这样发送请求（参见 清单 10）。

清单 10. 发送请求

```
function getCustomerInfo() {
  var phone = document.getElementById("phone").value;
  var url = "/cgi-local/lookupCustomer.php?phone=" + escape(phone);
  request.open("GET", url, true);
  request.send(null);
}
```

指定回调方法

现在我们所做的只有很少一点是新的、革命性的或异步的。必须承认，`open()` 方法中 “true” 这个小小的关键字建立了异步请求。但是除此之外，这些代码与用 Java servlet 及 JSP、PHP 或 Perl 编程没有什么两样。那么 Ajax 和 Web 2.0 最大的秘密是什么呢？秘密就在于 `XMLHttpRequest` 的一个简单属性 `onreadystatechange`。
首先一定要理解这些代码中的流程（如果需要请回顾 清单 10）。建立其请求然后发出请求。此外，因为是异步请求，所以 JavaScript 方法（例子中的 `getCustomerInfo()`）不会等待服务器。因此代码将继续执行，就是说，将退出该方法而把控制返回给表单。用户可以继续输入信息，应用程序不会等待服务器。
这就提出了一个有趣的问题：服务器完成了请求之后会发生什么？答案是什么也不发生，至少对现在的代

码而言如此！显然这样不行，因此服务器在完成通过 XMLHttpRequest 发送给它的请求处理之后需要某种指示说明怎么做。

现在 onreadystatechange 属性该登场了。该属性允许指定一个*回调函数*。回调允许服务器（猜得到吗？）*反向调用* Web 页面中的代码。它也给了服务器一定程度的控制权，当服务器完成请求之后，会查看 XMLHttpRequest 对象，特别是 onreadystatechange 属性。然后调用该属性指定的任何方法。之所以称为回调是因为服务器向网页发起调用，无论网页本身在做什么。比方说，可能在用户坐在椅子上手没有碰键盘的时候调用该方法，但是也可能在用户输入、移动鼠标、滚动屏幕或者点击按钮时调用该方法。它并不关心用户在做什么。

在 JavaScript 中引用函数

JavaScript 是一种弱类型的语言，可以用变量引用任何东西。因此如果声明了一个函数 updatePage()，JavaScript 也将该函数名看作是一个变量。换句话说，可用变量名 updatePage 在代码中引用函数。

这就是称之为异步的原因：用户在一层上操作表单，而在另一层上服务器响应请求并触发 onreadystatechange 属性指定的回调方法。因此需要像 清单 11 一样在代码中指定该方法。

清单 11. 设置回调方法

```
function getCustomerInfo() {
  var phone = document.getElementById("phone").value;
  var url = "/cgi-local/lookupCustomer.php?phone=" + escape(phone);
  request.open("GET", url, true);
  request.onreadystatechange = updatePage;
  request.send(null);
}
```

需要特别注意的是该属性在代码中设置的*位置* —— 它是在调用 send() *之前* 设置的。发送请求之前必须设置该属性，这样服务器在回答完成请求之后才能查看该属性。现在剩下的就只有编写 updatePage() 方法了，这是本文最后一节要讨论的重点。

回页首

处理服务器响应

发送请求，用户高兴地使用 Web 表单（同时服务器在处理请求），而现在服务器完成了请求处理。服务器查看 onreadystatechange 属性确定要调用的方法。除此以外，可以将您的应用程序看作其他应用程序一样，无论是否异步。换句话说，不一定要采取特殊的动作编写响应服务器的方法，只需要改变表单，让

用户访问另一个 URL 或者做响应服务器需要的任何事情。这一节我们重点讨论对服务器的响应和一种典型的动作 —— 即时改变用户看到的表单中的一部分。

回调和 Ajax

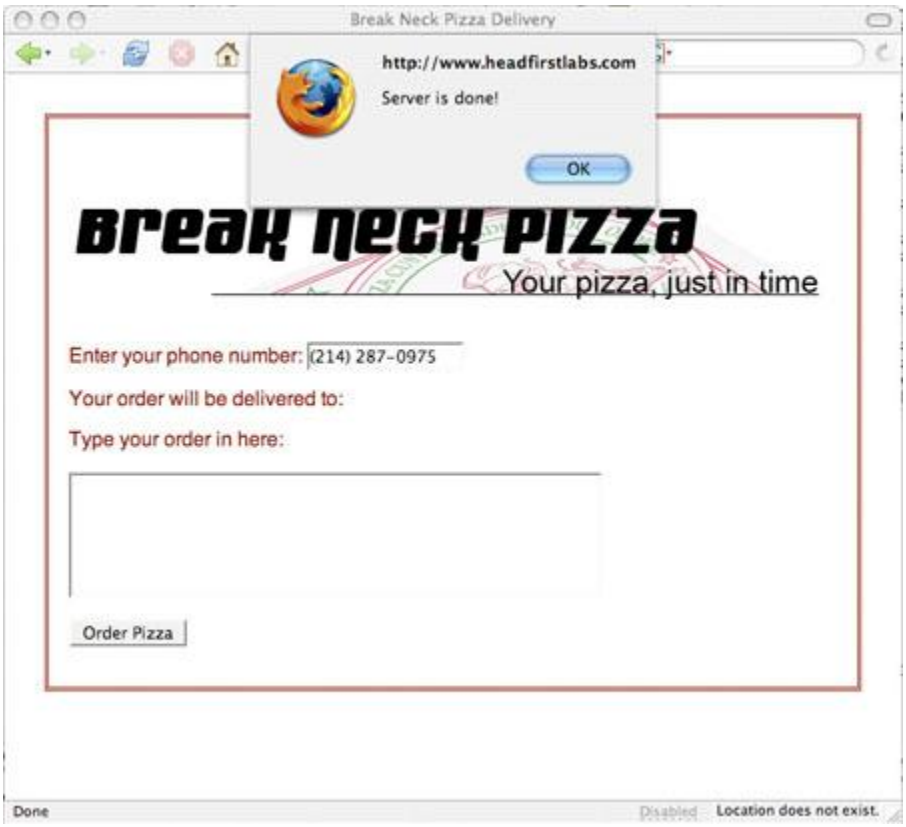
现在我们已经看到如何告诉服务器完成后应该做什么：将 `XMLHttpRequest` 对象的 `onreadystatechange` 属性设置为要运行的函数名。这样，当服务器处理完请求后就会自动调用该函数。也不需要担心该函数的任何参数。我们从一个简单的方法开始，如 清单 12 所示。

清单 12. 回调方法的代码

```
<script language="javascript" type="text/javascript">
   var request = false;
   try {
     request = new XMLHttpRequest();
   } catch (trymicrosoft) {
     try {
       request = new ActiveXObject("Msxml2.XMLHTTP");
     } catch (othermicrosoft) {
       try {
         request = new ActiveXObject("Microsoft.XMLHTTP");
       } catch (failed) {
         request = false;
       }
     }
   }
   if (!request)
     alert("Error initializing XMLHttpRequest!");
   function getCustomerInfo() {
     var phone = document.getElementById("phone").value;
     var url = "/cgi-local/lookupCustomer.php?phone=" + escape(phone);
     request.open("GET", url, true);
     request.onreadystatechange = updatePage;
     request.send(null);
   }
   function updatePage() {
     alert("Server is done!");
   }
</script>
```

它仅仅发出一些简单的警告，告诉您服务器什么时候完成了任务。在自己的网页中试验这些代码，然后在浏览器中打开（如果希望查看该例中的 XHTML，请参阅 清单 8）。输入电话号码然后离开该字段，将看到一个弹出的警告窗口（如 图 3 所示），但是点击 OK 又出现了.....

图 3. 弹出警告的 Ajax 代码



根据浏览器的不同，在表单停止弹出警告之前会看到两次、三次甚至四次警告。这是怎么回事呢？原来我们还没有考虑 HTTP 就绪状态，这是请求/响应循环中的一个重要部分。

HTTP 就绪状态

前面提到，服务器在完成请求之后会在 `XMLHttpRequest` 的 `onreadystatechange` 属性中查找要调用的方法。这是真的，但还不完整。事实上，每当 HTTP 就绪状态改变时它都会调用该方法。这意味着什么呢？首先必须理解 HTTP 就绪状态。

HTTP 就绪状态表示请求的状态或情形。它用于确定该请求是否已经开始、是否得到了响应或者请求/响应模型是否已经完成。它还可以帮助确定读取服务器提供的响应文本或数据是否安全。在 Ajax 应用程序中需要了解五种就绪状态：

- **0**：请求没有发出（在调用 `open()` 之前）。
- **1**：请求已经建立但还没有发出（调用 `send()` 之前）。
- **2**：请求已经发出正在处理之中（这里通常可以从响应得到内容头部）。
- **3**：请求已经处理，响应中通常有部分数据可用，但是服务器还没有完成响应。
- **4**：响应已完成，可以访问服务器响应并使用它。

与大多数跨浏览器问题一样，这些就绪状态的使用也不尽一致。您也许期望任务就绪状态从 0 到 1、2、3 再到 4，但实际上很少是这种情况。一些浏览器从不报告 0 或 1 而直接从 2 开始，然后是 3 和 4。其他浏览器则报告所有的状态。还有一些则多次报告就绪状态 1。在上一节中看到，服务器多次调用 `updatePage()`，每次调用都会弹出警告框 —— 可能和预期的不同！

对于 Ajax 编程，需要直接处理的惟一状态就是就绪状态 4，它表示服务器响应已经完成，可以安全地使用响应数据了。基于此，回调方法中的第一行应该如 清单 13 所示。

清单 13. 检查就绪状态

```
function updatePage() {
  if (request.readyState == 4)
    alert("Server is done!");
}
```

修改后就可以保证服务器的处理已经完成。尝试运行新版本的 Ajax 代码，现在就会看到与预期的一样，只显示一次警告信息了。

HTTP 状态码

虽然 清单 13 中的代码看起来似乎不错，但是还有一个问题 —— 如果服务器响应请求并完成了处理但是报告了一个错误怎么办？要知道，服务器端代码应该明白它是由 Ajax、JSP、普通 HTML 表单或其他类型的代码调用的，但只能使用传统的 Web 专用方法报告信息。而在 Web 世界中，HTTP 代码可以处理请求中可能发生的各种问题。

比方说，您肯定遇到过输入了错误的 URL 请求而得到 404 错误码的情形，它表示该页面不存在。这仅仅是 HTTP 请求能够收到的众多错误码中的一种（完整的状态码列表请参阅 参考资料 中的链接）。表示所访问数据受到保护或者禁止访问的 403 和 401 也很常见。无论哪种情况，这些错误码都是从*完成的响应* 得到的。换句话说，服务器履行了请求（即 HTTP 就绪状态是 4）但是没有返回客户机预期的数据。因此除了就绪状态外，还需要检查 HTTP 状态。我们期望的状态码是 200，它表示一切顺利。如果就绪状态是 4 而且状态码是 200，就可以处理服务器的数据了，而且这些数据应该就是要求的数据（而不是错误或者其他有问题的信息）。因此还要在回调方法中增加状态检查，如 清单 14 所示。

清单 14. 检查 HTTP 状态码

```
function updatePage() {
  if (request.readyState == 4)
    if (request.status == 200)
      alert("Server is done!");
}
```

为了增加更健壮的错误处理并尽量避免过于复杂，可以增加一两个状态码检查，请看一看 清单 15 中修改后的 `updatePage()` 版本。

清单 15. 增加一点错误检查

```
function updatePage() {
  if (request.readyState == 4)
    if (request.status == 200)
      alert("Server is done!");
    else if (request.status == 404)
      alert("Request URL does not exist");
```

```
      else
        alert("Error: status code is " + request.status);
  }
```

现在将 getCustomerInfo() 中的 URL 改为不存在的 URL 看看会发生什么。应该会看到警告信息说明要求的 URL 不存在 —— 好极了!很难处理所有的错误条件,但是这一小小的改变能够涵盖典型 Web 应用程序中 80% 的问题。

读取响应文本

现在可以确保请求已经处理完成（通过就绪状态），服务器给出了正常的响应（通过状态码），最后我们可以处理服务器返回的数据了。返回的数据保存在 XMLHttpRequest 对象的 responseText 属性中。

关于 responseText 中的文本内容，比如格式和长度，有意保持含糊。这样服务器就可以将文本设置成任何内容。比方说，一种脚本可能返回逗号分隔的值，另一种则使用管道符（即 | 字符）分隔的值，还有一种则返回长文本字符串。何去何从由服务器决定。

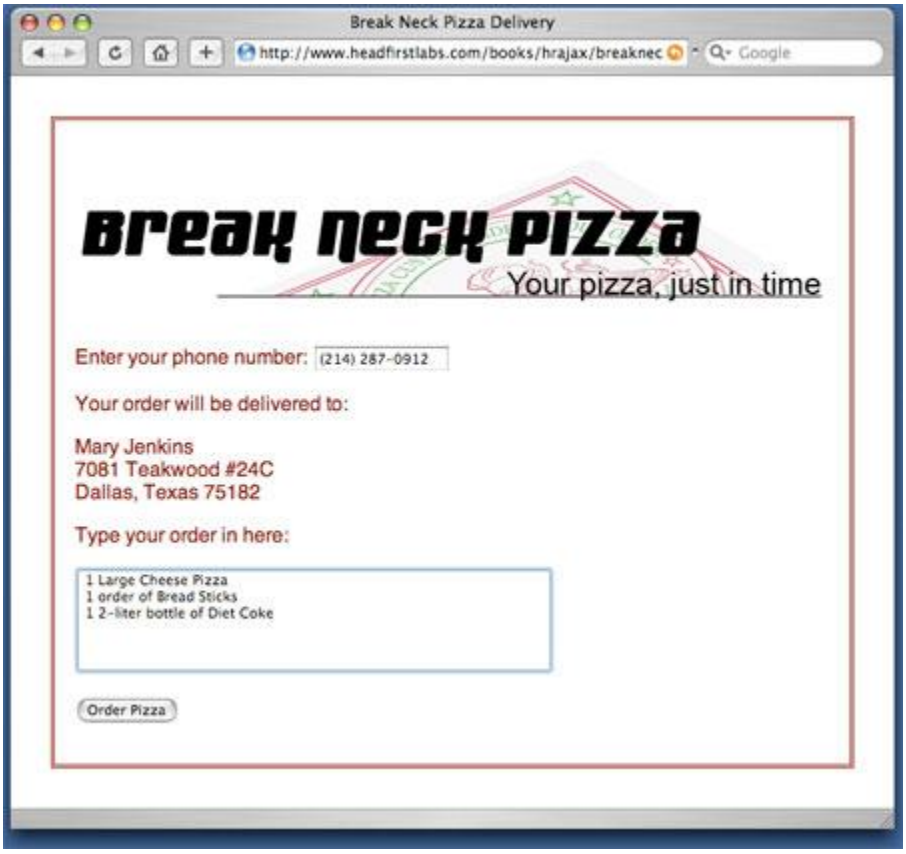
在本文使用的例子中，服务器返回客户的上一个订单和客户地址，中间用管道符分开。然后使用订单和地址设置表单中的元素值，清单 16 给出了更新显示内容的代码。

清单 16. 处理服务器响应

```
  function updatePage() {
    if (request.readyState == 4) {
      if (request.status == 200) {
        var response = request.responseText.split("|");
        document.getElementById("order").value = response[0];
        document.getElementById("address").innerHTML =
          response[1].replace(/\n/g, "
");
      } else
        alert("status is " + request.status);
    }
  }
```

首先，得到 responseText 并使用 JavaScript split() 方法从管道符分开。得到的数组放到 response 中。数组中的第一个值 —— 上一个订单 —— 用 response[0] 访问，被设置为 ID 为 “order” 的字段的值。第二个值 response[1]，即客户地址，则需要更多一点处理。因为地址中的行用一般的行分隔符（“\n”字符）分隔，代码中需要用 XHTML 风格的行分隔符
 来代替。替换过程使用 replace() 函数和正则表达式完成。最后，修改后的文本作为 HTML 表单 div 中的内部 HTML。结果就是表单突然用客户信息更新了，如图 4 所示。

图 4. 收到客户数据后的 Break Neck 表单



结束本文之前，我还要介绍 XMLHttpRequest 的另一个重要属性 responseXML。如果服务器选择使用 XML 响应则该属性包含（也许您已经猜到）XML 响应。处理 XML 响应和处理普通文本有很大不同，涉及到解析、文档对象模型（DOM）和其他一些问题。后面的文章中将进一步介绍 XML。但是因为 responseXML 通常和 responseText 一起讨论，这里有必要提一提。对于很多简单的 Ajax 应用程序 responseText 就够了，但是您很快就会看到通过 Ajax 应用程序也能很好地处理 XML。

回页首

结束语

您可能对 XMLHttpRequest 感到有点厌倦了，我很少看到一整篇文章讨论一个对象，特别是这种简单的对象。但是您将在使用 Ajax 编写的每个页面和应用程序中反复使用该对象。坦白地说，关于 XMLHttpRequest 还真有一些可说的内容。下一期文章中将介绍如何在请求中使用 POST 及 GET，来设置请求中的内容头部和从服务器响应读取内容头部，理解如何在请求/响应模型中编码请求和处理 XML。
再往后我们将介绍常见 Ajax 工具箱。这些工具箱实际上隐藏了本文所述的很多细节，使得 Ajax 编程更容易。您也许会想，既然有这么多工具箱为何还要对底层的细节编码。答案是，如果不知道应用程序在做

什么，就很难发现应用程序中的问题。

因此不要忽略这些细节或者简单地浏览一下，如果便捷华丽的工具箱出现了错误，您就不必挠头或者发送邮件请求支持了。如果了解如何直接使用 XMLHttpRequest，就会发现很容易调试和解决最奇怪的问题。只有让其解决您的问题，工具箱才是好东西。

因此请熟悉 XMLHttpRequest 吧。事实上，如果您有使用工具箱的 Ajax 代码，可以尝试使用 XMLHttpRequest 对象及其属性和方法重新改写。这是一种不错的练习，可以帮助您更好地理解其中的原理。

下一期文章中将进一步讨论该对象，探讨它的一些更有趣的属性（如 responseXML），以及如何使用 POST 请求和以不同的格式发送数据。请开始编写代码吧，一个月后我们再继续讨论。

对于很多 Web 开发人员来说，只需要生成简单的请求并接收简单的响应即可；但是对于希望掌握 Ajax 的开发人员来说，必须要全面理解 HTTP 状态代码、就绪状态和 XMLHttpRequest 对象。在本文中，Brett McLaughlin 将向您介绍各种状态代码，并展示浏览器如何对其进行处理，本文还给出了在 Ajax 中使用的比较少见的 HTTP 请求。

在本系列的 上篇文章 中，我们将详细介绍 XMLHttpRequest 对象，它是 Ajax 应用程序的中心，负责处理服务器端应用程序和脚本的请求，并处理从服务器端组件返回的数据。由于所有的 Ajax 应用程序都要使用 XMLHttpRequest 对象，因此您可能会希望熟悉这个对象，从而能够让 Ajax 执行得更好。

在本文中，我将在上一篇文章的基础上重点介绍这个请求对象的 3 个关键部分的内容：

- HTTP 就绪状态
- HTTP 状态代码
- 可以生成的请求类型

这三部分内容都是在构造一个请求时所要考虑的因素；但是介绍这些主题的内容太少了。然而，如果您不仅仅是想了解 Ajax 编程的常识，而是希望了解更多内容，就需要熟悉就绪状态、状态代码和请求本身的内容。当应用程序出现问题时 —— 这种问题总是存在 —— 那么如果能够正确理解就绪状态、如何生成一个 HEAD 请求或者 400 的状态代码的确切含义，就可以在 5 分钟内调试出问题，而不是在各种挫折和困惑中度过 5 个小时。

下面让我们首先来看一下 HTTP 就绪状态。

深入了解 HTTP 就绪状态

您应该还记得在上一篇文章中 XMLHttpRequest 对象有一个名为 readyState 的属性。这个属性确保服务器已经完成了一个请求，通常会使用一个回调函数从服务器中读出数据来更新 Web 表单或页面的内容。清单 1 给出了一个简单的例子（这也是本系列的上一篇文章中的一个例子 —— 请参见 参考资料）。

XMLHttpRequest 或 XMLHttp：换名玫瑰

Microsoft™ 和 Internet Explorer 使用了一个名为 XMLHttp 的对象，而不是 XMLHttpRequest 对象，而 Mozilla、Opera、Safari 和大部分非 Microsoft 浏览器都使用的是后者。为了简单性起见，我将这两个对象都简单地称为 XMLHttpRequest。这既符合我们在 Web 上看到的情况，又符合 Microsoft 在 Internet Explorer 7.0 中使用 XMLHttpRequest 作为请求对象的意图。(有关这个问题的更多内容，请参见 第 2 部分。)

清单 1. 在回调函数中处理服务器的响应

```
function updatePage() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var response =
request.responseText.split("|");
      document.getElementById("order").value
= response[0];

document.getElementById("address").innerHTML
=
```

```
        response[1].replace(/\n/g, "<br />");
    } else
      alert("status is " + request.status);
  }
}
```

这显然是就绪状态最常见（也是最简单）的用法。正如您从数字 "4" 中可以看出的一样，还有其他几个就绪状态（您在上一篇文章中也看到过这个清单 —— 请参见 参考资料）：

- **0**：请求未初始化（还没有调用 open()）。
- **1**：请求已经建立，但是还没有发送（还没有调用 send()）。
- **2**：请求已发送，正在处理中（通常现在可以从响应中获取内容头）。
- **3**：请求在处理中；通常响应中已有部分数据可用了，但是服务器还没有完成响应的生成。
- **4**：响应已完成；您可以获取并使用服务器的响应了。

如果您希望不仅仅是了解 Ajax 编程的基本知识，那么就不但需要知道这些状态，了解这些状态是何时出现的，以及如何来使用这些状态。首先，您需要学习在每种就绪状态下可能碰到的是哪种请求状态。不幸的是，这一点并不直观，而且会涉及几种特殊的情况。

隐秘就绪状态

第一种就绪状态的特点是 readyState 属性为 0（readyState == 0），表示未初始化状态。一旦对请求对象调用 open() 之后，这个属性就被设置为 1。由于您通常都是在一对请求进行初始化之后就立即调用 open()，因此很少会看到 readyState == 0 的状态。另外，未初始化的就绪状态在实际的应用程序中是没有真正的用处的。

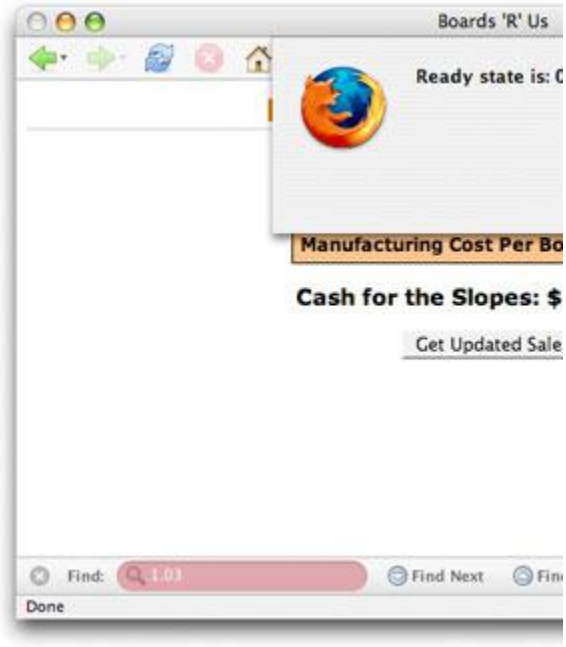
不过为了满足我们的兴趣，请参见 清单 2 的内容，其中显示了如何在 readyState 被设置为 0 时来获取这种就绪状态。

清单 2. 获取 0 就绪状态

```
function getSalesData() {
  // Create a request object
  createRequest();
  alert("Ready state is: " + request.readyState);
  // Setup (initialize) the request
  var url = "/boards/servlet/UpdateBoardSales";
  request.open("GET", url, true);
  request.onreadystatechange = updatePage;
  request.send(null);
}
```

在这个简单的例子中，getSalesData() 是 Web 页面调用来启动请求（例如点击一个按钮时）所使用的函数。注意您必须在调用 open() *之前* 来查看就绪状态。图 1 给出了运行这个应用程序的结果。

图 1. 就绪状态 0



显然，这并不能为您带来多少好处；需要确保 *尚未* 调用 `open()` 函数的情况很少。在大部分 Ajax 编程的真实情况中，这种就绪状态的唯一用法就是使用相同的 `XMLHttpRequest` 对象在多个函数之间生成多个请求。在这种（不常见的）情况中，您可能会在生成新请求之前希望确保请求对象是处于未初始化状态（`readyState == 0`）。这实际上是要确保另外一个函数没有同时使用这个对象。

当 0 等于 4 时

在多个 JavaScript 函数都使用相同的请求对象时，您需要检查就绪状态 0 来确保这个请求对象没有正在使用，这种机制会产生问题。由于 `readyState == 4` 表示一个已完成的请求，因此您经常会发现那些目前没在使用的处于就绪状态的请求对象仍然被设置成了 4 —— 这是因为从服务器返回来的数据已经使用过了，但是从它们被设置为就绪状态之后就没有进行任何变化。有一个函数 `abort()` 会重新设置请求对象，但是这个函数却不是真正为了这个目的而使用的。如果您 *必须* 使用多个函数，最好是为每个函数都创建并使用一个函数，而不是在多个函数之间共享相同的对象。

查看正在处理的请求的就绪状态

除了 0 就绪状态之外，请求对象还需要依次经历典型的请求和响应的其他几种就绪状态，最后才以就绪状态 4 的形式结束。这就是为什么您在大部分回调函数中都可以看到 `if (request.readyState == 4)` 这行代码；它确保服务器已经完成对请求的处理，现在可以安全地更新 Web 页面或根据从服务器返回来的数据来进行操作了。

要查看这种状态发生的过程非常简单。如果就绪状态为 4，我们不仅要运行回调函数中的代码，而且还要在每次调用回调函数时都输出就绪状态。 清单 3 给出了一个实现这种功能的例子。

清单 3. 查看就绪状态

```
function updatePage() {
  // Output the current ready state
  alert("updatePage() called with ready state of " + request.readyState);
}
```

如果您不确定如何运行这个函数，就需要创建一个函数，然后在 Web 页面中调用这个函数，并让它向服务器端的组件发送一个请求（例如 清单 2 给出的函数，或本系列文章的第 1 部分和第 2 部分中给出的例子）。确保在建立请求时，将回调函数设置为 `updatePage()`；要实现这种设置，可以将请求对象的 `onreadystatechange` 属性设置为 `updatePage()`。

这段代码就是 `onreadystatechange` 意义的一个确切展示 —— 每次请求的就绪状态发生变化时，就调用 `updatePage()`，然后我们就可以看到一个警告了。图 2 给出了一个调用这个函数的例子，其中就绪状态为 1。

图 2. 就绪状态 1

您可以自己尝试运行这段代码。将其放入 Web 页面中，然后激活事件处理程序（单击按钮，在域之间按 tab 键切换焦点，或者使用设置的任何方法来触发请求）。这个回调函数会运行多次 —— 每次就绪状态都会改变 —— 您可以看到每个就绪状态的警告。这是跟踪请求所经历的各个阶段的最好方法。

浏览器的不一致性

在对这个过程有一个基本的了解之后，请试着从几个不同的浏览器中访问您的页面。您应该会注意到各个浏览器如何处理这些就绪状态并不一致。例如，在 Firefox 1.5 中，您会看到以下就绪状态：

- 1
- 2
- 3
- 4

这并不奇怪，因为每个请求状态都在这里表示出来了。然而，如果您使用 Safari 来访问相同的应用程序，就应该看到 —— 或者看不到 —— 一些有趣的事情。下面是在 Safari 2.0.1 中看到的状态：

- 2
- 3
- 4

Safari 实际上把第一个就绪状态给丢弃了，也并没有什么明显的原因说明为什么要这样做；不过这就是 Safari 的工作方式。这还说明了一个重要的问题：*尽管在使用服务器上的数据之前确保请求的状态为 4 是一个好主意，但是依赖于每个过渡期就绪状态编写的代码的确会在不同的浏览器上得到不同的结果。*
例如，在使用 Opera 8.5 时，所显示的就绪状态情况就更加糟糕了：

- 3
- 4

最后，Internet Explorer 会显示如下状态：

- 1
- 2
- 3
- 4

如果您碰到请求方面的问题，这就是用来发现问题的 *首要之处*。最好的方式是在 Internet Explorer 和 Firefox 都进行一下测试 —— 您会看到所有这 4 种状态，并可以检查请求的每个状态所处的情况。
接下来我们再来看一下响应端的情况。

显微镜下的响应数据

一旦我们理解在请求过程中发生的各个就绪状态之后，接下来就可以来看一下 `XMLHttpRequest` 对象的另外一个方面了 —— `responseText` 属性。回想一下在上一篇文章中我们介绍过的内容，就可以知道这个属性用来从服务器上获取数据。一旦服务器完成对请求的处理之后，就可以将响应请求数据所需要的任何数据放到请求的 `responseText` 中了。然后回调函数就可以使用这些数据，如 清单 1 和 清单 4 所示。

清单 4. 使用服务器上返回的响应

```
function updatePage() {
  if (request.readyState == 4) {
    var newTotal = request.responseText;
    var totalSoldEl = document.getElementById("total-sold");
    var netProfitEl = document.getElementById("net-profit");
    replaceText(totalSoldEl, newTotal);
    /* 图 out the new net profit */
    var boardCostEl = document.getElementById("board-cost");
    var boardCost = getText(boardCostEl);
    var manCostEl = document.getElementById("man-cost");
    var manCost = getText(manCostEl);
    var profitPerBoard = boardCost - manCost;
    var netProfit = profitPerBoard * newTotal;
    /* Update the net profit on the sales form */
    netProfit = Math.round(netProfit * 100) / 100;
    replaceText(netProfitEl, netProfit);
```

```
    }
```

清单 1 相当简单；清单 4 稍微有点复杂，但是它们在开始时都要检查就绪状态，并获取 responseText 属性的值。

查看请求的响应文本

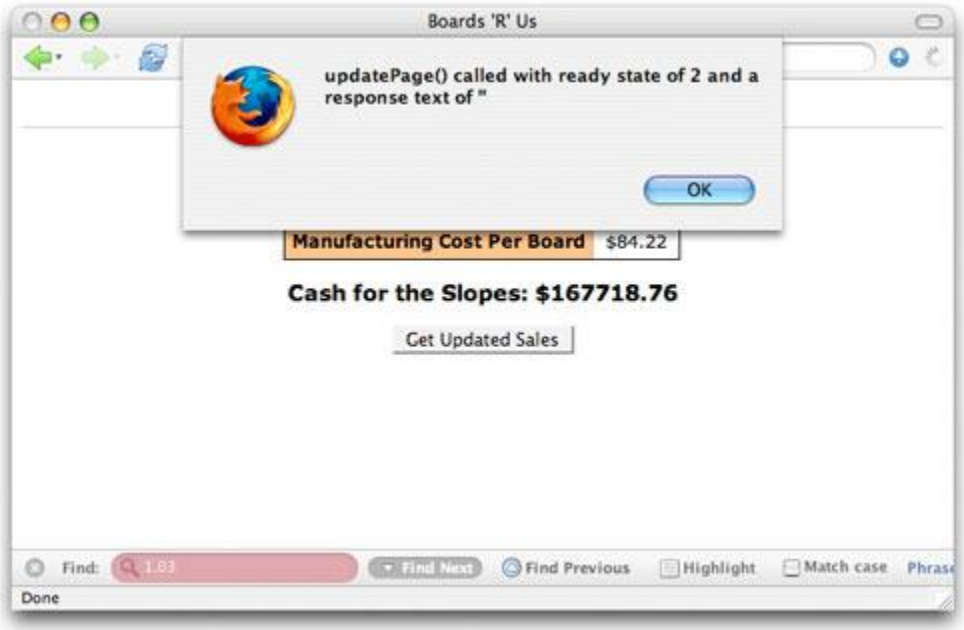
与就绪状态类似，responseText 属性的值在整个请求的生命周期中也会发生变化。要查看这种变化，请使用如 清单 5 所示的代码来测试请求的响应文本，以及它们的就绪状态。

清单 5. 测试 responseText 属性

```
function updatePage() {
  // Output the current ready state
  alert("updatePage() called with ready state of " + request.readyState +
       " and a response text of '" + request.responseText + "'");
  }
```

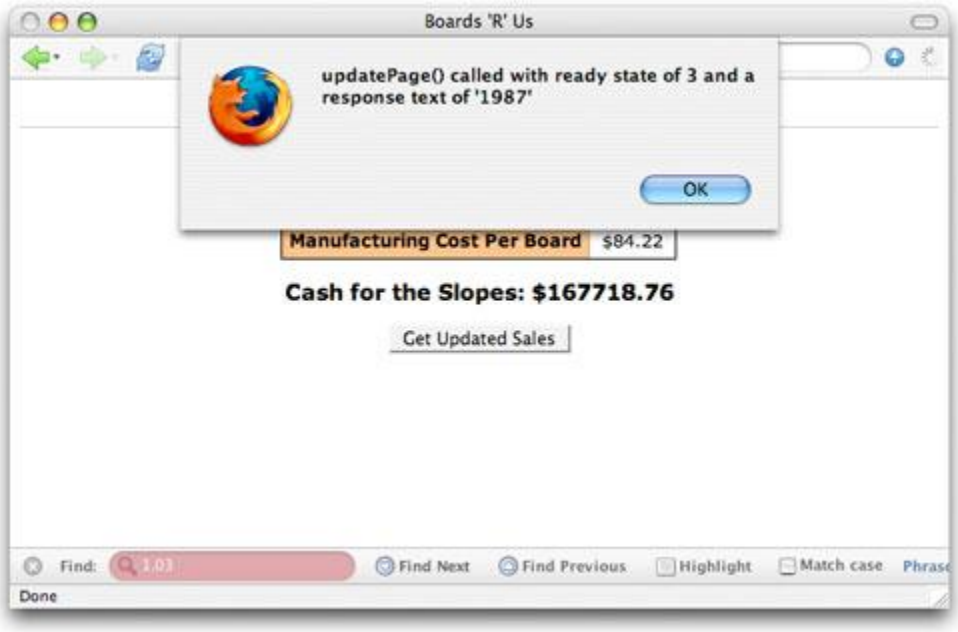
现在在浏览器中打开 Web 应用程序，并激活您的请求。要更好地看到这段代码的效果，请使用 Firefox 或 Internet Explorer，因为这两个浏览器都可以报告出请求过程中所有可能的就绪状态。例如在就绪状态 2 中，就没有定义 responseText （请参见 图 3）；如果 JavaScript 控制台也已经打开了，您就会看到一个错误。

图 3. 就绪状态为 2 的响应文本



不过在就绪状态 3 中，服务器已经在 responseText 属性中放上了一个值，至少在这个例子中是这样（请参见 图 4）。

图 4. 就绪状态为 3 的响应文本



您会看到就绪状态为 3 的响应在每个脚本、每个服务器甚至每个浏览器上都是不一样的。不过，这在调试应用程序中依然是非常有用的。

获取安全数据

所有的文档和规范都强调，只有在就绪状态为 4 时数据才可以安全使用。相信我，当就绪状态为 3 时，您很少能找到无法从 `responseText` 属性获取数据的情况。然而，在应用程序中将自己的逻辑依赖于就绪状态 3 可不是什么好主意 —— 一旦您编写了依赖于就绪状态 3 的完整数据的的代码，几乎就要自己来负责当时的数据不完整问题了。

比较好的做法是向用户提供一些反馈，说明在处于就绪状态 3 时，很快就会有响应了。尽管使用 `alert()` 之类的函数显然不是什么好主意 —— 使用 Ajax 然后使用一个警告对话框来阻塞用户显然是错误的 —— 不过您可以在就绪状态发生变化时更新表单或页面中的域。例如，对于就绪状态 1 来说要将进度指示器的宽度设置为 25%，对于就绪状态 2 来说要将进度指示器的宽度设置为 50%，对于就绪状态 3 来说要将进度指示器的宽度设置为 75%，当就绪状态为 4 时将进度指示器的宽度设置为 100%（完成）。当然，正如您已经看到的一样，这种方法非常聪明，但它是依赖于浏览器的。在 Opera 上，您永远都不会看到前两个就绪状态，而在 Safari 上则没有第一个（1）。由于这个原因，我将这段代码留作练习，而没有在本文中包括进来。

现在应该来看一下状态代码了。

深入了解 HTTP 状态代码

有了就绪状态和您在 Ajax 编程技术中学习到的服务器的响应，您就可以为 Ajax 应用程序添加另外一级复杂性了 —— 这要使用 HTTP 状态代码。这些代码对于 Ajax 来说并没有什么新鲜。从 Web 出现以来，它们就已经存在了。在 Web 浏览器中您可能已经看到过几个状态代码：

- **401**：未经授权
- **403**：禁止

- **404**：没找到

您可以找到更多的状态代码（完整清单请参见 参考资料）。要为 Ajax 应用程序另外添加一层控制和响应（以及更为健壮的错误处理）机制，您需要适当地查看请求和响应中的状态代码。

200：一切正常

在很多 Ajax 应用程序中，您将看到一个回调函数，它负责检查就绪状态，然后继续利用从服务器响应中返回的数据，如 清单 6 所示。

清单 6. 忽略状态代码的回调函数

```
function updatePage() {
  if (request.readyState == 4) {
    var response = request.responseText.split("|");
    document.getElementById("order").value = response[0];
    document.getElementById("address").innerHTML =
      response[1].replace(/\n/g, "<br />");
  }
}
```

这对于 Ajax 编程来说证明是一种短视而错误的方法。如果脚本需要认证，而请求却没有提供有效的证书，那么服务器就会返回诸如 403 或 401 之类的错误代码。然而，由于服务器对请求进行了应答，因此就绪状态就被设置为 4（即使应答并不是请求所期望的也是如此）。最终，用户没有获得有效数据，当 JavaScript 试图使用不存在的服务器数据时就可能会出现严重的错误。

它花费了最小的努力来确保服务器不但完成了一个请求，而且还返回了一个“一切良好”的状态代码。这个代码是 "200"，它是通过 `XMLHttpRequest` 对象的 `status` 属性来报告的。为了确保服务器不但完成了一个请求，而且还报告了一个 OK 状态，请在您的回调函数中添加另外一个检查功能，如 清单 7 所示。

清单 7. 检查有效状态代码

```
function updatePage() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var response = request.responseText.split("|");
      document.getElementById("order").value = response[0];
      document.getElementById("address").innerHTML =
        response[1].replace(/\n/g, "<br />");
    } else
      alert("status is " + request.status);
  }
}
```

通过添加这几行代码，您就可以确认是否存在问题，用户会看到一个有用的错误消息，而不仅仅是看到一个由断章取义的数据所构成的页面，而没有任何解释。

重定向和重新路由

在深入介绍有关错误的内容之前，我们有必要来讨论一下有关一个在使用 Ajax 时 *并不需要* 关心的问题 —— 重定向。在 HTTP 状态代码中，这是 300 系列的状态代码，包括：

- **301**：永久移动
- **302**：找到（请求被重新定向到另外一个 URL/URI 上）
- **305**：使用代理（请求必须使用一个代理来访问所请求的资源）

Ajax 程序员可能并不太关心有关重定向的问题，这是由于两方面的原因：

- 首先，Ajax 应用程序通常都是为一个特定的服务器端脚本、servlet 或应用程序而编写的。对于那些您看不到就消失了的组件来说，Ajax 程序员就不太清楚了。因此有时您会知道资源已经移动了（因为您移动了它，或者通过某种手段移动了它），接下来要修改请求中的 URL，并且不会再碰到这种结果了。
- 更为重要的一个原因是：Ajax 应用程序和请求都是封装在沙盒中的。这就意味着提供生成 Ajax 请求的 Web 页面的域必须是对这些请求进行响应的域。因此 ebay.com 所提供的 Web 页面就不能对一个在 amazon.com 上运行的脚本生成一个 Ajax 风格的请求；在 ibm.com 上的 Ajax 应用程序也无法对在 netbeans.org 上运行的 servlets 发出请求。

结果是您的请求无法重定向到其他服务器上，而不会产生安全性错误。在这些情况中，您根本就不会得到状态代码。通常在调试控制台中都会产生一个 JavaScript 错误。因此，在对状态代码进行充分的考虑之后，您就可以完全忽略重定向代码的问题了。

错误

一旦接收到状态代码 200 并且意识到可以很大程度上忽略 300 系列的状态代码之后，所需要担心的唯一一组代码就是 400 系列的代码了，这说明了不同类型的错误。回头再来看一下 清单 7，并注意在对错误进行处理时，只将少数常见的错误消息输出给用户了。尽管这是朝正确方向前进的一步，但是要告诉从事应用程序开发的用户和程序员究竟发生了什么问题，这些消息仍然是没有太大用处的。

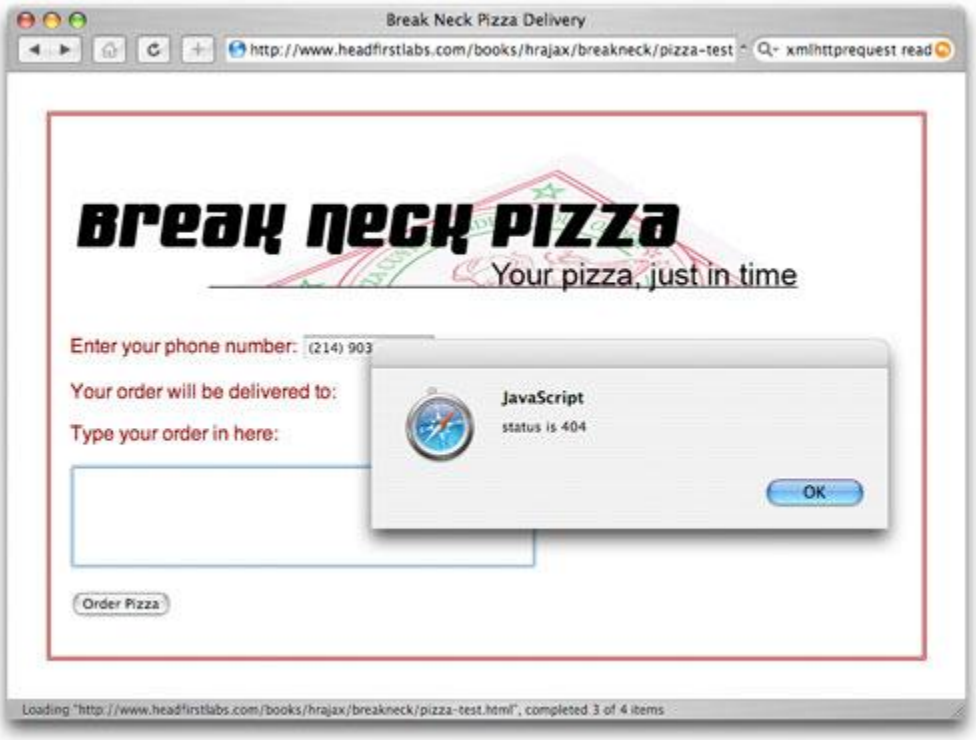
首先，我们要添加对找不到的页的支持。实际上这在大部分产品系统中都不应该出现，但是在测试脚本位置发生变化或程序员输入了错误的 URL 时，这种情况并不罕见。如果您可以自然地报告 404 错误，就可以为那些困扰不堪的用户和程序员提供更多帮助。例如，如果服务器上的一个脚本被删除了，我们就可以使用 清单 7 中的代码，这样用户就会看到一个如 图 5 所示的非描述性错误。

边界情况和困难情况

看到现在，一些新手程序员就可能会这究竟是要讨论什么内容。有一点事实大家需要知道：只有不到 5% 的 Ajax 请求需要使用诸如 2、3 之类的就绪状态和诸如 403 之类的状态代码（实际上，这个比率可能更接近于 1% 甚至更少）。这些情况非常重要，称为 *边界情况（edge case）* —— 它们只会在一些非常特殊的情况下发生，其中遇到的都是最奇特的问题。虽然这些情况并不普遍，但是这些边界情况却占据了大部分用户所碰到的问题的 80%!

对于典型的用户来说，应用程序 100 次都是正常工作的这个事实通常都会被忘记，然而应用程序只要一次出错就会被他们清楚地记住。如果您可以很好地处理边界情况（或困难情况），就可以为再次访问站点的用户提供满意的回报。

图 5. 常见错误处理



用户无法判断问题究竟是认证问题、没找到脚本（此处就是这种情况）、用户错误还是代码中有些地方产生了问题。添加一些简单的代码可以让这个错误更加具体。请参照 清单 8，它负责处理没找到的脚本或认证发生错误的情况，在出现这些错误时都会给出具体的消息。

清单 8. 检查有效状态代码

```
function updatePage() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var response = request.responseText.split("|");
      document.getElementById("order").value = response[0];
      document.getElementById("address").innerHTML =
        response[1].replace(/\n/g, "<br />");
    } else if (request.status == 404) {
      alert ("Requested URL is not found.");
    } else if (request.status == 403) {
      alert("Access denied.");
    } else
      alert("status is " + request.status);
  }
}
```

虽然这依然相当简单，但是它的确多提供了一些有用的信息。图 6 给出了与 图 5 相同的错误，但是这一次错误处理代码向用户或程序员更好地说明了究竟发生了什么。

图 6. 特殊错误处理

在我们自己的应用程序中，可以考虑在发生认证失败的情况时清除用户名和密码，并向屏幕上添加一条错误消息。我们可以使用类似的方法来更好地处理找不到脚本或其他 400 类型的错误（例如 405 表示不允许使用诸如发送 HEAD 请求之类不可接受的请求方法，而 407 则表示需要进行代理认证）。然而不管采用哪种选择，都需要从对服务器上返回的状态代码开始入手进行处理。

其他请求类型

如果您真希望控制 `XMLHttpRequest` 对象，可以考虑最后实现这种功能 —— 将 HEAD 请求添加到指令中。在前两篇文章中，我们已经介绍了如何生成 GET 请求；在马上就要发表的一篇文章中，您会学习有关使用 POST 请求将数据发送到服务器上的知识。不过本着增强错误处理和信息搜集的精神，您应该学习如何生成 HEAD 请求。

生成请求

实际上生成 HEAD 请求非常简单；您可以使用 "HEAD"（而不是 "GET" 或 "POST"）作为第一个参数来调用 `open()` 方法，如 清单 9 所示。

清单 9. 使用 Ajax 生成一个 HEAD 请求

```
function getSalesData() {
  createRequest();
  var url = "/boards/servlet/UpdateBoardSales";
  request.open("HEAD", url, true);
  request.onreadystatechange = updatePage;
```

```
  request.send(null);
}
```

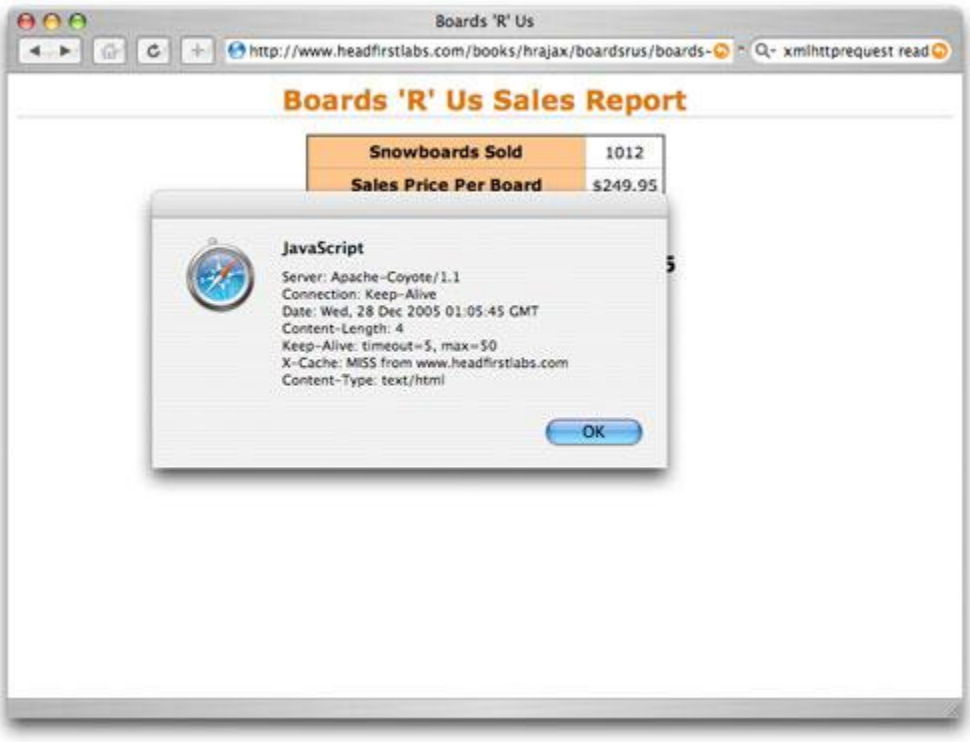
当您这样生成一个 HEAD 请求时，服务器并不会像对 GET 或 POST 请求一样返回一个真正的响应。相反，服务器只会返回资源的 *头（header）*，这包括响应中内容最后修改的时间、请求资源是否存在和很多其他有用信息。您可以在服务器处理并返回资源之前使用这些信息来了解有关资源的信息。
对于这种请求您可以做的最简单的事情就是简单地输出所有的响应头的内容。这可以让您了解通过 HEAD 请求可以使用什么。清单 10 提供了一个简单的回调函数，用来输出从 HEAD 请求中获得的响应头的内容。

清单 10. 输出从 HEAD 请求中获得的响应头的内容

```
function updatePage() {
  if (request.readyState == 4) {
    alert(request.getAllResponseHeaders());
  }
}
```

请参见 图 7，其中显示了从一个向服务器发出的 HEAD 请求的简单 Ajax 应用程序返回的响应头。

图 7. HEAD 请求的响应头



您可以单独使用这些头（从服务器类型到内容类型）在 Ajax 应用程序中提供其他信息或功能。

检查 URL

您已经看到了当 URL 不存在时应该如何检查 404 错误。如果这变成一个常见的问题 —— 可能是缺少了一个特定的脚本或 servlet —— 那么您就可能会希望在生成完整的 GET 或 POST 请求之前来检查这个 URL。要实现这种功能，生成一个 HEAD 请求，然后在回调函数中检查 404 错误；清单 11 给出了一个简单的回调函数。

清单 11. 检查某个 URL 是否存在

```
function updatePage() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      alert("URL exists");
    } else if (request.status == 404) {
      alert("URL does not exist.");
    } else {
      alert("Status is: " + request.status);
    }
  }
}
```

诚实地说，这段代码的价值并不太大。服务器必须对请求进行响应，并构造一个响应来填充内容长度的响应头，因此并不能节省任何处理时间。另外，这花费的时间与生成请求并使用 HEAD 请求来查看 URL 是否存在所需要的时间一样多，因为它要生成使用 GET 或 POST 的请求，而不仅仅是如 清单 7 所示一样来处理错误代码。不过，有时确切地了解目前什么可用也是非常有用的；您永远不会知道何时创造力就会迸发或者何时需要 HEAD 请求！

有用的 HEAD 请求

您会发现 HEAD 请求非常有用的一个领域是用来查看内容的长度或内容的类型。这样可以确定是否需要发回大量数据来处理请求，和服务器是否试图返回二进制数据，而不是 HTML、文本或 XML（在 JavaScript 中，这 3 种类型的数据都比二进制数据更容易处理）。

在这些情况中，您只使用了适当的头名，并将其传递给 `XMLHttpRequest` 对象的 `getResponseHeader()` 方法。因此要获取响应的长度，只需要调用 `request.getResponseHeader("Content-Length");`。要获取内容类型，请使用 `request.getResponseHeader("Content-Type");`。

在很多应用程序中，生成 HEAD 请求并没有增加任何功能，甚至可能会导致请求速度变慢（通过强制生成一个 HEAD 请求来获取有关响应的数据，然后在使用一个 GET 或 POST 请求来真正获取响应）。然而，在出现您不确定有关脚本或服务器端组件的情况时，使用 HEAD 请求可以获取一些基本的数据，而不需要对响应数据真正进行处理，也不需要大量的带宽来发送响应。

结束语

对于很多 Ajax 和 Web 程序员来说，本文中介绍的内容似乎是太高级了。生成 HEAD 请求的价值是什么呢？到底在什么情况下需要在 JavaScript 中显式地处理重定向状态代码呢？这些都是很好的问题；对于简单的应用程序来说，答案是这些高级技术的价值并不是非常大。

然而，Web 已经不再是只需实现简单应用程序的地方了；用户已经变得更加高级，客户期望能够获得更好的稳定性、更高级的错误报告，如果应用程序有 1% 的时间停机，那么经理就可能会因此而被解雇。

因此您的工作就不能仅仅局限于简单的应用程序了，而是需要更深入理解 XMLHttpRequest。

- 如果您可以考虑各种就绪状态 —— 并且理解了这些就绪状态在不同浏览器之间的区别 —— 就可以快速调试应用程序了。您甚至可以基于就绪状态而开发一些创造性的功能，并向用户和客户回报请求的状态。
- 如果您要对状态代码进行控制，就可以设置应用程序来处理脚本错误、非预期的响应以及边缘情况。结果是应用程序在所有的时间都可以正常工作，而不仅仅是只能一切都正常的情况下才能运行。
- 增加这种生成 HEAD 请求的能力，检查某个 URL 是否存在，以及确认某个文件是否被修改过，这样就可以确保用户可以获得有效的页面，用户所看到的信息都是最新的，（最重要的是）让他们惊讶这个应用程序是如何健壮和通用。

本文的目的并非是要让您的应用程序显得十分华丽，而是帮助您去掉黄色聚光灯后重点昭显文字的美丽，或者外观更像桌面一样。尽管这些都是 Ajax 的功能（在后续几篇文章中就会介绍），不过它们却像是蛋糕表面的一层奶油。如果您可以使用 Ajax 来构建一个坚实的基础，让应用程序可以很好地处理错误和问题，用户就会返回您的站点和应用程序。在接下来的文章中，我们将添加这种直观的技巧，这会让客户兴奋得发抖。（认真地说，您一定不希望错过下一篇文章!）

第 4 部分: 利用 DOM 进行 Web 响应

程序员（使用后端应用程序）和 Web 程序员（编写 HTML、CSS 和 JavaScript 上）之间的分水岭是长久存在的。但是，Document Object Model (DOM) 弥补了这个裂缝，使得在后端使用 XML 同时在前端使用 HML 切实可行，并成为极其有效的工具。在本文中，Brett McLaughlin 介绍了 Document Object Model，解释它在 Web 页面中的应用，并开始挖掘其在 JavaScript 中的用途。
与许多 Web 程序员一样，您可能使用过 HTML。HTML 是程序员开始与 Web 页面打交道的方式；HTML 通常是他们完成应用程序或站点前的最后一步——调整一些布局、颜色或样式。不过，虽然经常使用 HTML，但对于 HTML 转到浏览器呈现在屏幕上时到底发生了什么，人们普遍存在误解。在我分析您认为可能发生的事情及其可能错误的原因之前，我希望您对设计和服务 Web 页面时涉及的过程一清二楚：

1. 一些人（通常是您！）在文本编辑器或 IDE 中创建 HTML。
2. 然后您将 HTML 上载到 Web 服务器，例如 Apache HTTPD，并将其公开在 Internet 或 intranet 上。
3. 用户用 Firefox 或 SafariA 等浏览器请求您的 Web 页面。
4. 用户的浏览器向您的服务器请求 HTML。
5. 浏览器将从服务器接收到的页面以图形和文本方式呈现；用户看到并激活 Web 页面。

这看起来非常基础，但事情很快会变得有趣起来。事实上，步骤 4 和步骤 5 之间发生的巨大数量的“填充物（stuff）”就是本文的焦点。术语“填充物”也十分适用，因为多数程序员从来没有真正考虑过当用户浏览器请求显示标记时到底在标记身上发生了什么。

- 是否浏览器只是读取 HTML 中的文本并将其显示？
- CSS 呢？尤其是当 CSS 位于外部文件时。
- JavaScript 呢？它也通常位于外部文件中。
- 浏览器如何处理这些项，如果将事件处理程序、函数和样式映射到该文本标记？

实践证明，所有这些问题的答案都是 Document Object Model。因此，废话少说，直接研究 DOM。

Web 程序员和标记

对于多数程序员，当 Web 浏览器开始时他们的工作就结束了。也就是说，将一个 HTML 文件放入 Web 浏览器的目录上后，您通常就认为它已经“完成”，而且（满怀希望地）认为*再也不会考虑它*！说到编写干净、组织良好的页面时，这也是一个伟大的目标；希望您的标记跨浏览器、用各种版本的 CSS 和 JavaScript 显示它应该显示的内容，一点错都没有。
问题是这种方法限制了程序员对浏览器中真正发生的事情的理解。更重要的是，它限制了您用客户端 JavaScript 动态更新、更改和重构 Web 页面的能力。摆脱这种限制，让您的 Web 站点拥有更大的交互性和创造性。

程序员做什么

作为典型的 Web 程序员，您可能启动文本编辑和 IDE 后就开始输入 HTML、CSS 甚至 JavaScript。很容易认为这些标记、选择器和属性只是使站点正确显示而做的小小的任务。但是，在这一点上您需要拓展您的思路，要意识到您是在组织您的内容。不要担心；我保证这不会变成关于标记美观、您必须如何认识到 Web 页面的真正潜力或其他任何元物质的讲座。您需要了解的是您在 Web 开发中到底是什么角色。
说到页面的外观，顶多您只能提提建议。您提供 CSS 样式表时，用户可以覆盖您的样式选择。您提供字体大小时，用户浏览器可以为视障者更改这些大小，或者在大显示器（具有同等大的分辨率）上按比例缩

小。甚至您选择的颜色和字体也受制于用户显示器和用户在其系统上安装的字体。虽然尽您所能来设计页面样式很不错，但这*绝不是* 您对 Web 页面的最大影响。

您绝对控制的是 Web 页面的*结构*。您的标记不可更改，用户就不能乱弄；他们的浏览器只能从您的 Web 服务器检索标记并显示它（虽然样式更符合用户的品味而不是您自己的品味）。但页面组织，不管是在该段落内还是在其他分区，都只由您单独决定。要是想实际更改您的页面（这是大多数 Ajax 应用程序所关注的），您操作的是页面的结构。尽管很容易更改一段文本的颜色，但在现有页面上添加文本或整个区段要难得多。不管用户如何设计该区段的样式，都是由您控制页面本身的组织。

标记做什么

一旦意识到您的标记是真正与组织相关的，您就会对它另眼相看了。不会认为 `h1` 导致文本是大字号、黑色、粗体的，而会认为 `h1` 是标题。用户如何看待这个问题以及他们是使用您的 CSS、他们自己的 CSS 还是这两者的组合，这是次要的考虑事项。相反，要意识到只有标记才能提供这种级别的组织；`p` 指明文本在段落内，`img` 表示图像，`div` 将页面分成区段，等等。

还应该清楚，样式和行为（事件处理程序和 JavaScript）是在*事后* 应用于该组织的。标记就绪以后才能对其进行操作或设计样式。所以，正如您可以将 CSS 保存在 HTML 的外部文件中一样，标记的组织与其样式、格式和行为是分离的。虽然您肯定可以用 JavaScript 更改元素或文本的样式，但实际更改您的标记所布置的组织却更加有趣。

只要牢记您的标记只为您的页面提供组织、框架，您就能立于不败之地。再前进一小步，您就会明白浏览器是如何接受所有的文本组织并将其转变为超级有趣的一些东西的，即一组对象，其中每个对象都可被更改、添加或删除。

文本标记的优点

在讨论 Web 浏览器之前，值得考虑一下为什么纯文本*绝对* 是存储 HTML 的最佳选择（有关详细信息，请参阅 有关标记的一些其他想法）。不考虑优缺点，只是回忆一下在每次查看页面时 HTML 是通过网络发送到 Web 浏览器的（为了简洁，不考虑高速缓存等）。真是再没有比传递文本再有效的方法了。二进制对象、页面图形表示、重新组织的标记块等等，所有这一切都比纯文本文件通过网络传递要更困难。

此外，浏览器也为此增光添彩。今天的浏览器允许用户更改文本大小、按比例伸缩图像、下载页面的 CSS 或 JavaScript（大多数情况），甚至更多，这完全排除了将任何类型的页面图形表示发送到浏览器上。但是，浏览器需要原 HTML，这样它才能在浏览器中对页面应用任何处理，而不是信任浏览器去处理该任务。同样地，将 CSS 从 JavaScript 分离和将 CSS 从 HTML 标记分离要求一种容易分离的格式。文本文件又一次成为该任务的最好方法。

最后但同样重要的一点是，记住，新标准（比如 HTML 4.01 与 XHTML 1.0 和 1.1）承诺将内容（页面中的数据）与表示和样式（通常由 CSS 应用）分离。如果程序员要将 HTML 与 CSS 分离，然后强制浏览器检索粘结页面各部分的一些页面表示，这会失去这些标准的多数优点。保持这些部分到达浏览器时都一直分离使得浏览器在从服务器获取 HTML 时有了前所未有的灵活性。

关于标记的其他想法

纯文本编辑：是对是错？

纯文本是存储标记的理想选择，但是不适合*编辑* 标记。大行其道的是使用 IDE，比如 Macromedia DreamWeaver 或更强势点的 Microsoft® FrontPage®，来操作 Web 页面标记。这些环境通常提供快捷方式和帮助来创建 Web 页面，尤其是在使用 CSS 和 JavaScript 时，二者都来自实际页面标记以外的文件。许多人仍偏爱好用古老的记事本或 vi（我承认我也是其中一员），这并不要紧。不管怎样，最终结果都是充满标记的文本文件。

网络上的文本：好东西

已经说过，文本是文档的最好媒体，比如 HTML 或 CSS，在网络上被千百次地传输。当我说浏览器表示文本很难时，是特指将文本转换为用户查看的可视图形页面。这与浏览器实际上如何从 Web 浏览器检索页面没有关系；在这种情况下，文本仍然是最佳选择。

进一步了解 Web 浏览器

对于一些 Web 程序员来说，在前文阅读到的所有内容可能是对您在 Web 开发过程中角色的滑稽讲述。但说到浏览器的行为，许多最能干的 Web 设计人员和开发人员通常都没有意识到“内幕”的实际状况。我将在本节重点进行讲述。不要担心，代码很快就到，但是要克制您编码的急躁心情，因为真正了解 Web 浏览器的确切行为对于您的代码正确工作是非常关键的。

文本标记的缺点

正如文本标记对于设计人员和页面创建者具有惊人的优点之外，它对于浏览器也具有相当出奇的缺点。具体来说，浏览器很难直接将文本标记可视地表示给用户（详细信息请参阅 有关标记的一些其他想法）。考虑下列常见的浏览器任务：

- 基于元素类型、类、ID 及其在 HTML 文档中的位置，将 CSS 样式（通常来自外部文件中的多个样式表）应用于标记。
- 基于 JavaScript 代码（通常位于外部文件）将样式和格式应用于 HTML 文档的不同部分。
- 基于 JavaScript 代码更改表单字段的值。
- 基于 JavaScript 代码，支持可视效果，比如图像翻转和图像交换。

复杂性并不在于编码这些任务；其中每件事都是相当容易的。复杂性来自实际实现请求动作的浏览器。如果标记存储为文本，比如，想要在 `center-text` 类的 `p` 元素中输入文本 (`text-align: center`)，如何实现呢？

- 将内联样式添加到文本吗？
- 将样式应用到浏览器中的 HTML 文本，并只保持内容居中或不居中？
- 应用无样式的 HTML，然后事后应用格式？

这些非常困难的问题是如今很少有人编写浏览器的原因。（编写浏览器的人应该接受最由衷的感谢）
无疑，纯文本不是存储浏览器 HTML 的好办法，尽管文本是获取页面标记最好的解决方案。如果加上 JavaScript *更改* 页面结构的能力，事情就变得有些微妙了。浏览器应该将修改过的结构重新写入磁盘吗？如何才能保持文档的最新版本呢？
无疑，文本不是答案。它难以修改，为其应用样式和行为很困难，与今天 Web 页面的动态本质在根本上相去甚远。

求助于树视图

这个问题的答案（至少是由当今 Web 浏览器选择的答案）是使用树结构来表示 HTML。参见 清单 1，这是一个表示为本文标记的相当简单又无聊的 HTML 页面。

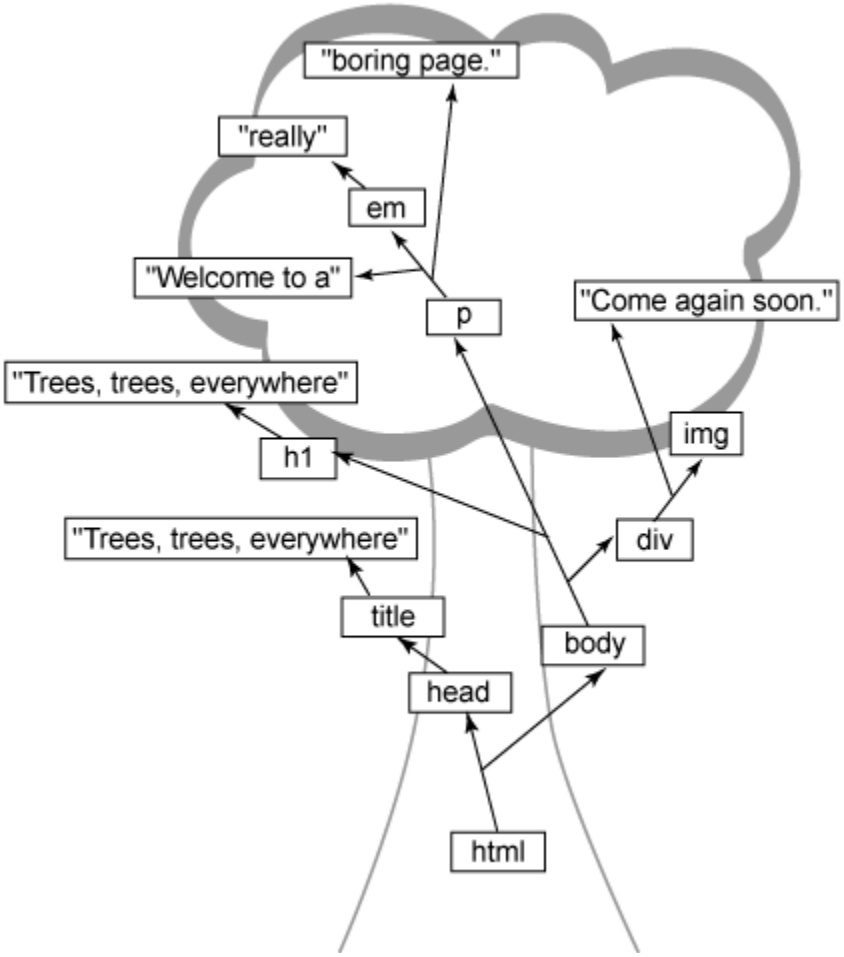
清单 1. 文本标记中的简单 HTML 页面

```
<html>
 <head>
  <title>Trees, trees, everywhere</title>
 </head>
 <body>
  <h1>Trees, trees, everywhere</h1>
  <p>Welcome to a <em>really</em> boring page.</p>
  <div>
```

```
    Come again soon.
    <img src="come-again.gif" />
  </div>
 </body>
</html>
```

浏览器接受该页面并将之转换为树形结构，如图 1 所示。

图 1. 清单 1 的树视图



为了保持本文的进度，我做了少许简化。DOM 或 XML 方面的专家会意识到空白对于文档文本在 Web 浏览器树结构中表示和分解方式的影响。肤浅的了解只会使事情变得模棱两可，所以如果想弄清空白的影响，那最好不过了；如果不想的话，那可以继续读下去，不要考虑它。当它成为问题时，那时您就会明白您需要的一切。

除了实际的树背景之外，可能会首先注意到树中的一切是以最外层的 HTML 包含元素，即 `html` 元素开始的。使用树的比喻，这叫做*根元素*。所以即使这是树的底层，当您查看并分析树的时候，我也通常以此开始。如果它确实奏效，您可以将整个树颠倒一下，但这确实有些拓展了树的比喻。

从根流出的线表示不同标记部分之间的关系。`head` 和 `body` 元素是 `html` 根元素的*孩子*；`title` 是 `head` 的孩子，而文本“Trees, trees, everywhere”是 `title` 的孩子。整个树就这样组织下去，直到浏览器获得与 图 1 类似的结构。

一些附加术语

为了沿用树的比喻，head 和 body 被叫做 html 的*分支*(*branches*)。叫分支是因为它们有自己的孩子。当到达树的末端时，您将进入主要的文本，比如“Trees, trees, everywhere”和“really”；这些通常称为*叶子*，因为它们没有自己的孩子。您不需要记住所有这些术语，当您试图弄清楚特定术语的意思时，只要想像一下树结构就容易多了。

对象的值

既然了解了一些基本的术语，现在应该关注一下其中包含元素名称和文本的小矩形了（图 1）。每个矩形是一个对象；浏览器在其中解决一些文本问题。通过使用对象来表示 HTML 文档的每一部分，可以很容易地更改组织、应用样式、允许 JavaScript 访问文档，等等。

对象类型和属性

标记的每个可能类型都有自己的对象类型。例如，HTML 中的元素用 Element 对象类型表示。文档中的文本用 Text 类型表示，属性用 Attribute 类型表示，以此类推。

所以 Web 浏览器不仅可以使用对象模型来表示文档（从而避免了处理静态文本），还可以用对象类型立即辨别出某事物是什么。HTML 文档被解析并转换为对象集合，如 图 1 所示，然后尖括号和转义序列（例如，使用 < 表示 <，使用 > 表示 >）等事物不再是问题了。这就使得浏览器的工作（至少在解析输入 HTML 之后）变得更容易。弄清某事物究竟是元素还是属性并确定如何处理该类型的对象，这些操作都十分简单了。

通过使用对象，Web 浏览器可以更改这些对象的属性。例如，每个元素对象具有一个父元素和一系列子元素。所以添加新的子元素或文本只需要向元素的子元素列表中添加一个新的子元素。这些对象还具有 style 属性，所以快速更改元素或文本段的样式非常简单。例如，要使用 JavaScript 更改 div 的高度，如下所示：

```
someDiv.style.height = "300px";
```

换句话说，Web 浏览器使用对象属性可以非常容易地更改树的外观和结构。将之比作浏览器在内部将页面表示为文本时必须进行的复杂事情，每次更改属性或结构都需要浏览器重新编写静态文件、重新解析并在屏幕上重新显示。有了对象，所有这一切都解决了。

现在，花点时间展开一些 HTML 文档并用树将其勾画出来。尽管这看起来是个不寻常的请求（尤其是在包含极少代码的这样一篇文章中），如果您希望能够操纵这些树，那么需要熟悉它们的结构。

在这个过程中，可能会发现一些古怪的事情。比如，考虑下列情况：

- 属性发生了什么？
- 分解为元素（比如 em 和 b）的文本呢？
- 结构不正确（比如当缺少结束 p 标记时）的 HTML 呢？

一旦熟悉这些问题之后，就能更好地理解下面几节了。

严格有时是好事

如果尝试刚提到的练习 I，您可能会发现标记的树视图中存在一些潜在问题（如果不练习的话，那就听我说吧!）。事实上，在 清单 1 和 图 1 中就会发现一些问题，首先看 p 元素是如何分解的。如果您问通常的 Web 开发人员“p 元素的文本内容是什么”，最常见的答案将是“Welcome to a really boring Web page.”。如果将之与图 1 做比较，将会发现这个答案（虽然合乎逻辑）是根本不正确的。

实际上，p 元素具有*三个*不同的子对象，其中没有一个包含完整的“Welcome to a really boring Web page.”文本。您会发现文本的一部分，比如“Welcome to a ”和“ boring Web page”，但不是全部。为了理解这一点，记住标记中的任何内容都必须转换为某种类型的对象。

此外，顺序无关紧要！如果浏览器显示正确的对象，但显示顺序与您在 HTML 中提供的顺序不同，那么

您能想像出用户将如何响应 Web 浏览器吗？段落夹在页面标题和文章标题中间，而这不是您自己组织文档时的样式呢？很显然，浏览器必须保持元素和文本的顺序。
在本例中，p 元素有三个不同部分：

- em 元素之前的文本
- em 元素本身
- em 元素之后的文本

如果将该顺序打乱，可能会把重点放在文本的错误部分。为了保持一切正常，p 元素有三个子对象，其顺序是在 清单 1 的 HTML 中显示的顺序。而且，重点文本“really”*不是* p 的子元素；而是 p 的子元素 em 的子元素。
理解这一概念非常重要。尽管“really”文本将可能与其他 p 元素文本一起显示，但它仍是 em 元素的直接子元素。它可以具有与其他 p 文本不同的格式，而且可以独立于其他文本到处移动。
要将之牢记在心，试着用图表示清单 2 和 3 中的 HTML，确保文本具有正确的父元素（而不管文本最终会如何显示在屏幕上）。

清单 2. 带有巧妙元素嵌套的标记

```
<html>
 <head>
  <title>This is a little tricky</title>
 </head>
 <body>
  <h1>Pay <u>close</u> attention, OK?</h1>
  <div>
   <p>This p really isn't <em>necessary</em>, but it makes the
     <span id="bold-text">structure <i>and</i> the organization</span>
     of the page easier to keep up with.</p>
  </div>
 </body>
</html>
```

清单 3. 更巧妙的元素嵌套

```
<html>
 <head>
  <title>Trickier nesting, still</title>
 </head>
 <body>
  <div id="main-body">
```

```
  <div id="contents">
   <table>
    <tr><th>Steps</th><th>Process</th></tr>
    <tr><td>1</td><td>Figure out the <em>root element</em>.</td></tr>
    <tr><td>2</td><td>Deal with the <span id="code">head</span> first,
        as it's usually easy.</td></tr>
    <tr><td>3</td><td>Work through the <span id="code">body</span>.
        Just <em>take your time</em>.</td></tr>
   </table>
  </div>
  <div id="closing">
   This link is <em>not</em> active, but if it were, the answers
   to this <a href="answers.html"><img src="exercise.gif" /></a> would
   be there. But <em>do the exercise anyway!</em>
  </div>
 </div>
</body>
</html>
```

在本文末的 GIF 文件 图 2 中的 tricky-solution.gif 和 图 3 中的 trickier-solution.gif 中将会找到这些练习的答案。不要偷看，先花些时间自动解答一下。这样能帮助您理解组织树时应用的规则有多么严格，并真正帮助您掌握 HTML 及其树结构。

属性呢？

当您试图弄清楚如何处理属性时，是否遇到一些问题呢？前已提及，属性确实具有自己的对象类型，但属性确实不是显示它的元素的子元素，嵌套元素和文本不在同一属性“级别”，您将注意到，清单 2 和 3 中练习的答案没有显示属性。

属性事实上存储在浏览器使用的对象模型中，但它们有一些特殊情况。每个元素都有可用属性的列表，且与子对象列表是分离的。所以 `div` 元素可能有一个包含属性“id”和另一个属性“class”的列表。

记住，元素的属性必须具有惟一的名称，也就是说，一个元素不能有两个“id”或两个“class”属性。这使得列表易于维护和访问。在下一篇文章将会看到，您可以简单调用诸如 `getAttribute("id")` 的方法来按名称获取属性的值。还可以用相似的方法调用来添加属性或设置（重置）现有属性的值。

值得指出的是，属性名的惟一性使得该列表不同于子对象列表。`p` 元素可以有多个 `em` 元素，所以子对象列表可以包含多个重复项。尽管子项列表和属性列表的操作方式相似，但一个可以包含重复项（对象的子项），而一个不能（元素对象的属性）。最后，只有元素具有属性，所以文本对象没有用于存储属性的附加列表。

凌乱的 HTML

在继续之前，谈到浏览器如何将标记转换为树表示，还有一个主题值得探讨，即浏览器如何处理不是格式良好的标记。*格式良好* 是 XML 广泛使用的一个术语，有两个基本意思：

- 每个开始标记都有一个与之匹配的结束标记。所以每个 `<p>` 在文档中与 `</p>` 匹配，每个 `<div>` 与 `</div>` 匹配，等等。
- 最里面的开始标记与最里面的结束标记相匹配，然后次里面的开始标记与次里面的结束标记相匹配，依此类推。所以 `<b><i>bold and italics</b></i>` 是不合法的，因为最里面的开始标记 `<i>`

> 与最里面的结束标记 <b> 匹配不当。要使之格式良好，*要么* 切换开始标记顺序，*要么* 切换结束标记顺序。（如果两者都切换，则仍会出现问题）。

深入研究这两条规则。这两条规则不仅简化了文档的组织，还消除了不定性。是否应先应用粗体后应用斜体？或恰恰相反？如果觉得这种顺序和不定性不是大问题，那么请记住，CSS 允许规则覆盖其他规则，所以，例如，如果 b 元素中文本的字体不同于 i 元素中的字体，则格式的应用顺序将变得非常重要。因此，HTML 的格式良好性有着举足轻重的作用。

如果浏览器收到了不是格式良好的文档，它只会尽力而为。得到的树结构在最好情况下将是作者希望的原始页面的近似，最坏情况下将面目全非。如果您曾将页面加载到浏览器中后看到完全出乎意料的结果，您可能在看到浏览器结果时会猜想您的结构应该如何，并沮丧地继续工作。当然，搞定这个问题相当简单：确保文档是格式良好的！如果不清楚如何编写标准化的 HTML，请咨询 参考资料 获得帮助。

DOM 简介

到目前为止，您已经知道浏览器将 Web 页面转换为对象表示，可能您甚至会猜想，对象表示是 DOM 树。DOM 表示 Document Object Model，是一个规范，可从 World Wide Web Consortium (W3C) 获得（您可以参阅 参考资料 中的一些 DOM 相关链接）。

但更重要的是，DOM 定义了对象的类型和属性，从而允许浏览器表示标记。（本系列下一篇文章将专门讲述在 JavaScript 和 Ajax 代码中使用 DOM 的规范。）

文档对象

首先，需要访问对象模型本身。这非常容易；要在运行于 Web 页面上的任何 JavaScript 代码中使用内置 document 变量，可以编写如下代码：

```
var domTree = document;
```

当然，该代码本身没什么用，但它演示了每个 Web 浏览器使得 document 对象可用于 JavaScript 代码，并演示了对象表示标记的完整树（图 1）。

每项都是一个节点

显然，document 对象很重要，但这只是开始。在进一步深入之前，需要学习另一个术语：*节点*。您已经知道标记的每个部分都由一个对象表示，但它不只是一个任意的对象，它是特定类型的对象，一个 DOM 节点。更特定的类型，比如文本、元素和属性，都继承自这个基本的节点类型。所以可以有文本节点、元素节点和属性节点。

如果已经有很多 JavaScript 编程经验，那您可能已经在使用 DOM 代码了。如果到目前为止您一直在跟踪本 Ajax 系列，那么现在您*一定* 使用 DOM 代码有一段时间了。例如，代码行 `var number = document.getElementById("phone").value;` 使用 DOM 查找特定元素，然后检索该元素的值（在本例中是一个表单字段）。所以即使您没有意识到这一点，但您每次将 document 键入 JavaScript 代码时都会使用 DOM。

详细解释已经学过的术语，DOM 树是对象的树，但更具体地说，它是*节点* 对象的树。在 Ajax 应用程序中或任何其他 JavaScript 中，可以使用这些节点产生下列效果，比如移除元素及其内容，突出显示特定文本，或添加新图像元素。因为都发生在客户端（运行在 Web 浏览器中的代码），所以这些效果立即发生，而不与服务器通信。最终结果通常是应用程序感觉起来响应更快，因为当请求转向服务器时以及解释响应时，Web 页面上的内容更改不会出现长时间的停顿。

在多数编程语言中，需要学习每种节点类型的实际对象名称，学习可用的属性，并弄清楚类型和强制转换；但在 JavaScript 中这都不是必需的。您可以只创建一个变量，并为它分配您希望的对象（正如您已经看到的）：

```
var domTree = document;
var phoneNumberElement = document.getElementById("phone");
var phoneNumber = phoneNumberElement.value;
```

没有类型，JavaScript 根据需要创建变量并为其分配正确的类型。结果，从 JavaScript 中使用 DOM 变得微不足道（将来有一篇文章会专门讲述与 XML 相关的 DOM，那时将更加巧妙）。

结束语

在这里，我要给您留一点悬念。显然，这并非是对 DOM 完全详尽的说明；事实上，本文不过是 DOM 的简介。DOM 的内容要远远多于我今天介绍的这些！

本系列的下一篇文章将扩展这些观点，并深入探讨如何在 JavaScript 中使用 DOM 来更新 Web 页面、快速更改 HTML 并为您的用户创建更交互的体验。在后面专门讲述在 Ajax 请求中使用 XML 的文章中，我将再次返回来讨论 DOM。所以要熟悉 DOM，它是 Ajax 应用程序的一个主要部分。

此时，深入了解 DOM 将十分简单，比如详细设计如何在 DOM 树中移动、获得元素和文本的值、遍历节点列表，等等，但这可能会让您有这种印象，即 DOM 是关于代码的，而事实上并非如此。

在阅读下一篇文章之前，试着思考一下树结构并用一些您自己的 HTML 实践一下，以查看 Web 浏览器是如何将 HTML 转换为标记的树视图的。此外，思考一下 DOM 树的组织，并用本文介绍的特殊情况实践一下：属性、有元素混合在其中的文本、*没有* 文本内容的元素（比如 `img` 元素）。

如果扎实掌握了这些概念，然后学习了 JavaScript 和 DOM 的语法（下一篇文章），则会使得响应更为容易。

而且不要忘了，这里有清单 2 和 3 的答案，其中还包含了示例代码！

图 2. 清单 2 的答案

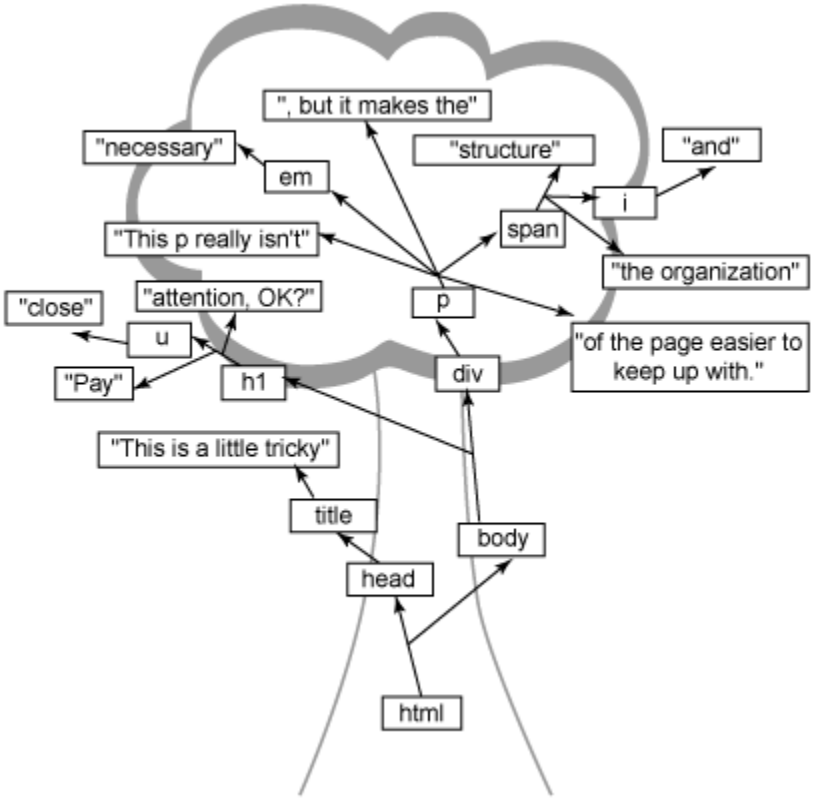
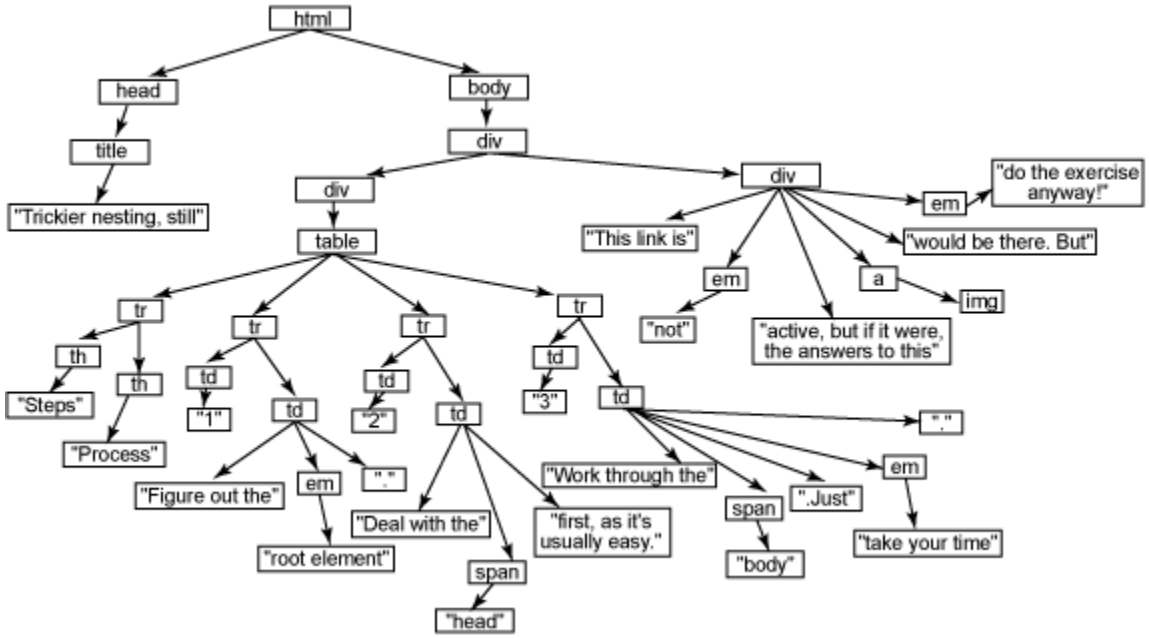


图 3. 清单 3 的答案


html
head
body
title
div
"Trickier nesting, still"
div
div
table
"This link is"
em
"do the exercise anyway!"
"would be there. But"
a
img
em
"not"
"active, but if it were, the answers to this"
tr
tr
tr
tr
th
th
td
td
td
td
td
"Steps"
"Process"
"1"
"2"
"3"
"."
"Figure out the"
em
"."
"Deal with the"
"first, as it's usually easy."
"Work through the"
span
".Just"
em
"root element"
"body"
"take your time"
span
"head"

第 5 部分: 操纵 DOM

上一期中 Brett 介绍了文档对象模型（DOM），它的元素在幕后定义了 Web 页面。这一期文章中他将进一步探讨 DOM。了解如何创建、删除和修改 DOM 树的各个部分，了解如何实现网页的即时更新！

如果阅读过本系列的 上一期文章，那么您就非常清楚当 Web 浏览器显示网页时幕后发生的一切了。前面已经提到，当 HTML 或为页面定义的 CSS 发送给 Web 浏览器时，网页被从文本转化成对象模型。无论代码简单或复杂，集中到一个文件还是分散到多个文件，都是如此。然后浏览器直接使用对象模型而不是您提供的文本文件。浏览器使用的模型称为*文档对象模型（Document Object Model，DOM）*。它连接表示文档中元素、属性和文本的对象。HTML 和 CSS 中所有的样式、值、甚至大部分空格都合并到该对象模型中。给定网页的具体模型称为该页面的 *DOM 树*。

了解什么是 DOM 树，以及知道它如何表示 HTML 和 CSS 仅仅是控制 Web 页面的第一步。接下来还需要了解如何处理 Web 页面的 DOM 树。比方说，如果向 DOM 树中增加一个元素，这个元素就会立即出现在用户的 Web 浏览器中 —— 不需要重新加载页面。从 DOM 树中删除一些文本，那些文本就会从用户屏幕上消失。可以通过 DOM 修改用户界面或者与用户界面交互，这样就提供了很强的编程能力和灵活性。一旦学会了如何处理 DOM 树，您就向实现丰富的、交互式动态网站迈出了一大步。

注意，下面的讨论以上一期文章“利用 DOM 进行 Web 响应”为基础，如果没有阅读过那一期，请在继续阅读之前首先阅读上一期文章。

跨浏览器、跨语言

> **首字母缩写的拼读问题**
> 从很多方面来说，文档对象模型应该被称为文档节点模型（Document Node Model，DNM）。当然，大多数人不知道*节点*一词的含义，而且“DNM”也不像“DOM”那么容易拼读，所以很容易理解 W3C 为何选择了 DOM。

文档对象模型是一种 W3C 标准（链接参见 参考资料）。因此，所有现代 Web 浏览器都支持 DOM —— 至少在一定程度上支持。虽然不同的浏览器有一些区别，但如果使用 DOM 核心功能并注意少数特殊情况和例外，DOM 代码就能以同样的方式用于任何浏览器。修改 Opera 网页的代码同样能用于 Apple's Safari®、Firefox®、Microsoft® Internet Explorer® 和 Mozilla®。

DOM 也是一种*跨语言*的规范，换句话说，大多数主流编程语言都能使用它。W3C 为 DOM 定义了几种*语言绑定*。一种语言绑定就是为特定语言定义的让您使用 DOM 的 API。比如，可以使用为 C、Java 和 JavaScript 定义的 DOM 语言绑定。因此可以从这些语言中使用 DOM。还有几种用于其他语言的语言绑定，尽管很多是由 W3C 以外的第三方定义的。

本系列文章主要讨论 JavaScript 的 DOM 绑定。这是因为多数异步应用程序开发都需要编写在 Web 浏览器中运行的 JavaScript 代码。使用 JavaScript 和 DOM 可以即时修改用户界面、响应用户事件和输入等等 —— 使用的完全是标准的 JavaScript。

总之，建议您也尝试一下其他语言中的 DOM 绑定。比如，使用 Java 语言绑定不仅能处理 HTML 还可处理 XML，这些内容将在以后的文章中讨论。因此本文介绍的技术还可用于 HTML 之外的其他语言，客户端 JavaScript 之外的其他环境。

回页首

节点的概念

节点是 DOM 中最基本的对象类型。实际上，您将在本文中看到，基本上 DOM 定义的其他所有对象都是节点对象的扩展。但是在深入分析语义之前，必须了解节点所代表的概念，然后再学习节点的具体属性和方法就非常简单了。

在 DOM 树中，基本上一切都是节点。每个元素在最底层上都是 DOM 树中的节点。每个属性都是节点。每段文本都是节点。甚至注释、特殊字符（如版权符号 ©）、DOCTYPE 声明（如果 HTML 或者 XHTML 中有的话）全都是节点。因此在讨论这些具体的类型之前必须清楚地把握什么是节点。

节点是.....

用最简单的话说，节点就是 DMO 树中的任何事物。之所以用“事物”这个模糊的字眼，是因为只能明确到这个程度。比如 HTML 中的元素（如 `img`）和 HTML 中的文本片段（如“Scroll down for more details”）没有多少明显的相似之处。但这是因为您考虑的可能是每种类型的*功能*，关注的是它们的不同点。

但是如果从另一个角度观察，DOM 树中的每个元素和每段文本都有一个*父亲*，这个父节点可能是另一个元素（比如嵌套在 `p` 元素中的 `img`）的孩子，或者 DOM 树中的顶层元素（这是每个文档中都出现一次的特殊情况，即使用 `html` 元素的地方）。另外，元素和文本都有一个*类型*。显然，元素的类型就是元素，文本的类型就是文本。每个节点还有某种定义明确的结构：下面还有节点（如子元素）吗？有*兄弟节点*（与元素或文本“相邻的”节点）吗？每个节点属于哪个文档？

显然，大部分内容听起来很抽象。实际上，说一个元素的类型是元素似乎有点冒傻气。但是要真正认识到将节点作为通用对象类型的价值，必须抽象一点来思考。

通用节点类型

DOM 代码中最常用的任务就是在页面的 DOM 树中导航。比方说，可以通过其“id”属性定位一个 `form`，然后开始处理那个 `form` 中内嵌的元素和文本。其中可能包含文字说明、输入字段的标签、真正的 `input` 元素，以及其他 HTML 元素（如 `img`）和链接（`a` 元素）。如果元素和文本是完全不同的*类型*，就必须为每种类型编写完全不同的代码。

如果使用一种通用节点类型情况就不同了。这时候只需要从一个节点移动到另一个节点，只有当需要对元素或文本作某种特殊处理时才需要考虑节点的*类型*。如果仅仅在 DOM 树中移动，就可以与其他节点类型一样用同样的操作移动到元素的父节点或者子节点。只有当需要某种节点类型的特殊性质时，如元素的属性，才需要对节点类型作专门处理。将 DOM 树中的所有对象都看作节点可以简化操作。记住这一点之后，接下来我们将具体看看 DOM 节点构造应该提供什么，首先从属性和方法开始。

回页首

节点的属性

使用 DOM 节点时需要一些属性和方法，因此我们首先来讨论节点的属性和方法。DOM 节点的属性主要有：

- nodeName 报告节点的名称（详见下述）。
- nodeValue 提供节点的“值”（详见后述）。
- parentNode 返回节点的父节点。记住，每个元素、属性和文本都有一个父节点。
- childNodes 是节点的孩子节点列表。对于 HTML，该列表仅对元素有意义，文本节点和属性节点都没有孩子。
- firstChild 仅仅是 childNodes 列表中第一个节点的快捷方式。
- lastChild 是另一种快捷方式，表示 childNodes 列表中的最后一个节点。
- previousSibling 返回当前节点*之前* 的节点。换句话说，它返回当前节点的父节点的 childNodes 列表中位于该节点前面的那个节点（如果感到迷惑，重新读前面一句）。
- nextSibling 类似于 previousSibling 属性，返回父节点的 childNodes 列表中的下一个节点。
- attributes 仅用于元素节点，返回元素的属性列表。

其他少数几种属性实际上仅用于更一般的 XML 文档，在处理基于 HTML 的网页时没有多少用处。

不常用的属性

上述大部分属性的意义都很明确，除了 nodeName 和 nodeValue 属性以外。我们不是简单地解释这两个属性，而是提出两个奇怪的问题：*文本节点的 nodeName 应该是什么*？类似地，*元素的 nodeValue 应该是什么*？

如果这些问题难住了您，那么您就已经了解了这些属性固有的含糊性。nodeName 和 nodeValue 实际上并非适用于*所有* 节点类型（节点的其他少数几个属性也是如此）。这就说明了一个重要概念：任何这些属性都可能返回空值（有时候在 JavaScript 中称为“未定义”）。比方说，文本节点的 nodeName 属性是空值（或者在一些浏览器中称为“未定义”），因为文本节点没有名称。如您所料，nodeValue 返回节点的文本。类似地，元素有 nodeName，即元素名，但元素的 nodeValue 属性值总是空。属性同时具有 nodeName 和 nodeValue。下一节我还将讨论这些单独的类型，但是因为这些属性是每个节点的一部分，因此在这里有必要提一提。

现在看看 清单 1，它用到了一些节点属性。

清单 1. 使用 DOM 中的节点属性

```
// These first two lines get the DOM tree for the current Web page,
//  and then the <html> element for that DOM tree
var myDocument = document;
var htmlElement = myDocument.documentElement;
// What's the name of the <html> element? "html"
alert("The root element of the page is " + htmlElement.nodeName);
// Look for the <head> element
var headElement = htmlElement.getElementsByTagName("head")[0];
if (headElement != null) {
```

```
    alert("We found the head element, named " + headElement.nodeName);
    // Print out the title of the page
    var titleElement = headElement.getElementsByTagName("title")[0];
    if (titleElement != null) {
      // The text will be the first child node of the <title> element
      var titleText = titleElement.firstChild;
      // We can get the text of the text node with nodeValue
      alert("The page title is '" + titleText.nodeValue + "'");
    }
    // After <head> is <body>
    var bodyElement = headElement.nextSibling;
    while (bodyElement.nodeName.toLowerCase() != "body") {
      bodyElement = bodyElement.nextSibling;
    }
    // We found the <body> element...
    // We'll do more when we know some methods on the nodes.
  }
```

回页首

节点方法

接下来看看所有节点都具有的方法（与节点属性一样，我省略了实际上不适用于多数 HTML DOM 操作的少数方法）：

- `insertBefore(newChild, referenceNode)` 将 `newChild` 节点插入到 `referenceNode` 之前。记住，应该对 `newChild` 的目标父节点调用该方法。
- `replaceChild(newChild, oldChild)` 用 `newChild` 节点替换 `oldChild` 节点。
- `removeChild(oldChild)` 从运行该方法的节点中删除 `oldChild` 节点。
- `appendChild(newChild)` 将 `newChild` 添加到运行该函数的节点之中。`newChild` 被添加到目标节点孩子列表中的*末端*。
- `hasChildNodes()` 在调用该方法的节点有孩子时则返回 true，否则返回 false。
- `hasAttributes()` 在调用该方法的节点有属性时则返回 true，否则返回 false。

注意，大部分情况下所有这些方法处理的都是节点的孩子。这是它们的主要用途。如果仅仅想获取文本节

点值或者元素名，则不需要调用这些方法，使用节点属性就可以了。清单 2 在 清单 1 的基础上增加了方法使用。

清单 2. 使用 DOM 中的节点方法

```
// These first two lines get the DOM tree for the current Web page,
    //   and then the <html> element for that DOM tree
    var myDocument = document;
    var htmlElement = myDocument.documentElement;
    // What's the name of the <html> element? "html"
    alert("The root element of the page is " + htmlElement.nodeName);
    // Look for the <head> element
    var headElement = htmlElement.getElementsByTagName("head")[0];
    if (headElement != null) {
      alert("We found the head element, named " + headElement.nodeName);
      // Print out the title of the page
      var titleElement = headElement.getElementsByTagName("title")[0];
      if (titleElement != null) {
        // The text will be the first child node of the <title> element
        var titleText = titleElement.firstChild;
        // We can get the text of the text node with nodeValue
        alert("The page title is '" + titleText.nodeValue + "'");
      }
      // After <head> is <body>
      var bodyElement = headElement.nextSibling;
      while (bodyElement.nodeName.toLowerCase() != "body") {
        bodyElement = bodyElement.nextSibling;
      }
      // We found the <body> element...
      // Remove all the top-level <img> elements in the body
      if (bodyElement.hasChildNodes()) {
        for (i=0; i<bodyElement.childNodes.length; i++) {
          var currentNode = bodyElement.childNodes[i];
          if (currentNode.nodeName.toLowerCase() == "img") {
            bodyElement.removeChild(currentNode);
          }
        }
      }
    }
```

测试一下！

目前虽然只看到了两个例子，清单 1 和 2，不过通过这两个例子您应该能够了解使用 DOM 树能够做什么。如果要尝试一下这些代码，只需要将 清单 3 拖入一个 HTML 文件并保存，然后用 Web 浏览器打

开。

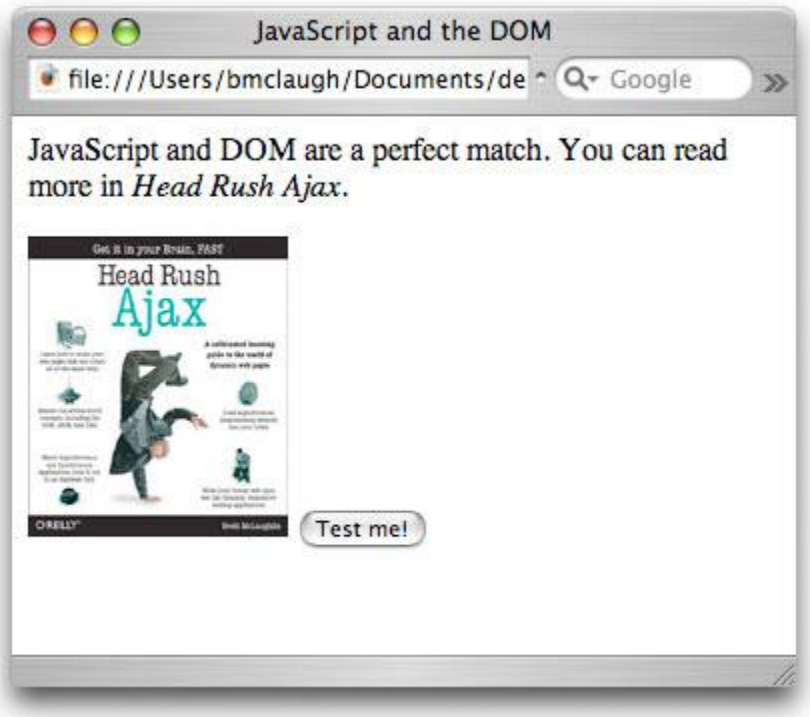
清单 3. 包含使用 DOM 的 JavaScript 代码的 HTML 文件

```
<html>
 <head>
  <title>JavaScript and the DOM</title>
  <script language="JavaScript">
   function test() {
    // These first two lines get the DOM tree for the current Web page,
    //  and then the <html> element for that DOM tree
    var myDocument = document;
    var htmlElement = myDocument.documentElement;
    // What's the name of the <html> element? "html"
    alert("The root element of the page is " + htmlElement.nodeName);
    // Look for the <head> element
    var headElement = htmlElement.getElementsByTagName("head")[0];
    if (headElement != null) {
      alert("We found the head element, named " + headElement.nodeName);
      // Print out the title of the page
      var titleElement = headElement.getElementsByTagName("title")[0];
      if (titleElement != null) {
        // The text will be the first child node of the <title> element
        var titleText = titleElement.firstChild;
        // We can get the text of the text node with nodeValue
        alert("The page title is '" + titleText.nodeValue + "'");
      }
      // After <head> is <body>
      var bodyElement = headElement.nextSibling;
      while (bodyElement.nodeName.toLowerCase() != "body") {
        bodyElement = bodyElement.nextSibling;
      }
      // We found the <body> element...
      // Remove all the top-level <img> elements in the body
      if (bodyElement.hasChildNodes()) {
        for (i=0; i<bodyElement.childNodes.length; i++) {
          var currentNode = bodyElement.childNodes[i];
          if (currentNode.nodeName.toLowerCase() == "img") {
            bodyElement.removeChild(currentNode);
          }
        }
      }
    }
```

```
 }
 </script>
</head>
<body>
 <p>JavaScript and DOM are a perfect match.
    You can read more in <i>Head Rush Ajax</i>.</p>
 <img src="http://www.headfirstlabs.com/Images/hraj_cover-150.jpg" />
 <input type="button" value="Test me!" onClick="test();" />
</body>
</html>
```

将该页面加载到浏览器后，可以看到类似 图 1 所示的画面。

图 1. 用按钮运行 JavaScript 的 HTML 页面

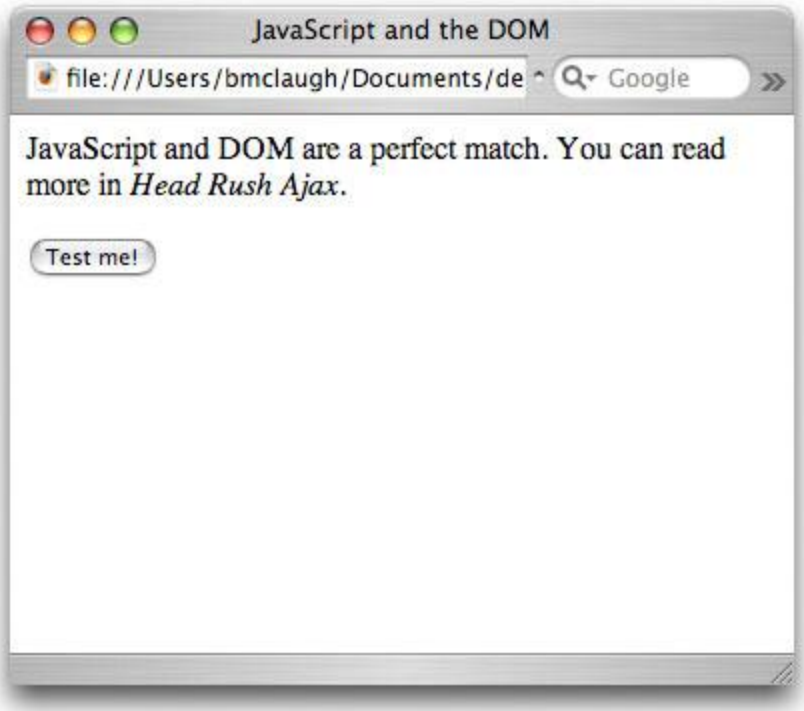


单击 **Test me!** 将看到 图 2 所示的警告框。

图 2. 使用 nodeValue 显示元素名的警告框

代码运行完成后，图片将从页面中实时删除，如 图 3 所示。

图 3. 使用 JavaScript 实时删除图像



回页首

API 设计问题

再看一看各种节点提供的属性和方法。对于那些熟悉面向对象（OO）编程的人来说，它们说明了 DOM 的一个重要特点：DOM 并非完全面向对象的 API。首先，很多情况下要直接使用对象的属性而不是调用节点对象的方法。比方说，没有 `getNodeName()` 方法，而要直接使用 `nodeName` 属性。因此节点对象（以及其他 DOM 对象）通过属性而不是函数公开了大量数据。

其次，如果习惯于使用重载对象和面向对象的 API，特别是 Java 和 C++ 这样的语言，就会发现 DOM 中的对象和方法命名有点奇怪。DOM 必须能用于 C、Java 和 JavaScript（这只是其中的几种语言），因此 API 设计作了一些折衷。比如，`NamedNodeMap` 方法有两种不同的形式：

- `getNamedItem(String name)`
- `getNamedItemNS(Node node)`

对于 OO 程序员来说这看起来非常奇怪。两个方法目的相同，只不过一个使用 String 参数而另一个使用 `Node` 参数。多数 OO API 中对这两种版本都会使用相同的方法名。运行代码的虚拟机将根据传递给方法的对象类型决定运行哪个方法。

问题在于 JavaScript 不支持这种称为*方法重载*的技术。换句话说，JavaScript 要求每个方法或函数使用不同的名称。因此，如果有了一个名为 `getNamedItem()` 的接受字符串参数的方法，就不能再有另一个方法或函数也命名为 `getNamedItem()`，即使这个方法的参数类型不同（或者完全不同的一组参数）。如果这样做，JavaScript 将报告错误，代码不会按照预期的方式执行。

从根本上说，DOM 有意识地避开了方法重载和其他 OO 编程技术。这是为了保证该 API 能够用于多种语言，包括那些不支持 OO 编程技术的语言。后果不过是要求您多记住一些方法名而已。好处是可以在任何语言中学习 DOM，比如 Java，并清楚同样的方法名和编码结构也能用于具有 DOM 实现的其他语言，如 JavaScript。

让程序员小心谨慎

如果深入研究 API 设计或者仅仅非常关注 API 设计，您可能会问：“为何节点类型的属性不能适用于所有节点？”这是一个很好的问题，答案和政治以及决策制定而非技术原因关系更密切。简单地说，答案就是，“谁知道！但有点令人恼火，不是吗？”

属性 `nodeName` 意味着允许每种类型的节点都有一个名字，但是很多情况下名字要么未定义，要么是对于程序员没有意义的内部名（比如在 Java 中，很多情况下文本节点的 `nodeName` 被报告为“#text”）。从根本上说，必须假设您得自己来处理错误。直接访问 `myNode.nodeName` 然后使用该值是危险的，很多情况下这个值为空。因此与通常的编程一样，程序员要谨慎从事。

回页首

通用节点类型

现在已经介绍了 DOM 节点的一些特性和属性（以及一些奇特的地方），下面开始讲述您将用到的一些特殊节点类型。多数 Web 应用程序中只用到四种节点类型：

- **文档节点**表示整个 HTML 文档。
- **元素节点**表示 HTML 元素，如 `a` 或 `img`。
- **属性节点**表示 HTML 元素的属性，如 `href`（`a` 元素）或 `src`（`img` 元素）。
- **文本节点**表示 HTML 文档中的文本，如 “Click on the link below for a complete set list”。这是出现在 `p`、`a` 或 `h2` 这些元素中的文字。

处理 HTML 时，95% 的时间是跟这些节点类型打交道。因此本文的其余部分将详细讨论这些节点。（将来讨论 XML 的时候将介绍其他一些节点类型。）

回页首

文档节点

基本上所有基于 DOM 的代码中都要用到的第一个节点类型是文档节点。*文档节点* 实际上并不是 HTML（或 XML）页面中的一个元素而是页面本身。因此在 HTML Web 页面中，文档节点就是整个 DOM 树。在 JavaScript 中，可以使用关键字 `document` 访问文档节点：

```
// These first two lines get the DOM tree for the current Web page,
//   and then the <html> element for that DOM tree
var myDocument = document;
var htmlElement = myDocument.documentElement;
```

JavaScript 中的 `document` 关键字返回当前网页的 DOM 树。从这里可以开始处理树中的所有节点。也可使用 `document` 对象创建新节点，如下所示：

- `createElement(elementName)` 使用给定的名称创建一个元素。
- `createTextNode(text)` 使用提供的文本创建一个新的文本节点。
- `createAttribute(attributeName)` 用提供的名称创建一个新属性。

这里的关键在于这些方法创建节点，但是并没有将其附加或者插入到特定的文档中。因此，必须使用前面所述的方法如 insertBefore() 或 appendChild() 来完成这一步。因此，可使用下面的代码创建新元素并将其添加到文档中：

```
var pElement = myDocument.createElement("p");
var text = myDocument.createTextNode("Here's some text in a p element.");
pElement.appendChild(text);
bodyElement.appendChild(pElement);
```

一旦使用 document 元素获得对 Web 页面 DOM 树的访问，就可以直接使用元素、属性和文本了。

回页首

元素节点

虽然会大量使用元素节点，但很多需要对元素执行的操作都是所有节点共有的方法和属性，而不是元素特有的方法和属性。元素只有两组专有的方法：

1. **与属性处理有关的方法：**
    - getAttribute(name) 返回名为 name 的属性值。
    - removeAttribute(name) 删除名为 name 的属性。
    - setAttribute(name, value) 创建一个名为 name 的属性并将其值设为 value。
    - getAttributeNode(name) 返回名为 name 的属性节点（属性节点在 下一节 介绍）。
    - removeAttributeNode(node) 删除与指定节点匹配的属性节点。
2. **与查找嵌套元素有关的方法：**
    - getElementsByTagName(elementName) 返回具有指定名称的元素节点列表。

这些方法意义都很清楚，但还是来看几个例子吧。

处理属性

处理元素很简单，比如可用 document 对象和上述方法创建一个新的 img 元素：

```
var imgElement = document.createElement("img");
imgElement.setAttribute("src",
"http://www.headfirstlabs.com/Images/hraj_cover-150.jpg");
imgElement.setAttribute("width", "130");
imgElement.setAttribute("height", "150");
bodyElement.appendChild(imgElement);
```

现在看起来应该非常简单了。实际上，只要理解了节点的概念并知道有哪些方法可用，就会发现在 Web 页面和 JavaScript 代码中处理 DOM 非常简单。在上述代码中，JavaScript 创建了一个新的 img 元素，设置了一些属性然后添加到 HTML 页面的 body 元素中。

查找嵌套元素

发现嵌套的元素很容易。比如，下面的代码用于发现和删除 清单 3 所示 HTML 页面中的所有 img 元素：

```
    // Remove all the top-level <img> elements in the body
    if (bodyElement.hasChildNodes()) {
      for (i=0; i<bodyElement.childNodes.length; i++) {
        var currentNode = bodyElement.childNodes[i];
        if (currentNode.nodeName.toLowerCase() == "img") {
          bodyElement.removeChild(currentNode);
        }
      }
    }
```

也可以使用 getElementsByTagName() 完成类似的功能：

```
// Remove all the top-level <img> elements in the body
    var imgElements = bodyElement.getElementsByTagName("img");
    for (i=0; i<imgElements.length; i++) {
      var imgElement = imgElements.item[i];
      bodyElement.removeChild(imgElement);
    }
```

回页首

属性节点

DOM 将属性表示成节点，可以通过元素的 attributes 来访问元素的属性，如下所示：

```
// Remove all the top-level <img> elements in the body
    var imgElements = bodyElement.getElementsByTagName("img");
    for (i=0; i<imgElements.length; i++) {
      var imgElement = imgElements.item[i];
```

```
      // Print out some information about this element
      var msg = "Found an img element!";
      var atts = imgElement.attributes;
      for (j=0; j<atts.length; j++) {
        var att = atts.item(j);
        msg = msg + "\n  " + att.nodeName + ": '" + att.nodeValue + "'";
      }
      alert(msg);
      bodyElement.removeChild(imgElement);
    }
```

需要指出的是，attributes 属性实际上是对节点类型而非局限于元素类型来说的。有点古怪，不影响您编写代码，但是仍然有必要知道这一点。
虽然也能使用属性节点，但通常使用元素类的方法处理属性更简单。其中包括：

- getAttribute(name) 返回名为 name 的属性值。
- removeAttribute(name) 删除名为 name 的属性。
- setAttribute(name, value) 创建一个名为 name 的属性并将其值设为 value。

属性的奇特之处
对于 DOM 来说属性有一些特殊的地方。一方面，属性实际上并不像其他元素或文本那样是元素的孩子，换句话说，属性并不出现在元素“之下”。同时，属性显然和元素有一定的关系，元素“拥有”属性。DOM 使用节点表示属性，并允许通过元素的专门列表来访问属性。因此属性是 DOM 树的一部分，但通常不出现在树中。有理由说，属性和 DOM 树结构其他部分之间的关系有点模糊。

这三个方法不需要直接处理属性节点。但允许使用简单的字符串属性设置和删除属性及其值。

回页首

文本节点

需要考虑的最后一种节点是文本节点（至少在处理 HTML DOM 树的时候如此）。基本上通常用于处理文本节点的所有属性都属于节点对象。实际上，一般使用 nodeValue 属性来访问文本节点的文本，如下所示：

```
var pElements = bodyElement.getElementsByTagName("p");
for (i=0; i<pElements.length; i++) {
  var pElement = pElements.item(i);
```

```
  var text = pElement.firstChild.nodeValue;
  alert(text);
}
```

少数其他几种方法是专门用于文本节点的。这些方法用于增加或分解节点中的数据：

- `appendData(text)` 将提供的文本追加到文本节点的已有内容之后。
- `insertData(position, text)` 允许在文本节点的中间插入数据。在指定的位置插入提供的文本。
- `replaceData(position, length, text)` 从指定位置开始删除指定长度的字符，用提供的文本代替删除的文本。

回页首

什么节点类型？

到目前为止看到的多数代码都假设已经知道处理的节点是什么类型，但情况并非总是如此。比方说，如果在 DOM 树中导航并处理一般的节点类型，可能就不知道您遇到了元素还是文本。也许获得了 `p` 元素的所有孩子，但是不能确定处理的是文本、`b` 元素还是 `img` 元素。这种情况下，在进一步的处理之前需要确定是什么类型的节点。
所幸的是很容易就能做到。DOM 节点类型定义了一些常量，比如：

1. `Node.ELEMENT_NODE` 是表示元素节点类型的常量。
2. `Node.ATTRIBUTE_NODE` 是表示属性节点类型的常量。
3. `Node.TEXT_NODE` 是表示文本节点类型的常量。
4. `Node.DOCUMENT_NODE` 是表示文档节点类型的常量。

还有其他一些节点类型，但是对于 HTML 除了这四种以外很少用到。我有意没有给出这些常量的值，虽然 DOM 规范中定义了这些值，永远不要直接使用那些值，因为这正是常量的目的！

nodeType 属性

可使用 `nodeType` 属性比较节点和上述常量 —— 该属性定义在 DOM node 类型上因此可用于所有节点，如下所示：

```
var someNode = document.documentElement.firstChild;
if (someNode.nodeType == Node.ELEMENT_NODE) {
  alert("We've found an element node named " + someNode.nodeName);
} else if (someNode.nodeType == Node.TEXT_NODE) {
```

```
  alert("It's a text node; the text is " + someNode.nodeValue);
} else if (someNode.nodeType == Node.ATTRIBUTE_NODE) {
  alert("It's an attribute named " + someNode.nodeName
                    + " with a value of '" + someNode.nodeValue + "'");
}
```

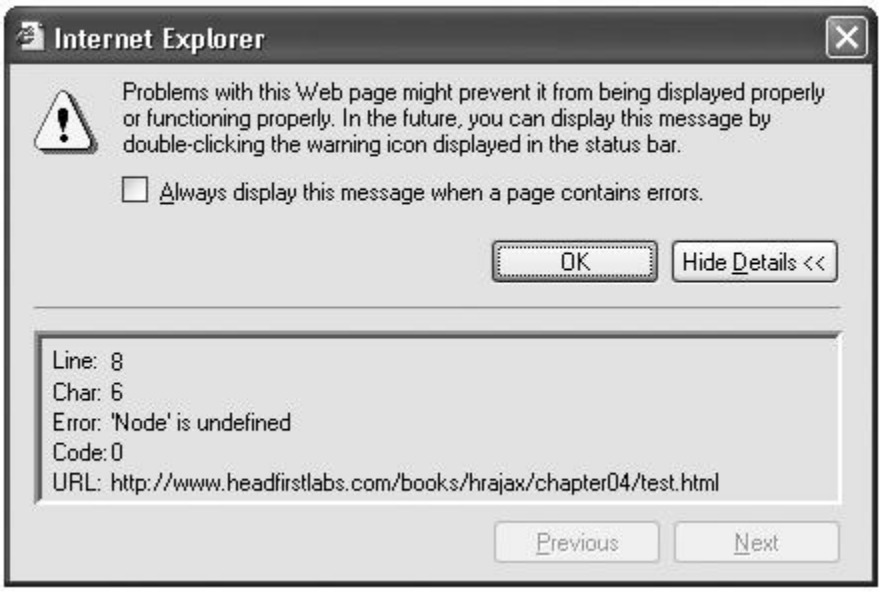
这个例子非常简单，但说明了一个大问题：得到节点的类型*非常* 简单。更有挑战性的是知道节点的类型之后确定能做什么，只要掌握了节点、文本、属性和元素类型提供了什么属性和方法，就可以自己进行 DOM 编程了。

好了，快结束了。

实践中的挫折

`nodeType` 属性似乎是使用节点的一个入场卷 —— 允许确定要处理的节点类型然后编写处理该节点的代码。问题在于上述 `Node` 常量定义不能正确地用于 Internet Explorer。因此如果在代码中使用 `Node.ELEMENT_NODE`、`Node.TEXT_NODE` 或其他任何常量，Internet Explorer 都将返回如 图 4 所示的错误。

图 4. Internet Explorer 报告错误



任何时候在 JavaScript 中使用 `Node` 常量，Internet Explorer 都会报错。因为多数人仍然在使用 Internet Explorer，应该避免在代码中使用 `Node.ELEMENT_NODE` 或 `Node.TEXT_NODE` 这样的构造。尽管据说即将发布的新版本 Internet Explorer 7.0 将解决这个问题，但是在 Internet Explorer 6.x 退出舞台之前仍然要很多年。因此应避免使用 `Node`，要想让您的 DOM 代码（和 Ajax 应用程序）能用于所有主要浏览器，这一点很重要。

回页首

结束语

在本系列的上几期文章中您已经学习了很多。现在，您*不*应该再坐等下一篇文章期待我介绍各种聪明的 DOM 树用法。现在的家庭作业是看看如何使用 DOM 创造出富有想像力的效果或者漂亮的界面。利用近几期文章中所学的知识开始实验和练习。看看能否建立感觉更与桌面应用程序接近的网站，对象能够响应用户的动作在屏幕上移动。

最好在屏幕上为每个对象画一个边界，这样就能看到 DOM 树中的对象在何处，然后再移动对象。创建节点并将其添加到已有的孩子列表中，删除没有嵌套节点的空节点，改变节点的 CSS 样式，看看孩子节点是否会继承这些修改。可能性是无限的，每当尝试一些新东西时，就学到了一些新的知识。尽情地修改您的网页吧！

准备成为顶尖的网页设计师吗？

如果您准备了解甚至掌握 DOM，您就会成为最顶尖的 Web 编程人员。多数 Web 程序员知道如何使用 JavaScript 编写图像滚动或者从表单中提取值，有些甚至能够向服务器发送请求和接收响应（阅读本系列的前几篇文章之后您也能做到）。但胆小鬼或者没有经验的人不可能做到即时修改网页结构。

在 DOM 三部曲的最后一期文章中，我*将*介绍如何把一些非常棒的有趣的 DOM 应用结合到编程中。我将不再是从概念上说教和解释 API，而会提供一些代码。在此之前先发挥您自己的聪明才智，看看能做些什么。

第 6 部分：建立基于 DOM 的 Web 应用程序

本系列的上一篇文章中考察了文档对象模型（DOM）编程中涉及到的概念——Web 浏览器如何把网页看作一棵树，现在您应该理解了 DOM 中使用的编程结构。本期教程将把这些知识用于实践，建立一个简单的包含一些特殊效果的 Web 页面，所有这些都使用 JavaScript 操纵 DOM 来创建，不需要重新加载或者刷新页面。

前面两期文章已经详细介绍了文档对象模型或者 DOM，读者应该很清楚 DOM 是如何工作的了。（前两期 DOM 文章以及 Ajax 系列更早文章的链接请参阅参考资料。）本教程中将把这些知识用于实践。我们将开发一个简单的 Web 应用程序，其用户界面可根据用户动作改变，当然要使用 DOM 来处理界面的改变。阅读完本文之后，就已经把学习到的关于 DOM 的技术和概念付诸应用了。

假设读者已经阅读过上两期文章，如果还没有的话，请先看一看，切实掌握什么是 DOM 以及 Web 浏览器如何将提供给它的 HTML 和 CSS 转化成单个表示网页的树状结构。到目前为止我一直在讨论的所有 DOM 原理都将在本教程中用于创建一个能工作的（虽然有点简单）基于 DOM 的动态 Web 页面。如果遇到不懂的地方，可以随时停下来复习一下前面的两期文章然后再回来。

从一个示例应用程序开始

关于代码的说明

我们首先建立一个非常简单的应用程序，然后再添加一点 DOM 魔法。要记住，DOM 可以移动网页中的任何东西而不需要提交表单，因此足以和 Ajax 媲美；我们创建一个简单的网页，上面只显示一个普通的旧式大礼帽，还有一个标记为 **Hocus Pocus!** 的按钮（猜猜这是干什么的？）

> 为了把注意力集中到 DOM 和 JavaScript 代码上，我编写 HTML 的时候有些随意地采用内联样式（比如 `h1` 和 `p` 元素的 `align` 属性）。虽然对实验来说这样做是可接受的，但是对于开发的任何产品应用程序，我建议花点时间把所有的样式都放到外部 CSS 样式表中。

初始 HTML

清单 1 显示了这个网页的 HTML。除了标题和表单外，只有一个简单的图片和可以点击的按钮。

清单 1. 示例应用程序的 HTML

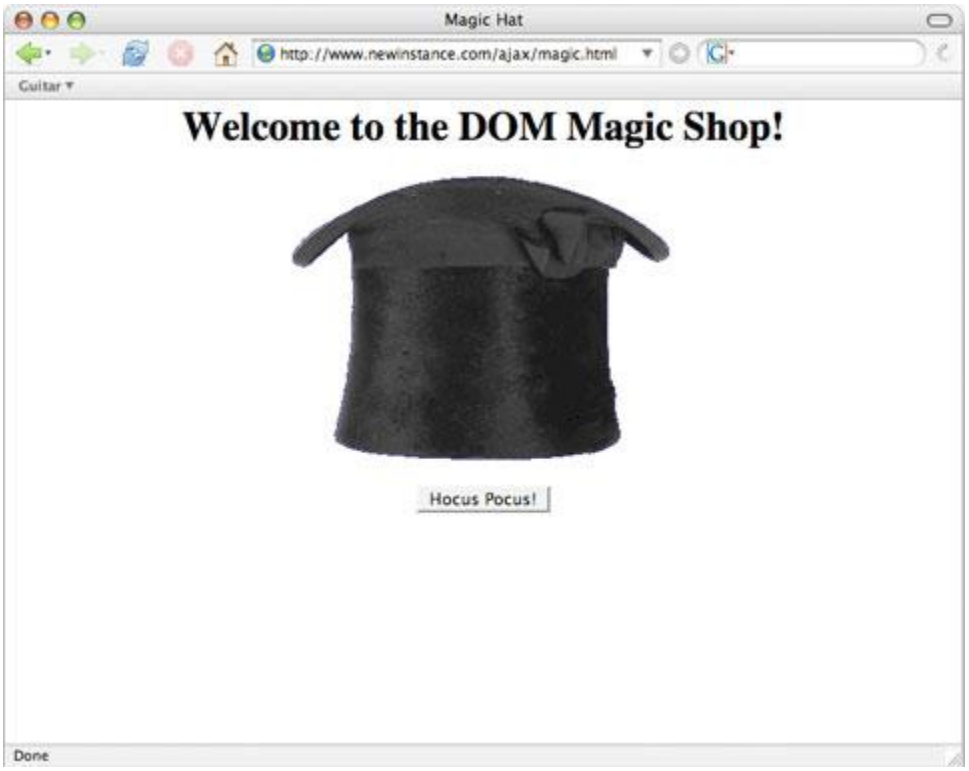
```
<html>
 <head>
  <title>Magic Hat</title>
 </head>

 <body>
  <h1 align="center">Welcome to the DOM Magic Shop!</h1>
  <form name="magic-hat">
   <p align="center">
    <img src="topHat.gif" />
    <br /><br />
    <input type="button" value="Hocus Pocus!" />
   </p>
  </form>
 </body>
</html>
```

可以在本文后面的下载中找到这段 HTML 和本文中用到的图片。不过我强烈建议您只下载那个图片，然后随着本文中逐渐建立这个应用程序自己动手输入代码。这样要比读读本文然后直接打开完成的应用程序能够更好地理解 DOM 代码。

查看示例网页

这里没有什么特别的窍门，打开网页可以看到图 1 所示的结果。

图 1. 难看的大礼帽



关于 HTML 的补充说明

应该 注意的重要一点是，清单 1 和图 1 中按钮的类型是 `button` 而不是提交按钮。如果使用提交按钮，单击该按钮将导致浏览器提交表单，当然表单没有 `action` 属性（完全是有意如此），从而会造成没有任何动作的无限循环。（应该自己试试，看看会发生什么。）通过使用一般输入按钮而不是提交按钮，可以把 JavaScript 函数和它连接起来与浏览器交互而*无需* 提交表单。

回页首

向示例应用程序添加元素

现在用一些 JavaScript、DOM 操作和小小的图片戏法装扮一下网页。

使用 getElementById() 函数

显然，魔法帽子没有兔子就没有看头了。这里首先用兔子的图片替换页面中原有的图片（再看看图 1），如图 2 所示。

图 2. 同样的礼帽，这一次有了兔子

完成这个 DOM 小戏法的第一步是找到网页中表示 `img` 元素的 DOM 节点。一般来说，最简单的办法是用 `getElementById()` 方法，它属于代表 Web 页面的 `document` 对象。前面已经见到过这个方法，用法如下：

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

为 HTML 添加 id 属性

这是非常简单的 JavaScript，但是需要修改一下 HTML：为需要访问的元素增加 `id` 属性。也就是希望（用带兔子的新图片）替换的 `img` 元素，因此将 HTML 改为清单 2 的形式。

清单 2. 增加 id 属性

```
<html>
 <head>
  <title>Magic Hat</title>
 </head>

 <body>
  <h1 align="center">Welcome to the DOM Magic Shop!</h1>
  <form name="magic-hat">
   <p align="center">
    <img src="topHat.gif" id="topHat" />
```

```
    <br /><br />
    <input type="button" value="Hocus Pocus!" />
   </p>
  </form>
 </body>
</html>
```

如果重新加载（或者打开）该页面，可以看到毫无变化，增加 `id` 属性对网页的外观没有影响。不过，该属性可以帮助 JavaScript 和 DOM 更方便地处理元素。

抓住 img 元素

现在可以很容易地使用 `getElementById()` 了。已经有了需要元素的 ID，即 `topHat`，可以将其保存在一个新的 JavaScript 变量中。在 HTML 页面中增加清单 3 所示的代码。

清单 3. 访问 img 元素

```
<html>
 <head>
  <title>Magic Hat</title>
  <script language="JavaScript">
    function showRabbit() {
      var hatImage = document.getElementById("topHat");
    }
  </script>
 </head>

 <body>
  <h1 align="center">Welcome to the DOM Magic Shop!</h1>
  <form name="magic-hat">
   <p align="center">
    <img src="topHat.gif" id="topHat" />
    <br /><br />
    <input type="button" value="Hocus Pocus!" />
   </p>
  </form>
 </body>
</html>
```

现在打开或重新加载该网页同样没有什么惊奇的地方。虽然现在能够访问图片，但是对它还什么也没有做。

回页首

修改图片，麻烦的办法

完成所需修改有两种方法：一种简单，一种麻烦。和所有的好程序员一样，我也喜欢简单的办法；但是运用较麻烦的办法是一次很好的 DOM 练习，值得您花点时间。首先看看换图片比较麻烦的办法；后面再重新分析一下看看有没有更简单的办法。
用带兔子的新照片替换原有图片的办法如下：

1. 创建新的 `img` 元素。
2. 访问当前 `img` 元素的父元素，也就是它的容器。
3. 在已有 `img` 元素*之前* 插入新的 `img` 元素作为该容器的子级。
4. 删除原来的 `img` 元素。
5. 结合起来以便在用户单击 **Hocus Pocus!** 按钮时调用刚刚创建的 JavaScript 函数。

创建新的 img 元素

通过上两期文章应该记住 DOM 中最关键的是 `document` 对象。它代表整个网页，提供了 `getElementById()` 这样功能强大的方法，还能够创建新的节点。现在要用到的就是这最后一种性质。
具体而言，需要创建一个新的 `img` 元素。要记住，在 DOM 中一切都是节点，但是节点被进一步划分为三种基本类型：

- 元素
- 属性
- 文本节点

还有其他类型，但是这三种可以满足 99% 的编程需要。这里需要一个 `img` 类型的新元素。因此需要下列 JavaScript 代码：

```
var newImage = document.createElement("img");
```

这行代码可以创建一个 `element` 类型的新节点，元素名为 `img`。在 HTML 中基本上就是：

```
<img />
```

要记住，DOM 会创建结构良好的 HTML，就是说这个目前为空的元素包括起始和结束标签。剩下的就是向该元素增加内容或属性，然后将其插入到网页中。
对内容来说，`img` 是一个空元素。但是需要增加一个属性 `src`，它指定了要加载的图片。您也许认为要使用 `addAttribute()` 之类的方法，但情况并非如此。DOM 规范的制定者认为程序员可能喜欢简洁（的确如此!），因此他们规定了一个方法同时用于增加新属性和改变已有的属性值：`setAttribute()`。
如果对已有的属性调用 `setAttribute()`，则把原来的值替换为指定的值。但是，如果调用 `setAttribute()` 并指定一个*不* 存在的属性，DOM 就会使用提供的值增加一个属性。一个方法，两种用途！因此需要增加下列 JavaScript 代码：

```
var newImage = document.createElement("img");
```

```
newImage.setAttribute("src", "rabbit-hat.gif");
```

它创建一个图片元素然后设置适当的资源属性。现在，HTML 应该如清单 4 所示。

清单 4. 使用 DOM 创建新图片

```
<html>
 <head>
  <title>Magic Hat</title>
  <script language="JavaScript">
    function showRabbit() {
      var hatImage = document.getElementById("topHat");
      var newImage = document.createElement("img");
      newImage.setAttribute("src", "rabbit-hat.gif");
    }
  </script>
 </head>

 <body>
  <h1 align="center">Welcome to the DOM Magic Shop!</h1>
  <form name="magic-hat">
   <p align="center">
    <img src="topHat.gif" id="topHat" />
    <br /><br />
    <input type="button" value="Hocus Pocus!" />
   </p>
  </form>
 </body>
</html>
```

可以加载该页面，但是不要期望有任何改变，因为目前所做的修改实际上还没有影响页面。另外，如果再看看任务列表中的第 5 步，就会发现还没有调用我们的 JavaScript 函数！

获得原始图片的父元素

现在有了要插入的图片，还需要找到插入的地方。但是不能将其插入到已有的图片中，而是要将其插入到已有图片之前然后再删除原来的图片。为此需要知道已有图片的父元素，实际上这就是插入和删除操作的真正关键所在。

应该记得，前面的文章中曾经指出 DOM 确实把网页看成一棵树，即节点的层次结构。每个节点都有父节点（树中更高层次的节点，该节点是它的一个子级），可能还有自己的子节点。对于图片来说，它没有子级——要记住图片是空元素，但是它肯定有父节点。甚至不需要知道父节点是什么，但是需要访问它。

为此，只要使用每个 DOM 节点都有的 `parentNode` 属性即可，比如：

```
var imgParent = hatImage.parentNode;
```

确实非常简单！可以肯定这个节点有子节点，因为已经有了一个：原来的图片。此外，完全不需要知道它

是一个 div、p 或者页面的 body，都没有关系！

插入新图片

现在得到了原来图片的父节点，可以插入新的图片了。很简单，有多种方法可以添加子节点：

- insertBefore(newNode, oldNode)
- appendChild(newNode)

因为希望把新图片放在旧图片的位置上，需要使用 insertBefore()（后面还要使用 removeChild() 方法）。可使用下面这行 JavaScript 代码把新图片元素插入到原有图片之前：

```
var imgParent = hatImage.parentNode;
imgParent.insertBefore(newImage, hatImage);
```

现在原图片的父元素有了*两个* 子元素：新图片和紧跟在后面的旧图片。必须指出，这里*包围* 这些图片的内容没有变，而且这些内容的顺序也和插入之前完全相同。仅仅是这个父节点中增加了一个子节点，即旧图片之前的新图片。

删除旧图片

现在只需要删除旧图片，因为网页中只需要新图片。很简单，因为已经得到了旧图片元素的父节点。只要调用 removeChild() 并把需要删除的节点传递给它即可：

```
var imgParent = hatImage.parentNode;
imgParent.insertBefore(newImage, hatImage);
imgParent.removeChild(hatImage);
```

现在，用新图片替换旧图片的工作已基本完成了。HTML 应该如清单 5 所示。

清单 5. 用新图片替换旧图片

```
<html>
 <head>
  <title>Magic Hat</title>
  <script language="JavaScript">
    function showRabbit() {
      var hatImage = document.getElementById("topHat");
      var newImage = document.createElement("img");
      newImage.setAttribute("src", "rabbit-hat.gif");
      var imgParent = hatImage.parentNode;
      imgParent.insertBefore(newImage, hatImage);
      imgParent.removeChild(hatImage);
    }
  </script>
 </head>

 <body>
  <h1 align="center">Welcome to the DOM Magic Shop!</h1>
```

```
  <form name="magic-hat">
   <p align="center">
    <img src="topHat.gif" id="topHat" />
    <br /><br />
    <input type="button" value="Hocus Pocus!" />
   </p>
  </form>
 </body>
</html>
```

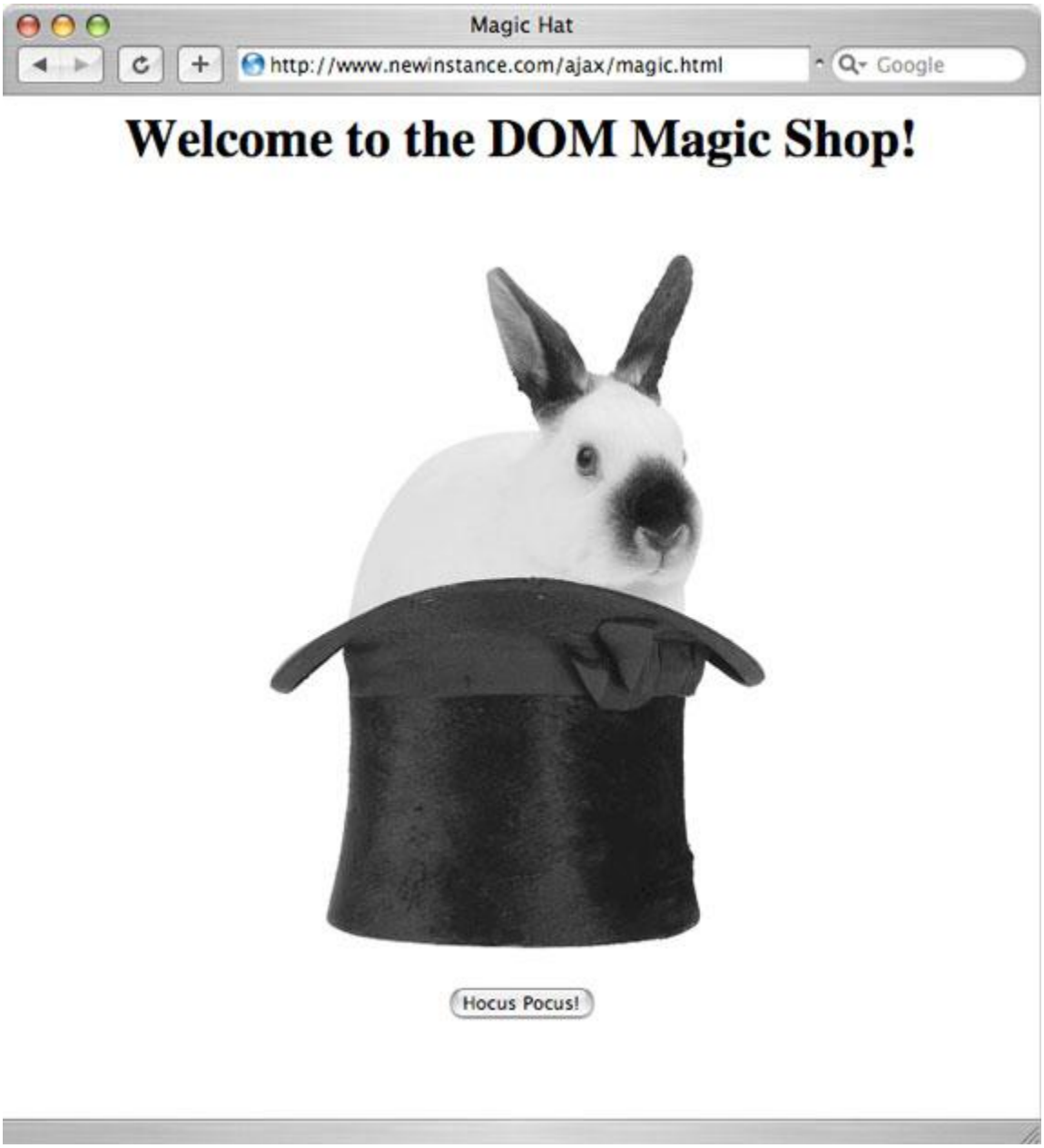
连接 JavaScript

最后一步，可能也是最简单的，就是把 HTML 表单连接到刚刚编写的 JavaScript 函数。需要每当用户点击 **Hocus Pocus!** 按钮的时候运行 `showRabbit()` 函数。为此只要向 HTML 中增加一个简单的 `onClick` 事件处理程序即可。

```
<input type="button" value="Hocus Pocus!" onClick="showRabbit();" />
```

这种简单的 JavaScript 编程应该非常容易了。将其添加到 HTML 页面中，保存它然后在 Web 浏览器中打开。页面初看起来应该和图 1 相同，但是点击 **Hocus Pocus!** 后应该看到图 3 所示的结果。

图 3. 兔子戏法



回页首

替换图片，简单的办法

如果回顾替换图片的步骤，再看看节点的各种方法，可能会注意到方法 `replaceNode()`。该方法可用于把

一个节点替换为另一个节点。再考虑一下前面的步骤：

1. 创建新的 `img` 元素。
2. 访问当前 `img` 元素的父元素，也就是它的容器。
3. 在已有 `img` 元素*之前* 插入新的 `img` 元素作为该容器的子元素。
4. 删除原来的 `img` 元素。
5. 连接起来以便在用户点击 **Hocus Pocus!** 的时候调用刚刚创建的 JavaScript 函数。

使用 `replaceNode()` 可以减少需要的步骤数。可以将第 3 步和第 4 步合并在一起：

1. 创建新的 `img` 元素。
2. 访问当前 `img` 元素的父元素，也就是它的容器。
3. 用创建的新元素替换旧的 `img` 元素。
4. 连接起来以便在用户点击 **Hocus Pocus!** 的时候调用刚刚创建的 JavaScript 函数。

这看起来不是什么大事，但确实能够简化代码。清单 6 说明了这种修改：去掉了 `insertBefore()` 和 `removeChild()` 方法调用。

清单 6. 用新图片替换旧图片（一步完成）

```
<html>
 <head>
  <title>Magic Hat</title>
  <script language="JavaScript">
    function showRabbit() {
      var hatImage = document.getElementById("topHat");
      var newImage = document.createElement("img");
      newImage.setAttribute("src", "rabbit-hat.gif");
      var imgParent = hatImage.parentNode;
      imgParent.replaceChild(newImage, hatImage);
    }
  </script>
 </head>

 <body>
  <h1 align="center">Welcome to the DOM Magic Shop!</h1>
  <form name="magic-hat">
   <p align="center">
    <img src="topHat.gif" id="topHat" />
    <br /><br />
    <input type="button" value="Hocus Pocus!" onClick="showRabbit();"  />
   </p>
  </form>
 </body>
```

```
</html>
```

当然这不是什么大的修改，但是说明了 DOM 编码中一件很重要的事：执行一项任务通常有多种方法。如果仔细审阅可用 DOM 方法看看是否有更简单的方法可以完成任务，很多时候都会发现可以将四五个步骤压缩为两三个步骤。

回页首

替换图片，（真正）简单的办法

既然指出了执行一项任务几乎总是有更简单的方法，现在就说明用兔子图片替换帽子图片的*简单得多* 的办法。阅读本文的过程中有没有想到这种方法？提示一下：与属性有关。
要记住，图片元素很大程度上是由其 `src` 属性控制的，他引用了某个地方的文件（不论是本地 URI 还是外部 URL）。到目前为止，我们一直用新图片替换图片节点，但是直接*修改已有图片的 `src` 属性要简单得多！*这样就避免了创建新节点、寻找父节点和替换旧节点的所有工作，只要一步就能完成了：

```
hatImage.setAttribute("src", "rabbit-hat.gif");
```

这样就够了！看看清单 7，它显示了这种解决方案，包括整个网页。

清单 7. 修改 src 属性

```
<html>
 <head>
  <title>Magic Hat</title>
  <script language="JavaScript">
    function showRabbit() {
      var hatImage = document.getElementById("topHat");
      hatImage.setAttribute("src", "rabbit-hat.gif");
    }
  </script>
 </head>

 <body>
  <h1 align="center">Welcome to the DOM Magic Shop!</h1>
  <form name="magic-hat">
```

```
   <p align="center">
    <img src="topHat.gif" id="topHat" />
    <br /><br />
    <input type="button" value="Hocus Pocus!" onClick="showRabbit();"  />
   </p>
  </form>
 </body>
</html>
```

这是 DOM 最棒的一点：更新属性的时候网页马上就会改变。只要图片指向新的文件，浏览器就加载该文件，页面就更新了。不需要重新加载，甚至不需要创建新的图片元素！结果仍然和图 3 相同，只不过代码简单得多了。

回页首

把兔子藏起来

现在网页看起来很漂亮，但是仍然有点原始。虽然兔子从帽子中跳出来了，但是屏幕下方的按钮仍然显示 **Hocus Pocus!** 和调用 `showRabbit()`。这就是说如果在兔子出来之后仍然点击按钮，就是在浪费处理时间。更重要的是，它毫无用处，而没有用的按钮不是好东西。我们来看看能否利用 DOM 再作一些修改，无论兔子在帽子里还是出来都让这个按钮派上用场。

修改按钮的标签

最简单的是当用户点击按钮之后改变它的标签。这样就不会看起来像还有什么魔法，网页中最糟糕的就是暗示用户错误的东西。在修改按钮的标签之前需要访问该节点，而在此之前需要引用按钮 ID。这是老套路了，清单 8 为按钮增加了 `id` 属性。

清单 8. 增加 id 属性

```
<html>
 <head>
  <title>Magic Hat</title>
  <script language="JavaScript">
    function showRabbit() {
      var hatImage = document.getElementById("topHat");
      hatImage.setAttribute("src", "rabbit-hat.gif");
```

```
    }
  </script>
 </head>

 <body>
  <h1 align="center">Welcome to the DOM Magic Shop!</h1>
  <form name="magic-hat">
   <p align="center">
    <img src="topHat.gif" id="topHat" />
    <br /><br />
    <input type="button" value="Hocus Pocus!" id="hocusPocus"
          onClick="showRabbit();" />
   </p>
  </form>
 </body>
</html>
```

现在用 JavaScript 访问按钮很简单了：

```
function showRabbit() {
  var hatImage = document.getElementById("topHat");
  hatImage.setAttribute("src", "rabbit-hat.gif");
  var button = document.getElementById("hocusPocus");
}
```

当然，您可能已经输入了下面这行 JavaScript 来改变按钮的标签值。这里再次用到了 `setAttribute()`：

```
function showRabbit() {
  var hatImage = document.getElementById("topHat");
  hatImage.setAttribute("src", "rabbit-hat.gif");
  var button = document.getElementById("hocusPocus");
  button.setAttribute("value", "Get back in that hat!");
}
```

通过这个简单的 DOM 操作，兔子跳出来之后按钮的标签马上就会改变。现在，HTML 和完成的 `showRabbit()` 函数如清单 9 所示。

清单 9. 完成的网页

```

<html>
 <head>
  <title>Magic Hat</title>
  <script language="JavaScript">
    function showRabbit() {
```

```
    var hatImage = document.getElementById("topHat");
    hatImage.setAttribute("src", "rabbit-hat.gif");
    button.setAttribute("value", "Get back in that hat!");
   }
  </script>
 </head>

 <body>
  <h1 align="center">Welcome to the DOM Magic Shop!</h1>
  <form name="magic-hat">
   <p align="center">
    <img src="topHat.gif" id="topHat" />
    <br /><br />
    <input type="button" value="Hocus Pocus!" id="hocusPocus"
          onClick="showRabbit();" />
   </p>
  </form>
 </body>
</html>
```

把兔子收回去

从此新的按钮标签中可能已经猜到，现在要把兔子收回帽子中去。基本上和放兔子出来完全相反：将图片的 `src` 属性再改回旧图片。创建一个新的 JavaScript 函数来完成这项任务：

```
function hideRabbit() {
  var hatImage = document.getElementById("topHat");
  hatImage.setAttribute("src", "topHat.gif");
  var button = document.getElementById("hocusPocus");
  button.setAttribute("value", "Hocus Pocus!");
}
```

实际上仅仅把 `showRabbit()` 函数的功能翻转了过来。将图片改为原来的没有兔子的大礼帽，抓取按钮，将标签改为 **Hocus Pocus!**

回页首

事件处理程序

现在这个示例应用程序有一个大问题：虽然按钮的*标签* 改变了，但是单击按钮时的动作*没有* 变。幸运的是，当用户单击按钮时可以使用 DOM 改变事件或者发生的动作。因此，如果按钮上显示 **Get back in that hat!**，点击的时候需要运行 `hideRabbit()`。相反，一旦兔子藏了起来，按钮又返回来运行 `showRabbit()`。

查看 HTML 就会发现这里处理的事件是 `onClick`。在 JavaScript 中，可以通过按钮的 `onclick` 的属性来引用该事件。(要注意，在 HTML 中该属性通常称为 `onClick`，其中 *C* 大写；而在 JavaScript 中则称为 `onclick`，全部小写。) 因此可以改变按钮触发的事件：只要赋给 `onclick` 属性一个新的函数。

但是有点细微的地方：`onclick` 属性需要提供函数引用——不是函数的字符串名称，而是函数本身的引用。在 JavaScript 中，可以按名称引用函数，不需要带括号。因此可以这样修改点击按钮时执行的函数：

```
button.onclick = myFunction;
```

因此在 HTML 中作这种修改很简单。看看清单 10，它切换按钮触发的函数。

避免使用 addEventHandler()

除了 `onclick` 属性外，还有一个方法可用于添加 `onClick` 或 `onBlur` 这样的事件处理程序，毫不奇怪这个方法就叫 `addEventHandler()`。不幸的是，Microsoft™ Internet Explorer™ 不支持这个方法，如果在 JavaScript 中使用它，就会有数百万 Internet Explorer 用户除了错误外从网页中什么也看不到（可能还有抱怨）。不使用这个方法，应用本文中介绍的办法也能达到同样的效果，而且在 Internet Explorer 上也*有效*。

清单 10. 改变按钮的 onClick 函数

```
<html>
 <head>
  <title>Magic Hat</title>
  <script language="JavaScript">
    function showRabbit() {
      var hatImage = document.getElementById("topHat");
      hatImage.setAttribute("src", "rabbit-hat.gif");
      var button = document.getElementById("hocusPocus");
      button.setAttribute("value", "Get back in that hat!");
      button.onclick = hideRabbit;
    }

    function hideRabbit() {
      var hatImage = document.getElementById("topHat");
      hatImage.setAttribute("src", "topHat.gif");
      var button = document.getElementById("hocusPocus");
      button.setAttribute("value", "Hocus Pocus!");
      button.onclick = showRabbit;
    }
  </script>
```

```
 </head>

 <body>
  <h1 align="center">Welcome to the DOM Magic Shop!</h1>
  <form name="magic-hat">
   <p align="center">
    <img src="topHat.gif" id="topHat" />
    <br /><br />
    <input type="button" value="Hocus Pocus!" id="hocusPocus"
          onClick="showRabbit();" />
   </p>
  </form>
 </body>
</html>
```

这样就得到了一个完成的、可以使用的 DOM 应用程序。自己试试吧！

回页首

结束语

现在您应该非常熟悉 DOM 了。前面的文章介绍了使用 DOM 所涉及到的基本概念，详细地讨论了 API，现在又建立一个简单的基于 DOM 的应用程序。一定要花点时间仔细阅读本文，并自己尝试一下。

虽然这是专门讨论文档对象模型的系列文章的最后一期，但肯定还会看到其他关于 DOM 的文章。事实上，如果在 Ajax 和 JavaScript 世界中不使用 DOM 就很难做多少事，至少在一定程度上如此。无论要创建复杂的突出显示还是移动效果，或者仅仅处理文本块或图片，DOM 都提供了一种非常简单易用的访问 Web 页面的方式。

如果对如何使用 DOM 仍然感觉没有把握，花点时间温习一下这三篇文章；本系列的其他文章在使用 DOM 的时候不再多作解释，读者也不希望迷失在这些细节之中而忽略关于其他概念的重要信息，比如 XML 和 JSON。为了保证能够熟练地使用 DOM，自己编写几个基于 DOM 的应用程序试试，这样就很容易理解后面将要讨论的一些数据格式问题了。

第 7 部分：在请求和响应中使用 XML

偶尔使用 Ajax 的开发人员也会注意到 Ajax 中的 *x* 并意识到它代表 *XML*。XML 是编程中最常用的数据格式之一，对于异步应用程序中的服务器响应能够带来切实的好处。在本文中，您将看到服务器如何在请求响应中发送 XML。

现在如果不使用 XML 就不能进行任何有意义的编程。无论考虑转向 XHTML 的网页设计人员、使用 JavaScript 的 Web 程序员、使用部署描述文件和数据绑定的服务器端程序员，还是研究基于 XML 的数据库的后端开发人员，都在使用这种可扩展标记语言。因此，XML 被认为是 Ajax 底层的核心技术之一就不足为奇了。

但是，这种观点反映到 Ajax 应用程序就表现在其核心对象所选的名称 —— `XMLHttpRequest`，这个名称不是很好，因为它并没有反映技术上的实际情况。换句话说，多数人之所以认为 XML 是 Ajax 的核心组成部分，仅仅是因为他们想当然地以为 `XMLHttpRequest` 对象在任何时候都使用 XML。但实情并非如此，本文第一部分给出了原因。实际上，您将看到在多数 Ajax 应用程序中 XML 很少出现。

XML 确实有应用在 Ajax 中，而且 `XMLHttpRequest` 也支持这种用法。也确实没有什么能阻挡您向服务器发送 XML。在本系列前面的文章中，我们使用普通文本和名/值参数发送数据，但 XML 也是一种可行的格式。本文将介绍如何来这样做。但最重要的是，我将讨论*为何*可以使用 XML 作为请求格式，以及为何在多数情况下*不应该*使用它。

XML：到底用没用？

对 Ajax 应用程序及它们使用 XML 的情况很容易犯想当然的错误：这种技术的名称（Ajax）及其使用的核心对象（`XMLHttpRequest`）都暗示了 XML 的使用，谈到 Ajax 应用程序的时候也经常听到 XML。但是，这种观点大错特错，如果希望在编写异步应用程序时真正做到胸有成竹，必须*知道*这种想法是错误的，而且最好知道*为什么*错误。

XMLHttpRequest：糟糕的名称和 HTTP

一项技术可能遇到的最糟的境况之一是它变得太炙手可热以至于无法再改变它的一些基本内容。`XMLHttpRequest` 恰恰是这种情形，它是 Ajax 应用程序中使用的基本对象。听起来它似乎是为通过 HTTP 请求发送 XML 或者以某种 XML 格式发出 HTTP 请求而设计的。但不论这个对象的名称*听起来*像什么，实际上它要*做*的只不过是为客户机代码（在网页中通常是 JavaScript）提供一种发送 HTTP 请求的方式。仅此而已，别无其他。

因此，如果将 `XMLHttpRequest` 改成某种更准确的名称可能更好一些，比如 `HttpRequest`，或者简简单单的 `Request`。但是，现在成千上万的人在应用程序中使用了 Ajax，而且我们知道需要几年时间（如果不是十几年的话）大部分用户才会改用 Internet Explorer 7.0 或 Firefox 1.5 这些新版本的浏览器，因此这么修改实际上是不可行的。最终我们不得不使用 `XMLHttpRequest`，这就要求开发人员要知道其名不符实的这一事实。

在一定程度上讲，对于不支持 `XMLHttpRequest` 的浏览器（特别是在 Windows 上）的最佳回溯方法之一就是使用 Microsoft `IFRAME` 对象。听起来可不像是 XML、HTTP 或请求，是不是？当然，所有这些都可能涉及到，但是这正清楚地说明了一点 —— `XMLHttpRequest` 对象更多的是关于在不重新加载页面的情况发出请求，而不会太多地涉及 XML 甚至 HTTP。

请求是 HTTP 而非 XML

另一种常见的错误是认为 XML 在幕后使用 —— 坦白地说，我也曾这么认为！但是，持这种观点表明您对这种技术还不甚了解。当用户打开浏览器从服务器上请求网页时，会输入 `http://www.google.com` 或者 `http://www.headfirstlabs.com` 这样的东西。即便不输入 `http://`，浏览器也会在地址栏的这部分

加上。第一部分，即 http://，是关于如何通信的很直观的线索：通过超文本传输协议 HTTP。在网页中编写代码与服务器通信时，无论使用 Ajax 还是普通的表单 POST，甚至超链接，打交道的都是 HTTP。

既然浏览器和服务器之间的所有 Web 通信都通过 HTTP 进行，认为 XML 是 XMLHttpRequest 幕后所用的某种传输技术的想法就毫无道理了。当然在 HTTP 请求*中*可以发送 XML，但是 HTTP 是一个精确定义的协议，短时间内不可能消失。除了在请求中明确使用 XML，或者服务器用 XML 发送响应之外，XMLHttpRequest 对象使用的只是普普通通的 HTTP。因此，当再有人对您说“哦，称为 XMLHttpRequest 是因为在幕后使用 XML”的时候，您最好一笑了之，并耐心地解释什么是 HTTP，告诉他们虽然 XML 可以*通过* HTTP 发送，但 XML 是一种数据格式而不是传输协议。通过这样的讨论，加深对它的理解。

HTTPS：仍然是 HTTP

那些刚接触 Web 的人可能对 https://intranet.nextel.com 这样的 URL 感到奇怪。https 表示安全的 HTTP，只是使用了比一般 Web 请求更安全的 HTTP 协议形式。因此即便是 HTTPS，实际上用的仍然是 HTTP，虽然增加了某些安全层来挡住那些好奇的眼睛。

回页首

使用 XML（真正）

到目前为止，我说的只是 Ajax 在哪些地方*不*使用 XML。但 Ajax 中的 *x* 和 XMLHttpRequest 中的 XML 仍然有其实际意义，在 Web 应用程序中使用 XML 有多种选择。这一节将讨论基本的选择，剩下的部分再深入探讨细节问题。

XML 选项

在异步应用程序中 XML 有两种基本的用法：

- 以 XML 格式从网页向服务器发送请求
- 以 XML 格式在网页中从服务器接收请求

其中第一种用法，即用 XML 发送请求，需要将请求的格式设置为 XML，可以使用 API 来完成，也可以与文本连成字符串，然后将结果发送到服务器。按照这种思路，主要的任务就是通过既符合 XML 规则又能被服务器理解的方式构造请求。因此这里的关键实际上是 XML 格式，得到需要发送的数据之后，只需要用 XML 语法将其包装起来。本文后面讨论 XML 在 Ajax 应用程序中的这种用法。

第二种用法，即用 XML 接收请求，需要从服务器上接收响应，然后从 XML 提取数据（同样，可以用 API 或者采用蛮力方法）。这种情况下，关键在于来自服务器的数据，而您恰好需要从 XML 中提取这些数据以便使用。这是本系列下一期文章的主题，到那时候我们再详加讨论。

一点忠告

再详细讨论使用 XML 的细节之前，首先给您一句忠告：XML 不是一种简洁、快速和节省空间的格式。在后面几节以及本系列的下一期文章中将看到，在上下文中使用 XML 确实有一些很好的理由，XML 与

普通文本的请求和响应（特别是响应）相比也确实有一些长处。但是，和普通文本相比，XML 通常总会占用更多的空间，速度也更慢，因为需要在消息中增加 XML 所需要的标签和语义。
如果需要编写速度很快、看起来像桌面应用的程序，XML 可能不是最佳选择。如果从普通文本开始，然后发现确实需要 XML，那么就使用它；但是如果从一开始就使用 XML，基本上可以肯定一定会降低应用程序的响应性。多数情况下，与将文本转化成下面这种 XML 相比，发送普通文本会更快一些（使用类似 `name=jennifer` 的名/值对）：

```
<name>jennifer</name>
```

看看哪些地方使 XML 增加了处理时间：将文本包装成 XML；发送额外信息（要注意我没有包含任何包围元素、XML 头或者可能出现在实际请求中的其他任何内容）；让服务器解析 XML、生成响应、用 XML 包装响应，并将它发送*回*网页；让网页解析响应，最后使用它。因此要清楚什么时候使用 XML，不要一开始就认为它在很多情况下都能够加快应用程序；但，它可以增强灵活性，这就是我们现在要讨论的。

回页首

从客户机到服务器的 XML

我们来看看将 XML 作为从客户机向服务器发送数据的格式。我们首先讨论技术上的实现，然后花些时间分析什么时候适合什么时候不适合使用它。

发送名/值对

在您编写的 90% Web 应用程序中，最终都会使用名/值对发送到服务器。比方说，如果用户在网页表单中输入姓名和地址，可能希望数据采用下列形式：

```
firstName=Larry
lastName=Gullahorn
street=9018 Heatherhorn Drive
city=Rowlett
state=Texas
zipCode=75080
```

如果使用普通文本把这些数据发送到服务器，可以使用清单 1 所示的代码。类似于本系列第一期文章中使用的那个例子。请参阅参考资料。

清单 1. 使用普通文本发送名/值对

```

function callServer() {
```

```
  // Get the city and state from the Web form
  var firstName = document.getElementById("firstName").value;
  var lastName = document.getElementById("lastName").value;
  var street = document.getElementById("street").value;
  var city = document.getElementById("city").value;
  var state = document.getElementById("state").value;
  var zipCode = document.getElementById("zipCode").value;

  // Build the URL to connect to
  var url = "/scripts/saveAddress.php?firstName=" + escape(firstName) +
    "&lastName=" + escape(lastName) + "&street=" + escape(street) +
    "&city=" + escape(city) + "&state=" + escape(state) +
    "&zipCode=" + escape(zipCode);

  // Open a connection to the server
  xmlHttp.open("GET", url, true);

  // Set up a function for the server to run when it's done
  xmlHttp.onreadystatechange = confirmUpdate;

  // Send the request
  xmlHttp.send(null);
}
```

将名/值对转化成 XML

如果希望这样使用 XML 作为数据格式，首先要做的是找到一种基本 XML 格式来存储数据。显然，名/值对可以全部转化成 XML 元素，以其中的名称作为元素名，值作为元素的内容：

```
<firstName>Larry</firstName>
<lastName>Gullahorn</lastName>
<street>9018 Heatherhorn Drive</street>
<city>Rowlett</city>
<state>Texas</state>
<zipCode>75080</zipCode>
```

当然，XML 要求有一个根元素；如果使用*文档片段*（XML 文档的一部分）的话则需要一个封闭元素。因此可能需要将上述 XML 转化成下面的形式：

```
<address>
  <firstName>Larry</firstName>
  <lastName>Gullahorn</lastName>
  <street>9018 Heatherhorn Drive</street>
  <city>Rowlett</city>
  <state>Texas</state>
  <zipCode>75080</zipCode>
```

```
</address>
```

现在基本上可以准备在 Web 客户机上创建这种结构并发送到服务器了。

通信，口头上的

在网络上传输 XML 之前，需要保证服务器以及发送数据的脚本能够接受 XML。现在对很多人来说这么强调似乎有点多余，认为这是理所当然的，但是很多新手往往认为只要通过网络发送 XML，就能够被正确地接收和解释。

实际上，需要两个步骤来保证发送的 XML 的数据能够被正确地接收：

1. 保证向其发送 XML 的脚本能够接受 XML 数据格式。
2. 保证脚本认可发送数据所采用的特定 XML 格式和结构。

这两方面都可能要求您进行人际沟通，必须明确地告知对方！严格地说，如果确实需要发送 XML 数据，多数脚本作者都会帮助您，因此寻找能够接受 XML 的脚本应该不难。但是，仍然需要保证格式是脚本所希望的格式。比方说，假设服务器接受下列格式的数据：

```
<profile>
  <firstName>Larry</firstName>
  <lastName>Gullahorn</lastName>
  <street>9018 Heatherhorn Drive</street>
  <city>Rowlett</city>
  <state>Texas</state>
  <zip-code>75080</zip-code>
</profile>
```

看起来和上面的 XML 类似，只有两点不同：

1. 来自客户机的 XML 封装在 `address` 元素，但是服务器要求数据封装在 `profile` 元素中。
2. 来自客户机的 XML 使用了 `zipCode` 元素，而服务器希望邮政编码放在 `zip-code` 元素中。

从大的层面上来说，这些小问题仅仅是服务器接收和处理数据的区别，但是服务器会彻底失败，在网页上（可能向其用户）显示意义含糊的错误消息。因此必须明确服务器的期望的格式，并把要发送的数据塞进那种格式。然后，只有在这时才会涉及到从客户机向服务器发送 XML 数据的真正的技术问题。

向服务器发送 XML

当向服务器发送 XML 的时候，更多的代码用于获取数据和包装成 XML，而不是真正的传输数据。实际上，只要准备好发送到服务器的 XML 字符串，发送工作就和普通文本一样了，如清单 2 所示。

清单 2. 用 XML 发送名/值对

```
function callServer() {
  // Get the city and state from the Web form
  var firstName = document.getElementById("firstName").value;
  var lastName = document.getElementById("lastName").value;
  var street = document.getElementById("street").value;
```

```
  var city = document.getElementById("city").value;
  var state = document.getElementById("state").value;
  var zipCode = document.getElementById("zipCode").value;

  var xmlString = "<profile>" +
    "  <firstName>" + escape(firstName) + "</firstName>" +
    "  <lastName>" + escape(lastName) + "</lastName>" +
    "  <street>" + escape(street) + "</street>" +
    "  <city>" + escape(city) + "</city>" +
    "  <state>" + escape(state) + "</state>" +
    "  <zip-code>" + escape(zipCode) + "</zip-code>" +
    "</profile>";

  // Build the URL to connect to
  var url = "/scripts/saveAddress.php";

  // Open a connection to the server
  xmlHttp.open("POST", url, true);

  // Tell the server you're sending it XML
  xmlHttp.setRequestHeader("Content-Type", "text/xml");

  // Set up a function for the server to run when it's done
  xmlHttp.onreadystatechange = confirmUpdate;

  // Send the request
  xmlHttp.send(xmlString);
}
```

大部分代码都很简单，只有少数地方值得提一下。首先，请求中的数据必须手工格式化为 XML。阅读了三篇关于使用文档对象类型的文章之后，再来讨论它是不是很简单了？虽然不禁止在 JavaScript 中使用 DOM 创建 XML 文档，但是在通过 GET 或 POST 请求发送到网络上之前必须将 DOM 对象转化成文本。因此使用常规字符串操作来格式化数据更简单一些。当然，这样很容易出现错误和误输入，因此在编写处理 XML 的代码时必须非常小心。

建立 XML 之后，按照和发送文本基本相同的方式打开连接。对于 XML 最好使用 POST 请求，因为有些浏览器限制了 GET 请求字符串的长度，而 XML 可能很长，可以看到清单 2 中把 GET 改成了 POST 方法。此外，XML 通过 `send()` 方法发送，而不是附加在请求 URL 最后的参数。这些都是非常细微的区别，很容易修改。

但是必须编写一行新的代码：

```
xmlHttp.setRequestHeader("Content-Type", "text/xml");
```

看起来很难理解，它只不过是告诉服务器要发送的是 XML 而不是一般的名/值对。无论哪种情况，发送的数据都是文本，但这里使用 `text/xml` 或者 XML 作为普通文本发送。如果使用名/值对，对应的行应该

是：

```
xmlHttp.setRequestHeader("Content-Type", "text/plain");
```

如果忘记告诉服务器发送的是 XML，就会出现问题，因此不要忘掉这一步骤。
完成这些之后，剩下的就是调用 send() 并传入 XML 字符串了。服务器将收到您的 XML 请求，并（假设已经做好了准备工作）接受 XML，解释它，然后返回响应。实际上要做的只有这么多 —— XML 请求只需要稍微修改代码。

回页首

发送 XML：好还是不好？

在结束 XML 响应的 XML 请求（以及本文）之前，我们花点时间讨论一下在请求中使用 XML 的感受。前面已经提到，就传输而言 XML 完全不是最快的方式，但是还有更多因素要考虑。

构造 XML 不是简单的事情

首先必须认识到，对于请求来说构造 XML 不是简单的事。如清单 2 所示，数据很快就会和 XML 语义纠缠在一起：

```
var xmlString = "<profile>" +
  "  <firstName>" + escape(firstName) + "</firstName>" +
  "  <lastName>" + escape(lastName) + "</lastName>" +
  "  <street>" + escape(street) + "</street>" +
  "  <city>" + escape(city) + "</city>" +
  "  <state>" + escape(state) + "</state>" +
  "  <zip-code>" + escape(zipCode) + "</zip-code>" +
  "</profile>";
```

似乎还不坏，但是要知道这是只有六个字段的 XML 片段。开发的多数 Web 表单都有十到十五个字段，虽然不一定所有的请求都使用 Ajax，但是应该考虑这种情况。至少要花和实际数据同样多的时间来处理尖括号和标签名称，有可能使本来很少的输入变得非常大。
这里的另一个问题前面已经提到，即必须手工创建 XML。使用 DOM 不是一种好的选择，因为没有简单易行的办法将 DOM 对象转化成在请求中发送的字符串。因此像这样使用字符串处理是最好的办法，不过也是一种维护起来最困难和新开发人员最难理解的方法。在这个例子中，所有 XML 都在一行中构造完成，如果分为多步只会更加混乱。

XML 没有为请求增加任何东西

除了复杂性的问题之外，和普通文本以及名/值对相比，在请求中使用 XML 实际上没有多少好处（如果有

的话）。要注意，本文坚持使用前面用名/值对发送的同一些数据（请参阅清单 1）来用 XML 发送。我没有提什么数据能用 XML 但是*不能*用普通文本发送，这是因为实际上*没有*任何东西可用 XML 而不能用普通文本发送。

事实上这就是 XML 和请求的底线：不是一定非要这么做不可。在本系列的下一期文章中将看到服务器可以使用 XML 实现普通文本很难做到的一些事情，但请求不属于这种情况。因此除非和*只*接受 XML 的脚本（确实存在这样的脚本）打交道，在请求中最好使用普通文本。

回页首

结束语

通过本文，您现在可能已经开始对 Ajax 中的 XML 有一些更深的理解了。您知道 Ajax 应用程序不一定要使用 XML，XML 也不是数据传输中的什么法宝。还知道从网页向服务器发送 XML 不是多么难的事情。更重要的是，您知道为了确保服务器能够处理和响应请求需要做什么：必须保证服务器脚本接受 XML，而且能够识别用于发送数据的格式。

您还应该非常清楚 XML 对于请求来说并不一定是很好的数据格式。在以后的文章中，您将看到 XML 在某些情况下是有利的，但在多数请求中，它只会降低速度和增加复杂性。因此虽然通常我都会建议您马上应用在文章中学到的内容，但是对本文来说，我建议您在应用这里学到的知识时最好三思而后行。XML 请求在 Ajax 应用程序中有自己的价值，但是并不像您所想象的那么大。

在下一期文章中，我们将讨论服务器如何使用 XML 做出响应，以及 Web 应用程序如何处理这些响应。令人高兴的是，服务器能够将 XML 发送回 Web 应用程序，这样做的理由比较充分，因此那篇文章中的技术细节更实用，目前您只需要知道 XML 为何并非一定是最佳选择 —— 至少对发送请求而言。您可以尝试使用 XML 作为请求数据格式实现某些 Web 应用程序，然后再换回普通文本，看看哪种办法更快更简单。下一期文章再见。

第 8 部分：在请求和响应中使用 XML

在 本系列的上一篇文章 中，您看到了 Ajax 应用程序如何以 XML 格式化发往服务器的请求。还了解了为什么这在大多数情况下并不是一个好主意。这篇文章主要探讨在大多数情况下*确实是* 好主意的一种做法：向客户机返回 XML 响应。

我其实并不喜欢写那种主要告诉您什么*不应该* 做的文章。很多时候，那都会是一篇非常愚蠢的文章。我要在前半篇文章中解释某些东西，然后在后半篇文章中说明使用您刚刚才学会的那种技术是一个多么糟糕的主意。在很大程度上，上一期文章正是这样一种情况（如果您错过了那一期文章，请查看 参考资料 中的链接），那篇文章教您如何将 XML 作为 Ajax 应用程序的请求数据格式使用。

但愿这篇文章能够弥补您花费在学习 XML 请求上的时间。在 Ajax 应用程序中，使用 XML 作为发送数据的格式的理由很少，但使服务器*向* 客户机*回发* XML 的理由很多。因此，您在上一篇文章中学到的关于 XML 的知识最终将在这篇文章中体现出某些价值。

服务器（有时）不能响应太多的请求

在深入钻研从服务器获取 XML 响应的技术之前，您需要理解，为什么说使服务器发送 XML 来响应请求是一个好主意（以及这与客户机发送 XML 请求不同的原因所在）。

客户机以名称/值对发送请求

回忆一下上一篇文章，就会知道，在大多数情况下，客户机不需要使用 XML，因为他们会使用名称/值对发送请求。因此，您可能会发送一个这样的名称：`name=jennifer`。只需简单地在连续的名称/值对之间添加一个“与”符号（`&`），即可将其放在一起，就像这样：`name=jennifer&job=president`。使用简单的文本和这些名称值对，客户机即可轻松向服务器请求多个值。很少需要用到 XML 提供的额外结构（及其带来的额外开销）。

实际上，需要向服务器发送 XML 的所有理由都差不多可以归入以下两个基本的类别中：

- **服务器*仅* 接受 XML 请求。**在这类情况下，您别无选择。上一期文章中介绍的基础知识应已使您掌握了发送此类请求所必需的工具。
- **您正在调用一个仅接受 XML 或 SOAP 请求的远程 API。**这实际上就是上一种情况的特例，但值得单独拿出来提一下。如果您希望在一个异步请求中使用来自 Google 或 Amazon 的 API，就会有一些特殊的考虑事项。在下一期的文章中，我将介绍这些考虑事项，还会给出一些向 API 发送此类请求的示例。

服务器无法（以一种标准方式）发送名称/值对

在您发送名称/值对时，Web 浏览器会发送请求，平台会响应该请求，并承载一个服务器程序，配合它将那些名称/值对转换成服务器程序可以轻松处理的数据。实际上，每一种服务器端技术 —— 从 Java™ servlet 到 PHP、再到 Perl、Ruby on Rails —— 都允许您调用多种方法来根据名称获取值。因此，获取 `name` 属性只是小事一桩。

这种情况并不会将我们引向另外一个方向。如果服务器使用字符串 `name=jennifer&job=president` 应答一个应用程序，客户机没有任何标准化的简便方法来将每个对拆分成名称和值。您必须手动解析所返回的数据。如果服务器返回一个由名称/值对构成的响应，这样的响应的解释难度与使用分号、竖线或其他任何非标准格式化字符相同。

> **给我一点空间！**
> 在绝大多数 HTTP 请求中，转义序列 `%20` 用于表示一个空格，文本“Live Together, Die Alone”将以

对于您来说，这就意味没有任何简单的方法在响应中使用纯文本、使客户机以一种标准的方法获取并解释响应，至少在响应包含多个值时是如此。假设您的服务器只是要发回数字 42，那么纯文本是很好的选择。但如果服务器要一次性发回电视剧 *Lost, Alias* 和 *Six Degrees* 的近期收视率又该怎么办呢？尽管可以选择许多种方法来使用纯文本发送这一响应（清单 1 给出了一些示例），但没有一种是不需客户机进行某些处理的极其简单的方法，也没有一种是标准化的方法。

```
Live%20Together,%20Die%20Alone
```
的形式通过 HTTP 发送。

清单 1. 收视率的服务器响应（不同版本）

```
show=Alias&ratings=6.5|show=Lost&ratings=14.2|show=Six%20Degrees&ratings=9.1

Alias=6.5&Lost=14.2&Six%20Degrees=9.1

Alias|6.5|Lost|14.2|Six%20Degrees|9.1
```

尽管不难找到拆分这些响应字符串的方法，但客户机将不得不根据分号、等号、竖线和与符号解析并拆分这些字符串。这不是编写使其他开发人员能够轻松理解和维护的健壮代码的方法。

进入 XML

意识到没有任何标准的方法可以使服务器使用名称/值对响应客户机之后，使用 XML 的原因也就显而易见了。向客户机发送数据时，名称/值对是非常好的选择，因为服务器和服务器端语言可以轻松解释名称/值对；向客户机返回数据时使用 XML 也是如此。在本系列前几期的文章中，您已经看到了利用 DOM 来解析 XML，在后续的文章中，还会看到 JSON 怎样提供了解析 XML 的另一种选择。在所有这一切之上，您可以将 XML 作为纯文本处理，并以这种方式获取其值。因此，有几种方法可从服务器获得 XML 响应，并使用较为标准的代码提取数据，在客户机中使用这些数据。

还有一个额外的好处，XML 非常易于理解。比如说，大多数编写程序的人都能理解 清单 2 中的数据。

清单 2. 收视率的服务器响应（XML 格式）

```
<ratings>
 <show>
  <title>Alias</title>
  <rating>6.5</rating>
 </show>
 <show>
  <title>Lost</title>
  <rating>14.2</rating>
 </show>
 <show>
  <title>Six Degrees</title>
  <rating>9.1</rating>
 </show>
</ratings>
```

在特定分号或撇号的含义方面，清单 2 中的代码没有任何隐晦之处。

回页首

从服务器接收 XML

由于本系列的重点在于 Ajax 应用模式的客户端，因此我不会深入探讨关于服务器端程序如何才能生成 XML 响应的细枝末节。但在您的客户机接收 XML 时，您需要了解一些特殊的考虑事项。
首先，您可使用两种基本的方式处理一个来自服务器的 XML 响应：

- 作为碰巧被格式化为 XML 的纯文本
- 作为一个 XML 文档，由一个 DOM `Document` 对象表示。

其次，举例来说，假设有一个来自服务器的简单 XML 响应。清单 3 展示了与上面介绍的内容相同的收视率程序清单（实际上，是与 清单 2 相同的 XML，在这里再次给出只是为了使您便于查看）。我将在这部分的讨论中使用这段样本 XML。

清单 3. XML 格式的收视率示例

```
<ratings>
 <show>
  <title>Alias</title>
  <rating>6.5</rating>
 </show>
 <show>
  <title>Lost</title>
  <rating>14.2</rating>
 </show>
 <show>
  <title>Six Degrees</title>
  <rating>9.1</rating>
 </show>
</ratings>
```

将 XML 作为纯文本处理

处理 XML 的最简单的选择（至少就学习新的编程技术而言），就是用与处理服务器返回的其他文本片段相同的方法来处理它。换言之，基本上就是忽略数据格式，只关注服务器的响应。

在这种情况下，您要使用请求对象的 `responseText` 属性，就像在服务器向您发送非 XML 响应时一样（参见 清单 4）。

清单 4. 将 XML 作为普通服务器响应处理

```
function updatePage() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var response = request.responseText;

      // response has the XML response from the server
      alert(response);
    }
  }
}
```

在这个代码片段中，`updatePage()` 是回调方法，`request` 是 `XMLHttpRequest` 对象。最终，您将得到把所有一切串联在一起的 XML 响应，位于 `response` 变量之中。如果输出此变量，您会得到类似于清单 5 的结果。（请注意，清单 5 中的代码在正常情况下应该是连续的一个代码行。这里采用多行形式是为了显示正常。）

前文回顾

为避免出现大量重复的代码，本系列这些后续的文章只给出与所探讨的主题相关的那部分代码。因此，清单 4 仅仅展示了 Ajax 客户机代码中的回调方法。如果您对于此方法在更大的异步应用程序环境中的位置尚不清楚，应回顾本系列的前几篇文章，那些文章介绍了 Ajax 应用程序的基础知识。参考资料 中列出了前几篇文章的链接。

清单 5. response 变量的值

```
<ratings><show><title>Alias</title><rating>6.5</rating>
</show><show><title>Lost</title><rating>14.2</rating></show><show>
<title>Six Degrees</title><rating>9.1</rating></show></ratings>
```

这里，要注意的最重要的地方就是 XML 是作为整体运行的。在大多数情况下，服务器不会使用空格和回车来格式化 XML，而是将一切串联在一起，就像您在 清单 5 中看到的那样。当然，您的应用程序不会过于在意空格，所以这算不上什么问题，但确实会使代码阅读起来较为困难。

在这里，您可以使用 JavaScript `split` 函数来拆分此数据，并通过基本的字符串操作来获得元素的名称和值。毫无疑问，这是个令人头疼的过程，它无视于您花费了大量时间来阅读前几期文章中 DOM（Document Object Model）相关内容这一事实。因此，我要强调，您应该牢记：利用 `responseText`，可以轻松使用和输出服务器的 XML 响应，但我不会为您展示太多的代码，在能够使用 DOM 时，您不应选择这种方法来获得 XML 数据，下面您会看到这一点。

将 XML 当成 XML

尽管*可以* 将服务器的 XML 格式的响应视同为其他任何文本响应来处理，但这样做没有很好的理由。首先，如果您一直忠实地阅读这个系列，那么您已经学会了如何使用 DOM 这种对 JavaScript 友好的 API 来操纵 XML。还有更好的事情，JavaScript 和 `XMLHttpRequest` 对象提供了一个属性，它能完美地获取服务器的 XML 响应，并且是以 DOM `Document` 对象的形式来获取它。

清单 6 给出了一个实例。这段代码与 清单 4 类似，但没有使用 `responseText` 属性，回调函数使用的是 `responseXML` 属性。该属性在 `XMLHttpRequest` 上可用，它以 DOM 文档的格式返回服务器的响应。

清单 6. 将 XML 当作 XML

```
function updatePage() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var xmlDoc = request.responseXML;

      // work with xmlDoc using the DOM
    }
  }
}
```

现在您有了一个 DOM `Document`，可以像处理其他任何 XML 一样处理它。例如，随后可能要获取所有 `show` 元素，如 清单 7 所示。

清单 7. 获取所有 show 元素

```
function updatePage() {
  if (request.readyState == 4) {
    if (request.status == 200) {
      var xmlDoc = request.responseXML;

      var showElements = xmlDoc.getElementsByTagName("show");
    }
  }
}
```

如果您熟悉 DOM，从这儿开始，看起来就应该有几分熟悉了。您可以使用您所了解的全部 DOM 方法，轻松操纵从服务器处接收到的 XML。

当然，您也可以混用普通的 JavaScript 代码。例如，可以遍历所有 `show` 元素，如 清单 8 所示。

清单 8. 遍历所有 show 元素

```
function updatePage() {
  if (request.readyState == 4) {
    if (request.status == 200) {
```

```
    var xmlDoc = request.responseXML;

    var showElements = xmlDoc.getElementsByTagName("show");
    for (var x=0; x<showElements.length; x++) {
      // We know that the first child of show is title, and the second is rating
      var title = showElements[x].childNodes[0].value;
      var rating = showElements[x].childNodes[1].value;

      // Now do whatever you want with the show title and ratings
    }
   }
  }
}
```

通过这段非常简单的代码，您正是将 XML 响应作为 XML 而不是无格式的纯文本进行了处理，还使用了一点 DOM 和简单的 JavaScript 来处理服务器响应。更重要的是，您使用了标准化的格式 —— XML，而不是以逗号分隔的值或以竖线分隔的名称/值对。换句话说，您将 XML 用在了最适合它的地方，避免了在不适合的地方使用它（比如说向服务器发送请求时）。

服务器端的 XML：一个简单的示例

我不会谈太多有关如何在服务器上生成 XML 的内容，但花点儿时间看一个简单的示例是值得的，我没有给出过多的注释，以便您自行思考如何应对这样的场景。清单 9 展示了一个简单的 PHP 脚本，假设有一个异步客户机发出了请求，该脚本将输出 XML 来应答此请求。

这是一种强力（brute force）的方法，PHP 脚本实际上是手动生成 XML 输出。您可以找到许多工具包和 API，用于 PHP 和其他众多允许您生成 XML 响应的服务器端语言。无论您的情况如何，这都至少能让您了解生成 XML 并以之应答请求的服务器端脚本看上去是什么样子。

清单 9. 返回 XML 的 PHP 脚本

```
<?php

// Connect to a MySQL database
$conn = @mysql_connect("mysql.myhost.com", "username", "secret-password");
if (!conn)
  die("Error connecting to database: " . mysql_error());

if (!mysql_select_db("television", $conn))
  die("Error selecting TV database: " . mysql_error());

// Get ratings for all TV shows in database
$select = 'SELECT title, rating';
$from   = '  FROM ratings';
$queryResult = @mysql_query($select . $from);
if (!$queryResult)
```

```
  die("Error retrieving ratings for TV shows.');

// Let the client know we're sending back XML
header("Content-Type: text/xml");
echo "<?xml version=\"1.0\" encoding=\"utf-8\"?>";
echo "<ratings>";

while ($row = mysql_fetch_array($queryResult)) {
  $title = $row['title'];
  $rating = $row['rating'];

  echo "<show>
  echo "<title>" . $title . "</title>";
  echo "<rating>" . $rating . "</rating>";
  echo "</show>";
}

echo "</ratings>";

mysql_close($conn);

?>
```

您应能够使用您偏爱的服务器端语言以类似的方式输出 XML。IBM developerWorks 上有许多文章可以帮助您了解如何使用您喜爱的服务器端语言生成 XML 文档（相关链接请参见 参考资料）。

回页首

解释 XML 的其他可选方法

除将 XML 作为无格式文本处理或使用 DOM 处理之外，还有一种非常流行的选择很重要，值得一提。那就是 *JSON*（JavaScript Object Notation），这是一种绑定在 JavaScript 内的自由文本格式。这篇文章中没有足够的空间介绍 JSON，在后续文章中将探讨相关内容；只要提起 XML 和 Ajax 应用程序，您就很可能会听到有人谈论 JSON，因此现在我将告诉您，您的工作伙伴们在谈论的究竟是什么。
大体上，可以用 JSON 做的事，用 DOM 都可以完成，反之亦然；选择主要取决于偏好，当然也要为特定的应用程序选择正确的方法。就现在而言，您应坚持使用 DOM，在接收服务器响应的过程中熟悉 DOM。

我将占用几期文章的篇幅、用大量的时间去探讨 JSON，之后您就可以随意选择在下一个应用程序中使用那种技术了。请继续关注本系列：后续文章中将介绍关于 XML 的更多内容。

回页首

结束语

从本系列上一篇文章开篇起，我几乎一直在谈论 XML，但对于 XML 在 Ajax 应用模式中的贡献而言，我触及的依然仅仅是表面。在下一期文章中，您将更详细地了解那些您将*希望* 发送 XML 的特殊场景（还会看到那些需要接收 XML 的场景）。具体来说，您会在 Ajax 交互的角度上观察 Web 服务 —— 包括专有 Web 服务和 Google 那样的 API。

简而言之，最重要的任务就是思考对于您的应用程序，XML 在什么时候才是有意义的。在很多情况下，如果您的应用程序工作良好，XML 只不过是又一个让您头疼的技术时髦词汇，您应该克制住仅仅为了宣称应用程序中有 XML 而尝试它的冲动。

如果您处于这样一个环境中：服务器向您发送的数据非常有限，或者采用的是奇怪的逗号分隔、竖线分隔的格式，那么 XML 可能会给您带来真正的收益。考虑处理或更改您的服务器端组件，以使它们用更为标准化的方式 —— 使用 XML —— 来返回响应，而不是使用那些健壮性比 XML 差的专用格式。

最重要的是，要认识到，您对 Ajax 相关技术了解的越多，就必须越谨慎地制定决策。编写那些 Web 2.0 应用程序确实很有趣（在后续文章中，您将重返用户界面，看看可以在这里做些什么很酷的东西），但要注意，确保您不是为了向朋友炫耀而在一个可以正常工作的 Web 页面上滥用这些技术。我确信，您能编写出优秀的应用程序，那么就动手去做吧。完成后，再回来阅读下一期文章，学习关于 XML 的更多知识。

第 9 部分：使用 Google Ajax Search API

发出异步请求并不意味着只是与您自己的服务器端程序交互。其实也可以与一些公共 API，例如来自 Google 或 Amazon 的 API 进行通信，从而为 Web 应用程序增加您自己的脚本和服务器端程序所不能提供的更多功能。在本文中，Brett McLaughlin 教您如何向公共 API，例如 Google 提供的 API 发出请求并接收其响应。

到目前为止，这个系列只涉及到客户机 Web 页面向服务器端脚本和程序发出请求的情况。这就是大约 80% 到 90% 的 Ajax 应用程序（使用 `XMLHttpRequest` 对象的异步 Web 应用程序）的工作方式。然而，这种方法有很严重的局限性：您将受到自己才智和编程技能的限制，就算不是这样，最起码也要受到公司团队中的程序员的才智和编程技能的限制。

有时候，您确实想实现一些功能，但是又不具备实现该目标所需的技术知识，几乎总能遇到这种情况。也许您不知道某些语法，也许不知道如何找出适当的算法。还有些时候，您手头上可能没有用于满足需求的数据或资源（无论是人力资源还是数据资源）。在这些情况下，也许您会想："唉，要是我能使用*其他*人的代码该多好啊！"本文就是要解决这个问题。

开放源码脚本和程序

在论述本文的实际内容（在 Web 应用程序中使用公共 API）之前，有必要说一说开放源码脚本和程序。浅显地讲，*开放源码* 是用于描述可以在一定程度上免费在您自己的应用程序中使用和重用的代码的一个术语。相关链接请参阅 参考资料。简言之，您可以获取别人编写的开放源码，然后将其放入自己的环境中，而不必为之付费，也不会受到（很多）限制。

如果使用开放源码，那么有时候需要为应用程序增加额外的文档，或者将您对开放源码程序或脚本作出的更改反馈给社区。不管如何使用这种程序，最终结果就是，您可以使用这么一块代码：该代码是您不必亲自编写的，或者，如果没有大量的帮助和资源的话，就无法编写该代码，而您手头上并没有这些资源。诸如 Apache 之类的项目为利用他人完成的工作提供了方便 -- 不必担心，他们还希望您使用他们的作品呢！

在线文章和教程

如果在 IBM developerWorks 上发表文章，而又不提及 Internet 上的文章、教程、白皮书之类的大量参考资料，这无疑很愚蠢。网上有成百上千份教材，您也许可以发现近千篇关于 Ajax 的文章 -- 在本系列中，我就已经发表了近十篇文章！这些文章大部分都有可用的代码、例子、下载以及其他各种类型的资源。

如果您没有能力编写要使用的服务器端程序或脚本，或者找不到所需的开放源码程序或脚本，那么可以打开 Google 网站，试着输入对要找内容的基本描述。然后再在 developerWorks 网站上执行相同的操作。您常常可以发现所需的代码，甚至是整个脚本，并且还有一些有帮助的注释和关于其工作方式的描述。

回页首

使用公共 API

很多时候，您会遇到非技术问题。您不需要帮助也能编写某个脚本或某段代码，然而，手头上却没有所需

的数据或资源。在这些情况下，即使有了教程或者开放源码脚本，也还需要更多的东西。例如，考虑在 Web 页面上增加一个搜索引擎的情况。这样做的前提是您已经有了要搜索的数据 -- 但是，如果要搜索您公司或组织以外的数据，那么该怎么办呢？

如果不是因为技术上的原因，而是因为数据而受到限制，那么，一个 *公共 API* 也许可以帮助您解决问题。公共 API 允许使用其他人的服务器上的程序并使用其他人的数据。通常，API 本身只定义如何与该程序交互。例如，通过一个用于 Google 搜索引擎的公共 API 可以发出搜索请求，但是实际上是由 Google 的代码搜索 Google 的数据，然后将结果返回给您的程序。您不仅可以利用他人在编写这些程序方面的技能，还可以利用远远超过您自己公司所能支持的数据。

回页首

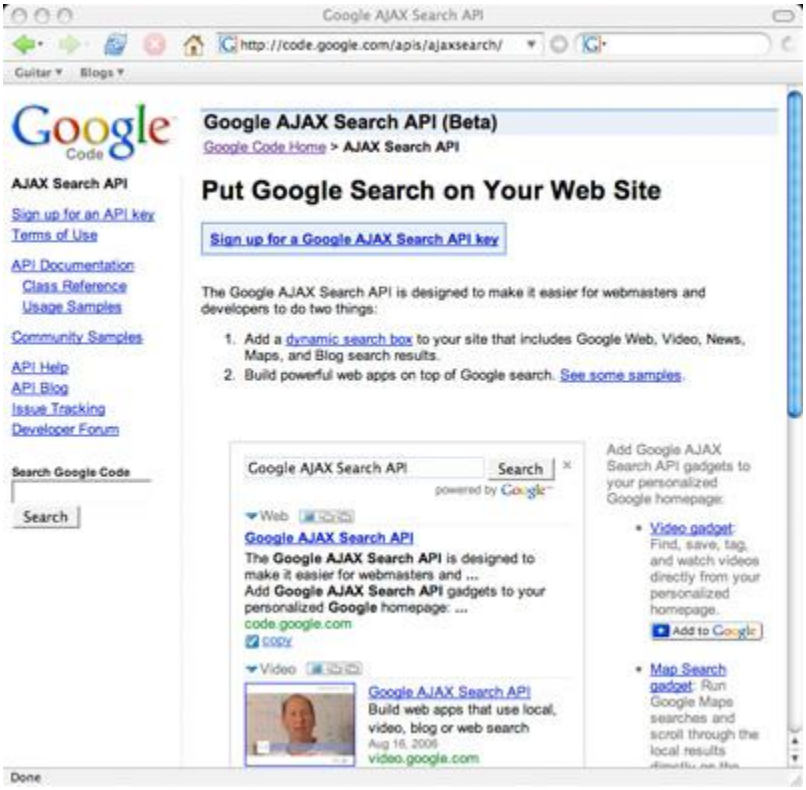
使用 Google Ajax Search API 的准备工作

毋庸置疑，Google 仍然是在线时代极其重要的应用。上至老奶奶，下至四岁小孩，即使不理解网上的其它东西是怎么回事，也一定知道 Google。Google 运行着如此流行、如此有用的搜索引擎，并且致力于提供（几乎全部）免费的服务，所以毫不奇怪，它能提供可以在您自己的程序中使用的公共 API。在本节中，您将完成使用 Google API 的准备工作，并清楚地了解如何与 Google 进行异步应用程序会话。

从 Google 获取开发者密钥

本文着重讨论 Google 的 Ajax Search API。通过访问 Google Ajax Search API 主页（如图 1 所示），可以找到关于这个 API 的更多信息。（该主页的链接见 参考资料。）

图 1. Google 的 Ajax Search API 页面



第一步是单击 **Sign up for a Google AJAX Search API key** 链接。这时会进入另一个页面，在此页面上可以登记使用这个 Google API。您需要接受所有使用条款（我认为所有条款都没有恶意）并提供您的应用程序所在 Web 站点的 URL（如图 2 所示）。

图 2. 登记使用 Google 的 Ajax Search API

阅读完协议并勾选了复选框之后，输入 URL，单击 **Generate API Key**，等待一二秒钟。此时必须登录 Google，或者创建一个帐户。这是一个相当标准的过程，您应该可以自己完成。完成上述操作后，可以看到一个回复页面，其中给出了一个非常长的密钥，并确认您的 URL，甚至还给出一个示例页面。这个密钥看上去类似于以下形式：

应该使用什么 URL？

Google 要求提供的 URL 大致就是站点所在的*域*。如果有自己的域，就像我一样，那么可以使用 http://www.newinstance.com（当然，要用您自己的域替换我的域）。只有当站点使用一个子域，或者一个较大域中的某个特定路径时，才需要指定更详细的 URL。在这种情况下，可能需要使用形如 http://www.earthlink.net/~bmclaugh 或 http://brett.earthlink.net 这样的 URL。但是，除了以上的特殊情况外，不要向 Google 提供过于详细的 URL -- 只需提供用于访问整个 Web 站点的根 URL，就可以在该 URL 内随处使用这个 API。

```
ABQIAAAAjtuhyCXrSHOFSz7zK0f8phSA9fWLQO3TbB2M9BRePlYkXeAu8lHeUgfgRs0eIWUaXg
```

Google 的 API 文档

在开始使用获得的密钥之前，要花点时间阅读一下 Google 的 API 文档（在提供密钥的页面的底端有一个链接，本文的参考资料中也提供了该链接）。即使您通过本文有了很好的初步认识，仍然会发现 Google 的 API 文档是一个很好的参考资料，通过该文档可能会得到关于如何在您自己特有的应用程序中、站点上使用 Google 的一些有趣的想法。

回页首

最简单的 Google 搜索 Web 应用程序

为了看看实际效果，我们以 Google 提供的示例 Web 页面为例，对它稍做修改，然后看看它会变成什么样子。

创建搜索框

清单 1 显示了一个很简单的 Web 页面。将这段代码输入到您喜欢使用的编辑器中，保存为文件，然后将该文件上传到上个小节中提供给 Google 的域或 URL 上。

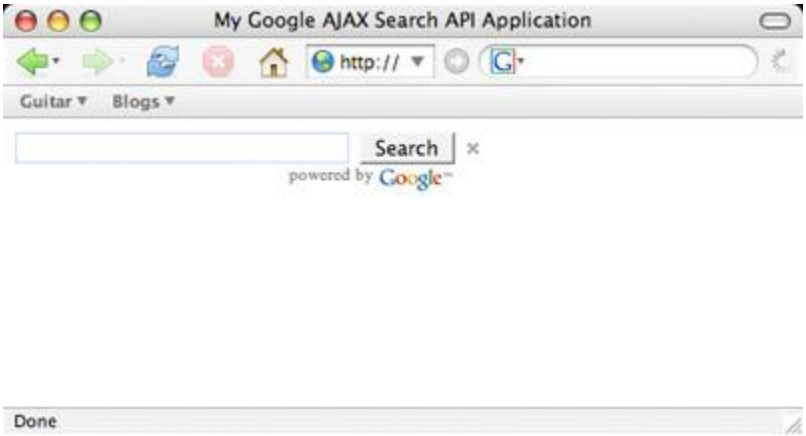
清单 1. 一个简单的 Google 搜索应用程序的 HTML 代码

```
<html>
 <head>
  <title>My Google AJAX Search API Application</title>
  <link href="http://www.google.com/uds/css/gsearch.css"
       type="text/css" rel="stylesheet" />
  <script
       src="http://www.google.com/uds/api?file=uds.js&v=1.0&key=
                                              YOUR KEY HERE
                                    "
       type="text/javascript"> </script>
  <script language="Javascript" type="text/javascript">
    function OnLoad() {
      // Create the Google search control
      var searchControl = new GSearchControl();

      // These allow you to customize what appears in the search results
      var localSearch = new GlocalSearch();
      searchControl.addSearcher(localSearch);
      searchControl.addSearcher(new GwebSearch());
      searchControl.addSearcher(new GvideoSearch());
      searchControl.addSearcher(new GblogSearch());

      // Tell Google your location to base searches around
      localSearch.setCenterPoint("Dallas, TX");
```

```
    // "Draw" the control on the HTML form
    searchControl.draw(document.getElementById("searchcontrol"));
   }
  </script>
 </head>

 <body onload="OnLoad()">
  <div id="searchcontrol" />
 </body>
</html>
```

注意使用从 Google 获得的密钥替换代码中的粗体文本。当装载该页面时，可以看到类似于图 3 的一个页面。

图 3. 最简单的 Google 搜索窗体

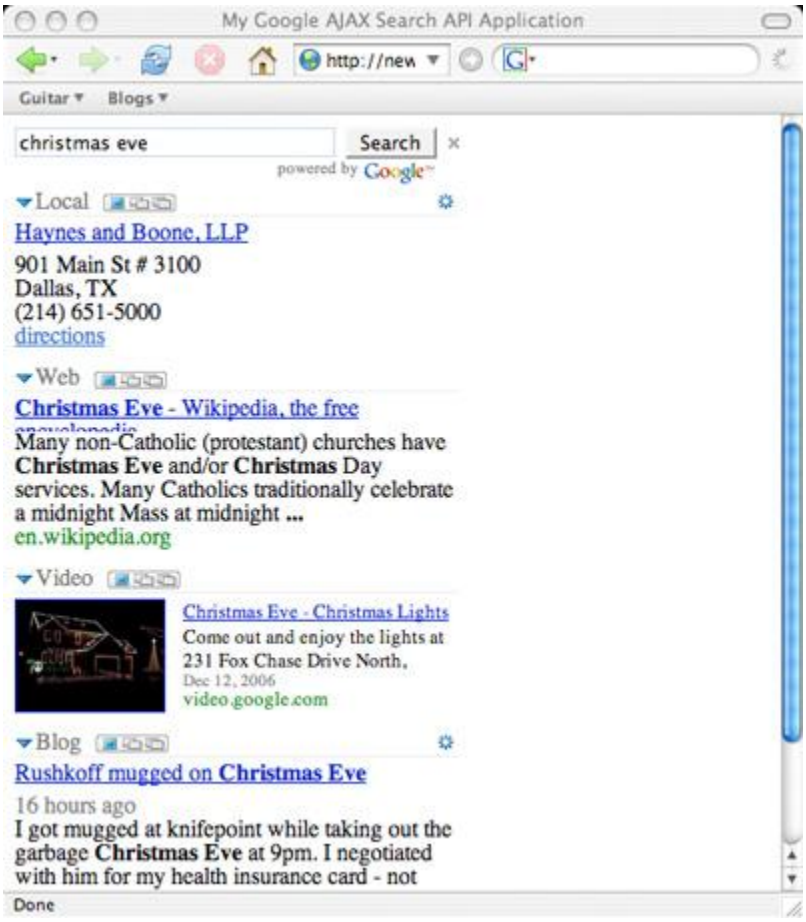


这个页面看上去很简单，但实际上那个小小的控件背后是 Google 的强大搜索能力。

运行搜索

输入一个搜索词并单击 **Search**，使 Google 开始工作。很快可以看到一些搜索结果，如图 4 所示。

图 4. Google 的搜索结果



添加预搜索页面

显然，执行一次搜索之后，页面看上去好多了。视频、博客和搜索结果使页面更加美观。因此，您可能想添加一个*预搜索*，即您定义的一个搜索词，当用户装载您的页面时，首先将看到该搜索词的搜索结果。为此，可以将清单 2 中以粗体显示的那行代码添加到 JavaScript 中。

清单 2. 添加预搜索词

```
function OnLoad() {
  // Create the Google search control
  var searchControl = new GSearchControl();

  // These allow you to customize what appears in the search results
  var localSearch = new GlocalSearch();
  searchControl.addSearcher(localSearch);
  searchControl.addSearcher(new GwebSearch());
  searchControl.addSearcher(new GvideoSearch());
  searchControl.addSearcher(new GblogSearch());

  // Tell Google your location to base searches around
  localSearch.setCenterPoint("Dallas, TX");

  // "Draw" the control on the HTML form
```

```
    searchControl.draw(document.getElementById("searchcontrol"));

    searchControl.execute("Christmas Eve");
  }
```

显然，您可以将自己的初始搜索词加入代码中，以定制页面装载时所显示的内容。

JavaScript 解析

在继续学习之前，简单看一下这些基本命令的作用。首先，创建一个新的 `GSearchControl`，如清单 3 所示。以下结构可用于执行所有搜索任务：

清单 3. 创建新的 GSearchControl

```
function OnLoad() {
  // Create the Google search control
  var searchControl = new GSearchControl();

  ...
}
```

接着，代码使用 `GlocalSearch` 设置一个新的本地搜索；这是特殊的 Google 结构，通过它可以对特定位置执行搜索。这个本地搜索如清单 4 所示。

清单 4. 设置新的本地搜索

```
function OnLoad() {
  // Create the Google search control
  var searchControl = new GSearchControl();

  // These allow you to customize what appears in the search results
  var localSearch = new GlocalSearch();
  ...

  // Tell Google your location to base searches around
  localSearch.setCenterPoint("Dallas, TX");

  ...
}
```

只要知道对象和方法调用，以上代码无需解释。清单 4 中的代码创建一个新的本地搜索器，然后设置搜索的中心位置。

清单 5 中的这几行代码告诉搜索控件应该执行何种类型的搜索。

清单 5. 允许的搜索类型

```
function OnLoad() {
  // Create the Google search control
  var searchControl = new GSearchControl();

  // These allow you to customize what appears in the search results
  var localSearch = new GlocalSearch();
  searchControl.addSearcher(localSearch);
  searchControl.addSearcher(new GwebSearch());
  searchControl.addSearcher(new GvideoSearch());
  searchControl.addSearcher(new GblogSearch());

  // Tell Google your location to base searches around
  localSearch.setCenterPoint("Dallas, TX");

  ...
}
```

其中大部分搜索类型都可以查阅到，以下是一个简短的总结：

- `GwebSearch`：该对象用于搜索 Web，这是 Google 最著名的一种搜索。
- `GvideoSearch`：该对象查找与搜索词相关的视频。
- `GblogSearch`：该对象专门搜索博客，博客的结构和标记与其它 Web 内容类型有所不同。

您已经了解如何预先装载特定的搜索。然后，剩下的只有 `draw()` 方法调用了，如清单 6 所示。您为该方法调用提供了 HTML 中的一个 DOM 元素（如果想回顾关于 DOM 的内容，请参阅 参考资料 中本系列的前几篇文章）。然后，这个控件将魔术般地出现在窗体上，供用户使用。

清单 6. 绘制搜索控件

```
function OnLoad() {
  // Create the Google search control
  var searchControl = new GSearchControl();

  // These allow you to customize what appears in the search results
  var localSearch = new GlocalSearch();
  searchControl.addSearcher(localSearch);
  searchControl.addSearcher(new GwebSearch());
  searchControl.addSearcher(new GvideoSearch());
  searchControl.addSearcher(new GblogSearch());

  // Tell Google your location to base searches around
  localSearch.setCenterPoint("Dallas, TX");
```

```
    // "Draw" the control on the HTML form
    searchControl.draw(document.getElementById("searchcontrol"));

    searchControl.execute("Christmas Eve");
  }
```

回页首

Ajax 在哪里？

到目前为止，还不能明显看出这个简单的搜索框中哪里存在异步性。当然，在 Web 应用程序中某个地方提供一个 Google 搜索框确实很棒，但是这毕竟是关于 Ajax 应用程序的系列文章，而不是关于 Google 搜索的系列文章。那么，Ajax 到底在哪里呢？

输入搜索词并单击 **Search** 按钮，您将注意到一个非常有 Ajax 风格的响应：搜索结果直接显示出来，并没有页面重新装载过程。这正是大多数 Ajax 应用程序的标志之一，即无需重新装载页面，直接显示内容更改。显然，这已经超出了常规请求/响应模型的能力范围。但是，`XMLHttpRequest` 在哪里呢？曾经在那么多文章中风光一时的 `request` 对象如今何在？除了那个 `getElementById()` 方法，DOM 和页面操作又在哪里？实际上，这一切都包含在 HTML 内的两行代码中。

Google 负责处理 JavaScript

第一行要注意的代码尚未多加讨论，该代码如清单 7 所示。

清单 7. 至关重要的 JavaScript 文件

```
<script
    src="http://www.google.com/uds/api?file=uds.js
&v=1.0&key=
[YOUR GOOGLE KEY]"
    type="text/javascript"> </script>
```

这里的语法并不特别值得关注，但要说明的是，Google 存放着一个名为 uds.js 的文件，该文件包含搜索框运行所需的所有 JavaScript。这就是使用他人的代码的最真实的感觉：甚至可以让第三方来存放您的应用程序所使用的代码。这一点非常重要，因为 Google 负责维护工作，当 Google 升级 JavaScript 文件时，您就能自动受益。Google 不会在不通知您的情况下改变 API，所以即使 JavaScript 文件发生了改变，您的代码仍然可以工作。

GSearchControl 对象

另一项比较隐蔽的操作就是在 `onLoad()` JavaScript 函数中创建的 `GSearchControl` 对象。要创建这个对象，只需调用清单 8 中的代码。

清单 8. 创建一个 GSearchControl 对象

```
// Create the Google search control
var searchControl = new GSearchControl();
```

所需的 HTML 代码非常简单：只需使用一个 `div` 标记，以及 JavaScript 可以引用的一个 ID，如清单 9 所示。

清单 9. 用于创建搜索控件的 HTML 代码

```
<div id="searchcontrol" />
```

同样，Google 的代码在幕后处理各种事情。它创建一个新的文本框，一些作为图标的图像，还有一个用于调用某个 JavaScript 函数的按钮。所以，您免费获得了所有行为。虽然您应该理解其中的基本工作原理，但更方便的是，您可以直接使用该代码，然后编写应用程序剩下的部分。

Google 的 JavaScript 是什么样子？

Google 的 JavaScript 并不太容易理解。首先，uds.js JavaScript 文件找到一些本地设置，处理一些 Google 特有的任务，验证您的 Google 密钥，然后装载*另外*两个脚本：http://www.google.com/uds/js/locale/en/uistrings.js 和 http://www.google.com/uds/js/uds_compiled.js。如果有兴趣的话，您可以找出并仔细理解这两个文件，但是要注意：完全理解先进的代码是很困难的，而且其格式令人生畏！对于大部分人来说，只需知道如何使用这些文件，而不必理解其中每一行的意义。

Ajax 不仅仅是您自己编写的代码

Ajax 应用程序不仅仅是指使用 `XmlHttpRequest`；可以说是一种基于异步方式开发 Web 应用程序的方法。即使您没有编写任何特定于 Ajax 的代码，也仍然创建了一个 Ajax 应用程序。多亏了 Google：它做了大部分工作，而您则坐享其成！

回页首

深度探索 Google 的 Ajax Search API

至此，就该由您来完成这些步骤，并应用于您自己的应用程序。最简单的应用是，将一个 `div` 拖入 Web 页面，并将 清单 1 中显示的 JavaScript 添加到 Web 页面中；然后就可以使用 Google 搜索了。

但是，有趣的事情不止于此。不必局限于这组特定的选项或控件。可以围绕 Web 结果、博客结果和视频结果做文章，合适的话，可以将每种结果集成到 Web 应用程序中。例如，可以提供多个搜索控件，每个搜索控件专门用于搜索一种类型的结果。还可以将 Google 搜索控件包括在一个 `span` 元素中，放在其余的应用程序内容的中间，而不是放在侧面的一个 `div` 中。不管那种情况，都应该确信，Google 的搜索是为您的需求而打造的，而不应该修改您自己的应用程序来适应 Google。

回页首

结束语

以本文学到的知识为基础，将 Google 搜索框和其他 Google API 应用到您自己的 Ajax 应用程序中，这应该不难。然而，更重要的是，您应该明白如何使用公共 API。例如，Amazon.com 也提供了一个公共 API，通过它可以对书籍和 Amazon 的其它商品执行同样的 Web 搜索。您可以着手寻找自己喜欢的公共 API，从而超越自己编程技能的限制。实际上，很容易创建一个集成了 Google、Amazon.com、Flickr 等内容的站点。

虽然弄清楚如何使用 Google 比较重要（因为 Google 提供了良好搜索算法和海量的数据存储），但*更*重要的是学习如何使用*任意的*公共 API。还应该开始转变观念，不要再将自己的应用程序看作自我编程技能的总和；相反，它可以是通向各种数据的一个大门。而这些数据可能存储在 Google、Amazon.com、del.icio.us 的服务器上或者其他任何地方。在这些数据的基础上，添加您自己的业务或项目内容，就可以得到非常强大、非常健壮的解决方案，这远远超过您自己编写的作品。

所以，把眼光放远一点，构建*大*应用程序。使用来自各种地方的数据，不要限于自己编写的代码。享受使用他人代码的乐趣，在本系列接下来的文章中，我将谈到更多技术问题，例如数据格式。

第 10 部分: 使用 JSON 进行数据传输

在异步应用程序中发送和接收信息时，可以选择以纯文本和 XML 作为数据格式。掌握 Ajax 的这一期讨论另一种有用的数据格式 JavaScript Object Notation（JSON），以及如何使用它更轻松地在应用程序中移动数据和对象。

如果您阅读了本系列前面的文章，那么应已对数据格式有了相当的认识。前面的文章解释了在许多异步应用程序中如何恰当地使用纯文本和简单的名称/值对。可以将数据组合成下面这样的形式：

```
firstName=Brett&lastName=McLaughlin&email=brett@newInstance.com
```

这样就行了，不需要再做什么了。实际上，Web 老手会意识到通过 GET 请求发送的信息就是采用这种格式。

然后，本系列讨论了 XML。显然，XML 得到了相当多的关注（正面和负面的评价都有），已经在 Ajax 应用程序中广泛使用。关于如何使用 XML 数据格式，可以回顾 本系列前面的文章：

```
<request>
  <firstName>Brett</firstName>
  <lastName>McLaughlin</lastName>
  <email>brett@newInstance.com</email>
</request>
```

这里的数据与前面看到的相同，但是这一次采用 XML 格式。这没什么了不起的；这只是另一种数据格式，使我们能够使用 XML 而不是纯文本和名称/值对。

本文讨论另一种数据格式，*JavaScript Object Notation*（*JSON*）。JSON 看起来既熟悉又陌生。它提供了另一种选择，选择范围更大总是好事情。

选择的意义

在深入研究 JSON 格式的细节之前，您应该了解为什么要用两篇文章讨论另一种数据格式（是的，本系列中的下一篇文章也讨论 JSON），尤其在已经了解了如何使用 XML 和纯文本的名称/值对的情况下。其实，原因很简单：解决任何问题的选择越多，找到问题的*最佳* 解决方案的可能性就越大，这比只能使用*一个* 解决方案要好得多。

回顾名称/值对和 XML

本系列已经用了大量篇幅讨论适合使用名称/值对和 XML 的场合。总是应该首先考虑使用名称/值对。对于大多数异步应用程序中的问题，使用名称/值对几乎总是最简单的解决方案，而且它只需要非常基本的 JavaScript 知识。

实际上，除非有某种限制迫使您转向 XML，否则用不着考虑使用别的数据格式。显然，如果要向需要 XML 格式的输入的服务器端程序发送数据，那么希望使用 XML 作为数据格式。但是，在大多数情况下，对于需要向应用程序发送多段信息的服务器，XML 是更好的选择；换句话说，XML 通常更适合用来*向* Ajax 应用程序做出响应，而不是*从* Ajax 应用程序发出请求。

添加 JSON

在使用名称/值对或 XML 时，实际上是使用 JavaScript 从应用程序中取得数据并将数据转换成另一种数据格式。在这些情况下，JavaScript 在很大程度上作为一种数据操纵语言，用来移动和操纵来自 Web 表单的数据，并将数据转换为一种适合发送给服务器端程序的格式。

但是，有时候 JavaScript 不仅仅作为格式化语言使用。在这些情况下，实际上使用 JavaScript 语言中的对象来表示数据，而不仅是将来自 Web 表单的数据放进请求中。在这些情况下，从 JavaScript 对象中提取数据，然后再将数据放进名称/值对或 XML，就有点儿*多此一举* 了。这时就合适使用 JSON：JSON 允许轻松地将 JavaScript 对象转换成可以随请求发送的数据（同步或异步都可以）。

JSON 并不是某种魔弹；但是，它对于某些非常特殊的情况是很好的选择。不要认为您不会遇到这些情况。阅读本文和下一篇文章来了解 JSON，这样，遇到这类问题时您就知道该怎么办了。

回页首

JSON 基础

简单地说，JSON 可以将 JavaScript 对象中表示的一组数据转换为字符串，然后就可以在函数之间轻松地传递这个字符串，或者在异步应用程序中将字符串从 Web 客户机传递给服务器端程序。这个字符串看起来有点儿古怪（稍后会看到几个示例），但是 JavaScript 很容易解释它，而且 JSON 可以表示比名称/值对更复杂的结构。例如，可以表示数组和复杂的对象，而不仅仅是键和值的简单列表。

简单 JSON 示例

按照最简单的形式，可以用下面这样的 JSON 表示名称/值对：

```
{ "firstName": "Brett" }
```

这个示例非常基本，而且实际上比等效的纯文本名称/值对占用更多的空间：

```
firstName=Brett
```

但是，当将多个名称/值对串在一起时，JSON 就会体现出它的价值了。首先，可以创建包含多个名称/值对的*记录*，比如：

```
{ "firstName": "Brett", "lastName":"McLaughlin", "email": "brett@newInstance.com" }
```

从语法方面来看，这与名称/值对相比并没有很大的优势，但是在这种情况下 JSON 更容易使用，而且可读性更好。例如，它明确地表示以上三个值都是同一记录的一部分；花括号使这些值有了某种联系。

值的数组

当需要表示一组值时，JSON 不但能够提高可读性，而且可以减少复杂性。例如，假设您希望表示一个人名列表。在 XML 中，需要许多开始标记和结束标记；如果使用典型的名称/值对（就像在本系列前面文章中看到的那种名称/值对），那么必须建立一种专有的数据格式，或者将键名称修改为 `person1-firstName` 这样的形式。

如果使用 JSON，就只需将多个带花括号的记录分组在一起：

```
{ "people": [
  { "firstName": "Brett", "lastName":"McLaughlin", "email":
"brett@newInstance.com" },
  { "firstName": "Jason", "lastName":"Hunter", "email": "jason@servlets.com" },
  { "firstName": "Elliotte", "lastName":"Harold", "email": "elharo@macfaq.com" }
]}
```

这不难理解。在这个示例中，只有一个名为 people 的变量，值是包含三个条目的数组，每个条目是一个人的记录，其中包含名、姓和电子邮件地址。上面的示例演示如何用括号将记录组合成一个值。当然，可以使用相同的语法表示多个值（每个值包含多个记录）：

```
{ "programmers": [
  { "firstName": "Brett", "lastName":"McLaughlin", "email":
"brett@newInstance.com" },
  { "firstName": "Jason", "lastName":"Hunter", "email": "jason@servlets.com" },
  { "firstName": "Elliotte", "lastName":"Harold", "email": "elharo@macfaq.com" }
 ],
"authors": [
  { "firstName": "Isaac", "lastName": "Asimov", "genre": "science fiction" },
  { "firstName": "Tad", "lastName": "Williams", "genre": "fantasy" },
  { "firstName": "Frank", "lastName": "Peretti", "genre": "christian fiction" }
 ],
"musicians": [
  { "firstName": "Eric", "lastName": "Clapton", "instrument": "guitar" },
  { "firstName": "Sergei", "lastName": "Rachmaninoff", "instrument": "piano" }
 ]
}
```

这里最值得注意的是，能够表示多个值，每个值进而包含多个值。但是还应该注意，在不同的主条目（programmers、authors 和 musicians）之间，记录中实际的名称/值对可以不一样。JSON 是完全动态的，允许在 JSON 结构的中间改变表示数据的方式。
在处理 JSON 格式的数据时，没有需要遵守的预定义的约束。所以，在同样的数据结构中，可以改变表示数据的方式，甚至可以以不同方式表示同一事物。

回页首

在 JavaScript 中使用 JSON

掌握了 JSON 格式之后，在 JavaScript 中使用它就很简单了。JSON 是 JavaScript 原生格式，这意味着在 JavaScript 中处理 JSON 数据不需要任何特殊的 API 或工具包。

将 JSON 数据赋值给变量

例如，可以创建一个新的 JavaScript 变量，然后将 JSON 格式的数据字符串直接赋值给它：

```
var people =
```

```
 { "programmers": [
   { "firstName": "Brett", "lastName":"McLaughlin", "email":
"brett@newInstance.com" },
   { "firstName": "Jason", "lastName":"Hunter", "email": "jason@servlets.com" },
   { "firstName": "Elliotte", "lastName":"Harold", "email": "elharo@macfaq.com" }
  ],
 "authors": [
   { "firstName": "Isaac", "lastName": "Asimov", "genre": "science fiction" },
   { "firstName": "Tad", "lastName": "Williams", "genre": "fantasy" },
   { "firstName": "Frank", "lastName": "Peretti", "genre": "christian fiction" }
  ],
 "musicians": [
   { "firstName": "Eric", "lastName": "Clapton", "instrument": "guitar" },
   { "firstName": "Sergei", "lastName": "Rachmaninoff", "instrument": "piano" }
  ]
 }
```

这非常简单；现在 `people` 包含前面看到的 JSON 格式的数据。但是，这还不够，因为访问数据的方式似乎还不明显。

访问数据

尽管看起来不明显，但是上面的长字符串实际上只是一个数组；将这个数组放进 JavaScript 变量之后，就可以很轻松地访问它。实际上，只需用点号表示法来表示数组元素。所以，要想访问 programmers 列表的第一个条目的姓氏，只需在 JavaScript 中使用下面这样的代码：

```
people.programmers[0].lastName;
```

注意，数组索引是从零开始的。所以，这行代码首先访问 `people` 变量中的数据；然后移动到称为 `programmers` 的条目，再移动到第一个记录（`[0]`）；最后，访问 `lastName` 键的值。结果是字符串值“McLaughlin”。

下面是使用同一变量的几个示例。

```
people.authors[1].genre                      // Value is "fantasy"

people.musicians[3].lastName                 // Undefined. This refers to the fourth
entry,
 and there isn't one

people.programmers.[2].firstName    // Value is "Elliotte"
```

利用这样的语法，可以处理任何 JSON 格式的数据，而不需要使用任何额外的 JavaScript 工具包或 API。

修改 JSON 数据

正如可以用点号和括号访问数据，也可以按照同样的方式轻松地修改数据：

```
people.musicians[1].lastName = "Rachmaninov";
```

在将字符串转换为 JavaScript 对象之后，就可以像这样修改变量中的数据。

转换回字符串

当然，如果不能轻松地将对象转换回本文提到的文本格式，那么所有数据修改都没有太大的价值。在 JavaScript 中这种转换也很简单：

```
String newJSONtext = people.toJSONString();
```

这样就行了！现在就获得了一个可以在任何地方使用的文本字符串，例如，可以将它用作 Ajax 应用程序中的请求字符串。

更重要的是，可以将*任何* JavaScript 对象转换为 JSON 文本。并非只能处理原来用 JSON 字符串赋值的变量。为了对名为 `myObject` 的对象进行转换，只需执行相同形式的命令：

```
String myObjectInJSON = myObject.toJSONString();
```

这就是 JSON 与本系列讨论的其他数据格式之间最大的差异。如果使用 JSON，只需调用一个简单的函数，就可以获得经过格式化的数据，可以直接使用了。对于其他数据格式，需要在原始数据和格式化数据之间进行转换。即使使用 Document Object Model 这样的 API（提供了将自己的数据结构转换为文本的函数），也需要学习这个 API 并使用 API 的对象，而不是使用原生的 JavaScript 对象和语法。

最终结论是，如果要处理大量 JavaScript 对象，那么 JSON 几乎肯定是一个好选择，这样就可以轻松地将数据转换为可以在请求中发送给服务器端程序的格式。

回页首

结束语

本系列已经用大量时间讨论了数据格式，这主要是因为几乎所有异步应用程序最终都要处理数据。如果掌握了发送和接收所有类型的数据的各种工具和技术，并按照最适合每种数据类型的方式使用它们，那么就能够更精通 Ajax。在掌握 XML 和纯文本的基础上，再掌握 JSON，这样就能够在 JavaScript 中处理更复杂的数据结构。

本系列中的下一篇文章将讨论发送数据以外的问题，深入介绍服务器端程序如何接收和处理 JSON 格式的数据。还要讨论服务器端程序如何跨脚本和服务器端组件以 JSON 格式发送回数据，这样就可以将 XML、纯文本和 JSON 请求和响应混合在一起。这可以提供很大的灵活性，可以按照几乎任何组合结合使用所有这些工具。