Broadview
www.broadview.com.cn

Savvy JavaScript

悟透JavaScript

（美绘本）

阿里软件资深架构师 李战 著
沉鱼 绘

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

悟透JavaScript

翻开此书的你，

也许是JavaScript的崇拜者，正想摩拳擦掌地尝试下学一学这一精巧的语言；

也许是80后，90后的程序员或者前端架构师，正被JavaScript魔幻般的魅力所吸引，所困惑，已经徘徊许久……

那么本书正是你所需要的！

通过本书，您可以

- 独辟蹊径学习、理解和运用JavaScript
- 更深入地理解AJAX技术
- 在学习技术本身的同时，领悟到编程的境界
- 更轻松地编写动态网页
- 更多地享受到读书的快乐和程序的魅力……

请尽情地享受快乐的程序人生吧！

上架建议：JavaScript

ISBN 978-7-121-07473-8
9 787121 074738 >
定价：49.00元

网上订购：www.dearbook.com.cn
第二书店·第一服务

责任编辑：孙学瑛
责任美编：李　玲

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

Savvy JavaScript

悟透JavaScript

（美绘本）

阿里软件资深架构师 李战 著
沉鱼 绘

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

翻开此书的你，也许是JavaScript的崇拜者，正想摩拳擦掌地尝试下学一学这一精巧的语言；也许是80后，90后的程序员或者前端架构师，正被JavaScript魔幻般的魅力所吸引，所困惑，已经徘徊许久……那么本书正是你所需要的！通过本书，您可以独辟蹊径学习、理解和运用JavaScript；通过本书，您可以更轻松地编写动态网页；通过本书，您可以更深入地理解AJAX技术；通过本书，您可以在学习技术本身的同时，领悟到编程的境界；通过本书，您可以更多地享受到读书的快乐和程序的魅力……

您能快乐地享用本书，是我们最大的期盼！

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

悟透JavaScript：美绘本 / 李战著.—北京：电子工业出版社，2008.12
ISBN 978-7-121-07473-8

Ⅰ. 悟… Ⅱ. 李… Ⅲ. JAVA语言—程序设计 Ⅳ. TP312

中国版本图书馆CIP数据核字（2008）第151618号

责任编辑：孙学瑛
印　　刷：北京天宇星印刷厂
装　　订：涿州市桃园装订有限公司
出版发行：电子工业出版社
北京市海淀区万寿路173信箱 邮编100036
开　　本：787×1092　1/16　　印张：12　　字数：220千字
印　　次：2008年12月第1次印刷
印　　数：5000册　　定价：49.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

目录

第一部 JavaScript真经

引子 数据与代码的纠缠 3
1 回归简单 10
2 没有类 13
3 函数的魔力 16
4 代码的时空 20
5 奇妙的对象 25
6 放下对象 28
7 对象素描 32
8 构造对象 34
9 初看原型 37
10 原型扩展 43
11 原型真谛 47
12 甘露模型 52
13 编程的快乐 63

第二部 手谈JavaScript

1 禅棋传说 66
2 标准网页 69
3 网页运行原理 75

4 文档对象模型 78
5 妆扮DOM对象 82
6 响应DOM事件 87
7 播放声音 93
8 别向复杂低头 102
9 珍珑棋局 111

第三部 点化AJAX

1 叩问AJAX 116
2 直捣AJAX 119
3 ASP.NET AJAX简介 124
4 AJAX与WebService 129
5 AJAX之双手互搏 139
6 域名的鸿沟 145
7 跨越域名的时空 152
8 特使协议 159
9 单点共享登录模型 163
10 编程之禅 175

引子 数据与代码的纠缠 3
1 回归简单 10
2 没有类 13
3 函数的魔力 16
4 代码的时空 20
5 奇妙的对象 25
6 放下对象 28
7 对象素描 32
8 构造对象 34
9 初看原型 37
10 原型扩展 43
11 原型真谛 47
12 甘露模型 52
13 编程的快乐 63

前言

JavaScript是一种轻量级的动态语言，主要用于动态网页的编程。由于JavaScript独特的语言特性和魔幻般的灵活性，使得她是那样的让人难于琢磨，一般的程序员也很难猜透她的心思。正是由于JavaScript有这样一些近乎神秘的色彩，使得JavaScript成了一种既简单，却又难学的编程语言。

也许你正好是JavaScript的初学者，或者是使用过JavaScript语言的熟手，甚至是精通JavaScript语言的顶尖高手。但相信你在阅读完本书之后，一定会有不同的收获。你会发现，原来可以这样来理解JavaScript，原来可以这样编写动态网页，原来可以这样来看待AJAX技术，原来可以这样地快乐编程。

本书起源于作者的一篇同名博客文章。因此，读者会发现本书的语言风格很是独特。文章的内容并不像传统技术文章那样的严谨，反倒让读者感受到轻松和幽默，阅读起来并不枯燥。书中穿插了一些有趣的小故事，这些故事背后都蕴含着一些简单的人生哲理。在讲解技术的同时点缀这些故事插曲，能让读者在学习技术本身的同时，领悟一些编程的境界。

禅，是一种让人快乐的东西。本书的字里行间总流动着一股清新的禅风，让读者在不经意间享受着读书的快乐。书中所讲述的那些原理和技巧，也许你已经非常熟悉，或许你也未曾见过。不过，通过作者蜻蜓点水般的说禅，或许你会豁然开朗，看破技术背后的禅机。

禅，是需要自己感悟的。因此，作者也准备了许多小巧的实例，让读者能自己去试一下。希望通过读者自己的揣摩和尝试，真正领悟JavaScript技术的真谛，达到“悟透”的境界。当然，真正要达到“悟透”的境界也是不容易的。因此，作者更多地讲述了程序员应该如何放下某些不正确的心态。

编程之禅是一种境界，也是一种放下的心态。只要我们放下无谓的争执，放下狭隘的观点，以乐观和包容的心态对待一切，我们将获得思想上的自由。编程之禅实际上就是一种快乐编程的感受，快乐编程的关键就是放下技术本身。这样，我们的思路就不会被狭隘的技术困住，就能创造一些新的思路和灵感。

为了让读者在轻松愉快的心情中阅读本书，书中还配了许多幽默的漫画。这些漫画又可以让读者从另外的视角去看待技术思想。或许就在你不经意的微微一笑中，又多了一种对技术思想新的感悟。这也正是本书与众不同的风格，文字是天马行空，漫画又幽默搞笑，读起来十分有趣。

如果你已经准备好了，就请

开始享受这轻松愉快的阅读之旅吧。

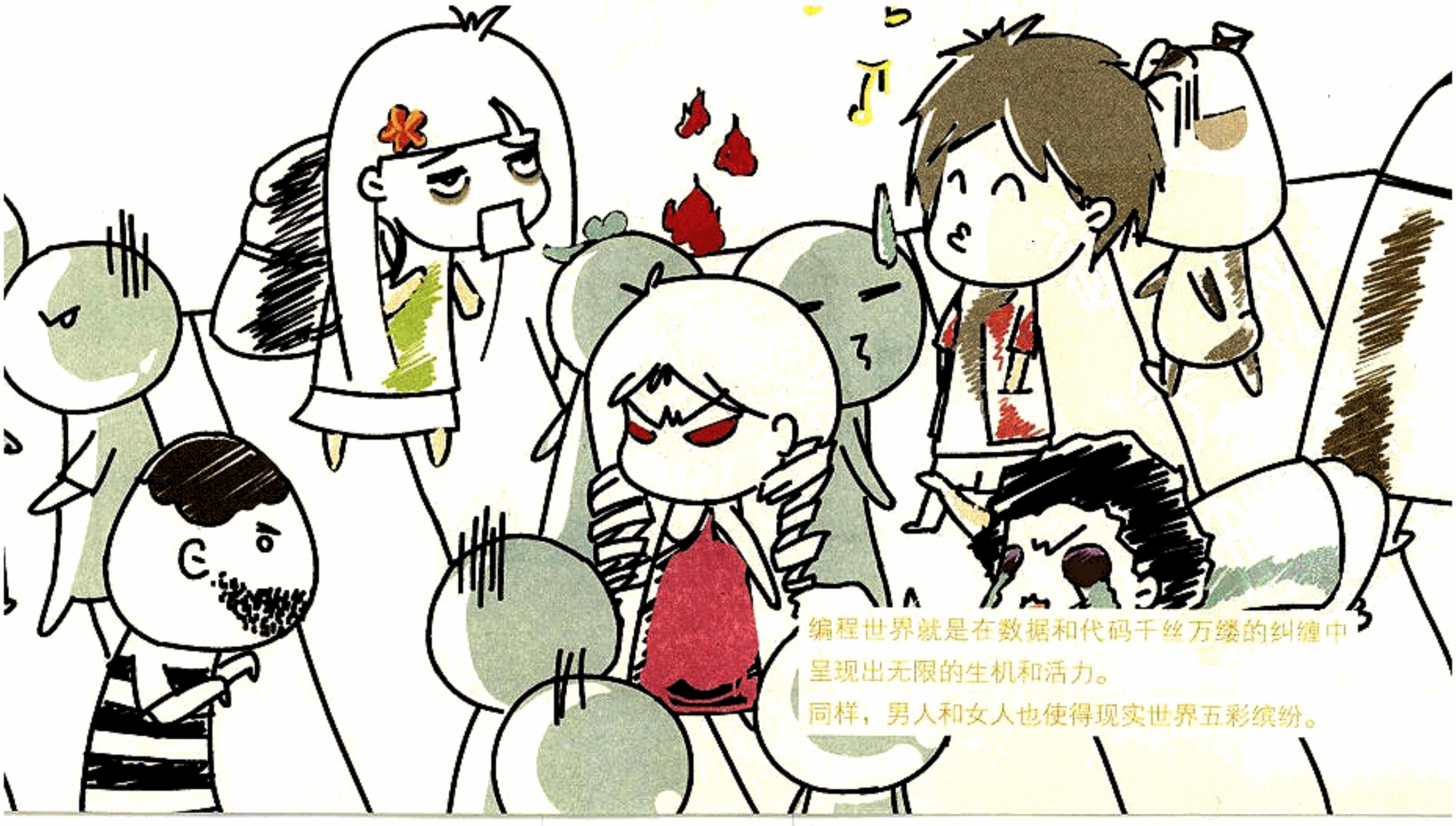
引子

数据与代码的纠缠

编程世界里只存在两种基本元素：一个是数据，一个是代码。编程世界就是在数据和代码千丝万缕的纠缠中呈现出无限的生机和活力的。

数据天生就是文静的，总想保持自己固有的本色；代码却天生活泼，总想改变这个世界。

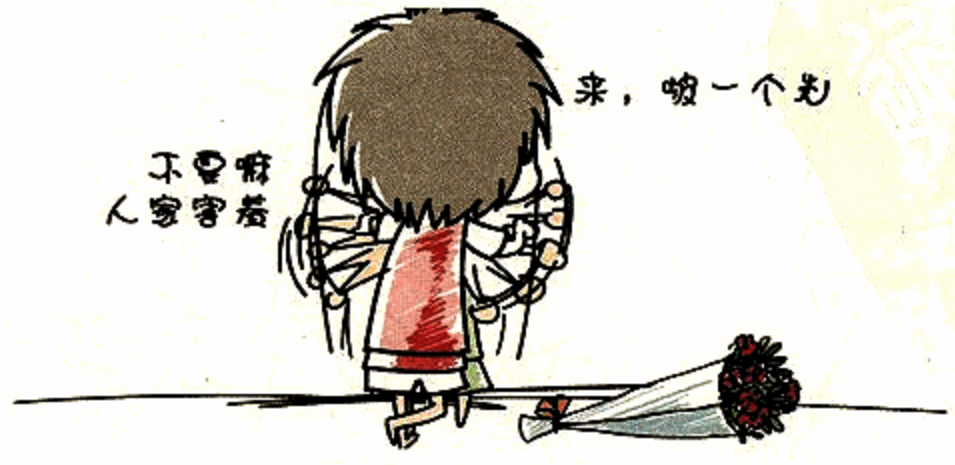
而你看，数据代码间的关系与物质能量间的关系有着惊人的相似。



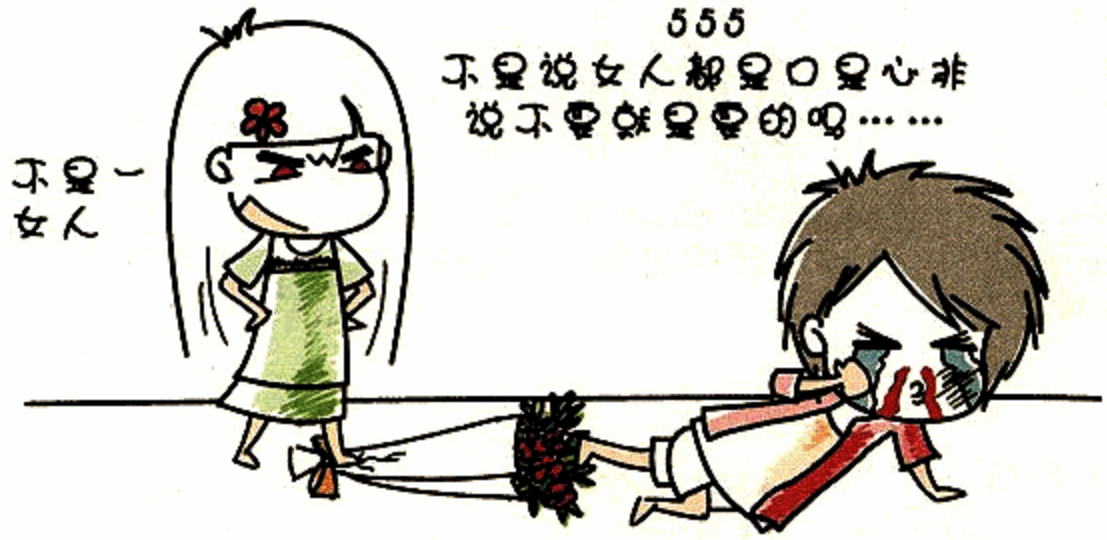
数据也是有惯性的，如果没有代码来施加外力，她总保持自己原来的状态。

而代码就象能量，他存在的唯一目的，就是要努力改变数据原来的状态。

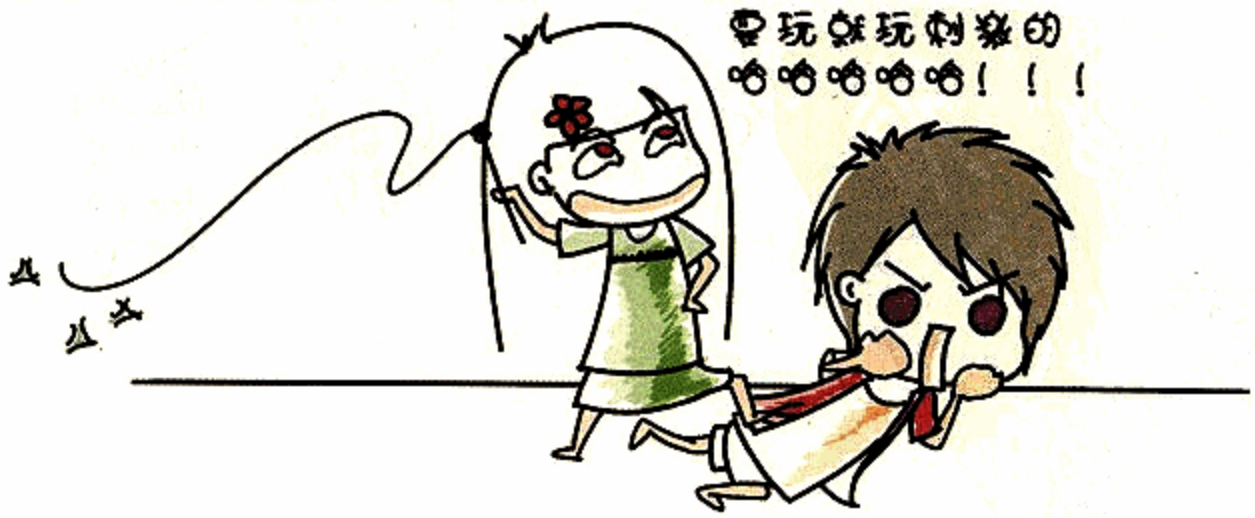
在代码改变数据的同时，也会因为数据的抗拒而反过来影响或改变代码原有的趋势。



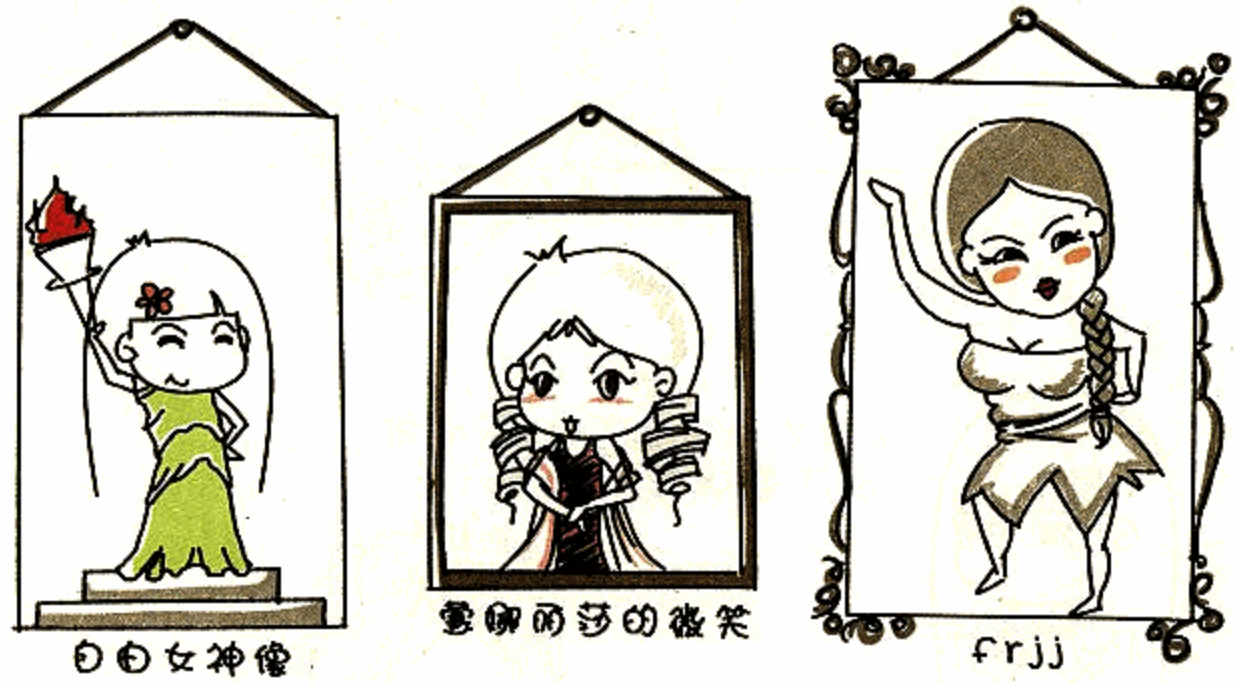
甚至在某些情况下，数据可以转变为代码，而代码却又有可能被转变为数据，或许还存在一个类似$E=MC^2$形式的数据转换方程呢。



然而，就是在数据和代码间这种既矛盾又统一的运转中，总能体现出计算机世界的规律，这些规律正是我们编写的程序逻辑。



不过，由于不同程序员有着不同的世界观，这些数据和代码看起来也就不尽相同。于是，不同世界观的程序员们运用各自的方法论，推动着编程世界的进化和发展。



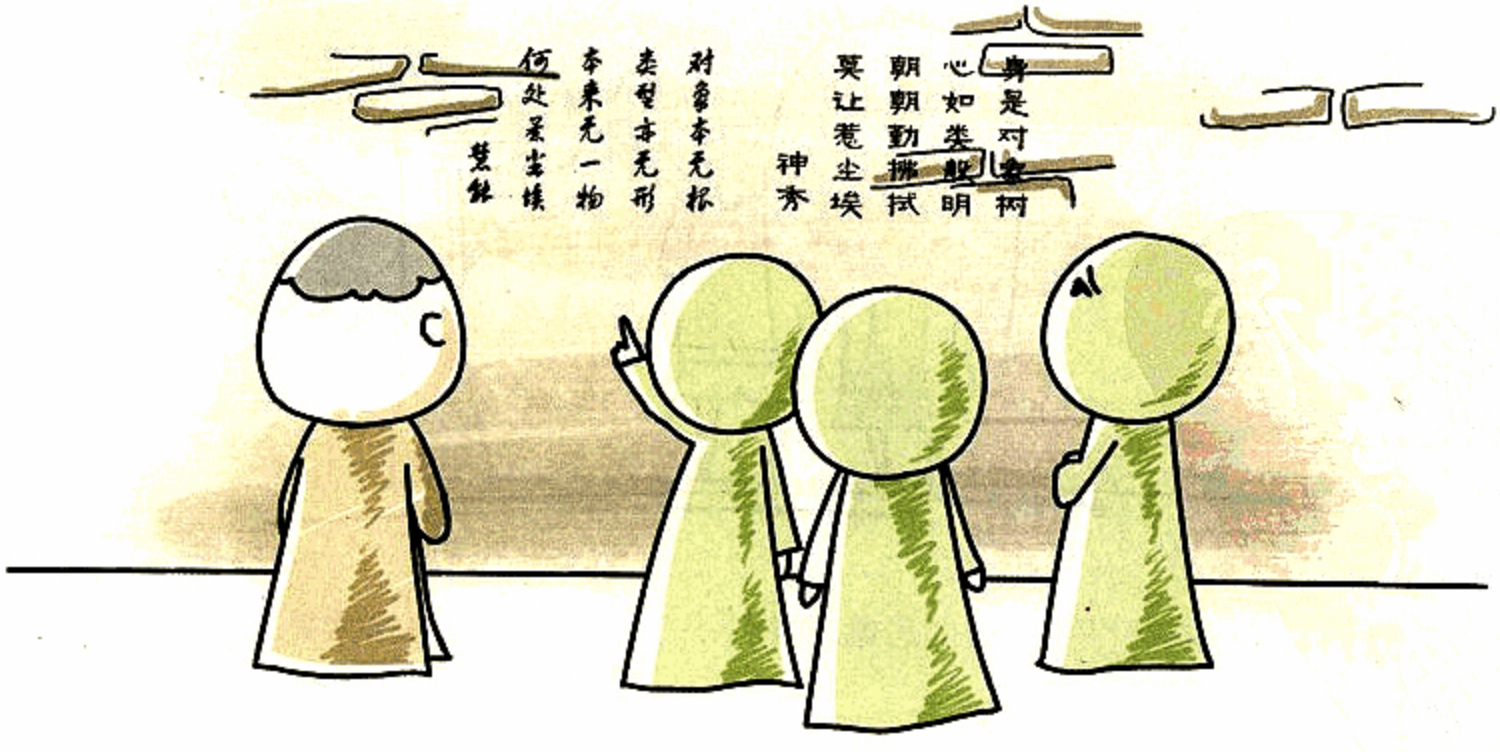
众所周知，当今最流行的编程思想莫过于面向对象编程的思想。为什么面向对象的思想能迅速风靡编程世界呢？因为面向对象的思想首次把数据和代码结合成统一体，并以一个简单的对象概念呈现给编程者。这一下子就将原来那些杂乱的算法与子程序，以及纠缠不清的复杂数据结构，划分成清晰而有序的对象结构，从而理清了数据与代码在我们心中那团乱麻般的结。

我们又可以有一个更清晰的思维，在另一个思想高度上去探索更加浩瀚的编程世界了。

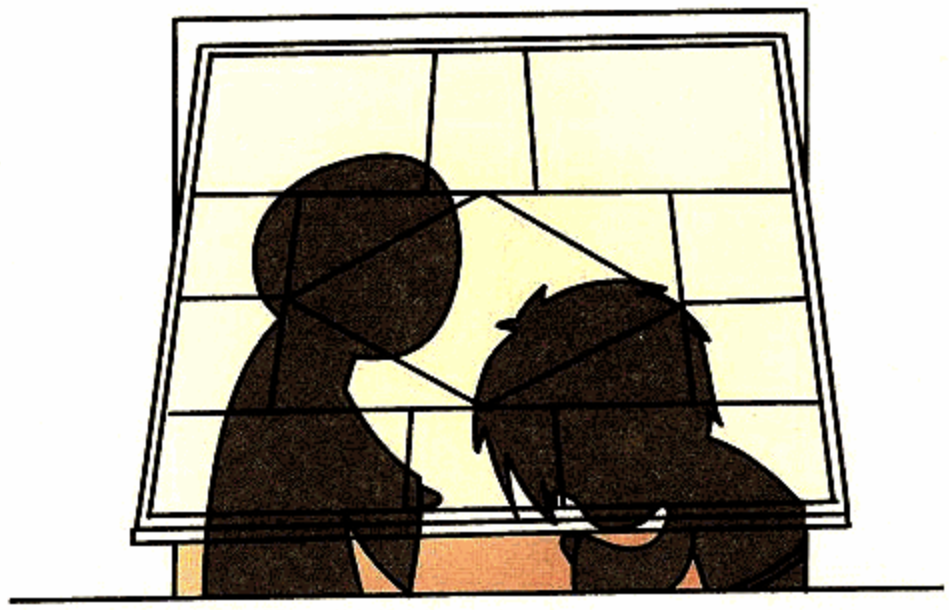
在五祖弘忍讲授完《对象真经》之后的一天，他对众弟子们说：“经已讲完，想必尔等应该有所感悟，请各自写个偈子来看”。

大弟子神秀是被大家公认为悟性最高的师兄，他的偈子写道：“身是对象树，心如类般明。朝朝勤拂拭，莫让惹尘埃！”。此偈一出，立即引起师兄弟们的轰动，大家都说写得太好了。

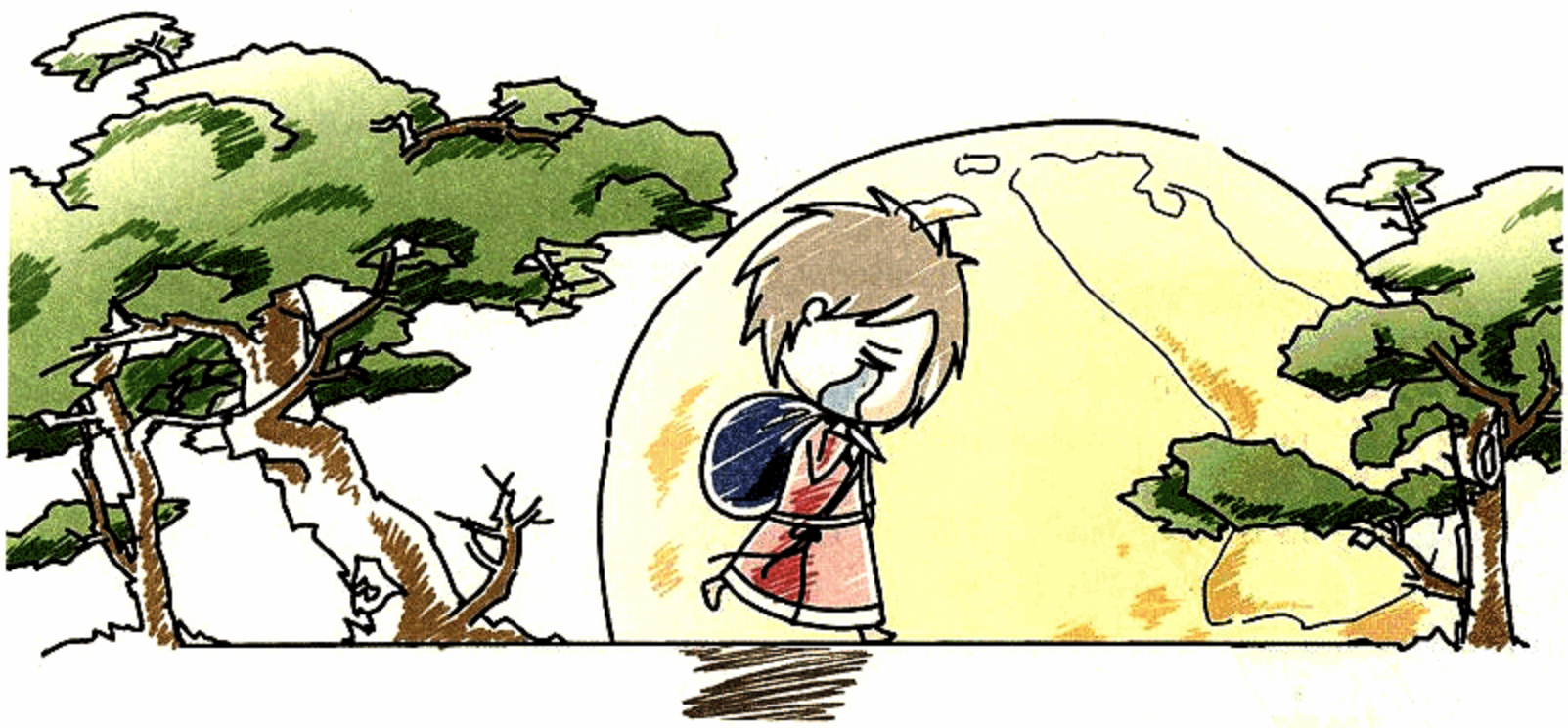
只有火头僧慧能看后，轻轻地叹了口气，又随手在墙上写道：“对象本无根，类型亦无形。本来无一物，何处惹尘埃？”然后摇了摇头，扬长而去。大家看了慧能的偈子都说：“写的什么乱七八糟的啊，看不懂”。师父弘忍看了神秀的诗偈也点头称赞，再看慧能的诗偈之后默然摇头。



就在当天夜里，弘忍却悄悄把慧能叫到自己的禅房，将珍藏多年的软件真经传授于他，然后让他趁着月色连夜逃走……



后来，慧能果然不负师父厚望，在南方开创了禅宗另一个广阔的天空。而慧能当年带走的软件真经中就有一本是《JavaScript真经》！



1
回归简单

要理解JavaScript，你得首先放下对象和类的概念，回到数据和代码的本原。前面说过，编程世界只有数据和代码两种基本元素，而这两种元素又有着纠缠不清的关系。JavaScript就是把数据和代码都简化到最原始的程度。

JavaScript中的数据是很简洁的。简单数据只有 undefined, null, boolean, number和string这五种，而复杂数据只有一种，即object。这就好比中国古典的朴素唯物思想，把世界最基本的元素归为金木水火土，其他复杂的物质都是由这5种基本元素组成的。

JavaScript中的代码只体现为一种形式，就是function。

注意：以上单词都是小写的，不要和Number, String, Object, Function等JavaScript内置函数混淆了。要知道，JavaScript语言是区分大小写的呀!

可以用typeof运算符来获取一个JavaScript元素的类型。由于typeof是运算符，因此可以有两用等价的写法：typeof X和typeof(X)。加不加括号无所谓，加了括号之后看起来似乎更习惯些。typeof运算之后得到的结果是一个string类型的值，也就是说：typeof 123的结果是“number”，typeof typeof 123的结果一定是“string”。

任何一个JavaScript的标识、常量、变量和参数都只是unfined, null, bool, number, string, object 和 function类型中的一种，也即typeof返回值表明的类型。除此之外没有其他类型了。

先说说简单数据类型吧。

undefined	代表一切未知的事物，啥都没有，无法想像，代码也就更无法去处理了。 注意：typeof(undefined) 返回的也是 undefined。 可以将undefined赋值给任何变量或属性，但并不意味清除了该变量，反而会因此多了一个属性。
null	有那么一个概念，但没有东西。无中似有，有中还无。虽难以想像，但已经可以用代码来处理了。 注意：typeof(null)返回object，但null并非object，具有null值的变量也并非object。
boolean	是就是，非就非，没有疑义。对就对，错就错，绝对明确。既能被代码处理，也可以控制代码的流程。
Number	线性的事物，大小和次序分明，多而不乱，便于代码进行批量处理，也控制代码的迭代和循环等。 注意：typeof(NaN)和typeof(Infinity)都返回number。 NaN参与任何数值计算的结构都是NaN，而且 NaN != NaN。 Infinity / Infinity = NaN。
String	面向人类的理性事物，而不是机器信号。人机信息沟通、代码据此理解人的意图等功能，都靠它了。

为了进一步简化编程，JavaScript还在这些简单数据之上，再规定了一些特殊含义。这些特殊含义又可以使得JavaScript的语句写得更简洁。但是，这些特殊含义有时也会给我们设下难以琢磨的陷阱。

例如，undefined, null, “”, 0 这四个值转换为逻辑值时就是false，除这四个家伙再加上false本身之外，其他的任何东西（包括简单类型值、所有对象和函数）转换为逻辑值时都是true。有趣的是在undefined, null, “”, 0, false这五个家伙中，除undefined==null之外，它们却又互不相等。使用这些规定，我们可以编写更简洁的逻辑判断语句。

再如，完全由数字组成的字符串与该字符串表示的值是相等的。因此，“123”==123的值是true，这也为我们编写代码提供了方便。

但是，“0123” == 0123的值却是false！奇怪吗？因为JavaScript将“0”开头的整数常量当八进制数处理，所以后面的0123实际是八进制数，而“0123”是按十进制转

换成数字值的，自然不会相等。这也常常给我们制造了非常奇怪的问题，当然不能怪JavaScript有问题，该反思的是自己的人品。

那么，我们不想让“123”==123该怎么办呢？这样，“123”===123就返回false了！

三个等号？

对！JavaScript里的三个等号“===”表示“全等”，也就是数据值与数据类型都必须相等才是true。因此，undefined==null是true，但undefined===null就是false。

当然，“!==”就是“不全等”了。注意，“不全等”与“全不等”是两个概念。”123”与123是“不全等”，但并非“全不等”！

“!==”这种“不全等”可以表示为这样的等价逻辑：

```
A != B || typeof(A) != typeof(B)
```

而“全不等”必须是值不相等而且类型也不相等，应该是这样的逻辑：

```
A != B && typeof(A) != typeof(B)
```

不过，在编程中基本遇不到“全不等”这样的判断，因此JavaScript也就没有专门的运算符了。

所有简单类型都不是对象，JavaScript没有将对象化的能力赋予这些简单类型。直接被赋予简单类型常量值的标识符、变量和参数都不是一个对象。

所谓“对象化”能力，就是可以将数据和代码组织成复杂结构的能力。JavaScript中只有object类型和function类型提供了对象化的能力。

2
没有类

object就是对象的类型。在JavaScript中不管多么复杂的数据和代码，都可以组织成object形式的对象。

但JavaScript却没有 “类”的概念!

对于许多面向对象的程序员来说，这恐怕是JavaScript中最难以理解的地方。是啊，几乎任何讲面向对象的书中，第一个要讲的就是“类”的概念，这可是面向对象的支柱。这突然没有了“类”，我们就像一下子没了精神支柱，感到六神无主。看来，要放下对象和类，达到“对象本无根，类型亦无形”的境界确实是件不容易的事情啊。

这样，我们先来看一段JavaScript程序：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS01.htm

```
<script type=" text/javascript" >
  var life = {}; //光溜溜的生命对象
  for(life.age = 1; life.age <= 3; life.age++)
  {
    switch(life.age)
    {
      case 1:
        life.body = “卵细胞”; //增加body属性
        life.say = function(){alert(this.age+this.body)}; //新建say方法
        break;
      case 2:
        life.tail = “尾巴”; //增加tail属性
        life.gill = “腮”; //增加gill属性
        life.body = “蝌蚪”;
        life.say = function(){alert(this.age+this.body+” -” +this.tail+” ,” +this.gill)};
        break;
      case 3:
                        delete life.tail; //删除tail属性
        delete life.gill; //删除gill属性
        life.legs = “四条腿”; //增加legs属性
```

```
        life.lung = "肺";     //增加lung属性
        life.body = "青蛙";
        life.say = function(){alert(this.age+this.body+"-"+this.legs+","+this.lung)};
        break;
         };
         life.say();    //调用say方法，此方法逻辑每次都会动态改变。
  };
</script>
```

不过，在看完这段程序之后，请你思考一个问题：

这段JavaScript程序一开始产生了一个生命对象life，life诞生时只是一个光溜溜的对象，没有任何属性和方法。

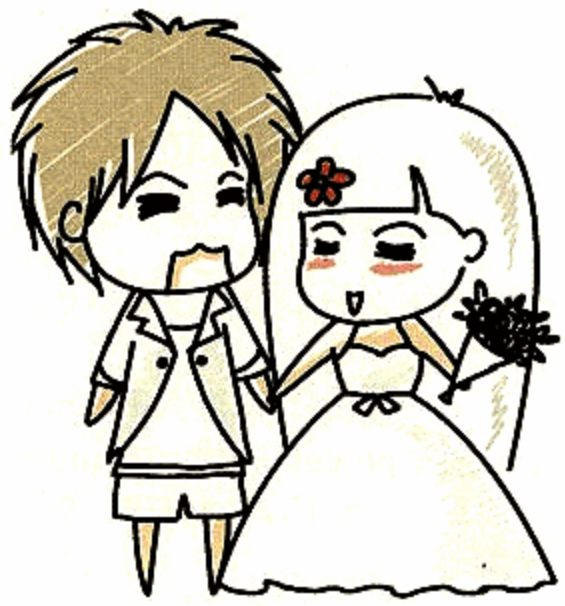
在第一次生命过程中，它有了一个身体属性body，并有了一个say方法，看起来是一个“卵细胞”。

在第二次生命过程中，它又长出了“尾巴”和“腮”，有了tail和gill属性，显示它是一个“蝌蚪”。

在第三次生命过程中，它的tail和gill属性消失了，但又长出了“四条腿”和“肺”，有了legs和lung属性，从而最终变成了“青蛙”。

如果，你的想像力丰富的话，或许还能安排他和命中注定的姑娘相遇。

真爱之吻让他变回英俊的王子。



两个人走进婚姻的殿堂从此过着幸福的生活直到永远、永远……

这个故事告诉我们：
要爱护青蛙，虽然他们长得很丑
可谁知道他不是个王子呢：）

我们一定需要类吗?

还记得儿时那个“小蝌蚪找妈妈”的童话吗？也许就在昨天晚，你的孩子刚好是在这个美丽的童话中进入梦乡的吧。可爱的小蝌蚪也就是在其自身类型不断演化过程中，逐渐变成了和妈妈一样的“类”，从而找到了自己的妈妈。这个童话故事中蕴含的编程哲理就是：对象的“类”是从无到有，又不断演化，最终又消失于无形之中……

“类”，的确可以帮助我们理解复杂的现实世界，这纷乱的现实世界也的确需要进行分类。但如果我们的思想被“类”束缚住了，“类”也就变成了“累”。想象一下，如果一个生命对象开始的时就被规定了固定的“类”，那么它还能演化吗？蝌蚪还能变成青蛙吗？还可以给孩子们讲小蝌蚪找妈妈的故事吗？

所以，JavaScript中没有“类”，类已化于无形，与对象融为一体。正是由于放下了“类”这个概念，JavaScript的对象才有了其他编程语言所没有的活力。

如果，此时你的内心深处开始有所感悟，那么你已经逐渐开始理解JavaScript的禅机了。

3
函数的魔力

接下来，我们再讨论一下JavaScript函数的魔力吧。

JavaScript的代码就只有function一种形式，function就是函数的类型。也许其他编程语言还有procedure或 method等代码概念，但在JavaScript里只有function一种形式。当我们写下一个函数的时候，只不过是建立了一个function类型的实体而已。请看下面的程序：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS02.htm

```
<script type=" text/javascript" >
  function myfunc()
  {
    alert( "hello" );
  };

  alert(typeof(myfunc));
</script>
```

这个代码运行之后可以看到typeof(myfunc)返回的是function。以上的函数写法我们称之为“定义式”的，如果将其改写成下面的“变量式”的，就更容易理解了：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS03.htm

```
<script type=" text/javascript" >
  var myfunc = function ()
  {
    alert( "hello" );
  };

  alert(typeof(myfunc));
</script>
```

这里明确定义了一个变量myfunc，它的初始值被赋予了一个function的实体。因此，typeof(myfunc)返回的也是function。其实，这两种函数的写法是等价的，除了一点

细微差别，其内部实现完全相同。也就是说，我们写的这些JavaScript函数只是一个命了名的变量而已，其变量类型即为function，变量的值就是我们编写的函数代码体。

聪明的你或许立即会进一步追问：既然函数只是变量，那么变量就可以被随意赋值并用到任意地方呢?

我们来看看下面的代码：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS04.htm

```
<script type=" text/javascript" >
  var myfunc = function ()
  {
    alert( "hello" );
  };
  myfunc(); //第一次调用myfunc，输出hello

  myfunc = function ()
  {
    alert( "yeah" );
  };
  myfunc(); //第二次调用myfunc，将输出yeah
</script>
```

这个程序运行的结果告诉我们：答案是肯定的！在第一次调用函数之后，函数变量又被赋予了新的函数代码体，使得第二次调用该函数时，出现了不同的输出。

好了，我们又来把上面的代码改成第一种定义式的函数形式：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS05.htm

```
<script type=" text/javascript" >
  function myfunc ()
  {
    alert( "hello" );
  };
  myfunc(); //这里调用myfunc，输出yeah而不是hello，奇怪吗?

  function myfunc ()
  {
    alert( "yeah" );
  };
  myfunc(); //这里调用myfunc，当然输出yeah
</script>
```

按理说，两个签名完全相同的函数，在其他编程语言中应该是非法的。但在

JavaScript中，这没错。不过，程序运行之后却发现一个奇怪的现象：两次调用都只是最后那个函数里输出的值！显然第一个函数没有起到任何作用。这又是为什么呢?

原来，JavaScript执行引擎并非一行一行地分析和执行程序，而是一段一段地分析执行的。而且，在同一段程序的分析执行中，定义式的函数语句会被提取出来优先执行。函数定义执行完之后，才会按顺序执行其他语句代码。也就是说，在第一次调用myfunc之前，第一个函数语句定义的代码逻辑，已被第二个函数定义语句覆盖了。所以，两次都调用都是执行最后一个函数逻辑了。

如果把这个JavaScript代码分成两段，将它们用<script>标签各自包含起来，如下面这两块代码：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS06.htm

```
<script type=" text/javascript" >
  function myfunc ()
  {
    alert( "hello" );
  };

  myfunc(); //这里调用myfunc，输出hello
</script>
```

```
<script type=" text/javascript" >
  function myfunc ()
  {
    alert( "yeah" );
  };

  myfunc(); //这里调用myfunc，输出yeah
</script>
```

这时，才是各自按顺序输出的，也证明了JavaScript的确是一段段地执行的。

一段代码中的定义式函数语句会优先执行，这似乎有点像静态语言的编译概念。所以，这一特征也被有些人称为：JavaScript的“预编译”。事实上呢，JavaScript执行引擎的预编译还包括对所有var变量的创建（初始值为undefined），以提高对代码的执行效率。

在大多数情况下，我们也没有必要去纠缠这些细节问题。只要你记住一点：JavaScript里的代码也是一种数据，同样可以被任意赋值和修改的，而它的值就是代码的逻辑。只是，与一般数据不同的是，函数是可以被调用执行的。

JavaScript函数的魔力就在于可以动态地改变代码的逻辑，仿佛孙猴子的七十二变。由于这种变化基于脚本解释机制，其灵活性远比静态语言的函数指针和虚函数要强大得多。灵活施展JavaScript函数的魔力，可以编写出强大而又简洁的代码。

接下来，我们讨论一下JavaScript代码在运动中的时空，进一步了解JavaScript代码的特点。

4
代码的时空

相对论告诉我们，时间和空间是不可分割的。我们只有把时间和空间结合起来才能确定一个事件发生的准确坐标。于是，时间和空间的结合，就形成了时空的概念。在一个时空中观察到的实验结果，对另一个时空的观察者是不适用的。

同样，对于过程式编程来说，代码执行的时间与数据标识的空间也是不可以分割的。我们只有把指令执行的具体时刻与标识映射的具体地址结合起来，才能确定程序在执行瞬间的上下文状态。于是，代码时刻与数据标识的结构，就形成了作用域的概念。在一个作用域中的上下文状态，对于另一个作用域来说是不适用的。

当定义了一个个标识符，写下一条条语句时，我们只是得到了程序的一个静态映像。还必须把这些元素放到动态的作用域环境去思考，才能理解整个程序的运行世界。下面我们来讨论一下JavaScript的作用域。

任何程序都会在一个原始的环境中开始运行，这个原始的环境就被称为全局环境。全局环境中包含了一些预定义的元素，这些元素对于我们的程序来说是自然存在的，它们本来就在那儿了，我们拿来即可使用。

在JavaScript里的全局环境就是一个对象，这个对象是JavaScript运行环境的根。对于浏览器中的JavaScript来说，这个根对象就是我们熟知的window对象（非浏览器宿主程序中可能不叫window）。对于全局的JavaScript语句来说，window对象就相当于当前作用域。

当我们写下：

```
var myName = “leadzen”;
```

就是定义了window作用域的一个变量myName，而当我们写下：

```
myName = “leadzen”;
```

就是定义了window对象的一个属性myName。

这里，“window作用域的一个变量myName”和“window对象的一个属性myName”几乎等价。对于全局的JavaScript语句来说，加不加“var”都无所谓，两种写法非常相似！

不过，对于一个函数体内的语句，有没有“var”就要小心了。

例如，下面的代码：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS07.htm

```
<script type=" text/javascript" >
  var yourName = "王菲";
  myName = "leadzen";

  alert(myName + " like " + yourName); //输出：leadzen like 王菲

  ChangeNames();  //调用改名函数

  function ChangeNames()  //改名函数，在函数体内改名
  {
    alert( 'Your old name is ' + yourName); //输出：Your old name is undefined
    alert( 'My old name is ' + myName); //输出：My old name is leadzen
    var yourName = "王靖雯";
    myName = "李战";
    alert(myName + " like " + yourName); //输出：李战 like 王靖雯
  };

  alert(myName + " like " + yourName); //输出：李战 like 王菲
</script>
```

显然用“var”修饰的yourName标识符在函数内外是两个东西，外面的“王菲”不会因为ChangeNames函数内改成“王靖雯”而改变，回到外面还是“王菲”。而myName没用“var”修饰，就是一个东西了，函数内的修改也就在函数外表现出来了。

值得注意的是，看ChangeNames函数中的第一句，我们本是希望输出最外面那个yourName的值，但此时的输出却表明yorName的值是undefined！而第二句输出myName又是我们期望的值。这是为什么呢？

原来，“var”定义的是作用域上的一个变量，而没有“var”的标识符却可能是全局根对象的一个属性。当代码运行在全局作用域时，作用域就是根对象window，所以，有没有“var”都无所谓。当然，不同的JavaScript执行引擎对此可能有不同的实现方式，但都可以大致这么来理解。

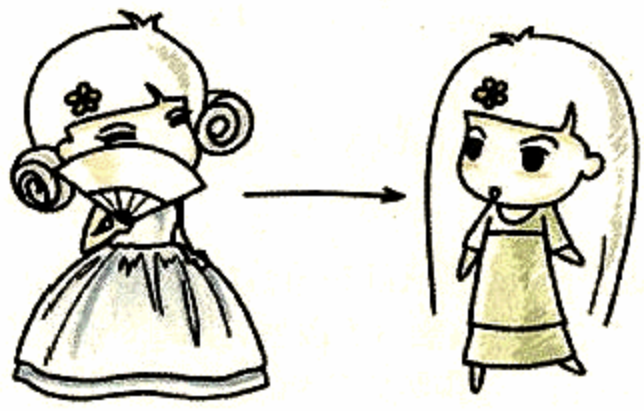
但当代码运行进入一个函数时，JavaScript会创建一个新的作用域，来作为当前作用域的子作用域。然后，将当前全局作用域切换为这个新建的子作用域，开始执行函数逻辑。JavaScript执行引擎会把此函数内的逻辑代码，当一个代码段单元来分析和执行。

在第一步的预编译分析中，JavaScript执行引擎将所有定义式函数直接创建为作用域上的函数变量，并将其值初始化为定义的函数代码逻辑，也就是为其建立了可调用的函数变量。而对于所有“var”定义的变量，也会在第一步的预编译中创建起来，并将初始值设为undefined。

随后，JavaScript开始解释执行代码。当遇到对函数名或变量名的使用时，JavaScript执行引擎会首先在当前作用域查找函数或变量，如果没有就到上层作用域查找，依次类推。因此，前面的语句引用后面语句定义的“var”变量时，该变量其实已经存在，只是初始值为undefined。

因此，用“var”定义的变量只对本作用域有效，尽管此时上层作用域有同名的东西，都与本作用域的“var”变量无关。退出本作用域之后，此“var”消失，回到原来的作用域该有啥就还有啥。

王菲就是王菲啊，不管王靖雯这个名字多么好，那都是在那个作用域里的临时名字。她还是喜欢父亲给取的王菲这个名字，当回到自己可以控制的作用域时，当然就得改回来。知道王菲改名这一故事的，多半是老程序员，这是闲话。



其实，函数在每次调用时都会产生一个子作用域，退出函数时，这个子作用域就消失，下次调用相同函数时又是另一个子作用域了。在运行的函数内再调用另外的函数时，又产生另一个作用域，这就会随着函数调用的深入自然形成一种叫“作用域链”的东西，甚至在递归调用的情况下，作用域链会变得很长很长。

JavaScript查找标识符时，除非指明特定对象，否则会沿着作用域链寻找符合这一标识符的变量或属性，甚至会自动创建该标识符。正是由于JavaScript的这种灵活性，我们就可能在不经意间覆盖了原来的变量或属性，或者无意中创建了大量无用的全局属性，甚至在毫不知情的情况下影响了其他代码的逻辑，从而造成许多很难排出的Bug。

因此，当我们编写和调试JavaScript代码时，我们的思绪应该尽量放到具体的作用域时空上去思考。这样，我们就能明白为什么标识符尽量别与已有的东西重名？为什么变量一定要加“var”？为什么访问属性一定要指明对象？理解和注意这些问题，也就能轻易地排出某些奇怪的Bug。多一些死记硬背不如多一点理解，多一些技巧方法不如多一点思想。

虽然，我们的代码没有办法访问到JavaScript内部的作用域对象，但JavaScript还是给我们提供了与作用域相关的某些可用元素。通过这些元素，我们还是能知道当前作用域上下文中的某系信息。JavaScript提供的调用上下文信息有几个：一个是函数本身，一个是函数的caller属性，另外还有this关键字和arguments隐含对象。

当代码运行在函数体内的作用域时空时，函数本身的标识符是可以用的。因为前面我们讨论过，函数本身也是一种数据嘛。比如下面的代码就可以输出函数自身的定义：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS09.htm

```
<script type=" text/javascript" >
  function aFunc()
  {
    alert(aFunc.toString());   //函数体内可以使用自身的标识符
  };

  aFunc();   //调用函数
</script>
```

更进一步，函数自身有一个caller属性，其表示调用当前函数的上层函数。这就提供了一个可以追溯函数调用来源的线索，特别对于调试JavaScript来说是不可多得的宝贝。下面的代码可以帮助我们理解caller属性的含义：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS09.htm

```
<script type=" text/javascript" >
  function WhoCallMe()
  {
    alert( "My caller is " + WhoCallMe.caller); //输出自己的caller
  };

  function CallerA(){WhoCallMe();};

  function CallerB(){WhoCallMe();};

  alert(WhoCallMe.caller) //输出：null

  WhoCallMe();   //输出：My caller is null
  CallerA();     //输出：My caller is function CallerA(){WhoCallMe();};
```

```
    CallerB();    //输出：My caller is function CallerB(){WhoCallMe();};
</script>
```

如果函数的caller属性是Null，表示函数没有被调用或者是被全局代码调用。函数的caller属性值实际上是动态变化的。函数的初始caller值都是null，当调用一个函数时，如果代码已经运行在某个函数体内，JavaScript执行引擎会将被函数的caller属性设置为当前函数。在退出被调函数作用域时，被调函数的caller属性又会恢复为null值。

灵活运用函数自身标识和caller属性，可以帮助我们简化代码或写一些通用的处理代码。许多优秀的AJAX框架也都充分利用了JavaScript的这些特性。不过，有点遗憾的是Opera浏览器目前不支持caller属性，但IE， Firefox 和 Safari浏览器都支持。

this关键字表示函数正在服务的“这个”对象，这个话题比较多，我们放到后面单独讨论吧。

arguments对象嘛，从字面上看是表示调用当前函数的参数们。除了我们可以直接使用函数参数定义列表中的那些标识符来访问之外，还可以用arguments对象按数组方式来访问参数，尽管arguments对象并非一个真正的数组。

有一个值得注意的情况是：用eval()函数动态执行的代码并不创建新的作用域，其代码就是在当前作用域中执行的。eval()中的动态代码可以访问到当前作用域的this, arguments等对象，因此可以使用eval()实现一些高级的多态和动态扩展方面的应用。

另外，也不要再使用arguments对象的caller和callee属性了。虽然通过它们也能知道上层调用函数和被调用函数自身，但arguments对象的这两个属性都早已被标记为“作废”，说不定哪天就彻底消失了。

不过，能在运行中判断自己前后调用的代码关系，这是非常有趣的事。这使得JavaScript的函数幻化能力比静态语言要强大得多！然而， JavaScript函数的神奇之处还体现在另外两个方面：一是函数function类型本身也具有对象化的能力，二是函数function与对象 object超然的结合能力。

5
奇妙的对象

前面我们说，在JavaScript中只有object和function两种东西才有对象化的能力。现在，我们先来说说函数的对象化能力。

任何一个函数都可以为其动态地添加或去除属性，这些属性可以是简单类型，可以是对象，也可以是其他函数。也就是说，函数具有对象的全部特征，你完全可以把函数当对象来用。其实，函数就是对象，只不过比一般的对象多了一个括号“()”操作符，这个操作符用来执行函数的逻辑，即函数本身还可以被调用，一般对象却不可以被调用，除此之外完全相同。请看下面的代码：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS10.htm

```
<script type=" text/javascript" >
  function Sing()
  {
    alert(Sing.author + "：" + Sing.poem);
  };

  Sing.author = "李白";
  Sing.poem = "汉家秦地月，流影照明妃。一上玉关道，天涯去不归...";
  Sing();

  Sing.author = "李战";
  Sing.poem = "日出汉家天，月落阴山前。女儿琵琶怨，已唱三千年...";
  Sing();
</script>
```

在这段代码中，Sing函数被定义后，又给Sing函数动态地增加了author和poem属性。将author和poem属性设为不同的作者和诗句，在调用Sing()时就能显示出不同的结果。这个示例用一种诗情画意的方式，让我们理解了JavaScript函数就是对象的本质，也感受到了 JavaScript语言的优美。

好了，以上的讲述，我们应该算理解了function类型的东西都是和object类型一样的东西，这种东西被我们称为“对象”。我们的确可以这样去看待这些“对象”，因

为它们既有“属性”也有“方法”嘛。但下面的代码又会让我们产生新的疑惑：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS11.htm

```
<script type=" text/javascript" >
   var anObject = {}; //一个对象
   anObject.aProperty = "Property of object"; //对象的一个属性
   anObject.aMethod = function(){alert("Method of object")}; //对象的一个方法

   //主要看下面：
   alert(anObject["aProperty"]); //可以将对象当数组以属性名作为下标来访问属性
   anObject["aMethod"]();        //可以将对象当数组以方法名作为下标来调用方法

   for( var s in anObject)        //遍历对象的所有属性和方法进行迭代化处理
      alert(s + " is a " + typeof(anObject[s]));
</script>
```

对于function类型的对象也是一样：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS12.htm

```
<script type=" text/javascript" >
   var aFunction = function() {}; //一个函数
   aFunction.aProperty = "Property of function"; //函数的一个属性
   aFunction.aMethod = function(){alert("Method of function")}; //函数的一个方法

   //主要看下面：
   alert(aFunction["aProperty"]); //可以将函数当数组以属性名作为下标来访问属性
   aFunction["aMethod"]();        //可以将函数当数组以方法名作为下标来调用方法

   for( var s in aFunction)        //遍历函数的所有属性和方法进行迭代化处理
      alert(s + " is a " + typeof(aFunction[s]));
</script>
```

是的，对象和函数可以如数组一样，用属性名或方法名作为下标来访问并处理。那么，它到底应该算是数组，还是算对象？

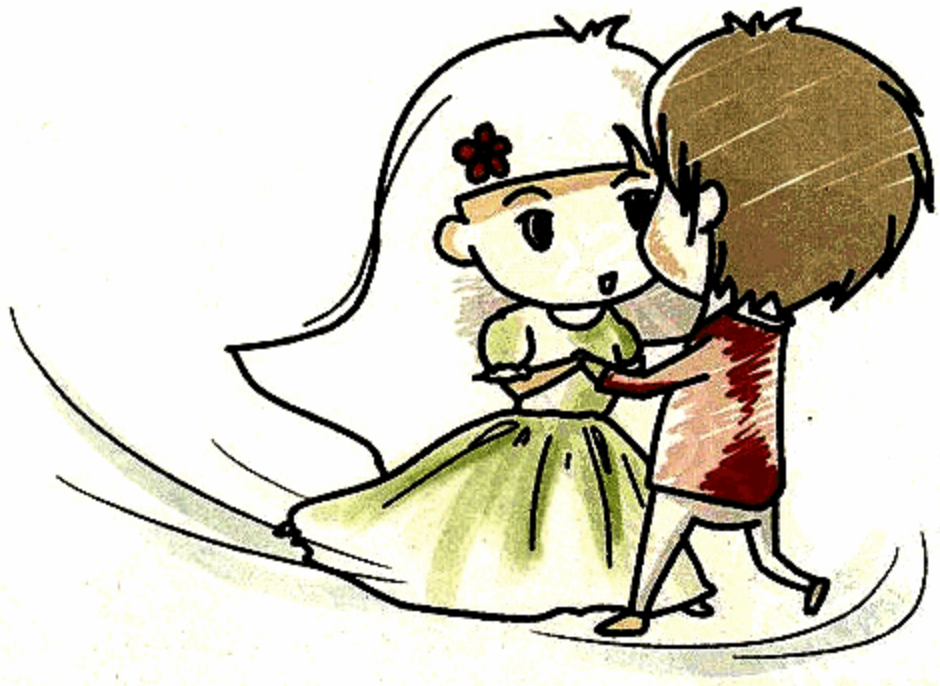
我们知道，数组应该算是线性数据结构，线性数据结构一般有一定的规律，适合进行统一的批量迭代操作等，有点像波。而对象是离散数据结构，适合描述分散和个性化东西，有点像粒子。因此，我们也可以这样问：JavaScript里的对象到底是波还是粒子？

如果存在对象量子论，那么答案一定是：波粒二象性！

因此，JavaScript里的函数和对象既有对象的特征也有数组的特征。这里的数组被称为“字典”，一种可以任意伸缩的名称值对儿的集合。其实， object和function的内

部实现就是一个字典结构，但这种字典结构却通过严谨而精巧的语法表现出了丰富的外观。

正如量子力学在一些地方用粒子来解释和处理问题，而在另一些地方却用波来解释和处理问题。你也可以在需要的时候，自由选择用对象还是数组来解释和处理问题。只要把握和善于运用JavaScript的这些奇妙特性，就可以编写出很多简洁而强大的代码来。



6
放下对象

我们再来看看function与object的超然结合吧。

在面向对象的编程世界里，数据与代码的有机结合就构成了对象的概念。自从有了对象，编程世界就被划分成两部分：一个是对象内的世界，一个是对象外的世界。对象天生具有自私的一面，外面的世界未经允许是不可访问对象内部的。对象也有大方的一面，它对外提供属性和方法，也为他人服务。不过，在这里我们要谈到一个有趣的问题，就是“对象的自我意识”。

什么？没听错吧？对象有自我意识？

可能对许多程序员来说，这的确是第一次听说。不过，请君看看C++、C#和Java的this，Delphi的self，还有VB的me，或许你会恍然大悟！当然，也可能只是说句“不过如此”而已。

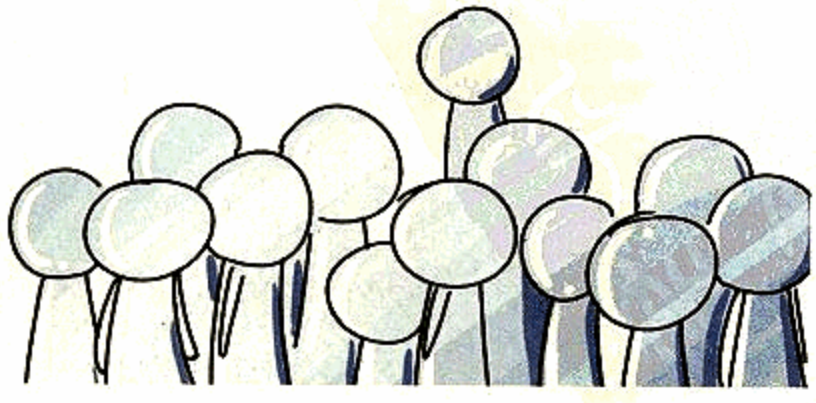
然而，就在对象将世界划分为内外两部分的同时，对象的“自我”也就随之产生。“自我意识”是生命的最基本特征！正是由于对象这种强大的生命力，才使得编程世界充满无限的生机和活力。

但对象的“自我意识”在带给我们快乐的同时也带来了痛苦和烦恼。我们给对象赋予了太多欲望，总希望它们能做更多的事情。然而，对象的自私使得它们互相争抢系统资源，对象的自负让对象变得复杂和臃肿，对象的自欺也往往带来挥之不去的错误和异常。我们为什么会有这么多的痛苦和烦恼呢？

为此，有一个人，在对象树下，整整想了九九八十一天，终于悟出了生命的痛苦来自于欲望，但究其欲望的根源是来自于自我意识。

于是他放下了“自我”，在对象树下成了佛。

从此他开始普度众生，传播真经。他的名字就叫释迦摩尼，而《JavaScript真经》正是他所传经书中的一本。



JavaScript中也有this，但这个this却与C++、C#或Java等语言的this不同。一般编程语言的this就是对象自己，而 JavaScript的this却并不一定！this可能是我，也可能是你，可能是他，反正是我中有你，你中有我，这就不能用原来的那个“自我”来理解JavaScript这个this的含义了。为此，我们必须首先放下原来对象的那个“自我”。

我们来看下面的代码：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS13.htm

```
<script type=" text/javascript" >
  function WhoAmI()      //定义一个函数WhoAmI
  {
    alert( "I' m  " + this.name +  " of  " + typeof(this));
  };

  WhoAmI();  //此时this是根对象window，其name属性为空字符串。
             //输出：I' m of object

  var BillGates = {name:  "Bill Gates" };
  BillGates.WhoAmI = WhoAmI;                    //将函数WhoAmI作为BillGates的方法。
  BillGates.WhoAmI();                           //this是BillGates。
                                                //输出：I' m Bill Gates of object

  var SteveJobs = {name:  "Steve Jobs" };
  SteveJobs.WhoAmI = WhoAmI;                    //将函数WhoAmI作为SteveJobs的方法。
  SteveJobs.WhoAmI();                           //this是SteveJobs。
                                                //输出：I' m Steve Jobs of object

  WhoAmI.call(BillGates);                       //直接将BillGates作为this，调用WhoAmI。
                                                //输出：I' m Bill Gates of object

  WhoAmI.call(SteveJobs);                       //直接将SteveJobs作为this，调用WhoAmI。
                                                //输出：I' m Steve Jobs of object

  BillGates.WhoAmI.call(SteveJobs);             //将SteveJobs作为this，却调用BillGates的方法。
                                                //输出：I' m Steve Jobs of object

  SteveJobs.WhoAmI.call(BillGates);             //将BillGates作为this，却调用SteveJobs的方法。
                                                //输出：I' m Bill Gates of object

  WhoAmI.WhoAmI = WhoAmI;                       //将WhoAmI函数设置为自身的方法。
  WhoAmI.name =  "WhoAmI" ;
  WhoAmI.WhoAmI();                              //此时的this是WhoAmI函数自己。
                                                //输出：I' m WhoAmI of function

  {name:  "nobody" , WhoAmI: WhoAmI}.WhoAmI();  //创建一个匿名对象并调用其方法。
                                                //输出：I' m nobody of object
  </script>
```

从上面的代码可以看出，同一个函数可以从不同的角度来调用，this并不一定是函数本身所属的对象。this只是在任意对象和function元素结合时的一个概念，这种结合比起一般对象语言的默认结合更加灵活，显得更加超然和洒脱。

在JavaScript函数中，你只能把this看成当前要服务的“这个”对象。this是一个特殊的内置参数，根据this参数，您可以访问到“这个”对象的属性和方法，但却不能给this参数赋值。在一般对象语言中，方法体代码中的this可以省略，成员默认都首先是“自己”的。但 JavaScript却不同，由于不存在“自我”，当访问“这个”对象时，this不可省略！

JavaScript提供了传递this参数的多种形式和手段，其中，像BillGates.WhoAmI()和SteveJobs.WhoAmI()这种形式，是传递this参数最正规的形式，此时的this就是函数所属的对象本身。而大多数情况下，我们也几乎很少去采用那些借花仙佛的调用形式。但我们要明白，JavaScript的这个“自我”与其他编程语言的“自我”是不同的，这是一个放下了的“自我”。这就是JavaScript特有的世界观！

最后要提醒大家一下，我们在用JavaScript编写网页脚本时，也经常会用到一个叫self的属性。特别是熟悉Delphi的朋友对这个self更是情有独钟，但它不是Delphi的那个self概念。这里的self只是表示网页结构的当前window对象，以及FRAME或IFRME元素的window对象，与我们这里说的对象自我是两回事情，大家别搞混了哟。

7
对象素描

已经说了许多许多话题，但有一个很基本的问题我们忘了讨论，那就是：怎样建立对象？

在前面的示例中，已经涉及对象的建立了。我们使用了一种被称为JavaScript Object Notation（缩写JSON）的形式，翻译为中文就是“JavaScript对象表示法”。

JSON为创建对象提供了非常简单的方法。例如，

创建一个没有任何属性的对象：

```
var o = {};
```

创建一个对象并设置属性及初始值：

```
var person = {name: "Angel", age: 18, married: false};
```

创建一个对象并设置属性和方法：

```
var speaker = {text: "Hello World", say: function(){alert(this.text)}};
```

创建一个更复杂的对象，嵌套其他对象和对象数组等：

```
var company =
```

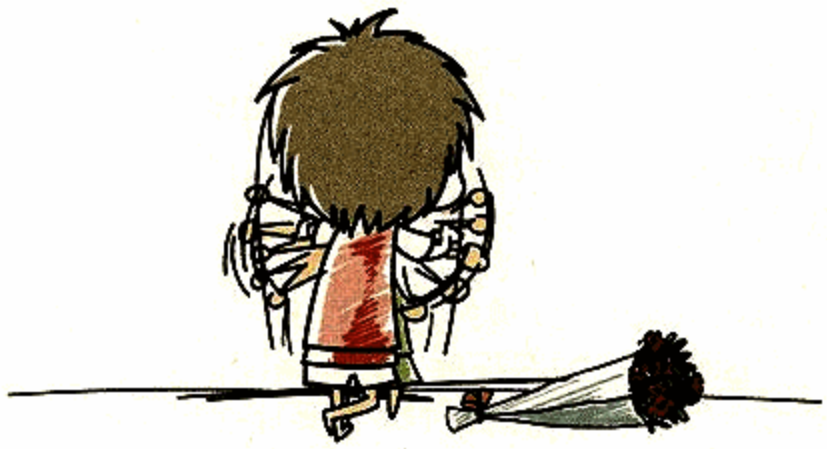
```
{
   name: “Microsoft”,
   product: “softwares”,
   chairman: {name: “Bill Gates”, age: 53},
   employees: [{name: “Angel”, age: 26}, {name: “Hanson”, age: 32}],
   readme: function() {document.write(this.name + “ product “ + this.product);}
};
```

JSON的形式就是用大括“{}”号包括起来的项目列表，每一个项目间并用逗号“,”分隔，而项目就是用冒号“:”分隔的属性名和属性值。这是典型的字典表示形式，也再次表明了JavaScript里的对象就是字典结构。不管多么复杂的对象，都可以被一句JSON代码来创建并赋值。

其实，JSON就是JavaScript对象最好的序列化形式，它比XML更简洁，也更省空间。对象可以作为一个JSON形式的字符串，在网络间自由传递和交换信息。而当需要将这个JSON字符串变成一个JavaScript对象时，只需要使用eval函数这个强大的数码转换引擎，就立即能得到一个 JavaScript内存对象。正是由于JSON的这种简单朴素的天生丽质，才使得她在AJAX舞台上成为璀璨夺目的明星。

JavaScript就是这样，把面向对象那些看似复杂的东西，用极其简洁的形式表达出来。卸下对象浮华的浓妆，还对象一个眉目清晰！

8
构造对象



好了，接下我们来讨论一下对象的另一种创建方法。

除JSON外，在JavaScript中我们可以使用new操作符结合一个函数的形式来创建对象。例如：

```
function MyFunc() {};      //定义一个空函数
var anObj1 = new MyFunc();  //使用new操作符，借助MyFun函数，就创建了一个对象
var anObj2 = new MyFunc;   //函数也可以没有括号，但仍将调用该函数！
```

JavaScript的这种创建对象的方式可真有意思，如何去理解这种写法呢？

其实，可以把上面的代码改写成这种等价形式：

```
function MyFunc(){};
var anObj = {};    //创建一个对象
MyFunc.call(anObj); //将anObj对象作为this指针调用MyFunc函数
```

我们就可以这样理解，JavaScript先用new操作符创建了一个对象，紧接着就将这个对象作为this参数调用了后面的函数。其实， JavaScript内部就是这么做的，而且任何函数都可以被这样调用！但从 “anObj = new MyFunc()” 这种形式，我们又看到一个熟悉的身影，C++和C#不就是这样创建对象的吗？原来，条条大路通灵山，殊途同归啊！

君看到此处也许会想，我们为什么不可以把这个MyFunc当做构造函数呢？恭喜你，答对了！JavaScript也是这么想的！请看下面的代码：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS14.htm

```
<script type=" text/javascript" >
  function Person(name)  //带参数的构造函数
  {
    this.name = name;  //定义并初始化name属性

    this.SayHello = function()  //定义对象方法SayHello
    {
      alert( "Hello, I' m " + this.name);
    };
  };

  function Employee(name, salary) //子构造函数
  {
    Person.call(this, name);  //调用父构造函数

    this.salary = salary;     //添加并初始化salary属性

    this.ShowMeTheMoney = function()   //添加对象方法ShowMeTheMoney
    {
      alert(this.name + " $" + this.salary);
    };
  };

  var BillGates = new Person( "Bill Gates" );          //创建Person类的BillGates对象
  var SteveJobs = new Employee( "Steve Jobs" , 1234);  //创建Empolyee类的SteveJobs对象

  BillGates.SayHello();  //输出：I' m Bill Gates
  SteveJobs.SayHello();  //输出：I' m Steve Jobs
  SteveJobs.ShowMeTheMoney();  //输出：Steve Jobs $1234

  alert(BillGates.constructor == Person);    //输出：true
  alert(SteveJobs.constructor == Employee);  //输出：true

  alert(BillGates.SayHello == SteveJobs.SayHello); //输出：false
</script>
```

这段代码表明，函数不但可以当作构造函数，而且还可以带参数，还可以为对象添加成员和方法。其中，Employee构造函数又将自己接收的 this作为参数调用Person构造函数，这就是相当于调用基类的构造函数。输出true的那两行代码，还表明这样一个意思：BillGates是由Person构造的，而SteveJobs是由Employee构造的。对象内置的constructor属性还指明了构造对象所用的具体函数！

其实，如果你愿意把函数当作“类”的话，她就是“类”，因为她本来就有“类”的那些特征。难道不是吗？她生出的儿子各个都有相同的特征，而且构造函数

也与类同名嘛！

但要注意的是，用构造函数操作this对象创建出来的每一个对象，不但具有各自的成员数据，而且还具有各自的方法数据。换句话说，方法的代码体（体现函数逻辑的数据）在每一个对象中都存在一个副本。尽管每一个代码副本的逻辑是相同的，但对象们确实是各自保存了一份代码体。上例中的最后一行代码输出false，说明了这一事实：两个对象的SayHello是不同的。这也解释了JavaScript中的函数就是对象的概念。

同一类的对象各自有一份方法代码显然是一种浪费。在传统的对象语言中，方法函数并不像JavaScript那样是个对象概念。即使也有像函数指针、方法指针或委托那样的变化形式，但其实质也是对同一份代码的引用。一般的对象语言很难遇到这种情况。

不过，JavaScript语言有大的灵活性。我们可以先定义一份唯一的方法函数体，并在构造this对象时使用这唯一的函数对象作为其方法，就能共享方法逻辑。例如：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS15.htm

```
<script type=" text/javascript" >
  function SayHello()    //定义全局SayHello函数
  {
    alert( "Hello, I' m " + this.name);
  };

  function Person(name)
  {
    this.name = name;
    this.SayHello = SayHello;  //设置SayHello方法为全局SayHello函数
  };

  var BillGates = new Person( "Bill Gates" );
  var SteveJobs = new Person( "Steve Jobs" );

  alert(BillGates.SayHello == SteveJobs.SayHello); //输出：true
</script>
```

其中，最后一行的输出结果表明两个对象确实共享了一个函数对象。虽然，这段程序达到了共享一份方法代码的目的，却不怎么优雅。因为，定义 SayHello方法时反映不出其与Person类的关系。“优雅”这个词用来形容代码，也不知道是谁先提出来的。不过，这个词反映了程序员已经从追求代码的正确、高效、可靠和易读等基础上，向着追求代码的美观感觉和艺术境界的层次发展，程序人生又多了些浪漫色彩。

显然，JavaScript早想到了这一问题，她的设计者们为此提供了一个有趣的prototype概念。

9
初看原型

prototype源自法语，软件界的标准翻译为“原型”，代表事物的初始形态，也含有模型和样板的意义。JavaScript中的prototype概念恰如其分地反映了这个词的内涵，我们不能将其理解为C++的prototype那种预先声明的概念。

JavaScript的所有function类型的对象都有一个prototype属性。这个prototype属性本身又是一个object类型的对象，因此我们也可以给这个prototype对象添加任意的属性和方法。既然prototype是对象的“原型”，那么由该函数构造出来的对象应该都会具有这个“原型”的特性。事实上，在构造函数的prototype上定义的所有属性和方法，都是可以通过其构造的对象直接访问和调用的。也可以这么说， prototype提供了一群同类对象共享属性和方法的机制。

我们先来看看下面的代码：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS16.htm

```
<script type=" text/javascript" >
 <script type=" text/javascript" >
   function Person(name)
   {
      this.name = name;  //设置对象属性，每个对象各自有一份属性数据
   };

   Person.prototype.SayHello = function()  //给Person函数的prototype添加SayHello方法。
   {
      alert( "Hello, I' m " + this.name);
   }

   var BillGates = new Person( "Bill Gates" );  //创建BillGates对象
   var SteveJobs = new Person( "Steve Jobs" );  //创建SteveJobs对象

   BillGates.SayHello();  //通过BillGates对象直接调用到SayHello方法
   SteveJobs.SayHello();  //通过SteveJobs对象直接调用到SayHello方法

   alert(BillGates.SayHello == SteveJobs.SayHello); //输出：true
</script>
```

程序运行的结果表明，构造函数的prototype上定义的方法确实可以通过对象直接调用到，而且代码是共享的。显然，把方法设置到prototype的写法显得优雅多了，尽管调用形式没有变，但逻辑上却体现了方法与类的关系，相对前面的写法，更容易理解和组织代码。

那么，对于多层次类型的构造函数情况又如何呢？

我们再来看下面的代码：

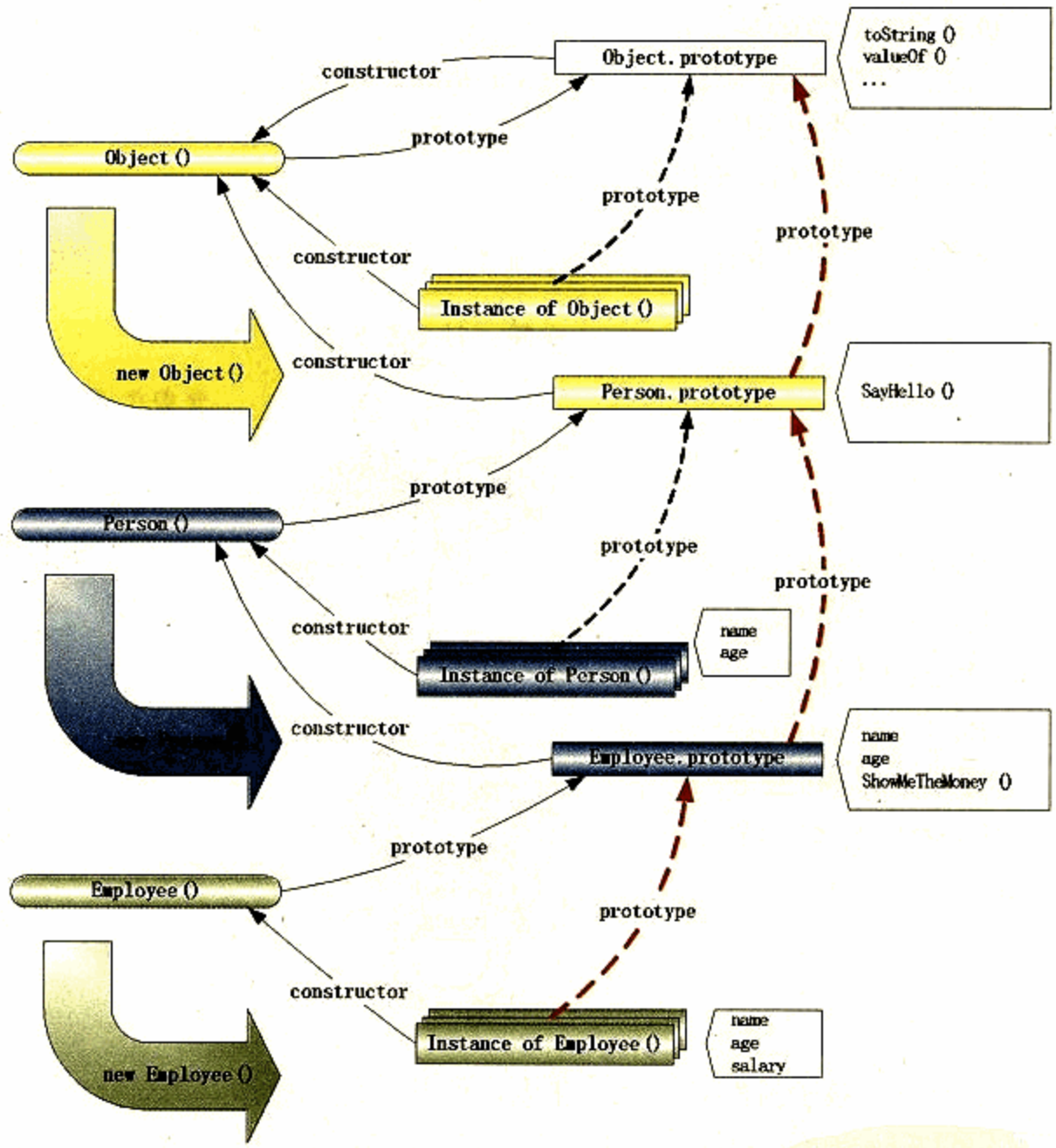
http://www.leadzen.cn/Books/WuTouJavaScript/1/JS17.htm

```
<script type=" text/javascript" >
  function Person(name)  //基类构造函数
  {
    this.name = name;
  };

  Person.prototype.SayHello = function()  //给基类构造函数的prototype添加方法
  {
    alert( "Hello, I' m  " + this.name);
  };

  function Employee(name, salary) //子类构造函数
  {
    Person.call(this, name);   //调用基类构造函数
    this.salary = salary;
  };

  Employee.prototype = new Person();  //建一个基类的对象作为子类原型的原型(原型继承)

  Employee.prototype.ShowMeTheMoney = function()  //给子类构造函数的prototype添加方法
  {
    alert(this.name +  " $"  + this.salary);
  };

  var BillGates = new Person( "Bill Gates" );  //创建基类Person的BillGates对象
  var SteveJobs = new Employee( "Steve Jobs" , 1234);  //创建子类Employee的SteveJobs对象

  BillGates.SayHello();     //通过对象直接调用到prototype的方法
  SteveJobs.SayHello();     //通过子类对象直接调用基类prototype的方法，关注！
  SteveJobs.ShowMeTheMoney(); //通过子类对象直接调用子类prototype的方法

  alert(BillGates.SayHello == SteveJobs.SayHello); //输出：true
</script>
```

这段代码的这一句：Employee.prototype = new Person()，构造了一个基类的对象，并将其设为子类构造函数的prototype，这是很有意思的。这样做的目的就是为了后

面，通过子类对象也可以直接调用基类prototype的方法。为什么可以这样呢？

我们先来看看下面这个原型示意图，注意其中红色虚线：



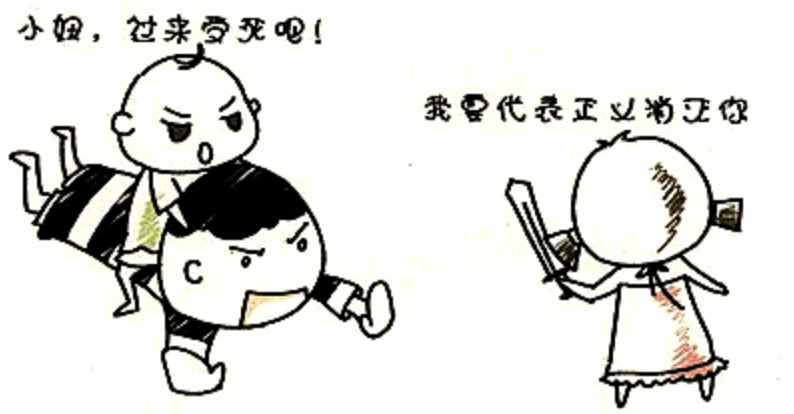
原来，在JavaScript中，prototype不但能让对象共享自己财富，而且prototype还有寻根问祖的天性，从而使得先辈们的遗产可以代代相传。当从一个对象那里读取属性或调用方法时，如果该对象自身不存在这样的属性或方法，就会去自己关联的prototype对象那里寻找；如果 prototype没有，又会去prototype自己关联的前辈prototype那里寻找，直到找到或追溯过程结束为止。

在JavaScript内部，对象的属性和方法追溯机制是通过所谓的prototype链来实现的。当用new操作符构造对象时，也会同时将构造函数的 prototype对象指派给新创建的对象，成为该对象内置的原型对象。对象内置的原型对象应该是对外不可见的，尽管

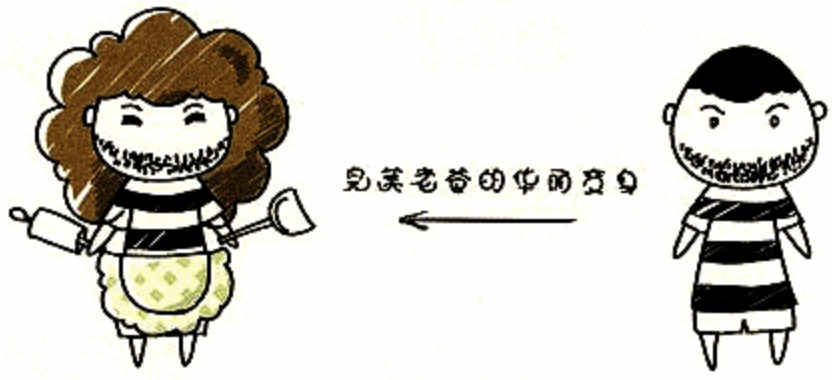
有些浏览器（如Firefox）可以让我们访问这个内置原型对象，但并不建议这样做。内置的原型对象本身也是对象，也有自己关联的原型对象，这样就形成了所谓的原型链。

在原型链的最末端，就是Object构造函数prototype属性指向的那一个原型对象。这个原型对象是所有对象的最老祖先，这个老祖宗实现了诸如 toString等所有对象天生就该具有的方法。其他内置构造函数，如Function, Boolean, String, Date和RegExp等的prototype都是从这个老祖宗传承下来的，但他们各自又定义了自身的属性和方法，从而他们的子孙就表现出各自宗族的那些特征。

这不就是“继承”吗？是的，这就是“继承”，是JavaScript特有的“原型继承”。



“原型继承”是慈祥而又严厉的。原形对象将自己的属性和方法无私地贡献给孩子们使用，也并不强迫孩子们必须遵从，允许一些顽皮孩子按自己的兴趣和爱好独立行事。从这点上看，原型对象是一位慈祥的母亲。



然而，任何一个孩子虽然可以我行我素，但却不能动原型对象既有的财产，因为那可能会影响到其他孩子的利益。从这一点上看，原型对象又像一位严厉的父亲。

我们来看看下面的代码就可以理解这个意思了：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS18.htm

```
<script type=" text/javascript" >
  function Person(name)
  {
    this.name = name;
  };

  Person.prototype.company = "Microsoft"; //原型的属性

  Person.prototype.SayHello = function()  //原型的方法
  {
    alert("Hello, I' m " + this.name + " of " + this.company);
  };

  var BillGates = new Person("Bill Gates");
  BillGates.SayHello();  //由于继承了原型的东西，规规矩矩输出：Hello, I' m Bill Gates

  var SteveJobs = new Person("Steve Jobs");
  SteveJobs.company = "Apple";  //设置自己的company属性，掩盖了原型的company属性
  SteveJobs.SayHello = function() //实现了自己的SayHello方法，掩盖了原型的SayHello方法
  {
    alert("Hi, " + this.name + " like " + this.company + ", ha ha");
  };

  SteveJobs.SayHello();  //都是自己覆盖的属性和方法，输出：Hi, SteveJobs like Apple, ha ha

  BillGates.SayHello();  //SteveJobs的覆盖没有影响原型对象，BillGates还是按老样子输出
</script>
```

对象可以掩盖原型对象的那些属性和方法，一个构造函数原型对象也可以掩盖上层构造函数原型对象既有的属性和方法。这种掩盖其实只是在对象自己身上创建了新的属性和方法，只不过这些属性和方法与原型对象的那些同名而已。JavaScript就是用这简单的掩盖机制实现了对象的“多态”性，与静态对象语言的虚函数和重载(override)概念不谋而合。

然而，比静态对象语言更神奇的是，我们可以随时给原型对象动态添加新的属性和方法，从而动态地扩展基类的功能特性。这在静态对象语言中是很难想像的。我们来看下面的代码：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS19.htm

```
<script type=" text/javascript" >
  function Person(name)
```

```
{
    this.name = name;
};

Person.prototype.SayHello = function()  //建立对象前定义的方法
{
    alert( "Hello, I' m  " + this.name);
};

var BillGates = new Person( "Bill Gates" );  //建立对象

BillGates.SayHello();

Person.prototype.Retire = function()   //建立对象后再动态扩展原型的方法
{
    alert( "Poor  " + this.name +  ", bye bye!" );
};

BillGates.Retire(); //动态扩展的方法即可被先前建立的对象立即调用
</script>
```

阿弥陀佛，原型继承竟然可以玩出有这样的法术！

10
原型扩展

想必君的悟性极高，可能你会这样想：如果在JavaScript内置的那些如Object和Function等函数的prototype上添加些新的方法和属性，是不是就能扩展JavaScript的功能呢？

那么，恭喜你，你得到了！

在AJAX技术迅猛发展的今天，许多成功的AJAX项目的JavaScript运行库都大量扩展了内置函数的prototype功能。比如微软的 ASP.NET AJAX，就给这些内置函数及其prototype添加了大量的新特性，从而增强了JavaScript的功能。

我们来看一段摘自MicrosoftAjax.debug.js中的代码：

```
String.prototype.trim = function String$trim() {
    if (arguments.length !== 0) throw Error.parameterCount();
    return this.replace(/^\s+|\s+$/g, '');
}
```

这段代码就是给内置String函数的prototype扩展了一个trim方法，于是所有String类对象都有了trim方法。有了这个扩展，今后要去除字符串两段的空白，就不用再分别处理了，因为任何字符串都有了这个扩展功能，只要调用即可，真的很方便。

当然，几乎很少有人去给Object的prototype添加方法，因为那会影响到所有的对象，除非在你的架构中这种方法的确是所有对象都需要的。

前两年，微软在设计AJAX类库的初期，用了一种被称为“闭包”（closure）的技术来模拟“类”。其大致模型如下：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS20.htm

```
<script type=" text/javascript" >
    function Person(firstName, lastName, age)
    {
```

```
    //私有变量：
    var _firstName = firstName;
    var _lastName = lastName;

    //公共变量:
    this.age = age;

    //方法：
    this.getName = function()
    {
       return(firstName + " " + lastName);
    };
    this.SayHello = function()
    {
       alert("Hello, I' m " + firstName + " " + lastName);
    };
};

var BillGates = new Person("Bill", "Gates", 53);
var SteveJobs = new Person("Steve", "Jobs", 53);

BillGates.SayHello();
SteveJobs.SayHello();
alert(BillGates.getName() + " " + BillGates.age);
alert(BillGates._firstName);    //这里不能访问到私有变量
</script>
```

很显然，这种模型的类描述特别像C#语言的描述形式，在一个构造函数里依次定义了私有成员、公共属性和可用的方法，显得非常优雅嘛。特别是“闭包”机制可以模拟对私有成员的保护机制，做得非常漂亮。

所谓“闭包”，就是在构造函数体内定义另外的函数作为目标对象的方法函数，而这个对象的方法函数反过来引用外层函数体中的临时变量。这使得只要目标对象在生存期内始终能保持其方法，就能间接保持原构造函数体当时用到的临时变量值。尽管最开始的构造函数调用已经结束，临时变量的名称也都消失了，但在目标对象的方法内却始终能引用到该变量的值，而且该值只能通这种方法来访问。即使再次调用相同的构造函数，只会生成新对象和方法，新的临时变量只是对应新的值，和上次那次调用的是各自独立的。的确很巧妙！

但是前面我们说过，给每一个对象设置一份方法是一种很大的浪费。还有，“闭包”这种间接保持变量值的机制，往往会给JavaSript的垃圾回收器制造难题。特别是遇到对象间复杂的循环引用时，垃圾回收的判断逻辑非常复杂。无独有偶，IE浏览器早期版本确实存在JavaSript垃圾回收方面的内存泄漏问题。再加上“闭包”模型在性能测试

方面的表现不佳，微软最终放弃了“闭包”模型，而改用“原型”模型。正所谓“有得必有失”嘛。

原型模型需要一个构造函数来定义对象的成员，而方法却依附在该构造函数的原型上。大致写法如下：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS21.htm

```
<script type=" text/javascript" >
  //定义构造函数
  function Person(name)
  {
    this.name = name;  //在构造函数中定义成员
  };

  //方法定义到构造函数的prototype上
  Person.prototype.SayHello = function()
  {
    alert( "Hello, I' m  " + this.name);
  };

  //子类构造函数
  function Employee(name, salary)
  {
    Person.call(this, name);   //调用上层构造函数
    this.salary = salary;      //扩展的成员
  };

  //子类构造函数首先需要用上层构造函数来建立prototype对象，实现继承的概念
  Employee.prototype = new Person()  //只需要其prototype的方法，其他东西没有任何意义！

  //子类方法也定义到构造函数之上
  Employee.prototype.ShowMeTheMoney = function()
  {
    alert(this.name + " $" + this.salary);
  };

  var BillGates = new Person( "Bill Gates" );
  BillGates.SayHello();

  var SteveJobs = new Employee( "Steve Jobs" , 1234);
  SteveJobs.SayHello();
  SteveJobs.ShowMeTheMoney();
</script>
```

原型类模型虽然不能模拟真正的私有变量，而且也要分两部分来定义类，显得不

怎么“优雅”。不过，对象间的方法是共享的，不会遇到垃圾回收问题，而且性能优于“闭包”模型。正所谓“有失必有得”嘛。

在原型模型中，为了实现类继承，必须首先将子类构造函数的prototype设置为一个父类的对象实例。创建这个父类对象实例的目的就是为了构成原型链，以起到共享上层原型方法的作用。但创建这个实例对象时，上层构造函数也会给它设置对象成员，这些对象成员对于继承来说是没有意义的。虽然，我们也没有给构造函数传递参数，但确实创建了若干没有用的成员，尽管其值是undefined，这也是一种浪费啊。

唉！世界上没有完美的事情啊！

11
原型真谛

正当我们感概万分时，天空中一道红光闪过，祥云中出现了观音菩萨。只见她手持玉净瓶，轻拂翠柳枝，洒下几滴甘露，顿时让JavaScript又添新的灵气。

观音洒下的甘露在JavaScript的世界里凝结成块，成为了一种称为“语法甘露”的东西。这种语法甘露可以让我们编写的代码看起来更像对象语言。

要想知道这“语法甘露”为何物，就请君侧耳细听。

在理解这些语法甘露之前，我们需要重新再回顾一下JavaScript构造对象的过程。

我们已经知道，用 var anObject = new aFunction() 形式创建对象的过程实际上可以分为三步：第一步是建立一个新对象；第二步将该对象内置的原型对象设置为构造函数prototype引用的那个原型对象；第三步就是将该对象作为this参数调用构造函数，完成成员设置等初始化工作。对象建立之后，对象上的任何访问和操作都只与对象自身及其原型链上的那串对象有关，与构造函数再扯不上关系了。换句话说，构造函数只是在创建对象时起到介绍原型对象和初始化对象两个作用。

那么，我们能否自己定义一个对象来当做原型，并在这个原型上描述类，然后将这个原型设置给新创建的对象，将其当作对象的类呢？我们又能否将这个原型中的一个方法当作构造函数，去初始化新建的对象呢？例如，我们定义这样一个原型对象：

```
var Person =  //定义一个对象来作为原型类
{
   Create: function(name, age)  //这个当构造函数
   {
      this.name = name;
      this.age = age;
```

```
    },
    SayHello: function()  //定义方法
    {
      alert( "Hello, I' m  " + this.name);
    },
    HowOld: function()  //定义方法
    {
      alert(this.name + " is " + this.age + " years old." );
    }
};
```

这个JSON形式的写法多么像一个C#的类啊！既有构造函数，又有各种方法。如果可以用某种形式来创建对象，并将对象的内置的原型设置为上面这个“类”对象，不就相当于创建该类的对象了吗？

但遗憾的是，我们几乎不能访问到对象内置的原型属性！尽管有些浏览器可以访问到对象的内置原型，但这样做的话就只能限定了用户必须使用那种浏览器。这也几乎不可行。

那么，我们可不可以通过一个函数对象来做媒介，利用该函数对象的prototype属性来中转这个原型，并用new操作符传递给新建的对象呢？

其实，像这样的代码就可以实现这一目标：

```
function anyfunc(){};          //定义一个函数躯壳
anyfunc.prototype = Person;    //将原型对象放到中转站prototype
var BillGates = new anyfunc(); //新建对象的内置原型将是我们期望的原型对象
```

不过，这个anyfunc函数只是一个躯壳，在使用过这个躯壳之后它就成了多余的东西，而且这和直接使用构造函数来创建对象也没啥不同，有点不爽。

可是，如果我们将这些代码写成一个通用函数，而那个函数躯壳也就成了函数内的函数，这个内部函数不就可以在外层函数退出作用域后自动消亡吗？而且，我们可以将原型对象作为通用函数的参数，让通用函数返回创建的对象。我们需要的就是下面这个形式：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS22.htm

```
<script type=" text/javascript" >
  function New(aClass, aParams)  //通用创建函数
  {
    function new_()  //定义临时的中转函数壳
    {
      aClass.Create.apply(this, aParams);  //调用原型中定义的的构造函数
                                           //中转构造逻辑及构造参数
```

```
    };
    new_.prototype = aClass;   //准备中转原型对象
    return new new_();         //返回建立最终建立的对象
};

var Person =       //定义的类
{
    Create: function(name, age)
    {
        this.name = name;
        this.age = age;
    },
    SayHello: function()
    {
        alert( "Hello, I' m  " + this.name);
    },
    HowOld: function()
    {
        alert(this.name + " is " + this.age + " years old." );
    }
};

var BillGates = New(Person, [ "Bill Gates" , 53]);   //调用通用函数创建对象
                                                     //并以数组形式传递构造参数
BillGates.SayHello();
BillGates.HowOld();
    alert(BillGates.constructor == Object);    //输出：true
</script>
```

这里的通用函数New()就是一个“语法甘露”！这个语法甘露不但中转了原型对象，还中转了构造函数逻辑及构造参数。

有趣的是，每次创建完对象退出New函数作用域时，临时的new_函数对象会被自动释放。由于new_的prototype属性被设置为新的原型对象，其原来的原型对象和new_之间就已解开了引用链，临时函数及其原来的原型对象都会被正确回收了。上面代码的最后一句证明，新创建的对象的 constructor属性返回的是Object函数。其实新建的对象自己及其原型里没有constructor属性，那返回的只是最顶层原型对象的构造函数，即Object。

有了New这个语法甘露，类的定义就很像C#那些静态对象语言的形式了，这样的代码显得多么文静而优雅啊！

当然，这个代码仅仅展示了“语法甘露”的概念。我们还需要多一些语法甘露，才能实现用简洁而优雅的代码书写类层次及其继承关系。好了，我们再来看一个更丰富的示例吧：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS23.htm

```
<script type=" text/javascript" >
  //语法甘露：
  var object =   //定义小写的object基本类，用于实现最基础的方法等
  {
    isA: function(aType)  //一个判断类与类之间以及对象与类之间关系的基础方法
    {
      var self = this;
      while(self)
      {
        if (self == aType)
          return true;
        self = self.Type;
      };
      return false;
    }
  };

  function Class(aBaseClass, aClassDefine)   //创建类的函数，用于声明类及继承关系
  {
    function class_()  //创建类的临时函数壳
    {
      this.Type = aBaseClass;   //我们给每一个类约定一个Type属性，引用其继承的类
      for(var member in aClassDefine)
        this[member] = aClassDefine[member];   //复制类的全部定义到当前创建的类
    };
    class_.prototype = aBaseClass;
    return new class_();
  };

  function New(aClass, aParams)  //创建对象的函数，用于任意类的对象创建
  {
    function new_()    //创建对象的临时函数壳
    {
      this.Type = aClass;    //我们也给每一个对象约定一个Type属性，
                             //据此可以访问到对象所属的类
      if (aClass.Create)
        aClass.Create.apply(this, aParams);  //我们约定所有类的构造函数都叫Create
    };
    new_.prototype = aClass;
    return new new_();
  };

  //语法甘露的应用效果：
  var Person = Class(object,     //派生自object基本类
  {
    Create: function(name, age)
    {
      this.name = name;
      this.age = age;
    },
```

```
    SayHello: function()
    {
        alert( "Hello, I' m " + this.name + ", " + this.age + " years old." );
    }
});

var Employee = Class(Person,   //派生自Person类，是不是和一般对象语言很相似？
{
    Create: function(name, age, salary)
    {
        Person.Create.call(this, name, age);  //调用基类的构造函数
        this.salary = salary;
    },
    ShowMeTheMoney: function()
    {
        alert(this.name + " $" + this.salary);
    }
});

var BillGates = New(Person, [ "Bill Gates", 53]);
var SteveJobs = New(Employee, [ "Steve Jobs", 53, 1234]);
BillGates.SayHello();
SteveJobs.SayHello();
SteveJobs.ShowMeTheMoney();

var LittleBill = New(BillGates.Type, [ "Little Bill", 6]); //用BillGate的类型建立LittleBill
LittleBill.SayHello();
    alert(BillGates.isA(Person));      //true
    alert(BillGates.isA(Employee));    //false
    alert(SteveJobs.isA(Person));      //true
    alert(Person.isA(Employee));       //false
    alert(Employee.isA(Person));       //true
</script>
```

“语法甘露”不用太多，只要那么一点点，就能改观整个代码的易读性和流畅性，从而让代码显得更优雅。有了这些语法甘露，JavaScript就很像一般对象语言了，写起代码来感觉也就爽多了！

令人高兴的是，受这些甘露滋养的JavaScript程序效率会更高。因为其原型对象里既没有了毫无用处的那些对象级的成员，而且还不存在 constructor属性体，少了与构造函数间的牵连，但依旧保持了方法的共享性。这让JavaScript在追溯原型链和搜索属性及方法时，少费许多工夫啊。

我们就把这种形式称为“甘露模型”吧！其实，这种“甘露模型”的原型用法才是符合prototype概念的本意，才是JavaScript原型的真谛！

接下来，我们将要把这种甘露模型进一步优化和提炼，这样才能让语法甘露更好地滋润JavaScript编程世界，不辜负观音姐姐的点化。

12
甘露模型

在上面的示例中，我们定义了两个语法甘露：一个是Class()函数，一个是New()函数。使用Class()甘露，我们已经可以用非常优雅的格式定义一个类。例如前例中的：

```
var Employee = Class(Person,   //派生至Person类
{
   Create: function(name, age, salary)
   {
      Person.Create.call(this, name, age); //调用基类的构造函数
      this.salary = salary;
   },
   ShowMeTheMoney: function()
   {
      alert(this.name + " $" + this.salary);
   }
});
```

这种类的写法已经和C#或Java的格式非常相似了。不过，其中调用基类的构造函数还需要用“Person.Create.call(this, name, age)”这样的方式来表达。这需要用到基类的类名，并要用call这种特殊的方式来传递this指针。这和C#的base()以及Java的super()那样的简单调用方式比起来，还需要进一步美化。

而New()函数的使用也不是很爽。前例中需要用“New(Employee, ["Steve Jobs", 53, 1234])”这样的方式来创建对象，其中第一个参数是类，其他构造参数需要用数组包起来。这和JavaScript本来那种自然的“new Employee("Steve Jobs", 53, 1234)”比起来，丑陋多了。这也需要美化。

为了实现这些美化工作，我们需要回顾一下new一个对象的实质。前面我们说过：

```
var anObj = new aClass();
```

相当于先创建一个空白对象anObj，然后将其作为this指针调用aClass()函数。其实，这个过程中还有一个关键步骤就是将aClass的prototype属性，赋值给anObj内置的prototype属性。尽管我们无法访问到anObj内置的prototype属性，但它却为对象提供了

可以调用的方法。

由于前例中的Class()语法甘露实际上是构造了一个原型，并将这个原型挂在了相应的原型链上。由于它返回的是一个对象而不是函数，因此由它定义出来的Person和Employee类也都只是对象而不是函数，无法用new Person()或new Employee()这样的方式来创建对象。要基于一个原型来创建对象，就得要借助New()语法甘露来中转这个原型。

那么，如果我们让Class()语法甘露返回一个函数而不是对象，不就可以用new Person()和new Employee()这种方式来创建对象了吗？而且，我们可为这个返回函数创建一个继承至相关原型链的原型对象，并设置到该函数的prototype属性。这样，我们用new方式创建这个类函数的对象时，就自然地继承该类的原型了。

那么，让Class()语法甘露返回什么函数呢？因为Class()语法甘露返回的函数是用来创建对象的，当然应该返回该类的构造函数了，正好可以是类定义参数中的Create方法啊。这样一来，我们也无需在New()语法甘露中间接地调用Create构造函数了，事实上New()语法甘露可以完全扔掉了。

于是，我们就有了下面这个精简甘露模型的例子：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS24.htm

```
<script type=" text/javascript" >
//定义类的语法甘露：Class()
//最后一个参数是JSON表示的类定义
//如果参数数量大于1个，则第一个参数是基类
//第一个和最后一个之间参数，将来可表示类实现的接口
//返回值是类，类是一个构造函数
function Class()
{
   var aDefine = arguments[arguments.length-1]; //最后一个参数是类定义
   if(!aDefine) return;
   var aBase = arguments.length>1 ? arguments[0] : object; //解析基类

   function prototype_(){}; //构造prototype的临时函数，用于挂接原型链
   prototype_.prototype = aBase.prototype;  //准备传递prototype
   var aPrototype = new prototype_();   //建立类要用的prototype

   for(var member in aDefine)  //复制类定义到当前类的prototype
      if(member!=" Create" )   //构造函数不用复制
         aPrototype[member] = aDefine[member];

   if(aDefine.Create)  //若有构造函数
      var aType = aDefine.Create  //类型即为该构造函数
   else   //否则为默认构造函数
```

```
        aType = function()
        {
            this.base.apply(this, arguments);
        };

    aType.prototype = aPrototype;  //设置类(构造函数)的prototype
    aType.Base = aBase;            //设置类型关系
    aType.prototype.Type = aType;  //为本类对象扩展一个Type属性
    return aType;  //返回构造函数作为类
};

//根类object定义：
function object(){}  //定义小写的object根类，用于实现最基础的方法等
object.prototype.isA = function(aType)  //判断对象是否属于某类型
{
    var self = this.Type;
    while(self)
    {
        if(self == aType) return true;
        self = self.Base;
    };
    return false;
};

object.prototype.base = function() //调用基类构造函数
{
    var Caller = object.prototype.base.caller;
    Caller && Caller.Base && Caller.Base.apply(this, arguments);
};

//语法甘露的应用效果：
var Person = Class     //默认派生自object基本类
({
    Create: function(name, age)
    {
        this.base();
        this.name = name;
        this.age = age;
    },
    SayHello: function()
    {
        alert( "Hello, I' m " + this.name + ", " + this.age + " years old." );
    }
});

var Employee = Class(Person,   //派生自Person类
{
    Create: function(name, age, salary)
```

```
    {
        this.base(name, age); //调用基类的构造函数
        this.salary = salary;
    },
    ShowMeTheMoney: function()
    {
        alert(this.name + " $" + this.salary);
    }
});

var BillGates = new Person("Bill Gates", 53);
var SteveJobs = new Employee("Steve Jobs", 53, 1234);
BillGates.SayHello();
SteveJobs.SayHello();
SteveJobs.ShowMeTheMoney();

var LittleBill = new BillGates.Type("Little Bill", 6); //用BillGate的类型建LittleBill
LittleBill.SayHello();

alert(BillGates.isA(Person));      //true
alert(BillGates.isA(Employee));    //false
alert(SteveJobs.isA(Person));      //true
</script>
```

这个精简甘露模型模拟出来的类更加自然和谐，而且比前面的甘露模型更加精简。其中的Class()函数虽然很简短，却是整个模型的关键。

Class()函数将最后一个参数当做类定义，如果有两个参数，则第一个参数就表示继承的基类。如果多于两个参数，则第一个和最后一个之间的参数都可以用作类需要实现的接口声明，保留给将来扩展甘露模型使用吧。

使用Class()函数来定义一个类，实际上就是为创建对象准备了一个构造函数，而该构造函数的prototype已经初始化为方法表，并可继承上层类的方法表。这样，当用new操作符创建一个该类对象时，也就很自然地将此构造函数的原型链传递给了新构造的对象。于是，就可以采用像“new Person("Bill Gates", 53)”这样的语法来创建对象了。

类定义中名为Create的函数是特别对待的，因为这就是构造函数。如果没有定义Create构造函数，Class()函数也会创建一个默认构造函数。事实上，这个构造函数就代表了类。除此之外，我们还为其定义了一个Base属性，以方便追溯继承关系。

在本例中的根类object的原型中，我们定义了一个base方法。有了这个方法之后，在类定义的构造函数中，就可以使用“this.base()”这样的方式来调用基类的构造函数。这种调用基类的方式和C#的base及Java的super就非常相似了。

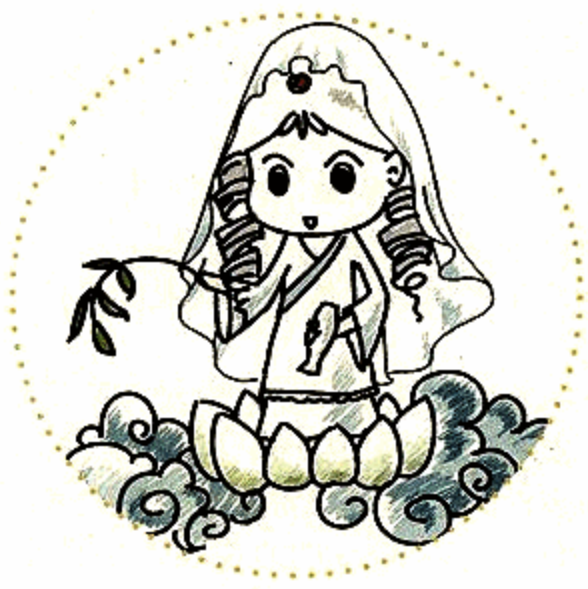
不过，Class()函数中还是有个小问题，那就是不支持toString()方法的覆写。也就是说，如果我们为一个类定义了自己的toString()方法，调用Class()函数来生成类时，toString()方法会丢失。

原来toString()方法被JavaScript规定为不可枚举的内置方法。除此之外，还有toLocaleString(), valueOf(), hasOwnProperty(), isPrototypeOf(), propertyIsEnumerable()等，都是不能枚举的内置方法。这样在Class()函数中的那个for(…in…)语句就不能遍历到这些属性，导致Bug的产生。

因此，这个Class()语法甘露还不能支持不可枚举内置方法的覆写。这不能不说是一个小小的遗憾。当然，遇到需要完全覆写这些内置方法的情况并不多。顶多偶尔会有toString(), toLocaleString(), valueOf()这三个方法的覆写，其他几个几乎不会有覆写的情况。

除此之外，object根类的base()方法也还有个小问题。这个方法用到了函数的caller属性，以此判断构造函数的层次。而Opera浏览器不支持函数的caller属性，因此base方法不适合于Opera浏览器，这不能不说是另一个更大的遗憾。

如果在甘露模型中留有这样的问题，想必观音姐姐也会遗憾。我们还需要继续努力，别让观音姐姐失望啊。



其实，要解决不能覆写非枚举属性的问题也并非难事。既然这些属性是特殊的，就可以对其进行特殊处理。我们可以在复制完可枚举的属性之后，加上类似下面的特殊判断处理语句：

```
if(aDefine.toString != Object.prototype.toString)
  aPrototype.toString = aDefine.toString;
```

因为，如果覆写了toString方法，那么它就肯定不等于原生的toString方法，这时就

可复制toString方法。当然，我们也可以不通过比较，而是用hasOwnProperty来判断是否覆写了toString方法：

```
if(aDefine.hasOwnProperty( "toString" ))
  aPrototype.toString = aDefine.toString;
```

使用哪种判断方式可以任选。我们建议采用直接比较方式，这样可以避免万一遇到覆写Object.prototype.hasOwnProperty的情况，也不至于出问题。当然，这也似乎有点太钻牛角尖了。

在实际的应用中，我们建议根据具体应用情况来决定是否需要支持特殊属性的覆写。如果在应用中根本不会覆写这些特殊属性，就无需加上这样的特殊处理。如果是打造专业的AJAX类库，最多支持toString(), toLocaleString(), valueOf()这三个方法的覆写就可以了。千万不要玩画蛇添足的游戏。

最头痛的是对base()函数的重写，也就要兼容不支持caller属性的Opera浏览器。虽然Opera浏览器目前只占很小的市场范围，但也算有名的四大浏览器之列。如果甘露模型不支持Opera浏览器，显然无法彰显观音菩萨的法力，也更不好意思说自己是观音老师的弟子。

其实，base()方法之所以要使用自身的caller属性，就是为了确定当前构造函数的层次，从而可以知道该调用更上层的构造函数。有没有别的办法来知道是那层构造函数调用了base()方法呢？残酷的事实告诉我们，除了函数自身的caller属性，没有办法知道是谁调用了自己。

既然改变不了别人，那就改变自己！我们为什么就不能在运行中改变base()自身呢？

事实上，第一层构造函数调用this.base()时，我们是可以通过this.Type属性知道一层构造函数的，而this.Type.Base就是第二层构造函数。只是，第二层构造函数又会调用this.base()，其本来是想调用第三层的构造函数，但再次进入base()函数时，就无法知晓构造函数的层次了。

如果我们在第一层构造函数调用进入this.base()时，先改变this.base本身，让其在下次被调用时能掉到第三层构造函数。完成这个变身动作之后再调第二层构造函数，而第二层构造函数再调用this.base()时就能调用到第三层构造函数了。这样，只要我们在每次的base()调用中都完成一个自我的变身动作，就可以按正确的顺序完成对构造函数的调用。这是多么有趣的调用方式啊！

于是，我们可以将原来的base()函数改写成下面的形式：

```
object.prototype.base = function()  //调用基类构造函数
{
   var Base = this.Type.Base;  //获取当前对象的基类
   if(!Base.Base)  //若基类已没有基类
      Base.apply(this, arguments)    //则直接调用基类构造函数
   else   //若基类还有基类
   {
      this.base = MakeBase(Base);    //先覆写this.base
      Base.apply(this, arguments);   //再调用基类构造函数
      delete this.base;              //删除覆写的base属性
   };
   function MakeBase(Type) //包装基类构造函数
   {
      var Base = Type.Base;
      if(!Base.Base) return Base; //基类已无基类，就无需包装
      return function()  //包装为引用临时变量Base的闭包函数
      {
         this.base = MakeBase(Base);    //先覆写this.base
         Base.apply(this, arguments);   //再调用基类构造函数
      };
   };
};
```

原来的base()函数只有两行代码，而新的base()函数却又十几行代码。看来为了支持Opera浏览器确实也付出了代价，好在这些代码只有十来行，代价并不是太大。当然，如果无须支持Opera浏览器，也就不必用这个base()函数了。

在这个新的base()函数中，对this.base的覆写实际上只是在this对象身上创建了一个临时的base方法。这个临时方法暂时遮住了object.prototype.base方法，而object.prototype.base方法却一直存在。而每次对this.base的变身操作，都是针对这个临时的方法的。当所有层次构造函数的调用都完成之后，即可删除this对象的这个临时base方法。

其中的MakeBase()函数非常有意思，如果基类还有基类，它就返回一个闭包函数。下次this.base()被构造函数调用时，即调用的是这个闭包函数。但这个闭包函数又可能会调用MakeBase()形成另一个闭包函数，直到基类再无基类。

如果说这是递归调用呢？却并非那种函数自身对自身的直接或间接调用，而是调用一个函数却返回另一个函数，再调用返回的函数又会在其中产生新的返回函数。如果说是函数式编程中的高阶函数调用呢？这函数的嵌套阶数却是与类层次相关的不确定数，而每一个阶梯都有新生成的函数。

这可真是闭包中嵌套着闭包，貌似递归却又不是递归，是高阶函数却又高不可

测。一旦整个对象创建完成，用过的内存状态都释放得干干净净，只得到一尘不染的新建对象。JavaScript玩到这样的境界，方显观音大士的法力！

下面就是重写后的完美甘露模型代码：

http://www.leadzen.cn/Books/WuTouJavaScript/1/JS25.htm

```
<script type=" text/javascript" >
  //定义类的语法甘露：Class()
  //最后一个参数是JSON表示的类定义
  //如果参数数量大于1个，则第一个参数是基类
  //第一个和最后一个之间参数，将来可表示类实现的接口
  //返回值是类，类是一个构造函数
  function Class()
  {
    var aDefine = arguments[arguments.length-1]; //最后一个参数是类定义
    if(!aDefine) return;
    var aBase = arguments.length>1 ? arguments[0] : object; //解析基类

    function prototype_(){}; //构造prototype的临时函数，用于挂接原型链
    prototype_.prototype = aBase.prototype;  //准备传递prototype
    var aPrototype = new prototype_();   //建立类要用的prototype

    for(var member in aDefine)  //复制类定义到当前类的prototype
      if(member!=" Create" )   //构造函数不用复制
        aPrototype[member] = aDefine[member];

  //根据是否继承特殊属性和性能情况，可分别注释掉下列的语句
  if(aDefine.toString != Object.prototype.toString)
    aPrototype.toString = aDefine.toString;
  if(aDefine.toLocaleString != Object.prototype.toLocaleString)
    aPrototype.toLocaleString = aDefine.toLocaleString;
  if(aDefine.valueOf != Object.prototype.valueOf)
    aPrototype.valueOf = aDefine.valueOf;

  if(aDefine.Create)  //若有构造函数
    var aType = aDefine.Create  //类型即为该构造函数
  else   //否则为默认构造函数
    aType = function()
    {
      this.base.apply(this, arguments);   //调用基类构造函数
    };

  aType.prototype = aPrototype;   //设置类(构造函数)的prototype
  aType.Base = aBase;             //设置类型关系，便于追溯继承关系
  aType.prototype.Type = aType;   //为本类对象扩展一个Type属性
  return aType;   //返回构造函数作为类
```

```
};

//根类object定义：
function object(){}  //定义小写的object根类，用于实现最基础的方法等
object.prototype.isA = function(aType)  //判断对象是否属于某类型
{
   var self = this.Type;
   while(self)
   {
      if(self == aType) return true;
      self = self.Base;
   };
   return false;
};

object.prototype.base = function() //调用基类构造函数
{
   var Base = this.Type.Base; //获取当前对象的基类
   if(!Base.Base) //若基类已没有基类
      Base.apply(this, arguments)    //则直接调用基类构造函数
   else   //若基类还有基类
   {
      this.base = MakeBase(Base);    //先覆写this.base
      Base.apply(this, arguments);   //再调用基类构造函数
      delete this.base;              //删除覆写的base属性
   };
   function MakeBase(Type) //包装基类构造函数
   {
      var Base = Type.Base;
      if(!Base.Base) return Base; //基类已无基类，就无需包装
      return function()  //包装为引用临时变量Base的闭包函数
      {
         this.base = MakeBase(Base);    //先覆写this.base
         Base.apply(this, arguments);   //再调用基类构造函数
      };
   };
};

//语法甘露的应用效果：
var Person = Class     //默认派生自object基本类
({
   Create: function(name, age)
   {
      this.base();  //调用上层构造函数
      this.name = name;
      this.age = age;
   },
```

```
    SayHello: function()
    {
        alert( "Hello, I' m " + this.name + ", " + this.age + " years old." );
    },
    toString: function()   //覆写toString方法
    {
        return this.name;
    }
});

var Employee = Class(Person,   //派生自Person类
{
    Create: function(name, age, salary)
    {
        this.base(name, age);  //调用基类的构造函数
        this.salary = salary;
    },
    ShowMeTheMoney: function()
    {
        alert(this + " $" + this.salary); //这里直接引用this将隐式调用toString()
    }
});

var BillGates = new Person( "Bill Gates", 53);
var SteveJobs = new Employee( "Steve Jobs", 53, 1234);
    alert(BillGates);  //这里将隐式调用覆写后的toString()方法
    BillGates.SayHello();
    SteveJobs.SayHello();
    SteveJobs.ShowMeTheMoney();

    var LittleBill = new BillGates.Type( "Little Bill", 6); //用BillGate的类型建LittleBill
    LittleBill.SayHello();

    alert(BillGates.isA(Person));      //true
    alert(BillGates.isA(Employee));    //false
    alert(SteveJobs.isA(Person));      //true
</script>
```

当今的JavaScript世界里，各式各样的AJAX类库不断出现。同时，在开放Web API的大潮中，AJAX类库作为Web API最重要的形式，起着举足轻重的作用。这些AJAX类库是否方便引用，是否易于扩展，是否书写优雅，都成了衡量Web API质量的重要指标。

甘露模型基于JavaScript原型机制，用极其简单的Class()函数，构造了一个非常优雅的面向对象的类机制。事实上，我们完全可以在这个甘露模型的基础上打造相关的AJAX类库，为开发人员提供简洁而优雅的Web API接口。

想必微软那些设计AJAX架构的工程师看到这个甘露模型时，肯定后悔没有早点把AJAX部门从美国搬到咱中国的观音庙来，错过了观音菩萨的点化。

当然，我们也只能是在代码的示例中，把Bill Gates当作对象玩玩，真要让他放弃上帝转而皈依我佛肯定是不容易的，机缘未到啊！如果哪天你在微软新出的AJAX类库中看到这种甘露模型，那才是真正的缘分！

13
编程的快乐

在软件工业迅猛发展的今天，各式各样的编程语言层出不穷，新语言的诞生，旧语言的演化，似乎已经让我们眼花缭乱。为了适应面向对象编程的潮流， JavaScript语言也在向完全面向对象的方向发展，新的JavaScript标准已经从语义上扩展了许多面向对象的新元素。与此相反的是，许多静态的对象语言也在向JavaScript的那种简洁而幽雅的方向发展。例如，新版本的C#语言就吸收了JSON那样的简洁表示法，以及一些其他形式的 JavaScript特性。

我们应该看到，随着RIA（富互联应用）的发展和普及，AJAX技术也将逐渐淡出江湖，JavaScript也将最终消失或演化成其他形式的语言。但不管编程语言如何发展和演化，编程世界永远都会在“数据”与“代码”这千丝万缕的纠缠中保持着无限的生机。

只要我们能看透这一点，我们就能很容易地学习和理解软件世界的各种新事物。不管是已熟悉的过程式编程，还是正在发展的函数式编程，以及未来量子纠缠态的大规模并行式编程，我们都有足够的法力来化解一切复杂的难题。

佛最后淡淡地说：只要我们放下那些表面的“类”，放下那些对象的“自我”，就能达到一种“对象本无根，类型亦无形”的境界，从而将自我融入到整个宇宙的生命轮环中。我们将没有自我，也没有自私的欲望，你就是我，我就是你，你中有我，我中有你。这时，我们再看这生机勃勃的编程世界时，我们的内心将自然生起无限的

慈爱之心，这种慈爱之心不是虚伪而是真诚的。关爱他人就是关爱自己，就是关爱这世界中的一切。那么，我们的心是永远快乐的，我们的程序是永远快乐的，我们的类是永远快乐的，我们的对象也是永远快乐的。这就是编程的极乐！

说到这里，在座的比丘都犹如醍醐灌顶，心中豁然开朗。看看左边这位早已喜不自禁，再看看右边那位也是心花怒放。

蓦然回首时，唯见君拈花微笑……

1 禅棋传说 66

2 标准网页 69

3 网页运行原理 75

4 文档对象模型 78

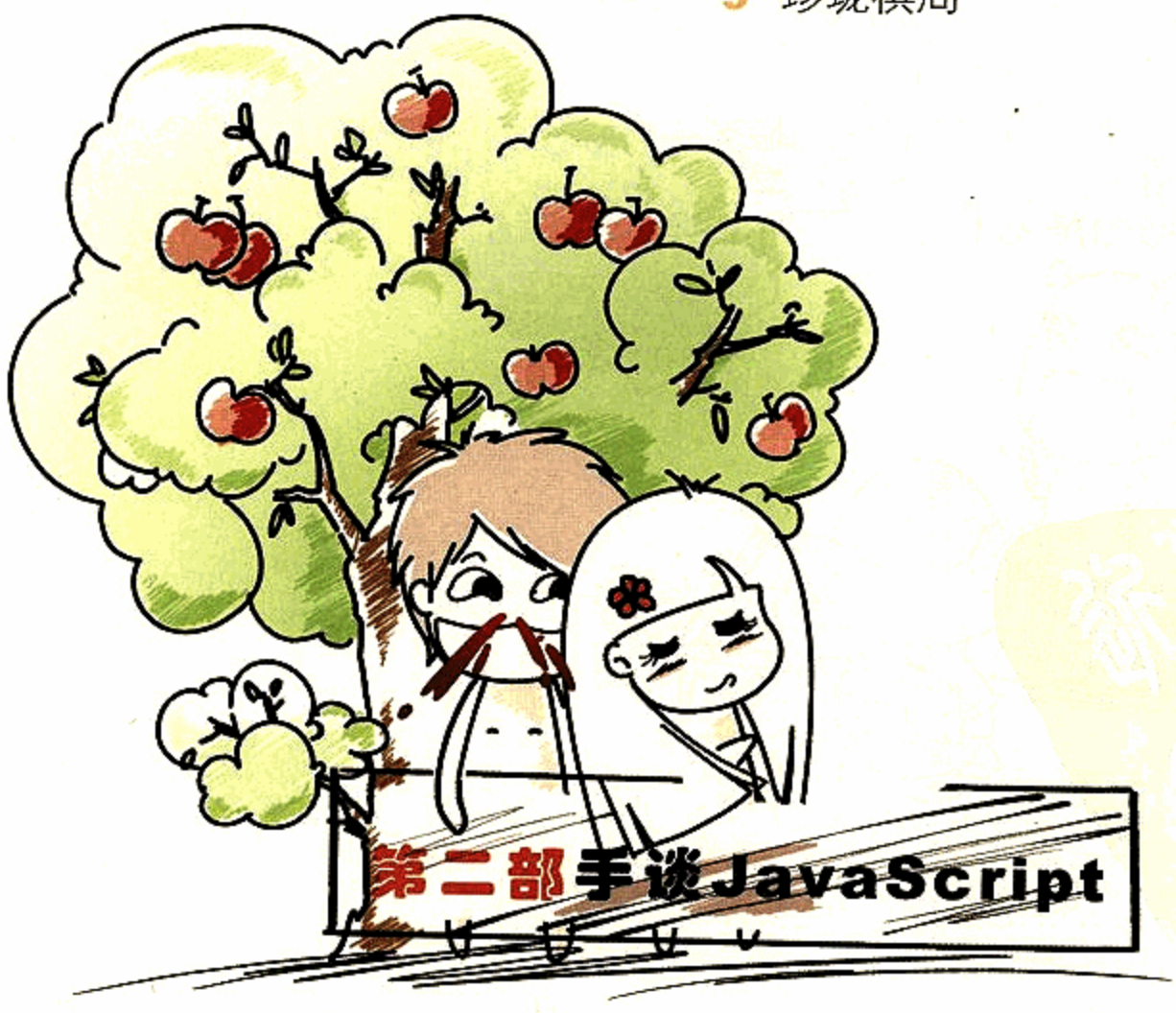
5 妆扮DOM对象 82

6 响应DOM事件 87

7 播放声音 93

8 别向复杂低头 102

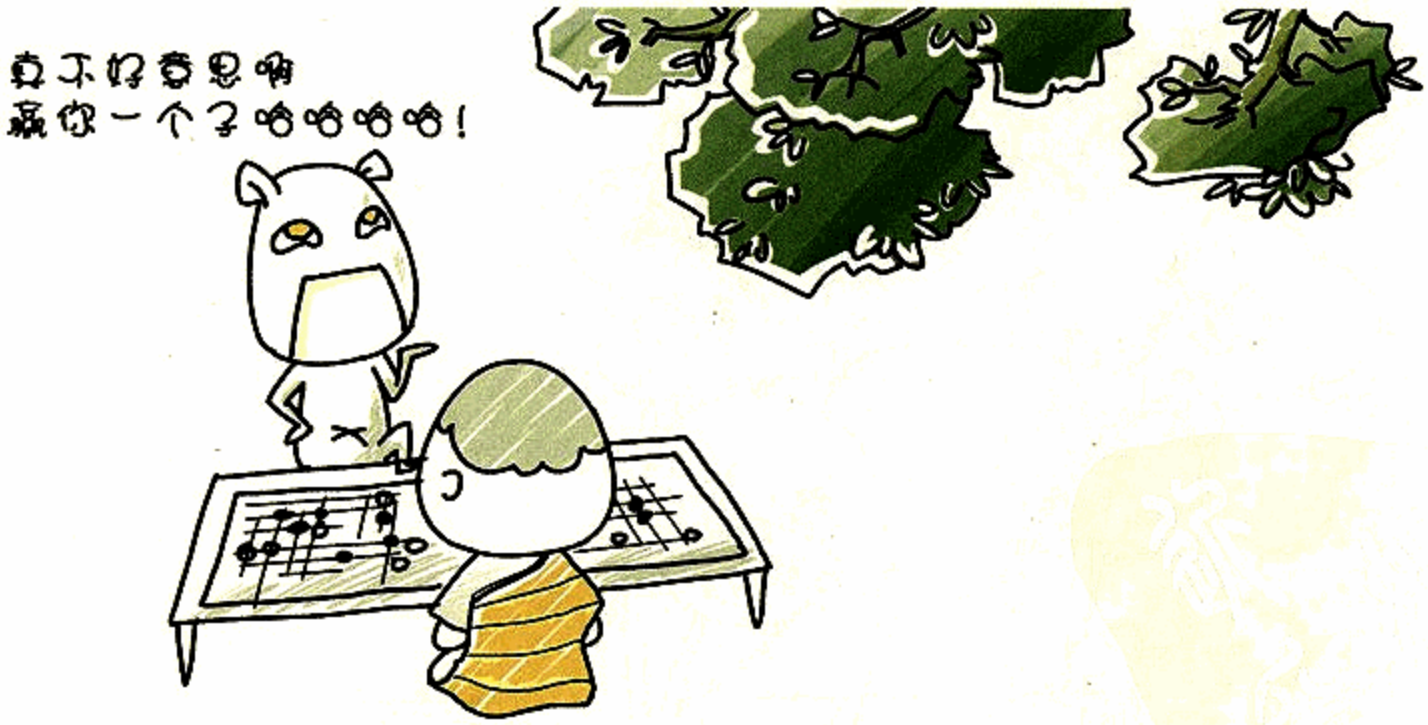
9 珍珑棋局 111



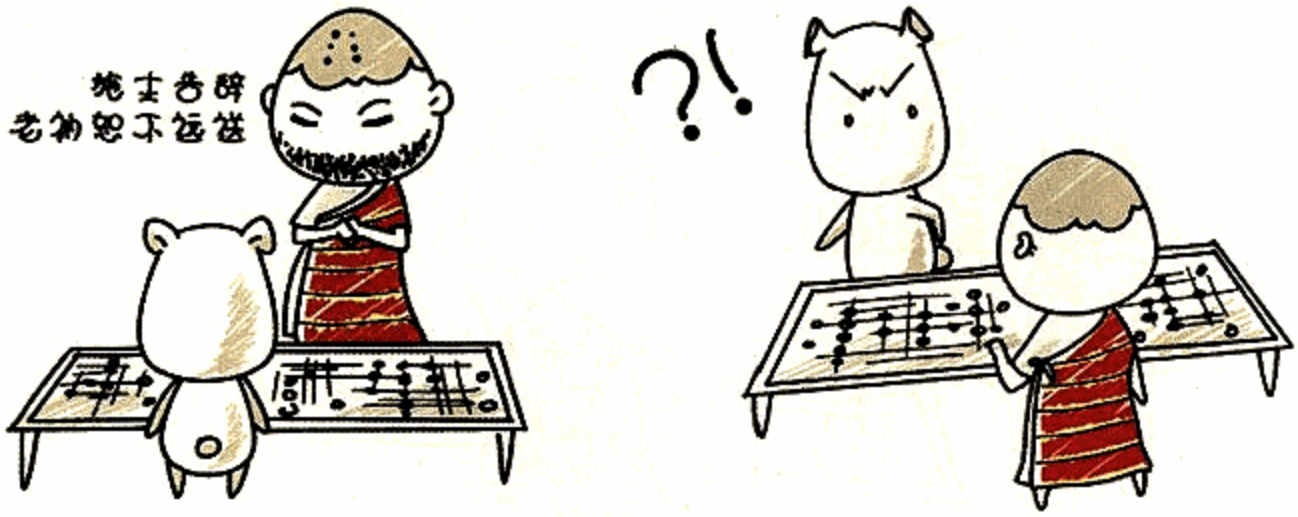
1
禅棋传说

相传，有一位日本棋僧曾随遣唐使来到长安。由于他棋艺超群，打败了长安城里不少的名士。后来，他听说城外太乙山的一座小庙里有个老和尚，年轻时也曾是顶尖的围棋高手，于是就上山讨教一盘，也好在佛门留下个好的名声。

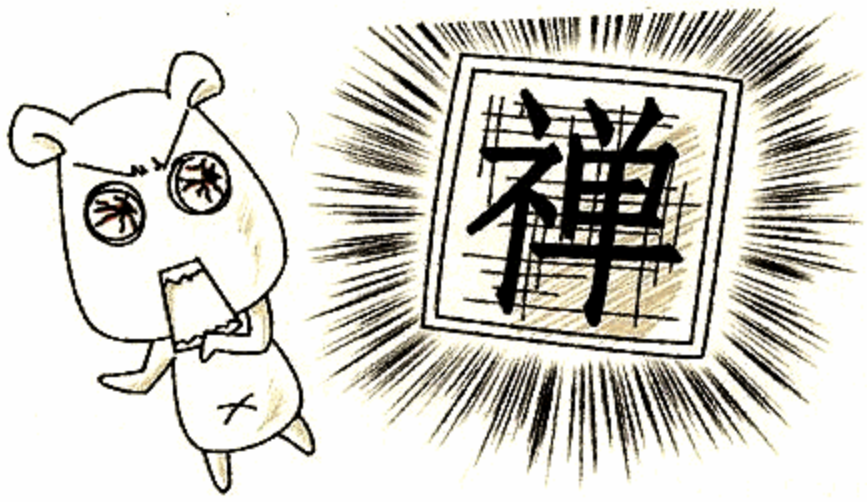
他来到小庙之后，老和尚经不起他的死缠烂磨，终于和他下了一盘。这一盘从开始布局就较上劲了，黑来白往盘中渐起风云，你争我夺江山几番易主，这是日本棋僧有生以来下得最艰难的一盘棋。



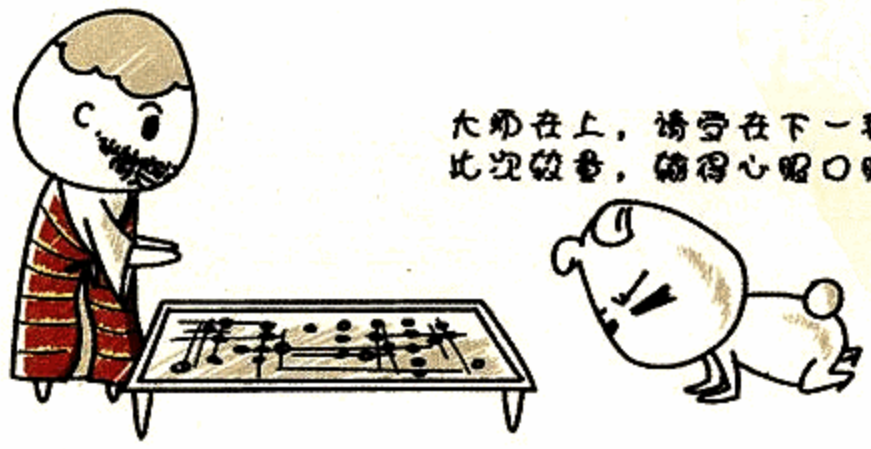
最后，日本棋僧经过痛苦的拼杀，大龙终于冲出重围，并最终于以一个子的优势战胜了老和尚。这时，他满意地笑了。而老和尚自始至终都不动声色。寒暄之后，正当日本棋僧站起身来正要告辞时，却突然像着魔一般地僵住了。



他不敢相信自己的眼睛，他的棋子在棋盘上竟然是一个巨大的“禅”字！



日本棋僧这才明白，原来老和尚一直在引领自己。顿悟之后，他当即拜伏在地，然后起身，哈哈大笑着扬长而去。



这时，老和尚才满意地点头微笑。这位棋僧后来成为了日本的一代名僧，他的名字就叫弁正。

在漫漫程序生涯中，我们何尝又不是如此呢？我们不断学习和积累知识，努力提高自己的技术水平，日夜追逐着层出不穷的新技术。每当有点成绩时，也会沾沾自喜，比人略高一筹时，就会自我膨胀。殊不知这人上有人，天外有天，要知道通向真理之路是永远没有尽头的。遥望大道无尽头，唯见境界有高低。

既然这样，我们为何不能放弃那些无谓的纷争，真正静下心来领悟一下技术背后的真谛呢？论剑者，无非是争个天下第一的虚名。论道者，才是追求世间永恒的真理！

好了，好了，大道理讲多了也是罗嗦。我们来轻松一下，下盘棋吧！

嗯，就用JavaScript来下盘棋如何？

2
标准网页

下棋，首先得找个地方，到依山傍水清凉幽静的松树林下当然不错，可JavaScript去不了。JavaScript只愿意在网页里下，网页才是JavaScript经常云游的地方。好，就在网页里下吧，其实网页里也有青山和绿水。

我们的目标就是：用JavaScript代码，建立一个能下围棋的网页程序。

首先，我们来建一个基本的网页，代码如下：

http://www.leadzen.cn/Books/WuTouJavaScript/2/go0.htm

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
  <title>禅棋传说</title>
</head>
<body>

</body>
</html>
```

为了方便大家，可以从http://www.leadzen.cn/Books/WuTouJavaScript/2/go0.htm打开这个文件。不过，除了标题外，go0.htm显示的一片空白。

我们这个网页就叫“禅棋传说”，这样写起程序来似乎带劲些。如果把工作当工作，工作也做不好，自己也不开心。如果把工作当兴趣嘛，既做好了工作，自己也快乐些。实在没兴趣，我们也要制造一下气氛。

不过，慢着，这个网页里怎么有两个URL超级链接？是啊，就是这两个：

http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd
http://www.w3.org/1999/xhtml

莫非，我们的网页还会偷偷去访问这两个地方？看来，有必要直接打开这两个链接来看个究竟。

晕啊，打开第一个链接之后，结果下载了一个名为xhtml1-transitional.dtd的文件。打开第二个链接，出现的是W3C组织的xHTML namespace的网页。

直觉告诉我们，这两个东西在这里不能被简单地理解为超级链接。用Visual Studio或Dreamweaver等建立的网页，大都有这些两个链接。如果它们是超级链接，那么岂不是每个网页都会链接到这些地址？如果我们的机器没有上网，或者www.w3.org网站不工作了，莫非打开网页都会出问题？显然，这是不可能的！看来我们直接去打开这两个地址的举动，多多少少有点愚笨啊。

原来，我们现在编写的网页已经是xHTML标准的网页。这些网页已经是XML格式的HTML文件，所以叫xHTML。由于XML有非常严谨的语法以及严格的标准，xHTML也就比原来的HTML更加规范。所以从现在起，我们都得写xHTML标准的网页了，否则人家就会以为你不懂规矩了。

xHTML文件的第一行一般是XML的DOCTYPE声明，用来表明当前XML文档的根元素和文档类型定义。在我们的网页文件中的第一行是：

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

“html”指明当前XML文档的根元素是html，注意是小写的。XML是区分大小写的，而且XHTML标准规定，所有的标签元素都用小写。所以，我们把它改成大写的HTML就会出错了，后面并不存在大写的XML根元素。

“PUBLIC”表明XML文档类型定义是可以通过公共标识符来识别的，这个公共标识符就是："-//W3C//DTD XHTML 1.0 Transitional//EN"。如果真的不认识这个公共标识符，也可以从"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"引用的地址去找此XML的文档类型定义。

所谓公共标识符，就是全世界人民都没有任何歧义的唯一标识符，地球人都知道，火星来的除外。标识符本身的内容并不重要，有意义或无意义的字符串都行，只要是唯一的。显然，我们用的浏览器都认识这个公共标识符，直接按公认的规则处理文档内容即可，无须去前面那个网址取文件了。

接下来的一行：

```
<html xmlns=" http://www.w3.org/1999/xhtml" >
```

其中有个“xmlns”的属性，其表明当前这个“html”元素标签的名字空间。同样，后面的"http://www.w3.org/1999/xhtml" 也只是名字空间的标识符。在XML中，相同名称的标签不一定会解释为相同的含义，除非名字空间也完全相同。所以，今后写网页时，我们也得加上这个，不能再是光秃秃的<html>了。

相信那些熟悉XML的大侠们，一定不会像我们这样傻乎乎地去打开这两个超级链接。有趣的是，我们却因为这种傻乎乎的举动知道了xHTML与XML的关系，相信有些朋友一定会顺着这个机缘，去进一步学习XML的相关知识。这也算是大智若愚吧。

后面的内容就都是我们熟悉的HTML了，闭着眼睛都能摸出来那些HTML标签是干啥的。真是这样吗？也许吧。不过，在xHTML中，标签可不能随便乱写，还是那句话：我们得懂规矩！

既然是要用JavaScript写一个网页围棋程序，我们要面临的第一个问题就是：包含JavaScript代码的<script>标签该写在哪里？

以前在HTML时代，我们几乎可以在HTML的任何位置写一段<script>标签包含的JavaScript代码，而今的xHTML就不应该这样了。

首先，我们得把xHTML文件当一个XML文档，其根节点是<html>，因此绝对不能把<script>标签写到<html>标签之外，这不符合XML文档的规范。

其次，要为这个XHTML指定字符编码格式，否则，浏览器可能会产生错误的解码，从而使网页或JavaScript 输出的文字出现乱码。最简单的做法就是在xHTML的最开始加入XML编码定义，例如将字符编码格式定义为utf-8：

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

不过，对于XHTML文件来说，最常用的做法是在<head>中加入：

```
<meta http-equiv= “Content-Type” content= “text/html;charset=utf-8” />
```

这个<meta>标签的http-equiv属性相当于建立一个等效的HTTP响应头项目Content-Type，并在content属性中指定MIME格式和字符编码。

然后，XHTML的文档类型定义规定，<html>元素中只能存在<head>和<body>元素。因此我们也不能把<script>标签写到<head>或<body>标签之外，也就是说<script>标签不能是<html>的子标签。<head>和<body>都可以包含<script>子标签。至于其他的标签是否可以包含<script>标签，这要看XHTML的文档类型定义文件规定，就是我们稀里糊涂下载的那个xhtml1-transitional.dtd文件，有趣吧?

还有，<script>标签必须得有一个type属性，这也是XHTML的文档类型定义规定的。也就是说，我们必须这样写：<script type="text/javascript">。同时得记住，以前那种language="javascript" 的属性及写法已经被淘汰了。即使是通过document.write动态写入的<script>标签，最好也遵循这个规矩。

最后，还要解决一个嵌入的JavaScript代码与XML语法冲突的问题。如果，<script>标签引入的是一个外部JS文件，标签内是没有JavaScript代码的，这不会出现问题。但对于<script>标签内包含的JavaScript代码，却不是非标准的XML形式，XML解析器是不能解析的。虽然，我们的网页也并不需要由XML解析器来解析，而是通过浏览器来解析的。但是，既然XHTML是XML文档，它必须符合XML。完全符合XML文档规范的网页才能算是标准网页。

对于这种情况，我们需要用到XML的CDATA段来包含这些JavaScript代码。即，将JavaScript代码包含在“<![CDATA[”和“]]>”之间。“<![CDATA[”和“]]>”是XML文档特殊的分割标签，表示里面的内容是嵌入XML的数据原样。不过，多了这对标签来包裹<script>里的JavaScript代码之后，JavaScript执行引擎又不认识这两个奇怪的标记了。于是，我们还得将这两个奇怪的标志分别放到一个单独的行，并用“//”注释掉，以便JavaScript执行引擎将其忽略。

我们的网页也并不需要由XML解析器来解析，而是通过浏览器来解析的。即使在JavaScript代码中含有XML的特殊字符，浏览器并不会将其识别为标签的开始或结束，也能正确处理。但是，既然XHTML是XML文档，它就应该符合XML规范。也只有完全符合XML文档规范的网页才能算是标准网页。

如果JavaScript代码中没有出现XML的特殊符号，不会引起XML解析器的误解。我们无需做额外处理。但如果JavaScript代码中确实含有XML的特殊符号，我们就必须要特殊处理。一般情况下，我们有两种方法来处理。

第一种方法是将JavaScript代码包含在XML的注释中，即，将代码包含在“<!--”和“-->”之间。看起来像这样：

```
<script type=" text/javascript" >
  <!--
```

```
    ......//JavaScript代码
    -->
  </script>
```

一般来说，浏览器的JavaScript解析器都会忽略“<!--” 和 “-->”符号，并不会报语法错误。而XML解析器也会把“<!--” 和 “-->”之间的内容当作XML的注释。因此，JavaScript代码可以正确执行，也能通过W3C的标准检测，双方都满意。如果实在担心有些浏览器不认识“<!--” 和 “-->”，还可以写成这样：

```
  <script type=" text/javascript" >
    //<!--
    ..... //JavaScript代码
    //-->
  </script>
```

也就是说，将“<!--” 和 “-->”也注释掉，这样就更保险了。不过，使用XML的注释要注意一个问题，就是JavaScript的代码里不能包含“--”字符串，因为这又将导致XML语法错误。比如你的JavaScript代码中有“i--”这样的递减语句，将导致无法通过XHTML的检查。

第二种方式是用XML的CDATA段来包含这些JavaScript代码。即，将JavaScript代码包含在“<![CDATA[” 和 “]]>”之间。“<![CDATA[” 和 “]]>”是XML文档特殊的分割标签，表示里面的内容是嵌入XML的数据原样。

不过，多了这对标签来包裹<script>里的JavaScript代码之后，JavaScript执行引擎又不认识这两个奇怪的标记了。于是，我们还得将这两个奇怪的标志分别放到一个单独的行，并用“//”注释掉，以便JavaScript执行引擎将其忽略。于是就有了下面这种写法：

```
  <script type=" text/javascript" >
    //<![CDATA[
    ..... //JavaScript代码
    //]]>
  </script>
```

下面是go1.htm文件的内容，其展示了XHTML标准网页文件以及<script>标签的正确写法：

http://www.leadzen.cn/Books/WuTouJavaScript/2/go1.htm

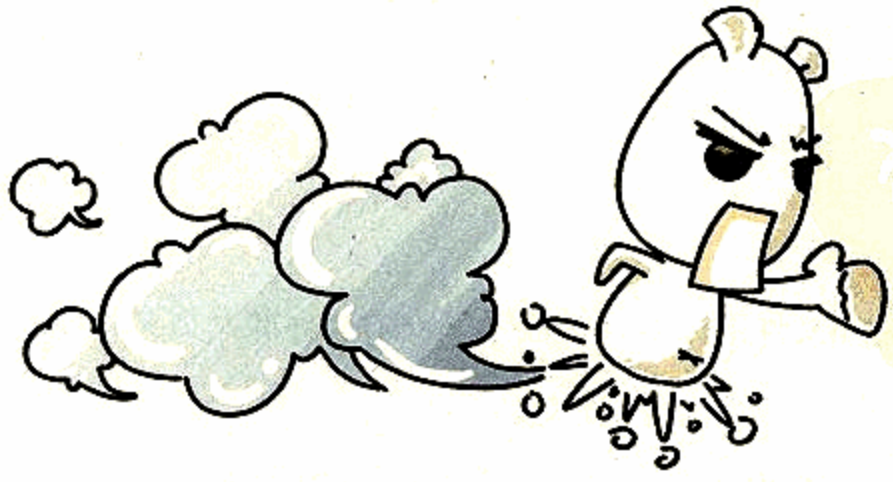
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd" >
```

```
<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
  <meta http-equiv=" Content-Type"  content=" text/html;charset=utf-8"  />
  <title>禅棋传说</title>
  <script type=" text/javascript" >
  //<![CDATA[
    alert( "可以将<script>写到<head>内" );
  //]]>
  </script>
</head>
<body>
  <script type=" text/javascript" >
  //<![CDATA[
    alert( "可以将<script>写到<body>内" );
  //]]>
  </script>
</body>
</html>
```

这就是所谓的xHTML标准网页，也是可以顺利通过W3C的检验的。如果将这个文件拿到http://validator.w3.org/去检验，得到的结果一定是“Congratulations”！

真的好罗嗦哟，讲了这么大一堆内容的目的只有一个：写xHTML标准网页！不过，浏览器开发商们依旧保持了对网页的宽容态度。尽管我们已经声明这个网页要按xHTML文档类型定义来解析和处理，但大多数浏览器还是允许我们随意地写网页。这也就产生了问题。

标准？是什么来着？是否遵守标准完全就看自己的性格和喜好了！



3
网页运行原理

我们知道，打开任何一个网页的时候，浏览器会首先创建一个窗口。这个窗口也就是一个window对象，也是JavaScript运行所依附的全局环境对象和全局作用域对象。

为了加载网页文档，当前窗口将为要打开的网页创建一个document对象，然后将网页加载到这个document中。网页的内容就是在这个过程中，一边加载一边呈现出来的。我们的JavaScript代码也是伴随这个过程，一边加载一边执行的。

在这一过程中，由<script>标签分割的每一段JavaScript代码，都是严格按照其在文档中的顺序来加载和执行的。对于那些由<script>标签引入的那些外部JS文件，虽然其加载过程有可能分派给多个线程去完成，但浏览器也会等待需要的JS文件加载完成，最终还是按正确的顺序来执行JavaScript代码的程序逻辑的。之后，再接着处理其后的其他内容。

进行这些加载和执行的目的，是为了建立文档对象模型（DOM）的主框架。这个主框架是由当前文档窗口的主线程来处理，并依照严格的顺序进行的。而网页中可以异步加载的其他附属资源部分，如图片、声音和视频等多媒体数据，是被安排到其他线程去处理的。

浏览器会中有多少线程可以同时加载网页内容呢？Web标准有一个限制，就是对同一个域名最多只允许使用两个连接来读取内容，大多数浏览器都遵循这一Web标准。也就是说，在同一个域名情况下，只有两个线程可以同时加载内容，其他的会处于等待状态。因此，浏览器使用多少线程在同时加载网页，要看网页内容涉及多少域名。反正，一个域名两个连接，线程多了也会被阻塞。

当然，你可以通过修改Windows注册表，迫使底层的Wininet模块突破这一限制，但这只是对你自己的IE有效。不过，许多网站更喜欢将附属资源分配到多个子域名去加载，这样就不会违背一个域名两个连接的规定，让浏览器能有更多的线程可以并行加载加载这些资源，从而可以充分利用网络带宽来加速网页内容的呈现。

对于外部JS文件的加载，有一种所谓的“异步加载JS文件”技术。其基本原理

就是，在当前被加载执行的JavaScript代码中，向document对象动态地一次写入多个<script>标签来引入要加载的多个JS文件。这时，浏览器在解析和处理动态添加的标签时，将开启相应的线程和连接去加载这些外部JS文件。不过，“异步加载JS文件”的技术只是实现了JavaScript代码的异步加载，而并非JavaScript代码的异步执行！而且，不同浏览器对这种动态引入的多个JS代码的执行顺序是不同的。

IE会在document.write语句写入动态<script>标签之后立即返回，它并不等待动态代码的加载和执行，而是继续执行后面的代码。因此，用多个document.write语句来一个一个地写入动态<script>标签与一次写入多个<script>动态标签没啥区别。

但Firefox却与IE不同，它会在发出document.write语句之后，等待动态的代码异步加载并顺序执行之后，才接着执行后面的JavaScript代码。于是，在Firefox中一个一个地document.write动态<script>标签就不会起到异步加载JS文件的作用，你必须一次性写入多个动态<script>标签来引入JS文件，才能起到异步加载多个JS文件的效果。

当然，随着IE和Firefox的不断升级和修改，上面说的这些情况就会发生变化。毕竟没有任何标准来规定到底该如何加载JS文件。所以，我们并不建议使用这些旁门左道的技巧。

当网页主框架加载和执行完毕，浏览器这才开始触发window对象或body对象的onload事件。在极少数情况下，网页如果没有body对象，浏览器会在网页加载完后自动创建一个body对象，并将其设置为document的body属性。

其实，window对象和body对象的onload事件是相通的。也就是说，你要么设置window对象的onload事件，要么设置body对象的onload事件，只能有一个起作用。甚至在IE中这两个onload属性干脆就是同一个东西，设置一个也就意味着改了另一个。

当网页主框架加载执行完成，由此引发的事件也处理完毕，JavaScript就暂停执行以等待下一次被触发的时机。当用户的点击、网页定时刷新或者setTimeout和setInterval语句到达指定时间，JavaScript执行引擎将再次被启动，以执行相应的JavaScript代码。

因此，我们可以这么认为：JavaScript总是被动执行的！

当打开或刷新网页时，最顶层的JavaScript代码被执行；设定的时间到达时，相关的JavaScript函数又被执行；用户出发网页元素的事件时又被执行；但一切都静寂下来之后，将没有任何JavaScript代码在执行。

更重要的是，目前JavaScript代码的执行绝对是单线程的！不管是窗口主线程加载

的JavaScript代码，还是异步加载的JavaScript代码，以及随后触发的各种代码执行，都是在一个单一的线程中执行的。甚至，在同一个浏览器进程中，不同窗口对象的相对独立的JavaScript代码也都是在同一个线程里执行的！

也就是说，你打开一个浏览器之后，不管你在这个浏览器进程中开启了多少个页面，新弹出多少个窗口，里面所有的JavaScript代码都是由同一个线程来执行的。即一个浏览器进程中的所有JavaScript代码是串行执行的。除非你再打开一个浏览器，创建出另一个浏览器进程，两个独立进程里的JavaScript代码才是可以并行执行的。

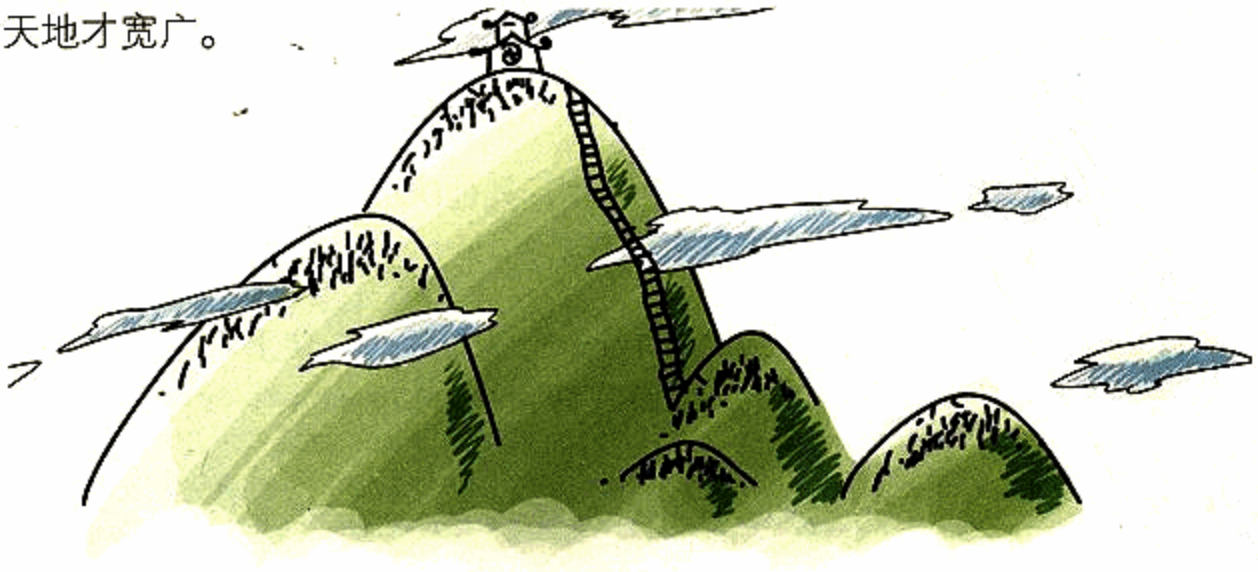
尽管，也有些通过使用setTimeout和setInterval函数来模拟多线程的技巧和模型，但那绝对不是真正的多线程。因为，目前的JavaScript根本就不支持多线程。不过，我们可以注意到JavaScript有一个叫“synchronized”的保留字，从字面上看应该是为了支持多线程而保留的。虽然这个“synchronized”是为了将来扩展而保留的，现在还不起任何作用，但相信JavaScript会有支持多线程那一天。

认识到这一点很重要。这可以让我们理解在AJAX应用中为什么总是需要回调函数和异步处理的原因。如果，JavaScript在发出一个AJAX请求后一直保持等待的话，就意味着整个浏览器进程的停滞，这种情况绝对不应当发生。

同时，我们在给网页编写JavaScript脚本时，就不能太自私地编写大量耗时的脚本。你得考虑到用户可能还在同时浏览其他网页，你的耗时脚本可能让其他网页中的脚本塞车，从而导致用户必须来等待你的拖拖拉拉。这也绝对不是一个好的用户体验！

因此，一个优秀的JavaScript程序员，必须拥有宽阔的胸襟，要随时随地为他人着想。这让我们想起了还有“抢占式多任务”的Windows 3.X年代，微软总要鼓励我们做一个有公德心的程序员，不要自私地长久占用消息循环。

如今看来，我们仍然需要提倡这种美德。只有我们的JavaScript心底无私了，网页的这片天地才宽广。



4
文档对象模型

下棋的地方有了，我们还需要整一个棋盘。那么，怎么整呢？这就需要用到在网页中动态创建DOM的方法了。

所谓的DOM，就是Document Object Model 的缩写，翻译过来就是“文档对象模型”。这种模型主要是为了方便处理HTML和XML文档而设计的。

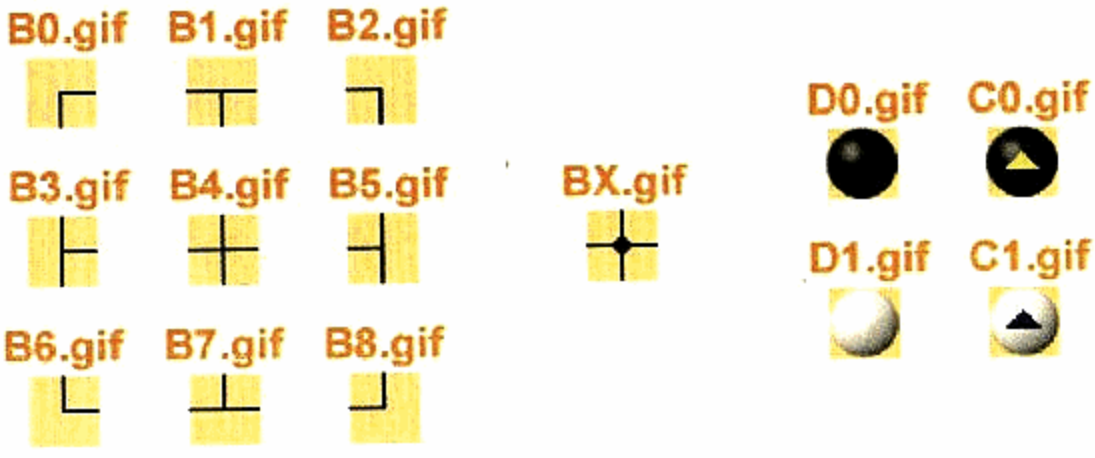
DOM这个术语反映三个意思：一是“文档”，也就是说它的表现是一份文档，即网页；二是“对象”，也就是说，其内部是由一个个可操控的对象构成的；三是“模型”，一个树形结构，一个可以编程的模型。

DOM是一棵树，一颗开花的树，长在JavaScript必经的路旁，为的就是要和JavaScript结一段尘缘。只要我们轻拂JavaScript的代码，就能和DOM一起翩翩起舞，让网页展现出一个多姿多彩的面貌。

使用JavaScript可以轻松操作DOM树，增加、修改或删除DOM树上的各个节点元素。对DOM对象的任何变化，会即时表现在网页上，从而让网页动起来。同时，我们还可以让DOM对象响应各种事件，实现与用户的交互，为用户提供更多的程序功能。

事实上，除了window和document对象之外，其他DOM对象都是可以动态创建的。我们的棋盘就要通过动态创建DOM对象来建立。

我们可以把一个围棋棋盘看成19X19的若干个块，每一个块上可以有一个放棋子的位。因此，每一个块的背景图像可能是以下几种：



我们将动态创建19X19个 标签的DOM对象，然后让它们绝对定位到指定的位置，并将其背景图像设置为正确的名称。这样我们将会在网页上放上一张棋盘。

动态创建DOM对象的标准方法是调用document的createElement方法，比如执行：

document.createElement(“div”);

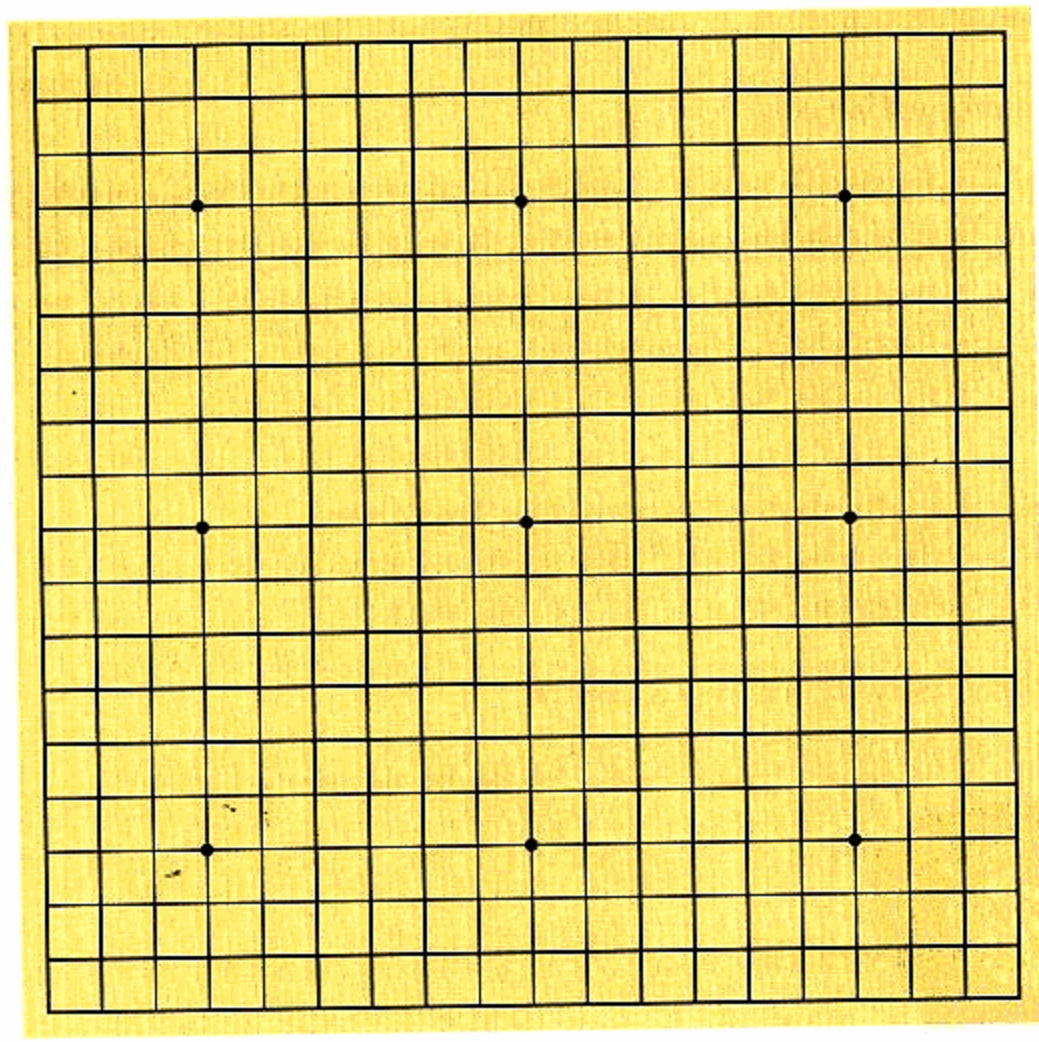
即可创建一个div对象。不过，createElement方法创建的DOM对象还是游离在DOM树之外的，需要将其附加到DOM树的某个节点才能真正地实现对象呈现。将DOM对象附加到DOM树某节点的方法是调用该节点的appendChild方法。我们只需要将这些表示棋盘块的div元素附加到document对象的body节点上即可。下面是生成棋盘的初步代码：

http://www.leadzen.cn/Books/WuTouJavaScript/2/go2.htm

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

<html xmlns=” http://www.w3.org/1999/xhtml” >
<head>
  <meta http-equiv=” Content-Type” content=” text/html;charset=utf-8” />
  <title>禅棋传说</title>
</head>
<body>
  <script type=” text/javascript” >
  //<![CDATA[
    for(var x = 0 ; x < 19; x++)
      for(var y = 0; y < 19; y++)
      {
        var div = document.createElement( “div” );
        document.body.appendChild(div);
        div.style.position = “absolute” ;
        div.style.width = “23px” ;
```

```
            div.style.height = "23px";
            div.style.left = x * 23 + "px";
            div.style.top = y * 23 + "px";
            var s = ((x-9)%9 ? 0 : (x-9)/9) + 1 + (((y-9)%9 ? 0 : (y-9)/9) + 1) * 3;
            if ( s == 4 && (x/3)%2==1 && (y/3)%2==1) s = "X";
            div.style.backgroundImage = "url(B" + s + ".gif)";
        };
    //]]>
    </script>
</body>
</html>
</body>
</html>
```

这段代码运行之后，网页上将出现一个美观的围棋棋盘，看起来还不错：



为生成棋盘，这里用了两层循环操作。其中判断每一块棋盘用什么背景图，用了些简单而巧妙的数字计算和判断，正好可以用JavaScript的语法来简洁地表达。君如果对此有兴趣可以自行揣摸一番，就不多做解释了。

上面的代码中，我们通过设置div元素style属性的各项参数，即可完成对该元素的

定位、设置大小及背景图片等等。除此之外，我们还可以访问到DOM元素的更多属性和方法，操控网页元素按我们的需要表演精彩舞蹈，这就是JavaScript操作DOM树的效果。

棋盘有了，但这个棋盘只是块木头，似乎并不能下棋。要能在这个棋盘上下棋，最起码得点出棋子出来才行。看来，我们还需要对这个棋盘进行深加工才行。

不过，在继续工作之前，我们需要先来学点额外知识。

5
妆扮DOM对象

DOM对象是一个JavaScript对象吗？

这个问题似乎问得很奇怪啊！莫非这还有啥疑问的吗？这一点也不奇怪，能提出这种问题的朋友，想必也是那种从小喜欢拆闹钟，或者跑到电视机后面看看人影子是怎样出来的，喜欢刨根问底的人。

其实，DOM对象并不是一个纯粹的JavaScript对象！

为什么说DOM对象并非一个纯粹的JavaScript对象呢？因为DOM树是网页的数据结构，浏览器是将其实现为土生土长的原生对象的，为的是处理网页数据时的高效率。显然，在需要快速处理和呈现复杂网页结构的情况下，用JavaScript那种高度抽象的数据来描述，以及用解释性的环境来处理的话，效率肯定跟不上。

因此，DOM对象都是紧凑高效的本机数据对象，这些对象大都是以本机二进制的接口模型出现。在IE中，DOM对象是一个个COM接口对象，而在Firefox里也有自己的接口模型。在浏览器扩展开发的工作中，我们可以直接建立相关DOM对象的接口，并通过访问接口的属性和方法来操纵这些DOM对象。

但为什么JavaScript又能像操作一般的JavaScript对象一样来操纵这些DOM对象呢？原来，浏览器把这些接口对象进行了再包装，让它们看起来像一个JavaScript对象。一般来说，浏览器都会为每一种DOM元素准备一个特殊的内置原型，而每一个JavaScript形式的DOM对象都是连接到相应的内置原型中，并将自己的属性和方法与DOM本机接口的属性和方法绑定到一起。

JavaScript形式的DOM对象往往被称为“包装的对象”（wrapped object），所以不是一个纯粹的JavaScript对象。DOM对象的固有属性和方法是被绑定死的，虽然我们可以遍历到这些属性和方法，但不能删除这些固有属性，也不能更改这些属性的类型。不过，我们却能给他们添加新的属性和方法，这点和一般的JavaScript对象一样。

因为有了浏览器的包装工作，我们才可以像使用一般JavaScript对象一样来操纵DOM对象，从而操纵网页内容，呈现生动的网页效果。

从某种意义上讲，我们可以把DOM树当作表现层的东西。但除了DOM本身的树形结构之外，我们还可以把DOM组织成需要的逻辑层数据结构，比如围棋棋盘的二维数组结构。显然，19X19的二维数组结构更适合于围棋子间的关系计算，树形结构肯定不适合。然后，针对DOM对象的鼠标事件，并按围棋规则控制棋子及其呈现，这可以算作控制层。这不就可以实现下围棋的目标了吗?

看来，我们还需要对DOM对象进行再加工，让DIV同时实现逻辑层、表现层和控制层的功能，才能让棋盘活起来。

俗话说，人靠衣服马靠鞍。土生土长的东西总要包装一下才能上得了台面。同样，在编程中，有些原生的对象也一定要精心打扮一番，才能体现一个优雅的面貌，这就叫封装。封装有两个作用：一是让代码规范好用，二是转换编程的思维角度。

现在，我们的思维得换一下。为了实现能下围棋的网页程序，我们不能一开始就思考表现层的问题，而应该抓住围棋的逻辑关系这条主线去思考。对于一个经验丰富的程序员来说，一个好的程序往往是首先从程序逻辑设计开始的，因为一个程序在逻辑上的正确总是第一位的。

当然，从其他角度入手也是可以的，比如做一些技术验证工作往往很必要。这些技术验证的结论是会给逻辑思考提供丰富素材的。前面我们所做的围棋棋盘虽然完全是从表现层看问题，但却让我们知道了这一切确实是可行的。这就足够了，现在是时候回归逻辑主线的时候了。

现在，我们思考围棋程序的逻辑模型时，脑袋里不能是DOM对象或者 标签，我们需要的是棋盘、棋位和棋子这些概念。

于是，我们可以把围棋棋盘看成是一个19X19的方块，每个方块就是一个棋位，一个棋位只有三种状态：无子、黑子和白子。因此，我们可以定义一个二维数组Board来当棋盘，定义一个类Site来做棋位，Board的每个元素就是Site类的对象，而Site类应该有一个dot状态来表示棋子的情况。

好了，现在可以来重新规划和设计这个围棋网页程序了。我们将做一个标准的网页框架，设计一个Site类来表示棋位，其中的dot属性表示棋子准状态。再定义一个叫

Board二维数组变量。除此之外，我们还可以定义一些CSS样式，以方便设置棋位的显示特征。

现在来看go3.htm文件的代码：

http://www.leadzen.cn/Books/WuTouJavaScript/2/go3.htm

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

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
  <title>禅棋传说</title>
  <style type="text/css">
    div { position: absolute; width: 23px; height: 23px; }
    .B0 { background-image: url('B0.gif'); }
    .B1 { background-image: url('B1.gif'); }
    .B2 { background-image: url('B2.gif'); }
    .B3 { background-image: url('B3.gif'); }
    .B4 { background-image: url('B4.gif'); }
    .B5 { background-image: url('B5.gif'); }
    .B6 { background-image: url('B6.gif'); }
    .B7 { background-image: url('B7.gif'); }
    .B8 { background-image: url('B8.gif'); }
    .BX { background-image: url('BX.gif'); }
    .D0 { background-image: url('D0.gif'); }
    .D1 { background-image: url('D1.gif'); }
    .C0 { background-image: url('C0.gif'); }
    .C1 { background-image: url('C1.gif'); }
  </style>
</head>
<body>
  <script type="text/javascript">
  //<![CDATA[
    var Site = //定义一个棋位类
    {
      Create: function(x, y) //棋位类的构造函数
      {
        var me = document.createElement("div"); //建一个div对象，将其扩展并封装成棋位。
        document.body.appendChild(me); //附加到DOM树，实现棋位的呈现。

        me.x = x; //记录棋位的X坐标
        me.y = y; //记录棋位的Y坐标
        me.style.left = x * 23 + "px"; //设置棋位水平方向的绝对位置
        me.style.top = y * 23 + "px"; //设置棋位垂直方向的绝对位置
```

```
        //计算并准备棋位的背景式样
        var s = ((x-9)%9?0:(x-9)/9)+1+(((y-9)%9?0:(y-9)/9)+1)*3;
        me._backStyle = "B" + ((s==4&&(x/3)%2==1&&(y/3)%2==1) ? "X" : s);

        me.Fill = this.Fill;  //关联一个填充棋位的方法。
        me.Fill(); //初始填充空子。

        return me; //返回棋位对象，其实是一个封装了的div对象。
    },

    Fill: function(dot)   //填充棋子的方法
    {
        if ( dot == undefined )
            this.className = this._backStyle   //如果无子，就设置为背景式样。
        else
            this.className = "D" + dot;        //有子，就设置为相应的式样。
        this.dot = dot;   //保存棋子状态
    }
};

var Board = new Array(19); //全局的Board数组，表示棋盘
for(var x = 0 ; x < 19; x++)
{
    Board[x] = new Array(19);
    for(var y = 0; y < 19; y++)
        Board[x][y] = Site.Create(x, y);   //按位置创建棋位对象。
    };
  //]]>
  </script>
</body>
</html>
```

这些代码运行之后，也会生成和前面那个一样的棋盘。不过，我们的程序主线已经不再考虑表现的问题，而转向围棋逻辑的描述。以前的div元素也被我们封装成了Site类的对象，同时给div元素扩展了若干属性和方法。其中的x,y属性指明当前棋位的纵横坐标，这将为我们计算棋子关系提供方便；而dot属性表示棋位的状态，undefined表示无子，0表示黑子，1表示白子。此外，我们准备好了一个Fill方法，当需要给棋位填子时，就可调用此方法。

将DOM对象包装成符合我们自己业务思想的JavaScript对象是一种非常好的经验。特别是在当前流行的AJAX应用中，这种封装就显得更加重要。比如在网页模式的电子表格应用中，就必须把各种<td>或 标签元素封装成电子表格单元对象，这样才能方便电子表格的逻辑处理。比如在网页电子地图中，也必须把相关的 或<img>标签元素封装成经纬线分割的地图块，这样才能适应地理逻辑的需要。

因此，我们的网页围棋程序也要进行DOM元素的封装。在后面的运用中，我们的代码将从这样的封装中受益。

好了，继续我们的围棋事业。接下来，要开始实现下棋了。

6
响应DOM事件

几乎每一个DOM对象都为我们提供了一堆事件属性，只要写上响应这些事件的代码，就可以让事件驱动我们的程序做出必要的动作。

实现下棋，需要在单击鼠标时把棋子放到相应的位子上。这就需要响应DOM对象的鼠标事件。

响应DOM对象的事件有两种方式：一种是直接在HTML标签的事件属性里写JavaScript代码的静态绑定方式；另一种是通过程序将DOM对象的事件属性与事件处理函数关联的动态绑定方式。

例如，下面的html文件就是一个静态绑定的例子：

```
<html>
  <body>
    <div onclick=" alert( 'This is a  ' + this.tagName)" >Click Me!</div>
  </body>
</html>
```

再如，下面的html文件展示了另一种动态绑定的例子：

```
<html>
  <body>
    <div id=" aDiv"  >Click Me!</div>
    <script>
      aDiv.onclick = ClickMe;
      function ClickMe()
      {
        alert( "This is a  " + this.tagName);
      };
    </script>
  </body>
</html>
```

这两个html文件都是可以打开运行的，单击其中的内容都会输出“This is a DIV”。

在这里我们会注意到一点，this参数都是触发事件的DOM对象本身，这显然是一个好的消息。这样，我们在写事件处理代码时，通过this参数即可知道是哪个DOM对象触发消息的了。

但在静态绑定时，有的朋友经常会犯一个小错误，就是把事件绑定写成了下面的这种形式：

```
<div onclick=" ClickMe" >Click Me!</div>
```

这种写法的目的本是要将 元素的onclick事件绑定到ClickMe函数上，但运行的时候却显示“ClickMe未定义”的错误。为什么不能这样写？这得从静态绑定事件的原理说起。

当我们写下onclick = “…”的时候，实际上可以看成下面的等价形式：

```
aDiv.onclick = function () { … };
```

也就是说，标签的事件属性的值，也就是引号内的内容，将被认为是一段JavaScript代码，这段代码也将成为一个匿名函数内的代码，而这个匿名函数才是真正的事件处理函数。

显然，单独的“ClickMe”根本就不是一个有效的JavaScript语句，这种写法就是错误的。

我们再来看看另一种静态绑定形式，这种形式虽不一定是错误的，但有时却会出问题：

```
<div onclick=" ClickMe()" >Click Me!</div>
```

同样，它等效于下面的代码：

```
aDiv.onclick = function () { ClickMe() };
```

这个匿名函数并没有任何错误。但要注意的是，此时把ClickMe函数内的this参数当作触发事件的DOM对象本身，就会出问题了。这里的this并非触发事件的DOM对象本身，而是全局的window对象。

为什么会这样呢?

当事件触发时，浏览器会调用aDiv.onclick()，也就是调用这个匿名函数。显然，此时匿名函数的this参数将是触发事件的DOM对象。并且，也只有在这个匿名函数的作用域上下文中，this参数才是响应事件的DOM对象。从这个匿名函数中调用ClickMe()时，并没有把this传递给ClickMe函数。作用域跳到ClickMe函数内时，this只是默认的window对象。所以，在这种情况下，你得不到触发事件的DOM对象。

那么，静态绑定事件应该怎样写呢?

显然，第一种写法肯定不行。而第二种写法对于不关心this参数，或者无须判断当前响应事件的DOM对象时，也是完全可行的。如果要让事件处理代码知道当前DOM对象是谁，可以给事件处理函数定义一个参数，并将当前的this带进去。如下所示:

```
<div onclick=" ClickMe(this)" >Click Me!</div>
```

还可以利用函数的call或者apply方法来传递默认的this参数，如下所示:

```
<div onclick=" ClickMe.call(this)" >Click Me!</div>
```

好了，我们的围棋也需要响应用户事件。不过，我们的 元素，应该叫棋位Site对象，是动态生成的，因此只能采用动态绑定的方法。当给棋位绑定了onclick事件处理函数后，就可以用鼠标下棋子了。

先来看看增加事件绑定后的代码，下面是go4.htm文件:

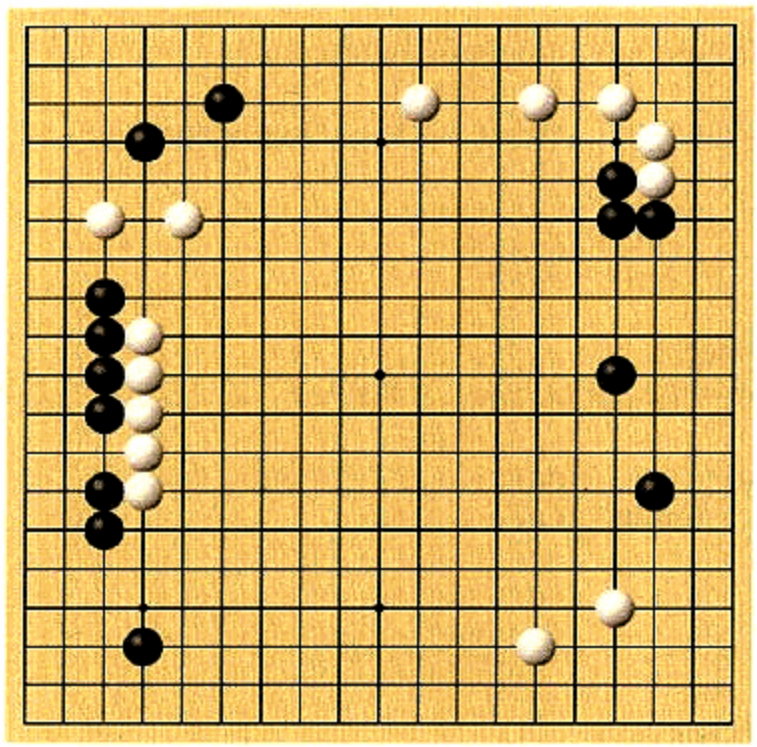
http://www.leadzen.cn/Books/WuTouJavaScript/2/go4.htm

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
  <meta http-equiv=" Content-Type" content=" text/html;charset=utf-8" />
  <title>禅棋传说</title>
  <style type=" text/css" >
    div { position: absolute; width: 23px; height: 23px; }
    .B0 { background-image: url( 'B0.gif' ); }
    .B1 { background-image: url( 'B1.gif' ); }
    .B2 { background-image: url( 'B2.gif' ); }
    .B3 { background-image: url( 'B3.gif' ); }
    .B4 { background-image: url( 'B4.gif' ); }
    .B5 { background-image: url( 'B5.gif' ); }
    .B6 { background-image: url( 'B6.gif' ); }
    .B7 { background-image: url( 'B7.gif' ); }
    .B8 { background-image: url( 'B8.gif' ); }
    .BX { background-image: url( 'BX.gif' ); }
    .D0 { background-image: url( 'D0.gif' ); }
    .D1 { background-image: url( 'D1.gif' ); }
    .C0 { background-image: url( 'C0.gif' ); }
    .C1 { background-image: url( 'C1.gif' ); }
  </style>
</head>
<body>
  <script type=" text/javascript" >
  //<![CDATA[
    var Site = //定义一个棋位类
    {
      Create: function(x, y) //棋位类的构造函数
      {
        var me = document.createElement( "div" ); //建一个div对象，将其扩展并封装成棋位。
        document.body.appendChild(me); //附加到DOM树，实现棋位的呈现。

        me.x = x; //记录棋位的X坐标
        me.y = y; //记录棋位的Y坐标
        me.style.left = x * 23 + "px"; //设置棋位水平方向的绝对位置
        me.style.top = y * 23 + "px"; //设置棋位垂直方向的绝对位置

        //计算并准备棋位的背景式样
        var s = ((x-9)%9?0:(x-9)/9)+1+(((y-9)%9?0:(y-9)/9)+1)*3;
        me._backStyle = "B" + ((s==4&&(x/3)%2==1&&(y/3)%2==1) ? "X" : s);
```

```
    me.Fill = this.Fill;   //关联一个填充棋位的方法。
    me.Fill(); //初始填充空子。

    me.onclick = this.Play; //绑定onclick事件到Play方法。

    return me; //返回棋位对象，其实是一个封装了的div对象。
},

Fill: function(dot)    //填充棋子的方法
{
    if ( dot == undefined )
        this.className = this._backStyle   //如果无子，就设置为背景式样。
    else
        this.className = "D" + dot;        //有子，就设置为相应的式样。
    this.dot = dot;    //保存棋子状态
},

Play: function()   //行棋方法，由onclick事件触发
{
    if ( this.dot == undefined )
    {
        this.Fill(current); //填入当前该填的子
        current ^= 1;      //用1来异或，正好反转黑白棋子
    };
}
    };

    var Board = new Array(19); //全局的Board数组，表示棋盘
    for(var x = 0 ; x < 19; x++)
    {
        Board[x] = new Array(19);
        for(var y = 0; y < 19; y++)
            Board[x][y] = Site.Create(x, y);   //按位置创建棋位对象。
    };

    var current = 0;   //当前要下的子，0表示黑子，1表示白子，互相交替。

//]]>
</script>
</body>
</html>
```

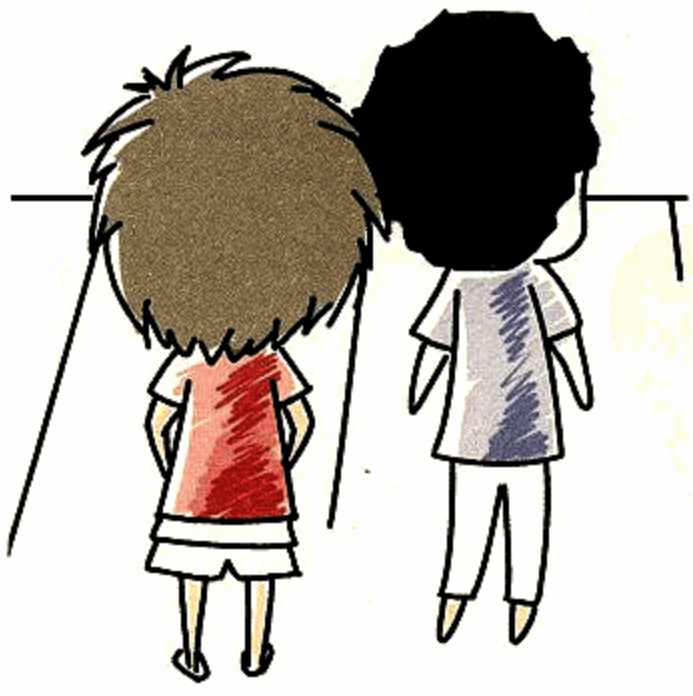
其中，我们给棋位Site类增加了一个Play方法，并在构造函数中将其绑定到Site对象的onclick事件。由于黑白棋子是交替行棋的，我们还定义了一个全局变量current来记录当前该走的棋子。每次此用户单击某棋位时，将调用Play方法，在单击的位置放上当前该下的棋子，然后current被反转为对方棋子。

现在我们来看一下程序运行的效果图，哈哈，不错吧！



图中，下棋的效果看起来非常棒。虽然我们还没有完全实现整个围棋程序，但已经有了一个大的程序框架。更重要的是，我们已经学会了给DOM元素绑定事件的方法。

继续努力，接下来，我们将实现落子有声！



7
播放声音

用JavaScript来播放声音不是太好做。不太好做的原因不是JavaScript本身的问题，主要是浏览器的兼容问题。

JavaScript本身没有提供声音播放函数，一般都是利用那些可以处理声音的HTML元素来操控声音的。也有用JavaScript创建ActiveX对象来处理控制声音，但由于其局限性，我们一般也很少用。

HTML中处理声音的标签有两个：<embed>和<bgsound>。<embed>可以给网页嵌入丰富的多媒体内容，不但包括声音还可以包含视频等，可谓功能强大。而<bgsound>是专门用来给网页嵌入背景音乐的，功能明确，使用简洁。

为了给下棋添加声音效果，首先我们准备一个名叫play.wav的声音文件。这个play.wav文件播放时，将发出“啪”的一声，这是一个棋子落下的声。我们将利用这个声音文件来逐一探讨各种控制声音的方法。

我们先来研究一下<embed>标签。下面的HTML文件在打开时就可以播放声音：

```
<html>
  <body>
    <embed src=" play.wav" ></embed>
  </body>
</html>
```

打开这个网页时，我们就能听到“啪”的一声。不过，如果想在Firefox下听到声音，一定要先安装可以播放声音的多媒体插件才行。而IE一般都行，因为Windows系统大都捆绑有微软自己的媒体播放器组件。

但是，这个网页打开时，也会同时显示一个媒体播放的控制面板。而且，第一次发声之后，要再次发声就要单击那个控制面板的播放按钮。这肯定不爽，下棋时总不能出现个播放器来发声吧，这不是我们期望的东西。

不过，我们可以设置<embed>标签的属性，让它不要显示出来，并不要自己发声，当我们需要时才控制其发声。例如，下面的HTML文件就演示了这种效果。

```
<html>
  <body>
    <embed name=" sound" src=" play.wav" hidden=" true" autostart=" false" ></embed>
    <span onclick=" sound.play()" >Click to play sound</span>.
  </body>
</html>
```

其中，<embed>标签被命名为sound，其hidden属性设为true将隐藏播放器控制面板，而autostart属性设为false将不会自动播放。只有在单击另一个标签时，再调用<embed>对象的play()方法来控制发声。

这一切在IE浏览器下表现得非常不错，但在Firefox中却严重失效！原因是Firefox的<embed>标签并不支持play()方法。

在网站程序开发中，浏览器的兼容问题从来都是让人头痛的第一病因。但作为一个负责的网站程序员，我们又不可能只考虑IE这一种浏览器，还必须兼顾到那些使用Firefox等其他浏览器的用户。

还好，对于这个问题，我们可以通过一种动态修改容器类标签内部HTML代码的方式来解决。比如，我们对一个标签的innerHTML属性动态写入<embed>那些内容，将可以导致浏览器重新呈现标签内部的内容，从而重新播放<embed>嵌入的声音。

例如，下面的HTML文件：

```
<html>
  <body>
    <span id=" sound" ></span>

    <span onclick=" playsound()" >Click to play sound</span>
```

```
    <script>
      function playsound()
      {
        sound.innerHTML = "<embed src=' play.wav' hidden=' true' ></embed>";
      };
    </script>
  </body>
</html>
```

这种方式在IE和Firefox中都有效。当然，对于IE来说，这种方式显然没有调用play()方法来得轻松。而且，由于IE在处理元素呈现的方式不同，还会偶尔出现页面闪烁的情况。于是，我们需要再看看其他的方法。

接下来，再来讨论一下<bgsound>标签。

<bgsound>标签最早是由IE引入的，目的是为了给网页嵌入背景音乐。早期的Firefox版本是不支持这个标签的，不过后来为了蚕食IE的市场，Firefox也加入了对<bgsound>标签的支持。

<bgsound>标签使用起来比<embed>简单些，下面的代码即可加入背景声音：

```
<html>
  <body>
    <bgsound src=" play.wav" />
  </body>
</html>
```

这个HTML文件和最开始的一样，在打开时会听到“啪”的一声，却没有多余的媒体播放器控制面板。看来，对于我们的围棋程序来说，使用<bgsound>标签来控制声音更适合些。

不过在IE中，<bgsound>对象却没有像<embed>对象那样的play()方法，来控制声音的播放。但有一个变通的方法，就是动态修改<bgsound>的src属性，可以达到控制声音的目的。下面的html文件演示了这一方法：

```
<html>
  <body>
    <bgsound id=" sound" />
    <span onclick=" sound.src = 'play.wav' " >Click to play sound</span>
  </body>
</html>
```

其中，由于没有为<bgsound>标签设置初始的src属性，所以打开网页时并不会听到声音。而在单击标签时，将会动态地设置<bgsound>标签的src属性，导致IE浏览器重新处理<bgsound>标签的呈现，从而播放出声音。而且，由于<bgsound>标签并非一个显示用的标签，所以在发出声音时，并不会出现页面闪烁情况。

但这种方法在Firefox也是无效的！目前，在Firefox内部，<bgsound>标签实际是被转化为一个<embed>标签来处理背景声音的，而且我们也无法动态设置其src属性。即时可以通过setAttribute方法来设置和修改src属性，但也不会导致Firefox重新呈现<bgsound>，因此还是不能控制声音。

看来，在Firefox中也只能用动态写入标签的形式来处理了。一般来说，用document.write()方法来动态写入标签，或者动态修改一个标签的innerHTML属性，一定会导致这些标签重新呈现。我们动态修改某容器标签的innerHTML来试试，请看下面的html文件：

```
<html>
  <body>
    <span id=" sound" ></span>

    <span onclick=" playsound()" >Click to play sound</span>

    <script>
      function playsound()
      {
        sound.innerHTML = "<bgsound src=' play.wav' />" ;
      };
    </script>
  </body>
</html>
```

在Firefox中测试一下。嗯，非常不错，单击后可以听到声音了。再用IE打开来测试一下，问题又来了，这种方式在IE中反而无效了！

看来，浏览器兼容问题真是让人头痛啊！

世界上从来没有完美的事情。在编程中如果过度追求完美，只会为程序员自身带来痛苦。聪明的程序员总是喜欢在了解各方面优劣性之后，找一个能解决实际问题的平衡点。只有这样，我们才能在现实世界里做一个快乐的程序员啊！

经过多方权衡，我们决定采用<bgsound>标签来控制声音播放，并根据浏览器类型

的不同采用不同的处理方式。好了，开始让落子有声吧！

按这种方案实现的HTML示例代码如下：

```
<html>
  <body>
    <span onclick=" sound.play()" >Click to play sound</span>

    <script>
      if (navigator.userAgent.indexOf( ' MSIE ') > -1) //IE浏览器
      {
        var sound = document.body.appendChild(document.createElement( "bgsound" ));
        sound.play = function ()
        {
          this.src = "play.wav" ;
        };
      }
      else   //Firefox等其他浏览器
      {
        var sound = document.body.appendChild(document.createElement( "span" ));
        sound.play = function (wav)
        {
          this.innerHTML = "<bgsound src=' play.wav' >" ;
        };
      };
    </script>
  </body>
</html>
```

在这个示例中，我们根据window对象的navigator属性判断浏览器的类型，然后建立一个全局的sound对象。我们对这个全局sound对象进行了简单的封装，尽管在不同浏览器中其DOM元素不同，但对外都表现出一个统一的sound对象。而且，我们还为sound对象定义了一个方法，来控制声音的播放。虽然在不同浏览器下实现play的方式不同，但对于JavaScript来说可以采用相同的调用方式。

其中，判断浏览器类型是根据navigator的userAgent是否含有某些特性来判断的，这是JavaScript判断浏览器类型的标准方法。在编写兼容多种浏览的JavaScript程序中，这会经常用到。为了方便大家参考，我们把微软的ASP.NET AJAX中一段设置浏览器类型的初始化代码展示给大家看一下：

```
Sys.Browser = {};

Sys.Browser.InternetExplorer = {};
Sys.Browser.Firefox = {};
Sys.Browser.Safari = {};
Sys.Browser.Opera = {};
```

```
Sys.Browser.agent = null;
Sys.Browser.hasDebuggerStatement = false;
Sys.Browser.name = navigator.appName;
Sys.Browser.version = parseFloat(navigator.appVersion);

if (navigator.userAgent.indexOf( ' MSIE ') > -1) {
    Sys.Browser.agent = Sys.Browser.InternetExplorer;
    Sys.Browser.version = parseFloat(navigator.userAgent.match(/MSIE (\d+\.\d+)/)[1]);
    Sys.Browser.hasDebuggerStatement = true;
}
else if (navigator.userAgent.indexOf( ' Firefox/' ) > -1) {
    Sys.Browser.agent = Sys.Browser.Firefox;
    Sys.Browser.version = parseFloat(navigator.userAgent.match(/ Firefox\/(\d+\.\d+)/)[1]);
    Sys.Browser.name = 'Firefox';
    Sys.Browser.hasDebuggerStatement = true;
}
else if (navigator.userAgent.indexOf( ' Safari/' ) > -1) {
    Sys.Browser.agent = Sys.Browser.Safari;
    Sys.Browser.version = parseFloat(navigator.userAgent.match(/ Safari\/(\d+(\.\d+)?)/)[1]);
    Sys.Browser.name = 'Safari';
}
else if (navigator.userAgent.indexOf( 'Opera/' ) > -1) {
    Sys.Browser.agent = Sys.Browser.Opera;
}
```

在这些代码中，不但判断了浏览器的类型，还提取了浏览器的版本信息。当然，这些代码只考虑了四大主流的浏览器特征，一般来说是足够了。如果真遇到什么其他浏览器，我们也可以自己加上相应的判断嘛。

顺便说一下，把不同浏览器的处理方式，整理为统一的接口形式，这也是一种封装。我们不能去改变既成的现实环境，唯有尽量去兼容和统一各种各样的形式，对上层代码封装出一致的模型和接口。这些封装就像花蕾外层的保护壳，当我们的程序从浏览器兼容性的深潭中绽放时，一定是一朵出淤泥而不染的白莲花。

好了，回到我们的棋局吧。现在已经有了可以发声的对象，该让落子有声了！

下面的**go5.htm**就是加上声音处理部分之后的**HTML**代码：

http://www.leadzen.cn/Books/WuTouJavaScript/2/go5.htm

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
  <title>禅棋传说</title>
  <style>
    div { position: absolute; width: 23px; height: 23px; }
    .B0 { background-image: url( 'O.gif' ); }
    .B1 { background-image: url( '1.gif' ); }
    .B2 { background-image: url( '2.gif' ); }
    .B3 { background-image: url( '3.gif' ); }
    .B4 { background-image: url( '4.gif' ); }
    .B5 { background-image: url( '5.gif' ); }
    .B6 { background-image: url( '6.gif' ); }
    .B7 { background-image: url( '7.gif' ); }
    .B8 { background-image: url( '8.gif' ); }
    .X { background-image: url( 'X.gif' ); }
    .D0 { background-image: url( 'D0.gif' ); }
    .D1 { background-image: url( 'D1.gif' ); }
    .X0 { background-image: url( 'X0.gif' ); }
    .X1 { background-image: url( 'X1.gif' ); }
  </style>
</head>
<body>
  <script type=" text/javascript" >
    var Site = //定义一个棋位类
    {
      Create: function(x, y) //棋位类的构造函数
      {
        var me = document.createElement( "div" ); //建一个div对象，将其扩展并封装成棋位。
        document.body.appendChild(me); //附加到DOM树，实现棋位的呈现。

        //me.className = "site"; //设置式样

        me.x = x;
        me.y = y;
        me.style.left = x * 23 + "px";
        me.style.top = y * 23 + "px";

        var s = ((x-9)%9?0:(x-9)/9)+1+(((y-9)%9?0:(y-9)/9)+1)*3;
```

```
    //if ( s == 4 && (x/3)%2==1 && (y/3)%2==1) s = "X";
    me._backStyle = ( s == 4 && (x/3)%2==1 && (y/3)%2==1) ? "X" : "B" + s;

    me.Fill = this.Fill;   //关联一个填充棋位的方法
    me.Fill(); //初始填充空子

    me.onclick = this.Play; //绑定事件

    return me;
},

Fill: function(dot)
{
    if ( dot == undefined )
        this.className = this._backStyle
    else
        this.className = "D" + dot;
    this.dot = dot;
},

Play: function()
{
    if ( this.dot == undefined )
    {
        this.Fill(current); //填入当前该填的子
        current ^= 1;      //用1来异或，正好反转黑白棋子
        sound.play();      //发出落子声
      };
    }
};

var Board = new Array(19); //全局的Board数组，表示棋盘
for(var x = 0 ; x < 19; x++)
{
    Board[x] = new Array(19);
    for(var y = 0; y < 19; y++)
        Board[x][y] = Site.Create(x, y);   //按位置创建棋位对象。
};

var current = 0;   //当前要下的子，0表示黑子，1表示白子

if (navigator.userAgent.indexOf( ' MSIE ') > -1) //IE浏览器
{
    var sound = document.body.appendChild(document.createElement( "bgsound" ));
    sound.play = function ()
    {
        this.src = "play.wav";
```

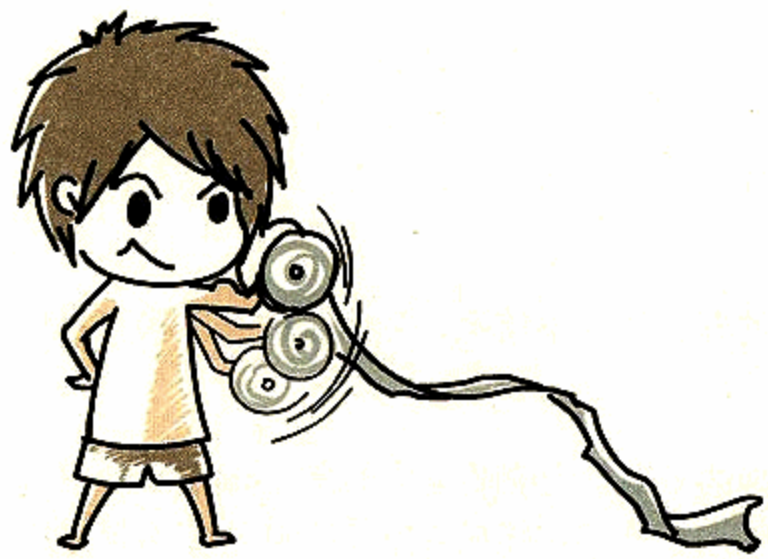
```
    };
  }
  else   //Firefox等其他浏览器
  {
    var sound = document.body.appendChild(document.createElement( "span" ));
    sound.play = function (wav)
    {
      this.innerHTML = "<bgsound src=' play.wav' >" ;
    };
  };

 </script>
</body>
</html>
```

啊哈，落子有声的感觉还真不错呢，就像真的在下棋一样啊。先歇一会，感受一下效果和气氛吧！

以上都是值得高兴的事情。但需要指出的是，**<bgsound>**目前并不是**xHTML**标准的标签。对于追求完美的朋友来说，这可是件不太高兴的事情。不过，事情并没有想象的那么坏，上面的代码还是可以顺利地通过**W3C**的**XHTML**验证的。为什么呢？因为，我们的**<bgsound>**标签都是用**JavaScript**来动态生成的，而**W3C**的验证并不知道动态生成的内容。

不管这算不算投机取巧，也算是个小小的补偿吧。还是开开心心地，继续下我们的棋吧！

8
别向复杂低头



虽然，已经落子有声，不过，还不能提子，我们还没有完成全部围棋的业务逻辑判断。接下来，我们一鼓作气，把行棋的线索、提子规则、打劫位子，还有悔棋等功能通通写完吧。

该如何记录行棋的线索？如何判断是否提子？打劫如何处理？悔棋又如何实现？这些似乎都有些复杂。但是，我们不能被复杂吓住了。比尔·盖茨有句经典的名言：永远别向复杂低头！

一些文静的程序员在面对复杂问题时，往往总会产生一点畏惧心理，这很正常。在我们觉得问题复杂的时候，往往意味着我们考虑得比较多。对任何问题尽量考虑周全，这是好事。特别对于程序员来说，这非常重要。至少这体现了程序员的一种认真的态度，也体现了对用户负责的精神。

一些活泼点的程序员在面对复杂问题时，喜欢先做再说，或者边做边学，这也对。对于复杂的局面，只有做进去了，才知道问题有多复杂。老是停留在思索的表面，畏难而不前进，也不是办法。至少边做边学，总是在不断进取，这至少体现了程序员的一种积极的心态。

而一些经验老道的程序员在面对复杂问题时，却总喜欢先跳出去。他们首先抛开那些复杂的技术问题，静下心来理清思路，把问题的来龙去脉摸清楚再说。用软件工程的话说就是：先把业务和需求搞懂再说。其实，只有把业务需求真正搞懂了，才可

能做出正确的系统。那些在交付试用时才发觉误解了用户需求的系统，往往都是想急于设计和开发的程序员干的好事。

诚然，在分析和设计时，确实是有些关系到需求能否实现的关键技术问题，是需要提前摸索的。这些关键技术问题的研究，能让我们在回答“能不能做”以及“如何做”时充满理智和信心。这些关键技术的提前研究被称为“技术验证”，是属于分析和设计的范畴。技术验证是为分析和设计服务的，虽然技术验证的结果可能会影响分析和设计工作，但不能反过来主导分析和设计工作，那样就本末倒置了。

系统分析员与程序员的一个显著区别就是：系统分析员是以业务需求为主线来思考和处理问题；而程序员总喜欢从纯技术层面来思考和解决问题。一般来说，系统分析员都是有多年编程经验的老程序员。他们经过无数次摸爬滚打，使自己的思想得到升华，从而具有了从总体到局部都把握整个系统大方向的能力。

好了，空话随后再说，先来看看如何解决围棋那些算法吧。

先思考一下，我们下一个子时要考虑哪些情况?

当然，第一是要下子的位置不能有子。如果这个位置无子，则要考虑下去这一个子是否能杀死对方的子。如果能杀死对方的子，就提掉那些死子，然后下这个子。如果不能杀死任何子，则要看看下了这一子之后，自己的关联的那片棋子是否还有气。如果下的这一子导致自己没气，按围棋规则是不能下这颗子的。

这么一描述，问题已经清楚了很多，至少经过以上的描述已经有了业务逻辑主线。再接着深入思考其中的两个更细的问题：一个是如何判断能否杀死对方的棋子?另一个是怎样判断一片棋是否还有气?

先思考一下杀死棋子的情况。当一个子落下时，它必然会影响其上下左右的四个子。如果其上下左右的子有对方的子，就要从对方那个子开始判断相连的对方棋子是否还有气。如果下这一子导致上下左右任何一块对方的子没气了，则杀死那块对方的子。

好了，判断杀死棋子的问题又归结到判断一片棋是否有气的问题上来了。显然，我们的思路又进了一步。那就继续理清思路，接下来看看该怎样判断一片棋是否有气?

判断一片棋是否有气总是从一个地方开始的。对于要杀死的子，是从一个有子的位子开始；对于不能杀子的能否落子判断，是从一个空位开始的。不管怎样，总要从

这个位子开始去找出那些相连的棋子。

我们知道，一个棋位的上下左右的棋子才可能是与其相连的。于是，我们需要先找出上下左右的子，然后再以找到的子为基础，继续找新的上下左右的子。对于一个相连的位置，如果无子，则算这块棋的一口气；如果是对方的子，则不能算气；如果是我方子，则作为继续寻找相连棋子的依据。

显然，我们需要记录哪些位置已经找过，不能找重复了。同时，在这一过程中，我们并不是要计算有多少子相连，也不是要计算一块棋有多少气，我们的目标是计算一块棋够不够气。如果发现一块棋有两口以上的气，则表示这块棋不能被杀死或者这个位子可以落子，就无须再继续下去了。

下一个子会发生的情况已经分析得比较清楚了，现在回过头来看看如果控制下棋的线索。显然，为了能让用户悔棋，我们必须记录下每一步行棋的过程。如果可以下一个子，就要记下这个子的位置及状态，如果这一步杀死了一些子，还需同时记录这些被杀死的子的位置。这样在悔棋的时候，一切都可以退回原态。

此外，还需考虑打劫的情况。就是如果被提的子是一个子，下一步对方不能再提回来，必须多隔一步。为此，我们针对这种情况需要特殊记录可能的打劫状态，并在隔一步之后清除这种状态。

到这里，我们的分析也算差不多了吧。应该不需要写一份需求分析报告吧，毕竟是一个网页小程序嘛。接下来该落实如何实现这些需求，并开始进行设计工作。

我们需要设计两个新算法来作为棋位对象的方法：一个用于计算可以杀死的棋子，取名为Kill；另一个用于计算是否气紧，取名为Tight。其中，Kill方法将用到Tight方法。

Kill的意思就是“杀”的意思，其返回当前下子位相关的可以被杀死的棋子的集合。Tight是“紧”的意思，其返回当前位相关的已经气紧的棋子集合。这些标识符的命名也是非常重要的，好命名一看就懂，写代码也爽些。

不管是Kill方法还是Tight方法，都会根据当前位来取上下左右的棋位进行分析。假如当前棋位为Board[x][y]，则左为Board[x-1][y]，右为Board[x+1][y]，上为Board[x][y-1]；下为Board[x][y+1]。

直接从棋盘数组里取相邻棋位并没有考虑边界条件。如果当前棋位在边或者角上，有些相邻位就不存在，就会导致数组超界错误。看来，我们需要封装一个专门取棋位的函数GetSite，如果超界就返回undefined而不要抛出异常。

对于上下左右的棋位坐标，可以分四种情况分别处理。分情况处理最后的表现形式无非是几个条件判断分支结构，有点像数学里面的分段函数的写法。不过，我们如果能找到各种条件的规律，就可以把分段函数写成连续函数，从而用更加适合计算机的循环结构来处理问题。

其实，找出上下左右四个位置的规律很简单，就是：*X*和*Y*方向的差值总在-1到1之间，并且*X*和*Y*方向的差值有且只有一个不为0。其规律可以用下面的JavaScript代码来表示：

```
for(var dx=-1;dx<=1;dx++)
  for(var dy=-1;dy<=1;dy++)
    if(!dx ^ !dy)
    {
      ...
    };
```

接下来的相连判断其实就是在上面循环中嵌套一些递归处理，将判断逻辑从一个相邻位转移到另一个相邻位，直到递归条件结束。在递归中，会用到一个记录已处理棋位的数组，使得递归过程不会处理重复的棋位，防止无限递归。

不过，对于JavaScript来说，递归调用太深，会产生很长的作用域链，不仅大量消耗内存，而且效率低下。所以，我们应该把相连判断的递归算法转化为基于堆栈的循环算法，而这个堆栈正好可以与记录已处理棋位的数组结合起来使用。

行棋的线索应该是一个动态数组，数组元素记录每一步的棋位，我们将其命名为Tracks。为了能悔棋，每一个棋步还可能要记录其杀死的棋子集合，悔棋时需要把这些棋子还原回去。至此，我们还需要专门的棋步类Step。

最后就是处理打劫的特殊状态。打劫的地方只是一个单独的棋位，该棋位只影响随后的一步棋，过了这步棋就没有意义了。所以，我们可以定义一个全局变量rob来记录这个棋位。如果出现打劫，就把打劫位记录到这个rob变量中，下一步棋之后清除rob变量即可。

悔棋的操作呢？我们用鼠标右键触发好了。对于可见的DOM对象来说，鼠标右键会引发一个oncontextmenu的事件，就是弹出一个菜单。我们就利用它好了，只是只需接管事件，调用悔棋函数Back()，但不弹出菜单。同时，并非每个棋位都需要绑定这个oncontextmenu事件，因为悔棋并不需要针对棋位，它是全局性的操作。我们就把悔棋操作绑定给<body>元素吧。

好了，实现上面的设计吧！于是，就有了go6.htm文件：

http://www.leadzen.cn/Books/WuTouJavaScript/2/go6.htm

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd" >
<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
  <meta http-equiv=" Content-Type" content=" text/html;charset=utf-8" />
  <title>禅棋传说</title>
  <style type=" text/css" >
    div { position: absolute; width: 23px; height: 23px; }
    .B0 { background-image: url( 'B0.gif' ); }
    .B1 { background-image: url( 'B1.gif' ); }
    .B2 { background-image: url( 'B2.gif' ); }
    .B3 { background-image: url( 'B3.gif' ); }
    .B4 { background-image: url( 'B4.gif' ); }
    .B5 { background-image: url( 'B5.gif' ); }
    .B6 { background-image: url( 'B6.gif' ); }
    .B7 { background-image: url( 'B7.gif' ); }
    .B8 { background-image: url( 'B8.gif' ); }
    .BX { background-image: url( 'BX.gif' ); }
    .D0 { background-image: url( 'D0.gif' ); }
    .D1 { background-image: url( 'D1.gif' ); }
    .C0 { background-image: url( 'C0.gif' ); }
    .C1 { background-image: url( 'C1.gif' ); }
  </style>
</head>
<body>
  <script type=" text/javascript" >
  //<![CDATA[
    Array.prototype.indexOf = function (item)  //给数组扩展一个indexOf方法
    {
      for ( var i=0; i<this.length; i++)
        if (this[i] == item)
          return i;
      return -1;
    };

    var Site =  //定义一个棋位类
    {
      Create: function(x, y)  //棋位类的构造函数
      {
        var me = document.createElement( "div" ); //建一个div对象，将其扩展并封装成棋位。
        document.body.appendChild(me); //附加到DOM树，实现棋位的呈现。
        me.x = x;  //记录棋位的X坐标
        me.y = y;  //记录棋位的Y坐标
        me.style.left = x * 23 + "px"; //设置棋位水平方向的绝对位置
        me.style.top = y * 23 + "px"; //设置棋位垂直方向的绝对位置
        var s = ((x-9)%9?0:(x-9)/9)+1+(((y-9)%9?0:(y-9)/9)+1)*3;  //计算并背景式样
```

```
    me._backStyle = "B" + ((s==4&&(x/3)%2==1&&(y/3)%2==1) ? "X" : s);
    me.Fill = this.Fill;   //关联一个填充棋位的方法。
    me.Tight = this.Tight; //关联计算紧气方法。
    me.Kill = this.Kill;   //关联计算死子方法。
    me.onclick = this.Play; //绑定onclick事件到Play方法。
    me.Fill(); //初始填充空子。
    return me; //返回棋位对象，其实是一个封装了的div对象。
},
Fill: function(dot, going)    //填充棋子的方法
{
    if ( dot == undefined )
        this.className = this._backStyle   //无子，就设置为背景式样。
    else
        this.className = (going ? "C" : "D") + dot;
    this.dot = dot;    //保存棋子状态
},
Play: function()   //行棋方法，由onclick事件触发
{
    if ( this.dot == undefined ) //无子
    {
        var deads = this.Kill(current^1);  //计算可以杀死的子
        if (deads.length == 1 && this == rob) return; //打劫状态
        for(var i=0; i<deads.length; i++)
            deads[i].Fill();
        if(i==1)
            rob = deads[0]  //记录打劫位置
        else if (i>0 || !this.Tight(current))
            rob = null  //清打劫位
        else return;
        sound.play();      //落子有声！
        var step = Tracks[Tracks.length-1];
        if(step) step.site.Fill(step.site.dot);
        this.Fill(current, true); //填入当前该填的子
        Tracks.push( new Step(this, deads) );
        current ^= 1;      //用1来异或，正好反转黑白棋子。
    };
},
Tight: function (dot)   //计算紧气的块
{
    var life = this.dot == undefined ? this : undefined; //当前位无子则算一口气
    dot = dot == undefined ? this.dot : dot;
    if (dot == undefined) return undefined;
    var block = this.dot == undefined ? [] : [this];
    var i = this.dot == undefined ? 0 : 1;
    var site = this;
    while (true)
```

```
        {
          for(var dx=-1;dx<=1;dx++) for(var dy=-1;dy<=1;dy++) if(!dx^!dy)
          {
            link = GetSite(site.x + dx, site.y + dy);
            if (link)  //有位
              if (link.dot != undefined)  //有子
              {
                if (link.dot == dot && block.indexOf(link) < 0 )
                  block.push(link);
              }
              else if (!life)
                life = link
              else if (life != link)
                return undefined;  //如果有两口气以上则无须再算
          };
          if ( i >= block.length) break;
          site = block[i];
          i ++;
        };
        return block;  //返回只有一口气的块
      },
      Kill: function(dot)    //计算杀死的子
      {
        var deads = [];
        for(var dx=-1;dx<=1;dx++) for(var dy=-1;dy<=1;dy++) if(!dx^!dy)
        {
          var site = GetSite(this.x + dx, this.y + dy);
          if (site && (site.dot == dot))
          {
            var block = site.Tight();
            if (block) deads = deads.concat(block);
          };
        };
        return deads;  //返回可以提子的死子块
      }
    };

    var Board = new Array(19); //全局的Board数组，表示棋盘。
    var Tracks = [];   //行棋线索数组，数组元素是Step对象。
    var current = 0;   //当前要下的子，0表示黑子，1表示白子，互相交替。
    var rob = null;    //如果有打劫时，记录打劫位置。
    for(var x = 0 ; x < 19; x++)
    {
      Board[x] = new Array(19);
      for(var y = 0; y < 19; y++)
        Board[x][y] = Site.Create(x, y);   //按位置创建棋位对象。
    };
```

```
if (navigator.userAgent.indexOf( ' MSIE ') > -1) //为IE浏览器构造声音对象
{
   var sound = document.body.appendChild(document.createElement( "bgsound" ));
   sound.play = function(){this.src = "play.wav" };
}
else   //为Firefox等其他浏览器构造声音对象
{
   var sound = document.body.appendChild(document.createElement( "span" ));
   sound.play = function(){this.innerHTML = "<bgsound src=' play.wav' >" };
};
document.body.oncontextmenu = function()    //悔棋事件
{
   var step = Tracks.pop();
   if (step)
   {
      step.site.Fill();
      for (var i=0; i<step.deads.length; i++)
         step.deads[i].Fill(current);
      step = Tracks[Tracks.length-1];
      if (step) step.site.Fill(current, true)
      current ^= 1;      //反转黑白棋子。
   };
   return false;  //不弹出菜单。
};
```

```
      function GetSite(x, y)  //从棋盘取棋位的函数，越界不抛出异常。
      {
         if (x>=0 && x<19 && y>=0 && y<19)
            return Board[x][y];
      };
      function Step(site, deads)   //棋步类，记录每一步棋的状态
      {
         this.site = site;   //记录棋步的位置
         this.deads = deads; //记录被当前棋步杀死的棋子集合
      };

      document.onkeypress = function(event)
      {
         var k = (window.event ? window.event.keyCode : event.which) - 49;
         if(k<0 || k>1) return;
         for(var x=0; x<19; x++) for(var y=0; y<19; y++) Board[x][y].Fill();
         Tracks.length = 0;
         current = 0;
         with(goes[k]) for(var i=0; i<length;i+=3)
            Board[charCodeAt(i+1)-65][charCodeAt(i)-65].Fill(charCodeAt(i+2)-48);
      };
var goes= [ "AA0AB0AC1AE0AF1AG1AI0BA0BC1BE0BF0BG1BI0BK1CA1CB1CC1CD0CF0CG1CI0CK1\
```

```
DA0DB0DC0DD0DE0DF0DG1DH1DI0DJ0DK1EC1ED1EE1EF1EH1EK1FA1FB1FC1FE0FF1FG1FH1FI0FJ0\
FK1GA1GB0GC1GE0GF0GG0GG0GH0GI0GJ1HA0HB0HC0HD0HE0HI1HJ1IB1IF1JC1JE1JG1",
 "AA0AB0AC0AF1AG1AH0AI0AJ0AK1AM0AO0AP0AR0AS0BA0BB0BF1BG1BJ0BK0BL1BM0BR0BS0CA0CB1\
CC0CD0CG0CH0CI1CJ0CK0CL1CM0CO1CP0CQ0CR1CS0DA1DB1DC0DD0DE1DF1DH1DI0DJ1DK1DL0DN0\
DP1DQ1DR1DS0EB0ED1EE0EG1EI0EK0EM0EO1ER0ES1FB0FC1FE0FF0FG1FN1FP0FR0GA0GF1GH0GJ0\
GL0GR0GS0HC0HD0HF0HI0HK0HM0HN1HP0HQ0HR1IA0IB0IC1ID0IE1IK0IL1IM0IQ1IR0IS0JA0JB0\
JC1JE1JG1JI0JJ1JK1JL0JM1JN1JO0JQ1JR0JS0KA1KC1KD1KE0KG0KI1KJ0KK0KL0KM0KN0KO1KQ0\
KR0KS0LD0LE1LI0LJ1LL0LN1LO0LP0LQ1LR0LS0MD0MG0MI0MJ0MK0ML1MM0MN1MR1MS0NC0NF0NL1\
NO0NQ0NR0NS1OB0OC0OG0OL1OO0PA1PB1PC0PD0PI0PJ0PK0PL1PM0PN0PR0QA0QC1QD0QE1QF1QG1\
QH1QK1QP0RA0RB0RC0RD1RE1RF0RG0RH1RI0RL0SA0SB0SC0SD0SE0SF0SG0SH0SI1SJ1" ];
  //]]>
  </script>
</body>
</html>
```

终于，我们可以轻松下来下盘棋了！

从一个空白的网页，到可以下棋的网页程序，我们历经了一次短暂的开发旅途。在这个过程中，我们知道了什么是标准网页，了解了网页运行机理。我们探讨了DOM及封装DOM的方法，涉足网页事件绑定和处理，还从浏览器兼容的烂摊子里找出控制声音的方法。在征服了貌似复杂的业务逻辑难关之后，最终实现了我们既定的目标。

这算是敏捷开发吗？也许吧，如果你愿意这么认为。

这是精心设计的教程吗？也许吧，只要我们学到了知识。

不管怎样，最后的结果是，我们确实用HTML和JavaScript代码下了一盘很有意思的棋。而更有趣的是：这个程序的全部编码竟然没有超过200行！

不但如此，这盘棋中还隐藏一个鲜为人知的秘密。这个秘密的真相一旦大白于天下，必将掀起江湖又一轮风波，引出武林新的传奇。

9
珍珑棋局

话说当年逍遥派大弟子，外号聋哑老人，江湖人称聪辩先生的苏星河，在擂鼓山摆下珍珑棋局，诚邀天下年轻才俊和高手前去破解。此珍珑棋局乃是逍遥派掌门无崖子，也就是苏星河的师傅设计的，希望世间有人能破解此局。

这珍珑棋局是劫中有劫，长生伴共活，或反扑，或收气，花五聚六，复杂无比。就连聪辩先生这样的顶尖棋手，苦心钻研三十年，也未能参透其中的玄机。

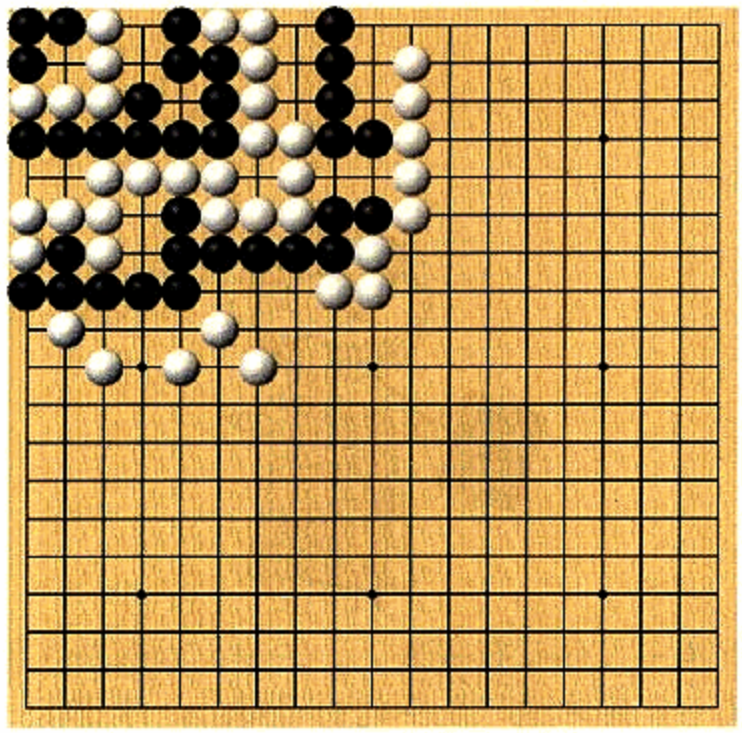
先后有段誉、慕容复和段延庆等，黑白两道的围棋高手上阵，都未能解开此局。不但没有解开珍珑，有些高手还险些被珍珑困住，导致走火入魔，竟然要拔剑自刎或举戈自戕。

一个名不见经传的年轻小和尚虚竹，出于善心，不忍看到有人为珍珑而丢了性命，出来搅局。结果，误打误撞地杀自己一大片棋子，却意外地打开了一片广阔的新天地，反而由此破解了这精心设计的珍珑棋局。

这是金庸先生著名武侠小说《天龙八部》中的一个经典故事。金庸先生厉害之处就在于，他写的故事既有个性鲜明的人物，层叠起伏的情节，虚虚实实的故事，还让我们在阅读中领悟到人生的哲理，从而在心灵上得到某些有价值的东西。故事里的这个珍珑棋局就像一心灵的透镜，透过珍珑棋局可以清清楚楚地看到武林江湖中的众生相。

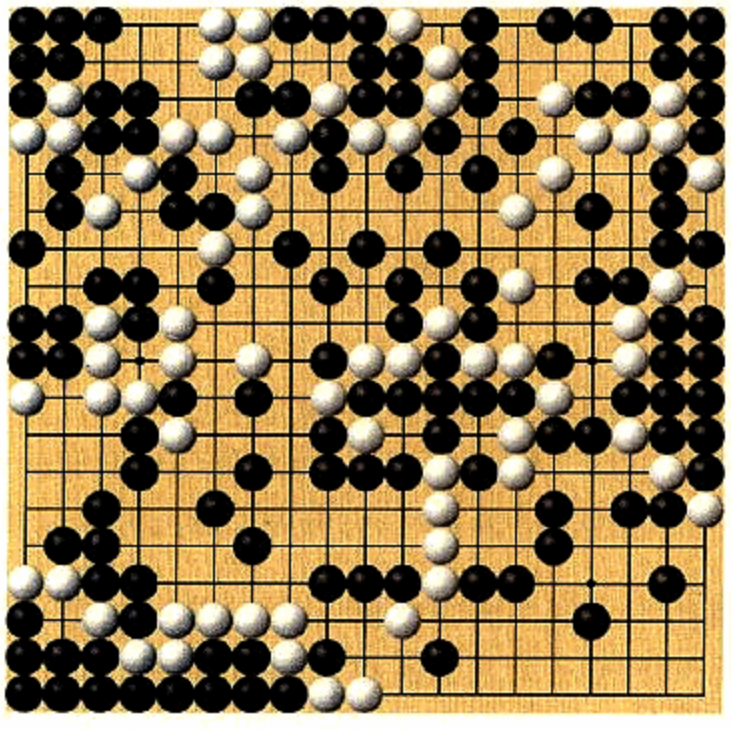
据说，有一位围棋高手依据《天龙八部》里的描写，编了个珍珑棋局的示意图出来。多年之后，金庸先生竟将这位大侠的珍珑棋局发给一位朋友，勉励他突破自我，这可是一件非常有趣的事情。而我们也假装卖一回关子，偷来这罕有的珍珑棋谱，藏在了“禅棋传说”的网页里。

网页的最后十几行程序，就是隐藏珍珑棋谱的数据和代码。当打开go6.htm网页之后，只要按键盘上的“1”，就可以打开金大侠的“珍珑棋局示意图”：



嗯，这是一道围棋的死活题。似乎并不太难，一般的围棋爱好者稍加思索，相信都能顺利破解。

如果上面的太简单，再来个复杂点的。按键盘上的“2”，将出现一个古代棋经中的大型死活题：



不是讲javaScript吗？怎么真的讲起围棋来了？

是啊，人生如棋，编程也是如棋啊！

我们的每一个软件产品，每一个项目，不都和下棋一样？

我们总得先规划和布局吧。不管是采用严谨的软件工程方法，还是边做边改的敏捷开发，大致的方向，总体的策略总得有吧。就如下棋时选择“宇宙流”还是“中国流”布局，采用以取势为主的策略还是以实地为主的策略。开发项目进行到一定程度，一定会遇到各种未曾预料到的问题，比如需求的变更，计划的调整，人员的变动，等等。就像围棋中盘时不得不面对的复杂的现实局面，我们需要考量得失，才能决定如何取舍。项目的收尾工作也就像围棋的收官，马虎不得，稍有不慎，说不定又会在劫争中反复厮杀。

我们总在不断地学习编程的新知识和新技术，昨天是算法研究，今天是数据结构。刚熟悉结构化编程，又流行面向对象。唉，这思想、概念和技术，不断地更新换代，学习的难度也越来越大。

还好，我们的前辈们经过多年的摸爬滚打，也给我们留下了某些经验之谈。比如，设计模式就是许多人总结下来的精华。这就像围棋里面的定式，按照定式行棋可以避免走很多弯路，少些风险和挫折。下围棋的人大都会些常用定式，新手往往比较严格的按定式行棋，而高手却通过领悟定式的棋理达到活学活用的境界，大师眼里或许就没有定式了。

编程不也一样，对设计模式的领悟程度，也一定程度上反映了编程境界的高低。说实在，很多经验丰富的老程序员，在听到“设计模式”这一名词时，还真不知设计模式是啥东西？等搞懂这个名词的含义时，才会心一笑，原来自己天天在用啊！所以啊，我们别去学“名词”，而要学思想和方法，否则就被“设计模式”给困住了。

没有一成不变的定式，也没有一成不变的设计模式。当年的围棋青年吴清源，就是在大胆打破日本棋界长期保守的布局和定式之后，开创了新式布局的广阔天地，也走出了小雪崩、大雪崩这样惊世骇俗的高难定式。吴清源之所以能成为一代围棋大师，并不在于他打遍日本棋坛无敌手的传奇，而在于由此开创的围棋革命新时代。但愿将来的某一天，我们的编程界也能出现一个吴清源啊！

所以啊，那些技术上的东西总在不断变化，昨天还是那片天空，今天又已经斗转星移。沧海桑田之后，能沉淀下来的往往只是思想的精华。这些思想的精华才是我们应追求的编程之道。领悟到一种技术的思想精华，我们就会发现，所谓的技术不过是一种表象。技术之争不过是两个人用身体打架，即使打赢了，对方也不见得服你。高手间的讨论从来都是点到为止，通过思想的交流，大家都有收获，境界不一样嘛。只有思想上的撞击，才有可能产生灵感的火花。

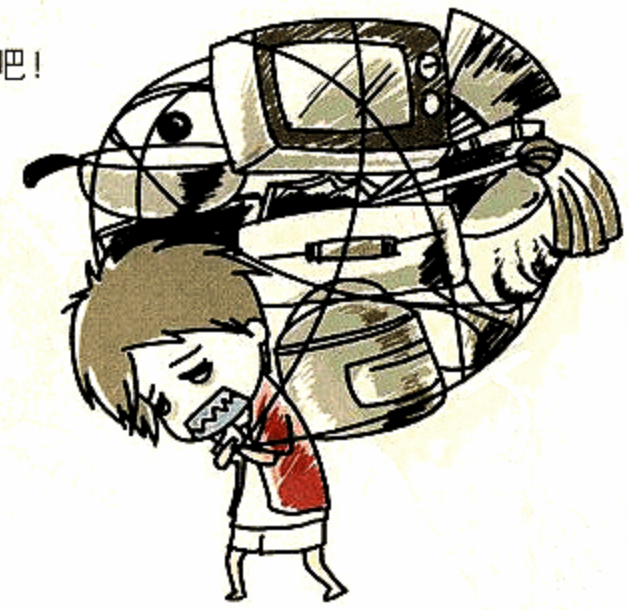
然而，学习是永无止境的。一旦学有所成，我们就会得到某些东西，领悟到一些道理。不过，这仅仅是茫茫学海里拾得的一只贝壳。虽然得来辛苦，但不应看得太重。如果因此而沾沾自喜，或者得意忘形，甚至自以为是，瞧不起他人，这小小的贝壳也许就变成了沉重的负担。在探索真知的旅途中，过于看重已得到的东西，就会让前进的脚步变得越来越蹒跚，甚至被活活累死。这就是为什么有些人学到了一定的程度就无法再提高了？这就是原因啊。

都说“人最难战胜的是自己”，这就是珍珑棋局所蕴含的人生哲理！

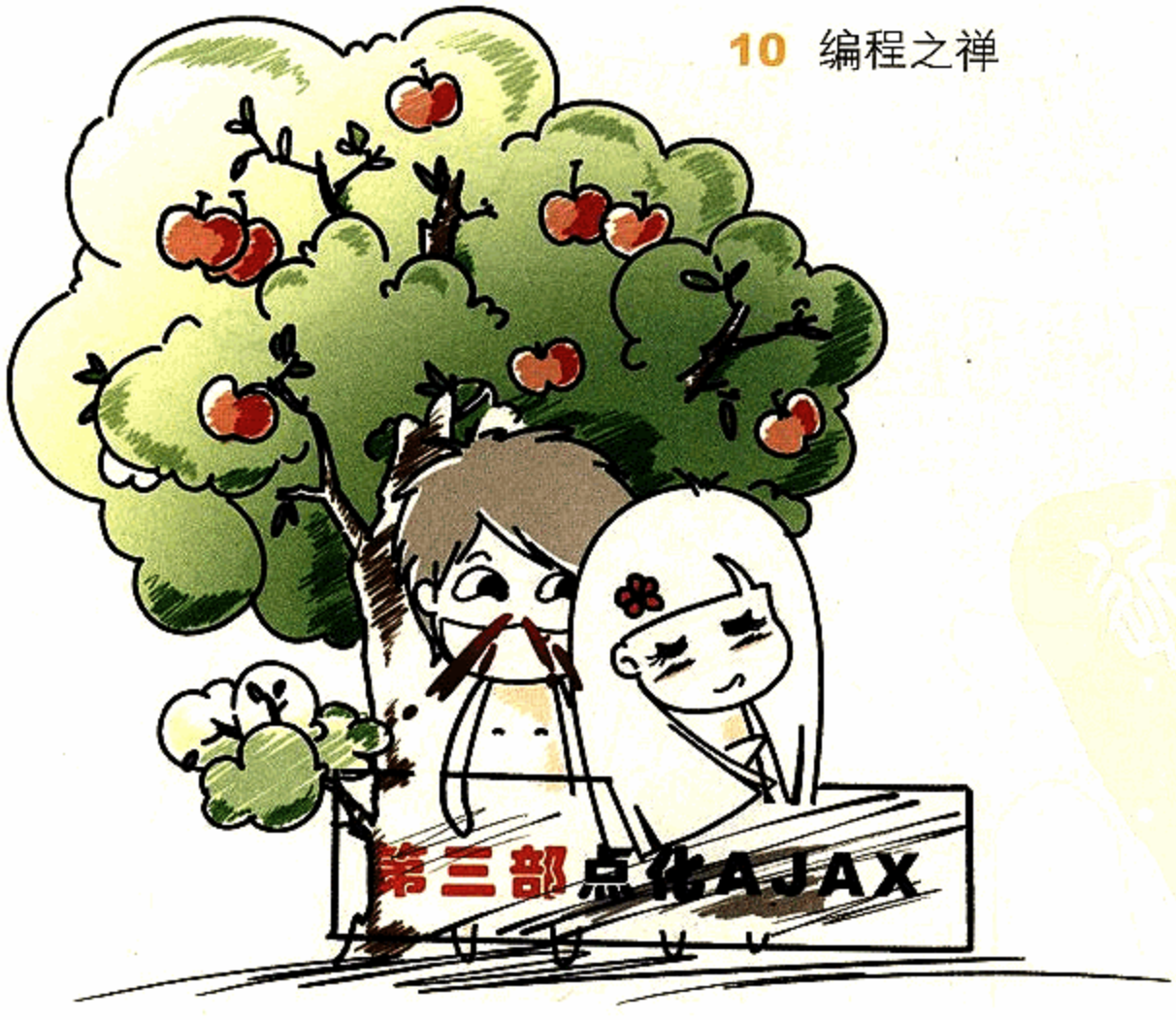
那么，在编程世界的珍珑里，我们能否战胜自己呢？从逻辑上说，没有人能战胜自己，因为这是一个悖论。所谓的战胜自己，也就是看看能否看轻所得的东西，能否跳出技术的纷争，能否放弃表面的虚名，能否放下狭隘的自我。

希望通过“禅棋传说”这一小程序的演练，以及“珍珑棋局”的破解，不但能学到JavaScript的基本技巧，也能让我们领悟到一些编程的真谛。

放下自我，拥有世界，做一个快乐的程序员吧！



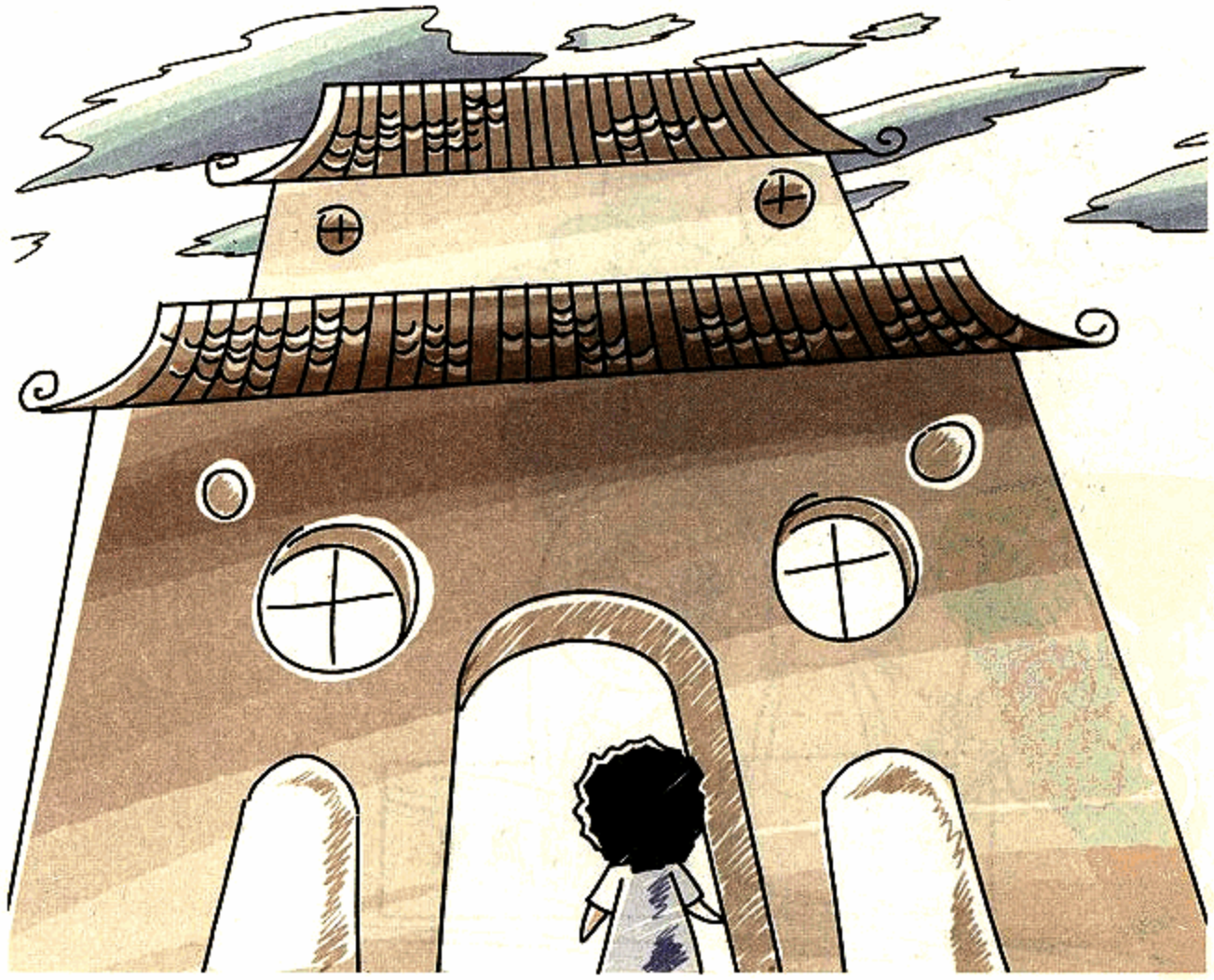
1 叩问AJAX 116
2 直捣AJAX 119
3 ASP.NET AJAX简介 124
4 AJAX与WebService 129
5 AJAX之双手互搏 139
6 域名的鸿沟 145
7 跨越域名的时空 152
8 特使协议 159
9 单点共享登录模型 163
10 编程之禅 175



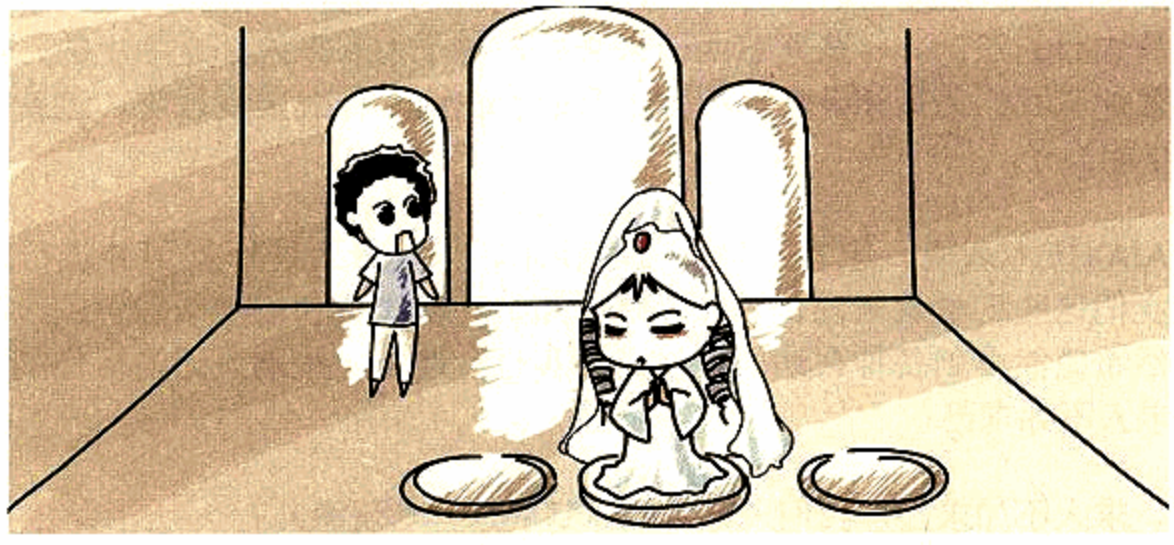
1
叩问AJAX

有一位AJAX程序员遇了到一个非常奇怪的问题。尽管经过几个昼夜的煎熬，依然没有能找出问题的原因。他在网上查了几天，也没有找到有关此类问题的网页，又咨询过许多资深的技术专家和顾问，也大都泛泛而谈，没人能给他具体的明示。

最终，由于这个问题实在太奇怪，程序员相信自己遇到了还没有人遇到过的新问题。他听说山里的观音庙很灵验，而这样的问题恐怕也只有求大慈大悲的观音菩萨才能解决。于是，他放下手里的事情，打点行装，前往山上的观音庙烧香。



程序员来到观音庙之后，庙里并没有什么人，只有一位年轻美貌的女子在向观音像参拜。细心的程序员突然发现这位女子的身形和着装，竟然与莲花座上的观音像非常地相似。于是，他就小心翼翼地走上前去。仔细一看，那位年轻美女正是观音大士！



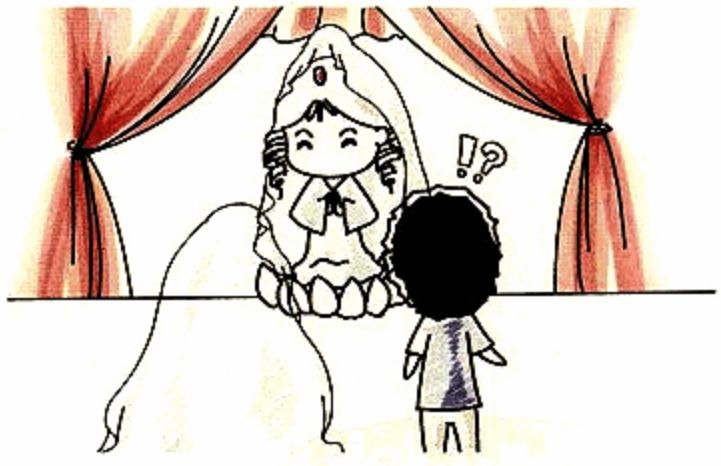
程序员惊奇地问：你是观音大士吗？

美女回答：正是！

程序员更奇怪了，又问：你怎么也向观音求救？

美女回答：求人不如求己！

程序员恍然大悟！



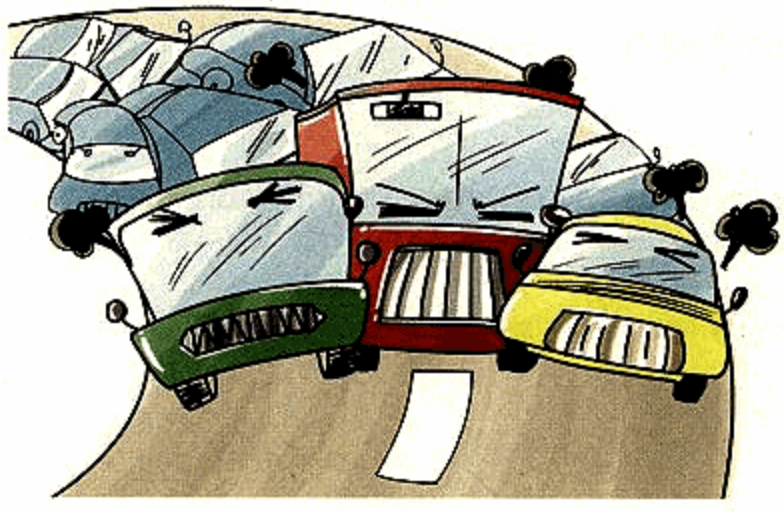
后来，这位程序员在下山的路上仔细反思了自己的程序，也逐渐理清了纷乱的思路。最终，他找到了解决新问题的新办法。

IT行业恐怕是知识更迭最快的行业。作为一名程序员，我们无时无刻不在学习新的知识和淘汰旧的知识。一些新东西出来之后，对于大家来说都是陌生的，很少有现成的经验教训供我们选择和参考。但我们又必须在这个行业里混饭吃，这就需要考验我们的自学能力了。

对于AJAX技术来说，原本不是什么新技术，只是最近两年规范并流行起来。当然，对于我们这些初学者来说，仍然算是新知识。在学习和应用新知识的过程中遇到问题，当然希望能得到各种帮助。但是，确实也会遇到一些别人无法帮你的问题。怎么办呢？求人不如求己！

好了，求人不如求己！我们来看看AJAX到底是些啥玩意儿？

2
直捣AJAX



AJAX是Asynchronous JavaScript and XML的缩写，翻译成中文就是“异步的JavaScript和XML”。其中只有三个关键字：“Asynchronous”，“JavaScript”，“XML”。“and”是个连词，不算关键字。估计发明这个名词的人一定是意大利阿贾克斯足球队的铁杆球迷，所以才把“and”整成“AND”，正好凑成AJAX。也好，这个词符合英文单词构词原则，也符合发音原则。

AJAX并非什么新技术，那只是马后炮。因为在“AJAX”这个名词出现之前，网页开发者早就在使用这些技术，即网页数据的局部更新和交互！

我们来现学现卖，先准备一个局部数据的来源服务，以及一个取名为ServerTime.aspx的动态网页文件：

ServerTime.aspx

```
<%= DateTime.Now %>
```

这个文件只有一行代码，输出服务器的当前时间，每次请求这个页面都会得到变化的服务器时间信息。把ServerTime.aspx文件放在可以运行ASP.NET页面的网站或目录下，作为更新局部网页信息的数据源。我们将让一个页面来请求这个动态网页，让页面上的时间不断变化。

奇怪，这个页面怎么没有像<html> <head> <body>等那样的标签？没错，确实没有！因为它并不输出HTML文档，而只输出数据。我们不要被Visual Studio的那些模板和向导迷惑了，她自动生成一些标签告诉你往那边走，其实也是可以往这边走的。所

有ASPX文件不过是向客户端返回一个信息流而已，既可以返回HTML文档流给浏览器呈现，也可以返回图片、字符串和其他形式的数据。而这里的ServerTime.aspx只返回服务器时间的字符串。

然后，我们要制作一个标准网页AJAX1.htm，让里面的部分内容从ServerTimer.aspx更新。AJAX1.htm是个静态网页，代码如下：

AJAX1.htm

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

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>服务器时间</title>
</head>
<body>
  <span id="clock"></span>
  <script type="text/javascript">
    //对于没有XMLHttpRequest对象的浏览器，需要构造一个：
    if(!window.XMLHttpRequest)
      window.XMLHttpRequest = function()
      {
        return new ActiveXObject("Microsoft.XMLHTTP");
      };

    function UpdateClock() //动态刷新时钟函数
    {
      //建立一个XMLHttpRequest对象去请求ServerTime.aspx页面：
      var aRequest = new XMLHttpRequest();
      aRequest.open("POST", "ServerTime.aspx", false);  //同步调用
      aRequest.send("");

      //根据返回的结果，去更新网页局部元素的信息：
      document.getElementById("clock").innerHTML = aRequest.responseText;
    };

    setInterval(UpdateClock, 1000); //每隔一秒请调用一次UpdateClock函数
  </script>
</body>
</html>
```

其中，我们在页面上定义了一个id为clock的标签，然后是一段JavaScript代码来操纵一个XMLHttpRequest对象来获取服务器时间，并用获取的时间更新标签的innerText属性。我们用了setInterval函数，让这个更新每隔一秒发生一次。

当我们用浏览器打开AJAX1.htm页面后，就可以看到一个不断跳动的时间。请注意，这里跳动的时间并非客户端时间，而是服务器时间。它是通过JavaScript操纵XMLHttpRequest对象去服务器请求数据，并更新网页局部内容的。整个页面并没有重新刷新，只有局部的数据在刷新。

这就是AJAX？没错，这基本就是AJAX，有了JavaScript，有了XML，哦，不对，是XMLHttpRequest，而且还没有Asynchronous。

上面的例子发出的XMLHttpRequest是同步的。同步的XMLHttpRequest请求发出之后，JavaScript程序需等待请求完成才会继续运行。由于JavaScript是单线程的，因此在等待请求过程中，浏览器会停止响应。如果等待时间很长，浏览器就像死掉了一样，一动不动，这可不是太好。

然而，XMLHttpRequest天生就具备异步处理请求的能力。所谓异步处理，就是当前的线程发出请求后调用会立即返回，当前线程就可以做其他事情了。一旦请求的数据处理完毕，将主动通知当前线程再来处理。这样的话，浏览器就不会死掉了。

我们需要再改造一下JavaScript代码，让它按异步方式更新局部程序。于是我们将前面的网页改写成AJAX2.htm文件：

AJAX2.htm

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
  <title>服务器时间</title>
</head>
<body>
  <span id=" clock" ></span>
  <script type=" text/javascript" >
    //对于没有XMLHttpRequest对象的浏览器，需要构造一个：
    if(!window.XMLHttpRequest)
      window.XMLHttpRequest = function()
      {
        return new ActiveXObject("Microsoft.XMLHTTP");
      };

    function AsynRequest()  //异步请求函数
    {
      //建立一个XMLHttpRequest对象去请求ServerTime.aspx页面：
      var aRequest = new XMLHttpRequest();
      aRequest.open("POST", "ServerTime.aspx", true);   //异步调用
```

```
        aRequest.onreadystatechange = function()   //请求完成将收到此事件
        {
          if(aRequest.readyState === 4)  //状态为4表示请求成功
            UpdateClock(aRequest.responseText); //调用UpdateClock
        };
        aRequest.send( " " );
      };

      function UpdateClock(aTime)    //更新时间函数
      {
        document.getElementById( "clock" ).innerHTML = aTime;
      };

      setInterval(AsynRequest, 1000);    //每隔一秒发出一次异步请求
    </script>
</body>
</html>
```

其中，给XMLHttpRequest对象的open方法最后一个参数设定为true，即可启动异步请求模式。同时，我们必须设置XMLHttpRequest对象的onreadystatechange事件，因为异步请求的状态变化时，这个事件将被触发。而我们需要在这个事件中判断请求是否完成，如果完成，就调用更新局部数据的函数UpdateClock，参数当然是请求的结果。

这样，我们发出XMLHttpRequest对象的请求后，线程将立即返回，浏览器就可以响应其他处理了。而当异步请求完成时，UpdateClock函数将被调用，这时才去更新网页的局部数据。这就实现了“Asynchronous”，也就是所谓的异步。

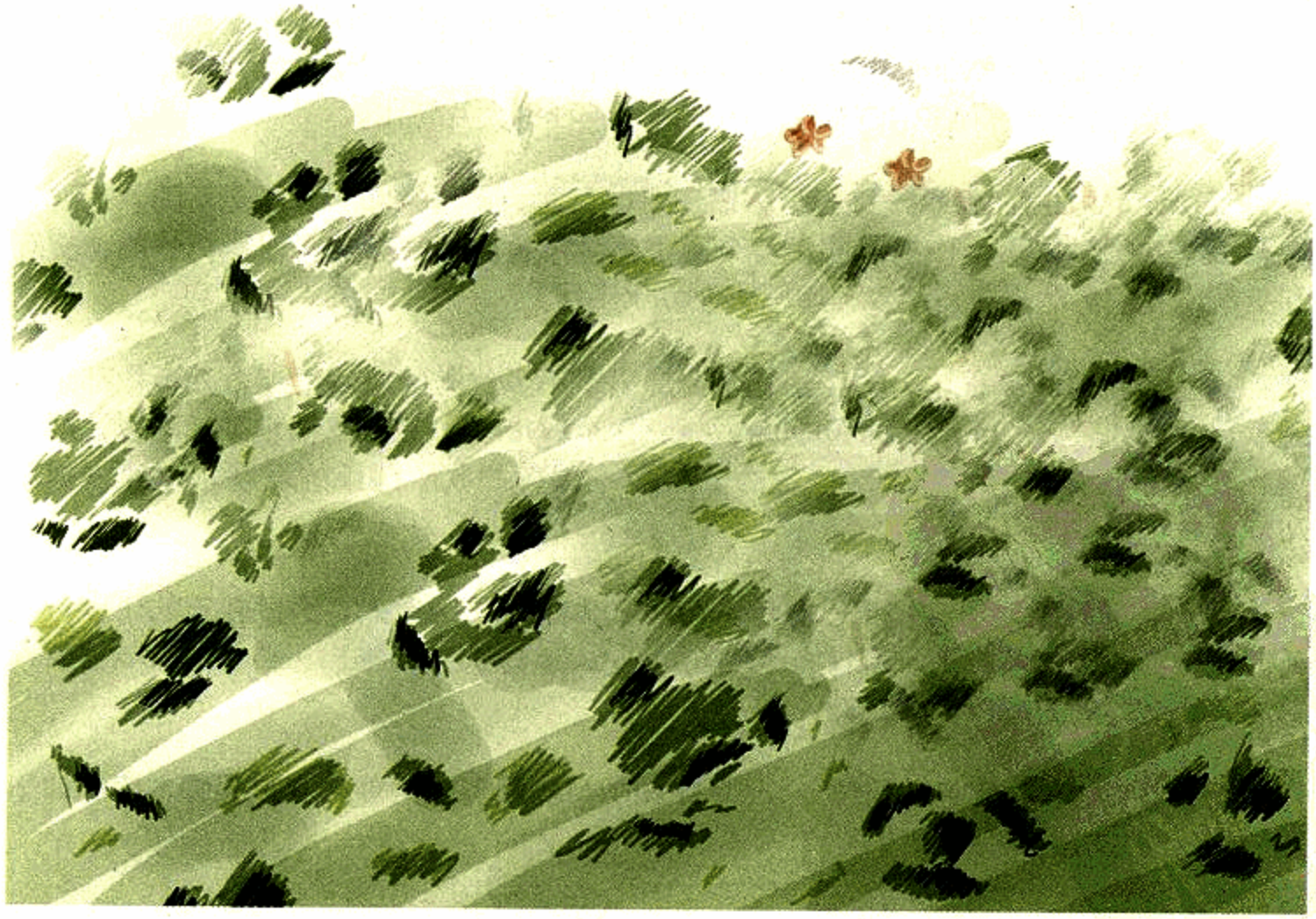
AJAX三个关键词中的“Asynchronous”和“JavaScript”都还说得过去，但“XML”就很牵强。我们并没有用到XML！唯一有点沾边的XMLHttpRequest，也只是用了其responseText属性，并没有用到那个货真价实的responseXML属性。

为什么不用这个XML呢？这就得翻翻旧账了。早些年，微软为了给浏览器增加处理XML的能力，在IE 5.0中引入了XML数据岛以及一个叫XMLHttpRequest的ActiveX对象。XML数据岛可以直接用HTML标签加载XML文档，而XMLHttpRequest的诞生，就是为了能让JavaScript可以动态加载XML文档。

有趣的是，IE浏览器提供的XML数据处理的能力几乎无人问津，反倒是XMLHttpRequest可以请求服务器数据的能力倍受人们的青睐。这仿佛一条客户端通向服务器的小山路，却是直接通向服务器的捷径。通过这一捷径，客户端和服务器可以更好地交互，实现许多新的功能。

微软本来打算用这条小路来作为运送XML数据的货运通道，但人们更愿意把它当成便捷的人行道。于是，走这条小路的人越来越多，以至其他没有开辟这一捷径的浏览器厂商，干脆提供一个内置的XMLHttpRequest对象，直接将小路变成高速公路。随后，IE 7也开通了XMLHttpRequest的直达专列，而W3C组织也赶紧考虑将XMLHttpRequest纳入标准。

恐怕微软自己也没想到，当初的XMLHttpRequest竟然会开创出AJAX这片广阔的天地，这可真是：有心栽花花不开，无心插柳柳成荫！



3
ASP.NET AJAX简介

当ASP.NET面世的时候，其独特的页面模型，让我们能用组件形式开发网页程序。而且，ASP.NET提供的WebForm和ViewState机制，硬是在无状态网页世界里模拟出有状态的编程模型。这使得ASP.NET迅速占领了网页编程的大片江山。

不过，ASP.NET的网页编程模型依旧还是以页面为单位来和服务器交互的。也就是说，每一次触发网页操作的时候，整个网页都会被重新加载。要实现网页的局部交互和更新，我们仍然需要写不少程序来实现AJAX应用。

为了简化AJAX编程，一些组织和个人退出了不少优秀的AJAX框架，大多数都是开源项目，比如著名的prototype和JQuery，还有AJAX.NET等。使用这些框架，可以让AJAX编程变得轻松许多。于是，这些优秀的AJAX框架得到欣欣向荣的发展，迅速占据了AJAX大片市场。

然而，微软绝对不会将自己开辟的AJAX江山拱手让给他人。经过几年的研发，在不断试验和改进的基础上，微软终于随Visual Studio 2008一起推出了自己的AJAX框架，并将其正式命名为ASP.NET AJAX。

ASP.NET AJAX框架包括两部分，即ASP.NET AJAX服务器组件和客户端脚本库。如下图所示：

在服务器端，ASP.NET AJAX框架提供了一个ScriptManager服务器控件，它也是ASP.NET AJAX程序必须加入的服务器控件。引入ScriptManager的目的是为了同一管理客户端的JavaScript脚本及外部引用文件。微软建议我们别再自己往客户端输出脚本或引用脚本文件，而应该通过ScriptManager来输出和引用来管理这些JavaScript脚本。

此外，ASP.NET AJAX还专门提供了System.Runtime.Serialization.Json命名空间及相关的类，以支持将对象序列化为JSON格式。这种序列化功能几乎可以把.NET对象任意对象序列化为JSON格式，也可以将JSON字符串反序列化成.NET对象。这造就了JavaScript与.NET程序直接交换数据和对象的强大功能。

服务器端在对象的JSON序列化功能之上，扩展了一种新的Web Service调用方式。这种新的Web Service调用方式允许客户端和服务器之间通过JSON格式进行远程调用，而不是XML格式。这些扩展使得客户端JavaScript可以直接调用服务器端的Web Service，通过JSON来高效地交换数据，而无需按传统的SOAP及XML数据来交互。

JSON本身是JavaScript原生的数据格式，而且对数据的表示比XML精简许多。JSON格式的Web Service调用，减少了数据格式的转换，也减少了传递数据的尺寸，比传统的SOAP调用效率高得多。这种技术已在AJAX开发中得到广泛应用。

为了让已经熟悉WebForm的开发人员也能用上AJAX技术，微软还煞费苦心地开发了一个UpdatePanel控件。使用这个UpdatePanel控件之后，只要将服务器控件放入一个神奇的UpdatePanel，这些控件就具有了局部异步更新的能力。

使用了UpdatePanel的客户端程序，可以自动识别要局部更新的元素，并以一种特

殊形式将局部更新的请求自动回发给当前页面。而当前页面也可以自动识别回发的这些局部AJAX请求，并针对这些特殊请求执行局部的网页呈现，而不是整个网页。而客户端收到局部信息后，再自动更新UpdatePanel中的那些局部内容。

这样，UpdatePanel就在传统的WebForm页面上实现了AJAX应用，而无需开发者编写额外的AJAX代码。真的很神奇！微软的AJAX开发人员可谓用心良苦啊。

然而，UpdatePanel在每一次处理局部更新和呈现的过程中，还是建立了一个完整的页面对象。因为，不管是整页回发还是局部的异步回发，页面依旧会经历整个ASP.NET页面生命周期，只是只有在识别到UpdatePanel异步回发时，才把局部内容返回给客户端。由此可见，UpdatePanel在服务器端做了大量的无用功，这绝对不是高效的AJAX模型。

因此，严格来说，基于UpdatePanel的ASP.NET AJAX应用只能算是准AJAX！目前，在微软的ASP.NET AJAX框架中，只有基于WebService调用的ASP.NET AJAX应用，才能算真正的AJAX应用。当然，我们也没必要去区分什么真正不真正的，关键是看是否适合自己的应用。

ASP.NET AJAX的客户端也叫做Microsoft AJAX Library，目前由三个.js文件构成，分别是MicrosoftAjax.js，MicrosoftAjaxTimer.js，MicrosoftAjaxWebForms.js。我们可以从微软的网站上下载到Microsoft AJAX Library的全部源代码，也可以从Visual Studio 2008抽取ASP.NET AJAX的内嵌资源来找到这些代码。

这三个.js文件一般会以两种形式出现：一个是给运行时用的紧凑文件，一个是给调试用的阅读版文件。紧凑文件是在保证JavaScript代码逻辑和对外接口不变的情况下，缩短标识符，去除注释和空白，尽量压缩文件尺寸而形成的JavaScript代码。这样做的目的是为了提高加载JavaScript文件的速度，加快JavaScript解析和执行代码的效率。而调试文件是人工排版的，并配有一定的注释，是用于帮助程序员阅读和理解的JavaScript代码文件。

调试版本的JavaScript文件尺寸要比运行版本大得多。下面就是Microsoft AJAX Library中三个文件各种版本的大小对比：

MicrosoftAjax.debug.js	281KB
MicrosoftAjax.js	89KB
MicrosoftAjaxWebForms.debug.js	73KB
MicrosoftAjaxWebForms.js	31KB
MicrosoftAjaxTimer.debug.js	6KB
MicrosoftAjaxTimer.js	3KB

对Microsoft AJAX Library的学习，除了查看MSDN的相关文档外，更深入的学习方法恐怕就是阅读Microsoft AJAX Library的源代码了。

Microsoft AJAX Library在客户端建立了一个庞大的编程环境，扩展了JavaScript原生对象的功能，模拟了若干命名空间，也模拟了委托等概念，还封装了大量的类和控件。这使得Microsoft AJAX Library看起来与.NET Framework的类库很像。

与其说Microsoft AJAX Library是用JavaScript模拟了一个类似.NET Framework那样的编程环境，还不如说它就是.NET Framework的一部分，因为在MSDN最新的文档中Microsoft AJAX Library的参考资料已经归入到.NET Framework 3.5参考文档的栏目中。

微软之所以要模拟出这个庞大的编程环境，目的就是要让使用ASP.NET AJAX的程序员，继续以.NET Framework的思维来编写客户端的JavaScript代码。微软不但把JavaScript的代码写得非常像C#语言，而且命名空间和封装的类库都尽量保持与.Net Framework一致。这样，.NET平台下的程序员就继续在微软铺就的康庄大道上前进，或许在不经意的某一天，我们会突然发现已经被微软带到RIA的世界了。总被微软牵着走，这就是我们这些.NET程序员的命运啊！

继续生活吧。我们来看看Microsoft AJAX Library里面有些啥玩意儿。

全局命名空间

扩展了JavaScript原生的类型和对象，包括Array、Boolean、Error、Number、Object和String，从最基础的地方起增加了.NET开发人员熟悉的成员。

Sys 命名空间

有点像.NET的System命名空间哦。它表示 Microsoft AJAX Library 的根命名空间，包含了所有重要的类和基类。其中，还模拟了一个客户端的Application模型，也构造了生命周期的概念。

Sys.Net 命名空间

和.NET的System.Net命名空间一样，提供网络通信服务等。AJAX核心的XMLHttpRequest被封装成WebRequest，与.NET Framework类库中的WebRequest相似吧？

Sys.Serialization 命名空间

目前，这个命名空间只包含对JSON的序列化和反序列化支持。与.NET Framework的System.Runtime.Serialization.Json命名空间提供的，服务器端JSON的序列化和反序列

化功能形成对应。

Sys.Services 命名空间

提供对 ASP.NET 身份验证服务、配置文件服务及其他应用程序服务的客户端脚本访问功能。

Sys.UI 命名空间

这是客户端控件的基础，提供将DOM元素封装成客户端控件的能力。经过这样的封装，.NET程序员又可以用熟悉的控件模型来操控DOM对象。

Sys.WebForms 命名空间

这个命名空间就是为UpdatePanel控件设计的，包含与 Microsoft AJAX Library 中的部分页呈现相关的类型。

这么看来，Microsoft AJAX Library也并不是太复杂嘛，除了架子搭得很大之外，也没什么了不起。微软之所以要搭这么大的一个框架，显然不仅仅是为了与jquery等精巧的框架去争夺AJAX市场，而是为了构建一个庞大的AJAX大厦。也正是因为Microsoft AJAX Library打下的宽阔而深厚的地基，才可能支撑起ASP.NET AJAX Control Toolkit那样丰富的AJAX控件库。

因此，我们去阅读Microsoft AJAX Library的JavaScript源代码时，就不会再为那些复杂的AJAX封装感到奇怪了。而我们了解了微软的深意之后，即不会被表面的封装迷惑了我们的思路，也能更加理解ASP.NET AJAX后面的运行机理。

尽管微软一直在牵着我们走，但我们应该要知道为什么往那个方向走。

4

AJAX与WebService

我们先来把前面那个服务器时间的例子用ASP.NET AJAX来改写一下。我们原来的后台服务是由ServerTime.aspx页面提供的，现在要将其改写成正规的Web Service形式。于是，我们写了一个ServerTime.asmx的Web Service如下：

ServerTime.asmx

```
<%@ WebService Language=" C#" Class=" ServerTime" %>

using System;
using System.Web.Services;
using System.Web.Script.Services;

[ScriptService] //允许AJAX调用此WebService
public class ServerTime : WebService
{
    [WebMethod]
    public string GetServerTime()
    {
        return DateTime.Now.ToString();
    }

}
```

啊哈，这比前面那个只有一行代码的ASPX文件复杂多了。没办法，Web Service就是这样形式的。其中，为了能让JavaScript以AJAX方式调用这个Web Service，我们必须给这个Web Service打上一个[ScriptService]标记。当然，其中的方法也得打上[WebMethod]标记。

客户端应该怎样来调用这个WebService呢？

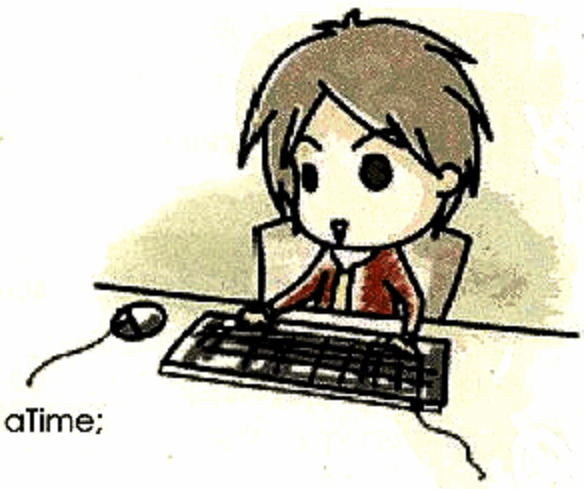
传统的WebService是用SOAP协议来调用的，即先用XML封装调用参数，然后用HTTP协议的POST方式来发送数据，服务器把返回的结果也封装成XML回传给客户端。好在，我们这个ServerTime服务的GetServerTime方法并不需要参数，返回的结果也只是

一个字符串而已。我们何不试试直接用HTTP的 POST方式调一下？看看会有啥结果。

说干就干，我们把原来的AJAX2.htm改写为AJAX3.htm：

AJAX3.htm

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
   <title>服务器时间</title>
</head>
<body>
   <span id=" clock" ></span>
   <script type=" text/javascript" >
      if(!window.XMLHttpRequest)
         window.XMLHttpRequest = function()
      {
         return new ActiveXObject( "Microsoft.XMLHTTP" );
      };

function AsynRequest()
{
   var aRequest = new XMLHttpRequest();
   aRequest.open( "POST" , "ServerTime.asmx/GetServerTime" , true); //请求WebService
   aRequest.onreadystatechange = function()
   {
      if(aRequest.readyState === 4)
         UpdateClock(aRequest.responseText);
   };
   aRequest.send( "" );
};

function UpdateClock(aTime)
{
   document.getElementById( "clock" ).innerHTML = aTime;
};
```



```
    setInterval(AsynRequest, 1000);
  </script>
</body>
</html>
```

这个AJAX3.htm中除了aRequest.open那一句以外，其他的与AJAX2.htm 完全相同。这修改的一句只是把请求的URL换成标准的WebService调用模式，这也是SOAP调用的URL形式，即，服务名后跟方法名的形式。

马上运行一下看看。哇，竟然和AJAX2.htm表现得一模一样！页面上也出现了一个跳动的时钟！

别高兴太早，这只是巧合。

原来，用POST发出的HTTP请求是一个SOAP形式的Web Service调用，虽然没有参数而无需封装传给服务器的XML数据，但服务器返回的确实是一个XML格式的数据。检查一下responseText就知道，里面的内容实际是类似下面的形式：

```
<string xmlns=" http://tempuri.org/" >2008-4-14 21:28:10</string>
```

只因为浏览器并不识别<string>标签而将其忽略了。<string>标签是SOAP中用来表示字符串类型的。如果返回的不是ServerTime，而是非常复杂的对象呢，我们就得去解析和读取responseXML的各个节点信息了。

不是说ASP.NET AJAX 的服务器端可以返回JSON形式的结果吗？

是的，不过需要客户端在发出HTTP POST请求之前，指明需要JSON的结果。这是通过给XMLHttpResquest对象设置Content-Type = application/json这样的头部信息来传达的。当标记为[ScriptService]的WebService识别到这样的头部信息是，它将返回JOSN形式的结果。于是，我们又将AJAX3.htm改写为AJAX4.htm：

AJAX4.htm:

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
  <title>服务器时间</title>
</head>
<body>
  <span id=" clock" ></span>
  <script type=" text/javascript" >
```

```
if(!window.XMLHttpRequest)
   window.XMLHttpRequest = function()
   {
      return new ActiveXObject("Microsoft.XMLHTTP");
   };

function AsynRequest()
{
   var aRequest = new XMLHttpRequest();
   aRequest.open("POST", "ServerTime.asmx/GetServerTime", true);

   //在请求头部说明本次调用是JSON形式的:
   aRequest.setRequestHeader("Content-Type", "application/json; charset=utf-8");

   aRequest.onreadystatechange = function()
   {
      if(aRequest.readyState === 4)
         UpdateClock(aRequest.responseText);
   };
   aRequest.send("");
};

function UpdateClock(aTime)
{
   document.getElementById("clock").innerHTML = aTime;
   };

   setInterval(AsynRequest, 1000);
  </script>
</body>
</html>
```

其中的aRequest.setRequestHeader一句，就是添加了头部信息，告诉服务器端以JSON形式进行交互。这个AJAX4.htm运行后，我们将看到以下形式的跳动信息：

```
{"d":"2008-4-14 22:33:26"}
```

明眼人一看就知道这是一个JSON对象字符串，这个对象有一个属性d，属性d的值是"2008-4-14 22:33:26"。

直接把这个JSON格式字符串显示出来肯定不是我们需要的结果，而是要将其序列化成一个对象，然后获取这个对象的d属性值。这就需要用到JavaScript的eval函数了。于是，我们又有了AJAX5.htm页面：

AJAX5.htm

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
  <title>服务器时间</title>
</head>
<body>
  <span id=" clock" ></span>
  <script type=" text/javascript" >
    if(!window.XMLHttpRequest)
      window.XMLHttpRequest = function()
      {
        return new ActiveXObject( "Microsoft.XMLHTTP" );
      };
    function AsynRequest()
    {
      var aRequest = new XMLHttpRequest();
      aRequest.open( "POST" , "ServerTime.asmx/GetServerTime" , true);
      aRequest.setRequestHeader( "Content-Type" , "application/json; charset=utf-8" );
      aRequest.onreadystatechange = function()
      {
        if(aRequest.readyState === 4)
        {
          var result = eval( "( "+aRequest.responseText+" )" );  //将结果反序列化
          UpdateClock(result.d); //取结果对象的d属性调用UpdateClock函数
        }
      };
      aRequest.send( "" );
    };

    function UpdateClock(aTime)
    {
      document.getElementById( "clock" ).innerHTML = aTime;
    };

    setInterval(AsynRequest, 1000);
  </script>
</body>
</html>
```

其中，我们在调用**eval**函数的时候给**JSON**字符串加了个小括号，这是为什么呢？原来，这个**JSON**对象的属性名是用字符串”d”来表示的，而不是标识符**d**，这可能会让**eval**解释引擎以为大括号中是程序语句而非**JSON**。在外面加上小括号之后就明确表示这是一个表达式，**eval**就不会误解了。

现在，运行AJAX5.htm又将看到正常的时间跳动了。

通过以上的实践，我们基本搞懂了如何在客户端用JavaScript调用Web Service的方法。不过，我们也为此编写了太多的代码。如果每调一个WebService的方法就要写这么一堆代码，那可成了麻烦事了。幸运的是，ASP.NET AJAX早已把这一切给我们封装好了。

如果用浏览器打开/ServerTime.asmx/js，将会看到一堆JavaScript代码或者下载一个相关代码的.js文件。也就是说，通过给Web Service链接增加/js后缀，就可以得到其代理类的JavaScript代码。为了方便阅读，用/jsdebug后缀来获取其调试版本看看：

ServerTime.asmx/js结果

```
var ServerTime=function() {
ServerTime.initializeBase(this);
this._timeout = 0;
this._userContext = null;
this._succeeded = null;
this._failed = null;
}
ServerTime.prototype={
_get_path:function() {
 var p = this.get_path();
 if (p) return p;
 else return ServerTime._staticInstance.get_path();},
GetServerTime:function(succeededCallback, failedCallback, userContext) {
/// <param name=" succeededCallback" type=" Function" optional=" true"
mayBeNull=" true" ></param>
/// <param name=" failedCallback" type=" Function" optional=" true"
mayBeNull=" true" ></param>
/// <param name=" userContext" optional=" true" mayBeNull=" true" ></param>
return this._invoke(this._get_path(), 'GetServerTime' ,false,{},succeededCallback,failedCallbac
k,userContext); }}
ServerTime.registerClass( 'ServerTime' ,Sys.Net.WebServiceProxy);
ServerTime._staticInstance = new ServerTime();
ServerTime.set_path = function(value) {
ServerTime._staticInstance.set_path(value); }
ServerTime.get_path = function() {
/// <value type=" String" mayBeNull=" true" >The service url.</value>
return ServerTime._staticInstance.get_path();}
ServerTime.set_timeout = function(value) {
ServerTime._staticInstance.set_timeout(value); }
ServerTime.get_timeout = function() {
/// <value type=" Number" >The service timeout.</value>
return ServerTime._staticInstance.get_timeout(); }
```

```
ServerTime.set_defaultUserContext = function(value) {
ServerTime._staticInstance.set_defaultUserContext(value); }
ServerTime.get_defaultUserContext = function() {
/// <value mayBeNull=" true" >The service default user context.</value>
return ServerTime._staticInstance.get_defaultUserContext(); }
ServerTime.set_defaultSucceededCallback = function(value) {
 ServerTime._staticInstance.set_defaultSucceededCallback(value); }
ServerTime.get_defaultSucceededCallback = function() {
/// <value type=" Function"  mayBeNull=" true" >The service default succeeded callback.</
value>
return ServerTime._staticInstance.get_defaultSucceededCallback(); }
ServerTime.set_defaultFailedCallback = function(value) {
ServerTime._staticInstance.set_defaultFailedCallback(value); }
ServerTime.get_defaultFailedCallback = function() {
/// <value type=" Function"  mayBeNull=" true" >The service default failed callback.</value>
return ServerTime._staticInstance.get_defaultFailedCallback(); }
ServerTime.set_path( "/Books/WuTouJavaScript/3/ServerTime.asmx" );
ServerTime.GetServerTime= function(onSuccess,onFailed,userContext) {
/// <param name=" succeededCallback"  type=" Function"  optional=" true"
mayBeNull=" true" ></param>
/// <param name=" failedCallback"  type=" Function"  optional=" true"
mayBeNull=" true" ></param>
/// <param name=" userContext"  optional=" true"  mayBeNull=" true" ></param>
ServerTime._staticInstance.GetServerTime(onSuccess,onFailed,userContext); }
```

这些代码就是ASP.NET AJAX自动生成的JavaScript代理类，使用这些代理类就可以简单地调用ServerTime服务的各种方法。当然，这些代理类是需要MicrosoftAjax.js文件的支持的，它也需要被先行引入。

现在，再把AJAX5.htm改写成使用代理类的形式。于是，我们有了AJAX6.htm文件：

AJAX6.htm

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
   <title>服务器时间</title>
   <script src=" MicrosoftAjax.js"  type=" text/javascript" ></script>
   <script src=" ServerTime.asmx/js"  type=" text/javascript" ></script>
</head>
<body>
   <span id=" clock" ></span>
   <script type=" text/javascript" >

      function AsynRequest()
```

```
    {
      ServerTime.GetServerTime(UpdateClock); //调用ServerTime服务的GetServerTime方法
    };

    function UpdateClock(aTime)
    {
      document.getElementById("clock").innerHTML = aTime;
    };

    setInterval(AsynRequest, 1000);

  </script>
</body>
</html>
```

很棒！少了好多代码，而且代码读起来更容易理解，更优雅，已看不到XMLHttpRequest的影子了，只看到对Web Service的调用。

在这个例子中，引入两个外部的js文件：一个是Microsoft Ajax Library的MicrosoftAjax.js文件；另一个是对ServerTime服务的代理类js文件的引用。引用这两个外部的JavaScript代码之后，只需用一句ServerTime.GetServerTime()即可完成对WebServer方法的调用。

同时，请注意，我们无需再对服务器返回的结果做反序列化处理，因为生成的Web Service代理类已经帮我们做了这一切，直接得到需要的结果。其实，ASP.NET AJAX为我们提供了强大的JSON序列化和反序列化能力，这使得JavaScript与Web Service之间不但可以交换简单的数据，还能传递非常复杂的对象数据。

不仅如此，微软还把对JavaScript脚本的管理封装成ScriptManager控件。这样，只需将ASP.NET AJAX的一个ScriptManager控件放到我们的ASP.NET页面上，也就完成了对Microsoft Ajax Libaray那些脚本文件的自动引用。此外，ASP.NET AJAX还提供了一个ServiceReference标签来专门引用Web Service。下面来看看如何使用这些服务器控件和标签。

既然是服务器端的控件或标签，就不能用.htm文件来写了，得用.aspx文件。于是，我们把服务器时间程序改写成AJAX.aspx文件：

AJAX.aspx

```
<%@ Page Language=" C#" %>

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

```
<html xmlns=" http://www.w3.org/1999/xhtml" >
<head runat=" server" >
  <title>服务器时间</title>
</head>
<body>
  <form runat=" server" >
    <asp:ScriptManager runat=" server" >
      <Services>
        <asp:ServiceReference Path=" ServerTime.asmx" />
      </Services>
    </asp:ScriptManager>
    <span id=" clock" ></span>
  </form>
  <script type=" text/javascript" >

    function AsynRequest()
    {
      ServerTime.GetServerTime(UpdateClock); //调用ServerTime服务的GetServerTime方法
    };

    function UpdateClock(aTime)
    {
      document.getElementById( "clock" ).innerHTML = aTime;
    };

    setInterval(AsynRequest, 1000);

  </script>
</body>
</html>
```

这个ASP.NET页面运行的结果和前面一样，只是将原来静态的.htm改成了动态的.aspx页面。其中的<asp:ScriptManager>标签将自动在客户端引入Microsoft Ajax Library，而<asp:ServiceReference>也将在客户端自动引入WebService的代理类文件。

从浏览器中打开AJAX.aspx页面，并查看其源代码，我们将发现下面的HTML片段：

```
<script src=" /ScriptResource.axd?d=ALaFTtkKJ873cx7d1npe6qN448v<script src=" /ScriptResource.axd?d=A
LaFTtkKJ873cx7d1npe6qN448vWWiEp06cP5Re69p-q3CJx_OujUVhOVvyn5fuogF2L8TSJH2pplfmSV1--
3kSfQP9OS4NCxFLBwGc4DaU1&amp;t=633383519014531250" type=" text/javascript" ></script>

<script src=" /ScriptResource.axd?d=ALaFTtkKJ873cx7d1npe6qN448vWWiEp06cP5Re69p-q3CJx_
OujUVhOVvyn5fuo-XFLv44hnZVQx1erPSUJY0iO6wl8foxhopJfOMOxCBjQmCups2KqFXOXU3Cv6UjZ0
&amp;t=633383519014531250" type=" text/javascript" ></script>

<script src=" ServerTime.asmx/jsdebug" type=" text/javascript" ></script>
```

其中，前两个<script src=”/ScriptResource.axd?....>形式的语句，是引入了ASP.NET AJAX内嵌的JavaScript脚本资源，这两个资源内容分别等同于MicrosoftAjax.js和MicrosoftAjaxWebForms.js文件（或者是其调试版本）。而最后一个显然是引入了ServerTime服务的JavaScript脚本。

由于ScriptManager控件必须放在一个<form runat=”server”>的表单中，这让某些有洁癖的程序员感到不爽。因此，他们往往宁愿像AJAX6.htm那样，自己引用MicrosoftAjax.js文件和WebService的代理类。这样可以让最终的HTML文档更简洁和清晰一些，没有<form runat=”server”>带来的ViewState以及相关JavaScript脚本的负担。

为啥ScriptManager必须放在一个<form runat=”server”>的表单中呢？因为，ScriptManager是个大杂烩，不但提供对JavaScript脚本和WebService代理类的管理，还提供UpdatePanel等AJAX控件的支持。由于UpdatePanel是通过PostBack时判断ViewState等信息来提供部分呈现功能的，因此必须有<form runat=”server”>的支持。

在实际应用中，是否使用ScriptManager来管理JavaScript脚本和引用WebService要看具体情况。我们的意见是，如果您的页面需要回发数据或用到UpdatePanel时，当然应该使用ScriptManager了，如果不是这样，最好自己引用MicrosoftAjax.js文件和WebService的代理类。

用ASP.NET页面模型独有的控件形式，来引用Microsoft Ajax Library和WebService的代理类，使得AJAX编程变得非常简洁和明快，代码也非常优雅。这对于我们这些开发人员来说，真的是方便啊！不得不佩服ASP.NET AJAX的设计者和开发者，微软的算盘也打得何其精明啊！

5
AJAX之双手互搏

老顽童双手互搏之术堪称一绝。左右手分别使出两种不同的武功，既可以自娱自乐相互切磋，也可以两路夹攻来击退强敌。

可是世上能学会如此神奇武功的只有两个人：一个是笨得出奇的郭靖，一个是不谙世事的小龙女。就连天资聪明的黄蓉和勤奋好学的耶律齐都无缘这套神奇的功夫。

有趣的是，双手互搏的入门功竟然简单得出奇，那就是：左手画圆，右手画方！

同样，在学习编写AJAX程序时，有一点非常重要，也是：左手画圆，右手画方！

左手画圆，右手画方，就是典型的异步思维！这种思维习惯的转变，恐怕比AJAX技术本身更重要！

在说异步思维之前，先得说说异步调用。什么是异步调用呢？先看前面示例中的这段：

```
function AsynRequest()
{
  ServerTime.GetServerTime(UpdateClock); //WebService成功后会异步调用UpdateClock
};

function UpdateClock(aTime)
{
  document.getElementById("clock").innerHTML = aTime;
};

setInterval(AsynRequest, 1000); //每隔1秒异步调用AsynRequest
```

示例中用到的JavaScript函数setInterval()就是典型的异步调用！当我们发出setInterval(AsynRequest, 1000)之后，JavaScript会继续执行随后的代码，但每隔一秒钟AsynRequest函数将被回调。

而AsynRequest函数中的那句ServerTime.GetServerTime(UpdateClock);又是一个异步调用！因为发出这个服务请求之后，JavaScript并没有等待服务器的返回结果，而是执行随后的程序。等到服务器返回结果之后，会自动回调UpdateClock函数。

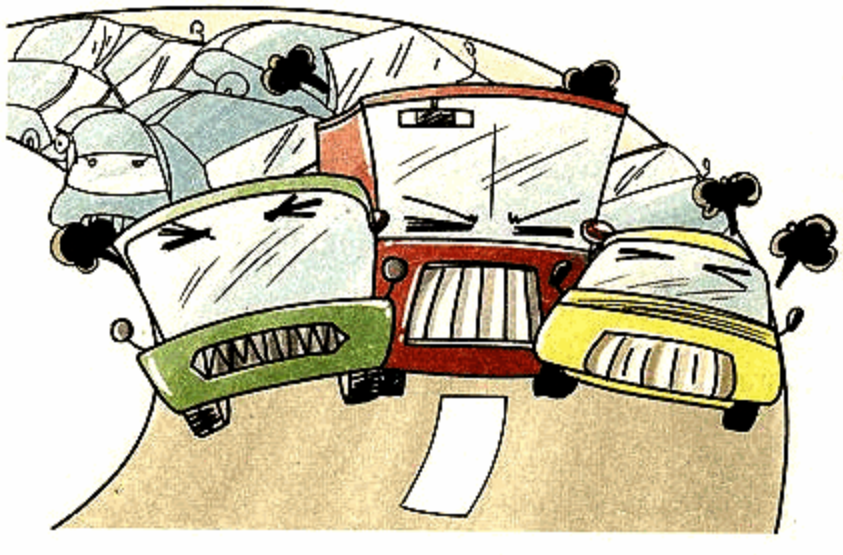
由于发出调用的代码线索启动了另一个步骤，而随后的回调代码线索并不与当前代码线索同步，因此这种调用方式就被称为“异步调用”。异步调用就需要用异步思维方式来思考程序的代码逻辑。

我们可以将以前思考程序的方式称为同步思维。所谓同步思维，就是指思绪总是从头到尾，并有始有终。不管思考和处理的问题如何复杂和深入，我们都始终保持着思维的连续性和完整性。

在同步思维的过程中，复杂的问题总是被划分成若干小问题来思考和处理，而小问题又可能被划分成更小的问题。同步思维的过程中，问题总是一个一个地解决的。如果，某个问题还没解决，我们必须等到把这个问题解决后，才会出解决下一个问题。

同步思绪结构就像一棵树，问题总是由一个节点开始思考，但不管其间可能深入到多少个子问题节点，但问题总要回归到开始的那个节点上来。

同步思维的优点是，思维的前后逻辑比较连贯，思维过程也非常严密，符合个人的思维习惯。我们在编程时，绝大多数情况都是在进行同步思维。我们已经习惯次序分明的代码，习惯了从过程到子过程的调用，也习惯了等待结果的返回，我们也因此对自己严谨的程序逻辑感到放心。



异步思维则有很大的不同！异步思维的思绪是发散和跳跃式的。在异步思维方式中，思绪经常会在不经意间跳转和发散，也会在一个思绪未果之前产生新的思绪，而完成的思绪也可能打断当前的思绪。

异步思维的各个思绪分支也并非没有关联，思绪间往往靠默契的沟通来传递和承接要处理的问题。异步思

维永远在处理问题，并不一定要求有个明确的起点和终点。看似思绪很混乱的样子，但问题总会在不断跳动的某个环节上得到全面解决。

异步思绪结构更像一个网，看似杂乱，但也有其运行的规律。这就像组织结构之间相互沟通和协作的过程一样，业务总在人与人，部门与部门之间传递和处理。一个部门处理完一件事，扔给另一部门即可，然后又处理自己的下一件事。在某一时刻看起来事情被分散得杂乱无章，但事情都会最终解决。

异步思维的优点是，处理问题的效率非常高，特别适合群体性或组织性思维。但对于个人来说，这似乎是教人不要只按一条线索思考问题，而是要一心二用，或者三心二意地做事。但在AJAX编程中的确需要这种思维方式。

我们来分析一下调用WebService的编程思路吧。按以前的同步思维习惯，一个典型的WebService的调用过程类似下面的语句：

```
try
{
   var result = TheWebService.TheMethod(param1, param2); //调用服务等待结果
   ... //处理返回结果
}
catch(err)
{
   ... //处理错误
}
... //处理其他事情
```

然而，在AJAX的异步调用过程中，就得习惯异步思维了。一个典型的WebService异步调用过程如下：

```
TheWebService.TheMethod(param1, param2, OnSuccess, OnFailed);      //异步调用服务
...          //接着处理其他事情

function OnSuccess(result)  //异步处理成功后会调用此函数
{
          ...          //处理返回结果
};

function OnFailed()          //异步处理失败后会调用此函数
{
          ...          //处理出错的情况
};
```

在我们发出对WebService调用之后，我们的思绪就得分三支了：一支是接下来处理的事情，一支是异步调用成功之后该怎样处理返回值，还有一支要考虑调用失败后

的情况。

在调用WebService之后，当前程序并不会等待异步操作返回的结果，而是立即执行随后的代码语句。而异步操作完成之后，将自动回调OnSuccess或OnFailed函数。似乎就在发出异步调用之后，程序的逻辑就被割裂开来，有点复杂了。然而，我们在各个逻辑分支中还会进一步发出其他的异步调用，问题将变得更加错综复杂。

一些老程序员们会惊异地发现，这些异步调用关系就像当年那充满争议的GOTO语句一样，阡陌相通而又变幻莫测。想当年，灵活使用GOTO语句来编写精巧的算法，可是一门高级的艺术活儿，需要超强的逻辑思维能力。但是，历经结构化编程思潮的革命和洗礼，GOTO语句已基本绝迹于当今的编程世界。如今的程序员们已经习惯了结构化编程的过程调用方式，习惯了已有的次序，习惯了等待需要的结果。

以桌面编程为主的程序员转向网页编程时，往往对网页间那种几乎没有过程性可言的松散逻辑关系感到头痛。因为用户打开我们的页面后，没有人能确定随后会发生什么。用户可能会点击一个链接或一个按钮，也可能去打开其他网站的网页，甚至干脆离开电脑去喝茶了，然后回来关机。这根本就不可能用那种有始必有终的连续思维来处理，而只能依靠离散和异步的思维。

在网页编程中，再用过去的那种过程式的同步思维来处理问题已经显得非常困难。因此，我们必须跳出过程式思维固有的模式，以发散和跳跃性的异步思维来思考和解决网页编程的问题。

尽管ASP.NET几乎为我们模拟了一个过程式的页面模型，但那只是单独的一个页面，而不同页面间的程序逻辑依然还是分散和异步的。然而，AJAX的出现让页面逻辑变得更加复杂，同一页面的各部分之间也都变成了异步处理的关系了。这使得AJAX的网页编程就像一团乱麻，剪不断，理还乱，搞得程序员个个都犯难，心似双丝网，中有千千结啊。

难道就没有一种可以指导异步编程的理论思想吗？当然有，只是目前现在这些理论思想都还不成熟，都还在发展过程中。尽管如此，我们还是可以进行一些初步探讨。

我们就随便来说说，先来看看编程语言的基础流程控制结构吧。

顺序结构最简单，上一条语句一定会处理数据，然后跳到下一条语句，这些数据自然就成为下一条语句的输入。而下一条语句也会根据输入产生输出数据，然后又跳到再下一条数据。

条件分支结构呢？更简单，它直接根据输入数据跳转到不同的语句，本身不再输

出数据。IF语句呢，分出两支；switch语句呢，分出多支；当然还有表驱动或指针驱动的那些多分支结构。

循环结构不过是顺序结构与条件分支的组合而已，不能算基础流程控制结构。

最后就是子程序调用结构，也简单，就是用明确地输入数据跳到一块代码，最后总要跳回来的一种流程控制结构。我们喜欢把跳回来的过程称为“返回”，同时也可能带回数据。

然后，我们来看看网页间的逻辑关系。

在浏览器中输入一个URL地址就打开一个网页，是否也可以把打开一个网页看作一条语句的执行呢？如果是，那么接下来就面临一个分支语句，用户点哪个链接，语句将跳转到另一个网页。点击网页的用户就像CPU的指令指针，用户的意愿就是影响指令指针跳动的控制数据。

当然，我们可以认为每一条语句都会等待很久，一条语句的指令周期会很长，甚至永远等待下去。然而，有些网页会立即重定向到新的网页，仿佛自动执行了两条顺序语句。

然而，与结构化编程不同的是，可以同时点击多个链接，打开多个网页。浏览网页的过程不需要回到原来的页，我们可以让尽可能多的网页来填满整个屏幕。就像一条语句之后，可以“同时”允许多条语句分支，而且根本不一定会返回。这对网页浏览来说是非常正常的，但对于结构化编程来说就不可理喻。

我们同样发现，跳转到一个网页时可能带有数据，如URL里的参数或者POSTDATA数据以及Cookie等。网页也会返回一些数据，但这些往往都是给跳转链接准备的。

那么，AJAX的异步调用呢？也有同样的想似性。

我们发出一个AJAX请求之后，也许还可以再发出另外的AJAX请求。但每一个AJAX请求会附带一些数据，而返回结果时也会跳到另一个回调函数地址。

我们可以将结构化的流程控制称为“强流程控制”，而把网页间的流程和AJAX异步调用流程控制称为“弱流程控制”。我们就可以这样认为，“强流程控制”的节点之间有连续性和同步性的特点，而“弱流程控制”的节点之间具有离散性和异步性的特点，但都是流程控制。

所谓的流程不就是“我做完事情就传递给你做事情”吗？我传递给你的时候，当

然会把我的结果同时传递给你，而你又会把我的结果作为输入来加以处理，从而延续未完的流程。如果，我非要等你做完之处理之后一定把结果给我，我才能继续处理的话，这就是同步流程。如果，我递交给你就不管了，转头做其他工作，而你呢，做完工作可以随时通知我，或者干脆又递交给其他人了，这就叫异步流程。

其中，我们能发现流程控制中的一个共同点，就是传递数据和延续处理。于是，我们可以将流程控制的传递和延续概念简称为“递延”。“递延”只是一个名词，如果要为它找一个恰当的英文单词应该就是Transfer and Continue！

“递延”的概念似乎很好地反映了流程控制的基本特点，因为不管是结构化编程，GOTO语句，网页间跳转，还是异步AJAX，都是流程递延的各种控制结构。

事实上，如果我们愿意放弃固有的连续和同步的思维方式，同样可以接受离散和异步的思维，久而久之也就习惯了。就像我们放弃了绝对的时间概念也就能理解相对论，而接受了离散的时空观也就习惯了量子论，难道不是吗？

要接受离散的异步思维，熟练掌握递延的思想，不是一件容易的事情。这需要首先放弃脑袋里装得满满的东西，将它们倒出来，至少暂时忘记。为啥只有傻乎乎的郭靖和天真的小龙女，才能学会老顽童双手互搏的绝技呢？心无他念，方可容物！

好了，接下来我们用一些实例来看看如何通过观念的改变，来解决一些现实开发中的问题，并带来一个怎样的新天地。

6
域名的鸿沟

现在，稍大一点的网站开发，往往都会面对多域名的局面。一个身份管理和登录服务的网站（如www.sunnyid.com），一个面向全球的主网站（如www.sunqee.com），一个面向国内用户的网站（www.sunqee.cn），还有网页邮箱访问（如mail.sunqee.com）、论坛（如bbs.sunqee.com）和博客（blog.sunqee.com）等等网站。

显然，多域名网站体系结构下的各个站点并非完全独立的。由于应用的需要，不同域名的站点之间需要互相沟通信息和相互提供服务。

在典型的单点登录或共享登录模式的多网站应用系统中，应用网站在确认用户身份的时候，需要与单一的或信任的身份认证网站进行沟通。如果需要，应用网站还会通过身份认证网站完成登录，然后身份认知网站通知应用网站继续处理。

用户一旦登录，随后进入其他应用网站时，该应用网站将自动识别登录状态，无须用户再次登录。当用户因为退出或认证过期导致登录状态结束，或者用户切换身份时，身份认证网站还需要即时通知已经登录过的应用网站，以保证用户状态的同步。

应用网站与身份认证网站间的沟通有两种典型的方式：一是应用网站的服务器直接与身份认证网站相互沟通，另一种是通过客户端来间接沟通双方的服务器信息。

服务器双方直接沟通的方式比较少用，主要是服务器本身一般都是无状态的，在用户会话同一性的确认上比较困难。不过，服务器间的直接沟通也可以开辟后台验证通道，辅助客户端更安全地完成身份验证，防止客户端到服务器间的信息被截听。

用户总在一个客户端登录多个网站，我们可以肯定客户端总是同一个用户。这种同一性假设不但要求用户是同一个人，而且同一时刻也只能有同一个人在登录状态，不存在两个人同时在一个客户端登录的状态。正是因为这个假设，不同的网站间才可以通过客户端完成身份认证等信息沟通。现在，绝大多数多域名的网站应用系统之间，都是采用客户端沟通方式来相互交换信息的。

一般来说，从一个网站的网页跳转到另一个网站的网页并带去本域名的信息，而另一个网站的网页也可以再跳转到本网站网页再带回需要的信息，这样可以实现两个域的沟通。不过，由于服务器是无状态的，这种沟通需要两个网站之间完成一系列连贯的网页跳转操作。频繁的网页请求导致效率低下，访问者会看到网址的不断跳转，用户体验很差。

此外，两个网站间页面跳转的沟通方式，对其他网站来说不会产生关联性效应。比如，用户在一个网站与身份认证网站间完成了登录，但进入另一个网站时，该网站还需跳转到身份认证网站核实。因此，网站间的相互交流还是应该以客户端直接沟通为主，页面跳转只能作为辅助手段。

然而，客户端的浏览器却总是很小心谨慎地区分和处理不同域名的程序和数据，禁止它们互相跨域操作。这是可以理解的，因为浏览器并不知道你的两个域名是有关联的，它只能把它们当成不同的网站。可是，没有人能保证所访问的每一个网站都是安全的，跨域操作会给网站访问者带来严重的安全性问题。

全世界的网站数量还在不断增长，网站间的信息链接千丝万缕、错综复杂。也许Google真的如自己宣称的那样永不作恶，但在利益的驱使下，且不说恶意网站比比皆是，一般的网站也总是在不断地收集和挖掘用户的隐私，甚至某些知名的大网站也喜欢玩玩擦边球。

给客户端带来安全性问题的主要是Cookie和JavaScript两样东西。尽管ActiveX等浏览器插件的安全性问题更严重，但不是我们今天讨论的主题，这里只谈论Cookie和JavaScript的跨域安全问题。

Cookie技术的发明为我们保存客户端状态带来了莫大的方便。Cookie的规范规定：一个域名的Cookie与其他域名相对独立，也就是说一个域名的Cookie只保存自己网站的客户端状态。然而，由于浏览器的一些漏洞，一些恶意网页可以窃取其他域名的Cookie，从而仿冒该域名的登录，窃取用户在该域名上的隐私资料，等等。

但随着浏览器厂商不断修补已知的问题和漏洞，现在要想在客户端跨域名访问Cookie几乎不可能了。Cookie变得越来越安全了。同时，一些浏览器还设计了Cookie的隐身策略管理，可以更好地保护用户隐私。

JavaScript几乎可以操纵网页中的任何元素，当然也可以读写Cookie。早期浏览器中的JavaScript不但可以操纵自己域名里的网页元素，还可以操纵其他域名的网页元素。这又使得恶意网站可以在自己的网页中嵌入frame或iframe来打开另一域名的网页，然后用JavaScript篡改嵌入网页的敏感元素，并向该域名服务器发出欺骗性的请求，从而窃取用户在该服务器上的私有信息。

当然，浏览器厂商们也一直在消除这些JavaScript的不安全漏洞。还记得刚升级到IE7的情形吗？原来运行得好好的AJAX程序纷纷报告“拒绝访问”的错误。那大都是利用了iframe可跨域名访问DOM的漏洞来相互通信的AJAX程序，要不就是用XMLHttpRequest来跨域访问数据的程序。现在，一个域名里的JavaScript是不可能操纵另一个域名里的DOM对象了，不管是用ifame嵌入的还是open打开的，通通拒绝访问。XMLHttpRequest跨域访问也同样被禁止了。

诚然，浏览器变得越来越安全了，但与此同时，AJAX编程也就越来越困难了。浏览器的安全增强确实是把双刃剑。浏览器厂商在修补一个个漏洞和增加一条条新的安全规则时，既堵住了恶意网站的攻击，也限制了一些合理的AJAX调用需求。

我们来看看现在的浏览器是怎样限制跨域访问的。所有的测试都基于IE7默认的安全设置来进行。请自行搭建一下测试用的运行环境。首先需要在本机的HOSTS文件中添加一些静态域名解析项，这个文件的完整路径一般是C:\WINDOWS\System32\drivers\etc\hosts。要添加的内容如下：

```
127.0.0.1    sunnyid.com
127.0.0.1    sunqee.com
127.0.0.1    sunqee.cn
```

接着，我们要在IIS中为这三个域名分别建立三个网站，各自有自己的网站主目录。这些网站都要支持ASP.NET 2.0以上的程序运行。

搞好测试环境后，我们先来看看跨域的XMLHttpRequest会有什么结果。编写一个跨域请求的静态页面CrossRequest.htm，放在sunqee.com网站下。它将发起一个对sunnyid.com网站的异步请求。

CrossRequest.htm

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
  <title>跨域XMLHttpRequest</title>
</head>
```

```
<body>
  <script type=" text/javascript" >
    var aRequest;
    if(!window.XMLHttpRequest)
      aRequest = new XMLHttpRequest()
    else
      aRequest = new ActiveXObject( "Microsoft.XMLHTTP" );

    //跨域的XMLHttpRequest请求，将抛出“没有权限”的异常。
    aRequest.open( "POST" , "http://sunnyid.com/AnyService.aspx" , true);

    alert( "End!" ); //由于上一句的异常，这条信息不会出现
  </script>
</body>
</html>
```

在IE7中打开http://sunqee.com/CrossRequest.htm之后，将看到下面的异常错误：

上面的JavaScript代码在aRequest.open()一句就被拒绝了，后面的语句也没有执行了。看来，XMLHttpRequest对象确实不能跨域操作了。

再来看看常用的iframe的情况又如何呢？

我们编写一个Home.htm文件，放到sunqee.com网站中。Home.htm中内嵌两个iframe元素，分别加载本域sunqee.com的Local.htm文件和外域sunnyid.com的Abroad.htm文件。三个文件分别如下：

Home.htm

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
  <title>主页</title>
</head>
<body>
  <iframe id=" Local" src=" http://sunqee.com/Local.htm" ></iframe>
    <input type=" button" value=" 访问本域框架" onclick=" alert(document.
getElementById( 'Local' ).contentWindow.location);" />
  <iframe id=" Abroad" src=" http://sunnyid.com/Abroad.htm" ></iframe>
    <input type=" button" value=" 访问外域框架" onclick=" alert(document.
getElementById( 'Abroad' ).contentWindow.location);" />
</body>
</html>
```

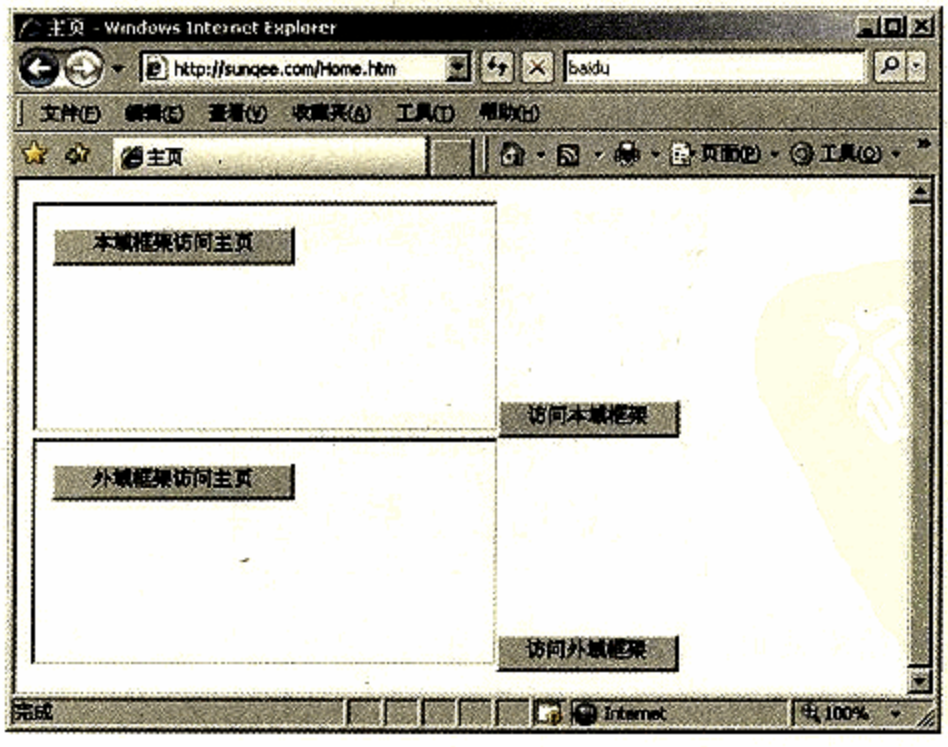
Local.htm

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
  <title>本域框架</title>
</head>
<body>
  <input type=" button" value=" 本域框架访问主页" onclick=" alert(parent.location);" />
</body>
</html>
```

Abroad.htm

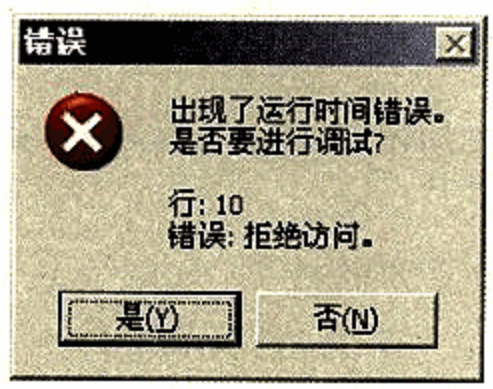
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd" >

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
  <title>外域框架</title>
</head>
<body>
  <input type=" button" value=" 外域框架访问主页" onclick=" alert(parent.location);" />
</body>
</html>
```

又来测试一下吧，在IE7中输入http://sunqee.com/Home.htm将看到这个页面：



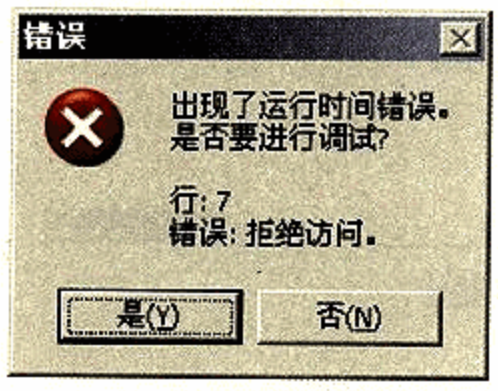
单击“访问本域框架”按钮之后，显示：

单击“访问外域框架”按钮之后，显示：



单击“本域框架访问主页”按钮之后，显示：

单击“外域框架访问主页”按钮之后，显示：



以上种种测试结果证明，iframe不再允许跨域访问对方的网页信息了。

我们还有别的途径吗?

还有一种跨域引用脚本的技术，就是利用<script>标签动态引入另一个域的JavaScript脚本文件。这个脚本往往是外域动态生成的JavaScript指令，在本域执行这个脚本，外域就可以操作本域的任何网页元素。尽管跨域引用脚本存在安全隐患，但目前跨域引用脚本还没有被限制。

然而，跨域引用脚本是建立在相信该域网站不会作恶的基础上的。如果两个域的网站都属于一个机构，这种技术也可以采用。但如果两个域并非属于同一机构，一旦提供脚本的一方作恶，就可以操纵另一方网页的所有内容。

现实中，有些网站引用了某些第三方网站的脚本服务之后，网页可能在某一天突然弹出不明窗口，或者被新嵌入广告，自己的网站也就变成了任由他人倾倒广告的垃圾场。显然，这样的跨域操作是不对等的，隐含严重的安全问题。

严峻的现实很残酷地告诉我们：安全的跨域通信确实是非常困难的!

我们该怎么办呢?

7
跨越域名的时空

当经过大量的努力都无法达到目的时，我们是不是应该静下心来反思一下呢？

是啊！浏览器安全性的不断增强，以前常用的客户端跨域名技术往往都建立在浏览器的漏洞之上，在如今的安全屏障下通通都无效了。我们还能找到突破安全屏障的缝隙吗？如果真的找到了，我们能过去，坏人也能过去，浏览器厂商还是会在随后的补丁中堵住这个漏洞。难道我们永远都会陷入与浏览器安全屏障的战争中吗？

显然，我们的方法完全错了！

可是，我们确实需要跨域名访问啊！

是啊，真的需要跨域名访问吗？

其实，我们需要的只是跨域名通信，而不是跨域名访问！我们必须转变思路，不要和浏览器的安全屏障硬碰硬地对着干，我们为什么不能顺着它走呢？

假如，身份认证网站所在域的Cookie保存了用户登录的状态，应用网站需要获得身份认证域的用户状态信息。因此，应用网站需要打开身份认证网站的一个页面，比如用iframe来加载，然后才有可能获取到身份认证域的信息。然而，浏览器的安全屏障不允许这样做！

我们能否换个思路呢？比如，应用网站域sunqee.com把获取用户登录状态的请求“递延”到身份认证域sunnyid.com，身份认证域sunnyid.com获得信息之后再想办法把结果“递延”到应用网站域sunqee.com，这样是不是就不用插手对方域的私有数据了呢？

是啊！我们的初衷并不是要操控另一个域的东西，而是要另一个域提供服务而已。这个服务本来就不该由原来的域来做，而应该由提供服务的域自己来做！难道不是吗？

也许，长期以来的同步思维使我们习惯了以自己为中心来思考问题，我们总在想当前域怎样去取得另一个域的信息，而很少考虑另一个域其实也可以自己做事的，只要你“递延”给它。另一个域做完事情，又可以“递延”给原来的域。事实上，“递延”不仅仅包含处理的转移和数据的交换，还可以包括身份的转换。

好了，接下来我们来看看能否实现这些思路。我们打算从sunqee.com域模拟一个服务请求到sunnyid.com域，输入参数“Hello”。sunnyid.com执行服务，给输入参数拼接一个“World”，得到结果“Hello World”，然后送回原域sunqee.com中显示。

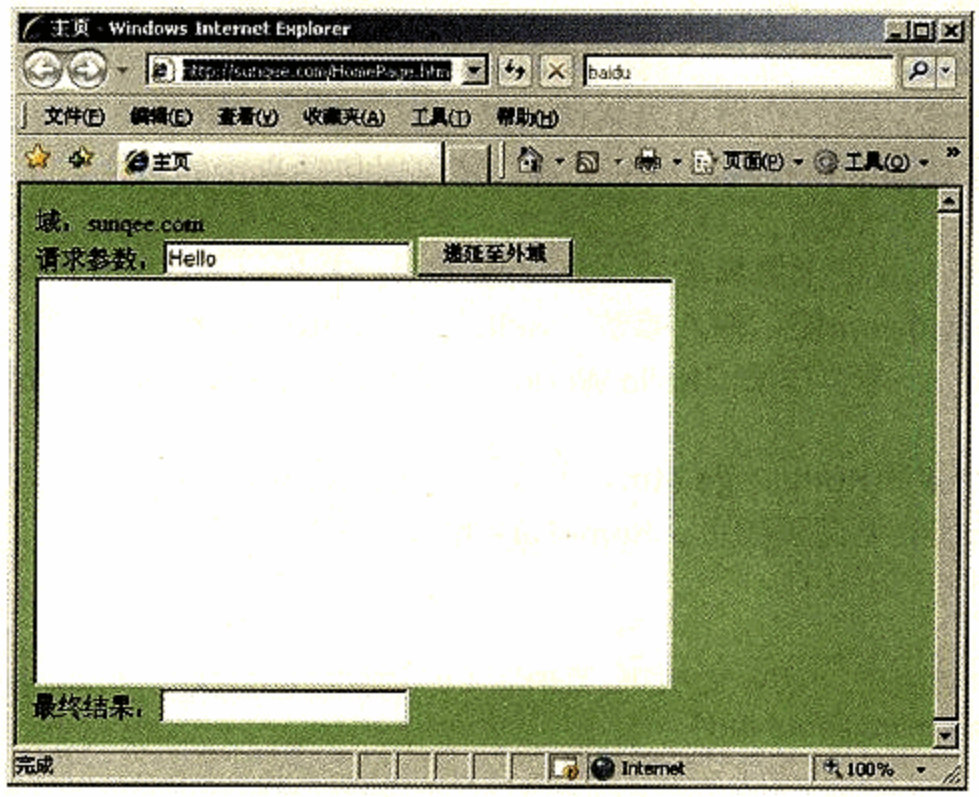
先编写一个主页HomePage.htm，网页流程将从这里开始。同时，我们将用一个内嵌框架iframe来加载要递延的页。HomePage.htm文件如下：

HomePage.htm

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

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>主页</title>
</head>
<body bgcolor="#99ff66">
  <span>域：sunqee.com</span><br />
  <span>请求参数：</span><input id="Param" type="text" value="Hello" />
  <input type="button" value="递延至外域" onclick="GotoAbroad();" />
  <br />
  <iframe id="Abroad" width="400px" height="260px" ></iframe>
  <br />
  <span>最终结果：</span><input id="Result" type="text" />
  <script type="text/javascript">
    var Abroad = document.getElementById("Abroad").contentWindow;
    var Param = document.getElementById("Param");
    var Result = document.getElementById("Result");

    function GotoAbroad()
    {
      //递延至外域AbroadFrame页面请求服务，并通过片段标识符传递参数
      Abroad.location = "http://sunnyid.com/AbroadFrame.htm#" + Param.value;
    };

  </script>
```

```
</body>
</html>
```

用IE7打开http://sunqee.com/HomePage.htm之后，将看到下面的信息：



其中，绿色背景表明当前域处在sunqee.com。页面上有请求的参数输入框，“递延至外域”的按钮：一个用来加载外域页的框架，一个接收最终结果的文本框。

我们为“递延至外域”的命令按钮绑定了一个onclick事件，来启动GotoAbroad方法,并让框架加载外域页面。请求参数是以片段标识符(fragment Identifier)的形式传递给后续页的，后续页可以通过其window对象location属性的hash值获得这些参数。

好了，接下来写一个外域的服务页AbroadFrame.htm。这个页将接收递延进来的参数，然后加以处理。AbroadFrame.htm文件如下：

AbroadFrame.htm

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

<html xmlns=” http://www.w3.org/1999/xhtml” >
<head>
  <title>外域框架</title>
</head>
<body bgcolor=” #ff6666” >
  <span>域：sunnyid.com</span><br />
```

```
    <span>收到参数：</span><input id=" Param" type=" text" />
    <input type=" button" value=" 递延至原域" onclick=" GotoLocal();" />
    <br />
    <iframe id=" Local" ></iframe>
    <br />
    <span>处理结果：</span><input id=" Result" type=" text" />
    <script type=" text/javascript" >
      var Local = document.getElementById( "Local" ).contentWindow;
      var Param = document.getElementById( "Param" );
      var Result = document.getElementById( "Result" );

      Param.value = location.hash.substring(1);  //从片段标识符析取参数
      Result.value = Param.value + " World";    //完成服务工作得到结果

      function GotoLocal()
      {
        //递延至原域的LocalFrame页面，并通过片段标识符传递结果
        Local.location = "http://sunqee.com/LocalFrame.htm#" + Result.value;
      };

    </script>
</body>
</html>
```

这个文件不是直接打开的，我们在主页中单击“递延至外域”按钮，将在主页的外域框架中加载**AbroadFrame.htm**文件。加载后的结果如下：

我们用红色来做外域页的背景，表明所在的域是与原域不同的。页面上显示了收到的参数，同时已经根据这些参数完成了服务处理，然后将结果显示到处理结果文本框中。

接下来是一个比较关键的问题，就是：怎样将结果送回主页面？

相信很多专研跨域名通信技术的朋友都往往在这里给困住了。这很正常，因为长期以来养成的结构化编程习惯，我们的潜意识想到就是“返回”。正如大多数人找到宝藏之后都想着尽快原路返回一样，然而原来的路已被浏览器的安全屏障给堵死了。

怎么办？我们为什么不能再往前走一步呢？再将结果递延到另一个内嵌框架如何？

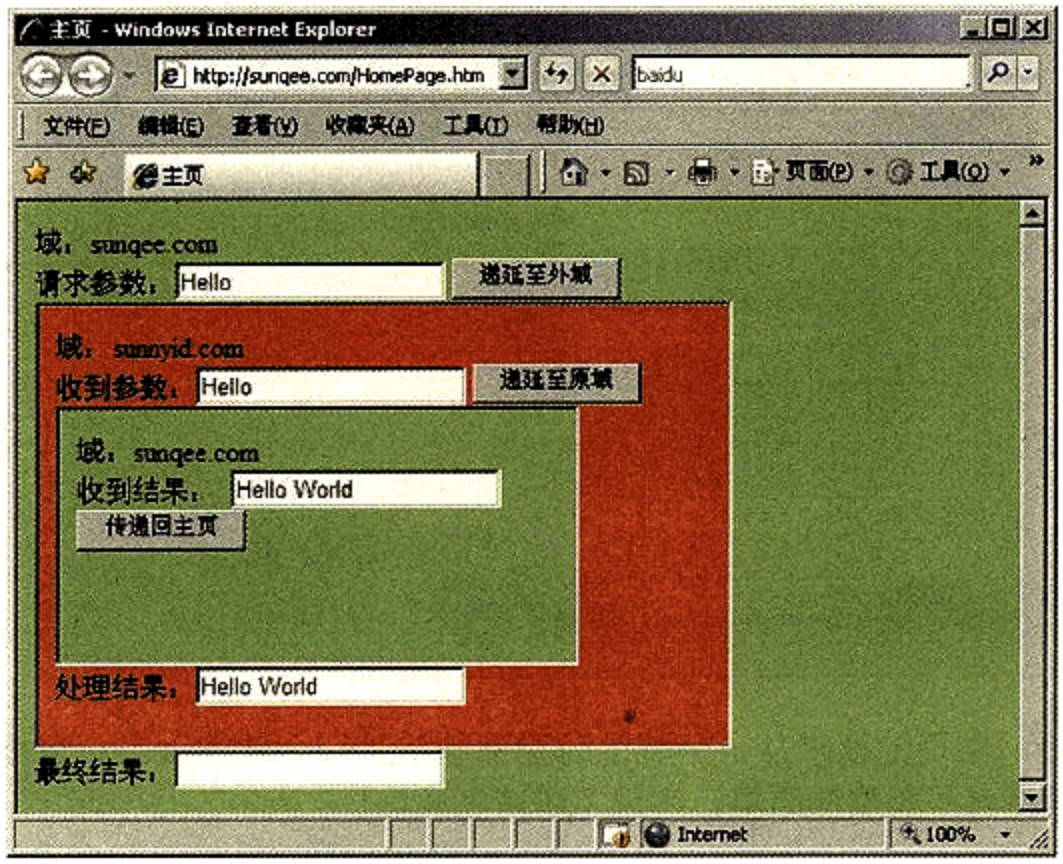
是啊！为什么不能在外域中再嵌入一个原域的网页呢？那试试吧！

我们将用“递延至原域”按钮来调用一个GotoLocal方法，这个方法将在新的内嵌框架中加载原域的一个LocalFrame.htm的文件。LocalFrame.htm文件如下：

LocalFrame.htm

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

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>本域框架</title>
</head>
<body bgcolor="#99ff66">
  <span>域：sunqee.com</span><br />
  <span>收到结果：</span>
  <input id="Result" type="text" />
  <input type="button" value="传递回主页" onclick="GotoHome();" />
  <script type="text/javascript">
    var Result = document.getElementById("Result");
    Result.value = location.hash.substring(1); //析取外域传递来的结果

    function GotoHome()
    {
      //通过parent.parent可以找到主页窗口
      var home = parent.parent;

      //由于本框架页与主页同域，浏览器是完全允许直接操作网页元素的
      home.document.getElementById("Result").value = Result.value;
    };
  </script>
</body>
</html>
```

有了原域的这个LocalFrame.htm文件之后，可以单击“递延至原域”按钮来加载此文件。加载后的结果如下：



我们看到红色的区域中又出现了一块绿色的区域，表明此内嵌页中的内嵌页与最外面的主页处在相同的域中了！

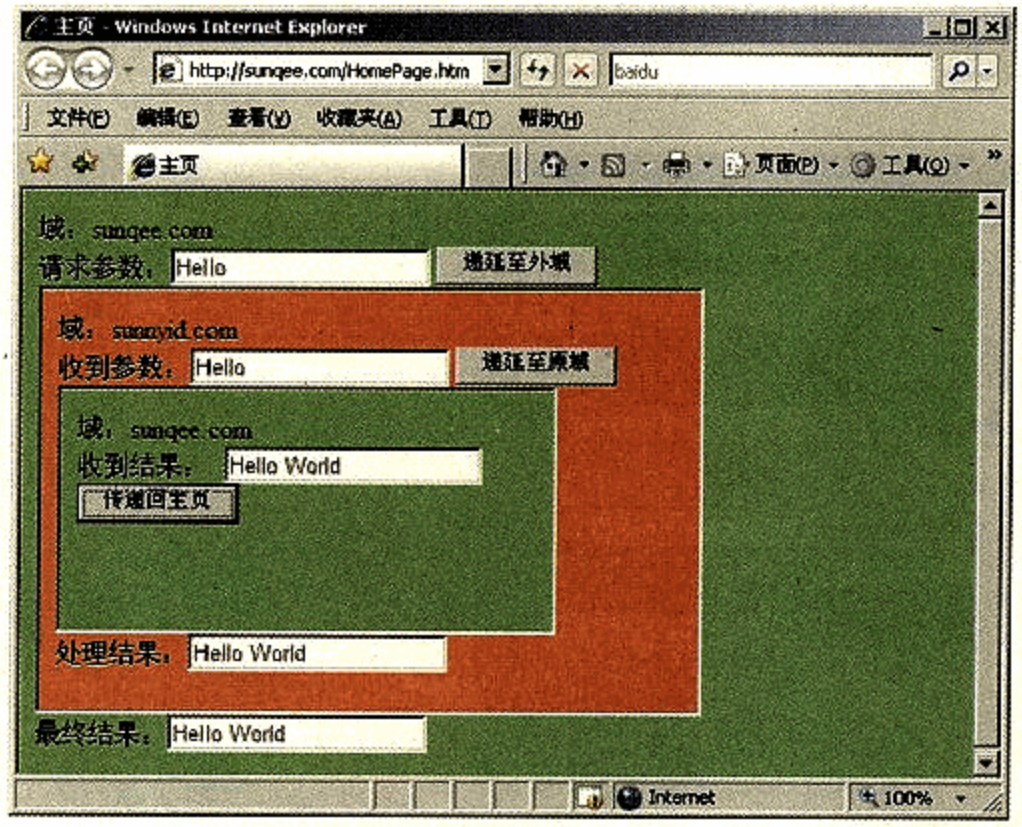
哇！我们看到了新的希望，因为往前跨一步，又到了一个新的时空。这个时空与我们的主页具有相同的域，尽管中间隔了一个禁区，一个外域的框架。那么，我们有没有什么合法的信道与主页的时空沟通呢？

有！就是parent.parent！

虽然域的安全屏障总是限制外域在自己的领土上捣鬼，却是允许外域过境访问的。因此，不管ifame的嵌套层次有多深，其parent属性总能访问到，而parent.parent也总能访问到，直到最上层网页。而当一个域的访问过境到自己域的领土时，又有权操作本域的任何元素了，同域的页面总是互相信任！

于是，我们在LocalFrame.htm中只需要以parent.parent为起点，就可以和主页沟通了。我们为此做了一个“传递回主页”的按钮，此按钮触发的JavaScript函数，直接将结果送回首页的最终结果文本框。单击“传递回主页”按钮之后，将在主页里看到我

们需要的结果：



非常不错！主页的请求从一个域的时空穿越了另一个域的时空，最终又能回到自己的时空，并带回了需要的结果，实现了跨域名通信！

更让我们高兴的是，这个穿越过程并没有违背域名安全的国际法，也并不存在一个域干涉另一个域内政的问题。因为我们改变了强流程的控制战略，而采用了弱流程的递延战略，从而将强攻安全防线的错误战术，转变为里应外合的协作战术，实现了两个域名间合理合法地沟通信息。

安全防线依然安全，域之间也可以自由地相互通信，一切都很自然和谐。这种跨域通信机制与跨域JavaScript脚本机制方式相比，是双方对等的安全沟通，不存在一方欺负另一方的问题。因此，我们可以基于此通信机制构建不同机构间的安全和对等的通信协议，实现公平和对称的跨域通信应用。

事实上，在这个过程中，我们并没有必要嵌入两层iframe，嵌入一个iframe就足够了。在递延的思想中，如果当前环境不需要保存状态并且也不再有后续分支出现，就完全可以利用现有环境完成递延。因此，在递延回原域的过程中，我们可以不再创建新的iframe，而是直接让当前iframe跳转到原域网页即可。随后要讨论的“特使协议”将继续研究和完善这些具体细节。

8
特使协议

不同域名网站间的关系就像国家与国家之间的关系。每个国家的政府都独立管辖本国的内部事务，没有任何一个国家愿意让其他国家来干涉自己的内政，这是基本的安全底线。但是，国家与国家之间也必须相互协作。由于存在经济往来和文化交流等活动，国家与国家间也存在相互服务的工作。

中国政府有自己的域名gov.cn，美国政府也有自己的域名gov.us。

显然，现在的域名安全标准是不允许gov.cn里的JavaScript去访问和控制gov.us域名里的数据和对象的、包括变量、代码，DOM等等，Cookie也不例外。

那么，中国gov.cn要求美国gov.us提供服务，比如，调查外逃贪官在美国的财产资料。但美国是绝对不准外人插手国内事务的，他并不相信中国，因此不允许中国政府直接派人调查。其实，中国也并不是要插手美国的内政，中国只需要一个调查结果，该怎么办呢？

中国gov.cn可以把请求提交给美国大使馆，美国使馆的域名也是gov.us，但在中国境内。美国使馆就像嵌入在中国网页里的iframe，背后代表的是美国政府。美国政府当然是相信自己的使馆的，美国政府将派自己人进行调查，然后取得了结果。

不过，美国政府也不能直接把调查结果放到中国政府的办公室，中国政府也是不允许的。因为，让外人直接进来就有可能窃取gov.cn的其他情报。又该怎么办呢？

美国gov.us同样也可以把结果交给美国境内的中国使馆，一个内嵌于美国境内的iframe。中国使馆的域名也是gov.cn，对中国使馆传递过来的结果，中国政府当然就放心了。

于是，通过中美两国使馆的相互传递，中国gov.cn就成功取得美国gov.us提供的信息和服务。但所有过程完全遵循国际法，大家都认为是安全可靠的！

为了更加形象地表达和理解这套跨域名的安全通信协议，我们将此协议称为“特使”协议，英文命名为Envoy Protocol好了。我们也给出了实现这个“特使”协议的程序文件，由envoy.js和envoy.htm文件构成。这两个文件只是示例，展示“特使”协议的实现原理。如果要用到正式的网站应用中，还需进行进一步的改写和封装。

envoy.js

```
//通讯服务函数
//domain: 被调用域
//method: 被调用方法名
//callback: 回调方法名
//随后可跟任意调用参数
function Envoy(domain, method, callback)
{
  //动态建立一个隐藏的iframe元素：
  var frame = document.createElement( “iframe” );
  frame.style.width = 0;
  frame.style.height = 0;
  frame.style.visibility = “hidden” ;
  document.body.appendChild(frame);

  //序列化请求服务的参数：
  var params = [];
  for(var i=3; i<arguments.length; i++)
    params.push(Sys.Serialization.JavaScriptSerializer.serialize(arguments[i]));

  //用window对象的name属性可传递超长参数，突破URL传递信息的长度限制
  frame.contentWindow.name = params;

  //构造URL并递延到被调用域的通讯页，片段标识符为服务命令，格式：方法名/回调方法名@回调域
   var url = “http://” + domain + “/envoy.htm#” + method + “/” + callback + “@” +
location.host;
  frame.contentWindow.location = url;
};
```

envoy.htm

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

```
<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
  <title></title>
  <script type=" text/javascript"  src=" /MicrosoftAjax.js" ></script>
  <script type=" text/javascript"  src=" /envoy.js" ></script>
  <script type=" text/javascript"  src=" /service.js" ></script>
</head>
<body>
<script type=" text/javascript" >
//<![CDATA[
  if(!parent||parent==this)
    alert( "envoy.htm can be visited in iframe only!" )
  else
  {
    //片段标识符去掉第一个#字符就是需要解释的命令
    var command = location.hash.substring(1);

    //如果命令中有"/"分隔，表示是请求服务的命令，否则是回调原域方法的命令
    var i = command.indexOf( "/" );
    if(i>=0)   //处理请求服务命令
    {
      //解析出回调方法名callback，回调域domain，请求方法名method
      var j = command.indexOf( "@" );
      var callback = command.substring(i+1, j);
      var domain = command.substring(j+1);
      var method = command.substring(0, i);

      //构造请求服务的可执行语句并执行，window.name属性是传过来的参数
      command = method + "(" + window.name + ")";
      var result = eval(command);   //解释执行服务命令，一般是service.js中的函数

      //将请求服务的结果序列化，并再次通过window.name传递给将要加载的原域通讯页
      window.name = Sys.Serialization.JavaScriptSerializer.serialize(result);

      //构造回调的命令格式，并递延到原域通讯页面
      window.location = "http://" + domain + "/envoy.htm#" + callback;
    }
    else      //处理回调命令
    {
      //构造回调用的可执行命令语句，并在父窗口环境中执行
      command = command + "(" +window.name + ")";
      parent.eval(command);
    };
  };
//]]>
</script>
</body>
</html>
```

只要将envoy.js和envoy.htm文件放入需要跨域通信的网站的根目录中，就可以实现异步调用对方的JavaScript函数，并能收到回调的结果。不同网站可以在自己的service.js中编写服务函数，以提供给其他人调用。当然，envoy.js和envoy.htm文件都需要MicrosoftAjax.js的支持，主要用到了里面的序列化函数。如果自己写一个JSON序列化函数，也可以去掉对MicrosoftAjax.js的引用。

在envoy.js中，我们提供了一个“特使”函数Envoy，只要指明被调用域名、被调用方法名、回调方法名，以及调用参数，特使函数就可以完成跨域的异步调用。

Envoy函数动态创建一个隐藏的iframe来加载被调用域的envoy.htm页面，实现对被调用的递延。其中，要调用的方法名、回调域和回调方法名，是通过片段标识符传递给被调掉域的envoy.htm页面的，即，URL中“#”之后的部分。而调用的参数并没有附加在URL的后面，而是通过iframe中window对象的name属性来传递的。

为什么要用window对象的name属性替代URL的片段标识符来传递参数呢？因为，name属性是一个string类型的属性，长度几乎没有限制。而URL是有最大长度限制的，遇到超长参数时将出现错误。采用iframe中window对象的name属性，可以传递非常大的JSON对象，能实现大数据量的跨域通服务。

envoy.htm文件是一个实现域名间通信的通用页面。此页面将被调用方的域以iframe形式加载，或者在内嵌iframe中从被调用方的envoy.htm中跳转回来。我们并没有创建两层iframe，即iframe中的iframe。因为，没有这个必要！当被调用域取得调用结果时，本身的页面状态也就失去意义，直接用当前框架完成回调页面跳转，即可将结果带回到自己的域里。只用一个隐藏iframe，可以让页面代码精简很多。

由于对envoy.htm通信页面的加载默认都是GET方式的，浏览器从服务器加载过一次envoy.htm文件后，envoy.htm文件就会被客户端缓存。再次用到envoy.htm文件时，就不会再往返服务器了。而URL中的片段标识符不断变化也与服务器无关，这都是客户端的事儿。这使得特使协议在跨域交换信息的应用中具有极高的效率。

这套特使协议建议将本网站可以提供的客户端服务都写在service.js文件中。我们可以根据具体应用编写多个对外服务的JavaScript函数，这些函数自由使用任意的参数和类型，也可以返回任意类型的结果，甚至返回复杂的对象。这些服务函数可以自由调用本域的服务器的Web Service，也可以再通过特使协议调用其他域的客户端服务。

接下来，我们将使用这套特使协议，配合Cookie的读写，来实现一个多域名的单点共享登录模型。

9
单点共享登录模型

假如，sunnyid.com是专门提供身份认证服务的网站，sunqee.com是英文的应用网站，而sunqee.cn是中文的应用网站。我们的目标是：

1. 应用网站可以轻松地判断当前用户是登录用户还是匿名用户；

2. 应用网站登录时将跳转到身份认证网站，完成登录再跳转回来；

3. 用户从一个应用网站完成登录后，进入另一个网站时，无须再登录；

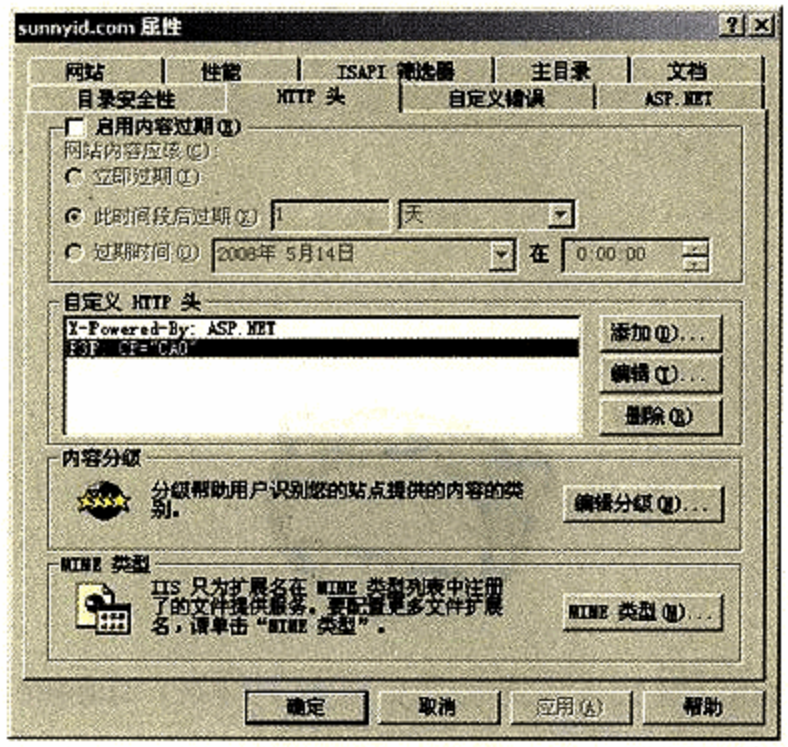
4. 用户从任意地方退出时，所有应用网站同时自动退出登录状态。

我们打算用一个Cookie来记录用户的登录状态，这个Cookie就叫PASSPORT好了。我们将通过身份认证服务器设置或清除这个PASSPORT，然后在不同域的客户端之间传递。

为了能让浏览器在默认的隐私设置下接受Cookie，我们需要为各个网站添加紧凑的隐私策略声明。隐私策略声明就是向访问的浏览器庄严宣告：本网站内容和应用是属于什么范畴的，本网站愿意为此声明承担法律责任。

隐私策略声明的紧凑形式就是在HTTP响应的头部增加P3P的键值，其内容为CP="…"的形式。例如，CP="CAO PSA OUR"，其中的CP是紧凑隐私策略的缩写，而引号中的每三个字母代表一个内容类别的缩写。

对于动态页面，可以直接给每一个页面响应增加P3P头部。对于静态页面，可以通过设置IIS网站的HTTP公用头部信息，来给响应增加P3P声明。下图就是给sunnyid.com网站添加自定义HTTP头之后的情况：



对于IE6和IE7等支持隐私策略声明的浏览器，在较高隐私的安全级别下，具有隐私策略声明的网站才可以读写Cookie。因此，我们需要给sunnyid.com，sunqee.com，sunqee.cn都增加相应的隐私策略，以确保在较高隐私安全级别下也能正常读写Cookie。

有了这些准备，我们就可以开始实现多域名的单点共享登录模型了。

在多域名的单点共享登录模型中，用户是否登录以及登录的PASSPORT，都存在身份认证网站sunnyid.com的客户端Cookie中。应用网站sunqee.com或sunqee.cn都是通过获取身份认证域中的PASSPORT来得知用户登录状态的。

对于应用网站，用户第一次访问应用网站sunqee.com的任意动态页面时，先从身份认证网站sunnyid.com的客户端获取用户的PASSPORT，并保存在自己域的Cookie中。以后再访问本应用网站的其他动态页面，将无需再去身份认证域获取PASSPORT了。

而应用网站第一次从身份认证网站获取PASSPORT时，身份认证网站需要记住该应用网站的域名。身份认证网站会在自己客户端Cookie中，记录曾经请求身份服务的每一个引用网站的域名。一旦用户在身份认证网站发生登录或注销行为，身份认证网站将在客户端通知每一个已知的应用网站域，更新相应的PASSPORT。

此外，身份认证网站sunnyid.com还需提供登录和注销页面，提供给应用网站请求登录或注销使用。应用网站sunqee.com或sunqee.cn需要用户登录时，会跳转到身份认证网站sunnyid.com的登录页面，完成登录后，再跳转会原来的应用网站页面。注销过程也与登录过程类似。

我们先看一下身份认证网站sunnyid.com的service.js文件，这是一个客户端的服务文件。

service.js (sunnyid.com)

```
//获取当前的PASSPORT
function GetPassport(domain)
{
   RegisterDomain(domain);
   var passport = GetCookie( "PASSPORT" );
   if(!passport)
      passport = "ANONYMOUS";
   return passport;
};

//记录应用网站域名函数:
function RegisterDomain(domain)
{
   var domains = GetCookie( "DOMAINS" );
   if(!domains)
      SetCookie( "DOMAINS", domain)
   else
   {
      domains = domains.split( "," );
      for(var i=0; i<domains.length; i++)
        if(domains[i] == domain)
          return;
      domains.push(domain);
      SetCookie( "DOMAINS", domains.toString());
   };
};

//通知所有应用网站域名更新PASSPORT
function BroadcastPassport(passport, continuator)
{
   window.domains = GetCookie( "DOMAINS" );
   if(!domains)
      coninuator()
   else
   {
      window.continuator = continuator;
      window.passport = passport;
      window.domains = domains.split( "," );
      window.domainIndex = 0;
      SetDomainPassport();
   };
}
```

```
//设置但一应用域名的PASSPART
function SetDomainPassport()
{
   if(window.domainIndex >= window.domains.length)
      window.continuator()
   else
   {
      Envoy(window.domains[domainIndex], "SetPassport", "SetDomainPassport", window.
passport);
      window.domainIndex ++;
   };
}

//设置PASSPORT
function SetPassport(passport)
{
   SetCookie("PASSPORT", passport);
};

//读取Cookie的支持函数
function GetCookie(name)
{
   var value = null;
   if (document.cookie && document.cookie != '')
   {
      var cookies = document.cookie.split(';');
      for (var i = 0; i < cookies.length; i++)
      {
         var cookie = cookies[i].replace(/^\s+|\s+$/g, '');
         if (cookie.substring(0, name.length + 1) == (name + '='))
         {
            value = decodeURIComponent(cookie.substring(name.length + 1));
            break;
         }
      }
   }
   return value;
};

//写入Cookie的支持函数
function SetCookie(name, value)
{
   if (typeof value != 'undefined')
      document.cookie = name + '=' + encodeURIComponent(value);
};
```

```
//用POST方式的强制刷新跳转函数
function GotoUrl(url, target)
{
   var form = document.body.appendChild(document.createElement("form"));
   form.action = url;
   if(target) form.target = target;
   form.method = "POST"
   form.submit();
   document.body.removeChild(form);
};
```

为了实现真正的AJAX应用，我们在身份认证网站sunnyid.com中编写了Identity.asmx服务。这个WebService只是示例性的，提供给登录页面Login.aspx和注销页面Logout.aspx调用。

Identity.asmx

```
<%@ WebService Language="C#" Class="Identity" %>

using System;
using System.Web;
using System.Web.Services;
using System.Web.Script.Services;

/// <summary>
///提供身份服务
/// </summary>
[ScriptService]
public class Identity : WebService
{
   [WebMethod]
   public string Login(string UserAcct, string UserPswd)
   {
      //示例用的帐号和密码，测试时请严格按此帐户密码登录
      if (UserAcct != "LEADZEN" || UserPswd != "1234")
         return "";
      return UserAcct;
      //此处直接返回帐号做PASSPORT仅作示例，实际应用中应该生成加密的PASSPORT返回
   }
   [WebMethod]
   public void Logout()
   {
      //实际应用中用于清除用户在服务器端的状态。
   }
}
```

提供给应用网站的登录页面Login.aspx和Logout.aspx都会在客户端调用Identity.asmx服务，然后再跳转回应用网站原来的页面。Login.aspx和Logoug.aspx文件如下：

Login.aspx

```
<%@ Page Language=" C#" %>

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head runat=" server" >
  <title>用户登录</title>
  <script type=" text/javascript" src=" /MicrosoftAjax.js" ></script>
  <script type=" text/javascript" src=" /envoy.js" ></script>
  <script type=" text/javascript" src=" /service.js" ></script>
  <script type=" text/javascript" src=" /Identity.asmx/js" ></script>
</head>
<body>
  <span>用户帐号：</span><input id=" userAcct" type=" text" />
  <br />
  <span>用户口令：</span><input id=" userPswd" type=" password" />
  <br />
  <input id=" btnLogin" type=" button" value=" 登录" />
  <br />
  <span id=" errorText" ></span>
  <script type=" text/javascript" >
    $get( "btnLogin" ).onclick = function () //绑定登录按钮的点击事件
    {
      var userAcct = $get( "userAcct" ).value;
      var userPswd = $get( "userPswd" ).value;
      //调用身份验证服务的Login方法，成功转GotoSuccess，出错转GotoBack
      Identity.Login(userAcct, userPswd, GotoSuccess, GotoBack);
    };

    function GotoSuccess(passport)
    {
      if(!passport)
        $get( "errorText" ).innerHTML = "登录错误，请重新登录！"
      else
      {
        //将PASSPORT保存在Cookie中，共其他网站请求时，无须再去服务端取。
        SetCookie( "PASSPORT" , passport);

        //向所有已知的应用网站域通知新的PASSPORT，然后递延到GotoBack函数
        BroadcastPassport(passport, GotoBack);
      };
    };

    function GotoBack()
    {
```

```
        GotoUrl( "<%=Request.UrlReferrer%>" ); //转到原应用网站页（带刷新）
      };
    </script>
</body>
</html>
```

Logout.aspx

```
<%@ Page Language=" C#" %>

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head runat=" server" >
   <title>用户注销</title>
   <script type=" text/javascript" src=" /MicrosoftAjax.js" ></script>
   <script type=" text/javascript" src=" /envoy.js" ></script>
   <script type=" text/javascript" src=" /service.js" ></script>
   <script type=" text/javascript" src=" /Identity.asmx/js" ></script>
</head>
<body>
   <script type=" text/javascript" >
      //调用身份认证服务的Logout方法，成功转GotoSuccess，出错转GotoBack
      Identity.Logout(GotoSuccess, GotoBack);

      function GotoSuccess()
      {
         //设置PASSPORT为匿名
         var passport = "ANONYMOUS";
         SetCookie( "PASSPORT", passport);

         //向所有已知的应用网站域发通知，然后递延到GotoEnd函数
         BroadcastPassport(passport, GotoBack);
      };

      function GotoBack()
      {
         GotoUrl( "<%=Request.UrlReferrer%>" );  //转回原应用网页
      };
   </script>
</body>
</html>
```

Login.aspx和Logout.aspx都采用了客户端调用WebService的方式，而没有用WebForm的PostBack机制。对MicrosoftAjax.js文件和Identity.asmx服务的引用都是直接的<script>引用，而没有用ScriptManager控件。

WebForm和PostBack机制并不适合当前的应用环境，因为服务器只管提供识别用户和提供PASSPORT，而域名间的PASSPORT的沟通都是在客户端进行，与服务器无关。这样也能最大限度减少客户端与服务器间的往来。

此外，我们在服务器端采用了Request.UrlReferrer属性来识别应用网站的原始页面。因而，应用网站请求登录或注销页面时，无须将自己的网页地址附加到URL的查询字符串中。用户在登录过程中，URL不会出现后跟的原网址，也不会出现交换的PASSPORT信息，PASSPORT完全通过客户端暗中交换了。这在相当程度上又增强了身份认证服务的安全性。

然后我们再来看看应用网站方面。

应用网站提供了一个全局的身份认证机制，这个身份认证机制并非ASP.NET提供的Windows, Passport 和 Forms身份认证方式。而是一个自行处理的，简单有效的身份认证方式。这个身份认证机制主要是编写了一个Global.asax应用程序文件，并在其中统一处理了对用户身份的鉴别和授权。Global.asax文件如下：

Global.asax

```
<%@ Application Language=" C#" %>
<%@ Import Namespace=" System.Security.Principal" %>
<script runat=" server" >
  void Application_AuthenticateRequest(object sender, EventArgs e)
  {
    string username;
    var path = Request.Url.LocalPath;
    if (path.EndsWith( ".aspx" ))
    {
      var passport = Request.Cookies[ "PASSPORT" ];
      if (passport == null)  //没有PASSPORT表示还未鉴别访问者身份
        Server.Transfer( "/identity.htm" );  //暗中转向身份鉴别页，后面代码将不再执行。

      //身份鉴别页面会重新加载访问页，以明确身份刷新访问页面
      if (passport.Value == "ANONYMOUS" )
        username = "" ; //匿名用户
      else
        username = passport.Value;  //从PASSPORT得到帐号信息
    }
    else
      username = "" ;

    //建立访问者身份对象
    var identity = new GenericIdentity(username);
    Context.User = new GenericPrincipal(identity, new string [0]);
  }
</script>
```

Global.asax文件处理了标准的Application_AuthenticateRequest事件，任何对ASPX页面都将引发此事件。当用户第一次访问应用网站的应用页面时，从Request中无法得到PASSPORT的信息。这时，Global.asax会将请求导向一个identity.htm静态页面，此页面将立即返回客户端完成PASSPORT的读取，并重新加载同一个URL请求。identity.htm文件的内容如下：

Identity.htm

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head>
  <meta http-equiv=" pragma" content=" no-cache" />
  <title></title>
  <script type=" text/javascript" src=" /MicrosoftAjax.js" ></script>
  <script type=" text/javascript" src=" /envoy.js" ></script>
  <script type=" text/javascript" src=" /service.js" ></script>
</head>
<body>
  <script type=" text/javascript" >
  //<![CDATA[
    //去身份验证域获取PASSPORT并递延到本域的PutPassport函数
    Envoy( "sunnyid.com", "GetPassport", "PutPassport", window.location.host);
    function PutPassport(passport)
    {
      SetPassport(passport); //设置本域的PASSPORT
      location.reload();    //重新装载页面，该页面能得到PASSPORT信息
    };
  //]]>
  </script>
</body>
</html>
```

identity.htm文件被发送回客户端之后，它立即请求身份认证域sunnyid.com提供登录用户的PASSPORT。而得到PASSPORT之后，先要保存在本域的Cookie中，然后立即再次加载原来的URL。这时，服务器的Global.asax就得到PASSPORT数据了。这个过程瞬间发生，用户从浏览器外表看不到任何痕迹。而且，这个过程也只发生在第一次访问应用网站页面时，以后的访问PASSPORT总是确定的了。

如果Global.asax从已知的PASSPORT中解码出登录的用户信息，则会建立一个已知用户名的Identity对象。否则，建立一个未知用户名的Identity对象。同时，Global.asax将根据Identity对象建立当前Context对象的User主体信息。有了这层身份认证信息，随

后的任何页面都能够从Context.User.Identity.IsAuthenticated属性来判断用户是否登录，并能通过Context.User.Identity.Name属性来获知登录的用户名。

我们给应用网站编写了一个Default.aspx页面来做示范。Default.aspx文件内容如下：

Default.aspx

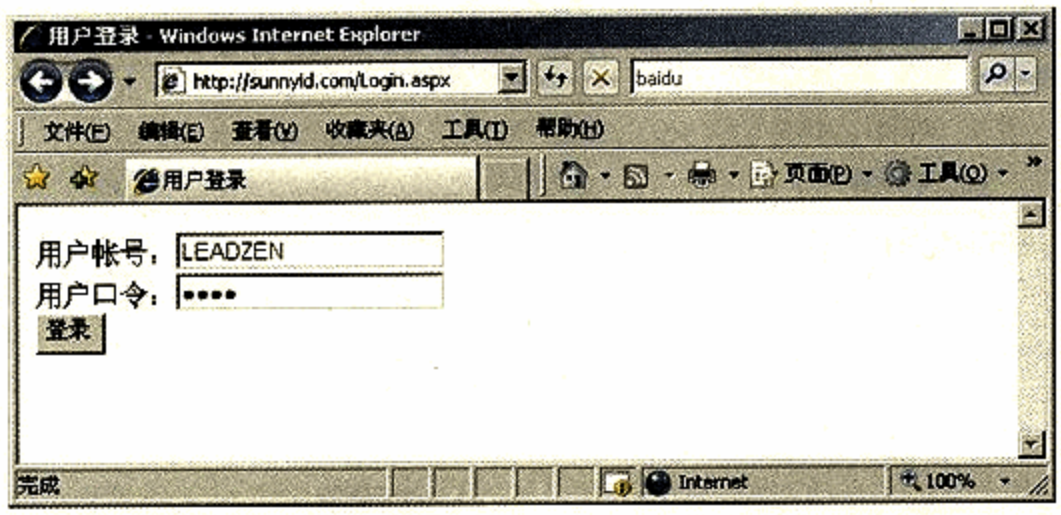
```
<%@ Page Language=" C#" %>

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

<html xmlns=" http://www.w3.org/1999/xhtml" >
<head runat=" server" >
  <title>无标题页</title>
</head>
<body>
  <% if (Context.User.Identity.IsAuthenticated)  //是否是授权用户
    {%>
     <span>欢迎您，<%= Context.User.Identity.Name %> ！ </span>
     <a href=" http://sunnyid.com/Logout.aspx" >注销</a>
  <%}
    else                                  //否则是匿名用户
    {%>
     <span>您还未登录！ </span>
     <a href=" http://sunnyid.com/Login.aspx" >登录</a>
  <%}%>
</body>
</html>
```

这个Default页面相当简单。如果用户没有登录，就显示登录按钮；如果用户已经登录，就显示欢迎信息和注销按钮。

好了，我们来演示一下这套单点共享登录的效果吧。

先用浏览器打开http://sunqee.com/Default.aspx，将看到下面的结果：

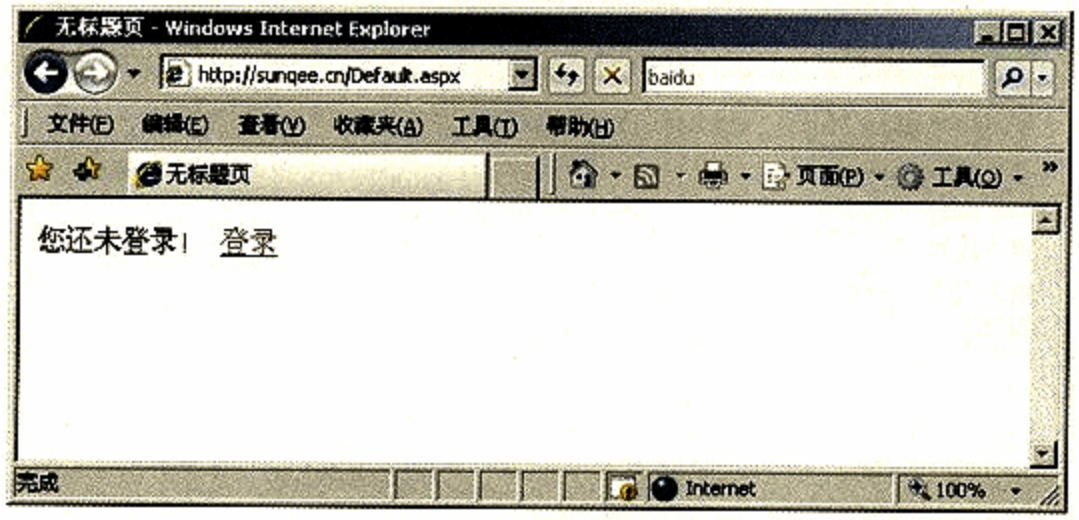
单击登录，浏览器将跳到 http://sunnyid.com/Login.aspx：



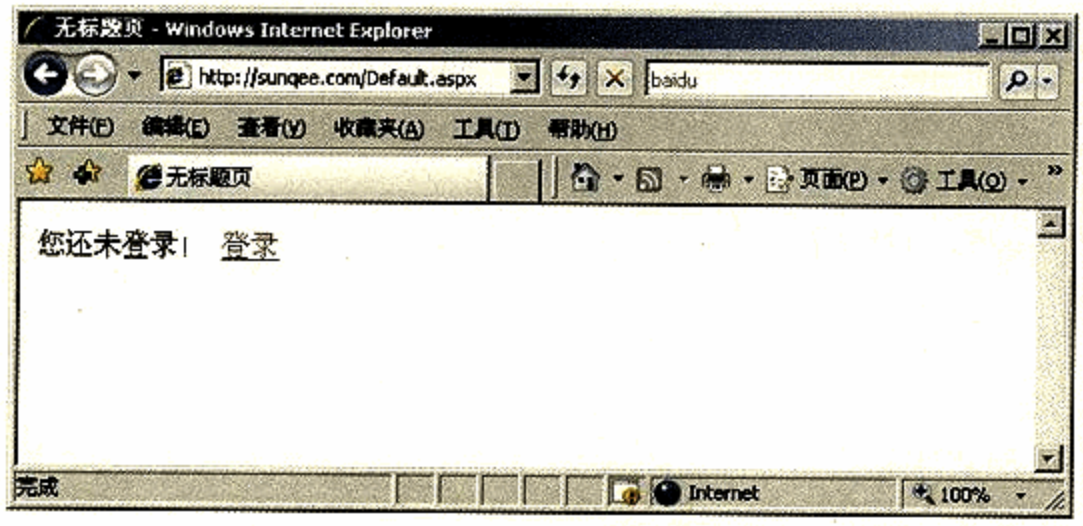
输入账号“LEADZEN”和密码“1234”之后单击“登录”，弹出如下图所示的界面。

成功实现了登录。然后马上访问另一个应用网站，输入http://sunqee.cn/Default.aspx之后，即可看到如下图所示的界面。

很好！新的应用网站已经识别了登录的用户。然后在此应用网站注销：



再回头访问原来的应用网站http://sunqee.com/Default.aspx，将看到：



非常棒！原来的网站自动识别到了用户的退出状态。

好了，到此为止，我们成功地实现了一套高效率的跨域名单点共享登录框架。

10
编程之禅

记得早些年看过一本叫《编程之禅》的小书，是美国的编程大师Geoffrey James写的。书中描写了一位传说中的编程大师，关于他的寓言故事写得非常神秘有趣。尽管有对东方文化存在不了解的地方，甚至将禅宗思想与神秘的西藏还有什么忍者柔到一起。但书里所讲述的内容都有深刻的寓意。那就是，除了理性的编程思维之外，还存在一种编程思想的感性升华，一种领悟编程本质的思想境界。这种思想境界就是编程之禅！

其实，编程之禅无处不在。把代码排列得更美观些；让标识符的命名更合理些；让代码更直观和简洁一些；排出一个Bug之后的喜悦；解决一个难题后的领悟。无时无刻不体现着编程之禅。禅本来就如人饮水，冷暖自知。各自的感悟不同，但都能得到自己的快乐。

编程之禅也是一种编程的态度，以一种轻松豁达的心态对待编程。无需将编程看着一项工作，而将其看作一种生活方式。领悟了禅机的程序员把程序看成自身思想的延续，他们总喜欢以一种天真快累的童心来看待现实世界。

编程之禅很难用言语来描述。编程之禅机就像生活经验一样不可以传授，只能由自己去经历，去体验，去思索，去领悟。编程之禅完全可以自行修炼，或渐悟，或顿悟，完全看自己的机缘。当然也可以求教禅师指点迷津。

禅师从来不会苦口婆心地教人该如何编程，如果人的内心还没有开悟，再多的外来灌输都是枉费心机。有人历经磨难苦苦思索而仍不得其法时，得遇禅师一语点化，或如当头棒喝间的猛然醒悟，或似醍醐灌顶般的豁然开朗，其喜悦的心情是不言而喻的。

禅师自己也还在修炼，因为他知道编程之禅是没有止境的。禅师总是虚心地向所有人学习，因为他知道每一个都有值得学习的地方。禅师从不怕犯错误，当他意识到自己错了的时候，会毫不犹豫地承认自己的错误。禅师也从不为自己的错误辩解，他会在放下错误后立即轻松前进。

禅师之所以能轻松快乐地编程，因为他放下了一切可以放下的东西。名利之争与禅师无关，禅师从来不把这些包袱背在身上。编程之禅从来不拘泥于某种固定的思维模式，禅师们也从来不把思维模式的包袱背在身上。禅师们甚至放下了自我，他们之所以能飞翔，是因为把自我看得很轻很轻。

我们在学习编程的过程中，的确需要来一点清新的禅风。清新的禅风可以净化我们的编程心灵，让我们看到一个更广阔的天地，思想也会更深邃和睿智。我们不会纠缠于无聊的争执中，不会局限于固定的思维模式，也不会受困于狭隘的私心。这道清新的禅风可以让我们自由自在地飞翔在广阔的编程天空中，享受着编程的快乐。

轻松一下，最后讲一个故事：

一位编程禅师应约去西湖边的咖啡馆，与一位程序员讨论编程的心得。程序员滔滔不绝地讲了许多最新技术以，及学习新技术中的快乐。又谈了许多实际应用中的问题，还有给自己带来的痛苦。言谈中既有沾沾自喜的夸耀，又有唠唠叨叨的抱怨。

禅师一直耐心听取，并不断点头微笑。程序员一边讲，禅师一边很客气地往程序员的咖啡杯里加了几勺子咖啡。程序员讲完之后，喝了一口咖啡马上吐出来。

“太苦了！”程序员说。

“苦吗？那就再加点儿糖。”禅师说。

然后，禅师又使劲地往杯子里加了许多糖，搅拌之后推给程序员。程序员尝了一小口。

“又太甜了！”程序员抱怨地说。

“哦，等我重新给你冲一杯。”禅师说着，就把这杯咖啡倒入湖中。然后俯身从湖里取了一杯湖水，放到程序员面前。

“你再试试。”禅师说。

“没有任何味道！”程序员喝了一口之后说。

禅师点了点头，微笑着说：“生命中的苦与甜取决于盛它的容积，就看你的心是一个杯还是一片湖！”

程序员高兴地说：“我得到了！谢谢大师指点！”

此时，禅师喝完了最后一滴咖啡，抿抿嘴说：“咖啡真是太香了！”

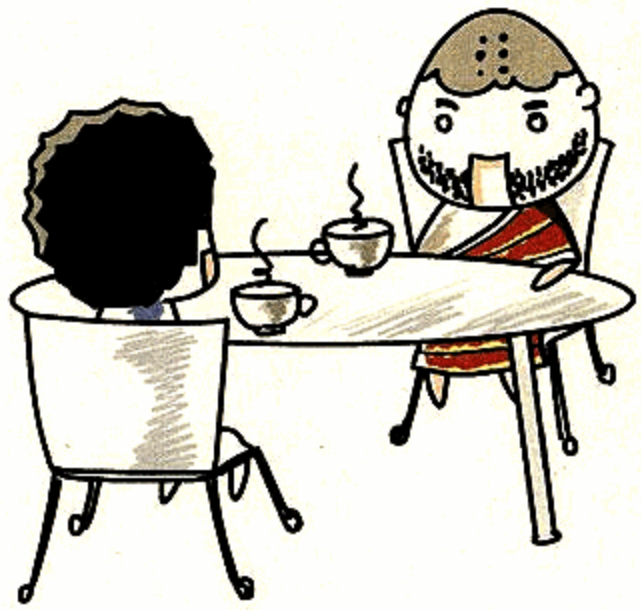
程序员这才回过神来问道：“我的咖啡呢？”

禅师狡笑着说：“有得必有失嘛！”

说完，两人都哈哈大笑起来。

那么，什么是编程之禅呢？

虚竹有节，真水无香……



后记

感谢您读完了本书，也谢谢各位读者的鼓励和支持，我也非常乐意接受各位老师的批评，我会尽力改正错误。我只是一名普通的程序员。当然，除了年纪大点儿之外，也没什么特别的地方。老程序员嘛，无非就是经验多一点儿，比年轻程序员接触的东西多一点。我也知道老程序员最大的毛病就是容易犯糊涂，所以请大家谅解。

2008年是多灾多难的一年。对我而言，2008年也是一个巨大的转折。其实，我根本没有想到会写成《悟透JavaScript》这本书。这本书源于我的一篇同名博客文章，这篇文章后来成为本书的第一部“JavaScript真经”。第二部“手谈JavaScript”和第三部“点化AJAX”是随后扩充编写的。

2008年春节休息期间，一边看着电视里关于大雪封路的新闻，一边思考明年如何让自己的小公司走出困境，同时规划未来的技术蓝图。我一直在www.cndev.org混的，碰巧那段时间CNDEV上不了，不能灌水。于是偶然溜到了博客园，还学习了老赵那些关于AJAX的精彩课程，由此知道了博客园的各位大师们。

我发现博客园的学术气氛很浓，文章的质量都很高，就也想在这个园子里混混，也好向各位讨教几招。于是，我申请了一个博客园的空间，并获得博客园的批准。建立这个博客之后，乘着春节的闲暇，零零星星搬了些以前的东西上去。

搬了些老文章之后，总觉得应该写篇新的。但是，写什么呢？前段时间正好学习了一些关于JavaScript的文章，心想，不妨写写自

己对JavaScript的理解好了。于是，一个人守在办公室的时候，就开始构思这篇文章。

忽悠忽悠技术方面的东西嘛，我也算是老油条了。想起多年前那个半吊子的《悟透Delphi》，也曾受到过朋友们的欢迎，于是就将这篇文章命名为《悟透JavaScript》。曾经辉煌一时的Delphi，如今已走下坡路了，回想起来这技术不过是过眼云烟。那么，沧海桑田之后，能否沉淀下来一些永恒的东西呢？可能没有人能回答这个问题，但我想应该留下一些具有思想深度的文章。只要在多年之后回味这些文章时，能找到一点点原来的感觉，也就足够了。

我特别喜欢看一些既讲解简单道理，又有浅显的实例来试试的文章。因为很多新东西我不懂，给我看源码呢，我又懒得去一句句地琢磨。大道理讲得太玄了，我一时半会也领悟不过来，名词太专业呢，我的英文又不是很好。只能努力学习，提高自己。但有些老师的文章的确能让我兴奋，因为我既能读懂它，又能领悟到其中的道理。这种文章我就很喜欢。

其实，我没有编写过多少JavaScript的代码，书架上也只有一本讲JavaScript方面的书籍来当手册查。因为最近开始学习AJAX技术，就在网上找了许多这方面的素材来提高一下自己的基本功。当学习过大量的关于AJAX和JavaScript的文章之后，慢慢对JavaScript有了感觉，好像可以出师了，也就大胆地开始了《悟透JavaScript》的写作。当然，是用我自己的风格。

这篇文章也是在空闲中零零星星拼凑而成的，前后用了近两周时间，难免会有错误。我想，只要我站在一个初学者的角度，把自己学习的感想写出来，就当自己写给自己看好了。当然，也想顺便显摆一下自己的厨艺。于是，我先用以前的老火汤打底，放些好素材后生火，烧开后立即加入现编现写的鲜活例子，起锅时来点古典思想的五香调料那么一浇，盛盘即可！

当你写进去的时候，你会发现原来抒发感想也会产生一种快感，于是就会越写越兴奋。当我写完第1章中的“原型扩展”之后，本想在世界不完美的感慨中收尾，但此时灵感一闪，又引发了要试一试的冲动。这一试，我才发现自己可能找到了另一种模拟类的表示方法，并迅速实现了一个看起来还不错的模型。我想，冥冥之中也许真的有观音菩萨在点化，于是，才有了“甘露模型”的那一段描写。

这时，我才深深理解了“教学相长”一词的真正含义。其实，当你把你学到的感悟表达出来的时候，你会发现自己还有很多没有理解的困惑，于是你必

须去理解之后才能表达出来。这又迫使你重新查阅资料，学习那些还没有理解的东西，甚至亲手实验。但当你再次理解了那些困惑之后，又会有一种升华的感觉，甚至会在顿悟的刹那间引发新的灵感。

这篇文章发出去之后，真没想到会在博客园引起小小的轰动。一时间好评不断，我也不断地结识一个又一个的新朋友。说实在，我还从来没有哪篇文章像这样受到过这么多朋友的认可和赏识，一时间有点受宠若惊，有点懵了。还好，凭借多年的修为，我还算知道自己是谁，假装谦虚地答复朋友们。

就在当天下午，我收到了电子工业出版社博文视点公司总经理郭立的邮件，希望我能将此文写成书之后出版。很快，我们通过网上交流确定了选题和主要事项，并由博文视点的编辑孙学瑛老师亲自负责本书各项事务。随后，在孙老师的指导和建议下，本书扩充了另外两个部分，并设计为配有精美漫画的技术图书。这中配有漫画的技术图书在出版界并不多见，也算是一个小小的尝试和突破。

也是因为《悟透JavaScript》这篇文章，我认识了淘宝网前端开发的领军人物赵泽欣（小马）。经小马介绍，我有幸到了淘宝网参观学习，随后又被推荐到阿里软件公司。如今，我已经在阿里软件公司工作，并受命打造阿里软件自己的前端开发框架。

阿里软件的技术氛围非常浓厚，而且每一位同事都充满激情。这里不但有日常的讨论交流，还有专业的培训和分享课程，能让我学到更多的东西。我在这里工作非常开心，也很庆幸能在这样的环境里搞自己喜爱的技术工作。

我一直提倡快乐编程，追求编程之禅的境界。我在阿里软件是最老的程序员，有时候都不好意思跟年轻人一起混。好在我还能放下老脸，放下那些虚无缥缈的东西，实实在在干活，快快乐乐编程。

要享受快乐编程，关键是要能放得下。放下技术的争执，放下已有的成就，放下曾经的挫折，放下一切可遇而不可求的东西。当然，有些东西也许很难放下。其实，放下并不等于放弃，放下只是一种乐观的心态。人嘛，难免会犯错误，不用太在乎，朋友指出了，就改过来就是。朋友对你的称赞，就当与大家同乐，朋友对你的批评，就当是自我反思嘛。

要感受编程之禅和快乐编程其实很简单，就看你是否能放下自我……