

AJAX——新手快车道

前言

AJAX 是什么？

首先、AJAX 是一种很酷的技术，一旦采用了 AJAX，就能让你的 Web 页面，你的网站，甚至连同你们公司，都变得很酷。在 Web2.0 的时代里，不使用一点 AJAX 技术的网站，就会显得很老土，很落伍。

但是，这样的理解，其实是很肤浅的。仅仅是从一个外行，从一个使用者的角度出发，来理解 AJAX，就像我在本书的第一章 AJAX 我也行中那样，开发出很愚蠢，甚至都没有资格被称之为 AJAX 应用的纯 IE、XMLHTTP 应用。

AJAX 更酷的一点在于，对于传统的 Web 开发人员来说，AJAX 所运用的，是更加先进的，更加标准化的，更加和谐高效的，完整的 Web 开发技术体系。遵循这样的体系开发 Web 应用，能让你的开发过程变得更加轻松，也能使你们的开发团队，显得很酷。在 Web2.0 的时代里，还在采用过时的技术来开发 Web，会显得很老土，很落伍。

AJAX 的相关组成技术，每一个都已经出现了 N 年以上了，对这些技术的组合运用，也远远早于 AJAX 这个名词出现之前。所以，我真正敬佩的，并非提出

AJAX 这个缩写的 Jesse James Garrett。而是那些早在 N 年以前，就已经在探索、实践的先行者，他们始终在追求的：**是更好的用户体验，以及更好的开发体验。**这样的精神，才是最可宝贵的，也是最值得我们学习的。许多年过去以后，当我们再回头来看当年的这些热门技术，也许早已经变得老土，变得落伍了。在这样的历程中，哪些人会成长为高手？会成长为大师呢？就是那些永不满足，**永远**

在追求更好的用户体验，永远在追求更好的开发体验的人！

新手如何上路

软件开发这个领域，永远都在飞速发展，大家都必须不断的学习新的知识、技能、框架、IDE、甚至新的语言。传说中的骨灰级高手们，就像传说中的大侠，任何武器、哪怕是一块木头到了他们手里，也能发挥惊人的威力，人家练了几十年的看家本领，他们随手使来，也竟然像是打娘胎里就开始练了一样。为什么？

就算不吹那么玄的，平常我们能够碰到的那些老手，在学新东西的时候，也比那些新手学得更快，理解得更深，运用得更熟练。而新手们呢？往往就会漫无头绪，焦头烂额，以一副张着茫然的大眼睛的经典表情，出现在各大论坛的新手求助区里。他们欠缺的，究竟是什么呢？为什么老手学新东西，就没遇到那么多困难呢？

泛泛地说，自然是经验上的欠缺。仔细地说来，又可以分为三个方面：

一、本质，一种技术与另一种技术之间，往往会有本质上的相通之处，当你以一种技术的理解与思考越来越深入时，学习一种新技术也会更加容易。触类旁通，举一反三的能力，就是来自于对于技术本质的追寻。

二、地图，本质上或多或少的相通，也提示着我们技术之间的相互关联，当你了解的技术越多，了解得越是深入，在你的内心，就能建立起越发清晰的**技术地图**。各种知识都有一个自然、合理的位置。那么当一个老手要学习一门新技术的时候，他其实并非在探索一个全新的、未知的领域，而是有很多脉络可寻，也很多已知可以帮助他们快速了解未知。

三、技巧，面对同样的未知，面对同样的难题，新手们一筹莫展，而老手们却掌握着更多的技巧和手段，帮助他们试探可能性、缩小问题的范围、迅速定位问题、不犯明显愚蠢的错误、甚至能够列举出更具命中力的搜索关键词，而这些技巧，都帮助老手在前进的道路上，更少跌倒，即使跌倒，也能更快的爬起来。

作为一本写给新手的入门书籍，我们希望展现给读者的，是一个老手如何学习新技术的过程。我们相信，这样的一个学习过程，对于新手来说，是更具有价值的。

何谓快车道

必须老老实实的承认，我吹牛了！老手虽然会比新手学习得更快一些，但是也同样会碰到麻烦，遇到障碍，感觉头痛。如果没有真正的专家的指导，我不

可能如此迅速地将 AJAX 掌握到目前这样的程度，要真是让我自学三个月，然后就写出书来的话，那真是在骗钱了。

老手能够快速学习的另一个重要的诀窍是：认识很多牛人朋友

如果没有李锐与赵泽欣的专家级指导与帮助，如果没有与李锐 AJAX 结对编程的体验，如果没有三个人在 MSN 上无数次的长聊，我想要在短期内建立起：

对于 AJAX **本质**的理解；

对于整个 AJAX 以及相关**技术地图**的理解；

对于 AJAX 编程开发所需要的很多**技巧、手段**的掌握；
几乎是不可能的。

如果没有（N 多需要感谢的人）的（N 多方面的帮助），我们这本书，也不可能以现在这样的深度，以（N 个月）内完成的速度，送到读者的面前。

希望这本书，能够对大家快速学习 AJAX，有所帮助。

一 AJAX 我也行

1.0 缘起

我是一个勤奋的 Blogger，坚持不懈的写 Blog，让我有了很多的收获。比如天南地北的朋友，比如千奇百怪的朋友，比如志同道合的朋友。不时会有朋友加我的 MSN，也不时会有一些有趣的对话、甚至诡异的对话。

但是，最为诡异的对话，是发生在 2005 年 11 月的某一天。一个叫 Yeka 的朋友在 MSN 上跟我 hi 了一下。

Yeka: hi, 你好，我是博文视点的。

我: 你好，什么事呀

Yeka: 想请你写一本书，看看你有没有兴趣。

我: 🤔

Yeka: 有朋友向我们推荐你的，我们也觉得你的文笔不错。

我: 是什么书呀? (内心颇有些得意啊 😊)

Yeka: 一本关于 AJAX 的新手入门的书。

我: 🤔🤔🤔 (我的朋友，难道这个推荐我的朋友，不知道我不懂 AJAX 的吗? 八成是要害我吧!)

我: 是谁呀。

Yeka: 熊节，还有孟岩。

我: …… (无语中……照理说，他们应该了解我的呀，怎么会推荐我来写书呢? 而且还是 AJAX 的书。)

可悲啊，我其实是一个意志不坚定的同志，经不起 Yeka、孟岩他们几个的撺掇，我也就跃跃欲试了。内心里想着，AJAX 嘛，新出来的技术，能有多难，

写一本新手入门嘛，也不用写得太深的。

现在拿在你手里的这本书，就是我的处女作了——应该叫我们，就像你在封面上看到的那样，有三个人，这个故事有点复杂——这是一本关于 AJAX 的书，也是一本关于我如何写出《AJAX——新手快车道》这样一本书的书。对于我来说，这既是一次极限的写作经历，也是一次极限的学习经历。

其中的刺激，让我们一起来领略吧，Let's GO!

1.1 AJAX 是什么呀？

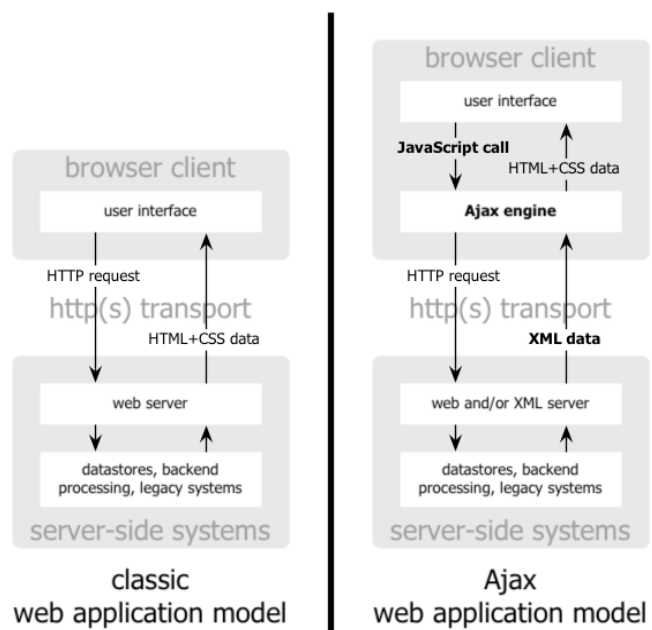
要写一本 AJAX 的书，想想都觉得有点心虚。AJAX 到底是什么呀？我得先 Google 一把。于是我就发现了促使 AJAX 诞生的那篇文章《a New Approach to Web Applications》^①，在这篇文章里，AJAX 是一个比较莫名的缩写单词：Asynchronous JavaScript + XML。什么叫做异步的 JavaScript 加上 XML 呀。在初步的浏览之后，我相信，AJAX 这个名字之所以能够如此风行，完全是因为那只世界闻名的荷兰球队^②。Jesse James Garrett 的贡献的伟大之处就在于，他想方设法的搞出了这么一个缩写，使得一些大家早就在使用的老技术，变得重新流行起来，热门起来，新鲜起来。

在 Garrett 的总结中，AJAX 包含以下五个部分：

- 基于标准的表示技术，使用 XHTML 与 CSS
- 动态显示和交互技术，使用 Document Object Model(文档对象模型)
- 数据互换和操作技术，使用 XML 与 XSLT
- 异步数据获取技术，使用 XMLHttpRequest
- 而 JavaScript 将以上的一切都结合在一起

而借助于这样的技术组合，我们可以开发出用户使用体验更加好的 Web 应用。

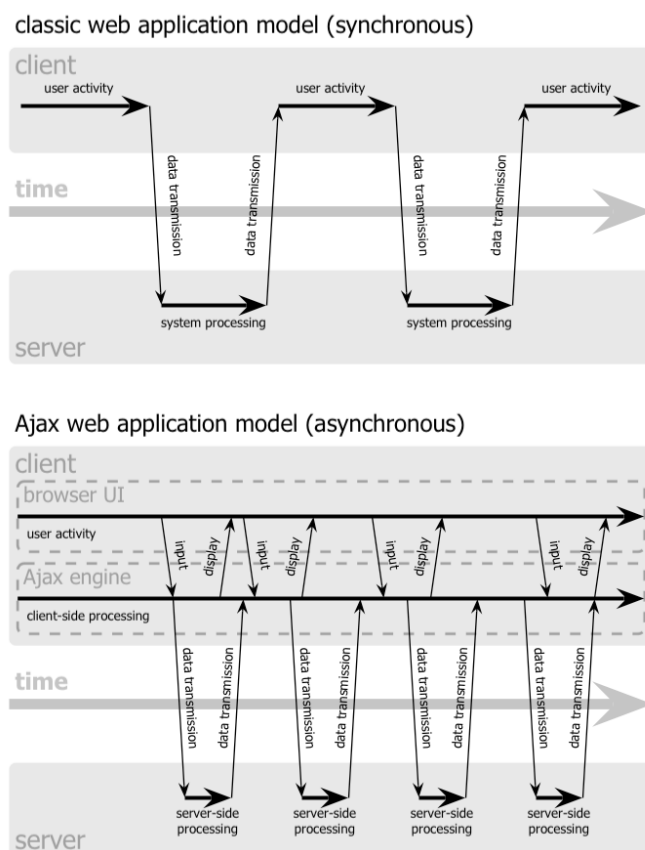
那么，这么好的技术，究竟有哪些东西呢？这些技术和传统的 WEB 开发技术，有多少区别呢？



右边的这两张图，就是那两张著名的 AJAX 介绍的图了。在 Garrett 的文章中，用这两张图，非常直观的表达了新老两种技术之前的区别。

第一张图尤其说明了传统 Web 应用程序的结构与采用了 AJAX 技术的 Web 应用程序的结构上的差别。

主要的差别，其实不是 JavaScript，不是 HTML/XHTML 和 CSS，而是采用了 XMLHttpRequest 来向服务器异步的请求 XML 数据。至于其它的技术，大家都是可以用的。



再来看第二张图，传统的 Web 应用模式，用户的体验是割裂的，点击->等待->看到新的页面->再点击->再等待。而采用了 AJAX 技术之后，大部分的计算工作，都是在用户不察觉的情况下，交由服务器去完成了，没有空白的屏幕，没有让人心烦的旋转地球，没有虚假的下载百分比进度条。用户不必在黑夜与白天之间穿行，只需要的安心的坐在显示器前，欢乐不间断，享受不打折。

真好！也真是简单，说到底，一切的秘密不就是个 XMLHttpRequest 吗？早知道我就不该用 AJAX 这个关键词，现在 Google 都有 1300 多万条 AJAX 相关的页面了。我还是直接用 XMLHttpRequest 这个关键词来查吧。赶紧进入 Coding，才是正道啊！

1.2 我的第一个 AJAX 的 Hello World

Google 果然是好用啊，经过 5 分钟的搜索，再加上 5 分钟的 EditPlus，我鼓捣出了自己的第一个 AJAX 的 Hello World 程序。总共只有 10 行，如果不是为了美观的话，还可以更短。

文件路径：CD-ROM/code/HelloWorld/step1.html

```
1 <SCRIPT LANGUAGE="JavaScript">
2 var xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");
3 xmlHttp.open("GET", "http://www.blogjava.net/zbw25/Rss.aspx", true);
4 xmlHttp.onreadystatechange=function() {
5     if (xmlHttp.readyState==4) {
6         alert(xmlHttp.responseText);
7     }
8 }
9 xmlHttp.send();
10 </SCRIPT>
```

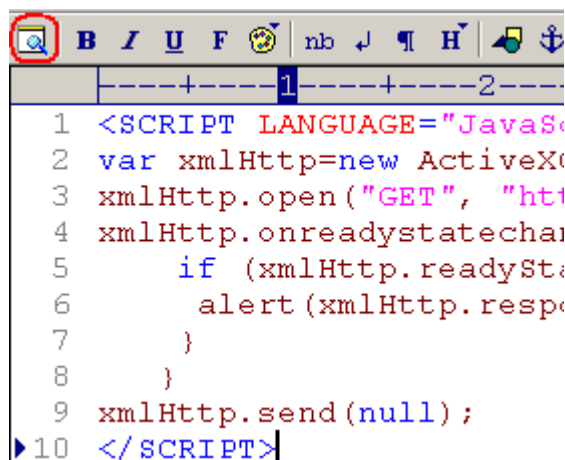
咳，下面就让我来跟大家解释一下吧。

首先，打开各位的 EditPlus，推荐大家使用这个软件，是因为他特别适合用来手写 HTML、JavaScript 代码。新建一个 HTML 文件，然后把里面那些自动生成的代码都删除掉。把上面的这个代码手工的敲进去，这样大有好处。

EditPlus 有一个很方便的地方，

就是我们在编辑静态 HTML 页面的时候，可以随时点击左上角的一个小按钮，在编辑窗口与 EditPlus 内置的 IE 浏览器窗口之间自由切换。

第 1 行，这是一个标签，用在 HTML 里面，表示开始一段 JavaScript。和第 10 行的</Script>相呼应，其中包含的就是 JavaScript 代码了。



第 2 行，这和 Java 中的 new 一个新对象非常的类似，区别在于，Java 是强类型语言，因此必须写成：

```
var xmlhttp=new ActiveXObject("Msxml2.XMLHTTP");
```

而 JavaScript 是动态类型的脚本语言，任何对象只要一个 var，就可以了。

这样，xmlhttp 就得到了一个浏览器内置的 XMLHttpRequest 对象的实例了。

(注，参见第二章 P30)

第 3 行，调用 XMLHttpRequest 的 open 方法。第一个参数，表示以 GET 的方式向服务器提交我们的 HTTP 请求，当然也可以选择 POST 的方式。HTTP 请求，常用的就是这两种方式了（注，共有 7 种，另外五种是 HEAD、PUT、DELETE、OPTIONS、TRACE）。第二个参数是一个 URL，表示我们打算 GET 的内容所谓的网络地址。第三个参数就神奇了，true 就代表着，我们打算提交的是一个异步的 XMLHttpRequest 请求。至于什么是异步、什么是同步，咱们后面再详细讨论。

因为是异步提交请求，所以，当浏览器执行到这一行的时候，其实并没有干什么事。要等到执行到第 9 行，调用 XMLHttpRequest 的 send() 方法的时候，才会真正的发出 XMLHttpRequest 请求。

第 4、5、6、7、8 行，其实只干了一件事情，就是指定了一个事件处理的函数句柄。这里需要解释几个概念：

对象的状态：当我们创建了一个 XMLHttpRequest 对象以后，这个对象是处于原始状态的。这时候如果我们访问它的 readyState 属性，得到的值应该是 0，称为**未初始化状态**。readyState 一共有 5 个可能的值，我们还是用一个表格来说明吧。

readyState	状态	解释
0	未初始化	对象已创建，尚未调用 open()
1	已初始化	调用 open() 方法以后
2	发送数据	调用 send() 方法以后
3	数据传送中	在已经接到部分数据，但尚未接收完成

事件：每当 `readyState` 的值发生改变时，就是产生了一个事件。

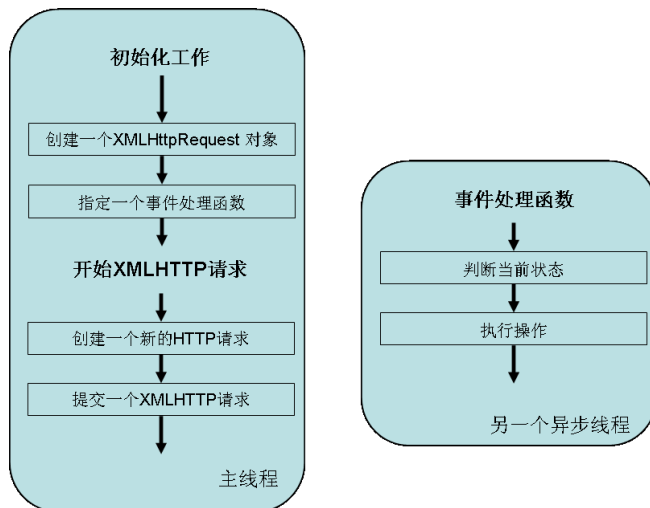
事件处理的函数句柄：我们可以选择在事件产生以后，如何处理。这个处理的函数，我们必须告诉这个 `XMLHttpRequest` 对象。这里借用了 C 语言中的函数句柄的概念。将这个函数的句柄，在 JavaScript 中也就是这个函数的名称，赋值给 `XMLHttpRequest` 对象中的 `onreadystatechange`，就可以了。（注：这里其实是创建了一个匿名的函数，`onreadystatechange` 只是属性名称，而不是函数名称。函数在 JavaScript 中是第一等的对象，可以赋值给任何变量。Java 开发人员尤其需要注意！）

再来看我们的这个程序的第 4 行，我还有一个概念没有解释，JavaScript 中，还支持匿名函数这样的东西。只要将一个 `function` 的定义，作为一个整句赋值给 `onreadystatechange`，就行了。

最后我们来看第 6 行，这是我们的这个 AJAX 程序，真正能够体现出我干成了！这一信息的关键一句。一个 `alert()`。呵呵，当程序判断 `readyState` 已经等于 4，说明数据已经全部读到，再将这个数据通过一个对话框弹出来，就大功告成了。

现在，我们可以点击 `EditPlus` 的那个窗口按钮，然后稍等片刻……我们的程序正在后台工作……然后就会有一个对话框弹出来，哈哈，第一个 AJAX 程序顺利执行了！

下面我们来总结一下 `XMLHttpRequest` 在异步请求远程数据时的工作流程吧：



1.3 正确的工作

说实话，能够弹出一个 alert 框来，实在是不算什么本事。第一个我可以想到的改进是：

1、增加一个第 11 行：

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

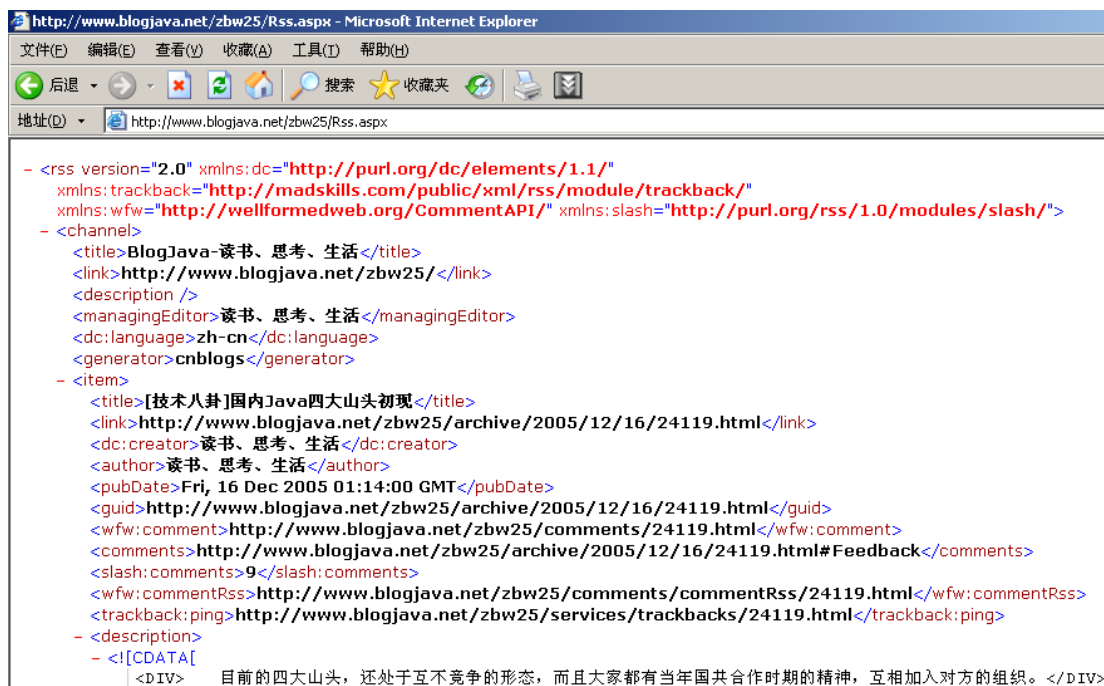
2、修改一下第 6 行：

```
content.innerHTML+xmlHttp.responseText;
完整的代码，在 CD-ROM/code/HelloWorld/step2.html
```

我们再点击一下 EditPlus 的窗口键，看看显示出来的效果。

好难看啊！一堆乱七八糟的英文、数字、HTML 标签、根本就是无法阅读啊。为什么会这样呢？

我们还是先来看看，这个 URL 的内容，究竟是些什么吧？我们直接在浏览器的地址栏里，输入这个 URL：<http://www.blogjava.net/zbw25/Rss.aspx>，回车以后，应该能看到类似于下面这样的画面：



```
- <rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:trackback="http://madskills.com/public/xml/rss/module/trackback/"
  xmlns:wfw="http://wellformedweb.org/CommentAPI/" xmlns:slash="http://purl.org/rss/1.0/modules/slash/">
- <channel>
  <title>BlogJava-读书、思考、生活</title>
  <link>http://www.blogjava.net/zbw25/</link>
  <description />
  <managingEditor>读书、思考、生活</managingEditor>
  <dc:language>zh-cn</dc:language>
  <generator>cnblogs</generator>
- <item>
  <title>[技术八卦]国内Java四大山头初现</title>
  <link>http://www.blogjava.net/zbw25/archive/2005/12/16/24119.html</link>
  <dc:creator>读书、思考、生活</dc:creator>
  <author>读书、思考、生活</author>
  <pubDate>Fri, 16 Dec 2005 01:14:00 GMT</pubDate>
  <guid>http://www.blogjava.net/zbw25/archive/2005/12/16/24119.html</guid>
  <wfw:comment>http://www.blogjava.net/zbw25/comments/24119.html</wfw:comment>
  <comments>http://www.blogjava.net/zbw25/archive/2005/12/16/24119.html#Feedback</comments>
  <slash:comments>9</slash:comments>
  <wfw:commentRss>http://www.blogjava.net/zbw25/comments/commentRss/24119.html</wfw:commentRss>
  <trackback:ping>http://www.blogjava.net/zbw25/services/trackbacks/24119.html</trackback:ping>
- <description>
  - <![CDATA[
    |<DIV> 目前的四大山头，还处于互不竞争的形态，而且大家都有当年国共合作时期的精神，互相加入对方的组织。</DIV>
```

这就是——大名鼎鼎的 RSS 了！所谓 RSS，简单的说，就是一种以 XML 格式描述的数据，用于共享新闻、Blog 等等 Web 内容。RSS 是以下三组单词的缩写，Rich Site Summary(丰富站点摘要)或 Really Simple Syndication(真正简单的聚合)。也可以是 RDF Site Summary(RDF 站点摘要)，RDF 又是 Resource Description Framework 的缩写。由此可见，老外们是多么的喜欢玩文字游戏啊。

现在我们访问的这个 RSS 呢，是 RSS2.0 版本的数据格式，至于详细的 RSS 的版本问题，这里就不深入讨论了，只来看看这 2.0 的 RSS，究竟是怎样一个格式。简单的介绍一些我们用得着的元素如下：

- rss:每个 RSS Feed，都有而且只能有一个 rss 标签，作为顶层元素
 - channel:在 rss 标签下，必须有且只能有一个 channel 标签
 - title:这个 blog 的标题
 - link:blog 的 URL 访问地址
 - description:简要描述
 - managingEditor:blog 的作者
 - item:可以出现多个 item，每个 item，描述一条日志信息
 - title:日志的标题
 - link:日志的 URL 访问地址
 - author:日志的作者

- pubDate:日志的发布日期
- description:日志的内容

要分析这样一种给定格式的 XML，自己去分析字符串自然是下下策。而 XMLHttpRequest 对象，对于 XML 的分析，正好能够得心应手。所以，我们接下来，就要看看，XMLHttpRequest 能够如何来帮助我们，简化劳动。

文件路径：CD-ROM/code/HelloWorld/step3.html

```

1 <SCRIPT LANGUAGE="JavaScript">
2
3 var bloglist=new Array();
4 var xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");
5 function startload(){
6     xmlHttp.open("GET", "http://www.blogjava.net/zbw25/Rss.aspx", true);
7     xmlHttp.onreadystatechange=function() {
8         if (xmlHttp.readyState==4&&xmlHttp.status==200) {
9             var xmlDom=xmlHttp.responseXML;
10            var channel=xmlDom.documentElement.childNodes.item(0);
11            var itemlist=channel.childNodes;
12            for(var i=0;i<itemlist.length;i++){
13                var node=itemlist.item(i);
14                if(node.tagName=="title"){
15                    blogtitle.innerHTML=node.text;
16                }
17                if(node.tagName=="link"){
18                    bloglink.href=node.text;
19                }
20                if(node.tagName=="description"){
21                    blogdescription.innerHTML=node.text;
22                }
23                if(node.tagName=="managingEditor"){
24                    managingEditor.innerHTML=node.text;
25                }
26                if(node.tagName=="item"){
27                    bloglist[bloglist.length]=node.childNodes;
28                }
29            }
30            showitem(bloglist[0]);
31        }
32    }
33    xmlHttp.send(null);

```

```

34 }
35
36 function showitem(itemlist){
37     for(var i=0;i<itemlist.length;i++){
38         var node=itemlist.item(i);
39         if(node.tagName=="title"){
40             itemtitle.innerHTML=node.text;
41         }
42         if(node.tagName=="link"){
43             itemlink.href=node.text;
44         }
45         if(node.tagName=="description"){
46             description.innerHTML=node.text;
47         }
48         if(node.tagName=="author"){
49             author.innerHTML=node.text;
50         }
51         if(node.tagName=="pubDate"){
52             pubDate.innerHTML=node.text;
53         }
54     }
55 }
56 </SCRIPT>
57 <body onload="startload()">
58 <font color=blue><a id="bloglink" href="#"><div id="blogtitle"></div></a></font>
59 <br>
60 <font color=blue><div id="managingEditor"></div></font><br>
61 <font color=blue><div id="blogdescription"></div></font><br>
62 <font color=#000066><a id="itemlink" href="#"><div id="itemtitle"></div></a></font>
63 <br>
64 <font color=blue><div id="author"></div></font><br>
65 <font color=blue><div id="description"></div></font><br>
66 <font color=blue><div id="pubDate" align="right"></div></font><br>
67 </body>

```

这个代码就比较长了，如果不愿意自己手工录入的朋友呢，可以直接 Copy，虽然我始终建议，学编程，最好是能够手工的键入很多代码，特别是在学习的初级阶段，每一个字母都自己敲进去，体会到那样的繁琐，才会有动力不断重构代码，减少自己低级劳动的比例。也只有这样，你对于一种技术、一种语言、一种算法，才会有更加真切的体会。那些看编程书一目十行的初学者，那些连书中的 demo 都不准备在自己的机器上跑出来的初学者，基本上就很难有

什么提高了。(当然，那些能够在自己的脑子里跑代码的天才，不在此列)

我们还是来解释一下这个程序吧，已经不算短了，不过要紧的知识点还不算太多。

xmlHttp.status==200

第八行的程序，与 step1、step2 的有所不同，以后都推荐像这样写。因为 xmlHttp.status==200，代表服务器端返回的是正确的结果，这样才有可能正确的解析 XML。

responseXML

第 9 行的程序是这样的：var xmldom=xmlHttp.responseXML;

一个 XMLHttpRequest，在顺利读取了数据之后，我们其实可以有四种方法，得到这个数据。responseText 只是其中一种。另外还有三种：

responseText	将响应信息作为字符串返回
responseXML	将响应信息格式化为 XMLDOM 对象并返回
responseBody	将响应信息以 unsigned byte 数组的形式返回
responseStream	将响应信息以 IStream 对象的形式返回

其中我们需要深入了解的，只有 responseText 和 responseXML 两种，至于 responseBody 和 responseStream，基本上可以不用考虑。(注 1)

前面我们已经使用了 responseText，这个实在是很简单，就是一个 String，想怎么处理就怎么处理。(注 2)

另外一个就是我们这回用到的，返回 XML DOM 的方法。对于所谓 AJAX 应用，其实主要的工作，就是在浏览器客户端与服务器端之间，传递各种 XML 数据。如果要自己读取 String 数据，然后再全部自己手工解析 XML，那就会非常复杂；而使用 responseXML，就可以直接得到一个已经解析好了的 XML DOM，这样就方便了。

XMLDOM 相关 API

就算是直接得到了 XMLDOM，事情也还没有结束。我们还需要运用 XMLDOM 提供的 API，来正确的读取其中的数据。XMLDOM 实在是太多内容了，咱们先只介绍这段程序里用到的 API 吧。

XMLDOM 树：一个 XMLDOM，其实是一个内存中的树，一个 XML 数据中的每一个 `<>` 中的第一个单词，就代表着一个节点。比如：

```
<rss>
  <channel/>
</rss>
```

`<channel />`，就代表 `channel` 节点。如果 `channel` 这个节点，直接以 `/>` 结尾，那么它就不包含子节点。而 `rss` 节点，就是一个包含子节点 `channel` 的节点。我们也可以说，`rss` 是 `channel` 的父节点，`channel` 就是 `rss` 的子节点。根节点，就是没有父节点的节点，也就是顶层节点。

documentElement：这样就能够得到一个 XML DOM 树的根节点，具体到我们想要分析的 RSS 文档，就是 `rss` 那个节点。

childNodes 与 item：这样就能够得到一个节点的所有子节点，以一个数组的形式返回。

```
1 var childs=node.childNodes;
2 alert(childs.length);
3 var itemnode=childs.item(0);
```

上面的 `childs` 就是一个 JavaScript 的数组，我们可以通过 `childs.length`，得到这个数组的长度。也可以通过 `item(int)`，得到其中的某一个元素，这个元素，自然也是一个节点。

tagName：可以获得一个节点的名称。`<rss>` 的 `tagName` 自然就是 `rss`，`<title>` 的 `tagName`，自然就是 `title`。我们的程序就是通过判断 `tagName`，来识别信息的。

text：就是一个节点包含的内容。比如 `< author>读书、思考、生活</ author>`。那么这个节点的 `text`，就是 读书、思考、生活。

OK，就说这么多，XML DOM API 当然不会这么简单，不过在现在这个程序里，我们用到的也就只有这么多。

innerHTML 与 href

比如程序的第 15 行：`blogtitle.innerHTML=node.text;`

第 18 行：`bloglink.href=node.text;`

其实我们可以发现 innerHTML 与 XML DOM 中的 `node.text`，有某种相似之处。在初始的页面上，我们写着：

```
<font color=blue><a id="bloglink" href="#"><div id="blogtitle"></div></a></font>
```

这样的内容，在静态的 HTML 页面里，其实什么都看不到。但是有了 ``和`<div id="blogtitle">`这样的两个元素，我们就可以借助 JavaScript 对于 HTML DOM 的访问能力，动态的在页面里显示出内容来。

```
blogtitle.innerHTML="test";  
就相当于  
<div id="blogtitle">test</div>  
而  
bloglink.href="url";  
就相当于  
<a id="bloglink" href="url">
```

这就是初步的动态 HTML(DHTML)了。

onload 事件

最后需要讨论的是第 57 行：`<body onload="startload()">`

这是由于我发现了自己的一个 bug，一开始我的写法还是和 step1、step2 一样的，直接执行 JavaScript。当我们第一次打开这个 step3.html 的时候，一切正常，而当我们再次刷新这个页面的时候，浏览器却报告说 ‘某某’ 未定义。究其原因，还是因为 JavaScript 代码写在 HTML 代码前面，当他们执行的时候，HTML

DOM 可能尚未创建导致的。因此，改成 onload 事件触发以后，也就是整个页面装载完成之后，再执行 JavaScript，就每次都能正常工作了。比较不能让人理解的是，为什么 IE 在第一次，却能正确执行呢？（注，参见第二章 DOM 部分）

最后的工作，检查了一下代码，发现重复的部分实在是太多了，必须重构一下。写这么一个函数，来做一些重复劳动。

```
1 function getString(itemlist, tag) {
2     for(var i=0; i<itemlist.length; i++) {
3         var node=itemlist.item(i);
4         if(node.tagName==tag) {
5             return node.text;
6         }
7     }
8     return "";
9 }
```

重构后的完整代码，在 CD-ROM/code/HelloWorld/step4.html

1.4 漂亮的工作

接下来我们就可以乘胜追击了。让我们来规划一个简单的 RSS 阅读器，采用最先进的 AJAX 技术，就在一个页面里，读取各种各样的 RSS。界面我们还得弄的漂亮点，还要能够显示多个不同的 RSS。咱们就模仿一下 bloglines，但是要比他做得方便，不用刷新页面的。

我们先参照他的样子，设计个界面：



这个界面的元素也相当简单，就是对 Bloglines 做很多的减法，只保留基础

的功能：

- 1、左边一个 Frame，用来显示我订阅的 RSS Feeds 列表。
- 2、在左边 Frame 的最上方，要有一个添加 RSS 的按钮，我们不需要像 Bloglines 那种智能的添加，而只接受 RSS 2.0 格式的 URL。
- 3、右边是一个 Blog 的浏览页面，右上角是退订和修改 URL 的两个 link
- 4、显示的 BLOG，有几个元素：
 - a) Blog 名称
 - b) Blog 描述
 - c) Blog 图片(如果有的话)
 - d) 显示的 Item 的条数
 - e) 更新时间
 - f) Item 的标题
 - g) Item 的正文
 - h) Item 发表时间
 - i) 选择显示的范围

Step5 的成果：

在努力工作了 4 个小时之后，我完成了 step5 的所有文件。包括 3 个 html 和 4 个 jpg 图片，还有从 bloglines 搞到的两张 gif。我又不想使用 Fireworks 那种大型专业工具，还好我找到了一个屏幕取颜色的小工具（color.exe），所以颜色肯定是正确的，但是我还是没有找到能够屏幕取文字的字体和大小的工具，所以字体上与 bloglines 肯定是不一样的。唉，Web Build，真是辛苦啊！

文件路径：CD-ROM/code/HelloWorld/step5/

到目前为止，step5，就是一个静态的页面，所有的动态的内容都还没有添加进去。OK，下面我们来编程吧！

1、全局变量

首先我们需要考虑的，是数据保存在哪里？既然添加 blog 的 URL 的 link，是放在 tree 页面的，很合理的一个想法就是，将所有通过网络取得的 RSS 数据，也都放在 tree 页面的某个全局变量里。

```
4 var xmlhttp=new ActiveXObject("Msxml2.XMLHTTP");  
5 var blogList=new Array();  
6 var currentBlog=new Array();
```

xmlhttp: 这个就不用解释了吧，要读取数据，都得靠它了。

blogList: 所有我们能够读到的 blog 的 RSS 数据，在解析以后，就都放在这里了。

currentBlog: 这是给 base 页面使用的，当我们点击了某个 Blog 的标题之后，在右边的页面里将会显示出内容，根据就是这个 currentBlog。

2、数据结构

一个 blogList，是一个数组，每一个元素，就是一个 Blog 的信息。这些信息又是存放在一个数组里的。

blogtitle:Blog 的标题

bloglink:URL

blogdescription:Blog 的简要描述

managingEditor:作者

pubDate:发布时间

blogitemlist:日志列表，又一个数组

image:Blog 的 Logo 图片信息，

然后，在 blogitemlist 中的每一个 item，以及 image，又都是一个 XMLDOM

中的 NodeList 对象，可以通过 getString() 函数，取得其中的信息。

3、addBlogURL

```
37 function addBlogURL(){
38   var url=prompt("请输入 RSS 的地址: ");
39   loadRSS(url);
40 }
```

这里其实是有问题的，在这个最简单的 addBlogURL 函数里，我们没有对 URL 进行任何的校验，无论是否正确，都直接就交给 loadRSS 函数去处理了。再者，我们也没有判断，用户是否重复输入了 URL，也就有可能出现重复的 Blog。当然，这两件事，我们也可以交给 loadRSS 来做，不过，更加合理的做法应该是：

```
37 function addBlogURL(){
38   var url=prompt("请输入 RSS 的地址: ");
39   var errmsg=verifyURL(url);
40   if(errmsg==""){
41     loadRSS(url);
42   } else {
43     alert(errmsg);
44   }
45 }
```

loadRSS 就不用解释了，基本上就是 step4 中的 startload，区别在于，startload 在 load 完成之后，就直接设置各个 HTML 元素的 innerHTML 了，而现在我们得将活做得更加漂亮一些，也就是把需要完成的工作分配给多个不同的函数，以分工更加明确的方式来进行。

原来的处理次序是：load->show，因为只有一个 blog，所以也可以放在一个函数里面。

现在的处理复杂了：loadRSS->showBlogList-> showBlogItem，在 loadRSS 中，我们仅仅是将获得的数据，填入一个数据结构 blogList 之中。其余的工作，就交给其它函数来做了。

4、showBlogList 与 showBlogItem

这个函数，主要是做两件事，一个是修改一下 blog 的总数，也就是修改

count 的 innerHTML。还有就是调用 showBlogItemList。这个函数，就是实实在在的脏活、苦活、累活了。

```
13 function showBlogItemList(){
14   var showhtml="";
15   if(blogList.length>0){
16     showhtml="<table border=\"0\" width=\"260\">";
17     for(var i=0;i<blogList.length;i++){
18       showhtml=showhtml+"<tr>";
19       showhtml=showhtml+"<td width=\"20\">&nbsp;</td>";
20       showhtml=showhtml+"<td width=\"24\"><img src=\"blog.gif\"></td>";
21       showhtml=showhtml+"<td><a href=\"#\" onclick=\"showBlogItem(\"+i+\")\"><font size=\"
22 2px\" color=\"000000\" face=\"Verdana\">"+blogList[i].blogtitle+"</font></a></td>";
23       showhtml=showhtml+"</tr>";
24     }
25     showhtml=showhtml+"</table>";
26   }
27   itemlist.innerHTML=showhtml;
```

Itemlist 这个元素，必须是一个 table，其中的

<table><tr><td><a>等林林种种的标签，还有标签里面林林种种的属性，我都得一一给他用字符串正确的拼出来。拼对了，就能漂漂亮亮的显示出来，拼错了，就会很难看。

5、showBlogItem

```
32 function showBlogItem(id){
33   currentBlog=blogList[id];
34   top.baseframe.showBlog();
35 }
```

当用户点击了某个 Blog 的标题，就会触发一个 onclick 事件，然后调用这个 showBlogItem 函数。这个函数又会再调用 base.html 页面里的 showBlog，将所有的内容显示出来。

6、隐藏与显示

在一开始打开 base.html 的时候，我们应该看到一个没有内容的页面，因为那时候，我们还没有订阅任何一个 Blog，只是在 step5 的时候，为了看到显示效果，没有把他们隐藏起来。

要隐藏这个页面，也不难，就是用一个 DIV，框住整个 body 里的内容，然后写成：`<div id="main" style="display:none">`，就看不见了。在需要显示的时候，我们再执行：`main.style.display=""`；就 OK 了。

7、showBlog 与 showItemList

从第 4 行开始，到第 44 行，就是显示一个 Blog 的内容的两个函数。同样是脏活、苦活、累活，一点技术含量都没有，实在是不好意思跟大家讲解了，各位自己去看吧。

1.5 聪明的工作

step6 的工作，只能算是一个半成品，因为就算是在界面上的元素所显示出来功能，我也没有全部完成，比如：`unsubscribe`、`edit subscription`，还有显示的时间段选择，以及点击 `n Feeds` 那个 link，显示所有 blog 内容的功能。

还有一些缺陷，不知道朋友们有没有看出来？比如，我不能刷新这个窗口，要是一不当心按了一下 F5，所有的东西就都没有了。再者，我订阅了一个 blog 之后，也不能看看他还有没有写新的内容，如果我们将 `unsubscribe` 功能做出来的话，还可以将一个 blog 先退订，然后再重新订阅，这样不但傻，而且以前的老文章也就无法保留了。

由于整个页面过于脆弱，不可能保留过多的内容，按照时间范围来显示文章，也变得失去意义。

说到底，我们需要持久化一些数据。这样才能够长久的享用这个软件，如果这样简单的东西，也能称之为软件的话。首先被排除的方案是 `Cookie`，`Cookie` 当然可以在一段时间内保存数据，但是容量只有 4K，仅仅只是一个字符串，要保存复杂数据，就得自己做一套编码、解码的工作。权衡之下，我还是打算在自己的机器上架一个 Web 服务器和一个 MySQL 的数据库，然后将数据保存在数据库里。

这得算是大动作了！我们要好好的筹划筹划，以我多年的 Web 开发经验，

我打算装这样两个软件：AppServ 和 Resin。这可是两个好东西啊，我且简单介绍一下。

1.5.1 准备工作

AppServ 是一个 All In One 的安装程序，一个 exe 文件，下载安装，就能够帮你装好：Apache、MySQL、PHP、phpMyAdmin。甚至还有更加方便的 AddOns 版本，能够一次安装 drupal、gallery、Mambo、Moodle、phpBB、PHP-Nuke、Torrent-Tracker、WordPress、ZendOptimizer 等等 PHP 语言开发的知名软件。总之就是乐趣尽享，快乐无忧啊！

AppServ 的下载地址在：

http://sourceforge.net/project/showfiles.php?group_id=37459

AppServ 的首页在：<http://www.appservnetwork.com/>

AppServ 在 SourceForge 的主页在：<http://appserv.sourceforge.net/>

这个软件的下载安装，都相当简单，我就不再贴图撑页面了

需要提醒的一点是，在安装 MySQL 数据库的时候，最好选择 UTF-8 编码方式，这将使得我们的开发少掉非常多的麻烦。

说实话，Java 平台下的东西，大多难装、难用、难懂。远不如 PHP 或者 Ruby 下的东西。只有一个 Resin，是为 Java 争光的。Resin 是一个 Java 的 Web Server。你可以下载一个 zip 文件，解压缩在硬盘的任何地方，然后执行 bin 目录下的 httpd.exe 文件，一个 Java Web Server 就 Run 起来了。何等的简单方便！还不只如此，任何 resin 下的程序，你都可以直接修改，不需要系统重新启动，resin 就会帮你自动编译。通常的开发步骤几乎就和 PHP 的 Web 应用一样，改代码，F5 刷新页面，看看效果，再改代码，再 F5 刷新一下。非常的爽。

Resin 的公司主页在：<http://www.caucho.com/>

下载地址在: <http://www.caucho.com/download/index.xtp>

下一个最新的 resin-3.x.zip 就 OK 了。

为什么会装这样两个软件呢? 有的同学可能会说: 为啥不装一个 Ruby on Rails 呀? 为啥不装一个支持 Python 的 Web Server, 比如 Zope 呀? 为啥不装一个那个啥啥~~呀。OK, 我老实交待, 我也就是 Java 和 PHP 熟悉一些, 其它的语言, 都没有深入的研究过, 实在是尺有所短啊。

1.5.2 功能设计

有了服务器的帮助, 我们就可以完成更多的功能了。初步规划一下打算实现的功能吧:

1、保存订阅列表

每次打开页面的时候, 我的订阅列表都能够显示出来。然后再帮我去获取最新的 blog 记录。有新的订阅, 也能够记录下来。

2、保存阅读记录

能够记下我最后一次阅读的时间, 以便下次显示的时候, 告诉我有几篇新的。

3、收藏喜欢的文章

我的数据库里, 并不需要保存所有的 blog, 毕竟值得保留的帖子也不多。只有在我点击收藏以后, 才将帖子保存起来。

1.5.3 数据库设计

既然要在服务器端保存数据, 自然要考虑数据库设计的问题。该如何设计保存数据的表结构呢? 我们有两个 XML 格式的规范可以参考, 一个就是 RSS, 另外一个就是 OPML。

在使用众多的 RSS 阅读器的时候, 我们都可以导入导出 OPML 格式的 xml

文件，以记录自己的订阅情况。根据这个格式，我们就可以设计一个 opml 表了。

```
<opml version="1.0">
<head>
  <title>Blog 列表</title>
  <dateCreated></dateCreated>
  <ownerName></ownerName>
</head>
<body>
  <outline title="订阅列表">
    <outline title="读书、思考、生活" htmlUrl="http://spaces.msn.com/members/zbw25/"
type="rss" xmlUrl="http://spaces.msn.com/members/zbw25/feed.rss" />
  </outline>
</body>
</opml>
```

OPML 的格式，其实是可以支持订阅的层次结构的，不过我们这里就简化为一层了，这样开发起来可以简单一些。

```
CREATE TABLE opml (
  ID int(11) NOT NULL auto_increment,
  Title varchar(255) NOT NULL,
  HtmlUrl varchar(255),
  Type varchar(255),
  XmlUrl varchar(255),
  LastDateTime varchar(19),
  PRIMARY KEY (ID)
) TYPE=InnoDB;
```

这其中除了 OPML 同样具有的四个要素之外，还另外增加了两个字段：ID 和 LastDateTime。ID 是一个标准的逻辑主键，LastDateTime 则记录了改 Blog 的最近更新时间。

```
CREATE TABLE rss (
  ID int(11) NOT NULL auto_increment,
  OpmlID int(11) NOT NULL,
  Title varchar(255) NOT NULL,
  Link varchar(255),
  Author varchar(255),
  PubDate varchar(255),
  Description text,
```

```
PRIMARY KEY (ID),  
KEY OpmlID (OpmlID),  
FOREIGN KEY (OpmlID) REFERENCES opml (ID) ON DELETE CASCADE ON UPDATE CASCADE  
) TYPE=InnoDB;
```

类似的，参照 RSS 2.0 的格式，我们也简单的设计了一个叫做 rss 的表。其中 ID 也是一个逻辑主键，OpmlID 则是关联到 opml 表的 ID 字段。

两个表的 ID，都设置为自增长字段，这样也是为了开发简化的考虑，在实际开发中，并不推荐。

1.5.4 以 POST 方式向服务器提交数据

在开始 step7 之前，我得先做一个小的 test。试试看怎么把数据提交给服务器。以前我们都是简单的连接到服务器上的某一个 URL，Get 一堆东西回来解析。当然了，Get 方式也是可以提交数据的，不过像一篇 Blog 这样的大数据，Get 就不够了。我们需要 Post 一个 Form。

文件路径：CD-ROM/code/HelloWorld/PostTest/

一个是 server.php，首先需要在 AppServ 安装路径，比如 D:\AppServ\下找到 www 目录，在 www 下再建一个 ajax 目录，将 server.php 复制过去就可以了。另外一个是 client.html，同样是直接在 IE 里面打开。

代码很简单，只有三行代码值得解释一下：

```
9  xmlhttp.open('POST', 'http://localhost/ajax/server.php');  
10 xmlhttp.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');  
11 xmlhttp.send("test1=a&test2=b");
```

第 9 行与以前的代码的区别，就是将 GET，换成了 POST。

第 10 行，表示我这个请求，是一个普通的 Form 请求，去掉这一行的话，服务器端就收不到数据了。

第 11 行，就是一个字符串拼出来的请求数据，表示有两个参数，test1 和

test2。

Server.php 的代码也极其简单，就一行。将接收到的数据，再送回去。如果你能看到



这样一个窗口的话，那就说明使用 XMLHTTP，以 POST 方式提交数据，已经成功了。

1.5.5 读取和管理 OPML 数据

现在的任务是，在页面最初打开的时候，就能够以 AJAX 的方式，从数据库取得我的订阅列表，当我阅读了某个 RSS 以后，数据库应该将最近的一次阅读时间记录下来。当我增加新的订阅 RSS 时，不但在页面上能够显示出来，而且还能保存到服务器端。当然，修改和删除，也是必须的工作。

文件路径：CD-ROM/code/HelloWorld/step7/opml.php

```
1 <?
2 header("Content-type: text/xml; charset=gb2312");
3 $dbh=mysql_connect("localhost:3306","root","");
4 mysql_select_db("ajax");
5 $query="select * from opml order by ID";
6 $res=mysql_query($query, $dbh);
7 $err=mysql_error();
8 if($err){
9     exit();
10 }
.....
21 <?
22 while($row = mysql_fetch_array($res)) {
23     echo "<outline title=\"". XMLreplace($row["Title"]). "\"
htmlUrl=\"". $row["HtmlUrl"]. "\" type=\"". $row["Type"]. "\"
xmlUrl=\"". $row["XmlUrl"]. "\" lastDateTime=\"". $row["LastDateTime"]. "\"
ID=\"". $row["ID"]. "\"/>";
24 }
```

25 ?>

.....

//XML 转义符号替换

其中，第 2 行：`header("Content-type: text/xml; charset=gb2312");`是告诉浏览器，我这输出的是一个 XML 格式的文件，编码是 gb2312。余下的代码分为两个部分，一部分是最基础的 PHP 操作 mysql 数据的代码，另一部分就是拼接字符串，输出 XML 数据的代码。都非常简单，就不再解释了。

代码很简单，但是输出的结果却要解释一下，对比原来的那个 OPML 格式的文件，我们的输出多了两个属性。**lastDateTime**：最近一篇 Blog 的时间，用来查找新 Blog。**ID**：OPML 中一条记录的 ID 号。这是为了配合实现增加、修改、删除用的。

相应的 JavaScript，就是 `loadOPML()` 函数，也就是解析 XML 数据的工作，有点区别的是，这次主要的数据，不是在 Node 里，而是在 Attribute 里，所以我另外还写了一个 `getAttributeString()` 的函数，并将原来的 `getString()` 函数，`rename` 为 `getNodeString()`。这也是小小的一个重构了

然而，实际的 `loadOPML()` 函数，其实并不简单，具体的原因，我在下一节再来跟大家解释。

至于增加、修改、删除，我就想到了经典的 DAO 模式了。我打算做一个 `opmlDAO.php`，然后在里面实现增修删的 SQL 操作，然后由浏览器里的 JavaScript 通过 XMLHTTP 来调用。

文件路径：CD-ROM/code/HelloWorld/step7/opmlDAO.php

```
1 <?
2 $dbh=mysql_connect("localhost:3306","root","");
3 mysql_select_db("ajax");
4 if($action=="insert"){
```

```

5     $query="insert into opml (Title,HtmlUrl,Type,XmlUrl,LastDateTime) values ('$
title','$htmlUrl','rss','$xmlUrl','$lastDateTime');";
6 } else if($action=="update"){
7     $query="update opml set Title='$title',HtmlUrl='$htmlUrl',XmlUrl='$xmlUrl',L
astDateTime='$lastDateTime' where ID=$ID";
8 } else if($action=="delete"){
9     $query="delete from opml where ID=$ID";
10 }
11 $res=mysql_query($query, $dbh);
12 if($action=="insert"){
13     $lastID=mysql_insert_id($dbh);
14     echo "position=".$position.";";
15     echo "lastID=".$lastID;
16 }
17 ?>

```

这个 DAO ， 接受 这样 一些 参 数 ：

action, **position**, **ID**, **title**, **htmlUrl**, **xmlUrl** 以及 **datetime**。**action** 提示 DAO 期望进行的操作。**position** 是一个回传数据，用在 insert 操作的时候，将新插入数据的 ID 号，返回给客户端，而客户端通过定位数据，将新的 ID 号设置回去。

文件路径：CD-ROM/code/HelloWorld/step7/step7-tree.html (部分代码)

```

function insertOPML(position, blogfeed) {
    xmlHttp.open('POST', 'http://localhost/ajax/opmlDAO.php', true);
    xmlHttp.onreadystatechange=function() {
        if (xmlHttp.readyState==4&&xmlHttp.status==200) {
            .....//解析返回数据
        }
    }
    xmlHttp.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
    var poststr=makePostStr(position, blogfeed)+"&action=insert";
    xmlHttp.send(poststr);
}

function updateOPML(position, blogfeed) {
    xmlHttp.open('POST', 'http://localhost/ajax/opmlDAO.php', true);
    xmlHttp.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
    var poststr=makePostStr(position, blogfeed)+"&action=update";
    xmlHttp.send(poststr);
}

function deleteOPML(ID) {
    xmlHttp.open('POST', 'http://localhost/ajax/opmlDAO.php', true);
    xmlHttp.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');

```

```

var poststr="";
poststr=poststr+"ID="+ID;
poststr=poststr+"&action=delete";
xmlHttp.send(poststr);
}

```

因为 Insert 的特殊之处在于，需要处理返回的结果，以获得一条新增记录的 ID 号，updateOPML 与 deleteOPML，就简单很多，不需要解析返回值了。

1.5.6 排队读取 RSS 数据

本来我还认为，在获得了数据库中记录的 OPML 数据之后，事情就会很简单了，最初我写下的 JS 代码是这样的：

```

function loadOPML() {
    xmlHttp.open("GET", "http://localhost/ajax/opml.php", true);
    xmlHttp.onreadystatechange=function() {
        if (xmlHttp.readyState==4&&xmlHttp.status==200) {
            var xmlDom=xmlHttp.responseXML;
            var outlines=xmlDom.documentElement.childNodes.item(1).childNodes.item(
0).childNodes;
            for(var i=0;i<outlines.length;i++){
                var attributes= outlines.item(i).attributes;
                var xmlUrl=getAttributeString(attributes,"xmlUrl");
                loadRSS(xmlUrl);
            }
        }
    }
    xmlHttp.send(null);
}

```

令人遗憾的是，我没有得到自己想要的结果，而且每次运行的结果往往都不一样。当时我就眼前一黑！为什么呢？我给 loadRSS 的函数开头，加上了 alert(url)这样的调试语句，嘿，居然就对了。

知道什么 bug 最恐怖吗？就是不能跟踪的 bug，当你以调试代码的方式运行的时候，一切就正常，当你正常运行的时候，一切就都不正常！还好，以我多

年的 Java 开发经验，我直接这肯定属于多线程上的 bug，要解决这种多线程的 bug，往往我们必须在自己的脑子里跑代码，来分析问题可能会出在哪里。

整个代码运行的流程中，只有一个地方的运行时间，是不可预测的，那就是 load 远程的 RSS 数据的时候。当我们以一个循环的方式来调用 loadRSS 函数的时候，极有可能第一次 load 的数据还没有返回，第二次 load 的请求就发出了，当我插入了 alert 代码以后呢，因为每个请求我都要点击一次鼠标，这样就使得请求之间产生了间隔，反而就能正常了。也就是说，要解决这个问题，我的 load 请求，不能一拥而上，必须排队！

于是我将代码做了如下修改，在 loadOPML 函数处理数据时，预先将所有数据，都放入 blogList 数组，然后调用 waitToShow 函数。

新增一个全局变量 currentLoad，是一个 int 型的数组指针（这个说法并不准确，参见脚注），初始值为 0，用来指向 blogList 数组，，添加了一个 waitToShow 函数。

```
function waitToShow() {  
    if(blogList.length>currentLoad) {  
        loadRSS(currentLoad, blogList[currentLoad].xmlLink, "update");  
    } else {  
        showBlogList();  
    }  
}
```

当任务没有完成前，每次读取一个 RSS。

在一次 loadRSS 完成以后，再次调用 waitToShow 函数，判断是应该读取下一个 RSS，还是正确的显示 Blog 列表。

在 loadRSS 中还有一个地方，我的写法与前面的各个步骤不同。

```
if (xmlHttp.readyState==4) {  
    if(xmlHttp.status==200) {  
        var xmlDoc=xmlHttp.responseXML;
```



```

.....//解析 XML 数据，将 blogfeed 存入数组
    if(action=="insert"){
        insertOPML(position, blogfeed); //向数据库插入新记录
    } else if(action=="update"){
        updateOPML(position, blogfeed); //更新数据
    }
}
currentLoad++;
waitForShow();
}

```

这里将对于 `readyState` 的判断与对于 `status` 的判断分开了。因为无论对于这个数据的读取是否正确，我都得接着读下一个了。当然，按理还得再标个错什么的，我这里就不接着做细节了。

1.5.7 区分新老数据

我们前面设定的 `lastDateTime` 字段，最算能够派上用场了。通过 `loadOPML`，我们可以获得获得一个 `lastDateTime` 的日期数据，这代表着这个 **Blog**，我们最近一次看到的，他所发表的文章中的，最新一篇的日期。

在 `loadRSS` 的时候，我们可以通过对具体日期的比对，计算出在 `lastDateTime` 之后发布的文章，同时在显示 **Blog** 列表的时候，对于有新文章的 **blog**，以 **BlogTitle(2)** 这样的形式来显示。

还有很多的细节要做，在 `tree` 页面中，我们要考虑以下问题：

- 1、为每一个 `blogfeed`，添加 `newitem` 属性，以记录新文章的数量。
- 2、修改 `showBlogItemList` 函数，以特殊形式显示 `newitem` 大于 0 的 `blogfeed`。
- 3、修改 `loadRSS` 函数，计算新文章的数量，并且要注意区分已经存在的 **Blog** 的新文章计算，与新增 **Blog** 的新文章计算。因为已经存在的

Blog，是有 lastDateTime 数据的，而新增的 Blog，lastDateTime 只能是一个空的字符串。

- 4、在 loadRSS 同时还要记录新 load 的 RSS 的最新一篇文章的时间，并且在 blogfeed 的 tempLastTime 属性中保存，如果用户点击了 link，那么数据库就要将 lastDateTime 更新为 tempLastTime 时间，如果没有点击，则数据库中的数据应该没有变化。
- 5、修改 showBlogItem 函数，当用户点击 link 以后，要将 newitem 重新设为零，并且重画 Blog 列表，更新数据库中的 lastDateTime。
- 6、新增 formatDateTime 函数，将 GMT 格式的时间，转换为：yyyy-mm-dd hh:ss:ss 格式的时间。

在 base 页面中，我们也要做一些修改：修改 showBlog 函数，让它接收一个 newitem 参数，以决定显示文章的数量。这样在点击以后，就能够只显示新文章。同样也要修改 showItemList 函数，因为真正决定显示文章的，是这个函数。

到目前为止，我们的 Step7，已经基本上可以见人了。

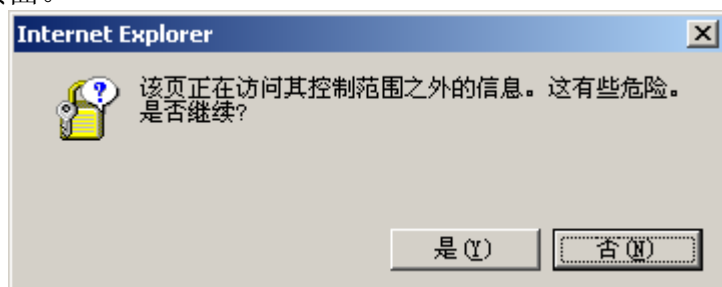
1.6 本章小结与思考

我现在做出来的这个 RSS Reader，说实话，还有很多不足之处，只能算是个半成品：

- 1、添加 Feed，没有做重复判断
- 2、Blog 的文章没有保存
- 3、显示文章的选择只有一个 Display All
- 4、只能处理标准的 RSS 2.0 格式，比如 RDF 这样的格式就不行

5、没有实现 showAllBlogItem 函数

但是呢，我还是打算把它拿出来见人，因为已经实现的部分，感觉还是相当不错的。如果能够给其它朋友用的话，岂不是一桩美事。而我现在的代码，都是一个一个的静态 HTML 页面，似乎有点不雅，反正我也装了 Apache，干脆就把这个静态页面也放到 Web Server 里去，然后只要访问 <http://domain/ajax/rssread.html> 这样的页面，不就搞定了吗？说干就干，我把文件 copy 到 AppServ 的 /www/ajax 目录下，再改改名字，然后在浏览器里访问这个页面。



于是我的 IE，就弹出了这样一个对话框，很是令人不爽。搞来搞去，也不知道如何才能让 IE 不出现这样的警告信息，这样的小问题，确实也很让人头痛的，我得找个人问问。

二 幡然悔悟

2.0 棒喝

我有一个老朋友，是一个 AJAX 的老法师，也是和我一样常年在 JavaEye 上晃的。大约在 2004 年 1 月初的时候，他就已经在 JavaEye 发起了一场非常深入的，关于基于 XMLHTTP 做表示层开发的讨论，有很多人都参与进去讨论，我却基本上没有插嘴，这方面的东西，也一直没有时间自己去多了解一下，心里那个悔恨哪。还好我跟他相当熟，看他 MSN 在线，就直接问了：

我：hi，有个小问题。

老法师：什么事？

我：嘿嘿，你最熟悉的领域，XMLHTTP 的，为啥我在本地调用一个 HTML，就能够访问各个 XML 数据源，但是从服务器端得到的一模一样的 HTML，就会弹一个安全警告的对话框呢？有没有办法，让 IE 不弹这样的东西呢？

老法师：咦，你写 XMLHTTP 的东西干啥？

我：其实呢，是出版社想要找我写本 AJAX 方面的书。

老法师：……

我：要不我把文字的部分和代码的部分都发给你吧，你有空了都帮我看看。别的问题倒是小事，就是这个对话框…

老法师：行啊，你发过来吧。

~~~嗖~~~OK

~~~嗖~~~2 个小时以后

老法师：你这个书写给初学者的？

我：是啊，书名叫做《新手快车道》，你觉得如何？

老法师：还快车道呢，出门就要出车祸的！你这简直就是误人子弟啊！

我：啊，人家的子弟，难道不就是拿来误的吗？

老法师：😡，我不是在跟你开玩笑，你这代码里有非常多的问题，要是人家照着你的代码写程序，肯定要被炒鱿鱼的。

我：😱，不会吧？咱们熟归熟，你这样乱说，我一样告你诽谤哦！

老法师：真的，我跟你从头说说吧。

能够认识这样的朋友，真是我的运气啊。

2.1 XMLHTTP 解惑

2.1.1 关于安全

老法师：我们来假设一个 IFrame 的应用场景吧。

如果你访问了一个黑银行，他其实是黑客用一个 IFrame，将真正的网上银行登录窗口套在他的网站里，显示给你看的。然后你就在这真实的登录窗口里面输入了你的卡号和密码。然后他的通过 JavaScript，读取到这个 IFrame 里面的 Form 中的 Input 的内容，你的卡号密码，不就到手了吗？

因此，从有 JavaScript 这样的脚本语言开始，同源策略，就是 Web 应用中浏览器端最为重要的安全保障策略。**任何一个 JavaScript 脚本，都不能访问与其不同域的 HTMLDOM 中的任何信息。**

另一个重要的安全策略则是，最初设计的 JavaScript 没有任何类型的**联网原语**（参见犀牛书 P447）。以免恶意网页利用客户端浏览器作为攻击平台，攻击其它的主机。

我：这和 XMLHTTP 有什么关系呢？

老法师：我们可以修改 Frame 的 href，或者 IFrame 的 src，以实现 GET 方式的跨域提交，但是却不能访问跨域的 Frame/IFrame 中的内容。

XMLHTTP 同样能够与远程的服务器进行信息交互，而且更加危险的是，**XMLHTTP 是一个纯粹的 JavaScript 对象，这样的交互过程，是在后台进行的，可以不被用户察觉的。因此，XMLHTTP 实际上已经突破了原有的 JavaScript 安全限制。**

如果我们又想利用 XMLHTTP 的无刷新异步交互能力，又不愿意公然突破

JavaScript 的安全策略，可以选择的方案就是给 XMLHTTP 加上严格的同源限制。

这样的安全策略，很类似于 Applet 的安全策略。IFrame 的限制还仅仅是不能访问跨域 HTMLDOM 中的数据，而 XMLHTTP 则根本上限制了跨域请求的提交。

而 IE 其实给这个安全策略开了两个想当然的后门，一个是：他假设你的本地文件，自然清楚将会访问什么内容，所以任何你的本地文件访问外部数据，都不会收到任何的警告。另一个是：当你访问的网站脚本打算访问跨域的信息时，他居然仅仅是弹出一个对话框来提醒你一下。如果一个欺诈网站，采用这样的手段，提供一个假页面给你，然后通过 XMLHTTP 帮你远程登录真实的银行服务器。只要 10 个用户里，有一个用户糊涂一下，点了一个确定。他们的盗取帐号行为，就成功了！

你想想看，这是何等危险的事情！

Firefox 就不是这样的做法，缺省的情况下，Firefox 根本就不支持跨域的 XMLHTTP 请求，根本就不给黑客这样的机会。

我：哦，果然是危险啊，那这去掉对话框，看来是不行了？那我看人家那么多 AJAX 应用，似乎都没有这样的对话框跳出来呀？

2.1.2 XMLHTTP Proxy

老法师：那是因为，他们其实都是访问自己本域的服务器，如果需要访问跨域的数据的话，就要通过一个被称为 XMLHTTP Proxy 的访问代理。我发一个 Java 的 XMLHTTP Proxy 代码给你参考吧

Server 端的 Servlet，我给他起的名字是：GetRawFile

文件路径：CD-ROM/code/HelloWorld/step8/GetRawFile.java

```
1 import java.io.PrintWriter;
2 import java.io.DataInputStream;
3 import java.io.IOException;
4 import java.net.URL;
5
6 import javax.servlet.http.HttpServlet;
7 import javax.servlet.http.HttpServletRequest;
8 import javax.servlet.http.HttpServletResponse;
```

```

9 import javax.servlet.ServletException;
10
11 public class GetRawFile extends HttpServlet {
12
13     protected void service(HttpServletRequest req, HttpServletResponse resp)
14         throws ServletException, IOException {
15         resp.setContentType("text/xml");
16         PrintWriter pw = resp.getWriter();
17
18         URL url = new URL(req.getParameter("url"));
19         DataInputStream dis = new DataInputStream(url.openStream());
20         String line;
21         while ((line = dis.readLine())!=null) {
22             pw.println(line);
23         }
24     }
25 }

```

而在 Client 端，则只需要增加一句：

```
url="getrawfile?url="+url;
```

以替换原来的 URL，就可以了。

我：我懂了，我现在用的还是 PHP 的，那就可以这样写：

文件路径：CD-ROM/code/HelloWorld/step8/GetRawFile.php

```

1 <?
2 header("Content-type: text/xml");
3 $fd = fopen( $url, "r" );
4 while ($buffer = fgets($fd, 4096))
5     echo $buffer;
6 fclose( $fd );
7 ?>

```

就达到那个 Servlet 的效果了。

但是我还想到一个问题，这个 URL 请求，其实两种都是同时存在的，我既有可能请求跨域服务器的某个 URL，也有可能请求本域的某个 URL。比如在 loadRSS 的时候，我要访问跨域的一个 URL，在 loadOPML 的时候，在添加、修改、删除 RSS 订阅的时候，我要访问的，又得是一个本域的 opml.php，或者 opmlDAO.php

看来我得这么写我的 XMLHTTP 调用了：

原来是：xmlHttp.open("GET", url, true);

现在可以写成：

```
var URL_PROXY="GetRawFile.php?url=";  
.....  
xmlHttp.open("GET", convertURL(url), true);  
.....  
function convertURL(url) {  
    if(url.substring(0, 7)=="http://") {  
        url = URL_PROXY + url;  
    }  
    return url;  
}
```

老法师：这样写，固然是可以达到你的要求，但是我总觉得味道很坏啊。

2.1.3 封装 XMLHTTP

我：什么意思？

老法师：我们先不接着说这 XMLHTTP Proxy，换一个话题，说说跨浏览器的问题。你现在的这个程序，仅仅只能在 IE6 下才能正确运行，任何其它的浏览器，都看不到效果，这样的程序风格，其实很不好！

我：但是现在国内的用户，几乎有 99.9% 都是在用 IE 呀。

老法师：这是一个非常常见的误区。

首先 IE Only 策略是很危险的。按照《网站重构》的说法：你不应该仅仅因为一个产品目前的市场占有率，就相信他能够永远垄断这个市场。而且你现在的代码，还不只是 IE Only，简直就是 IE6 Only。在支持 XMLHTTP 的不同的 IE 版本中，获取一个 XMLHttpRequest 的方法，也是各有不同的。

比如：

```
new ActiveXObject('Microsoft.XMLHTTP');  
new ActiveXObject('Msxml2.XMLHTTP');  
new ActiveXObject('Msxml2.XMLHTTP.3.0');  
new ActiveXObject('Msxml2.XMLHTTP.4.0');  
new ActiveXObject('Msxml2.XMLHTTP.5.0');
```

因为 IE 对于 XMLHTTP 的支持，是采用 ActiveXObject 的方式，这样的做法，就要求你的代码，依赖于具体的 ActivXObject 实现版本。所以你就算仅仅是为了在 IE 下运行正常，也得这么写：

```
var MSXML = ['MSXML2.XMLHTTP.5.0', 'MSXML2.XMLHTTP.4.0', 'MSXML2.XMLHTTP.3.0', 'MSXML2.XMLHTTP', 'Microsoft.XMLHTTP'];
for(var n = 0; n < MSXML.length; n ++){
    try{
        var xmlHttp = new ActiveXObject(MSXML[n]);
        break;
    }catch(e) {}
}
```

而且这样的做法，其实味道很坏。比较起来，FireFox 的接口就清爽得多。

```
var xmlHttp = new XMLHttpRequest();
```

作为开发人员，根本就不需要关心这个 XMLHttpRequest 具体是如何实现的。这样才是比较 OO 的做法。除非你铁了心，只跟 M\$ 的 IE 走，而且只打算将自己的产品运行在一个唯一的 IE 版本上，那么你就可以不比考虑封装 XMLHTTP。

一个用起来更加漂亮的方案（注），应该是类似于这样一段代码：

```
if (window.ActiveXObject && !window.XMLHttpRequest) {
    window.XMLHttpRequest = function() {
        var MSXML = ['Msxml2.XMLHTTP.5.0', 'Msxml2.XMLHTTP.4.0', 'Msxml2.XMLHTTP.3.0', 'Msxml2.XMLHTTP', 'Microsoft.XMLHTTP'];
        for (var i = 0; i < msxmls.length; i++) {
            try {
                return new ActiveXObject(msxmls[i]);
            } catch (e) {}
        }
        return null;
    };
}
```

这样，你在平时使用的时候，可以采取统一的：

```
var xmlHttp = new XMLHttpRequest();
```

这样的代码，来开始你的 AJAX 应用。

我：有点理解了，看起来这个采用对象化方式封装变化，主要还是因为 IE

很麻烦啊。

老法师：据说 IE 7.0 推出的时候，也将支持标准的 `new XMLHttpRequest()` 的形式了。

我：这倒是个好消息，问题是你前面把我的话岔开了呀，你说的 `convertURL` 味道很坏，是什么意思？

老法师：咱们接着 `XMLHTTP` 封装的思路讲下去。

`convertURL` 这样的工作，其实 `XMLHttpRequest` 的使用者，应该是不必了解的。我们可以用代理模式加一个装饰器模式，来封装一个实际的 `XMLHttpRequest`。从浏览器与服务器的关系来说，服务器实际上通过 `Servlet` 暴露了一个服务。客户端的代码是服务器端这个服务的代理。而从对于 `XMLHttpRequest` 的使用来说，我们通过一个装饰器，透明的扩展出了跨域访问的能力。

对于用户来说，他可以这么写：

```
var xmlHttp = new _XMLHttpRequest();
xmlHttp.open(GET,url,true);
```

至于 `_XMLHttpRequest`，则是对 `XMLHttpRequest` 的再次封装。

```
function _XMLHttpRequest() {
    this.xhr=new XMLHttpRequest();
}

_XMLHttpRequest.prototype.open=function(method,url){
    if(url.substring(0,7)=="http://"){
        url = URL_PROXY + url;
    }
    this.xhr.open(method,url);
}

_XMLHttpRequest.prototype.send=function (data) {
    this.xhr.onreadystatechange = wrapCallback(this);
    return this.xhr.send(data);
}

wrapCallback = function (_this) {
    return function () {
        _this.readyState = _this.xhr.readyState;
        if (_this.readyState == 4) {
            _this.status=_this.xhr.status;
        }
    }
}
```

```

        _this.responseXML = _this.xhr.responseXML;
        _this.responseText = _this.xhr.responseText;
    }
    _this.onreadystatechange();
}
}

```

2.1.4 XMLHttpRequest 的其它问题

IE 在 Cache 方面的 bug

老法师：还有其它一些问题也要跟你讲一下，虽然属于细节，但是却也相当要紧的。

先给你一段代码：

```

<script language="javascript">
function sendRequest(url) {
    req = new ActiveXObject("Microsoft.XMLHTTP");
    req.open("GET",url,true);
    writeStuff('Opening ' + url);
    req.onreadystatechange = function () {
        writeStuff('readyState is now ' + req.readyState);
    }
    req.send(null);
    writeStuff('Sending request');
}
function writeStuff(msg) {
    log.innerHTML += msg + "<br>";
}
</script>
<body onload="sendRequest('http://www.google.com');">
<div id="log"></div>

```

然后在 IE 里面运行一次，看看结果是怎么样的？

我：

```

Opening http://www.google.com
readyState is now 1
Sending request
readyState is now 2
readyState is now 3
readyState is now 4

```

老法师：你再按一下 F5 看看

我：咦，怎么次序变成这样了？

```
Opening http://www.google.com
readyState is now 1
readyState is now 2
readyState is now 3
readyState is now 4
Sending request
```

老法师：这是因为，当 IE 从本地 Cache 而非远程获取数据的时候，`onreadystatechange` 事件处理器在发送请求前就已经先执行完成了。如果再配合 `sniffer` 工具，你会发现，IE 根本就没有真正向远程网络发送请求。

我：🤔，这么大的 bug，IE 也好意思出来见人？

老法师：呵呵，这也算是 Web 开发中的正常现象了，Web 开发其实就是在 一个远非完美的环境中，努力做出尽可能完美的应用来。当年 IE4/NS4 的时代，那才是真的苦啊。闲话不说了，还是解决切实存在的问题吧。

IE 在提交 POST 请求的时候，并不会产生 Cache 问题。但是并非所有的 URL，都既可以接受 GET 请求，又可以接受 POST 请求。所以这个方案是肯定不行的。

首先可以考虑的方案，是追加 URL。使得每次提交的 URL，都是不同的。比如再次修改我们的 `open` 函数，加这么一句：

```
url += (url.indexOf("?") == -1 ? "?" : "&") + "timestamp=" + new Date().getTime();
```

这样 IE 就不会再缓存了。如果是自己的玩具应用，这么写也就够了，但是在实际的应用中，这样的做法却又是不可取的。因为每次都是一个不同 URL，这样每次都会到服务器端取一次数据回来，如果真的没有数据更新的话，这样的做法就是对于网络流量的巨大浪费了。

`XMLHttpRequest`，也支持通常 HTTP 请求中的 "If-Modified-Since" Header。

在 `open` 一个 url 之后，再加一句：

```
xmlHttp.setRequestHeader("If-Modified-Since", lastModified);
```

服务器就会判断客户端所请求的页面在 `lastModified` 这个时间点之后，是否更新过。如果有过更新，就会正常返回数据，否则就会仅仅返回一个 304 状态值。这是 IE 的 Cache 就派上正用了。而这个 `lastModified` 的时间，则是由上一次取得的 HTTP 返回中的 Header 提供的，这个值可以通过：

```
xmlHttp.getResponseHeader("Last-Modified");  
来得到。
```

但是，事情还是没那么简单，因为不同的 Web Server，对于 HTTP Header 的支持并不相同，有些服务器可能并不返回 "Last-Modified" Header，或者不支持 "If-Modified-Since" Header。因此这个办法，主要还是用在我们自己的 AJAX 应用中。在 Server 端，我们可以写这样的一段代码(PHP)，来确保浏览器不会做无谓地 Cache。

```
header("Expires: Sat, 1 Jan 2005 00:00:00 GMT");  
header("Last-Modified: ".gmdate( "D, d M Y H:i:s")."GMT");  
header("Cache-Control: no-cache, must-revalidate");  
header("Pragma: no-cache");
```

唉，这些问题，Firefox 都是不存在的。Firefox 的 XMLHttpRequest 对象每次发送 Request 的时候，都会自动加上取消 cache 的 Http Header。

其他可选择的通讯方式

我：XMLHTTP 很麻烦啊！我本来只是想弄一个异步的 HTTP 请求的，怎么这么复杂，要考虑这么多问题。有没有其它的方案呢？比如我就用 IFRAME 来请求，也不用管 IE/FireFox 的区别，也不用依赖于各种不同的 MS 的 XMLHTTP 版本。甚至连不支持 XMLHTTP 的浏览器，只要他支持 IFRAME，或者我干脆就搞一个宽度/高度为零的 FRAME 来整，还不是一样的吗？而且通用性还更强呢。

老法师：方案还有很多啊，比你设想的更加邪门的都有。我稍微跟你介绍一

下吧。

在 AjaxPatterns 网站上，将我们前面讨论的 XMLHTTP 方式，称为：XMLHttpRequest Call。另外还有 IFrame Call、Frame Call、Stylesheet Call、On-Demand Javascript、Image-Cookie Call、Import XML Document 等等不同的方式。

（注，参考地址：
http://ajaxpatterns.org/XMLHttpRequest_Call#Alternatives
http://zingzoom.com/ajax/ajax_with_image.php
）

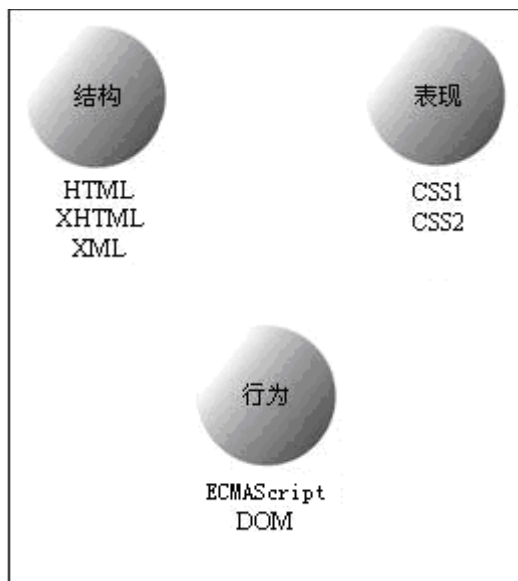
我：打住、打住，太多了。能不能给点指导原则之类的东西，这么多感觉很晕啊。

老法师：你现在对于 AJAX 的了解实在是片面得厉害。你以为 AJAX 就等于 XMLHTTP 应用吗？如果这样的话，多少年前的 DHTML，不是也应该叫做 AJAX 了？

你的问题远远不是对 XMLHTTP 不清楚那么简单，咱们这才说了一个开头呢。我得好好跟你补补课，介绍一下基于 Web 标准的网站开发，然后才能够来谈 AJAX 应用开发的指导原则之类的高深话题。

2.2 Web 标准布道^(注：鸣谢)

老法师：我们先来看一张图，这是《网站重构》中的一个经典图示：



结构（Structure）、表现（Presentation）和行为（Behavior），在书中被称为 Web 标准三剑客。其核心思想是，原本我们一股脑混在 HTML 中那么多要素，应该重新分解出来，各自采用独立的标准与技术。因为混在一起，只会制造不幸。

我：你是不是又打算来跟我推销‘兼容多种浏览器’的好处了？我还是不能明白，这兼容多种浏览器，能有多少好处？就算在国外，FireFox 的市场占有率比较高，也不过百分之十几。在国内 FireFox 能够占有多少？你说我为了这 5~10% 的 FireFox 用户，多花一倍的精力，值得吗？再说了，要兼容是不是还得兼容 Opera 呢？按照这样的目标，我岂不是要花三倍的精力？

老法师：呵呵，你这个误解，真是好典型啊！

你对于 Web 开发的认识，还停留在当年，针对不同的浏览器必须写不同的代码的年代。那真是一个不堪回首的年代。从 1996 年到 1999 年，那是 MS 与 Netscape 疯狂而愚蠢的浏览器大战的年代。大家竞相推出互不兼容的私有技术，除了迫使程序员苦不堪言之外，就是给大家留下了深刻的印象：**兼容异种浏览器，就得写分支代码。**

而事实上，发起成立于 1998 年的 Web 标准组织（WaSP），目的就是为了说服 MS 和 Netscape 终止这样的分裂。他采用从容进行、坚持和策略性的方法（又叫‘大喊、大叫和辩论’方法）注。最终使得浏览器生产厂商同意引进 WaSP。

从 2000 年开始，事情就开始逐渐改观。2000 年 3 月，IE5/Macintosh 版本发布，开始支持 XHTML、ECMAScript、几乎所有的 CSS1 规范、大部分 CSS2 规范，以及大部分 DOM。

紧随其后的是 Netscape，他们痛下决心，废弃所有原来的代码，完全从零开始，开发了 Mozilla，一个开放源代码的，能在所有平台上运行的，支持 XML、XHTML、CSS、ECMAScript 和 DOM，完全适应标准的浏览器。

接着是 Opera6(2001)、以及表现更好的 Opera7(2002)，也已经几乎完全支持了以上这些标准。

最后是 2002 年微软发布了 IE6 版本，几乎全面支持 Web 标准，并且非常强大。因此，现在我们可以非常轻松，仅仅需要开发一套页面，却能同时支持 **IE5+、Mozilla/Netscape 6+、FireFox 以及 Opera 5+**。甚至在更低版本的浏览器中，符合 Web 标准的网页，也至少能够被完整的查看其内容。（注参见《网站重构》

因此，可以这样来看待 Web 标准：

我们不是为了兼容异种浏览器，去依照 Web 标准开发；

而是因为 Web 标准开发本身的科学性与合理性，已经说服了浏览器厂商全面支持标准；

所以，符合 Web 标准的开发，不但更加科学、合理，而且同时能够获得浏览器兼容的好处。

我：我有点明白你的意思了，但是你还得说服我，这样的结构、表现和行为三分离的模式，是真正科学的、合理的。

2.3 结构、表现和行为

老法师：我先简单分别介绍一下结构、表现和行为，都是什么含义吧。

我们平时浏览网站，能够看到的 HTML、XHTML、XML，都是结构。也就是通过一系列的标记，结构化的描述文本数据。比如：

<h1>标题</h1>

<h2>子标题</h2>

<p>段落 1</p>

<p>段落 2</p>

这样常见的 HTML/XHTML。(注 HTML 与 XHTML 的区别随后再说。)

```
<user id="1">
```

```
  <name>庄表伟</name>
```

```
  <age>30</age>
```

```
</user>
```

或者向这样的 XML。都描述了结构化的数据。问题在于，原来的 HTML，实在是太具有包容性了，在 HTML 中，我们可以通过<style>标签来插入 CSS，可以通过<script>标签来插入脚本语言，最糟糕的是，HTML 本身还支持很多表示数据结构之外的标签，比如、、<u>、<small>等等等等。这些描述 Web 该如何显示的标签，与描述 Web 具体内容结构的标签，肆无忌惮的混杂在成千上万的网页里，使得网站的源码，根本就无法阅读和理解。

我：打住、打住，你说标签不能用了？我这么多年，都用到现在了呀。

老法师：你做了这么多年的程序员，也知道冗余代码是一种罪恶吧。多年以来，我们从不将 Web 页面开发，看做是编程，既厌恶，又轻视。Web 开发一向被认为是苦力活，似乎非常低级，开发人员的工资收入低，于是喜欢研究这个领域的人就更少，造成恶性循环。

但是，如果你以编程的眼光来看待 Web 开发，以消除冗余代码、重构、优化单元测试这样的眼光来看待 Web 开发，就会发现，这是一个充满趣味的工作。

而且，你不需要自己从头摸索，如何以开发程序的思路来开发 Web 页面，W3C 的众多标准已经制定出来，仅仅是遵循这些标准，就能大大简化你的劳动，何乐而不为呢？所以，观念的转变，是第一位的。

所以在 Web 标准中的结构这一部分，并不是对原有的 HTML 做加法，而是在做减法，从 HTML 到 XHTML 1.0，从 XHTML 过渡版（Transitional），到 XHTML 严格版（Strict），以及将来的 XHTML 2.0，一个根本性的目标，就是逐渐恢复结构化的文本数据这一本质。然后将原本由 HTML 标签实现的功能，分离出来交给 CSS、DOM、ECMAScript 这样的部分来完成。特别是 CSS，你应该

仔细的研究。

过去，我们常常采用相当**苟且偷安**的开发方式，修改一下标签，然后刷新一下 IE 浏览器看看效果，再改一改，再看看效果。这样形成的习惯记忆就是：什么样的 HTML 标签以及标签组合=什么样的视觉效果，而且仅仅是在 IE 中的视觉效果，比如使用<H1>，很多人的理由是他看起来大一些，使用，因为他能够给某一行字前面加一个圆点，使用<BLOCKQUOTE>，是因为他能够将一段文字缩进排列，还有、<I>等等等等。

而基于 Web 标准开发，有一个非常重要的观念上的转变——**XHTML 所表达的就是文档的内容**！虽然 XHTML 有着和 HTML 几乎完全一致的标签集，但是你在使用它们的时候，**应该根据标签的含义，而非它们的视觉效果来挑选**。

那么我们应该如何来达到想要展现的视觉效果呢，答案是交给 CSS。

看看你自己的那些代码，比如你的 step3 的 HTML 部分：

```
<body onload="startload()">
<font color=blue><a id="bloglink" href="#"><div id="blogtitle"></div></a></font><br>
<div id="managingEditor"></div></font><br>
<font color=blue><div id="blogdescription"></div></font><br>
<br>
<font color=#000066><a id="itemlink" href="#"><div id="itemtitle"></div></a></font>
<br>
By <font color=blue><div id="author"></div></font><br>
<div id="description"></div><br>
<div id="pubDate" align="right"></div>
</body>
```

这样的代码，简直就是糟糕透顶！如果是我来改的话，应该是写成这样：

```
<style>
span{color:"blue";}
span.date{color:"black";display:block;text-align:right;}
li{list-style:none;}
</style>
<body onload="startload()">
<ul>
<li><a id="bloglink"><span id="blogtitle"></span></a></li>
<li>By <span id="managingEditor"></span></li>
<li><span id="blogdescription"></span></li>
```

```
<li><a id="itemlink"><span id="itemtitle"></span></a></li>
<li>By <span id="author"></span></li>
<li><div id="description"></div></li>
<li><span id="pubDate" class="date"></span></li>
</ul>
</body>
```

然后，更应该将 CSS 的部分，也就是<style>标签包含的那一部分，保存在一个单独的文件里，比如 style.css，然后在 HTML 里只写一句：

```
<link href="style.css" rel="stylesheet" type="text/css" />
```

来引用这个 CSS 文件，这样才算是结构与表现相分离的，符合 Web 标准的 HTML 页面。

我：你这样的修改，究竟好处在哪里呀？我对 CSS 又不熟，要不你一一解释一下吧。

老法师：先说你原来的 HTML 里，有哪些问题：

1、仅仅使用<div>标签，没有区分<div>和。在 CSS 的布局概念中，SPAN 是一个内联元素(inline)，而 DIV 是一个块级元素(block)。在一个内联元素的前后出现的内容，能够和他们显示在同一行里。而每一个块级元素，都是从一个新行开始显示，而且其后的元素也必须另起一行进行显示。当你明确自己要描述的是一行内容时，就应该使用标签，比如 blog 的标题，作者，日期等待。而对于 blog 的正文，则肯定是一篇文章，这是你就应该使用<div>标签。而不是像你原来那样什么内容都用一个 div 套起来。

2、标签是最丑陋的 HTML 标签之一，如果你看到一个网站，他的源代码里充斥着又一个又一个的，那么基本上可以肯定，那个网站的设计师，从来没有好好学过 CSS，更不要说基于 Web 标准的开发了。更加合理的做法是，通过设置 CSS，来决定显示的字体、颜色、大小。

比如：span{color:"blue";}就能够让所有的 span，都显示成蓝色。根本不用你一个一个一个的去写。如果你将来要改颜色，比如将蓝色，改成深蓝色

#000066。那么你也只需要修改一个地方，而不是每一个地方。

同样的道理，也是一个应该被废弃的标签，正确的用法是在 CSS 中，设置：font-weight:bold;

3、你用
作为换行符，而这也是一个臭名昭著的垃圾标签。因为这是一个毫无意义的，纯粹为了达到某种显示效果的标签。如果是为了区分段落，那么应该使用<p>标签，如果是为了显示列表。像这样的写法：

```
item1<br>
item2<br>
item3<br>
就应该替换为
<ul>
<li>item1</li>
<li>item2</li>
<li>item3</li>
</ul>
```

我：你的意思我能明白，就还有两个地方没搞懂，一个是 span.date{...}，还有一个是 list-style:none 这两个地方是什么意思？

老法师：这两个地方，的确很可以说道说道。

CSS 是 Cascading Style Sheets——层叠样式表的缩写，这其中最精华的部分，就是这层叠二字了！CSS 中定义的内容，其实是一条一条的样式规则，这些规则的句型都是一致的：所有符合某种条件的元素，都按照以下规则显示。比如：
span{color:"blue";}

就规定了所有的 span 元素，都显示成蓝色。

这样固然是大大的减少了重复冗余的代码，但是，如果我有一个 span 元素，又不想显示成蓝色怎么办呢？不但不想显示成蓝色，我还希望能够在布局上，让他右对齐显示。这就需要定义第二条规则：

```
span.date{color:"black";display:block;text-align:right;}
```

以表明这是 span 元素中，有点特殊的一个。

当一个 HTML 元素，在一组 CSS 规则中，能够同时满足很多条规则的选择条件时，究竟应该具体应用哪一条规则，这就需要存在一个解决冲突的规则。比

如 `span.date` 的权重就大于 `span` 的权重。除了计算权重，还有一个继承的规则，在低权重规则中定义的属性，如果没有被高权重规则显式的覆盖，那么这个属性也将被继承下来。比如：我们如果不在 `span.date` 中特别申明 `color:"black"`，那么日期也将显示成蓝色。同样的，因为 `span` 缺省就是一个内联元素，如果我们不将其显式的定义为块级元素(`display:block;`)，那么后面的 `text-align:right` 规则，也将不起作用。

同样的道理，由于 `` 属于列表项元素(`list-item`)，因此在缺省情况下，每一个列表项之前，就会出现一个小圆点。而我们通过 `list-style:none` 规则，就能取消掉这样的圆点了。

因此，通过合理的运用 CSS 中的冲突规则与继承规则，我们既可以精确、灵活的控制 Web 页面的显示效果，同时又能够大大的节省冗余代码的数量。这就是层叠的力量。

但是，另外一个需要注意的问题是：缺省情况！

`` 缺省为内联元素、`<div>` 缺省为块级元素、`` 缺省会将字体显示为粗体、`` 缺省会显示一个圆点，等等等等。这样的缺省其实是一把双刃剑。一方面，缺省显示效果，能够大大的减轻开发人员的工作量，而另一方面，这又意味着，这些标签背后不可避免的隐含着某种视觉效果，而这样的缺省效果又是开发人员必须深入了解，并且在设计过程中纳入考虑范畴的。因此，要将内容与表现分离，总是不能做到足够的彻底。

明白了这个两难的境况，我们才能够来谈 XHTML 与 HTML、XML 的关系，以及将 XHTML 作为一种过渡方案的意义所在。

用 XML 来表示内容时，没有任何预定义的标签，自然每一个标签也没有任何缺省的显示效果。这样的 XML，自然全部都是结构化的内容，不可能包含任何关于如何显示的暗示。所有的表现定义，都必须由 CSS 或者 XSLT 来完成。由此造成的繁杂的工作量，已经根本违背了减轻程序员劳动的原意。

但是，HTML 那样的发展方向，是一条不归路！必须尽可能在一个合理的限度内，将结构与表现相分离。这就是制定 XHTML 标准的根本目的。

根据 W3C 的官方解释，XHTML 是一个用 XML 语法对 HTML 重新阐释的语言。XHTML 与原来的 HTML 4.0，看起来很像。但是有一些重要的不同。

- 以正确的 DOCTYPE 和名字空间开始
- 使用 META 内容元素声明内容编码语言
- 文档必须是良构的，所有的标签都必须层次分明
- 元素与属性都必须使用小写字母
- 非空元素都必须正确关闭
- 属性的值要加上引号
- 所有的属性都必须有值，比如<option selected=" selected "></option>
- 空元素也要关闭，比如

- 不要在注释内容中使用--
- 用<和&来表示<和&。

而这些规则，其实都是为了保证 XHTML 能够作为一个正确的 XML 被解析。这是内容能够结构化表达的第一步。

这还是一个开始，接下来，XHTML 定义了三种 DOCTYPE。

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

过渡的(Transitional):要求非常宽松的 DTD，它允许你继续使用 HTML4.01 的标识(仅仅是要求符合 xhtml 的写法)。

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

严格的(Strict):要求严格的 DTD，你不能使用任何表现层的标识和属性，例如
。

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

框架的(Frameset):专门针对框架页面设计使用的 DTD，如果你的页面中包含有框架，需要采用这种 DTD。

除了框架(Frameset)要用到的 XHTML, 过渡与严格的两种标准, 就是 W3C 希望能够引导 Web 开发前进的方向, 再进一步的 XHTML 1.1; 以及还处在草案阶段, 尚未有任何浏览器实现的 XHTML 2.0。都是希望能够引导 Web 开发, 越来越清晰的分离结构与表现。

我: 结构与表现应该分离, 这个我已经基本上理解了, 这个分离出行为又是怎么回事呢? 是不是就是不在 XHTML 里面写 JavaScript, 而是用<script>标签引入外部的 js 文件呢?

老法师: 行为能够分离出来的关键, 不在于 JS, 而在于 DOM。JavaScript 只是一种编程语言, 而 DOM, 则是 Web 编程所能发挥能力的对象与舞台。

在 W3C 的 DOM 标准出现之前, Microsoft 与 Netscape 的 IE4 和 NS4 各有自己的一套 DOM, 他们互不兼容, 遵照 Web 标准进行开发时, 应该主要针对支持 W3C 的 DOM 标准的浏览器:

- Netscape 6.0+、Mozilla 0.9+
 - IE/Win 5+、IE/Mac 5+
 - Opera7+
 - 以及其它一些用户数量极少的浏览器
- 这基本上已经是浏览器中的绝对主流了。

但是, 当年那些不支持 W3C DOM 的浏览器, 以及完全不支持 DOM 的浏览器, 还支持着另外一种 Web 编程的思维方式(**基于文本的 Web 编程**)。

先说文档对象模型所体现的编程思维(**基于 DOM 的 Web 编程**)。

在 DOM 看来, HTML 文档的层次结构, 被表示为树型结构。树的节点表示文档中的各种内容。我们来看这样一个 HTML。

```
<html>
  <head>
    <title>HTML DOM Page</title>
  </head>
```

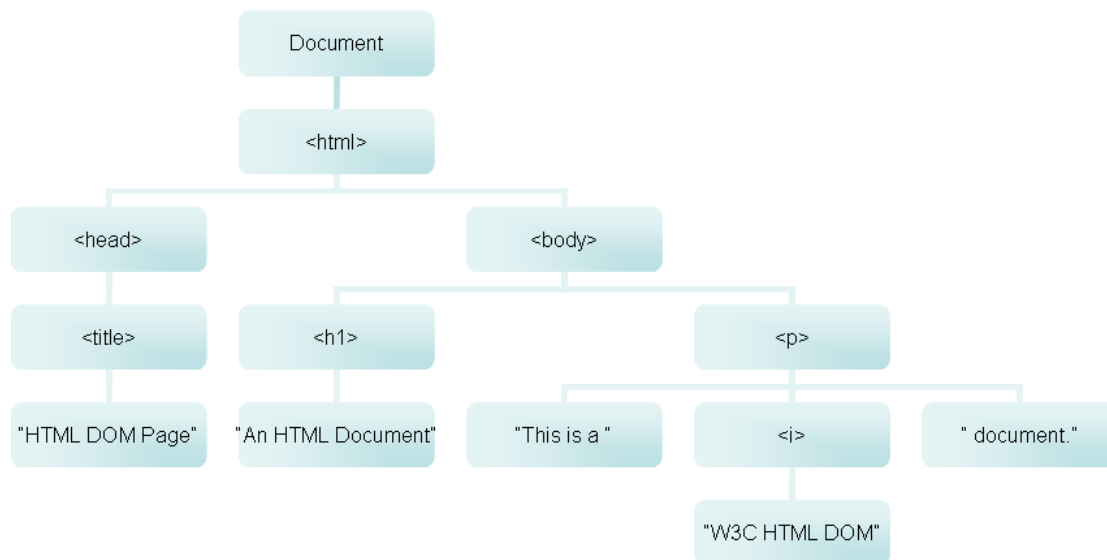


```

<body>
  <h1>An HTML Document</h1>
  <p>This is a <i>W3C HTML DOM</i> document.</p>
</body>
</html>

```

这个文档的 DOM 树形图，应该是这样的：



DOM 标准中，最为重要的概念自然是节点(Node)，整个 DOM 树，就是由一个一个的节点组合而成的，常用的节点类型有这样一些：

| 接口 | nodeType 常量 | nodeType 值 |
|------------------|-----------------------------|------------|
| Element | Node.ELEMENT_NODE | 1 |
| Text | Node.TEXT_NODE | 3 |
| Document | Node.DOCUMENT_NODE | 9 |
| Comment | Node.COMMENT_NODE | 8 |
| DocumentFragment | Node.DOCUMENT_FRAGMENT_NODE | 11 |
| Attr | Node.ATTRIBUTE_NODE | 2 |

我们可以运行这样一段代码，来体会一下。

文件路径：CD-ROM/code/DOM/DOMTest1.html

```

5 <script>
6 function showElement() {
7     var element=document.getElementById("h1");//h1 是一个<h1>标签
8     alert(element.nodeType); //nodeType=1
9     alert(element);
10 }
11 function showText() {
12     var element=document.getElementById("h1");
13     var text=element.childNodes[0]; //取得 h1 中的文字
14     alert(text.nodeType); //nodeType=3
15     alert(text);

```

```

16     alert(text.data);
17 }
18 function showDocument() {
19     alert(document.nodeType); //document 的 nodeType=9
20     alert(document);
21 }
22 function showAttr() {
23     var btn1=document.getElementById("btn1");//btn1 是一个按钮，它有很多属性
24     var attrs=btn1.attributes; //取得这个按钮的属性
25     for(var i=0;i<attrs.length;i++){
26         var attr=attrs[i];
27         alert(attr.nodeType); //nodeType=2
28         alert(attr);
29         alert(attr.name+"="+attr.value);
30     }
31 }
32 </script>

```

这段代码，在 IE 和 FireFox 里都能够正常运行，你可以运行一下，比较他们的区别，尤其是在 IE 中运行 showAttr()，你要有心里准备，哈哈。

这就是最简单的，运用 DOM 的程序了。W3C 的 DOM 标准，提供了大量的 API，让你可以对各种节点，执行各种各样的操作。实现了 DOM 标准的浏览器，通过对外提供 HTML DOM 的内存对象树，以及丰富的 DOM API，使得基于 DOM 的 Web 编程，能够有效的操作所有的页面元素，实现丰富的动态交互效果。

基于文本的 Web 编程，你应该是比较熟悉的，很多 JavaScript 的开篇 Hello World 程序，就是这么一句：

```
document.write("Hello World");
```

以显示 JavaScript 能够动态的在 HTML 里写点什么东西。更加高级一点的，自然就是 innerHTML，甚至 outerHTML 了。但是像 document.write、innerHTML、outerHTML 这样的用法，背后所隐含的基于文本的编程概念，都是与 DOM 编程模型，相冲突的。

基于文本的 Web 编程，时刻都在关注浏览器将把我的 HTML，解析成何等模样，要想实现动态交互的效果，就得关注如何改变页面中的某段 HTML 甚至更多的 HTML 文本。这样的思考模式，是很容易出现疏漏的。

而基于 DOM 的 Web 编程，不需要直接考虑 HTML、不需要直接考虑浏览器如何解析并显示 HTML，它的思考对象，就是内存中的那棵对象树，那里面的每一个都是对象，你通过面向对象的方式来操纵其中的各个节点，整个思考模型是清晰的，始终结构完整的，自然也是更容易开发与重用的。这才是结构、表现与行为三分离的根本原因！

再回过头来说说你前面的那个问题，关于 AJAX 开发的指导原则。

- AJAX 的开发，首先必须是基于 Web 标准的开发

AJAX 的应用，较之传统的 Web 应用，在表现层方面更加复杂。如果在开发过程中，不遵循 Web 标准，巨大的复杂性，将是难以把握的。

- AJAX 的传输可以分为三种类型

结构、表现与行为三中要素分离之后，加入 AJAX 的技术元素，也就是使得结构、表现与行为，都能够动态的、无刷新的改变。因此，概念的分解也应该贯彻到 AJAX 的传输模式之中。

- HTML DOM 的动态变化，应该依靠 XMLHttpRequest

XMLHttpRequest 技术，是典型的，从服务器端取回数据的方式。因此，这样的取得的数据，也应该主要用于改变浏览器中的 HTML DOM。通常传回的数据有两种形式，responseXML 和.responseText。各种合适的解析方式，都可以采用。

- 表现形式的动态变化，可以考虑采用 CSS Load

这时候的应用，一般都是完整替换整个 CSS 规则，比较少有局部 CSS 规则的改变。

- 浏览器端交互行为的运行逻辑，如果需要变化，可以考虑采取

JavaScript Load

- AHAH 方式，需要慎用

通过将返回的 `responseText`，直接赋值给某个 DOM 节点的 `innerHTML` 的方式，被成为 AHAH (Asynchronous HTML and HTTP)。这样的方式，从思维模式来说，往往难以脱离基于文本的 Web 的范畴，但是，这样的方式，在效率方面却很不错，因此需要慎用。

2.4 细节问题

我：老兄，我现在有点明白了，为什么我前面自己鼓捣那个 Ajax 的 demo，那么辛苦。如今新的先进的方法论在手，咱们就着手重构我原来的那些代码吧。

老法师：且慢，在真正改你的代码之前，还有一些概念要讲。

HTML DOM 与 XML DOM

老法师：在当年提出 Web 标准的时候，开发者大多数情况下都只需要考虑 HTML 的 DOM，但是在进入 AJAX 应用领域的开发之后，就出现了处理 XML DOM 的情况，由于两种 DOM 的 API 有大致重合，因此，也比较容易混淆。他们之间，最大的区别是，HTML DOM 作为一个内存对象树，在浏览器中，只存在一份。而 XML DOM，在浏览器内却可以创建多个。任何对于 HTML DOM 的修改，都会直接反应在浏览器的显示区中（`style.display=='none'`的情况例外。）应该明确的了解这两种 DOM 的区别。

XPath

老法师：还有一个问题，这得先看你的代码。比如：

```
var channel=xmlDom.documentElement.childNodes.item(0);
```

尤其是这句：

```
var  
outlines=xmldom.documentElement.childNodes.item(1).childNodes.item(0).childNodes;
```

实在是既复杂又脆弱！

先说他为什么脆弱，我们先来运行一段简单的 JavaScript。

```
<script type="text/javascript" src="xhr.js"></script>  
<script type="text/javascript">  
var xmlHttp=new XMLHttpRequest();  
xmlHttp.open("GET", "opml.php", true);  
xmlHttp.onreadystatechange=function() {  
    if (xmlHttp.readyState==4) {  
        var xml=xmlHttp.responseXML;  
        var nodelist=xml.documentElement.childNodes;  
        for(var i=0;i<nodelist.length;i++){  
            alert(nodelist.item(i).tagName);  
            //alert(nodelist.item(i).nodeType);  
            //alert(nodelist.item(i).data);  
        }  
    }  
}  
xmlHttp.send(null);  
</script>
```

看看在 IE 里面，会弹出几次对话框？在 FireFox 里会弹出几次对话框？

我：不会吧？opml.php，输出的是一个 XML 文件呀。我自己写的 php 程序，输出的 XML，根节点是<opml>，下面有两个子节点<head>和<body>。FireFox 的 bug 很大啊，除了弹出两个正确的名字之外，还夹花着弹出了三个 undefined。这算什么明堂？我再试试多输出一个 nodeType。

嗯，那三个输出 undefined 的节点，nodeType=3。

哦，原来是多出来 3 个 Text 类型的节点。那就再看看究竟是什么 Text。

拜托，FireFox 把我两个标签之间的空格，也算成是一个节点了，有这么乱来的吗？

老法师：你觉得这是乱来吗？我倒觉得 IE 的 XML 解析，有点乱来呀。他自作主张，忽略两个标签之间的连续空白，凭什么就可以断定这个空白不是 XML 数据的一部分呢？且不管他们究竟是谁在乱来，你这样的程序，

在不同的浏览器里一跑，肯定报错。

当然，微软也有他技术进步的地方，在 IE 中，就实现了 XPath 的功能，这正好可以解决我们的问题。

在 IE 中，通过提供 `selectNodes()` 与 `selectSingleNode()` 这样两个函数，我们就可以以类似文件路径的方式，来访问任意深度的 XML 数据。假设你要取得一个 `channel`，以前你就从 `item(0)` 那里取得，而在使用 XPath 之后，就可以这么写：

```
var channel=xmlDOM.documentElement.selectSingleNode("channel");
```

要取得 `<channel>` 下的 `<title>` 标签内的文字，就可以写成：

```
var  
blogTitle=xmlDOM.documentElement.selectSingleNode("channel/title/text()").nodeValue  
;
```

如果要取得 `<channel>` 下的所有 `<item>` 下的 `<description>` 标签内的文字，就可以写成：

```
var items = channel.selectNodes("item/description/text()");
```

当然，我们从顶层开始写起，也是可以的。这样，我们就可以得到一个名为 `items` 的数组，其中的每一个元素，都是 `String` 型的，我们需要的数据。

相对于你那样的复杂的写法，这样的写法，不但更加牢靠，而且也更加灵活。你原来那样的写法，一旦遇到 XML 数据结构调整、层次调整，改代码的工作量，就会非常惊人，而在使用了 XPath 之后，这些取得数据的字符串，就都变成可配置的了。

遗憾的是，到目前未知，Firefox 还没有提供 XPath 的功能，不过呢，可以写一段小程序，来帮 Firefox 打上这个补丁。这段代码就不具体列出了，参见：CD-ROM/code/common/xpath.js。

注：<http://km0ti0n.blunted.co.uk/mozXPath.xap>

只要包含了这个 `xpath.js` 文件，我们就可以在 Firefox 的浏览器里，像在 IE 里一样的使用 XPath 的功能了。

2.5 RSS Reader 翻新

我长吁一口气，说到：老兄，这下总算是差不多了吧。我现在心得很多啊，我们来重构原来的那个 RSS Reader 吧。

老法师：普遍的问题的，基本上就说得差不多了，要说到具体你的这个应用，问题还有不少啊。

我倒！

老法师：我们先从你的 JavaScript 看起吧。

第一步：DOM 节点的定义

在你的程序里，有一个用法，也是属于 IE 专用方法。当你定义了一个 DIV，比如：<div id="author"></div>

其中的 id 名称 author，在 IE 之中是可以直接引用的。

```
author.innerHTML=".....";
```

而在 FireFox 中，这个变量是不存在的。必须首先取得 HTML DOM 中的相应的节点。

```
var author=document.getElementById("author");
```

然后才能对这个节点进行操作。

我：那这样看来，还是 IE 体贴程序员啊。

老法师：那也不见得啊！你写一段简单的代码，自然觉得方便，但是要是复杂的应用，其中可能出现非常多的变量。这些变量相互混淆，怎么才分得清啊。比较合理的做法应该这样：

```
<script language="javascript">
function initVar(){
    var author=document.getElementById("author");
}
</script>
<body onload="initVar();">
```

通过这样的约定，在复杂的应用中，我们总能在 initVar 函数里，找到所有需要操作的 DOM 节点。

第二步：分割 JavaScript

老法师：现在你的这个代码，一股脑的堆在一个 HTML 页面里，根据我们前面介绍的基于 Web 标准开发的原则，应该将 JavaScript 分离出来，放在单独的文件里，更重要的是，我们应该根据这些代码的功能，进行适当的模块划分。

可以划分的出来的模块有以下一些：

- 封装 XMLHttpRequest: xhr.js
- XML DOM 分析处理相关函数: xpath.js
- OPML 操作相关函数: opml.js
- RSS 操作相关函数: rss.js
- 页面显示相关函数: show.js
- 其它工具类函数: util.js
- 主程序: main.js

当然，就你目前的项目规模来说，这样分实在是过细了。不过为了将概念都整理清楚，分得细一点，也不要紧。将来还是可以再合起来嘛。

XMLHttpRequest 的问题比较少，前面其实已经讨论过不少了，但是有一个关键性的问题，我们现在必须解决了，否则你这程序根本就不能算是 Ajax。

你自己写的代码里，为了不出现线程之间的冲突，你写了一个叫做 waitToShow 的函数。这个函数一用上来，就把异步 HTTP 请求所带来的好处，全都抵消掉了。

每 load 一个 RSS 数据成功之后，就会调用一次这个 waitToShow，如果这个时候没有 load 完所有的 RSS 地址，就会去 load 下一个 RSS。直到全部读取数据的工作完成，才会将内容显示出来。

那你有没有想过，我们为什么要用异步 XMLHttpRequest 请求呢？你这样的搞法，用户不是还是等在那里吗？这用户体验如何改善呢？

我：我理解你的意思了，这段我自己来试着改一改吧。这件事情还是得

分成两个线程来做，一个线程就管读取数据，另一个线程就定时刷新显示。

所以，我打算在你前面介绍的 XMLHttpRequest 封装的外面，再加一层封装，就叫做 **httpRequestManager**。由这个对象来管理一个请求队列和一个 XHR 连接池。另外还需要定义一个 **httpRequest** 对象，来封装一个个的 **http** 请求。

```
function httpRequest(method,url,header,data,callback,tryAgain)    //略

function httpRequestManager(xhrNumber,objName,timeLength){
    this(xhrNumber=xhrNumber;
    this(xhrArray=new Array();
    for(var i=0;i<xhrNumber;i++){
        this(xhrArray[i]=new XMLHttpRequest();
        this(xhrArray[i].parentName=objName;
        this(xhrArray[i].busy=false;
    }
    this(objName=objName;
    this(timeLength=timeLength;
    this(taskQueue=new Array();
}

httpRequestManager.prototype.start    //略
httpRequestManager.prototype.stop    //略
httpRequestManager.prototype.getFreeXHR    //略
httpRequestManager.prototype.addTask    //略
httpRequestManager.prototype.doTask=function(){
    var xhr=this.getFreeXHR();
    if(this.taskQueue.length>0&&xhr!=null){
        var hr=this.taskQueue.shift();
        xhr.onreadystatechange=hr.callback;
        try{
            xhr.open(hr.method,hr.url);
            if(hr.header!=null){
                xhr.setRequestHeader(hr.header.name,hr.header.value);
            }
            xhr.send(hr.data);
        }catch(e){
            putLog(this.objName+".doTask.e:"+e+"<br>");
            if(hr.tryAgain==true){
                this.taskQueue.push(hr);
            }
        }
    }
}
```

```
}
```

然后具体使用的时候，就应该是这样：

```
var xmlHttpRequestManager=new httpRequestManager(10,"xmlHttpRequestManager",100);  
xmlHttpRequestManager.start();
```

这样就能够建立一个同时可以发起 10 个并发 http 请求的管理器，而且以 100 毫秒一次的速率轮询请求队列。为了能够实现轮询，还得对之前的那个 `_XMLHttpRequest` 进行一下改造，使得管理器能够掌握每一个请求连接的 busy 信息。

老法师：嗯，我基本上就是这个意思，不过你这还没完呀，显示部分的配合代码呢？你也自己试试。

我：我想要改这么几个地方：

最关键的一点是 JavaScript 中的 `blogList` 中的每一个 `blogfeed` 得多加一个属性叫做 `state`，分为三种情况：`free`、`loading`、`complate`。

然后是改写 `waitForShow` 函数，将所有状态为 `free` 的 `blogfeed` 加入请求队列。并将状态置为 `loading`，如果所有的 `blogfeed` 的状态都为 `complate`，则停止对自己的定时调用。

接着改写 `loadRSS`，改名叫做 `addLoadRSSTask`，原本它的工作是读取 RSS 数据，现在是将一个的请求封装成一个 `httpRequest` 对象，提交给 `xmlHttpRequestManager`。

而且，在回调函数中，确保读取完成的 `blogfeed`，状态被置为 `complate`。

最后才是重写 `showBlogItemList` 函数：

将

```
showhtml=showhtml+"<td width=\"24\"><img src=\"blog.gif\"></td>";
```

改成

```
if(blogList[i].state=="complate"){  
    showhtml=showhtml+"<td width=\"24\"><img src=\"blog.gif\"></td>";  
} else {  
    showhtml=showhtml+"<td width=\"24\"><img src=\"wait.gif\"></td>";  
}
```

基本上就这样了。

老法师：不错不错，看来你基本功还是扎实的嘛！

我：切，对于一个 Java 程序员来说，搞定 JavaScript，还不是小菜一碟？

这个 JavaScript 与 Java 最大的区别，我看也就是 prototype 了吧，只要搞通了 prototype，其它的都不在话下。

关于 JavaScript 的几条重要规则

老法师：看来你呀，还是不能夸，一夸就飘起来。

JavaScript 本身不是一种复杂的语言，学习起来也很容易上手，要想学好 JavaScript，第一个障碍就是拿着 Java 去比附 JavaScript。首先要牢记的是 JavaScript 不是脚本化的 Java，他们之间有着重大的区别。正好最近 dojo 的作者之一 alex 发表了一篇文章，JavaScript Idioms Every Webdev Should Grok（注）。要开发 Web 应用，必须深入理解的几条规则，我来跟你解说一下吧：

- JavaScript 里一切皆是对象，包括函数。

“一切皆是对象”是一个诡辩的句子，从不同的角度此句可有不同的解释。所以 java 也号称是完全面向对象的语言。但是我以为 alex 这里所指的并非语义上的理解，而是从语法的角度。所以这一句应该理解为——在 JavaScript 语言中，操作的一切都是对象。

在 Java 中，数据类型分为基本数据类型(number, boolean, char 等)和对象数据类型——而其实 JavaScript 所操作的数据类型都是对象，类似于 Ruby。怎么理解呢？例如，整型和布尔值在 Java 代码中是一个基本数据类型，它是没有方法和属性的，在 JavaScript 中是怎样呢？看如下的代码：

```
//画蛇添足的方法，仅作为 example
Number.prototype.abs = function() {
    return Math.abs(this);
}
```

```
alert((-100).abs()); // 弹出返回值 100
Boolean.prototype.reverse = function() {
    return !this;
}
alert(true.reverse()); // 弹出返回值 false
```

所以， "-100" 和 "true" 在 JavaScript 中竟然是作为对象来处理的，JavaScript 代码中出现的所谓基本数据类型其实都是其 Wrapper 对象 (Number, Boolean 等)的直接量，JavaScript 在处理它们时实际上是在操作它们的 Wrapper 对象。从语法上来说，JavaScript 是比 Java 更面向对象的语言。

alex 更强调在 JavaScript 中函数也是对象，这是动态脚本语言的重要特性(作为对比，在 Java 中函数(方法)只是一种语法)。函数成为一个对象，所以函数完全可以看作数据：赋给变量、设为对象的属性、甚至作为其他函数的参数。函数是对象带来了巨大的灵活性，也是 JavaScript 面向对象编程的设计核心之一。当然与之而来的复杂性，也让 JavaScript 倍受误解。

- 所有对象都是可变的。

此句还好理解，因为 JavaScript 中虽然有 final 保留字，但并没有提供语法上的实现。所以 JavaScript 中无法定义一个变量是不可变的。另外一个重要理解就是，JavaScript 的对象在创建后可以随意添加或删除属性和方法。

- 原型对象的属性也是可变的

此句算是对第二句的补充，JavaScript 中每个对象类型(就是构造器函数)都有原型对象，我们可以在任何时候任意改变原型对象的属性，而且这个改变将影响已经通过 new 操作符实例化的对象和将来会实例化的对象。

- "." 操作符等同于"[]"操作符。

此句也是针对 JavaScript 的对象而言。对象的属性可以用"."加标识符"bar"来存取，也可以使用"[]"配合标识符的字符串形式来存取。

foo.bar === foo["bar"]; （注===三连等，代表完全等于，是 JS 中的一个算符）

这样就给存取对象属性带来了灵活性——可以动态构造一个字符串并以它来存取对象的属性。此句也揭示了：在 JavaScript 中数组就是对象，对象本质上就是一个关联数组。

- new 关键字实际上是先创建一个对象，再让构造器方法在该对象域中执行。

此句有些难理解，以一个简单的构造器函数举例：

```
function Dog(name) {  
    this.name = name;  
}  
var dog = new Dog("小明");
```

我以我的理解模拟 new 操作符的执行步骤：

```
var dog = new Object(); //首先创建一个新的 Object 对象  
dog.prototype = Dog.prototype; // 设置构造器函数的原型对象为 dog 的原型对象  
Dog.apply(dog, arguments); //将新对象传递给构造函数，此时构造函数可以用 this 来初始化新对象
```

所以 javascript 并非象 java/C++ 的构造器函数那样，先执行然后返回一个对象。而是先创建一个未定义属性的干净的 Object 对象，接着把构造器函数的原型对象设置为新对象的原型对象(构造器函数的原型对象是 JavaScript 自动为每个函数创建的)。最后再把这个对象交给构造器函数，在构造器函数中设置该对象其他的属性和方法等等。

通过这样的方式带来的巨大动态特性就是，可以象操作普通函数一样操作构造器函数。所有自定义的对象甚至可以使用同一个构造器函数。实际上这也是 Prototype 和 Dojo 的核心设计思想。通过这样的方式和原型对象的设计，JavaScript 实现了与 Java 完全不同的面向对象设计。

- 所有函数都是闭包
- this 关键字关联于执行时的作用域，而非定义时的作用域。

这两个问题，都是关于 JavaScript 中的作用域问题的。

JavaScript 上的作用域，是词法作用域，而非动态作用域。我们来解释一下这是什么意思：

在《JavaScript 权威指南》一书中，给出了一个非常好的例子，咱们抄一下里面的代码，文字就不好意思抄了。

```

var x="global";
function f(){
    var x="local";
    function g(){
        alert(x);
    }
    g();
}
f();

```

这个代码本身很好理解，这段程序的输出结果，自然是 **local**。

因为，**g** 是在 **f** 中定义的，因此在 **g** 中调用的 **x**，一定是在 **f** 中定义的 **x**。所以，无论在什么时候调用 **f** 的结果，都肯定是 **local**。这就是词法作用域的含义。

还有一个概念，单独理解，也很容易：在 **JavaScript** 中，函数是一等公民，可以作为参数传入，也可以作为返回值被返回。

```

function makefunc(){
    return function(x) { return x; }
}
var f1=makefunc();
alert(f1(1));//这时 alert 的是 1
alert(f1(2));//这时 alert 的是 2

```

说到底，因为这个 **f1** 虽然是由 **makefunc** 造出来的一个函数，但是他其实和普通的函数没有任何区别。

JavaScript 的这两个特性，分开来都很容易理解。但是一旦结合起来，就相当的令人困惑了。

```

function makefunc(x){
    return function() { return x; }
}
var f1=makefunc(1);
alert(f1());//这时 alert 的是 1
var f2=makefunc(2);
alert(f2());//这是 alert 的是 2

```

这两个函数 **f1** 和 **f2**，都没有接受任何输入参数，那么，他的返回值是哪里来的呢？

就是因为，函数的作用域，是他的词法作用域，也就是他被定义时，所处的位置中，能够访问到的那个 **x**。而这个 **x**，也就是 **makefunc** 所接受的

输入参数。

当我们把 `makefunc` 返回的函数，在 `makefunc` 之外使用的时候，在 `f1` 和 `f2` 的中依然要记得自己被定义时的上级作用域中的各种变量的值。这些需要被 `f1`、`f2` 记住的值，就是“闭包”。严谨一点的定义可以这么说：“为了函数能够正确的执行，需要被函数使用的，词法作用域中的，非全局数据，存在于函数的闭包之中。”

接下来再说这个 `this` 的问题。`this` 关键字关联于执行时的作用域，而非定义时的作用域。为什么会这样呢？

我们来看这么一段古怪的代码：

```
Object.prototype.str="Hello";
function Obj() {
    this.echo=function () {
        alert(this.str);
    }
}
var o=new Obj();
o.echo();
o.str="World";
o.echo();
```

如果我们注释掉第一行代码，程序在**运行时**输出的第一个对话框，就是 `undefined`。第二个是 `World`。而在第一行代码运行之后，程序输出的第一个对话框，就是 `Hello` 了。`Object` 的 `prototype` 与 `Obj` 这个对象，有什么关系呢？

原来，在 `JavaScript` 中，并没有严格的面向对象概念，自然也没有类的构造函数这样的概念。`var o=new Obj();`这样的语法，看起来似乎和 `Java/C++` 相当类似，但是他背后的执行过程是不同的。

首先，解释器会 `new` 一个空的 `Object` 对象。然后将这个空的 `Object`，作为隐藏的参数传递给 `function Obj()`。在 `Obj` 函数中访问到的 `this`，其实就是这个传入的空的 `Object` 对象。所以，如果这个 `Object`，并非为空，那么 `o.echo()` 这个方法的执行效果，就发生了变化。这就是所谓：“`this` 关键字关联于执行时的作用域”的含义。

再多说一句，如果你想把一个函数作为“构造函数”，那么就不要在

函数的最后加上 `return` 语句。因为如果没有 `return` 语句，`new` 算符返回的就是那个被操作过以后的 `this`。一旦你通过 `return` 返回了别的东西，这个 `this` 就被废弃掉了。对于函数的调用者来说，就会相当的困惑。

结束：各位，不好意思，这本《Ajax 新手快车道》就写到这里了。希望对各位学习 Ajax，能够有所帮助。