



Web前端开发者的内功修炼秘笈

4大社区鼎力推荐！

實戰



曹刘阳 著

*Writing Solid Web front-end Code*

# 编写高质量代码<sup>①</sup>

## Web前端开发修炼之道



机械工业出版社  
China Machine Press



互联网进入Web 2.0时代以后，Web应用敲响了传统桌面应用的丧钟，它一路摧城拔寨，如今几乎所有的应用都打上了“Web”的烙印。与之相应的，Web开发技术得到了空前的发展，尤其是前端技术。近年来，随着用户对使用体验的要求越来越高，前端开发的技术难度越来越大，昔日设计和制作不分的网页设计师这一职位终于“拆分”成了视觉设计师和前端开发工程师两个职位，分别向着艺术和技术的方向纵深发展。

Web前端开发工程师是一个很新的职业，在国内乃至国际上真正开始受到重视的时间也不超过5年，这类专业人才一直供不应求。从知识体系上讲，前端开发工程师需要掌握和了解的知识非常之多，甚至可以用庞杂来形容。作为一名没有太多经验的前端开发工程师，我们应该如何去全面认识自己的工作，如何找准自己的定位，如何从合格成为优秀，最后迈向卓越？本书尝试从如何编写易于维护的、高质量的Web前端代码的角度给出答案。

### 如果你在思考下面这些问题，也许本书就是你想要的！

- 作为一名合格的Web前端开发工程师，究竟需要具备哪些技能和素质？为什么说如果要精通Web前端开发这一行，必须先精通十行？
- 在Web应用的实现代码中，有哪些技术因素会导致应用难以维护？
- 高质量的Web前端代码应该满足哪些条件？如何才能提高Web前端代码的可读性和可重用性？
- 在HTML代码中，为什么要使用语义化标签？如何检查你使用的标签是否语义良好？语义化标签时应该注意哪些问题？
- 如何编写CSS代码和JavaScript代码可以避免团队合作时产生冲突？
- 如何组织CSS文件才能让它们更易于管理？如何让CSS模块化，从而提高代码的重用率？CSS的命名应该注意哪些问题？何谓优良的CSS编码风格？
- 如何在CSS编码中引入面向对象的编程思想？这样做有哪些好处？
- 原生JavaScript和JavaScript类库之间有何关系？如何编写自己的JavaScript类库？
- JavaScript有哪些常见的跨浏览器兼容问题？如何解决这些问题？
- 如何组织JavaScript才能让代码的结构更清晰有序，从而更易于维护？如何才能编写出弹性良好的JavaScript代码？编写过程中应该注意哪些问题？
- JavaScript的面向对象编程是如何实现的？如何用面向对象的方式重写原有的代码？
- 编写高质量的JavaScript代码有哪些实用的技巧？又有哪些常见的问题需要注意？
- 为了提高Web前端代码的可维护性，我们应该遵循哪些规范？

客服热线：(010) 88378991, 88361066  
购书热线：(010) 68326294, 88379649, 68995259  
投稿热线：(010) 88379604  
读者信箱：hzjsj@hzbook.com



华章网站 <http://www.hzbook.com>

网上购书：[www.china-pub.com](http://www.china-pub.com)

封面设计：陈子平

上架指导：计算机/程序设计

ISBN 978-7-111-30595-8



9 787111 305958

定价：49.00元

實  
戰



# 编写高质量代码<sup>①</sup>

## Web前端开发修炼之道

曹刘阳 著



机械工业出版社  
China Machine Press

本书以网站重构为楔子，深刻而直接地指出了 Web 前端开发中存在的一个重要问题——代码难以维护。如何才能提高代码的可维护性？人是最关键的因素！于是本书紧接着全方位地解析了作为一名合格的前端开发工程师应该掌握的技能和承担的职责，这对刚加入前端开发这一行的读者来说有很大的指导意义。同时，还解读了制定规范和团队合作的重要性。

本书的核心内容是围绕 Web 前端开发的三大技术要素——HTML、CSS 和 JavaScript 来深入地探讨编写高质量的 HTML 代码、CSS 代码和 JavaScript 代码的方法、技巧、规范和最佳实践，从而为编写易于维护的 Web 前端代码打下坚实的基础。这不是一本单纯的“技术”书籍，没有系统地讲解 Web 前端开发的基础知识，它更专注于“技巧”，探索如何为“技术”提供最佳“技巧”。

本书包含了大量的开发思想和原则，都是作者在长期开发实践中积累下来的经验和心得，不同水平的 Web 前端开发者都会从中获得启发。尤其是对于那些中初级水平的读者而言，本书是一本不可多得的内功修炼秘籍。

**封底无防伪标均为盗版**

**版权所有，侵权必究**

**本书法律顾问 北京市展达律师事务所**

### **图书在版编目（CIP）数据**

**编写高质量代码：Web 前端开发修炼之道 / 曹刘阳著. —北京：机械工业出版社，2010.5**

**ISBN 978-7-111-30595-8**

**I . 编… II . 曹… III . 主页制作-代码-程序设计 IV . TP393.092**

**中国版本图书馆 CIP 数据核字（2010）第 082612 号**

**机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）**

**责任编辑：陈佳媛**

**北京京师印务有限公司印刷**

**2010 年 6 月第 1 版第 1 次印刷**

**186mm×240mm • 18.75 印张（含 2.5 印张彩插）**

**标准书号：ISBN 978-7-111-30595-8**

**定价：49.00 元**

**凡购本书，如有缺页、倒页、脱页，由本社发行部调换**

**客服热线：(010) 88378991；88361066**

**购书热线：(010) 68326294；88379649；68995259**

**投稿热线：(010) 88379604**

**读者信箱：hzjsj@hzbook.com**

## Foreword 推荐序

Web 前端开发是从网页制作演变而来的，名称上有很明显的时代特征。在互联网的演化进程中，网页制作是 Web 1.0 时代的产物，那时网站的主要内容都是静态的，用户使用网站的行为也以浏览为主。2005 年以后，互联网进入 Web 2.0 时代，各种类似桌面软件的 Web 应用大量涌现，网站的前端由此发生了翻天覆地的变化。网页不再只是承载单一的文字和图片，各种富媒体让网页的内容更加生动，网页上软件化的交互形式为用户提供了更好的使用体验，这些都是基于前端技术实现的。

以前会 Photoshop 和 Dreamweaver 就可以制作网页，现在只掌握这些已经远远不够了。无论是开发难度上，还是开发方式上，现在的网页制作都更接近传统的网站后台开发，所以现在不再叫网页制作，而是叫 Web 前端开发。Web 前端开发在产品开发环节中的作用变得越来越重要，而且需要专业的前端工程师才能做好，这方面的专业人才近两年来备受青睐。Web 前端开发是一项很特殊的工作，涵盖的知识面非常广，既有具体的技术，又有抽象的理念。简单地说，它的主要职能就是把网站的界面更好地呈现给用户。

如何才能做得更好呢？

第一，必须掌握基本的 Web 前端开发技术，其中包括：CSS、HTML、DOM、BOM、Ajax、JavaScript 等，在掌握这些技术的同时，还要清楚地了解它们在不同浏览器上的兼容情况、渲染原理和存在的 Bug。

第二，在一名合格的前端工程师的知识结构中，网站性能优化、SEO 和服务器端

的基础知识也是必须掌握的。

第三，必须学会运用各种工具进行辅助开发。

第四，除了要掌握技术层面的知识，还要掌握理论层面的知识，包括代码的可维护性、组件的易用性、分层语义模板和浏览器分级支持，等等。

可见，看似简单的网页制作，如果要做得更好、更专业，真的是不简单。这就是前端开发的特点，也是让很多人困惑的原因。如此繁杂的知识体系让新手学习起来无从下手，对于老手来说，也时常不知道下一步该学什么。

目前市面上关于 Web 前端开发的书主要都是针对单一技术的，本书与这些书有着本质的区别。它主要想实现两个目标：第一，为不太有经验的 Web 前端开发工程师建立大局观，让他们真正了解和理解这个职业；第二，帮助有一定 Web 前端开发经验的工程师修炼内功，通过编写高质量的代码来提高前端代码的可维护性。这是很多前端开发工程师感兴趣的内容。

本书的前两章讨论了网站重构和团队合作，这是很有必要的。网站重构的目的仅仅是为了让网页更符合 Web 标准吗？不是！重构的本质应该是构建一个前端灵活的 MVC 框架，即 HTML 作为信息模型（Model），CSS 控制样式（View），JavaScript 负责调度数据和实现某种展现逻辑（Controller）。同时，代码需要具有很好的复用性和可维护性。这是高效率、高质量开发以及协作开发的基础。建立了这种大局观后，学习具体技术的思路就更清晰了。

代码质量是前端开发中应该重点考虑的问题之一。例如，实现一个网站界面可能会有无数种方案，但有些方案的维护成本会比较高，有些方案会存在性能问题，而有些方案则更易于维护，而且性能也比较好。这里的关键影响因素就是代码质量。CSS、HTML、JavaScript 这三种前端开发语言的特点是不同的，对代码质量的要求也不同，但它们之间又有着千丝万缕的联系。本书中包含着很多开发的思想和经验，都是在长期的开发实践中积累下来的，不同水平的 Web 前端工程师都会从中获得启发。

张克军（著名 Web 前端开发工程师）

2010 年 4 月

## Praise 赞誉

这是一本能修炼前端开发人员内功的书，没有过多的名词解释和理论讲述，都是来自实战中的经验。本书强调的团队合作和开发习惯对于每一个前端开发团队来说都非常重要，尤其是大公司的前端开发团队，所有成员都应该了解一下这些内容。对于所有前端开发人员而言，本书是不可或缺的参考书。

——马波 百度前端开发组项目经理

组织前端技术交流活动是作者最大的喜好之一，对前端技术痴迷，乐于分享。本书由浅入深地讲解了开发高质量的、易于维护的 Web 前端应用的技巧和注意事项。是辅助初级 Web 前端开发者进阶的一本好书，强烈推荐。

——赵立元 新浪产品部前端开发技术经理

前端开发工程师是一个易学难精的职业，很多企业常常抱怨招聘不到合适的人才。市面上此类书籍质量参差不齐，要么是泛泛而谈的理论知识，要么是用大量简单的代码示例和插图充数。本书的作者是一位有着丰富经验的资深前端开发工程师，无论是他的理论功底，还是实践经验，你都能在本书中有所体会。本书适合于中初级前端开发工程师、技术经理和项目经理阅读，选择此书一定不会让你失望。

——罗金 网易的前端开发项目经理

对于经验不太丰富的 Web 前端开发者而言，本书可谓不可多得。书中丰富的编写高质量前端代码的技巧是在大量实践和总结中结出的硕果，极具学习和参考价值。

——周裕波 百度前端研发工程师、Web 标准化交流会发起人之一

如今的 Web 应用正朝着规模化和复杂化的方向发展，应用的可维护性变得越来越重要。本书从编写高质量代码的角度探讨了如何提高 Web 前端代码的可维护性，包含大量难得的编程技巧，值得学习和研究。

—— Ajax 中国 ([www.okajax.com](http://www.okajax.com))

有经验的 Web 前端开发工程师都知道，如果要精通这一行，必须先“精通”十行。在 Web 前端开发工程师近乎庞杂的知识体系中，HTML、CSS、JavaScript 这三大技术是最基本的，也是最核心的。本书没有系统地介绍这三个方面的基础知识，重点是讲解了大量编写高质量 HTML、CSS 和 JavaScript 代码的技巧，旨在提高中初级开发者的水平。

—— jQuery 中文社区 (<http://bbs.jquery.org.cn>)

如果你只是初窥 Web 前端开发的门径，强烈建议你阅读本书，它能帮助你修炼内功。它不仅从知识体系的角度为准备从事 Web 前端开发工作的朋友指明了学习方向，而且还从技术的角度给出了大量的技巧和最佳实践。

—— 一起 Ext (<http://www.17ext.com/>)

HTML、CSS、JavaScript 是 Web 前端开发者必须精通的三把利器，本书是关于这三把利器的使用秘籍，如能善加利用，必定例无虚发。强烈推荐！

—— CSS 中文社区

## Preface 前言

前端开发工程师是一个很新的职业，在国内乃至国际上真正开始受到重视的时间不超过 5 年。但是，随着 Web 2.0 概念的普及和 W3C 组织的推广，网站重构的影响力正以惊人的速度增长。XHTML+CSS 布局、DHTML 和 Ajax 像一阵旋风，铺天盖地席卷而来，包括新浪、搜狐、网易、腾讯、淘宝等在内的各种规模的 IT 企业都对自己的网站进行了重构。

为什么它们会把自己的网站进行重构呢？有两个方面的原因：

第一，根据 W3C 标准进行重构后，可以让前端的代码组织更有序，显著改善网站的性能，还能提高可维护性，对搜索引擎也更友好；

第二，重构后的网站能带来更好的用户体验，用 XHTML+CSS 重新布局后的页面，文件更小，下载速度更快。

DHTML 可以让用户的操作更炫，更吸引眼球；Ajax 可以实现无刷新的数据交换，让用户的操作更流畅。对于普通用户来说，一个网站是否专业、功能是否强大，服务器端是用 J2EE+Oracle 的强大组合，还是用 ASP+Access 的简单组合，并没有太明显的区别。但是，前端的用户体验却给了用户直观的印象。

随着人们对用户体验的要求越来越高，前端开发的技术难度越来越大，前端开发工程师这一职业终于从设计和制作不分的局面中独立出来。前端开发技术包括三个要素：HTML、CSS 和 JavaScript，但随着 RIA 的流行和普及，Flash/Flex、Silverlight、XML 和服务器端语言也是前端开发工程师应该掌握的。前端开发工程师既要与上游的交互设计

师、视觉设计师和产品经理沟通，又要与下游的服务器端工程师沟通，需要掌握的技能非常多。这就从知识的广度上对前端开发工程师提出了要求。如果要精于前端开发这一行，也许要先精十行。然而，全才总是少有的。所以，对于不太重要的知识，我们只需要“通”即可。但“通”到什么程度才算够用呢？对于很多初级前端开发工程师来说，这个问题是非常令人迷惑的。

前端开发的门槛其实非常低，与服务器端语言先慢后快的学习曲线相比，前端开发的学习曲线是先快后慢。所以，对于从事 IT 工作的人来说，前端开发是个不错的切入点。也正因为如此，前端开发领域有很多自学成“才”的同行，但大多数人都停留在会用的阶段，因为后面的学习曲线越来越陡峭，每前进一步都很难。另一方面，正如前面所说，前端开发是个非常新的职业，对一些规范和最佳实践的研究都处于探索阶段。总有新的灵感和技术不时闪现出来，例如 CSS sprite、负边距布局、栅格布局等；各种 JavaScript 框架层出不穷，为整个前端开发领域注入了巨大的活力；浏览器大战也越来越白热化，跨浏览器兼容方案依然是五花八门。为了满足“高可维护性”的需要，我们需要更深入、更系统地去掌握前端知识，这样才可能创建一个好的前端架构，保证代码的质量。

一位好的前端开发工程师在知识体系上既要有广度，又要要有深度，所以很多大公司即使出高薪也很难招聘到理想的前端开发工程师。市面上有很多关于前端开发的书，这些书籍能很好地指导读者掌握前端开发的基础知识，能让读者达到会用的水平。然而，几乎还没有书能告诉开发者们如何才能用得更好，如何才能编写出高质量的前端代码，如何才能系统有效地组织前端架构……本书弥补了这方面的市场空白，它假定读者已经具有一定的 Web 前端开发基础，不会对基础知识进行详细介绍，主要精力放在如何编写和组织高质量代码上，从而提高代码的可维护性。

本书的重点不在于讲解技术，而是更侧重于对技巧的讲解。技术非黑即白，只有对和错，而技巧则见仁见智。本书是笔者个人的经验分享，尽信书不如无书，大家可以有选择地吸收，如果对书中的观点或技巧有不同的见解，非常欢迎与笔者讨论交流。大家可以通过邮箱 cly84920@gmail.com 与作者取得联系，也可以通过 QQ 群 8791223 参与到“如何编写高质量前端代码”的讨论中来。

在编写本书的过程中，如何组织目录一直是让笔者非常纠结的事情：HTML、CSS 和 JavaScript 是三门截然不同的语言，在实际应用过程中涉及的深度也各不相同，HTML 需注意的事项较少，CSS 次之，JavaScript 最为复杂。所以，本书虽然会同时对这三个方面进行探讨，但所用篇幅与它们的复杂度是成正比的。

## 致 谢 Acknowledgment

感谢在本书写作过程中为我提供宝贵意见的朋友们，他们是周裕波、钟志、刘运周、李海玲、钟伟明、邹渤一、黄海宝、胡淑芳，没有你们的反馈，这本书将失色不少。

感谢克军为我写的推荐序，很怀念我们一起吃午饭的那段日子，下次音乐节我一定到。感谢卢海军为我提供了三张精美的插图，它们真的很漂亮。

感谢华章编辑们的细心工作，谢谢你们一直耐心友好地对待我一次又一次的拖稿，与你们合作非常愉快。

感谢我的家人，谢谢你们在我最没有耐心写下去的时候一直陪着我。谢谢我的老婆张霞，没有你每天的督促，真不知道这本书要写到哪天才写得完。

推荐序

赞誉

前言

致谢

## 第1章 从网站重构说起/1

- 1.1 糟糕的页面实现，头疼的维护工作/2
- 1.2 Web 标准——结构、样式和行为的分离/4
- 1.3 前端的现状/6
- 1.4 打造高品质的前端代码，提高代码的可维护性——精简、重用、有序/8

## 第2章 团队合作/9

- 2.1 揭秘前端开发工程师/10
- 2.2 欲精一行，必先通十行/13
- 2.3 增加代码可读性——注释/15
- 2.4 提高重用性——公共组件和私有组件的维护/15
- 2.5 冗余和精简的矛盾——选择集中还是选择分散/16
- 2.6 磨刀不误砍柴工——前期的构思很重要/17

- 2.7 制订规范/18
- 2.8 团队合作的最大难度不是技术，是人/18

## 第3章 高质量的 HTML/19

- 3.1 标签的语义/20
- 3.2 为什么要使用语义化标签/21
- 3.3 如何确定你的标签是否语义良好/26
- 3.4 常见模块你真的很了解吗/36
  - 3.4.1 标题和内容/36
  - 3.4.2 表单/38
  - 3.4.3 表格/40
  - 3.4.4 语义化标签应注意的一些其他问题/43

## 第4章 高质量的 CSS/44

- 4.1 怪异模式和 DTD/45
- 4.2 如何组织 CSS/46
- 4.3 推荐的 base.css/49
- 4.4 模块化 CSS——在 CSS 中引入面向对象编程思想/55
  - 4.4.1 如何划分模块——单一职责/55
  - 4.4.2 CSS 的命名——命名空间的概念/60
  - 4.4.3 挂多个 class 还是新建 class ——多用组合，少用继承/66
  - 4.4.4 如何处理上下 margin/72
- 4.5 低权重原则——避免滥用手选择器/81
- 4.6 CSS sprite/85
- 4.7 CSS 的常见问题/88
  - 4.7.1 CSS 的编码风格/88

- 4.7.2 id 和 class/89
- 4.7.3 CSS hack/89
- 4.7.4 解决超链接访问后 hover 样式不出现的问题/93
- 4.7.5 hasLayout/94
- 4.7.6 块级元素和行内元素的区别/95
- 4.7.7 display:inline-block 和 hasLayout/97
- 4.7.8 relative、absolute 和 float/103
- 4.7.9 居中/104
- 4.7.10 网格布局/112
- 4.7.11 z-index 的相关问题以及 Flash 和 IE 6 下的 select 元素/122
- 4.7.12 插入 png 图片/129
- 4.7.13 多版本 IE 并存方案——CSS 的调试利器 IETester/131

## 第 5 章 高质量的 JavaScript/133

- 5.1 养成良好的编程习惯/134
  - 5.1.1 团队合作——如何避免 JS 冲突/134
  - 5.1.2 给程序一个统一的入口——window.onload 和 DOMReady/148
  - 5.1.3 CSS 放在页头，JavaScript 放在页尾/159
  - 5.1.4 引入编译的概念——文件压缩/160
- 5.2 JavaScript 的分层概念和 JavaScript 库/162
  - 5.2.1 JavaScript 如何分层/162
  - 5.2.2 base 层/163
  - 5.2.3 common 层/181
  - 5.2.4 page 层/184
  - 5.2.5 JavaScript 库/185
- 5.3 编程实用技巧/187
  - 5.3.1 弹性/187

5.3.2	getElementsByClassName、getElementsByTagName 和 getElementsByTagName/193
5.3.3	可复用性/196
5.3.4	避免产生副作用/199
5.3.5	通过传参实现定制/203
5.3.6	控制 this 关键字的指向/207
5.3.7	预留回调接口/211
5.3.8	编程中的 DRY 规则/212
5.3.9	用 hash 对象传参/215
5.4	面向对象编程/217
5.4.1	面向过程编程和面向对象编程/217
5.4.2	JavaScript 的面向对象编程/224
5.4.3	用面向对象方式重写代码/245
5.5	其他问题/251
5.5.1	prototype 和内置类/251
5.5.2	标签的自定义属性/255
5.5.3	标签的内联事件和 event 对象/260
5.5.4	利用事件冒泡机制/263
5.5.5	改变 DOM 样式的三种方式/267
<b>附录 A</b>	<b>写在规则前面的话/271</b>
<b>附录 B</b>	<b>命名规则/272</b>
<b>附录 C</b>	<b>分工安排/274</b>
<b>附录 D</b>	<b>注释规则/276</b>
<b>附录 E</b>	<b>HTML 规范/278</b>
<b>附录 F</b>	<b>CSS 规范/280</b>
<b>附录 G</b>	<b>JavaScript 规范/282</b>

## 第1章

# 从网站重构说起

## 本章内容

- 糟糕的页面实现，头疼的维护工作
- Web 标准——结构、样式和行为的分离
- 前端的现状
- 打造高品质的前端代码，提前代码的可维护性——精简、重用、有序

## 1.1 糟糕的页面实现，头疼的维护工作

从 1989 年诞生至今，网页技术已有 20 年历史了，其用途由最初的纯学术交流，延伸到如今的门户网站、电子商务网站、博客、邮箱、Web Game、SNS、维基百科等，涉及我们的工作、生活、学习和娱乐的方方面面。互联网世界有数以万亿计的网页，负责承载和展示信息。

事实上，制作网页并不难，只要有一个文本编辑器，买本入门书，几分钟就可以动手做出一个简单的网页来。但是，要做一个好的网页，却不是一件容易的事情。任何一个有经验的工程师都知道，工作中的最大考验和最不可回避的问题就是“变化”。我们在制作网页的时候，不仅要实现需求，更重要的是要考虑实现代码的可维护性，为未来可能出现的“变化”提前做好准备。

先来看一个可维护性不好的网页实现案例，实现过程如代码清单 1-1 所示，这是一个历史有点悠久的老网页，几乎有着所有老网页都有的典型毛病。

代码清单 1-1 一个糟糕的老网页的实现

---

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<style type=text/css>
td, body { font-size: 15px; font-family: arial, sans-serif, 宋体; }
body{margin-top:0px; margin-left:0px; margin-right:0px; background-color:#fcffff}
a:link{ color:#000000; text-decoration:none; padding-left:4px; }
a:visited{COLOR: #000000; TEXT-DECORATION: none; padding-left:4px; }
a:active{color:green; text-decoration:none; padding-left:4px; }
a:hover{color:red; text-decoration:underline; padding-left:4px; }
a.m:link{ color:#000000; text-decoration:none; padding-left:0px; }
a.m:visited{COLOR: #000000; TEXT-DECORATION: none; padding-left:0px; }
a.m:active{color:green; text-decoration:none; padding-left:0px; }
a.m:hover{color:red; text-decoration:underline; padding-left:0px; }
.t1{border-width:1px 1px 1px 1px; border-style:solid; font-size:12px;
text-align:center}
.bgg{border-color:#8AB78A; width:779px}
.f9pt{font-size: 12px; }
#sfont a, #sfont b{font-size:13px; }
</style>
<base target="_blank">
<title>网址之家——动漫</title>
<script src="js/usertrack.js"></script>
```

```
</head>
<body>
<div align="center"><center>
<table border="0" cellpadding="0" cellspacing="0" width="778" height="51">
    <tr>
        <td width="230" height="51"><a href="http://www.xxxx.com" target="_self" class="m"></a></td>
        <td height="51" align="center"><table width="100%" border=0 cellpadding=0 cellspacing=0>
            <form name=form1 action=http://www.baidu.com/s>
                <input type=hidden name=tn value=abc>
                <tr>
                    <td colspan="2" id=sfont>
                        <a href=http://news.baidu.com>新闻</a>
                        <b>网页</b>
                        <a href=http://tieba.baidu.com>贴吧</a>
                        <a href=http://zhidao.baidu.com>知道</a>
                    </td>
                </tr>
                <tr>
                    <td height="30" valign="top">
                        <input type=text name=wd size=40 onMouseOver=this.focus()
                            onFocus=this.select() style="margin-bottom:-5px; font-size:16px; height:1.78em; font-family:arial,sans-serif,宋体; padding-top:2px; padding-left:1px" maxlength=100>
                        <input type=submit style="height:1.82em; width:6.4em; font-size:16px; margin-bottom:-5px; padding-top:2px" value="百度一下">
                    </td>
                    <td width="80" valign="top" class="f9pt"> </td>
                </tr>
            </form></table></td>
        </tr>
    </table>
    </center></div>
<script language=javascript>
<!--
UserTrack.init(1,"动漫")
document.form1.wd.focus()
//-->
</script>
</body>
</html>
```

这个页面的实现代码都有哪些地方不利于维护呢？

- div 布局和 table 布局混用；
- 标签名有大写的，也有小写的；
- 标签属性有的加了引号，有的没有加引号；

- 历史遗留的、被淘汰的属性泛滥；
- 样式组织混乱，有用<style>标签写进网页里的，有用<link>外链的，也有直接写在标签内的；
- JavaScript 有外链的，有写在<script>标签内的，也有写在标签里的；
- JavaScript 和 CSS 的位置零乱；
- JavaScript 的编码风格很不一致；
- 无论是 JavaScript 代码还是 CSS 代码，都看不到任何注释；
- .....

这是一段非常典型的网页代码，毫无章法，结构（HTML）、样式（CSS）和行为（JavaScript）非常混乱地耦合着，在网站重构这场革命以前，大多数的网页都有这些毛病。维护这样的网页难度非常大，成本自然也非常高。

那么，如何组织网页代码才算是有章法呢？说到这个问题，就不得不提 Web 标准，它是网站重构的基础。

## 1.2 Web 标准——结构、样式和行为的分离

W3C 是一个专门负责制定网页标准的非营利性组织，致力于结束网页制作领域混乱不堪的局面，Web 标准就是由 W3C 组织推行的。要了解最权威、最详细的关于 Web 标准的信息，可以访问其官方网站：<http://www.w3c.org>。

Web 标准由一系列标准组合而成，其核心理念就是将网页的结构、样式和行为分离开来，所以它可以分为三大部分：结构标准、样式标准和行为标准。结构标准包括 XML 标准、XHTML 标准、HTML 标准；样式标准主要是指 CSS 标准；行为标准主要包括 DOM 标准和 ECMAScript 标准。

一个符合标准的网页，标签中的标签名应该全部都是小写的，属性要加上引号，样式和行为不再夹杂在标签中，而应该分别单独存放在样式文件和脚本文件中。理想状态下，网页源代码由三部分组成：.html 文件、.css 文件和.js 文件。标签中混有样式和行为的写法是不推荐的，如代码清单 1-2 所示。

### 代码清单 1-2 结构、样式和行为混杂的网页实现

```
<td width="100%" height="20" class="f9pt" align="center">
    <font color="#346F0E">下载</font>
    <input type="text" name=wd size=40 onMouseOver=this.focus() onFocus=
this.select() style="margin-bottom:-5px;font-size:16px; height:1.78em;
font-family:arial,sans-serif,宋体;padding-top:2px; padding-left:1px" maxlength=100>
</td>
```

这种段代码应该改成下面这样的形式，如代码清单 1-3 所示。

### 代码清单 1-3 结构、样式和行为单独分开的网页实现

test.html 文件：

```
<link rel="stylesheet" type="text/css" href="test.css" />
<td class="f9pt myTd">
    <span class="myFont">下载</span>
    <input type="text" name="wd" size="40" id="myInput" maxlength="100" />
</td>
<script type="text/Javascript" src="test.js"></script>
```

---

test.css 文件：

```
.myTd{width:100%;height:20px;text-align:center;}
.myFont{color:#346F0E;}
#myInput{margin-bottom:-5px;font-size:16px;height:1.78em;font-family:
arial,sans-serif,宋体;padding-top:2px;padding-left:1px}
```

---

test.js 文件

```
var myInput = document.getElementById("myInput");
myInput.onmouseover = function(){
    this.focus();
}
myInput.onfocus = function(){
    this.select();
}
```

---

如此一来，原来混乱不堪的 HTML 变得清爽多了。HTML 标签只用负责承载内容，而样式交给了 CSS，行为交给了 JavaScript。

将结构、样式和行为分成三个文件，这是比较推荐的做法。但事实上，我们在工作中经常会因为各种原因而仍然把样式和行为放在.html 文件中。即使是在这样的情况下，仍然可以将样式和行为从标签中分离出来，如代码清单 1-4 所示。

#### 代码清单 1-4 将样式和行为从标签内分离开

```
<style type="text/CSS">
.myTd{width:100%;height:20px;text-align:center;}
.myFont{color:#346F0E;}
#myInput{margin-bottom:-5px;font-size:16px;height:1.78em;font-family:
arial,sans-serif,宋体;padding-top:2px;padding-left:1px}
</style>
<td class="f9pt myTd"><span class="myFont">下载</span><input type= "text"
name="wd" size="40" id="myInput" maxlength="100" /></td>
<script type="text/Javascript">
var myInput = document.getElementById("myInput");
myInput.onmouseover = function(){
    this.focus();
}
myInput.onfocus = function(){
    this.select();
}
</script>
```

如果不将 CSS 和 JavaScript 作为独立文件外链出去，而是在 HTML 页面里用 `<style>` 和 `<script>` 标签来承载样式和行为也是可以的，只要不在 HTML 标签上书写样式和行为，仍然实现了结构、样式和行为的分离。代码的职能已经非常非常清晰了，只是没有分开成三个文件而已。

但是，仅仅将结构、样式和行为分离开就可以带来足够的可维护性吗？不，这仅仅只是个开始。

### 1.3 前端的现状

这几年来，Web 开发技术发展十分迅速。随着《网站重构》一书的问世，CSS 布局代替传统 `table` 布局之风迅速刮起，国内外大大小小的网站都纷纷加入到这场技术革命中。Gmail 的上线，让 Ajax 一夜之间成为了 Web 开发领域的明星。之后，随着 Ajax 的火热，DHTML 再次受到热捧，各种 JavaScript 框架如雨后春笋般涌现出来，让人应接不暇。

这种发展变化是一把双刃剑，一方面它使得网页的表现力越来越强，我们可以用网页做出惊艳的效果；另一方面，漂亮的界面背后隐藏着的是越来越难维护的实现代码。

为什么网页的维护工作会变得越来越难？原因主要来自三个层面：

- **浏览器层面。**浏览器的向前兼容使得前端开发中被淘汰的技术、不推荐的方法依然广为流传和应用，而新一轮的浏览器大战却愈演愈烈，除了 Firefox、Opera、Safari、Chrome 这些 IE 的挑战者外，IE 本身也同时流行着 IE 6<sup>⊖</sup>、IE 7 和 IE 8 三个不同的版本。不同的浏览器对网页代码的解析存在着或大或小的差异。
- **技术层面。**Web 标准被重视和普遍采用的时间不长，整个大环境对 Web 标准的理解还停留在概念层面，对“好的实现方案”仍处于摸索阶段。不同的公司，不同的团队，不同的工程师，对“好的实现方案”有自己的理解，或深或浅。理解不深，就很容易写出可维护性差的代码。
- **团队合作层面。**随着用户对使用体验的要求的不断增加，对网页的表现力的要求也越来越高，从而导致实现代码越来越复杂，这无疑给团队合作带来了麻烦。页面越复杂，对团队合作的要求就越高。如果合作不默契，很可能需要不停地打补丁，最后让代码变得千疮百孔，满是地“雷”，没有人愿意去维护它们。

随着维护难度的增加，网页制作对技术的要求越来越高，Web 开发领域长期以来形成的设计和制作不分的局面终于有所好转。细心的朋友可能会发现，在 51job 这样的招聘网站上，前两年几乎只有“网页设计师”这个职位，而现在已经有了前端开发工程师和页面工程师<sup>②</sup>这样的职位。之前既要负责设计又要负责制作的网页设计师，已经分离成了两个岗位：一个专门负责设计，属于艺术类；另一个专门负责制作，属于技术类。这是一个可喜的变化，设计师（designer）和开发者（developer）本来就是两个完全不同的方向，将两者明确分开，表示网页制作的分工向着合理成熟的方向又迈出了一大步。专注于网页制作的技术方向，有了更专业的叫法——“前端开发”。

相比于服务器端开发曲折陡峭的学习曲线，前端开发的学习曲线平滑得多。正因为前端开发上手快、门槛低，所以非常多的人通过自学成才，从这一方向进入 IT 领域。

---

<sup>⊖</sup> IE 6 可能会逐渐退出历史舞台了。

<sup>②</sup> 前端开发工程师和页面工程师是指同一职位，只是称呼不同。

也正因为如此，这一领域的大多数技术人员并没有比较系统的知识结构和良好的编码习惯。虽然有 Web 标准的概念，但往往并没有一个很好的实践方案。并不是结构、样式和行为分离了就完了，如何在多人合作时做到有条不紊、如何让代码的可维护性更好等这些“实践”性的技巧往往比 Web 标准本身更重要。如果重理论，而忽视了实践，那么代码的质量依然很难得到提高。

然而，什么样的代码算是高品质的代码呢？

## 1.4 打造高品质的前端代码，提高代码的可维护性——精简、重用、有序

所谓高质量的代码，在 Web 标准的思想指导下，在实现结构、样式和行为分离的基础上，还要做到三点：精简、重用、有序。精简的代码可以让文件变小，有利于客户端快速下载；重用可以让代码更易于精简，同时有助于提升开发速度；有序可以让我们更清晰地组织代码，使代码易于维护，有效应对变化。

Web 标准是一套理论性的指导思想，它的最终目的是让代码更易于维护，标准本身是手段，而不是目的。在应用 Web 标准的实践中，有时候不遵循标准反而能带来更好的可维护性，如果你确信你的方案利大于弊，那么就去做吧，尽信标准不如无标准，过于教条主义是一件很愚蠢的事情。

## 第2章

# 团队合作

## 本章内容

- 揭秘前端开发工程师
- 欲精一行，必选通十行
- 增加代码可读性——注释
- 提高重用性——公共组件和私有组件的维护
- 冗余和精简的矛盾——选择集中还是选择分散
- 磨刀不误砍柴工——前期的构思很重要
- 制订规范
- 团队合作的最大难度不是技术，是人

## 2.1 揭秘前端开发工程师

本章主要探讨团队合作的注意事项，在正式讲解如何进行团队合作之前，先让我们了解什么是前端开发工程师。

前端开发工程师是一个新兴的职业，关于它的职能定位，包括很多同行在内，都比较模糊。下面结合一些互联网公司对前端开发工程师的招聘要求来看一下这个职位到底需要具备哪些技能和素质。

下面是盛大公司的一则招聘中对前端开发工程师的要求：

1. 绝对熟练 div+CSS 制作方式，对网页标准有独特的见解；
2. 精通 XHTML \ CSS \ JavaScript，熟练使用 JavaScript 前端脚本，要有常用效果和功能的积累。会用常见的 JavaScript 类库，如 jQuery 或 Mootools 等。理解异步请求原理，可以配合程序员开发一些简单的 Ajax 应用。
3. 至少了解一门后台语言，例如 PHP 或 ASP.NET。
4. 熟练使用网页设计相关的软件。
5. 两年以上团队协作开发经验，有责任心、爱钻研、爱思考。
6. 具有本科及以上学历。

从前端开发工程师的角度来解读这则招聘中的要求，我们可以得出如下结论：

**第一：** CSS 布局是前端开发工程师的基本功，一定要熟练。这点我十分赞同，因为前端开发工程师的本职工作就是制作网页。在前端开发工程师的日常工作中，CSS 的使用比重占到了所有技能的 50%~70%，有些公司可能更高。所以，有些公司在招前端开发工程师时甚至只要求会用 CSS。对于 CSS，最基本的要求就是能兼容主流浏览器——IE 6、IE 7、IE 8 和 Firefox，有些公司可能还会要求兼容 Opera、Safari、Chrome。

**第二：** 对 JavaScript 的使用有所要求，不仅要知道会使用原生的 JavaScript，还要会使用 JavaScript 类库和 Ajax。在这个用户体验至上的时代，网页已经不再只是内容的承载

体，只负责简单地显示页面内容，现在的网页不仅要求能吸引眼球，而且还要为用户提供良好的交互体验。

这里提到了三个概念：原生 JavaScript、JavaScript 类库和 Ajax，有些刚从事前端开发工作的朋友可能不太明白它们三者之间的区别，本书略微解释一下。

原生 JavaScript 是浏览器默认支持的脚本语言，Ajax 是一种利用 JavaScript 和 XMLHttpRequest 对象在客户端和服务器端传送数据的技术，因为 XMLHttpRequest 对象也是用 JavaScript 来创建的对象，所以从某种意义上来说，Ajax 是 JavaScript 的一个子集。很多刚进入这个行业的朋友将 Ajax 和 JavaScript 并列起来讲，甚至认为 Ajax 比 JavaScript 复杂得多。其实这是误解！

事实上，Ajax 只是种提交数据的方式，与传统的表单提交方式相比的确有所不同，但其核心内容其实非常少，学习起来并不困难。前端开发中最复杂的技术其实是 JavaScript。JavaScript 类库又是什么呢？因为浏览器默认支持的 JavaScript（我们常称为原生 JavaScript）会因浏览器的不同而有所差异，例如 IE 支持 `document.all` 属性，而 Firefox 不支持。同时，原生 JavaScript 提供的方法可能不太好用，比如只提供了 `document.getElementById` 和 `document.getElementsByTagName`，却没有提供 `document.getElementsByClassName`。又比如，原生 JavaScript 并不提供富文本编辑器和拾色器这种复杂的 UI 工具。基于这些原因，于是出现了 JavaScript 类库。JavaScript 类库是在原生 JavaScript 的基础上，封装了跨浏览器兼容的特性并扩展了一些功能，提供了一些原生 JavaScript 没有的 API，供开发者快速开发 JavaScript 程序使用。

第三：前端开发工程师为什么需要了解后台语言呢？需要了解到什么程度呢？这是让很多同行感到迷惑的地方。事实上，前端开发工程师的工作只是制作网页，对于网页中动态生成的内容，也就是来自数据库的内容完全不用关心。前端工程师不必了解系统的数据库设计，不必了解数据在数据库中存取的细节。前端开发工程师之所以需要了解后台语言，主要因为以下三个方面的原因：

- 知道服务器端工程师在生成页面时会如何进行输出，以便编写出方便他们套脚本的模板；

- 在写 Ajax 应用的时候，可以自己模拟服务器端输出，方便调试；
- 对前端和服务器端如何配合有清晰的大局观认识，了解数据传递的整个流程，以配合工程师共同制定复杂效果的实现方案。

要做到这三点其实并不困难，我们甚至只需要掌握服务器端语言的部分语法就可以了！以 PHP 为例，只需要知道 \$、echo、\$\_REQUEST[]、单引号和双引号的区别、session、cookie 即可。我们不需要了解如何使用 PHP 的高级语法，不需要了解如何配置 Apache，甚至完全不需要知道 SQL 语言。当然，服务器端语言掌握得越深，对我们处理复杂问题越有帮助。

事实上，前端开发工程师并不需要写服务器端语言，之所以对服务器端语言有要求，完全是为了在工作中与服务器端工程师配合默契。所以，前端开发工程师只需要了解服务器端工作原理，会用服务器端语言写输出接口即可。而无论哪种服务器端语言，例如 PHP、ASP/ASP.NET、JSP，虽然具体语法有差异，但工作原理大同小异，所以一般公司的招聘要求并不指定具体是哪门语言，只要熟悉任何一门服务器端语言即可。

大家可能注意到，在盛大的这则招聘信息中，Flash 并没有被提及！这是因为，随着 Web 标准的推广，前端开发工程师的本职工作已经重点落到了网页的制作上，网页三要素——结构、样式、行为，也就是 HTML、CSS 和 JavaScript 才是前端开发工程师的核心工作，而 Flash 作为富媒体应用，从前端开发中被分离出来，由专门的 Flash 开发工程师负责。当然，由于过去将“网页设计师”等同于“网页制作三剑客”的历史原因，前端开发工程师和 Flash 仍然还是有着剪不断理还乱的关系。虽然 Flash 已经越来越强大，越来越需要专攻 Flash 的人来负责开发，但还是有很多人认为 Flash 应该是前端开发工程师应该掌握的技能，至少是附属技能。可喜的是，持这种观点的人已经越来越少。广义上来讲，富媒体开发也属于前端开发的范畴。

前端开发位于网站开发的中游，上游有交互设计师和视觉设计师，下游有服务器端工程师，前端开发工程师需要与他们沟通，所以需关注的知识面非常广。图 2-1<sup>①</sup>真实地反映了一位前端开发工程师应该关注和掌握的技能。

---

① 本图的作者是张克军，曾任职于雅虎，现任职于豆瓣网。

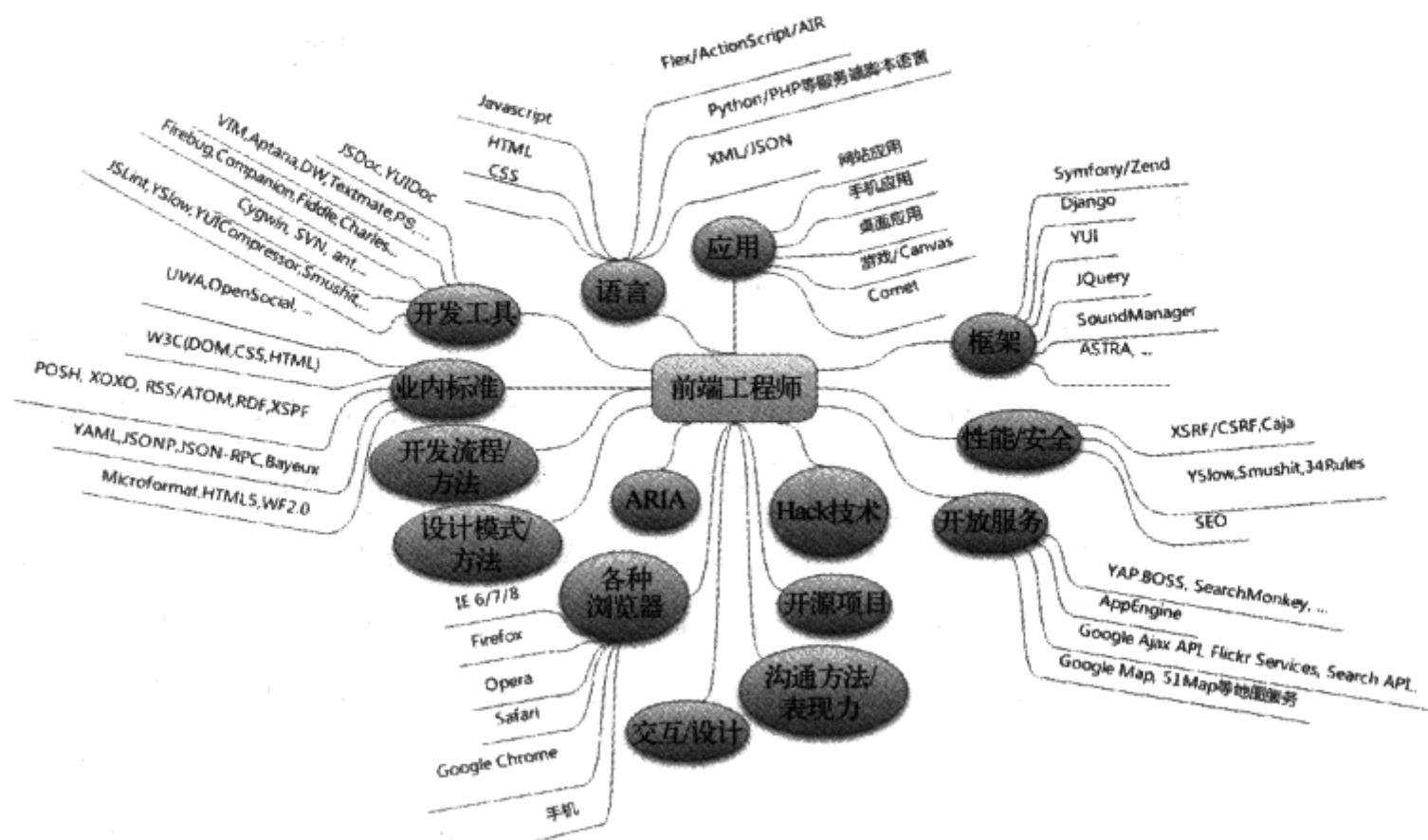


图 2-1 前端开发工程师应该关注的知识领域

很多朋友可能会问了，我只是前端开发工程师，只想要专精一个领域！涉猎面如此之广，真的有必要吗？

## 2.2 欲精一行，必先通十行

将前端开发和服务器端开发做一个比较，前端开发没有服务器端开发“深”，服务器端开发没有前端开发“广”。经常听到做前端的同行抱怨需要学的东西太多，东学一点西学一点，什么都会，但也什么都不精。很直接的结果就是沦为打杂的程序员，对能力没自信，在团队说话也不够有分量。于是越来越多的同行们得出了一个结论：“通十行不如精一行！”

其实这是个误区。精通一行？在前端开发领域，不通十行就无法精一行。

先来说说“精一行”这个很重要的概念吧。具体细化到什么程度叫做“一行”？是具体到前端/服务器端，还是具体到设计/DIV+CSS/JavaScript/RIA？细化的粒度越小，我

们需要掌握的也就越少。很多工程师为了能够快速“精”一行，尽量让“一行”的粒度细化。可是，有两个问题：

- 精的粒度越小，我们的就业范围就越小。显而易见，如果你精通的范围越小，你的实用价值也就越小。
- 这个行业的界限非常不明显，各个领域互相渗透。比如说。你想成为 ActionScript 3 方面的专家，你选择了走 Developer 的路。Designer 相关的知识可以不用考虑太多。你不用去学配色，不用去学 PhotoShop 质感处理，不用去学 AI，不用去学 CD，不用去管用户交互，不用去管版式设计，你只管程序就 OK。而且，我只想成为 ActionScript 3 方面的专家，我只要学好 ActionScript，而且是 ActionScript 3，ActionScript 2 我都可以不用去学，多轻松啊！是吧？真是这样吗？当你决定只去钻这一个方向的时候，你会发现原来 ActionScript 还要与前台和服务器交互，ActionScript 自己不是万能的，它需要与其他程序配合。好吧，那么前端的和后端的你可以不学吗？如果不学，你会发现自己很多时候搞不明白整个流程，你的工作会困难重重。是的，只是知道就可以了，并不需要精。技术与技术之间会互相依赖、交叠和渗透，就算你只想成为一名视觉设计师，如果你不懂 div+CSS，你设计的图前端工程们可能很难实现。也就是说，想要做个好的视觉设计师，掌握一些 CSS 的知识也是必要的。

我们再回到前面的盛大招聘的例子。盛大招聘的前端工程师有些什么要求？前端各种技术该有的都有了，为何还要求会服务器端技术？是它们不懂技术乱提条件吗？相反，是它们懂技术，知道“精一行，得通十行”的道理。它们的招聘岗位又怎么样呢？有细分到 ActionScript 3 工程师、jQuery 工程师、YUI 工程师、PS 设计师、AI 设计师吗？没有，分工如此细的岗位是不存在的。没办法，只专精一个极细领域的岗位的需求是极少的，我们不得不选择“粗粒度”的精，也就是说，不必精十行，至少要通十行。

专精很难，甚至不可能，一专多能才是现实的。在前端开发这个领域，一专多能更是非常必要的。

## 2.3 增加代码可读性——注释

知道了什么是前端开发工程师，那么接下来讲讲前端开发工程师的团队合作需要注意些什么问题。

我们的开发过程，很可能是需要多人合作完成的。大体来说，可以分成两种情况：一种是事先商量好，有组织有计划的分工合作，我们称其为“直接团队合作”好了；另一种是事先并没有考虑到的，因为种种原因而导致你去维护他人开发过的系统，我们称其为“间接团队合作”。

不要断定你现在编写的代码，将来一定不会有其他人来维护，“间接团队合作”常常是出乎意料的！就算你十分确定这代码只可能由你自己来维护，也一定要做好“间接团队合作”的准备，因为哪怕是自己写的代码，3个月之后你再来看它，一样会觉得它十分陌生！

无论是“直接团队合作”还是“间接团队合作”，一个保证代码可读性良好的绝佳武器就是注释。关于注释甚至有一种说法：“一个好的代码，注释要占 1/3 的篇幅”。虽然有些夸张了点，不过它表明了注释的重要性。

## 2.4 提高重用性——公共组件和私有组件的维护

团队合作很容易产生的问题之一就是冗余。如果没有事先做好规划，很容易出现这样一种情况：工程师甲在页面 A 上为了实现某种效果写了一段代码，工程师乙在页面 B 上遇到同样的问题时又重写了一遍。如果系统升级，这个效果需要改变，那么工程师甲和工程师乙都需要更改这部分代码，明显是一种资源浪费。

避免出现这种冗余的最好办法就是根据代码的重用度，把它们分成公共组件和私有组件两类。设计公共组件时需要考虑让接口保持弹性，并且高度模块化，这是很考验能力和经验的工作。关于前端开发的公共组件和私有组件的设计细节详见 4.2 节和 5.2 节。

因为公共组件在设计之初就是考虑到要被多人多处重复使用的，如果公共组件一旦被修改会造成很大影响！所以我们需要对公共组件的稳定性保持高度警惕，不允许轻易做修改。一般来说，公共组件对于绝大多数人只提供“读”的权限，不允许它们进行修

改，只对少数公共组件的维护者提供“写”的权限。关于“读”和“写”的权限问题，可以通过相关软件实现，例如 svn，也可以通过“规范”的形式对团队成员进行约定。

一般来说，公共组件需要由专人来维护。负责维护公共组件的工程师需要将公共组件做好注释，必要时甚至需要提供规范的 API 文档和演示 Demo。其他工程师在工作中如果遇到需要高度重用的模块，而公共组件中尚未提供，可以向维护公共组件的工程师提交需求，让他来添加相应的公共组件。

私有组件因为不考虑其他人重用的问题，所以对于修改操作会自由得多。但不要忘记，虽然你的私有组件不会被他人重用，但仍然可能会被他人维护，所以必要的注释还是需要的。

## 2.5 冗余和精简的矛盾——选择集中还是选择分散

有了公共组件和私有组件的区分，可以有效避免代码的重复。但又会带来新的问题——如何组织公共组件。

一般来说，合理的前端架构中 CSS 和 JavaScript 都是会提取公共组件的，如何组织公共组件是需要权衡的。如果没有将公共组件载入到系统中，那么组件是没法被使用的。一个最方便的做法就是将公共组件全部打包好，然后一次性全部载入，这样可以保证所有的组件都是可用的。这种做法的好处是加载方便，但加载的代码量可能过大，而很多代码事实上很可能是没有被用到的。另一种做法是将代码精确划分成一小块一小块，然后按需加载相应的模块。这种做法的好处是加载灵活，保证代码的加载量小，但坏处是使用起来比较麻烦。

我们在组织公共组件时，组织的粒度越大，文件越集中，加载和管理越方便，但无用代码越多；组织的粒度越小，文件越分散，加载和管理越麻烦，但无用代码越少。我们要加载方便就要适当舍弃精简性，要保证精简性就要适当放弃加载的方便性。我们需要认识到一点：完美的解决方案是不存在的，我们只能在冗余和精简中尽量找到最佳的平衡点。

以 JavaScript 为例，拿最流行的两个 JavaScript 框架 jQuery 和 YUI 来说，jQuery 就选择了“集中”，而 YUI 选择了“分散”。jQuery 将所有组件全放在了一个文件中，但

它通过非常优秀甚至诡异的算法，让代码本身尽可能精简，从而尽量让代码在“加载和使用方便”的基础上，尽量往“精简”靠拢。而 YUI 将组件按功能分成了一个个不同的文件，在页面中只根据需求加载相关的文件。YUI 通过智能的 loader 方式，尽可能让这种按需加载对用户友好。YUI 在保证“精简”的基础上，尽量往“加载和使用方便”靠拢。两者出发点不同，但都找到了一个不错的平衡点。

我个人更偏向 YUI 的按需加载方式，因为这样的方式扩展性更好。jQuery 之所以成功，和 js 的原生 API 比较少有直接关系，jQuery 提供的为数不多的 API 已经基本能够满足我们日常开发的绝大部分需求。但这种情况并不适用于所有的语言，比如 API 数量庞大的 as 和 java。“import”这种按需加载的方式才是真正的主流。

因为公共组件是预写好的，弹性才是它们最应该重点考虑的，毕竟不是特定为完成某功能而定制的，所以就算是按需加载，仍然可能会存在无用代码。这个是无法解决的，我们得认识到，只可能尽量减少冗余，不可能根除冗余。

## 2.6 磨刀不误砍柴工——前期的构思很重要

很多同行容易犯一个错误，就是拿到一个任务后就马上开始写代码，其实这并不是一个好习惯。

对于简单的独立页面而言，这么做还不会有太大麻烦。但是，对于一个中大型的网站而言，如果在还没有一个考虑成熟的前端架框前就开始动手写代码是会带来非常多问题的，比如代码冗余、多人合作容易冲突、代码组织没有规律等。

犯这种错误主要因为两个方面的原因：一方面可能是工程师本身经验不足；另一方面可能是客户或 Boss 给的压力很大，拼命赶工期。如果是后者，我们一定要顶住压力，客户和 Boss 往往很可能并不了解技术，它们可能更希望尽快看到工作成果。如果在没有一个成熟的框架前就开始写代码，很可能会出现先快后慢的局面，越到后期开发速度越慢，反复修改 bug、打补丁，系统的开发和维护成本越来越大。如果一开始不急于马上进行开发，而是先根据用户需求进行分析，先考虑好框架，会让整个开发过程更有规划、更顺畅，是一个先慢后快的过程。

前期的构思非常重要。具体来说，构思的内容主要包括规范的制定、公共组件的设计和复杂功能的技术方案等。一般来说，前期构思占整个项目 30%~60% 的时间都算是正常的。要知道，磨刀不误砍柴工！

## 2.7 制订规范

规范对于团队合作来说是最重要的。它能有效指导团队成员按照一个统一的标准进行开发，尽量让开发过程平滑，减少不必要的冲突，提高开发率并保证软件的质量。

规范会随制定者的经验和风格而不同，没有一个统一标准和准则，主观性非常强。因为规范的好坏会直接影响整个开发过程，所以它通常是由团队中经验最丰富的人来制定的。正是由于规范的主观性强，所以不同的公司有不同的规范。

笔者抛砖引玉，奉上自己制订的一份规范，供大家参考详见附录。

## 2.8 团队合作的最大难度不是技术，是人

依笔者的经验来看，团队合作的最大难度其实并不是技术，而是人。

只要有一定经验，构思良好，有完善的规范，尽管最终代码可能有好有坏，但团队合作在技术上并不太可能出现大的困难。团队合作最大的问题还是来自于团队中成员之间的交流。不同的工程师的说话方式、工作习惯、性格特点也可能各异，工作中出现观点不同的情形在所难免。不同工程师在处理这样的问题时表现出的态度也不一样，有些比较温柔，有些则可能稍显强硬，有些可能比较容易接受他人观点，有些则可能固执己见。如果处理不好，可能会产生火药味，让大家带着情绪工作，影响开发进度。

工程师往往都更专注于技术，不太善于处理人际关系。比起复杂的人，大多数工程师往往更喜欢与非黑即白，非 0 即 1，非 true 即 false 的代码相处。学会与人相处也是工程师们必修的一门课，它的重要性甚至超过了技术本身。

切记，自己能独立决策的问题都是小问题，要与人合作商讨的问题才可能会是最大的问题，要学会与人相处。

## 第3章

# 高质量的 HTML

### 本章内容

- 标签的语义
- 为什么要使用语义化标签
- 如何确定你的标签是否语义良好
- 常见模块你真的很了解吗

### 3.1 标签的语义

HTML 标签的设计都是有语义考虑的，表 3-1 是部分标签的全称和中文翻译。

表 3-1 标签语义对照表

标 签 名	英 文 全 拼	中 文 翻 译
div	division	分隔
span	span	范围
ol	ordered list	排序列表
ul	unordered list	不排序列表
li	list item	列表项目
dl	definition list	定义列表
dt	definition term	定义术语
dd	definition description	定义描述
del	deleted	删除
ins	inserted	插入
h1~h6	header 1 to header 6	标题 1 到标题 6
p	paragraph	段落
hr	horizontal rule	水平尺
a	anchor	锚
abbr	abbreviation	缩写词
acronym	acronym	取首字母的缩写词
address	address	地址
var	variable	变量
pre	preformatted	预定义格式
blockquote	block quotation	区块引用语
strong	strong	加重
em	emphasized	加重
b	bold	粗体
i	italic	斜体

(续)

标 签 名	英 文 全 拼	中 文 翻 译
big	big	变大
small	small	变小
sup	superscripted	上标
sub	subscripted	下标
br	break	换行
center	center	居中
font	font	字体
u	underlined	下划线
s	Strikethrough	删除线
fieldset	fieldset	域集
legend	legend	图标
caption	caption	标题

其中，div 和 span 其实是没有语义的，它们只是分别用作块级元素和行内元素的区域分隔符。既然它们没有语义，那么设计它们的目的是什么呢？先给大家留个悬念，在后面的章节将会予以说明。

## 3.2 为什么要使用语义化标签

如果你从设计师手上拿到这样一张设计图如图 3-1 所示：

如果哪位设计师果真把界面设计成这样，那他离失业可能就不远了。这张图只是为了演示标签语义化而特意做的示意图，它包含了一些布局中常见的模块。

传统的布局方式是在 Dreamweaver 中拖曳表格来布局，制作网页并不需要手写代码，生成网页的速度很快，而且也并没有太高的门槛，并不需要工程师来做，设计师就足以胜任。但是，通过这种方式制作的网页自动生成的代码量非常庞大，含有大量的 table、tr、td 标签，它们的语义分别是“表格”、“单元行”、“单元格”，从中看不到页面真正需要的语义。

**登录**

账号： 密码：

课程大纲：

- 为什么要语义化标签？
  - 网页布局的传统方法（代码演示）
  - 传统方法的缺点分析
  - 新的方法——语义化标签
- 标签的语义
  - 标签语义简介
  - 语义化标签的布局实现（代码演示）
  - 语义化标签的优点分析
- 标签语义化的误区
  - CSS很强大
  - 乱用标签的问题分析（代码演示）
- 如何很好地语义化你的标签
  - 浏览器对标签有默认样式
  - 如何对标签是否语义良好进行调试
  - 语义化良好的站点欣赏
- 常用模块的实现详解
  - 标题和内容
  - 菜单
  - table
  - 语义化标签应注意的一些问题总结
- 作业

为什么要语义化标签？ [更多>>](#)

几种页面实现的比较			
实现方式	代码量	搜索引擎友好	特殊终端兼容
table布局	多	差	一般
乱用标签的CSS布局	少	一般	差
标签语义良好的CSS布局	少	好	好

标签的语义 [更多>>](#)

使用具有良好语义的标签，能够很好地实现自我解释，方便搜索引擎理解网页的结构，抓取重要内容。去样式后也会形成清晰的语义，很好地组织网页内容，具有很好的可读性，从而实现对特殊终端的兼容。

虽然现有的标签很多，但有些标签已经不推荐使用了，我们经常用到的标签其实相当少，我们只需要重点掌握这些常用标签就可以了。

语义良好的布局优点： [更多>>](#)

- 页面表现力强
- 标签语义化非常清晰，结构良好；
- 代码量少，无用信息最小化，减小了下载时间，对搜索引擎友好；
- 对于网页来说，结构处于绝对的核心位置；
- 代码组织良好，容易维护；
- 对样式的依赖降低，去样式可读性良好，最大程度兼容特殊终端。

图 3-1 简单设计图

可以通过视觉上的效果来判断内容的语义，比如说，通过文字大小、排版位置、配色等视觉效果很容易得到这样一个印象：“标签语义化”是这个网页的一级标题，“课程大纲”、“为什么要语义化标签”、“标签的语义”、“语义良好的布局优点”是这个网页的二级标题。

但是搜索引擎看不到视觉效果，看到的只是代码，只能通过标签来判断内容的语义。

`table` 布局的网页有如下特点：

- 代码量大，结构混乱；
- 标签语义不明确，对搜索引擎不友好。

为什么 `table` 布局会这么糟糕呢？回顾一下网页技术的发展史就不难理解了。

网页最早用于大学间交流论文，只是简单的文档，对于网页的样式并没有太多要求。随着互联网开始深入我们的生活，简陋的样式已经不能够满足网页功能的需要了，人们迫切需要网页有更强的表现力。由于当时的网页还不具备布局的功能，于是人们想到了利用 `table` 来强行实现布局。`table` 标签在设计之初只是想用来呈现表格式的二维数据的，被用来布局其实是一种 `hack` 用法。在此之后，早在 1996 年，W3C 组织就已经推出了 CSS 1.0 版，1998 年又推出了 CSS 2.0 版，CSS 的推出就是为了解决网页表现力不够强的问题的。但是，一方面因为 `table` 布局已经成为一种习惯性的做法；另一方面，人们还没有深入认识到 CSS 布局的优点，使得 `table` 布局成为事实上的主流布局方式，这种情况一直到近几年才有所扭转——CSS 布局的时代来临了。

CSS 布局也就是俗称的 `div+CSS` 布局，或者(X)HTML+CSS 布局。其核心思想就是用 CSS 来控制网页中元素的样式，包括位置、大小、颜色等。CSS 布局可以摒弃 `table` 布局方式中为强制定位而添加的大量标签，从而让 HTML 可以从样式、结构混杂的局面中挣脱出来，专注于结构。

什么是专注于结构呢？通俗地说，标签的职能只限于告诉你：“这是一个标题”，“这是一个段落”，“这是一个无序列表”，并不会告诉你：“这是红色的”，“这个有 500px 宽”，“这个背景是绿色的”。

CSS 布局弱化了标签的“布局”能力，将“布局”完全放到了样式中进行控制，而标签重新恢复了原来的作用。与 `table` 布局相比，CSS 布局具有代码量少、结构精简、语义清晰等优点。代码量少，浏览器端的下载时间就会更短，语义清晰就会对搜索引擎更友好。也正因为如此，所以 CSS 布局取代 `table` 布局是必然的趋势。

CSS 很强大，一般情况下，我们可以随心所欲控制页面内几乎所有可见元素的样式。

举个简单的例子，如代码清单 3-1 所示。

代码清单 3-1 一个简单页面的实现

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>阿当的简单示例</title>
</head>
<body>
<h1>h1 标签</h1>
<h2>h2 标签</h2>
<h3>h3 标签</h3>
<p>p 标签</p>
<div>div 标签</div>
<span>span 标签</span>
<strong>strong 标签</strong>
<input type="text" value="input 标签" />
<textarea>textarea 标签</textarea>
<input type="button" value="提交" />
<ul>
<li>1</li>
<li>2</li>
<li>3</li>
<li>4</li>
<li>5</li>
<li>6</li>
<li>7</li>
<li>8</li>
</ul>
</body>
</html>
```

页面中有各种不同的标签，我们先看看不用 CSS 控制的情况下它在浏览器中的表现，如图 3-2 所示：

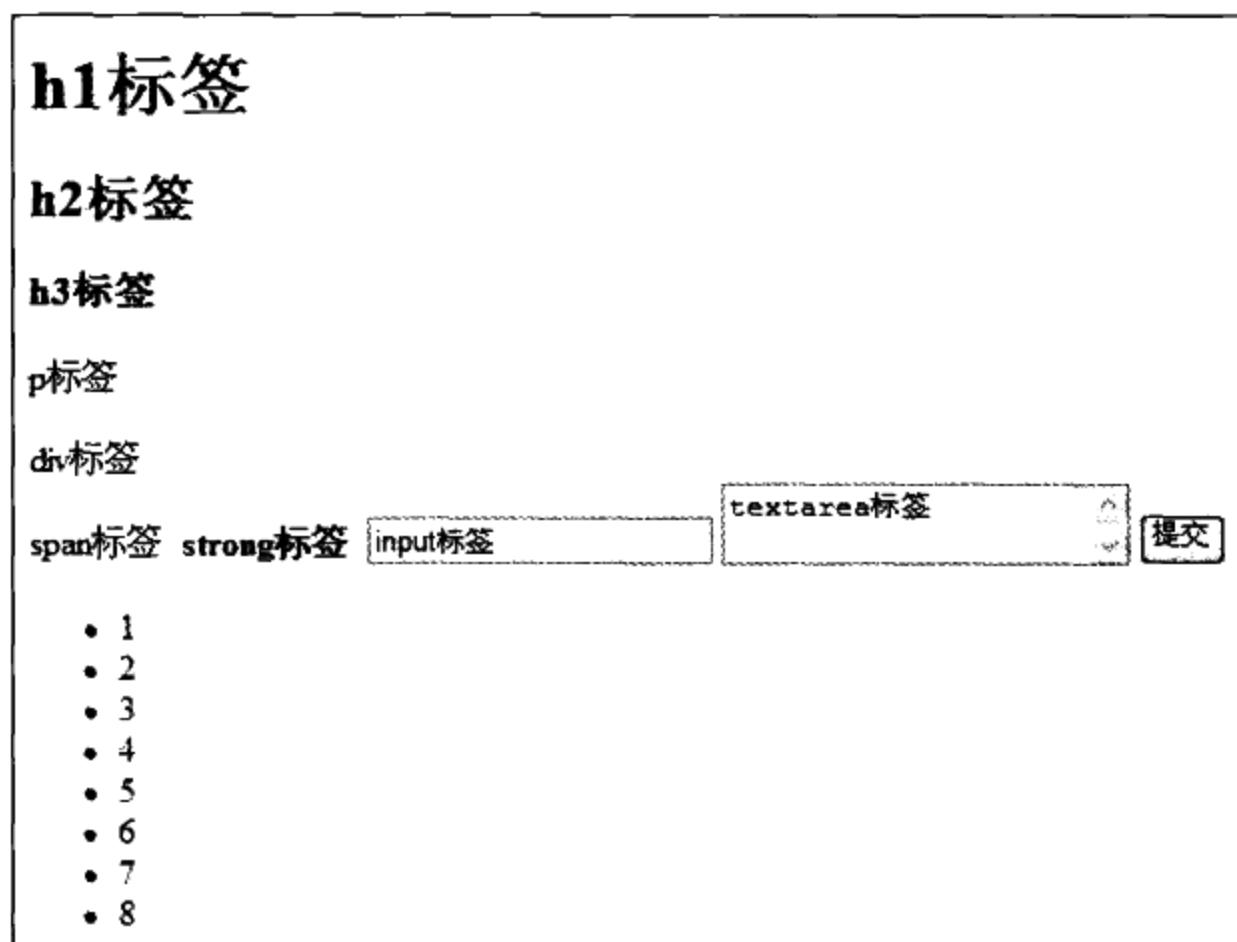


图 3-2 不加 CSS 代码的页面

我们给它链上一段 CSS 代码，如代码清单 3-2 所示。

代码清单 3-2 简单页面的 CSS 代码

---

```

h1{font-size:12px;font-weight:normal;}
h2{font-size:16px;}
h3{font-size:20px;font-weight:normal;}
p{size:normal;background:#333;color:#fff;padding:50px;font-size:30px;
font-weight:bold;}
div{display:inline;}
strong{display:block;padding:10px;border:1px dashed #000;margin:20px
50px;}
.text{width:300px;height:100px;}
textarea{width:150px;height:20px;border:4px solid #999;background:#ccc;
overflow:auto;}
.btn{border:3px solid green;background:black;color:#fff;}
ul{list-style:none;}
li{padding:10px;border:1px dashed #ccc;float:left;margin:0 2px;}

```

---

再来看看它在浏览中的表现，如图 3-3 所示。

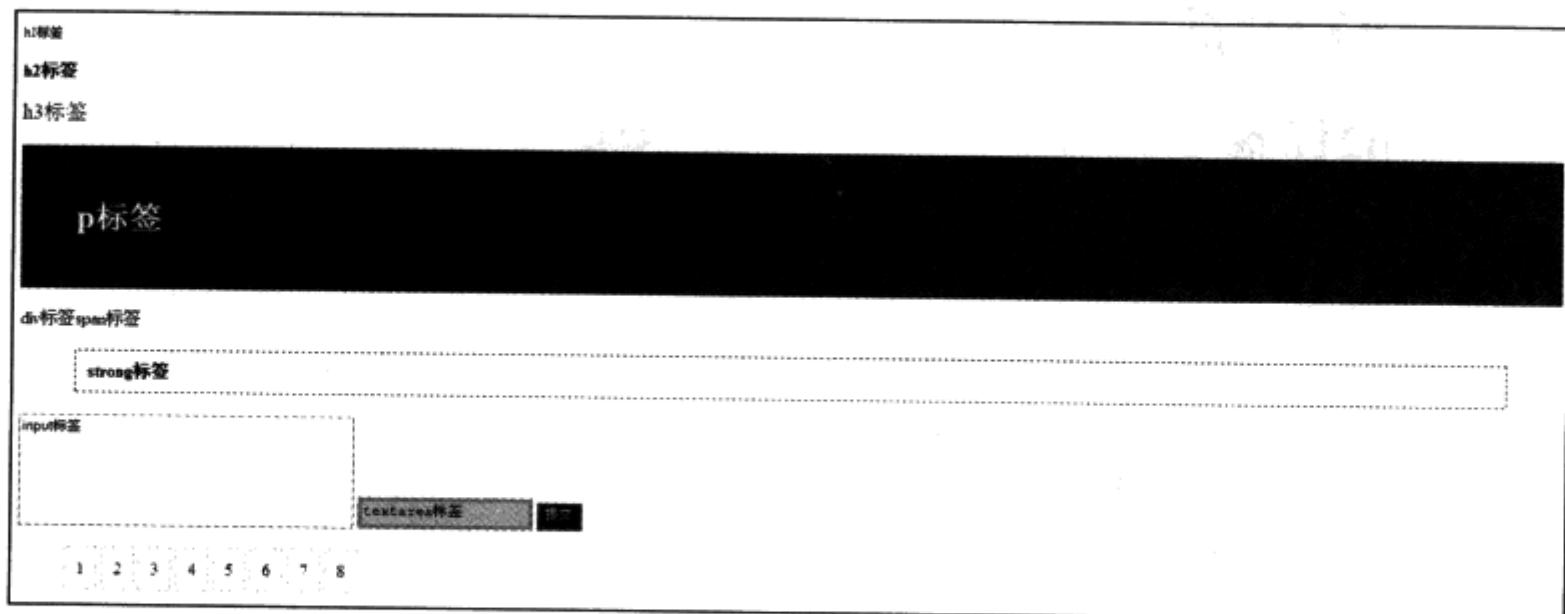


图 3-3 添加 CSS 代码后的页面

可以看到，加入 CSS 代码后可以让页面的外观截然不同。CSS 很强大，我们可以无视标签的默认样式，用 CSS 任意控制它们的表现形式。也正是因为 CSS 的强大，所以可以用完全不同的 HTML 代码制作出在视觉上相同的页面。

因为 CSS 很强大，所以无论有没有按照语义选择标签，我们都可以通过 CSS 实现需要的设计。但这也是把双刃剑，如果使用不当，容易使我们陷入 CSS 布局的一个误区——只要不是 table 布局，只要是通过 CSS 布局的，就是对的，就是符合 Web 标准的。如果只考虑最终视觉效果，而不考虑标签语义，其实又走上了 table 布局的老路。

事实上，CSS 布局只是 Web 标准的一部分。在 HTML、CSS、JavaScript 这三大元素中，HTML 才是最重要的，结构才是重点，样式是用来修饰结构的。正确的做法是，先确定 HTML，确定语义的标签，再来选用合适的 CSS。

如果考虑标签的语义，又如何判断选用的标签是否合适呢？

### 3.3 如何确定你的标签是否语义良好

CSS 是用来控制网页的样式的，那么是不是说，如果一个网页不使用 CSS，网

页就没有样式了呢？不是的，浏览器会根据标签的语义给定一个默认的样式。比如 h1、h2、h3 系列标签，会有加粗、上下边距等默认样式，而且字体会依次减小；ul 会有缩进、黑点的默认样式；strong 会有加粗的默认样式，em 会有斜体的默认样式。

也就是说，判断网页标签语义是否良好的一个简单方法就是：去掉样式，看网页结构是否组织良好有序，是否仍然有很好的可读性。语义良好的网页去掉样式后结构依然很清晰。对 Web 标准稍有了解的同行应该都知道“CSS 裸体日”吧？设立这个日子的目的就是为了提醒大家选用合适的 HTML 标签的重要性。

同样的设计图，不同的 HTML 标签可以通过不同的 CSS 实现同样的页面，但语义不好的 CSS 布局和语义良好的 CSS 布局在去样式后的表现却可能截然不同（见图 3-4 和图 3-5）。

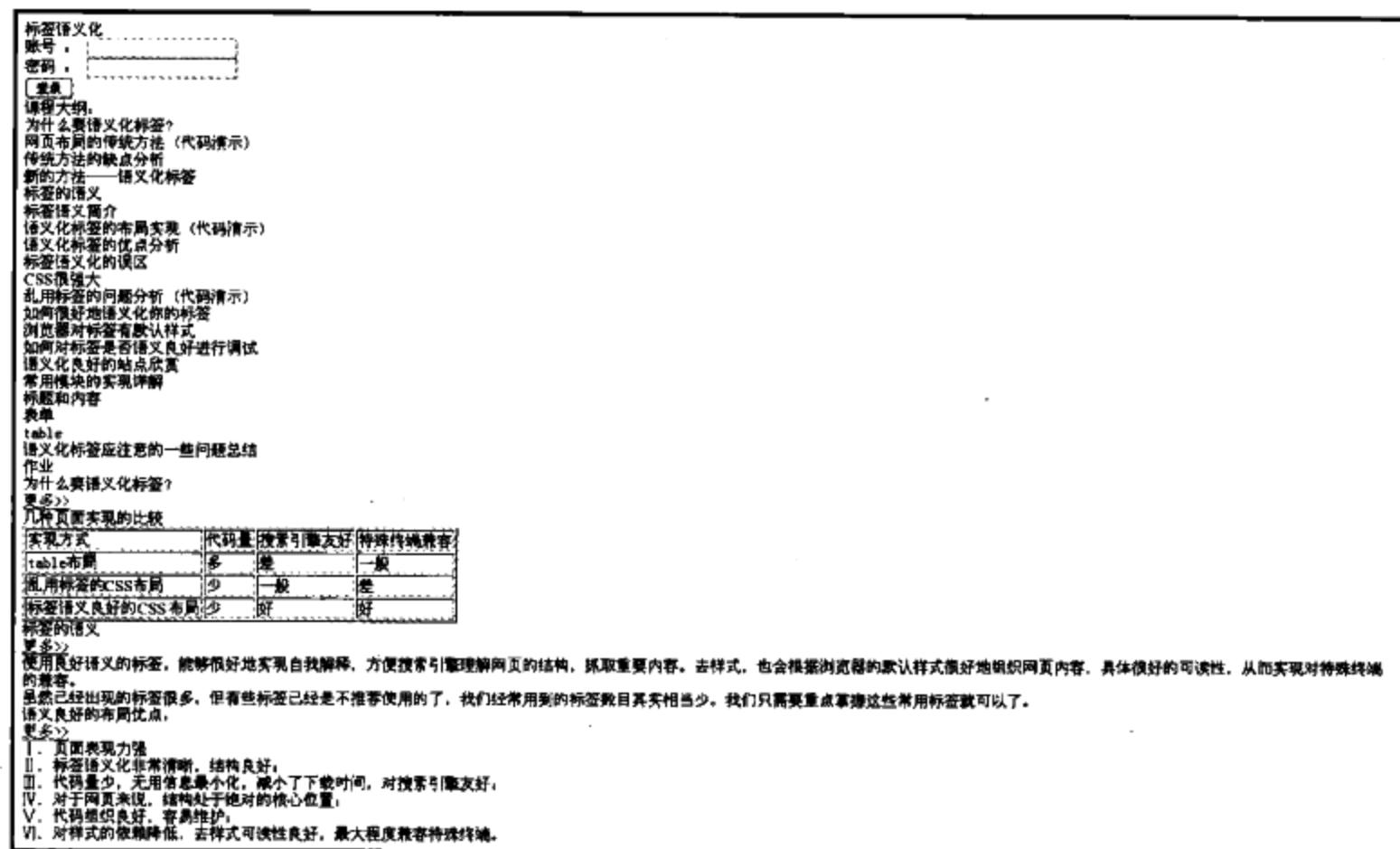


图 3-4 语义不好的结构去样式后在浏览器中的表现

**登录表单**

账号：

密码：

**课程大纲：**

- 为什么要语义化标签？
  - 网布间的传统方法（代码演示）
  - 传统方法的缺点分析
  - 新的方法——语义化标签
- 标签的语义
  - 标签语义简介
  - 语义化标签的布局实现（代码演示）
  - 语义化标签的优点分析
- 标签语义化的误区
  - CSS很强大
  - 乱用标签的问题分析（代码演示）
- 如何很好地语义化你的标签
  - 浏览器对标签的默认样式
  - 如何对标签是否语义良好进行调试
  - 语义化良好的站点欣赏
- 常用模块的实现详解
  - 标题和内容
  - 表单
  - `table`
  - 语义化标签应注意的一些问题总结
- 作业

**为什么要语义化标签？**

更多>

几种页面实现的比较		
实现方式	代码量	搜索引擎友好程度
table布局	多	差
乱用标签的CSS布局	少	一般
标签语义良好的CSS布局	少	好

**标签的语义**

更多>

使用良好语义的标签，能够很好地实现自我解释，方便搜索引擎理解网页的结构，抓取重要内容，去样式，也会根据浏览器的默认样式很好地组织网页内容，具体很好的可读性，从而实现对特殊终端的兼容。

虽然已经出现的标签很多，但有些标签已经是不推荐使用的了，我们经常用到的标签数目其实相当少。我们只需要重点掌握这些常用标签就可以了。

**语义良好的布局优点：**

更多>

1. 页面表现力强
2. 标签语义化非常清晰，结构良好。
3. 代码量少，无用信息最小化，减小了下载时间，对搜索引擎友好。
4. 对于网页来说，结构处于绝对的核心位置。
5. 代码组织良好，容易维护。
6. 所样式的依赖降低，去样式可读性良好，最大程度兼容特殊终端。

图 3-5 语义良好的结构去样式后在浏览器中的表现

从图 3-4 和图 3-5 中可以看出，如果选用的标签几乎全是不带语义的，那么在去样式后网页中几乎看不到任何结构信息，可读性非常差；如果选用的都是语义适合的标签，去样式后网页依然具有非常好的可读性。

除了去样式后的可读性外，值得重点提及的还有 `h` 标签。`h` 标签的含义是“标题”，搜索引擎对这个标签比较敏感，尤其是 `h1` 和 `h2`。一个语义良好的页面，`h` 标签应该是完整有序没有断层的。也就是说，要按照 `h1`、`h2`、`h3`、`h4` 这样依次排列下来，不要出现类似 `h1`、`h3`、`h4`，漏掉 `h2` 的情况。

如何对标签是否语义良好进行调试呢？推荐大家使用一个 Firefox 的插件——Web Developer。

图 3-6 是 Web Developer 的安装画面，图 3-7 中用粗线条圈起的部分是 Web Developer 安装完成后在 Firefox 中的样子。

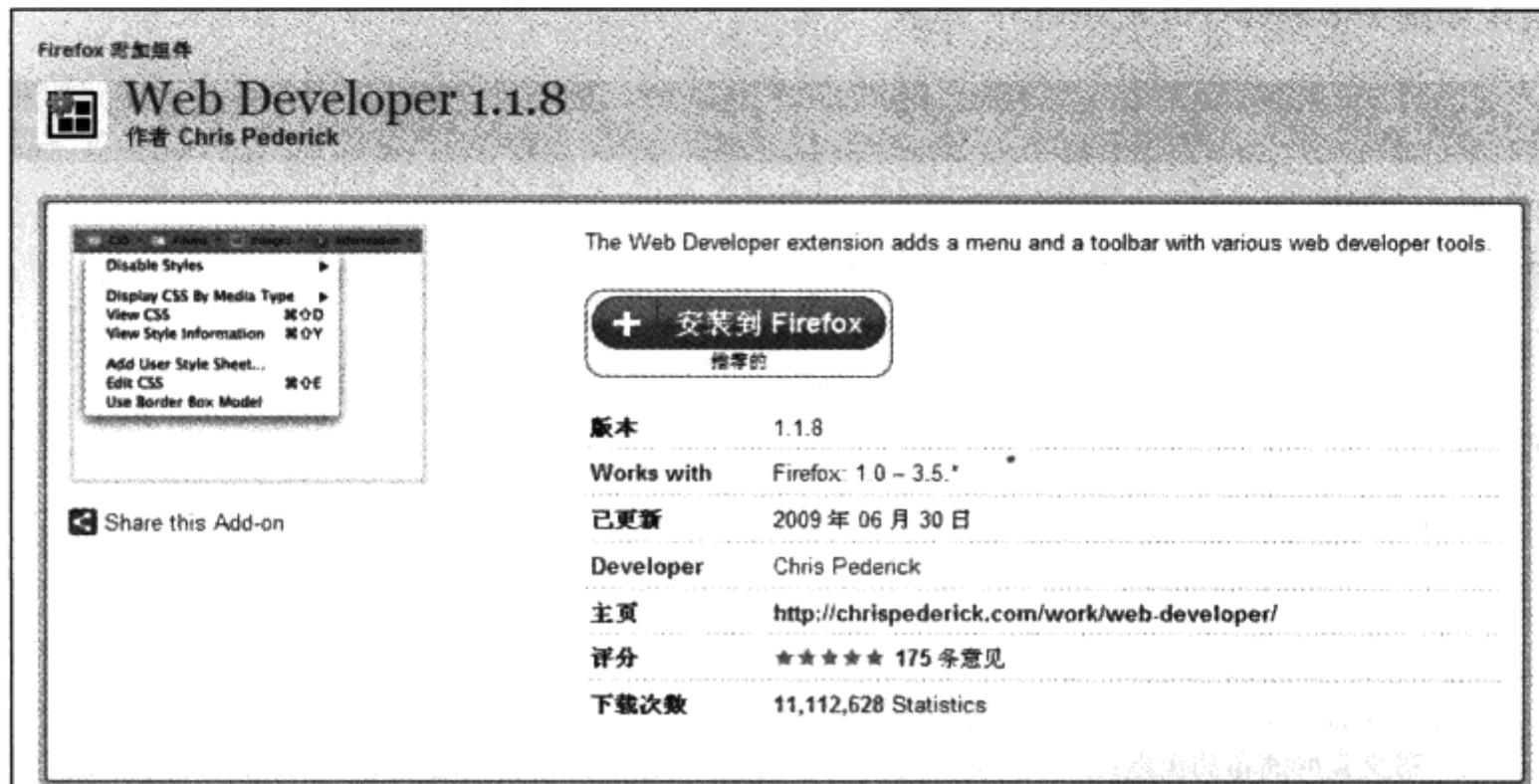


图 3-6 Web Developer 安装界面



图 3-7 Web Developer 工具条

Web Developer 和著名的 Firebug 一样，是 Firefox 上的一款网页调试插件。它提供了丰富的调试功能，其中有一项是禁用网页中的 CSS，快捷键是 Ctrl + Shift + S。有了

这个功能，我们可以轻松而快速地查看网页去样式后的表现。

依次点击“网页信息”→“查看文件大纲”，可以看到网页由 h 系列标签组织的大纲。

h 标签使用得当的网页和使用不当的网页、查看文件大纲时它们的效果会截然不同，如图 3-8 和图 3-9 所示。

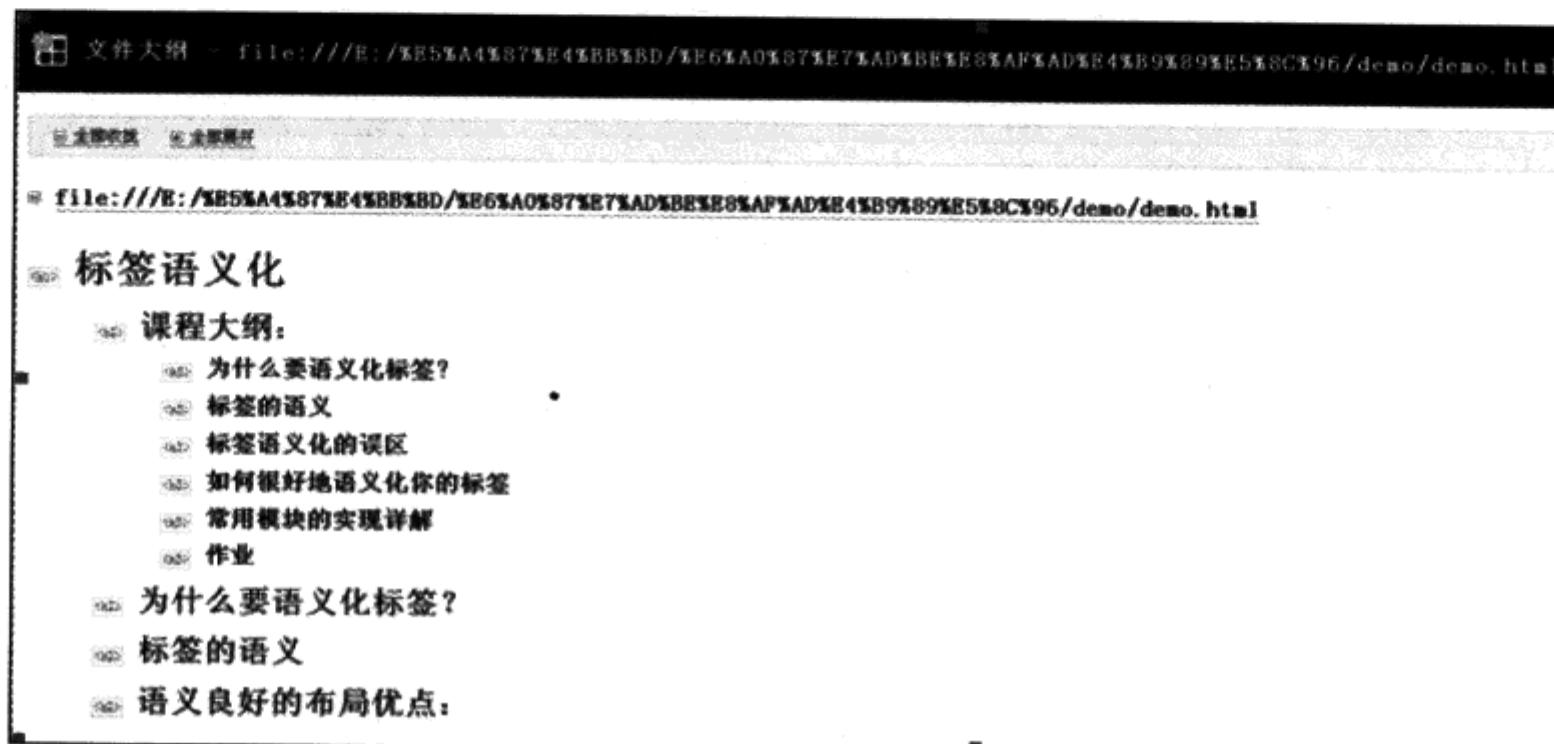


图 3-8 h 标签使用得当的网页的文件大纲

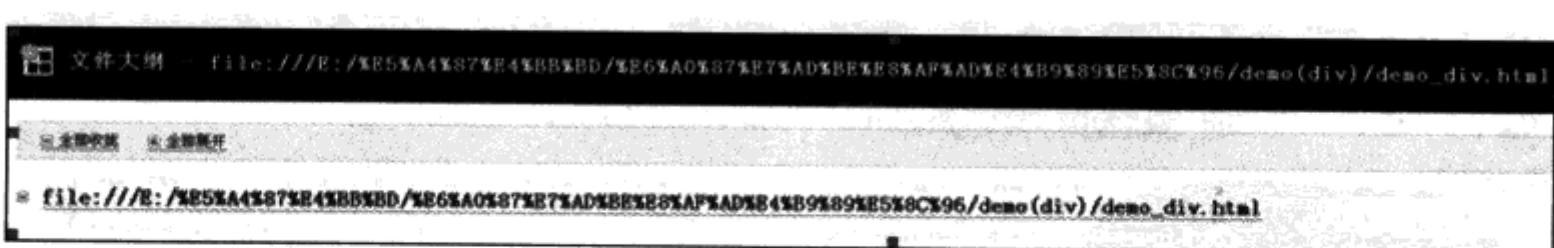


图 3-9 h 标签使用不当的网页的文件大纲

没有使用 h 系列标签的网页没有清晰的文件大纲，选用了合适 h 标签的网页的文件大纲看起来非常有条理。

W3C 的官方站语义非常优秀，我们一起来欣赏一下，其首页截图 3-10 所示。



**W3C WORLD WIDE WEB**  
Leading the Web to its Full Potential..

[Activities](#) | [Technical Reports](#) | [Site Index](#) | [New Visitors](#) | [About W3C](#) | [Join W3C](#) | [Contact W3C](#)

The World Wide Web Consortium (W3C) develops interoperate technologies (specifications, guidelines, software, and tools) to lead the Web to its full potential. W3C is a forum for information, commerce, communication, and collective understanding. On this page, you'll find W3C news, links to W3C specifications and ways to participate. New visitors can find help in [Getting Started at W3C](#). We encourage organizations to learn more about [W3C](#) and about [W3C Member](#).

**Validator Distortion Program**

[View Validator](#) [View Validator or Provider in Source of the Validator](#)

**W3C Supporters**

Help W3C by making a donation through the [W3C Intranet](#).

**Employment**

W3C is seeking a [Marketing Executive](#).  
Current W3C Fellow Positions available are [Business and Technology Communications Executive](#), Software Engineer.

**W3C Activities**

- Accessibility
- Africa
- Arabic
- CSS
- Current Document Formats (CDF)
- CSS Validator
- Data
- DCTE
- Efficient XML Interchange
- Geolocation
- Government
- GRDDL
- Health Care and Life Sciences
- HTML
- I18N
- HTTP
- HTML5
- IML
- JavaScript
- Media
- Mobile
- Mobile Web Initiatives (MWI)-MWIS
- Multilingual Initiatives
- XML
- Patent Policy
- PEG
- PIG
- SCALable
- Script and Page
- Test
- Rules
- Services
- Semantic Web
- Service Navigation Languages (SNL)
- SWRL
- SPARQL
- SPARQL API
- SPARQL Path
- SPARQL Update
- TAP
- Thread Test
- URI
- Vocabulary
- X3D
- XSL
- XSD
- XSLT
- XSLT 2.0
- XSLT 3.0
- XSLT 3.1
- XSLT 3.2
- XSLT 3.3
- XSLT 3.4
- XSLT 3.5
- XSLT 3.6
- XSLT 3.7
- XSLT 3.8
- XSLT 3.9
- XSLT 3.10
- XSLT 3.11
- XSLT 3.12
- XSLT 3.13
- XSLT 3.14
- XSLT 3.15
- XSLT 3.16
- XSLT 3.17
- XSLT 3.18
- XSLT 3.19
- XSLT 3.20
- XSLT 3.21
- XSLT 3.22
- XSLT 3.23
- XSLT 3.24
- XSLT 3.25
- XSLT 3.26
- XSLT 3.27
- XSLT 3.28
- XSLT 3.29
- XSLT 3.30
- XSLT 3.31
- XSLT 3.32
- XSLT 3.33
- XSLT 3.34
- XSLT 3.35
- XSLT 3.36
- XSLT 3.37
- XSLT 3.38
- XSLT 3.39
- XSLT 3.40
- XSLT 3.41
- XSLT 3.42
- XSLT 3.43
- XSLT 3.44
- XSLT 3.45
- XSLT 3.46
- XSLT 3.47
- XSLT 3.48
- XSLT 3.49
- XSLT 3.50
- XSLT 3.51
- XSLT 3.52
- XSLT 3.53
- XSLT 3.54
- XSLT 3.55
- XSLT 3.56
- XSLT 3.57
- XSLT 3.58
- XSLT 3.59
- XSLT 3.60
- XSLT 3.61
- XSLT 3.62
- XSLT 3.63
- XSLT 3.64
- XSLT 3.65
- XSLT 3.66
- XSLT 3.67
- XSLT 3.68
- XSLT 3.69
- XSLT 3.70
- XSLT 3.71
- XSLT 3.72
- XSLT 3.73
- XSLT 3.74
- XSLT 3.75
- XSLT 3.76
- XSLT 3.77
- XSLT 3.78
- XSLT 3.79
- XSLT 3.80
- XSLT 3.81
- XSLT 3.82
- XSLT 3.83
- XSLT 3.84
- XSLT 3.85
- XSLT 3.86
- XSLT 3.87
- XSLT 3.88
- XSLT 3.89
- XSLT 3.90
- XSLT 3.91
- XSLT 3.92
- XSLT 3.93
- XSLT 3.94
- XSLT 3.95
- XSLT 3.96
- XSLT 3.97
- XSLT 3.98
- XSLT 3.99
- XSLT 3.100
- XSLT 3.101
- XSLT 3.102
- XSLT 3.103
- XSLT 3.104
- XSLT 3.105
- XSLT 3.106
- XSLT 3.107
- XSLT 3.108
- XSLT 3.109
- XSLT 3.110
- XSLT 3.111
- XSLT 3.112
- XSLT 3.113
- XSLT 3.114
- XSLT 3.115
- XSLT 3.116
- XSLT 3.117
- XSLT 3.118
- XSLT 3.119
- XSLT 3.120
- XSLT 3.121
- XSLT 3.122
- XSLT 3.123
- XSLT 3.124
- XSLT 3.125
- XSLT 3.126
- XSLT 3.127
- XSLT 3.128
- XSLT 3.129
- XSLT 3.130
- XSLT 3.131
- XSLT 3.132
- XSLT 3.133
- XSLT 3.134
- XSLT 3.135
- XSLT 3.136
- XSLT 3.137
- XSLT 3.138
- XSLT 3.139
- XSLT 3.140
- XSLT 3.141
- XSLT 3.142
- XSLT 3.143
- XSLT 3.144
- XSLT 3.145
- XSLT 3.146
- XSLT 3.147
- XSLT 3.148
- XSLT 3.149
- XSLT 3.150
- XSLT 3.151
- XSLT 3.152
- XSLT 3.153
- XSLT 3.154
- XSLT 3.155
- XSLT 3.156
- XSLT 3.157
- XSLT 3.158
- XSLT 3.159
- XSLT 3.160
- XSLT 3.161
- XSLT 3.162
- XSLT 3.163
- XSLT 3.164
- XSLT 3.165
- XSLT 3.166
- XSLT 3.167
- XSLT 3.168
- XSLT 3.169
- XSLT 3.170
- XSLT 3.171
- XSLT 3.172
- XSLT 3.173
- XSLT 3.174
- XSLT 3.175
- XSLT 3.176
- XSLT 3.177
- XSLT 3.178
- XSLT 3.179
- XSLT 3.180
- XSLT 3.181
- XSLT 3.182
- XSLT 3.183
- XSLT 3.184
- XSLT 3.185
- XSLT 3.186
- XSLT 3.187
- XSLT 3.188
- XSLT 3.189
- XSLT 3.190
- XSLT 3.191
- XSLT 3.192
- XSLT 3.193
- XSLT 3.194
- XSLT 3.195
- XSLT 3.196
- XSLT 3.197
- XSLT 3.198
- XSLT 3.199
- XSLT 3.200
- XSLT 3.201
- XSLT 3.202
- XSLT 3.203
- XSLT 3.204
- XSLT 3.205
- XSLT 3.206
- XSLT 3.207
- XSLT 3.208
- XSLT 3.209
- XSLT 3.210
- XSLT 3.211
- XSLT 3.212
- XSLT 3.213
- XSLT 3.214
- XSLT 3.215
- XSLT 3.216
- XSLT 3.217
- XSLT 3.218
- XSLT 3.219
- XSLT 3.220
- XSLT 3.221
- XSLT 3.222
- XSLT 3.223
- XSLT 3.224
- XSLT 3.225
- XSLT 3.226
- XSLT 3.227
- XSLT 3.228
- XSLT 3.229
- XSLT 3.230
- XSLT 3.231
- XSLT 3.232
- XSLT 3.233
- XSLT 3.234
- XSLT 3.235
- XSLT 3.236
- XSLT 3.237
- XSLT 3.238
- XSLT 3.239
- XSLT 3.240
- XSLT 3.241
- XSLT 3.242
- XSLT 3.243
- XSLT 3.244
- XSLT 3.245
- XSLT 3.246
- XSLT 3.247
- XSLT 3.248
- XSLT 3.249
- XSLT 3.250
- XSLT 3.251
- XSLT 3.252
- XSLT 3.253
- XSLT 3.254
- XSLT 3.255
- XSLT 3.256
- XSLT 3.257
- XSLT 3.258
- XSLT 3.259
- XSLT 3.260
- XSLT 3.261
- XSLT 3.262
- XSLT 3.263
- XSLT 3.264
- XSLT 3.265
- XSLT 3.266
- XSLT 3.267
- XSLT 3.268
- XSLT 3.269
- XSLT 3.270
- XSLT 3.271
- XSLT 3.272
- XSLT 3.273
- XSLT 3.274
- XSLT 3.275
- XSLT 3.276
- XSLT 3.277
- XSLT 3.278
- XSLT 3.279
- XSLT 3.280
- XSLT 3.281
- XSLT 3.282
- XSLT 3.283
- XSLT 3.284
- XSLT 3.285
- XSLT 3.286
- XSLT 3.287
- XSLT 3.288
- XSLT 3.289
- XSLT 3.290
- XSLT 3.291
- XSLT 3.292
- XSLT 3.293
- XSLT 3.294
- XSLT 3.295
- XSLT 3.296
- XSLT 3.297
- XSLT 3.298
- XSLT 3.299
- XSLT 3.300
- XSLT 3.301
- XSLT 3.302
- XSLT 3.303
- XSLT 3.304
- XSLT 3.305
- XSLT 3.306
- XSLT 3.307
- XSLT 3.308
- XSLT 3.309
- XSLT 3.310
- XSLT 3.311
- XSLT 3.312
- XSLT 3.313
- XSLT 3.314
- XSLT 3.315
- XSLT 3.316
- XSLT 3.317
- XSLT 3.318
- XSLT 3.319
- XSLT 3.320
- XSLT 3.321
- XSLT 3.322
- XSLT 3.323
- XSLT 3.324
- XSLT 3.325
- XSLT 3.326
- XSLT 3.327
- XSLT 3.328
- XSLT 3.329
- XSLT 3.330
- XSLT 3.331
- XSLT 3.332
- XSLT 3.333
- XSLT 3.334
- XSLT 3.335
- XSLT 3.336
- XSLT 3.337
- XSLT 3.338
- XSLT 3.339
- XSLT 3.340
- XSLT 3.341
- XSLT 3.342
- XSLT 3.343
- XSLT 3.344
- XSLT 3.345
- XSLT 3.346
- XSLT 3.347
- XSLT 3.348
- XSLT 3.349
- XSLT 3.350
- XSLT 3.351
- XSLT 3.352
- XSLT 3.353
- XSLT 3.354
- XSLT 3.355
- XSLT 3.356
- XSLT 3.357
- XSLT 3.358
- XSLT 3.359
- XSLT 3.360
- XSLT 3.361
- XSLT 3.362
- XSLT 3.363
- XSLT 3.364
- XSLT 3.365
- XSLT 3.366
- XSLT 3.367
- XSLT 3.368
- XSLT 3.369
- XSLT 3.370
- XSLT 3.371
- XSLT 3.372
- XSLT 3.373
- XSLT 3.374
- XSLT 3.375
- XSLT 3.376
- XSLT 3.377
- XSLT 3.378
- XSLT 3.379
- XSLT 3.380
- XSLT 3.381
- XSLT 3.382
- XSLT 3.383
- XSLT 3.384
- XSLT 3.385
- XSLT 3.386
- XSLT 3.387
- XSLT 3.388
- XSLT 3.389
- XSLT 3.390
- XSLT 3.391
- XSLT 3.392
- XSLT 3.393
- XSLT 3.394
- XSLT 3.395
- XSLT 3.396
- XSLT 3.397
- XSLT 3.398
- XSLT 3.399
- XSLT 3.400
- XSLT 3.401
- XSLT 3.402
- XSLT 3.403
- XSLT 3.404
- XSLT 3.405
- XSLT 3.406
- XSLT 3.407
- XSLT 3.408
- XSLT 3.409
- XSLT 3.410
- XSLT 3.411
- XSLT 3.412
- XSLT 3.413
- XSLT 3.414
- XSLT 3.415
- XSLT 3.416
- XSLT 3.417
- XSLT 3.418
- XSLT 3.419
- XSLT 3.420
- XSLT 3.421
- XSLT 3.422
- XSLT 3.423
- XSLT 3.424
- XSLT 3.425
- XSLT 3.426
- XSLT 3.427
- XSLT 3.428
- XSLT 3.429
- XSLT 3.430
- XSLT 3.431
- XSLT 3.432
- XSLT 3.433
- XSLT 3.434
- XSLT 3.435
- XSLT 3.436
- XSLT 3.437
- XSLT 3.438
- XSLT 3.439
- XSLT 3.440
- XSLT 3.441
- XSLT 3.442
- XSLT 3.443
- XSLT 3.444
- XSLT 3.445
- XSLT 3.446
- XSLT 3.447
- XSLT 3.448
- XSLT 3.449
- XSLT 3.450
- XSLT 3.451
- XSLT 3.452
- XSLT 3.453
- XSLT 3.454
- XSLT 3.455
- XSLT 3.456
- XSLT 3.457
- XSLT 3.458
- XSLT 3.459
- XSLT 3.460
- XSLT 3.461
- XSLT 3.462
- XSLT 3.463
- XSLT 3.464
- XSLT 3.465
- XSLT 3.466
- XSLT 3.467
- XSLT 3.468
- XSLT 3.469
- XSLT 3.470
- XSLT 3.471
- XSLT 3.472
- XSLT 3.473
- XSLT 3.474
- XSLT 3.475
- XSLT 3.476
- XSLT 3.477
- XSLT 3.478
- XSLT 3.479
- XSLT 3.480
- XSLT 3.481
- XSLT 3.482
- XSLT 3.483
- XSLT 3.484
- XSLT 3.485
- XSLT 3.486
- XSLT 3.487
- XSLT 3.488
- XSLT 3.489
- XSLT 3.490
- XSLT 3.491
- XSLT 3.492
- XSLT 3.493
- XSLT 3.494
- XSLT 3.495
- XSLT 3.496
- XSLT 3.497
- XSLT 3.498
- XSLT 3.499
- XSLT 3.500
- XSLT 3.501
- XSLT 3.502
- XSLT 3.503
- XSLT 3.504
- XSLT 3.505
- XSLT 3.506
- XSLT 3.507
- XSLT 3.508
- XSLT 3.509
- XSLT 3.510
- XSLT 3.511
- XSLT 3.512
- XSLT 3.513
- XSLT 3.514
- XSLT 3.515
- XSLT 3.516
- XSLT 3.517
- XSLT 3.518
- XSLT 3.519
- XSLT 3.520
- XSLT 3.521
- XSLT 3.522
- XSLT 3.523
- XSLT 3.524
- XSLT 3.525
- XSLT 3.526
- XSLT 3.527
- XSLT 3.528
- XSLT 3.529
- XSLT 3.530
- XSLT 3.531
- XSLT 3.532
- XSLT 3.533
- XSLT 3.534
- XSLT 3.535
- XSLT 3.536
- XSLT 3.537
- XSLT 3.538
- XSLT 3.539
- XSLT 3.540
- XSLT 3.541
- XSLT 3.542
- XSLT 3.543
- XSLT 3.544
- XSLT 3.545
- XSLT 3.546
- XSLT 3.547
- XSLT 3.548
- XSLT 3.549
- XSLT 3.550
- XSLT 3.551
- XSLT 3.552
- XSLT 3.553
- XSLT 3.554
- XSLT 3.555
- XSLT 3.556
- XSLT 3.557
- XSLT 3.558
- XSLT 3.559
- XSLT 3.560
- XSLT 3.561
- XSLT 3.562
- XSLT 3.563
- XSLT 3.564
- XSLT 3.565
- XSLT 3.566
- XSLT 3.567
- XSLT 3.568
- XSLT 3.569
- XSLT 3.570
- XSLT 3.571
- XSLT 3.572
- XSLT 3.573
- XSLT 3.574
- XSLT 3.575
- XSLT 3.576
- XSLT 3.577
- XSLT 3.578
- XSLT 3.579
- XSLT 3.58

## > XHTML 2 Working Group Expected to Stop Work End of 2009, W3C to Increase Resources on HTML 5

2009-07-02 Today the Director announces that when the XHTML 2 Working Group charter expires at the end of 2009, the charter will not be renewed. By doing so, and by increasing resources in the HTML Working Group, W3C hopes to accelerate the progress of HTML 5 and clarify W3C's position regarding the future of HTML. A FAQ answers questions about the future of deliverables of the XHTML 2 Working Group, and the status of various discussions related to HTML. Learn more about the [HTML Activity](#) (External)

## > Summary of Workshop on Speaker Biometrics and VoiceXML 3.0 Available

2009-07-02 W3C has published a summary and fatalities of the [Workshop on Speaker Biometrics and VoiceXML 3.0](#) that took place in Menlo Park, California on 5-6 March. Participants from 15 organizations focused discussion on Speaker Identification and Verification (SIV) functionality within VoiceXML 3.0, and identifying and prioritizing directions for the functionality. The major outcomes from the workshop are confirmation that SIV fits into the VoiceXML space and creation of a "Menlo Park Model" & SIV extendible VoiceXML architecture. The Working Group will continue to discuss how to include the requirements expressed at the Workshop into VoiceXML 3.0 and improve the specification. Learn more about the [Voice Browser Activity](#) (External)

## > First Draft of SPARQL New Features and Rationale

2009-07-01 The SPARQL Working Group has published the First Public Working Draft of [SPARQL New Features and Rationale](#). This document provides an overview of the main new features of SPARQL and their rationale. This is an update to SPARQL adding several new features that have been agreed by the SPARQL WG. These language features were determined based on real applications and user and tool-developer experience. Learn more about the [SPARQL Web Activity](#) (External)

## > W3C Talks in July

2009-07-01 Browse [W3C presentations and reports](#) also available as an [RSS feed](#) (External)

- 2 July, Bratislava, Austria: W3C: a Use Case for Web Technologies Klaus Birkensek presents at NCITA, Government Research Laboratory
- 7 July, Barcelona, Spain: Acces a la Informació Pública i Reutes Soïisches José-Manuel Alonso participates in a panel at "Catalunya Internet, Director i Política"
- 14 July, Linz, Austria: WCAG 2.0 ist da - was nun? Shadi Abou-Zahra presents at IUI Forum 2009
- 20 July, Boston, USA: Web Accessibility, Universal Design, and Standardization Jude Brumbaugh participates in a panel of "Accessing the Future: A global collaboration's vision for accessibility in the next decade"
- 20 July, Seattle, WA, USA: Accessibility It's for Everyone And Everything Shawn Henry presents at Web Design Week 2009 Session
- 21 July, Seattle, WA, USA: Accessibility In a Web 2.0 World Shawn Henry presents at Web 2.0 Seattle 2009 Session: "Designing Usability: Designing Code" (Session)
- 24 July, San Diego, USA: WCAG 2.0 Test Samples Shadi Abou-Zahra presents at WCAG 2.0 Test Samples 2009
- 27 July, Raleigh, North Carolina, USA: Extensible XML, HTML, Doug Schepers presents at The Summer XML 2009 Conference
- 29 July, Raleigh, North Carolina, USA: Open Graphics and the Semantic Web Scalable Vector Graphics and Canvas Doug Schepers presents at The Summer XML 2009 Conference
- View upcoming W3C by country
- More info...

## > Last Call: CSS3 module: Multi-column layout

2009-06-30 The Cascading Style Sheets (CSS) Working Group has published a Last Call Working Draft of [CSS3 module: Multi-column layout](#). This module describes multi-column layout in CSS. It builds on the CSS3 Box model module and adds functionality to how the content of an element uses multiple columns. Comments are welcome through 01 October. Learn more about the [CSS Activity](#) (External)

## > Two SML Notes: XLink Reference Scheme, EPR-Based Reference Schemes

2009-06-30 The Semantic Markup Language Working Group has published two Working Group Notes: [The SML XLink Reference Scheme and Languages for SML: SML-based Reference Schemes](#). The Element Modelling language specification extends the Extensible Markup Language and XML Schema with a mechanism for incorporating into XML documents references to other documents or document fragments. The first note addresses the construction of an SML reference scheme based on the XLink language. The second addresses the construction of SML reference schemes for document or document fragment references that employ Web Addressing endpoint references (EPRs). Learn more about the Extensible Markup Language (XML) Activity (External)

## > Steve Bratt to Assume Full-Time Role as Web Foundation CEO

2009-06-29 As of 30 June, Steve R. Bratt will step down from his role as W3C CEO in order to pursue full-time the role of CEO of the [World Wide Web Foundation](#). The Web Foundation was established in September 2008 with a mission to advance the Web, connect humanity, and empower people. Steve has been part-time CEO of the Web Foundation on since then.

While W3C CEO and then CEO, Steve was responsible for W3C's worldwide operations and outreach, including overall management of Member relations, the W3C Process, the staff, strategic planning, budget, legal matters, external liaison and more. Steve's progressive and thoughtful leadership at W3C was informed by previous experiences in research, industry, and government, where he served in scientific and administrative delegations among others.

While W3C seeks to fit the coat around Ralph Swick assuming Steve's leadership responsibilities, Thomas Roemmer steps up in the interim to take on the role of Technology and Society Domain Lead.

The mission of the Web Foundation complements that of W3C, and the two organizations will collaborate to coordinate their efforts to make the Web useful and available to all. W3C looks forward to Steve's successful leadership of the Web Foundation. (External)

## > W3C Invites Implementations of Widgets 1.0: Digital Signatures

2009-06-25 The Web Applications Working Group invites implementation of the Candidate Recommendation of [Widgets 1.0: Digital Signatures](#). Widgets are full-fledged client-side applications that are authored using Web standards and packaged for distribution. This document defines a profile of the XSD, Signature Syntax and Processing 1.1 specification to allow a widget package to be digitally signed, helping to ensure continuity of authorship and distributorship. Learn more about the [Rich Web Client Activity](#) (External)

## > First Authorized Translation of WCAG 2.0 Published

2009-06-20 W3C announces the French Authorized Translation of the [Candidate Recommendation of WCAG 2.0](#) (Status: 2009-06-19/2009-06-20). This is the first of seven planned WCAG 2.0 translations: British Portuguese, Chinese (Chinese, Cantonese), Czech, Danish, Dutch, German, Hindi, Hungarian, Italian, Japanese, Korean, Portuguese, Russian, Spanish, Swedish, and other languages. Translations are listed on the [WCAG 2.0 Translations](#) page and announced via the W3C Interest Group mailing list and W3C RSS feed. Learn more about [Translations at W3C](#), [Authorized Translations](#), [WCAG 2.0](#), and the [Web Accessibility Initiative \(WAI\)](#) (External)

## Fast News

W3C issues fast track for [Digital Signatures](#) who have established financial, at though a donation of goods to W3C.

Read about it [here](#). W3C fast tracks about this page. Download this page with [RSS 1.0](#) or [PDF](#) (recommended for site summaries).

W3C logo: [W3C](#) (W3C logo) Status: 2009-07-09 17:56:53

W3C logo:

图 3-10 (续)



首页去样式后截图如下：

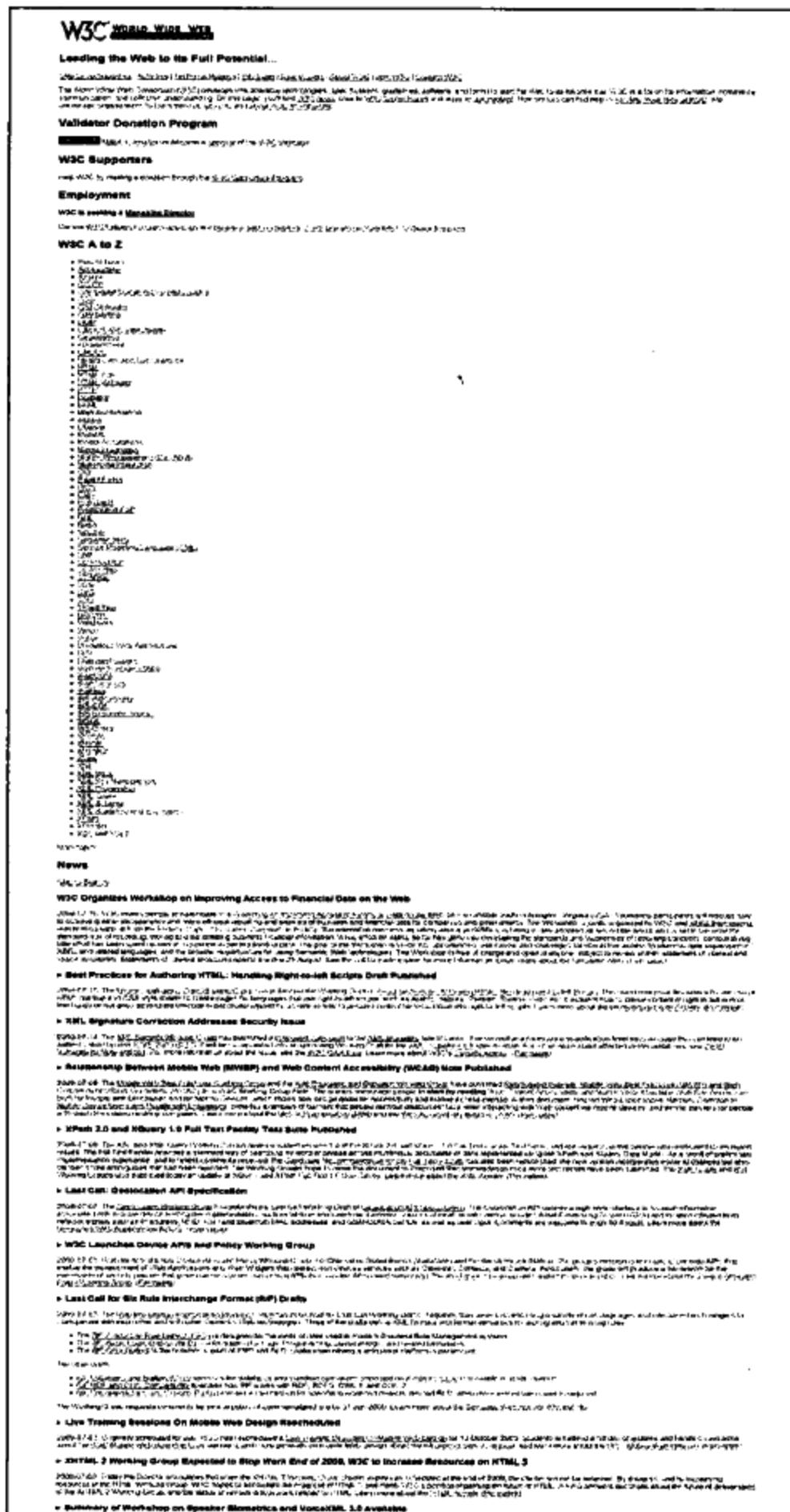


图 3-11 W3C 官方站首页去样式后截图

The screenshot shows the W3C homepage from 1999. At the top, there's a banner with the text "W3C Home Page". Below it, a large image of a person sitting at a computer. To the right of the image, the text reads: "A group of 15-20 people gathered together and working on their computers in their Park Extension in CA office. Participants from 15 organizations received grants from the Open Source Foundation and the Mozilla Foundation to work on the XUL interface. The main "baseline" from the workshop were determined that the XUL was the standard, flexible and extensible language for the XUL interface. The workshop group will now take the XUL and standardize the interface and continue the development. The Mozilla Foundation will help the XUL interface become more standards based or non-explicitly defined."

**News**

- First Draft of SPARC New Features and Rationale**
- W3C Takes in July**
- W3C News: XML Reference Scheme, SPM-Based Reference Schemes**
- Steve Bratt Assumes Full-Time Role as Web Foundation CEO**
- W3C Invites Implementations of Widgets 1.0: Digital Signatures**
- First Authorized Translation of WCAG 2.0 Published**

**Exhibits**

**Search**

**Members**

**Get Involved**

**Introduction**

**W3C Team**

**Presentations**

**News Room**

**World Offices**

**Systems**

**World Wide Web Foundation**

© 1999-2000, W3C, MIT, INRIA, Keio University. All Rights Reserved. 1999 © MIT, INRIA, Keio University. All Rights Reserved. W3C, World Wide Web Foundation, and the W3C logo are trademarks of the W3C Consortium. W3C, the W3C logo, and the W3C Consortium logo are registered trademarks of the W3C Consortium.

图 3-11 (续)

其首页文件大纲如图 3-12 所示。

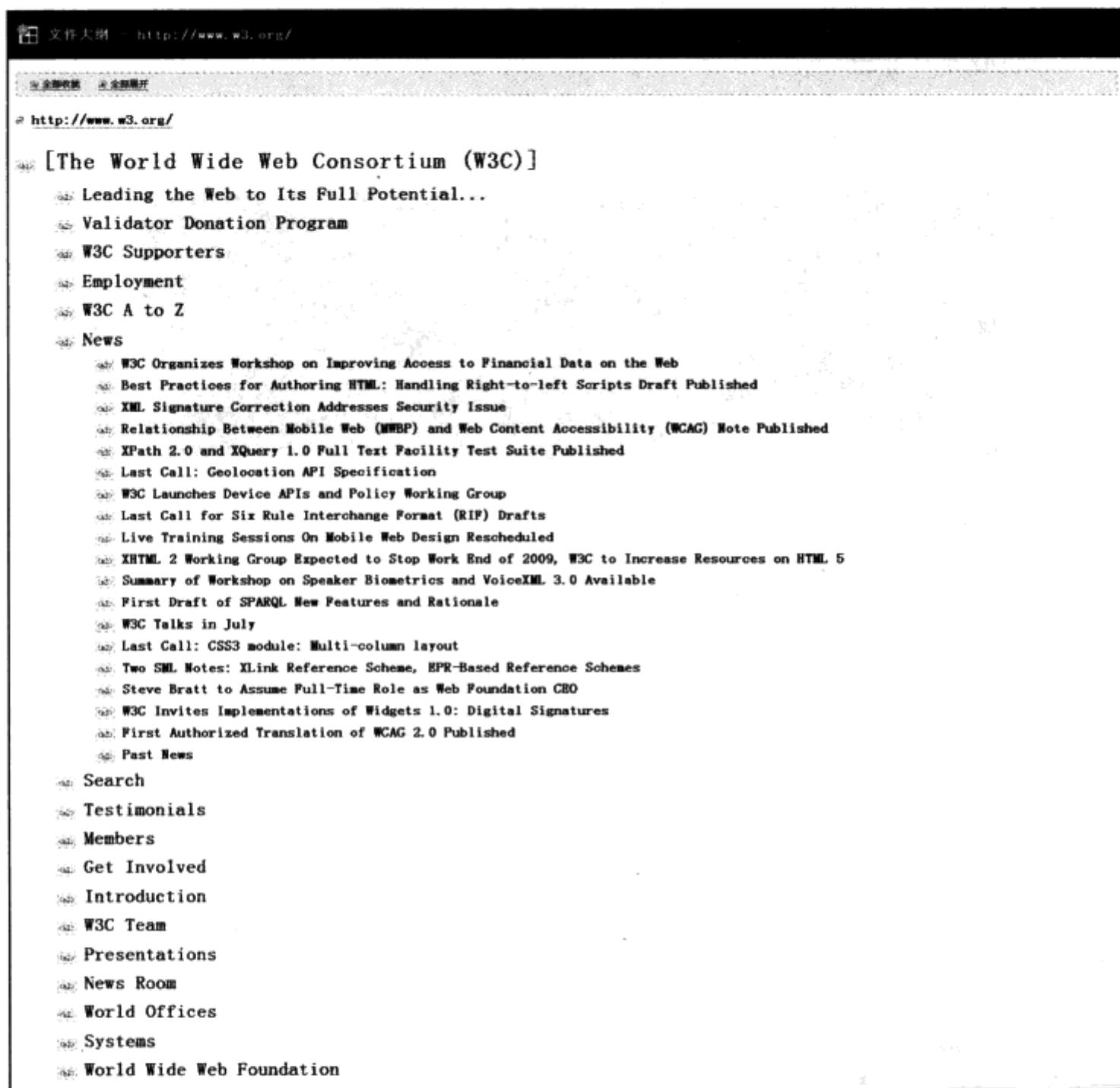


图 3-12 W3C 官方站首页文件大纲

W3C 的官方站在去样式后依然有良好的结构和可读性，而且文件大纲也脉络分明、井然有序，为“什么是语义化良好的网页”做出了标榜作用。

实际上，在开发中会遇到各种各样的情况，如何选用正确的标签呢？下节我将以图 3-1 为例，对几个简单的但又非常典型的模块进行详细介绍。

## 3.4 常见模块你真的很了解吗

### 3.4.1 标题和内容

标题和内容模块如图 3-13 所示。

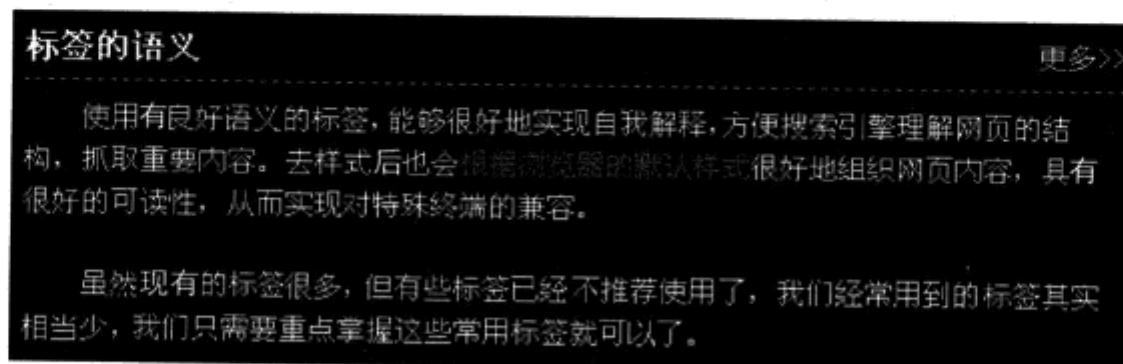


图 3-13 标题和内容模块

这种标题配内容的模块，我们可以用如下几种方案来实现同样的效果。

**方案一** 实现方法如代码清单 3-3 所示。

代码清单 3-3 标题和内容模块实现方案一

html 部分：

```
<div class="h2">标签的语义<a href="#">更多>&gt;</a></div>
<div class="p">段落一的内容……<span class="strong">根据浏览器的默认样式
</span>. . . </div>
<div class="p">段落二的内容……</div>
```

CSS 部分：

```
.h2{position:relative; border-bottom:1px dashed #ffff;}
.h2 a{position:absolute; right:0; top:0;}
.p{text-indent:2em; line-height:150%; margin:0 0 20px 0;}
.strong{color:red;}
```

视觉效果达到了，再来看看它的语义吧，如代码清单 3-4 所示。

代码清单 3-4 标题和内容模块实现方案一的语义

```
<分隔 class="h2">标签的语义<锚点 href="#">更多>&gt;</锚点></分隔>
<分隔>段落一的内容……<范围 class="strong">根据浏览器的默认样式 </范围>……
</分隔>
<分隔>段落二的内容</分隔>
```

我们只能看到“分隔”、“范围”这样的无语义标签，从标签上看不出结构的逻辑。

这显然是不行的，我们需要用标签让代码能够清晰地透露出“标题”，“内容”，“被强调的文本”等信息。我们把它改进一下，换用第二种方案。

**方案二** 实现方法如代码清单3-5所示。

代码清单3-5 标题和内容模块实现方案二

---

html部分：

```
<h2>标签的语义 <a href="#">更多 &gt;&gt;</a> </h2>
<p>段落一的内容……<strong>根据浏览器的默认样式 </strong>……</p>
<p>段落二的内容</p>
```

CSS部分：

```
h2{position:relative; border-bottom:1px dashed #ffff;}
h2 a{position:absolute; right:0; top:0; }
p{text-indent:2em; line-height:150%; margin:0 0 20px 0; }
strong{color:red; font-weight:normal}
```

---

再来看看方案二的语义吧，如代码清单3-6所示。

代码清单3-6 标题和内容模块实现方案二的语义

---

```
<二级标题>标签的语义<锚点 href="#">更多 &gt;&gt;</锚点> </二级标题>
<段落>段落一的内容……<强调>根据浏览器的默认样式 </强调>……</段落>
<段落>段落二的内容</段落>
```

---

方案二大有改进，从标签中能清楚地看到标题和内容的划分，也能看到“根据浏览器的默认样式”被强调了。嗯，不错。

等等，`<锚点 href="#">更多 &gt;&gt;</锚点>`被包在了`<二级标题>`标签中，它属于二级标题吗？不对，虽然在视觉设计上它和“标签语义”是放在同一行的，但事实上它并不属于标题！嗯，让我们再改进一下，如方案三所示。

**方案三** 如代码清单3-7所示。

代码清单3-7 标题和内容模块实现方案三

---

html部分：

```
<h2>标签的语义</h2>
<a href="#">更多 &gt;&gt;</a>
<p>段落一的内容……<strong>根据浏览器的默认样式 </strong>……</p>
<p>段落二的内容</p>
```

CSS部分：

```
h2{}
a{}
```

---

```
p{text-indent:2em;line-height:150%;margin:0 0 20px 0;}  
strong{color:red;font-weight:normal}
```

---

方案三的语义就很理想了，如代码清单 3-8 所示。

代码清单 3-8 标题和内容模块实现方案三的语义

---

```
<二级标题>标签的语义</二级标题>  
<锚点 href="#">更多 &gt;&gt;</锚点>  
<段落>段落一的内容……<强调>根据浏览器的默认样式 </强调>……</段落>  
<段落>段落二的内容</段落>
```

---

语义是不错了，但根据方案三的 HTML 结构，我们很难通过 CSS 完成设计图中的设计。怎么办呢？还记得在 3.1 节中留下的那个悬念吗？这个时候就该无意义标签派上用场了！我们在方案三的 HTML 基础上添加适当的无语义标签 div 和 span，如代码清单 3-9 所示。

代码清单 3-9 标题和内容模块实现方案三的改进版

---

html 部分：

```
<div class="title">  
    <h2>标签的语义</h2>  
    <a href="#">更多 &gt;&gt;</a>  
</div>  
<p>段落一的内容……<strong>根据浏览器的默认样式 </strong>……</p>  
<p>段落二的内容</p>
```

CSS 部分：

```
.title{border-bottom:1px dashed #fff;text-align:right;}  
.title h2{float:left;}  
p{text-indent:2em;line-height:150%;margin:0 0 20px 0;}  
strong{color:red;font-weight:normal}
```

---

OK，现在我们的代码既精简，语义又清晰。这里需要特别说明的是，当页面内标签无法满足设计需要时，才会适当添加 div 和 span 等无语义标签来辅助实现。

### 3.4.2 表单

表单模块如图 3-14 所示。



图 3-14 表单模块

这种表单模块，我们可以用如下两种方案实现其效果。

**方案一** 实现方法如代码清单 3-10 所示。

代码清单 3-10 表单模块实现方案一

---

```
<form action="" method="" class="fieldset">
    <div><span>账号: </span> <input type="text" id="name" /></div>
    <div><span>密码: </span> <input type="password" id="pw" /></div>
    <input type="submit" value="登录" class="subBtn" />
</form>
```

---

来分析一下它的语义，如代码清单 3-11 所示。

代码清单 3-11 表单模块实现方案一的语义

---

```
<表单 action="" method="" class="fieldset">
    <分隔><范围>账号 : </范围> <表单项 type="text" id="name" /></分隔>
    <分隔><范围>密码 : </范围> <表单项 type="password" id="pw" /></分隔>
    <表单项 type="submit" value="登录" class="subBtn" />
</表单>
```

---

这种方法虽然能实现视觉效果，但“账号:”、“密码:”和它们对应的输入框之间没有语义上的照应，表单的用途也不清楚。

我们将方案一改进一下，如方案二所示。

**方案二** 实现方法如代码清单 3-12 所示。

代码清单 3-12 表单模块实现方案二

---

```
<form action="" method="">
    <fieldset>
        <legend>登录表单</legend>
        <p><label for="name">账号: </label><input type="text" id="name"/>
    </p>
        <p><label for="pw">密码: </label> <input type="password" id="pw" />
    </p>
        <input type="submit" value="登录" class="subBtn" />
    </fieldset>
</form>
```

---

我们来看看方案二的语义，如代码清单 3-13 所示。

代码清单 3-13 表单模块实现方案二的语义

---

```
<表单 action="" method="" class="fieldset">
  <域集>
    <域集名>登录表单</域集名>
    <段落><表单项说明 for="name">账号 : </表单项说明> <表单项 type=
"text" id="name" /></段落>
    <段落><表单项说明 for="pw">密码 : </表单项说明> <表单项 type=
"password" id="pw" /></段落>
    <表单项 type="submit" value="登录" class="subBtn" />
  </域集>
</表单>
```

---

方案二改进了方案一的不足，语义上清晰了很多。一般来说，表单域要用 `fieldset` 标签包起来，并用 `legend` 标签说明表单的用途。因为 `fieldset` 默认有边框，而 `legend` 也有默认的样式，为满足设计需要，我们可以将 `fieldset` 的“`border`”设为“`none`”，把 `legend` 的“`display`”设为“`none`”，以此来兼顾语义和设计两方面的要求。每个 `input` 标签对应的说明文本都需要使用 `label` 标签，并且通过为 `input` 设置 `id` 属性，在 `label` 标签中设置“`for = someld`”来让说明文本和相应的 `input` 关联起来。

### 3.4.3 表格

表格如图 3-15 所示。

几种页面实现的比较			
实现方式	代码量	搜索引擎友好	特殊终端兼容
table 布局	多	差	一般
乱用标签的 CSS 布局	少	一般	差
标签语义良好的 CSS 布局	少	好	好

图 3-15 表格模块

在 CSS 布局日渐流行的今天，很多人患上了“`table` 恐惧症”，仿佛用了 `table` 就落后了，就是不注重语义，就是违反 Web 标准，于是宁愿用 `ul` 来模拟 `table` 也不使用 `table`。这从一个极端走到了另一个极端。其实，`table` 标签有它自己的语义和用途，虽然

在 CSS 布局中 table 不推荐用来布局，但它仍有自己的一席之地——在二维数据展示方面它是最好的选择。

我们可以选择 table 来实现图 3-15 的效果，有如下两种方案。

**方案一** 实现方法如代码清单 3-14 所示。

代码清单 3-14 表格模块实现方案一

---

```
<div class="caption">几种页面实现的比较</div>
<table border="1">
    <tr class="thead"><td class="th">实现方式</td><td class="th">代码量
    </td><td class="th">搜索引擎友好</td><td class="th">特殊终端兼容</td></tr>
        <tr><td class="th">table 布局</td><td>多</td><td>差</td><td>一般
        </td></tr>
        <tr><td class="th">乱用标签的 CSS 布局</td><td>少</td><td>一般
        </td><td>差</td></tr>
        <tr><td class="th">标签语义良好的 CSS 布局</td><td>少</td><td>好
        </td><td>好</td></tr>
    </table>
```

---

来分析一下它的语义，如代码清单 3-15 所示。

代码清单 3-15 表格模块实现方案一的语义

---

```
<分隔 class="caption">几种页面实现的比较</分隔>
<表格 border="1">
    <表格行 class="thead"><表格单元格 class="th">实现方式</表格单元格><表格单元格 class="th">代码量
    </表格单元格><表格单元格 class="th">搜索引擎友好</表格单元格><表格单元格 class="th">特殊终端兼容</表格单元格></表格行>
        <表格行><表格单元格 class="th">table 布局</表格单元格><表格单元格>多</表格单元格><表格单元格>差</表格单元格><表格单元格>一般</表格单元格></表格行>
        <表格行><表格单元格 class="th">乱用标签的 CSS 布局</表格单元格><表格单元格>少</表格单元格><表格单元格>一般</表格单元格><表格单元格>差</表格单元格></表格行>
        <表格行><表格单元格 class="th">标签语义良好的 CSS 布局</td><表格单元格>少</表格单元格><表格单元格>好</表格单元格><表格单元格>好</表格单元格></表格行>
    </表格>
    <分隔><范围>账号 : </范围> <表单项 type="text" id="name" /></分隔>
    <分隔><范围>密码 : </范围> <表单项 type="password" id="pw" /></分隔>
    <表单项 type="submit" value="登录" class="subBtn" />
</表单>
```

---

虽然在视觉上可以看出“几种页面实现的比较”应该是该表格的标题，“实现方式”、“代码量”、“搜索引擎友好”、“特殊终端兼容”应该是几个比较特殊的单元格，“table 布局”、“乱用标签的 CSS 布局”、“标签语义良好的 CSS 布局”也是几个特殊的单元格，但方案一的 HTML 从语义上看不到这样的描述。

我们将它改进一下，如方案二所示。

**方案二** 实现方法如代码清单 3-16 所示。

代码清单 3-16 表格模块实现方案二

---

```
<table border="1">
    <caption>几种页面实现的比较</caption>
    <thead>
        <tr><th>实现方式</th><th>代码量</th><th>搜索引擎友好</th><th>
特殊终端兼容</th></tr>
    </thead>
    <tbody>
        <tr><th>table 布局 </th><td> 多 </td><td> 差 </td><td> 一
般</td></tr>
        <tr><th>乱用标签的 CSS 布局</th><td>少</td><td>一般</td><td>差
</td></tr>
        <tr><th>标签语义良好的 CSS 布局</th><td>少</td><td>好</td><td>
好</td></tr>
    </tbody>
</table>
```

---

我们来看看方案二的语义，如代码清单 3-17 所示。

代码清单 3-17 表格模块实现方案二的语义

---

```
<表格 border="1">
    <表格标题>几种页面实现的比较</表格标题>
    <表格头部>
        <表格行> <表头>实现方式</表头><表头>代码量</表头><表头>搜索引擎友
好</表头><表头>特殊终端兼容</表头></表格行>
    </表格头部>
    <表格主体>
        <表格行><表头>table 布局</表头><表格单元格>多</表格单元格><表格单
元格>差</表格单元格><表格单元格>一般</表格单元格></表格行>
        <表格行><表头>乱用标签的 CSS 布局</表头><表格单元格>少</表格单元格>
<表格单元格>一般</表格单元格><表格单元格>差</表格单元格></表格行>
        <表格行><表头>标签语义良好的 CSS 布局</表头><表格单元格>少</表格单
元格><表格单元格>好</表格单元格><表格单元格>好</表格单元格></表格行>
    </表格主体>
</table>
```

---

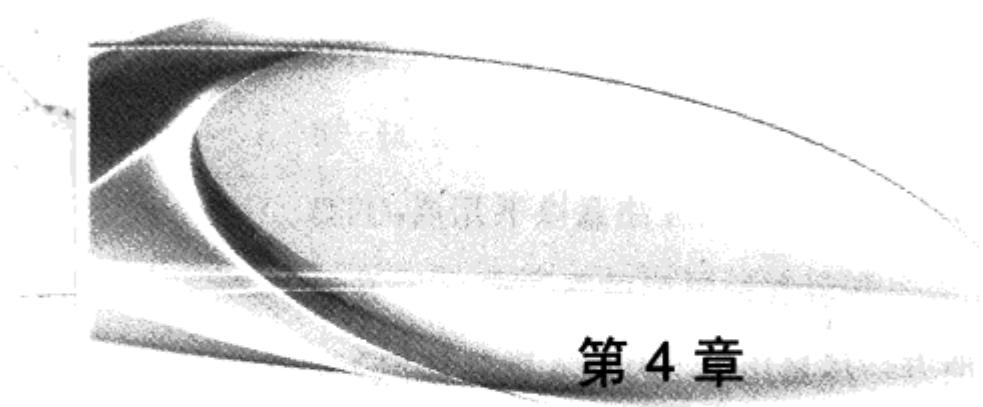
方案二的语义清晰了许多。虽然我们从 table 布局时代开始就已经接触了 table 标签，但对 table 并不一定十分了解。过去，我们在用 table 布局时，常常只使用 table、tr、td 标签。事实上，table 常用的标签还包括 caption、thead、tbody、tfoot 和 th。我们在使用表格的时候，应该注意选用合适的标签，表格标题要用 caption，表头要用 thead 包围，主体部分用 tbody 包围，尾部要用 tfoot 包围，表头和一般单元格要区分开，表

头用 th，一般单元格用 td。

#### 3.4.4 语义化标签应注意的一些其他问题

为了保证网页去样式后的可读性，并且又符合 Web 标准，我们应注意以下几点：

- 尽可能少地使用无语义标签 div 和 span；
- 在语义不明显，既可以用 p 也可以用 div 的地方，尽量用 p，因为 p 默认情况下有上下间距，去样式后的可读性更好，对兼容特殊终端有利；
- 不要使用纯样式标签，例如 b、font 和 u 等，改用 CSS 设置。语义上需要强调的文本可以包在 strong 或 em 标签里，strong 和 em 有“强调”的语意，其中 strong 的默认样式是加粗，而 em 的默认样式是斜体。



## 第4章

# 高质量的 CSS

## 本章内容

- 怪异模式和 DTD
- 如何组织 CSS
- 推荐的 base.css
- 模块化 CSS——在 CSS 中引入面向对象编程思想
- 低权重原则——避免滥用子选择器
- CSS sprite
- CSS 的常见问题

## 4.1 怪异模式和 DTD

为了确保向后兼容，浏览器厂商发明了标准模式和怪异模式这两种方法来解析网页。在标准模式中，浏览器根据规范表现页面；而怪异模式通常模拟老式浏览器（比如 Microsoft IE 4 和 Netscape Navigator 4）的行为以防止老站点无法工作。这两种模式的差异比较大，比较典型的就是 IE 对盒模型的解析：在标准模式中，网页元素的宽度是由 padding、border、width 三者的宽度相加决定的；而在怪异模式中，width 本身就包括了 padding 和 border 的宽度。此外，标准模式下块级元素的经典居中方法——设定 width，然后 margin-left:auto, margin-right:auto——在怪异模式下也无法正常工作。

同样的代码，在怪异模式和标准模式下的表现很可能相差甚远。因为发明怪异模式的目的就是为了兼容老式浏览器下的代码，它的很多解析方式是不符合标准的。所以，一般情况下，我们应该避免触发怪异模式，应选用标准模式。

怪异模式是如何被触发的呢？与 DTD 有关。DTD 全称 Document Type Definition，即文档类型定义。DTD 是一种保证 HTML 文档格式正确的有效方法，可以通过比较 HTML 文档和 DTD 文件来看文档是否符合规范，以及元素和标签使用是否正确。一个 DTD 文档包含元素的定义规则、元素间关系的定义规则、元素可使用的属性、可使用的实体或符号规则。

在网页中最常用的 DTD 类型包括代码清单 4-1 中所示的 4 种。

代码清单 4-1 HTML 中常见的 4 种 DTD 类型

用于 HTML 4.01 的严格型

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  
"http://www.w3.org/TR/html4/strict.dtd">
```

用于 HTML 4.01 的过渡型

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  
"http://www.w3.org/TR/html4/loose.dtd">
```

用于 XHTML 1.0 的严格型

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

用于 XHTML 1.0 的过渡型

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

按照 W3C 的标准，我们需要在 HTML 的最开始声明文件的 DTD 类型。如果漏写 DTD 声明，Firefox 仍然会按照标准模式来解析网页，但在 IE 中（包括 IE 6、IE 7、IE 8）就会触发怪异模式。在 table 布局时代，盒模型和 CSS 我们都接触得比较少，所以 DTD 声明并不是很重要，被我们忽视了。到了 CSS 布局时代，DTD 的声明就变得非常重要了。为了避免怪异模式给我们带来不必要的麻烦，我们要养成书写 DTD 声明的好习惯。

## 4.2 如何组织 CSS

我们应用 CSS 的能力应该分成两部分：一部分是 CSS 的 API，重点是如何用 CSS 控制页面内元素的样式；另一部分是 CSS 框架，重点是如何对 CSS 进行组织。前者属于基础部分，这部分的能力是用“对”和“错”来评判的，比如说，要把文字设置为红色，只能用 `color:red;` 这样的写法是对的，其他任何写法都是错的，不存在“好”和“坏”的区别，只有“对”和“错”。CSS 的 API 并不多，掌握到“会用”的程度并不难，但如果要用得“好”，我们需要在前者的基础上更进一步，研究如何组织 CSS。如何组织 CSS 是一个见仁见智的问题，不是用“对”和“错”来评判的，我们更可能会用“好”、“比较好”、“很烂”、“非常棒”这样的字眼来评判。

正因为见仁见智，所以如何组织 CSS 可以有多种角度，例如按功能划分：将控制字体的 CSS 集中在 `font.css` 文件里，将控制颜色的 CSS 集中在 `color.css` 文件里，将控制布局的 CSS 放在 `layout.css` 文件里；或者按区块划分：将头部的 CSS 放在 `head.css` 里，底部放在 `foot.css` 里，侧边栏放在 `sidebar.css` 里，主体放在 `main.css` 里。不同角度的组织方法都有自己的道理，也有自己的优点和缺点。

这里推荐笔者最喜欢的一种组织 CSS 的方法：`base.css + common.css + page.css`。将网站内的所有样式，按照职能分成三大类：`base`、`common` 和 `page`。在一般情况下，任何一个网页的最终表现都是由这三者共同完成的。这三者不是并列结构，而是层叠结构，如图 4-1 所示。

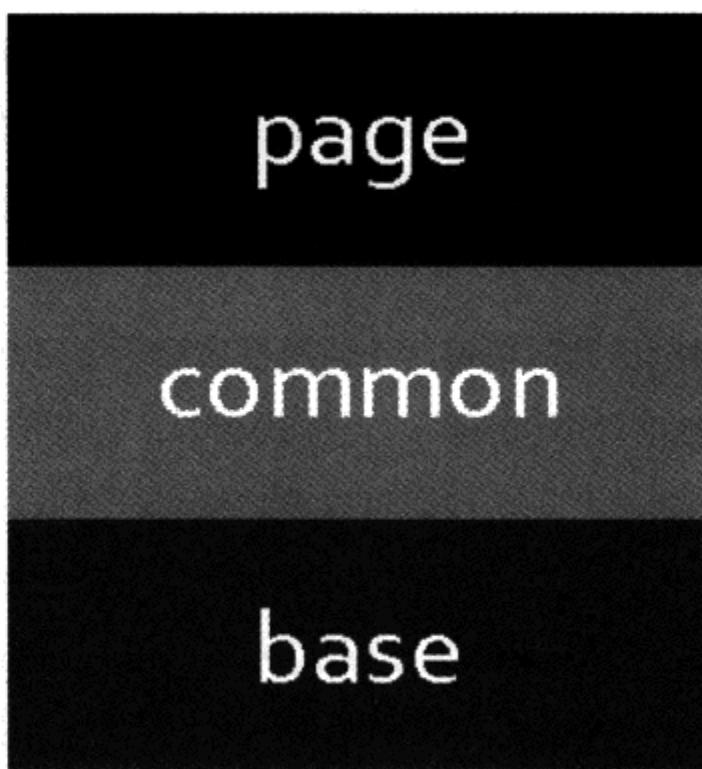


图 4-1 base.css+common.css+page.css 框架的层叠关系

### 1. base 层

这一层位于三者的最底层，提供 CSS reset 功能和粒度最小的通用类——原子类。这一层会被所有页面引用，是页面样式所需依赖的最底层。这一层与具体 UI 无关，无论何种风格的设计都可以引用它，所以 base 层要力求精简和通用。如果将用 CSS 控制页面样式比喻为建房子，这一层的核心职能是为房子打好地基（CSS reset），并将建房用的砖块（原子类）准备充足。因为几乎所有的房子都要打地基，也都需要砖块，所以 base 层具有高度可移植性，不同设计风格的网站可以使用同一个 base 层。因为这一层的内容很少，所以可以简单地放在一个文件里，例如 base.css。

base 层相对稳定，基本上不需要维护。

### 2. common 层

这一层位于中间，提供组件级的 CSS 类。提到组件，就不得不提“模块化”。“模块化”可以从样式和行为两个层面来考虑，与 common 层相关的是样式的模块化。我们可以将页面内的元素拆分成一小块一小块功能和样式相对独立的小“模块”，这些“模块”有些是很少重复的，有些是会大量重复的，我们可以将大量重复的“模块”视为一

个组件。我们从页面里尽可能多地将组件提取出来，放在 `common` 层里。`common` 层就相当于 MVC 模式中的 M (Model，模型)。为了保证重用性和灵活性，M 需要尽可能将内部实现封装，对可能会经常变化的部分提供灵活的接口。关于 `common` 层的技巧，详见 4.4 节。

`common` 层就像建房时用到的门窗，不同风格的房子会用到不同样式的门窗，各个房间用到的门窗的数量和位置可以不同，但样式相同。门窗有自己的小元件，比如玻璃、门闩、门框、门板、钥匙孔等。门窗可以整体移动、增减，但门窗本身的构造是相对稳定的。不同风格的房子就好比不同风格的网站，房子选用的门窗就好比这个网站选用的 UI 组件。门窗最好与整个房子的风格保持一致，同样，网站最好让 UI 组件的风格保持相同。UI 组件是网站中的单位，在网站内部可以高度重用，但不同的网站可能会用不同的 UI 组件。

所以，`common` 层是网站级的，不同的网站有不同的 `common` 层，同一个网站只有一个 `common` 层。`common` 层是放在一个 `common.css` 文件里，还是按照功能划分放在诸如 `common_form.css`、`common_imagelist.css` 的多个文件里，需要根据网站规模来决定。在团队合作中，`common` 层最好由一个人负责，统一管理。

### 3. `page` 层

网站中高度重用的模块，我们把它们视为组件，放在 `common` 层；非高度重用的模块，可以把它们放在 `page` 层。`page` 层位于最高层，提供页面级的样式。同样以建房子为比喻，`page` 层就好比是房间内的装饰画，不同的房间张贴的装饰画各不相同。它不像砖块，所有房子都相同；也不像门窗，同一房子里都相同。它对重用性没有要求，可根据各个房间布置的需要任意张贴。

`page` 层是页面级的，每个页面都可能会有自己的 `page` 层 CSS。`page` 层的文件可以用`<style type="text/CSS">`标签内置于页面中，但这么做没有将样式彻底从 HTML 文件中分离出来。也可以根据页面写在诸如 `page1.css`、`page2.css`、`page3.css` 的文件里，这样做可以将样式很好地从 HTML 中分离出来，但可能会产生大量 CSS 文件，有些 CSS 文件可能非常小，带来维护上的麻烦。如果网站规模不会过于庞大，笔者建议将网站内所

有 page 层的代码放在一个 page.css 文件里，根据页面配上注释，分块书写，便于维护，如代码清单 4-2 所示。

代码清单 4-2 page.css 的注释

```
/*首页*/
.test{}
.test2{}

/*关于我们*/
.test3{}
.test4{}

/*联系我们*/
.test5{}
.test6{}
```

这么做可能会带来些冗余，比如，“首页”的 CSS 文件里带有“关于我们”、“联系我们”页的 page 层 CSS 文件，而这些对首页的样式毫无影响。如 2.5 节所述，对于文件过于分散和集中的问题并没有完美的解决办法，我们需要根据实际情况做些适当的折中。比起让 page 层的 CSS 文件过于繁多和零散，把它们集中在一个文件中更便于维护，且便于浏览器缓存，浏览网站时只有首页的下载时间较长，浏览其他页面时反而较快。当然，page.css 还是应当越精简越好，能用 base 层和 common 层的 CSS 解决的，就尽量不要用到 page 层。

base 层基本上不需要维护，common 层修改的幅度不会很大，通常只由一个人负责维护，但到了 page 层，代码可能由多人开发，如何避免冲突是个需要注意的问题。通常我们通过命名规则来避免这种冲突。具体方法见 4.4.2 节。

### 4.3 推荐的 base.css

base 层是高度重用的基础层，它提供的通用类是否够用直接关系到网站的开发效率和 CSS 文件的总大小，对于一个网站来说，base 层设计是否良好非常重要。因为它具有无视美工设计，适用于任何网站的特点，所以不同于 common 层和 page 层，可脱离具体网站来讲解。下面是一个笔者推荐使用的 base.css 代码，如代码清单 4-3 所示。

## 代码清单 4-3 base.css 文件

```
/*CSS reset*/
body,div,dl,dt,dd,ul,ol,li,h1,h2,h3,h4,h5,h6,pre,form,fieldset,input,
textarea,p,blockquote,th,td {margin:0;padding:0;}
table {border-collapse:collapse; border-spacing:0;}
fieldset,img {border:0}
address,caption,cite,code,dfn,em,strong,th,var {font-style:normal;font-
weight:normal}
ol,ul {list-style:none}
caption,th {text-align:left}
h1,h2,h3,h4,h5,h6 {font-size:100%;font-weight:normal}
q:before,q:after {content:''}
abbr,acronym { border:0}

/*文字排版*/
.f12{font-size:12px}
.f13{font-size:13px}
.f14{font-size:14px}
.f16{font-size:16px}
.f20{font-size:20px}
.fb{font-weight:bold}
.fn{font-weight: normal}
.t2{text-indent:2em}
.lh150{line-height:150%}
.lh180{line-height:180%}
.lh200{line-height:200%}
.uln{text-decoration:underline;}
.no_unl{text-decoration:none;}

/*定位*/
.tl{text-align:left}
.tc{text-align:center}
.tr{text-align:right}
.bc{margin-left:auto; margin-right:auto;}
.fl{float:left; display:inline}
.fr{float:right; display:inline}
.cb{clear:both}
.cl{clear:left}
.cr{clear:right}
.clearfix:after{content:"."; display:block; height:0; clear:both; visibility:
hidden}.clearfix{display:inline-block}* html
.clearfix{height:1%}.Clearfix {display:block}
.vm{vertical-align:middle}
.pr{position:relative}
.pa{position:absolute}
.abs-right{position: absolute; right:0}
.zoom{zoom:1}
.hidden{visibility:hidden}
.none{display:none}

/*长度高度*/
.w10{width:10px}
```

```
.w20{width:20px}
.w30{width:30px}
.w40{width:40px}
.w50{width:50px}
.w60{width:60px}
.w70{width:70px}
.w80{width:80px}
.w90{width:90px}
.w100{width:100px}
.w200{width:200px}
.w250{width:250px}
.w300{width:300px}
.w400{width:400px}
.w500{width:500px}
.w600{width:600px}
.w700{width:700px}
.w800{width:800px}
.w{width:100%}
.h50{height:50px}
.h80{height:80px}
.h100{height:100px}
.h200{height:200px}
.h{height:100%}

/*边距*/
.m10{margin:10px}
.m15{margin:15px}
.m30{margin:30px}
.mt5{margin-top:5px}
.mt10{margin-top:10px}
.mt15{margin-top:15px}
.mt20{margin-top:20px}
.mt30{margin-top:30px}
.mt50{margin-top:50px}
.mt100{margin-top:100px}
.mb5{margin-bottom:5px}
.mb10{margin-bottom:10px}
.mb15{margin-bottom:15px}
.mb20{margin-bottom:20px}
.mb30{margin-bottom:30px}
.mb50{margin-bottom:50px}
.mb100{margin-bottom:100px}
.ml5{margin-left:5px}
.ml10{margin-left:10px}
.ml15{margin-left:15px}
.ml20{margin-left:20px}
.ml30{margin-left:30px}
.ml50{margin-left:50px}
.ml100{margin-left:100px}
.mr5{margin-right:5px}
.mr10{margin-right:10px}
.mr15{margin-right:15px}
.mr20{margin-right:20px}
.mr30{margin-right:30px}
```

```
.mr50{margin-right:50px}
.mr100{margin-right:100px}
.p10{padding:10px;}
.p15{padding:15px;}
.p30{padding:30px;}
.pt5{padding-top:5px}
.pt10{padding-top:10px}
.pt15{padding-top:15px}
.pt20{padding-top:20px}
.pt30{padding-top:30px}
.pt50{padding-top:50px}
.pb5{padding-bottom:5px}
.pb10{padding-bottom:10px}
.pb15{padding-bottom:15px}
.pb20{padding-bottom:20px}
.pb30{padding-bottom:30px}
.pb50{padding-bottom:50px}
.pb100{padding-bottom:100px}
.pl5{padding-left:5px}
.pl10{padding-left:10px}
.pl15{padding-left:15px}
.pl20{padding-left:20px}
.pl30{padding-left:30px}
.pl50{padding-left:50px}
.pl100{padding-left:100px}
.pr5{padding-right:5px}
.pr10{padding-right:10px}
.pr15{padding-right:15px}
.pr20{padding-right:20px}
.pr30{padding-right:30px}
.pr50{padding-right:50px}
.pr100{padding-right:100px}
```

这个 base.css 文件可以分为两大部分：CSS reset 和通用原子类。

什么是 CSS reset 呢？HTML 标签在浏览里有默认的样式，例如 p 标签有上下边距，strong 标签有字体加粗样式，em 标签有字体倾斜样式。不同浏览器的默认样式之间也会有差别，例如 ul 默认带有缩进的样式，在 IE 下，它的缩进是通过 margin 实现的，而在 Firefox 下，它的缩进却是由 padding 实现的。在切换页面的时候，浏览器的默认样式往往会展开给我们带来麻烦，影响开发效率。现在很流行的解决办法是一开始就将浏览器的默认样式全部去掉，更确切地说，应该通过重新定义标签的样式，“覆盖”掉浏览器提供的默认样式，这就是 CSS reset。

关于 CSS reset，有一种写法相信大家都很熟悉，即 “\* {margin:0;padding:0;}”，因为浏览器默认的边距是最影响 CSS 布局的，所以很多人都会在 CSS 文件的最开始加入

这么一句代码。这应该是最早的 CSS reset 了。但一方面它并不完善，更好的写法还应该去掉 ol 和 ul 的列表样式，th 的加粗，caption 的居中，h 系列标签的加粗、字号等；另一方面它本身也存在性能问题，“\*”是个通配符，表示所有标签，其中包括大量生僻标签和为向前兼容而留下来的淘汰标签。

更好的写法是将常用的标签显式地罗列出来，避免使用“\*”，例如“body, div, dl, dt, dd, ul, ol, li, h1, h2, h3, h4, h5, h6, pre, form, fieldset, input, textarea, p, blockquote, th, td{margin:0;padding:0}”。有些工程师虽然使用了这种方法，但他们除了 margin 和 padding，还喜欢将 color、font-size 之类的样式写进去，例如“body, div, dl, dt, dd, ul, ol, li, h1, h2, h3, h4, h5, h6, pre, form, fieldset, input, textarea, p, blockquote, th, td{margin:0;padding:0;color:#000;font-size:12px; }”，这种做法非常不好，因为 CSS 的很多样式是具有继承性的，但继承性的权重非常低，比标签选择符的权重更低（选择符权重的问题详见 4.4.4 节），这种写法会破坏 CSS 的继承性，设置样式时会额外增加很多代码。

代码清单 4-3 所示的 base.css 中，CSS reset 部分来自于 YUI（雅虎的前端框架，详见：<http://developer.yahoo.com/yui/>），这是段很成熟的代码，推荐大家都使用它来进行 CSS reset。

通用原子类是一系列常用的基本类，包括：文字、定位、长宽和边距。除了极少数特殊类，绝大部分通用原子类都只包含一句 CSS，例如“.f12{font-size:12px}”。通用原子类具有两个特点：“通用性”和“原子性”。“通用性”表现在它们是网站最常用的类，任何页面都可以随意使用它们，“原子性”表现在它们是最基础的样式，一个类只设置一个样式，不可再分。由于它们具有通用性，所以在保证命名有语义的前提下，命名应尽量简短，方便调用。例如“fontSize12”可以写成“f12”，“paddingTop20”可以写成“pt20”，这样既方便记忆又方便调用。关于 CSS 命名的更多问题，详见 4.4.1 节。

通用原子类里有几个类比较特殊，需要特别说明一下。

- “.fl”类和“.fr”类。这两个类，除了设置 float:left 和 float:right 之外，还设置了 display:inline。熟悉 CSS Bug 的工程师应该一眼就能认出其作用——解决 IE 6 的双外边距 Bug。在 IE 6 下，如果对元素设置了浮动，同时又设置了 margin-left

或者 margin-right, margin 值会加倍。例如，设置 margin-left:10px 在 IE 6 下会显示为 margin-left:20px。解决这个 Bug 的办法就是设置 display:inline。我们在设计通用原子类时，将 display:inline 直接添加到 “.fl” 和 “.fr” 类里，可以避免挂浮动类时引入 Bug。

- “.bc” 类。这个类的全称是 “.blockCenter”，作用为使块级元素居中。直接使用它是不足以使块级元素居中的，我们还需设定宽度。通常情况下，我们可以把它和 “.w100”、“.w200” 类同时使用，例如：“<div class="bc w200"></div>”。
- “.clearfix” 类。这个类用于在父容器直接清除子元素浮动。通常情况下，为了让浮动元素的父容器能够根据浮动元素的高度而自适应高度，有三种做法：
  - ◆ 让父容器同时浮动起来，例如：“<div class="fl"><div class="fl"> </div> </div>”；
  - ◆ 让浮动元素后面紧跟一个用于清除浮动的空标签，例如“<div><div class="fl"></div><div class="cb"></div></div>”；
  - ◆ 给父容器挂一个特殊 class，直接从父容器清除浮动元素的浮动，例如“<div class="clearfix"><div class="fl"></div></div>”。

第一种方法会让父容器也浮动起来，影响父元素后面的元素的布局，有副作用。第二种方法增加了一个空标签，破坏了语义化。第三种方法没有任何副作用，推荐使用。

- “.zoom” 类。这个类有些特殊，它设置的样式是 zoom:1，可能有些工程师对这个样式比较陌生。它并不是 CSS 标准中的标准属性，而是 IE 的专有属性。提到 zoom 属性，就不得不提到 IE 的 hasLayout 问题。在 IE 7 发布之前，有些人推荐设置 height:1% 来触发 hasLayout，因为 IE 6 会将 height 按照 min-height 来解析，所以 height:1% 不会引入副作用，但 IE 7 发布以后，这种方式就不适用了。更好的触发 hasLayout 的方式是设置 zoom:1，不用担心引入任何副作用。

除了以上这几个类，读者可能还会注意到用于设置边距的其他的类，例如 “.mt5”、“.mt10”、“.mt20”。这些类虽然繁杂，但依笔者的经验来看，它们非常有用，设置大量这样的类有助于减少 page 层的代码量，也为 CSS 的模块化提供了帮助，详见 4.3.3 节。

## 4.4 模块化 CSS——在 CSS 中引入面向对象编程思想

### 4.4.1 如何划分模块——单一职责

模块化可以让代码高度重用，显著提高开发效率。关于模块化，比较成熟的是编程领域的“类”。在面向对象编程方式中，“类”是个非常核心的概念，可以说面向对象思想的基础就是“类”。关于模块化，“类”有很多成熟的技巧，例如封装、多用组合少用继承原则等。CSS 的模块化是个有趣的话题，它是个新的领域，很多人都有自己的见解。下面笔者谈谈自己的理解——借鉴编程中的“类”，将面向对象的编程思想引入到 CSS 的模块化里。

从视觉上进行划分，样式和功能相对独立且稳定的一部分就可以视为模块。举个例子说明，对于图 4-2 所示的设计图，我们该如何划分模块呢？



图 4-2 模块化示意图 1

很明显，除了第三列，前两列的上下两部分从视觉上来看结构相同。因此可以将设计图按图 4-3 所示的四种模块来编写代码。每个模块都要完整包含自己区域的内容，相

同类型的模块可以重用。



图 4-3 模块化示意图 2

但仔细观察，会发现模块 1 和模块 2 有相似的部分，模块 3 和模块 4 也有相似的部分，这些相似的部分分别在两个模块里都写了一遍。一方面，增加了冗余的代码量，另一方面如果需要修改相似部分的代码，两个模块都需要修改，增加了修改的成本。

我们可以将这些相似的部分提取出来，再进一步拆分成更小的模块。我们可以将设计图拆成 6 个模块，如图 4-4 所示。如此一来，每个模块都相对独立，和其他模块没有重复的地方，模块的重用率提高了。

图 4-5 中框出的部分如果需要修改，按 4 种模块拆分的方式就需要在模块 1 和模块 2 中分别修改，而按 6 种模块拆分的话，只需要修改模块 4 就可以了。可见拆分成 6 个模块可以提高可维护性，方便修改。



图 4-4 模块化示意图 3



图 4-5 模块化示意图 4

至此，我们可以归纳出拆分模块的第一个技巧：模块与模块之间尽量不要包含相同的部分，如果有相同部分，应将它们提取出来，拆分成一个独立的模块。

按 6 种模块拆分的方式已经足够好了吗？虽然模块与模块之间已经不再有相同部分了，每个模块都相对独立，但模块 1 和模块 2 包含的结构比较复杂，如果需要添加如下的模块会怎么样呢？

图 4-6 中的设计是不是和图 4-2 很像呢？我们能不能直接使用图 4-4 中拆分的模块来拆分图 4-6 呢？答案是不行！虽然图 4-6 中的设计与图 4-4 中的模块 1 和模块 2 很像，但并不一样。图 4-4 中的模块 1 和模块 2 除了标题和边框之外，还包含其他内容。



图 4-6 模块化示意图 5

将图 4-4 中的模块 1 和模块 2 进一步拆分，得到如图 4-7 所示的 8 个模块，其中的模块 1 和模块 2 就可以匹配图 4-6 中的设计了，如图 4-8 所示。



图 4-7 模块化示意图 6



图 4-8 模块化示意图 7

按照 6 种模块来拆分，其中模块 1 和模块 2 会因为结构复杂，而难以在相似的设计中重用。按照 8 种模块来拆分，模块的结构更加简单，能重用在相似的设计中。我们在拆分模块时，应将模块拆得尽可能简单，以提高弹性。模块功能越简单其重用性越高，但数量也会相应增加，增加了维护难度。所以，拆分模块时应该在“数量少”和“结构简单”之间取一个最合适的平衡点。

这就是拆分模块的第二个技巧：模块应在保证数量尽可能少的原则下，做到尽可能简单，以提高重用性。

在面向对象编程中也有相似的思想，设计类的时候，为了保证重用性需遵守“单一职责”原则，一般情况下，类的功能应该相对简单稳定，一个类只具有一个职责。一个功能超复杂的拥有多个职责的大类，其重用性远不如一群职责单一的简单小类。

#### 4.4.2 CSS 的命名——命名空间的概念

CSS 的命名应该注意哪些问题呢？我们以图 4-9 为例，进行详细解析。

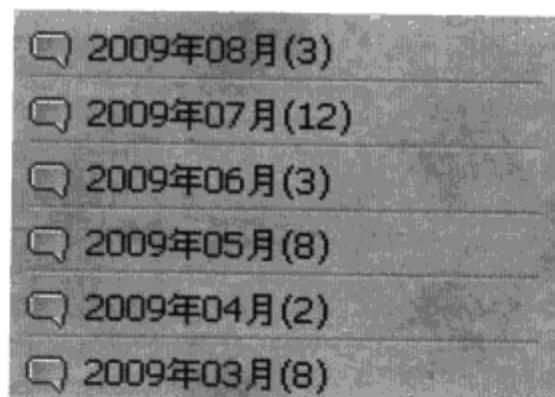


图 4-9 CSS 命名示例图

首先，CSS 的命名推荐使用英语，不要使用汉语拼音。我们可以根据内容来选用合适的英文单词命名 CSS。比如头部用 head，底部用 foot，主体部分用 main，导航用 nav，菜单用 menu 等，这些不是硬性的要求，但为了方便阅读和理解，提高可维护性，我们推荐使用这样的命名方式。很明显，图 4-9 是一个关于时间的无序列表，我们可以用 timelist 来命名它。如代码清单 4-4 所示。

## 代码清单 4-4 CSS 命名方案一

---

```

<style type="text/CSS">
.timeList{xxxxxxx}
</style>
<ul class="timelist">
<li>2009年08月(3)</li>
<li>2009年07月(12)</li>
<li>2009年06月(3)</li>
<li>2009年05月(8)</li>
<li>2009年04月(2)</li>
<li>2009年03月(8)</li>
</ul>

```

---

`timelist` 用到了两个单词： `time` 和 `list`，但它们连在一起，很难一眼认出来，可读性很差。如果命名需要用到两个或两个以上单词，常见的做法有两种：骆驼命名法和划线命名法。骆驼命名法是从第二个单词开始，将每个单词的首字母大写，例如 `dropMenu`、`subNavMenu`。而划线命名法是将每个单词用中划线“-”或下划线“\_”分隔，例如 `drop-menu`、`sub_nav_menu`。使用骆驼命名法和划线命名法都可以清楚地将单词划分开来，提高命名的可读性。但笔者认为单独使用骆驼命名法或划线命名法并不是最好的实践方式，推荐将这两种方法组合使用。

代码清单 4-4 中每个 `li` 都有一条下划线，但最后一个没有，因为 IE 6 不支持`:last` 伪类，所以我们不能简单地使用`.timeList li{border-bottom:×××` 和 `.timeList li:last{border-bottom:none;}` 来实现样式。我们只能为最后一个 `li` 挂个特殊的 `class` 来实现想要的效果，如代码清单 4-5 所示。

## 代码清单 4-5 CSS 命名方案二

---

```

<style type="text/CSS">
.timeList{xxxxxxx}
.timeList li{border-bottom:xxxx;}
.timeList .last{border-bottom:none;}
</style>
<ul class="timeList">
<li>2009年08月(3)</li>
<li>2009年07月(12)</li>
<li>2009年06月(3)</li>
<li>2009年05月(8)</li>
<li>2009年04月(2)</li>
<li class="last">2009年03月(8)</li>
</ul>

```

---

为最后一个 li 标签挂上名为 last 的 class，可以实现我们想要的效果，但这样的命名还不够好。这涉及一个问题——滥用子选择符。很多工程师喜欢使用子选择符，但滥用子选择符容易留下冲突隐患，我们在编写代码时应该时刻保持“团队合作”的意识，将冲突隐患降到最低，提高代码的可维护性。last 是个过于简单且常用的命名，如果工程师 A 过于依赖子选择符，可能使 CSS 代码里出现大量类似于“.timeList .last”、“.nameList .last”、“.ageList .last”这样的选择符，如果多人合作，工程师 B 可能习惯直接使用“.last”作为选择符，从而和“.timeList .last”等选择符设置的样式层叠，产生意外之外的影响。除了“.last”，常见的容易产生冲突的命名还有“.first”、“.item”等。

为了将风险降至最低，不推荐轻易使用子选择符，可以改用如代码清单 4-6 所示的命名。

#### 代码清单 4-6 CSS 命名方案三

---

```

<style type="text/CSS">
.timeList{xxxxxxx}
.timeList li{border-bottom:xxxx;}
.timeListLastItem{border-bottom:none;}
</style>
<ul class="timeList">
    <li>2009 年 08 月 (3)</li>
    <li>2009 年 07 月 (12)</li>
    <li>2009 年 06 月 (3)</li>
    <li>2009 年 05 月 (8)</li>
    <li>2009 年 04 月 (2)</li>
    <li class="timeListLastItem">2009 年 03 月 (8)</li>
</ul>

```

---

我们使用“.timeListLastItem”代替“.timeList .last”来作为选择符，就可以降低冲突隐患了。但这样的命名仍然不够好——将整个 ul 视为一个模块，在结构上“.timeListLastItem”是从属于“.timeList”的，但命名上看不出明显的从属关系。虽然 CSS 没有真正意义上的“封装”功能，但如果 CSS 命名能够表现出从属关系，就可以相对地让模块的封装性更好。如前所述，我们可以结合骆驼命名法和划线命名法来进行命名，其中骆驼命名法用于区别不同单词，划线用于表明从属关系。“.timeListLastItem”可以改进为“.timeList-lastItem”，如此一来，进一步提高了 CSS 命名的可读性，不仅能够从命名中清楚看出各个单词，还能了解到从属关系。很明显，“.timeList-lastItem”是

“.timeList”的“.lastItem”。

既然划线用于表明从属关系，那么不推荐使用诸如“.timeListLastItem”、“.time-list-first-item”这些命名方式。

在 4.2 节中我们讲过 CSS 可以分为 `base`、`common` 和 `page` 三层。其中 `base` 层和 `common` 层是公共的，由于统一加载到所有页面里，一般只会由一个人来负责，不会出现冲突问题。而 `page` 层是页面级的，可能会由多人合作完成，有可能存在冲突隐患。我们在命名 CSS 时，首先要判断它是位于什么层的，如果模块多次反复出现，那么应该将它归为 `common` 层，不用考虑冲突问题，而如果出现的次数少，那么应该将它归为 `page` 层，需考虑如何避免冲突。

假设图 4-9 所示的模块位于 `page` 层，这个页面有多个模块，因为某种原因，图 4-9 所示的模块由工程师 A 制作，而另一个模块由工程师 B 制作。工程师 B 有可能为自己负责的模块取了同样的名字 `timeList`，如代码清单 4-7 所示。

代码清单 4-7 CSS 命名冲突

---

```

<style type="text/CSS">
/*made by 工程师 A*/
.timeList{xxxxxx}
.timeList li{border-bottom:xxxx;}
.timeList-lastItem{border-bottom:none;}

...
/*made by 工程师 B*/
.timeList{}
.timeList li{border-bottom:xxxx;}

</style>
<!--made by 工程师 A-->
<ul class="timeList">
<li>2009 年 08 月 (3)</li>
<li>2009 年 07 月 (12)</li>
<li>2009 年 06 月 (3)</li>
<li>2009 年 05 月 (8)</li>
<li>2009 年 04 月 (2)</li>
<li class="timeList-lastItem">2009 年 03 月 (8)</li>
</ul>

...
<!--made by 工程师 B-->
<ol class="timeList">
```

```
<li>2009-08-07</li>
<li>2009-08-06 </li>
<li>2009-08-05</li>
</ol>
```

工程师 B 因为没有注意到工程师 A 写的代码，给自己的 HTML 标签挂了同名的 class，在无意中和工程师 A 编写的代码产生了冲突。如何避免这样的冲突呢？可以通过给命名加前缀的方式解决这个问题。每个加入到团队中的工程师，需分配一个唯一的标识符，这个标识符加上划线作为自己所负责的 page 层 CSS 命名时的前缀。比如说，可以使用姓名首字母缩写作为标识符，工程师 A 叫做“阿当”，其姓名拼音为“a dang”，取首字母缩写，其标识符为 ad，工程师 B 叫做“张霞”，其姓名拼音为“zhang xia”，取首字母缩写，其标识符为 zx。重写代码清单 4-7，如代码清单 4-8 所示。

代码清单 4-8 CSS 命名加前缀

```
<style type="text/CSS">
/*made by 工程师 A*/
.ad-timeList{xxxxxxxx}
.ad-timeList li{border-bottom:xxxx;}
.ad-timeList-lastItem{border-bottom:none;}

...
/*made by 工程师 B*/
.zx-timeList{}
.zx-timeList li{border-bottom:xxxx;}

</style>
<!--made by 工程师 A--&gt;
&lt;ul class="ad-timeList"&gt;
&lt;li&gt;2009 年 08 月 (3)&lt;/li&gt;
&lt;li&gt;2009 年 07 月 (12)&lt;/li&gt;
&lt;li&gt;2009 年 06 月 (3)&lt;/li&gt;
&lt;li&gt;2009 年 05 月 (8)&lt;/li&gt;
&lt;li&gt;2009 年 04 月 (2)&lt;/li&gt;
&lt;li class="ad-timeList-lastItem"&gt;2009 年 03 月 (8)&lt;/li&gt;
&lt;/ul&gt;

...
<!--made by 工程师 B--&gt;
&lt;ol class="zx-timeList"&gt;
&lt;li&gt;2009-08-07&lt;/li&gt;
&lt;li&gt;2009-08-06 &lt;/li&gt;
&lt;li&gt;2009-08-05&lt;/li&gt;
&lt;/ol&gt;</pre>
```

加了唯一前缀之后，工程师 A 和工程师 B 都可以专心于自己负责的代码，不用担心冲突的问题了。

但如此一来，存在一个问题——CSS 命名过长。在 base 层，因为 CSS 只包含原子类，并且是所有团队成员共享的一层，所以功能单一，不存在划线分隔和命名前缀的问题，这层的 CSS 命名大多非常简短。在 common 层，由于有了组件概念，根据组件的复杂程度，CSS 命名可能会比较长，例如“infoList-firstItem-img”、“nav-item-select”等。到了 page 层，因为需要加前缀，命名可能会更长，例如“ad-dropMenu-lastItem-img”、“zx-subNav-item-select”。命名长可以带来很好的可读性，可以避免多人合作的冲突，但也会增加文件大小，有些工程师可能会因为这点而反对使用过长的命名。这的确是个无法兼顾的问题，但权衡两者，过长命名影响的只是纯文件，比起图片和 Flash 等资源，文件大小并不会很大，它带来的坏处在可接受范围内，而它带来的好处却是非常明显的。

面向对象编程里有“公共”（public）和“私有”（private）的概念，它是用于“封装”的强有力武器。通常，我们将一个类看成一个黑盒，公共属性和公共方法是这个黑盒与外界交流的接口，而所有私有属性和私有方法全部封装在黑盒内部。一个优秀的面向对象设计会控制类提供尽可能少的接口，除了必要的属性和方法会设计成公共的，其他属性和方法全部会设计成私有的。由于外界能与类交流的只有公共的部分，所以无论如何修改类的私有部分，只要保证公共部分不产生变化，就不会造成对其他类的影响，类的修改就是安全的。类的公共部分越少，类就越容易维护，所以我们应尽量将属性和方法设计成私有的。

可以将同样的思想借鉴到 CSS 领域。用划线作为从属关系分隔符，其实是将模块视为了类，从属于模块的元素被视为了类的属性，通过命名上的直观，得到一种“封装”。类似于“.last”这样的命名，相当于是公共变量，如果对其修改，很容易产生全局范围内的影响，所以并不推荐使用；而“.timeList-lastItem”相当于“.timeList”范围内的私有变量，“.timeList”起到了命名空间的作用，对其修改只会作用于“.timeList”组件内部，而不会产生全局的影响。从加前缀的角度看，不加前缀的 base 层和 common 层就像是公共属性，适用于全局（所有页面）范围内，而加前缀的 page 层就像是私有属

性，只适用于特定页面。

比起代码清单 4-9 所示的命名方式，笔者更倾向于使用代码清单 4-10 所示的命名方式。

代码清单 4-9 不加前缀的命名方式

```
<div="box">
  <div class="hd"></div>
  <div class="bd"></div>
  <div class="ft"></div>
</div>
```

代码清单 4-10 加前缀的命名方式

```
<div="box">
  <div class="box-hd"></div>
  <div class="box-bd"></div>
  <div class="box-ft"></div>
</div>
```

#### 4.4.3 挂多个 class 还是新建 class ——多用组合，少用继承

假设有如图 4-10 所示的模块。

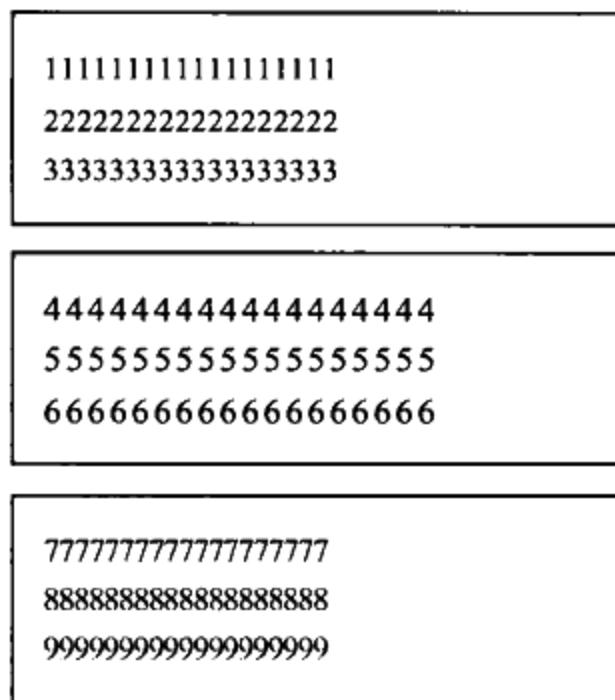


图 4-10 三个简单模块

我们如何设置它的 CSS 呢？方案一如代码清单 4-11 所示。

代码清单 4-11 方案一

---

```

<style type="text/CSS">
    .numberList1{border:1px solid #ccc;padding:10px}
    .numberList1 li{height:20px;line-height:20px;font-size:12px}
    .numberList2{border:1px solid #ccc;padding:10px;}
    .numberList2 li{height:20px;line-height:20px;font-size:16px}
    .numberList3{border:1px solid #ccc;padding:10px;}
    .numberList3 li{height:20px;line-height:20px;font-size:12px;color:red}
</style>

<body>
    <ul class="numberList1">
        <li>111111111111111111</li>
        <li>222222222222222222</li>
        <li>333333333333333333</li>
    </ul>
    <ul class="numberList2">
        <li>444444444444444444</li>
        <li>555555555555555555</li>
        <li>66666666666666666666</li>
    </ul>
    <ul class="numberList3">
        <li>777777777777777777</li>
        <li>888888888888888888</li>
        <li>99999999999999999999</li>
    </ul>
</body>

```

---

方案一可以实现我们想要的效果，但它非常冗余，“.numberList1”、“.numberList2”和“.numberList3”的 CSS 设置相同，“.numberList1 li”、“.numberList2 li”和“.numberList3 li”有部分 CSS 一致。我们对它们进行改进，产生方案二，如代码清单 4-12 所示。

代码清单 4-12 方案二

---

```

<style type="text/CSS">
    .numberList1,.numberList2,.numberList3{border:1px solid #ccc;padding:10px}
    .numberList1 li,.numberList2 li,.numberList3 li{height:20px;line-height:
20px;font-size:12px}
    .numberList2 li{font-size:16px}
    .numberList3 li{color:red}
</style>

<body>
    <ul class="numberList1">
        <li>111111111111111111</li>

```

---

```

        <li>2222222222222222</li>
        <li>3333333333333333</li>
    </ul>
    <ul class="numberList2">
        <li>4444444444444444</li>
        <li>5555555555555555</li>
        <li>6666666666666666</li>
    </ul>
    <ul class="numberList3">
        <li>7777777777777777</li>
        <li>8888888888888888</li>
        <li>9999999999999999</li>
    </ul>
</body>

```

除了方案二，还有另一种思路，方案三如代码清单 4-13 所示。

代码清单 4-13 方案三

```

<style type="text/CSS">
.f12{font-size:12px}
.f16{font-size:16px}
.red{color:red}
.numberList{border:1px solid #ccc;padding:10px;}
.numberList li{height:20px;line-height:20px;}
</style>

<body>
    <ul class="numberList f12">
        <li>1111111111111111</li>
        <li>2222222222222222</li>
        <li>3333333333333333</li>
    </ul>
    <ul class="numberList f16">
        <li>4444444444444444</li>
        <li>5555555555555555</li>
        <li>6666666666666666</li>
    </ul>
    <ul class="numberList f12 red">
        <li>7777777777777777</li>
        <li>8888888888888888</li>
        <li>9999999999999999</li>
    </ul>
</body>

```

方案一将图中的三个模块视为完全不同且彼此独立的三个类，分别命名为 numberList1、numberList2 和 numberList3，并对它们分别设置样式。其缺点是代码冗余。方案二和方案一思路相同，仍将模块视为完全不同且彼此独立的三个类，只是使用 CSS 技巧将三个类相同的部分提取出来，去除了代码的冗余。方案三换了种思路，提取

了更多粒度更小的类，通过类的组合实现设计图的效果。

方案二和方案三看似都是不错的解决方案，其中方案二的优势是调用简单，一个模块只需挂一个类；方案三调用稍麻烦，但也有效控制了冗余，代码精简。看起来似乎方案二和方案三都不错，但如果想实现如图 4-11 所示的效果，又会如何呢？

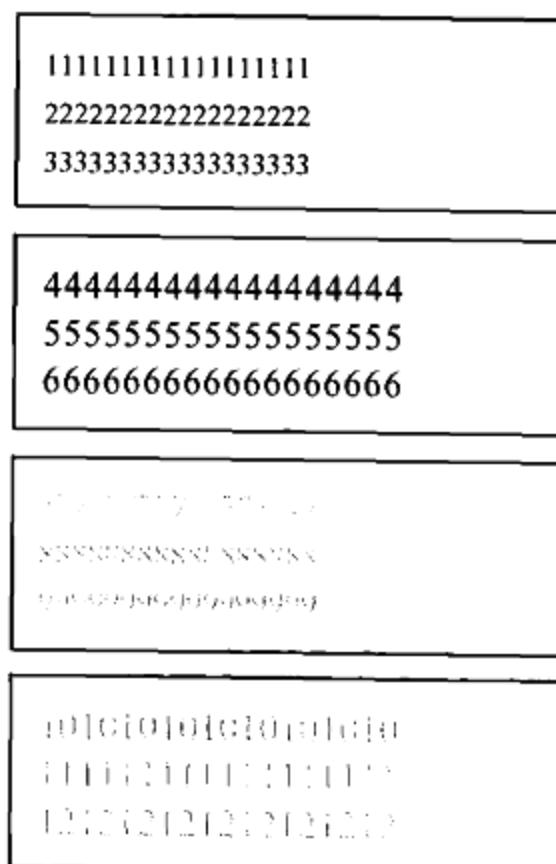


图 4-11 新增了一个模块

按照方案二的思路，代码如代码清单 4-14 所示。

#### 代码清单 4-14 方案二面对扩展

---

```

<style type="text/CSS">
    .numberList1,.numberList2,.numberList3,.numberList4{border:1px solid #ccc;
padding:10px}
    .numberList1 li,.numberList2 li,.numberList3 li,.numberList4 li
{height: 20px;line-height:20px;font-size:12px}
    .numberList2 li,.numberList4 li{font-size:16px}
    .numberList3 li,.numberList4 li{color:red}
</style>

<body>
    <ul class="numberList1">
        <li>1111111111111111</li>

```

```

<li>222222222222222222</li>
<li>333333333333333333</li>
</ul>
<ul class="numberList2">
    <li>444444444444444444</li>
    <li>555555555555555555</li>
    <li>666666666666666666</li>
</ul>
<ul class="numberList3">
    <li>777777777777777777</li>
    <li>888888888888888888</li>
    <li>999999999999999999</li>
</ul>
<ul class="numberList4">
    <li>101010101010101010</li>
    <li>111111111111111111</li>
    <li>121212121212121212</li>
</ul>
</body>

```

按照方案三的思路，代码如代码清单 4-15 所示。

代码清单 4-15 方案三面对扩展

```

<style type="text/CSS">
    .f12{font-size:12px}
    .f16{font-size:16px}
    .red{color:red}
    .numberList{border:1px solid #ccc;padding:10px;}
    .numberList li{height:20px;line-height:20px;}
</style>

<body>
    <ul class="numberList f12">
        <li>1111111111111111</li>
        <li>2222222222222222</li>
        <li>3333333333333333</li>
    </ul>
    <ul class="numberList f16">
        <li>4444444444444444</li>
        <li>5555555555555555</li>
        <li>6666666666666666</li>
    </ul>
    <ul class="numberList f12 red">
        <li>7777777777777777</li>
        <li>8888888888888888</li>

```

```
<li>9999999999999999</li>
</ul>
<ul class="numberList f16 red">
    <li>1010101010101010</li>
    <li>1111111111111111</li>
    <li>1212121212121212</li>
</ul>
</body>
```

按照方案二的思路，我们需要再定义一个新的类 numberList4，在 CSS 里需要修改好几处；按照方案三的思路，我们无需扩展新的类，只需在 HTML 标签的 class 里将之前定义的类重新组合即可。

在面向对象编程里，有类似的情况：继承与组合。继承的思路是将一个复杂且包含变化的类，拆分成几个复杂但稳定的子类。首先明确一个抽象的父类，父类有着几乎所有的方法和属性，子类继承自父类，根据需求，添加新的方法和属性，覆盖掉与父类有变化的方法和属性。但使用继承的话，任何一点小的变化也需要重新定义一个类，很容易引起类的爆炸式增长，产生一大堆有着细微不同的子类。组合的思路是将一个复杂的类分成容易产生变化的部分和相对稳定的部分，将容易变化的部分拆分出去，每一种可能的变化设计成一个个单独的类，相对稳定的部分设计成一个主体类，这样，将一个复杂的类拆分成几个简单的类，类之间没有继承关系，这遵循了面向对象设计的“单一职责”原则。这些容易变化的类的实例赋值给主体类作为一个属性，实现了类的组合。用组合的方式，可以大大减少类的数量。在面向对象编程里，有个很重要的原则就是“多用组合，少用继承”。一些偏激的工程师甚至认为继承是错误的，是造成维护性差的罪魁祸首，主张完全使用组合，拒绝使用继承。

方案三就是借鉴了编程领域类的组合的思想，将方案二中复杂的 numberList1 类、numberList2 类和 numberList3 类拆分成了几个相对简单的类，其中相对稳定的部分拆分成 numberList 类，而可能变化的部分拆成 f12 类、f16 类和 red 类。通过类的组合，很容易实现类的扩展，避免产生类爆炸。

HTML 标签的 class 属性和 id 属性不同，id 只能挂一个，而 class 可以挂多个，用空格分隔。例如“<p id='test'></p><div class="f12 red box"></div>”。HTML 的 class 与

程序中“类”有相同的“味道”，`class` 可以挂多个，从技术上支持了“组合”的用法。我们在使用 CSS 时，如果能灵活运用这点就可以大大减少类的数量，一方面减少了代码量，提高了可维护性，另一方面使类的职责更单一，弹性更强，增加了类的重用性，提高了开发效率。

挂多个 `class` 会不会让 HTML 标签看起来过于臃肿呢？这样做真的好吗？臃肿固然不好看，但它带来的好处却是不容忽视的，笔者推荐挂多个 `class`，哪怕它让 HTML 标签看起来不太轻盈。Yahoo 的 YUI3 官方演示文档中的一部分代码如代码清单 4-16 所示。

代码清单 4-16 YUI3 中的 `class`

---

```
<div class="yui-widget yui-overlay yui-widget-positioned yui-widget-stacked"> <!--Bounding Box-->

    <div class="yui-overlay-content yui-widget-stdmod">
<!--Content Box-->

    <div class="yui-widget-hd">Overlay Header Content</div> <!--
Header Section-->
    <div class="yui-widget-bd">Overlay Body Content</div> <!--
Body Section-->
    <div class="yui-widget-ft">Overlay Footer Content</div> <!--
Footer Section-->

</div>

<iframe class="yui-widget-shim"></iframe> <!-- Stacking shim,
if enabled-->

</div>
```

---

可以看出，Yahoo 的前端开发工程师也喜欢挂多个 `class` 的方式。

#### 4.4.4 如何处理上下 margin

对于模块来说，其上下 `margin` 不确定，因为美术设计的需求不同，可能同样样式的模块，在不同位置上有不同的上下 `margin`。如图 4-12 所示。

### 服务理念

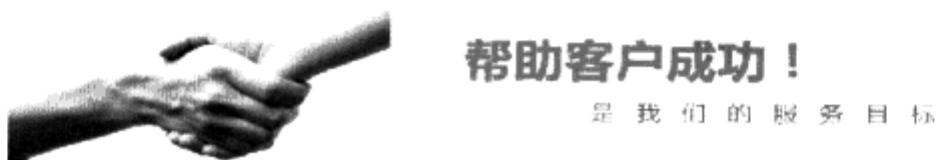
对于我们来说，服务不仅仅代表一种制度，它已经融入到我们的企业文化中，并为所有员工接受和奉行。服务内容来自用户的需求，我们一如既往地向用户提供优质、高效服务，并通过有偿服务方式增加服务自我造血机制，同时寻求服务内容和服务方式的改进，使双方从中获得最佳效益。

我们向用户提供标准化、专业化、多元化、产品化服务，标准化代表服务规范，专业化代表服务质量，多元化代表服务内容，产品化代表不同的服务质量与价值，都是以用户满意度为衡量标准。



### 服务宗旨

- 向用户提供持续、高效、优质的服务。
- 建立完善的服务体系，向用户提供专业化、标准化、多元化、产品化服务。
- 树立以用户为中心的工作作风，以用户满意作为质量工作的基础。



### Gsns产品的优势

#### Web2.0 社区应用和社交网络 (SNS) 全面解决方案

Gsns提供一整套完整的、全面集成的SNS和web2.0商业解决方案，能帮助社区类网站迅速向SNS(和web2.0)转型或者升级，是企业级商用客户的最佳选择。

到目前为止，无论是其它论坛产品提供商还是网站建设公司、或是独立的外包开发团队，还没有任何一家其他的产品能像Gsns互动社区解决方案一样，完全地将SNS网站和web2.0应用如此完美的全面的融合在一起。



##### 全面完善的产品组合

Gsns包括真实的空间主页系统和模板(Space)、日志系统(Blog)、相册照片系统(Photo)、圈子系统(Group)、交友应用(Friend)、活动(Event)社会化网络(Network)、SNS(好友动态)等主要系统构成，还包括积分等级/币管理、礼物、消息管理、邀请、音乐等应用！

##### 领先的技术架构，独特的设计理念

拥有独创的高效缓存机制，大大降低服务器负载，提高访问速度。具有可扩展、可管理、超负载支持超前的技术理念，让您博得先机，轻松形成行业竞争的门槛！

##### 优秀的用户体验

采用Ajax 和 RIA富技术，以用户为中心的设计理念，操作方便、快捷。使网络用户的在线体验更加愉悦！

##### 灵活开放

客户可根据自己的需求对Gsns通过API接口进行二次开发，充分满足更多的客户功能需求，同时我们也提供这样的服务，具体可与我们详细沟通！

图 4-12 简单示意图

图 4-13 中用线框出的部分有相同的样式、字号、颜色，且都有下划线。很明显，我们可以将它提取成一个通用的组件。如代码清单 4-17 所示。



图 4-13 简单示意图中样式相同的模块

## 代码清单 4-17 提取标题组件

```

<style type="text/CSS">
    .title{border-bottom:1px dashed #B2BCC6;color:#0066CF;font-
size:16px;font-weight:bold;}
</style>

<h2 class="title">服务理念</h2>
...
<h2 class="title">服务宗旨</h2>

```

```
...
<h2 class="title">Gsns 产品的优势</h2>
...
```

这些样式相同的模块还有上下 margin，我们应该如何设置它们的上下 margin 呢？提取组件时，需要将上下 margin 也包含进来吗？

需要按图 4-14 所示重新提取组件吗？可是这三个模块的 margin-top 并不相同，如图 4-15 所示。



图 4-14 包含上下 margin 的模块

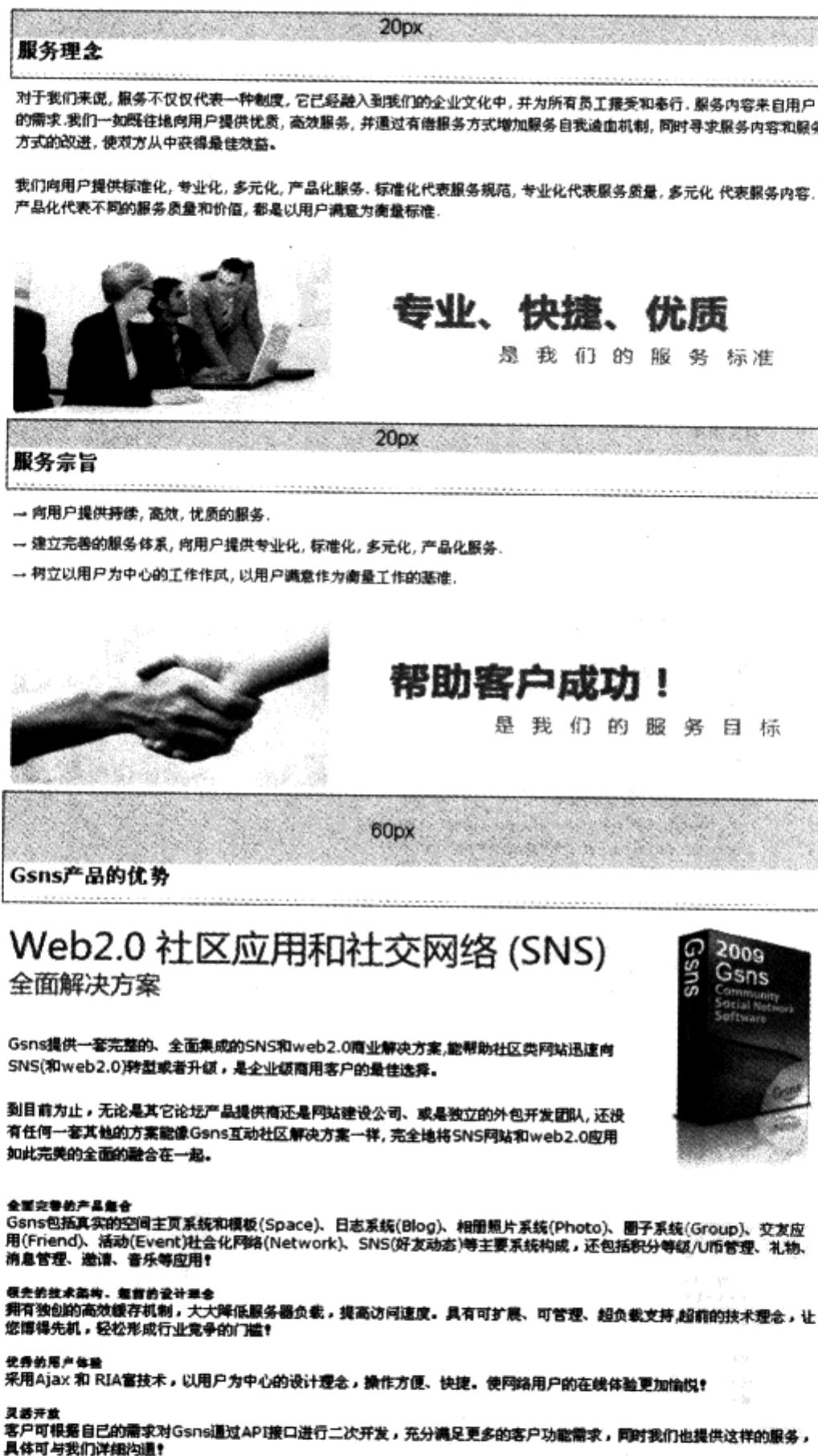


图 4-15 模块的 margin-top 不同

如果组件需包含 margin 值，那么其代码如代码清单 4-18 所示。

### 代码清单 4-18 包含 margin 的标题组件

---

```

<style type="text/CSS">
    .title1{border-bottom:1px dashed #B2BCC6;color:#0066CF;font-size:16px;
font-weight:bold;margin-top:20px;}
    .title2{border-bottom:1px dashed #B2BCC6;color:#0066CF;font-size:16px;
font-weight:bold;margin-top:60px;}
</style>

<h2 class="title1">服务理念</h2>
...
<h2 class="title1">服务宗旨</h2>
...
<h2 class="title2">Gsns 产品的优势</h2>
...

```

---

如果又出现一个新的设计，需要改变 margin-top 值，我们需要再定义 title3 和 title4 吗？包含 margin 这个不稳定样式到组件里，大大限制了类的重用性！如果再多设计几个 margin 值不同的模块，很容易产生类数量的大爆炸。这就是我们上节讲到过的“多用组合，少用继承”，解决这个问题的方法就是将类中不稳定的部分分离出来，单独设置成一个类，将相对稳定的剩下的那部分设置成另一个类，通过类的组合——挂多个 class 实现最终样式。模块的上下 margin 是类的组合的一个典型应用。应用类的组合如代码清单 4-19 所示。

### 代码清单 4-19 应用类的组合

---

```

<style type="text/CSS">
    .mt20{margin-top:20px;}
    .mt60{margin-top:60px;}
    .title{border-bottom:1px dashed #B2BCC6;color:#0066CF;font-size:16px;
font-weight:bold;}
</style>

<h2 class="title mt20">服务理念</h2>
...
<h2 class="title mt20">服务宗旨</h2>
...
<h2 class="title mt60">Gsns 产品的优势</h2>
...

```

---

除了给图中所示模块挂上 margin-top 的类，其实也可以给别的模块挂上 margin-bottom 的类，如图 4-16 所示。

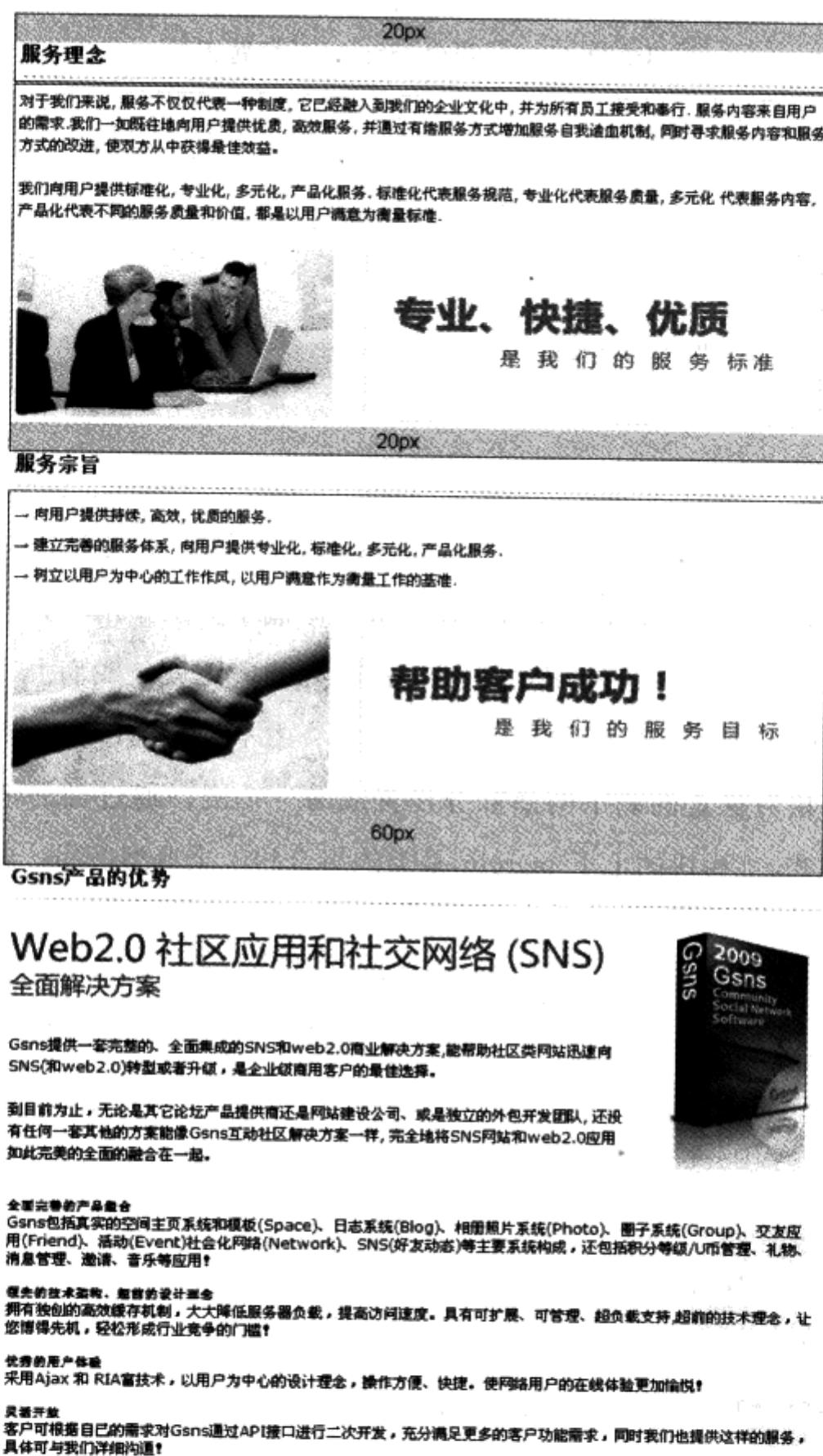


图 4-16 给其他模块加 margin-bottom

还有些工程师会采取如图 4-17 所示的添加方法。

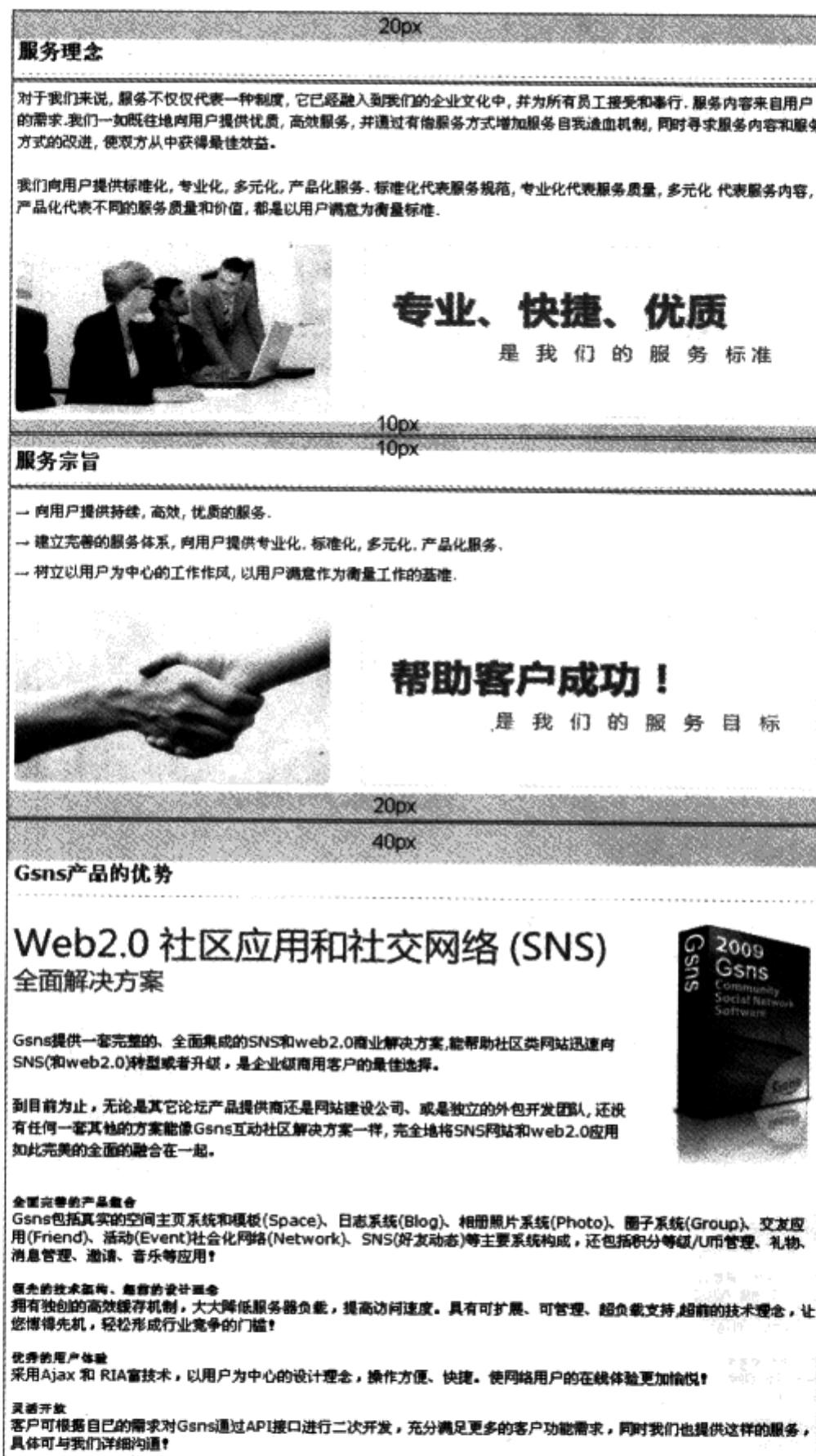


图 4-17 margin-top 和 margin-bottom 混用

图 4-17 混用了 margin-top 和 margin-bottom，但事实上，这种做法会带来意外的效果——上下 margin 重合问题。margin 是个有点特殊的样式，相邻的 margin-left 和

`margin-right` 是不会重合的，但相邻的 `margin-top` 和 `margin-bottom` 会产生重合。图 4-17 的真实效果如图 4-18 所示。

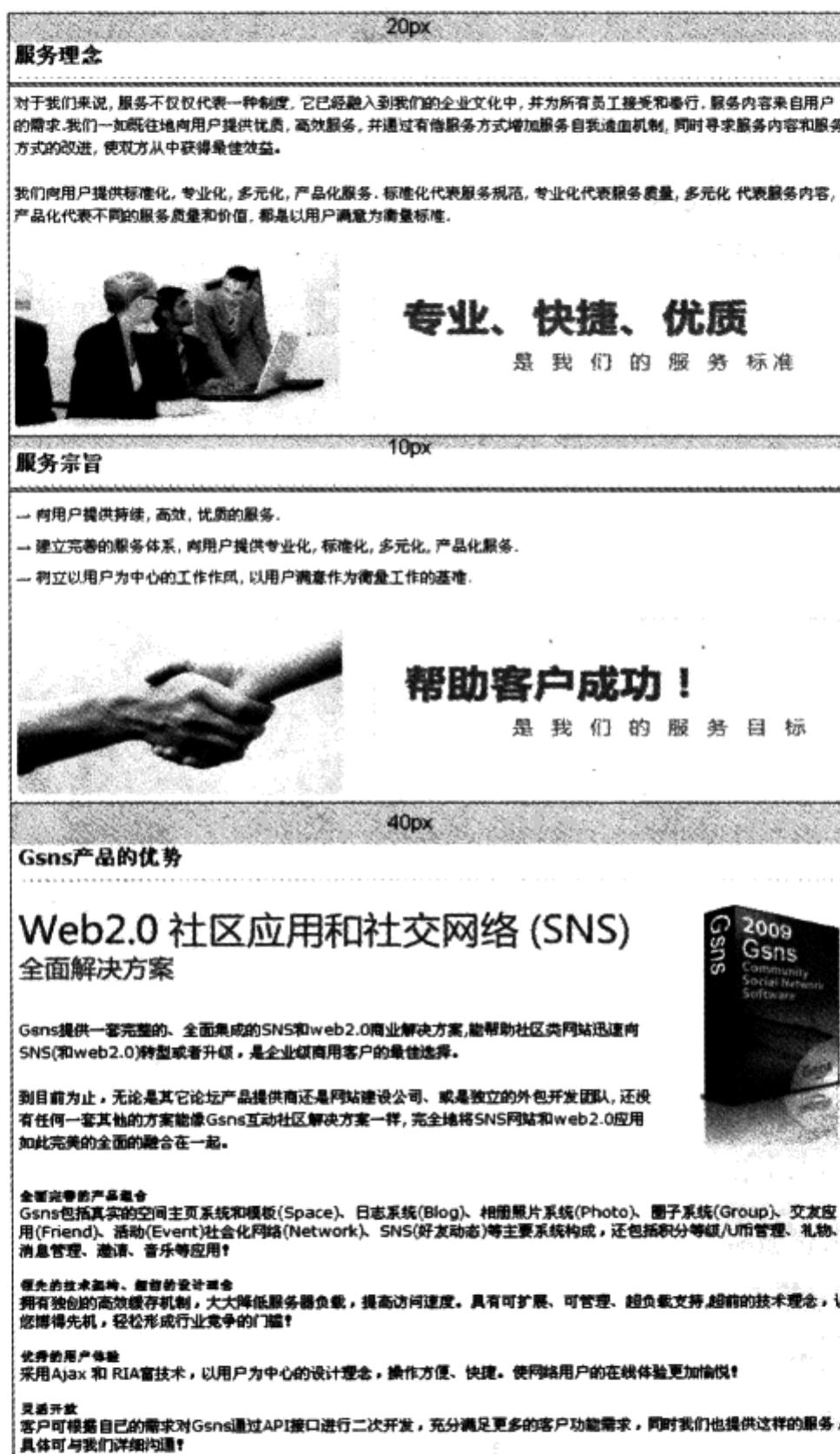


图 4-18 `margin-top` 和 `margin-bottom` 重合

如果对相邻的模块同时使用了 margin-top 和 margin-bottom，边距会重合带来不必要的麻烦，所以最好统一使用 margin-top 或者 margin-bottom，不要混合使用，从而降低出现问题的风险。这并不是技术上必需的，但却是一个良好的习惯。

**总结：**如果不确定模块的上下 margin 特别稳定，最好不要将它写到模块的类里，而是使用类的组合，单独为上下 margin 挂用于边距的原子类（例如 mt10、mb20）。模块最好不要混用 margin-top 和 margin-bottom，统一使用 margin-top 或 margin-bottom。

## 4.5 低权重原则——避免滥用子选择器

CSS 设置的样式是可以层叠的，如代码清单 4-20 所示。

代码清单 4-20 CSS 的层叠

---

```
<style type="text/CSS">
span{font-size:40px}
.test{color:red}
</style>

<span class="test">1234567890</span>
```

---

“1234567890”既可以得到“font-size:40px”的样式，又可以得到“color:red”的样式。如果两个不同选择符设置的样式有冲突，又会如何呢？如代码清单 4-21 所示。

代码清单 4-21 CSS 层叠有冲突的情况

---

```
<style type="text/CSS">
span{font-size:40px;color:green}
.test{color:red}
</style>

<span class="test">1234567890</span>
```

---

“1234567890”的颜色会是什么呢？是对“span”设置的“color:green”呢，还是对“.test”设置的“color:red”呢？这就涉及 CSS 选择符的权重问题了。

CSS 的选择符是有权重的，当不同选择符的样式设置有冲突时，会采用权重高的选择符设置的样式。权重的规则是这样的：HTML 标签的权重是 1，class 的权重是 10，id

的权重是 100，例如 p 的权重是 1，“div em”的权重是  $1+1=2$ ，“strong.demo”的权重是  $10+1=11$ ，“#test .red”的权重是  $100+10=110$ 。

代码清单 4-21 中，选择符 span 的权重是 1，“.test”的权重是 10，所以“1234567890”的 color 应该是 green。

但如果如代码清单 4-22，又会如何呢？

代码清单 4-22 选择符权重相同的情况

---

```
<style type="text/CSS">
span{font-size:40px}
.test{color:red}
.test2{color:green}
</style>

<span class="test test2">1234567890</span>
```

---

span 标签同时挂了“test”和“test2”两个 class，它们的权重都是 10，那么“1234567890”的 color 又该是哪个呢？如果 CSS 选择符权重相同，那么样式会遵循就近原则，哪个选择符最后定义，就采用哪个选择符的样式。代码清单 4-22 中，“.test2”定义晚于“.test”，所以“1234567890”的 color 会采用“.test2”的设置，值为 green。

如果改变“.test”和“.test2”定义的位置，如代码清单 4-23。

代码清单 4-23 调换样式的位置

---

```
<style type="text/CSS">
span{font-size:40px}
.test2{color:green}
.test{color:red}
</style>

<span class="test test2">1234567890</span>
```

---

那么“1234567890”的 color 就为 red 了。

需要强调的是“就近原则”指的是选择符定义的先后顺序，而不是挂 class 名的先后顺序，`<span class="test test2">`和`<span class="test2 test">`没有区别。

CSS 选择符权重是个容易被忽视的问题，但事实上如果不太注意选择符权重，常会出现意想不到的小麻烦。例如有段代码如代码清单 4-24 所示。

#### 代码清单 4-24 需标红的代码

```
<style type="text/CSS">
#test{font-size:14px;}
</style>
<p id="test">CSS 选择符权重很重要</p>
```

现在需要将“很重要”三个字设置为红色，我们应该怎么做呢？

方案一，利用子选择器，如代码清单 4-25 所示。

#### 代码清单 4-25 用子选择器

```
<style type="text/CSS">
#test{font-size:14px;}
#test span{color:red;}
</style>
<p id="test">CSS 选择符权重很重要</span></p>
```

方案二，新建 class，如代码清单 4-26 所示。

#### 代码清单 4-26 新建 class

```
<style type="text/CSS">
#test{font-size:14px;}
.font{color:red;}
</style>
<p id="test">CSS 选择符权重很重要</span></p>
```

很多工程师推荐使用方案一，因为使用子选择器可以避免新增 class，让 HTML 代码更简洁。这么考虑是有道理的，但如果这时需求有变化，需要添加新的文字进来，如代码清单 4-27 所示。

#### 代码清单 4-27 添加新的文本

```
<style type="text/CSS">
#test{font-size:14px;}
#test span{color:red;}
</style>
<p id="test">CSS 选择符权重很重要, 我们要小心处理</p>
```

要求我们将“小心处理”这几个字设置为绿色，我们可以这么做，如代码清单 4-28 所示。

## 代码清单 4-28 需标绿的代码

---

```
<style type="text/CSS">
#test{font-size:14px;}
#test span{color:red;}
.font{color:green;}
</style>
<p id="test">CSS 选择符权重<span>很重要</span>, 我们要<span class="font">小心处理</span></p>
```

---

本以为“小心处理”会被选择符“.font”设置为绿色，但它却被选择符权重更高的“#test span”设置成了红色，子选择器在无意中影响到了我们新添加的代码。如果想要达到我们的预期，我们需要按代码清单 4-29 重写代码。

## 代码清单 4-29 被迫加重权重的选择器

---

```
<style type="text/CSS">
#test{font-size:14px;}
#test span{color:red;}           /*选择符权重为 100+1=101*/
#test .font{color:green;}       /*选择符权重为 100+10=110*/
</style>
<p id="test">CSS 选择符权重<span>很重要</span>, 我们要<span class="font">小心处理</span></p>
```

---

而如果使用方案二，情况又会如何呢？如代码清单 4-30 所示。

## 代码清单 4-30 新增 class 作为标绿容器的选择符

---

```
<style type="text/CSS">
#test{font-size:14px;}
.font{color:red;}
.font2{color:green;}
</style>
<p id="test">CSS 选择符权重<span class="font">很重要</span>, 我们要<span class="font2">小心处理</span></p>
```

---

因为没有使用子选择器，所以我们给新添的代码挂上新的 class，就可以顺利地完成样式设置了。

使用子选择器，会增加 CSS 选择符的权重，CSS 选择符的权重越高，样式越不易被覆盖，越容易对其他选择符产生影响。所以，除非确定 HTML 结构非常稳定，一定不会再修改了，否则尽量不要使用子选择器。为了保证样式容易被覆盖，提高可维护性，CSS 选择符需保证权重尽可能低。

少使用子选择器，就需要多添加 class，在 Web 标准盛行的初期，很多工程师认为多添加 class 是不好的，如果能使用子选择器就应尽量使用，使用大量 class 的做法叫做“多 class 痘”。在经过大量实践后，我并不认为多 class 有太大坏处，相反，与使用子选择器相比，新添 class 反而更利于维护。

## 4.6 CSS sprite

图 4-19 所示为一个导航条。

当鼠标滑过时，需要有如图 4-20 所示的高亮效果。



图 4-19 导航条



图 4-20 鼠标滑过时高亮

实现代码，如代码清单 4-31 所示。

代码清单 4-31 用两张图实现高亮

---

```

<style type="text/CSS">
    .nav li{float:left; display:inline; margin-right:10px; font-family: 黑体;}
    .nav a{float:left; width:139px; height:31px; line-height:31px; font-size:24px; color:#fff; text-decoration:none; text-align:center; background: url(img1.gif);}
    .nav a:hover{background:url(img2.gif);}
</style>
<ul class="nav">
    <li><a href="#">联系我们</a></li>
    <li><a href="#">产品答疑</a></li>
    <li><a href="#">广告服务</a></li>
</ul>

```

---

我们根据设计图，将默认状态的背景和滑过状态的背景切成两张图片，分别命名为 img1 和 img2，如图 4-21 和图 4-22 所示。



图 4-21 img1.gif —— 默认状态的背景



图 4-22 img2.gif——滑过状态的背景

给 a 标签的默认状态设置样式 background:url (img1.gif)，给滑过状态设置样式 background:url (img2.gif)，这样便可以实现需求。但这种做法存在一个问题，img2.gif 并不会马上加载，只有当鼠标滑过时，才会开始加载，在图片加载完成以前，背景会显示为空白。

为了解决这个问题，我们修改了部分代码，如代码清单 4-32 所示。

代码清单 4-32 利用一张大图的背景移动实现高亮

```
<style type="text/CSS">
    .nav li{float:left; display:inline; margin-right:10px; font-family:黑体;}
    .nav a{float:left; width:139px; height:31px; line-height:31px; font-size:24px; color:#fff; text-decoration:none; text-align:center; background:url(img3.gif);}
    .nav a:hover{background-position:0 -31px;}
</style>
<ul class="nav">
    <li><a href="#">联系我们</a></li>
    <li><a href="#">产品答疑</a></li>
    <li><a href="#">广告服务</a></li>
</ul>
```

这种方法将默认状态和滑过状态的两张图片合在一起，如图 4-23 所示。



图 4-23 img3.gif ——默认状态和滑过状态的背景图合并

它的思路是通过改变背景图的 background-position 属性来在指定大小的容器里展示部分位置的背景图，如图 4-24 所示。



图 4-24 默认状态和滑过状态背景图的位置

我们将这种技术称为图片翻转技术。

图片翻转技术将多张图片合并为一张，然后利用 `background-position` 属性来展示我们需要的部分。按照这种思路，后来衍生出的另一种技术将这种思路发挥到了极致——将网站的多张背景图片合并到一张大图片上，这便是 CSS sprite。

CSS sprite 将网站的背景图合到一张大图片上，它的出发点已经不仅是“解决滑过状态时背景图片出现空白”的问题了。图片的加载是会发出 HTTP 请求的，一张图片需要一条 HTTP 请求，如果一个页面需要加载多张图片，那么它会相应地发出多条 HTTP 请求。HTTP 请求数越多，访问服务器的次数就越多，服务器的压力也就越大。将多张图片合并成一张大图，会大大减少网页的 HTTP 请求数，减小服务器压力。

对于流量大的网站来说，使用 CSS sprite 技术非常有价值，Amazon 和腾讯的网站便是两个非常典型的例子，两个网站的 CSS sprite 大图分别如图 4-25 和图 4-26 所示。



图 4-25 www.amazon.com 的 CSS sprite 大图

图 4-26 www.qq.com 的 CSS sprite 大图

CSS sprite 技术看似简单，其实不容易掌握，主要有如下原因：

- 1) 它能合并的只能是用于背景的图片，对于`<img src="" />`设置的图片，是不能合并到 CSS sprite 大图中的，如果合并这些图片会影响页面可读性。
- 2) 对于横向和纵向都平铺的图片，也不能使用 CSS sprite；如果是横向平铺的，只能

将所有横向平铺的图合并成一张大图，只能竖直排列，不能水平排列；如果是纵向平铺的，我们只能将所有纵向平铺的图合并成一张大图，只能水平排列，不能竖直排列。

图片如何排列能够尽量紧凑，同时保证不会影响扩展性。这点是 CSS sprite 技术最困难也是最具挑战性的地方。

CSS sprite 技术虽然越来越受人追捧，但它也存在一定问题——将多张图片合并为一张图片，通过 background-position 进行定位，这对于图片的位置精确程度要求非常高，一方面在制作网页时，我们需要精确测量坐标，还需要考虑如何让图案尽可能密集地排列，这影响了开发速度；另一方面，大图中每个小图的坐标值都不可轻易改动，因为每改动一个小图，都可能影响到周边的其他图片，这大大降低了可维护性。

CSS sprite 的最大好处是减少 HTTP 请求数，减轻服务器的压力，但它却需要付出“降低开发效率”和“增大维护难度”的代价。对于流量并不大的网站来说，CSS sprite 带来的好处并不明显，而它付出的代价却很大，其实并不划算。所以，是否使用 CSS sprite 主要取决于网站流量。

## 4.7 CSS 的常见问题

### 4.7.1 CSS 的编码风格

CSS 常见的编码风格有两种：多行式和一行式。

多行式，如代码清单 4-33 所示。

代码清单 4-33 多行式的 CSS 编码风格

```
.test{  
    width:100px;  
    height:50px;  
    color:#ccc;  
}  
.demo{  
    background-color:green;  
    font-size:20px;  
}
```

一行式，如代码清单 4-34 所示。

#### 代码清单 4-34 一行式的 CSS 编码风格

```
.test{width:100px;height:50px; color:#ccc;}  
.demo{background-color:green; font-size:20px;}
```

多行式的编码风格可读性更强，但缺点是容易使 CSS 文件的行数过多，编辑样式时，需要来回拖动文本编辑器的滚动条，影响开发速度，另外，过多的空白，也会增大 CSS 文件的大小。一行式的编码风格在可读性方面稍差一点，但可以有效减少 CSS 文件的行数，有利于提高开发速度，同时也可以减小 CSS 文件的大小。

CSS 布局刚刚兴起时，因为缺少强大的调试工具，很多时候，样式调试的工作需要通过仔细阅读 CSS 文件来完成，所以良好的可读性对于调试样式来说，起到了非常重要的作用。这一时期，多行式的编码风格是主流。随着前端调试工具的日益强大，特别是 Firefox + firebug 的流行，样式调试的工作已经不再需要仔细阅读 CSS 文件了，一行式编码风格逐渐取代多行式编码风格成为了新的主流。笔者更推荐使用一行式编码风格。

### 4.7.2 id 和 class

`id` 和 `class` 作为 CSS 选择符最常用的挂钩，其区别已经成了一个经典问题。

- 1) 同一个网页，相同的 `id` 只能出现一次，它不可重复，而 `class` 可以任意出现多次；
- 2) `id` 的 CSS 选择符权重为 100，而 `class` 的选择符权重为 10；
- 3) 原生 JS 提供 `getElementById()` 方法，支持通过 `id` 对应到相关的 `HTMLLIElement`，但原生 JS 不支持通过 `class` 对应到相关 `HTMLLIElement`。更多关于 `id` 和 `class` 的 JS 用法，详见 5.3.2 节。

一般来说，`id` 因为不能重用，使用 `id` 会限制网页的扩展性。比如，设计图中有某个模块只出现过一次，我们使用了 `id` 作为挂钩，这本身没有问题。但如果需求变更，设计图中同样的模块需要再增加一个，这时问题就出现了，因为使用 `id` 作为挂钩，不能重用。所以，一般情况下，建议尽量使用 `class`，少用 `id`。

### 4.7.3 CSS hack

CSS hack 的方式有很多种，这里我对常用的方式做一下归纳。

## 1. IE 条件注释法

首先，因为 IE 浏览器在 CSS 的解析上存在很多问题，所以绝大多数时候我们的 CSS hack 都是针对 IE 进行的。微软也知道自己浏览器的不足，所以推出了官方的 hack 方法——IE 条件注释。

如果我们想针对 IE 引入一个 CSS 文件，代码如代码清单 4-35 所示。

代码清单 4-35 只在 IE 下生效

---

```
<!--[if IE]>
<link type="text/CSS" href="test.css" rel="stylesheet" />
<![endif]-->
```

---

CSS 文件 test.css 就只加载到 IE 浏览器了，对于非 IE 浏览器就会忽略这条语句。

如果我们想针对某个特定版本的 IE 浏览器加载 CSS 文件，代码如代码清单 4-36 所示。

代码清单 4-36 只在 IE6 下生效

---

```
<!--[if IE 6]>
<link type="text/CSS" href="test.css" rel="stylesheet" />
<![endif]-->
```

---

如果我们想针对某个范围以内版本的 IE 进行 hack，可以结合 `lte`、`lt`、`gte`、`gt`、`!` 关键字来进行注释，其中 `lte` 表示“小于等于”，`lt` 表示“小于”，`gte` 表示“大于等于”，`gt` 表示大于，“`!`”表示“不等于”。其用法如代码清单 4-37 和代码清单 4-38 所示。

代码清单 4-37 只在 IE6 以上版本生效

---

```
<!--[if gt IE 6]>
<link type="text/CSS" href="test.css" rel="stylesheet" />
<![endif]-->
```

---

代码清单 4-37 表示当浏览器版本号大于 6 时，会加载 test.css。

代码清单 4-38 只在 IE7 上不生效

---

```
<!--[if ! IE 7]>
<link type="text/CSS" href="test.css" rel="stylesheet" />
<![endif]-->
```

---

代码清单 4-38 表示当浏览器版本号不等于 7 时，会加载 test.css。

事实上，虽然条件注释最常用于 CSS 的 hack，但它能包含的内容其实不仅仅是 link 标签，它还可以用这样的形式来进行 hack，如代码清单 4-39 所示。

代码清单 4-39 条件注释和 style 标签

---

```
<!--[if IE 6]>
<style type="text/CSS">
.test{}
</style>
<![endif]-->
```

---

甚至可以进行一些 JS 的 hack，如代码清单 4-40 所示。

代码清单 4-40 条件注释和 script 标签

---

```
<!--[if IE 6]>
<script type="text/JavaScript">
alert("我是IE 6");
</script>
<![endif]-->
```

---

上面这段代码只会在 IE6 下执行。

IE 条件注释是微软官方推荐的 hack 方式，从向前兼容性方面考虑，它是最安全的 hack 方式，理论上它是最好的选择。但这种方式需要将所有的 hack 根据浏览器类型集中在相应的文件中，比如，有个.test 类，要求在 IE 6 下 width 为 60px，在 IE 7 下 width 为 70px，在 IE 8 下 width 为 80px，用 IE 条件注释的方式 hack，我们就需要写三个 CSS 文件，如代码清单 4-41 所示。

代码清单 4-41 ie6.css、ie7.css 和 ie8.css

---

- 1) ie6.css  
.test{width:60px;}
- 2) ie7.css  
.test{width:70px;}
- 3) IE 8.css  
.test{width:80px;}

---

通过 IE 条件注释，将 ie6.css、ie7.css 和 ie8.css 导入到网页中，如代码清单 4-42 所示。

### 代码清单 4-42 针对不同 IE 版本分别导入样式

---

```
<!--[if IE 6]>
<link type="text/CSS" href="ie6.css" rel="stylesheet" />
<![endif]-->
<!--[if gt IE 7]>
<link type="text/CSS" href="ie7.css" rel="stylesheet" />
<![endif]-->
<!--[if gt IE 8]>
<link type="text/CSS" href="IE 8.css" rel="stylesheet" />
<![endif]-->
```

---

虽然它的向后兼容性是最好的，但它的缺点也非常明显：将同一 CSS 选择符下的样式分散到了三个文件中去控制，增加了开发和维护成本。

### 2. 选择符前缀法

选择符前缀法是另一种常用的 hack 方法。它的原理是在 CSS 选择符前加一些只有特定浏览器才能识别的前缀，从而让某些样式只对特定浏览器生效。例如“\*html”前缀只对 IE 6 生效，“\*+html”前缀只对 IE 7 生效。

代码清单 4-42 的效果用选择符前缀法来实现，如代码清单 4-43 所示。

### 代码清单 4-43 选择符前缀 hack 法

---

```
<style type="text/CSS">
  .test{width:80px;}                                /*IE 6,IE 7,IE 8*/
  *html .test{width:60px;}                          /*only for IE 6*/
  *+html .test{width:70px;}                         /*only for IE 7*/
</style>
```

---

选择符前缀法相较于 IE 条件注释法来说，“.test 类”的样式可以集中起来，可维护性强了很多。但不能保证以后的 IE 9、IE 10 不识别\*html 和\*+html，在向后兼容性上存在一点风险。

另外，选择符前缀法不能用于内联样式上，比如说

### 3. 样式属性前缀法

样式属性前缀法的原理是在样式的属性名前加前缀，这些前缀只在特定浏览器下才

生效。例如“\_”只在 IE 6 下生效，“\*”在 IE 6 和 IE 7 下生效。

代码清单 4-42 的效果用样式属性前缀法来实现，如代码清单 4-44 所示。

代码清单 4-44 样式属性前缀 hack 法

---

```
<style type="text/CSS">
  .test{width:80px; *width:70px; _width:60px;}
</style>
```

---

样式属性前缀法比起选择符前缀法聚合程度更高，代码更精简，可维护性很强，但和选择符前缀法一样，它在向后兼容性上存在一点风险。

另外，样式属性前缀法可以用于内联样式上，比如`<div class="test" style="width:80px; *width:70px; _width:60px" ></div>`。

虽然 IE 条件注释法在理论上是首推的 hack 方法，但因为它不利于开发和维护，所以事实上最流行的 hack 方式是选择符前缀法和样式属性前缀法，我们可以根据需要选择使用。

#### 4.7.4 解决超链接访问后 hover 样式不出现的问题

有时候我们同时设置了 `a:visited` 和 `a:hover` 的样式，但一旦超链接访问后，`hover` 的样式就不再出现，这是怎么回事呢？

这是因为将 `a:visited` 和 `a:hover` 的顺序放错了。如代码清单 4-45 所示。

代码清单 4-45 不正确的伪类顺序

---

```
<style type="text/CSS">
  a:hover{color:yellow;}
  a:visited{color:green;}
</style>
<a href="#">hello world</a>
```

---

代码清单 4-45 所示代码，如果没有点击 `a` 标签，鼠标滑过的时候，颜色会显示为黄色，但一旦点击 `a` 标签，颜色就将变成绿色，鼠标滑过时颜色也不再显示为黄色。

我们将 `a:visited` 和 `a:hover` 的排列顺序换一下，如代码清单 4-46 所示。

代码清单 4-46 正确的伪类顺序

---

```
<style type="text/CSS">
  a:visited{color:green;}
```

---

```
a:hover{color:yellow;}  
</style>  
<a href="#">hello world</a>
```

---

这样，不管 a 标签是否被点击过，鼠标滑过时都会显示为黄色。

关于 a 标签的四种状态的排序问题，有个简单好记的原则，叫做 love hate 原则，即 l(link)ov(visited)e h(hover)a(active)te。

#### 4.7.5 hasLayout

很多时候，CSS 在 IE 下的解析十分奇怪，明明在 Firefox 中显示得非常正确，但到了 IE 下却出现了问题，有的时候，这些问题甚至表现得非常诡异——例如一个比较经典的 Bug 就是设置 border 的时候，有时候 border 会断开，刷新页面或者滚动滚动条的时候，断掉的部分又会连接起来。

这些诡异的问题往往大部分都和 IE 下一个神秘的属性相关——hasLayout。hasLayout 是 IE 浏览器专有的一个属性，用于 CSS 的解析引擎。有时候在 IE 下一些复杂的 CSS 设置解析起来会出现 Bug，其原因可能与 hasLayout 没有被自动触发有关，我们通过一些技巧，手动触发 hasLayout 属性就可以解决 Bug 了。这也算是针对 IE 下疑难杂症的特殊偏方了，很多时候，触发了 hasLayout 就可以药到病除了。

hasLayout 的触发方法有很多种，例如设置 width、height 值，设置 position 为 relative 等。但如果设置了 width、height 或 position 都会在触发 hasLayout 的同时带来一些副作用的。早期的一些工程师推荐使用 “height:1%” 来触发 hasLayout，那时还没有出现 IE 7，而 height 属性在 IE 6 下其实是按照 “min-height” 来解析的，所以只要对 IE 6 进行 hack，“\* html{height:1%}” 就可以触发 hasLayout，同时又不带来副作用了。后来出现的 IE 7 仍然存在很多 hasLayout 方方面的问题，但 IE 7 已经能够正确识别 height 属性了，“height:1%” 的方法已经不再适用了。

于是，一个更好的解决方法开始流行，它使用了一个生僻的 CSS 属性 zoom 来触发 hasLayout——“zoom:1”。使用 “zoom:1” 可以触发 hasLayout，并且不会像 height 等属性一样引入副作用，更妙的是，我们可以不用 CSS hack 了。但 “zoom:1” 并不是一定可以触发 hasLayout 的，在极少数特殊的情况下，例如非常复杂的 CSS 设置，特别是使

用 DHTML 的时候，使用“zoom:1”有时也会无效，这时，我们可能需要借助更为强大的“position:relative”来帮助触发 hasLayout。总之，“zoom”是最常用、最安全、成本最小的触发 hasLayout 的方式，一般情况下，使用它就完全可以触发 hasLayout 了。如果遇到很特殊的情况，“zoom:1”无效的情况下，我们可以通过设置“position:relative”来触发 hasLayout，尽管它会带来一点副作用。

值得强调的是，hasLayout 的设计初衷是用于辅助块级元素的盒模型解析的，它是用于块级元素的。如果用于行内元素，会引发一些特殊的效果，详见 4.7.7 节。

#### 4.7.6 块级元素和行内元素的区别

块级元素和行内元素是布局最基本的两种元素，常见的块级元素有 div、p、form、ul、ol、li 等，常见的行内元素有 span、strong、em 等。

块级元素和行内元素有什么区别呢？块级元素会独占一行，默认情况下，其宽度自动填满其父元素宽度，行内元素不会独占一行，相邻的行内元素会排列在同一行里，直到一行排不下，才会换行，其宽度随元素的内容而变化，如代码清单 4-47 所示。

代码清单 4-47 块级元素和行内元素的区别

---

```
<style type="text/CSS">
p{background:red}
div{background:yellow}
span{background:blue}
strong{background:green}
</style>
<p>块级元素 p</p><div>块级元素 div</div><span>行内元素 span</span><strong>行
内元素 strong</strong>
```

---

在浏览器中的效果如图 4-27 所示



图 4-27 块级元素和行内元素

块级元素可以设置 width、height 属性。行内元素设置 width、height 属性无效。如代码清单 4-48 所示。

## 代码清单 4-48 为块级元素和行内元素设置长宽

```
<style type="text/CSS">
p{background:red;width:200px;height:200px;}
div{background:yellow;width:400px;height:100px;}
span{background:blue;width:200px;height:200px;}
strong{background:green;width:400px;height:100px;}
</style>
<p>块级元素 p</p><div>块级元素 div</div><span>行内元素 span</span><strong>行
内元素 strong</strong>
```

在浏览器中的效果如图 4-28 所示。

注意，块级元素即使设置了宽度，仍然是独占一行的。

块级元素可以设置 margin 和 padding 属性。行内元素的 margin 和 padding 属性很奇怪，水平方向的 padding-left、padding-right、margin-left、margin-right 都产生边距效果，但竖直方向的 padding-top、padding-bottom、margin-top、margin-bottom 却不会产生边距效果。如代码清单 4-49 所示。



图 4-28 设置了长宽的块级元素

和行内元素

## 代码清单 4-49 块级元素和行内元素的 margin 和 padding

```
<style type="text/CSS">
p{background:red;padding:20px;margin:20px;}
div{background:yellow;padding:20px;}
span{background:blue;padding:20px;margin:20px;}
strong{background:green;padding:20px;margin:20px;}
</style>
<p>块级元素 p</p><div>块级元素 div</div><span>行内元素 span</span><strong>行
内元素 strong</strong>
```

在浏览器中的效果如图 4-29 所示。

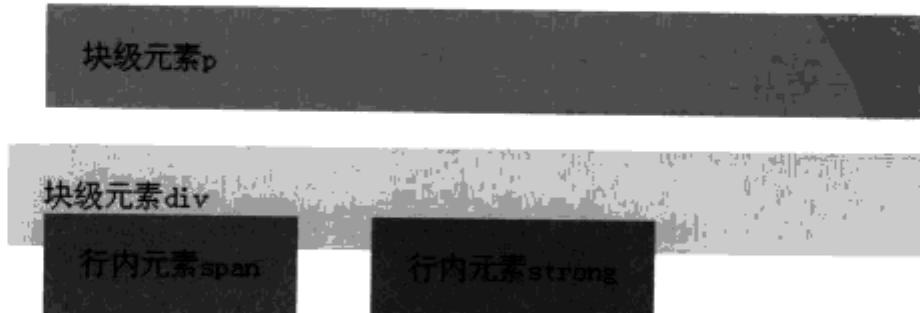


图 4-29 设置了 padding 和 margin 的块级元素和行内元素

如图 4-29 所示，竖直方向的 margin 看不到任何效果，竖直方向的 padding 虽然增大了行内元素的面积，但并没有和相邻元素拉开距离。

块级元素和行内元素的 CSS 相关属性是 display，其中块级元素对应于 display:block，行内元素对应于 display:inline。我们可以通过修改 display 属性来切换块级元素和行内元素，如代码清单 4-50 所示。

代码清单 4-50 改变块级元素和行内元素的显示

---

```
<style type="text/CSS">
p{background:red;display:inline;}
div{background:yellow;display:inline;}
span{background:blue;display:block;}
strong{background:green;display:block;}
</style>
<p>块级元素 p</p><div>块级元素 div</div><span>行内元素 span</span><strong>行
内元素 strong</strong>
```

---

在浏览器中的效果如图 4-30 所示。



图 4-30 更改了 display 的块级元素和行内元素

#### 4.7.7 display:inline-block 和 hasLayout

display 的值除了 block 和 inline，还有其他值，例如 list-item、table-cell 等，但因为 IE 6 和 IE 7 浏览器支持的 display 类型很少，所以为了兼容 IE，我们真正能用的 display 类型只有 block、inline 和 none 三种。

对于另一种非常有用的 display 类型 inline-block，其实在 IE 6 和 IE 7 下也是有办法实现的，但需要注意的是，并不是说 IE 6 和 IE 7 支持 display:inline-block，它的实现其实是一种 hack——触发行内元素的 hasLayout。在 4.7.5 节中我们讲到过，hasLayout 是 IE 浏览器为解析盒模型而设计的一个专有属性，它的设计初衷是用于块级元素的，如果触发行内元素的 hasLayout，就会让行内元素拥有一些块级元素的特性。

先说说 display: inline-block 的特性吧，顾名思义，它是行内的块级元素，它拥有块级元素的特点，可以设置长宽，可以设置 margin 和 padding 值，但它却不是独占一行

的，它的宽度并不占满父元素，而是和行内元素一样，可以和其他行内元素排在同一行里。它集块级元素和行内元素的特点于一身，是个非常有用的 display 类型。

IE 6 和 IE 7 是不支持 display:inline-block 的，如代码清单 4-51 所示。

代码清单 4-51 设置 display:inline-block

```
<style type="text/CSS">
p{color:red;width:100px;background:#ccc;height:30px;display:inline-
block;}
</style>
<body>
abcdefg <p>12345</p>
</body>
```

在 IE 6、IE 7、IE 8 和 Firefox 下的截图分别如下：

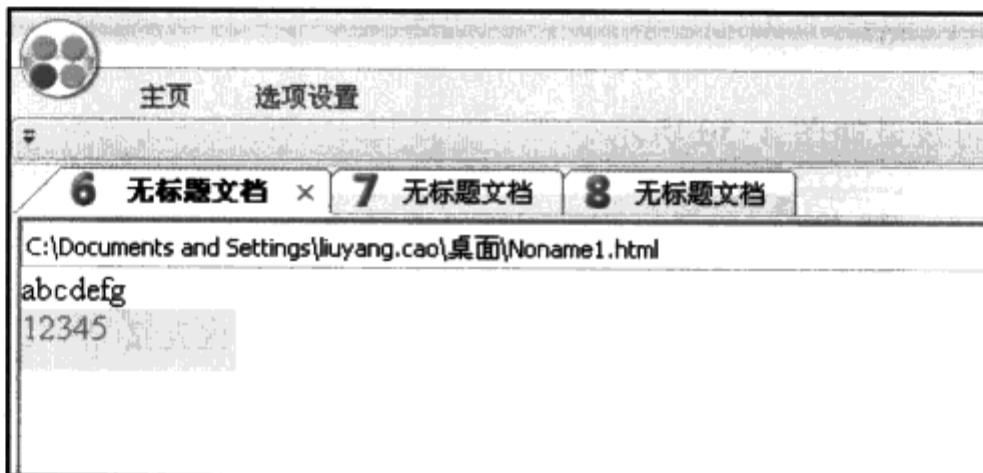


图 4-31 IE 6 下的截图

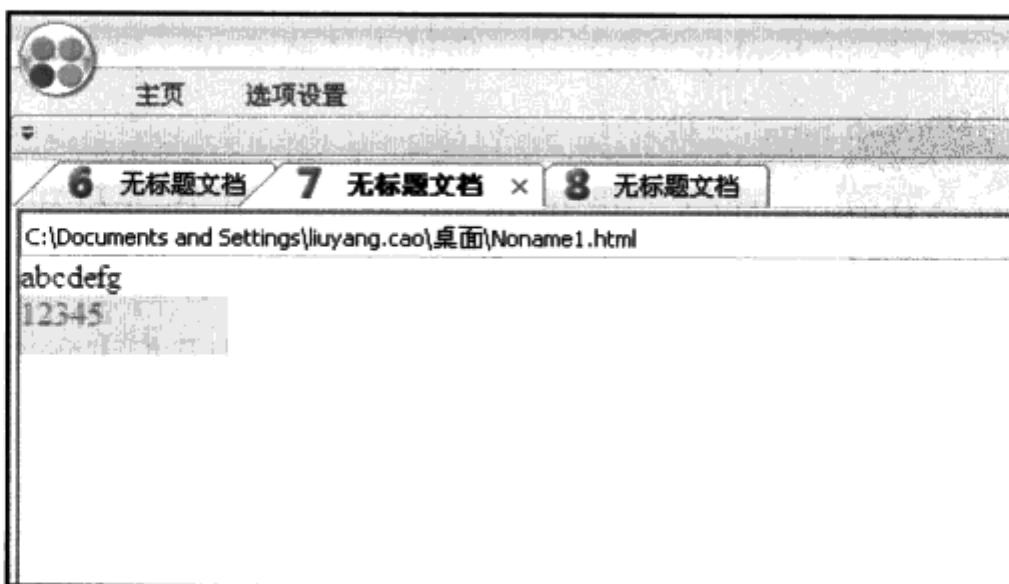


图 4-32 IE 7 下的截图

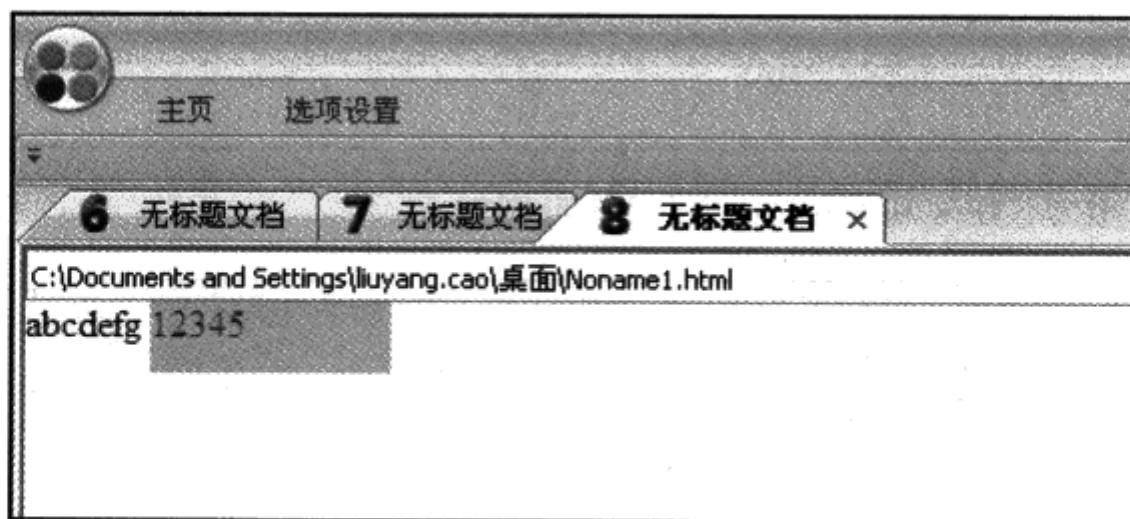


图 4-33 IE 8 下的截图



图 4-34 Firefox 下的截图

正常情况下，p 内的 12345 会独占一行，显示在 abcdefg 下的，如果对 p 设置 display:inline-block，它会显示在 abcdefg 后面，长宽设置保留（如果设置成 display:inline 长宽就丢失了）。我们看到 Firefox 和 IE 8 都可以正常解析 display:inline-block，而 IE 6 和 IE 7 还不支持，考虑兼容性，我们必须舍弃这种用法。

如前所述，如果我们对行内元素触发 hasLayout，就可以模拟 display:inline-block 的效果，将代码清单 4-51 修改一下，如代码清单 4-52 所示。

#### 代码清单 4-52 触发 hasLayout 在 IE6、IE7 下模拟 inline-block

---

```
<style type="text/CSS">
  span{color:red;width:100px;background:#ccc;height:30px;display:inline-
block;}
</style>
```

```
<body>
abcdefg <span>12345</span>
</body>
```

在 IE 6、IE 7、IE 8 和 Firefox 下的截图分别如下：

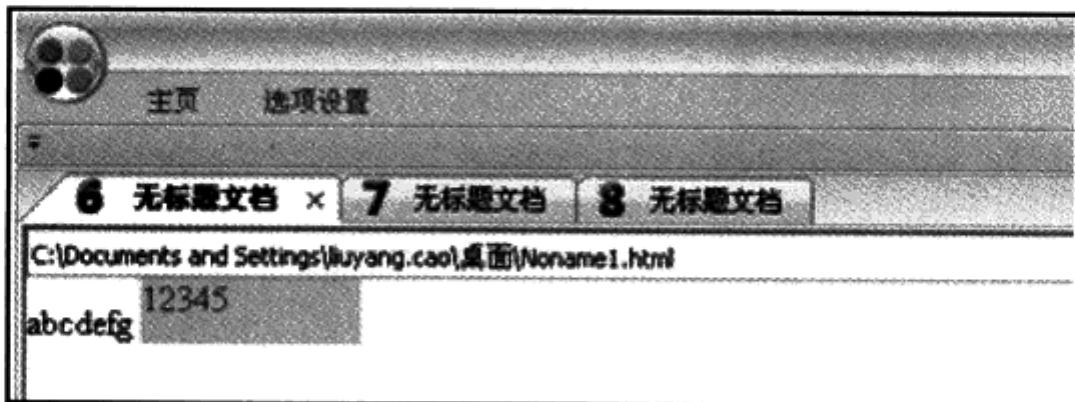


图 4-35 IE 6 下的截图



图 4-36 IE 7 下的截图

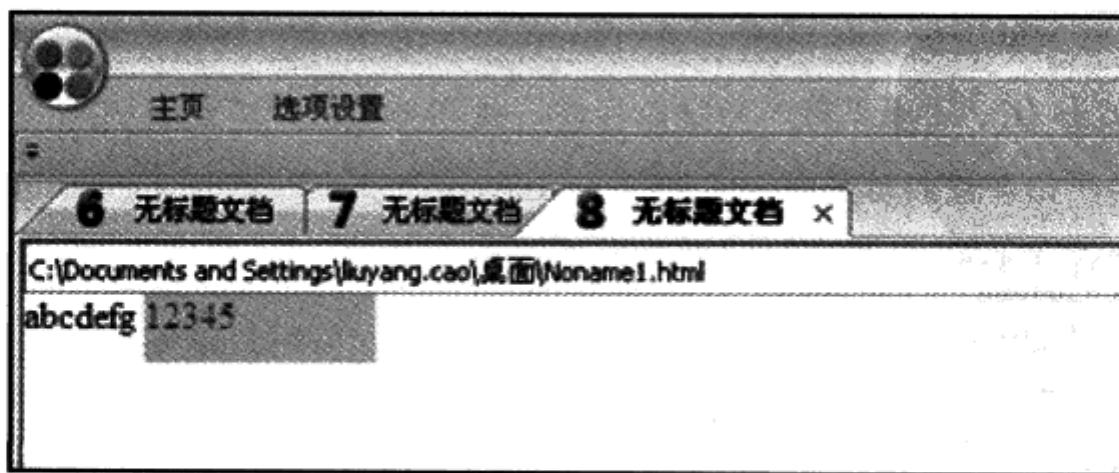


图 4-37 IE 8 下的截图

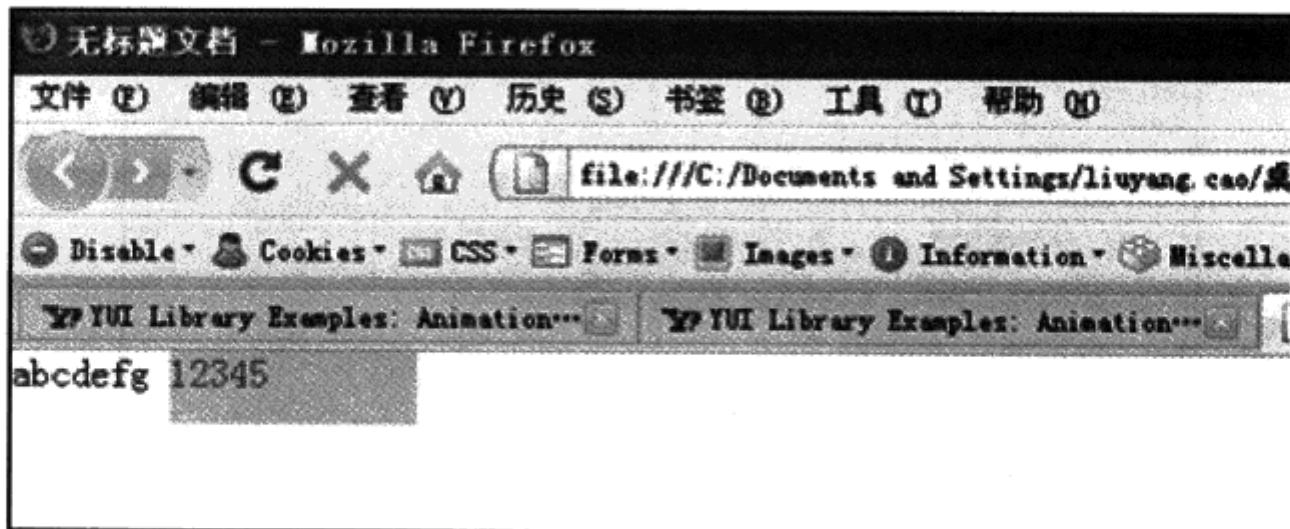


图 4-38 Firefox 下的截图

可以看到，IE 8 和 Firefox 正常显示，IE 6 和 IE 7 中 span 标签也有了长宽，并且显示在一行内。这不正是 display:inline-block 的表现吗？这里 display:inline-block 并不是真的在 IE 6 和 IE 7 下生效了，它们只是令 span 触发了 hasLayout 属性，其效果和设置 zoom:1 是一模一样的。在这里我们可以把 display:inline-block 改成 zoom:1，可以看到在 IE 6 和 IE 7 下效果仍然和现在一样，只是为了兼容 IE 8 和 Firefox，我们还是使用 display:inline-block。虽然 IE 6 和 IE 7 不支持 display:inline-block，但它可以触发 hasLayout。

但此时我们对 IE 6 和 IE 7 的 hack 还不完美，我们注意到左边的文字 abcdefg 和 span 是底对齐的，而 IE 8 和 Firefox 是顶对齐的。这个可以通过设置 span 的 vertical-align 解决。如代码清单 4-53 所示。

#### 代码清单 4-53 调整竖直排列的位置

---

```

<style type="text/CSS">
  span{color:red;width:100px;background:#ccc;height:30px;display:inline-
block;* vertical-align:-10px;}
</style>
<body>
  abcdefg <span>12345</span>
</body>

```

---

此时 IE 6、IE 7、IE 8 和 Firefox 下的截图分别如下。

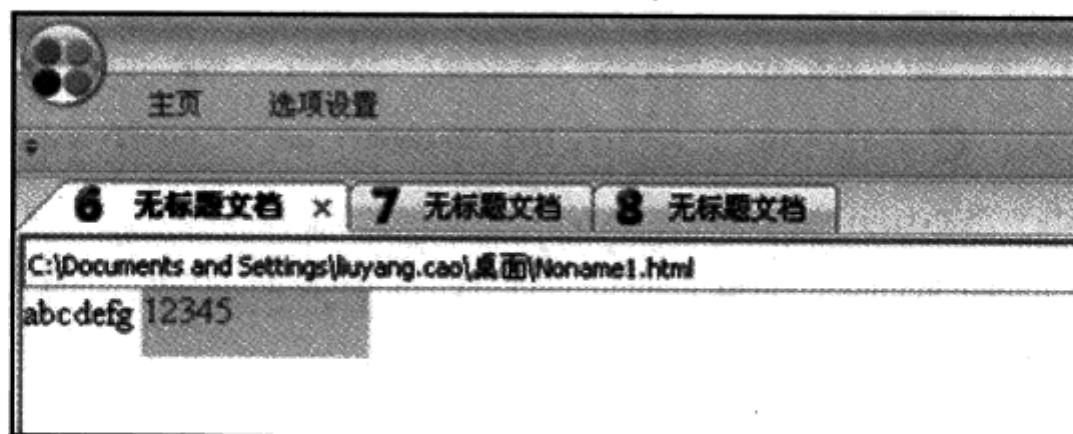


图 4-39 IE 6 下的截图

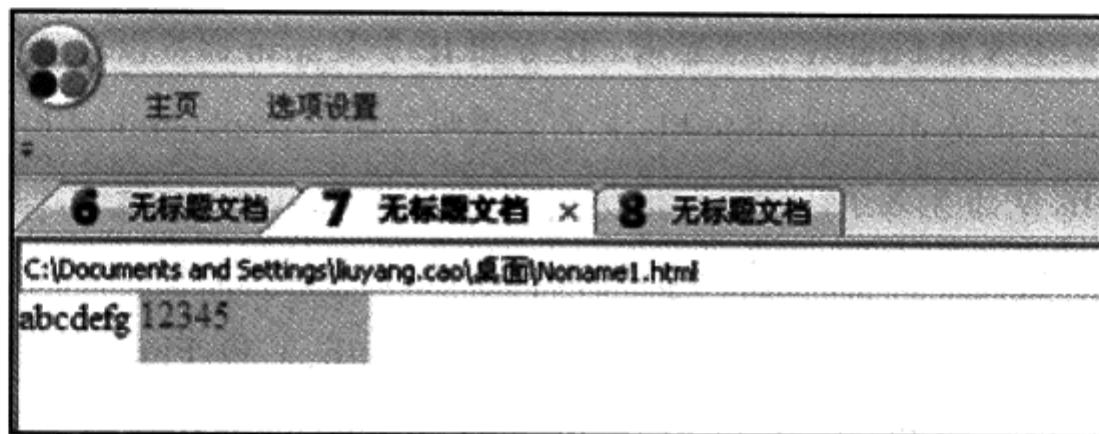


图 4-40 IE 7 下的截图

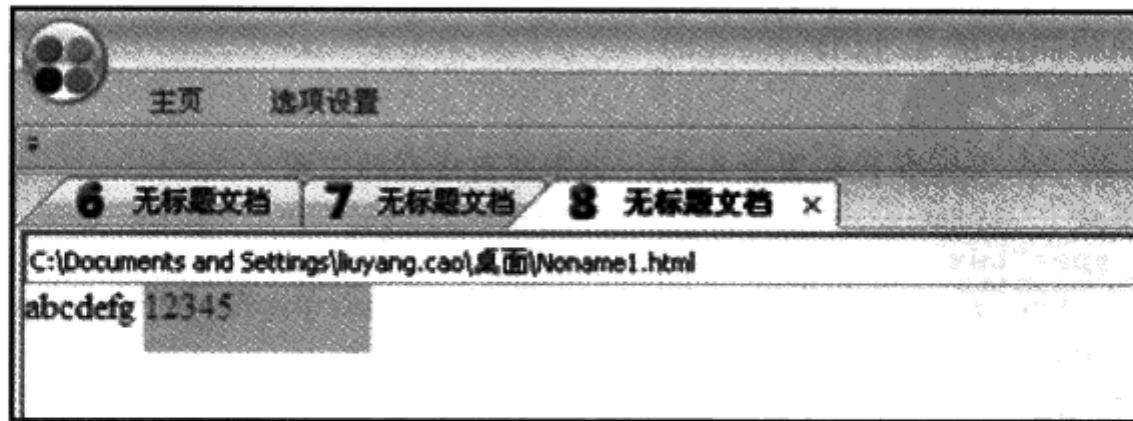


图 4-41 IE 8 下的截图

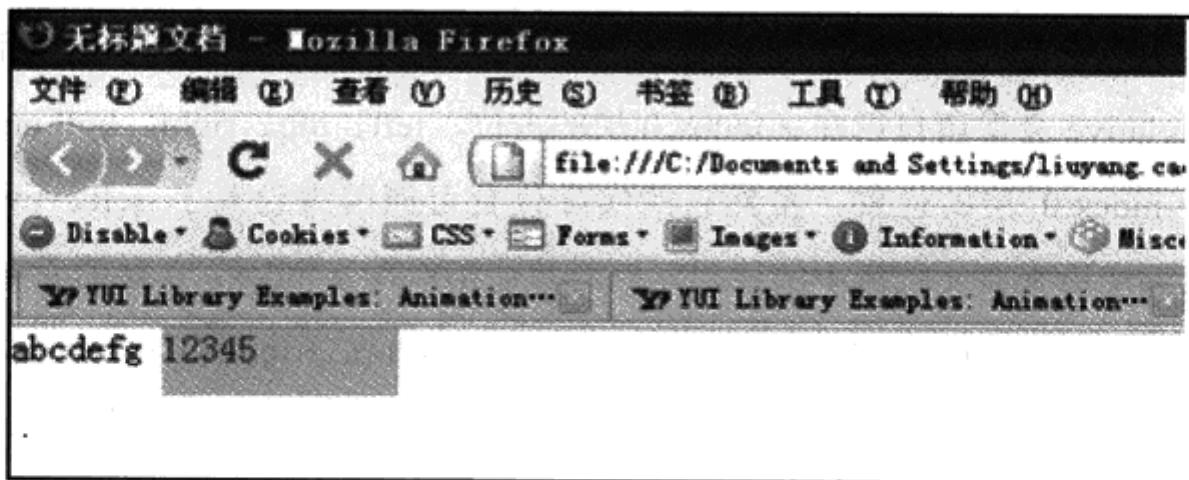


图 4-42 Firefox 下的截图

利用 hasLayout，可以在不支持 display:inline-block 的 IE 6 和 IE 7 下模拟出 display:inline-block 的效果，实现 IE 6、IE 7、IE 8 和 Firefox 都兼容的 display:inline-block 的应用，但它也存在一定的问题：1) 它只能对行内元素实现 display:inline-block；如果是块级元素就不行。所以，这就限制了我们能使用的标签只能是行内元素的标签。2) 这个方法中还用到了针对 IE 的 hack，\* vertical-align，这也是不好的，能不用 hack 就尽量不要使用 hack，所以这个方法并不值得推荐。可是实际项目中各种需求和情况都有可能发生，所以，如果需要用到 display:inline-block，可以通过这种方法实现。

虽然 IE 6 和 IE 7 不支持 CSS 设置为 display:inline-block，但事实上 IE 6 和 IE 7 的 CSS 解析引擎还是有 display:inline 的，比如说 img 标签和 button 标签都具有 display:inline-block 的特性，可以设置长宽但却并不独占一行。

#### 4.7.8 relative、absolute 和 float

position:relative 和 position:absolute 都可以改变元素在文档流中的位置，设置 position:relative 或 position:absolute 都可以让元素激活 left、top、right、bottom 和 z-index 属性（默认情况下，这些属性未激活，设置了也无效）。网页虽然看起来是平面的二维结构，但它其实是有 z 轴的，z 轴的大小由 z-index 控制，默认情况下，所有元素都是在 z-index:0 这一层的。元素根据自己的 display 类型、长宽、内外边距等属性顺序排列在 z-index: 0 这一层里，这就是文档流。设置 position:relative 或 position:absolute

会让元素“浮”起来，也就是 z-index 值大于 0，它会改变正常情况下的文档流。不同的是 position:relative 会保留自己在 z-index:0 层的占位，left、top、right、bottom 值是相对于自己在 z-index:0 层的位置，虽然它的实际位置可能因为设置了 left、top、right、bottom 值而偏离原来在文档流中的位置，但对其他仍然在 z-index:0 层的元素位置不会造成影响。而 position:absolute 会完全脱离文档流，不再在 z-index:0 层保留占位符，其 left、top、right、bottom 值是相对于自己最近的一个设置了 position:relative 或 position:absolute 的祖先元素的，如果祖先元素全都没有设置 position:relative 或 position:absolute，那么就相对于 body 元素。

除了 position:relative 和 position:absolute，float 也能改变文档流，不同的是，float 属性不会让元素“上浮”到另一个 z-index 层，它仍然让元素在 z-index:0 层排列，float 不像 position:relative 和 position:absolute 那样，它不能通过 left、top、right、bottom 属性精确地控制元素的坐标，它只能通过 float:left 和 float:right 来控制元素在同层里“左浮”和“右浮”。float 会改变正常的文档流排列，影响到周围元素。

另一个有趣的现象是 position:absolute 和 float 会隐式地改变 display 类型，不论之前什么类型的元素（display: none 除外），只要设置了 position:absolute、float:left 或 float:right 中任意一个，都会让元素以 display:inline-block 的方式显示：可以设置长宽，默认宽度并不占满父元素。就算我们显示地设置 display:inline 或者 display:block，也仍然无效（float 在 IE 6 下的双倍边距 bug 就是利用添加 display:inline 来解决的）。值得注意的是，position:relative 却不会隐式改变 display 的类型。

#### 4.7.9 居中

CSS 的居中会遇到很多种情况，不同的情况使用的方法不同。

##### 1. 水平居中

###### (1) 文本、图片等行内元素的水平居中

给父元素设置 text-align:center 可以实现文本、图片等行内元素的水平居中，如代码清单 4-54 所示。

#### 代码清单 4-54 行内元素的水平居中

```
<style type="text/CSS">
.wrap{background:#000; width:500px; height:100px; margin-bottom:10px;
color:#fff; text-align:center}
</style>

<div class="wrap">hello world</div>
<div class="wrap"></div>
```

效果如图 4-43 所示。



图 4-43 文本、图片等行内元素的水平居中

#### (2) 确定宽度的块级元素的水平居中

确定宽度的块级元素水平居中是通过设置 `margin-left:auto` 和 `margin-right:auto` 来实现的，如代码清单 4-55 所示。

#### 代码清单 4-55 确定宽度的块级元素的水平居中

```
<style type="text/CSS">
.wrap{background:#000; width:500px; height:100px}
.test{background:red; width:200px; height:50px; margin-left:auto;
margin-right:auto}
</style>

<div class="wrap"><div class="test"></div></div>
```

效果如图 4-44 所示。



图 4-44 文本、图片等行内元素的水平居中

### (3) 不确定宽度的块级元素的水平居中

不确定宽度的块级元素有三种方式可以实现居中。以分页模块为例，因为分页的数量是不确定的，所以我们不能通过设置宽度来限制它的弹性。方法一如代码清单 4-56 所示。

代码清单 4-56 不确定宽度的块级元素的水平居中方法一

```

<style type="text/CSS">
ul{list-style:none; margin:0; padding:0}
table{margin-left:auto; margin-right:auto;}
.test li{float:left; display:inline; margin-right:5px;}
.test a{float:left; width:20px; height:20px; text-align:center; line-height:20px; background:#316AC5; color:#fff; border:1px solid #316AC5; text-decoration:none;}
.test a:hover{background:#fff; color:#316AC5}
</style>

<div class="wrap">
    <table><tbody><tr><td>
        <ul class="test">
            <li><a href="#">1</a></li>
        </ul>
    </td></tr></tbody></table>
    <table><tbody><tr><td>
        <ul class="test">
            <li><a href="#">1</a></li>
            <li><a href="#">2</a></li>
            <li><a href="#">3</a></li>
        </ul>
    </td></tr></tbody></table>
    <table><tbody><tr><td>
        <ul class="test">
            <li><a href="#">1</a></li>
            <li><a href="#">2</a></li>
            <li><a href="#">3</a></li>
            <li><a href="#">4</a></li>
            <li><a href="#">5</a></li>
        </ul>
    </td></tr></tbody></table>

```

```

</td></tr></tbody></table>
</div>

```

效果如图 4-45 所示。

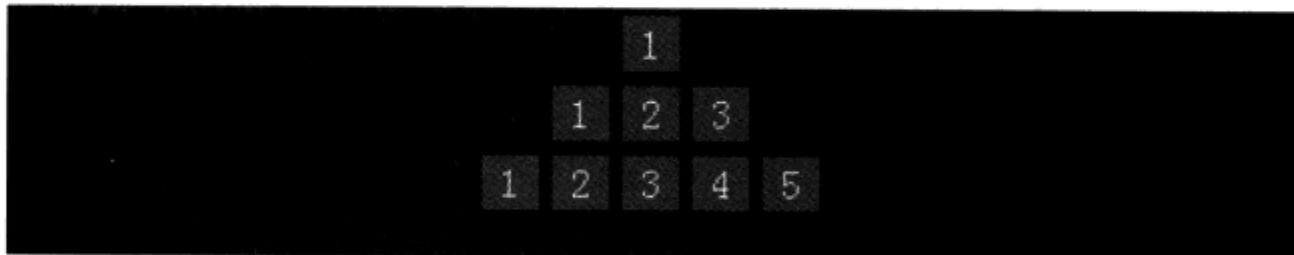


图 4-45 不确定宽度的块级元素的水平居中效果图

方法一演示了有一个分页，三个分页和五个分页的情况下，分页模块都能居中的示例。`ul` 标签是整个分页模块的根元素，它包含的 `li` 数量不确定，所以 `ul` 本身的宽度也没办法确定，不能通过设置固定宽度，`margin-left:auto` 和 `margin-right:auto` 来实现居中。这里用到了一个有趣 的标签 `table` 来帮助实现了不确定宽度的块级元素的水平居中，`table` 有趣的地方在于它本身并不是块级元素，如果不给它设定宽度的话，它的宽度由内部元素的宽度“撑起”，但即使不设定它的宽度，仅设置 `margin-left:auto` 和 `margin-right:auto` 就可以实现水平居中！将 `ul` 包含在 `table` 标签内，对 `table` 设置 `margin-left:auto` 和 `margin-right:auto` 就可以使 `table` 水平居中，间接使 `ul` 实现了水平居中。

这种做法很巧妙，但缺点是增加了无语义标签，加深了标签的嵌套层数。

方法二如代码清单 4-57 所示。

#### 代码清单 4-57 不确定宽度的块级元素的水平居中方法二

```

<style type="text/CSS">
ul{list-style:none;margin:0;padding:0}
.wrap{background:#000; width:500px; height:100px}
.test{text-align:center;padding:5px;}
.test li{display:inline;}
.test a{padding:2px 6px;background:#316AC5;color:#fff;border:1px solid #316AC5;text-decoration:none;}
.test a:hover{background:#fff;color:#316AC5}
</style>

<div class="wrap">

```

```

<ul class="test">
    <li><a href="#">1</a></li>
</ul>
<ul class="test">
    <li><a href="#">1</a></li>
    <li><a href="#">2</a></li>
    <li><a href="#">3</a></li>
</ul>
<ul class="test">
    <li><a href="#">1</a></li>
    <li><a href="#">2</a></li>
    <li><a href="#">3</a></li>
    <li><a href="#">4</a></li>
    <li><a href="#">5</a></li>
</ul>
</div>

```

效果如图 4-45 所示。

方法二换了种思路, 改变块级元素的 display 为 inline 类型, 然后使用 text-align:center 来实现居中。相较于方法一, 它的好处是不用增加无语义标签, 简化了标签的嵌套深度, 但它也存在一定问题: 它将块级元素的 display 类型改为 inline, 变成了行内元素, 而行内元素比起块级元素缺少一些功能, 比如设定长宽值, 在某些特殊需求的 CSS 设置中, 这种方法可能会带来一些限制。

方法三如代码清单 4-58 所示。

#### 代码清单 4-58 不确定宽度的块级元素的水平居中方法三

```

<style type="text/CSS">
ul{list-style:none; margin:0; padding:0}
.wrap{background:#000; width:500px; height:100px}
.test{clear:both; padding-top:5px; float:left; position:relative;
left:50%;}
.test li{float:left; display:inline; margin-right:5px; position:relative;
left:-50%;}
.test a{float:left; width:20px; height:20px; text-align:center; line-
height:20px; background:#316AC5; color:#fff; border:1px solid #316AC5;
text-decoration:none;}
.test a:hover{background:#fff; color:#316AC5}
</style>

<div class="wrap">
<ul class="test">

```

```

<li><a href="#">1</a></li>
</ul>
<ul class="test">
    <li><a href="#">1</a></li>
    <li><a href="#">2</a></li>
    <li><a href="#">3</a></li>
</ul>
<ul class="test">
    <li><a href="#">1</a></li>
    <li><a href="#">2</a></li>
    <li><a href="#">3</a></li>
    <li><a href="#">4</a></li>
    <li><a href="#">5</a></li>
</ul>
</div>

```

效果如图 4-45 所示。

方法三通过给父元素设置 float，然后父元素设置 position:relative 和 left:50%，子元素设置 position:relative 和 left:-50% 来实现水平居中。它可以保留块级元素仍以 display:block 的形式显示，而且不会添加无语义标签，不增加嵌套深度，但它的缺点是设置了 position:relative，带来了一定的副作用。

这三种方法使用得都非常广泛，各有优缺点，具体选用哪种方式可以视具体情况而定。

## 2. 竖直居中

### (1) 父元素高度不确定的文本、图片、块级元素的竖直居中

父元素高度不确定的文本、图片、块级元素的竖直居中是通过给父容器设置相同上下边距实现的，如代码清单 4-59 所示。

**代码清单 4-59 父元素高度不确定的文本、图片、块级元素的竖直居中**

```

<style type="text/CSS">
    .wrap{background:#000; width:500px; color:#fff; margin-bottom:10px;
padding-top:20px; padding-bottom:20px}
    .test{width:200px;height:50px;background:red;}
</style>

<div class="wrap">hello world</div>
<div class="wrap"></div>
<div class="wrap"><div class="test"></div></div>

```

效果如图 4-46 所示。

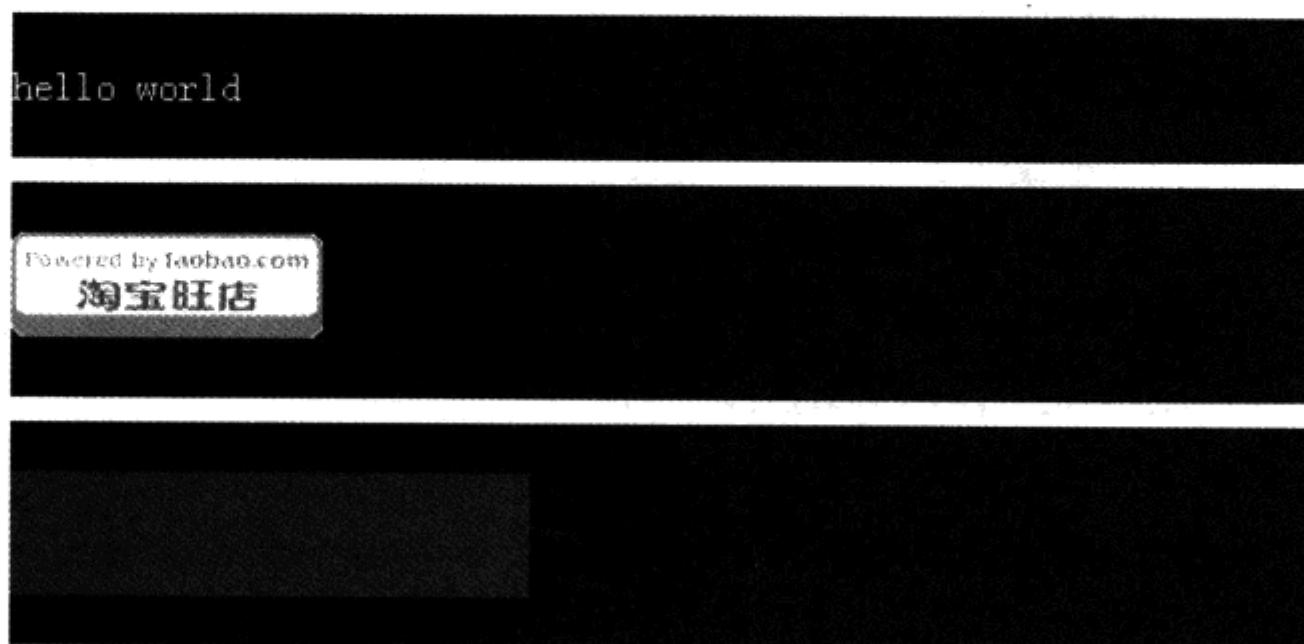


图 4-46 父元素高度不确定的文本、图片、块级元素的竖直居中

## (2) 父元素高度确定的单行文本的竖直居中

父元素高度确定的单行文本的竖直居中，是通过给父元素设置 `line-height` 来实现的，`line-height` 值和父元素的高度值相同，如代码清单 4-60 所示。

代码清单 4-60 父元素高度确定的单行文本的竖直居中

```
<style type="text/CSS">
    .wrap{background:#000;      width:500px;      color:#fff;height:100px;line-
height:100px;}
</style>

<div class="wrap">hello world</div>
```

效果如图 4-47 所示。



图 4-47 父元素高度确定的单行文本的竖直居中

### (3) 父元素高度确定的多行文本、图片、块级元素的竖直居中

父元素高度确定的多行文本、图片、块级元素的竖直居中有两种方法。

方法一：说到竖直居中，CSS 中有一个用于竖直居中的属性 `vertical-align`，但只有当父元素为 `td` 或者 `th` 时，这个 `vertical-align` 属性才会生效，对于其他块级元素，例如 `div`、`p` 等，默认情况下是不支持 `vertical-align` 属性的。在 Firefox 和 IE 8 下，可以设置块级元素的 `display` 类型为 `table-cell`，激活 `vertical-align` 属性，但 IE 6 和 IE 7 并不支持 `display:table-cell`，所以这种方法没办法跨浏览器兼容。但我们可以使用最原始的笨方法来实现兼容——既然不支持块级元素设置为 `display:table-cell` 来模拟表格，那么，我们就直接使用表格好了。如代码清单 4-61 所示。其效果如图 4-46 所示。

代码清单 4-61 父元素高度确定的多行文本、图片、块级元素的竖直居中方法一

---

```

<style type="text/CSS">
.wrap{background:#000; width:500px; color:#fff; height:100px}
.test{width:200px; height:50px; background:red;}
</style>

<table><tbody><tr><td class="wrap">
hello world<br />
hello world<br />
hello world
</td></tr></tbody></table>

<table><tbody><tr><td class="wrap">

</td></tr></tbody></table>

<table><tbody><tr><td class="wrap">
<div class="test"></div>
</td></tr></tbody></table>

```

---

因为 `td` 标签默认情况下就隐式地设置了 `vertical-align` 的值为 `middle`，所以我们不需要再显式地设置一遍。

方法一可以很好地实现竖直居中效果，且不会带来任何样式上的副作用，但它添加了无语义标签，并增加了嵌套深度。

方法二：对支持 `display:table-cell` 的 IE 8 和 Firefox 用 `display:table-cell` 和 `vertical-align:middle` 来实现居中，对不支持 `display:table-cell` 的 IE 6 和 IE 7，使用特定格式的 hack，如代码清单 4-62 所示。其效果如图 4-46 所示。

## 代码清单 4-62 父元素高度确定的多行文本、图片、块级元素的竖直居中方法二

```

<style type="text/CSS">
    .mb10{margin-bottom:10px}
    .wrap{background:#000; width:500px; color:#fff; margin-
bottom:10px;height:100px;display:table-cell;vertical-
align:middle;*position:relative}
    .test{width:200px;height:50px;background:red}
    .verticalWrap{*position:absolute;*top:50%}
    .vertical{*position:relative;*top:-50%}
</style>
<div class="mb10">
    <div class="wrap">
        <div class="verticalWrap">
            <div class="vertical">
                hello world<br />
                hello world<br />
                hello world
            </div>
        </div>
    </div>
</div>
<div class="mb10">
    <div class="wrap">
        <div class="verticalWrap">
            
        </div>
    </div>
</div>
<div class="mb10">
    <div class="wrap">
        <div class="verticalWrap">
            <div class="test vertical"></div>
        </div>
    </div>
</div>

```

方法二利用 hack 技术区别对待 Firefox、IE 8 和 IE 6、IE 7，在不支持 `display:table-cell` 的 IE 6 和 IE 7 下，通过给父子两层元素分别设置 `top:50%` 和 `top:-50%` 来实现居中。这种方法的好处是没有增加额外的标签，但它的缺点也很明显，一方面它使用了 hack，不利于维护，另一方面，它需要设置 `position:relative` 和 `position:absolute`，带来了副作用。

#### 4.7.10 网格布局

图 4-48 是个典型的两栏式简单布局。

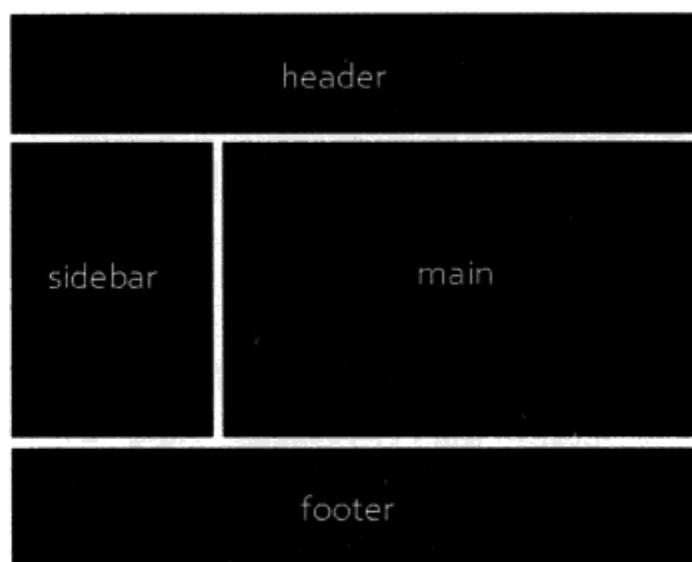


图 4-48 简单的两栏式布局

我们可以用代码清单 4-63 所示代码来实现它。

#### 代码清单 4-63 两栏式布局方案一

---

```

<style type="text/CSS">
.header{}
.footer{clear:both}
.sidebar{width:25%; float:left}
.main{width:70%; float:right}
</style>

<div class="header"></div>
<div class="content">
    <div class="sidebar"></div>
    <div class="main"></div>
</div>
<div class="footer"></div>

```

---

这么做可以实现我们预期的效果，但并不理想。和 main 相比，一般来说，sidebar 里都是一些附属内容，main 里才是网页的主体内容。如果网速很慢，网页加载时间很长，代码清单 4-63 中的代码会让我们在漫长的等待过程中先看到不太重要的 sidebar 的内容，之后才会看到网页的主体内容。

我们可以对它改进一下，调换 main 和 sidebar 的标签的顺序，如代码清单 4-64 所示。

#### 代码清单 4-64 两栏式布局方案二

---

```

<style type="text/CSS">
.header{}
.footer{clear:both}
.sidebar{width:25%; float:left}

```

---

```
.main{width:70%; float:right}  
/>  
  
<div class="header"></div>  
<div class="content">  
    <div class="main"></div>  
    <div class="sidebar"></div>  
</div>  
<div class="footer"></div>
```

如此一来，就可以保证先加载 main，后加载 sidebar。注意：main 的内容比起 sidebar 更重要，无论 sidebar 和 main 在样式上谁左谁右，在 html 标签上要保证 main 的标签在 sidebar 之前被加载。

那么现在的方案足够好吗？未必。如果设计图有修改呢？修改后的设计图如图 4-49 所示，sidebar 和 main 的位置调换了。

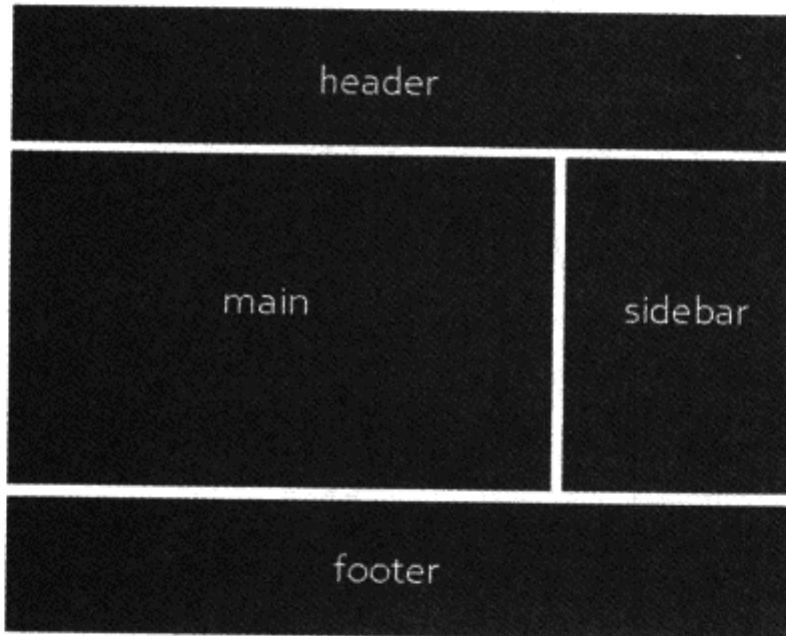


图 4-49 调换 main 和 sidebar 的位置

我们需要对代码清单 4-64 进行修改，如代码清单 4-65 所示。

#### 代码清单 4-65 调换左右两栏的位置

```
<style type="text/CSS">  
.header{}  
.footer{clear:both}  
.sidebar{width:25%; float:right}  
.main{width:70%; float:left}  
/>
```

```
<div class="header"></div>
<div class="content">
    <div class="main"></div>
    <div class="sidebar"></div>
</div>
<div class="footer"></div>
```

这种修改成本还算好，并不太麻烦。但如果设计图又被修改，如图 4-50 所示。



图 4-50 新增一对 sidebar 和 main

之前的代码就又不适用了，将浮动和宽度绑定在同个类中，会限制类的灵活性，我们又得重新修改样式。浮动方向是个不稳定的因素，它很可能会变化，在 4.4.3 节中我们谈到过“多用组合”原则，如果一个类中有些部分会经常变化，我们可以将这个经常变化的部分抽离出来单独设成一个类，然后用类的组合来实现最终样式。按照这种思路我们重新编写图 4-48 的代码，如代码清单 4-66 所示。

代码清单 4-66 用类的组合的方式设定浮动方向

```
<style type="text/CSS">
.f1{float:left}
.fr{float:right}
```

```

.header{}
.footer{clear:both}
.sidebar{width:25%}
.main{width:70%}
</style>

<div class="header"></div>
<div class="content">
    <div class="main fr"></div>
    <div class="sidebar fl"></div>
</div>
<div class="footer"></div>

```

看看组合的威力吧，如果设计图的修改如图 4-49 所示，我们的代码如何应对呢？如代码清单 4-67 所示。

**代码清单 4-67 用类的组合的方式面对简单修改**

```

<style type="text/CSS">
.fl{float:left}
.fr{float:right}
.header{}
.footer{clear:both}
.sidebar{width:25%}
.main{width:70%}
</style>

<div class="header"></div>
<div class="content">
    <div class="main fl"></div>
    <div class="sidebar fr"></div>
</div>
<div class="footer"></div>

```

设计图修改成如图 4-50 所示呢？代码修改如代码清单 4-68 所示。

**代码清单 4-68 用类的组合的方式面对复杂修改**

```

<style type="text/CSS">
.fl{float:left}
.fr{float:right}
.content{clear:both}
.header{}
.footer{clear:both}
.sidebar{width:25%}
.main{width:70%}
</style>

<div class="header"></div>
<div class="content">
    <div class="main fl"></div>

```

---

```

<div class="sidebar fr"></div>
</div>
<div class="content">
    <div class="main fr"></div>
    <div class="sidebar fl"></div>
</div>
<div class="footer"></div>

```

---

浮动从 main 和 sidebar 中抽离出来，main 和 sidebar 的灵活性增加了，提高了类的重用性。面对设计的修改，我们只需简单地修改一下相关类的挂钩(fl、fr)就可从容应对！

但即使如此，这仍然不是最佳方案。到目前为止，我们考虑的都是 main 和 sidebar 宽度固定的情况，如果 main 和 sidebar 的宽度也发生变化呢？设计图修改成图 4-51 呢？



图 4-51 两个 sidebar 和 main 的宽度不同

图 4-51 中上下两个 main 和 sidebar 的宽度不同。对于这种情况，我们该如何处理呢？按照之前的思路，如果宽度也是个容易变化的因素，那么将它也抽离出来，单独设一个类，如代码清单 4-69 所示。

#### 代码清单 4-69 将宽度提取出来

---

```

<style type="text/CSS">
.fl{float:left}

```

```

.fr{float:right}
.content{clear:both}
.header{}
.footer{clear:both}
.sidebar{}
.main{}
.w25{width:25%}
.w70{width:70%}
.w35{width:35%}
.w60{width:60%}
</style>

<div class="header"></div>
<div class="content">
    <div class="main fl w70"></div>
    <div class="sidebar fr w25"></div>
</div>
<div class="content">
    <div class="main fr w60"></div>
    <div class="sidebar fl w35"></div>
</div>
<div class="footer"></div>

```

这种思路可以解决我们遇到的问题，而且具有非常高的灵活性。但它也有不足之处，如果要修改，需要成对地进行修改，例如 `main` 和 `sidebar` 要对调位置，需要同时修改 `main` 和 `sidebar` 两者挂的 `fl`、`fr` 类。

有另一种思路可以帮助我们解决这个问题——利用子选择器。如代码清单 4-70 所示：

代码清单 4-70 用子选择器应对复杂变化

```

<style type="text/CSS">
.content{clear:both}
.header{}
.footer{clear:both}
.main{}
.sidebar{}
.content-lr-7025 .main{float:left; width:70%}
.content-lr-7025 .sidebar{float:right; width:25%}
.content-rl-7025 .main{float:right; width:70%}
.content-rl-7025 .sidebar{float:left; width:25%}
.content-lr-6035 .main{float:left; width:60%}
.content-lr-6035 .sidebar{float:right; width:35%}
.content-rl-6035 .main{float:right; width:60%}
.content-rl-6035 .sidebar{float:left; width:35%}
</style>

<div class="header"></div>
<div class="content content-lr-7025">
```

```
<div class="main"></div>
<div class="sidebar"></div>
</div>
<div class="content content-rl-6035">
    <div class="main"></div>
    <div class="sidebar"></div>
</div>
<div class="footer"></div>
```

利用子选择器，我们可以削弱 `main` 和 `sidebar` 作为样式挂钩的能力，单独的 `main` 和 `sidebar` 选择符就像是一个抽象类，我们可以在其中定义一些公共的样式。用于布局的浮动和宽度属性，不由 `main` 和 `sidebar` 这两个抽象类直接提供，而由它们的衍生类——`content-xx-xxxx .main` 和 `content-xx-xxxx .sidebar` 提供。如此一来，其实在样式的控制上，由原来的一层拆分成了两层，`main` 和 `sidebar` 的责任削弱了，它们只用关心自己的位置（`main` 的 `html` 要位于 `sidebar` 前面）和自己内部的样式（例如 `color`、`background`、`font-size` 等），不用再关心布局，而将布局的控制交给了它们的父元素，由 `content-xx-xxxx` 来决定布局的方式。其中 `xx` 代表浮动，前面一位代表 `main` 的浮动方向，后面一位代表 `sidebar` 的浮动方向，如果 `main` 向左浮，`sidebar` 向右浮，`xx` 为 `lr`，反之为 `rl`；`xxxx` 代表宽度，前两位代表 `main` 的宽度，后两位代表 `sidebar` 的宽度，如果 `main` 的宽度为 70%，`sidebar` 的宽度为 25%，那么 `xxxx` 为 `7025`。这就是利用子选择器实现布局控制的大体思路，这里 `content-xx-xxxx` 的形式是笔者个人比较喜欢的，当然具体形式可以根据个人喜好决定，例如 `layout-xxxx-xx` 也不错。

代码清单 4-68 和代码清单 4-70 分别代表了组合类和子选择器这两种不同思路的布局方式。如果要对样式做修改，比如对调 `main` 和 `sidebar` 的位置，用组合类的方式，我们需要同时修改 `main` 和 `sidebar` 的 `float` 挂钩，而用子选择器方式，我们只需要更改 `xx` 的值；如果需要修改 `main` 和 `sidebar` 的宽度，用组合类的方式，我们需要同时修改 `main` 和 `sidebar` 的宽度挂钩，而用子选择器方式，我们只需要更改 `xxxx` 的值。比较而言，子选择器方式更便于修改。

但子选择器方式便于修改是通过预设大量分支实现的，不够轻便。比如，如果我们新增一种 `main` 宽度 80%，`sidebar` 宽度 15% 的布局，用组合类的方式，代码见代码清单 4-71。

## 代码清单 4-71 组合类的方式面对扩展

```

<style type="text/css">
.fl{float:left}
.fr{float:right}
.content{clear:both}
.header{}
.footer{clear:both}
.sidebar{}
.main{}
.w25{width:25%}
.w70{width:70%}
.w35{width:35%}
.w60{width:60%}
.w80{width:80%}
.w15{width:15%}
</style>

<div class="header"></div>
<div class="content">
    <div class="main fl w70"></div>
    <div class="sidebar fr w25"></div>
</div>
<div class="content">
    <div class="main fr w60"></div>
    <div class="sidebar fl w35"></div>
</div>
<div class="content">
    <div class="main fl w80"></div>
    <div class="sidebar fr w15"></div>
</div>
<div class="footer"></div>

```

用子选择器的方式，代码见代码清单 4-72。

## 代码清单 4-72 子选择器方式面对扩展

```

<style type="text/css">
.content{clear:both}
.header{}
.footer{clear:both}
.main{}
.sidebar{}
.content-lr-7025 .main{float:left; width:70%}
.content-lr-7025 .sidebar{float:right; width:25%}
.content-rl-7025 .main{float:right; width:70%}
.content-rl-7025 .sidebar{float:left; width:25%}
.content-lr-6035 .main{float:left; width:60%}
.content-lr-6035 .sidebar{float:right; width:35%}
.content-rl-6035 .main{float:right; width:60%}
.content-rl-6035 .sidebar{float:left; width:35%}
.content-lr-8015 .main{float:left; width:80%}

```

```

.content-lr-8015 .sidebar{float:right; width:15%}
.content-rl-8015 .main{float:right; width:80%}
.content-rl-8015 .sidebar{float:left; width:15%}

</style>

<div class="header"></div>
<div class="content content-lr-7025">
    <div class="main"></div>
    <div class="sidebar"></div>
</div>
<div class="content content-rl-6035">
    <div class="main"></div>
    <div class="sidebar"></div>
</div>
<div class="content content-lr-8015">
    <div class="main"></div>
    <div class="sidebar"></div>
</div>
<div class="footer"></div>

```

可以看出，组合类的方式比子选择符方式更易于扩展，更轻便。是不是组合类的方式更好呢？一般情况下，扩展性是好的，是需要尽量提供的，但扩展性太好并不一定有利于维护。对某些经常使用的功能，而且可能是让多人团队合作的，统一的接口、统一的格式可以避免实现方案五花八门，可以站在全局的角度严格控制。一般来说，网站的 `main` 和 `sidebar` 的宽度是有严格限制的，大概只有几种类型，它们需要统一的管理和严格的控制。`content-xx-xxxx` 限制了类的扩展性，只有通过它才能完成布局的功能，它就像是面向对象编程中的接口，每个衍生类都必须实现这个接口。接口最大的作用是统一了格式，它是带有强制性的，不按照预设的统一格式编码，是无法顺利运行的，比如说我们只设置了`.content-lr-7025`，我们就不能使用`.content-lr-6035`。所以，虽然两种方式都各有好坏，但笔者认为对于布局来说，还是子选择器的方式更适合。

`content-xx-xxxx` 的宽度设置 `xxxx` 的具体单位最好使用百分比这种相对单位，而不要使用 `px`。因为使用百分比，可以嵌套使用，提高重用性，如代码清单 4-73 所示。

代码清单 4-73 子选择器布局的嵌套使用

```

<style type="text/CSS">
.content{clear:both}
.header{}
.footer{clear:both}

```

```

.main{}
.sidebar{}
.content-lr-7025 .main{float:left; width:70%}
.content-lr-7025 .sidebar{float:right; width:25%}
.content-rl-7025 .main{float:right; width:70%}
.content-rl-7025 .sidebar{float:left; width:25%}
</style>

<div class="header"></div>
<div class="content content-lr-7025">
    <div class="main content-lr-7025">
        <div class="main"></div>
        <div class="sidebar"></div>
    </div>
    <div class="sidebar"></div>
</div>
<div class="footer"></div>

```

---

每个 main 和 sidebar 本身又可以包含 main 和 sidebar，只要挂上 content-xx-xxxx 就可以控制自己包含的 main 和 sidebar 的布局，一层套一层，只在最外层的容器给定具体宽度，所有其他容器的宽度均用百分比设置。这种方式的布局极具灵活性，又因为 content-xx-xxxx 是全站统一管理的，所以非常稳定，它有一个专有的名称，叫做网格布局。

#### 4.7.11 z-index 的相关问题以及 Flash 和 IE 6 下的 select 元素

在 4.7.8 节中，我们讲到网页其实是三围结构的，除了 x、y 轴，它还有 z 轴。z 轴在元素设置 position 为 absolute 或 relative 后被激活，其大小由 z-index 设置，z-index 越大，元素位置越靠上。如代码清单 4-74 和代码清单 4-75 所示。

代码清单 4-74 z-index 设置高低

```

<style type="text/CSS">
    #one{width:300px;height:300px; background:black}
    #two{width:100px;height:100px; background:red;position:absolute;
z-index:1; left:100px; top:250px}
    #three{width:100px;height:100px; background:green; position:relative;
z-index:2; left:120px; top:-100px}
</style>

<div id="one"></div>
<div id="two"></div>
<div id="three"></div>

```

---

效果如图 4-52 所示。

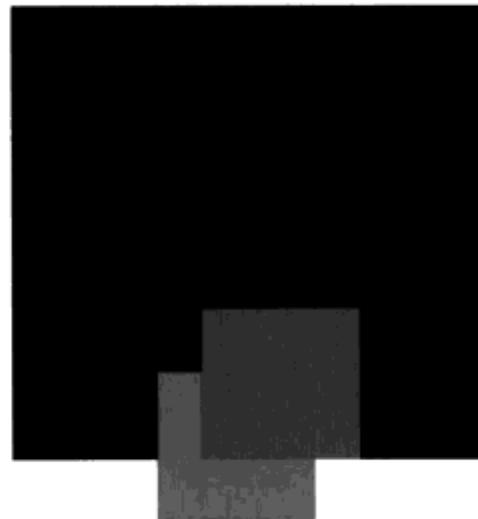


图 4-52 z-index 效果图 1

#### 代码清单 4-75 调整 z-index 的高低

```
<style type="text/CSS">
  #one{width:300px; height:300px; background:black}
  #two{width:100px; height:100px; background:red; position:absolute;
z-index:3; left:100px; top:250px}
  #three{width:100px; height:100px; background:green; position:relative;
z-index:2; left:120px; top:-100px}
</style>

<div id="one"></div>
<div id="two"></div>
<div id="three"></div>
```

效果如图 4-53 所示。

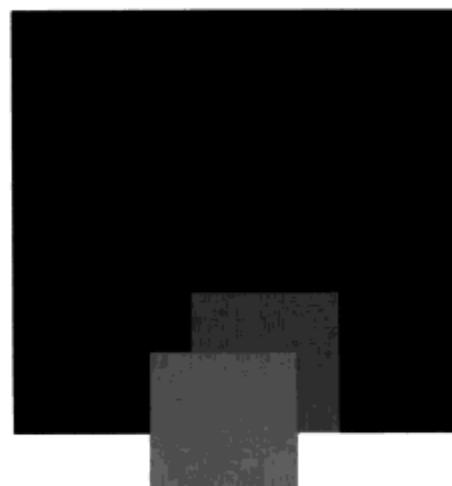


图 4-53 z-index 效果图 2

值得注意的是，z-index 值也可以为负数，如代码清单 4-76 所示。

代码清单 4-76 z-index 设为负数

```
<style type="text/CSS">
  #one{width:300px;height:300px; background:black}
  #two{width:100px;height:100px; background:red;position:absolute;
z-index:-1; left:100px; top:250px}
  #three{width:100px;height:100px; background:green; position:relative;
z-index:2; left:120px; top:-100px}
</style>

<div id="one"></div>
<div id="two"></div>
<div id="three"></div>
```

效果如图 4-54 所示。

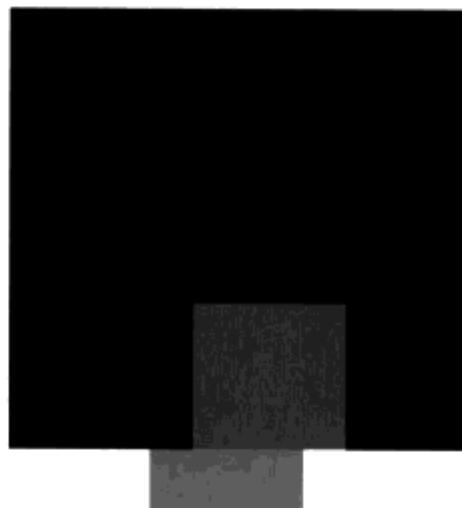


图 4-54 z-index 效果图 3

z-index 为负值，可以创造一些特殊的效果，但也会带来一些麻烦，比如，我们对 id 分别为 one、two、three 的三个 div 监听 click 事件，代码如代码清单 4-77 所示。

代码清单 4-77 为 one、two、three 监听 click 事件

```
<script type="text/JavaScript">
  var one = document.getElementById("one"), two =
document.getElementById("two"), three = document.getElementById("three");
  one.onclick = function(){
    alert("one");
  }
  two.onclick = function(){
    alert("two");
  }
```

---

```
three.onclick = function() {
    alert("three");
}
```

---

可以看到，点击黑色区域和绿色区域，都可以顺利地弹出提示信息，但点击红色区域，却无法弹出提示信息。这是因为红色区域的 `z-index` 为负数，位于 `z-index` 为 0 的 `body` 之下，被透明的 `body` 挡住了。

如果我们不设置 `z-index` 值，那么默认 `z-index` 为 0，但它仍然会浮于 `body` 之上。如果多个元素的 `z-index` 值相同，那么 `HTML` 标签中后出现的元素，会浮在先出现的元素之上，如代码清单 4-78 所示。

代码清单 4-78 `z-index` 值相同

---

```
<style type="text/CSS">
#two{width:100px;height:100px; background:red;position:absolute;
z-index:2; left:100px; top:250px}
#three{width:100px;height:100px; background:green; position:relative;
z-index:2; left:120px; top:-100px}
</style>
<div id="two"></div>
<div id="three"></div>
```

---

`id` 为 `two` 和 `three` 的元素，`z-index` 值都为 2，但因为 `<div id="three"></div>` 出现在 `<div id="two"></div>` 之后，所以 `id` 为 `three` 的元素会浮于 `two` 之上。

说到多个元素位置发生重叠，除了设置 `position` 为 `relative` 或 `absolute` 激活 `z-index` 外，还有一种特殊情况，它并未激活 `z-index`，但仍然让元素位置发生了重叠——负边距。如代码清单 4-79 和代码清单 4-80 所示。

代码清单 4-79 负边距引起的竖直重叠

---

```
<style type="text/CSS">
#one{width:300px; height:300px; background:black}
#two{width:100px; height:100px; background:red; margin-top:-50px}
</style>

<div id="one"></div>
<div id="two"></div>
```

---

效果如图 4-55 所示。

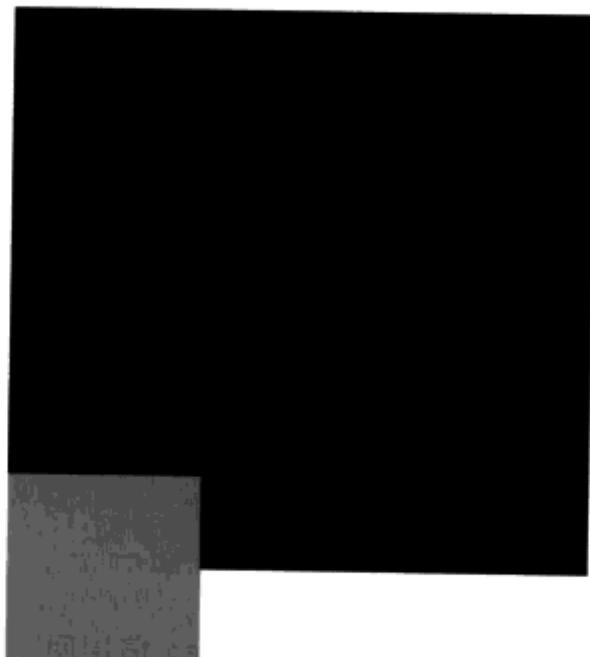


图 4-55 负边距引起的竖直位置重叠

#### 代码清单 4-80 负边距引起的水平重叠

```
<style type="text/CSS">
  #one{width:300px; height:300px; background:black; float:left}
  #two{width:100px; height:100px; background:red; display:inline; float:left;
margin-left:-50px;}
</style>

<div id="one"></div>
<div id="two"></div>
```

效果如图 4-56 所示。



图 4-56 负边距引起的水平位置重叠

设置负边距可以让相邻元素的位置产生重叠，谁浮在上面，取决于HTML标签出现的先后，后出现的标签浮于先出现的标签之上。

关于z-index，有个不得不提到的问题，就是Flash和IE 6下select元素的显示问题。Flash插入网页中，如果和其他元素有重叠，无论我们如何设置z-index，Flash都会浮在其他元素之上。这是为什么呢？其实与z-index无关，浏览器解析页面时，会先判断元素的类型，如果是窗口类型的，会优先于非窗口类型的元素，显示在页面最顶端，如果同属非窗口类型的，才会去判断z-index的大小。

Flash嵌入网页中，有个wmode属性，用于指定窗口模式，其值有window(窗口)、opaque(非窗口不透明)、transparent(非窗口透明)三种。其中window表示Flash以窗口形式显示，opaque和transparent表示Flash以非窗口形式显示，如果不显示设置wmode属性，默认wmode的值为window。所以，如果我们没有设置wmode值，或者设置wmode为widnow的话，Flash是以窗口类型显示的，其z轴的优先级是高于所有非窗口类型的元素的，无论我们如何设置z-index也不会有效果。解决的办法就是设置wmode属性为opaque或transpartent。因为Flash在IE和Firefox下是用不同的标签嵌入的(IE下以object标签嵌入，Firefox下以embed标签嵌入)，所以我们需要对两种标签分别进行设置，如代码清单4-81所示。

#### 代码清单4-81 设置flash的显示模式

---

```
<object width="640" height="90" type="application/x-shockwave-Flash"
classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000">
  <param value="xxxxxx.swf" name="movie"></param>
  <param value="opaque" name="wmode"></param>
  <embed width="640" height="90" wmode="opaque" type="application/x-
shockwave-Flash" src="xxxxxx.swf.swf"></embed>
</object>
```

---

同样的，select元素在IE 6下也是以窗口形式显示的，这是IE 6的一个Bug。如代码清单4-82所示

#### 代码清单4-82 select表单元素

---

```
<style type="text/CSS">
```

```
#test{width:200px; height:200px; background:green; position:absolute; left:50px; top:10px;}
</style>
<select><option>-请选择-</option></select>
<div id="test"></div>
```

它在 IE 7 下的截图如图 4-57 所示。

它在 IE 6 下的截图如图 4-58 所示。

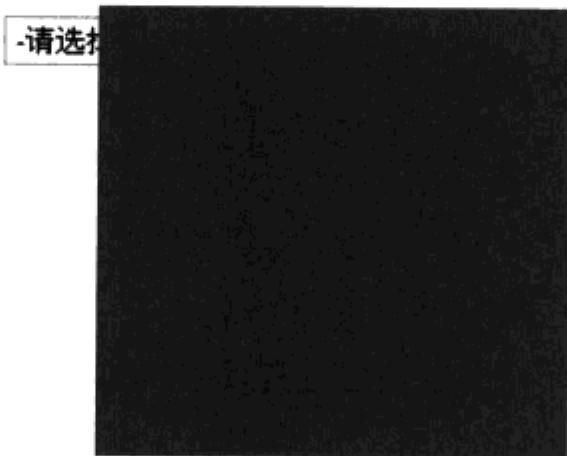


图 4-57 select 元素和绝对定位元素在  
IE 7 下的截图

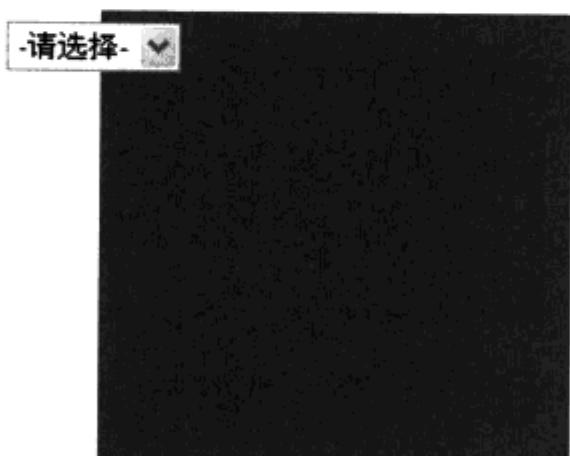


图 4-58 select 元素和绝对定位元素在  
IE 6 下的截图

IE 7 下 select 会被绝对定位的 test 遮住，这是正常现象。在 IE 6 下，select 会浮在绝对定位的 test 之上，这是不符合 CSS 原理的，是 IE 6 下的一个 Bug。解决这个问题稍有点麻烦，如代码清单 4-83 所示。

#### 代码清单 4-83 用 iframe 解决 IE6 下 select 与浮起元素的显示 Bug

```
<style type="text/CSS">
#test{width:200px; height:200px; background:green; position:absolute; left:50px; top:10px; z-index:2}
#testMask{width:200px; height:200px; position:absolute; left:50px; top:10px; z-index:1;}
</style>
<select><option>-请选择-</option></select>
<div id="test"></div>
<iframe id="testMask" frameborder="0" scrolling="no"></iframe>
```

我们可以用一个和 test 同样大小的 iframe 放在 test 下面，select 上面，用 iframe 遮住 select。

### 4.7.12 插入 png 图片

png 格式因为其优秀的压缩算法和对透明度的完美支持，成为 Web 中最流行的图片格式之一。但它也存在一个众所周知的头疼问题——IE 6 下对 png 的透明支持并不好，如代码清单 4-84 所示。

代码清单 4-84 插入 png 图片

---

```

```

---

在 IE7、IE 8 和 Firefox 下的效果如图 4-59 所示。

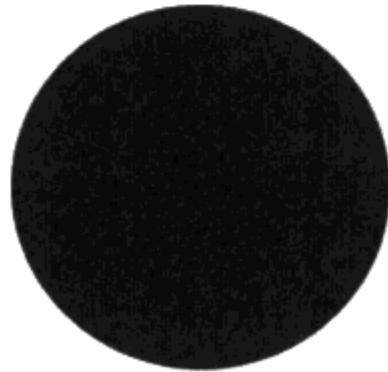


图 4-59 带透明的 png 图片在非 IE 6 下

在 IE 6 下的效果如图 4-60。

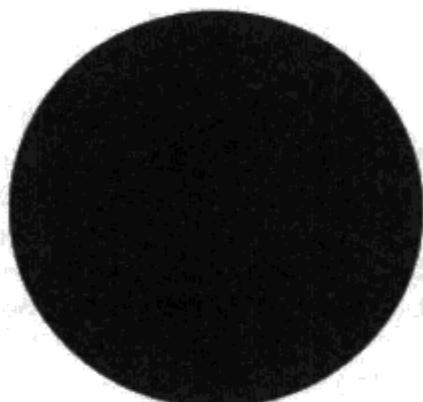


图 4-60 带透明的 png 图片在 IE 6 下

本该是透明的地方，在IE 6 下会显示为浅蓝色。可以使用IE下私有的滤镜功能来解决这个问题，格式如下：`progid:DXImageTransform.Microsoft.AlphaImageLoader(src='png' 图片路径 ,sizingMethod='crop')`。

可以使用以下代码一次性解决页面内所有以``这种图片形式插入网页中的 png 图片，和以`background:url(xxx.png)`这种背景形式插入网页的 png 图片的透明问题，如代码清单 4-85：

**代码清单 4-85 一次性解决 png 透明度在 IE6 下显示问题的脚本**

---

```

function correctPNG()
{
    for(var i=0; i<document.images.length; i++)
    {
        var img = document.images[i]
        var imgName = img.src.toUpperCase()
        if (imgName.substring(imgName.length-3, imgName.length) == "PNG")
        {
            var imgID = (img.id) ? "id='" + img.id + "' " : ""
            var imgClass = (img.className) ? "class='" + img.className + "' " :
            ""
            var imgTitle = (img.title) ? "title='" + img.title + "' " :
            "title='" + img.alt + "' "
            var imgStyle = "display:inline-block;" + img.style.cssText
            if (img.align == "left") imgStyle = "float:left;" + imgStyle
            if (img.align == "right") imgStyle = "float:right;" + imgStyle
            if (img.parentElement.href) imgStyle = "cursor:hand;" + imgStyle
            var strNewHTML = "<span " + imgID + imgClass + imgTitle
            + " style=\"" + "width:" + img.width + "px; height:" + img.height
            + "px;" + imgStyle + ";"
            + "filter:progid:DXImageTransform.Microsoft.AlphaImageLoader"
            + "(src=\"'" + img.src + "\', sizingMethod='scale');\"></span>"
            img.outerHTML = strNewHTML
            i = i-1
        }
    }
}

function alphaBackgrounds(){
    var rslt = navigator.appVersion.match(/MSIE (d+.d+)/, '');
    var itsAllGood = (rslt != null && Number(rslt[1]) >= 5.5);
    for (i=0; i<document.all.length; i++){
        var bg = document.all[i].currentStyle.backgroundImage;
        if (bg){
            if (bg.match(/.png/i) != null){
                var mypng = bg.substring(5,bg.length-2);
                document.all[i].style.filter =
"progid:DXImageTransform.Microsoft.AlphaImageLoader(src='"+mypng+"',
sizingMethod='crop')";
                document.all[i].style.backgroundImage = "url('')";
            }
        }
    }
}

```

```

        }
    }

}

if (navigator.platform == "Win32" && navigator.appName == "Microsoft
Internet Explorer" && window.attachEvent) {
    window.attachEvent("onload", correctPNG);
    window.attachEvent("onload", alphaBackgrounds);
}

```

---

它的实现原理是当页面加载完成后，先遍历页面内所有的图片元素，找到后缀为.png 的图片，将它们改成 span 标签，设置为 display:inline-block 的类型，长宽和原 png 图片一样，然后配上 progid:DXImageTransform.Microsoft.AlphaImageLoader(src='png 图片路径 ',sizingMethod='crop') 的滤镜，实现 png 的透明效果，再遍历页面内所有元素，检查它们是否有 png 的背景图，如果有，则将背景图去掉，改用滤镜。这种方法调用非常方便，只要将这段代码放入网页中任意地方，当页面加载完成后，png 图片都可以实现透明。但他也存在一定问题：

- 1) 当页面全部加载完成前，png 图片的透明部分仍然会显示为浅蓝；
- 2) 遍历所有元素，执行效率不高；
- 3) 当 png 是以背景形式插入网页的，它可能有 background-position 和 background-repeat 属性，而滤镜是不支持这两个属性的，如果设置了这两个属性，这两个属性的效果会丢失。

#### 4.7.13 多版本 IE 并存方案——CSS 的调试利器 IETester

目前 IE 6、IE 7 和 IE 8 都有不小的市场份额，我们的网页需要在这 3 个不同版本的 IE 下兼容。通常情况下，我们只能安装一个版本的 IE，不能多个版本并存，这就为调试工作带来了麻烦。

在早期的时候，我们可能更多的在使用 Multiple IE 这个软件，它能提供 IE 3 ~ IE 7 多版本的 IE 共存。但现在我们有了更好的利器 IETester，它可以提供 IE5.5、IE 8 多个版本 IE 共存，与 Multiple IE 不同的是，Multiple IE 安装完成后，会有多个不同的桌面快捷方式，分别对应不同版本的 IE，IETester 安装完成后桌面只有一个快捷方式，IE 的不同版本是通过标签的形式实现的。

由于 IE 3、IE 4、IE 5 的市场份额几乎可以忽略不计，目前流行的 IE 版本只有 IE 6、IE 7 和 IE 8，一般来说，只用兼容这几个版本就可以了，如果网页流量巨大，要求稍高点的网站，可能还会要求兼容 IE 5.5。Multiple IE 已经淘汰了，而 IETester 正好能满足我们开发调试的需求。更多详情请访问 IETester 的官方网站 <http://www.my-debugbar.com/wiki/IETester/HomePage>。

## 第 5 章

# 高质量的 JavaScript

### 本章内容

- 养成良好的编程习惯
- JavaScript 的分层概念和 JavaScript 库
- 编程实用技巧
- 面向对象编程

## 5.1 养成良好的编程习惯

### 5.1.1 团队合作——如何避免 JS 冲突

网页中需要某个功能 A，工程师甲为其写了一段 JavaScript 代码，如代码清单 5-1 所示。

代码清单 5-1 工程师甲编写功能 A

---

```
<div>
    xxxxxxxxxxxx
</div>
<script type="text/Javascript">
var a = 123,    b = "hello world";
...
</script>
<div>
    xxxxxxxxxxxx
</div>
```

---

这时代码运行很正常，维护方面也不会有什么明显的问题。但后来这个页面又有了新的需求，需要添加功能 B，工程师乙为其写了另一段 JavaScript 代码，如代码清单 5-2 所示。

代码清单 5-2 工程师乙添加功能 B

---

```
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
var a = 123,    b = "hello world";
...
</script>
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
var a,    c = "abc";
...
</script>
<div>
    xxxxxxxxxxxx
</div>
```

---

我们需要注意的是，工程师甲的脚本定义了一个变量 a，工程师乙在不知情的情况下，也定义了一个同名的变量 a，它们都是直接定义在 window 作用域下的，同名的变量会互相冲突。如果没有工程师甲的脚本在前，单独运行工程师乙的脚本是没有任何问题的，但事实上，由于工程师甲和乙在同一作用域内定义了同名的变量，所以他们的脚本有可能产生冲突，出现意料之外的错误。在我们的示例中，只涉及两段脚本而已，出现问题还比较容易找到原因，但在实战中，我们可能经常面对多段脚本，几百甚至几千行代码，全局变量很容易产生冲突，在我们意想不到的地方运行出错。全局变量泛滥是件非常可怕的事，它就像一个躲在暗中的杀手，不知道什么时候会跳出来给你一刀。

如何避免这种冲突隐患呢？我们需要对全局变量进行有效控制，切忌全局变量泛滥。一种最简单也是最有效的方法就是用匿名函数将脚本包起来，让变量的作用域控制在匿名函数之内，如代码清单 5-3 所示。

代码清单 5-3 使用匿名函数控制变量的作用域

---

```
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var a = 123, b = "hello world";
    ...
})();
</script>
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var a, c = "abc";
    ...
})();
</script>
<div>
    xxxxxxxxxxxx
</div>
```

---

“(function(){})()”这种形式很巧妙，先定义一个匿名的 function，然后立即执行它。包在这个匿名 function 里的变量，作用域就不再是 window，而是局限在函数内

部。代码清单 5-3 里两处脚本中的变量 a 因为各自包在不同的匿名函数里，也就不再互相冲突了。

### 用匿名函数将脚本包起来，可以有效控制全局变量，避免冲突隐患。

为了预防程序冲突，工程师甲在编写程序的时候，就应该考虑到以后有可能会有新的程序加入，在一开始就应该主动使用匿名函数封装起自己脚本的变量。

每段脚本都各自包含在不同的匿名函数中，可以避免冲突隐患，但如果不同脚本之间需要通信怎么办呢？比如这时又有了新需求，需要在网页中加入功能 C。功能 B 是完全和功能 A 没有任何关联的，脚本之间不需要任何通信，和功能 B 不同的是，功能 C 和功能 A 之间有关联，需要用到功能 A 脚本中的变量 b。最简单的办法当然是将功能 C 的脚本直接写到功能 A 的匿名函数中，这样功能 C 的脚本就可以访问到变量 b，如代码清单 5-4 所示。

代码清单 5-4 功能 C 和功能 A 的通信

---

```

<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var a = 123, b = "hello world";
    ...
    var d="adang is very handsome!";
    d=b + ", " + d;
    ...
})();
</script>
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var a, c = "abc";
    ...
})();
</script>
<div>
    xxxxxxxxxxxx
</div>

```

---

但不巧的是，处理功能 C 的是工程师丙，为了避免功能 C 的程序和功能 A 不产生冲突，他只有两种选择：1) 阅读和理解工程师甲写的关于功能 A 的程序，小心地避开

冲突；2) 不在工程师甲的匿名函数中写程序，写一个自己的匿名函数。而功能A的脚本非常复杂，它和功能C需要关联的地方又不多，工程师丙不愿意花大量精力去理解工程师甲写的脚本，他更愿意写一个自己的匿名函数，如代码清单5-5所示。

代码清单5-5 工程师丙添加功能C

---

```

<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var a = 123, b = "hello world";
    ...
})();
</script>
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var a, c = "abc";
    ...
})();
</script>
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var d="adang is very handsome!";
    d=b + ", " + d;
    ...
})();
</script>

```

---

于是问题来了，功能A脚本中的变量b和功能C脚本中的变量b不在同一个作用域，两段脚本没法进行通信，功能C脚本中的变量b不再等于“hello world”，而是undefined。

如何解决这个问题呢？为了解决匿名函数之间的通信问题，我们可以在window作用域下定义一个全局变量，把它当做一个桥梁，完成各匿名函数之间的通信，如代码清单5-6所示。

代码清单5-6 利用全局作用域的变量在各匿名函数间搭起桥梁

---

```
<div>
```

```

xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
var str;
</script>
<script type="text/JavaScript">
(function() {
    var a = 123, str = b = "hello world";
    ...
})();
</script>
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var a, c = "abc";
    ...
})();
</script>
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var b = str;
    var d = "adang is very handsome!";
    d=b + ", " + d;
    ...
})();
</script>

```

定义一个全局作用域的变量 `str`, 可以帮助我们在不同匿名函数间通信。但代码清单 5-6 中全局变量的形式还不够好, 它还存在一个致命的弱点。如果功能 C 还需要功能 A 中的变量 `a` 呢? 按照代码清单 5-6 的思路, 新的代码可能如代码清单 5-7 所示。

#### 代码清单 5-7 添加新的全局变量

---

```

<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
var str,str2;
</script>
<script type="text/JavaScript">
(function() {
    var str2 = a = 123, str = b = "hello world";
    ...
})();
</script>
<div>

```

```

xxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var a, c = "abc";
    ...
})();
</script>
<div>
xxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var a = str2, b = str;
    var d="adang is very handsome! ";
    d=b + ", " + d + a;
    ...
})();
</script>

```

我们需要新增一个全局变量 str2，作为变量 a 在不同匿名函数间通信的桥梁。不同匿名函数间需要通信的变量越多，我们需要的全局变量也就越多！前面我们说过，全局变量是个可怕的杀手，有引起程序冲突的隐患，也正是因为这个原因，所以我们需要用匿名函数将不同功能的脚本封装起来。如果为了让匿名函数之间能够通信而添加大量全局变量，就违背了我们使用匿名函数的初衷。所以，应该严格控制全局变量的数量！改进后的代码如代码清单 5-8 所示。

代码清单 5-8 用 hash 对象作为全局变量

```

<div>
xxxxxx
</div>
<script type="text/JavaScript">
var GLOBAL={};
</script>
<script type="text/JavaScript">
(function() {
    var a=123, b="hello world";
    GLOBAL.str2=a;
    GLOBAL.str=b;
    ...
})();
</script>
<div>
xxxxxx
</div>
<script type="text/JavaScript">
(function() {

```

```

    var a, c="abc";
    ...
})();
</script>
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var a=GLOBAL.str2, b=GLOBAL.str;
    var d="adang is very handsome! ";
    d=b + ", " + d + a;
    ...
})();
</script>

```

使用普通的变量作为全局变量，扩展性会非常差，所以我们可以使用一个{}对象类型的变量作为全局变量，如果匿名函数间需要多个变量来做通信桥梁，可以将这些变量都作为全局变量的属性。这样做可以保证全局变量的个数足够少，同时扩展性非常好。因为全局变量的作用域是在 window 下的，它有可能会和匿名函数中的变量重名引发冲突，所以我们给全局变量约定一个特定的变量名，作为规范的一部分，让团队中所有的成员都知晓并遵守。笔者推荐使用大写的 GLOBAL 作为全局变量的变量名。

代码清单 5-8 的方法可以解决多个变量需要在不同匿名函数间通信的问题，但它还有一个隐患——因为 GLOBAL 是全局变量，用做通信桥梁的变量是作为 GLOBAL 的属性存在的，如果变量的命名比较简单，还是很容易互相覆盖掉的。例如新增功能 D，功能 D 需要和功能 B 通信，使用功能 B 脚本中的变量 c。开发功能 D 的是工程师丁，借用 GLOBAL，让功能 B 和功能 D 的脚本间建立起了联系，如代码清单 5-9 所示。

### 代码清单 5-9 全局变量的冲突

```

<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
var GLOBAL={};
</script>
<script type="text/JavaScript">
(function() {
    var a=123, b="hello world";
    GLOBAL.str2=a;
    GLOBAL.str=b;
    ...
})();

```

```

</script>
<div>
  XXXXXXXXXXXX
</div>
<script type="text/JavaScript">
(function(){
  var a, c = "abc";
  GLOBAL.str = c;
  ...
})();
</script>
<div>
  XXXXXXXXXXXX
</div>
<script type="text/JavaScript">
(function(){
  var a=GLOBAL.str2, b=GLOBAL.str;
  var d="adang is very handsome!";
  d=b + ", " + d + a;
  ...
})();
</script>
<div>
  XXXXXXXXXXXX
</div>
<script type="text/JavaScript">
(function(){
  var test=GLOBAL.str;
  alert(test);
  ...
})();
</script>

```

---

工程师丁只关心自己的匿名函数和功能 B 的匿名函数，他的初衷是用 GLOBAL.str 建立起功能 B 和功能 D 脚本之间的通信，但无意中覆盖掉了功能 A 中设置的 GLOBAL.str，导致功能 C 的脚本出错。如何避免这种冲突呢？难道每个工程师在使用 GLOBAL 对象之前，都要查找一下 GLOBAL 已经绑定了哪些属性吗？

当然不是。我们可以使用命名空间来解决这个问题。命名空间是一种特殊的前缀，在 JavaScript 中它其实是通过一个 {} 对象实现的。在不同的匿名函数中，我们根据功能声明一个不同的命名空间，然后每个匿名函数中 GLOBAL 对象的属性都不要直接挂在 GLOBAL 对象上，而是挂在此匿名函数的命名空间下，如代码清单 5-10 所示。

#### 代码清单 5-10 使用命名空间

---

```

<div>
  XXXXXXXXXXXX

```

```

</div>
<script type="text/JavaScript">
var GLOBAL={};
</script>
<script type="text/JavaScript">
(function() {
    var a=123, b="hello world";
    GLOBAL.A={};
    GLOBAL.A.str2=a;
    GLOBAL.A.str=b;
    ...
})();
</script>
<div>
    xxxxxxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var a, c="abc";
    GLOBAL.B={};
    GLOBAL.B.str=c;
    ...
})();
</script>
<div>
    xxxxxxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var a=GLOBAL.A.str2, b=GLOBAL.A.str;
    var d="adang is very handsome!";
    d=b + ", " + d + a;
    ...
})();
</script>
<div>
    xxxxxxxxxx
</div>
<script type="text/JavaScript">
(function() {
    var test=GLOBAL.B.str;
    alert(test);
    ...
})();
</script>

```

现在我们只用考虑命名空间不会冲突就可以了，属性名可以任意设置而不用担心会和哪个匿名函数里的某个 GLOBAL 属性冲突，一般情况下，命名空间的数量不会特别多，避免命名空间冲突相对而言比较容易。

如果同一个匿名函数中的程序非常复杂，变量名很多，命名空间还可以进一步扩

展，生成二级命名空间，以功能A为例，如代码清单5-11所示。

代码清单5-11 使用多级命名空间

---

```
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
var GLOBAL={};
</script>
<script type="text/JavaScript">
(function() {
    var a=123, b="hello world";
    GLOBAL.A={};
    GLOBAL.A.CAT={};
    GLOBAL.A.DOG={};
    GLOBAL.A.CAT.name="mimi";
    GLOBAL.A.DOG.name="wangcai";
    GLOBAL.A.CAT.move=function() {
    }
    GLOBAL.A.DOG.move=function() {
    }
    GLOBAL.A.str2=a;
    GLOBAL.A.str=b;
    ...
})();
</script>
```

---

因为生成命名空间是个非常常用的功能，我们可以进一步将生成命名空间的功能定义成一个函数，方便调用，如代码清单5-12所示。

代码清单5-12 定义命名空间函数

---

```
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
var GLOBAL={};
GLOBAL.namespace=function(str) {
    var arr=str.split("."),o=GLOBAL;
    for (i=(arr[0]=="GLOBAL") ? 1 : 0; i<arr.length; i++) {
        o[arr[i]]=o[arr[i]] || {};
        o=o[arr[i]];
    }
}
</script>
<script type="text/JavaScript">
(function() {
    var a=123, b="hello world";
```

---

```

GLOBAL.namespace("A.CAT");
GLOBAL.namespace("A.DOG");
GLOBAL.A.CAT.name="mimi";
GLOBAL.A.DOG.name="wangcai";
GLOBAL.A.CAT.move=function() {
}
GLOBAL.A.DOG.move=function() {
}
GLOBAL.A.str2=a;
GLOBAL.A.str=b;
...
})();
</script>

```

我们给 GLOBAL 对象添加 namespace 方法，然后可以在需要命名空间的匿名函数中调用，非常方便。

现在我们来完整地看一下改写后的代码，如代码清单 5-13 所示。

代码清单 5-13 使用命名空间解决冲突的完整代码

```

<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
var GLOBAL={};
GLOBAL.namespace=function(str){
    var arr=str.split("."),o=GLOBAL;
    for (i=(arr[0]=="GLOBAL") ? 1 : 0; i<arr.length; i++) {
        o[arr[i]]=o[arr[i]] || {};
        o=o[arr[i]];
    }
}
</script>
<script type="text/JavaScript">
(function(){
    var a=123, b="hello world";
    GLOBAL.namespace("A.CAT");
    GLOBAL.namespace("A.DOG");
    GLOBAL.A.CAT.name="mimi";
    GLOBAL.A.DOG.name="wangcai";
    GLOBAL.A.CAT.move=function() {
    }
    GLOBAL.A.DOG.move=function() {
    }
    GLOBAL.A.str2=a;
    GLOBAL.A.str=b;
    ...
})

```

```
)();
</script>
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
(function(){
    var a, c="abc";
    GLOBAL.namespace("B");
    GLOBAL.B.str=c;
    ...
})();
</script>
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
(function(){
    var a=GLOBAL.A.str2, b=GLOBAL.A.str;
    var d="adang is very handsome!";
    d=b + ", " + d + a;
    ...
})();
</script>
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
(function(){
    var test=GLOBAL.B.str;
    alert(test);
    ...
})();
</script>
```

代码的冲突问题已经很好地解决了，但可维护性并不强。比如，现在需要让工程师甲去修改功能 B。因为工程师甲写的脚本是关于功能 A 的，他并不知道功能 B 的脚本情况，工程师甲接到修改功能 B 的任务后，应该会很头疼，页面中有四段匿名函数，除了功能 A 是自己写的，对其他的脚本一无所知，他不得不阅读其他三段匿名函数，猜测它们的用途。工程师甲在粗略阅读完几段脚本后，大概猜到了几段脚本的功能，功能 B 是哪段脚本完成的，于是将注意力集中在相应的区域，但是很快发现有些代码不是很容易理解，希望能从原来编写这段脚本的工程师那里寻求帮助，遗憾的是现有的代码中找不到该工程师的信息。

为了改善这种局面，我们需要给代码添加适当的注释，以提高代码的可维护性。如

代码清单 5-14 所示。

代码清单 5-14 给代码添加注释

```

<div>
    xxxxxxxxxx
</div>
<script type="text/JavaScript">
var GLOBAL={};
GLOBAL.namespace=function(str) {
    var arr=str.split("."),o=GLOBAL;
    for (i=(arr[0]=="GLOBAL") ? 1 : 0; i<arr.length; i++) {
        o[arr[i]]=o[arr[i]] || {};
        o=o[arr[i]];
    }
}
</script>
<script type="text/JavaScript">
//=====
// 功能 A
// 工程师甲
// email:cly84920@gmail.com msn:cly84920@hotmail.com
// 2009-11-04
//=====
(function() {
    var a=123, b="hello world";
    GLOBAL.namespace("A.CAT");
    GLOBAL.namespace("A.DOG");
    GLOBAL.A.CAT.name="mimi";
    GLOBAL.A.DOG.name="wangcai";
    GLOBAL.A.CAT.move=function() {
    }
    GLOBAL.A.DOG.move=function() {
    }
    GLOBAL.A.str2=a;
    GLOBAL.A.str=b;
    ...
})();
</script>
<div>
    xxxxxxxxxx
</div>
<script type="text/JavaScript">
//=====
// 功能 B
// 工程师乙
// email:a@gmail.com msn:a@hotmail.com
// 2009-11-05
//=====
(function() {
    var a, c="abc";

```

```
GLOBAL.namespace("B");
GLOBAL.B.str=c;
...
})();
</script>
<div>
    XXXXXXXXXX
</div>
<script type="text/JavaScript">
//=====
// 功能C
// 工程师丙
// email:b@yahoo.com msn:b@hotmail.com
// 2009-11-09
//=====
(function() {
    var a=GLOBAL.A.str2, b=GLOBAL.A.str;
    var d="adang is very handsome!";
    d=b + ", " + d + a;
    ...
})();
</script>
<div>
    XXXXXXXXXX
</div>
<script type="text/JavaScript">
//=====
// 功能D
// 工程师丁
// email:c@126.com msn:c@hotmail.com
// 2009-11-20
//=====
(function() {
    var test=GLOBAL.B.str;
    alert(test);
    ...
})();
</script>
```

给每段脚本加上适当的注释，如代码清单 5-14 中添加的信息包括：功能说明、工程师姓名、工程师联系方式以及代码的最后修改时间，如果需要修改某个功能，很容易就能找到相关代码的位置和原开发工程师的联系方式。添加必要的注释，可以大大提高代码的可维护性，对于团队合作来说，更是十分有必要的。

无论是多人的直接团队合作，还是一个人的间接团队合作，都需要良好的可维护性！让 JS 不产生冲突，需要避免全局变量的泛滥，合理使用命名空间以及为代码添加必要的注释。

### 5.1.2 给程序一个统一的入口——window.onload 和 DOMReady

JavaScript 是种脚本语言，浏览器下载到哪儿就会执行到哪儿，这种特性会为编程带来方便，但也容易使程序支离破碎，过于零散。代码清单 5-14 就存在这样的问题：功能 A、B、C、D 分别放在 4 个不同位置的<script>标签里，下载后立即执行，整个网页似乎没有一个明显的程序“入口”。我们的例子中还只是涉及 4 段功能，似乎问题还不太大，但如果网页中的功能非常多，<Script>标签零散地分布在网页里，下载后就立即执行，这就像是几支没有组织的小队伍，各自相对独立地进行着游击，虽然也能发挥作用，但纪律性没有保障。

为了解决这一问题，首先我们需要从功能上对程序进行职能划分。网页中的 JavaScript 从功能上，应该分成两大部分——框架部分和应用部分。框架部分提供的是对 JavaScript 代码的组织作用，包括定义全局变量，定义命名空间方法等，它和具体应用无关，每个页面都需要包括相同的框架，所以框架部分的代码在每个页面都相同。应用部分提供的是页面功能逻辑，不同页面会有不同的功能，不同页面应用部分的代码也不相同。以代码清单 5-14 为例，我们来看看哪些地方属于框架部分，哪些地方属于应用部分，应用部分用黑体表示，其余则为框架部分，如代码清单 5-15 所示。

代码清单 5-15 区分 javascript 的框架部分和应用部分

---

```

<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
var GLOBAL={};
GLOBAL.namespace=function(str){
    var arr=str.split("."),o=GLOBAL;
    for (i=(arr[0]=="GLOBAL") ? 1 : 0; i<arr.length; i++) {
        o[arr[i]]=o[arr[i]] || {};
        o=o[arr[i]];
    }
}
</script>
<script type="text/JavaScript">
//=====
// 功能A
// 工程师甲
// email:cly84920@gmail.com    msn:cly84920@hotmail.com
// 2009-11-04

```

```
//=====
(function() {
    var a=123, b="hello world";
    GLOBAL.namespace("A.CAT");
    GLOBAL.namespace("A.DOG");
    GLOBAL.A.CAT.name="mimi";
    GLOBAL.A.DOG.name="wangcai";
    GLOBAL.A.CAT.move=function() {
        ...
    }
    GLOBAL.A.DOG.move=function() {
        ...
    }
    GLOBAL.A.str2=a;
    GLOBAL.A.str=b;
    ...
})();
</script>
<div>
    XXXXXXXXXX
</div>
<script type="text/JavaScript">
//=====
// 功能B
// 工程师乙
// email:a@gmail.com msn:a@hotmail.com
// 2009-11-05
//=====
(function() {
    var a, c="abc";
    GLOBAL.namespace("B");
    GLOBAL.B.str=c;
    ...
})();
</script>
<div>
    XXXXXXXXXX
</div>
<script type="text/JavaScript">
//=====
// 功能C
// 工程师丙
// email:b@yahoo.com msn:b@hotmail.com
// 2009-11-09
//=====
(function() {
    var a=GLOBAL.A.str2, b=GLOBAL.A.str;
    var d="adang is very handsome!";
    d=b + ", " + d + a;
    ...
})();
</script>
<div>
    XXXXXXXXXX
</div>
```

```

<script type="text/JavaScript">
//=====
// 功能 D
// 工程师丁
// email:c@126.com msn:c@hotmail.com
// 2009-11-20
//=====
(function() {
    var test=GLOBAL.B.str;
    alert(test);

})
</script>

```

接下来，我们需要将应用部分的代码组织起来，给它们一个统一的“入口”，如代码清单 5-16 所示。

代码清单 5-16 给应用部分的代码一个统一的“入口”

```

<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
var GLOBAL={};
GLOBAL.namespace=function(str){
    var arr=str.split("."),o=GLOBAL;
    for (i=(arr[0]=="GLOBAL") ? 1 : 0; i<arr.length; i++) {
        o[arr[i]]=o[arr[i]] || {};
        o=o[arr[i]];
    }
}
</script>
<script type="text/JavaScript">
function init(){
//=====
// 功能 A
// 工程师甲
// email:cly84920@gmail.com msn:cly84920@hotmail.com
// 2009-11-04
//=====
(function() {
    var a=123, b="hello world";
    GLOBAL.namespace("A.CAT");
    GLOBAL.namespace("A.DOG");
    GLOBAL.A.CAT.name="mimi";
    GLOBAL.A.DOG.name="wangcai";
    GLOBAL.A.CAT.move=function(){

    }
    GLOBAL.A.DOG.move=function(){

})

```

```
GLOBAL.A.str2=a;
GLOBAL.A.str=b;
...
})();

//=====
// 功能B
// 工程师乙
// email:a@gmail.com msn:a@hotmail.com
// 2009-11-05
//=====
(function(){
    var a, c="abc";
    GLOBAL.namespace("B");
    GLOBAL.B.str=c;
    ...
})();

//=====
// 功能C
// 工程师丙
// email:b@yahoo.com msn:b@hotmail.com
// 2009-11-09
//=====
(function(){
    var a=GLOBAL.A.str2, b=GLOBAL.A.str;
    var d="adang is very handsome!";
    d=b + ", " + d + a;
    ...
})();

//=====
// 功能D
// 工程师丁
// email:c@126.com msn:c@hotmail.com
// 2009-11-20
//=====
(function(){
    var test=GLOBAL.B.str;
    alert(test);
    ...
})();
}

</script>
<div>
    XXXXXXXXXXXX
</div>
<div>
    XXXXXXXXXXXX
</div>
<div>
    XXXXXXXXXXXX
</div>
```

如此一来，所有应用部分的代码都集中到 init 函数里了，所有的初始化工作都在这里，它便是网页的程序入口。如果需要修改已有功能或者添加新功能，都可以直接找到它。

接下来，我们需要在适当的时候调用这个入口函数（init），完成页面程序的初始化。什么时候最合适呢？首先我们需要知道的是，JavaScript 是脚本语言，加载到哪儿就执行到哪儿，如果程序控制某个 DOM 节点，而该 DOM 节点当时还没有被载入，程序就会报错。如代码清单 5-17 所示。

**代码清单 5-17 在 DOM 节点加载进来之前就调用**

---

```
<script type="text/JavaScript">
alert(document.getElementById("test").innerHTML);
</script>
<div id="test">hello world</div>
```

---

在上面的代码中，我们想弹出 id 为 test 的 DOM 节点的内容，但执行脚本的时候，事实上 test 节点还未加载下来，所以 document.getElementById("test") 不能按预期的定位到 test 节点。为了解决这个问题，我们可以将脚本移到后面，如代码清单 5-18 所示。

**代码清单 5-18 调整脚本的位置**

---

```
<div id="test">hello world</div>
<script type="text/JavaScript">
alert(document.getElementById("test").innerHTML);
</script>
```

---

但有时我们希望程序能够无视其位置，无视是否在相应的 DOM 节点之后，我们该如何操作呢？我们可以监听 window 对象的 onload 事件，当 window 触发 onload 事件后调用脚本，如代码清单 5-19 所示。

**代码清单 5-19 监听 window.onload**

---

```
<script type="text/JavaScript">
window.onload=function(){
    alert(document.getElementById("test").innerHTML);
}
</script>
<div id="test">hello world</div>
```

---

如此一来，alert(document.getElementById("test").innerHTML) 并不会马上执行，而

是等到 window.onload 之后才会去执行。window 对象会在网页内元素全部加载完毕之后触发 onload 事件，而此时，test 节点已经加载，所以程序不会报错。前面我们讲过，应用部分的代码最好包在一个约定好的入口函数里，函数名并没有硬性规定，但按照编程习惯，最好还是叫做 init 比较容易理解，如代码清单 5-20 所示。

代码清单 5-20 定义初始化方法的名称为 init

---

```
<script type="text/JavaScript">
function init(){
    alert(document.getElementById("test").innerHTML);
}
window.onload=init;
</script>
<div id="test">hello world</div>
```

---

看似已经非常完美了，但还存在一个问题，window 的 onload 事件要求网页内所有的元素全部加载完毕后才会触发，如果网页里有很大的图片，加载时间非常长，那么我们的初始化函数会延时很久后才会执行！在网速较慢的情况下，这样的延时往往是不可接受的。为了解决这个问题，很多 JS 框架提供了 DOMReady 事件代替 window.onload。DOMReady 和 window.onload 的作用很像，但和 window.onload 不同的是，window.onload 需要当页面完全加载完成时才会触发，包括图片、Flash 等富媒体，DOMReady 只判断页面内所有的 DOM 节点是否已经全部生成，至于节点的内容是否加载完成，它并不关心。所以 DOMReady 触发的速度比 window.onload 更快，特别是当页面内有大量图片等富媒体资源时，DOMReady 触发很久之后，才会触发 window.onload。通常情况下我们编程只关心 DOM 节点是否已生成，并不关心是否已加载完，所以 DOMReady 比 window.onload 更适合用来调用初始化函数。

值得注意的是，DOMReady 并不是原生 JavaScript 支持的事件，它不能像 window.load 那样直接调用，一般我们都是结合 JS 框架来使用它。如果使用 jQuery，如代码清单 5-21 所示。

代码清单 5-21 jQuery 中监听 DOMReady

---

```
<script type="text/JavaScript" src="http://ajax.googleAPIs.com/ajax/libs/
jquery/1.3/jquery.min.js"></script>
<script type="text/JavaScript">
```

```

function init(){
    alert(document.getElementById("test").innerHTML);
}
$(document).ready(init);
</script>
<div id="test">hello world</div>

```

使用 YUI，如代码清单 5-22 所示。

代码清单 5-22 YUI 中监听 DOMReady

```

<script type="text/JavaScript" src="http://yui.yahooapis.com/combo?2.8.0r4/
build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/JavaScript">
function init(){
    alert(document.getElementById("test").innerHTML);
}
YAHOO.util.Event.onDOMReady(init);
</script>
<div id="test">hello world</div>

```

如果不使用任何 JS 框架，我们当然也可以自己模拟 DOMReady 事件，但 DOMReady 的触发原理比较复杂，不同浏览器的原理不同。其实我们可以绕开 DOMReady 事件，用一种最简单的方法来达到相似的功能，如代码清单 5-23 所示。

代码清单 5-23 模拟 DOMReady 效果

```

<script type="text/JavaScript">
function init(){
    alert(document.getElementById("test").innerHTML);
}
</script>
<div id="test">hello world</div>
...
<script type="text/JavaScript">
init();
</script>
</body>

```

我们定义 init 函数，但并不急着立即调用它，我们可以在页面的最后，即</body>标签之前再调用 init 函数，此时页面内的 DOM 节点不一定都“加载完成”了，但一定都“生成”了，从而模拟 DOMReady 的效果。

现在我们回头接着完善代码清单 5-16，在合适的时候调用程序的初始化函数，因为代码清单 5-16 中我们没有使用任何 JS 框架，所以我们通过在</body>标签之前调用它

来实现 DOMReady 效果，如代码清单 5-24 所示。

代码清单 5-24 调用 init 函数

```
<div>
    xxxxxxxxxxxx
</div>
<script type="text/JavaScript">
var GLOBAL={};
GLOBAL.namespace=function(str){
    var arr=str.split("."),o=GLOBAL;
    for (i=(arr[0]==="GLOBAL") ? 1 : 0; i<arr.length; i++) {
        o[arr[i]]=o[arr[i]] || {};
        o=o[arr[i]];
    }
}
</script>
<script type="text/JavaScript">
function init(){
//=====
//  功能 A
//  工程师甲
//  email:cly84920@gmail.com  msn:cly84920@hotmail.com
//  2009-11-04
//=====
(function(){
    var a=123, b="hello world";
    GLOBAL.namespace("A.CAT");
    GLOBAL.namespace("A.DOG");
    GLOBAL.A.CAT.name="mimi";
    GLOBAL.A.DOG.name="wangcai";
    GLOBAL.A.CAT.move=function() {
    }
    GLOBAL.A.DOG.move=function() {
    }
    GLOBAL.A.str2=a;
    GLOBAL.A.str=b;
    ...
})();
//=====
//  功能 B
//  工程师乙
//  email:a@gmail.com  msn:a@hotmail.com
//  2009-11-05
//=====
(function(){
    var a, c="abc";
    GLOBAL.namespace("B");
    GLOBAL.B.str=c;
    ...
})();
}
```

```

))();

//=====
// 功能 C
// 工程师丙
// email:b@yahoo.com msn:b@hotmail.com
// 2009-11-09
//=====

(function() {
    var a=GLOBAL.A.str2, b=GLOBAL.A.str;
    var d="adang is very handsome!";
    d=b + ", " + d + a;
    ...
})();

//=====
// 功能 D
// 工程师丁
// email:c@126.com msn:c@hotmail.com
// 2009-11-20
//=====

(function() {
    var test=GLOBAL.B.str;
    alert(test);
    ...
})();
}

</script>
<div>
    XXXXXXXXXXXX
</div>
<div>
    XXXXXXXXXXXX
</div>
<div>
    XXXXXXXXXXXX
</div>
...
<script type="text/JavaScript">
init();
</script>
</body>

```

现在我们再分析一下代码清单 5-24，不难看出 JS 的框架部分如代码清单 5-25 所示。

代码清单 5-25 框架部分的代码

---

```

<div>
    XXXXXXXXXXXX
</div>
<script type="text/JavaScript">
```

```

var GLOBAL={};
GLOBAL.namespace=function(str) {
    var arr=str.split("."),o=GLOBAL;
    for (i=(arr[0]==="GLOBAL") ? 1 : 0; i<arr.length; i++) {
        o[arr[i]]=o[arr[i]] || {};
        o=o[arr[i]];
    }
}
</script>
<script type="text/JavaScript">
function init(){
    //应用部分的JavaScript
}
</script>
<div>
    xxxxxxxxxxxx
</div>
<div>
    xxxxxxxxxxxx
</div>
<div>
    xxxxxxxxxxxx
</div>
...
<script type="text/JavaScript">
init();
</script>
</body>

```

---

框架部分的思路很清晰，首先定义一个全局作用域的变量 GLOBAL 对象，然后给这个全局变量绑定一个命名空间的方法 namespace。如果除了 namespace 外，我们还需要别的全局作用域下的函数，我们同样可以将它作为一个方法挂在 GLOBAL 对象上，这么做好处是所有全局作用域下的函数都放在了 GLOBAL 命名空间下，有效控制了 window 作用域下函数的数量，减小了和应用部分 JS 冲突的隐患。然后提供了一个应用部分 JS 的统一入口函数 init，最后在 DOMReady 的时候调用它。

在实际工作中，网站的头部和尾部通常会做成单独的文件，用服务器端语言 include 到网页中，所以我们的网页通常拆分成三个文件，其中头部文件的形式大致如代码清单 5-26 所示。

#### 代码清单 5-26 头部文件

---

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/
TR/html4/strict.dtd">
<html>
```

```

<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>阿当制作</title>
<link rel="stylesheet" type="text/CSS" href="xxx.css" />
</head>
<body>
<div class="head">
    ...
</div>
<script type="text/JavaScript">
var GLOBAL={};
GLOBAL.namespace=function(str) {
    var arr=str.split("."),o=GLOBAL;
    for (i=(arr[0]==="GLOBAL") ? 1 : 0; i<arr.length; i++) {
        o[arr[i]]=o[arr[i]] || {};
        o=o[arr[i]];
    }
}
</script>

```

尾部文件的形式大致如代码清单 5-27 所示。

代码清单 5-27 尾部文件

```

<div class="foot">
    ...
</div>
<script type="text/JavaScript">
    init();
</script>

```

主体文件的形式大致如代码清单 5-28 所示。

代码清单 5-28 主体文件

```

<div>...</div>
<script type="text/JavaScript">
    function init(){
        //应用部分的 JavaScript
    }
</script>

```

一般来说，头尾文件是非常稳定的，全站共用统一的头尾，这部分并不会经常改动，也不存在多工程师配合的问题。而主体部分的文件是各不相同的，可能会由多个工程师来制作，其中有的页面可能并不需要用 JS 写应用，也就是说，主体部分可能并不需要 init 函数。现在我们的 JS 框架会在</body>前调用 init 函数，所以如果页面主体部分没有写 init 函数，就会报错。当然，我们可以要求每位工程师在制作主体部分的文件

时，必须包含 init 函数，哪怕是个空函数，但其实我们可以有更好的处理方式。将尾部文件的 JS 稍微再修改下，就可以解决这个问题，改进后的尾部文件大致如代码清单 5-29 所示。

代码清单 5-29 更健壮的 init 调用

---

```
<div class="foot">
    ...
</div>
<script type="text/JavaScript">
    if(init){
        init();
    }
</script>
```

---

在调用 init 函数前，先判断页面内是否已经定义了 init 函数，如果定义了才会去执行。如此一来，主体部分就可以根据需要决定要不要添加 init 函数。

### 5.1.3 CSS 放在页头，JavaScript 放在页尾

接下来，我们要考虑的是 JavaScript 文件放在网页的什么位置更合适。一般来说，有个简单的原则：CSS 放在页头，JavaScript 放在页尾，比如有一段代码如代码清单 5-30 所示。

代码清单 5-30 有 JavaScript 和 CSS 的网页

---

```
<script type="text/JavaScript">
window.onload=function(){
    //非常多的 JS 代码
    ...
}
</script>
<div id="test">xxxxx</div>
<!--非常多的代码-->
...
<style type="text/CSS">
#test{}
</style>
```

---

为了能够更好地说明问题，所以我举的例子中用到了两个“非常多”加以注释，在这种极端情况下，更容易发现问题。在浏览器中显示这个网页时会出现什么情况呢？浏览器加载网页，加载到 JavaScript 时，脚本非常多，加载脚本花很长时间，而这段时间

里，因为还没有加载 HTML 代码，所以网页显示为空白，脚本“阻塞”了 HTML 的加载。好不容易 JavaScript 加载完了，终于加载到 HTML 了，但这时却发现这些网页元素居然没有样式，直到非常多的 HTML 加载完之后，才开始加载 CSS，此时网页才有了样式。这样的用户体验是不是很糟糕呢？

所以一个好的习惯就是将 CSS 放在页头，在载入 HTML 元素之前，先载入它们的样式，这样可以避免 HTML 出现无样式状态；将 JavaScript 放在页尾，先将网页呈现给用户，再来加载页面内的脚本，避免 JavaScript 阻塞网页的呈现，减少页面空白的时间。修改后的代码如代码清单 5-31 所示。

代码清单 5-31 将 CSS 放在页头，JavaScript 放在页尾

---

```
<style type="text/CSS">
#test{}
</style>
<div id="test">xxxxx</div>
<!--非常多的代码-->
...
<script type="text/JavaScript">
window.onload=function(){
    //非常多的 JavaScript 代码
    ...
}
</script>
```

---

#### 5.1.4 引入编译的概念——文件压缩

为了减小网页的大小，缩短网页的下载时间，在正式发布 JavaScript 之前，我们可以先对它进行一下压缩。

目前最流行的 JavaScript 压缩工具有 Packer 和 YUI Compressor。Packer 的网址是：<http://dean.edwards.name/packer/>，它使用非常方便，网站上提供了一个 textarea 输入框，我们将 JavaScript 代码复制到输入框里，点击压缩按钮，就可以从另一个 textarea 框里得到压缩后的 JavaScript 代码了，界面如图 5-1 所示。

YUI Compressor 的网址是：<http://developer.yahoo.com/yui/compressor/>。YUI Compressor 的使用比 Packer 要麻烦一些，YUI Compressor 是基于 Java 的 jar 应用，在使用之前需要先安装 JDK，配置 Java 环境，然后要下载 jar 文件到本地，通过命令行进

行调用。最典型的调用命令形如“`java -jar yuicompressor-x.y.z.jar myfile.js -o myfile-min.js`”。值得一提的是，YUI Compressor 除了可以压缩 JavaScript 代码，也可以用来压缩 CSS 代码。

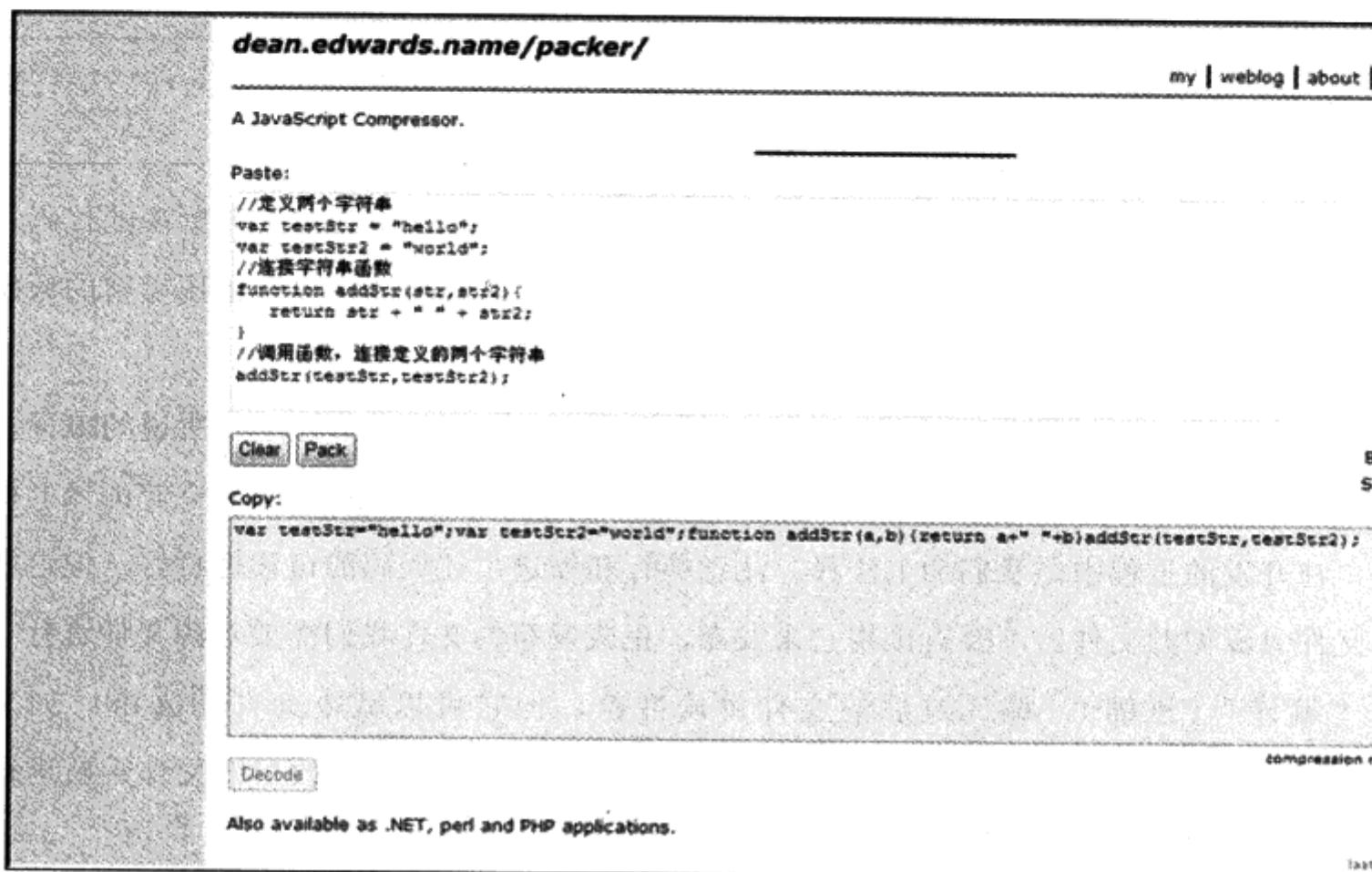


图 5-1 packer 界面

如果你不想安装 JDK，也可以通过 <http://refresh-sf.com/yui/> 在线使用 YUI Compressor。

JS 压缩通常的做法是去掉空格和换行，去掉注释，将复杂变量名替换成简单的变量名。有一段简单的 JavaScript 脚本，如代码清单 5-32 所示。

代码清单 5-32 未压缩的 JavaScript 脚本

---

```
//定义两个字符串
var testStr="hello";
var testStr2="world";
//连接字符串函数
function addStr(str,str2){
    return str + " " + str2;
}
```

```
//调用函数，连接定义的两个字符串  
addStr(testStr,testStr2);
```

用 Packer 压缩之后，变成如代码清单 5-33 所示的脚本。

代码清单 5-33 压缩后的 JavaScript 脚本

```
var testStr="hello";var testStr2="world";function addStr(a,b){return  
a+" "+b}addStr(testStr,testStr2);
```

压缩之后 JavaScript 文件的确变小了，但影响了代码的可读性，如此紧凑的代码，很难阅读，对于维护非常不利。所以除了压缩后的文件，我们还需要保留压缩前的原始文件，以备维护之需。

JavaScript 本身是无须编译的，放入浏览器中即可运行。本节标题中所说的编译，并不是指类似于 Java 那样的编译，而是指代码的压缩。JavaScript 应该建立起编译的概念，在开发的过程中，我们加上注释，注意换行和缩进，让代码的可读性良好，但这样的文件只算做源文件，不要直接用它来发布，正式发布的文件我们需要对源文件进行一下“编译”（压缩），编译过后的文件可读性差，但它可以减小文件的大小。如果 JavaScript 需要修改，我们应找到源文件对它进行修改，然后将修改后的文件重新进行编译，再将编译后的文件发布。

为了方便维护，源文件的文件名和压缩后文件的文件名应建立起对应关系，比如源文件叫做 head.js，压缩后的文件可以叫做 head-min.js，以此来表明它们之间的关系。

除了压缩，我们还可以对 JavaScript 进行反压缩，<http://jsbeautifier.org/>这个工具就是一个在线的反压缩工具，它可以把压缩后的 JavaScript 文件重新格式化。但需要注意的是，它仅仅是从缩进上对 JavaScript 进行了反压缩，已经去掉的注释和已经改名的变量是没办法恢复的。

## 5.2 JavaScript 的分层概念和 JavaScript 库

### 5.2.1 JavaScript 如何分层

分层可以让我们的代码组织条理更清晰，减少冗余，提高代码重用率。和 CSS 一

样，我把 JavaScript 也分成三层，从下往上依次是 **base** 层、**common** 层和 **page** 层。需要说明的是，如何分层是一件主观的事情，为的是“把事情变得更好”，没有对和错之分，只有好和坏，我并不是说“只有这样分层才是对的”，我在这里分享的是我自己的经验，推荐给大家参考。

### 1. **base** 层

位于三层的最底端，这一层有两个职责。职责一是封装不同浏览器下 JavaScript 的差异，提供统一的接口，我们可以依靠它来完成跨浏览器兼容的工作。职责二是扩展 JavaScript 语言底层提供的接口，让它提供更多更为易用的接口。**base** 层的功能是给 **common** 层和 **page** 层提供接口。

### 2. **common** 层

位于三层的中间，依赖于 **base** 层提供的接口。**common** 层提供可供复用的组件，它是典型的 mvc 模式中的 **m**，和页面内的具体功能没有直接关系。**common** 层的功能是给 **page** 层提供组件。

### 3. **page** 层

位于三层的最顶端。这一层和页面里的具体功能需求直接相关，是 mvc 模式中的 **c**。**page** 层依赖于 **base** 层和 **common** 层。**page** 层的功能是完成页面内的功能需求。

## 5.2.2 **base** 层

具体来说，**base** 层应该包含哪些内容呢？

**base** 层的功能是为 **common** 层和 **page** 层提供接口。接口由两部分组成，与这一层的两个职责相对应。职责一是“封装不同浏览器下 JavaScript 的差异，提供统一的接口”。让我们先来看一段代码，如代码清单 5-34 所示。

代码清单 5-34 简单的代码

---

```
<ul>
  <li id="item1"></li>
  <li id="item2"></li>
  <li id="item3"></li>
```

```
</ul>
<script type="text/JavaScript">
    var item1=document.getElementById("item1");
    alert(item1.nextSibling.id);
</script>
```

这段代码在 IE 下弹出 item2，在 Firefox 下弹出 undefined。这是为什么呢？我们把代码修改一下，如代码清单 5-35 所示。

代码清单 5-35 查看 IE 和 firefox 下的区别

```
<ul>
    <li id="item1"></li>
    <li id="item2"></li>
    <li id="item3"></li>
</ul>
<script type="text/JavaScript">
    var item1=document.getElementById("item1");
    alert(item1.nextSibling.nodeType);
    alert(document.getElementsByTagName("ul")[0].childNodes.length);
</script>
```

在 IE 下，会弹出 1 和 3，表明 item1 的 nextSibling 类型为 DOM 节点，ul 里的子节点个数为 3。在 Firefox 下，会弹出 3 和 7，表明 item1 的 nextSibling 类型为文本节点，ul 里的子节点个数为 7。找到问题了，Firefox 会将包括空白、换行等文本信息也当做 childNodes 中的一员，而 IE 则会忽略它，只将 DOM 节点当做是 childNodes 的一员。如此一来，我们编写的程序就无法兼容 IE 和 Firefox 了。

一般情况下，我们更支持 IE 的解析方式，只视 DOM 节点为 childNodes 中的一员。如何解决这个问题，让 IE 和 Firefox 下表现一致呢？方案一如代码清单 5-36 所示。

代码清单 5-36 去掉空格

```
<ul><li id="item1"></li><li id="item2"></li><li id="item3"></li></ul>
<script type="text/JavaScript">
    var item1=document.getElementById("item1");
    alert(item1.nextSibling.id);
</script>
```

我们可以将所有的空白和换行去掉，这样 IE 和 Firefox 下弹出的结果就会一致了。但这种解决方案绑定了 HTML 的结构，让程序和 HTML 产生了耦合。

方案二如代码清单 5-37 所示。

## 代码清单 5-37 分别对 IE 和 Firefox 写程序

```

<ul>
    <li id="item1"></li>
    <li id="item2"></li>
    <li id="item3"></li>
</ul>
<script type="text/JavaScript">
    var item1=document.getElementById("item1");
    var nextNode=item1.nextSibling;
    //Firefox 不支持 document.all. 这里用以判别浏览器类型
    if(!document.all){
        while(true){
            if(nextNode.nodeType==1){
                break;
            } else {
                if(nextNode.nextSibling){
                    nextNode=nextNode.nextSibling;
                } else {
                    break;
                }
            }
        }
    }
    alert(nextNode.id);
</script>

```

`document.all` 是 IE 支持的属性，Firefox 不支持，我们先通过 `document.all` 判断浏览器的种类，如果是 Firefox，我们取 `nextSibling` 时一直往下查询，直到找到下一个 DOM 节点为止。这是一种解决方案，但它的缺点也很明显，一方面它使代码的可读性变差了，另一方面，如果在别的地方也需要访问 DOM 节点的 `nextSibling`，我们又要再写一遍 `if` 判断，代码的重用性不高，如代码清单 5-38 所示。

## 代码清单 5-38 针对 IE 和 Firefox 分别写程序如何面对扩展

```

<ul>
    <li id="item1"></li>
    <li id="item2"></li>
    <li id="item3"></li>
</ul>
<script type="text/JavaScript">
    var item1=document.getElementById("item1");
    var nextNode=item1.nextSibling;
    if(!document.all){
        while(true){
            if(nextNode.nodeType==1){
                break;
            } else {

```

```

        if(nextNode.nextSibling) {
            nextNode=nextNode.nextSibling;
        } else {
            break;
        }
    }
}
alert(nextNode.id);

var item2=document.getElementById("item2");
var nextNode2=item2.nextSibling;
if(!document.all){
    while(true){
        if(nextNode2.nodeType==1){
            break;
        } else {
            if(nextNode2.nextSibling){
                nextNode2=nextNode2.nextSibling;
            } else {
                break;
            }
        }
    }
}
alert(nextNode2.id);
</script>

```

方案三如代码清单 5-39 所示。

代码清单 5-39 用 getNextNode 函数封装 IE 和 Firefox 的差异

---

```

<ul>
    <li id="item1"></li>
    <li id="item2"></li>
    <li id="item3"></li>
</ul>
<script type="text/JavaScript">
    function getNextNode(node){
        node=typeof node=="string" ? document.getElementById(node) : node;
        var nextNode=node.nextSibling;
        if(!nextNode) return null;
        if(!document.all){
            while(true){
                if(nextNode.nodeType==1){
                    break;
                } else {
                    if(nextNode.nextSibling){
                        nextNode=nextNode.nextSibling;
                    } else {
                        break;
                    }
                }
            }
        }
    }

```

```

        }
    return nextNode;
};

var nextNode=getNextNode("item1");
alert(nextNode.id);
var nextNode2=getNextNode("item2");
alert(nextNode2.id);
</script>

```

方案三的思路是定义一个函数 `getNextNode`，函数接收 DOM 节点或者 DOM 节点的 `id` 作为参数，返回我们需要的 `nextSibling`。在需要 `nextSibling` 的地方，我们不再直接调用原生 JS 的 `nextSibling` 属性，而是调用我们定义的 `getNextNode` 函数。`getNextNode` 函数内部对浏览器类型做了 `if` 判断，做了大量工作，但它本身充当的是 `mvc` 中 `m` 的角色，供 `c` 调用。对于 `c` 来说，`m` 的内部实现是透明的。由于分开了 `m` 和 `c` 的角色，所以 `c` 的代码变得异常简洁清晰，而 `m` 则可以被反复调用，提高了代码的重用率。通过 `getNextNode` 函数，我们轻松地实现了跨浏览器的兼容，这里的 `getNextNode` 函数，就是我所说的接口。`base` 层的作用之一就是提供大量类似的接口，用以屏蔽原生 JavaScript 在不同浏览器下的差异。

下面我们来看看其他一些典型的 JavaScript 兼容问题。

### 1. 透明度

透明度问题与 `nextSibling` 类似，IE 下透明是通过滤镜实现的，而在 Firefox 下透明是通过 CSS 的 `opacity` 属性实现的。如果不封装接口，如代码清单 5-40 所示。

**代码清单 5-40 针对 IE 和 Firefox 分别设置透明度**

```

<style type="text/CSS">
#test1{background:blue;height:100px;}
#test2{background:green;height:100px;}
</style>
<div id="test1"></div>
<div id="test2"></div>
<script type="text/JavaScript">
var test1=document.getElementById("test1"),test2=document.getElementById
("test2");
if (document.all){
    test1.style.filter='alpha(opacity=20)';
    test2.style.filter='alpha(opacity=80)';
} else {
    test1.style.opacity=0.2;
}
</script>

```

```

        test2.style.opacity=0.8;
    }
</script>

```

将设置透明度功能封装成 setOpacity 函数，再来看看代码，如代码清单 5-41 所示。

代码清单 5-41 封装 setOpacity 函数

```

<style type="text/CSS">
#test1{background:blue;height:100px;}
#test2{background:green;height:100px;}
</style>
<div id="test1"></div>
<div id="test2"></div>
<script type="text/JavaScript">
function setOpacity(node, level){
    node=typeof node=="string" ? document.getElementById(node) : node;
    if (document.all){
        node.style.filter='alpha(opacity=' + level + ')';
    } else {
        node.style.opacity=level / 100;
    }
}
setOpacity("test1",20);
setOpacity("test2",80);
</script>

```

## 2. event 对象

不同浏览器中除了 DOM 的表现会有所不同，事件的表现也有很大的不同。在 IE 下，event 对象是作为 window 的属性作用于全局作用域的，而在 Firefox 中，event 对象是作为事件的参数存在的，如代码清单 5-42 所示。

代码清单 5-42 点击事件和参数

```

<script type="text/JavaScript">
var btn=document.getElementById("btn");
btn.onclick=function(){
    alert(arguments.length);
}
</script>

```

这段代码在 IE 下会弹出 0，在 Firefox 下会弹出 1，它就是 event 对象。为了兼容 IE 和 Firefox，通常我们会用一个变量指向 event 对象，然后通过这个变量来访问 event 对象，如代码清单 5-43 所示。

**代码清单 5-43 兼容 IE 和 Firefox 的 event 对象**


---

```
<script type="text/JavaScript">
var btn=document.getElementById("btn");
btn.onclick=function(e){
    e=window.event || e;
    //下面可以用 e 来做点什么事，e 在 IE 和 Firefox 下都指向了 event 对象
}
</script>
```

---

event 对象的部分属性名在 IE 和 Firefox 下也是不同的，比如派生事件的对象在 IE 下是通过 event 对象的 srcElement 属性访问的，而在 Firefox 下，是通过 event 对象的 target 属性访问的，如代码清单 5-44 所示。

**代码清单 5-44 兼容 srcElement 和 target**


---

```
<span id="span">hello world</span>
<script type="text/JavaScript">
document.getElementById("btn").onclick=function(e) {
    e=window.event || e;
    var el=e.srcElement || e.target;
    alert(el.tagName);
}
document.getElementById("span").onclick=function(e) {
    e=window.event || e;
    var el=e.srcElement || e.target;
    alert(el.tagName);
}
</script>
```

---

在 IE 和 Firefox 下点击按钮都能弹出 INPUT，点击 span 弹出 SPAN。

对 event 对象，我们也可以封装一下，将上面的代码进行修改，如代码清单 5-45 所示。

**代码清单 5-45 封装 getEventTarget 函数**


---

```
<span id="span">hello world</span>
<script type="text/JavaScript">
function getEventTarget(e){
    e=window.event || e;
    return e.srcElement || e.target;
}
document.getElementById("wrapper").onclick=function(e) {
    var node=getEventTarget(e);
    alert(node.tagName);
}
```

---

```

document.getElementById("span").onclick=function(e) {
    var node=getEventTarget(e);
    alert(node.tagName);
}
</script>

```

可以通过我们封装的 `getEventTarget` 函数来得到派生事件的对象。

### 3. 冒泡

先来看一下代码清单 5-46。

代码清单 5-46 event 对象的冒泡

```

<style type="text/CSS">
#infoBox{padding:5px;border:2px dashed #ccc;margin-bottom:10px;width:136px;height:20px;line-height:20px;text-align:center;}
#wrapper{padding:20px 0;background:black;width:150px;text-align:center;}
</style>
<p id="infoBox"></p>
<div id="wrapper">
    <input type="button" value="click me" id="btn" />
</div>
<script type="text/JavaScript">
var infoBox=document.getElementById("infoBox"), wrapper=document.getElementById("wrapper"), btn=document.getElementById("btn");
wrapper.onclick=function() {
    infoBox.innerHTML="你点击的是: div";
}
btn.onclick=function() {
    infoBox.innerHTML="你点击的是: input";
}
</script>

```

效果如图 5-2 所示。

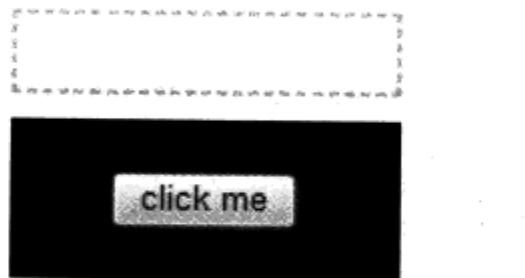


图 5-2 冒泡事件截图

我们给 `div` 和 `input` 分别监听 `click` 事件，希望在点击黑色区域的时候，虚线框内显

示“你点击的是：div”，点击按钮的时候，显示“你点击的是：input”。

但事实上，不论我们点击黑色区域还是按钮，虚线框内显示的都是“你点击的是：div”，这是为什么呢？因为 input 包含在 div 内，点击 input 时，其实也点击到了 div，这时会先触发 input 的点击事件，再触发 div 的点击事件，虚线框内会先显示“你点击的是：input”，然后非常快地又显示成“你点击的是：div”，因为速度很快，所以我们察觉不到虚线框内显示过“你点击的是 input”。JavaScript 对这种先触发子容器监听事件，后触发父容器监听事件的现象称为事件的冒泡。

现在我们了解了事件的触发过程，我们将代码修改一下，如代码清单 5-47 所示。

代码清单 5-47 监测冒泡的过程

---

```

<style type="text/CSS">
    #infoBox{padding:5px; border:2px dashed #ccc; margin-bottom:10px; width:136px; height:20px; line-height:20px; text-align:center;}
    #wrapper{padding:20px 0; background:black; width:150px; text-align:center;}
</style>
<p id="infoBox"></p>
<div id="wrapper">
    <input type="button" value="click me" id="btn" />
</div>
<script type="text/JavaScript">
    var infoBox=document.getElementById("infoBox"), wrapper=document.getElementById("wrapper"), btn=document.getElementById("btn");
    wrapper.onclick=function() {
        infoBox.innerHTML="你点击的是：div";
        alert(1);
    }
    btn.onclick=function() {
        infoBox.innerHTML="你点击的是：input";
        alert(2);
    }
</script>

```

---

当我们点击黑色区域的时候，会弹出 1，点击按钮的时候，会先弹出 2，之后又弹出 1。

如何解决这个问题呢？我们需要阻止点击按钮时的事件冒泡。阻止事件冒泡在 IE 下是通过设置 event 对象的 cancelBubble 属性为 true 实现的，而在 Firefox 下则是通过调用 event 对象的 stopPropagation 方法实现的。如代码清单 5-48 所示。

## 代码清单 5-48 阻止事件冒泡

```

<style type="text/CSS">
#infoBox{padding:5px; border:2px dashed #ccc; margin-bottom:10px; width:136px; height:20px; line-height:20px; text-align:center;}
#wrapper{padding:20px 0; background:black; width:150px; text-align:center;}
</style>
<p id="infoBox"></p>
<div id="wrapper">
    <input type="button" value="click me" id="btn" />
</div>
<script type="text/JavaScript">
var infoBox=document.getElementById("infoBox"), wrapper=document.getElementById("wrapper"), btn=document.getElementById("btn");
wrapper.onclick=function(){
    infoBox.innerHTML="你点击的是: div";
}
btn.onclick=function(e){
    infoBox.innerHTML="你点击的是: input";
    e=window.event || e;
    if(document.all){
        e.cancelBubble=true;
    } else {
        e.stopPropagation();
    }
}
</script>

```

由于阻止了事件的冒泡，现在点击按钮就可以在虚线框区域内正常显示“你点击的是：input”了。我们可以进一步将阻止事件冒泡的动作封装起来，如代码清单 5-49 所示。

## 代码清单 5-49 封装 stopPropagation 函数

```

<style type="text/CSS">
#infoBox{padding:5px; border:2px dashed #ccc; margin-bottom:10px; width:136px; height:20px; line-height:20px; text-align:center;}
#wrapper{padding:20px 0; background:black; width:150px; text-align:center;}
</style>
<p id="infoBox"></p>
<div id="wrapper">
    <input type="button" value="click me" id="btn" />
</div>
<script type="text/JavaScript">
function stopPropagation(e){
    e=window.event || e;
    if(document.all){
        e.cancelBubble=true;
    } else {
        e.stopPropagation();
    }
}

```

```

        }
    }
    var infoBox=document.getElementById("infoBox"), wrapper=document.get-
ElementById("wrapper"), btn=document.getElementById("btn");
    wrapper.onclick=function(){
        infoBox.innerHTML="你点击的是: div";
    }
    btn.onclick=function(e){
        infoBox.innerHTML="你点击的是: input";
        stopPropagation(e);
    }
</script>

```

---

#### 4. on、attachEvent 和 addEventListener

如果我们要让某个 DOM 节点监听事件，最简单方法就是使用 `on xxx` 方法，如代码清单 5-50 所示。

代码清单 5-50 监听 click 事件

```

<script type="text/JavaScript">
var btn=document.getElementById("btn");
btn.onclick=function(){
    alert(1);
}
</script>

```

---

按钮监听 `click` 事件，点击时弹出 1。如果我们需要让 `click` 事件再触发另一个监听函数，如代码清单 5-51 所示。

代码清单 5-51 监听两个 click 事件

```

<script type="text/JavaScript">
var btn=document.getElementById("btn");
btn.onclick=function(){
    alert(1);
}
btn.onclick=function(){
    alert(2);
}
</script>

```

---

我们希望点击按钮时，能将两个监听函数都触发，先弹出 1，接着弹出 2。但事实上，后面的 `btn.onclick` 会把前面的 `btn.onclick` 覆盖掉，点击按钮时，只会弹出 2。

`on xxx` 的监听方式没有叠加的效果，最后定义的 `on xxx` 会将前面的覆盖掉。所以

这样的写法其实是非常危险的，特别是多人合作时。如代码清单 5-52 所示。

代码清单 5-52 on xxx 在多人合作时的冲突问题

---

```
<script type="text/JavaScript">
//=====
// 功能 A , made by 工程师甲
//=====
(function() {
    ...
    var btn=document.getElementById("btn");
    btn.onclick=function(){
        alert(1);
    }
    ...
})();
//=====
// 功能 B , made by 工程师乙
//=====
(function() {
    ...
    var btn=document.getElementById("btn");
    btn.onclick=function(){
        alert(2);
    }
    ...
})();
</script>
```

---

工程师甲和工程师乙分别编写了一段程序，都让按钮监听了 click 事件。如果单独只有工程师甲的程序或单独只有工程师乙的程序，运行是正常的，但如果同时将两段程序放入页面中，工程师甲和工程师乙的程序就会产生冲突，工程师乙无意中覆盖掉了工程师甲给按钮监听的 click 处理函数！不仅是全局变量会让多人合作中产生意料外的冲突，on xxx 监听也会产生冲突。

为了解决这个问题，可以使用 attachEvent 和 addEventListener 方法来代替 on xxx 监听事件。其中 attachEvent 是 IE 支持的方法，而 addEventListener 是 Firefox 支持的方法。attachEvent 和 addEventListener 方法支持监听处理函数的叠加，而不是覆盖，用 attachEvent 和 addEventListener 改写代码清单 5-51 的代码，如代码清单 5-53 所示：

代码清单 5-53 使用 attachEvent 和 addEventListener 代替 on xxx

---

```
<script type="text/JavaScript">
var btn=document.getElementById("btn");
if(document.all){
```

```

btn.attachEvent("onclick",function(){
    alert(1);
});
} else {
    btn.addEventListener("click",function(){
        alert(1);
    },false);
}
if(document.all){
    btn.attachEvent("onclick",function(){
        alert(2);
    });
} else {
    btn.addEventListener("click",function(){
        alert(2);
    },false);
}
</script>

```

在 IE 和 Firefox 下点击按钮都可以顺利地弹出 1 和 2。只是现在的代码可读性太差了，让我们将 `attachEvent` 和 `addEventListener` 封装一下吧，如代码清单 5-54 所示。

代码清单 5-54 封装 on 函数

```

<script type="text/JavaScript">
function on(node,eventType,handler){
    node=typeof node=="string" ? document.getElementById(node) : node;
    if(document.all){
        node.attachEvent("on"+eventType,handler);
    } else {
        node.addEventListener(eventType,handler,false);
    }
}
var btn=document.getElementById("btn");
on(btn,"click",function(){
    alert(1);
});
on(btn,"click",function(){
    alert(2);
})
</script>

```

`base` 层的第二个职责是“扩展 JavaScript 语言底层的接口”。原生 JavaScript 底层提供的接口不算丰富，比如，虽然 JavaScript 支持面向对象，但却没有为继承提供 `extend` 方法，它也没有类似于 PHP 的类型判断的函数，也不提供 `getElementsByClassName` 方法等。

我们可以自定义一些常用的方法来弥补原生 JavaScript 的缺漏。下面简单列举几个

常用的方法：

(1) trim ()

在做表单验证的时候，我们常常需要检查输入框是否为空，一个简单地做法是判断 `inputNode.value == ""`，但如果输入空格、缩进等字符，就可以绕过这个判断了。PHP 有个很好用的函数 `trim` 可以去除字符串首尾的空白字符，虽然原生 JavaScript 并没有提供这个函数，但我们可以自定义这么一个函数。如代码清单 5-55 所示：

代码清单 5-55 定义 trim 函数

---

```
function trim(ostr){  
    return ostr.replace(/^\s+|\s$/g,"");  
}  
var str=" abc ";  
alert(trim(str).length);           // 3  
alert(" "=="");                  // false  
alert(trim(" ")=="");            // true
```

---

(2) isNumber、isString、isBoolean、isFunction、isNull、isUndefined、isEmpty、isArray

添加类型判断的方法，如代码清单 5-56 所示。

代码清单 5-56 定义类型判断函数

---

```
function isNumber(s){  
    return !isNaN(s);  
}  
function isString(s){  
    return typeof s==="string";  
}  
function isBoolean(s){  
    return typeof s==="boolean";  
}  
function isFunction(s){  
    return typeof s==="function";  
}  
function isNull(s){  
    return s==null;  
}  
function isUndefined(s){  
    return typeof s==="undefined";  
}  
function isEmpty(s){  
    return /^\s*$/.test(s);  
}  
function isArray(s){
```

---

```

        return s instanceof Array;
    }

```

### (3) get()、\$()

在写 JavaScript 代码时，最常用的函数就是 `document.getElementById()`，但它实在太长了，写起来很麻烦，所以很多人干脆用一个简单的函数名来代替它，用得比较多的是 `get()` 或者 `$()`。还可以再稍微往前一步，参数除了 DOM 节点 id 外，还可以直接使用 DOM 节点本身，如代码清单 5-57 所示。

代码清单 5-57 定义 get 和 \$ 函数

```

function get(node) {
    node=typeof node=="string" ? document.getElementById(node) : node;
    return node;
}
function $(node) {
    node=typeof node=="string" ? document.getElementById(node) : node;
    return node;
}
alert(get("test1").innerHTML);           // Hello
alert($( "test2" ).innerHTML)           // World

```

### (4) getElementsByClassName

`getElementsByClassName` 是个非常有用的函数，但可惜的是原生 JavaScript 却并没有提供它。没关系，我们自定义一个，如代码清单 5-58 所示。

代码清单 5-58 定义 getElementsByClassName 函数

```

<span class="a">1</span>
<span class="a">2</span>
<p class="a">3</p>
<div class="b">4</div>
<strong class="b">5</strong>
<div id="wrapper"><strong class="b">6</strong></div>
<script type="text/JavaScript">
function getElementsByClassName(str,root,tag){
    if(root){
        root=typeof root=="string" ? document.getElementById(root) : root;
    } else {
        root=document.body;
    }
    tag=tag || "*";
    var els=root.getElementsByTagName(tag),arr=[];
    for(var i=0,n=els.length;i<n;i++){
        for(var j=0,k=els[i].className.split(" "),l=k.length;j<l;j++){
            if(k[j]==str){

```

```

        arr.push(els[i]);
        break;
    }
}
return arr;
}
var aEls=getElementsByClassName("a"), bEls=getElementsByClassName("b",
"wrapper"), bEls2=getElementsByClassName("b",null,"strong");
alert(aEls.length);           // 3 (1,2,3)
alert(bEls.length);           // 1 (6)
alert(bEls2.length);          // 2 (5,6)

```

`getElementsByClassName` 函数接收三个参数，其中第一个参数是必选的，后两个参数可选。第一个参数是 `class` 名，第二个参数是父容器，缺省为 `body` 节点，第三个参数为 DOM 节点的标签名。

#### (5) extend

JavaScript 是支持面向对象的语言，但它并不提供 `extend` 方法用于继承。我们需要自己定义 `extend` 方法，如代码清单 5-59 所示。

代码清单 5-59 定义 `extend` 函数

```

function extend(subClass, superClass) {
    var F=function() {};
    F.prototype=superClass.prototype;
    subClass.prototype=new F();
    subClass.prototype.constructor=subClass;
    subClass.superclass=superClass.prototype;
    if(superClass.prototype.constructor==Object.prototype.constructor){
        superClass.prototype.constructor=superClass;
    }
}

function Animal(name){
    this.name=name;
    this.type="animal"
}
Animal.prototype={
    say : function(){
        alert("I'm a(an) " + this.type + ", my name is " + this.name);
    }
}
function Bird(name){
    this.constructor.superclass.constructor.apply(this,arguments);
    this.type="bird";
}
extend(Bird,Animal);
Bird.prototype.fly=function(){

```

---

```

        alert("I'm flying");
    }
var canary=new Bird("xiaocui");
canary.say();           // I'm a(an) bird, my name is xiaocui
canary.fly();           // I'm flying

```

---

更多关于 JavaScript 面向对象的知识，详见 5.4 节。

以上列举的是部分常用接口，它并不是全部，我们可以根据实际情况自定义更多类似的接口。

在 5.1.2 节中我们将网页里的 JavaScript 分成了两大部分：框架级的代码和应用级的代码，很明显，base 层的 JavaScript 和页面里的具体应用逻辑无关，是属于框架级的，它提供的接口需要供全局作用域调用。前面我们讲过，应该尽量使用命名空间来减少全局作用域变量的个数，所以 base 层的代码也应当加上命名空间。base 层的接口从大的方向上来看，可以分成三块，一块用来操作 DOM，包括获取 DOM 节点和设置 DOM 属性，一块用来操作事件，包括访问 event 对象的属性和设置事件监听，还有一块用来模仿其他语言提供原生 JavaScript 不提供的函数，比如 trim、extend、isNumber。据此，我们可以分别用 GLOBAL.Dom，GLOBAL.Event，GLOBAL.Lang 作为 base 层函数的命名空间，如代码清单 5-60 所示：

代码清单 5-60 使用命名空间

---

```

var GLOBAL={};
GLOBAL.namespace=function(str){
    var arr=str.split("."),o=GLOBAL;
    for (i=(arr[0]==="GLOBAL") ? 1 : 0; i<arr.length; i++) {
        o[arr[i]]=o[arr[i]] || {};
        o=o[arr[i]];
    }
}

//Dom 相关
GLOBAL.namespace("Dom");
GLOBAL.Dom.getNextNode=function(node){
    node=typeof node=="string" ? document.getElementById(node) : node;
    var nextNode=node.nextSibling;
    if(!nextNode) return null;
    if(!document.all){
        while(true){
            if(nextNode.nodeType==1){
                break;
            } else {
                if(nextNode.nextSibling){

```

```

        nextNode=nextNode.nextSibling;
    } else {
        break;
    }
}
return nextNode;
};

GLOBAL.Dom.setOpacity=function(node, level){
    node=typeof node=="string" ? document.getElementById(node) : node;
    if (document.all){
        node.style.filter='alpha(opacity=' + level + ')';
    } else {
        node.style.opacity=level / 100;
    }
}

GLOBAL.Dom.get=function(node){
    node=typeof node=="string" ? document.getElementById(node) : node;
    return node;
}

GLOBAL.Dom.getElementsByClassName=function(str,root,tag){
    if(root){
        root=typeof root=="string" ? document.getElementById(root) : root;
    } else {
        root=document.body;
    }
    tag=tag || "*";
    var els=root.getElementsByTagName(tag),arr=[];
    for(var i=0,n=els.length;i<n;i++){
        for(var j=0,k=els[i].className.split(" "),l=k.length;j<l;j++){
            if(k[j]==str){
                arr.push(els[i]);
                break;
            }
        }
    }
    return arr;
}

//Event 相关
GLOBAL.namespace("Event");
GLOBAL.Eevent.getEventTarget=function(e){
    e=window.event || e;
    return e.srcElement || e.target;
}
GLOBAL.Event.stopPropagation=function(e){
    e=window.event || e;
    if(document.all){
        e.cancelBubble=true;
    } else {
        e.stopPropagation();
    }
}

```

```

GLOBAL.Event.on=function(node,eventType,handler){
    node=typeof node=="string" ? document.getElementById(node) : node;
    if(document.all){
        node.attachEvent("on"+eventType,handler);
    } else {
        node.addEventListener(eventType,handler,false);
    }
}

//Lang 相关
GLOBAL.namespace("Lang");
GLOBAL.Lang.trim=function(ostr){
    return ostr.replace(/^\s+|\s+$/g,"");
}
GLOBAL.Lang.isNumber=function(s){
    return !isNaN(s);
}
GLOBAL.Lang.extend=function(subClass,superClass){
    var F=function() {};
    F.prototype=superClass.prototype;
    subClass.prototype=new F();
    subClass.prototype.constructor=subClass;
    subClass.superclass=superClass.prototype;
    if(superClass.prototype.constructor==Object.prototype.constructor){
        superClass.prototype.constructor=superClass;
    }
}

```

在实际应用中，我们可以将 `base` 层的代码放在一个 JS 文件中，例如 `base.js`，然后通过`<script src="base.js"></script>`链接到网页中。如果对代码的大小要求很高，还可以将 `base.js` 分成粒度更小的 `base_dom.js`、`base_event.js` 和 `base_lang.js`，按需导入。

### 5.2.3 common 层

`common` 层本身依赖 `base` 层提供的接口，所以如果要使用 `common` 层的组件，必须先加载 `base` 层的代码，如代码清单 5-61 所示。

代码清单 5-61 引入 `common.js`

---

```

<script type="text/JavaScript" src="base.js"></script>
<script type="text/JavaScript" src="common.js"></script>

```

---

`common` 层和 `base` 层都是供 `page` 层调用的，二者的区别在于 `common` 层提供的不是简单的接口，而是相对更庞大的组件。例如 `cookie`，原生 JavaScript 对设置和读取 `cookie` 的方法显得非常的笨拙！设置 `cookie` 的方法如代码清单 5-62 所示。

**代码清单 5-62 设置 cookie**


---

```
document.cookie="cookieName=cookieValue; expirationdate=timeValue; path=
pathValue";
```

---

其中 `cookieName` 是我们要保存在 `cookie` 中的键，`cookieValue` 是对应于键的值，`expirationdate` 表示过期时间，`timeValue` 就是要设置的过期时间，它必须是 GMT 格式，`path` 表示可访问此 `cookie` 的域，`pathValue` 就是具体的路径值。

读取 `cookie` 更是件痛苦的事！`cookie` 无法直接读取某个键中保存的值，只能从 `document.cookie` 中将所有的键值对全部读出来，再使用 `split`、`indexOf`、`slice` 等方法操作字符串截取自己需要的键的值！

例如，先在 `cookie` 中保存一下我的个人信息：`name`、`sex`、`blog`，然后读取其中的 `name`，代码如代码清单 5-63 所示。

**代码清单 5-63 读写 cookie**


---

```
// 设置 cookie
document.cookie="name=adang; expires= Mon, 30 Nov 2009 05:49:47 GMT;
path=/";
document.cookie="sex=male; expires= Mon, 30 Nov 2009 05:49:47 GMT;
path=/";
document.cookie="blog=http://www.adanghome.com; expires= Mon, 30 Nov
2009 05:49:47 GMT; path=/";
/* 读取 cookie
** 此时 cookie 里的值为"name=adang; sex=male; blog=http://www.adanghome.com"
*/
var cookieStr=document.cookie;
// 对字符进行操作，取出 name 对应的值
var name=cookieStr.split("name")[1].split(";")[0].split("=")[1];
```

---

十分繁琐的操作！如果 JavaScript 也能像 PHP 操作 `cookie` 那样简单就好了，用 `SetCookie("cookieName", "cookieValue", "timeValue")` 设置，用 `$cookieName` 读取多方便啊！没关系，我们自己来封装一个用于操作 `cookie` 的组件，提供类似于 PHP 那样简单易用的操作 `cookie` 的接口，如代码清单 5-64 所示。

**代码清单 5-64 封装 cookie 组件**


---

```
GLOBAL.namespace("Cookie");
GLOBAL.Cookie={
    // 读取
```

```

read : function(name){
    var cookieStr="; "+document.cookie+" ";
    var index=cookieStr.indexOf("; "+name+"=");
    if (index!=-1){
        var s=cookieStr.substring(index+name.length+3,cookieStr.length);
        return unescape(s.substring(0, s.indexOf(";" )));
    }else{
        return null;
    }
};

// 设置
set : function(name,value,expires){
    var expDays=expires*24*60*60*1000;
    var expDate=new Date();
    expDate.setTime(expDate.getTime()+expDays);
    var expString=expires ? ";" : expires="+expDate.toGMTString() : "";
    var pathString=";path=/";
    document.cookie=name + "=" + escape(value) + expString + pathString;
};

// 删除
del : function(name){
    var exp=new Date(new Date().getTime()-1);
    var s=this.read(name);
    if(s!=null) {document.cookie=name+"="+s+";expires="+exp.toGMTString()+
    ";path=/";}
}
};

```

---

我们定义的 GLOBAL.Cookie 组件封装原生 JavaScript 对 cookie 进行的操作，为用户提供了更好用的 read、set 和 del 方法。对用户来说，并无需了解 GLOBAL.Cookie 的内部实现，不用和原生 JavaScript 中 cookie 相关的 API 打交道，只需要知道 GLOBAL.Cookie 提供了哪些接口就可以了。

除了 Cookie，common 层常见的组件还有用于异步通信的 Ajax、用于拖曳的 Drag，用于拖拉元素大小的 Resize，用于动画的 Animation，用于标签切换的 Tab，用于树型目录的 Tree，用于模拟弹出窗口的 Msg，用于拾色器的 ColorPicker，用于日历的 Calendar，用于富文本编辑器的 RichTextEditor 等。本书的目的在于给读者一个框架的概念，这些组件的代码我就不一一列举了。

base 层提供的接口和任何具体的功能无关，非常底层，所以也非常通用。common 层提供的组件并不像 base 层那么通用，它和具体的功能相关，如果页面里不需要相关的功能，就没必要加载。而且一个易用性、重用性和可扩展性都非常好的组件，代码量

往往偏高，所以 common 层的 JS 需要按功能分成一个个单独的文件，例如 common\_cookie.js、common\_drag.js、common\_tab.js 等，按需加载。

common 层的组件不仅依赖 base 层接口，有时也依赖 common 层的其他组件，例如模拟弹出窗口的 Msg 组件，如果需要拖曳功能的话，可以调用拖曳组件 Drag，这时它就依赖 Drag 组件了，在加载时注意导入 JS 文件的顺序，如代码清单 5-65 所示。

代码清单 5-65 引用可拖曳的 Msg 组件

```
<script type="text/JavaScript" src="base.js"></script>
<script type="text/JavaScript" src="common_drag.js"></script>
<script type="text/JavaScript" src="common_msg.js"></script>
```

#### 5.2.4 page 层

page 层是 mvc 中的 c，和页面里的具体功能需求直接相关。如果页面里的功能需求很简单，页面里可以没有 base 层代码，可以没有 common 层代码，但一定会有 page 层代码。base 层和 common 层都是属于框架级的，page 层是属于应用级的，它可以调用 base 层的接口和 common 层的组件。

我们平时绝大部分工作都是在 page 层完成的，有了分层的概念，page 层的工作就非常轻松了。如果 base 层和 common 层足够丰富和稳定，我们就可以将精力全部放在具体业务逻辑上，跨浏览器兼容和常用组件都可以交给 base 层和 common 层去完成，一方面可以极大地提高开发效率，另一方面也可以大大提高代码的重用率，减小网页的大小。

如果没有 base 层和 common 层，直接用原生 JavaScript 写程序，就像是开着汽车行驶在一条不平坦的泥路上，一路上有凸起的石块（浏览器兼容带来的阻碍），也有凹陷的沼泽（原生 JavaScript 提供的底层接口不足）。如图 5-3 所示。



图 5-3 用原生 JavaScript 写程序

引入 `base` 层，就像铲平了泥路上凸起的石块（提供跨浏览器兼容的接口），填满了凹陷的沼泽（弥补原生 JavaScript 底层接口的不足）。在 `base` 层的基础上写程序，就像汽车行驶在一条平坦的泥路上，如图 5-4 所示。



图 5-4 `base` 层铺平道路

继 `base` 层之后，再引入 `common` 层，就像在平坦的泥路上铺上了一层沥青，路再也不是普通的泥路了，它成了高速公路。在 `base` 层和 `common` 的基础上写 JavaScript，就像开着汽车在高速公路上飞驰！如图 5-5 所示。

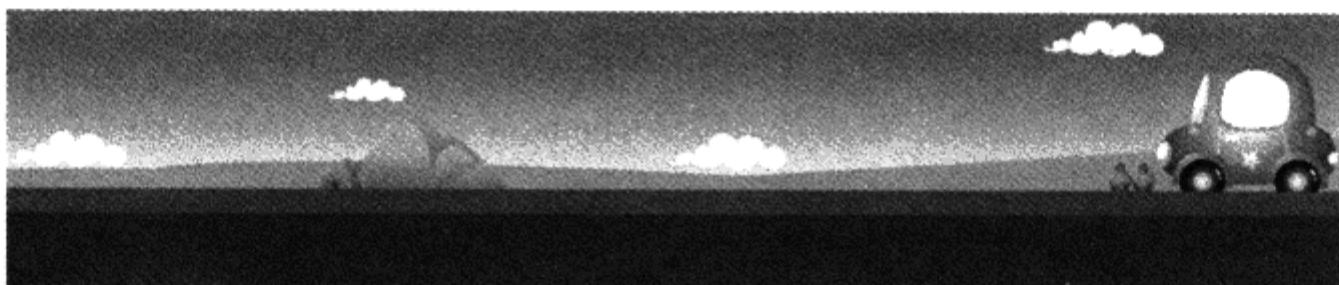


图 5-5 `common` 层给泥路铺上沥青

## 5.2.5 JavaScript 库

很多开源的 JavaScript 库可以为我们提供强大的 `base` 层和 `common` 层。最常见的 JavaScript 库有 Prototype、Dojo、Mootools、Ext JS、jQuery 和 YUI 等，各有优劣，我们可以根据需要和喜好选择其中一个，或者结合其中几个一起使用。

下面我们来看看 jQuery 和 YUI 开源 JavaScript 库是如何提供 `base` 层和 `common` 层的。

### 1. YUI

YUI 分 YUI 2 和 YUI 3 两个版本，目前 YUI 3 还没有完善和稳定，我们选择比较成

熟的 YUI 2 来分析。

YUI2 本身分成三大部分：Core、Utilities、Widgets。其中 Core 是核心部分，位于底层，它提供 Dom 相关的方法，Event 相关的方法和 YUI 全局作用域的方法，例如 YAHOO.util.Dom.getElementsByClassName、YAHOO.util.Event.getTarget、YAHOO.lang.extend。Utilities 属于组件层，提供和具体 UI 无关的功能，例如 Ajax、Drag、Resize、Cookie 等。Widgets 也属于组件层，提供和 UI 相关的功能，例如 ColorPicker、RichTextEditor 等。Core 对应 base 层，而 Utilities 和 Widgets 对应着 common 层。

## 2. jQuery

jQuery 本身分成两大部分：jQuery 核心文件和 jQuery UI 文件。jQuery UI 文件依赖 jQuery 核心文件。

jQuery 核心文件不仅提供了对 DOM 和 Event 相关方法的封装，例如`$(().find())`、`$(().on())`，提供了对原生 JavaScript 语言底层的扩展，例如`$.isArray()`，还提供了动画和 Ajax 的功能。jQuery UI 文件提供了大量常用功能，包括 Drag、Resize、Tab 等。jQuery 核心文件提供了 base 层功能，还提供了部分 common 层功能，jQuery UI 文件提供了 common 层功能。

因为 base 层和具体功能无关，只是供 common 层和 page 层调用，所以它的接口数量相对比较少，只包含了编程时最常用的部分。common 层提供各种组件，包括与 UI 无关的组件（例如 Drag、Ajax）和与 UI 相关的组件（例如 RichTextEditor），这一层可涵盖的内容非常丰富，我们可以使用官方提供的组件，也可以自己编写组件。编写组件是一件非常考验工程师能力的事，一个高品质的组件需具备以下几点：

- 跨浏览器兼容
- 组件易用
- 组件可重用
- 组件可扩展
- 代码组织有序，高内聚低耦合

前面我们一再说过，common 层依赖 base 层，在编写组件时，我们可以直接调用

`base` 层提供的接口，如果组件很复杂，还可能依赖 `common` 层的其他组件，从而提高代码的重用率，减小组件本身的代码量。正是因为这种依赖，所以我们常常可以看到网上有大量的组件标明基于某某 JavaScript 库。

坚持不使用 JavaScript 库写代码是件非常愚蠢的事情，就好比明明有强大的 IDE 可以使用，却偏偏坚持用记事本做开发。另外有些工程师喜欢使用自己编写的 JavaScript 库，我不太赞同，现在流行的开源 JavaScript 库基本上都非常强大，无论从代码品质还是易用性上都是非常好的，更重要的是开源 JavaScript 库一般都会有大量第三方为其制作组件，这样我们在开发项目时，可以直接使用第三方的组件，从而大大提高工作效率。

## 5.3 编程实用技巧

### 5.3.1 弹性

在前面的章节中，我们了解了 JavaScript 编程要养成的好习惯，以及如何组织 JavaScript 文件才能形成高效稳定的层级关系，这些都是一些大局的、概念性的東西，接下来，让我们一起深入编程的细节，掌握一些编程上的技巧。

我们以一个非常常用的组件 Tab 选项卡为例，逐步完成从粗糙到精致的转变。不同于 Drag 和 Rise 这种和具体 UI 无关的组件，Tab 和 UI 是紧密相关的。常见的 Tab 选项卡如图 5-6 所示。

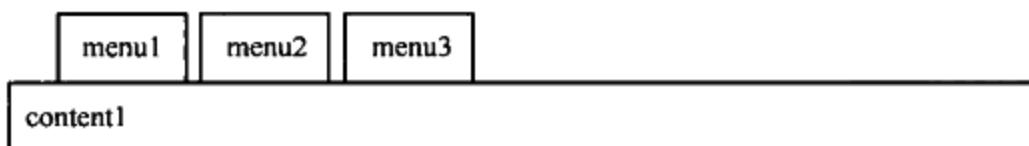


图 5-6 常见的 Tab 选项卡

它由一个标签区和一个内容区组成，当点击不同标签时，内容区显示对应的内容，标签 `menu1`、`menu2`、`menu3` 分别对应 `content1`、`content2`、`content3`。它的工作原理是什么呢？虽然我们同一时间内只能看到 `content1`、`content2` 和 `content3` 中的一个，但其

实在 DOM 结构里，它们三个都同时存在，每当我们点击某个标签时，都会将与之对应的内容显示出来，而将其他内容利用 CSS 隐藏起来。让我们快点开始动手制作吧，先来设置结构和样式，如代码清单 5-66 所示。

代码清单 5-66 开始编写 Tab 组件

---

```

<style type="text/CSS">
    ul{padding:0;margin:0}
    .tab{width:400px;}
    .tab-menuWrapper{padding-left:20px;}
    .tab-menuWrapper li{float:left;display:inline;padding:5px;border:1px solid #333;border-bottom:none;margin-right:5px;}
    .tab-contentWrapper{border:1px solid #333;clear:left;padding:5px;}
</style>
<div class="tab">
    <ul class="tab-menuWrapper">
        <li>menu1</li>
        <li>menu2</li>
        <li>menu3</li>
    </ul>
    <div class="tab-contentWrapper">
        <div>content1</div>
        <div style="display:none">content2</div>
        <div style="display:none">content3</div>
    </div>
</div>

```

---

接下来，我们为 Tab 组件编写程序。点击不同标签可以显示相应的内容，我们需要在程序中控制 DOM 节点了，获得 DOM 节点最简单的方法是 `getElementById()` 方法，我们就给 HTML 相关元素设置 id，便于程序挂钩。虽然我们可以简单地使用 `menu1`、`content1` 来设置 id，但为了让 id 更像是 Tab 组件私有的，不容易和网页中其他地方冲突（虽然我们的例子中并没有其他部分，但在真实环境中，一个网页当然不可能只存在一个 Tab），我们使用 `tab-menu1`、`tab-content1` 来命名。Tab 程序如代码清单 5-67 所示。

代码清单 5-67 给 HTML 标签挂上 id

---

```

...
<div class="tab">
    <ul class="tab-menuWrapper">
        <li id="tab-menu1">menu1</li>
        <li id="tab-menu2">menu2</li>
        <li id="tab-menu3">menu3</li>
    </ul>
    <div class="tab-contentWrapper">
        <div id="tab-content1">content1</div>

```

```

<div id="tab-content2" style="display:none">content2</div>
<div id="tab-content3" style="display:none">content3</div>
</div>
</div>
<script type="text/JavaScript">
    //获得 tabMenu 和 tabContent 的 DOM 节点，并保存在变量中
    var tabMenu1=document.getElementById("tab-menu1"),
    tabMenu2=document.getElementById("tab-menu2"),
    tabMenu3=document.getElementById("tab-menu3"),
    tabContent1=document.getElementById("tab-content1"),
    tabContent2=document.getElementById("tab-content2"),
    tabContent3=document.getElementById("tab-content3");

    //让 tabMenu 监听点击 click 事件
    tabMenu1.onclick=function(){
        //显示 tabContent1，隐藏 tabContent2 和 tabContent3
        tabContent1.style.display="block";
        tabContent2.style.display="none";
        tabContent3.style.display="none";
    }
    tabMenu2.onclick=function(){
        //显示 tabContent2，隐藏 tabContent1 和 tabContent3
        tabContent1.style.display="none";
        tabContent2.style.display="block";
        tabContent3.style.display="none";
    }
    tabMenu3.onclick=function(){
        //显示 tabContent3，隐藏 tabContent1 和 tabContent2
        tabContent1.style.display="none";
        tabContent2.style.display="none";
        tabContent3.style.display="block";
    }
</script>

```

非常好，工作得很顺利。但如果此时客户（或者老板、产品经理）说：“我需要 Tab 有 4 个标签，麻烦你改一下”。我们该怎么办呢？

在你动手添加一个 `tab-menu4` 和 `tab-content4` 之前，我们想想，如果在你修改完之后，客户说：“不好意思，我们还需要增加两个标签，一共要 6 个标签。”那我们不是得没完没了地做着重复性的工作吗？不行，我们得想个更好的主意，让程序更具有弹性，标签的数量可以自适应，不管是添加几个都无需修改 JavaScript，只简单修改 HTML 标签就可以了。很明显我们需要数组来帮忙了，将 `menu` 和 `content` 的 DOM 节点都存入数组中，然后通过对数组进行遍历来绑定监听器，至于数组中有多少个元素，我们就可以不用关心了。如何批量获得 DOM 节点呢？除了 `getElementById` 之外，我们可以用 `getElementsByName` 来获得 DOM 节点。新修

改的代码如代码清单 5-68 所示。

代码清单 5-68 提高 Tab 的扩展性

```

...
<div class="tab">
    <ul class="tab-menuWrapper" id="tab-menuWrapper">
        <li>menu1</li>
        <li>menu2</li>
        <li>menu3</li>
        <li>menu4</li>
    </ul>
    <div class="tab-contentWrapper" id="tab-contentWrapper">
        <div>content1</div>
        <div style="display:none">content2</div>
        <div style="display:none">content3</div>
        <div style="display:none">content4</div>
    </div>
</div>
<script type="text/JavaScript">
    //获得 tabMenu 和 tabContent 的节点，并以数组的形式保存在变量中
    var tabMenus=document.getElementById("tab-menuWrapper").getElements-
    ByTagName("li"),
        tabContents=document.getElementById("tab-contentWrapper").get-
    ElementsByTagName("div");

    //遍历数组，让 tabMenu 监听 click 事件
    for(var i=0;i<tabMenus.length;i++){
        tabMenus[i].onclick=function(){
            //遍历数组，隐藏所有 tabContent
            for(var j=0;j<tabContents.length;j++){
                tabContents[j].style.display="none";
            }
            //显示被点击的 tabMenu 对应的 tabContent
            tabContents[i].style.display="block";
        }
    }
</script>

```

修改之后程序更精简了，刷新页面，点击标签会发现情况并没有我们预想中那么顺利。点击之后所有 content 都隐藏了，JavaScript 报错 “tabContents[i] is undefined”。这是 JavaScript 一个非常经典的问题：在遍历数组时对 DOM 监听事件，索引值始终等于遍历结束后的值。我们简单地测试一下，在遍历的时候，弹出当前索引值，如代码清单 5-69 所示。

代码清单 5-69 循环中的 i

```
for(var i=0;i<tabMenus.length;i++){
```

```

tabMenus[i].onclick=function(){
    //弹出当前索引值
    alert(i);
    //遍历数组，隐藏所有 tabContent
    for(var j=0;j<tabContents.length;j++){
        tabContents[j].style.display="none";
    }
    //显示被点击的 tabMenu 对应的 tabContent
    tabContents[i].style.display="block";
}

```

---

当我们分别点击 tabMenu1、tabMenu2、tabMenu3、tabMenu4 时，会弹出什么呢？出乎意料的是，不会弹出 0、1、2、3，而是 4、4、4、4。所以无论当我们点击哪个标签，都会设置“tabContents[4].style.display='block'”，tabContents[4]当然是 undefined。

解决这个问题有两个方法，方法一是利用闭包，如代码清单 5-70 所示。

代码清单 5-70 解决循环 Bug 方法一

```

for(var i=0;i<tabMenus.length;i++){
    (function(_i){
        tabMenus[_i].onclick=function(){
            for(var j=0;j<tabContents.length;j++){
                tabContents[j].style.display="none";
            }
            tabContents[_i].style.display="block";
        }
    })(i);
}

```

---

方法二是给 DOM 节点添加 `_index` 属性，属性值等于索引，如代码清单 5-71 所示。

代码清单 5-71 解决循环 Bug 方法二

```

for(var i=0;i<tabMenus.length;i++){
    tabMenus[i]._index=i;
    tabMenus[i].onclick=function(){
        for(var j=0;j<tabContents.length;j++){
            tabContents[j].style.display="none";
        }
        tabContents[this._index].style.display="block";
    }
}

```

---

DOM 节点在 JavaScript 里作为对象存在，默认情况下 DOM 对象有 `innerHTML`、`tagName` 等属性，我们还可以给 DOM 对象添加自定义属性，这里 `_index` 就是我们为

DOM 节点添加的自定义属性。更多关于 DOM 对象的自定义属性详见 5.4.1。

我们可以根据喜好自由选择方法一或者方法二来修改程序。现在无论需要几个标签，我们都只需要修改相应的 HTML 标签，而无需修改 JavaScript 了。如代码清单 5-72 和代码清单 5-73 所示：

代码清单 5-72 两个标签的 Tab

---

```
<!-- 两个标签 -->
<div class="tab">
    <ul class="tab-menuWrapper" id="tab-menuWrapper">
        <li>menu1</li>
        <li>menu2</li>
    </ul>
    <div class="tab-contentWrapper" id="tab-contentWrapper">
        <div>content1</div>
        <div style="display:none">content2</div>
    </div>
</div>
```

---

代码清单 5-73 五个标签的 Tab

---

```
<!-- 五个标签 -->
<div class="tab">
    <ul class="tab-menuWrapper" id="tab-menuWrapper">
        <li>menu1</li>
        <li>menu2</li>
        <li>menu3</li>
        <li>menu4</li>
        <li>menu5</li>
    </ul>
    <div class="tab-contentWrapper" id="tab-contentWrapper">
        <div>content1</div>
        <div style="display:none">content2</div>
        <div style="display:none">content3</div>
        <div style="display:none">content4</div>
        <div style="display:none">content5</div>
    </div>
</div>
```

---

但仍然存在一个很严重的问题，现在我们的 content 还只是非常简单的结构，在真实的环境里，content 里包含的很可能不是简单的文本 content1、content2，content 里很可能有复杂的 HTML 结构，如代码清单 5-74 所示。

代码清单 5-74 结构稍复杂的 Tab

---

```
<div class="tab">
```

---

```

<ul class="tab-menuWrapper" id="tab-menuWrapper">
    <li>menu1</li>
    <li>menu2</li>
    <li>menu3</li>
</ul>
<div class="tab-contentWrapper" id="tab-contentWrapper">
    <div><div>content1</div><ul><li>abc</li></ul></div>
    <div style="display:none"><p>content2</p><div>abc</div></div>
    <div style="display:none">content3</div>
</div>
</div>

```

---

如果 content 里还包含 div 标签，那么 document.getElementById("tab-contentWrapper").getElementsByTagName("div") 返回的就不再是单纯的 content 组成的数组，也包含 content 内部的 div，如代码清单 5-75 所示。

代码清单 5-75 复杂 Tab 结构带来的问题

---

```

<div class="tab">
    <ul class="tab-menuWrapper" id="tab-menuWrapper">
        <li>menu1</li>
        <li>menu2</li>
        <li>menu3</li>
    </ul>
    <div class="tab-contentWrapper" id="tab-contentWrapper">
        <div><div>content1</div><ul><li>abc</li></ul></div>
        <div style="display:none"><p>content2</p><div>abc</div></div>
        <div style="display:none">content3</div>
    </div>
</div>
<script type="text/JavaScript">
    var tabMenus=document.getElementById("tab-menuWrapper").getElementsBy-
    TagName("li"),
    tabContents=document.getElementById("tab-contentWrapper").getElements-
    ByTagName("div");
    alert(tabContents.length);
    //弹出 5, 不是 3
...

```

---

我们需要一个方法，可以不依赖 tagName 来得到 DOM 对象组成数组——是的，我们需要 getElementsByClassName。

### 5.3.2 getElementById、getElementsByTagName 和 getElementsByClassName

原生 JavaScript 为获得 DOM 节点提供了三种方法，getElementById、

`getElementsByName` 和 `getElementsByTagName`。因为 `getElementsByName` 在某些浏览器下解析有问题，加上 `name` 属性一般只对表单项添加，适用性不强，所以很少使用 `getElementsByName`。

`getElementById` 是通过 `id` 作为挂钩来获得 DOM 节点的，它的返回值是单个 DOM 节点。`id` 属性的设计本身就是为了成为 CSS 和 JavaScript 的挂钩，它调用非常方便，所以几乎是最常用的获得 DOM 节点的方法，但因为一个页面内，相同的 `id` 只能出现一次，所以它不适合用来获取一组有“相似功能”的 DOM 节点。

`getElementsByTagName` 是通过标签名作为挂钩来获得 DOM 节点的，它的返回值是由 DOM 节点组成的数组。通常情况下 `getElementsByTagName` 是结合 `document.getElementById` 来使用的，先由 `document.getElementById` 来获得某个 DOM 节点，充当父容器，再用 `getElementsByTagname` 来获得这个父容器里标签名为 ××× 的 DOM 节点。`getElementsByTagName` 可以获得某个 DOM 节点（父容器）下有着相同标签的 DOM 对象，如果我们需要获取一组有“相似功能”的 DOM 节点，可以在设计 HTML 结构时，将它们定义成相同的标签，这在一定程度上弥补了 `getElementById` 无法获取一组有“相似功能”的 DOM 节点的缺陷。但 HTML 可用的标签并不算太多，而且如果需要符合语义，我们可用的标签数量就非常有限。如果想使用相同标签名来标识“相似功能”的 DOM 节点，用 `getElementsByTagName` 来获得它们，那么父容器内部 HTML 结构必须非常稳定，一旦父容器内部 HTML 结构产生变化，很可能我们用 `getElementsByTagName` 获得的 DOM 节点就会跟着改变，产生意料外的错误。用标签名来获得 DOM 节点，让程序和 HTML 结构耦合太紧。

所以我们很需要一种获得 DOM 节点的方法，它能获取一组有“相似功能”的 DOM 节点，而且不像 `tag` 那样可用的标签名有限，也不会让程序和 HTML 结构耦合太紧。很明显这种挂钩不像 `id` 一样，一个页面只能出现一次，也不像 `tag` 那样要承担反映内容语义的责任。`class` 理所当然成了最合适的选择，但遗憾的是原生 JavaScript 并不提供 `getElementsByClassName`，所以我们需要自己定义 `getElementsByClassName`，因为它是如此的重要，所以你可以看到几乎每种 JavaScript 库都会提供它。

在 5.2.2 节中，我们已经定义了自己的 `getElementsByClassName`。需要注意的是，

为了区别 class 是用于 CSS 的挂钩还是 JavaScript 的挂钩（方便后期维护），我们可以给用于 JavaScript 的 class 加上 “J\_” 作为前缀。

接下来让我们继续看 5.3.1 节中未完成的 Tab。Tab 的 menu 部分 HTML 结构非常稳定，所以可以使用 getElementsByTagName，但 content 部分的 HTML 结构就充满未知，我们只能使用 getElementsByClassName，修改后的代码如代码清单 5-76 所示。

代码清单 5-76 使用 getElementsByClassName

```
...
<div class="tab">
    <ul class="tab-menuWrapper" id="tab-menuWrapper">
        <li>menu1</li>
        <li>menu2</li>
        <li>menu3</li>
    </ul>
    <div class="tab-contentWrapper">
        <div class="J_tab-content"><div>content1</div><ul>abc</ul></div>
        <div class="J_tab-content" style="display:none"><p>content2</p>
<div>abc</div></div>
        <div class="J_tab-content" style="display:none">content3</div>
    </div>
<script type="text/JavaScript">
    var GLOBAL={};
    GLOBAL.namespace=function(str){
        var arr=str.split("."),o=GLOBAL;
        for (i=(arr[0]=="GLOBAL") ? 1 : 0; i<arr.length; i++) {
            o[arr[i]]=o[arr[i]] || {};
            o=o[arr[i]];
        }
    }
    GLOBAL.namespace("Dom");
    GLOBAL.Dom.getElementsByClassName=function(str,root,tag){
        if(root){
            root=typeof root=="string" ? document.getElementById(root) :
root;
        } else {
            root=document.body;
        }
        tag=tag || "*";
        var els=root.getElementsByTagName(tag),arr=[];
        for(var i=0,n=els.length;i<n;i++){
            for(var j=0,k=els[i].className.split(" "),l=k.length;j<l;j++){
                if(k[j]==str){
                    arr.push(els[i]);
                    break;
                }
            }
        }
    }
}
```

```

        return arr;
    }

    var tabMenus=document.getElementById("tab-menuWrapper").getElements-
ByTagName("li"),
    tabContents=GLOBAL.Dom.getElementsByClassName("J_tab-content");
for(var i=0;i<tabMenus.length;i++){
    tabMenus[i]._index=i;
    tabMenus[i].onclick=function(){
        for(var j=0;j<tabContents.length;j++){
            tabContents[j].style.display="none";
        }
        tabContents[this._index].style.display="block";
    }
}
</script>

```

---

当我们做完这一切还没来得及高兴时，客户又提新需求了“这个页面里我需要 3 个 Tab，那个页面里我需要 5 个 Tab，谢谢”。

### 5.3.3 可复用性

我们来看看我们的 Tab 能够满足复用的条件吗？似乎有点糟糕，我给 Tab 的 menu 部分使用了 id 作为挂钩——tab-menuWrapper。同一个页面里 id 只能出现一次，所以如果你的程序需要被多处复用，就一定不能使用 id 作为 JavaScript 获得 DOM 节点的挂钩。

我们去掉 id，改用 class 作为挂钩，如代码清单 5-77 所示。

代码清单 5-77 将 id 换为 class

```

...
<div class="tab">
    <ul class="tab-menuWrapper">
        <li class="J_tab-menu">menu1</li>
        <li class="J_tab-menu">menu2</li>
        <li class="J_tab-menu">menu3</li>
    </ul>
...
var tabMenus=GLOBAL.Dom.getElementsByClassName("J_tab-menu");
...

```

---

程序工作得很顺利。接下来，让我们在页面里写 3 个 Tab，如代码清单 5-78 所示。

## 代码清单 5-78 3个 Tab

```

<div class="tab">
  <ul class="tab-menuWrapper">
    <li class="J_tab-menu">menu1-1</li>
    <li class="J_tab-menu">menu1-2</li>
    <li class="J_tab-menu">menu1-3</li>
  </ul>
  <div class="tab-contentWrapper">
    <div class="J_tab-content"><div>content1-1</div><ul>abc</ul></div>
    <div class="J_tab-content" style="display:none"><p>content1-2</p>
    <div>abc</div></div>
    <div class="J_tab-content" style="display:none">content1-3</div>
  </div>
</div>
<div class="tab">
  <ul class="tab-menuWrapper">
    <li class="J_tab-menu">menu2-1</li>
    <li class="J_tab-menu">menu2-2</li>
  </ul>
  <div class="tab-contentWrapper">
    <div class="J_tab-content"><div>content2-1</div><ul>abc</ul></div>
    <div class="J_tab-content" style="display:none"><p>content2-2</p>
    <div>abc</div></div>
  </div>
</div>
<div class="tab">
  <ul class="tab-menuWrapper">
    <li class="J_tab-menu">menu3-1</li>
    <li class="J_tab-menu">menu3-2</li>
    <li class="J_tab-menu">menu3-3</li>
    <li class="J_tab-menu">menu3-4</li>
    <li class="J_tab-menu">menu3-5</li>
  </ul>
  <div class="tab-contentWrapper">
    <div class="J_tab-content"><div>content3-1</div><ul>abc</ul></div>
    <div class="J_tab-content" style="display:none"><p>content3-2</p>
    <div>abc</div></div>
    <div class="J_tab-content" style="display:none">content3-3</div>
    <div class="J_tab-content" style="display:none"><p>content3-4</p>
    <div>abc</div></div>
    <div class="J_tab-content" style="display:none">content3-5</div>
  </div>
</div>

```

效果如图 5-7 所示。

但运行时出了点问题，当我们点击某人标签时，不仅当前的 Tab 中其他的 content 隐藏了，连另外两个 Tab 的 content 也全都隐藏了。例如点击 menu2-2 时，截图如图 5-8 所示。

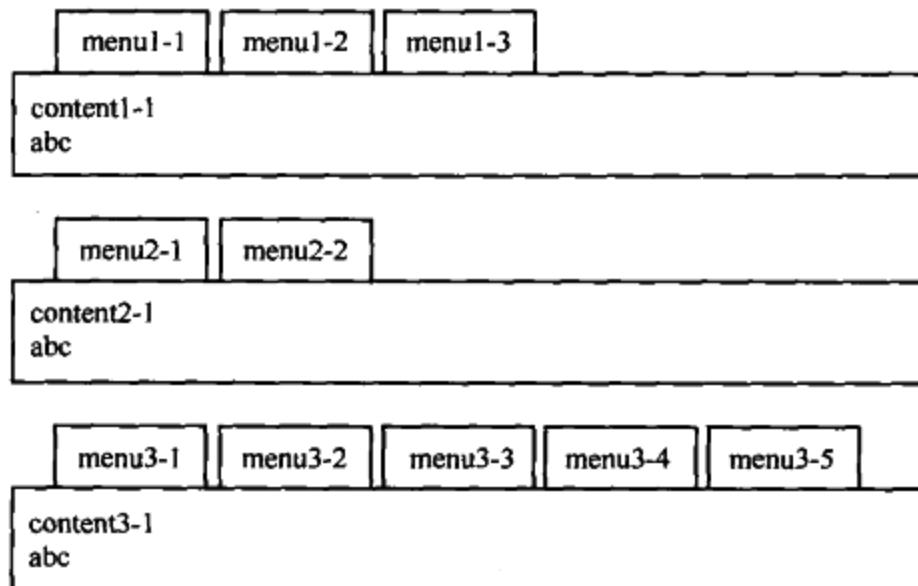


图 5-7 页面里有 3 个 Tab

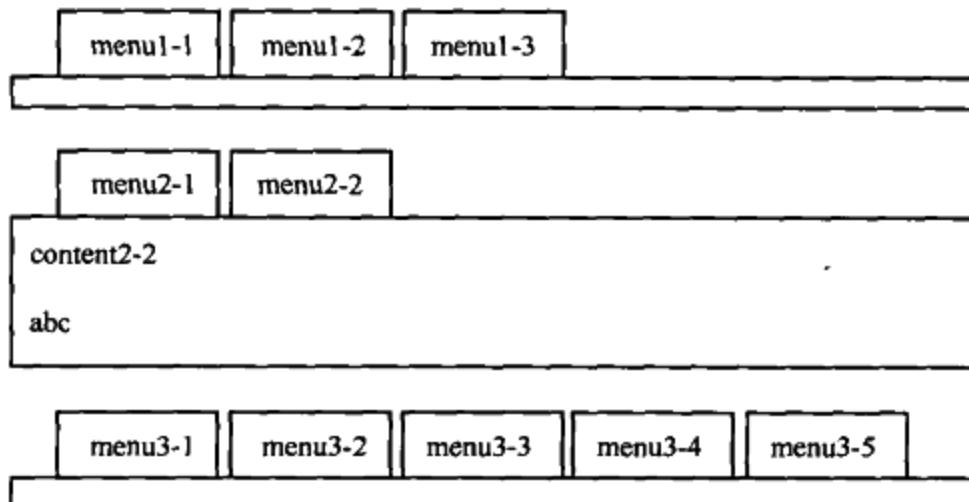


图 5-8 点击 menu2-2

这是因为现在的程序里没有区分 menu 和 content 从属于哪个 Tab，虽然样式看起来是 3 个 Tab，但程序仍然把它们当成 1 个 Tab 在处理。我们需要给 menu 和 content 指定从属于哪个 Tab，也就是在 `getElementsByClassName` 的时候指定父容器，从而将 3 个 Tab 分开。组件需要指定一个根节点，以保持每个组件之间的独立性。如代码清单 5-79 所示。

代码清单 5-79 给 Tab 添加根结点挂钩

---

```
...
<div class="tab J_tab">
  <ul class="tab-menuWrapper">
    ...

```

```

</ul>
<div class="tab-contentWrapper">
    ...
</div>
</div>
<div class="tab J_tab">
    <ul class="tab-menuWrapper">
        ...
    </ul>
    <div class="tab-contentWrapper">
        ...
    </div>
</div>
<div class="tab J_tab">
    <ul class="tab-menuWrapper">
        ...
    </ul>
    <div class="tab-contentWrapper">
        ...
    </div>
</div>
<script type="text/JavaScript">
    ...

function setTab(root) {
    var tabMenus=GLOBAL.Dom.getElementsByClassName("J_tab-menu",root),
        tabContents=GLOBAL.Dom.getElementsByClassName("J_tab-content",root);
    for(var i=0;i<tabMenus.length;i++) {
        tabMenus[i]._index=i;
        tabMenus[i].onclick=function() {
            for(var j=0;j<tabContents.length;j++) {
                tabContents[j].style.display="none";
            }
            tabContents[this._index].style.display="block";
        }
    }
}

var tabs=GLOBAL.Dom.getElementsByClassName("J_tab");
for(var i=0;i<tabs.length;i++) {
    setTab(tabs[i]);
}
</script>

```

### 5.3.4 避免产生副作用

客户又有新需求了，“我想让当前 Tab 能够高亮显示，样式用灰底白字吧”。修改代码如代码清单 5-80 所示。

代码清单 5-80 高亮当前 Tab

```

<style type="text/CSS">
...
.tab .tab-currentMenu{background-color:#333;color:#fff;}
</style>
<div class="tab J_tab">
    <ul class="tab-menuWrapper">
        <li class="J_tab-menu .tab-currentMenu">menu1-1</li>
        <li class="J_tab-menu">menu1-2</li>
        <li class="J_tab-menu">menu1-3</li>
    </ul>
    <div class="tab-contentWrapper">
        ...
    </div>
</div>
<div class="tab J_tab">
    <ul class="tab-menuWrapper">
        <li class="J_tab-menu .tab-currentMenu">menu2-1</li>
        <li class="J_tab-menu">menu2-2</li>
    </ul>
    <div class="tab-contentWrapper">
        ...
    </div>
</div>
<div class="tab J_tab">
    <ul class="tab-menuWrapper">
        <li class="J_tab-menu .tab-currentMenu">menu3-1</li>
        <li class="J_tab-menu">menu3-2</li>
        <li class="J_tab-menu">menu3-3</li>
        <li class="J_tab-menu">menu3-4</li>
        <li class="J_tab-menu">menu3-5</li>
    </ul>
    <div class="tab-contentWrapper">
        ...
    </div>
</div>
<script type="text/JavaScript">
...
function setTab(root){
    var tabMenus=GLOBAL.Dom.getElementsByClassName("J_tab-menu",root),
        tabContents=GLOBAL.Dom.getElementsByClassName("J_tab-content",root);
    for(var i=0;i<tabMenus.length;i++){
        tabMenus[i]._index=i;
        tabMenus[i].onclick=function(){
            for(var j=0;j<tabContents.length;j++){
                tabContents[j].style.display="none";
            }
            tabContents[this._index].style.display="block";
            //如果有当前选中标签，则去掉“tab-currentMenu”
            var currentMenu=GLOBAL.Dom.getElementsByClassName("tab-current-
                Menu",root)[0];
            if(currentMenu){

```

```

        currentMenu.className="";
    }
    //给当前被点击的按钮挂上当前选中的 class
    this.className="tab-currentMenu";
}
}

...
</script>

```

截图如图 5-9 所示。

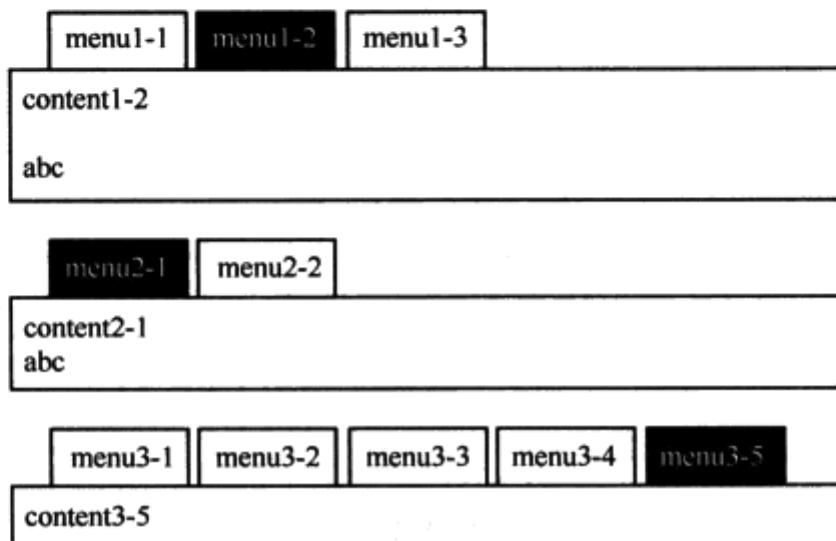


图 5-9 给标签加上“当前”状态

程序虽然能正常工作，但有个地方需要我们注意，如代码清单 5-81 所示。

#### 代码清单 5-81 修改 className

```

if(currentMenu){
    currentMenu.className="";
}
this.className="tab-currentMenu";

```

这段代码的本意是想添加和删除 “tab-currentMenu” 这个用于标识当前状态的 class，如果 DOM 节点没有包含其他用于挂接样式的 class（用于监听事件的不算）就不会出现意外，但如果 DOM 节点挂接了其他用于样式的 class，就会出现冲突了，如代码清单 5-82 所示。

#### 代码清单 5-82 修改 className 引起的冲突

```

<style type="text/CSS">
...

```

```

.tab .tab-currentMenu{background-color:#333;color:#fff;}
.underLine{text-decoration:underline;}
</style>
...
<div class="tab">
    <ul class="tab-menuWrapper">
        <li class="J_tab-menu">menu1-1</li>
        <li class="J_tab-menu">menu1-2</li>
        <li class="J_tab-menu underLine">menu1-3</li>
    </ul>
...

```

初始的时候，menu1-3 带有下划线，如图 5-10 所示。

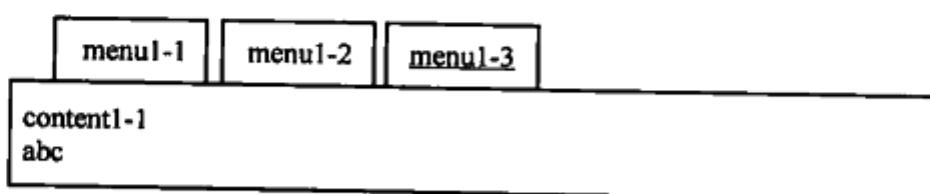


图 5-10 menu1-3 初始时带下划线

当点击 menu1-3 之后，下划线就会消失，如图 5-11 所示。

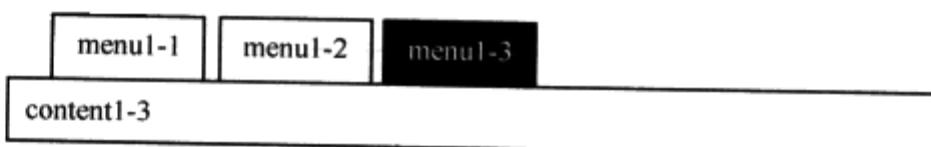


图 5-11 点击 menu1-3 之后，下划线消失

而这并不是我们想要的结果，我们的 Tab 产生了副作用。我们需要的是添加和删除 tab-currentMenu，而不是设置 class 为 tab-currentMenu 和清空所有 class。修改后的代码如代码清单 5-83 所示。

#### 代码清单 5-83 修改至无副作用

---

```

if(currentMenu) {
    currentMenu.className=currentMenu.className.replace(new RegExp("(^|\\s+)tab-currentMenu"), "");
}
if(!new RegExp("(^|\\s+)tab-currentMenu").test(this.className)) {
    this.className =this.className + " tab-currentMenu";
}

```

---

考虑到这种添加和删除 class 的操作在其他地方也有可能会被使用，我们可以在

GLOBAL.Dom 这个命名空间下，新增两个 base 层的接口：addClass(node,str) 和 removeClass(node,str)，如代码清单 5-84 所示。

代码清单 5-84 定义addClass 和removeClass 函数

---

```

...
GLOBAL.Dom.addClass=function(node,str){
    if(!new RegExp("(^|\\s+)" + str).test(node.className)){
        node.className=node.className + " " + str;
    }
}
GLOBAL.Dom.removeClass=function(node,str){
    node.className=node.className.replace(new
RegExp("(^|\\s+)" + str),"");
}
...
if(currentMenu){
    GLOBAL.Dom.removeClass(currentMenu,"tab-currentMenu");
}
GLOBAL.Dom.addClass(this,"tab-currentMenu");
...

```

---

我们在编写程序时，应不断积累经验注意避免产生副作用。

### 5.3.5 通过传参实现定制

客户又提出了需求“这个高亮效果真不错，但我改变主意了：我想让第一个 Tab 的当前标签不加高亮，第二个 Tab 的当前标签需要高亮，第三个 Tab 也需要高亮，同时我希望它能和第二个 Tab 的高亮样式有所区别”。

我们当然可以复制 setTab 函数，编写 setTab2 和 setTab3，如代码清单 5-85 所示。

代码清单 5-85 编写 setTab、setTab2 和 setTab3

---

```

<style type="text/CSS">
...
.tab .tab-currentMenu{background-color:#333;color:#fff;}
.tab .tab-currentMenu2{background-color:blue;color:#fff;}
.underline{text-decoration:underline;}
</style>
<div class="tab J_tab">
    <ul class="tab-menuWrapper">
        <li class="J_tab-menu">menu1-1</li>
        <li class="J_tab-menu">menu1-2</li>
        <li class="J_tab-menu">menu1-3</li>
    </ul>

```

---

```

<div class="tab-contentWrapper">
    ...
</div>
<div class="tab J_tab">
    <ul class="tab-menuWrapper">
        <li class="J_tab-menu .tab-currentMenu">menu2-1</li>
        <li class="J_tab-menu">menu2-2</li>
    </ul>
    <div class="tab-contentWrapper">
        ...
    </div>
</div>
<div class="tab J_tab">
    <ul class="tab-menuWrapper">
        <li class="J_tab-menu .tab-currentMenu2">menu3-1</li>
        <li class="J_tab-menu">menu3-2</li>
        <li class="J_tab-menu">menu3-3</li>
        <li class="J_tab-menu">menu3-4</li>
        <li class="J_tab-menu">menu3-5</li>
    </ul>
    <div class="tab-contentWrapper">
        ...
    </div>
</div>
<script type="text/JavaScript">
...
function setTab(root) {
    ...
    //去掉高亮当前标签的功能
    /*
        var currentMenu=GLOBAL.Dom.getElementsByClassName("tab-currentMenu2",
root)[0];
        if(currentMenu){
            GLOBAL.Dom.removeClass(currentMenu,"tab-currentMenu");
        }
        GLOBAL.Dom.addClass(this,"tab-currentMenu");
    */
    ...
}
function setTab2(root) {
    ...
    //当前标签挂上高亮样式 1
    var currentMenu=GLOBAL.Dom.getElementsByClassName("tab-currentMenu",
root)[0];
    if(currentMenu){
        GLOBAL.Dom.removeClass(currentMenu,"tab-currentMenu");
    }
    GLOBAL.Dom.addClass(this,"tab-currentMenu");
    ...
}
function setTab3(root) {
    ...
    //当前标签挂上高亮样式 2
}

```

```

var currentMenu=GLOBAL.Dom.getElementsByClassName("tab-currentMenu2",
root)[0];
if(currentMenu){
    GLOBAL.Dom.removeClass(currentMenu,"tab-currentMenu2");
}
GLOBAL.Dom.addClass(this,"tab-currentMenu2");
...
}

var tabs=GLOBAL.Dom.getElementsByClassName("J_tab");
setTab(tabs[0]);
setTab2(tabs[1]);
setTab3(tabs[2]);
</script>

```

但这种做法有两个糟糕的地方：1) setTab、setTab2 和 setTab3 的代码大部分相同，可以考虑重用它们；2) 扩展性很糟糕，如果以后还要添加绿底白字或者红底白字的高亮样式，我们就不得不添加 setTab4 和 setTab5。

因为 setTab、setTab2 和 setTab3 之间的区别仅仅是给当前标签挂上不同的 class（或者不挂 class），而 setTab、setTab2 和 setTab3 都将 class 名“写死”在函数内部，所以无法重用。我们可以将这个容易变化的因素从函数内部分离，以参数的形式传进来，函数内部引用这个参数，从而将 class 名“写活”，如代码清单 5-86 所示。

代码清单 5-86 用参数写活变化的部分

```

function setTab(root,currentClass){
    ...
    if(currentClass){
        var currentMenu=GLOBAL.Dom.getElementsByClassName(currentClass,root)[0];
        if(currentMenu){
            GLOBAL.Dom.removeClass(currentMenu,currentClass);
        }
        GLOBAL.Dom.addClass(this,currentClass);
    }
    ...
}

var tabs=GLOBAL.Dom.getElementsByClassName("J_tab");
setTab(tabs[0]);
setTab(tabs[1],"tab-currentMenu");
setTab(tabs[2],"tab-currentMenu2");
</script>

```

如果一个函数内某个因素很不稳定，我们可以将它从函数内部分离出来，以参数的

形式传入，从而将不稳定因素和函数解耦。

除了当前高亮样式可以通过传参定制，另一个经常需要定制的是标签的激活事件。在本例中，激活标签是通过 click 事件，但标签还有另一种常见的激活方式 mouseover。我们最好将标签激活条件也写活，可以通过一个参数轻易实现定制，如代码清单 5-87 所示。

代码清单 5-87 写活事件触发方式

```

...
GLOBAL.namespace("Event");
GLOBAL.Event.on=function(node,eventType,handler){
    node=typeof node=="string" ? document.getElementById(node) : node;
    if(document.all){
        node.attachEvent("on"+eventType,handler);
    } else {
        node.addEventListener(eventType,handler,false);
    }
}

...
function setTab(root,currentClass,trigger){
    var tabMenus=GLOBAL.Dom.getElementsByClassName("J_tab-menu",root),
        tabContents=GLOBAL.Dom.getElementsByClassName("J_tab-content",root);
    //如果不传入激活类型，默认激活类型为点击
    trigger=trigger || "click";
    for(var i=0;i<tabMenus.length;i++){
        tabMenus[i]._index=i;
        //调用 base 层提供的 on 方法
        GLOBAL.Event.on(tabMenus[i],trigger,function(){
            for(var j=0;j<tabContents.length;j++){
                tabContents[j].style.display="none";
            }
            tabContents[this._index].style.display="block";
            if(currentClass){
                var currentMenu=GLOBAL.Dom.getElementsByClassName(currentClass,
                root)[0];
                if(currentMenu){
                    GLOBAL.Dom.removeClass (currentMenu,currentClass);
                }
                GLOBAL.Dom.addClass(this,currentClass);
            }
        });
    }
}
var tabs=GLOBAL.Dom.getElementsByClassName("J_tab");
//将 Tab1 和 Tab3 的标签激活方式改为 mouseover
setTab(tabs[0],null,"mouseover");
setTab(tabs[1],"tab-currentMenu");

```

---

```
setTab(tabs[2],"tab-currentMenu2","mouseover");
</script>
```

---

让我们来试试新代码运行得怎么样？先来试试 Firefox，恩，正常运行。再来看看 IE，奇怪，怎么报错了？

### 5.3.6 控制 this 关键字的指向

来看看是什么错误？“'tabContents[...].style' 为空或不是对象”。我们加个 alert，看看问题出在哪儿，如代码清单 5-88 所示。

代码清单 5-88 调试 Bug

---

```
...
GLOBAL.Event.on(tabMenus[i],trigger,function(){
    for(var j=0;j<tabContents.length;j++) {
        tabContents[j].style.display="none";
    }
    alert(this._index);
    tabContents[this._index].style.display="block";
```

---

在 Firefox 可以正常弹出当前 menu 的索引值，在 IE 下弹出 undefined。这是为什么呢？this 在 IE 和 Firefox 下指向不同的对象吗？

在 JavaScript 里，this 指针的确是个让人捉摸不定的东西。例如 JavaScript 伪协议和内联事件对于 this 的指向不同，如代码清单 5-89 所示。

代码清单 5-89 JavaScript 伪协议和内联事件

---

```
// 弹出 “A”
<a href="#" onclick="alert(this.tagName)">click me</a>
// 弹出 “undefined”
<a href="JavaScript:alert(this.tagName)">click me</a>
// 弹出 “true”
<a href="JavaScript:alert(this==window)">click me</a>
```

---

setTimeout 和 setInterval 也会改变 this 的指向，如代码清单 5-90 所示。

代码清单 5-90 setTimeout 和 setInterval 改变 this 指向

---

```
var name="somebody";
var adang={
    name : "adang",
    say : function(){
```

```

        alert("I'm " + this.name);
    }
};

adang.say();                                // I'm adang
setTimeout(adang.say,1000);                 // I'm someBody
setInterval(adang.say,1000);                 // I'm someBody

```

另外，“DomNode.on×××”也会改变 this 的指向，如代码清单 5-91 所示。

代码清单 5-91 on×××改变 this 指向

```

var name="somebody";
var btn=document.getElementById("btn");
var adang={
    name : "adang",
    say : function(){
        alert("I'm " + this.name);
    }
};
btn.onclick=adang.say;                      // I'm BUTTON

```

使用匿名函数可以解决这个问题，如代码清单 5-92 所示。

代码清单 5-92 匿名函数调整 this 指向

```

var name="somebody";
var btn=document.getElementById("btn");
var adang={
    name : "adang",
    say : function(){
        alert("I'm " + this.name);
    }
};
adang.say();                                // I'm adang
setTimeout(function(){adang.say()},1000);    // I'm adang
setInterval(function(){adang.say()},1000);    // I'm adang
btn.onclick=function(){adang.say()};          // I'm adang
setTimeout(function(){alert(this==window)},1000); // true
btn.onclick=function(){alert(this==btn)};      // true

```

setTimeout、setInterval 和 DomNode.on××× 改变的都是直接调用的函数里 this 的指向，其中 setTimeout 和 setInterval 将直接调用的函数里的 this 指向 window，DomNode.on××× 将直接调用的函数里的 this 指向 DomNode。使用匿名函数将我们的处理函数封装起来，可以将我们的处理函数由直接调用变成通过匿名函数间接调用。

另外，还可以通过 call 和 apply 函数来改变处理函数的 this 指向，如代码清单 5-93 所示。

## 代码清单 5-93 call 和 apply 调整 this 指向

```

var name = "somebody";
var btn = document.getElementById("btn");
var adang = {
    name : "adang",
    say : function(){
        alert("I'm " + this.name);
    }
};
adang.say.call(btn);                                // I'm BUTTON
setTimeout(function(){adang.say.call(btn)},1000);    // I'm BUTTON
setInterval(function(){adang.say.apply(btn)},1000);   // I'm BUTTON
btn.onclick = function(){adang.say.apply(btn)};       // I'm BUTTON

```

在 JavaScript 里使用继承就需要用到 call 或 apply 函数。

在 this 改变指向之前，将它指向的对象保存到一个变量中也是非常常用的方法。如代码清单 5-94 所示。

## 代码清单 5-94 将 this 指向的对象保存到变量

```

<input type="button" value="click me" id="btn" name="BUTTON" />

var name = "somebody";
var adang = {
    name : "adang",
    say : function(){
        alert("I'm " + this.name);
    },
    init : function(){
        //this 指向 adang 对象
        var This = this;
        document.getElementById("btn").onclick=function(){
            // this 指向 btn 的 DOM 节点，This 指向 adang 对象
            This.say();          //I'm adang
            this.say();         //报错, this.say is not a function
        };
    }
};
adang.init();

```

this 关键字会改变指向，只要避开这个关键字就可以得到一个稳定的引用。

回到我们的 Tab 中来，刚才我们查出 this 在 IE 和 Firefox 下指向不同的对象，其中 Firefox 下可以顺利弹出相应 DOM 节点上的属性\_index，可见 Firefox 下指向的是 DOM 节点，那么 IE 下指向的是什么呢？我猜是 window（实在想不出还能指向什么），你说

呢？动手验证我的猜测是否准确，如代码清单 5-95 所示。

代码清单 5-95 调试 Bug

---

```
...
GLOBAL.Event.on(tabMenus[i], trigger, function() {
    for(var j=0;j<tabContents.length;j++) {
        tabContents[j].style.display = "none";
    }
    alert(this == window);
    tabContents[this._index].style.display = "block";
})
```

---

果然，在 IE 下弹出 true，在 Firefox 下弹出 false。追踪代码，不难发现是 GLOBAL.Event.on 在封装 attachEvent 和 addEventListener 时引入的问题。attachEvent 和 addEventListener 除了传参不同，对 this 指针的处理也不同，如代码清单 5-96 所示。

代码清单 5-96 attachEvent、addEventListener 和 this 指针

---

```
<input type="button" value="click me" id="btn" />
<script type="text/JavaScript">
    var btn = document.getElementById("btn");
    if(document.all) {
        //ie
        btn.attachEvent("onclick",function(){
            alert(this.tagName);
        });
        //undefined
        alert(this == window);//true
    };

    } else {
        // Firefox
        btn.addEventListener("click",function(){
            alert(this.tagName);//INPUT
            alert(this == window);//false
        },false);
    }
</script>
```

---

所以我们可以对 GLOBAL.Event.on 方法进一步强化，允许显式地指定它的 this 指针指向，如代码清单 5-97 所示。

代码清单 5-97 加强 on 函数的功能

---

```
GLOBAL.Event.on=function(node, eventType, handler, scope){
    Node=typeof node=="string"?document.getElementById(node) : node;
    scope=scope || node;
    if(document.all){
```

```

        node.attachEvent("on"+eventType,function(){handler.apply(scope,
arguments)} );
    } else {
        node.addEventListener(eventType,function(){handler.apply(scope,
arguments)}),false);
    }
}

```

---

增加 scope 参数，用于设置 handler 中的 this 指针指向，默认指向 node 节点。

修改了 GLOBAL.Event.on 方法之后，Tab 在 IE 和 Firefox 下都可以正常运行了。

### 5.3.7 预留回调接口

对于 Tab 来说，存在一个菜单的激活事件，有时候我们可能需要监听菜单的激活事件，并相应地执行一些操作。这个操作并不在主流程中执行，而是由菜单的激活事件来触发的，它可以由我们自己来定制，并且是可有可无的。对于这种需求，我们是通过设置回调函数来实现的，如代码清单 5-98 所示。

代码清单 5-98 设置回调函数

```

...
function setTab(root,currentClass,trigger,handler){
    var tabMenus = GLOBAL.Dom.getElementsByClassName("J_tab-menu",root),
    tabContents = GLOBAL.Dom.getElementsByClassName("J_tab-content",root);
    trigger = trigger || "click";
    for(var i=0;i<tabMenus.length;i++){
        tabMenus[i]._index = i;
        GLOBAL.Event.on(tabMenus[i],trigger,function(){
            for(var j=0;j<tabContents.length;j++){
                tabContents[j].style.display = "none";
            }
            tabContents[this._index].style.display = "block";
            if(currentClass){
                var currentMenu =
GLOBAL.Dom.getElementsByClassName(currentClass,root)[0];
                if(currentMenu){
                    GLOBAL.Dom.removeClass(currentMenu,currentClass);
                }
                GLOBAL.Dom.addClass(this,currentClass);
            }
            if(handler){
                handler(this._index);
            }
        });
    }
}

```

---

```

var tabs = GLOBAL.Dom.getElementsByClassName("J_tab");
setTab(tabs[0],null,"mouseover");
setTab(tabs[1],"tab-currentMenu");
setTab(tabs[2],"tab-currentMenu2","mouseover",function(index){alert("您
激活的是第"+(index+1)+"个标签")});

```

---

我们给 setTab 方法添加了 handler 参数，用于执行菜单激活事件的回调，handler 是 function 类型，接收被激活菜单的索引作为参数。

添加回调的接口可以增加 Tab 的可扩展性。

### 5.3.8 编程中的 DRY 规则

Tab 还有个很常用的功能叫做自动切换，如果用户不去激活菜单，菜单本身也可以自动切换。我们可以为 Tab 程序加上自动切换的功能，按照前面的思路，将它做成可选的：如果在 setTab 函数的参数中，我们设置 Tab 可以自动切换，那么它就带自动切换功能，如果不设置，那么默认为不带自动切换功能，如代码清单 5-99 所示。

代码清单 5-99 添加自动切换功能

---

```

...
function setTab(root,currentClass,trigger,handler,autoPlay,playTime){
    var tabMenus = GLOBAL.Dom.getElementsByClassName("J_tab-menu",root),
        tabContents = GLOBAL.Dom.getElementsByClassName("J_tab-content",root);
    trigger = trigger || "click";
    playTime = playTime || 3000;
    var currentIndex = 0;
    function autoHandler(){
        currentIndex++;
        if(currentIndex >= tabMenus.length){
            currentIndex = 0;
        }
        for(var i=0;i<tabContents.length;i++){
            tabContents[i].style.display = "none";
        }
        tabContents[currentIndex].style.display = "block";
        if(currentClass){
            var currentMenu =
GLOBAL.Dom.getElementsByClassName(currentClass,root)[0];
            if(currentMenu){
GLOBAL.Dom.removeClass(currentMenu,currentClass);
}
            GLOBAL.Dom.addClass(tabMenus[currentIndex],currentClass);
}
        if(handler){
            handler(currentIndex);
}
    }
}

```

```

}
if(autoPlay){
    setInterval(autoHandler,playTime);
}
for(var i=0;i<tabMenus.length;i++){
    tabMenus[i]._index = i;
    GLOBAL.Event.on(tabMenus[i],trigger,function(){
        for(var j=0;j<tabContents.length;j++){
            tabContents[j].style.display = "none";
        }
        tabContents[this._index].style.display = "block";
        if(currentClass){
            var currentMenu =
GLOBAL.Dom.getElementsByClassName(currentClass,root)[0];
            if(currentMenu){
GLOBAL.Dom.removeClass(currentMenu,currentClass);
}
            GLOBAL.Dom.addClass(this,currentClass);
}
        if(handler){
            handler(this._index);
}
        currentIndex = this._index;
    });
}
}

var tabs = GLOBAL.Dom.getElementsByClassName("J_tab");
setTab(tabs[0],null,"mouseover");
setTab(tabs[1],"tab-currentMenu",null,null,true,5000);
setTab(tabs[2],"tab-currentMenu2","mouseover",function(index){alert("您
激活的是第"+(index+1)+"个标签")});

```

我们给 setTab 函数内部添加了一个 autoHandler 函数用于处理自动切换，通过传参数给 autoPlay（布尔型）和 playTime（数值型，单位为微秒），可以指定 Tab 是否启用自动切换功能，如果启用，切换间隔为多少微秒（默认为 3000 毫秒）。

功能虽然是实现了，但程序存在问题。autoHandler 函数内有很大一部分代码和 GLOBAL.Event.on(tabMenus[i],trigger,function(){}) 中的代码相似。在编程里有一个很出名的规则叫做 DRY——don't repeat yourself，强调在程序中不要将相同的代码重复编写多次，更好的做法是只写一次，然后在多处引用。这么做的好处一方面是可以提高代码的重用率，从而减小代码量，另一方面，如果重复的部分需要修改，只用修改一处，有利于提高代码的可维护性。

我们将两者相似的部分提取出来，修改后的代码如代码清单 5-100 所示。

## 代码清单 5-100 重构代码，将相同代码提取出来

```

...
function setTab(root,currentClass,trigger,handler,autoPlay,playTime) {
    var tabMenus = GLOBAL.Dom.getElementsByClassName("J_tab-menu",root),
        tabContents = GLOBAL.Dom.getElementsByClassName("J_tab-content",root);
    trigger = trigger || "click";
    playTime = playTime || 3000;
    var currentIndex = 0;
    function showItem(n) {
        for(var i=0;i<tabContents.length;i++) {
            tabContents[i].style.display = "none";
        }
        tabContents[n].style.display = "block";
        if(currentClass) {
            var currentMenu =
GLOBAL.Dom.getElementsByClassName(currentClass,root)[0];
            if(currentMenu) {
GLOBAL.Dom.removeClass(currentMenu,currentClass);
            }
            GLOBAL.Dom.addClass(tabMenus[n],currentClass);
        }
        if(handler) {
            handler(n);
        }
    }
    function autoHandler() {
        currentIndex++;
        if(currentIndex >= tabMenus.length) {
            currentIndex = 0;
        }
        showItem(currentIndex);
    }
    if(autoPlay) {
        setInterval(autoHandler,playTime);
    }
    for(var i=0;i<tabMenus.length;i++) {
        tabMenus[i]._index = i;
        GLOBAL.Event.on(tabMenus[i],trigger,function(){
            showItem(this._index);
            currentIndex = this._index;
        });
    }
}

var tabs = GLOBAL.Dom.getElementsByClassName("J_tab");
setTab(tabs[0],null,"mouseover");
setTab(tabs[1],"tab-currentMenu",null,null,true,5000);
setTab(tabs[2],"tab-currentMenu2","mouseover",function(index){alert(" 您
激活的是第"+(index+1)+"个标签" )});
```

### 5.3.9 用 hash 对象传参

现在程序给人的感觉就已经很不错了，只有 `setTab(tabs[1],"tabcurrentMenu",null, null,true,5000)` 的那两个 `null` 有点碍眼。因为我们不需要定制 `trigger` 和 `handler`，但是又需要启动自动切换功能，所以我们不得不给 `trigger` 和 `handler` 传参 `null`，如果以后我们还会再扩展 `setTab`，是否需要用：`setTab(tabs[1],"tabcurrentMenu",null,null,null,null,null,null,null,null,null,true,5000)` 调用它？不行，我们要想想办法。

解决这个问题最常见的方法是使用 `hash` 对象来传参。我们来比较一下用普通方式传参和用 `hash` 对象传参的区别，如代码清单 5-101 所示。

代码清单 5-101 用普通方式传参和用 `hash` 对象传参

---

```
//用普通方式传参
function test(a,b,c){
    var oA = a || 1, oB = b || 2, oC = c || 3;
}
test(4,5,6);
test(null,7,8);
test(null,null,9);

// 用 hash 对象传参
function test2(config){
    var oA = config.a || 1, oB = config.b || 2, oC = config.c || 3;
}
test2({a:4,b:5,c:6});
test2({b:7,c:8});
test2({c:9});
```

---

普通传参方式，参数如同数组，位置和顺序都非常重要，而用 `hash` 对象传参，参数的位置和顺序就不重要了。类似于 `test(null,null,null,nul,null,null,null, "hello")` 这样的调用就可以改用 `test({str: "hello"})`。使用 `hash` 对象传参，可以提高函数调用的灵活性，也可以提高函数的扩展性。

`setTab` 函数现在接收 6 个参数，除了 `root` 是必选的，其他 5 个全是可选的。我们将它修改一下，改用 `hash` 对象传参，如代码清单 5-102 所示。

代码清单 5-102 改用 `hash` 对象传参

---

```
...
function setTab(config){
var root = config.root;
```

```

var currentClass = config.currentClass;
var trigger = config.trigger || "click";
var handler = config.handler;
var autoPlay = config.autoPlay;
var playTime = config.playTime || 3000;
var tabMenus = GLOBAL.Dom.getElementsByClassName("J_tab-menu", root),
tabContents = GLOBAL.Dom.getElementsByClassName("J_tab-content", root);
var currentIndex = 0;
function showItem(n){
    for(var i=0;i<tabContents.length;i++){
        tabContents[i].style.display = "none";
    }
    tabContents[n].style.display = "block";
    if(currentClass){
        var currentMenu =
GLOBAL.Dom.getElementsByClassName(currentClass, root)[0];
        if(currentMenu){
            GLOBAL.Dom.removeClass(currentMenu, currentClass);
        }
        GLOBAL.Dom.addClass(tabMenus[n], currentClass);
    }
    if(handler){
        handler(n);
    }
}
function autoHandler(){
    currentIndex++;
    if(currentIndex >= tabMenus.length){
        currentIndex = 0;
    }
    showItem(currentIndex);
}
if(autoPlay){
    setInterval(autoHandler, playTime);
}
for(var i=0;i<tabMenus.length;i++){
    tabMenus[i]._index = i;
    GLOBAL.Event.on(tabMenus[i], trigger, function(){
        showItem(this._index);
        currentIndex = this._index;
    });
}
}

var tabs = GLOBAL.Dom.getElementsByClassName("J_tab");
setTab({root:tabs[0], trigger:"mouseover"});
setTab({root:tabs[1], currentClass:"tab-
currentMenu", autoPlay:true, playTime:5000});
setTab({root:tabs[2], currentClass:"tab-
currentMenu2", trigger:"mouseover", handler:function(index){alert("您激活的是第
"+(index+1)+"个标签")}}); .

```

程序到了现在，似乎已经没有大的毛病了，它是不是已经足够好了呢？在下一节我

们试着用面向对象的方式来写 Tab，并比较一下它和现在这种方式的区别。

## 5.4 面向对象编程

### 5.4.1 面向过程编程和面向对象编程

电话本是个生活中很常见的东西，它可以用来记录一些人名和相应的电话号码。我们为电话本写个程序，帮助我们查询电话本中某个人的电话号码，可以为电话本添加新的记录和删除某个已有的记录。

实现起来最简单的方式如代码清单 5-103 所示。

代码清单 5-103 电话本程序

```
//定义电话本 1
var phonebook = [
    {name:"adang",tel:"111111"},  

    {name:"zhangxia",tel:"222222"},  

    {name:"yangfuchuan",tel:"333333"},  

    {name:"qiaojiwu",tel:"444444"},  

    {name:"zhouyubo",tel:"555555"}  

];  
  
//定义电话本 2
var phonebook2 = [
    {name:"niaoren",tel:"111111"},  

    {name:"j2ee",tel:"222222"},  

    {name:"baobao",tel:"333333"}  

];  
  
//查询电话
function getTel(oPhonebook,oName){
    var tel = "";
    for(var i=0;i<oPhonebook.length;i++){
        if(oPhonebook[i].name == oName){
            tel = oPhonebook[i].tel;
            break;
        }
    }
    return tel;
}  
  
//添加记录
function addItem(oPhonebook,oName,oTel){
    oPhonebook.push({name:oName,tel:oTel});
}
```

```

//删除记录
function removeItem(oPhonebook,oName) {
    var n;
    for(var i=0;i<oPhonebook.length;i++) {
        if(oPhonebook[i].name == oName) {
            n = i;
            break;
        }
    }
    if(n != undefined) {
        oPhonebook.splice(n,1);
    }
}

//从电话本1中查询"niaoren"的电话
var str = getTel(phonebook,"niaoren");
alert(str); // ""

//在电话本1中添加"niaoren"的记录
addItem(phonebook,"niaoren","666666");
str = getTel(phonebook,"niaoren");
alert(str); // "666666"

//在电话本1中删除"niaoren"的记录
removeItem(phonebook,"niaoren");
str = getTel(phonebook,"niaoren");
alert(str); // ""

// 在电话本2中查询"niaoren"的记录
str = getTel(phonebook2,"niaoren");
alert(str); // "111111"

```

首先，我们定义电话本，它是由 hash 对象组成的数组，一个 hash 对象中记录着 name 和 tel，对应一条记录。然后我们定义了 3 个函数 getTel、addItem 和 removeItem 分别用于查询某人的电话、添加一条记录和删除一条记录。在需要执行查询、添加或删除操作时，我们将电话本作为参数调用对应的方法。

这种编程方式将程序分成了“数据”和“处理函数”两部分，程序以“处理函数”为核心，如果要执行什么操作，就将“数据”传给相应的“处理函数”，返回我们需要的结果。这种编程方式就是面向过程编程。

面向过程的思路很好掌握，上手容易。但它存在三方面的问题：

- 1) 数据和处理函数没有直接的关联，在执行操作的时候，我们不但要选择相应的处理函数，还要自己准备处理函数需要的数据，也就是说，在执行操作时，我们需要同

时关注处理函数和数据。

2) 数据和处理函数都暴露在同一作用域内，没有私有和公有的概念，整个程序中所有的数据和处理函数都可以互相访问，在开发阶段初期也许开发速度会很快，但到了开发后期和维护阶段，由于整个程序耦合得非常紧，任何一个处理函数和数据都有可能关联到其他地方，容易牵一发而动全身，从而加大了修改难度。

3) 面向过程的思维方式是典型的计算机思维方式——输入数据给处理器，处理器内部执行运算，处理器返回结果。而实际生活中，我们的思路却不是这样——实际生活中所有的东西都是有状态有动作的物件，例如笔者就是实际生活中的一个客观物件，我有“姓名”，我叫“阿当”，我有“状态”，我现在是“醒着”的，我有动作，我可以说话，向人介绍我的“姓名”，我可以去睡觉，如果睡着了我的“状态”就由“醒着”变成“睡着”。如果要用面向过程思维来描述我，是很难做到的，因为面向过程的思维方式是在描述一个个“动作”，有动作的起点（初始数据），有动作的过程（初始数据传给处理函数进行处理），有动作的终点（处理函数返回处理结果），而客观世界中存在的却是一个个“物件”，物件有状态，有动作，物件本身只是一个客观存在，它没有起点，没有终点。能用面向过程思维描述的只是物件的动作，例如我开始睡觉（起点），意识逐渐模糊（过程），睡着了（终点）。用面向过程的思维方式编程，是无法描绘客观世界的事物的，我们编程的时候无法直接使用生活中的思维方式。

没错，接下来我们要说说面向对象编程了。

什么是面向对象编程呢？面向对象编程就是抛开计算机思维，使用生活中的思维进行编程的编程方式。面向过程的思维就是描述一个个“动作”，而面向对象的思维就是描述一个个“物件”，客观生活中的物件，都可以通过面向对象思维映射到程序中——如果你使用的编程语言支持面向对象，在程序中我们管“物件”叫做“对象”，对象由两部分组成：“属性”和“行为”，对应客观世界中物件的“状态”和“动作”。用面向过程的方式来描述笔者，代码如代码清单5-104所示。

#### 代码清单5-104 面对过程方式编程

---

```
var name = "adang";
var state = "awake";
var say = function(oName) {
```

```

        alert("I'm " + oName);
    }
var sleep = function(oState){
    oState = "asleep"
}
say(name);
sleep(state);

```

程序由变量和函数组成，而变量和函数无法联系起来共同描述同一个物件，无法很语义化地表达“阿当说”、“阿当睡觉”。改用面向对象的编程方式，代码如代码清单 5-105 所示。

代码清单 5-105 面向对象方式编程

```

var adang = {
    name : "adang",
    state : "awake",
    say : function(){
        alert("I'm " + this.name);
    },
    sleep : function(){
        this.state = "asleep";
    }
};
adang.say();
adang.sleep();

```

首先在程序中映射出笔者这个物件，笔者对应程序中的 hash 对象 adang，name 为 adang，state 为 awake，有 say 和 sleep 两个行为（此时只是定义了这两个行为，并未调用）。然后调用 adang.say() 和 adang.sleep() 轻易表述出“阿当说”和“阿当睡觉”。

属性本质其实是个变量，也就是面向过程中的数据，而行为的本质其实是函数，也就是面向过程中的处理函数。不同的是，面向过程中，数据和处理函数并没有关联起来，共同属于某个物件。而面向对象将数据和处理函数定义到了一个对象的内部，作为这个对象的属性和行为存在。在对象外部，属性和行为可以用对象的属性和对象的行为来调用，从而让程序有了按真实世界的思维方式进行描述的能力。在对象内部，对象的属性和行为通过 this 关键字（或者其他类似的关键字，例如有的语言用 SELF 作为这个关键字）关联起来，例如 say 这个行为在调用 name 属性的时候，就可以使用 this.name 来调用自己对象的属性。

面向过程编程，数据和处理函数的组织方式如图 5-12 所示。

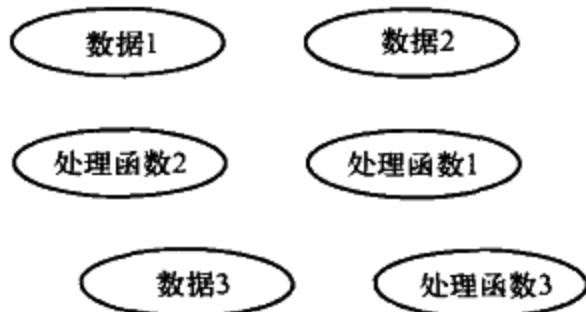


图 5-12 面向过程编程中数据和处理函数的组织方式

面向对象编程中数据（属性）和处理函数（行为）的组织方式如图 5-13 所示。

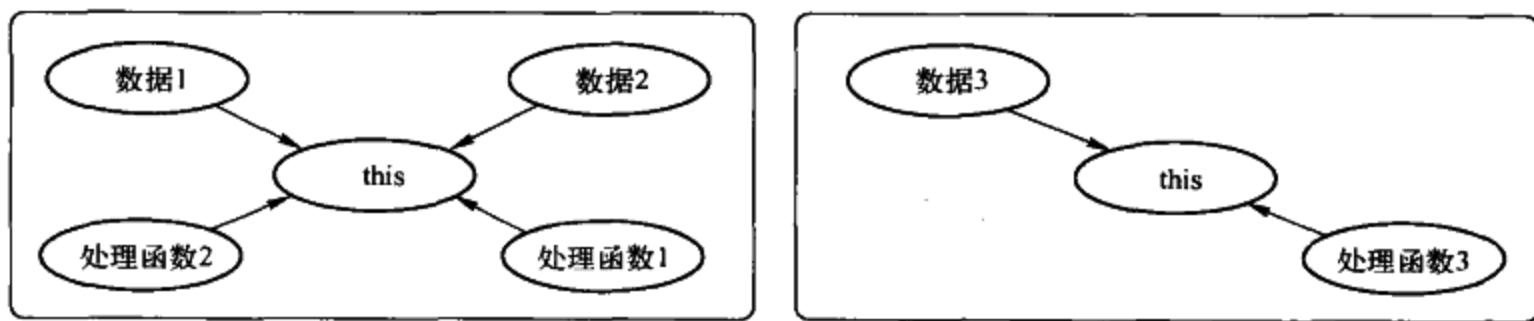


图 5-13 面向对象编程中数据（属性）和处理函数（行为）的组织方式

面向过程编程所有的数据和处理函数都是公有的，整个编程的思维过程就是定义数据，定义处理函数，然后将数据传给处理函数进行处理，处理函数之间也可以互相调用，数据和处理函数紧密耦合。面向对象编程的思维过程是定义一个个对象，对象有自己的属性和行为，因为属性和行为都是从属于对象的，于是有了“对象内”和“对象外”的概念，整个程序可以由一堆对象组成，对象与对象之间可能会有通信，为了实现这种通信，对象会将自己的部分属性和行为设计成公有，暴露出来成为通信的接口。对象和对象之间的通信都是建立在接口的基础上的。当然我们可以将对象所有的属性和行为都设为公有的，全部都作为接口，但接口越多，会让对象之间耦合越紧密，增加维护难度，所以一般情况下，我们都会尽量将对象的属性和方法设为私有，只将必要的属性和行为设为公有。但对象的公有属性和公有行为越少，整个程序的扩展性会越差，所以我们在设计公有和私有的时候需要权衡一下，在不影响扩展性的前提下，尽量将属性和行为设为私有。

面向过程编程程序的耦合关系如图 5-14 所示。

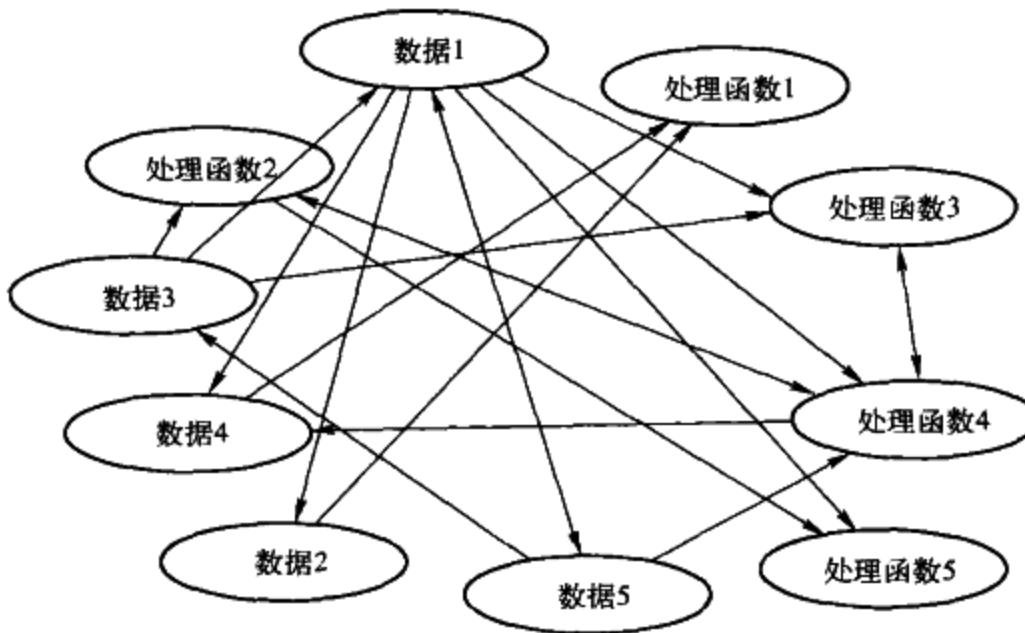


图 5-14 面向过程编程程序的耦合关系

面向对象编程程序的耦合关系如图 5-15 所示。

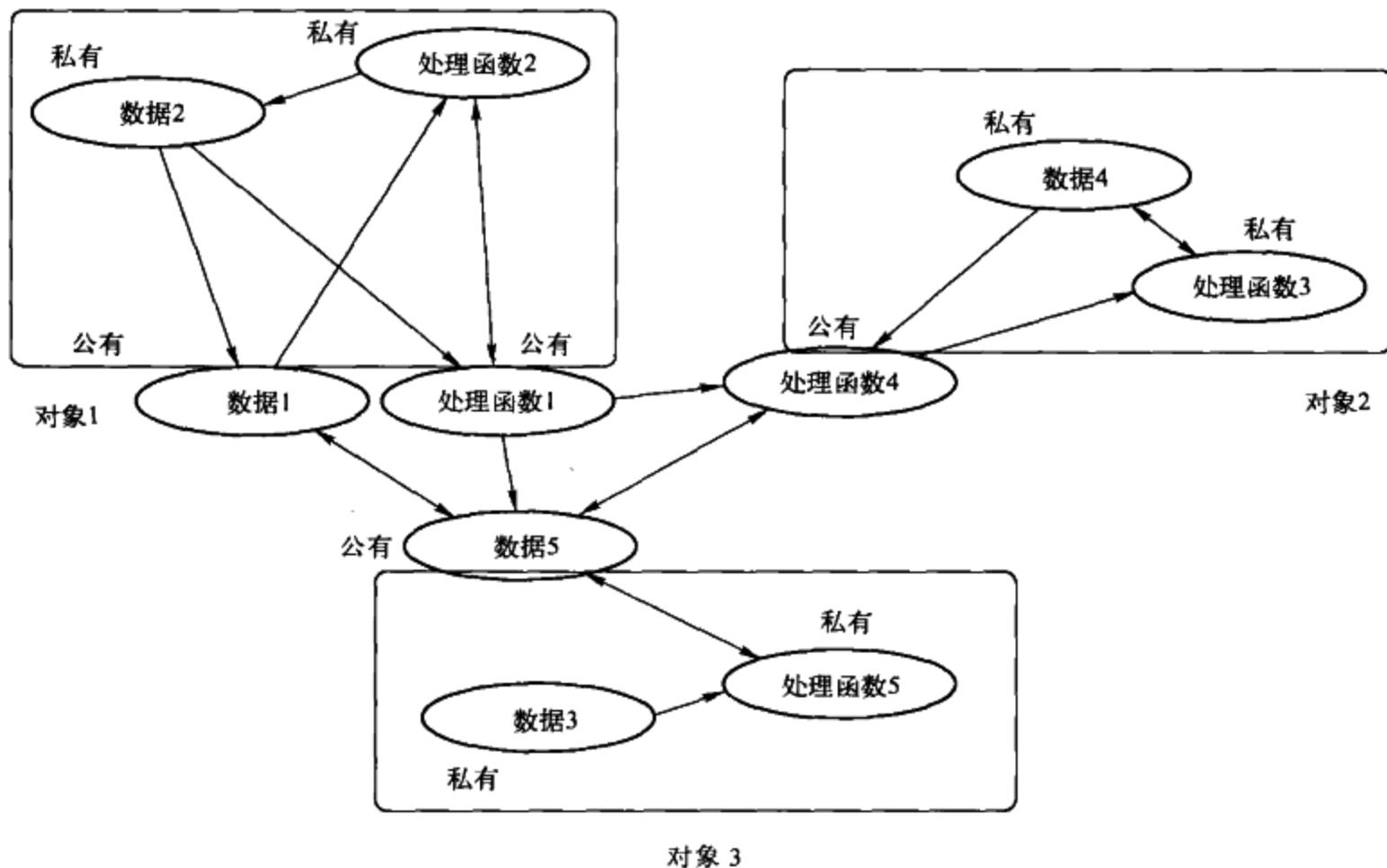


图 5-15 面向对象编程程序的耦合关系

经常听到这么一个比喻：面向过程编程程序的耦合关系就像是一个老旧的半导体收

音机，整个收音机的线路错综复杂地交织在一起，形成一个不可分割的整体，如果哪一部分坏掉了，整个收音机就报废了；面向对象编程程序的耦合关系就像是一台先进的计算机，计算机由一系列的零件组装而成，有硬盘、内存、屏幕、声卡等，每个零件都是互相独立的，通过接口连接起来形成一个整体，如果哪一部分坏了，只需更换相应的零件，其他零件可以不受影响。

在实际工作中，很多工程师过于关注面向对象的语法了，甚至误以为这就是面向对象的全部。其实不然，面向对象英文全称叫做 Object Oriented，简称 OO。OO 其实包括 OOA（Object Oriented Analysis，面向对象分析）、OOD（Object Oriented Design，面向对象设计）和 OOP（Object Oriented Programming，面向对象的程序设计）。面向对象的语法只对应 OOP，只是 OO 的一部分。

一个典型的 OO 编程过程应该是先整理需求，根据需求进行 OOA，将真实世界的客观物件抽象成程序中的类或对象，这个过程经常会用到的是 UML 语言，也称 UML 建模，OOA 的输出结果是一个个类或对象的模型图。接下来要进行 OOD，这一步的目的是处理类之间的耦合关系，设计类或对象的接口，此时会用到各种设计模式，例如观察者模式、责任链模式等。OOA 和 OOD 是个反复迭代的过程，它们本身也没有非常清晰的边界，是相互影响、制约的。等 OOA 和 OOD 结束之后，才到 OOP，进行实际的编码工作。OOA 和 OOD 是面向对象编程的思想和具体语言无关，而 OOP 是面向对象编程的工具，和选用的语言相关。OOP 是 OOA 和 OOD 的底层，不同语言的语法不同，所以 OOP 不同，但 OOA 和 OOD 与具体要求语言无关，一般情况下可以轻易跨语言重用。比如，我们经常会看到设计模式方面的书会声明是基于 Java 或者基于 C# 描述的，设计模式和具体语言无关，无论哪种支持面向对象的语言都可以使用设计模式。

OOP 是使用面向对象技术的基础，面向对象的思维最后是要通过 OOP 来实施的。但 OOP 并不等于 OO，甚至并不是 OO 最重要的部分。笔者认为 OOA 才是 OO 最重要、最关键的部分，绝大多数情况下，我们的程序可能用不上很复杂的 OOD 设计，OOA 会占据大部分的开发时间，OOP 反而只占很少量的时间。OOA 的熟练度直接关系到开发的速度，而 OOA 能力的提升很需要经验的积累。

我们经常说优秀的程序是“高内聚、低耦合”的。聚合指的是把一个复杂的事物看

成若干比较简单的事物的组装体，从而简化对复杂事物的描述，“高内聚”就是指对象（或类）对外提供的接口非常简单易懂，复杂的底层操作都封装在对象（或类）的接口内部，对用户透明。用户不用关心过多的底层细节，只需知道类提供了哪些接口，用户知道的底层细节越少，对象的聚合程度就越高。耦合指的是类与类之间关联和依赖的程度，低耦合就是指类与类之间依赖的程度低，类与类通信需要关联的接口越少，耦合程度越低。很多工程师误以为高内聚、低耦合是 OOP 的工作，其实不然，决定聚合和耦合程度的是 OOA 和 OOD，OOA 和 OOD 是工作在架构层面的，而 OOP 是工作在编码层面的。从大局上决定程序品质的，不是 OOP，而是 OOA 和 OOD，这是很多工程师需要注意的。

#### 5.4.2 JavaScript 的面向对象编程

前面我们说过，OOA 和 OOD 与具体语言无关，而 OOP 则直接跟语言相关，例如 Java、C#、C++、PHP、ActionScript 3、Ruby 都是支持面向对象的语言，但它们的语法却并不相同，例如 Java 中实例化一个类是通过 new SomeClass()，而 Ruby 中却是通过 SomeClass.new() 来实现；C# 的构造函数可以设为私有的，而 ActionScript 3 的构造函数却只能是公有的；C# 只支持单继承，而 C++ 支持多继承。

与 Java、C# 这种正统面向对象语言相比，JavaScript 的面向对象语法显得十分另类。很多人说 JavaScript 与其说是面向对象语言，不如说是基于对象语言（也有人说它是基于原型语言）。下面我们一起看看 JavaScript 的面向对象语法都有哪些奇怪的地方。

##### 1. JavaScript 的类定义

一般的面向对象语言都是通过 Class 来定义类，然后在 Class 内部定义构造函数和其他的属性、行为。例如，用 Java 定义一个类，如代码清单 5-106 所示。

代码清单 5-106 Java 的类

---

```
//定义 Animal 类
public class Animal{
    String name , type = "animal";
    public Animal(String a) {
```

```

        name = a;
    }
    public void say(){
        System.out.println("I'm a(an) " + type + " , my name is "
+ name);
    }
}
...
// 实例化 Animal 类
Animal myDog = new Animal("wangcai");
myDog.say();

```

JavaScript 很奇怪，它没有 Class 关键字，在 JavaScript 中是用函数来充当类的。函数在 JavaScript 中既可以当作普通函数使用，也可以当作类来使用，在充当类的时候，它本身又担负着构造函数的责任。下面我们通过一个实例来了解一下，如代码清单 5-107 所示。

代码清单 5-107 JavaScript 中的类

```

// 函数作为普通函数
function sayHi(){
    alert("hi");
}
// 调用函数
sayHi();

// 函数作为类
function Animal(name){
    this.name=name;
    this.type="animal";
    this.say=function(){
        alert("I'm a(an) "+this.type+", my name is "+this.name);
    }
}
// 实例化 Animal 类
var myDog = new Animal("wangcai");
myDog.say();

```

函数作为普通函数使用时，通常直接使用“()”进行调用，而作为类使用时，通常使用 new 来实例化。当然，两者其实并没有语法上的明显分隔，在调用 sayHi 时，你也可以使用 new sayHi () 来进行调用。只是通常情况下，作为函数时我们更倾向于用动词来命名，而作为类时用名词来命名。按照习惯，类名的首字母大写。

通过上面的例子，我们可以看到 JavaScript 中关于类的用法，在实例化类时 JavaScript 和 Java 这种正统面向对象语言并没有明显的差别，差别主要在类的定义方式上。

另外，需要说明的是，JavaScript 是基于原型的语言，通过 new 实例化出来的对象，其属性和行为来自于两部分，一部分来自于构造函数，另一部分来自于原型。什么是原型呢？当我们声明一个类时，其实同时生成了一个对应的原型，例如我们定义 Animal 这个类时，会生成一个与 Animal 类对应的原型，通过 Animal.prototype 可以指向这个原型，原型可以通过 constructor 指向 Animal 类，更确切地说，是指向 Animal 类的构造函数，如代码清单 5-108 所示。

代码清单 5-108 构造函数和原型之间的引用

---

```
// 定义 Animal 类的构造函数
function Animal() {
    ...
}
var a = Animal.prototype;           // a 指向 Animal 类对应的原型
var b = a.constructor;             // b 指向 a 对应的类的构造函数
alert(b == Animal)                // true
```

---

当 new Animal () 的时候，返回的 Animal 实例会结合构造函数中定义的属性、行为和原型中定义的属性、行为，生成最终属于 Animal 实例的属性和行为。构造函数中定义的属性和行为的优先级比原型中定义的属性和行为优先级高，如果构造函数和原型定义了同名的属性或行为，构造函数中的属性或行为会覆盖原型中的同名的属性或行为，如图 5-16 所示。

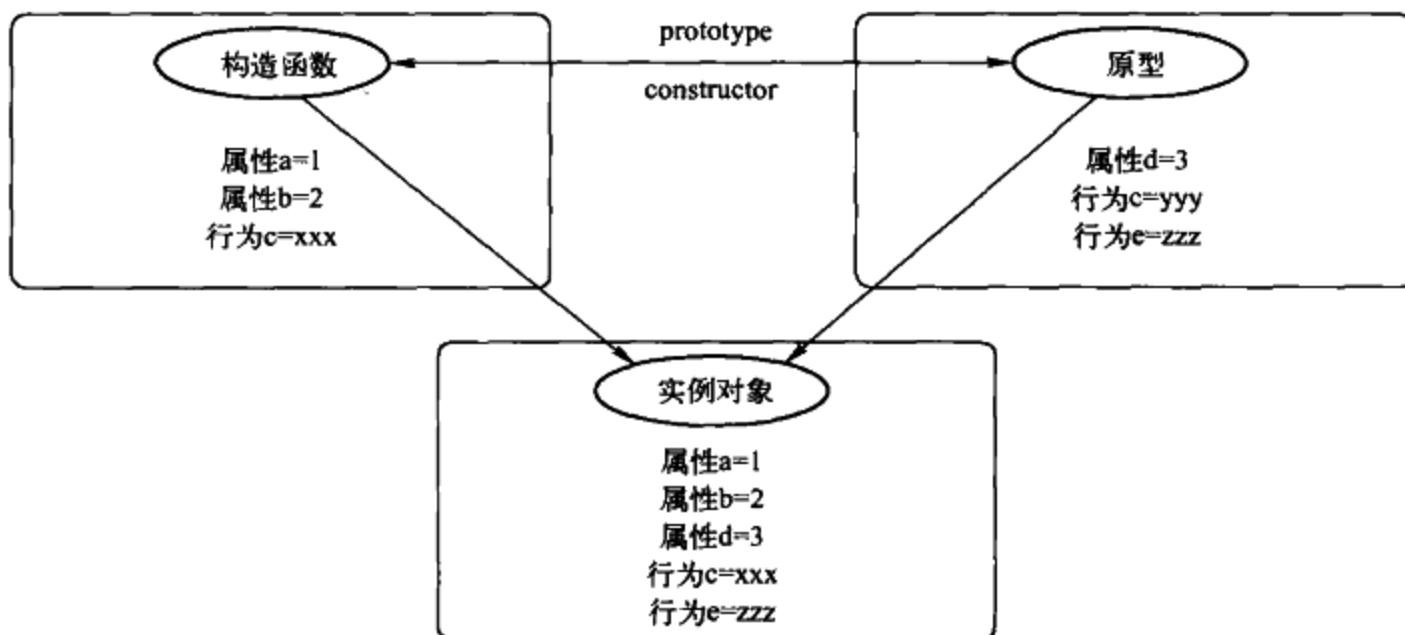


图 5-16 类、原型和实例的关系

代码清单 5-107 定义的 Animal 类没有使用原型，我们将它改用原型的方式重写，如代码清单 5-109 所示。

代码清单 5-109 使用原型

---

```
//定义 Animal 类
function Animal() {
}

//修改 Animal 类的原型
Animal.prototype = {
    name : "xxx",
    type : "animal",
    say : function(){
        alert("I'm a(an) "+this.type+", my name is "+this.name);
    }
}
//实例化 Animal 类
var myDog=new Animal();
myDog.say();           //I'm a(an) animal , my name is xxx
```

---

原型是个 hash 对象，所以我们也可以把它写成代码清单 5-110 所示的形式。

代码清单 5-110 使用原型的另一种方法

---

```
//修改 Animal 类的原型
Animal.prototype.name="xxx";
Animal.prototype.type="ANIMAL";
Animal.prototype.say=function(){
    alert("I'm a(an) "+this.type+", my name is"+this.name);
};
```

---

一般来说，我们习惯把属性放在构造函数里，而不是放在原型里，方便从构造函数里接收参数，如代码清单 5-111 所示。

代码清单 5-111 JavaScript 类的典型用法

---

```
//定义 Animal 类
function Animal(name){
    this.name=name || "xxx";
    this.type="animal";
}
// 修改 Animal 类的原型
Animal.prototype = {
    say : function(){
        alert("I'm a(an) "+this.type+", my name is"+this.name);
    }
}
```

---

```
// 实例化 Animal 类
var myDog = new Animal("wangcai");
myDog.say();           // I'm a(an) animal , my name is wangcai
```

---

myDog 的 name 和 type 属性来自构造函数，而 say 行为来自原型。需要注意的是，this 关键字无论出现在构造函数中，还是出现在原型中，指向的都是实例对象，通过 this 关键字，可以让属性和方法在构造函数和原型间通信。

## 2. 公有和私有

正统的面向对象语言会提供 public、protect、private 等关键字来声明属性和行为的可访问性是公有还是私有。但 JavaScript 并不提供这些关键字，在 JavaScript 中公有还是私有是通过作用域实现的，如代码清单 5-112 所示。

代码清单 5-112 JavaScript 中的公有和私有

---

```
// 定义 Animal 类
function Animal(name) {
    // 公有属性
    this.name=name || "xxx";
    this.type="animal";
    // 私有属性
    var age=20;
    // 私有方法
    var move=function(){
        alert("I'm moving now");
    }
}
// 修改 Animal 类的原型
Animal.prototype={
    // 公有方法
    say : function(){
        alert("I'm a(an) "+this.type+", my name is"+this.name);
    }
}
// 实例化 Animal 类
var myDog=new Animal("wangcai");
alert(myDog.name);      // wangcai
alert(myDog.age);       // undefined
myDog.move();            // 报错, myDog.move is not function
```

---

用 this××× 定义的属性是公有的，而用 var ××× 定义的属性是私有的，通过在 Animal 构造函数中用 var 定义变量 age，我们给 Animal 类添加了私有属性 age。因为私有属性没有和 this 挂钩，所以定义在构造函数中的私有属性，没办法被原型中的方法访

问，私有属性的作用域只在类的构造函数中，如代码清单5-113所示。

代码清单5-113 私有属性的作用域

---

```
//定义Animal类
function Animal(name){
    //公有属性
    this.name=name||"xxx";
    this.type="animal";
    //私有属性
    var age=20;
    //私有方法
    var move=function(){
        alert("I'm moving now");
    }
}
//修改Animal类的原型
Animal.prototype={
    //公有方法
    say : function(){
        alert("I'm a(an) "+this.type+", my name is"+this.name+", I'm "+
            age);
    },
    act:function(){
        move();
    }
}
//实例化Animal类
var myDog=new Animal("wangcai");
myDog.say();           //报错，age未定义
myDog.act();           //报错，move is not defined
```

---

让公有行为能够访问私有属性和私有行为，最简单的解决方法是将公有行为也写在类的构造函数里，这样属性和行为无论是公有还是私有都共同作用在构造函数的作用域里，如代码清单5-114所示。

代码清单5-114 行为访问私有属性的方法

---

```
//定义Animal类
function Animal(name){
    //公有属性
    this.name=name||"xxx";
    this.type="animal";
    //私有属性
    var age=20;
    //私有方法
    var move=function(){
        alert("I'm moving now");
    }
    this.say=function(){
```

---

```

        alert("I'm a(an) "+this.type+", my name is "+this.name+",  

              I'm "+age);  

    }  

    this.act=function(){  

        move();  

    }  

}  

//修改 Animal 类的原型  

Animal.prototype={  

}  

//实例化 Animal 类  

var myDog=new Animal("wangcai");  

myDog.say();           //I'm a(an) animal , my name is wangcai,I'm 20  

myDog.act();          //I'm moving now

```

将所有属性和行为，无论公有还是私有全部写在构造函数里，的确是最方便的方式，但并不推荐这么做。因为在内存中一个类的原型只有一个，写在原型中的行为，可以被所有实例所共享，实例化的时候，并不会在实例的内存中再复制一份，而写在类里的行为，实例化的时候会在每个实例里复制一份，如图 5-17 所示。

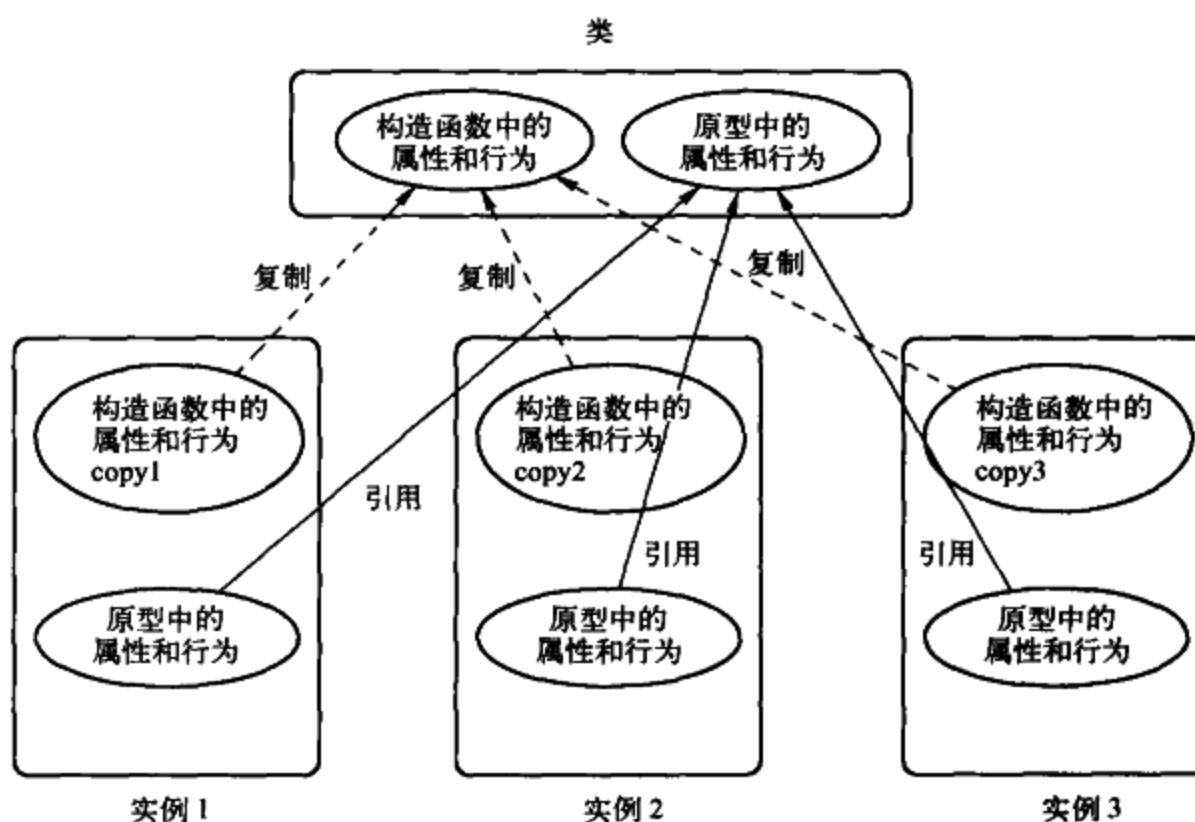


图 5-17 原型中的行为，类中的行为和实例中的行为在内存中的关系

把行为写在原型里可以减少内存消耗，没有特殊原因，推荐尽量把行为写在原型里。写在原型中的行为一定是公有的，而且无法访问私有属性，所以如何处理私有行为和私有属性是个难题。一般来说，如果对属性和行为的私有性有非常高的强制性，比如

说多人合作，为了确保维护不会出现问题，在开发之初明确好各个类的接口，除了必要的接口设为公有，其他所有接口一律设为私有，以此来降低类之间的耦合程度，确保可维护性，这时我们不得不牺牲内存，将私有行为放在构造函数里，实现真正的私有；如果对属性和行为的私有性要求不高，更常见的做法是约定私有行为，我们还是用 `this._XXX` 定义私有属性，在原型中定义私有行为，但通过给属性和行为的名称前面加上“`_`”来约定它是私有的，这是一种命名约定，它并不能真正实现行为的私有，但它能够让工程师知道它是设计成私有的，从而注意避开像公有行为那样调用它，如代码清单 5-115 所示。

代码清单 5-115 使用命名约定来模拟私有属性

---

```
//定义 Animal 类
function Animal(name){
    //公有属性
    this.name=name || "xxx";
    this.type="animal";
    //私有属性
    this._age=20;
}
//修改 Animal 类的原型
Animal.prototype={
    //以_开头为私有方法
    _move : function(){
        alert("I'm moving now");
    },
    //公有方法
    say : function(){
        alert("I'm a(an) "+this.type+" , my name is "+this.name+" , I'm "+this._age);
    },
    act : function(){
        this._move();
    }
}
//实例化 Animal 类
var myDog=new Animal("wangcai");
myDog.say();           //I'm a(an) animal , my name is wangcai , I'm 20
myDog.act();           //I'm moving now
myDog._move();         //I'm moving now (不推荐实例直接调用_move, 违反命名约定)
alert(myDog._age);     //20      (不推荐实例直接调用_age, 违反命名约定)
```

---

一些 OO 的极端主义者认为，直接访问类的属性是不好的，所有的属性都应该设为私有，然后通过 `get` 和 `set` 方法来获取和设置属性，如代码清单 5-116 所示。

## 代码清单 5-116 通过 get 和 set 方法访问属性

```

function Animal(name){
    var name;
    var type="animal";
    var _age=20;
    this.getName=function(){
        return name;
    }
    this.setName=function(o){
        name=o;
    }
    this.getType=function(){
        return type;
    }
    this.setType=function(o){
        type=o;
    }
    this._getAge=function(){
        return _age;
    }
    this._setAge=function(o){
        _age=o;
    }
    this.setName(name);
}
Animal.prototype={
    _move : function(){
        alert("I'm moving now");
    },
    say : function(){
        alert("I'm a(an) "+this.getType()+", my name is "+this.getName()+" , I'm "+this._getAge());
    },
    act : function(){
        this._move();
    }
}

var myDog=new Animal("wangcai");
myDog.say(); //I'm a(an) animal , my name is wangcai, I'm 20
myDog.setType("dog"); //通过 set 方法设置 type 属性的值
alert(myDog.getType()); //通过 get 方法获取 type 属性的值

```

通过 set 和 get 方法来访问属性需要将相关的方法都放到构造函数里，和把所有方法都放到原型里相比，这么做会占用更多内存，但它可以更好地保护属性，例如我们想让 Animal 类的 name 属性只能为 wangcai 或 xiaoqiang，而 type 属性为只读的，我们该

怎么做呢？我们可以在 `setName` 方法里做些判断，在给 `name` 属性赋值以前，判断一下赋的值是否满足条件，对 `type` 属性，我们可以在 `setType` 方法里去掉赋值语句，改用一个提示提醒用户这是个只读属性，如代码清单 5-117 所示。

代码清单 5-117 通过 `set` 方法保护属性

```

function Animal(name) {
    var name;
    var type="animal";
    var _age=20;
    this.getName=function(){
        return name;
    }
    this.setName=function(o){
        if(o != "wangcai" && o != "xiaoqiang"){
            alert("您设置的 name 值不合要求");
            return;
        }
        name=o;
    }
    this.getType=function(){
        return type;
    }
    this.setType=function(o){
        alert("赋值失败，Animal 类的 type 属性是只读的");
    }
    this._getAge=function(){
        return _age;
    }
    this._setAge=function(o){
        _age=o;
    }
    this.setName(name);
}
Animal.prototype={
    _move : function(){
        alert("I'm moving now");
    },
    say : function(){
        alert("I'm a(an) "+this.getType()+", my name is "+this.getName()+" , I'm "+this._getAge());
    },
    act : function(){
        this._move();
    }
}

var myDog=new Animal("wangcai");
myDog.say(); //I'm a(an) animal , my name is wangcai , I'm 20
myDog.setName("abc"); //您设置的 name 值不合要求

```

```

alert(myDog.getName());           //wangcai
myDog.setName("xiaoqiang");
alert(myDog.getName());           //xiaoqiang
myDog.setType("dog");            //赋值失败, Animal 类的 type 属性是只读的

```

你当然可以在编程的时候小心地注意着哪些属性有什么限制要求，不通过 set 和 get 方法对属性进行保护，仍然使用 this.xxx 来对属性进行直接的读写，从而节省一点内存。对于简单的应用，我推荐使用 this.xxx 来读写属性；还有一些复杂的应用，对扩展性和健壮性的要求很高，此时使用 set 和 get 方法读写属性更为合适。

另外，如果使用 set 方法来设置属性，那么我们就有了监听属性 valueChange 的入口，如代码清单 5-118 所示。

代码清单 5-118 监听属性的 valueChange

```

function Animal(name) {
    var name=name;
    var type="animal";
    var _age=20;
    //添加 master 属性，默认值为 adang
    var master="adang";
    this.getName=function(){
        return name;
    }
    this.setName=function(o){
        if(o != "wangcai" && o != "xiaoqiang"){
            alert("您设置的 name 值不合要求");
            return;
        }
        name=o;
        //触发 name 属性的 valueChange 事件
        this._valueChangeHandler("name");
    }
    //master 属性的获取方法
    this.getMaster=function(){
        return master;
    }
    //master 属性的设置方法
    this.setMaster=function(o){
        master=o;
        //触发 master 属性的 valueChange 事件
        this._valueChangeHandler("master");
    }
    this.getType=function(){
        return type;
    }
    this.setType=function(o){
        alert("赋值失败, Animal 类的 type 属性是只读的");
    }
}

```

```

        this._getAge=function(){
            return _age;
        }
        this._setAge=function(o){
            _age=o;
        }
    }
Animal.prototype={
    _move : function(){
        alert("I'm moving now");
    },
    say : function(){
        alert("I'm a(an) "+this.getType()+", my name is"+this.getName()
+" , I'm "+this._getAge());
    },
    act : function(){
        this._move();
    },
    //公有行为，用于注册属性的 valueChange 事件
    onChange : function(valueName,fun){
        this["_"+valueName+"ChangeHadlers"] = this["_"+valueName+
ChangeHadlers"] || [];
        this["_"+valueName+"ChangeHadlers"].push(fun);
    },
    //私有行为，属性 valueChange 的处理函数
    _valueChangeHandler : function(valueName){
        var o=this["_"+valueName+"ChangeHadlers"];
        if(o){
            for(var i=0,n=o.length;i<n;i++){
                var methodName="get"+valueName.charAt(0).toUpperCase()+
valueName.slice(1);
                o[i](this[methodName]());
            }
        }
    }
}

var myDog=new Animal("wangcai");
//给 myDog 注册 name 属性的 valueChange 事件
myDog.onChange("name",function(o){
    //回调函数接收新的 value 值作为参数，可以对其进行匹配
    if(o == "xiaoqiang"){
        alert(1);
    } else {
        alert(2);
    }
});
//给 myDog 换个新名字 xiaoqiang
myDog.setName("xiaoqiang");           //1
//给 myDog 再注册一个 name 属性的 valueChange 事件
myDog.onChange("name",function(o){
    alert("my new name is "+o);
});
//给 myDog 换个新名字 wangcai

```

```

myDog.setName("wangcai");           //2 , my new name is wangcai
//给 myDog 注册 master 属性的 valueChange 事件
myDog.onChange("master",function(o){
    alert("my new master is "+o);
})
//给 MyDog 换个主人
myDog.setMaster("Yang Fuchuan");    //my new master is Yang Fuchuan
var myDog2=new Animal("xiaoqiang");
//给 myDog2 注册 master 属性的 valueChange 事件
myDog2.onChange("master",function(o){
    alert(o+" is my new master");
})
myDog2.setMaster("Zhou Yubo");       //Zhou Yubo is my new master

```

真实世界中，我们的很多思维习惯都是状态驱动的，编程时监听属性的 valueChange 事件可以帮我们更接近真实世界的思维习惯。

### 3. 继承

正统面向对象语言都会提供 extend 之类的方法用于处理类的继承，但 JavaScript 并不提供 extend 方法，在 JavaScript 中使用继承需要用点技巧。

由图 5-18 可以看出，JavaScript 中实例的属性和行为是由构造函数和原型两部分共同组成的，我们定义两个类：Animal 和 Bird，它们在内存中的表现如图 5-18 所示。



图 5-18 Animal 类和 Bird 类

如果想让 Bird 继承自 Animal，那么我们需要把 Animal 构造函数和原型中的属性和行为全都传给 Bird 的构造函数和原型，如图 5-19 所示。

Ok，了解继承的思路后，那么让我们一步一步完成 Animal 和 Bird 的继承功能。首先，我们需要一个 Animal 类，如代码清单 5-119 所示。



图 5-19 Animal 类和 Bird 类的继承示意图

## 代码清单 5-119 定义 Animal 类

---

```
//定义 Animal 类
function Animal(name) {
    this.name=name;
    this.type="animal";
}
Animal.prototype={
    say : function(){
        alert("I'm a(an) "+this.type+" , my name is "+this.name);
    }
}
```

---

接下来，我们要定义一个 Bird 类，如代码清单 5-120 所示。

## 代码清单 5-120 定义 Bird 类

---

```
//定义 Animal 类
function Animal(name) {
    this.name=name;
    this.type="animal";
}
Animal.prototype={
    say : function(){
        alert("I'm a(an) "+this.type+" , my name is "+this.name);
    }
}

//定义 Bird 类
function Bird(){
}
Bird.prototype={}
```

---

Bird 类虽然有自己特有的属性和行为，但它大部分属性和行为和 Animal 相同，需要继承自 Animal 类。如前所述，JavaScript 中继承是要分别继承构造函数和原型中的属

性和行为的。我们先让 Bird 继承 Animal 的构造函数中的属性和行为，如代码清单 5-121 所示。

代码清单 5-121 继承构造函数中的属性和行为

---

```
//定义 Animal 类
function Animal(name) {
    this.name=name;
    this.type="animal";
}
Animal.prototype={
    say : function(){
        alert("I'm a(an) "+this.type+" , my name is "+this.name);
    }
}

//定义 Bird 类
function Bird(name) {
    this.name=name;
    this.type="animal";
}
Bird.prototype={

}

//实例化 Bird 对象
var myBird=new Bird("xiaocui");
alert(myBird.type);
```

---

运行正常，但我怎么没看出继承的“味道”呢？我们在 Bird 的构造函数中将 Animal 的属性和行为全都复制了一份，与其说是“继承”，不如说是“真巧，这两个类的构造函数除了函数名不同，其他地方都长得一样”。它的缺点很明显：如果 Animal 类的构造函数有任何变动，我们也需要手动地修改同步 Bird 类的构造函数，同样一份代码，我们复制了一份写在了程序中的不同地方，这违反了 DRY 原则，降低了代码的可维护性。

是的，我们需要改进它，如代码清单 5-122 所示。

代码清单 5-122 改进构造函数中的继承

---

```
//定义 Animal 类
function Animal(name) {
    this.name=name;
    this.type="animal";
}
Animal.prototype={
```

```

say : function(){
    alert("I'm a(an) "+this.type+" , my name is "+this.name);
}
}

//定义 Bird类
function Bird(name){
    Animal(name);
}
Bird.prototype={

}

//实例化 Bird 对象
var myBird=new Bird("xiaocui");
alert(myBird.type);           //undefined

```

---

我们在 Bird 类的构造函数里调用 Animal () 函数，希望它内部的 this.×××可以在 Bird 类的构造函数里执行一遍，但奇怪的是，出现 “alert(myBird.type)” 时，弹出的居然是 undefined，这是怎么回事呢？

这和 Animal 的调用方式有关。在 JavaScript 中，function 有两种不同的用法：

1) 作为函数存在，直接使用 “()” 进行调用，例如 “function test () {}; test ();” test 被用做函数，直接被 “()” 符号调用；2) 作为类的构造函数存在，使用 new 调用，例如 “function test () {}; new test ();” test 作为类的构造函数，通过 new 进行 test 类的实例化。这两种方法的调用，function 内部的 this 指向会有所不同——作为函数的 function，其 this 指向的是 window 对象，而作为类构造函数的 function，其 this 指向的是实例对象。

代码清单 5-122 中，Bird 类构造函数中的 Animal 是通过函数方式调用的，它内部的 this 指向的是 window 对象，其效果等同于代码清单 5-123。

#### 代码清单 5-123 实际效果

---

```

//定义 Animal类
function Animal(name){
    this.name=name;
    this.type="animal";
}
Animal.prototype={
    say : function(){
        alert("I'm a(an) "+this.type+" , my name is "+this.name);
    }
}

```

```
//定义 Bird 类
function Bird(name) {
    window.name=name;
    window.type="animal";
}
Bird.prototype={

}

//实例化 Bird 对象
var myBird=new Bird("xiaocui");
alert(myBird.type);           //undefined
alert(type);                 //animal (window.type 可省略写成 type)
```

如果想达到代码清单 5-121 的效果，让 Animal 内部的 this 指向 Bird 类的实例，可以通过 call 或 apply 方法实现，如代码清单 5-124 所示。

代码清单 5-124 使用 call 方法确定 this 的指向

```
//定义 Animal 类
function Animal(name) {
    this.name=name;
    this.type="animal";
}
Animal.prototype={
    say : function(){
        alert("I'm a(an) "+this.type+" , my name is "+this.name);
    }
}

//定义 Bird 类
function Bird(name){
    Animal.call(this,name);
}
Bird.prototype={

}

//实例化 Bird 对象
var myBird=new Bird("xiaocui");
alert(myBird.type);           //animal
```

构造函数的属性和行为已经成功实现了继承，接下来我们要实现原型中属性和行为的继承。既然 Bird 类需要和 Animal 类原型中同样的属性和行为，那么能否将 Animal 类的原型直接传给 Bird 类的原型，如代码清单 5-125 所示。

## 代码清单 5-125 原型的继承

```
// 定义 Animal 类
function Animal(name) {
    this.name=name;
    this.type="animal";
}
Animal.prototype={
    say : function(){
        alert("I'm a(an) "+this.type+" , my name is "+this.name);
    }
}

// 定义 Bird 类
function Bird(name){
    Animal.call(this,name);
}
Bird.prototype=Animal.prototype;

// 实例化 Bird 对象
var myBird=new Bird("xiaocui");
myBird.say(); // I'm a(an) animal , my name is xiao cui
```

通过将 Animal 类的原型传给 Bird 类的原型，Bird 类成功获得了 say 行为。但事情并不像想象中那么简单，如果我们想要给 Bird 类添加 fly 行为，会怎么样呢？如代码清单 5-126 所示。

## 代码清单 5-126 添加 fly 行为

```
// 定义 Animal 类
function Animal(name) {
    this.name=name;
    this.type="animal";
}
Animal.prototype={
    say : function(){
        alert("I'm a(an) "+this.type+" , my name is "+this.name);
    }
}

// 定义 Bird 类
function Bird(name){
    Animal.call(this,name);
}
Bird.prototype=Animal.prototype;
Bird.prototype.fly=function(){
    alert("I'm flying");
}

// 实例化 Bird 对象
```

---

```

var myBird=new Bird("xiaocui");
myBird.say(); //I'm a(an) animal , my name is xiao cui
myBird.fly(); //I'm flying
var myDog=new Animal("wangcai");
myDog.fly();           //I'm flying

```

---

我们只想给 Bird 类添加 fly 行为，为什么 Animal 类也获得了 fly 行为呢？这涉及传值和传址两个问题——在 JavaScript 中，赋值语句会用传值和传址两种不同的方式进行赋值，如果是数值型、布尔型、字符型等基本数据类型，在进行赋值时会将数据复制一份，将复制的数据进行赋值，也就是通常所说的传值，如果是数组、hash 对象等复杂数据类型(数组、hash 对象可包括简单类型数据)，在进行赋值时会直接用内存地址赋值，而不是将数据复制一份，用复制的数据进行赋值，也就是通常所说的传址。如代码清单 5-127 所示。

代码清单 5-127 传值与传址

---

```

var a=10;           //基本数据类型
var b=a;           //将变量 a 保存的值复制一份，传给变量 b，a 和 b 各保存一份数据
var c=[1,2,3];     //复杂数据类型
var d=c;           //将变量 c 指向的数据的内存地址传给变量 d，c 和 d 指向同一份数据
b++;
d.push(4);
alert(a);          // 10
alert(b);          // 11      变量 b 保存的数据更改不会影响到变量 a
alert(c);          // 1,2,3,4 变量 c 和 d 指向同一份数据，数据更改会互相影响
alert(d);          // 1,2,3,4

```

---

在原生 JavaScript 中，选择传值还是传址是根据数据类型自动判定的，但传址有时候会给我们带来意想不到的麻烦，所以我们需要对复杂类型数据的赋值进行控制，让复杂数据类型也可以进行传值。

最简单的做法是遍历数组或 hash 对象，将数组或 hash 对象这种复杂的数据拆分成一个个简单数据，然后分别赋值，如代码清单 5-128 所示。

代码清单 5-128 对复杂数据类型进行传值

---

```

var a=[1,2,3] , b={name:"adang",sex:"male",tel:"1234567"};
var c=[], d={};
for(var p in a){
    c[p]=a[p];
}
for(var p in b){
    d[p]=b[p];
}

```

---

```

}
c.push("4");
d.email="xxx@gmail.com";
alert(a);           //1,2,3
alert(c);           //1,2,3,4
alert(b.email);    //undefined
alert(d.email);    //xxx@gmail.com

```

值得一提的是，对于数组的传值还可以使用数组类的 slice 或 concat 方法实现，如代码清单 5-129 所示。

代码清单 5-129 数组传值的简单方法

```

var a=[1,2,3];
var b=a.slice() , c=a.concat();
b.pop();
c.push(4);
alert(a);           // 1,2,3
alert(b);           // 1,2
alert(c);           // 1,2,3,4

```

prototype 本质上也是个 hash 对象，所以直接用它赋值时会进行传址，这也是为什么在代码清单 5-126 中，myDog 居然会 fly 的原因。我们可以用 for in 来遍历 prototype，从而实现 prototype 的传值。但因为 prototype 和 function（用做类的 function）的关系，我们还有另一种方法实现 prototype 的传值——new SomeFunction()，如代码清单 5-130 所示。

代码清单 5-130 改进原型继承

```

// 定义 Animal 类
function Animal(name){
    this.name=name;
    this.type="animal";
}
Animal.prototype={
    say : function(){
        alert("I'm a(an) "+this.type+" , my name is "+this.name);
    }
}

// 定义 Bird 类
function Bird(name){
    Animal.call(this,name);
}
Bird.prototype=new Animal();
Bird.prototype.constructor=Bird;
Bird.prototype.fly=function(){

```

```

        alert("I'm flying");
    }

// 实例化 Bird 对象
var myBird=new Bird("xiaocui");
myBird.say(); // I'm a(an) animal , my name is xiao cui
myBird.fly(); // I'm flying
var myDog=new Animal("wangcai");
myDog.fly();           //报错, myDog.fly is not a function

```

细心的读者一定注意到了“`Bird.prototype.constructor=Bird`”这句，这是因为“`Bird.prototype=new Animal () ;`”时，`Bird.prototype.constructor` 指向了 `Animal`，我们需要将它纠正，重新指向 `Bird`。

这样的方式可以顺利实现 JavaScript 的继承，但我们还可以进一步将它进行封装，定义一个 `extend` 函数，如代码清单 5-131 所示。

代码清单 5-131 定义 extend 函数

```

function extend(subClass,superClass){
    var F=function() {};
    F.prototype=superClass.prototype;
    subClass.prototype=new F();
    subClass.prototype.constructor=subClass;
    subClass.superclass=superClass.prototype;
    if(superClass.prototype.constructor == Object.prototype.constructor){
        superClass.prototype.constructor=superClass;
    }
}

function Animal(name){
    this.name=name;
    this.type="animal";
}
Animal.prototype={
    say : function(){
        alert("I'm a(an) "+this.type+" , my name is "+this.name);
    }
}
function Bird(name){
    this.constructor.superclass.constructor.apply(this,arguments);
    this.type="bird"
}
extend(Bird,Animal);
Bird.prototype.fly=function(){
    alert("I'm flying");
}
var canary=new Bird("xiaocui");
canary.say();           // I'm a(an) bird , my name is xiaocui

```

```
canary.fly();           // I'm flying
```

### 5.4.3 用面向对象方式重写代码

在代码清单 5-103 中，我们已经用面向过程的方式写了电话本的程序，现在我们改用面向对象方式重写它。

首先，我们要进行 OOA，从现实逻辑中抽象出类。使用面向过程的编程方式时，处理函数是非常核心的部分，在命名的时候它很可能是个动词，例如 `getTel`、`addItem`，而面向对象的编程方式，最核心的部分是类，类的命名往往是个名词，例如 `Animal`、`Bird`。我们说明面向对象编程的时候，往往用一些客观世界真实存在的东西来举例，例如人、猫、狗，但类并不一定是客观存在的某个物件，例如这次我们要写的电话本程序，就很难对应到客观世界的某个真实存在的物件，它更像是一个逻辑上的物件，管理着关于电话记录的所有逻辑。它保存着许多电话记录，它可以用来添加、删除和查询电话记录。我们给它取个什么名字好呢？我觉得叫 `PhonebookManager` 不错，电话本管理者，是不是很符合这个类的职责？

现在让我们开始动手完成它吧，首先明确类名以及类的接口，至于接口里的具体逻辑如何实现，我们暂时先不用管。在 OOA 这一步，我们只需要用 UML 语言将类描述出来即可，如图 5-20 所示。

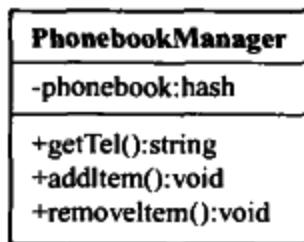


图 5-20 PhonebookManager 的 UML 图

UML 描述类的方式很简单，一个方框代表一个类，将方框划成上中下三栏，第一栏填入类名，第二栏填入类的属性，第三栏填入类的行为，其中公有属性和公有行为需要在属性和行为名前加上“+”号，而私有属性和私有行为需要在属性和行为名前加上“-”号。本例中，`PhonebookManager` 类有一个电话本属性 `phonebook`，记录着保存在

PhonebookManager 里的电话记录，有 `getTel`、`addItem` 和 `removeItem` 行为，分别用于查询电话、添加记录和删除记录。因为我们需要让类对外提供的接口尽可能少，除了必要的接口应该设为公有，其他都应该设为私有。我们并不需要 PhonebookManager 直接提供 `phonebook` 信息给用户，所以 `phonebook` 应设为私有。

一般来说，OOA 结束之后，我们需要进行 OOD。但本例实在太简单了，用不上 OOD。只有一些复杂的逻辑处理才可能用得上 OOD。

接下来，我们要将 OOA 和 OOD 的成果用程序编写出来，也就是 OOP 的环节。我们将一步步迭代完成它，首先，将类的框架搭好，如代码清单 5-132 所示。

代码清单 5-132 用面向对象方式重写电话本程序

---

```
// 定义电话本管理类
function PhonebookManager(){
    this._phonebook=null;
}
PhonebookManager.prototype={
    // 查询电话
    getTel : function(){

    },
    // 添加记录
    addItem : function(){

    },
    // 删除记录
    removeItem : function(){

    }
}
```

---

接下来，我们需要进一步细化接口。PhonebookManager 类的私有属性“`_phonebook`”在什么地方赋值比较好呢？因为每个 PhonebookManager 实例保存的“`_phonebook`”都很可能会有所不同，所以我想将“`_phonebook`”的值作为类构造函数的参数传进来应该是最好的选择；`getTel` 方法是用于查询电话号码的，它应该接受一个 `name` 参数作为查询的条件；`addItem` 方法是用于添加一条记录的，它应该接受 `name` 和 `tel` 两个参数，提供新记录需保存的信息；`removeItem` 方法是用于删除一条记录的，它应该接受 `name` 参数，用于指定要删除的记录，如代码清单 5-133 所示。

代码清单 5-133 定义行为的参数

```
//定义电话本管理类
function PhonebookManager(o) {
    this._phonebook=o;
}
PhonebookManager.prototype={
    //查询电话
    getTel : function(oName) {

    },
    //添加记录
    addItem : function(oName,oTel) {

    },
    //删除记录
    removeItem : function(oName) {

    }
}
```

接口细化之后，我们就要编写底层的具体逻辑了，如代码清单 5-134 所示。

代码清单 5-134 编写具体的行为代码

```
//定义电话本管理类
function PhonebookManager(o) {
    this._phonebook=o;
}
PhonebookManager.prototype={
    // 查询电话
    getTel : function(oName){
        var tel="";
        for(var i=0;i<this._phonebook.length;i++){
            if(this._phonebook[i].name == oName){
                tel=this._phonebook[i].tel;
                break;
            }
        }
        return tel;
    },
    //添加记录
    addItem : function(oName,oTel){
        this._phonebook.push({name:oName,tel:oTel});
    },
    //删除记录
    removeItem : function(oName){
        var n;
        for(var i=0;i<this._phonebook.length;i++){
            if(this._phonebook[i].name == oName){
                n=i;
                break;
            }
        }
        if(n!=undefined)
            this._phonebook.splice(n,1);
    }
}
```

```
        }
    }
    if(n != undefined){
        this._phonebook.splice(n,1);
    }
}
```

`PhonebookManager` 类定义完了，接下来，让我们看看如何使用它吧，如代码清单 5-135 所示。

代码清单 5-135 使用面向对象的电话本程序

```

//实例化两个电话本管理对象
var myPhonebookManager=new PhonebookManager([
    {name:"adang",tel:"111111"},  

    {name:"zhangxia",tel:"222222"},  

    {name:"yangfuchuan",tel:"333333"},  

    {name:"qiaojiwu",tel:"444444"},  

    {name:"zhouyubo",tel:"555555"}  

]),  

myPhonebookManager2=new PhonebookManager([  

    {name:"niaoren",tel:"111111"},  

    {name:"j2ee",tel:"222222"},  

    {name:"baobao",tel:"333333"}  

]);  

//从电话本1中查询niaoren的电话
var str=myPhonebookManager.getTel("niaoren");
alert(str); // ""  

//在电话本1中添加niaoren的记录
myPhonebookManager.addItem("niaoren","666666");
str=myPhonebookManager.getTel("niaoren");
alert(str); // "666666"  

//在电话本1中删除niaoren的记录
myPhonebookManager.removeItem("niaoren");
str=myPhonebookManager.getTel("niaoren");
alert(str); // ""  

//在电话本2中查询niaoren的记录
str=myPhonebookManager2.getTel("niaoren");
alert(str); // "111111"

```

电话本的程序顺利重构完了，接下来，再用面向对象的方式将 4.3 节中的 Tab 重构一下，如代码清单 5-136 所示。

## 代码清单 5-136 用面向对象方式重构 Tab

```

var GLOBAL={};
GLOBAL.namespace=function(str){
    var arr=str.split("."),o=GLOBAL;
    for (i=(arr[0] == "GLOBAL") ? 1 : 0; i<arr.length; i++) {
        o[arr[i]]=o[arr[i]] || {};
        o=o[arr[i]];
    }
}
GLOBAL.namespace("Dom");
GLOBAL.Dom.getElementsByClassName=function(str,root,tag){
    if(root){
        root=typeof root == "string" ?
document.getElementById(root) : root;
    } else {
        root=document.body;
    }
    tag=tag || "*";
    var els=root.getElementsByTagName(tag),arr=[];
    for(var i=0,n=els.length;i<n;i++){
        for(var j=0,k=els[i].className.split(" "),l=k.length;j<l;j++)
            if(k[j]==str){
                arr.push(els[i]);
                break;
            }
    }
    return arr;
}
GLOBAL.Dom.addClass=function(node,str){
    if(!new RegExp("(^|\\s+)" + str).test(node.className)){
        node.className=node.className+" "+str;
    }
}
GLOBAL.Dom.removeClass=function(node,str){
    node.className=node.className.replace(new
RegExp("(^|\\s+)" + str),"");
}
GLOBAL.namespace("Event");
GLOBAL.Event.on=function(node,eventType,handler,scope){
    node=typeof node=="string" ? document.getElementById(node) : node;
    scope=scope || node;
    if(document.all){

        node.attachEvent("on"+eventType,function(){handler.apply(scope,
            arguments)} );
    } else {

        node.addEventListener(eventType,function(){handler.apply(scope,
            arguments)} ,false);
    }
}

```

```
}

function Tab(config){
    this._root=config.root;
    this._currentClass=config.currentClass;
    var trigger=config.trigger || "click";
    this._handler=config.handler;
    var autoPlay=config.autoPlay;
    var playTime=config.playTime || 3000;
    this._tabMenus=GLOBAL.Dom.getElementsByClassName("J_tab-
menu",this._root);
    this._tabContents=GLOBAL.Dom.getElementsByClassName("J_tab-
content",this._root);
    this.currentIndex=0;
    var This=this;
    if(autoPlay){
        setInterval(function(){This._autoHandler()},playTime);
    }
    for(var i=0;i<this._tabMenus.length;i++){
        this._tabMenus[i]._index=i;
        GLOBAL.Event.on(this._tabMenus[i],trigger,function(){
            This.showItem(this._index);
            this.currentIndex=this._index;
        });
    }
}
Tab.prototype={
    showItem : function(n){
        for(var i=0;i<this._tabContents.length;i++){
            this._tabContents[i].style.display="none";
        }
        this._tabContents[n].style.display="block";
        if(this._currentClass){
            var currentMenu=GLOBAL.Dom.getElementsByClassName(this._currentClass,
this._root)[0];
            if(currentMenu){
                GLOBAL.Dom.removeClass(currentMenu,this._currentClass);
            }
            GLOBAL.Dom.addClass(this._tabMenus[n],this._currentClass);
        }
        if(this._handler){
            this._handler(n);
        }
    },
    _autoHandler : function(){
        this.currentIndex++;
        if(this.currentIndex >= this._tabMenus.length){
            this.currentIndex=0;
        }
        this.showItem(this.currentIndex);
    }
}
```

---

```

var tabs=GLOBAL.Dom.getElementsByClassName("J_tab");
new Tab({root:tabs[0],trigger:"mouseover"});
new Tab({root:tabs[1],currentClass:"tabcurrentMenu",autoPlay:true,play
Time:5000});
new Tab({root:tabs[2],currentClass:"tabcurrentMenu2",trigger:"mouseover"
,handler:function(index){alert("您激活的是第"+(index+1)+"个标签")}});

```

---

关于面向对象和面向过程的优缺点比较，我们已经在 5.4.1 节中进行了详细说明，当程序越复杂时，这些优缺点会表现得越明显。从面向过程到面向对象，是每个工程师早晚都要跨越的一道槛，它并不神秘，也不困难，如果你还在坚守面向过程的编程方式，你需要做的仅仅是勇于尝试换种思维方式去编程，相信我，你能掌握它，并且你会爱上它。

## 5.5 其他问题

### 5.5.1 prototype 和内置类

在 5.4.2 节中我们讲到了 JavaScript 语言中 prototype 和类的关系，本节我们将进一步了解 prototype 一些有趣的地方。

JavaScript 语言提供了一些内置类，包括 Array、String、Function 等，它们提供了 JavaScript 的大部分基本数据类型。这些内置类通常会提供一些方法和属性，例如 Array 类提供 length 属性，push、pop 方法，String 提供 length 属性，replace、split 方法，Function 提供 call、apply 方法等。我们在使用这些内置类时，可以直接调用它们提供的属性和方法。

需要说明的是，这些内置类不一定需要通过 new 的方式进行实例化，我们平时习惯用更简短的方式调用它们，但其本质上是一样的，如代码清单 5-137 所示。

代码清单 5-137 调用内置类

---

```

var a=new String("hello world"); //通过 new String() 实例化 string 类型对象
var b="hello world";           //直接通过""实例化 string 类型对象
alert(a.length);
alert(b.length);
var c=new Array(1,2,3);        // 通过 new Array() 实例化 array 类型对象
var d=[1,2,3];                // 直接通过[]实例化 array 类型对象

```

```
c.push(4);
d.pop();
alert(c);
alert(d);
```

只要是类就会有原型，不管它是自定义类还是 JavaScript 的内置类，我们可以通过修改内置类的原型，让 JavaScript 基本类型的对象获得一些有趣的功能。例如，在很多语言中，Array 具有 each、map 等方法的，但遗憾的是，JavaScript 语言中 Array 并不提供这些好用的方法。没关系，既然原生 JavaScript 并不提供这些方法，那么我们自己扩展它好了，如代码清单 5-138 所示：

代码清单 5-138 扩展内置类的行为

```
Array.prototype.each=function(fun){
    for(var i=0,n=this.length;i<n;i++){
        fun(this[i],i);
    }
}
Array.prototype.clone=function(){
    var o=[];
    this.each(function(v,k){
        o[k]=v;
    });
    return o;
}
Array.prototype.map=function(fun){
    var o=[];
    this.each(function(v,k){
        o[k]=fun(v,k);
    });
    return o;
}
//因为在 IE 中 delete 是保留字，所以方法名改用 Delete
Array.prototype.Delete=function(a){
    var o=this.clone();
    for(var i=o.length,n=0;i>n;i--){
        if(o[i] == a){
            o.splice(i,1);
        }
    }
    return o;
}
var a=[1,2,3,2,4,5];
var str="";
a.each(function(v,k){
    str += k+" : "+v+"\n";
});
alert(str);                                // 0 : 1 1:2 2:3 3:2 4:4 5:5
var b=a.map(function(v,k){
```

```

        return v * 10;
});
alert(a);                                // 1,2,3,2,4,5
alert(b);                                // 10,20,30,20,40,50
var c=b.Delete(20);
alert(c);                                // 10,30,40,50

```

这段代码最难理解的地方就在于扩展方法中 `this` 代表什么，在 5.4 节中我们讲过，无论在类的构造函数中还是在原型中，`this` 都指向实例化的对象。明白了这一点，以上代码也就不难理解了。在代码清单 5-138 中，我们给 `Array` 类扩展了 `each`、`clone`、`map` 和 `Delete` 方法，现在，我们的 `Array` 类型对象更加强大了。

除了可以扩展内置类的方法，我们还可以重写内置类的方法，如代码清单 5-139 所示。

代码清单 5-139 重写内置类的方法

```

var a=[1,2,3];
alert(a);                                // 1,2,3
Array.prototype.toString=function(str){
    return "I'm an array";
}
alert(a);                                // I'm an array

```

值得一提的是，“`alert(a)`”时，自动调用了 `a` 的 `toString` 方法。在需要字符串时，对象会隐式地自动调用 `toString` 方法，包括我们自定义的对象，如代码清单 5-140 所示。

代码清单 5-140 自定义 `toString` 方法

```

function Dog(o){
    this.name=o;
}
var myDog=new Dog("wang cai");
alert(myDog);                // [object Object]
Dog.prototype.toString=function(){
    return "my name is "+this.name;
}
alert(myDog);                // my name is wang cai
var me={
    name : "adang",
    email : "cly84920@gmail.com",
    toString : function(){
        return "I'm adang,my email is cly84920@gmail.com";
    }
}
alert(me);                    // I'm adang,my email is cly84920@gmail.com

```

给自定义类定义 `toString` 方法，可以为我们在调试时提供更多有用的信息。

内置类的方法可以重写，但属性却不能重写，如代码清单 5-141 所示。

代码清单 5-141 重写属性

---

```
Array.prototype.length=1;
String.prototype.length=1;
alert([1,2,3].length);           // 3
alert("abc".length);            // 3
```

---

在 JavaScript 中，包括内置类和自定义类，所有类的祖先类都是 `Object`，所以如果想对所有对象都扩展方法，可以通过修改 `Object` 类的原型实现，如代码清单 5-142 所示。

代码清单 5-142 扩展 Object 类

---

```
Object.prototype.test=function(){
    alert("hello world");
}
var a=[1,2,3],b="abc",c={},d=true,e=function()();
a.test();
b.test();
c.test();
d.test();
e.test();
function Dog(o){
    this.name=o;
}
Dog.prototype.toString=function(){
    return "my name is "+this.name;
}
var f=new Dog("wangcai");
f.test();
```

---

修改内置类的原型可以在编程时给我们带来很大方便，但也有些人非常排斥这种做法，认为它对内置类的原型造成了“污染”，因为内置类的原型也可以理解为是全局作用域的，如果对它进行修改，在多人合作时有可能对别人的代码造成影响。修改内置类的原型等于修改了统一的游戏规则，虽然可以带来很大的方便，但同时也会给多人合作带来冲突隐患，它是有副作用的。

所以有些人更愿意使用这样的方式来扩展内置类的方法，如代码清单 5-143 所示。

**代码清单 5-143 使用自定义类**

```
function myArray(o) {
    this.getArray=function() {
        return o;
    };
}
myArray.prototype={
    each : function(fun) {
        var o=this.getArray();
        for(var i=0,n=o.length;i<n;i++) {
            fun(o[i],i);
        }
    }
}
var a=new myArray([1,2,3]),str="";
a.each(function(v,k) {
    str += k+" : "+v+"\n";
});
alert(str);                                //0 : 1 1 : 2 2 : 3
```

代替直接修改内置类原型的做法，定义一个自定义类，将内置类的实例作为参数传给构造函数，在自定义类里定义扩展方法。这种做法的思路是将内置类再封装一层，以此保护内置类的原型不被污染。

两种方法都各有优缺点，修改内置类的原型非常方便，缺点是可能会带来冲突隐患；自定义类可以保护原型不被修改，但它需要用 new 来实例化自定义类，相对麻烦一点。如果是小应用，不用过多考虑可维护性，推荐使用前者，如果是大中型应用，需要考虑可维护性，推荐使用后者。

### 5.5.2 标签的自定义属性

JavaScript 和 HTML 标签之间是存在映射关系的，HTML 标签在 JavaScript 中作为 DOM 节点对象存在，如代码清单 5-144 所示。

代码清单 5-144 HTML 标签和 JavaScript 的映射

```
<a id="a" class="b" title="" href="http://www.adanghome.com"  
onclick="alert(this.href);return false;">my blog</a>  
<script type="text/JavaScript">  
    var node=document.getElementById("a");  
    alert(typeof node);  
    // object  
</script>
```

定义在 HTML 标签中的属性，在 JavaScript 里也可以获取到。获取方式有两种：

- 1) 通过 DOM 节点对象的 `getAttribute` 方法；2) 通过 DOM 节点对象的属性，如代码清单 5-145 所示。

**代码清单 5-145 在 JavaScript 中获得 DOM 节点对象的属性**

---

```
<a id="a" class="b" title="" href="http://www.adanghome.com"
onclick="alert(this.href);return false;">my blog</a>
<script type="text/JavaScript">
var node=document.getElementById("a");
alert(node.getAttribute("id"));                                // a
alert(node.id)                                              // a
```

---

大部分情况下，在 IE 和 Firefox 里 `node.getAttribute("×××")` 得到的结果相同。但对于某些属性，IE 和 Firefox 下用 `node.getAttribute("×××")` 得到的结果却是不同的。相比之下，通过 `node.×××` 得到的结果在 IE 和 Firefox 得到的结果会更一致，如代码清单 5-146 所示。

**代码清单 5-146 IE 和 firefox 获得 DOM 节点对象属性的差异**

---

```
<a id="a" class="b" title="我的博客" href="http://www.adanghome.com"
onclick="alert(this.href);return false;">my blog</a>
<script type="text/JavaScript">
var node=document.getElementById("a");

alert(node.getAttribute("id"));
/*
    IE 和 Firefox : a
*/

alert(node.getAttribute("class"));
/*
    IE      : null
    Firefox : b
*/

alert(node.getAttribute("className"));
/*
    IE      : b
    Firefox : null
*/

alert(node.getAttribute("title"));
/*
    IE 和 Firefox : 我的博客
*/
```

---

```
alert(node.getAttribute("href"));
/*
    IE 和 Firefox : http://www.adanghome.com
*/

alert(node.getAttribute("onclick"));
/*
    IE      : function onclick(){
                alert(this.href);return false;
            }
    Firefox : alert(this.href);return false;
*/

alert(node.getAttribute("innerHTML"));
/*
    IE      : my blog
    Firefox : null
*/

alert(node.id);
/*
    IE 和 Firefox : a
*/

alert(node.className);
/*
    IE 和 Firefox : b
*/

alert(node.title);
/*
    IE 和 Firefox : 我的博客
*/

alert(node.href);
/*
    IE 和 Firefox : http://www.adanghome.com
*/

alert(node.onclick);
/*
    IE      : function onclick(){
                alert(this.href);return false;
            }
    Firefox : function onclick(event){
                alert(this.href);
                return false;
            }
*/

alert(node.innerHTML);
/*
    IE 和 Firefox : my blog
*/
```

需要注意的是，class 是 JavaScript 的保留字，所以在获取 HTML 标签的 class 属性时，要改用 className。从上面的示例中可以看出，使用 node.×××的方式获取 HTML 标签的属性值，跨浏览器兼容性比 node.getAttribute("×××")好。

在 HTML 标签中定义的属性，在 JavaScript 里可以获取。反之，在 JavaScript 中定义的属性，也可以映射到 HTML 标签中，如代码清单 5-147 所示。

代码清单 5-147 JavaScript 和 HTML 间的映射

---

```
<style type="text/CSS">
    .red{color:red;}
</style>
hello world
<script type="text/JavaScript">
    var node=document.getElementById("a");
    node.className="red";
</script>
```

---

其效果等同于hello world。

以上示范的都是 HTML 标签的常规属性，除了常规属性，我们还可以给 HTML 标签定义一些自定义属性，这些自定义属性同样可以在 JavaScript 中获取。但和常规属性不同，Firefox 下无法通过 node.××× 获取到自定义属性值，只能使用 node.getAttribute("×××")获取，如代码清单 5-148 所示。

代码清单 5-148 自定义属性的获取

---

```
<a id="a" href="http://www.adanghome.com" blogName="阿当的博客"
    blogType="前端开发">my blog</a>
<script type="text/JavaScript">
    var node=document.getElementById("a");

    alert(node.blogName);
    /*
        IE: 阿当的博客
        Firefox : undefined
    */

    alert(node.blogType);
    /*
        IE: 前端开发
        Firefox : undefined
    */

    alert(node.getAttribute("blogName"));
```

---

```

/*
    IE 和 Firefox : 阿当的博客
*/

alert(node.getAttribute("blogType"));
/*
    IE 和 Firefox : 前端开发
*/
</script>

```

从兼容性考虑，笔者建议对于常规属性，统一使用 `node.×××` 的方式读取，对于自定义属性，统一使用 `node.getAttribute("×××")` 读取。

自定义属性是个非常有用的技巧，我们不但可以用它来保存普通字符串，借助一点小技巧，还可以用它保存其他类型的数据，例如数组、hash 对象。因为属性只可能是字符串类型的，哪怕它的格式和 hash 对象（或数组）一模一样，它也只是长得像 hash 对象（或数组）的字符串而已。如果我们想将 hash 对象（或数组）等数据保存到属性里，就涉及一个数据反序列化问题。

将复杂类型的数据转化成字符串，称为数据的序列化，其逆操作叫做数据的反序列化。数据的反序列化，最经典的应用当数 Ajax 了。Ajax 本来只支持字符串和 xml 这两种格式的返回数据，但聪明的工程师们创造出了另一种格式：大名鼎鼎的 json——选择返回数据为字符串格式，但让这个字符串长得像 hash 对象（或数组），将返回数据进行反序列化，得到真正的 hash 对象（或数组）。

字符串的反序列化是通过 `eval` 函数实现的。只要字符串长得像 JavaScript 支持的数据格式，就可以进行反序列化，而与是不是 Ajax 的返回数据无关。我们的自定义属性，虽然类型为字符串，但如果它的值长得像 hash 对象（或数组），我们就可以将它反序列化成真正的 hash 对象（或数组），如代码清单 5-149 所示。

代码清单 5-149 自定义属性的反序列化

```

<a id="a" href="http://www.adanghome.com" blogInfo="{name:'阿当的博客',
    type:'前端开发'}">my blog</a>

<script type="text/JavaScript">
    var node=document.getElementById("a");
    var info=node.getAttribute("blogInfo");
    alert(typeof info);           // string
    alert(info.name);            // undefined

```

```

    alert(info.type);                      // undefined
    info=eval("(+"+info+")");
    alert(typeof info);                  // object
    alert(info.name);                   // 阿当的博客
    alert(info.type);                   // 前端开发
</script>

```

---

### 5.5.3 标签的内联事件和 event 对象

`event` 对象在 IE 和 Firefox 下的表现是不同的。在 IE 下，`event` 是 `window` 对象的一个属性，是在全局作用域下的，而在 Firefox 里，`event` 对象作为事件的参数存在，如代码清单 5-150 所示。

代码清单 5-150 简单的点击事件

---

```

<input type="button" id="btn" value="click me" />

<script type="text/JavaScript">
document.getElementById("btn").onclick=function(){
    alert(arguments.length);
};
</script>

```

---

这段代码在 IE 下弹出 0，而在 Firefox 下弹出 1。在 Firefox 下这个参数就是 `event` 对象了。

如果在标签内联事件中触发事件又会如何呢？如代码清单 5-151 所示。

代码清单 5-151 内联事件中的点击事件

---

```

<input type="button" id="btn" value="click me" onclick="handler()" />

<script type="text/JavaScript">
function handler(){
    alert(arguments.length);
};
</script>

```

---

在 IE 和 Firefox 下，这段代码弹出的都是 0。也就是说，标签内联事件并没有被替换成代码清单 5-152 所示的代码。

**代码清单 5-152 替换可能一**


---

```
btn.onclick=handler;
function handler(){
    alert(arguments.length);
}
```

---

它被替换成代码清单 5-153 所示的代码。

**代码清单 5-153 替换可能二**


---

```
btn.onclick=function(){
    handler();
}
function handler(){
    alert(arguments.length);
}
```

---

在标签内联事件中，我们使用 `arguments[0]`可以在 Firefox 下访问到 `event` 对象，如代码清单 5-154 所示。

**代码清单 5-154 内联事件和参数**


---

```
<input type="button" id="btn" value="click me"
onclick="alert(arguments[0].type)" />
```

---

不使用标签内联事件时，我们可以给处理函数传参，从而指定 `arguments[0]` 的变量名，如代码清单 5-155 所示。

**代码清单 5-155 event 对象的兼容处理**


---

```
<input type="button" id="btn" value="click me" />

<script type="text/JavaScript">
document.getElementById("btn").onclick=function(e){
    e=window.event || e; //e 兼容 IE 和 Firefox，指向 event 对象
};
</script>
```

---

在标签内联事件中，我们没办法指定参数名，是不是就没办法直接写个变量在 IE 和 Firefox 下兼容地指 `event` 对象呢？不是的，可以用 `event` 这个变量名兼容地指向 `event` 对象，注意，只能是 `event`，诸如 `a`、`b`、`Event` 之类的全都不行，如代码清单 5-156 所示。

**代码清单 5-156 在内联事件中兼容 event 对象**


---

```
<input type="button" id="btn" value="click me"
onclick="alert(event.type);"/>
```

---

这段代码在 IE 和 Firefox 下都可以正确地弹出 click。

有趣的是，标签内联事件中我们甚至可以写注释，可以使用字符串，如代码清单 5-157 所示。

**代码清单 5-157 内联事件的使用**


---

```
//只弹出 1
<input type="button" id="btn" value="click me"
onclick="alert(1);//alert(2);alert(3);"/>

//弹出 1 和 3
<input type="button" id="btn" value="click me"
onclick="alert(1);/*alert(2);*/alert(3);"/>

//弹出 string
<input type="button" id="btn" value="click me" onclick="var a='abc';
alert(typeof a);"/>
```

---

如果我们既用标签内联事件绑定了事件，又用 `DomNode.onclick` 绑定了事件，又会如何呢？如代码清单 5-158 所示。

**代码清单 5-158 在两处同时监听事件**


---

```
<input type="button" id="btn" value="click me" onclick="alert(123);"/>

<script type="text/JavaScript">
document.getElementById("btn").onclick=function(){
    alert(456);
};
</script>
```

---

运行上述代码会弹出 456，不弹出 123。等同于代码清单 5-159。

**代码清单 5-159 等同效果**


---

```
<input type="button" id="btn" value="click me" />

<script type="text/JavaScript">
document.getElementById("btn").onclick=function(){
    alert(123);
};
```

```
document.getElementById("btn").onclick=function(){
    alert(456);
};
</script>
```

后面的处理函数覆盖了前面的处理函数。如果通过 `attachEvent` 和 `addEventListener` 来绑定事件的呢？如代码清单 5-160 所示。

代码清单 5-160 改进方案，同时监听两处事件

```
<input type="button" id="btn" value="click me" onclick="alert(123);"/>

<script type="text/JavaScript">
function handler(){
    alert(456);
}
if(document.all){
    btn.attachEvent("onclick",handler);
} else {
    btn.addEventListener("click",handler,false);
}
</script>
```

运行很顺利，先弹出了 123，后又弹出了 456。

#### 5.5.4 利用事件冒泡机制

在本节我们要编写一个打分的程序，假如我开了家饭店，我希望让顾客对我的饭店进行评分，它的界面如图 5-21 所示。

打分的功能非常常见，我就不详述其功能了。现在让我们开始动手制作吧，首先，我们需要两张星星的图片，一张是白色的星星，一张是黄色的星星，我们分别将这两张图片命名为 `star.gif` 和 `star2.gif`（当然，你也可以使用其他类型的图片，例如 `png` 和 `jpg`），如图 5-22 和图 5-23 所示。



图 5-22 star.gif

图 5-23 star2.gif

阿当饭店

卫生：



价格：



味道：



图 5-21 阿当饭店  
打分界面

接下来，我们开始编写打分程序的代码，如代码清单 5-161 所示。

## 代码清单 5-161 打分程序

```

<!-- 编写 rate 相关的 html 结构 -->
<h1>阿当饭店</h1>
<p>卫生: <p>
<p class="J_rate">
    
    
    
    
    
</p>
<p>价格: <p>
<p class="J_rate">
    
    
    
    
    </p>
<p>味道: <p>
<p class="J_rate">
    
    
    
    
    
</p>
<script type="text/JavaScript">
    // 封装 DOM、Event 接口
    var GLOBAL={};
    GLOBAL.namespace=function(str){
        var arr=str.split("."),o=GLOBAL;
        for (i=(arr[0] == "GLOBAL") ? 1 : 0; i<arr.length; i++) {
            o[arr[i]]=o[arr[i]] || {};
            o=o[arr[i]];
        }
    }
    GLOBAL.namespace("Dom");
    GLOBAL.Dom.getElementsByClassName=function(str,root,tag){
        if(root){
            root=typeof root == "string" ? document.getElementById
(root) : root;
        } else {
            root=document.body;
        }
        tag=tag || "*";
        var els=root.getElementsByTagName(tag),arr=[];
        for(var i=0,n=els.length;i<n;i++){
            for(var j=0,k=els[i].className.split(" "),l=k.
length;j<l;j++){
                if(k[j] == str){
                    arr.push(els[i]);
                    break;
                }
            }
        }
        return arr;
    }
}

```

```
        }
    }
    return arr;
}
GLOBAL.namespace("Event");
GLOBAL.Event.on=function(node,eventType,handler,scope){
    node=typeof node == "string" ? document.getElementById
(node) : node;
    scope=scope || node;
    if(document.all){

        node.attachEvent("on"+eventType,function(){handler.apply(scope,
arguments)} );
    } else {

        node.addEventListener(eventType,function(){handler.apply(scope,
arguments)},false);
    }
}

// 定义 Rate类
function Rate(rateRoot){
    var root=typeof rateRoot == "string" ? document.getElementById
ById(rateRoot) : rateRoot;
    var items=root.getElementsByTagName("img");
    var imgs=["star.gif","star2.gif"];
    var rateFlag;
    for(var i=0,n=items.length;i<n;i++) {
        items[i].index=i;
        GLOBAL.Event.on(items[i],"mouseover",function(){
            if(rateFlag) return;
            for(var j=0;j<n;j++){
                if(j<=this.index){
                    items[j].src=imgs[1];
                } else {
                    items[j].src=imgs[0];
                }
            }
        });
        GLOBAL.Event.on(items[i],"mouseout",function(){
            if(rateFlag) return;
            for(var j=0;j<n;j++){
                items[j].src=imgs[0];
            }
        });
        GLOBAL.Event.on(items[i],"click",function(){
            if(rateFlag) return;
            rateFlag=true;
            alert("您打了"+(this.index+1)+"分");
        });
    }
}
```

```
// 实例化 Rate 类
var rateNodes=GLOBAL.Dom.getElementsByClassName("J_rate");
for(var i=0,n=rateNodes.length;i<n;i++){
    new Rate(rateNodes[i]);
}
</script>
```

Rate 类有个私有属性 items，对应 HTML 中的五颗星星，这段程序中，对每颗星星监听了 mouseover、mouseout 和 click 事件。效果等同于代码清单 5-162。

代码清单 5-162 等同效果

```
<p class="J_rate">
    
    
    
    
    
</p>
```

之前我们讲到过事件冒泡，利用事件冒泡我们可以进一步优化代码，如代码清单 5-163 所示。

代码清单 5-163 利用事件冒泡进行优化

```
// 定义 Rate 类
function Rate(rateRoot){
    var root=typeof rateRoot == "string" ?
document.getElementById(rateRoot) : rateRoot;
    var items=root.getElementsByTagName("img");
    var imgs=["star.gif","star2.gif"];
    var rateFlag;
    for(var i=0,n=items.length;i<n;i++){
        items[i].index=i;
    }
    GLOBAL.Event.on(root,"mouseover",function(e){
        if(rateFlag) return;
        var target=e.target || e.srcElement;
        if(target.tagName.toLowerCase() != "img") return;
        for(var i=0;i<n;i++){
            if(i<=target.index){
                items[i].src=imgs[1];
            } else {
                items[i].src=imgs[0];
            }
        }
    });
}
```

```

        }
    });
GLOBAL.Event.on(root,"mouseout",function(e){
    if(rateFlag) return;
    var target=e.target || e.srcElement;
    for(var i=0,n=items.length;i<n;i++) {
        items[i].src=imgs[0];
    }
});
GLOBAL.Event.on(root,"click",function(e){
    if(rateFlag) return;
    rateFlag=true;
    var target=e.target || e.srcElement;
    alert("您打了"+(target.index+1)+"分");
});
}

```

---

冒泡的思路是在祖先节点上监听事件，结合 `event.target/event.srcElement` 来实现最终效果，其效果等同于代码清单 5-164。

**代码清单 5-164 等同效果**

---

```

<p class="J_rate" onmouseover="..." onmouseout="..." onclick="...">





</p>

```

---

利用冒泡可以让事件挂钩更干净，有效减小内存开销。

### 5.5.5 改变 DOM 样式的三种方式

JavaScript 编程很重要的一个功能就是用于改变 DOM 节点的样式。

改变 DOM 节点样式最简单最直接的方法是通过设置 `DomNode` 的 `style` 属性，如代码清单 5-165 所示。

**代码清单 5-165 改变 DOM 节点样式的方法一**

---

```

<span id="test">hello world</span>
<script type="text/JavaScript">
    var node=document.getElementById("test");
    node.style.color="red";
</script>

```

---

“hello world”的颜色会变成红色，其效果等同于代码清单 5-166。

#### 代码清单 5-166 等同效果

---

```
<span id="test" style="color:red">hello world</span>
```

---

如果我们需要同时设置多个样式呢？我们仍然可以通过设置 DomNode 的 style 属性实现，如代码清单 5-167 所示。

#### 代码清单 5-167 用方法一改变多个样式

---

```
<span id="test">hello world</span>
<script type="text/JavaScript">
    var node=document.getElementById("test");
    node.style.color="red";
    node.style.backgroundColor="black";
    node.style.fontSize="40px";
    node.style.fontWeight="bold";
</script>
```

---

将每一项我们需要设置的样式都设置一遍，其效果等同于代码清单 5-168。

#### 代码清单 5-168 等同效果

---

```
<span id="test" style="color:red;background-color:black;font-size:40px;font-weight:bold">hello world</span>
```

---

但这种写法会让 JavaScript 代码较长，而且过多地承担起了表现层的职责。表现层应该是由 CSS 控制的，所以我们有个更好的方法来处理这种样式的批量操作，如代码清单 5-169 所示：

#### 代码清单 5-169 改变 DOM 节点样式的方法二

---

```
<style type="text/CSS">
    .testStyle{color:red;background-color:black;font-size:40px;font-weight:bold;}
</style>
<span id="test">hello world</span>
<script type="text/JavaScript">
    var node=document.getElementById("test");
    node.className="testStyle";
</script>
```

---

我们在 CSS 中预设好某个 class，然后给需要样式的 DOM 节点设置 className 属性

性，从而和样式中预设的 class 挂接上。其效果等同于代码清单 5-170。

代码清单 5-170 等同效果

---

```
<style type="text/CSS">
  .testStyle{color:red;background-color:black;font-size:40px;font-
weight:bold;}
</style>
hello world</span>
```

---

如果我们需要给多个 DOM 节点批量设置样式呢？例如代码清单 5-171 所示的结构。

代码清单 5-171 需设置样式的 DOM 节点

---

```
<span id="test">hello world</span>
<span>aaaaaa</span>
<span>bbbbbb</span>
<span>cccccc</span>
```

---

我们想要批量设置标签的样式，设置 font-size 为 40px，background 为 #000，除了 id 为 test 的 span 的 color 为 red，其他 span 的 color 都为#fff。

无论设置 DomNode 的 style 属性，还是设置 className 属性，都是针对单个 DOM 节点的操作。我们可以逐个进行设置，但其效果等同于代码清单 5-172 和 5-173。

代码清单 5-172 等同效果一

---

```
<span id="test" style="font-size:40px;background:#000;color:red">hello
world</span>
<span style="font-size:40px;background:#000;color:#fff">aaaaaa</span>
<span style="font-size:40px;background:#000;color:#fff">bbbbbb</span>
<span style="font-size:40px;background:#000;color:#fff">cccccc</span>
```

---

代码清单 5-173 等同效果二

---

```
<style type="text/CSS">
  .testStyle{font-size:40px;background:#000;color:#fff}
  #test{color:red}
</style>
hello world</span>
aaaaaa</span>
bbbbbb</span>
cccccc</span>
```

---

这两种方式都不支持批量处理。接下来，我们介绍第三种改变 DOM 节点样式的方法，如代码清单 5-174 所示。

代码清单 5-174 改变 DOM 节点样式的方法三

---

```

<span id="test">hello world</span>
<span>aaaaaa</span>
<span>bbbbbb</span>
<span>cccccc</span>
<script type="text/JavaScript">
    function addStyleNode(str){
        var styleNode=document.createElement("style");
        styleNode.type="text/CSS";
        if(styleNode.styleSheet){
            styleNode.styleSheet.cssText=str;
        } else {
            styleNode.innerHTML=str;
        }
        document.getElementsByTagName("head")[0].appendChild(styleNode);
    }
    addStyleNode("span{font-size:40px;background:#000;color:#fff}
#test{color:red}");
</script>

```

---

这种方式的思路是创建一个`<style>`标签，然后往里填入 CSS，再将它添加到页面里。这种方式让我们可以直接在 JavaScript 里按 CSS 的语法规则来编写样式，更灵活地控制 DOM 节点的样式，其效果相当于代码清单 5-175。

代码清单 5-175 等同效果

---

```

<style type="text/CSS">
    span{font-size:40px;background:#000;color:#fff}
    #test{color:red}
</style>
<span id="test">hello world</span>
<span>aaaaaa</span>
<span>bbbbbb</span>
<span>cccccc</span>

```

---

需要注意的是，`<style>`的 DOM 节点在 Firefox 可以直接对 `innerHTML` 属性进行写操作，但在 IE 下，它的 `innerHTML` 属性是只读的。IE 下要通过 `styleSheet.cssText` 进行写操作。

## 写在规则前面的话

项目的可维护性第一。我们并不是一个人在做事，项目的维护和二次开发可能是直接或间接的团队合作。好的可维护性可以从四个方面获得：

- 代码的松耦合，高度模块化，将页面内的元素视为一个个模块，相互独立，尽量避免耦合过高的代码，从 HTML、CSS、JavaScript 三个层面考虑模块化。
- 良好的注释。
- 注意代码的弹性，在性能和弹性的选择上，一般情况下以弹性为优先考虑条件，在保证弹性的基础上，适当优化性能。
- 严格按照规范编写代码。

## 附录 B

# 命名规则

为避免命名冲突，命名规则如下：

- 公共组件因为高度重用，命名从简，不要加前缀。
- 各栏目的相应代码，需加前缀，前缀为工程师姓名拼音的首字母，例如：阿当前缀为“ad\_”，分隔符为下划线“\_”，例如：“ad\_imgList”。
- 模块组件化，组件中的 class 或 id 名采用骆驼命名法和下划线相结合的方式，单词之间的分隔靠大写字母分开，从属关系靠下划线分隔，如下代码所示。

---

```
html:  
<ul class="textList">  
  <li class="textList_firstItem">1) XXXXXXXXXXXXXXXX</li>  
  <li>2) XXXXXXXXXXXXXXXX</li>  
  <li>3) XXXXXXXXXXXXXXXX</li>  
</ul>  
  
CSS:  
.textList{ } V           .text_list{ } X  
.textList_firstItem{ } V   .textListFirstItem{ } X
```

---

- 命名清晰，不怕命名长，怕命名容易冲突，长命名可以保证不会产生冲突，所以 CSS 选择时可以尽量不使用子选择符，也能确保 CSS 优先级权重足够低，方便扩

展时的覆盖操作，如下代码所示。

---

```
.textList_firstItem{ } V  
.textList .firstItem{ } X
```

---

□ 命名要有意义，不要使用没有意义的命名，尽量用英语命名，不要用拼音。

## 附录 C

# 分工安排

分工安排主要包含以下内容：

- 公共组件（包括 common.css 和 common.js）一人维护，各子频道专人负责，每个频道正常情况下由一人负责，要详细写明注释，如果多人合作，维护的人员注意添加注释信息，具体注释细则，参见附录 D。
- 视觉设计师设计完设计图后，先和交互设计师沟通，确定设计可行，然后先将设计图给公共组件维护者，看设计图是否需要提取公共组件，然后再提交给相应频道的前端工程师。如果有公共组件要提取，公共组件维护者需对频道前端工程师说明。
- 如果确定没有公共组件需提取，交互设计师直接和各栏目的前端工程师交流，对照着视觉设计师的设计图进行需求说明，前端工程师完成需求。
- 前端工程师在制作页面时，需先去 common 文件中查询是否已经存在设计图中的组件，如果有，直接调用；如果没有，则在 app.css 和 app.JavaScript 中添加相应的代码（app 指各频道自己的文件）。
- 前端工程师在制作过程中，发现有高度重用的模块，却未被加入到公共组件中，需向公共组件维护人进行说明，然后工作组件维护人决定是否添加该组件。如果

确定添加，则向前端工程师们说明添加了新组件，让前端工程师们检查之前是否添加了类似组件，统一更新成新组件的用法，删除之前自定义的 CSS 和 JavaScript。虽然麻烦，但始终把可维护性放在首位。

- 公共组件维护者的公共组件说明文档，需提供配套的图片和说明文字，方便阅读。

## 附录 D

# 注释规则

注释的主要规则如下：

- 公共组件和各栏目的维护者都需要在文件头部加上注释说明：

---

```
/**  
 * 文件用途说明  
 * 作者姓名  
 * 联系方式  
 * 制作日期  
 **/
```

---

- 大的模块注释方法：

---

```
//=====  
// 代码用途  
//=====
```

---

小的注释：

---

```
//代码说明
```

---

注释单占一行，不要在代码后的同一行内加注释。

例如：

---

```
//姓名  
var name="abc";           ✓
```

---

```
var name ="abc"; //姓名  ✗
```

---

## 附录 E

# HTML 规范

HTML 规范包含以下内容：

- DTD 统一用 `<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml/DTD/xhtml1-transitional.dtd">`。
- 统一 UTF-8 编码。

HTML、CSS、JavaScript 缩进统一使用 TAB 键。

- 标签名，属性名全部小写，属性需加引号，单标签需闭合，例如`<hr> V. <hr /> X.`
- HTML 应在保证弹性的基础上尽量减少嵌套层数。

严格区分作为内容的图片和作为背景的图片。作为背景的图片采用 CSS sprite 技术，放在一张大图里。大图的安排也遵从 common+app 的方式，全站采用的图片应告之公用组件维护者，添入 common.gif 中，各栏目的图片，应放入 app.gif 中。CSS sprite 技术的优点是减少了 http 请求数，但使图片面向 CSS 的 background-position 增加了耦合度，也增加了维护成本。如果图片有修改，不要删除已添加的图片，在空白处新增修改后的图片，减小修改的风险。

- 标签语义化，Web developer 去样式可读性良好。
- 方便服务器端程序员套模板，HTML 需为模块添加注释。格式为：“`<!-- ××× 开`

始{-->××××××<!--}something 结束-->”，如下代码所示。

---

```
<!--头部开始{-->
<div class="head">
<div class="header clearfix">
    <h1 class="fr">阿当制作</h1>
    <h2 class="fb f1">阿当测试站点<span class="gray fn">的页面管理模式
</span></h2>
    <input type="button" value="完成并退出" class="m150 f1" />
</div>
</div>
<!--)头部结束-->
```

---

## 附录 F

# CSS 规范

CSS 规范主要包括以下内容：

- CSS Reset 用 YUI 的 CSS Reset。
- CSS 采用 CSSReset+common.css+app.css 的形式。
- app.css 采用分工制，一个前端工程师负责一个栏目，如果多人维护，需要添加注释。
- 为方便组件模块化和提高弹性，正常情况下，为避免外边界冲突，组件不设置外边界，外边界用组合 CSS 的方式实现，如下代码所示。

---

```
html:  
<p>12345</p>  
<ul class="textList">  
<li>1) XXXXXXXXXXXX</li>  
<li>2) XXXXXXXXXXXX</li>  
</ul>  
<p>abcde</p>  
<ul class="textList2">  
<li>1) XXXXXXXXXXXX</li>  
<li>2) XXXXXXXXXXXX</li>  
</ul>  
CSS:  
.textList,.textList2{margin-top:10px;XXXXXX  
.textList2{margin-top:20px;}}
```

X

```
=====

```

```
html:
<p>12345</p>
<ul class="textList marginTop10">
<li>1) XXXXXXXXXXXX</li>
<li>2) XXXXXXXXXXXX</li>
</ul>
<p>abcde</p>
<ul class="textList marginTop20">
<li>1) XXXXXXXXXXXX</li>
<li>2) XXXXXXXXXXXX</li>
</ul>
CSS:
.textList{XXXXXXXXXXXXXX}
.marginTop10{margin-top:10px;}
.marginTop20{margin-top:20px;}
```

V

- 为避免组件的上下外边距重合问题和 IE 的 haslayout 引发的 Bug，各模块除特殊需求，一律采用 marginTop 设置上下外边距，如下代码所示。

```
<p>XXXXXXXXXXXXXX</p>
<p class="marginTop10 marginBottom10">XXXXXXXXXXXXXX</p>
<p>XXXXXXXXXXXXXX</p>
```

X

```
=====

```

```
<p>XXXXXXXXXXXXXX</p>
<p class="marginTop10">XXXXXXXXXXXXXX</p>
<p class="marginTop10">XXXXXXXXXXXXXX</p>
```

V

- 优先对已存在的 common.css 中的类进行组合，减少自定义类的数量。
- CSS 用一行的写法，避免行数太长，不利查找，如下代码所示。

```
.menu{margin:0;float:left;font-weight:bold;}    ✓
.menu{
  margin:0;
  float:left;
  font-weight:bold;
}
```

X

- 正式发布前应进行压缩，压缩后文件的命名应添加 “\_min” 后缀。

## 附录 G

# JavaScript 规范

JavaScript 规范主要包含以下内容：

- 底层 JavaScript 库采用 YUI 2.8.0。
- 统一头部中只载入 YUI load 组件，其他组件都通过 loader 对象加载。
- JavaScript 尽量避免使用全局变量，通过命名空间或匿名函数将变量封装到闭包中。
- 正式发布前应进行压缩，压缩后文件的命名应添加“\_min”后缀。



一本打开的书，  
一扇开启的门。  
通向科学圣殿的阶梯。  
托起一流人才的基石。



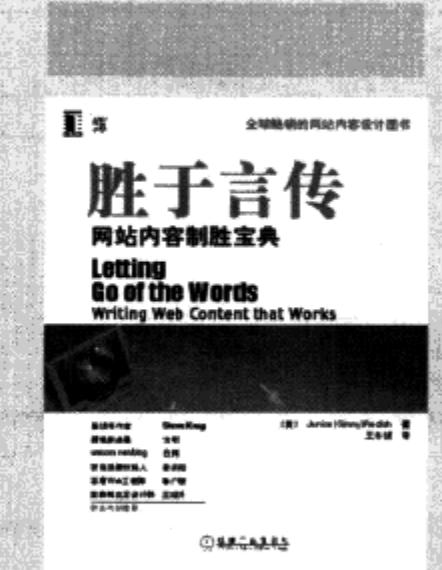
**用户体验的要素**  
作者：Jesse James Garrett  
书号：978-7-111-22310-8  
定价：25.00元



**一目了然：Web软件显性设计之路**  
作者：Robert Hoekman  
书号：978-7-111-22362-7  
定价：39.00元



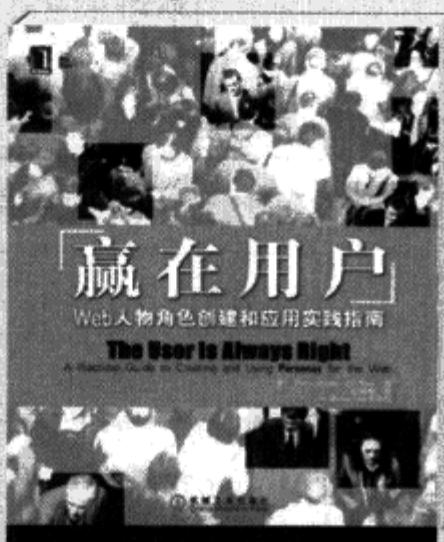
**用户界面设计指南**  
作者：Eric Butow  
书号：978-7-111-22947-6  
定价：36.00元



**胜于言传：网站设计成功宝典**

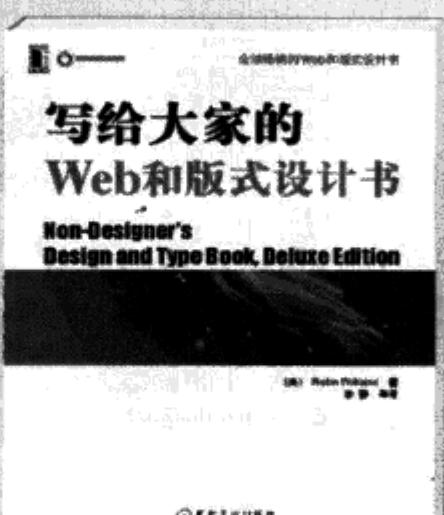


**点石成金：  
访客至上的网页设计秘笈（原书第2版）**  
作者：Steve Krug  
书号：978-7-111-18482-3  
定价：39.00元



**赢在用户：  
Web人物角色创建和应用实践指南**

作者：Steve Mulder; Ziu Yaar  
书号：978-7-111-21888-3  
定价：29.00元



**写给大家的  
Web和版式设计书**  
作者：Robin Williams  
书号：978-7-111-23792-1  
定价：49.00元

# 教师服务登记表

尊敬的老师：

您好！感谢您购买我们出版的 \_\_\_\_\_ 教材。

机械工业出版社华章公司本着为服务高等教育的出版原则，为进一步加强与高校教师的联系与沟通，更好地为高校教师服务，特制此表，请您填妥后发回给我们，我们将定期向您寄送华章公司最新的图书出版信息。为您的教材、论著或译著的出版提供可能的帮助。欢迎您对我们的教材和服务提出宝贵的意见，感谢您的大力支持与帮助！

## 个人资料（请用正楷完整填写）

教师姓名		<input type="checkbox"/> 先生 <input type="checkbox"/> 女士	出生年月		职务		职称： <input type="checkbox"/> 教授 <input type="checkbox"/> 副教授 <input type="checkbox"/> 讲师 <input type="checkbox"/> 助教 <input type="checkbox"/> 其他
学校				学院			系别
联系电话	办公： 宅电： 移动：			联系地址及邮编			
				E-mail			
学历		毕业院校		国外进修及讲学经历			
研究领域							
主讲课程			现用教材名		作者及出版社	共同授课教师	教材满意度
课程：  <input type="checkbox"/> 专 <input type="checkbox"/> 本 <input type="checkbox"/> 研 人数： 学期： <input type="checkbox"/> 春 <input type="checkbox"/> 秋							<input type="checkbox"/> 满意 <input type="checkbox"/> 一般 <input type="checkbox"/> 不满意 <input type="checkbox"/> 希望更换
课程：  <input type="checkbox"/> 专 <input type="checkbox"/> 本 <input type="checkbox"/> 研 人数： 学期： <input type="checkbox"/> 春 <input type="checkbox"/> 秋							<input type="checkbox"/> 满意 <input type="checkbox"/> 一般 <input type="checkbox"/> 不满意 <input type="checkbox"/> 希望更换
样书申请							
已出版著作			已出版译作				
是否愿意从事翻译/著作工作 <input type="checkbox"/> 是 <input type="checkbox"/> 否 方向							
意见和建议							

填妥后请选择以下任何一种方式将此表返回：（如方便请赐名片）

地 址：北京市西城区百万庄南街1号 华章公司营销中心 邮编：100037

电 话：(010) 68353079 88378995 传 真：(010) 68995260

E-mail:hzedu@hzbook.com marketing@hzbook.com 图书详情可登录<http://www.hzbook.com>网站查询