Web标准之道

博客园精华集

阿一　棕熊
李战　丁学　等 编著

人民邮电出版社
POSTS & TELECOM PRESS


CSS
JavaScript
HTML
CSS
JavaScript
HTML
博客园
cnblogs
CHEN
BG 学堂
良轩

人民邮电出版社

北京

图书在版编目（CIP）数据

博客园精华集：Web标准之道 / 阿一等编著. —北京：人民邮电出版社，2009.8
ISBN 978-7-115-20897-2

Ⅰ. 博… Ⅱ. 阿… Ⅲ. 主页制作—程序设计 Ⅳ. TP393.092

中国版本图书馆CIP数据核字（2009）第100523号

内 容 提 要

本书由博客园知名博主联手打造，涉及 Web 标准、HTML/CSS、JavaScript、SEO 优化等诸多领域，内容新颖，观点独特，妙语连珠。

本书并不是一本由代码和技巧堆积而成的集合，更多的是探讨了 Web 设计中若干理念和心得，其中多为经验之谈。

无论对于从事 Web 前端设计的人士，还是对于那些从事 Web 后端编程的技术人员，本书都极具参考价值。其中时常有颠覆传统之作，个中滋味，请读者自行品味。

博客园精华集——Web 标准之道

◆ 编　　著　阿　一　棕　熊　李　战　丁　学　等
责任编辑　刘　浩

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京隆昌伟业印刷有限公司印刷

◆ 开本：700×1000　1/16
印张：20
字数：356 千字　　2009 年 8 月第 1 版
印数：1－4 000 册　　2009 年 8 月北京第 1 次印刷

ISBN 978-7-115-20897-2/TP

定价：35.00 元

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154

序

2004 年 1 月，作为一名痴迷于技术的业余程序员，我在网上苦苦寻觅，却找不到真正属于程序员的网上家园。软件开发是创造性的劳动，交流和分享实在是太重要了，可能别人的一个想法或一点经验，就会让你事半功倍。由于是业余程序员，对这种交流就更加渴望，既然找不到，既然自己也一直在寻找事业的起点，那就自己建立一个吧。于是，博客园（cnblogs.com）就这样诞生了。博客代表每个程序员的小家，园即家园，代表着由小家构成的大家，希望博客园能服务好程序员，成为真正属于程序员的网上家园。

一个人，“一杆枪”（服务器是一台旧的台式机），博客园就这样开始自己的发展征程。这样一个简单的网站，却吸引了一些痴迷技术、喜欢交流和分享、正在寻找属于自己的网上家园的程序员，他们不嫌弃这里的简陋，他们选择在这里安家落户，仅仅是因为这里的纯净、专注、对程序员的真正理解和关心。博客园幸运地聚集起这些技术精英，这些技术精英痴迷于技术、激情于代码，更可贵的是他们喜欢交流与分享。就是这样一群人，每天为博客园贡献很多精彩内容；就是这样一群人，吸引着更多的这样一群人；就是这样一群人，不仅在分享中帮助了很多人，而且自己在分享过程中不断地成长。博客园成为记载他们成长过程的载体。

经历了 5 年多的发展，博客园记载了太多技术精英们的贡献，为了把这些精彩内容给更多人分享，博客园精华集编委会通过艰辛的努力，收集整理成为《博客园精华集》。因为无法通过《博客园精华集》的几本书承载所有的内容，只能让《博客园精华集》作为代表，希望能给读者一些启迪。同时，也通过《博客园精华集》表达我们最诚挚的感谢，感谢所有在博客园中作出过贡献的朋友！

杜勇

2009.05

前言

博客园是以.NET为主旋律的社区，然而在其“冰山一隅”，Web设计却又独领风骚，代表人物有鸟食轩、Cat Chen、阿一、丁学、爆牙齿、李战等，他们在Web标准、CSS、JavaScript上各有所长，在博客园留下了对Web设计领域的诸多美文佳作。

很奇怪棕熊的手指有老赵两个粗，却居然能做出那么“灵”的JS效果而不费吹灰之力；很惊讶阿一连普通话都说不利索，却陆续推出了“震惊于世”的播客系列《阿一Web标准学堂》；很佩服李战的八卦水准，居然能从JavaScript扯到和尚坐禅；而Cat Chen更是标新立异，大力鼓吹“欲练CSS必先宫IE”；最后，狂赞一下自己，因为原本想夸一夸上述这些世外高人，可是脑海里浮现的却是月圆之夜皇城之巅西门吹雪天外飞仙的yy画面。

是啊，难道大侠就不能秃头吗？更何况是只有几颗爆牙呢！

在众多美文之中，爆牙齿的《重构之美》系列作为扛鼎之作，全部选入本书之中。他的文中洋溢着对技术的自信以及对完美的追求，末了，如果你能仔细地品味，还能感受到一丝凄美和淡淡的无奈。所谓曲高和寡——啥意思呢，就是说，毛驴嗓门大。

Web设计之一：灵感

采菊东篱下，悠然见南山。

但凡一个人达到了上述境界，真可谓挥洒自如、点石成金——也就是所谓的灵感，亦或是创意。这不是靠学习所能得到的，而是需要对生活多多观察与接触。比如说，徜徉在西子湖畔，流连于淮海路边——自然与人文景观都是创作的源泉。时常看到一些让人心动的Web站点，一些点线色彩的简单搭配，就能使主题深入人心。

说了半天,就是为了推荐一篇因为时间关系而未收录的文章:《无敌博皮之乾坤大变色》（作者丁学，地址http://www.cnblogs.com/dingxue/archive/2009/01/04/1367732.html）。据小道消息透露，丁学为写此文，放弃元旦长假与老婆卿卿我我，而一门心思忘我研究，终成正果。可见代码之中自有颜如玉。

朋友啊，希望你也能够拿起画笔，绘制出自己的蓝图和白云。灵感就在一瞬间，须臾，刹那。

Web 设计之二：标准

曾几何时，Web 设计工作不如程序员。因为后者被认为是“真刀真枪”地编程，涉及了大量的业务逻辑；而前者，也就是美工，属于“鸡肋”的角色，可有可无，于是在薪资和地位上都是相对偏低的，于是，大量的美工转行做了程序员。

但是，事实证明，Web 设计并没有那么简单。

为什么同样的 HTML 代码在不同的浏览器中生成不一样的界面呢？

越来越多的程序员开始怀疑并抱怨工作的繁芜。于是终于到了 Web 设计师扬眉吐气的时候了。他们管这叫做 Web 标准并制定了一大坨共同遵守的规则，使用 CSS 统一排版，并不断地进行重构。于是，设计和开发又各安其职了。

这使我想到了一条哲理：随着生产力的提高，社会分工越来越细。

本书关于标准的讨论，占了一半的篇幅，风格迥异但殊途同归。

Web 设计之三：JS 原罪

成也萧何，败也萧何。

在软件世界中，脚本语言扮演的就是这样的角色，其中以 JavaScript 为代表。数起数落毁誉参半之后，随着 Ajax 的大行其道，这玩意儿居然摇身一变，也要封装设计模式了。弱弱地问，我们在实际的项目中真地需要么？我们的开发到底是方便了还是复杂了？带着这样的疑问，研读老赵的《挣脱浏览器的束缚》系列和鸟食轩的《在 JavaScript 面向对象编程中使用继承》系列，别有一番滋味在心头。毕竟，这两个老鸟都是这方面的 Geek。

预计，将来的编程世界是属于脚本的，但是在理念上可能会有很大的变化。也许你会指出脚本语言这样那样的缺点，比如说自上而下解析性能较差等诸多问题。现在并不急着下结论，让我们拭目以待，也许明天，我们的开发工具也都是 Web 的形式了。

大道至简

作为《博客园精华集》的第一个分册，本书并没有太多令人费解的概念，而是 50 余篇 Web 设计方面的经验之谈。我们并没有强迫大家必须接受这些观点，毕竟，Web 标准之道——这个道

字，是一种很玄妙的东西，没有固定的模式可以遵循。

本书话题轻松明快，适合各年龄阶段、各层次阶段的朋友在各种场合阅读。本来这本书就是供大家在茶余饭后消遣之用，如果能达到这个目的，那么也就不枉编者和读者辛苦一场了。

博客园精华集编委会

2009.5.12

编者及作者

丁学

常年活跃于国内各大技术社区和各类线下活动，10 多年的 Web 前端开发经验，对 Web 标准、SEO、用户体验等有深入研究。现就职于当当网，从事交易平台开发，并致力于高性能 Web 开发技术的研究与推广。博客地址 http://www.cnblogs.com/dingxue。

李蓓卿（网名棕熊）

资深前端开发工程师。现任某知名国际游戏公司首席前端架构师，负责指导前端架构、Web 用户交互、视觉设计与开发，并专注于 front-end globalization、front-end international integrating 等前端开发新领域的研究。博客地址 http://www.cnblogs.com/ruxpinsp1。

李战

阿里软件资深架构师，具有 20 年软件开发经验。在互联网软件开发，特别是 Web 标准及前端 JavaScript 技术方面有较高的造诣，《悟透 JavaScript》一书的作者。目前从事 SaaS 及云计算方面的研究工作。博客地址 http://www.cnblogs.com/leadzen/。

杨正祎（网名阿一）

从事网页前端开发 5 年，关注前端技术开发、用户体验等前端技术领域。写有系列博客《IE 的 Web 标准之道》，并有视频教程《阿一 Web 标准学堂》。主要关注的技术领域包括：前端开发技能、用户体验、用户行为研究、搜索引擎优化、Web 标准设计等。博客地址 http://www.cnblogs.com/JustinYoung。

爆牙齿

英来网（www.englive.cn）创始人兼 CEO。拥有 9 年前端设计、5 年 Web 标准应用部署经验，完成方欣内网软件、卡当网、爆米花网、海词网的大团队 Web 标准应用部署。博客地址 http://yuntian.cnblogs.com。

陈广琛（网名 Cat Chen）

现就职于 Baidu Web 前端开发部，微软 MVP，《Prototype and Scriptaculous in Action》与《Adobe AIR in Action》的译者。个人网站：http://catchen.biz，博客地址 http://cathsfz.cnblogs.com。

冯震球（网名 cloudgamer）

热爱 Web 前端技术开发，特别是 JavaScript。活跃于 CSDN 的 Web 开发板块，在 blog 中喜欢研究各种 js 特效。博客地址 http://www.cnblogs.com/cloudgamer。

侯建勋（网名 Lion）

CSDN 项目经理及产品运营经理，对为企业、组织提供项目管理方面的咨询感兴趣，最近关注高科技及成长型企业项目风险投资、资本运作及创业企业成长管理。博客地址 http://lion.net.cnblogs.com。

蒋飞勇（网名在路上）

先后从事 Oracle ERP、政府、制造业销售、售后方面软件项目的实施、管理工作。擅长于 Web 方面开发及项目管理，对.NET 方面的加密、算法有较深入的研究。博客地址 http://midea0978.cnblogs.com。

黎志（网名鸟食轩）

就职于微软公司在线广告平台部门，从事 Web 领域的开发和研究。精通 JavaScript、DHTML 及 CSS，设计并实现过具有复杂交互的大型在线商业软件。对软件界面设计、用户交互等领域有独到的见解。博客地址 http://birdshome.cnblogs.com。

梁逸晨

专注于.NET、javascript 应用和基于 Renderman 的图形学研究。开发有一个小型 Ajax 库和基于 XML 的数据库及一个 Feed 桌面客户端。曾从事平面设计、摄影、MAYA 渲染和照明、ASP 和 ASP.Net 程序员、WinForm 开发、.Net

分布式架构设计师等工作。个人网站 http://www.kvspas.com，博客地址 http://kvspas.cnblogs.com。

龙武（网名 LongWay）

热爱程序开发，具有多年 B/S 前后台开发经验，平时钻研于 Web 前端开发，尤其热衷于 JavaScript。博客地址 http://www.cnblogs.com/LongWay/。

史久锋（网名 netcorner）

曾工作于中国万网华东区技术研发部，现就职于上海久茂国际物流有限公司，主要是做一些物流相关软件开发工作。博客地址 http://www.cnblogs.com/netcorner。

谢晖（网名 MaxIE）

2003 年毕业后，一直在武汉工作，先后从事过互联网、教育、GIS、政府、物流、房地产相关领域的系统和项目开发、管理和实施。精通.NET、Flash、FMS、PHP、Flex，关注互联网行业、RIA、SEO 与移动通信业前景。博客地址 http://maxie.cnblogs.com。

赵劼（网名 Jeffrey Zhao，又名老赵）

乐于交流，热衷写博，好为人师。致力于技术和谐，希望能以绵薄之力改变业界不良风气。常陷于编程之美无法自拔。在 Ajax 如火如荼之际、工作闲暇之余涉猎浏览器的脚本编写，略有心得，撰文数十，反响强烈。博客地址 http://www.cnblogs.com/JeffreyZhao。

朱磊（网名 Truly）

对.Net Framework 核心、JavaScript 和 Ajax 技术的应用有深入研究。在 blog 上发表过许多技术文章，翻译过 MSDN 和网上的一些优秀技术文章，《ASP.NET 第一步》的作者之一。博客地址 http://truly.cnblogs.com。

曾飞鹏（网名 3zfp）

擅长 Microsoft .Net、SQL Server、JavaScript、Window

Mobile 等技术方向的政府电子政务及 GIS 应用开发，目前从事政府“数字城市”领域的项目管理及过程控制工作；致力于大型软件项目的工程控制和资源优化方面的实践和研究。博客地址 http://3zfp.cnblogs.com。

邹克文（网名 Kevin Zou）

Web 程序设计师，从事开发已有八年，热衷于寻找“银弹”，虽然至今无果，但是还将一直索求下去。除了电脑，最大的梦想就是有一天能够飞到地球以外，参透这美丽而又神秘的宇宙是由哪位伟大的系统设计师如何设计出来的。博客地址 http://tsoukw.cnblogs.com。

目录

第一部分　HTML/CSS

第二部分　Web 标准

第四部分　JavaScript

HTML/CSS

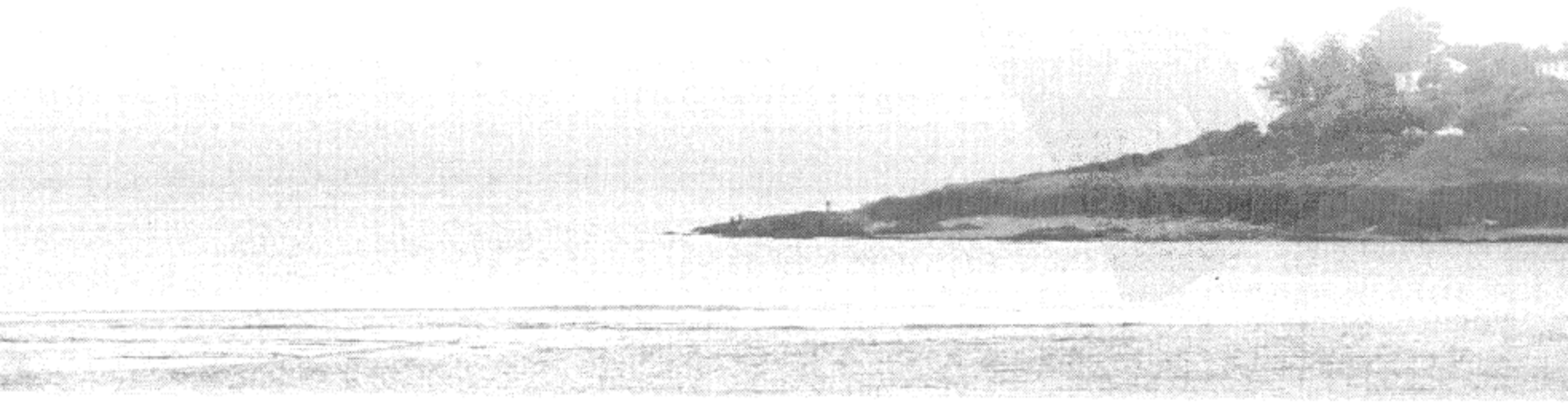
——最简单的HTML撑起了一片天，
而CSS让这片天开始绚烂

很难想象如果没有 HTML，Web 将如何生存，也很难想象如果没有持续更新的 HTML，Web 将如何发展。幸运的是我们并不需要想象，HTML 已经顺着时代的发展，产生了并一直发展着。至今都保持着一副朴素面孔的 HTML，却实实在在地改变着世界，在这本书最开始的地方，让我们展现给你这简单中的美妙。

Web，最重要的事情当然是传播信息给众人，那么最需要处理的事情，也就是“文字”，每天都和文字打交道，每天都要看很多的文字，而每一个 Web 开发者，每天要处理的文字也最多，怎么样的展现方式最吸引人？什么样的字体看着最舒服？看似简单的问题，内里却隐藏着很大的学问。在前 4 篇中，让我们随棕熊一起开始探索网页设计中字体应用的奥秘。

在第五篇和第六篇文章里，阿一给我们带来了前端开发中经常遇到的两个问题：各种<input>在 CSS 中的区分、让层覆盖<select>标签。相信多数人都曾经遇到这两个问题，并通过各种各样的方式去解决，那么，让我们听听阿一对这两个问题根源的探讨，并看看他给我们带来的解决方案。

在最后一篇文章里，同样是一个常用的技巧：图片自动缩放，我们经常使用 JavaScript 来完成这个任务，但我们打心眼儿里并不想为这件事写 JavaScript，我们甚至不想在任何地方使用 JavaScript 而期望 CSS 能够做更多的事，在这里，MaxIE 给了我们一个很小的示例，实现了兼容 IE 和 Firefox 的 CSS 实现方式，示例很小巧，但却给我们提供了一种思路：原来 CSS 还可以这样做……



谈谈网页设计中的字体应用（1）

——Font Set

作者：棕熊[http://www.cnblogs.com/ruxpinsp1]

Hihi，大家好~

最近有不少人都提及了网页上该如何选择字体的问题。问题虽然小，但是却是前端开发中的基本，因为目前的网页，还是以文字信息为主，而字体，作为文字表现形式的最重要参数之一，自然有着相当重要的地位。可惜字体的重要性在很长时间内并没有得到足够的重视。很多人对字体的概念还是停留在 font-family：宋体、Arial、Helvetica、serif 的阶段，却不明白为什么这样设置，这样设置是否合理。现在就让我说说字体的来龙去脉吧。

font-family

大家知道 CSS 规则中定义字体是通过 font-family 这条规则来实现的。仔细翻翻 CSS 的文档，却没有发现任何能指定某一个特定字体的规则。

想想十年前，可以随处看见类似于这样的代码：

```
<font face="Frankin Gothic Book">Lorem Ipsum</font>
```

几乎不会有人考虑到，Frankin Gothic Book 是一个 Windows only 的字体。在一台 Mac 上根本看不到 Frankin Gothic Book 字体的效果，系统因为找不到这种字体，就改用 Mac 的默认字体显示了。于是，网页的风格就和原来完全不一样，根本达不到 Frankin Gothic Book 的效果。于是 W3C 提出了 font set 的概念——将一系列近似的字体按照优先级顺序组成一个列表，浏览器从列表头部开始匹配，直到找到第一个可用的字体，并使用该字体进行显示。

比如上面这个例子，我们可以创建这样的一个 font set：

```
<span style='font-family: "Franklin Gothic Book","Lucida Grande";'>Lorem Ipsum</span>
```

我们来看看浏览器怎么呈现这段文字。

- Windows 下：浏览器从列表的第一个字体开始搜索——系统中存在 Frankin Gothic Book，所以使用 Frankin Gothic Book 字体显示。
- Mac 下：浏览器从列表的第一个字体开始搜索——系统中不存在 Frankin Gothic Book，搜索失败，继续搜索下一个字体——Lucida Grande。系统中存在 Lucida Grande 字体，终止搜索，并用 Lucida Grande 字体显示。

这样在 Mac 上，Mac 就能以与 Frankin Gothic Book 类似的 Lucida Grande 字体显示这段文字。

但是可能存在一台电脑，上面既没有 Frankin Gothic Book 字体，也没有 Lucida Grande 字体，那么它仍然无法正确显示上面的这段文字。于是开发人员不得不在这个字体列表中不断增加字体以适应各种系统，导致这个 font set 失去原本的“组织近似字体”的作用。于是 font set 中引入了“通用字体族”，也就是我们经常看见的 serif 和 sans-serif。我会在今后的文章中详细介绍这两个以及一些其他的通用字体族。在这里，我们可以简单地将它们理解为一种“在所有指定字体都失效的情况下，浏览器指定的一种最终的代用字体”。

比如我们再改进一下上面的那段示例文字：

```
    <span style='font-family: "Franklin Gothic Book","Lucida
Grande",sans-serif;'>
      Lorem Ipsum
</span>
```

我们再看看浏览器怎么呈现这段改进后的文字吧。

- Windows 下：浏览器从列表的第一个字体开始搜索——系统中存在 Frankin Gothic Book，使用 Frankin Gothic Book 字体显示。
- Mac 下：浏览器从列表的第一个字体开始搜索——系统中不存在 Frankin Gothic Book，搜索失败；继续搜索下一个字体——Lucida Grande，系统中存在 Lucida Grande 字体，终止搜索，并用 Lucida Grande 字体显示。
- 某系统：浏览器从列表的第一个字体开始搜索——系统中不存在 Frankin Gothic Book，搜索失败；继续搜索下一个字体——系统中也不存在 Lucida Grande 字体；继续搜索下一个字体——通用字体 sans-serif，浏览器应用它的默认 sans-serif 字体“Arial”来显示这段文字。

请注意两点。首先，通用字体族具体对应哪个字体是由浏览器决定的。上面例子中浏览器指定 Arial 为 sans-serif 字体，但完全有可能另一个浏览器指定 Helvetica 为它的 sans-serif 字体。具体哪个字体被最终应用是无法预期的。其次，通用字体族只是一种在 font set 中其他字体都无效时的代用方案。因此设计者应该尽可能给出齐全的 font set，以尽可能覆盖所有的系统，而不应该依赖于通用字体族。

类似于以下的两种写法都是错误的：

```
<span style="font-family:sans-serif;">Lorem Ipsum</span>
<span style="font-family:sans-serif,Arial;">Lorem Ipsum</span>
```

第一种写法的错误在于——它相当于根本没有指定字体，仍旧是交由浏览器选择字体——写了相当于没写。

第二种写法的错误在于顺序。因为通用字体族应该在一个 font set 中其他所有字体都失效时才起作用，因此将指定字体放在通用字体之后，会造成制定字体尚未匹配时就使用了通用字体。所以，务必要使通用字体处在 font set 中的最后一位。

另外，这里要说明两件事情。

首先，浏览器应用 font set 中哪个字体的规则虽然看上去很简单，但其实非常 trickish，我会在以后的文章中做出具体的说明。

其次，虽然字体的 CSS 规则名称叫 font-family，但它的实质是一个 font set，而不等于是印刷意义上的 font family。印刷上的 font family 是指一系列相同字样的不同强度组合，比如 Lucida Family（包括 Lucida Sans、Lucida Sans Typewriter、Lucida Console、Lucida Grande 等）和 Arial Family（Arial、Arial Black、Arial Rounded MT 等），但显然这些 font family 都不适合直接拿来当作一个 font set 来使用。

谈谈网页设计中的字体应用（2）

——serif 和 sans-serif

作者：棕熊[http://www.cnblogs.com/ruxpinsp1]

Howdy，大家好，又是我～

上一篇我们简单谈了一下 font set 和一些要注意的基本问题。今天我们继续字体这一话题，深入讲讲上次提到的“通用字体族”。首先是最常用的 serif 和 sans-serif 这两个通用字体族。

serif

serif 在印刷学上指衬线字体。为了理解衬线字体的概念，大家先看几个典型的衬线字体的例子。

单词 My 中的字母“M”上下方突出的短横线就是所谓的衬线。同样，y 的上方、K 的上下、i 和 n 的下方也都有衬线，所以这些字体都被称为衬线字体。但衬线字体并不一定都有衬线，比如上面例子中的 g、“汉”和“字”。事实上，只要满足末端加强原则的字体都是衬线字体。所谓的末端加强，就是使用衬线或粗细变化，使字体笔画的末端得到加强，以改善小号文字的可读性。比如上面例子中的 y 的下半部分，还有宋体的中文字符，都是采取加粗笔划的末端来达到末端加强的效果。除此之外，很多衬线字体还会采用加强竖向笔划（比如宋体中竖比横粗）和夸张字形（最明显的就是小写 g 这个字符了）等方法进一步改善它的可读性。

因为衬线字体的可读性非常好，所以它应用得最多的地方就是出版物或

者印刷品的正文内容等以大段文字作为表现形式的作品。

比较常见的衬线字体有 Georgia、Garamond、Times New Roman 和中文的宋体等。

sans-serif

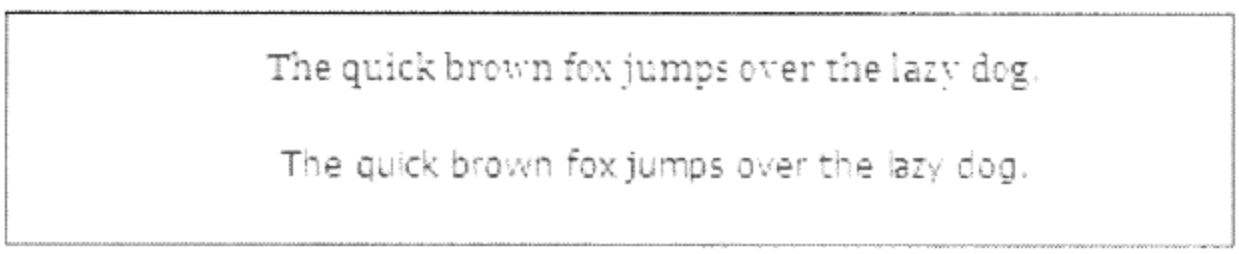
衬线字体以外的一切字体都是无衬线字体。sans- 这个前缀其实是法语，所以比较标准的发音是/san/，而不是/sans/，它的意思是“没有”，所以 sans-serif 就是无衬线字体。

无衬线字体比较圆滑，线条一般粗细均匀。比较适合用作艺术字、标题等。因为无衬线字体通常粗细比较均匀，所以在小字体显示的时候，可读性会降低，容易引起视觉疲劳。

常见的无衬线字体有 Trebuchet MS、Tahoma、Verdana、Arial、Helvetica、中文的幼圆和隶书等。

什么时候用 serif？什么时候用 sans-serif？

从上面的介绍中我们可以知道，衬线字体之所以被设计出来，就是为了用作正文内容的。大家可以随手抄起一张报纸，看看上面的文章是不是宋体，如果手头有外语读物的话，也可以翻来看一下，正文都是衬线字体。同样大小的衬线字体比无衬线字体容易阅读。



然后大家可以把报纸翻到头版头条，标题一般都会是各种粗细一致的综艺体或者是中黑体。英文报纸的标题大多也是无衬线的。这就是应用它们的基本原则。

大家可以看很多网站的正文内容恰恰是无衬线的 Tahoma、Verdana 和 Arial 等字体。中文网站可能因为字体的局限性，使用宋体的居多，但查看它们的样式表，就会发现候补字体也大多是无衬线的。这样是不是不好呢？

当然不是。

衬线字体的可读性其实仅仅体现在小字体上。大家可以拿出报纸，和你显示器上的文字比较一下，你会发现，报纸上的文字比显示器上的文字整整小一圈。实际上，新民晚报上通常大小的宋体文字，在点距为 0.25mm 的高质量液晶显示器上，大小大约只相当于 10px ~ 11px 的显示字符；在普通的液晶显示器（点距一般为 0.28mm）上，甚至可能只相当于 8px ~ 10px 的显示字符。

这个就是 print media 和 screen media 的最大区别。印刷业为了节约成本，会尽可能地在保证可读的情况下，把文字印小。显示器不存在这样的成本，因此可以显示比较大的文字。在文字足够大的情况下，无衬线字体也是同样可读的，而且，因为无衬线字体通常有艺术性，因此在显示器上显示通常比较赏心悦目，而且无衬线字体种类比衬线字体多得多，因此选择余地也很大，所以大家尽可以放心去使用。但是以下原则必须保证：凡是使用无衬线字体的，必须保证其在正文内容中的可读性，否则使用衬线字体。换而言之，如果你要使用无衬线字体显示网页的正文内容，那么必须把它的 font-size 设得足够大，以保证用户能轻易阅读。

至于具体将 font-size 设多大，是因字体而异的。12px 对于 Verdana 字体来说已经完全足够，但是要能轻易地阅读隶书，可能需要 24px 以上才行。

对于 11px 以下的英文字体，推荐使用衬线字体。至于中文，因为显示器的硬件限制，不论是什么字体，都不推荐使用 11px 以下的 font-size 来显示。

其他的通用字体族

印刷学中，除了 serif 和 sans-serif 之外，通常还有 monospace 等宽字体、scripts 手写体（比如花体）、blackletter 铅字体（也叫 gothic 哥特体。严格地说，很多常用的 serif 字体其实是 gothic 字体）、ornamental 装饰体（那些在文字笔划上或者周围有装饰花纹的字体。很多中世纪书籍上很常见。如果脑残体真的成了字体，那么应该可以算装饰体吧……）和 symbol 符号字体（比如有名的 wedding123……）。

不过CSS对通用字体族的定义有点不一样。除了 serif 和 sans-serif 之外，CSS 还允许以下几个通用字体族。

- monospace 等宽字体。所谓的等宽字体，是指每个字符宽度都一致的字体，一个著名的例子就是 Courier New 字体。因为字符宽度一致，所以特别容易对齐，能快速精确地定位到某行某列，因此经常用来显示代码。

要注意的是，等宽字体同时也可以是衬线（或者非衬线）字体，比如 Courier New 这个字体也可以看作是一个 serif（严格地说是 gothic）字体。

- cursive 书写体：相当于印刷学中的手写体。中文的华文行草就是这样的一个字体。

- fantasy 梦幻体：相当于印刷学中的装饰体。非常少见的一种字体，基本没有参考价值。

要注意的是，CSS 中不支持 symbol 字体族，使用 symbol 类的字体请用图片。

一些你不知道的事情

中文的黑体其实是衬线字体

看看下面这个图：

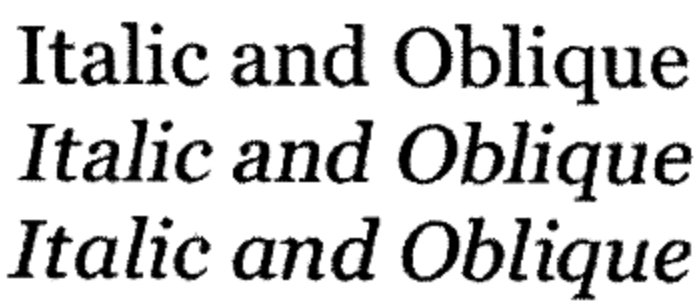
其实，黑体是经过末端加强的，所以很多印刷品的正文也会使用黑体。像这种使用温和的末端加强，笔划粗细大致一致的字体，其实也可以被称为 petit-serif/小衬线体（那些类似于宋体一样有显著末端加强，并且笔划粗细有明显区别的，通常称为 slab-serif/雕版衬线体）。

只是很遗憾，因为诸多的硬件原因，在显示器上实际显示黑体时，大家还是可以把它看作一个无衬线字体。

Italic 不是斜体

斜体是 oblique。Italic 顾名思义，是意大利体。Italic 是一种书写方式（calligraphy script），而 oblique 是一种印刷样式，两者是不同的东西。中学英语习字册教授的书写方式就是意大利体。除了意大利体外，比较流行的书写方式还有法兰西体（就是传说中的花体字，正名是 French Script）、哥特体和亚伯拉罕体等。

很多考究的字体都会为意大利体定制一套特殊的字体，而不是简单地显示成斜体。比如下面的图片里，三行文字都是 Georgia 字体。第一行普通；第二行是 oblique，也就是斜体；第三行才是真正的 italic 意大利体。



大家仔细看第三行的 a, l, i, e 等字母，很明显地看出区别了吧。实际上，Georgia Italic 和 Georgia 在系统内是两个不同的字体文件。当我们指定

font-style: italic 的时候，系统就会自动搜寻是不是存在 Georgia Italic 这个字体，并尝试使用这个字体来显示文字内容。

按理说当我们用 font-style: oblique 指定字体样式时，浏览器不应该去寻找 Georgia Italic 这个字体，而直接将 Georgia 字体倾斜显示，所以理论上应该得到图中第二行文字的效果。可惜，连 W3C 在 CSS 规范中，自己的参考实现也说“如果 UA 不能正确显示 italic 和 oblique，可以使用 italic 来代替 oblique 显示”，所以几乎没有浏览器实现区分 italic 和 oblique。哪怕设置的 font-style 是 oblique，你也会发现，浏览器显示的也还是 italic 。

本篇就到这里了。下一讲我会谈谈如何构建一个合理的 font-family，并推荐几个实用的字体组合给大家。

谈谈网页设计中的字体应用（3）

——实战应用篇·上

作者：棕熊[http://www.cnblogs.com/ruxpinsp1]

大家看过 font set 和一些要注意的基本问题以及通用字体族两篇文章后，应该对字体基本有了一些了解。现在我们开始把这些知识都应用到实战中吧！

规范中 font-family 的解释方式

我们定义下面这个字体表：

```
font-family: "Comic Sans MS", "幼圆", "黑体", sans-serif;
```

按照 W3C 的规范，浏览器在使用这个 font-family 显示一个字符时，首先应该寻找 Comic Sans MS 字体。如果找不到 Comic Sans MS 字体，那么顺序搜寻下一个字体，即幼圆字体。如果找到 Comic Sans MS 这个字体，那么浏览器会在 Comic Sans MS 字体中寻找这个字符。如果找到这个字符，就使用 Comic Sans MS 字体来显示这个字符。如果没有找到这个字符，或者这个字符对应一个缺字符（缺字符是字体文件中的一种特殊字符，用来表示字体文件中没有这个字符。通常就是显示一个方块），那么就要到下一个字体，也就是幼圆体中继续搜寻这个字符。如此搜索整个字体表，直到搜索到这个字符为止。如果在通用字体，也就是这里的 sans-serif 字体中也找不到这个字符的话，那么浏览器就应该显示该字体的缺字符。

所以，如果有下面这句话：

所以我说：“这也是没办法的，it ain't going nowhere”。

那么，在一个正常的 Windows XP 系统上，所有中文字符都会被显示为幼圆，英文字符都被显示为 Comic Sans MS 字体。比如“说”字，浏览器先搜索 Comic Sans MS 字体，得到一个缺字符，于是搜索幼圆。系统中存在幼圆字体，于是终止搜索，将“说”字显示为幼圆字体。对于英文字符 i，浏览

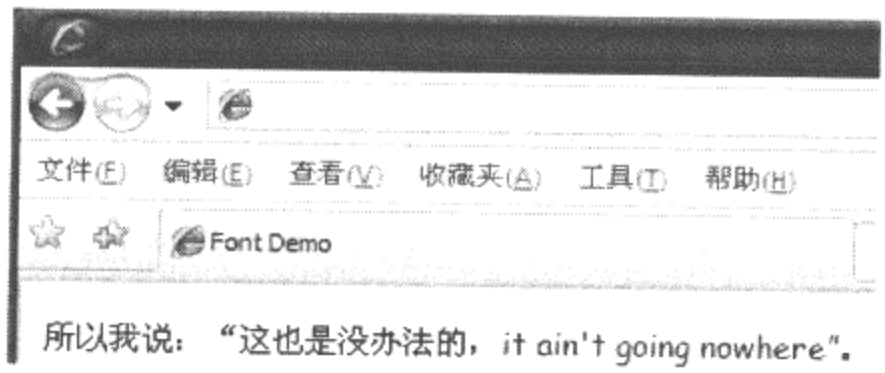
器在 Comic Sans MS 这个字体中就能找到这个字符，于是就用 Comic Sans MS 显示 i 这个字符。

另外，双引号——“”，这两个字符其实在 Comic Sans MS 中也有。所以浏览器会用 Comic Sans MS 中的双引号来显示。

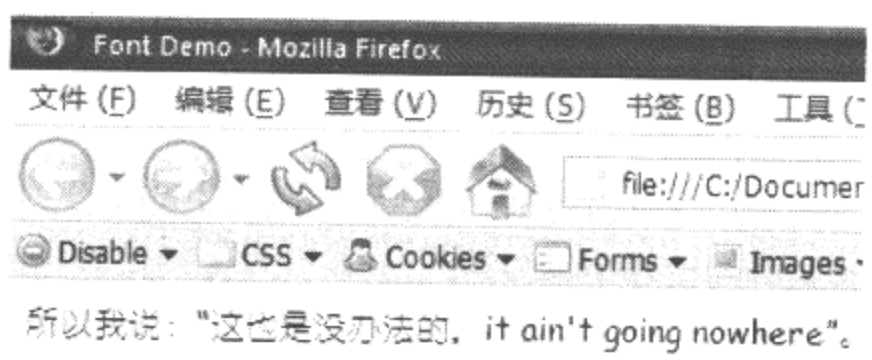
事实上呢?

大家来看看截图。

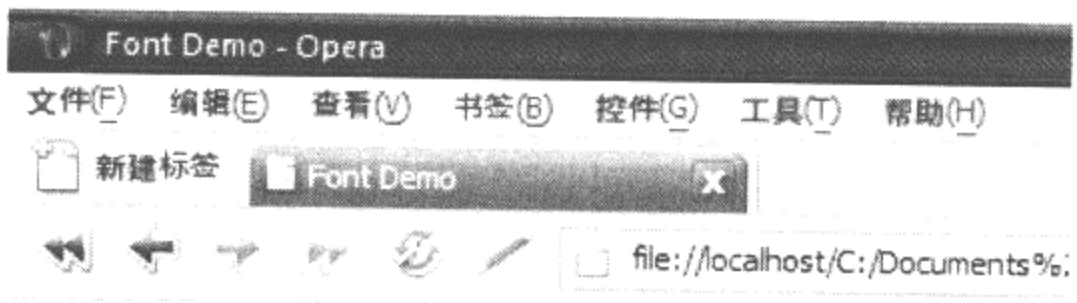
Internet Explorer 7



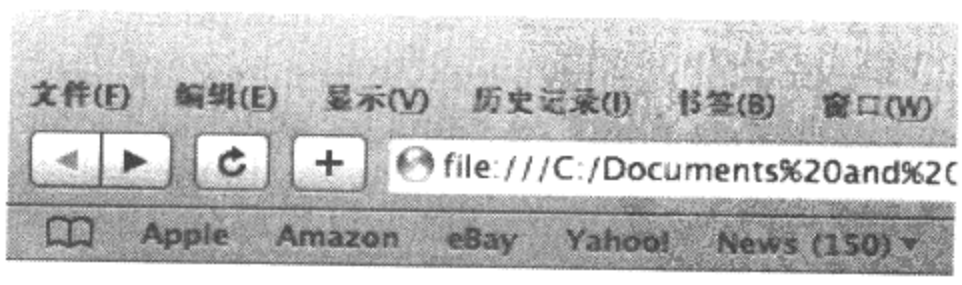
Firefox 2



Opera 9



Safari 3.1 Windows



……简直是一个浏览器一个样子，这样还叫人怎么正经干活嘛。

仔细看看，其实 Firefox 和 Safari 显示的还算靠谱，在这个例子里，显示的都正确。IE 和 Opera 都没有能用正确的字体显示中文字符。因为在 Comic Sans MS 搜索失效后，理应搜索幼圆字体。但不知道什么原因，IE 和 Opera 都没有顺序搜索下一个字体，甚至也没有搜索后面的黑体和 sans-serif，而是直接跳到系统默认字体了——请注意，是系统默认字体，因为我已经在 Opera 里把 sans-serif 设成了雅黑，如果 Opera 还有良心搜索 sans-serif 的话，还是应该用雅黑显示中文字符的。而且，Comic Sans MS 中明明存在的双引号，也没能在 Opera 中得到正确的显示。这是什么号称最完美支持 CSS 的浏览器？简直浪得虚名。

IE 7 起码还好些，至少认了和英文字符直接相连的双引号。但是除此之外，也算是完败。

另外大家也不要认为 Safari 很完美——某些版本的 Safari 3 for Windows 在第一个字体中寻找不到中文字符时，它干脆就显示了那个字体的缺字符，于是所有的中文网页变成了整屏的□□□□□□□□，根本无法阅读。经本人和其他许多发现这个 Bug 的人多次向 Apple 交涉，他们才最终修正了这个 bug。

至于 Firefox，其实也不完美，因为 Firefox 不支持字体别名。于是幼圆你只能写成“幼圆”，黑体你只能写成“黑体”，而不能用他们在系统中的正式字体名称——YouYuan 和 SimHei。

对于浏览器为什么会产生这么多五花八门的奇怪渲染，偶也不知道，估计只有问这些浏览器的开发人员了。

解决方案

因为主流浏览器在中文显示中实在无法统一，因此，解决上面这个问题也只能采取折衷和妥协的方案。至于如何折衷，那么要看你到底想要保证英文字符的显示效果，还是中文字符的显示效果。

如果你希望保证中文的显示效果，必须把想要显示的中文字体放在 font-family 定义的第一位。比如上面例子里的样式定义，可以写成：

```
font-family: "幼圆", "Comic Sans MS", sans-serif;
```

这样就可以保证所有中文字符都显示为幼圆。至于为什么 IE 和 Opera 又认了在 font-family 首位的幼圆，这个也不要问偶，总之它们就是认了。

这样做的缺点也是显而易见的。一般中文字体中都会包含英文字符，比如上面的幼圆。所以网页中的英文字符也会优先应用这些中文字体来显示。而中文字体中的英文字符，通常都不怎么好看。比如还是这个幼圆，里面的

英文字符和宋体一模一样，根本和幼圆中的中文字符不搭调。于是中英混排的文章就极其难看。而且很遗憾，一般网页上，中英混排的情况还是很多的，比如用户名、日期时间、URL，等等。

另外，这个方案也不能从根本上解决浏览器对中文字符支持的缺陷。比如这种情况：有人非常喜欢黑体字的效果，所以想用微软雅黑来显示你的网页，但是考虑到只有 Windows Vista 才有微软雅黑字体，所以打算在没有雅黑的电脑上用黑体来显示文字，于是他写了这么个样式规则：

```
font-family: "微软雅黑", "黑体", sans-serif;
```

但实际测试下来，他会发现，即使第一个字体设置成了中文字体，在这个字体缺失的情况下，IE 和 Opera 还是不会使用第二位的黑体，而继续它们自己的莫名其妙的规则，使用了系统默认字体——宋体。这显然还是不能满足设置 font-family 属性的初衷。

第二个方案是，仍旧按照 CSS 标准的解释方式来写 font-family，但是在 font-size 上做些手脚，只用 12px、14px、16px 等稳扎稳打的字体大小。这样做最大的好处是能优先用最合适的字体显示英文字符。至于中文字符，XP 的宋体也好，Vista 的雅黑也好，OS 的新宋体也好，在上面几个字体大小下的显示都不算难看。何况中文字体的选择范围本来就比较小，无外乎也就是那几个系统默认字体，因此自然也就凑合了。个人比较倾向使用这种方案。

至于具体选用哪种方案，还需要大家根据实际情况斟酌而定。

谈谈网页设计中的字体应用（4）

——实战应用篇·下

作者：棕熊[http://www.cnblogs.com/ruxpinsp1]

上次我讲了在实际应用 font-family 时会遇到浏览器兼容性问题。这次我要从操作系统方面来讲如何安排字体族。另外，由于中文字体的选择范围实在太小，所以本章中涉及的内容主要以西文字体为主，比较适合上一章中的“方案二”。

不同操作系统的常用字体

如何让你的字体在任何系统、任何电脑上都看起来一致？

原则很简单：尽可能使用所有操作系统都存在的字体。虽然听起来简单，但是其实还是很 tricky 的一件事情。为此，你首先需要了解常用的操作系统的字体。

下面我会列出一些除了 Windows 以外的常用操作的默认字体。windows 么……想来大家应该已经很熟悉了。

Mac OS X 中的常用字体

一个典型安装的 Mac OS X 10.4 会包含以下常用西文字体。

sans-serif	serif	monospace	sans-serif	serif	monospace
Helvetica	Times	Courier	Gill Sans		
Arial	Times New Roman	Courier New	Impact		
Arial Narrow	Georgia		Trebuchet MS		
Arial Black			Verdana		
Comic Sans MS			Lucida Grande		

典型的 Linux 字体

典型的 Linux 只有 kernel，所以字体要自己安装。

既然这样，自然无法正确预测使用 Linux 的用户装了啥字体。不过好在大家都会装一些常用的字体，因此不会有什么大问题。

比较各个操作系统的字体，我们会发现——

其实，Windows 常用的字体在其他操作系统中都有，甚至很多人认为 Windows only 的 Arial 字体也不例外。

不少设计师都认为 Arial 是个不典雅的字体，所以希望在 Mac 上能用更经典的 Helvetica 字体来代替，于是产生了这种代码：

```
font-family: Arial, Helvetica, sans-serif;
```

但是因为 Mac OS 其实也有 Arial 字体，所以永远都只会显示 Arial。

其实这种问题，只要稍加修改就 OK 了：

```
font-family: Helvetica, Arial, sans-serif;
```

然而，事情往往不是这么简单的。比如上面的 Mac OS X 字体表中，有 Lucida Grande 字体。照理说这个字体是 Mac only 的，所以大家理应可以放心地这么写：

```
font-family: "Lucida Grande", Arial, sans-serif;
```

那么 Mac 用户可以看到 Lucida Grande，而 PC 用户可以看到 Arial 字体。多好的应用典范。

但是实际上呢，不少 PC 用户居然看到了乱码，而不是 Arial 字体。

怎么回事呢？因为市面上有不少字体下载网站，而上面就有那个 Lucida Grande 下载。可惜这个广为流传的 Lucida Grande 是个 rip 版，而且 rip 的时候有缺陷，导致所有换行字符都会显示成一个乱码。

——囧大了

不要说这种事情只会在乱装英文字体的用户上发生哟。能在 XP 上显示微软雅黑的，不都是 rip 版的——那个网上广为流传的版本，也存在着类似缺陷，只不过不至于严重到产生乱码而已。所以在选择字体时需要注意一下。

常用西文字体介绍

Tahoma 16px Tahoma 14px Tahoma 12px

Tahoma 是我本人比较喜欢的一种非衬线字体。首先几乎所有的系统都默认安装了这个字体，所以不会存在兼容性问题，其次，这个字体也比较均衡，

显示段落也不错。

Verdana 16px Verdana 14px Verdana 12px

说老实话，Verdana 太宽了，不适合中英文混排。很多时候 Verdana 的一个字母都要比同样 size 的中文字符宽。国外设计师喜欢用 Verdana，很多时候是因为 Verdana 11px 以下的小字效果的确十分理想，但是国内很多设计师想也不想就照搬过来，并用在 12px 乃至 14px 的布局上，导致本来就局促的空间更显紧张，所以不推荐作为 font-family 打头阵的字体。

如果要使用 Verdana 字体的话，就一定要考虑它和一般系统 default 的 sans-serif 字体之间的大小差距。不论和 Helvetica 或者 Arial 比起来，Verdana 都大得多了。不过好在几乎所有的系统也都会默认安装这个字体。

Trebuchet MS 16px Trebuchet MS 14px Trebuchet MS 12px

Trebuchet MS 是个很多人都会忽视的字体。其实我个人也比较欣赏这个字体。与其使用 Verdana，还不如用这个线条更圆润的字体来替代。它对各种操作系统也有很好的支持。

缺点和 Verdana 一样，因为过宽，而不适合用于中英文混排。也要注意和 default sans-serif font 宽度差距的问题。

因为考虑到有些 Linux 系统可能不会安装这个字体，所以如果要用在 font-family 的开头话，可以使用 Verdana 做后续字体。

Arial 16px Arial 14px Arial 12px

Windows 操作系统默认的 sans-serif 字体。没啥好说的，永远都不会用到的默认字体。

Helvetica 16px Helvetica 14px Helvetica 12px

为啥同样是默认字体，Helvetica 就这么典雅呢？哪怕就是用在 font-family 的开头也是能独挡一面的。

另外，这里有个 Helvetica 和 Arial 打架的 flash 游戏～ 像超级玛利一样踩 Arial 字符就可以了。没有 Helvetica 字体的人还可以顺便看一下两个字体的具体区别。

Georgia 16px Georgia 14px Georgia 12px

我最喜欢用的 serif 字体。不仅很适合做正文，也适合做标题。尤其是意大利体的 Georgia Italic 更是魅力难挡。缺点仍旧是不适宜和汉字混排，因为 Georgia 的衬线哪怕对于宋体来说也太重了，所以看上去硬邦邦的。

Times New Roman 16px Times New Roman 14px Times New Roman 12px

Windows 的默认 serif 字体。没啥好说的，西文字体的元老了，很多字体，比如大家都熟悉的 Courier New 都是从 Times New Roman 派生出来的。

不过现在印刷业都很少用这个字体了，更多的是在用它的后代——Times Europa 和 Times Europa Office。

在具体的网页字体应用上，要注意同样字号的 Times New Roman 比普通字体小得多，所以一定要考虑字体大小的变化。

Courier New 16px Courier New 14px Courier New 12px

常用的等宽字体之一。其实等宽字体的选择面比较小，所以基本上要兼容所有系统，也就只能选这个字体了。

不过还好，等宽字体通常都是在写代码的时候有用，所以只要等宽就没什么大问题。类似于 Lucida Sans Typewriter、Lucida Console、Monaco 之类的字体也都很好用。

综上所述，总结几套实用而简单的 font-family

font-family: Tahoma, Helvetica, Arial, sans-serif;
Tahoma 系的中性字体。推荐使用在 13px 以上的环境中。

font-family: Trebuchet MS, Verdana, Helvetica, Arial, sans-serif;
Verdana 系的宽扁字体。推荐在 11px 以下环境中使用。

font-family: Geogia, Times New Roman, Times, serif;
衬线字体的不二之选。

font-family: Lucida Console, Monaco, Courier New, mono, monospace;
一系列等宽字体。写代码很好用。另外，如果觉得 Lucida Console 太宽的话，可以换成比较窄的 Lucida Sans Typewriter。老赵 blog[①]上的代码块使用的就是 Lucida Sans Typewriter 哟。

你知道吗?

字体的别名

系统中的字体是允许有多种别名形式存在的。比如，在 Windows 下，Georgia 也可以用 Georgia MS 来命名，它们其实是同一种字体。宋体的正式

[①] http://www.cnblogs.com/JeffreyZhao

名称是 SimSon，而“宋体”只是它的别名。

按照规范，浏览器应该能自动识别字体的别名，并映射到正确的字体文件。比如，font-famliy: SimSon 和 font-family: "宋体" 应该具有等价的效果。可惜，似乎很多浏览器都不能正确执行前一条定义。

什么时候在字体名称前面加引号

大家来看这个字体样式定义：

```
font-family: Times New Roman, 宋体, serif;
```

很多人都会说，这个样式写法是错的，因为 Times New Roman 和宋体都应该用引号括起来，像下面这样：

```
font-family: "Times New Roman", "宋体", serif;
```

实际上呢，上面两种写法都是对的。和很多人想象中的不一样，字体名称外面的引号其实并非必须的。那么加引号和不加引号有什么区别呢？

其实最大的不同在于对字体名称中空白字符（如空格、制表符）的解释。

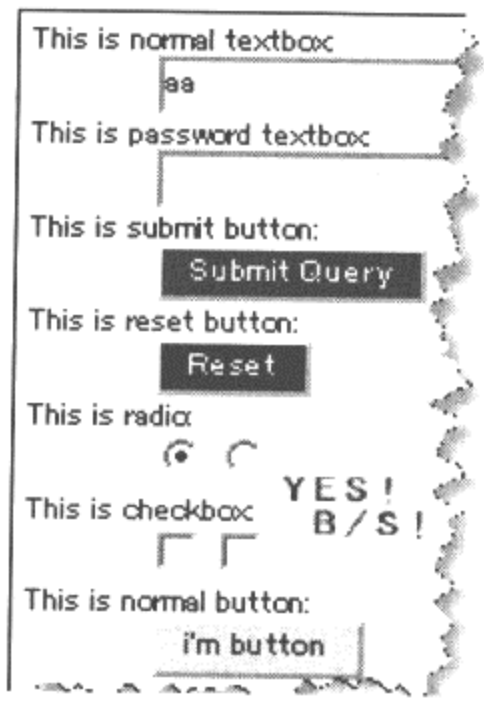
不加引号的时候，浏览器对于字体名称中空白字符的解释应该和 XML 中一样，即忽略字体名称左右的空白字符，并且单词中间的空白字符被解释为一个空格。比如 font-family: Times New Roman, serif; 会被解释成 font-family: Times New Roman, serif;

加引号的时候，浏览器必须保留引号内所有的空白字符。如果写成 font-family: "Times New Roman"; 那么浏览器不会显示 Times New Roman 字体，而是搜索一个叫做“Times New Roman”的字体。

至于“宋体”这样的字体名称，因为中间没有空白字符，因此完全没有必要加引号。但是考虑到并非所有的操作系统都有汉字支持，并且并非所有的程序员都会注意 CSS 文件的正确编码问题，所以为保险起见，一般会加上引号。当然，解决这种问题的最好方法是使用别名。比如宋体，其实应该写成 SimSon，这样哪怕浏览者的系统不支持中文，并且这个 CSS 文件被错误地编码成了 GB2132 也没问题，浏览器还是知道这是宋体，并且做出正确的字体搜索。可惜，不是所有浏览器都支持就是了。

让 CSS 区分各种各样的<input>

作者：阿一（杨正祎）[http://www.cnblogs.com/JustinYoung]



当你看到<input>这个 html 标签的时候，你会想到什么？一个文本框？一个按钮？一个单选框？一个复选框？……对，对，对，它们都对。也许你可能想不到，这个小小的 input 竟然可以创造出 10 个不同的东西，下面是个列表，看看哪些是你没有想到的。

- <input type="text" /> 文本框
- <input type="password" /> 密码框
- <input type="submit" /> 提交按钮
- <input type="reset" /> 重置按钮
- <input type="radio" /> 单选框
- <input type="checkbox" /> 复选框
- <input type="button" /> 普通按钮
- <input type="file" /> 文件选择控件
- <input type="hidden" /> 隐藏框
- <input type="image" /> 图片按钮

所以你可能会说，input 真是一个伟大的东西，竟然这么有“搞头”，但是当你真正在项目中试图给不同的控件设置不同的样式时，你就会发现，input 真的可以把“你的头搞大”。我不知道为什么当初要给 input 赋予那么多身份，但是，他的“N 重身份”给网站设计者的确带来了不少的麻烦。好在，劳动人民是伟大的，解决问题的办法还是有滴～，虽然它们都有各自致命的缺点。解决方法大致归纳一下，列表如下：

- 用 CSS 的 expression 判断表达式。
- 用 CSS 中的 type 选择器。
- 用 JavaScript 脚本实现。
- 如果你用 Microsoft Visual Studio 2005 或者后续版本开发项目，恭喜，你还可以使用 skin。

下面就来讲解一下各个办法的详细实现和它们的优缺点。

1：用 CSS 的 expression 判断表达式

实现代码参考：

```
<!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> diffInput2 </title>
    <meta name="Author" content="JustinYoung"/>
    <meta name="Keywords" content=""/>
    <meta name="Description" content=""/>
    <meta http-equiv="Content-Type" content="text/html; charset=
utf-8"/>
    <style type="text/CSS">
    input{background-color:expression(this.type=="text"?'#FFC':'');}
    </style>
</head>
<body>
    <dl>
        This is normal textbox:<input type="text" name=""><br/>
        This is normal button:<input type="button" value="i'm button">
    </dl>
</body>
</html>
```

优点：简单，轻量级。

缺点：FireFox 不支持 expression 判断表达式。致命的是只能区分出一个（例子中就只能区分出 text 文本框），不要试图设置多个，下面的会将上面的覆盖掉。

2：用 CSS 中的 type 选择器

实现参考代码：

```
<!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> diffInput2 </title>
    <meta name="Author" content="JustinYoung"/>
    <meta name="Keywords" content=""/>
```

```
<meta name="Description" content=""/>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/CSS">
input[type="text"]
{
    background-color:#FFC;
}

input[type="password"]
{
    background-image:url(BG.gif);
}

input[type="submit"]
{
    background-color:blue;
    color:white;
}

input[type="reset"]
{
    background-color:navy;
    color:white;
}

input[type="radio"]
{
    /*In FF,Some radio style like background-color not been
      supported*/
    margin:10px;
}

input[type="checkbox"]
{
    /*In FF,Some checkbox style like background-color not been
      supported*/
    margin:10px;
}

input[type="button"]
{
    background-color:lightblue;
}
```

```
    </style>
</head>

<body>
    This is normal textbox: <input type="text" name=""><br/>
    This is password textbox:<input type="password" name=""><br/>
    This is submit button:<input type="submit"><br/>
    This is reset button:<input type="reset"><br/>
    This is radio:<input type="radio" name="ground1">
                  <input type="radio" name="ground1"><br/>
    This is checkbox:<input type="checkbox" name="ground2">
                     <input type="checkbox"name="ground2"><br/>
    This is normal button:<input type="button" value="i'm button">
</body>
</html>
```

优点：简单、明了，可以区分出各个 input 控件形态。

缺点：IE 6 之前的对 Web 标准支持得不太好的浏览器不能支持 type 选择器（致命呀）。

3：用 JavaScript 脚本实现

前台 html 代码：

```
    <!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> diffInput </title>
        <meta name="Author" content="JustinYoung">
        <meta name="Keywords" content="">
        <meta name="Description" content="">
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" >
        <style type="text/CSS">
        input{behavior:url('CSS.htc');}
        </style>
    </head>

    <body>
        This is normal textbox:<input type="text" name=""><br/>
        This is password textbox:<input type="password" name=""><br/>
        This is submit button:<input type="submit"><br/>
        This is reset button:<input type="reset"><br/>
        This is radio:<input type="radio" name="ground1">
```

```
                    <input type="radio"
name="ground1"><br/>
    This is checkbox:<input type="checkbox" name="ground2">
                        <input type="checkbox"
name="ground2"><br/>
    This is normal button:<input type="button" value="i'm button">
</body>
</html>
```

Css.htc 代码：

```
<script language=JavaScript>
switch(type)
{
    case 'text':
    style.backgroundColor="red";
    break;

    case 'password':
    style.backgroundImage="url(BG.gif)";
    break;

    case 'submit':
    style.backgroundColor="blue";
    style.color="white";
    break;

    case 'reset':
    style.backgroundColor="navy";
    style.color="white";
    break;

    case 'radio':
    style.backgroundColor="hotpink";
    break;

    case 'checkbox':
    style.backgroundColor="green";
    break;

    case 'button':
    style.backgroundColor="lightblue";
    break;

    default: ;//others use default style.
}
</script>
```

优点：可以分区出各个 input 控件形态。多种技术的混合使用，满足“我是高手”的虚荣心。

缺点：技术牵扯面较广，因为用 js 后期处理，所以在 js 没有起作用之前，各个 input 还是原始状态。较致命的是 FireFox 不支持。

4：在 Microsoft Visual Studio 2005 中使用 skin

Skin 文件参考代码：

```
<%--Style for common TextBox--%>
<asp:TextBox runat="server" style="background-color:#FFC ">
          </asp:TextBox>
<asp:Button runat="server" style="background-color:red">
          </asp:Button>
```

注意里面的样式是用 style 加上的，而不是用 CSSClass。道理很简单，如果用 CSSClass，前面的再用 CSSClass 就会覆盖这个 CSSClass，导致失败。当然，skin 不能单独使用，还要配合 CSS 样式表。

优点：可以区分出各个控件形态（注意：skin 只能对服务器端控件使用，所以现在已经不是单纯的 input 标签了，虽然这些服务器端控件“打到”前台的时候仍然是 input 控件）。除了 CSS，又被分离一层，使得样式的设置能有更好的定制性。其他优点可参考 skin 的优点。

缺点：只能对服务器端控件使用。不是所有的项目都能使用 skin 功能。

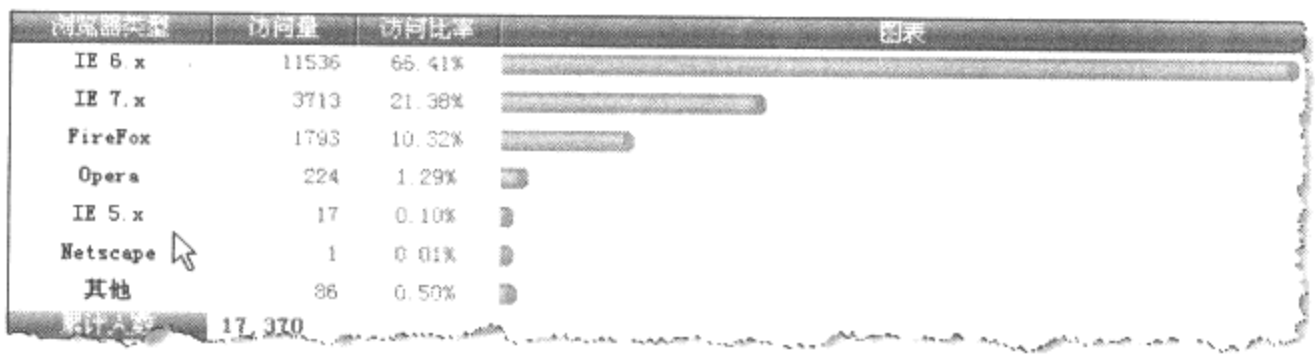
总结： 上面的方法各有优点和缺点，单独使用任何一个都不能很好地解决问题。所以应该多个方法配合一起使用，这样才能较好地解决问题。但是多个方法配合使用就完美了吗？NO～！它也有致命的缺点——多套方案的维护需要更大的成本！

后记： 这是一个以 IE 6 为首，非 Web 标准浏览器横扫天下的乱世年代，不知有多少网页初学者“惨死”在 IE 6 的诡异解析模式之下，又有多少程序员被 IE 6 所奴役，还有无数 Web 设计者在 IE 6 的胯下忍辱偷生。虽然黑暗中我们欣慰地看到 Firefox 这个反对暴统的勇者的出现，以及 IE 7 对 Web 标准越来越好的支持这道曙光，但是黑夜仍旧会持续很长一段时间。对于 Web 标准一统天下的年代，我们既喜且悲。喜的是，到那个时候，做网页设计和规划将会如同吃饭般简单，悲的是，如果真的到了那个时候，我们吃饭的饭碗还能那么重吗？不过，为了人类社会的进步，拯救地球的科技，为了发展宇宙的技术文化 -_-b... 我依然期待 Web 标准一统天下时刻的到来。

一个常被问到的问题：如何让层盖住<select>

作者：阿一（杨正祎）[http://www.cnblogs.com/JustinYoung]

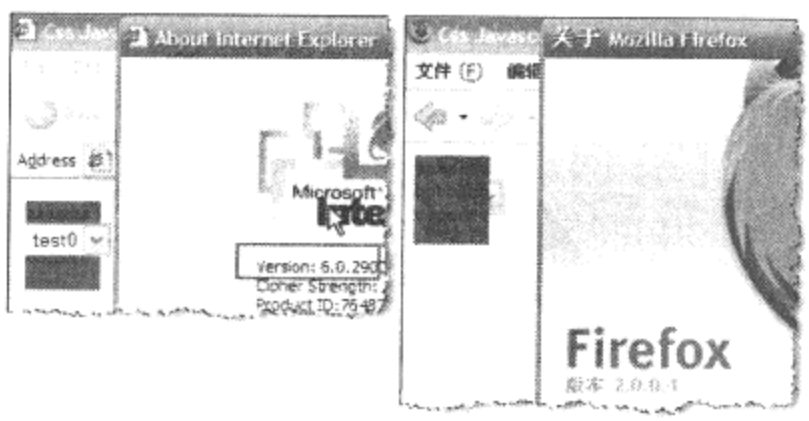
首先，我们不得不承认 IE 7 以前的 IE 系列浏览器对 Web 标准支持得真的很差。IE 6 的诡异解析模式让一些开始学习 Web 标准的朋友老是碰到不能理解的问题。特别是这个 IE 6 向 IE 7 过渡的是非年代。IE 6 真的让人很郁闷。但是就目前而言，我们还是不能放弃对 IE 6 的兼容。从我的 blog 访问统计分析数据来看，使用 IE 6 的还是占绝对主流的。



浏览器类型	访问量	访问比率	图表
IE 6.x	11536	66.41%	
IE 7.x	3713	21.38%	
FireFox	1793	10.32%	
Opera	224	1.29%	
IE 5.x	17	0.10%	
Netscape	1	0.01%	
其他	86	0.50%	
总计	17,370		

本来想顺便说说 Web 标准中这个“标准”到底是什么，但还是明日另起一篇吧。因为这个不是“顺便说说”就能说清楚的。今天我们还是步入正题——如何让层盖住下拉列表框？

非常郁闷或者非常幸运地说一下：这个问题只会出现在 IE 7 之前那些对 Web 标准支持不好的浏览器中（例如现在非常主流的 IE 6 -_-b... ），IE 7 和 FF 都不会出现这个问题，截图为证。



出现上面情况的参考代码如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "
```

```
http://www.w3.org/TR/
    xhtml1/DTD/xhtml1-transitional.dtd">
    <html>
    <head>
    <title>Css JavaScript Demo</title>
    <meta name="Generator" content="EditPlus"/>
    <meta name="Author" content="JustinYoung"/>
    <meta name="Keywords" content="CssStandard JavaScriptDemo,B/S,
JustinYoung"/>
    <meta name="Description" content="This demo from JustinYoung's
Blog:Yes!B/S!"/>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <style type="text/CSS">
    #DIVUp{
       z-index:99;
       position:absolute;
       background-color:red;
       width:100;
       height:18;
       overflow:hidden;
       height:60px;
    }
    #ddlTest{
       width:200;
       z-index:1;
    }
    </style>
    <body>
    <DIV id="DIVUp">aaaaaaa<br>bbbbbbb<br>ccccccc</DIV>
    <br/>
    <select id="ddlTest"><option>test0<option>test1<option>test2
<option>test3</select>
    </html>
```

对于 IE 6，其实我们也并不是没有办法，虽然我们不得不承认这个办法很“挫”，但这是目前最有效的办法。那就是在下拉列表上方加一个 iframe，然后让 DIV 层浮在 iframe 上方，这样就能使 DIV“盖住”下拉列表。如果你要问为什么，那么，首先恭喜你，你是个好同学，不像很多人只在网上找解决办法，而不是找知识（例如我-_-b...），然后我会告诉你，没有为什么，这个就是 IE 6 的诡异解析。如果一定要问为什么，我只能告诉你，在 IE 6 看来，如果只有 DIV 和 select，无论你的 z-index 怎么设置，DIV 的层永远会被 select

标签踩在脚底，而 iframe 则可以爬到 select 头上。所以，下面的方法之所以能解决问题，是因为 iframe 在 select 上方，而 DIV 搭着 iframe 的顺风车也爬到了 select 的头上。这有点像这样：一条京叭狗（DIV）平时老是被大狼狗（select）踩到脚底欺负，这天，京叭的主人（iframe）抱着京叭把大狼狗踩到了脚底，这时候京叭自然就在大狼狗的头上了。扯远了，下面给出解决方案代码：

```
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "
http://www.w3.org/
    TR/xhtml1/DTD/xhtml1-transitional.dtd">
    <html>
    <head>
    <title>Css Javascript Demo</title>
    <meta name="Generator" content="EditPlus"/>
    <meta name="Author" content="JustinYoung"/>
    <meta name="Keywords" content="CssStandard JavascriptDemo,B/S,
JustinYoung"/>
    <meta name="Description" content="This demo from JustinYoung's
Blog:Yes!B/S!"/>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <style type="text/CSS">
    body{
       font-size:small;
    }

    #zindexDiv{
    position:absolute;
    z-index:50;
    width:expression(this.nextSibling.offsetWidth);
    height:expression(this.nextSibling.offsetHeight);
    top:expression(this.nextSibling.offsetTop);
    left:expression(this.nextSibling.offsetLeft);
    /*background-color:green;在 ff 中将这句话放出来，你就会明白京叭、狼狗、主
      人的比喻*/
    }

    #DIVUp{
    z-index:99;
    position:absolute;
    background-color:red;
    width:100;
    height:18;
    overflow:hidden;
```

```
    height:60px;
    }
    #ddlTest{
    width:200;
    z-index:1;
    }
    </style>
    <body>
    <iframe id="zindexDiv" frameborder="0"></iframe>
    <DIV id="DIVUp">aaaaaaa<br>bbbbbbb<br>ccccccc</DIV>
    <br/>
    <select id="ddlTest"><option>test0<option>test1<option>test2
<option>test3</select>
    </html>
```

兼容 IE、Firefox 的图片自动缩放的 CSS

作者：MaxIE[http://www.cnblogs.com/MaxIE]

一直以来有个很头疼的问题困扰着我，那就是网页中图片缩放的问题。相关代码写到 JS 里面，不太容易修改；写到 CSS 里面，在 IE 6 中又不支持 max-width。今天用了很久时间终于解决了这个问题，基本算是完美了，唯一不完美的就是 IE 6 只有等图片完全下载完成后才会自动调整大小。不过聊胜于无，总比进入页面后看到长长的横向滚动条舒服得多，这里使用了 expression，但是利用了一次加载，所以 expression 不会造成内存泄漏。

将下面的代码加入到你的 CSS 代码中，便可以实现图片自动缩放。

```
.image {
    max-width:600px;height:auto;cursor:pointer;
    border:1px dashed #4E6973;padding: 3px;
    zoom:expression( function(elm) {
        if (elm.width>560) {
            var oldVW = elm.width; elm.width=560;
            elm.height = elm.height*(560 /oldVW);
        }
        elm.style.zoom = '1';
    }(this));
}
```

Web 标准

——标准无罪，有罪的是做标准的人和不按标准做的人

宣扬标准的人都说，标准可以让我们的工作更有效率，标准也可以让用户感觉更好，你感觉到了吗？宣扬标准的人还说，标准是个很简单的东西，不需要你去修改太多的东西，你觉得呢？事实上，我们对标准有着太多误解，而标准本身又有太多的不完善。

在《欲练 CSS，必先宫 IE》、《你有<table />强迫症吗？》、《根本不存在 DIV + CSS 布局这回事》、《慎用 XHTML 标签的自关闭写法》这 4 篇文章中，让 Cat Chen 来告诉我们，为什么我们对标准的感觉不像布道者说得那样好，又为什么我们总是很难做出标准的网页。

阿一的两篇文章《Web 标准页面设计——要注意的很多》和《Web 标准不标准》告诉我们，我们还有很多事情要做，而标准本身也有很长的路要走。

之后是两个很好的系列文章，爆牙齿的《走在 Web 标准化设计的路上》和阿一的《IE 7 标准之道》。

爆牙齿功力深厚，让他来写《走在 Web 标准化设计的路上》这系列文章，实在是恰当之极。这并不是一系列的技术文章，而是希望能够让大家明白怎么样“正确理解和运用”Web 标准。在这里，爆牙齿不但为我们阐释了为什么要标准，还道出了标准真正的奥义——标准并不是 DIV+CSS；与此同时，他还指出了一些不尽合理的标签，比如 h 系列标签；之后，在 和的问题上花了不小的篇幅来告诉我们到底什么是 Web 标准中的“结构”；在系列的最后，还为我们揭开了“语义”的神秘面纱，这是一个系列的结束，也是一个征程的开始。

相对爆牙齿，阿一绝对是一个更加务实的实践者，我一度很怀疑怎么可能有人会对 IE 7 下的标准如此之熟悉，最终的事实是：阿一就是这么熟悉！常见的问题、奇怪的行为、神奇的 Bug，等等，正常的、疯狂的事情《IE 7 标准之道》里全都有，看这些文章，你一定会奇怪，原来 IE 7 还有这么多奇妙的东东啊。

Web 标准页面设计——要注意的很多

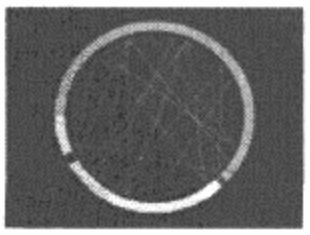
作者：阿一（杨正祎）[http://www.cnblogs.com/JustinYoung]

来上海的第一个项目，无论是公司还是自己都有很多的不足。第一次做娱乐型的门户网站，对 Web 标准的掌控的确是个很大的挑战。说实话，这个项目中对于 Web 标准的使用自己只能给自己打 65 分，分数之所以那么低，项目时间短是一个原因，美工的设计稿比较差是个原因，自己对大型娱乐型网站的网页框架的设计不足也是个原因。正是因为分数这么低，所以项目的总结就显得特别重要。不断地总结，不断地改进，不断地完善。我很高兴，因为我依然走在“好好学习，天天向上”的路上……

美工和策划都生活在乌托邦

不要急着马上动手，美工给过来的页面一定会隐藏着很多不合理的地方。把页面打印出来，看！根据以前的经验把这些不合理的地方找出来，标记出来，和策划进行商讨和改进。不合理的地方可以通过以下方式查找出来——对一个地方问一下：这地方的文字（或者图片、记录、信息块）如果很长怎么办？如果很短怎么办？如果没有怎么办？如果很多怎么办？如果很少怎么办？如果有权限怎么办？如果没有权限怎么办？……多考虑一些情况下的显示和展现方式。

唯一不变的只有改变



即使经过策划重新改进和商讨的版本也一定会隐藏很多不合理的地方，所以动手开始页面设计之前，一定要注意为即将使用的框架预留变更的可能性和可行性。页面设计也是软件开发的一部分，自然会遵循软件开发的一句神语——唯一不变的只有改变！

网页设计框架

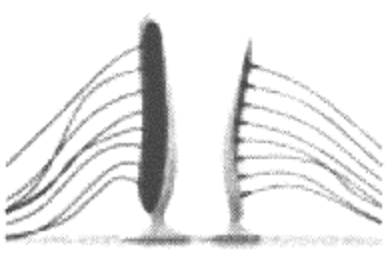
如何避免大量重复的工作，不再为做了一遍又一遍的东西浪费时间和精力？Framework，框架。编程有编程的框架，网页设计也可以有网页设计的框架（例如 yahoo 的 YUI 框架）。所以将常用的、通用的东西集合到框架里面去，将是种成熟的做法。我正在试着搭建一个符合 Web 标准的网页设计用的框架，但是自己能力的确有限，加上后期项目时间太紧，所以这次项目中虽然开头设计了一部分，但是后来终究没有完成和使用。如果有高人有相同的想法，欢迎指教。如果对网页设计框架有兴趣，但是还不是很了解的朋友，推荐看一篇文章《Frameworks for Designers》[①]。

建设健壮的页面

因为美工和策划都生活在乌托邦，所以他们的设计很多都太理想了，工工整整煞是好看。但是很多的时候，数据并不都是那么工整和漂亮。所以页面的布局还需要通过 CSS 去强制限制，确保布局在异常状态下不会混乱。

鉴于这个比较重要，而且不是一句话两句话就可以说明白，所以我决定以后单独为此写一篇文章，敬请关注我的博客。

id 和 class 到底要用哪一个?



首先要明白 id 和 class 各自的优缺点，这样才能根据他们各自的特点进行使用。

id 的优点（class 的缺点）：id 写在 CSS 中用“#”选择器，class 写在 CSS 中用“.”选择器。“#”选择器的优先级高于“.”选择器大约 10 倍，所以当你需要提升优先级的时候，id 标签或者 id 容器内的标签将是很容易和有效的，而 class 标签或者 class 容器内的标签将可能导致优先级的提升失败。

[①] http://www.alistapart.com/articles/frameworksfordesigners

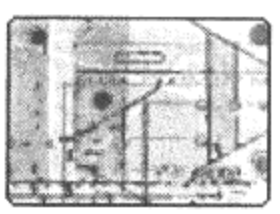
id 的缺点（class 的优点）：id 应该是唯一的，所以它的可复用性是很差的，而 class 是可以复用的。所以如果一块东西是多个页面，甚至一个页面都会使用多次的，那么一定要使用 class 来作为样式选择器。id 是唯一的，当一个控件的 id 的产生是不可控的，那么这个 id 选择器将失去意义，但是任何一个控件即使是动态产生的，他的 CSSClass 仍然是可定制的，所以当你的这个标签需要用服务器端控件替代的，而服务器端控件的 id 是不确定的时候，那么请使用 class 选择器，这样只要将服务器端控件的 CSSClass 设为你 class 选择器的名称即可。（当然，这个还需要大量的经验的积累，项目做的多了就会逐步的改进）

padding 和 margin 到底要用哪一个？

padding 和 margin 可以让一块区域的外观显示完全一样，所以可能让很多人认为 padding 和 margin 是可以互换的。其实它们的差别很大，而且选择哪个需要认真和慎重地考虑。我认为对容器使用 padding 还是对容器内的标签使用 margin 的原则是，当隐藏这个容器或者容器内的标签时（现实项目中经常需要将某个部件隐藏、显示），对整体布局影响最小为宜。

对于 padding 再说一句：IE 6，IE 7（FF）对带有 padding 样式的标签的宽度的解析是不一样的。IE 6 的标签宽度不包含 padding-left 和 padding-right 的值，而 IE 7 和 FF 则是包含的。例如一个 DIV 的 width 设置 100px，padding 设为 10px，而在 IE 6 中它要占据的宽度是 120px（包含 10 个 padding-left 和 10 个 padding-right），而在 IE 7 和 FF 中则占据 100px 的宽度。因为 IE 7 和 FF 会认为 100 已经包含了 20px 的 padding。

min-height 和 height



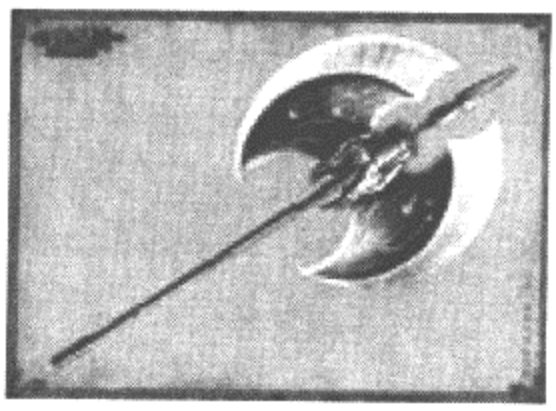
如果你只需要兼容 IE 6，那么完全不需要注意 min-height 这个样式，因为 IE 6 根本就不支持这个样式。但是当你的页面需要照顾到 IE 7 和 ff 的时候，这个样式一定要注意。因为很多在 IE 6 下设置了 height=固定值的样式，当容器被里面的东西撑得大于这个高度的时候，IE 7 和 FF 是不会自适应高度的，从而导致布局的混乱。要想在 IE 6、IE 7 和 FF 中都可以自适应高度，正确的做法是设置 min-height 和用 CSSHack 设置 height。例如：

```
min-height:600px;
_height:600px;
```

这样，在容器里面的东西很少的时候，它显示固定高度 600px；当里面的东西很多的时候，它也会自适应的增长高度。

对于 height 的设置一定要特别注意。如果是布局用的容器的 height 则需要特别的注意，否则在 FF 中会导致无法浮起，从而使布局混乱。

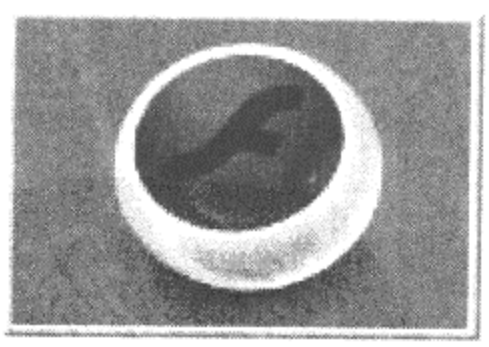
找把顺手的斧头



工欲善其事，必先利其器。页面搭建的工作量是很大的，所以为了提高工作效率，找一个适合自己的开发工具是很有必要的。

自从 Macromedia 被 Adobe 收购以后，我就不再使用 Dreamweaver 了。因为对 Web 标准支持得不是特别好。我比较喜欢的 Web 开发工具是 editplus，配置好符合自己习惯的自动完成功能，作为轻量级的页面开发工具，开发效率还是很高的。重量级的开发工具，我比较喜欢 Expression Web 2。习惯 editplus 开发 Web 的朋友可以很好地过渡过来，它的智能提示和可配置的自动完成功能可以很好地提高效率。再加上它有站点的概念，所以批量替换和修改比 editplus 要方便很多。一些 editplus 和 Expression Web 2 的使用心得和技巧我也会在以后的日子里陆续的汇总，请你订阅我的播客，以便尽快得知。谢谢。

Flash 是个捣乱的家伙



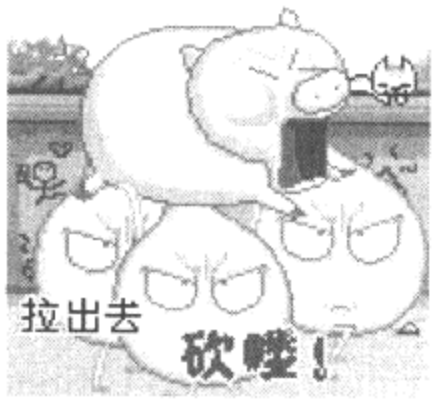
虽然很强悍，而且很流行，但是我对它一直没有好感（因为我不会 ^-^）。网上问得最多的就是如何让含有 flash 的网页通过 w3c 的 xhtml 验证。其实很简单，就是使用 object 标签。但是如果你的页面需要在 Microsoft Visual Studio 2005 打开再次进行工作，那么请不要让页面包含 flash 的 object 标签。因为他会让你的 Microsoft Visual Studio 2005 虚脱。出现非常令人费解的问题——没有办法打开设计模式，back 和 del 键不能使用，只能打字，不能删除。反正 flash 就是个捣乱的家伙，让你的 Microsoft Visual Studio 2005 堕落。（在我们开发团队多台电脑出现此问题。但是依然不能排除是我们的 Microsoft Visual Studio 2005 或者电脑环境的问题。如果是我们的问题，我

对 flash 的误解表示真心的抱歉）

排队，排队！

多列布局的网页十分多见，别的不说，就是现在我使用的博客园的这个风格就是个两列的布局。左面的是 side，右面的是 main。依照大家的阅读习惯，html 里面当然应该先写左面的 side，然后再写右面的 main。其实这样做是不合理的，因为浏览器的解析是从上到下的，先解析出来的先显示，后解析出来的后显示。而左面的导航栏并不是用户急于想看到的，用户急于想看到的是 main 里面的文章的内容。所以正确的写法是侧边栏写在下面，网页主题写在上面，然后利用 float 样式，让它们出现在左边或者右边。

What's your Name? "AD"?拉出去砍了！



如果这块区域要显示广告，所以就把这个 DIV 命名为“divAD”，OK，没有问题，命名很准确，而且采用了驼峰式。但是为什么很多浏览器下看不到你的广告？很简单，因为他们被屏蔽了。被谁？浏览器、杀毒软件、甚至防火墙。很多东西都很乐意“玩弄”这些广告，不要把你的东西命名为“AD”或者“banner”。还是换个名字吧，“牡丹”、“芙蓉”随便选。

Short Live the "Button" Tag

我刚发现 button 标签的时候高兴了一阵，这个标签好呀！里面可以包含很多其他的标签来构成不同的按钮样式（例如按钮图片可以这样写(<button></button>)，而且点击的时候还有“偏移量”这样华丽的动态。所以我在这次项目中使用了。但是后来发现，当需要把这些华丽的图片按钮 button 替换成服务器端控件时，问题出来了。因为 vs2005 中没有什么控件打到前台是 button 标签

（采用控件编程实现的方法除外）。“imageButton”？不是，它打到前台不是 button，而且它也不是你想象中的是 img，它打到前台是 type 为 image 的 input，这点让程序员们很无措。所以我决定以后不再采用 button 标签。“Butoon”标签，在我这里还真是个短命的种。

图片按钮！img or input?

美工真的是生活在童话里，他们将页面设计得花里胡哨，到处都是渐变和色彩绚丽的按钮，以至于我的电脑一打开他们发过来的 psd 文件就一定会死机。他们上网好像用的从来都是光纤，不知道一个按钮图片就有 50 多 KB，需要下载多长时间。不过，也没有办法，中国的美工也是靠花哨吃饭的，所以网页上从来都没有系统默认样式的按钮，从来都是图片按钮。问题是作为网页设计师，你需要决定将这些按钮图片用何种标签表现出来。网页标签的种类合适与否决定着程序员在后期开发中的工作效率，所以选择使用何种标签是要经过慎重考虑的。图片嘛？自然是 img 啦。但是如果这个图片按钮需要处理一些服务器端代码，那么 img 将是不合适的，所以图片按钮如果用服务器端控件的话，一定是 imageButton。而 imageButton 打到前台解析出来是 type 为 image 的 input，所以如果进行服务器端处理的图片按钮还是用 type 为 image 的 input 吧。这样程序员就知道直接拉 imageButton 了。

“a 君请过来!” A 君：“谁是 a 君？”

CSS 是大小写敏感的，所以 #DIVTest 不会为一个 id 为 DIVtest 的 DIV 渲染效果。“为什么没有效果呀！”，找啊找，找啊找，“哎呀！原来你是小写的！”顺便说一个我在此次项目中犯的一个很小但是后果很严重的错误。因为大意，我开始将 video（视频）写成了“vedio”。而且很多的样式都含有单词 video 或者 Video。后来发现这个问题，就用批量替换将 vedio 替换成 video。而且是不 care 大小写的那种批量替换。后果可想而知呀 ~ ~

这是什么？——/*缁旀牎鍞滈噦鎺戝瀛閤ㄥ婺顫嬫0鎴濆瀛鍀?/

这是什么？如果我告诉你这是注释你相信吗？不对吧，CSS 的注释后面应该是“*/”结尾吧。是呀，我本来写的也是像“/*中文注释*/”这样的，但是 vs2005 将我的中文变成了该死的乱码，而且还把后面的那个注释用的“*”也变成了乱码！结果我大批的样式都失效了。

版本控制失败！轻则令人抓狂，重则吐血身亡！

当项目很小的时候，版本控制不那么重要，但是项目越大，版本控制就越重要了。如果版本控制失败，轻则令人抓狂，重则吐血身亡！

所以一开始不要怕麻烦，还有尽量使用 vss 或者 smc 这样的工具进行控制。不要为了一次的方便而破坏版本的控制流程。如果要维护多处的时候，混乱的程度将是你不可想象的。

欲练 CSS，必先宫 IE

作者：Cat Chen[http://www.cnblogs.com/cathsfz]

“Win 国天下，欲练 CSS 之人不在少数，大多不得要领，又或是走火入魔，全为 IE 所累。故曰：欲练 CSS，必先宫 IE。”

曾经，我也属于为 IE 所累的行列，如今见到很多人仍然不愿意对自己的宝贝 IE 下手，所以决定特意写篇文章说说此事，以明辨 IE 到底是宝贝还是累赘。

好了，funny 部分结束，按回我的习惯直入正题。之所以说 IE 不好，是因为 IE 会误导你对 CSS 模型的理解，让你以为 IE 的理解是对的，之后无论如何你都无法用你的 IE 模型理论去为你那个无法在 FF 正常显示的 CSS 提供 fix。更坏的事情是，即使你仅仅针对 IE 设计，不考虑其他浏览器，由于 IE 模型绝对可以说是一只让人难以捉摸其脾气的怪物，所以你单纯为 IE 设计也会遇到众多难题，总是会发现很多的效果绕来绕去都难以实现。

我们都知道，XHTML+CSS 的目标就是实现内容与表现分离，理论上对于任何特定一份内容，我们都可以通过 CSS 实现任何我们想要的表现形式，或者细致地说是布局形式。虽然现实与这个目标有一定差距，但是 CSS 已经能够满足大多数常见的布局需求，这有 CSS Zen Garden 为证。然而，如果你用的是 IE，因为它难以捉摸，所以如果你想用一种简单优雅的 CSS 去让 IE 实现“任何你想要的布局形式”，那是不可能的，只有复杂繁缛的 CSS 才能够在 IE 上满足你的需求。我曾经提到过一种理论——“一个人对一个研究方向是否感兴趣很可能是完全靠偶然事件决定的，这就好像人第一次打羽毛球，如果你赢了几盘你就会感兴趣，如果你一直都赢不了你就会没兴趣”。IE 在需要复杂繁缛的 CSS 这一点上，就足以令大多数的入门者却步。你总感觉到不得要领，你自然没兴趣学下去。

举一个例子说明这个问题，例如你不知道 IE 有 hasLayout 这回事，一个元素是否 hasLayout 对它的布局方式有重大影响，于是你肯定用最简单的思维去思考 CSS，认为不同的 CSS 规则之间应该是松耦合的。“CSS 应该被设计为简单优雅的”，你肯定会这样想，没错，它确实被设计为这样，不过 IE 不是这样去实现 CSS 罢了。我们用下面的代码去证明 IE 在 quirks mode 与 standards mode 之间的区别：

```
<div style="background-color: red; height: 30px">
```

```
        <div>Hello</div>
        <img style="float: left; width: 200px; height: 160px" src=
"blank.gif" />
        <div>Hello</div>
    </div>
```

首先，我们用 quirks mode 看看结果如何，并且一个初学者看到这样的结果会去如何理解 CSS 规则。在 quirks mode 中，我们可以看到背景为红色的 包含了上面一行的文本，以及下面向左浮动的<img />（自然也就包括在浮动块右边的文本）。在这里，我们可以建立两种认识：

- 容器是完整包含内容的，当内容的总高度比容器大的时候，容器就会自然伸展以确保容纳内容。
- 浮动块也属于上述条件所要求的通过伸展确保容纳内容。

以上规则是完全错误的，一个懂得标准 CSS 以及理解 quirks mode 的设计师将会如此解释：

- 因为 IE 在 quirks mode 中会将 height 理解为 min-height，所以它认为<DIV />的高度不小于 height 指定的 30px 即可。而根据 CSS 标准，当 height 设置为 30px 时，高度就一定是 30px，超出部分如何处理则由专门的 CSS 规则决定。
- 因为 被设置了 height 属性，在 IE 中这就让它 hasLayout 了，并导致它一定要包含所有的内容，包括浮动块。而根据 CSS 标准，浮动块是无需被完全包含的，它就浮动在那里，除非遇到设置了 clear 属性的元素，否则后继内容只会侧移避让。

好了，相信这个对比足以说明问题的严重性了。通过 IE 的效果去理解 CSS，最终只会让你的理解与真实的CSS 相差甚远。详细的 standards mode 与 quirks mode 带来的标准执行差别，可以参考这篇文章：《*CSS Quirks mode and strict mode*》。

然后肯定有人要问我，如果通过 doctype 确保使用的是 standards mode，那是不是就没问题了呢？standards mode 确实会让 IE 对 CSS 的解释合理很多，但事情并没有那么简单，这你可以通过实践去慢慢体会。可以尝试在 standards mode 中设计 CSS，并且尽力保持它们在 IE/FF/Opera/Safari 这 4 大主流浏览器中显示一致，随着设计的进行，你会发现这不是那么容易做到的。或许你不乐意花时间去 fix 其中的一些小问题，宁愿任由其中一些浏览器的用户看到比较丑陋的布局，但至少你已经了解到一个和上面例子类似的道理：不同浏览器即使同样在 standards mode 下，其对 CSS 的理解仍然有所差异，而差异当中最多只可能有一个是正确的，甚至可能全部都是错误的。这篇《*CSS contents and browser compatibility*》就列举了众多浏览器对 CSS 支持的差异，一份 CSS 总会因为其中有一些规则在某些浏览器上是不支持或者是 buggy 的，而导致

你难以保持它们在不同浏览器上显示一致。

接下来可能还有人会问我，既然 IE 的市场份额最大（特别是在入门级的用户当中），又或者说我的客户指定使用 IE 作为客户端，仅仅针对 IE 设计 CSS 不好吗？为什么要针对 FF 之类的标准浏览器设计 CSS 然后再为 IE 进行 fix？因为 IE 难以捉摸的脾气，让你无法将它的行为理解为一种简单优雅的规则，然后让你陷入 CSS 规则高度耦合的困境中。请看下面的例子：

```
<div style="background-color: red; border: 2px black solid">
  <img style="float: left; width: 200px; height: 160px" src=
"blank.gif" />
  <div>Hello</div>
</div>
<div>Hello</div>
```

现在，你在 IE 中看到的效果应该是左边出现<img />，然后两个<DIV />内的 Hello 都向右偏移以避让<img />这个浮动块了。其中上面的 仅仅占用移行的高度，因为它没有声明高度，所以就是自然高度，也就是一样，这些都很好理解，所有规则都是解耦的。然后向例子中增加对第一个 的 width 属性复制，看看结果会如何：

```
<div style="background-color: red; border: 2px black solid; width: 600px">
  <img style="float: left; width: 200px; height: 160px" src=
"blank.gif" />
  <div>Hello</div>
</div>
<div>Hello</div>
```

这时候第一个 完全容纳了<img />，把第二个 挤到下面了。这该怎么解释呢？我们可没有设置它的 height 属性哦，难道又是之前例子所说的因为 hasLayout 而必须容纳所有内容的问题？正解，这就是 IE 难以驯服的地方，一个应该是完全独立的 width 属性，设置之后引起了高度以外的其他影响，这让人无法尝试以一种简单优雅的方式去理解 IE 的行为。这就证明了，如果要学习如何为 IE 设计 CSS，就要先学习标准 CSS，再加上对 IE 怪异行为的理解，这比仅仅学习如何为一个标准浏览器设计要难多了。这时候你是不是想说，“如果客户愿意放弃 IE，甚至全世界都愿意放弃 IE，那就实在太美好了”，没错，这才是正确的想法，一心想着仅针对 IE 设计以求方便只会让你走火入魔。

最后，如果你已经有了一定的 CSS 基础，对 CSS 规则都理解无偏差，却缺乏组合 CSS 规则的想象力，无法做到所谓的“实现任何你想要的布局效果”，这也就是说，你的内功已练成，仅仅差一些表面的套路，这时候我推荐你去看《*CSS Mastery/精通 CSS*》。看完这本书，相信你只会觉得自己缺乏布局的创造能力，而不会有布局却不知道如何实现的情况。

你有<table />强迫症吗？

作者：Cat Chen[http://www.cnblogs.com/cathsfz]

上次讲到“欲练 CSS ，必先宫 IE”，如果你宫了 IE 然而还是觉得不得要领，那就该怀疑自己是不是有传说中的 Table 强迫症了。

在 CSDN 社区上，时不时能够看到一些页面整体布局的问题，要求用 DIV 做一些 Table 才能做到的，否则就以此为把柄说 XHTML+CSS 布局方法不好。其实，首先要做的是改变思维，以适应 XHTML+CSS 的布局。

面向页面设计而非面向浏览器设计

XHTML+CSS 能够实现的是一种流布局，也就是随着内容的长度自动增长，并且最终导致整个页面增长，这时候浏览器就必须显示滚动条。Table 强迫症的一个征兆就是极力避免流布局，希望以浏览器的可视区域为布局目标，要求在可视区域中划分内容区域而不是在页面上划分内容区域。实际上 XHTML 是无法针对浏览器设计的，因为它仅仅包含语义，或者说是内容，而浏览器如何去表现这些内容是我们无法确定的。CSS 提供了我们控制表现方式的一种途径，但这仅仅是针对主流浏览器的，而且浏览器支持的“指令集”还有稍微的差别（说到这，我真希望能够为一个浏览器写 CSS 然后编译为全平台兼容代码），最后这些指令暂时还仅仅支持针对页面的流式布局控制。因此，如果你决定要开始写符合语义的 XHTML 并且仅仅用 CSS 控制布局，首先就要把思路转变为面向页面（或者说是文档）的布局控制，而非面向浏览器可视区域的布局控制。

接下来肯定有人要说“那你就是承认了有些布局老方法很容易做到，但新方法很难做到”。这是当然的，然而这不成为我们继续使用 Table 的理由。这时候要反过来探讨原始目标——我们为什么要控制布局？低层次的需求是为了美观，谁都希望同样的内容能够以更好的视觉效果展示在用户眼前；高层次的需求是为了控制受众的浏览方式，让他们能够按我们预先设计好的方式来区分页面内容的轻重点，按我们的期望优先浏览某些内容，同时也帮助他们更快地找到他们想要的内容，而不会在我们的网站内感到沮丧。既然我们确定了这时控制布局的目标，那么我们再来看看 CSS 是不是“没办法把事

情做好”。首先，CSS 也能做出美观的页面，虽然某些布局做不到，但是在 CSS 的限制下做到同等美观程度的页面是肯定没问题的。其次，CSS 也能让设计变得友善，不会说 CSS 的设计就肯定是“干净”到用户无法一眼找到他想要的功能。因此，虽然 CSS 无法实现某些特定的布局效果，但对于设计师来说它能够达到老方法所能达到的同等效果，这就足够了。

从 XHTML 中去掉内容无关的视觉元素

另一个 Table 强迫症的征兆就是，习惯为每一个视觉上的元素对应一个 XHTML 元素。在 Table 中，无论视觉效果有多复杂，我们总能不停地切割 Table，甚至 Table 套 Table，直到准确定位每一个特定的元素。然而应用了 CSS 之后，这就是不必要的，甚至会给设计师带来麻烦，因为 XHTML+CSS 就是为了内容和布局分离，所以如果一个视觉元素与内容无关，那么它就不应当出现在 XHTML 中，自然也就不会对应一个 XHTML 元素。

例如有一个网站当前栏目的徽标，这个徽标没有任何的语义，而 XHTML 中也有文字内容描述当前栏目了，那么这个徽标就并不一定要对应一个<img/>元素。如何让徽标显示出来呢？它可以是当前栏目文字描述区域的 background-image，同时通过一些定位技巧让它显示出来。如果你认为有这个徽标就不需要文字描述时，你还可以通过定位技巧将文字隐藏掉，这样单纯看 XHTML 或者在不支持 CSS 的浏览器上就只见文字描述，而在支持 CSS 的浏览器中则看见徽标。从这个例子我们可以看到，一个视觉元素不一定要对应 XHTML 中一个实实在在的内容元素，或者对应一个文本元素而非图形元素。XHTML 包含的是内容，那就不应该包含与内容无关的视觉元素描述，而通过 CSS 你可以事后增加有关的视觉元素。

又例如，:before 和:after 这两个伪选择器，允许创建插入在匹配元素前后的元素，这样就能够实现非内容视觉效果仅在 CSS 中插入。常见的用法包括：插入 clear 到浮动元素之后以确保浮动元素的完整包含，或者是引用语句的前后自动加上引号。事实表明，CSS 是很适合于将非内容的元素从 XHTML 中分离出来的，因此我们在设计 XHTML 时就不能够总想着要有什么效果，而应该单纯想着信息的组织形式。

最后，如果要我为 Table 强迫症开处方的话，我还是会选择《CSS Mastery/精通 CSS》。看完之后，你自然能够解除上述的烦恼，理解 CSS 布局带来的便利，从而选择开始用纯 CSS 的思维来进行设计。

根本不存在 DIV + CSS 布局这回事

作者：Cat Chen[http://www.cnblogs.com/cathsfz]

在《欲练 CSS，必先宫 IE》和《你有<table /> 强迫症吗？》这两篇文章中，看到有不少评论用到 DIV+CSS 布局这个说法，用来和 Table 布局比较。实际上 DIV 不是用来布局的，DIV 只是用来表示一个其他元素都无法准确表达语意的一个块区，只有 CSS 是用于布局的，所以根本就不存在 DIV+CSS 布局这回事。反过来，Table 布局的时候经常依赖于 CSS 定义一个单元格的布局属性，所以可以说是 Table+CSS 布局。也就是说，我们讨论的两种主流布局方法应该是纯 CSS 布局和 Table+CSS 布局，如果你觉得你在用的是 DIV+CSS 布局，那么有可能你也有强迫症了。

接下来我们说说如何进行纯 CSS 布局，因为 CSS 布局依赖于 XHTML，所以我们先要说说如何书写一个 CSS 无关的 XHTML。其实书写 CSS 无关的 XHTML 并不难，虽然你不能再像书写 Table 布局代码那样集中精力于最重的视觉效果上，但其难度也不过是中学生写作文那样。

中学生写作文如何写呢？首先看看题目，然后想想整篇文章分为哪几个大的段落，每个大的段落说些什么，把你要说的东西说清楚。对于 XHTML 来说，这相当于用 DIV 把文档切割为几大块。这时候你不要想着这些 DIV 将构建一个怎样的 DOM 啊、CSS 如何选择 DOM 中元素设置规则实现布局之类的事情，就大概划分一下文档的大区域就好了。

然后当然是用一些常用的手法来表现感情或者论证问题，在 XHTML 中就是用特定的元素来完成一些常见的信息组织。下面就是信息组织形式与元素的对应列表。

img

作为内容的图片是一定要放到 img 里面的，这没有更好的选择了。然而如果图片不是作为内容，而是作为修饰性的，则千万不要用 img。对于非内容的图片，应该在 CSS 中引用，而不在 XHTML 中出现。例如每一个导航链接有一个前导的箭头指示，那么这些箭头就应该通过 CSS 的 background-image 属性加上去，而不是直接作为 img 出现。

a

这也是一个非常准确定义的元素，链接都需要使用它。或许已经有很多人忘记了 a 的本意是锚点，其实这是一个十分有用的语义，你可以用它来标记文档中一些重要的引用位置。

ul, ol

ul 和 ol 分别是什么意思呢？如果你回答不上来，却知道它们可以用来干什么，那证明你是被可视化工具宠坏了，要转换过来编写符合语义的 XHTML 需要先补充基础知识，这时候你最好先找一些看起来非常基础非常全面的 XHTML 书籍看看，因为没有扎实的基础你在上面构建更多的知识都是不牢固的。ul 和 ol 分别代表 unordered list 和 ordered list，也就是无序列表和有序列表。在语义上，它们都用于表示一类并列关系的内容，例如我们去商店购物之前列一张 shopping list，上面要买的东西就是并列关系，在中文可以用顿号隔开那种。它们的差别在于是否有顺序，例如 shopping list 是没顺序的，先买什么后买什么是没关系的，但是一份旅游行程安排上面的景点列表是有游览的先后顺序的。

ul 常用于导航栏，因为导航元素符合上面所说的并列关系，树状导航结构还可以通过嵌套 ul 来表述。在这里，导航可以是我们常见的水平或竖直导航栏，甚至可以是地图导航，例如在中国地图上不同的省份热区其实是不同的 li。如果我说，在主流浏览器上用户看到了中国地图和可以直接点击省份热区，在不支持 CSS 的浏览器上用户能看到一份纯文本的省份名称列表，使用的是同一份 XHTML，而这完全通过 CSS 实现，甚至不依赖于 JavaScript，你相信吗？

另外，如果你要显示一个图库的缩略图，这些图片也可以放在 ul 中，因为这些图片也是并列关系。它们可以自动先横排，排满一行就自动排第二行，CSS 可以让他们乖乖排队，而不需好像 Table 那样把图片定死在一个格子里。其实 Table 用于布局就如同用监狱关押内容一样，把内容锁死在一个格子里不让它到处乱跑；符合语义的 XHTML 就如同一个开放的舞台，你只要懂得利用 CSS 的规则，内容就自然会找一个适合表现自己的地方站着。

dl

没有听说过 dl 吗？因为那些可视化工具生成的代码中从来不会出现 dl？dl 的意思是 definition list，也就是定义列表。它包含的子元素不是 li，而是 dt 和 dd，也就是 definition term 和 definition description。dl 本身设计为字典单词与解释列表这样的语义，例如：

```
<dl>
  <dt>Apple</dt>
```

```
  <dd>苹果</dd>
  <dt>Boy</dt>
  <dd>男孩</dd>
</dl>
```

如果你需要表示的语义也是类似的，一个列表既包含定义也包含解释，那么也可以考虑用 dl。

form, input

form 也就是表单，这没什么好说的，就算再不顾及语义的人在书写 XHTML 时也会考虑到它与各种 input 对提交数据的影响，从而小心谨慎。

Table

Table 自然是用来表示表格的，这不是废话！如果是数据表，当然可以用 Table 来表示，但如果不是，就最好别用 Table 了。

人名列表呢？例如一个 3 行 4 列的人名列表。如果这 12 个人名是并列关系，我建议你用 ul 和 12 个 li 来表示，再通过 CSS 来让它们在一行内并列显示多个。名片表呢？也就是 3 行 8 列，每两列中左侧一列显示人名右侧一列显示电话地址等联系方式。我觉得 dl 在一定程度上能满足此需求，dt 放人名，dd 放联系方式，不过这时候就涉及了 dl 滥用的争论，因为人名与联系方式当作定义与解释有点牵强。

接下来还有一个关于你是否系统学习过 XHTML 的小提问，那就是你是否知道 Table 下面的 caption、col、colgroup、thead、tbody、tfoot 元素及 summary 属性分别用于定义什么，还有就是你书写 Table 时是否会使用 thead 和 tbody。

DIV, SPAN

再次审阅上面的列表，如果你需要表示一个块区却无法在上面找到更适合的元素，那么就可以考虑使用 DIV 和 SPAN 这两个最没有语义的元素了。DIV 与 SPAN 的区别，历史上的不说了，现在通常大块的区域用 DIV，行内的小文本片段就用 SPAN。在上面我已经说了 DIV 一般用于全局划分为几个大的区域，所以一般不需要使用了。SPAN 其实也很少使用，因为行内的强调通常可以用语义更强的元素，例如 strong 和 em。

在理解上述那么多常用元素后，写一个 XHTML 就真地如同中学生写作文一样容易啦，还是搭积木那样，其实和以前使用可视化工具搭积木没什么不同，唯一不同是现在你理解了你在搭的是什么，而以前你只在乎搭出你想要的视觉效果来。写代码与写作文类似的地方就在于你写的越多就越熟练，也就越能写出好东西来。在写好 XHTML 后我们就要开始考虑如何写 CSS 了，或许还需要在 XHTML 中略作修改以方便 CSS 中规则的选择与匹配，不过这是以后再说的内容了，今天就说到这里。

慎用 XHTML 标签的自关闭写法

作者：Cat Chen[http://www.cnblogs.com/cathsfz]

我们都知道 XHTML 里面的 img 标记应该这样写：，这种写法也就是所谓的自关闭，在 XML 中是完全合法的写法。如果你熟悉 XML 相关的开发，可能也就习惯于这种写法，XML 中任何不含子节点的元素都可以这样写，那么 XHTML 中没有内容的标签也都可以这样写。XHTML 中理论上当然允许任何标签以自关闭的方法来书写，然而浏览器兼容性却带来了新问题，那就是 IE 无法正确识别某些标签的自关闭写法。

请尝试输入以下 XHTML 代码并在 IE 中浏览： hello <script type="text/JavaScript" /> world ，你会发现只能看到前面的 hello 而不见后面的 world，这事情让人挺无法解释的吧。可能有不少人都曾经遇到过这个问题，并且花了几个小时都找不到合理的解释。

解释源自另外一段类似的代码： hello <textarea /> world ，在 IE 中看看其显示效果，能够得到合理的解释了吗？能够看到前面的 hello 正常显示了，而后面的 world 则显示在 textarea 里面，这证明 IE 并没有正确识别 textarea 标签已经自关闭了，而是当它没有关闭，并将后面的内容识别为 textarea 内部的内容。

这时候我们就明白前面那段代码为什么看不到后面的 world 了，因为它被当作 script 的一部分来识别了。这就说明，在我们使用 XHTML 时并不能像 XML 那样随意使用自关闭的写法，只有少数原本不需要关闭的标签可以用自关闭的写法，其他标签即使没有任何内容最好也用成对的关闭写法。

最后需要提醒大家的是，其实弱智的 parser 不仅仅 IE 有，很多地方都可能碰到由于 parser 不严谨而引起的问题，所以我们在书写 XHTML 的时候还是要迁就一些老 HTML 继承下来的习惯，不能好像真的 XML 那样自以为符合标准了就随意写。不信？那么再试一个吧： hello
</br> world ，留意 IE 与 Opera 中的显示效果。

有部分读者认为我举的例子是不符合 XHTML 规范的，那么请先阅读 XHTML 规范。Empty Elements 一节的中文翻译如下："空元素必须要么有一个结束标记，要么以/>结束，例如
或<hr></hr>。请参考 HTML 兼容性标准以获取关于确保向后兼容 HTML4 浏览器的信息。"可以看得到，规

范中也给出了<hr></hr>这样的例子，说明
</br>的写法是符合 XHTML 规范的，只是没有兼容 HTML4 标准。那么到底 XHTML 是否兼容 HTML4 呢？我们来看 Compatibility Issues 一节，中文翻译如下："虽然并没有要求 XHTML1.0 文档兼容现有的浏览器，但在实践中这并不难做到。"因此，XHTML 是没有规定文档必须向下兼容，我给出的例子都是合法的 XHTML 文档片段，当出现在完整的 XHTML 里面时也全部能通过 W3C Markup Validation Service 的验证。

其实我写这篇文章的目的不是为了强调只符合 XHTML 规范就行了，也不是强调符合 XHTML 同时兼容 HTML4 就够了，而是应该考虑更多需要兼容的情况。例如你的 CMS 中允许用户提交 HTML，提交的 HTML 经过 SgmlReader 或者其他方法格式化为 XHTML，同时或许还做了其他 XML 处理，这时候就有可能将用户提交的<textarea></textarea>转换为<textarea />，这种情况下你需要通过跟踪调试，找出问题并不容易，因为 XML 处理并没有违反任何规范，每一步的处理都是符合语义的。另外最好不要把
写成
，因为确实有些弱智的 parser 仅仅因为少了一个空格就无法正确识别。

Web 标准不标准

作者：阿一（杨正祎）[http://www.cnblogs.com/JustinYoung]

引言

一群会用 Table 蹩脚布局的网页初学者嘲笑着那些对网页制作一窍不通的门外汉；而一群自认为 Table 布局无所不能的 Table 布局拥护者则嘲笑着那群用 Table 蹩脚布局的网页初学者；那些刚试着将几个页面中的 Table 换成 DIV 的所谓的 Web 标准设计者则嘲笑这那群死抱 Table 布局不放的 Table 布局设计者；而一群焦头烂额终于在网站上贴上“W3C 验证通过 HTML 网站”图标的自认高高人的 Web 标准设计者则嘲笑这那群以为“DIV+CSS”就是 Web 标准的 Web 标准设计初学者；但是当我们把我们的网页放在不同的浏览器中的时候，却发现我们全部都被“Web 标准设计”嘲笑了……

正文

标准：衡量事物的准则。例句：惟极贫无依，则械系不稍宽，为标准以警其余。—清·方苞《狱中杂记》

Standard: An acknowledged measure of comparison for quantitative or qualitative value; a criterion.

无论是古今中外，对于标准一词的解释都很相近，即标准是一个准则。那么冠以“标准”前缀的词，则必须符合此准则。如：

标准大气压（standard atmospheric pressure）：指在纬度为 45°的海平面上，温度为 0°C 时的大气压，相当于 76cm 高的水银柱所产生的压强。

标准照（official portrait）：指人的正面半身免冠相片。

那么冠以“标准”前缀的 Web 标准设计，也必须有一个衡量的准则方可。但是我们发现所谓的衡量 Web 标准是否标准的准则并不存在，至少目前还没有成型。没有衡量准则，标准何以言之为标准？

推荐遵循标准不是严格意义上的标准

当你试图在网上查找“什么是 Web 标准”时，找到的多数是将 Web 标准引入中国的先驱阿捷的文章《什么是 Web 标准》。虽然里面提到了各个组成部分的“推荐”遵循标准，但是那些也都只是 W3C 组织“推荐遵循”的标准。大家都知道 W3C 标准不是强制性标准，所以像微软这样喜欢“卖标准”的一流公司并不完全买 W3C 的账。所以那些所谓的“推荐标准”从严格意义上来说并不是全面的、严格意义上的认证标准。

注： 虽然 W3C 的推荐标准不是严格意义上的标准，但是我们绝对不可以无视。因为它们的确很有指导意义。而且我们非常欣慰地看到，虽然 IE 7 为了向前兼容，保留了一些 IE 6 的诡异解析方式，但是它也正在逐渐地遵循这些标准。所以这些标准现在是“推荐遵循标准”，但是在不久的将来极有可能成为真正的标准准则，那个时候就是“Web 标准大统一”的黄金年代。所以还没有看过下面这些文章的朋友要抓紧时间啦。

W3C 推荐遵循的 Web 标准文档列表。

- XML 方面：《Extensible Markup Language (XML) 1.0 (Second Edition)》[①]
- XHTML 方面：《XHTML 1_0 The Extensible HyperText Markup Language (Second Edition)》[②]
- DOM 方面：《W3C DOM 规范》[③]
- CSS 方面：《Cascading Style Sheets, level 2 CSS2 Specification ver.12-May-1998》[④]
- Script 方面：《ECMAScriptLanguage Specification》[⑤]

Web 标准与否不能被有效验证

即使我们暂且将那些“推荐遵循标准”视为标准，我们仍然有很多问题。例如，怎么检验我们的页面就是符合那些“推荐遵循标准”的呢？的确，我们有验证工具，但是那些验证工具的验证结果就是正确的吗？Web 标准一个重要的部分就是根据标签的语义来使用它们。

---

① http://www.w3.org/TR/2000/REC-xml-2000/006.html
② http://www.w3.org/TR/xhtml1
③ http://www.w3.org/DOM/
④ http://www.w3.org/TR/CSS2/
⑤ http://www.ecma-international.org/publications/standards/Ecma-262.html

例如 Table 是用来呈现表格数据的，而不是用来分割文档的（也就是平时说的布局）。但是一些利用 Table 布局的页面也能顺利通过验证工具的验证。所以那些验证工具的验证结果也只能作为一种参考。

语义是道多选题

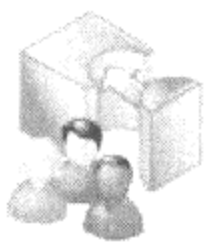
说道标签的语义，这也会带出一些问题。因为当我们决定使用哪种标签的时候，我们会发现我们面对的是多选题，而不是单选题，因为某些情况利用多种标签时都是符合语义的。我们举个最简单的例子：

你说“Name:”应该用什么标签包起来？Label？SPAN？其实无论使用哪个都是符合各自的语义的。正是这种“多选题”让 Web 标准显得有些朦胧。PS：下面五个网址可以从这里得到：http://www.cnblogs.com/JustinYoung/archive/2007/07/22/827602.html

树欲静而风不止

即使上面的问题都不再是问题了，我们仍然有一个大问题——各种浏览器对 Web 标准的支持不统一。也正是这种标准不统一的大环境让 Web 标准显得缥缈、遥不可及。看到论坛上天天有人在骂“垃圾 FF”、“垃圾 IE 7”、“垃圾 IE 6”……纵观世界，能生存下来的只有两种生物——改变环境以适应自身发展的，例如人类；改变自身以适应环境发展的，例如变色龙。当我们不能改变这个世界的时候，那么就改变自己去适应这个改变的世界吧。

有限的资源，努力地去做



Web 标准不标准，至少是现在没有统一的标准，但是网页还是要做，工程还是要赶。我们能做的只能是合理利用手头的有限的资源，努力地去做到最好。这句话包含了两个方面的含义：有限的资源和努力去做。

有限的资源：

当你躺在象牙塔里，努力地让你的个人网站贴上 W3C 认证标签的时候，你是无可厚非的，因为你的资源是那么地充足，时间、青春、没有项目时间的催促，没有客户的最后通牒。但是当你真正在做商业项目的时候，你会发现我们能利用的资源是极其有限的。人力资源、物质资源和时间资源都是那么捉襟见肘。当你有分配这些资源的权利的时候，你才会发现分配这些资源的责任。我们的项目绝对不会为了那张 W3C 认证标签而浪费宝贵的资源。

努力地去做：

但是有限的资源绝对不是粗制滥造的借口。事实上我们项目对于质量是极其重视的。项目经理提出的“质量是项目之本”的结论已经在我们团队达成共识。给测试部门的测试资源，我们向来都安排在整个工程的 1/5 以上（分析设计过程 3/5 以上，代码 coding1/5 以下，其余的给测试部）。

但是努力去做，到底要做成什么样？其实也并不是没有参考。在 Andy Budd 的《样式指南示例》中我们可以看到一些国际上对此问题的处理方法的端倪，他在 1.4 小节提到了 Browser Support 的概念。然后他提出了对不同浏览器支持的不同程度。

- Target – Most popular browsers at present. Everything must work as intended.
- Supported – Old but popular browser. All content and functionality must work with minimal degradation.
- Partially supported – Old and buggy browsers. Not supported but not officially unsupported. Content and functionality must work. Degradation must be graceful and should not obscure content.
- Unsupported – Buggy and unsupported browsers. Advice current users to upgrade.

我认为这是正确的做法，选定主流的浏览器，然后对主流的浏览器进行 Target 支持，对于次之的浏览器进行 Supported 支持，而对于一些老版本的浏览器进行 Partially Supported 甚至完全不去管。这样就能最大可能地让网站被正常浏览。当然将哪些浏览器、哪些版本进行 Target 支持，需要根据项目的使用群来决定。例如我们最近的一个日本母公司内部使用的 B/S 系统，因为我们知道他们那边使用者的电脑软件配置情况，所以我们进行了如下的浏览器支持定义：

- IE 7+：Target
- IE 6+：Supported
- FF ：Partially Supported
- Other：Partially Supported or Unsupported

正是有了这个浏览器支持表单，让我们调画面的时候有的放矢，有所侧重，而不是一味地让所有的浏览器都正常支持而浪费时间（而且让所有的浏览器正常支持也只能是一种理想状态）。

Web 标准难不难?

没进来的人说：很难!

刚站在门框上的人说：不过如此。

站在门里面的人则说：Web 标准的路还有很长一段要走……

走在 Web 标准化设计的路上[唠叨先]

作者：爆牙齿[http://www.cnblogs.com/yuntian]

对自己没信心了，不指望能集中续写《重构之美》，还是散开记录比较好。

看见一个评论这么说："晕，现在才谈 XHTML 是不是太晚了点，这东东 2004 年就火了一把了。"其实我觉得，作为一项技术，没有火与不火的说法，也没有早与晚的说法。技术的生命力和火没有关系的，不知道、不理解、没学会，怎么都不晚。再说了，Web 标准这个东西再过几个月就是进入中国两年的时间段了（这里我们暂且以傅捷所译《网站重构》2004 年 5 月出版这事作为一个起点好吧）。两年的时间，你敢说 Web 标准已经全面推广开了吗？

有多少人还不知道？

有多少人只知道"Web 标准"这个词？

有多少人正在学习？

有多少人把 Web 标准理解为 DIV+CSS 或者是 ul/li 代替 Table/tr/td？

有多少人把 Web 标准理解为 CSS 的崛起？

……

我不知道，但是我去年底到目前的公司，40 多名研发人员没一个知道 Web 标准。当我把 Web 标准带到他们面前，他们的理解只是我的 CSS 很强……恰恰相反，我个人觉得我自己的 CSS 水平很一般，本来 CSS 就没太多的技术含量，都是些技巧的东西。七八年的思维模式哪那么容易就改过来了？HTML 也倔得要命，要连根拔出也不是件容易的事，所以 Web 标准之路还长着呢。我琢磨着 xhtml 2.0 发布的时候，估计 xhtml 1.0 都在国内普及不开来，或者说不能正确地被普及开来。

既然叫随想随说，我就懒得整理了，想到哪里说到哪里，不分先后。我说出我的一些个人理解，正确与否自行判断吧，如果觉得我有错，请无情地指出，谢谢！

啊哟，回头想想，快两年了，好快！不过就眨了眨眼。

——2006-2-22

最后决定就将这次散开记录的系列文章由最初的《随想随说，正确理解与运用 Web 标准》重新命名为《重构之美——走在 Web 标准化设计的路上》。

本来我取名为“正确理解与应用”其实有赚眼球的不良动机。因为我确实仍处于理解中，并不能保证是绝对正确，而且很多文章我都会以一种求证的态度表达我自己的理解，这不已经有人指出：你完全不熟悉 xhtml。虽然他没说理由，或许是随口一说，但仍让我感到汗颜，所以还是把翘起来的尾巴夹住，以免出现一看标题就想攻击的情况。

不过我已经不间断地使用 Web 标准近两年时间，在多个项目内多次运用于实践，现在作为发起者将对目前所在公司内 40 人规模的研发团队进行 Web 标准全面应用部署系列培训，并主导实施。所以我觉得，我应该不算初学者了，理解可能不一定正确，但至少不会肤浅。我希望每一个对我提出意见，甚至抨击我的朋友，对自己的话负责任，拿出您的理由来我们讨论，如果确实您正确，我立刻修改文章并注明您的名字和链接。对于不负责任的评论，将在一天后删除。希望大家评论时多敲一点字完整说出您的意见，类似于“好!”“顶!”“不好!”“乱说!”之类毫无意义的评论我也会随时整理删除，因为如果您做无意义的评论还真不如保持安静，谢谢。

如果说《重构之美——迎接网站标准化设计的来临》是一篇 Web 标准入门的文章，做到了形似，那么时隔近 1 年半，我希望和大家一起讨论，交换意见，努力地走近神似，所以取名为《重构之美——走在 Web 标准化设计的路上》，我们都在路上。这次的核心不在技术技巧上，而是像老标题一样，我希望尽量在“正确理解与运用”上做文章，在思路上做文章。

——2006-2-24

走在 Web 标准化设计的路上

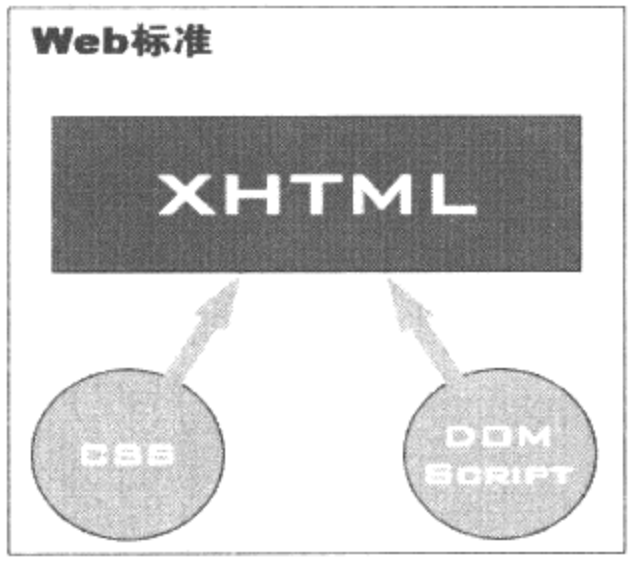
——振臂一呼：CSS，Stop!

作者：爆牙齿 [http://www.cnblogs.com/yuntian]

这个副标题让我琢磨了很久，和之前的“随想随说”不一样，重新命名为《重构之美》后就给了我压力，让我认真对待仔细斟酌，这样其实也好。

——2006-2-25

Web 标准在概念描述上涵盖了三个部分：结构[xhtml]、表现[CSS]和交互[DOM、ECMAScript]，准确的定义我就不摘抄凑字数了，百度、google 上遍地都是。这三个部分我认为并非处于同一个等级，xhtml 是最重要的部分，是第一级，而 CSS 和 Script 则并列处于第二级，如下图例所示。



我认为不要小看了这个认识，目前很多人都没有意识到的问题，即便意识到了，行为上也没有跟上。怎么说呢？Script 不是我所擅长的，所以我基本上不会涉及 Web 标准中交互这部分，即便涉及也只是很浅，个人能力有限。CSS 部分会有针对性地涉及，但不会很多，因为我不想在 CSS 上做太多的文章，因为我感觉现在国内 Web 标准界对 CSS 的追捧有点过了，介绍 Web 标准的网站和书籍主要都是在介绍 CSS 的各种技巧，而对于 xhtml 部分的介绍很少，也就泛泛提及用 DIV 代替 Table 进行布局和书写规则，多一点的会提到语义。

有没有深入地理解过？为什么要严格书写？我想大部分的答案是通过认证。再问，为什么要通过认证？答不出来了？好，再来，又为什么要严格书写？又是认证？这不扯蛋嘛！鬼大爷管你认证与否。那么严格书写需要吗？不需要

吗？再来说语义，说起来估计还有很多“Web 标准”者连语义这两个字都不知道。我认为语义是 xhtml 的两个核心之一，另外一个核心就是今天要谈到的结构。比如对表格 Table 的使用，都是这么说的：表状数据还是要用 Table 标记。那么有没有想过什么样的数据是属于表状数据？我说把一个三栏式布局的页面视为一行三列表状数据行不行？我是在扯蛋，那么什么是表状数据？什么时候用 Table？现在网上关于 xhtml 语义理解的文章真的很少，为什么？CSS 啊，从上到下都追捧 CSS 去了，以至于那天我在蓝色理想上见回帖：学 DIV+CSS，但不准备遵守 xhtml……类似的还有很多，什么花样都有。无语中，我想每个真正理解了 Web 标准的人都会很无奈地摇头，近两年 Web 标准的推广演变为 CSS 的推广。CSS 很重要吗？不重要吗？我说不要 CSS 行不行？你找一大堆完全合理的理由……“行不行？”“行！”那就对了，我说不要你的 CSS，我要他的 CSS，又行不行？那么和 xhtml 相比，CSS 重要在哪里？

最后我们来说说关于“用 DIV 代替 Table 进行布局”这种说法，这么说吧，如果你是抱着这种思路使用 DIV，我认为是错误的，布局这个概念其实是 Table 带来的，如果你又把布局加到对 DIV 的理解中去，那么对不起，你还是一个“Table 者”。最典型的，有位朋友针对我上一篇[复杂表单]评论到：你这个表单看似复杂，其实很简单，不过左右两列式布局，左二右六……他还提到了“拼装”两个字，然后说我的代码不过是用 DIV 代替 Table，说我是 Table 思路。看看他对页面的分析，“左右两列”、“左二右六”和“拼装”，多么熟悉啊，即便他用 DIV 实现了这样的布局，你认为他抛开了 Table 吗？所以我说他完全没看懂我的代码。我只听说过“不要使用 Table 布局”，没有在很官方的地方看见过“用 DIV 代替 Table 进行布局”这种说法，都是人为造出来的，或许是为了更好推广 Web 标准，但是现在我们要知道，这种说法是错误的！DIV 从来不是布局元素，也没有哪个标记是布局元素。

像上面的图示，xhtml 是根基，表现和交互虽然也很重要，但毕竟可以不要表现，也可以不要交互，但是不能不要 xhtml，所以，现在狂热的追捧 CSS，几乎达到忽略 xhtml 这个根基的环境下（比如上面我说的那个回帖），我要站出来，振臂一呼：CSS，Stop!（不知道有多少人响应我，鄙视我也欢迎，当我是疯子一笑而过也可以。）

本来打算简单说说 Web 标准的概念和对现状的不满，然后专心写[深入结构：理解 h 系列的不合理。]，结果说了这么多，也好，换个标题发布，[深入结构：理解 h 系列的不合理]放到下篇来写。

——2006-2-27

走在 Web 标准化设计的路上

——对 HTML/XHTML/XML/XSL 的一些认识

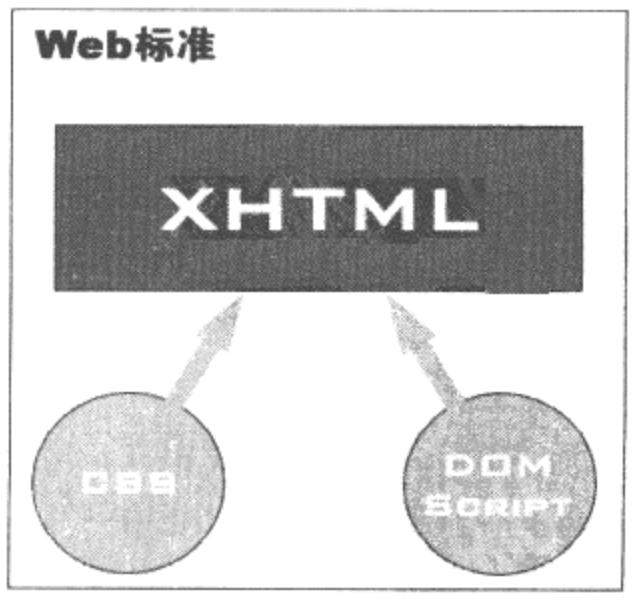
作者：爆牙齿 [http://www.cnblogs.com/yuntian]

xhtml 1.x 一定是 html，而 html 则不一定是 xhtml 1.x。如果说 IE 不支持 xhtml，就等于说 IE 不支持 html，不会吧？所以 IE 不支持 xhtml 一说不知何来之有，不过 IE 不完全支持 CSS 倒是真的，但是 CSS 是表现，表现和结构是无关的。xhtml 的产生正是因为在 html 中，表现和结构混为一团，不利于向 xml 平稳过渡。

如果要说担心，xhtml 2.0 倒值得担心，因为有很多新的东西被加入进来，就需要各浏览器作出相应的支持，不像目前 xhtml1.x，浏览器不用变化。但是这种担心或许有点早和多余了。首先，xhtml 2.0 还处于草案的提议与设计阶段，而 1.x 也还没有被普及开来，时间还长着呢。其次，即便 xhtml 2.0 确定被推荐，由于不能完全向下兼容，还需要等浏览器跟上节奏，这又是一长段时间。所以完全不用着急，我认为，即便 xhtml 只是被视为一种过渡技术，但是仍然有着相当的生命力。

说到这里，想起曾经看过这么一篇文章《了解了 XHTML2.0 后，有感》（实在找不到原出处），文中最后提出的观点：我的意见是，直接做 XML 的网页！这样的观点我是不认同的。

再来说表现与结构，老生长谈了，两年前我就推荐过阿捷写的一篇《理解表现与结构相分离》，今天我仍要不遗余力地推荐！文中写出了 4 个部分：数据、结构、表现和行为。上次我给出一个简单的关系图：



现在我再给出一个更为详细的关系图：

数据和结构是无法分割的整体，脱离了结构的数据几乎不能使用。所以纯数据需要用 xhtml 或者 xml 来格式化，展示其结构。我这里着重谈谈结构。在我的理解里，结构目前划分为两部分，一是语义结构，二是代码结构。语义结构是靠语义产生，代码结构则是面向程序的。XML 拥有完美的代码结构，但是却因为高度的可扩展和自定义性，很难拥有语义结构，除非在它的基础上定义一个通用的格式，比如现在很火的 RSS。所以在目前通过浏览器上网浏览 html 的模式下，直接应用 xml 写网页是无法通用的，也是困难的。而 xhtml 则是一种折中，它不允许扩展，从而继承了 html 的语义性，拥有现代浏览器都可以识别的语义结构以适应目前互联网应用的大环境，同时用 xml 的规则规范它，让它继承完美的代码结构以便顺利过渡。所以我说 CSS 相对不重要，Web 2.0 时代的一个标志就是数据跟着用户走，这里的数据当然包括结构（语义结构和代码结构）。再说了，CSS 这个东西专心学习，我想一个月足以精通，但是并不代表你页面就能做得很漂亮了，那是设计，和 CSS 无关，你敢说 1 个月精通设计吗？我做了 6 年设计了，仍觉得欠缺太多，设计难啊！所以从重要性上，CSS 比不上 xhtml，从技术含金量上，CSS 比不上设计。呵呵，好像把 CSS 说得一无是处了，我错了我错了，不要骂我了。

至于所谓的 xml+xsL，并不是 xhtml + CSS 的升级版本，XSL 的意义在于转换而非控制表现，xml 过于开放了，所以需要 XSL 来转换，将 xml 中的语

义结构和代码结构转换成不同领域相应的标准结构，比如移动中的 WML 又或者 Web 中的 XHTML。所以不要说我要 xml+XSL，若你不懂 xhtml，转换出看起来像 xhtml 的 html，貌合神离又有什么意义呢?

最后，我认为 xhtml 一定会代替 html（XHTML is aimed to replace HTML），但是不一定被 xml 所代替，或者就如上所说，有相当的生命力。xml 只有数据和结构，它的语义是面向程序的，而不是面向浏览器。如果要全面 xml 化，首先使用浏览器浏览网页这种上网模式就一定要先转变，再或者浏览器本身的内核会有较大的改变甚至需要重新设计。这两个都不是一蹴而就的事情，所以 xml 的意义更多在于桥梁，要作为主体目前还不行，不管作为数据的主体（有数据库）还是结构的主体（有 xhtml）。因此即便 xhtml 被视为过渡技术，2000 年出生以来，经历了 1.0 和 1.1，现在仍然在向 2.0 发展，直到所有条件成熟，xml 能够全面接管过来。而 xhtml 的作用就是等到 xml 能够接管之时，顺利平稳过渡。我个人认为现在谈过渡还早，2003 年说 99%是过时的，现在呢? 2006 年了，纠正了多少? 我想 xhtml 会再存在 3、5 年或者更久。不过互联网谁又知道呢，也许明天一切都变了。

走在 Web 标准化设计的路上

——深入结构：理解 h 系列的不合理

作者：爆牙齿 [http://www.cnblogs.com/yuntian]

回头看上篇文章，确实有点乱，感觉自己像个愤青，怨妇一样。我的意思是不要过多的把关注投向 CSS，CSS 并非 Web 标准最核心的东西。我认为 Web 标准的核心在从 html 到 xhtml 这个看似变化很小却意义非凡的事情上。Web 标准更多的是思想的变革和思想的重构，而不是技术，这一点几乎全部体现在 xhtml 上。

菩提树朋友在评论中做了个比喻：

“XHTML 像是一块白肉，不能吃，就算能吃，吃起来也是十一分难吃。CSS 就是那酱料，沾着吃才有味，Script 就好比加了一道火，烤一下再吃，原来完全不一样。”

非常感谢你，这个比喻非常好，不过理解重心不一样。

还是这个比喻，我把 xhtml 视为牛排好不好？一份牛排好吃与否关键在牛肉本身的品质，其价值也体现在这里。甚至牛肉本身的品质可以好到任何外部因素都显多余，生吃是最好的选择。不知道你喜欢烹饪吗？我比较喜欢。首先要肉好菜好、新鲜，其次才是调料、火候、技巧。一堆好菜，不用费什么功夫就能得到很好的效果，而一堆烂菜，怎么都掩饰不住本质的缺陷。任何厨师都最重视原材料的筛选，事半功倍。话说回来，生存第一位，所以一块白肉哪怕再难吃也强过调料，没别的意思，只是为了说明轻重，应该首先重视 xhtml 的品质，其次才是 CSS 的技巧。By The Way，我很喜欢吃牛排，而且只要 3 成熟。

我还是继续说下去吧，或许听完后再回头看，就能理解我为什么说：CSS，Stop！并非要否定 CSS，而是指对 CSS 的过度追捧 Stop。

问题：HTML 中的 6 个标题 Tag（h1/h2/h3/h4/h5/h6）设计是否合理？理由？解决办法？

这是我在培训中提出的两个问题之一，不知道大家有没有考虑过？或许你要说这个不是我们该考虑的问题，这个是 W3 的工作。我承认，其实我最初也没主动去考虑过，直到有一天接触了 xhtml2.0，我才知道 h 系列标题的设计

是不合理的，在 xhtml2.0 中不推荐使用<hx>系列 Tag，理由是结构不好，推荐使用这样更合理的结构进行代替：

```
<section>
    <h>
    <section>
        <h>
        <section>
            <h>
        </section>
    </section>
</section>
```

嗯，好像答案都有了，不合理，结构不好，解决方法如上。其实我想问的是为什么 hx 系列结构不好？为什么要这样改？现在你先别急着向下看，试试想想。

琢磨这个问题是因为我想试着去理解 W3 到底想干什么。现在我们清楚了为什么在 XHTML 1.1 中不认可类似 font 之类的 Tag，XHTML 1.1 是 2001 年的产物，如果当时我们就理解了，那么 Web 标准早就推广开来了。现在 W3 如法炮制，在 2.0 中不推荐现有的 h 系列 Tag，那么会不会在“2.1”中完全抛弃它，如同抛弃 font 一样。XHTML 2.0 尚处于草稿和提议当中，还未被 W3 正式推荐，当然我无意从中引出 Web 大标准、超标准之类的东西，我只是希望能对目前的 Web 标准加深理解有所帮助，实际上我得到了较大启发或者说之前的一些模糊的、不确定的感觉变清晰了，所以我认为这是一个对于目前 Web 标准非常经典的问题。

好，现在说说我的理解。

首先我们想想 h1/h2/h3/h4/h5/h6 这 6 个 Tag 是什么东西，当然是标题 Tag。它们有什么不同？h1 页面唯一（一个页面只能存在一个 h1），并且字体最大。h2~h6 可以多个，字体逐步减小。曾经有个朋友说他认为 h 系列都是和表现相关的东西，有可能在以后的 XHTML 中被删除，所以他从来不用也搞不懂为什么那么多人用。其实他搞错了开头，却猜中了结局。确实已经在 2.0 中不被推荐，却不是因为表现而是因为结构。不知道还有没有人和他有同样看法？h 系列绝对不是类似 font 的表现 Tag，如果因为粗体和大小就算的话，那么所有标签都属于表现标签了，ul 还有点，ol 还有数字呢，p 有间距，DIV 会换行……那么还需要 XHTML 干嘛呢？直接 XML 好了。

XML 里的 X 是指可扩展，也就是可以随意定制标签。而 XHTML 中的 X 虽然也叫 Extensible 可扩展，但是并非可扩展，你不能为 XHTML 自定义标签，它更多是指 XML 化，也就是 XML 化的 HTML。所以，XHTML 每个标签都

需要有默认的样式来支撑它的语义化，这些样式是最基础的样式，为什么 h1 和 h2 不一样？因为它们默认样式不一样。为什么两个 DIV 会形成两行，而两个 SPAN 却在一行上？因为 DIV 默认一个 display:block 的样式。具体的大家有兴趣可以查一下。好了，hx 系列 Tag 的不合理就出在这里——语义和结构。h1～h6 的语义通通都是一样的，那就是标题。他们之间有没有结构？有的。同样的语义却设计了 6 个 Tag，为什么？因为结构的存在。hx 系列之间的结构（节，俗点叫上下级关系）是靠 123456（不同的默认样式来决定），但是这些数字并没有语义的，你说 1 大还是 6 大？不同的环境下截然相反，人的判断都如此，作为程序是无法理解凭什么 2 就是 1 的子级，3 就是 2 的子级。再有，假如一篇极端的文档需要用到 h7，h8……怎么办呢？所以有了以上的改变，语义用 h 表示，把结构分离出来用 section 表示，这样一来，程序喜欢不喜欢，我想不用多说了。^_^

——2006-2-27

很久以前，我第一次看到 IBM 网站上《Web 的未来：XHTML 2.0》[①]这篇译文时，第一个反应是怎么把标题 Tag 搞得这么复杂，再定睛一看，马上就联想到自己的迷惑和一直在追求的东西，真的立刻就有一种醍醐灌顶、豁然开朗、近视眼戴上眼镜的感觉。其实早在一年多前，我就有过这样的迷惑：如何固定 xhtml 文档？后来还在《重构之美——迎接 Web 标准化设计的来临[总结一：网页设计回归？][②]》中对这个问题做过这样的总结：

...2、根据规划完成 XHTML 文档，组织好文档结构，设计纯文档。这里我要提醒，纯文档同样具有 UE，它只是没有了 UI 而已，所以需要仔细推敲标记的选用并确定下最简洁的 XHTML 文档。...4、根据设计稿为 XHTML 文档添加样式进行还原，通过样式表的设计技巧尽可能地不修改 XHTML，如果 UI 实在是复杂，则可以在不影响 XHTML 文档结构的情况下加入一些额外的标记或者进行一些嵌套……

可以看到当时的我其实是迷惑的，怎样选用标记？怎样的 xhtml 文档最简洁又灵活？我一直在努力，却始终好像很难找到一个方法固定 xhtml。对于一个页面，不同的人有不同的分析，不同的分析又将产生不同的 xhtml 文档。哪怕两人在不加载样式表情况下设计出浏览界面完全一样的页面，背后也可能因为分析的不同得到两份不同结构的 xhtml，也就是语义一样，但结构不一样。语义含有部分结构的概念，却不等于是结构。

很早就开始关心纯文档的结构，从最初的关注浏览器中 xhtml 的浏览结构，慢慢也开始关注 xhtml 的代码结构。简单说明一下这两个结构，希望大家

① http://www-128.ibm.com/developerworks/cn/xml/x-wa-xhtml/index.html
② http://www.cnblogs.com/yuntian/articles/316550.html

能够多关注浏览结构，更多关注代码结构，而不是仅仅把目光注视在应用了CSS 后的最终页面效果上。比如：

```
<h1>文档名</h1>
<h2>标题 1</h2>
<h3>标题 1.1</h3>
<h3>标题 1.2</h3>
<h4>标题 1.2.1</h4>
<h4>标题 1.2.2</h4>
<h3>标题 1.3</h3>
<h2>标题 2</h2>
<h3>标题 2.1</h3>
```

这段代码的浏览效果如下：

文档名

标题1

标题1.1

标题1.2

标题1.2.1

标题1.2.2

标题1.3

标题2

标题2.2

我们可以看到，浏览已经完整了，有了结构，这种结构是通过语义而产生，然而代码结构完整吗？所以我说过，语义带有部分结构的含义，却不等于结构。完整的代码结构我认为如下：

```
<h1>文档名</h1>
<div>
    <h2>标题 1</h2>
    <div>
```

```
        <h3>标题 1.1</h3>
        <h3>标题 1.2</h3>
        <div>
            <h4>标题 1.2.1</h4>
            <h4>标题 1.2.2</h4>
        </div>
        <h3>标题 1.3</h3>
    </div>
    <h2>标题 2</h2>
    <div>
        <h3>标题 2.2</h3>
    </div>
</div>
```

上面的代码还不是很合理，有心的朋友应该能看出来，不过意思到了就好。

这有没有觉得这段代码眼熟？回头看看开头 xhtml 2.0 中推荐的标题表达形式。是的，当我看见 xhtml 2.0 的时候马上想起的就是我的这种写法，然后更加坚定地将这种写法延伸使用下去，并且发现似乎找到了固定 xhtml 的方法：用标题划分代码结构。这样对于同一个页面，不同的人都能写出同样的 xhtml 结构。例如对于大部分页面，都可以视为页头、内容和页脚三部分，那么好了，每个人都能写出一模一样的 xhtml 大结构，而不管页面具体是如何设计的，看看：

```
<h1>网站标题</h1>
<div>
    <h2>页头</h2>
    <h2>内容</h2>
    <h2>页脚</h2>
</div>
```

或许你会说，这样简单了，那么你可以去看我在[复杂表格]中的几段代码，或者看《粗略整理博客园的页面 Xhtml 代码》[③]，统统是这样的结构。

这样写优势太多了，因为它不论在哪一方面都吻合 xml 的要求，不论在写法上还是结构上，程序都会非常喜欢，比如无论通过 JS 还是后台编程，我们都可以很轻松地通过 xpath 查询提出所有的 h2 或者任何部分。我想这也是 xhtml 的目标，让 html 尽可能靠向 xml。xhtml 会代替 html，但是我想 xml 不会代替 xhtml，至少在浏览器没变革前，xml 没有直接的语义，只有结构。xml

[③] http://www.cnblogs.com/yuntian/archive/2006/02/17/332849.html

这里就不多说了。这种写法是有缺点的，那就是会感觉 h 不够用，6 个 h 标签，只有 3 个可用，h4 已经显小。那么当我们的结构层次高于 3 层或者高于 4 层的时候就麻烦了，所以开始向往 xhtml 2.0，可以无限层地表达下去，虽然还不清楚它怎么来区分每个 h 标签。

这样的写法很接近一个结构很好的 Word 文档。对，就是以写 Word 文档的方式来写 xhtml 页面，固定下来。

好了，现在你理解了吗？认可吗？

下一篇继续深入结构，咱们聊聊关于 DIV 和 SPAN。

——2008-3-8

走在 Web 标准化设计的路上

——深入结构：合理运用 DIV 和 SPAN

作者：爆牙齿[http://www.cnblogs.com/yuntian]

特意上网搜索了一下，关于 DIV，说法很多。

把 DIV 看成是布局元素这种观点我想是最多的，类似有“用 DIV 代替 Table 进行布局”和“实战 CSS+DIV 布局”，等等，太多了。还有不少人沿用 Dreamweaver 的定义，称 DIV 为层，按 Photoshop 的层的概念来使用……有朋友干脆就直接称 DIV 和 SPAN 为辅助布局元素。

怎么说呢？虽然我很想说对 DIV 类似的这种认识是错误的，DIV 不是一个布局元素，没有一个 tag 是用来布局的，但是我是对的吗？我也不知道。几乎所有人对 DIV 的宣传都是布局，不管是“民间”的还是“官方”的。如果我们找根源，中文中 DIV 是一个结构化标签，是一个块级元素。我们首先看看 DIV 拥有的语义，DIVision（分隔），按语义它的作用是将两个部分分隔开来。然后我们再回到 W3 去看看怎么定义 DIV 和 SPAN 的：The DIV and SPAN elements, in conjunction with the id and class attributes, offer a generic mechanism **for adding structure to documents.** These elements define content to be inline (SPAN) or block-level (DIV) but impose no other presentational idioms on the content.

注意到上面加粗的话了吗？W3 可没说是 for layout，而是 for structure，是结构！因为分隔从而产生（定义）一个代码结构。结构和布局应该是两个概念。或许因为 Table 确实被用于布局了，所以这种根深蒂固的布局思路又自然而然地转嫁到 DIV 上，我曾在很长一段时间里也是这么理解的。但是，现在我要说，这绝对是一个错误，并且这是极度严重的错误!!! 这纯粹个人观点个人理解，自己取舍好了。

为什么严重？理解的错误直接导致的就是使用的错误。因为如果按照这个思路，把 DIV 作为布局元素使用，那么你永远无法固定 xhtml！永远陷在 CSS 的怪圈中！永远不会去思考和理解结构！永远擦不干净 Table 烙下的痕迹！永远无法接近神（貌合神离的神哈，呵呵）……

或许把 DIV 称为布局元素还是为了更好地推行标准，但是却将人们从一

个错误带向了另一个错误。两年前我刚接触标准时就在《重构之美》首篇中迷惑过关于改版的事情，虽然随着理解的深入好像有了突破，在我写下 xhtml 后不变动，然后通过 CSS 的技巧来完成新版面，比如像著名的 CSSzengarden。但是很快我又有了新的迷惑，一个人这样做好像没什么问题，团队呢？如果同样的内容设计成两个版式，然后交给两个人来写 xhtml，会一样吗？就像如果把 CSSzengarden 的形式颠倒一下，基于同一份数据先做好 100 个设计稿，让 100 个人按照这个设计稿写 100 份 xhtml，会一样吗？按照 DIV 布局模式，对于同样的版式，不同人不同的页面分析都会产生不同的 xhtml，更何况不同的版式呢？但是既然表现与结构无关，那么同样的内容不应该有 2 份以上的 xhtml。不要小看这个问题，对于团队中前后台的有效分离与快速协同，这是关键！我在培训中提出一个观点：**最理想的境界是前台闭着眼睛都能知道后台输出的是什么样的 xhtml 结构代码**。那么问题出在哪里？DIV 布局！尤其是在理解了 h 系列标签不合理之后，体会更深刻。

上篇文章我提出的关于结构应当分为语义结构和代码结构两种。理解了这两个结构之后，DIV 的用处就比较明朗了，稍稍动动脑筋就能想到，用于组织代码结构。所以 hx 标签的问题我认为经典。不要说 html 了，即便对于 xhtml，大部分人关心的仍是如何表现，小部分人关心语义结构，很少人去关心代码结构，似乎 xml 有了，xhtml 就不需要代码结构了。但是从 hx 系列的问题可以看出并延伸知道 W3 可一直在关心代码结构，从 1.0、1.1 直到 2.0，一直希望 xhtml 拥有 xml 般严谨的代码结构。说到这里再多看 xhtml 2.0 的另一个变化，br 不再被推荐，应该很好理解了，br 的语义是产生一个截断(break)，但实际作用是产生一个行，语义结构上仍不完美，所以使用 line 进行替代 <line>this is one line</line>。同样 br 也无代码结构可言，如果想提取第三行的数据如何操作？所以很有可能类似 br、hr 这类标签都将被废弃。我琢磨着，xhtml 1.x 是 W3 清理表现，将人们往语义结构[Semantic]的方向牵引，而 xhtml 2.0 则是展示和突出代码结构[structure]。

那么怎么组织？首先对于一个设计稿，一定不要被设计所迷惑和左右，只提取看得见和看不见的数据，然后就扔掉设计稿，先完成数据的语义结构，再添加代码结构（**adding structure to documents.**），完成 xhtml，最后一步才是重新拾起设计稿打开 CSS，还原。当然实际做的时候不可能不看设计稿。但是怎么看？只提数据！再说一点，数据在文档中的先后顺序由什么定？当然是由文档而定，不是由设计稿所定。举个例子，假如有两个栏目：新闻头条和普通新闻。谁在前谁在后，很显然在文档中应该是头条在前普通在后，这是由 UE（用户体验）和栏目轻重的综合考虑决定的。但是按照 DIV 布局的话，是按照设计稿上前下后、左前右后的顺序来决定的。如果设计稿中将普通新闻栏目设计在左栏，头条设计在中栏，文档中普通新闻就跑到头条新闻

上面去了。所以我打开一个 Web 标准站点文档浏览，如果文档的先后顺序是按照页面布局上前下后、左前右后的顺序而定的，那么我……再特例一点，如果一个单屏设计的网站，标题和导航设计在页面下方，那文档岂不是最下面才是标题和导航，这是什么 UE？这不是扯蛋嘛。DIV 布局的恶果是文档结构仍然在为表现所左右！貌合神离!!

代码结构怎么做？大处按照上篇文章所写，用 h 系列划分大结构。那么小处呢？这里就要牵涉进 DIV 的另外一个概念：块级元素。什么块？模块！用 DIV 模块化小处。举例：

```
<div>
    <h3><span>用户登录</span></h3>
    <div>
      <label for="name">用户名</label>
      <input id="name" />
    </div>
    <div>
       <label for="pw">密码</label>
       <input id="pw" />
    </div>
    <p><button /></p>
</div>
```

这是一个很简单的例子，我们来详细分析 DIV 在小处如何模块化运用。其实很简单，h3/lable/p 是语义结构，用户名和相应的输入框显然是不可分割的整体，DIV 将其标识为一个块，对应的密码部分同理。最后，两者一起与标题和按钮又构成一个不可分割的登录整体，DIV 之。这样就拥有了很好的语义结构和代码结构。好的代码结构不仅仅可以便于固定 xhtml，便于程序操作节点，还对 CSS 提供了很高的自由度。如上例结构，只需要给最外层 DIV 一个 class，比如“loginarea”。那么：

可以这么按节点/路径层层定义下去：.loginarea label{} .loginarea input{} .loginarea div label{} .loginarea div input{}。如果需要横向登录，只需要定义一个关键点：.loginarea DIV{float: left}，如果纵向则去掉这个关键点，模块化的登录就这么简单。这样还可以少写不少 class，尤其对于有些看似复杂的结构其实模块化设计好了，模块内部是简单的，一个路径定义过去，根本无需 class，还不会引起样式冲突和干扰，CSS 的可读性也很好。当然这里会涉及 CSS 的技巧，我认为 CSS 技巧最重要的就是分析页面。页面分析得好，写出来的 CSS 简单明了，充分利用 tag 还有多余的以备扩展，否则 class 一大堆复杂冗长还会觉得 tag 不够用，又去添加破坏结构。复杂表单那套系统的 CSS 大约写了 48K，还未做最后优化，全部图片总共只有 5K，还全是无损 PNG

格式。整套系统几十个大模块，又有无限级菜单、树、页签、复杂表单、合同、frame、iframe、报表、控件套控件等，CSS 加图片全部表现部分可以做到 50K 以内。这个项目四个程序员一起开发，我一个人负责所有前台，三个月时间程序员不管任何有关表现部分，我都是玩玩做做就搞定了。中后期，临着交付客户时候我还觉得公司提供的设计不好，又自己花 1 天重新设计，花不到两天另外写了一个 CSS，整个系统全变了且以前的设计未丢失。功能不变的情况下界面大换，再大的系统也不过一个人几天时间，且程序员不用管。这就是 Web 标准的威力之一！（因为是内网应用，所以我几乎没考虑浏览器兼容性，没必要，也是快的一个因素）

当前各大网站上以各种方式事先列出什么单行一列、两行一列诸如此类的几行几列的 DIV + CSS 布局代码，不好说他们不对，你完全可以去理解是如何使用 CSS 实现几行几列的布局，然后合理运用到自己的结构上。但是如果你按照他提供的代码去套、去添加内容，你就错了。不过话说回来，在被一篇一篇标题着斗大的“布局”两个字的潜移默化下，您还有心思去关心结构吗？所以很多人都去琢磨 CSS 了。这些善意的 Web 标准推广者还是有错的，包括我在内，我 2004 年撰写的《重构之美》代码示例部分带有更大的误导性（好在当初我一再强调代码毫无借鉴的意义，也算在文字上有所弥补）。现在呢？我也不知道，在路上，在路上……

写很多了，SPAN 的合理运用留给 Update 吧。

走在 Web 标准化设计的路上深入结构：
DIV 再议以及对 SPAN 的迷惑

作者：爆牙齿 [http://www.cnblogs.com/yuntian]

上篇文章中主要否定了使用 DIV 进行布局这种说法，提出 DIV 应当用于组织代码结构，现在我们再深入一点，DIV 拥有语义吗？这个问题前段时间在研究群里曾激烈争论过，当时米随随发问："什么是语义化 Web，DIV 是什么？"小毅答曰："DIV 表示无意义容器。"我说："否定。"然后旁边有人嘀咕："...又要打起来了。"我大笑着进入战斗状态，结果迅速被围攻了。总是和主流格格不入的我又一次站在主流的对立面。我还是不赞成将 DIV 视为无意义容器。容器这个概念是模糊的，是与设计挂钩的，理解成容器以后又远离结构了。再说每一个不是自我关闭的标签都可以视为容器，有什么区别？难道 DIV 可以包含一切，于是就可以随意使用了吗？那又如何固定 xhtml？所以还是要回到 DIV 的语义上来，DIV 是有语义的，只不过它的语义是面向代码结构的，是面向程序的。

division（分割），对了。前段时间浏览 w3schools 时，看到它是这样定义 DIV 的："The DIV tag defines a DIVision/section in a document."我想我对 DIV 的理解是没错的。在文档中定义一个分割或者节点。我说 DIV 用于模块化页面内容，实际上从代码结构角度是展现 xml 化的节点结构。除了定义一个节点以外，DIV 目前还用于定义一个分割，产生具有结构的行。还是以登录为例：

```
<div>
    <h3>用户登录</h3>
    <div>
      <label for="name">用户名</label>
      <input id="name" />
    </div>
    <div>
       <label for="pw">密码</label>
       <input id="pw" />
    </div>
    <p><button /></p>
</div>
```

最外层的 DIV 是作为产生节点使用，而用户名和密码部分实际上是为了产生具有结构的行，这里若使用 br 同样能够产生行，但是缺乏结构，所以 DIV 代替了 br。猜到我要说什么了吗？呵呵，又是 xhtml 2.0，2.0 中的 section 和 line 标签，是的。在 1.x 中，DIV 同时扮演了 section 和 line 的角色，因为分割产生节点，因为分割产生行。但是很明显 section 和 line 具有比 DIV 更为明确的语义，那么我们可不可以认为 DIV 的语义和 br 一样是模糊的。既然是模糊的，br 已经被毙了，我们现在大量使用的 DIV 会不会落到同样的下场呢？不知道，至少目前的 xhtml 2.0 中，DIV 仍然存在。看看上面的结构代码在 xhtml 2.0 中应该如何展示（没考虑 XForm）：

```
<section>
   <h>用户登录</h>
   <line>
      <label for="name">用户名</label>
      <input id="name" />
   </line>
   <line>
      <label for="pw">密码</label>
      <input id="pw" />
   </line>
   <div><button /></div>
</section>
```

有些人单纯地认为好像是 DIV 在不断嵌套，其实不是的，是没有办法而产生出来的假象。这里再请大家注意一个情况，需要和 CSS 结合起来看待按钮那个部分，在 xhtml 1.x 中我使用了 p，严格说从结构上是错误的，很明显按钮不是一个段落，仅仅是希望它换行呈现。但是如果使用 DIV，那么就必须给予这个 DIV 一个 class="button"以区分，并且在设定 CSS 的时候必须先清除公有的样式属性，这样会带来不少麻烦。另外作为节点的 DIV 和作为行的 DIV 同样会出现这种问题。示例：如果定义节点 div{width: 300px; padding: 10px;}，那么就必须在定义行 DIV 时要么覆盖要么清除以避免冲突，div div{width: 200px /*覆盖*/; margin: 10px; padding: 0 /*清除*/; color: #333;}，然后在定义 div div.button{margin: 0 /*清除*/; color: #F60 /*覆盖*/; background: #999;}的时候再做对行 DIV 的样式冲突避免，为了避免这种情况，采用对节点 DIV 增加 class="loginarea"和 p，这样就可以避开两次样式清除和覆盖操作。这样的情况在结构复杂的页面中更为明显，不是加 class 就行了，class 越多，文档通用性越差，xhtml 越难固定。这就是在 xhtml 1.x 中因为 DIV 的语义模糊带来的麻烦，在 xhtml 2.0 的结构中就很好办了，section{}、section line{}、section div{}，无需 class 也互不干扰，这里的 DIV 貌似很适合它分隔的语义，

不是行也不是节点，仅仅就是一个分隔。

我认为标签中最难理解的两个之一的 DIV 现在应该算是很清楚了。剩下的一个就是 SPAN，至今我仍未能理解 SPAN 如何产生结构，只好说说自己的迷惑了。

先说说 DIV 和 SPAN 的区别，从大的方面来说，DIV 被归类到 Structural Module（结构模块），而 SPAN 被归类到 Text Module（文本模块）。小的方面，DIV 是 block-elements（块级元素），SPAN 是 inline-elements（行内元素）。在所有 Structural Module 中，DIV 是唯一一个语义模糊的，在所有 Text Module 中，SPAN 也是唯一一个语义模糊的，两个 Tag 唯一的共性是语义模糊。

回到 SPAN 的语义：跨度和范围。比 DIVision（分割）更为抽象，难以理解。在一阵疯狂 google 后还是没找到我想要的那种解释，接近的都没有，也许根本就没有，所有的结果都指向表现，无论中英文都是指为字体添加样式，可是 W3 中明文写着“The SPAN element, in conjunction with the id, class and role attributes, offers a generic mechanism for adding structure to documents.”，这里的 for adding structure to documents 做何解释？百思不得其解，后来打开 W3 的源码查看他是如何使用 SPAN 的，虽说获得了一些提示，但依旧不足以领悟到 structure 的真谛，我想应该是我的 XML 功力还不够。既然语义上、结构上行不通，那么只好换个角度，从实际应用中试着去理解。SPAN 是行内元素，主要应用于文本，这点没什么异议，关键在于如何运用？我始终不认为 SPAN 是个样式容器，google 的时候发现清一色的容器解释，DIV 是大容器，SPAN 是小容器，我郁闷。如果 SPAN 因为文本的样式而存在，它凭什么存在？一段文本为什么要添加样式？如果你想强调应该使用 em，如果想特别强调应该使用 strong，Text Module 里还有很多语义明确的标签可以使用。所以 SPAN 应该不是作为样式容器而存在的，就像 DIV 不是作为布局容器而存在一样。但是我领悟不到 SPAN 的真谛，不过我可以抛砖引玉，在某一个地方我一定会使用 SPAN 的，那就是表单中。还是以登录为例，如果登录的数据需要展现出来，比如很多 edit 页面和 view 页面，结构应该完全相同，不同的是在 edit 页面中是输入框，而 view 页面中则用 SPAN 展现数据。类似如下代码：

```
<div>
    <h3>用户登录</h3>
    <div>
      <label for="name">用户名</label>
      <span>MyName</span>
    </div>
    <div>
      <label for="pw">密码</label>
      <span>MyPassword</span>
```

```
    </div>
    <p><button /></p>
</div>
```

这样的好处有两点：1. 和 label 区分开来，便于应用样式，如 div div SPAN{}；2. 可以通过节点提取所有录入的数据。这是我目前唯一非常明确的使用 SPAN 的地方，这里除了 SPAN 好像没有更合适的了，也符合它的语义（范围和结构化）。其他 SPAN 的运用仍在摸索中，包括从 W3 源代码中获得的提示。

差不多要说完了，这时我对关于容器的说法又耿耿于怀了，于是再次以容器为关键词疯狂 google，凭什么上上下下都说是容器，我要找出根源来，终于在最后，皇天不负有心人，在我执迷不悟地，怀着容器是错误理解的信念下，挖出了根源。W3 在这里对 DIV 和 SPAN 进行了这样的解释：generic language/style container。两者都一样。怪不得说所有的中文翻译都是容器，我想很少有人去看英文追根到底吧。确实 style container 应当翻译为样式容器。这一点都没错，错的是，这是 html 中的 DIV 和 SPAN，而不是 xhtml 中的 DIV 和 SPAN。随即我再查 W3 对 xhtml 中的 DIV 和 SPAN 的解释，已经不一样了。对于 DIV 是 Define the characteristics of a block，而对于 SPAN 是 Define characteristics of text。这才是我的理解，也是我想要的正确解释!! 因为这个是 xhtml 2.0 中的解释，由于 2.0 中 section 的存在，所以在对 DIV 的解释中，节点的含义被取消了。现在看刚才试着写下的 xhtml 2.0 登录结构中的 DIV 和最后一句话。DIV 既不做节点也不做行，可能还是有用的。

说到这里问一句，html 和 xhtml 最大的不同在哪里？是语法吗？是名称吗？是严格 xml 化了吗？不，两者本质区别是：**html** 是面向表现的语言，而 **xhtml** 是面向结构的语言。以我们应当从结构的角度去审视、理解与运用 xhtml 中的每一个 Tag。比如容器的理解，在面向表现的 html 中是正确的，但是在面向结构的 xhtml 中则错了，应该理解为节点。理解直接影响运用，以表现的理解显然无法写出结构化的代码。

好了，SPAN 现在总结不出来，只好先对 DIV 做个总结收尾：在当前 xhtml1.x 环境下，需要产生节点（section）和行（line）的时候选用 DIV。

阿弥陀佛，最烦人的两个东西总算告一段落，虽然未完，但是遗憾也是美嘛，以后再说了。结构也算告一段落，下篇可以换个口味了，正式进入语义。

——2006-4-27~29

走在 Web 标准化设计的路上
——复杂表单

作者：爆牙齿 [http://www.cnblogs.com/yuntian]

一直有种说法：Table 用于数据表，对于复杂表单，Table 也是最好的选择。由于一直没有遇见过，也就没有认真去研究到底复杂表单是否应该使用 Table。

现在，机会来了。我拿着复杂表单的图样，看来看去都觉得不应该用 Table，除非是有行标和列标的数据表表单。反而类似于登录这种简单表单，我倒是一直使用 Table，理由是能够在纯文档的时候对齐文本与输入框，对于复杂表单就不一样了，复杂表单涉及到页面布局了。

为什么要研究？因为我希望程序员不要涉及到界面的任何部分，对于页面，他只需要关注结构，而复杂表单如果采用 Table，很容易就将程序员带进对布局的操作中去。

xhtml 部分自己觉得还算摸清了一些规律，页面的分析信手捻来，DIV 结构下的复杂表单真是漂亮，但是 CSS 部分真是难搞，干了几个复杂表单的 CSS 都没摸清规律，尤其是文本长度不一致，表单控件又各种各样交错，还有错误提示隐藏的文本，时不时中间又加个按钮……迷茫了……算了，继续搞 CSS，希望最后能得出结论，对于复杂表单到底用 Table 还是 DIV?

先给一个对于登录界面这样简单的表单（我最常用的 xhtml 代码）使用 Table，理由见上：

```
<div>
    <h3><span>用户登录</span></h3>
    <table>
        <tr>
            <td><label for="name">用户名</label></td>
            <td><input id="name" />
        </tr>
        <tr>
            <td><label for="pw">密码</label></td>
            <td><input id="pw" /></td>
        </tr>
    </table>
```

```
    <p><button /></p>
</div>
```

另外不使用 Table 的如下：

```
<div>
    <h3><span>用户登录</span></h3>
    <div>
      <label for="name">用户名</label>
      <input id="name" />
    </div>
    <div>
       <label for="pw">密码</label>
       <input id="pw" />
    </div>
    <p><button /></p>
</div>
```

怎么说呢？第一种这样的简单表单为两行两列数据，用了 Table；第二种则是 DIV 模块化操作。一般我都使用第一种，除非文本长度一样（比如姓名，密码）才用第二种。当然我觉得第二种是正确的，所以我会优先在文案上先做文章使之长度一致。因为只有模块化才能固定 xhtml 而通过 CSS 随意布局，比如形式上为 1 行 4 列之时，第一种就做不到（其实 FF 可以正确解释对 tr 的浮动操作，例如两列 tr，但是 IE 不支持，一个 tr 怎么都得占 Table 的一行。）

——2006-2-23

此篇精彩评论：

#5 楼　　2006-02-23 23:12　avill

从模块化方面上来说，当然应该用 Web 标准，就像楼上说的，目前很多小的公司程序员跟界面设计师是同一个人，对于程序员来说，做界面的设计方面是很吃力的，如果再按 Web 标准化来作难度就更大了。如果我选择还是用 DIV，虽然复杂的表单用 Table 实现起来比较容易一些，但是考虑标准化可以给团队带来好处，给后期的维护工作带来方便，所以宁愿在前期的时候多花一些时间，多写一些慢慢也会上手的。

#10 楼　　2006-02-27 10:35　畅想自由

我个人觉得现在确实是个尴尬的时期，但不知道会持续多久。

现在很多美工还都使用全表格布局，当然也包括表单；当页面到我这里后，我又将它全改为了 布局，一是便于程序的编写，二是考虑到以后的重写。以前为了不要如此的麻烦，我曾经狠学了 photoshop，但最后因为精力有限，效果并不大，费了半天劲页面还是很丑；为此我决定告诉美工，要最大可能地多使用 布局，当然我没提什么“标准”，因为只要客户满意谁也

不会去关心那个狗屎标准！说实在的，用 CSS 来布局，还要考虑多种浏览器对 Web 标准的支持度，那么你肯定要发疯！表格就不存在这种问题，即使再复杂的表单，它的表现力也是很强的。

对于程序员来说，当然更喜欢 式的表单，因为如果有人让你将一个以前用<table>实现的表单修改或改写，你肯定会起火！但如果是 DIV 实现的表单你就会更轻松。

最终造成的局面就是：前端表现对标准的支持几乎为零，后端程序员又迫切要支持标准。

幸好没打起来!!

走在 Web 标准化设计的路上

[复杂表单：Reload]

作者：爆牙齿[http://www.cnblogs.com/yuntian]

在复杂表单上，标准的优势吸引着我，标准的劣势折磨着我。坚持住，几个月后，劣势开始变小，优势开始展现出来了。

还是先回到《一个简单又不简单的标准化表单设计实例》[①]这篇文章来。有个朋友（署名是 onestab）评论：你的 CSS 写得干净简洁，但是仍不够漂亮，最重要的，不管你是否真的来自曾经给予过我帮助的 OneStap，您还是没有给出正确办法，因为我的“大便蛔虫的表单主标题”部分到哪里去了？你怎能无视它的存在？没有它就简单了，难就难在它存在，又不能定义在 body 中也不能为它而改变结构增加 Tag。当然我相信多想一下你肯定能想出来。现在，我来说说看。

首先我很遗憾，竟然没有一个人指出：您的结构不完整。我前面呕心沥血写的文章都白写了，一到实例上大家眼中仍然没有关注结构。我记得以前在标准群中提这个问题的时候，小毅第一句话就是你的结构不完整。任我怎么解释他都咬死这句话，拒绝继续，气死我的同时还是感到欣慰。我们抛开设计稿，结构中在 body 和 DIV 之间最少都应该有个标题 h1 来指明这是“大便蛔虫的表单主标题”。“新增联系人”只是模块标题，还需要一个归属，这个模块到底是用在北京 WC 研究院还是上海 WC 研究院。这是一个成熟开发平台中发布了的通用组件，在我到来之前就已经存在很久了、作为通用组件，这个主标题应该具备并且动态生成，但是由于在我，把 Web 标准带去之前，这个公司 30 多个程序员都在用传统方式研发，没人关心过结构，所以这个功能一直没有，而要添加的话，牵一发动全身。最初我选择的是直接改动 ascx，增加 h1 写死在页面，但是这样的话就降低了组件的重用性和灵活性，尤其是有很多这样的表单，所以我最后选择放弃 h1。结构的不完整带来的就是 CSS 设计的困难，不能按正常思维去完成，我曾有那么几分钟很想把 h1 添加上去，但是最终还是从 CSS 上找到了办法。CSS 设计唯一的难点是结合结构分析页

---

① http://www.cnblogs.com/yuntian/archive/2006/05/11/397207.html

面，而 Table 不一样，按照设计稿切分表格就是了。当然 Table 也有页面分析，分析得好嵌套少，想省事，那么就使劲嵌套吧，所以传统的 Table 制作手段只有还原没有设计。而我在做 Css 设计的时候，10 分钟会有 8 分钟进行对着电脑发呆：单个页面的分析、整个站点的分析和整体设计的分析……打住，跑题了，说远了，回来回来……

没有 h1 的难点在于主标题图片放在哪里？边框怎么解决？

- 主标题如果放在 body 中，那么为了避免冲突，必须给每个这样的页面的 body 增加 class。
- 如果主标题放在 DIV 中，左右边框怎么办？如果把边框做成图片固定，那么怎么适应分辨率？如果用内部的 DIV 来拼凑边框，问题更大。

怎么办，走投无路的时候，反向思维！我们还有一个 Tag 可用，那就是 h3。

- 把“大便蛔虫的表单主标题”这张图片定义到 h3 中去，看似违背常规设计思路，却解决了问题。
- 给 DIV 样式 margin-top 和 body 拉开距离，然后用相对于 DIV 的绝对定位使 h3 飘起来去填充拉开的距离以定位图片标题。
- 再给 h3 样式 padding-top 定位“新增联系人”字体的位置，而“新增联系人”下面的渐变背景则定义在 DIV 中 repeat-x。
- 最后给 DIV 样式 padding-top 使表单内容在渐变下方。搞定！^_^

其实如果静心分析，可以一下就出来了。要实现自适应边框一定不能用图片实现，只能用 DIV 定义边框，因为表单内容会变，所以不能使用第三层的 DIV 拼凑边框。主标题在边框外，正常设计下，这就意味着 DIV 外需要 tag，而目前 DIV 外只有不能使用的 body，那么要解决这个问题唯一的办法就是从 h3 身上想，h3 能否跑到 DIV 外面去？可以！绝对定位。看，分析完了也就做完了。这种手法有一点类似我在《Web 标准下，OA 系统常见左栏定宽，右栏自适应宽度的“爆（牙齿）力解决法”》[①]一文中的手法。活用绝对定位，跳出常规设计思维。要解决这个问题不仅必须对 CSS 很熟悉，还要思维灵活，关键是要能从上到下、各角度去分析页面、站点和整体设计，而不是急急忙忙地动手。思想的高度决定手中活的容度。关键样式如下：

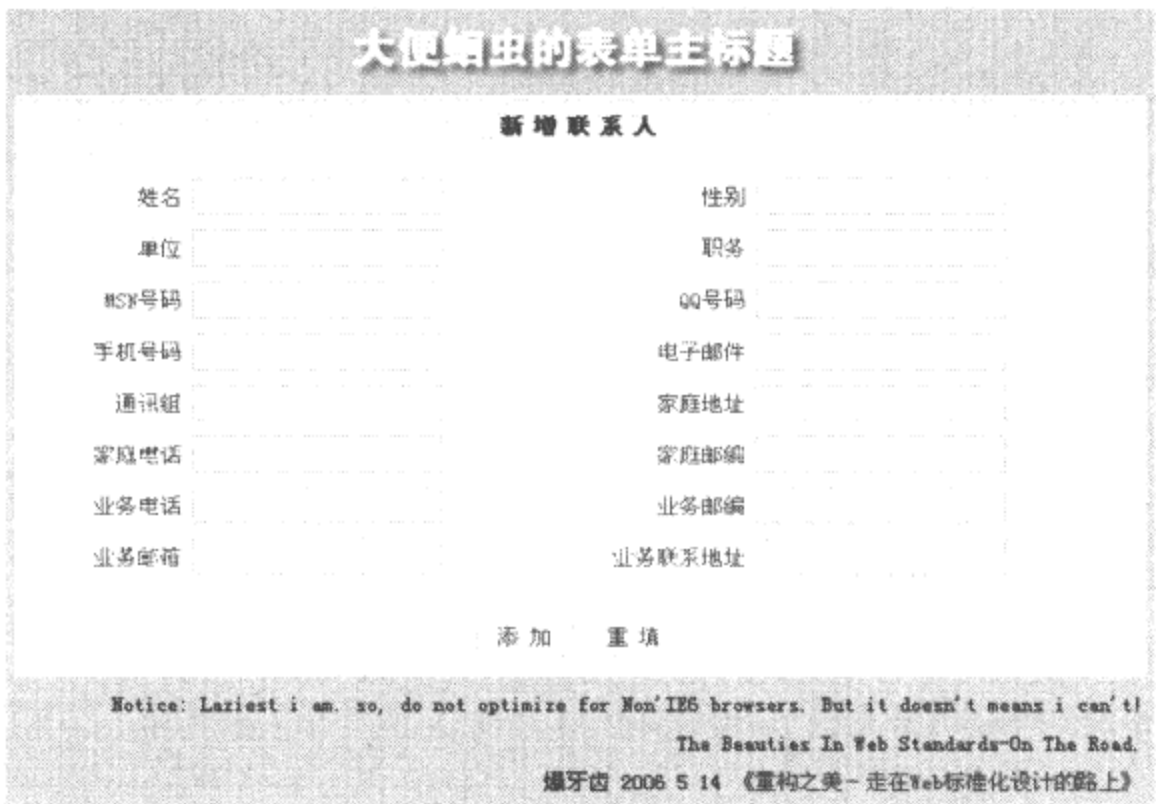
```
*
{
    top: 0;
    left: 0;
    margin: 0;
    padding: 0;
    font: normal 12px "宋体";
```

[①] http://www.cnblogs.com/yuntian/archive/2006/03/28/361281.html

```
}
body{background: #AEB4C5;}
input{} /*全局输入框*/
.buttonarea input{} /*全局按钮*/


#PopPage /*相对定位*/
{
    position: relative;
    width: 98%;
    margin: auto;
    margin-top: 50px;
    padding-top: 45px;
    text-align: center;
    border: 1px solid #FFF;
    background: url(bg_02.png) repeat-x #F2F3F8;
}
#PopPage h3 /*绝对定位*/
{
    position: absolute;
    top: -50px;
    width: 100%;
    padding-top: 60px;
    background: url(bg_01.png) center 10px no-repeat;
}
#PopPage div /*定宽浮动*/
{
    float: left;
    width: 295px;
}
#PopPage div label,
#PopPage div input{float: left;}
#PopPage div label
{
    width: 75px;
    text-align: right;
}
#PopPage div.buttonarea
{
    clear: left;
    width: 100%;
    background: url(bg_04.png) repeat-x;
}
```

```
#PopPage div.buttonarea input{float: none;} /*弹出表单按钮浮动清除*/
```

自适应自己下载去试试，1024、1280 都可以，我给出这个固定宽度的弹出窗口最终设计效果如下：



The End？结束了吗？没有，小小的例子可以引出很多东西来，慢慢来说。我想大家肯定是有很多疑问的，说不定看了设计效果会有不少人觉得明显和设计稿不一样嘛！文字中间的间距和输入框长度也不对……我笑笑，说：凭什么要和设计稿完全一样！

说了这句话，我也静了一下。程序和设计是两条截然不同的路，设计是感性思维，而程序是理性思维。两者全通的人很少，所以作为程序员一般而言是没有资格去否定设计的，作为设计师也没有资格去否定程序。问题就出来了：那么制作谁来做？我从一开始进入互联网就不认为设计和制作是可以分开的，就算在 Table 布局时代，一个优秀的设计师完成设计稿后其实制作已经完成，因为设计时就已经充分考虑到制作（如何切图），精确到像素，脱离设计稿靠记忆都能迅速把表格准确切分出来（当然很多的所谓设计师只是随心所欲地在画画根本不考虑制作，所以被称为美工）。一个优秀设计师的设计稿绝对不仅仅就是一张图，它还涵盖了很多理念在里面（UI、UE、交互等），到底有多少在于设计师对网页这个词的理解深度。你说平面设计师能不去考虑印刷吗？他在做设计的时候虽然不用考虑制作但必须考虑印刷的限制。对于平面设计师，他的设计水平不是设计稿，是印刷后的效果，同样对于网页设计师，设计水平体现在已经成形的网页上，网站设计师范围更大——网站的整体。一个室内设计师如果不知道材料和施工，只会玩 3DS MARKS 把效果图做得天花乱坠而又实现不了，是没有价值的。上面的例子，如果设计稿

是我做的，我压根就不会把标题放在外面，或者不会选择这种表现形式来进行设计，我会充分考虑如何通过设计来降低制作的难度从而优化制作。还有，文字的间距，此类表单如果文字无规律可寻，我也不会这么设计。间距是表现，是排版的一种手法（说到这里，既然间距是表现也应当写进 CSS，所以提醒大家，Web 标准下不要随便使用空格。空格不是办法，有本事用空格把 4 个字和 5 个字对齐，^O^！只有 CSS 才能做到）。文案如果不能做到有规律可寻，那么就不该设计间距，否则给予 CSS 设计很大难度，要在 xhtml 中增加很多无谓的 class，还要一个一个定义，不是做不到而是很不值得。请搞清楚一件事，要做的不是 CSS 还原，而是 CSS 设计！哎呀，不说了，严重跑题。大家别看了这段话就去和自己的设计师们叫板哈，那我就成罪人了，除非你自信自己对设计的理解比他们深，否则还是请尊重设计稿。

还是回到这篇文章的目的：复杂表单可以标准化吗？应该标准化吗？第一个问题答案是可以。第二个问题分两方面来论证一下。

求证：理论上来说，复杂表单应该使用 Table 进行设计吗？

如果用 Table 来处理上面的例子，表单部分很明显是 4 列多行数据。但是它是吗？Table 是表格对吧，什么是表格？数据库应该是最好的表格了吧，像上面的例子，怎么在数据库中定义字段？当只有一条记录的时候，我们把数据库的行当列，列当行，字段作为值，它是一个两列多行的表，而不是 4 列。为什么会产生 4 列？那是因为表现的需要，或许我还会设计成 3 列显示，再或许我希望它随分辨率的变动自适应。你的数据库中会设计多个重复的字段吗？所以，复杂表单虽然带个“表”字，但是大部分都是伪表格数据，只有部分表单是真表格表单，类似 DataGrid 在线编辑模式下的那种。好了，论证完毕。

求证：实践中，从效率上讲，复杂表单应该使用 Table 进行设计吗？

这个是大家最关心的事情。这个是 Web 标准的特性：对于个人，尤其对于单个页面，用标准设计的速度永远比不上 Table 的可视化设计速度。页面设计如此，表单设计同样如此。两年前，我在《重构之美 – 迎接 Web 标准化设计的来临[总结一：网页设计回归？]》[①]中对标准的速度做过如下总结：

> 关于速度和效率的问题。
>
> 很多人都在想这个问题，被这个问题所困扰。cloudchen 曾经回复过我这么一句话：网页就象快餐，在这种东西上面浪费时间不值。我深以为是，速度是非常重要的。但是我要说的是，如果使用标准，对于网站来说，开发速度会比使用表格快 *N* 倍。从单个页面来开，使用标准再熟练再快都快不过表格，说不定差距还会很大。但是对于网站来说就不一样了，随着网站规模的

[①] http://www.cnblogs.com/yuntian/articles/316550.html

扩大，差距会缩小，到某个临界点时，两者会持平，而后使用 Web 标准的网站在开发速度和效率上会将表格远远抛在后面。所以对于小网站、小应用来说，使用标准在速度上是完全没有优势甚至会有明显劣势，但是对于一个大网站，尤其是非一次性开发而是持续性不断开发的网站来说，呵呵，不用我说了吧。我使用表格 5 年，我的表格设计速度可以说已经练到极为准确和快速了，几乎无法再大幅度提升了，而我使用标准才 2、3 个月，我有这样的体会，你做何选择？

今天同样说到表单设计上，对于团队、对于大量的表单，标准化后的优势会淋漓尽致地凸现。我不止一次说过：Web 标准从一个方面上讲是一项为团队而生的技术和标准！回到上例吧，先给一张图片，这个是我让程序员整理的该项目需要设计的弹出表单列表，先看看、数数，再看看旁边的下拉条。

感受如何？我有两个感受：第一站在传统的角度，我也是从传统过来的，我感到可怕。第二很庆幸标准的出现和我两年来的应用积累成精了，100 个表单页面，就算斩一半也有近 50，我们所有的程序员都解脱了，所有表现全部压在我头上，我一点都不怕，虽然依旧麻烦，但是相比之下，对于团队已是数量级的效率提升了。而且 Web 标准的高灵活性、高重用性和对未来的高度适用性等所有优势将驻留在标准化下的表单中，为未来带来极大的便利。

最后，当你的系统和网站页面全部标准化了，但是每个表单仍然 Table 化，那么它将成为水桶的最短的那块板。还有什么疑问吗？那么我下结论了。

结论一：理论上来说，复杂表单应该使用 Table 进行设计吗？回答是 NO!

结论二：实践中，从效率上讲，复杂表单应该使用 Table 进行设计吗？回答仍是 NO!

不要说这个表单简单，复杂也一样的，想想最初的标准只是在很简单的页面上试验，最简单的页面→博客→个人网站→门户网站，一路走来，发现其实复杂页面也没什么大不了（当然现在很多标准化过的网站实际上很差），我的意思是表单同样如此，不在于它简单还是复杂，关键还是在于结构化和语义化。

好了，两次的铺垫都为了下一篇，究竟什么是表格？什么是表状数据？

——2006-5-15

走在 Web 标准化设计的路上

[深入语义：列表和表格的抉择]

作者：爆牙齿 [http://www.cnblogs.com/yuntian]

问题：XHTML 中的列表 Tag（ul/ol）和表格 Tag（Table）区别何在？

对于单列多行下的数据表，如何判断和选择？

这是我在培训中提的第二个问题。如果说上一个问题“理解 h 系列的不合理”能够把人带入对结构（代码结构）的思考中，那么这第二个问题我认为则是把人带进对语义（语义结构）的思考中。

说到标题“深入语义”，其实并不是从现在才开始的，大家应该发现在之前的文章中，我总是会回到语义这个点上去展开思路，论述结构。关于语义这个概念非常重要，什么是语义？一展开就又是长篇大论，我现在也很难说得清，所以还是回到这篇文章的主题吧。

我想大家肯定变狡猾了，先去 W3 查过对 ul/ol 和 Table 的定义，我也是。但是遗憾的是 W3 给的答案是 Define a List, Define a Table. 这不是废话嘛？说了等于没说。什么是 List？什么是 Table？还有办法，金山词霸！

List：英文【A series of names, words, or other items written, printed, or imagined one after the other】

中文【目录，名单，列表，序列，数据清单，明细表，条纹，[总称]各种上市证券】

Table：英文【An orderly arrangement of data, especially one in which the data are arranged in columns and rows in an essentially rectangular form.】

中文【表：数据中的一种有序排列，尤指其中的数据按基本上构成一个矩形的竖行和横行进行排列的一张表格】

好像也得不出什么结果，虽然好像已经有所提示了，问题的最关键在于单列多行的时候如何抉择？

首先严正申明：由于没有准确的 List 和 Table 定义，所以和之前有法可依，有矩可循不一样，这次纯粹是个人的理解了。我想还是和上次一样分两方面

来分析—理论和实践。

理论先行，分别用 List 和 Table 列一段单行数据如下：

List

```
<ol>
 <li>数据一</li>
 <li>数据二</li>
 <li>数据三</li>
 <li>数据四</li>
 <li>数据五</li>
</ol>
```

1.数据一
2.数据二
3.数据三
4.数据四
5.数据五

table

```
<Table>
<tr>
    <td>数据一</td>
</tr>
<tr>
    <td>数据二</td>
</tr>
<tr>
    <td>数据三</td>
</tr>
<tr>
    <td>数据四</td>
</tr>
<tr>
    <td>数据五</td>
</tr>
</table>
```

数据一
数据二
数据三
数据四
数据五

看出来什么没有？好像什么也看不出来，ul 和 ol 的圆点数字说明不了什么问题，ul/ol 的代码量比 Table 少很多，也不能说明什么问题，Table 的边框是我故意加的。好，我们接着继续看，让我们把数据扩展一次再看。

List Expand

```
<ol>
 <li>数据一
 & nbsp;   <ol><li>AAA</li></ol>
 </li>
 <li>数据二
 & nbsp;   <ol><li>BBB</li></ol>
 </li>
 <li>数据三
 & nbsp;   <ol><li>CCC</li></ol>
 </li>
 <li>数据四
 & nbsp;   <ol><li>DDD</li></ol>
 </li>
 <li>数据五
 & nbsp;   <ol><li>EEE</li></ol>
```

table Expand

```
<table class="Table">
<tr>
    <td>数据一</td>
    <td>AAA</td>
</tr>
<tr>
    <td>数据二</td>
    <td>BBB</td>
</tr>
<tr>
    <td>数据三</td>
    <td>CCC</td>
</tr>
<tr>
    <td>数据四</td>
```

```
  </li>
</ol>

  1.数据一
  1.AAA
  2.数据二
  1.BBB

  3.数据三
  1.CCC
  4.数据四
  1.DDD
  5.数据五
  1.EEE
```

```
      <td>DDD</td>
  </tr>
  <tr>
      <td>数据五</td>
      <td>EEE</td>
  </tr>
  </table>
```

数据一	AAA
数据二	BBB
数据三	CCC
数据四	DDD
数据五	EEE

现在看出来了吗？仔细看看？还没有吗？还需要我继续扩展吗？

我直接说我的认知了：**Table 和 ul/ol 都能产生数据行，但是 Table 的重心应该是在产生数据列，而 ul/ol 的重心应该是在产生数据级。所以对于单列多行的数据，扩展的趋势是产生级的时候，使用 ul/ol；扩展的趋势是产生列的时候，使用 Table。**

是的，我是这么区分的，趋势、级、列。我认为是隐藏的语义。

现在我再回到实践中来分析我的观点。

最常见的是网站中的新闻列表，特别是首页上的各栏目新闻列表，绝大部分都在使用 ul/ol。我认为是种滥用，应该用 Table。这里要到后台程序开发中走走，新闻列表从数据库里产生出来，在数据库里，一条新闻由许多字段组成，首页上的简短新闻表和内页中的完整新闻表在一些情况下有可能是调用同一个存储过程或者 SQL 语句，不同的仅仅是数据的绑定。首页上的可能只是绑定标题和时间，内页中或许会更完整一点，比如加上点击数、作者之类。如果一个新闻表有 3 列以上，你肯定不会使用 ul/ol 了。那么两列的和 3 列的有本质区别吗？我知道两列可以很容易用 ul/ol 实现，增加 SPAN 即可。那么 3 列呢？你说可以，给 SPAN 加 class。好吧，4 列了，你还说可以吗？

本来一个样式就可以内外定义完同样的新闻表，分开后你又要定义 ul/ol，又要定义 Table，不是多此一举吗？最重要的还是在扩展的趋势上，如果数据一旦向列扩展，ul/ol 将非常困难。

ul/ol 不是用来取代或者模拟 Table 的，Table 用于表状数据，扩展趋势是列的数据，哪怕它只有一列也应该视为表状数据而使用 Table。

当然有些时候数据又有列又有级，这时就会出现混用的情况，相对复杂点了，但我觉得判断还是在级和列上，谁为重？

IE 7 标准之道

——1. 更丰富的 CSS 选择符

作者：阿一（杨正祎）[http://www.cnblogs.com/JustinYoung]

IE 历来被 Web 标准的拥护者所诟病，而当 FireFox 横空出世以后，更多的网页制作者开始关注 Web 标准设计。看着 FireFox 的市场占有率不停上升，微软终于推出了 IE 7。但 IE 7 是否真的能够力挽狂澜，是否真的能够得到用户的信任，是否真的能够得到网页设计者的认可呢？

且看《IE 7 的 Web 标准之道》系列文章，和你一起见证 IE 7 的改变！

CSS 选择符

IE 7 最令网页设计者兴奋的改进，便是支持更多、更丰富的 CSS 选择符（也有翻译为选择器的）了。这样通过 CSS 便能够更进一步方便地控制、定位前端结构元素，从而更丰富、多样地制定样式。

CSS 选择符其实大家都见过，甚至都用过。看看下面的例子，自然就会明白了。

```
<!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>IE 7 的 Web 标准之道示例页面</title>
<style type="text/CSS">
#DIV1{
color:red;
}
.DIV2{
color:blue;
}
```

```
</style>
</head>
<body>
<div id="div1">
id 选择符示例
</div>
<div class="DIV2">
类选择符示例
</div>
</body>
</html>
```

示例中的“#”和“.”就是 CSS 选择符，正是因为有选择符，红色字体和蓝色字体这两种样式才准确定位到了 id 为“DIV1”和 class 为“DIV2”的两个 DIV 标签上。**选择符其实就是 CSS 为样式找到前台结构元素目标的一种机制。**

选择符有很多种，上面的示例中利用“#”符号的称为“ID 选择符”，因为它是根据前台结构元素的 id 定位的。利用“.”符号的称为“类选择符”，因为它是根据前台结构元素的 class 名定位的。其他的还有“通配选择符”、“类型选择符”和“包含选择符”。

更多高级选择符

IE 7 与 IE 6 相比，支持了更多的选择符，正是因为支持了这些丰富的选择符，使得 IE 7 可以更方便地实现一些以前在 IE 6 中很难实现或者无法实现的效果。下面就让我们看看这些令人兴奋的、IE 7 新支持的选择符。

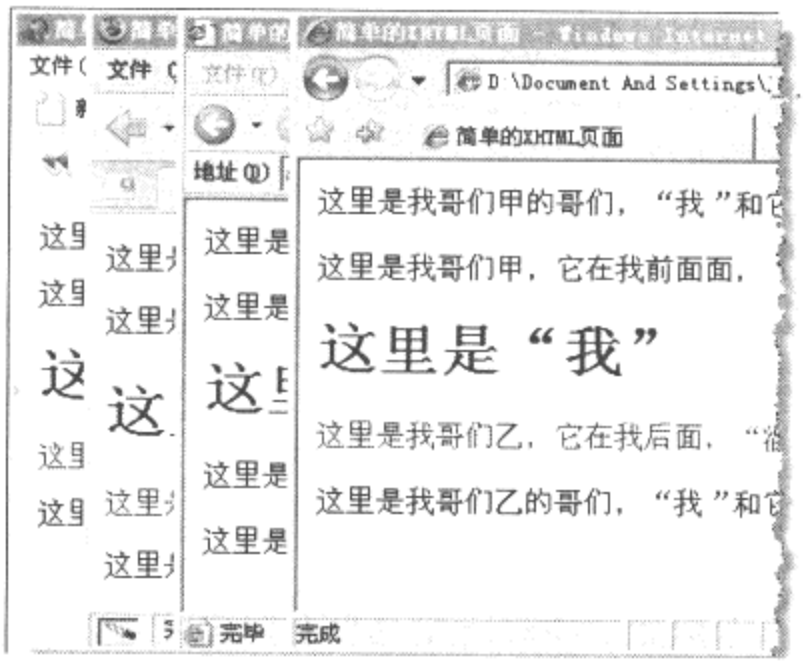
相邻同胞选择符

利用“相邻选择符”，可以根据一个元素定位到与之相邻的另一个元素，并应用样式。“相邻选择符”就是只对“与自己平行关系的”、“相邻的”、“在我下面的”“哥们元素”起作用。下面的例子更有助于理解（注意，只有处于 h1 后面的 p3 字体颜色发生了变化）。

```
    <!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=u
```

```
tf-8" />
        <meta name="Keywords" content="简单的 XHTML 页面" />
        <meta name="Description" content="这是一个简单的 XHTML 页面" />
        <title>简单的 XHTML 页面</title>
        <style type="text/CSS">
            h1 + p {color:blue}
        </style>
    </head>
    <body>
    <p id="p1">这里是我哥们甲的哥们，“我”和它不太熟。</p>
    <p id="p2">这里是我哥们甲，它在我前面面，“混”的比我好，所以我就不管它了。</p>
    <h1>这里是“我”</h1>
    <p id="p3">这里是我哥们乙，它在我后面，“混”的还不如我，所以我要照顾它一下。</p>
    <p id="p4">这里是我哥们乙的哥们，“我”和它不太熟。</p>
    </body>
    </html>
```

下面是分别在 IE 6、IE 7，FireFox（版本 2.0.0.12）和 Oepra（版本 9.25）的显示效果截图。



下面是关于“子选择符”和“后代选择符”的一些补充资料，对于初学者不建议阅读。

很多朋友分不清“子选择符”和“后代选择符”的区别，其实它们的差别还是蛮大的。“后代选择符”在 IE 6 甚至更低 IE 版本中就已经支持了。也许你已经被这些“拗口”、“深奥”的名词搞迷糊了。到底什么是“后代选择符”呢？其实大家都用过，看看下面的例子就明白了。现在无论是 FireFox 还是 IE 7，对于“子选择符”的支持还都存在一定的问题。

```
    <!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=u
tf-8" />
    <meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,实
例代码" />
    <meta name="Description" content="这是一个简单 YES!B/S!文章示例页
面,来自杨正祎的博客,
http://justinyoung.cnblogs.com/" />
    <title>YES!B/S!文章示例页面</title>
    <style type="text/CSS">
    /*后代选择符*/
    #DIV1 p{  color:red; }
    /*子选择符*/
    #DIV1>p{ font-size:150%; }
    </style>
</head>
<body>
<div id="DIV1">
 <p id="p1">我是 DIV1 的儿子 1
   <p id="p1_1">我是 DIV1 的孙子</p>
 </p>
 <p id="p2">我是 DIV1 的儿子 2</p>
</div>
</body>
</html>
```

示例中，"p1"和"p2"包含在“DIV1”内，那么"p1"和"p2"就是“DIV1”的儿子，是后代；“p1_1”包含在“p1”中，那么“p1_1”是“p1”的儿子，是后代；“p1_1”也包含在“DIV1”中，则“p1_1”是“DIV1”的孙子，也是后代。使用“#DIV1 p{color:red;}”（后代选择器）会将 DIV1 下面的所有段落的字体颜色都设置为红色。无论是儿子还是孙子，都要听话。“只要是我的后代，就得听我的话!”—这就是“后代选择符”。

而“子选择符”则不会那么霸道，它只管它的“儿子”，不会去管“孙子”、“重孙子”、“重重孙子”……

属性选择符

“属性选择符”可以根据某个属性是否存在或者属性的值来寻找元素。巧妙地利用“属性选择符”，将可以轻松地实现很多实用而且强大的效果。

很多“以用户为核心”的拥护者一直强烈要求去掉<a>标签的 target 属性，

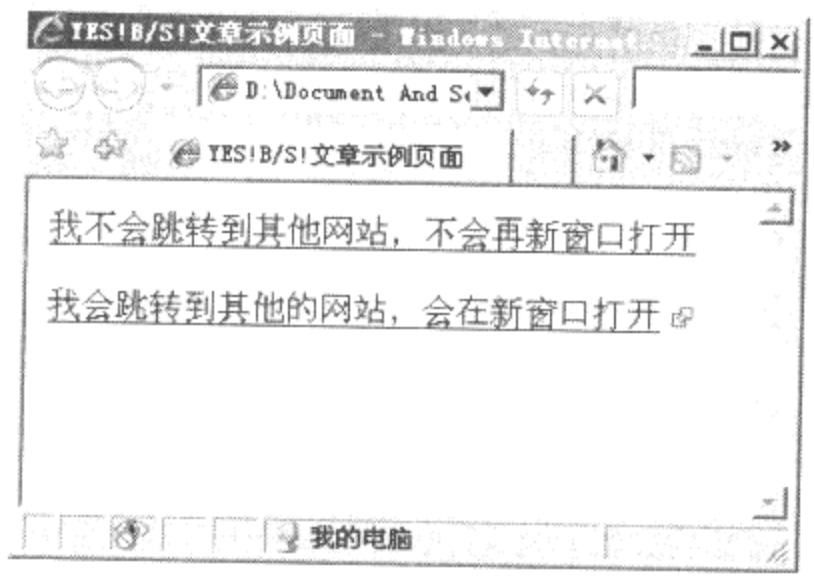
一个很重要的原因就是“没有经过用户的同意，就打开了新的浏览器页面，是一种不尊重用户的表现”。其实，我个人感觉，大可不必如此兴师动众。因为就算是去掉了这个属性，还是无法尊重用户，因为如果有的用户就是想在新的页面打开这个链接呢？（假使，他不知道使用 shift 单击链接可以达到目的，而且他也没有安装类似于“拖曳打开新页面”的浏览器或者插件）。

其实，使用“属性选择符”可以比较有效地解决上面的问题。在新窗口打开的超链接时明确标识出来，由用户决定是否去点击超链接（很多外国的网站已经在使用这种方式，而且已经形成了一种共识：在一个网站各页面之间的跳转链接，不在新窗口打开。而“会跳转到外部网站的链接”将在新的页面窗口打开，“会跳转到外部网站的链接”的标识图片也已经形成了共识，就是这个图标：“”）。下面的示例便是一种供参考的解决方案。

```
    <!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=u
tf-8" />
       <meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,实
例代码" />
       <meta name="Description" content="这是一个简单 YES!B/S!文章示例页
面,来自杨正祎的博客,
    http://justinyoung.cnblogs.com/" />
       <title>YES!B/S!文章示例页面</title>
    <style type="text/css">
       a[target="_blank"]{
          padding-right:16px;
          background:url('http://images.cnblogs.com/cnblogs_com/just
inyoung/common/
    outLink.gif') no-repeat right;
       }
    </style>
    </head>
    <body>
    <p>
       <a href="#" title="我不会跳转到其他网站，不会再新窗口打开">我不会跳转
到其他网站，不会再新窗
    口打开</a>
    </p>
    <p>
       <a href="http://www.163.com" title="我会跳转到其他的网站，会在新窗
```

```
口打开" target="
    _blank">我会跳转到其他的网站，会在新窗口打开</a>
    </p>
    </body>
    </html>
```

下面是 IE 7 的显示效果截图。注意第二个超链接后的图标。



更强大的是，“属性选择符”也可以判断一些自定义的属性，这对于一些开发第三方插件的程序员是一个极大的方便。例如，对于开发网页“网页翻译”的朋友，可以将需要翻译的关键字用包起来，然后对这个 SPAN 设置一个自己的属性。例如中文到英文的翻译，就加个"lang='c2e'"，如果是中文翻译到日文，就加个"lang='c2j'"。然后通过“属性选择符”为两种情况设置不同的样式，从而区分开来。

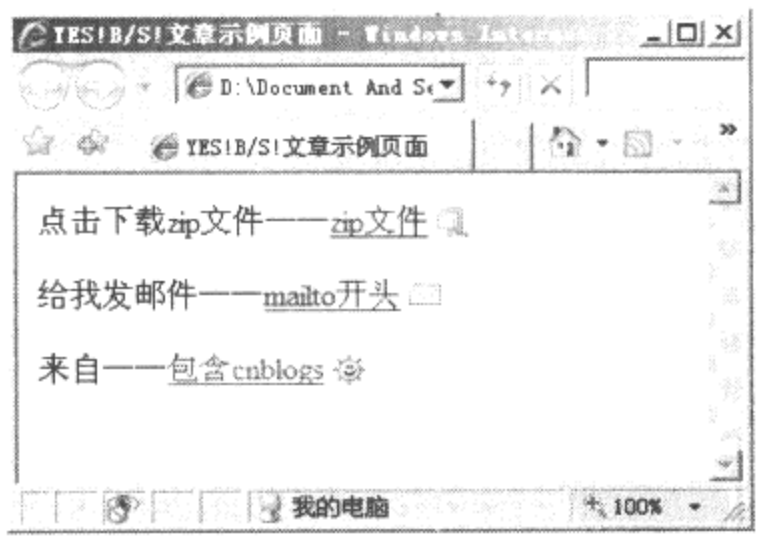
扩展资料：“属性选择符”高级使用技巧

“属性选择符”功能是令人惊讶的强大，它不仅能够识别简单的属性，判断属性的值，甚至可以根据简单的正则表达式来匹配属性的值。看下面的示例。

```
    <!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=u
tf-8" />
        <meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,实
例代码" />
        <meta name="Description" content="这是一个简单 YES!B/S!文章示例页
面,来自杨正祎的博客,
    http://justinyoung.cnblogs.com/" />
        <title>YES!B/S!文章示例页面</title>
```

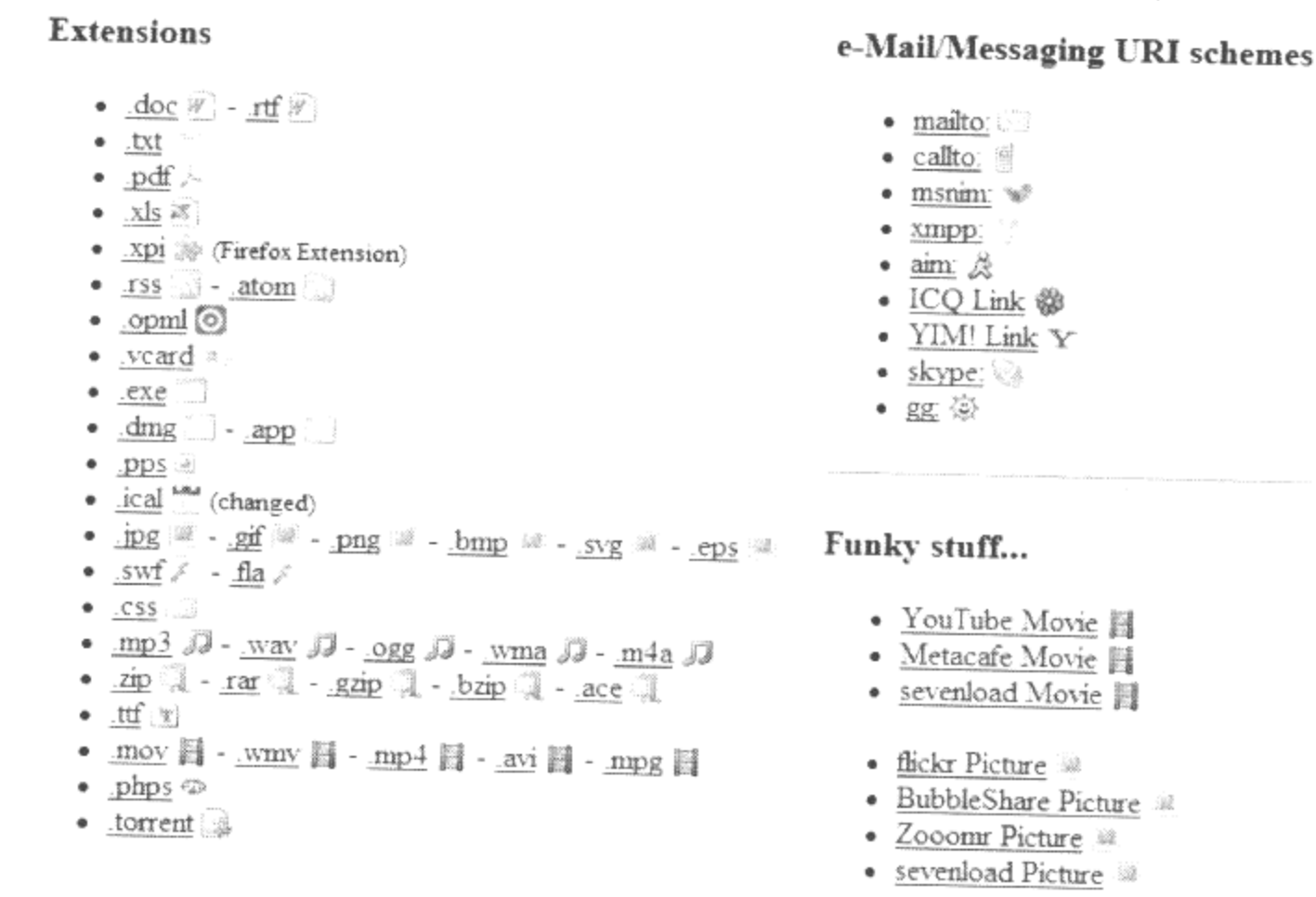
```
    <style type="text/css">
       /*以条件字符串结尾*/
       a[href$='.zip'] {
       padding: 5px 20px 5px 0;
       background: url(http://images.cnblogs.com/cnblogs_com/justiny
oung/common/
    icon_zip.gif) no-repeat center right;
       }    /*以条件字符串开头*/
       a[href^='mailto:'] {
       padding: 5px 20px 5px 0;
       background: url(http://images.cnblogs.com/cnblogs_com/justiny
oung/common/icon_
    mailto.gif) no-repeat center right;
       }    /*任意位置包含*/
       a[href *="cnblogs"]{
       padding: 5px 20px 5px 0;
       background: url(http://images.cnblogs.com/cnblogs_com/justiny
oung/common/icon_cnblog.
    gif) no-repeat center right;
       }
    </style>
    </head>
    <body>
    <p>点击下载 zip 文件——<a href="download.zip" title="zip 文件">zip 文件
</a></p>
    <p>给我发邮件——<a href="mailto:123456@163.com">mailto 开头</a></p>
    <p>来自——<a href="http://www.cnblogs.com" title="包含博客园网址">包
含 cnblogs</a></p>
    </body>
    </html>
```

下面是 IE 7 的显示效果截图。



图：“属性选择符”高级使用技巧示例效果图

如果开动脑筋和手指，你将得到更多令自己惊讶的效果。



Extensions

- .doc - .rtf
- .txt
- .pdf
- .xls
- .xpi (Firefox Extension)
- .rss - .atom
- .opml
- .vcard
- .exe
- .dmg - .app
- .pps
- .ical (changed)
- .jpg - .gif - .png - .bmp - .svg - .eps
- .swf - .fla
- .css
- .mp3 - .wav - .ogg - .wma - .m4a
- .zip - .rar - .gzip - .bzip - .ace
- .ttf
- .mov - .wmv - .mp4 - .avi - .mpg
- .phps
- .torrent

e-Mail/Messaging URI schemes

- mailto:
- callto:
- msnim:
- xmpp:
- aim:
- ICQ Link
- YIM! Link
- skype:
- gg:

Funky stuff...

- YouTube Movie
- Metacafe Movie
- sevenload Movie

- flickr Picture
- BubbleShare Picture
- Zooomr Picture
- sevenload Picture

伪类选择符和伪对象选择符

这又是令人头疼的“名词”，到底什么是“伪类”、“伪对象”呢？到底“伪”在哪里呢？我们依然从一个大家都用过，很熟悉的示例说起。

```
    <!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=u
tf-8" />
       <meta name="Keywords" content="简单的 XHTML 页面" />
       <meta name="Description" content="这是一个简单的 XHTML 页面" />
       <title>简单的 XHTML 页面</title>
    <style type="text/CSS">
       a:link,a:visited,a:active{
           color:red;
       }
       a:hover{
           color:blue;
       }
    </style>
    </head>
```

```
<body>
<a href="#" title="测试">鼠标附上去字会变成蓝色</a>
</body>
</html>
```

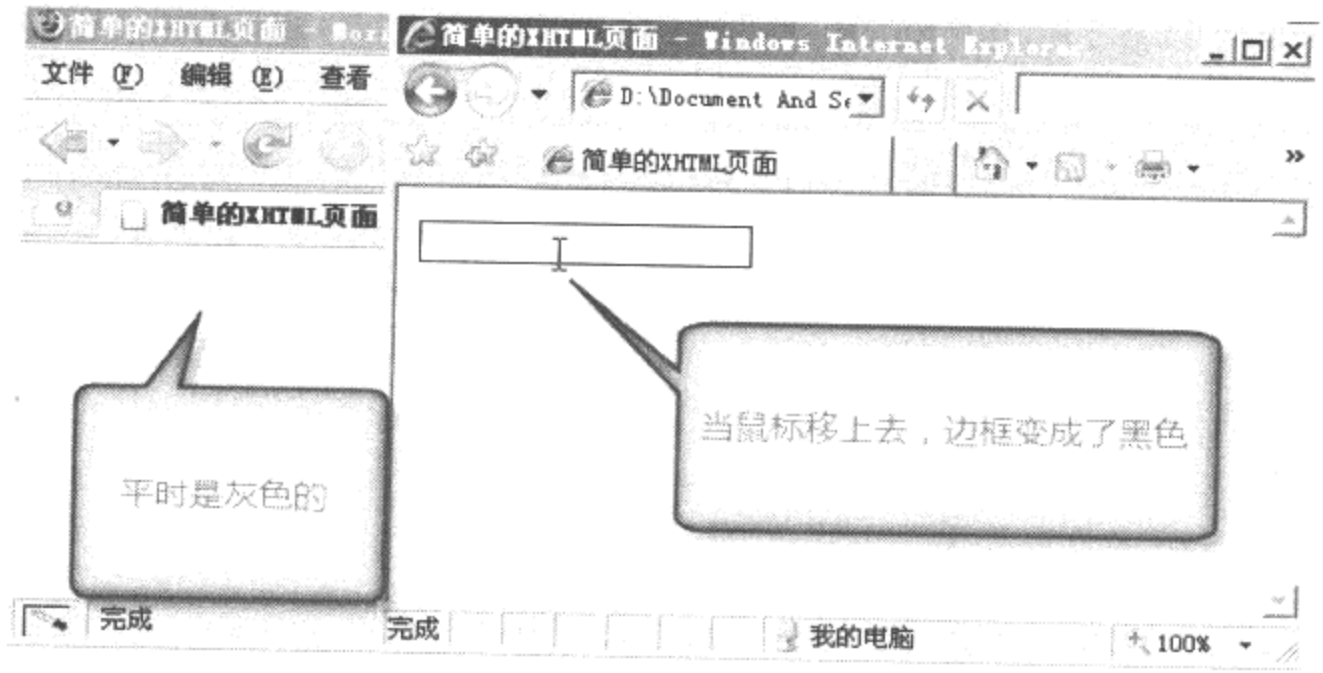
很简单且常见的情况，给超链接设置“鼠标移上不同字体颜色发生变化”的样式。这里的“:link”、“:visited”、“:active”和“:hover”就是“伪类”。之所以说是“伪”，是因为这些东西一定要依附在某种标签上（示例中是<a>标签），并不能单独存在，当它们单独存在的时候将没有任何意义。

在IE 6中只支持超链接<a>标签的伪类，而在IE 7中则支持几乎所有“可见标签元素”的伪类。也就是说，就算是一个DIV，你也可以设置div:hover的样式。可以预测的未来是：一些简单的样式方面的变化，将不再需要JS去控制，用CSS就可以实现简单的“动态”效果。看下面的例子，将更有助于理解。

```
<!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=u
tf-8" />
   <meta name="Keywords" content="简单的XHTML页面" />
   <meta name="Description" content="这是一个简单的XHTML页面" />
   <title>简单的XHTML页面</title>
<style type="text/CSS">
#txtName{
   border:1px solid #eee;
}
#txtName:hover{
   border:1px solid black;
}
</style>
</head>
<body>
<input type="text" id="txtName" />
</body>
</html>
```

上面的示例很简单，也很常用。一个输入名字的文本框，平时状态下是“灰色”边框，而当用户将光标移上去的时候，边框“变成”黑色，从而达到提醒的目的。

下面是 IE 7 和 FireFox 的显示效果截图。

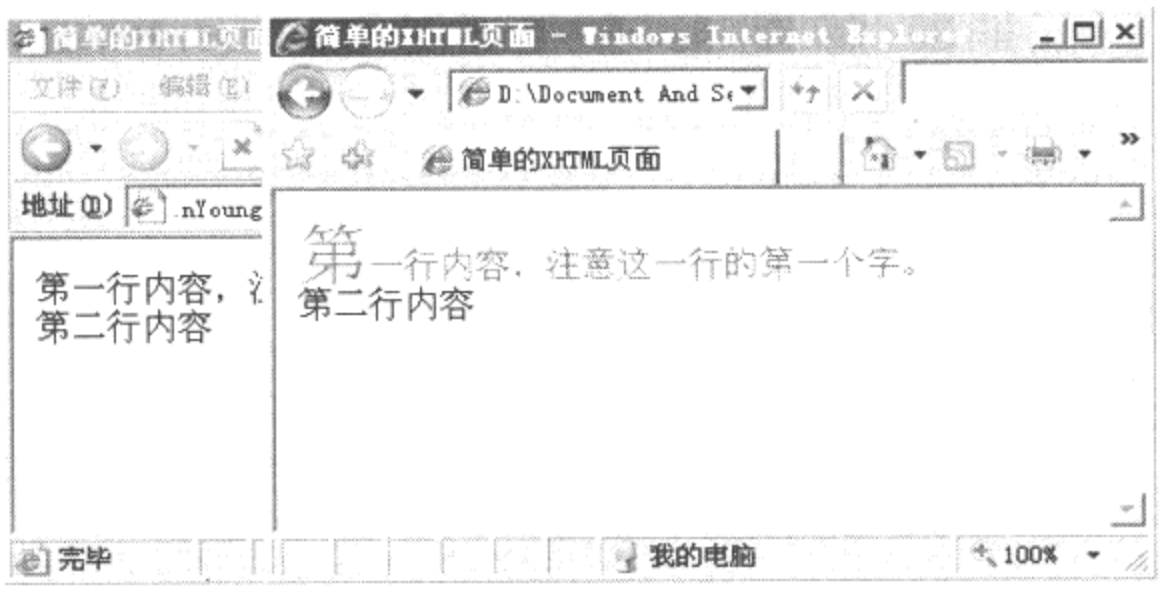


下面来讲讲“伪对象”。“伪对象”也是“伪”，自然也必须依附其他元素，而不能单独存在，“对象”意味着“有实体”的东西。最常用的“伪对象”就是“:first-letter”（子对象的第一个字）和“:first-line”（子对象的第一行）。下面的示例有助于理解“:first-letter”和“:first-line”伪对象。

```
    <!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=u
tf-8" />
        <meta name="Keywords" content="简单的 XHTML 页面" />
        <meta name="Description" content="这是一个简单的 XHTML 页面" />
        <title>简单的 XHTML 页面</title>
    <style type="text/CSS">
    #DIV1:first-letter{
        font-size:200%;
    }
    #DIV1:first-line{
        color:red;
    }
    </style>
    </head>
    <body>
    <div id="DIV1">
        第一行内容，注意这一行的第一个字。<br/>
        第二行内容
    </div>
```

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

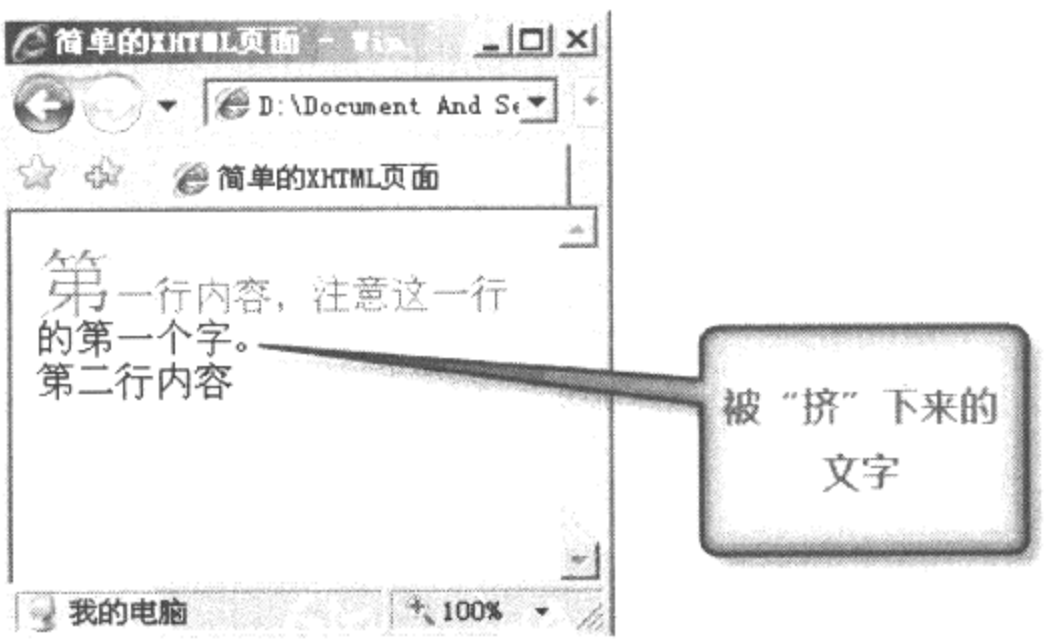
下面是 IE 6 和 IE 7 的显示效果截图。



关于“:first-line”特别注意

1. 伪对象“:first-line”指定是第一行，而不是第一段。行的划分是利用
标签，而“段”的划分是利用<p>标签。“:first-line”并不会对<p>标签划分的段落使用样式。

2. 因为容器的宽度过小，使得第一行“放”不下而“挤”到第二行的内容，将失去伪对象“:first-line”设置的样式。下面的截图更有助于理解。依然是上面的那个示例，当将浏览器的宽度缩小一定程度时，第一行的文字被“挤”到了第二行，同时失去了伪对象“:first-line”指定的样式。



后记

这些 IE 6 不支持、IE 7 才支持的高级选择符，也是搞定 IE 6 和 IE 7 网页

兼容性的有效手段之一。甚至一些朋友直接误以为这些高级选择符就是 CSS hack 的一种。其实这些不是 CSS hack，而是一种改进，一种升级。所以，可以放心使用“高级选择符”手段达到 IE 6 和 IE 7 的兼容，不会有 CSS hack 的后顾之忧，在微软后续的浏览器中一定会支持的。

开动脑筋，利用这些丰富的选择符，将能够实现更多、更强大的效果。IE 7 只是帮我们搭好了舞台，如何唱一出好戏，则是靠大家一起的努力了。

IE 7 改进了很多的东西，“更丰富的选择符支持”只是其中比较耀眼的一颗而已。对于其他的，我将在后续的《IE 7 的 Web 标准之道》系列文章中继续和大家一起探讨，还请各位朋友以后多多指教。

IE 7 标准之道

——2：引起页面布局混乱的祸首

作者：阿一（杨正祎）[http://www.cnblogs.com/JustinYoung]

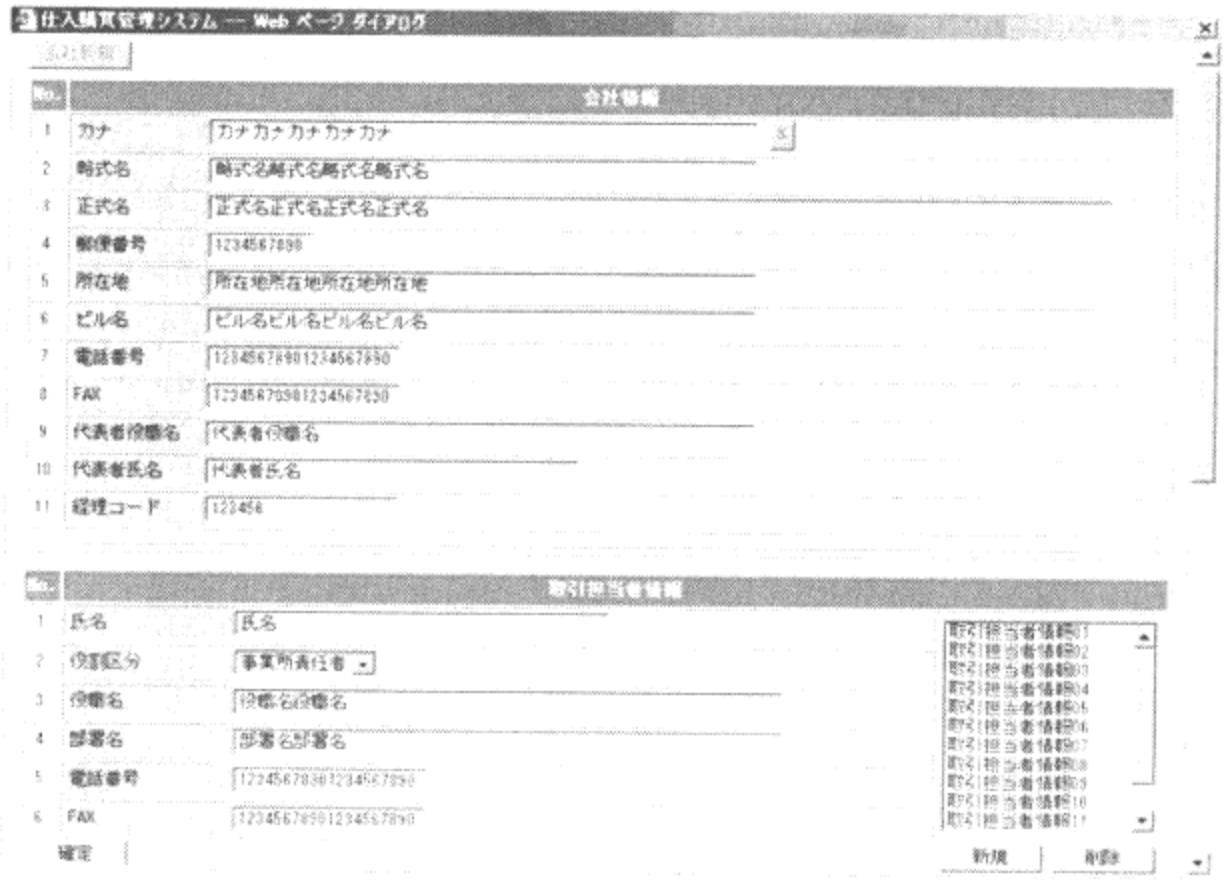
前言

现在，最令网页设计者头痛的问题就是网页在各个浏览器中的兼容性。兼容性差最常见的，也是最令人恐惧的便是“页面布局混乱”。常常一个页面在 IE 6 下显示得非常完美，而到了 IE 7（或者 FireFox）中，则惨得“不堪入目”。到底是什么让这些页面那么地“水土不服”呢？

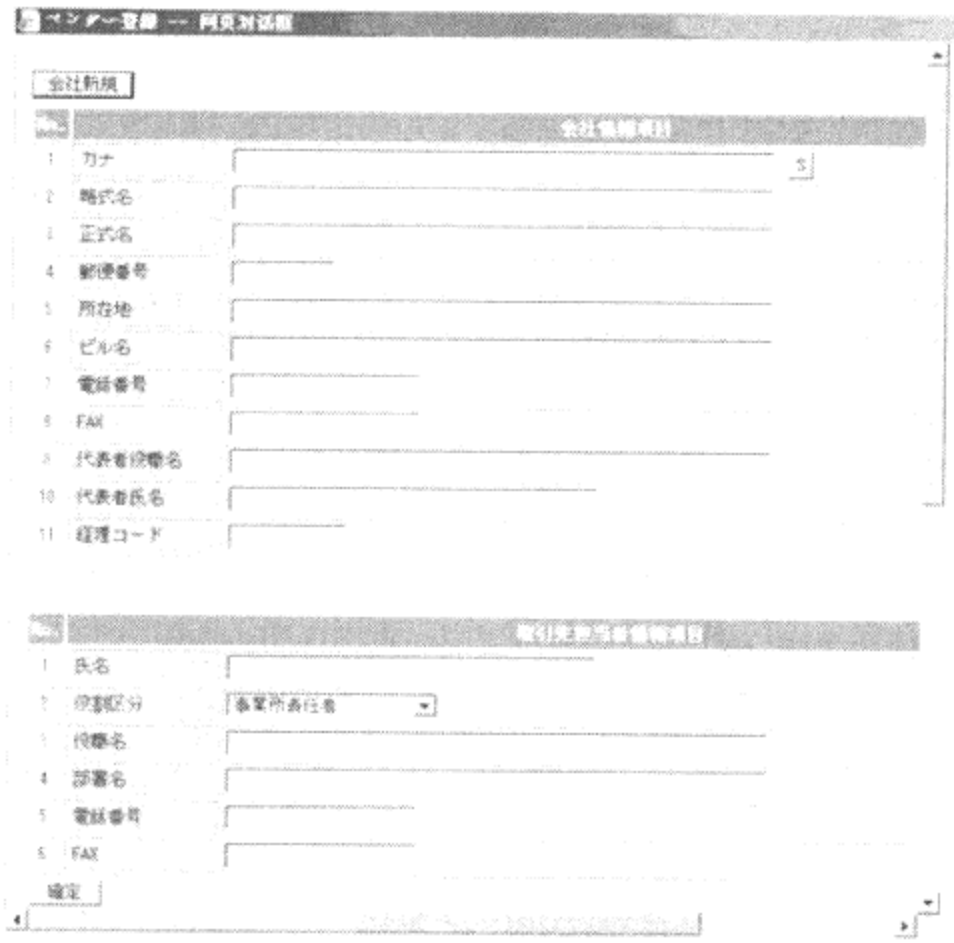
其实这些都是 IE 6 酿下的“恶果”。IE 6 对 Web 标准的支持过于不足，甚至理解得有偏差，才导致了这些页面的“脆弱”。而 IE 7 则修正了很多的“IE 6 对 CSS 解释和渲染”的 bug。这种 bug 有很多。这里只讲其中一个，但却是最重要的一个，很多的“十分”混乱的页面都是它造成的。可以不客气地说，它简直就像“页面布局混乱黑帮”的幕后黑手，是引起页面布局混乱的祸首之一，而且是最大的一个。它就是潜伏在网页背后的“‘overflow:visible’IE 6 渲染 bug”。

“不堪入目”的网页截图

如果只是简单地说“‘overflow:visible’IE 6 渲染 bug”，你可能完全没有印象。但是看看下面的这些“不堪入目”的网页截图，便能引起你心中那无限的伤感……



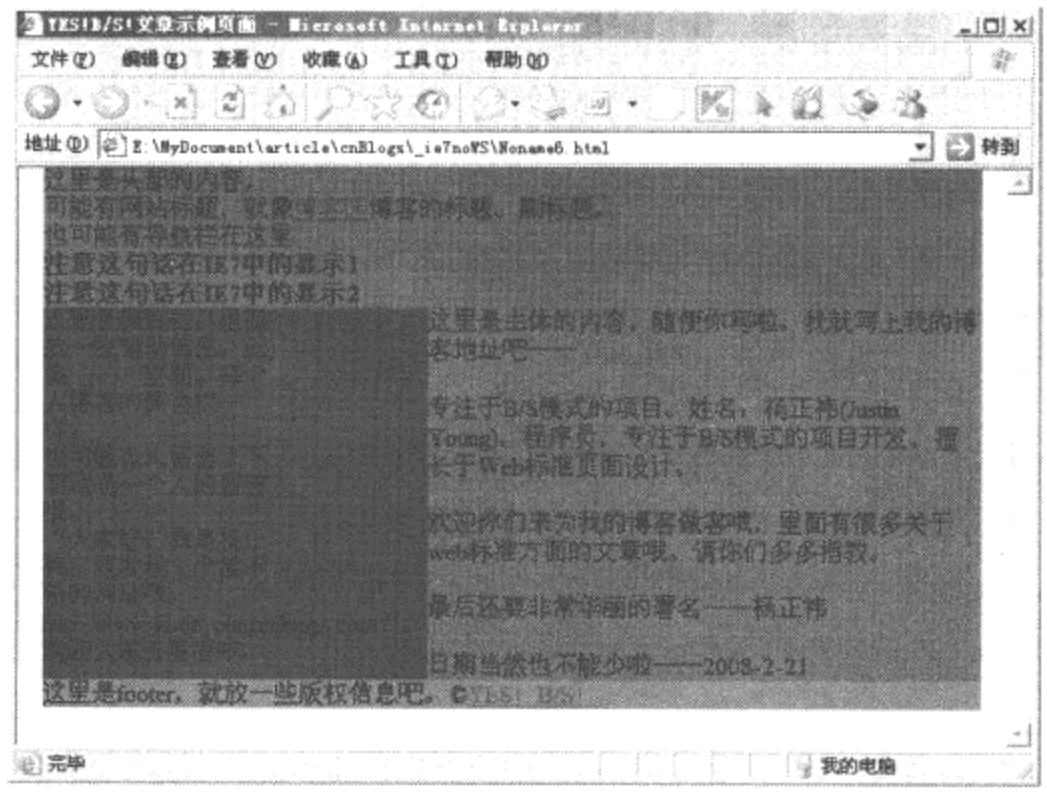
图：这是在 IE 6 中显示的效果截图，“十分完美”



这是在 IE 7 中显示的“不堪入目”的效果截图

上面的两张截图，是我 2007 年在高达软件公司的真实项目截图。可以看出，在 IE 7 下的显示已经严重变形，虽然不影响软件的使用，但是已经严重影响了用户的使用体验（没有人喜欢拖动横向滚动条）。

再看看下面这个网页截图，它是我们今天将要使用的例子（源代码在下方有提供），是一个标准的“上左右下”带侧边栏的虚拟网页。



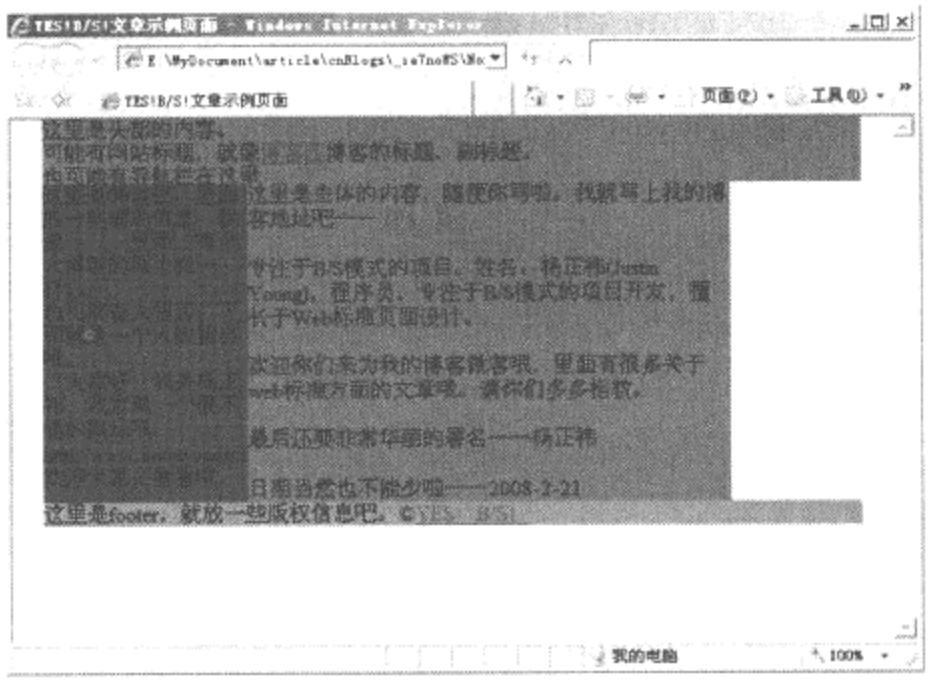
图：这是在 IE 6 中显示的效果截图，还算“整齐”

```
<!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" />
<meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,实例代
码" />
    <meta name="Description" content="这是一个简单 YES!B/S!文章示例页面,
来自杨正祎的博客,
    http://justinyoung.cnblogs.com/" />
    <title>YES!B/S!文章示例页面</title>
    <style>
    body { margin: 0; padding: 0; }
    #header {
    width: 600px;
    height: 50px;
    background-color: red;
    margin:0 auto;/*居中显示*/
    }
    #body{
    width:600px;
    margin:0 auto;/*居中显示*/
    }
    #sideBar{
    width:150px;
    background-color:#0000ff;
```

```
float:left;
height:244px;
}
#main{
width:354px;
float:left;
background-color:green;
height:
}
#footer{
width:600px;
margin:0 auto;
background-color:#666;
}
</style>
</head>
<body>
<div id="header">
这里是头部的内容。<br/ />
可能有网站标题，就像<a target="_blank" href="" title=" title="">博客园</a>博客的标题、副标题。<br/ />
也可能有导航栏在这里<br/ />
<strong>注意这句话在 IE 7 中的显示 1</strong><br/ />
<strong>注意这句话在 IE 7 中的显示 2</strong><br/ />
</div>
<div id="body">
<div id="sideBar">
这里是侧边栏，里面放一些辅助信息。就像<a target="_blank" href="" title="
title="">博客园</a>里面，每个人博客的侧边栏一样。<br/ />
也可能有人留言，下面就是一个人的留言哦。<br/ />
“大家好，我是杨正祎，我发现一个很不错的网址哦。
http://www.justinyoungcnblogs.com”。欢迎大家去看看哦。
</div><!--end: sideBar -->
<div id="main">
这里是主体的内容，随便你写啦。我就写上我的博客地址吧—<a target="_blank"
href="http://justinyoung.cnblogs.com/" title="IE 7 的 Web 标准之道">YES!
B/S! </a>
<p> 专注于 B/S 模式的项目。姓名：杨正祎(Justin Young)，程序员，专注于 B/S
模式的项目开发，擅长于 Web 标准页面设计。</p>
<p>欢迎你们来为我的博客做客哦，里面有很多关于 Web 标准方面的文章哦。请你们多多
指教。</p>
<p>最后还要非常华丽的署名—杨正祎</p>
<p>日期当然也不能少啦—2008-2-21</p>
```

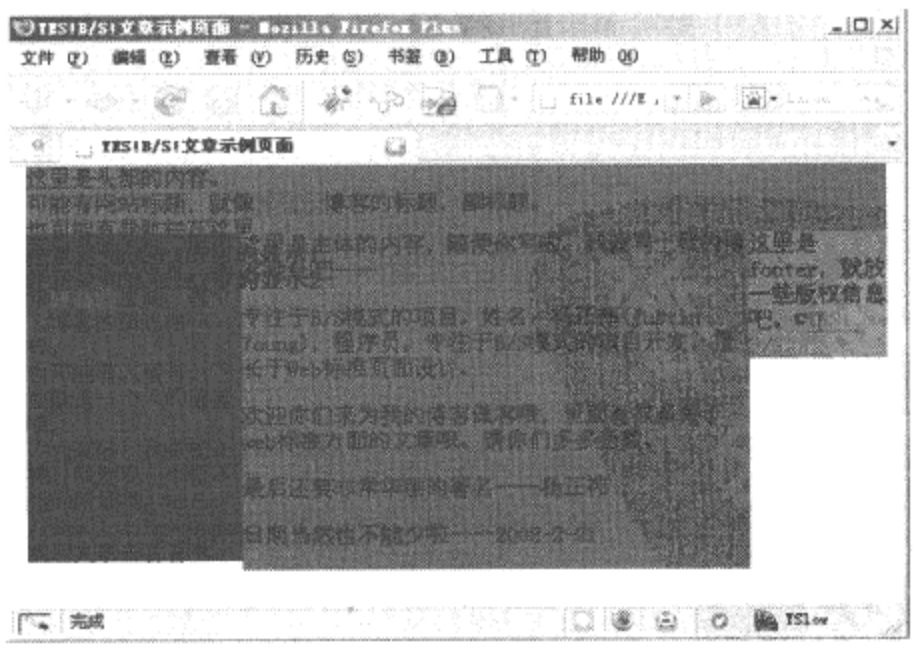
```
    </div><!--end: main -->
    </div><!--end: body -->
    <div id="footer">
    这里是 footer，就放一些版权信息吧。©<a target="_blank"
href="http://justinyoung.cnblogs.com/" title="IE 7 的 Web 标准之道">YES!
B/S! </a>
    </div><!--end: footer -->
    </body>
    </html>
```

当你看到这个页面在 IE 7 下显示的效果图的时候，可能便会大吃一惊了。



图：这是在 IE 7 中显示的效果截图，已经“不堪入目”了

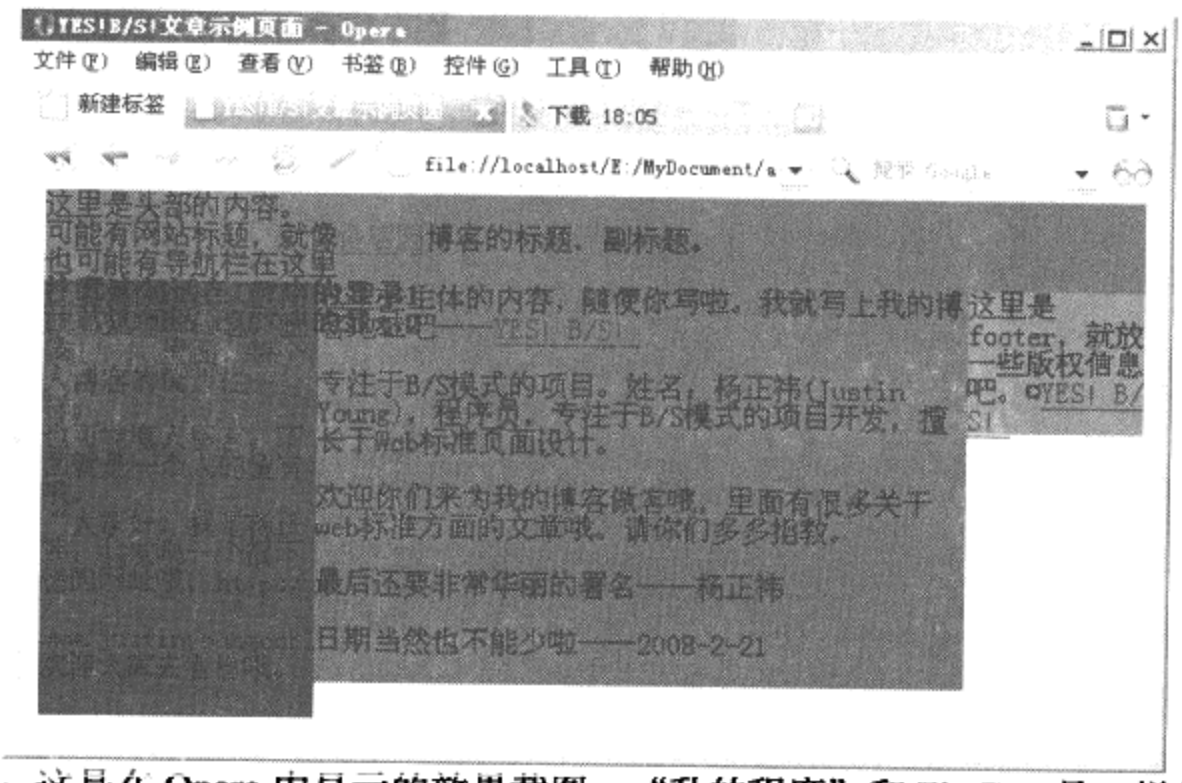
我们惊讶地看到，网页“头部”变“矮”了，最后两句重要的句子“消失”了；侧边栏变“窄”了，那个重要的网址的后半部消失了（其实是被右边绿色的区域遮盖住了）；而最令人沮丧的是，右面“缺了个大口子”。原本整齐的布局，已经完全消失，出现的是一个“一塌糊涂”的页面。到底是什么将一个原本好好的页面“糟蹋”成这样？



图：这是在 FireFox 中显示的效果截图，已经乱的“令人抓狂”了

为什么在 FireFox 又有这么令人抓狂的显示呢？原来，这便是 IE 7 的 Web 标准之道的精髓了。随着 Web 标准的推广和认可度的提高，IE 7 必须向 Web 标准靠拢，但是又必须兼顾到那些在 IE 6 中还显示正常的亿万个已经存在的页面。这样矛盾就产生了—遵循标准就意味着页面会显示得乱七八糟，甚至无法浏览；但是如果太过于兼容 IE 6 的那些烂摊子网页，又必然会离 Web 标准越来越远。于是 IE 7 走出了自己的 Web 标准之道—绝对重视 Web 标准，又稍微兼顾 IE 6 的烂摊子。于是，IE 7 显示的那个页面虽然已经乱了，但是还不像在 FireFox 中显示的那样令人抓狂。

附：测试页面在 Opera（版本 9.25）中的显示效果截图（写文章的时候 Opera 正好有了新的升级版本）。



图：这是在 Opera 中显示的效果截图，“乱的程度”和 FireFox 是一样的

“非也，非也”

“千万别用 IE 7，IE 7 太垃圾了，浏览页面会出现布局混乱，一些在 IE 6 中显示好好的页面，用 IE 7 浏览布局就会混乱。”这种言论在网上会经常见到，好像是 IE 7 才导致了那些页面的混乱。其实，非也，非也。

悟空说：“师父快快回避，且待我一棒打死这妖精！”

八戒说：“师父，那个姑娘俊俏得很，怎么会是妖怪呢？大师兄他骗人的！”

唐僧说：“那位施主只是一平常人家的姑娘，定然不会是什么妖魔鬼怪。悟空你休得胡言乱语。”

白骨精说：“ohYeah！2 比 1，看来这下安全了！”

那些 IE 7 浏览时会出现布局混乱的页面，就如同利用妖术变了身的白骨精一样，在八戒和唐僧的凡眼看来就是个俊俏的姑娘。但是在猴哥的“火眼

金睛”下便立刻现了形。而那些布局和样式隐藏着 Bug 的页面，在 IE 6 的袒护和包庇下化身成“完美页面”，招摇过市。但是在 IE 7 的严厉的审核下，自然“原形毕露”、“Bug 层出”，从而导致布局混乱。但是，令人遗憾的却是—IE 7 被那些不知情的“凡骨俗胎”的人们咒骂、贬低、踩在脚底……这是一出悲剧！

表面原因——放纵的孩子和严厉的父亲

在 2007 年 5 月份的时候，我曾经写过一篇文章叫做《IE 6 与 IE 7，放纵的孩子与严厉的父亲》[①]。当时技术水平和对 Web 标准的认知有限，所以写出来的这篇文章虽然从表面上合理地解释了造成布局混乱的原因，但是并没有说到根本。可谓“只知其一，不知其二”，但是这里依然推荐你阅读。因为“先知其一，再知其二”将更有助于理解这“其二”。

“其二”原因——IE 6 对“overflow:visible”的误解

为了兼顾到对 overflow 可能还不是很了解的朋友，这里是 W3C 关于 overflow 的资料。请注意 W3C 对于 visible 参数的解释。

Visible: "This value indicates that content is not clipped, i.e.,it may be rendered outside the block box."（注：后面这句可能是后续版本补充上来的）

注： W3C 只是说超出容器的内部不会被剪切。但是它并没有说超出来的内容可以“撑开”容器。所以下面这个例子中 IE 7 和 FireFox 的解释和渲染是正确的，而 IE 6 则是错误的（因为它错误的认为，只有让容器内的内容“撑开”容器，才能让容器内的内容在超出时不被剪切）。

```
    <!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=u
tf-8" />
       <meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,实
例代码" />
       <title>YES!B/S!文章示例页面</title>
```

[①] http://www.cnblogs.com/JustinYoung/articles/754637.html

```
        <style type="text/CSS">
        #DIV1{ border:1px solid red; width:40px; }
    </style>
    </head>
    <body>
    <div id="DIV1">
     alonglonglonglonglonglonglonglongword from <a href="http://
justinyoung.cnblogs.com/
    " title="">http://justinyoung.cnblogs.com/</a>
    </div>
    </body>
    </html>
```

下面是上面示例分别在 IE 6、IE 7、FireFox（版本 2.0.0，12）和 Oepra（版本 9.25）中的显示效果截图。

从图片中我们可以看到 IE 7 和 FireFox 的渲染结果是一样，IE 6 是个“坏孩子”，而 Opera 的渲染结果和 FireFox 以及 IE 7 也是有点差距的。但是这不是因为对 overflow 样式的理解有误差造成的。

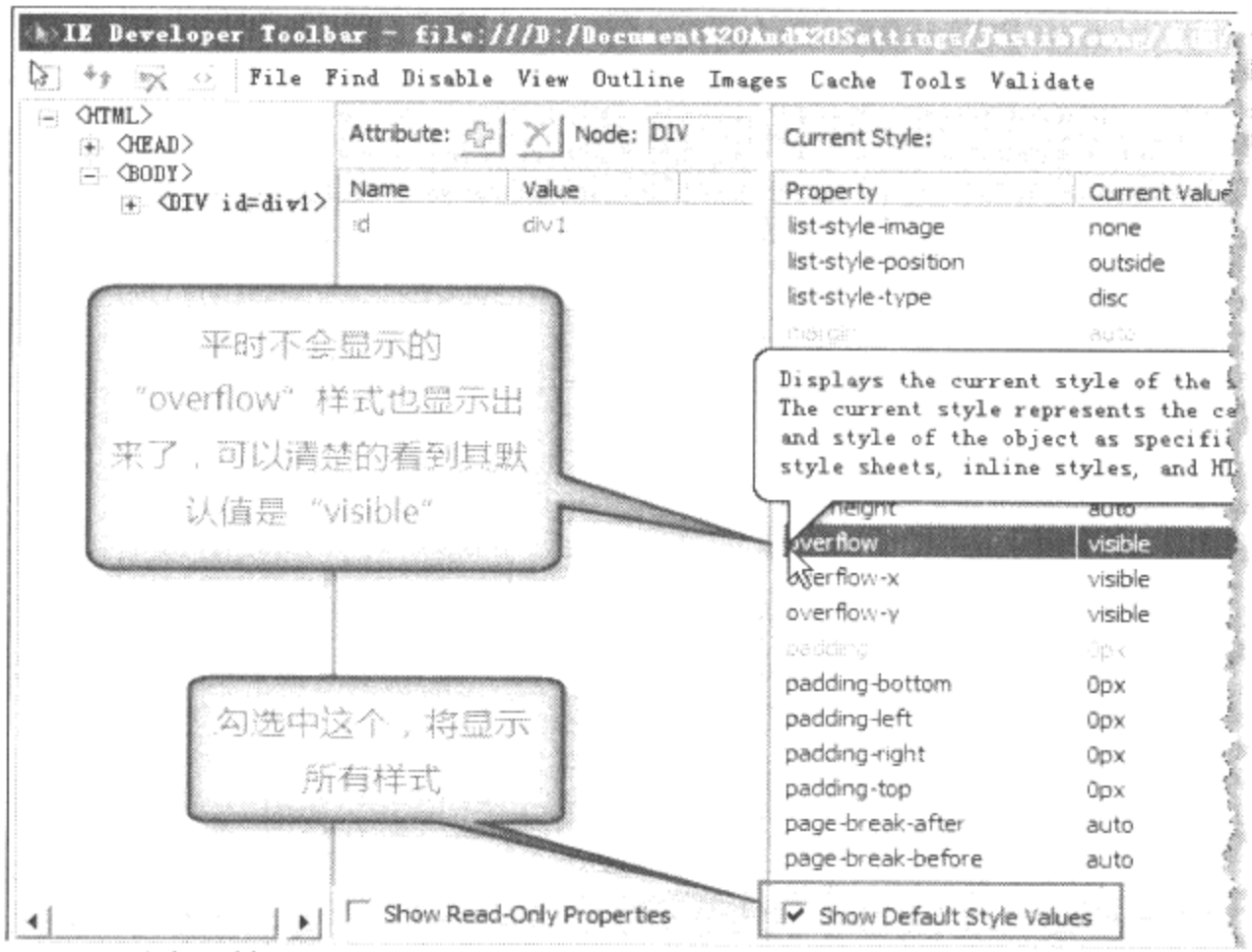
何以称之为“祸首”

这篇文章的题目中，将这个 bug 称之为“引起页面布局混乱的祸首”。能被称之为“祸首”，自然有其“强悍”的地方。那它到底强悍在什么地方呢？其实很简单，就 3 条。

- 无论是“宽度”的内容过长，还是“高度”的内容过长，都会引发此 bug。
- 无论是文字、图片，还是任意有宽度和高度概念的“可见元素”，它们的“过宽”和“过高”都会引发此 bug。
- 任意有宽度和高度概念的“可见元素”，它们在默认状态下的“overflow”样式的值都是“visible”（即使没有设置这个样式）。

有些朋友可能会问，你怎么知道任意有宽度和高度概念的“可见元素”，它们在默认状态下的“overflow”样式的值都是“visible”的呢?

其实方法很简单，利用 IE Developer Toolbar 这个工具就可以知道了。



图：利用“IE Developer Toolbar”得到元素样式的默认值

如何修复 bug

其实这个 bug，我们还是有办法修复的，但都不是很完美的解决方案，想要取得较好的效果，还需要一些技巧。下面便是我工作中总结的一套解决方案。

修正这个 bug 首先要洗脑一下，因为错误的认识不利于理解解决方法。

1. 虽然，那个虚拟的示例网页在 IE 6 中能够“完美的”显示，但是它并不是正确的。我们不能通过 CSS hack 的方法让它在 FireFox 和 IE 7 中显示“靠近”IE 6，而是应该“扒下”IE 6 的那层虚假的“皮”，重新塑造网页，从而让它在 IE 6、IE 7 和 FireFox 中都能正常显示。

2. 就算让网页在 IE 6、IE 7 和 FireFox 中，都可以正常显示，但也未必

就是最终想要的效果。

3. 为了达到最终想要的结果，可能需要使用不推荐使用的措施—CSS hack。

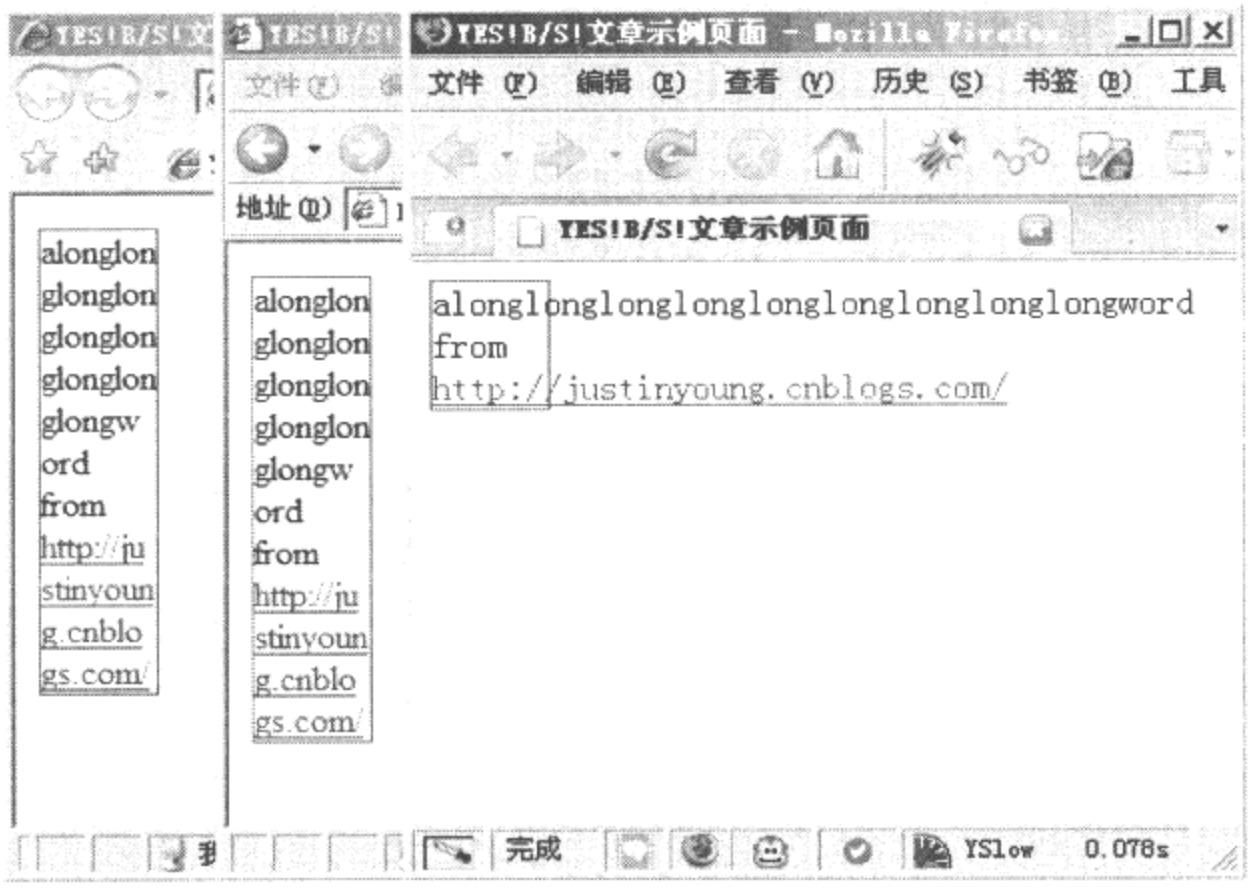
如何解决“横向撑开”问题

用“word-wrap: break-word”解决

布局混乱的主要原因是 IE 6 对 overflow 的 visible 的错误解释，导致宽度被“撑开”。所以，我们必须采取措施让 IE 6 中容器不能那么“放纵孩子”。方法就是使用“word-wrap: break-word”样式（IE 特有，FireFox 不起任何作用），强制要求容器内的内容不允许“撑开”父容器。看下面的示例可能有助于理解。

```
    <!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=u
tf-8" />
       <meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,实
例代码" />
       <meta name="Description" content="这是一个简单 YES!B/S!文章示例页
面,来自杨正祎的博客,
    http://justinyoung.cnblogs.com/" />
       <title>YES!B/S!文章示例页面</title>
       <style type="text/CSS">
       #DIV1{ border:1px solid red; width:50px;
word-wrap:break-word; }    </style>
    </head>
    <body>
    <div id="DIV1">
     alonglonglonglonglonglonglonglongword from <a href="http://
justinyoung.cnblogs.com/
    " title="">http://justinyoung.cnblogs.com/</a>
    </div>
    </body>
    </html>
```

利用“word-wrap: break-word”可以让 IE 6 中的“孩子”乖乖地待在“父亲”的允许访问内。如下图所示。



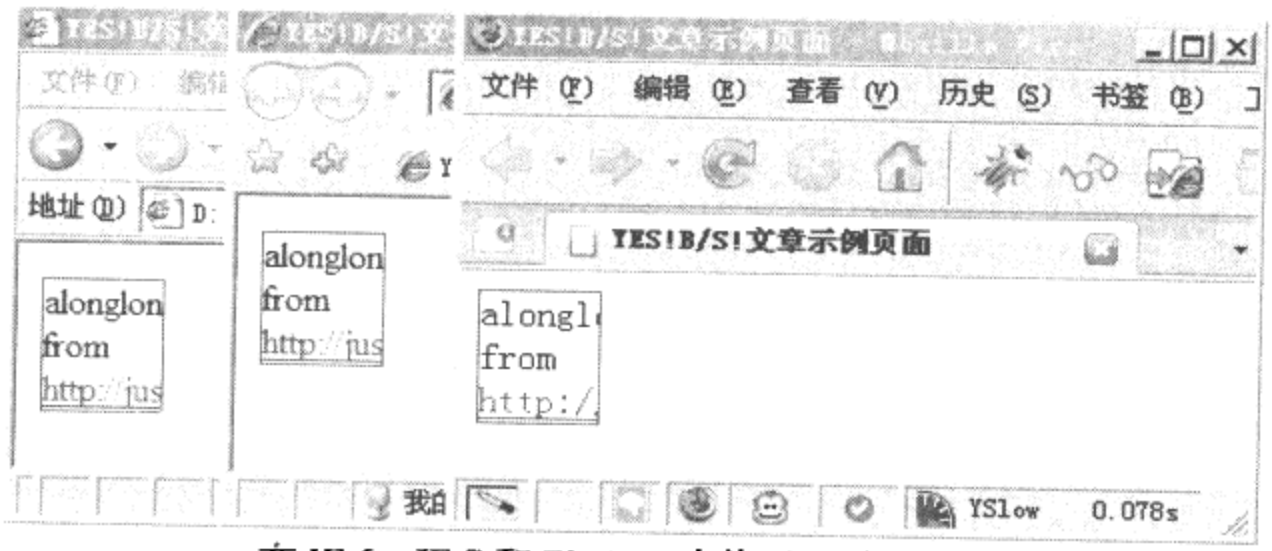
用“overflow: hidden”解决

显然，用“word-wrap: break-word”又导致了IE（IE 6和IE 7）和FireFox的显示结果不一致。那还有没有其他的办法呢？“擒贼先擒王”，既然是“overflow: visible”导致的bug，那直接改变“overflow”的值不就可以了吗？所以，使用“overflow: hidden”便能让IE 6、IE 7和FireFox显示一致。下面的这个示例，可能会有助于你的理解。

```
    <!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=u
tf-8" />
        <meta name="Keywords" content="YES!B/S!,Web标准,杨正祎,博客园,实
例代码" />
        <meta name="Description" content="这是一个简单YES!B/S!文章示例页
面,来自杨正祎的博客,
    http://justinyoung.cnblogs.com/" />
        <title>YES!B/S!文章示例页面</title>
        <style type="text/CSS">
          #DIV1{
          border:1px solid red;
          width:50px;
          overflow:hidden;
```

```
        }
      </style>
    </head>
    <body>
    <div id="DIV1">
     alonglonglonglonglonglonglonglongword from <a href="http://
justinyoung.cnblogs.com/"
     title="">http://justinyoung.cnblogs.com/</a>
    </div>
    </body>
    </html>
```

下面是在 IE 6、IE 7 和 FireFox 中的现实效果截图。



在 IE 6、IE 7 和 FireFox 中终于显示一致了

一个大问题与残缺的美丽

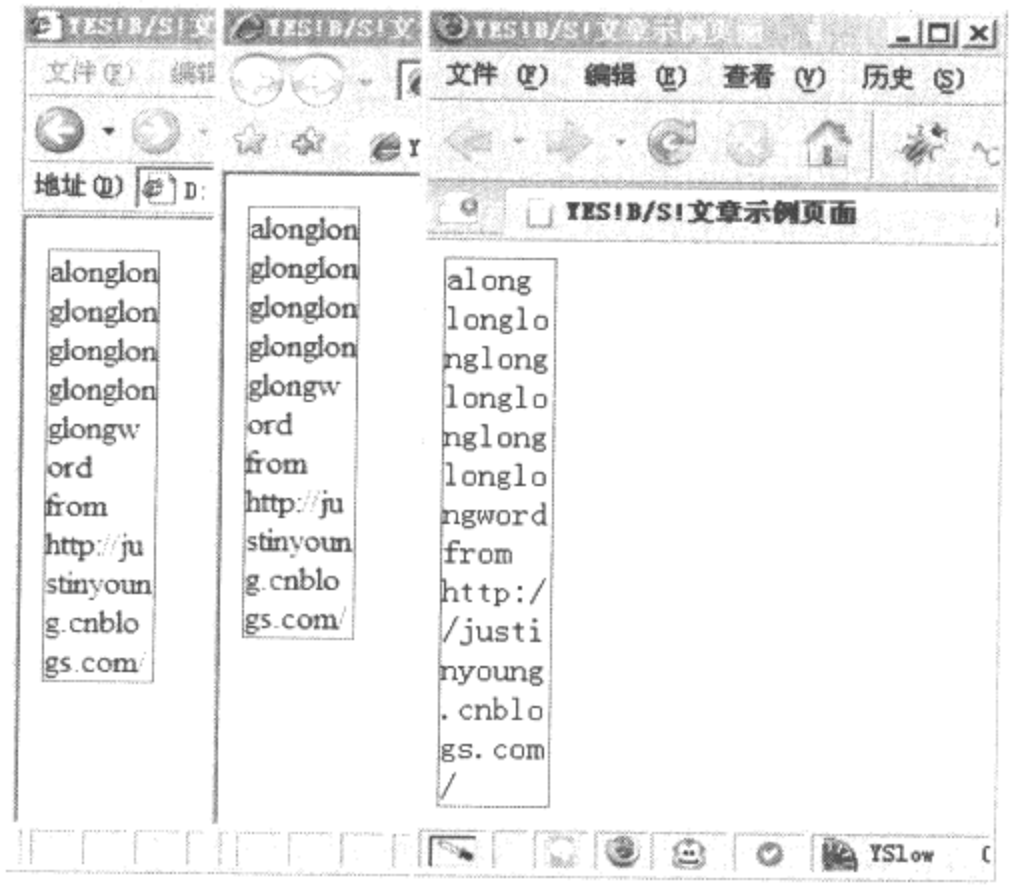
从截图看，网页在 IE 6、IE 7 和 FireFox 中的确显示一致了（就布局显示而言）。但是，却发现了一个大问题！那就是—这并不是我想要的结果。假使这里的 DIV 是一个侧边栏，我们只是要求它老老实实地那么“宽”，不要乱“撑”宽度就可以了，内容我们还是要看的，你不能把内容都剪切了啊！

如何让“很长文字”换行显示呢？其实在前面我们已经使用到了，那就是“word-wrap: break-word”。虽然它是 IE 的特有样式，但是足以先解决 IE 6 和 IE 7 中的问题。而 FireFox 中没有这个样式，那 FireFox 下如何使“很长文字”自动换行显示呢？我们遗憾地发现 FireFox 并没有提供类似的样式供我们使用，目前唯一的解决方案是利用 JavaScript 实现。原理很简单，就是根据宽度，将文本截取成多段，在每段后面强制加上换行符。下面的实现示例可能会有助于你的理解。

```
    <!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=u
tf-8" />
       <meta name="Keywords" content="YES!B/S!,Web标准,杨正祎,博客园,实
例代码" />
       <meta name="Description" content="这是一个简单YES!B/S!文章示例页
面,来自杨正祎的博客,
    http://justinyoung.cnblogs.com/" />
       <title>YES!B/S!文章示例页面</title>
       <style type="text/CSS">
        #DIV1{
        border:1px solid red;
        width:50px;
        word-wrap: break-word;
        }
    </style>
    </head>
    <body>
    <div id="DIV1">
    alonglonglonglonglonglonglonglongword from http://justinyoun
g.cnblogs.com/
    </div>
    <script type="text/javascript">
      if(document.getElementById  &&  !document.all)  wordWarp4ff(6)
/*数值6根据宽度需要发生变化*/
      function wordWarp4ff(intLen){
      var obj=document.getElementById("DIV1");
      var strContent=obj.innerHTML;
      var strTemp="";
      while(strContent.length>intLen){
      strTemp+=strContent.substr(0,intLen)+" ";
      strContent=strContent.substr(intLen,strContent.length);
      }
      strTemp+=" "+strContent;
      obj.innerHTML=strTemp;
      }
    </script>
    </body>
    </html>
```

看着下面的截图，终于能既满足要求，又在IE 6、IE 7和FireFox中显示一致了！



但是，如同残缺的美丽，惊艳的美隐藏着巨大的缺憾。如果容器中的内容不是文字而是图片时，这种方法将无能为力。只能将容器放宽，或者缩小图片，当然，也可以使用“overflow: hidden”将超出的内容剪切掉。另一个遗憾是—在 FireFox 中，DIV1 容器里面的标签和样式也将失去，只留下文本。

另一个“焦油坑”——“纵向撑开”

上面方法解决的只是“横向”的、宽度的问题，其实“‘overflow:visible’IE 6 渲染 bug”，同样也会引起纵向的、高度方面的页面布局混乱。解决“纵向撑开 bug”和解决“横向撑开 bug”需要采用完全不同的解决方案。但是，相比“纵向撑开 bug”解决方案，“横向撑开 bug”解决方案却简单很多—只要我们让 IE 7 和 FireFox 也能像 IE 6 中那样根据内容自适应高度即可。如何才能让容器在 IE 7 和 FireFox 中能够自适应高度呢？其实很简单，也是 IE 7 的重要改进之一，使用“min-height”样式。虽然 IE 7 中已经支持“min-height/min-width”和"max-height/max-width"样式。但是 IE 6 却不认识这些"min-"、"max-"开头的样式，所以我们还需要使用一个 CSS hack 为 IE 6 设置 height，只让 IE 6 认识，IE 7 和 FireFox 都不认识。通过这篇文章《实例讲解符合中国特色的和网络现状的实用 CSS Hack（附源码）》[①]便可以找到应该使用的 CSS hack。下面的示例可能会有助于你理解。

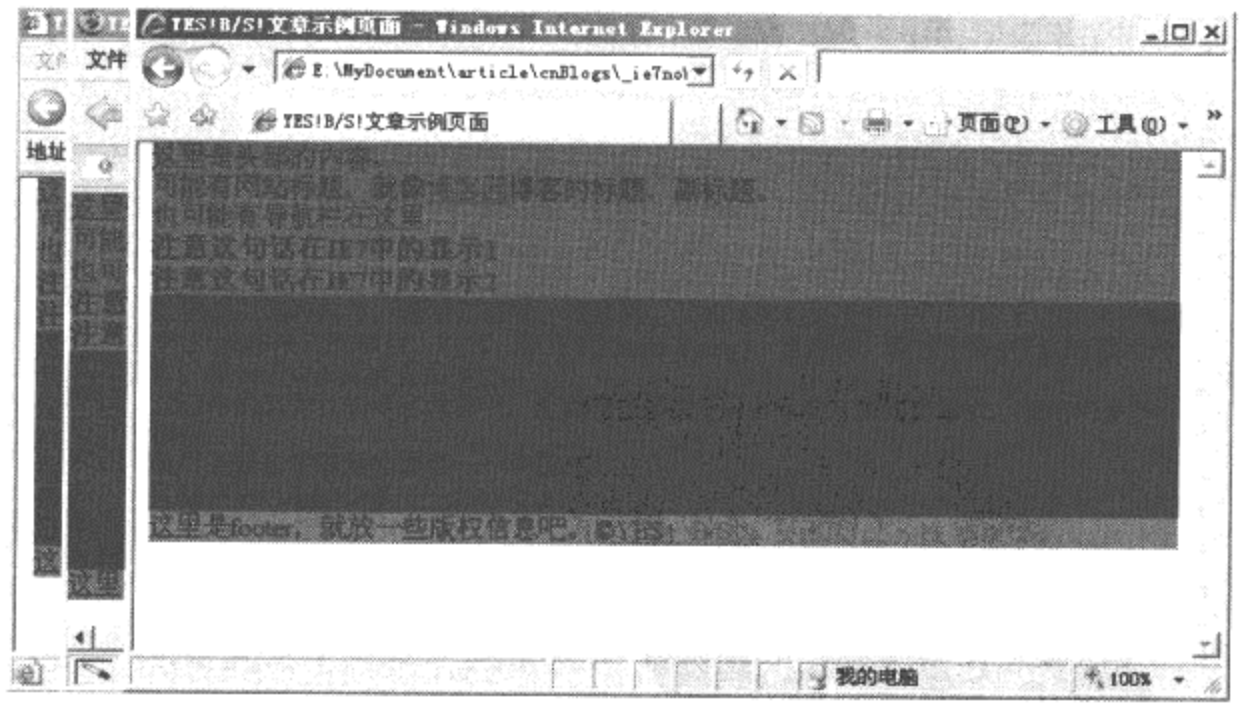
```
<!DOCTYPE html public "-//W3C//DTD XHTML 1.0 Transitional//EN" "
```

[①] http://www.cnblogs.com/JustinYoung/archive/2007/09/14/892414.html

```
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=u
tf-8" />
       <meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,实
例代码" />
       <meta name="Description" content="这是一个简单 YES!B/S!文章示例页
面,来自杨正祎的博客,
    http://justinyoung.cnblogs.com/" />
       <title>YES!B/S!文章示例页面</title>
    <style>
    * { margin: 0; padding: 0; }
    #header {
       width: 600px;
       /*height:50px;注释掉下面两句,放出这一句,在 FireFox 和 IE 7 中便能呈现 bug*/
       min-height:50px;/*只设置最小高度，让 IE 7 和 FireFox 自适应高度*/
       _height: 50px;/*采用只有 IE 6 才认识到 CSS hack，让不认识 min-height
的 IE 6 也有很好的兼容性。*/
       background-color: red;
       margin:0 auto;/*居中显示*/
    }
    #body{
       width:600px;
       margin:0 auto;/*居中显示*/
       background-color:blue;
    }
    #footer{
       width:600px;
       margin:0 auto;
       background-color:#666;
       clear:both;/*clear:both 让 footer 在新的一行显示，很多朋友对 clear 理
解得不够透彻，我以后会特意出篇文章介绍这个样式，有兴趣的朋友可以关注我的博客。
http://justinyoung.cnblogs.com*/
    }
    </style>
    </head>
    <body>
    <div id="header">
       这里是头部的内容。<br/>
       可能有网站标题，就像<a target="_blank" href="" title="">博客园</a>
博客的标题、副标题。
```

```
    <br/>
        也可能有导航栏在这里<br/>
        <strong>注意这句话在 IE 7 中的显示 1</strong><br/>
        <strong>注意这句话在 IE 7 中的显示 2</strong><br/>
    </div>
    <div id="body">  这里是主体的内容，随便你写啦。我就写上我的博客地址吧—
<a target="_blank"
 href="http://justinyoung.cnblogs.com/" title="IE 7 的 Web 标准之道">YES!
B/S! </a>
     <p> 专注于 B/S 模式的项目。姓名：杨正祎(Justin Young)，程序员，专注于 B/S
模式的项目开发，擅长于 Web 标准页面设计。</p>
     <p>欢迎你们来为我的博客做客哦，里面有很多关于 Web 标准方面的文章哦。请你们多
多指教。</p>
     <p>最后还要非常华丽的署名—杨正祎</p>
     <p>日期当然也不能少啦—2008-2-21</p>
    </div><!--end: body -->
    <div id="footer">
     这里是 footer，就放一些版权信息吧。&copy;<a target="_blank" href="
    http://justinyoung.cnblogs.com/" title="IE 7 的 Web 标准之道">YES!
B/S! </a>
    </div><!--end: footer -->
    </body>
    </html>
```

下面是修正后页面的效果截图，在 IE 6、IE 7 和 FireFox 中显示结果都是令人满意的。利用 min-height 和 CSS hack 让容器在 IE 7 和 FireFox 中自适应高度。



IE 7 标准之道

——3：歌剧院魅影 bug

作者：阿一（杨正祎）[http://www.cnblogs.com/JustinYoung]

打倒标题党

估计很多朋友都是因为这个华丽的“歌剧院魅影”一词进来的。其实这纯粹是一个标题党作为，纯粹是吸引眼球而已。其实这个 bug 和歌剧院半毛钱关系都没有。这个 bug 在国际上比较获得认可的名字叫做—“IE 6 重复文字 bug”。这是一个非常好玩但是又很令人摸不到头脑的 bug。如果你不知道产生原因的话，将会非常头痛。这也是我在现实工作中真正遇到过的情况。

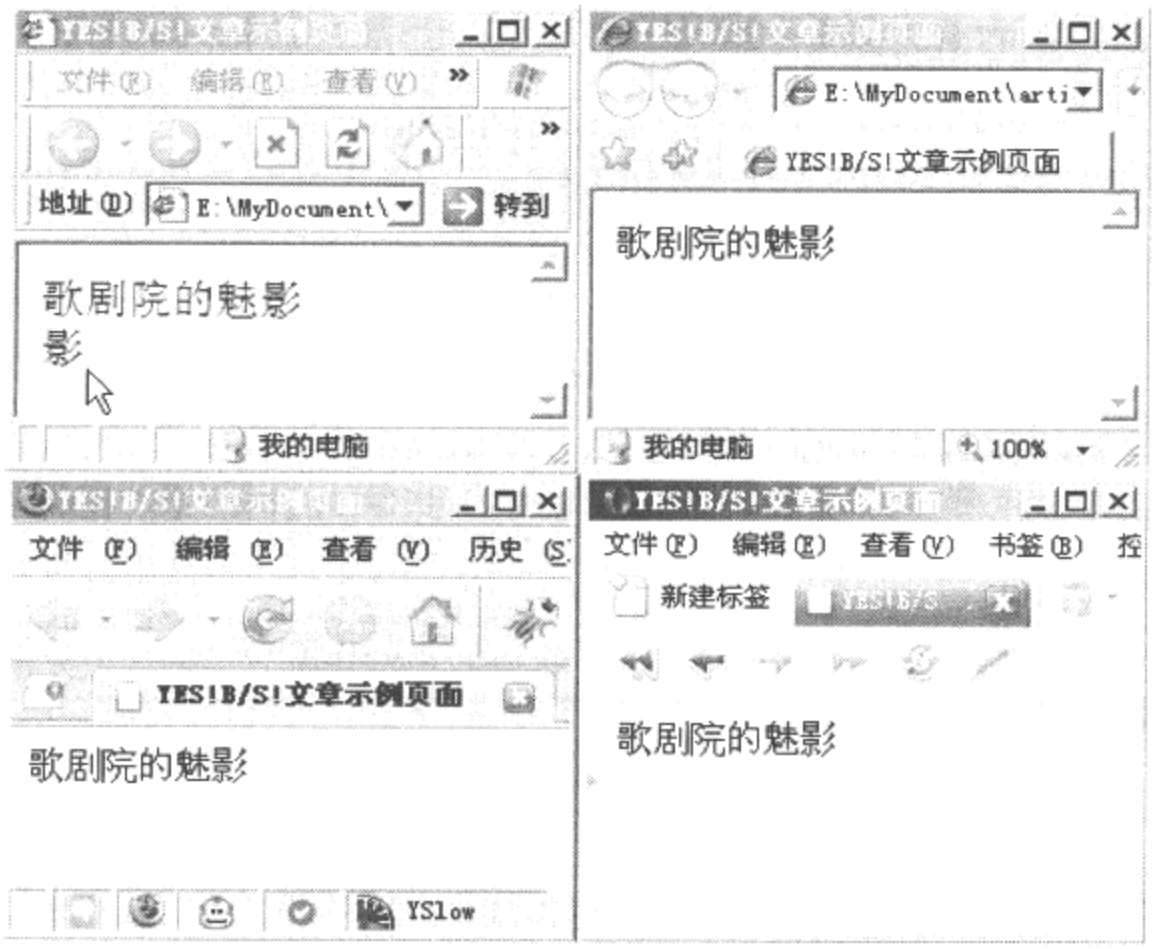
魅影再现

在 IE 6 浏览器下运行下面的代码，便可以重现 bug。IE 7 已经修正了此 bug。

```
<!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" />
<meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,实例代
码" />
<meta name="Description" content="这是一个简单 YES!B/S!文章示例页面,
来自杨正祎的博客,
http://justinyoung.cnblogs.com/" />
<title>YES!B/S!文章示例页面</title>
</head>
<body>
<div style="width:200px;">
<div style="float:left;"></div>
```

```
<!-- 如果是 IE 6，你将多看到一个“影”字 -->
<div style="float:left;width:200px;">歌剧院的魅影</div>
</div>
</body>
</html>
```

下面是上面测试页面分别在 IE 6，IE 7，FireFox（版本 2.0.0，12）和 Oepra（版本 9.25）中的显示效果截图。注意 IE 6 中光标所指位置。



通过截图，你会惊讶地看到在 IE 6 中多出了一个“影”字。下面来讲讲出现这个“影”字的一些条件（bug 重现条件）。

- 一个容器包含两个具有“float”样式的子容器。
- 第二个容器的宽度大于父容器的宽度，或者父容器宽度减去第二个容器宽度的值小于 3（说到 3，这里稍微多说一句——IE 7 还修正了 IE 6 中的一个 bug，bug 名字就叫做“3 像素 bug”）。
- 在第二个容器前存在注释（这也是为什么此 bug 也叫做“IE 6 注释 bug”的原因）。
- 其他的，我暂时尚未发现的条件。

为何会出现魅影

bug 虽然的的确确存在，但是为什么会出现这样的 bug 依然没有统一的定论。不同的高手也是各执一词，谁也说服不了谁。真正的原因也许只有当时

的 IE 6 团队才能道出来，但是现在仍然没有官方的说法。下面列出来的这两种说法，只是现在网上认可度比较高的而已。

- IE 6 浏览器对<!-- -->注释的解释存在 bug 引起的。
- “3 像素 bug”的扩展后遗症（“3 像素 bug”我们将在《IE 7 的 Web 标准之道》系列以后的文章中讲到）。
- 其他的一些说法。

如何消灭魅影

引起的原因也许我们可以不知道，但是如何去消除却是我们一定要关注的。

“歌剧院魅影 bug”已经在 IE 7 中得到修正，在 FireFox 和 Opera 中也不会出现，所以 bug 的修正主要是针对 IE 6 的。

针对于上文中讲到的“bug 重现条件”，如果要修正 bug，只要让任何一个条件不满足即可。

- 改变结构，不出现【一个容器包含 2 两个具有“float”样式的子容器】的结构。

  ——此解决方案的评论：疯了！因噎废食的做法。
- 减小第二个容器的宽度，使父容器宽度减去第二个容器宽度的值大于 3，例如将本文示例中第二个子容器的宽度改为 197px。

  ——此解决方案的评论：在满足页面布局的前提下可以使用。但是当情况比较复杂的时候，可能实施起来比较困难。
- 去掉所有的注释。

  ——此解决方案的评论：最直接的做法，但是“没有注释的代码”，的确不是一个好的代码写作习惯。
- 修正注释的写法。将 <!-- 这里是注释内容 -->写成<!--[if !IE]>这里是注释内容<![endif]-->。

  ——此解决方案的评论：还不错的解决方案，但是并不是每个人都对<!--[if !IE]>这里是注释内容[endif]-->这种注释写法很欣赏。
- 在第二个容器后面加一个或者多个 来解决。

  ——此解决方案的评论：令人感觉很不爽的解决方案，的确能解决问题，但影响网页效率。
- 其他你提供的方法。

关于此 bug 的一些文章资料

其实很早以前就有外国的朋友关注过这个 bug，而且在中国也有一些朋友

关注过这个 bug。我在写这篇文章的时候，也一定程度上参照了他们的研究成果，在此向研究此问题的前辈们表示感谢。下面是两篇研究此 bug 的文章，希望对你有进一步的帮助。

- Holly 'n John 于 2004 年 2 月 18 号发表的一篇文章:《Explorer 6 Duplicate Characters Bug 》[①]，这是关于此 bug 比较权威的一篇文章。
- 经典论坛版主怿飞的《注释在 IE 中造成文字溢出的研究》[②]。顺便说一下怿飞是一个在 Web 标准方便很有研究的朋友。虽然没有直接和他接触过，但是却一直拜读他的文章。

① http://www.positioniseverything.net/explorer/dup-characters.html
② http://www.planabc.net/2006/10/06/comment_ie_bug/

IE 7 标准之道

——4：上去了！终于上去了！

作者：阿一（杨正祎）[http://www.cnblogs.com/JustinYoung]

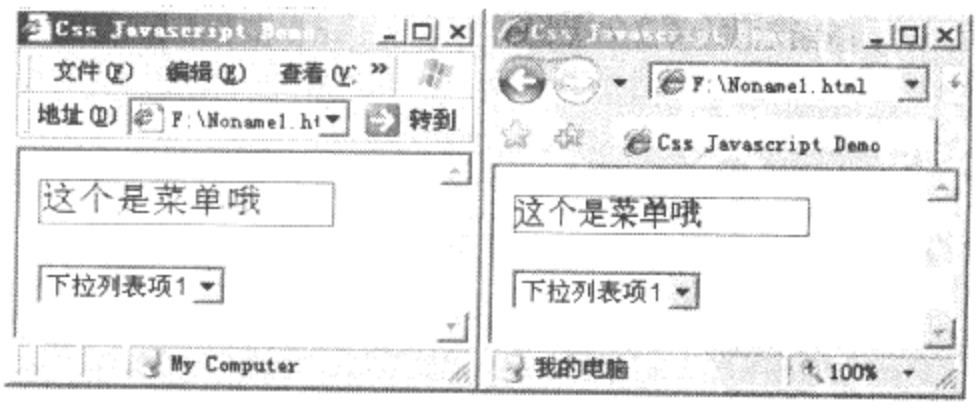
select 对 DIV 说："小样！就踩着你！"

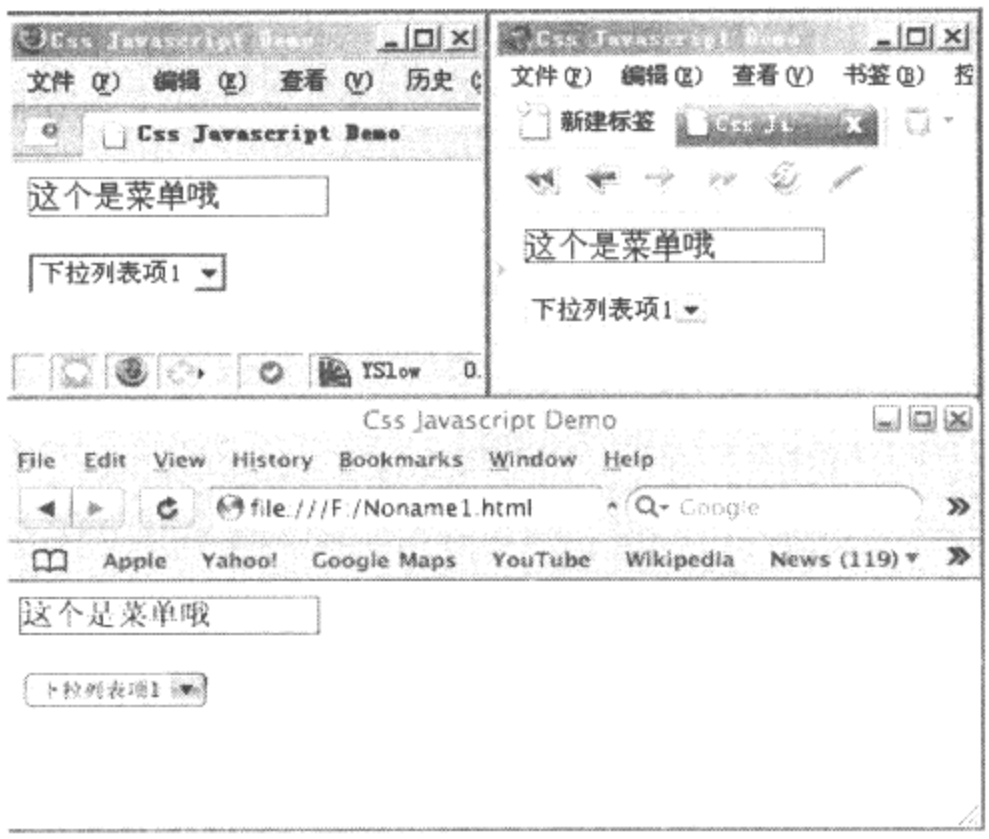
这个 IE 6 中很著名且诡异的 bug 很简单，也很容易重现，说白了就是列表框（select）一直把 DIV 踩在脚底下。因为这个 bug，不知道多少浮动菜单被破坏。下面就模拟了这种情景，你可以运行代码来查看示例页面。

```
<!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" />
<meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,实例代码" />
<meta name="Description" content="这是一个简单 YES!B/S!文章示例页面,
来自杨正祎的博客,
http://justinyoung.cnblogs.com/" />
<title>YES!B/S!文章示例页面</title>
<style type="text/CSS">
#DIVMenu{
border:1px solid red;
width:150px;
}
#DIVUp{
position:absolute;
background-color:red;
width:100;
height:100px;
display:none;
}
</style>
```

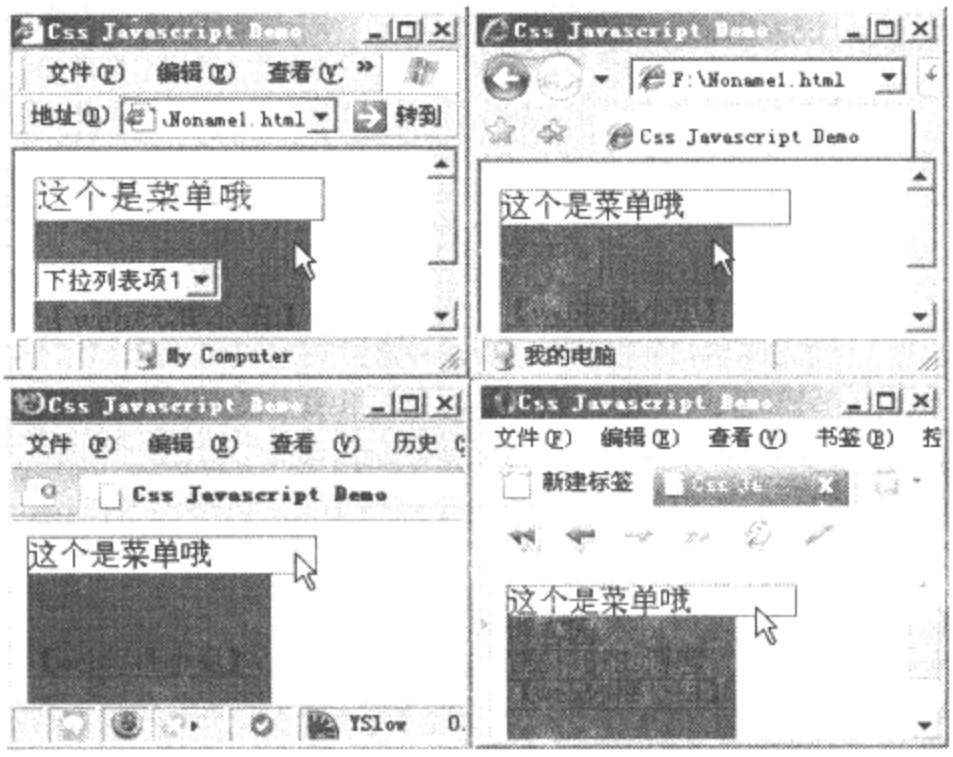
```
    </head>
    <body>
    <DIV id="DIVMenu" onmouseover="showMenu();" onmouseout=
"hideMenu();">这个是菜单哦
    </DIV>
    <DIV id="DIVUp" onmouseover="showMenu();"  onmouseout=
"hideMenu();">
    <a target="_blank" href="http://www.cnblogs.com" title="博客园">博
客园</a> <br/ />
    <a target="_blank" href="http://justinyoung.cnblogs.com/"
title="YES! B/S!">YES! B/S!博客
    </a> <br/ />
    <a target="_blank" href="http://space.cnblogs.com/w3c/" title="Web
标准小组">【Web 标准小组】
    </a> <br/ />
    </DIV>
    <br/ />
    <select id="ddlTest">
    <option>下拉列表项 1</option>
    <option>下拉列表项 2</option>
    <option>下拉列表项 3</option>
    <option>下拉列表项 4</option>
    </select>
    <!--============================-->
    <script type="text/javascript" language="javascript" >
      function showMenu(){
       document.getElementById("DIVUp").style.display="block";
      }
      function hideMenu(){
       document.getElementById("DIVUp").style.display="none";
      }
     </script>
</body>
</html>
```

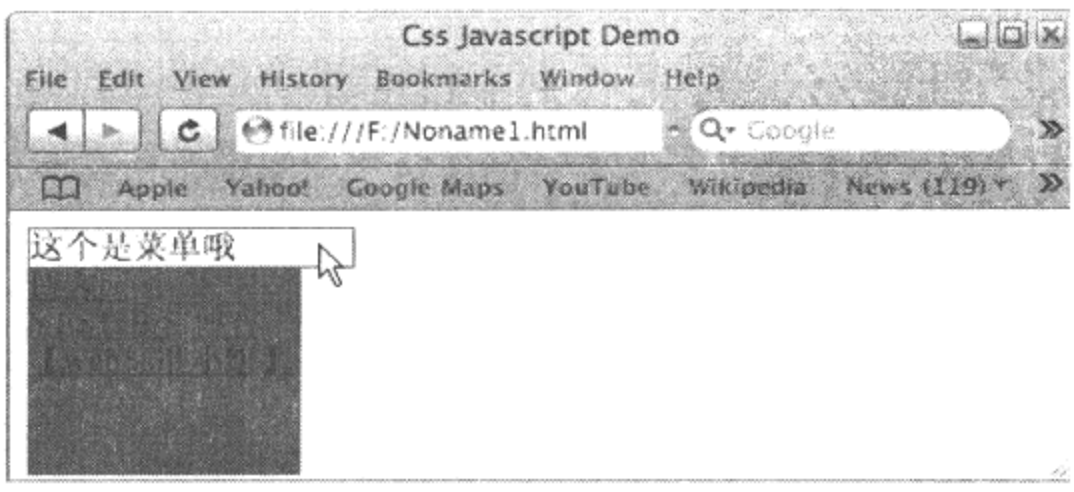
示例页面分别在 IE 6、IE 7、FireFox2、Opera(ver9.25)、Safari(windows3.04 版) 下的渲染效果如下。





当光标移到菜单上，出现下拉菜单的时候，IE 6 的这个诡异 bug 就出现了，如下图所示。注意观察 IE 6 中的列表框。





DIV 对 iframe 说："大哥，select 老是欺负我！"

这个 bug 我在《一个常被问道的问题：如何让层盖住下拉列表框？问题解决方案》[①]中介绍过，并且给出了供参考的解决方法，就是利用 iframe 来压 select，然后 DIV 放在 iframe 中。但是这就如同请董卓进京来消灭宦官一样，代价太大了些。但是在 IE 6 这样的乱世，也是一种没有办法的办法呀！

IE 7 说："社会解放了！DIV 可以翻身做主人了"

现如今 IE 7 已经修正了这个 bug，如果你网页的浏览者多数已经在使用 IE 7，并且你不在乎那些 IE 6 显示缺陷的话，完全可以不用再修正这个 bug。毕竟，为了这么一个视觉上的 bug，引入一个 iframe，是有点浪费了。

至此，到了 IE 7 时代，DIV 终于可以老泪纵横地说道："上去了！终于上去了！不再受到 select 的压迫，不用借助 iframe，凭着 DIV 我自己，凭着 IE 7 大环境，我终于上去了！"为了表达我激动的心情，我要把这幅对联献给 IE 7 浏览器和千千万万的 DIV 同胞。

上联：昔日恶狼列表框横行霸道

下联：如今 Web 标准化主持公正

横批：IE 7 好

[①] http://www.cnblogs.com/JustinYoung/archive/2007/07/18/821868.html

IE 7 标准之道

——5：置换元素与行距 bug

作者：阿一（杨正祎）[http://www.cnblogs.com/JustinYoung]

互联网上没有邮局！

看看下面这幅图，感觉是不是很亲切呢？有种“信纸”的感觉吧。其实这种感觉真的很不错，很怀旧、很小资的味道（突然想起以前我写的那些情书了^_^）。但是，你游遍互联网便会发现，很少有模拟信纸布局的网页。这到底是为什么呢？难道是大家都没有想到吗？难道是大家真的已经把“信”这种东西忘记了？还是因为互联网上没有邮局来寄出这些信件呢？

亲爱的朋友：

您好！

用网页来模拟一张信纸是不是很酷呀。为什么没有很少看到这种网页布局呢？

其实并不是没有人想到，恰恰相反，想到这种效果的人很多哦。但是在IE6时代有一个很奇怪的bug，叫做“置换元素与行距bug”。正是这个bug导致很多喜欢这种效果的朋友，不得不放弃这种布局方式。如果，你有兴趣了解这个bug，请阅读《IE7的web标准之道——6：(修正)置换元素与行距bug》这篇文章吧。

代我向你的朋友问好哦。就祝他们天天都有好心情吧！

你的朋友：杨正祎

2008-03-19 星期三

其实都不是，不是没有人想到，也不是没有人愿意用，而是被一种“虫”咬怕了。而这只“虫”就是我们今天要讲的—“置换元素与行距 bug”。

臭虫显身！

这里有两个测试用的示例页面。第一个是文章主体没有包含置换元素（replaced element），bug 不会作怪的例子，第二个是文章主体中包含了置换

元素（示例中为<img>）时，bug 在 IE 6 浏览器下会作怪的例子。

在举例之前，先让我们了解一下什么是置换元素（replaced element）：

“An element for which the CSS formatter knows only the intrinsic dimensions. In HTML, IMG, INPUT, TEXTAREA, SELECT, and OBJECT elements can be examples of replaced elements. For example, the content of the IMG element is often replaced by the image that the "src" attribute designates. CSS does not define how the intrinsic dimensions are found.”

资料来源于 W3C 网站对 replaced element 的定义。

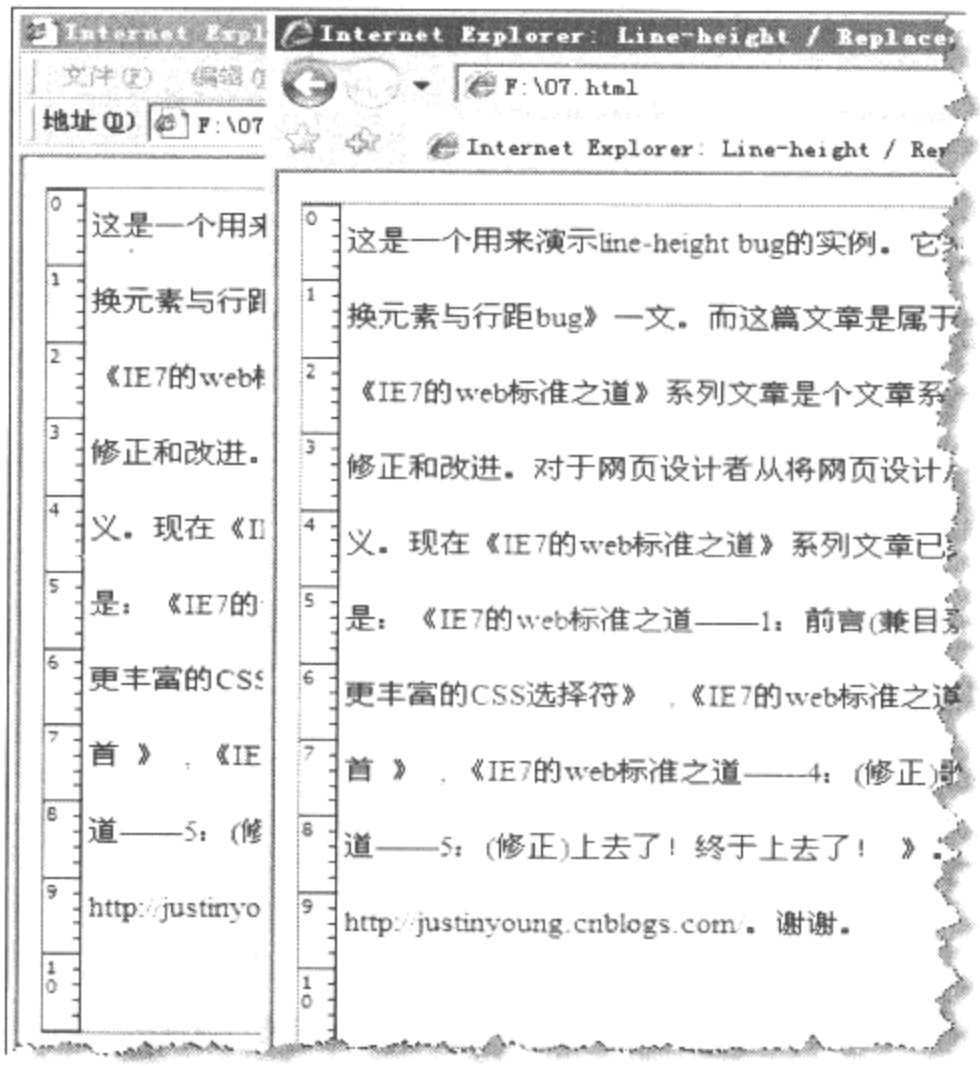
一来自己的英语水平太烂，二来这段英语比较简单，所以这里就不翻译了，大致意思大家应该都看得懂。

```
<!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" />
<meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,实例代
码" />
<meta name="Description" content="这是一个简单 YES!B/S!文章示例页面,
来自杨正祎的博客,
http://justinyoung.cnblogs.com/" />
<title>YES!B/S!文章示例页面</title>
<style type="text/CSS">
#lineheight_bug {
line-height: 39px;
font-size:14px;
background:url('http://images.cnblogs.com/cnblogs_com/justinyoun
g/2008_1q/rule.gif')
no-repeat;
padding:0;
padding-left:20px;
height:435px;
width:530px;
border:1px solid red;
}
</style>
</head>
<body>
<div id="lineheight_bug">
<p>这是一个用来演示 line-height bug 的实例。它来自《IE 7 的 Web 标准之道——6:
(修正)置换元素与行距 bug》一文。而这篇文章是属于《IE 7 的 Web 标准之道》系列文章的。
```

《IE 7的Web标准之道》系列文章是个文章系列，主要讲解了IE 7相对于IE 6各个方面的修正和改进。对于网页设计者从将网页设计从IE 6平稳的过渡到IE 7平台有一定的指导意义。现在《IE 7的Web标准之道》系列文章已经出道第六篇了。前面五篇的标题分别是：《IE 7的Web标准之道——1：前言(兼目录)》，《IE 7的Web标准之道——2：(改进)更丰富的CSS选择符》，《IE 7的Web标准之道——3：(修正)引起页面布局混乱的祸首》，《IE 7的Web标准之道——4：(修正)歌剧院魅影bug》以及《IE 7的Web标准之道——5：(修正)上去了！终于上去了！》。如果你有兴趣，可以访问http://justinyoung.cnblogs.com/。谢谢。</p>

</body>
</html>

让我们来看看示例页面1在IE 6和IE 7下的显示结果，可以看出其实没有什么区别的。

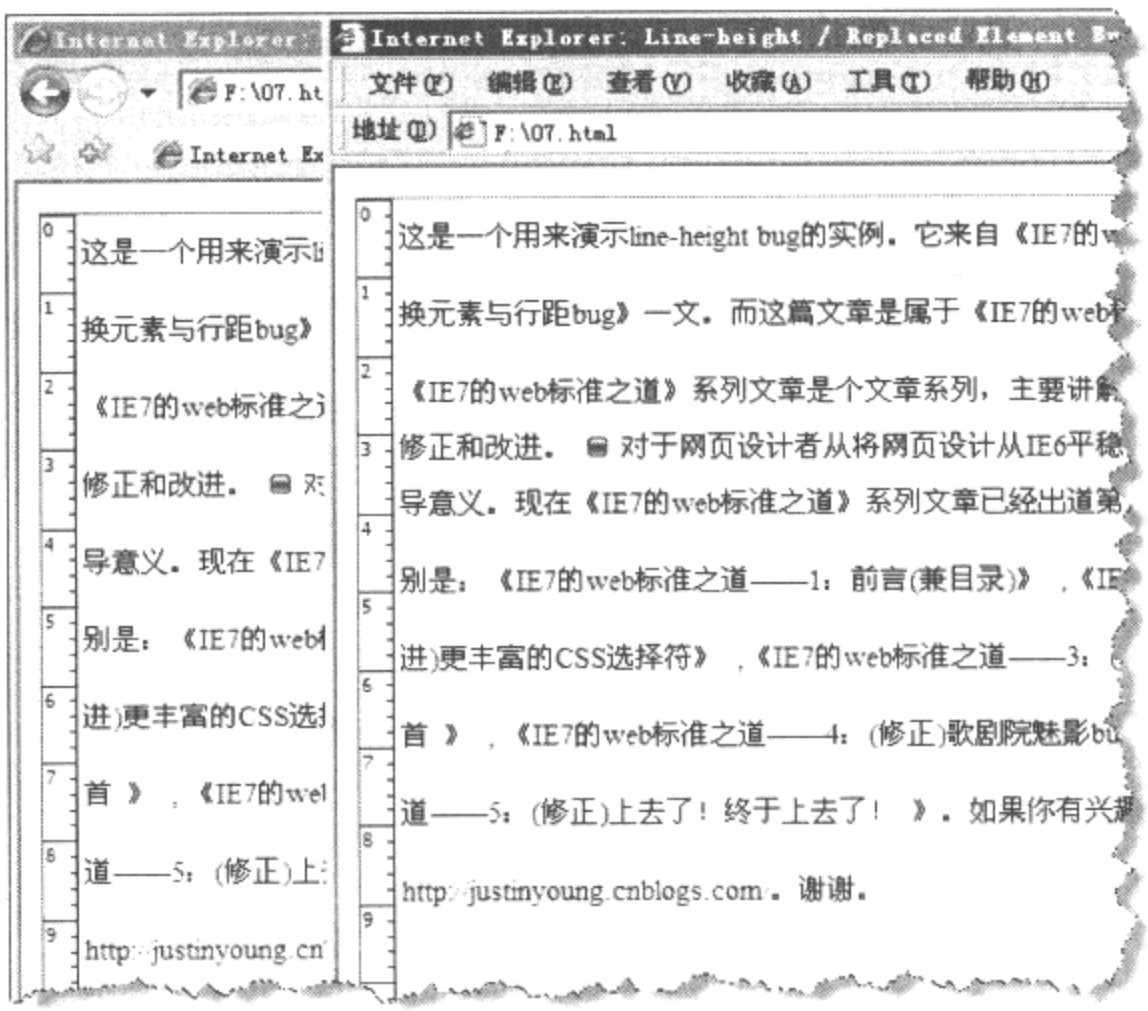


下面是测试用示例页面2源码。

```
<!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" />
<meta name="Keywords" content="YES!B/S!,web标准,杨正祎,博客园,实例代
```

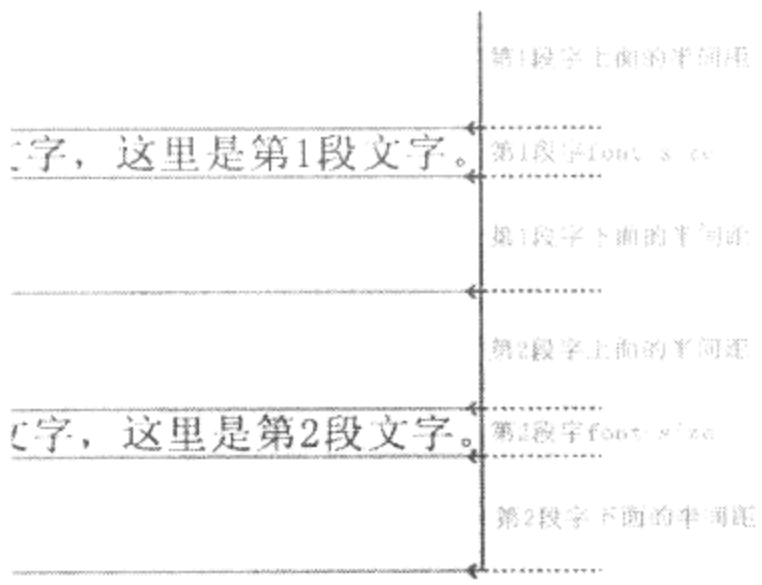
```
码" />
    <meta name="Description" content="这是一个简单 YES!B/S!文章示例页面,
来自杨正祎的博客,http://justinyoung.cnblogs.com/" />
    <title>YES!B/S!文章示例页面</title>
    <style type="text/css">
    #lineheight_bug {
    line-height: 39px;
    font-size:14px;
    background:url('http://images.cnblogs.com/cnblogs_com/justinyoun
g/2008_1q/rule.gif') no-repeat;
    padding:0;
    padding-left:20px;
    height:435px;
    width:530px;
    border:1px solid red;
    }
    </style>
    </head>
    <body>
    <div id="lineheight_bug">

    <p>这是一个用来演示 line-height bug 的实例。它来自《IE7 的 web 标准之道—6:
(修正)置换元素与行距 bug》一文。而这篇文章是属于《IE7 的 web 标准之道》系列文章的。
《IE7 的 web 标准之道》系列文章是个文章系列，主要讲解了 IE7 相对于 IE6 各个方面的修正
和改进。<img src="http://images.cnblogs.com/cnblogs_com/justinyoung/
common/wedgits_red.gif" alt="这就是置换元素的一种" /> 对于网页设计者从将网页
设计从 IE6 平稳的过渡到 IE7 平台有一定的指导意义。现在《IE7 的 web 标准之道》系列文
章已经出道第六篇了。前面五篇的标题分别是:《IE7 的 web 标准之道——1: 前言(兼目录)》,
《IE7 的 web 标准之道——2: (改进)更丰富的 CSS 选择符》，《IE7 的 web 标准之道——3:
(修正)引起页面布局混乱的祸首》，《IE7 的 web 标准之道——4: (修正)歌剧院魅影 bug 》
以及《IE7 的 web 标准之道——5: (修正)上去了! 终于上去了! 》。如果你有兴趣，可以访
问 http://justinyoung.cnblogs.com/。谢谢。</p>
    </div>
    </body>
    </html>
```

在这个测试页面中，我们在文章主体部分放入了一个小的图片（一个红色的方形，在第 4 行文字中间）。通过上面对置换元素（replaced element）的定义，我们知道<img>是一种置换元素，这样就满足了 bug 出现的条件。于是在 IE 6 中，我们便能够看到这个 bug 了（第 4 行文字的上下文间距出现了问题，从而导致整个布局发生混乱）。截图如下。



没有正解的“WHY？”

首先让我们提两个概念：line-height 和 font-size，行高和字体大小。line-height 减去 font-size 称为“间距（leading）”，间距的一半称为“半间距”。而“半间距”会被加在每行文字的上面和下面，于是行与行之间的空隙（有上一段下面的半间距和下一段上面的半间距相加而成）就出来了。下面的这幅图片可能对你理解有所帮助。



行距的细节分析图

之所以出现了这种 bug，是因为 IE 6 错误地将带有置换元素的那行文字

的上下半间距和相邻的上下两行的下上半间距合并到了一起。于是，带有置换元素的那行文字的上下行距就被减少了一半，所以页面出现了混乱。

虽然上面的文字很好地解释了 bug 产生的原因，但是仅仅是一种参考，并非官方解释。

残缺不完美的“HOW？”

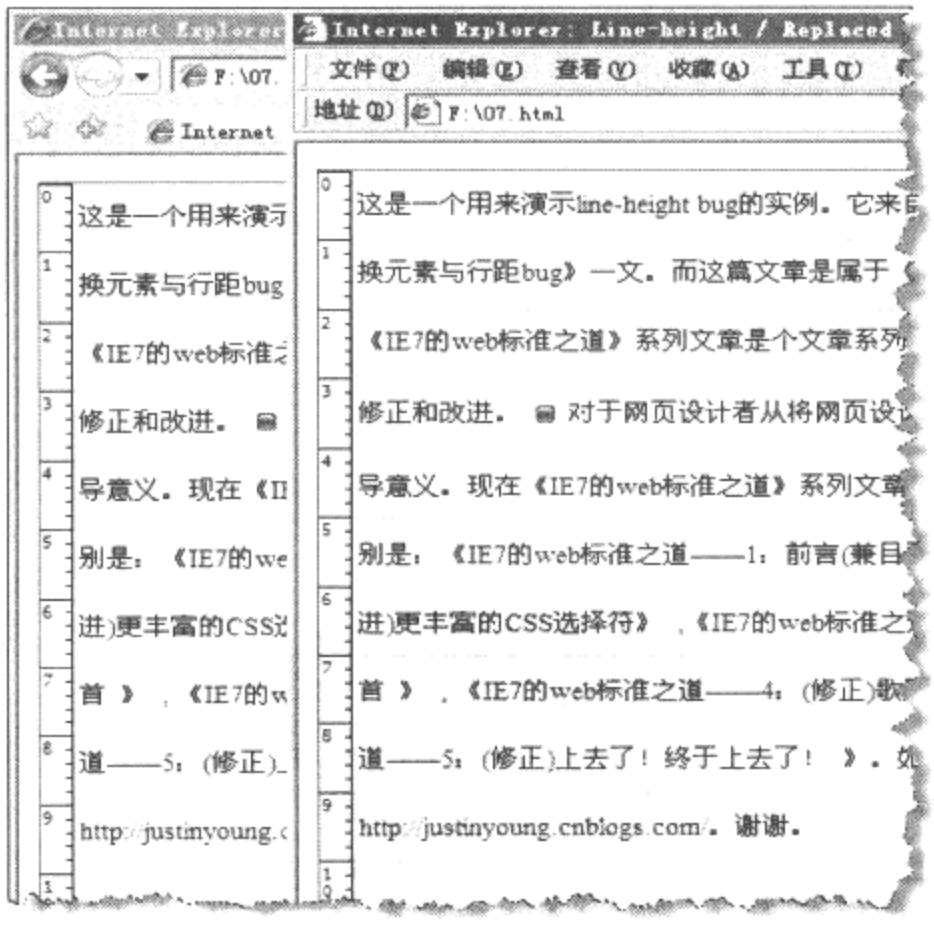
也许产生的原因并不重要，但是如何修复一定是重要的。毕竟现在 IE 6 的市场份额还是不能忽视的。

非常遗憾，虽然有解决方案，但是并非完美。方法就是对那些置换元素设置 margin-top 和 margin-bottom，以便把被“压缩”的行间距“撑开”。下面这个示例代码供你参考。如果你有更好的解决方案，请赐教，谢谢。

```
<!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" />
<meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,实例代
码" />
<meta name="Description" content="这是一个简单 YES!B/S!文章示例页面,
来自杨正祎的博客,
http://justinyoung.cnblogs.com/" />
<title>YES!B/S!文章示例页面</title>
<style type="text/CSS">
#lineheight_bug {
line-height: 39px;
font-size:14px;
background:url('http://images.cnblogs.com/cnblogs_com/justinyoun
g/2008_1q/rule.gif')
no-repeat;
padding:0;
padding-left:20px;
height:435px;
width:530px;
border:1px solid red;
}
/* 利用 IE 6 的 CSS hack 针对 IE 6 修正 bug，关于 CSS hack 的知识你可以访问:
http://www.cnblogs.com/JustinYoung/archive/2007/09/14/CSS-hack.html*/
#lineheight_bug img{
_margin:17px 0;
```

```
_vertical-align: middle;
}
</style>
</head>
<body>
<div id="lineheight_bug">
<p>这是一个用来演示 line-height bug 的实例。它来自《IE 7 的 Web 标准之道—6: (修正)置换元素与行距 bug》一文。而这篇文章是属于《IE 7 的 Web 标准之道》系列文章的。《IE 7 的 Web 标准之道》系列文章是个文章系列，主要讲解了 IE 7 相对于 IE 6 各个方面的修正和改进。 <img src="http://images.cnblogs.com/cnblogs_com/justinyoung/common/wedgits_red.gif" alt="这就是置换元素的一种" /> 对于网页设计者从将网页设计从 IE 6 平稳的过渡到 IE 7 平台有一定的指导意义。现在《IE 7 的 Web 标准之道》系列文章已经出道第六篇了。前面五篇的标题分别是：《IE 7 的 Web 标准之道——1: 前言(兼目录)》，《IE 7 的 Web 标准之道——2: (改进)更丰富的 CSS 选择符》，《IE 7 的 Web 标准之道——3: (修正)引起页面布局混乱的祸首》，《IE 7 的 Web 标准之道——4: (修正)歌剧院魅影 bug》以及《IE 7 的 Web 标准之道——5: (修正)上去了！终于上去了！》。如果你有兴趣，可以访问 http://justinyoung.cnblogs.com/。谢谢。</p>
</div>
</body>
</html>
```

下面是修复后的效果截图。



IE 7 标准之道

——6: float 双倍 margin bug

作者：阿一（杨正祎）[http://www.cnblogs.com/JustinYoung]

原来你这么出名呀！

这个又是IE 6中非常著名的bug了。在谭振林（Thin）翻译的《超越 CSS》一书第 6 页，有这么一句玩笑话：我在想象我 14 岁的儿子，如果现在让他开始学习网页设计，几年后当他再读到“double-marginfloat”或者“peekaboo”（俺注：躲猫猫 bug，也是很著名且好玩的一个 bug，我们以后会讲到哦）时估计会大笑一场……

之所以它会那么出名，是因为这个 bug 引发的条件极其简单，所以很多人都碰到过。只要对块状容器元素设置了 float 和与 float 相同方向的 margin 值就会出现，例如：对一个 DIV 设置了 float:left 和 margin-left:100px，那么在 IE 6 中，这个 bug 就会出现。下面是一个示例。

```
<!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" />
<meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,示例代
码" />
<meta name="Description" content="这是一个简单 YES!B/S!文章示例页面,
来自杨正祎的博客,
http://justinyoung.cnblogs.com/" />
<title>YES!B/S!文章示例页面</title>
<style type="text/CSS">
body{background:url('http://images.cnblogs.com/cnblogs_com/justi
nyoung/myPic/rule.gif')
no-repeat; margin:0;padding:0; }
.floatbox { float: left; width: 100px; height: 100px;
```

```
background-color:deeppink;
    margin-top: 20px; margin-left:100px; }
    </style>
    </head>
    <body>
    <DIV class="floatbox"></DIV>
    </body>
    </html>
```

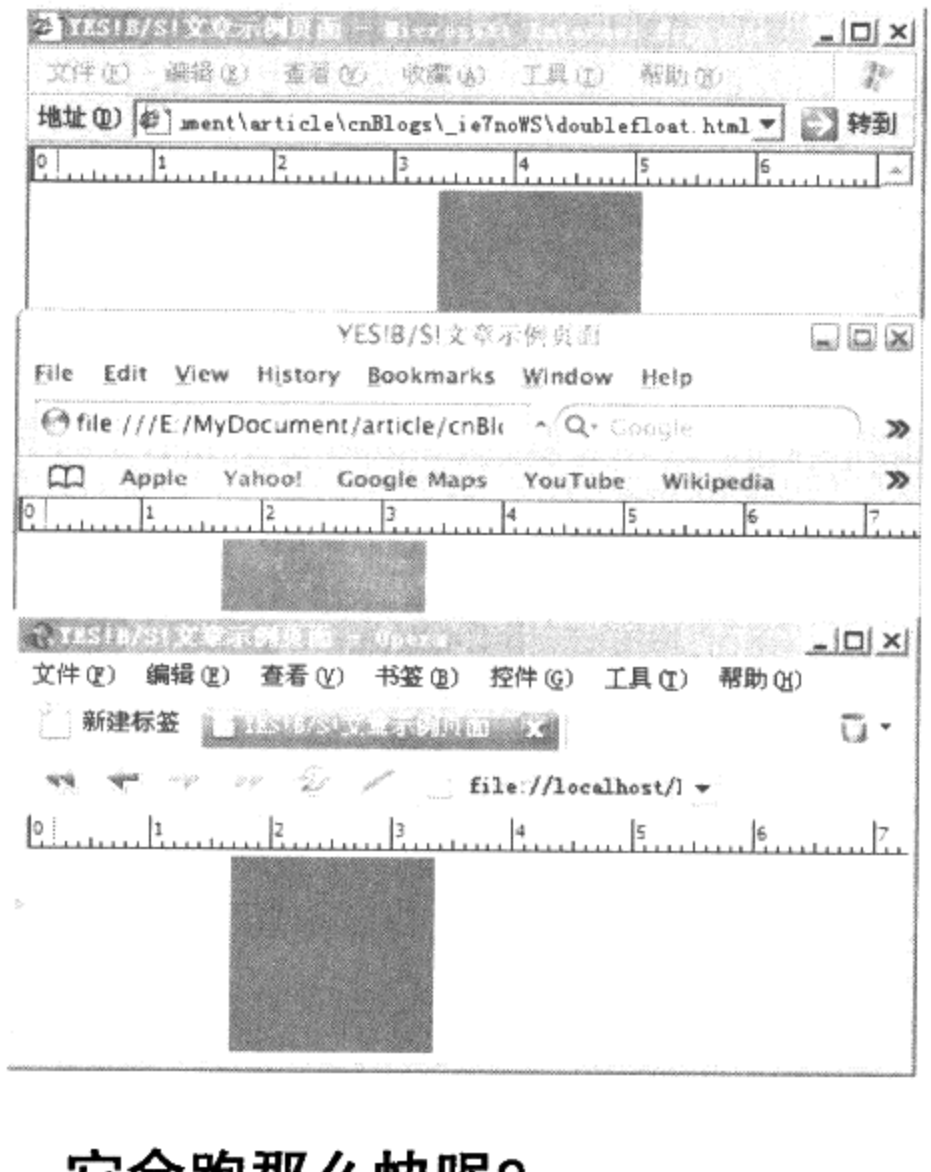
往哪跑!?

在上面的示例中，我们将一个 设置了 float:left 和 margin-left:100px;，我们的原意是将这个粉红色的 DIV 向左漂浮，并在左边留出 100 像素的空隙。但是结果会如何？让我们来看看这个示例分别在 IE 6、IE 7 和 FireFox2 中的效果截图。

可以明显看到，这个粉红色的方块在 IE 6 中，左边的空隙要比 IE 7 和 FireFox 中大得多，通过上面的刻度可以知道，整整大了一倍。也就是说 margin-left:100px 在 IE 6 中变成了 margin-left:200px;的效果。

这就是著名的“float 双倍 margin bug”了。这个 bug 对于那种背景很花哨，但是内容很少，所以索性使用 position 定位的网页来说（例如沪江 2008 春节专题页面），是很致命的。它会导致网页中 positon 定位的元素在

不同浏览器中显示在不同的位置。这对于那种想偷懒用 position 直接定位的设计者是个不小的打击。下面是示例页面在 Opera 和 Safari 浏览器中的效果截图。



为什么，它会跑那么快呢？

非常令人遗憾，也许除了当初的 IE 6 开发团队，没有人说得出来这是为什么。我们只能解释说："这是 IE 6 的一个 bug!"。虽然这种解释连我们自己都觉得很勉强，但是实在不知道怎么解释。也许是小弟才疏，如果哪位高手知道，还请不吝赐教，这里先谢过了。

这个可咋整呀？

现在我终于知道为什么我的普通话那么不标准了，原来是因为我老是在文章中用方言和土话囧 rz~ 。

言归正传，虽然我们不知道为什么会出现这个现象，但是好在我们可以很好地解决这个 bug。方法还不只一个，而是两个：一个是推荐的，另一个是不推荐的。当然，我们要从推荐的那个方法开始。

推荐的修正方法

非常简单，只要对产生 bug 的容器设置一个“display:inline;”样式就可以了。例如上面的例子，我们就可以将 class 为 floatbox 的 重新设置样式，如下：

```
.floatbox {
    float: left;
    width: 100px;
    height: 100px;
    background-color:deeppink;
    margin-top: 20px;
    margin-left:100px;
    display:inline; /*多设置这个样式即可消除 bug! */
}
```

也许有些朋友会对 display:inline 很不放心，心想如果我以前设置的是块状元素（display:block），会不会因此变成级联元素（display:inline）呢？会不会从而影响我的页面效果呢？其实大可不必，因为当元素 float 的时候，display 样式的值就会自动转换为“block”，无论 display 先前设置的属性是什么都会失去效果（当然 dsiplay:none 除外^_^）。让我们看看 W3C 对 float 样式的解释：

"This property specifies whether a box should float to the left, right, or not at all. It may be set for elements that generate boxes that are not absolutely positioned. The values of this property have the following meanings:

left

The element generates a block box that is floated to the left. Content flows on the right side of the box, starting at the top (subject to the 'clear' property). The 'display' is ignored, unless it has the value 'none'.

right

Same as 'left', but content flows on the left side of the box, starting at the top.

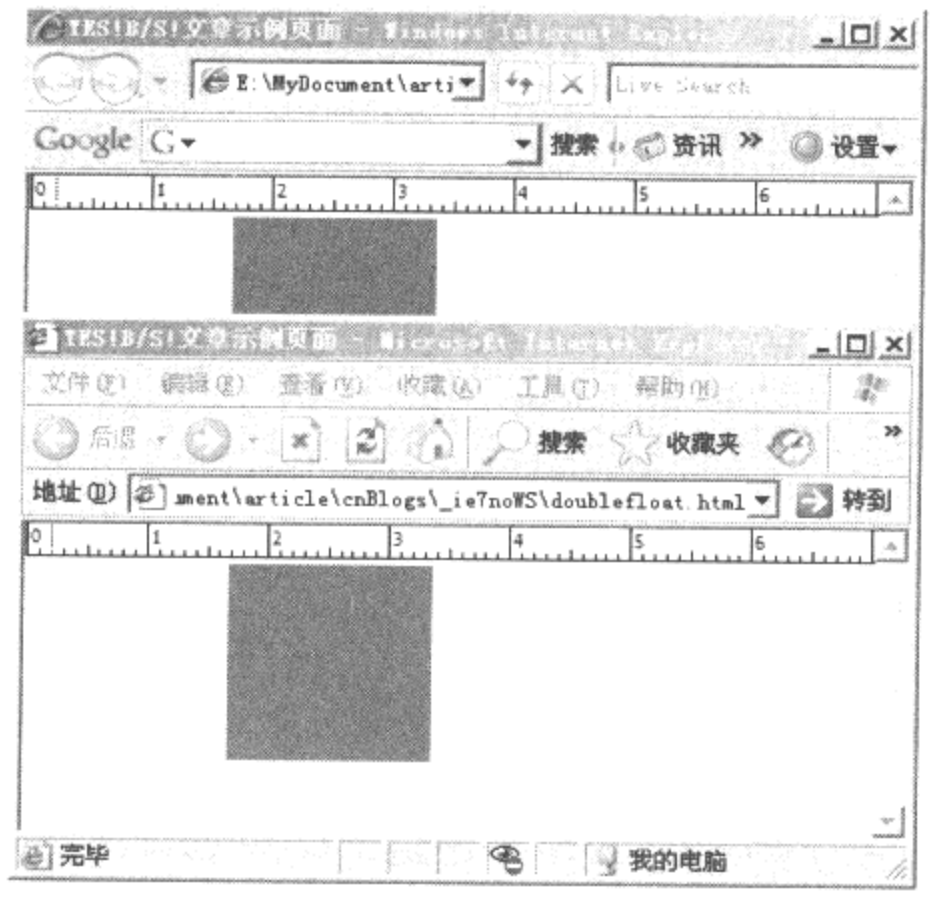
none

The box is not floated. "

也就是说，将浮动元素的 display 设置为“inline”值，是完全没有问题的。因为无论怎么设置，它都会失效。只要以后的浏览器还把 W3C 推荐的标准当作标准去执行，通过设置“display:inline;”方式来修正这个 bug 就不会产生副作用，这也是这个方式得到推荐的原因。

通过设置“display:inline;”就可以将造成这个 bug 的条件之一（块状元素）消灭掉。但是上面不是说过 float 会将 display 样式失效吗？那为什么通过设置“display:inline;”又能够修正 bug 呢？如同这个 bug 产生的原因一样—“只有鬼才知道!”，翻译成英文就是“only God knows!”。

下面是修正后的结果截图。



通过 CSS hack 修正 bug

只要提到 CSS hack，那么就一定是不推荐的方法。使用 CSS hack 就像在屁股上打补丁一样，不到迫不得已，谁穿打了补丁的裤子？不过有时候，CSS hack 的确能快速修补问题。既然只有 IE 6 会出这个问题，我们就采用一个只对 IE 6 起作用的 CSS hack。我们知道采用下划线"_"即可让样式只让 IE 6 认识。你不是会将 margin 加倍吗？那我就设置一半好了，你加倍后不就正好了吗？所以，采用 CSS hack，我们可以将原来的样式修改为：

```
.floatbox {
    float: left;
    width: 100px;
    height: 100px;
    background-color:deeppink;
    margin-top: 20px;
    margin-left:100px;
    _margin-left:50px;/*只对 IE 6 起作用的 CSS hack，对数值减半*/
}
```

知道为什么这个方法是不推荐的吧！一来是使用了 CSS hack，二来是因为我的数学不太好，所以这里只敢举个 50、100 的例子。万一出个“margin-left:187px;”那我不就要 win 键+R，然后 calc 了？（注：玩笑而已，主要原因是浏览器不支持小于 1px 的像素精确显示）

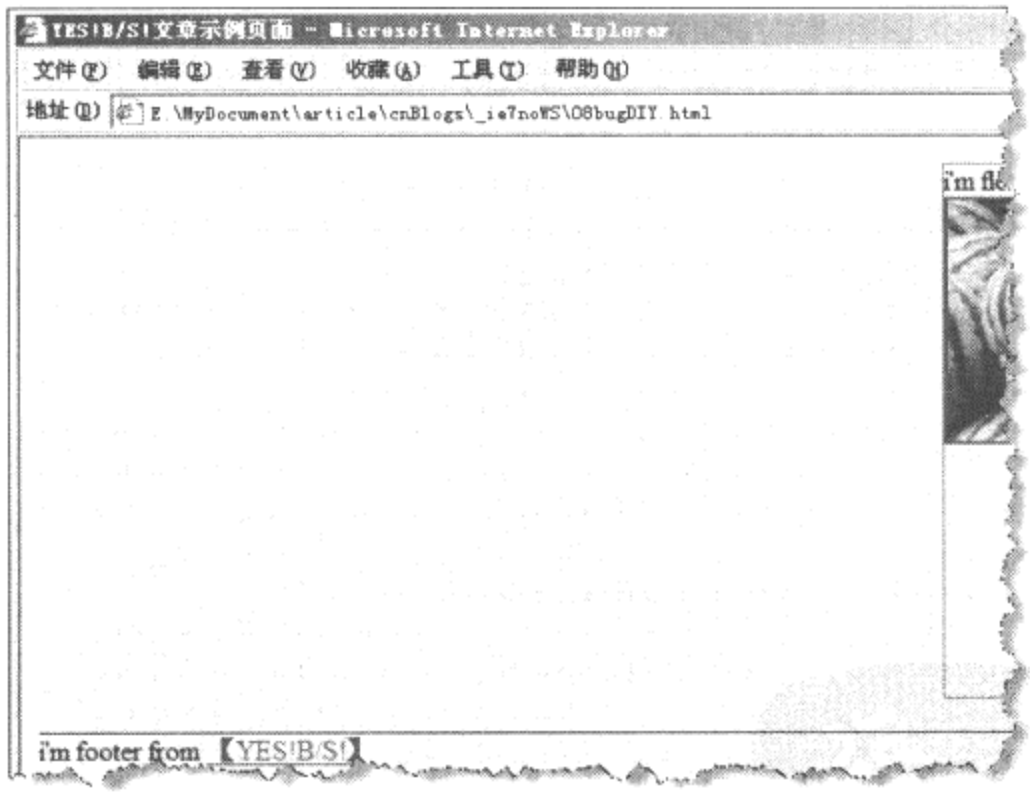
IE 7 标准之道

——7：躲猫猫 bug

作者：阿一（杨正祎）[http://www.cnblogs.com/JustinYoung]

躲猫猫，你来找我呀！

在上一篇文章中，我们曾经提到过这个 bug，它的著名来源于它所表现出来的诡异现象。虽然今天是 4 月 1 日，但是这些都是真实的。为了大家能够进一步记住这个诡异的 bug，这里我将改变以往的写作方式，首先将现象演示给大家看。下面是 bug 截图。



IE 6 躲猫猫 bug 效果截图

原来那儿有字呀！

如果只看上面的效果截图，你可能感觉有些纳闷。这不就是一个普通的没啥字的网页吗，没啥了不起的。但是当你看到这个网页的源码的时候，也

许就会感到惊讶了。

```
<!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" />
<meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,实例代码" />
<meta name="Description" content="这是一个简单 YES!B/S!文章示例页面,来自杨正祎的博客,
http://justinyoung.cnblogs.com/" />
<title>YES!B/S!文章示例页面</title>
<style type="text/css">
#holder{
background-color:pink;  /***引起 bug 的重要因素，一种解释就是：那些消失的文字躲到了背景之后***/
/*** width:100%; 对最大的容器设置宽度，相对宽度和绝对宽度都可以，即可修正 bug ***/
}
#holder a:hover{
background-color:deeppink; /***为了增强视觉效果而已，可以去掉。***/
}
#floater{
float:right;   /***引起 bug 的重要因素***/
width:135px;
height:310px; /***引起 bug 的重要因素，高度一定要大于那些文字的高度***/
border:1px solid green;
}
.clear{
clear:both;  /***引起 bug 的重要因素***/
}
#footer{ /***为了增强视觉效果而已，可以去掉。***/
height:50px;
border-top:1px solid blue;
}
</style>
</head>
<body>
<div id="holder">
<div id="floater">
i'm  floater!<br/ />
<a href="#" title=""><img alt="躲猫猫 bug"
```

```
src="http://images.cnblogs.com/cnblogs_com/
    justinyoung/2008_2q/ie7road81.jpg" /></a>
    </div><!--end: floater -->
    <div id="ghostHolder">
    这个示例是为了演示 IE 6 的“躲猫猫 bug”，如果你用 IE 6 浏览器浏览这个页面的时候，就会发现你看不到这些字了 囧 rz~ 。<br/ />
    IE 7 已经修正了这个 bug，所以你用 IE 7 浏览这个页面的时候，就能看到这些文字啦。<br/ />
    这个示例页面来自杨正祎的博客<a target="_blank"
href="http://justinyoung.cnblogs.com/" title="Web 标准推荐博客">【YES!B/S!】</a>，是<a target="_blank" href="http://www.cnblogs.com/JustinYoung/archive/2008/02/18/IE 7_wsRoad_foreword.html" title="《IE 7 的 Web 标准之道》">《IE 7 的 Web 标准之道》</a>的系列文章中的一篇，此系列文章还在连载中，下面是已经发表的文章列表——
    <ul style="list-style-type: disc">
    <li><a title="《IE 7 的 Web 标准之道——1: 前言》 "
href="http://www.cnblogs.com/JustinYoung/archive/2008/02/18/IE 7_
wsRoad_foreword.html" target="_blank">《IE 7 的 Web 标准之道——1: 前言（兼目录）》 </a> </li>
    <li><a title="IE 7 的 Web 标准之道——2: （改进）更丰富的 CSS 选择符"
href="http://www.cnblogs.com/JustinYoung/archive/2008/02/20/IE 7_
wsRoad_selector.html" target="_blank">《IE 7 的 Web 标准之道——2: （改进）更丰富的 CSS 选择符》</a> </li>
    <li><a title="IE 7 的 Web 标准之道——3: （修正）引起页面布局混乱的祸首"
href="http://www.cnblogs.com/JustinYoung/archive/2008/02/20/IE
7_wsRoad_overflow.html" target="_blank">《IE 7 的 Web 标准之道—3: （修正）引起页面布局混乱的祸首》</a> </li>
    <li><a title="IE 7 的 Web 标准之道——4: （修正）歌剧院魅影 bug"
href="http://www.cnblogs.com/JustinYoung/archive/2008/03/03/IE
7_wsRoad_dup_characters.html" target="_blank">
《IE 7 的 Web 标准之道——4: （修正）歌剧院魅影 bug》</a> </li>
    <li><a title="IE 7 的 Web 标准之道—5: （修正）上去了！终于上去了！"
href="http://www.cnblogs.com/JustinYoung/archive/2008/03/11/IE
7_wsRoad_DIV_select.html" target=
"_blank">《IE 7 的 Web 标准之道——5: （修正）上去了！终于上去了！》</a> </li>
    <li><a title="IE 7 的 Web 标准之道——6: (修正)置换元素与行距 bug"
href="http://www.cnblogs.com/JustinYoung/archive/2008/03/20/line-hei
ght-bug.html" target="_blank">《IE 7 的 Web 标准之道—6: （修正）置换元素与行距 bug》</a> </li>
    <li><a title="IE 7 的 Web 标准之道——7: （修正）float 双倍 margin bug "
href="http://www.cnblogs.com/JustinYoung/archive/2008/03/27/double-m
argin-float-bug.html" target="_blank">《IE 7 的 Web 标准之道—7: (修正)float 双倍 margin bug 》</a> </li>
```

```
    </ul>
    </div><!--end: ghostHolder -->
    <div class="clear"></DIV>
    <div id="footer">
    i'm footer from <a  target="_blank"
href="http://justinyoung.cnblogs.com/" title="Web 标准推荐博客">
【YES!B/S!】</a>
    </div>
    </div><!--end: holder -->
    </body>
    </html>
```

猫猫躲到哪里去了呢？

看过源码是否感到奇怪？那大片大片的文字都跑到那里去了？答案和往常一样令人遗憾—没有任何官方的说明和解释，只有江湖上比较得到认可的答案：被 id 为“#holder”那个 DIV 的背景颜色给盖住了。至于为什么会盖住，比较认可的说法是 IE 6 对 z-index 的解释有问题。无论原因如何，但是效果的确产生了，这也的确是 IE 6 的一个 bug。好在我们有办法可以解决它。

如何找到猫猫？

其实方法还蛮多的，如果仔细看网页源代码中关于 CSS 部分的注释，其实就已经可以找到一些答案。这里列出一些方案，任意一种都可消灭 bug。如果你有更好的办法，请到【博客园 Web 标准设计小组】赐教，小弟这里先谢过了。

- 明确指定最外面 DIV 容器（#holder）的宽度（相对宽度和绝对宽带都可以）
- 去掉最外面 DIV 容器（#holder）的背景颜色（或者背景图片）
- 缩小浮动 DIV 容器（#floater）的高度到一定程度
- 不浮动 DIV 容器（#floater）
- 不使用 技术

上面的方法只是理论上的，请根据实际需要采用不同的方法。这个 bug 在 FireFox、safari 和 opera 等浏览器下不会出现。

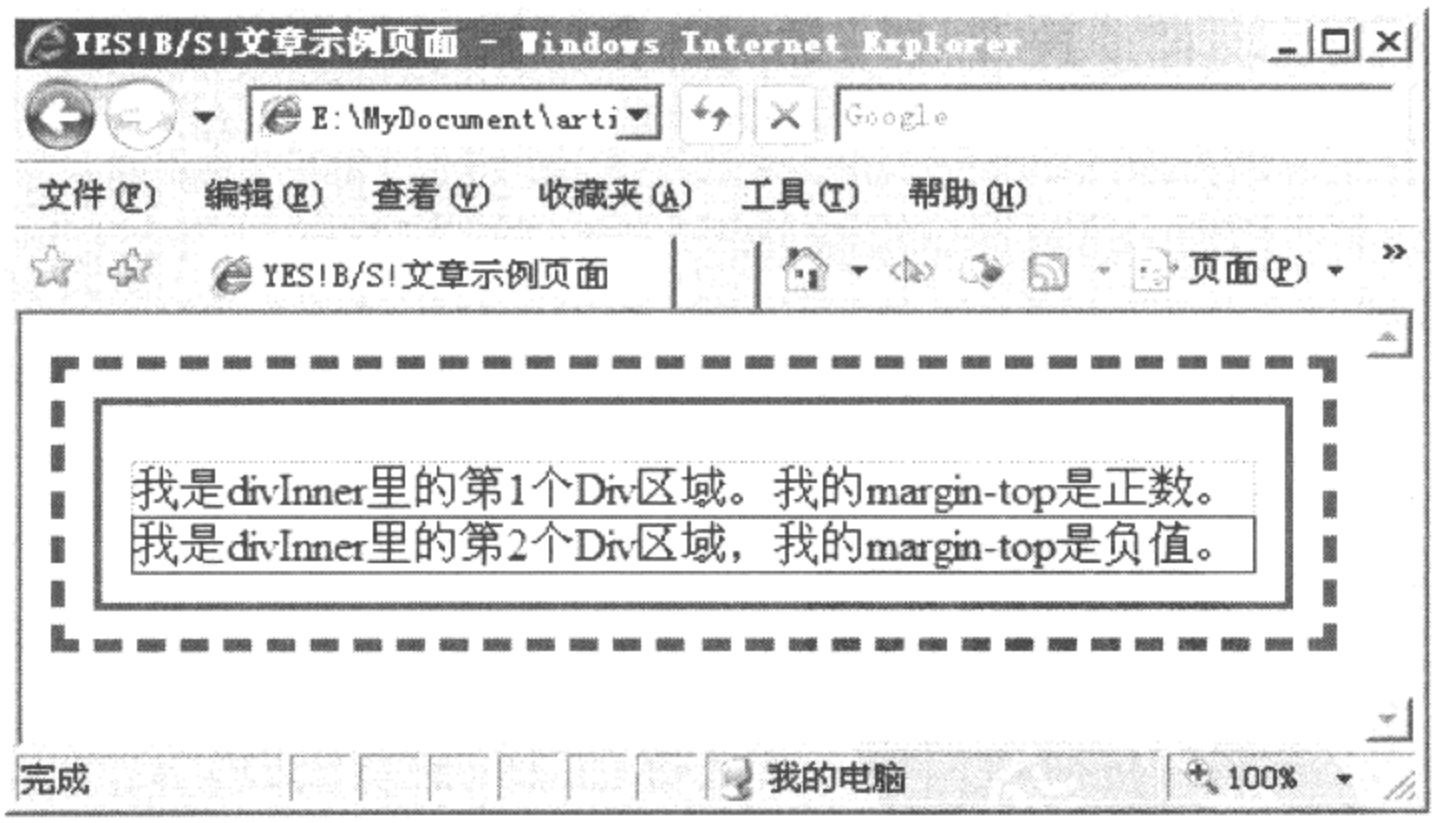
IE 7 标准之道

——8：疯了的边框线

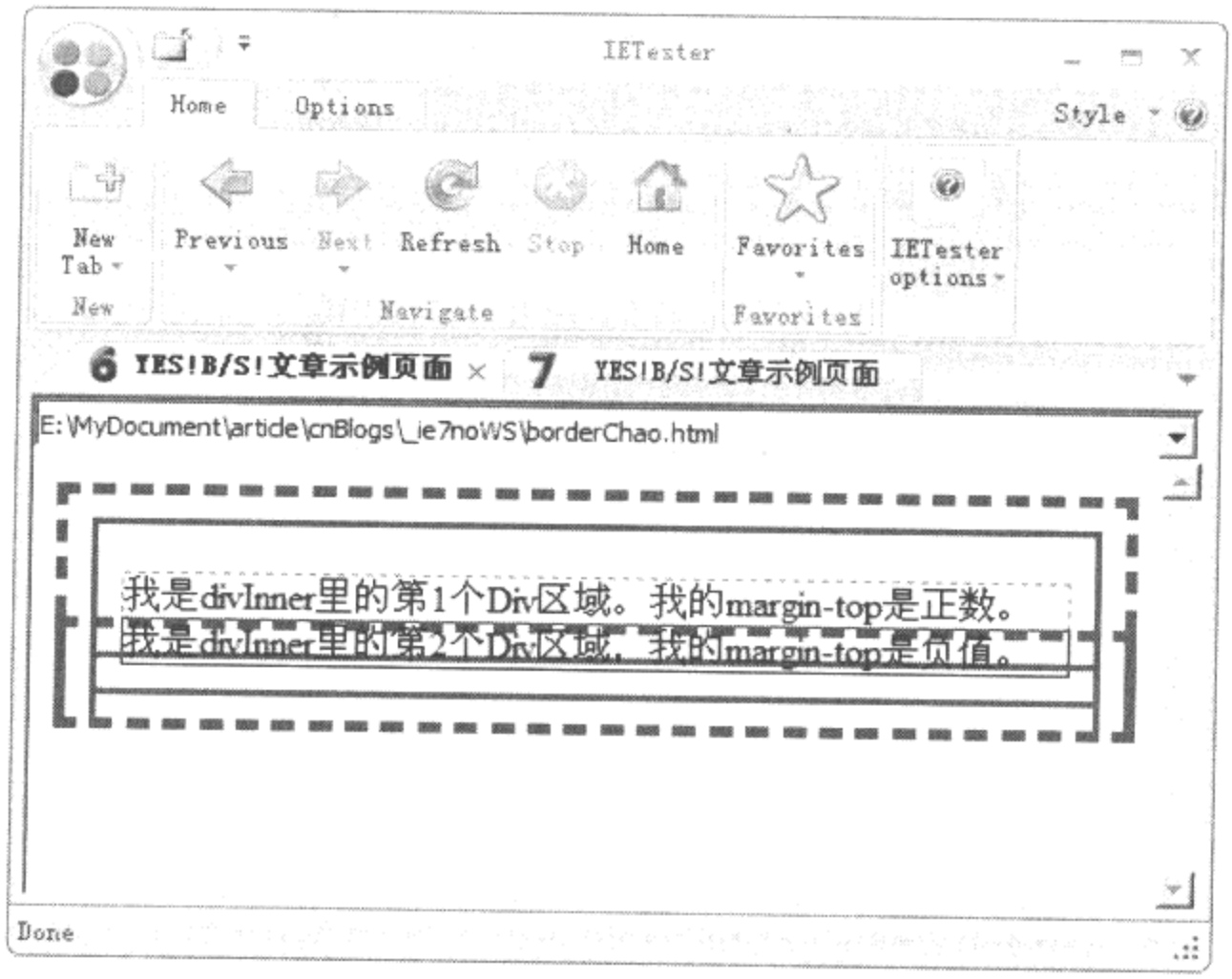
作者：阿一（杨正祎）[http://www.cnblogs.com/JustinYoung]

边框线疯了！

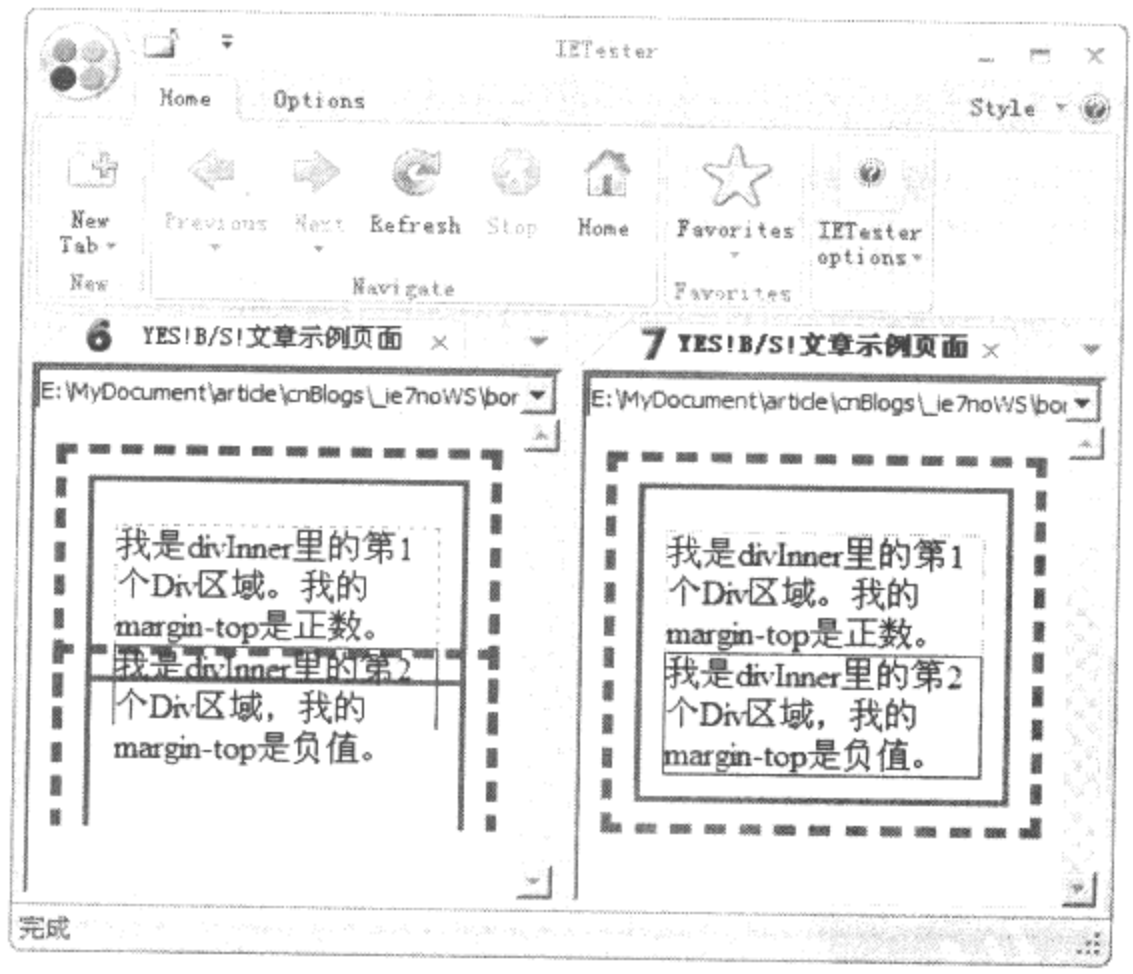
为了能给大家带来比较直观的印象，决定先把最终效果图给大家看看。结构其实很简单，就是一个大的 DIV（#DIVInner：3px 红色边框）里面套两个小的 DIV(#testDiv1 和#testDiv2)，然后再在最外面套了一个最大的 DIV（#DIVOuter：5px 绿色边框）。其实最外面的那个#DIVOuter 不是必须的，只是为了突出视觉效果而已。先看看在 IE 7 中的正常表现。



再来看看相同的页面在 IE 6 中的表现。看后也许你就会惊呼—哇！边框线疯了！



图中的工具叫做“IETester”，它可以方便地让 IE5.5 到 IE8 的各版本 IE 共存，从而方便测试网页的浏览器兼容性，如果你对它感兴趣可以阅读《IE 多版本共存的解决方案—IETester》[①]。下图是 IE 6 和 IE 7 显示效果对比图。



[①] http://www.cnblogs.com/justinyoung/archive/2008/05/04/IETester.html

如此简单的结构在 IE 6 的表现只能用一个字来表述，那就是—惨！

其实结构真的很简单

其实这个网页的结构真的很简单，下面是源码：

```
<!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" />
<meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,实例代
码" />
<meta name="Description" content="这是一个简单 YES!B/S!文章示例页面,
来自杨正祎的博客,
http://justinyoung.cnblogs.com/" />
<title>YES!B/S!文章示例页面</title>
<style type="text/CSS">
.DIVOuter {/*其实 DIVOuter 不是必须的，这里纯粹是为了提升视觉冲击力，你完全
可以将其去掉*/
border:5px dashed green;
padding:10px;
}
.DIVInner {
border:3px solid red;
padding:10px;
}
.testDiv1 {
border:1px dotted deeppink;
margin-top:10px;
}
.testDiv2 {
border:1px solid blue;
margin-top:-1px;/*这个是重点*/
}
</style>
</head>
<body>
<div class="DIVOuter">
<div class="DIVInner">
<div class="testDiv1">我是 DIVInner 里的第 1 个 Div 区域。我的 margin-top
是正数。</div>
```

```
    <div class="testDiv2">我是 DIVInner 里的第 2 个 Div 区域，我的 margin-top
是负值。</div>
    </div>
    </div>
    </body>
    </html>
```

而且里面的那个#DIVOuter 是无关紧要的东西，纯粹是为了提升视觉冲击而已，其实完全可以去掉。如果你想重现这个 bug，只需要以下条件就可以了。

- 两个块状元素(#testDiv1 和#testDiv2)
- 第二个块状元素（#testDiv2）有一个负值的 margin-top
- 然后把这两个块状元素放在一个大的块状元素中（#DIVInner）
- 当然他们都有可见的边框线（否则乱了你也看不到）

对不起，这里没有“为什么”

不要问，为什么会出现这么诡异的现象？边框线这些可爱的孩子为什么会疯掉？我只能说“要怨就怨那万恶的旧社会（IE 6）吧!”，在万恶的旧社会很多东西是没有“为什么”的，这些疯掉的边框线只是其中一部分受害者而已。

还好，这里有“怎么办”

不幸中的万幸，在这里你可以找到“怎么办”，这也是《IE 7 的 Web 标准之道》系列的风格。

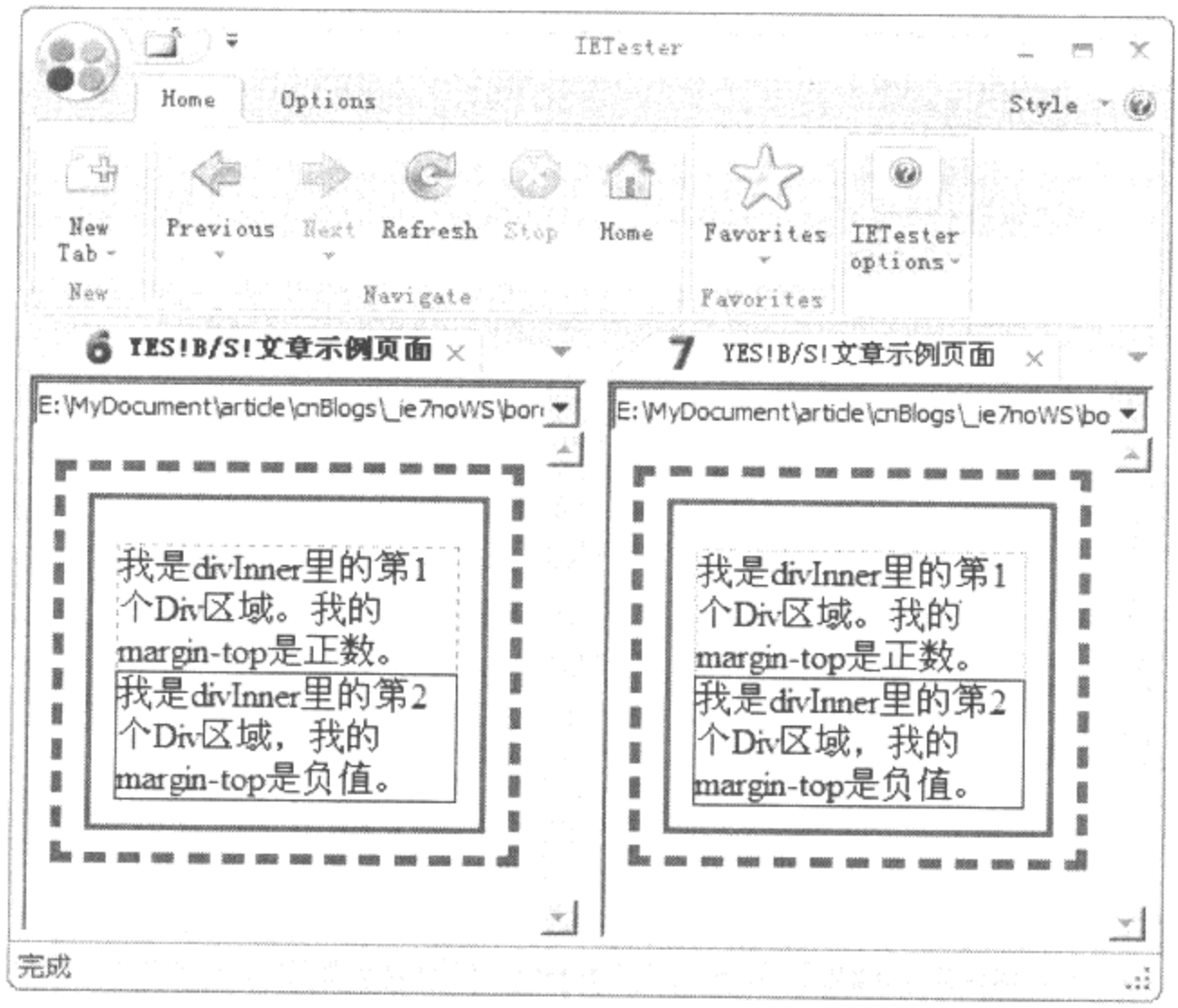
虽然目前的浏览器市场占有率中，IE 6 在走下坡路，IE 7 在上扬，但是仍然有接近 50%的网友在使用 IE 6，而且这种现象在短期内很难改变（虽然微软已经停止发售 windowsXP 了，但是那些老爷台式机还需要一段时间才能退出历史舞台）。所以，做网页现在仍然需要兼顾到 IE 6。

其实要解决这个 bug 也不难，只要从“*第二个块状元素（#testDiv2）有一个负值的 margin-top*”这条下手就可以了。如果你真的想让“#testDiv2”向上有一定距离的偏移，只要使用带负值的 top 样式。所以只要做下面的工作。

- 对#DIVInner 设置一个 position:relative;
- 对#testDiv2 也设置一个 position:relative;
- 对#testDiv2 设置一个负值的 top，例如 top:-1px;

对，很简单，用定位的方式来改变对象的位置，而不是使用负值的 margin。这样就可以很方便地解决这个 bug。让我们来看看修正后的代码和截图：

```
<!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" />
<meta name="Keywords" content="YES!B/S!,Web 标准,杨正祎,博客园,实例代
码" />
<meta name="Description" content="这是一个简单 YES!B/S!文章示例页面,
来自杨正祎的博客,
http://justinyoung.cnblogs.com/" />
<title>YES!B/S!文章示例页面</title>
<style type="text/CSS">
.DIVOuter {/*其实 DIVOuter 不是必须的，这里纯粹是为了提升视觉冲击力，你完全
可以将其去掉*/
border:5px dashed green;
padding:10px;
}
.DIVInner {
border:3px solid red;
padding:10px;
position:relative;
}
.testDiv1 {
border:1px dotted deeppink;
margin-top:10px;
}
.testDiv2 {
border:1px solid blue;
position:relative;
top:-1px;
}
</style>
</head>
<body>
<div class="DIVOuter">
<div class="DIVInner">
<div class="testDiv1">我是 DIVInner 里的第 1 个 Div 区域。我的 margin-top
是正数。</div>
<div class="testDiv2">我是 DIVInner 里的第 2 个 Div 区域，我的 margin-top
是负值。</div>
</div>
</div>
</body>
</html>
```


IETester
Home
Options
Style
New Tab
New
Previous
Next
Refresh
Stop
Home
Navigate
Favorites
Favorites
IETester options
6 YESIB/S!文章示例页面
7 YESIB/S!文章示例页面
E:\MyDocument\article\cnBlogs_ie7noWS\bor
E:\MyDocument\article\cnBlogs_ie7noWS\bo
我是divInner里的第1个Div区域。我的margin-top是正数。
我是divInner里的第2个Div区域，我的margin-top是负值。
我是divInner里的第1个Div区域。我的margin-top是正数。
我是divInner里的第2个Div区域，我的margin-top是负值。
完成

安全与优化

——安全和优化都是很重要的事情，花再多的努力都值得

安全是一个永恒的问题，没有安全，我们所做的一切都将没有意义；优化是老板们最喜欢的事情，可以以最小的代价获得最大的收益，当然，这也是我们开发者喜欢的事，因为它总能为我们带来成就感。我们这里的优化和平时说的 SEO 并不是一回事，SEO 只是优化中很小的一部分，甚至是最不重要的一部分。

安全永远是一个网站安身立命所必须考虑的事情，没有彻底的安全，但我们总是可以做得更好。在这一部分里，关于安全的只有“一篇”。之所以加引号是因为这一篇本来是一个系列很多篇文章，是作者 Kevin Zhou 为了方便大家阅读而重新编排成为了一篇。文章从 Request/Response 说起，一直讲到通用安全组件，是一个非常完整的系列，相信每一个人都可以从中学到很多东西。比如 WEB 是如何动作的，我们系统到底有多安全，权限如何抽象，如何转换我们的管控观念，怎么样脱离安全管控的苦海，等等，根本不需要犹豫，马上翻到《关于 Web 应用程序安全的思考》去看看吧。

在开篇《Web 开发中你注意这些问题了吗？（前台构架篇）》中，netcorner 告诉我们，什么样的前台构架可以有更好的表现，有多少问题需要注意；接下来，Lion 在《如何利用客户端缓存对网站进行优化》中告诉我们如何发挥客户端的能力，而不是让所有的问题都 server 扛。

在这些之后，依然还是阿一，他为我们带来了两篇《如何提高网页的效率》。在第一篇中阿一为我们带来了提高网页效率的 14 条法则，这并不是《High Performance Web Sites》一书中 14 条法则的简单抄录，而是一种全新的诠释。第二篇中，更是详细介绍了 YSlow 中各种数据的用法和含义，值得所有 WEB 开发者一看。

在这一部分的最后，有一篇关于 SEO 的文章，也是本书唯一的一篇。然而，因为网上太多 SEO 的内容，每一个 WEB 开发者都对这个领域非常熟悉，所以本书没有想要收录 SEO，之所以这个文章在最后入选，是因为我们看到太多的人去做 SEO，用太多的方式去做，以至于很多人迷失了，甚至很多新来者认为：优化=SEO，这是多么要命的一个公式！让我们回头来看，《SEO——我们是不是走错了路》给我们带来反思……

Web 开发中你注意这些问题了吗？

（前台构架篇）

作者：netcorner [http://www.cnblogs.com/netcorner]

Web 2.0 带给我们更好的用户体验和更炫更酷的效果，JavaScript、Flash、Silverlight 都在跃跃欲试。目前应用最多的还是 JavaScript，所以你会经常看到很多 Web 2.0 网站有 n 多的 JS 和 CSS。管理这些文件和如此多的代码，怎么能提升性能？以下就来讲讲我目前想到的一些问题。

JS 和 CSS 引用时如何做到让请求进行并发下载

我们通过 firebug，就会发现通过 link 和 script 标记在页面上的引用资源的每个请求都是以一个队列形式排队等候，一个资源下载完成后才会下载另外一个请求资源。它不像我们页面里面的图片（img 标记和样式中引用的图片，样式里面引用的图片必须等到 CSS 文件加载完毕后才能下载图片），可以并发下载资源文件。YSlow 曾经对 Web 站点优化提出，尽量把 CSS 放在 head 中（样式突然在其他资源下载完毕后才展现，那太有戏剧性了），但是有点搞不明白，为什么浏览对 CSS 加载也是一个队列，难道怕在样式中有重名部分的冲突？YSlow 还提出过把 JS 放在页面的尾部，那样的话整个页面下载 JS 资源差不多在 onload 完后。这点我深有体会。当你的 script 放在 head 中的时候，整个页面展现都得 script 一个个加载完毕再发生，这直接影响着 Web 性能，我想网站的速度比用户体验来得更重要吧，所以我们应该把 JS 放在尾部。那么是不是说直接放在尾部就好了呢？我想还有一点可以优化的，就是让其并发下载。那么如何解决这些资源的并发下载问题呢？

我的方法是通过动态追加 dom 的方法（appendChild，动态追加 link 和 script 节点标记到 head 下）。使用这种方式，我们会发现我们的队列突然变成了百米冲刺，一声哨下，都冲向终点了（当然每个并发请求数肯定还是有一定限制的）。不过在 IE 下 appendChild 这种方法在 window.onload 事件中无法引用资源的函数，所以在 IE 下我用 docoument.write 去输出（IE 下用 document.write 也是并发下载，而 firefox 是不行的）。所以通常在引用文件的时候使用 include

的方法，以下列出 include 代码。

```
var $include=function(file, callback){
    var isScript=file.indexOf(".CSS")==-1;
    var attLink,path,extName="";
    if(isScript){
        attLink="src";
        path=COREJSPATH;
        scripts = $tag("script");
        if(file.indexOf(".js")==-1) extName=".js";
    }else{
        scripts = $tag("link");
        attLink="href";
        path=CORECSSPATH;
    }
    if(file.indexOf("/")==-1){
        file=path+file+extName;
    }
    for (var i = 0; i < scripts.length; i++) {
        if(scripts[i][attLink]==file){
            return;
        }
    }
        var fileref;
        if (isScript){ //If object is a js file
            if(window.ie){
                fileref="script"+$time;
                document.write("<script src=\""+file+"\" id=\"
"+fileref+"\" type=\
    "text/javascript\"></script>");
            }else{
                fileref=$create('script');
                fileref.setAttribute("type","text/JavaScript");
                fileref.setAttribute("src", file);
                document.head.appendChild(fileref);
            }
        }
        else { //If object is a CSS file
            if(window.ie){
                fileref="script"+$time;
                document.write("<link rel=\"stylesheet\" rev=\"
stylesheet\" href=\"
    "+f+"\"  id=\""+fileref+"\"  type=\"text/CSS\" media=\"
```

```
screen\"/>");
            }else{
                fileref=$create("link");
                fileref.setAttribute("rel", "stylesheet");
                fileref.setAttribute("type", "text/CSS");
                fileref.setAttribute("href", file);
                document.head.appendChild(fileref);
            }
        }
        if(callback) {
            fileref=$(fileref);
            addEvent(fileref,'load',callback);
            fileref.onreadystatechange = function(){
                if(this.readyState=="loaded" || this.readyState==
"complete"){
                    callback();
                }
            };
        }
    };
```

注：

（1）CORECSSPATH 是当前 CSS 存放的相对路径、COREJSPATH 是当前 JS 存放的相对路径，用这个的原因主要是路径问题，这个稍后再讲。

（2）callback 是当文件加载完成后再调用方法。

（3）顺便说一下，我主要用 mootools 的一些函数。

我们的 JS 文件管理及引用太讲究了，侵入性太强了，一个不小心把顺序弄错或者依赖没引用那就惨了，如果有像 C#的 using 引用多好！

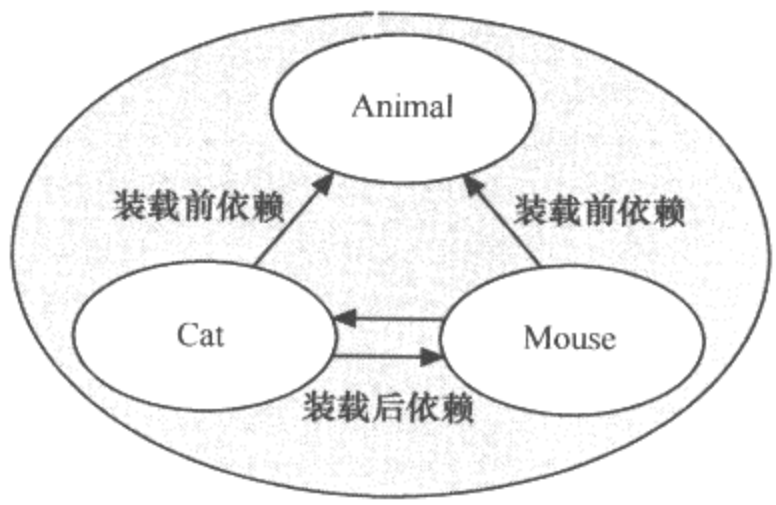
项目一天一天在扩大，此时发现已经有一大堆 JS 文件了，问题就来了——管理这些文件依赖和顺序很麻烦。JS 类库中可能存在着依赖关系，每个引用都得知道该类的依赖关系，然后再写入<script src="你的 JS 文件" type="text/javascript"></script>，而且说不定你的 JS 有装载前的依赖关系（也就是说引用一个 JS 前，需要把某个 JS 放在你引用的那个 JS 前面），不然运行

结果肯定是 error，所以我们急需管理这样一些类的解决方案。这方面看到 JSI 好像做得挺不错的（JSI 还没深入研究，不太懂其原理，看似好复杂）。

JS 的依赖关系示例（引自 JSI 的文档说明）

有 3 个类：动物（Animal），猫（Cat），老鼠（Mouse）；
他们有如下依赖关系：

- 猫、鼠装载前依赖动物类（装载这两个类时，需要创建动物实例作为其原型；该操作必须在装载时完成，为装载前依赖）。
- 猫鼠装载后，在使用过程中，相互依赖（猫鼠的行为中需要判别对方的行为，使用到了相互的引用，为相互装载后依赖）。



图例如下：

来说说我的想法（目前还未实现），最好存在一个依赖关系的配置文件，而且所有的依赖关系都存放在一个配置文件中并说明依赖关系（最好 vs 自动能生成关系），我看到 JSI 好像每个依赖得写一个__package___.JS 文件，那样是不是麻烦了？

关于路径问题

JS 和 CSS 的路径管理也是我们管理的一部分，哪天我们的页面换了个文件夹位置，就得改代码里的路径那实在是太糟糕了。所以我用上面的 $include 方法，只要核心 JS 文件路径确定，其他路径也随之确定（用$include 方法引进的资源文件）。如果想通过改变文件位置而不用改路径的话，那么通过核心文件在服务器输出核心 JS，问题也就迎刃而解了（图片的路径也是一样的道理，所以 w3c 好像要取消 img 标记，统一放在 CSS 中）。但是这样做会产生对某种开发语言的依赖，不知道园子里的朋友有没有更好的解决办法？

关于 JS 和 CSS 的缓存

我的解决办法是：通过核心部分的 JS 后加参数（随后 include 进来的 JS 后面都给其定义和核心 JS 一样的参数）。如果想控制所有页面，只能在服务器

端输出核心脚本（同路径问题的服务器端输出）。

如何减少多张图片的连接数

可以合并一张图片，用 CSS 样式定位。如：

```
.afirst,.alast,.anext,.apre,.nnext,.nlast,.nfirst,.npre{backgrou
nd:url(imgs/ gridbg.gif) 0px -116px;}
.anext{background-position:0px -153px;}
.alast{background-position:0px -221px;}
.apre{background-position:0px -185px;}
.nnext{background-position:0px -135px;}
.nlast{background-position:0px -200px;}
.nfirst{background-position:0px -97px;}
.npre{background-position:0px -169px;}
```

其实上面的图片是同一张，只是定位在不同位置（CSS 的路径引用图片的路径好像比 JS 好）。

合并一张图片还有一个好处是，比如按钮可能 onmouseover 时是一张图片，而 onmouserout 又是另一张图片时，用户光标经过就会感觉闪了一下，如果网速慢还要有下载那个图片的过程，这时呈现的是一片空白（在一定程度上提升用户体验）。

站点发布后 CSS 和 JS 的压缩

我的想法是这样的（没实现），vs 在生成网站的时候可以自动将 js 或者 CSS 自动压缩（当然可能还是会遇到点问题的，因为很有可能 JS 压缩后出现问题，在写 JS 时可能会少一个分号，那样必定会造成出错）。

按需装载和延迟装载问题

上面如果我用$include 引进文件还得在 window.onload 事件里面去执行，如果$include 下面的代码能够引用包含进来的文件的 function 的话，那就可以实现按需装载的过程了。可是通常这种解决办法是一种同步的阻塞式的装载过程，用户体验很差（电脑像死机一样了）。JSI 号称可以延迟装载这个过程，不知道这个过程是怎么实现的。

以上是我在 Web 开发和构架中的一些待解决问题，不知道圈子里的朋友还遇到哪些问题，希望能一起分享和探讨。

如何利用客户端缓存对网站进行优化?

作者: Lion [http://www.cnblogs.com/lion.net]

介绍

你的网站在并发访问很大并且无法承受压力的情况下,你会选择如何优化?

很多人首先会想从服务器缓存方面着手对程序进行优化,许多不同的服务器缓存方式都有他们自己的特点,像我曾经参与的一些项目中根据缓存的命中率不同使用过 Com+/Enterprise Libiary Caching/Windows 服务、静态文件等方式的服务器端缓存和 HTTP Compression 技术,但客户端缓存往往却被人们忽略了。即使服务器的缓存让你的页面访问起来非常地快,但它依然需要依赖浏览器下载并输出。而当你加入客户端缓存时,会给你带来非常多的好处。因为它可以对站点中访问最频繁的页进行缓存,充分地提高 Web 服务器的吞吐量(通常以每秒的请求数计算),提升应用程序性能和可伸缩性。

一个在线购物调查显示,大多数人愿意去商店排队,但在线购物时却不愿意等待。Websense 调查公司称多达 70%的上网者表示不愿意在页面读取上超过 10 秒钟。超过 70%的人会因为中途速度过慢而取消当前的订单。

基础知识

什么是“Last-Modified”?

在浏览器第一次请求某一个 URL 时,服务器端的返回状态会是 200,内容是你请求的资源,同时有一个 Last-Modified 的属性标记此文件在服务期端最后被修改的时间,格式类似这样:

```
Last-Modified: Fri, 12 May 2006 18:53:33 GMT
```

客户端第二次请求此 URL 时,根据 HTTP 协议的规定,浏览器会向服务器传送 If-Modified-Since 报头,询问该时间之后文件是否有被修改过:

```
If-Modified-Since: Fri, 12 May 2006 18:53:33 GMT
```

如果服务器端的资源没有变化，则自动返回 HTTP 304 （Not Changed.）状态码，内容为空，这样就节省了传输数据量。当服务器端代码发生改变或者重启服务器时，则重新发出资源，返回和第一次请求时类似，从而保证不向客户端重复发出资源，也保证当服务器有变化时，客户端能够得到最新的资源。

什么是“ETag”？

HTTP 协议规格说明定义 ETag 为“被请求变量的实体值”。另一种说法是 ETag 是一个可以与 Web 资源关联的记号（token）。典型的 Web 资源可以是一个 Web 页，但也可能是 JSON 或 XML 文档。服务器单独负责判断记号是什么及其含义，并在 HTTP 响应头中将其传送到客户端，以下是服务器端返回的格式：

```
ETag: "50b1c1d4f775c61:df3"
```

客户端的查询更新格式是这样的：

```
If-None-Match: W/"50b1c1d4f775c61:df3"
```

如果 ETag 没改变，则返回状态 304，然后不返回，这也和 Last-Modified 一样。本人测试 ETag 主要在断点下载时比较有用。

Last-Modified 和 ETag 如何帮助提高性能？

聪明的开发者会把 Last-Modified 和 ETag 请求的 HTTP 报头一起使用，这样可利用客户端（例如浏览器）的缓存。因为服务器首先产生 Last-Modified/ETag 标记，服务器可在稍后使用它来判断页面是否已经被修改。本质上，客户端通过将该记号传回服务器要求服务器验证其（客户端）缓存。

过程如下

1. 客户端请求一个页面（A）。
2. 服务器返回页面 A，并给 A 加上一个 Last-Modified/ETag。
3. 客户端展现该页面，并将页面连同 Last-Modified/ETag 一起缓存。
4. 客户再次请求页面 A，并将上次请求时服务器返回的 Last-Modified/ETag 一起传递给服务器。
5. 服务器检查该 Last-Modified 或 ETag，并判断出该页面自上次客户端请求之后还未被修改，直接返回响应 304 和一个空的响应体。

示例代码

下面的例子描述如何使用服务器端代码去操作客户端缓存：

```
//默认缓存的秒数
nt secondsTime = 100;
```

```
    //判断最后修改时间是否在要求的时间内
    //如果服务器端的文件没有被修改过，则返回状态是 304，内容为空，这样就节省了传输数据量。如果服务器端
      的文件被修改过，则返回和第一次请求时类似。
    if (request.Headers["If-Modified-Since"] != null && TimeSpan.FromTicks
(DateTime.Now.Ticks -
    DateTime.Parse(request.Headers["If-Modified-Since"]).Ticks).Seconds <
secondsTime)
    {
        //测试代码,在这里会发现,当浏览器返回 304 状态时,下面的日期并不会输出
        Response.Write(DateTime.Now);
        response.StatusCode = 304;
        response.Headers.Add("Content-Encoding", "gzip");
        response.StatusDescription = "Not Modified";
    }
    else
    {
        //输出当前时间
        Response.Write(DateTime.Now);
        //设置客户端缓存状态
        SetClientCaching(response, DateTime.Now);
    }
    // 设置客户端缓存状态
    private void SetClientCaching(HttpResponse response, DateTime
lastModified)
    {
        response.Cache.SetETag(lastModified.Ticks.ToString());
        response.Cache.SetLastModified(lastModified);
        //public 以指定响应能由客户端和共享（代理）缓存进行缓存。
        response.Cache.SetCacheability(HttpCacheability.Public);
        //是允许文档在被视为陈旧之前存在的最长绝对时间。
        response.Cache.SetMaxAge(new TimeSpan(7, 0, 0, 0));
        //将缓存过期从绝对时间设置为可调时间
        response.Cache.SetSlidingExpiration(true);
    }
```

如果你的缓存是基于文件的方式，如 XML 或 HTTP 中的.ashx 处理，也可以使用下面的基于文件方式的客户端缓存：

```
    // 基于文件方式设置客户端缓存
    private void SetFileCaching(HttpResponse response, string fileName)
    {
        response.AddFileDependency(fileName);
        //基于处理程序文件依赖项的时间戳设置 ETag HTTP 标头。
        response.Cache.SetETagFromFileDependencies();
```

```
        //基于处理程序文件依赖项的时间戳设置 Last-Modified HTTP 标头。
        response.Cache.SetLastModifiedFromFileDependencies();
        response.Cache.SetCacheability(HttpCacheability.Public);
        response.Cache.SetMaxAge(new TimeSpan(7, 0, 0, 0));
        response.Cache.SetSlidingExpiration(true);
    }
```

使用后的效果如下图所示：

Started	Time	Size	Method	Result	Type	URL
00:00:00.000	0.738	850	GET	200	text/html; char...	http://localhost:50313/a/Default.aspx
00:00:02.446	0.019	281	GET	304	text/html; char...	http://localhost:50313/a/Default.aspx
00:00:03.644	0.018	281	GET	304	text/html; char...	http://localhost:50313/a/Default.aspx

上图所使用的工具是在 IE 下运行的 HttpWatchPro，在 Firefox 下可以使用 FireBug+YSlow 进行测试。YSlow 是建立在 FireBug 基础上运行的一个小工具，它可以对你的网页进行分析为什么缓存，并给出评分和缓慢的原因。这个工具来自 Yahoo 的研发团队，所以规则也是 Yahoo 制定的。

结论

我们已经看了如何使用客户端缓存减少带宽和计算的方法。如前所述，如果能正确合理的利用各种不同的缓存，他们会给你带来很多好处。我希望本文已为你当下或将来基于 Web 的项目提供了精神食粮，并正确地在底层利用 Last-Modified 和 ETag 响应头去优化你的项目。

如何提高网页的效率（上篇）
——提高网页效率的 14 条准则

作者：阿一（杨正祎） [http://www.cnblogs.com/JustinYoung]

图：你的网页太臃肿了！

网站最基本的东西是什么？

网站最重要的东西是什么？——内容？SEO（搜索引擎优化）？UE（用户体验）？都不对！是速度！

内容再丰富的网站，如果慢到无法访问也是毫无意义的；SEO 做得再好的网站，如果搜索蜘蛛抓不到也是白搭；UE 设计得再人性化的网站，如果用户连看都看不到也是空谈。

所以网页的效率绝对是最值得关注的方面。如何才能提高一个网页的效率呢？Steve Souders（Steve Souders 的资料 http://www.oreillynet.com/pub/au/2951）提出的提高网页效率的 14 条准则，而这些准则也将是我们下篇中介绍到的 YSlow 工具的理论基础：

- Make Fewer HTTP Requests
- Use a Content Delivery Network
- Add an Expires Header
- Gzip Components
- Put CSS at the Top
- Move Scripts to the Bottom
- Avoid CSS Expressions
- Make JavaScript and CSS External
- Reduce DNS Lookups

- Minify JavaScript
- Avoid Redirects
- Remove Duplicate Scripts
- Configure ETags
- Make Ajax Cacheable

这里我们将逐一讲解这些准则，对其中与开发者密切相关的准则我将详细讲解。小弟个人技术实在有限，问题在所难免，还请高人指点。

第一条：Make Fewer HTTP Requests 尽可能减少 HTTP 的请求数

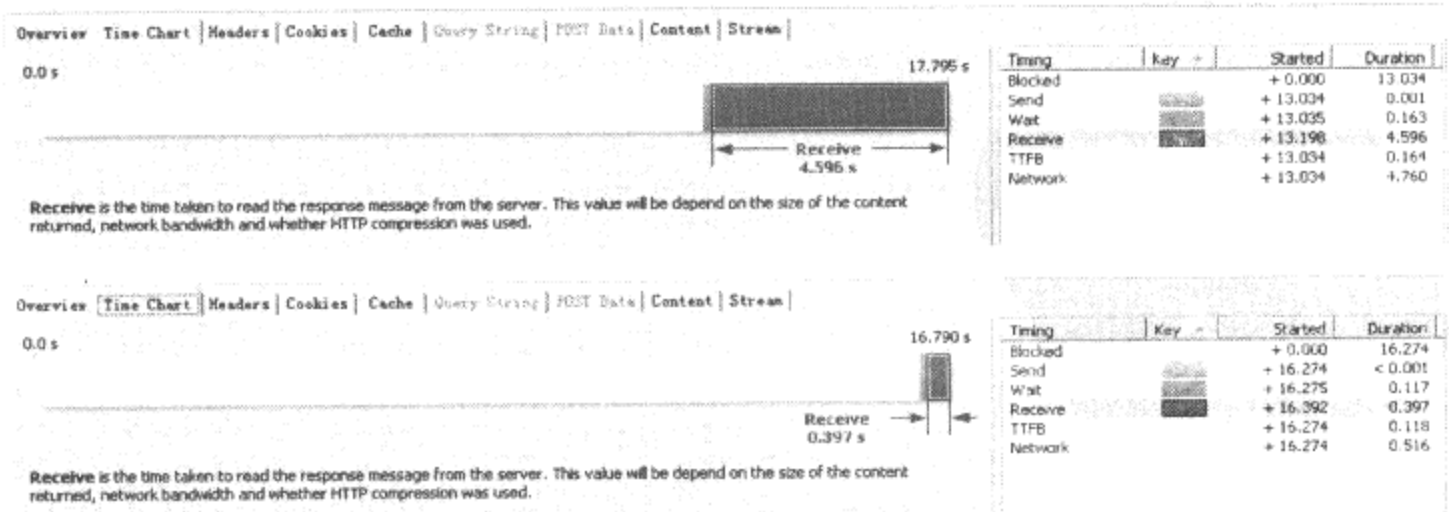
80%的用户响应时间都是浪费在前端。而这些时间主要又是因为下载图片、样式表、JavaScript 脚本、Flash 等文件造成的。减少这些资源文件的 Request 请求数是提高网页显示效率的重点。

这里好像有个矛盾，就是如果我减少了很多的图片、样式、脚本或者 Flash，那么网页岂不是光秃秃的，那多难看呢？其实这是一个误解。我们只是说尽量减少，并没有说完全不能使用。减少这些文件的 Request 请求数，当然也有一些技巧和建议的。

1. 用一个大图片代替多个小图片

这的确有点颠覆传统的思维了。以前我们一直以为多个小图片的下载速度之和会小于一个大图片的下载速度，但是现在利用 httpwatch 工具对多个页面进行分析的结果表明事实并不是这样。

第一张图是一个大小为 40528bytes 的 337×191px 的大图片的分析结果。第二张图是一个大小为 13883bytes 的 280×90px 的小图片的分析结果。



第一张大图片花费时间为：

Blocked：13.034s

Send：0.001s

Wait：0.163s

Receive：4.596s

TTFB：0.164s

NetWork：4.760s

功耗时：17.795s

真正用于传输大文件花费的时间为 Reveive 时间，即 4.596s，多数的时间是用来检索缓存和确定链接是否有效的 Blocked 时间，共花费 13.034s，占总时间的 73.2%。

第二张小图片花费时间为：

Blocked：16.274s

Send：小于 0.001s

Wait：0.117s

Receive：0.397s

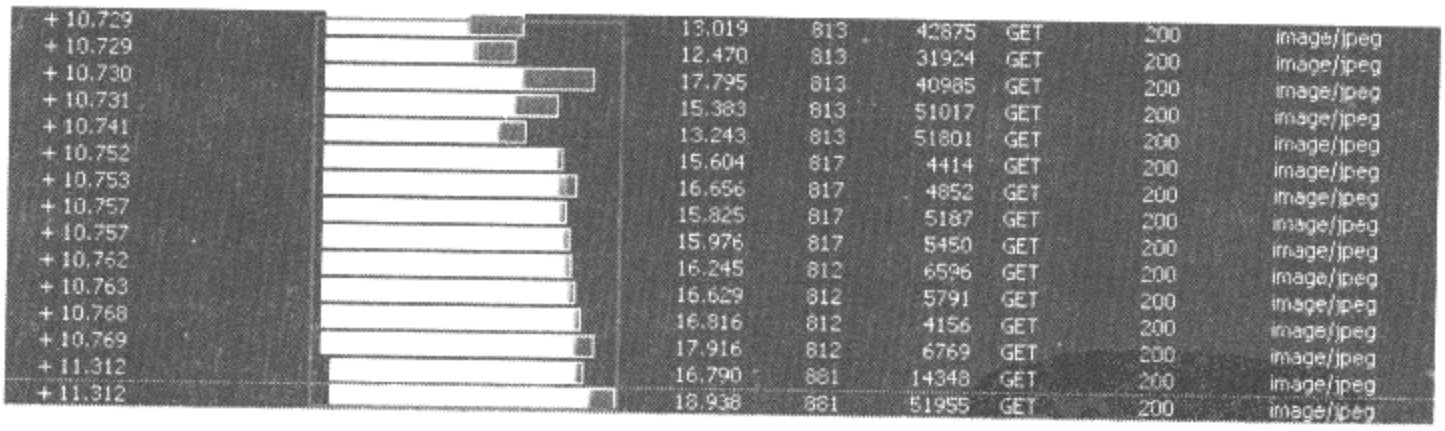
TTFB：0.118s

NetWork：0.516s

功耗时：16.790s

真正用于传输文件的花费时间是 Reveive 时间，即 0.397s，这的确要比刚才大文件的 4.596s 小很多。但是它的 Blocked 时间为 16.274s，占总时间的 97%。

如果这些数据还不够说服你的话，让我们看看下面这张图。这里列出了某个网页中所有图片中的花费时间示意图。当然，里面的图片有大有小，规格不一。



大约 80%以上的时间是用来检索缓存和确定链接是否有效的 Blocked 时间。

其中藏青色的为传输文件花费的 Reveive 时间，而前面白色的为检索缓存和确认链接是否有效的 Blocked 时间。铁一样的事实告诉我们：

- 大文件和小文件下载所需时间的确是不同的，差异的绝对值不大，而且下载所需时间占总耗费时间比例很小。
- 大约 80%以上的时间是用来检索缓存和确定链接是否有效的 Blocked 时间。无论文件大小，这个时间的花费大致是相同的，而且所占总耗

费时间的比例是极大的。

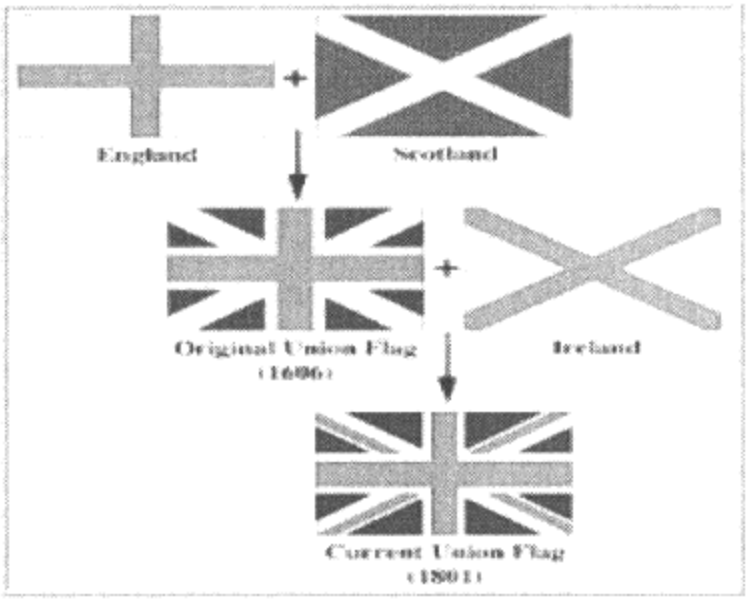
- 一个 100KB 的大图片总耗费时间绝对大于 4 个 25KB 的小图片的总耗费时间，而且主要差别就是 4 个小图片的 Blocked 时间绝对大于 1 个大图片的 Blocked 时间。

所以如果可能，还是使用大图片来替代过多的琐碎的小图片吧。这也是为什么翻转门的效率要高于图片替换实现的滑动门的原因。

但是请注意，也不能用太大的单张图片，因为那样会影响到用户体验。例如几兆的背景图片的使用绝对不是一个好主意。

2. 合并你的 CSS 文件

我以前犯了一个错误，你在看我《样式表的组织与规划》的系列文章中会知道。当时，我为了方便组织和规划样式表，将用于不同用途的样式表文件分离开来，形成不同的 CSS 文件。然后在页面中根据需要引用多个 CSS 文件。根据“尽可能减少 HTTP 的 Request 请求数”准则我们知道，那样的确是不合理的，因为那样会产生更多的 HTTP 的 Request 请求数。从而降低网页的效率。所以从提高网页效率的角度上而言，我们还是应该将所有的 CSS 写在同一个 CSS 文件中。但是问题又来了。那么怎么来很好地组织和规划样式表呢？这的确是个矛盾。我现在的做法是采用两套版本——编辑版和发布版。编辑版仍然使用多个 CSS 文件以便于规划和组织。而等到发布的时候，再将多个 CSS 文件合并到一个文件中去，从而达到减少 HTTPRequest 请求数的目的。



图：合并与融合

3. 合并你的 JavaScript 文件

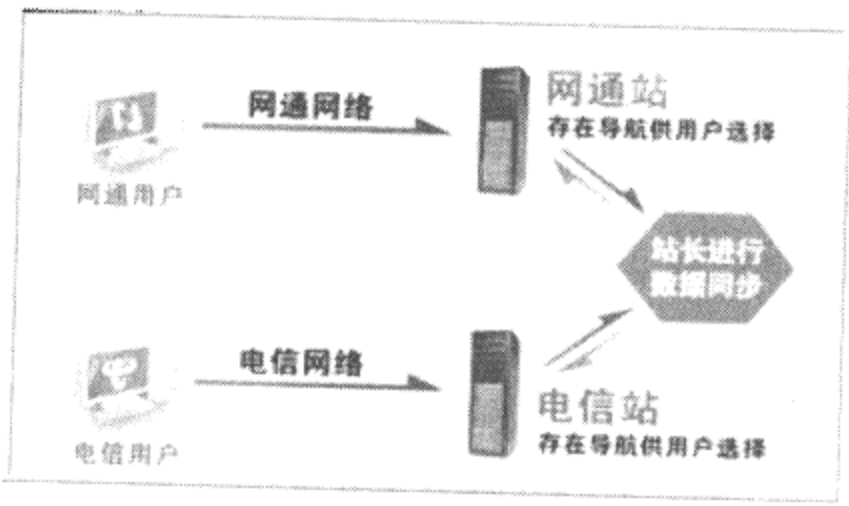
原因和处理方法同上，不再赘言。

第二条：Use a Content Delivery Network 使用 CDN

这个看上去好像很深奥的样子，但是只要结合中国的网络特色，这个便不难理解了。“北方服务器”、“南方服务器”、“电信服务器”、“网通服务器”……这些词听起来是那么熟悉和压抑。看个例子，当一个北京的电信用户试图从广东的网

通服务器上打开一个类似《壁纸合集》帖子的网页，你就能很深刻的理解。

鉴于这个不是我们开发人员力所能及的准则，所以这里也就不多言了。



这个图也算有点中国特色了。

第三条：Add an Expires Header 添加周期头

这个也并非开发人员来控制，而是网站服务器管理员的职责。所以如果作为开发人员的你不了解和明白也没有关系，把这个准则告诉公司的网站服务器管理员吧。

第四条：Gzip Components 启用 Gzip 压缩

这个大家应该比较熟悉。Gzip 的思想就是把文件先在服务器端进行压缩，然后再传输。这对于体积较大的纯文字型的文件有特效。鉴于这也并非开发人员，而是网站服务器管理员的工作范畴，就不详细讲解了。

第五条：Put CSS at the Top 把 CSS 样式放在页面的上方

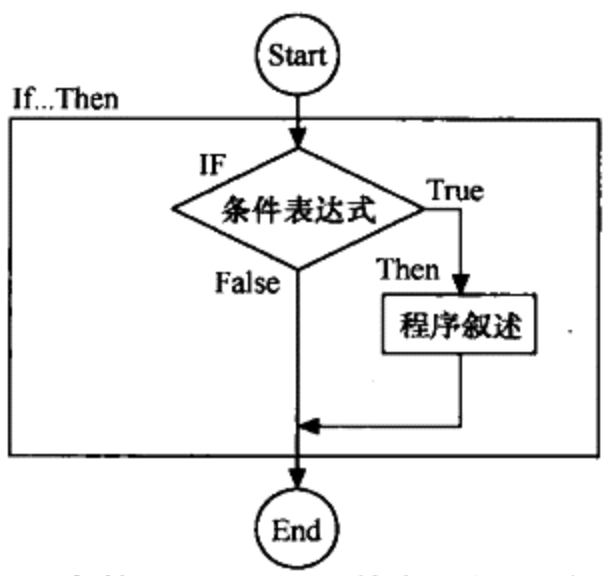
无论是 HTML、XHTML 还是 CSS 都是解释型的语言，而非编译型的。所以 CSS 到上方的话，在浏览器解析结构的时候，就已经可以对页面进行渲染。这样就不会页面结构光秃秃的先出来，然后 CSS 渲染，页面又突然华丽起来，这样太具有“戏剧性”的页面浏览体验了。

第六条：Move Scripts to the Bottom 将脚本放在底部

原因同第五条一样。只是脚本一般是用于用户交互的。所以如果页面还

没有出来，用户连页面都不知道什么样子，根本就谈不上交互。所以脚本和CSS正好相反，脚本应该放在页面的底部。

第七条：Avoid CSS Expressions 避免使用CSS中的Expressions



图：CSS中的Expressions其实也是一种if判断

首先有必要先说明一下CSS Expressions是什么。其实它就像其他语言中的if……else……语句，在CSS中可以进行简单的逻辑判断。举个简单的例子：

```
    <style>
    input{background-color:expression((this.readOnly && this.readOnl
y==true)?"#0000ff":"#ff0000")}
    </style>
    <INPUT TYPE="text" NAME="">
    <INPUT TYPE="text" NAME="" readonly="true">
```

这样CSS就可以根据一些情况分别使用不同的样式了。如果你对这个感兴趣可以到我的博客上阅读相关的文章——《CSS中的expression系列文章》[①]。但是CSS中Expressions的代价却是极高的。当你的页面需要根据判断来渲染效果的元素很多的时候，浏览器会长期处于假死状态，从而给用户带来极差的用户体验。

第八条：Make JavaScript and CSS External 将JavaScript和CSS独立成外部文件

这一条好像和第一条有点矛盾。的确，如果从HTTP的request请求数来

[①] http://www.cnblogs.com/justinyoung/articles/768899.html

讲的话，这样做的确是降低了效率。但是之所以这么做，是因为另外一个重要的考虑因素——缓存。因为外部的引用文件会被浏览器缓存，所以如果JavaScript 和 CSS 体积较大的时候，我们将它们独立成外部文件。这样只要用户浏览一次以后，这些体积较大的 JS 和 CSS 文件就能被缓存起来，从而极高地提高用户再次访问时的效率。

第九条：Reduce DNS Lookups 减少 DNS 查询

DNS 域名解析系统。大家都知道我们之所以能记住那么多的网址，是因为我们记住的都是单词，而非 http://202.153.125.45 这样的东西，而帮我们把那些单词和 202.153.125.45 这样的 IP 地址联系起来的就是 DNS。那这一条对我们到底有什么真正的指导意义呢？其实有两条：

1. 如果不是必须，请不要把网站放到两台服务器上。

2. 网页中的图片、CSS 文件、JS 文件、Flash 文件，等等，不要太多地分散在不同的网络空间中。这就是为什么那种只发一个网站中的壁纸图片的帖子，要比壁纸图片来源于不同网站的帖子显示要快得多的原因。

第十条：Minify JavaScript and CSS 减少 JavaScript 和 CSS 文件的体积

这点很好理解。在你的最终发布版本中把没有必要的空行、空格和注释全部去掉。显然手工去处理效率太低，好在网上到处都是用于压缩这些东西的工具。压缩 JavaScript 代码体积的工具随处可见，我便不再列举了，这里我只提供一个用于压缩 CSS 代码体积的在线工具网站——http://www.CSSdrive.com/index.php/main/CSScompressor。它提供了多种压缩方式，可以适应多种要求。

第十一条：Avoid Redirects 避免跳转

我只从网页开发人员的角度来解读此条。那么我们可以解读到什么呢？

1. “此域名已过期，5 秒钟以后，页面将跳转到 http://www.xxxxxx.com/index.html 页面”，这句话看起来的确很熟悉。但是为什么不直接链接到那个页面呢？

2. 一些链接地址请更明确的写出来。例如：将 http://justinyoung.cnblogs.com/ 写成 http://justinyoung.cnblogs.com（注意最后面一个“/”符号）。的确，这两

个网址都能访问到我的博客，但是事实上它们是有区别的。http://justinyoung.cnblogs.com 的结果是个 301 响应，它会被重新指向 http://justinyoung.cnblogs.com/ 。但是显然中间多浪费了一些时间。

第十二条 Remove Duplicate Scripts 移除重复的脚本

这个准则的道理很浅显，但是真正在工作中，很多人却因为“项目时间紧”、“太累了”、“初期没有规划好”这样的理由搪塞过去了。的确可以找很多理由不去处理这些多余重复的脚本代码，如果你的网站不需要更高的效率和后期维护的话。

也正是这点，我提醒大家：一些 JavaScript 框架、JavaScript 包一定要慎用，至少要问一下用了这个 JS kit 到底给我们多少方便，提高了多少工作效率，然后再与它因为多余的、重复的代码带来的负面效果比较一下。

第十三条：Configure ETags 配置实体标签

首先来讲讲什么是 ETag。ETag（Entity Tags）实体标签。这个 Tag 和你在网上经常看到的标签云那种 Tag 有点区别。这个 ETag 不是给用户用的，而是给浏览器缓存用的。ETag 是服务器告诉浏览器缓存，缓存中的内容是否已经发生变化的一种机制。通过 ETag，浏览器就可以知道现在的缓存中的内容是不是最新的，需不需要重新从服务器上下载。这和“Last-Modified”的概念有点类似。很遗憾作为网页开发人员对此无能为力。他依然是网站服务器人员的工作范畴。

第十四条：Make Ajax Cacheable 上面的准则也适用 Ajax

现在的 Ajax 好像有点被神话了，好像网页只要 Ajax 了，那么就不存在效率问题了。其实这是一种误解。拙劣的使用 Ajax 不会让你的网页效率更高，反而会降低你的网页效率。Ajax 的确是个好东西，但是请不要过分的神话它。使用 Ajax 的时候也要考虑上面的那些准则。

图：Ajax 的使用要恰当

后记

当然，上面的这些也只是供你参考的理论上的准则。具体的情况还是要具体的去对待。理论和准则只是用来指导现实工作的，却是万万不可死记硬套。

如何提高网页的效率（下篇）

——Use YSlow to know why your Web Slow

作者：阿一（杨正祎） [http://www.cnblogs.com/JustinYoung]

图：你的网页太臃肿了！

虽然我们在《如何提高网页的效率（上篇）——提高网页效率的 14 条准则》提到了如何提高网页效率的 14 条准则，但是如何知道我们现在的网页的效率到底如何？到底处于怎样一个级别？又有哪些方面做得不够好，需要改进呢？也许你会说，问一下用户不就知道了吗？但是相比感性因素比例太大的用户感受而言，理性的工具和数据更具有说服力。本篇就将向你介绍一款评测网页效率的工具——YSlow（why slow，这个名字起的太好了）。

yslow

YSlow 是由 Yahoo 开发者团队发布的一款基于 Firebug 的插件，而 Firebug 又是一款基于 FireFox 的插件。所以说 YSlow 是一款基于 FireFox 插件的插件。虽然有点绕，但是最终说明的问题是：

- 很遗憾，微软的 IE 系列浏览器不能使用 YSlow。
- YSlow 只能使用在 FireFox 浏览器上。
- 如果要想使用 YSlow，必须先安装 FireFox。
- 如果要想使用 YSlow，就要安装 FireFox 上的 Firebug 插件。

这看上去好像有点令人沮丧，但是事实上它并不像想象中的那么麻烦，只要按照下面的步骤就能很快地使用 Yslow。

1. 到 http://www.mozilla.net.cn/firefox/ 下载最新版的 FireFox，并安装它。当然如果你已经安装了 FireFox 可以跳过此步。

2. 到 https://addons.mozilla.org/en-US/firefox/addon/1843/ 下载最新版的 Firebug，并安装它。当然如果你已经安装了 Firebug 可以跳过此步。

3. 到 https://addons.mozilla.org/en-US/firefox/addon/5369/ 下载最新版的 YSlow，并安装它。当然如果你已经安装了 YSlow 可以跳过此步。

这时候打开 FireFox，你将在【工具】菜单中看到【firebug】(见图 2)。打开 firebug，然后在 firebug 中点击 YSlow 菜单，便进入 YSlow 的主界面(见图 3)。

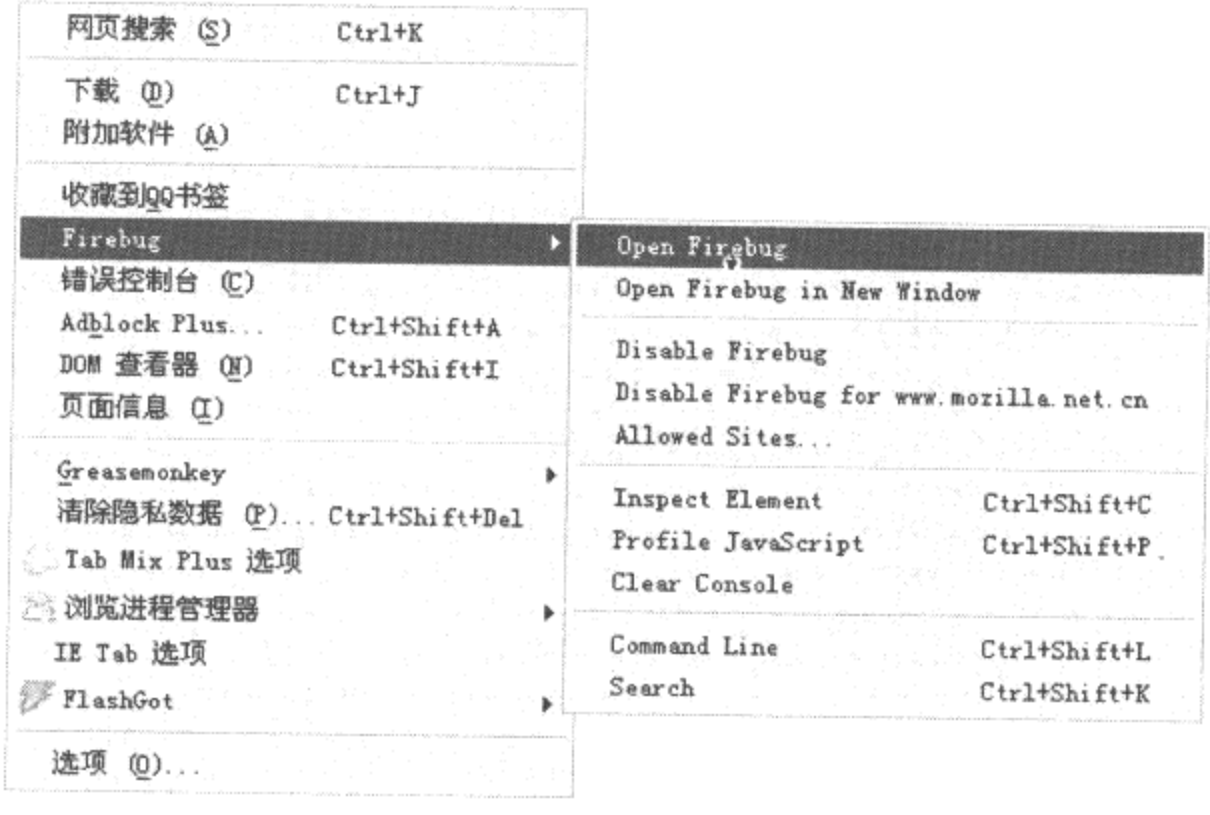


图 2：在菜单中先打开 Firebug 插件

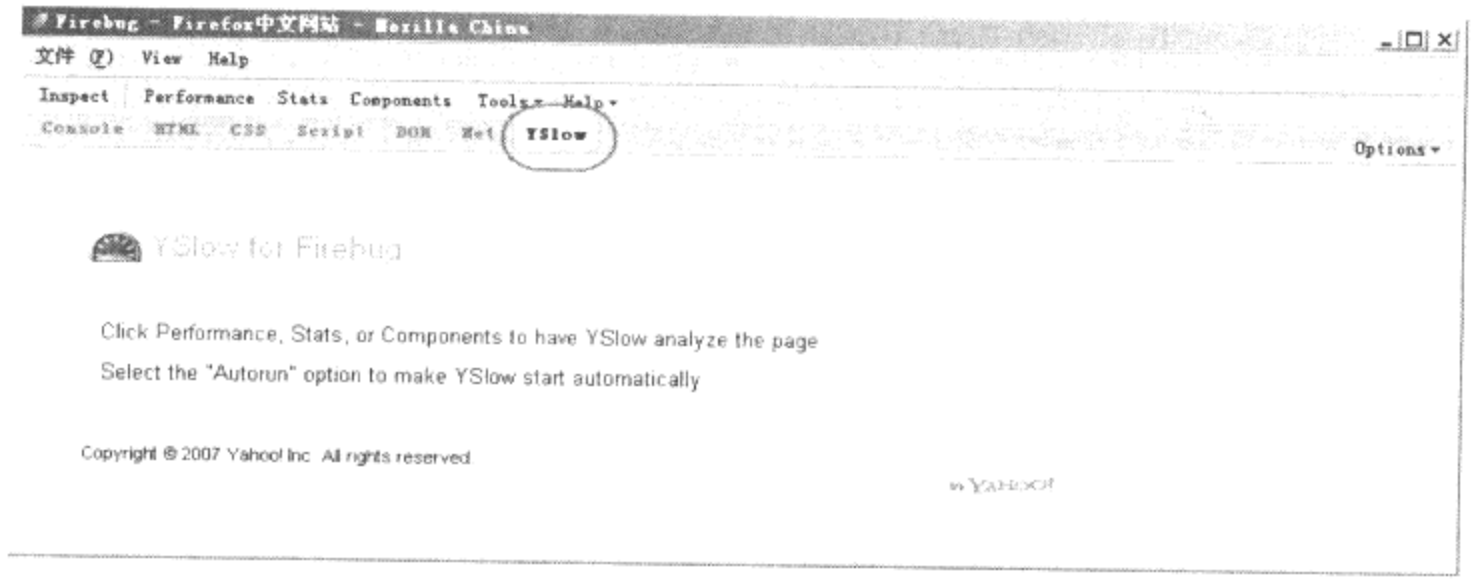


图 3：在菜单中先打开 Firebug 插件

单击【Performace】菜单

YSlow 便开始分析此页的效率，并从 13 个最影响网页效率的方面给出评估（见图 4）。

可以看出来，YSlow 评估的依据就是我们在《如何提高网页的效率（上篇）——提高网页效率的 14 条准则》中提到的前面 13 条。第 1 例的字母表示这一条准则的得分（A 最高）。点击右面的三角形可以得到更多的信息和建议，有些

信息里面还有“放大镜”图标，点击也将展示更为详细的信息和建议（见图 5）。

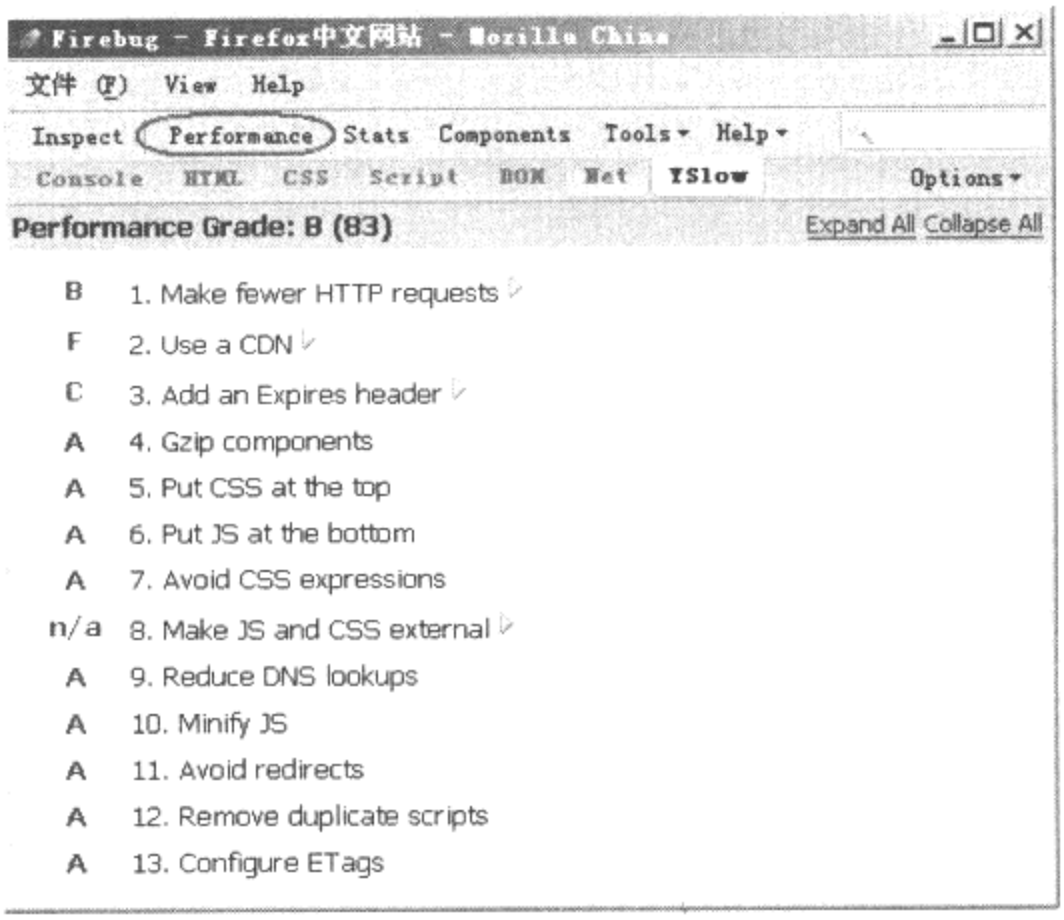


图 4：YSlow 给出的本页面效率评估

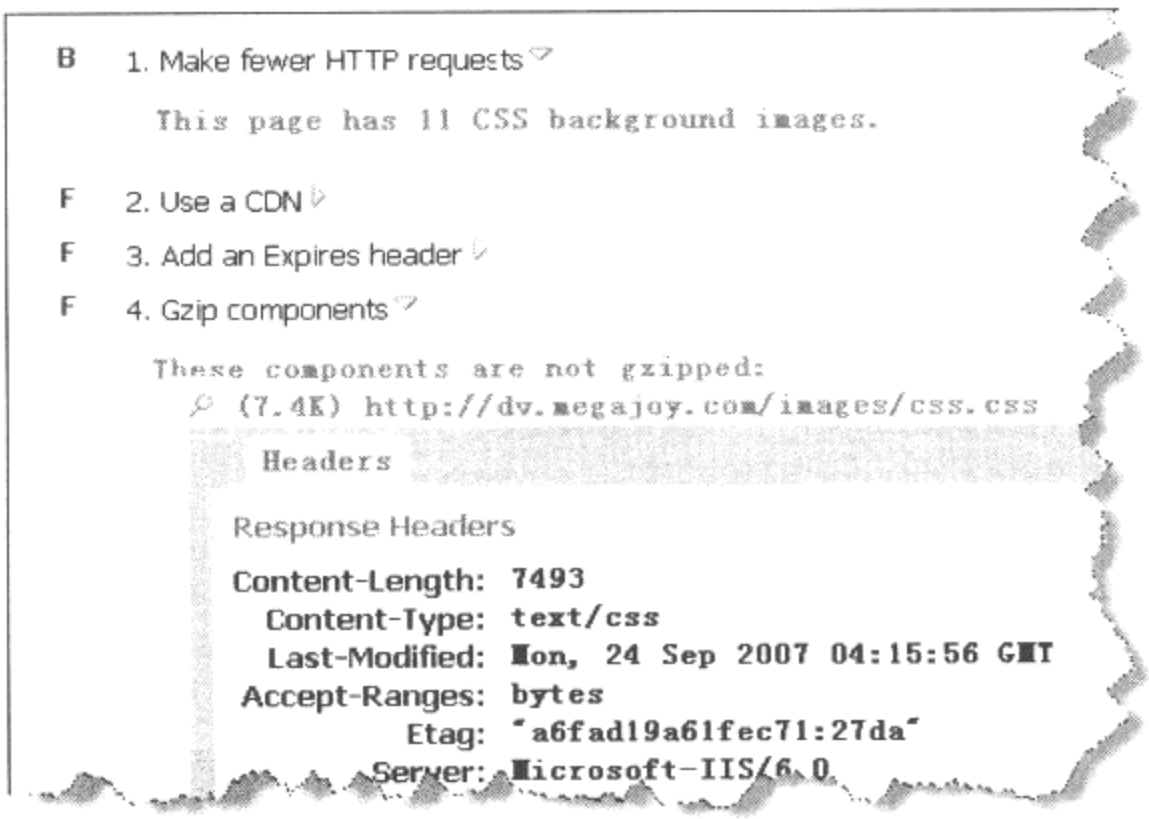


图 5：YSlow 可以给出每条准则的详细评估信息和建议

点击【Stats】菜单

这个视图会告诉你页面的总体统计信息（见图 6）。包括页面大小、CSS 样式表大小、脚本文件大小、总体图片大小、Flash 文件大小和 CSS 中用到的

图片文件大小。还会告诉你，哪些东西被缓存了，缓存了多少等。

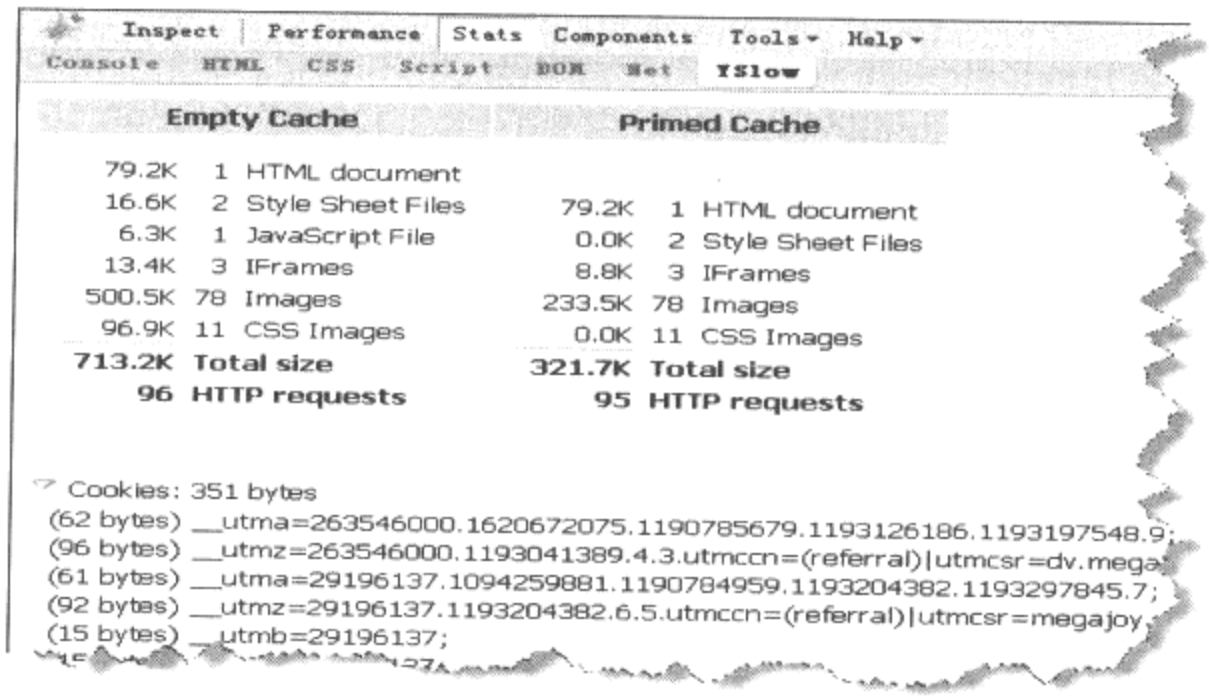


图 6：【Stats】视图信息

单击【Components】菜单

这个视图是一个页面所有部件的信息列表（见图 7）。从中我们可以得知每个部件的各种详细信息，如类型、URL、Expires 数据、状态、大小、读取时间、ETag 信息等。通过对这个列表的分析，我们就可以知道到底是什么东西最耗费我们的资源，从而有针对性地进行优化。

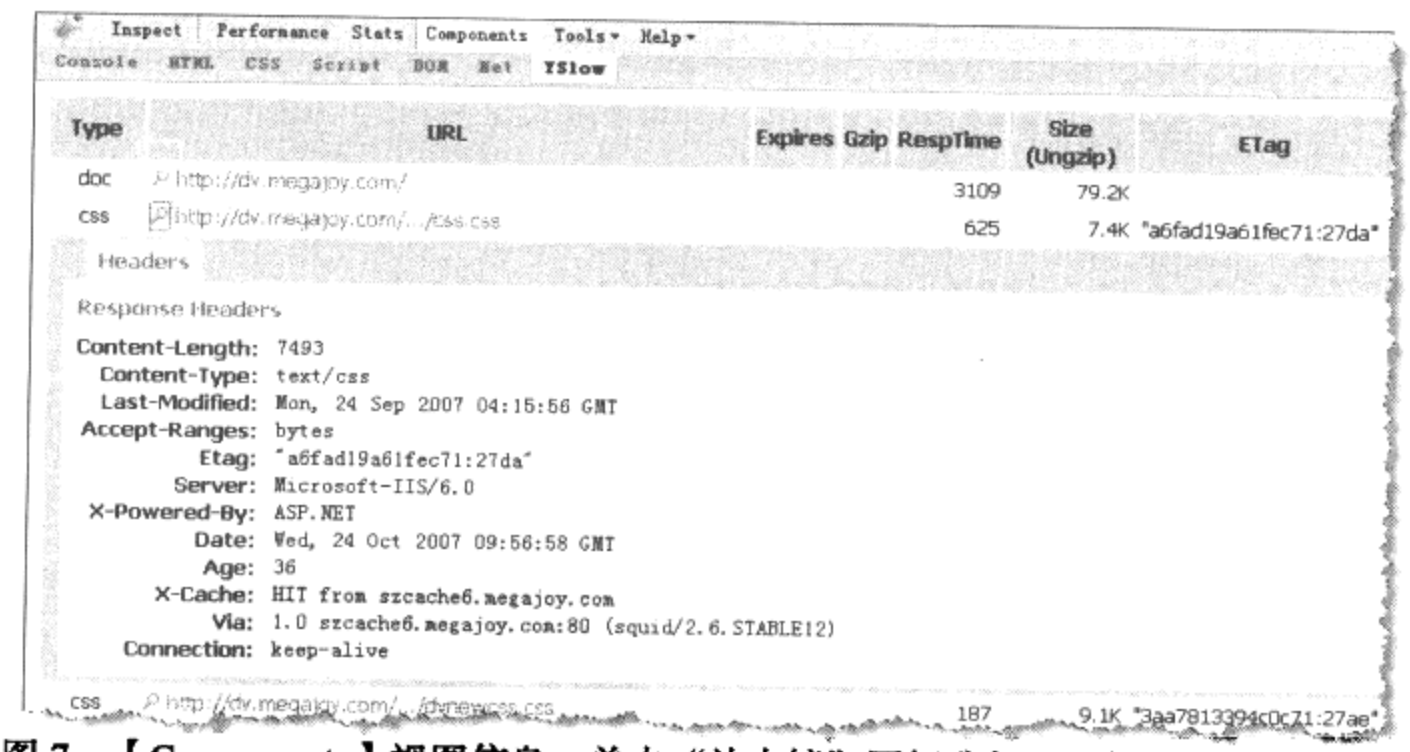


图 7：【Components】视图信息，单击“放大镜”图标我们可以知道更详细的信息

单击【Tools】菜单

【Tools】菜单包含 4 个子菜单，就是 4 个实用工具（见图 8）。【JSLine】工具会生成 JSLine 报表，报表是对本网页中 JS 脚本的分析报告，包含错误和

建议。【ALL JS】工具，将生成本页面所有脚本代码便于阅读和打印的报表页面。【ALL CSS】工具，将生成本页面所有 CSS 样式表代码便于阅读和打印的报表页面。【PrinTable View】将【Performance】和【Stats】视图中的信息生成一份更适合阅读和打印的报表页面。

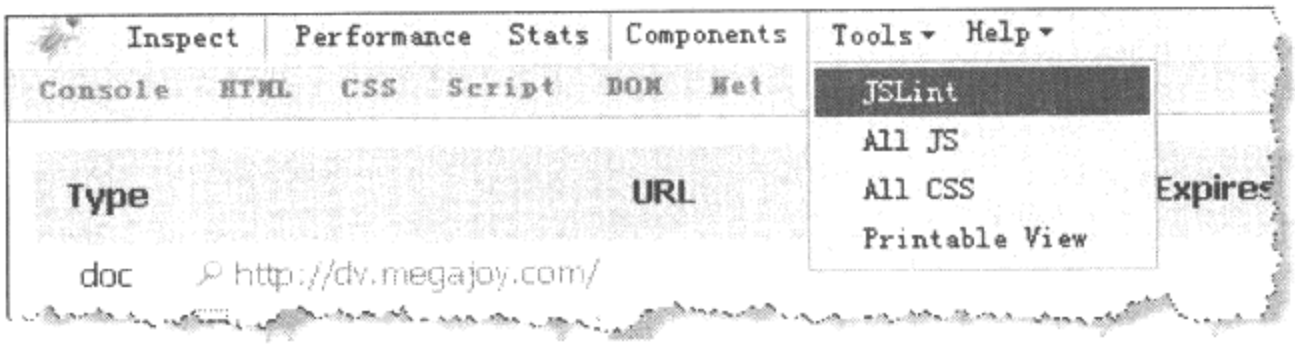


图 8：【Tools】菜单，包含了 4 个子菜单

单击【Help】菜单

【Help】主要是些常用的帮助途径的入口（见图 9）。从这里你可以很方便地访问 YSlow 的官方网络和博客。如果你还对 YSlow 的使用有什么疑惑的话，那么在这里你将获得满意的解答。

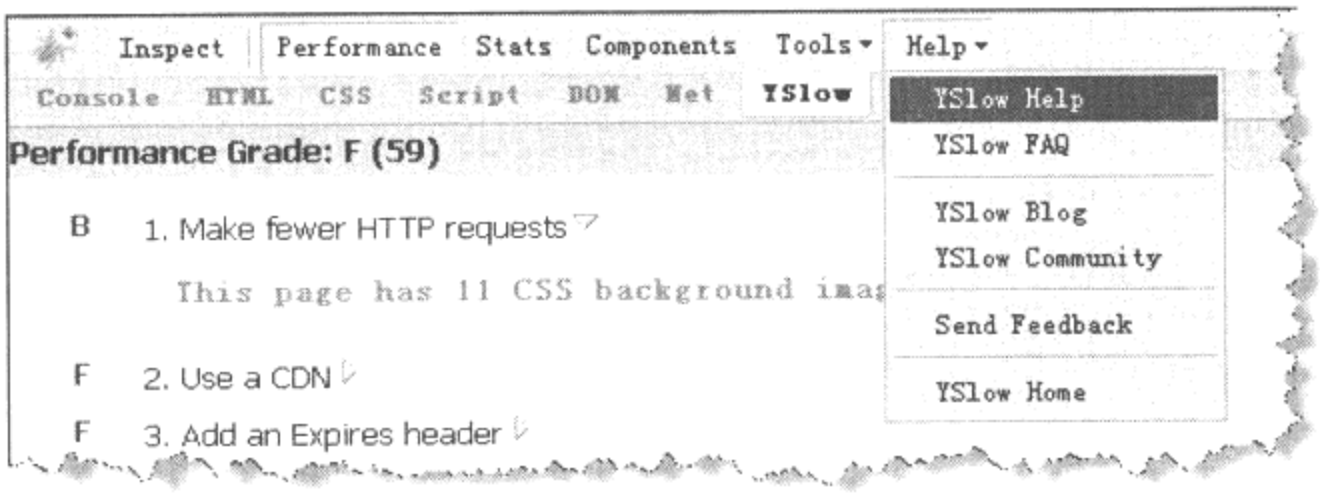


图 9：【Help】菜单是些常用的帮助入口

后记

“工欲善其事，必先利其器!”好的工具的确能够提高我们的工作效率。但是“阿斗”就算手里拿着“方天画戟”，估计也没有几个人怕他。好的工具是一方面，但是更重要的还是提高我们自身的知识水平。就如同这款 YSlow，如果没有《如何提高网页的效率（上篇）——提高网页效率的 14 条准则》中的理论知识，工具提供的信息我们看到的可能只是表面，就算看懂了数据，也很难知道对应的手段和措施。壮汉拿厉斧，这样才能伐木。

关于 Web 应用程序安全的思考

作者： Kevin Zou [http://www.cnblogs.com/tsoukw/]

序

曾经在一家公司有过这样的经历：上班第一天，同事在公司的内部网上帮我开了一个账号，要我登录公司的管理系统学习一下公司的管理制度。看完这些“文件”后，我随便点了一下系统左边的“员工信息查询”菜单，随即右边网页的数据区域显示“您无权查看此页”的错误信息，本想退出，但发现该页面的查询条件输入区域仍在，而且查询按钮也只是灰掉而已，在查看了网页原代码后，抱着随便试一下的心态，我在浏览器的地址栏里输入了一行 JS 代码：javascript:alert(document.all['querybtn'].disabled=false) 使查询按钮启用，然后单击它，居然真的把人事基本资料给查了出来，随后我又打开这个系统的其他页面，发现都只是把动作按钮给 disable 掉来管理权限。

当把人事薪资等非常敏感的资料放在 Web 系统中时，如果只是通过上面这种方式来保证数据不被非法读取，很明显这个系统没有达到它应该达到的安全级别。

作为一个 Web 系统设计师，在规划一个系统时，必然会考虑到系统的安全性。如何有效地保证系统的安全，如何规划和实现一个可重用、可扩展的安全管控方案都是在安全管控时要考虑的主题。

借着这个机会，笔者打算将自己从事 Web 系统安全设计的经验和大家分享。从一个系统设计师的角度说明 Web 系统安全管控。

- Web 运作原理，您的系统到底有多安全?
- 权限抽象，还一个统一的权限接口。
- 管控观念转换，柳暗花明又一村。
- 通用安全组件，从此不再苦海挣扎。

Web 运作原理

Web 是由客户端（Client）的请求（Request）和服务器（Web Server）的

响应（Response）构成，同一个客户端的多次 Request 对于 Web Server 来说都一样，服务器不会将当前收到的 Request 和以往任何的 Request 联系起来，因为它们交互的依据是 HTTP，而此协议规定了 HTTP 连接的无状态特征。

以下为一个最简单的 Request 请求：

```
GET /TestWeb/test.htm HTTP/1.1
Host: localhost
Connection: close
```

它表示向 localhost 主机请求路径为/TestWeb/test.htm 的 html 网页，使用 GET 方法，1.1 版本的 HTTP。

对此请求，Windows 的 IIS 6.0 是这样给出 Response 的：

```
HTTP/1.1 200 OK
Content-Length: 12
Content-Type: text/html
Last-Modified: Wed, 05 Nov 2008 01:01:17 GMT
Accept-Ranges: bytes
Server: Microsoft-IIS/6.0
Date: Wed, 05 Nov 2008 01:01:52 GMT
Connection: close

Hello World!
```

包括响应状态码 200、body 的长度、类型、所请求文件的最后修改日期等响应头（Response Header），还有简单的 Hello World! 12 个字符的 html 响应体（Response Body）。

Request 请求不依赖于浏览器，事实上您可以使用任何程序语言通过网络编程来做到，以下是一个 C#发送 Request 的例子：

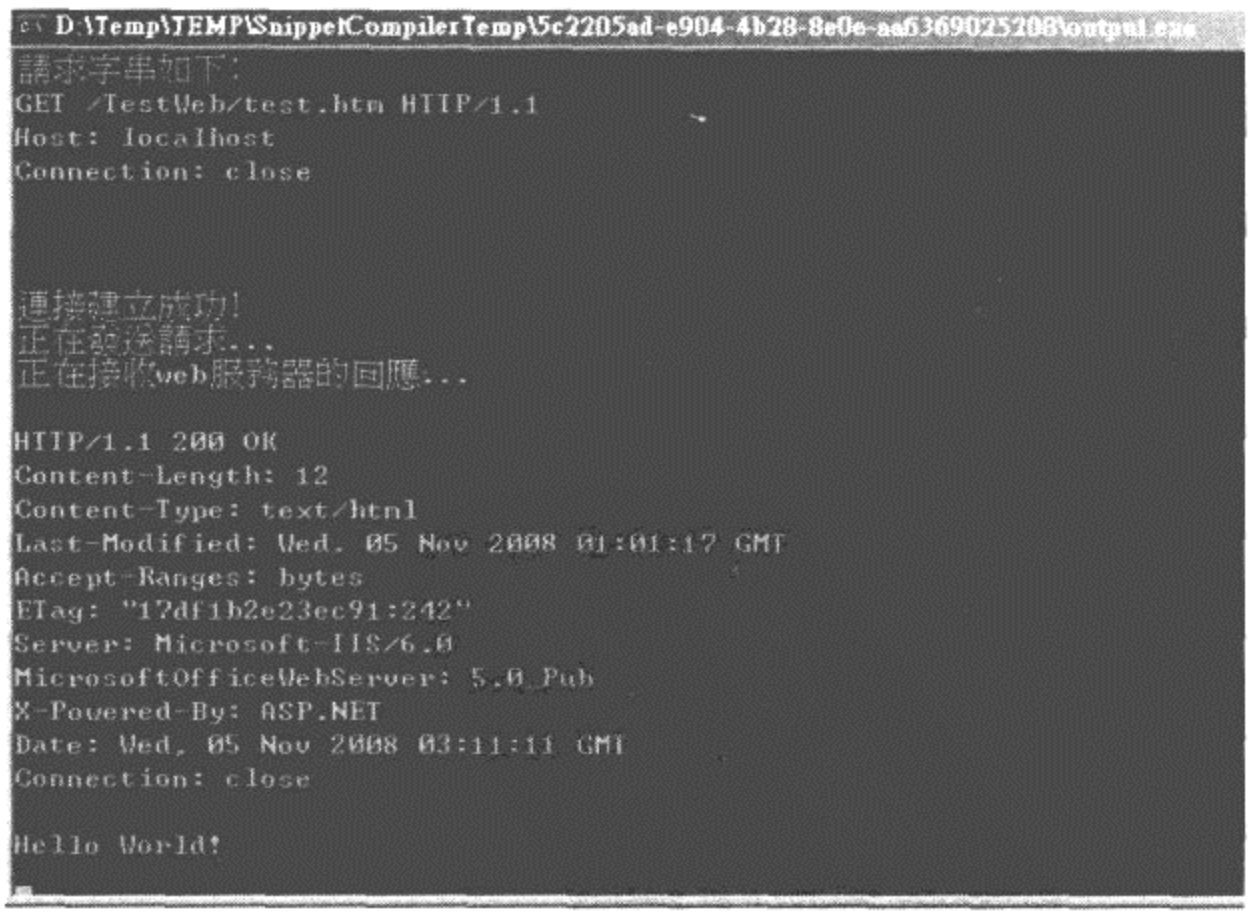
```
using System;
using System.Text;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text.RegularExpressions;

public class RequestDemo
{
    //建立 socket 连接
    private static Socket ConnectSocket(string server, int port)
    {
```

```
            Socket s = null;
            IPHostEntry hostEntry = null;
            hostEntry = Dns.GetHostEntry(server);
            foreach (IPAddress address in hostEntry.AddressList)
            {
                IPEndPoint ipe = new IPEndPoint(address, port);
                Socket tempSocket =
                    new Socket(ipe.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);
                tempSocket.Connect(ipe);
                if (tempSocket.Connected)
                {
                    s = tempSocket;
                    break;
                }
                else
                    continue;
            }
            Console.WriteLine(s == null ? "" : "连接建立成功！");
            return s;
        }

        //发送 request 请求并接收响应字符串
        private static string SocketSendReceive(string request, string
server, int port)
        {
            Byte[] bytesSent = Encoding.ASCII.GetBytes(request);
            Byte[] bytesReceived = new Byte[256];
            Socket s = ConnectSocket(server, port);
            if (s == null)
                return ("连接失败！");
            Console.WriteLine("正在发送请求...");
            s.Send(bytesSent, bytesSent.Length, 0);
            int bytes = 0;
            StringBuilder responsestr = new StringBuilder();
            Console.WriteLine("正在接收 Web 服务器的回应...");
            do
            {
                bytes = s.Receive(bytesReceived, bytesReceived.Length, 0);
                responsestr.Append(Encoding.UTF8.GetString(bytesReceived,
0, bytes));
            }
            while (bytes > 0);
```

```
            return responsestr.ToString();
        }

        public static void Main(string[] args)
        {
            //读取在 Request.txt 中的 Request 字符串(request.txt 末尾至少要留个空
行，表明 Request 结束)
        string requeststr = File.ReadAllText("C:\\tmp\\request.txt")
        Console.WriteLine("请求字符串如下：\n{0}\n", requeststr;

            //发送且接收 Response
            string result = SocketSendReceive(requeststr, "localhost", 80);
            Console.WriteLine("\n{0}", result);
            Console.ReadLine();
        }
    }
```


注： C:\tmp\request.txt 中的内容就是前面的 Request 字符串。

程序执行的结果如下：



除了直接进行 Request 外，大部分时候，我们在网页上单击某个链接或按

钮时，浏览器和 Web 服务器也在背后进行着这样的请求和响应。

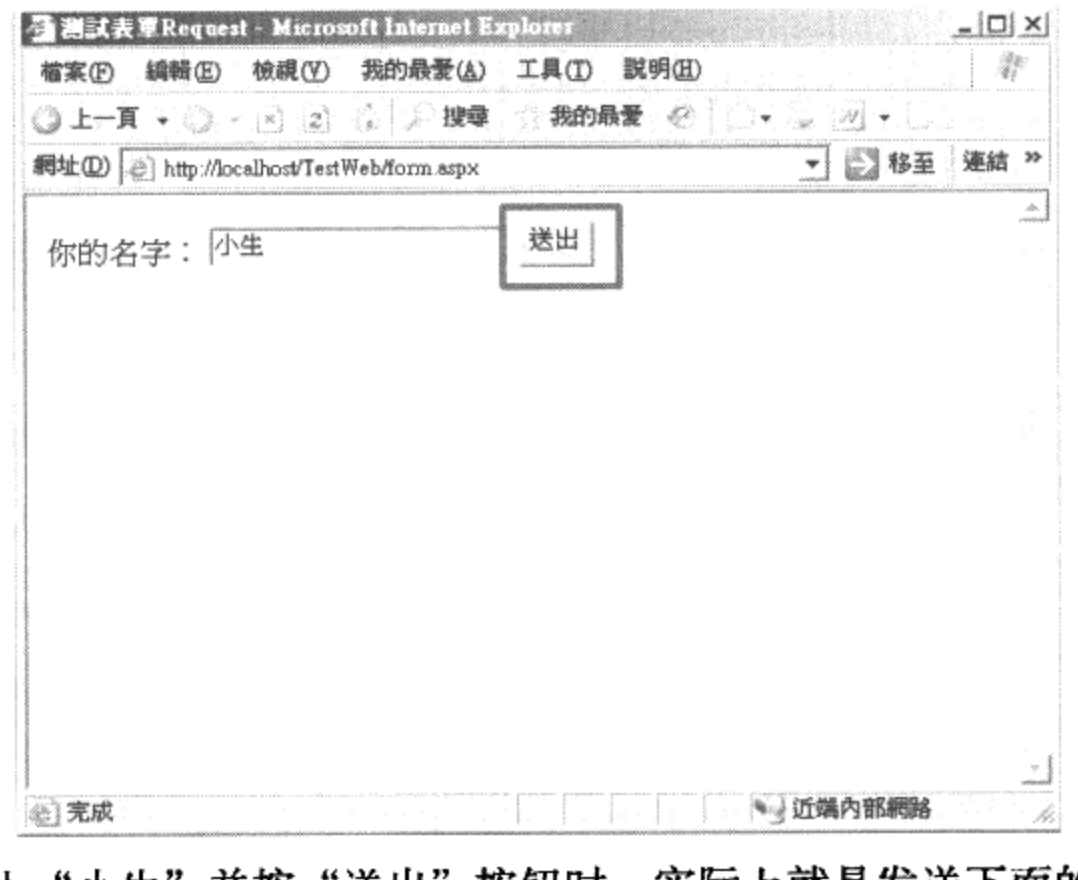
例如以下网页程序：

```
    <%@ Page Language="C#" AutoEventWireup="true" CodeFile=
"form.aspx.cs" Inherits="form"
    %>
    <!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>测试窗体 Request</title>
    </head>
    <body>
        <form id="form1" runat="server">
        <div>
            <asp:Label ID="Label1" runat="server" Text="你的名字:
"></asp:Label>
            <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
            <asp:Button ID="Button1" runat="server" OnClick=
"Button1_Click" Text="送出" />
            <asp:Label ID="Label2" runat="server" ForeColor=
"OrangeRed"></asp:Label></div>
        </form>
    </body>
    </html>
```

Aspx.cs 代码如下：

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class form : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
```

```
    }
    protected void Button1_Click(object sender, EventArgs e)
    {
        Label2.Text = "你输入的名字是: " + TextBox1.Text;
    }
}
```



当输入“小生”并按“送出”按钮时，实际上就是发送下面的这样一段Request：

```
    POST /TestWeb/form.aspx HTTP/1.1
    Cache-Control: no-cache
    Connection: close
    Content-Length: 206
    Content-Type: application/x-www-form-urlencoded
    Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
    Accept-Encoding: gzip, deflate
    Accept-Language: zh-tw
    Cookie: ASP.NET_SessionId=jd14mp2k4e0dyga4hjz1zgby
    Host: localhost
    Referer: http://localhost/TestWeb/form.aspx
    User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2;
SV1; .NET CLR 1.1.4322;
    .NET CLR 2.0.50727; .NET CLR 3.0.04506.30)
    UA-CPU: x86
```

```
__VIEWSTATE=%2FwEPDwUJODUwMjI0NzE3ZGQgpj%2Fm%2BSOD2vEbxBDW9BpDvo
gpgA%3D%3D&TextBox
1=%E5%B0%8F%E7%94%9F&Button1=%E9%80%81%E5%87%BA&__EVENTVALIDATIO
N=%2FwEWAwKdv8y5Bw
Ls0bLrBgKM54rGBj5ZvpRog0Ox8f9YoKD3sYnCmNxG
```

其中 TextBox1 后面的%E5%B0%8F%E7%94%9F 就是“小生”(encodeURI 函数编码的结果)，修改此行数据，并对 Content-Length 作适当调整，就完全可以模仿按下“送出”按钮的动作。

实际上，不管是直接在地址栏输入 url，还是在网页上单击链接、提交窗体、Ajax 请求、Web Service 呼叫等，客户端都需要向 Web Server 发送相关的 Request。因此对于管控比较全面的安全方案，必须对每一次 Request 都进行验证。而且这个验证最好在所有的被请求程序执行之前就完成，就好比您的 Web 应用程序是一个大型游乐场，而您的安全管控就是这个游乐场唯一入口的检票处，凡进入游乐场的人员，都是买了票的，至于游乐场里面一些还要买票的项目，那就不属于入口检票处的责任了。

可以对比一下手上的系统，看是否每次 Request 都进行了管控？对没有管控到的 Request 会不会发生问题？安全隐患的机率有多大?值不值得再加一次管控?

权限本质探讨

所谓安全管控，其实就是权限判断，即对用户能否访问某一权限对象进行判断，并且在无权访问时进行相应处理。

权限，实际上就是使用者与权限对象的一种多对多的关系。

如一个权限厂别的数据如下：

```
UserID      FactID
-------------------------
1           1
1           2
2           3
```

表示 UserID 为 1 的用户拥有 FactID 为 1 和 2 的权限，UserID 为 2 的用户拥有 Fact ID 为 3 的权限。权限的本质在于对权限的使用方式进行抽象。

以下为常见的权限接口，主要包括判断权限和获取权限列表：

```
/// <summary>
/// 权限接口
/// </summary>
```

```
interface IRightProvider
{
/// <summary>
/// 判断权限
/// </summary>
/// <param name="userID">用户 ID</param>
/// <param name="objectID">权限对象 ID</param>
/// <returns>true:有权限 false 无权限</returns>
bool HasRight(string userID, string objectID);

/// <summary>
/// 获取权限列表
/// </summary>
/// <param name="userID">用户 ID</param>
/// <returns>objectID 列表</returns>
List<string> GetRights(string userID);
}
```

判断权限 HasRight 方法

使用示例如下某 aspx.cs 的 Page_Load 代码：

```
…
string currentUserID = getUserID();          //获取当前登录用户
    string deleteUserRightID = "DeleteUser";    //删除用户的权限
(HardCode 删除用户的功能 ID)

    IRightProvider functionRight;
    //实例化权限功能对象
    //如: functionRight = new FunctionRightProvider()
    …

    //根据是否有权限,决定“删除”按钮是否显示
    if(functionRight.HasRight(currentUserID,deleteUserRightID))
         deleteBtn.Visible = true;
    else
         deleteBtn.Visible = false;
    …
```

获取权限列表 GetRights 方法

使用示例如下某 aspx.cs 的 Page_Load 代码:

```
    …
    string currentUserID = getUserID();                    //获取当前登录用户
```

```
IRightProvider factRight;
//实例化权限厂别对象
…

//根据厂别 ID 获取权限厂别名称
DataTable dt = GetRightData(factRight.GetRights(currentUserID));

//将权限厂别数据绑定厂别下拉列表控件（这样用户只可以选择权限内的厂别）
factDropDownList1.DataSource = dt;
factDropDownList1.DataTextField = "FactName";
factDropDownList1.DataValueField = "FactID";
factDropDownList1.DataBind();
…
```

上面这个例子，基于更安全的考虑，在 Query 按钮按下后，应该再对 factDropDownList1.SelectedValue 进行 HasRight 判断。

实际开发中，也常常会将权限绑定和取值时的 HasRight 判断的代码整个封装成一个 UserControl，这样就可以像普通 DropDownList 一样使用了。

另外权限数据库设计、权限管理方式（分配和移除权限）、群组策略、管理员策略等都是可以随系统大小来进行自定义，当然最好给出一个比较稳定的方案，这样就不用每次都去考虑这个问题。

以下为笔者常用权限架构方案

群组表：

```
Groups(GroupID,GroupDesc,AppID)
```

AppID 为系统 ID，因为笔者的所用系统基本上共享一套权限管控方案。如果权限只是 For 单个系统建表，AppID 可以省略，以下不再解释。

群组成员表：

```
GroupMembers (GroupID,UserID)
```

用户权限表：

```
UserRights(UserID,ObjectID,ObjectType,AppID)
```

ObjectType 表示 ObjectID 是什么数据，如 Fact 表示是厂别 ID，Function 表示是功能 ID。这样一次就可以满足多种权限的设定了。

群组权限表

```
GroupRights(GroupID,ObjectID,ObjectType,AppID)
```

管理员表

```
UserAdmin(UserID,ObjectType,AppID)
```

某个权限类别的管理员，如有这样一笔数据 UserID:1,ObjectType:Fact。表明 UserID 为 1 的用户拥有所有厂别权限。

有了这些架构，就可以轻松实现 IrightProvider：

```
/// <summary>
/// 默认权限实做(简单的判断权限)
/// </summary>
class DefaultRightProvider : IRightProvider
{
    IDataProvider _dataProvider;        //权限数据提供者
    public DefaultRightProvider(IDataProvider dataProvider)
    {
        _dataProvider = dataProvider;
    }

    #region IRightProvider 成员

    public bool HasRight(string userID, string objectID)
    {
        List<string> data = GetRights(userID);
        if (data != null && data.Contains(objectID))
            return true;
        return false;
    }
    public List<string> GetRights(string userID)
    {
        if (_dataProvider != null)
        {
            return new List<string>(_dataProvider.GetData(userID));
        }
        return null;
    }

    #endregion
}
/// <summary>
/// 权限数据的读取策略
/// </summary>
interface IDataProvider
{
```

```
        /// <summary>
        /// 获取某个用户的权限数据
        /// </summary>
        /// <param name="userID"></param>
        /// <returns></returns>
        IList<string> GetData(string userID);
    }

    /// <summary>
    /// 使用管理员，群组策略的权限机制
    /// </summary>
    class GroupAdminDataProvider : IDataProvider
    {
        string _appID;
        string _objectType;

        /// <summary>
        /// 某一系统，某一权限类别的权限数据访问接口
        /// </summary>
        /// <param name="appID"></param>
        /// <param name="objectType"></param>
        public GroupAdminDataProvider(string appID, string objectType)
        {
            _appID = appID;
            _objectType = objectType;
        }
        List<string> getGroupData(string groupID)
        {
            //调用数据访问层(或其他接口层)，获取 group 的权限对象
        }
        List<string> getUserData(string userID)
        {
            //调用数据访问层(或其他接口层)，获取 user 的权限对象
        }
        List<string> getAllData()
        {
            //调用数据访问层(或其他接口层)，获取当前类别的所有对象
        }
        List<string> getUserGroup(string userID)
        {
            //调用数据访问层(或其他接口层)，获取 user 所在的所有群组 ID
        }
        public bool IsAdmin(string userID)
        {
```

```
    //调用数据访问层(或其他接口层)，判断管理员权限
}
void addNoReplica(List<string> desList, List<string> addList)
{
    if (addList != null && addList.Count > 0)
    {
        foreach (string addItem in addList)
        {
            if (!desList.Contains(addItem))
                desList.Add(addItem);
        }
    }
}
public IList<string> GetData(string userID)
{
    if (IsAdmin(userID))
        return getAllData();
    else
    {
        List<string> ret = new List<string>();

        //加入本身权限
        List<string> userDatas = getUserData(userID);
        addNoReplica(ret, userDatas);

        //加入群组权限
        List<string> groups = getUserGroup(userID);
        if (groups != null)
        {
            foreach (string groupID in groups)
            {
                List<string> groupDatas = getGroupData(groupID);
                addNoReplica(ret, groupDatas);
            }
        }
        return ret;
    }
  }
}
```

其中省略的部分都是通过对上述权限数据库的访问来完成的。

除了 HasRight 和 GetRights 接口外，涉及权限最多的就是权限管理了，包括群组建立、群组权限分配、用户加入群组、用户权限分配、用户管理员设定等，都可以封装成控件并且完成对上述权限数据库的增删改查，这样在

以后涉及新权限时，权限管理部分都不用再额外投入开发成本了。

这里特别的说明就是用户权限表：

```
UserRights(UserID,ObjectID,ObjectType,AppID)
```

任何权限抽象成 ObjectID 统一了各种权限的处理，如果对于某种权限单凭一个 ObjectID 不好说明时，可以增加一个 ObjectID 的描述档。例如有这样一种权限，用户在一个系统中对于不同的报表所拥有的权限不一样，有的只可以查询，有的却可以转 Excel，有的还可以转 PDF。这时候可以设计一个报表权限主文件：

ReportRightID	报表 ID	动作方式
1	订单报表	查询
2	订单报表	转 Excel
3	采购报表	转 PDF

而 ObjectID 存的就是 ReportRightID 了，当然这个报表权限主文件的增删改就是在分配权限时动态完成的，以避免加入很多用不到的数据。

判断权限时，需要增加一个动作，即先根据需要判断的报表 ID 和动作方式获取 ReportRightID，再调用 HasRight 方法判断。而获取权限如某 user 能够查询的报表有哪些，某 user 对订单报表的权限动作有哪些，同样只要扩展 GetRights 方法就可完成（基于时间的关系，就不再深入。如果有兴趣，可以联系笔者再行探讨）。

上述工作完成后，判断权限就十分容易了，如厂别权限的判断：

```
    string FACTRIGHTTYPE = "FACT";      //权限类别
    //实例化权限数据获取器
    GroupAdminDataProvider dp = new GroupAdminDataProvider(appID,
FACTRIGHTTYPE);
    //实例化权限类别
    IRightProvider factRight = new DefaultRightProvider(dp);
    factRight.HasRight(userID, factID); //判断权限
```

有了这样互相支持，又相互独立的权限使用和权限管理分离理念后，权限使用就变得非常简单，权限架构又可以任意扩展，灵活变化，最终达到权限接口的统一使用。

换一种安全管控观念

先给出一段在 aspx 中常见的安全管控代码。假设登录后 UserID 存放在 Session["UserID"]中，此页面是一个用户信息的修改删除页面，有两个按钮：

SaveBtn（保存修改按钮），DeleteBtn（删除用户按钮）。

有三个角色：管理员：可以进行修改和删除；主管：只可以修改；普通用户：只能浏览。

以下为权限管控相关的代码片段：

```
if(Session["UserID"] == null)
    Response.Redirect("Login.aspx");   //未登录转向登录页面
string userID = Session["UserID"].ToString();

//预设不能进行任何动作(普通登录用户只能看)
SaveBtn.Visible = false;
DeleteBtn.Visible = false;

if(IsAdmin(userID)){          //如果用户是管理员，则可以修改和删除
    SaveBtn.Visible = true;
    DeleteBtn.Visible = true;
}
else if(IsManager(userID)){  //如果用户是主管,则只能修改
    SaveBtn.Visible = true;
}
```

顺便提一下：如果安全级别要求更高的话，在 SaveBtn_Click 和 DeleteBtn_Click 的事件中也要加入 IsAdmin 和 IsManager 的相关代码，以保证不被修改并仿真这些按钮的动作（要验证的话，可以修改 Web 运作原理一节中的 form.aspx，将送出按钮的 Visible 设为 False，看是否可以触发 Click 事件）。

这样的权限管控方式对于小型、简单的系统能够满足，但是系统比较大或者对权限的要求复杂一些时，就会发生问题。

1. 每个页面都需要写权限管控代码（即使能将这样的类似代码封装成一个方法，由于涉及具体页面的具体控件，所以在 aspx 中还是少不了这样的权限管控代码）。

2. 权限代码与业务逻辑本质上彼此无关，勉强放在一起会违反低耦合原则，因此在权限变动或修改时（例如增加一种角色，增加一个按钮）都可能会影响到这些代码。

3. IsAdmin 和 IsManager 等都属于硬编码（hardcode）方式，在角色或权限变动时要重新修改代码，而且更重要的是无法动态管理权限（如动态分配角色的权限）。

4. 适用范围有限，基本上只适用于 aspx 文件，而如果系统需要管控 xml、jpg、ashx、Web service 等其他文件或程序的权限时，又要进行权限设计。

5. 权限管控代码无法重用，每个新系统开发时都要重新考虑权限管控。

要解决上面的这些问题，关键就是权限管控观念的转变，避免在业务程

序中直接或间接加入权限判断、角色管控等代码（原因见上述第 2、3 点）。

具体来讲，就是程序员在开发每一个程序时，就应该想到，user 在操作这个程序时，他就应该已通过安全管控，而程序员在写这个程序时，也只要考虑如何实现这个程序要完成的功能就行。

例如上面这个例子，权限管控与程序本身的显示、修改和删除逻辑纠缠在一起，耦合度过大，使得这两者在有变化时互相影响，增加了程序的复杂性。如果转成这个程序只处理与业务逻辑有关的功能，成为上节中所讲的一种权限对象——系统功能，这样就可以像普通的厂别权限一样进行统一的权限判断和管理了。

从用户需求得知，有 4 个这样的系统功能需要实现，分别是新增、删除、修改和查看用户信息。

那可能首先就会写四个程序，分别实现 4 个功能：

- 查看用户——UserView.aspx
- 修改用户——UserEdit.aspx
- 删除用户——UserDelete.aspx
- 新增用户——UserAdd.aspx

但是在开发过程中，发现前三个程序的代码差不多，因此重构改为一个程序，并且使用一个 Kind 的 QueryString 来区别：

- 查看使用者——User.aspx?Kind=View
- 修改使用者——User.aspx?Kind=Edit
- 删除使用者——User.aspx?Kind=Delete

代码如下：

```
string kind = Request.QueryString["kind"];
if(kind==null)
    kind = "View";   //如果没有 kind，则默认为只读

ConfirmBtn.Visible = false;   //确定按钮
DeleteBtn.Visible = false;    //删除按钮

if(kind == "Edit")
   ConfirmBtn.Visible = true;   //如果是修改，显示确定按钮
else if(kind=="Delete")
   DeleteBtn.Visible = true;    //如果是删除，显示删除按钮

...
```

这样也可以完成相同的功能。

随着需求进一步的明确，user 希望管理员能够在同一个页面中进行修改

和删除用户。

因为功能变得可以组合，所以要换一下 Kind 的定义方式，将 Kind 分为 2 码 01 字符，分别表示删除和修改功能，如：

User.aspx?Kind=01 表示这支程序实现的功能是修改用户，而 User.aspx?Kind=11 则表示这支程序实现的功能是新增和修改。

代码如下：

```
string kind = Request.QueryString["kind"];
if(kind==null)
    kind = "00";   //如果没有 kind，则默认为只读

ConfirmBtn.Visible = false;   //确定按钮
DeleteBtn.Visible = false;    //删除按钮

bool isDelete = kind[0]=='1';
bool isEdit = kind[1]=='1';

if(isEdit)
   ConfirmBtn.Visible = true;   //如果有修改功能，显示确定按钮
if(isDelete)                    //改为 if，而不是 else if，这样功能可
任意组合
   DeleteBtn.Visible = true;    //如果有删除功能，显示删除按钮
……
```

最后实现的系统功能有：

- 用户管理　　User.aspx?Kind=11
- 用户修改　　User.aspx?Kind=01
- 用户查看　　User.aspx
- 新增用户　　UserAdd.aspx

这样就完成了权限管控之前的准备工作，即将系统功能看作是一种权限对象，到时再对这种权限对象进行相应的管理和判断即可。

接下来的事交由权限管控组件统一进行。

可能有的读者会认为如果修改和删除权限动态组合怎么办，即有的 user 只可以删除，有的 user 只可以修改，有的 user 可删可改，而且如果一个程序实现了很多的功能组合，那这边的系统功能是不是很难产生？

的确会有这种问题，可以通过以下方式解决：

1. 进一步明确需求，看是否真的有随意的权限组合这样的需求。像上面这个程序，真实应用中可能就只有两种权限——管理员可以增、删、改、查，而普通用户则只可以查询。这样就可以静态地添加两个系统功能及其对应的

程序。

2. 而如果用户确实存在系统运行时权限动态变化的可能时，那也只要再加强权限分配程序，当在给用户分配系统功能时，提供相关系统功能的子选项（如用户信息功能，有查看、删除、可修改的子功能）让用户自行勾选，然后程序再动态地增加系统功能与程序的对应就行。

不管如何，由于统一了权限使用接口，抽象了权限对象的观念，因此使得上层的权限判断代码能够保持稳定而且共享权限判断组件。

在实际开发过程中，大部分都没有这么复杂，往往一个程序就实现一个系统功能，或者多个程序共同实现一个系统功能，或多个系统功能需要一个程序来实现（程序中不需要区分不同功能，如一些共享程序）。这样在程序中连进行最简单的区分不同功能的代码都可以省下，转为纯粹的业务逻辑实现，更好地实现安全管控。

类似的，其他各种的需要权限管控的资源，如 Web service、ashx、xml 和 jpg 等都可以建立类似的对应。为了方便，可以灵活运用多种对应方式，如整个目录对应到一个系统功能，省却将一个个程序加入数据库的麻烦。

换一个角度，转变一下观念，正是山穷水复疑无路，柳暗花明又一村。

通用安全组件设计

有了权限统一接口，有了新的 Web 安全管控思维，接下来的安全组件设计也就顺理成章了。

安全管控组件其实就是一个管控流程类（SecurityProvider），而管控流程（Valid）也就是：认证→写入 UserID→授权→拒绝访问处理。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Web;

namespace WebSecurity
{
    /// <summary>
    /// 安全管控流程提供者
    /// </summary>
    public class SecurityProvider
    {

        /// <summary>
```

```
/// 进行安全管控（管控流程:认证->写 UserID->授权->拒绝访问处理）
/// </summary>
/// <param name="context"></param>
public void Valid()
{
    string userid = authenticate();   //认证(获取 user id)
    if (userid != null)
        setUserID(userid);            //写 UserID
    if (!authroize(userid))           //授权(是否可以访问)
        forbidAction();               //拒绝访问处理
}

/// <summary>
/// 识别当前 Request 的 User ID
/// </summary>
/// <returns>
/// 返回当前 Request 的 User ID,如果未登录,则返回 null
/// </returns>
string authenticate()
{

}
```

```
        /// <summary>
        /// 与应用程序的接口，指定 UserID 存放地点(如: Session 或
context.Items["UserID"])，以便应用程序使用
        /// </summary>
        /// <param name="context"></param>
        /// <param name="userid"></param>
        void setUserID(string userid)
    {

    }

        /// <summary>
        /// 对当前用户(包括匿名用户)进行授权
        /// </summary>
        /// <param name="context">可以匿名访问的 URL 也在这里处理</param>
        /// <param name="userid"></param>
        /// <returns></returns>
        bool authroize(string userid)
        {
```

```
            }

            /// <summary>
            /// 拒绝访问采取的措施（如转向登录页面，提示无权登录信息，输出 soap
error message 等）
            /// </summary>
            /// <param name="context"></param>
            void forbidAction()
            {

            }
        }
    }
```

1. 认证

认证就是识别当前发出请求(Request)的用户，相对于程序来说就是UserID。识别 UserID，与具体系统、具体程序类型有关。

一般通过浏览器直接发出的 Request，其 UserID 的获取与系统的登录程序相关，如登录后 UserID 存在 Session 或 Cookie 中，那在这里 UserID 就是通过 Session 或 Cookie 的获取。

如下面这个类，可以提供给登录 aspx 和认证方法共享，其中登录时调用 LoginInPage 方法，而在认证中就可以调用相应的 GetUserID 方法。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Web;

namespace WebSecurity
{
    class LoginHelper
    {
        /// <summary>
        /// 登录状态码
        /// </summary>
        enum LoginStatus
        {
            Success,                    //成功
            FirstLogin,                 //成功，但是首次登录，客户程序可以选择导
向修改密码程序
```

```
        NoApproval,          //账号还未审核(未启用)
        AccountNotExists,    //账号不存在
        PasswordInvalid,     //密码错误(为了防止 hack，也可以不提供这么明确的错误信息但这个可以给客户程序选择)
        AccountDisabled      //账号被停用
    }
    /// <summary>
    /// 用户在网页上登录(需要写入登录票据)
    /// </summary>
    /// <param name="account">账号</param>
    /// <param name="password">密码</param>
    /// <param name="statusCode">登录状态</param>
    /// <return>登录状态</return>
    public LoginStatus LoginInPage(string account, string password)
    {
        LoginStatus retStatus;
        string retUserID = Login(account, password, out retStatus);
        if (retUserID != null)
            writeUserID(retUserID);     //写入 UserID
        return retStatus;
    }
    /// <summary>
    /// 登录
    /// </summary>
    /// <param name="account">账号</param>
    /// <param name="password">密码</param>
    /// <param name="statusCode">登录状态</param>
    /// <returns>返回 UserID,null 表示登录失败</returns>
    public string Login(string account, string password, out LoginStatus statusCode)
    {
        //一般是访问数据库以判断账号密码是否 OK，以及其他登录管控策略(如被停用等)
    }

    public static readonly string UserIDKey = "UserID";

    /// <summary>
    /// 写入登录 UserID
    /// </summary>
    /// <param name="userID"></param>
    /// <returns></returns>
    void writeUserID(string userID)
```

```
    {
        //Session 示例
        HttpContext.Current.Session.Add(UserIDKey, userID);
    }

    /// <summary>
    /// 注销
    /// </summary>
    public void Logout()
    {
        clearUserID();
    }
    /// <summary>
    /// 清除登录票据
    /// </summary>
    /// <returns></returns>
    void clearUserID()
    {
        //Session 示例
        if (GetUserID() != null)
            HttpContext.Current.Session.Remove(UserIDKey);
    }
```

```
        /// <summary>
        /// 获取登录成功后写入的 UserID
        /// </summary>
        /// <returns></returns>
        public string GetUserID()
        {
            string ret = null;
            //Session 示例
            if (HttpContext.Current.Session != null &&
HttpContext.Current.Session[UserIDKey] != null)
                ret = (string)HttpContext.Current.Session[UserIDKey];
            return ret;
        }
    }
}
```

对于一些比较特别的 Request，有其自己的 UserID 获取方式。如 Web Service，可能 UserID 会以加密方式存放在 SoapHeader 中；也有可能直接在 SoapHeader 中传送账号密码，而在认证过程中调用 Login 方法来取得当前 UserID。

```
    string authenticate()
    {
        HttpRequest request = HttpContext.Current.Request;
        string lowerNoQueryPath =
request.Url.PathAndQuery.Split('?')[0].ToLower();
        LoginHelper loginHelper = new LoginHelper();
        if (lowerNoQueryPath.EndsWith(".aspx"))
            return loginHelper.GetUserID();
        else if (lowerNoQueryPath.EndsWith(".asmx"))
        {
            //在 Request 的 Body 中解析 Soap 头，读取 userid 和 password 值
            //然后调用 loginHelper.Login 方法登录

        }
    }
```

此方法内代码可以灵活编写，便于重用、动态加载等。

2. 写入 UserID

安全管控组件由于与应用系统无关，因此必须提供一种统一的方式以便各种程序（如 aspx、asmx）在自己的代码中获取 userID。

```
    void setUserID(string userid)
    {
    HttpContext.Current.Items.Add("UserID", userid);
    }
```

3. 授权

授权的代码较为简单，基本上就循环判断当前 Request 对应的系统功能就行。这里最重要的就是识别 Request 的系统功能，并依照各种规则映射到系统功能上（特别注意，如果请求的是无权限时转向的 url，则一定要返回 true，否则会形成死循环）。

```
    bool authroize(string userid)
    {
    //不同的方式可以不同的处理
    //以下为伪码实现
    HttpRequest request = HttpContext.Current.Request;
    string lowerNoQueryPath =
request.Url.PathAndQuery.Split('?')[0].ToLower();
    if (lowerNoQueryPath.EndsWith("login.aspx"))
```

```
    return true;


    AppUrl url = getUrl(HttpContext.Current.Request);  //获取当前的 url
对象
    if (url.MustAuth == "N")        //如果这支程序不需要授权，则通过
    return true;
    else if (userid == null)        //挡掉匿名登录者
    return false;
    string appid = getCurrentAppID();  //获取当前访问的 AppID;
    IRightProvider functionProvider = new DefaultRightProvider(new
GroupAdminDataProvider(appid, "FUNCTION"));   //Function 表示系统功能类别
的权限
    List<string> functions = url.GetMapFunctions();   //获取 url 所对应
的 Function ID 列表
    if (functions != null)
    {
    //只要拥有任何一个所对应的系统功能权限则放行
    foreach (string funid in functions)
    {
    if (functionProvider.HasRight(userid, funid))
    return true;
    }
    }
    return false;
    }
```

4. 拒绝访问处理

至于拒绝访问，也需要针对不同的类型进行处理，如：

```
    void forbidAction()
    {
                HttpRequest request = HttpContext.Current.Request;
                string lowerNoQueryPath =
request.Url.PathAndQuery.Split('?')[0].ToLower();
                LoginHelper loginHelper = new LoginHelper();
                if (lowerNoQueryPath.EndsWith(".aspx"))
                    HttpContext.Current.Response.Redirect("Login.aspx");
//转向的无权页面一定要和 authorize 方法结合起来，避免形成死循环
                else if (lowerNoQueryPath.EndsWith(".asmx"))
                {
                    //可引发 soap 异常报告无权登录
```

```
        }
    }
```

在将 SecurityProvider 组件设计 OK 后，就需要选择在何时调用该组件进行管控了。asp.net 的 HttpModule 机制刚好提供了这样的管控时机。做一个 HttpModule，捕捉一个真正的 Request 程式执行之前的事件，如 PreRequestHandlerExecute 中调用 SecurityProvider.Valid 方法进行管控。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Web;

namespace WebSecurity
{
    public class SecurityModule : IHttpModule
    {
        //安全管控对象
        SecurityProvider _provider;

        public SecurityModule()
        {
            _provider = new SecurityProvider();
        }

        public void Init(HttpApplication application)
        {
            //捕获 PreRequest 事件
            application.PreRequestHandlerExecute += new EventHandler
                        (application_PreRequestHandlerExecute);
        }

        void application_PreRequestHandlerExecute(object sender,
EventArgs e)
        {
            //每次请求都进行权限管控
            _provider.Valid();
        }
        public void Dispose()
        {
        }
    }
}
```

然后在 Web.Config 中配置，完成权限管控。

```
    <httpModules>
        <add name="WebSecurity" type="WebSecurity.SecurityModule,
WebSecurity"/>
    </httpModules>
```

当然如果简单点，也可以在 Global.asax 中捕获事件进行 SecurityProvider.Valid 验证。

当安全组件建立后，每个系统的开发都不再需要考虑权限管控，而是只要实现系统本身的业务逻辑和功能。通过后期的权限管控配置，系统功能与程序对应关系的建立，权限分配控件的灵活使用，最终脱离“苦海”，不再挣扎。

SEO——我们是不是走错了路?

作者：丁学[http://www.cnblogs.com/dingxue]

静态化

这是一个跨越了太长时间的话题，很多人一聊到 SEO，就说“静态化很重要”。其实怎么说呢，我觉得静态化更重要的是用来解决系统负载和运行效率的问题，而并非 SEO，相信最初发明“静态化”的人也是出于解决系统负载的目的。之所以有人说静态化利于 SEO，很可能源自很古老的一篇出自 google 的文章，大意就是说搜索引擎更喜欢收录“静态页面”（.htm or .html）。随着技术发展，有人就针对这一条，开始了 URL Rewirte 之旅。但是同前面静态化一样，我相信最初发明 URL Rewrite 的人，目的应该是让网站拥有更容易记住、对用户更友好的 URL。到了今天，我相信搜索引擎不至于傻到动态地址无法收录，只要是对用户有用。作为一个程序员，我十分相信搜索引擎的开发者一定早已解决了这些极其弱智的问题。不管历史曾经怎样，也不管搜索引擎的开发者是否和我这个程序员是一样的想法。Google 已经明确的说了：“请不要将一个动态网址改换格式，使其看起来是静态的。”我们是不是应该把更多的精力交给用户和服务器？

关键词

SEOer 们的主战场无处不在，主占领地：title、meta 和 h1 ~ h6，另外分散于页面的任何位置，数量不定，密度 1% ~ 5%不等。我们每天都在猜，<title>中 google 会承认多长，顺序对关键词有什么影响。可是，google 在做什么？google 在猜哪些 title 是对用户有用的，哪些是能够真正表述页面内容的，我们为什么不直接做成这样的？Description 现在作用已经越来越小了，不过我们应该明白，这个东西本来是搜索引擎索引了网页后显示给用户的那一段简介，随着我们越来越多地利用这里做 SEO，搜索引擎已经严重不信任 Description，于是好端端一个地方被浪费了；语义化的 WEB 标准被我们用来

做 SEO，于是搜索引擎开始努力地想区别哪些是真的哪些是假的。所有这一切构成一个恶性的循环，我们与搜索引擎互相不信任，浪费了资源，也流失了客户。我们是不是应该回过头来，让 title 是 title、description 是 description，内容是内容？不欺骗用户，是不是我们会轻松一些？1 个谎言需要 100 个谎言去圆，何必呢？

反向链接

就是这个造就了一个群体，无数的 SEOer 们没日没夜奋战在 QQ 群、专业论坛等战场一线。为了反向链接，我们去和任何有价值的网站去交换链接，直到我们再没有地方去放交换来的链接。于是我们去百度贴吧发帖，去各大论坛发帖，但是依然不够，于是诞生了一个东西叫“链接联盟”。与此同时，搜索引擎也发觉了，于是就有了“封杀”、“屏蔽”等，多少站长捶胸顿足，多少站长夜不能寐。如果我们把这些精力放在改进程序和充实内容上，结果会是什么样子？我们在做什么？我们是不是忘记了最初的目的？让链接回归本色，让“相关”这个词真实地呈现在用户面前，是不是更好一些？身体上、心理上是不是都会轻松一些？让网络回归网络，让网络不再像个战场，让网络为人所用，再不要让网络把人逼得发疯……

Email – 新闻 – 论坛 – 博客

曾经多么美好的 4 个词，引领了互联网十余载一步步走来，我们的网络如此美好。自从有了 SEO 这个词，所有的事情都出了那么一点点差错，垃圾邮件、灌水机、软文、枪手……我们花更多的时间，我们花更多的努力，我们花更多的预算，生活变得紧张而忙碌，我们要得到什么？我们为什么这么做？如果只有一个人这样做，效果或许不错，但是当大家都这样做的时候呢？是不是世界本来就应该简单一些？我们拿我们希望的内容去填充互联网，同时也在为自己查找资料设置着障碍，一边骂网络垃圾太多，一边自己疯狂地制造垃圾。回来想想搜索引擎，它们想做什么？他们在想用户！那么用户在想什么？他们想最快得到“有用”的信息，那我们为什么不把写软文的时间用来写一点对用户有用的内容呢？为什么不把发垃圾邮件用的服务器用来提高哪怕 0.01 秒的浏览速度呢？我们走了一条错误的路，因为大家都不肯做第一个撤退的人，所以大家都往前冲，明知道前面是悬崖，也义无反顾……

忘掉 SEO 吧

世界上本来就不应该存在这样一件事。想想搜索引擎们每天在做什么？他们希望让用户看到最想看的东西，希望让用户用到最好用的东西，为什么我们不直接努力让用户看起来更有兴趣一些，用起来更方便一些呢？

是跟在搜索引擎那些五花八门又极其 BT 的算法后面奔跑？还是直接奔向目标，让搜索引擎跟着你跑？自己的路，还要是自己选择……

JavaScript 变量作用域及可访问性的探讨

作者：3zfp [http://www.cnblogs.com/3zfp]

每一种语言都有变量的概念，变量是用来存储信息的一个元素。比如下面这个函数：

```
function Student(name,age,from)
{
    this.name = name;
    this.age = age;
    this.from = from;
    this.ToString = function()
    {
        return "my information is name: "+this.name+",age : "+this.age+",
from :" +this.from;
    }
}
```

Student 类有 3 个变量，分别为 name（名字）、age（年龄）和 from（籍贯），这三个变量构成了描述一个对象的信息。当然，这里还有一个方法用来返回 Student 的信息。

但是，我们是不是定义了一个变量，它就能一直存在着，并且还有可能在任何地方都能被访问、使用、直到被销毁？仔细想想，上面的需求是比较过分的，因为某些变量在某个功能实现后就不再利用了，但如果这个变量还存在的话，就占用了系统资源。

于是变量的及时和按需求地销毁成了一个可探讨的话题。

就本人所接触过的来讲，js 中支持如下几种类型的变量：局部变量、类变量、私有变量、实例变量、静态变量和全局变量。接下来我们就一一探讨。

局部变量

局部变量一般指在{ }范围内有效的变量，也就是语句块内有效的变量，如：

```
function foo(flag)
```

```
{
 var sum = 0;
 if(flag == true)
 {
     var index;
  for(index=0;index<10;index++)
  {
        sum +=index;
 }
 }
 document.write("index is :"+index+"<br>");
 return sum;
}
//document.write("sum is :" +sum+"<br>");
document.write("result is :"+foo(true)+"<br>");
```

该代码执行后输出的结果为“index is :undefined”和“result is :0”。我们可以看到希望输出的 index 变量的值为 undefined，也就是未定义，index 变量在 if 语句块结束后被销毁了。那么“sum”变量呢？这个变量在 foo()函数段执行完毕后被销毁了，如果您去掉我注释的那条语句再执行，系统将报错。值得注意的是，如果我把上面的 foo()函数改成如下：

```
function foo(flag)
{
 var sum = 0;
 for(var index=0;index<10;index++)
 {
     sum +=index;
 }
 document.write("index is :"+index+"<br>");
 return sum;
}
```

则可以输出 index 值("index is :10")，这个是 JS 和其他语言不同的地方，因为 index 是在 for 循环的{}外面定义的，因此其作用范围在 foo()函数使用完毕后才销毁。

类变量

类变量实际上就是类的一个属性或字段或一个方法，该变量在该类的一个实例对象被销毁后自动销毁，比如我们开始时举的 Student 类。

私有变量

私有变量指的是某个类自己内部使用的一个属性，外部无法调用，其定义是用 var 来声明的。注意如果不用 var 来声明，该变量将是全局变量（我们下面将会讨论），如：

```
function Student(name,age,from)
{
   this.name = FormatIt(name);
   this.age = age;
 this.from = from;
 var origName = name;
 var FormatIt = function(name)
 {
     return name.substr(0,5);
 }
 this.ToString = function()
 {
  return "my information is name: "+origName+",age : "+this.age+",
from :" +this.from;
 }
```

这里，我们分别定义了 origName 和 FormatIt()两个私有变量（按面向对象的解释，应该用类的属性来称呼）。

我们把这种情况下的方法也称为变量，因为该情况下的变量是个 function 类型的变量，而 function 也属于 Object 类的继承类。在这种情形下，如果我们定义了 var zfp = new Student("3zfp",100,"ShenZhen")，但无法通过 zfp.origName 和 zfp.FormatIt()方式来访问这两个变量。

注意以下 3 点。

1. 私有变量是不能用 this 来指示的。

2. 私有方法类型的变量的调用必须在该方法声明后，如我们将 Student 类改造如下：

```
function Student(name,age,from)
{
  var origName = name;
 this.name = FormatName(name);
 this.age = age;
 this.from = from;
```

```
    var FormatName = function(name)
    {
     return name+".china";
    }
    this.ToString = function()
    {
     return "my information is name: "+origName+",age : "+this.age+",
from :" +this.from;
    }
   }
   var zfp = new Student("3zfp",100,"ShenZhen");
```

代码执行后，将会报“找不到对象”的错误，意思是FormatName()未定义。

3. 私有方法无法访问 this 指示的变量（公开变量），如下：

```
function Student(basicinfo)
{
 this.basicInfo = basicinfo;
 var FormatInfo = function()
 {
  this.basicInfo.name = this.basicInfo.name+".china";
 }
 FormatInfo();
 }
 function BasicInfo(name,age,from)
 {
  this.name = name;
  this.age = age;
  this.from = from;
 }
 var zfp = new Student(new BasicInfo("3zfp",100,"ShenZhen"));
```

执行代码后，系统将会提示“this.basicInfo 为空或不是对象”的错误。

基本结论是，私有方法只能访问私有属性，私有属性在声明并赋值后可以在类的任何地方访问。

实例变量

实例变量即某个实例对象所拥有的变量，如：

```
function BasicInfo(name,age,from)
{
```

```
 this.name = name;
 this.age = age;
 this.from = from;
}
var basicA = new BasicInfo("3zfp",100,"ShenZhen");
basicA.generalInfo = "is 3zfp owned object";
document.write("basicA's generalInfo is : "+ basicA.generalInfo+"<br>");
var basicB = new BasicInfo("zfp",100,"ShenZhen");
document.write("basicB's generalInfo is : "+ basicB.generalInfo+"<br>");
```

执行该代码后，可以看到如下结果：

```
basicA's generalInfo is : is 3zfp owned object
basicB's generalInfo is : undefined
```

静态变量

静态变量即为某个类所拥有的属性，通过 类名+"."+静态变量名的方式访问该属性。下面进行清晰地解释，执行下面代码：

```
function BasicInfo(name,age,from)
{
 this.name = name;
 this.age = age;
 this.from = from;
}
BasicInfo.generalInfo = "is 3zfp owned object";
var basic = new BasicInfo("zfp",100,"ShenZhen");
document.write(basic.generalInfo+"<br>");
document.write(BasicInfo.generalInfo+"<br>");
BasicInfo.generalInfo = "info is changed";
document.write(BasicInfo.generalInfo+"<br>");
```

将会得到如下结果：

```
undefined
is 3zfp owned object
info is changed
```

注意以下几点：

- **以“类名+"."+静态变量名”的方式来声明一个静态变量。**
- **静态变量并不属于类的某个实例对象所独有的属性，为对象的共享。**
- **能以“实例对象名+"."+静态变量名”来访问。**

全局变量

全局变量即整个系统运行期间有效访问控制的变量，通常是在 JS 代码开头定义，如：

```
var copyright = "3zfp owned";
var foo =function()
{
 window.alert(copyright);
}
```

注意以下几点。

1. 如果变量不用 var 来声明，则其被视为全局变量，如：

```
var copyright = "3zfp owned";
var foo =function(fooInfo)
{
   _foo = fooInfo;
 document.write(copyright+"<br>");
}
new foo("foo test");
document.write(_foo+"<br>");
```

执行代码，将得到如下结果：

```
3zfp owned
foo test
```

这里又有一个注意的地方，function 是编译期对象，也就是说_foo 这个全局变量要在 foo 对象被实例化后才能被初始化，也就是说如果将

```
new foo();
document.write(_foo+"<br>");
```

对调成

```
document.write(_foo+"<br>");
new foo();
```

系统将提示 "_foo 未定义"。

2. 如果定义了一个和全局变量同名的局部变量属性，如下：

```
var copyright = "3zfp owned";
var foo =function(fooInfo)
```

```
{
 var copyright = fooInfo; //同名变量
 this.showInfo = function()
 {
  document.write(copyright+"<br>");
 }
}
new foo("foo test").showInfo();
document.write(copyright+"<br>");
```

执行代码，将得到如下结果：

```
foo test
3zfp owned
```

原因是 function 在编译期间完成变量的定义，也就是 foo 内部的 copyright 的定义是在编译期间完成的，其作用域只在 foo 对象内有效，而与外部定义的全局变量 copyright 无关。

JavaScript 中的 this 关键字

作者：LongWay[http://www.cnblogs.com/LongWay]

“In JavaScript this always refers to the ‘owner’ of the function we're executing, or rather, to the object that a function is a method of.”

这是来自 http://www.quirksmode.org/js/this.html 这篇文章里对 this 的定义，直接看定义似乎什么也不知道，下面通过实例来说明各种情况下 this 所指代的对象以及原理。

关于 JS 中的 this 关键字的文章已经不少了，我看过几篇，我写这篇文章的目的是从实例中分析出 this 的工作原理，希望对大家有所帮助。

一、基本的

```
function doSomething(){
    alert(this.id);
}
alert(window.doSomething);//证明了 doSomething 是属于 window 的
doSomething();//undefined
window.onload = function(){
    document.getElementById("div2").onclick = doSomething;//div2
    document.getElementById("div3").onclick =
function(){doSomething();}//undefined
}
```

1. doSomething 函数

```
function doSomething(){
    alert(this.id);
}
```

这个函数是全局函数，这种全局函数实际上是属于 window 的（可以通过 window.doSomething 来访问），如果直接调用，那么根据“this always refers to the ‘owner’ of the function we're executing”，函数中的 this 就是 window，但是 window 没有 id 属性，所以显示“undefined”。

2. 在 HTML 元素中这样调用

```
<div id="div1" onclick="doSomething();">div1</div>
```

这时也会显示“undefined”，这就相当于如下代码：

```
document.getElementById("div1").onclick = function(){doSomething();}
```

当点击 div1 时，调用属于 window 的 doSomething 函数，所以也是显示“undefined”。

3. 通过 js 来绑定事件，在 div2 载入过后

```
document.getElementById("div2").onclick = doSomething;
```

当点击 div2 时显示“div2”，因为给 div2 的 onclick 赋值是将 doSomething 复制了一次，这时复制的这个函数是属于 div2 的了，跟属于 window 的 doSomething 没有任何关系。点击 div2 时，就会触发属于 div2 的 doSomething，这里的 this 就是指 div2。

二、attachEvent 和 addEventListener

attachEvent 是在 IE 中绑定事件的方法，会将相应函数复制到全局（即响应函数的 owner 为 window）。但是在 DOM 标准中，addEventListener 绑定的事件复制的响应函数的 owner 为事件所绑定的对象。

```
function doSomething(){
    alert(this.id);
    alert(this == window);
}
window.onload = function(){
    var div1 = document.getElementById("div1");
    if(div1.attachEvent){
        div1.attachEvent("onclick",doSomething);
        document.body.appendChild(document.createTextNode
("attachEvent"));
    }else if(div1.addEventListener){
        div1.addEventListener("click",doSomething,false);
        document.body.appendChild(document.createTextNode
("addEventListener"));
    }else{
        div.onclick = doSomething;
    }
}
```

函数 doSomething（1）使用 attachEvent 绑定到 div1 的 click 事件上，doSometing 会被复制到 window，这时 doSomething 里面的 this 指的是 window，点击 div1 时会显示“undefined”和“true”。（2）使用 addEventListener 绑定 div1 的 click 事件，这时将 doSomething 复制，这个复制过后的函数是属于 div1 的，所以点击 div1 时会显示“div1”和“false”。

http://www.quirksmode.org/js/this.html 里认为 attachEvent 只是使用了函数的引用，看如下代码：

```
var obj = new Object();
obj.color = "black";
obj.showColor = function(){
    alert(this.color);
    alert(this == window);
}
obj.showColor();

var div1 = document.getElementById("div1");
div1.attachEvent("onclick",obj.showColor);
```

此时单击 div1 的时候，会显示“undefined”和“true”。如果 attachEvent 仅仅是引用 obj.showColor 的话，那么 this 指的还是 obj，但是实际上这里的 this 指的是 window，所以我认为这里不是引用，而是复制到全局。

三、关于对象冒充的继承方式

1. new 与不 new 的区别

看看如下 function：

```
function ClassA(sColor){
    this.color = sColor;
    this.sayColor = function(){
        alert(this.color);
    }
}
```

这是一个类还是一个函数？随你而定！

如果你认为这是一个函数，那么我们可以这样来调用它：

```
ClassA("red");
```

“red”是传递的一个参数，ClassA 中的 this 当然就是指 window 了，所以

现在 window 有了 color 属性和 sayColor 方法，并且 color 有“red”这个值。

这时调用 sayColor 或者 window.sayColor 都可以显示“red”。

```
window.sayColor();
```

如果你认为这是一个类，那么我们应该这样使用它：

```
var obj = new ClassA("red");
```

new 这个关键词的出现让上面这一句代码增加了不少内容：首先创建一个 Object 实例，然后将 ClassA 中的 this 指向创建的这个 Object 中，最后返回这个 Object，并赋值给 obj。所以我们可以说 this 指向的是 obj，obj 拥有了 color 属性和 sayColor 方法，并且 color 属性值为“red”。

2. 函数的 owener

```
function showId(){
    alert(this.id);
}
window.onload = function(){
    var div1 = document.getElementById("div1");
    div1.onclick = showId;
    div1.show = showId;
    div1.show();

    var obj = new Object();
    obj.id = "obj";
    obj.show = showId;
    obj.show();
}
```

我们可以将 showId 这个函数赋值给 click 事件，也可以赋值给任何一个对象的任何一个属性，这时也会拷贝 showId 这个方法。所以我们在调用 div1.show 方法时，this 是指向 div1 的，在调用 obj.show 时，this 指向的是 obj。

3. 对象冒充的原理

下面的代码是通过对象冒充方法实现的继承。

```
function ClassA(sColor){
    this.color = sColor;
    this.sayColor = function(){
        alert(this.color);
    }
```

```
}
function ClassB(sColor,sName){
    this.newMethod = ClassA;
    this.newMethod(sColor);
    delete this.newMethod;

    this.name = sName;
    this.sayName = function(){
        alert(this.name);
    }
}
var objB = new ClassB("color of objB","name of objB");
objB.sayColor();
```

objB 是 ClassB 的一个实例，objB 是如何拥有 color 属性和 sayColor 方法的呢?

首先从实例化的代码看起：

```
var objB = new ClassB("color of objB","name of objB");
```

这里 ClassB 是个类，ClassB 中的 this 当然就是指的 objB 这个对象；在 ClassB 中，前三行代码会用到 ClassA，这时就把 ClassA 看作一个函数，而不是类。

我们如果直接调用 ClassA 这个函数，那么很显然，ClassA 中的 this 指的就是 window 对象了，所以我们先将 ClassA 拷贝到 objB 的 newMethod 这个属性中（this.newMethod = ClassA），然后再调用 this.newMethod，这时这个方法的 owener 明显已经成了 this，而 ClassB 中的 this 当前指的是 objB，所以此时 ClassA 中（严格说是 newMethod 中，因为这是复制过的，跟 ClassA 已经是两个方法了）的 this 指的就是 objB。这样再通过 newMethod 的调用，就给 objB 赋值了 color 属性和 sayColor 方法。用 call 和 apply 方法来实现继承实际上也是这个原理，call 和 apply 可以看作是改变方法的 owner 的方法，而这里 ClassB 中的前三句代码也就是起这个作用的。

四、prototype1.6 中的 Class.create

prototype1.6 中的 Class.create 方法大致如下：

```
var Class = {
  create: function() {
    //
```

```
    function klass() {
      this.initialize.apply(this, arguments);
    }
    //
    for (var i = 0; i < properties.length; i++)
      klass.addMethods(properties[i]);
    //
    return klass;
  }
};
```

在使用的时候是这样的：

```
var Person = Class.create({
    initialize:function(name){
        this.name = name;
    },
    say:function(message){
        alert(this.name + ":" + message);
    }
});

var aPerson = new Person("name1");
aPerson.say("hello1");
```

Person 实际上是通过 Class.create 这个方法所返回的 klass（klass 是 Class.create 中的局部变量，是一个 function），Class.create 所传递的参数（initialize 方法和 say 方法）传递到 create 方法中的 properties 数组中，并且通过 addMethods 方法让 klass 的 prototype 拥有这些方法。最关键的地方也是最难以理解的地方是：klass 中的 this 究竟指的是什么。仔细想一想就不难得到答案，Person 实际上就是 klass，而我们在实例化 Person 对象的时候，是用了 new 关键词的：

```
var aPerson = new Person("name1");
```

这就等价于

```
var aPerson = new klass("name1");
```

虽然 klass 在外面不能被访问到，但是这样能很轻易地说明问题，klass 是一个类而不是简单的一个函数（我们看作如此，因为用了 new 关键字），那么 klass 中的 this 就指的是声明的实例，在这里就是 aPerson，aPerson 通过 klass 的 prototype 能够拥有 initialize 方法和 say 方法。在 new 的过程中，也会执行 klass 中的代码，所以 initialize 在实例化的时候会执行，即构造函数。（在 klass

里两个 this 都指的是 aPerson，为什么还要通过 apply 调用一次呢？这主要是为了传递构造函数的参数，用 apply 方法可以将数目不定的多个参数通过数组方便地传到 initialize 方法中去。）

五、再分析几个例子

从别的文章里看到的例子，我在这里分析一下。

1. 运行如下代码：

```
function OuterFoo(){
    this.Name = 'Outer Name';

    function InnerFoo(){
        var Name = 'Inner Name';
        alert(Name + ', ' + this.Name);
    }
    return InnerFoo;
}
OuterFoo()();
```

所显示的结果是“Inner Name, Outer Name”。

OuterFoo 是一个函数（而不是类），那么第一句 this.Name = 'Outer Name'; 中的 this 指的是 window 对象，所以 OuterFoo（）过后 window.Name = 'Outer Name'，并且将 InnerFoo 返回，此时 InnerFoo 同样是一个函数（不是类）。执行 InnerFoo 的时候，this 同样指 window，所以 InnerFoo 中的 this.Name 的值为“Outer Name”（window.Name 充当了一个中转站的角色，让 OuterFoo 能够向 InnerFoo 传递“Outer Name”这个值），而 Name 的值即为局部变量“Inner Name”。

2. 运行如下代码：

```
function JSClass(){

    this.m_Text = 'division element';
    this.m_Element = document.createElement('div');
    this.m_Element.innerHTML = this.m_Text;

    if(this.m_Element.attachEvent)
        this.m_Element.attachEvent('onclick', this.ToString);
    else if(this.m_Element.addEventListener)
        this.m_Element.addEventListener('click', this.ToString,false);
```

```
    else
        this.m_Element.onclick = this.ToString;
}

JSClass.prototype.Render = function(){
    document.body.appendChild(this.m_Element);
}

JSClass.prototype.ToString = function(){
    alert(this.m_Text);
    alert(this == window);
}

window.onload = function(){
    var jc = new JSClass();
    jc.Render();
    jc.ToString();
}
```

单击“division element”会显示“undefined”，在IE下还要显示“true”，其他浏览器中还要显示“false”。

实例声明和调用实例方法都没什么可说的，元素的click事件绑定到了一个实例的方法，那么通过addEventListener绑定到的方法是复制后的，所以this指的是html元素，这个元素没有m_Text属性（m_Text属性是属于JSClass的实例的，即属于jc的），所以单击元素显示undefined，attachEvent绑定的事件会将函数复制到全局，此时this指的是window对象，点击元素也会显示“undefined”。只有在调用jc.ToString()方法时，this指的是jc这个对象，因为jc拥有m_Text，所以能够显示“division element”。

六、总结

怎样在一个代码环境中快速找到this所指的对象呢？我想要注意以下3个方面：

1. 要清楚地知道对于函数的每一步操作是复制还是引用（调用）。
2. 要清楚地知道函数的拥有者（owner）是什么。
3. 对于一个function，要搞清楚我们是把它当做函数使用还是在当做类使用。

你不知道的 JavaScript - “this”

作者：棕熊[http://www.cnblogs.com/ruxpinsp1]

JavaScript 里的 this 到底指的是什么？很多人都会告诉你 this 指的是当前对象。这样理解对吗？在大多数情况下确实没错。比如我们经常会在网页上写这样的 JavaScript：

```
<input type="submit" value="提交" onclick="this.value='正在提交数据'" />
```

这里的 this 显然指的是当前对象，即这个提交按钮。通常我们使用 this 的情况都与此类似。但是有什么情况不是这样的呢？

大家看看这个例子：

```
var foo = function() {
    console.log(this);
}
foo();
new foo();
```

比较一下 foo() 和 new foo() 的运行结果，你会发现前者 this 指向的并非 foo 本身，而是当前页面的 window 对象，而后者才真正地指向 foo。这是为什么呢？

其实这牵涉到 JavaScript 的一条重要特性，就是所谓的闭包。闭包这个概念也不复杂，但也不是简单到能用一两句话说清。我会在以后的文章中深入探讨这个 JavaScript 最重要的特性。现在我要告诉大家的是，因为闭包的存在，JavaScript 中的作用域变得相当重要。

所谓的作用域，简单说，就是创建一个函数是在什么环境下创建的。而 this 变量的值，如果没有指定的话，就是函数当前的作用域。

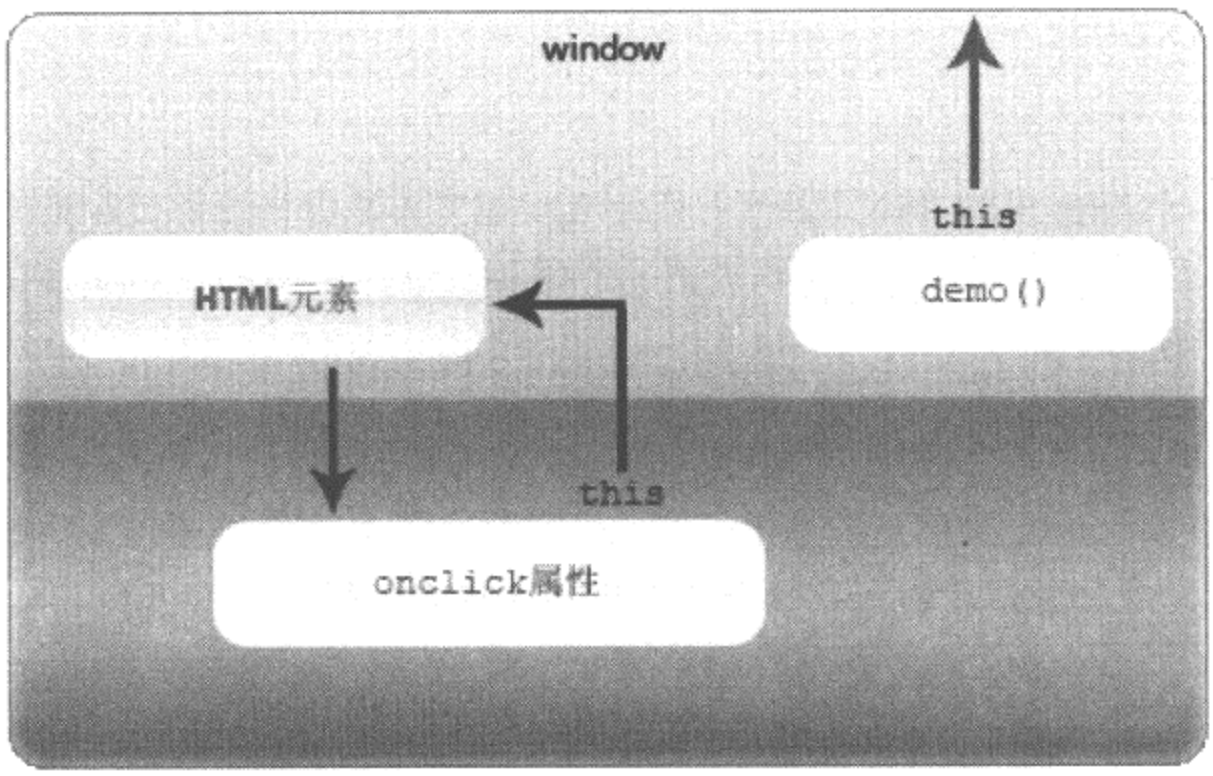
在前面的例子里，foo() 函数属于全局作用域（这里就是 window 对象），所以 this 的值是当前的 window 对象。而 new foo()这样的形式，其实是创建了一个 foo()的副本，并在这个副本上进行操作，所以这里的 this 就是 foo()的这个副本。

这样讲可能有点抽象，大家来看个实际的例子：

```
<input type="button" id="aButton" value="demo" onclick="" />
<script type="text/JavaScript">
function demo() {    this.value = Math.random();
```

```
}
</script>
```

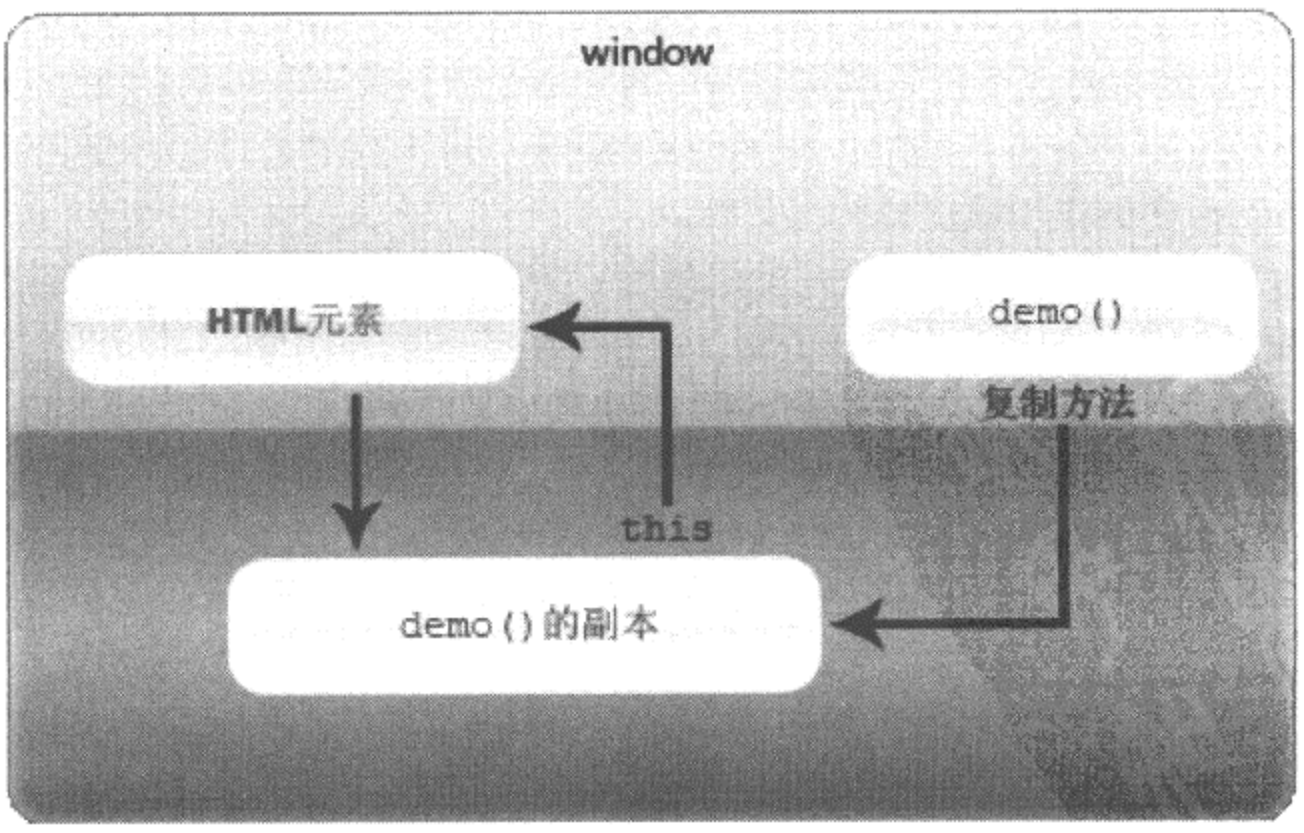
如果直接调用 demo() 函数，程序就会报错，因为 demo 函数是在 window 对象中定义的，所以 demo 的拥有者（作用域）是 window，demo 的 this 也是 window，而 window 是没有 value 属性的，所以就报错了。



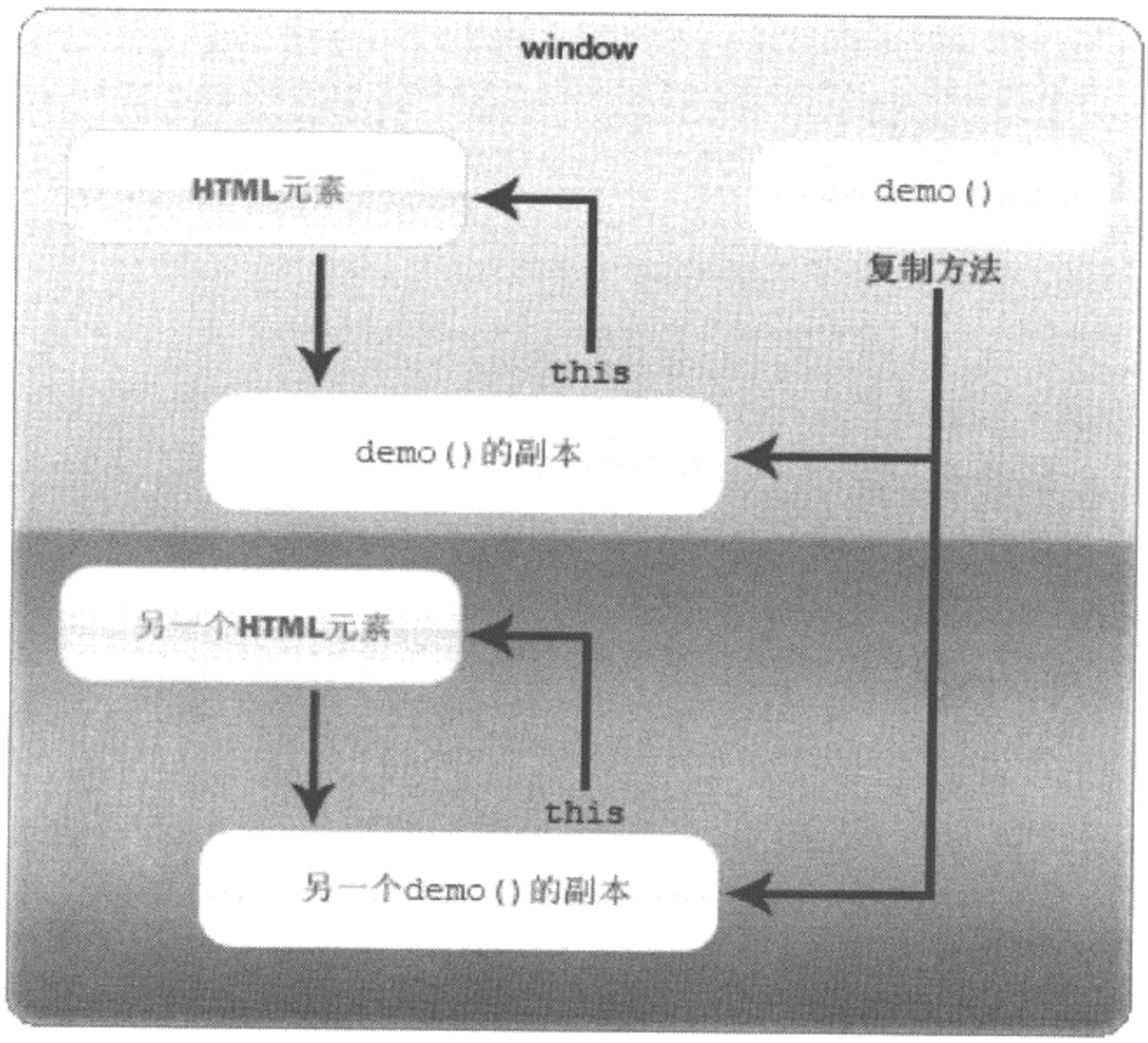
如果我们通过创建副本的方式，将这个函数的**副本**添加到一个 HTML 元素中，那么它的所有者就成了这个元素，this 也指代了这个元素：

```
document.getElementById("aButton").onclick = demo;
```

这样就将 aButton 的 onlick 属性设置为 demo()的一个副本，this 也指向了 aButton。



甚至可以为多个不同的 HTML 元素创建不同的函数副本。每个副本的拥有者都是相对应的 HTML 元素，各自的 this 也都指向他们的拥有者，不会造成混乱。



但是如果这样定义某个元素的 onlick 事件：

```
<input type="button" id="aButton" value="demo" onclick="demo()" />
```

单击这个 button 之后，你会发现程序又会报错了——this 又指向了 window!

其实，这种方法并没有为程序创建一个函数，而只是引用了这个函数。

具体看一下区别吧，使用创建函数副本的方法：

```
<input type="button" id="aButton" value="demo" />
<script type="text/JavaScript">
var button = document.getElementById("aButton");
function demo() {
    this.value = Math.random();
}
button.onclick= demo;
alert(button.onclick);
```

```
</script>
```

得到的输出是：

```
function demo() {
    this.value = Math.random();
}
```

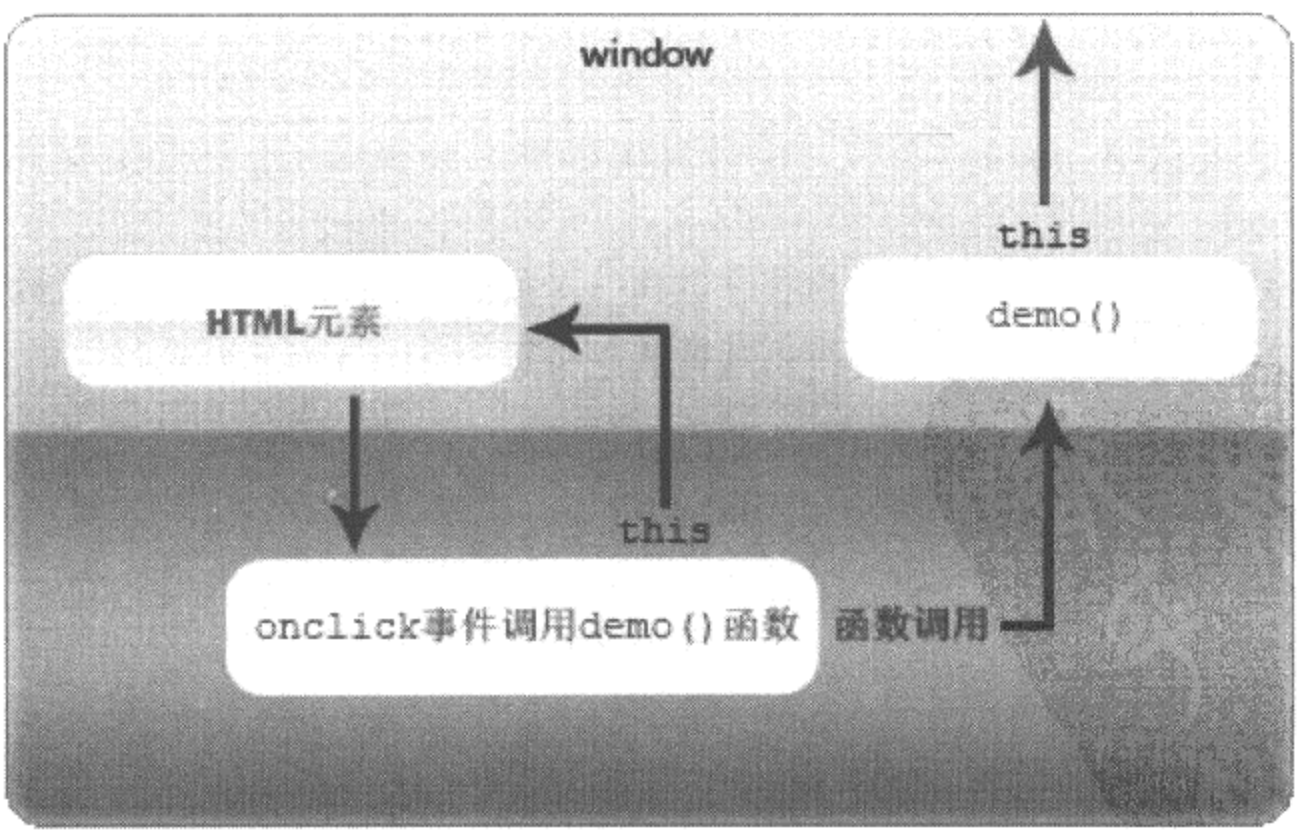
使用函数引用的方法：

```
<input type="button" id="aButton" value="demo" onclick="demo()" />
<script type="text/JavaScript">
var button = document.getElementById("aButton");
function demo() {
    this.value = Math.random();
}
alert(button.onclick);
</script>
```

得到的输出是：

```
function onclick() {
    demo();
}
```

这样就能看出区别了。函数引用的方式中，onclick 事件只是直接调用 demo()函数，而 demo()函数的作用域仍旧是 window 对象，所以 this 仍然指向 window。



这样就又引出了一个问题：既然函数副本这么好用，为什么还需要函数引用的方法呢？答案是性能。每新建一个函数的副本，程序就会为这个函数副本分配一定的内存。而实际应用中，大多数函数并不一定会被调用，于是这部分内存就被白白浪费了。而使用函数引用的方式，程序就只会给函数的本体分配内存，而引用只分配指针，这样效率就高很多。

程序员么，节约为主，所以我们来看一个更好的解决方案：

```
<script type="text/JavaScript">
function demo(obj)  {
    obj.value = Math.random();
}
</script>
<input type="button" value="demo" onclick="demo(this)" />
<input type="button" value="demo" onclick="demo(this)" />
<input type="button" value="demo" onclick="demo(this)" />
```

这样，效率和需求就能兼顾了。

最后再多讲一句：在前面的文章里，我特别强调了“如果没有指定 this 的话”。其实 **this** 是可以指定的。Function 对象有两个方法：call()和 apply()。这两个方法都支持指定函数中的 this。可以去查一下 JavaScript 的手册，看看这两个函数是干什么用的。我们经常用的 new foo() 可以用以下这段伪代码来描述：

```
function new (somefunction)  {
var args = [].slice.call(arguments, 1);
    somefunction.prototype.constructor = somefunction;
    somefunction.apply(somefunction.prototype, args);
    return somefunction.prototype;
}
```

现在明白在本文开头的第一个例子里，new foo() 的 this 为什么是 foo 了吧。

JavaScript 代码压缩、加密算法分析及工具实现

作者：在路上... [http://www.cnblogs.com/midea0978]

简介

现在网上很多 JavaScript 代码都进行了压缩，代码变得不可直接阅读，也相当于一种简单的加密。本文对一种典型的压缩算法进行分析，介绍如何解密源代码以及重新实现压缩算法。

一段经压缩后的代码如下：

```
eval(function(E,I,A,D,J,K,L,H){function C(A){return
A<62?String.fromCharCode(A+=A<26?65:A<52?71:-4):A<63?'_':A<64?'$':C(A>
>6)+C(A&63)}while(
A>0)K[C(D--)]=I[--A];function N(A){return
K[A]==L[A]?A:K[A]}if(''.replace(/^/,String)){var
M=E.match(J),B=M[0],F=E.split(J),G=0;if(E.indexOf(F[0]))F=[''].conc
at(F);do{H[A++]=
F[G++];H[A++]=N(B)}while(B=M[G]);H[A++]=F[G]||'';return H.join
('')}return E.replace
(J,N)}('Bl Bm=Bn;Bo(Bl
Bp=Bq;Bp<Bn;Bp++){   Br.Bs(Bm+Bp+"<Bt>");}','var|index|100|for|
a|0|document|write|br'.split('|'),9,109,/[\w\$]+/g, {}, {}, []))
```

代码解密

对于这类代码，无非是把 JS 程序采用某种算法进行压缩，然后自行用提供的函数还原，采用 eval(SCRIPT)的方式执行来完成调用。还原的方法很简单，把前面的 eval（和后面的）去掉，然后把代码显示出来就完成了，例如下面的页面就可以实现代码的还原：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
 <HTML>
 <HEAD>
```

```
        <TITLE> 代码还原 </TITLE>
    </HEAD>

    <BODY>
    <TEXTAREA NAME="tx1" ROWS="10" COLS="100"></TEXTAREA>
    <SCRIPT LANGUAGE="JavaScript">
    document.all.tx1.value =function(E,I,A,D,J,K,L,H){function
C(A){return
    A<62?String.fromCharCode(A+=A<26?65:A<52?71:-4):A<63?'_':A<64?'$':C(A
>>6)+C(A&63)}while(A>0)K[C(D--)]=I[--A];function N(A){return K[A]==L[A]?
A:K[A]}if(''.replace(/^/,String)){varM=E.match(J),B=M[0],F=E.split(J),G=0;
if(E.indexOf(F[0]))F=[''].concat(F);do{H[A++]=F[G++];H[A++]=N(B)}while(B=M[
G]);H[A++]=F[G]||'';return H.join('')}return E.replace(J,N)}('Bl Bm=Bn;Bo(Bl
Bp=Bq;Bp<Bn;Bp++){    Br.Bs(Bm+Bp+"<Bt>");}','var|index|100|for|a|0|document
|write|br'.split('|'),9,109,/[\w\$]+/g, {}, {}, [])
    </SCRIPT>
    </BODY>
    </HTML>
```

通过上面方式运行，就可以在文本框中看到代码了，实际的代码是：

```
var index=100;for(var a=0;a<100;a++){ document.write(index+a+"<br>");}
```

很简单，不是吗？

算法研究

由于代码全部在一行中，不便于阅读，可以通过格式化软件进行格式化，本文使用 Intellij IDEA 进行格式化，代码如下：

```
eval(function(E, I, A, D, J, K, L, H) {
    function C(A) {
        return A < 62 ? String.fromCharCode(A += A < 26 ? 65 : A <
52 ? 71 : -4) : A < 63 ? '_' : A < 64 ? '$' : C(A >> 6) + C(A & 63)
    }
    while (A > 0)K[C(D--)] = I[--A];
    function N(A) {
        return K[A] == L[A] ? A : K[A]
    }
    if (''.replace(/^/, String)) {
        var M = E.match(J),B = M[0],F = E.split(J),G = 0;
        if (E.indexOf(F[0]))F = [''].concat(F);
        do{
```

```
            H[A++] = F[G++];
            H[A++] = N(B)
        } while (B = M[G]);
        H[A++] = F[G] || '';
        return H.join('')
    }
    return E.replace(J, N)
}('Bl Bm=Bn;Bo(Bl Bp=Bq;Bp<Bn;Bp++){Br.Bs(Bm+Bp+"<Bt>");}','
var|index|100|
for|a|0|document|write|br'.split('|'), 9, 109, /[\w\$]+/g, {}, {}, []))
```

Step 1：首先我们可以看出这段代码函数定义与调用是合并在一起的，因此可以如下分解（不再考虑 eval）：

```
//E:加密压缩后的 script 信息
//I:字符串数组,可以理解为解密需要字典
//A:int 9
//D:int 109
//J:regexpr 正则表达式
//K:object
//L:object
//H:array
function decode(E, I, A, D, J, K, L, H) {
    function C(A) {
        return A < 62 ? String.fromCharCode(A += A < 26 ? 65 : A <
52 ? 71 : -4) : A < 63 ? '_' : A < 64 ? '$' : C(A >> 6) + C(A & 63)
    }
    while (A > 0)K[C(D--)] = I[--A];
    function N(A) {
        return K[A] == L[A] ? A : K[A]
    }
    if (''.replace(/^/, String)) {
        var M = E.match(J),B = M[0],F = E.split(J),G = 0;
        if (E.indexOf(F[0]))F = [''].concat(F);
        do{
            H[A++] = F[G++];
            H[A++] = N(B)
        } while (B = M[G]);
        H[A++] = F[G] || '';
        return H.join('')
    }
    return E.replace(J, N)
}
```

```
    var decode_str=decode('Bl Bm=Bn;Bo(Bl Bp=Bq;Bp<Bn;Bp++)
{Br.Bs(Bm+Bp+"<Bt>");}',
    'var|index|100|for|a|0|document|write|br'.split('|'), 9, 109,
/[\w\$]+/g, {}, {}, []));
```

Step 2：其中对于函数 function C(A)采用了多重 3 元表达式的处理方式，可以用 if/else 语法分解如下：

```
function C(A){
    var res;
    if (A < 62) {
        var r = null;
        if (A < 26) r = 65; //对应 res 在'A'-'Z' 之间
        else {
            if (A < 52) r = 71;  //对应 res 在'a'-'z'之间
            else r = -4;
        }
        res = String.fromCharCode(A + r);
    }
    else {
        if (A < 63) res = '_'; //即 A=62
        else {
            if (A < 64) res = '$';//即 A=63
                else res = C(A >> 6) + C(A & 63); //如果 A>63,进行 64 进制
的高低位分解为 2 部分
            }
        }
        return res;
    }
```

上面的算法其实就是一个仿 base64 编码变换的算法，可以参见文章《Base64 转换：AQAB=65537，你知道为什么吗？》(http://www.cnblogs.com/midea0978/archive/2007/05/22/755826.html)。

变换的算法就是将 0 ~ 63 的数字按照顺序变换为码表：

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789_$
```

对应序列位置的字母。例如 0 对应 A，1 对应 B，2 对应 C……

Step 3：代码 while (A > 0)K[C(D--)] = I[--A];的分析。

实际上这里就是将字典内容与序号值进行对照，记录到 Object 对象中，运算顺序如下：

```
D=109,A=9,K["Bt"]=br
D=108,A=8,K["Bs"]=write
```

```
D=107,A=7,K["Br"]=document
D=106,A=6,K["Bq"]=0
D=105,A=5,K["Bp"]=a
D=104,A=4,K["Bo"]=for
D=103,A=3,K["Bn"]=100
D=102,A=2,K["Bm"]=index
D=101,A=1,K["Bl"]=var
……
```

Step 4：代码 if (".replace(/^/, String)) 的分析。

看起来很高深的一个代码，你想空字符串无论怎么替换，还是空字符串，在 JavaScript 中，空字符串=false，非空字符串=true。

所以这个 if 语句怎么都不会执行，这里是一个混淆视听的代码。你如果愿意，也可以写上更多乱七八糟的代码来达到同样效果。

Step 5：关键代码 return E.replace(J, N)，这里用到了函数 N：

```
function N(A) {
 return K[A] == L[A] ? A : K[A]
}
```

注意 L 对象从来没有赋值，所以 L[A]返回的应该是 undefined，所以可以翻译为：

```
function N(A) {
 return K[A] == undefined ? A : K[A]
}
```

这下看起来就很好理解，关键代码其实就下面这些：

```
function decode(E, I, A, D, J, K, L, H) {
    function C(A) {
        return A < 62 ? String.fromCharCode(A += A < 26 ? 65 : A <
52 ? 71 : -4) : A < 63 ? '_' : A < 64 ? '$' : C(A >> 6) + C(A & 63)
    }
    while (A > 0)K[C(D--)] = I[--A];
    function N(A) {
        return K[A] == undefined ? A : K[A]
    }
    return E.replace(J, N)
}
```

综上分析，该算法的原理就是从 JS 脚本文件中提取单词，存入字典表中。这里使用符号“|”分割字符串数组，然后将单词对应的序号（仿 base64 编码

值）写入原来代码的地方进行替换，这样就构成了该算法的核心。所以实现该压缩算法的代码也就不难了。

工具 JSEncoder 的实现

下面代码采用 Java 实现，写完代码之后才发现这是 JSA(http://sourceforge.net/project/showfiles.php?group_id=175776)压缩算法的再实现。好像作者并没有开源，所以本文的研究算是一种技术上的探讨。通过使用该工具对 jquery-1.2.3.min.js 文件进行压缩测试、调用运行成功，压缩率可以达到 40%。

```
package com.cngd.jstool;

import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.Vector;
import java.text.DecimalFormat;

/**
 * JSEncoder: JS 脚本压缩工具
 * <p/>
* 针对 jquery-1.2.3.min.js 这个文件的压缩比率结果比较如下
 * ------------------------------------------------------------
 * 原始大小 | JSEncoder | JSA-20071021 (2.0 pre-alpha) | jquery packer 算法
 * ------------------------------------------------------------
 * 53kb     | 32kb      | 29kb                         |    29kb
 * ------------------------------------------------------------
 * 因为 JSA 进一步将局部变量进行了压缩，因此相比较更小
 * <p/>
 * User: (在路上 http://www.cnblogs.com/midea0978)
 * Date: 2008-4-18
 * Version:1.0
 */
public class JSEncoder {
    public static final String ENCODE_BASE64 =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789_$";
    public boolean isDebug = false;
```

```
        /**
         * @param filename js filename
         * @param offset   offset>=0 指定偏移变量，不同的 offset 可以实现代码
表位置的变换，较小的 offset 可以获得更大的压缩率
         * @return 压缩后的代码
         */
        public String encode(String filename, int offset) throws
Exception {
            String jscript = readFileData(filename);
            int size = jscript.length();
            jscript = jscript.replaceAll("\n", " ");
            //替换\->\\
            jscript = jscript.replaceAll("\\\\", "\\\\\\\\");
            //替换单引号'=>\'
            jscript = jscript.replaceAll("\\'", "\\\\'");

            Pattern p = Pattern.compile("([\\w\\$]+)");
            Matcher m = p.matcher(jscript);
            String element;
            Vector<String> dict = new Vector<String>();
            int index;
            StringBuffer encscript = new StringBuffer();
            StringBuffer dicttab = new StringBuffer();
            debugInfo("=====编码字典对应表=====");
            while (m.find()) {
                element = m.group(1).trim();
                if (!dict.contains(element)) {
                    dict.add(element);
                    index = dict.size() - 1;
                } else {
                    index = dict.indexOf(element);
                }
                debugInfo(index + "==>" + element);
                m.appendReplacement(encscript, Base64Encode(offset +
index + 1));
            }
            for (String o : dict) dicttab.append(o + "|");
            m.appendTail(encscript);
            debugInfo("=====  编码字典结束  =====");
            debugInfo("Offset=" + offset + ",字典大小=" + dict.size());
            debugInfo("压缩后的代码: \n" + encscript.toString());
            String dictstr = dicttab.substring(0, dicttab.length() -
```

```
1).toString();
            debugInfo("字典字符串:\n" + dictstr);
            String res = formatCode(encscript.toString(), dictstr,
dict.size(), offset);
            int packsize = res.length();
            DecimalFormat df = new DecimalFormat("######.0");
            System.out.println("\n原始文件大小: " + size + "\n压缩后文件大
小: " + packsize);
            System.out.println("================\n压缩比率: " +
df.format((size - packsize) * 100.0 / size) + "%");
            return res;
        }

        private String readFileData(String filename) throws IOException {
            BufferedReader in = new BufferedReader(new FileReader
(filename));
            StringBuffer sb = new StringBuffer();
            while (in.ready()) {
                sb.append(in.readLine() + "\n");
            }
            in.close();
            return sb.toString();
        }

        private void debugInfo(String txt) {
            if (isDebug) System.out.println(txt);
        }

        public static void main(String[] args) {
            System.out.println("JSEncoder 0.5 by midea0978 2008.4");
            System.out.println("================================");
            System.out.println("http://www.cnblogs.com/midea0978\n");
            if (args.length < 2) {
                System.out.println("Usage:java JSEncoder.jar jsfile
outputfile [offset].");
                System.exit(0);
            }
            try {
                System.out.println("输入文件: " + args[0]);
                System.out.println("输出文件: " + args[1]);
                JSEncoder util = new JSEncoder();
                int offset = args.length >= 3 ? Integer.parseInt(args[2]) : 0;
                String code = util.encode(args[0], offset);
```

```
            FileOutputStream fs = new FileOutputStream(args[1]);
            fs.write(code.getBytes());
            fs.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

/**
 * 仿Base64解码
     *
     * @param c 待编码的数字
     * @return 编码值
     */
    private String Base64Encode(int c) throws Exception {
        String res;
        if (c < 0) throw new Exception("Error:Offset必须>=0.");
        if (c > 63)
            res = Base64Encode(c >> 6) + Base64Encode(c & 63);
        else {
            //为了配合appendReplacement方法的使用，将$替换为\$
            res = c == 63 ? "\\$" : String.valueOf(ENCODE_
BASE64.charAt(c));
        }
        return res;
    }

    private String formatCode(String enc, String dict, int size, int
offset) {
        StringBuffer str = new StringBuffer();
        str.append("/* Compressed by JSEncoder */\neval(function(E,I,A,
D,J,K,L,H){function C(A){return A<62?String.fromCharCode(A+=A<26?65:
A<52?71:-4):A<63?'_':A<64?'$':C(A>>6)+C(A&63)}while(A>0)K[C(D--)]=I
[--A];function N(A){return K[A]==L[A]?A:K[A]}if(''.replace(/^/,String))
{var M=E.match(J),B=M[0],F=E.split(J),G=0;if(E.indexOf(F[0]))F=[''].concat
(F);do{H[A++]=F[G++];H[A++]=N(B)}while(B=M[G]);H[A++]=F[G]||'';return H.join
('')}return E.replace(J,N)}(");
        str.append("'" + enc + "',");
        str.append("'" + dict + "'.split('|'),");
        str.append(size + "," + (size + offset) + ",/[\\w\\$]+/g, {},
{}, []))");
        return str.toString();
```

```
        }

    }
```

这是一篇关于 JavaScript 代码压缩混淆技术的文章。在多数情况下，网站使用 JavaScript 压缩混淆是为了有效地减小 js 文件大小。毕竟任何 JavaScript 代码最终都是下载到客户端运行的，针对 JavaScript 加密并没有实际意义。不过了解一下一般 JavaScript 压缩器的工作原理，对深入理解 JavaScript 的动态特性是有帮助的。

JavaScript Table 排序

作者：Cloudgamer [http://www.cnblogs.com/cloudgamer]

前一阵做了个网盘，用到了 Table 的排序，趁热打铁做了一个完整的 Table 排序类出来。程序实现的是在客户端对表格进行排序，有以下特点：

- 自定义排序列、排序属性（例如 innerHTML）、排序数据类型（包括 int、float、date、string）、排序顺序（顺序和倒序）；
- 自定义排序函数；
- 可同时设置多个排序列；
- 支持 IE/FF。

网上也有很多其他的 Table 排序函数，但有的是基于数组，有的不够灵活。我的这个能在原有 Table 结构上加入功能，不用太多改动，基于 OO 的结构也易于使用（当然前提是对 JS 有一定认识）。这里只是满足基本需求，接口可能不够完善，可以自己动手扩展。

先看效果：

名称 / 类型	上传时间 ▲	大小
new.rar	2008-9-12 8:51:09	423.09 K
Scroller.js	2008-9-23 11:26:57	2.5 K
AlertBox.js	2008-9-23 11:26:57	3.48 K
1.htm	2008-10-4 20:21:54	11.13 K
4.htm	2008-10-4 20:21:54	351 b
function.js	2008-10-4 20:24:11	2.78 K
news.xml	2008-10-4 20:24:11	13.74 K
详细功略+剧情流程（一）.doc	2008-10-7 0:07:43	62 K
神秘园 - Nocturne.mp3	2008-10-7 0:07:43	2.97 M
详细功略+剧情流程（二）.doc	2008-10-7 0:07:54	160.5 K
禁止文件预览功能.txt	2008-10-7 0:07:58	860 b

有x的排前面

基本步骤

1. 将需要排序的行放到 tbody 中（程序会直接取 tbody 的 rows）。
2. 把排序行放到一个数组中：

```
Each(this.tBody.rows, function(o){ this.Rows.push(o); }.bind(this));
```

3. 按需求对数组进行排序（用数组的 sort 方法）。

```
this.Rows.sort(!this._order.Compare ? this.Compare.bind(this) :
this._order.Compare);
```

4. 用一个文档碎片（document.createDocumentFragment()）保存排好序的行。

```
var oFragment = document.createDocumentFragment();
Each(this.Rows, function(o){ oFragment.appendChild(o); });
```

5. 把文档碎片 appendChile 放到 tbody 中。

```
this.tBody.appendChild(oFragment);
```

程序说明

排序函数

说到排序，就不得不说数组中 sort 这个方法。手册是这样介绍的：返回一个元素已经进行了排序的 Array 对象。也就是对一个数组进行排序，很多跟排序相关的操作都用到了这个方法。

默认按照 ASCII 字符顺序进行升序排列，使用参数的话可以自定义排序方法。这里为了使排序能适合各种类型的值，定义了一个排序函数作为参数：

```
Compare: function(o1, o2) {
  var value1 = this.GetValue(o1), value2 = this.GetValue(o2);
  return value1 < value2 ? -1 : value1 > value2 ? 1 : 0;
},
```

对于字符来说，用 localeCompare 会更方便，但它不支持日期和数字格式，所以这里用了大于、小于号来做比较。要注意，如果为 sortfunction 参数提供了一个函数，那么该函数必须返回下列值之一：

- 负值，如果所传递的第一个参数比第二个参数小。
- 零，如果两个参数相等。

- 正值，如果第一个参数比第二个参数大。

获取比较值

很多时候要比较的值并不是直接取 innerHTML 的值，那怎么放这个比较值呢？我这里的方法是给 td 设置一个属性来放这个值（例如_ext 和_order）。取这个值也有一点技巧，对 IE 来说用一般的方法取都可以，但 FF 就麻烦一点，对于自定义的数序需要用 getAttribute 来获取，对原有的属性（例如 innerHTML）就需要 td["属性"]这样的方式来取：

```
    var td = tr.getElementsByTagName("td")[this._order.Index], data
= td[this._order.Attri] || td.getAttribute(this._order.Attri);
```

取得值后就根据需要的数据类型进行转换：

```
    switch (this._order.DataType.toLowerCase()) {
        case "int":
            return parseInt(data) || 0;
        case "float":
            return parseFloat(data) || 0;
        case "date":
            return Date.parse(data) || 0;
        case "string":
        default:
            return data.toString() || "";
    }
```

这里要说明的是，添加自定义属性并不是一个符合标准的方法，可以考虑放在 title 之类的属性中。

排序对象

排序对象主要是用来保存该排序的属性的，这里包括：

```
属性 默认值//说明
Attri "innerHTML";//获取数据的属性
DataType "string";//比较的数据类型
Down true;//是否按顺序
onSort function(){};//排序时执行
Compare null;//自定义排序函数
还有两个固定属性:
Index: td 索引
Sort: 设置当前排序对象为排序类的排序对象，并执行排序
```

一个 Table 通常都有多个排序方式，排序对象的作用是保存各个排序方式的参数，排序时就直接使用当前排序对象的属性，这样各个排序方式就互相独立，不会互相影响了。

还有文档碎片，这里并不是必须的，但建议使用，大量 dom 操作时使用文档碎片会更有效率。

这里的触发对象是 a，但按上去是没有边框的，因为我设置了这个样式：

```
    a{outline:none;/*ff*/hide-focus:expression(this.hideFocus=true);
/*ie*/}
```

使用方法

首先实例化一个主排序对象，参数是 Table 的 id：

```
    var to = new TableOrder("idTable");
```

接着添加一个排序对象，第一个参数是 td 索引，第二个参数是需要设置的属性（参考“排序对象”）：

```
    var order2 = to.Add(0, {
        onSort: function(){ Each(SetOrder._arr, function(o){ o.class
Name = ""; }); },
        Compare: function(o1, o2) {
            var value1 = /x/i.test(to.GetValue(o1)), value2 = /x/i.test(
to.GetValue(o2));
            return value1 && !value2 ? 1 : !value1 && value2 ? -1 : 0;
        }
    });
```

然后设置一个触发对象执行排序对象的 Sort 方法进行排序：

```
    $("idBtn").onclick = function(){ order2.Sort(); }
```

程序源码

```
var $ = function (id) {
    return "string" == typeof id ? document.getElementById(id) : id;
};
var Class = {
  create: function() {
    return function() {
      this.initialize.apply(this, arguments);
    }
  }
```

```
    }
    Object.extend = function(destination, source) {
        for (var property in source) {
            destination[property] = source[property];
        }
        return destination;
    }
    Function.prototype.bind = function(object) {
      var __method = this, args = Array.prototype.slice.call(argumen
ts); args.shift();
      return function() {
        return __method.apply(object, args.concat(Array.prototype.slice
.call(arguments)));
      }
    }
    function Each(list, fun){
        for (var i = 0, len = list.length; i < len; i++) { fun(list[
i], i); }
    };
    var TableOrder = Class.create();
    TableOrder.prototype = {
      initialize: function(Table) {
        this.tBody = $(Table).tBodies[0];//tbody对象
        this.Rows = [];//行集合
        this._order = null;//排序对象
        Each(this.tBody.rows, function(o){ this.Rows.push(o); }.bind
(this));
      },
      //排序并显示
      Sort: function() {
        //没有排序对象返回
        if(!this._order){ return false };
        //排序
        this.Rows.sort(this._order.Compare || this.Compare.bind(this));
        this._order.Down && this.Rows.reverse();//取反
        //显示表格
        var oFragment = document.createDocumentFragment();
        Each(this.Rows, function(o){ oFragment.appendChild(o); });
        this.tBody.appendChild(oFragment);
        //执行附加函数
        this._order.onSort();
      },
      //比较函数
```

```
        Compare: function(o1, o2) {
          var value1 = this.GetValue(o1), value2 = this.GetValue(o2);
          return value1 < value2 ? -1 : value1 > value2 ? 1 : 0;
        },
        //获取比较值
        GetValue: function(tr) {
          var td = tr.getElementsByTagName("td")[this._order.Index],
                    data = td[this._order.Attri] || td.getAttribute(th
is._order.Attri);

          //数据转换
          switch (this._order.DataType.toLowerCase()) {
              case "int":
                  return parseInt(data) || 0;
              case "float":
                  return parseFloat(data) || 0;
              case "date":
                  return Date.parse(data) || 0;
              case "string":
              default:
                  return data.toString() || "";
          }
        },
        //添加并返回一个排序对象
        Add: function(index, options) {
          var oThis = this;
          return new function(){
              //默认属性
              this.Attri = "innerHTML";//获取数据的属性
              this.DataType = "string";//比较的数据类型
              this.Down = true;//是否按顺序
              this.onSort = function(){};//排序时执行
              this.Compare = null;//自定义排序函数
              Object.extend(this, options || {});
              //td索引
              this.Index = index;
              this.Sort = function(){ oThis._order = this; oThis.Sort(); };
          };
      }
    }
```

设计模式在 JavaScript 中的应用（1）
——MVC

作者：Truly [http://www.cnblogs.com/truly]

前言

我过去撰写专栏介绍了在 JavaScript 中使用面向对象，本文将讨论软件工程领域的另一个革新——设计模式在 JavaScript 中的应用。

模式的概念诞生于 20 世纪 70 年代，最初用于描述建筑领域的一些特定问题的解决方案。后来这一方案也被应用到软件开发这一领域。在使用 Java 或 C++ 构建大型应用程序的时候，我们几乎无法离开设计模式，并且在这些领域有着丰富的设计模式文化。微软最近也极大地推动了.Net 框架下设计模式的应用。时至今日，您可能已经在上述领域积聚了相当丰富的理论和实践经验，然而，本文要讨论的是如何在 JavaScript 程序设计中使用模式。同时，为了不使这篇文章满是学究味，这次我决定用一个项目作为示例进行讨论。

开始

正如我以前的一篇文章《在 JavaScript 中使用面向对象》[①]中讲到的，JavaScript 这一语言的特点就是语法简单宽松，所以对于 JavaScript，即使在编程时不使用面向对象、设计模式，一样可以开发出可用的程序来。但是如果采用了设计模式，无疑程序将具有更好的健壮性、可维护性以及易读性。所以，作为能工巧匠的您，也一定不会放过令程序蓬荜生辉的机会。

这里首先要介绍一下作为演示的项目。这是一个内容管理系统，用来管理公司的新闻内容，通过后台文章在线编辑，自动发布到公司的几个网站上面。为了获取更好的性能和简便的发布方式，我们最终决定使用静态 HTML 作为文章的呈现载体。讲到这里，也许您已经想到动态生成 HTML 的方案，但是这样会有几个重要弊端，比如网站改版需要重新生成所有的

[①] http://www.cnblogs.com/Truly/archive/2007/07/24/830013.html

HTML 页面，而且要占据相当大的磁盘空间，同时生成模板的修改也是对美工的一大考验，最终整个过程变得非常复杂。有什么好的解决方案吗？答案是肯定的。现在该向大家介绍今天的主角了：MVC（模型视图控制器）模式。这一设计模式被我们反复使用了多年，但是今天我们还是再一次回顾一下 MVC 的知识。

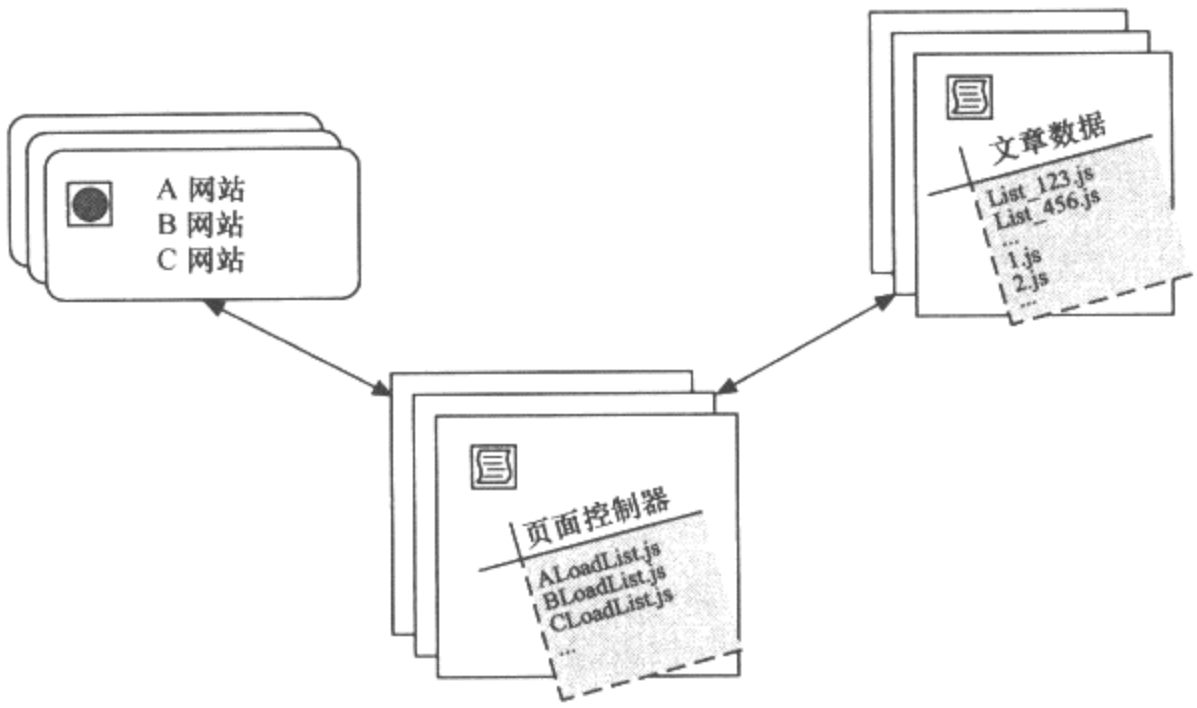
MVC 模式

MVC，英文全名是 Model/View/Controller，常被用来构建用户界面。它最典型的作用就是将程序与用户交互的部分进行分离，可能覆盖应用的所有层或者跨越多个层，比如经典书籍《设计模式——可复用面向对象软件的基础》一书中对文字处理程序界面的处理。MVC 中最重要的一点是视图不应该与模型相互通信，例如当用户操纵页面上的一个按钮时，不应该直接与数据模型进行交互，而应该将这一交互过程通过控制器来完成。整个过程见下图：

如果用户想从模型中获取某个数据并显示在页面上，那么这一请求必须通过控制器来完成获取数据并更新视图。这样的好处就在于视图和模型间呈松散耦合，他们之间没有直接的联系，不需要了解对方的内部实现。MVC 的优点在 Web 应用中表现更为明显，对从事 Web 程序开发的人而言，分离视图和模型几乎是非常自然的，MVC 的应用司空见惯。例如 ASP.NET 1.x 时代的 code-behind 和 2.0 时代的 code-beside，等等，我们总是把视图写在 aspx 页面，而把代码写到 cs 文件中。当然也有人将 MVC 定义得更为苛刻，他们将所有产生的 HTML 的代码和相关代码都归结为视图。但是无论从广义还是狭义上，在 Web 应用程序的开发中，MVC 都得以普遍应用。

好了，现在回到我们的项目上来，同样是出于性能的考虑，我们决定将数据和图片保存在与前端 Web Server 不同的服务器上，以分散前端服务器的压力。但是由于 XML 跨域的安全限制，使得我们最终放弃了使用 XML 作为数据载体。作为 XML 最好的替代品的正是 JSON，恐怕找不到比它更为合适的技术了，而且使用 JSON 带来的好处远远超过 XML 所能给项目带来的好处。如果您长期从事于 Web 2.0 技术研究，那么这应该是您非常熟悉的一个技术了。如果您还不够了解 JSON，那么您可以阅读我的另一篇文章《深入浅出 JSON》。

至此，基本技术已经确定下来了，最终的解决方案架构图见下图：



整个系统已经一目了然了，接下来就是实施。对于内容管理系统，除了后台的文章编辑和分类管理等，每个网站的前台显示都需要两大功能：一个是新闻列表，一个是新闻查看。这里需要编写几个 js 文件，并且分别对应不同的 HTML 页面，所有相关的文件分别如下：

1. 控制器

- base.js　存放了一些服务器配置列表配置等信息
- loadlist.js
- loaddata.js

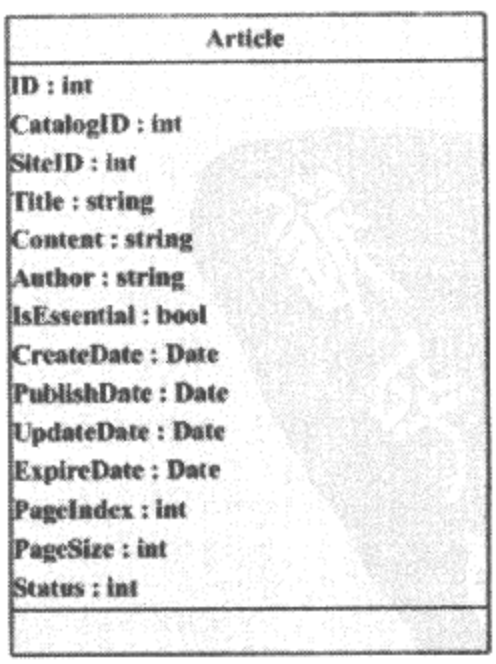
2. 视图

- List.html
- detail.html

3. 数据模型

- list_xx.js　　文章列表的数据
- xx.js　　文章详细内容的详细数据

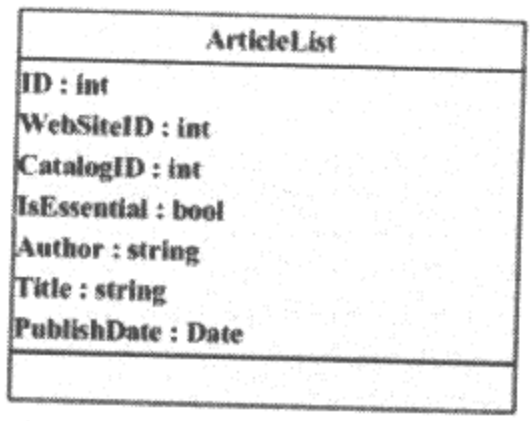
首先我们定义了文章的数据模型，见下图。

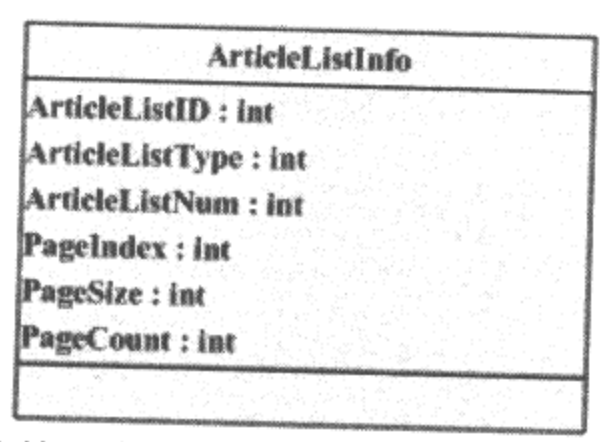


与数据库对应，这个模型包含了文章中几乎所有需要用于显示的详细数据，并最终映射为 JSON 数据类型，封装生成到文章对应的 js 数据文件 19.js，参见下面代码：

```
var Article={
"ID":19,
"CatalogID":10,
"CatalogID":1,
"Title":"Hello world!",
"Content":"Welcome to learn the JavaScript patterns.",
"Author":"Truly",
"IsEssential":false,
"CreateDate":new Date('Wed, 25 Jul 2007 09:59:00 GMT+8'),
"PublishDate":new Date('Wed, 25 Jul 2007 09:59:21 GMT+8'),
"UpdateDate":new Date('Wed, 25 Jul 2007 09:59:21 GMT+8'),
"ExpireDate":new Date('Mon, 01 Jan 2046 00:00:00 GMT+8'),
"PageIndex":1,
"PageCount":1
"Status":1,
}
```

然后是为列表设计的两个数据模型：ArticleList 和 ArticleListInfo，见下面两个图：





这两个数据模型都是为文章列表服务的，所以我们将其合并生成到同一个 js 文件中，比如罗列 CatalogID = 10 的所有文章的一个 js 文件 list_22.js：

```
var ArticleList=[
{"ID":"866","WebSiteID":"10","CatalogID":"42","IsEssential":false,
"Author":"Truly",
"Title":"Title1","PublishDate":new Date('Thu, 19 Jul 2007 11:18:00
GMT+8')},
{"ID":"864","WebSiteID":"10","CatalogID":"42","IsEssential":false,
"Author":"Truly",
"Title":"Title2","PublishDate":new Date('Thu, 19 Jul 2007 11:12:00
GMT+8')},
{"ID":"836","WebSiteID":"10","CatalogID":"42","IsEssential":false,
```

```
"Author":"Truly",
    "Title":"Title3","PublishDate":new Date('Wed, 18 Jul 2007 10:35:00
GMT+8')},
    {"ID":"817","WebSiteID":"10","CatalogID":"42","IsEssential":false,
"Author":"Truly",
    "Title":"Title4","PublishDate":new Date('Mon, 16 Jul 2007 17:17:00
GMT+8')}
    ]
    var ArticleListInfo= {"ArticleListID":44, "ArticleListType":2,
"ArticleListNum" : 20,
    "PageIndex":1, "PageSize":20, "PageCount":1 }
```

根据JavaScript的特性我们将一些配置信息定义为全局变量存放在base.js文件（如下）中，方便以后的部署维护工作。

```
// js数据服务器
var GLOBAL_DATAHOST = 'http://data.domain.com/list/';
// 新闻完整列表ID常量
var OFFICIALNEWS = 33;
var SYSTEMANNOUNCEMENT = 34;
var ACTIVITYNEWS = 35;
var OTHERNEWS = 36;
```

…

最后，编写控制器代码：

```
// 显示列表
function RunShowMethod()
{
    var iPageCount = ArticleListInfo.PageCount;
    var iPageSize = ArticleListInfo.PageSize;
    var iPageIndex = ArticleListInfo.PageIndex;
    var strNewIcon="";
    var count = ArticleList.length;
    var arr = new Array(count);
    for (var i=0;i<count; i++)
    {
        if(typeof ArticleList[i].ATPublishDate == "string")
            dt = new Date(Date.parse(ArticleList[i].ATPublishDate));
        else
            dt = Date.parse(ArticleList[i].ATPublishDate);

        // 列表模板
        arr[i] = "<tr>\
```

```
                        <td> </td>\
                        <td><img src=\"images/icon.jpg\" width=\"12\"
height=\"15\"></td>\
                        <td><a href='" + strCommunionDetailPage + "?id="
+ ArticleList[i].ATID + "&catID=" + ArticleList[i].ATCategoryID + "'
target='_blank' class='S9pt'>" + ArticleList[i].ATTitle + "</a></td>\
                        <td>"  + dt.toString() +  "</td>\
                        <td> </td>\
                      </tr>";
            // 分隔行模板
            if(i != count -1 )
               arr[i]+="<tr><td colSPAN=\"5\" nowrap
align=\"center\"><hr></td></tr>";
        }
        // 显示列表内容

        $("lbContent").innerHTML = "<Table width=\"100%\" border=\"0\"
cellspacing=\"0\" cellpadding=\"0\">\
                                    <tr><td height=\"5\"></td></tr>"
                                       + arr.join("") + " \
                                    <tr><td height=\"10\"></td></tr>\
                                  </Table>";
        // 显示分页导航条
        $('pager').innerHTML = getPagerHTML(getParm("page"), iPageCount);
    }
    // 获取分页 HTML 代码
    function getPagerHTML(pageIndex, pageCount, group)
    {
        // 如果只有 1 页，不再生成页码
        if(pageCount == 1) return "";
        if(!pageIndex) pageIndex = 1;
        strPager = "";
        if(!group ) group = 10;
        var m = Math.floor(group / 2) + 1;
        var start = pageIndex - m +1;
        if(start < 1) start = 1;
        var end = group+start-1;
        if(end > pageCount) end = pageCount;
        if(end - start < group)
        {
            start = end - group + 1;
            if(start < 1) start = 1;
        }
        for(var i=start;i<=end;i++)
```

```
        {
            if(i == pageIndex)
                strPager += " <SPAN class=\"STYLE13\" ><font color=
\"#d20d67\"><b>" + i + "</b></font></SPAN>";
            else
                strPager += " <a href='"+PageName+"?page=" + i + "'
class='Blue9pt'>" + i + "</a>";
        }
        if(pageIndex > 1)
            strPager = "<a href='"+PageName+"?id=" + listID + "&page="
+ (pageIndex -1) + "' class='Blue9pt'>上一页</a> " + strPager;
        else
            strPager = "<font class='Blue9pt'>上一页</font> " + strPager;
        if(pageIndex < pageCount)
            strPager += "  <a href='"+PageName+"?id=" + listID
+ "&page=" + (parseInt(pageIndex)+1) + "' class='Blue9pt'>下一页</a>";
        else
            strPager += "  <font class='Blue9pt'>下一页
</font>";
        return strPager;
    }
```

这些控制器依赖 HTML 页面上的一些容器 ID，例如 lbTitle、lbContent 和 lbCreateDate 等，最后还需要在 HTML 适当位置增加一些加载语句，或者在控制器中处理文档的 onload 事件加载。例如：

```
var tet = document.createElement('<SCRIPT>');
tet.src = jsToLoaded[LoadedCount] + "?tmp=" + Math.random();
document.body.appendChild(tet);
```

同时结合 JavaScript 文件加载，我们在数据 js 中还增加了主动更新语句：

```
if(typeof(RunShowMethod) == "function") RunShowMethod();
```

这样当页面加载结束的时候，可以主动填充到 HTML 中，我们后面讲 Observe 模式的时候再详细讨论。还有部分函数我没有放出源码，大家可以自己进行完善，这个系统是支持文章分页的。

通过上面的具体分析可以看出，在使用了 MVC 模式后整个系统变得简单明了。通常情况下数据模型不需要进行任何调整，如果网站需要改版，只需调整对应的 list.html 和 detail.htm 两个页面就可令这个网站全面更新，同时数据的独立存放也极大地减少了重复 HTML 代码造成的空间浪费。

至此，我们的第一个模式 MVC 的讨论就结束了。这个方案很好地演示了 MVC 在实际项目中的应用，但是需要注意的是它的一些缺陷。

设计模式在 JavaScript 中的应用（2）
——Observer

作者：Truly [http://truly.cnblogs.com]

上次我们讨论了 Web 开发中最重要的设计模式 MVC，今天我们要讨论的是 Observer 模式，与 MVC 这样的大型设计模式相比，Observer 模式则要轻量很多。

Obsever 简单应用

请先看一段代码：

```
// the process array calling after page loaded for page listener.
var PageLoadListener = new Array();
// page listener
function onDocumentLoaded()
{
    for (var a in PageLoadListener)
    {
        if(typeof  PageLoadListener[a]  == 'function')
PageLoadListener[a]();
    }
}
// Add a listener to current page to run all function on the page.
if (document.addEventListener)
 document.addEventListener('DOMContentLoaded', onDocumentLoaded,
false);
else
 window.attachEvent('onload', onDocumentLoaded);
```

而在另外一个 js 中我们定义：

```
PageLoadListener.push(domLoaded);    // push the domLoaded function
into the listener array.
// a method need to call after page is loaded
```

```
function domLoaded()
{
    alert('document loaded');
}
```

通常我们经常要处理 window.onload 事件，例如使用下面代码来指定 onload 事件：

```
window.onload=aFunction
```

用这样的方式声明的时候，很可能会覆盖已经定义过的 window.onload 事件，或者还有很多事件要在 onload 中执行，那么如何应对这种情况呢？

Observer 模式恰好可以用来处理这种情况。首先需要为页面定义一个监听器，检测页面中需要处理的事件，然后定义一个全局的监听器数组。这样，需要处理的事件都可以注册到这个监听器数组中，然后统一进行调用。

如上面代码中，我们将需要处理的事件名通过下面代码注册到 Listener 数组中，这样的注册过程可能遍布到不同的 js 文件或脚本块中，最后使用监听器集中对数组中的元素进行调用，这样一来就很好地解决了 window.onload 事件冲突的问题。

```
PageLoadListener.push(domLoaded);
```

Obsever 进阶应用

下面我们演示一个更加复杂的 Obsever 模式应用，来自著名的 Prototype 框架，请先看代码：

Hello.htm

```
<html>
<head>
<title>Obsever Demo</title>
<script language="JavaScript" type="text/JavaScript" src=
"Obsever.js"></script>
<script language="JavaScript" type="text/JavaScript" src=
"Controller.js"></script>
</head>
<body>
    <input id='textbox1' name='textbox1'/>
    <select id='selElement1' >
```

```
        <option >choose</option>
        <option value='1' >1</option>
        <option value='2'>2</option>
        <option value='3'>3</option>
    </select>
</body>
</html>
```

Obsever.js

```
function $(id){return document.getElementById(id);}
var $A = Array.from = function(iterable) {
  if (!iterable) return [];
  if (iterable.toArray) {
    return iterable.toArray();
  } else {
    var results = [];
    for (var i = 0, length = iterable.length; i < length; i++)
      results.push(iterable[i]);
    return results;
  }
}
var Browser={
    isWebKit : navigator.userAgent.indexOf('AppleWebKit/') > -1
    }
Function.prototype.bind = function() {
  var __method = this, args = $A(arguments), object = args.shift();
  return function() {
    return __method.apply(object, args.concat($A(arguments)));
  }
}
if (!window.Event) {
  var Event = new Object();
}
Object.extend = function(destination, source) {
  for (var property in source) {
    destination[property] = source[property];
  }
  return destination;
}
Object.extend(Event,
{
observe: function(element, name, observer, useCapture) {
   if(typeof element != 'object')
```

```
    element = $(element);
    useCapture = useCapture || false;

    if (name == 'keypress' &&
      (isWebKit || element.attachEvent))
      name = 'keydown';

    Event._observeAndCache(element, name, observer, useCapture);
  },

stopObserving : function(element, name, observer, useCapture) {
    element = $(element);
    useCapture = useCapture || false;

    if (name == 'keypress' &&
        (Browser.WebKit || element.attachEvent))
      name = 'keydown';

    if (element.removeEventListener) {
      element.removeEventListener(name, observer, useCapture);
    } else if (element.detachEvent) {
      try {
        element.detachEvent('on' + name, observer);
      } catch (e) {}
    }
  },
  observers: false,
  _observeAndCache: function(element, name, observer, useCapture) {
    if (!this.observers) this.observers = [];
    if (element.addEventListener) {
      this.observers.push([element, name, observer, useCapture]);
      element.addEventListener(name, observer, useCapture);
    } else if (element.attachEvent) {
      this.observers.push([element, name, observer, useCapture]);
      element.attachEvent('on' + name, observer);
    }
  }
}
)
```

Controller.js

```
function changeHandler()
{
```

```
        $('textbox1').value=this.value;
    }
    Event.observe(window, 'load',
        function(){Event.observe('selElement1','change',changeHandler.bind
($('selElement1')));}
    );
```

上面代码的功能：当选择下拉框的时候，调整文本框的值。我们演示了onchange 和 onload 事件的监听，同样也可以应用到任何 DOM 节点的各个事件上。你可能说在<select>标签中直接添加 onchange 事件就可以了，但为什么要这么做?

首先这样可以更好地分离代码和视图,就像我上篇文章中讨论的 MVC 模式，我们应该尽可能地分离代码和视图，尤其是当你构建一个大型的应用程序的时候（例如飞鸽这样的网站），可以从中受益。

同时，通过这种方式可以设计出一个完整的客户端事件流程。关于 JavaScript 事件模型的讨论，将是我们后面文章的讨论内容。

注： 文中代码部分取自著名的 Prototype 框架，不过根据行文需要，我做了适当改动。

JavaScript 面向对象之属性实现

作者：Truly [http://www.cnblogs.com/Truly]

序言

在前面的《在 JavaScript 中使用面向对象》[①]中我介绍了 MSDN 的一篇文章《使用面向对象的技术创建高级 Web 应用程序》[②]，作者简单介绍了 JavaScript 面向对象的一些关键技术，但是作者在讲到闭包概念的时候犯了一个明显的错误："正常情况下，无法从函数以外访问函数内的本地变量。函数退出之后，由于各种实际原因，该本地变量将永远消失"（详见原文）事实上这段描述是错误的。

请先看如下代码：

```
<script>
function Test(abc)
{
    this.g = function(){debugger;};
}

var p = new Test(2);
p.g();
</script>
```

开始

如果启用 IE 的调试功能，并安装了脚本调试器（例如 VS），那么在程序提示调试的时候进入调试，此时你可以醒目地发现 abc 依然存在，并且完好保存了正确的值，而非永远消失。但是这也不是本文要讨论的重点，只是希望大家以后能够多动手、多实践，像 MS ASP.NET AJAX 团队的软件设计工程

① http://www.cnblogs.com/Trulg/archive/2007/07/24/830013.html
② http://msdn.microsoft.com/zh-cn/magazine/cc163419.aspx

师都会犯这种错误，更何况我们呢?

本文要讨论的是面向对象编程中常用的属性，但是在 JavaScript 中属性无法像高级编程语言那样可以直接使用，看起来更像方法，这种实现方式也有人称之为闭包，但本文以属性相称。

属性是对私有变量的一种保护手段，同时提供了像 public 变量一样的使用效果，近代的高级编程语言（例如 C#和 Java）都支持属性这一特点。

我们知道，函数的入口参数被声明为该函数的本地变量。对于本地变量，像我前面《在 JavaScript 中使用面向对象》关于全局变量和局部变量中描述的那样，由于其作用域仅限于函数内部，所以无法在外部对其进行访问，例如 p.abc 不会返回 p 内部的 abc 变量。这一点跟高级编程语言完全一致，你无法在类外部访问其 private 变量，但是我们可以借助 public 方法来返回私有变量。所以高级编程语言（如 Java，C#等）中属性的作用就是保护私有变量。像 C#这门语言，属性最终会由编译器编译为 get_属性名()这样的方法，当我们使用某个属性时，实质上是调用一个方法。

《使用面向对象的技术创建高级 Web 应用程序》的作者认为是由于方法的定义才使局部变量存活下来，这一点是不正确的，具体内容我们前面已经分析过了，如果你仍有疑问，那么再仔细研究下面的代码：

```
function Person(name, age) {
    this.getName = function() {debugger; return 1; };
}
var o = new Person(1,2);
o.getName();    // 进入调试后发现 name=1
var t = new Person(2,4);
t.getName();    // 进入调试后发现 name=2
o.getName();    // 进入调试后发现 name=1,并未受到其他实例的影响
```

对于这个问题，我起初也认为是因为变量有引用才没被销毁，最后证明局部变量在对象销毁前其内部的变量不会销毁。

同时那篇文章中另外一段也是不准确的：

这些私有成员与我们期望从 C#中产生的私有成员略有不同。在 C# 中，类的公用方法可以访问它的私有成员。但在 JavaScript 中，只能通过在其闭包内拥有这些私有成员的方法来访问私有成员（由于这些方法不同于普通的公用方法，它们通常被称为特权方法）。因此，在 Person 的公用方法中，仍然必须通过私有成员的特权访问器方法才能访问私有成员。

关于这一点，他的表述相当模糊，事实上我们可以这样理解：在 C#中，我们可以在类的任何方法中访问类的私有成员变量，而在 JavaScript 中，只能使用在 function 方式中定义的方法对私有成员访问，而无法在 prototype 方式

定义的方法中访问。

如果这样讲还不能理解的话，那么还可以这样理解，在 JavaScript 中的私有变量无法在其声明的函数外访问，例如：

```
function Person()
{
    var ttt;
}
```

永远不能在{}外部试图访问 ttt。

现在我们更加深入地理解了变量作用域在 JavaScript 中的特点。

前面讲了高级编程语言中属性的种种好处，又研究了 JavaScript 对私有变量的保护，那么您对 JavaScript 中属性的实现应该非常清楚了，这里引用《使用面向对象的技术创建高级 Web 应用程序》文中的一段示例代码：

```
function Person(name, age) {
    this.getName = function() { return name; };
    this.setName = function(newName) { name = newName; };
    this.getAge = function() { return age; };
    this.setAge = function(newAge) { age = newAge; };
}
```

小结

通过本文实例演示，我们讲解了在 JavaScript 中的属性封装方法，这种方法也称为“闭包”，通过这种方式，我们可以在 JavaScript 模拟实现属性这一高级编程语言的特性，对对象引用进行封装，通过访问器进行访问，有效保护了对象的私有成员。

结合 Truly 的《JavaScript 面向对象之属性实现》，可以说研究 JavaScript 变量作用域的一个重要意义是在使用面向对象方法进行 JavaScript 开发时，分清一个类的私有成员和公有成员，以更好地提高开发效率和代码质量。尤其对于 JavaScript 这样的动态语言，最好能够在成员变量命名时就显式表现出该成员是公有的还是私有的。这点在团队合作开发时尤为重要。就像 Truly 在他另一篇文章中提到的，无论如何使用 JavaScript，都能开发出可用的程序，但是区别在于，开发出的程序是否是一个良好、高效、易维护的程序。

基于“甘露模型”的多重继承和接口实现

附带“准”桥接模式的验证

作者：梁逸晨 [http://www.cnblogs.com/kvspas]

看了李战师兄的《悟透 JavaScript》[①]，受益匪浅。基于甘露模型，我稍微做了些修改，支持了多重继承和接口（浏览器、ASP-jscript 和 jscript.net 三种环境下调试通过）。

关于多重继承，也许会有些朋友存在意见，特别是在.NET 和 Java 程序员领域。我的观点是，用与不用和用得好用得坏是人的问题，而不是编译器的问题。

在接口的继承上，函数 Class 会自动查找当前类是否已经实现了接口的方法，如果没有实现，则会抛出错误，终止执行，这也相当于尽量接近了 C#。

```
class 某某某:接口 1，接口 2，接口 3
{
    //------------------------
}
```

在最后面，通过一个桥接模式来验证这些功能。再次感谢李战师兄的点化。

```
var Function = {};
 Function.Amethod = null;
 function Class()                          //创建类的函数，用于声明类及继承关系
  {
     var args = Class.arguments;           //获取参数集合
     function class_()                     //创建类的临时函数壳
      {
         this.Type = args[0];              //我们给每一个类约定一个 Type 属性，
引用其继承的类
         try
          {
             for (var i = 0; i < args.length; i++)
```

[①] http://www.cnblogs.com/leadzen/archive/2008/02/25/1073404.html

```
                {
                    for (var member in args[i])
                    {
                        if (args[i][member] != Function.Amethod)
                        {
                            this[member] = args[i][member];    //复制类的
全部定义到当前创建的类
                        }
                        else
                        {
                            //如果检测到这是一个虚函数,则判断继承链中存在接口
                            //检测当前类是否已经实现了接口的方法
                            if (!args[args.length - 1][member]) {
                                throw ("错误信息: 未实现" + member);
                                return false;
                            }
                        }
                    }
                }
            }
            catch (err) {
                Response.Write(err); //在 asp 和 asp.net 中
                Response.End();
                window.alert(err); //在浏览器中
            }
        };
        return new class_();
    }
    var Interface = function(method) {
        function temp() {
            for (var a in method) {
                this[a] = Function.Amethod;
            }
        }
        return new temp();
    }
    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，这和 DELPHI 比较相似
        };
        new_.prototype = aClass;
        return new new_();
    }
    /* -------------------- 甘露滋养完毕，下面是 桥接模式，下面的测试仅限于浏览
器环境中 -----------
       -------------------- 测试环境要求 HTML 文档中必须有一个 id 值为"re"的
DIV 标签 ---------- */
    var AbstractCar = Interface({ run: null });
    var AbstractRoad = Interface({ car: null, run: null });
    var Car = Class
    (
        AbstractCar,
        {
            run: function() {
                document.getElementById("re").innerHTML += "Car ";
            }
        }
    );
    var Bus = Class
    (
        AbstractCar,
        {
            run: function() {
                document.getElementById("re").innerHTML += "Bus ";
            }
        }
    );
    var SpeedWay = Class
    (
        AbstractRoad,
        {
            car: {},
            run: function()
            {
                this.car.run();
                document.getElementById("re").innerHTML += "in the
SpeedWay";
            }
        }
    );
```

```
var Street = Class
(
    AbstractRoad,
    {
        car: {},
        run: function()
        {
            this.car.run();
            document.getElementById("re").innerHTML += "in the Street";
        }
    }
 );
window.onload = function() {
    var Road1 = New(Street); //桥接模式特征: 变化点 1
    Road1.car = New(Bus); //变化点 2
    Road1.run();
    document.getElementById("re").innerHTML += "<br />";
    var Road2 = New(SpeedWay); //桥接模式特征: 变化点 1
    Road2.car = New(Car); //变化点 2
    Road2.run();
}
```

在下面的服务器端应用执行时，需要把上面的甘露模型代码中的“window.alert(err);”注释掉，并开启原来已经注释的 Response.Write。

接口应用在测试（ASP 和 ASP.NET）如下。

```
var AbstractCar = Interface({ run: null });//抽象车
var AbstractRoad = Interface({ car: null, run: null });//抽象路
var Car = Class //实体小汽车,继承于抽象车
 (
    AbstractCar,
     {
         run: function() {
            Response.Write("Car ");
        }
    }
 );

 var Bus = Class //实体公交车,继承于抽象车
 (
    AbstractCar,
     {
```

```
        run: function() {
            Response.Write("Bus ");
        }
    }
);

var SpeedWay = Class
(
    AbstractRoad,
    {
        car: {},
        run: function()
        {
            this.car.run();
            Response.Write("in the SpeedWay");
        }
    }
);

var Street = Class
(
    AbstractRoad,
    {
        car: {},
        run: function()
        {
            this.car.run();
            Response.Write("in the Street");
        }
    }
);

var Road1 = New(Street); //桥接模式特征: 变化点 1
Road1.car = New(Bus); //变化点 2
Road1.run();
Response.Write("<br />");
var Road2 = New(SpeedWay); //桥接模式特征: 变化点 1
Road2.car = New(Car); //变化点 2
Road2.run();
```

通过以上的两个例子，虽然我们在代码层面和执行层面上都实现了接口，但是回过头来说，JavaScript 语言本身就属于动态语言，它的变量声明是可以指向任何对象的："var a = 1;"、"var a = "text";" 和 "var a = new Array();" 都

是正确的，这也说明了 JavaScript 本身就内置了“多态性”，似乎我们再构造一个接口就显得有些多余了。

其实不然，接口在强类型语言中除了在多态性应用上起到至关作用外，还有另一个特征：“你必须实现某某某方法!”，这在大型程序的开发上显得至关重要。或许我们会说使用 JavaScript 谈不上开发大型程序，但是要注意到，哪怕是一个小组件，连续不断地升级和修改都是常见的，两个月前写的代码很难保证两个月后自己还能看得懂多少。此时如果在代码中出现接口继承，多多少少可以起到一个提示的作用——“想起来了，原来是这么回事”，以及一个强制性检测作用。另一种情况是在其他程序员基于你的组件库平台之上开发新的应用时，你也只需要告诉他：“你继承这个接口就行了，它会自动帮你验证的。”

多重继承，办法很简单，不再多说了，给个例子：

```
var p1 = Class({pp1:"pp1"});
var p2 = Class({pp2:"pp2"});
var p3 = Class({pp3:"pp3"});
var xxx = Class(p1,p2,p3,{/*     当前类主体定义     */});
```

另外提到一点，通过 JavaScript 本身的：

```
function Foo()
{
    new 类1();
    new 类2();
}
```

也可以实现多重继承，这里仅仅是做了一个比较方便的封装，但这是在甘露模型的基础上做的动作，可以显著提升性能，避免了不必要的资源消耗。这里要注意一点，这是类继承，它不会像接口那样去验证是否实现了某某函数。

好了，本文就此结束，如果读者遇到什么难题，或者发现了什么 Bug 的话，可以到我的博客上给我留言，大家共同讨论。

在JavaScript面向对象编程中使用继承(1)

作者：鸟食轩 [http://www.cnblogs.com/birdshome]

前几天做了一个JavaScript版的CollecionBase类(http://download.cnblogs.com/birdshome/archive/2005/01/25/96739.html)，对于需要使用集合作为主要数据结构的类，可以作为它的基类。不过当时没有给出继承的示例，搞得有几位博友对JavaScript继承比较迷惑，于是我今天使用4种方式分别实现4个ArrayList派生类。(http://www.cnblogs.com/birdshome/archive/2005/01/28/95933.html)

关于使用JavaScript进行面向对象编程(OOP)，网上已有很多文章说过了。这里我推荐几篇文章大家看看，如果没有理解怎么使用JavaScript的Function对象的 prototype 属性来实现类定义及其原理，那么就仔细看看“面向对象的JavaScript 编程”(http://blog.csdn.net/liuruhong/archive/2004/05/19/1926.aspx)、“面向对象的 Jscript”(http://bbs.blueidea.com/viewthread.php?tid=1822005)和“Classical Inheritance in JavaScript”(http://www.crockford.com/JavaScript/inheritance.html)(特别是第一篇及其相关讨论的文章)，否则后面会一头雾水。

那个CollectionBase的代码就不列了，前面第一个链接就是它。下面是用四种方法实现的JavaScript“继承”(以后再提到“继承”就不打引号了，反正知道JavaScript的“继承”不是经典OO里的继承就行了)。

构造继承法

```
<script language="JavaScript">
function ArrayList01()
{
    this.base = CollectionBase;
    this.base();

    this.m_Array = this.m_InnerArray;

    this.foo = function()
    {
        document.write(this + ': ' + this.m_Count + ': ' + this.m_Array
```

```
+ '<br />');
        };

        this.Add = function(item)
        {
            this.InsertAt(item, this.m_Count);
        };

        this.InsertAt = function(item, index)
        {
            this.m_InnerArray.splice(index, 0, item);
            this.m_Count++;
        };

        this.toString = function()
        {
            return '[class ArrayList01]';
        };
    }
    </script>
```

原形继承法

```
<script language="JavaScript">
function ArrayList02()
{
    this.InsertAt = function(item, index)
    {
        this.m_InnerArray.splice(index, 0, item);
        this.m_Count++;
    };

    this.m_Array = this.m_InnerArray;

    this.toString = function()
    {
        return '[class ArrayList02]';
    };
}

ArrayList02.prototype = new CollectionBase();
```

```
    ArrayList02.prototype.foo = function()
    {
        document.write(this + ': ' + this.m_Count + ': ' + this.m_Array
+ '<br />');
    };

    ArrayList02.prototype.InsertAt = function(item, index)
    {
        this.m_InnerArray.splice(index, 0, item);
        this.m_Count++;
    };
    </script>
```

实例继承法

```
    <script language="JavaScript">
    function ArrayList03()
    {
        var base = new CollectionBase();

        base.m_Array = base.m_InnerArray;

        base.foo = function()
        {
            document.write(this + ': '+ this.m_Count + ': ' + this.m_Array
+ '<br />');
        };

        base.InsertAt = function(item, index)
        {
            this.m_InnerArray.splice(index, 0, item);
            this.m_Count++;
        };

        base.toString = function()
        {
            return '[class ArrayList03]';
        };
        return base;
    }
    </script>
```

附加继承法

```
<script language="JavaScript">
function ArrayList04()
{
    this.base = new CollectionBase();

    for ( var key in this.base )
    {
        if ( !this[key] )
        {
            this[key] = this.base[key];
        }
    }

    this.m_Array = this.m_InnerArray;

    this.InsertAt = function(item, index)
    {
        this.m_InnerArray.splice(index, 0, item);
        this.m_Count++;
    };
}

ArrayList04.prototype.foo = function()
{
    document.write(this + ': ' + this.m_Count + ': ' + this.m_Array
+ '<br />');
}

ArrayList04.prototype.toString = function()
{
    return '[class ArrayList04]';
}
</script>
```

派生类中的 foo 是一个新增加的函数，用来输出类的类型和 m_InnerArray 里的数据。toString()相当于 override 了 CollectionBase 中的 toString()，不过其实就是赋值覆盖，和“从 JavaScript 函数重名看其初始化方式”（http://download.cnblogs.com/birdshome/archive/2005/01/08/87913.html）一文中说到的原理是一样的。这 4 种方法的原理和区别我接下来会详细分析和介绍。

在JavaScript面向对象编程中使用继承(2)

作者：鸟食轩 [http://www.cnblogs.com/birdshome]

昨天扔了一堆JavaScript类“继承”的代码，这些代码其实并不是所有的都能正常执行。不是我不原意写出都能好好执行的继承类代码，而是这些方法本身就各有优缺点。下面我分别说说它们的原理和使用时的注意事项。

构造继承法的原理

构造继承法关键代码是function ArrayList01()中的：

```
this.base = CollectionBase;
this.base();
```

这里的base不是C#派生类中的那个base的概念，完全就是一个任意的JavaScript变量名。调用this.base();其实就是执行的CollectionBase();，不过不是new CollectionBase();。没有new基类，那么如何得到CollectionBase中的方法和属性呢？这里使用了this作用域的一个hack，“欺骗”了脚本引擎。当我们从类ArrayList01的构造函数中调用this.base();时，在基类CollectionBase中的this就是ArrayList01的一个实例，于是执行CollectionBase的构造函数，就动态地把基类的成员和方法attach（附加）到ArrayList01实例中了。构造法的问题也就是从这里产生了。

构造继承法的缺陷

第一个问题，关键是出在上面那两段代码上，因为始终没有new基类。这样带来的问题就是不能把基类CollectionBase中的原型属性和方法attach（附加）到ArrayList01的实例中。这样还算是什么继承啊？！所以在上篇文章中我为了不改动CollectionBase类，而单独为其实现了一个Add()方法（只是为了统一示例代码而已）。解决这个缺陷的方法其实也很简单，就是要求基类不能使用prototype来导入属性和方法，而要把所有的属性和方法都写

到构造函数中去，分别是：this.Attribute = ...; 和 this.Method = function() {...}; 这种形式。

第二个问题是，this.base = CollectionBase;和 this.base();必须写在派生类构造函数的最开头（不是一定要是第一行和第二行，而是它们的前面不能有 this.xxx 这种定义为 ArrayList01 导入的任何属性或方法），因为调用 this.base(); 时会向 this（ArrayList01 的一个实例）中注入基类的属性和方法，如果基类中有和 this 中已导入的属性和方法重名的，就自动覆盖掉子类中的方法。

第三个问题是，子类也不能使用 prototype 来导入属性和方法，这和问题二中的重名覆盖道理一样。由于 prototype 导入的属性和方法在子类 new 的时候就生成了，所以也存在和基类重名而被覆盖的潜在错误威胁。解决办法和基类编写规则一样，不要使用 prototype，并且将子类的属性和方法定义代码放在导入继承的代码（this.base = CollectionBase;this.base();）之后。

构造继承法的示例

```
<script language="JavaScript">
document.write('构造继承法:<br>');
var arrayList11 = new ArrayList01();
arrayList11.Add('a');
arrayList11.Add('b');
arrayList11.foo();
var arrayList12 = new ArrayList01();
arrayList12.Add('a');
arrayList12.Add('b');
arrayList12.Add('c');
arrayList12.foo();
</script>
```

示例运行结果为：

```
构造继承法:
[class ArrayList01]: 2: a,b
[class ArrayList01]: 3: a,b,c
```

小结

JavaScript 的构造继承法其实看起来还是比较直观的，因为子类构造函数中有对基类构造函数的调用，似乎在语法上还比较容易被接受。可是由于没

有 new 基类，带来了不能获得 prototype 导入的属性和方法的缺陷。解决这个缺陷的方法虽然不难，可是由此而给基类编写限制一个凌驾于 JavaScript 语法规范之上的规则，可操作性不是很好。子类也不能利用 prototype 特性来导入属性和方法，因此会与基类之间存在潜在的重名覆盖问题。所以对于复杂的基类，不推荐这种继承方法，因为类复杂了，使用 prototype 可以规范类代码，使类的定义看起来比较舒服。

应用场景： 小规模类之间的继承，基类和子类的属性方法在 5～8 个。还有就是以构造函数中赋值方式导入类的属性和方法，而不用 prototype 导入的类编写习惯的时候。

在JavaScript面向对象编程中使用继承(3)

作者：鸟食轩 [http://www.cnblogs.com/birdshome]

上次讲了在使用JavaScript进行面向对象编程中，采用构造法来实现类继承的一些优缺点。下面我们接着把“原型继承法”的优缺点也讲一讲，希望大家能积极提意见并探讨其中的一些问题。

原型（prototype）是JavaScript实现面向对象编程的一个基础，但它并不是唯一的构造类的方法，我们完全可以不使用prototype而实现类的编写（把属性和方法的附加全都写在构造函数里面就行了）。不过原型除了可以为Object的子类添加新的属性和方法外，还可以为脚本环境中的内部对象继续添加原型属性和方法，比如我们最常用的给内部对象String添加Trim方法来删除字符串两端的空格，代码为：

```
String.prototype.Trim = function()
{
    return this.replace(/(^\s*)|(\s*$)/g, '');
}
```

这样我们就可以在任何的String实例中使用Trim方法了，用惯了这种原型系统，有的时候反而还觉得传统OOP没有它好。言归正传，继续讲我们的原型继承法。

原型继承法的原理

原型继承法的关键代码是其构造函数function ArrayList02()下的第一句：

```
ArrayList02.prototype = new CollectionBase();
ArrayList02.prototype.constructor = ArrayList02;
```

Ae... 把prototype都覆盖成基类的一个实例了，子类还怎么使用prototype呢？这里不要着急，反正JavaScript的对象实例都是可以动态增删属性和方法的，基类实例作为prototype不就正好等于extends（扩展）了CollectionBase吗？之后再使用XXX.prototype.xxx = function()，可以继续获得新增了属性和

方法的对象。**注意 ArrayList02.prototype 是在 ArrayList02 的构造函数外被初始化为基类的实例的。**

再来看第二句，为什么要把 ArrayList02 赋值给新 prototype 的 constructor 呢？如果不做这个赋值，当我们从 ArrayList02 的实例中通过???.prototype.constructor 构造函数，将会获得 CollectionBase。这不是我们的本意，而且这时使用 instanceof 关键字比较对象实例和对象也会出错。

原型继承法的缺陷

原型继承法有两个缺陷。

第一个是由于类的原型（prototype）实际上是一个 Object 的实例，它不能再次被实例化（它的初始化在脚本装载时已经执行完毕）。什么意思呢？我们知道在新建对象实例时，使用语句 new ArrayList02()可以看做 JavaScript 脚本引擎把 prototype 的一个浅复制作为 this 返回给类的实例（这里其实没有发生复制，只是利用浅复制这个概念来帮助理解），如果类没有附加任何原型属性和原型方法，就等于返回了一个 new Object()实例。问题就出在这里了，由于 new 对 prototype 执行的是浅复制，如果 prototype 的原型属性里有对象类型的属性，就会造成共享对象实例的问题（类似于在传统 OOP 的类定义中使用了 static 修饰符来修饰属性）。这个缺陷下面会有示例演示，避免的办法就是不要在基类中定义对象类型的属性，比如 Array、Object 和 Date 等。

第二个缺陷和上次讲的“构造继承法”的缺陷差不多，也是关于子类定义时语句顺序的。就是说代码 ArrayList02.prototype = new CollectionBase();必须在所有 prototype 定义之前执行。很简单，如果在原型属性和方法都导入完成后再执行这个语句，就等于把之前的导入全都覆盖掉了。解决办法就是按我给上篇中的那个顺序来写。

原型继承法的示例

```
<script language="JavaScript">
  document.write('原形继承法:<br>');
  var arrayList21 = new ArrayList02();
  arrayList21.Add('a');
  arrayList21.Add('b');
  arrayList21.foo();
  var arrayList22 = new ArrayList02();
  arrayList22.Add('a');
  arrayList22.Add('b');
```

```
arrayList22.Add('c');
arrayList22.foo();
</script>
```

示例运行结果为：

```
原型继承法:
[class ArrayList02]: 2: a,b
[class ArrayList02]: 3: a,b,c,a,b
```

发现问题了吧？实例 arrayList22 的 foo()居然输出了 a,b,c,a,b@_@... 这就是前面说的 prototype 对象浅复制带来的问题。不过为什么 arrayList22 中的集合计数器仍然是 3 呢？这是因为 this.m_Count 是整数类型，这种类型又叫值类型（和 C#里的值类型、对象类型概念一样的）。值类型不存在浅复制和深复制的问题，可以看成都是深复制。

小结

JavaScript 的原型继承法虽然有 prototype 浅复制那么严重的 bug，不过它却是使用得比较多的继承方式。因为我们很少在基类里定义属性，更别说对象类型的属性了，所以引发这个错误的可能性不是很大，只是它是个潜在的隐患。至于第二个缺陷，也是一个潜在 bug，只要自己定义类的时候能比较清醒，就不会犯错误。“重载”也比较简单，如果在 ArrayList02.prototype = new CollectionBase();后有重名的属性或方法导入，自动覆盖基类中的属性或方法就相当于重载了。

应用场景： 基类没有属性，至少是没有对象类型的属性。这种继承的优点是保持了子类构造函数的完整，可以不在里面添加任何和继承有关系的代码，所有继承和重载操作都由对原型（prototype）的操作来完成。

在JavaScript面向对象编程中使用继承(4)

作者：鸟食轩 [http://www.cnblogs.com/birdshome]

大家好像对 JavaSript 面向对象编程的继承不是很感兴趣，都没有什么讨论。也许是大家暂时都遇不到如此复杂的脚本开发 solution，不过以后有问题也欢迎来讨论。毕竟经典的教程是不可能包括这些高级应用的，所以我总结的东西也可能还有谬误。

今天说说脚本面向对象编程中的实例继承法，这个方法是经典论坛上介绍 JScript 面向对象编程的文章中使用的继承方法。它是怎么工作的呢？

实例继承法的原理

实例继承法的关键代码是其构造函数 function ArrayList03()中的：

```
var base = new CollectionBase();
// ...
return base;
```

其实就是在子类构造时创建基类的一个实例，然后把基类实例作为返回值实返回。这时的所谓继承操作都是对 Object 实例（基类也就是一个 Object 的派生类的实例）的动态读写，为其添加属性和方法等，根本没有涉及任何与继承相关的范畴，不过最终的效果还是让人觉得是实现了继承。

实例继承法的缺陷

这种继承法看起来还是比较清楚的，特别是如果你已理解了 JavaScript 对象的动态特性。不过这种方法最大的缺陷也是来自于对类代码书写的要求上。由于我们在子类构造函数中创建了基类实例，所以对基类的书写没有任何要求，只要是脚本引擎认为正确的类就可以了。不过子类就不能随便写了，由于子类的属性和方法通过对象的动态特性来实现，所以子类也不能使用原型属性（prototype）来实现属性和方法的导入，而必须用 inline 的方式写在子类的构造函数里，这个和构造法实现继承的限制很相似，不过前者是限制基类的书写不

能使用 prototype 属性。

实例继承法的示例

```
<script language="JavaScript">
    document.write('实例继承法:<br>');
    var arrayList31 = new ArrayList03();
    arrayList31.Add('a');
    arrayList31.Add('b');
    arrayList31.foo();
    var arrayList32 = new ArrayList03();
    arrayList32.Add('a');
    arrayList32.Add('b');
    arrayList32.Add('c');
    arrayList32.foo();
</script>
```

示例运行结果为：

```
实例继承法:
[class ArrayList03]: 2: a,b
[class ArrayList03]: 3: a,b,c
```

小结

实例继承法其实有些偷梁换柱的味道，因为这样得到的实例，针对 instanceOf 来说的话，完全是其基类的一个扩展。而子类的实例是被扔掉了的，因为 new ArrayList03()返回的是基类实例（return base;）。这完全没有了任何继承的味道，叫做类扩展还贴切些。优点是对基类的编写没有任何特殊要求，不过同样需要规定子类的写法，子类不能使用 prototype 来导入原型方法，并且在子类构造函数中创建基类实例 var base = new CollectionBase();需要在构造函数开头（即任何向 base 添加属性和方法之前）。

应用场景： 没有太经典的应用场景，不过对于基类比较复杂，而子类需要添加的属性方法很少的继承，实例法还是显得挺清晰的。特别是对于 JavaScript 对象动态扩展很熟悉的人，就更觉得明确了。

在JavaScript面向对象编程中使用继承(5)

作者：鸟食轩 [http://www.cnblogs.com/birdshome]

明天就要回老家去过年了，关于这个“在 JavaScript 面向对象编程中使用继承”的话题居然还没有说完。下面继续来看茴香豆的“茴”字第四种写法。

这“茴”字的第四种写法——附加继承法，虽然是我自己杜撰出来的，而且还有一些前面三种继承法的影子，不过这个方法不可否认地可以把前面说到继承的问题都 cut 掉。下面我们就来仔细说说到底它是为什么这么有武功和智慧？

附加继承法的原理

附加继承法的关键代码是其构造函数 ArrayList04()中的：

```
this.base = new CollectionBase();
for ( var key in this.base ) {
    if ( !this[key] ) {
        this[key] = this.base[key];
    }
}
```

这里其实给不给 this 附加一个 base 并不重要，也一点不会影响我们的这个继承方法。首先我们看到在构造函数的第一句话中，立马就 new 了一个基类实例出来，这就说明继承对基类的书写是没有任何要求的，用前面实例继承法中的说法就是，只要脚本引擎认为正确的类就都可以。我们知道构造继承法为什么有问题呢？就是因为它始终没有创建基类的实例。而原型继承法虽然创建了基类实例，不过它把积累实例直接赋给了子类的 prototype 属性，以至于对子类书写有了特殊的要求。

接下来的 for(in)循环，把基类具有的所有属性和方法都附加到子类的实例 this 中了，这也是我把这个继承方法叫附加法的原因。这一步和构造继承法的原理类似，只是构造继承法是用了 this 作用域置换的一个技巧，把这个附加的过程让基类构造函数来完成了，不过同时也给构造继承法带来基类书写的特别要求，不

能使用其 prototype 特性。当然附加法仍然是没有这个要求的。

附加继承法的 Update（升级）

```
Object.prototype.Extends = function(BaseClass)
 {
    if ( arguments.length >= 6 )
     {
        throw new Error('Only can supprot at most 5 parameters.');
     }
     var base;
     if ( arguments.length > 1 )
     {
        var arg01 = arguments[1];
        var arg02 = arguments[2];
        var arg03 = arguments[3];
        var arg04 = arguments[4];
        base = new BaseClass(arg01, arg02, arg03, arg04);
     }
     else
     {
        base = new BaseClass();
     }
     for ( var key in base )
     {
        if ( !this[key] )
        {
            this[key] = base[key];
            if ( typeof(base[key]) != 'function' )
            {
                delete base[key];
            }
         }
     }
    this.base = base;
    // base.Inherit = this;
 };
```

这样我们的继承就可以直接写成：

```
function ArrayList04()
{
```

```
    this.Extends(CollectionBase);
    // ...
}
```

同时还提供了对基类继承时，传递参数给基类的支持，比如：

```
function ListItem()
{
    this.Extends(ListItemBase, text, value);
    // ...
}
```

对于基类，会执行 new ListItemBase(text, value);这样的操作来生成基类的实例。

附加继承法的缺陷

从目前我使用的情况来看，如果不使用 override 技术来重写方法，然后还在新的方法中去调用基类的方法（这个技术我会以后再讲，因为它不影响也不属于我们今天讨论的继承方式这个话题）的话。附加法基本没有缺陷，一定要说有的话就是：使用一个 for(in)循环来进行基类的导入，语法上很 ugly:(

附加继承法的示例

```
<script language="JavaScript">
  document.write('附加继承法:<br>');
  var arrayList41 = new ArrayList04();
  arrayList41.Add('a');
  arrayList41.Add('b');
  arrayList41.foo();
  var arrayList42 = new ArrayList04();
  arrayList42.Add('a');
  arrayList42.Add('b');
  arrayList42.Add('c');
  arrayList42.foo();
</script>
```

示例运行结果为

```
附加继承法:
[class ArrayList04]: 2: a,b
```

```
[class ArrayList04]: 3: a,b,c
```

小结

附加继承法是看起来最不像继承，但却是实际使用中最 sexy（PS:这是我们 boss 对好代码的称呼）的解决方案。其 override 也非常清晰明了，只要在 this.Extends(BaseClass);语句后有同名的方法被导入子类，就会自动覆盖从基类中导入的方法，实现 override 的效果。

使用场景： anywhere, anytime, anybody...

这话似乎说大了，完美的东西一定是没有的，附加继承法也是有缺陷的，只不过这个缺陷不属于继承这个范畴，而是在对其他 OO 编程特性的模拟中出现的问题。再唐僧一下：光是类的继承和使用，附加继承法是没有任何问题的。

挣脱浏览器的束缚（1）

——前言

作者：Jeffrey Zhao [http://www.cnblogs.com/JeffreyZhao/]

最近在为某个人门户站点作优化。

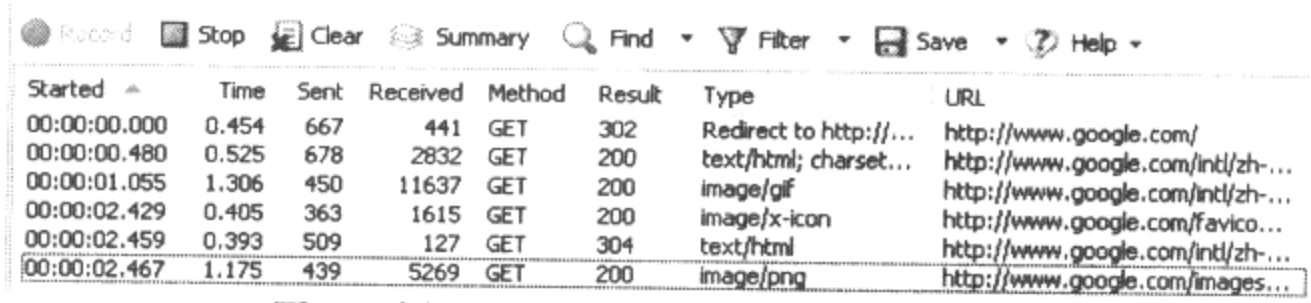
从传统意义上来说，这个站点的各方面都属中规中矩。不过作为一个以客户端为中心的 Web 应用，其性能，尤其是它的感知性能（Perceived Performance），经常会严重受制于浏览器本身。一个没有对客户端数据访问模型经过精心设计和优化的应用，其导致的结果往往就是无法充分利用带宽，让用户等待的时间变长。换句话说，其 Perceived Performance 需要进一步地提高。

突破浏览器限制，充分利用带宽，提高性能，尤其是 Perceived Performance，就是我这次优化的目的。在接下来的几篇文章里，我将以数据说话，探讨浏览器的限制，并从多个方面来谈一下这次优化的各种方式。由于该个人门户使用了 ASP.NET Ajax 进行开发，因此我也将会给出一些基于 ASP.NET Ajax 的解决方案，希望会有一定参考价值，对朋友们能有所帮助。

工具

本着实事求是的原则，我们需要使用数据来说话，于是我们也就需要一些好用的工具。它们可以帮助我们统计各种数据，以便我们进行分析和优化。

在 IE 中，我们需要使用 Http Watch 这个工具来统计页面中每个请求的信息，例如开始时间、持续长度等。利用此工具，能够轻松得出详细的数据（图 1），非常好用。而且对于我们来说，一个 Free Edition 已经足够使用了。Free Edition 虽然无法得到每次请求的所有信息，但是我们已经有了再熟悉不过的 Fiddler。我们完全可以通过那些数据使用 Excel 作出统计图表（图 2）进行分析。



Started	Time	Sent	Received	Method	Result	Type	URL
00:00:00.000	0.454	667	441	GET	302	Redirect to http://...	http://www.google.com/
00:00:00.480	0.525	678	2832	GET	200	text/html; charset...	http://www.google.com/intl/zh-...
00:00:01.055	1.306	450	11637	GET	200	image/gif	http://www.google.com/intl/zh-...
00:00:02.429	0.405	363	1615	GET	200	image/x-icon	http://www.google.com/favico...
00:00:02.459	0.393	509	127	GET	304	text/html	http://www.google.com/intl/zh-...
00:00:02.467	1.175	439	5269	GET	200	image/png	http://www.google.com/images...

图 1　访问 http://www.google.com 的统计数据

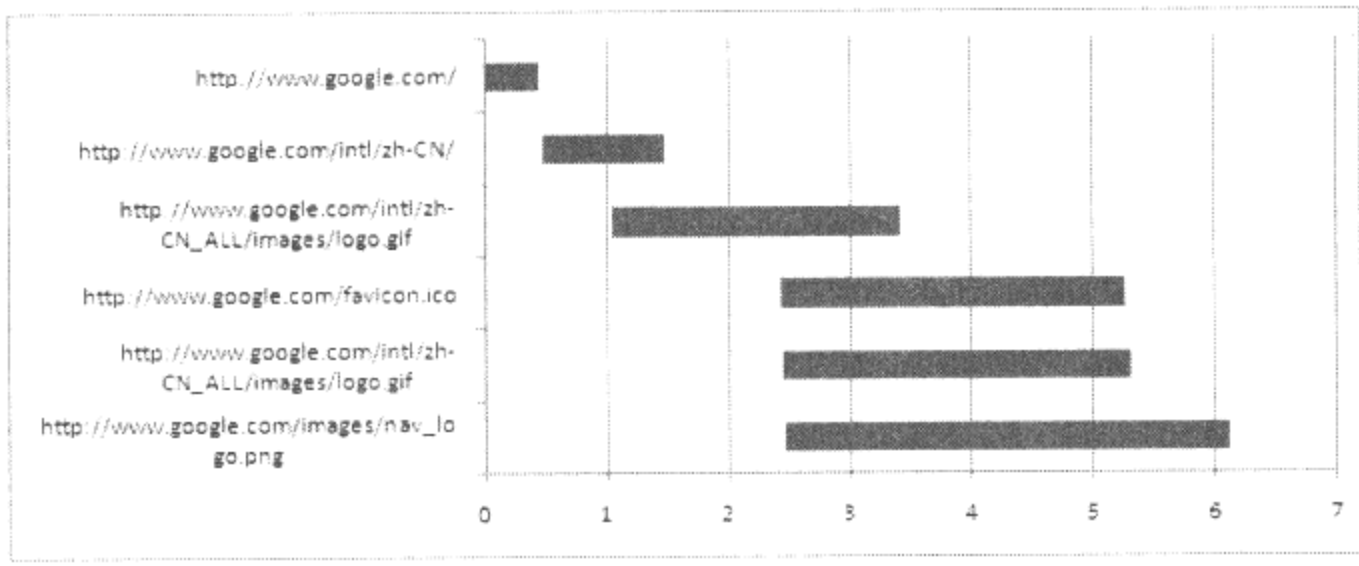


图 2　使用 Office 2007 绘制的统计图表

在 FireFox 下面，我一开始使用的是 Google Page Load Analyzer，但是发现使用起来实在不方便，它无法向 Http Watch 一样得到详细的信息，以便我们作出统计图表；而且它自动生成的示意图又非常难看，很难进行分析。后经人提醒，最新的 FireBug 也有类似的功能。装上一看，果然好用。虽然无法获得精确数据，但是它生成的示意图（图 3）已经可以直接进行分析了。

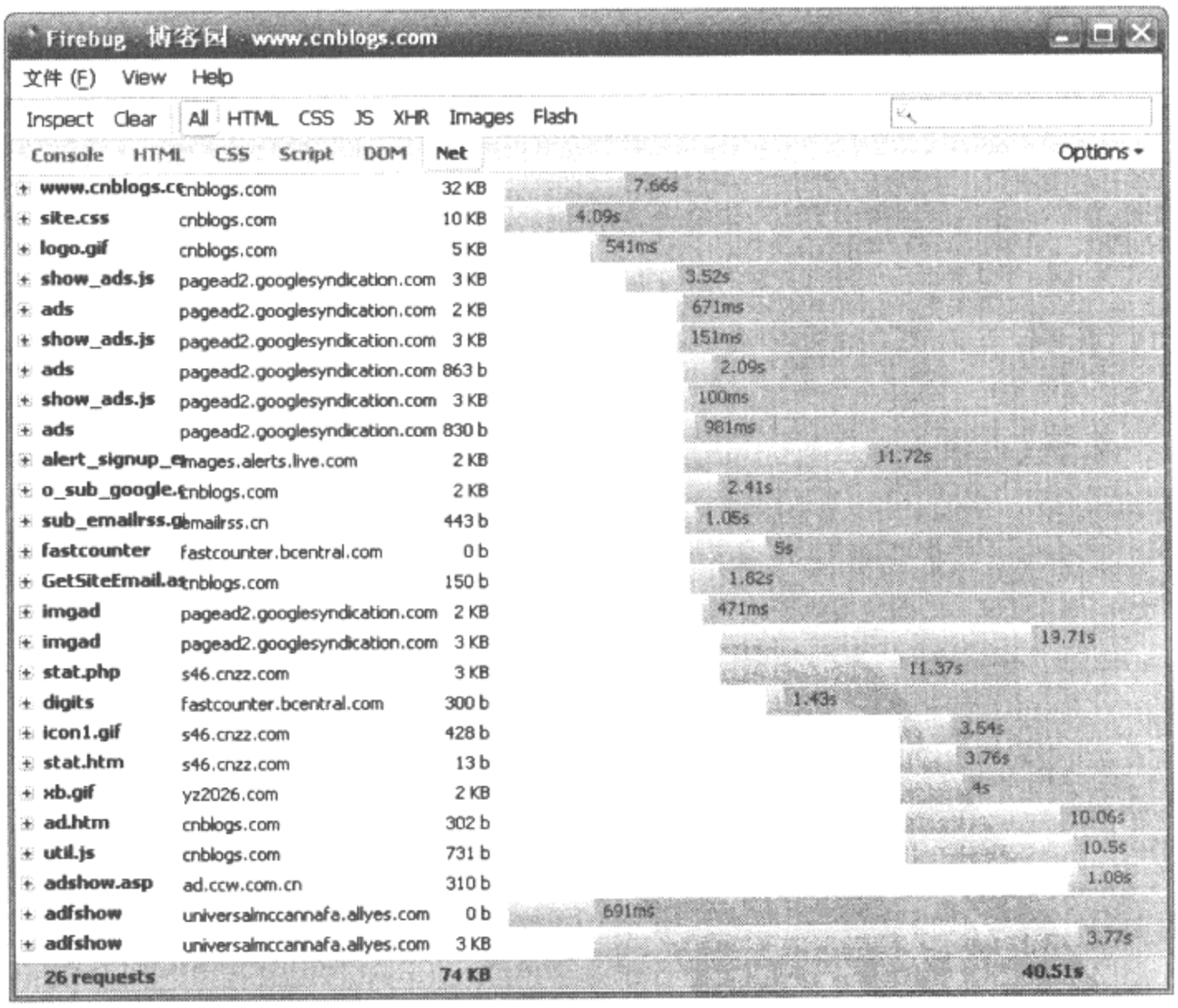


图 3　访问 http://www.cnblogs.com 时 FireBug 绘制的统计示意图

此外，为了在本地或局域网内模拟低网速的情况，我再推荐一款工具NetLimiter 2 Pro。它能够对于某个程序、进程甚至某个连接在访问网络时的带宽进行限制，无论是因特网、局域网还是本机（图 4）。最后，IE Dev Toolbar等工具自然也是必备的，我们可以在需要的时候使用它们。

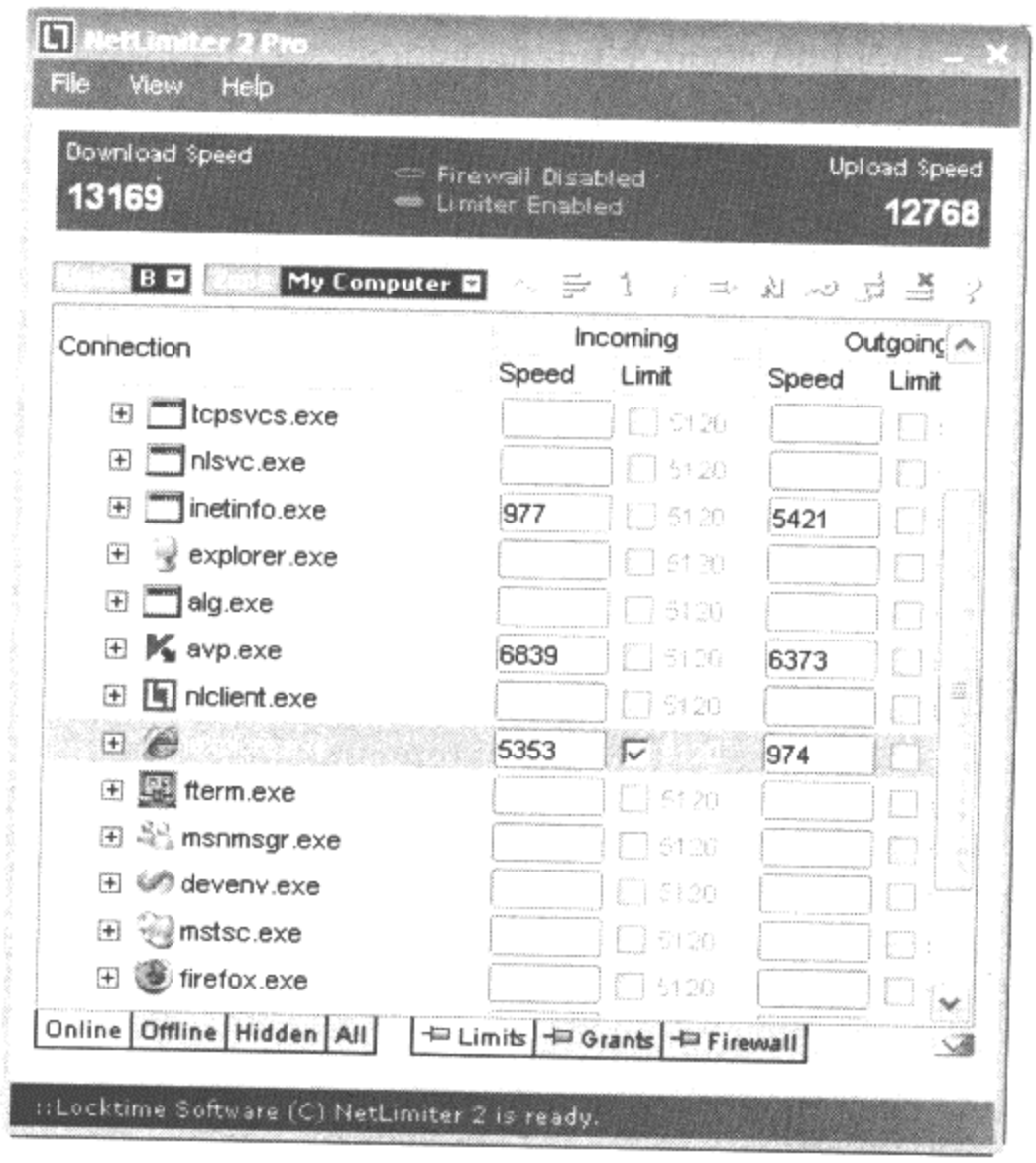


图 4 使用 NetLimiter 2 Pro 限制 IE 的带宽

有了上面这些工具，就可以开始我们的分析优化之旅了。

挣脱浏览器的束缚（2）
——别让脚本引入坏了事

作者：Jeffrey Zhao [http://www.cnblogs.com/JeffreyZhao/]

现在哪里还找得到不引入 JavaScript 脚本文件的 Web 应用？使用脚本文件的好处多多，其中最重要的可能就是提供缓存能力了。使用脚本文件之后再加上缓存，可以大大降低数据传输量，提高页面打开的速度。不过脚本文件的引入也不是简单得不值一提，我们完全有能力来优化它。

小心传统的脚本引入方式带来的性能问题

现在的 Web 应用所需的脚本越来越多，一张页面下载几百 KB 的脚本也不再是难以想象的事情了，这就直接导致页面需要更长的时间来加载脚本。传统的脚本引入方式（使用<script />）会造成什么问题？在查看这点之前，先写一个 HttpHandler 来模拟一个需要较长时间才能加载的脚本。这很简单，只需创建一个 Http Handler 来做到这一点，如下：

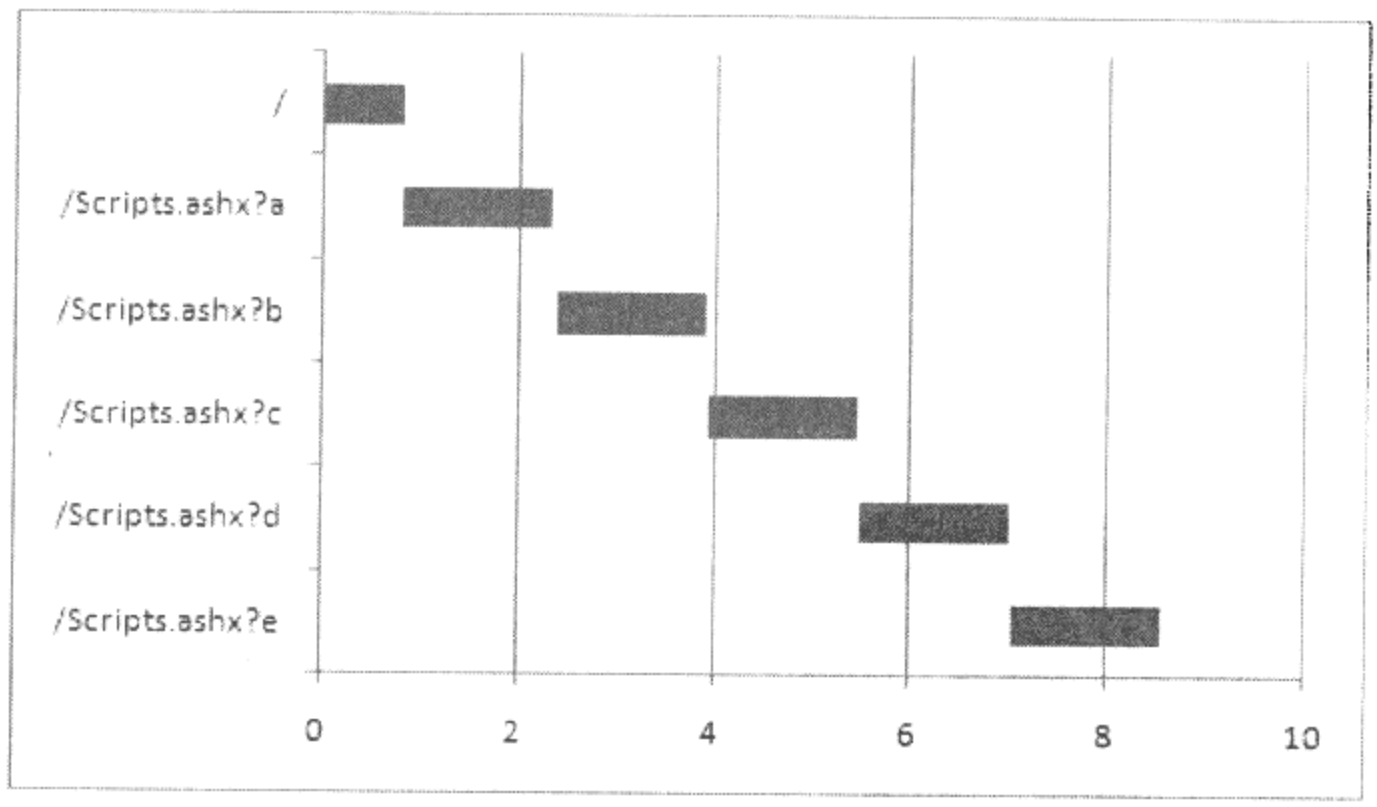
```
//Scripts.ashx
public class Scripts : IHttpHandler {

    public void ProcessRequest (HttpContext context) {
        context.Response.ContentType = "application/x-JavaScript";
        System.Threading.Thread.Sleep(1500);
        context.Response.Write("//");
    }
    public bool IsReusable {
        get {
            return false;
        }
    }
}
```

我使用 Thread.Sleep 函数使线程休眠 1.5 秒，然后输出一个注释符。这样就保证了页面加载该文件需要比较长的时间，也可以将脚本的执行时间降到最低。

然后我们就写个最简单的页面，来测试一下加载这些文件的结果：

```
//Title
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server" id="aaa">
    <title>Untitled Page</title>
    <script type="text/JavaScript" language="JavaScript" src=
"Scripts.ashx?a"></script>
    <script type="text/JavaScript" language="JavaScript" src=
"Scripts.ashx?b"></script>
    <script type="text/JavaScript" language="JavaScript" src=
"Scripts.ashx?c"></script>
    <script type="text/JavaScript" language="JavaScript" src=
"Scripts.ashx?d"></script>
    <script type="text/JavaScript" language="JavaScript" src=
"Scripts.ashx?e"></script>
</head>
<body>
    ...
</body>
</html>
```

在 IE 里打开页面，看看这些脚本加载的情况。请注意，您可以使用 IE Dev Toolbar 来禁用 Cache。



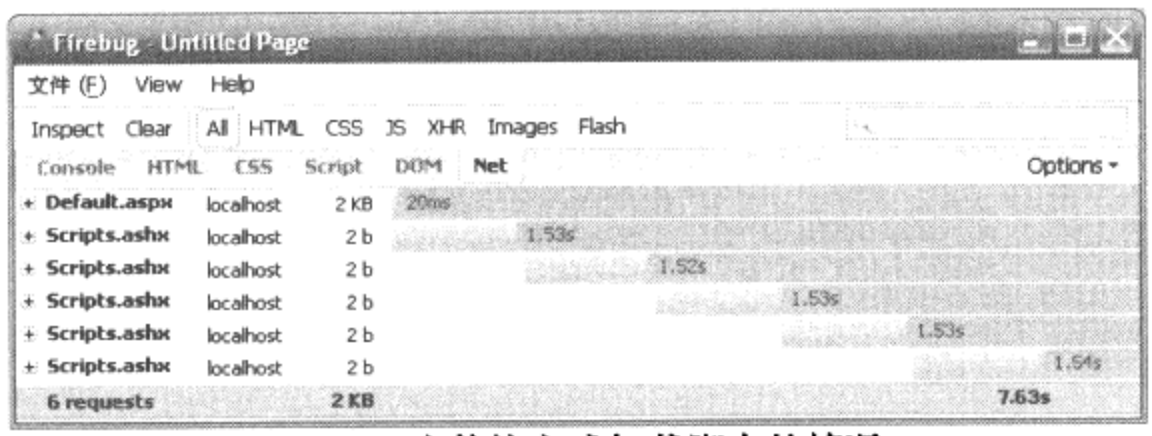
IE 中传统方式加载脚本的情况

真可谓是相当地整齐。不过整齐的背后是较低的性能：脚本文件一个一个被加载，所有脚本文件被加载完需要用 8 秒多时间。

那么 FireFox 的表现又如何？我们使用同样的页面来测试一下。

情况差不多！

其实出现这个状况是 By Design（设计上）的。从上面这个简单的例子里可能还无法看出。事实上，当浏览器遇到<script />标签时，它会开始加载脚本文件，而此时页面的其他加载行为则会全部停止，包括 HTML 的呈现、页面或图片的下载，等等。这是因为浏览器“怀疑”这些脚本文件中的一些行为可能会在页面中输出 HTML。自然，我们可以使用 document.write 方法这么做。而很多可以放在网站中的第三方小部件，都是靠脚本文件里的 document.write 方法来生成 HTML 的。



FireFox 中传统方式加载脚本的情况

这就让用户不太好受了。为什么我的浏览器只能建立一个连接？为什么不能一起下载？我们的带宽不是浪费了很多吗？这些都没错。还记得前一段时间台湾地震使一些 Blog 无法打开或者打开很慢吗？这很可能就是在页面中使用<script />引入脚本文件时造成的问题：文件下载特别慢，甚至会超时。而且当时我的 blog 也遇到这个问题。解决方案很简单，把<script />去掉便是，或者将<script />元素放置在“页尾”代码中，这样页面就会打开得比较快了。不过当然，那个文件很可能还在继续加载脚本中。

这就是提高了所谓的“感知性能（Perceived Performance）”。简单地说，就是用户“感受”到的性能。用户会发现页面已经打开了，虽然还没有完全加载完，例如 Snap Preview 还无法工作。

尝试打破传统脚本引入的瓶颈

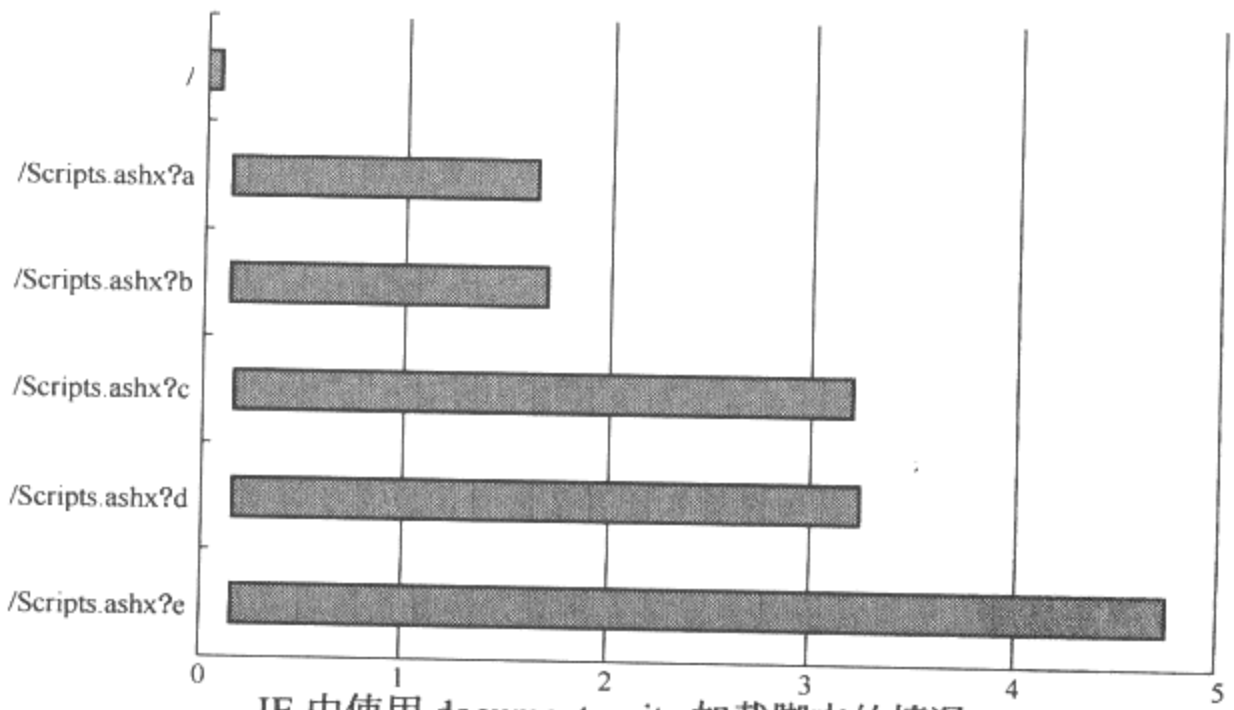
现在的脚本越做越大了，一个 200KB 的文件，如果以 20KB/s 的速度下载也要 10 秒。如果这 10 秒结束之后又来个 10 秒……这样的网页加载速度太可怕了。我们必须尝试着打破这个瓶颈。

很有趣的是，如果在页面中使用 document.write 来写一个<script/>元素的话，这些脚本就可以并行下载了。我们就用下面的代码进行尝试：

使用 document.write 来引入文件：

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server" id="aaa">
    <title>Untitled Page</title>
    <script type="text/JavaScript" language="JavaScript">
        document.write(
            '<script type="text/JavaScript" language="JavaScript"' +
            ' src="Scripts.ashx?a"><' + '/script>');
        document.write(
            '<script type="text/JavaScript" language="JavaScript"' +
            ' src="Scripts.ashx?b"><' + '/script>');
        document.write(
            '<script type="text/JavaScript" language="JavaScript"' +
            ' src="Scripts.ashx?c"><' + '/script>');
        document.write(
            '<script type="text/JavaScript" language="JavaScript"' +
            ' src="Scripts.ashx?d"><' + '/script>');
        document.write(
            '<script type="text/JavaScript" language="JavaScript"' +
            ' src="Scripts.ashx?e"><' + '/script>');
    </script>
</head>
<body>
    ...
</body>
</html>
```

这样的做法似乎有些复杂，不过应该还算直观。上面代码的目的就是在页面中“写入”<script />元素，以达到引入脚本文件的目的。还是用事实说话，先来看一下 IE 中打开页面的效果吧：



IE 中使用 document.write 加载脚本的情况

状况好多了。可以看出总是有两个脚本文件在同时下载，虽然还是受制于浏览器对于每个 Domain 只有 2 个连接的限制，但是页面加载时间已经从 8 秒多锐减到不到 5 秒了。这实在是一个绝好的消息。那么再公布一个好消息，使用这种方式引入脚本文件的话，脚本文件的执行顺序与脚本文件出现的顺序相同。我们只要安排好脚本文件的顺序，这样就可以保证脚本执行的正确性了。

不管怎么说这个方法还是非常容易使用的，不是吗？那么让我们欢呼雀跃吧，因为优化就是这么简单！

很可惜事情的发展并不如我们想象的那么单纯。我们还没有试过 FireFox 下的状况。看了 FireFox 加载页面的数据统计图，可能就会知道，我们离目标还有很大的距离。因为它的状况和原来的显示状况完全相同，document.write 这种做法在 FireFox 里没有起到任何作用。

为什么 IE 的表现和 FireFox 的表现不同呢？可能这就要问一下浏览器的开发者了，我们现在要做的，可能只是根据结果来为我们的应用想出更好的解决方案。

路漫漫其修远兮。

挣脱浏览器的束缚（3）
——两个连接还不够“并行”

作者：Jeffrey Zhao [http://www.cnblogs.com/JeffreyZhao/]

在讨论这次的主题之前，我们看一下脚本优化的另一个问题，就是“优化难度”。在这里我所说的“优化难度”是指优化一张页面时的修改难度。例如在前一篇文章中，使用 document.write 来引入脚本的话，其“优化难度”会非常的低——没有任何副作用，不用修改其他任何代码。不过它的效果似乎还不太理想，因为仅仅优化了 IE 下的体验，在 FireFox 里却没有任何作用。

很可惜，我回想了几乎所有的优化方式，再也没有找到优化难度如此低的做法了。对于其他的方式，都必须在页面的别处进行修改，优化效果越好，修改量越大。对于这些优化方式，必须编写合适的组件，将一些逻辑封装起来。这样可以在一定程度上方便使用，降低优化难度。

比较 document.write 与 defer

那么这又和 document.write 或者 defer 有什么关系？且听我慢慢道来。

<script />的 defer 属性在标准里的定义是这样的：

```
When set, this boolean attribute provides a hint to the user agent
that the script is not going to generate any document content (e.g., no
"document.write" in JavaScript) and thus, the user agent can continue
parsing and rendering.
```

JS 无法并行下载的原因就是浏览器认为在脚本中可能会输出 HTML 内容。defer 属性的作用就是告诉浏览器，脚本里不会输出任何信息。果然，当我们在 IE 里使用 defer 属性时，脚本没有被阻塞，其效果和 document.write 一样。不过在 FireFox 里依旧不行，这样的实现实在让人费解。

都说 FireFox 标准，看来在细节上也不尽然。

那么为什么我们在之前使用了 document.write 而不是 defer 属性呢？两者效果相同，但是明显使用 defer 属性更加直观啊。

defer 属性使用起来的确直观和方便。不过，效果真的相同吗？我们通

过以下的例子试试看。

document.write

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
    <script type="text/JavaScript" language="JavaScript">
        document.write(
            '<script type="text/JavaScript" language="JavaScript"' +
            ' src="Scripts.ashx?a"><' + '/script>');
        document.write(
            '<script type="text/JavaScript" language="JavaScript"' +
            ' src="Scripts.ashx?b"><' + '/script>');
        document.write(
            '<script type="text/JavaScript" language="JavaScript"' +
            ' src="Scripts.ashx?c"><' + '/script>');
    </script>
</head>
<body>
    <input type="button" value="Click" />
    <script type="text/JavaScript" language="JavaScript" src=
"Scripts.ashx?a">
        alert('Hello World');
    </script>
</body>
</html>
```

然后再使用<script defer="defer"></script>的方式引入一下。打开两个页面进行比较就会发现，如果使用 document.write 的话，在脚本加载完毕之前按钮不会显示，也不会出现提示框；而如果使用 defer 属性的话，按钮就立即出现了，也会马上出现提示。

这可麻烦了。如果页面上的元素过早出现，用户在脚本加载完之前进行操作是否会有问题？如果页面里存在直接执行的脚本（如上例的 alert 调用），在脚本文件加载完之前是否能够执行？如果上面两个问题的答案有任何一个是肯定的话，那么“恭喜您”，使用 defer 属性就会造成错误了。而且这个问题的解决方案实在不太容易找到，这大大增加了“优化难度”。

而且更为关键的是，FireFox 同样不支持 defer 属性的效果。这直接导致了 defer 属性全面落后于使用 document.write 的优化方式。既然这样，我们为什么要用它？事实上 defer 属性用得实在不多，这是个非常典型的“鸡肋”特性。

那么，哪里有使用 defer 属性的应用呢？我想应该是有的，虽然我不知道。

突破两个连接的限制

在上一篇文章里我们可以看到，虽然 document.write 方法可以让脚本文件并行加载，但是它依旧受到浏览器的限制。根据 HTTP 的标准，对于同一个 Domain，只能同时存在两个连接。在这点上，亲爱的浏览器们都乖乖地实现了。我们如果想要突破这种限制，就要增加域名。不过其实浏览器判断域名的方式是非常严格的，同一域名下的子域名，同一域名不同端口，都不算相同。一般来说，使用子域名来增加并行加载的连接数是比较常用的做法。

应该已经有不少朋友知道这个方法，它的应用实在太普遍了。不过请注意，请求任意资源时都会建立连接，浏览器对于某一域名的连接并不区分其作用。因此，无论下载图片、CSS 文件、JavaScript 文件或者是 XMLHttpRequest 对象建立的 Ajax 连接，都属于“两个连接”之内，在优化时往往需要注意这一点。另外，一个浏览器里同时建立的连接数也不是越多越好，根据实验资料显示，浏览器同时建立 6～7 个连接最为合适。因此，我们使用 3～4 个子域名是比较妥当的。

现在来看一下使用效果。在开发时要出现这个效果，我们可以修改 C:\WINDOWS\system32\drivers\etc\hosts 文件来设置本地的 DNS 映射。如下：

```
127.0.0.1 www.test.com
127.0.0.1 sub0.test.com
127.0.0.1 sub1.test.com
127.0.0.1 sub2.test.com
127.0.0.1 sub3.test.com
127.0.0.1 sub4.test.com
127.0.0.1 sub5.test.com
```

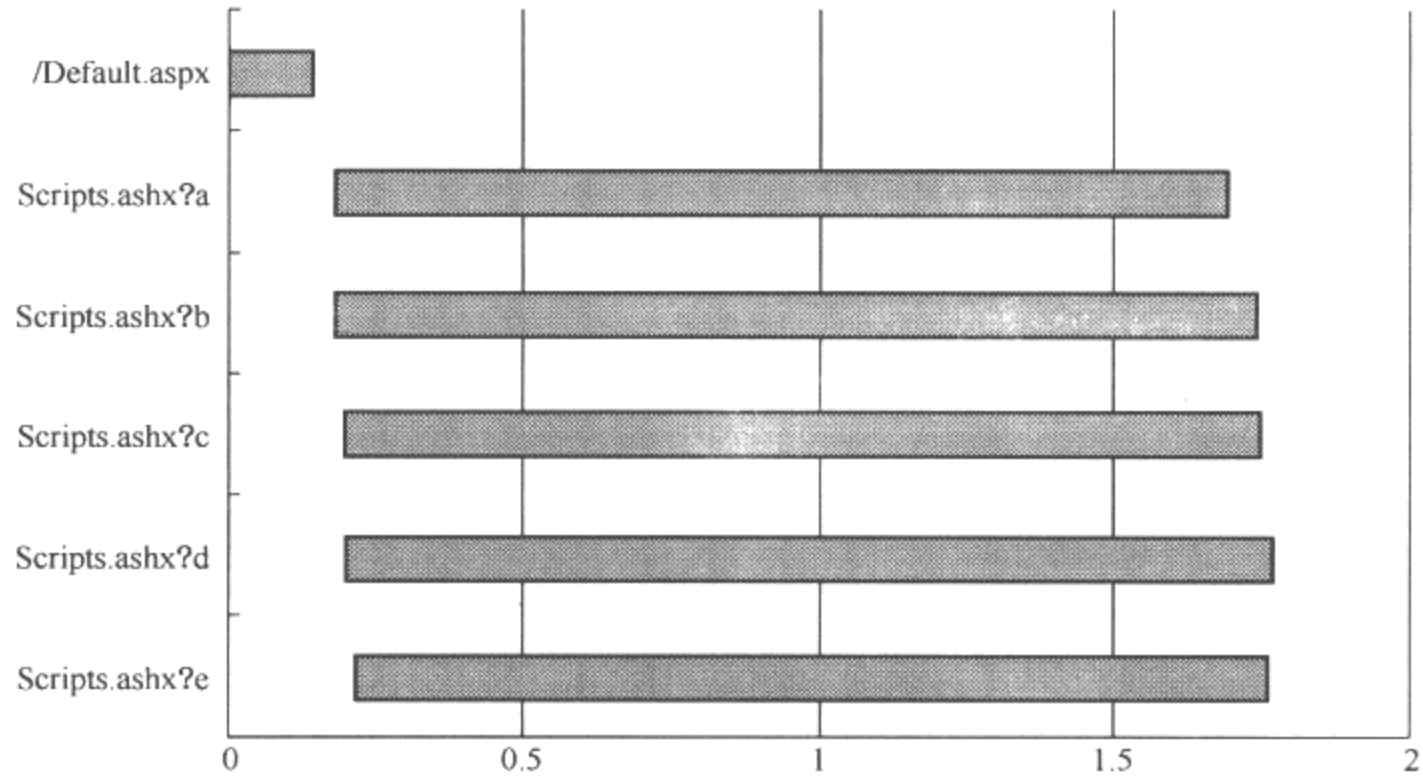
我们可以多加一些子域名，方便以后使用。

接下来我们就可以在页面里从多个不同的子域名加载脚本文件，如下：

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
    <script type="text/JavaScript" language="JavaScript">
        document.write('<script type="text/JavaScript" language=
"JavaScript"' +
            ' src="http://sub0.test.com/Scripts.ashx?a"><' + '/script>');
        document.write('<script type="text/JavaScript"
language="JavaScript"' +
            ' src="http://sub0.test.com/Scripts.ashx?b"><' + '/script>');
```

```
            document.write('<script type="text/JavaScript"
language="JavaScript"' +
                ' src="http://sub1.test.com/Scripts.ashx?c"><' + '/script>');
            document.write('<script type="text/JavaScript"
language="JavaScript"' +
                ' src="http://sub1.test.com/Scripts.ashx?d"><' + '/script>');
            document.write('<script type="text/JavaScript" language=
"JavaScript"' +
                ' src="http://sub2.test.com/Scripts.ashx?e"><' + '/script>');
        </script>
    </head>
    <body>
        ...
    </body>
    </html>
```

在浏览器中打开页面试试看？还记得当初我们加载页面用了多少时间吗？8 秒多！而现在已经能够在不到 2 秒内加载完毕了。



使用多个子域名进行并行加载

可惜我们还要优化 FireFox 浏览器里的情况，下次我们就来讨论这个问题。接下来的优化方案会有一定的难度，不过只要我们利用得当，将会大大提高 Perceived Performance。

挣脱浏览器的束缚（4）

——王道！动态添加 script 元素

作者：Jeffrey Zhao [http://www.cnblogs.com/JeffreyZhao/]

我们已经知道，脚本文件的并行下载能够提高页面的加载速度。但是目前还有一个急需解决的问题，那就是对于 FireFox 浏览器的优化。之前使用的优化方法，无论是简单实用的 document.write 还是食之无味的 defer 属性，FireFox 浏览器都对之置若罔闻。不过 FireFox 也不是绝对地“冥顽不灵”，开发人员还是有方法对它进行优化的。

这个方法就是动态添加 script 元素。

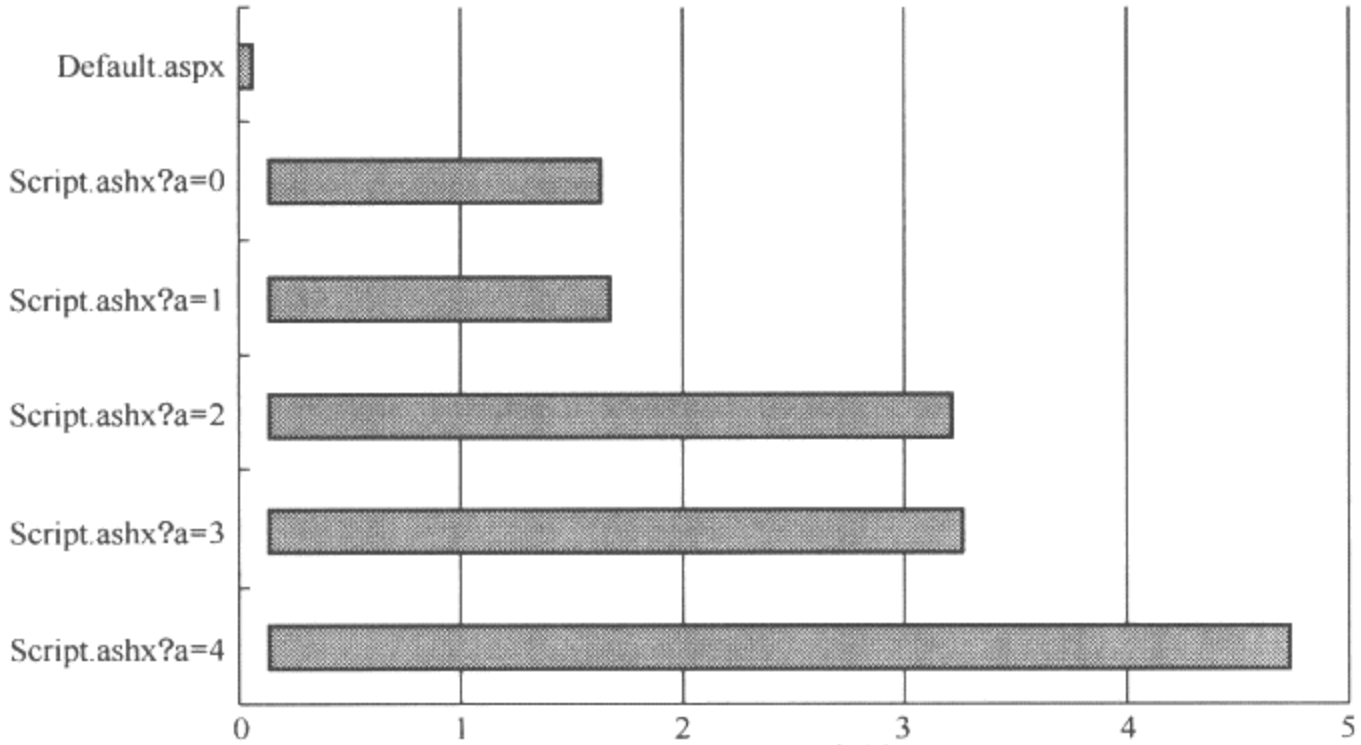
动态添加 script 元素

不知道“动态添加 script 元素”这个说法是否正确，我在这里的意思是使用 JavaScript 编程，向<head />里添加 script 元素。下面的代码动态添加了 5 个 script 元素：

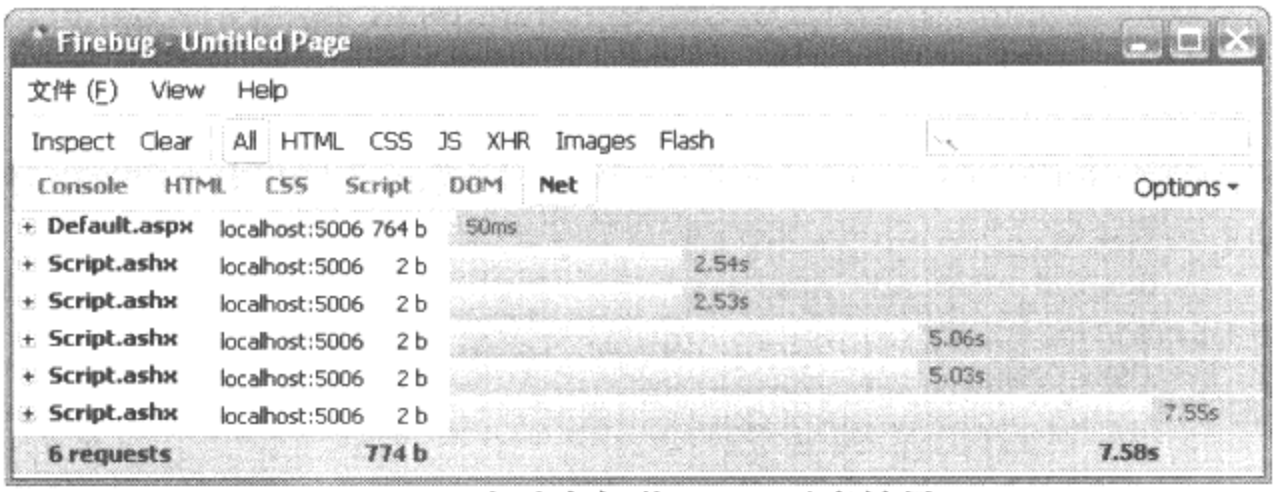
```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server" id="head">
    <title>Untitled Page</title>
    <script type="text/JavaScript" language="JavaScript">
        for (var i = 0; i < 5; i ++)
        {
            var script = document.createElement("script");
            script.type = "text/JavaScript";
            script.src = "Script.ashx?a=" + i;
            document.getElementById('head').appendChild(script);
        }
    </script>

</head>
<body>
    ...
</body>
```

```
</html>
```

请注意，由于在 JavaScript 代码执行时页面还没有加载完毕，因此还不能使用 document.getElementsByTagName 方法来获得 head 元素。我们只能为 head 元素添加一个 id，并使用 document.getElementById 方法来获得它。打开这张页面，就会发现，无论是 IE 还是 FireFox 的元素加载都会发现优化的效果：



IE 中动态加载 script 元素效果



FireFox 中动态加载 script 元素效果

我们姑且不关心为什么 FireFox 中每个脚本文件会使用 2.5 秒进行加载，但是并行加载的效果切切实实地出现了！加上多域名，效果更明显。

细心的朋友是否回想起什么了吗？没错，当年在 ASP.NET AJAX 某个版本中（我记得是 Beta 1，有些模糊了）加载自定义脚本时使用了 Sys.Application.queueScriptReference 方法，它能够让脚本文件并行加载。但是由于接下来会谈到的几个问题，ASP.NET AJAX 最终还是选择了传统的加载方式。不过 ASP.NET AJAX 还是细心地考虑到脚本加载的影响，ScriptManager 和 ScriptReference 已经提供了 LoadScriptsBeforeUI 属性，我们现在就能够控制 script 元素是出现在 UI 之前还是之后了，我们可以将影响性能但是无需“急用”的脚本放在所有 UI 的最后进行加载，以降低它对于性能的影响（这个是在刚发

布的 ASP.NET AJAX 正式版中新增的功能，我在阅读代码时无意发现的）。

再说句题外话，虽然这个脚本加载方法已经被取消了，但是功能依旧存在，因为 UpdatePanel 在 Partial Rendering 之后只能选择动态加载脚本文件。我们也能够自己使用这样的加载方式，然而这就超出了今天这篇文章讨论的范围。

动态添加 script 元素的缺陷

世界上很少有完美的事物。动态添加的 script 元素能够使 IE 和 FireFox 里都得到优化，它应该也会有些麻烦，否则为什么这个方法没有被推广呢？

而且事实上，动态添加 script 元素的做法是“优化难度”最高的方法。我现在就来一一列举它的“缺陷”。

1. 无法阻碍页面加载

其实这个问题和在 IE 中使用 defer 属性遇到的问题相同。如果您需要在页面中直接使用脚本文件里定义的函数，就不能轻易使用这个做法，即使它的确能够优化页面的加载速度。

2. 影响 window.onload 事件的触发

如果对于 window.onload 事件的触发有所影响，但是这种影响能够在不同浏览器中得到统一倒也罢了，还相对容易处理一些。现在的问题就在于，在 IE 中 window.onload 事件会在页面其他元素被加载完毕之后立即触发，而 FireFox 里的 window.onload 事件会等待动态添加的那些脚本文件也被加载完毕后才触发。虽然我们开发人员是伟大的，可是要兼容这两种情况依旧不是一件易如反掌的事情。

3. 动态加载脚本的执行顺序

这一点才是最致命的。

虽然动态加载的 script 元素是有严格顺序的，但是浏览器可不一定这样认为。在 FireFox 中，脚本文件会按照它动态加载的 script 元素的顺序执行，而 IE 会根据脚本文件下载完毕的顺序执行。

那还得了？

那么为何称之为王道？

既然麻烦这么多，为什么还称之为“王道”？其实我们只要合理地使用

这个方法，就能够大大提高页面的 Perceived Performance。

可能在这里我需要重新定义一下“Perceived Performance”的概念。它的意思是“用户感受到的性能”。我们打开一个页面，例如 Windows Live 个人主页，会发现页面的框架都被加载了，但是每一个框架都是 Loading 状态，然后每一个模块陆陆续续地加载成功。

我们来想象一下这个场景。一个页面的所有内容（包括模块）需要 20 秒钟才能加载完毕。但是它用了 10 秒钟就显示出了模块的框架，在接下来 10 秒钟内每个模块慢慢地出现。还有一种情况，就是等待整整 20 秒才能看到页面。从用户角度来看，哪个性能比较高呢?

这个就是 Perceived Performance 的经典案例。从所谓的 Web 2.0 开始，Perceived Peformance 的重要性可以说被提高到了一个前所未有的高度。

那么我们现在就用语言来简单描述一下应该如何实现这样的效果。

- 首先，在页面上用传统方式（最好使用 document.write）加载所需要的基础脚本以及所有的 HTML，这时候所有的模块处于 Loading 状态。
- 在 window.onload 事件被触发后，动态加载每个模块所需的脚本。我们只需要在 IE 浏览器中响应 script 元素的 onload 事件或者在其他浏览器中响应 script 元素的 onreadystatechange 事件，就可以捕捉脚本文件的加载情况。
- 在上述事件的 handler 中，如果 script 元素的 readyState 为 complete 或 loaded（script 元素的 readyState 使用字符串表示），那么判断某个模块需要的脚本有没有加载完毕。如果完毕，则显示那个模块的具体内容。

大体方式就是这样，逻辑非常简单，不过在编码上可能就会遇到一些问题。对于使用 ASP.NET AJAX 的开发人员可能就略有些福气了，因为 ASP.NET AJAX 内置就有动态添加脚本元素的机制，已经实现了很好的跨浏览器特性。它们能够稍稍便于我们的开发，有机会我将详细的介绍它们，并且和大家一起来设计和实现一个好用的脚本库。

我对于加载脚本文件的优化心得就只有这些了，不过我们还可以在其他方面进行优化。例如，AJAX 应用里最常见的 XMLHttpRequest 对象，我们也可以有技巧地使用它。不过这些内容，就要等下次再和大家分享了。

挣脱浏览器的束缚（5）
——哭笑不得的 IE Bug

作者：Jeffrey Zhao [http://www.cnblogs.com/JeffreyZhao/]

还记得《ASP.NET AJAX Under the Hood Secrets》[①]吗？这是我在自己的 Blog 上推荐过的唯一一篇文章（不过更可能是一时兴起）。在这片文章里，Omar Al Zabir 提出了他在使用 ASP.NET AJAX 中的一些经验。其中提到的一点就是：Browsers do not respond when more than two calls are in queue。简单说，就是在 IE 中，如果同时建立了超过两个连接在“连接状态”中，但是没有连接成功（连接成功之后就没有问题了，即使在传输数据），浏览器会停止对其他操作的响应，例如单击超级链接进行页面跳转，直到除了正在尝试的两个连接就没有其他连接时，浏览器才会重新响应用户操作。

出现这个问题一般需要 3 个条件：

- 同时建立太多连接，例如一个门户上有许多个模块，它们在同时请求服务器端数据。
- 响应比较慢，从浏览器发起连接，到服务器端响应连接，所花的时间比较长。
- 使用 IE 浏览器，无论 IE 6 还是 IE 7 都会有这个问题，而 FireFox 则一切正常。在 IE 7 里居然还有这个 bug，真是令人哭笑不得。但是我们必须解决这个问题，不是吗？

编写代码维护队列

与《ASP.NET AJAX Under the Hood Secrets》一文中一样，最容易想到的解决方案就是编写代码来维护队列。这个队列非常容易编写，代码如下（RequestQueue.js）：

```
if (!window.Global)
{
```

[①] http://www.codeproject.com/Ajax/aspnetajaxtips.asp

```
    window.Global = new Object();
}

Global._RequestQueue = function()
{
    this._requestDelegateQueue = new Array();
    this._requestInProgress = 0;
    this._maxConcurrentRequest = 2;
}

Global._RequestQueue.prototype =
{
    enqueueRequestDelegate : function(requestDelegate)
    {
        this._requestDelegateQueue.push(requestDelegate);
        this._request();
    },
    next : function()
    {
        this._requestInProgress --;
        this._request();
    },
    _request : function()
    {
        if (this._requestDelegateQueue.length <= 0) return;
        if (this._requestInProgress >= this._maxConcurrentRequest) return;

        this._requestInProgress ++;
        var requestDelegate = this._requestDelegateQueue.shift();
        requestDelegate.call(null);
    }
}

Global.RequestQueue = new Global._RequestQueue();
```

我在实现这个队列时使用了最基本的 JavaScript，可以让这个实现不依赖于任何 AJAX 类库。这个实现非常容易实现，我简单介绍一下它的使用方式。

- 在需要发起 AJAX 请求时，不能直接调用最后的方法来发起请求。需要封装一个 delegate 然后放入队列。
- 在 AJAX 请求完成时，调用 next 方法，可以发起队列中的其他请求。

例如，在使用 prototype 1.4.0 版时可以这样：

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>Request Queue</title>
    <script type="text/JavaScript"
src="js/prototype-1.4.0.js"></script>
    <script type="text/JavaScript"
src="js/RequestQueue.js"></script>

    <script language="JavaScript" type="text/JavaScript">
        function requestWithoutQueue()
        {
            for (var i = 0; i < 10; i++)
            {
                new Ajax.Request(
                    url,
                    {
                        method: 'post',
                        onComplete: callback
                    });
            }

            function callback(xmlHttpRequest)
            {
                ...
            }
        }

        function requestWithQueue()
        {
            for (var i = 0; i < 10; i++)
            {
                var requestDelegate = function()
                {
                    new Ajax.Request(
                        url,
                        {
                            method: 'post',
                            onComplete: callback,
                            onFailure: Global.RequestQueue.next,
                            onException: Global.RequestQueue.next
                        });
                }
```

```
                Global.RequestQueue.enqueueRequestDelegate
(requestDelegate);
            }

            function callback(xmlHttpRequest)
            {
                ...
                Global.RequestQueue.next();
            }
        }
    </script>
</head>
<body>
    ...
</body>
</html>
```

在上面的代码中，requestWithoutQueue 方法发起了普通的请求，requestWithQueue 则使用了 Request Queue，大家可以比较一下它们的区别。

使用 Request Queue 的缺陷

这个 Request Queue 能够工作正常，但是使用起来实在不方便。为什么？

我们来想一下，如果一个应用已经写得差不多了，现在需要在页面里使用这个 Request Queue，我们需要怎么做？我们需要修改所有发起请求的地方，改成使用 Request Queue 的代码，也就是建立一个 Request Delegate。而且我们需要把握所有的异常情况，保证在出现错误时，Global.RequestQueue.next 方法也能够被及时地调用，否则这个队列就无法正常工作了。还有 ASP.NET AJAX 中有 UpdatePanel，该怎么建立 Request Delegate？该如何访问 Global.RequestQueue.next 方法？

可怜的 JavaScript，太容易受骗了

需要找出一种方式，能够轻易地用在已有的应用中，解决已有应用中的问题。怎么样才能让已有应用修改尽可能地少呢？我们来想一个最极端的情况：一行代码都不用改，这可能么？

似乎是可能的，我们只需要“骗”过 JavaScript 就可以。可怜的 JavaScript，太容易骗了。直接来看代码，一目了然：

FakeXMLHttpRequest.js

```
window._progIDs = [ 'Msxml2.XMLHTTP', 'Microsoft.XMLHTTP' ];
if (!window.XMLHttpRequest)
{
    window.XMLHttpRequest = function()
    {
        for (var i = 0; i < window._progIDs.length; i++)
        {
            try
            {
                var xmlHttp = new _originalActiveXObject(window._
progIDs[i]);
                return xmlHttp;
            }
            catch (ex) {}
        }

        return null;
    }
}

if (window.ActiveXObject)
{
    window._originalActiveXObject = window.ActiveXObject;

    window.ActiveXObject = function(id)
    {
        id = id.toUpperCase();

        for (var i = 0; i < window._progIDs.length; i++)
        {
            if (id === window._progIDs[i].toUpperCase())
            {
                return new XMLHttpRequest();
            }
        }

        return new _originaActiveXObject(id);
    }
}
```

```
window._originalXMLHttpRequest = window.XMLHttpRequest;

window.XMLHttpRequest = function()
{
    this._xmlHttpRequest = new _originalXMLHttpRequest();
    this.readyState = this._xmlHttpRequest.readyState;
    this._xmlHttpRequest.onreadystatechange =
        this._createDelegate(this, this._internalOnReadyStateChange);
}

window.XMLHttpRequest.prototype =
{
    open : function(method, url, async)
    {
        this._xmlHttpRequest.open(method, url, async);
        this.readyState = this._xmlHttpRequest.readyState;
    },

    send : function(body)
    {
        var requestDelegate = this._createDelegate(
            this,
            function()
            {
                this._xmlHttpRequest.send(body);
                this.readyState = this._xmlHttpRequest.readyState;
            });

        Global.RequestQueue.enqueueRequestDelegate(requestDelegate);
    },

    setRequestHeader : function(header, value)
    {
        this._xmlHttpRequest.setRequestHeader(header, value);
    },

    getResponseHeader : function(header)
    {
        return this._xmlHttpRequest.getResponseHeader(header);
    },

    getAllResponseHeaders : function()
    {
```

```
        return this._xmlHttpRequest.getAllResponseHeaders();
    },

    abort : function()
    {
        this._xmlHttpRequest.abort();
    },

    _internalOnReadyStateChange : function()
    {
        var xmlHttpRequest = this._xmlHttpRequest;

        try
        {
            this.readyState = xmlHttpRequest.readyState;
            this.responseText = xmlHttpRequest.responseText;
            this.responseXML = xmlHttpRequest.responseXML;
            this.statusText = xmlHttpRequest.statusText;
            this.status = xmlHttpRequest.status;
        }
        catch(e){}

        if (4 === this.readyState)
        {
            Global.RequestQueue.next();
        }

        if (this.onreadystatechange)
        {
            this.onreadystatechange.call(null);
        }
    },

    _createDelegate : function(instance, method)
    {
        return function()
        {
            return method.apply(instance, arguments);
        }
    }
}
```

本来在想出这个解决方案时，我心中还比较忐忑，担心这个方法的可行

性。当真正完成时，可真是欣喜不已。这个解决方案的关键就在于“伪造JavaScript 对象”。JavaScript 只会直接根据代码来使用对象，我们如果将一些原生对象保留起来，并且提供一个同名的对象，JavaScript 就会使用你提供的伪造的 JavaScript 对象了。在上面的代码中，主要“伪造”了两个对象：

- **window.XMLHttpRequest** 对象。我们将 XMLHttpRequest 原生对象保留为 window._originalXMLHttpRequest，并且提供一个新的（或者说是伪造的）window.XMLHttpRequest 类型。在新的 XMLHttpRequest 对象中，我们封装了一个原生的 XMLHttpRequest 对象，同时也会定义 XMLHttpRequest 原生对象存在的所有方法和属性，大多数的方法都会委托给原生 XMLHttpRequest 对象（例如 abort 方法）。需要注意的是，我们在新的 XMLHttpRequest 类型的 send 方法中，创造了一个 delegate 放入队列中，并且_internalOnReadyStateChange 方法在合适的情况下（readyState 为 4，表示 completed）调用 Global.RequestQueue.next 方法，然后再触发 onreadystatechange 的 handler。
- **ActiveXObject** 对象。由于类库创建 XMLHttpRequest 对象的实现不同，有的类库会首先使用 ActiveX 进行尝试（例如 prototype），有些则会首先尝试 window.XMLHttpRequest 对象（例如 Yahoo! UI Library），因此我们必须保证在通过 ActiveX 创建 XMLHttpRequest 对象时也能够使用我们伪造的 window.XMLHttpRequest 类。实现相当简单：保留原有的 window.ActiveXObject 对象，在通过新的 window.ActiveXObject 创建对象时判断传入的 id 是否为 XMLHttpRequest 所需的 id，如果是，则返回伪造的 window.XMLHttpRequest 对象，否则使用原来的 ActiveXObject（保存在 window._originaActiveXObject 变量里）创建所需的 ActiveX 控件。

其实“骗取”JavaScript 的“信任”非常简单，这也是 JavaScript 灵活的体现。在扩展一个 JS 类库时，完全可以想一下，是否能够使用一些“巧妙”的办法来改变原有的逻辑呢？

“伪造”XMLHttpRequest 对象的优点与缺点

现在，要在已有的应用中修改浏览器僵死的状况太容易了，只需在 IE 浏览器中引入 RequestQueue.js 和 FakeXMLHttpRequest.js 即可。而且我们只需要把“判断”浏览器类型的任务交给浏览器本身就行了，实现一个队列，如下：

```
<!--[if IE]>
    <script type="text/JavaScript" src="js/RequestQueue.js"></script>
```

```
        <script type="text/JavaScript" src="js/FakeXMLHttpRequest.js">
</script>
    <![endif]-->
```

这样，只有在 IE 浏览器中这两个文件才会被下载，何其容易！

那么，这么做会有什么缺点呢？可能最大的缺点，就是伪造的对象无法完全模拟 XMLHttpRequest 的“行为”。如果在服务器完全无法响应时，访问 XMLHttpRequest 的 status 会抛出异常。请注意，这里说的“完全无法响应”不是指 Service Unavailable（很明显，它的 status 是 503），而是彻底访问不到，比如机器的网络连接断了。而在伪造的 XMLHttpRequest 中，status 无法模拟一个方法调用（IE 没有 FireFox 里的__setter__），因此无法抛出异常。

这个问题很严重吗？个人认为没有什么问题。看看常见的类库封装，都是直接访问 status，而不会判断它到底会不会出错。这也说明，这个状况本身已经被那些类库忽略了。

那么我们也忽略一下吧，这个解决方案还是比较让人满意的。至少目前看来，在使用过程中没有出现问题。我们的“欺骗”行为没有被揭穿，异常成功。:)

封面设计：王永瑞/邢晓静

半年前，在博客园这杆大旗下的感召下，一群园友聚到了一起，组成了博客园精华集编委会，酝酿着一出好戏。

这期间，有过争执，但更多的是合作。长期的磨合，使得来自五湖四海的编委会成员互相熟悉，甚至无话不谈。这，也是一种财富。

今天，博客园这坛好酒，在陈酿了五年之后，终于要开封了。

这是一本关于Web标准、前端编程、网站优化的集大成之作，代表了博客园在Web领域的最高水准。所选文章大多是经验之谈，所谓技术人士的“心灵老鸭汤”：全书最有撼力的文章，莫过于Cat Chen的《欲练 CSS ，必先宫 IE》；鸟食轩和老赵的文章偏实用，主要集中在JavaScript领域；爆牙齿的《重构之美》系列，则有“采菊东篱下，悠然见南山”的意境。

妙笔生花之作不胜枚举，正所谓：备美酒兮以飨佳朋，身心俱醉。

博客园推荐

ISBN 978-7-115-20897-2

9 787115 208972 >

ISBN 978-7-115-20897-2/TP

定价：35.00 元

分类建议：计算机 / Web设计

人民邮电出版社：www.ptpress.com.cn