CSS: The Definitive Guide

第三版

CSS

权威指南

O'REILLY®
中国电力出版社

Eric A. Meyer 著
尹志忠 侯妍 译

Web Design

O'REILLY®

CSS 权威指南

你是否既想获得丰富复杂的网页样式，同时又想节省时间和精力？本书为你展示了如何遵循CSS最新规范（CSS2和CSS2.1）将层叠样式表的方方面面应用于实践。通过本书提供的诸多示例，你将了解如何做到仅在一处建立样式表就能创建或修改整个网站的外观，以及如何得到HTML力不能及的更丰富的表现效果。

资深CSS专家Eric A. Meyer利用他独有的睿智和丰富的经验对属性、标记、标记属性和实现做了深入的研究，另外在浏览器支持和设计原则等实际问题上也有独到的见解。你所需要的就是HTML 4.0的知识即可以为网站布局和分页创建简明而且易于维护的脚本，同时兼具桌面系统的美观性和可控性。在本书中你将学到以下内容：

- 用多种方式对文本应用样式
- 用户界面、表布局、列表和生成内容
- 浮动和定位的优缺点
- 字体系列和后路
- 框模型的工作原理
- IE7、Firefox和其他浏览器支持的新CSS3选择器

最新版《CSS权威指南》一书经过全面更新，涵盖了Internet Explorer 7，详细介绍了各个CSS属性以及属性之间的相互作用，并指导你如何避免一些常见的错误。不论你是一位经验丰富的Web创作人员，还是一无所知的新手，都可以把它作为内容翔实的CSS参考资料放在手边。

Eric A. Meyer 在HTML、CSS和Web标准领域是国际上公认的专家，他从1993年就开始从事Web方面的工作。他也是Complex Spiral Consulting公司的奠基人，其客户包括美国在线、苹果计算机公司、富国银行和Macromedia等著名公司。

ISBN 978-7-5083-5594-8

9 787508 355948 >

定价：58.00元

O'Reilly Media, Inc. 授权中国电力出版社出版

第三版

CSS 权威指南

Eric A. Meyer 著
尹志忠 侯妍 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社

图书在版编目（CIP）数据

CSS 权威指南：第 3 版 /（美）迈耶（Meyer, E. A.）著；尹志忠，侯妍译．－北京：中国电力出版社，2007

书名原文：CSS: The Definitive Guide, Third Edition

ISBN 978-7-5083-5594-8

I. C... II. ①迈 ... ②尹 ... ③侯 ... III. 主页制作－软件工具 IV. TP393.092

中国版本图书馆 CIP 数据核字（2007）第 076349 号

北京市版权局著作权合同登记

图字：01-2007-2734 号

书　　名 / CSS 权威指南（第三版）

书　　号 / ISBN 978-7-5083-5594-8

责任编辑 / 牛贵华

封面设计 / Karen Montgomery，张健

出版发行 / 中国电力出版社（www.infopower.com.cn）

地　　址 / 北京三里河路 6 号（邮政编码 100044）

经　　销 / 全国新华书店

印　　刷 / 北京丰源印刷厂

开　　本 / 787 毫米 ×980 毫米　16 开本　32.5 印张　642 千字

版　　次 / 2007 年 10 月第一版　2007 年 10月第一次印刷

印　　数 / 0001-4000 册

定　　价 / 58.00 元（册）

O'Reilly Media, Inc. 介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc. 授权中国电力出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 Unix、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的 *The Whole Internet User's Guide & Catalog*（被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商 —— 每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体 —— 他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以 O'Reilly Media, Inc. 知道市场上真正需要什么图书。

作者简介

Eric A.Meyer 从1993年底就开始从事Web方面的工作，他在HTML、CSS和Web标准领域是国际上公认的专家。在发表文章和著作的同时，Eric还是Complex Spiral Consulting公司（*www.complexspiral.com*）的奠基人，他的客户包括美国在线、苹果计算机公司、Well Fargo银行和Macromedia等著名公司，特别是Macromedia公司这样描述Eric："在我们将Macromedia Dreamweaver MX 2004转变成这样一个基于CSS的设计的革命性工具的过程中，他起到了举足轻重的作用。"

从1994年初开始，Eric成为Case Western Reserve大学网站的视觉效果设计师和校园Web协调员，在这里他还著有3册广受好评的HTML系列教程，他也是《克利夫兰历史百科全书》和《克利夫兰传记辞典》在线版的项目协调员，这是完全免费在Web上发布的第一部关于城市历史的百科全书。

作为《Eric Meyer on CSS》和《More Eric Meyer on CSS》（New Riders出版）、《Cascading Style Sheets:The Definitive Guide》（O'Reilly出版）和《CSS2.0 Programmer's Reference》（Osborne/McGraw-Hill出版），以及O'Reilly Network、Web Techniques和Web Review上许多文章的作者，Eric还创建了CSS Browser Compatibility Charts，并协调完成了W3C官方CSS Test Suite的创作和建成。他曾在众多大型机构发表演说，包括Los Alamos国家实验室、纽约公共图书馆、Cornell大学和Northern Iowa大学。Eric还在很多会议上做过技术演讲，包括他合作创办的An Event Apart、IW3C2 WWW系列、Web Design World、CMP、SXSW、user Irterface会议系列以及The Other Dreamweaver Conference。

在工作之余，Eric担任着css-discuss邮件列表（*www.css-discuss.org*）的管理员，这是他与Western Civilisation的John Allsopp共同创立的一个相当活跃的邮件列表，目前这个邮件列表由evolt.org提供支持。Eric生活在美国俄亥俄州的克利夫兰，你可能想象不到他是一个多优秀的文明市民。他作为"Your Father's Oldsmobile"节目的主持人已经有9年之久，这是克利夫兰WRUW 91.1 FM电台每周一次的一个怀旧爵士乐节目。

可以从Eric个人网页（*http://www.meyerweb.com/eric*）了解他的更多详细信息。

封面介绍

本书封面上的动物是鲑鱼，这一鱼种有很多不同门类。其中最著名的两种是太平洋鲑鱼和大西洋鲑鱼。

太平洋鲑鱼生活在北美洲和亚洲沿岸的北太平洋。太平洋鲑鱼又细分为5个亚种，平均重量在10~30磅之间。太平洋鲑鱼于秋天出生在淡水河的砂砾河床上，经过冬天的孵化长到一英寸长。它们在溪流或湖水中生活一到两年，然后顺流而下游向海洋。在那里生活几年后，再逆流而上游回出生的地方产卵，然后死去。

大西洋鲑鱼生活在北美洲和欧洲沿岸的北大西洋，有很多亚种，包括鳟鲑鱼和红点鲑。大西洋鲑鱼的平均重量在10~20磅之间。大西洋鲑鱼的生命周期与太平洋鲑鱼很类似，也是从淡水河河床游向海洋。不过，它们之间有一个显著的不同，大西洋鲑鱼产卵后不会死去，它们还会回到海洋，然后再返回到淡水河中产卵，通常会这样来回两三次。

总地来讲，鲑鱼是一种漂亮的鱼，体呈银色，背和鳍上有斑点。它们通常吃浮游生物、昆虫幼虫、小虾和小鱼。它们的味觉很好，人们通常认为鲑鱼就是靠味觉从海洋重新游回到它们原来的出生地，这往往要逆流而上经过很多的障碍。有些鲑鱼一直生活在内陆，一生都在淡水河中。

鲑鱼是生态系统中很重要的一部分，因为它们的尸体能为河床供应肥料。不过，鲑鱼的数量在逐年递减。造成鲑鱼数量下降的原因有很多，包括生活环境的破坏、捕鱼、挡住鲑鱼产卵路线的水坝、酸雨、干旱、洪水和污染。

目录

前言

作为一个网页设计人员或文档创作人员，如果你对丰富多彩的页面样式感兴趣，想了解如何改善网站的可访问性，力求节省时间和精力，那么这本书正是为你而写。在学习本书之前，你只需对HTML 4.0有适当的了解。当然，对HTML了解得越多，你的准备也越充分。不过除此之外则没有太多要求，即使你没有其他知识储备也能轻松阅读本书。

本书是《CSS权威指南》的第3版，涵盖了CSS2和CSS2.1（直至2006年4月11日发布的工作草案），其中后者在很多方面都是对前者的澄清。在写这本书时，有些CSS3模块已经达到“候选推荐（Candidate Recommendation）”状态，尽管如此，这一版还是选择不涉及这些模块（只是谈到了部分CSS3选择器）。之所以这么做，原因是这些模块的实现还不完备，或者还不存在相应的模块实现。我认为，现在重要的是关注当前已经得到支持而且得到充分理解的CSS，至于CSS将来提供的功能则留待以后的版本再做介绍。

本书约定

本书使用了以下排版约定：

斜体（*Italic*）

指示新术语、URL、正文中的变量、用户定义的文件和目录、命令、文件扩展、文件名、目录或文件夹名，以及UNC路径名。

等宽字体（`Constant width`）

指示命令行计算机输出、代码示例、注册表键，以及键盘快捷键。

等宽粗体（**`Constant width bold`**）

指示示例中的用户输入。

等宽斜体（*Constant width italic*）

指示示例和注册表键中的变量。斜体文本中的变量或用户自定义元素（如路径名或文件名）也用等宽斜体表示。例如，在路径*\Windows\username*中，要把username替换成你自己的名字。

属性约定

在本书中会有一些方框来描述给定CSS属性。这些是从CSS规范如实照搬来的，不过本书中还会提供一些语法解释。

总地说来，每个属性的可取值采用以下语法列出：

Value: [<length> | thick | thin]{1,4}
Value: [<family-name> ,]* <family-name>
Value: <url>? <color> [/ <color>]?
Value: <url> || <color>

“<”和“>”之间的词给出了某种值的类型，或者是对另一个属性的引用。例如，属性font将接受具体属于font-family属性的值，由文本<font-family>标示。用等宽字体显示的词都是必须原样显示的关键字，不要再加引号。斜线（/）和逗号（,）也必须原样使用。

一些关键字串在一起，这意味着所有这些关键字必须以给定顺序出现。例如，help me表示这个属性必须准确地以此顺序使用这些关键字（即先help后me）。

如果候选项由一个竖线分隔（X | Y），那么必须出现其中之一。双竖线（X || Y）表示出现X或Y，或者二者都必须出现（但是必须以先X后Y的顺序出现）。中括号（[...]）用于分组。两项并列优先级要高于双竖线，而双竖线则高于单竖线。所以，“V W | X || Y Z”等价于“[V W] | [X || [Y Z]]”。

每个单词或加中括号的分组后面可以跟有以下修饰符之一：

- 星号（*）指示前面的值或分组重复0或多次。因此，bucket*就表示bucket一词可以使用任意多次（包括0次）。使用次数没有上限。
- 加号（+）表示前面的值或分组可以重复1或多次。因此，mop+表示mop一词必须至少使用1次，可能还可以使用多次。
- 问号（?）表示前面的值或分组是可选的。例如，[pine tree]?表示不一定使用pine tree分组（如果使用了这个分组，就必须以指定的顺序出现）。

- 大括号里的一对数（{M,N}）表示前面的值或分组至少重复M次，最多N次。例如，ha{1,3}表示词ha可以出现1次、2次或3次。

以下是一些例子：

give || me || liberty

至少使用这三个词中的一个，而且可以以任何顺序使用。例如，give liberty、give me、liberty me give和give me liberty都是合法的。

[I | am]? the || walrus

可以使用单词I或am，但不能二者都使用，而且是否使用其中之一也是可选的。此外，必须跟有the或walrus，或者二者都有，其顺序不限。因此，可以构造出I the walrus、am walrus the、am the、I walrus、walrus the等等。

koo+ ka-choo

koo可以有一个或多个实例（即出现1次或多次），其后必须跟有ka-choo。因此，koo koo ka-choo、koo koo koo ka-choo和koo ka-choo都是合法的。koo的个数可以是无限的，不过这要受实现的特定限制所限。

I really{1,4}? [love | hate] [Microsoft | Netscape | Opera | Safari]

这是Web设计人员通用的一个选项表达。这个示例可以解释为I love Netscape、I really love Microsoft以及类似的表达式。really可以使用0～4次。尽管这个示例中只出现了love，不过除了选择love，还可以选择hate。

[[Alpha || Baker || Cray],]{2,3} and Delphi

这是一个又长又复杂的表达式。以下是一种可能的结果：Alpha, Cray, and Delphi。这里有逗号，因为逗号放在嵌套分组中。

代码示例的使用

本书用于帮助你更好地完成工作。一般来说，你可以在自己的程序和文档里使用本书中的代码。除非你复制使用了本书中的大部分代码，否则不需要获得我们的许可。例如，如果你写了一个程序，其中使用了本书中的几个代码段，这是不需要许可的。不过销售或发行O'Reilly图书的示例光盘则需要得到许可。如果引用本书和利用书中的示例代码回答一个问题，并不需要获得特别的许可。但是如果在你的产品文档中大量使用本书中的示例代码，就需要许可了。

我们希望大家使用代码时能注明出处，但并不强求。相关信息通常包括书名、作者、出版商和ISBN。比如："*CSS: The Definitive Guide, Third Edition*, by Eric A. Meyer. Copyright 2007 O'Reilly Media, Inc., 978-0-596-52733-4"。

如果你感觉对代码示例的使用超出了合理的使用范围或上述的许可范围，请尽管联系我们：*permissions@oreilly.com*。

建议与评论

本书的内容都经过测试，尽管我们做了最大的努力，但疏漏之处在所难免。如果读者发现有什么错误，或者是对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室
奥莱理软件（北京）有限公司

本书的网页上列出了勘误表、示例和任何额外的信息。可登录以下网址查询：

http://www.oreilly.com/catalog/csstdg3/
http://www.oreilly.com/book.php?bn=978-7-5083-5594-8

如果想要发表关于本书的评论和技术问题，请发邮件至：

bookquestions@oreilly.com
info@mail.oreilly.com.cn

关于图书、会议、资源中心和 O'Reilly 网络的更多信息，请查看我们的站点：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

在这里我要用一些篇幅来感谢那些一直站在我身后的人，这本书能奉献给读者，离不开他们长期以来的支持和帮助。

首先，我要感谢O'Reilly的每一个人，感谢他们多年来的工作，让我有幸参与图书出版，而且还给了我这样一个机会可以出版自己的书。在第三版中，我要特别感谢 Tatiana Apandi，感谢她的幽默、耐心和对我的理解，特别是当最后期限临近我还在逗弄小鸡时，她也能泰然处之。

还要向我的技术审校人员致以最真挚的谢意。第一版的技术审校是David Baron和Ian Hickson，另外Bert Bos和Håkon Lie做了一些补充。第二版的技术审校是Tantek Çelik和Ian Hickson。而第三版则经过了Darrell Austin、Liza Daly和Neil Lee的精心审校。所有这些人都无私地贡献了他们的丰富经验和远见卓识，使我能准确而及时地跟上CSS最新的变化，并且避免了罗嗦的描述和含混的解释。所有这三版（至少是这一版）如果没有他们共同的努力就不会像现在这么好，不过当然如果你在书中发现有错误，那绝对是我的错，与他们无关。这是不是有些陈辞滥调？不错，但这确实是我的肺腑之言。

同样地，我还要感谢指出勘误的所有人。我可能没有立即回复你们的邮件，不过我确实读了你们提出的所有问题，而且做了必要的更正。希望大家一如继往地反馈，提出建设性的批评意见，就像从前一样，这只会让本书变得更好。

还有一些人需要特别感谢。

感谢WRUW, 91.1 FM Cleveland的员工，感谢你们9年来的支持、绝妙的音乐，还有那种率直的风趣。也许哪一天我会把“第一频道”调回到你们的频道，也许不会；但无论怎样，都希望你们继续努力。

感谢Jeffrey Zeldman，你一直是一个很棒的同事和伙伴；还要感谢整个Zeldman家庭，你们真是无与伦比的朋友。

感谢Molly“阿姨”，感谢你一直那么实在。

感谢Jim“叔叔”，不论从公从私，总之感谢你的一切。如果没有你的影响，我肯定不是现在这样，没有你在身边，我们可能要穷多了，这么说绝对不为过。

感谢“Bread and Soup”全体成员——Jim、Genevieve、Jim、Gini、Ferrett、Jen、Jenn和Molly——感谢你们的精湛厨艺和有关美食的探讨。

感谢我的所有家人，谢谢你们一直以来的支持和爱。

感谢我应当感谢但在上面未能列出的每一位，这既是道歉，也是致谢。

还要感谢我的妻子和女儿，对你们的感谢无以言表，是你们让我的生活如此丰富，这是我以前不敢想象的；你们给了我那么多爱，我都不知道该如何回报。当然，我会一直努力的。

——Eric A. Meyer

Cleveland Heights, Ohio

1 August 2006

第1章

CSS和文档

层叠样式表（Cascading Style Sheets，CSS）的功能非常强大，可以影响一个或一组文档的表现。显然，如果不存在某种文档，CSS基本上毫无用处，因为这样一来它将没有要表现的内容。当然，“文档”的定义相当宽泛。例如，Mozilla和相关的浏览器就使用CSS来影响浏览器Chrome本身的表现。不过，如果没有Chrome的内容——按钮、地址栏输入、对话框、窗口等等——也就没有使用CSS（或其他任何表现信息）的必要。

Web的衰落

也许你还能大致记得，在Web早期（1990～1993），HTML是一个很有限的语言。它几乎完全由用于描述段落、超链接、列表和标题的结构化元素组成。我们可能认为表、框架或复杂标记等等内容是创建Web页面必不可少的，可是那时在HTML中连与之稍有些相似的东西都没有。HTML原本是要作为一种结构化标记语言，用于描述文档的各个部分；而对于这些部分应当如何显示则很少谈及。这种语言并不关心外观，它只是一种简洁的小型标记机制。

接下来Mosaic出现了。

仿佛突然之间，在网上流连时间超过10分钟的人几乎都认识到了万维网的强大。从一个文档跳到另一个文档很容易，无非是把光标指向有特殊颜色的部分文本，甚至指向一个图像，再点击鼠标。更妙的是，文本和图像可以同时显示，只需要一个纯文本编辑器就能创建页面。这是免费而且开放的，确实很酷。

网站开始到处涌现。期刊网站、大学网站、公司网站等等应运而生。随着网站数目的增

加，人们越来越需要新的HTML元素，希望这些元素各自完成一个特定的功能。创作人员开始要求能够将文本变为粗体或斜体。

而此时HTML却不足以处理这样一些需求。利用HTML可以声明强调部分文本，但不一定是将其置为斜体，这取决于用户的浏览器和首选项，可能只是改为粗体，或者仍然是正常文本，只不过有不同的颜色而已。这就无法保证读者看到的正是创作人员所创建的文档。

迫于这些压力，开始出现诸如<FONT>和<BIG>之类的标记元素。突然之间，原来描述结构的语言开始描述外在表现了。

一片混乱

几年之后，这种随便的做法所存在的问题开始暴露出来。例如，HTML 3.2和HTML 4.0的很大一部分都是关于表现问题。此时能够通过font元素对文本设置颜色和大小，对文档和表格应用背景色和图像，使用table属性（如cellspacing），并且还能让文本闪烁，这些都是原先要求“有更多控制”的后果。

下面举例说明这种混乱的具体情况，简单地看一下几乎所有公司网站都用到的标记。这些网站中往往标记相当多，而真正有用的信息并不多，二者的差距大得惊人。更糟糕的是，在大多数网站中，标记几乎完全由表和font元素组成，它们对于所要表现的内容不能传达任何实际含义。从结构化的角度看，这些网页比随机的字母串强不了多少。

例如，来看页面标题，如果创作人员使用了font元素而不是h1之类的标题元素：

```
<font size="+3" face="Helvetica" color="red">Page Title</font>
```

从结构上来讲，font标记没有任何含义。这会使文档的可用性降低。例如，对于一个语音合成浏览器，font标记有什么意义呢？不过，如果创作人员使用标题元素而不是font元素，语音浏览器就可以使用某种语音样式来读相关文本。倘若使用font标记，这种语音浏览器就无法知道这个文本与其他文本有什么区别。

为什么创作人员这么不看重结构和含义呢？因为他们希望读者看到的页面正如他们设计的那样。使用结构化HTML标记意味着要放弃对页面外观的很多控制，而结构化HTML标记显然不支持多年来已经深入人心的那些流行的页面设计。不过还需要考虑上述方法存在的如下一些问题：

- 非结构化页面使得建立内容索引极为困难。真正强大的搜索引擎允许用户只搜索页面标题，或者搜索页面内的小节标题，或者只搜索段落文本，也可能只搜索那些标记为重要的段落。不过，要完成这样一个任务，页面内容必须包含在某种结构化标

记中，而这正是大多数页面所缺少的。例如，Google 在索引页面时就会注意标记结构，所以如果你的页面是一个结构化页面，被 Google 搜中的机会就会增加。

- 缺乏结构性会降低可访问性。假设你是一个盲人，要依赖一个语音合成浏览器上网搜索。下面的两种页面你会选择哪一个呢？是一个结构化页面，使得你的浏览器可以只读出小节标题，让你选择想听哪一小节；还是一个无结构性的页面，浏览器必须读出所有内容，因为没有提示来指出哪些是标题、哪些是段落、哪些是重要的内容。再来看 Google，实际上这个搜索引擎就是世界上最活跃的盲人用户，有数百万的朋友在接受它的建议，了解在哪里网上冲浪和购物。
- 高级页面表现只能应用于某种文档结构。假设有这样一个页面，其中只显示了小节标题，各标题旁分别有一个箭头。用户可以决定哪些小节标题对他来说需要深入了解，点击相应的箭头就能显示出这一节的文本。
- 结构化标记更易于维护。你可能曾经花很长时间在别人（甚至你自己）的HTML中查找一个小错误，由于这个错误，让你的页面在某个浏览器中显示得一片混乱，这种情况是不是屡屡出现？你是不是曾经花很长时间来编写嵌套表和font元素，而这只是为了得到一个包含白色超链接的边栏？为了正确地分隔一个标题和它后面的文本，你插入过多少换行元素？通过使用结构化标记，就能清理你的代码，更容易地找到所要寻找的东西。

必须承认，完全结构化的文档有些太古板、太平常了。“一白遮百丑”，就因为这么一个原因，尽管有几百个理由要求使用结构化标记，但仍然不能阻挡HTML的使用，直到20世纪末它还如此流行，甚至到今天它还依然盛行。我们需要一种合适的方法，将结构化标记与丰富多彩的页面表现结合起来。

CSS 作救星

当然，HTML中充斥着的表现标记的问题并没有被W3C（World Wide Web Consortiam，万维网联盟）忽视，他们开始寻找一种速效的解决方法。1995 年，W3C 开始发布一种正在进行的计划（work-in-progress），称为CSS。到了1996年，这已经成为一个成熟的推荐草案（Recommendation），其地位与HTML同样举足轻重。下面来说明这是为什么。

丰富的样式

首先，与 HTML 相比，CSS 支持更丰富的文档外观，其表现程度也远非 HTML 力所能及。CSS 可以为任何元素的文本和背景设置颜色；允许在任何元素外围创建边框，同时能增大或减少元素外的空间；允许改变文本的大小写、装饰方式（如下划线）、间隔，甚至可以确定是否显示文本；还允许完成许多其他的效果。

以页面上的第一个标题（即主标题）为例，这通常就是页面本身的标题。以下是一个正确的标记：

```
<h1>Leaping Above The Water</h1>
```

现在，假设你希望这个标题是暗红色，使用某种字体，采用斜体，而且有下划线，还有一个黄色的背景。如果用HTML来达到上述目的，就必须把h1放在表中，而且还要有数十个其他的元素，如font和U。如果使用CSS，所需的则只是简单的一条规则：

```
h1 {color: maroon; font: italic 2em Times, serif; text-decoration: underline;
    background: yellow;}
```

如此而已。可以看到，用HTML能够做到的，用CSS也能做到。不过，还不仅限于此：

```
h1 {color: maroon; font: italic 2em Times, serif; text-decoration: underline;
    background: yellow url(titlebg.png) repeat-x;
    border: 1px solid red; margin-bottom: 0; padding: 5px;}
```

现在h1的背景上有一个只能水平重复的图像，而且它有一个边框，与文本之间至少间隔5个像素。还去除了元素底端的外边距（空白）。这些工作是HTML做不到的，甚至连类似的事情都办不到，而这还只是CSS的冰山一角。

易于使用

如果以上所述还不能说服你，下面的理由可能会让你改变想法：样式表能大大减少Web创作人员的工作量。

首先，样式表将实现某些视觉效果的命令集中在一个方便的位置，而不是在文档中分散得到处都是。举例来说，假设你希望一个文档中的所有h2标题都是紫色。若使用HTML，则要在每个标题中增加一个font标记，如下所示：

```
<h2><font color="purple">This is purple!</font></h2>
```

所有二级标题都要增加这个标记。如果文档中有40个这样的标题，就必须总共插入40个font元素，每个标题插入一个font！为了达到这样小的一个效果，就要做这么多的工作。

假设你早有打算，已经插入了所有这些font元素。大功告成，你很满意——不过，可能接下来你认为这些h2标题实际上应该是暗绿色而不是紫色（或者你的老板决定让你这么做）。这样一来，就必须再回过头来逐个地调整这些font元素。当然，只要你的文档中只是标题有紫色文本，就可以利用“查找－替换”来完成调整。但是如果文档中还有其他元素也有紫色font，就不能使用“查找－替换”，因为这将影响那些元素（将把那些元素也改成暗绿色）。

更好的办法是使用一条规则：

```
h2 {color: purple;}
```

这样做不仅输入起来更快，修改起来也更容易。如果确实要从紫色改为暗绿色，所要做的只是修改这一条规则。

再来看上一节谈到的有丰富样式的h1元素：

```
h1 {color: maroon; font: italic 2em Times, serif; text-decoration: underline;
    background: yellow;}
```

这看上去比写HTML还要糟糕，不过请考虑这样的情况：一个页面上有12个看上去和h1一样的h2元素。这12个h2元素需要多少标记呢？如果使用HTML，就需要非常多的标记。另一方面，如果用CSS，所要做的只是：

```
h1, h2 {color: maroon; font: italic 2em Times, serif; text-decoration: underline;
        background: yellow;}
```

现在这些样式会同时应用到h1和h2元素，这里只多敲了3次键而已。

如果你想改变h1和h2元素的外观，CSS的优势则更为突出。考虑一下，与对前面的样式作如下修改相比，需要花多少时间才能修改h1和12个h2元素的HTML标记：

```
h1, h2 {color: navy; font: bold 2em Helvetica, sans-serif;
        text-decoration: underline overline; background: silver;}
```

如果用秒表来记录上述两种方法所花的时间，我打赌使用CSS的创作人员肯定会让使用HTML的人输得哑口无言。

此外，大多数CSS规则都集中在文档中的某一个位置。也可以将其分组为相关的样式或单个元素分散到文档中，但是把所有样式都放在一个样式表中的做法往往高效得多。这样，在一个位置上就能创建（或修改）整个文档的外观。

在多个页面上使用样式

不过请等等，还不只如此！不仅能把一个页面的所有样式信息集中到一个位置，还可以创建一个样式表，然后把这个样式表应用到多个页面。这是通过以下过程实现的：将样式表单独保存为一个文档，然后由要使用该文档的页面导入。通过使用这个功能，可以很快地为整个网站创建一致的外观。为此只需将这个样式表链接到网站上的所有文档。在此之后，如果你想改变网站页面的外观，只编辑一个文件就够了，所做的修改便会自动地传播到整个服务器！

考虑这样一个网站，它的所有标题都是灰色，背景为白色。这种颜色设置是由一个样式表得到的，其中指出：

```
h1, h2, h3, h4, h5, h6 {color: gray; background: white;}
```

现在假设这个网站有700个页面，每个页面都使用了这个要求标题为灰色的样式表。在某个时刻，网站管理员决定标题应当是白色，而背景才是灰色。所以将样式表编辑如下：

```
h1, h2, h3, h4, h5, h6 {color: white; background: gray;}
```

然后将样式表保存到磁盘，修改就完成了。如果使用HTML，则要编辑700个页面，每个标题都要包含在一个表中，还要有一个font标记，这与使用CSS的方法绝对无法相比，不是吗？

层叠

还有呢！CSS还规定了冲突规则；这些规则统称为层叠（cascade）。例如，还是考虑前面的情况，将一个样式表导入到多个Web页面。现在增加一组页面，其中有些样式是相同的，不过还包含一些专用于这些页面的特定规则。可以另外创建一个样式表，将这个样式表与先前的样式表一起导入到这些页面中，或者可以在需要特殊样式的页面中直接放入那些样式。

例如，在700个原有页面之外的某个页面上，可能希望标题是黄色，背景是深蓝色，而不是灰色背景上的白色标题。在该文档中，可以插入以下规则：

```
h1, h2, h3, h4, h5, h6 {color: yellow; background: blue;}
```

由于层叠，这个规则会覆盖先前导入的灰底白字标题规则。如果理解了层叠规则，并充分利用了这些规则，就能创建相当复杂的样式表，这样不仅很容易地修改，还会使你的页面看上去很专业。

层叠并不仅限于创作人员使用。在网上冲浪的人（或读者）还可以利用某些浏览器创建自己的样式表（这称为读者样式表，原因显而易见），这些样式表可以与创作人员创建的样式表以及浏览器使用的样式层叠。因此，如果读者是一个色盲，就可以创建这样一个样式，使超链接突出显示：

```
a:link, a:visited {color: white; background: black;}
```

读者样式表可以包含几乎所有内容：可以是一个指令，如果用户视力不好，这个指令可以让文字足够大以便用户阅读；也可以是一些规则，能够删除图像以便更快地阅读和浏览；甚至可以是一些样式，将用户最喜欢的图片放在每个文档的背景上（当然，不推荐

这种做法，不过这种做法确实是可能的）。这样就能让读者自己定制网上体验，而且不必去掉创作人员的所有样式。

由于提供了导入、层叠和丰富的效果，CSS对所有创作人员或读者来说都是一个绝佳的工具。

缩减文件大小

除了视觉上的功能以及对创作人员和读者都很有用之外，CSS还有一些很让读者喜欢的特性。它有助于尽可能地缩减文档大小，以便加快下载。这是怎么做到的呢？前面已经提到，很多页面都使用了表和font元素来得到漂亮的视觉效果。遗憾的是，这些方法都会创建额外的HTML标记，以至于增加文件大小。通过将视觉样式信息分组放到集中的区域中，并使用一种相当简洁的语法表示这些规则，就可以去除font元素和其他一些常用的标记。因此，CSS一方面可以大大减少下载时间，另一方面又能大大提升读者的满意度。

为将来做准备

前面提到过，HTML是一种结构化语言，而CSS是它的补充：这是一种样式语言。认识到这一点，W3C（也就是讨论并批准Web标准的组织）开始从HTML去除样式元素。之所以这么做，是因为可以用样式表提供某些HTML元素目前提供的效果，既然如此，还有谁需要使用这些HTML元素呢？

因此，XHTML规范中有很多已经不鼓励使用的元素，也就是说，这些元素正逐步从语言中消失。最终，这些元素会被标志为废弃，这说明不要求也不鼓励浏览器支持这些元素。不鼓励使用的元素包括 <font>、<basefont>、<u>、<strike>、<s>和 。样式表的出现使得已经不再需要这些元素。随着时间的推移，还会有更多元素步入这个不鼓励使用的行列。

不仅如此，HTML还有可能逐步被可扩展标记语言（Extensible Markup Language，XML）所取代。XML是一种比HTML更复杂的语言，不过它也更强大、更灵活。尽管如此，XML没有任何提供声明样式元素（如<i>或 ）的方法。相反，XML文档很可能要依赖于样式表来确定其外观。尽管XML使用的样式表可以不是CSS，不过它很可能遵循CSS而且与之非常相似。因此，现在学习CSS对创作人员是很有好处的，等到HTML被XML取代时这种好处就会更显著。

所以，重要的是首先要理解CSS和文档结构之间有何关联。使用CSS可能对文档表现有深远的影响，但是到底能做些什么，对此还是有一些限制。先来学习一些基本术语。

元素

元素(element)是文档结构的基础。在HTML中，最常用的元素很容易识别，如p、table、span、a和div。文档中的每个元素都对文档的表现起一定作用。在CSS中，至少在CSS2.1中，这意味着每个元素生成一个框（box，也称为盒），其中包含元素的内容。

替换和非替换元素

尽管CSS依赖于元素，但并非所有元素都以同样的方式创建。例如，图像和段落就不是同类元素，span和div也不相同。在CSS中，元素通常有两种形式：替换和非替换。这两种类型将在第7章详细讨论，其中将介绍框模型（也称盒模型）的具体内容，但是这里还是先做一个简要介绍。

替换元素

替换元素（replaced element）是指用来替换元素内容的部分并非由文档内容直接表示。在XHTML中，我们最熟悉的替换元素例子就是img元素，它由文档本身之外的一个图像文件来替换。实际上，img没有具体的内容，通过以下的简单例子可以了解这一点：

```
<img src="howdy.gif" />
```

这个标记片段不包含任何具体内容，只有一个元素名和一个属性。除非将其指向一个外部内容（在这里，就是由src属性指定的一个图像），否则这个元素没有任何意义。input元素与之类似，取决于input元素的类型，要由一个单选钮、复选框或文本输入框替换。替换元素显示时也生成框。

非替换元素

大多数HTML和XHTML元素都是非替换元素（nonreplaced element）。这意味着，其内容由用户代理（通常是一个浏览器）在元素本身生成的框中显示。例如，hi there就是一个非替换元素，文本“hi there”将由用户代理显示。段落、标题、表单元格、列表和XHTML中的几乎所有元素都是非替换元素。

元素显示角色

除了替换和非替换元素，CSS2.1还使用另外两种基本元素类型：块级（block-level）元素和行内（inline-level）元素。如果创作人员以前用过HTML或XHTML标记，并了解它们在Web浏览器中的显示，就不会对这些类型感到陌生；这些元素如图1-1所示。

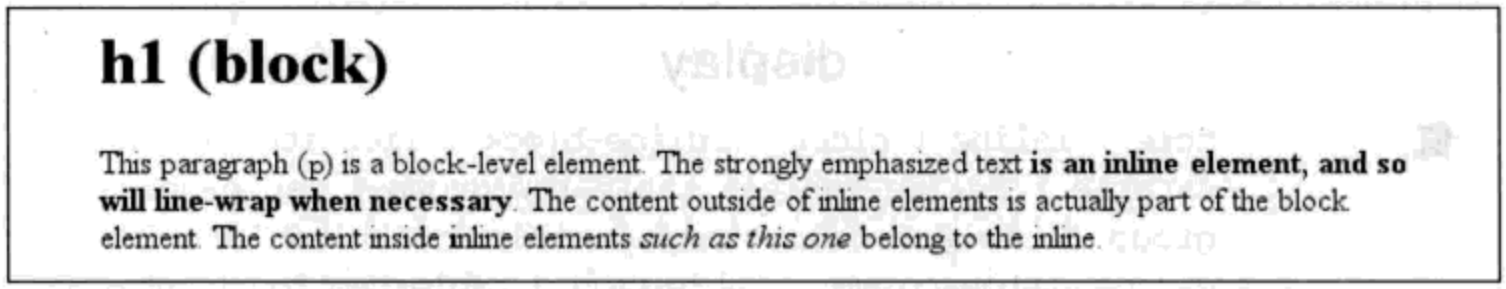

图1-1：XHTML文档中的块级和行内元素

块级元素

块级元素生成一个元素框，（默认地）它会填充其父元素的内容区，旁边不能有其他元素。换句话说，它在元素框之前和之后生成了“分隔符”。我们最熟悉的HTML块元素是p和div。替换元素可以是块级元素，不过通常都不是。

列表项是块级元素的一个特例。除了表现方式与其他块元素一致，列表项还会生成一个标记符——无序列表中这通常是一个圆点，有序列表中则是一个数字——这个标记符会“关联”到元素框。除了这个标记符外，列表项在所有其他方面都与其他块元素相同。

行内元素

行内元素在一个文本行内生成元素框，而不会打断这行文本（译注1）。行内元素最好的例子就是XHTML中的a元素。strong和em也属于行内元素。这些元素不会在它本身之前或之后生成“分隔符”，所以可以出现在另一个元素的内容中，而不会破坏其显示。

注意，尽管“块”和“行内”这两个词与XHTML中的块级和行内元素有很多共同点，但也存在一个重要的差别。在HTML和XHTML中，块级元素不能继承自行内元素（即不能嵌套在行内元素中）。但是在CSS中，对于显示角色如何嵌套不存在任何限制。

要了解这是如何工作的，下面来考虑一个CSS属性：display。

你可能已经注意到，display有很多值，其中只有3个值在前面提到过：block、inline和list-item。我们并不打算现在就讨论其他的值，因为它们将在第2章和第7章中更详细地介绍。

目前，我们只关心block和inline。考虑以下标记：

```
<body>
<p>This is a paragraph with <em>an inline element</em> within it.</p>
</body>
```

译注1：从概念上讲，这里inline的准确含义就是“行内”，而不是一般认为的“内联”。

display	
值	none | inline | block | inline-block | list-item | run-in | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | inherit
初始值	inline
应用于	所有元素
继承性	无
计算值	对于浮动元素、定位元素和根元素可变（参见CSS2.1第9.7节）；否则为指定值

这里有两个块元素（body和p）和一个行内元素（em）。按照XHTML规范，em可以继承p，但是反过来不行。一般地，XHTML层次结构要求：行内元素可以继承块元素，而反之不允许。

与此不同，CSS没有这种限制。仍然是上述标记，不过可以改变两个元素的显示角色，如下：

```
p {display: inline;}
em {display: block;}
```

这会使得元素在一个行内框中生成一个块框。这是完全合法的，不违反任何规范。唯一的问题是，如果试图如下反转元素的嵌套关系：

```
<em><p>This is a paragraph improperly enclosed by an inline element.</p></em>
```

不论通过CSS对显示角色做了什么改变，在XHTML中这都是不合法的。

对于XHTML文档来说，尽管改变元素的显示角色可能很有用，不过对XML文档的意义则更为重大。XML文档不太可能有固有显示角色，所以要由创作人员来定义。例如，你可能想知道如何摆放以下XML片段：

```
<book>
 <maintitle>Cascading Style Sheets: The Definitive Guide</maintitle>
 <subtitle>Second Edition</subtitle>
 <author>Eric A. Meyer</author>
 <publisher>O'Reilly and Associates</publisher>
 <pubdate>2004</pubdate>
 <isbn>blahblahblah</isbn>
</book>
<book>
```

```
 <maintitle>CSS2 Pocket Reference</maintitle>
 <author>Eric A. Meyer</author>
 <publisher>O'Reilly and Associates</publisher>
 <pubdate>2004</pubdate>
 <isbn>blahblahblah</isbn>
</book>
```

由于display的默认值是inline，默认地其内容会显示为行内文本，如图1-2所示。这种显示用处不大。

Cascading Style Sheets: The Definitive Guide Second Edition Eric A. Meyer O'Reilly and Associates 2004 blahblahblah CSS2 Pocket Reference Eric A. Meyer O'Reilly and Associates 2004 blahblahblah

图1-2：XML文档的默认显示

可以用display来定义基本布局：

```
book, maintitle, subtitle, author, isbn {display: block;}
publisher, pubdate {display: inline;}
```

现在将7个元素中的5个设置为块元素，另外两个设置为行内元素。这意味着，每个块元素都会像XHTML中的div元素一样处理，而两个行内元素的处理方式将类似于span。由于CSS具有这种影响显示角色的基本功能，使得CSS在很多情况下都非常有用。可以将以上规则作为起点，再增加另外一些样式，得到图1-3所示的结果。

Cascading Style Sheets: The Definitive Guide
Second Edition
Eric A. Meyer
O'Reilly and Associates (2004)
blahblahblah

CSS2 Pocket Reference
Eric A. Meyer
O'Reilly and Associates (2004)
blahblahblah

图1-3：增加样式后XML文档的显示

在本书余下部分中，我们将讨论各种属性和值来支持诸如此类的表现。不过，首先需要了解如何将CSS与文档关联。毕竟，如果不将二者结合，CSS就没有办法影响文档。我们将通过一个XHTML设置进行讨论，因为这是我们最熟悉的情况。

结合 CSS 和 XHTML

我已经提到过，HTML和XHTML文档有一个固有结构，这里需要重申这一点。事实上，正是这一点导致了以前网页所存在的部分问题：我们之中太多的人已经忘记文档要有一个内部结构，而且这与其视觉结构完全是两码事。我们可能急于创建最酷的页面，可能会以各种方式摆放页面的内容，而通常忽略了一点：页面应当包含有某种结构含义的信息。

这种结构正是XHTML和CSS之间关系中的一个固有部分；如果没有这种结构，就根本不会有任何关系。为了更好地理解这一点，下面来看一个XHTML文档的例子，后面将逐一介绍这个文档中的各个部分：

```
<html>
<head>
<title>Eric's World of Waffles</title>
<link rel="stylesheet" type="text/css" href="sheet1.css" media="all" />
<style type="text/css"> @import url(sheet2.css);
h1 {color: maroon;}
body {background: yellow;}
/* These are my styles! Yay! */
</style>
</head>
<body>
<h1>Waffles!</h1>
<p style="color: gray;">The most wonderful of all breakfast foods is
the waffle--a ridged and cratered slab of home-cooked, fluffy goodness
that makes every child's heart soar with joy. And they're so easy to make!
Just a simple waffle-maker and some batter, and you're ready for a morning
of aromatic ecstasy!
</p>
</body>
</html>
```

这个标记如图 1-4 所示。

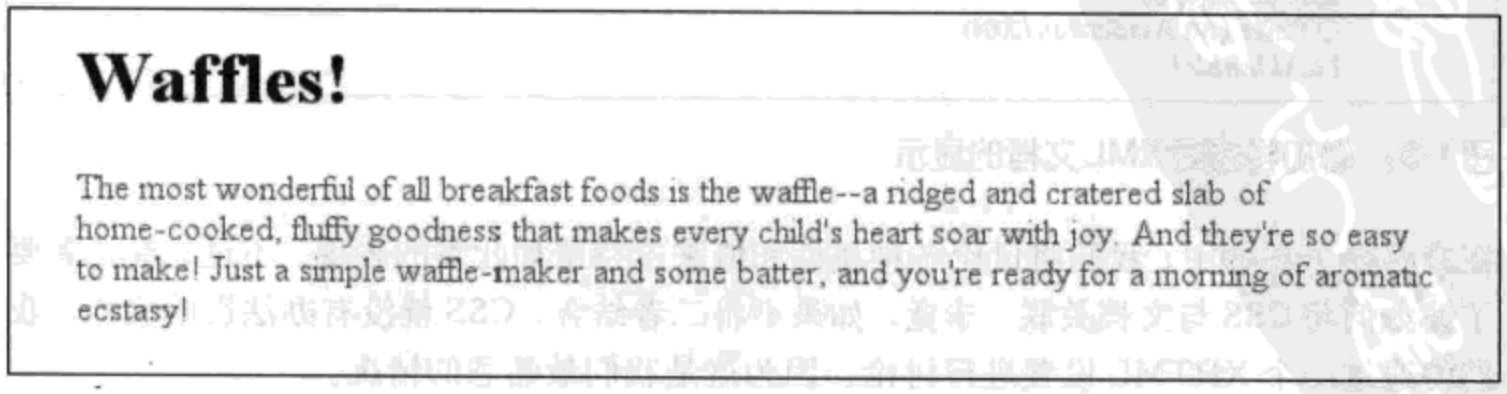

图1-4：一个简单的文档

下面来分析可以将这个文档与CSS关联的多种方式。

link 标记

首先考虑 link 标记的使用：

```
<link rel="stylesheet" type="text/css" href="sheet1.css" media="all" />
```

link 标记是一个很少用但完全合法的标记，它在 HTML 规范中已经存在多年，一直等待着“施展才华”。其基本目的是允许 HTML 创作人员将包含 link 标记的文档与其他文档相关联。CSS 使用这个标记来链接样式表和文档；在图 1-5 中，一个名为 *sheet1.css* 的样式表链接到这个文档。

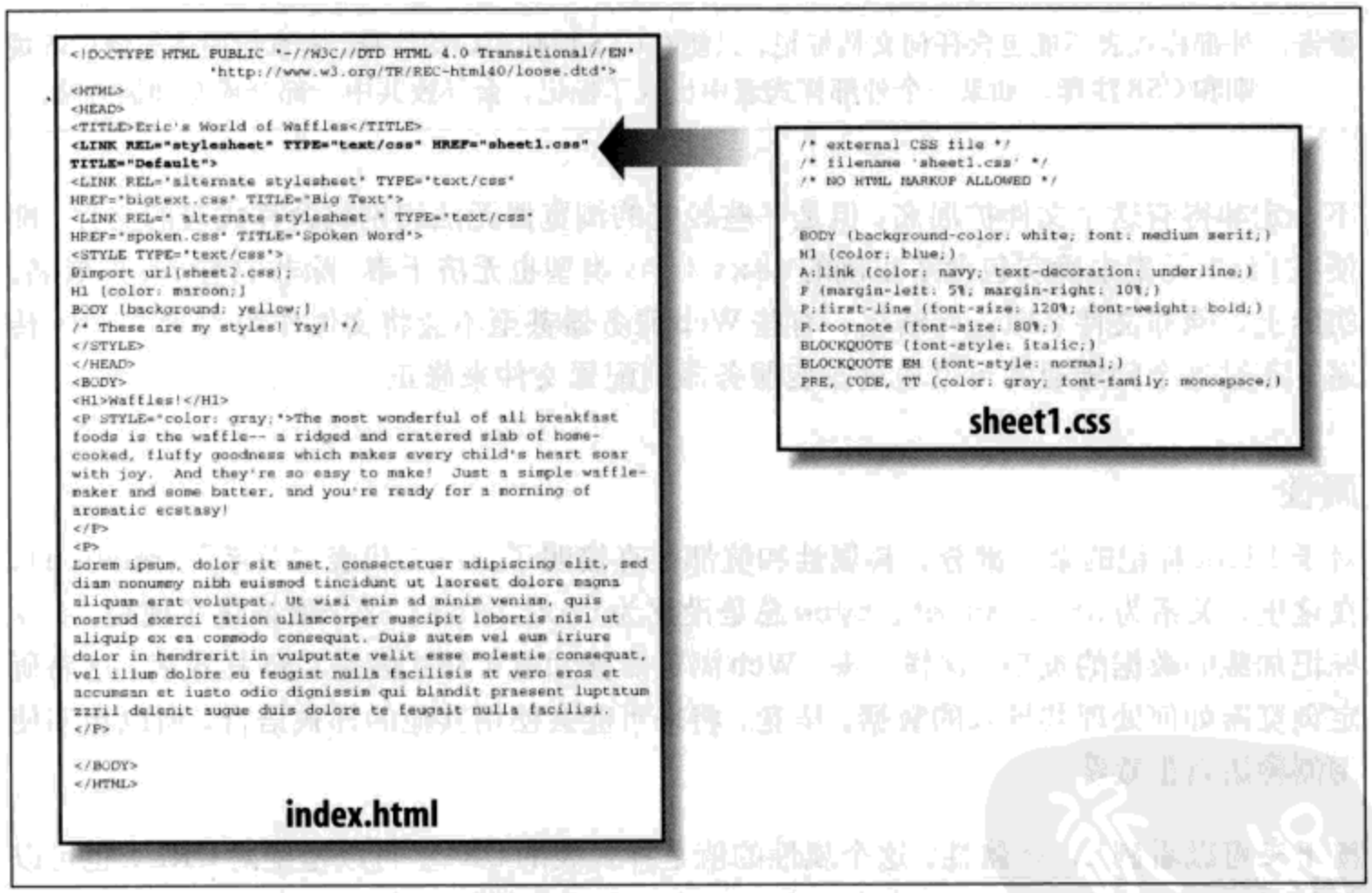

图 1-5：外部样式表如何应用到文档

这些样式表并不是 HTML 文档的一部分，但是仍会由 HTML 文档使用，这称为外部样式表（external style sheet），因为它们是 HTML 文档外部的样式表（如图 1-5 所示）。

为了成功地加载一个外部样式表，link 必须放在 head 元素中，但不能放在其他元素内部（如 title）。这样一来，Web 浏览器就能找到并加载样式表，然后使用其中包含的样式来显示 HTML 文档，如图 1-5 中所示。

外部样式表有什么格式呢？这只是一个规则列表，就像上一节和示例 XHTML 文档中见到的一样，不过在这种情况下，规则会保存到其自己的文件中。只是要记住，样式表

中不能包含XHTML或任何其他标记语言，只能有样式规则。以下是一个外部样式表的内容：

```
h1 {color: red;}
h2 {color: maroon; background: white;}
h3 {color: white; background: black;
 font: medium Helvetica;}
```

仅此而已 —— 完全没有HTML标记或注释，只有简单的无格式样式声明。这些会保存到一个纯文本文件中，通常有*.css*扩展名，如*sheet1.css*。

警告： 外部样式表不能包含任何文档标记，只能有CSS规则和CSS注释，本章后面将解释CSS规则和CSS注释。如果一个外部样式表中出现了标记，会导致其中一部分或全部被忽略。

不一定非得有这个文件扩展名，但是一些较老的浏览器无法识别包含样式表的文件，即使在link元素中确实包含了正确的text/css类型也无济于事，除非它有*.css*扩展名。实际上，除非文件名以*.css*结尾，有些Web服务器甚至不会将文件作为text/css传递，不过这个问题通常可以通过改变服务器的配置文件来修正。

属性

对于link标记的余下部分，其属性和值都很直接明了。rel代表“关系”(relation)，在这里，关系为stylesheet。type总是设置为text/css。这个值描述了使用link标记加载的数据的类型。这样一来，Web浏览器就知道了样式表是CSS样式表，这将确定浏览器如何处理其导入的数据。毕竟，将来可能会使用其他的样式语言，所以声明使用何种语言很重要。

接下来可以看到href属性。这个属性的值是样式表的URL。可以是绝对URL，也可以是相对URL，具体取决于要做的工作。当然，在我们的例子中，该URL是相对URL。它可以很简单，如http://www.meyerweb.com/sheet1.css。

最后还有一个media属性。这里使用的值是all，说明这个样式表要应用于所有表现媒体。CSS2为这个属性定义了很多可取值。

all

用于所有表现媒体。

aural

用于语音合成器、屏幕阅读器和文档的其他声音表现。

braille
: 用Braille设备表现文档时使用。

embossed
: 用Braille打印设备打印时使用。

handheld
: 用于手持设备，如个人数字助理或支持Web的蜂窝电话。

print
: 为视力正常的用户打印文档时使用，另外还会在显示文档的“打印预览”时使用。

projection
: 用于投影媒体，如发表演讲时显示幻灯片的数字投影仪。

screen
: 在屏幕媒体（如桌面计算机监视器）中表现文档时使用。在这种系统上运行的所有Web浏览器都是屏幕媒体用户代理。

tty
: 在固定间距环境（如电传打字机）中显示文档时使用。

tv
: 在电视上显示文档时使用。

以上大部分媒体类型在当前的Web浏览器中并不支持。其中3个得到最广泛支持的类型是all、screen和print。写这本书时，Opera还支持projection，允许文档作为幻灯片显示。

可以在多个媒体中使用一个样式表，为此要提供应用此样式表的媒体列表，各媒体用逗号分隔。例如，可以在屏幕和投影媒体中使用一个链接样式表：

```
<link rel="stylesheet" type="text/css" href="visual-sheet.css"
   media="screen, projection" />
```

注意，一个文档可能关联有多个链接样式表。如果是这样，文档最初显示时只会使用rel为stylesheet的link标记。因此，如果希望链接名为*basic.css*和*splash.css*的两个样式表，可以如下设置：

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

这将会让浏览器加载这两个样式表，合并它们的规则，并将其全部应用于文档（第3章中将清楚地看到这些样式表如何合并，不过现在只要了解它们会合并就足够了）。例如：

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

<p class="a1">This paragraph will be gray only if styles from the
stylesheet 'basic.css' are applied.</p>
<p class="b1">This paragraph will be gray only if styles from the
stylesheet 'splash.css' are applied.</p>
```

上述示例标记中没有title属性，不过title确实是link的一个属性。这个属性不经常使用，但将来可能会很重要，而且如果使用不当，可能会有意想不到的后果。为什么呢？这个问题将在下一节讨论。

候选样式表

还可以定义候选样式表（alternate style sheet）。将rel属性的值置为alternate stylesheet，就可以定义候选样式表，只有在用户选择这个样式表时才会用于文档表现。

如果浏览器能使用候选样式表，它会使用link元素的title属性值生成一个候选样式列表。可以设置以下元素：

```
<link rel="stylesheet" type="text/css"
   href="sheet1.css" title="Default" />
<link rel="alternate stylesheet" type="text/css"
   href="bigtext.css" title="Big Text" />
<link rel="alternate stylesheet" type="text/css"
   href="zany.css" title="Crazy colors!" />
```

然后用户就能选择他们想用的样式，浏览器会从第一个样式（在这里，也就是标记为“Default”的样式）切换为用户选择的任何样式。图1-6显示了完成这种选择的一种办法。

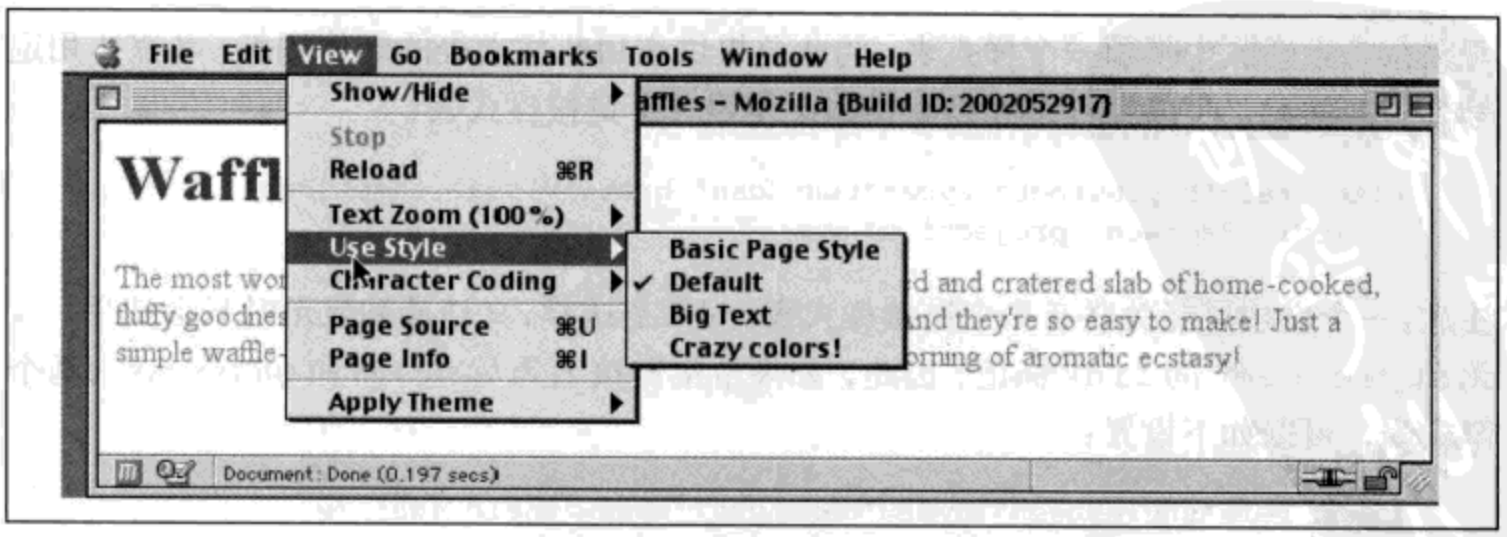

图1-6：提供候选样式表选择的浏览器

注意： 候选样式表在大多数基于Gecko的浏览器中都得到了支持，如Mozilla、Netscape 6+以及Opera 7。通过使用JavaScript，Internet Explorer也能支持候选样式表，不过Internet Explorer本身对此并不提供支持。

还可以为候选样式表指定同样的title值，把它们分组在一起。这样一来，用户就可以在屏幕和打印媒体中为网站选择不同的表现。例如：

```
<link rel="stylesheet" type="text/css"
   href="sheet1.css" title="Default" media="screen" />
<link rel="stylesheet" type="text/css"
   href="print-sheet1.css" title="Default" media="print" />
<link rel="alternate stylesheet" type="text/css"
   href="bigtext.css" title="Big Text" media="screen" />
<link rel="alternate stylesheet" type="text/css"
   href="print-bigtext.css" title="Big Text" media="print" />
```

如果用户在一个兼容用户代理中使用候选样式表选择机制选择了“Big Text”，就会在屏幕媒体中使用*bigtext.css*创建文档的样式，而在打印媒体中使用*print-bigtext.css*。在任何媒体中都不会使用*sheet1.css*或*print-sheet1.css*。

为什么呢？原因在于，如果为一个rel为stylesheet的link指定了标题（title），也就指定了该样式表要作为首选样式表（preferred style sheet）。这意味着，这个样式表要优先于候选样式表使用，而且第一次显示文档时会使用这个首选样式表。不过，一旦选择了候选样式表，就不会再使用首选样式表了。

此外，如果将一组样式表指定为首选样式表，那么只会使用其中的某一个，除此以外，其他的都会被忽略。请考虑以下元素：

```
<link rel="stylesheet" type="text/css"
   href="sheet1.css" title="Default layout" />
<link rel="stylesheet" type="text/css"
   href="sheet2.css" title="Default text sizes" />
<link rel="stylesheet" type="text/css"
   href="sheet3.css" title="Default colors" />
```

所有这三个link元素现在都指示首选样式表，因为所有这3个元素中都出现了一个title属性，但是只有其中的一个会真正作为首选样式表。另外两个会被完全忽略。会是哪两个呢？这是无法确定的，因为HTML和XHTML都没有提供任何方法来确定哪些首选样式表会被忽略，或者应当使用哪个样式表。

如果没有为样式表指定title，那么它将作为一个永久样式表（persistent style sheet），始终用于文档的显示。通常，这正是创作人员所希望的。

style 元素

可以用 style 元素包含样式表，它在文档中单独出现：

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

style 一定要使用 type 属性；对于 CSS 文档，正确的 type 属性值是 "text/css"，这与 link 元素类似。

style 元素始终要以 <style type="text/css"> 开头，如上例所示。其后可以有一个或多个样式，最后以一个结束 </style> 标记结尾。还可以为 style 元素指定一个 media 属性，其可取值与先前讨论链接样式表时介绍的 media 属性值相同。

开始和结束 style 标记之间的样式称为文档样式表（document style sheet），或嵌套样式表（embedded style sheet），因为这个样式表嵌套在文档中。其中可能包含应用到文档的多个样式，还可以使用 @import 指令包含多个外部样式表链接。

@import 指令

下面我们来讨论 style 标记中的内容。首先来看与 link 非常类似的 @import 指令：

```
@import url(sheet2.css);
```

与 link 类似，@import 用于指示 Web 浏览器加载一个外部样式表，并在表现 HTML 文档时使用其样式。唯一的区别在于命令的具体语法和位置。可以看到，@import 出现在 style 容器中。它必须放在这里，也就是要放在其他 CSS 规则之前，否则将根本不起作用。考虑下面的例子：

```
<style type="text/css"> @import url(styles.css); /* @import comes first */
h1 {color: gray;}
</style>
```

类似于 link，一个文档中可以有不只一个 @import 语句。但不同于 link，每个 @import 指令的样式表都会加载并使用；用 @import 无法指定候选样式表。因此，给定以下标记：

```
@import url(sheet2.css);
@import url(blueworld.css);
@import url(zany.css);
```

这 3 个外部样式表都会加载，而且其中的所有样式规则都会在文档的显示中使用。

警告： 许多较老的浏览器无法处理不同形式的@import指令。可以适当利用这一点，对这些浏览器“隐藏”样式。相关的更多细节，请参见 *http://w3development.de/css/hide_css_from_browsers*。

与link一样，可以限制所导入的样式表应用于一种或多种媒体，可以在样式表的URL之后列出要应用此样式的媒体：

```
@import url(sheet2.css) all;
@import url(blueworld.css) screen;
@import url(zany.css) projection, print;
```

如果有一个外部样式表，它需要使用其他外部样式表中的样式，此时@import就非常有用。由于外部样式表不能包含任何文档标记，所以不能使用link元素，但能用@import。因此，可以有一个包含以下内容的外部样式表：

```
@import url(http://example.org/library/layout.css);
@import url(basic-text.css);
@import url(printer.css) print;
body {color: red;}
h1 {color: blue;}
```

不错，尽管具体的样式可能并非如此，不过从中你应该能了解到我的意思。注意，前例中不仅使用了绝对URL，还使用了相对URL。这两种URL形式都可用，这一点与link元素一样。

还要注意，@import 指令出现在样式表的开头，如示例文档所示。CSS要求@import指令出现在样式表中的其他规则之前。如果一个@import 出现在其他规则（如body {color: red;}）之后，兼容用户代理会将其忽略。

警告： Windows平台的Internet Explorer不会忽略任何@import指令，甚至出现在其他规则之后的@import也不会忽略。由于其他浏览器确实会忽略放置不当的@import指令，所以很容易错误地将@import指令放在不合适的位置，以至于改变了其他浏览器中的显示。

具体的样式规则

本例中，@import语句之后就是一些正常的样式规则。这些规则对于这里的讨论意义不大，不过你可能会猜出它们将h1元素设置为紫红色，并设置body元素有一个黄色背景。

```
h1 {color: maroon;}
```

```
body {background: yellow;}
```

所有嵌套样式表都主要由这种样式构成，可以简单也可以复杂，可以很短也可以很长。如果一个文档中 style 元素不包含任何规则，这倒是很少见。

向后可访问性

如果你想知道如何让较老的浏览器也能访问你的文档，需要注意一个很重要的问题。你可能知道浏览器会忽略其无法识别的标记。例如，如果 Web 页面中包含一个 blooper 标记，浏览器会完全忽略这个标记，因为浏览器无法识别这个标记。

对于样式表也同样如此。如果一个浏览器无法识别 <style> 和 </style>，就会将其统统忽略。不过，这些标记中的声明不一定会被忽略，因为对浏览器而言，它们看上去就像是正常的文本。所以，样式声明会出现在页面的最上面！（当然，浏览器应当忽略这些文本，因为这不是 body 元素的一部分，但是事实总是相反。）

为了解决这个问题，建议将声明包含在一个注释标记中。在此给出的示例中，注释标记的开始标记（<!--）紧接着出现在 style 开始标记之后，注释结束标记（-->）出现在 style 结束标记之前：

```
<style type="text/css"><!--
@import url(sheet2.css);
h1 {color: maroon;}
body {background: yellow;}
--></style>
```

这样一来，较老的浏览器不仅会将 style 标记完全忽略，还会忽略声明，因为 HTML 注释不会显示出来。与此同时，能理解 CSS 的浏览器仍能正常地读取样式表。

CSS 注释

CSS 还支持注释。与 C/C++ 注释非常相似，CSS 注释也用 /* 和 */ 包围：

```
/* This is a CSS1 comment */
```

类似于 C++，注释可以跨多行：

```
/* This is a CSS1 comment, and it
can be several lines long without
any problem whatsoever. */
```

要记住重要的一点，CSS 注释不能嵌套。所以，举例来说，以下注释是不正确的：

```
/* This is a comment, in which we find
 another comment, which is WRONG
```

```
  /* Another comment */
and back to the first comment */
```

不过，一般不太需要嵌套注释，所以这个限制不算什么问题。

警告：可能偶尔会创建这种“嵌套”注释，比如说暂时将已经包含注释的一个样式表块注释掉。由于CSS不允许嵌套的注释，“外部”注释会在“内部”注释结束处结束。

如果希望将注释放在标记的同一行上，要谨慎处理注释的放置。例如，以下是正确的做法：

```
h1 {color: gray;}   /* This CSS comment is several lines */
h2 {color: silver;} /* long, but since it is alongside */
p {color: white;}   /* actual styles, each line needs to */
pre {color: gray;}  /* be wrapped in comment markers. */
```

对于这个例子，如果没有对每一行分别增加注释符号，那么样式表中的大部分都会成为注释，以至于无法正常工作：

```
h1 {color: gray;}   /* This CSS comment is several lines
h2 {color: silver;}  long, but since it is not wrapped
p {color: white;}    in comment markers, the last three
pre {color: gray;}   styles are part of the comment. */
```

在这个例子中，只有第一个规则（`h1 {color: gray;}`）会应用到文档。余下的规则都会作为注释的一部分，将被浏览器的表现引擎忽略。

继续看这个例子，会看到XHTML标记中还有更多CSS信息！

内联样式

如果你只是想为单个元素指定一些样式，而不需要嵌套或外部样式表，就可以使用HTML的`style`属性来设置一个内联样式（inline style）（译注2）：

```
<p style="color: gray;">The most wonderful of all breakfast foods is
the waffle--a ridged and cratered slab of home-cooked, fluffy goodness...
</p>
```

除了在`body`外部出现的标记（例如，`head`或`title`），`style`属性可以与任何其他HTML标记关联。

译注2：此处“inline”不能理解为“行内”，而应当是“内联”，有“内部自带”的意思。

style属性的语法很普通。实际上，看上去它与style容器中的声明非常相似，只不过这里大括号要换成双引号。所以<p style="color: maroon; background: yellow;">只会把段落的文本颜色设置为紫红色，背景设置为黄色。文档的其他部分不受此声明影响。

注意，一个内联style属性中只能放一个声明块，而不能放整个样式表。因此，不能在style属性中放@import，也不能包含完整的规则。style属性的值中只能是规则中出现在大括号之间的部分。

通常并不推荐使用style属性。实际上，XHTML 1.1已经将其标注为不建议使用，而且除了XHTML外，XML语言中不太可能出现这个属性。如果把样式放在style属性中，会抵消CSS的一些重要优点，如原本CSS可以组织管理能控制整个文档外观（或者一个Web服务器上所有文档的外观）的集中式样式，而内联样式会削弱这个功能。从很多方面来讲，内联样式并不比font标记强多少，不过内联样式确实提供了更大的灵活性。

小结

利用CSS，可能会完全改变用户代理表现元素的方式。可以使用display属性采用基本方式来显示，也可以将样式表与文档关联，以另一种不同的方式表现。用户不会知道这是通过外部样式表还是嵌套样式表完成的（甚至有可能是利用一个内联样式做到的）。外部样式表真正的意义在于，它允许创作人员将网站的所有表现信息都放在一个位置，将所有文档指向这个位置。这不仅使网络的更新和维护相当容易，还有助于节省带宽，因为文档中去除了所有表现信息。

为了充分利用CSS的强大功能，创作人员需要了解如何将一组样式与文档中的元素相关联。要全面地理解CSS如何做到这些，创作人员则需要深入地掌握CSS以何种方式选择文档中要应用样式的部分，这正是下一章要讨论的主题。

第2章

选择器

CSS的主要优点之一（特别是对于设计人员来说），就是它能很容易地向所有同类型的元素应用一组样式。是不是听上去还不够震撼？那么请这样想想看：只需编辑一行CSS，就能改变所有标题的颜色！是不是不喜欢现在用的蓝色？那就改变那行代码，标题将会全部变成紫色、黄色、紫红色，或者是你想要的任何其他颜色。这就能让作为设计人员的你更多地关注设计而不是那些琐事。下一次开会讨论时，如果有人希望标题还要有绿色阴影，只需编辑样式，再单击Reload按钮就行了。很酷吧？只需区区几秒就大功告成。

当然，CSS并不能解决你的所有问题，比如说你不能用它来改变GIF的颜色，但是利用CSS确实能更容易地完成一些全局性的修改。所以下面先来介绍选择器和结构。

基本规则

前面已经提到，CSS的一个核心特性就是能向文档中的一组元素类型应用某些规则。例如，假设你想让所有h2元素的文本都显示为灰色。如果使用传统的HTML，就必须在所有h2元素中插入<FONT COLOR="gray">...</FONT>标记：

```
<h2><font color="gray">This is h2 text</font></h2>
```

显然，如果你的文档中包含了大量h2元素，这将是一个很繁琐的过程。更糟糕的是，如果你后来又决定要让所有h2元素都是绿色而不是灰色，就必须再重来一次，手动地为所有h2元素设置相应的标记。

利用CSS，可以创建易于修改和编辑的规则，并且能很容易地将其应用到你定义的所有

文本元素（下一节将解释这些规则如何工作）。例如，只需将以下规则写一次，就能让所有 h2 元素变成灰色：

```
h2 {color: gray;}
```

如果希望将所有 h2 文本都改为其他颜色，如银色，只需把这个规则修改如下：

```
h2 {color: silver;}
```

规则结构

为了更详细地说明规则的概念，下面将规则的结构分解开逐一介绍。

每个规则都有两个基本部分：选择器（selector）和声明块（declaration block）。声明块由一个或多个声明（declaration）组成，每个声明则是一个属性－值对（property-value）。每个样式表由一系列规则组成。图 2-1 显示了规则的各个部分。

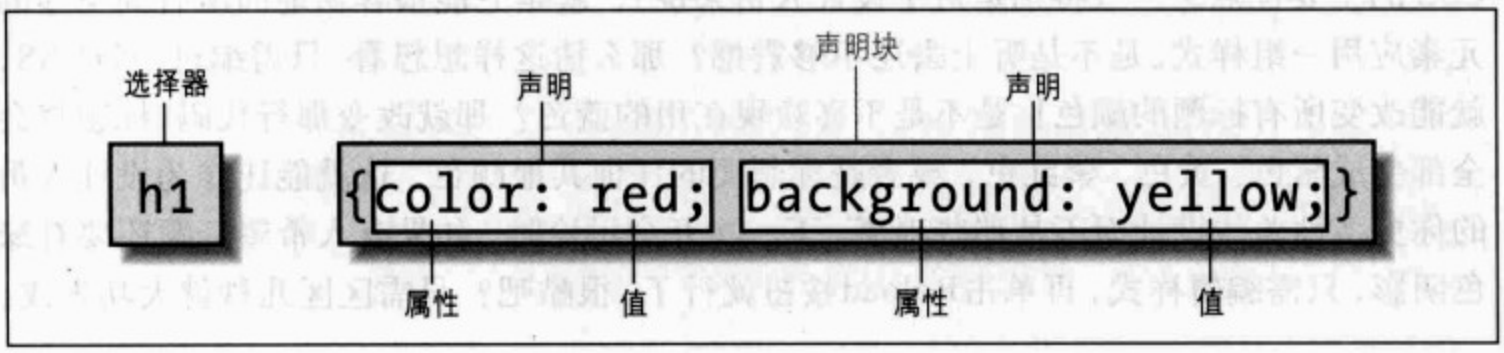

图2-1：规则的结构

如规则左边所示，选择器定义了将影响文档中的哪些部分。图 2-1 中选择了 h1 元素。如果选择器是 p，则会选择所有 p（段落）元素。

规则右边包含声明块，它由一个或多个声明组成。每个声明是一个CSS属性和该属性的值的组合。图 2-1 中，这个声明块包含两个声明。第一个声明指出这个规则将导致文档中某些部分的颜色为红色，第二个声明指出文档中该部分的背景为黄色。所以，文档中的所有 h1 元素（由选择器定义）将有以下样式，即有红色文本和黄色背景。

元素选择器

最常见的选择器往往是HTML元素，但也不完全是这样。例如，如果CSS文件包含XML文档的样式，选择器可能如下所示：

```
QUOTE {color: gray;}
BIB {color: red;}
BOOKTITLE {color: purple;}
MYElement {color: red;}
```

换句话说，文档的元素就是最基本的选择器。在XML中，什么都可以作为选择器，因为XML允许创建新的标记语言，这可能只是一个元素名而已。另一方面，如果设置一个HTML文档的样式，选择器通常将是某个HTML元素，如p、h3、em、a，甚至可以是html本身。例如：

```
html {color: black;}
h1 {color: gray;}
h2 {color: silver;}
```

这个样式表的结果如图2-2所示。

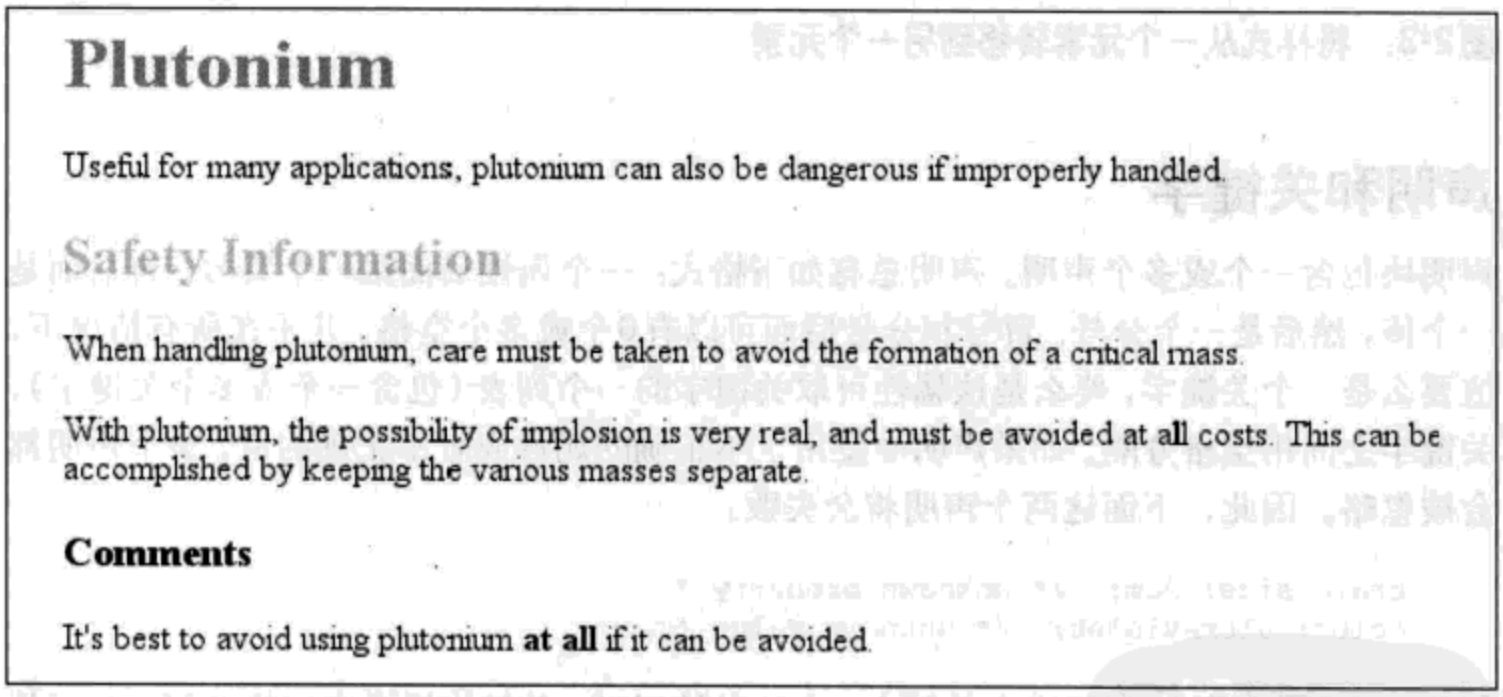

图2-2：一个简单文档的简单样式

一旦直接向元素全局地应用样式，可以将这些样式从一个元素切换应用到另一个元素。假设你决定将图2-2中的段落文本（而不是h1元素）设置为灰色。没问题。只需把h1选择器改为p：

```
html {color: black;}
p {color: gray;}
h2 {color: silver;}
```

结果如图2-3所示。

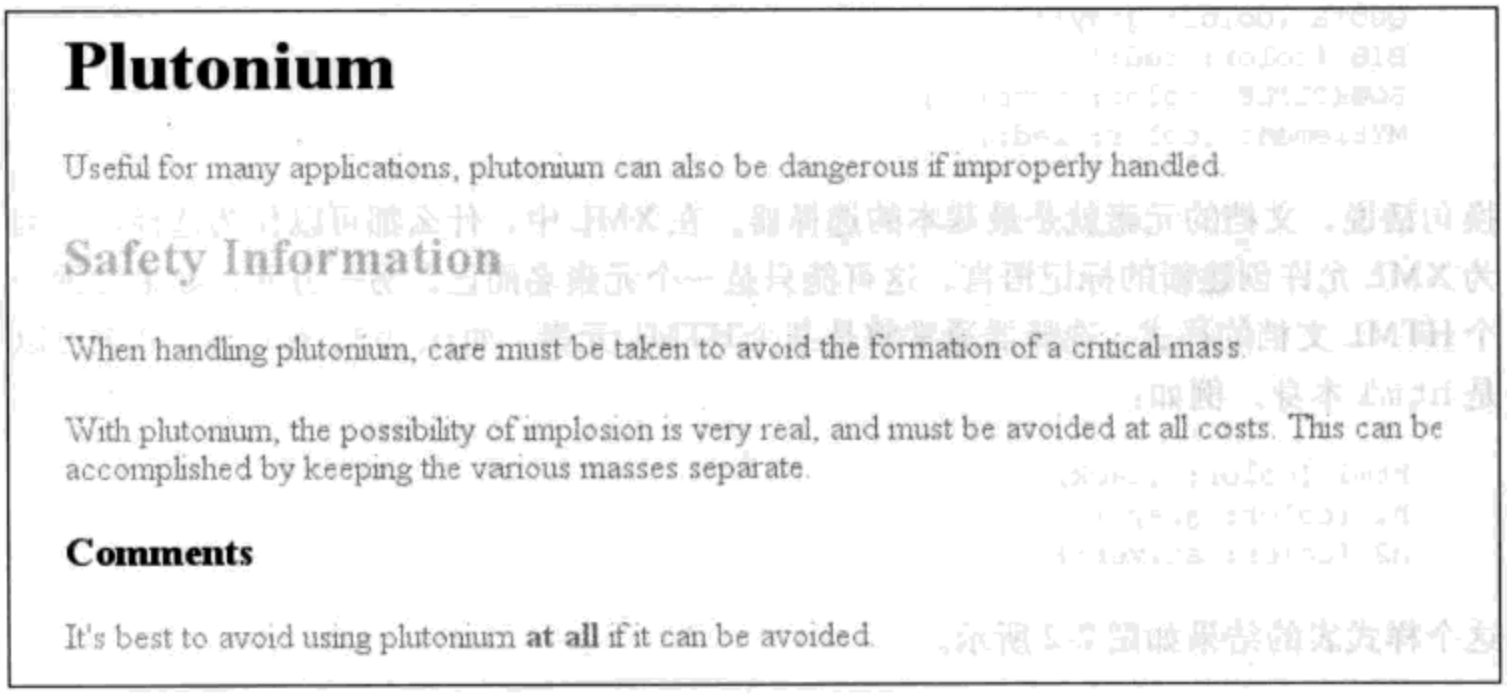

图2-3：将样式从一个元素转移到另一个元素

声明和关键字

声明块包含一个或多个声明。声明总有如下格式：一个属性后面跟一个冒号，再后面是一个值，然后是一个分号。冒号和分号后面可以有0个或多个空格。几乎在所有情况下，值要么是一个关键字，要么是该属性可取关键字的一个列表（包含一个或多个关键字），关键字之间用空格分隔。如果声明中使用了不正确的属性或者不正确的值，整个声明都会被忽略。因此，下面这两个声明将会失败：

```
brain-size: 2cm;  /* unknown property */
color: ultraviolet;  /* unknown value */
```

如果一个属性的值可以取多个关键字，在这种情况下，关键字通常由空格分隔。并不是所有属性都能接受多个关键字，不过确实有许多属性是这样，例如font属性。假设你想为段落文本定义中等大小的Helvetica字体，如图2-4所示。

这个规则可以读作：

```
p {font: medium Helvetica;}
```

注意medium和Helvetica之间的空格，medium和Helvetica都是关键字（前一个指定了字体的大小，后一个是具体的字体名）。两个关键字之间的空格使用户代理能够区分这两个关键字，并适当地应用。后面的分号指示声明结束。

用空格分隔的这些词称为关键字，这是因为，它们加在一起构成了当前属性的值。例如，考虑以下假想规则：

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, the possibility of implosion is very real, and must be avoided at all costs. This can be accomplished by keeping the various masses separate.

Comments

It's best to avoid using plutonium **at all** if it can be avoided.

图2-4：属性值含多个关键字的结果

```
rainbow: red orange yellow green blue indigo violet;
```

当然并没有rainbow这样的属性，另外这里使用的颜色也不合法，不过可以用这个例子来说明有关的概念。rainbow的值是red orange yellow green blue indigo violet，这7个关键字加在一起构成了一个唯一的值。可以把rainbow的值重新定义如下：

```
rainbow: infrared red orange yellow green blue indigo violet ultraviolet;
```

现在rainbow就有了一个新值，由9个而不是7个关键字组成。尽管这两个值的名字是相同的，但是它们就像0和1一样黑白分明，完全不同。

注意： 可以看到，CSS关键字往往由空格分隔，只有一种情况例外。在CSS的font属性中，只有一种情况下可以使用斜线（/）来分隔两个特定关键字。下面是一个例子：

```
h2 {font: large/150% sans-serif;}
```

斜线分隔了用来设置元素的字体大小和行高的两个关键字。只有在这里才允许font声明中出现斜线。font允许的所有其他关键字都用空格分隔。

以上介绍了简单声明的基础知识，不过声明可以比这复杂得多。下一节将带你了解CSS的功能有多强大。

分组

到目前为止，我们已经学习了如何向一个选择器应用一个样式，这个技术相当简单。但

是如果想为多个元素应用同一个样式该怎么做呢？倘若如此，可能要使用多个选择器，或者向一个元素或一组元素应用多个样式。

选择器分组

假设希望h2元素和段落都有灰色文本。为达到这个目的，最容易的做法是使用以下声明：

```
h2, p {color: gray;}
```

将h2和p选择器放在规则的左边，并用一个逗号来分隔，这样就定义了一个规则，其右边的样式（color: gray;）将应用到这两个选择器所引用的元素。逗号告诉浏览器，规则中包含两个不同的选择器。如果没有这个逗号，那么规则的含义则完全不同，有关内容将在“后代选择器”一节中详细介绍。

可以将任意多个选择器分组在一起，对此没有任何限制。例如，如果你想把很多元素显示为灰色，可以使用类似如下的规则：

```
body, table, th, td, h1, h2, h3, h4, p, pre, strong, em, b, i {color: gray;}
```

通过分组，创作人员可以将某些类型的样式“压缩”在一起，这就能得到一个更简短的样式表。以下的两组规则能得到同样的结果，不过可以很清楚地看出哪一个写起来更容易：

```
h1 {color: purple;}
h2 {color: purple;}
h3 {color: purple;}
h4 {color: purple;}
h5 {color: purple;}
h6 {color: purple;}

h1, h2, h3, h4, h5, h6 {color: purple;}
```

分组提供了一些有意思的选择。例如，下例中的所有规则分组都是等价的，每个组只是展示了对选择器和声明分组的不同方法：

```
/* group 1 */
h1 {color: silver; background: white;}
h2 {color: silver; background: gray;}
h3 {color: white; background: gray;}
h4 {color: silver; background: white;}
b {color: gray; background: white;}

/* group 2 */
h1, h2, h4 {color: silver;}
h2, h3 {background: gray;}
h1, h4, b {background: white;}
```

```
h3 {color: white;}
b {color: gray;}

/* group 3 */
h1, h4 {color: silver; background: white;}
h2 {color: silver;}
h3 {color: white;}
h2, h3 {background: gray;}
b {color: gray; background: white;}
```

所有这些分组都能得到图2-5所示的结果（这些样式使用了分组声明，有关内容将在后面的“声明分组”一节中再做解释）。

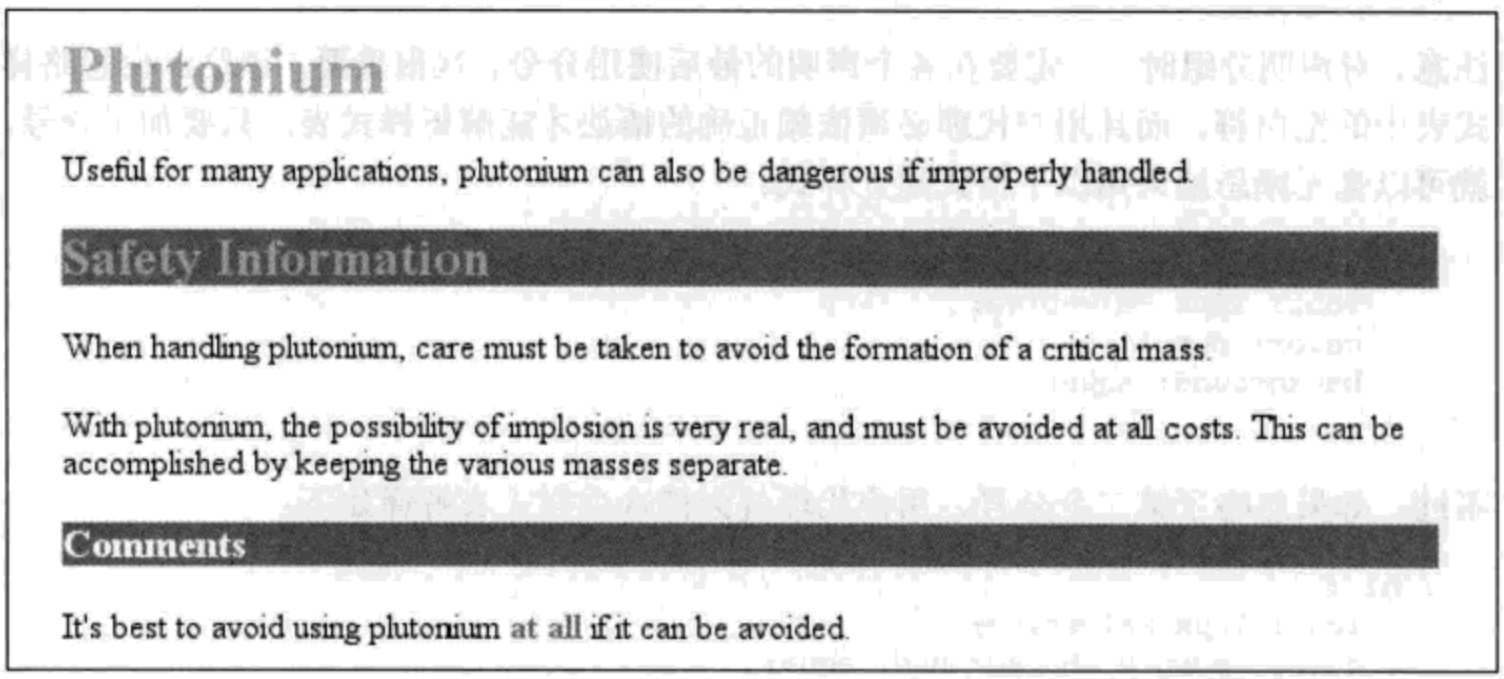

图2-5：等价样式表的结果

通配选择器

CSS2引入了一种新的简单选择器，称为通配选择器（universal selector），显示为一个星号（*）。这个选择器可以与任何元素匹配，就像是一个通配符。例如，要让一个文档中的每一个元素都为红色，可以写为以下规则：

```
* {color: red;}
```

这个声明等价于列出了文档中所有元素的一个分组选择器。利用通配选择器，只需敲一次键（仅一个星号）就能把文档中所有元素的`color`值都指定为`red`。不过要当心，尽管通配选择器很方便，但它也有一些意想不到的后果，下一章将对此进行详细说明。

声明分组

既然可以将选择器分组到一个规则中，当然也可以对声明分组。假设你希望所有`h1`元素都有浅绿色背景，并使用18像素高的Helvetica字体显示为紫色文本（你并不关心读

者能不能看清楚)。可以写以下样式:

```
h1 {font: 18px Helvetica;}
h1 {color: purple;}
h1 {background: aqua;}
```

但是这种方法效率不高;想想看,如果为一个有10个或15个样式的元素创建这样一个列表会多麻烦!相反,可以将声明分组在一起:

```
h1 {font: 18px Helvetica; color: purple; background: aqua;}
```

这与前面的3行样式表的效果完全一样。

注意,对声明分组时,一定要在各个声明的最后使用分号,这很重要。浏览器会忽略样式表中的空白符,而且用户代理必须依赖正确的语法才能解析样式表。只要加了分号,就可以毫无顾忌地采用以下格式建立样式:

```
h1 {
  font: 18px Helvetica;
  color: purple;
  background: aqua;
}
```

不过,如果忽略了第二个分号,用户代理就会把这个样式表解释如下:

```
h1 {
  font: 18px Helvetica;
  color: purple  background: aqua;
}
```

因为background: 对于color来说不是一个合法值,而且由于只能为color指定一个关键字,所以用户代理会完全忽略这个color声明(包括background: aqua部分)。这样h1标题只会显示为紫色文本,而没有淡绿色背景,不过更有可能根本得不到紫色的h1。相反,这些标题只会显示为默认颜色(通常是黑色),而且根本没有背景色(font: 18px Helvetica声明仍能正常起作用,因为它确实正确地以一个分号结尾)。

注意: 尽管从技术上讲没有必要让规则的最后一个声明也以分号结尾,不过这通常是一个好的实践做法。首先,这会让你养成在声明后加上分号的好习惯。声明的最后缺少分号,是导致表现出错的最常见的原因之一。其次,如果你决定向规则增加另一个声明,就不必担心忘记再插入一个分号。最后一点,如果规则中的最后一个声明少了分号,一些较老的浏览器(如Internet Explorer 3.x)很可能会不知所措。一定要避免所有这些问题,所以只要出现规则,就要在声明的后面加一个分号。

与选择器分组一样，声明分组也是一种便利的方法，可以缩短样式表，使之表述更为清晰，而且易于维护。

结合选择器和声明的分组

现在应该了解到，可以对选择器分组，还可以对声明分组。如果在一个规则中结合这两种分组，就可以使用很少的一些语句定义相当复杂的样式。现在，如果你想为文档中的所有标题指定某种复杂的样式，而且希望向所有这些标题应用同样的样式，该怎么办呢？可以这么做：

```
h1, h2, h3, h4, h5, h6 {color: gray; background: white; padding: 0.5em;
  border: 1px solid black; font-family: Charcoal, sans-serif;}
```

以上对选择器进行了分组，所以规则右边的样式会应用到所列的所有标题上，而且对声明分组意味着所列的所有样式都会应用到规则左边的选择器。这个规则的结果如图2-6所示。

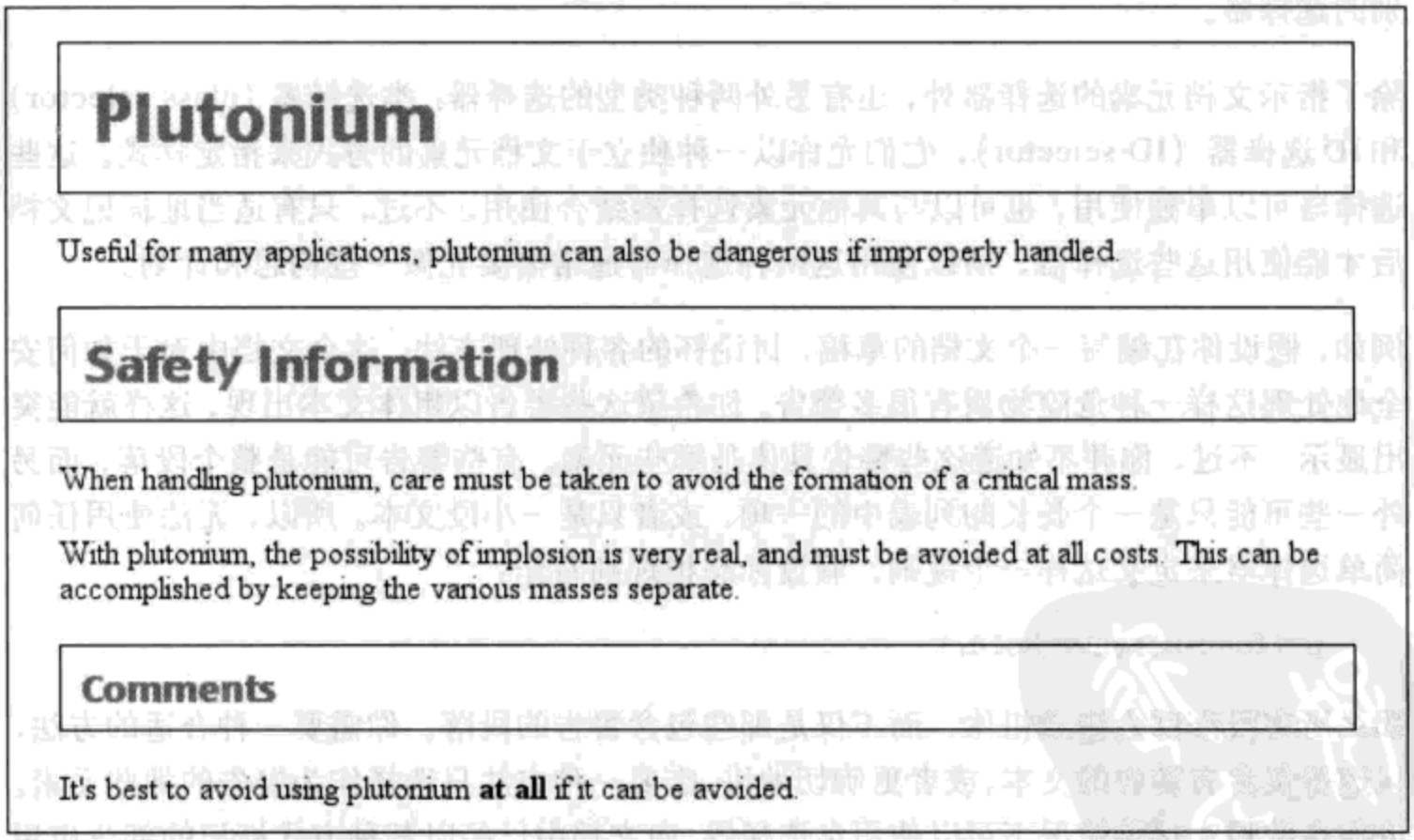

图2-6：选择器和声明的分组

这种方法对比较长的样式很适用，该样式可能如下：

```
h1 {color: gray;}
h2 {color: gray;}
h3 {color: gray;}
h4 {color: gray;}
```

```
h5 {color: gray;}
h6 {color: gray;}
h1 {background: white;}
h2 {background: white;}
h3 {background: white;}
```

而且后面还有很多行。用这种方式写出长长的样式也是可以的，不过我不推荐这样的方法，编辑这样一个冗长的样式就像到处使用 font 标记一样麻烦！

还可以向选择器增加更多的表达式，并按信息的类型以“快捷”方式选择元素来应用样式。当然，要想得到如此强大的功能，还需要先做一点准备工作，但这是很值得的。

类选择器和 ID 选择器

到目前为止，我们已经采用多种方式对选择器和声明进行了分组，不过之前使用的选择器还很简单，只指示文档元素。从某种意义上讲它们很不错，但有时你可能还需要更特别的选择器。

除了指示文档元素的选择器外，还有另外两种类型的选择器：类选择器（class selector）和 ID 选择器（ID selector），它们允许以一种独立于文档元素的方式来指定样式。这些选择器可以单独使用，也可以与其他元素选择器结合使用。不过，只有适当地标记文档后才能使用这些选择器，所以使用这两种选择器通常需要先做一些构想和计划。

例如，假设你在编写一个文档的草稿，讨论钚的各种处理方法。这个文档中对于如何安全地处理这样一种危险物质有很多警告。你希望这些警告以粗体文本出现，这样就能突出显示。不过，你并不知道这些警告具体是哪些元素。有些警告可能是整个段落，而另外一些可能只是一个长长的列表中的一项，或者只是一小段文本。所以，无法使用任何简单选择器来定义这样一个规则。假设你想把规则写作：

```
p {font-weight: bold;}
```

那么所有段落都会变成粗体，而不仅是那些包含警告的段落。你需要一种合适的方法，只选择包含有警告的文本，或者更确切地讲，需要一种方法只选择作为警告的那些元素。该怎么做呢？这种情况下可以使用类选择器，向文档中已经以某种方式标记的部分应用样式，而不考虑具体涉及到哪些元素。

类选择器

要应用样式而不考虑具体涉及的元素，最常用的方法就是使用类选择器。不过，在使用类选择器之前，需要修改具体的文档标记，以便类选择器正常工作。输入以下 class 属性：

```
<p class="warning">When handling plutonium, care must be taken to avoid
the formation of a critical mass.</p>
<p>With plutonium, <span class="warning">the possibility of implosion is
very real, and must be avoided at all costs</span>. This can be accomplished
by keeping the various masses separate.</p>
```

为了将一个类选择器的样式与元素关联，必须将class属性指定为一个适当的值。在前面的代码中，有两个元素的class值指定为warning：第一段（第一个p元素）以及第二段中的span元素。

现在所需要的仅是向这些归类的元素应用样式的方法。对于HTML文档，可以使用一种很简洁的记法，即类名前有一个点号（.），而且可以结合一个简单选择器：

```
*.warning {font-weight: bold;}
```

结合前面所示的示例标记，这个简单规则的效果如图2-7所示。也就是说，font-weight: bold的样式会应用到class属性值为warning的所有元素（因为这里有一个通配选择器）。

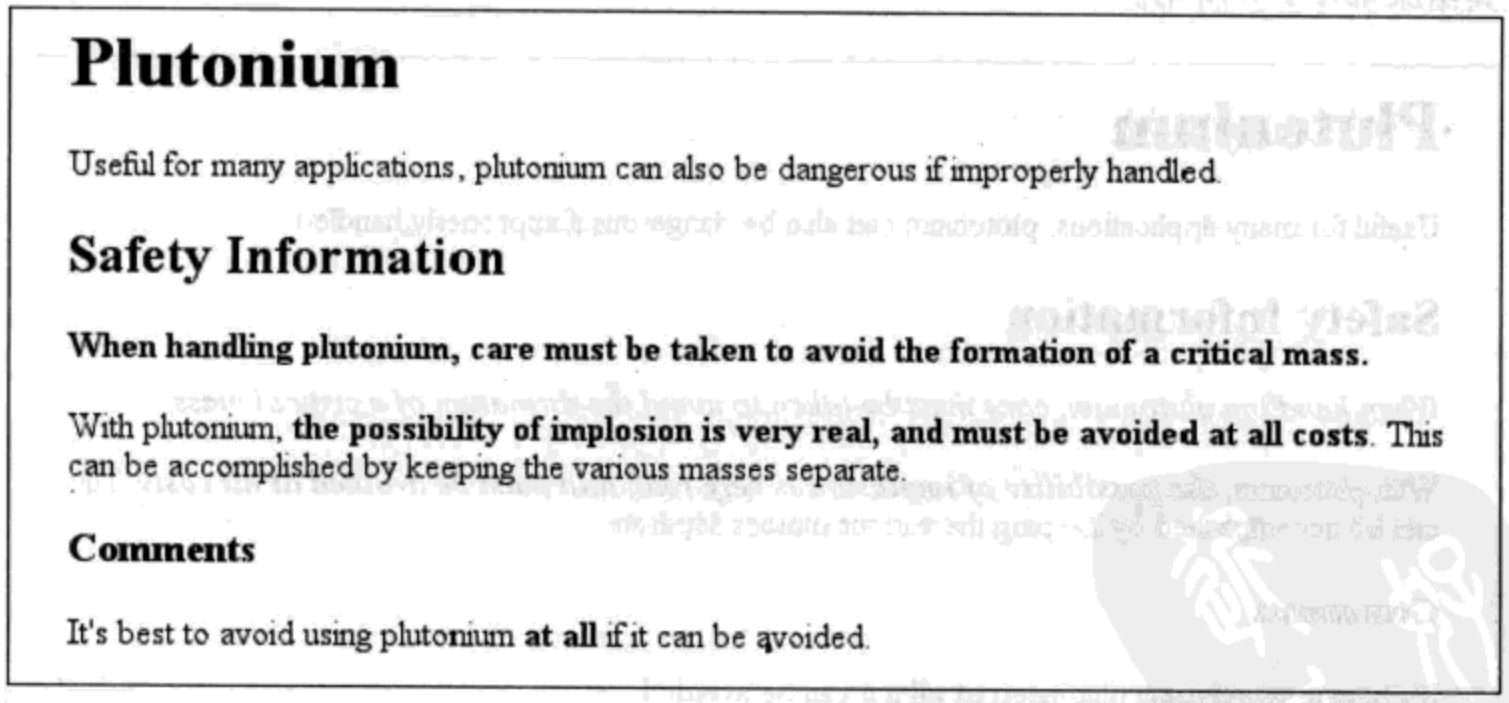

图2-7：使用类选择器

可以看到，类选择器要正常工作，需要直接引用一个元素的class属性中的值。这个引用前面往往有一个点号（.），标记这是一个类选择器。通过这个点号，可以帮助类选择器与它可能结合的其他部分分隔开（如类选择器可能结合了元素选择器）。例如，你可能希望只在整个段落是警告时才显示为粗体文本：

```
p.warning {font-weight: bold;}
```

选择器现在会匹配class属性包含warning的所有p元素，但是其他任何类型的元素

都不匹配，不论是否有此class属性。选择器p.warning解释为："其class属性包含词warning的所有段落。"因为span元素不是一个段落，这个规则的选择器与之不匹配，因此span元素不会变成粗体文本。

如果你确实希望为span元素指定不同的样式，可以使用选择器span.warning：

```
p.warning {font-weight: bold;}
span.warning {font-style: italic;}
```

在这种情况下，警告段落会变成粗体，而警告span会变成斜体。每个规则只应用于某种特定类型的元素/类组合，所以不会影响其他元素。

另一种选择是组合使用一个通用类选择器和一个元素特定类选择器，这样可以得到更有用的样式，如以下标记所示：

```
.warning {font-style: italic;}
span.warning {font-weight: bold;}
```

其结果如图2-8所示。

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, ***the possibility of implosion is very real, and must be avoided at all costs***. This can be accomplished by keeping the various masses separate.

Comments

It's best to avoid using plutonium **at all** if it can be avoided.

图2-8：结合使用通用选择器和特定选择器设置样式

在这种情况下，所有警告文本都会变成斜体，不过只有class为warning的span元素中的文本会变为粗斜体。

注意前例中通用类选择器的格式：这里只有一个类名，前面有一个点号而没有任何元素名。如果你只想选择所有类名相同的元素，可以在类选择器中忽略通配选择器，这没有任何不好的影响。

多类选择器

在上一节中，我们处理了class值中包含一个词的情况。在HTML中，一个class值中有可能包含一个词列表，各个词之间用空格分隔。例如，如果希望将一个特定的元素同时标记为紧急（urgent）和警告（warning），就可以写作：

```
<p class="urgent warning">When handling plutonium, care must be taken to
avoid the formation of a critical mass.</p>
<p>With plutonium, <span class="warning">the possibility of implosion is
very real, and must be avoided at all costs</span>. This can be accomplished
by keeping the various masses separate.</p>
```

这两个词的顺序无关紧要，写成warning urgent也可以。

现在假设希望class为warning的所有元素都是粗体，而class为urgent的所有元素为斜体，class中同时包含warning和urgent的所有元素还有一个银色的背景。就可以写作：

```
.warning {font-weight: bold;}
.urgent {font-style: italic;}
.warning.urgent {background: silver;}
```

通过把两个类选择器链接在一起，仅可以选择同时包含这些类名的元素（类名的顺序不限）。可以看到，HTML源代码中包含class="urgent warning"，但CSS选择器写作.warning.urgent。不过，这个规则还是会导致“When handling plutonium . . . ”段落有一个银色的背景，如图2-9所示。

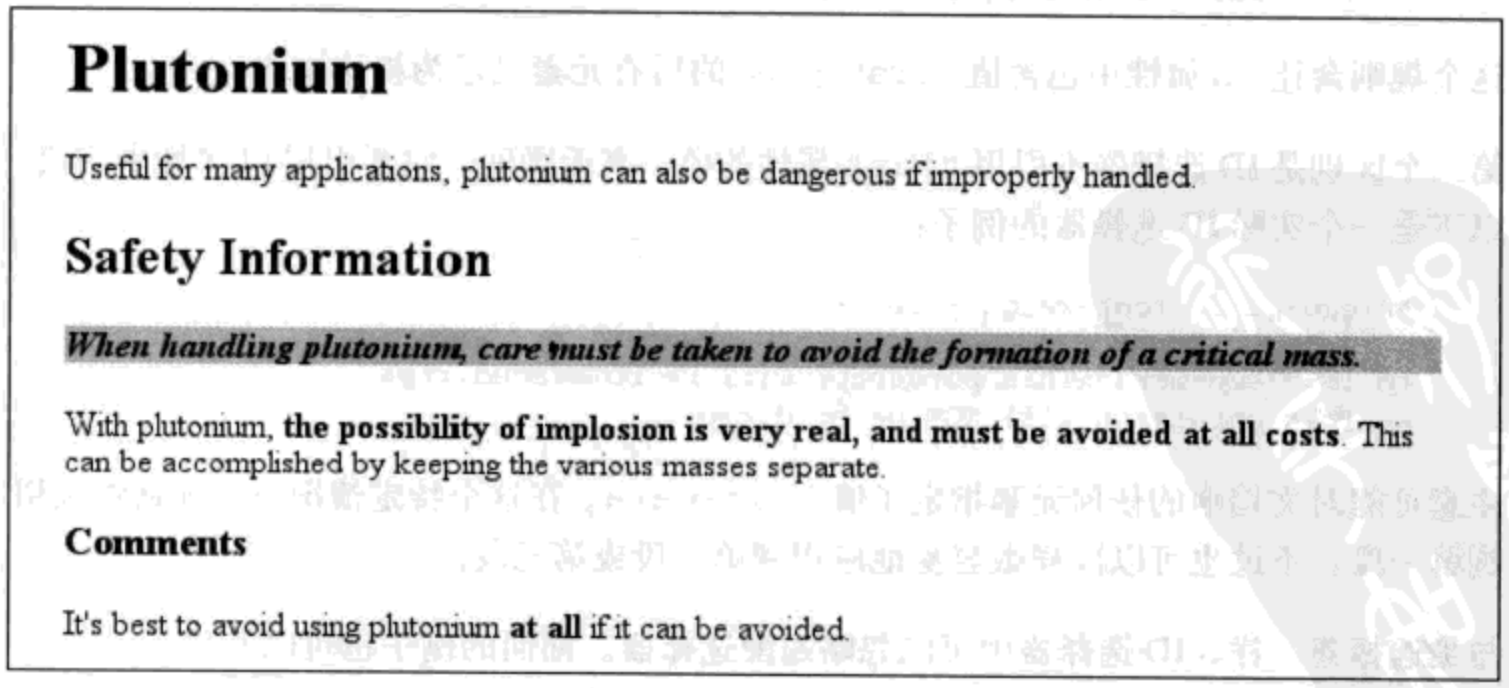

图2-9：选择有多个类名的元素

如果一个多类选择器包含类名列表（类名以空格分隔）中所没有的一个类名，匹配就会失败。考虑以下规则：

```
p.warning.help {background: red;}
```

不出所料，这个选择器将只匹配class包含词warning和help的那些p元素。因此，如果一个p元素的class属性中只有词warning和urgent，将不能匹配。不过，它能匹配以下元素：

```
<p class="urgent warning help">Help me!</p>
```

警告： 在IE7之前的版本中，不同平台的Internet Explorer都不能正确地处理多类选择器。对于这些较早的版本，尽管可以选择列表中的一个类名，但是要根据列表中的多个类名进行选择时则无法工作。因此，p.warning可以正常工作，但p.warning.help会匹配class属性中包含help的所有p元素，因为help在选择器中最后出现。如果写作p.help.warning，较老版本的Explorer就会匹配class值中包含warning的所有p元素，而不论其中是否出现help。

ID选择器

在某些方面，ID选择器类似于类选择器，不过也有一些重要的差别。首先，ID选择器前面有一个#号——也称为棋盘号而不是点号。因此，可以看到如下的一个规则：

```
*#first-para {font-weight: bold;}
```

这个规则会让id属性中包含值first-para的所有元素显示为粗体文本。

第二个区别是ID选择器不引用class属性的值，毫无疑问，它要引用id属性中的值。以下是一个实际ID选择器的例子：

```
*#lead-para {font-weight: bold;}

<p id="lead-para">This paragraph will be boldfaced.</p>
<p>This paragraph will NOT be bold.</p>
```

注意可能对文档中的任何元素指定了值lead-para。在这个特定情况下，规则将应用到第一段，不过也可以同样很容易地应用到第二段或第三段。

与类选择器一样，ID选择器中可以忽略通配选择器。前面的例子也可以写作：

```
#lead-para {font-weight: bold;}
```

这个选择器的效果将是一样的。

类选择器还是ID选择器？

如前所述，可以为任意多个元素指定类；前例中类名warning被应用到p和span元素，而且它还可以应用到更多的元素。与此不同，在一个HTML文档中，ID选择器会使用一次，而且仅一次。因此，如果有一个元素的id值为lead-para，那么该文档中所有其他元素的id值都不能是lead-para。

注意： 实际中，浏览器通常并不检查HTML中ID的唯一性，这意味着如果你在HTML文档中设置了多个有相同ID属性值的元素，就可能为这些元素应用相同的样式。这种行为是不正确的，不过这种情况常会发生。如果一个文档中有多个相同的ID值，还会导致编写DOM脚本更为困难，因为像getElementById()之类的函数的前提是仅一个元素有给定的ID值。

不同于类选择器，ID选择器不能结合使用，因为ID属性不允许有以空格分隔的词列表。

从纯语法意义上讲，点号加类名的记法（如.warning）对XML文档不一定能奏效。在写这本书时，点号加类名的记法在HTML、SVG和MathML中能正常工作，也许将来的语言也支持，不过这要由各个语言的规范来决定。#号加ID的记法（如#lead）可以在任何文档语言中使用，只要其中有一个属性在文档中能保证唯一即可。唯一性可以用名为id的属性来保证，或者可以是任何其他属性，只要属性的内容在文档中定义为唯一。

class名和id名之间的另一个区别是，如果你想确定应当向一个给定元素应用哪些样式，ID能包含更多含义。这一内容将在下一章更深入地讨论。

类似于类，还可以独立于元素来选择ID。有些情况下，你知道文档中会出现某个特定的ID值，但是并不知道它会出现在哪个元素上（就像处理钚的警告一样），所以你想声明独立的ID选择器。例如，你可能知道在一个给定的文档中会有一个ID值为mostImportant的元素。你不知道这个最重要的东西是一个段落、一个短语、一个列表项还是一个小节标题。你只知道每个文档中都会有这么一个最重要的内容，它可能出现在任何元素中，而且只能出现一次。在这种情况下，可以编写如下规则：

```
#mostImportant {color: red; background: yellow;}
```

这个规则会与以下各个元素匹配（前面已经提到，这些元素不能在同一个文档中同时出现，因为它们都有相同的ID值）：

```
<h1 id="mostImportant">This is important!</h1>
<em id="mostImportant">This is important!</em>
<ul id="mostImportant">This is important!</ul>
```

还要注意，类选择器和ID选择器可能是区分大小写的，这取决于文档语言。HTML和XHTML将类和ID值定义为区分大小写，所以类和ID值的大小写必须与文档中的相应值匹配。因此，对于以下的CSS和HTML，元素不会变成粗体：

```
p.criticalInfo {font-weight: bold;}

<p class="criticalinfo">Don't look down.</p>
```

由于字母*i*的大小写不同，所以选择器不会匹配以上元素。

警告： 一些较老的浏览器不区分类名和ID名的大小写，但是写这本书时，所有当前的浏览器都要求区分大小写。

属性选择器

对于类选择器和ID选择器，你所做的实际上只是选择属性值。前面两小节中使用的语法是HTML、SVG和MathML文档特定的（写这本书时是如此）。在其他标记语言中，不能使用这些类和ID选择器。为了解决这个问题，CSS2引入了属性选择器（attribute selector），它可以根据元素的属性及属性值来选择元素。共有4种类型的属性选择器。

警告： Safari、Opera和所有基于Gecko的浏览器都支持属性选择器，不过在IE5/Mac和IE6/Win之前，Internet Explorer并不支持属性选择器。IE7全面支持所有CSS2.1属性选择器，还支持一些CSS3属性选择器，这一节将讨论这个内容。

简单属性选择

如果希望选择有某个属性的元素，而不论该属性的值是什么，可以使用一个简单属性选择器。例如，要选择有`class`属性（值不限）的所有`h1`元素，使其文本为银色，可以写作：

```
h1[class] {color: silver;}
```

所以，给定如下标记：

```
<h1 class="hoopla">Hello</h1>
<h1 class="severe">Serenity</h1>
<h1 class="fancy">Fooling</h1>
```

其结果如图2-10所示。

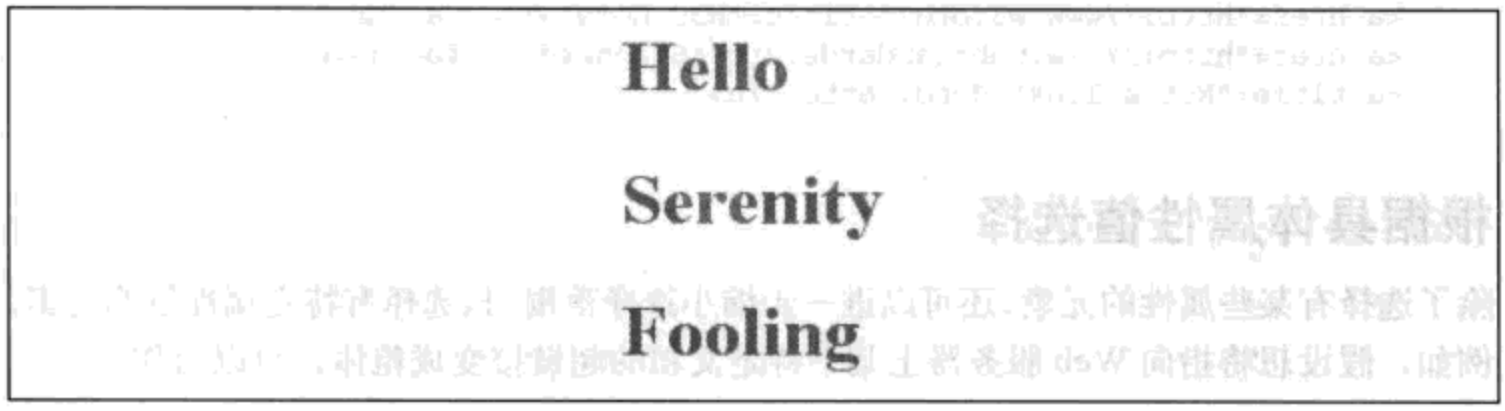

图2-10：根据元素属性来选择元素

这个策略在XML文档中相当有用，因为XML语言主张要针对元素和属性的用途指定元素名和属性名。考虑描述太阳系行星的一个XML语言（我们称之为PlanetML）。如果你想选择有moons属性的所有planet元素，使之显示为粗体，以便能更关注有moons的行星，就可以写作：

```
planet[moons] {font-weight: bold;}
```

这会让以下标记片段中第二个和第三个元素的文本显示为粗体，但第一个元素的文本不是粗体：

```
<planet>Venus</planet>
<planet moons="1">Earth</planet>
<planet moons="2">Mars</planet>
```

在HTML文档中，可以采用一些创造性的方法使用这个特性。例如，可以对所有带有alt属性的图像应用某种样式，从而突出显示这些有效的图像：

```
img[alt] {border: 3px solid red;}
```

（这个特例更适合用来诊断而不是设计，即用来确定图像是否确实有效。）

如果你想把包含标题（title）信息的所有元素变为粗体显示（光标停留在这些元素上时大多数浏览器都会将其显示为“工具提示”），就可以写作：

```
*[title] {font-weight: bold;}
```

类似地，可以只对有href属性的锚（a元素）应用样式。

还可以根据多个属性进行选择，只需将属性选择器链接在一起即可。例如，为了将同时有href和title属性的HTML超链接的文本置为粗体，可以写作：

```
a[href][title] {font-weight: bold;}
```

这会将以下标记中的第一个链接变成粗体，但第二个和第三个链接不变：

```
<a href="http://www.w3.org/" title="W3C Home">W3C</a><br />
<a href="http://www.webstandards.org">Standards Info</a><br />
<a title="Not a link">dead.letter</a>
```

根据具体属性值选择

除了选择有某些属性的元素，还可以进一步缩小选择范围，只选择有特定属性值的元素。例如，假设想将指向 Web 服务器上某个特定文档的超链接变成粗体，可以写作：

```
a[href="http://www.css-discuss.org/about.html"] {font-weight: bold;}
```

可以为任何元素指定属性和值组合。不过，如果文档中没有出现该组合，选择器将无法匹配。同样地，XML语言也可以利用这种方法来设置样式。下面再回到我们的PlanetML例子。假设只想选择 moons 属性值为 1 的那些 planet 元素：

```
planet[moons="1"] {font-weight: bold;}
```

这会把以下标记片段中第二个元素的文本变成粗体，但第一个和第三个元素不受影响：

```
<planet>Venus</planet>
<planet moons="1">Earth</planet>
<planet moons="2">Mars</planet>
```

与属性选择类似，可以把多个属性－值选择器链接在一起来选择一个文档。例如，为了将href值为http://www.w3.org/而且title属性值为W3C Home的所有HTML超链接的文本大小加倍，可以写作：

```
a[href="http://www.w3.org/"][title="W3C Home"] {font-size: 200%;}
```

这会把以下标记中第一个链接的文本大小加倍，但第二个或第三个链接不受影响：

```
<a href="http://www.w3.org/" title="W3C Home">W3C</a><br />
<a href="http://www.webstandards.org"
  title="Web Standards Organization">Standards Info</a><br />
<a href="http://www.example.org/" title="W3C Home">dead.link</a>
```

结果如图 2-11 所示。

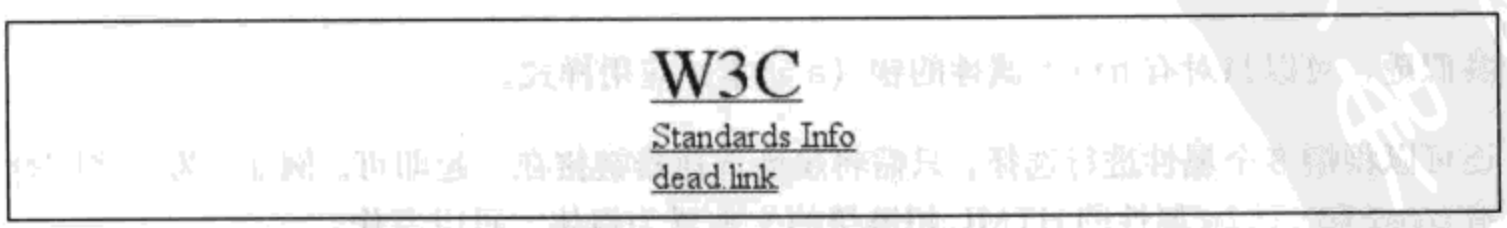

图2-11： 根据属性及其值选择元素

注意，这种格式要求必须与属性值完全匹配。如果遇到的值本身包含一个用空格分隔的值列表（如HTML属性class），匹配就会出问题。例如，考虑以下标记片段：

```
<planet type="barren rocky">Mercury</planet>
```

要根据具体属性值匹配这个元素，唯一的办法就是写作：

```
planet[type="barren rocky"] {font-weight: bold;}
```

如果写成planet[type="barren"]，这个规则不能匹配示例标记，因而会失败。HTML中的class属性也是如此。考虑以下片段：

```
<p class="urgent warning">When handling plutonium, care must be taken to
avoid the formation of a critical mass.</p>
```

要根据具体属性值来选择这个元素，必须写作：

```
p[class="urgent warning"] {font-weight: bold;}
```

这不同于先前介绍的点号类名记法，有关内容将在下一节讨论。相反，这个规则会选择class属性值为urgent warning的所有p元素，要求属性值中urgent在前warning在后，而且有一个空格将其分隔。这实际上就是一个完全串匹配。

另外，要注意ID选择器与指定id属性的属性选择器不是一回事。换句话说，h1#page-title和h1[id="page-title"]之间存在着微妙但很重要的差别。这个差别将在下一章解释。

根据部分属性值选择

如果属性能接受词列表（词之间用空格分隔），可以根据其中的任意一个词进行选择。在HTML中，这方面最经典的例子就是class属性，它能接受一个或多个词作为其属性值。以下是我们经常使用的示例文本：

```
<p class="urgent warning">When handling plutonium, care must be taken to
avoid the formation of a critical mass.</p>
```

假设你想选择class属性中包含warning的元素，可以用一个属性选择器做到这一点：

```
p[class~="warning"] {font-weight: bold;}
```

注意选择器中出现了一个波浪号（~）。这正是部分选择的关键，即根据属性值中出现的一个用空格分隔的词来完成选择。如果忽略了这个波浪号，如上一节所述，则说明需要完成完全值匹配。

这个选择器构造等价于先前讨论的点号类名记法。因此，p.warning和p[class~="warning"]应用到HTML文档时是等价的。以下是一个例子，这是前面“PlanetML”标记的HTML版本：

```
<span class="barren rocky">Mercury</planet>
<span class=" cloudy barren">Venus</planet>
<span class="life-bearing cloudy">Earth</planet>
```

为了把class属性中有barren的所有元素变为斜体，可以写作：

```
span[class~="barren"] {font-style: italic;}
```

这个规则的选择器会匹配示例标记中的前两个元素，因此这两个元素的文本将变成斜体，如图2-12所示。写作span.barren {font-style: italic;}也能得到同样的结果。

Mercury Venus Earth

图2-12：根据部分属性值选择元素

那么HTML中为什么还要有这种“~=”属性选择器呢？因为它能用于任何属性，而不只是class。例如，可以有一个包含大量图像的文档，其中只有一部分是图片。对此，可以使用一个基于title文本的部分值属性选择器，只选择这些图片。

```
img[title~="Figure"] {border: 1px solid gray;}
```

这个规则会选择title文本包含Figure的所有图像。因此，只要图片有诸如“Figure 4. A bald-headed elder statesman”之类的title文本，就能与这个规则匹配。由于这个原因，选择器img[title~="Figure"]还会匹配值为“How To Figure Out Who's In Charge”的title属性。没有title属性的图像或者title值中不包含“Figure”的图像都不会匹配。

还有一个更高级的CSS选择器模块，这是在CSS2完成之后发布的，其中包含更多的部分值属性选择器（或者按规范的说法，称之为“子串匹配属性选择器”）。由于这些属性选择器在很多现代浏览器中都得到了支持，包括IE7，所以我们在表2-1中对这些属性选择器做一个简单的总结。

表2-1：子串匹配属性选择器

类型	描述
[foo^="bar"]	选择foo属性值以"bar"开头的所有元素
[foo$="bar"]	选择foo属性值以"bar"结尾的所有元素
[foo*="bar"]	选择foo属性值中包含子串"bar"的所有元素

因此，给定以下规则和标记，可以得到图 2-13 所示的结果。

```
span[class*="cloud"] {font-style: italic;}
span[class^="bar"] {background: silver;}
span[class$="y"] {font-weight: bold;}

<span class="barren rocky">Mercury</span>
<span class="cloudy barren">Venus</span>
<span class="life-bearing cloudy">Earth</span>
```


图 2-13：根据属性值中的子串选择元素

这三个规则中，第一个规则与 class 属性包含子串 cloud 的 span 元素匹配，所以两个“cloudy”行星都会匹配。第二个规则与 class 属性以子串 bar 开头的 span 元素匹配，所以只有 Mercury 与之匹配，其 class 值为 barren rocky。Venus 不匹配，因为 barren 中的 bar 出现在 class 值的后面，而不是一开始就出现。最后，第三个规则与 class 属性以子串 y 结尾的 span 元素匹配，所以 Mercury 和 Earth 都会选中。很遗憾，Venus 再一次未能匹配，因为其 class 值的结尾不是 y。

可以想到，这些选择器有许多用途。举例来说，假设你希望对指向 O'Reilly Media 网站的所有链接应用特殊的样式。不必为所有这些链接指定类，再根据这个类编写样式，而只需编写以下规则：

```
a[href*="oreilly.com"] {font-weight: bold;}
```

当然，并不只限于 class 和 href 属性，在此任何属性都可用。完全可以根据 title、alt、src、id 等等属性的值或其部分值应用样式。利用以下规则，会特别突出老式表格布局中作为间隔的 GIF 图片（以及 URL 中包含串“space”的其他图像）：

```
img[src*="space"] {border: 5px solid red;}
```

警告： 在写这本书时，只有 Safari、基于 Gecko 的浏览器、Opera 和 IE7/Win 对这种子串选择器提供了支持。

特定属性选择类型

最后一类属性选择器即特定属性选择器，这个选择器不太好介绍，不过可以通过例子来说明。考虑以下规则：

```
*[lang|="en"] {color: white;}
```

这个规则会选择 lang 属性等于 en 或者以 en- 开头的所有元素。因此，以下示例标记中的前三个元素将被选中，而不会选择后两个元素：

```
<h1 lang="en">Hello!</h1>
<p lang="en-us">Greetings!</p>
<div lang="en-au">G'day!</div>
<p lang="fr">Bonjour!</p>
<h4 lang="cy-en">Jrooana!</h4>
```

一般地，[att|="val"]可以用于任何属性及其值。假设一个 HTML 文档中有一系列图片，其中每个图片的文件名都形如 *figure-1.gif* 和 *figure-3.jpg*。就可以使用以下选择器匹配所有这些图像：

```
img[src|="figure"] {border: 1px solid gray;}
```

这种属性选择器最常见的用途是匹配语言值，本章后面还会介绍。

使用文档结构

前面已经提到，CSS 的功能很强大，因为它要使用 HTML 文档的结构来确定适当的样式，并确定如何应用这些样式。还不仅如此，因为这说明使用文档结构是CSS 确定和应用样式的唯一途径。在确定以何种方式向文档应用样式时，结构还承担着更重要的角色。下面先来讨论结构，然后再介绍一些功能更强大的选择器。

理解父子关系

要理解选择器和文档之间的关系，需要再次分析文档的结构。考虑下面这个非常简单的 HTML 文档：

```
<html>
<head>
 <base href="http://www.meerkat.web/">
 <title>Meerkat Central</title>
</head>
<body>
 <h1>Meerkat <em>Central</em></h1>
 <p>
 Welcome to Meerkat <em>Central</em>, the <strong>best meerkat web site
 on <a href="inet.html">the <em>entire</em> Internet</a></strong>!</p>
 <ul>
  <li>We offer:
   <ul>
    <li><strong>Detailed information</strong> on how to adopt a meerkat</li>
    <li>Tips for living with a meerkat</li>
```

```
      <li><em>Fun</em> things to do with a meerkat, including:
       <ol>
        <li>Playing fetch</li>
        <li>Digging for food</li>
        <li>Hide and seek</li>
       </ol>
      </li>
     </ul>
    </li>
    <li>...and so much more!</li>
   </ul>
   <p>
   Questions? <a href="mailto:suricate@meerkat.web">Contact us!</a>
   </p>
  </body>
  </html>
```

CSS之所以强大，主要就在于元素之间存在父子关系。HTML文档以元素的一种层次结构为基础（实际上，大多数结构化文档都是如此），可以从文档的“树”视图了解这种层次结构（见图2-14）。在这个层次结构中，每个元素在整个文档结构中都有自己的一个位置。文档中的每个元素要么是另一个元素的父元素，要么是另一个元素的子元素，而且通常兼而有之（即同时作为一个元素的父元素，又作为另一个元素的子元素）。

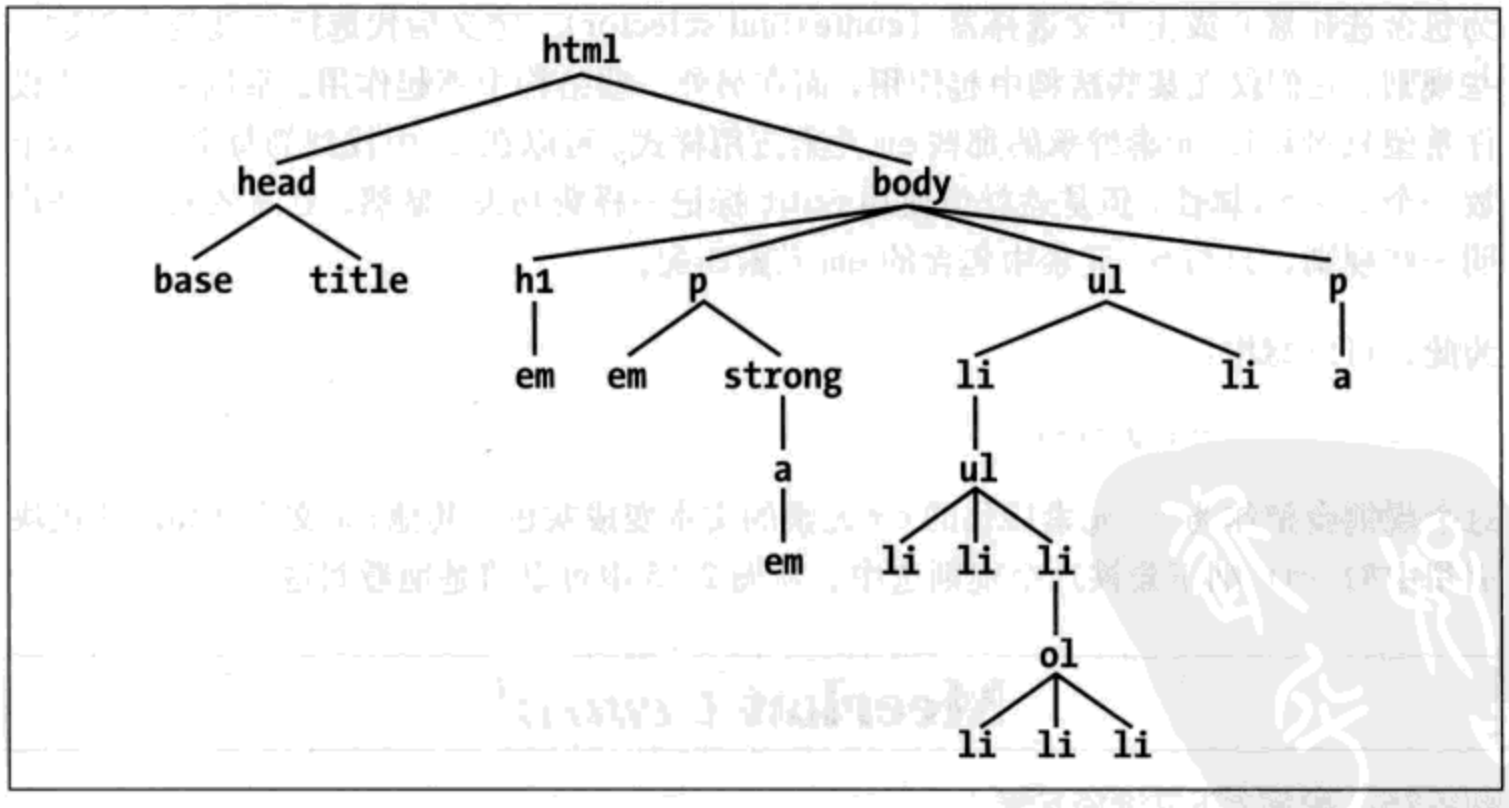

图2-14：文档树结构

如果一个元素出现在文档层次结构中另一个元素的上一层，则称前者是后者的父元素。例如，在图2-14中，第一个p元素是em和strong元素的父元素，而strong是一个anchor（a）元素的父元素，anchor元素自己又是另一个em元素的父元素。反之，如果

一个元素出现在另一个元素的下一层，则称前者是后者的子元素。因此，在图2-14中，上述anchor元素就是strong元素的子元素，而strong元素则是p元素的子元素，依此类推。

父子关系是祖先-后代关系的特例。这二者之间有一个区别：在树视图中，如果一个元素在另一个元素的直接上一层，它们就有父子关系。如果从一个元素到另一个元素的路径上要经过两层或多层，这些元素则有祖先-后代关系，而不是父子关系（当然，子元素也算是后代，父元素也算是祖先）。在图2-14中，第一个ul元素是两个li元素的父元素，但是另外第一个ul元素还是其li元素所有后代元素的祖先（一直到嵌套最深的li元素）。

同样地，在图2-14中，还有一个anchor元素，这是strong的子元素，它也是paragraph(p)、body和html元素的后代。body元素是浏览器默认显示的所有元素的祖先，html元素则是整个文档的祖先。出于这个原因，html元素也被称为根元素。

后代选择器

理解这个结构模型后，第一个好处是可以定义后代选择器（descendant selector，也称为包含选择器）或上下文选择器（contextual selector）。定义后代选择器就是来创建一些规则，它们仅在某些结构中起作用，而在另外一些结构中不起作用。举例来说，假设你希望只对从h1元素继承的那些em元素应用样式。可以在h1中找到的每个em元素上放一个class属性，但是这就像使用font标记一样费功夫。显然，更高效的做法是声明一些规则，只与h1元素中包含的em元素匹配。

为此，可以写作：

```
h1 em {color: gray;}
```

这个规则会把作为h1元素后代的em元素的文本变成灰色。其他em文本（如段落或块引用中的em）则不会被这个规则选中。从图2-15中可以清楚地看到这一点。

图2-15：根据上下文选择元素

在一个后代选择器中，规则左边的选择器一端包括两个或多个用空格分隔的选择器。选择器之间的空格是一种结合符（combinator）。每个空格结合符可以解释为“……在……中找到”、“……作为……的一部分”，或“……作为……的后代”，但是要求必须从右向

左读选择器。因此，h1 em可以解释为“作为h1元素后代的任何em元素”（译注1）。（如果要从左向右读选择器，可以换成以下说法：“包含一个em的所有h1会把以下样式应用到该em”。）

当然并不仅限于两个选择器，例如：

```
ul ol ul em {color: gray;}
```

如图2-16所示，在这种情况下，会把一个无序列表中的强调文本置为灰色，这个无序列表是一个有序列表的一部分，而这个有序列表本身又是另一个无序列表的一部分（尽管很复杂，不过这确实没有错）。显然这是一个很特定的选择原则。

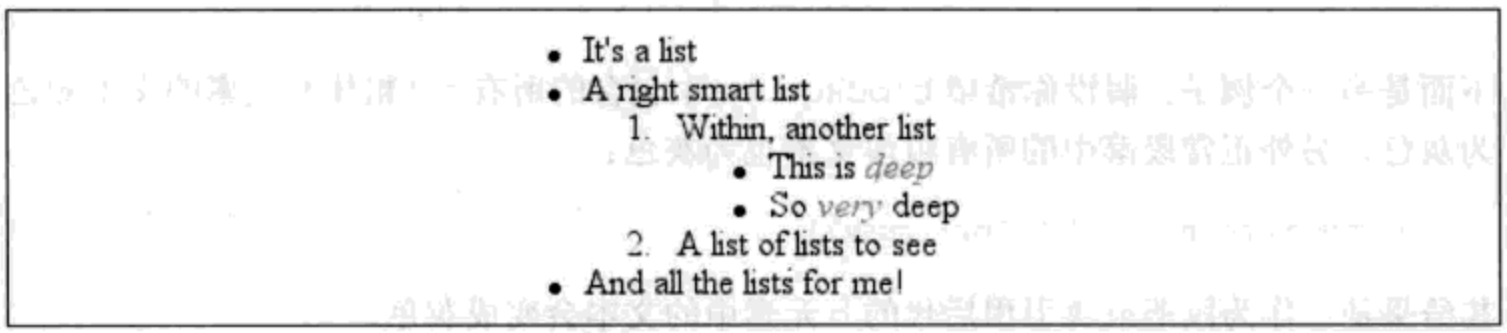

图2-16：一个非常特定的后代选择器

后代选择器功能极其强大。有了这些后代选择器，使得HTML中不可能实现的任务（至少是如果不大量使用font标记就无法做到的事情）成为可能。下面来看一个常见的例子。假设有一个文档，其中有一个边栏，还有一个主区。边栏的背景为蓝色，主区背景为白色，这两个区都包含链接列表。不能把所有这些链接都设置为蓝色，因为这样一来边栏中的蓝色链接将无法看到。

解决方法是使用后代选择器。在这种情况下，可以为包含边栏的表单元格指定一个值为sidebar的class属性，并把主区的class属性值设置为main。然后编写以下样式：

```
td.sidebar {background: blue;}
td.main {background: white;}
td.sidebar a:link {color: white;}
td.main a:link {color: blue;}
```

图2-17显示了这些样式的结果。

译注1：按作者的说法，从右向左读意味着h1 em读作“em元素作为h1元素的后代”，但是为了在中文中更为通顺，我们将语序做了调整。

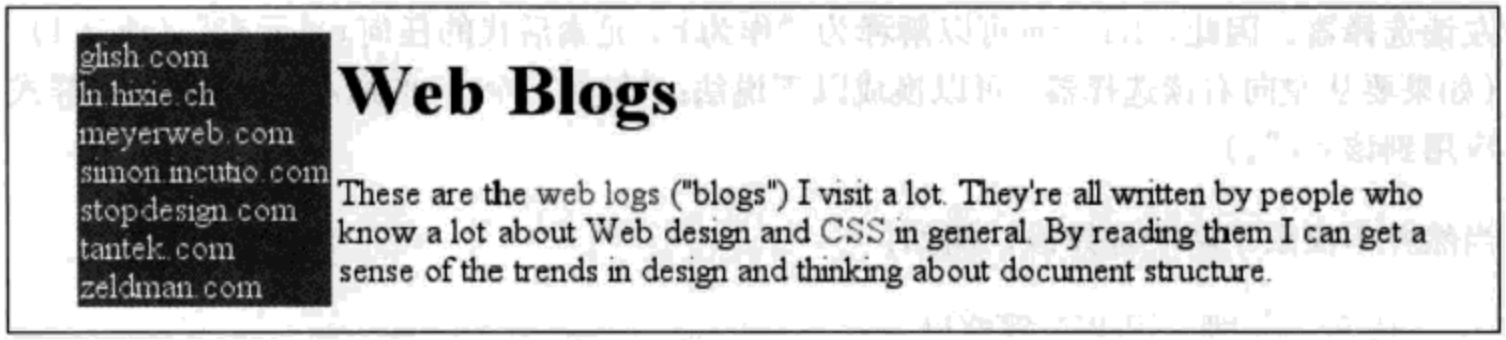

图2-17：使用后代选择器向同类元素应用不同的样式

注意： :link 指示尚未访问的资源的链接。本章后面将详细讨论这个内容。

下面是另一个例子：假设你希望 blockquote 中包含的所有 b（粗体）元素的文本颜色为灰色，另外正常段落中的所有粗体文本也为灰色：

```
blockquote b, p b {color: gray;}
```

其结果是，作为段落或块引用后代的 b 元素中的文本会变成灰色。

关于后代选择器有一个常被忽视的方面，即两个元素之间的层次间隔可以是无限的。例如，如果写作 ul em，这种语法就会选择从 ul 元素继承的所有 em 元素，而不论 em 的嵌套层次多深。因此，ul em 将会选择以下标记中的 em 元素：

```
<ul>
<li>List item 1
<ol>
<li>List item 1-1</li>
<li>List item 1-2</li>
<li>List item 1-3
<ol>
<li>List item 1-3-1</li>
<li>List item <em>1-3-2</em></li>
<li>List item 1-3-3</li>
</ol></li>
<li>List item 1-4</li>
</ol></li>
</ul>
```

选择子元素

在某些情况下，可能并不想选择一个任意的后代元素；而是希望缩小范围，只选择另一个元素的子元素。例如，你可能想选择只作为一个 h1 元素子元素（而不是后代元素）的 strong 元素。为此，可以使用子结合符，即大于号（>）：

```
h1 > strong {color: red;}
```

这个规则会把第一个h1下面出现的strong元素变成红色，但是第二个出现的strong元素不受影响。

```
<h1>This is <strong>very</strong> important.</h1>
<h1>This is <em>really <strong>very</strong></em> important.</h1>
```

如果从右向左读（译注2），选择器h1 > strong可以解释为“选择作为h1元素子元素的所有strong元素”。子结合符两边可以有空白符，这是可选的。因此，h1 > strong、h1> strong和h1>strong都是一样的。只要你愿意，可以使用空白符，也可以忽略空白符。

查看文档的树结构时，可以很容易地看到，子选择器限制为只匹配树中直接相连的元素。图2-18显示了一个文档树的一部分。

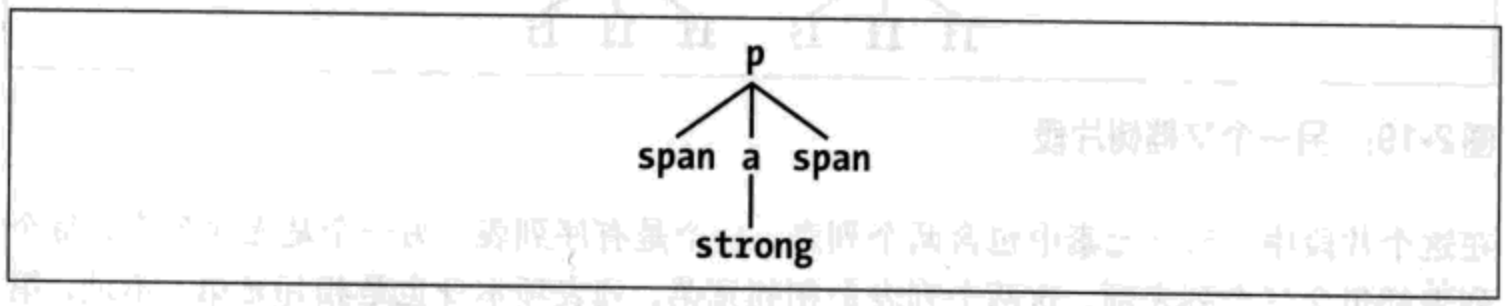

图2-18：一个文档树片段

在这个树片段中，可以很容易地看出父子关系。例如，a元素是strong元素的父元素，但是它又是p元素的子元素。可以用p > a和a > strong选择器来匹配这个片段中的元素，但是p > strong不行，因为strong是p的后代而不是其子元素。

还可以在同一个选择器中结合使用后代选择器和子选择器。因此，table.summary td > p会选择作为一个td元素子元素的所有p元素，这个td元素本身从table元素继承，该table元素有一个包含summary的class属性。

选择相邻兄弟元素

假设你希望对一个标题后紧接着的段落应用样式，或者向一个段落后紧接着的列表指定特殊的外边距。要选择紧接在另一个元素后的元素，而且二者有相同的父元素，可以使用相邻兄弟结合符（adjacentsibling combinator），这表示为一个加号（+）。与子结合符一样，相邻兄弟结合符旁边可以有空白符，这要看创作人员的喜好。

译注2：按作者的说法，从右向左读意味着h1 > strong读作“strong元素作为h1元素的子元素”，但是为了在中文中更为通顺，我们将语序作了调整。

要去除紧接在一个h1元素后出现的段落的上边距，可以写作：

```
h1 + p {margin-top: 0;}
```

这个选择器读作“选择紧接在一个h1元素后出现的所有段落，h1要与p元素有共同的父元素”。

为了形象地展示这个选择器是如何工作的，最容易的办法是再来考虑一个文档树的片段，如图2-19所示。

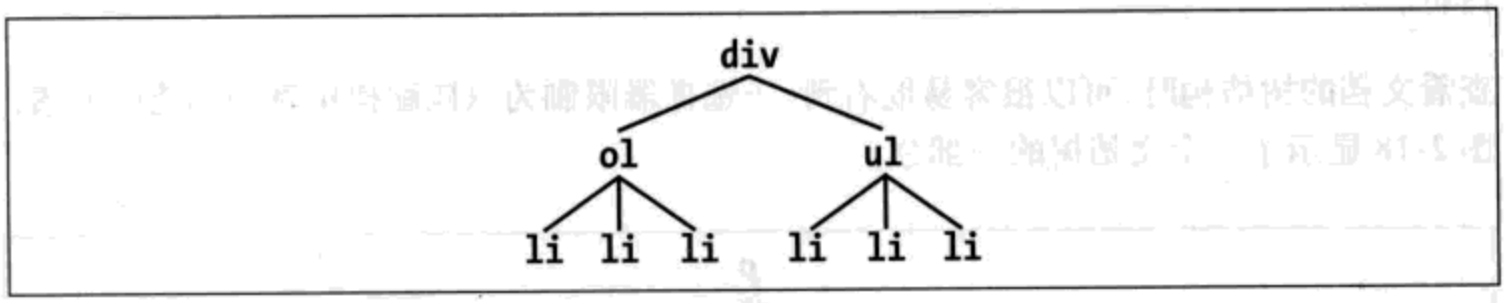

图2-19：另一个文档树片段

在这个片段中，div元素中包含两个列表，一个是有序列表，另一个是无序列表，每个列表都包含三个列表项。这两个列表是相邻兄弟，列表项本身也是相邻兄弟。不过，第一个列表中的列表项不是第二个列表中列表项的兄弟，因为这两组列表项的父元素不同（最多只能算堂兄弟）。

要记住，用一个结合符只能选择两个相邻兄弟中的第二个元素。因此，如果写作li+li {font-weight: bold;}，只会把各列表中的第二个和第三个列表项变成粗体。第一个列表项将不受影响，如图2-20所示。

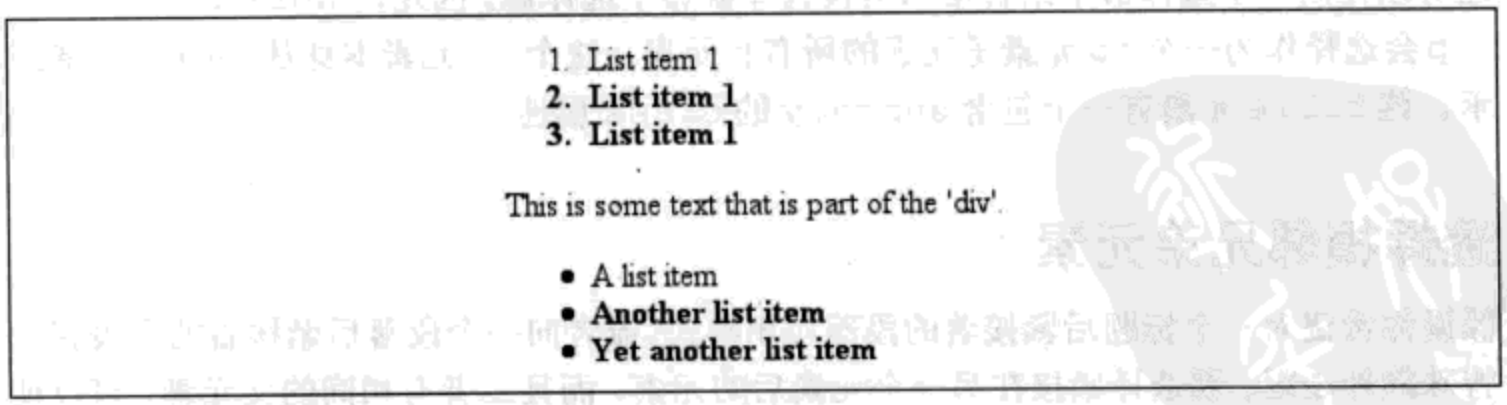

图2-20：选择相邻兄弟

要想正常地工作，CSS要求两个元素按“源顺序”出现。在前面的例子中，ol元素后面有一个ul元素。因此可以用ol+ul选择第二个元素，但是用这个语法无法选择第一个元素。要想与ul+ol匹配，有序列表必须紧跟在无序列表后面。

另外，两个元素之间的文本内容不会影响相邻兄弟结合符起作用。考虑以下标记片段，其树视图与图 2-19 相同：

```
<div>
<ol>
<li>List item 1</li>
<li>List item 1</li>
<li>List item 1</li>
</ol> This is some text that is part of the 'div'.
<ul>
<li>A list item</li>
<li>Another list item</li>
<li>Yet another list item</li>
</ul>
</div>
```

尽管两个列表间多了一行文本，不过还是可以用选择器ol+ul来匹配第二个列表。这是因为，中间的文本并不包含在兄弟元素中，而只是父元素div的一部分。如果将这个文

赝醋 l+p+ul。

如以下示例所示，相邻兄弟结合符还可以结合其他结合符：

```
html > body table + ul{margin-top: 1.5em;}
```

这个选择器解释为“选择紧接在一个table元素后出现的所有兄弟ul元素，该table元素包含在一个body元素中，body元素本身是html元素的子元素”。

警告： Windows 平台的 Internet Explorer 在 IE6 之前不支持子选择器和相邻兄弟选择器。IE7 对二者则提供了支持。

伪类和伪元素

伪类选择器和伪元素选择器才真正有些意思。利用这些选择器，可以为文档中不一定具体存在的结构指定样式，或者为某些元素（甚至是文档本身）的状态所指示的幻像类指定样式。换句话说，会根据另外某种条件而非文档结构向文档中的某些部分应用样式，而且无法通过研究文档的标记准确地推断出采用何种方式应用样式。

听上去好像是在随机地应用样式，不过并非如此。实际上，这里是根据某种无法提前预测的暂时条件来应用样式。不过，什么情况下将出现样式实际上是明确定义的。可以这样想：在一个赛事中，只要主场球队得分，观众就会欢呼。在比赛中你并不知道球队什

么时候会得分，但是一旦得分，不出所料，观众就会大声欢呼。尽管无法预测哪个时刻出现前因，但并不影响对后果的预测。

伪类选择器

先来分析伪类选择器，因为浏览器对伪类选择器提供了更好的支持，相应地使用更广泛。

我们来考虑锚元素（a），在HTML和XHTML中，锚元素会建立从一个文档到另一个文档的链接。这样描述锚固然不错，不过有些锚指示的是已经访问过的页面，而另外一些则指示尚未访问的页面。只是看HTML标记是无法区别二者的差别的，因为在标记中所有锚看上去都一样。要想区别哪些链接已经访问过，唯一的办法就是将文档中的链接与用户的浏览历史相比较。所以，实际上有两种基本的锚类型：已访问的和未访问的。这些类型称为伪类（pseudo-class），使用这些伪类的选择器则称为伪类选择器（pseudo-class selector）。

为了更好地理解这些类和选择器，下面来考虑浏览器如何处理链接。Mosaic协定指定未访问页面的链接为蓝色，已访问页面的链接为红色（在以前的浏览器中，如Internet Explorer，已访问链接可能不是红色而是紫色）。所以，如果可以向锚中插入类，使已经访问过的锚有一个类（如“visited”），就可以写如下样式来使这些锚变成红色：

```
a.visited {color: red;}
<a href="http://www.w3.org/" class="visited">W3C Web site</a>
```

不过，这种方法要求每次访问一个新页面时都要修改锚的类，这种做法有些傻。与此不同，CSS定义了伪类，使已访问页面的锚就好像已经有一个“visited”类一样：

```
a:visited {color: red;}
```

现在，指向已访问页面的锚都会是红色，甚至你不必为任何锚增加class属性。注意规则中的冒号（:）。分隔a和visited的冒号是伪类或伪元素的“名片”。所有伪类和伪元素关键字前面都有一个冒号。

链接伪类

CSS2.1定义了两个只应用于超链接的伪类。在HTML和XHTML 1.0及1.1中，超链接是有href属性的所有a元素；在XML语言中，超链接则可以是任何元素，只要它作为另一个资源的链接。表2-2描述了这两个伪类。

表2-2：链接伪类

伪类名	描述
:link	指示作为超链接（即有一个href属性）并指向一个未访问地址的所有锚。注意，有些浏览器可能会不正确地将:link解释为指向任何超链接，包括已访问和未访问的超链接
:visited	指示作为已访问地址超链接的所有锚

表2-2中的第一个伪类看上去有些多余。毕竟，如果一个锚尚未访问过，那它肯定是未访问的链接，不是吗？如果是这样，我们所需要的应该只是：

```
a {color: blue;}
a:visited {color: red;}
```

尽管这种格式看上去是合理的，但这确实还不够。以上规则中，第一个规则不仅应用于未访问的链接，还会应用到以下锚（译注3）：

```
<a name="section4">4. The Lives of Meerkats</a>
```

相应的文本就会变成蓝色，因为如上所示，a元素与规则a {color: blue;}匹配。因此，为了避免将链接样式应用到目标锚，要使用:link伪类：

```
a:link {color: blue;}     /* unvisited links are blue */
a:visited {color: red;}   /* visited links are red */
```

你可能已经意识到了，:link和:visited伪类选择器在功能上与body属性link和vlink是等价的。假设一个创作人员希望所有未访问页面的锚都是紫色，而所有已访问页面的锚是银色。在HTML 3.2中，这要如下指定：

```
<body link="purple" vlink="silver">
```

在CSS中，利用以下规则可以达到同样的效果：

```
a:link {color: purple;}
a:visited {color: silver;}
```

当然，对于CSS伪类，不仅可以应用颜色，还可以应用更多样式。假设你希望已访问链接为斜体，而且除了银色外还有一条贯穿线，如图2-21所示。

译注3：即非链接，因为它没有href属性。

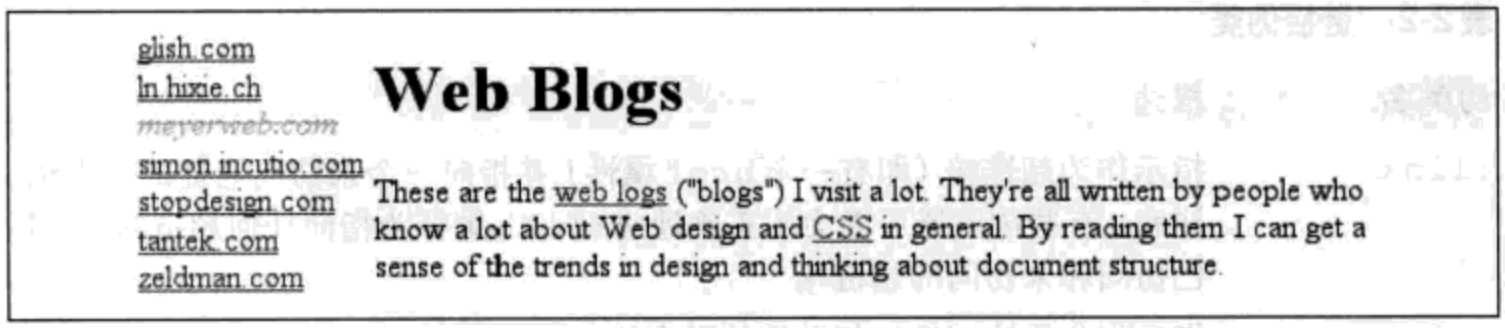

图2-21：为已访问链接应用多个样式

只需以下样式就可以达到目的：

```
a:visited {color: silver; text-decoration: line-through; font-style: italic;}
```

现在有必要再来回顾类选择器，并说明类选择器如何与伪类结合。例如，假设你希望指向网站外资源的链接改变颜色。如果为每一个这样的锚指定一个类，就很容易做到：

```
<a href="http://www.mysite.net/">My home page</a>
<a href="http://www.site.net/" class="external">Another home page</a>
```

要为外部链接应用不同的样式，只需如下的一条规则：

```
a.external:link, a.external:visited {color: maroon;}
```

这个规则会使以上标记中的第二个锚变成紫红色，而第一个锚仍保持为超链接的默认颜色（通常是蓝色）。

这个一般语法也适用于ID选择器：

```
a#footer-copyright:link{font-weight: bold;}
a#footer-copyright:visited {font-weight: normal;}
```

尽管:link和:visited非常有用，但它们是静态的——第一次显示之后，它们一般不会再改变文档的样式。CSS2.1中还有一些没有这种静态性的伪类，接下来将详细说明。

动态伪类

CSS2.1定义了3个动态伪类，它们可以根据用户行为改变文档的外观。这些动态伪类以前总用来设置超链接的样式，不过它们还有很多其他用途。表2-3描述了这些伪类。

表2-3：动态伪类

伪类名	描述
:focus	指示当前拥有输入焦点的元素，也就是说，可以接受键盘输入或者能以某种方式激活的元素

表2-3：动态伪类（续）

伪类名	描述
:hover	指示鼠标指针停留在哪个元素上，例如，鼠标指针可能停留在一个超链接上，:hover就会指示这个超链接
:active	指示被用户输入激活的元素，例如，鼠标指针停留在一个超链接上时，如果用户点击鼠标，就会激活这个超链接，:active将指示这个超链接

类似于:link和:visited，这些伪类最常用于超链接的上下文中。很多Web页面都有类似的样式：

```
a:link {color: navy;}
a:visited {color: gray;}
a:hover {color: red;}
a:active {color: yellow;}
```

前两个规则使用了静态伪类，后两个利用了动态伪类。:active类似于HTML 3.2中的alink属性，不过，如前所述，对激活链接不仅能改变颜色，还可以应用你喜欢的任何样式。

注意： 伪类的顺序很重要，这一点最初可能不太明显。通常的建议是“link-visited-hover-active”，不过现在已经改为“link-visited-focus-hover-active”。下一章将解释为什么这种顺序很重要，并讨论在哪些情况下可能要改变甚至忽略这个推荐顺序。

注意，动态伪类可以应用到任何元素，这一点很好，因为对非链接的元素应用动态样式通常很有用。例如，使用以下标记：

```
input:focus {background: silver; font-weight: bold;}
```

可以突出显示一个准备接受键盘输入的表单元素，如图2-22所示。

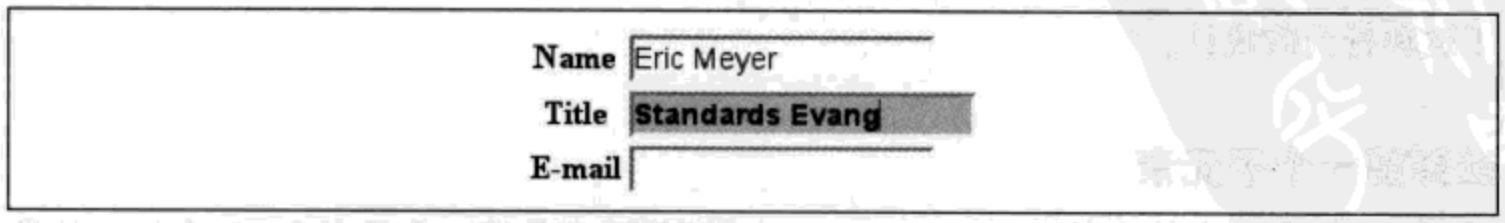

图2-22：突出显示一个有输入焦点的表单元素

向任意元素应用动态伪类还可以做一些有点奇怪的事情。你可能想通过以下规则为用户提供一种“强调”的效果：

```
body *:hover {background: yellow;}
```

根据这个规则，从body元素继承的所有元素（即包含在body下的元素）在鼠标指针停留时（处于悬停状态时）会显示一个黄色背景。标题、段落、列表、表、图像和body中的所有元素都会改为有黄色背景。另外还可以改变字体，在鼠标停留的元素外加一个边框，或者改变浏览器允许的所有其他方面。

警告： Windows平台的Internet Explorer在IE6之前只允许动态伪类选择超链接，而不允许选择其他元素。IE7支持对所有元素都能应用:hover，但是不支持对表单元素应用:focus样式。

动态样式的实际问题

动态伪类带来了一些有意思的怪问题。例如，可以将已访问和未访问的链接设置为一种字体大小，而让鼠标停留的链接有更大的字体，如图2-23所示：

```
a:link, a:visited {font-size: 13px;}
a:hover {font-size: 20px;}
```

图2-23：使用动态伪类改变布局

可以看到，鼠标指针停留在锚上时用户代理增加了锚的大小。支持这种行为的用户代理在锚处于悬停状态时必须重绘文档，这就要求重新显示该链接之后的所有内容。

不过，CSS规范指出，文档第一次显示之后，用户代理不必重绘文档，所以你不能完全依赖预想的效果，也就是说，不要指望你预想的效果肯定会发生。强烈建议要避免依赖于这种行为的设计。

选择第一个子元素

还可以使用另一个静态伪类:first-child来选择元素的第一个子元素。这个特定伪类很容易遭到误解，所以有必要举个例子来说明。考虑以下标记：

```
<div>
<p>These are the necessary steps:</p>
<ul>
<li>Insert key</li>
```

```
<li>Turn key <strong>clockwise</strong></li>
<li>Push accelerator</li>
</ul>
<p> Do <em>not</em> push the brake at the same time as the accelerator.
</p>
</div>
```

在这个例子中，作为第一个子元素的元素包括第一个p、第一个li和strong及em元素。给定以下两个规则：

```
p:first-child {font-weight: bold;}
li:first-child {text-transform: uppercase;}
```

可以得到图2-24所示的结果。

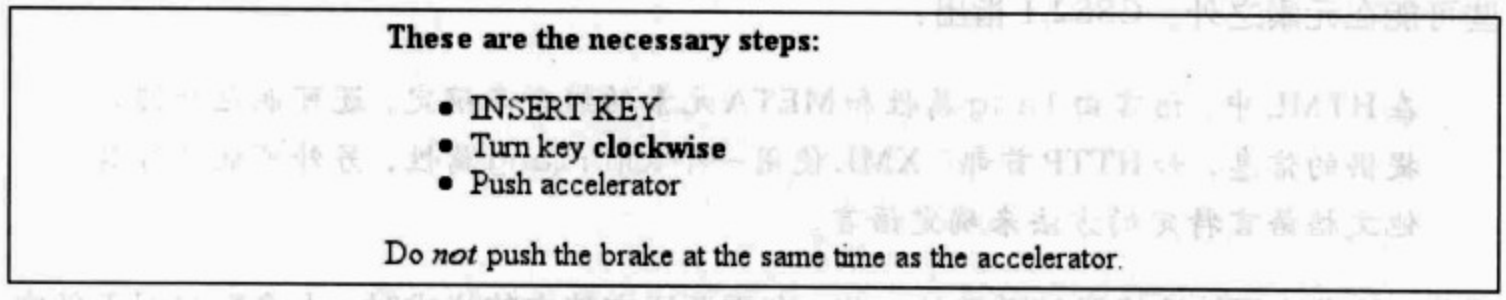

图2-24：对第一个子元素应用样式

前一个规则将作为某元素第一个子元素的所有p元素设置为粗体。第二个规则将作为某个元素（在HTML中，这肯定是一个ol或ul元素）第一个子元素的所有li元素变成大写。

最常见的错误是认为p:first-child之类的选择器会选择p元素的第一个子元素。不过，要记住伪类的实质，它是把某种幻像类关联到与伪类相关的元素。如果要向标记增加具体类，可能如下所示：

```
<div>
<p class="first-child">These are the necessary steps:</p>
<ul>
<li class="first-child">Insert key</li>
<li>Turn key <strong class="first-child">clockwise</strong></li>
<li>Push accelerator</li>
</ul>
<p> Do <em class="first-child">not</em> push the brake at the same time as the
accelerator.
</p>
</div>
```

因此，如果想选择作为某元素第一个子元素的em元素，应当写作em:first-child。这个选择器才能真正达到你的目的，例如，对列表中的第一项、div中的第一段或表行中的第一个td应用样式。

警告： Windows 平台的 Internet Explorer 在 IE6 之前不支持 :first-child，不过 IE7 支持。

根据语言选择

有些情况下，你可能想根据元素的语言来选择，此时可以使用 :lang( ) 伪类。从对应的模式来讲，:lang( ) 伪类就像是 |= 属性选择器。例如，要把所有法语元素变成斜体，可以写作：

```
*:lang(fr) {font-style: italic;}
```

伪选择器和属性选择器之间的主要差别在于语言信息可以从很多来源得到，而且其中一些可能在元素之外。CSS2.1 指出：

> 在 HTML 中，语言由 lang 属性和 META 元素的组合来确定，还可能包括协议提供的信息，如 HTTP 首部。XML 使用一个 xml:lang 属性，另外可能还有其他文档语言特定的方法来确定语言。

因此，伪类比属性选择器稍微健壮一些，在需要语言特定的样式时，大多数情况下伪类都是更好的选择。

结合伪类

在 CSS2.1 中，可以在同一个选择器中结合使用伪类。例如，鼠标指针停留在未访问链接上时，可以让这些链接变成红色，而鼠标指针停留在已访问链接上时，链接变成紫红色。

```
a:link:hover {color: red;}
a:visited:hover {color: maroon;}
```

用哪种顺序指定并不重要；写成 a:hover:link 会得到与 a:link:hover 一样的效果。还可以为另一种语言的未访问和已访问链接指定不同的悬停样式，例如可以如下指定德语元素的样式：

```
a:link:hover:lang(de) {color: gray;}
a:visited:hover:lang(de) {color: silver;}
```

要当心，不要把互斥的伪类结合在一起使用。例如，一个链接不能同时是已访问和未访问的，所以 a:link:visited 没有任何意义。用户代理往往会忽略这种选择器，相应地忽略整个规则，不过这一点不能保证，因为不同的浏览器可能有不同的错误处理行为。

警告： Windows平台的Internet Explorer在IE6之前不能正确地识别这种组合伪类。与类－值组合类似，它只会注意到组合伪类中的最后一个伪类。因此，对于a:link:hover，较老版本的IE/Win只会注意:hover而不会去管选择器中的:link部分。IE7不存在这种限制，它会正确地处理这些组合的伪类。

伪元素选择器

就像伪类为锚指定幻像类一样，伪元素能够在文档中插入假想的元素，从而得到某种效果。CSS2.1中定义了4个伪元素：设置首字母样式、设置第一行样式、设置之前和之后元素的样式。

设置首字母样式

第一个伪元素用于设置一个块级元素首字母的样式，而且仅对该首字母设置样式：

```
p:first-letter {color: red;}
```

这个规则会把每一段的第一个字母变成红色。或者，如果让每个h2中第一个字母的大小是标题中其余字母大小的两倍：

```
h2:first-letter {font-size: 200%;}
```

这个规则的结果如图2-25所示。

This is an h2 element

图2-25：实际应用的:first-letter伪元素

前面提到过，这个规则会导致用户代理对一个假想的元素作出响应，这个假想元素包含每个h2中的第一个字母。可能如下所示：

```
<h2><h2:first-letter>T</h2:first-letter>his is an h2 element</h2>
```

:first-letter样式只应用到上例所示假想元素的内容。这个<h2:first-letter>元素并不出现在文档源代码中。相反，它是由用户代理动态构造的，用于向相应文本块应用:first-letter样式。换句话说，<h2:first-letter>是一个伪元素。要记住，不必增加任何新标记。这会由用户代理完成。

设置第一行的样式

类似地，:first-line可以用来影响元素中第一个文本行。例如，可以让一个文档中第一段的第一行变成紫色：

```
p:first-line {color: purple;}
```

在图2-26中，这个样式应用于每一段所显示的第一行文本。不论显示区域多大或多小，都是如此。如果第一行只包含该段的5个词，那么只有这5个词会变成紫色。如果第一行包含了元素的前30个词，那么所有这30个词都会是紫色。

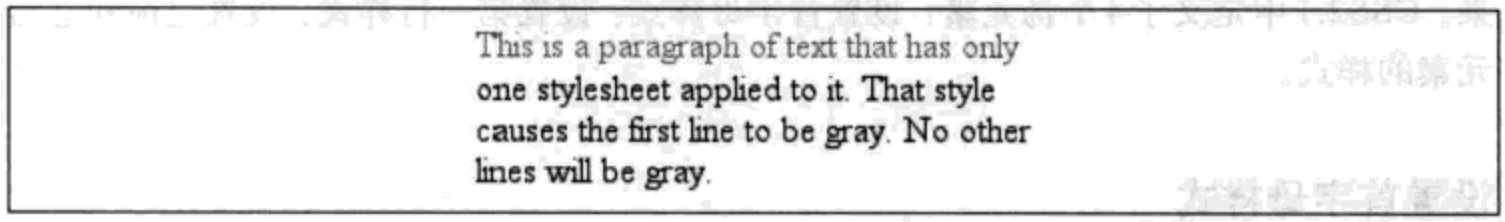

图2-26：实际应用的:first-line伪元素

由于从“This”到“only”的文本都应当是紫色，所以用户代理使用了一个假想标记，如下所示：

```
<p><p:first-line>This is a paragraph of text that has only</p:first-line>
one stylesheet applied to it. That style
causes the first line to be purple. No other ...
```

如果第一行文本编辑为只包含这一段的前7个词，那么假想的</p:first-line>会前移，出现在“that”后面。

:first-letter 和 :first-line 的限制

在CSS2中，:first-letter和:first-line伪元素只能应用于标记或段落之类的块级元素，而不能应用于超链接等的行内元素。在CSS2.1中，:first-letter能应用到所有元素。不过能应用:first-line和:first-letter的CSS属性还是有一些限制。表2-4显示了这些限制。

表2-4：伪元素所允许的属性

:first-letter	:first-line
所有font属性	所有font属性
color	color
所有background属性	所有background属性
所有margin属性	word-spacing

表2-4：伪元素所允许的属性（续）

:first-letter	:first-line
所有 padding 属性	letter-spacing
所有 border 属性	text-decoration
text-decoration	vertical-align
vertical-align（如果 float 设置为 none）	text-transform
text-transform	line-height
line-height	clear（仅适用于 CSS2，CSS2.1 已去除）
float	text-shadow（仅适用于 CSS2）
letter-spacing （CSS2.1 中新增）	
word-spacing （CSS2.1 中新增）	
clear（仅适用于 CSS2，CSS2.1 已去除）	
text-shadow（仅适用于 CSS2）	

另外，所有伪元素都必须放在出现该伪元素的选择器的最后面。因此，如果写成 p:first-line em 就是不合法的，因为伪元素在选择器主体前面出现（主体是这里所列的最后一个元素，即 em）。这个规则同样适用于 CSS2.1 定义的另外两个伪元素。

设置之前和之后元素的样式

假设想设置一种排版效果，在每个 h2 元素前加一对银色中括号：

```
h2:before {content: "}}"; color: silver;}
```

CSS2.1 允许插入生成的内容，然后使用伪元素 :before 和 :after 直接设置样式。图 2-27 给出了一个例子。

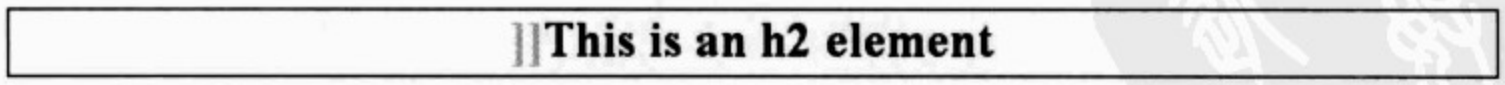

图2-27：在元素之前插入内容

伪元素用于插入生成的内容，并设置其样式。要在一个元素后面插入内容，可以使用伪元素 :after。可以在文档的最后用一个适当的结束语结束：

```
body:after {content: "  The End.";}
```

所生成的内容是一个单独的主体，有关内容（包括 :before 和 :after 的更多信息）将在第 12 章更全面地介绍。

小结

通过根据文档的语言来使用选择器，创作人员可以创建丰富的CSS规则。我们可以构建只应用于少数元素的简单规则，对大量类似元素应用样式也同样简单。由于可以对选择器和规则分组，这使得样式表相当简洁，而且非常灵活，相应地可以缩小文件的大小，缩短下载时间。

用户代理通常必须慎用选择器，因为如果不能正确地解释选择器，会导致用户代理根本无法使用CSS。另一方面，创作人员要正确地编写选择器，这很关键，因为一旦有错误，用户代理将不能按预想的那样应用样式。要想正确地理解选择器以及如何组合选择器，需要深入地掌握选择器与文档结构的关系，并了解继承和层叠等机制在确定如何为元素设置样式时有怎样的作用。这正是下一章要讨论的内容。

第3章

结构和层叠

第2章介绍了如何利用文档结构和CSS选择器为元素应用各种丰富的样式。每个合法的文档都会生成一个结构树，了解了这一点，就能根据元素的祖先、属性、兄弟元素等等创建选择器来选择元素。有了这个结构树，选择器才能起作用，这也是CSS另一个重要方面（即继承）的核心。

继承（Inheritance）是从一个元素向其后代元素传递属性值所采用的机制。确定应当向一个元素应用哪些值时，用户代理不仅要考虑继承，还要考虑声明的特殊性，另外需要考虑声明本身的来源。这个过程就称为层叠（cascade）。本章将讨论这3种机制之间的关联：特殊性、继承和层叠。

不管怎样，无论问题看上去多抽象、多难懂，都要继续努力！你的努力不会白费。

特殊性

从第2章了解到，可以使用多种不同的方法选择元素。实际上，可能同一个元素可以使用两个或多个规则来选择，每个规则都有其自己的选择器。下面考虑以下3对规则。假设每一对规则都匹配同样的元素：

```
h1 {color: red;}
body h1 {color: green;}

h2.grape {color: purple;}
h2 {color: silver;}

html > body table tr[id="totals"] td ul > li {color: maroon;}
```

```
li#answer {color: navy;}
```

显然，每一对规则中只有一个胜出，因为所匹配的元素只能是某一种颜色（或此或彼）。那么怎么知道哪一个规则更强呢？

答案就在于每个选择器的特殊性（specificity）。对于每个规则，用户代理会计算选择器的特殊性，并将这个特殊性附加到规则中的各个声明。如果一个元素有两个或多个冲突的属性声明，那么有最高特殊性的声明就会胜出。

注意：这并不是解决冲突的全部。实际上，所有样式冲突的解决都由层叠来处理，本章后面专设了一节介绍这个内容。

选择器的特殊性由选择器本身的组件确定。特殊性值表述为4个部分，如：0,0,0,0。一个选择器的具体特殊性如下确定：

- 对于选择器中给定的各个ID属性值，加0,1,0,0。
- 对于选择器中给定的各个类属性值、属性选择或伪类，加0,0,1,0。
- 对于选择器中给定的各个元素和伪元素，加0,0,0,1。伪元素是否有特殊性？在这方面CSS2有些自相矛盾，不过CSS2.1很清楚地指出，伪元素有特殊性，而且特殊性为0,0,0,1。
- 结合符和通配选择器对特殊性没有任何贡献（后面还会更多地介绍这些值）。

例如，以下规则中选择器的特殊性见注释：

```
h1 {color: red;}  /* specificity = 0,0,0,1 */
p em {color: purple;}  /* specificity = 0,0,0,2 */
.grape {color: purple;}  /* specificity = 0,0,1,0 */
*.bright {color: yellow;}  /* specificity = 0,0,1,0 */
p.bright em.dark {color: maroon;}  /* specificity = 0,0,2,2 */
#id216 {color: blue;}  /* specificity = 0,1,0,0 */
div#sidebar *[href] {color: silver;}  /* specificity = 0,1,1,1 */
```

假设有以下情况，一个em元素既与上例中的第2条规则匹配，又与第5条规则匹配，这个元素将是紫红色，因为第5条规则的特殊性高于第2条规则的特殊性。

下面做个练习，回顾本节前面给出的几组规则，看看它们有怎样的特殊性：

```
h1 {color: red;}  /* 0,0,0,1 */
body h1 {color: green;}  /* 0,0,0,2  (winner)*/

h2.grape {color: purple;}  /* 0,0,1,1 (winner) */
h2 {color: silver;}  /* 0,0,0,1 */
```

```
html > body table tr[id="totals"] td ul > li {color: maroon;}  /* 0,0,1,7 */
li#answer {color: navy;}  /* 0,1,0,1  (winner) */
```

上面已经指出每组规则中的胜出规则；在上述各种情况下，那些规则之所以胜出是因为其特殊性更高。要注意特殊性是如何排序的。在第二组中，选择器 h2.grape 能“赢”是因为它多了一个 1：0,0,1,1 大于 0,0,0,1。在第三组中，第二个规则胜出是因为 0,1,0,1 大于 0,0,1,7。实际上，特殊性值 0,0,1,0 比值 0,0,0,13 更高。

之所以会这样，是因为值是从左向右排序的。特殊性值 1,0,0,0 大于以 0 开头的所有特殊性值，而不论后面的数是什么。所以 0,1,0,1 比 0,0,1,7 高，因为前一个值中第二位上的 1 大于第二个值中第二位上的 0。

声明和特殊性

一旦确定一个选择器的特殊性，这个值将授予其所有相关声明。考虑以下规则：

```
h1 {color: silver; background: black;}
```

由于特殊性的缘故，用户代理必须相应地处理这个规则，将其“解组”为单独的规则。因此，前面的例子将变成：

```
h1 {color: silver;}
h1 {background: black;}
```

这两个规则的特殊性都是 0,0,0,1，各声明得到的特殊性值也就是 0,0,0,1。分组选择器也同样会完成这种分解过程。给定以下规则：

```
h1, h2.section {color: silver; background: black;}
```

用户代理将把它处理为：

```
h1 {color: silver;}  /* 0,0,0,1 */
h1 {background: black;}  /* 0,0,0,1 */
h2.section {color: silver;}  /* 0,0,1,1 */
h2.section {background: black;}  /* 0,0,1,1 */
```

如果多个规则与同一个元素匹配，而且有些声明相互冲突，在这种情况下特殊性就很重要。例如，考虑以下规则：

```
h1 + p {color: black; font-style: italic;}  /* 0,0,0,2 */
p {color: gray; background: white; font-style: normal;}  /* 0,0,0,1 */
*.aside {color: black; background: silver;}  /* 0,0,1,0 */
```

当这些规则应用到以下标记时，显示的内容将如图 3-1 所示：

```
<h1>Greetings!</h1>
```

```
<p class="aside"> It's a fine way to start a day, don't you think?
</p>
<p> There are many ways to greet a person, but the words are not as
important as the act of greeting itself.
</p>
<h1>Salutations!</h1>
<p> There is nothing finer than a hearty welcome from one's fellow man.
</p>
<p class="aside"> Although a thick and juicy hamburger with bacon and
mushrooms runs a close second.
</p>
```

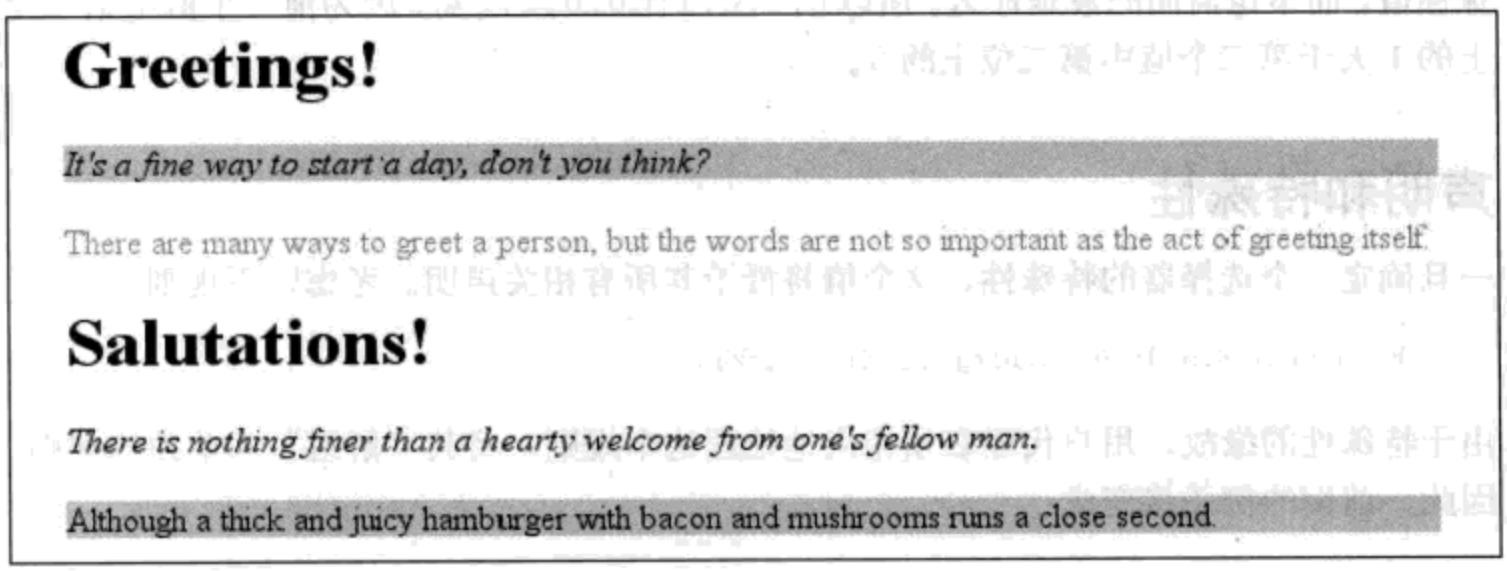

图3-1：不同规则如何影响文档

任何情况下，用户代理都会确定哪些规则与一个元素匹配，计算出所有相关的声明及其特殊性，确定哪些规则胜出，然后将胜出的规则应用到元素，从而得到应用样式后的结果。每个元素、选择器和声明上都必须完成这些工作。幸运的是，用户代理会自动完成所有这些工作。这个行为是层叠的一个重要部分，本章后面还将深入讨论层叠。

通配选择器特殊性

前面提到过，通配选择器对一个选择器的特殊性没有贡献。换句话说，其特殊性为0,0,0,0，这与根本没有特殊性有区别（有关内容将在“继承”一节中介绍）。因此，给定以下两条规则，div下包含的段落将是黑色，而其他元素都是灰色：

```
div p {color: black;}  /* 0,0,0,2 */
* {color: gray;}  /* 0,0,0,0 */
```

如你所料，这意味着如果一个选择器中包含通配选择器和其他选择器，该选择器的特殊性不会因通配选择器的出现而改变。下面两个选择器的特殊性完全相同：

```
div p  /* 0,0,0,2 */
body * strong  /* 0,0,0,2 */
```

相比之下，结合符则根本没有特殊性，甚至连0特殊性都没有。因此，它们对选择器的总特殊性没有任何影响。

ID和属性选择器的特殊性

需要着重指出，ID选择器和指定id属性的属性选择器在特殊性上有所不同。再来看示例代码中的第三组规则，可以看到：

```
html > body table tr[id="totals"] td ul > li {color: maroon;}  /* 0,0,1,7 */
li#answer {color: navy;}  /* 0,1,0,1  (winner) */
```

第二个规则中的ID选择器（#answer）为选择器的总特殊性贡献了0,1,0,0。而在第一个规则中，属性选择器（[id="totals"]）只对总特殊性贡献了0,0,1,0。因此，给定以下规则，id为meadow的元素将变成绿色：

```
#meadow {color: green;}  /* 0,1,0,0 */
*[id="meadow"] {color: red;}  /* 0,0,1,0 */
```

内联样式特殊性

到目前为止，我们已经见过以0开头的特殊性，所以你可能会奇怪为什么会有这些特殊性。一般地，第一个0是为内联样式声明保留的，它比所有其他声明的特殊性都高。考虑以下规则和标记片段：

```
h1 {color: red;}

<h1 style="color: green;">The Meadow Party</h1>
```

假设这个规则应用到h1元素，h1的文本还将是绿色。CSS2.1中就是如此，这是因为每个内联声明的特殊性都是1,0,0,0。

这意味着，即使有id属性的元素与某个规则匹配，也必须遵循内联样式声明。下面把上例修改为包括一个id属性：

```
h1#meadow {color: red;}

<h1 id="meadow" style="color: green;">The Meadow Party</h1>
```

由于内联声明的特殊性最高，h1元素的文本还是绿色。

注意： 为内联样式声明保留一位，这是CSS2.1才新增的，这样做是为了反映写CSS2.1当时的Web浏览器表现。在CSS2中，内联样式声明的特殊性是1,0,0（CSS2特殊性包含3个值，而不是4个）。换句话说，它与ID选择器的特殊性相同，所以ID选择器很容易覆盖内联样式。

重要性

有时某个声明可能非常重要，超过了所有其他声明。CSS2.1称之为重要声明（原因显而易见），并允许在这些声明的结束分号之前插入!important来标志。

```
p.dark {color: #333 !important; background: white;}
```

在此为颜色值#333加了标志!important，而背景值white未加这个标志。如果你希望把两个声明都标志为重要，那么每个声明都需要它自己的!important标志：

```
p.dark {color: #333 !important; background: white !important;}
```

必须正确地放置!important，否则声明将无效。!important总是放在声明的最后，即分号前面。如果一个属性的值可以包含多个关键词，如font，这一点则尤其重要，必须将!important标志放在声明的最后：

```
p.light {color: yellow; font: smaller Times, serif !important;}
```

如果!important放在font声明的任何其他位置，整个声明都将无效，相应地不会应用其任何样式。

标志为!important的声明并没有特殊的特殊性值，不过要与非重要声明分开考虑。实际上，所有!important声明会分组在一起，重要声明的特殊性冲突会在重要声明内部解决，而不会与非重要声明相混。类似地，我们认为所有非重要声明也归为一组，使用特殊性来解决冲突。如果一个重要声明和一个非重要声明冲突，胜出的总是重要声明。图3-2展示了以下规则和标记片段的结果：

```
h1 {font-style: italic; color: gray !important;}
.title {color: black; background: silver;}
* {background: black !important;}

<h1 class="title">NightWing</h1>
```

图3-2：重要规则总会胜出

注意： 本章稍后“层叠”一节中还会更详细地讨论重要声明及其处理。

继承

特殊性对于理解如何向文档应用声明很重要，同样的，还有一个很重要的概念，即继承。基于继承机制，样式不仅应用到指定的元素，还会应用到它的后代元素。例如，如果向一个h1元素应用一个颜色，那么这个颜色将应用到h1中的所有文本，甚至应用到该h1的子元素中的文本：

```
h1 {color: gray;}

<h1>Meerkat <em>Central</em></h1>
```

不仅正常的h1文本会变成灰色，连em文本也会是灰色，因为em元素继承了这个color值。如果属性值不能被后代元素继承，em文本应当是黑色，而不是灰色，则必须单独地为该元素指定颜色。

继承对无序列表也适用。假设为ul元素应用样式color: gray;：

```
ul {color: gray;}
```

你希望应用到ul的样式也应用到其列表项，并且应用到这些列表项中的所有内容。正是因为有继承，你将如愿以偿，如图3-3所示。

- Oh, don't you wish
- That you could be a fish
- And swim along with me
- Underneath the sea

1. Strap on some fins
2. Adjust your mask
3. Dive in!

图3-3：样式继承

要想了解继承是如何工作的，更容易的办法是查看文档的树图。图3-4显示了一个非常简单的文档树图，其中包含两个列表：一个无序，另一个有序。

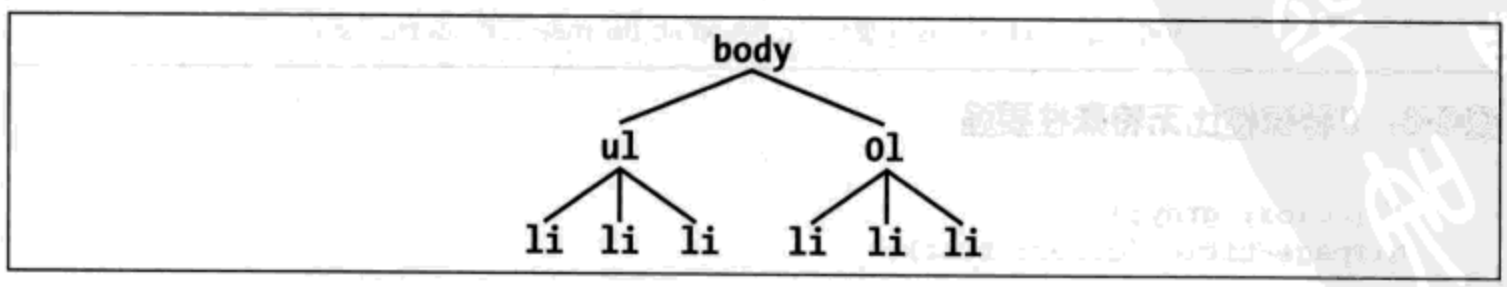

图3-4：一个简单的树图

将声明color:gray;应用到ul元素时，这个元素会采用该声明。这个值再沿着树向下

传播到后代元素，并一直继续，直到再没有更多的后代元素继承这个值为止。值绝对不会向上传播；也就是说，元素不会把值向上传递到其祖先。

注意：在HTML中，对于向上传播规则有一个例外：应用到`body`元素的背景样式可以传递到`html`元素（html是文档的根元素），相应地可以定义其画布。

继承是CSS中最基本的内容之一，除非有必要的理由，否则一般不会特别考虑，正所谓“熟视无睹”。不过，还是有几点需要记住。

首先，注意有些属性不能继承，这往往归因于一个简单的常识。例如，属性`border`（用于设置元素的边框）就不会继承。从图3-5可以了解到这是为什么。如果边框能继承，文档会变得更混乱，除非创作人员另外花功夫去掉继承的边框。

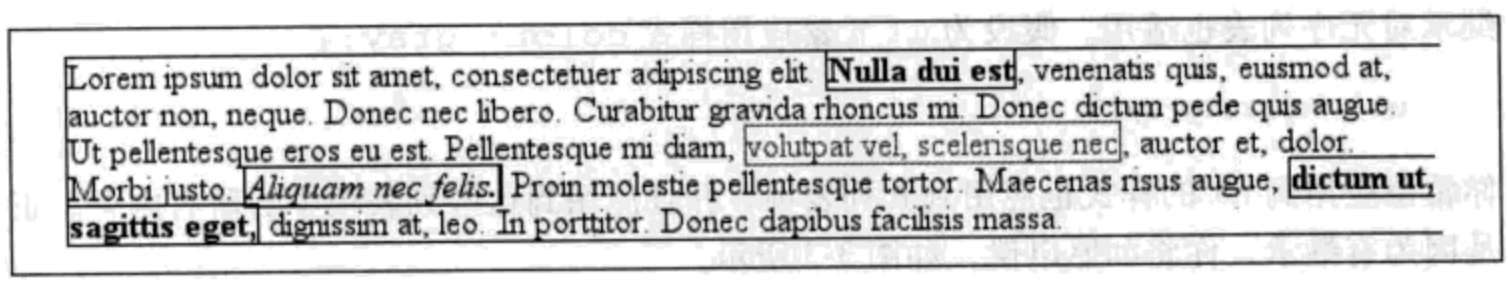

图3-5：为什么边框不能继承

一般地，大多数框模型属性（包括外边距、内边距、背景和边框）都不能继承，其原因是一样的。毕竟，你肯定不希望一个段落中的所有链接都从其父元素继承30像素的左外边距！

继承的值根本没有特殊性，甚至连0特殊性都没有。这看上去似乎只是理论上不同，不过等你了解到继承值没有特殊性会有什么结果时，就会知道这种差别绝不能忽视。考虑以下规则和标记片段，并对照图3-6所示的结果：

图3-6：0特殊性比无特殊性要强

```
* {color: gray;}
h1#page-title {color: black;}

<h1 id="page-title">Meerkat <em>Central</em></h1>
<p>
Welcome to the best place on the web for meerkat information!
</p>
```

继承 bug

由于不同的浏览器实现中可能存在的问题，创作人员不能指望依靠继承在所有情况下得到预想的结果。例如，Navigator 4（另外，从某种程度上讲，Explorer 4和5）的表不会继承样式。因此，以下规则会导致文档中除表之外的所有文本都有更小的字体：

```
body {font-size: 0.8em;}
```

在CSS中，这是不合适的，但这种行为确实存在，所以创作人员一直以来都求助于以下技巧：

```
body, table, th, td {font-size: 0.8em;}
```

如果浏览器本身在继承方面存在bug，在这样的浏览器中，利用上述技巧比较有可能得到所要的结果，不过仍然不能保证。

遗憾地是，在能正确实现继承的浏览器中，以上“修正方法”会导致更严重的问题，如IE6/Win、IE5/Mac、Netscape 6+等浏览器。在这些浏览器中，最终文本会放在一个表单元格中，其字体大小是用户默认字体大小设置的41%。应当针对更新的浏览器编写正确的CSS，相比之下，试图在老浏览器中解决继承bug的做法通常比较危险。

因为通配选择器适用于所有元素，而且有0特殊性，其颜色声明指定的值gray要优先于继承值（black），因为继承值根本没有特殊性。因此，em元素会显示为灰色而不是黑色。

这个例子生动地说明了不加区别地使用通配选择器可能存在的问题之一。由于它能匹配任何元素，所以通配选择器往往有一种短路继承的效果。这个问题可以解决，不过通常更合理的做法是从一开始就避免不加区别地使用通配选择器，从而从根本上避免这个问题。

继承值完全没有特殊性，这一点很关键，绝不能等闲视之。例如，假设一个样式表编写如下，使“工具条”中的所有文本都是黑底白字：

```
#toolbar {color: white; background: black;}
```

只要id为toolbar的元素只包含纯文本而不包含其他内容，这就能正常起作用。不过，如果这个元素中的文本都是超链接（a元素），用户代理的超链接样式就会占上风。在一个Web浏览器中，这意味着它们的颜色很可能是蓝色，因为浏览器的样式表可能包含以下规则：

```
a:link {color: blue;}
```

为克服这个问题，必须如下声明：

```
#toolbar {color: white; background: black;}
#toolbar a:link {color: white;}
```

通过向工具条中的a元素直接指定规则，可以得到如图3-7所示的结果。

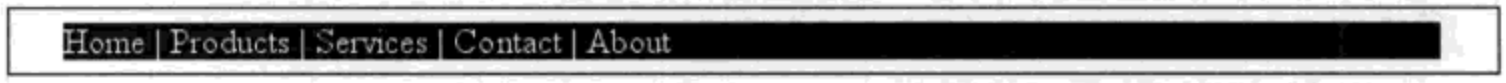

图3-7：向相关元素直接指定样式

层叠

在这一章中，我们回避了一个相当重要的问题：如果特殊性相等的两个规则同时应用到同一个元素会怎么样呢？浏览器如何解决这个冲突？例如，假设有以下规则：

```
h1 {color: red;}
h1 {color: blue;}
```

哪一个会占上风？这两个规则的特殊性都是0,0,0,1，所以它们的权重相等，都应当应用到元素。但这是不可能的，因为一个元素不可能既是红色又是蓝色。但是到底应用哪一个规则呢？

“层叠样式表”这个名字可能会提供一点提示。CSS所基于的方法就是让样式层叠在一起，这是通过结合继承和特殊性做到的。CSS2.1的层叠规则相当简单：

1. 找出所有相关的规则，这些规则都包含与一个给定元素匹配的选择器。
2. 按显式权重对应用到该元素的所有声明排序。标志!important的规则的权重要高于没有!important标志的规则。按来源对应用到给定元素的所有声明排序。共有3种来源：创作人员、读者和用户代理。正常情况下，创作人员的样式要胜过读者的样式。有!important标志的读者样式要强于所有其他样式，这包括有!important标志的创作人员样式。创作人员样式和读者样式都比用户代理的默认样式要强。
3. 按特殊性对应用到给定元素的所有声明排序。有较高特殊性的元素权重要大于有较低特殊性的元素。
4. 按出现顺序对应用到给定元素的所有声明排序。一个声明在样式表或文档中越后出现，它的权重就越大。如果样式表中有导入的样式表，一般认为出现在导入样式表中的声明在前，主样式表中的所有声明在后。

为了清楚地说明如何做到这些，下面来看三个例子，由此介绍上述4条层叠规则中的后三条规则。

按权重和来源排序

根据第二条规则，如果两个样式规则应用到同一个元素，而且其中一个规则有!important标志，这个重要规则将胜出：

```
p {color: gray !important;}

<p style="color: black;">Well, <em>hello</em> there!</p>
```

尽管段落的style属性中指定了一个颜色，不过有!important标志的规则会胜出，所以段落为灰色。em 元素也会继承这个灰色。

另外，还要考虑规则的来源。如果一个元素与创作人员样式表中的正常权重样式匹配，另外还与读者样式表中的正常权重样式匹配，则会使用创作人员的样式。例如，假设以下样式分别来自注释中指定的来源：

```
p em {color: black;}     /* author's style sheet */

p em {color: yellow;}    /* reader's style sheet */
```

在这种情况下，段落中强调的文本为黑色，而不是黄色，因为有正常权重的创作人员样式要优先于正常权重的读者样式。不过，如果两个规则都标志有!important，情况就不同了：

```
p em {color: black !important;}     /* author's style sheet */

p em {color: yellow !important;}    /* reader's style sheet */
```

现在段落中的强调文本将是黄色而不是黑色。

一般地，用户代理的默认样式（通常受用户首选项的影响）也要考虑在内。默认样式声明在所有声明当中影响是最小的。因此，如果创作人员定义的一个规则应用到锚（例如，声明这些锚颜色为white），该规则会覆盖用户代理的默认样式。

总结一下，在声明权重方面要考虑5级。权重由大到小的顺序依次为：

1. 读者的重要声明
2. 创作人员的重要声明
3. 创作人员的正常声明
4. 读者的正常声明
5. 用户代理声明

创作人员通常只需要考虑前4个权重级别，因为任何声明都会胜过用户代理样式。

按特殊性排序

根据第3条规则，如果向一个元素应用多个彼此冲突的声明，而且它们的权重相同，则要按特殊性排序，最特殊的声明最优先。例如：

```
p#bright {color: silver;}
p {color: black;}

<p id="bright">Well, hello there!</p>
```

给定以上所示的规则，段落的文本将是银色，如图3-8所示。为什么呢？因为p#bright的特殊性(0,1,0,1)大于p的特殊性(0,0,0,1)，尽管后一条规则在样式表中较后出现。

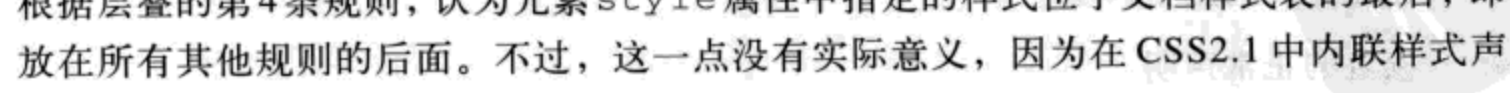

图3-8：较高特殊性强于较低特殊性

按顺序排序

最后，根据第4条规则，如果两个规则的权重、来源和特殊性完全相同，那么在样式表中后出现的一个会胜出。因此，再来看前面的例子，在文档样式表中可以看到以下两个规则：

```
h1 {color: red;}
h1 {color: blue;}
```

因为指定blue的规则在样式表中较后出现，所以文档中所有h1元素的color值将是blue，而不是red。倘若文档中包含的规则比导入的规则权重高，那么文档中包含的规则将胜出。即使这个规则是文档样式表的一部分而不是元素style属性的一部分，也是如此。考虑以下规则：

```
p em {color: purple;}  /* from imported style sheet */

p em {color: gray;}    /* rule contained within the document */
```

在这种情况下，以上第二个规则会胜过导入的规则，因为第二个规则是文档样式表的一部分。

根据层叠的第4条规则，认为元素style属性中指定的样式位于文档样式表的最后，即放在所有其他规则的后面。不过，这一点没有实际意义，因为在CSS2.1中内联样式声明的特殊性要高于所有样式表选择器。

注意：要记住，在CSS2中，内联样式声明与ID选择器的特殊性相等。在一个CSS2（而非CSS2.1）用户代理中，认为style属性声明出现在文档样式表的最后，并且与样式表中所有其他声明一样，按权重、来源、特殊性和顺序排序。

正是由于这种按顺序排序，所以才有了通常推荐的链接样式顺序。一般建议按link-visited-hover-active（LVHA）的顺序声明链接样式如下：

```
:link {color: blue;}
:visited {color: purple;}
:hover {color: red;}
:active {color: orange;}
```

根据这一章的介绍，现在你知道了所有这些选择器的特殊性都是一样的：0,0,1,0。因为它们都有相同的权重、来源和特殊性，因此与元素匹配的最后一个选择器才会胜出。正在“点击”的未访问链接可以与其中3个规则匹配——:link、:hover和:active——所以在这三个规则当中最后声明的一个将胜出。如果按照LVHA顺序，:active会胜出，这可能正是创作人员所期望的。

假设你想忽略这种常用的顺序，而是按字母顺序排列链接样式。这就会得到：

```
:active {color: orange;}
:hover {color: red;}
:link {color: blue;}
:visited {color: purple;}
```

按照这种顺序，任何链接都不会显示:hover或:active样式，因为:link和:visited规则后出现。所有链接都必须要么是已访问，要么是未访问的，所以:link和:visited样式总是会覆盖:hover规则。

下面考虑创作人员可能想使用的LVHA顺序的一个变种。采用这种顺序，只有未访问的链接会有悬停样式，已访问的链接没有这种样式。已访问和未访问的链接都可以有激活样式（active）：

```
:link {color: blue;}
:hover {color: red;}
:visited {color: purple;}
:active {color: orange;}
```

当然，有时如果所有规则试图设置同一个属性，就会出现这种冲突。如果各规则为不同的属性设置样式，那么顺序无关紧要。在下面的情况中，链接样式可以按任何顺序指定，而且不论采用何种顺序都能正常起作用：

```
:link {font-weight: bold;}
```

```
:visited {font-style: italic;}
:hover {color: red;}
:active {background: yellow;}
```

你可能还发现，:link和:visited样式的顺序并不重要。可以按LVHA或VLHA的顺序指定样式，这没有任何不良后果。不过，LVHA更好一些，因为这是CSS2规范中推荐的顺序，另一个原因是LVHA拼作“LoVe-HA!”，这个词更易于记忆，流通很广。

由于可以把伪类链接起来，所以可以不必担心这些问题。以下规则可以用任何顺序列出，而不必担心有什么负面影响：

```
:link {color: blue;}
:visited {color: purple;}
:link:hover {color: red;}
:visited:hover {color: gray;}
```

因为每个规则都应用到一组唯一的链接状态，它们不会冲突。因此，改变其顺序将不会改变文档的样式。后两个规则确实特殊性相同，不过这也没有关系。鼠标停留的未访问链接不会与针对悬停状态已访问链接的规则相匹配，反之亦然。如果要增加激活状态样式，就要重新考虑顺序了，请看下面的例子：

```
:link {color: blue;}
:visited {color: purple;}
:link:hover {color: red;}
:visited:hover {color: gray;}
:link:active {color: orange;}
:visited:active {color: silver;}
```

如果把激活样式移到悬停样式前面，它们会被忽略。同样地，这是由于特殊性冲突所致。可以向链中增加更多的伪类来避免这种冲突，如下：

```
:link:hover:active {color: orange;}
:visited:hover:active {color: silver;}
```

通过将伪类链接在一起，能缓解特殊性和顺序带来的问题，如果Internet Explorer以前就一直支持这种串链的伪类，它的应用可能会更广（有关的更多内容见第2章）。

非CSS表现提示

文档有可能包含非CSS的表现提示，例如font元素。非CSS提示被处理为特殊性为0，并出现在创作人员样式表的最前面。只要有创作人员或读者样式，这种表现提示就会被覆盖，但是用户代理的样式不能将其覆盖。

小结

层叠样式表中最基本的一个方面可能就是层叠了 —— 冲突的声明要通过这个层叠过程排序，并由此确定最终的文档表示。这个过程的核心是选择器及其相关声明的特殊性，以及继承机制。

在下一章中，我们将介绍多种用于为属性值提供含义的单位。讨论完下一章后，你就能清楚地了解全部基础知识，并做好进一步学习指定文档样式的属性的准备。

第4章

值和单位

在利用CSS能做的几乎所有工作中，其基础都是单位（units），这是影响所有属性的颜色、距离和大小的一种元素，本章就要讨论单位。如果没有单位，就不能声明某个段落应当是紫色，或者某个图像周围应当有10像素的空白，也不能声明一个标题的文本应当是某种大小。如果理解了这里介绍的概念，你就能更快地学习和使用CSS的余下内容。

数字

CSS中有两类数字：整数（“完整”的数）和实数（小数）。这些数字类型主要作为其他值类型的基础，不过在某些情况下，这些基本类型数字也可以用作属性的值。

在CSS2.1中，实数定义为一个整数后可以跟有一个小数点和小数部分。因此，以下都是合法的数字值：15.5、−270.00004和5。整数和实数都可以是正数或负数，不过属性可能（而且通常会）限制所允许的数字范围。

百分数

百分数值是一个计算得出的实数，其后跟有一个百分号（%）。百分数值几乎总是相对于另一个值，这个值可以是任意的：可能是同一元素另一个属性的值，也可以是从父元素继承的一个值，或者是祖先元素的一个值。接受百分数值的属性会对所允许的百分数值定义某些限制，还会定义百分数计算到什么程度。

颜色

每一位Web创作人员最早提出的问题之一可能是："怎么在页面上设置颜色？"使用HTML时，你有两种选择：可以按名使用为数不多的几种颜色，如red或purple；或者可以采用一种不那么一目了然的方法，即使用十六进制代码。这两种描述颜色的方法在CSS中都仍然保留，另外还提供了一些其他的描述方法，而且在我看来，新增的这些方法更为直观。

命名颜色

假设你觉得从一个很小的基本颜色集中选择就足够了，那么最容易的办法就是使用你想要的颜色名。CSS称这些有名字的颜色为命名颜色（道理很显然）。

一些浏览器制造商可能会让你相信命名颜色很丰富，然而恰好相反，合法的命名颜色关键字很有限。例如，你不能选择"mother-of-pearl"，因为这不是一个已定义的颜色。在CSS2.1中，CSS规范定义了17个颜色名。这包括HTML 4.01中定义的16个颜色，并外加一个橙色：

aqua	fuchsia	lime	olive	red	white
black	gray	maroon	orange	silver	yellow
blue	green	navy	purple	teal	

因此，假设你希望所有一级标题都是紫红色，最好的声明就是：

```
h1 {color: maroon;}
```

是不是很直截了当？图4-1显示了更多例子。

```
h1 {color: gray;}
h2 {color: silver;}
h3 {color: black;}
```

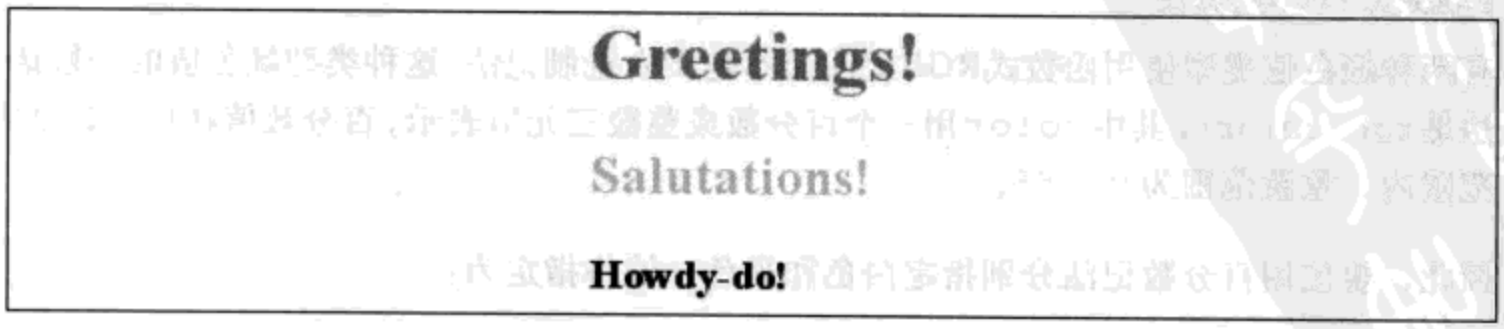

图4-1：命名颜色

当然，你可能见过（甚至用过）其他的颜色名，而不只是以上所列。例如，如果你指定以下规则：

```
h1 {color: lightgreen;}
```

很可能所有h1元素实际上会变成淡绿色，尽管lightgreen并不在CSS2.1的命名颜色表之列。它之所以能起作用，原因是大多数Web浏览器能识别多达140个颜色名，包括前面的17个标准颜色。另外的这些颜色都在CSS3颜色规范中定义，对此本书不作讨论。与那140种左右的颜色相比（这个颜色表就要长得多了），这17种标准颜色往往更可靠（在写作本书时只有17种标准颜色，至于以后则不能断言），因为这17种颜色的颜色值在CSS2.1中定义。CSS3中包含的140种颜色的扩展颜色表基于标准X11 RGB值，X11 RGB值已经使用了数十年，所以这些颜色能得到很好的支持。

幸运地是，CSS中还有一些更详细、更精确的方法用于指定颜色。其好处在于，采用这些方法可以指定色谱中的任何颜色，而不只是17种（或140种）命名颜色。

用RGB指定颜色

计算机通过组合不同的红色、绿色和蓝色分量来创造颜色，这种组合通常称为RGB颜色。实际上，如果你拆开一个老式的CRT计算机显示器，再向下“挖掘”找到显像管，你会发现三支“枪”（不过，如果你担心因为动了显示器而使显示器授权失效，那么建议你还是别去找这些枪）。这些枪会在屏幕各个点上以不同强度发射电子束，然后在屏幕的这些点上结合各个光束的强度，构成你看到的各种颜色。各个点称为像素（pixel），本章后面还会讨论这个概念。尽管当前大多数显示器并不使用电子枪，但是其颜色输出还是以RGB混合为基础。

根据显示器上创建颜色的方式，应该可以直接访问这些颜色，由你确定如何混合红、绿、蓝分量，从而最大程度地控制颜色。这种解决方法很复杂，不过这确实是可以的，而且是值得的，因为这样一来，对于能够产生哪些颜色几乎没有什么限制。可以采用4种方法以这种方式控制颜色。

函数式RGB颜色

有两种颜色值类型使用函数式RGB记法而不是十六进制记法。这种类型颜色值的一般语法是rgb(color)，其中color用一个百分数或整数三元组表示。百分数值在0%~100%范围内，整数范围为0~255。

因此，要使用百分数记法分别指定白色和黑色，值将指定为：

```
rgb(100%,100%,100%)
rgb(0%,0%,0%)
```

如果使用整数三元组记法，相同的颜色表示如下：

```
rgb(255,255,255)
rgb(0,0,0)
```

假设你希望h1元素有一个红色阴影，其颜色值在红色与紫红色之间。red等价于rgb(100%,0%,0%)，而maroon等于(50%,0%,0%)。要得到一个介于二者之间的颜色，可以试试下面的规则：

```
h1 {color: rgb(75%,0%,0%);}
```

这会让这种颜色的红色分量比maroon深，但比red浅。另一方面，如果你想创建一种灰红色，则要增加绿色和蓝色分量：

```
h1 {color: rgb(75%,50%,50%);}
```

如果使用整数三元组记法，与之最接近的颜色是：

```
h1 {color: rgb(191,127,127);}
```

要想看看这些值与颜色如何对应，最容易的办法就是建立一个灰度值表。另外，因成本所限，本书无法彩色印刷，只能显示灰度，所以在此将建立一个灰度表，如图4-2所示：

```
p.one {color: rgb(0%,0%,0%);}
p.two {color: rgb(20%,20%,20%);}
p.three {color: rgb(40%,40%,40%);}
p.four {color: rgb(60%,60%,60%);}
p.five {color: rgb(80%,80%,80%);}
p.six {color: rgb(0,0,0);}
p.seven {color: rgb(51,51,51);}
p.eight {color: rgb(102,102,102);}
p.nine {color: rgb(153,153,153);}
p.ten {color: rgb(204,204,204);}
```

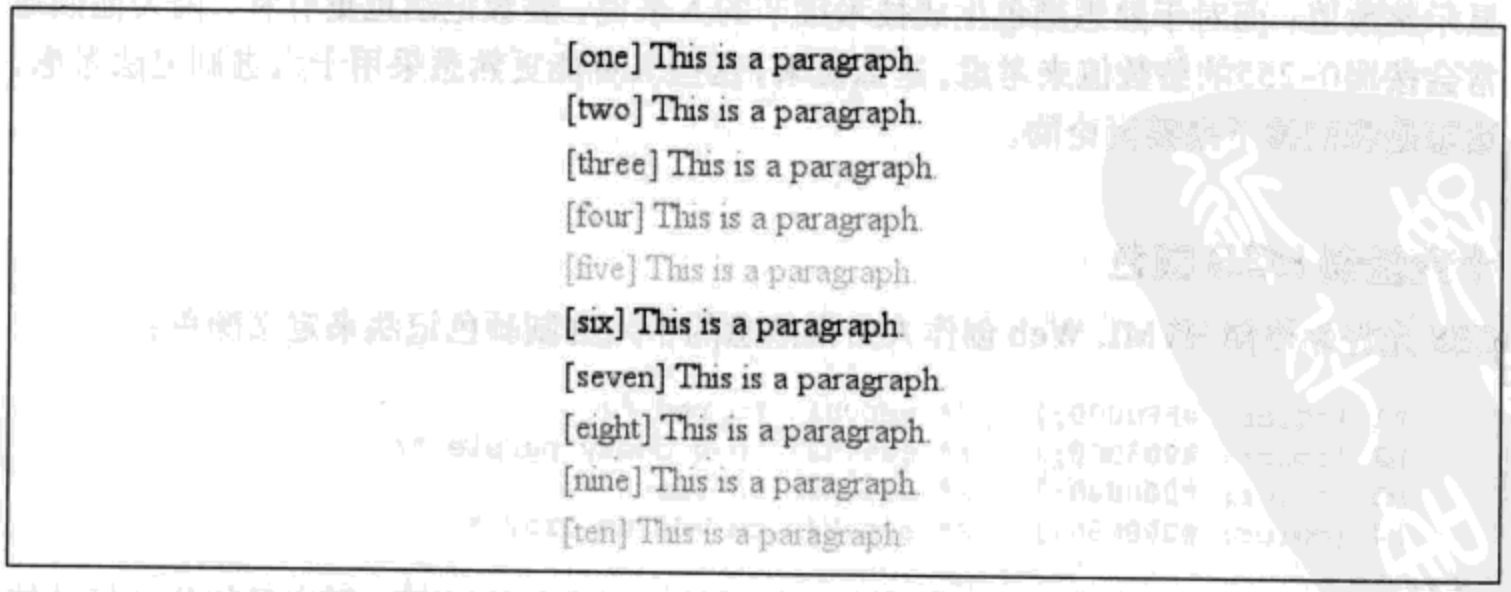

图4-2：用灰度设置的文本

当然，由于我们处理的是灰度，上述各个规则中RGB值的3个分量都一样。如果其中任

意一个分量不同于另外的分量，就会出现一种新颜色，例如，如果将rgb(50%,50%,50%)修改为rgb(50%,50%,60%)，就会得到一种中暗色，其中稍稍泛一点蓝色。

百分数记法中也可以使用分数。出于某种原因，你可能想指定某种颜色的红色分量为25.5%，绿色为40%，蓝色为98.6%：

```
h2 {color: rgb(25.5%,40%,98.6%);}
```

如果用户代理忽略小数点（有些用户代理确实会这么做），就会把这些值取整为与之最接近的整数，从而得到值rgb(26%,40%,99%)。当然，在整数三元组中只能使用整数。

无论哪一种记法，如果值落在可取范围之外，都会“剪裁”到最接近的范围边界，这意味着，如果一个值大于100%或小于0%，就会默认地调整为100%或0%（这是可取的最大和最小极限）。因此，以下声明会处理为好像声明了注释中指定的值：

```
P.one {color: rgb(300%,4200%,110%);}   /*  100%,100%,100%  */
P.two {color: rgb(0%,-40%,-5000%);}    /*  0%,0%,0%  */
p.three {color: rgb(42,444,-13);}     /* 42,255,0  */
```

百分数和整数之间的转换看上去似乎是任意的，不过没有必要去猜测想要的相应整数，对此有一个简单的计算公式。如果知道所要的各RGB分量的百分数，只需将它们应用到255（乘以255），就能得到结果值。假设有一个颜色的红色分量为25%，绿色分量为37.5%，蓝色分量为60%。将这些百分数乘以255，就会得到63.75、95.625和153。再把这些值取整为最接近的整数，这就得到了*voilà*：rgb(64,96,153)。

当然，如果你已经知道百分数值，那么将其转换为整数并没有什么意义。对于使用Photoshop之类程序的人来说，整数记法更为有用，因为这些程序可以在Info对话框中显示整数值；而对于熟悉颜色生成技术细节的人来说，整数记法也更有利，因为他们通常会按照0~255的整数值来考虑。这么说来，这些人可能更熟悉采用十六进制记法考虑，这正是我们接下来要讨论的。

十六进制RGB颜色

CSS允许你使用HTML Web创作人员很熟悉的十六进制颜色记法来定义颜色：

```
h1 {color: #FF0000;}   /* set H1s to red */
h2 {color: #903BC0;}   /* set H2s to a dusky purple */
h3 {color: #000000;}   /* set H3s to black */
h4 {color: #808080;}   /* set H4s to medium gray */
```

到目前为止，计算机使用“十六进制记法”已经有相当长的时间，程序员往往会经过培训或由实战经验来了解如何使用十六进制。由于他们对十六进制记法相当熟悉，所以在老式的HTML中往往会使用十六进制记法来设置颜色。这种做法也沿袭到了CSS中。

以下介绍其工作原理：将三个介于00~FF的十六进制数连起来，就可以设置一种颜色。这种记法的一般语法是#RRGGBB。注意，在这三个数之间没有空格、逗号或其他分隔符。

十六进制记法在数学上等价于上一节讨论的整数三元组记法。例如，rgb(255,255,255)就完全等价于#FFFFFF，而rgb(51,102,128)则与#336680完全相同。你完全可以使用你想用的任何记法，无论哪一种记法，大多数用户代理给出的表示都是相同的。如果你有一个计算器；可以在十进制和十六进制之间转换，那么在这两种记法间切换相当简单。

如果组成十六进制数的3组数各自都是成对的，CSS还允许采用一种简写记法。这种记法的一般语法是#RGB：

```
h1 {color: #000;}   /* set H1s to black */
h2 {color: #666;}   /* set H2s to dark gray */
h3 {color: #FFF;}   /* set H3s to white */
```

从标记可以看到，每个颜色值中只有3位。不过，因为00~FF之间的十六进制数需要2位，而你总共只有3位，这又是怎么做到的呢？

答案是，浏览器会取每一位，并将其复制成两位。因此，#F00等价于#FF0000，#6FA与#66FFAA相同，#FFF则变成#FFFFFF，这就是white。显然，并非每种颜色都可以采用这种方式表示。例如，中灰色（medium gray）用标准十六进制记法可以写作#808080。这就不能用简写记法来表示，与其最接近的记法是#888，而这等同于#888888。

颜色汇总

对于我们前面讨论的所有颜色，表4-1做了一个总结。浏览器可能无法识别这些颜色关键字，因此，为了更为安全，要用RGB或十六进制三元组值来定义。另外，有些简写的十六进制值根本不会出现。在这些情况下，采用标准记法的较长值（6位）不能简写为3位，因为这些值不能进行复制。例如，值#880会扩展为#888800而不是#808000(即olive)。因此，#808000没有简写版本，表中相应的项为空。

表4-1：等价颜色表

颜色	百分数	数值	十六进制	简写十六进制
red	rgb(100%,0%,0%)	rgb(255,0,0)	#FF0000	#F00
orange	rgb(100%,40%,0%)	rgb(255,102,0)	#FF6600	#F60
yellow	rgb(100%,100%,0%)	rgb(255,255,0)	#FFFF00	#FF0
green	rgb(0%,50%,0%)	rgb(0,128,0)	#008000	

表4-1：等价颜色表（续）

颜色	百分数	数值	十六进制	简写十六进制
blue	rgb(0%,0%,100%)	rgb(0,0,255)	#0000FF	#00F
aqua	rgb(0%,100%,100%)	rgb(0,255,255)	#00FFFF	#0FF
black	rgb(0%,0%,0%)	rgb(0,0,0)	#000000	#000
fuchsia	rgb(100%,0%,100%)	rgb(255,0,255)	#FF00FF	#F0F
gray	rgb(50%,50%,50%)	rgb(128,128,128)	#808080	
lime	rgb(0%,100%,0%)	rgb(0,255,0)	#00FF00	#0F0
maroon	rgb(50%,0%,0%)	rgb(128,0,0)	#800000	
navy	rgb(0%,0%,50%)	rgb(0,0,128)	#000080	
olive	rgb(50%,50%,0%)	rgb(128,128,0)	#808000	
purple	rgb(50%,0%,50%)	rgb(128,0,128)	#800080	
silver	rgb(75%,75%,75%)	rgb(192,192,192)	#C0C0C0	
teal	rgb(0%,50%,50%)	rgb(0,128,128)	#008080	
white	rgb(100%,100%,100%)	rgb(255,255,255)	#FFFFFF	#FFF

Web安全颜色

所谓“Web安全”颜色是指，在256色计算机系统上总能避免抖动的颜色。Web安全颜色可以表示为RGB值20%和51（相应的十六进制值为33）的倍数。另外，0%或0也是一个安全值。因此，如果使用RGB百分数，要让所有这3个分量都要么是0%，要么是一个能被20整除的数，例如rgb(40%,100%,80%)或rgb(60%,0%,0%)。如果使用0~255范围的RGB值，则各分量值要么是0要么是能被51整除的数，如rgb(0,204,153)或rgb(255,0,102)。

采用十六进制记法，使用值00、33、66、99、CC和FF的三元组都认为是Web安全的。这种例子有#669933、#00CC66和#FF00FF。这说明，Web安全颜色的简写十六进制值是0，3，6，9，C，和F；因此，#693、#0C6和#F0F都是Web安全颜色的例子。

长度单位

很多CSS属性（如外边距）都依赖于长度度量来适当地显示各种页面元素。因此，CSS中有很多度量长度的方法不足为奇。

所有长度单位都可以表示为正数或负数，其后跟有一个标签（不过有些属性只接受正

数）。另外还可以使用实数，也就是有小数部分的数，如10.5或4.561。所有长度单位后面都有一个两字母缩写，它表示所指定的具体长度单位，如in（英寸）或pt（点）。这个规则只有一个例外，这就是长度为0（零）时，其后不需要跟单位。

这些长度单位可以划归为两类：绝对长度单位和相对长度单位。

绝对长度单位

首先来介绍绝对长度单位，因为它们最容易理解（尽管在Web设计中几乎很少使用绝对长度单位）。有5种绝对长度单位，如下：

英寸（in）

可以想见，这种记法是指美国尺子上都有的单位：英寸（在全世界都使用米制体系的今天，规范中居然还有这个单位，这是一个很有意思的现象，由此可以看出美国对Internet的影响之大，不过现在还是不要介入这些社会问题吧）。

厘米（cm）

这是指全世界尺子上都有的单位：厘米。1英寸是2.54厘米，1厘米等于0.394英寸。

毫米（mm）

对于不想用米制的美国人来说，10毫米等于1厘米，所以1英寸等于25.4毫米，1毫米等于0.0394英寸。

点（pt）

点是一个标准印刷度量单位，在打印机和打字机上已经使用了数十年，另外字处理程序使用点作为度量单位也有很多年了。以往，一英寸是72点（点是米制体系广泛使用之前定义的）。因此，如果文本的首字母设置为12点，这就是1英寸的1/6高。例如，`p{font-size: 18pt;}`等价于`p{font-size: 0.25in;}`。

派卡（pc）

派卡（Pica）也是一个印刷术语。1派卡相当于12点，这意味着，6派卡等于1英寸。如上所示，如果文本的首字母设置为1派卡，这应当是1英寸的1/6高。例如，对于前面定义点时给出的示例声明，`p{font-size: 1.5pc;}`会把文本设置为与上例大小相同。

当然，只有当浏览器知道用来显示页面的显示器、所用的打印机或其他任何用户代理的所有细节时，这些单位才真正有用。在一个Web浏览器上，显示会受显示器的尺寸影响，另外所设置的显示器分辨率也会有影响——作为创作人员，对于这些因素你往往无计可施。你只能希望（如果没有别的想法）这些度量相互之间要一致，也就是说，设置为`1.0in`将是`0.5in`的两倍大，如图4-3所示。

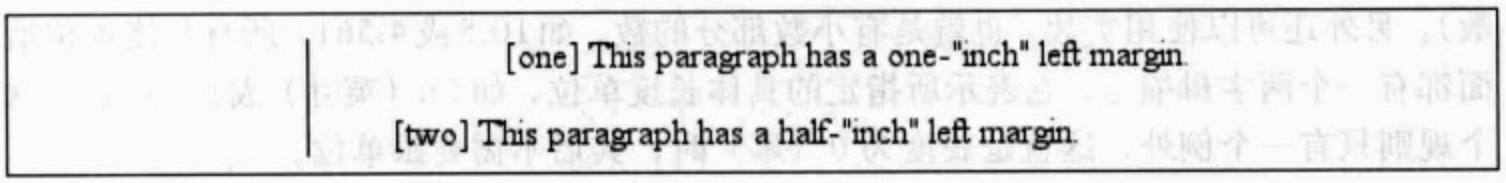

图4-3：用绝对长度设置左边距

处理绝对长度

如果一个显示器的分辨率设置为1,024像素×768像素，其屏幕大小为14.22英寸宽、10.67英寸高，而且显示区完全覆盖整个屏幕，那么每个像素的宽和高都为1英寸的1/72。可以预料，这种情况非常非常少见（你见过哪个显示器的大小是这样的？）。所以，在大多数显示器上，每英寸的实际像素数（ppi）都高于72，有时还会高很多，达到120 ppi甚至更高。

作为一个Windows用户，你可以设置显示驱动程序，使元素的显示完全对应于实际度量。可以试一下，单击开始→设置→控制面板。在控制面板中双击“显示”，单击“设置”页，再单击“高级”，可以看到一个对话框（不同PC上的对话框可能不同）。应该能看到标有“字体大小”的一部分；选择“其他”，然后拿把尺子贴到屏幕上，移动滚动条，直到屏幕上的尺子与真正的尺子完全一致。单击OK按钮，退出对话框，设置完成。

如果你使用的是一个Mac Classic操作系统，那么在操作系统中无法设置这个信息，Mac Classic OS（即OS X之前的所有版本）对于屏幕上像素和绝对度量之间的关系有一个假设，它声明显示器每英寸有72像素。这个假设完全是错误的，不过它内置在操作系统中，因此无法避免。所以，在许多基于Classic Mac的Web浏览器上，点值是多少，相应的像素值就是多少： `24pt` 文本就是24像素高，`8pt` 文本则是8像素高。遗憾的是，这实在太小了，以至于往往看不到。图4-4展示了这个问题。

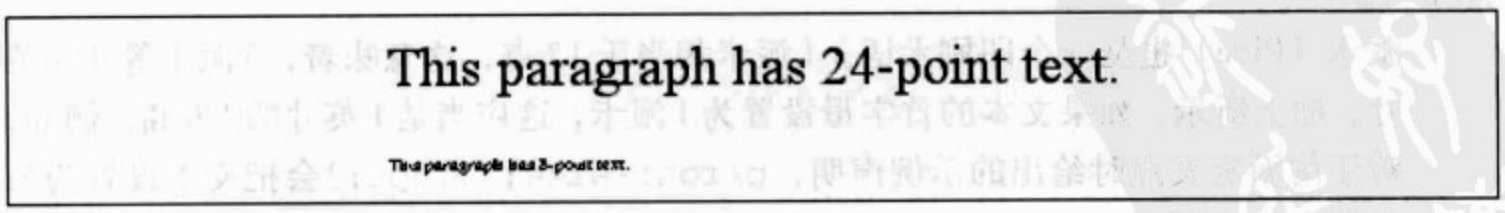

图4-4：太小的文本很难阅读

在OS X中，内置的假设ppi值与Windows的内置值很接近：96ppi。这也不见得有多正确，不过至少与Windows计算机一致。

Classic Mac显示问题作为一个例子很好地说明了设置Web页面时为什么要避免使用点。在浏览器显示方面，em、百分数甚至像素都比点要好。

注意： 从面向 Macintosh 的 Internet Explorer 5 和基于 Gecko 的浏览器（如 Netscape 6+）开始，浏览器本身包含了一个首选项设置来设置 ppi 值。可以选择标准 Macintosh 分辨率 72ppi，常用的 Windows 分辨率 96ppi，或者是适合你的显示器的 ppi 分辨率。最后这种选择类似于前面介绍的 Windows 设置，也就是用一个滑动尺与真正的尺子比较，从而使显示器与真实世界中的实际距离完全一致。

暂不考虑我们看到的，下面做一个可能性不大的假设，假设你的计算机相当了解其显示系统，可以准确地再现真实世界的度量。在这种情况下，通过声明 `p{margin-top: 0.5in;}` 可以确信每个段落的上边距都是半英寸。不论字体大小，也不管任何其他情况，段落的上边距都是半英寸。

绝对单位在定义打印文档的样式表时更为有用，在此通常会以英寸、点和派卡来度量长度。可以看到，在 Web 设计中试图使用绝对度量往往不是最佳的做法，所以下面来看一些更有用的度量单位。

相对长度单位

相对单位之所以得名，是因为它们是根据与其他事物的关系来度量的。所度量的实际（或绝对）距离可能因为不在其控制之下的其他因素而改变，如屏幕分辨率、可视区的宽度、用户的首选项设置，以及很多其他方面。另外，对于某些相对单位，其大小几乎总是对应于使用该单位的元素，因此会因元素的不同而不同。

共有 3 种相对长度单位：`em`、`ex` 和 `px`。前两个单位代表“em-height”和“x-height”，这是常用的印刷度量单位；不过，如果你很熟悉印刷术语，会发现在 CSS 中它们有另外的含义。最后一种长度单位是 `px`，这代表“像素”。如果仔细查看屏幕，一个像素就是你在计算机显示器上看到的一个点。这个值被定义为相对单位，因为它取决于显示设备的分辨率，稍后将介绍这个内容。

em 和 ex 单位

首先我们来考虑 `em` 和 `ex`。在 CSS 中，1 个“em”定义为一种给定字体的 `font-size` 值。如果一个元素的 `font-size` 为 14 像素，那么对于该元素，`1em` 就等于 14 像素。

显然，这个值可能随元素的不同而不同。例如，假设一个 `h1` 的字体大小为 24 像素，一个 `h2` 元素的字体大小为 18 像素，还有一个段落的字体大小为 12 像素。如果将所有这三个元素的左边距都设置为 `1em`，那么它们的左边距就分别为 24 像素、18 像素和 12 像素：

```
h1 {font-size: 24px;}
h2 {font-size: 18px;}
```

```
p {font-size: 12px;}
h1, h2, p {margin-left: 1em;}
small {font-size: 0.8em;}

<h1>Left margin = <small>24 pixels</small></h1>
<h2>Left margin = <small>18 pixels</small></h2>
<p>Left margin = <small>12 pixels</small></p>
```

另一方面，在设置字体的大小时，em的值会相对于父元素的字体大小改变，如图4-5所示。

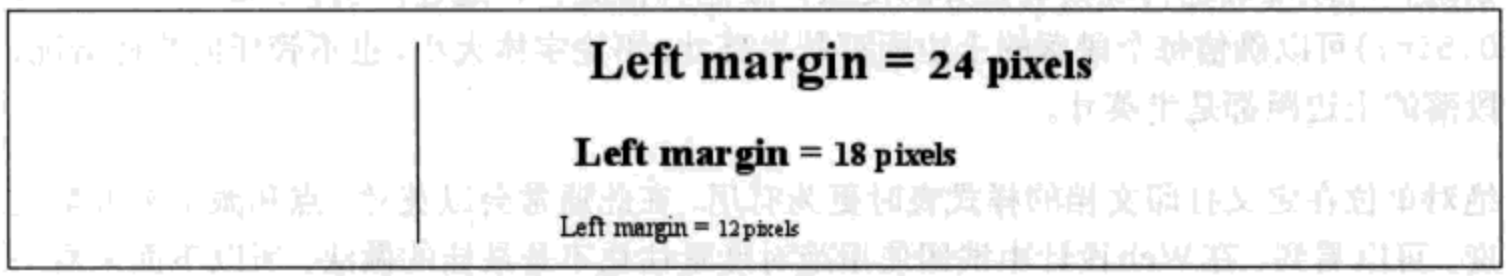

图4-5：对外边距和字体大小使用em作为长度单位

与此不同，ex是指所用字体中小写*x*的高度。因此，如果有两个段落，其中文本的大小为24点，但是各段使用了不同的字体，那么各段相应的ex值可能不同。这是因为，不同字体中*x*的高度可能不同，如图4-6所示。尽管这些例子的文本都使用了24点——相应地它们的em值都是24点——但是x的高度却各不相同。

em和ex的实际问题

当然，上述所有解释都只是理论上的。前面只是指出了可能会发生什么，但是在实际中，很多用户代理的做法是：取em的值，再取其一半作为ex值。为什么呢？显然，大多数字体都没有内置ex高度值，而且计算这个值相当困难。由于大多数字体的小写字母都是相应大写字母高度的一半，所以可以方便地假设1ex等于0.5em。

一些浏览器（包括面向Mac的Internet Explorer 5）会在内部显示一个小写的*x*，并计算相应的像素值来确定其高度与此字符font-size值之比，从而试图确定给定字体的x高度。这不是一个最佳的方法，但是这比简单地假设1ex等于0.5em好多了。作为使用CSS的人来说，我们可以相信，随着时间的推移，更多的用户代理都会开始使用ex的实际值，那种取em的一半作为ex的简便做法会逐渐淡出历史。

像素长度

从表面来看，像素很直接。如果仔细地查看一个显示器，你会看到，它被划分成一个由小框组成的网格。每个框就是一个像素。如果将一个元素的高和宽定义为某个像素数，如以下标记所示：

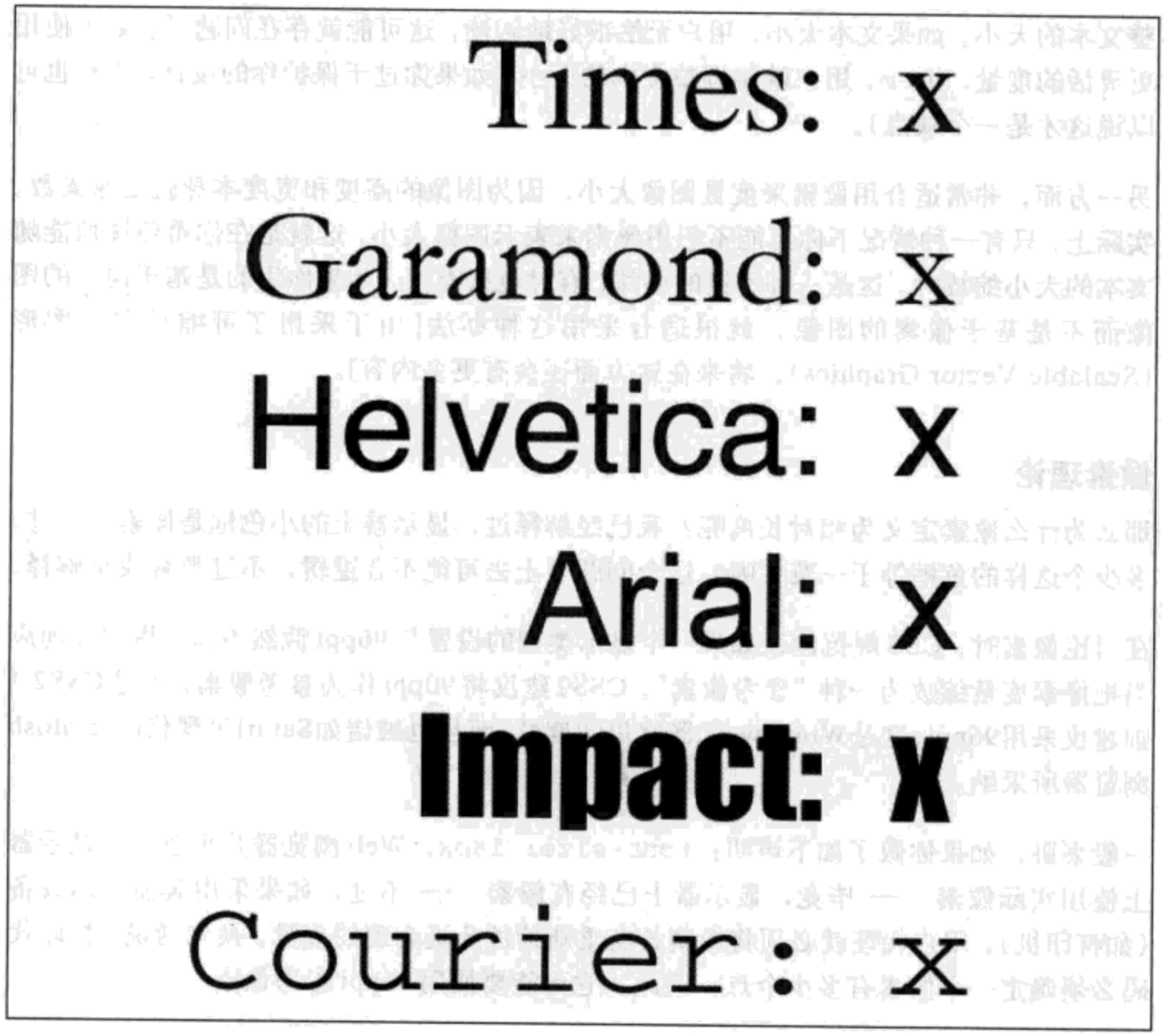

图4-6：x高度不同

```
<p>
The following image is 20 pixels tall and wide: <img src="test.gif"
  style="width: 20px; height: 20px;" alt="" />
</p>
```

那么这个元素的高和宽就会由相应多个显示器元素组成，如图 4-7 所示。

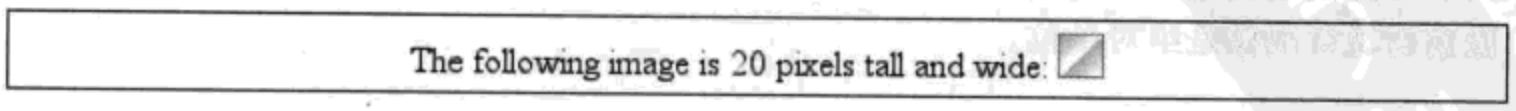

图4-7：使用像素长度

遗憾的是，使用像素有一个潜在的缺点。如果按像素设置字体大小，那么 Windows 平台 Internet Explorer （IE7 之前）的用户将无法使用其浏览器中的“文本大小”菜单调

整文本的大小。如果文本太小，用户无法很好地阅读，这可能就存在问题了。如果使用更灵活的度量，如em，用户就能调整文本的大小（如果你过于保护你的设计，当然也可以说这才是一个缺点）。

另一方面，非常适合用像素来度量图像大小，因为图像的高度和宽度本身就是像素数。实际上，只有一种情况下你可能不想用像素来表示图像大小，这就是在你希望图像能随文本的大小缩放时。这是一种不错的办法，有时也很有用，如果你用的是基于向量的图像而不是基于像素的图像，就很适合采用这种办法[由于采用了可缩放向量图形(Scalable Vector Graphics)，将来在这方面还会有更多内容]。

像素理论

那么为什么像素定义为相对长度呢？我已经解释过，显示器上的小色框是像素。不过，多少个这样的色框等于一英寸呢？这个问题看上去可能不合逻辑，不过暂容我来解释。

在讨论像素时，CSS规范建议如果一个显示类型的设置与96ppi截然不同，用户代理应当把像素度量缩放为一种“参考像素”。CSS2建议将90ppi作为参考像素，不过CSS2.1则建议采用96ppi，这是Windows机器常用的度量，而且也被诸如Safari等现代Macintosh浏览器所采纳。

一般来讲，如果你做了如下声明：`font-size: 18px`，Web浏览器几乎总会在显示器上使用实际像素 —— 毕竟，显示器上已经有像素 —— 不过，如果采用其他显示设备(如打印机)，用户代理就必须将像素长度重新缩放为更合理的度量。换句话说，打印代码必须确定一个像素有多少个点，为此，它可能要使用96ppi参考像素。

警告： 关于像素度量存在一些问题，在较早的CSS1实现中就可以看到这些问题的一个例子。在Internet Explorer 3.x中，当打印一个文档时，IE3认为`18px`就等于18点，而在一个600dpi的打印机上，则是一英寸的18/600或3/100 —— 如果愿意也可以记为`.03in`。这样的文本实在太小了！

由于可能会进行这样的重新缩放，像素被定义为一种相对度量单位，尽管在Web设计中像素表现得很像是绝对单位。

如何取舍？

根据上述所有问题，最好的度量可能是相对度量，特别是em，如果合适也可以使用px。在大多数当前使用的浏览器中，由于ex实际上是em的一部分（0.5em），所以对目前来说ex

还不太有用。如果更多的用户代理支持实际的x高度度量，ex可能就会有其用武之地了。总地来说，em是最灵活的，因为它随字体大小缩放，所以元素和元素操作都能更为一致。

对于元素的其他方面，可能更适合使用像素，如元素的边框或定位。这完全由具体情况而定。例如，在原先使用空格GIF来分隔各部分的设计中，如果边距使用像素长度，会得到一种等间距的效果。将这个间距转换为em，会使得设计随文本大小的变化而扩大或收缩，这可能很好，也可能并不是件好事。

URL

如果你写过Web页面，肯定对URL（或者在CSS2.1中称为URI）很熟悉。如果需要引用一个URL（如@import语句中，导入外部样式表时就会使用这条语句），一般格式则为：

```
url(protocol://server/pathname)
```

这个例子定义了一个所谓的绝对URL（absolute URL）。这里的绝对是指，不论这个URL放在哪里（或者放在哪个页面上），它都能正常工作，因为它定义了Web空间中的一个绝对位置。假设你有一个名为*www.waffles.org*的服务器。该服务器上有一个名为pix的目录，在这个目录中有一个图像*waffle22.gif*。这种情况下，该图像的绝对URL将是：

```
http://www.waffles.org/pix/waffle22.gif
```

不管这个URL放在哪里，它都是合法的，而不论包含这个URL的页面在服务器*www.waffles.org*上还是在*web.pancakes.com*上。

另一种URL是相对URL（relative URL），之所以得名是因为它指定的是一个相对于该URL所在文档的位置。如果指向一个相对位置，如Web页面所在目录中的一个文件，则一般格式为：

```
url(pathname)
```

如果图像与包含该URL的页面在同一个服务器上，这就能正常工作。为了说明这一点，假设有一个位于*http://www.waffles.org/syrup.html*的页面，而且希望这个页面上出现图像*waffle22.gif*。在这种情况下，URL将是：

```
pix/waffle22.gif
```

这个路径是合法的，因为Web浏览器知道它要从Web文档所在位置开始，然后加上这个相对URL来找到图像。在这里，路径名*pix/waffle22.gif*要增加到服务器名*http://www.waffles.org*，得到了*http://www.waffles.org/pix/waffle22.gif*。使用相对URL的地

方几乎总能用一个绝对URL取而代之，使用哪一种URL并不重要，只要它能定义一个合法的位置。

在CSS中，相对URL要相对于样式表本身，而不是相对于使用该样式表的HTML文档。例如，可能有一个外部样式表，其中导入了另一个样式表。如果使用相对URL来导入第二个样式表，它必须相对于第一个样式表。举例来说，来考虑一个位于 *http://www.waffles.org/toppings/tips.html* 的HTML文档，它包含一个指向样式表 *http://www.waffles.org/styles/basic.css* 的链接：

```
<link rel="stylesheet" type="text/css"
    href="http://www.waffles.org/styles/basic.css">
```

在文件 *basic.css* 中有一个 `@import` 语句指向另一个样式表：

```
@import url(special/toppings.css);
```

这个`@import`会导致浏览器查找位于*http://www.waffles.org/styles/special/toppings.css*的样式表，而不是在 *http://www.waffles.org/toppings/special/toppings.css* 查找样式表。如果在后面这个位置上有一个样式表，那么 *basic.css* 中的 `@import` 应当读作：

```
@import url(http://www.waffles.org/toppings/special/toppings.css);
```

警告： Netscape Navigator 4会相对于HTML文档而不是相对于样式表解释相对URL。如果有很多NN4.x用户访问你的网页，或者如果你想确保NN4.x能找到你的所有样式表和背景图像，通常最容易的做法是让所有URL都是绝对URL，因为Navigator可以正确地处理绝对URL。

注意，`url` 和开始括号之间不能有空格：

```
body {background: url(http://www.pix.web/picture1.jpg);}    /* correct */
body {background: url  (images/picture2.jpg);}             /* INCORRECT */
```

如果存在空格，整个声明都将无效，以至于被忽略。

关键字

有时一个值需要用某个词来描述，这种词就称为关键字。对此一个很常见的例子就是关键字none，它不同于0（零）。因此，要去除一个HTML文档中链接的下划线，应写作：

```
a:link, a:visited {text-decoration: none;}
```

类似地，如果想对链接加下划线，则要使用关键字`underline`。

如果一个属性接受关键字，那么其关键字将只针对该属性的作用域定义。如果两个属性都使用同一个词作为关键字，一个属性的关键字与另一个属性的同一关键字可能就有不同的行为。举例来说，为 letter-spacing 定义的 normal 与为 font-style 定义的 normal 含义就大不相同。

inherit

CSS2.1中有一个关键字是所有属性共有的，这就是inherit。inherit使一个属性的值与其父元素的值相同。在大多数情况下，不必指定继承，因为大多数属性本身会自然地继承；不过，inherit还是很有用的。

例如，考虑以下样式和标记：

```
#toolbar {background: blue; color: white;}

<div id="toolbar">
<a href="one.html">One</a> | <a href="two.html">Two</a> |
<a href="three.html">Three</a>
</div>
```

div本身将有一个蓝色背景和一个白色前景色，但是链接还是会根据浏览器的首选顶设置来应用样式。最后往往会是蓝色背景上的蓝色文本，之间有白色的竖线将其分隔。

可以编写一个规则，明确地将“工具条”中的链接设置为白色，不过通过使用inherit可以更健壮地做到这一点。只需向样式表增加以下规则：

```
#toolbar a {color: inherit;}
```

这会让链接使用继承的color值而不是用户代理的默认样式。正常情况下，直接指定的样式总会优先于继承的样式，但是通过使用 inherit 可以把情况反过来。

CSS2 单位

除了已经介绍的CSS2.1中的单位，CSS2还包含另外几个单位，所有这些单位都与声音样式表有关（支持语音的浏览器会使用这种样式表）。这些单位并没有包含在CSS2.1中，但是由于它们可能是 CSS 将来版本的一部分，在此做一个简单的讨论。

角度值

用于定义给定的声音从哪个位置发出。共有3种角度：度（deg）、梯度（grad）和弧度（rad）。例如，直角可以声明为 90deg、100grad 或 1.57rad；不论如何声明，这些值都会解释为 0~360 度范围内的度数。负数值也是如此（允许是负数），-90deg 等同于 270deg。

时间值

用于指定语音元素之间的延迟。可以表示为毫秒（ms），也可以表示为秒（s）。因此，100ms和0.1s是相同的。时间值不能是负值，因为CSS的设计要求避免这种不可能的情况。

频率值

用于为语音浏览器可以产生的声音声明一个给定频率。频率值可以表示为赫兹（Hz）或兆赫（MHz），而且不能是负值。值后面跟的标签（Hz或MHz）不区分大小写，因此10MHz和10Mhz是一样的。

在写这本书时，已经知道的支持所有这些值的唯一一个用户代理是*Emacspeak*，这是一个声音样式表实现。有关声音样式的更多详细信息见第14章。

除了这些值以外，还有一个“老朋友”，不过它有了一个新名字，这就是URI（Uniform Resource Identifier，统一资源标识符），这是统一资源定位符（Uniform Resource Locator，URL）的另一个名字。CSS2和CSS2.1规范都要求URI要以url(...)形式声明，因此实际上没有什么改变。

小结

单位和值的覆盖面很广，从长度单位到描述效果（如underline）的特殊关键字，再到颜色单位，还包括文件（如图像）的位置。大多数情况下，在单位方面，用户代理几乎能做到完全正确，不过也存在少量bug和奇怪问题会来烦你。例如，Navigator 4.x不能正确地解释相对URL，这被许多创作人员过分夸大，以至于过分地依赖于绝对URL。用户代理在颜色领域也几乎完全胜任，但同样不乏一些小问题。不过，由于存在太多不同的长度单位，尽管这绝对不是bug，但实际上这才是所有创作人员需要解决的一个有意思的问题。

这些单位都各有优缺点，这取决于它们在什么情况下使用。我们已经了解了这样的一些环境，本书余下的内容将重点讨论这些场合，首先从CSS属性开始，CSS属性描述了如何改变文本的显示方式。

第5章

字体

CSS规范的作者清楚地认识到，字体选择是一个常见（而且重要）的特性。毕竟，有多少页面分布着数十个甚至数百个<FONT FACE="...">标记呢？实际上，规范中“字体属性”一节最开始就有这样一句话：“设置字体属性是样式表的最常见用途之一。”

不过，尽管字体选择很重要，但是目前还没有一种办法能确保在Web上一致地使用字体，因为没有一种统一描述字体和字体的变形的方法。例如，字体Times、Times New Roman和TimesNR可能很类似，甚至完全相同，不过用户代理怎么能知道这一点呢？创作人员可能在一个文档中指定字体为TimesNR,但是如果用户机器上没有安装这种字体，用户查看文档时会看到什么呢？即使安装了Times New Roman，用户代理也不知道这两个字体（Times New Roman和TimesNR）实际上是可以互换的。如果你希望一个阅读器上一定采用某种字体，请别妄想了。

尽管CSS2支持可下载字体，并定义了相应属性，不过这些字体在Web浏览器中并未得到很好的实现，而且出于性能方面的原因，阅读器总会拒绝下载字体。与字处理器相比，CSS对字体并没有提供更多的最终控制；别人加载你创建的一个Microsoft Office文档时，其显示可能取决于他已经安装的字体。如果他安装的字体与你的字体不同，那么文档看上去会大不相同。使用CSS设计的文档也是如此。

涉及到各种繁杂的字体变形时，如粗体或斜体文本，字体命名的问题就更是混乱。大多数人都知道，斜体文本看上去很像，但是很少有人能解释它与倾斜文本有什么区别，甚至不知道二者之间存在区别。Slanted并不是斜体风格（italic-style）文本唯一的别名，例如，你可能还会看到*oblique*、*incline*（或*inclined*）、*cursive*和*kursiv*等等字眼。因此，一种字体可能有一个TimesItalic变形，而另一种字体可能使用GeorgiaOblique作为

变形。尽管这两种字体实际上就相当于Times和Georgia字体的“斜体形式”，但是它们的“称呼”有很大不同。类似地，字体变形词*bold*、*black*和*heavy*可能表示同一个意思，也可能不同。

CSS试图为所有这些字体问题提供一些解决机制，不过它不能提供一个全面的解决方案。CSS字体处理中最复杂的部分是字体系列（font-family）匹配和字体加粗（font-weight）匹配，其次是字体大小（font-size）计算。CSS中与字体有关的方面还包括字体风格（如斜体）和字体变形（如小型大写字母）；相对而言，这些方面都比较直接。字体样式的所有这些方面都集中到一个属性，即`font`，本章后面将讨论这个属性。首先，先来讨论字体系列，因为在为文档选择适当的字体时这是最基本的一步。

字体系列

前面讨论过，实际上相同的字体可能有很多不同的称呼，不过CSS迈出了勇敢的一步，力图帮助用户代理把这种混乱状况理清楚。毕竟，我们所认为的“字体”可能由许多字体变形组成，分别用来描述粗体、斜体文本，等等。例如，你可能已经对字体Times很熟悉。不过，Times实际上是多种变形的一个组合，包括TimesRegular、TimesBold、TimesItalic、TimesOblique、TimesBoldItalic、TimesBoldOblique，等等。Times的每种变形都是一个具体的字体风格（font face），而我们通常认为Times是所有这些变形字体的一个组合。换句话说，Times实际上是一个字体系列（font family），而不只是单个的字体，尽管我们大多数人都认为字体就是某一种字体。

除了各种特定字体系列外（如Times、Verdana、Helvetica或Arial），CSS还定义了5种通用字体系列。

Serif字体

这些字体成比例，而且有上下短线。如果字体中的所有字符根据其不同大小有不同的宽度，则称该字体是成比例的。例如，小写*i*和小写*m*的宽度就不同。上下短线是每个字符笔划末端的装饰，如小写*l*顶部和底部的短线，或大写*A*两条“腿”底部的短线。serif字体的例子包括Times、Georgia和New Century Schoolbook。

Sans-serif字体

这些字体是成比例的，而且没有上下短线。sans-serif字体的例子包括Helvetica、Geneva、Verdana、Arial和Univers。

Monospace字体

Monospace字体不是成比例的。它们通常用于模拟打字机打出的文本、老式点阵打印机的输出，甚至更老式的视频显示终端。采用这些字体，每个字符的宽度都完全

相同，所以小写的*i*与小写的*m*有相同的宽度。这些字体可能有上下短线，也可能没有。如果一个字体的字符宽度完全相同，则归类为 monospace 字体，而不论是否有上下短线。monospace字体的例子包括Courier、Courier New和Andale Mono。

Cursive 字体

这些字体试图模仿人的手写体。通常，它们主要由曲线和 serif 字体中没有的笔划装饰组成。例如，大写*A*在其左腿底部可能有一个小弯，或者完全由花体部分和小的弯曲部分组成。cursive 字体的例子包括 Zapf Chancery、Author 和 Comic Sans。

Fantasy 字体

这些字体无法用任何特征来定义，只有一点是确定的，那就是我们无法很容易地将其划归到任何一种其他的字体系列当中。这样的字体包括 Western、Woodblock 和 Klingon。

理论上讲，用户安装的任何字体系列都会落入到上述某种通用系列当中。但实际上可能并非如此，不过例外情况（如果有的话）往往很少。

使用通用字体系列

可以使用属性 `font-family` 在文档中采用上述任何字体系列。

font-family	
值：	[[<family-name> | <generic-family>],]* [<family-name> | <generic-family>] | inherit
初始值：	用户代理指定的值
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

如果你希望文档使用一种sans-serif字体，但是你并不关心是哪一种具体字体，以下就是一个合适的声明：

```
body {font-family: sans-serif;}
```

这样用户代理就会从 sans-serif 字体系列选择一个字体（如 Helvetica），并将其应用到 `body` 元素。因为有继承，这种字体选择还将应用到 `body` 元素中包含的所有元素，当然，除非有一种更特定的选择器将其覆盖。

只使用这些通用系列（而不做其他指定），创作人员就能创建相当复杂的样式表，以下规则的显示见图 5-1 所示：

```
body {font-family: serif;}
h1, h2, h3, h4 {font-family: sans-serif;}
code, pre, tt, span.input {font-family: monospace;}
p.signature {font-family: cursive;}
```

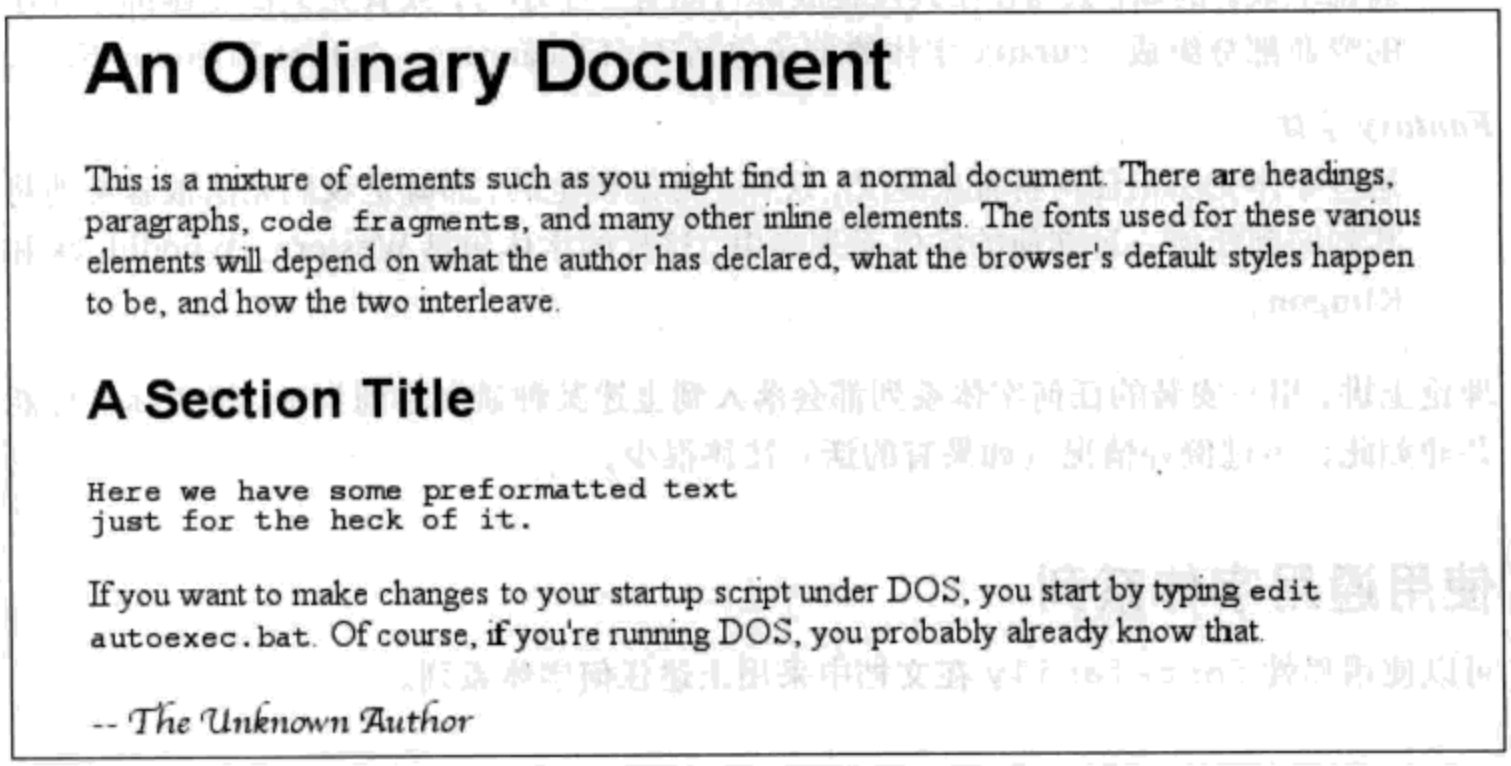

图5-1：不同字体系列

因此，文档大部分会采用某种 serif 字体（包括大多数段落），如 Times，但 class 为 signature 的段落除外，这些段落会用一种 cursive 字体显示，如 Author。1～4 级标题都采用 sans-serif 字体，如 Helvetica，而元素 code、pre、tt 和 span.input 将使用某种 monospace 字体，如 Courier —— 很巧合，本书中的这些元素通常也正是这样显示的。

指定字体系列

另一方面，创作人员对于文档或元素的显示中使用何种字体可能有一些更特定的取向。类似地，用户也可能想创建一个用户样式表，定义所有文档显示中使用的具体字体。不论哪一种情况，还是要使用 font-family 属性。

假设目前所有 h1 都应当使用 Georgia 字体。对此最简单的规则如下：

```
h1 {font-family: Georgia;}
```

这会使显示文档的用户代理对所有 h1 都使用 Georgia 字体，如图 5-2 所示。

A Heading-1 Element

图5-2：使用Georgia字体的h1元素

当然，这个规则假设用户代理上Georgia字体可用。如果Georgia字体不可用，用户代理就根本不能使用这个规则。它不会忽略这个规则，但是如果无法找到一个名叫Georgia的字体，它可能只能使用用户代理的默认字体来显示h1元素，除此以外什么也不做。

不过，不必万念俱灰。通过结合特定字体名和通用字体系列，可以创建与你预想的完全相同（或者至少很接近）的文档。再来看前面的例子，以下标记告诉一个用户代理使用Georgia（如果可用），如果Georgia字体不可用，则使用另外一种serif字体。

```
h1 {font-family: Georgia, serif;}
```

如果读者没有安装Georgia字体，但是安装了Times字体，用户代理就可能对h1元素使用Times。尽管Times与Georgia并不完全匹配，但至少足够接近。

出于这个原因，强烈建议在所有font-family规则中都提供一个通用字体系列。这样一来，就提供了一条后路，在用户代理无法提供与规则匹配的特定字体时，就可以选择一个候选字体。这种候补策略很有帮助，因为在一个跨平台环境中，将无法知道谁安装了什么字体。没错，世界上所有Windows机器都可能安装了Arial和Times New Roman字体，但是有些Macintosh并没有（特别是较老的机器），Unix机器可能也是如此。相反地，尽管在所有现代Macintosh机器上都安装了MarkerFelt和Charcoal，但Windows和Unix用户不太可能安装其中某种字体，同时安装这两种字体的更是少之又少。以下是另外一些例子：

```
h1 {font-family: Arial, sans-serif;}
h2 {font-family: Charcoal, sans-serif;}
p {font-family: TimesNR, serif;}
address {font-family: Chicago, sans-serif;}
```

如果你对字体很熟悉，显示一个给定元素时可能会想到很多类似的字体。假设你希望一个文档中的所有段落都使用Times显示，不过也接受TimesNR、Georgia、New Century Schoolbook和New York（所有这些都是serif字体）。首先，先确定这些字体的优先顺序，然后用逗号把它们连起来：

```
p {font-family: Times, TimesNR, 'New Century Schoolbook', Georgia,
      'New York', serif;}
```

根据这个列表，用户代理会按所列的顺序查找这些字体。如果列出的所有字体都不可用，就会简单地选择一种可用的serif字体。

使用引号

你可能注意到了，前面的例子中出现了单引号，这在以前没有出现过。只有当一个字体名中有一个或多个空格（如New York），或者如果字体名包括#或$之类的符号，才需要在font-family声明中加引号。在这些情况下，整个字体名应当用引号括起，这样用户代理才能搞清楚字体名到底是什么（你可能认为有逗号就足够了，但并非如此）。因此，名为Karrank%的字体就应当加引号：

```
h2 {font-family: Wedgie, 'Karrank%', Klingon, fantasy;}
```

如果没有引号，尽管规则的余下部分还会得到处理，但用户代理有可能忽略这个特定的字体名。注意，根据CSS2.1规范，包含符号的字体名并不一定要加引号，但这是一种推荐做法，它类似于CSS规范中描述的“最佳实践”。类似地，对包含空格的字体名也建议加引号。可以看到，唯一必须加引号的是与所接受关键字匹配的字体名。因此，如果需要一种名为“cursive”的字体，就必须加引号。

显然，字体名中如果只包含一个词，而且这个词与font-family 的任何关键字都不冲突，就不需要加引号，通用字体系列名（serif、monospace等等）在指示具体的通用系列时就不能加引号。如果将一个通用名用引号引起，用户代理就会认为你需要一个与此同名的特定字体（例如，serif），而不是一个通用字体系列。

至于使用单引号还是双引号，这两种都是可以接受的。要记住，如果把一个font-family规则放在style属性中，则需要使用该属性本身未曾使用的那种引号。因此，如果使用双引号将font-family规则括起，就必须在规则内部使用单引号，如以下标记所示：

```
p {font-family: sans-serif;}  /* sets paragraphs to sans-serif by default
*/
<!-- the next example is correct (uses single-quotes) -->
<p style="font-family: 'New Century Schoolbook', Times, serif;">...</p>
<!-- the next example is NOT correct (uses double-quotes) -->
<p style="font-family: "New Century Schoolbook", Times, serif;">...</p>
```

在这种情况下如果使用双引号，它们会与属性语法冲突，如图5-3所示。

Greetings! This paragraph is supposed to use either 'New Century Schoolbook', Times, or an alternate serif font for its display.

Greetings! This paragraph is also supposed to use either 'New Century Schoolbook', Times, or an alternate serif font for its display.

图5-3：不正确使用引号的后果

字体加粗

尽管你可能还没有意识到，但实际上你已经对字体加粗很熟悉了，粗体文本就是字体加粗的一个很常见的例子。利用font-weight属性，CSS允许对字体加粗有更多控制，至少在理论上是这样。

font-weight

值：	normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | inherit
初始值：	normal
应用于：	所有元素
继承性：	有
计算值：	数字值（如100等等），或某个数字值加上某个相对数（bolder或lighter）

一般来讲，字体加粗越重，字体看上去就越黑且“越粗”。标示加粗字体的方法有很多。例如，名为Zurich的字体系列有很多变形，如Zurich Bold、Zurich Black、Zurich UltraBlack、Zurich Light和Zurich Regular。这些变形都使用相同的基本字体，但是每个变形都有不同的加粗。

现在假设你希望一个文档使用Zurich，但是你想使用所有这些不同的加粗级别。可以直接通过font-family属性来指定，不过最好不要这样做。另外，如果必须写下面这样的样式表也没意思：

```
h1 {font-family: 'Zurich UltraBlack', sans-serif;}
h2 {font-family: 'Zurich Black', sans-serif;}
h3 {font-family: 'Zurich Bold', sans-serif;}
h4, p {font-family: Zurich, sans-serif;}
small {font-family: 'Zurich Light', sans-serif;}
```

写这样一个样式表显然很乏味，不仅如此，只有当所有人都安装了这些字体时，样式表才能起作用，但更有可能的情况是，大多数人都没有安装所有这些字体。更有意义的做法是为整个文档指定一个字体系列，然后为不同元素指定不同的加粗。从理论上讲，可以对font-weight属性使用不同的值来做到这一点。以下是一个很简单的font-weight声明：

```
b {font-weight: bold;}
```

很简单，这个声明指出b元素应当使用一种粗体字体显示；换句话说，要用一种比文档正常字体更粗的字体来显示。当然，我们经常见到这种情况，因为b确实会让文本加粗。

不过，实际情况是使用字体的一种加粗变形来显示b元素。因此，如果使用Times显示一个段落，而且其中一部分是粗体，那么实际上当前使用了同一字体的两种变形：Times和TimesBold。常规的文本使用Times显示，加粗的文本使用TimesBold显示。

加粗如何起作用

为了理解用户代理如何确定一个给定字体变形的加粗度，我们暂不考虑加粗如何继承，首先介绍关键字100~900，这样最容易理解。定义这些数字关键字是为了映射字体设计中的一个很常见的特性，即为字体指定了9级加粗度。例如OpenType采用了一个数值梯度，其中包含9个值。如果一个字体内置了这些加粗级别，那么这些数字就直接映射到预定义的级别，100对应最细的字体变形，900对应最粗的字体变形。

实际上，这些数字本身并没有固有的加粗度。CSS规范只是指出，每个数对应一个加粗度，它至少与前一个数指定的加粗度相同。因此，100、200、300和400可能都映射到同样的较细变形；500和600可能对应到同样的较粗字体变形；而700、800和900可能都生成同样的很粗的字体变形。只要一个关键字对应的变形不会比前一个关键字所对应变形更细，就都是允许的。

一般地，这些数都被定义为与某个常用变形名等价（先不考虑font-weight的其他值）。400定义为等价于normal，700对应于bold。其他数不对应font-weight的任何其他值，不过它们可能对应于常用变形名。如果有一个字体变形标为Normal、Regular、Roman或Book，就会为之指定400，而标为Medium的变形会指定为500。不过，如果一个标为Medium的变形是唯一可用的变形，它不会指定为500而会是400。

如果给定的字体系列中定义的加粗度少于9种，用户代理还必须多做一些工作。在这种情况下，它必须以一种预定的方式填补这些“空白”：

- 如果未指定值500的加粗度，其字体加粗与400的相应加粗相同。
- 如果未指定300 的加粗度，则为之指定下一个比400更细的变形。如果没有可用的较细变形，为300指定的变形等同于400的相应变形。在这种情况下，通常是Normal或Medium。这种方法同样适用于200和100。
- 如果未指定600的加粗度，会为之指定下一个比400更粗的变形。如果没有可用的较粗变形，为600指定的变形则等同于500的相应变形。这种方法同样适用于700、800和900。

为了更清楚地说明这种加粗机制，下面来看指定字体加粗的三个例子。在第一个例子中，假设字体系列Karrank%是一种OpenType字体，所以它已经定义了9个加粗度。在这种情况下，数字已经对应到各个加粗级别，关键字normal和bold分别对应数字400和700。

第二个例子中，我们考虑了字体系列Zurich，这个字体系列在本节最开始讨论过。假设已经为其变形指定了不同的 font-weight 数字值，见表 5-1 所示。

表5-1：特定字体系列的假想加粗指定

字体	指定的关键字	指定的数字
Zurich Light		100,200,300
Zurich Regular	normal	400
Zurich Medium		500
Zurich Bold	bold	600,700
Zurich Black		800
Zurich UltraBlack		900

前三个数字值被指定为最细的字体加粗。不出所料，对于标有 Regular 的变形，相应的关键字为normal，数字值为400。因为在此有一个Medium字体，它指定为数字值500。这里没有为 600 指定任何变形，因此它映射到 Bold 字体，700 也指定为这个字体变形，相应的关键字是 bold。最后，800 和 900 分别指定为 Black 和 UltraBlack 变形。注意，只有已经指定了有最大两级加粗的字体时，800 和 900 才会分别指定为 Black 和 UltraBlack 变形。否则，用户代理可能会将其忽略，而将 800 和 900 指定为 Bold 字体，或者可能会把它们都指定为某种 Black 变形。

最后，来考虑 Times 的一种缩减版本。表 5-2 中只有两种加粗变形：TimesRegular 和 TimesBold。

表5-2：Times的假想加粗指定

字体	指定的关键字	指定的数字
TimesRegular	normal	100,200,300,400,500
TimesBold	bold	600,700,800,900

当然，关键字normal和bold的指定很直接。至于数字，100~300 被指定为 Regular 字体，因为没有更细的字体了。400 按预想的那样被指定为 Regular，但是 500 呢？它也被指定为 Regular（或 normal）字体，因为没有可用的 Medium 字体，因此它与 400 被

指定为同样的字体。至于余下的数字，与往常一样，700被指定为bold，由于没有一种更粗的字体，800和900只能被指定到下一个较细的字体，即Bold字体。最后，600要被指定到下一个更粗的字体，当然这就是Bold字体。

font-weight是可以继承的，所以如果将一个段落设置为bold：

```
p.one {font-weight: bold;}
```

那么它的所有子元素都会继承这个加粗度，如图5-4所示。

Within this paragraph we find some *italicized text*, a bit of underlined text, and the occasional stretch of hyperlinked text for our viewing pleasure.

图5-4：继承的font-weight

这很平常，但是如果你使用最后两个值（bolder和lighter），情况会变得很有意思，有必要对这两个值详细讨论。一般来讲，这些关键字的效果正如你期望的那样：它们会让文本比其父元素的字体加粗更粗或更细。首先，我们来考虑bolder。

让字体更粗

如果将一个元素的加粗设置为bolder，用户代理首先必须确定从父元素继承的font-weight值。然后选择一个数，它对应于比所继承值更粗的一个字体加粗，而且在满足这个条件的所有数中，要选择最小的一个数。如果没有可用的字体，用户代理会把该元素的字体加粗设置为下一个更大的数字值，除非这个值已经是900，如果确实如此，加粗则保持为900。因此，可能会遇到以下情况，如图5-5所示：

```
p {font-weight: normal;}
p em {font-weight: bolder;}  /* results in bold text, evaluates to '700' */

h1 {font-weight: bold;}
h1 b {font-weight: bolder;}  /* if no bolder face exists, evaluates to '800' */

div {font-weight: 100;} /* assume 'Light' face exists; see explanation */
div strong {font-weight: bolder;} /* results in normal text, weight '400' */
```

在第一个例子中，用户代理将加粗从normal上移为bold；按数字来讲，它从400跳至700。在第二个例子中，h1文本已经设置为bold。如果没有更粗的字体，用户代理就会把h1中b文本的加粗设置为800，因为700（与bold等价的数）的下一个数就是800。由于800和700被指定为同一个字体，所以正常的h1文本和粗体h1文本之间看上去没有任何差别，不过其加粗确实不同。

Within this paragraph we find some ***emphasized text***.

This H1 contains bold text.

Meanwhile, this DIV tag has some **strong text** but it shouldn't look much different, at least in terms of font weight.

图5-5：设置为更粗的文本

在最后一个例子中，段落的字体加粗设置为最细，在此假设存在Light变形。另外，这个字体系列中的其他字体是Regular和Bold。段落中的任何em文本都会计算为normal，因为这是该字体系列中下一个最粗的字体。不过，如果字体中只有Regular和Bold会怎么样呢？在这种情况下，声明将计算如下：

```
/*   assume only two faces for this example: 'Regular' and 'Bold'    */
p {font-weight: 100;}   /* looks the same as 'normal' text */
p span {font-weight: bolder;}   /* maps to '700' */
```

可以看到，加粗数100被指定为normal字体，但是font-weight值还是100。因此，p元素中包含的所有span文本都会继承这个值100，然后计算下一个最粗的字体，即数字加粗值为700的Bold字体。

下面增加另外两个规则以及一些标记，进一步说明所有这些是如何工作的（结果见图5-6所示）：

```
/*   assume only two faces for this example: 'Regular' and 'Bold'    */
p {font-weight: 100;}   /* looks the same as 'normal' text */
p span {font-weight: 400;}   /* so does this */
strong {font-weight: bolder;}   /* even bolder than its parent */
strong b {font-weight: bolder;}   /*bolder still */

<p>
This paragraph contains elements of increasing weight: there is a
<span>span element that contains a <strong>strongly emphasized
element and a <b>boldface element</b></strong></span>.
</p>
```

This paragraph contains elements of increasing weight: there is a span element that contains a **strongly emphasized element, and that contains a boldface element**.

图5-6：加粗度上移

在最后两个嵌套元素中，font-weight的计算值增加了，因为这里使用了关键字

bolder。如果把段落中的文本替换为表示各元素font-weight的数字，可以得到如下的结果：

```
<p>
100 <span> 400 <strong> 700 <b> 800 </b></strong></span>.
</p>
```

前面两个加粗提升很大，因为它们表示从100跃至400，以及从400跃至bold(700)。从700向上再没有更粗的字体了，所以用户代理只是将font-weight值向上移一步，移到800。另外，如果你想在b元素中插入一个strong元素，会得到如下的结果：

```
<p>
100 <span> 400 <strong> 700 <b> 800 <strong> 900
</strong></b></strong></span>.
</p>
```

如果再有另一个b元素插入到最内层strong元素中，其加粗也是900，因为font-weight值不可能比900更高。假设只有两种可用的字体风格，那么文本要么显示为Regular要么显示为Bold，如图5-7所示：

```
<p>
regular <span> regular <strong> bold <b> bold
<strong> bold </strong></b></strong></span>.
</p>
```

regular regular **bold bold bold** .

图5-7：有说明的加粗显示

让字体更细

可以预料，lighter的做法完全一样，只不过它会导致用户代理将加粗度下移而不是上移。对前面的例子稍做修改，可以很清楚地看到这一点：

```
/*   assume only two faces for this example: 'Regular' and 'Bold'   */
p {font-weight: 900;}   /* as bold as possible, which will look 'bold' */
p span {font-weight: 700;}   /* this will also be bold */
strong {font-weight: lighter;}   /* lighter than its parent */
b {font-weight: lighter;}   /* lighter still */

<p>
900 <span> 700 <strong> 400 <b> 300 <strong> 200
</strong></b></strong></span>.
</p>
<!-- ...or, to put it another way... -->

<p> bold <span> bold <strong> regular <b> regular
```

```
<strong> regular </strong></b></strong></span>.
</p>
```

尽管直观上看这完全不对劲，不过暂且不考虑这个问题。从图5-8可以看到主段落文本的加粗为900。如果将strong文本设置为lighter，它会计算为下一个较细的字体，即regular字体，或数字值400(与normal相同)。再向下一级是300，这等同于normal，因为不存在更细的字体。从300向下，用户代理一次只能将加粗值下移一步，直到达到100（本例中没有减到100)。第二个段落显示了哪些文本是bold，哪些是regular。

900 700 400 300 200 .

bold bold regular regular regular .

图5-8：使文本更细

字体大小

确定字体大小的方法对我们来说既很熟悉，也有很大不同。

font-size	
值：	xx-small | x-small | small | medium | large | x-large | xx-large | smaller | larger | <length> | <percentage> | inherit
初始值：	medium
应用于：	所有元素
继承性：	有
百分数：	根据父元素的字体大小来计算
计算值：	绝对长度

与font-weight关键字bolder和lighter的方式类似，属性font-size也有两个相对大小关键字：larger和smaller。类似于相对字体加粗，这些关键字会导致font-size的计算值上移或下移，在讨论larger和smaller之前需要先了解这一点。不过，首先需要分析如何确定字体的大小。

实际上，font-size属性与你看到的实际字体大小之间的具体关系由字体的设计者来确定。这种关系设置为字体本身中的一个em方框（有人也称之为em框）。这个em方框（以及相应的字体大小）不一定指示字体中字符建立的任何边界。相反，它指示如果没有额外行间距（CSS中的line-height）设置字体时基线间的距离。某种字体的字符可能比默认的基线间距离要高，这是完全有可能的。出于这种原因，定义字体时可能要求所有字符都小于其em方框，很多字体就是这样做的。图5-9显示了一些假想的例子。

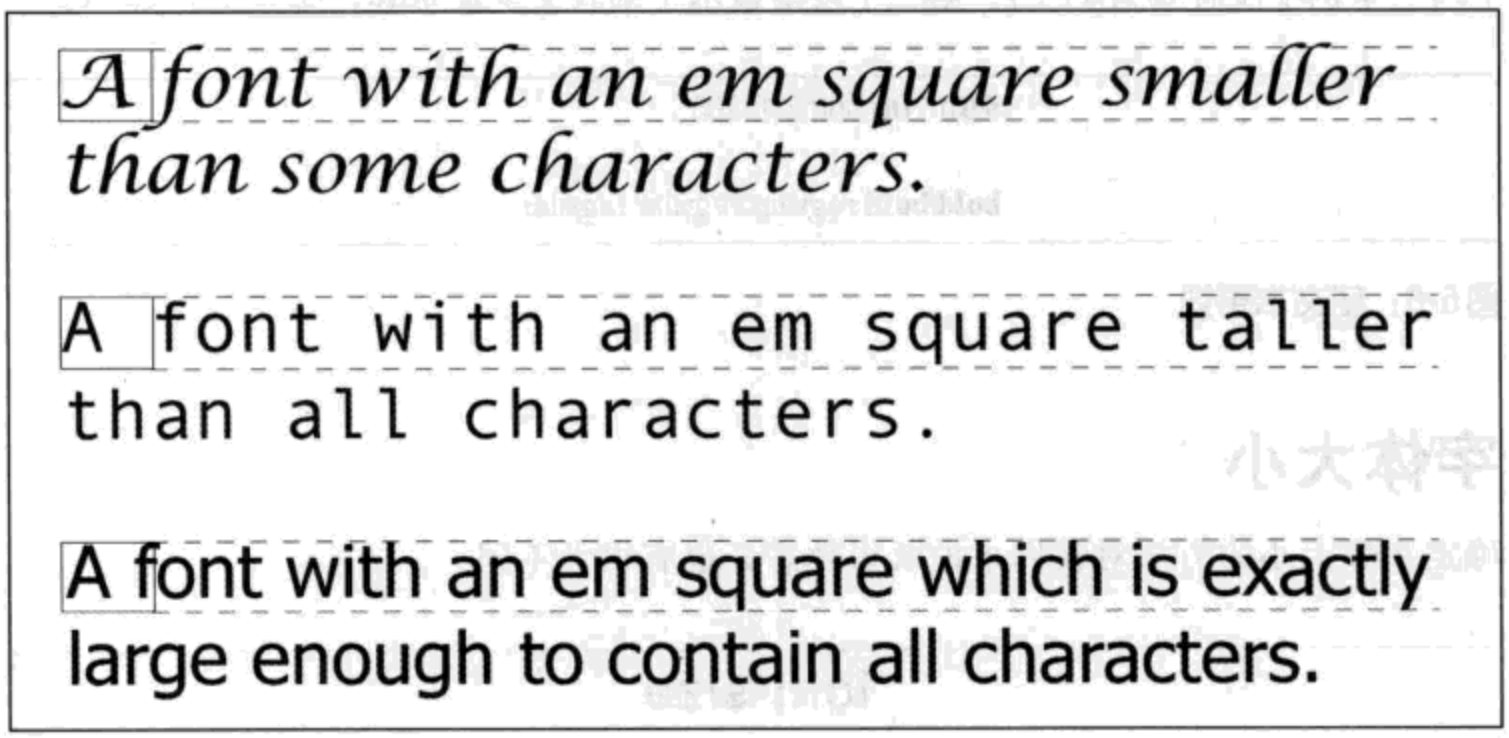

图5-9：字体字符和em方框

因此，font-size的作用是为给定字体的em框提供一个大小，而不能保证实际显示的字符就是这种大小。

绝对大小

有了以上了解，现在来看绝对大小关键字。font-size有7个绝对大小值：xx-small、x-small、small、medium、large、x-large和xx-large。这些关键字并没有明确地定义，而是相对地来定义，如图5-10所示。

```
p.one {font-size: xx-small;}
p.two {font-size: x-small;}
p.three {font-size: small;}
p.four {font-size: medium;}
p.five {font-size: large;}
p.six {font-size: x-large;}
p.seven {font-size: xx-large;}
```

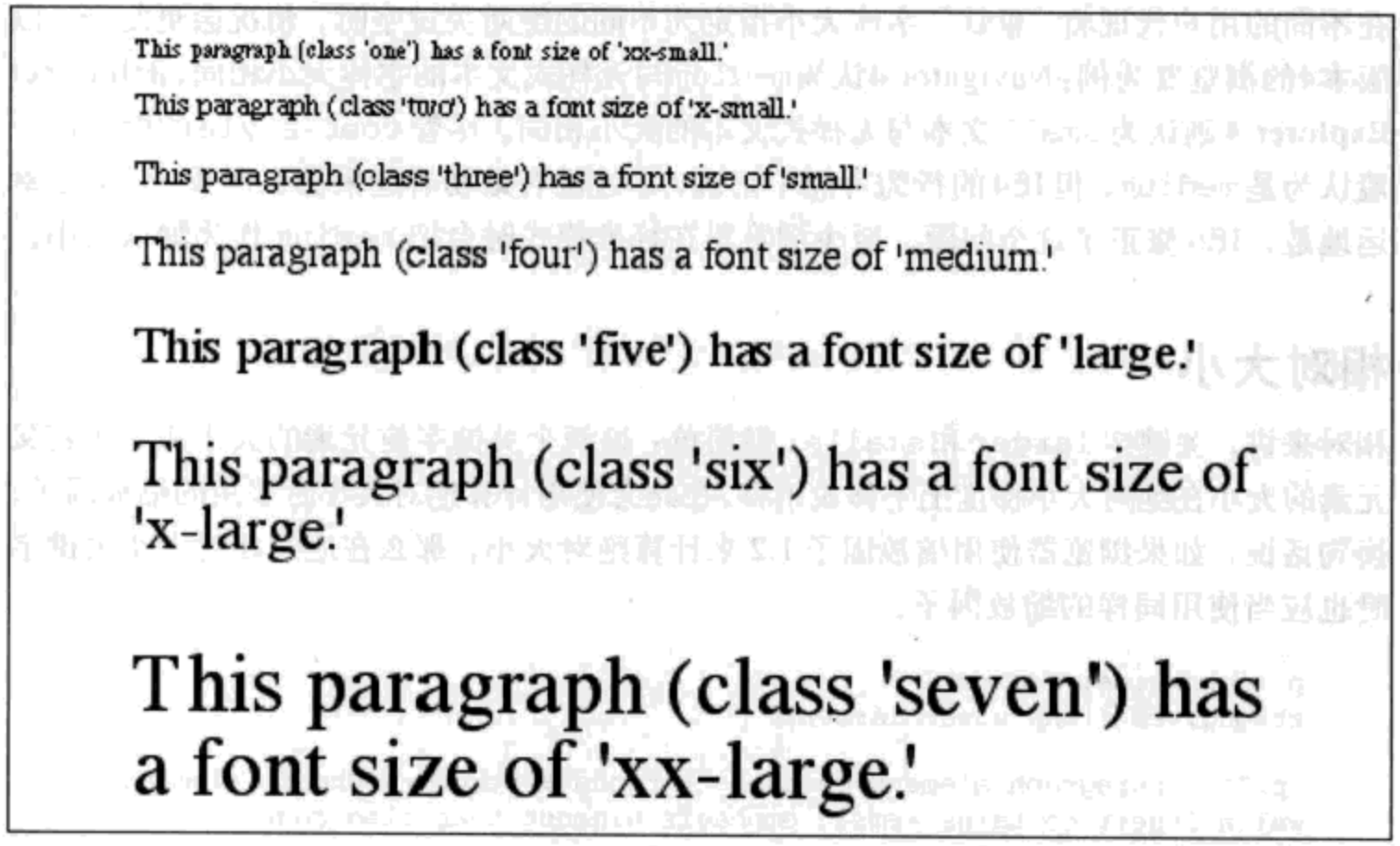

图5-10：绝对字体大小

根据CSS1规范，一个绝对大小与下一个绝对大小之间的差别（或缩放因子）应当是向上1.5，或向下0.66。因此，如果medium等于10px，那么large就应当等于15px。另一方面，缩放因子不一定非得是1.5，不仅因为对于不同的用户代理缩放因子可能不同，此外还有一个原因：CSS2中缩放因子可能介于1.0~1.2之间。

假设medium等于16px，在此基础上，对于不同的缩放因子，可以得到表5-3所示的绝对大小（当然，下面的值是近似的）。

表5-3：缩放因子转换为像素

关键字	缩放：1.5	缩放：1.2
xx-small	5px	9px
x-small	7px	11px
small	11px	13px
medium	16px	16px
large	24px	19px
x-large	36px	23px
xx-large	54px	28px

在不同的用户代理将“默认”字体大小指定为不同的绝对关键字时，情况会更复杂。以版本4的浏览器为例：Navigator 4认为medium与无样式文本的字体大小相同，而Internet Explorer 4则认为small文本与无样式文本的大小相同。尽管font-style的默认值一般认为是medium，但IE4的行为可能不正确，不过没有最初看起来那么严重（注1）。幸运地是，IE6修正了这个问题，至少浏览器在标准模式时会把medium作为默认大小。

相对大小

相对来讲，关键字larger和smaller很简单：这两个关键字使元素的大小相对于其父元素的大小在绝对大小梯度上上移或下移，在此会使用计算绝对大小时采用的缩放因子。换句话说，如果浏览器使用缩放因子1.2来计算绝对大小，那么在应用相对大小关键字时也应当使用同样的缩放因子：

```
p {font-size: medium;}
strong, em {font-size: larger;}

<p>This paragraph element contains <strong>a strong-emphasis element
which itself contains <em>an emphasis element that also contains
<strong>a strong element.</strong></em></strong></p>

<p> medium <strong>large <em> x-large
<strong>xx-large</strong></em></strong></p>
```

不同于加粗的相对值，相对大小值不必限制在绝对大小范围内。因此，一个字体的大小可以超过xx-small和xx-large的大小。例如：

```
h1 {font-size: xx-large;}
em {font-size: larger;}

<h1>A Heading with <em>Emphasis</em> added</h1>
<p>This paragraph has some <em>emphasis</em> as well.</p>
```

如图5-11所示，h1元素中的强调文本比xx-large稍微大一点。缩放的程度由用户代理决定，往往就是缩放因子1.2。当然，段落中的em文本会在绝对大小梯度上上移一步(large)。

注1：注意，有7个绝对大小关键字，因为有7个字体大小（例如，<font size="5">）。由于一般的默认字体大小历来都是3，所以CSS绝对大小关键字中的第3个值就指示默认字体大小。因为第3个关键字刚好是small，所以Explorer会有这样的行为。

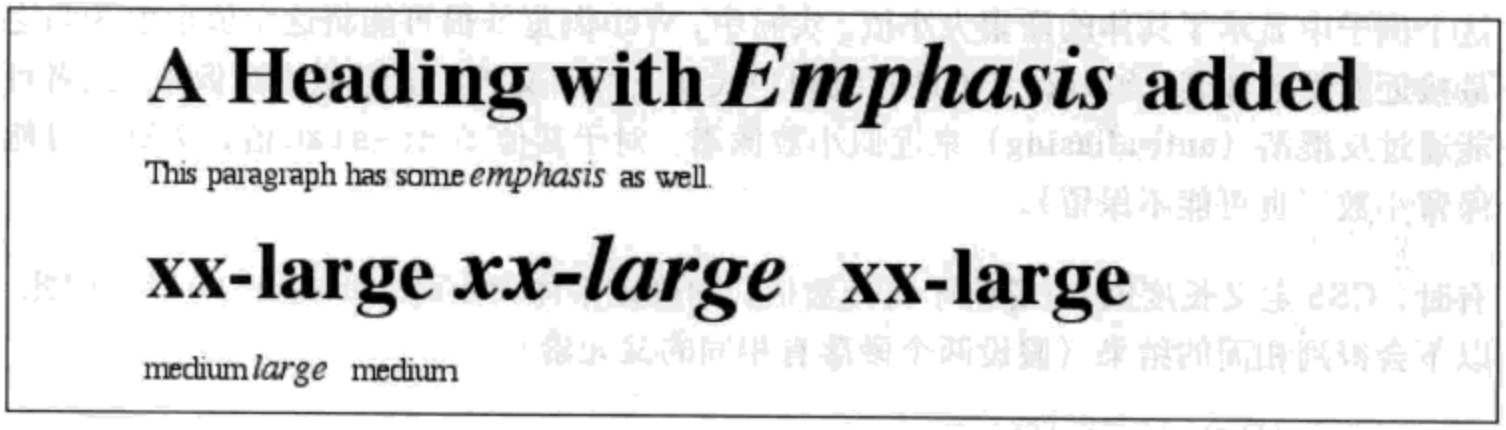

图5-11：超出绝对大小边界的相对字体大小

注意：用户代理不需要将字体大小增加或减少到超出绝对大小关键字的限制。

百分数和大小

在某种程度上讲，百分数值与相对大小关键字很相似。百分数值总是根据从父元素继承的大小来计算。不同于相对大小关键字，百分数允许对计算的字体大小有更细的控制。考虑以下例子，如图 5-12 所示：

```
body {font-size: 15px;}
p {font-size: 12px;}
em {font-size: 120%;}
strong {font-size: 135%;}
small, .fnote {font-size: 75%;}

<body>
<p>This paragraph contains both <em>emphasis</em> and <strong>strong
emphasis</strong>, both of which are larger than their parent element.
The <small>small text</small>, on the other hand, is smaller by a quarter.</p>
<p class="fnote">This is a 'footnote' and is smaller than regular text.</p>

<p> 12px <em> 14.4px </em> 12px <strong> 16.2px </strong>  12px
<small> 9px </small> 12px </p>
<p class="fnote"> 10.5px </p>
</body>
```

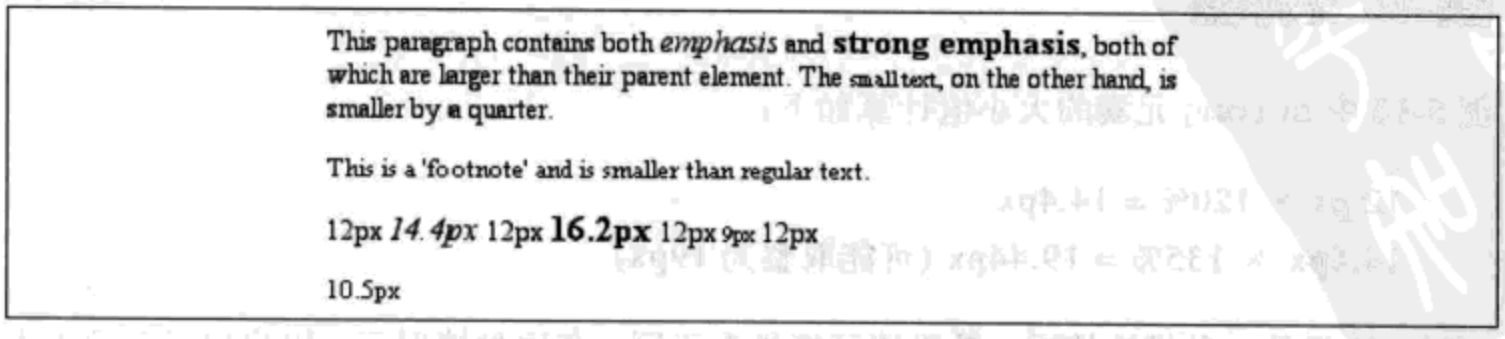

图5-12：考虑百分数

这个例子中显示了具体的像素大小值。实际中，Web浏览器很可能将这个值取整为与之最接近的整数像素，如14px，不过高级用户代理在打印文档时会近似小数像素，或者可能通过反混淆（anti-aliasing）来近似小数像素。对于其他font-size值，浏览器可能保留小数（也可能不保留）。

有时，CSS定义长度值em等价于百分数值，即确定字体大小时1em等于100%。因此，以下会得到相同的结果（假设两个段落有相同的父元素）：

```
p.one {font-size: 166%;}
p.two {font-size: 1.6em;}
```

在使用em度量时，会应用百分数的相同规则，如计算大小的继承规则等等。

字体大小和继承

尽管在CSS中font-size是可以继承的，不过继承的是计算值而不是百分数，如图5-12所示。因此，strong元素继承的值是12px，根据声明值135%，继承的这个值修改为16.2px（这很可能取整为16px）。对于“脚注”段落，百分数相对于由body元素继承的font-size值来计算，即15px。将这个值乘以75%就得到11.25px。

与相对大小关键字一样，百分数可以积累。因此，以下标记显示为如图5-13所示：

```
p {font-size: 12px;}
em {font-size: 120%;}
strong {font-size: 135%;}

<p>This paragraph contains both<em>emphasis and <strong>strong
emphasis</strong></em>, both of which are larger than the paragraph text. </p>

<p> 12px <em>14.4px <strong> 19.44px </strong></em> 12px  </p>
```

This paragraph contains both *emphasis and* ***strong emphasis***, both of which are larger than the paragraph text.

12px *14.4px* ***19.44px*** 12px

图5-13：继承问题

图5-13中strong元素的大小值计算如下：

12 px × 120% = 14.4px

14.4px × 135% = 19.44px (可能取整为19px)

不过，这只是一种候选情况，最终值可能稍有不同。在这种情况下，用户代理对像素大

小进行取整，取整后的值再由子元素正常继承。尽管根据规范这种行为可能并不正确，不过我们假设用户代理会这么做。因此，会有：

12px × 120% = 14.4px [14.4px ≈ 14px]
14px × 135% = 18.9px [18.9px ≈ 19px]

如果有人认为用户代理每一步都完成取整，那么这个计算和前一个计算的最终结果是一样的：都是19像素。不过，如果与更多的百分数相乘，取整错误就会开始积累。

缩放失控的问题还可能朝着另一个方向发展。下面来考虑一个文档，这个文档中只有一系列无序列表，其中很多列表嵌套在另外的一些列表中。有些列表的嵌套多达4层。假设对这样一个文档应用以下规则，可以想象一下最后的效果：

```
ul {font-size: 80%;}
```

对于一个4层嵌套，嵌套最深的无序列表的font-size计算值将是顶层列表父元素大小的40.96 %。每个嵌套列表的字体大小都是其父列表字体大小的80%，这就导致一层比一层难读。如果一个文档使用嵌套表来建立布局，也可能发生类似的问题。你可能会把规则写作：

```
td {font-size: 0.8em;}
```

不论怎样，都可能得到一个无法阅读的页面。

使用长度单位

可以使用第4章详细讨论的任何长度值来设置font-size。以下所有font-size声明都是等价的：

```
p.one {font-size: 36pt;}
p.two {font-size: 3pc;}
p.three {font-size: 0.5in;}
p.four {font-size: 1.27cm;}
p.five {font-size: 12.7mm;}
```

图5-14假设用户代理知道显示媒体中每英寸使用多少点。不同的用户代理会做不同的假设，有些基于操作系统，有些基于首选项设置，还有些会基于编写用户代理的程序员的假设。不过，这5行应该始终有相同的大小。因此，尽管结果与现实可能不能完全匹配（例如，p.three的实际大小可能不是半英寸），但是度量彼此之间是一致的。

还有一个值可能与图5-14所示大小相同，即36px，如果显示媒体是72 ppi（pixels-per-inch，每英寸像素数），那么物理距离就可能一样。不过，有这种设置的显示器并不多。大多数显示器都有更高的分辨率，范围为96ppi~120ppi。许多较老的Macintosh Web浏

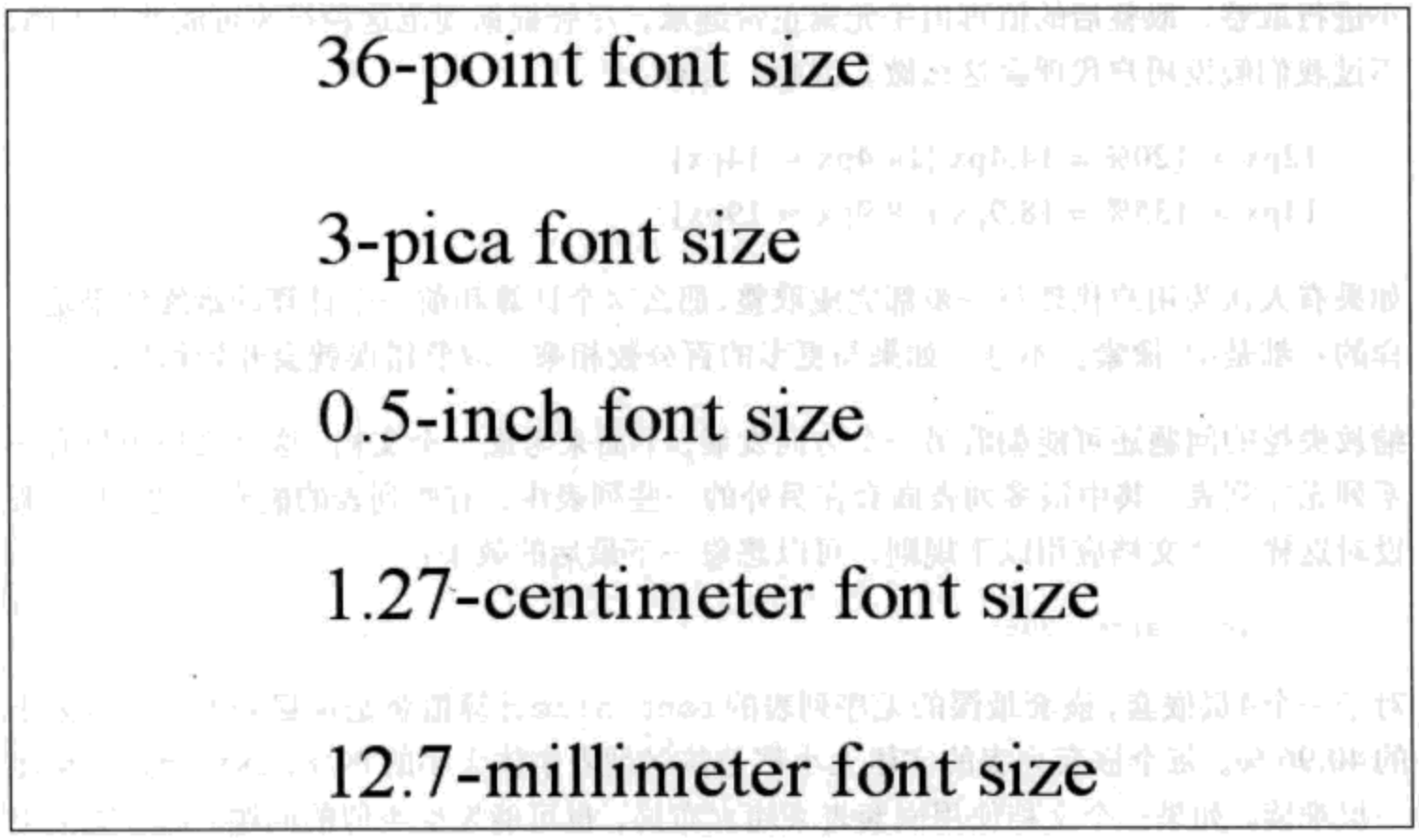

图5-14：不同字体大小

览器把点和像素看成是一样的，所以在这些浏览器上，值14pt和14px看上去可能一样。不过，对于Windows和其他平台（包括Mac OS X）却不是这样。正因如此（这是一个主要原因），所以我们说在文档设计中点很难使用。

由于不同操作系统之间的这些差别，许多创作人员选择用像素值指定字体大小。如果一个Web页面上既有文本又有图像，就很适合采用这种方法，这是因为通过声明font-size: 11px;（或者类似声明），理论上文本可以设置为与页面上的图形元素等高，如图5-15所示。

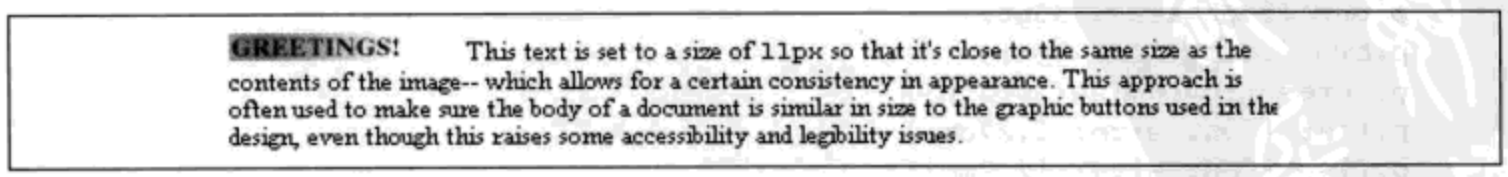

图5-15：用像素大小保持文本和图形等高

对font-size使用像素度量，这当然是用font-size（实际上是用任何长度）得到“一致”结果的一种办法，不过，这里存在一个重要的缺点。Windows Internet Explorer在6.0之前不允许用户轻松地调整用像素设置的文本大小。其他浏览器，包括Mozilla、Netscape 6+、IE5+/Mac、Opera甚至IE7，则允许用户调整文本的大小，而不论文本大小是如何设置的。因此，使用像素来设置文本大小与使用其他方法一样，无法保证总有

同样的大小。本章中讨论的其他方法（如关键字和百分数）都更为健壮（也更为用户友好），因为可以用这些方法在用户默认字体大小的基础上缩放文本。

风格和变形

与本章前面讨论的内容相比，这一节实际上极其简单。这里讨论的属性简单易懂，所以阅读这一部分可能很轻松。首先，我们将讨论font-style，再转向font-variant，最后对字体属性做一个总结。

有风格的字体

font-style很简单：用于在normal文本、italic文本和oblique文本之间进行选择。就这么简单！唯一有点复杂的是要明确italic文本和oblique文本之间的差别，并了解为什么浏览器并不能始终提供选择。

font-style

值：	italic | oblique | normal | inherit
初始值：	normal
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

可以看到，font-style的默认值是normal。这是指“竖直”的文本，可能最好描述为“非斜体或倾斜的文本”。例如，本书中的绝大多数文本都是竖直的。接下来只需对italic文本和oblique文本的差别做一个解释。对此，最容易的办法是参考图5-16，这里很清楚地展示了二者的区别。

基本说来，斜体（italic）是一种单独的字体风格，对每个字母的结构有一些小改动，来反映变化的外观。对于serif字体尤其如此，除了文本字符“有些斜”以外，serifs可以修改为一种斜体字体。与此不同，倾斜（oblique）文本则是正常竖直文本的一个倾斜版本。标为Italic、Cursive和Kursiv的字体总是映射到italic关键字，而oblique总是对应到标为Oblique、Slanted和Incline的字体。

如果想确保文档以你熟悉的方式使用斜体文本，可以编写以下样式表：

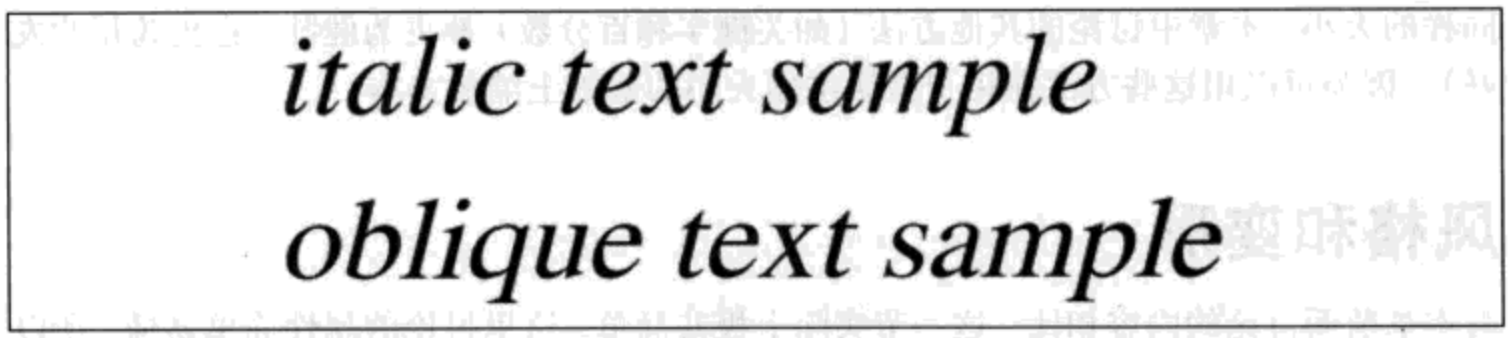

图5-16：斜体和倾斜文本展示

```
p {font-style: normal;}
em, i {font-style: italic;}
```

这些样式使段落像平常一样使用竖直字体，而让 em 和 i 元素一如平常地使用一种斜体字体（italic）。另一方面，可能决定在 em 和 i 之间应该有点区别：

```
p {font-style: normal;}
em {font-style: oblique;}
i {font-style: italic;}
```

如果仔细查看图 5-17，会看到 em 和 i 元素之间并没有明显的区别。实际中，并不是每一种字体都如此复杂，同时有斜体（italic）和倾斜（oblique）字体，甚至即使这两种字体同时存在，也很少有浏览器复杂到足以区别它们。

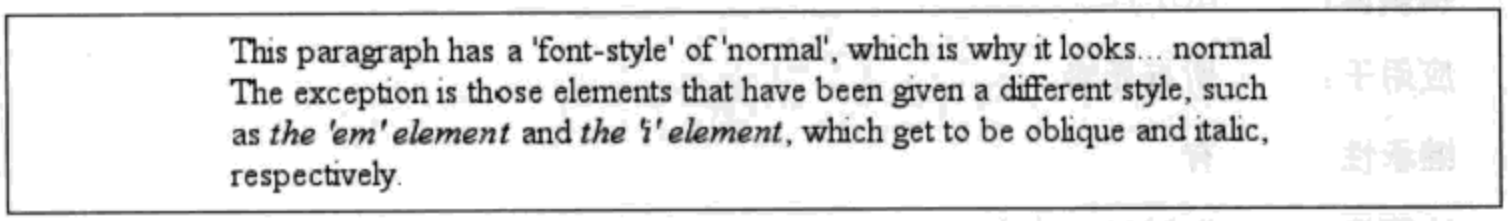

图5-17：更多字体风格

如果存在这些情况，可能要做更多工作。如果没有 Italic 字体，但是有一种 Oblique 字体，则要在需要前者的地方使用后者。如果情况反过来，存在 Italic 字体，但是没有 Oblique 字体，根据规范，用户代理可能不会把后者换成前者。最后，用户代理可能只是计算竖直字体的一个倾斜版本来生成 oblique 字体。实际上，数字世界中通常都会这样做，用一个简单的计算来完成字体的倾斜相当容易。

此外，你可能发现在许多操作系统中声明为 italic 的给定字体可能会从 italic 切换到 oblique，这取决于字体的具体大小。例如，在运行 Classic OS (Mac OS 9)的一个 Macintosh 机器上，Times 的显示如图 5-18 所示，其唯一的区别是大小有一个像素之差。

遗憾的是，对此做不了什么，除非操作系统提供了更好的字体处理，如 Mac OS X 和 Windows XP 就提供了更好的字体处理。通常，italic 和 oblique 字体在 Web 浏览器中看上去完全一样。

This paragraph contains *a stretch of italicized text* within.

This paragraph contains*a stretch of italicized text* within.

图5-18：相同字体、相同风格、不同大小

不过，font-style还是很有用的。例如，一种常用的排版约定要求块引用应当是斜体，而引用中特别强调的文本应当是竖直的。为了达到这种效果，如图5-19所示，应当使用以下样式：

```
blockquote {font-style: italic;}
blockquote em, blockquote i {font-style: normal;}
```

Once upon a time, on a 'net not so far away, someone was heard to say:

> *Of course, workarounds, compatibility charts, and "bug list" pages are just a symptom of the problem, not the solution. Suffice it to say that, in light of all our problems, the only real solution is this:* browsers must become conformant with the CSS specifications. *Otherwise, we'll be stuck with a Web so fragmented it will hurt. A lot.*

图5-19：通过CSS实现的常用排版约定

字体变形

除了大小和风格，字体还可以有变形。CSS提供了一种办法来确定非常常见的变形。

font-variant

值：	small-caps | normal | inherit
初始值：	normal
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

对于font-variant，它只有两个非继承值：默认值normal和small-caps，normal描述正常文本，small-caps要求使用小型大写字母文本。如果你对这种效果还不熟悉，

IT LOOKS SOMETHING LIKE THIS. 小型大写字母既不是一般的大写字母，也不是小写字母，这种字体采用不同大小的大写字母。因此，可能会看到图 5-20 所示的结果。

```
h1 {font-variant: small-caps;}
h1 code, p {font-variant: normal;}

<h1>The Uses of <code>font-variant</code> On the Web</h1>
<p>
The property <code>font-variant</code> is very interesting...
</p>
```

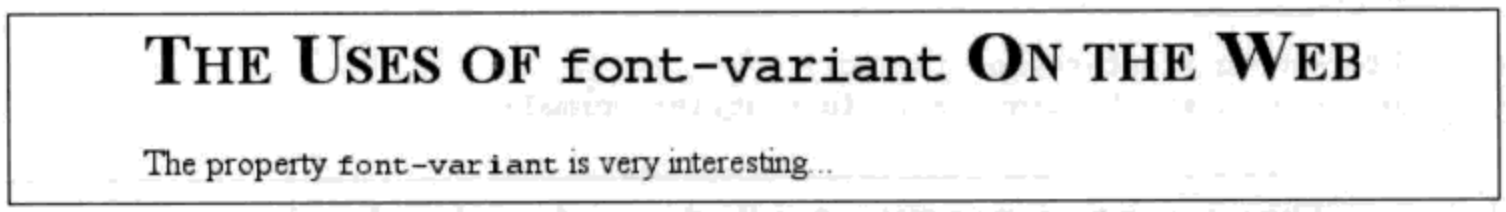

图5-20：实际使用small-caps值

可以注意到，在 h1 元素的显示中，只要文本源中出现大写字母，会显示一个更大的大写字母，而且只要文本源中出现一个小写字母，就会显示一个小型的大写字母。这与 text-transform: uppercase 很类似，唯一的区别在于，在此大写字母的大小不同。不过，之所以使用一个字体属性来声明 small-caps，原因是有些字体有特定的 small-caps 字体，这要通过一个字体属性来选择。

如果不存在这样的字体会怎么样呢？规范中提供了两种选择。第一种是让用户代理自己缩放大写字母来创建一个 small-caps 字体。第二种方法是让所有字母都大写，而且大小相同，就好像使用了声明 text-transform: uppercase; 一样。这显然不是最理想的解决办法，不过确实允许这样做。

注意： Windows Internet Explorer 在 IE6 之前采用的是后一种做法，即全变成大写。而大多数其他浏览器会在有要求时显示 small-caps 文本。

拉伸和调整字体

CSS2 中有两个字体属性在 CSS2.1 中未出现。它们已经从 CSS2.1 中去除，这是因为，尽管它们在规范中已经存在多年，但是还没有任何一个浏览器实现这两个属性。前一个属性允许将字体水平拉伸，第二个属性允许在创作人员的首选字体不可用时，对替换字体进行智能缩放。首先来看拉伸。

font-stretch

值：	normal | wider | narrower | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded | inherit
初始值：	normal
应用于：	所有元素
继承性：	有

顾名思义，这个属性用于让一种字体的字符更胖或更瘦。就像font-size属性的绝对大小关键字（如xx-large）一样，这个属性有一系列绝对值，还有两个值允许创作人员增加或减少字体的拉伸度。例如，一个创作人员可能决定强调一个已经很强调的元素，将其字体字符拉伸得比其父元素的字体字符更宽，如图5-21所示：

```
strong {font-stretch: wider;}
```

If there's one thing I can't **stress enough**, it's the value of Photoshop in producing a book like this one.

图5-21：拉伸字体字符

注意： 图5-21用Photoshop做了修改，因为在写作本书时Web浏览器不支持font-stretch。

调整字体大小的过程也未实现，这个过程稍微有些复杂。

font-size-adjust

值：	<number> | none | inherit
初始值：	none
应用于：	所有元素
继承性：	有

这个属性的目标是，当所用字体并非创作人员的首选时，仍然保证可以辨识。由于字体外观上的差异，一种字体在某个大小时可能可以辨识，而另一种同样大小的字体则可能很难辨识，甚至无法阅读。

影响字体是否能辨识的因素包括其大小和其x-height。x-height除以font-size的结果称为方面值（aspect value）。如果字体的方面值较高，随着字体大小的减少这种字体往往还能辨识；反过来，如果字体的方面值比较低，就会更快地变得不可辨识。

下面来比较常用的字体Verdana和Times，这是一个很好的例子。考虑图5-22和以下标记，在此font-size设置为10px，此时这两种字体显示如下：

```
p {font-size: 10px;}
p.cl1 {font-family: Verdana, sans-serif;}
p.cl2 {font-family: Times, serif; }
```

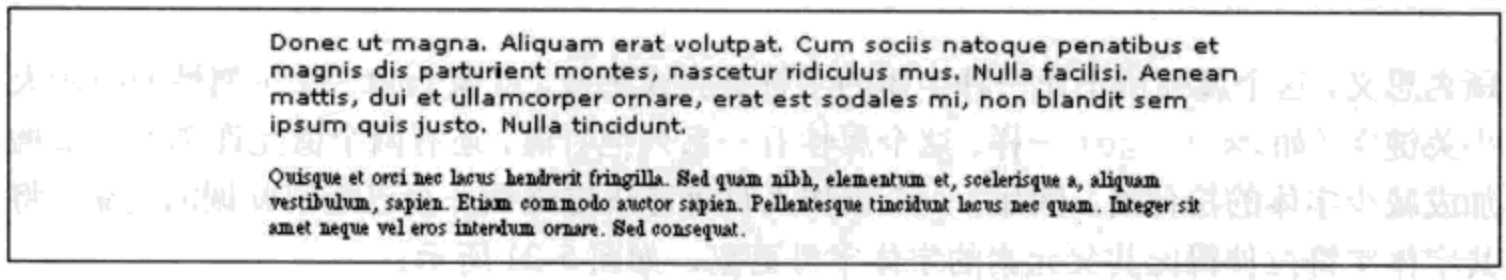

图5-22：比较Verdana和Times

Times字体的文本比Verdana文本更难读。部分原因在于基于像素的显示所存在的限制，另一个原因是：对于较小的字体大小，Times会变得更难读。

可以看到，对于Verdana字体，x-height与字符大小的比值是0.58，而对于Times则是0.46。在这种情况下，所能做的就是声明Verdana的方面值，用户代理会调整具体使用的文本大小。这通过使用以下公式实现：

> 声明的font-size × （font-size-adjust值÷可用字体的方面值）=
> 调整后的font-size

因此，在使用Times而不是Verdana的情况下，调整如下：

```
10px × (0.58 ÷ 0.46) = 12.6px
```

这就会得到如图5-23所示的结果。

```
p {font: 10px Verdana, sans-serif; font-size-adjust: 0.58;}
p.cl1 {font-family: Times, serif; }
```

注意：图5-23用Photoshop做了修改，因为写作本书时很少有浏览器支持font-size-adjust。

当然，要让用户代理聪明地完成大小调整，还必须知道你的首选字体的方面值。在CSS2中没有一种办法能简单地从字体得到这个值，很多字体可能根本没有这个信息。

Donec ut magna. Aliquam erat volutpat. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla facilisi. Aenean mattis, dui et ullamcorper ornare, erat est sodales mi, non blandit sem ipsum quis justo. Nulla tincidunt.

Quisque et orci nec lacus hendrerit fringilla. Sed quam nibh, elementum et, scelerisque a, aliquam vestibulum, sapien. Etiam commodo auctor sapien. Pellentesque tincidunt lacus nec quam. Integer sit amet neque vel eros interdum ornare. Sed consequat.

图5-23：调整Times

font 属性

当然，所有这些属性都很复杂，不过要使用所有这些属性可能更麻烦：

```
h1 {font-family: Verdana, Helvetica, Arial, sans-serif; font-size: 30px;
   font-weight: 900; font-style: italic; font-variant: small-caps;}
h2 {font-family: Verdana, Helvetica, Arial, sans-serif; font-size: 24px;
   font-weight: bold; font-style: italic; font-variant: normal;}
```

通过对选择器分组可以部分地解决这个问题，不过将所有内容都合并到一个属性中不是更简单吗？这就是 font 属性，它是涵盖所有其他字体属性（以及少数其他内容）的一个简写属性。

font

值：	[[<font-style> || <font-variant> || <font-weight>]?<font-size> [/<line-height>]?<font-family>] | caption | icon | menu | message-box | small-caption | status-bar | inherit
初始值：	根据单个属性
应用于：	所有元素
继承性：	有
百分数：	对于 <font-size> 要相对于父元素来计算；对于 <line-height> 则相对于元素的 <font-size> 来计算
计算值：	见单个属性（font-style 等）

一般来讲，font 声明可以有上述各个字体属性的任何值，或者有一个“系统字体”值（见“使用系统字体”一节的介绍）。因此，前面的例子可以简写如下：

```
h1 {font: italic 900 small-caps 30px Verdana, Helvetica, Arial, sans-serif;}
h2 {font: bold normal italic 24px Verdana, Helvetica, Arial, sans-serif;}
```

其效果完全一样（如图 5-24 所示）。

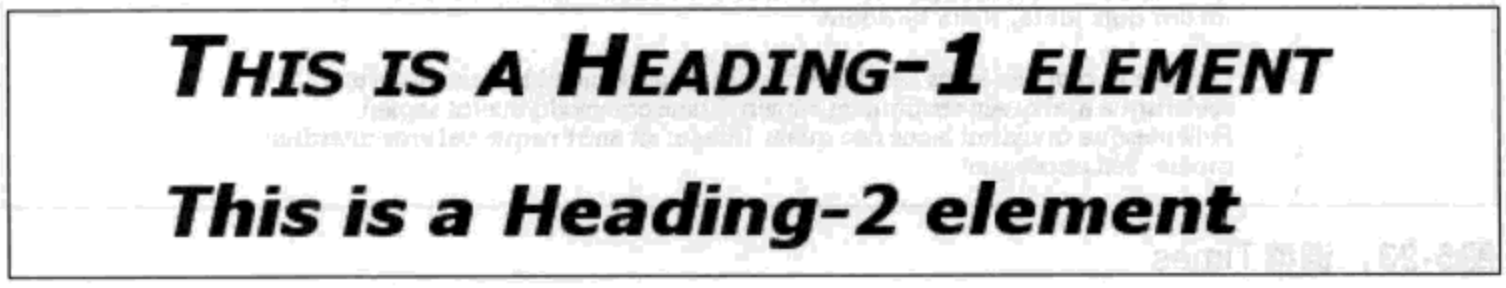

图5-24：典型字体规则

我说过，样式“可以”用这种方式简写，因为还可以有其他写法，font 属性可以用很宽松的方式来指定。如果仔细看前面的例子，会看到前三个值出现的顺序并不一样。在 h1 规则中，前三个值依次分别是 font-style、font-weight 和 font-variant 的值，而在第二个规则中，其顺序则为 font-weight、font-variant 和 font-style。这里没有出错，因为这三个属性值可以按任何顺序来写。此外，如果其中某个属性的值为 normal，则可以忽略。因此，以下规则等价于前面的例子：

```
h1 {font: italic 900 small-caps 30px Verdana, Helvetica, Arial, sans-serif;}
h2 {font: bold italic 24px Verdana, Helvetica, Arial, sans-serif;}
```

在这个例子中，h2 规则中忽略了 normal 值，但效果还是与前例完全相同。

不过，要认识到重要的一点，只是 font 的前三个值允许采用任意的顺序。后两个值则要严格得多。font-size 和 font-family 不仅要以此顺序（font-size 在前，font-family 在后）作为声明中的最后两个值，而且 font 声明中必须有这两个值。这很明确。如果少了其中某个属性，那么整个规则都是无效的，很可能被用户代理完全忽略。因此，以下规则会得到如图 5-25 所示的结果：

```
h1 {font: normal normal italic 30px sans-serif;}   /*no problem here */
h2 {font: 1.5em sans-serif;}   /* also fine; omitted values set to 'normal' */
h3 {font: sans-serif;}   /* INVALID--no 'font-size' provided */
h4 {font: lighter 14px;}   /* INVALID--no 'font-family' provided */
```

增加行高

到目前为止，我们将 font 处理为就好像它只有 5 个值，但事实并非如此。还可以使用 font 设置 line-height，尽管 line-height 是一个文本属性而不是字体属性。这可以作为对 font-size 值的一个补充，并用一个斜线（/）与之分隔：

```
body {font-size: 12px;}
h2 {font: bold italic 200%/1.2 Verdana, Helvetica, Arial, sans-serif;}
```

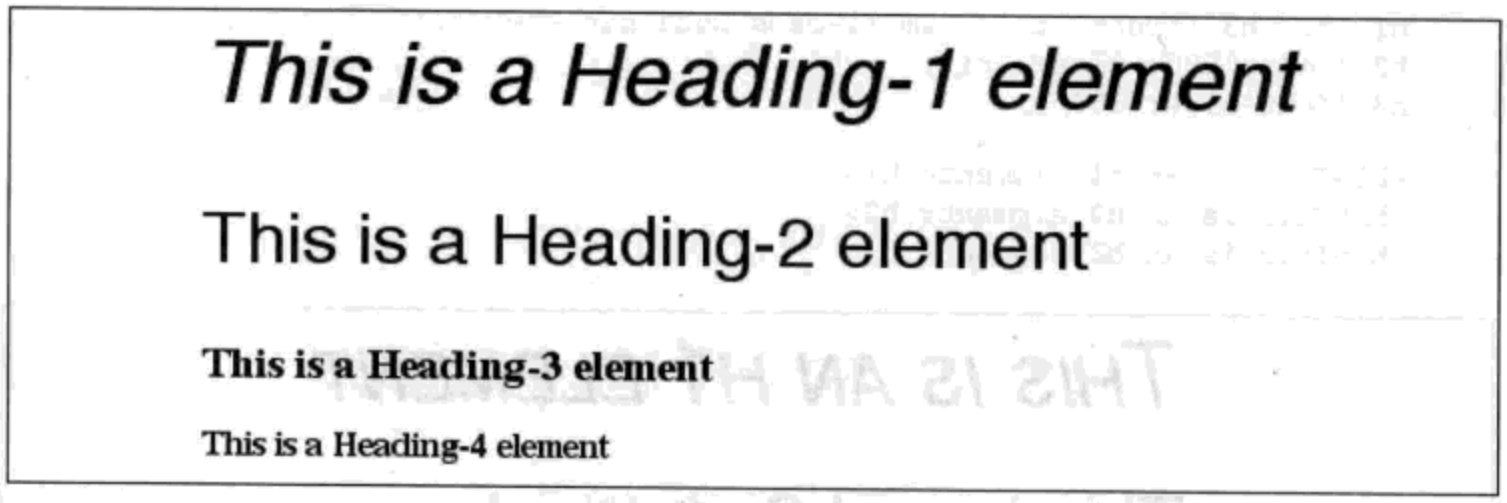

图5-25：字体大小和字体系列的必要性

这些规则如图5-26所示，将所有h2元素设置为粗斜体（使用sans-serif字体系列中的某个字体），将font-size设置为24px（body大小的两倍），并设置line-height为30px。

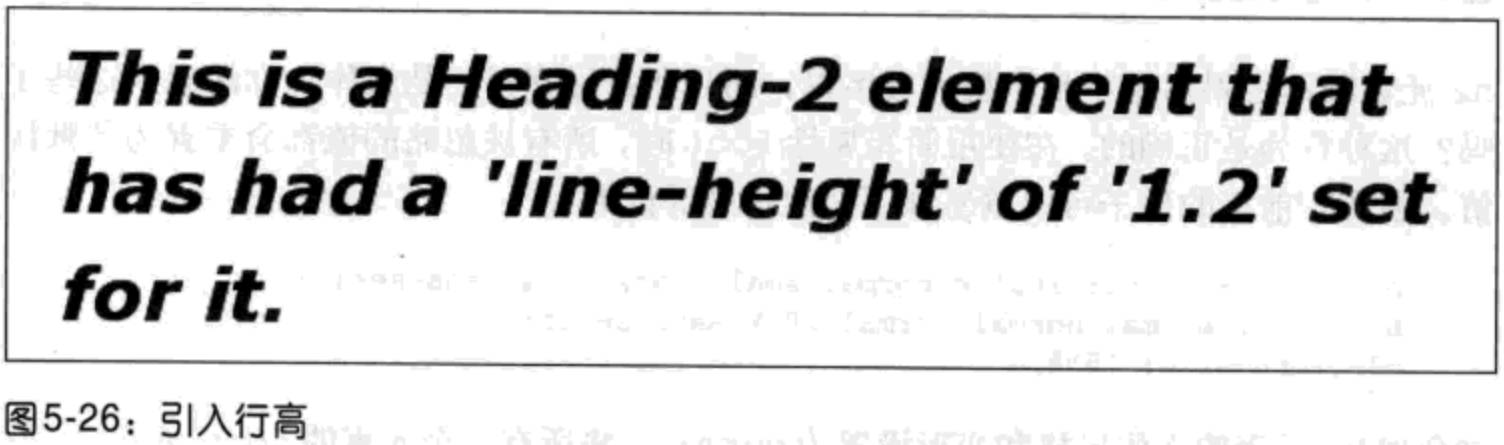

图5-26：引入行高

增加这个line-height值完全是可选的，就像前三个font值一样。如果确实包含了一个line-height，要记住font-size总是在line-height之前，绝对不能在line-height后面，而且这两个属性总要用一个斜线分隔。

尽管听上去有些啰嗦，不过要知道这是CSS创作人员最常犯的错误之一，所以再强调也不为过：font中font-size和font-family值是必要的，而且顺序不能变。不过所有其他值都是可选的。

注意：line-height将在下一章讨论。

适当地使用简写

要记住重要的一点，font作为一个简写属性，如果使用不小心，可能会有预想不到的作用。考虑以下规则，如图5-27所示。

```
h1, h2, h3 {font: italic small-caps 250% sans-serif;}
h2 {font: 200% sans-serif;}
h3 {font-size: 150%;}

<h1>This is an h1 element</h1>
<h2>This is an h2 element</h2>
<h3>This is an h3 element</h3>
```

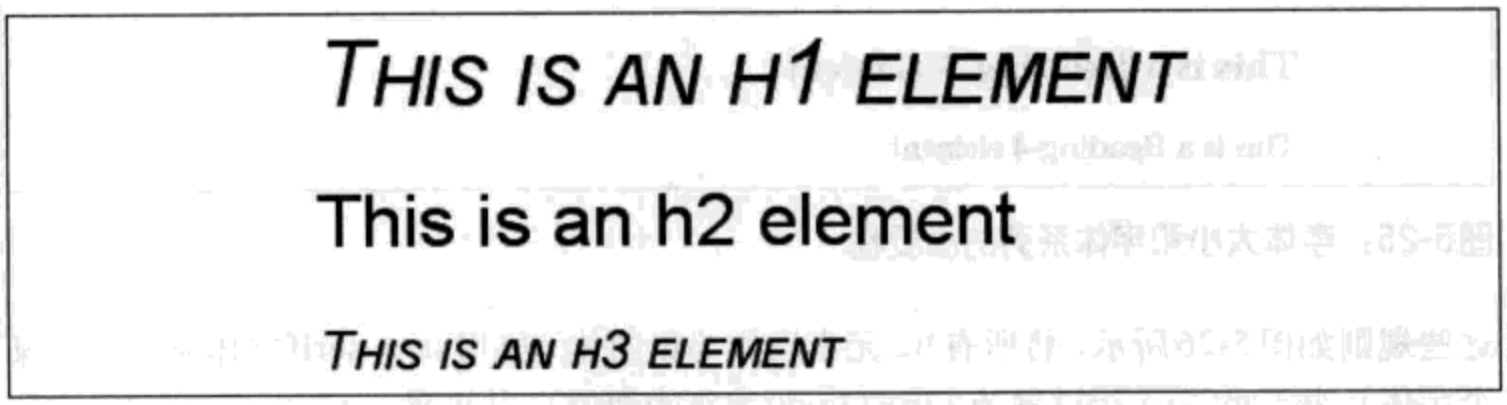

图5-27：简写的变化

h2元素既不是斜体也不是小型大写字母，而且所有元素都不是粗体，你注意到这些了吗？这种行为是正确的。在使用简写属性font时，所有被忽略的值都会重置为其默认值。因此，前面的例子可以重写如下，这仍是等价的：

```
h1, h2, h3 {font: italic normal small-caps 250% sans-serif;}
h2 {font: normal normal normal 200% sans-serif;}
h3 {font-size: 150%;}
```

这会把h2元素的字体风格和变形设置为normal，将所有三个元素的font-weight设置为normal。这是简写属性的期望行为。h3与h2的命运不同，因为这里使用了属性font-size，这不是一个简写属性，因此只会影响该属性自己的值。

使用系统字体

如果希望一个Web页面结合用户操作系统的设置，在这种情况下，font的系统字体值会很方便。这些值用于取得操作系统中元素的字体大小、字体系列、字体加粗、字体风格和字体变形，并将其应用到一个元素。这些值如下：

caption
: 用于有标题的控件，如按钮。

icon
: 用于对图标加标签。

menu
: 用于菜单，即下拉菜单和菜单列表。

message-box
: 用于对话框。

small-caption
: 用于对小控件加标签。

status-bar
: 用于窗口状态条。

例如，你可能想把一个按钮的字体设置为与操作系统中的按钮字体完全相同。例如：

```
button {font: caption;}
```

利用这些值，可以创建一个基于Web的应用，使之看上去非常类似于用户操作系统自带的应用。

注意，系统字体可能只能整体设置；也就是说，字体系列、大小、加粗、风格等等都要一起设置。因此，上例中的按钮文本看上去与操作系统中的按钮文本完全相同，而不论其大小与按钮周围的内容是否匹配。不过，一旦已经设置了系统字体，就可以修改其中的单个值。因此，以下规则会确保按钮的字体与其父元素的字体大小相等：

```
button {font: caption; font-size: 1em;}
```

如果需要一种系统字体，但是用户机器上不存在这样一种字体，用户代理可能会试图找出一种近似的字体，如缩小caption字体的大小，从而得到small-caption字体。如果无法找到这样的近似字体，用户代理就要使用它自己的一种默认字体。如果可以找到一个系统字体，但是无法读取其所有值，就应当使用默认值。例如，用户代理可能可以找到一个status-bar字体，但是无法得到这种字体是否是small-caps的有关信息。在这种情况下，用户代理会对small-caps属性使用值normal。

注意：用户界面风格将在第13章更详细地讨论。

字体匹配

可以看到，CSS允许匹配字体系列、加粗和变形。所有这些都是通过字体匹配完成的，这是一个相当复杂的过程。如果创作人员想帮助用户代理在显示其文档时做出正确的字体选择，了解这个过程就很重要。我把这个内容留到这一章的最后才介绍，因为这对于理解字体属性如何工作并不是必要的，有些读者可能想跳过这个部分直接看下一章。如果你还对这个内容感兴趣，下面将介绍字体匹配是如何工作的。

1. 用户代理创建（或访问）一个字体属性数据库。这个数据库列出了用户代理能访问的所有字体的各种CSS属性。一般地，这将是机器上安装的所有字体，虽然可能还包含另外的一些字体（例如，用户代理可以有自己的内置字体）。如果用户代理遇到两种相等的字体，会把其中一个忽略。
2. 用户代理取得应用了字体属性的元素，并构建一个字体属性列表，其中列出显示该元素的必要字体属性。基于这个列表，用户代理先对显示元素时使用的字体系列做第一个选择。如果完全匹配，那么用户代理就可以使用这个字体。否则，需要多做一些工作。
   a. 字体首先根据 font-style 进行匹配。关键字 italic 可以与所有标有“italic”或“oblique”的字体匹配。如果没有这样的字体，则匹配失败。
   b. 接下来再根据 font-variant 进行匹配。未标“small-caps”的字体都认为是 normal。与 small-caps 匹配的字体可以是标为“small-caps”的字体，也可以是允许合成small-caps风格的字体，或者是用大写字母替换小写字母的字体。
   c. 然后根据 font-weight 匹配，由于CSS中处理 font-weight 的特殊方式，这个匹配绝不会失败（这在本章前面已经解释过）。
   d. 之后再针对 font-size 进行匹配。必须在某个可忍受的范围内匹配，不过这种忍受程度要由用户代理定义。因此，一个用户代理可能认为错误不超过20%都能匹配，而另一个用户代理则只允许指定大小与实际使用的大小之间有10%的差异。
3. 如果在第2步中未匹配任何字体，用户代理就会在这个字体系列中查找下一个候选的字体。如果找到了，则对该字体重复第 2 步。
4. 假设找到一个通用匹配，但是其中不包含显示给定元素所需的一切 —— 例如，这种字体没有版权符号 —— 用户代理就要回到第 3 步，再搜索另一个候选字体，然后再通过第 2 步来验证这种字体是否匹配。
5. 最后，如果没有找到匹配，而且所有候选字体都已经试过了，用户代理就会为给定的通用字体系列选择默认字体，尽其所能正确地显示这个元素。

整个过程很长，也很麻烦，不过这有助于理解用户代理如何选择字体。例如，你可能想指定在一个文档中使用 Times 或任何其他 serif 字体：

```
body {font-family: Times, serif;}
```

对每个元素，用户代理要检查该元素中的字符，并确定Times是否能提供匹配的字符。在大多数情况下，确实能做到这一点而没有任何问题。不过，假设段落中有一个汉字字符，Times没有与这个汉字匹配的字符，所以用户代理必须忽略这个字符，或者查找另一个能满足该元素显示需求的字体。当然，任何西方字体都不太可能包含中文字符，不过假设

存在这样一种字体（暂且称之为AsiaTimes），用户代理显示该元素时可以使用这个字体——或者只是在显示这个字符时使用该字体。因此，可能整个段落用AsiaTimes显示，或者段落中的所有内容都用 Times 显示，只有那个中文字符除外，它用 AsiaTimes 显示。

font-face 规则

CSS2 引入了一种方法，可以通过 `@font-face` 规则对字体匹配有更多控制。在 2003 年春天前，所有 Web 浏览器都没有充分实现这个规则，所以 `@font-face` 已经从 CSS2.1 去除。我不打算在这上面多花功夫，因为这个规则的各个方面相当复杂，单是说明这个规则就可能需要整个一章（甚至一本书）的篇幅才能做到。

有 4 种方法可以确定文档中使用的字体。我们将简要讨论这 4 种方法，因为CSS 的将来版本可能使用这种机制，而且大多数SVG显示器至少部分支持CSS2中描述的font-face匹配。如果你需要实现 `@font-face`，请参考 CSS2 规范，或者任何最新版本的 CSS（如 CSS3 Web Fonts 模块），下面的描述并不完备。

字体名匹配

要匹配字体名，用户代理会使用与所请求字体有相同系列名的一种可用字体。这种字体的外观和度量与所请求的字体可能并不相同。这是本节前面介绍的方法。

智能字体匹配

在这种情况下，用户代理使用外观上与所请求字体最接近的一种可用字体。这两种字体可能并不能完全匹配，但是它们应当尽可能地接近。

用于匹配两种字体的信息包括字体种类（文本或符号）、是否有上下短线、加粗、大写字母高度、x-height、上升、下降、倾斜等等。例如，一个创作人员可能要求一个字体与某倾斜字体尽可能地接近，为此写出以下规则：

```
@font-face {font-style: normal; font-family: "Times"; slope: -5;}
```

然后由用户代理找到一个 serif normal （竖直）字体，如果 Times 无法满足要求，将其向右倾斜5度尽可能接近倾斜字体。CSS2中还描述了字体的很多其他方面，如果用户代理支持这些方面，所有这些都可以用于完成匹配过程。

字体合成

用户代理也可以选择实时地生成一个字体，使其外观和度量与@font-face规则中指定的描述相匹配。CSS2 这样描述这个过程：

> 在这种情况下，用户代理创建一个字体，它不仅在外观上几乎匹配，而且与所请求字体的度量匹配。合成信息包括匹配信息，通常需要比某些匹配机制使用更准确的参数值。具体地，如果要保留指定字体的所有布局特征，合成需要准确的宽度度量和字符来完成字形和位置信息替换。

如果你对这些已经了解，可能就不需要我再多做解释了。如果你还不了解，也许根本不需要担心这个问题。

字体下载

采用这种方法，用户代理可以在文档中下载一个远程字体来使用。要声明一个下载字体，可以写如下规则：

```
@font-face {font-family: "Scarborough Fair";
   src: url(http://www.example.com/fonts/ps/scarborough.ps);}
```

然后就可以在整个文档中使用该字体了。

即使用户代理允许字体下载，可能也要花一些时间才能获取到字体文件（这种文件往往很大），这就会延迟文档的显示，或者至少延迟最终输出。

小结

尽管创作人员不能指望文档中一定使用某个特定的字体，不过可以很容易地指定通用字体系列。这种特殊行为得到了很好的支持，因为如果用户代理不允许创作人员（甚至读者）指定字体，会很快发现这种用户代理不受欢迎。

对于字体管理的其他方面，相应的支持程度则有所不同。往往能很好地改变字体大小，不过在这方面20世纪的用户代理实现差别很大，有些极其简化，有些则非常正确。对创作人员来说，让人困惑的往往不是以何种方式支持字体大小调整，而是他们想用的单位（点）在不同媒体上得到的结果可能差异很大，甚至在不同操作系统和用户代理上也可能得到不同的结果。使用点作为单位有很多危险，在Web设计中使用长度单位通常不是一个好主意。百分数、em单位和ex单位往往更适合修改字体大小，因为这些单位在所有常用显示环境中能很好地缩放。

另一个让人困惑的方面可能是一直缺少一种机制来指定要下载并在文档中使用的字体。这意味着创作人员仍依赖于用户可用的字体，因此，他们无法预计文本会有怎样的外观。

说到为文本指定样式，还有很多不涉及字体的方法，这是下一章要讨论的内容。

第6章

文本属性

不错，很多Web设计都需要选择适当的颜色，让页面有最酷的外观，不过在真正的Web设计中，可能更多的时间都花在这样一些问题上：文本放在哪里，文本的外观是怎样的。由于存在这些问题，出现了一些HTML标记（如<FONT>和<CENTER>），允许你对文本的外观和放置有所控制。

因为文本如此重要，所以有很多CSS属性以这样或那样的方式影响文本。文本和字体之间有什么不同呢？简单地讲，文本是内容，而字体用于显示这个内容。使用文本属性，可以控制文本相对于该行余下内容的位置、使其作为上标、加下划线，以及改变大小写等。甚至还可以有限地模拟打字机的Tab键的使用。

缩进和水平对齐

下面先来讨论如何影响文本在行中的水平定位。写一个时事通讯或做一份报告时，你可能会采取一些步骤，可以把这些基本动作看作是这些步骤的一部分。

缩进文本

将Web页面上一个段落的第一行缩进，这是一种最常用的文本格式化效果（去除段落之间的空行是第二常用的方法，这个内容将在第7章讨论）。有些网站在段落的第一个字母前放一个很小的透明图像，这些图像将文本推到后面来制造一种缩进文本的感觉。另外一些网站则使用完全非标准的SPACER标记。在CSS中，有一种更好的方法实现文本缩进，即 `text-indent` 属性。

text-indent	
值：	<length> | <percentage> | inherit
初始值：	0
应用于：	块级元素
继承性：	有
百分数：	相对于包含块的宽度
计算值：	对于百分数值，要根据指定确定；对于长度值，则为绝对长度

通过使用text-indent属性，所有元素的第一行都可以缩进一个给定长度，甚至该长度可以是负值。当然，这个属性最常见的用途是将段落的首行缩进：

```
p {text-indent: 3em;}
```

这个规则会使所有段落的首行缩进3 em，如图6-1所示。

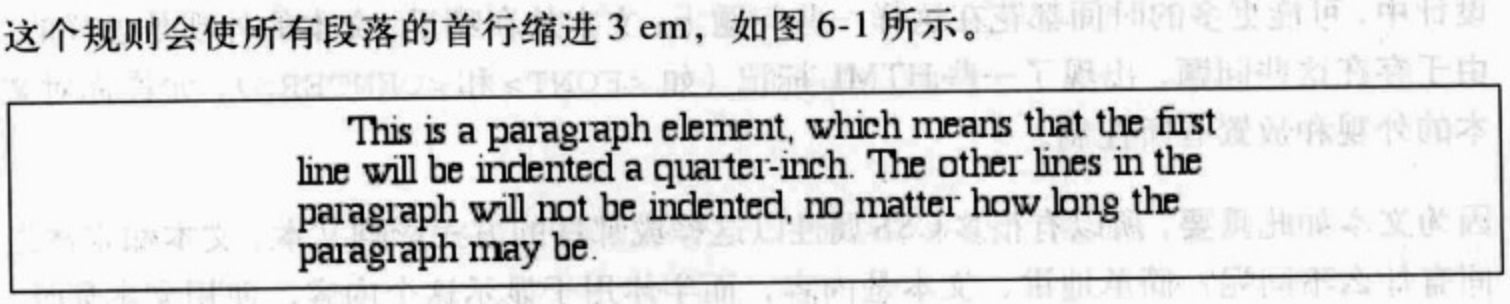

图6-1：文本缩进

一般地，可以为所有块级元素应用text-indent，但无法将这个属性应用到行内元素，图像之类的替换元素上也无法应用text-indent属性。不过，如果一个块级元素（如段落）的首行中有一个图像，它会随着该行的其余文本移动。

注意： 如果想把一个行内元素的第一行“缩进”，可以用左内边距或外边距创造这种效果。

---

text-indent还可以设置为负值，利用这种技术，可以实现很多有意思的效果。最常见的用途是一种“悬挂缩进”，即第一行悬挂在元素中余下部分的左边：

```
p {text-indent: -4em;}
```

在为text-indent设置负值时要当心：上例中前3个词（“This is a”）可能会超出浏览器窗口的左边界。为了避免出现这种显示问题，建议针对负缩进再设置一个外边距或一些内边距：

```
p {text-indent: -4em; padding-left: 4em;}
```

不过，负缩进也可以得到充分利用。考虑下面的例子，如图 6-2 所示，这里增加了一个浮动图像：

```
p.hang {text-indent: -25px;}

<img src="star.gif" style="width: 60px; height: 60px;
float: left;" alt="An image of a five-pointed star."/>
<p class="hang"> This paragraph has a negatively indented first
line, which overlaps the floated image that precedes the text.  Subsequent
lines do not overlap the image, since they are not indented in any way.</p>
```

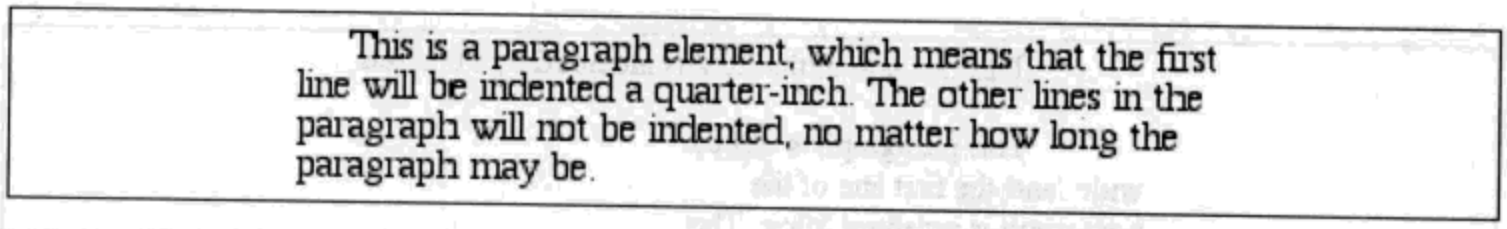

图6-2：浮动图像和负文本缩进

使用这个简单的技术可以实现很多有意思的设计。

text-indent 可以使用所有长度单位（包括百分数值）。在下面的例子中，百分数要相对于缩进元素父元素的宽度。换句话说，如果将缩进值设置为10%，所影响元素的第一行会缩进其父元素宽度的10%，如图 6-3 所示：

```
div {width: 400px;}
p {text-indent: 10%;}

<div>
<p>This paragraph is contained inside a DIV, which is 400px wide, so the
first line of the paragraph is indented 40px (400 * 10% = 40).  This is
because percentages are computed with respect to the width of the element.</p>
</div>
```

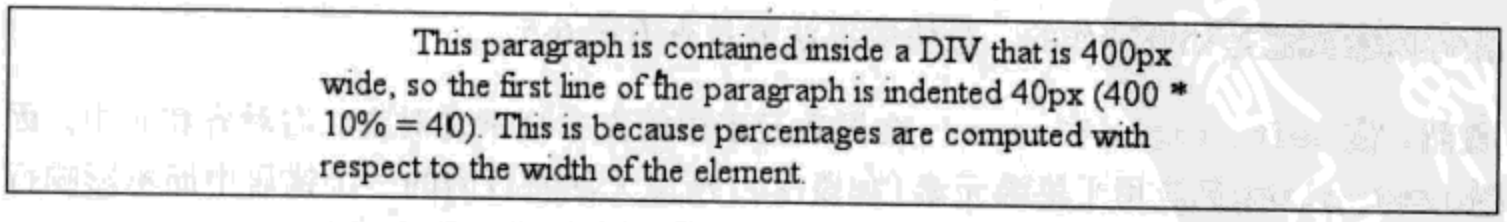

图6-3：用百分数实现文本缩进

注意，即使插入了行分隔符，这种缩进也只应用于一个元素的第一行。关于 text-indent 有意思的是，由于这个属性可以继承，它可能有预想不到的效果。例如，考虑以下标记（如图 6-4 所示）：

```
div#outer {width: 500px;}
div#inner {text-indent: 10%;}
p {width: 200px;}

<div id="outer">
<div id="inner"> This first line of the DIV is indented by 50 pixels.
<p>
This paragraph is 200px wide, and the first line of the paragraph
is indented 50px.  This is because computed values for 'text-indent'
are inherited, instead of the declared values.
</p>
</div>
</div>
```

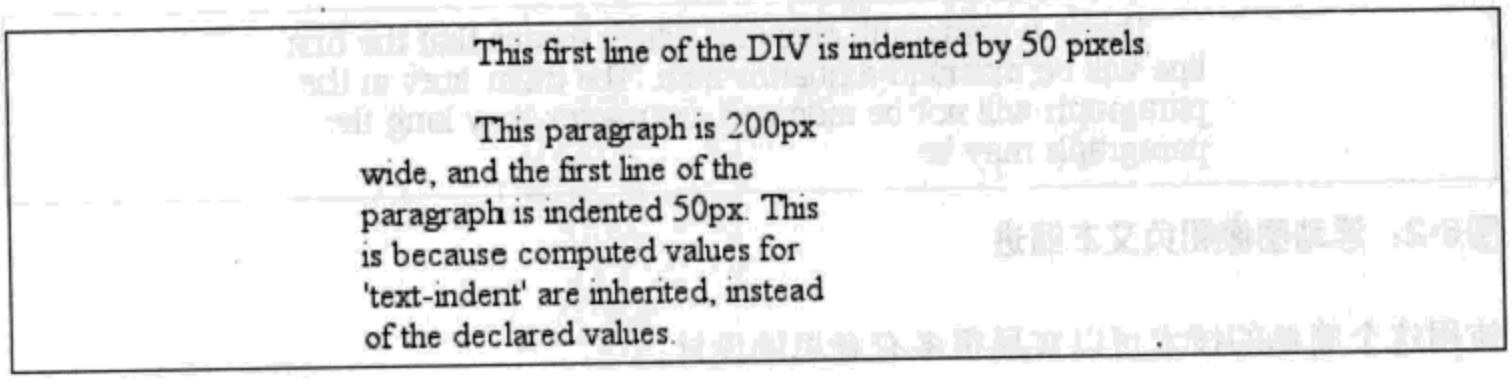

图6-4：文本缩进继承

注意：在CSS 2.1之前，text-indent总是继承计算值，而不是声明值。

水平对齐

与text-indent相比，text-align是一个更基本的属性，它会影响一个元素中的文本行相互之间的对齐方式。前3个值相当直接，不过第4个和第5个则略有些复杂。

要理解这些值是如何工作的，最快的办法就是查看图6-5。

显然，值left、right和center会导致元素中的文本分别左对齐、右对齐和居中。因为text-align只应用于块级元素（如段落），所以无法将行内的一个锚居中而不影响行中的其余部分（你可能也不想这么做，因为这很可能导致文本重叠）。

西方语言都是从左向右读，所以text-align 的默认值是left。文本在左边界对齐，右边界呈锯齿状（称为“从左到右”文本）。对于希伯来语和阿拉伯语之类的语言，text-align则默认为right，因为这些语言从右向左读。不出所料，center会使每个文本行在元素中居中。

text-align

CSS2.1 值： left | center | right | justify | inherit

CSS2 值： left | center | right | justify | <string> | inherit

初始值： 用户代理特定的值；还可能取决于书写方向

应用于： 块级元素

继承性： 有

计算值： 根据指定确定

说明： CSS2包含一个<string>值，因为没有相应实现，所以在CSS2.1中已经去除

This paragraph assumes the style `text-align: left;`, which causes the line boxes within the element to line up along the left inner content edge of the paragraph.

This paragraph assumes the style `text-align: right;`, which causes the line boxes within the element to line up along the right inner content edge of the paragraph.

This paragraph assumes the style `text-align: center;`, which causes the line boxes within the element to line up their centers with the center of the content area of the paragraph.

This paragraph assumes the style `text-align: justify;`, which causes the line boxes within the element to align their left and right edges to the left and right inner content edges of the paragraph. The exception is the last line box, whose right edge does not align with the right content edge of the paragraph. (In right-to-left languages, the left edge of the last line box would not be so aligned.)

图6-5：text-align属性的行为

注意： 将块级元素或表元素居中，这要通过在这些元素上适当地设置左、右外边距来实现。详细内容见第7章。

虽然你可能认为 text-align: center 与 <CENTER> 元素的作用一样，但实际上二者大不相同。<CENTER> 不仅影响文本，还会把整个元素居中，如表。text-align 不会控制元素的对齐，而只影响其内部内容。图6-5 很清楚地展示了这一点。实际元素没有从一端移到另一端。只是其中的文本受影响。

警告： IE6之前的IE/Win有一个危害较大的bug：它确实会把`text-align: center`处理为`<CENTER>`元素，不仅将文本居中，还会将元素居中。在IE6和更高版本IE的标准模式中就不会这样了，但在IE5.x和较早版本中仍是如此。

最后一个水平对齐属性是`justify`，它会带来自己的一些问题。在两端对齐文本中，文本行的左右两端都放在父元素的内边界上，如图6-6所示。然后，调整单词和字母间的间隔，使各行的长度恰好相等。两端对齐文本在打印领域很常见（例如，本书就使用了两端对齐文本），不过在CSS中，还需要多做一些考虑。

This is a paragraph of justified text. Notice that the spacing between words, or even between individual letters, depends greatly on the number of words in each line. Intraword and intracharacter spacing is adjusted to create the justification effect, so it can effectively override values for properties such as **`word-spacing`** and **`letter-spacing`**.

图6-6：两端对齐文本

要由用户代理（而不是CSS）来确定两端对齐文本如何拉伸，以填满父元素左右边界之间的空间。例如，有些浏览器可能只在单词之间增加额外的空间，而另外一些浏览器可能会平均分布字母间的额外空间（不过CSS规范特别指出，如果`letter-spacing`属性指定为一个长度值，“用户代理不能进一步增加或减少字符间的空间”）。还有一些用户代理可能会减少某些行的空间，使文本挤得更紧密。所有这些做法都会影响元素的外观，甚至改变其高度，这取决于用户代理的对齐选择影响了多少文本行。

CSS也没有指定应当如何处理连字符（注1）。大多数两端对齐文本都使用连字符将长单词分开放在两行上，从而缩小单词之间的间隔，改善文本行的外观。不过，由于CSS没有定义连字符行为，用户代理不太可能自动加连字符。因此，在CSS中，两端对齐文本看上去没有打印出来好看，特别是元素可能太窄，以至于每行只能放下几个单词。当然，使用窄设计元素是可以的，不过要当心相应的缺点。

注1： CSS中没有说明如何处理连字符，因为不同的语言有不同的连字符规则。规范没有尝试去调合这样一些很可能不完备的规则，而是干脆不提这个问题。

垂直对齐

前面已经讨论了水平对齐，下面来看垂直对齐。因为文本行的构造将在第7章更详细地讨论，这里只是提供一个简单的概述。

行高

line-height属性是指文本行基线之间的距离，而不是字体的大小，它确定了将各个元素框的高度增加或减少多少。在最基本的情况下，指定line-height可以用来增加（或减少）文本行之间的垂直间隔，人们认为这是一种了解line-height如何工作的简单方法，但其实并不简单。line-height控制了行间距，这是文本行之间超出字体大小的额外空间。换句话说，line-height值和字体大小之差就是行间距。

line-height

值：	<length> | <percentage> | <number> | normal | inherit
初始值：	normal
应用于：	所有元素（不过请参考关于替换元素和块级元素的介绍）
继承性：	有
百分数：	相对于元素的字体大小
计算值：	对于长度和百分数值是绝对数值；否则，根据指定确定

在应用到块级元素时，line-height定义了元素中文本基线之间的最小距离。注意，它定义的是最小距离，而不是一个绝对数值，文本基线拉开的距离可能比line-height值更大。line-height并不影响替换元素的布局，不过确实可以应用到替换元素（这个小秘密将在第7章揭开）。

构造文本行

文本行中的每个元素都会生成一个内容区，这由字体的大小确定。这个内容区则会生成一个行内框（inline box），如果不存在其他因素，这个行内框就完全等于该元素的内容区。由line-height产生的行间距就是增加或减少各行内框高度的因素之一。

要确定一个给定元素的行间距，只需将line-height的计算值减去font-size的计算值。这个值是总的行间距。而且要记住，这可能是一个负值。然后行间距再除2，将行间距的一半分别应用到内容区的顶部和底部。其结果就是该元素的行内框。

举个例子，假设font-size为14像素高（相应地，内容区的高度也是14像素），而且line-height计算为18像素。其差（4像素）除以2，将其一半分别应用到内容区的顶部和底部。这会得到一个18像素高的行内框，在内容区的上面和下面分别有2个额外的像素。听上去用这种方法描述line-height如何工作好像很绕，不过这样描述有充分的理由。

一旦给定内容行已经生成了所有行内框，接下来在行框的构造中就会考虑这些行内框。行框的高度恰好足以包含最高行内框的顶端和最低行内框的底端。图6-7展示了这个过程。

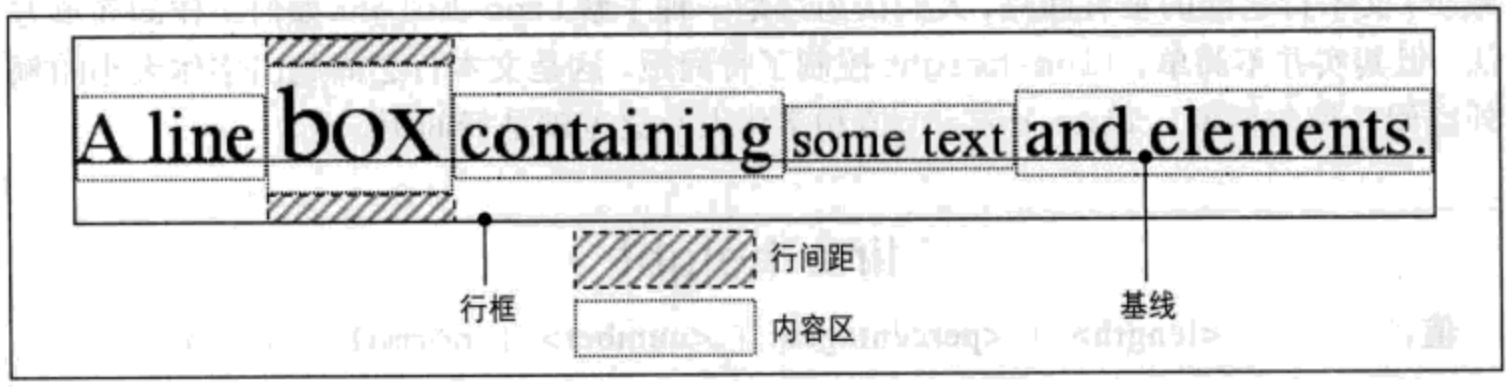

图6-7：行框图

指定line-height值

下面来考虑line-height的可取值。如果使用默认值normal，用户代理必须计算行间的垂直空间。不同的用户代理计算出的值可能不同，不过通常都是字体大小的1.2倍，这使得行框要高于给定元素的font-size值。

大多数值都是简单的长度度量（例如，18px或2em）。注意，即使使用一个合法的长度度量，如4cm，但浏览器（或操作系统）在实际度量中使用的标准可能并不正确，所以在你的显示器上行高可能不是4厘米。更多细节参见第4章。

em、ex和百分数值都相对于元素的font-size值计算。以下标记很直接明了，其结果如图6-8所示：

```
body {line-height: 14px; font-size: 13px;}
p.cl1 {line-height: 1.5em;}
p.cl2 {font-size: 10px; line-height: 150%;}
p.cl3 {line-height: 0.33in;}

<p>This paragraph inherits a 'line-height' of 14px from the body, as well as
a 'font-size' of 13px.</p>
<p class="cl1">This paragraph has a 'line-height' of 19.5px(13 * 1.5), so
it will have slightly more line-height than usual.</p>
<p class="cl2">This paragraph has a 'line-height' of 15px (10 * 150%), so
```

```
it will have slightly more line-height than usual.</p>
<p class="cl3">This paragraph has a 'line-height' of 0.33in, so it will have
slightly more line-height than usual.</p>
```

This paragraph inherits a 'line-height' of 14px from the body, as well as a 'font-size' of 13px.

This paragraph has a 'line-height' of 21px(14 * 1.5), so it will have slightly more line-height than usual.

This paragraph has a 'line-height' of 15px (10 * 150%), so it will have slightly more line-height than usual.

This paragraph has a 'line-height' of 0.33in, so it will have slightly more line-height than usual.

图6-8：line-height属性的简单计算

行高和继承

当一个块级元素从另一个元素继承 line-height 时，问题会变得更为复杂。line-height值从父元素继承时，要从父元素计算，而不是在子元素上计算。以下标记的结果如图 6-9 所示。但创作人员原来可能并不想这样：

```
body {font-size: 10px;}
div {line-height: 1em;}  /* computes to '10px' */
p {font-size: 18px;}
```

```
<div>
<p>This paragraph's 'font-size' is 18px, but the inherited 'line-height'
value is only 10px.  This may cause the lines of text to overlap each
other by a small amount.</p>
</div>
```

This paragraph's 'font-size' is 18px, but the inherited 'line-height' value is only 10px. This may cause the lines of text to overlap each other by a small amount.

图6-9： line-height小，font-size大，这就带来了问题

为什么这些行挨得这么近？因为段落从其父元素 div 继承了 line-height 的计算值 10px。如图6-10所示，对于这种 line-height 太小的问题，一种解决办法是为每个元素设置一个显式的 line-height，但是这种方法不太实用。更好的办法是指定一个数，由它设置缩放因子：

```
body {font-size: 10px;}
div {line-height: 1;}
p {font-size: 18px;}
```

指定一个数时，缩放因子将是继承值而不是计算值。这个数会应用到该元素及其所有子元素，所以各元素都根据其自己的 font-size 计算 line-height（见图 6-10 所示）：

```
div {line-height: 1.5;}
p {font-size: 18px;}

<div>
<p>This paragraph's 'font-size' is 18px, and since the 'line-height'
set for the parent div is 1.5, the 'line-height' for this paragraph
is 27px (18 * 1.5).</p>
</div>
```

This paragraph's 'font-size' is 18px, and since the 'line-height' set for the parent div is 1.5, the 'line-height' for this paragraph is 27px (18 * 1.5).

图6-10：使用line-height缩放因子解决继承问题

尽管看上去line-height在每个文本行的上面和下面平均分配了额外空间，实际上，它是在行内元素的内容区顶部和底部增加（或减少）一定的量来创建一个行内框。假设一个段落的默认 font-size是12pt，考虑以下规则：

```
p {line-height: 16pt;}
```

由于12点文本的“固有”行高是12点，前面的规则将在段落中各行文本外围增加额外的4点空间。这个额外的量平均分为两部分，一半放在各行的上面，另一半放在各行的下面。现在基线间则有16点空间，这是分配额外空间的间接结果。

如果指定值 inherit，元素则会使用其父元素的计算值。这与值自然继承没有什么不同，不过特殊性和层叠解决方案不同。这些内容在第3章曾做过详细讨论。

既然已经基本了解了如何构造文本行，下面来讨论相对于行框垂直对齐元素。

垂直对齐文本

如果你曾用过元素sup和sub(上标和下标元素)，或者曾用过有<img src="foo.gif" align="middle">之类标记的图像，说明你已经做过一些基本的垂直对齐。在CSS中，vertical-align 属性只应用于行内元素和替换元素，如图像和表单输入元素。vertical-align属性不能继承。

vertical-align只接受8个关键字、一个百分数值或一个长度值。这些关键字有些我们很熟悉，有些可能不熟悉，包括：baseline（默认值）、sub、super、bottom、text-bottom、middle、top和text-top。我们将分析各关键字如何作用于行内元素。

vertical-align

值：	baseline | sub | super | top | text-top | middle | bottom | text-bottom | <percentage> | <length> | inherit
初始值：	baseline
应用于：	行内元素和表单元格
继承性：	无
百分数：	相对于元素的 line-height 值
计算值：	对于百分数和长度值，为绝对长度；否则，根据指定确定
说明：	应用到表单元格时，只能识别baseline、top、middle和bottom等值

警告： 要记住：vertical-align不影响块级元素中内容的对齐。不过，可以用它来影响表单元格中元素的垂直对齐。详细内容参见第11章。

基线对齐

vertical-align: baseline要求一个元素的基线与其父元素的基线对齐。大多数情况下，浏览器都会这么做，因为你显然希望一行中所有文本元素的底端都对齐。

如果一个垂直对齐元素没有基线 —— 也就是说，如果这是一个图像或表单输入元素，或者是其他替换元素 —— 那么该元素的底端与其父元素的基线对齐，如图 6-11 所示：

```
img {vertical-align: baseline;}

<p>The image found in this paragraph <img src="dot.gif" alt="A dot" /> has its
bottom edge aligned with the baseline of the text in the paragraph.</p>
```

The image found in this paragraph • has its bottom edge aligned with the baseline of the paragraph.

图6-11：图像的基线对齐

这个对齐规则很重要，因为它使得一些Web浏览器总把替换元素的底边放在基线上，即使该行中没有其他文本。例如，假设一个表单元格中只有一个图像。这个图像可能实际在基线上，不过在某些浏览器中，基线下面的空间会导致图像下出现一段空白。另外一

些浏览器则会把图像“紧包”在表单元格中，所以不会出现空白。根据CSS工作组的意见，这种加空白的行为是正确的，不过大多数创作人员都不喜欢这种做法。

注意： 对于这种加空白的行为以及相应的解决办法，更详细的解释见我的文章《Images, Tables, and Mysterious Gaps》（*http://developer.mozilla.org/en/docs/Images,_Tables,_and_Mysterious_Gaps*）。第7章也会更详细地介绍行内布局的这个方面。

上标和下标

vertical-align: sub声明会使一个元素变成下标，这意味着其基线（或者如果这是一个替换元素，则是其底端）相对于其父元素的基线降低。规范并没有定义元素降低的距离，所以对于不同的用户代理，这个距离可能有所不同。

super刚好与sub相反；它将元素的基线（或替换元素的底端）相对于父元素的基线升高。同样地，文本升高的距离取决于具体的用户代理。

注意，值sub和super不会改变元素的字体大小，所以下标或上标文本不会变小（或变大）。相反，下标或上标元素中的所有文本默认地都应当与父元素中的文本大小相同，如图6-12所示：

```
span.raise {vertical-align: super;}
span.lower {vertical-align: sub;}

<p>This paragraph contains <span class="raise">superscripted</span>
and <span class="lower">subscripted</span> text.</P>
```

This paragraph contains superscripted and subscripted text

图6-12：上标和下标对齐

注意： 如果想让上标或下标文本小于其父元素的文本，可以使用属性font-size，这在第5章曾做过介绍。

底端对齐

vertical-align: bottom将元素行内框的底端与行框的底端对齐。例如，以下标记的结果如图6-13所示：

```
.feeder {vertical-align: bottom;}

<p>This paragraph, as you can see quite clearly, contains
a <img src="tall.gif" alt="tall" class="feeder" /> image and
a <img src="short.gif" alt="short" class="feeder" /> image,
and then some text that is not tall.</p>
```

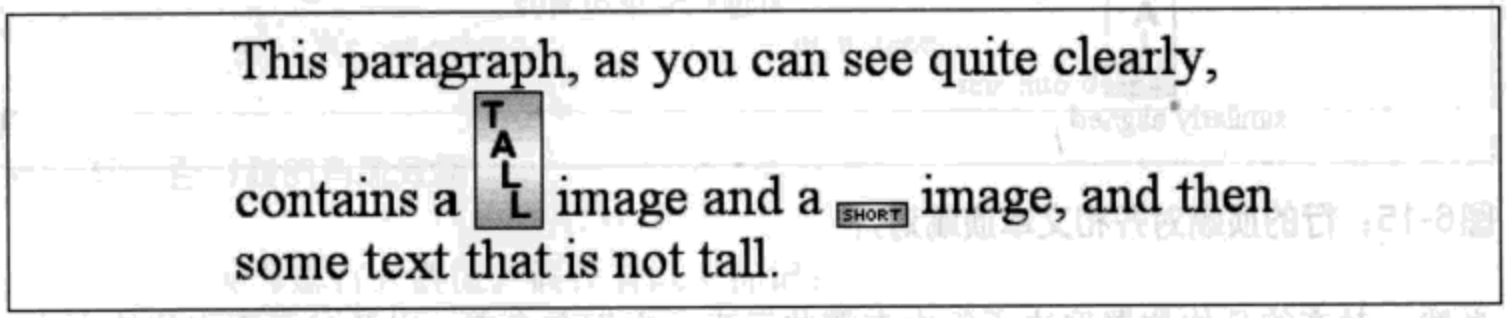

图6-13：底端对齐

图6-13中，段落的第二行包含两个行内元素，其底边彼此对齐。它们都在文本基线之下。

vertical-align: text-bottom是指行内文本的底端。替换元素或任何其他类型的非文本元素会忽略这个值。对于这些元素，将考虑一个“默认”的文本框。这个默认框由父元素的 font-size 得到。要对齐的元素的行内框底端再与这个默认文本框的底端对齐。因此，给定以下标记，可以得到如图 6-14 所示的结果：

```
img.tbot {vertical-align: text-bottom;}

<p>Here: a <img src="tall.gif" style="vertical-align: middle;" alt="tall" />
image, and then a <img src="short.gif" class="tbot" alt="short" /> image.</p>
```

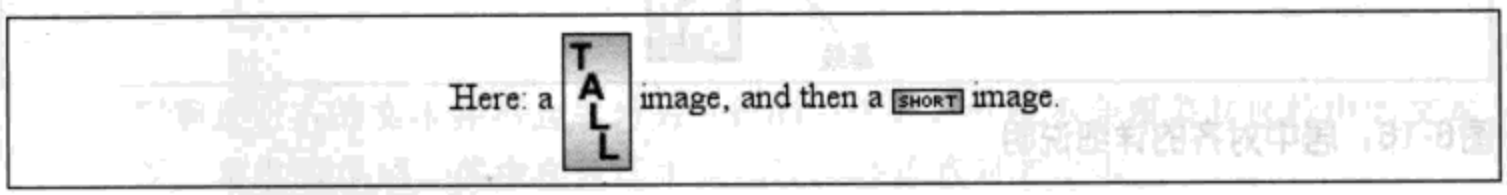

图6-14：文本底端对齐

顶端对齐

vertical-align: top的效果与bottom刚好相反。类似地，vertical-align: text-top 则与 text-bottom 的作用相反。图 6-15 显示了以下标记的结果：

```
.up {vertical-align: top;}
.textup {vertical-align: text-top;}

<p>Here: a <img src="tall.gif" alt="tall image"> tall image, and then
<span class="up">some text</span> that's been vertically aligned.</p>
<p>Here: a <img src="tall.gif" class="textup" alt="tall"> image that's been
vertically aligned, and then a <img src="short.gif" class="textup" alt="short" />
image that's similarly aligned.</p>
```

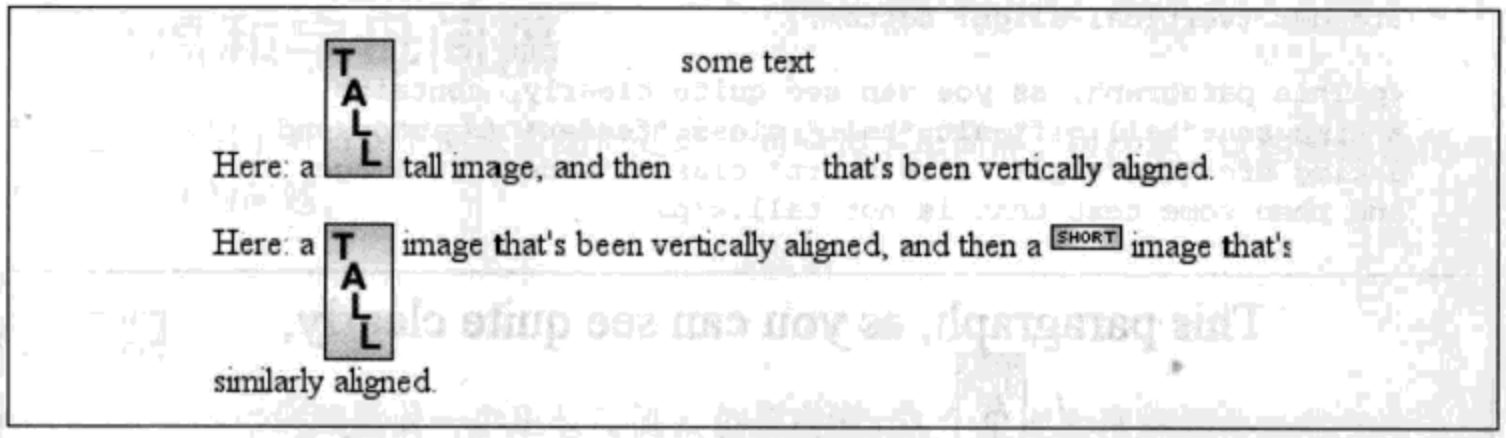

图6-15：行的顶端对齐和文本顶端对齐

当然，对齐的具体位置取决于行内有哪些元素，它们有多高，以及父元素字体的大小。

居中对齐

还有一个值middle，它往往（但并不总是）应用于图像。你可能会从它的名字想象其效果，但你的想象与其实际效果并不完全一样。middle会把行内元素框的中点与父元素基线上方0.5ex处的一个点对齐，这里的1ex相对于父元素的font-size定义。图6-16更详细地说明了这一点。

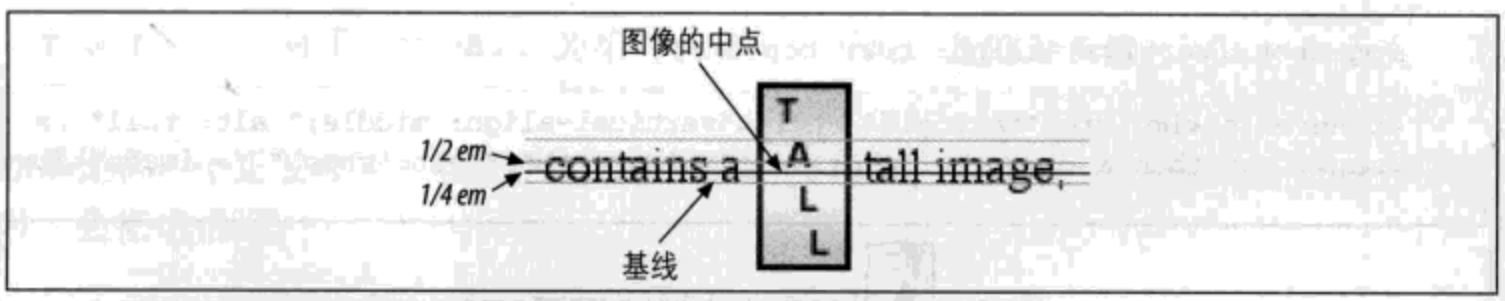

图6-16：居中对齐的详细说明

因为大多数用户代理都把1ex处理为0.5em，middle往往将元素的垂直中点与父元素基线上方0.25em处的一个点对齐。不过，不要指望总会这样，因为有些用户代理确实会为各元素计算准确的x-height（关于x-height的更多详细内容见第5章）。

百分数

使用百分数不能模仿图像的align="middle"对齐。相反，如果为vertical-align设置一个百分数，会把元素的基线（或替换元素的底边）相对于父元素的基线升高或降低指定的量（你指定的百分数要计算为该元素line-height 的百分数，而不是相对于其父元素的line-height）。正百分数值会使元素升高，负值则会使其降低。取决于文本的升高或降低，可能看上去它放在了相邻的行上，如图6-17所示，所以使用百分数值时要当心：

```
sub {vertical-align: -100%;}
sup {vertical-align: 100%;}

<p>We can either <sup>soar to new heights</sup> or, instead,
<sub>sink into despair...</sub></p>
```

soar to new heights
We can either or, instead,
sink into despair...

图6-17：百分数的有趣效果

下面更详细地考虑百分数值。假设有以下标记：

```
<div style="font-size: 14px; line-height: 18px;">
I felt that, if nothing else, I deserved a
<span style="vertical-align: 50%;">raise</span> for my efforts.
</div>
```

设置为50%的span元素对齐时，会使其基线升高9像素，这是元素继承的line-height值（18px）的一半，而不是7像素。

长度对齐

最后，来考虑根据指定长度垂直对齐。vertical-align很明确：它把一个元素升高或降低指定的距离。因此，vertical-align: 5px;会把一个元素与对齐前相比上升5像素。负长度值会使元素下降。这种简单的对齐形式在CSS1中不存在，但在CSS2中已经增加。

要认识到垂直对齐的文本并不会成为另一行的一部分，它也不会覆盖其他行中的文本，这很重要。考虑图6-18，其中在段落中间出现一些垂直对齐文本。

This paragraph contains a lot of text to be dislpayed, and a part of that
boldfaced text
text is some which is raised up 100%. This makes it
look as though the boldfaced text is on a preceding line, but notice that
the line is actually taller than ordinary.

图6-18：垂直对齐会使文本行变高

可以看到，所有垂直对齐的元素都会影响行高。应该记得行框的描述，其高度要足以包含最高行内框的顶端和最低行内框的底端。这包括因垂直对齐上升或下降的行内框。

字间隔和字母间隔

前面已经介绍了对齐，下面来看如何处理字间隔和字母间隔。同样地，这些属性存在一些不太直观的问题。

字间隔

word-spacing属性接受一个正长度值或负长度值。这个长度会增加到字之间的标准间隔。实际上，word-spacing用于修改字间间隔。因此，默认值normal与设置值为0是一样的。

word-spacing	
值：	<length> | normal | inherit
初始值：	normal
应用于：	所有元素
继承性：	有
计算值：	对于normal，为绝对长度0；否则，是绝对长度

如果提供一个正长度值，那么字之间的间隔就会增加。为word-spacing设置一个负值时，会把字拉近：

```
p.spread {word-spacing: 0.5em;}
p.tight {word-spacing: -0.5em;}
p.base {word-spacing: normal;}
p.norm {word-spacing: 0;}

<p class="spread">The spaces between words in this paragraph will be
increased
  by 0.5em.</p>
<p class="tight">The spaces between words in this paragraph will be
decreased
  by 0.5em.</p>
<p class="base">The spaces between words in this paragraph will be normal.</p>
<p class="norm">The spaces between words in this paragraph will be normal.</p>
```

处理这些设置后的效果如图6-19所示。

到目前为止，我还没有给出“字”的明确定义。用最简单的CSS术语来讲，“字”可以是任何非空白符字符组成的串，并由某种空白符包围。这个定义没有实际语义；它只是假设一个文档包含由一个或多个空白符包围的字。支持CSS的用户代理不一定能确定一

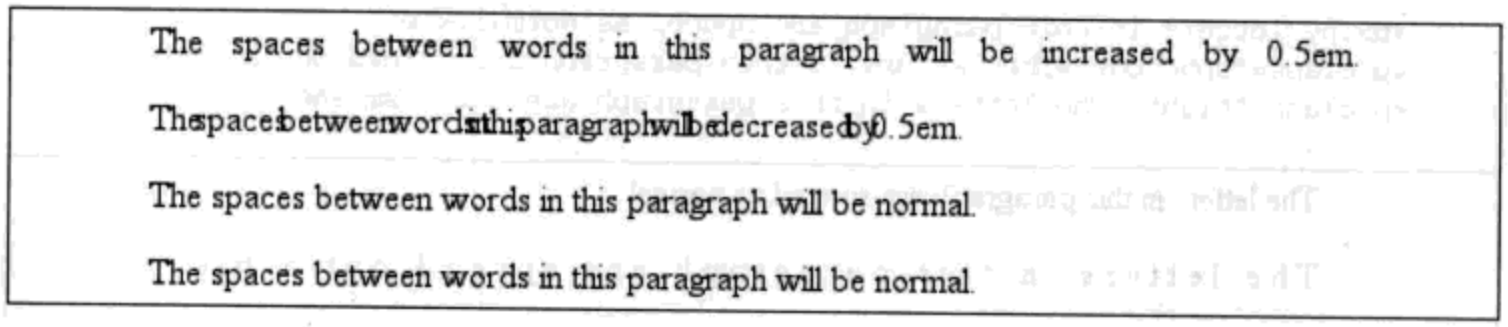

图6-19：改变字间间隔

个给定语言中哪些是合法的字，而哪些不是。尽管这个定义没有多大价值，不过它意味着采用象形文字的语言或非罗马书写体往往无法指定字间隔。利用这个属性，可能会创建很不可读的文档，图6-20 清楚地展示了这一点。所以，使用word-spacing时要当心。

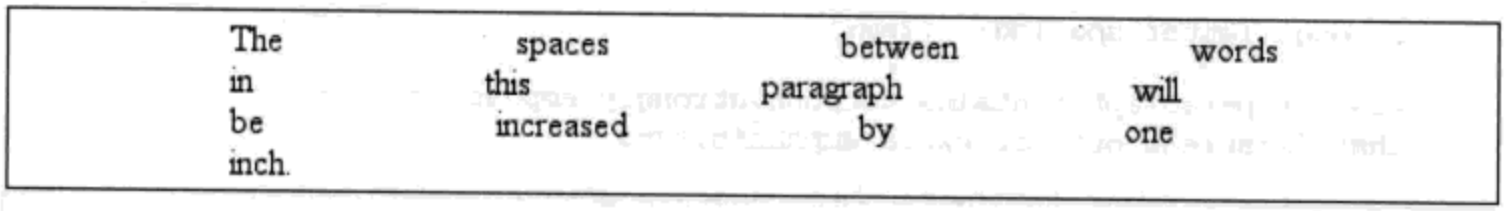

图6-20：字间隔太宽

字母间隔

word-spacing遇到的许多问题也同样出现在letter-spacing中。这二者之间唯一的真正区别是字母间隔修改的是字符或字母之间的间隔。

letter-spacing

值：	<length> | normal | inherit
初始值：	normal
应用于：	所有元素
继承性：	有
计算值：	对于长度值，为绝对长度；否则，为normal

像word-spacing属性一样，letter-spacing属性的可取值包括所有长度。默认关键字是normal（这与letter-spacing: 0相同）。输入的长度值会使字母间的间隔增加或减少指定的量。图6-21 所示为以下标记的结果：

```
p {letter-spacing: 0;}     /*  identical to 'normal'  */
p.spacious {letter-spacing: 0.25em;}
p.tight {letter-spacing: -0.25em;}
```

```
<p>The letters in this paragraph are spaced as normal.</p>
<p class="spacious">The letters in this paragraph are spread out a bit.</p>
<p class="tight">The letters in this paragraph are a bit smashed together.</p>
```

The letters in this paragraph are spaced as normal.

The letters in this paragraph are spread out a bit.

图6-21：各种字母间隔

可以使用letter-spacing来突出强调效果，这个技术可谓历史悠久。你可能会写以下声明，其效果如图6-22所示：

```
strong {letter-spacing: 0.2em;}

<p>This paragraph contains <strong>strongly emphasized text</strong>
that is spread out for extra emphasis.</p>
```

This paragraph contains **strongly emphasized text** that is spread out for extra emphasis.

图6-22：使用letter-spacing突出强调效果

间隔和对齐

word-spacing的值可能受text-align属性值的影响。如果一个元素是两端对齐的，字母和字之间的空间可能会调整，以便文本在整行中刚好放下。这可能又会改变创作人员用word-spacing声明的字间隔。如果为letter-spacing指定一个长度值，字符间隔则不会受text-align影响，但是如果letter-spacing的值是normal，字符间的间隔就可能改变，以便将文本两端对齐。CSS没有指定应当如何计算间隔，所以用户代理只是将其填满。

一般地，一个元素的子元素会继承该元素的计算值。无法为word-spacing或letter-spacing定义一个可继承的缩放因子来取代计算值（像line-height那样）。因此，可能会遇到图6-23所示的问题：

```
p {letter-spacing: 0.25em; font-size: 20px;}
small {font-size: 50%;}

<p>This spacious paragraph features <small>tiny text that is just
as spacious</small>, even though the author probably wanted the
spacing to be in proportion to the size of the text.</p>
```

This spacious paragraph features
tiny text that is just as spacious, even
though the author probably wanted
the spacing to be in proportion to
the size of the text.

图6-23：继承的字母间隔

如果字母间隔与文本大小成比例，得到字母间隔的唯一办法就是显式地设置，如下：

```
p {letter-spacing: 0.25em;}
small {font-size: 50%; letter-spacing: 0.25em;}
```

文本转换

下面来看如何使用text-transform属性处理文本的大小写。

text-transform

值：	uppercase | lowercase | capitalize | none | inherit
初始值：	none
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

默认值none对文本不做任何改动，将使用源文档中原有的大小写。顾名思义，uppercase和lowercase将文本转换为全大写或全小写字符。最后，capitalize只对每个单词的首字母大写。图6-24以多种方式展示了这些设置：

```
h1 {text-transform: capitalize;}
strong {text-transform: uppercase;}
p.cummings {text-transform: lowercase;}
p.raw {text-transform: none;}

<h1>The heading-one at the beginninG</h1>
<p>
By default, text is displayed in the capitalization it has in the source
document, but <strong>it is possible to change this</strong> using
the property 'text-transform'.
</p>
<p class="cummings">
```

```
For example, one could Create TEXT such as might have been Written by
the late Poet e.e.cummings.
</p>
<p class="raw">
If you feel the need to Explicitly Declare the transformation of text
to be 'none', that can be done as well.
</p>
```

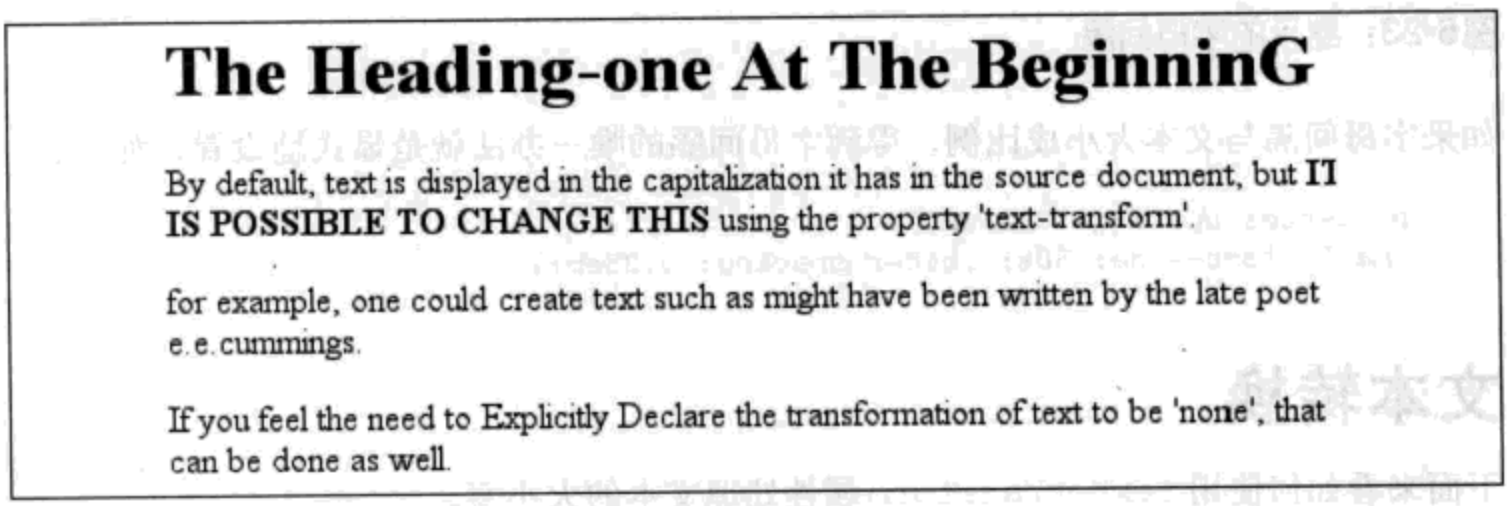

图6-24：各种文本转换

不同用户代理可能会用不同的方法来确定单词从哪里开始，相应地确定哪些字母要大写。例如，如图6-24所示，h1元素中的文本“heading-one”可以用两种方式显示：“Heading-one”或“Heading-One”。CSS并没有说哪一种是正确的，所以这两种都是可以的。

你可能还注意到，图6-24中h1元素的最后一个字母还是大写。这没有错：在应用text-transform值capitalize时，CSS只要求用户代理确保每个单词的首字母大写，可以忽略单词的余下部分。

作为一个属性，text-transform看上去可能无关紧要，不过如果你突然决定将所有h1元素都变成大写，这个属性就很有用。不必单独地修改所有h1元素的内容，只需使用text-transform为你完成这个修改：

```
h1 {text-transform: uppercase;}

<h1>This is an H1 element</h1>
```

使用text-transform有两方面的好处。首先，只需写一个简单的规则来完成这个修改，而无需修改h1元素本身。其次，如果你以后决定将所有大小写再切换为原来的大小写，可以更容易地完成修改，如图6-25所示：

```
h1 {text-transform: capitalize;}

<h1>This is an H1 element</h1>
```

This Is An H1 Element

图6-25：转换H1元素

文本装饰

接下来我们讨论text-decoration，这是一个很有意思的属性，它提供了很多非常有趣的行为。

text-decoration

值：	none |[underline || overline || line-through || blink]| inherit
初始值：	none
应用于：	所有元素
继承性：	无
计算值：	根据指定确定

不出所料，underline会对元素加下划线，就像HTML中的U元素一样。overline的作用恰好相反，会在文本的顶端画一个上划线。值line-through则在文本中间画一个贯穿线，这也称为贯穿文本，等价于HTML中的S和strike元素。blink会让文本闪烁，类似于Netscape支持的颇招非议的blink标记。图6-26显示了这些值的一些例子：

```
p.emph {text-decoration: underline;}
p.topper {text-decoration: overline;}
p.old {text-decoration: line-through;}
p.annoy {text-decoration: blink;}
p.plain {text-decoration: none;}
```

注意： 当然，无法在本书中显示闪烁的效果，不过很容易想像（也许实在太容易了）。有时，用户代理不要求支持blink，在写作本书时，Internet Explorer就不支持blink。

none值会关闭原本应用到一个元素上的所有装饰。通常，无装饰的文本是默认外观，但也不总是这样。例如，链接默认地会有下划线。如果你想去掉超链接的下划线，可以使用以下CSS规则来做到这一点：

```
a {text-decoration: none;}
```

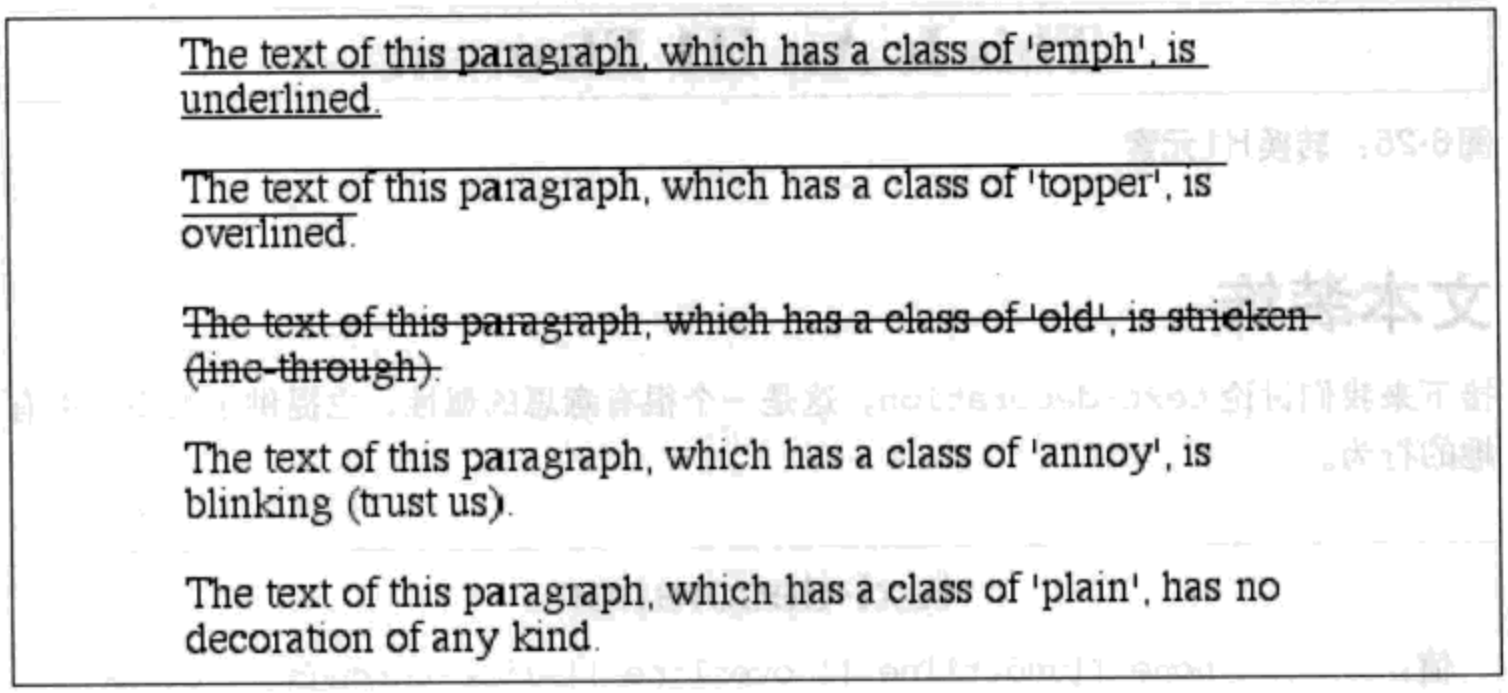

图6-26：各种文本装饰

如果显式地用这样一个规则去掉链接的下划线，那么锚与正常文本之间在视觉上的唯一差别就是颜色（至少默认是这样，不过也不能完全保证其颜色肯定有区别）。

注意： 尽管我个人对此没有什么意见，不过许多用户如果发现你去掉了链接的下划线会很不高兴。这取决于个人观点，所以你自己看着办。不过要记住：如果链接的颜色与正常文本的差别不够明显，用户将很难在文档中找到超链接。

还可以在一个规则中结合多种装饰。如果希望所有超链接既有下划线又有上划线，则规则如下：

```
a:link, a:visited {text-decoration: underline overline;}
```

不过要当心：如果两个不同的装饰都与同一个元素匹配，胜出规则的值会完全取代另一个值。考虑以下规则：

```
h2.stricken {text-decoration: line-through;}
h2 {text-decoration: underline overline;}
```

给定这些规则，所有class为stricken的h2元素都只有一个贯穿线装饰，而没有下划线和上划线装饰，因为text-decoration值会替换而不是累积起来。

怪异的装饰

下面来看text-decoration不寻常的一面。第一个奇怪的地方是text-decoration不

能继承。没有继承性意味着文本上画的任何装饰线（上划线或贯穿线）与父元素的颜色相同。即使后代元素本身有其他颜色也是如此，如图6-27所示：

```
p {text-decoration: underline; color: black;}
strong {color: gray;}

<p>This paragraph, which is black and has a black underline, also contains
<strong>strongly emphasized text</strong> which has the black underline
beneath it as well.</p>
```

This paragraph, which is black and has a black underline, also contains strongly emphasized text which has the black underline beneath it as well.

图6-27：下划线的颜色一致

为什么会这样？因为text-decoration的值不能继承，strong元素认为text-decoration默认值为none。因此，strong元素没有下划线。现在，在strong元素下面很显然有一条线，所以说它无下划线看上去很傻。但这并不傻。你在strong元素下面看到的是段落的下划线，它实际上只是“经过”了strong元素而已。如果修改粗体元素的样式，可以更清楚地看到这一点，如下：

```
p {text-decoration: underline; color: black;}
strong {color: gray; text-decoration: none;}

<p>This paragraph, which is black and has a black underline, also contains
<strong>strongly emphasized text</strong> which has the black underline beneath it as
well.</p>
```

其结果与图6-27一样，因为你所做的就是明确地声明已经有什么。换句话说，没有办法去掉父元素生成的下划线（或者上划线或贯穿线）。

text-decoration与vertical-align结合时，还会发生更奇怪的事情。图6-28显示了这样一种情况。因为sup元素没有自己的装饰，但是它在一个有上划线的元素中，这个上划线穿过了sup元素：

```
p {text-decoration: overline; font-size: 12pt;}
sup {vertical-align: 50%; font-size: 12pt;}
```

This paragraph, which is black and has a black overline, also contains superscripted text through which the overline will cut.

图6-28：正确但有些奇怪的装饰行为

因为文本装饰可能会造成这样一些问题，现在你可能声称再也不会使用文本装饰了。实

际上，我还只是指出了其中最简单的一些可能的情况，因为我们只是讨论了按照规范来讲会怎么样。在实际中，尽管不该去掉子元素的下划线，但有些Web浏览器确实会这么做。这些浏览器之所以违反规范，原因很简单：创作人员希望如此。考虑以下标记：

```
p {text-decoration: underline; color: black;}
strong {color: silver; text-decoration: none;}

<p>This paragraph, which is black and has a black underline, also contains
<strong>boldfaced text</strong> which does not have black underline
beneath it.</p>
```

图6-29所示为一个Web浏览器中去掉了strong元素的下划线。

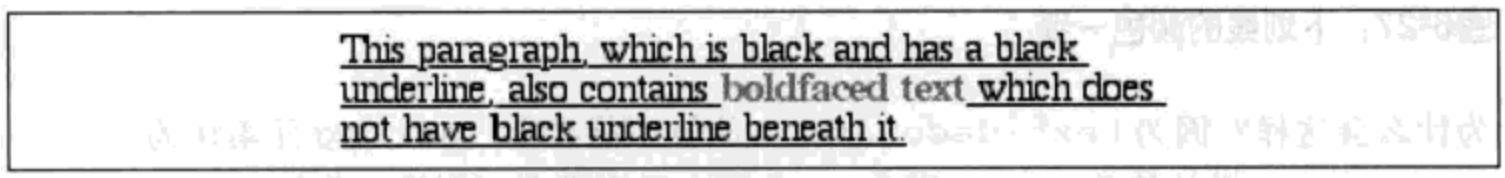

图6-29：一些浏览器的实际表现

这里有一个警告，很多浏览器确实会遵循规范，而且现有浏览器（或所有其他用户代理）的将来版本可能有一天会严格遵循规范。所以，如果你依赖于使用none来去掉装饰，要认识到重要的一点：将来这可能会给你带来麻烦，甚至现在就会出问题。而且，CSS的将来版本可能会包含一些去掉装饰的方法，而不必不正确地使用none，所以这方面还是有希望的。

还有一种方法可以改变装饰的颜色而不会违反规范。你应该记得，在一个元素上设置文本装饰意味着整个元素都有同样的颜色装饰，即使子元素有不同颜色。为了使装饰颜色与一个元素匹配，必须显式地声明其装饰，如下：

```
p {text-decoration: underline; color: black;}
strong {color: silver; text-decoration: underline;}

<p>This paragraph, which is black and has a black underline, also contains
<strong>strongly emphasized text</strong> which has the black underline
beneath it as well, but whose gray underline overlays the black underline
of its parent.</p>
```

在图6-30中，strong元素被设置为灰色，而且有一个下划线。灰色的下划线看上去“覆盖”了父元素的黑色下划线，所以装饰的颜色与strong元素的颜色匹配。

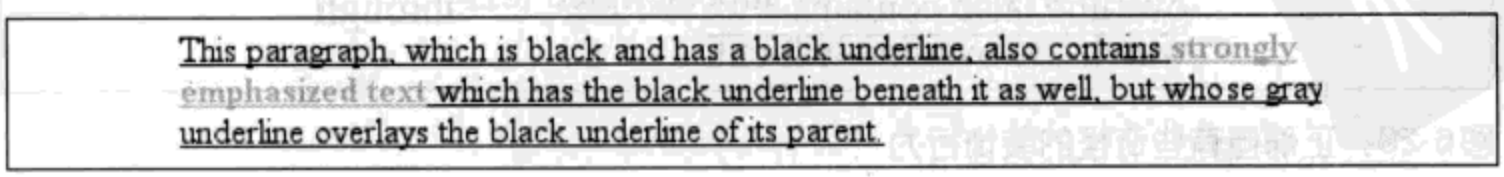

图6-30：覆盖下划线的默认行为

文本阴影

CSS2包含一个属性来为文本增加阴影，不过这个属性并没有在CSS2.1中保留，因为在CSS2.1完成前没有一个浏览器对此提供了充分的支持。可以考虑一下，如果要让Web浏览器确定一个元素中文本的轮廓，再计算一个或多个阴影——所有这些阴影还必须混合在一起而不与文本本身重叠——想想看这需要做多少工作，这样看来，规范中去掉阴影也是情有可原的。

text-shadow	
值：	none | [<color> || <length> <length> <length>? ,]* [<color> || <length> <length> <length>?] | inherit
初始值：	none
应用于：	所有元素
继承性：	无

显然，默认情况是文本没有阴影。否则，理论上可以定义一个或多个阴影。每个阴影都由一个颜色和3个长度值来定义。这个颜色当然设置了阴影的颜色，所以可以定义绿色、紫色甚至白色阴影。

前两个长度值确定了阴影与文本的偏移距离，第三个长度值可选，定义了阴影的“模糊半径”。要定义一个相对于文本向右偏移5像素向下偏移0.5em的绿色阴影，而且不模糊，可以写作：

```
text-shadow: green 5px 0.5em;
```

负长度值会使阴影落在原文本的左上方。

模糊半径定义为从阴影轮廓到模糊效果边界的距离。如果半径为2像素，模糊效果就会作用于阴影轮廓到模糊边界之间的空间。具体的模糊方法并未定义，所以不同的用户代理可能会有不同的效果。举例来说，以下样式可能得到如图6-31所示的显示：

```
p.cl1 {color: black; text-shadow: silver 2px 2px 2px;}
p.cl2 {color: white; text-shadow: 0 0 4px black;}
p.cl3 {color: black; text-shadow: 1em 1em 5px gray, -1em -1em silver;}
```

注意： 图6-31是用Photoshop生成的，因为写作本书时Web浏览器不支持text-shadow。

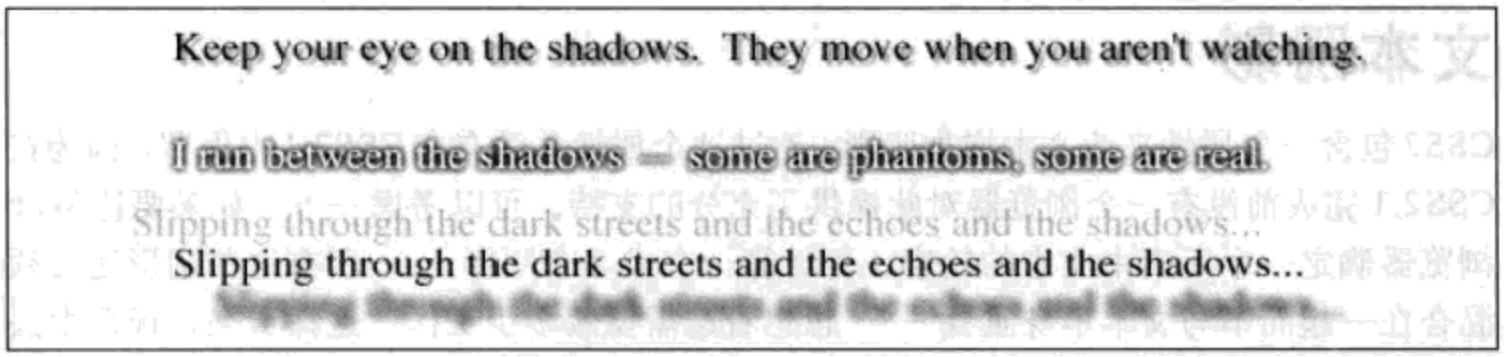

图6-31：到处都是阴影

处理空白符

我们已经介绍了很多对文本应用样式的方法，下面来讨论属性white-space，它会影响用户代理对源文档中的空格、换行和tab字符的处理。

white-space

值：	normal | nowrap | pre | pre-wrap | pre-line | inherit
初始值：	normal
应用于：	所有元素（CSS2.1）；块级元素（CSS1和CSS2）
继承性：	无
计算值：	根据指定确定

使用这个属性，可以影响浏览器处理字之间和文本行之间的空白符的方式。从某种程度上讲，默认的XHTML处理已经完成了空白符处理：它会把所有空白符合并为一个空格。所以给定以下标记，它在Web浏览器中显示时，各个字之间只会显示一个空格，而且忽略元素中的换行。

```
<p>This    paragraph    has      many
    spaces          in it.</p>
```

可以用以下声明显式地设置这种默认行为：

```
p {white-space: normal;}
```

这个规则告诉浏览器按平常的做法去做：丢掉多余的空白符。给定这个值，换行字符（回车）会转换为空格，一行中多个空格的序列也会转换为一个空格。

不过，如果将white-space设置为pre，受这个属性影响的元素中，空白符的处理就有所不同，就好像元素是XHTML pre元素一样：空白符不会被忽略，如图6-32所示：

```
p {white-space: pre;}

<p>This    paragraph    has      many
    spaces          in it.</p>
```

This paragraph has many
spaces in it.

图6-32：保留标记中的空格

不过，如果white-space值为pre，浏览器将会注意额外的空格，甚至回车。在这个方面，而且仅在这个方面，任何元素都可以相当于一个pre元素。

与之相对的值是nowrap，它会防止元素中的文本换行，除非使用了一个br元素。在CSS中使用nowrap非常类似于HTML 4中用<td nowrap>将一个表单元格设置为不能换行，不过white-space值可以应用到任何元素。以下标记的效果如图6-33所示：

```
<p style="white-space: nowrap;">This paragraph is not allowed to wrap,
which means that the only way to end a line is to insert a line-break
element.  If no such element is inserted, then the line will go forever,
forcing the user to scroll horizontally to read whatever can't be
initially displayed <br/>in the browser window.</p>
```

This paragraph is not allowed to wrap, which means that the only way to end a line is tc
in the browser window.

图6-33：用white-space属性防止换行

还可以使用white-space来替换表单元格上的nowrap属性：

```
td {white-space: nowrap;}

<table><tr>
<td>The contents of this cell are not wrapped.</td>
<td>Neither are the contents of this cell.</td>
<td>Nor this one, or any after it, or any other cell in this table.</td>
<td>CSS prevents any wrapping from happening.</td>
</tr></table>
```

CSS2.1引入了值pre-wrap和pre-line，这在以前版本的CSS中是没有的。这些值的作用是允许创作人员更好地控制空白符处理。

如果一个元素的white-space被设置为pre-wrap，那么该元素中的文本会保留空白符序列，但是文本行会正常地换行。如果设置为这个值，源文本中的行分隔符以及生成的行分隔符也会保留。pre-line与pre-wrap相反，会像正常文本中一样合并空白符序列，但保留换行符。例如，考虑以下标记，其效果如图6-34所示：

```
<p style="white-space: pre-wrap;">
This paragraph      has a great   many   s p a c e s   within  its textual
  content,   but their    preservation     will    not    prevent   line
    wrapping or line breaking.
</p>
<p style="white-space: pre-line;">
This paragraph      has a great   many   s p a c e s   within  its textual
  content,   but their collapse  will    not    prevent   line
    wrapping or line breaking.
</p>
```

This paragraph has a great many s p a c e s within its textual content, but their preservation will not prevent line wrapping or line breaking.

This paragraph has a great many s p a c e s within its textual content, but their collapse will not prevent line wrapping or line breaking.

图6-34：处理空白符的两种不同方法

表6-1总结了white-space属性的行为。

表6-1：white-space属性

值	空白符	换行符	自动换行
pre-line	合并	保留	允许
normal	合并	忽略	允许
nowrap	合并	忽略	不允许
pre	保留	保留	不允许
pre-wrap	保留	保留	允许

文本方向

如果你读的是本书英文版或其他语言的版本，就会从左到右、从上到下地阅读，这就是英语的流方向。不过，并不是所有语言都是如此。还有许多从右向左读的语言，如希伯来语和阿拉伯语等，CSS2引入了一个属性来描述其方向性。

direction	
值：	ltr | rtl | inherit
初始值：	ltr
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

direction属性影响块级元素中文本的书写方向、表中列布局的方向、内容水平填充其元素框的方向，以及两端对齐元素中最后一行的位置。对于行内元素，只有当unicode-bidi 属性设置为 embed 或bidi-override时才会应用 direction 属性（见以下关于unicode-bidi 的描述）。

注意： 在CSS3之前，CSS规范中没有涵盖从上到下读的语言。在写作本书时，CSS3 Text Module还是一个候选推荐，它解决了这个问题，引入了一个新属性writing-mode。

尽管ltr是默认值，但可以想见，如果浏览器在显示从右向左读的文本，默认值会改为rtl。因此，浏览器可能会有以下内部规则：

```
*:lang(ar), *:lang(he) {direction: rtl;}
```

实际的规则可能更长一些，将涵盖所有从右向左读的语言，而不只是阿拉伯语和希伯来语，不过这也能说明问题。尽管CSS试图处理书写方向，但Unicode有一种更健壮的方法来处理方向性。利用属性unicode-bidi，CSS创作人员可以充分利用Unicode的某些功能。

unicode-bidi	
值：	normal | embed | bidi-override | inherit
初始值：	normal
应用于：	所有元素
继承性：	无
计算值：	根据指定确定

在此我们将简要地引用CSS2.1规范中关于这些值的描述，这些描述很好地说明了各个值的实质。

`normal`

元素不会对双向算法打开附加的一层嵌套。对于行内元素，顺序的隐式重排会跨元素边界进行。

`embed`

如果是一个行内元素，这个值对于双向算法会打开附加的一层嵌套。这个嵌套层的方向由`direction`属性指定。会在元素内部隐式地完成顺序重排。这对应于在元素开始处增加一个LRE（对于`direction: ltr`；U+202A）或RLE（对于`direction: rtl`；U+202B），并在元素的最后增加一个PDF（U+202C）。

`bidi-override`

这会为行内元素创建一个覆盖。对于块级元素，将为不在另一块中的行内后代创建一个覆盖。这说明，顺序重排在元素内部严格按`direction`属性进行；忽略了双向算法的隐式部分。这对应于在元素开始处增加一个LRO（对于`direction: ltr`；U+202D）或RLO（对于`direction: rtl`；U+202E），并在元素最后增加一个PDF（U+202C）。

小结

即使不改变所用的字体，还是有很多方法来改变文本的外观。除了一些经典的效果（如加下划线）外，CSS还允许在文本上面画线，或穿越文本画线；改变单词和字母间的间隔大小；将段落（或其他块级元素）的首行缩进；将文本左对齐或右对齐，等等。甚至可以修改文本行间的间隔大小，不过这个操作太过复杂，在第7章再详细说明。

这些行为有些得到了很好的支持，有些则根本不被支持。文本两端对齐就是一个没有得到充分支持的行为，20世纪发布的大多数用户代理在文本装饰和垂直对齐方面都存在bug，另外在行高计算上也存在问题。另一方面，如果用户代理支持单词和字母间隔，则总能正常工作，另外文本缩进也只是有很少的一些小bug。改变大小写也是如此，用户代理总能正确地支持这个方面。

在本章中我提到过，行布局比我们谈到的还要复杂，下一章将介绍这个过程的详细内容，同时还将介绍很多其他知识。

第7章

基本视觉格式化

在前面几章中，我们针对CSS如何处理文档中的文本和字体介绍了很多实用知识。这一章中，我们将介绍视觉表现的一些理论方面，另外，前面为了强调CSS如何实现而跳过了一些问题，这些问题也将在这一章中回答。

为什么要用一整章来讨论CSS中关于视觉显示的这些理论基础呢？答案是，CSS包含如此开放、如此强大的一个模型，对于这样一个模型，可以有无数种方法结合应用各种属性，可以得到的效果也数不胜数，所以没有哪本书能全面涵盖每一种可能。你肯定还会发现新的方法来通过使用CSS得到你自己想要的文档效果。

在研究CSS的过程中，你可能会发现用户代理有一些看上去很奇怪的行为。如果全面地掌握了CSS中视觉表现模型是如何工作的，你就能确定一种行为到底是CSS所定义表现引擎的正确结果（尽管出乎意料），还是一个需要报告的bug。

基本框

CSS假定每个元素都会生成一个或多个矩形框，这称为元素框（规范的将来版本可能允许非矩形的框，不过对现在来说，框都是矩形的）。各元素框中心有一个内容区（content area）。这个内容区周围有可选的内边距、边框和外边距。这些项之所以被认为是可选的，原因是它们的宽度可以设置为0，实际上这就从元素框去除了这些项。图7-1显示了一个示例内容区，这个内容区的周围还有内边距、边框和外边距。

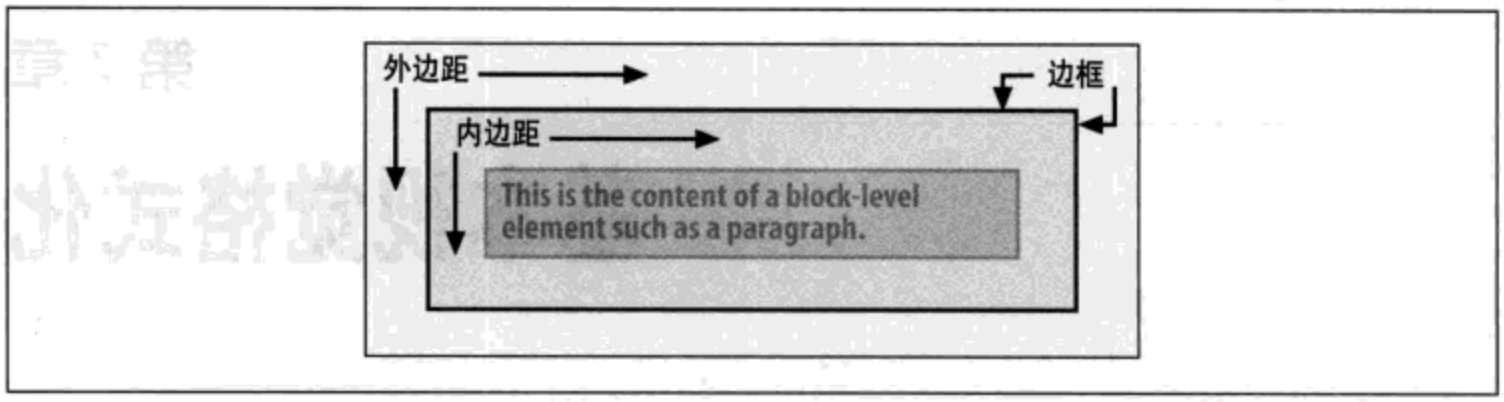

图7-1：内容区及其外围项

可以用多种属性设置各外边距、边距和内边距，如margin-left或border-bottom。内容的背景（例如某种颜色或平铺图像）也会应用到内边距。外边距通常是透明的，从中可以看到父元素的背景。内边距不能是负值，但是外边距可以。本章后面将探讨外边距值为负时的效果。

边框使用已定义样式生成，如solid或inset，边框的颜色使用border-color属性设置。如果没有设置颜色，那么边框将取元素内容的前景色。例如，如果一个段落的文本是白色，那么该段落的所有边框都是白色，除非创作人员显式地声明了另外一种边框色。如果边框样式有某种缝隙，则可以通过这些缝隙看到元素的背景。换句话说，边框与内容和内边距有相同的背景。最后要说明的是，边框的宽度绝对不能为负。

注意： 元素框的各个部分可能受很多属性的影响，如width或border-right。其中很多属性将在本章中用到，尽管目前我们还没有讨论过这些属性。具体的属性定义将在第8章给出，那时才会介绍有关的概念。

不过你会发现，对不同类型的元素格式化时存在着差别。块级元素的处理就不同于行内元素，而浮动元素和定位元素也分别有各自不同的表现。

包含块

每个元素都相对于其包含块摆放；可以这么说，包含块就是一个元素的“布局上下文”。CSS2.1定义了一系列规则来确定元素的包含块。这里介绍的只是其中的部分规则，这些规则的概念都将在本章加以明确，而其他规则将在后面的章节中介绍。

对于正常的西方语言文本流中的一个元素，包含块由最近的块级祖先框、表单元格或行内块祖先框的内容边界（content edge）构成。考虑下面的标记：

```
<body>
 <div>
```

```
    <p>This is a paragraph.</p>
  </div>
</body>
```

在这个非常简单的例子中，p元素的包含块是div元素，因为作为块级元素、表单元格或行内块元素，这是最近的祖先元素（本例中是一个块元素框）。类似地，div的包含块是body。因此。p的布局依赖于div的布局，而div的布局则依赖于body的布局。

不必担心行内元素，因为它们的摆放方式并不直接依赖于包含块。本章后面还会讨论有关内容。

快速复习

下面来快速地回顾一下我们讨论的各种元素，同时还将谈到一些重要的术语，这些术语对于理解本章的概念很重要。

正常流

这是指西方语言文本从左向右、从上向下显示，这也是我们熟悉的传统HTML文档的文本布局。注意，在非西方语言中，流方向可能不同。大多数元素都在正常流中，要让一个元素不在正常流中，唯一的办法就是使之成为浮动或定位元素（将在第10章介绍）。要记住，本章只讨论正常流中的元素。

非替换元素

如果元素的内容包含在文档中，则称之为非替换元素。例如，如果一个段落的文本内容都放在该元素本身之内，这个段落就是一个非替换元素。

替换元素

这是指用作为其他内容占位符的一个元素。替换元素的一个经典例子就是img元素，它只是指向一个图像文件，这个文件将插入到文档流中该img元素本身所在位置。大多数表单元素也可以替换（例如，<input type="radio">）。

块级元素

这是指段落、标题或div之类的元素。这些元素在正常流中时，会在其框之前和之后生成“换行”，所以处于正常流中的块级元素会垂直摆放。通过声明display: block，可以让元素生成块级框。

行内元素

这是指strong或span之类的元素。这些元素不会在之前或之后生成“行分隔符”，它们是块级元素的后代。通过声明display: inline，可以让元素生成一个行内框。

根元素

位于文档树顶端的元素。在HTML文档中，这就是元素html。在XML文档中，则可以是该语言允许的任何元素。

块级元素

块级元素的表现有时可以预测，有时则很让人惊讶。例如，元素沿横轴和竖轴摆放时，其处理就可能不同。为了充分了解如何处理块级元素，必须对一些边界和区域很清楚。图7-2详细显示了这些边界和区域。

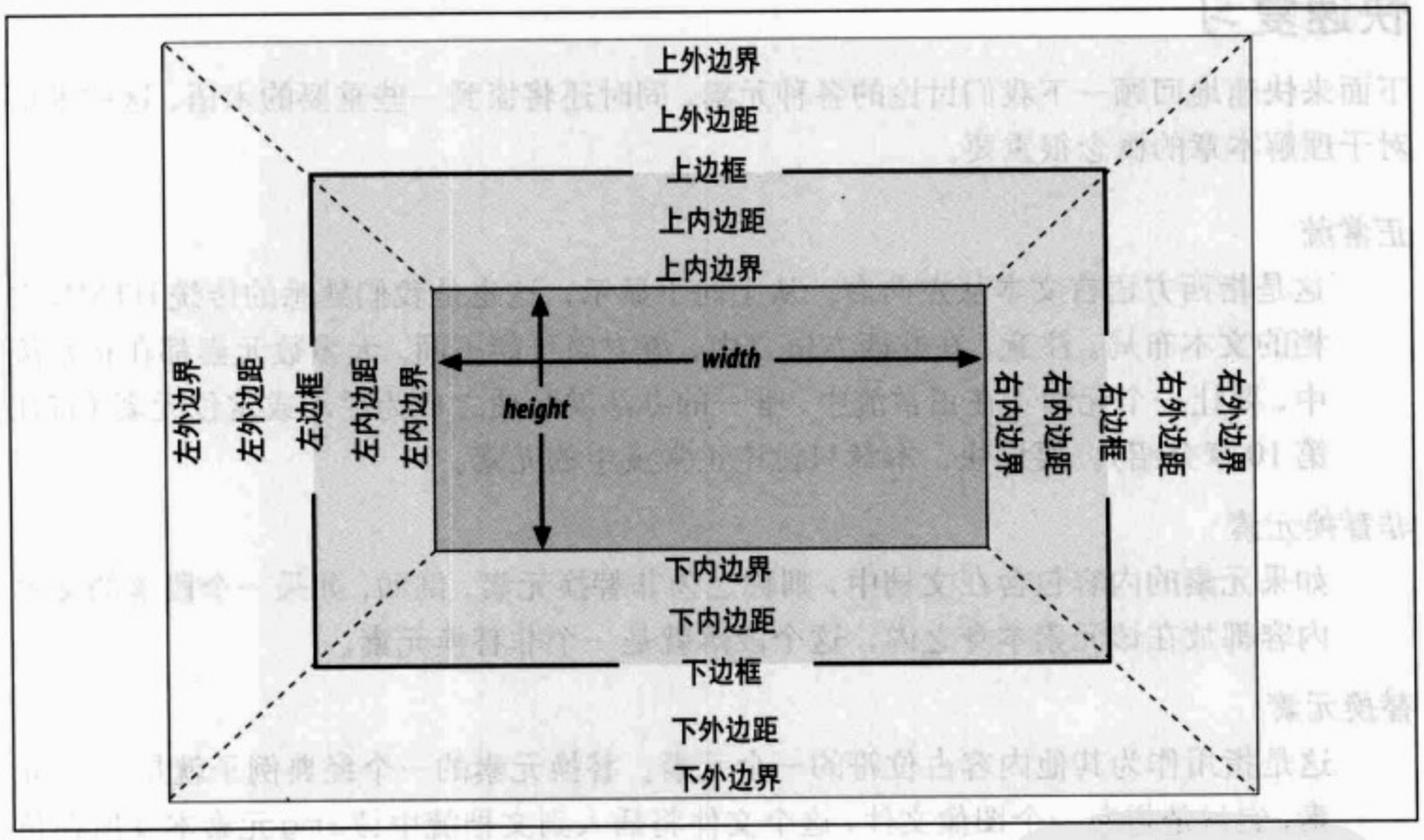

图7-2：完整的框模型

一般地，一个元素的width被定义为从左内边界到右内边界的距离，height则是从上内边界到下内边界的距离。这些属性都可以应用到元素。

不同的宽度、高度、内边距和外边距相结合，就可以确定文档的布局。在大多数情况下，文档的高度和宽度由浏览器自动确定，这要基于可用的显示区域和其他一些因素。当然在CSS下，可以更直接地控制元素的大小以及显示方式。对于水平和垂直布局，可以选择不同的效果，所以我们将分别介绍。

水平格式化

水平格式化往往比你想象得更复杂。其部分复杂性在于width影响的是内容区的宽度，而不是整个可见的元素框。考虑以下例子：

```
<p style="width: 200px;">wideness?</p>
```

这行代码使段落的内容区宽度为200像素。如果为元素指定一个背景，就能很清楚地看出。不过，如果还指定了内边距、边框或外边距，这些都会增加到宽度值。假设指定如下：

```
<p style="width: 200px; padding: 10px; margin: 20px;">wideness?</p>
```

可见元素框的宽度现在是220像素，因为在内容的左边和右边分别增加了10像素的内边距。外边距则会在左右两边再延伸20像素，使整个元素框的宽度为260像素。

一定要知道这样会隐式地增加width值，理解这一点很重要。大多数用户认为，width是指可见元素框的宽度，如果为一个元素声明了内边距、边框以及宽度，他们指定的宽度值则是左外边界到右外边界的距离。但在CSS中并不是这样。一定要牢牢记住这一点，以免以后糊涂。

注意： 在写作本书时，CSS的Box Model模块提供了一些建议，提出了一些方法来允许创作人员决定width是指内容宽度还是可见框宽度。

对此有一个很简单的规则，正常流中块级元素框的水平部分总和就等于父元素的width。假设一个div中有两个段落，这两个段落的外边距设置为1em。段落的内容宽度（width的值）再加上其左、右内边距，边框或外边距，加在一起正好是div内容区的width。

假设div的width为30em，那么各段落内容宽度、内边距、边框或外边距的总和就是30em。在图7-3中，段落外的“空白”实际上是其外边距。如果div有内边距，还会有更大的空白，不过这里div没有内边距。稍后就会讨论内边距。

水平属性

水平格式化的“7大属性”是：margin-left、border-left、padding-left、width、padding-right、border-right和margin-right。这些属性与块级框的水平布局有关，如图7-4所示。

这7个属性的值加在一起必须是元素包含块的宽度，这往往是块元素的父元素的width值（因为块级元素的父元素几乎都是块级元素）。

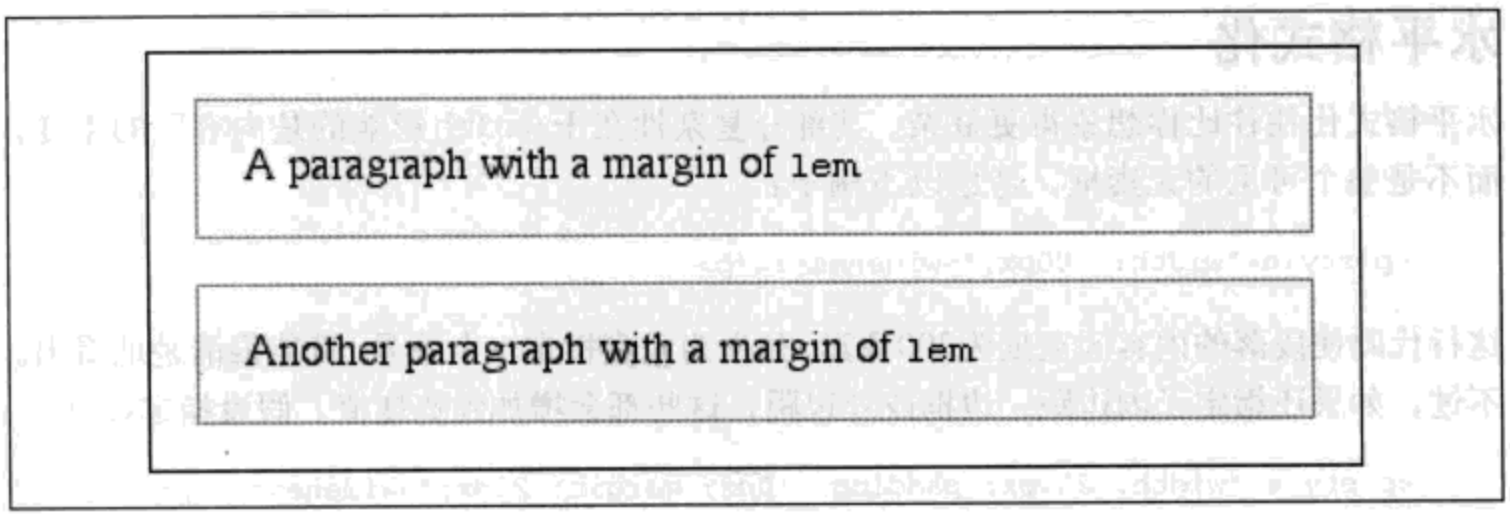

图7-3：元素框与其父元素的width相同

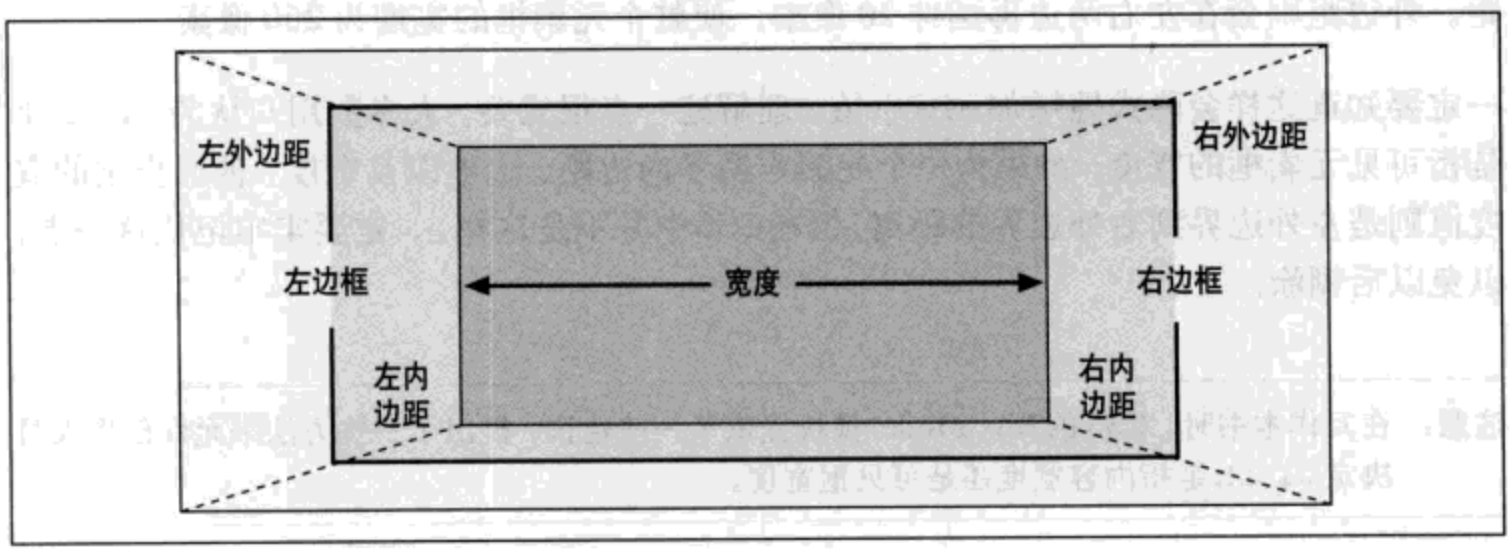

图7-4：水平格式化的“7大属性”

在这7个属性中，只有3个属性可以设置为auto:元素内容的width，以及左、右外边距。其余属性必须设置为特定的值，或者默认宽度为0。图7-5显示了框中的哪些部分可以取值为auto，而哪些部分不能。

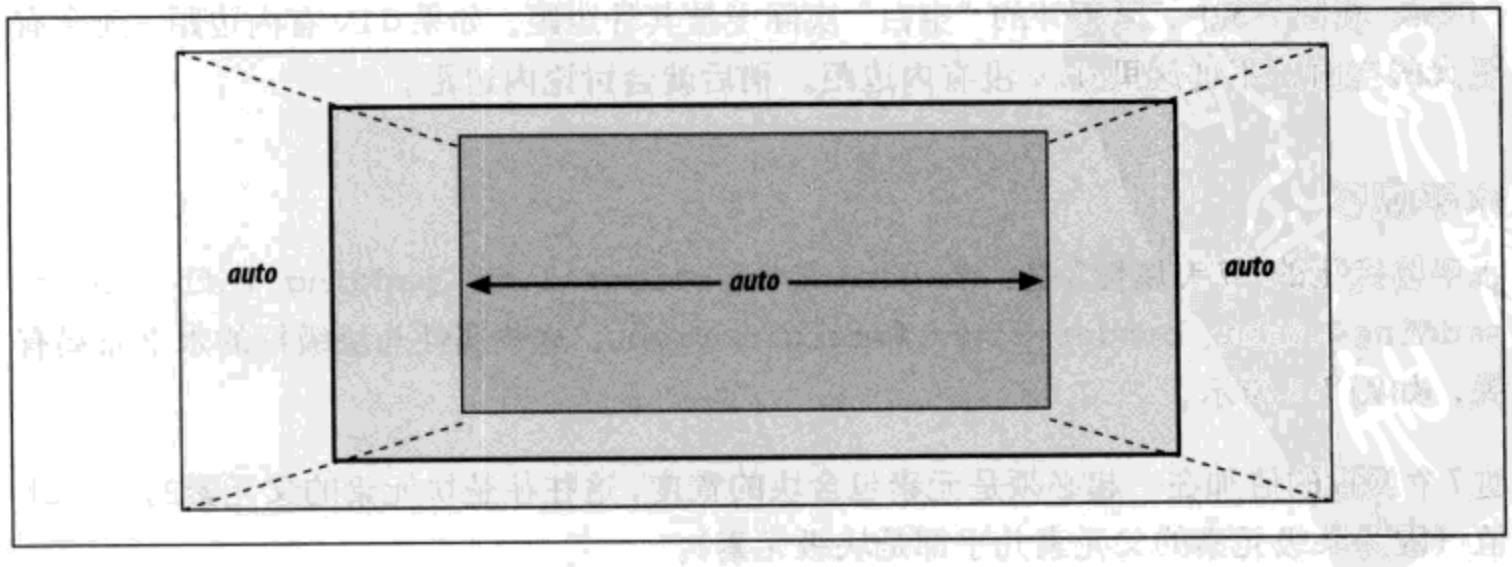

图7-5：可以设置为auto的水平属性

width必须设置为auto或某种类型的非负值。如果在水平格式化中确实使用了auto，会得到不同的效果。

注意： CSS允许浏览器为width设置一个最小值；块级元素的width不能小于这个值。对于不同浏览器，这个最小值可能不同，因为在规范中对此没有定义。

使用 auto

如果设置width、margin-left或margin-right中的某个值为auto，而余下两个属性指定为特定的值，那么设置为auto的属性会确定所需的长度，从而使元素框的宽度等于父元素的width。换句话说，假设7个属性的和必须等于400像素，没有设置内边距或边框，而且右外边距和width设置为100px，左外边距设置为auto。那么左外边距的宽度将是200像素：

```
p {margin-left: auto; margin-right: 100px;
  width: 100px;}    /* 'auto' left margin evaluates to 200px */
```

从某种程度上讲，可以用auto弥补实际值与所需总和的差距。不过，如果这3个属性都设置为100px，即没有任何一个属性设置为auto会怎么样呢？

如果所有这3个属性都设置为非auto的某个值——或者，按CSS的术语来讲，这些格式化属性过分受限（overconstrained）——或者，此时总会把margin-right强制为auto。这意味着，如果外边距和width都设置为100px，用户代理将把右外边距重置为auto。右外边距的实际宽度则会根据有一个auto值时的规则来设置，即由这个auto值“填补”所需的距离，使元素的总宽度等于其包含块的width。图7-6显示了以下标记的结果：

```
p {margin-left: 100px; margin-right: 100px;
  width: 100px;}    /* right margin forced to be 200px */
```

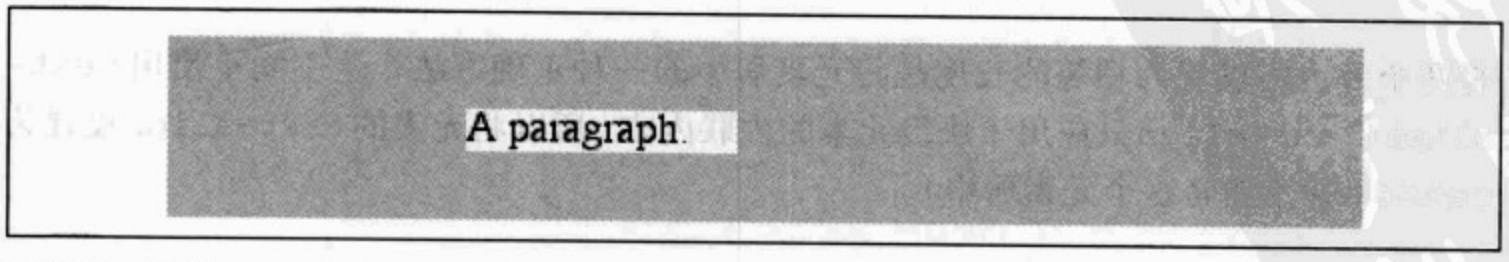

图7-6：覆盖margin-right设置

注意： 只是对从左向右读的语言（如英语）将margin-right强制为auto。如果是从右向左读的语言，一切正相反，所以会把margin-left强制为auto，而不是margin-right。

如果两个外边距都显式地设置，而width设置为auto，width值将设置为所需的某个值，从而达到需要的总宽度（即父元素的内容宽度）。以下标记的结果如图7-7所示：

```
p {margin-left: 100px; margin-right: 100px; width: auto;}
```

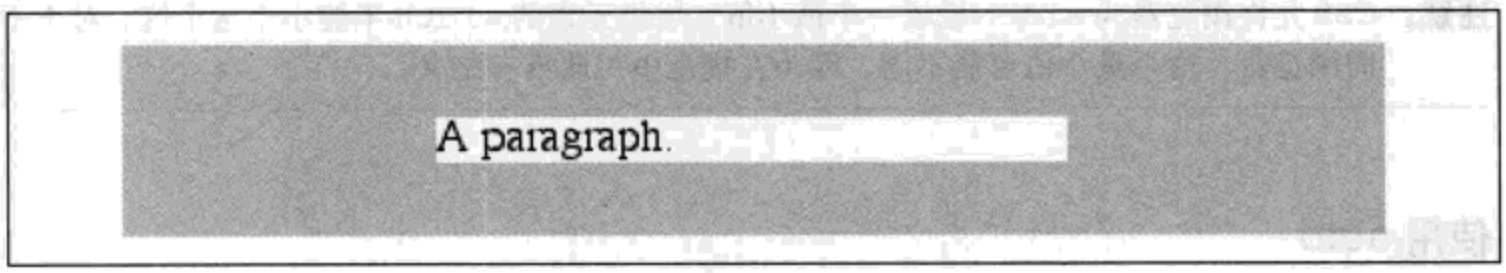

图7-7：自动宽度

图7-7中所示例子最为常见，因为这等价于只设置外边距，而没有为width作任何声明。以下标记的结果与图7-7所示完全相同：

```
p {margin-left: 100px; margin-right: 100px;} /* same as before */
```

不只一个auto

下面来看如果这3个属性（width、margin-left或margin-right）中有两个都设置为auto会出现什么情况。如果两个外边距都设置为auto，如以下代码所示，它们会设置为相等的长度，因此将元素在其父元素中居中，如图7-8所示：

```
p {width: 100px; margin-left: auto; margin-right: auto;}
```

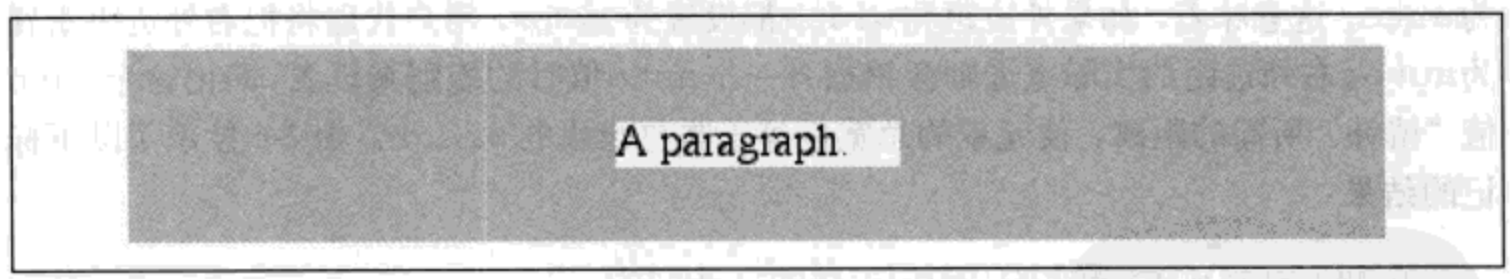

图7-8：显式设置width

将两个外边距设置为相等的长度是将元素居中的一种正确方法，这不同于使用text-align（text-align只应用于块级元素的内联内容，所以将元素的text-align设置为center并不能将这个元素居中）。

警告： 在实际中，只有1999年2月以后发布的浏览器能正确地处理auto外边距居中，而不是所有浏览器都能完全做对。不能正确处理auto外边距的浏览器表现也各不相同，不过可以肯定的一点是，老式的浏览器会把两个外边距都重置为0。

设置元素大小的另一种方法是将某个外边距以及width设置为auto。设置为auto的外边距会减为0：

```
p {margin-left: auto; margin-right: 100px;
 width: auto;}  /* left margin evaluates to 0 */
```

然后width会设置为所需的值，使得元素完全填充其包含块。

最后一点，如果这3个属性都设置为auto会怎么样呢？答案很简单：两个外边距都会设置为0，而width会尽可能宽。这种结果与默认情况是相同的，即没有为外边距或width显式声明任何值。在这种情况下，外边距默认为0，width默认为auto。

注意，由于水平外边距不会合并，父元素的内边距、边距和外边距可能影响其子元素。这种效果是间接的，即一个元素的外边距（以及内边距、边距等等）可能会为子元素带来偏移。以下标记的结果如图7-9所示：

```
div {padding: 30px; background: silver;}
p {margin: 20px; padding: 0; background: white;}
```

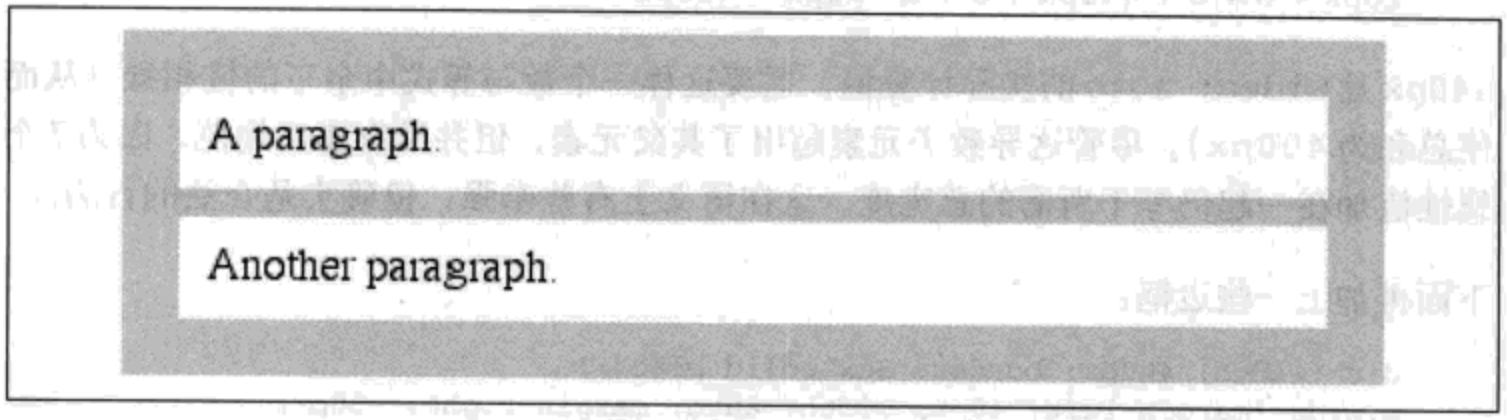

图7-9：父元素的外边距和内边距带来的隐式偏移

负外边距

到目前为止，看上去一切都很直接明了，你可能会奇怪前面为什么会说情况可能很复杂。外边距还有一个方面很特殊：外边距可以为负。这种设置是对的，完全可以将外边距设置为负值。这么做会带来一些有意思的效果（假设用户代理完全支持这种负外边距）。

注意： 按照CSS规范，用户代理不要求完全支持负外边距。规范指出："外边距属性允许为负值，不过可以有一些特定于具体实现的限制。"不过在写作本书时，当前浏览器中这种限制很少（几近没有）。

要记住，7个水平属性的总和要等于父元素的width。只要所有属性都是大于或等于0的，元素就不会大于其父元素的内容区。不过，考虑以下标记，其结果如图7-10所示：

```
div {width: 400px; border: 3px solid black;}
p.wide {margin-left: 10px; width: auto; margin-right: -50px; }
```

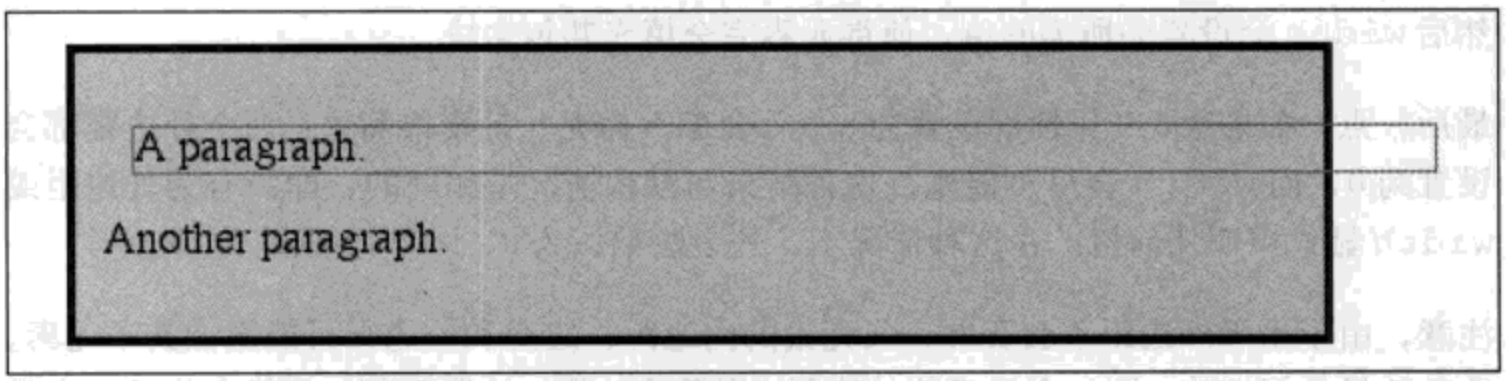

图7-10：通过指定负外边距得到更宽的子元素

不错，子元素确实比其父元素还宽！数学计算并没有错误：

10px + 0 + 0 + 440px + 0 + 0 - 50px = 400px

440px是width: auto的实际计算值，需要这样一个数与等式中余下的值相抵（从而使总和为400px）。尽管这导致子元素超出了其父元素，但并没有违反规范，因为7个属性值加在一起仍等于所需的总宽度。这在语义上有些牵强，但确实是合法的行为。

下面再加上一些边框：

```
div {width: 400px; border: 3px solid black;}
p.wide {margin-left: 10px; width: auto; margin-right: -50px;
  border: 3px solid gray;}
```

这样一来，计算出的width值会减少：

10px + 3px + 0 + 434px + 0 + 3px - 50px = 400px

如果还要设置内边距，width值会进一步减少。

与此相反，还有可能将auto右外边距计算为负值。如果其他属性的值要求右外边距为负，以便满足元素不能比其包含块更宽的需求（译注1），就会出现这种情况。考虑以下规则：

```
div {width: 400px; border: 3px solid black;}
p.wide {margin-left: 10px; width: 500px; margin-right: auto;
  border: 3px solid gray;}
```

译注1：严格地说，需求应该是“元素的7项水平属性的总和”不能比其包含块更宽。

等式如下：

10px + 3px + 0 + 500px + 0 + 3px - 116px = 400px

右外边距计算为-116px。即使为它指定了另一个值(即所有水平属性都指定为特定值，而不是auto)，由于元素水平属性过分受限时有一个规则，要求重置右外边距，这也会得到一个负右外边距。此时右外边距重置为所需的值，以保证元素水平属性的总和等于其父元素的内容宽度(不过，从右向左读的语言例外，对于这些语言，将重置左外边距)。

下面考虑另一个例子，如图7-11所示，这里左外边距设置为负值：

```
div {width: 400px; border: 3px solid black;}
p.wide {margin-left: -50px; width: auto; margin-right: 10px;
  border: 3px solid gray;}
```

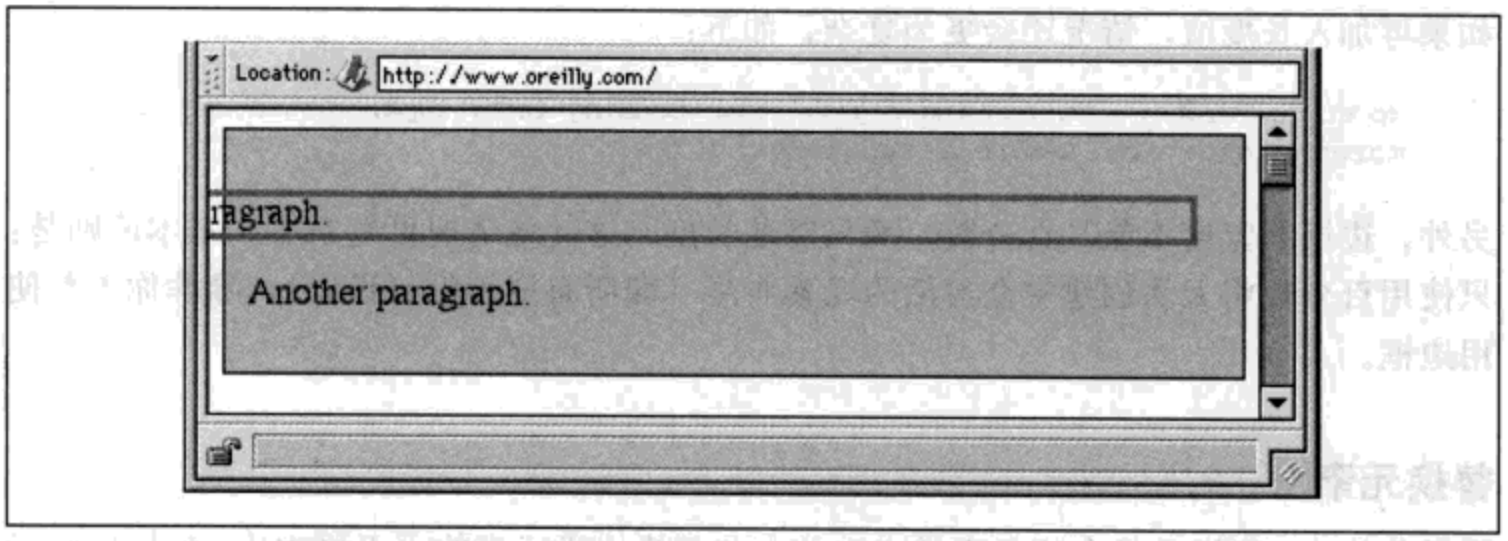

图7-11：设置一个负左外边距

如果左外边距为负，不仅段落会超出div的边框，还会超出浏览器窗口本身的边界！

注意：要记住，内边距、边框和内容宽度（及高度）绝对不能为负。只有外边距能小于0。

百分数

如果width、内边距和外边距设置为百分数值，会应用同样的基本规则。值声明为长度还是百分数并不重要。

百分数可能很有用。假设你希望一个元素的内容是其包含块宽度的2/3，左、右内边距分别设置为5%，左外边距为5%，余下的为右外边距。可以写作：

```
<p style="width: 67%; padding-right: 5%; padding-left: 5%; margin-right: auto;
margin-left: 5%;">playing percentages</p>
```

右外边距会计算为包含块宽度的18%（100% − 67% − 5% − 5% − 5%）。

不过，如果混合使用百分数和长度单位，可能很麻烦。考虑以下例子：

```
<p style="width: 67%; padding-right: 2em; padding-left: 2em; margin-right: auto;
margin-left: 5em;">mixed lengths</p>
```

在这种情况下，元素框可能定义如下：

5em + 0 + 2em + 67% + 2em + 0 + auto = 包含块宽度

为了让右外边距的宽度计算为0，元素包含块的宽度必须是27.272727em（元素内容区宽度为18.272727em）。如果比这宽，右外边距就会计算为一个正值；而比这窄时，右外边距则会计算为一个负值。

如果再加入长度值，情况还会更为复杂，如下：

```
<p style="width: 67%; padding-right: 15px; padding-left: 10px; margin-right: auto;
margin-left: 5em;">more mixed lengths</p>
```

另外，边框的宽度不能是百分数，而只能是长度，这就使情况更复杂了。基本原则是：只使用百分数将无法创建完全灵活的元素布局（即所有属性都可设置），除非你不想使用边框。

替换元素

到目前为止，我们已经介绍了正常文本流中非替换块级元素的水平格式化。替换块级元素管理起来更简单一些。非替换块元素的所有规则同样适用于替换块元素，只有一个例外：如果width为auto，元素的宽度则是内容的固有宽度。下例中的图像宽度是20像素，因为这正是原图像的宽度：

```
<img src="smile.png" style="display: block; width: auto; margin: 0;">
```

如果实际图像的宽度是100像素，那么元素的宽度也将是100像素。

可以为width指定一个特定值覆盖这个规则。假设将前例修改如下，将这个图像显示3次，每一次的width值都不同：

```
<img src="smile.png" style="display: block; width: 25px; margin: 0;">
<img src="smile.png" style="display: block; width: 50px; margin: 0;">
<img src="smile.png" style="display: block; width: 100px; margin: 0;">
```

结果如图7-12所示。

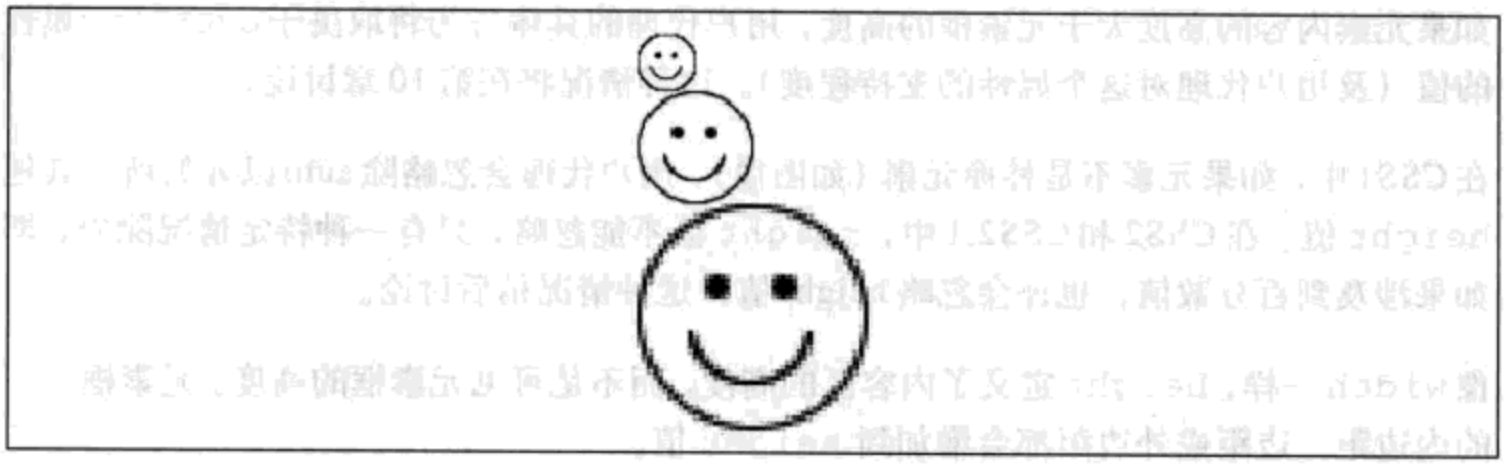

图7-12：改变替换元素的宽度

注意，元素的高度也会增加。如果一个替换元素的width不同于其固有宽度，那么height值也会成比例变化，除非height自己也显式设置为一个特定值。反过来也一样：如果设置了height，但width保持为auto，则width将随height的变化成比例调整。

既然谈到height，下面就来讨论块级正常流元素的垂直格式化。

垂直格式化

类似于水平格式化，块级元素的垂直格式化也有自己一系列有意思的行为。一个元素的默认高度由其内容决定。高度还会受内容宽度的影响；段落越窄，相应地就会越高，以便容纳其中所有的内联内容。

在CSS中，可以对任何块级元素设置显式高度。如果这样做，其结果取决于另外一些因素。假设指定高度大于显示内容所需的高度：

```
<p style="height: 10em;">
```

在这种情况下，多余的高度会产生一个视觉效果，就好像有额外的内边距一样。不过假设高度小于显示内容所需的高度：

```
<p style="height: 3em;">
```

如果是这样，浏览器会提供某种方法来查看所有内容，而不是增加元素框的高度。浏览器可能会向元素增加一个滚动条，如图7-13所示。

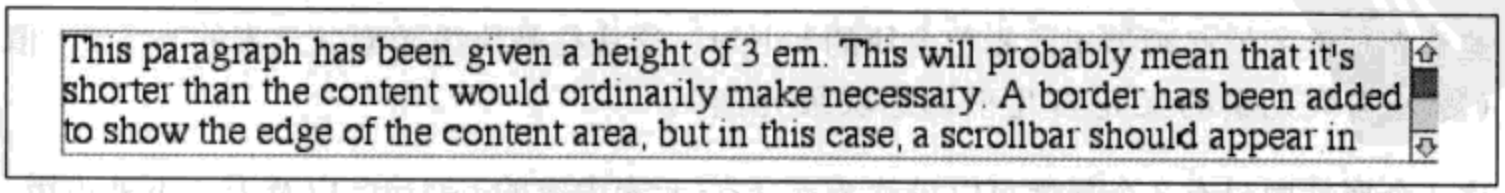

图7-13：高度与元素内容高度不匹配

如果元素内容的高度大于元素框的高度，用户代理的具体行为将取决于overflow属性的值（及用户代理对这个属性的支持程度）。这种情况将在第10章讨论。

在CSS1中，如果元素不是替换元素（如图像），用户代理会忽略除auto以外的所有其他height值。在CSS2和CSS2.1中，height值不能忽略，只有一种特定情况除外，即如果涉及到百分数值，也许会忽略height值，这种情况稍后讨论。

像width一样，height定义了内容区的高度，而不是可见元素框的高度。元素框上下的内边距、边距或外边距都会增加到height值。

垂直属性

与水平格式化的情况一样，垂直格式化也有7个相关的属性：margin-top、border-top、padding-top、height、padding-bottom、border-bottom和margin-bottom。这些属性如图7-14所示。

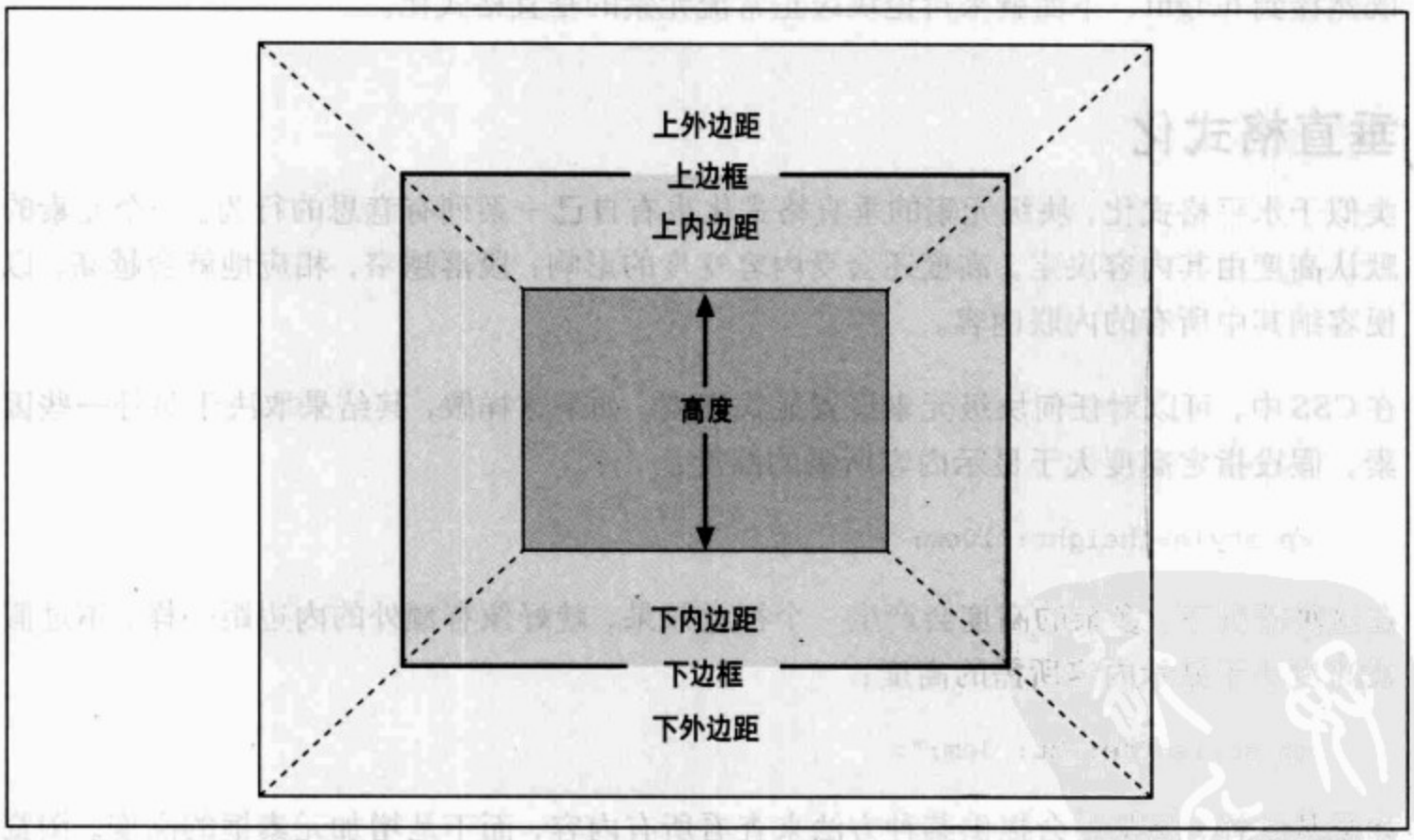

图7-14：垂直格式化的“7大属性”

这7个属性的值必须等于元素包含块的height。这往往是块级元素父元素的height值（因为块级元素的父元素几乎都是块级元素）。

这7个属性中只有3个属性可以设置为auto：元素内容的height以及上、下外边距。上、下内边距和边框必须设置为特定的值，或者默认为0（如果没有声明border-

style)。如果已经设置了border-style，边框的宽度会设置为值medium（这个值的定义并不明确）。图7-15展示了如何记住元素框中的哪些部分可以有auto值，而哪些部分不可以。

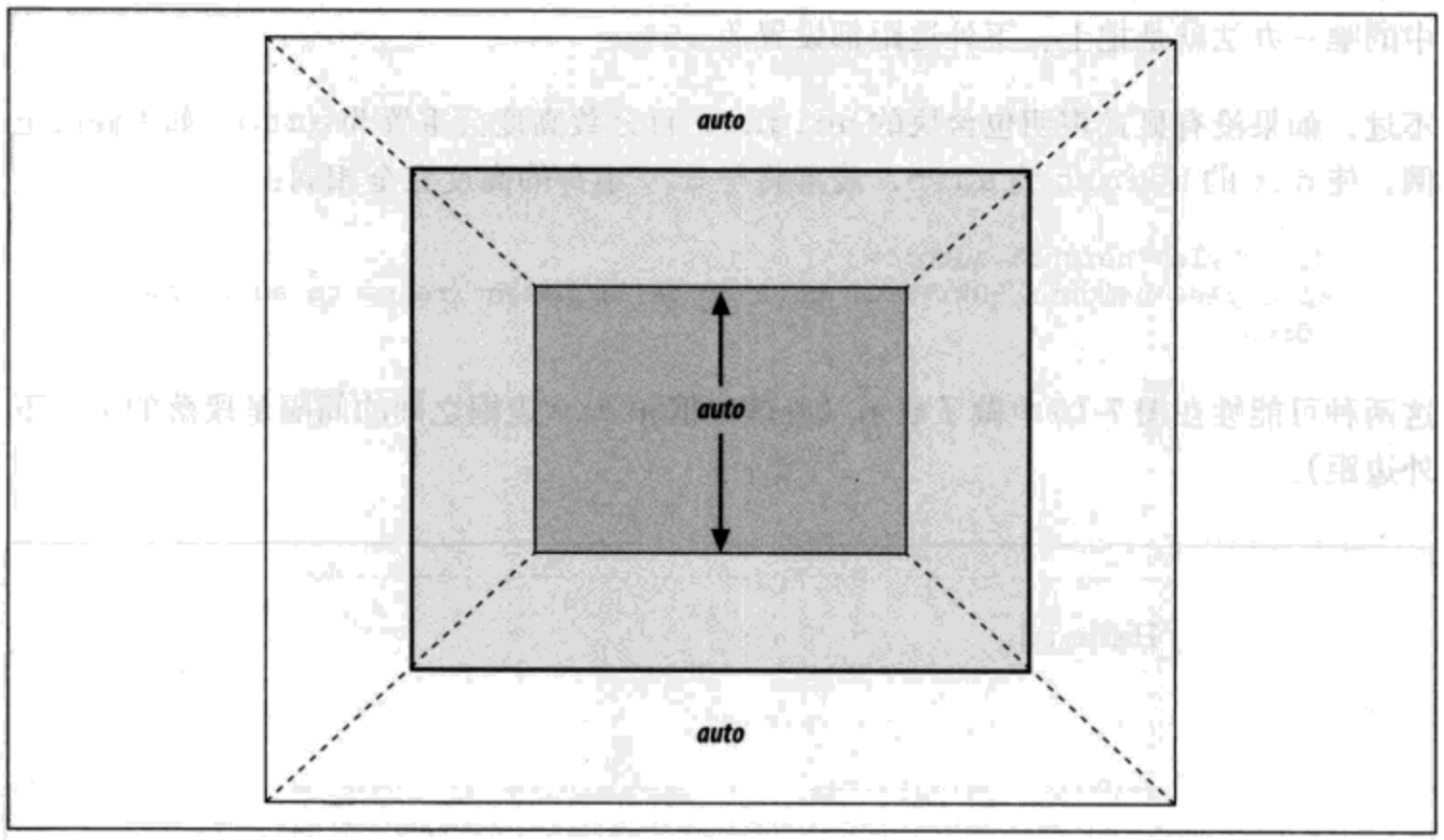

图7-15：可以设置为auto的垂直属性

有意思的是，如果正常流中一个块元素的margin-top或margin-bottom设置为auto，它会自动计算为0。遗憾的是，如果值为0，就不能很容易地将正常流元素在其包含块中垂直居中。这也说明，如果将一个元素的上、下外边距设置为auto，实际上它们都会重置为0，使元素框没有外边距。

注意：对于定位元素来说，上、下外边距为auto时，其处理有所不同。有关的更多详细内容参见第10章。

---

height必须设置为auto或者是某种类型的非负值。

百分数高度

前面已经了解了如何处理设置为长度值的高度，下面再花点时间介绍百分数高度。如果一个块级正常流元素的height设置为一个百分数，这个值则是包含块height的一个百分数。给定以下标记，相应的段落高度将是3em：

```
<div style="height: 6em;">
 <p style="height: 50%;">Half as tall</p>
</div>
```

由于将上、下外边距设置为auto时，实际上它们的高度将是0，因此，将元素垂直居中的唯一办法就是把上、下外边距都设置为25%。

不过，如果没有显式声明包含块的height，百分数高度会重置为auto。如果修改上例，使div的height为auto，段落将与div本身的高度完全相同：

```
<div style="height: auto;">
 <p style="height: 50%;">NOT half as tall; height reset to auto</p>
</div>
```

这两种可能性在图7-16中做了展示（段落边框和div边框之间的间隔是段落的上、下外边距）。

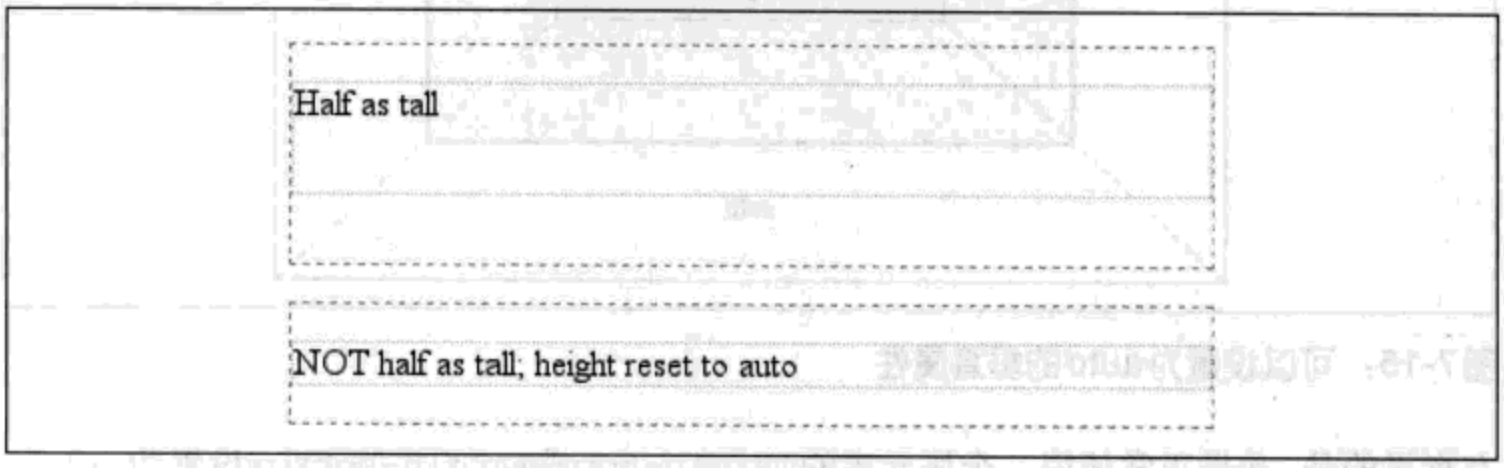

图7-16：不同情况下的百分数高度

auto高度

在最简单的情况下，如果块级正常流元素设置为height：auto，显示时其高度将恰好足以包含其内联内容（包括文本）的行盒。高度为auto时，会在段落上设置一个边框，并认为没有内边距，这样下边框刚好在文本最后一行的下面，上边框则刚好在文本第一行的上面。

如果块级正常流元素的高度设置为auto，而且只有块级子元素，其默认高度将是从最高块级子元素的外边框边界到最低块级子元素外边框边界之间的距离。因此，子元素的外边距会“超出”包含这些子元素的元素（这种行为将在下一节解释）。不过，如果块级元素有上内边距或下内边距，或者有上边框或下边框，其高度则是从其最高子元素的上外边距边界到其最低子元素的下外边距边界之间的距离：

```
<div style="height: auto; background: silver;">
<p style="margin-top: 2em; margin-bottom: 2em;">A paragraph!</p>
```

```
</div>
<div style="height: auto; border-top: 1px solid; border-bottom: 1px solid;
background: silver;">
<p style="margin-top: 2em; margin-bottom: 2em;">Another paragraph!</p>
</div>
```

这两种行为都在图 7-17 中得到了展示。

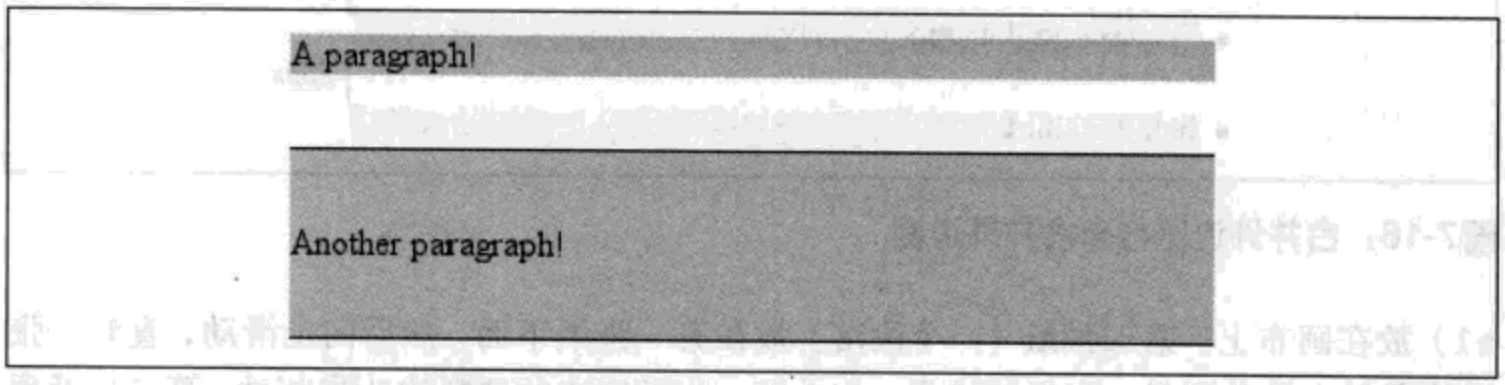

图7-17：有块级子元素且高度为auto

如果在上例中将边框改为内边距，对div高度的作用还是一样：同样会把段落的外边距包含在内。

合并垂直外边距

垂直格式化的另一个重要方面是垂直相邻外边距的合并。这种合并行为只应用于外边距。如果元素有内边距和边框，它们绝对不会合并。

来看一个无序列表，其列表项前后相邻，这是一个展示外边距合并的绝好例子。假设为一个包含 5 个列表项的列表作以下声明：

```
li {margin-top: 10px; margin-bottom: 15px;}
```

每个列表项有 10 像素的上外边距和 15 像素的下外边距。不过，在显示这个列表时，相邻列表项之间的距离是 15 像素，而不是 25 像素。之所以会这样，是因为相邻外边距会沿着竖轴合并。换句话说，两个外边距中较小的一个会被较大的一个合并。图 7-18 显示了合并外边距与未合并外边距之间的差别。

如果用户代理实现正确，将合并垂直相邻的外边距，如图 7-18 中第一个列表所示，这里各列表项之间有15像素的间隔。第二个列表显示了用户代理没有合并外边距时会出现什么情况，此时列表项之间有 25 像素的间隔。

如果你不喜欢用“合并”这个词，也可以用“重叠”。尽管外边距实际上并不会重叠，不过可以打个比方来看会发生什么情况。假设每个元素（例如一个段落）是一小张纸，上面写着元素的内容。每张纸外围有一些塑料边，这表示外边距。第一张纸（假设是一个

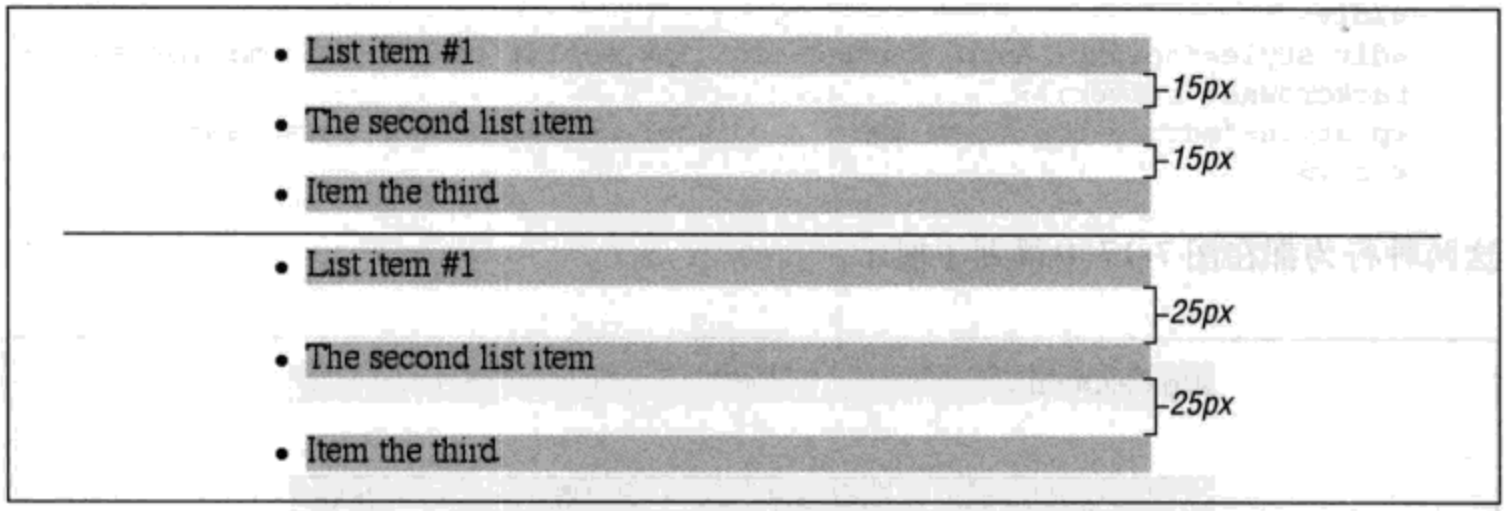

图7-18：合并外边距与未合并外边距

h1）放在画布上。第二张纸（一个段落）放在第一张纸下面，然后向上滑动，直到一张纸的塑料边界碰到另一张纸的边界。如果第一张纸底边的塑料边为半英寸，第二张纸顶边的塑料边为1/3英寸，等这两张纸滑到一起时，第一张纸的塑料边界会碰到第二张纸的上边界。现在这两张纸都放在画布上，它们的塑料边相重叠。

如果相邻有多个外边距，也会出现合并，如列表的最后。对前面的例子做些补充，假设应用以下规则：

```
ul {margin-bottom: 15px;}
li {margin-top: 10px; margin-bottom: 20px;}
h1 {margin-top: 28px;}
```

列表中最后一项的下外边距为20像素，ul的下外边距为15像素，后面的h1的上外边距为28像素。所以一旦合并这些外边距，li的结尾到h1的开始之间有28像素的距离，如图7-19所示。

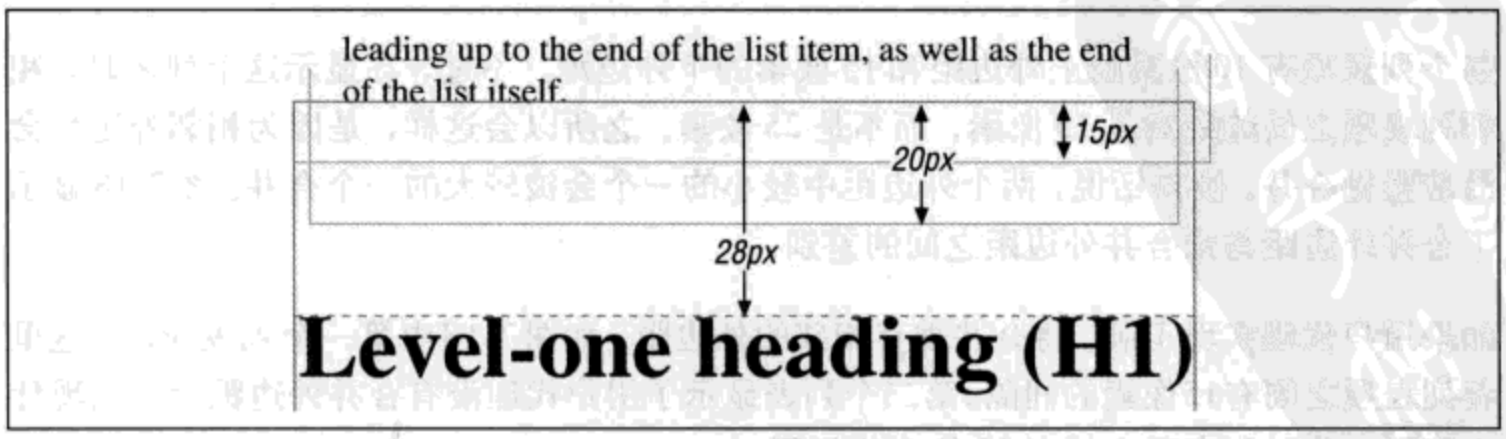

图7-19：合并外边距详解

现在回顾上一节的例子，在包含块上设置边框或内边距时，会使其子元素的外边距包含在包含块内。为了解这种行为，可以在前面的例子中为ul元素增加一个边框：

```
ul {margin-bottom: 15px; border: 1px solid;}
li {margin-top: 10px; margin-bottom: 20px;}
h1 {margin-top: 28px;}
```

做了这个改变后，li元素的下外边距现在放在其父元素内部（ul）。因此，这里只会在ul和h1之间发生外边距合并，如图7-20所示。

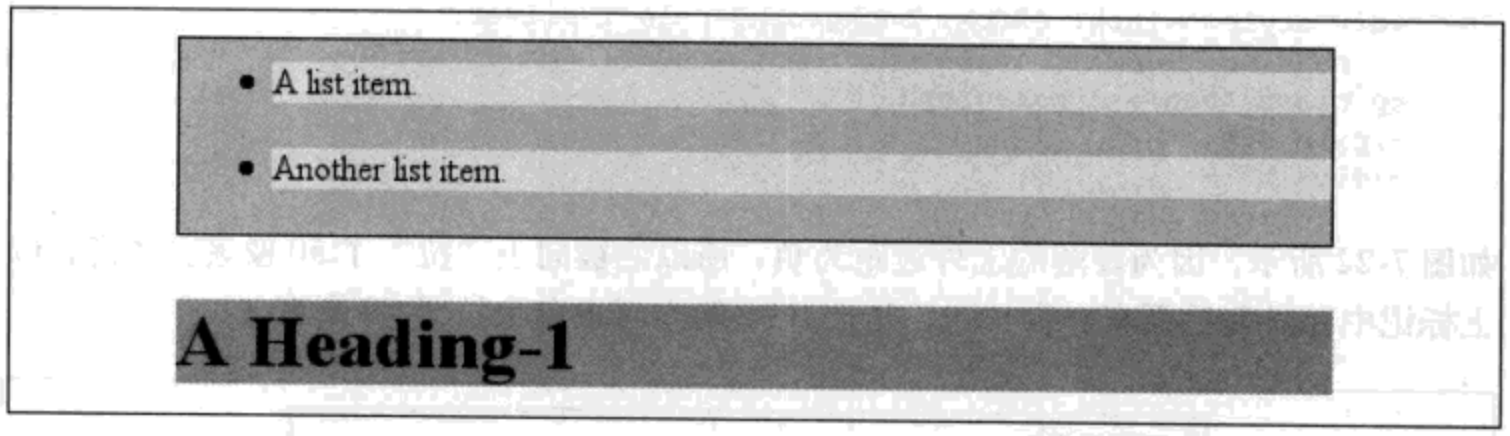

图7-20：增加边框后的合并

负外边距

负外边距确实对垂直格式化有影响，而且它们会影响外边距如何合并。如果垂直外边距都设置为负值，浏览器会取两个外边距绝对值的最大值。如果一个正外边距与一个负外边距合并，会从正外边距减去这个负外边距的绝对值。换句话说，负值要增加到正值，所得到的就是元素间的距离。图7-21给出了两个具体的例子。

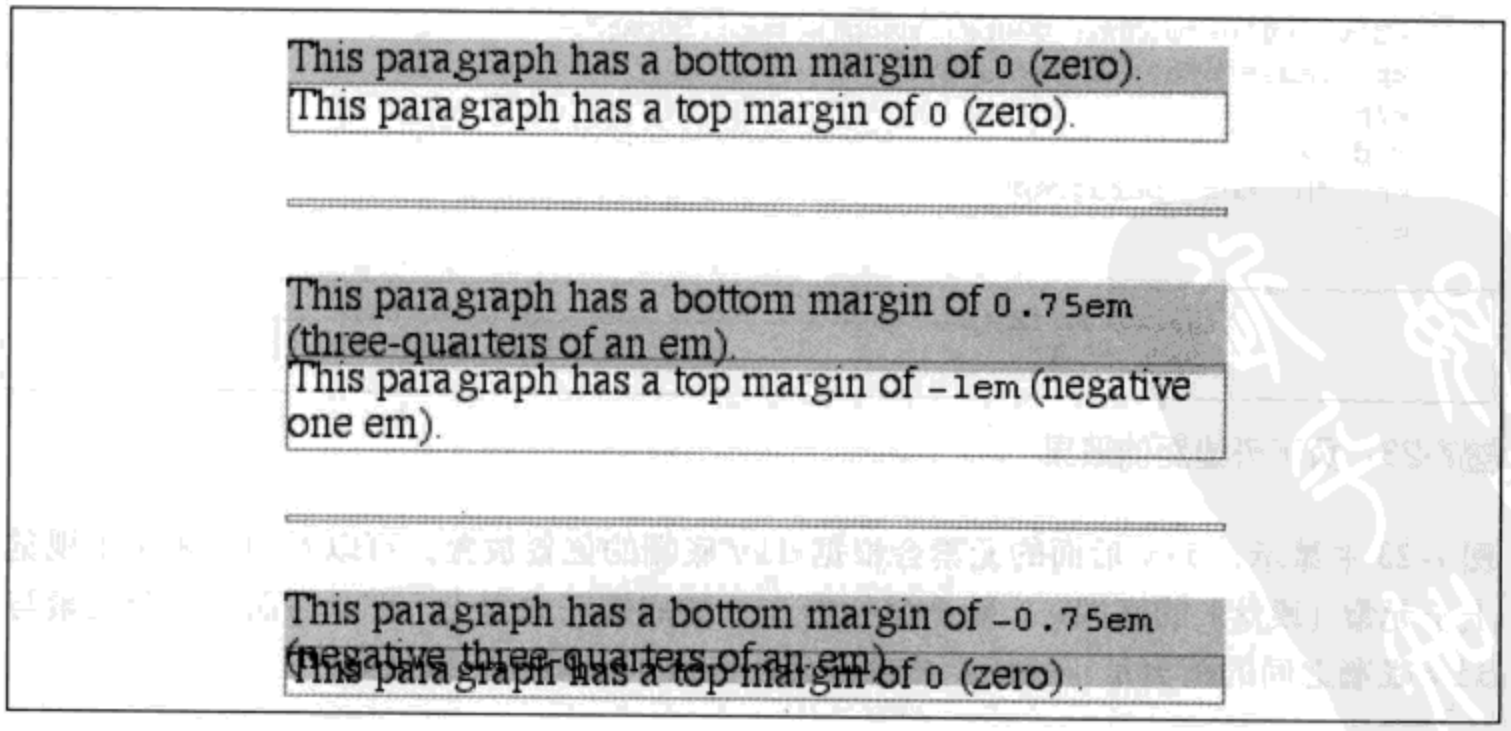

图7-21：负垂直外边距的例子

注意，上、下外边距为负时有一种“拉近”效果。实际上，这与负水平外边距使元素超出其父元素没有什么区别。请考虑：

```
p.neg {margin-top: -50px; margin-right: 10px;
  margin-left: 10px; margin-bottom: 0;
  border: 3px solid gray;}

<div style="width: 420px; background-color: silver;
   padding: 10px; margin-top: 50px; border: 1px solid;">
<p class="neg"> A paragraph.
</p> A div.
</div>
```

如图7-22所示，因为段落的上外边距为负，所以它被向上“拉”了50像素。注意，以上标记中在段落后面有一个div，这个div的内容也向上拉了50像素。

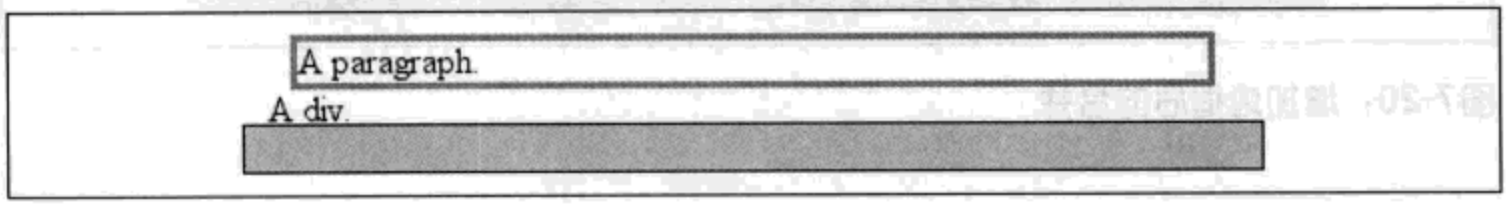

图7-22：负上外边距的效果

负的下外边距会使段落看上去被向下拉。将以下标记与图7-23所示对照：

```
p.neg {margin-bottom: -50px; margin-right: 10px;
  margin-left: 10px; margin-top: 0;
  border: 3px solid gray;}

<div style="width: 420px; margin-top: 50px;">
<p class="neg"> A paragraph.
</p>
</div>
<p> The next paragraph.
</p>
```

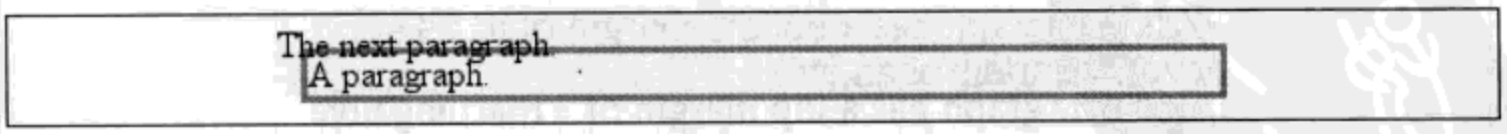

图7-23：负下外边距的效果

图7-23中显示，div后面的元素会根据div底端的位置放置。可以看到，相对于视觉上子元素（段落）的底端，实际上父元素div的底端在它之上。div后面下一个元素与div底端之间的距离是正确的。根据你使用的规则，确实应该如此。

下面来考虑一个例子，其中有一个列表项、一个无序列表以及一个标题，它们的外边距都合并。在这里，为无序列表和标题指定了负外边距：

```
li {margin-bottom: 20px;}
ul {margin-bottom: -15px;}
h1 {margin-top: -18px;}
```

两个负外边距中较大的一个（-18px）增加到了最大的正外边距上（20px），这就得到了20px − 18px = 2px。因此，列表项内容底端与h1内容顶端之间只有2个像素的距离，如图7-24所示。

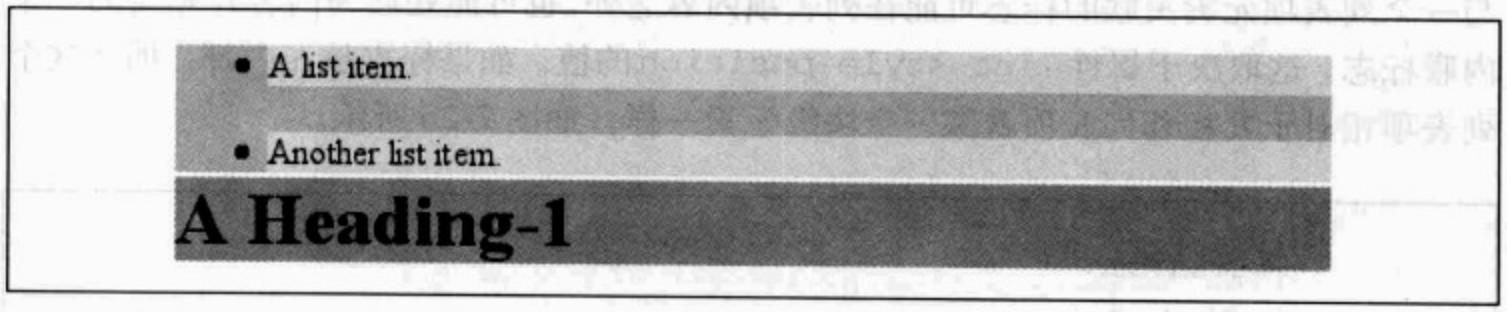

图7-24：正负外边距合并详解

还有一种行为没有讨论，如果由于负外边距而导致元素彼此重叠，很难区分哪些元素在上面。你可能还会注意到，本节中所有例子都没有使用背景色。如果确实使用了背景色，其内容可能会被后面元素的背景色所覆盖。这是一种可以预见的行为，因为浏览器总会按从前到后的顺序显示元素，所以文档中后出现的正常流元素可能会覆盖较早出现的元素（如果这两个元素重叠）。

列表项

列表项有自身的一些特殊规则。这些列表项前面通常有一个标志，如一个圆点或一个数字。这个标志实际上并不是列表项内容区的一部分，所以如图7-25所示的效果很常见。

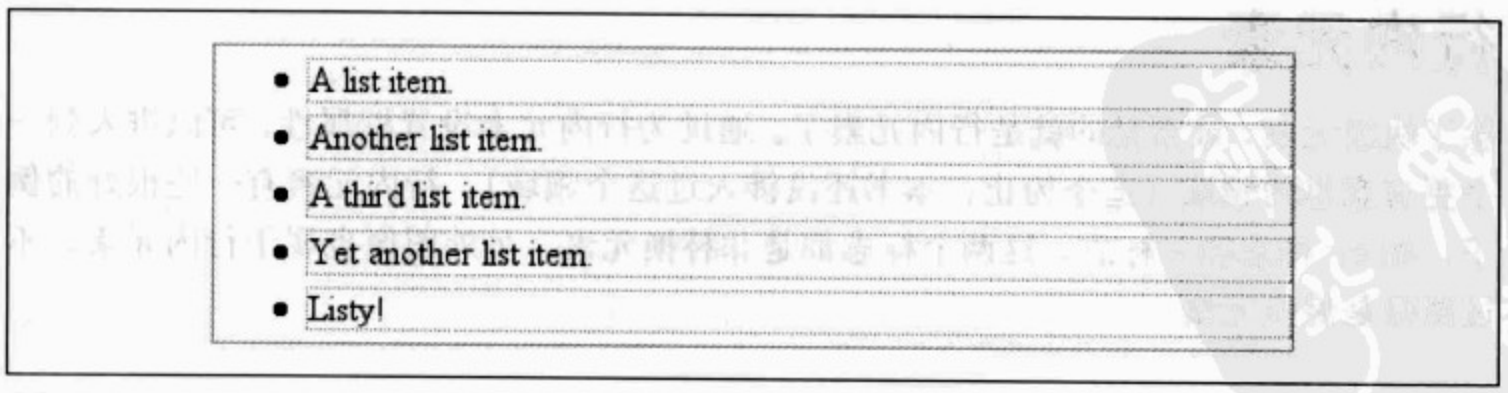

图7-25：列表项的内容

CSS1对这些标志相对于文档布局的放置和效果涉及很少。但CSS2引入了专门为解决这个问题而设计的一些属性，如marker-offset。不过，光有想法但缺少实现，这就使得这个属性在CSS2.1中又被去掉了，很有可能CSS的将来版本还会引入另外一种方法来

定义内容和标志之间的距离。因此，标志的放置不在创作人员的控制范围内（至少在写作本书时是这样）。

注意： 第12章将更详细地研究列表及如何设置列表的样式。

与一个列表项元素关联的标志可能在列表项内容之外，也可能处理为内容开始处的一个内联标志，这取决于属性`list-style-position`的值。如果标志放在内部，那么这个列表项相对于其相邻列表项就像一个块级元素一样，如图7-26所示。

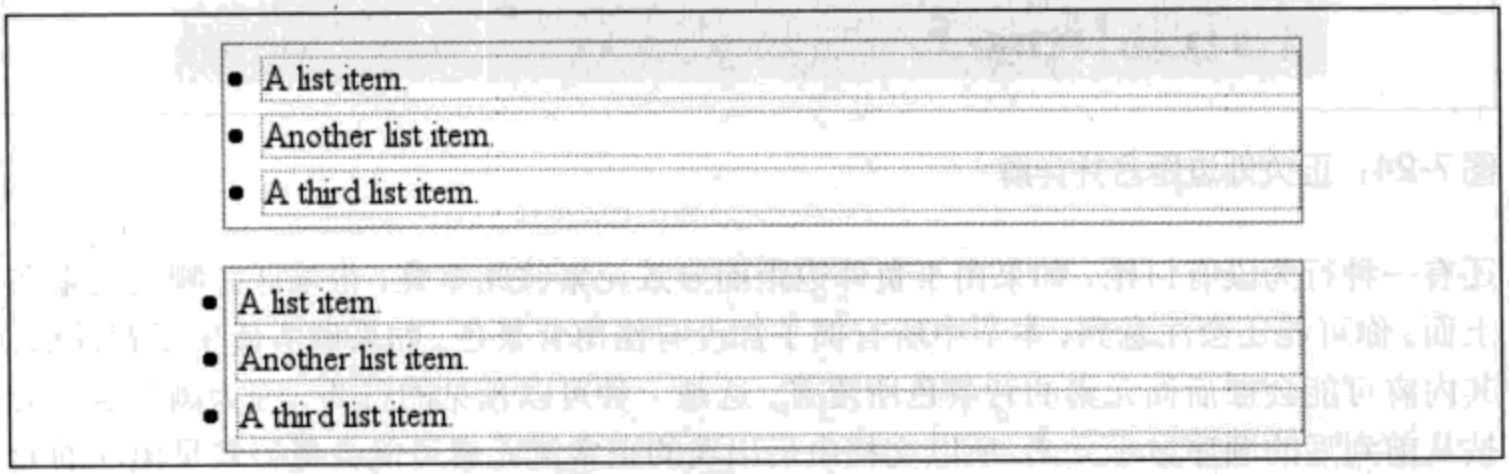

图7-26：标志放在列表内部和外部

如果标志在内容之外，它会放在与内容左边界有一定距离的位置上（对于从左向右读的语言）。不论如何改变列表的样式，标志与内容边界的距离都不变。有时，标志可能会放在列表元素本身之外，如图7-26所示。

行内元素

除了块级元素，最常见的就是行内元素了。通过为行内元素设置框属性，可以进入到一个更有意思的领域（迄今为止，本书还没涉入过这个领域）。行内元素有一些很好的例子，如em标志和a标志，这两个标志都是非替换元素，另外图像也属于行内元素，不过图像是替换元素。

警告： 本节介绍的所有行为都不适用于表元素。CSS2对于表和表内容的处理引入了一些新的属性和行为，表元素的表现与块级元素或行内元素大相径庭。表样式将在第11章讨论。

非替换元素和替换元素在内联内容方面的处理稍有不同，讨论行内元素的构造时我们将分别进行讨论。

行布局

首先，需要理解内联内容如何布局。对于行内元素来说，这没有块级元素那么简单和直接，块级元素只是生成框，通常不允许其他内容与这些框并存。下面做个对照，来看一个块级元素的内部（如一个段落）。你可能会问："所有文本行是怎么放在这里的？是谁在控制它们的摆放？我能做些什么？"

为了理解如何生成行，首先来考虑这样一种情况，一个元素包含一个很长的文本行，如图7-27所示。注意，这里将整行包围在一个span元素中，从而为这一行加了一个边框，然后为之指定边框样式：

```
span {border: 1px dashed black;}
```

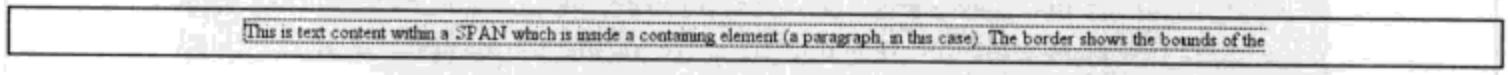

图7-27：单行行内元素

图7-27显示了行内元素包含在一个块级元素中的最简单的情况。就其本身来说，这与包含两个词的段落没有什么区别。唯一的区别是，图7-27中有几十个词，而大多数段落不会包含诸如span之类的显式行内元素。

如果不是这种最简单的状况，来看我们更熟悉的情况，你所要做的只是确定元素应当有多宽，然后断行，使断开的各部分能适应元素的宽度刚好放下。这样就得到了如图7-28所示的结果。

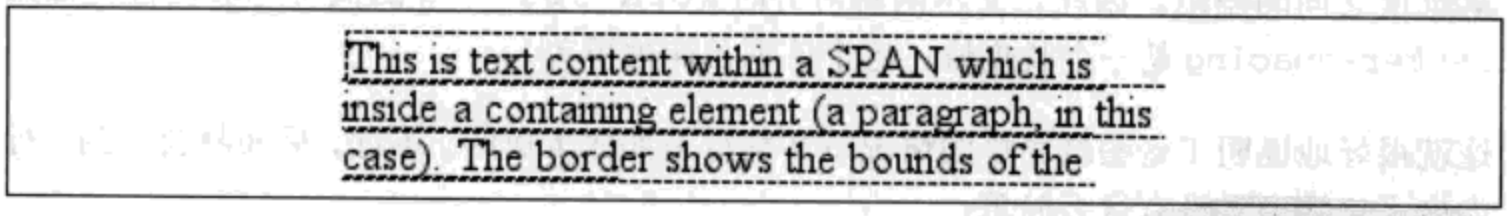

图7-28：多行行内元素

什么都没变。你所做的只是将一行分成多个部分，然后把这些部分一个挨一个地堆放。

在图7-28中，每个文本行的边框刚好与各行的顶端和底端吻合。这是因为没有为行内文本设置内边距或外边距。注意，实际上边框彼此稍有些重叠；例如，第一行的下边框就在下一行上边框的下面。这是因为边框实际上画在各行之外的下一个像素上（假设你使用的是显示器）。由于各行紧挨着，所以其边框会重叠，如图7-28所示。

如果改变span样式，使之有一个背景色，这些行的具体摆放就很清楚了。下面来看图7-29，其中包含4个段落，每个段落有不同的text-align值，而且每个段落的文本行都有背景。

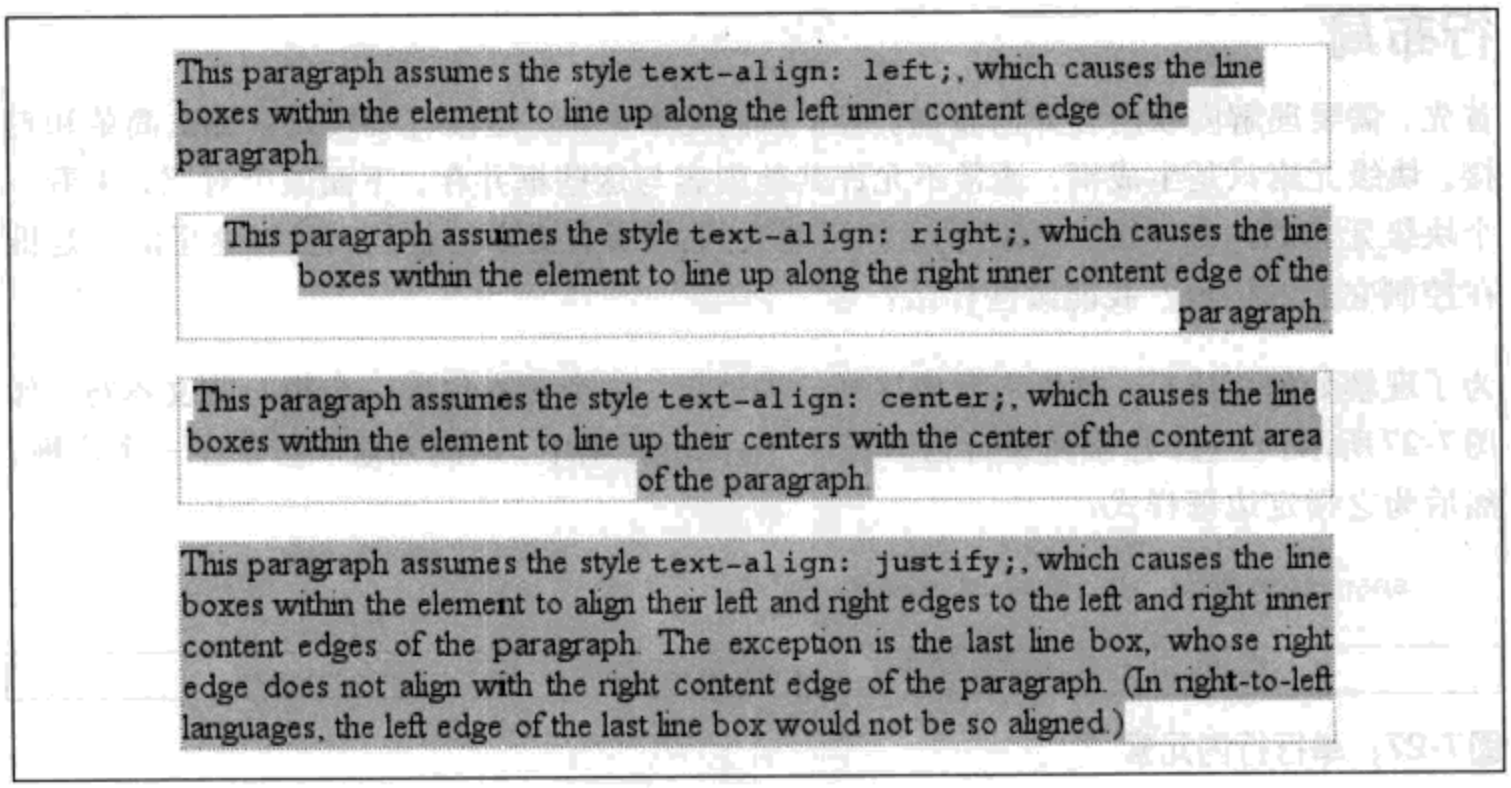

图7-29：显示有不同对齐方式的文本行

可以看到，并不是每一行都达到其父段落内容区的边界，父段落内容区用一个灰色虚线边框指示。对于左对齐的段落，行都压到段落内容区的左边对齐，各行在换行处结束。右对齐的段落则恰好相反。居中段落中，文本行的中心与段落的中心对齐。在最后一种情况下，即text-align的值为justify，各行必须与段落的内容区宽度相等，所以行边界要达到段落的内容边界。要调整各行中字母和字之间的间隔来弥补行实际长度与段落宽度之间的差距。因此，文本两端对齐时word-spacing的值可能被覆盖（如果letter-spacing是一个长度值，这个值则不能被覆盖）。

这就很好地说明了这些最简单的情况下文本行是如何生成的。不过，后面将会看到，行内格式化模型远没有这么简单。

基本术语和概念

进一步介绍后面的内容之前，先来回顾行内布局的一些基本术语，这对于学习后面的小节非常重要：

匿名文本

匿名文本（anonymous text）是指所有未包含在行内元素中的字符串。因此，在标记 I'm so happy! 中，序列“I'm”和“happy!”都是匿名文本。注意，空格也是匿名文本的一部分，因为空格与其他字符一样都是正常的字符。

em 框

em 框在字体中定义，也称为字符框（character box）。实际的字形可能比其 em 框更高或更矮，见第 5 章的讨论。在 CSS 中，font-size 的值确定了各个 em 框的高度。

内容区

在非替换元素中，内容区可能有两种，CSS2.1 规范允许用户代理选择其中任意一种。内容区可以是元素中各字符的 em 框串在一起构成的框，也可以是由元素中字符字形描述的框。本书中，为简单起见采用了前一种定义，即 em 框定义。在替换元素中，内容区就是元素的固有高度再加上可能有的外边距、边框或内边距。

行间距

行间距（leading）是 font-size 值和 line-height 值之差。这个差实际上要分为两半，分别应用到内容区的顶部和底部。毫不奇怪，为内容区增加的这两部分分别称为半间距（half-leading）。行间距只应用于非替换元素。

行内框

这个框通过向内容区增加行间距来描述。对于非替换元素，元素行内框的高度刚好等于 line-height 的值。对于替换元素，元素行内框的高度则恰好等于内容区的高度，因为行间距不应用到替换元素。

行框

这是包含该行中出现的行内框的最高点和最低点的最小框。换句话说，行框的上边界要位于最高行内框的上边界，而行框的底边要放在最低行内框的下边界。

根据前面介绍的术语和定义，CSS 还提供了一组行为和有用的概念：

- 内容区类似于一个块级元素的内容框。
- 行内元素的背景应用于内容区及所有内边距。
- 行内元素的边框要包围内容区及所有内边距和边框。
- 非替换元素的内边距、边框和外边距对行内元素或其生成的框没有垂直效果；也就是说，它们不会影响元素行内框的高度（也不会影响包含该元素的行框的高度）。
- 替换元素的外边距和边框确实会影响该元素行内框的高度，相应地，也可能影响包含该元素的行框的高度。

还有一点需要注意：行内框在行中根据其 vertical-align 属性值垂直对齐。这一点在第 6 章已经谈到，本章将更深入地解释。

在继续介绍之前，先来看如何逐步构造一个行框，可以通过这个过程来了解一行的各部分如何共同确定其高度：

1. 按以下步骤确定行中各元素行内框的高度：
   a. 得到各行内非替换元素及不属于后代行内元素的所有文本的font-size值和line-height值，再将line-height减去font-size，这就得到了框的行间距。这个行间距除以2，将其一半分别应用到em框的顶部和底部。
   b. 得到各替换元素的height、margin-top、margin-bottom、padding-top、padding-bottom、border-top-width和border-bottom-width值，把它们加在一起。
2. 对于各内容区，确定它在整行基线的上方和下方分别超出多少。这个任务并不容易：你必须知道各元素及匿名文本各部分的基线的位置，还要知道该行本身基线的位置；然后把它们对齐。另外，对于替换元素，要将其底边放在整行的基线上。
3. 对于指定了vertical-align值的元素，确定其垂直偏移量。由此可知该元素的行内框要向上或向下移动多远，并改变元素在基线上方或下方超出的距离。
4. 既然已经知道了所有行内框会放在哪里，再来计算最后的行框高度。为此，只需将基线与最高行内框顶端之间的距离加上基线与最低行内框底端之间的距离。

下面详细考虑整个过程，这对于聪明地设置内联内容的样式很关键。

行内格式化

第6章曾讨论过，所有元素都有一个line-height。这个值会显著地影响行内元素如何显示，所以要特别注意。

首先来看如何确定一行的高度。行的高度（或行框的高度）由其组成元素和其他内容（如文本）的高度确定。有一点很重要，line-height实际上只影响行内元素和其他行内内容，而不影响块级元素，至少不会直接影响块级元素。也可以为一个块级元素设置line-height值，但是这个值只是应用到块级元素的内联内容时才会有视觉影响。例如，考虑以下空段落：

```
<p style="line-height: 0.25em;"></p>
```

由于没有内容，这个段落没有任何显示，你什么也看不到。这个段落的line-height可能是某个值（不论是0.25em还是25in），但是如果没有内容，line-height是多少对于创建行框来说都没有任何区别。

当然可以为一个块级元素设置line-height值，并将这个值应用到块中的所有内容，而不论内容是否包含在行内元素中。从某种程度上讲，块级元素中包含的各文本行本身都是行内元素，而不论是否真正用行内元素的标记包围起来。只要你愿意，可以像下面这样写一个虚构的标记序列：

```
<p>
<line>This is a paragraph with a number of</line>
<line>lines of text which make up the</line>
<line>contents.</line>
</p>
```

尽管line标记并不真的存在，但是段落表现得就像有这些标记一样，每个文本行从段落继承了样式。因此，只需为块级元素创建line-height规则，而不必显式地为其所有行内元素（也许只是虚构的行内元素）声明line-height。

虚构的line元素确实可以说明对块级元素设置line-height会有怎样的行为。根据CSS规范，在块级元素上声明line-height会为该块级元素的内容设置一个最小行框高度。因此，声明p.spacious {line-height: 24pt;}意味着每个行框的最小高度为24点。从理论上讲，只有行内元素的内容才会继承这个行高。大多数文本并未包含在行内元素中。因此，如果假装各行包含在虚构的line元素中，这个模型就能很好地工作。

行内非替换元素

在前面的格式化知识基础上，来讨论如果行中只包含非替换元素（或匿名文本）将如何构造。了解这一点后，你就能更好地理解行内布局中非替换元素和替换元素之间的区别。

建立框

首先，对于行内非替换元素或匿名文本某一部分，font-size值确定了内容区的高度。如果一个行内元素font-size为15px，则内容区的高度为15像素，因为元素中所有em框的高度都是15像素，如图7-30所示。

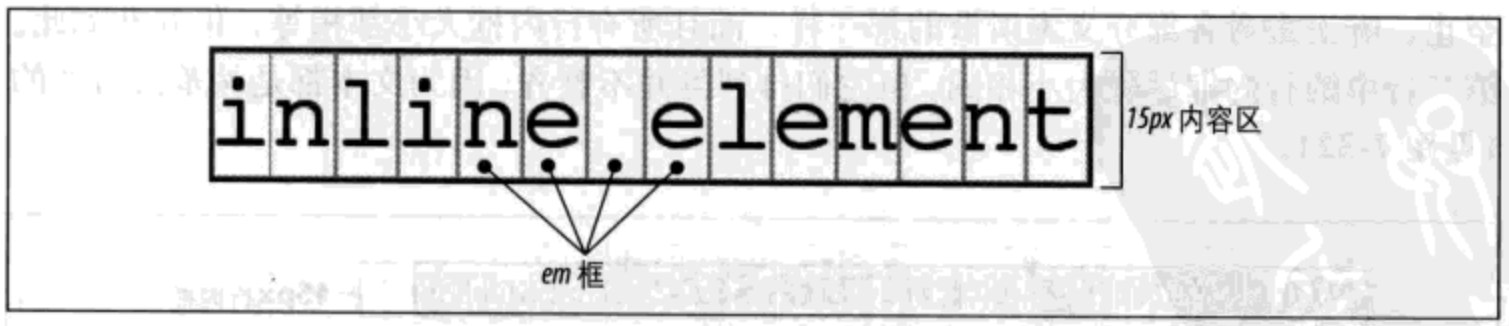

图7-30：em框确定内容区高度

下面再来考虑元素的line-height值，以及它与font-size值之差。如果一个行内非替换元素的font-size为15px，line-height为21px，则相差6像素。用户代理将这6像素一分为二，将其一半分别应用到内容区的顶部和底部，这就得到了行内框。这个过程如图7-31所示。

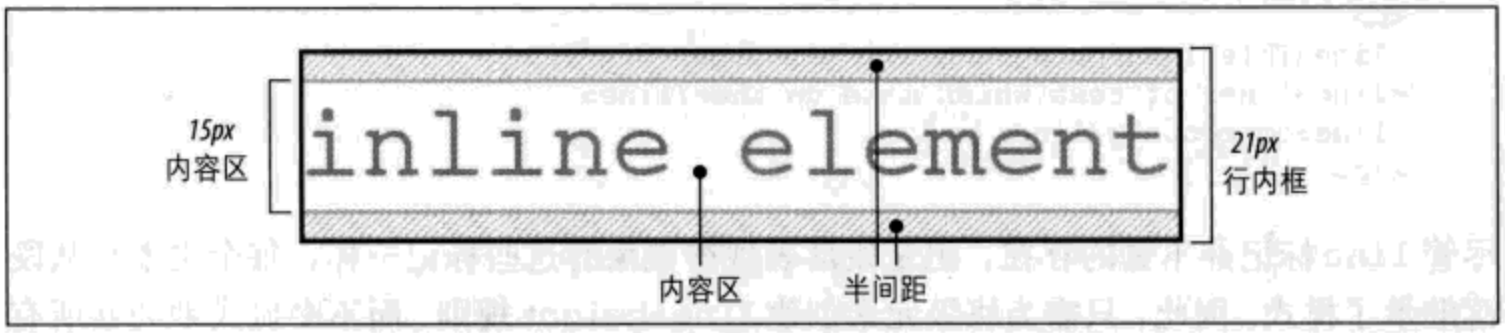

图7-31：内容区加上行间距就等于行内框

假设有以下标记：

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>, plus other text<br>
which is <strong style="font-size: 24px;">strongly emphasized</strong>
and which is<br>
larger than the surrounding text.
</p>
```

在这个例子中，大多数文本的font-size都是12px，只有一个行内非替换元素中的文本大小是24px。不过，所有文本的line-height都是12px，因为line-height是一个继承属性。因此，strong元素的line-height也是12px。

所以，对于font-size和line-height都是12px的各部分文本，内容高度没有改变（因为12px和12px之差为0），因此，行内框的高度为12像素。不过，对于strong文本，line-height和font-size之差是-12px。将其除2来确定半间距（-6px），再把这个半间距分别增加到内容区的顶部和底部，就得到了行内框。由于这里增加的都是负数，所以最后行内框高度为12像素。12像素高的行内框在元素内容区（24像素高）中垂直居中，所以行内框实际上小于内容区。

至止，听上去对各部分文本所做的都一样，而且所有行内框大小都相等，但并非如此。第二行中的行内框尽管大小相同，但它们排列得并不整齐，因为文本都是按基线对齐的（见图7-32）。

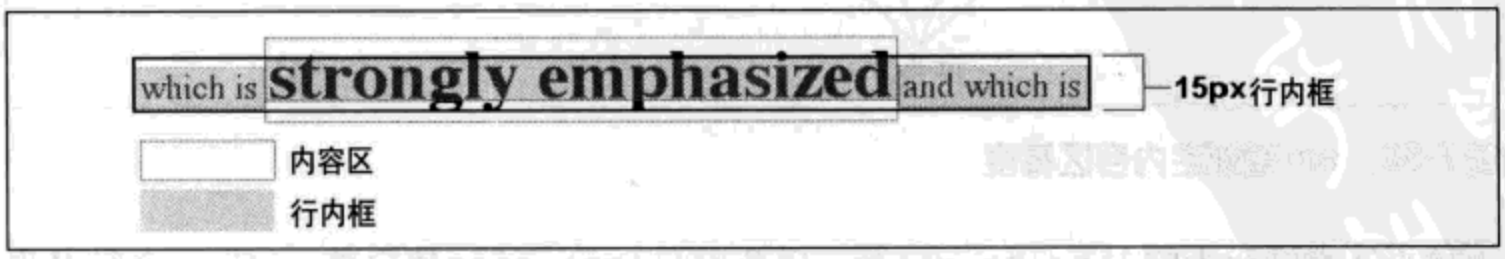

图7-32：行中的行内框

由于行内框确定了整个行框的高度，其相互位置很重要。行框定义为行中最高行内框的顶端到最低行内框底端之间的距离，而且各行框的顶端挨着上一行行框的底端。根据图7-32所示的结果，段落将如图7-33所示。

图7-33：段落中的行框

注意： 在图7-33中可以看到，中间一行比另外两行要高，不过还是不够大，不能把所有文本都包含在内。匿名文本的行内框确定了行框的底端，strong元素行内框的顶端则设置了行框的顶端。由于行内框的顶端在元素内容区内部，所以元素的内容落在了行框的外面，实际上与其他行框发生了重叠。其结果是，文本行看上去很不规则。本章后面还将介绍一些方法来处理这种行为，另外会介绍一些能得到一致基线间隔的方法。

垂直对齐

如果改变行内框的垂直对齐，会应用同样的高度确定原则。假设为strong元素指定垂直对齐为4px：

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>, plus other text<br>
that is <strong style="font-size: 24px; vertical-align: 4px;">strongly
emphasized</strong> and that is<br>
larger than the surrounding text.
</p>
```

这个小小的改动会把元素上升4像素，这会同时提升其内容区和行内框。由于strong元素的行内框顶端已经是行中的最高点，对垂直对齐的这个修改会把整个行框的顶端也向上移4像素，如图7-34所示。

图7-34：垂直对齐影响行框高度

下面来考虑另一种情况。strong文本所在行上还有一个行内元素，其对齐方式未设置为基线对齐：

```
<p style="font-size: 12px; line-height: 12px;">
this is text, <em>some of which is emphasized</em>, plus other text<br>
that is <strong style="font-size: 24px;">strong</strong>
and <span style="vertical-align: top;">tall</span> and that is<br>
larger than the surrounding text.
</p>
```

现在的结果与前面的例子相同，即中间的行框比另外两行要高。不过，注意图7-35中tall文本是如何对齐的。

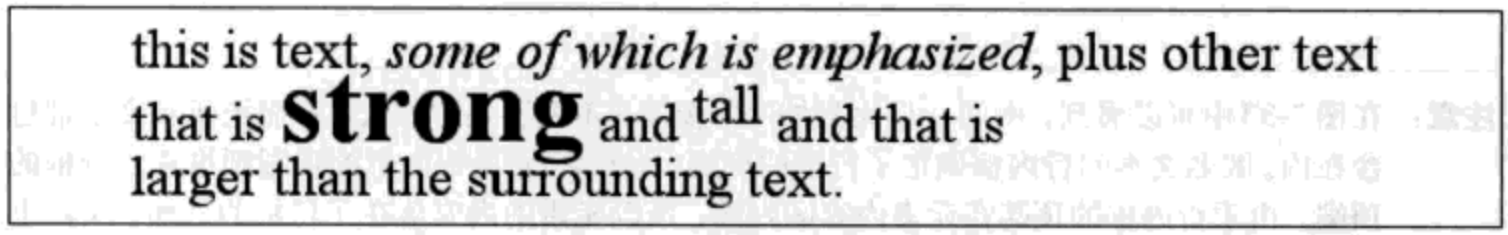

图7-35：行内元素与行框对齐

在这种情况下，tall文本行内框的顶端与行框的顶端对齐。由于tall文本的font-size和line-height值相等，所以其内容高度与行内框相同。不过，再考虑以下情况：

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>, plus other text<br>
that is <strong style="font-size: 24px;">strong</strong>
and <span style="vertical-align: top; line-height: 4px;">tall</span>
and that is<br>
larger than the surrounding text.
</p>
```

由于tall文本的line-height小于其font-size，该元素的行内框比其内容区要小。这会改变文本本身的放置，因为其行内框的顶端必须与该行行框的顶端对齐。所以，可以得到图7-36所示的结果。

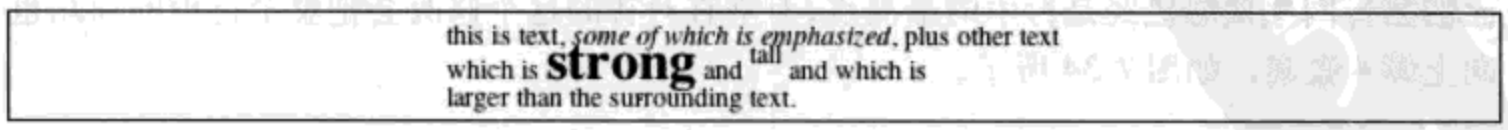

图7-36：文本再一次超出行框

另一方面，可以将tall文本设置为line-height实际上大于其font-size。例如：

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>, plus other text<br>
that is <strong style="font-size: 24px;">strong</strong>
and <span style="vertical-align: top; line-height: 18px;">tall</span>
```

```
and that is<br>
larger than the surrounding text.
</p>
```

因为为tall文本指定了line-height等于18px，line-height与font-size之差是6像素。半间距为3像素，将半间距增加到内容区，得到行内框为18像素高。这个行内框的顶端与行框的顶端对齐。类似地，如果vertical-align值设置为bottom，则把行内元素行内框的底端与行框的底端对齐。

采用本章所用的术语，vertical-align各个关键字值的效果描述如下：

top
: 将元素行内框的顶端与包含该元素的行框的顶端对齐。

bottom
: 将元素行内框的底端与包含该元素的行框的底端对齐。

text-top
: 将元素行内框的顶端与父元素内容区的顶端对齐。

text-bottom
: 将元素行内框的底端与父元素内容区的底端对齐。

middle
: 将元素行内框的垂直中点与父元素基线上0.5ex处的一点对齐。

super
: 将元素的内容区和行内框上移。上移的距离未指定，可能因用户代理的不同而不同。

sub
: 与super相同，只不过元素会下移而不是上移。

<percentage>
: 将元素上移或下移一定距离，这个距离由相对于元素line-height值指定的一个百分数确定。

这些值在第6章做过详细解释。

管理line-height

在前几节中我们已经了解到，改变一个行内元素的line-height可能导致文本行相互重叠。不过，在所有情况下，这种修改都是针对单个元素的。所以，如何以一种更一般的方式影响元素的line-height而避免内容重叠呢？

一种办法是对 font-size 有改变的元素结合使用 em 单位。例如：

```
p {font-size: 14px; line-height: 1em;}
big {font-size: 250%; line-height: 1em;}

<p> Not only does this paragraph have "normal" text, but it also<br>
contains a line in which <big>some big text </big> is found.<br>
This large text helps illustrate our point.
</p>
```

通过为 big 元素设置一个 line-height，就提高了行框的总高度，从而提供足够的空间来显示这个 big 元素，而不会与任何其他文本重叠，也不会改变段落中所有行的 line-height。这里使用了值 1em，所以 big 元素的 line-height 将设置为与 big 的 font-size 大小相等。要记住，line-height 相对于元素本身的 font-size 设置，而不是相对于父元素设置。其结果如图 7-37 所示。

图7-37：为行内元素指定line-height属性

注意，以下样式也会生成如图 7-37 所示的结果：

```
p {font-size: 14px; line-height: 1;}
big {font-size: 250%;}
```

除非 line-height 值作为缩放因子被继承，否则 p 和 big 元素的 line-height 值都为 1。因此，行内框的高度与内容区的高度一致，如图 7-37 所示。

一定要真正理解前几节的介绍，因为如果再增加边框，问题会更复杂。假设在所有超链接上加一个 5 像素的边框：

```
a:link {border: 5px solid blue;}
```

倘若没有设置一个足够大的 line-height 来容纳这个边框，就有覆盖其他行的危险。可以使用 line-height 增加未访问链接行内框的大小，就像前例中对 big 元素的做法一样；在这里，只需让 line-height 值比这些链接的 font-size 值大 10 像素。不过，如果你不知道字体大小是多少像素，这可能很困难。

另一种解决方法是增加段落的 line-height。这将会影响整个元素中的每一行，而不只是出现加边框超链接的那一行：

```
p {font-size: 14px; line-height: 24px;}
a:link {border: 5px solid blue;}
```

由于各行上下都增加了额外的空间，超链接外的边框不会覆盖其他行，如图7-38所示。

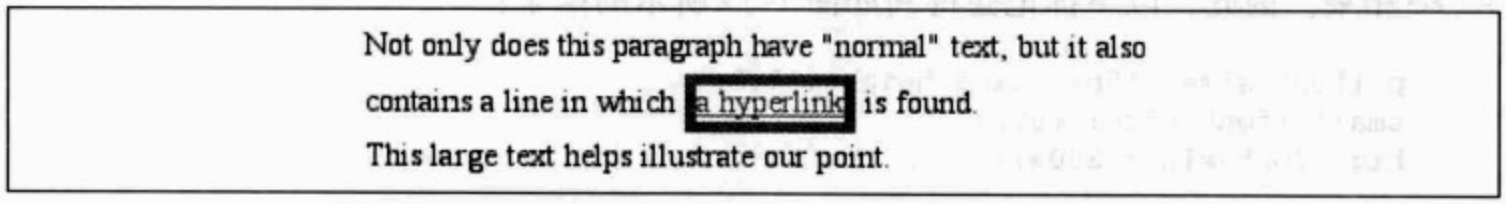

图7-38：增加line-height以为行内元素边框留出空间

当然，这种方法在这里是可行的，因为所有文本的大小都相同。如果行中还有另外一些元素改变了行框的高度，边框情况也可能发生变化。考虑以下规则：

```
p {font-size: 14px; line-height: 24px;}
a:link {border: 5px solid blue;}
big {font-size: 150%; line-height: 1.5em;}
```

根据这些规则，段落中big元素行内框的高度将是31.5像素（14×1.5×1.5），这也是行框的高度。为了保证基线间隔一致，必须让p元素的line-height等于或大于32px。

基线与行高

各行框的具体高度取决于其组成元素相互之间如何对齐。这种对齐往往很大程度上依赖于基线落在各元素（或匿名文本各部分）中的哪个位置，因为这个位置确定了其行内框如何摆放。基线在各em框中的位置对于不同的字体是不同的。这个信息内置在字体文件中，除非直接编辑字体文件，否则无法修改。

因此，要得到一致的基线间隔，这更像是一门艺术而不只是一门科学。如果使用一种单位（如em）来声明所有字体大小和行高，就很有可能得到一致的基线间隔。不过，如果混合使用了不同的单位，就会困难得多，甚至是不可能的。在写作本书时，为了让创作人员能够保证一致的基线间隔而不论内联内容是什么，已经提出了很多属性提案，这会大大简化行内格式化的某些方面。不过，所建议的这些属性都没有具体实现，所以其采纳还遥遥无期。

缩放行高

可以看到，设置line-height的最好办法是使用一个原始数字值。之所以说这种方法最好，原因是这个数会成为缩放因子，而该因子是一个继承值而非计算值。假设你希望一个文档中所有元素的line-height都是其font-size的1.5倍，可以如下声明：

```
body {line-height: 1.5;}
```

缩放因子1.5在元素间逐层传递，在各层上，这个因子都作为一个乘数与各元素的font-size相乘。因此，以下标记会得到如图7-39所示的结果：

```
p {font-size: 15px; line-height: 1.5;}
small {font-size: 66%;}
big {font-size: 200%;}

<p>This paragraph has a line-height of 1.5 times its font-size. In addition,
any elements within it <small>such as this small element</small> also have
line-heights 1.5 times their font-size...and that includes <big>this big
element right here</big>. By using a scaling factor, line-heights scale
to match the font-size of any element.</p>
```

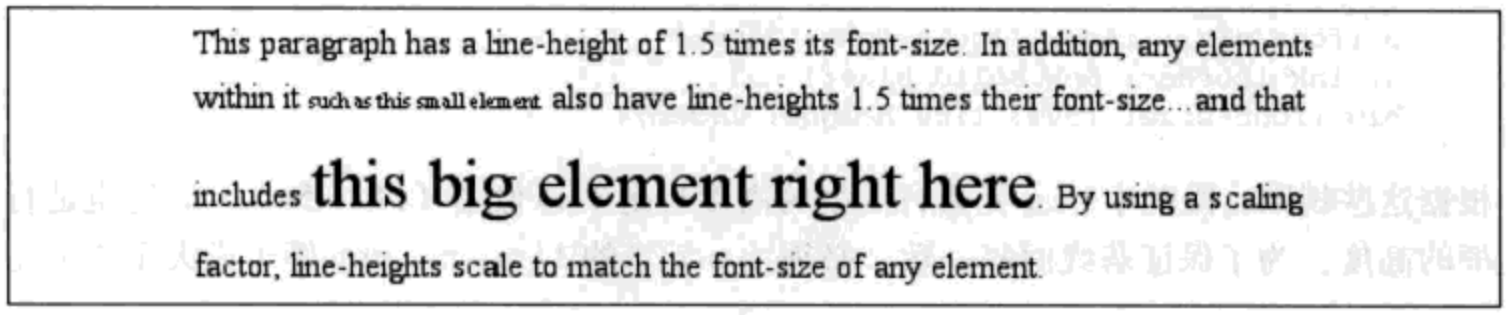

图7-39：对line-height使用缩放因子

在本例子中，small元素的行高为15px，而对于big元素则为45px（这些数看上去有些过分，不过它们与总的页面设计是一致的）。当然，如果你不希望big文本生成太多行间距，可以为它另外指定一个line-height值，这会覆盖继承的缩放因子：

```
p {font-size: 15px; line-height: 1.5;}
small {font-size: 66%;}
big {font-size: 200%; line-height: 1em;}
```

还有一种解决方案（可能这是最简单的一种方法），即适当地设置样式，使行高恰好能包含行内容，而没有多余的空间。这里你可能会使用line-height值1.0。这个值乘以每个font-size，其结果与各元素的font-size值完全相等。因此，对于每个元素，行内框与内容区相同，这意味着会使用所需的绝对最小大小来包含各元素的内容区。

注意：大多数字体在字符字形行之间还显示有一点空间，因为字符往往比其em框要小。只有script（“cursive”）字体例外，其字符字形往往大于其em框。

增加框属性

从前面的讨论可以了解到，内边距、外边距和边框都可以应用于行内非替换元素。行内元素的这些方面根本不会影响行框的高度。如果对一个无内外边距的span元素应用某个边框，可能得到如图7-40所示的结果。

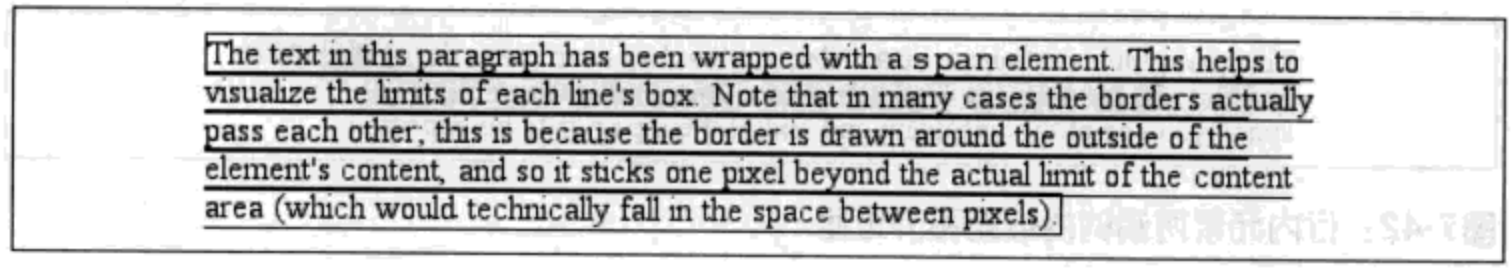

图7-40：行内边框和行框布局

行内元素的边框边界由font-size而不是line-height控制。换句话说，如果一个span元素的font-size为12px，line-height为36px，其内容区就是12px高，边框将包围该内容区。

或者，可以为行内元素指定内边距，这会把边框从文本本身拉开：

```
span {border: 1px solid black; padding: 4px;}
```

注意，这个内边距并没有改变内容区的具体形状，不过它会影响这个元素行内框的高度。类似地，向一个行内元素增加边框也不会影响行框的生成和布局，如图7-41所示。

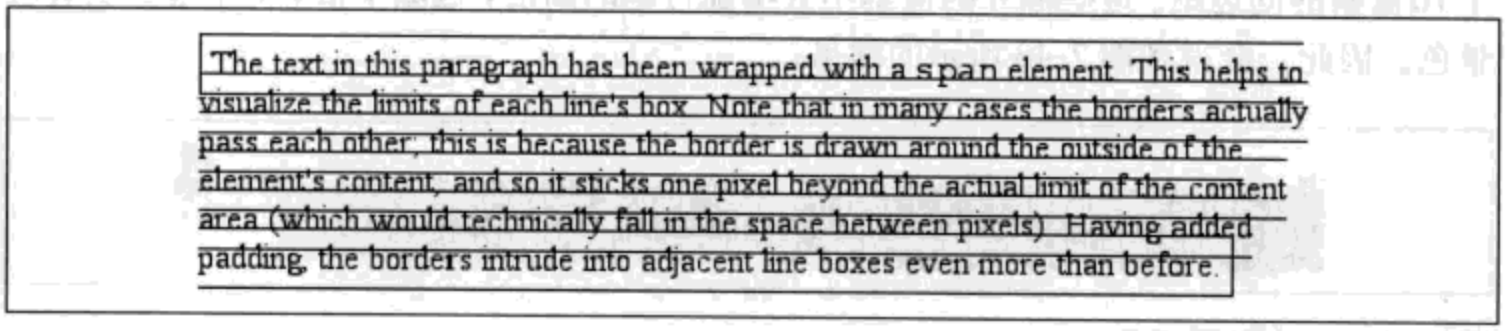

图7-41：内边距和边框不改变行高

至于外边距，实际上，外边距不会应用到行内非替换元素的顶端和底端，它们不影响行框的高度。不过，行内元素的两端则是另一回事。

注意： CSS2.1中明确指定了外边距的放置：它定义了margin-top和margin-bottom（可以应用到不是行内非替换元素的所有其他元素），而不是简单地说用户代理应当忽略上、下外边距。

应当还记得，行内元素基本上会作为一行放置，然后分成多个部分。所以，如果向一个行内元素应用外边距，这些外边距将出现在其开始和末尾；分别为左、右外边距。内边距也出现在边界上。因此，尽管内边距和外边距（以及边框）不影响行高，但是它们确实能影响一个元素内容的布局，可能将文本推离其左右两端。实际上，如果左、右外边距为负，可能会把文本拉近行内元素，甚至导致重叠，如图7-42所示。

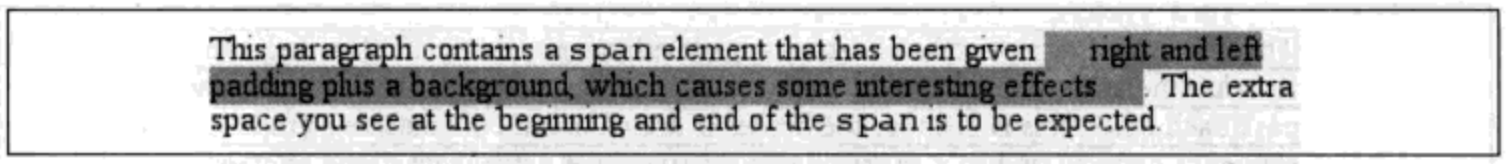

图7-42：行内元素两端的内边距和外边距

可以把行内元素想成是一个纸片，外围有一些塑料边。在多行上显示行内元素就像是把一个大纸片剪成一些小纸片。不过，每个小纸片上不会增加额外的塑料边。小纸片上的塑料边还是最初那个大纸片上的塑料边，所以看上去只是原来纸片（行内元素）最前和最后两端上出现塑料边。

所以，如果行内元素有一个背景，而且内边距足够大以至于行背景重叠，此时会发生什么情况呢？看下面的例子：

```
p {font-size: 15px; line-height: 1em;}
p span {background: #999; padding-top: 10px; padding-bottom: 10px;}
```

span元素中的所有文本都有15像素高的内容区，而且为各内容区的顶部和底部各增加了10像素的内边距。这些额外的像素不会增加行框的高度，这原本很好，不过这里有背景色。因此，会得到图7-43所示的结果。

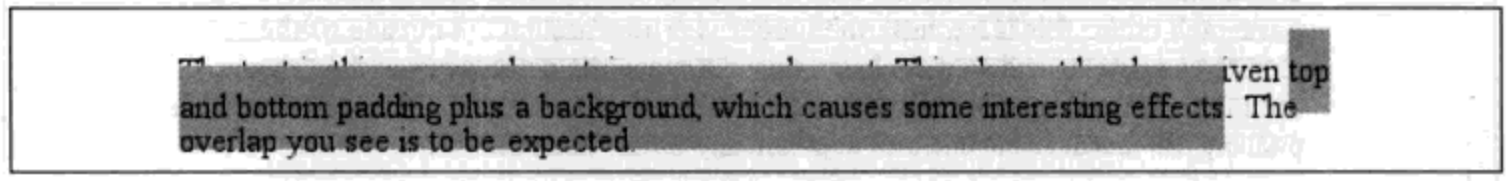

图7-43：行内背景重叠

CSS 2.1明确指出行框按文档的顺序绘制："这会导致后续行的边框在前面行的边框和文本上绘制。"这个原则同样适用于背景，如图7-43所示。另一方面，CSS2允许用户代理"'切掉'边框和内边距区（也就是不显示边框和内边距）"。因此，具体结果可能很大程度上取决于用户代理遵循哪一个规范。

行内替换元素

一般认为行内替换元素（如图像）有固有的高度和宽度；例如，一个图像的高度和宽度可能是某个像素数。因此，有固有高度的替换元素可能导致行框比正常要高。这不会改变行中任何元素的line-height值，包括替换元素本身。相反，只是会让行框的高度恰好能包含替换元素（以及所有框属性）。换句话说，会用替换元素整体（包括内容、外边距、边距和内边距）来定义元素的行内框。以下样式就能得到这样一个例子，如图7-44所示：

字形与内容区

你可能会尽力避免行内非替换元素的背景重叠，尽管如此，这种情况还是可能发生，这取决于使用何种字体。问题在于一个字体的em框与其字符字形之间可能存在差别。可以看到，对于大多数字体，其em框的高度与字符字形的高度都不一致。

这听上去可能很抽象，不过有很实际的后果。CSS2.1中指出："内容区的高度应当基于字体，但是这个规范并没有指定如何基于字体确定内容区的高度。用户代理可能……使用em框，也可能使用字体的最大上升变形和下降变形（如果使用字体的最大上升变形和下降变形，能确保字形中落在em框上面或下面的部分仍在内容区内，但是这样一来，不同的字体会有不同大小的框）。"

换句话说，一个行内非替换元素的"绘制区"要由用户代理来决定。如果一个用户代理使em框的高度作为内容区的高度，那么行内非替换元素的背景就与em框的高度相等（即值font-size）。如果用户代理使用字体的最大上升变形和下降变形，背景就可能比ex框高或矮。因此，尽管可以为行内非替换元素指定line-height为1em，但其背景还是有可能与其他行的内容重叠。

在CSS2或CSS2.1中没有办法避免这种重叠，不过已经建议CSS3增加一些属性，允许创作人员对用户代理的行为进行控制。在这些属性得到广泛实现之前，用CSS无法得到真正准确的格式编排。

```
p {font-size: 15px; line-height: 18px;}
img {height: 30px; margin: 0; padding: 0; border: none;}
```

The text in this paragraph contains an img element. This element has been given a height that is larger than a typical line box height for this paragraph, which leads to some potentially unwanted consequences. The extra space you see between lines of text is to be expected.

图7-44：替换元素可以增加行框的高度，但不影响line-height值

尽管有所有这些空白，但段落或图像本身的line-height有效值并没有因此改变。line-height对图像的行内框没有任何影响。由于图7-44中的图像没有内边距、外边距或边框，其行内框与其内容区相同，在这里就是30像素高。

然而，行内替换元素还是有一个line-height值。为什么呢？在最常见的情况下，行内替换元素需要这个值，从而在垂直对齐时能正确地定位元素。例如，要记住，vertical-align的百分数值要相对于元素的line-height来计算。所以：

```
p {font-size: 15px; line-height: 18px;}
img {vertical-align: 50%;}

<p>The image in this sentence <img src="test.gif" alt="test image">
will be raised 9 pixels.</p>
```

line-height的继承值使图像上升9个像素(而不是其他数字)。如果没有line-height值，它就无法完成百分数值指定的垂直对齐。对于垂直对齐来说，图像本身的高度无关紧要；关键是 line-height 的值。

不过，对于其他替换元素，将line-height值传递到该替换元素中的后代元素可能很重要。SVG 图像就是这样一个例子，它使用 CSS 对图像中的所有文本设置样式。

增加框属性

有了以上了解，看上去向行内替换元素应用外边距、边距和内边距似乎很简单。

内边距和边框像平常一样应用到替换元素；内边距在具体内容外插入空间，边框围绕着内边距。这个过程的不寻常之处在于，内边距和边框确实会影响行框的高度，因为它们要作为行内替换元素的行内框的一部分（不同于行内非替换元素）。考虑图 7-45，这是由以下样式得到的：

```
img {height: 20px; width: 20px;}
img.one {margin: 0; padding: 0; border: 1px dotted;}
img.two {margin: 5px; padding: 3px; border: 1px solid;}
```

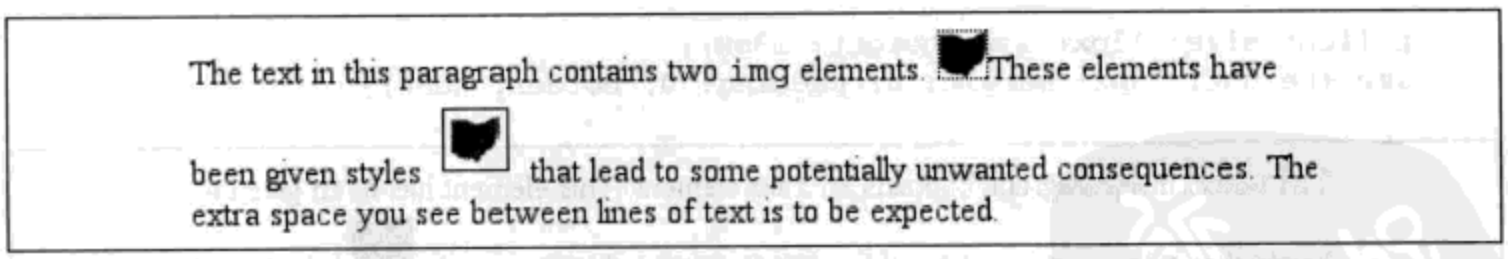

图7-45：为行内替换元素增加内边距、边框和外边距会增大其行内框

注意，第一个行框的高度足以包含这个图像，第二个行框的高度则足以包含图像、其内边距和边框。

外边距也包含在行框中，不过外边距有自己的问题。设置正外边距没有什么特殊的地方，只是使替换元素的行内框更高。设置负外边距也有类似的效果：这会减少替换元素行内框的大小。如图 7-46 所示，可以看到，负的上外边距会把图像上面的一行向下拉：

```
img.two {margin-top: -10px;}
```

当然，负外边距对块级元素有也同样的作用。在这种情况下，负外边距会使替换元素的行内框小于正常大小。负外边距是使行内替换元素挤入其他行的唯一办法。

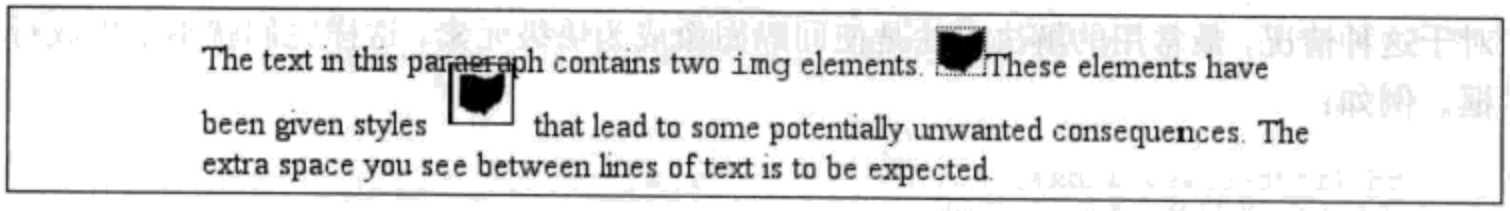

图7-46：行内替换元素有负外边距的效果

替换元素和基线

你现在可能注意到了，默认地，行内替换元素位于基线上。如果向替换元素增加下内边距、外边距或边框，内容区会上移。替换元素并没有自己的基线，所以相对来讲最好的办法是将其行内框的底端与基线对齐。因此，实际上是下外边距边界与基线对齐，如图7-47所示。

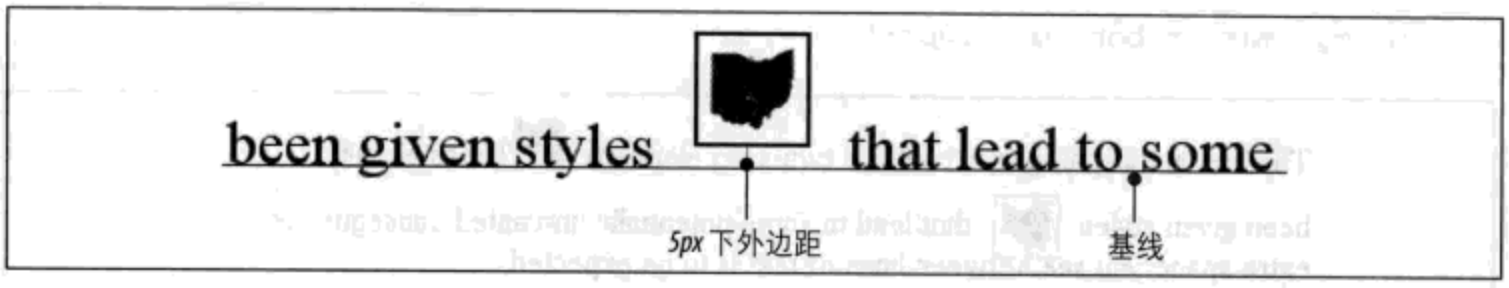

图7-47：行内替换元素位于基线上

这种基线对齐会有一个意想不到（而且不受欢迎）的后果：如果一个表单元格中只有一个图像，这个图像要让表单元格足够高，从而能把包含该图像的行框包含在内。即使没有具体的文本，甚至没有空白符，包含图像的表单元格中还是会出现这种大小调整。因此，已经使用多年的分片图像和间隔GIF设计在现代浏览器中可能表现很糟糕。考虑以下最简单的情况：

```
td {font-size: 12px;}

<td><img src="spacer.gif" height="1" width="10"></td>
```

在CSS行内格式化模型中，表单元格将是12像素高，图像位于单元格的基线上。所以图像下面可能有3像素的空间，上面有8像素的空间，不过具体的距离要取决于所用的字体系列及其基线的位置。这种行为并不仅限于表单元格中的图像；只要一个行内替换元素是块级元素或表单元格元素中的唯一后代，都会有这种行为。例如，div中的一个图像也会放在基线上。

注意：写作本书时，许多浏览器实际上会忽略这个CSS行内格式化模型，不过基于Gecko的浏览器在显示XHTML和严格的HTML文档时确实会像上面那样做。更多相关信息请参考我的文章《Images, Tables, and Mysterious Gaps》（*http://developer.mozilla.org/en/docs/Images,_Tables,_and_Mysterious_Gaps*）。

对于这种情况，最常用的解决方法是使间隔图像成为块级元素，这样它们就不会生成行框。例如：

```
td {font-size: 12px;}
img.block {display: block;}

<td><img src="spacer.gif" height="1" width="10" class="block"></td>
```

另一个可取的修正办法是，将包含图像的表单元格的font-size和line-height都设置为1px，这会使行框的高度只能放下1像素的图像。

行内替换元素位于基线上还有一个有意思的效果：如果应用一个负的下外边距，元素实际上会被向下拉，因为其行内框的底端将比其内容区的底端高。因此，以下规则会得到如图7-48所示的结果：

```
p img {margin-bottom: -10px;}
```

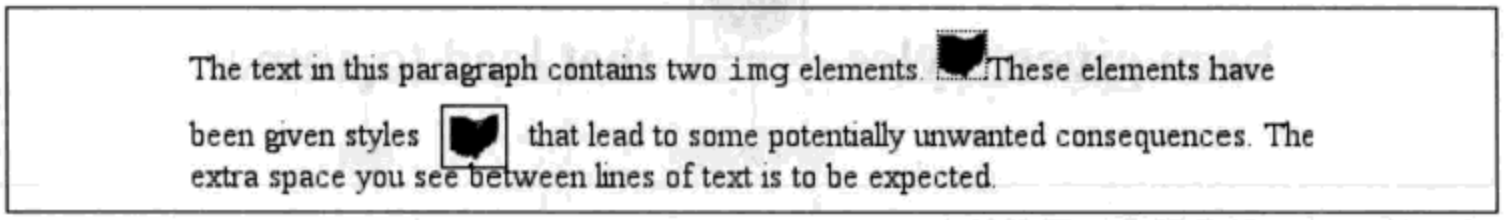

图7-48：负的下外边距会使行内替换元素向下拉

这很可能使得一个替换元素挤进后面的文本行，如图7-48所示。

警告：有些浏览器只是把内容区的底端放在基线上，而忽略负的下外边距。

行内模型历史溯源

CSS行内格式化模型看上去可能有些没必要的复杂，而且在某些方面，甚至与创作人员的意愿相违背。遗憾的是，我们现在要创建的是这样一种样式语言，它既能与CSS之前的Web浏览器向后兼容，还要为将来扩展到更复杂的领域敞开大门，它将过去和现在笨拙地混合在一起，而上述行内格式化模型的这种复杂性正是这样做的直接后果。另外还有一个原因，我们可能会做一些合理的决策来避免一个不期望的后果，但这可能又会导致另一个不期望的后果出现。

例如，有图像和垂直对齐文本的文本行会“散开”，究其原因，这要归根于Mosaic 1.0的做法。在这种浏览器中，段落中的所有图像都会留出足够大的空间来包含该

图像。这种做法很好，因为这样可以避免图像与其他行中的文本重叠。所以，在CSS引入为文本和行内元素设置样式的方法时，设计者则尽力创建这样一个模型，（默认地）它不会导致行内图像与其他文本行重叠。不过，这个模型也意味着存在另外一些问题，例如，上标元素（sup）很可能也会使行拉开距离。

这种效果使一些创作人员很恼火，他们希望行基线之间的距离应该固定，不过再来看另一种情况。如果line-height要求基线之间的距离是指定的，最后很可能使行内替换元素和垂直移动元素与其他文本行重叠——这也会使创作人员不满意。幸运的是，CSS有足够强大的功能，总能以这样或那样的某种方式得到你想要的效果，CSS的将来还会有更大潜力。

改变元素显示

第1章曾简单地提到，可以为属性display设置一个值来影响用户代理显示的方式。既然我们已经深入地了解了视觉格式化，下面再使用本章的概念复习display属性，并讨论它的另外两个值。

display

值：	none | inline | block | inline-block | list-item | run-in | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | inherit
初始值：	inline
应用于：	所有元素
继承性：	无
计算值：	对于浮动、定位和根元素，计算值可变（见CSS2.1第9.7节）；否则，根据指定确定
说明：	CSS2中还有值compact和marker，不过由于缺乏广泛的支持，在CSS2.1中已经去掉

我们将忽略与表有关的值，它们将在第11章讨论；在此还忽略了值list-item，因为这个值将在第12章详细介绍。我们已经花了很大篇幅讨论块元素和行内元素，下面介绍inline-block和run-in元素，不过之前先花点时间介绍改变元素显示角色将如何改变布局。

改变角色

设置一个文档的样式时，如果能改变元素的显示角色显然很方便。例如，假设一个div中有一系列链接，你想把这个div布局为一个垂直边栏：

```
<div id="navigation">
<a href="index.html">WidgetCo Home</a><a href="products.html">Products</a>
<a href="services.html">Services</a><a href="fun.html">Widgety Fun!</a>
<a href="support.html">Support</a><a href="about.html" id="current">About Us</a>
<a href="contact.html">Contact</a>
</div>
```

可以把所有链接都放在表单元格中，或者每个链接都包在其自己的div中，或者还可以让它们都是块级元素，如下：

```
div#navigation a {display: block;}
```

这会让导航栏div中的每个a元素都是一个块级元素。如果再增加一些样式，可以得到图7-49所示的结果。

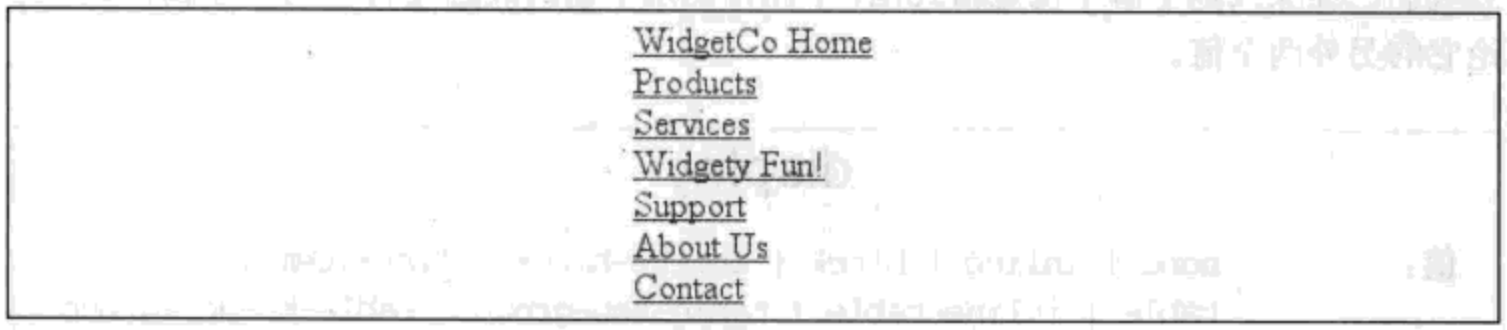

图7-49：将显示角色由inline改为block

对于不支持CSS的浏览器，尽管导航链接作为行内元素，但你希望浏览器能把它们作为块级元素摆放，在这些情况下，改变显示角色就很有用。将链接作为块级元素，就可以像处理div元素一样设置样式，这有一个好处，即整个元素框会成为链接的一部分。因此，如果用户的鼠标指针停留在元素框的某处，他就能点击这个链接。

你可能还想让元素作为行内元素。假设有一个无序的人名列表：

```
<ul id="rollcall">
<li>Bob C.</li>
<li>Marcio G.</li>
<li>Eric M.</li>
<li>Kat M.</li>
<li>Tristan N.</li>
<li>Arun R.</li>
<li>Doron R.</li>
<li>Susie W.</li>
</ul>
```

对于以上标记，假设你想让这些名字成为一系列行内名，其间用竖线间隔（另外在列表的左右两端也有竖线）。为此，唯一的办法就是修改其显示角色。以下规则将得到如图7-50所示的效果：

```
#rollcall li {display: inline; border-right: 1px solid; padding: 0 0.33em;}
#rollcall li:first-child {border-left: 1px solid;}
```

| Bob C. | Marcio G. | Eric M. | Kat M. | Tristan N. | Arun R. | Doron R. | Susie W. |

图7-50：将显示角色由list-item改为inline

还可以采用很多其他方法来在设计中充分利用display。开动脑筋，看看你能有哪些创造！

不过有一点要注意，你改变的只是元素的显示角色，而不是其本质。换句话说，让一个段落生成行内框并不会把这个段落真正变成一个行内元素。例如，在XHTML中，有些元素是块元素，另外一些是行内元素（还有一些是“流”元素，不过暂时不考虑这些元素）。行内元素可能是一个块元素的后代，但是反过来则不行。因此，尽管链接可以放在一个段落中，但是链接却不能包围段落。不论如何设计元素的样式，都是如此。考虑以下标记：

```
<a href="http://www.example.net" style="display: block;">
<p style="display: inline;">this is wrong!</p>
</a>
```

这个标记将是无效的，因为块元素（p）嵌套在一个行内元素（a）中。改变显示角色也不会使以上标记变得合法。display之所以得名就是因为它影响的是元素如何显示，而不影响它是何种元素。

行内块元素

看上去值名inline-block是一个混合产物，实际上也确实如此，行内块元素（inline-block element）确实是块级元素和行内元素的混合。这个display值是CSS2.1中新增的。

行内块元素作为一个行内框与其他元素和内容相关。换句话说，它就像图像一样放在一个文本行中，实际上，行内块元素会作为替换元素放在行中。这说明，行内块元素的底端默认地位于文本行的基线上，而且内部没有行分隔符。

在行内块元素内部，会像块级元素一样设置内容的格式。就像所有块级或行内替换元素一样，行内块元素也有属性width和height，如果比周围内容高，这些属性会使行高增加。

下面来考虑一些示例标记，它们能更清楚地说明这一点：

```
<div id="one">
This text is the content of a block-level level element.  Within this
block-level element is another block-level element.  <p>Look, it's a
block-level
paragraph.</p>  Here's the rest of the DIV, which is still block-level.
</div>
<div id="two">
This text is the content of a block-level level element.  Within this
block-level element is an inline element.  <p>Look, it's an inline
paragraph.</p>  Here's the rest of the DIV, which is still block-level.
</div>
<div id="three">
This text is the content of a block-level level element.  Within this
block-level element is an inline-block element.  <p>Look, it's an inline-
block
paragraph.</p>  Here's the rest of the DIV, which is still block-level.
</div>
```

对以上标记，应用下面的规则：

```
div {margin: 1em 0; border: 1px solid;}
p {border: 1px dotted;}
div#one p {display: block; width: 6em; text-align: center;}
div#two p {display: inline; width: 6em; text-align: center;}
div#three p {display: inline-block; width: 6em; text-align: center;}
```

样式表的结果如图7-51所示。

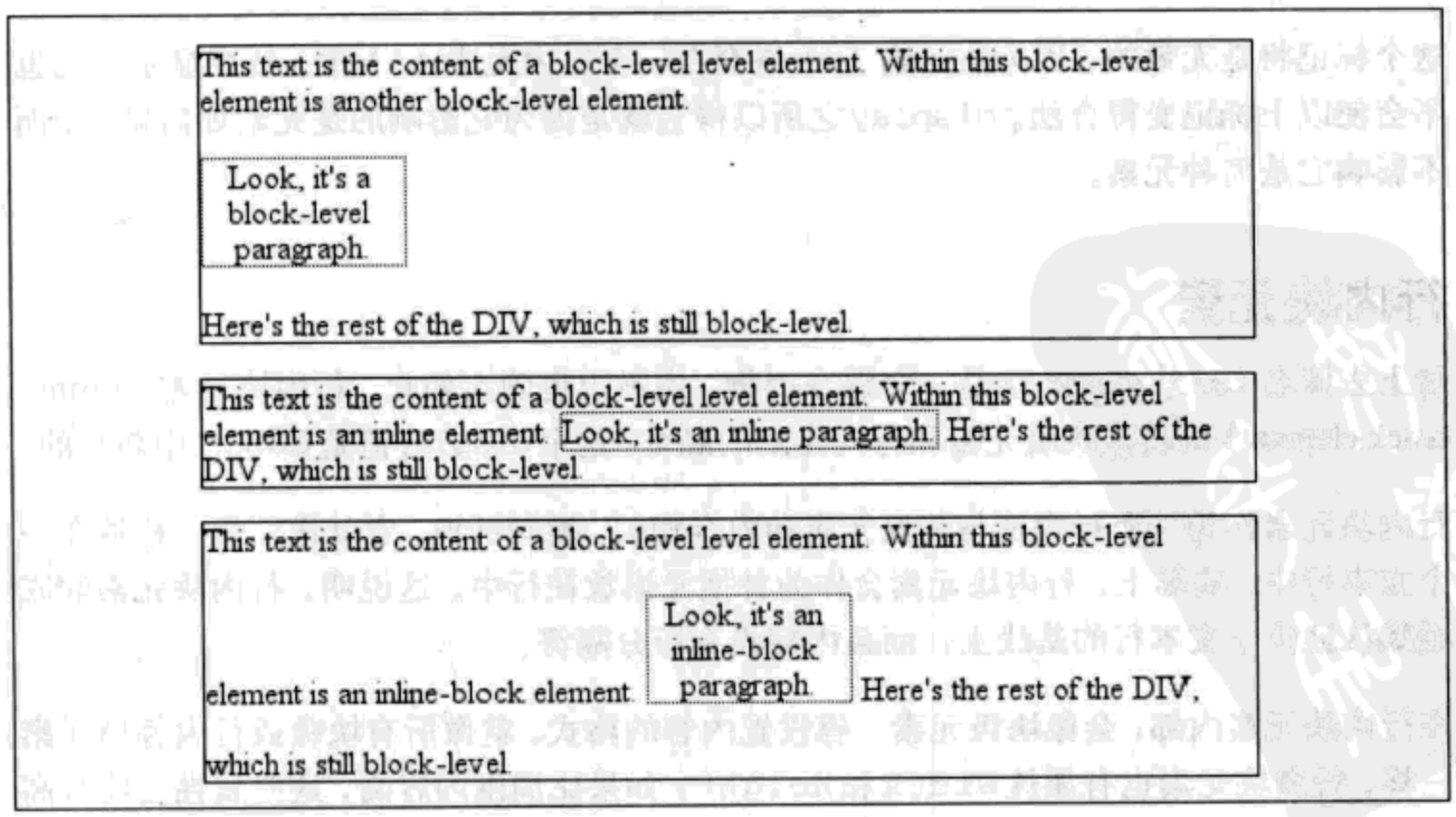

图7-51：行内块元素的行为

注意，在第二个div中，行内段落格式化为正常的行内内容，这说明width和text-align被忽略了（它们不能应用于行内元素）。不过，对于第三个div元素，作为行内块元素的段落则有这两个属性，因为它作为一个块级元素被格式化。这个段落还要求文本行更高一些，因为它会影响行高，就好像这是一个替换元素一样。

如果行内块元素的width未定义，或者显式声明为auto，元素框会收缩以适应内容。也就是说，元素框的宽度刚好足够包含该内容，而没有多余的空间。行内框也会这样做，不过行内框可能会跨多个文本行，而行内块元素不能。因此，以下规则应用到前面的示例标记时：

```
div#three p {display: inline-block; height: 2em;}
```

会创建一个较高的框，它的宽度刚好能包含内容，如图7-52所示。

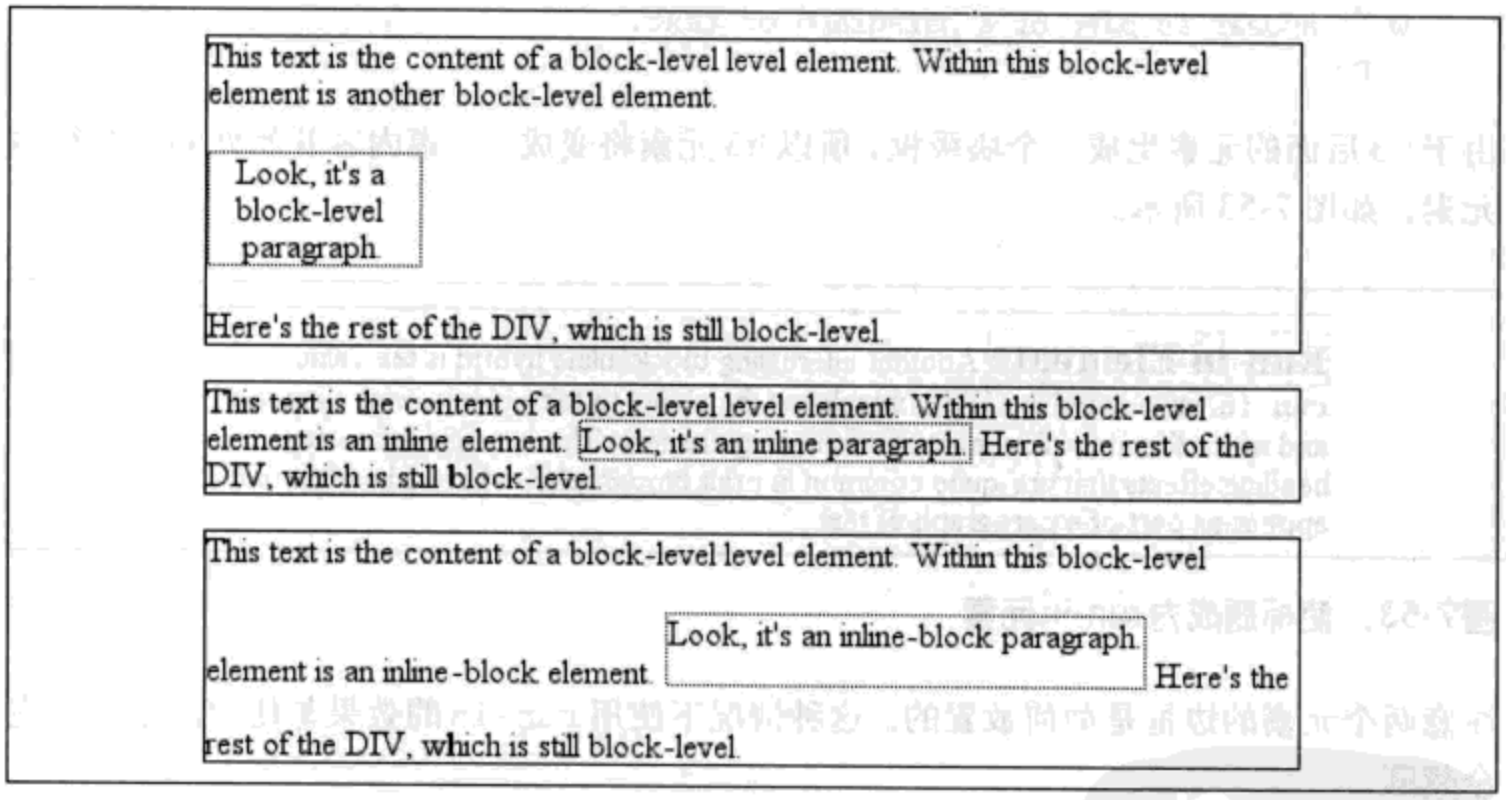

图7-52：行内块元素自动调整大小

有时行内块元素很有用，例如，如果有5个超链接，你希望它们在一个工具条中宽度相等。为了让它们分别占其父元素宽度的20%，但是仍保持其为行内元素，可以声明如下：

```
#navbar a {display: inline-block; width: 20%;}
```

run-in元素

CSS2引入了一个值run-in，这也是一个有意思的块/行内元素混合，可以使某些块级元素成为下一个元素的行内部分。这种功能对于某些标题效果很有用，这在打印排版中相当常见，即标题作为文本段落的一部分出现。

在CSS中，只需改变元素的display值，并使下一个元素框作为块级元素框，就可以使元素成为run-in元素。注意，这里我所说的是框，而不是元素本身。换句话说，不论元素是块元素还是行内元素都无关紧要，重要的是元素生成的框。设置为display: block的strong元素会生成一个块级框，设置为display: inline的段落则会生成一个行内框。

所以，重申一句：如果一个元素生成run-in框，而且该框后面是一个块级框，那么该run-in元素将成为块级框开始处的一个行内框。例如：

```
<h3 style="display: run-in; border: 1px dotted; font-size: 125%;
  font-weight: bold;">Run-in Elements</h3>
<p style="border-top: 1px solid black; padding-top: 0.5em;">
Another interesting block/inline hybrid is the value <code>run-in</code>,
introduced in CSS2, which has the ability to take block-level elements and
make them an inline part of a following element. This is useful for certain
heading effects that are quite common in print typography, where a heading
will appear as part of a paragraph of text.
</p>
```

由于h3后面的元素生成一个块级框，所以h3元素将变成p元素内容开始处的一个行内元素，如图7-53所示。

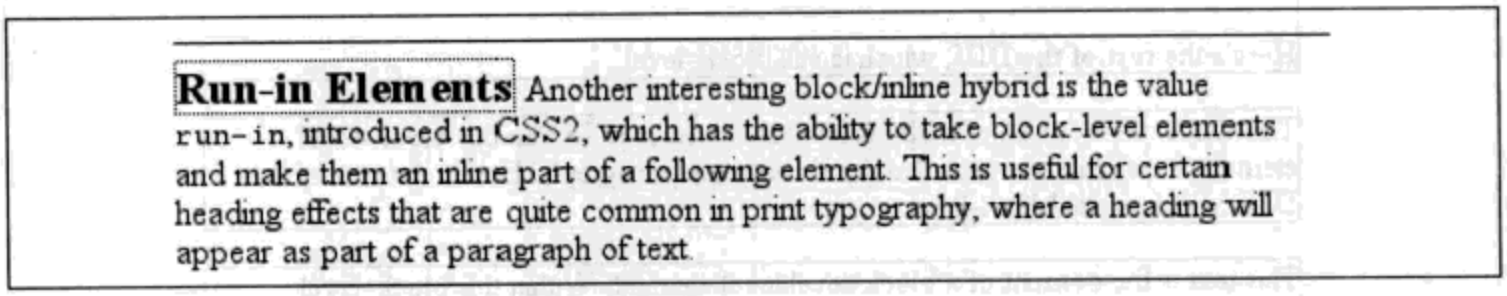

图7-53：使标题成为run-in元素

注意两个元素的边框是如何放置的。这种情况下使用run-in的效果与使用以下标记完全相同：

```
<p style="border-top: 1px solid black; padding-top: 0.5em;">
<span style="border: 1px dotted; font-size: 125%; font-weight: bold;">Run-in
Elements</span> Another interesting block/inline hybrid is the value
<code>run-in</code>, introduced in CSS2, which has the ability to take
block-level elements and make them an inline part of a following element.
This is useful for certain heading effects that are quite common in print
typography, where a heading will appear as part of a paragraph of text.
</p>
```

不过，run-in框与前面的标记示例之间还存在一个小小的差别。即使run-in框格式化为另一个元素中的行内框，它们仍从文档中的父元素继承属性，而不是说它们放在哪个元素中就从哪个元素继承属性。再来扩展前面的例子，在最外面加一个div，并增加一些颜色：

```
<div style="color: silver;">
<h3 style="display: run-in; border: 1px dotted; font-size: 125%;
  font-weight: bold;">Run-in Elements</h3>
<p style="border-top: 1px solid black; padding-top: 0.5em; color: black;">
Another interesting block/inline hybrid is the value <code>run-in</code>,
introduced in CSS2, which has the ability to take block-level elements and
make them an inline part of a following element.
</p>
</div>
```

在这种情况下，h3将是银色而不是黑色，如图7-54所示。这是因为在插入到段落之前，它从其父元素（div）继承了颜色值。

Run-in Elements Another interesting block/inline hybrid is the value run-in, introduced in CSS2, which has the ability to take block-level elements and make them an inline part of a following element. This is useful for certain heading effects that are quite common in print typography, where a heading will appear as part of a paragraph of text.

图7-54：Run-in元素从其原先的父元素继承属性

要记住重要的一点，只有当run-in框后面是一个块级框时run-in才起作用。如果不是这样，run-in框本身将成为块级框。因此，给定以下标记，h3仍然是（或者会变成）块级元素，因为table元素的display值是（非常奇怪）table：

```
<h3 style="display: run-in;">Prices</h3>
<table>
<tr><th>Apples</th><td>$0.59</td></tr>
<tr><th>Peaches</th><td>$0.79</td></tr>
<tr><th>Pumpkin</th><td>$1.29</td></tr>
<tr><th>Pie</th><td>$6.99</td></tr>
</table>
```

创作人员不太可能将值run-in应用到一个自然的行内元素，但是一旦发生这种情况，这个元素则极有可能生成一个块级框。例如，以下标记中的em元素将成为块级元素，因为它后面没有跟一个块级框：

```
<p>
This is a <em>really</em> odd thing to do, <strong>but</strong> you could
do it
if you were so inclined.
</p>
```

警告： 写作本书时，很少有浏览器对run-in提供支持。

计算值

如果元素是浮动元素或定位元素，display的计算值可以改变。如果为一个根元素声明display值，计算值也可以改变。实际上，display、position和float值会以很有意思的方式相互影响。

如果一个元素是绝对定位元素，float的值设置为none。对于浮动元素或绝对定位元素，计算值由声明值确定，如表7-1所示。

表7-1：display计算值

声明值	计算值
inline-table	table
inline, run-in, table-row-group, table-column, table-column-group, table-header-group, table-footer-group, table-row, table-cell, table-caption, inline-block	block
所有其他	根据指定确定

对于根元素，如果声明为值inline-table或table，都会得到计算值table，声明为none时则会得到同样的计算值（none）。所有其他display值都计算为block。

小结

尽管CSS格式化模型的某些方面乍看起来有些不太直观，不过等你多熟悉一些就会发现这是有道理的。很多情况下，最初看上去没道理甚至荒谬的规则最后看来确实是合理的，它们会防止一些奇怪的或我们不期望的文档显示结果。在很多方面，块级元素都很容易理解，调整其布局通常是一个简单的任务。另一方面，行内元素则可能很难管理，因为有很多影响因素，其中也包括元素是替换元素还是非替换元素。既然我们已经对文档布局的基础知识有所了解，下面再来关注如何使用各种布局属性。后面的几章都在讨论这个方面，首先来看最常用的框属性：内边距、边框和外边距。

第8章

内边距、边框和外边距

如果你像大多数Web设计人员一样，也是在20世纪90年代末期开始投入到这个工作，你的页面很可能全部使用表来建立布局。当然，之所以这样设计，原因是表可以用来创建边栏，还可以为整个页面的外观建立一种复杂的结构。你甚至还会使用表来完成一些比较简单的任务，如把文本放在一个有边框的有色框中。不过，再想想看，对于这么简单的任务实际上并不需要使用表。如果你只想要一个有红色边框、黄色背景的段落，与其把它包围在只包含一个单元格的表中，干脆直接创建不是更容易吗？

CSS的作者认为，这样确实更容易一些，所以他们下了很大功夫，允许你为段落、标题、`div`、锚和图像（几乎是Web页面可以包含的一切）定义边框。这些边框可以将一个元素与其他元素区别开，强调其外观，将某类数据标志为有改变，或者达到其他效果。

CSS还允许在一个元素外围定义一些区域，来控制如何相对于内容摆放边框，以及其他元素与该元素边框可以有多近。在一个元素的内容及其边框之间，可以看到元素的内边距（padding），边框外则是外边距（margins）。当然，这些属性影响着整个文档如何布局，不过，更重要的是，它们会严重影响给定元素的外观。

基本元素框

第7章曾讨论过，所有文档元素都生成一个矩形框，这称为元素框（element box），它描述了一个元素在文档布局中所占的空间大小。因此，每个框影响着其他元素框的位置和大小。例如，如果文档中第一个元素框是1英寸高，下一个框就至少会从文档顶端向

下1英寸处开始。如果第一个元素框改为2英寸高，后面的各元素框都会向下移1英寸，第二个元素框将至少从文档顶端向下2英寸处开始，如图8-1所示。

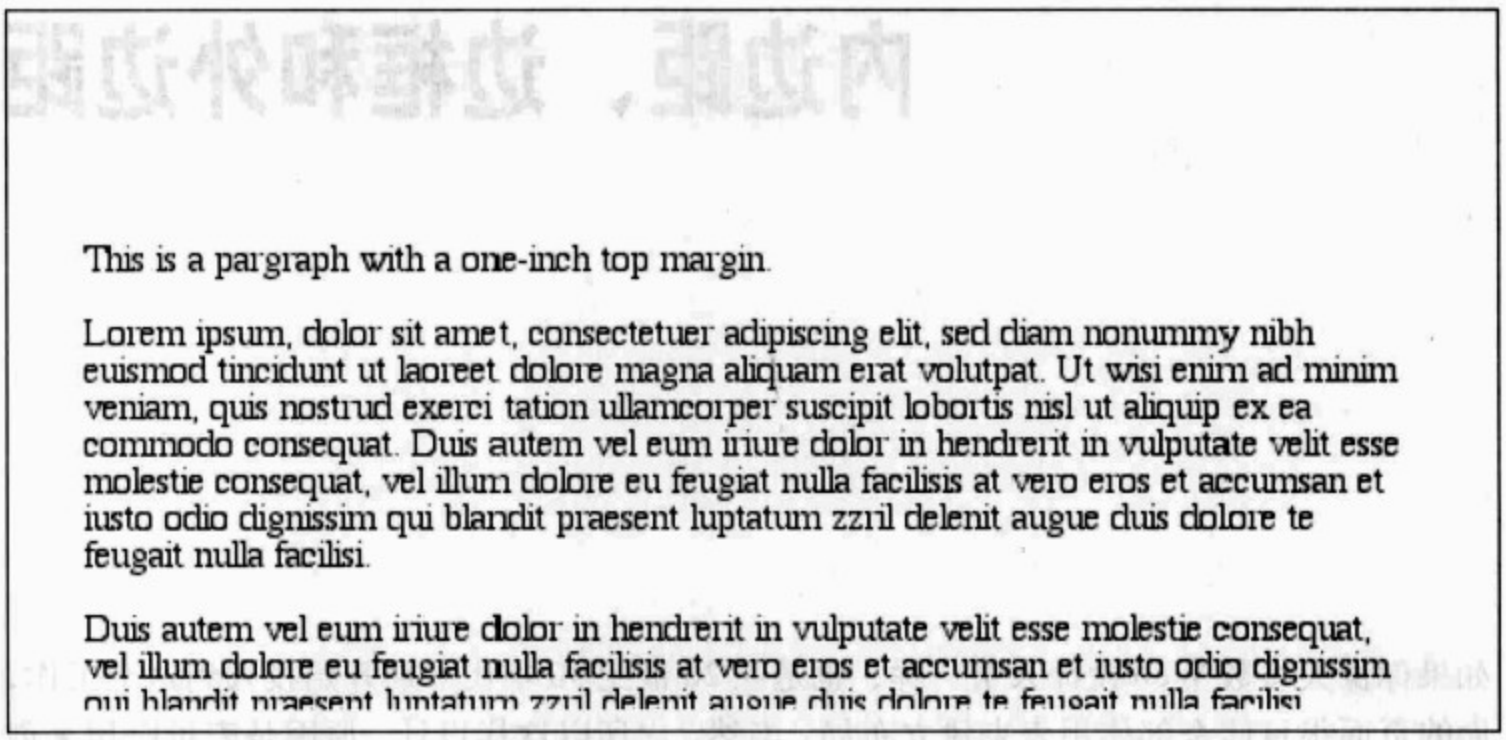

图8-1：一个元素如何影响所有元素

默认地，一个可以显示的文档由多个矩形框组成，这些矩形框分布开，从而不会相互重叠。另外，根据某些限制，这些框要尽可能地少占空间，同时还要保证相互之间有足够的空间，以便清楚地看出哪些内容属于哪个元素。

注意： 如果采用手工定位，框可能会重叠，另外如果在正常流元素上使用负的外边距，也可能出现视觉重叠。

为了充分理解如何处理外边距、内边距和边框，必须清楚地掌握框模型（这在第7章做过解释）。为了便于参考，这里摘录了第7章的框模型图（见图8-2所示）。

宽度和高度

如图8-2所示，一个元素的`width`被定义为从左内边界到右内边界的距离，`height`被定义为上内边界到下内边界的距离。

对于这两个属性有一点很重要：它们不能应用到行内非替换元素。例如，如果你想声明一个超链接的`height`和`width`，CSS兼容的浏览器必须忽略这些声明。假设应用以下规则：

```
a:link {color: red; background: silver; height: 15px; width: 60px;}
```

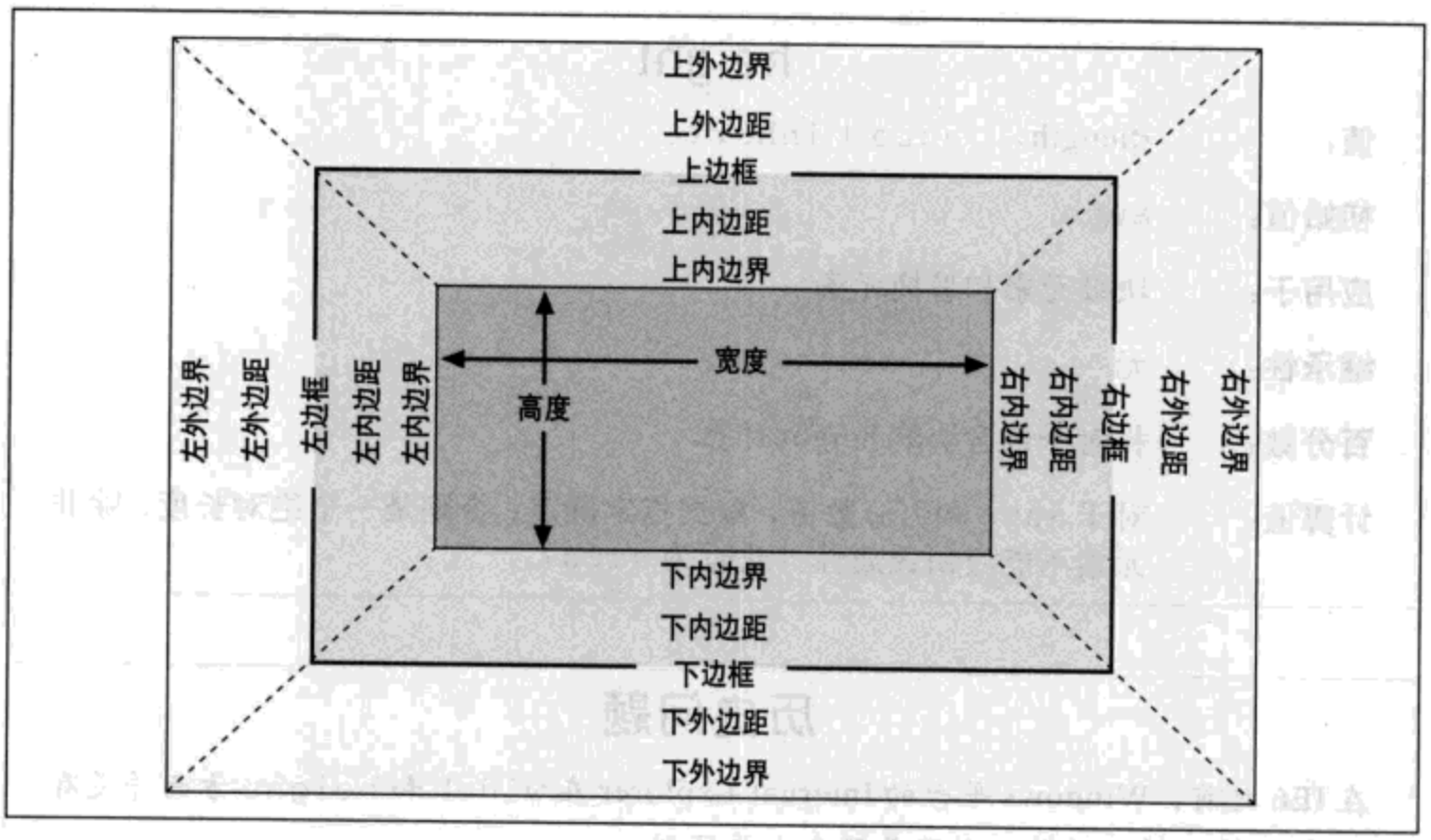

图8-2：CSS框模型

width

值：	<length> | <percentage> | auto | inherit
初始值：	auto
应用于：	块级元素和替换元素
继承性：	无
百分数：	相对于包含块的 width
计算值：	对于auto和百分数值，根据指定确定；否则是一个绝对长度，除非元素不能应用该属性（此时为 auto）

最后会得到有银色背景的红色链接，其高度和宽度由链接的内容确定，将不会是15像素高60像素宽。

在本章中，为了使讨论更简单一些，我们假设元素的高度总是自动计算。如果一个元素有8行，每一行高度为1/8英寸，则元素的高度为1英寸。如果有10行，高度则为1.25英寸。无论哪一种情况，高度都由元素的内容来定，而不是由创作人员确定。正常流中的元素很少有设定的高度。

height

值：	<length> | auto | inherit
初始值：	auto
应用于：	块级元素和替换元素
继承性：	无
百分数：	相对于包含块的 height 计算
计算值：	对于auto和百分数值，根据指定确定；否则是一个绝对长度，除非元素不能应用该属性（此时为 auto）

历史问题

在IE6之前，Windows平台的Internet Explorer在width和height方面并没有按CSS保证的那样做。以下是两个主要区别：

- IE/Win使用width和height来定义可见元素框的尺寸，而不是定义元素框的内容。如果定义一个元素的width为400px，IE/Win会使左外边框边界到右外边框边界之间的距离是400像素。换句话说，IE/Win使用width来描述元素内容区、左右内边距以及左右边框的总和。CSS3对此包含一些建议，允许创作人员决定width和height究竟是什么含义。
- IE/Win对行内非替换元素应用了width和height属性。例如，如果对一个超链接应用了width和height，将根据所提供的值来绘制。

这两种行为在IE6中得到了修正，不过仅限于“标准”模式。如果IE6以“quirks”模式显示文档，还是会有前面描述的行为。

外边距与内边距

元素框在元素之间只提供了很少的空间。有3种方法可以在元素外围生成额外的空间：可以增加内边距，或者增加外边距，还可以同时增加内边距和外边距。某些情况下，选择哪种方法并不重要。不过，如果元素有背景，则会影响你的决定，因为背景会延伸到内边距中，但不会延伸到外边距。

因此，为元素指定的内边距和外边距会影响元素的背景何时结束。如果为元素设置了背景色，如图8-3所示，可以清楚地看出二者的差别。有内边距的元素的背景范围更大，而有外边距的元素的背景则不受影响。

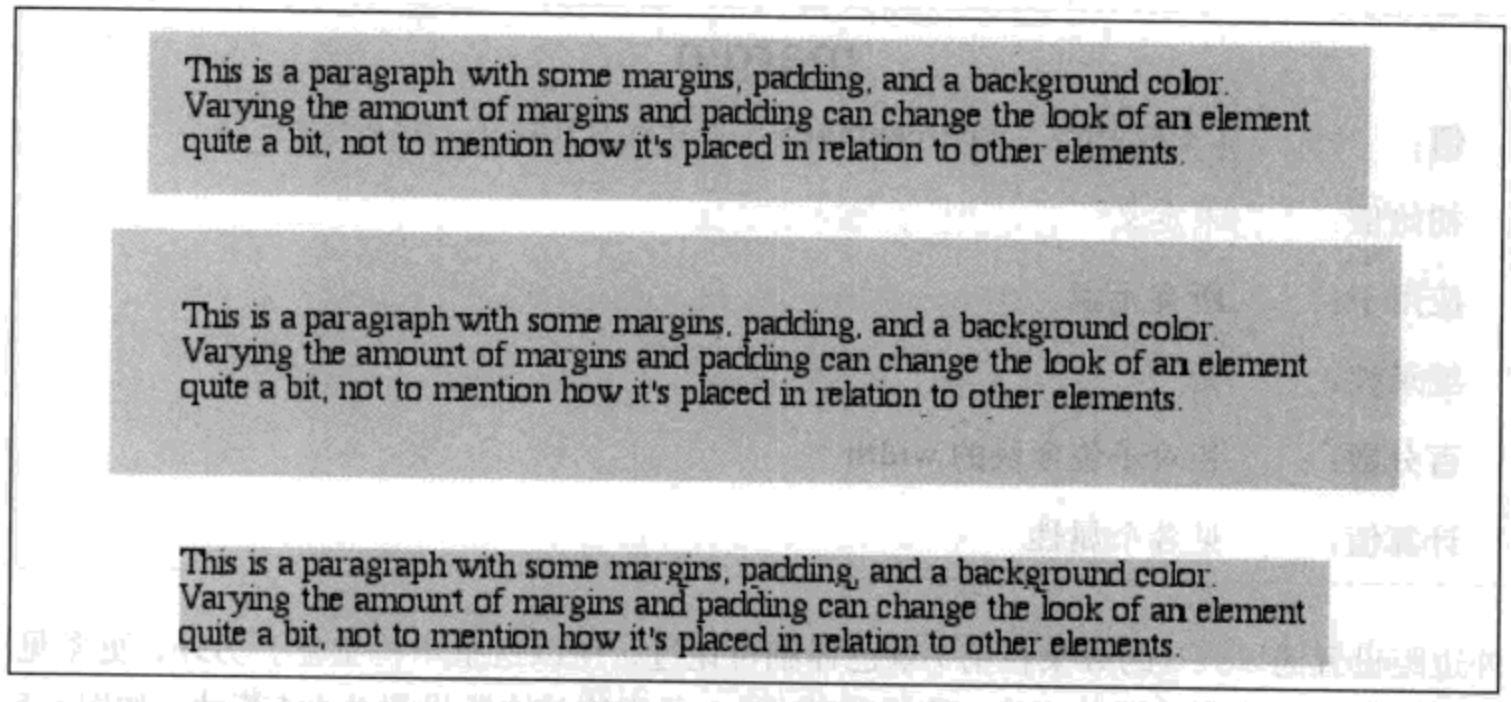

图8-3：有不同外边距和内边距的段落，利用背景说明其差别

最后，要由设计人员来决定如何设置外边距和内边距，他必须针对所要的效果权衡各种可能性，并选出最佳的做法。当然，为了做出这些选择，先了解可以使用哪些属性会很有帮助。

外边距

大多数正常流元素间出现的间隔都是因为存在元素外边距。设置外边距会在元素外创建额外的“空白”。“空白”通常指不能放其他元素的区域，而且在这个区域中可以看到父元素的背景。例如，图8-4显示了两个无外边距段落与两个有外边距段落之间的差别。

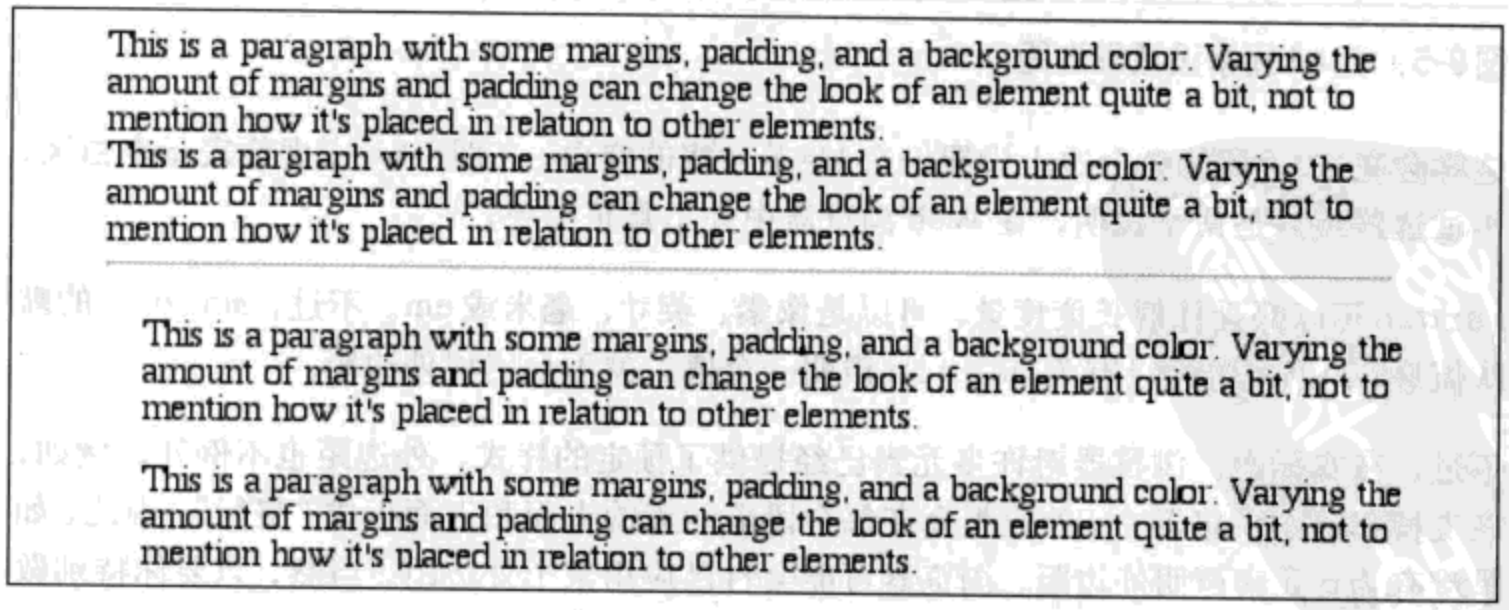

图8-4：有外边距和无外边距的段落

设置外边距的最简单的办法就是使用属性`margin`。

margin	
值：	[<length> | <percentage> | auto]{1,4} | inherit
初始值：	未定义
应用于：	所有元素
继承性：	无
百分数：	相对于包含块的 width
计算值：	见各个属性

外边距设置为auto的效果在第7章已详细讨论过，所以这里不再重复。另外，更常见的做法是为外边距设置长度值。假设你想将h1元素的外边距设置为1/4英寸，如图8-5所示（这里增加了背景色，以便你看清楚内容区的边界）。

```
h1 {margin: 0.25in; background-color: silver;}
```

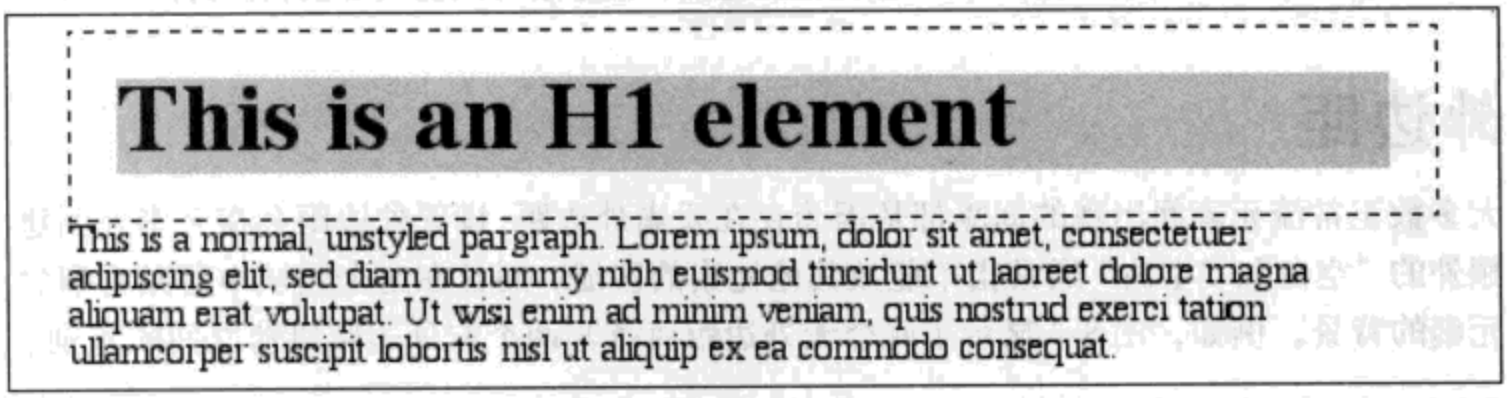

图8-5：为h1元素设置外边距

这样会在h1元素的各个边上设置一个1/4英寸宽的空白。在图8-5中，虚线表示空白区，不过这些线只是用于说明，在Web浏览器中并不真正出现。

margin可以接受任何长度度量，可以是像素、英寸、毫米或em。不过，margin的默认值是0，所以如果没有为margin声明一个值，就不会出现外边距。

不过，在实际中，浏览器对许多元素已经提供了预定的样式，外边距也不例外。例如，在支持CSS的浏览器中，外边距会在每个段落元素的上面和下面生成“空行”。因此，如果没有为p元素声明外边距，浏览器可能会自己应用某个外边距。当然，只要你特别做了声明，就会覆盖默认样式。

最后一点，还可以为margin设置一个百分数值。这个值类型的详细内容将在后面的“百分数和外边距”一节中讨论。

长度值和外边距

前面提到过，设置元素的外边距时，可以使用任何长度值。例如，要在段落元素外围应用一个10像素的空白区，这相当简单。以下规则会为段落指定一个银色背景和一个10像素的外边距，如图8-6所示：

```
p {background-color: silver; margin: 10px;}
```

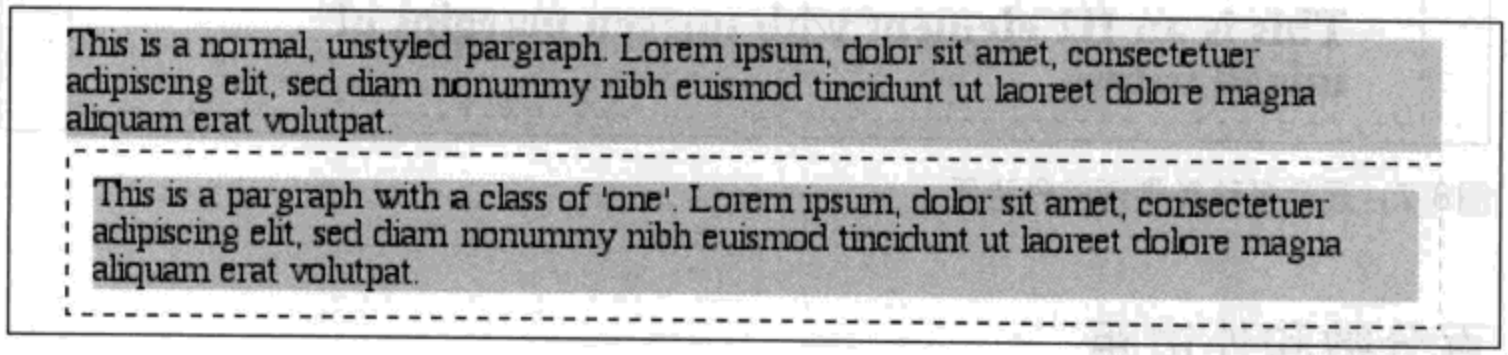

图8-6：段落对照

（同样地，增加背景色是为了帮助显示内容区，这里的虚线只是用于说明，在浏览器中并不真正出现。）如图8-6所示，为内容区的各边增加了10像素的空白。这个结果有些类似于HTML中使用hspace和vspace属性。实际上，可以使用margin设置一个图像周围的额外空间。假设希望所有图像周围都有1em的空白：

```
img {margin: 1em;}
```

这就大功告成了。

有时，你可能希望一个元素各边上的空白不同。这也很简单。如果希望所有h1元素的上外边距为10像素，右外边距为20像素，下外边距为15像素，左外边距为5像素，只需以下规则：

```
h1 {margin: 10px 20px 15px 5px;}
```

这些值的顺序很重要，应当遵循以下模式：

```
margin: top right bottom left
```

要记住这个模式有个好办法，就是记住这4个值是从上（top）开始围着元素顺时针旋转的。一定要以这个顺序应用值，所以要得到你想要的效果，就必须正确地安排值的顺序。

注意： 还有一种很容易的方法可以记住应当以什么顺序声明各边的外边距，这就是要记住，以正确的顺序设置各边外边距有助于你避免“TRouBLe”，即**TRBL**，这代表“**T**op **R**ight **B**ottom **L**eft”。

也可以混合使用各种类型的长度值。一个规则中并不限制只能使用一种长度类型，如下所示：

```
h2 {margin: 14px 5em 0.1in 3ex;}  /* value variety! */
```

图 8-7 显示了这个声明的效果，这里提供了一点注解。

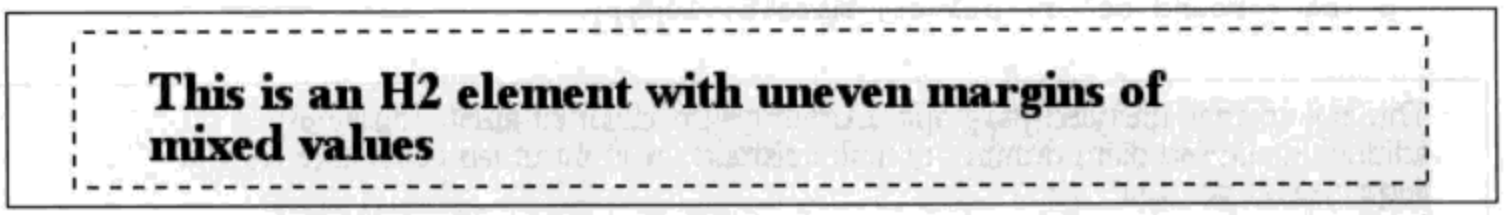

图8-7：混合多种类型值的外边距

百分数和外边距

前面已经提到，可以对元素的外边距设置百分数值。百分数是相对于父元素的 width 计算的，所以如果父元素的 width 以某种方式发生改变，百分数也会改变。例如，假设有以下规则，如图 8-8 所示：

```
p {margin: 10%;}
<div style="width: 200px; border: 1px dotted;">
<p>This paragraph is contained within a DIV that has a width of 200
pixels,
so its margin will be 10% of the width of the paragraph's parent (the
DIV).
Given the declared width of 200 pixels, the margin will be 20 pixels on
all sides.</p>
</div>
<div style="width: 100px;  border: 1px dotted;">
<p>This paragraph is contained within a DIV with a width of 100 pixels,
so its margin will still be 10% of the width of the paragraph's parent.
There will, therefore, be half as much margin on this paragraph as that
on the first paragraph.</p>
</div>
```

与之对照，再考虑另一种情况，没有为元素声明width。在这种情况下，元素框的总宽度（包括外边距）取决于父元素的的width。这有可能得到“流式”页面，即元素的外边距会扩大或缩小以适应父元素（或显示画布）的实际大小。如果对一个文档设置样式，使其元素使用百分数外边距，当用户修改浏览器窗口的宽度时，外边距会随之扩大或缩小。具体的设计选择取决于你。

你可能已经注意到，图 8-8 中的段落有些奇怪。不仅其左右两边的外边距会根据父元素的宽度改变，上下外边距也会随之改变。在CSS中这是期望行为。再来看属性定义，可

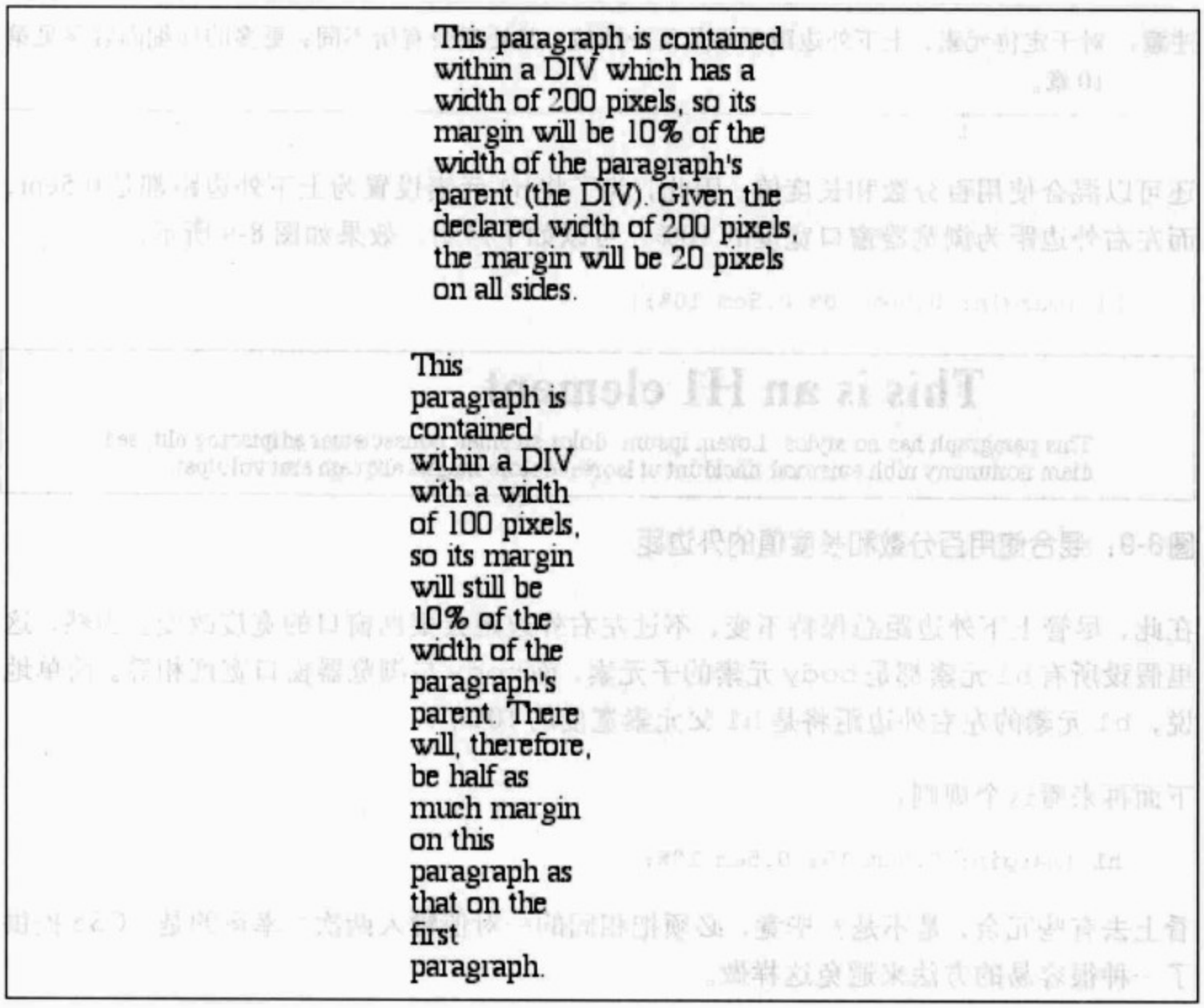

图8-8：父元素width与百分数

以看到，百分数值定义为相对于父元素的*width*。这不仅应用于左右外边距，也应用于上下外边距。因此，给定以下样式和标记，段落的上外边距将是50px：

```
div p {margin-top: 10%;}

<div style="width: 500px;">
<p>This is a paragraph, and its top margin is 10% the width of its parent
element.</p>
</div>
```

如果div的width改变，段落的上外边距也会改变。看上去有些奇怪，是不是？这样来考虑，我们认为，正常流中的大多数元素都会足够高以包含其后代元素（包括其外边距）。如果一个元素的上下外边距是父元素的height的一个百分数，就可能导致一个无限循环，父元素的height会增加，以适应后代元素上下外边距的增加，而相应地，上下外边距又必须增加，以适应新的父元素height，如此继续。规范的作者没有简单地忽略上下外边距百分数，而是决定让它与父元素的width相关，不会根据其后代元素的width而改变。

注意：对于定位元素，上下外边距如果是百分数值，其处理会有所不同；更多的详细内容参见第10章。

还可以混合使用百分数和长度值。因此，为了将h1元素设置为上下外边距都是0.5em，而左右外边距为浏览器窗口宽度的10%，可以如下声明，效果如图8-9所示。

```
h1 {margin: 0.5em 10% 0.5em 10%;}
```

图8-9：混合使用百分数和长度值的外边距

在此，尽管上下外边距总保持不变，不过左右外边距会根据窗口的宽度改变。当然，这里假设所有h1元素都是body元素的子元素，而body与浏览器窗口宽度相等。简单地说，h1元素的左右外边距将是h1父元素宽度的10%。

下面再来看这个规则：

```
h1 {margin: 0.5em 10% 0.5em 10%;}
```

看上去有些冗余，是不是？毕竟，必须把相同的一对值键入两次。幸运的是，CSS提供了一种很容易的方法来避免这样做。

值复制

有时，为外边距输入的值会有些重复：

```
p {margin: 0.25em 1em 0.25em 1em;}
```

不过，不必像这样重复地键入这对数字。不需要用上面的规则，你可以试试以下规则：

```
p {margin: 0.25em 1em;}
```

这两个值足以取代前面的4个值。但这是怎么做到的呢？CSS定义了一些规则，允许为外边距指定少于4个值。规则如下：

- 如果缺少左外边距的值，则使用右外边距的值。
- 如果缺少下外边距的值，则使用上外边距的值。
- 如果缺少右外边距的值，则使用上外边距的值。

如果需要一种更直观的方法来了解这一点，可以看看图 8-10。

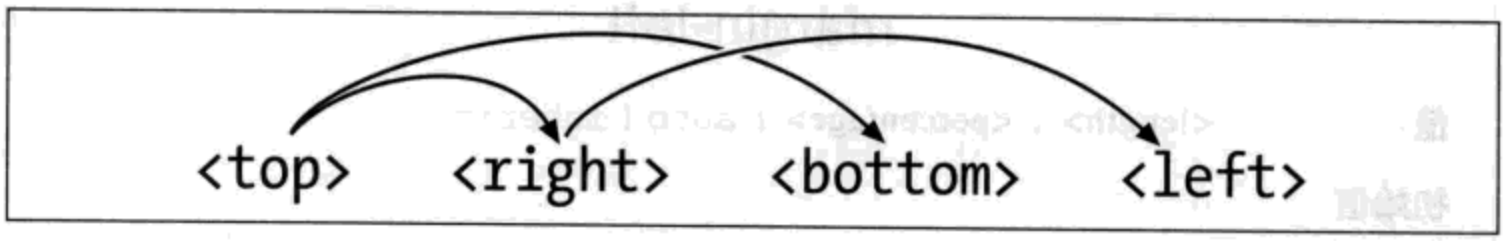

图8-10：值复制模式

换句话说，如果为外边距指定了 3 个值，则第 4 个值（即左外边距）会从第 2 个值（右外边距）复制得到。如果给定了两个值，第 4 个值会从第 2 个值复制得到，第 3 个值（下外边距）会从第 1 个值（上外边距）复制得到。最后一种情况，如果只给定了一个值，那么其他 3 个外边距都由这个值（上外边距）复制得到。

利用这种简单的机制，创作人员只需指定必要的值，而不必全部都应用 4 个值，如下所示：

```
h1 {margin: 0.25em 0 0.5em;} /* same as '0.25em 0 0.5em 0' */
h2 {margin: 0.15em 0.2em;}  /* same as '0.15em 0.2em 0.15em 0.2em' */
p {margin: 0.5em 10px;}     /* same as '0.5em 10px 0.5em 10px' */
p.close {margin: 0.1em;}  /* same as '0.1em 0.1em 0.1em 0.1em' */
```

这种方法有一个小缺点，你最后肯定会遇到这个问题。假设想将h1元素的上外边距和左外边距设置为10像素，下外边距和右外边距设置为20像素。在这种情况下，必须写作：

```
h1 {margin: 10px 20px 20px 10px;} /* can't be any shorter */
```

这样才能得到你想要的效果，但是要把这些全部键入需要一点的时间。遗憾的是，在这种情况下，所需值的个数没有办法更少了。再来看另一个例子，在此你希望除了左外边距以外所有其他外边距都是 auto（左外边距为 3em）：

```
h2 {margin: auto auto auto 3em;}
```

同样地，这样才能得到你想要的效果。问题在于，键入这些 auto 有些麻烦。你想做的只是控制元素单边上的外边距，这就引入了下一个话题。

单边外边距属性

幸运的是，确实有一种办法可以为元素单边上的外边距设置值。假设你只想把h2元素的左外边距设置为 3em。不必使用 margin（这需要键入很多 auto），而是可以采用以下方法：

```
h2 {margin-left: 3em;}
```

margin-left是专门用来设置元素框各边外边距的4个属性之一。它们的名字一目了然。

margin-top、margin-right、margin-bottom、margin-left

值：	<length> | <percentage> | auto | inherit
初始值：	0
应用于：	所有元素
继承性：	无
百分数：	相对于包含块的 width
计算值：	对于百分数，根据指定确定；否则，为绝对长度

使用其中任何一个属性将只设置该边上的外边距，而不会直接影响所有其他外边距。

一个规则中可以使用多个这种单边属性，例如：

```
h2 {margin-left: 3em; margin-bottom: 2em;
  margin-right: 0; margin-top: 0;
  background: silver;}
```

在图 8-11 中可以看到，已经按你的意愿设置了外边距。当然，对于这种情况，使用 margin 可能更容易一些：

```
h2 {margin: 0 0 2em 3em;}
```

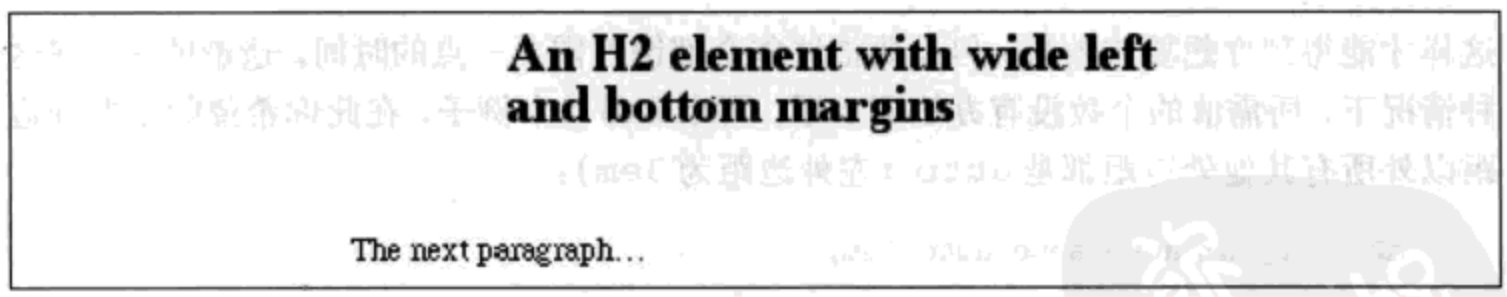

图8-11：多个单边外边距

不论使用上述单边属性还是使用margin，得到的结果都一样。一般地，如果想为多个边设置外边距，使用margin会更容易一些。不过，从文档显示的角度看，实际上使用哪种方法都不重要，所以应该选择对你来说更容易的一种方法。

负外边距和合并外边距

第7章曾经讨论过，可以为元素设置负外边距。这会导致元素框超出其父元素，或者与其他元素重叠，但并不违反框模型。考虑以下规则，如图 8-12 所示：

```
div {border: 1px dotted gray; margin: 1em;}
p {margin: 1em; border: 1px dashed silver;}
p.one {margin:  0 -1em;}
p.two {margin: -1em 0;}
```

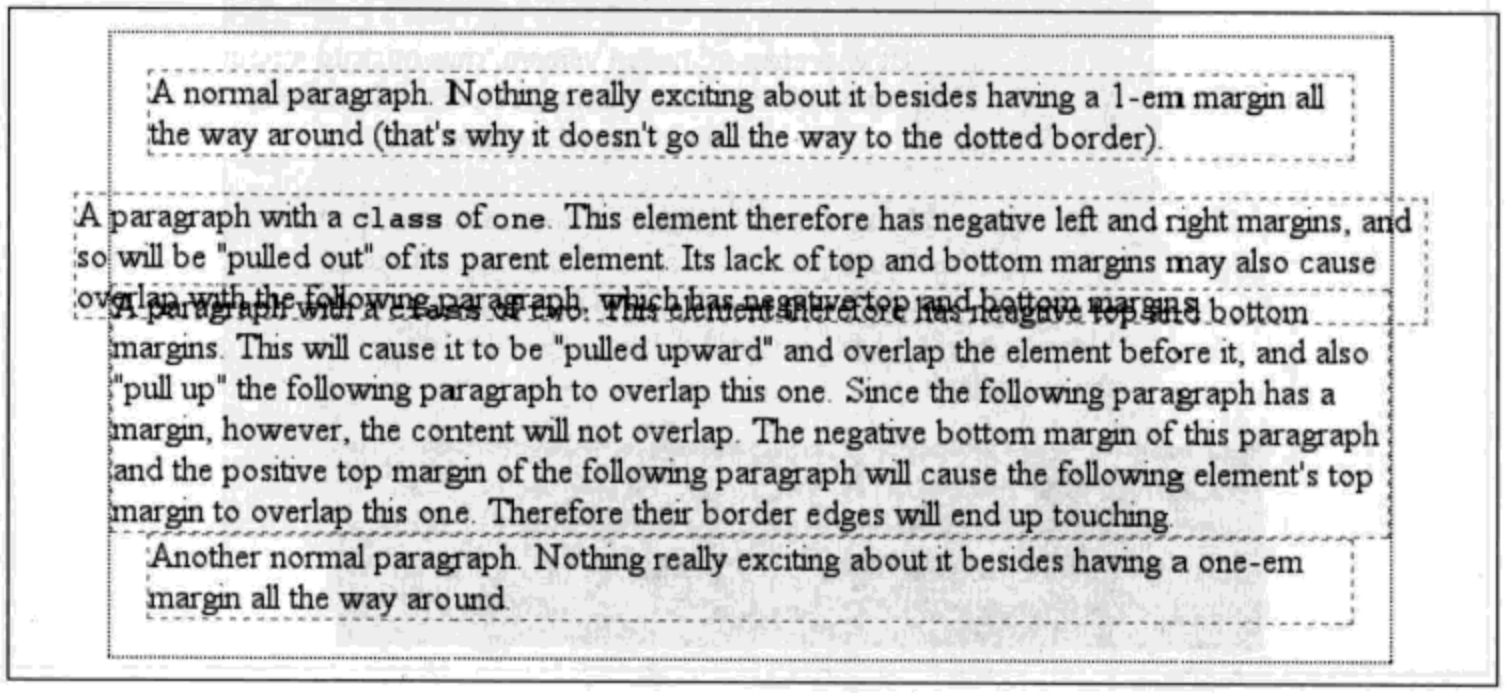

图8-12：负外边距的实际使用

在第一个例子中，根据数学计算，这个段落的width计算值加上其左右外边距刚好等于父元素div的width。所以，这个段落最后将比其父元素宽2em，但从数学角度讲实际上并没有更“宽”。在第二个例子中，负的上下外边距实际上增加了元素的height计算值，将其上下外边界向外移，这就与它之前和之后的段落发生了重叠。

结合使用正负外边距会很有用。例如，可以创造性地结合正负外边距，使一个段落“超出”其父元素，或者可以创建一种蒙德里安风格的效果，有多个重叠或随机放置的框，如图8-13所示。

```
div {background: silver; border: 1px solid;}
p {margin: 1em;}
p.punch {background: white; margin: 1em -1px 1em 25%;
  border: 1px solid; border-right: none; text-align: center;}
p.mond {background: #333; color: white; margin: 1em 3em -3em -3em;}
```

由于“mond”段落的下外边距为负，其父元素的底端会向上拉，使得段落超出其父元素的底端。

说到上下外边距，还要记住重要的一点，正常流中垂直相邻外边距会合并，这个内容在上一章已经介绍过。外边距合并在设置了样式的每一个文档中都在起作用。例如，以下是一个简单的规则：

```
p {margin: 15px 0;}
```

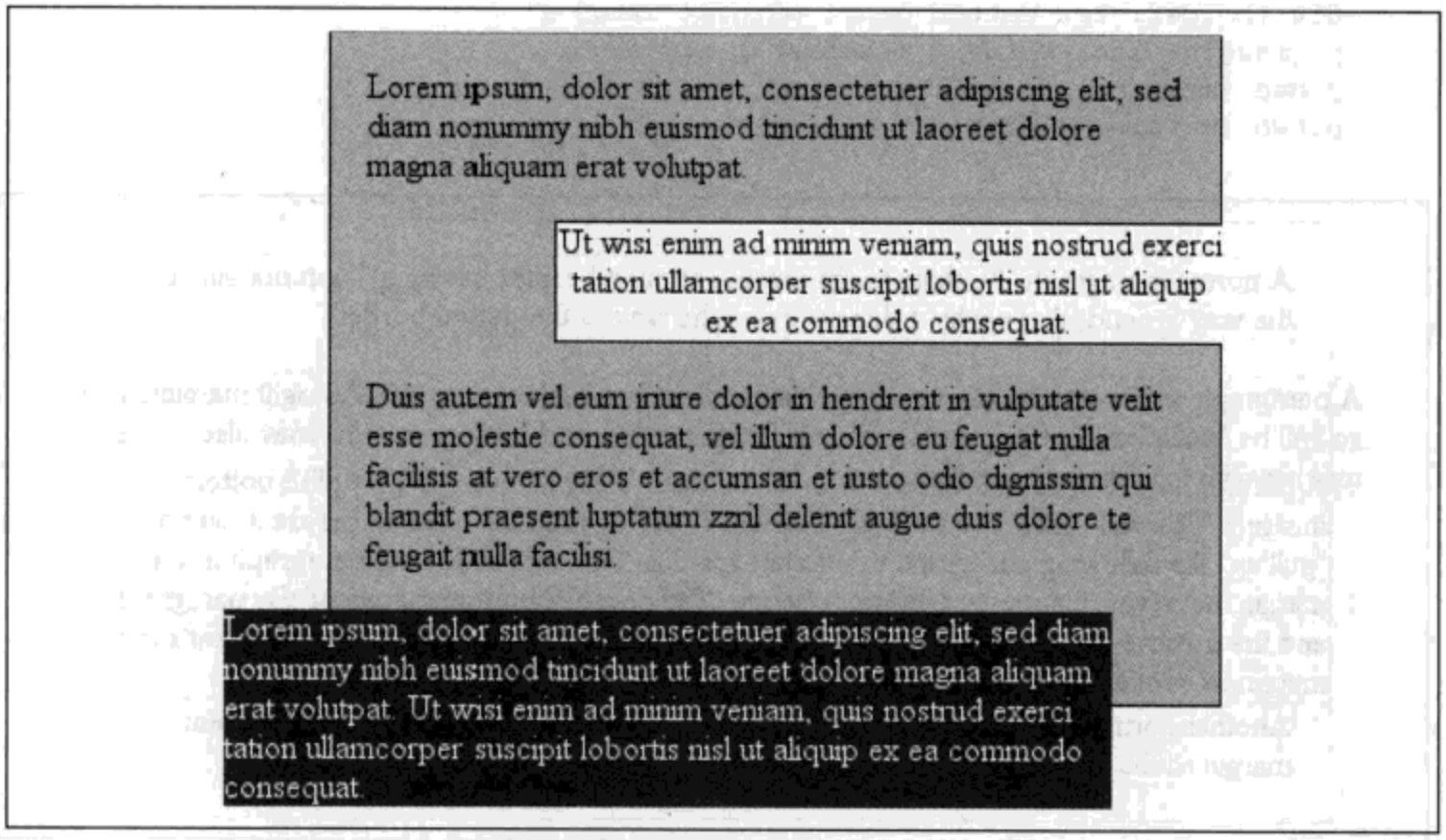

图8-13：超出父元素

这会导致一个段落跟在另一个段落后，二者之间有15像素的“外边距空间”。如果外边距没有合并，那么这两个相邻段落之间应当有30像素的空间，但是创作人员不希望这样。

不过，这确实说明设置外边距时必须小心。你很有可能想去掉标题和下一段之间的空间。由于HTML文档中的段落都有一个上外边距，如果只是将标题的下外边距设置为0是不够的，还必须去掉段落的上外边距。利用CSS2的相邻兄弟选择器很容易做到这一点：

```
h2 {margin-bottom: 0;}
h2 + p {margin-top: 0;}
```

遗憾的是，浏览器对相邻兄弟选择器的支持还很有限（在写作本书时），大多数用户都会看到标题与其下一段之间有1em的间隔。不使用CSS2选择器也可以得到所要的效果，不过要麻烦一些：

```
h2 {margin-bottom: 0;}
p {margin: 0 0 1em;}
```

这会去除所有段落的上外边距，不过由于段落都还有一个1em的下外边距，所以还会保留所要的段间间隔，如图8-14所示。

这样能正常工作，因为段落之间的1em间隔是外边距合并的结果。因此，如果去掉其中一个外边距（在这里就是上外边距），视觉效果与保留该外边距的效果是一样的。

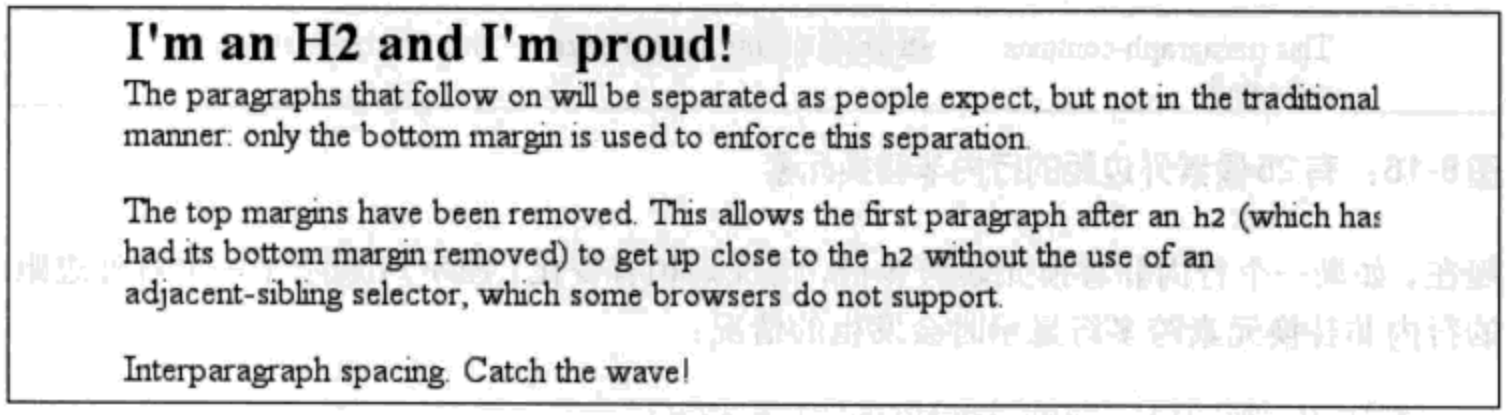

图8-14：聪明地设置外边距

外边距和行内元素

外边距还可以应用到行内元素，不过效果稍有不同。假设你想在重点强调的文本上设置上下外边距：

```
strong {margin-top: 25px; margin-bottom: 50px;}
```

这在规范中是允许的，不过由于在向一个行内非替换元素应用外边距，它对行高没有任何影响。由于外边距实际上是透明的，所以这个声明没有任何视觉效果。其原因就在于行内非替换元素的外边距不会改变一个元素的行高。

注意：对于只包含文本的行，能改变行间距离的属性只有`line-height`、`font-size`和`vertical-align`，见第7章的介绍。

这只对行内非替换元素的上下边成立，左右两边则是另一回事。先来考虑一个简单的情况，将一个小的行内非替换元素放在一行上。在此，如果设置了左外边距或右外边距值，左外边距或右外边距则是可见的，从图8-15可以更清楚地看到：

```
strong {margin-left: 25px; background: silver;}
```

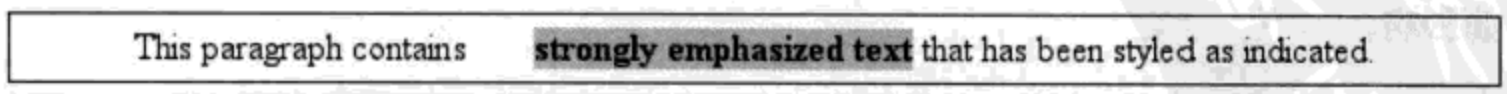

图8-15：有左外边距的行内非替换元素

注意，行内非替换元素之前的字的结尾与行内元素背景的边界之间有额外的一个空间。如果愿意，可以在行内元素的两端都增加这个额外的空间：

```
strong {margin: 25px; background: silver;}
```

可以看到，图8-16在行内元素的左右两边都显示了额外的一点空间，但它上面和下面则没有。

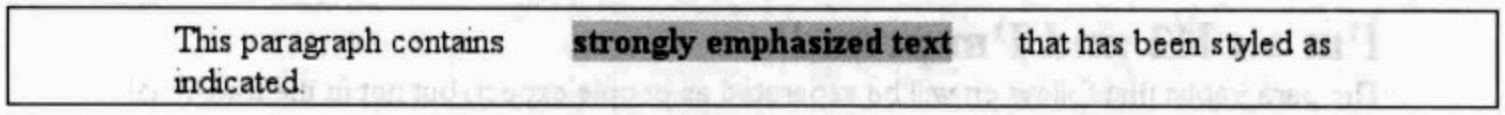

图8-16：有25像素外边距的行内非替换元素

现在，如果一个行内非替换元素跨多行，情况则稍有变化。图8-17展示了一个有外边距的行内非替换元素跨多行显示时会发生的情况：

```
strong {margin: 25px; background: silver;}
```

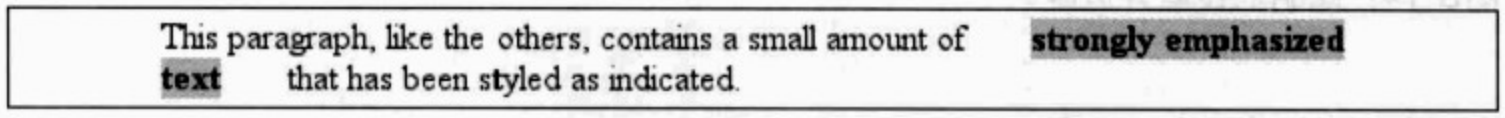

图8-17：有25像素外边距的行内非替换元素跨两个文本行显示

左外边距应用到这个元素的开始处，右外边距应用到该元素的末尾。外边距没有应用到各行的左右两边。另外还可以看到，如果不是有外边距，行可能会在“text”后断开，而不是在“strongly emphasized”之后断开。外边距只是通过改变元素内容在行中的起点来影响换行。

如果向行内非替换元素应用负外边距，情况会更有意思。元素的上下外边距不受影响，行高也不受影响，不过元素的左右两端可能与其他内容重叠，如图8-18所示。

```
strong {margin: -25px; background: silver;}
```

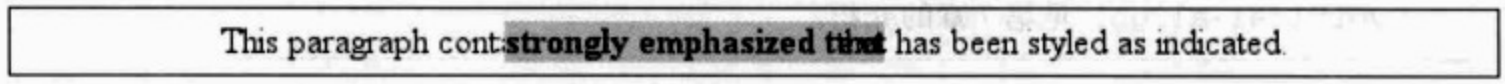

图8-18：有负外边距的行内非替换元素

替换元素又有所不同：为替换元素设置的外边距确实会影响行高，可能会使行高增加或减少，这取决于上下外边距的值。行内替换元素的左右外边距与非替换元素的左右外边距的作用一样。图8-19显示了对行内替换元素设置外边距时，对布局产生的一系列不同的影响。

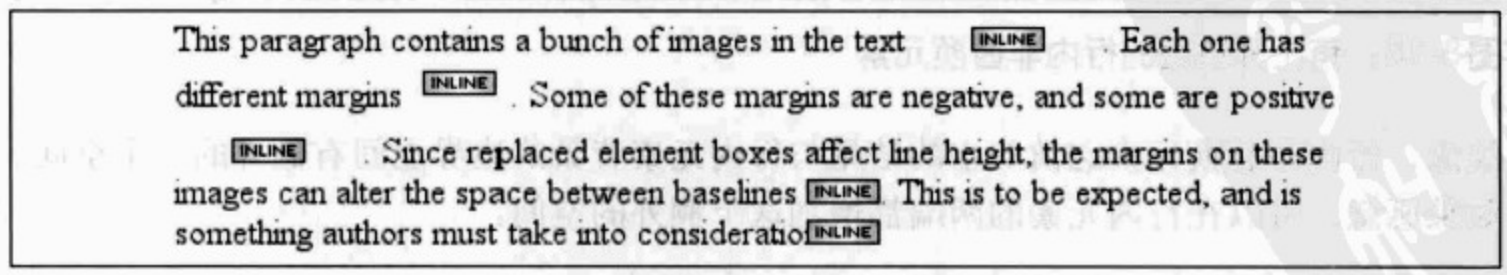

图8-19：有不同外边距值的行内替换元素

外边距历史问题

外边距很有用，但同时也存在很多问题——毫不奇怪，这些问题都围绕着Netscape Navigator 4.x（NN4.x）。

第一个问题是，Navigator 4.x会把外边距规则指定的外边距增加到其默认外边距，而不是用其取代默认值。例如，请考虑：

```
h1 {margin-bottom: 0;}
p {margin-top: 0;}
```

NN4.x 显示元素时其间通常有空白，因为它将0增加到其自己的默认外边距。如果想覆盖这个空白，可以使用负外边距，如将段落的上外边距设置为 -1em。这种解决方法的问题是缺乏通用性。如此一来，CSS 兼容的浏览器就会使文本重叠，因为这些浏览器会替换段落的上外边距（而不是使之相加）。

遗憾的是，还有更糟糕的问题。如果对行内元素应用外边距，布局将或多或少地过于分散。NN4.x认为所有元素（不论是否是行内元素）的外边距都相对于浏览器窗口的左边界。这绝对是错误的。遗憾的是，如果你有很多使用NN4.x的访问者，在行内元素上使用外边距就很危险，决不能等闲视之。好在对NN4.x隐藏CSS很容易，从而可以对文档设置样式，而不必担心NN4.x破坏样式（当然，这里假设你仍然关心页面在NN4.x 中的外观）。

边框

元素外边距内就是元素的边框（border）。元素的边框就是围绕元素内容和内边距的一条或多条线。因此，元素的背景会在外边框边界处停止，因为背景不会延伸到外边距以内，而边框就在外边距内部。

每个边框都有 3 个方面：其宽度或粗细、其样式或外观，以及其颜色。边框宽度的默认值为 medium，这个值没有明确定义，不过通常是 2 个像素。尽管如此，你不一定能看到边框，原因是边框的默认样式为 none，这样一来，就不会有边框了。如果一个边框没有样式，它就不必存在（不存在边框还会重置 width 值，不过稍后再讨论这个问题）。

最后，默认的边框颜色是元素本身的前景色。如果没有为边框声明颜色，它将与元素的文本颜色相同。另一方面，如果一个元素没有任何文本，假设它有一个表，其中只包含图像，那么该表的边框颜色就是其父元素的文本颜色（因为color可以继承）。这个父元素很可能是 body、div 或另一个 table。因此，如果一个 table 有边框，而且其父元素是 body，给定以下规则：

```
body {color: purple;}
```

默认地，table外围的边框将是紫色（假设用户代理没有为表设置颜色）。当然，要让边框显示，必须先做一点工作。

边框和背景

CSS规范清楚地指出元素的背景会延伸到边框边界之外，因为规范中提到，边框绘制在“元素的背景之上”。这很重要，因为有些边框是“间断的”（例如，点线边框或虚线框），元素的背景应当出现在边框的可见部分之间。

发布CSS2时，它指出背景只延伸到内边距，而不是边框。后来又对此做了更正，CSS2.1明确指出元素的背景是内容、内边距和边框区的背景。大多数浏览器都遵循CSS2.1定义，不过一些较老的浏览器可能会有不同的表现。背景颜色问题将在第9章更详细地讨论。

有样式的边框

先来讨论边框样式，这是边框最重要的一个方面，并不只是因为样式控制着边框的显示（当然，样式确实控制着边框的显示），而是因为如果没有样式，将根本没有边框。

CSS为属性border-style定义了10个不同的非inherit样式，包括默认值none。这些样式如图8-20中所示。

样式值hidden等价于none，不过应用于表时除外，对于表，hidden用于解决边框冲突（更多详细信息见第11章）。

border-style

值：	[none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset]{1,4} | inherit
初始值：	对简写属性没有定义
应用于：	所有元素
继承性：	无
计算值：	见各个属性（border-top-style等）
说明：	根据CSS1和CSS2，HTML用户代理只需支持solid和none；其余的值（除hidden外）可能被解释为solid；这个限制在CSS2.1中被去除

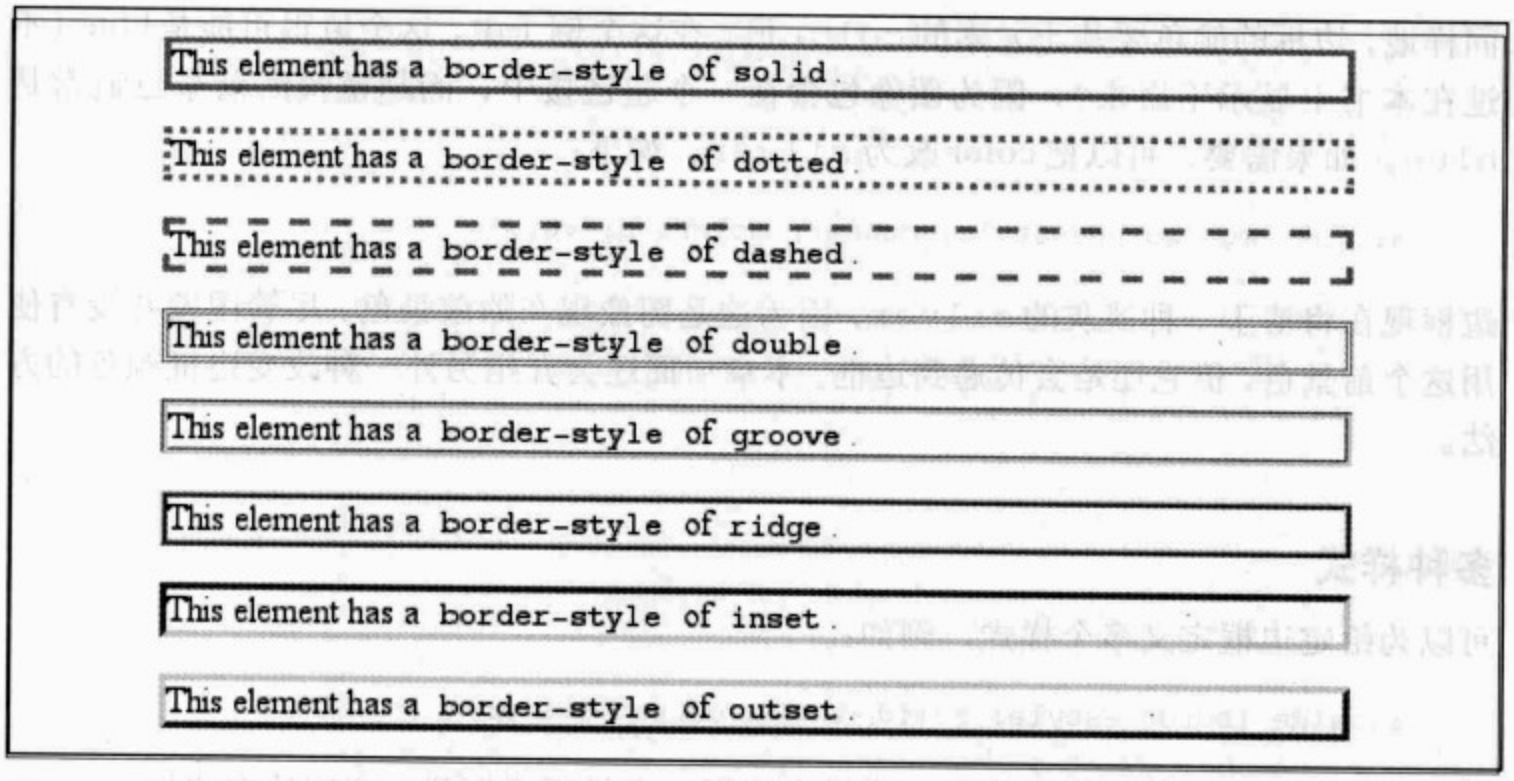

图8-20：边框样式

最不可预测的边框样式是double。它定义为两条线的宽度再加上这两条线之间的空间等于border-width值（这将在下一节讨论）。不过，CSS规范并没有说其中一条线是否比另一条线粗，或者两条线是否应该一样粗，也没有指出线之间的空间是否应当比线粗。所有这些都由用户代理决定，创作人员对这个决定没有任何影响。

图8-20所示的所有边框color值都是gray，这就能更容易地看出视觉效果。边框样式的外观总是以某种方式基于边框的颜色，虽然具体的方式可能随用户代理的不同而有所不同。例如，图8-21展示了显示一个inset边框的两种不同方法。

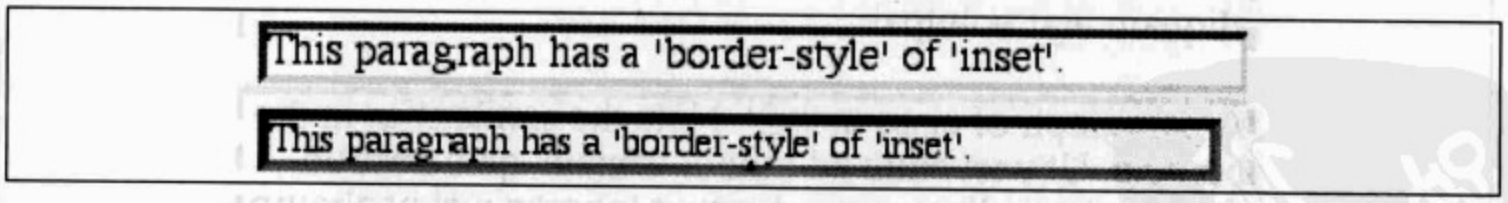

图8-21：显示inset边框的两种有效方法

假设你想为包含在未访问超链接内部的图像定义一个边框样式。可以将边框设置为outset，使之看上去像是“凸起按钮”，如图8-22所示：

```
a:link img {border-style: outset;}
```

图8-22：向包含在超链接中的图像应用outset边框

同样地，边框的颜色要基于元素的color值。在这个例子中，这个值很可能是blue（不过在本书上显示不出来），因为图像包含在一个超链接中，而超链接的前景色通常是blue。如果需要，可以把color改为silver，如下：

```
a:link img {border-style: outset; color: silver;}
```

边框现在将基于一种淡灰的silver，因为这是图像现在的前景色，尽管图像并没有使用这个前景色，但它还是会传递到边框。本章后面还会介绍另外一种改变边框颜色的方法。

多种样式

可以为给定边框定义多个样式，例如：

```
p.aside {border-style: solid dashed dotted solid;}
```

其结果是段落有一个实线上边框、虚线右边框、点线下边框和一个实线左边框。

我们又看到了这里的值采用了top-right-bottom-left的顺序，讨论用多个值设置不同外边距时也见过这个顺序。关于外边距和内边距值复制的规则同样适用于边框样式。因此，以下两个规则应该有相同的效果，如图8-23所示：

```
p.new1 {border-style: solid dashed none;}
p.new2 {border-style: solid dashed none dashed;}
```

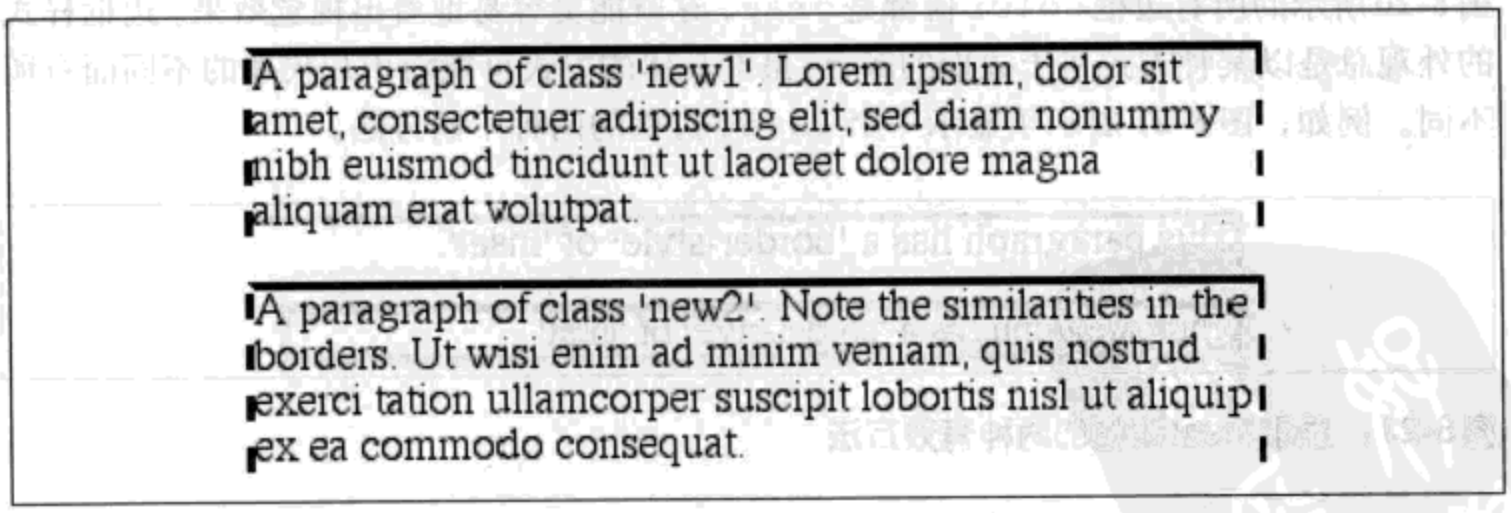

图8-23：相等的样式规则

单边样式

有时你可能只想为元素框的一边设置边框样式，而不是设置所有4个边的边框样式。这就要用到单边边框样式属性了。

border-top-style、border-right-style、border-bottom-style、border-left-style

值：	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset | inherit
初始值：	none
应用于：	所有元素
继承性：	无
计算值：	根据指定确定

单边边框样式属性的含义不言自明。例如，如果你想改变下边框的样式，可以使用border-bottom-style。

border与单边属性结合使用的情况很常见。假设你想在一个标题的三个边上设置实线边框，但是没有左边框，如图 8-24 所示。

H1 border fun!

图8-24：去掉左边框

为此有两种等价的方法：

```
h1 {border-style: solid solid solid none;}
  /* the method above is the same as the one below */
h1 {border-style: solid; border-left-style: none;}
```

要记住重要的一点，如果你要使用第二种方法，必须把单边属性放在简写属性之后，对于简写属性通常都是这样。这是因为如果声明border-style: solid，实际上是在声明border-style: solid solid solid solid。倘若把border-style-left: none放在border-style声明之前，简写属性的值就会覆盖单边值none。

到目前为止，你可能已经注意到，边框例子使用的都是相同宽度的边框。这是因为你没有定义 width，所以它默认为某个值。接下来，我们将介绍这个默认值以及更多内容。

边框宽度

一旦为边框指定一个样式，下一步就是使用border-width为它指定一个宽度。

border-width	
值：	[thin|medium | thick | <length>]{1,4} | inherit
初始值：	对简写属性未定义
应用于：	所有元素
继承性：	无
计算值：	见各个属性（border-top-style等）

还可以使用它的某个相关（单边）属性。

border-top-width、border-right-width、border-bottom-width、border-left-width	
值：	thin | medium | thick | <length> | inherit
初始值：	medium
应用于：	所有元素
继承性：	无
计算值：	绝对长度；如果边框的样式为none或hidden，则为0

当然，这些属性用于设置某个特定边框边的宽度，这与单边外边距属性类似。

注意：在CSS2.1中，边框宽度还不能指定为百分数值，这确实有待改进。

为边框指定宽度有4种方法：可以指定一个长度值，如4px或0.1em；或者使用3个关键字之一。这3个关键字分别是thin、medium（默认值）和thick。这些关键字不一定对应某个特定的宽度，它们只是相对定义。根据规范，thick总是比medium宽，而medium则总是比thin宽。

不过，具体的宽度没有定义，所以一个用户代理可能将thin、medium和thick分别设置为等于5px、3px和2px，而另一个用户代理则分别设置为3px、2px和1px。不论用户代理对各个关键字具体使用多大的宽度，不管边框出现在哪里，各关键字对应的宽度在文档中总保持不变。所以，如果medium等于2px，宽度为medium的边框总是2像素宽，而不论边框在h1元素外还是在p元素外。图8-25展示了处理这3个关键字的一种方法，还显示了它们彼此之间有何关系，以及与所包围的内容有何关系。

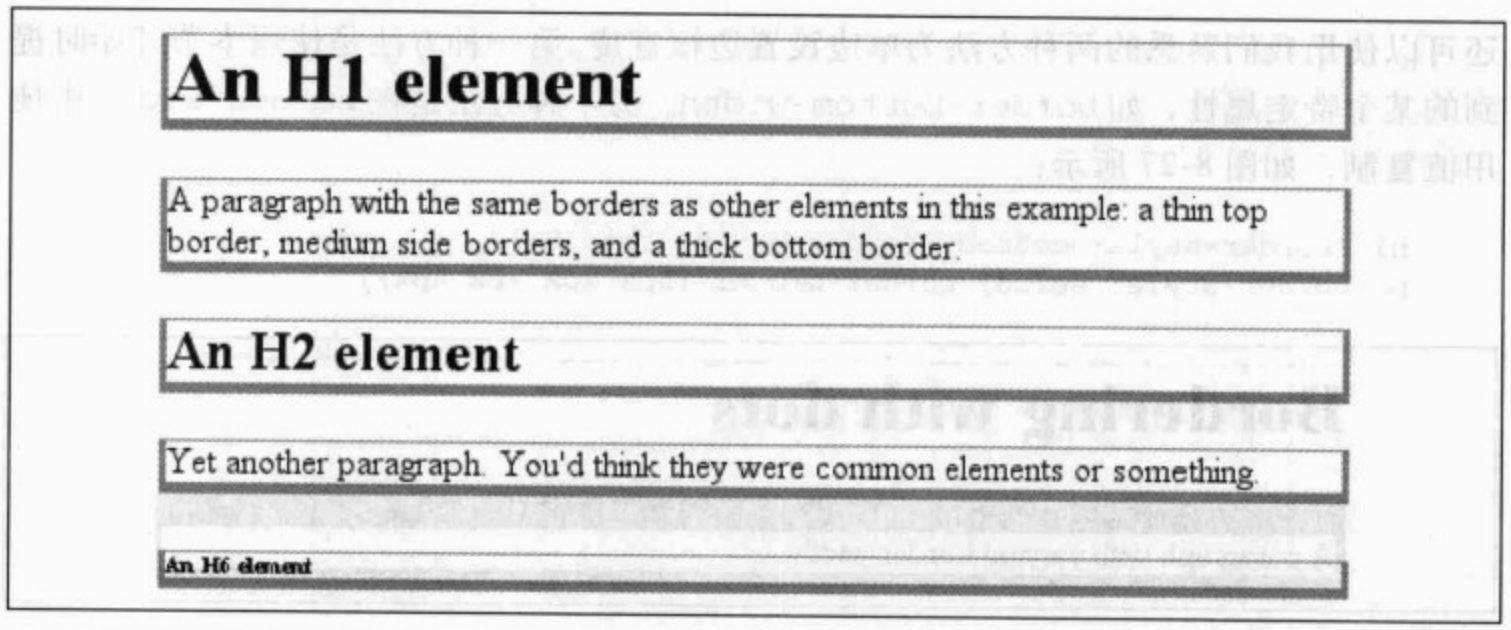

图8-25：border-width关键字之间的关系

假设一个段落设置了外边距、背景颜色以及一个边框样式：

```
p {margin: 5px; background-color: silver;
   border-style: solid;}
```

默认地，边框的宽度为medium。这很容易改变：

```
p {margin: 5px; background-color: silver;
   border-style: solid; border-width: thick;}
```

当然，边框宽度可以设置得很极端，如设置50像素的边框，如图8-26所示：

```
p {margin: 5px; background-color: silver;
   border-style: solid; border-width: 50px;}
```

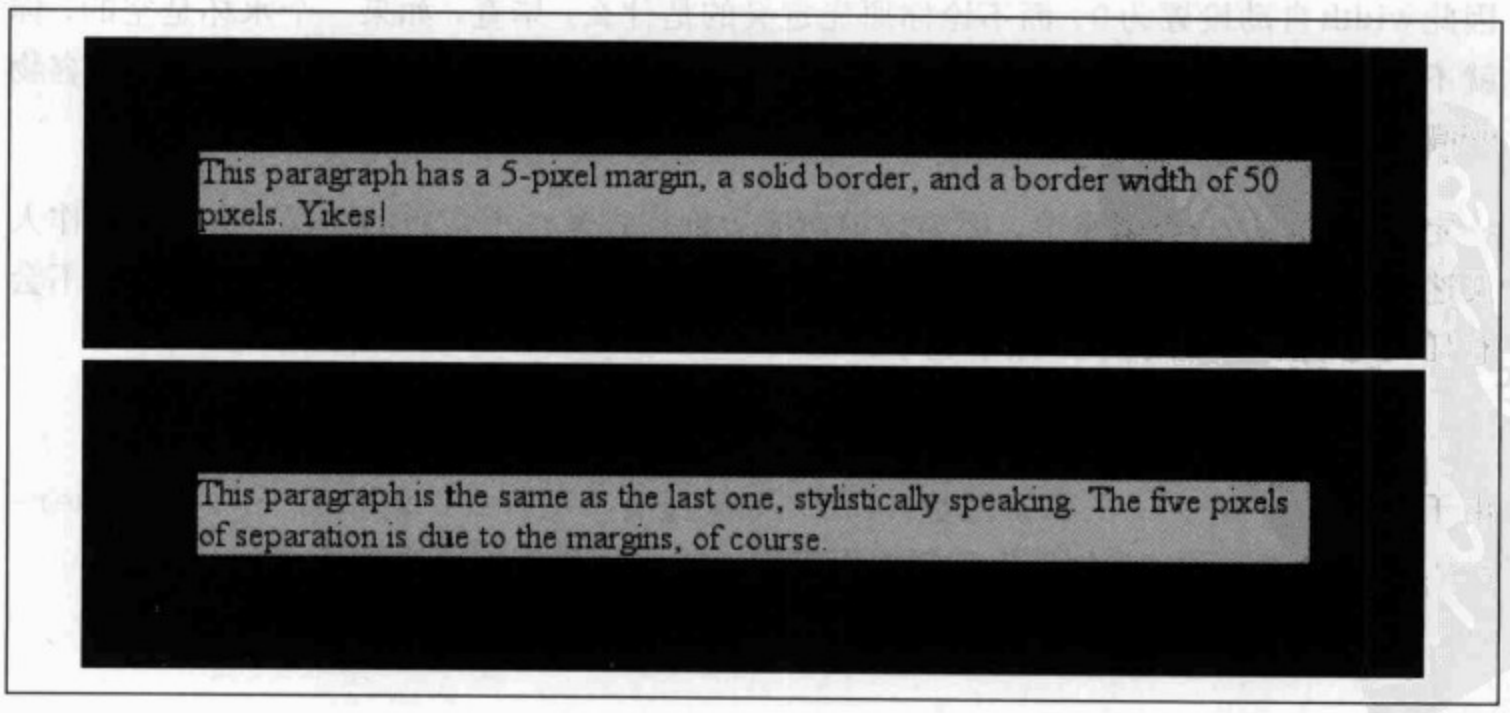

图8-26：实在太宽的边框

还可以使用我们熟悉的两种方法为单边设置边框宽度。第一种方法是使用本节开始时提到的某个特定属性，如border-bottom-width。另一种方法是在border-width中使用值复制，如图8-27所示：

```
h1 {border-style: dotted; border-width: thin 0;}
p {border-style: solid; border-width: 15px 2px 7px 4px;}
```

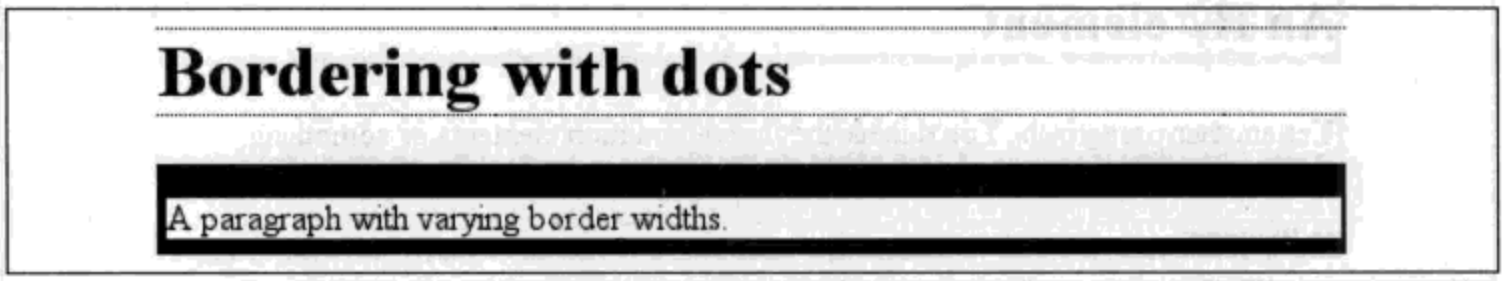

图8-27：值复制和不均匀的边框宽度

根本没有边框

迄今为止，我们只谈到了如何使用可见的边框样式，如solid或outset。下面考虑在将border-style设置为none时会出现什么情况：

```
p {margin: 5px; border-style: none; border-width: 20px;}
```

尽管边框的宽度为20px，但style设置为none。在这种情况下，不仅边框的样式没有了，其宽度也会变成0。边框消失了。为什么呢？

如果你还记得，本章是这样说的：样式为none的边框不存在。这些词是经过仔细斟酌才选定的，因为这有助于解释到底发生了什么。由于边框不存在，所以它不可能有宽度，因此width自动设置为0，而不论你原先定义的是什么。毕竟，如果一个水杯是空的，你就不能把它描述为装着半杯“没有东西”。杯子里确实有内容物时才能讨论杯子内容物的高度。同样，只有当边框存在时才能讨论边框的宽度。

一定要记住，这一点很重要，因为忘记声明边框样式是一个常犯的错误。这会让创作人员迷惑不已，因为乍看上去样式应该能正确出现。根据以下规则，所有h1元素都不会有任何边框，更不用说20像素宽了：

```
h1 {border-width: 20px;}
```

由于border-style的默认值是none，如果没有声明样式，就相当于声明border-style: none。因此，如果你希望边框出现，就必须声明一个边框样式。

边框颜色

与边框的其他方面相比，设置颜色很简单。CSS使用了一个简单属性border-color，它一次可以接受最多4个颜色值。

border-color

值：	[<color> | transparent]{1,4} | inherit
初始值：	对简写属性未定义
应用于：	所有元素
继承性：	无
计算值：	见单个属性（border-top-color等）

如果值小于4个，值复制就会起作用。所以如果你希望h1元素有细的黑色上下边框，而且有粗的灰色左右边框，另外希望p元素外有中等粗细的灰色边框，就可以用以下标记，其结果见图8-28：

```
h1 {border-style: solid; border-width: thin thick; border-color: black gray;}
p {border-style: solid; border-color: gray;}
```

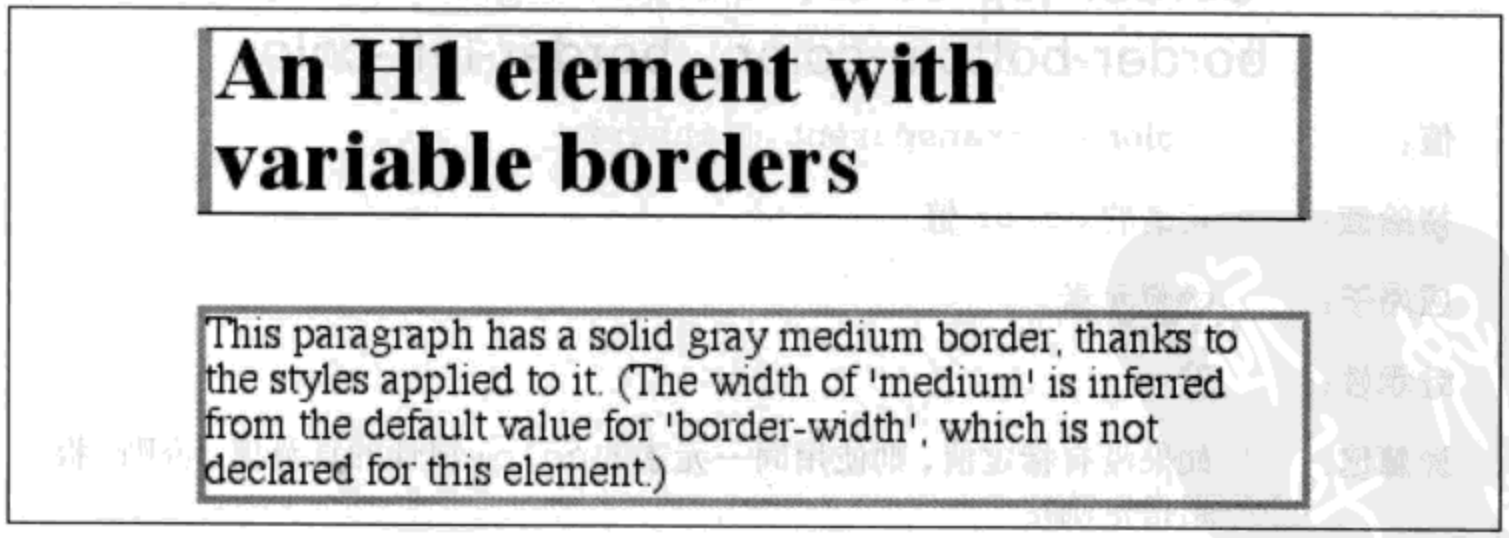

图8-28：边框有很多方面

当然一个颜色值会应用到所有4个边，前例中的段落就是如此。另一方面，如果应用了4个颜色值，那么每边都会有不同的颜色。可以使用任何类型的颜色值，例如可以是命名颜色，也可以是十六进制和RGB值：

```
p {border-style: solid; border-width: thick;
  border-color: black rgb(25%,25%,25%) #808080 silver;}
```

本章前面提到过，如果没有声明颜色，默认颜色则是元素的前景色。因此，以下声明会显示为如图 8-29 所示情景：

```
p.shade1 {border-style: solid; border-width: thick; color: gray;}
p.shade2 {border-style: solid; border-width: thick; color: gray;
   border-color: black;}
```

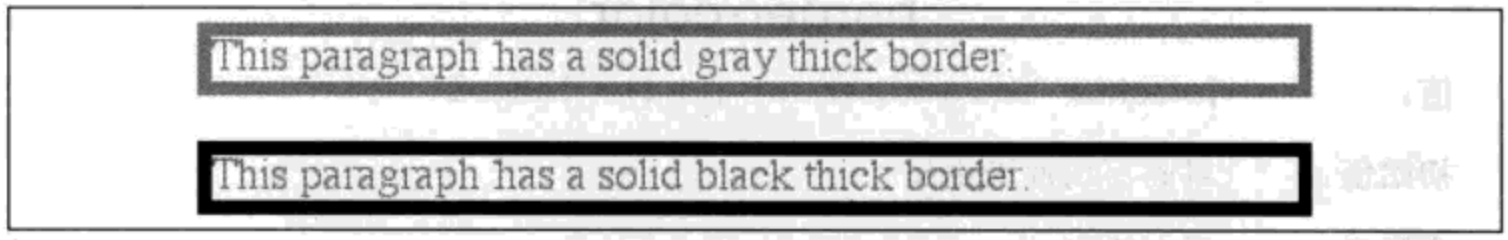

图8-29：根据元素的前景色及border-color属性值确定的边框颜色

其结果是第一段有一个灰色边框，值gray取自段落的前景色。不过，第二段有一个黑色边框，因为这是使用border-color显式指定的颜色。

还有一些单边border-color属性。其原理与单边样式和宽度属性相同。要为标题指定一个实线黑色边框，而且右边框为实线灰色，可以如下指定：

```
h1 {border-style: solid; border-color: black; border-right-color: gray;}
```

border-top-color、border-right-color、border-bottom-color、border-left-color

值：	<color> | transparent | inherit
初始值：	元素的color值
应用于：	所有元素
继承性：	无
计算值：	如果没有指定值，则使用同一元素的color属性的计算值；否则，根据指定确定

透明边框

你应该还记得，如果一个边框没有样式，就没有宽度。不过，有些情况下你可能想创建一个不可见的边框。这就引入了边框颜色值transparent（在CSS2中引入）。这个值用于创建有宽度的不可见边框。

假设你希望包含3个链接的一组链接有边框，默认地这些边框不可见，不过，鼠标停留在链接上时边框要凸起。为此可以让边框在链接处于非悬停状态下透明：

```
a:link, a:visited {border-style: solid; border-width: 5px;
  border-color: transparent;}
a:hover {border-color: gray;}
```

其效果如图8-30所示。

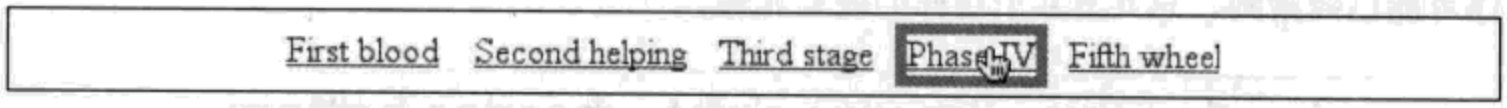

图8-30：使用透明边框

从某种意义上说，利用transparent，使用边框就像是额外的内边距一样；此外还有一个好处，就是能在你需要的时候使其可见。这种透明边框相当于内边距，因为元素的背景会延伸到边框区（假设有可见的背景）。

警告： 在IE7之前，IE/Win没有提供对transparent的支持。在以前的版本中，IE会根据元素的color值来设置边框颜色。

简写边框属性

遗憾的是，简写属性（如border-color和border-style）并不总像你想得那么有用。例如，你可能想在所有h1元素上应用一个粗的灰色实线边框，不过只是底边上有此边框。如果只使用我们目前已经讨论过的属性，应用这样一个边框时会遇到很大困难。以下是两个例子：

```
h1 {border-bottom-width: thick;    /* option #1 */
  border-bottom-style: solid;
  border-bottom-color: gray;}

h1 {border-width: 0 0 thick;       /* option #2 */
  border-style: none none solid;
  border-color: gray;}
```

根据以上键入的规则，这两种做法都很不方便。幸运的是，还有一种更好的解决方法：

```
h1 {border-bottom: thick solid gray;}
```

这只是向底边边框应用值，如图8-31所示，而所有其他边的边框仍为其默认值。由于默认的边框样式为none，所以元素其他三个边上不出现边框。

图8-31：用简写属性设置底边边框

你可能已经猜到，总共有4个这样的简写属性。

border-top、border-right、border-bottom、border-left

值：	[<border-width> || <border-style> || <border-color>] | inherit
初始值：	对简写属性未定义
应用于：	所有元素
继承性：	无
计算值：	见单个属性（border-width等）

可以使用这些属性创建一些复杂的边框，如图8-32所示的边框：

```
h1 {border-left: 3px solid gray;
  border-right: black 0.25em dotted;
  border-top: thick silver inset;
  border-bottom: double rgb(33%,33%,33%) 10px;}
```

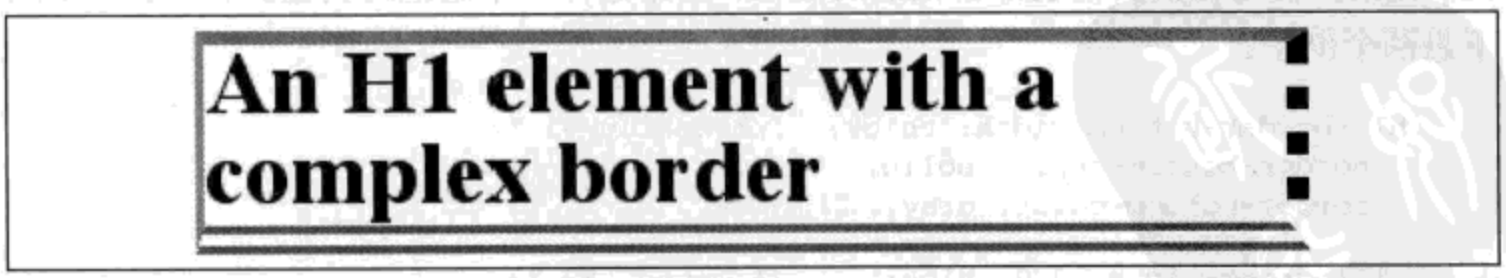

图8-32：非常复杂的边框

可以看到，具体值的顺序并不重要。以下三个规则会得到完全相同的边框效果：

```
h1 {border-bottom: 3px solid gray;}
h2 {border-bottom: solid gray 3px;}
h3 {border-bottom: 3px gray solid;}
```

还可以省略一些值，使用其默认值，如下：

```
h3 {color: gray; border-bottom: 3px solid;}
```

由于没有声明任何边框颜色，因此会应用默认值（元素的前景色）。要记住，如果没有边框样式，默认样式值none会使得边框不复存在。

相反，如果只设置了样式，还是会得到一个边框。假设你只想有一个样式为dashed的上边框，而且希望其宽度仍为默认值medium，颜色与元素本身的文本颜色相同。在这种情况下，所需的就是以下标记（见图8-33）：

```
p.roof {border-top: dashed;}
```

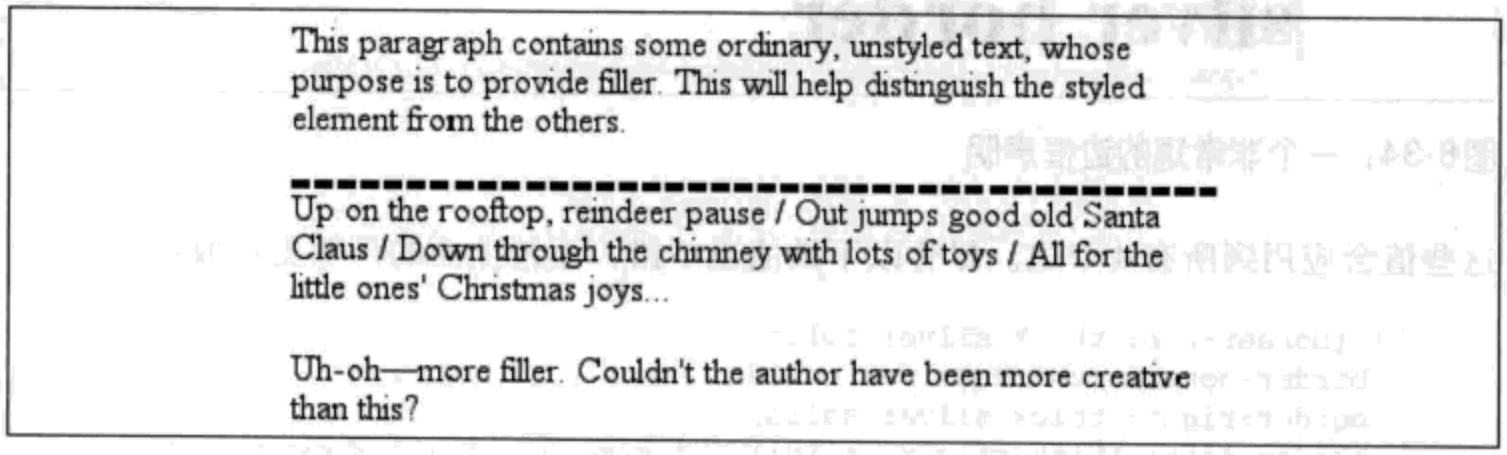

图8-33：元素顶端的虚线边框

另一个要注意的是，由于这些“单边边框”属性只应用到一个特定的边，所以不可能进行值复制——这是没有任何意义的。每种值只能有一个：也就是说，只能有一个宽度值、一个颜色值和一个边框样式。所以不要为同一个类型声明多个值：

```
h3 {border-top: thin thick solid purple;}  /* two width values--WRONG */
```

在这种情况下，整个语句都将无效，用户代理会将其全部忽略。

全局边框

下面我们来看所有边框属性中最简短的简写属性：border。

border

值：	[<border-width> || <border-style> || <border-color>] | inherit
初始值：	根据单个属性
应用于：	所有元素
继承性：	无
计算值：	根据指定确定

这个属性有一个好处，它相当简洁，虽然这种简洁性也带来一些限制。在讨论这些限制之前，先来看border如何工作。如果你希望所有h1元素都有一个粗的银色边框，这很简单。以下声明的结果如图8-34所示：

```
h1 {border: thick silver solid;}
```

图8-34：一个非常短的边框声明

这些值会应用到所有4个边。尽管以下做法也不错，不过上面的声明更可取：

```
h1 {border-top: thick silver solid;
  border-bottom: thick silver solid;
  border-right: thick silver solid;
  border-left: thick silver solid;}  /* same as previous example */
```

使用border的缺点在于，只能定义“全局”的样式、宽度和颜色。换句话说，为border提供的值将完全相等地应用到所有4个边。如果你希望一个元素有不同的边框，则需要使用另外的某个边框属性。当然，可以充分利用层叠：

```
H1 {border: thick silver solid;
  border-left-width: 20px;}
```

第二个规则会覆盖第一个规则为左边框设置的width值，将宽度替换为20px，如图8-35所示。

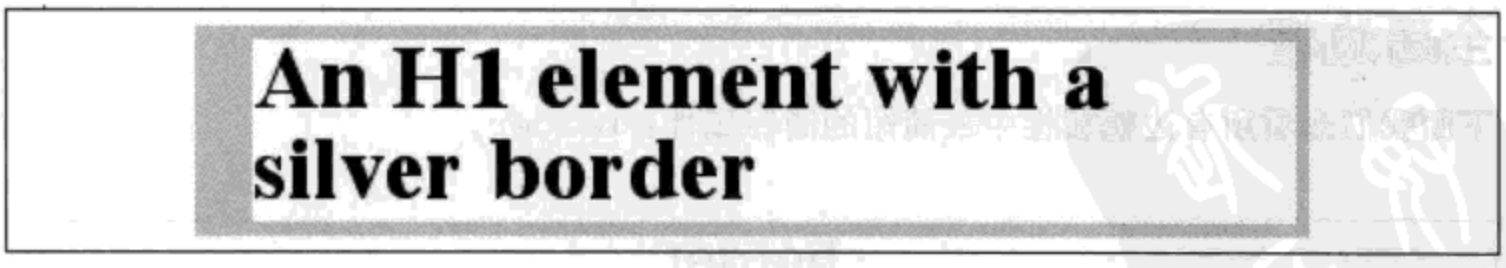

图8-35：充分利用层叠

与以往一样，要简写属性仍要特别当心：如果少了一个值，就会自动地填入默认值，这可能会有意想不到的后果。考虑以下情况：

```
h4 {border-style: dashed solid double;}
h4 {border: medium green;}
```

在此，第二个规则中没有指定 border-style，这意味着会使用默认值 none，这样一来，所有 h4 元素都不会有任何边框。

边框和行内元素

对于处理边框和行内元素我们应该很熟悉了，因为其规则很大程度上与外边距和行内元素的相应规则相同，这在第 7 章做过介绍。不过，在此还是简要地介绍一下。

首先，不论为行内元素的边框指定怎样的宽度，元素的行高都不会改变。下面为粗体文本设置上下边框：

```
strong {border-top: 10px solid gray; border-bottom: 5px solid silver;}
```

重申一句，这个语法在规范中是允许的，不过它对行高绝对没有任何影响。但是，由于边框是可见的，所以将会绘制出来，如图 8-36 所示。

This paragraph contains some ordinary, unstyled text, whose purpose is to provide filler. It also contains **strongly emphasized text** that has been styled as indicated. This can lead to unwanted layout consequences, so authors should exercise caution.

图8-36：行内非替换元素的边框

边框该放在哪里就会放在哪里。

同样地，所有这些只是对行内元素的上下边成立；左右边则是另一回事。如果应用一个左边框或右边框，不仅该边框可见，而且文本会在其旁边显示（即不会覆盖），如图 8-37 所示：

```
strong {border-left: 25px double gray; background: silver;}
```

This paragraph contains some ordinary, unstyled text, whose purpose is to provide filler. It also contains **strongly emphasized text** that has been styled as indicated. This could lead to unwanted layout consequences, so authors should exercise caution.

图8-37：有左边框的行内非替换元素

就像外边距一样，如果设置了边框，浏览器计算换行时并不受行内元素所设置的任何框属性的直接影响。唯一的作用是边框所占空间可能把行中的某些部分向后移一点点，而这有可能改变位于行尾的词。

警告： CSS边框的兼容性问题很少。最麻烦的是，Navigator 4.x不会在块级元素内边距区周围画边框，而会在内边距和边框之间插入一些空间。对于Navigator 4.x，在行内元素上设置边框（或任何其他框属性）都可能极其危险。边框如此，外边距也是如此，原因是一样的（这在本章前面讨论过）。

内边距

元素框的内边距在边框和内容区之间。毫不奇怪，控制这个区的最简单的属性为padding。

padding	
值：	[<length> | <percentage>]{1,4} | inherit
初始值：	对于简写元素未定义
应用于：	所有元素
继承性：	无
百分数：	相对于包含块的 width
计算值：	见单个属性（padding-top 等）
说明：	内边距绝对不能为负

可以看到，这个属性接受任何长度值或某个百分数值。所以，如果你希望所有h1元素的各边都有10像素的内边距，这很容易：

```
h1 {padding: 10px; background-color: silver;}
```

另一方面，你可能希望h1元素的内边距不均匀，而h2元素有规则的内边距：

```
h1 {padding: 10px 0.25em 3ex 3cm;} /* uneven padding */
h2 {padding: 0.5em 2em;} /* values replicate to the bottom and left sides */
```

不过，如果只增加内边距，要真正看到所设置的内边距可能有些困难，所以下面加上一个背景色，如图8-38所示：

```
h1 {padding: 10px 0.25em 3ex 3cm; background: gray;}
h2 {padding: 0.5em 2em; background: silver;}
```

如图8-38所示，元素的背景延伸到其内边距。前面讨论过，它还会延伸到边框的外边界，不过背景到达边框之前必须先经过内边距。

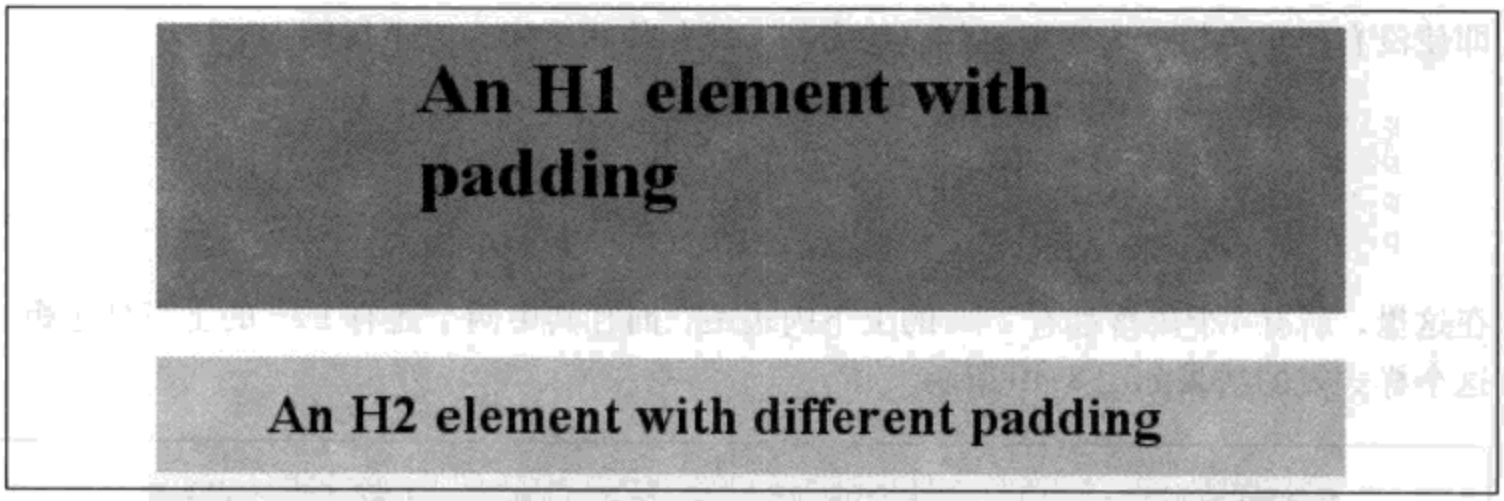

图8-38：利用背景色查看内边距

默认地，元素没有内边距。例如，段落之间的间隔传统上只由外边距保证。如果没有内边距，元素的边框会与元素本身的内容相当接近。因此，在元素上放边框时，同时增加内边距通常是一个好主意，如图 8-39 所示。

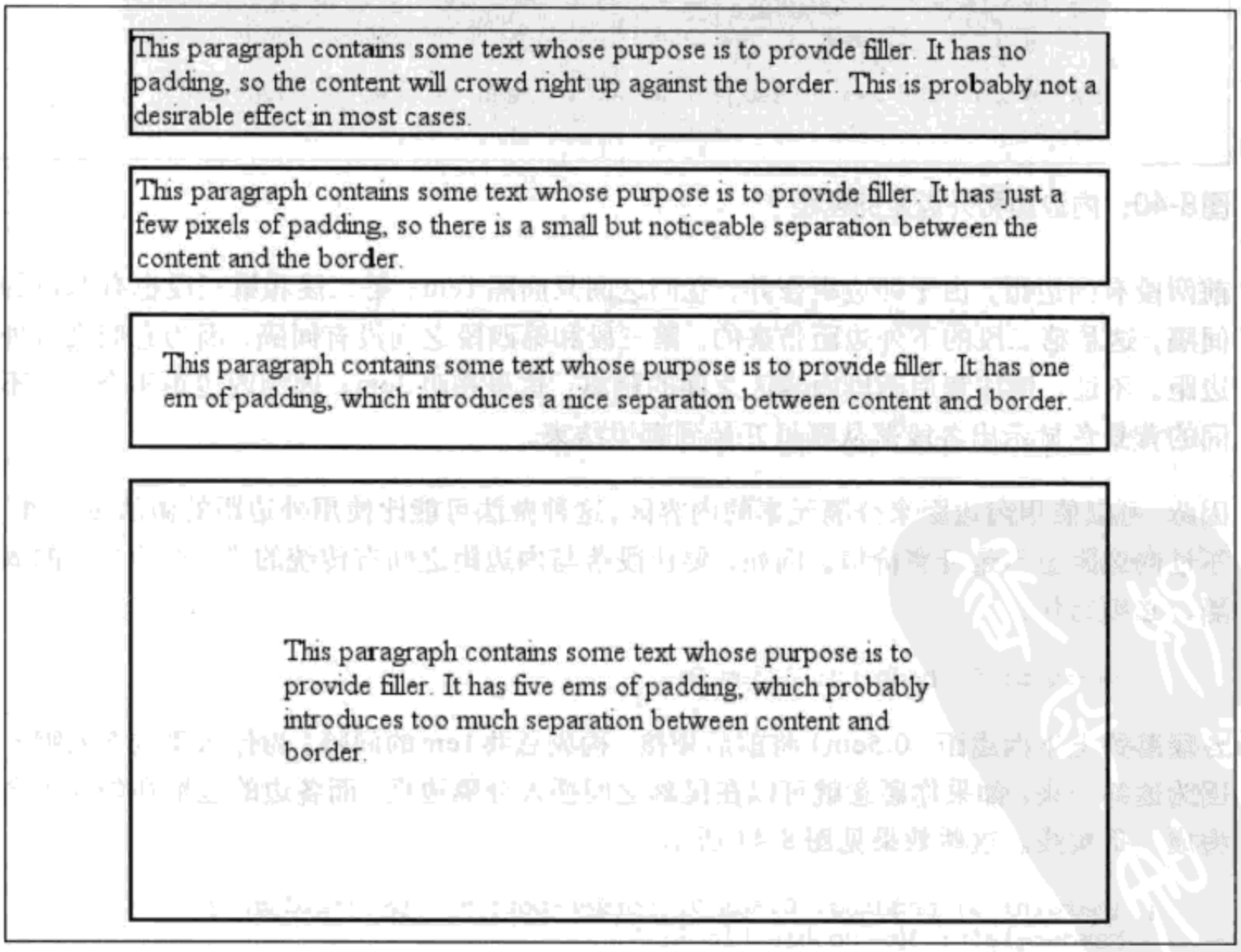

图8-39：在有边框的块级元素上设置内边距的效果

即使没有使用边框，内边距也能以特有的方式起作用。考虑以下规则：

```
p {margin: 1em 0; padding: 1em 0;}
p.one, p.three {background: gray;}
p.two, p.four {background: silver;}
p.three, p.four {margin: 0;}
```

在这里，所有4个段落都有1em的上下内边距，而且其中两个还有1em的上下外边距。这个样式表的结果如图8-40所示。

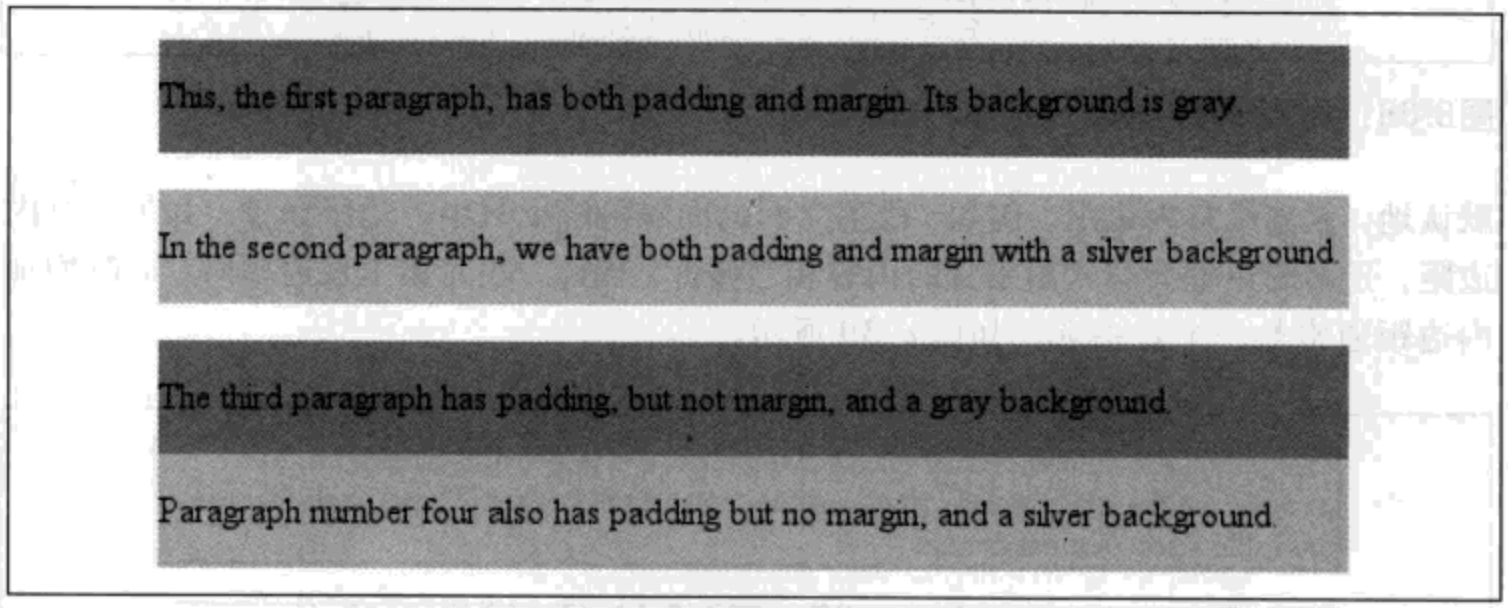

图8-40：内边距和外边距的差别

前两段有内边距，由于外边距合并，它们之间只间隔1em。第二段和第三段也有1em的间隔，这是第二段的下外边距带来的。第三段和第四段之间没有间隔，因为它们没有外边距。不过，请注意后两段内容区之间的距离：这里相距2em，因为内边距不合并。不同的背景色显示出各段落从哪里开始到哪里结束。

因此，可以使用内边距来分隔元素的内容区，这种做法可能比使用外边距的做法更困难，不过内边距也不是没有价值。例如，要让段落与内边距之间有传统的“一个空行”的间隔，必须写作：

```
p {margin: 0; padding: 0.5em 0;}
```

各段落的上下内边距（0.5em）将前后相接，构成总共1em的间隔。为什么要这么做呢？因为这样一来，如果你愿意就可以在段落之间插入分隔边框，而各边的边框在外观上会构成一条实线。这些效果见图8-41所示：

```
p {margin: 0; padding: 0.5em 0; border-bottom: 1px solid gray;
  border-left: 3px double black;}
```

I bet birth is actually when all the organs get together and throw the baby out. It's a lot like what happens when one of your roommates is a jerk, taking up a lot more than his fair share of spaces and always raiding the fridge but never buying any groceries. Finally the rest of you get fed up with him and tell him that maybe he ought to find alternate living arrangements.

It's the same thing with birth. It usually takes about nine months, but sooner or later the organs decide that enough is enough, and send out some signal which is the chemical equivalent of, "Look, kid, you're cute and all but you've become a major pain. You're cramping up the entire place, all you do is just lie around all day kicking us, you suck down more food than the rest of us put together, and you haven't paid your rent since you got here. We're really sorry, but you're going to have to find someplace else to live. Now beat it! And don't let the cervix hit you in the ass on your way out."

(This is probably why babies tend to cry right after they've been born. They start life by having been rejected by the internal organs, plus now they have no idea where they're going to live. How would you feel in that situation? You'd be crying your eyes out.)

图8-41：使用内边距而不是外边距

百分数值和内边距

前面提到过，可以为元素的内边距设置百分数值。像外边距一样，百分数值要相对于其父元素的 width 计算，所以如果父元素的 width 改变，它们也会改变。例如，假设有以下规则，如图 8-42 所示：

```
p {padding: 10%; background-color: silver;}

<div style="width: 200px;">
<p>This paragraph is contained within a DIV that has a width of 200 pixels,
so its padding will be 10% of the width of the paragraph's parent element.
Given the declared width of 200 pixels, the padding will be 20 pixels on
all sides.</p>
</div>
<div style="width: 100px;">
<p>This paragraph is contained within a DIV with a width of 100 pixels,
so its padding will still be 10% of the width of the paragraph's parent.
There will, therefore, be half as much padding on this paragraph as that
on the first paragraph.</p>
</div>
```

注意，上下内边距与左右内边距一致；也就是说，上下内边距的百分数会相对于父元素宽度计算，而不是相对于高度。当然，这在前面已经见过（如果你忘了，可以提醒一下，我们在“外边距”一节中讨论过这个问题），不过还是有必要再回顾一下，看看它是如何作用的。

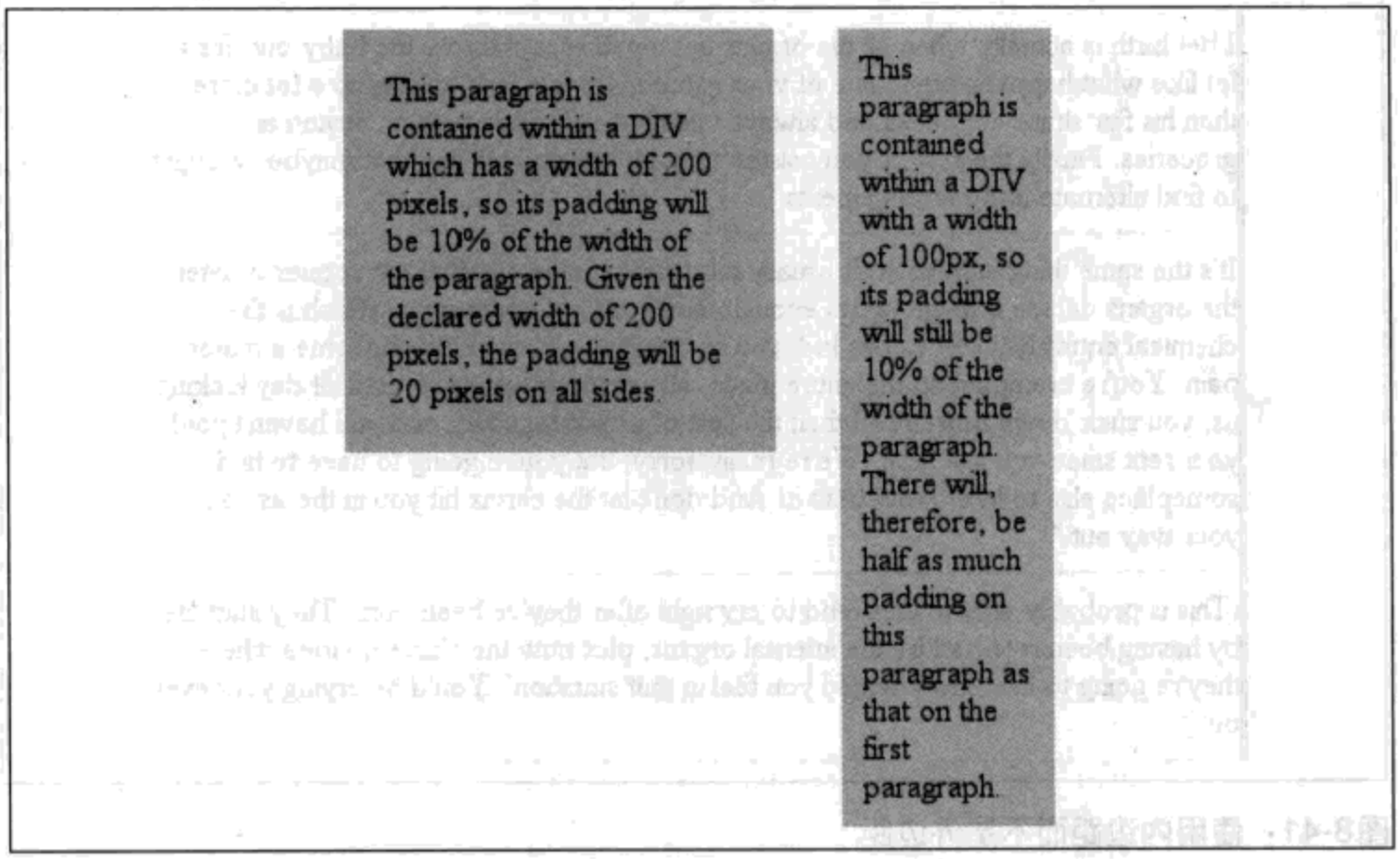

图8-42：内边距、百分数和父元素的width

单边内边距

可以猜得到，CSS提供了一些属性来设置框的单边内边距，而不影响其他边上的内边距。

padding-top、padding-right、padding-bottom、padding-left

值：	<length> | <percentage> | inherit
初始值：	0
应用于：	所有元素
继承性：	无
百分数：	相对于包含块的width
计算值：	对于百分数值，根据指定确定；对于长度值，则为绝对长度
说明：	内边距绝对不能为负

这些属性的作用如你所料。例如，以下两个规则会指定同样的内边距：

```
h1 {padding: 0 0 0 0.25in;}
h2 {padding-left: 0.25in;}
```

内边距和行内元素

对于行内元素，外边距和内边距存在一个重要的区别。为说明这一点，下面先来看左右内边距。如果为左右内边距设置了值，左右内边距将是可见的，如图 8-43 所示：

```
strong {padding-left: 10px; padding-right: 10px; background: silver;}
```

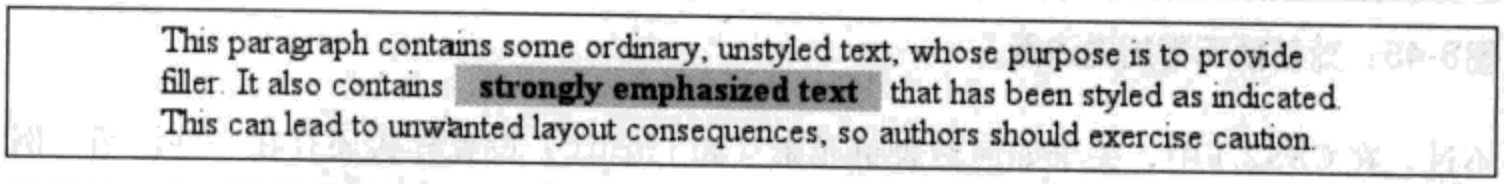

图8-43：行内非替换元素的内边距

注意，行内非替换元素的两端都出现了额外的空背景。这是所设置的内边距。像外边距一样，左内边距应用到元素的开始处，右内边距应用到元素的最后；不过，内边距不会应用到各行的左右两边。对于替换元素也是如此，不过当然这种元素不会跨行。

理论上，对于有背景色和内边距的行内非替换元素，背景可以向元素上面和下面延伸：

```
strong {padding-top: 0.5em; background-color: silver;}
```

从图 8-44 可以了解其效果。

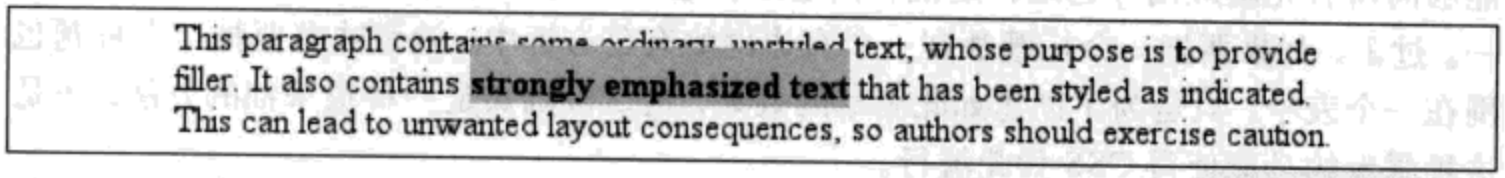

图8-44：行内非替换元素的更多内边距

当然，行高没有改变，不过由于内边距确实能延伸背景，所以背景应该可见，是这样吗？不错，背景确实可见，它与前面的行重叠，这正是我们期望的结果。

内边距和替换元素

尽管看上去可能有些奇怪，不过确实可以向替换元素应用内边距，但在写作本书时这方面还存在一些限制。

最让人奇怪的是，可以向图像应用内边距，如下：

```
img {background: silver; padding: 1em;}
```

不论替换元素是块级元素还是行内元素，内边距都会围绕其内容，背景色将填入该内边距，如图 8-45 所示。还可以看到内边距会把元素的边框推离其内容。

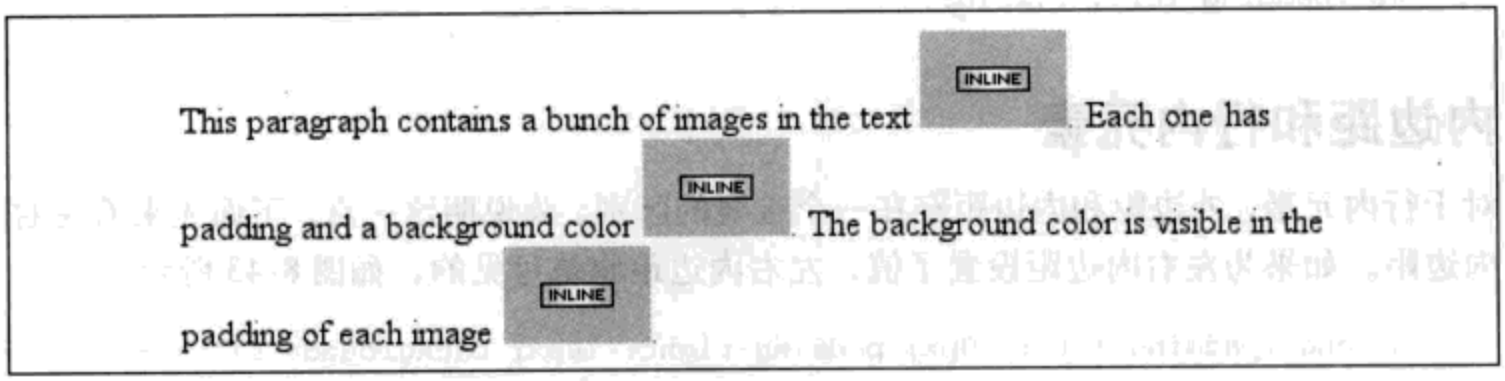

图8-45：对替换元素加内边距

不过，在CSS2.1中，关于如何对表单元素（如 input）设置样式还存在一些混淆。例如，复选框的内边距在哪里就不是很清楚。因此，在写作本书时，有些浏览器（如Mozilla）会忽略表单元素的内边距（或其他形式的样式）。可能将来会出现一个CSS规范描述表单元素的样式。

另一个可能的限制是，很多较老的浏览器不向图像应用内边距，包括 Windows 平台的 IE5。

小结

能够向所有元素应用外边距、边框和内边距，是CSS超越传统Web标记语言的要素之一。过去，如果要把一个标题放在一个有边框的有色方框中，就意味着要把这个标题包围在一个表中，只是为了创建如此简单的效果，这么做确实是一种很笨拙的办法。正是这种强大的功能使得CSS如此流行。

第9章

颜色和背景

还记得你第一次改变Web页面的颜色是什么时候吗？以前总是灰色背景上的黑色文本，其间有一些蓝色的链接，突然之间，你可以按自己的意愿使用任何颜色组合，可以是黑色背景上的淡蓝色文本，而超链接是浅绿色的。这只是朝着有色文本前进的一个小进步（最终目标是页面上的文本有多种颜色），这要归功于 <FONT COLOR=" . . . ">。一旦还能增加背景图像，那么一切皆有可能，或者起码看上去是这样。CSS 在颜色和背景方面则走得更远，它允许向一个页面应用多种不同的颜色和背景，而且根本不需要任何 FONT 或 TABLE 标记。

颜色

设计页面时，要在开始前先做好计划。不论什么情况下通常都是如此，不过对于颜色，这一点更为重要。如果你要让所有超链接都是黄色，它们会与文档中某些部分的背景色冲突吗？如果使用了太多的颜色，会不会让用户无所适从？（提示：确实如此）。如果改变了默认的超链接颜色，用户还能找出链接在哪里吗？（例如，如果让常规文本和超链接文本颜色相同，要找出链接就会困难得多，实际上，如果链接没有加下划线，则几乎不可能识别出超链接。）

尽管需要先做一些规划，不过修改元素颜色的功能还是让人趋之若鹜，这几乎是每一位创作人员都想使用的功能，而且也确实相当常用。如果使用得当，颜色确实能强化文档的表示。例如，假设你有一个设计，其中所有 h1 元素都是绿色，大多数 h2 元素是蓝色，所有超链接都应当是暗红色。不过，在某些情况下你希望 h2 元素是深蓝色，因为与之

关联有不同类型的信息。要处理这种情况，最容易的办法是为应当为深蓝色的各个h2指定一个class，然后作以下声明：

```
h1 {color: green;}
h2 {color: blue;}
h2.dkblue {color: navy;}
a {color: maroon;}   /* a good dark red color */
```

注意： 选择的类名最好描述其中包含的信息类型，而不是你想要达到什么视觉效果。例如，假设你希望对所有作为下级标题的h2元素应用深蓝色。更可取的做法是选择subsec甚至sub-section作为类名，这就能反映某种含义，而且更重要的是，这不依赖于任何表示概念。毕竟，也许你以后会决定所有下级标题是深红色而不是深蓝色，如果是这样，写作h2.dkblue {color: maroon;}看上去就会有些傻。

从这个简单的例子可以看到，使用样式之前最好先做些规划，这样才能充分合理地使用所有工具。例如，假设向前例中的页面增加一个导航条。在这个导航条中，超链接应当是黄色而不是深红色。如果导航条标志为ID是navbar，则只需增加以下规则：

```
#navbar a {color: yellow;}
```

这会改变导航条中超链接的颜色，而不影响文档中的其他超链接。

CSS中实际上只有一种颜色类型，即纯色。如果将一个文档的color设置为red，文本都将是红色。当然，HTML的做法也一样。使用HTML 3.2时，如果声明<BODY LINK="blue" VLINK="blue">，那么所有超链接都将是蓝色，而不论它们放在文档中的哪个位置。

使用CSS时也是如此。如果使用CSS将所有超链接（包括已访问和未访问）的color都设置为blue，它们将全是蓝色。同样，如果使用样式设置body的背景为green，那么整个body的背景都是同样的绿色。

在CSS中，可以为任何元素设置前景和背景色，从body到强调元素和超链接元素，再到几乎所有一切（列表项、整个列表、标题、表单元格，甚至从某种程度上图像也可以设置前景和背景色）。不过，要理解这是如何工作的，重要的是应当理解元素前景中有什么、没有什么。

先来讨论前景本身；一般来说，前景是元素的文本，不过前景还包括元素周围的边框。因此，有两种方式直接影响一个元素的前景色：可以使用color属性，也可以使用某个边框属性设置边框颜色，这在上一章讨论过。

前景色

要设置一个元素的前景色，最容易的办法是利用属性color。

color	
值：	<color> | inherit
初始值：	用户代理特定的值
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

在第4章曾经讨论过，这个属性接受任何合法颜色类型的值，例如#FFCC00或rgb(100%,80%,0%)，也可以接受第13章将介绍的系统颜色关键字。

对于非替换元素，color设置了元素中文本的颜色，如图9-1所示：

```
<p style="color: gray;">This paragraph has a gray foreground.</p>
<p>This paragraph has the default foreground.</p>
```

This paragraph has a gray foreground.

This paragraph has the default foreground.

图9-1：声明颜色与默认颜色

注意： 在图9-1中，默认前景色为黑色。并不总是如此，因为用户可能让其浏览器（或其他用户代理）使用另外一种前景（文本）颜色。如果默认文本设置为绿色，上例中的第二段将是绿色而不是黑色，但是第一段还是灰色。

当然，并不仅限于完成这种简单的操作。color有很多用法。例如，可能有些段落中包含一些文本，提醒用户某个可能的问题。为了突出这些文本，可能决定将其设置为红色。只需为包含警告文本的各个段落提供一个class值warn （<p class="warn">），并设置以下规则：

```
p.warn {color: red;}
```

在同一个文档中，你可能认为警告段落中的未访问链接应当是绿色：

```
p.warn {color: red;}
p.warn a:link {color: green;}
```

然后你又改主意了，认为警告文本应当是暗灰色，这些文本中的链接应当是中灰色。只需修改前面的规则来反映这些新值，如图9-2所示：

```
p.warn {color: #666;}
p.warn a:link {color: #AAA;}
```

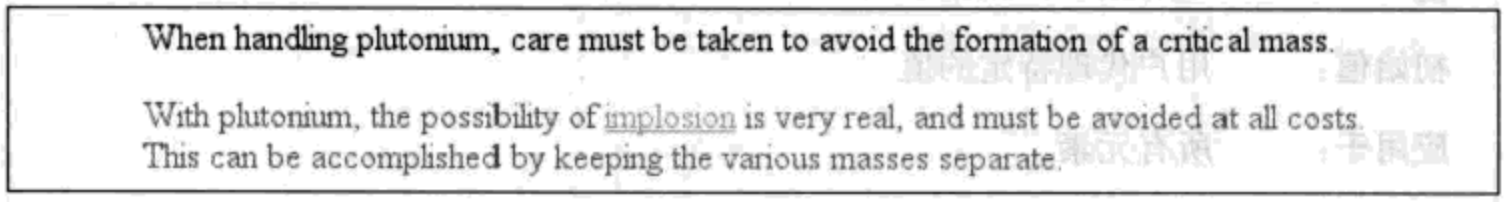

图9-2：改变颜色

color的另一个用法是让用户注意某类文本。例如，粗体文本已经很明显了，不过你可能想让它们有另一种颜色，使之更显突出——例如可以设置为紫红色：

```
b, strong {color: maroon;}
```

然后可能决定class为highlight的所有表单元格要包含淡黄色文本：

```
td.highlight {color: #FF9;}
```

当然，如果你没有为任何文本设置背景色，可能会有风险，因为用户的设置与你自己的设置可能不一致。例如，如果一个用户将其浏览器的背景色设置为灰黄色，如#FFC，那么前面的规则就会在灰黄色背景上生成淡黄色文本。因此，通常最好同时设置前景和背景色（本章后面将谈到背景色）。

警告： 当心Navigator 4中的颜色使用，它会替换掉它不认识的color值。这种替换并不完全是随机的，不过这确实不太好。例如，invalidValue会变成深蓝色，inherit（这是一个合法的CSS2值）则会变成一种很难看的黄绿色。在另外一些情况下，transparent背景会变成黑色。

替换属性

color有很多用法，其中最基本的是替换HTML 3.2的BODY属性TEXT、LINK、ALINK和VLINK。利用锚伪类，color完全可以替换这些BODY属性。下例中的第一行可以用后面的CSS重写，其结果如图9-3所示：

```
<body text="black" link="#808080" alink="silver" vlink="#333333">
```

```
body {color: black;}      /* replacement css */
a:link {color: #808080;}
a:active {color: silver;}
a:visited {color: #333333;}
```

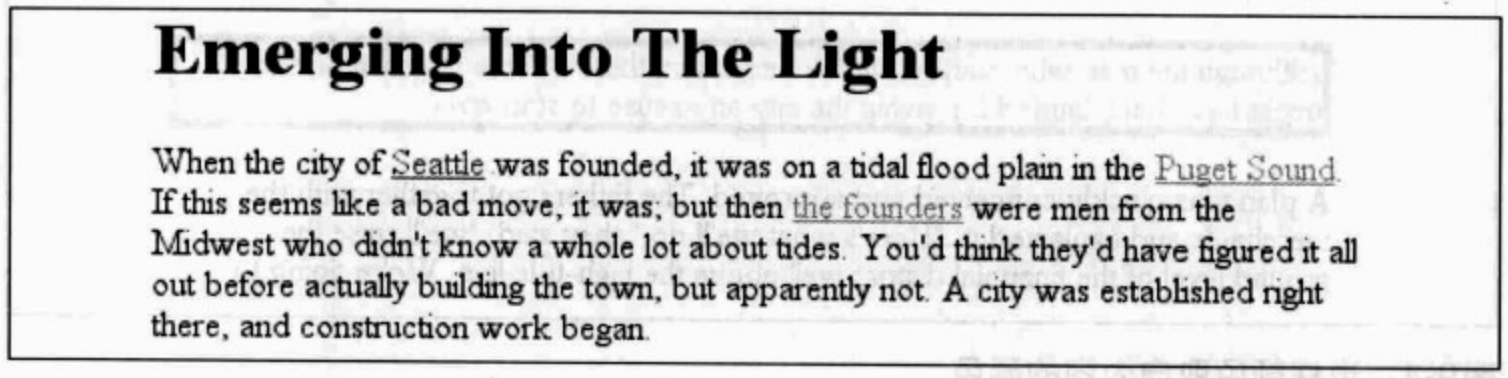

图9-3：用CSS替换BODY属性

尽管这看上去好像要键入很多额外的代码，不过请考虑两个问题。首先，与使用BODY属性的老办法相比，这是一个重大改进，使用老办法只能在文档级进行修改。那时，如果你希望某些链接是中灰色，而另外一些是很深的深灰色，利用BODY属性是办不到的。相反，你必须在每个需要是深灰色的锚上使用<FONT COLOR="#666666">。但CSS不同；现在，只需为所有灰色的锚增加一个类，再相应地修改样式：

```
body {color: black;}
a:link {color: #808080;}      /* medium gray */
a.external:link {color: silver;}
a:active {color: silver;}
a:visited {color: #333;}   /* a very dark gray */
```

这会把class为external的所有锚设置为银色而不是中灰色。一旦被访问过，这些链接将是深灰色，除非对此再增加一个特殊规则：

```
body {color: black;}
a:link {color: #808080;}      /* medium gray */
a.external:link  {color: #666;}
a:active {color: silver;}
a:visited {color: #333;}   /* a very dark gray */
a.external:visited {color: black;}
```

这样一来，在访问之前，所有external链接是中灰色，在访问后将变成黑色，而所有其他链接访问后为暗灰色，未访问时为中灰色。

影响边框

color值还可以影响元素周围的边框。假设已经声明了以下样式，其结果如图9-4所示：

```
p.aside {color: gray; border-style: solid;}
```

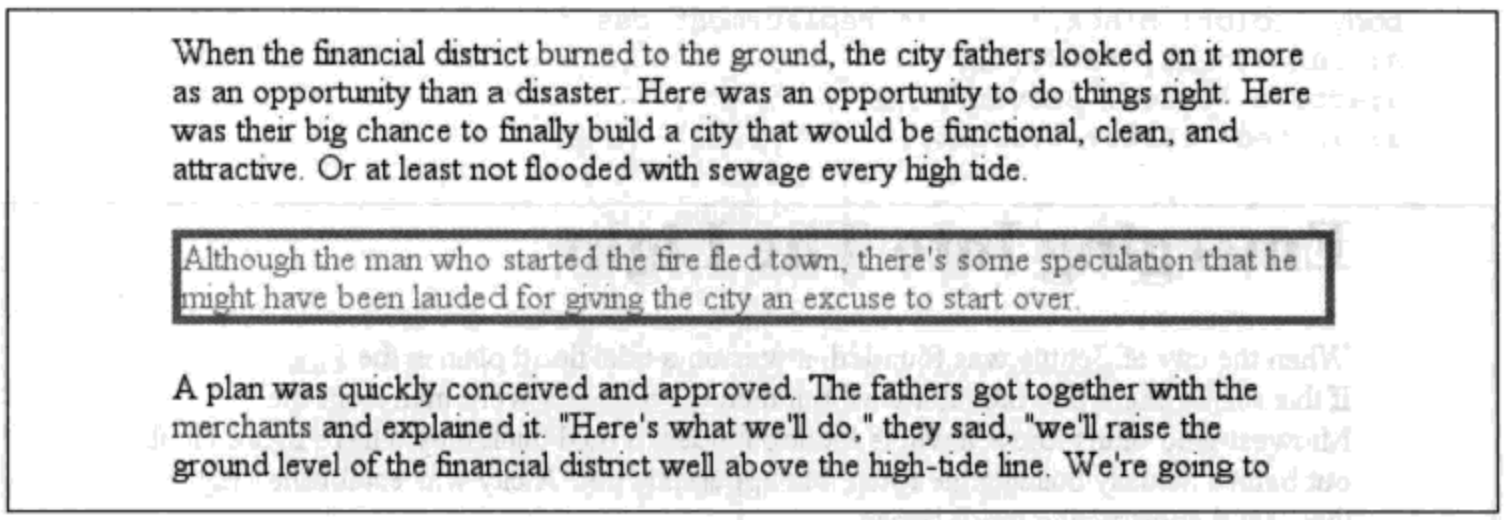

图9-4：边框颜色取自内容的颜色

元素<p class="aside">有灰色文本和灰色中等宽度的实线边框。这是因为，默认地前景色会应用到边框。要覆盖这种行为，最基本的办法是使用属性border-color：

```
p.aside {color: gray; border-style: solid; border-color: black;}
```

这会使文本为灰色，但边框是黑色。为border-color设置的值总会覆盖color值。

有时，利用边框可以影响图像的前景色。由于图像本身就由颜色组成，所以实际上这些使用color是无法影响的，不过可以改变图像周围任何边框的颜色。这可以利用color或border-color做到。因此，以下规则对class为*type1*和*type2*的图像有相同的视觉效果，如图9-5所示：

```
img.type1 {color: gray; border-style: solid;}
img.type2 {border-color: gray; border-style: solid;}
```

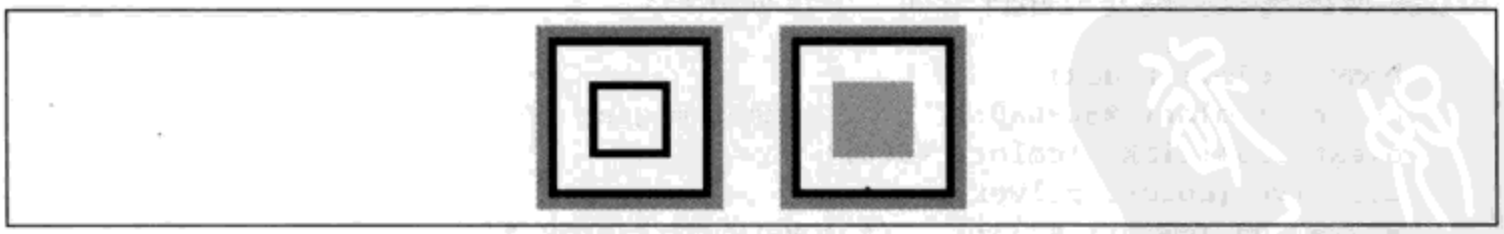

图9-5：为图像设置边框颜色

影响表单元素

（从理论上讲）也可以为表单元素设置color值。声明select元素有暗灰色文本就很简单，如下：

```
select {color: rgb(33%,33%,33%);}
```

这可能还会设置select元素周围边框的颜色，也可能不会。这完全取决于用户代理及其默认样式。

另外，还可以设置input元素的前景色，如图9-6所示，这会把所设置的颜色应用到所有输入元素，从文本框到单选钮再到复选框都会使用这种颜色：

```
select {color: rgb(33%,33%,33%);}
input {color: gray;}
```

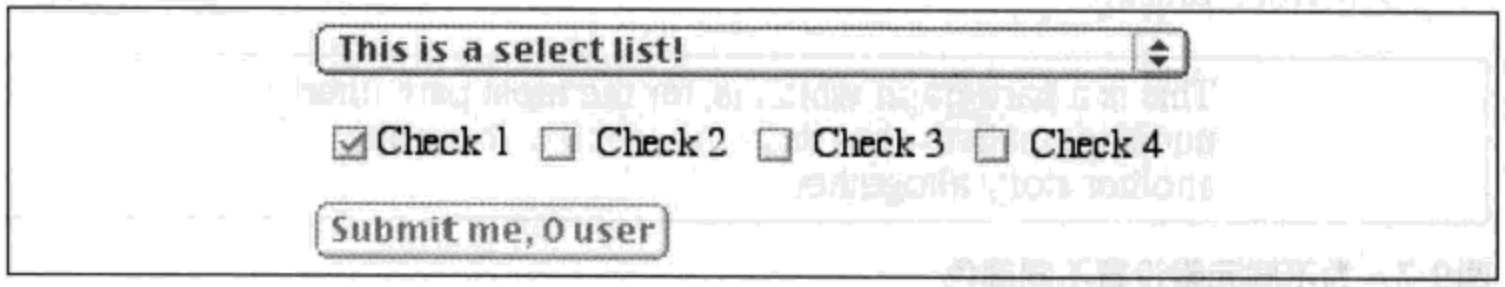

图9-6：改变表单元素前景色

在图9-6中可以注意到，挨着复选框的文本颜色还是黑色。这是因为你只是为input和select之类的元素指定了样式，而没有为常规的段落（或其他）文本指定样式。

CSS1无法区分不同类型的input元素。所以，如果要让复选框的颜色不同于单选按钮，就必须为它们分别指定不同的类，以得到所需的结果：

```
input.radio {color: #666;}
input.check {color: #CCC;}

<input type="radio" name="r2" value="a" class="radio" />
<input type="checkbox" name="c3" value="one" class="check" />
```

在CSS2及以后版本中则要容易一些，可以根据不同元素的属性来区分元素，这归功于属性选择器：

```
input[type="radio"] {color: #333;}
input[type="checkbox"] {color: #666;}

<input type="radio" name="r2" value="a " />
<input type="checkbox" name="c3" value="one "/>
```

利用属性选择器就可以不再使用类，至少这里是如此。遗憾的是，许多用户代理并不支持属性选择器，所以目前使用类可能还是必要的。

警告： Navigator 4不会向表单元素应用颜色，不过在Internet Explorer 4及以上版本和Opera 3.5+中，为表单元素设置颜色是起作用的。但其他浏览器的许多版本甚至不允许对表单元素设置样式，因为它们不确定应该如何对这些元素应用样式。

继承颜色

由color的定义可知，这个属性是可以继承的。这是有道理的，因为如果声明了p {color: gray;}，你可能希望该段落中的所有文本也应当是灰色，即便是强调文本、粗体文本等也不例外。当然，如果你希望这些元素有不同的颜色，这也很容易，如图9-7所示：

```
em {color: gray;}
p {color: black;}
```

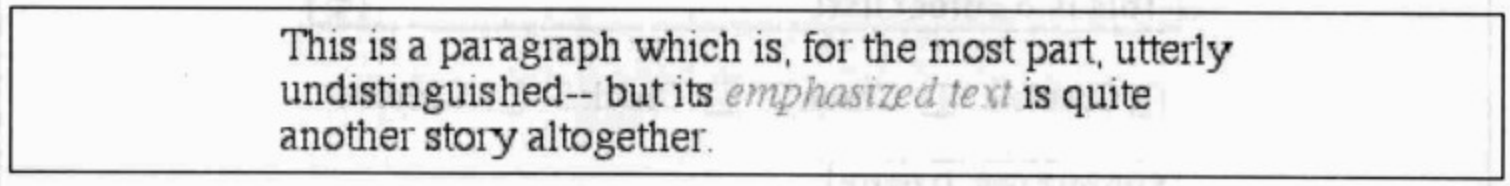

图9-7：为不同元素设置不同颜色

由于color是可以继承的，理论上讲，可以把一个文档中的所有正常文本设置为某种颜色，如通过声明body {color: red;}设置为红色。这会把所有没有其他样式的文本变成红色（如锚就不包含在内，锚有其自己的颜色样式）。不过，还有一些浏览器对表之类的元素设置有预定义的颜色，这就使得body颜色无法继承到表单元格中。在这种浏览器中，由于table元素的color值由浏览器定义，浏览器的值会比继承的值更优先。这一点很讨厌，也没有必要，不过好在解决这个问题（往往）很容易，只需使用列出各表元素的选择器。例如，要让所有表内容与文档体一样都是红色，可以用以下规则：

```
body, table, td, th {color: red;}
```

这一般都能解决问题。注意，对于大多数现代浏览器来说，都没有必要使用这种选择器，因为现代浏览器早已经修正了先前版本中存在的这种继承bug。

背景

元素的背景区包括前景之下直到边框外边界的所有空间；因此，内容框和内边距都是元素背景的一部分，且边框画在背景之上。

CSS允许应用纯色作为背景，也允许使用背景图像创建相当复杂的效果；CSS在这方面的能力远远在HTML之上。

背景色

类似于设置前景色，可以为元素的背景声明一个颜色。为此，可以使用属性background-color，毫不奇怪，它接受所有合法的颜色，还可以接受一个使背景透明的关键字。

background-color	
值：	<color> | transparent | inherit
初始值：	transparent
应用于：	所有元素
继承性：	无
计算值：	根据指定确定

如果你希望背景色从元素中的文本向外稍有延伸，只需增加一些内边距，如图9-8所示。

```
p {background-color: gray; padding: 10px;}
```

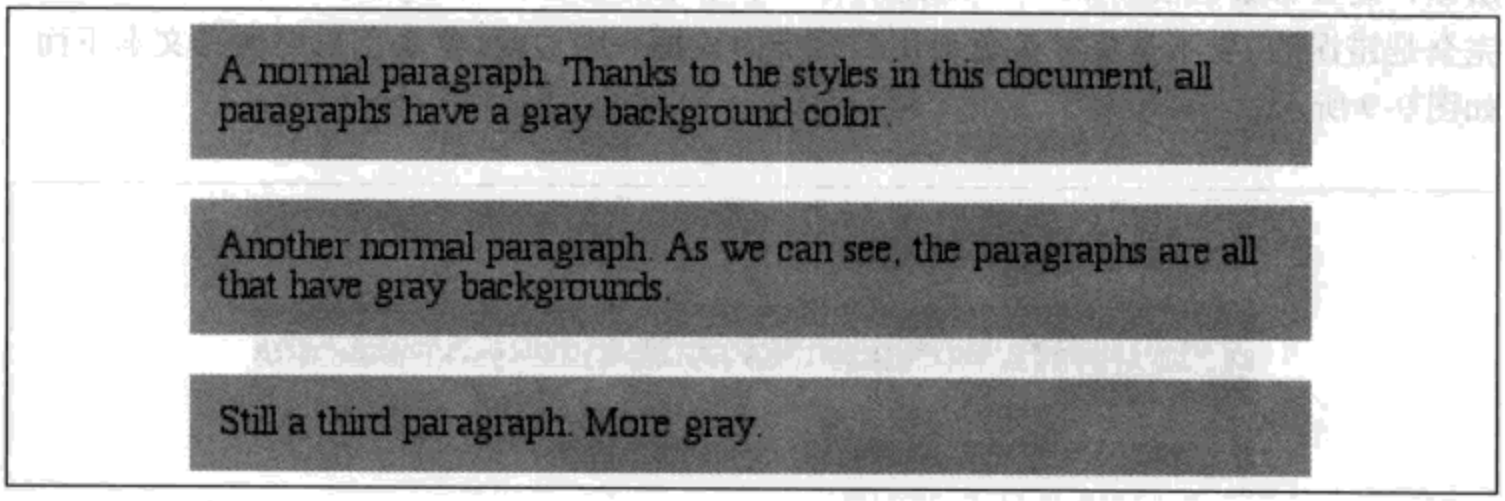

图9-8：背景和内边距

可以为几乎所有元素设置背景色，这包括body一直到em和a等行内元素。background-color不能继承。其默认值是transparent，这是有道理的：如果一个元素没有指定的颜色，那么背景就应当是透明的，这样其祖先元素的背景才能可见。

要了解这种继承，可以想象订在一个文字墙上的塑料标志。透过这个标志还可以看到墙，不过这不是标志的背景，这是墙的背景（按CSS的术语来讲）。类似地，如果为画布设置一个背景，文档中有些元素没有自己的背景，那么透过所有这些元素都能看到这个背景。这些元素并没有继承背景，只是能透过它们看到背景而已。看上去好像没有什么区别，不过在关于背景图像的一节中你将了解到，这确实是一个重大差别。

大多数情况下都没有必要使用关键字transparent，因为这是默认值。不过，有些情况下这个关键字可能很有用。假设一个用户将其浏览器设置为使所有链接都有一个白色背景。在你设计页面时，将锚设置为有一个白色前景，而且你不希望这些锚有背景。为了确保你的设计选择可行，需要以下声明：

```
a {color: white; background-color: transparent;}
```

如果没有背景色，白色前景和用户指定的白色背景就会混在一起，这样一来链接将变得完全不可读。尽管这个例子不太实际，不过确实是可能的。

注意： 创作人员样式和读者样式有可能结合，出于这个原因，CSS 检验器会生成以下警告，“color没有相应的background-color”。这是在提醒你，创作人员指定的颜色与用户指定的颜色可能发生交互，而你的规则没有考虑到这种可能性。这些警告并不表示你的样式无效：只有错误才会导致检验失败。

历史问题

所以，设置背景色很简单 —— 对此只有一个小小的警告：Navigator 4 对背景色的处理完全是错误的。它不是将背景色应用到整个内容框和内边距，背景色只出现在文本下面，如图 9-9 所示。

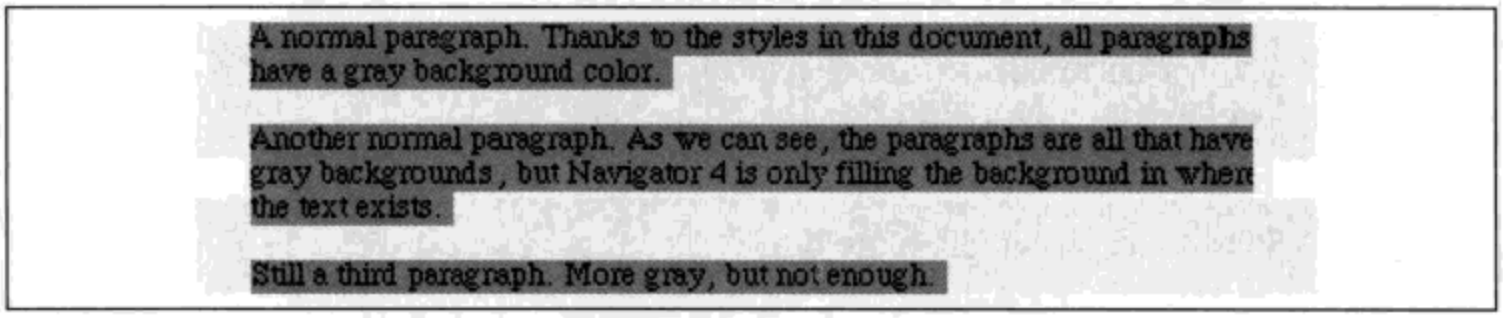

图9-9：Navigator 4.x的错误行为

重申一句：这种行为是完全错误的。为了解决这种错误，必须在元素上设置一个边框，然后将其设置为与文档背景色相同的颜色：

```
body {background: silver;}
p {background-color: gray; padding: 0.1px; border: 1px solid silver;}
```

注意，要使用这个技术，必须设置一个 border-style。可以使用这个特定属性，也可以使用一个 border 属性值，使用哪一个并不重要。

当然，这样做是在设置元素的边框，这个边框也会在其他用户代理中出现。除此之外，Navigator还不能很好地处理内边距，所以前例中内容框和边框之间会出现少量空白。好在新的浏览器都不存在这些问题。

特殊效果

只需结合 color 和 background-color，就可以创建一些有用的效果：

```
h1 {color: white; background-color: rgb(20%,20%,20%);
  font-family: Arial, sans-serif;}
```

这个例子如图 9-10 所示。

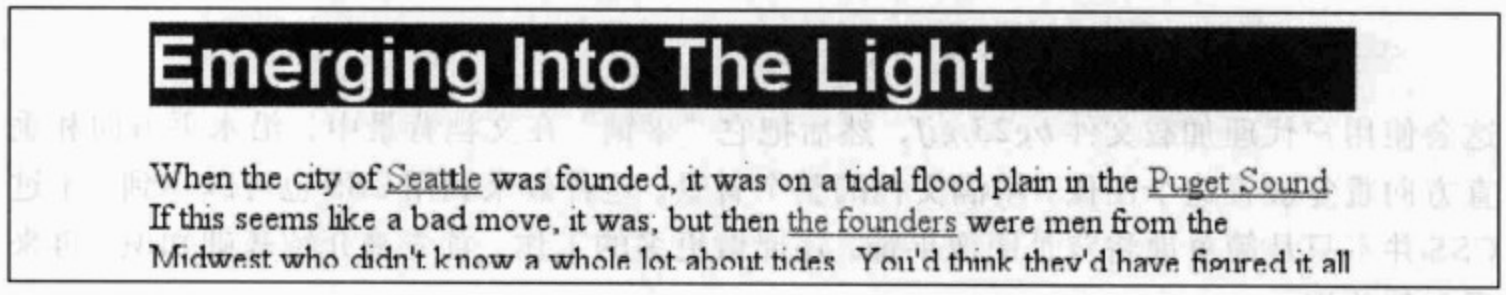

图9-10：H1元素的一个漂亮的效果

当然，颜色有很多，颜色组合就更多了，但这里无法将所有颜色组合都显示出来，我们只能像前面一样还是用灰度表示。不过，我还是尽量让你对此有一些认识。

这个样式表稍有些复杂，如图 9-11 所示。

```
body {color: black; background-color: white;}
h1, h2 {color: yellow; background-color: rgb(0,51,0);}
p {color: #555;}
a:link {color: black; background-color: silver;}
a:visited {color: gray; background-color: white;}
```

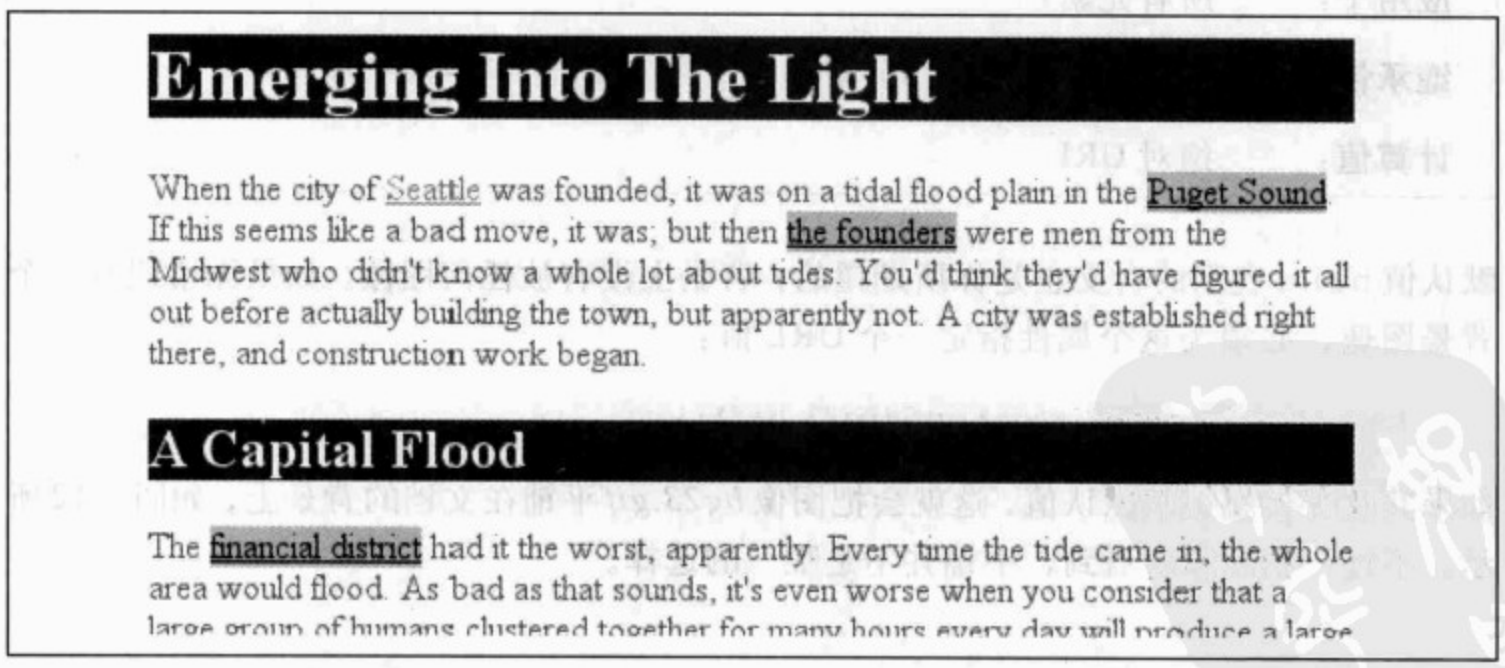

图9-11：一个更复杂的样式表的结果

当然，还能做得更漂亮，这还只是开始而已。你可以充分发挥主动性，自己尝试些例子！

背景图像

前面已经介绍了前景和背景色的基础知识，下面来讨论背景图像。在HTML3.2中，可以通过使用BODY属性BACKGROUND向文档的背景关联一个图像：

```
<BODY BACKGROUND="bg23.gif">
```

这会使用户代理加载文件*bg23.gif*，然后把它“平铺”在文档背景中，沿水平方向和垂直方向重复放置这个图像，填满文档的整个背景。这种效果利用CSS也可以达到，不过CSS并不只是简单地将背景图像平铺，还能做更多的工作。首先来介绍基础知识，再来看如何使用。

使用图像

首先，要把图像放入背景，需要使用属性background-image。

background-image	
值：	<uri> | none | inherit
初始值：	none
应用于：	所有元素
继承性：	无
计算值：	绝对URI

默认值none表示的含义正是你所期望的：背景上没有放任何图像。如果你希望有一个背景图像，必须为这个属性指定一个URL值：

```
body {background-image: url(bg23.gif);}
```

如果其他背景属性取默认值，这就会把图像*bg23.gif*平铺在文档的背景上，如图9-12所示。不过，稍后你会看到，平铺并不是唯一的选择。

注意： 指定背景图像的同时可以再指定一个背景色，这往往是个好主意；本章稍后还会再谈到这个概念。

允许向任何元素应用背景图像，可以是块级元素也可以是行内元素。当然，大多数背景都应用到body元素，不过并不仅限于此：

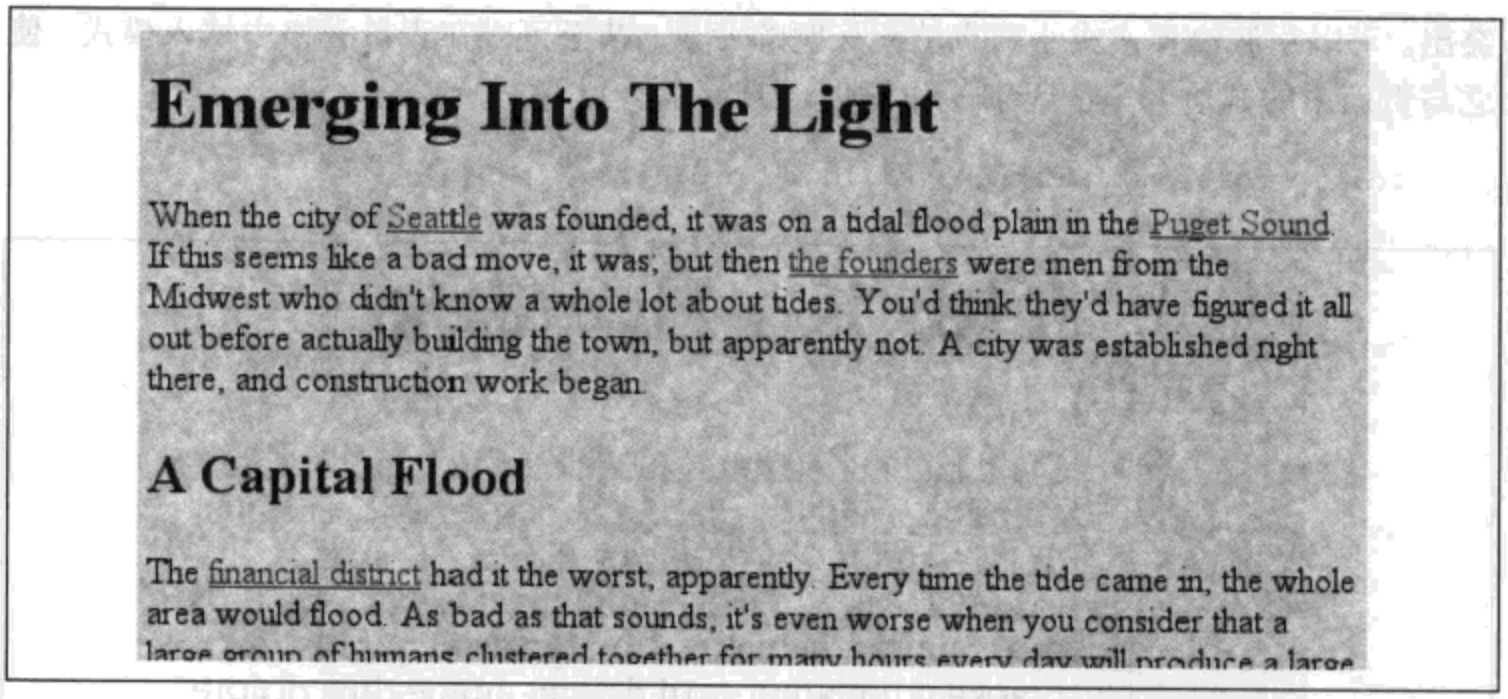

图9-12：在CSS中应用一个背景图像

```
p.starry {background-image: url(http://www.site.web/pix/stars.gif);
   color: white;}
a.grid {background-image: url(smallgrid.gif);}

<p class="starry">It's the end of autumn, which means the stars will be
brighter than ever!  <a href="join.html" class="grid">Join us</a> for
   a fabulous evening of planets, stars, nebulae, and more...
```

在图9-13中可以看到，这里向一个段落应用了一个背景，而没有对文档的其他部分应用背景。还可以进一步定制，如把背景图像放在超链接等行内元素上，这在图9-13中也有显示。当然，如果你希望能看到平铺模式，可能要求这个图像必须很小。毕竟，单个字母不会太大！

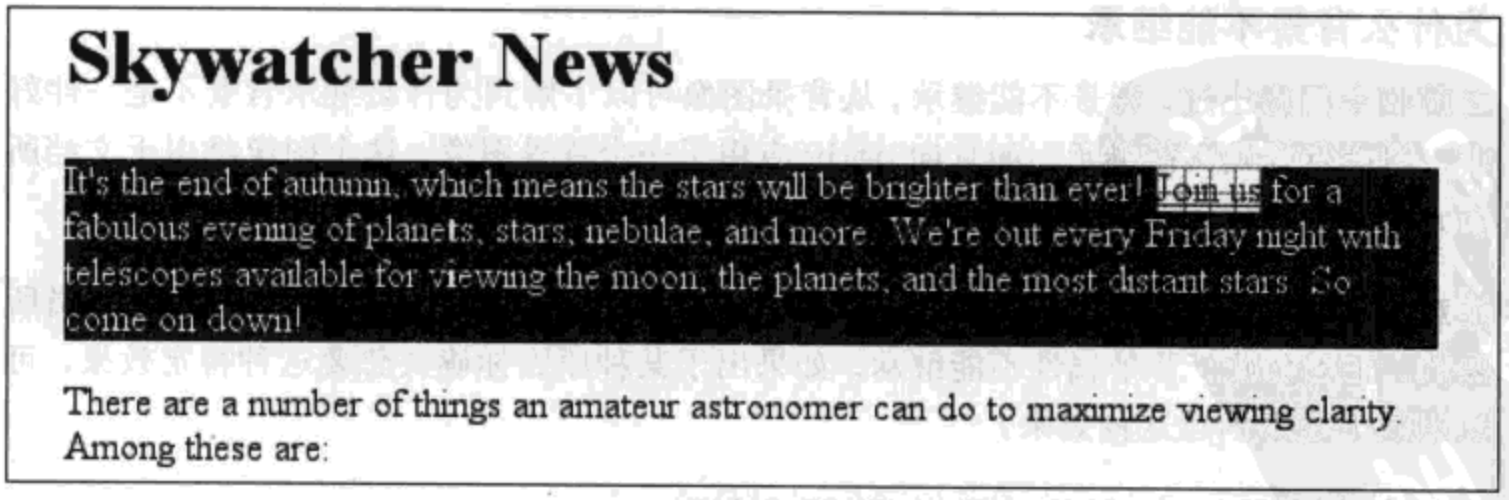

图9-13：为块级元素和行内元素应用背景图像

有很多方法可以利用特定的背景图像。可以把图像放在strong元素的背景中，使之更

突出。可以用波浪模式或小圆点填充标题的背景。甚至可以在表单元格中填入模式，使之与页面中的其他部分相区别，如图 9-14 所示：

```
td.nav {background-image: url(darkgrid.gif);}
```

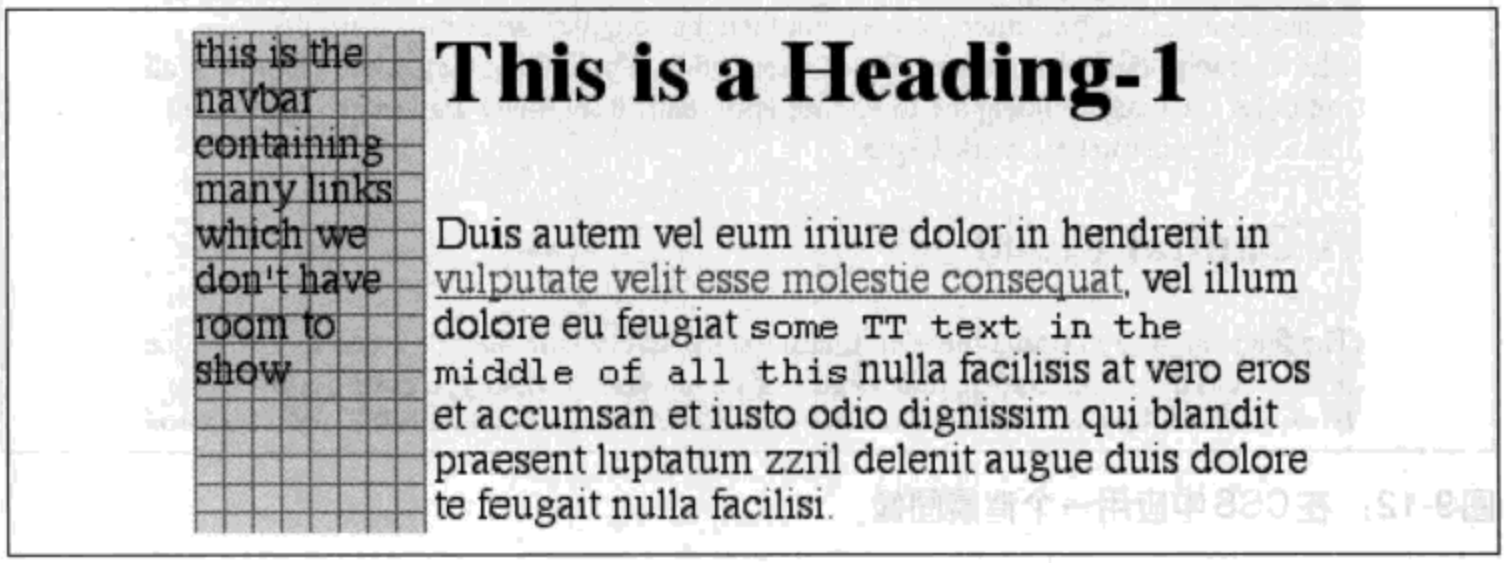

图9-14：设置表单元格的背景图像

理论上讲，甚至可以向 textareas 和 select 列表等替换元素的背景应用图像，不过并不是所有用户代理都能很好地处理这种情况。

类似于 background-color，background-image 也不能继承 —— 实际上，所有背景属性都不能继承。还要记住，指定背景图像的 URL 时，关于 url 值的限制和警告还是一如从前：相对 URL 要结合样式表来解释，不过 Navigator 4.x 不能正确地做到这一点，所以绝对 URL 可能是一个更好的选择。

为什么背景不能继承

之前我专门指出过，背景不能继承。从背景图像可以了解到为什么继承背景不是一件好事。假设背景确实能继承，而且向 body 应用了一个背景图像。这个图像将用于文档所有元素的背景，而且每个元素都完成自己的平铺，如图 9-15 所示。

注意，模式在每个元素的左上角都重新开始，也包括链接。这并不是大多数创作人员所要的，正因如此，背景属性不能继承。如果出于某种原因你确实想要这种特定效果，可以用以下规则创建这种效果：

```
* {background-image: url(yinyang.gif);}
```

或者，可以使用值 inherit，如下：

```
body {background-image: url(yinyang.gif);}
* {background-image: inherit;}
```

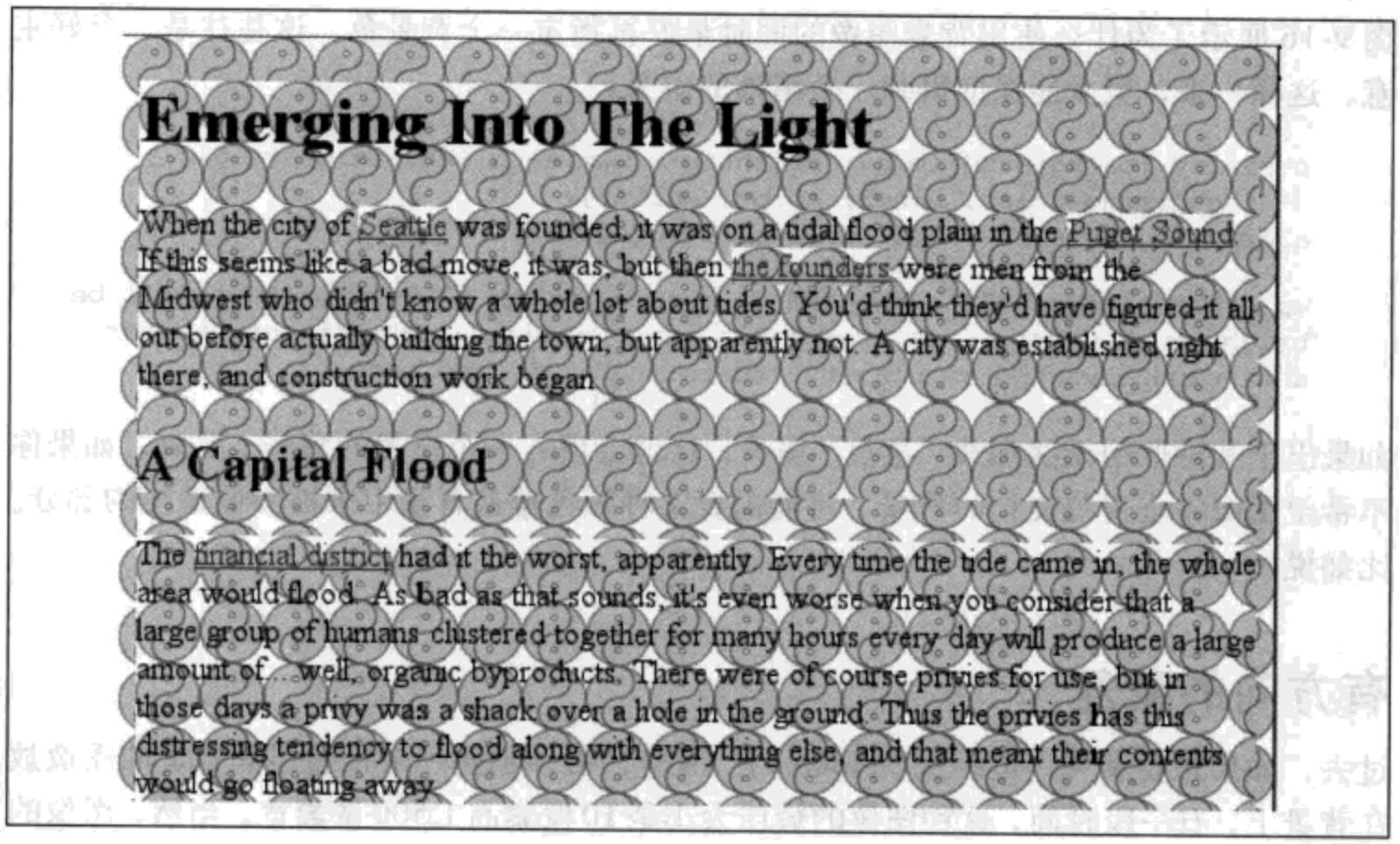

图9-15：继承背景对布局的作用

关于背景的良好实践

图像放在所指定的背景颜色之上。如果完全平铺GIF、JPEG或其他不透明的图像类型，图像在背景颜色之上并不会造成任何不同，因为完全平铺的图像会填满文档背景，可以这么说，没有地方能让颜色“透出来”。不过对于有alpha通道的图像格式，如PNG，可能会部分或完全透明，这会导致图像与背景色结合。另外，如果出于某种原因无法加载图像，用户代理就会使用指定的背景色取代图像。考虑一下，对于一个本该“布满星星的段落”，如果无法加载背景图像，它将如何显示，如图9-16所示。

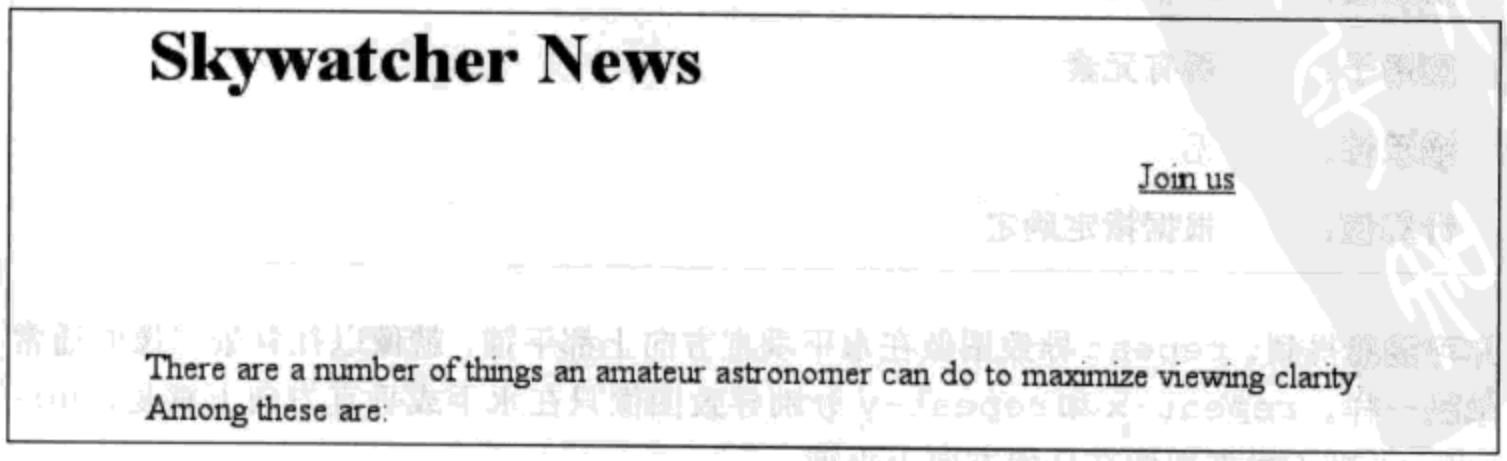

图9-16：没有背景图像的后果

图9-16展示了为什么使用背景图像的同时最好再指定一个背景色，这往往是一个好主意，这样一来，至少能保证得到一个清晰的结果：

```
p.starry {background-image: url(http://www.site.web/pix/stars.gif);
   background-color: black; color: white;}
a.grid {background-image: url(smallgrid.gif);}

<p class="starry">It's the end of autumn, which means the stars will be
brighter than ever!  <a href="join.html" class="grid">Join us</a> for
a fabulous evening of planets, stars, nebulae, and more...
```

如果出于某种原因无法加载“星星”图像，这就会填入一个全黑的背景。另外，如果你不希望图像完全覆盖文档的背景，可能需要某种颜色覆盖背景中未被图像盖住的部分。比如说……

有方向的重复

过去，如果想要某种“边栏式（sidebar）”背景，必须创建一个很短但非常宽的图像放在背景上。有一段时间，这种图像的最佳大小是10像素高1,500像素宽。当然，图像的大部分都是空白；只有左边100像素左右包含“边栏”图像。图像的余下部分基本上都被浪费了。

如果能创建一个只有10像素高100像素宽的边栏图像，而没有浪费的空白空间，然后只在垂直方向上重复这个图像，这样不是更高效吗？这肯定会让你的设计工作容易一些，而且用户的下载也会快得多。下面来看background-repeat。

background-repeat

值：	repeat | repeat-x | repeat-y | no-repeat | inherit
初始值：	repeat
应用于：	所有元素
继承性：	无
计算值：	根据指定确定

你可能猜得到，repeat导致图像在水平垂直方向上都平铺，就像以往背景图像的通常做法一样。repeat-x和repeat-y分别导致图像只在水平或垂直方向上重复，no-repeat则不允许图像在任何方向上平铺。

默认地，背景图像将会从一个元素的左上角开始（本章后面将介绍如何改变这种默认行为）。因此，以下规则将得到如图 9-17 所示的结果：

```
body {background-image: url(yinyang.gif);
   background-repeat: repeat-y;}
```

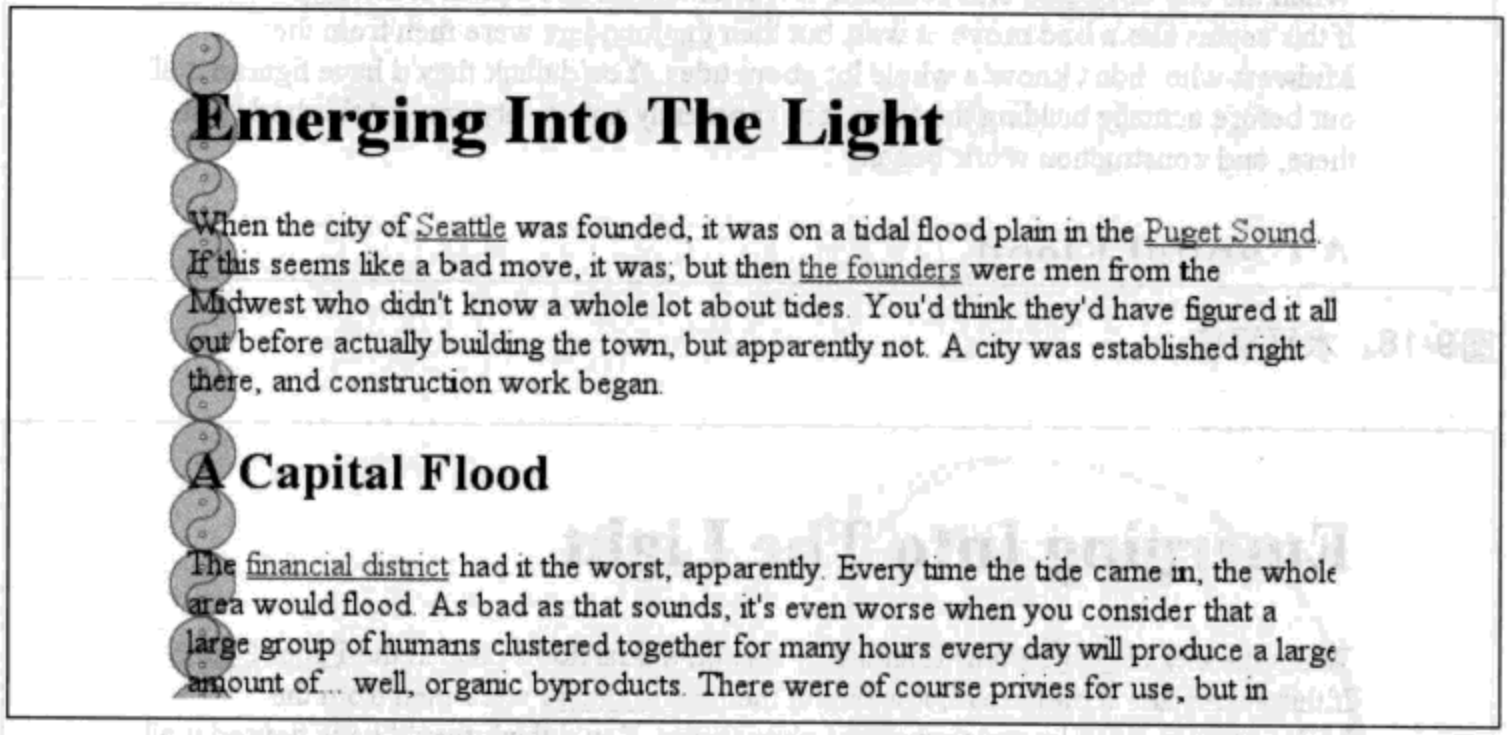

图9-17：垂直平铺背景图像

（为了让规则尽量简短，这里省略了背景颜色，不过要记住，只要有背景图像就应该同时包含一个背景颜色。）

不过，假设你希望图像在文档的顶端重复。不必为此创建一个特殊图像，使其下面全部都是空白，只需对规则做一个小改动：

```
body {background-image: url(yinyang.gif);
   background-repeat: repeat-x;}
```

如图 9-18 所示，这个图像会从其起点沿着 *x* 轴重复（也就是水平重复），在这里就是从浏览器窗口的左上角开始水平重复。

最后，你可能根本不希望重复背景图像。在这种情况下，可以使用值 no-repeat：

```
body {background-image: url(yinyang.gif);
   background-repeat: no-repeat;}
```

这个值看上去可能用处不大，因为这个声明只会把一个小图像放在文档的左上角，不过再来看一个大得多的符号，如图 9-19 所示：

```
body {background-image: url(bigyinyang.gif);
   background-repeat: no-repeat;}
```

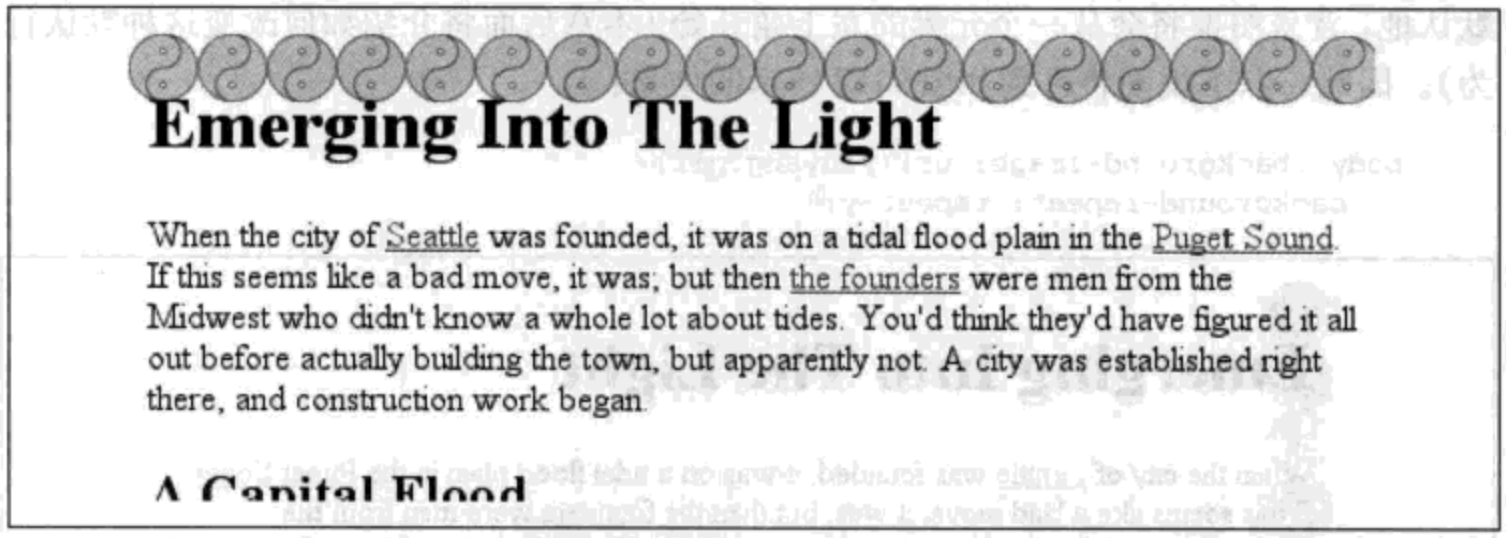

图9-18：水平平铺

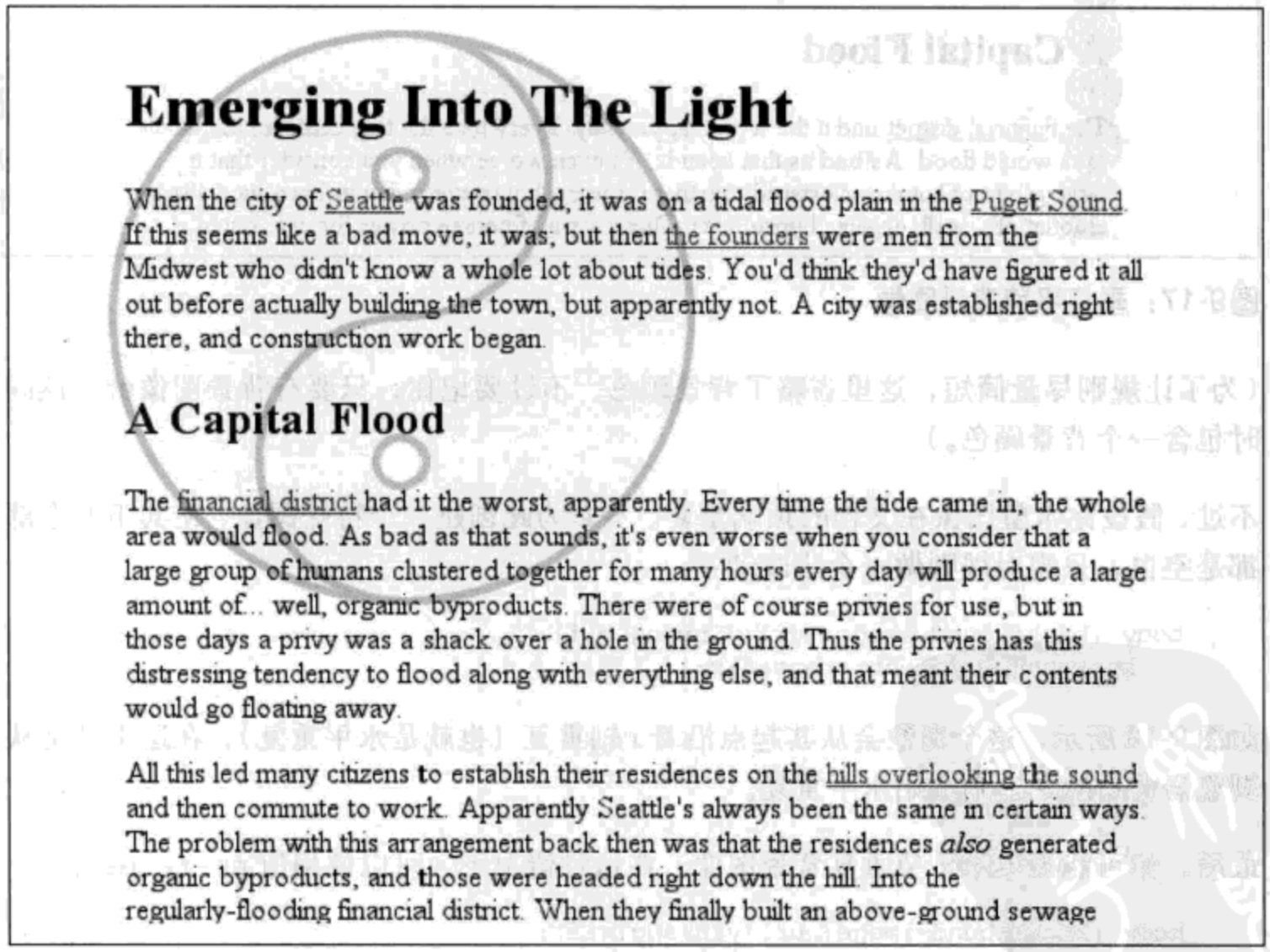

图9-19：放一个大背景图像

由于能控制重复方向，这使得可能达到的效果大大增加。例如，假设你希望文档中每个h1元素的左边有一个三重边框。可以把这个概念再进一步，在每个h2元素的顶端设置

一个波浪线边框。图像采用特定方式着色，使之能与背景色混合，得到图9-20所示的波浪效果：

```
h1 {background-image: url(triplebor.gif); background-repeat: repeat-y;}
h2 {background-image: url(wavybord.gif); background-repeat: repeat-x;
    background-color: #CCC;}
```

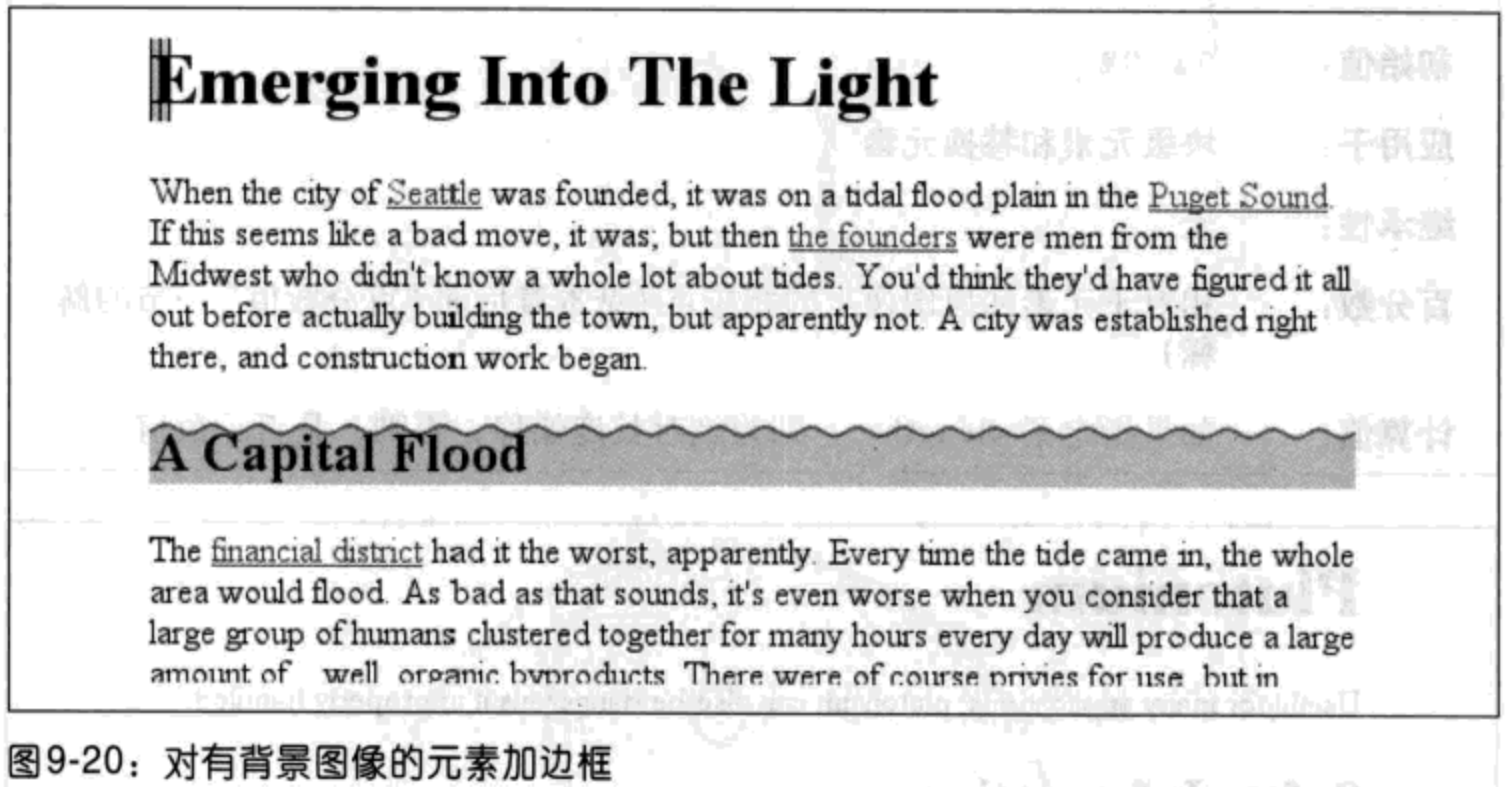

图9-20：对有背景图像的元素加边框

仅仅通过选择适当的图像，并用某种创造性的方式加以使用，就可以创建一些非常有意思的效果。还不仅如此。既然你已经知道了如何限制一个背景图像的重复，那么如何让它在背景区内移动呢？

背景定位

利用background-repeat，可以把一个大图像放在文档的背景中，然后使之一直重复。在此基础上，下面来看如何改变图像在背景中的位置。

例如，可以在body元素中将一个背景图像居中放置，其结果如图9-21所示：

```
body {background-image: url(bigyinyang.gif);
   background-repeat: no-repeat;
   background-position: center;}
```

这里在背景上放了一个图像，然后使用值no-repeat使之不能重复。每个包含图像的背景都从一个图像开始，再根据background-repeat 的值重复（或不重复）。这个起点称为原图像（origin image）。

background-position

值：	[[<percentage> | <length> | left | center | right] [<percentage>] | <length> | top | center | bottom]?] || [[left | center | right] || [top | center | bottom]] | inherit
初始值：	0% 0%
应用于：	块级元素和替换元素
继承性：	无
百分数：	相对于元素和原图像上的相应点（见本章后面“百分数值”一节的解释）
计算值：	如果指定了<length>，则为绝对长度偏移；否则，是百分数值

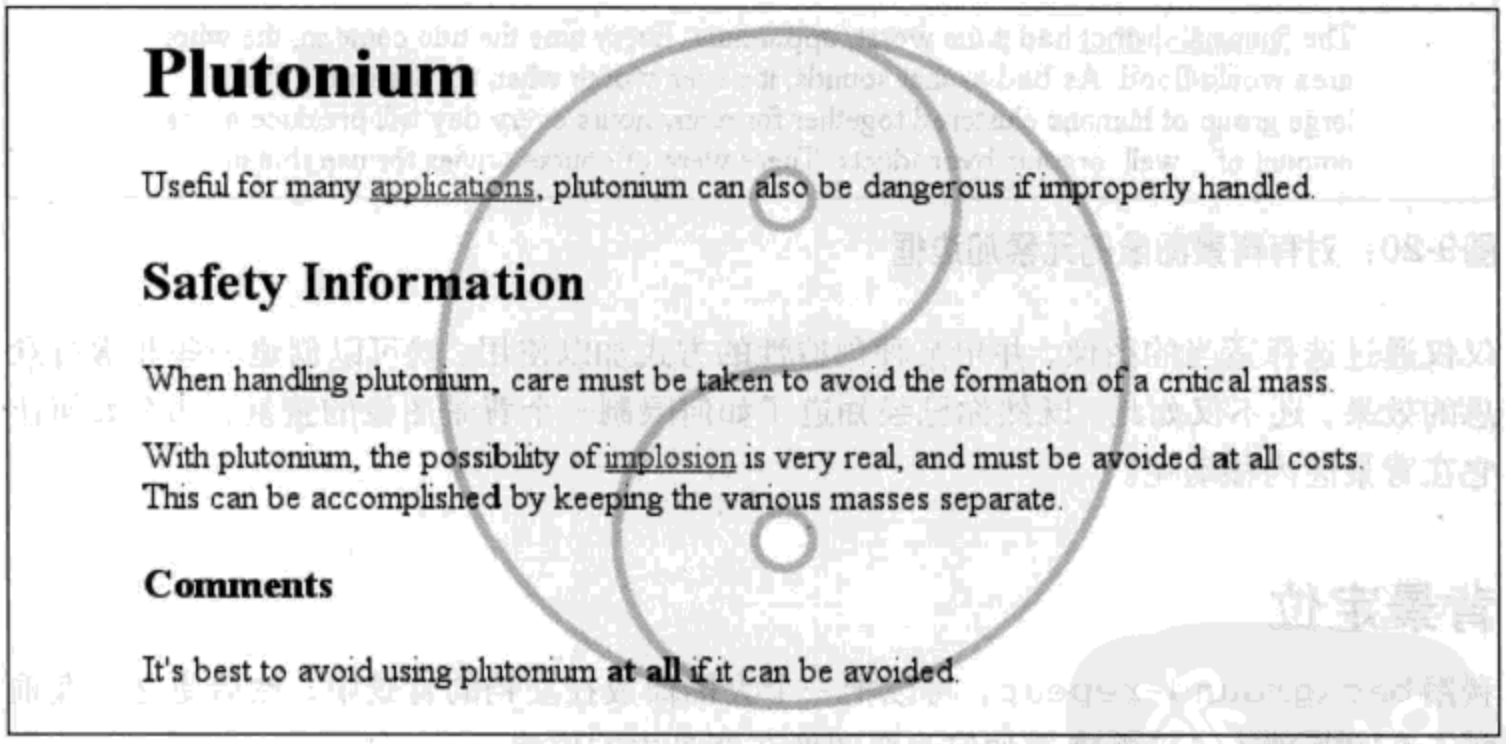

图9-21：背景图像居中

原图像的放置由background-position属性决定，为这个属性提供值有很多方法。首先，可以使用一些关键字：top、bottom、left、right和center。通常，这些关键字会成对出现，不过（如前面的例子所示）也并不总是这样。还可以使用长度值，如50px或2cm，最后也可以使用百分数值。不同类型的值对于背景图像的放置稍有差别。

还要提到一点：这就是放置背景图像的上下文。CSS2和CSS2.1指出，根据background-position，将相对于元素的内边距边界放置原图像。换句话说，放置图像的上下文是内边框边界，尽管背景区会延伸到外边框边界。并非所有浏览器都能正确地放置图像；有

些浏览器就会相对于外边框边界而不是内边框边界来放置原图像。不过如果没有边框，无论是哪一种浏览器，效果都是一样的。

注意：如果有人想了解多年来CSS有什么改变，可以告诉你，CSS1相对于内容区定义图像的放置。

尽管有图像放置的上下文，不过完全平铺的背景图像确实会填充边框区的背景，因为平铺图像会在4个方向上延伸。稍后会更详细地讨论这一点。首先，需要了解原图像在元素中如何定位。

关键字

图像放置关键字最容易理解。其作用如其名所表明的；例如，`top right`使原图像放在元素内边距区的右上角。再回过头来看先前的那个小“阴阳”符号：

```
p {background-image: url(yinyang.gif);
   background-repeat: no-repeat;
   background-position: top right;}
```

这会在每个段落内边距右上角放置一个不重复的原图像。其结果（如图9-22所示）与将位置声明为`right top`是一样的。（根据规范）位置关键字可以按任何顺序出现，只要保证不能超过两个关键字——一个对应水平方向，另一个对应垂直方向。

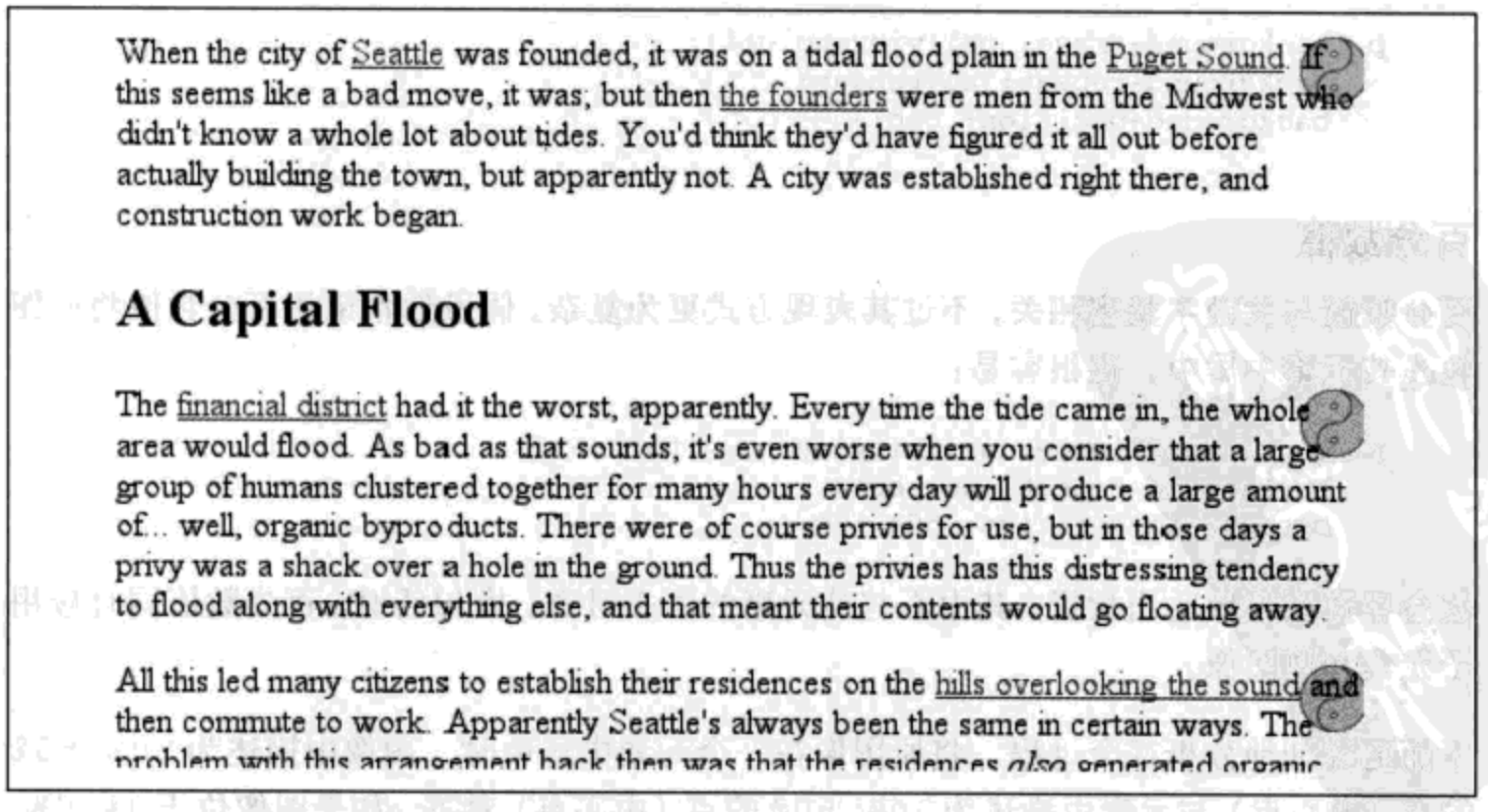

图9-22：在段落右上角放置背景图像

警告： Netscape 6.x系列有一个bug，如果规则中background-position关键字采用了某种特定顺序，这个bug会使浏览器忽略这条规则。为了避免这个bug，要确保关键字先指定水平放置，然后是垂直放置。因此，应当写作 left center 而不是center left。

如果只出现了一个关键字，则认为另一个关键字是center。表9-1显示了等价的关键字。

表9-1：等价的位置关键字

单个关键字	等价关键字
center	center center
top	top center center top
bottom	bottom center center bottom
right	center right right center
left	center left left center

所以，如果希望每个段落的中部上方出现一个图像，只需如下声明：

```
p {background-image: url(yinyang.gif);
   background-repeat: no-repeat;
   background-position: top;}
```

百分数值

百分数值与关键字紧密相关，不过其表现方式更为复杂。假设你希望用百分数值将原图像在其元素中居中。这很容易：

```
p {background-image: url(bigyinyang.gif);
   background-repeat: no-repeat;
   background-position: 50% 50%;}
```

这会导致原图像适当放置，其中心与其元素的中心对齐。换句话说，百分数值同时应用于元素和原图像。

下面更详细地分析这个过程。将原图像在一个元素中居中时，图像中描述为50% 50%的点（中心点）与元素中描述为50% 50%的点（中心点）对齐。如果图像位于0% 0%，

其左上角将放在元素内边距区的左上角。如果图像位置是100% 100%，会使原图像的右下角放在内边距区的右下角：

```
p {background-image: url(oransqr.gif);
   background-repeat: no-repeat;
   padding: 5px;  border: 1px dotted gray;}
p.c1 {background-position: 0% 0%;}
p.c2 {background-position: 50% 50%;}
p.c3 {background-position: 100% 100%;}
p.c4 {background-position: 0% 100%;}
p.c5 {background-position: 100% 0%;}
```

图 9-23 展示了这些规则的效果。

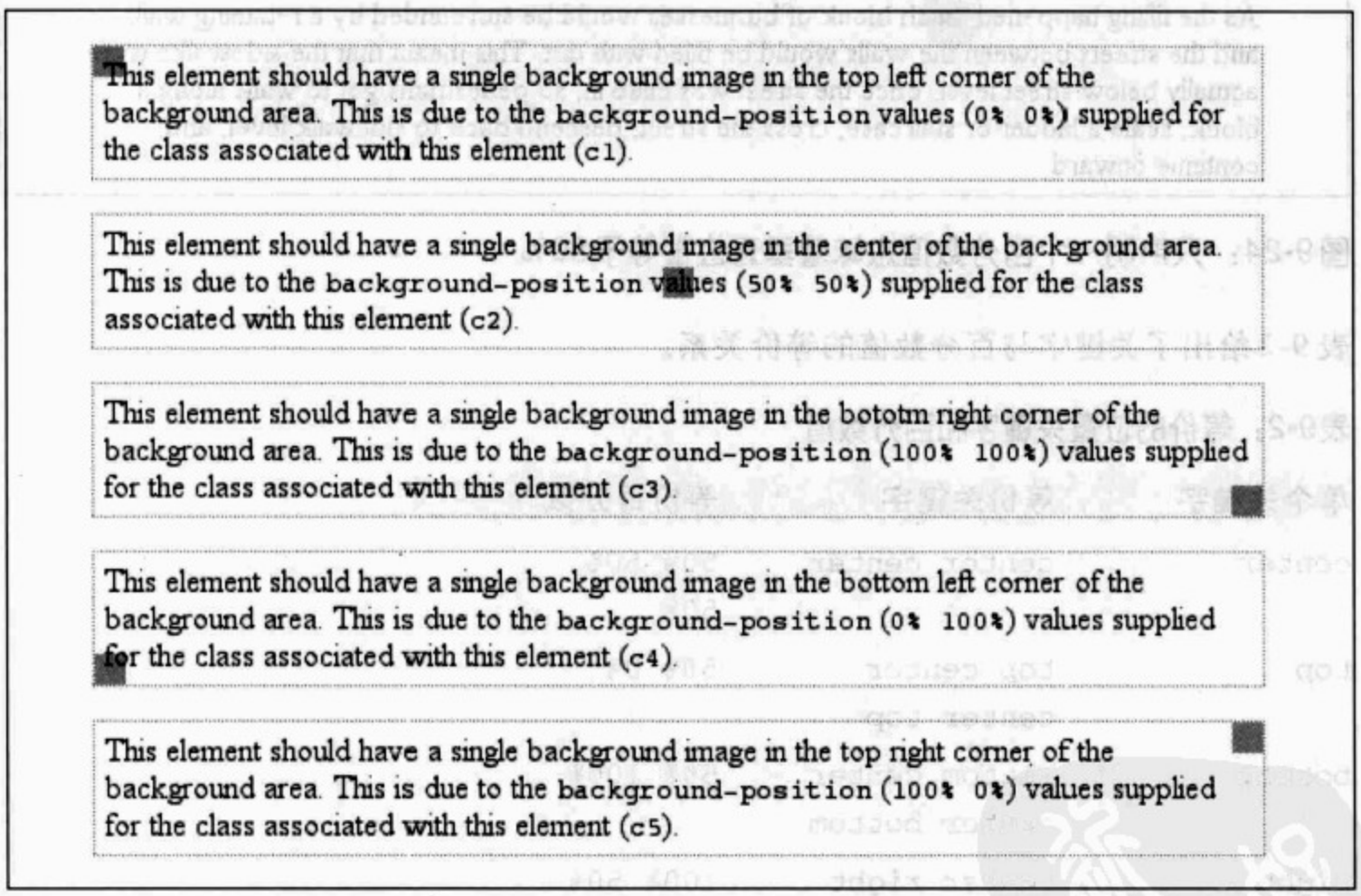

图9-23：百分数设置的各种位置

因此，如果你想将一个原图像放在水平方向 1/3、垂直方向 2/3 处，可以声明如下：

```
p {background-image: url(bigyinyang.gif);
   background-repeat: no-repeat;
   background-position: 33% 66%;}
```

利用这些规则，原图像中从图像左上角水平向右1/3、垂直向下2/3处的点将与离包含元素左上角最远的点对齐。注意，如果用百分数设置位置，水平值总是先出现。如果在上例中将百分数值的顺序换一下，图像将放在元素中水平向右 2/3、垂直向下 1/3 处。

如果只提供了一个百分数值，所提供的这个值将用作为水平值，垂直值假设为50%。这与关键字类似，即如果只指定了一个关键字，另一个关键字则假设为center。例如：

```
p {background-image: url(yinyang.gif);
   background-repeat: no-repeat;
   background-position: 25%;}
```

原图像位于元素内容区和内边距区水平向右1/4、垂直向下1/2处，如图9-24所示。

What they did seems bizarre, but it worked. The merchants rebuilt their businesses right away (using stone and brick this time instead of wood), as they had to do. In the meantime, the project to raise the financial district went ahead more or less as planned, but with one modification. Instead of filling in the whole area, the *streets* were raised to the desired level. As the filling happened, each block of businesses would be surrounded by a retaining wall, and the streets between the walls would be filled with dirt. This meant that the sidewalks were actually below street level, once the street was filled in, so pedestrians got to walk along a block, scale a ladder or staircase, cross the street, descend back to sidewalk level, and continue onward.

图9-24：只声明一个百分数值意味着垂直位置等于50%

表9-2给出了关键字与百分数值的等价关系。

表9-2：等价的位置关键字和百分数值

单个关键字	等价关键字	等价百分数
center	center center	50% 50%
	50%	
top	top center	50% 0%
	center top	
bottom	bottom center	50% 100%
	center bottom	
right	center right	100% 50%
	right center	100%
left	center left	0% 50%
	left center	0%
	top left	0% 0%
	left top	
	top right	100% 0%
	right top	
	bottom right	100% 100%
	right bottom	

表9-2：等价的位置关键字和百分数值（续）

单个关键字	等价关键字	等价百分数
	bottom left	0% 100%
	left bottom	

你可能会奇怪，background-position的默认值是0% 0%，这在功能上就相当于top left。正是因为这个原因，背景图像总是从元素内边距区的左上角开始平铺，除非你设置了不同的位置值。

长度值

最后，我们来看定位的长度值。在为原图像的位置提供长度值时，这些长度值将解释为从元素内边距区左上角的偏移。偏移点是原图像的左上角；因此，如果设置值为20px 30px，原图像的左上角将在元素内边距区左上角向右20像素、向下30像素的位置上，如图9-25所示：

```
p {background-image: url(yinyang.gif);
   background-repeat: no-repeat;
   background-position: 20px 30px;
   border: 1px dotted gray;}
```

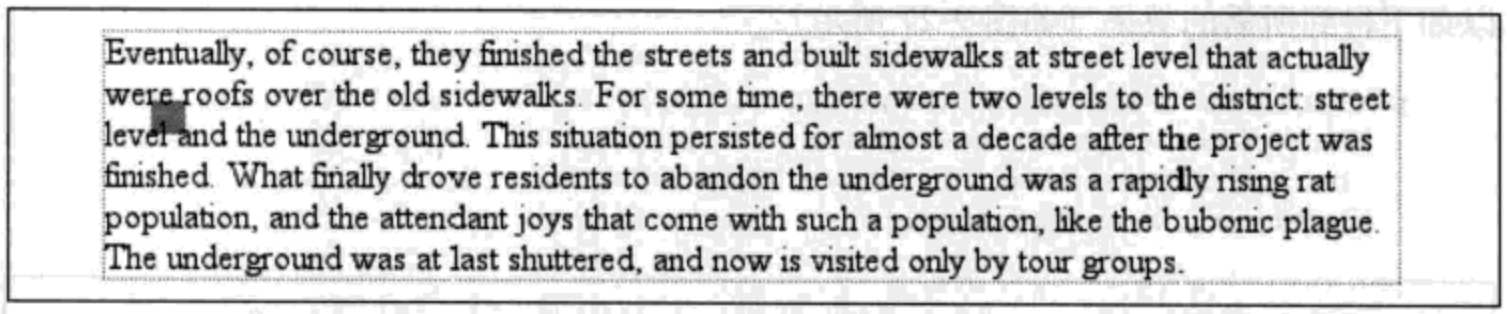

图9-25：使用长度度量设置背景图像的偏移

这与百分数值大不相同，因为偏移只是从一个左上角到另一个左上角的偏移。换句话说，原图像的左上角与background-position声明中指定的点对齐。不过，可以结合使用长度和百分数值，得到“二者兼俱”的效果。假设你需要一个背景图像，它要一直延伸到元素的右边，并且要从顶部向下延伸10像素，如图9-26所示。像以往一样，先指定水平值：

```
p {background-image: url(bg23.gif);
   background-repeat: no-repeat;
   background-position: 100% 10px;
   border: 1px dotted gray;}
```

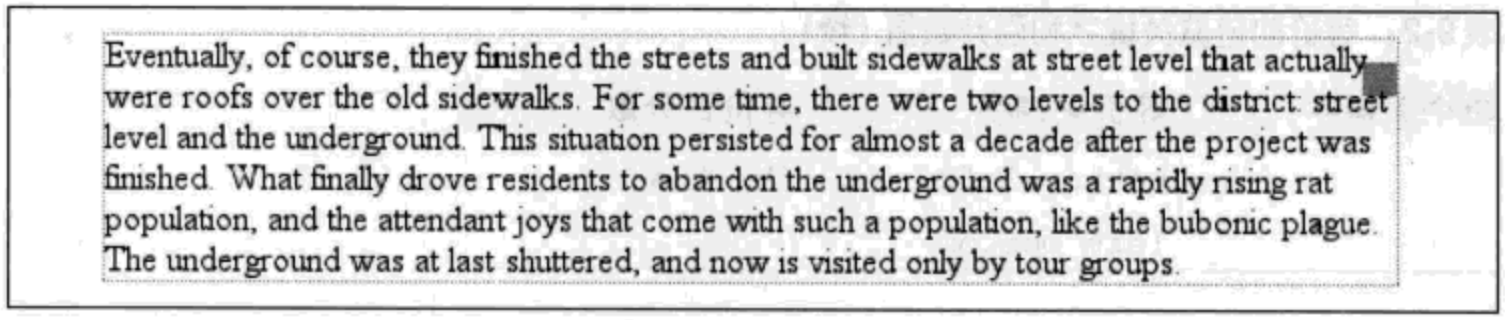

图9-26：结合使用百分数值和长度值

警告： 在CSS 2.1之前的版本中，不能将关键字与其他值混合使用。因此，`top 75%`是非法的，如果使用了一个关键字，就只能一直使用关键字。CSS2.1为了让创作人员更容易地设计布局，且考虑很多其他浏览器已经支持这种功能，所以在这方面有所改变，允许关键字与其他值混用。

如果使用长度值或百分数值，可以使用负值将原图像拉出元素的背景区。考虑一个例子，使用一个很大的“阴阳”符号作为背景。有时可能要将其居中，不过如果你只希望其中一部分在元素内边距区的左上角可见，该怎么做呢？没问题，至少从理论上讲是可以做到的。

首先，假设原图像为300像素高300像素宽。再假设只有其右下方1/3的部分可见。可以如下得到所需的效果（如图9-27所示）：

```
p {background-image: url(bigyinyang.gif);
   background-repeat: no-repeat;
   background-position: -200px -200px;
   border: 1px dotted gray;}
```

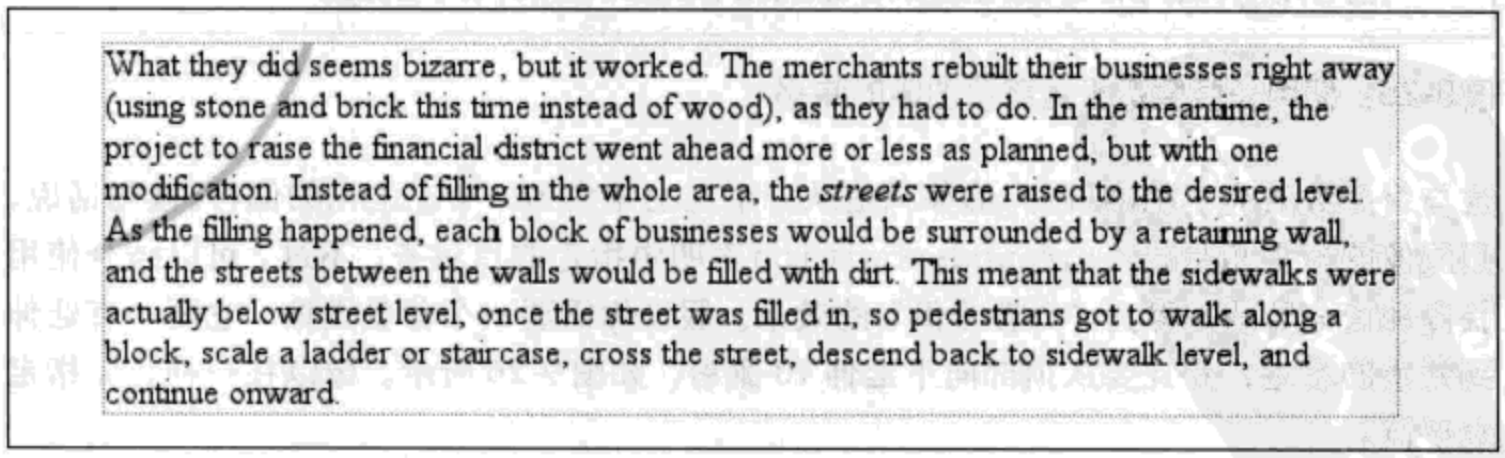

图9-27：使用负长度值定位原图像

或者，假设你只希望原图像的右半部分可见，并在元素中居中，可以如下设置规则：

```
p {background-image: url(bigyinyang.gif);
```

```
    background-repeat: no-repeat;
    background-position: -150px 50%;
    border: 1px dotted gray;}
```

理论上负百分数值也是允许的，不过对此存在两个问题。第一个问题是用户代理可能有限制，无法识别负的background-position值。另一个问题是，负百分数值计算起来很有意思。比方说，原图像和元素很可能大小不同，而这会导致意想不到的后果。例如，考虑以下规则，其结果见图9-28：

```
p {background-image: url(pix/bigyinyang.gif);
   background-repeat: no-repeat;
   background-position: -10% -10%;
   border: 1px dotted gray;
   width: 500px;}
```

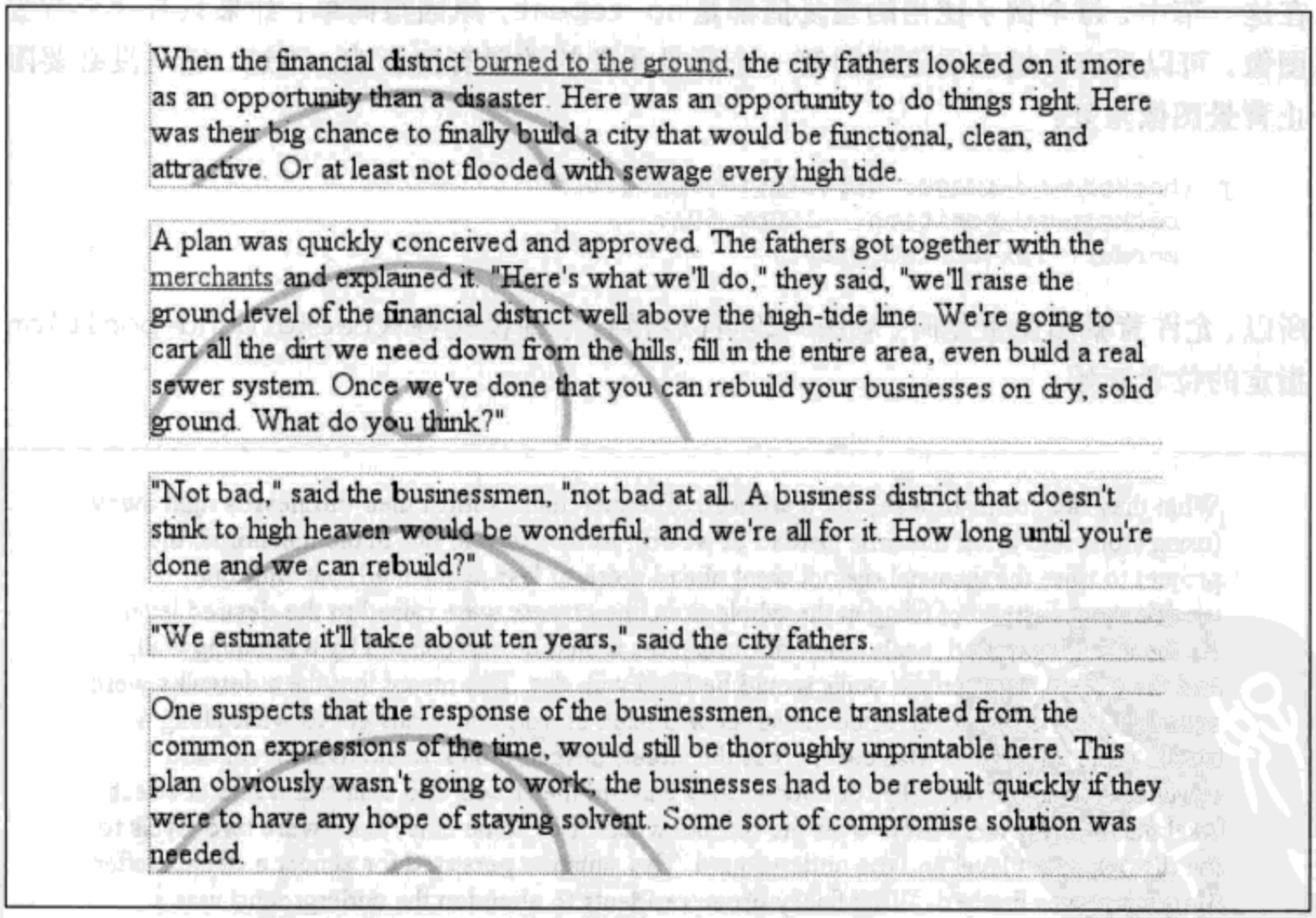

图9-28：负百分数值的不同效果

这个规则要求原图像外由-10% -10%定义的点与各段落中同样定义为-10% -10%的一个点对齐。图像的宽度和高度都是300像素，所以我们知道，其对齐点可以描述位于图像顶部之上30像素、左边界再向左30像素的位置（也就是-30px ×-30px）。段落元素的宽度都相等（500px），所以水平对齐点是其内边距区左边界再向左50像素处。

这说明，每个原图像的左边界将在段落左内边距边界向左20像素的位置。这是因为，图像的 -30px 对齐点与段落的 -50px 点对齐。二者之间相差 20 像素。

不过，各段落的高度不同，所以每个段落的垂直对齐点都不同。半随机地选择一个例子，如果一个段落为300像素高，原图像的顶端将与元素内边距区的顶端对齐，因为二者的垂直对齐点都是 -30px。如果一个段落高度为50像素，其对齐点将是 -5px，相应地，原图像的顶端实际上将在内边距区顶端向下25像素处。

正百分数值也可能出现同样的问题（可以想象一下，如果将原图像与比该图像矮的一个元素的底端对齐，会发生什么情况），所以，以上介绍并不是说不应该使用负值，而只是提醒你往往存在一些要考虑的问题。

在这一节中，每个例子使用的重复值都是no-repeat。原因很简单：如果只有一个背景图像，可以更容易地查看定位对第一个背景图像的放置有何影响。不过，完全没必要阻止背景图像重复：

```
p {background-image: url(bigyinyang.gif);
   background-position: -150px 50%;
   border: 1px dotted gray;}
```

所以，允许背景图像重复时，从图9-29可以看到，平铺模式将从background-position指定的位置开始。

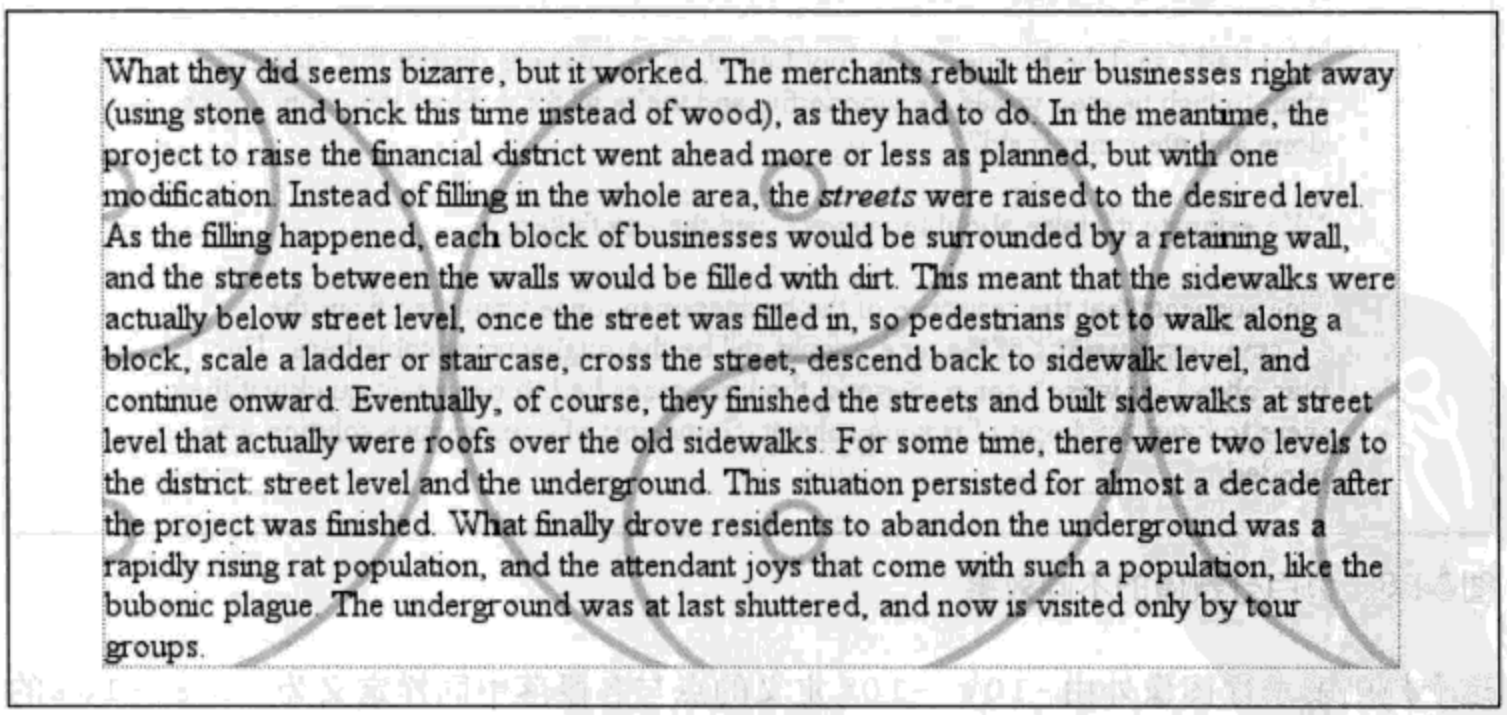

图9-29：使用background-position属性设置平铺模式的起点

这再一次说明了原图像的概念，对于理解下一节非常重要。

有方向的重复（深入）

在前面关于重复的一节中，我们介绍了值repeat-x、repeat-y和repeat，并说明了它们对背景图像的平铺有何影响。不过，对于这些值，平铺模式总是从包含元素（例如p）的左上角开始。当然，并不要求如此；我们已经看到，background-position的默认值是0% 0%。所以，除非改变原图像的位置，否则平铺就会从左上角开始。不过，既然你知道了如何改变原图像的位置，接下来需要了解用户代理如何处理这种情况。

要介绍这个内容，最容易的办法就是先提供一个例子，再作相应的解释。考虑以下标记，其结果如图9-30所示：

```
p {background-image: url(yinyang.gif);
   background-position: center;
   border: 1px dotted gray;}
p.c1 {background-repeat: repeat-y;}
p.c2 {background-repeat: repeat-x;}
```

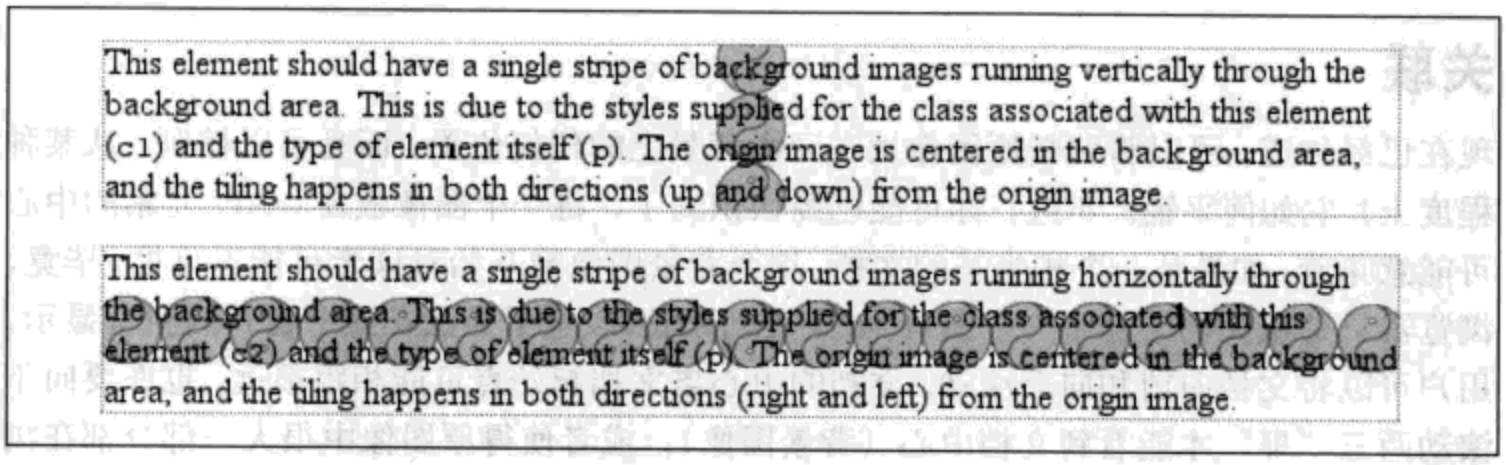

图9-30：居中原图像并重复

可以看到结果：“图条”穿过了元素的中心。看上去可能不对，但这并没有错。

图9-30所示的例子是正确的，因为原图像放在第一个p元素的中心，然后沿着y轴在两个方向上平铺；换句话说，同时向上和向下平铺。对于第二个段落，图像则分别向右和向左重复。

因此，将一个大图像设置在p 的中心，再让它充分重复，将导致它在4个方向上都平铺，即向上、向下、向左和向右。background-position造成的唯一差别是确定平铺从哪里开始。图9-31显示了从元素中心平铺和从元素左上角平铺的差别。

注意元素各边界上的差别。当背景从中心重复时（如第一段中），阴阳符号网格在元素内居中，这会在各边界上得到一致的“剪裁”效果。在第二段中，平铺从内边距区左上角开始，所以剪裁是不一致的。看上去差别可能很小，不过在你的设计生涯中，这两种方法都很可能需要用到。

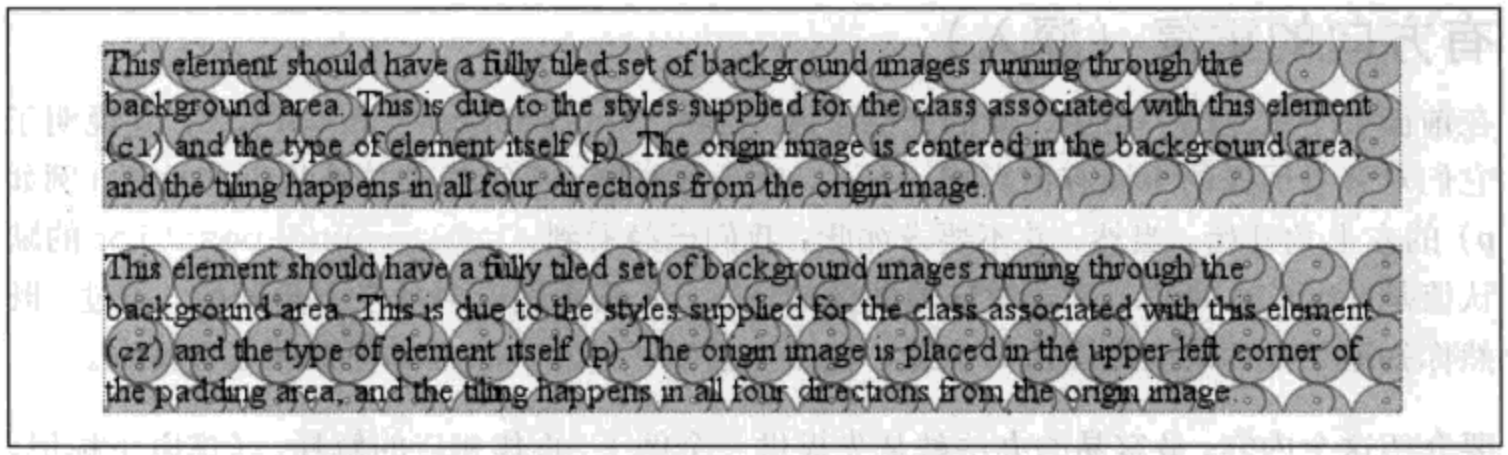

图9-31：从中心重复和从左上角重复的差别

如果你还想知道其他控制重复的方法，需要指出，除了前面讨论的再没有别的方法了。例如，不存在 `repeat-left`，不过可能CSS的某个将来版本中会增加这样的值。对现在来说，全部方法就是这些，只能水平平铺、垂直平铺，或者根本不平铺。

关联

现在已经知道，可以将背景原图像放在元素背景上的任何位置，而且可以控制（从某种程度上）它如何平铺。不过，你可能已经认识到了，将一个图像放在 `body` 元素的中心可能意味着：如果是一个相当长的文档，这个背景图像最开始对读者可能不可见。毕竟，浏览器只是在文档上开了一个窗口。如果文档太长，以至于无法在这个窗口内完全显示，用户可以将文档向前和向后滚动。文档的中心离文档起始点可能相距很远，也许要向下滚动两三“屏”才能看到文档中心（背景图像），或者使得原图像中很大一部分都在浏览器窗口底端之外。

此外，即使你认为原图像开始时是可见的，它往往会随文档滚动，而且每次用户滚动如果超过了图像的位置，原图像就会消失。不用担心：有办法防止这种滚动。

background-attachment

值：	`scroll | fixed | inherit`
初始值：	`scroll`
应用于：	所有元素
继承性：	无
计算值：	根据指定确定

通过使用属性background-attachment，可以声明原图像相对于可视区是固定的(fixed)，因此不会受滚动的影响：

```
body {background-image: url(bigyinyang.gif);
   background-repeat: no-repeat;
   background-position: center;
   background-attachment: fixed;}
```

这样做有两个直接后果，如图9-32所示。首先，原图像不会随文档滚动。其次，原图像的放置由可视区的大小确定，而不是由包含该图像的元素的大小(或在可视区中的位置)决定。

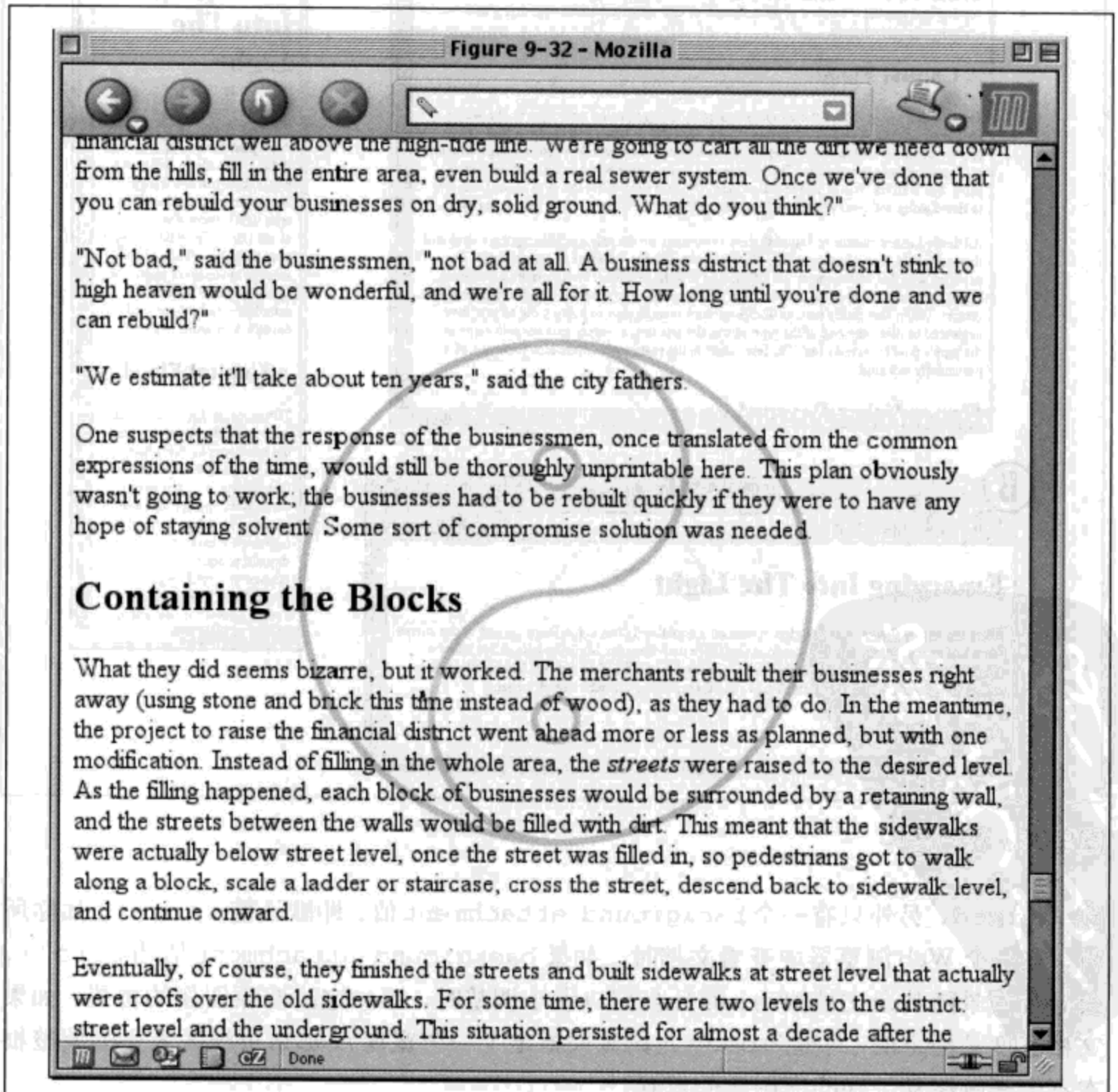

图9-32：固定背景

在一个Web浏览器中，随着用户调整浏览器窗口的大小，可视区可能会改变。这会导致背景的原图像随着窗口大小的改变移动位置。图9-33显示了同一个文档的多个视图。所以从某种意义上说，图像并不是固定的，它只是在可视区大小不改变的情况下保持固定。

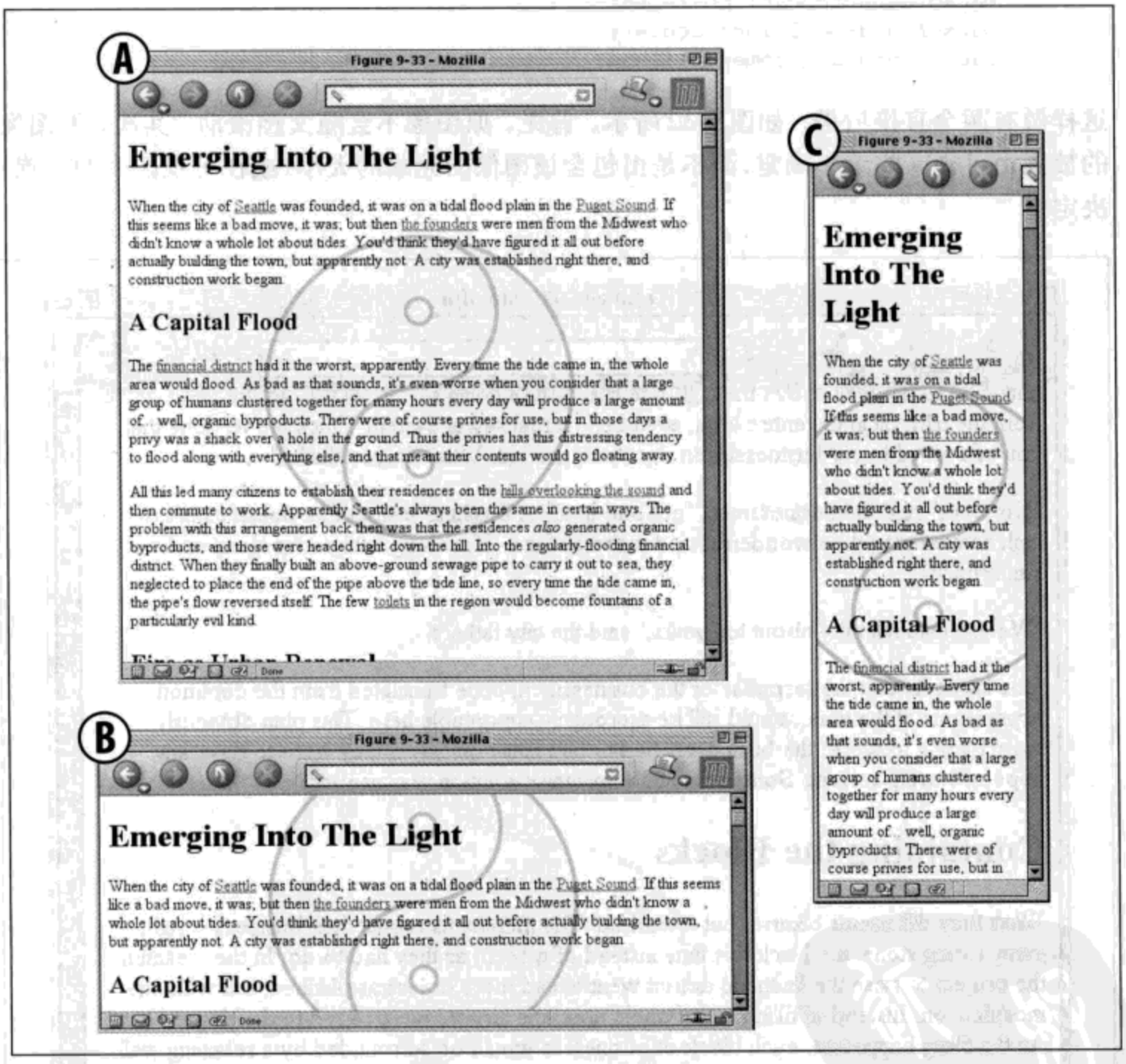

图9-33：保持居中

除了fixed，另外只有一个background-attachment值，即默认值scroll。如你所料，在一个Web浏览器中查看文档时，如果background-attachment值为scroll，会导致背景随其余文档滚动，而且当窗口大小调整时，不一定改变原图像的位置。如果文档宽度是固定的（可能为body元素显式指定了一个宽度），那么可视区大小的调整根本不会影响滚动关联的原图像的放置。

有意思的效果

从技术上讲，如果一个背景图像已经固定（fixed），它会相对于可视区定位，而不是相对于包含该图像的元素定位。不过，背景将只在其包含元素中可见。这带来一个很有意思的后果。

假设有一个文档，其中有一个看上去像是平铺的砖块背景，还有一个有相同模式的h1元素，只不过颜色不同。body和h1元素都设置为有固定（fixed）背景，这会得到如图9-34所示的结果：

```
body {background-image: url(grid1.gif); background-repeat: repeat;
   background-attachment: fixed;}
h1 {background-image: url(grid2.gif); background-repeat: repeat;
   background-attachment: fixed;}
```

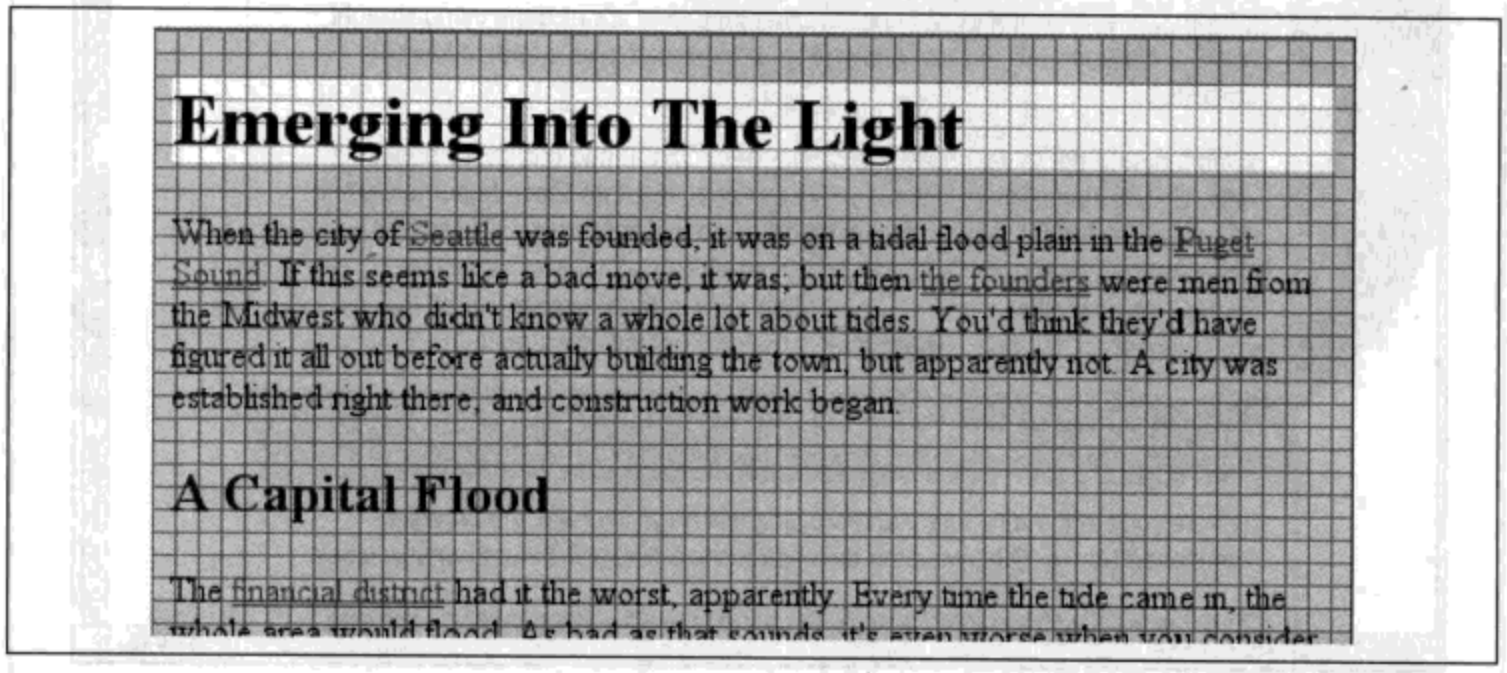

图9-34：背景的理想对齐

这种理想的对齐是怎么做到的呢？要记住，一个背景如果是固定的（fixed），原元素会根据视窗定位。因此，背景模式都从视窗的左上角开始平铺，而不是从单个元素的左上角开始。对于body，可以看到整个重复模式。不过，对于h1，只是在h1本身的内边距和内容区能看到它的背景。由于两个背景图像大小相同，而且它们有相同的起点，所以看上去就会像图9-34那样“对齐”。

这种功能可以用来创建一些非常复杂的效果。其中最有名的一个例子是“复螺旋变形”演示（*http://www.meyerweb.com/eric/css/edge/complexspiral/glassy.html*），见图9-35所示。

这种视觉效果是通过为非body元素指定不同的固定关联背景图像生成的。整个演示由一个HTML文档、4个JPEG图像和一个样式表驱动。由于所有这4个图像都位于浏览

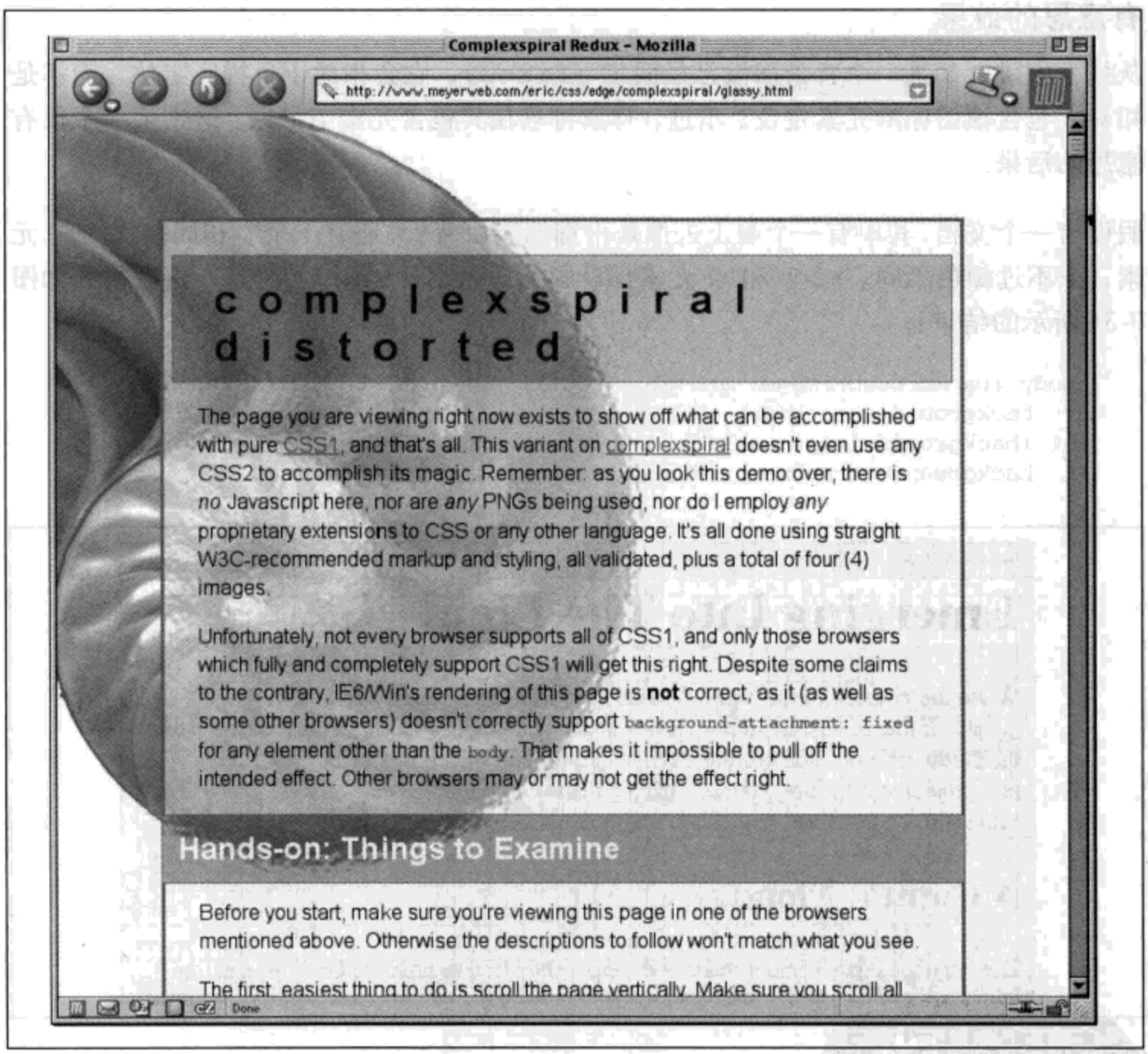

图9-35：复螺旋变形

器窗口的左上角，但只是与其元素相交的部分可见，这些图像实际上会交叠，产生一种半透明波纹玻璃的感觉。

警告： Windows平台的Internet Explorer在IE6以前还不能正确地处理非body元素上的固定关联背景。换句话说，body元素上的固定背景可以得到预想的效果，但其他元素不行。所以，可能得不到如图9-34和图9-35所示的效果。Internet Explorer 7对所有元素都支持固定关联背景。

还有一种情况，对于分页媒体（如打印输出），每个页面都生成自己的视窗。因此，固

定关联背景在打印输出的每一页上都应当显示。这可以用于实现某些效果，例如，对文档中的所有页面加水印。对此存在两方面问题：一来使用CSS无法强制打印背景图像，二来并非所有浏览器都能适当地处理固定关联背景的打印。

汇总

与字体属性一样，背景属性可以汇总到一个简写属性：background。这个属性可以从各个其他背景属性取一个值，而且可以采用任何顺序。

background

值：	[<background-color> || <background-image> || <background-repeat> || <background-attachment> || <background-position>] | inherit
初始值：	根据单个属性
应用于：	所有元素
继承性：	无
百分数：	<background-position> 允许的值
计算值：	见单个属性

因此，以下语句都等价，其效果如图9-36所示：

```
body {background-color: white; background-image: url(yinyang.gif);
   background-position: top left; background-repeat: repeat-y;
   background-attachment: fixed;}
body {background: white url(yinyang.gif) top left repeat-y fixed;}
body {background: fixed url(yinyang.gif) white top left repeat-y;}
body {background: url(yinyang.gif) white repeat-y fixed top left;}
```

实际上，对background中值的顺序有一个小小的限制：如果background-position有两个值，它们必须一起出现，而且如果这两个值是长度或百分数值，则必须按水平值在前垂直值在后的顺序。这并不奇怪，不过记住这一点很重要。

就像所有简写属性一样，如果省略了某些值，就会自动填入相应属性的默认值。因此，以下两个语句是等价的：

```
body {background: white url(yinyang.gif;}
body {background: white url(yinyang.gif) top left repeat scroll;}
```

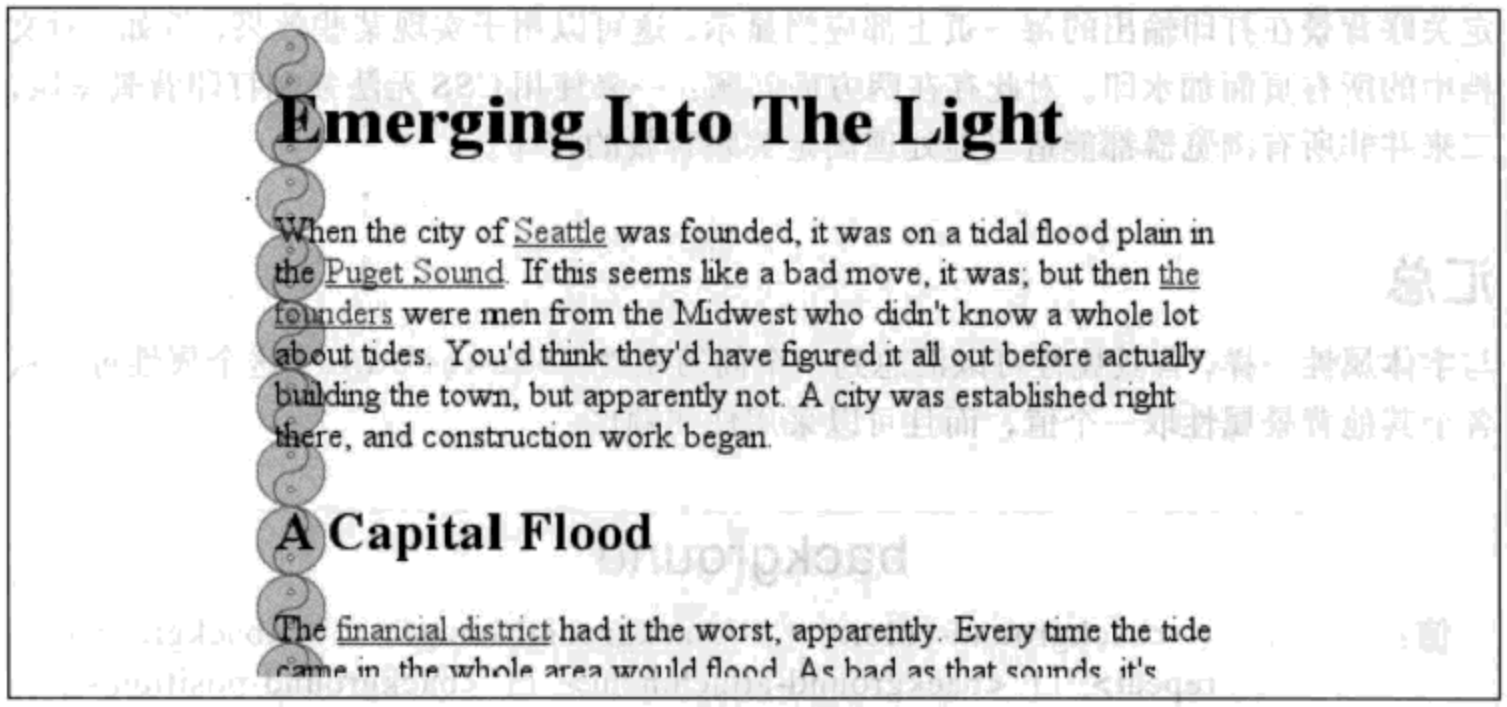

图9-36：使用简写属性

更妙的是，background没有必不可少的值 —— 只要至少出现了一个值就行，而省略所有其他属性值。因此，可以使用这个简写属性只设置背景色，这是一个很常用的做法：

```
body {background: white;}
```

这是完全合法的，而且在某些方面更为可取，这样可以减少击键次数。另外，其效果是所有其他背景属性都设置为相应的默认值，这意味着background-image将会被设置为none。这有助于确保可读性，防止其他规则（例如，读者样式表中的规则）在背景上设置图像。

以下的所有规则也是合法的，如图9-37所示：

```
body {background: url(yinyang.gif) bottom left repeat-y;}
h1 {background: silver;}
h2 {background: url(h2bg.gif) center repeat-x;}
p {background: url(parabg.gif);}
```

最后要记住一点：background是一个简写属性，因此，其默认值会覆盖先前为给定元素指定的值。例如：

```
h1, h2 {background: gray url(thetrees.jpg) center repeat-x;}
h2 {background: silver;}
```

给定上述规则，h1元素将根据第一个规则设置样式。h2元素则将根据第二个规则设置样式，这意味着它们都将有银色背景。但不会对h2背景应用任何图像，更不用说让背景图像居中和水平重复了。创作人员可能原本想这样做：

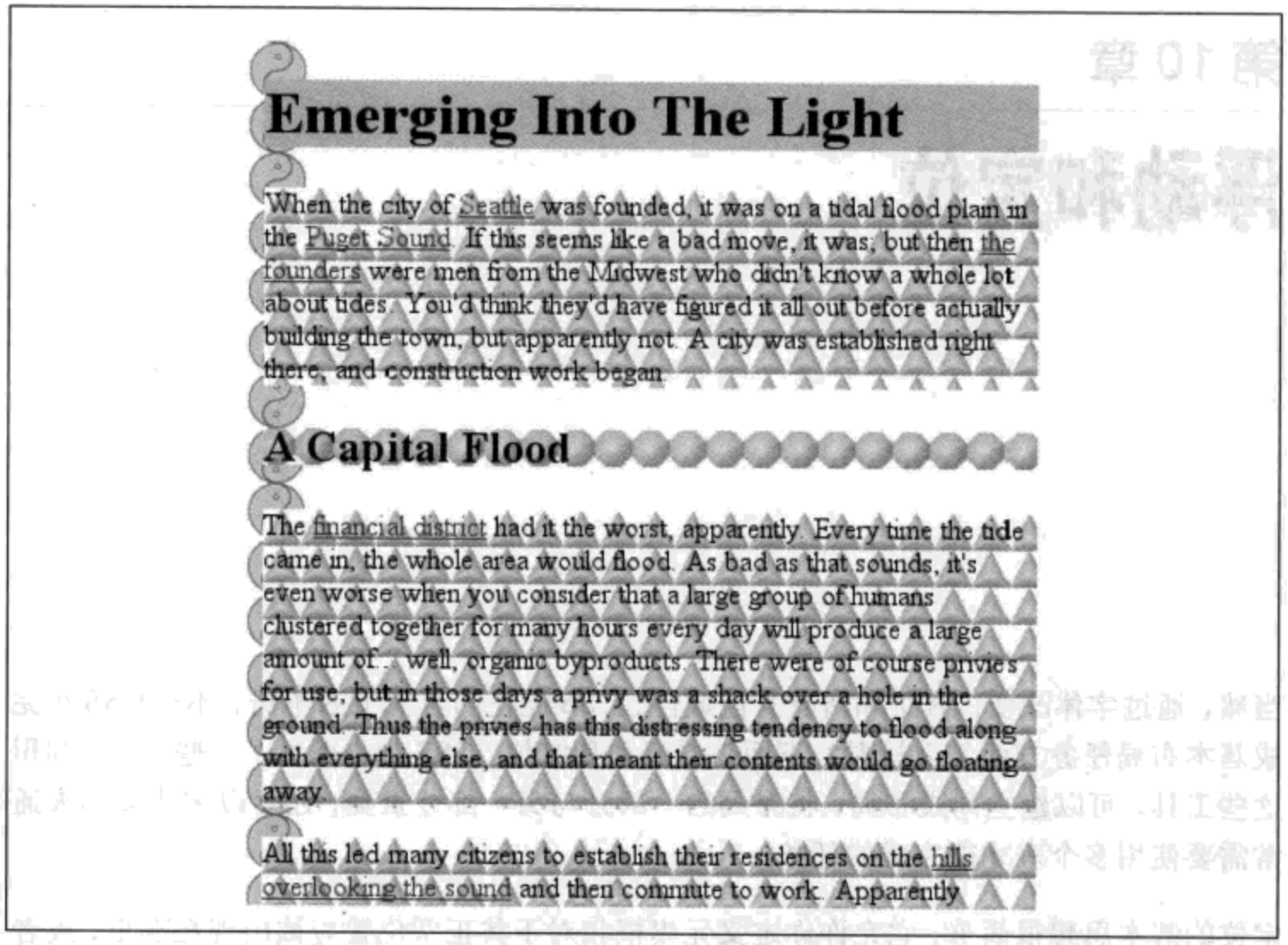

图9-37：向一个文档应用多个背景

```
h1, h2 {background: gray url(trees.jpg) center repeat-x;}
h2 {background-color: silver;}
```

这会改变背景色，而不会覆盖所有其他值。

小结

通过设置元素的颜色和背景，可以大大增强创作人员的能力。CSS超越传统方法的优点就在于可以向文档中的任何元素应用颜色和背景，而不只是表单元格或FONT标记中包含的内容。尽管某些实现中存在一些bug（如Navigator 4不能向元素的整个内容区应用背景），不过背景属性确实使用非常广泛。不难理解它们为什么这么流行，因为利用颜色可以很容易地区分不同的页面。

不过，对于如何为元素应用样式，CSS还提供了更多可能性：可以在任何元素上放置边框，可以设置额外的外边距和内边距，甚至可以“浮动”元素（不只是图像）。下一章将介绍这些概念。

第 10 章

浮动和定位

当然，通过字体改变、背景和所有其他属性，CSS能让内容看上去不错，不过CSS在完成基本布局任务方面怎么样呢？下面来看浮动和定位。CSS对此提供了一些工具，利用这些工具，可以建立列式布局，将布局的一部分与另一部分重叠，还可以完成多年来通常需要使用多个表才能完成的任务。

定位的基本思想很简单，它允许你定义元素框相对于其正常位置应该出现在哪里，或者相对于父元素、另一个元素甚至浏览器窗口本身的位置。显然这个特性功能很强大，也很让人惊叹。要知道，用户代理对CSS2中定位的支持远胜于对其他方面的支持，对此不应感到奇怪。

另一方面，CSS1中首次提出了浮动，它以Netscape在Web发展初期增加的一个功能为基础。浮动不完全是定位，不过，它当然也不是正常流布局。本章后面将明确浮动的含义。

浮动

你可能对浮动元素的概念已经有所认识。从Netscape 1以来，就可以通过声明让图像浮动，如声明`<img src="b5.gif" align="right">`。这会导致一个图像浮动到右边，而允许其他内容（如文本）“围绕”该图像。实际上，“浮动”一词源自文档“HTML 2.0的扩展”，其中指出：

> ALIGN选项有所增补，对此需要做一些解释。首先来看值“left”和“right”。可以把这些对齐方式看作是全新的浮动图像类型。

过去，只可能浮动图像（某些浏览器可能还支持表的浮动）。但CSS允许浮动任何元素，从图像到段落再到列表，所有元素都可以浮动。在CSS中，这种行为使用属性 `float` 实现。

float

值：	`left` | `right` | `none` | `inherit`
初始值：	`none`
应用于：	所有元素
继承性：	无
计算值：	根据指定确定

例如，要把一个图像浮动到左边，可以使用以下标记：

```
<img src="b4.gif" style="float: left;" alt="b4">
```

图 10-1 清楚地指出，图像“浮动”到浏览器窗口的左边，文本则围绕着该图像。这正是你所期望的。

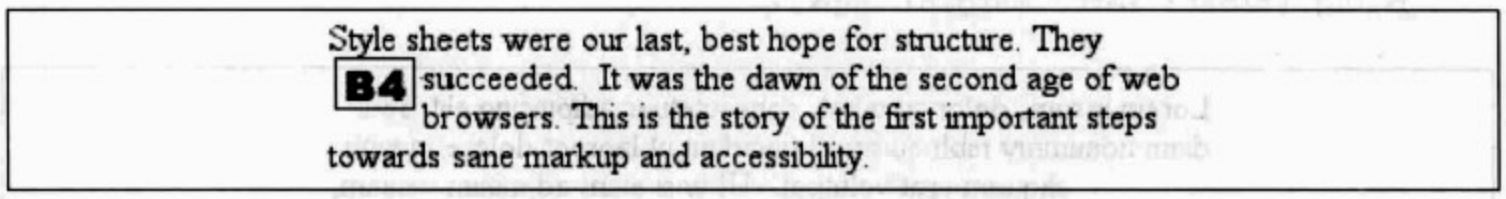

图10-1：浮动图像

不过，用CSS浮动元素时会出现一些有意思的问题。

浮动元素

对于浮动元素有几点要记住。首先，会以某种方式将浮动元素从文档的正常流中删除，不过它还是会影响布局。采用CSS的特有方式，浮动元素几乎“自成一派”，不过它们还是对文档的其余部分有影响。

这种影响源自于这样一个事实：一个元素浮动时，其他内容会“环绕”该元素。对于浮动图像来说，这种行为我们已经很熟悉了，不过要知道，元素浮动时也同样如此（例如，浮动一个段落）。在图 10-2 中，由于为浮动段落增加了外边距，可以很清楚地看到这种行为：

```
p.aside {float: right; width: 15em; margin: 0 1em 1em; padding: 0.25em;
  border: 1px solid;}
```

So we browsed the shops, buying here and there, but browsing at least every other store. The street vendors were less abundant, but *much* more persistent, which was sort of funny. Kat was fun to watch, too, as she haggled with various sellers. I don't think we paid more than two-thirds the original asking price on anything!

All of our buying was done in shops on the outskirts of the market area. The main section of the market was actually sort of a letdown, being more expensive, more touristy, and less friendly, in a way. About this time I started to wear down, so we caught a taxi back to the New Otani.

Of course, we found out later just how badly we'd done. But hey, that's what tourists are for.

图10-2：浮动段落

可以注意到一些有意思的情况，首先，浮动元素周围的外边距不会合并。如果浮动一个有20像素外边距的图像，在这个图像周围将至少有20像素的空间。如果其他元素与此图像相邻（这表示水平相邻和垂直相邻），而且这些元素也有外边距，那么这些外边距不会与浮动图像的外边距合并，如图10-3所示：

```
p img {float: left; margin: 25px;}
```

Lorem ipsum, dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Lorem ipsum, dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

图10-3：浮动有外边距的图像

第7章做过一个比喻，将元素比作是有塑料边的纸片，再来看这个比喻，图像周围的塑料边（外边距）绝对不会与其他浮动元素周围的塑料边重叠。

如果确实要浮动一个非替换元素，则必须为该元素声明一个width。否则，根据CSS规范，元素的宽度趋于0。因此，假设浏览器的最小width值是1个字符，那么浮动段落可能只有1个字符宽。如果没有为浮动元素声明width值，最后可能得到如图10-4所示的结果（坦率地说，这种情况很少见，但确实是可能的）。

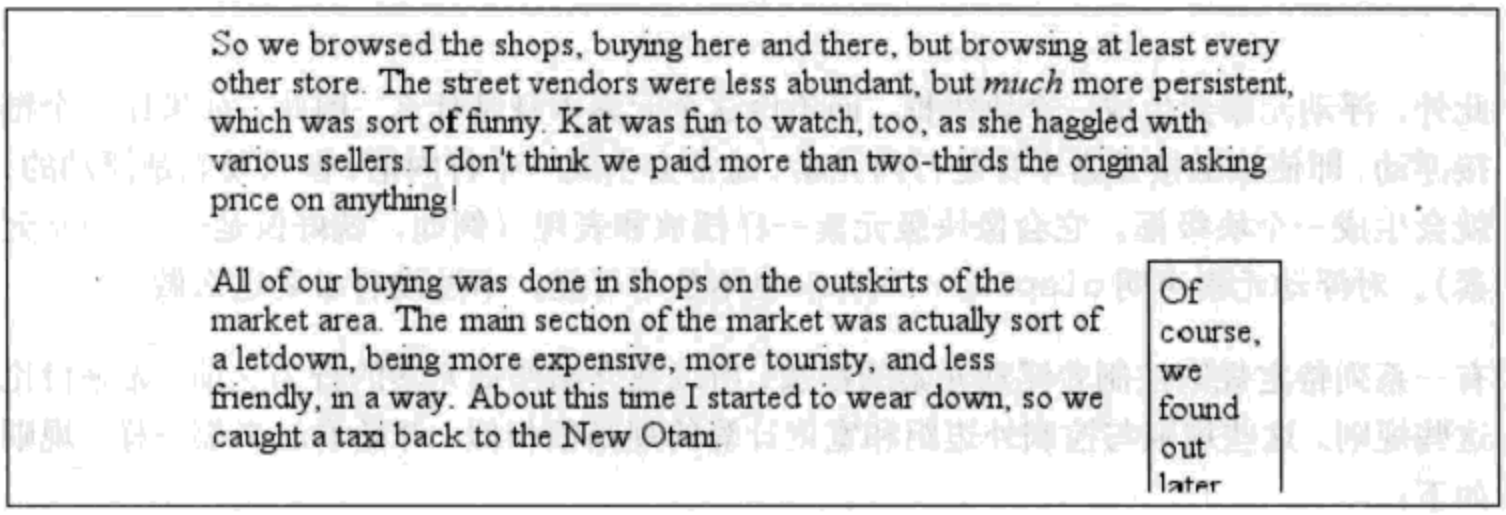

图10-4：没有显式宽度的浮动文本

不浮动

除了left和right外，float属性还有一个值。float: none用于防止元素浮动。

这看上去可能有些傻，因为要让一个元素不浮动，最容易的办法就是不声明float，这样不就行了吗？确实如此。首先，float的默认值是none。换句话说，要得到正常的非浮动行为，必须有这个值；如果没有这个none值，所有元素都会以某种方式浮动。

其次，你可能想覆盖导入样式表中的某个样式。假设你在使用一个服务器端样式表，这个样式表要让图像浮动。不过在某个特定的页面上，你不希望图像浮动。不必为此再编写一个全新的样式表，而只需在文档的嵌套样式表中增加img {float: none;}。不过，除了这种情况，确实很少有必要真正使用float: none。

浮动的详细内幕

深入讨论浮动的详细内容之前，首先要建立包含块（containning block）的概念，这很重要。浮动元素的包含块是其最近的块级祖先元素。因此，在以下标记中，浮动元素的包含块就是包含该浮动元素的段落元素：

```
<h1>Test</h1>
<p>
This is paragraph text, but you knew that.  Within the content of this
paragraph is an image that's been floated.
<img src="testy.gif" style="float: right;">
The containing block for the floated image is the paragraph.
</p>
```

注意：本章后面讨论定位时还会讨论包含块的概念。

此外，浮动元素会生成一个块级框，而不论这个元素本身是什么。因此，如果让一个链接浮动，即使该链接元素本身是行内元素，通常会生成一个行内框，但只要它是浮动的，就会生成一个块级框。它会像块级元素一样摆放和表现（例如，就好像是一个 div 元素）。对浮动元素声明 display: block 也不是不可能，不过没有必要这么做。

有一系列特定规则控制着浮动元素的摆放，所以在介绍浮动元素的行为之前，先来讨论这些规则。这些规则与控制外边距和宽度计算的规则很相似，开始看起来都一样。规则如下：

1. 浮动元素的左（或右）外边界不能超出其包含块的左（或右）内边界。

   这一点很显然。左浮动元素的左外边界向左最远只能到其包含块的左内边界；类似地，右浮动元素向右最远只能到达其包含块的右内边界，如图 10-5 所示（在这个图和后续的图中，带圆圈的数字显示了标记元素在源文档中的位置，有数字的框显示了可见浮动元素的位置和大小）。

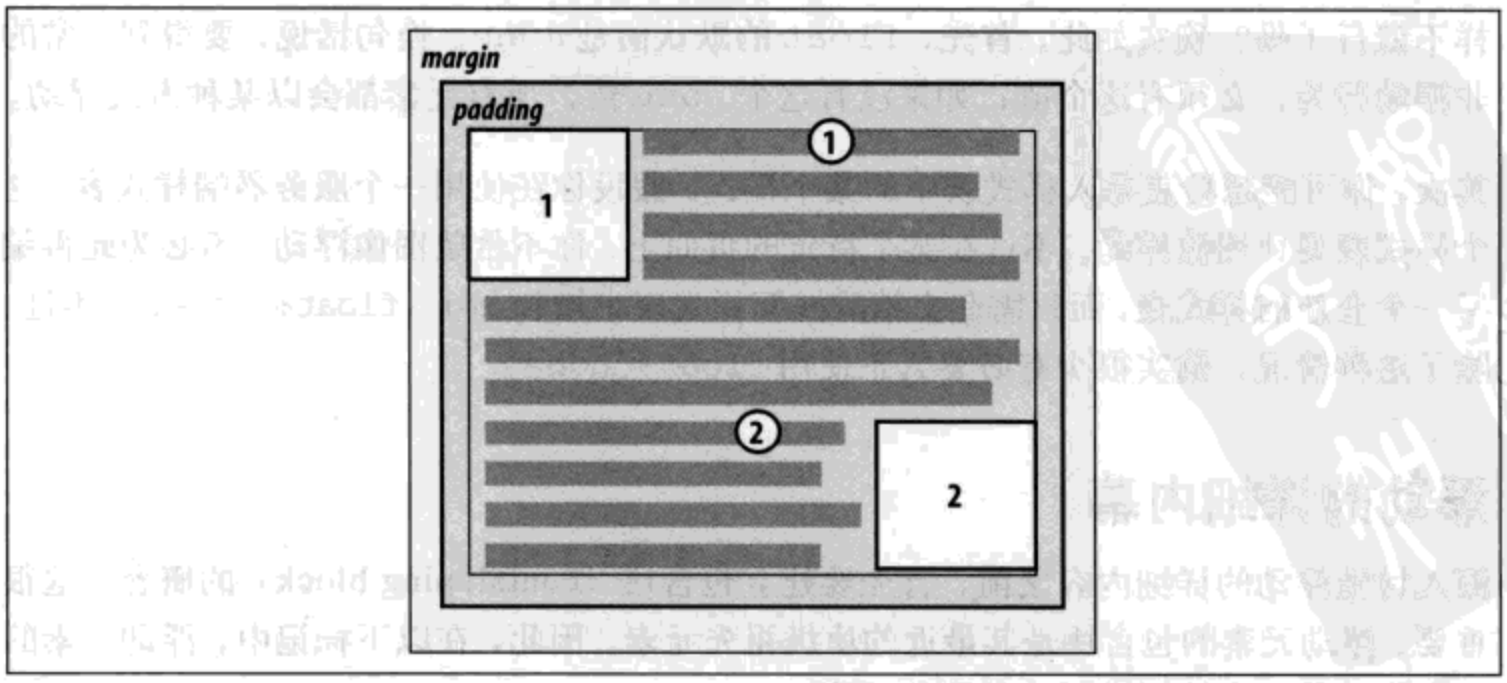

图 10-5：向左（或向右）浮动

2. 浮动元素的左（或右）外边界必须是源文档中之前出现的左浮动（或右浮动）元素的右（左）外边界，除非后出现浮动元素的顶端在先出现浮动元素的底端下面。

   这条规则可以防止浮动元素彼此“覆盖”。如果一个元素向左浮动，而另一个元素已经在那个位置，后放置的元素将挨着前一个浮动元素的右外边界放置。不过，如果一个浮动元素的顶端在所有之前浮动图像（元素）的底端下面，它可以一直浮动到父元素的左内边界。图 10-6 显示了这种行为的一些例子。

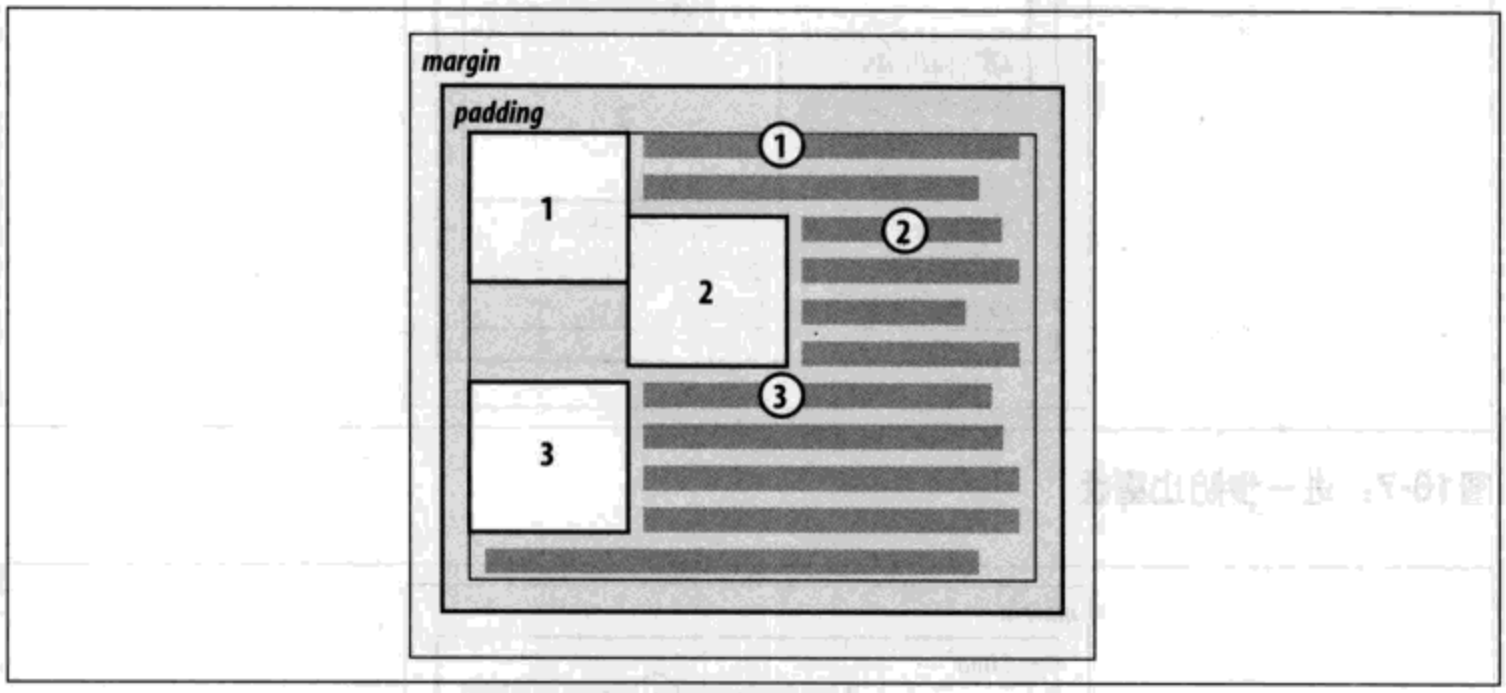

图10-6：避免浮动产生重叠

   这个规则的好处是，所有浮动内容都是可见的，因为你不必担心一个浮动元素与另一个浮动元素重叠。这使得浮动很安全。使用定位时情况则完全不同，定位很容易导致元素相互覆盖。

3. 左浮动元素的右外边界不会在其右边右浮动元素的左外边界的右边。一个右浮动元素的左外边界不会在其左边任何左浮动元素的右外边界的左边。

   这条规则可以防止浮动元素相互重叠。假设有一个 `body`，宽为 500 像素，它只有两个 300 像素宽的图像。第一个图像浮动到左边，第二个浮动到右边。这个规则可以防止第二个图像与第一个图像重叠 100 像素。实际上，它会要求第二个图像向下浮动，直到其顶端在左浮动图像的底端之下，如图 10-7 所示。

4. 一个浮动元素的顶端不能比其父元素的内顶端更高。如果一个浮动元素在两个合并外边距之间，放置这个浮动元素时就好像在两个元素之间有一个块级父元素。

   这个规则的前半部分很简单，可以防止浮动元素一直浮动到文档的顶端。正确的行为见图 10-8 所示。这个规则的第二部分则是对某些情况下的对齐进行微调 —— 例如，如果有三个段落，其中中间的段落浮动。在这种情况下，浮动段落就会像有一

个块级父元素一样（如div）浮动。这能防止浮动段落一直向上移动到三个段落共同的父元素的顶端。

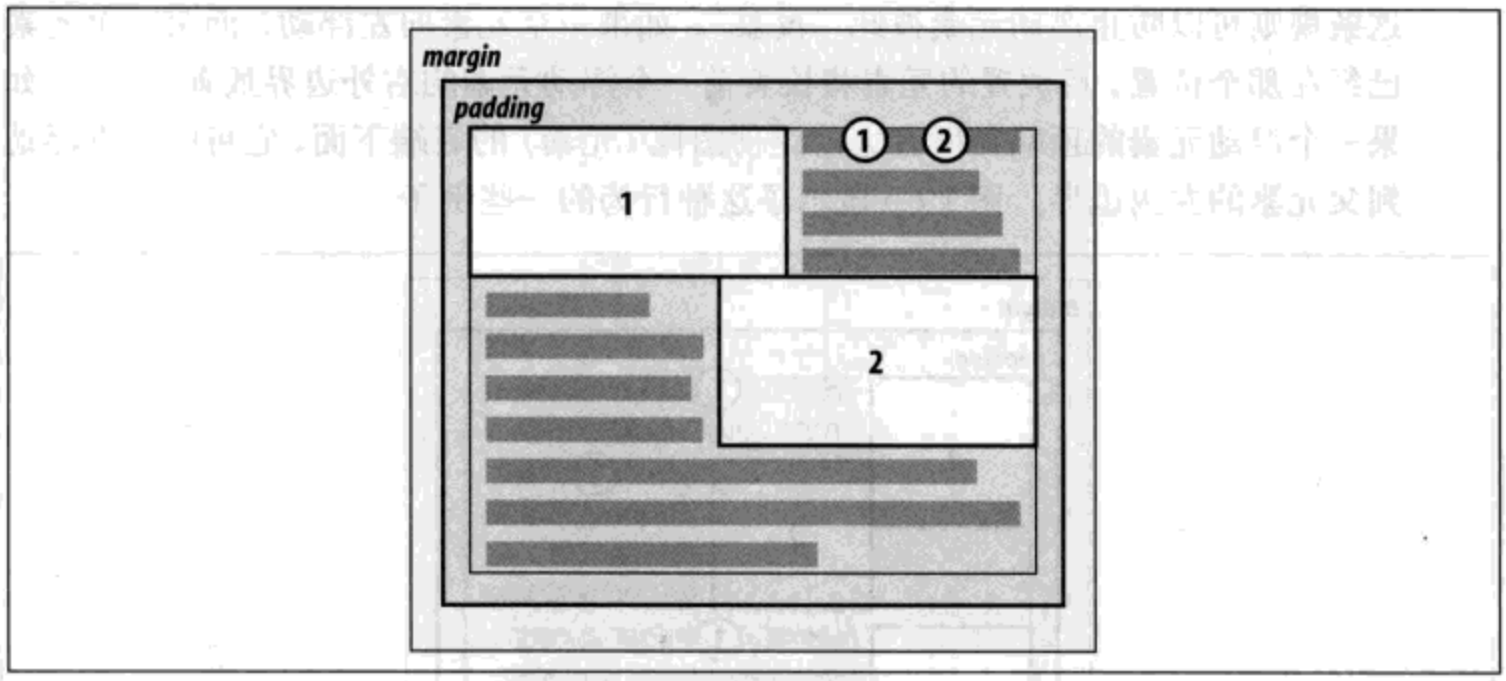

图10-7：进一步防止重叠

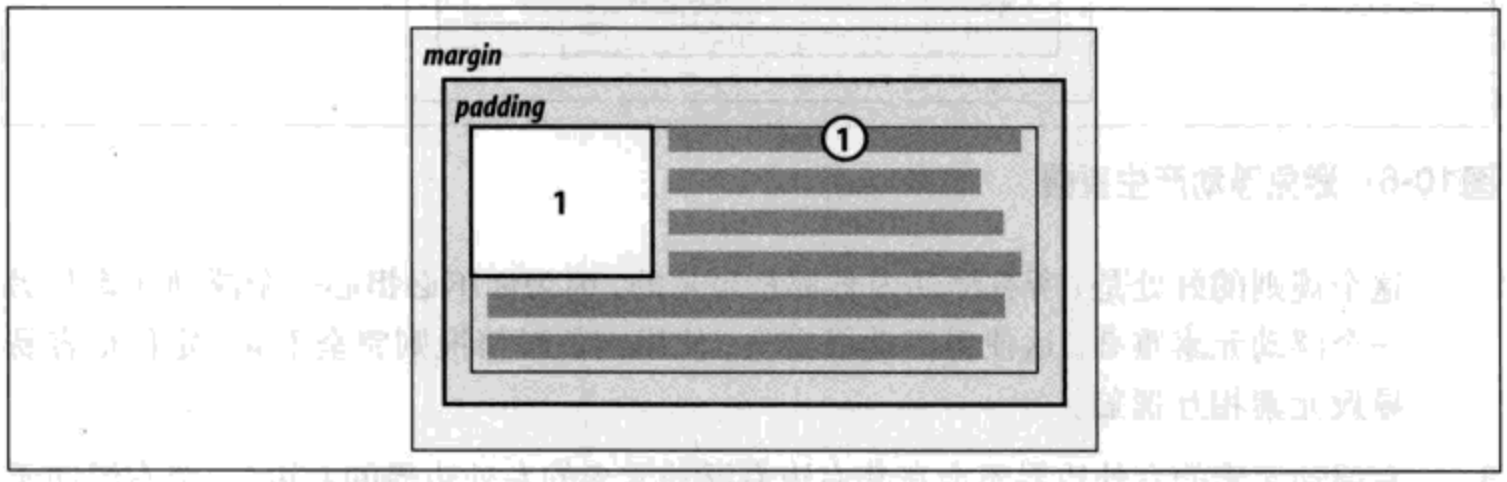

图10-8：不同于气球，浮动元素不能向上一直浮动

5. 浮动元素的顶端不能比之前所有浮动元素或块级元素的顶端更高。

   类似于规则4，规则5要求浮动元素不能一直浮动到其父元素的顶端。另外，浮动元素的顶端也不可能比之前出现的浮动元素的顶端更高。图10-9所示就是这样一个例子；在这里可以看到，由于第二个浮动元素必须在第一个浮动元素的下面，第三个浮动元素的顶端则是第二个浮动元素的顶端，而不是第一个浮动元素的顶端。

6. 如果源文档中一个浮动元素之前出现另一个元素，浮动元素的顶端不能比包含该元素所生成框的任何行框的顶端更高。

   类似于规则4和规则5，这个规则进一步限制了元素的向上浮动，不允许元素浮动到包含该浮动元素之前内容的行的顶端之上。假设一个段落的正中间有一个浮动图

像。这个图像顶端最高只能放在该图像所在行框的顶端。从图 10-10 中可以看到，这样图像就不会向上浮动太远。

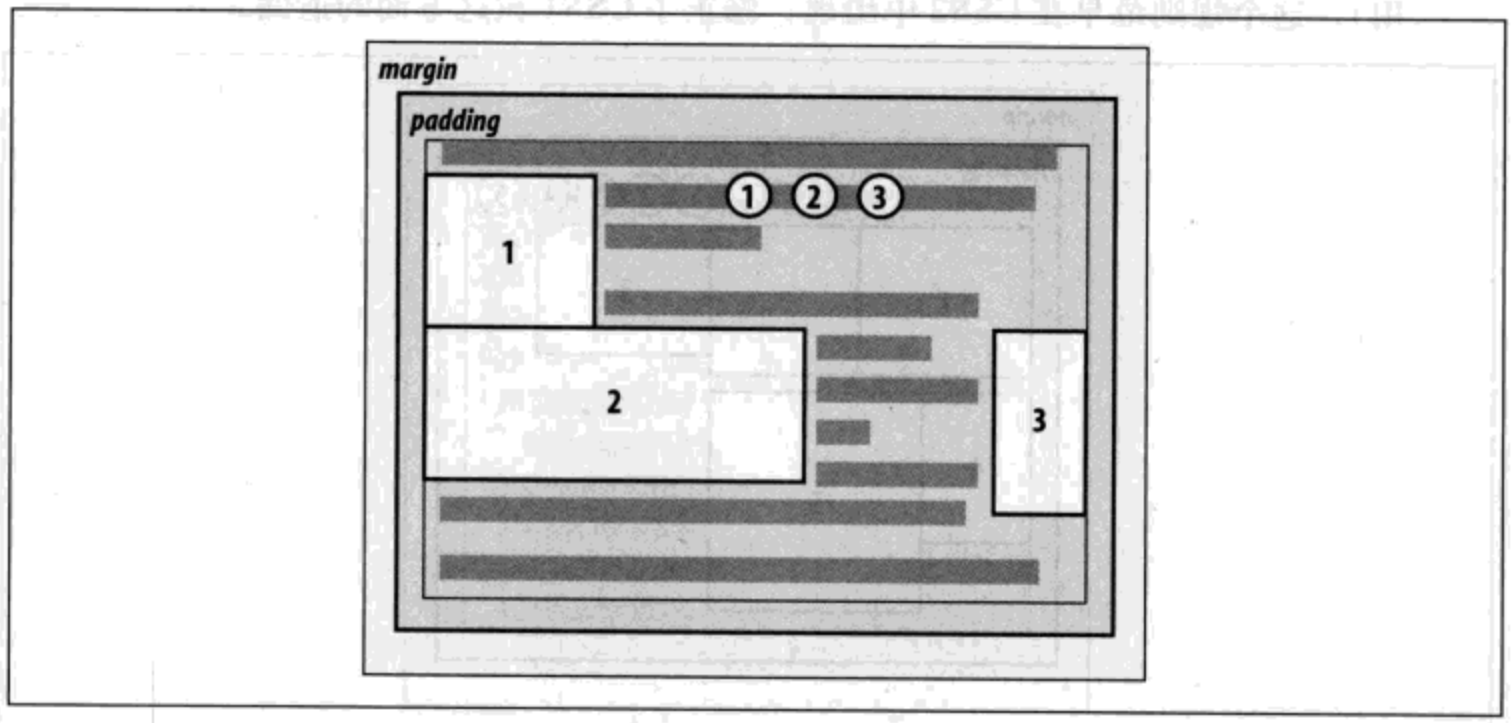

图10-9：使浮动元素总在其之前浮动元素的下面

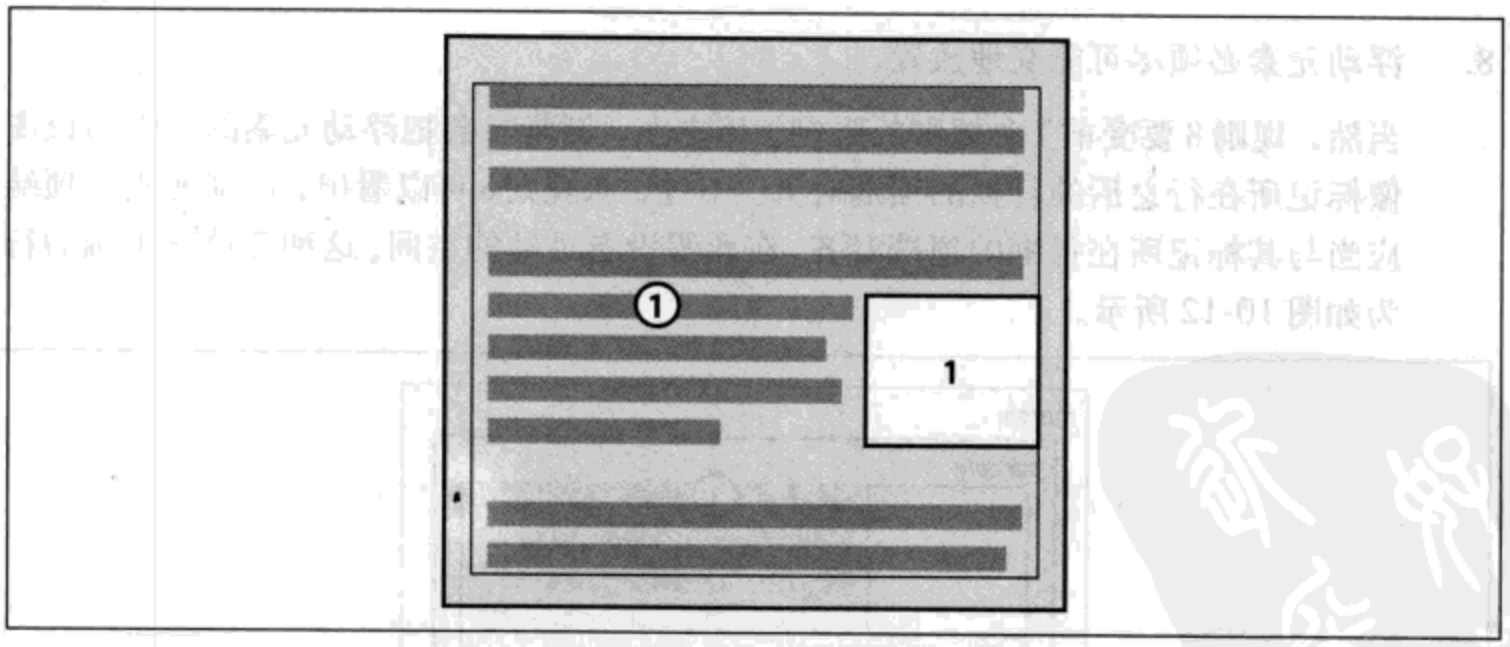

图10-10：使浮动元素在其上下文内

7. 左（或右）浮动元素的左边（右边）有另一个浮动元素，前者的右外边界不能在其包含块的右（左）边界的右边（左边）。

   也就是说，浮动元素不能超出其包含元素的边界，除非它太宽，本身都无法放下。这就能防止出现以下情况，即多个浮动元素出现在一个水平行上，远远超出其包含块的边界。相反，如果浮动元素出现在另一个浮动元素的旁边，而因此可能导致超

出包含块的话，实际上这个浮动元素会向下浮动到之前浮动元素下面的某一点，如图 10-11 所示（图中浮动元素从下一行开始，由此更清楚地说明这个规则在起作用）。这个规则最早在 CSS2 中出现，修正了 CSS1 在这方面的遗漏。

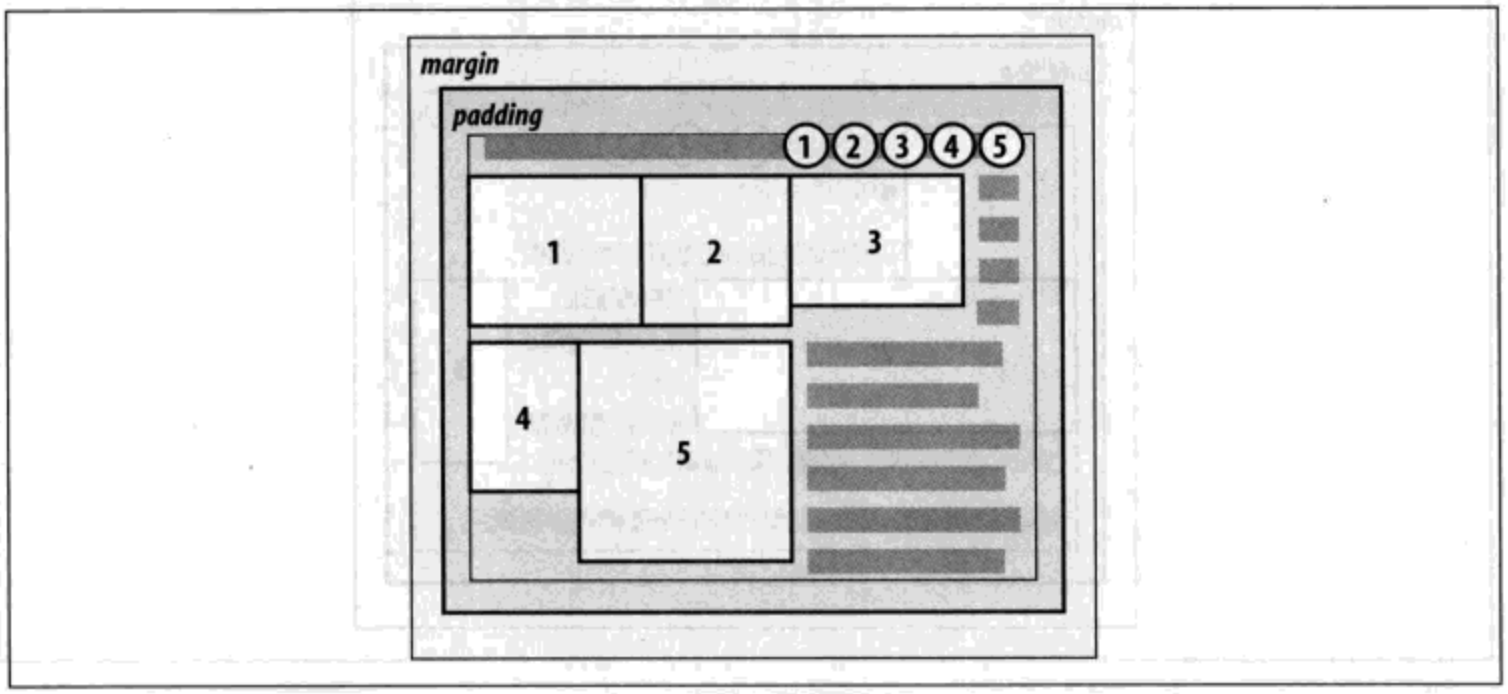

图 10-11：如果没有足够的空间，浮动元素会被挤到一个新的“行”上

8. 浮动元素必须尽可能高地放置。

当然，规则 8 要受前 7 条规则的限制。历史上，浏览器会把浮动元素的顶端与该图像标记所在行之后的行框的顶端对齐。不过，从规则 8 可以看出，浮动元素的顶端应当与其标记所在行框的顶端对齐，在此假设有足够的空间。这种理论上正确的行为如图 10-12 所示。

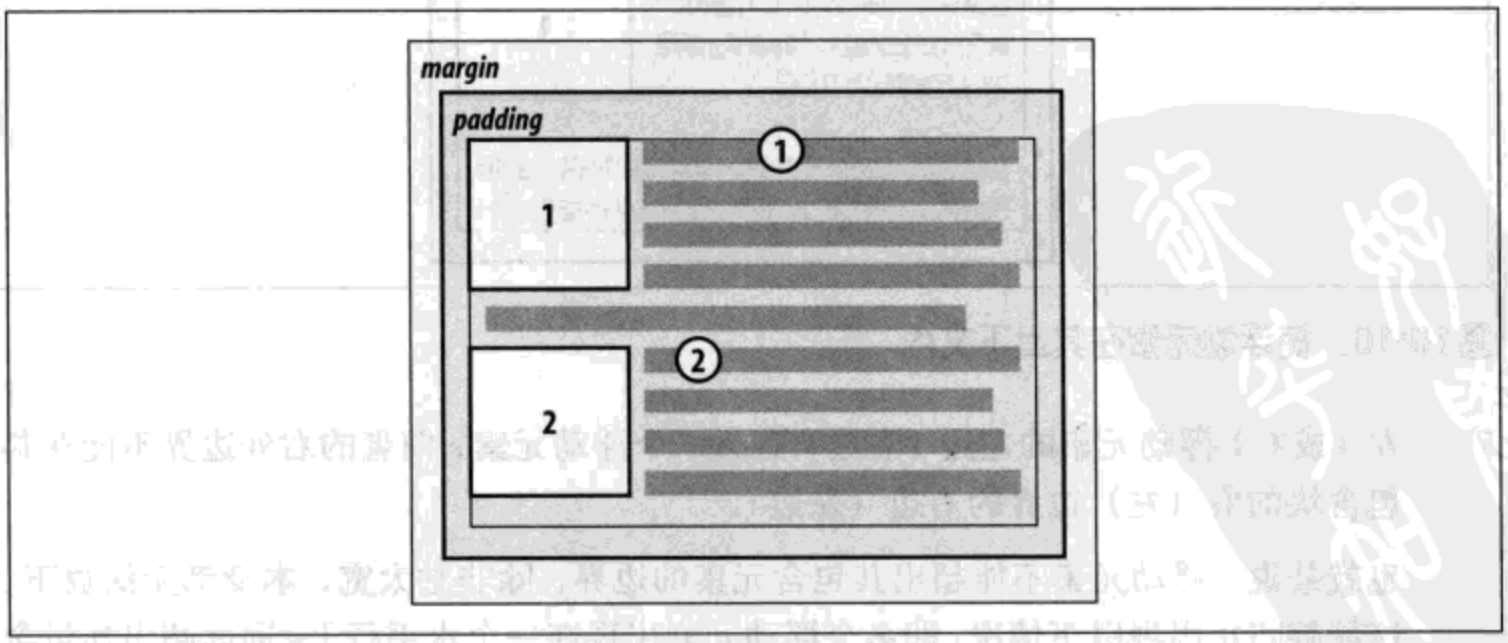

图 10-12：满足其他约束条件的前提下，浮动得尽可能高

警告： 遗憾的是，由于对于“尽可能高”没有明确的定义（这可能表示“按常规尽可能高”，实际上对这种含义存在争议），所以不能指望被认为CSS1兼容的所有浏览器都有一致的行为。有些浏览器会遵循历史做法，将图像浮动到下一行，而另外一些浏览器会在空间足够的情况下将图像浮动到当前行。

9. 左浮动元素必须向左尽可能远，右浮动元素则必须向右尽可能远。位置越高，就会向右或向左浮动得越远。

   同样的，这条规则要受前几条规则的限制。这里同样存在规则8中的警告，不过不那么模糊。从图10-13可以看到，很容易确定元素已经向右或向左走得尽可能远。

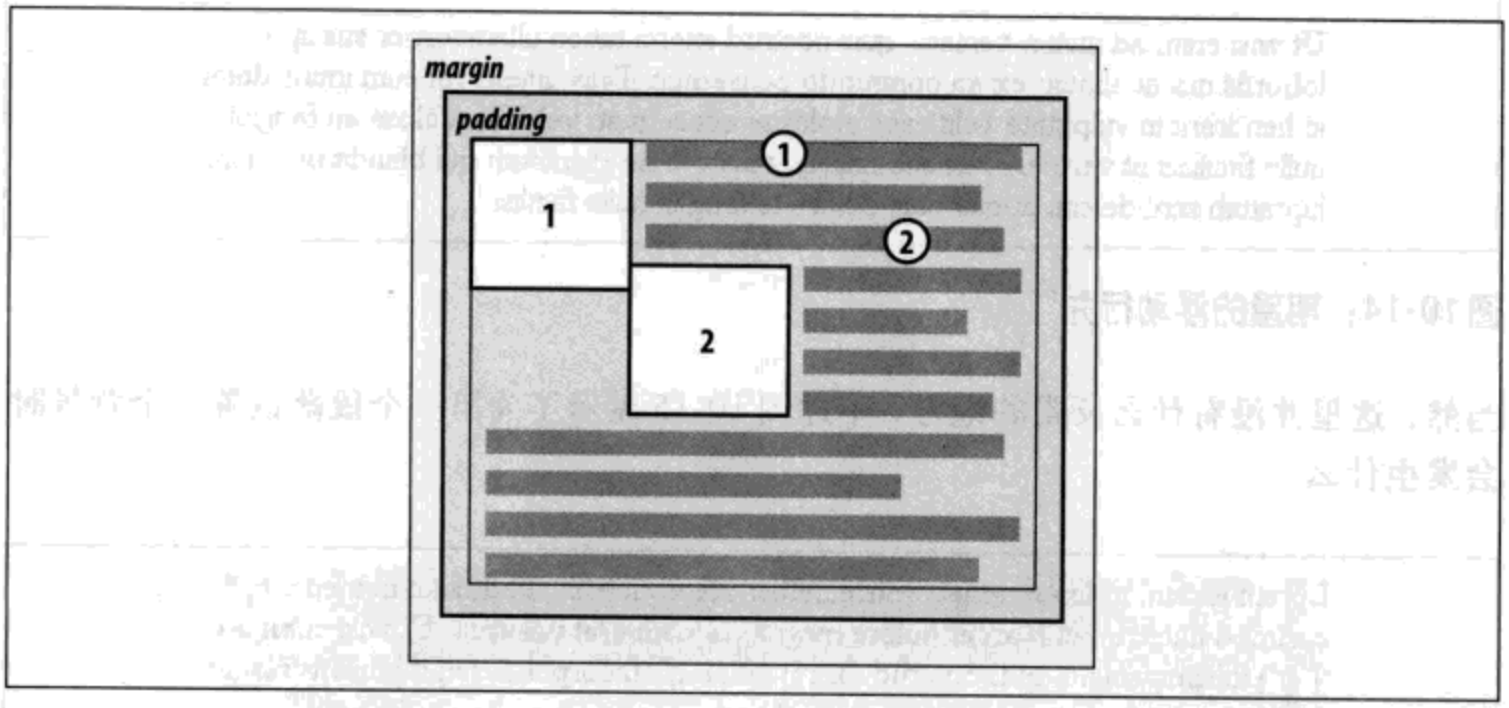

图10-13：向右（或向左）尽可能远

实用行为

前面介绍的规则有一些有意思的后果，这些结果源于两方面，一方面是规则中指出了一些要求，另一方面是规则中有些方面没有谈到。首先要讨论浮动元素比其父元素高时会有什么结果。

实际上，这种情况经常发生。以一个小文档为例，这个文档只包含几个段落和h3元素，其中第一个段落包含一个浮动图像。另外，这个浮动图像有5像素的外边距（5px）。你可能认为这个文档应该显示如下，如图10-14所示。

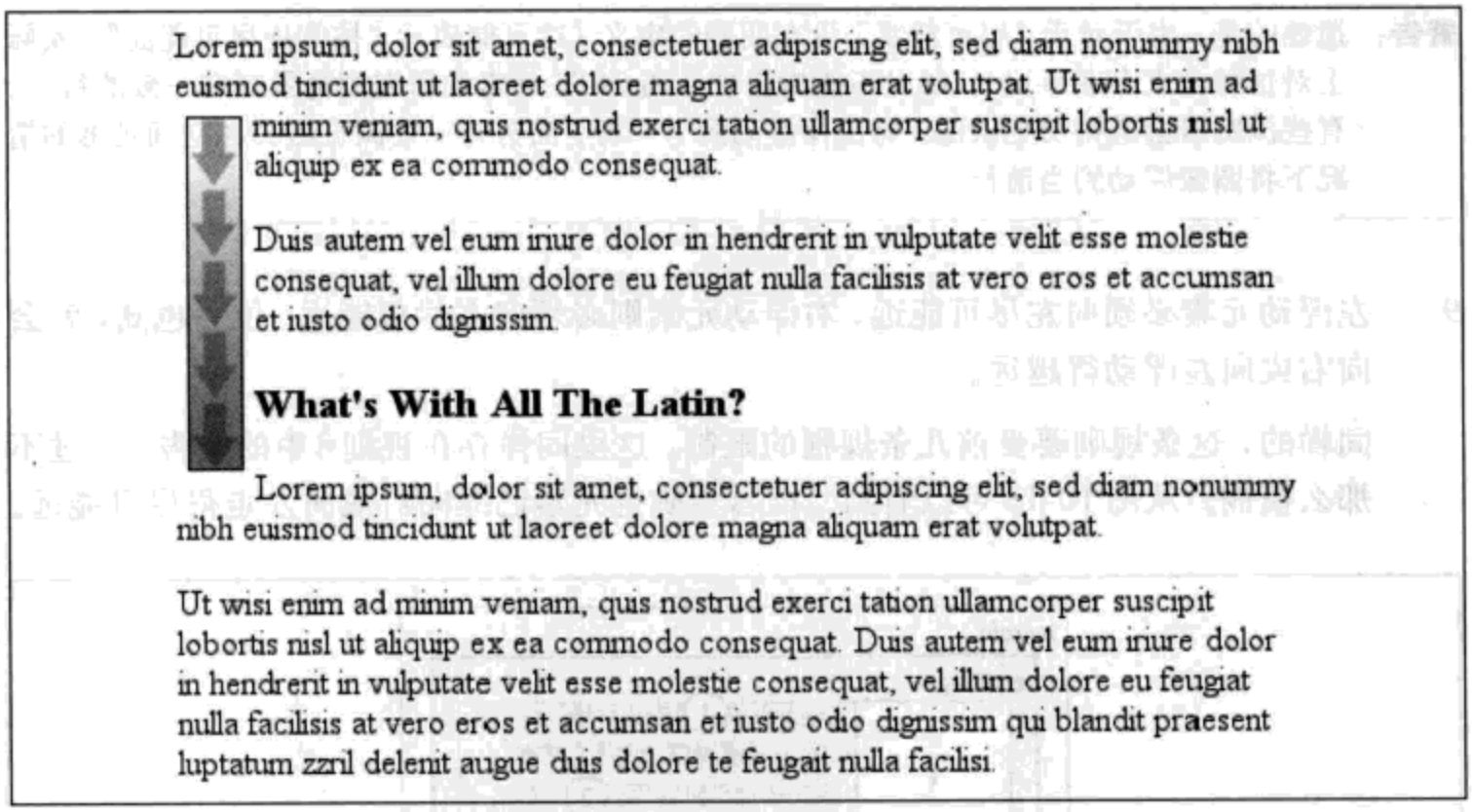

图10-14：期望的浮动行为

当然，这里并没有什么反常的地方，不过图10-15显示了为第一个段落设置一个背景时会发生什么。

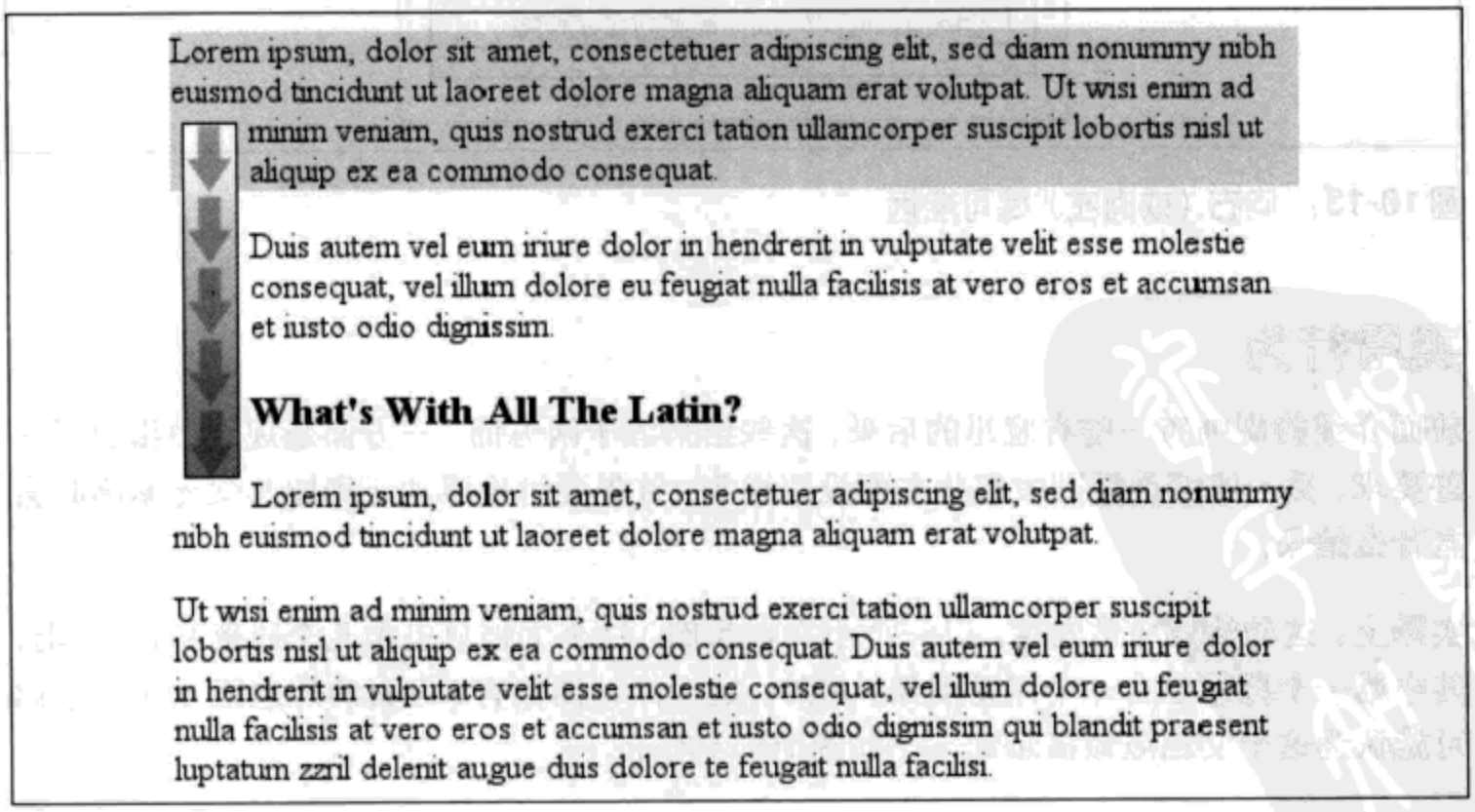

图10-15：背景和浮动元素

这与第二个例子基本上没有什么不同，只不过背景可见。可以看到，浮动图像超出了其父元素的底端。当然，在第一个例子中也是如此，只不过第一个例子中不明显，因为无法看到背景。前面讨论的浮动规则只处理了浮动元素和其父元素的左、右和上边界，而没有涉及下边界，这个遗漏是故意的，这就会得到如图 10-15 中所示的行为。

警告： 实际上，有些浏览器不能正确地做到这一点。相反，它们会增加父元素的高度，使浮动元素能够包含在父元素中，即使这会导致父元素中出现大量多余的空白。

CSS2.1澄清了浮动元素行为的一个方面：浮动元素会延伸，从而包含其所有后代浮动元素（而CSS的先前版本没有明确指出会发生什么情况）。所以，通过将父元素置为浮动元素，就可以把浮动元素包含在其父元素内，如下例所示：

```
<div style="float: left; width: 100%;">
  <img src="hay.gif" style="float: left;">
  The 'div' will stretch around the floated image
  because the 'div' has been floated.
</div>
```

与此相关，考虑背景及其与文档中之前出现的浮动元素的关系，见图 10-16 所示。

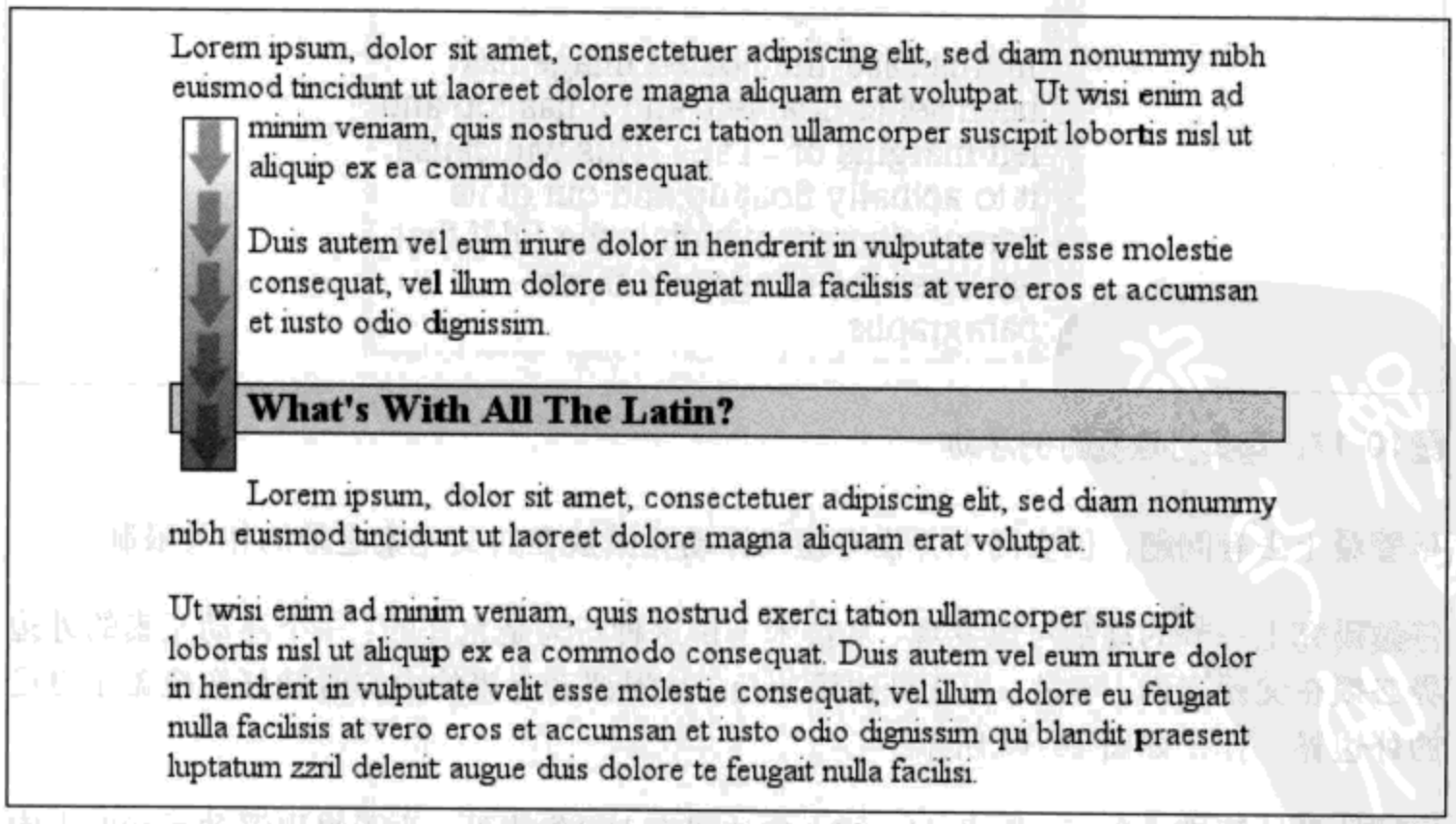

图 10-16：元素背景“滑到”浮动元素下面

由于浮动元素同时处于流内和流外，所以这种情况肯定会发生。然后会怎样呢？标题的内容由浮动元素“显示”。不过，标题的元素宽度还是与父元素宽度相等。因此，其内

容区将跨父元素的宽度，相应地，背景宽度也与父元素宽度相等。为了避免盖在浮动元素下面，具体内容并不从其内容区左边界开始显示。

负外边距

有意思的是，负外边距可能导致浮动元素移到其父元素的外面。这看上去与先前的规则相矛盾，不过其实并不矛盾。通过设置负外边距，元素可能看上去比其父元素更宽，同样的道理，浮动元素也可能超出其父元素。

下面考虑一个向左浮动的浮动图像，其左外边距和上外边距都是-15px。这个图像放在一个div中，该div没有内边距、边框和外边距。结果如图10-17所示。

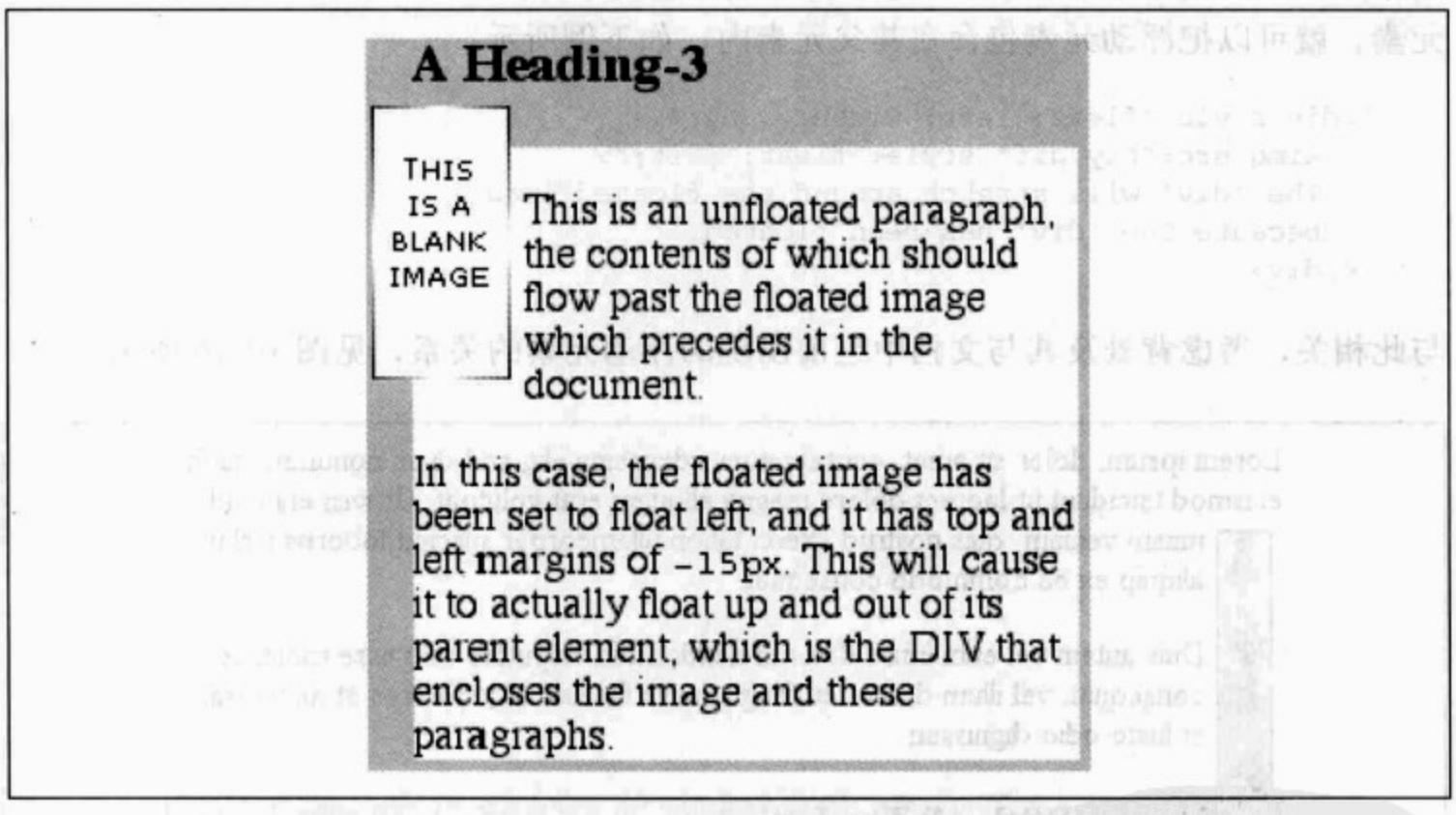

图10-17：有负外边距时的浮动

尽管看上去有问题，但实际上并没有违反浮动元素放在其父元素之外的相关限制。

仔细研究上一节的规则可以发现，从技术上讲这种行为是允许的：一个浮动元素的外边界必须在父元素内。不过，由于外边距为负，放置浮动元素的内容时就好像覆盖了自己的外边界一样，如图10-18所示。

通过数学计算描述如下：假设div的上内边界在100像素处。为了得出浮动元素的上内边界应该在哪里，浏览器会做以下计算：100px + (-15px)外边距 +0内边距 =85px。因此，浮动元素的上内边界应当在85像素处；尽管比浮动元素父元素的上内边界还要

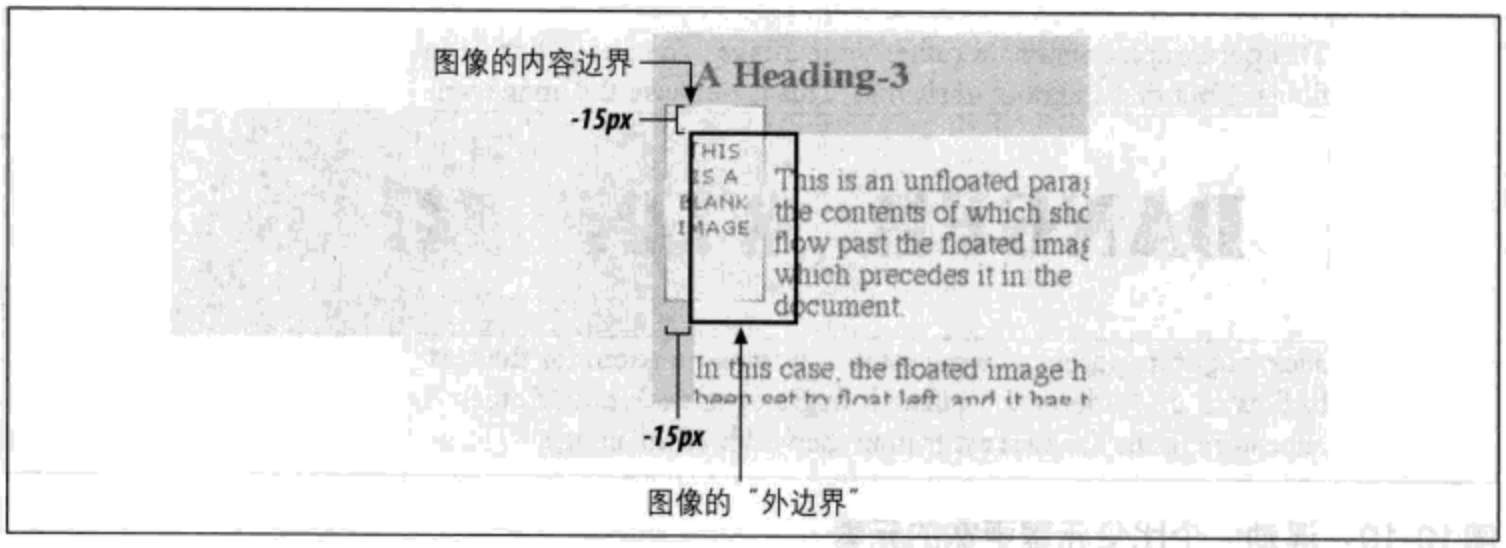

图10-18：利用负外边距向上和向左浮动详解

高，但从数学计算可知，这并没有违反规范。出于类似的原因，同样可以解释为什么浮动元素的左内边界可以放在其父元素左内边界的左边。

现在可能很多人都想大叫“犯规！”。从我个人来说，不会为此批评你们。例如，上内边界比上外边界还要高，这看上去是完全错误的；不过，如果有一个负的上外边距，确实会产生这种结果，这与负外边距使正常的非浮动元素视觉上比其父元素还要宽是一样的。浮动元素框的4个边都是如此：如果将外边距设置为负值，内容就会超出外边界，但从技术上讲并没有违反规范。

这里有一个重要问题：在使用负外边距时，如果浮动元素元素超出其父元素，文档会如何显示？例如，一个图像可能浮动得太远，超出了用户代理已显示的一个段落。在这种情况下，要由用户代理决定文档是否重新显示。CSS1和CSS2规范明确地指出，用户代理不必重新显示已显示内容来适应文档中后来出现的内容。换句话说，如果一个图像浮动到之前已经显示的段落中，它可能只是覆盖该位置上原有的内容。另一方面，用户代理也可能采用一种不同的方法处理这种情况，让内容环绕浮动元素重新显示。不论采用哪种方式，都不要指望肯定会发生某一种行为，否则为浮动元素设置负外边距的作用会受到限制。让元素浮动可能很安全，不过试图将元素在页面上向上推往往不是好主意。

还有另外一种方法可以让浮动元素超出其父元素的左右内边界：即浮动元素比其父元素更宽。在这种情况下，浮动元素会超出右或左内边界，从而尽可能正确地显示，究竟是超出右内边界还是左内边界，取决于元素以何种方式浮动。这会得到如图10-19所示的结果。

浮动元素、内容和重叠

还有一个更有意思的问题：如果一个浮动元素与正常流中的内容发生重叠会怎么样呢？

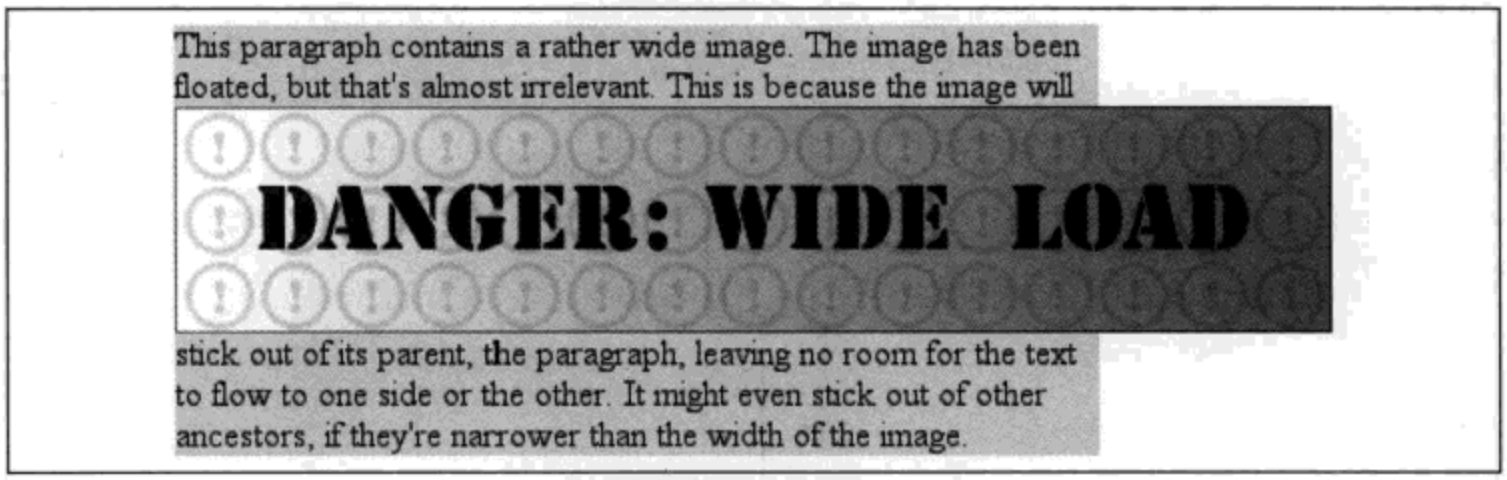

图10-19：浮动一个比父元素更宽的元素

例如，如果一个浮动元素在内容流过的边上有负外边距（例如，右浮动元素上的左外边距为负），就会发生这种情况。你已经看到了对块级元素的边框和背景有什么影响。那么行内元素呢？

CSS1和CSS2对于这种情况下会有何种行为并不完全明确。CSS2.1则澄清了这个问题，指出以下明确的规则：

- 行内框与一个浮动元素重叠时，其边框、背景和内容都在该浮动元素“之上”显示。
- 块框与一个浮动元素重叠时，其边框和背景在该浮动元素“之下”显示，而内容在浮动元素“之上”显示。

为了说明这些规则，考虑以下情况：

```
<img src="testy.gif" class="sideline">
<p class="box">
This paragraph, unremarkable in most ways, does contain an inline element.
This inline contains some <strong>strongly emphasized text, which is
so marked to make an important point</strong>.  The rest of the element's
content is normal anonymous inline content.
</p>
<p>
This is a second paragraph.  There's nothing remarkable about it, really.
Please move along.
</p>
<h2 id="jump-up">A Heading!</h2>
```

对该标记应用以下样式，其结果如图10-20所示：

```
img.sideline {float: left; margin: 10px -15px 10px 10px;}
p.box {border: 1px solid gray; padding: 0.5em;}
p.box strong {border: 3px double black; background: silver; padding: 2px;}
h2#jump-up {margin-top: -15px; background: silver;}
```

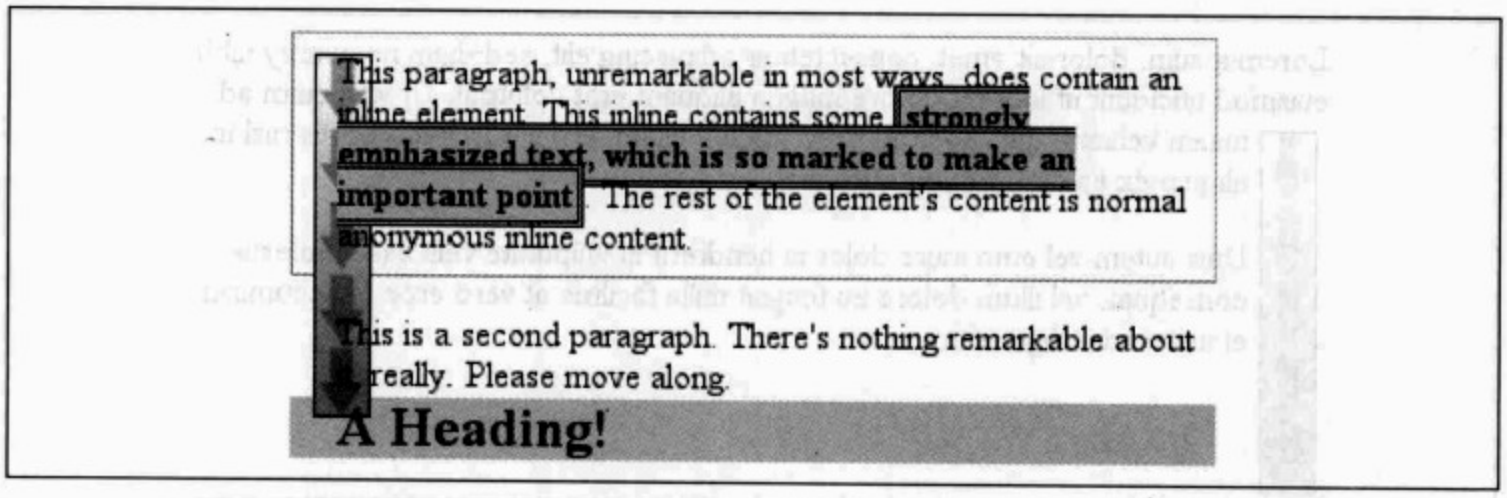

图10-20：与浮动元素重叠时的布局行为

行内元素（strong）完全覆盖了浮动图像（包括背景、边框、内容和一切）。另一方面，块元素只是将其内容显示在浮动元素之上，但其背景和边框都放在浮动元素之下。

注意： 这里所述的重叠行为与源文档中的顺序无关。元素在浮动元素之前还是之后出现并不重要；不论有怎样的顺序，都会有同样的行为。

清除

我们已经讨论了一些浮动行为，介绍定位前还有一个内容没有谈到。你可能并不总是希望内容流过浮动元素，某些情况下，可能要特意避免这种行为。如果你的文档分组为小节，可能不希望一个小节的浮动元素浮动到另一个小节中。在这种情况下，你希望将每小节的第一个元素设置为禁止浮动元素出现在它旁边。如果第一个元素可能放在一个浮动元素旁边，则会向下推，直到出现在浮动元素的下面，而且所有后续内容都在其后面出现，如图 10-21 所示。

这可以利用 clear 属性完成。

例如，为了确保所有h3元素不会放在左浮动元素的右边，可以声明h3{clear: left;}。这可以解释为“确保一个 h3 的左边没有浮动图像”，其效果非常类似于 HTML 中的 <br clear="left">（有讽刺意味的是，大多数浏览器的默认行为都是为 br 元素生成行内框，所以clear不能应用于br，除非改变其display值!）。以下规则使用clear来防止 h3 元素左边有浮动元素：

```
h3 {clear: left;}
```

这会把 h3 推过所有左浮动元素，不过还允许浮动元素出现在 h3 元素的右边，如图 10-22 所示。

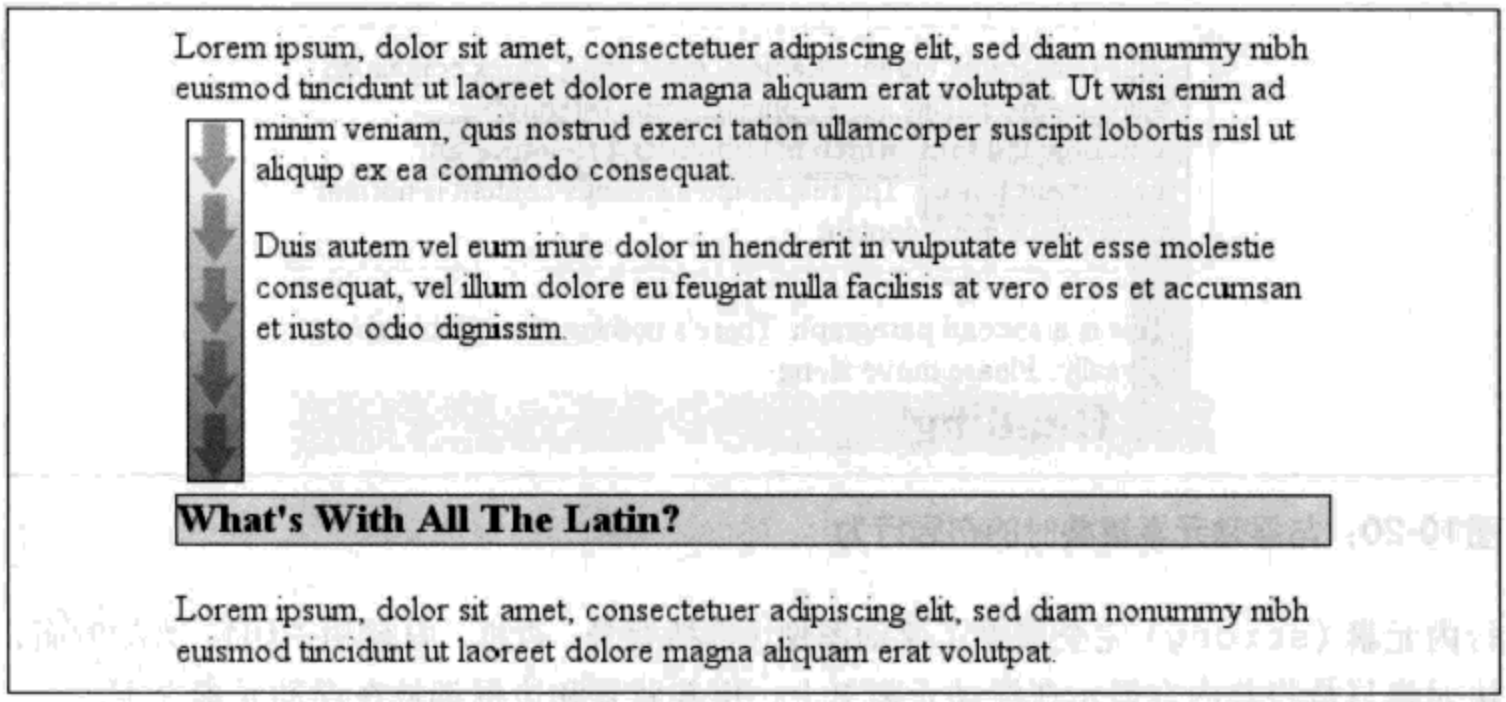

图10-21：显示一个有clear属性的元素

clear	
值：	left | right | both | none | inherit
初始值：	none
应用于：	块级元素
继承性：	无
计算值：	根据指定确定

为了避免这种情况，确保h3元素不会与任何浮动元素在同一行上，要使用值both：

```
h3 {clear: both;}
```

这很好理解，这个值会防止指定了clear的元素两边存在浮动元素，如图10-23所示。

另一方面，如果只是不希望h3元素的右边有浮动元素，要使用h3{clear: right;}。

最后还有一个clear: none，它允许元素浮动到另一个元素的任意两边。float: none值之所以存在，主要是为了支持正常的文档行为，即元素允许其两边有浮动元素。当然，可以用none来覆盖其他样式，如图10-24所示。尽管有规则指出h3元素不允许两边有浮动元素，不过，有一个h3特别设置为允许两边有浮动元素：

```
h3 {clear: both;}

<h3 style="clear: none;">What's With All The Latin?</h3>
```

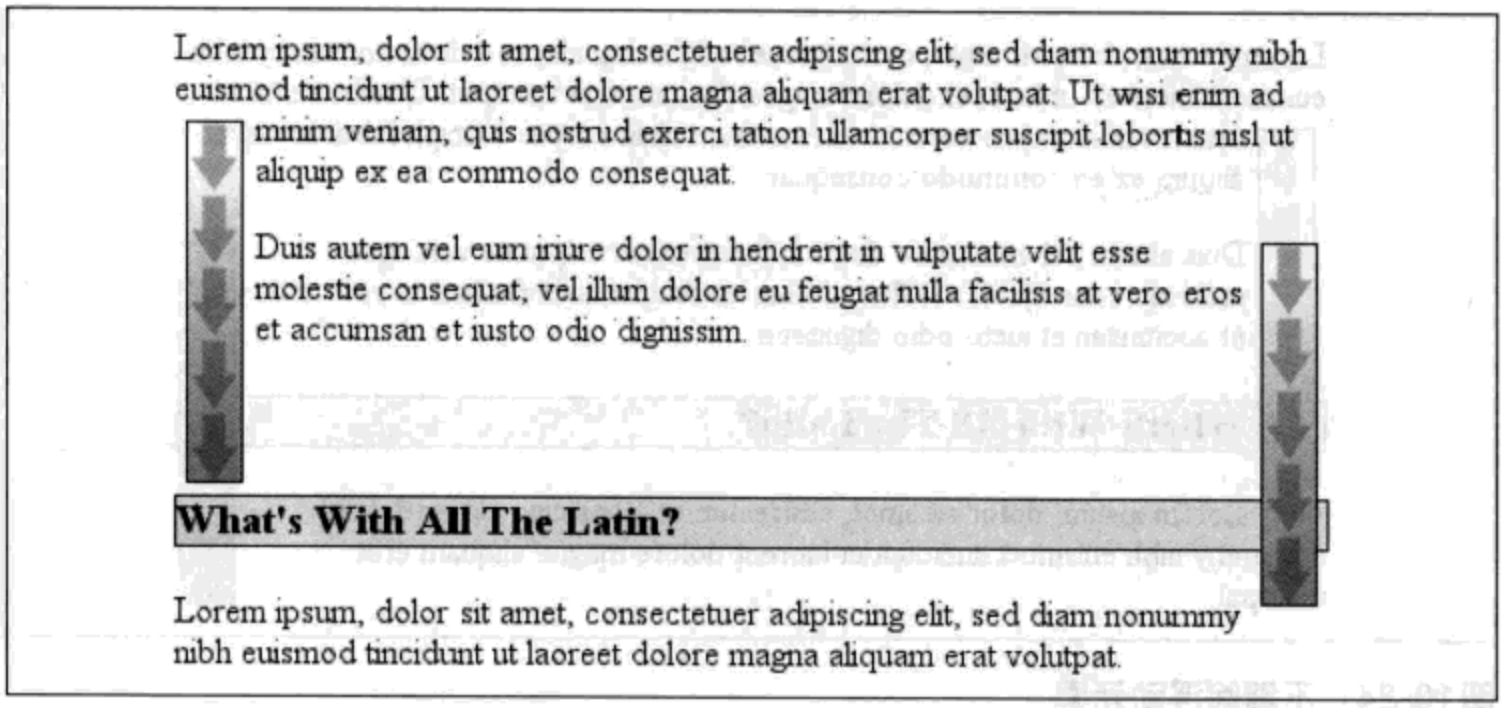

图10-22：清除左边，但不清除右边的浮动元素

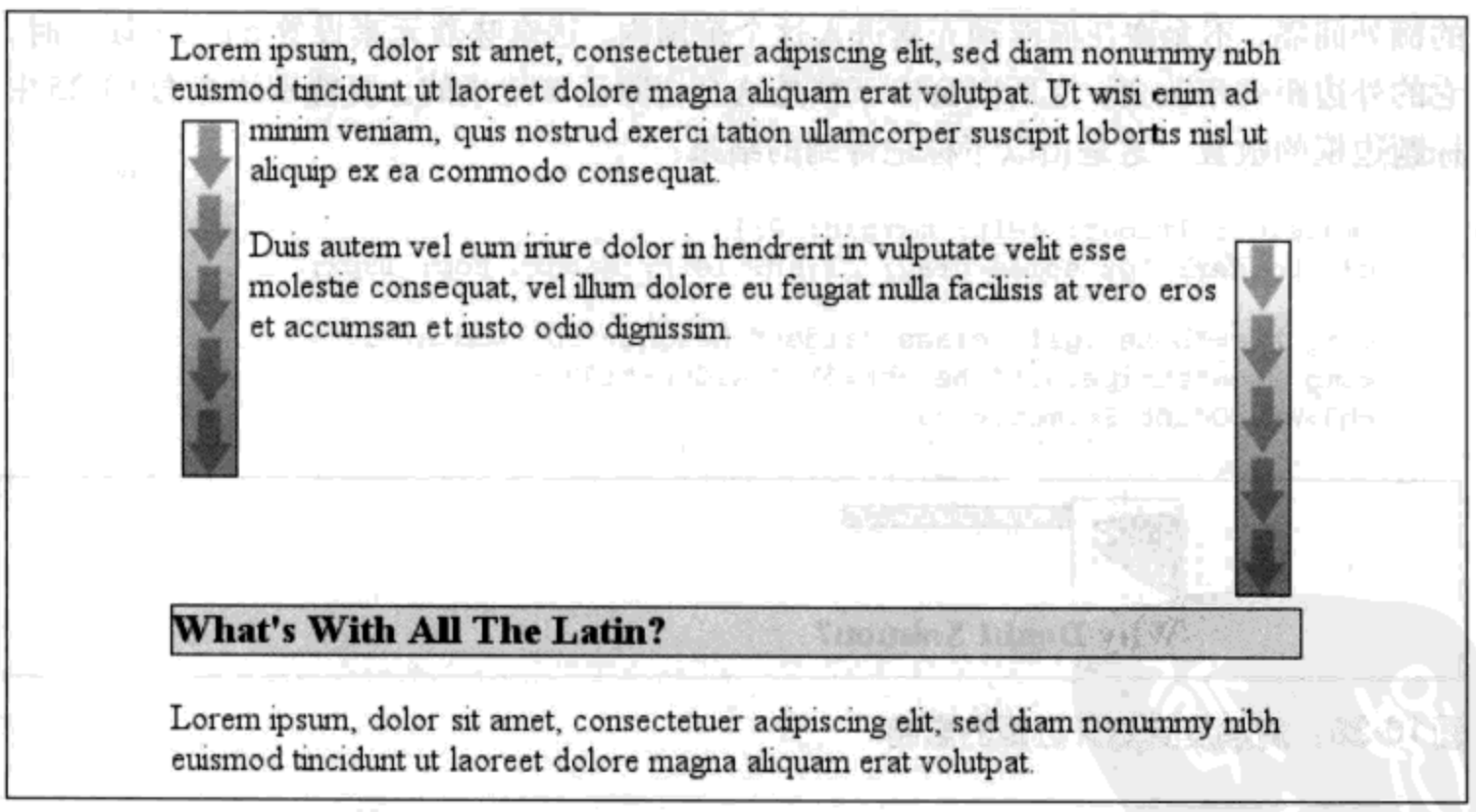

图10-23：两边都清除

在 CSS1 和 CSS2 中，`clear` 工作如下，它会增加元素的上外边距，使之最后落在浮动元素的下面，这实际上会忽略为清除元素（设置了clear的元素）顶端设置的外边距宽度。也就是说，清除元素的上外边距可能会调整，例如，并不是`1.5em`，而可能增加到`10em`或`25px`，甚至`7.133in`，或者是将元素下移足够远（使内容区在浮动元素下边界的下面）所需的任何长度。

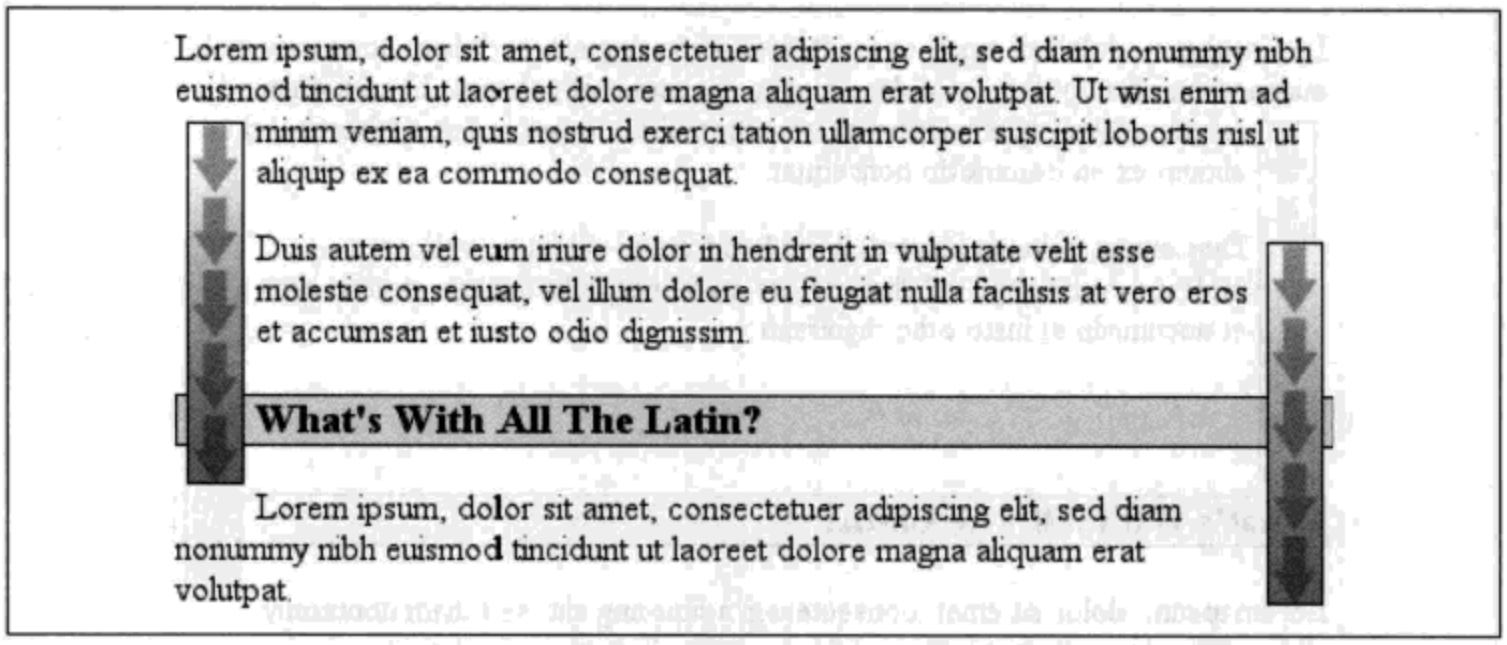

图10-24：不清除浮动元素

在CSS2.1中，引入了一个清除区域（clearance）。清除区域是在元素上外边距之上增加的额外间隔，不允许任何浮动元素进入这个范围内。这意味着元素设置clear属性时，它的外边距并不改变。之所以会向下移是这个清除区域造成的。要特别注意图10-25中标题边框的放置，这是由以下标记得到的结果：

```
img.sider {float: left; margin: 0;}
h3 {border: 1px solid gray; clear: left; margin-top: 15px;}

<img src="boxer.gif" class="sider" height="50" width="50">
<img src="stripe.gif" height="10" width="100">
<h3>Why Doubt Salmon?</h3>
```

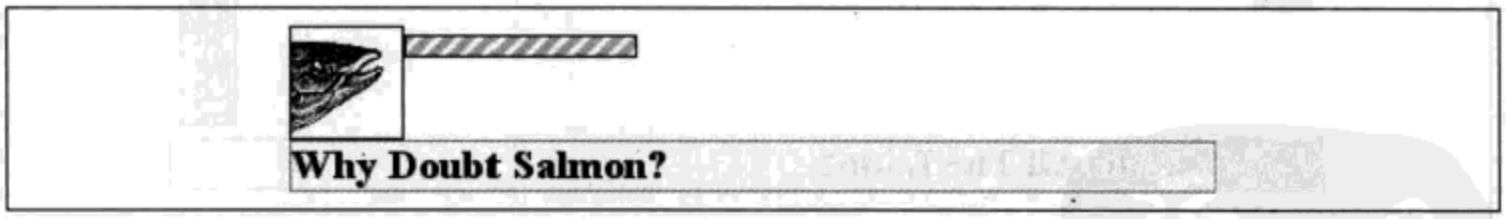

图10-25：清除及其对外边距的影响

h3上边框与浮动图像下边框之间没有间隔，因为在h3的上外边距（15像素）之上增加了25像素的清除区域，将h3的上边框边界推到刚好越过浮动元素的下边界。一般都会这样，除非h3的上外边距计算为40像素或更多，在这种情况下，h3会很自然地放在浮动元素下面，clear值是什么则无关紧要。

当然，大多数情况下无法知道一个元素周围多大范围内不允许有浮动元素。要确保一个清除元素（设置clear属性的元素）的顶端与一个浮动元素的底端之间有一定空间，可

以为浮动元素本身设置一个下外边距。所以，如果希望上例中的浮动元素下面至少有15像素的空间，可以修改如下：

```
img.sider {float: left; margin: 0 0 15px;}
h3 {border: 1px solid gray; clear: left;}
```

浮动元素的下外边距会增加浮动框的大小，且由于清除元素必须推到浮动框下面的某个点以下，浮动框大小的增加会使这个点下移。前面已经看到过，这是因为浮动元素的外边距边界定义了浮动框的边界。

定位

定位的原理很简单。利用定位，可以准确地定义元素框相对于其正常位置应该出现在哪里，或者相对于父元素、另一个元素甚至浏览器窗口本身的位置。

基本概念

深入讨论各种定位之前，最好先来了解有哪些类型的定位，以及这些不同类型之间有什么区别。我们还要对一些基本思想给出定义，这些思想是理解定位如何工作的基础。

定位的类型

通过使用position属性，可以选择4种不同类型的定位，这会影响元素框生成的方式。

position	
值：	static | relative | absolute | fixed | inherit
初始值：	static
应用于：	所有元素
继承性：	无
计算值：	根据指定确定

position值的含义如下：

static

元素框正常生成。块级元素生成一个矩形框，作为文档流的一部分，行内元素则会创建一个或多个行框，置于其父元素中。

`relative`

元素框偏移某个距离。元素仍保持其未定位前的形状，它原本所占的空间仍保留。

`absolute`

元素框从文档流完全删除，并相对于其包含块定位，包含块可能是文档中的另一个元素或者是初始包含块（见下一节的介绍）。元素原先在正常文档流中所占的空间会关闭，就好像该元素原来不存在一样。元素定位后生成一个块级框，而不论原来它在正常流中生成何种类型的框。

`fixed`

元素框的表现类似于将 position 设置为 `absolute`，不过其包含块是视窗本身。

现在先不要太担心具体的细节，本章后面还会逐个地介绍上述各类定位。在此之前，需要先来讨论包含块。

包含块

本章较早前讨论过浮动元素的包含块。对于浮动元素，其包含块定义为最近的块级祖先元素。对于定位，情况则没有这么简单。CSS2.1 定义了以下行为：

- “根元素”的包含块（也称为初始包含块）由用户代理建立。在 HTML 中，根元素就是 `html` 元素，不过有些浏览器会使用 `body` 作为根元素。在大多数浏览器中，初始包含块是一个视窗大小的矩形。
- 对于一个非根元素，如果其 `position` 值是 `relative` 或 `static`，包含块则由最近的块级框、表单元格或行内块祖先框的内容边界构成。
- 对于一个非根元素，如果其 `position` 值是 `absolute`，包含块设置为最近的 `position` 值不是 `static` 的祖先元素（可以是任何类型）。这个过程如下：
  - 如果这个祖先是块级元素，包含块则设置为该元素的内边距边界；换句话说，就是由边框界定的区域。
  - 如果这个祖先是行内元素，包含块则设置为该祖先元素的内容边界。在从左向右读的语言中，包含块的上边界和左边界是该祖先元素中第一个框内容区的上边界和左边界，包含块的下边界和右边界是最后一个框内容区的下边界和右边界。在从右向左读的语言中，包含块的右边界对应于第一个框的右内容边界，包含块的左边界则取自最后一个框的左内容边界。上下边界也是一样。
  - 如果没有祖先，元素的包含块定义为初始包含块。

这里有一点很重要：元素可以定位到其包含块的外面。这与浮动元素使用负外边距浮动到其父元素内容区外面很类似。这也说明，“包含块”一词实际上应该是“定位上下文”，

不过，由于规范使用的是“包含块”，所以我也沿用了这个说法（我已经在尽力减少由此带来的误解，真的！）。

偏移属性

上一节介绍的三种定位机制（relative、absolute和fixed）使用了4个属性来描述定位元素各边相对于其包含块的偏移。我们将这4个属性称为偏移属性（offset properties），这对于完成定位是很重要的一部分。

top、right、bottom、left

值：	<length> | <percentage> | auto | inherit
初始值：	auto
应用于：	定位元素（也就是position值不是static的元素）
继承性：	无
百分数：	对于top和bottom，相对于包含块的高度；对于right和left，相对于包含块的宽度
计算值：	对于相对定位元素，见以下说明；对于static元素为auto；对于长度值，是相应的绝对长度；对于百分数值，则为指定的值；否则，为auto
说明：	计算值取决于一系列因素；见附录A中的相关例子

这些属性描述了距离包含块最近边的偏移（所以得名*offset*）。例如，top描述了定位元素上外边距边界离其包含块的顶端多远。如果top为正值，会把定位元素的上外边距边界下移，若为负值，则会把定位元素的上外边距移到其包含块的顶端之上。类似地，left描述了定位元素的左外边距边界在其包含块左边界右边（正值）或左边（负值）有多远。如果是正值，会把定位元素的外边距边界移到包含块左边界右边，而负值则将其移到包含块左边界左边。

还可以这样来讲，正值会导致向内偏移，使边界朝着包含块的中心移动，而负值会导致向外偏移。

注意： 最初的CSS2规范实际上指出，偏移的是内容边界，而不是外边距边界，不过这与CSS2的其他部分不一致。这个错误在后来的勘误和CSS2.1中得到修正。所有当前开发的浏览器（写作本书时）都使用外边距边界来完成偏移计算。

偏移定位元素的外边距边界时，带来的影响是元素的所有一切（包括外边距、边框、内边距和内容）都会在定位的过程中移动。因此，可以为定位元素设置外边距、边框和内边距；这些会随着定位元素一直保留，并包含在偏移属性定义的区域内。

要记住重要的一点，偏移属性定义了距离包含块相应边的偏移（例如，left定义了距离包含块左边的偏移），而不是距包含块左上角的偏移。正是因为这个原因，填充一个包含块的右下角时要使用以下值：

```
top: 50%; bottom: 0; left: 50%; right: 0;
```

在这个例子中，定位元素的左外边界放在包含块中间的位置上，这是距包含块左边界的偏移。不过，定位元素的右外边界没有从包含块的右边界偏移，所以二者不矛盾。定位元素的顶端和底端也是如此：外上边界放在包含块中间的位置上，但是外下边界没有从包含块底端上移。这就得到了如图10-26所示的结果。

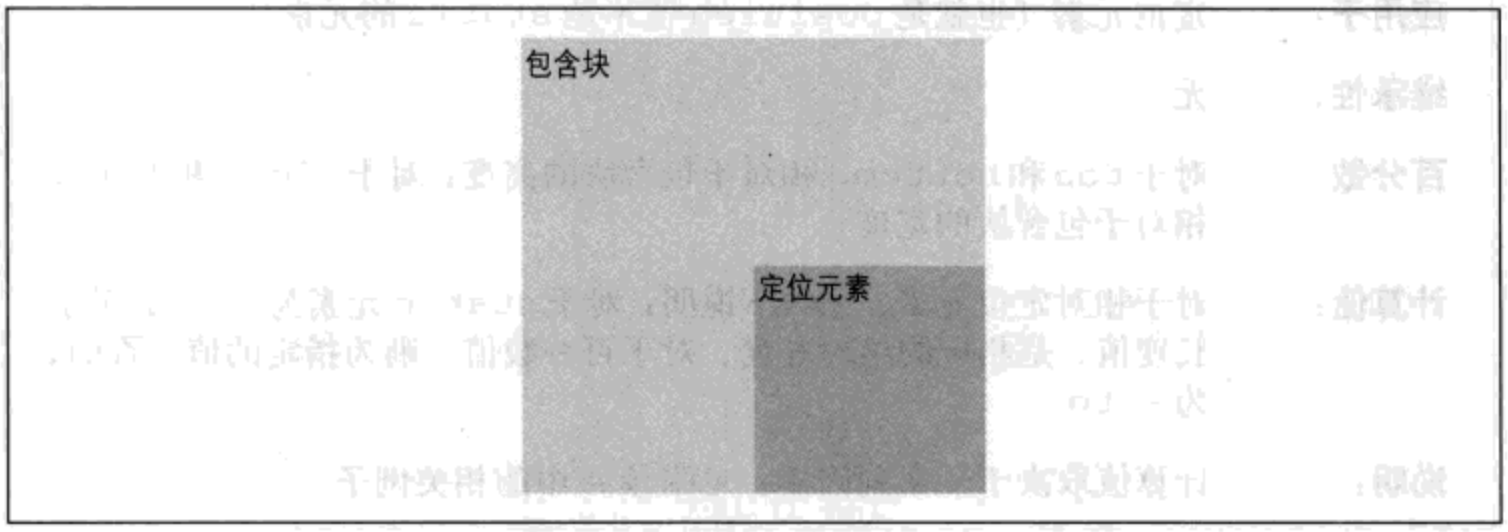

图10-26：填充包含块的右下1/4部分

注意：在图10-26以及本章大多数例子中，都是基于绝对定位的。由于绝对定位对于展示top、right、bottom和left如何工作是最简单的机制，所以我们一直都会用绝对定位。

需要说明，定位元素的背景色稍有不同。图10-26中没有外边距，但是如果确实有外边距，在边框和偏移边界之间就会出现空白。看上去就好像定位元素没有完全填充包含块右下方1/4部分一样。实际上，它确实填满了这个区域，不过人眼不能直接看出。因此，以下两组样式会得到几乎相同的外观效果，假设包含块为100em高，100em宽：

```
top: 50%; bottom: 0; left: 50%; right: 0; margin: 10em;
top: 60%; bottom: 10%; left: 60%; right: 10%; margin: 0;
```

重申一句，二者只是在视觉效果上相似。

通过使用负值，可以将一个元素定位到其包含块之外。例如，指定以下值会得到如图10-27所示的结果：

```
top: -5em; bottom: 50%; left: 75%; right: -3em;
```

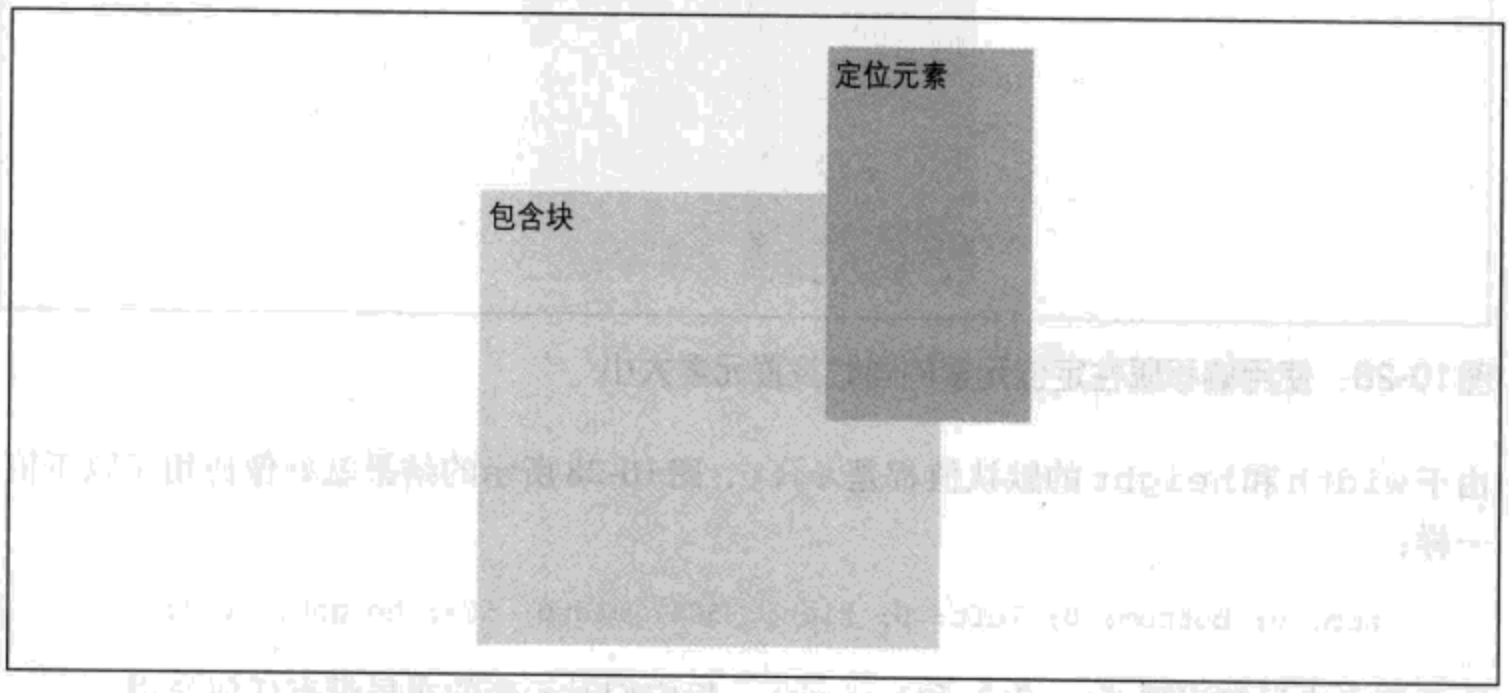

图10-27：将一个元素定位到其包含块之外

除了长度值和百分数值，偏移属性还可以设置为auto，这是默认值。对于auto没有定义行为；它会根据所用的定位类型改变。本章后面将逐个考虑各种定位类型，并介绍auto如何作用。

宽度和高度

很多情况下，如果已经确定将元素定位到哪里，接下来可能希望声明该元素应当有多高、多宽。另外，有时你可能想限制一个定位元素的高度和宽度，还有一些情况下也许你希望浏览器自动地计算宽度或高度（或者宽度和高度都由浏览器自动计算）。

设置宽度和高度

如果想为定位元素指定一个特定的宽度，显然要用width属性。类似地，利用height可以为定位元素声明特定的高度。

尽管有时设置一个元素的宽度和高度很重要，但对于定位元素来说则不一定必要。例如，如果使用top、right、bottom和left来描述元素4个边的放置位置，那么元素的高度和宽度将由这些偏移隐含确定。假设你希望一个绝对定位元素从上到下填充其包含块的左半部分。可以使用以下值，其结果见图10-28所示：

```
top: 0; bottom: 0; left: 0; right: 50%;
```

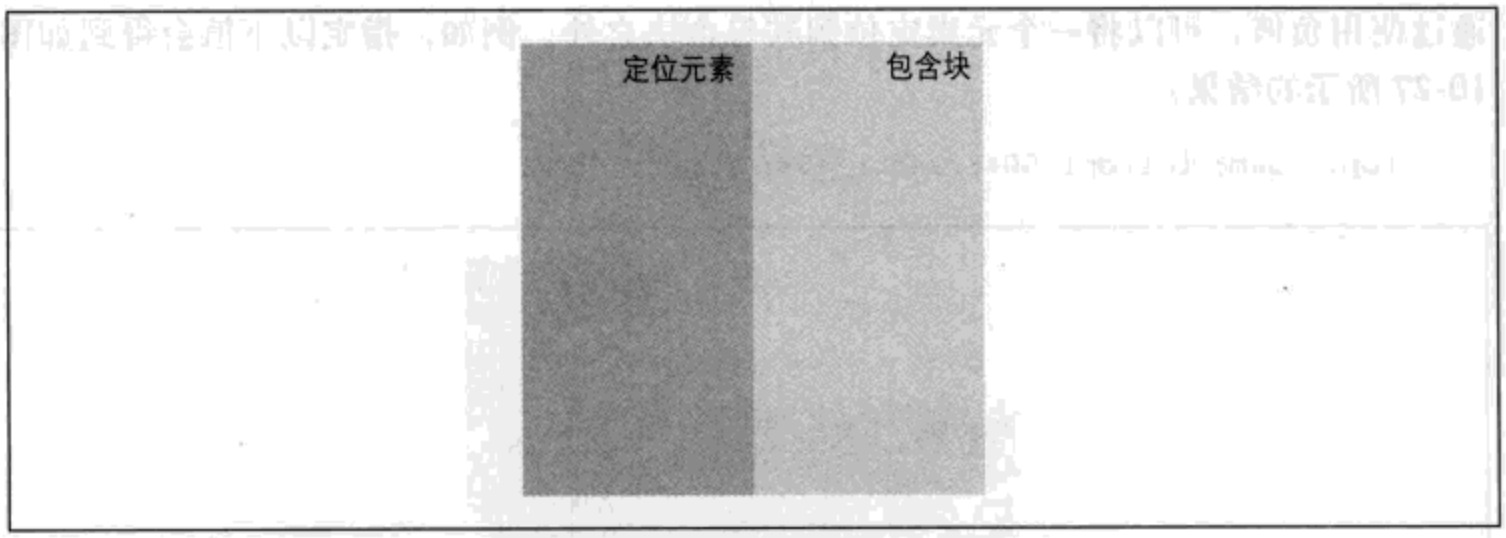

图10-28：使用偏移属性定位元素的同时设置元素大小

由于width和height的默认值都是auto，图10-28所示的结果就好像使用了以下值一样：

```
top: 0; bottom: 0; left: 0; right: 50%; width: 50%; height: 100%;
```

这个例子中尽管出现了width和height，但它们对元素的布局没有任何作用。

当然，倘若向元素增加了外边距、边框或内边距，此时如果为height和width显式指定了值，情况就不同了：

```
top: 0; bottom: 0; left: 0; right: 50%; width: 50%; height: 100%;
  padding: 2em;
```

这会使定位元素延伸超出其包含块，如图10-29所示。

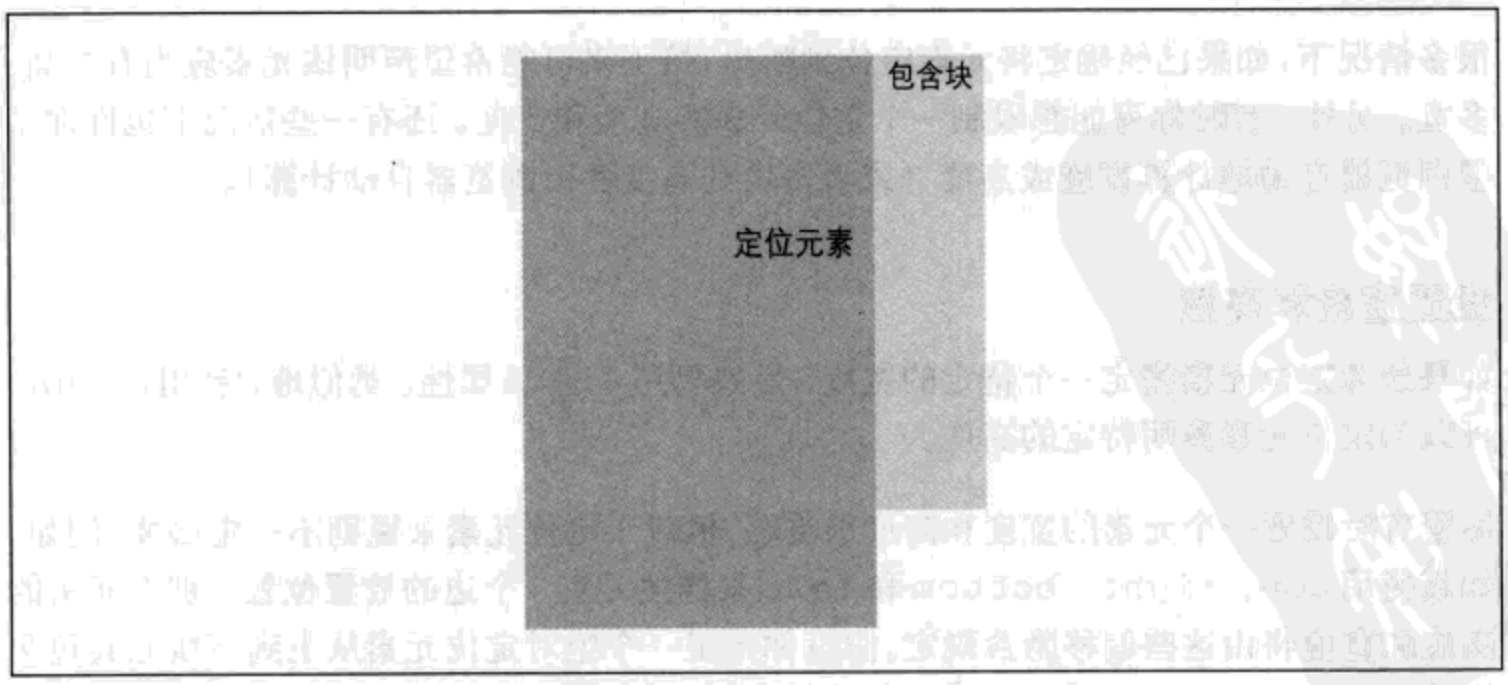

图10-29：定位元素，使其部分超出包含块

在前几章已经看到，之所以会发生这种情况，原因在于向内容区增加了内边距，而且内容区的大小由height和width的值确定。要得到你想要的内边距，而且仍保证元素不超出其包含块，可以去除height和width声明，或者显式地将其设置为auto。

限制宽度和高度

如果有必要（或需要），可以使用以下CSS2属性对元素的宽度加一些限制，我把这些属性称为最小最大属性（min-max properties）。通过使用min-width和min-height，可以为元素的内容区定义一个最小尺寸。

min-width、min-height

值：	<length> | <percentage> | inherit
初始值：	0
应用于：	除了非替换行内元素和表元素以外的所有元素
继承性：	无
百分数：	相对于包含块的宽度
计算值：	对于百分数，根据指定确定；对于长度值，则为绝对长度；否则，为none

类似地，还可以使用属性max-width和max-height来限制元素的尺寸。

max-width、max-height

值：	<length> | <percentage> | none | inherit
初始值：	none
应用于：	除了非替换行内元素和表元素以外的所有元素
继承性：	无
百分数：	相对于包含块的高度
计算值：	对于百分数，根据指定确定；对于长度值则为绝对长度；否则，为none

这些属性的作用可以顾名思义。还有一点最初看来可能不太明显，不过仔细考虑会发现很有道理，这就是这些属性的值不能为负。

警告： Windows平台的Internet Explorer在IE7之前都不支持min-height、min-width、max-height和max-width。

以下样式要求定位元素至少10em宽、20em高，如图10-30所示：

```
top: 10%; bottom: 20%; left: 50%; right: 10%;
  min-width: 10em; min-height: 20em;
```

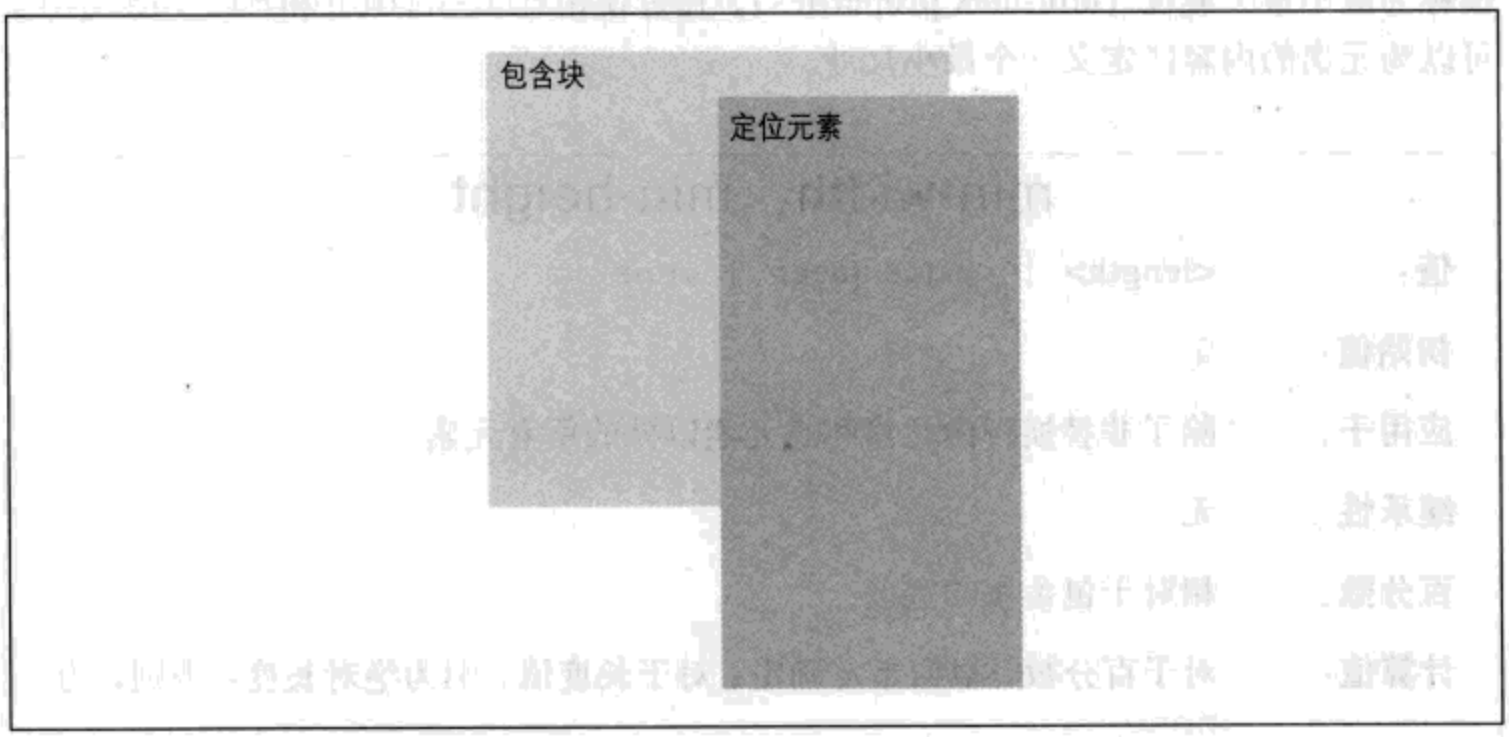

图10-30：为定位元素设置最小和最大高度

这个解决方案不是很健壮，因为它要求元素至少是某个特定的大小，而不论其包含块有多大。下面是一个更好的解决方案：

```
top: 10%; bottom: auto; left: 50%; right: 10%; height: auto;
  min-width: 15em;
```

在此，元素宽度应当是包含块的40%，不过不能小于15em。在此还改变了bottom和height，使之自动确定。这样元素就会足够高（足以显示其内容），而不论它会多窄（当然不能小于15em!）。

注意： 下一节将介绍auto对于定位元素的height和width所起的作用。

还可以换个角度，通过使用max-width和max-height使得元素不至于太宽或太高。下面考虑这样一种情况，出于某种原因，希望一个元素的宽度是其包含块的3/4，不过如果达到400像素就不能更宽了。以下是一个合适的样式：

```
left: 0%; right: auto; width: 75%; max-width: 400px;
```

最小最大属性的一个很大的好处是，可以相对安全地混合使用不同的单位。使用百分数大小的同时，可以设置基于长度的限制，反之亦然。

需要指出，这些最小最大属性结合浮动元素使用时可能也非常有用。例如，可以允许一个浮动元素的宽度相对于其父元素（即其包含块）的宽度，同时确保该浮动元素的宽度不小于10em。反过来使用也是可以的：

```
p.aside {float: left; width: 40em; max-width: 40%;}
```

这会把浮动元素设置为40em宽，除非这超过了包含块宽度的40%，在这种情况下浮动元素会变窄。

注意：讨论各种类型的定位时还会再来讨论元素的大小设置。

内容溢出和剪裁

如果一个元素的内容对于元素大小来说过大，就有可能溢出元素本身。对于这种情况，有一些候选解决办法，CSS2允许你从中选择。它还允许定义一个剪裁区域，如果出了这个剪裁区域，这种溢出就会带来问题。

溢出

假设出于某种原因，一个元素固定为某个特定大小，但内容在元素中放不下。此时就可以利用overflow属性控制这种情况。

overflow	
值：	visible | hidden | scroll | auto | inherit
初始值：	visible
应用于：	块级元素和替换元素
继承性：	无
计算值：	根据指定确定

默认值visible表明，元素的内容在元素框之外也可见。一般地，这会导致内容超出其自己的元素框，但不会改变框的形状。以下标记会得到如图10-31所示的结果：

```
div#sidebar {position: absolute; top: 0; left: 0; width: 25%; height: 7em;
  background: #BBB; overflow: visible;}
```

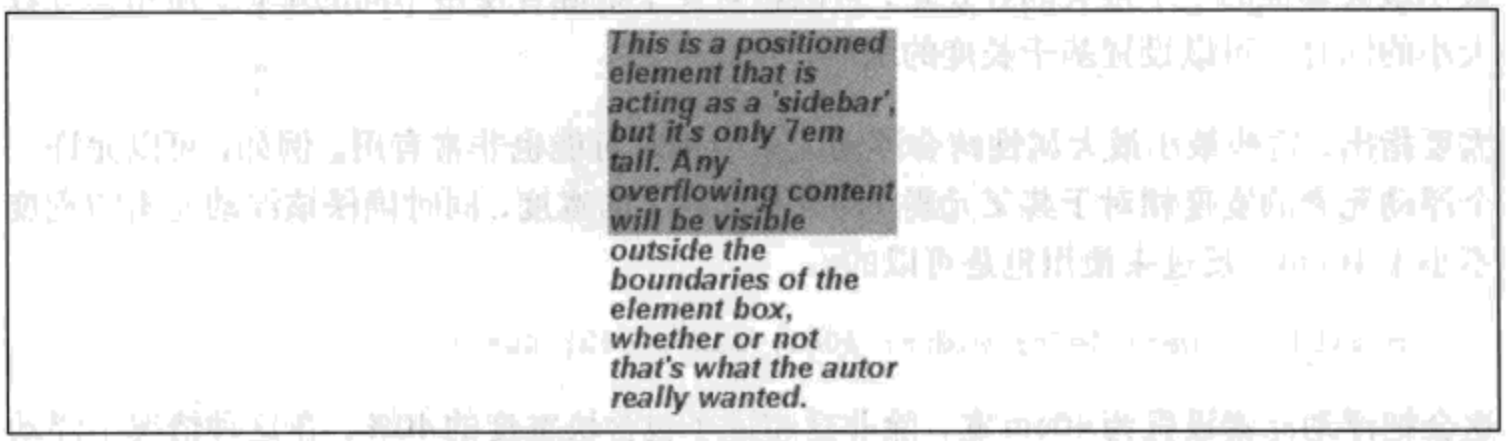

图10-31：可以看到内容溢出了元素框

如果overflow设置为scroll，元素的内容会在元素框的边界处剪裁——也就是说，溢出的部分将无法看到，不过还是有办法让用户得到这些额外的内容。在Web浏览器中，这可能意味着使用一个滚动条（或类似的东西）或者使用另外某种方法访问内容而不会改变元素本身的形状。图10-32所示就是一种可能的情况，这是由以下标记得到的：

```
div#sidebar {position: absolute; top: 0; left: 0; width: 15%; height: 7em;
  overflow: scroll;}
```

如果使用scroll，应该始终提供某种滚动机制（例如滚动条）。援引规范中的说法，“这会避免动态环境中滚动条出现或消失所带来的问题”。因此，即使元素有足够的空间显示所有内容，也应当显示滚动条。另外，打印一个页面或在打印媒体中显示文档时，内容可能显示为就好像overflow值声明为visible一样。

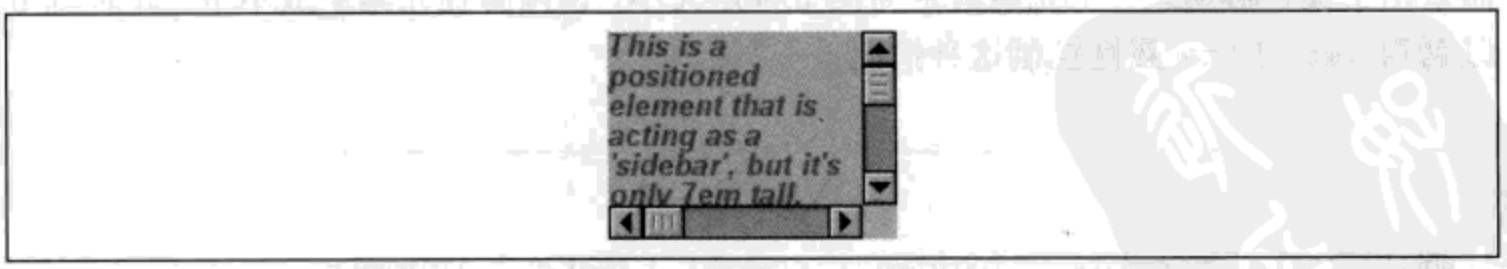

图10-32：通过滚动机制使溢出内容可用

如果overflow被设置为hidden，元素的内容会在元素框的边界处剪裁，不过不会提供滚动接口使用户访问超出剪裁区域的内容。考虑以下标记：

```
div#sidebar {position: absolute; top: 0; left: 0; width: 15%; height: 7em;
  overflow: hidden;}
```

在这种情况下，被剪裁的内容对用户来说不可用。这会得到如图10-33所示的结果。

图10-33：在内容区边界处剪裁内容

最后还有一个值overflow: auto。这允许用户代理来确定采用何种行为，不过都建议在必要时提供一个滚动机制。这可能是overflow的一个有用的用法，因为用户代理可以把它解释为“只在必要时提供滚动条”（也可能有些用户代理不这样解释，不过一般来讲，用户代理当然可以而且往往应当这样解释）。

内容剪裁

如果一个绝对定位元素的内容溢出其内容框，而且overflow设置为要求剪裁该内容，通过使用属性clip可以改变剪裁区域的形状。

默认值auto表示元素的内容不应剪裁。还可以相对于元素内容区定义一个剪裁形状。这不会改变内容区的形状，而只是改变将显示内容的区域形状。

注意： 尽管CSS2中唯一可用的剪裁形状是矩形，不过规范确实指出，在将来的规范中可能会有其他形状。

clip

值：	rect(*top*,*right*,*bottom*,*left*) | auto | inherit
初始值：	auto
应用于：	绝对定位元素（在CSS2中，clip应用于块级元素和替换元素）
继承性：	无
计算值：	对于矩形，4个计算长度表示剪裁矩形区域的4个边；否则，根据指定确定

这是利用形状值rect(*top*,*right*,*bottom*,*left*)实现的。可以如下指定剪裁区域内不做修改：

```
clip: rect(0, auto, auto, 0);
```

rect的语法很有意思。从技术上讲，可以是rect(*top*,*right*,*bottom*,*left*)，注意这里的逗号，不过CSS2规范提供的例子中有些有逗号，有些没有逗号，而且将rect定义为这两种版本都能接受。在此，我们还是采用有逗号的版本，主要是因为这样更易读，而且这是CSS2.1中推荐的做法。

有一点极为重要，rect(...)的值不是边偏移，而是距元素左上角的距离（在从右向左读的语言中，则是与元素右上角的距离）。因此，如果一个剪裁矩形涵盖元素左上角20 × 20像素的一个正方形，可以定义如下：

```
rect(0, 20px, 20px, 0)
```

rect(...)只允许长度值和auto，如果设置为auto，这相当于将剪裁边界设置为适当的内容边界。因此，以下两个语句含义相同：

```
clip: rect(auto, auto, 10px, 1em);
clip: rect(0, 0, 10px, 1em);
```

由于clip中的所有偏移都是距左上角的偏移，所以不允许有百分数，实际上不可能创建一个“中心”剪裁区域，除非你知道元素本身的大小。考虑以下情况：

```
div#sidebar {position: absolute; top: 0; bottom: 50%; right: 50%; left: 0;
  clip: rect(1em,4em,6em,1em);}
```

由于无法知道元素会有多高或多宽，所以无法定义这样一个剪裁矩形，要求它最后在元素内容区向右1em或向下1em处。要想知道元素的大小，唯一的办法就是设置元素本身的高度和宽度：

```
div#sidebar {position: absolute; top: 0; left: 0; width: 5em; height: 7em;
  clip: rect(1em,4em,6em,1em);}
```

这会得到如图10-34所示的结果，这里增加了一个虚线表示剪裁区域的边界。这条线在显示文档的用户代理中并不会真的出现。

不过，可以设置负长度值，这会使剪裁区域延伸到元素框之外。如果你想将剪裁区域向上向左移0.25英寸，可以使用以下样式（如图10-35所示）：

```
clip: rect(-0.25in, auto, auto, -0.25in);
```

可以看到，这没什么好处。尽管剪裁区域向上向左延伸了，不过由于延伸区域内没有任何内容，所以没有多大差别。

另一方面，完全可以超越下边界和右边界，但不能超越上边界或左边界。图10-36显示了以下样式的结果（要记住，这里的虚线只是为了说明，并不实际存在！）：

```
div#sidebar {position: absolute; top: 0; left: 0; width: 5em; height: 7em;
  clip: rect(0,6em,9em,0);}
```

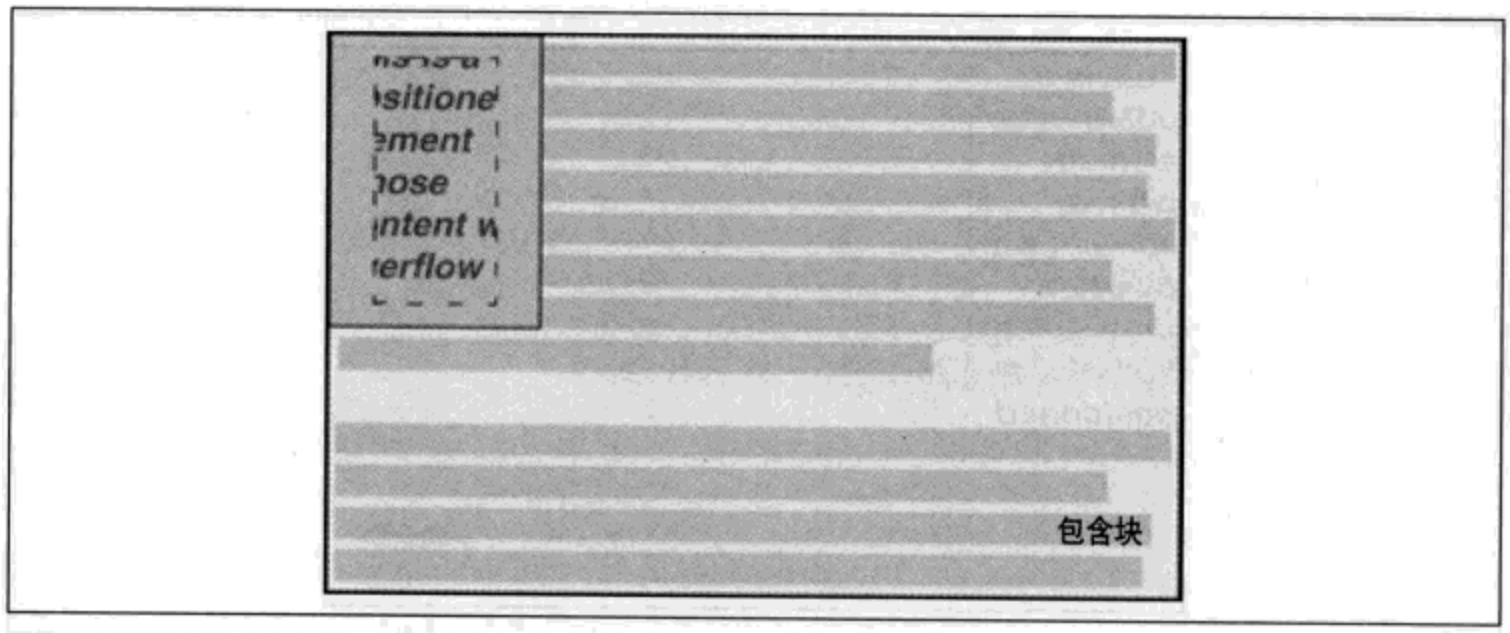

图10-34：为溢出内容设置剪裁区域

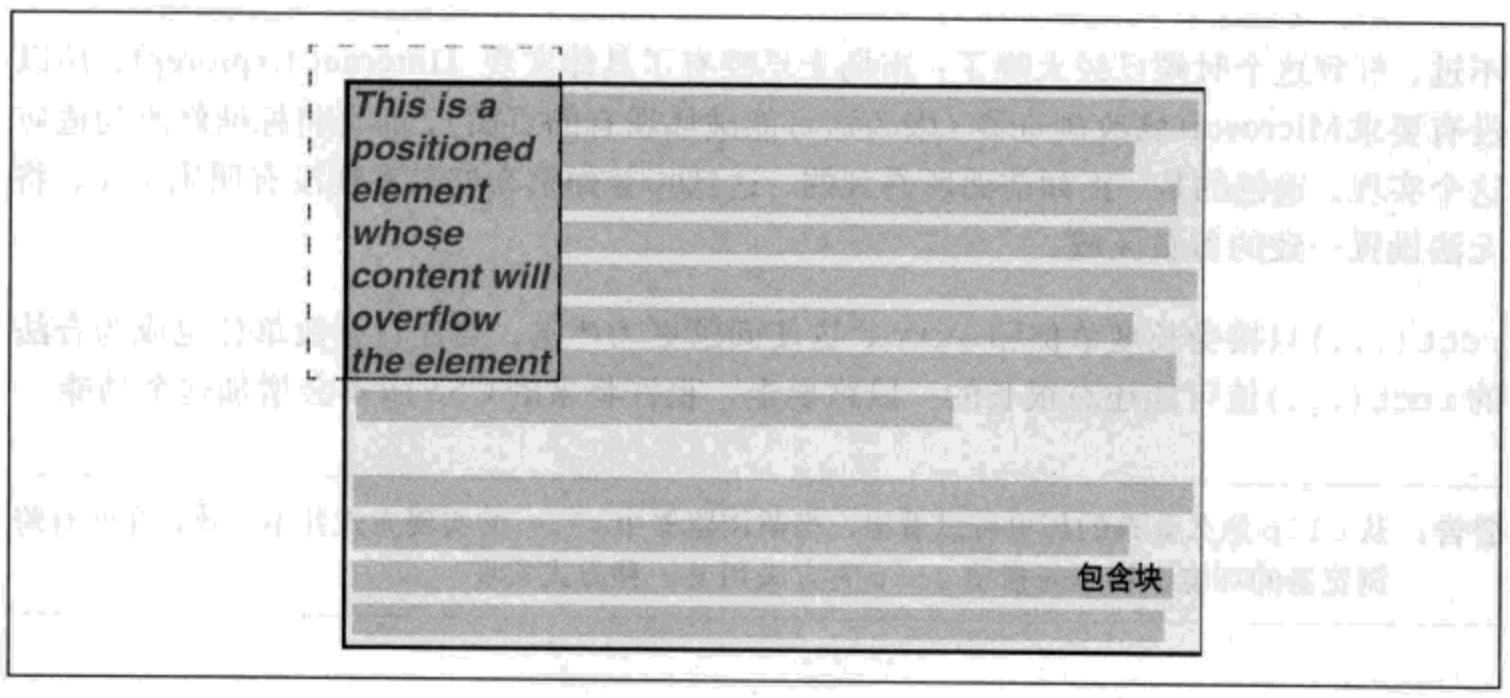

图10-35：将剪裁区域延伸到元素框之外

这会扩展可以看到内容的区域。不过，内容流并没有改变，所以唯一的视觉效果就是元素下面可以看到更多的内容。文本没有向右流，因为其行框的宽度仍受定位元素宽度的限制。如果有一个比定位元素宽的图像，或者有一个很长的预定义格式文本行，则可能在定位元素的右边也可见（直到剪裁区域结束的位置）。

你可能已经认识到，`rect(...)`的语法与CSS的其他语法相比不太一样。原因是它基于早期的定位草案，而该草案使用了左上偏移机制。在CSS2之前，实现这个语法的Internet Explorer已经成为完备推荐，所以它与最终的`rect(...)`语法冲突，因为最后已经将`rect(...)`修改为使用边偏移（类似于CSS2的其余部分）。这么做是为了保证定位一致，这是合情合理的。

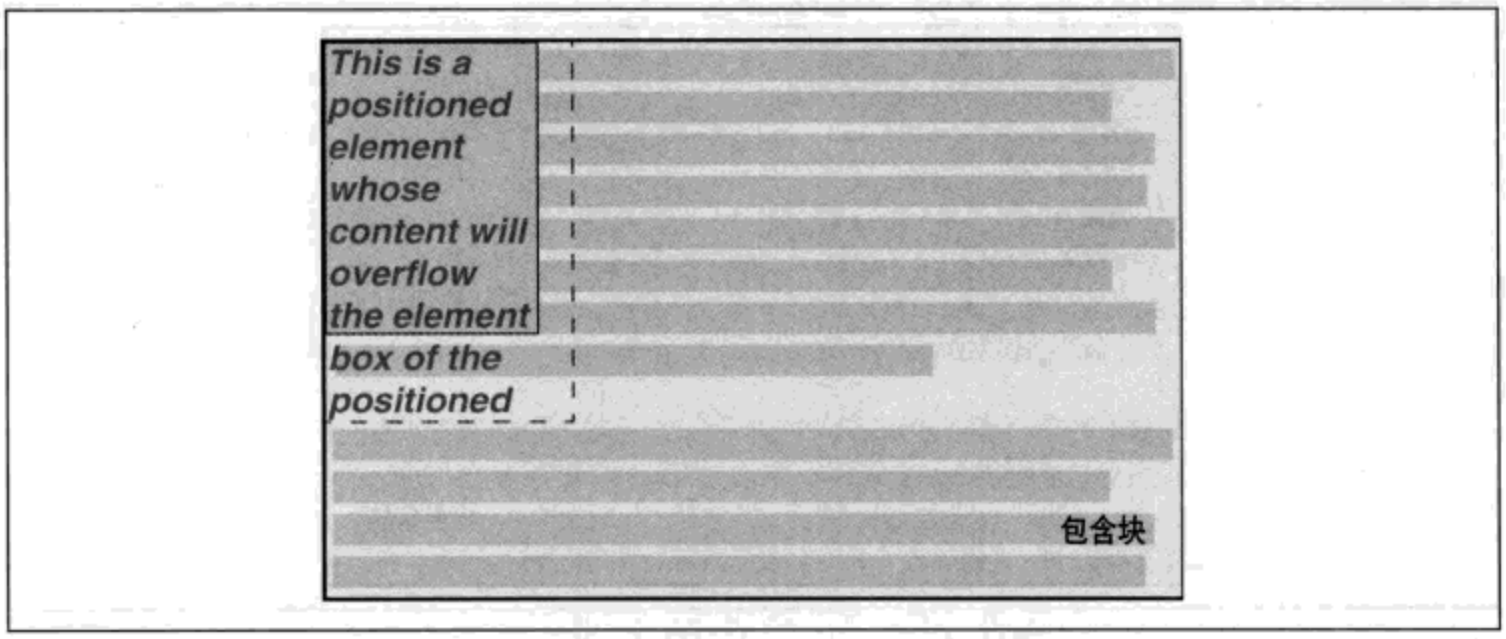

图10-36：剪裁区域从元素框向下向右延伸

不过，等到这个时候已经太晚了：市场上已经有了具体实现（Internet Explorer），所以没有要求Microsoft修改浏览器（因为这可能破坏现有的页面），而是将标准修改为适应这个实现。遗憾的是，正如前面所看到的，这意味着如果高度和宽度没有明确定义，将无法设置一致的剪裁区域。

rect(...)只接受长度单位和auto，这使问题更为严重。要让百分数单位也成为合法的rect(...)值可能还有很长的一段路要走，也许将来的CSS版本会增加这个功能。

警告： 从clip悠久复杂的历史可以看出，当前浏览器中clip的实现方式并不一致，在所有跨浏览器的环境中都不能指望clip肯定采用某一种方式实现。

元素可见性

除了剪裁和溢出，还可以控制整个元素的可见性。

visibility

值：	visible | hidden | collapse | inherit
初始值：	visible
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

这个属性相当简单。如果元素设置为有visibility: visible，当然它就是可见的。

如果元素设置为visibility: hidden，则会置为“不可见”（按规范中的说法）。处于不可见状态时，元素还是会影响文档的布局，就好像它还可见一样。换句话说，元素还在那里，只不过你看不见它。注意这与display: none有区别。对于后者，元素不仅不显示，还会从文档中删除，所以对文档布局没有任何影响。图10-37显示了一个文档，根据以下样式和标记，将其中一个段落设置为hidden：

```
em.trans {visibility: hidden; border: 3px solid gray; background: silver;
  margin: 2em; padding: 1em;}

<p>
This is a paragraph that should be visible. Lorem ipsum, dolor sit amet,
<em class="trans">consectetuer adipiscing elit, sed diam nonummy nibh </em>
euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.
</p>
```

hidden元素中原本可见的部分（如内容、背景和边框）都会置为不可见。注意，这些空间还留在原处，因为元素仍是文档布局的一部分。只不过你看不到它。

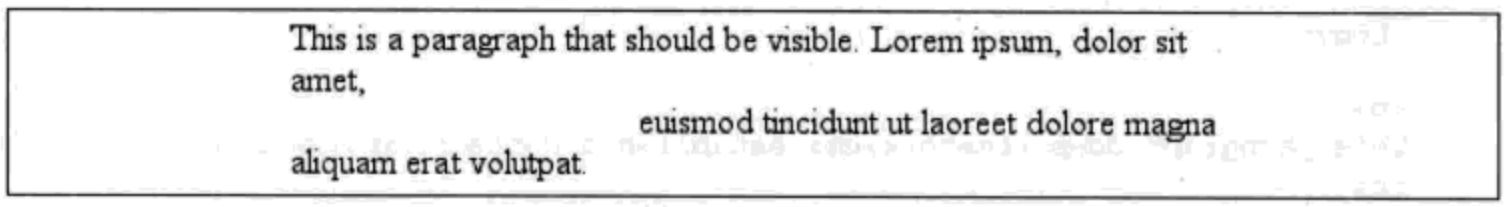

图10-37：使元素不可见，但不删除其元素框

还要注意，有可能将一个hidden元素的后代元素置为visible。这会使该后代元素正常出现，尽管其祖先元素（以及兄弟）是不可见的。为此，必须显式地声明后代元素为visible，因为visibility属性可以继承：

```
p.clear {visibility: hidden;}
p.clear em {visibility: visible;}
```

visibility: collapse值在CSS表显示中使用，这在下一章讨论。根据CSS2规范，如果用于非表元素，collapse与hidden含义相同。

绝对定位

由于上一节中的大多数例子和例图描述的都是绝对定位，你应该对绝对定位如何工作有了一定的了解。接下来主要介绍使用绝对定位时的具体细节。

包含块和绝对定位元素

元素绝对定位时，会从文档流中完全删除。然后相对于其包含块定位，其边界根据偏移属性（top、left等）放置。定位元素不会流入其他元素的内容，反之亦然。这说明，绝对定位元素可能覆盖其他元素，或者被其他元素覆盖（本章后面会看到，你可以影响这种覆盖顺序）。

绝对定位元素的包含块是最近的position值不为static的祖先元素。创作人员通常会选择一个元素作为绝对定位元素的包含块，将其position指定为relative而且没有偏移：

```
p.contain {position: relative;}
```

考虑图10-38中的例子，这是由以下样式得到的结果：

```
p {margin: 2em;}
p.contain {position: relative;}    /* establish a containing block*/
b {position: absolute; top: auto; right: 0; bottom: 0; left: auto;
   width: 8em; height: 5em; border: 1px solid gray;}
```

```
<body>

<p>
This paragraph does <em>not</em> establish a containing block for any of
its
descendant elements that are absolutely positioned.  Therefore, the
absolutely
positioned <b>boldface </b> element it contains will be positioned with
respect to the initial containing block.
</p>
<p class="contain">
Thanks to 'position: relative', this paragraph establishes a containing
block for any of its descendant elements that are absolutely positioned.
Since there is such an element-- <em>that is to say, <b>a boldfaced
element
that is absolutely positioned,</b> placed with respect to its containing
block (the paragraph)</em>, it will appear within the element box
generated
by the paragraph.
</p>
</body>
```

两个段落中的b元素都是绝对定位。其区别在于各元素所用的包含块。第一段中的b元素相对于初始包含块定位，因为它的所有祖先元素的position都是static。不过，第二个段落设置为position: relative，所以它为其后代元素建立了一个包含块。

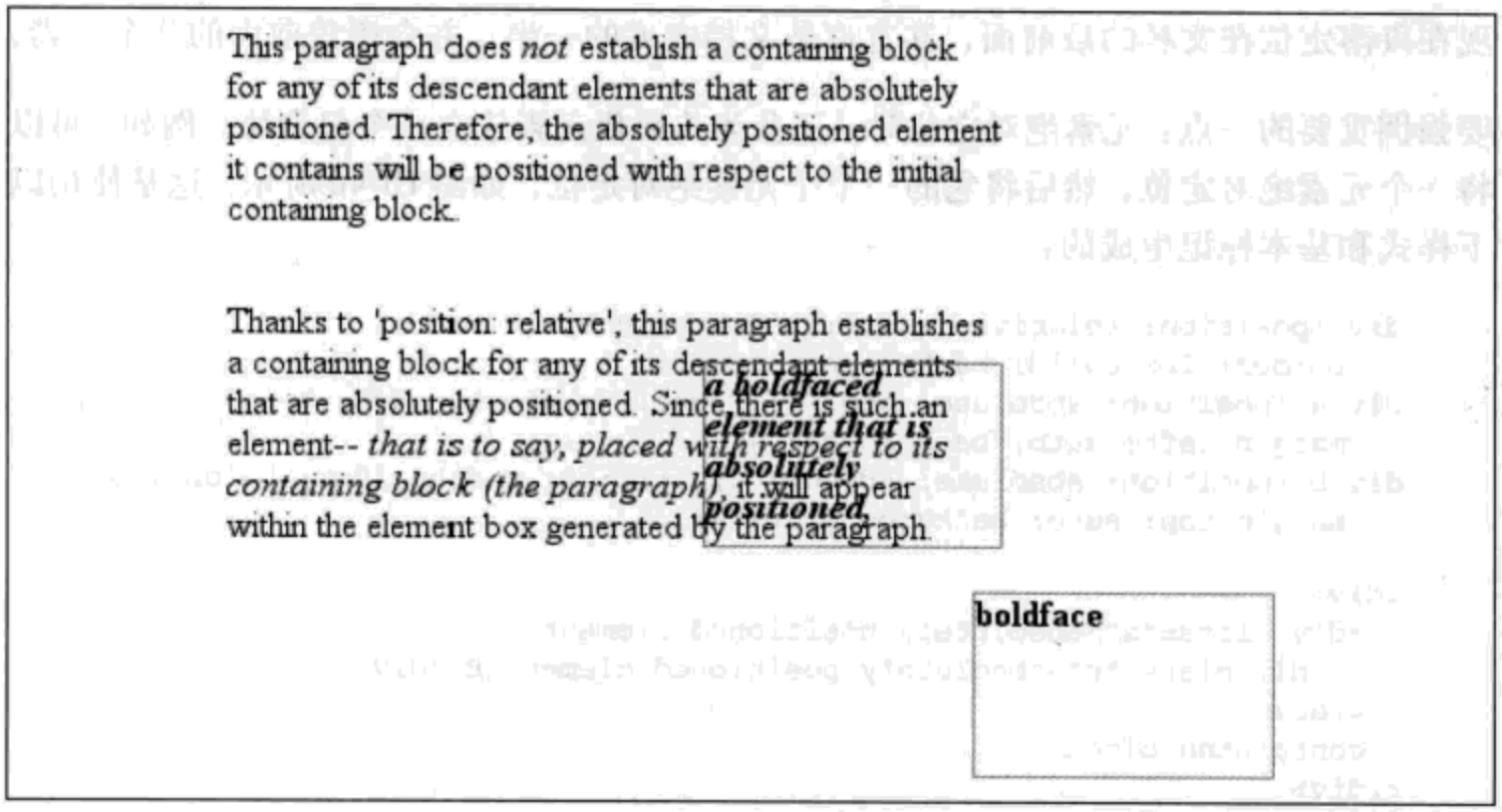

图10-38：使用相对定位来定义包含块

你可能已经注意到，在第二段中，定位元素覆盖了段落中的部分文本内容。这是没有办法避免的，因为无法将b元素定位到段落之外（比如right或其他某个偏移属性使用负值），也无法为段落指定一个足够宽的内边距来容纳定位元素。另外，由于b元素有一个透明背景，所以会透过这个定位元素看到段落的文本。要避免这种情况，唯一的办法就是为定位元素设置一个背景，或者将其从段落中完全去除。

有时，你可能想确保body元素为其所有后代建立一个包含块，而不是让用户代理选择初始包含块。这很简单，只需如下声明：

```
body {position: relative;}
```

在这样一个文档中，可以随便放置绝对定位段落（如下），这会得到如图10-39所示的结果：

```
<p style="position: absolute; top: 0; right: 25%; left: 25%; bottom: auto;
 width: 50%; height: auto; background: silver;">...</p>
```

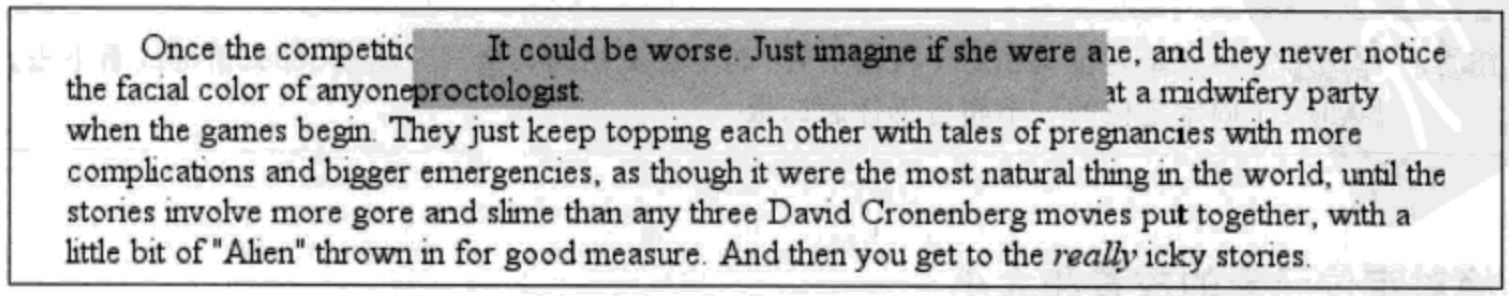

图10-39：定位一个元素，其包含块为根元素

现在段落定位在文档的最前面，其宽度是文档宽度的一半，并会覆盖前面的几个元素。

要强调重要的一点：元素绝对定位时，还会为其后代元素建立一个包含块。例如，可以将一个元素绝对定位，然后将它的一个子元素绝对定位，如图 10-40 所示，这是使用以下样式和基本标记生成的：

```
div {position: relative; width: 100%; height: 10em;
   border: 1px solid; background: #EEE;}
div.a {position: absolute; top: 0; right: 0; width: 15em; height: 100%;
   margin-left: auto; background: #CCC;}
div.b {position: absolute; bottom: 0; left: 0; width: 10em; height: 50%;
   margin-top: auto; background: #AAA;}

<div>
  <div class="a">absolutely positioned element A
    <div class="b">absolutely positioned element B</div>
  </div>
  containing block
</div>
```

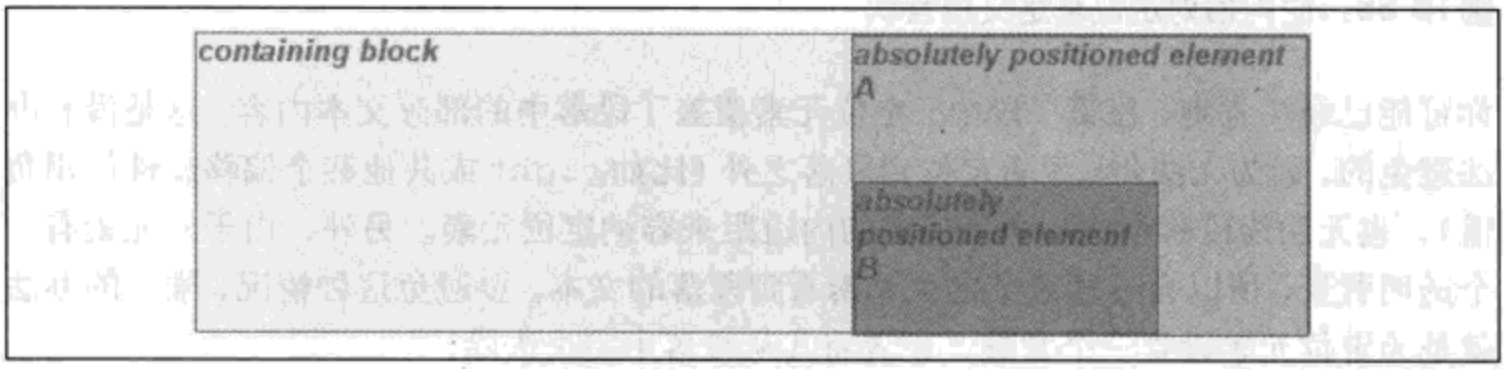

图 10-40：绝对定位元素建立包含块

要记住，如果文档可滚动，定位元素会随着它滚动。只要绝对定位元素不是固定定位元素的后代，情况都是如此。其原因是元素最终会相对于正常流的某一部分定位。例如，如果将一个表绝对定位，其包含块是初始包含块，那么这个表就会随文档滚动，因为初始包含块是正常流的一部分。类似地，即使建立一个嵌套 4 层的绝对定位元素，“最外层”元素总是会相对于初始包含块定位。因此，它会随着初始包含块滚动，而其所有后代元素也会随之滚动。

注意： 如果你想将元素定位为相对于视窗放置，而不随文档的其余部分滚动，那么请继续看下去。后面关于固定定位的一节将告诉你怎么做。

绝对定位元素的放置和大小

把“放置”和“大小”这两个概念放在一起看上去有些奇怪，不过对于绝对定位元素来

说，这是必要的，因为规范把它们紧密地绑在一起。如果仔细考虑，这也不那么奇怪。请考虑如果一个元素使用4个偏移属性来定位会发生什么情况，如下：

```
#masthead h1 {position: absolute; top: 1em; left: 1em; right: 25%; bottom: 10px;
  margin: 0; padding: 0; background: silver;}
```

在此，h1元素框的高度和宽度由其外边距边界的放置决定，如图10-41所示。

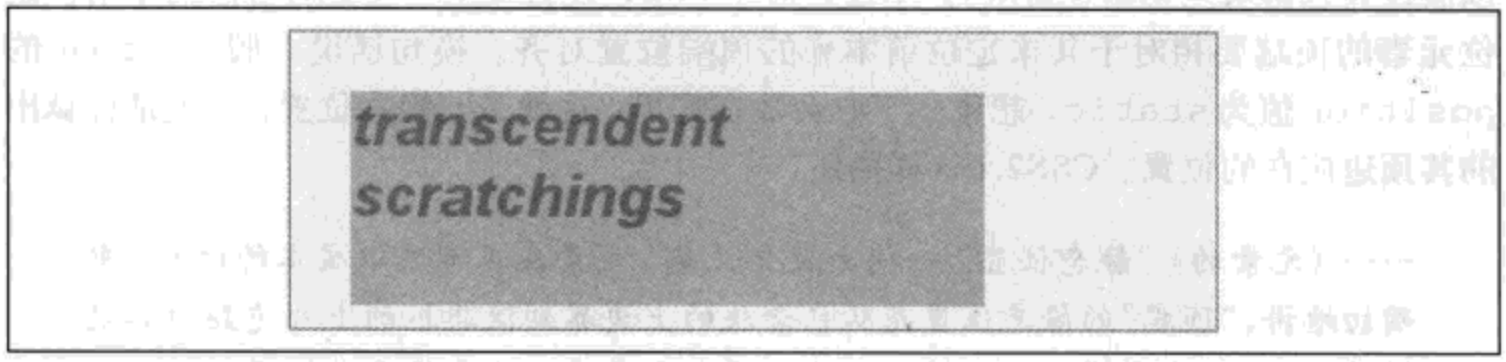

图10-41：根据偏移属性确定元素的高度

如果包含块更高，那么h1也会更高；如果包含块更窄，h1也会跟着更窄。如果向h1增加外边距或内边距，这会对h1的高度和宽度计算值有进一步的影响。

但是这样做之后，再试图设置一个显式的高度和宽度会怎么样呢？

```
#masthead h1 {position: absolute; top: 0; left: 1em; right: 10%; bottom: 0;
  margin: 0; padding: 0; height: 1em; width: 50%; background: silver;}
```

必须做些工作，因为所有这些值都正确是不太可能的。实际上，包含块宽度必须是h1的font-size计算值的2.5倍，这样以上的值才能正确。如果是其他宽度，则说明至少有一个值是错误的，相应地必须将其忽略。到底哪一个值是错误的，这取决于很多因素，而且根据元素是替换元素还是非替换元素还会有所不同。

为此考虑以下规则：

```
#masthead h1 {position: absolute; top: auto; left: auto;}
```

结果会是什么呢？可以看到，答案并不是“将值重置为0”。下一节将介绍真正的答案。

自动边偏移

元素绝对定位时，如果除bottom外某个任意偏移属性设置为auto，会有一种特殊的行为。下面以top为例，考虑以下标记：

```
<p>
When we consider the effect of positioning, it quickly becomes clear that
authors
```

```
can do a great deal of damage to layout, just as they can do very
interesting
things.<span style="position: absolute; top: auto; left: 0;">[4]</span>
This is usually the case with useful technologies: the sword always has
at least two edges, both of them sharp.
</p>
```

会发生什么呢？对于left，很简单：元素的左边界会相对于其包含块的左边界放置（可以假设其包含块是初始包含块）。不过，对于top，还会发生一些更有意思的事情。定位元素的顶端要相对于其未定位前本来的顶端位置对齐。换句话说，假设span的position值为static，想象一下它会放在哪里；这就是其静态位置，也就是计算出的其顶边应在的位置。CSS2.1这样描述：

> ……（元素的）“静态位置”一词大致含义是：元素在正常流中原本的位置。更确切地讲，“顶端”的静态位置是从包含块的上边界到假想框的上外边距边界之间的距离（假想框是假设元素“position”属性为“static”时元素的第一个框）。如果这个假想的元素框在包含块的上面，则这个值为负。

因此，可以得到如图10-42所示的结果。

When we consider the effect of positioning, it quickly becomes clear that authors can do a great deal of damage to layout, just as [4] they can do very interesting things. This is usually the case with useful technologies: the sword always has at least two edges, both of them sharp.

图10-42：绝对定位一个元素，与其“静态”位置一致

“[4]”位于段落内容之外，因为初始包含块的左边界在该段落左边界的左边。

如果left和right设置为auto，也适用同样的基本规则。在这些情况下，定位元素的左（或右）边界与元素未定位时该边界原本的位置对齐。下面修改前面的例子，使top和left都设置为auto：

```
<p>
When we consider the effect of positioning, it quickly becomes clear that
authors
can do a great deal of damage to layout, just as they can do very
interesting
things.<span style="position: absolute; top: auto; left: auto;">[4]</span>
This is usually the case with useful technologies: the sword always has
at least two edges, both of them sharp.
</p>
```

这会得到如图10-43所示的结果。

When we consider the effect of positioning, it quickly becomes clear that authors can do a great deal of damage to layout, just as they can do very interesting things.[4] This is usually the case with useful technologies: the sword always has at least two edges, both of them sharp.

图 10-43：绝对定位一个元素，与其“静态”位置一致

“[4]”现在就位于其本来位置（即未定位前的位置）。注意，由于它已经定位，其正常流空间关闭。这会导致定位元素与正常流内容重叠。

注意： 应该注意到，CSS2和CSS2.1都指出在这样的情况下，“……用户代理可以自由猜测可能的[静态]位置”。当前浏览器在这方面做得很好，会按要求处理 top 和 left 的 auto 值，使元素的位置与其在正常流中原本的位置一致。

这种自动放置只在某些情况下可行，这些情况下通常对定位元素的其他尺寸没有什么限制。前面的例子可以自动放置，因为它对其高度或宽度没有任何限制，对下边界和右边界的放置也没有限制。不过，假设出于某种原因确实有这样一些限制会怎么样呢？请考虑以下标记：

```
<p>
When we consider the effect of positioning, it quickly becomes clear that
authors
can do a great deal of damage to layout, just as they can do very
interesting
things.<span style="position: absolute; top: auto; left: auto; right: 0;
bottom: 0; height: 2em; width: 5em;">[4]</span>  This is usually the case
with
useful technologies: the sword always has at least two edges, both of them
sharp.
</p>
```

此时无法满足以上的全部值。要确定此时会发生什么，这是下一节要讨论的问题。

非替换元素的放置和大小

一般地，元素的大小和位置取决于其包含块。各个属性（width、right、padding-left等）的值也会有一些影响，不过最主要的还是其包含块。

考虑一个定位元素的宽度和水平放置。这可以表示为一个等式：left+margin-left+border-left-width+padding-left+width+padding-right+border-right-width+margin-right+right= 包含块的 width。

这个计算是很合理的。基本上会由这个等式来确定正常流中块级元素的大小，除非增加了left和right。所有这些属性之间有什么关系呢？以下是一组规则。

首先，如果left、width和right都设置为auto，会得到上一节所示的结果：左边界置于其静态位置（假设使用从左向右读的语言）。在从右向左读的语言中，则是右边界置于其静态位置。元素的width设置为“收放得正好合适”，这说明该元素的内容区宽度恰好只能包含其内容（而没有多余空间）。这与表单元格的行为很类似。非静态位置属性（对于从左向右读的语言是right，对于从右向左读的语言是left）要适当设置，以保证余下的距离。例如：

```
<div style="position: relative; width: 25em; border: 1px dotted;">
An absolutely positioned element can have its content
<span style="position: absolute; top: 0; left: 0; right: auto; width:
auto;
  background: silver;">shrink-wrapped</span>
thanks to the way positioning rules work.
</div>
```

这会得到如图10-44所示的结果。

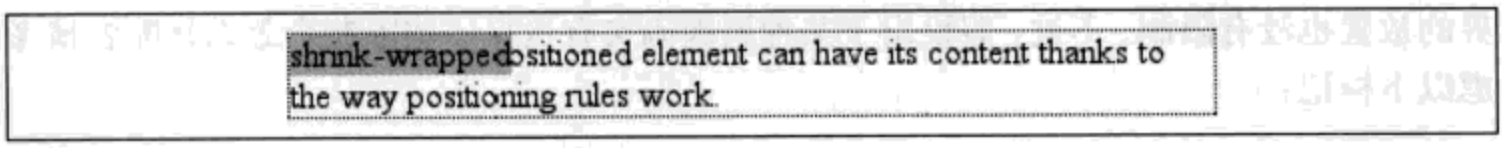

图10-44：绝对定位元素会“恰当收放”

元素的顶端根据其包含块（在这里就是div）的顶端放置，且元素的宽度刚好足够包含内容。从元素右边界到包含块右边界之间余下的距离则是right的计算值。

假设左右外边距都设置为auto，而left、width和right不是auto，如下例所示：

```
<div style="position: relative; width: 25em; border: 1px dotted;">
An absolutely positioned element can have its content
<span style="position: absolute; top: 0; left: 1em; right: 1em; width: 10em;
  margin: 0 auto; background: silver;">shrink-wrapped</span>
thanks to the way positioning rules work.
</div>
```

此时，左右外边距（原来设置为auto）会设置为相等的值。这实际上会让元素居中，如图10-45所示。

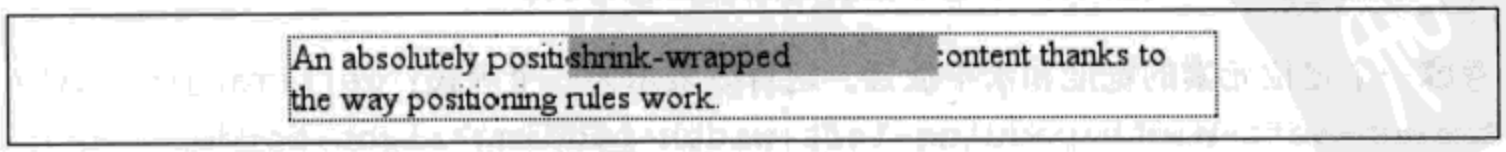

图10-45：将外边距为auto的绝对定位元素水平居中

这与正常流中的auto外边距居中行为基本上是一样的。所以，假设外边距不为auto：

```
<div style="position: relative; width: 25em; border: 1px dotted;">
An absolutely positioned element can have its content
<span style="position: absolute; top: 0; left: 1em; right: 1em; width: 10em;
  margin-left: 1em; margin-right: 1em; background: silver;">shrink-wrapped</span>
thanks to the way positioning rules work.
</div>
```

现在就有问题了。定位元素span的属性只增加为14em，而包含块宽度为25em。这里有11em的差额，必须从某个地方弥补。

规则指出，在这种情况下，用户代理会忽略right的值（这是在从左向右读的语言中；否则，在从右向左读的语言中将忽略left），并重置right的值。换句话说，其结果就好像声明了以下规则一样：

```
<span style="position: absolute; top: 0; left: 1em; right: 12em; width: 10em;
  margin-left: 1em; margin-right: 1em; background: silver;">shrink-wrapped</span>
```

其结果如图10-46所示。

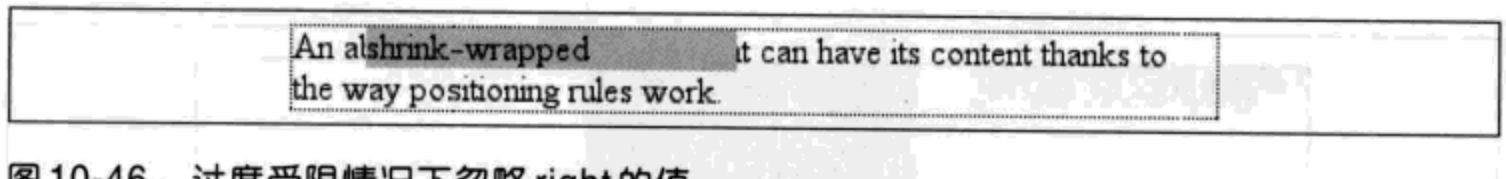

图10-46：过度受限情况下忽略right的值

如果某个外边距保持为auto，则会改变这个外边距。假设将样式改为：

```
<span style="position: absolute; top: 0; left: 1em; right: 1em; width: 10em;
  margin-left: 1em; margin-right: auto; background: silver;">shrink-wrapped</span>
```

视觉效果与图10-46所示相同，只不过这是通过将右外边距重新计算为12em得到的，而不是覆盖为属性right指定的值。另一方面，如果将左外边距置为auto，则会重置左外边距，如图10-47所示：

```
<span style="position: absolute; top: 0; left: 1em; right: 1em; width: 10em;
  margin-left: auto; margin-right: 1em; background: silver;">shrink-wrapped</span>
```

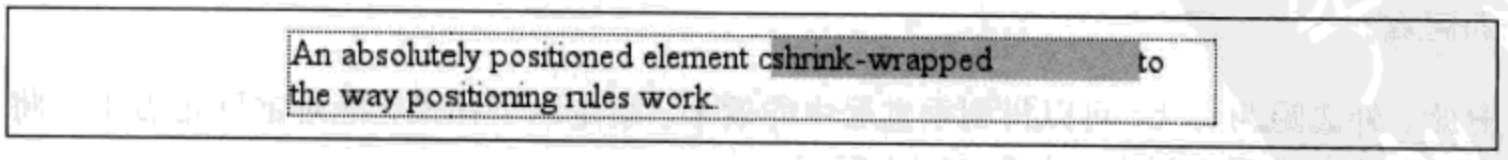

图10-47：过度受限情况下忽略margin-left的值

一般地，如果只有一个属性设置为auto，就会修改这个属性来满足本节前面给出的等式。给定以下样式，元素会延伸到必要的宽度，而不是“收缩”内容：

```
<span style="position: absolute; top: 0; left: 1em; right: 1em; width: auto;
  margin-left: 1em; margin-right: 1em; background: silver;">shrink-wrapped</span>
```

到目前为止，我们实际上只考虑了水平轴的行为，不过对于垂直轴，规则非常类似。还是看前面的讨论，只要将其旋转90度就会得到几乎相同的行为。例如，以下标记会得到如图10-48所示的结果：

```
<div style="position: relative; width: 30em; height: 10em;
  border: 1px solid;">
<div style="position: absolute; left: 0; width: 30%; background: #CCC;
  top: 0;">
element A
</div>
<div style="position: absolute; left: 35%; width: 30%; background: #AAA;
  top: 0; height: 50%;">
element B
</div>
<div style="position: absolute; left: 70%; width: 30%; background: #CCC;
  height: 50%; bottom: 0;">
element C
</div>
</div>
```

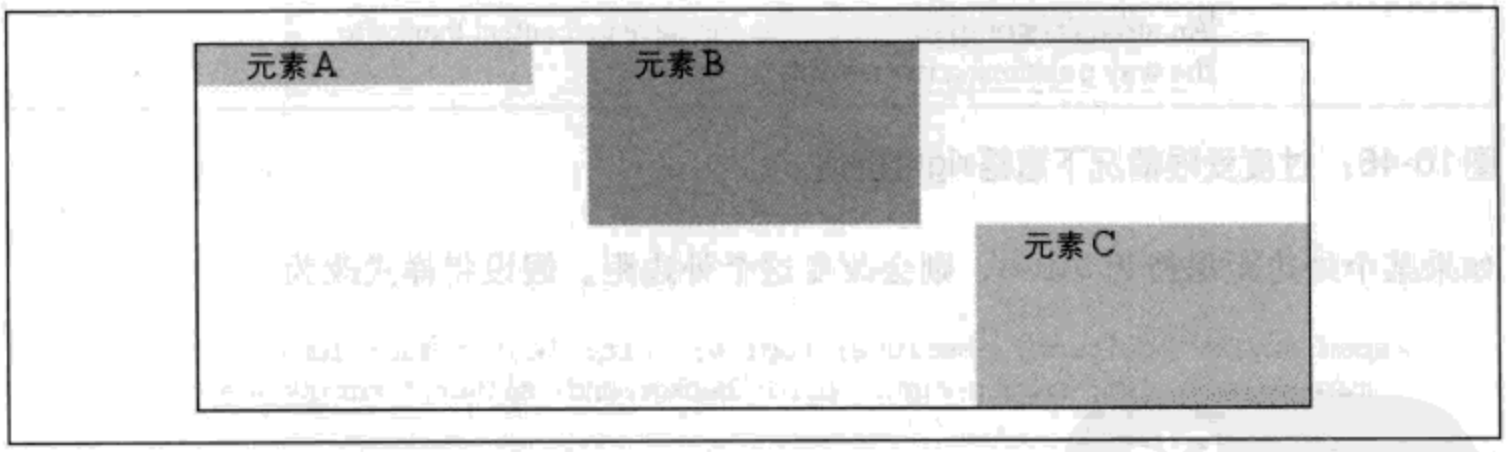

图10-48：绝对定位元素的垂直布局行为

在第一个例子中，元素的高度收缩为内容的高度。在第二个例子中，未指定的属性(bottom)设置为适当的值，来弥补定位元素底端与其包含块底端之间的距离。在第三个例子中，未指定的属性是top，所以由top来弥补定位元素顶端与其包含块顶端之间的距离。

对此，外边距为auto可以得到垂直居中的效果。给定以下样式，绝对定位元素div将在其包含块中垂直居中，如图10-49所示：

```
<div style="position: relative; width: 10em; height: 10em;
  border: 1px solid;">
<div style="position: absolute; left: 0; width: 100%; background: #CCC;
  top: 0; height: 5em; bottom: 0; margin: auto 0;">
```

```
element D
</div>
</div>
```

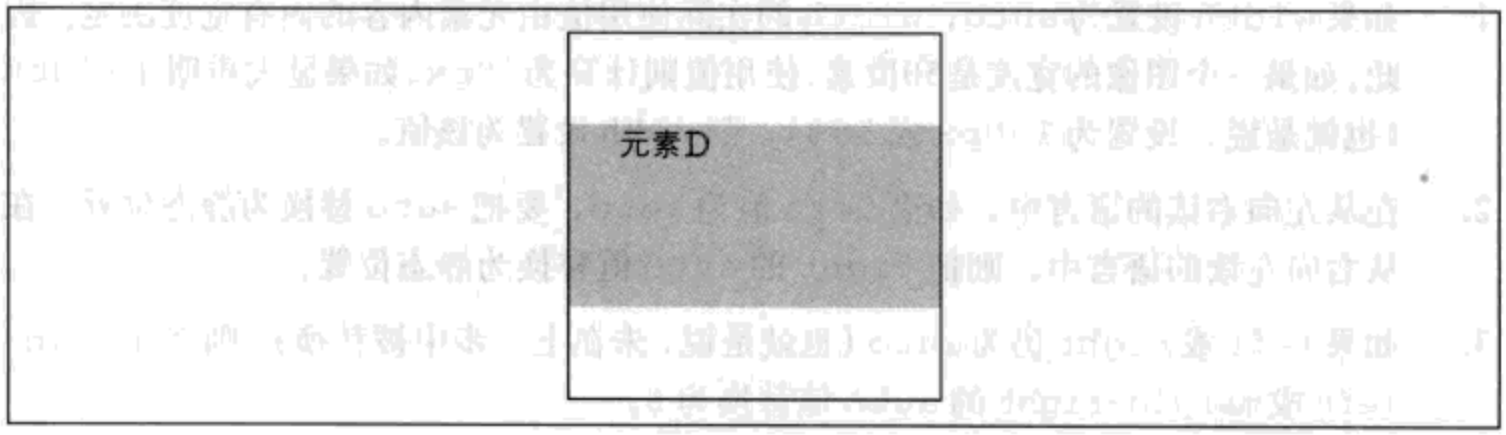

图10-49：将外边距为auto的绝对定位元素垂直居中

还要指出两个小变化。在水平布局中，如果值设置为auto，right或left都可以根据其静态位置放置。但在垂直布局中，只有top可以取静态位置，出于某种原因，bottom做不到。

注意： 写作本书时，Internet Explorer的所有版本（包括IE7）都不支持绝对定位元素通过将上下外边距设置为auto来实现垂直居中的行为。

另外，如果一个绝对定位元素的大小在垂直方向上过度受限，将忽略bottom。因此，对于以下情况，bottom的声明值会被计算值5em覆盖：

```
<div style="position: relative; width: 10em; height: 10em;
  border: 1px solid;">
<div style="position: absolute; left: 0; width: 100%; background: #CCC;
  top: 0; height: 5em; bottom: 0; margin: 0;">
element D
</div>
</div>
```

如果属性过度受限，没有规定将top忽略。

替换元素的放置和大小

非替换元素和替换元素的定位规则大不相同。这是因为替换元素有固有的高度和宽度，因此其大小不会改变，除非创作人员有意显式地修改。因此，在替换元素定位中没有“恰当收放”行为的概念。

确定替换元素位置和大小时，所涉及的行为用以下规则描述最为容易，这组规则按顺序分别：

1. 如果width设置为auto，width的实际使用值由元素内容的固有宽度决定。因此，如果一个图像的宽度是50像素，使用值则计算为50px。如果显式声明了width（也就是说，设置为100px或50%），则width设置为该值。
2. 在从左向右读的语言中，如果left值为auto，要把auto替换为静态位置。在从右向左读的语言中，则把right的auto值替换为静态位置。
3. 如果left或right仍为auto（也就是说，未在上一步中被替换），则将margin-left或margin-right的auto值替换为0。
4. 如果此时margin-left和margin-right都还定义为auto，则把它们设置为相等的值，从而将元素在其包含块中居中。
5. 在此之后，如果只剩下一个auto值，则将其修改为等于等式的余下部分（使等式满足）。

这与非替换元素绝对定位时的基本行为相同（只要假设非替换元素有一个显式的宽度）。因此，下面两个元素会有相同的宽度和位置，假设图像的固有宽度是100像素（见图10-50）：

```
<div style="width: 200px; height: 50px; border: 1px dotted gray;">
 <img src="frown.gif" alt="a frowny face"
  style="position: absolute; top: 0; left: 50px; margin: 0;">
 <div style="position: absolute; top: 0; left: 50px;
   width: 100px; height: 100px; margin: 0;">
   it's a div!
 </div>
</div>
```

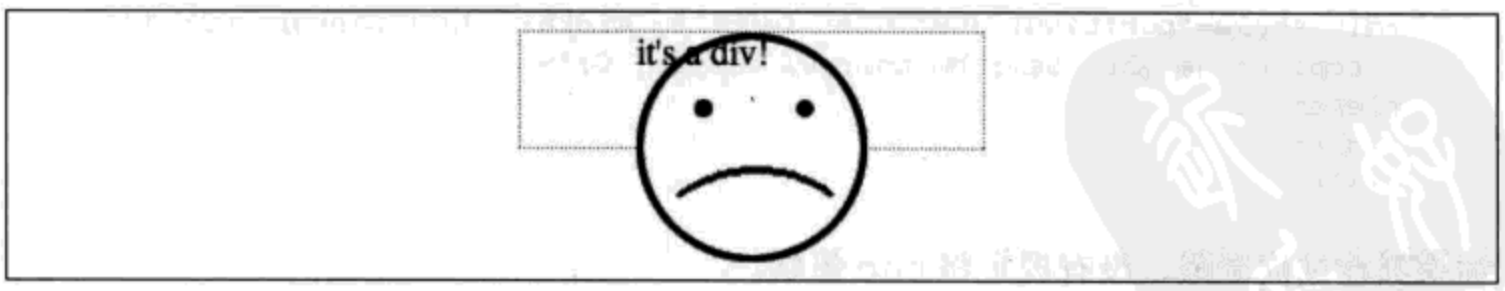

图10-50：绝对定位一个替换元素

与非替换元素一样，如果值过度受限，用户代理就会忽略right的值（对于从左向右读的语言），而在从右向左读的语言中会忽略left的值。因此，在以下例子中，right的声明值会被计算值50px覆盖：

```
<div style="position: relative; width: 300px;">
 <img src="frown.gif" alt="a frowny face"
```

```
    style="position: absolute; top: 0; left: 50px; right: 125px; width: 200px;
      margin: 0;">
</div>
```

类似地，沿垂直轴的布局受以下一组规则控制：

1. 如果height设置为auto，height的计算值由元素内容的固有高度确定。因此，对于一个50像素高的图像，其height计算为50px。如果height显式声明为某个值（如100px或50%），则height会设置为该值。
2. 如果top的值为auto，将其替换为替换元素的静态位置。
3. 如果bottom的值为auto，将margin-top或margin-bottom的所有auto值替换为0。
4. 如果此时margin-top和margin-bottom都还定义为auto，将其设置为相等的值，从而使元素在其包含块中居中。
5. 在此之后，如果仅剩下一个auto值，则将其修改为等于等式中的余下部分（使等式满足）。

与非替换元素一样，如果值过度受限，用户代理会忽略bottom。

因此，以下标记会得到如图10-51所示的结果：

```
<div style="position: relative; height: 200px; width: 200px; border: 1px solid;">
<img src="one.gif" alt="one" width="25" height="25"
  style="position: absolute;
  top: 0; left: 0; margin: 0;">
<img src="two.gif" alt="two" width="25" height="25"
  style="position: absolute;
   top: 0; left: 60px; margin: 10px 0; bottom: 4377px;">
<img src="three.gif" alt=" three" width="25" height="25"
  style="position: absolute;
   left: 0; width: 100px; margin: 10px; bottom: 0;">
<img src="four.gif" alt=" four" width="25" height="25"
  style="position: absolute;
   top: 0; height: 100px; right: 0; width: 50px;">
<img src="five.gif" alt="five" width="25" height="25"
  style="position: absolute;
   top: 0; left: 0; bottom: 0; right: 0; margin: auto;">
</div>
```

Z轴上的放置

对于所有定位，最后都不免遇到这样一种情况：两个元素试图放在同一个位置上。显然，其中一个必须盖住另一个——不过，如何控制哪个元素放在“上层”呢？这就引入了属性z-index。

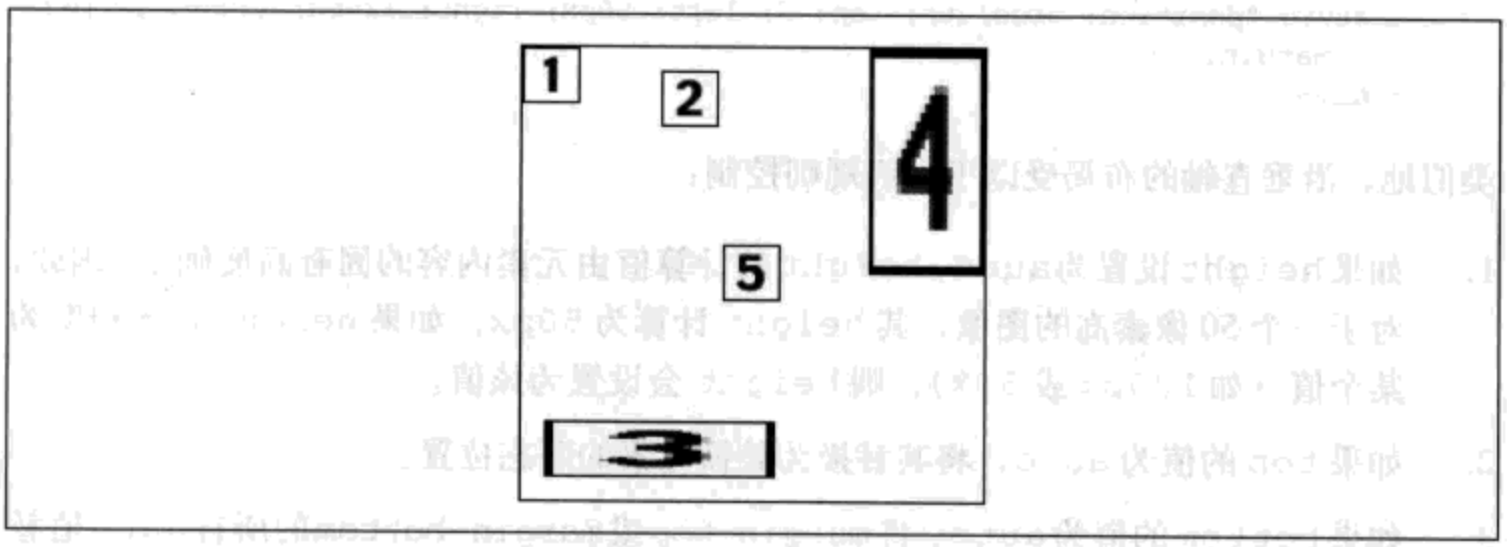

图10-51：通过定位拉伸替换元素

z-index	
值：	<integer> | auto | inherit
初始值：	auto
应用于：	定位元素
继承性：	无
计算值：	根据指定确定

利用z-index，可以改变元素相互覆盖的顺序。这个属性的名字由坐标系统得来，其中从左向右是x轴，从上到下是y轴。在这种情况下，第三个轴——即从前向后，或者如果你愿意，也可以理解为越来越远离用户——则称为z轴。因此，使用z-index为元素指定沿z轴的值，并相应表示。图10-52描述了这个坐标系。

在这个坐标系中，有较高z-index值的元素比z-index值较低的元素离读者更近。这会导致有较高z-index值的元素覆盖其他元素，如图10-53所示，这是图10-52的正面图。这也称为叠放（stacking）。

所有整数都可以用作为z-index的值，包括负数。如果为元素指定一个负z-index值，会将其移到到离读者更远的位置；也就是说，它会移到叠放栈的更低层。考虑以下样式，如图10-54所示：

```
p#first {position: absolute; top: 0; left: 0;
  width: 20%; height: 10em; z-index: 8;}
p#second {position: absolute; top: 0; left: 10%;
  width: 30%; height: 5em; z-index: 4;}
```

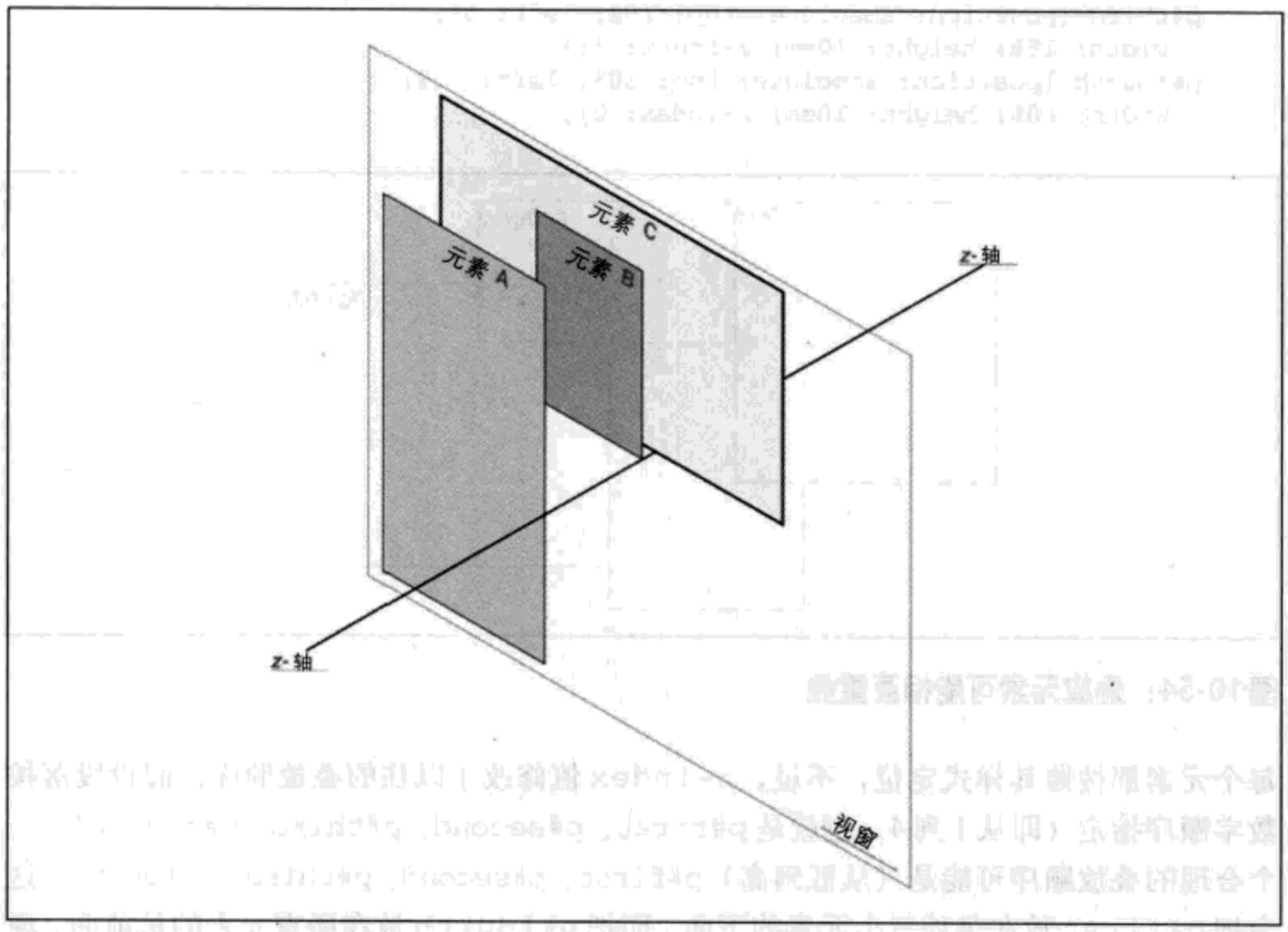

图10-52：z-index叠放的概念视图

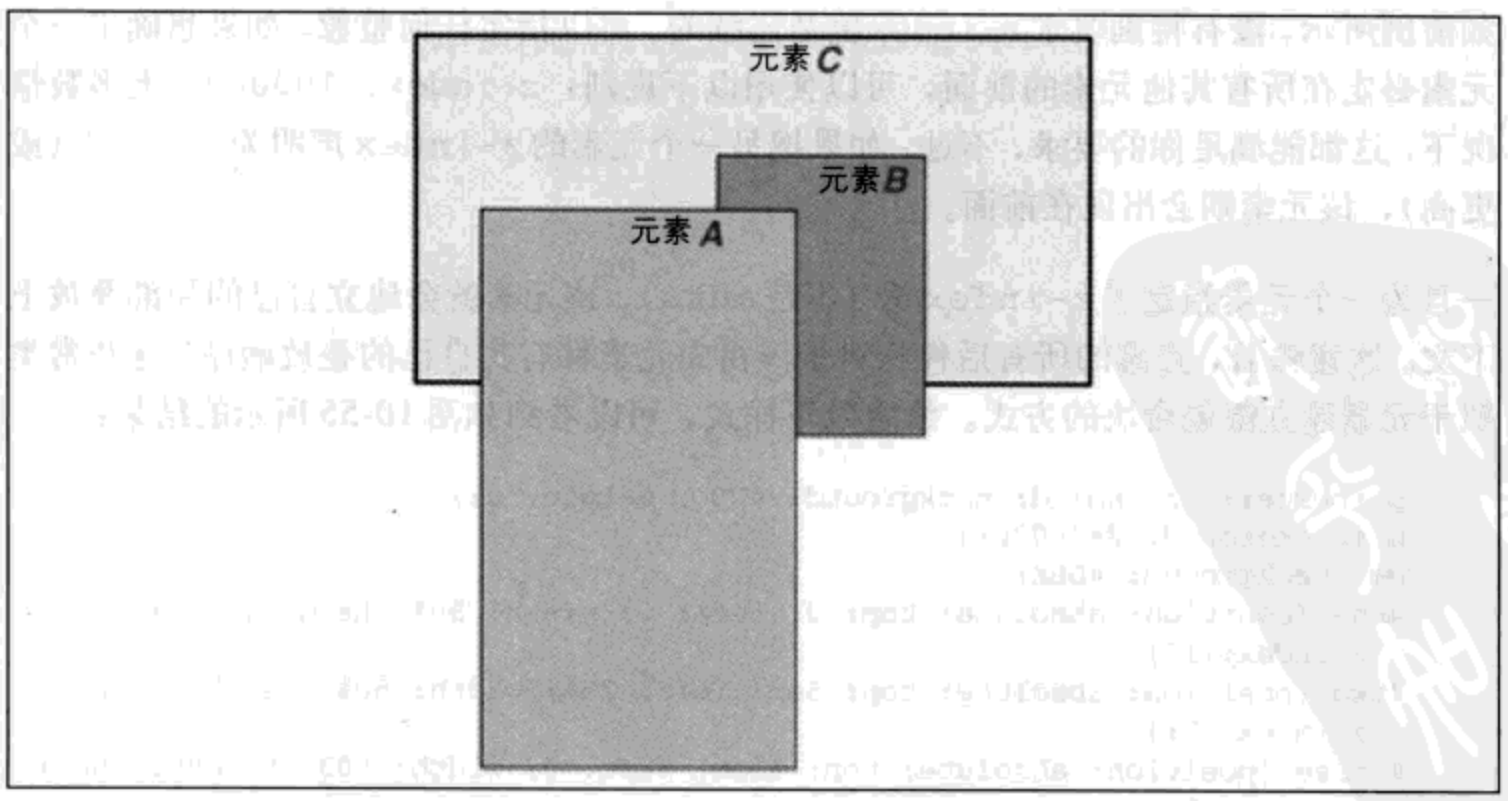

图10-53：元素如何叠放

```
p#third {position: absolute; top: 15%; left: 5%;
  width: 15%; height: 10em; z-index: 1;}
p#fourth {position: absolute; top: 10%; left: 15%;
  width: 40%; height: 10em; z-index: 0;}
```

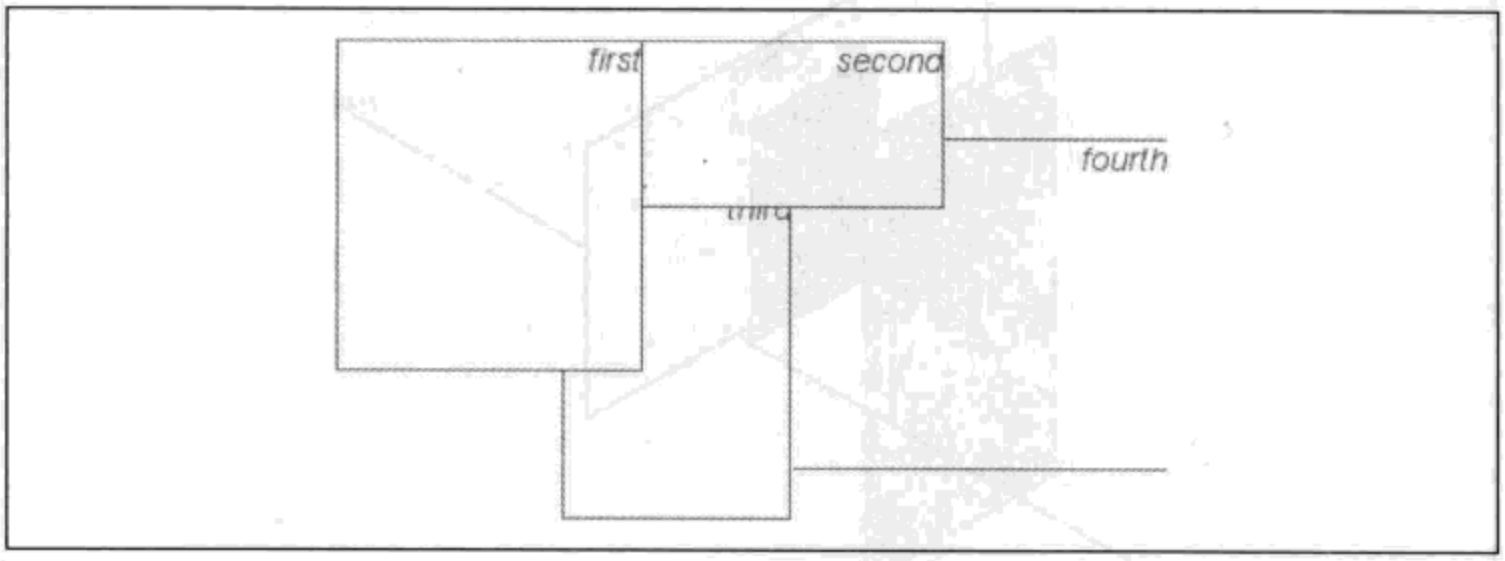

图10-54：叠放元素可能相互重叠

每个元素都按照其样式定位，不过，z-index值修改了以往的叠放顺序。假设段落按数字顺序指定（即从1到4，也就是p#first、p#second、p#third、p#fourth），一个合理的叠放顺序可能是（从低到高）p#first、p#second、p#third、p#fourth。这会把p#first放在另外三个元素的下面，而把p#fourth放在所有元素的最前面。现在，因为有z-index属性，叠放顺序则在你的控制之下。

如前例所示，没有特别要求z-index值是连续的。可以指定任何整数。如果想确定一个元素必定在所有其他元素的前面，可以使用以下规则：z-index: 100000。大多数情况下，这都能满足你的要求，不过，如果把另一个元素的z-index声明为100001（或更高），该元素则会出现在前面。

一旦为一个元素指定了z-index值（不是auto），该元素就会建立自己的局部叠放上下文。这意味着，元素的所有后代相对于该祖先元素都有其自己的叠放顺序。这非常类似于元素建立新包含块的方式。给定以下样式，可以看到如图10-55所示的结果：

```
p {border: 1px solid; background: #DDD; margin: 0;}
b {background: #808080;}
em {background: #BBB;}
#one {position: absolute; top: 0; left: 0; width: 50%; height: 10em;
  z-index: 10;}
#two {position: absolute; top: 5em; left: 25%; width: 50%; height: 10em;
  z-index: 7;}
#three {position: absolute; top: 11em; left: 0; width: 50%; height: 10em;
  z-index: 1;}
#one b {position: absolute; right: -5em; top: 4em; width: 20em;
  z-index: -404;}
```

```
#two b {position: absolute; right: -3em; top: auto;
  z-index: 36;}
#two em {position: absolute; bottom: -0.75em; left: 7em; right: -2em;
  z-index: -42;}
#three b {position: absolute; left: 3em; top: 3.5em; width: 25em;
  z-index: 23;}
```

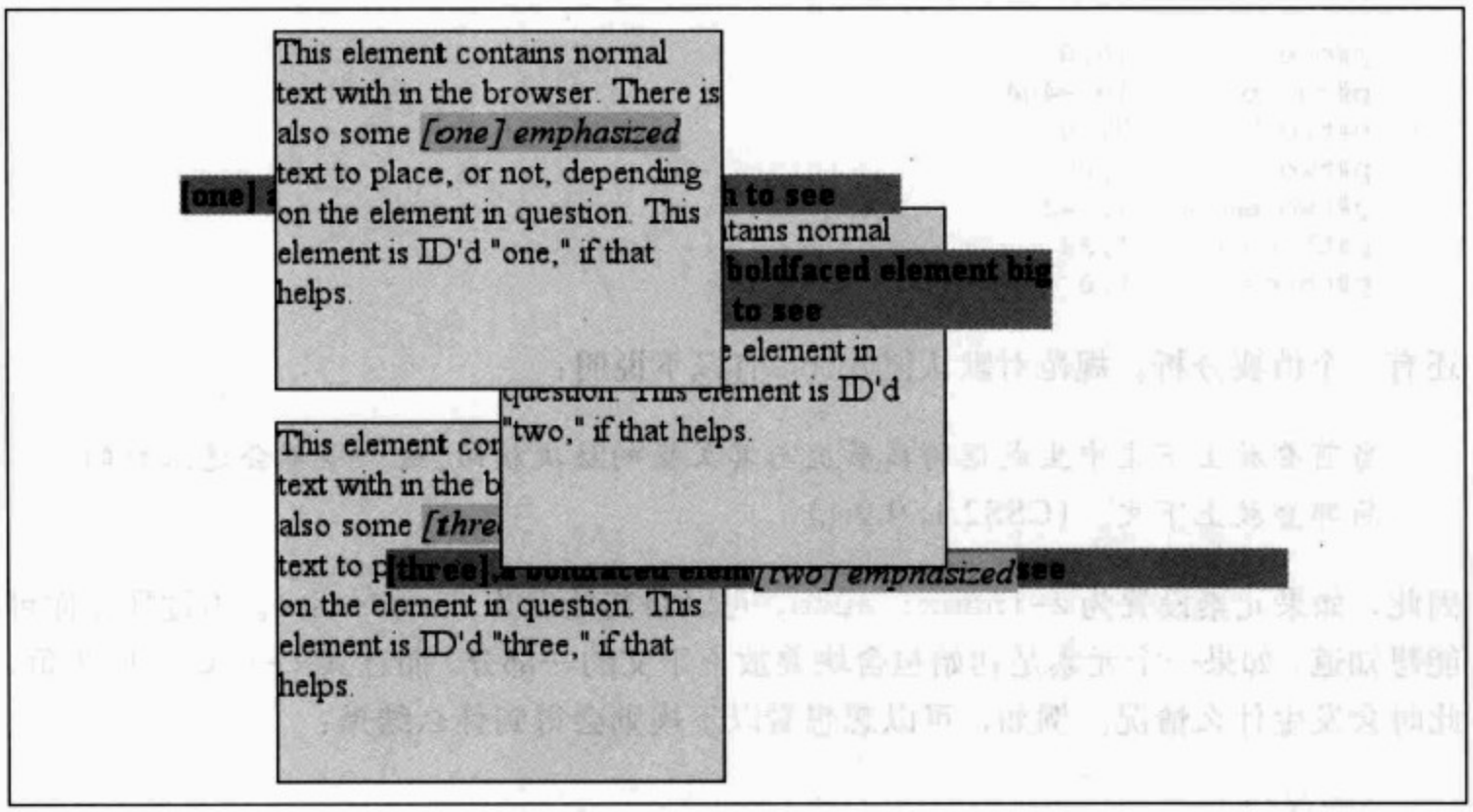

图10-55：定位元素建立局部叠放上下文

注意b和em元素在叠放顺序中的位置。当然，这些元素都相对于其父元素准确定位。不过，要特别注意p#two的子元素。尽管b元素在其父元素前面，而em在后面，但它们都在p#three的前面！这是因为z-index值36和-42都相对于p#two，而不是相对于总的文档。从某种意义上讲，p#two及其所有子元素都有共同的z-index值7，而在这个p#two上下文中各元素又有其自己的“次级”z-index。

换句话说，就好像b元素的z-index为7,36，而em的z-index值为7,-42。这些只是反映概念值；规范中没有相应的规定。不过，这样的系统有助于说明如何确定总的叠放顺序。请考虑：

```
p#one          10
p#one b        10,-404
p#two b        7,36
p#two          7
p#two em       7,-42
p#three b      1,23
p#three        1
```

这个概念框架准确地描述了这些元素以何种顺序叠放。从这个叠放顺序来看，尽管一个元素的后代可能在该元素的上面或下面，但它们与其祖先元素都归为一组。

如果一个元素为其后代元素建立了叠放上下文，而且该元素位于此上下文z轴的0位置上，也是如此。因此，可以将框架扩展如下：

```
p#one          10,0
p#one b        10,-404
p#two b        7,36
p#two          7,0
p#two em       7,-42
p#three b      1,23
p#three        1,0
```

还有一个值要分析。规范对默认值auto有以下说明：

> 当前叠放上下文中生成框的栈层次与其父框的层次相同。这个框不会建立新的局部叠放上下文。(CSS2.1: 9.9.1)

因此，如果元素设置为z-index: auto，可以将其处理为z-index: 0。不过现在你可能想知道，如果一个元素是初始包含块叠放上下文的一部分，而且其z-index值为负，此时会发生什么情况。例如，可以想想看以下规则会得到什么结果：

```
<body>
 <p style="position: absolute; z-index: -1;">Where am I?</p>
</body>
```

根据叠放规则，body元素与其父元素框在同一个叠放上下文中，因此将是0。它不会建立一个新的叠放上下文，所以绝对定位的p元素与body元素置于相同的叠放上下文中(即初始包含块的叠放上下文)。换句话说，段落放在body元素的后面（被body元素盖住)。如果body有不透明的背景，这个段落会消失。

不过，只是CSS2中会如此。在CSS2.1中，叠放规则有所改变，要求元素绝对不会叠放在其叠放上下文的背景之下。换句话说，考虑以下情况，body元素为其后代建立了一个包含块（例如，假设它是相对定位元素)。作为body元素后代的一个绝对定位元素就不能叠放在body的背景之下，不过可以叠放在body的内容下面。

写作本书时，即使将body和html元素都设置为有透明的背景，Mozilla和相关浏览器也会完全隐藏段落。这种做法是错误的。即使body有背景，其他用户代理（如Internet Explorer）也会把段落放在body的背景之上。根据CSS2.1，这才是正确的行为。由于用户代理的做法不同，所以z-index值为负时会导致不可预料的后果，所以使用要小心。

固定定位

从上一节可以看出，固定定位与绝对定位很类似，只不过固定元素的包含块是视窗。固定定位时，元素会完全从文档流中去除，不会有相对于文档中任何部分的位置。

可以采用一些有意思的方式充分利用固定定位。首先，可以使用固定定位创建帧式界面。考虑图 10-56，这里显示了一个相当常见的布局机制。

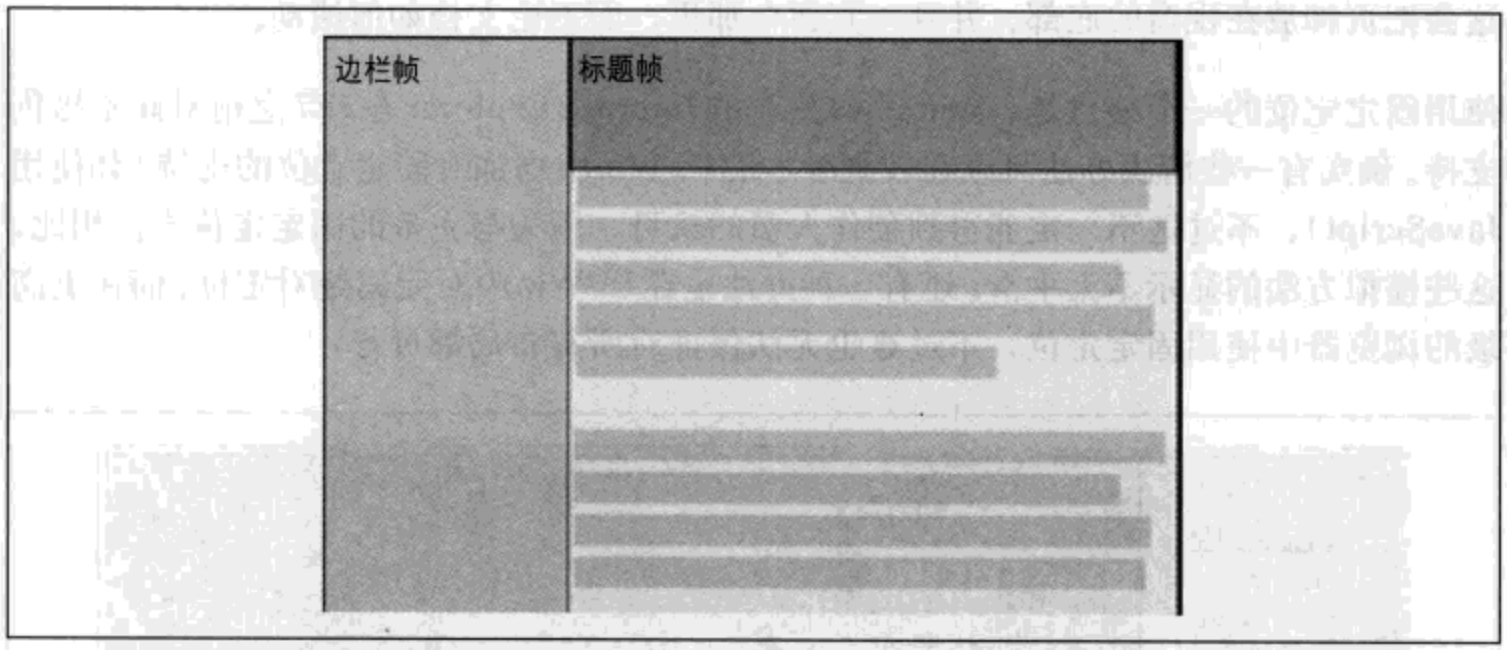

图10-56：使用固定定位模拟帧

这可以使用以下样式得到：

```
div#header {position: fixed; top: 0; bottom: 80%; left: 20%; right: 0;
  background: gray;}
div#sidebar {position: fixed; top: 0; bottom: 0; left: 0; right: 80%;
  background: silver;}
```

以上样式会把标题和边栏固定到视窗的顶部和旁边，并保持不动，而不论文档如何滚动。不过这样做的缺点是，文档的其余部分会被固定元素覆盖。因此，可能应当把其余内容包含在其自己的 div 中，并应用以下规则：

```
div#main {position: absolute; top: 20%; bottom: 0; left: 20%; right: 0;
  overflow: scroll; background: white;}
```

甚至可以适当地增加外边距，在三个定位 div 之间建立一些小缝隙，如图 10-57 所示：

```
body {background: black; color: silver;}  /* colors for safety's sake */
div#header {position: fixed; top: 0; bottom: 80%; left: 20%; right: 0;
  background: gray; margin-bottom: 2px; color: yellow;}
div#sidebar {position: fixed; top: 0; bottom: 0; left: 0; right: 80%;
  background: silver; margin-right: 2px; color: maroon;}
div#main {position: absolute; top: 20%; bottom: 0; left: 20%; right: 0;
  overflow: auto; background: white; color: black;}
```

对于这种情况，可以向body背景应用一个平铺图像。这个图像会透过外边距创建的缝隙显示出来，如果创作人员认为合适，还可以使缝隙加宽。

固定定位的另一个用途是在屏幕上放置一个“永久性”元素，如一个小的链接列表。还可以创建一个包含版权和其他信息的永久性页脚，如下所示：

```
div#footer {position: fixed; bottom: 0; width: 100%; height: auto;}
```

这会把页脚放在视窗的底部，并且一直留在那里，而不论文档如何滚动。

使用固定定位的一个缺点是，Windows平台的Internet Explorer在IE7之前对此不提供支持。确实有一些解决办法可以在较老版本的IE/Win中增加对固定定位的支持（如使用JavaScript），不过这不一定能得到创作人员的认可，因为与完备的固定定位支持相比，这些模拟方法的显示不太平滑。还有一种办法是在IE/Win中对元素绝对定位，而在更高级的浏览器中使用固定定位，不过这也无法保证对所有布局都可行。

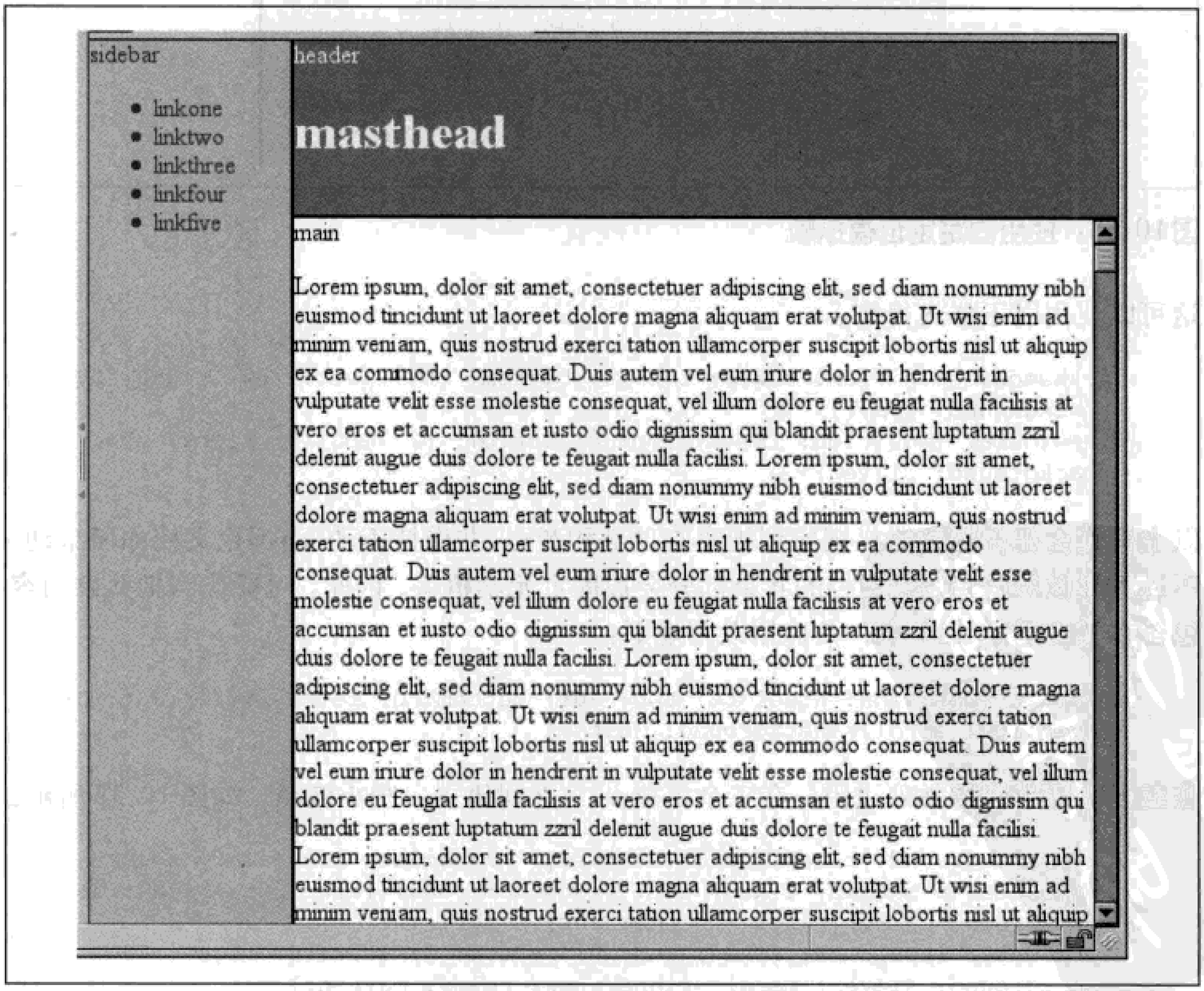

图10-57：利用外边距分隔“帧”

注意： 要了解如何在较老版本的IE/Win中模拟固定定位，还可以参考 *http://css-discuss.incutio.com/?page=EmulatingFixedPositoning*（注1）。

相对定位

理解起来最简单的定位机制就是相对定位。采用这种机制时，将通过使用偏移属性移动定位元素。不过，这可能有一些有意思的后果。

从表面看来，似乎这就足够了。假设希望将一个图像向上向左移动。图10-58显示了以下样式的结果：

```
img {position: relative; top: -20px; left: -20px;}
```

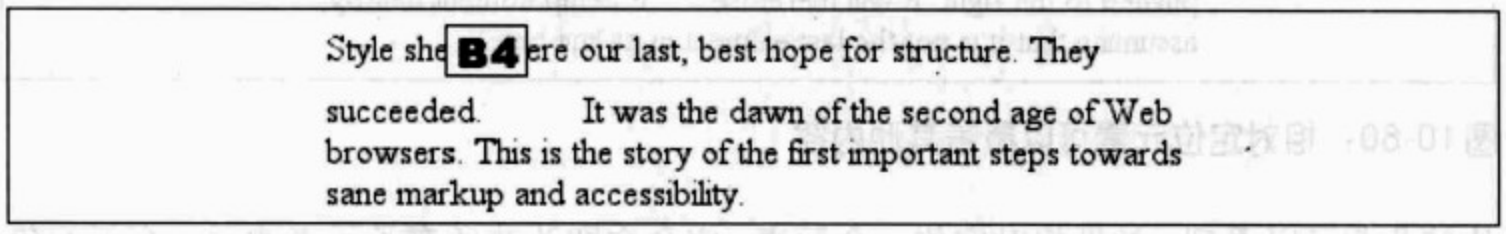

图10-58：相对定位元素

这里所做的只是将图像的上边界向上偏移20像素，左边界向左偏移20像素。不过，注意这里的空白，如果该图像未定位，它本该放在这里。之所以会发生这种情况，原因是当元素相对定位时，它会从其正常位置移走，不过，原来所占的空间并不会因此消失。考虑以下样式的结果，如图10-59所示：

```
em {position: relative; top: 8em; color: gray;}
```

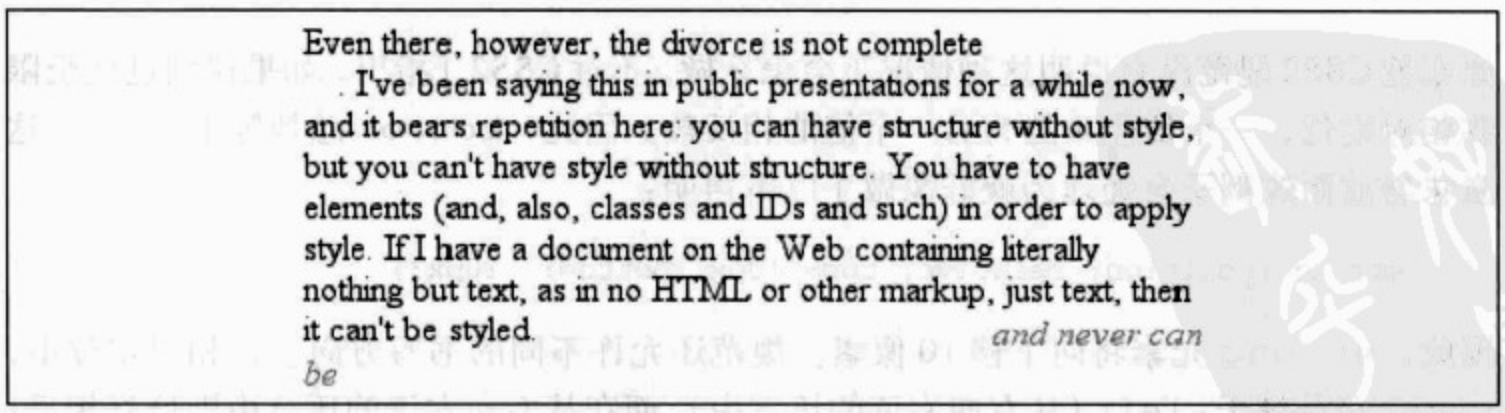

图10-59：相对定位元素

注1： 这里确实把“positioning”错拼成了“Positoning”，不过这只是在英语拼法上有问题，这个页面本身还是提供了不错的信息。

可以看到，这个段落里有一些空白。本来这是em元素的位置，而em元素的新位置正好在这个空白的后面。

当然，还可以移动一个相对定位元素，让它覆盖其他内容。例如，以下样式和标记的结果如图10-60所示：

```
img.slide {position: relative; left: 30px;}
<p>
In this paragraph, we will find that there is an image that has been pushed to
the right.  It will therefore <img src="star.gif" alt="A star!" class="slide">
overlap content nearby, assuming that it is not the last element in its line box.
</p>
```

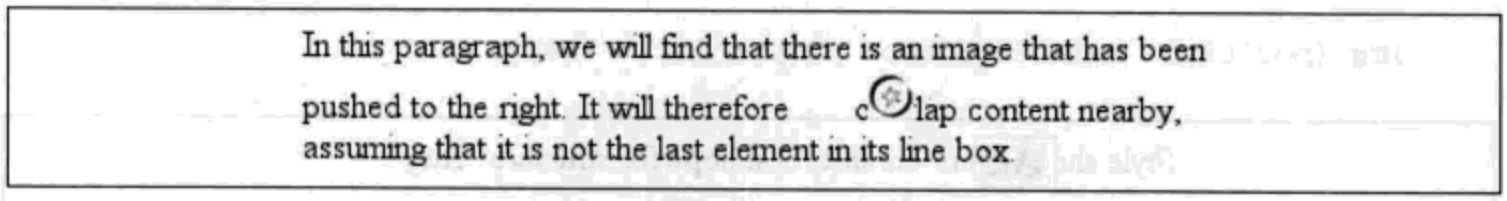

图10-60：相对定位元素可以覆盖其他内容

从前几节可以看到，如果相对定位一个元素，它会立即为其所有子元素建立一个新的包含块。这个包含块对应于该元素原本所在的位置。

相对定位有一个有意思的小问题。如果一个相对定位元素过度受限会发生什么情况呢？例如：

```
strong {position: relative; top: 10px; bottom: 20px;}
```

这里指定的值要求两种完全不同的行为。如果只考虑top: 10px，则元素应当向下移10像素，但bottom: 20px则要求元素应该向上移20像素。

原来的CSS2规范没有说明这种情况下会怎么做。不过CSS2.1指出，如果遇到过度受限的相对定位，一个值会重置为另一个值的相反数。因此，bottom总是等于-top。这意味着前面的例子会处理为就好像做了以下声明：

```
strong {position: relative; top: 10px; bottom: -10px;}
```

因此，strong元素将向下移10像素。规范还允许不同的书写方向。在相对定位中，right总是等于-left(从左向右读的语言中)，而在从右向左读的语言中则恰好相反：left总是等于-right。

小结

浮动和定位是CSS的两个很吸引人的特性。不过，如果使用时不小心，也会让人很迷惑。对元素定位时，元素重叠、叠放顺序、大小和放置等都需要仔细考虑，另外还必须考虑浮动元素与正常流的关系。因此，使用浮动和定位创建布局需要有所顾忌，不过还是利大于弊。

利用这些特性，很多布局中确实已经不需要使用表了，不过出于某些原因，Web中还是要使用表，如表示股票行情和运动成绩等。下一章我们将介绍CSS做了哪些改进来处理表布局的问题。

第 11 章

表布局

看到本章的标题，你可能会奇怪，“表布局？这不是我们一直都极力避免的吗？”确实如此，不过本章并非讨论如何使用表来建立布局，而是要介绍CSS中表本身如何布局，这个问题乍看起来可能很简单，但实际上要复杂得多。正因如此，我们专门用一章来介绍。

与文档布局的其他方面相比，表很特别。在CSS2.1中，表本身就能够确定其他元素的元素大小，例如，可以让一行中的所有单元格都有相同的高度，而不论每个单元格中可能包含多少内容。还可以让一列中的单元格都有相同的宽度。建立文档布局时，没有哪种情况能够以这样一种直接的方式让文档树中不同部分的元素相互影响大小和布局。

可以看到，这种特殊性使表具有很多特殊的行为和规则，而且这些行为和规则是表所独有的。本章中我们将介绍表在视觉上如何组装，另外会介绍绘制单元格边框的两种方法，还将介绍确定表及其内部元素高度和宽度的机制。

表格式化

你可能在考虑如何绘制单元格边框以及如何确定表的大小，不过在此之前，我们先来学习组装表的基本方法，并了解表中的元素相互之间有什么关系。这称为表格式化（table formatting），它与表布局有很大不同：只有介绍完表格式化，才有可能介绍表布局。

表的视觉编排

首先要了解CSS如何定义表的编排。尽管这看上去很基本，不过对于理解如何最佳地设置表样式至关重要。

CSS对于表元素和内部表元素有很分明的界限。在CSS中，内部表元素生成矩形框，这些框有内容、内边距和边框，但是没有外边距。因此，不能通过指定外边距来定义单元格之间的间隔。如果试图对单元格、行或任何其他内部表元素应用外边距，CSS兼容浏览器只会将其忽略（只有总标题例外，此内容将在本章后面讨论）。

表的编排有6个规则。这些规则的基础是“表格单元”，这是绘制表的表格线之间的区域。请看图11-1所示，这里显示了两个表及其表格单元，由表上所画的虚线指示。

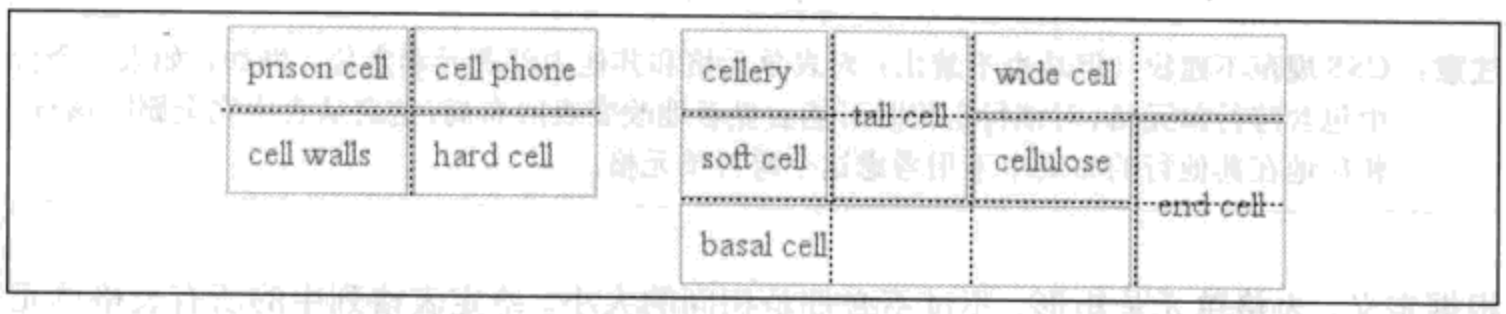

图11-1：表格单元构成了表布局的基础

在一个简单的2×2表中（如图11-1中左边的表）表格单元对应于单元格。在更复杂的表中（如图11-1中右边的表），表格单元的边界对应于表中所有单元格的单元格边框，并穿过跨多行或多列的单元格。

这些表格单元很大程度上是理论性的构造，不能为它们设置样式，甚至不能通过文档对象模型来访问。这些构造只是用于描述如何组装表来设置样式。

表编排规则

- 每个行框包含一行表格单元。表中的所有行框按其在源文档中出现的顺序从上到下地填充表（表的标题行框和脚注行框有所例外，它们分别出现在表的最前面和最后面）。因此，有多少个行元素，表中就包含多少表格行。
- 一个行组包含多少个行框，该行组框就包含多少个表格单元。
- 列框包含一列或多列表格单元。所有列框都按其出现的顺序依次相邻放置。对于从左向右读的语言，第一个列框放在左边，而对于从右向左读的语言，第一个列框则放在右边。
- 列组中包含多少个列框，该列组框中就包含多少个表格单元。

- 尽管单元格可能跨多行或多列，不过CSS对此并没有做出定义，而是由文档语言来定义这种跨行或跨列。每个跨行或跨列的单元格是一个矩形框，其宽度和高度分别为一个或多个单元格。这个矩形框的顶行在作为该单元格父元素的行中。在从左向右读的语言中，该单元格的矩形框必须尽可能向左，不过不能覆盖任何其他单元格框。在从左向右读的语言中，这个单元格还必须在同一行上所有之前单元格（在源文档中较早出现的单元格）的右边。在从右向左读的语言中，跨行或跨列的单元格则必须尽可能向右，但不能覆盖其他单元格，而且要在同一行上所有之前单元格（源文档中在其之前出现的单元格）的左边。
- 单元格框不能超出表或行组的最后一个行框。如果表结构可能造成这种情况，单元格则必须缩小，使之能放在包含它的表或行组中。

注意：CSS规范不建议（但是也不禁止）对表单元格和其他内部表元素定位。例如，如果一个行中包含跨行单元格，对该行定位时可能会显著地改变表的布局，这会从表中完全删除该行，相应地在其他行的布局中不用考虑这个跨行单元格。

根据定义，表格单元是矩形，不过不必都是相同的大小。给定表格列中的所有表格单元宽度相等，一个表格行中的所有表格单元则高度相等，不过，一个表格行的高度可能与另一个表格行的高度不同。类似地，表格列也可能有不同的宽度。

根据这些基本规则，你可能会问：那么，怎么知道哪些元素是单元格而哪些不是呢？下一节就来回答这个问题。

表显示值

在HTML中，很容易知道哪些元素属于表，因为像`tr`和`td`之类元素的处理已经内置在浏览器中。与此不同，在XML中则没有办法从根本上知道哪些元素可能是表的一部分。因此引入了一组`display`值。

这一章中，我们只讨论与表有关的值，因为其他值（`block`、`inline`、`inline-block`、`run-in`和`list-item`）已经在其他章节讨论过。与表相关的值可以总结如下：

`table`

这个值指定一个元素定义了一个块级表。因此，它定义了一个生成块框的矩形块。相应的HTML元素当然是`table`。

display	
值：	none | inline | block | inline-block | list-item | run-in | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | inherit
初始值：	inline
应用于：	所有元素
继承性：	无
计算值：	对于浮动、定位和根元素，计算值可变（见CSS2.1第9.7节）；否则，根据指定确定
说明：	CSS2中还有值compact和marker，不过由于缺乏广泛的支持，在CSS2.1中已经去掉

inline-table

这个值指定一个元素定义了一个行内级表。这说明，该元素定义了一个生成行内框的矩形块。与之最接近的非表值是inline-block。最接近的HTML元素为table，不过，默认情况下HTML表不是行内元素。

table-row

这个值指定一个元素是一个单元格的行。相应的HTML元素是tr元素。

table-row-group

这个值指定一个元素是一个或多个行的组。相应的HTML值是tbody。

table-header-group

这个值与table-row-group非常相似，只是视觉格式化方面有所不同，标题行组总是在所有其他行和行组之前显示（如果最上面有总标题，要在总标题之后显示）。打印时，如果一个表需要多页打印，用户代理可以在各页顶端重复标题行。规范没有定义如果为多个元素指定table-header-group值会发生什么情况。标题组可以包含多个行。与之对应的HTML元素是thead。

table-footer-group

这个值与table-header-group很类似，不过脚注行组总是在所有其他行之后显示，如果最下面有页脚标题，要在该总标题之前显示。打印时，如果一个表需要多页打印，用户代理可以在各页底端重复页脚行。规范没有定义如果为多个元素指定table-footer-group值会有什么结果。与之对应的HTML元素是tfoot。

table-column

这个值声明元素描述了一个单元格的列。按CSS的术语来说，有这个display值的元素并不显示，就好像它的display值为none一样。之所以存在这个值，主要是为了帮助定义列中单元格的表示。相应的HTML元素是col元素。

table-column-group

这个值声明一个元素是一个或多个列的组。类似于table-column元素，table-column-group元素也不显示，不过这个值有助于定义列组中元素的表示。相应的HTML元素是colgroup元素。

table-cell

这个值指定一个元素表示表中的单个单元格。HTML元素th和td都属于table-cell元素。

table-caption

这个值定义一个表的总标题。CSS没有定义如果多个元素的display值都为caption时会发生什么情况，不过CSS确实明确地警告，“……创作人员不要在一个表或行内表元素中放多个有display: caption的元素。”

下面从附录C给出的示例HTML 4.0样式表中节取一部分，对这些值的一般效果做了一个简短的总结：

```
table          {display: table;}
tr             {display: table-row;}
thead          {display: table-header-group;}
tbody          {display: table-row-group;}
tfoot          {display: table-footer-group;}
col            {display: table-column;}
colgroup       {display: table-column-group;}
td, th         {display: table-cell;}
caption        {display: table-caption;}
```

在XML中，默认地元素没有显示语义，所以这些值非常有用。考虑以下标记：

```
<scores>
 <headers>
   <label>Team</label>
   <label>Score</label>
 </headers>
 <game sport="MLB" league="NL">
   <team>
     <name>Reds</name>
     <score>8</score>
   </team>
   <team>
     <name>Cubs</name>
     <score>5</score>
```

```
    </team>
  </game>
</scores>
```

可以用以下样式将其格式化为一种表格形式：

```
scores {display: table;}
headers {display: table-header-group;}
game {display: table-row-group;}
team {display: table-row;}
label, name, score {display: table-cell;}
```

然后根据需要对各个单元格应用样式，例如，将label元素变为粗体，对scores右对齐。

注意： 尽管理论上可以为任何HTML元素指定与表有关的display值，不过在IE7之前，Internet Explorer并不支持这个功能。

以行为主

CSS将其表模型定义为“以行为主（row primacy）”。换句话说，这个模型假设创作人员创建的标记语言会显式声明行，而列是从单元格行的布局推导出来的。因此，第一列由各行中的第一个单元格组成，第二列则由各行中第二个单元格组成，依此类推。

以行为主在HTML中不算大问题，因为HTML本身就是面向行的。而在XML中则影响比较大，因为它限制了创作人员定义表标记的方式。由于CSS表模型本质是面向行的，倘若标记语言中表布局的基础是列，要想使用CSS来表示这种文档是无法做到的。

在本章余下部分，随着我们继续讨论表表示的详细内容，也可以清楚地看出CSS模型具有这种以行为主的性质。

列

尽管CSS表模型是面向行的，列在布局中仍有很重要的地位。虽然单元格在文档源中是行元素的后代，但它们可能同时属于两个上下文（行和列）。不过，在CSS中列和列组只能接受4种样式：border、background、width和visibility。

另外，这4个属性有一些只能应用于列上下文的特殊规则：

`border`

只有当border-collapse属性值为collapse时才能为列和列组设置边框。在这种情况下，列和列组边框会参与设置各单元格边界边框样式的合并算法（见本章后面“合并单元格边框”部分）。

background

只有当单元格及其行有透明背景时，列或列组的背景才可见（见本章后面的“表层”部分）。

width

width属性定义了列或列组的最小宽度。列（或列组）中单元格的内容可能要求列更宽。

visibility

如果一个列或列组的visibility为collapse，则该列（或列组）中所有单元格都不显示。从合并列跨到其他列的单元格会被剪裁，这类似于从其他列跨到隐藏列中的单元格。另外，表的总宽度会减去已合并列的宽度。如果对列或列组将visibility声明为任何非collapse值（译注1），则会被忽略。

匿名表对象

标记语言中可能未包含足够的元素，以至于无法按CSS的定义充分表示表，也可能创作人员没有加入所有必要的元素。例如，考虑以下HTML：

```
<table>
  <td>Name:</td>
  <td><input type="text"></td>
</table>
```

乍看到这个标记，你可能认为它定义了一个包含两个单元格的单行表，不过从结构上讲，这里没有定义行的元素（因为没有tr）。

考虑到这种可能性，CSS定义了一种机制，可以将“遗漏的”组件作为匿名对象插入。为了说明这是如何工作的，下面再来看前面少了行元素的XHTML示例。按CSS的术语来讲，实际上会在table元素和它的后代表单元格之间插入一个匿名table-row对象：

```
<table>
 [anonymous table-row object begins]
  <td>Name:</td>
  <td><input type="text"></td>
 [anonymous table-row object ends]
</table>
```

这个过程示意图如图11-2所示。

译注1： 除collapse外。

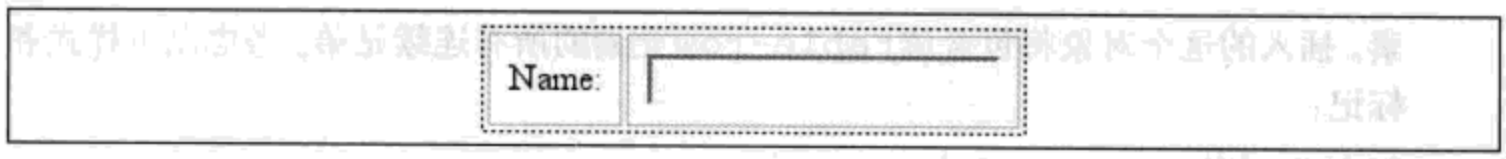

图11-2：表格式化中的匿名对象生成

CSS表模型中可能出现7种不同的匿名对象插入。类似于继承和特殊性，这7个规则作为一种机制，试图对CSS的表现方式赋予一种直观意义。

对象插入规则

1. 如果一个table-cell元素的父元素不是table-row元素，则会在该table-cell元素及其父元素之间插入一个匿名table-row对象。所插入的这个对象将包含该table-cell元素的所有连续兄弟。考虑以下样式和标记：

```
system {display: table;}
name, moons {display: table-cell;}

<system>
  <name>Mercury</name>
  <moons>0</moons>
</system>
```

在单元格元素（table-cell）和system元素之间插入了一个匿名table-row对象，它包含name和moons元素。

即使父元素是table-row-group，这个规则也同样成立。扩展这个例子，假设应用以下样式：

```
system {display: table;}
planet {display: table-row-group;}
name, moons {display: table-cell;}

<system>
  <planet>
  <name>Mercury</name>
  <moons>0</moons>
 </planet>
 <planet>
  <name>Venus</name>
  <moons>0</moons>
 </planet>
</system>
```

在这个例子中，两组单元格都包含在一个匿名table-row对象中，该对象插入到该组单元格和相应planet元素之间。

2. 如果一个table-row元素的父元素不是table、inline-table或table-row-group元素，则会在该table-row元素及其父元素之间插入一个匿名table元

素。插入的这个对象将包含该table-row元素的所有连续兄弟。考虑以下样式和标记：

```
docbody {display: block;}
planet {display: table-row;}

<docbody>
  <planet>
  <name>Mercury</name>
  <moons>0</moons>
 </planet>
  <planet>
  <name>Venus</name>
  <moons>0</moons>
 </planet>
</docbody>
```

由于planet元素的父元素的display值为block，所以会在planet元素和docbody元素之间插入匿名table对象。这个对象将包含这两个planet元素，因为它们是连续兄弟。

3. 如果一个table-column元素的父元素不是table、inline-table或table-column-group元素，则在该table-column元素及其父元素之间插入一个匿名table元素。这与前面讨论的table-row元素的相应规则很相似，只不过这里是面向列的。

4. 如果一个table-row-group、table-header-group、table-footer-group、table-column-group或table-caption元素的父元素不是table元素，则会在该元素及其父元素之间插入一个匿名table元素。

5. 如果一个table或inline-table元素的子元素不是table-row-group、table-header-group、table-footer-group、table-row或table-caption元素，则在该table元素与其子元素之间插入一个匿名table-row对象。这个匿名对象将包含该子元素的所有不是able-row-group、table-header-group、table-footer-group、table-row或table-caption元素的连续兄弟。考虑以下样式和标记：

```
system {display: table;}
planet {display: table-row;}
name, moons {display: table-cell;}

<system>
  <planet>
  <name>Mercury</name>
  <moons>0</moons>
 </planet>
  <name>Venus</name>
  <moons>0</moons>
</system>
```

这里，在system元素和第二组name和moons元素之间插入了一个匿名table-row对象。planet元素未包含在这个匿名对象中，因为其display值为table-row。

6. 如果一个table-row-group、table-header-group或table-footer-group元素的子元素不是table-row元素，则在该元素及其子元素之间插入一个匿名table-row对象。这个匿名对象包含该子元素的所有本身非table-row对象的连续兄弟。考虑以下样式和标记：

```
system {display: table;}
planet {display: table-row-group;}
name, moons {display: table-cell;}

<system>
  <planet>
  <name>Mercury</name>
  <moons>0</moons>
 </planet>
  <name>Venus</name>
  <moons>0</moons>
</system>
```

在这里，各组name和moons元素都包含在一个匿名table-row元素中。第二组name和moons元素与system之间之所以插入匿名table-row元素，是根据规则5完成的。对于第一组，匿名对象插入到planet元素和其子元素之间，因为planet元素是一个table-row-group元素。

7. 如果一个table-row元素的子元素不是table-cell元素，则在该元素和其子元素之间插入一个匿名table-cell对象。这个匿名对象包含该子元素的所有本身非table-cell元素的连续兄弟。考虑以下样式和标记：

```
system {display: table;}
planet {display: table-row;}
name, moons {display: table-cell;}

<system>
  <planet>
  <name>Mercury</name>
  <num>0</num>
 </planet>
</system>
```

由于元素num没有与表有关的display值，将在planet元素和num元素之间插入一个匿名table-cell对象。

这种行为还会扩展到匿名行内框。假设未包含num元素：

```
<system>
  <planet>
  <name>Mercury</name>
```

```
  0
 </planet>
</system>
```

0仍将包含在匿名table-cell对象中。为了进一步说明这一点，来看一个选自CSS规范的例子：

```
example {display: table-cell;}
row {display: table-row;}
hi {font-weight: 900;}

<example>
  <row>This is the <hi>top</hi> row.</row>
  <row>This is the <hi>bottom</hi> row.</row>
</example>
```

在每个row元素中，文本片段和hi元素都包含在一个匿名table-cell对象中。

表层

为了完成表的显示，CSS定义了6个不同的“层”，可以分别放表的不同方面。图11-3显示了这些层。

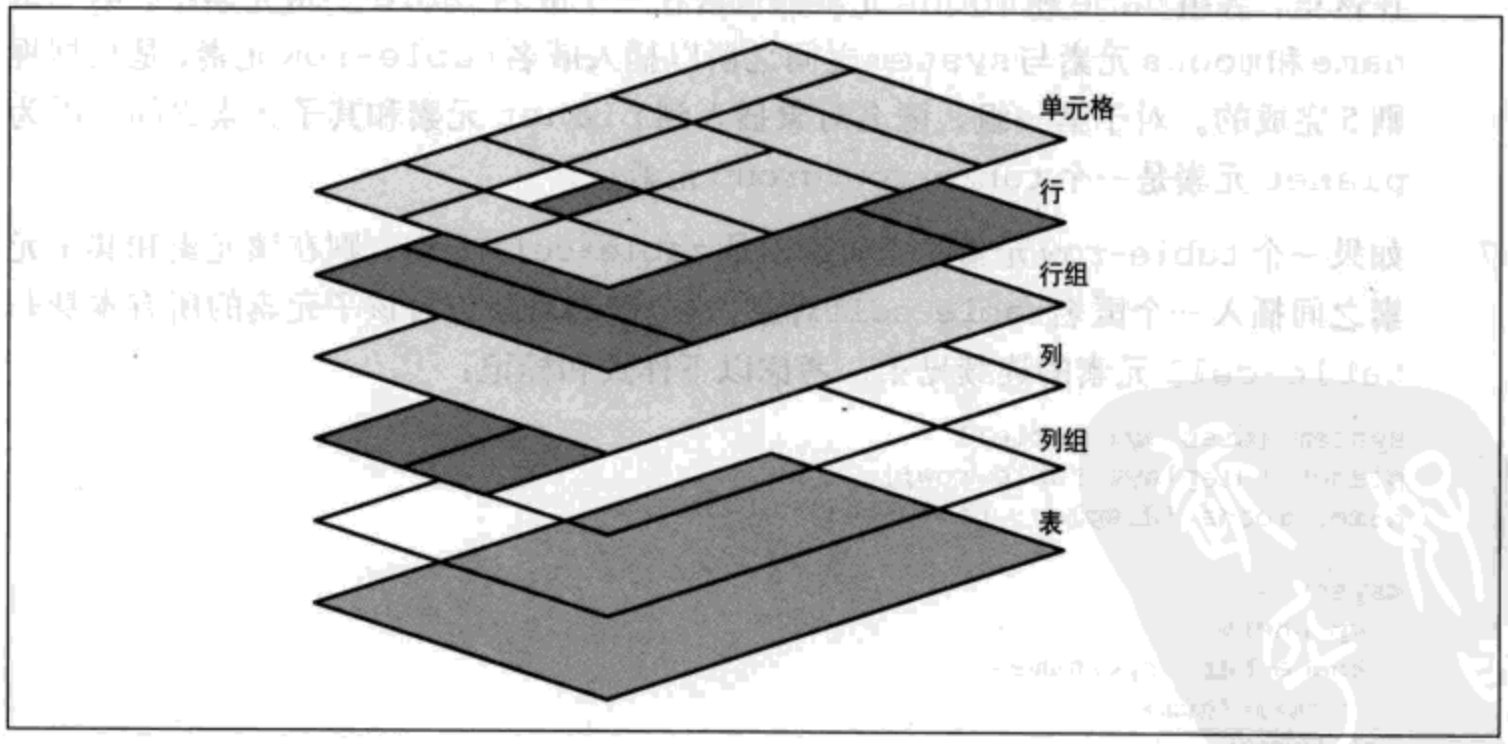

图11-3：表显示中使用的格式化层

基本说来，对应表各个方面的样式都在其各自的层上绘制。因此，如果table元素有一个绿色背景，并有1像素的黑色边框，这些样式就会在最低一层绘制。对应列组的样式在其上面一层绘制，列本身在更上一层，如此继续。顶层对应于表单元格，将最后绘制。

在极大程度上，这只是一个逻辑过程；毕竟，如果声明了表单元格的背景色，则希望绘

制在table元素的背景之上。图11-3反映出的最重要一点是，列样式在行样式的下面，所以行背景会覆盖列背景。

要记住重要的一点，默认地，所有元素背景都是透明的。因此，在以下标记中如果单元格、行、列等没有自己的背景，table元素的背景将“透过”这些内部元素可见，如图11-4所示：

```
<table style="background: #888;">
 <tr>
  <td>hey</td>
  <td style="background: #CCC;">there</td>
 </tr>
<tr>
  <td>what's</td>
  <td>up?</td>
 </tr>
<tr style="background: #AAA;">
  <td>tiger</td>
  <td style="background: #CCC;">lilly</td>
 </tr>
</table>
```

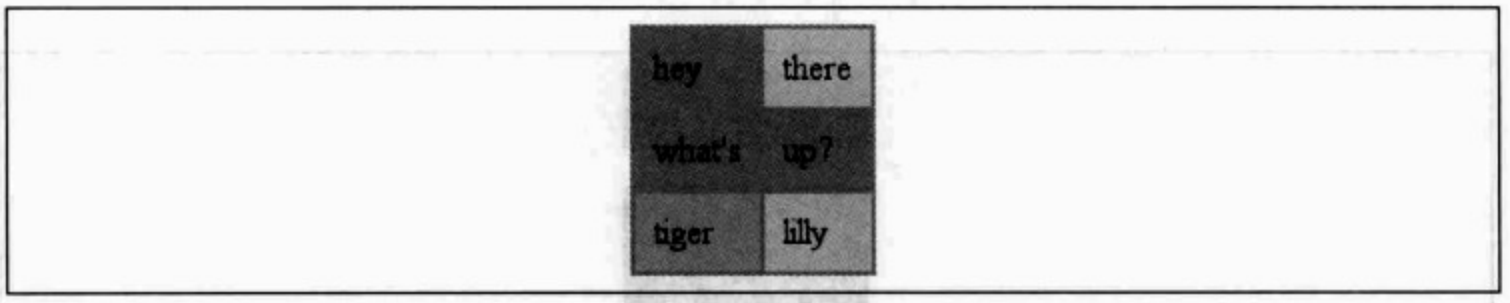

图11-4：透过其他层看到表格式化层的背景

表标题

顾名思义，表标题是一小段文本，描述了表内容的本质。因此，对于2003年第4季度的一个股票行情表，表标题元素的内容可能读作“Q4 2003 Stock Performance”。利用属性caption-side，可以把这个元素放在table之上，也可以放在table的底下，而不论该表标题出现在表结构中的哪个位置（在HTML中，caption元素只能出现在开始table元素的后面，不过其他语言可能有不同的规则）。

表标题有点奇怪，至少从视觉上看有些奇怪。CSS规范指出，表标题格式化为就好像它是直接放在表框之前（或之后）的一个块框，只有两个不同。首先，表标题仍能从表继承值；其次，用户代理在考虑如何处理表前面的run-in元素时会忽略表标题框。因此，如果一个run-in元素在表之前，它不会进入表的上标题，也不会进入表中，而是处理为好像其display值为block。

caption-side	
值：	top | bottom
初始值：	top
应用于：	display 值为 table-caption 的元素
继承性：	有
计算值：	根据指定确定
说明：	CSS2 中还有值 left 和 right，不过由于缺乏广泛的支持，这两个值在 CSS2.1 中已经去掉

通过一个简单的例子就可以说明有关表标题表示的大部分重要方面。考虑以下样式，如图 11-5 所示：

```
caption {background: gray; margin: 1em 0;
  caption-side: top;}
table{color: white; background: black; margin: 0.5em 0;}
```

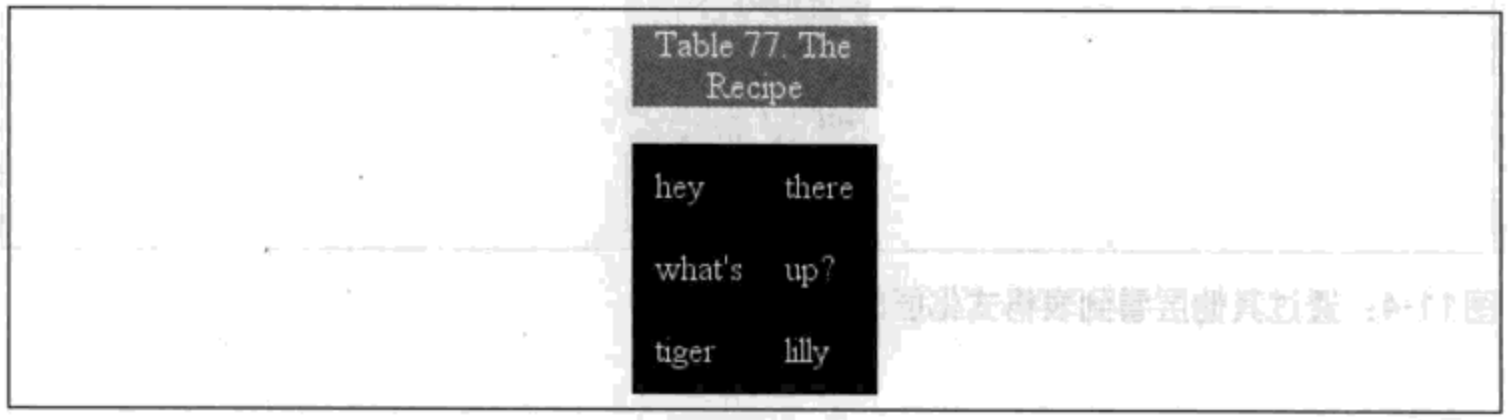

图11-5：对表标题和表应用样式

caption元素中的文本从table继承了color值white，但caption有自己的背景。table的外边框边界和 caption 的外边距边界之间的间隔为 1em，因为 table 的上外边距与 caption 的下外边距合并，见第 7 章的描述。最后，caption 的宽度要基于 table 元素的内容宽度，这是 caption 的包含块。如果把 caption-side 的值改为 bottom，会得到同样的结果，只不过 caption 将出现在表框的后面，另外将合并 caption 的上外边距和 table 的下外边距。

大多数情况下，为 caption 应用样式非常类似于块级元素；它们可以有内边距、边框、背景等。例如，如果需要改变 caption 中文本的水平对齐，可以使用属性 text-align。因此，要把上例中的标题右对齐，可以写作：

```
caption {background: gray; margin: 1em 0;
  caption-side: top; text-align: right;}
```

注意： 到2006年中期，为表标题设置样式还是一个很危险的事情。有些浏览器支持表标题的上下外边距，但有些并不支持；有些浏览器会相对于表的宽度计算表标题宽度，而另外一些浏览器则用完全不同的方法来计算。把所有这些行为都列出来是没有意义的，因为可以预见到这个领域的变化很快。之所以在此特别说明，主要是为了让读者对可能遇到的问题提前有所认识。

表单元格边框

CSS中实际上有两种截然不同的边框模型。按布局术语来讲，如果单元格相互之间是分隔的，就是分隔边框模型在起作用。另一种选择是合并边框模型，采用这种模型，单元格之间没有可见的间隔，单元格边框会相互合并。前者是默认模型，不过在CSS的较早版本中，后者才是默认模型。

创作人员可以用属性border-collapse在这两种模型中做出选择。

border-collapse	
值：	collapse | separate | inherit
初始值：	separate
应用于：	display值为table或inline-table的元素
继承性：	有
计算值：	根据指定确定
说明：	在CSS2中，默认值为collapse

这个属性的主要目的是为创作人员提供一个途径，来决定用户代理采用哪一种边框模型。如果值collapse起作用，则使用合并边框模型。如果选择了值separate，就会使用分隔边框模型。我们先来看后一种模型，因为它描述起来简单得多。

分隔单元格边框

采用这种模型，表中的每个单元格都与其他单元格分开一定距离，而且单元格的边框彼此不会合并。因此，给定以下样式和标记，可以看到如图11-6所示的结果：

```
table{border-collapse: separate;}
td {border: 3px double black; padding: 3px;}

<table cellspacing="0">
  <tr>
    <td>cell one</td>
    <td>cell two</td>
  </tr>
  <tr>
    <td>cell three</td>
    <td>cell four</td>
  </tr>
</table>
```

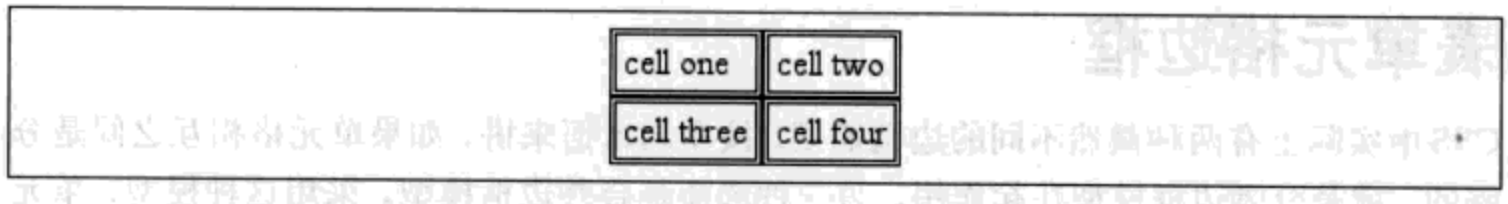

图11-6：分隔（并因此独立）单元格边框

注意，单元格边框相互挨着，但彼此保持区别。单元格之间的三条线实际上是相邻放置的两个double边框。

前例中加入了HTML属性cellspacing来确保单元格之间没有间隔，但是用这个属性好像有些麻烦。毕竟，如果能定义分隔边框，就应该有办法使用CSS改变单元格之间的间隔。幸运的是，确实有这样的办法。

边框间隔

可能有这样一种情况，你希望表单元格边框分隔一段距离。利用属性border-spacing很容易做到，它提供了强大的功能，完全可以替换HTML属性cellspacing。

border-spacing

值：	<length> <length>? | inherit
初始值：	0
应用于：	display值为table或inline-table的元素
继承性：	有
计算值：	两个绝对长度
说明：	除非border-collapse值为separate，否则会忽略该属性

可以为这个属性指定一个或两个长度值。如果希望所有单元格都分隔1个像素的距离，声明border-spacing: 1px;就足够了。另一方面，如果你希望单元格水平间隔1个像素，而垂直间隔5个像素，就要写作border-spacing: 1px 5px;。如果提供两个长度值，则要求第一个值始终是水平间隔，第二个值始终是垂直间隔。

表外围的单元格边框与表元素本身的内边距之间也可以指定间隔值。给定以下样式，可以得到如图11-7所示的结果：

```
table{border-collapse: separate; border-spacing: 3px 5px;
 padding: 10px; border: 2px solid black;}
td { border: 1px solid gray;}
td#squeeze {border-width: 5px;}
```

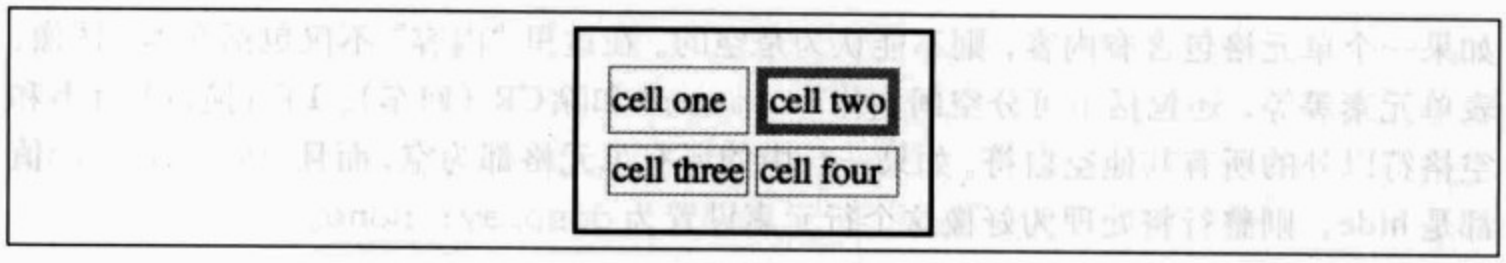

图11-7：单元格与其所在表之间设置边框间隔的效果

在图11-7中，两个水平相邻单元格的边框之间有3像素的空间，最右单元格的边框与table元素右边框之间以及最左单元格的边框与table元素左边框之间都有13像素的空间。类似地，垂直相邻单元格的边框相距5像素，最上面一行中单元格的边框与表的上边框之间以及最下面一行中单元格的边框与表的下边框之间都分别有15像素的空间。不论单元格本身的边框宽度是多少，单元格边框之间的间隔在整个表中是固定不变的。

还要注意，如果要声明一个border-spacing值，这会应用于表本身，而不是单个的单元格。如果为上例中的td元素声明border-spacing，则会被忽略。

在分隔边框模型中，不能为行、行组、列和列组设置边框。如果为这些元素声明了边框属性，都会被CSS兼容的用户代理所忽略。

处理空单元格

从视觉来看，由于每个单元格都区别于表中的所有其他单元格，对于空单元格（即没有内容）该如何处理呢？对此有两种选择，从empty-cells属性的值可以反映出来。

如果empty-cells设置为show，会画出空单元格的边框和背景，就好像这些表单元格有内容一样。如果值为hide，则不会画出该单元格的任何部分，就好像这个单元格被设置为visibility: hidden。

empty-cells	
值：	show | hide | inherit
初始值：	show
应用于：	display值为table-cell的元素
继承性：	有
计算值：	根据指定确定
说明：	除非border-collapse值为separate，否则会忽略该属性

如果一个单元格包含有内容，则不能认为是空的。在这里“内容”不仅包括文本、图像、表单元素等等，还包括不可分空间实体( )和除CR（回车）、LF（换行）、tab和空格符以外的所有其他空白符。如果一行中的所有单元格都为空，而且empty-cells值都是hide，则整行将处理为好像这个行元素设置为display: none。

注意：写作本书时，empty-cells还未得到Internet Explorer的充分支持。

合并单元格边框

合并单元格模型基本上描述了没有单元格间隔时HTML表通常如何布局，不过，这比分隔边框模型要更复杂一些。以下规则使合并单元格边框与分隔单元格边框有所区别：

- display值为table或inline-table的元素不能有任何内边距，不过它们可以有外边距。因此，表的外围边框与其最外单元格的边界之间不会有任何间隔。
- 边框可以应用到单元格、行、行组、列和列组。表元素本身通常都有一个边框。
- 单元格边框之间绝对不会有任何间隔。实际上，如果边框相邻，就会相互合并，使得实际上只画其中一个合并边框。这有些类似于外边距合并，即最大的一个外边距“胜出”。单元格边框合并时，“最有意思的”边框会胜出。
- 一旦合并，单元格之间的边框会在单元格间的假想表格线上居中。

下面两节将更详细地讨论后两点。

合并边框布局

为了更好地理解合并边框模型如何工作，下面来看一个表行的布局，如图11-8所示。

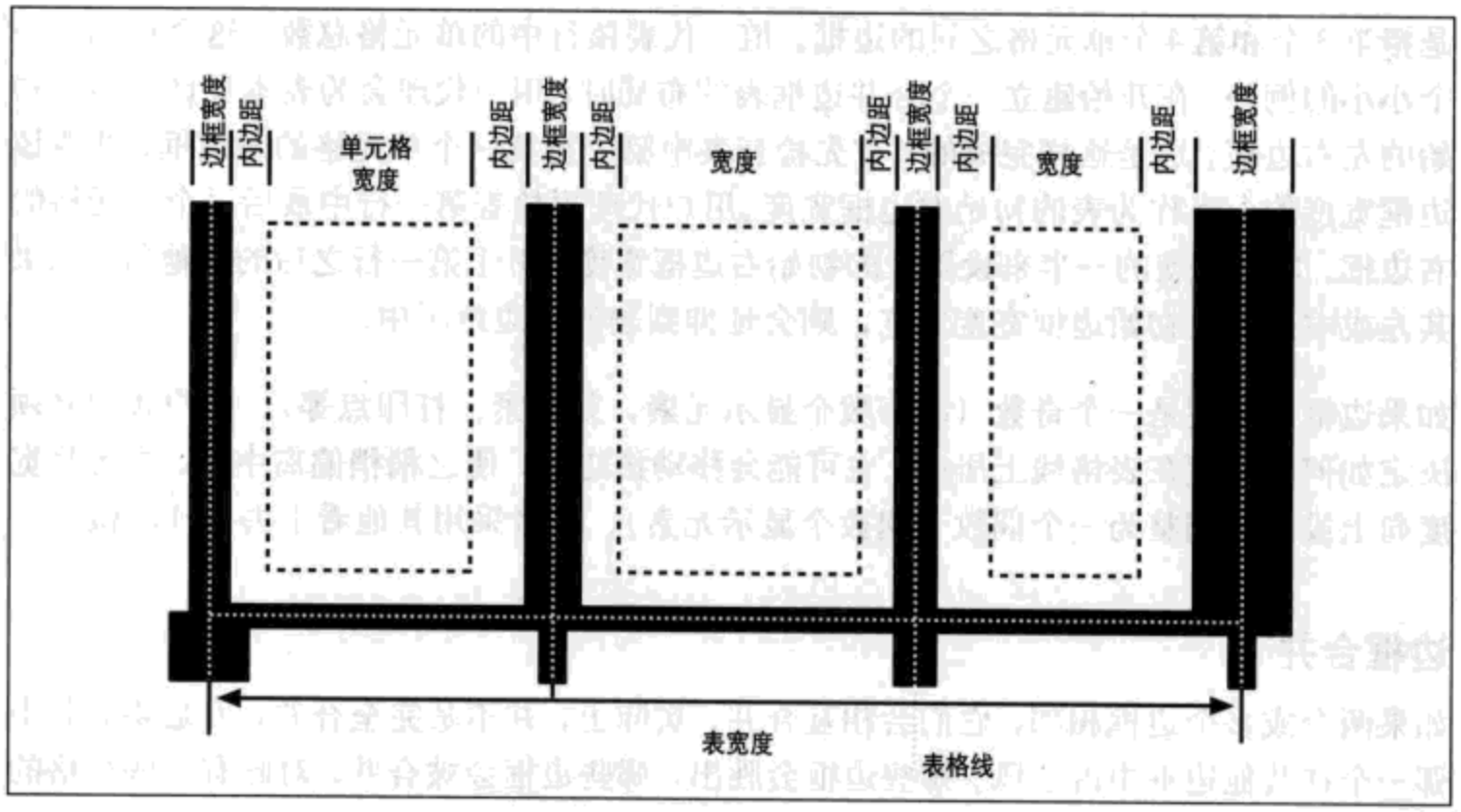

图11-8：使用合并边框模型的表行布局

不出所料，每个单元格的内边距和内容宽度都在边框以内。对于单元格之间的边框，一半边框放在两个单元格之间表格线的一边，另一半放在另一边。不论哪一种情况，沿着各单元格边界只会画一个边框。你可能认为总是在表格线两边分别画各单元格的一半边框（译注2），但实际上并不是这样。

例如，假设中间单元格的实线边框是绿色，外面两个单元格的实线边框是红色。中间单元格左右两边的边框（与外面两个单元格的相邻边框合并）都将是绿色，或都是红色，这取决于哪个边框胜出。下一节我们将讨论如何区分哪个边框会取胜。

你可能已经注意到，外边框超出了表的宽度。这是因为，在这个模型中，宽度只包含表边框的一半。另一半在此距离之外，落在外边距中。看上去可能有些怪异，不过模型就是这样定义的。

规范提供了一个布局公式，为了方便大家学习，以下列出这个公式：

```
row width = (0.5 * border-width0) + padding-left1 + width1 + padding-right1 +
 border-width1 + padding-left2 +...+ padding-rightn + (0.5 * border-widthn)
```

各$\texttt{border-width}_i$是指单元格i与下一个单元格之间的边框；因此，border-width3

译注2：例如，表格线左边总是画前一个单元格右边框的一半，表格线右边总是画单元格左边框的一半。

是指第3个和第4个单元格之间的边框。值n代表该行中的单元格总数。这个机制有一个小小的例外。在开始建立一个合并边框表的布局时，用户代理会为表本身计算一个初始的左右边框。这是这样完成的：首先检查表中第一行第一个单元格的左边框，并取该边框宽度的一半作为表的初始左边框宽度。用户代理再检查第一行中最后一个单元格的右边框，取其宽度的一半来设置表的初始右边框宽度。对于第一行之后的其他行，如果其左或右边框比初始边框宽度更宽，则会延伸到表的外边距区中。

如果边框的宽度是一个奇数（即奇数个显示元素，如像素、打印点等），用户代理必须决定如何将边框在表格线上居中。它可能会移动该边框，使之稍稍偏离中心，或者将宽度向上或向下调整为一个偶数（偶数个显示元素），或者采用其他看上去合理的做法。

边框合并

如果两个或多个边框相邻，它们会相互合并。实际上，并不是完全合并，而是要看其中哪一个在其他边框中占上风。哪些边框会胜出，哪些边框会被合并，对此有一些严格的规则：

- 如果某个合并边框的border-style为hidden，它会优先于所有其他合并边框。这个位置上的所有边框都隐藏。
- 如果某个合并边框的border-style为none，它的优先级最低。这个位置上不会画出该边框，除非所有其他合并边框的border-style值都是none。注意，none是border-style的默认值。
- 如果至少有一个合并边框的border-style值不是none，而且所有合并边框的border-style值都不是hidden，则窄边框不敌更宽的边框。如果多个合并边框有相同的宽度，则会考虑边框样式，并采用以下顺序（从最优先到最不优先）：double、solid、dashed、dotted、ridge、outset、groove、inset。因此，如果两个有相同宽度的边框合并，而其中一个是dashed边框，另一个是outset边框，该位置上的边框将是虚线边框。
- 如果合并边框的样式和宽度都一样，但是颜色不同，则按下表顺序使用元素的颜色（从最优先到最不优先）：cell、row、row group、column、column group、table。因此，如果一个单元格和一个列的边框合并（除颜色外，所有其他方面都一样），会使用单元格的边框颜色（和样式及宽度）。如果合并边框来自相同类型的元素，如两个有相同样式和宽度但不同颜色的行边框，则颜色取最上最左边框的颜色（在从左向右读的语言中是这样；而在从右向左读的语言中，则取最上最右边框的颜色）。

以下样式和标记有助于说明这4个规则（结果见图11-9所示）：

```
table{border-collapse: collapse;
  border: 3px outset gray;}
td {border: 1px solid gray; padding: 0.5em;}
#r2c1, #r2c2 {border-style: hidden;}
#r1c1, #r1c4 {border-width: 5px;}
#r2c4 {border-style: double; border-width: 3px;}
#r3c4 {border-style: dotted; border-width: 2px;}
#r4c1 {border-bottom-style: hidden;}
#r4c3 {border-top: 13px solid silver;}

<table>
<tr>
<td id="r1c1">1-1</td><td id="r1c2">1-2</td>
<td id="r1c3">1-3</td><td id="r1c4">1-4</td>
</tr>
<tr>
<td id="r2c1">2-1</td><td id="r2c2">2-2</td>
<td id="r2c3">2-3</td><td id="r2c4">2-4</td>
</tr>
<tr>
<td id="r3c1">3-1</td><td id="r3c2">3-2</td>
<td id="r3c3">3-3</td><td id="r3c4">3-4</td>
</tr>
<tr>
<td id="r4c1">4-1</td><td id="r4c2">4-2</td>
<td id="r4c3">4-3</td><td id="r4c4">4-4</td>
</tr>
</table>
```

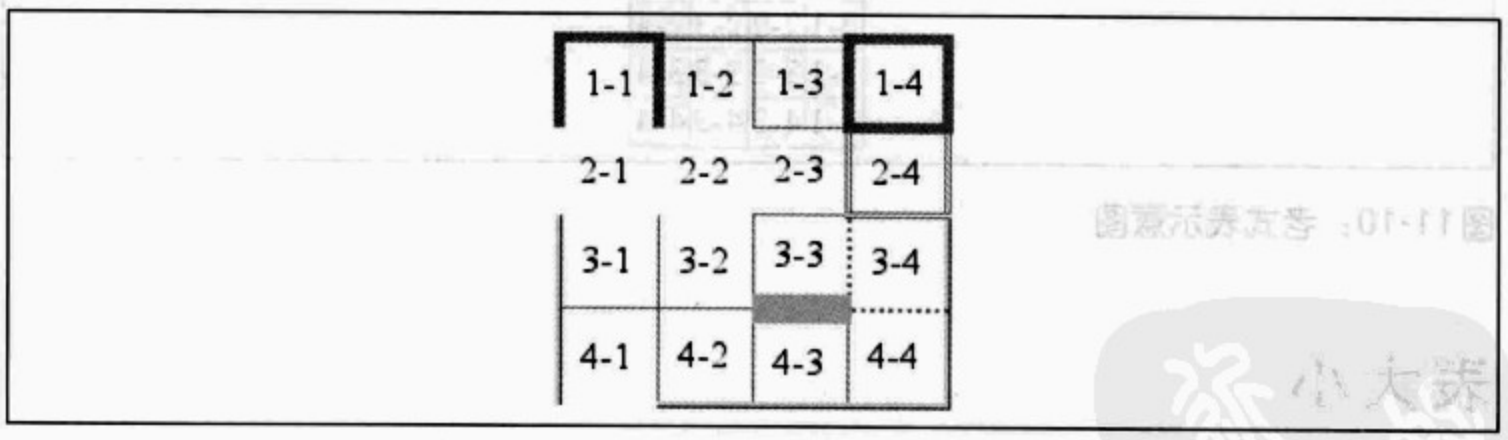

图11-9：管理边框宽度、样式和颜色，得到一些不寻常的结果

下面依次考虑对各个单元格发生了什么：

- 对于单元格1-1和1-4，5像素的边框比其他所有相邻边框都宽，所以5像素的边框不仅会胜出其相邻单元格边框，还会优先于表本身的边框。只有单元格1-1的下边框例外，在此不再显示5像素的边框。
- 单元格1-1的下边框之所以没有胜出，是因为单元格2-1和2-2显式声明了边框隐藏，这就从这些单元格的边界上去除了所有边框。同样地，表的边框（单元格2-1

左边界上的边框）也落败于该单元格的边框。单元格4-1的下边框也隐藏，所以这个单元格下面不会出现任何边框。

- 单元格2-4的3像素宽double边框顶部被单元格1-4的5像素实线边框所覆盖，不过这个double边框会覆盖该单元格本身与单元格2-3之间的边框，因为这个double边框不仅更宽而且“更有意思”。单元格2-4的边框还覆盖了它自己与单元格3-4之间的边框，尽管二者宽度相同，但单元格2-4边框的double样式定义为比单元格3-4的dotted边框“更有意思”。
- 单元格3-3的13像素银色下边框不仅会覆盖单元格4-3的上边框，还会影响这两个单元格以及包含这两个单元格的行中的内容布局。
- 对于沿着表外边界而且没有指定样式的单元格，其1像素实线边框会被table元素本身的3像素outset边框所覆盖。

听上去很复杂，也确实如此，不过这些行为都很直观，通过动手实践就会更清楚。但是需要指出，对于Netscape 1时代的基本HTML表，其表示仅用很简单的一组规则就可以描述，如下（结果见图11-10所示）：

```
table{border-collapse: collapse; border: 2px outset gray;}
td {border: 1px inset gray;}
```

1-1	1-2	1-3	1-4
2-1	2-2	2-3	2-4
3-1	3-2	3-3	3-4
4-1	4-2	4-3	4-4

图11-10：老式表示意图

表大小

既然我们已经深入讨论过表的格式化和单元格边框显示，下面需要了解如何确定表及其内部元素的大小。谈到确定表的宽度，有两种不同的方法：固定宽度布局和自动宽度布局。不论使用何种宽度算法，高度都会自动计算。

宽度

由于有两种不同方法可以得出表的宽度，因此必须有办法声明一个给定表应当使用哪一种方法。创作人员可以使用属性table-layout来选择采用哪种方法计算表的宽度。

table-layout	
值：	auto | fixed | inherit
初始值：	auto
应用于：	display 值为 table 或 inline-table 的元素
继承性：	有
计算值：	根据指定确定

尽管这两个模型针对一个特定表布局可能有不同的结果，但二者之间最显著的差异是速度。使用固定宽度表布局时，相对于自动宽度模型，用户代理可以更快地计算出表的布局。

固定布局

固定布局模型的速度之所以快，主要原因是布局不依赖于表单元格的内容。其布局是根据该表以及表中列和单元格的 width 值决定的。

固定布局模型的工作包括以下简单步骤：

1. width 属性值不是 auto 的所有列元素会根据 width 值设置该列的宽度。
2. 如果一个列的宽度为 auto —— 不过，表首行中位于该列的单元格 width 不是 auto —— 则根据该单元格宽度设置此列的宽度。如果这个单元格跨多列，则宽度在这些列上平均分配。
3. 在以上两步之后，如果列的宽度仍为 auto，会自动确定其大小，使其宽度尽可能相等。

此时，表的宽度设置为表的 width 值或列宽度之和（取其中较大者）。如果表宽度大于其列宽总和，将二者之差除以列数，再把得到的这个宽度增加到每一列上。

这种方法的速度很快，因为所有列宽度都由表的第一行定义。首行后所有行中的单元格都根据首行所定义的列宽确定大小。后面这些行中的单元格不会改变列宽，这意味着为这些单元格指定的 width 值都会被忽略。如果一个单元格的内容无法放下，该单元格的 overflow 值将决定单元格内容是剪裁、可见还是生成一个滚动条。

考虑以下样式和标记（如图 11-11 所示）：

```
table{table-layout: fixed; width: 400px;
  border-collapse: collapse;}
```

```
td {border: 1px solid;}
col#c1 {width: 200px;}
#r1c2 {width: 75px;}
#r2c3 {width: 500px;}

<table>
<colgroup>
<col id="c1"><col id="c2"><col id="c3"><col id="c4">
</colgroup>
<tr>
<td id="r1c1">1-1</td><td id="r1c2">1-2</td>
<td id="r1c3">1-3</td><td id="r1c4">1-4</td>
</tr>
<tr>
<td id="r2c1">2-1</td><td id="r2c2">2-2</td>
<td id="r2c3">2-3</td><td id="r2c4">2-4</td>
</tr>
<tr>
<td id="r3c1">3-1</td><td id="r3c2">3-2</td>
<td id="r3c3">3-3</td><td id="r3c4">3-4</td>
</tr>
<tr>
<td id="r4c1">4-1</td><td id="r4c2">4-2</td>
<td id="r4c3">4-3</td><td id="r4c4">4-4</td>
</tr>
</table>
```

1-1	1-2	1-3	1-4
2-1	2-2	2-3	2-4
3-1	3-2	3-3	3-4
4-1	4-2	4-3	4-4

图11-11：固定宽度表布局

在图11-11中可以看到，第一列是200像素宽，这正好是表宽度400像素的一半。第二列宽度为75像素，因为该列中首行单元格明确指定了宽度。第三列和第四列分别为61像素宽。为什么呢？因为第一列和第二列的列宽之和（275px）再加上列之间各个边框（3px）一共等于278像素。400减去278等于122，而122分为两半就是61，所以这就是第三列和第四列的宽度。为`#r2c3`指定的500像素宽度如何处理呢?这会被忽略，因为该单元格不在表的首行中。

注意，使用固定宽度布局模型时，没有必要非得为表指定一个显式宽度，不过如果指定一个宽度确实有所帮助。例如，给定以下样式，用户代理可能计算出表的宽度比父元素的`width`窄50像素。它就会在固定布局算法中使用计算得到的这个宽度：

```
table{table-layout: fixed; margin: 0 25px;
  width: auto;}
```

不过，这不是必要的。用户代理也可以使用自动宽度布局模型对width值为auto的表完成布局。

自动布局

自动布局模型尽管没有固定布局那么快，不过对你来说可能更熟悉，因为这正是HTML表使用多年的模型。在大多数当前用户代理中，只要表的width为auto就会触发使用这个模型，而不论table-layout的值是什么，不过这一点不能保证。

为什么自动布局比较慢，因为在用户代理查看完表的所有内容之前无法确定表的布局。也就是说，自动布局要求用户代理每得到一个新单元格时都完成整个表的布局。这通常需要用户代理完成一些计算，然后再返回头来对表完成第二组计算。必须充分分析单元格的内容，因为与HTML表类似，表布局完全取决于单元格的内容。如果最后一行一个单元格中有一个400像素宽的图像，就会要求它上面的所有单元格（同列中的单元格）都是400像素宽。因此，必须计算每个单元格的宽度，而且在确定表的布局之前还必须做一些调整（可能触发另一轮内容宽度计算）。

这个模型的详细过程见以下步骤：

1. 对于一列中的各个单元格，计算最小和最大单元格宽度。
   a. 确定显示内容所需的最小宽度。要记住，内容可以流入多行，不过不能超出单元格框。如果单元格的width值大于最小可能宽度，则把最小单元格宽度设置为该width值。如果单元格的width值为auto，最小单元格宽度则设置为最小内容宽度。
   b. 对于最大宽度，要确定完全显示内容而且不包括换行符所需的宽度（除非明确要求，例如指出可以有
元素）。这个值就是最大单元格宽度。
2. 对于各一列，计算最小和最大列宽。
   a. 列的最小宽度由该列中所有单元格的最小单元格宽度的最大值确定。如果为该列指定的width值大于列中所有最小单元格宽度，最小列宽则设置为这个width值。
   b. 要计算最大宽度，取该列中所有单元格的最大单元格宽度的最大值。如果已经为列指定了一个width值，而且大于该列中的所有最大单元格宽度，最大列宽则设置为该width值。这两种行为改写了传统的HTML表行为，对于HTML表，会强制列扩展为与其最宽的单元格同宽。
3. 如果一个单元格跨多列，最小列宽之和必须等于这个跨列单元格的最小单元格宽

度。类似地，最大列宽之和必须等于跨列单元格的最大宽度。如果列宽之和与单元格宽度有差距，用户代理会把这个差距在所跨的列上平均分配。

另外，用户代理必须考虑到这样一个问题：如果一个列的width值为百分数值，这个百分数要相对于表的宽度计算，即便它还不知道这个宽度是多少！它必须把这个百分数保存起来，在算法的下一部分使用。

此时，用户代理已经确定了各列可能是多宽或多窄。有了这个信息，可以再真正得出表的宽度。这个过程如下：

1. 如果表的计算宽度值不是auto，将这个计算表宽度值与所有列宽再加上所有边框和单元格间隔之和相比较（设置为百分数宽度的列往往在此时计算具体宽度）。二者中较大的一个就是表的最终宽度。如果表的计算宽度值大于列宽、边框和单元格间隔之和，所有列的宽度都会增加一个相等的量，使得刚好将表完全填充。
2. 如果表的计算宽度值为auto，通过将列宽、边框和单元格间隔相加来确定表的最终宽度。这说明表的宽度只能恰好显示其内容，而不能有多余，这类似于传统的HTML表。设置为百分数宽度的列会以这个百分数作为一个限制，不过用户代理有可能并不满足这个限制。

只有在完成了最后一步之后，用户代理才算真正建立了表的布局。

以下样式和标记（如图11-12所示）有助于说明这个过程：

```
table{table-layout: auto; width: auto;
  border-collapse: collapse;}
td {border: 1px solid;}
col#c3 {width: 25%;}
#r1c2 {width: 40%;}
#r2c2 {width: 50px;}
#r2c3 {width: 35px;}
#r4c1 {width: 100px;}
#r4c4 {width: 1px;}
```

```
<table>
<colgroup>
<col id="c1"><col id="c2"><col id="c3"><col id="c4">
</colgroup>
<tr>
<td id="r1c1">1-1</td><td id="r1c2">1-2</td>
<td id="r1c3">1-3</td><td id="r1c4">1-4</td>
</tr>
<tr>
<td id="r2c1">2-1</td><td id="r2c2">2-2</td>
<td id="r2c3">2-3</td><td id="r2c4">2-4</td>
</tr>
```

```
<tr>
<td id="r3c1">3-1</td><td id="r3c2">3-2</td>
<td id="r3c3">3-3</td><td id="r3c4">3-4</td>
</tr>
<tr>
<td id="r4c1">4-1</td><td id="r4c2">4-2</td>
<td id="r4c3">4-3</td><td id="r4c4">4-4</td>
</tr>
</table>
```

1-1	1-2	1-3	1-4
2-1	2-2	2-3	2-4
3-1	3-2	3-3	3-4
4-1	4-2	4-3	4-4

图11-12：自动表布局

下面依次考虑对各个列会发生什么：

- 对于第一列，唯一明确的单元格或列的宽度是单元格4-1的宽度，其width指定为100px。由于它的内容太短，所以最小和最大列宽都是100px（如果列中一个单元格的文本有很多句，则要把最大列宽增加到足以不换行显示所有文本所需的宽度值）。
- 第二列中声明了两个宽度：单元格1-2的width指定为40%，单元格2-2的width指定为50px。这一列的最小宽度为50px，最大宽度为最终表宽度的40%。
- 对于第三列，只是单元格3-3有明确的宽度（35px），不过该列本身也指定了宽度，width为25%。因此，最小列宽为35px，最大宽度为最终表宽度的25%。
- 对于第四列，只为单元格4-4指定了宽度为1px。这比最小内容宽度小，所以最小和最大列宽都等于单元格的最小内容宽度，计算为25像素。

用户代理现在知道了，这4个列的最小和最大列宽如下：

1. 最小 100px / 最大 100px
2. 最小 50px / 最大 40%
3. 最小 35px / 最大 25%
4. 最小 25px / 最大 25px

因此，表的最小宽度就是所有这些列的最小宽度再加上边框之和，总共是215像素。表的最大宽度是130px + 65%，最后得出371.42857143像素（假设130px表示总表宽度

的35%)。将这个小数取整为371像素后，假设用户代理实际使用的就是这个宽度。因此，第二列将是148像素宽，第三列将是93像素宽。用户代理不必真正使用最大值，完全可以选择其他方案。

当然，这是一个相当简单直接的例子（不过看上去可能并非如此）：所有内容基本上都有相同的宽度，声明的大多数宽度都使用像素数指定。如果表中包含间隔GIF、文本段落、表单元素等，要得出表的布局，过程可能要麻烦得多。

高度

确定表的宽度要花这么大功夫，现在你可能想知道计算高度又会多复杂。实际上，在CSS中，高度计算相当简单，不过浏览器开发人员可能不这么认为。

最容易的一种情况是，直接由height属性显式设置高度。在这种情况下，表的高度就由height值确定。这说明表可能比行高总和高或矮，不过2006年4月11日发布的CSS 2.1规范草案指出，将height看作是表框的最小高度。对于这些情况，CSS2.1规范明确地拒绝定义会发生什么，而只是指出CSS的将来版本可能解决这个问题。用户代理可以扩展或收缩表中的表行来适应表的高度，也可以在表框中留白或者采取其他完全不同的做法。这完全由各个用户代理决定。

如果表的高度是auto，其高度则是表中所有行的行高再加上所有边框和单元格间隔的总和。要确定各行的高度，用户代理需要完成一个与确定列宽类似的过程。它要计算各单元格内容的最小和最大高度，然后使用这些最小和最大高度得出行的最小和最大高度。确定了所有行的最小和最大高度后，用户代理再得出各行的高度应当是多少，把这些行加在一起，然后使用这个计算确定表的高度。这与行内布局很类似，只不过对于结果不那么确定。

除了处理有明确高度的表以及表中的行高，还可以增加以下内容，这是CSS2.1中没有定义的：

- 表单元格高度为百分数时的效果
- 表行和行组高度为百分数时的效果
- 跨行单元格如何影响所跨行的高度（除了这些行必须包含该跨行单元格）

可以看到，表的高度计算在很大程度上留给用户代理来决定。历史证明，各用户代理很可能有不同的做法，所以你要尽可能避免设置高度。

对齐

有意思的是，相对于单元格和行的高度，定义单元格中内容的对齐要容易得多。甚至垂直对齐也很容易定义（垂直对齐很容易影响行高）。

水平对齐是最简单的。要让一个单元格中的内容对齐，可以使用text-align属性。实际上，单元格会处理为一个块级框，其中的所有内容都根据text-align值对齐（有关text-align的详细内容见第6章）。

要将一个表单元格中的内容垂直对齐，可以使用vertical-align属性。它使用的很多值与垂直对齐行内内容是一样的，不过应用到表单元格时这些值的含义有所变化。下面对三种最简单的情况做个总结：

top
: 单元格内容的顶端与其行顶端对齐；对于跨行单元格，单元格内容的顶端与其所跨的第一行的顶端对齐。

bottom
: 单元格内容的底端与其行底端对齐；对于跨行单元格，单元格内容的底端与其所跨的最后一行的底端对齐。

middle
: 单元格内容的中间与其行中间对齐；对于跨行单元格，单元格内容的中间与其所跨行的中间对齐。

这三种情况见图11-13所示，其中使用了以下样式和标记：

```
table{table-layout: auto; width: 20em;
  border-collapse: separate; border-spacing: 3px;}
td {border: 1px solid; background: silver;
  padding: 0;}
div {border: 1px dashed gray; background: white;}
#r1c1 {vertical-align: top; height: 10em;}
#r1c2 {vertical-align: middle;}
#r1c3 {vertical-align: bottom;}

<table>
<tr>
<td id="r1c1">
<div>
The contents of this cell are top-aligned.
</div>
</td>
<td id="r1c2">
<div>
The contents of this cell are middle-aligned.
```

```
</div>
</td>
<td id="r1c3">
<div>
The contents of this cell are bottom-aligned.
</div>
</td>
</tr>
</table>
```

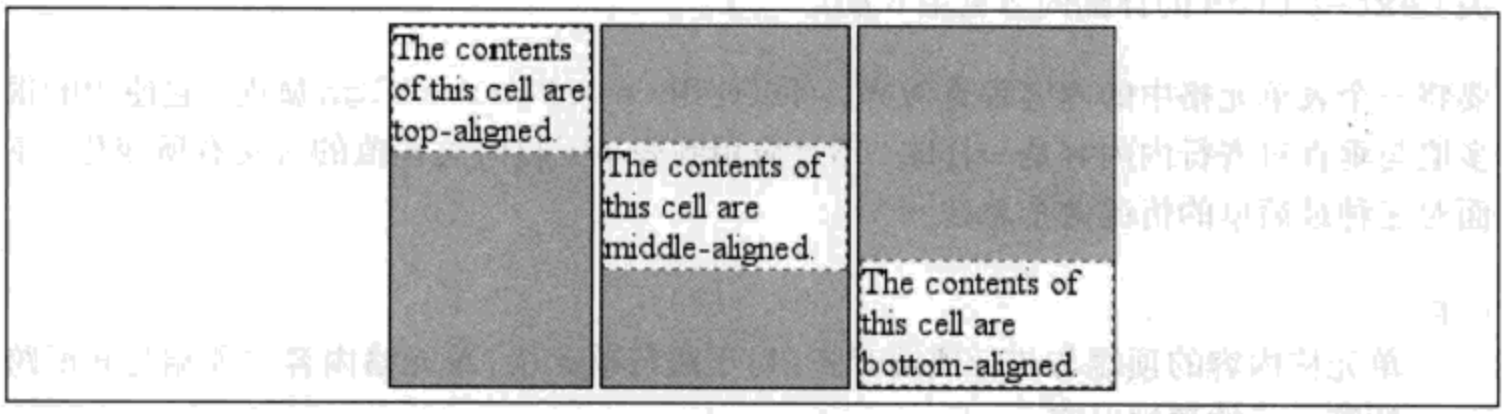

图11-13：单元格内容的垂直对齐

在上述各种情况下，会自动地增加单元格本身的内边距完成对齐，来达到所需的效果。图11-13的第一个单元格中，单元格的下内边距修改为等于单元格框的高度与单元格中内容的高度之差。对于第二个单元格，单元格的上下内边距重置为相等，从而将单元格的内容垂直居中。最后一个单元格中，单元格的上内边距得到修改。

第4个可能的对齐值是baseline，这比前三个值要稍微复杂一些：

baseline

单元格的基线与其行的基线对齐；对于跨行单元格，该单元格的基线与所跨的第一行的基线对齐。

最好先提供一个示例演示（见图11-14所示），再讨论到底发生了什么。

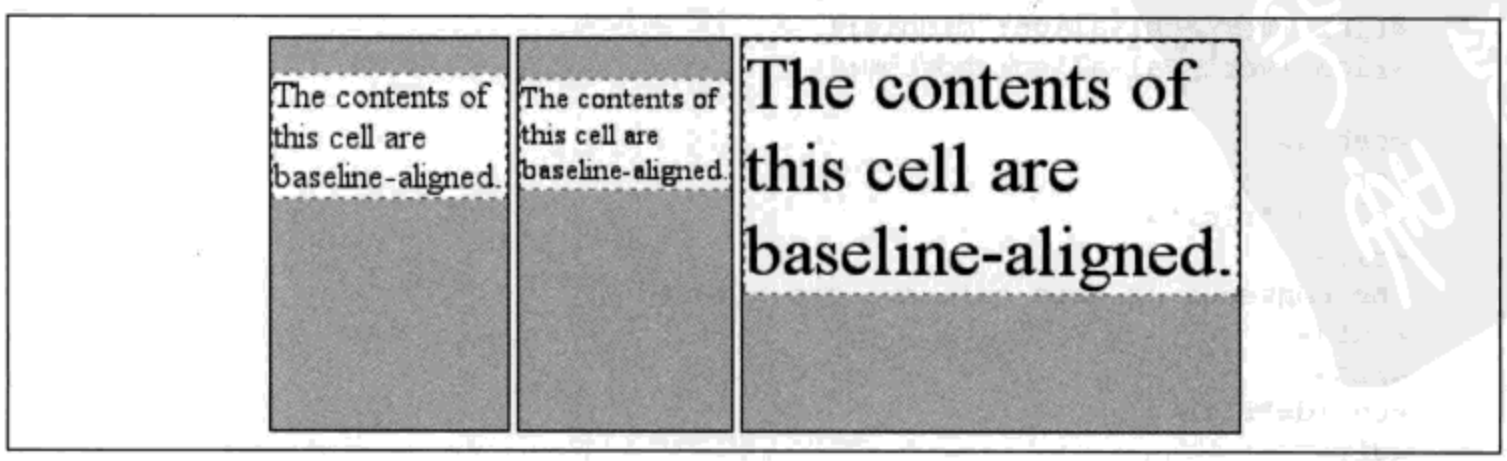

图11-14：单元格内容的基线对齐

行的基线由该行所有单元格中最低初始单元格基线定义(也就是第一个文本行的基线)。因此，在图 11-14 中，行的基线由第三个单元格定义，这个单元格初始基线最低。前两个单元格则将其第一个文本行的基线与该行基线对齐。

与顶端对齐、居中对齐和底端对齐类似，基线对齐的单元格内容也是通过调整单元格的上下内边距来放置。如果一行中的所有单元格都不是基线对齐，那么这一行甚至没有基线，它也不需要有基线。

对一行中的单元格内容对齐的详细过程如下：

1. 如果某些单元格是基线对齐，则确定该行的基线，并放置这些基线对齐单元格的内容。
2. 放置所有顶端对齐单元格的内容。行现在有了一个临时高度，由已经放置了内容的单元格的最低单元格底端确定。
3. 如果剩下的单元格是居中对齐或底端对齐，而且内容高度大于临时行高，行高则增加到可以包含其中最高的单元格。
4. 放置所有余下单元格的内容。如果单元格内容的高度比行高小，则增加单元格的内边距，使之适应行高。

vertical-align 值 sub、super、text-top 和 text-bottom 应用到表单元格时会被忽略。因此，以下规则的结果与如图 11-14 所示相同：

```
th {vertical-align: text-top;}
```

小结

可能你使用表和间隔设计已经有很多年，对表布局非常熟悉，但要知道这种布局的基本机制相当复杂，而且不是完全确定的。由于HTML表构造的历史渊源，CSS表模型也是以行为中心，不过好在它确实能提供列布局，还能应用有限的列样式。由于CSS提供了一些新特性来控制单元格对齐和表宽度，现在已经有了更多的工具能以更漂亮的方式表示表。

由于能够向任意元素应用与表有关的display值，这就大开了方便之门，可以使用div等HTML元素创建类似于表的布局（或者在XML语言中，任何元素都可以用来描述布局组件）。在写作本书时，大多数浏览器（除Internet Explorer）都支持向任意元素应用与表有关的 display 值。甚至在最新版本中，CSS还支持更复杂的表示，这正是下一章的主题：生成内容。

第12章

列表与生成内容

在CSS布局领域，列表是很有意思的一个方面。列表中的项其实就是块框，不过比起平常稍微多了一点，多出的这部分不属于文档布局，而只是“挂”在一边。对于一个有序列表，额外的这部分包含一系列递增的数字（或字母），这些数字（或字母）由用户代理计算，并且主要由用户代理格式化，而不是由创作人员完成。按文档结构的“指示”，用户代理会生成这些数字并提供基本表示。

这种内容生成在CSS1中是无法描述的，所以CSS1也无法控制，不过CSS2引入了一些特性，可以描述这种列表项编号。因此，现在CSS允许创作人员定义自己的计数模式和格式，而且可以将这些计数器与任何元素关联，而不只是有序列表。另外，利用这种基本机制还有可能向文档中插入其他类型的内容，包括文本串、属性值，甚至外部资源。所以，完全可以使用CSS在设计中插入链接图标、编辑符号等，而不必创建额外的标记。

要了解所有这些列表选项应如何加以利用，先来讨论基本的列表样式，然后再讨论内容和计数器的生成。

列表

从某种意义上讲，不是描述性文本的任何内容都可以认为是列表。人口普查、太阳系、家谱、餐馆菜单，甚至你的所有朋友都可以表示为一个列表或者是列表的列表。由于形式如此多样，这使得列表相当重要，所以说，CSS中列表样式不太丰富确实是一大憾事。

要影响一个列表的样式，最简单（同时支持最充分）的办法就是改变其标志类型。例如，在一个无序列表中，列表项的标志（marker）是出现在各列表项旁边的圆点。在有序列

表中，标志可能是一个字母、数字或另外某种计数体系中的一个符号。甚至可以将标志替换为图像。所有这些都可以使用不同的列表样式属性完成。

列表类型

要修改用于列表项的标志类型，可以使用属性 list-style-type。

list-style-type

CSS2.1 值：	disc | circle | square | decimal | decimal-leading-zero | lower-roman | upper-roman | lower-greek | lower-latin | upper-latin | armenian | georgian | none | inherit
CSS2 值：	disc | circle | square | decimal | decimal-leading-zero | upper-alpha |lower-alpha | upper-roman | lower-roman | lower-greek | hebrew | armenian | georgian | cjk-ideographic | hiragana | katakana | hiragana-iroha | none | inherit
初始值：	disc
应用于：	display 值为 list-item 的元素
继承性：	有
计算值：	根据指定确定

没错，这里的关键字确实不少；其中一些在CSS2中引入，但是在CSS2.1中已经去除。表 12-1 列出了还留在CSS2.1 中的关键字。

表 12-1：CSS2.1 中的 list-style-type 属性关键字

关键字	效果
disc	使用一个实心圆作为列表项标志
circle	使用一个空心圆作为列表项标志
square	使用一个方块（实心或空心）作为列表项标志
decimal	1, 2, 3, 4, 5,……
decimal-leading-zero	01, 02, 03, 04, 05, ……
upper-alpha	A, B, C, D, E, ……
upper-latin	

表12-1：CSS2.1中的list-style-type属性关键字（续）

关键字	效果
lower-alpha	a, b, c, d, e, ……
lower-latin	
upper-roman	I, II, III, IV, V,……
lower-roman	i, ii, iii, iv, v, ……
lower-greek	传统小写希腊符号
armenian	传统亚美尼亚序号
georgian	传统乔治序号
none	不使用标志

表12-2列出了CSS2中存在但已被CSS2.1去除的关键字。

表12-2：CSS2中的list-style-type属性关键字

关键字	效果
hebrew	传统希伯来序号
cjk-ideographic	表意数字
katakana	日文序号（A, I, U, E, O……）
katakana-iroha	日文序号（I, RO, HA, NI, HO……）
hiragana	日文序号（a, i, u, e, o……）
hiragana-iroha	日文序号（i, ro, ha, ni, ho……）

用户代理会把它无法识别的值处理为decimal。

list-style-type属性（以及所有其他与列表相关的属性）只能应用于display值为list-item的元素，不过CSS无法区别有序列表项和无序列表项。因此，完全可以设置一个有序列表使用实心圆而非数字作为列表项标志。实际上，list-style-type的默认值就是disc，所以可以得出结论，如果没有显式地声明其他列表类型，所有列表（有序或无序）对各列表项都会使用实心圆作为标志。这是合理的，不过最终要由用户代理来决定。尽管用户代理没有预定的规则，如{list-style-type: decimal;}，但它可能会禁止对无序列表使用有序标志，反之亦然。不过不能依赖于此，所以一定要当心。

对于hebrew和georgian等CSS2值，CSS2规范并没有明确地指出这些计数体系如

何工作，也没有说明用户代理应当如何处理。这种不确定性导致这些值未能广泛实现，所以表 12-2 中的值未出现在 CSS2.1 中。

如果想完全禁止显示标志，只能使用值none。none会导致用户代理在原本放标志的位置上不显示任何内容，不过它不会中断有序列表中的计数。因此，以下标记会得到如图 12-1 所示的结果：

```
ol li {list-style-type: decimal;}
li.off {list-style-type: none;}

<ol>
<li>Item the first
<li class="off">Item the second
<li>Item the third
<li class="off">Item the fourth
<li>Item the fifth
</ol>
```

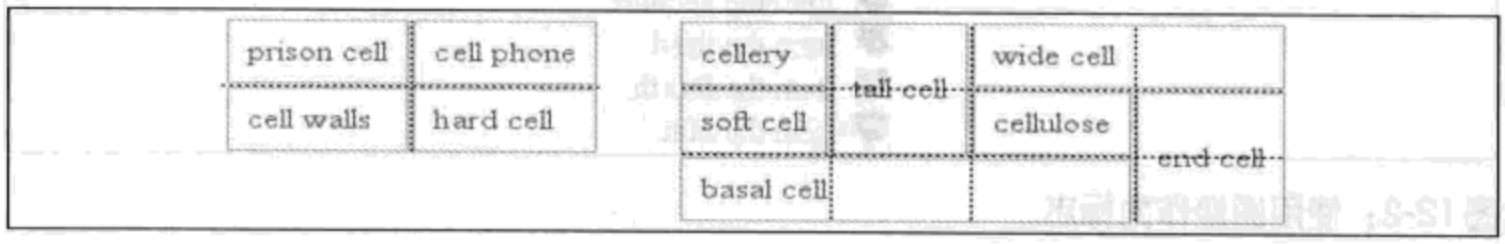

图12-1：切换列表项标志

list-style-type属性可以继承，所以如果希望嵌套列表中使用不同样式的标志，可能需要单独定义。也许还必须显式地声明嵌套列表的样式，因为用户代理的样式表可能已经有定义。例如，假设用户代理定义了以下样式：

```
ul {list-style-type: disc;}
ul ul {list-style-type: circle;}
ul ul ul {list-style-type: square;}
```

如果是这样（而且很可能是这样），就必须声明你自己的样式来覆盖用户代理的样式，仅仅使用继承是不够的。

列表项图像

有时，常规的标志还不够。你可能想对各标志使用一个图像，这可以利用list-style-image属性做到。

下面介绍这个属性如何使用：

```
ul li {list-style-image: url(ohio.gif);}
```

list-style-image	
值：	<uri> | none | inherit
初始值：	none
应用于：	display值为list-item的元素
继承性：	有
计算值：	对于<uri>值，为绝对URI；否则，为none

不错，就这么简单。只要一个简单的url()值，就可以使用图像作为标志，如图12-2所示。

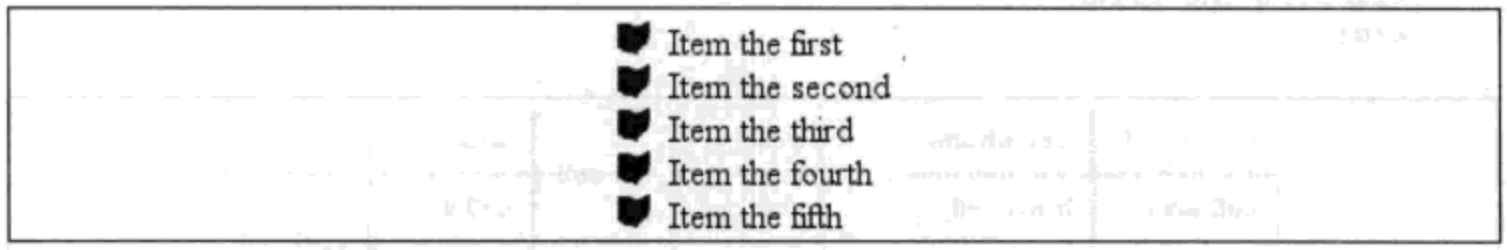

图12-2：使用图像作为标志

当然，对于使用的图像要非常小心，从图12-3所示的例子可以清楚地看出这一点：

```
ul li {list-style-image: url(big-ohio.gif);}
```

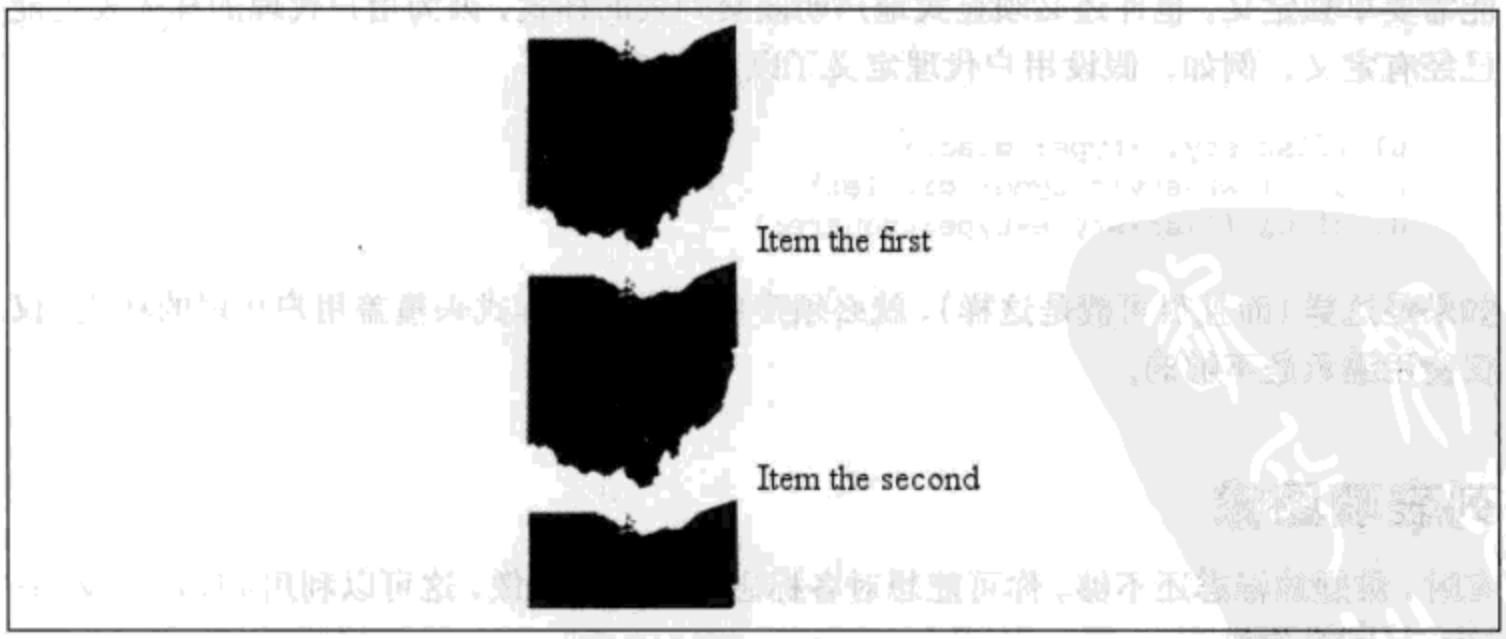

图12-3：使用非常大的图像作为标志

通常可以提供一个作为“后路”的标志类型以应付意外情况，如图像未能加载、被破坏

或者是某种用户代理无法显示的格式等等，这往往是一个好主意。为此可以为列表定义一个后备的 list-style-type：

```
ul li {list-style-image: url(ohio.png); list-style-type: square;}
```

使用 list-style-image 时，还可以将其设置为默认值 none。这是一个很好的实践做法，因为 list-style-image 是可以继承的，所以所有嵌套列表都会继续使用该图像作为标志，除非你采取措施避免这种情况发生：

```
ul {list-style-image: url(ohio.gif); list-style-type: square;}
ul ul {list-style-image: none;}
```

由于嵌套列表继承了列表项类型 square，但是已经设置为不使用图像作为标志，所以嵌套列表会使用方块标志，如图 12-4 所示。

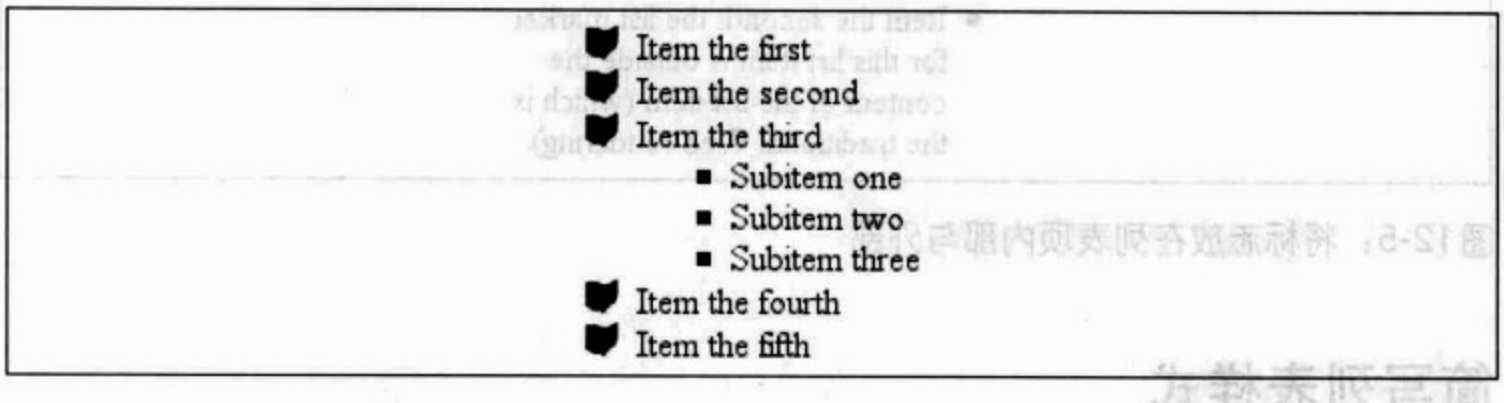

图12-4：切换子表中的图像标志

警告： 要记住，真实世界中可能不会出现这种情况：用户代理有可能已经为 ul ul 定义了一个 list-style-type，所以根本不会继承值 square。你的浏览器可能有不同的做法。

列表标志位置

CSS2.1中还可以做一件事来影响列表项的外观，即确定标志出现在列表项内容之外还是在内容内部。这是利用 list-style-position 完成的。

如果标志的位置设置为 outside（默认值），则显示与一般 Web 上列表项的显示无二。不过，如果你需要一个稍微不同的外观，可以把这个值设置为 inside，将标志拉向内容。这会使标志放在列表项的内容“以内”。对于具体如何做没有明确定义，不过图 12-5 显示了一种可能的做法：

```
li.first {list-style-position: inside;}
li.second {list-style-position: outside;}
```

list-style-position

值：	inside | outside | inherit
初始值：	outside
应用于：	display值为list-item的元素
继承性：	有
计算值：	根据指定确定

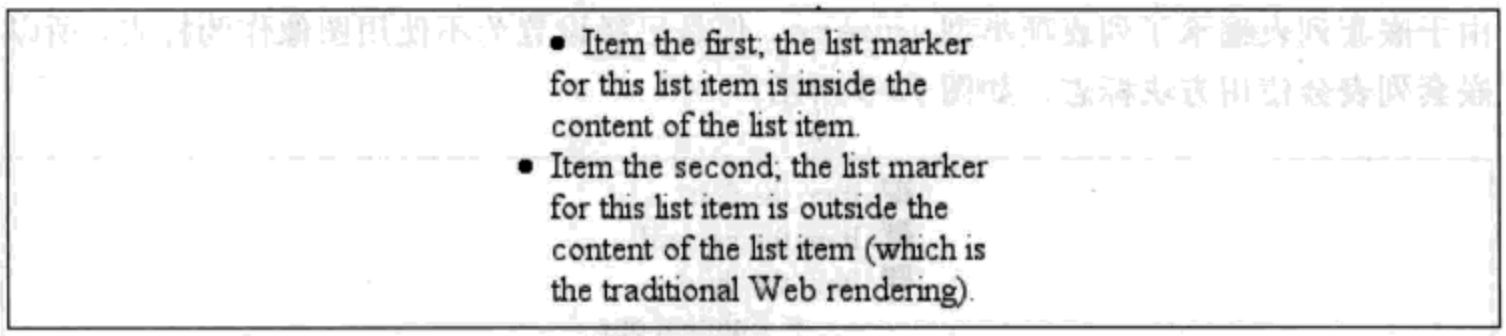

图12-5：将标志放在列表项内部与外部

简写列表样式

为简单起见，可以将以上3个列表样式属性合并为一个方便的属性：list-style。

list-style

值：	[<list-style-type> || <list-style-image> || <list-style-position>] | inherit
初始值：	相对于各个属性。
应用于：	display值为list-item的元素
继承性：	有
计算值：	见单个属性

例如：

```
li {list-style: url(ohio.gif) square inside;}
```

在图12-6中可以看到，这3个值都已经应用到列表项：

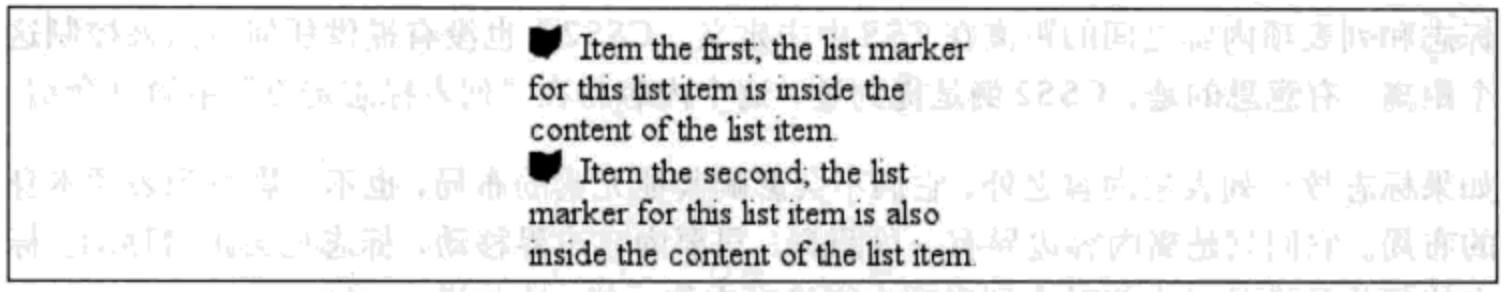

图12-6：集中应用列表样式

list-style的值可以按任何顺序列出，而且这些值都可以忽略。只要提供了一个值，其他的就会填入其默认值。例如，以下两个规则有同样的视觉效果：

```
li.norm {list-style: url(img42.gif);}
li.odd {list-style: url(img42.gif) disc outside;} /* the same thing */
```

还可以用同样的方法覆盖之前的规则。例如：

```
li {list-style-type: square;}
li.norm {list-style: url(img42.gif);}
li.odd {list-style: url(img42.gif) disc outside;} /* the same thing */
```

其结果与图12-6所示相同，因为规则li.norm隐含的list-style-type值disc会覆盖先前声明的square值，同样地，li.odd规则中显式声明的值disc也会覆盖先前声明的square值。

列表布局

前面已经了解了应用列表标志样式的基本知识，下面来考虑不同浏览器中列表如何布局。首先来看3个没有任何标志而且尚未放到列表中的列表项，如图12-7所示。

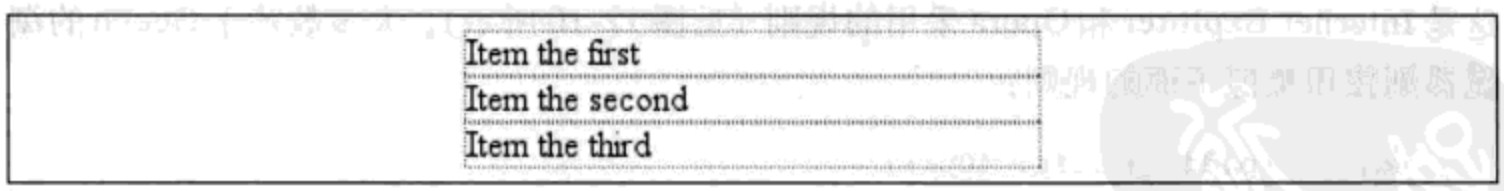

图12-7：3个列表项

从列表项周围的边框可以看出，它们就像是块级元素。实际上，值list-item确实定义为生成块框。下面增加标志，如图12-8所示。

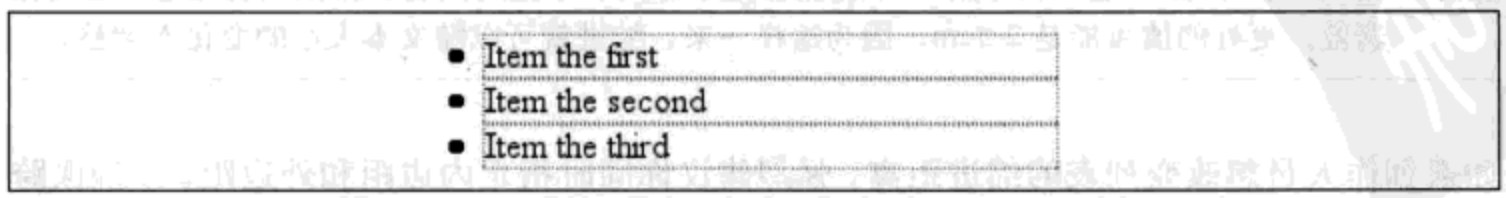

图12-8：增加标志

标志和列表项内容之间的距离在CSS中未定义，CSS2.1也没有提供任何方法来控制这个距离。有意思的是，CSS2倒是提到过，这个内容将在“列表标志定位”中简单介绍。

如果标志放在列表项内容之外，它们不会影响其他元素的布局，也不会影响列表项本身的布局。它们只是离内容边界有一段距离，只要内容边界移动，标志也会跟着移动。标志的行为就像是标志相对于列表项内容绝对定位一样，比如说`position: absolute; left: -1.5em;`。如果标志在列表项内容内部，则相当于放在内容开始处的一个行内元素。

迄今为止，还没有增加具体的列表容器；换句话说，图中没有`ul`也没有`ol`元素。可以增加这样一个容器，如图12-9所示（用一个虚线边框表示）。

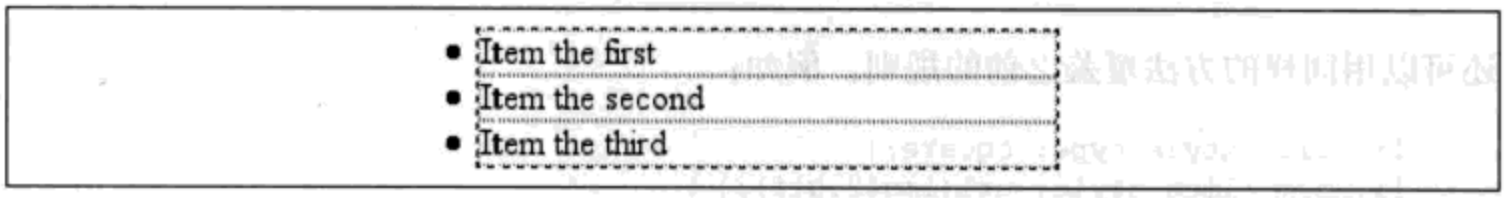

图12-9：增加列表元素

类似于列表项，列表元素也是一个块框，其中包含所有后代元素。不过，可以看到，标志不仅放在列表项内容之外，也放在列表元素的内容区外。在此并没有指定列表通常有的“缩进”。

在写作本书时，大多数浏览器会通过设置列表元素的内边距或外边距来完成列表项的缩进。例如，用户代理可能应用以下规则：

```
ul, ol {margin-left: 40px;}
```

这是Internet Explorer和Opera采用的规则（见图12-10所示）。大多数基于Gecko的浏览器则使用类似下面的规则：

```
ul, ol {padding-left: 40px;}
```

不能说这两种做法不正确，但是如果你想消除列表项的缩进，二者的差异会导致一些问题。图12-10显示了这两种方法的不同。

注意：这里40px的距离是从较早的Web浏览器继承来的，早先的浏览器会把列表缩进一定的像素数。更好的值可能是2.5em，因为这样一来，缩进就可以随文本大小的变化而调整。

如果创作人员想改变列表的缩进距离，强烈建议你同时指定内边距和外边距，以确保跨浏览器兼容性。例如，如果想使用内边距缩进一个列表，可以使用以下规则：

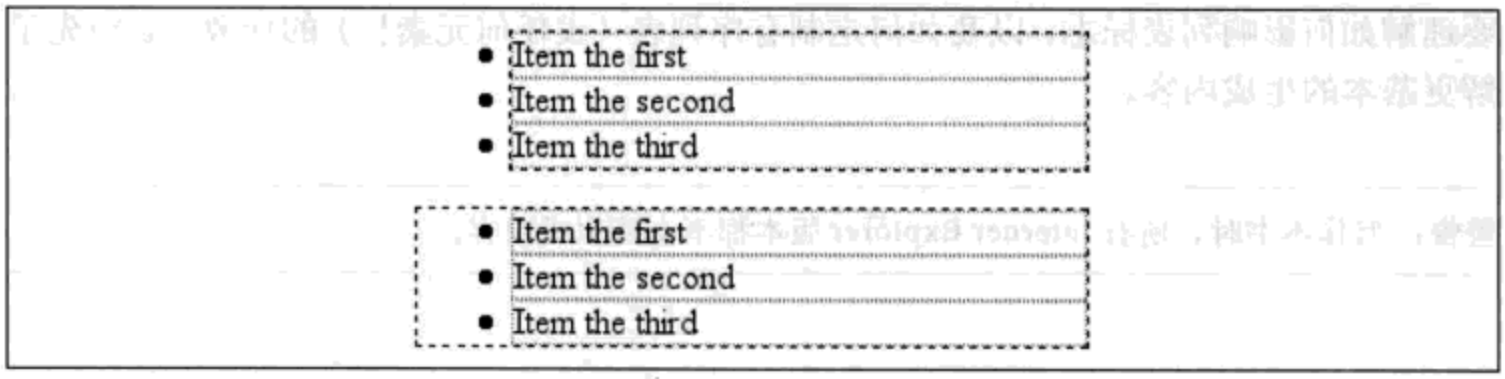

图12-10：使用外边距和内边距实现缩进

```
ul {margin-left: 0; padding-left: 1em;}
```

如果想用外边距，可以写作：

```
ul {margin-left: 1em; padding-left: 0;}
```

不论哪一种情况，要记住标志总会相对于列表项的内容放置，因此可能会“挂”在文档主文本之外，甚至超出浏览器容器边界。

列表标志定位

许多创作人员都希望有这样一个特性，能够控制标志与列表项内容之间的间隔。CSS2定义了一些办法来做到这一点，这包括一个名为marker-offset的属性和一个display值marker。从具体的实现来看，这是一个很笨拙的办法，所以这些特性已经从CSS2.1中去除了。

写作本书时，CSS3列表模型的当前工作草案定义了一个更简洁的新方法来影响标志位置，即::marker伪元素。假设这个模块在成为完备推荐之前没有改动，有朝一日你可能可以编写类似li::marker {margin-right: 0.125em;}的规则，使标志与列表项内容之间有适当的距离，而不用把标志放在内容内部。

生成内容

CSS2和CSS2.1包含一个称为生成内容（generated content）的新特性。这是指由浏览器创建的内容，而不是由标志或内容来表示。

例如，列表标志就是生成内容。在列表项的标记中，没有任何部分直接表示这些标志，而且作为创作人员，你不必在文档内容中写上标志。浏览器会自动生成合适的标志。对于无序列表，标志是某种圆点，如空心圆、实心圆或方块。对于有序列表，标志则是一个计数器，对每个后续列表项不断增1。

要理解如何影响列表标志，以及如何定制有序列表（或任何元素！）的计数，必须先了解更基本的生成内容。

警告： 写作本书时，所有 Internet Explorer 版本都不支持生成内容。

插入生成内容

为了向文档中插入生成内容，可以使用:before和:after伪元素。这些伪元素会根据content属性把生成内容放在一个元素内容的前面或后面（content属性见下一节的介绍）。

例如，你可能希望所有超链接前面增加前缀文本“(link)”加以标志，从而在打印时更明显。可以利用以下规则来做到，其效果如图12-11所示：

```
a[href]:before {content: "(link)";}
```

(link)Jeffrey seems to be (link)very happy about (link)something, although I can't quite work out whether his happiness is over (link)OS X, (link)Chimera, the ability to run the Dock and (link)DragThing at the same time, the latter half of my (link)journal entry from yesterday, or (link)something else entirely.

图12-11：生成文本内容

注意，生成内容和元素内容之间没有空格。这是因为前例中content的值未包含空格。可以将这个声明修改如下，确保生成内容和实际内容之间有一个空格：

```
a[href]:before {content: "(link) ";}
```

这个差别很小，但很重要。

采用类似方式，还可以在指向PDF文档的链接最后插入一个小图标。为此，规则可能如下：

```
a.pdf-doc:after {content: url(pdf-doc-icon.gif);}
```

假设希望进一步设置这些链接的样式，再增加一个边框。可以利用以下第二个规则完成：

```
a.pdf-doc {border: 1px solid gray;}
```

这两个规则的结果见图12-12所示。

Jeffrey seems to be very happy about something, although I can't quite work out whether his happiness is over OS X, Chimera, the ability to run the Dock and DragThing at the same time, the latter half of my journal entry from yesterday, or something else entirely (link).

图12-12：生成图标

类似于图 12-11 中链接下划线已经延伸到“(link)”文本的下面，可以注意到这里链接的边框已经扩展到包围了生成内容。之所以会这样，是因为生成内容放在元素框的内部。在 CSS2.1 中，除了列表标志，无法把生成内容放在元素框之外。

你可能认为定位能解决问题，不过 CSS2 和 CSS2.1 明确地禁止浮动或定位 :before 和 :after 内容，还禁止使用列表样式属性以及表属性。另外还有以下限制：

- 如果 :before 或 :after 选择器的主体是块级元素，则 display 属性只接受值 none、inline、block 和 marker。其他值都处理为 block。
- 如果 :before 或 :after 选择器的主体是一个行内元素，属性 display 只能接受值 none 和 inline。所有其他值都处理为 inline。

例如，考虑以下规则：

```
em:after {content: " (!) "; display: block;}
```

由于 em 是一个行内元素，生成内容不能是块级内容。因此，值 block 重置为 inline。不过，在下一个例子中，生成内容被置为块级内容，因为目标元素就是块级元素：

```
h1:before {content: "New Section"; display: block; color: gray;}
```

其结果见图 12-13。

New Section

The Secret Life of Salmon

图12-13：生成块级内容

生成内容有一个很有意思的方面，它由与之关联的元素继承值。因此，给定以下规则，生成文本将是绿色，与段落内容的颜色相同：

```
p {color: green;}
p:before {content: "::: ";}
```

如果希望生成文本是紫色，只需一个简单的声明：

```
p:before {content: ":::: "; color: purple;}
```

当然，这种值继承只适用于可继承的属性。特别指出这一点是因为这会影响达到某些效果的方式。请考虑以下规则：

```
h1 {border-top: 3px solid black; padding-top: 0.25em;}
h1:before {content: "New Section"; display: block; color: gray;
  border-bottom: 1px dotted black; margin-bottom: 0.5em;}
```

由于生成内容放在h1的元素框内部，它会放在元素上边框的下面，而且在所有内边距以内，如图12-14所示。

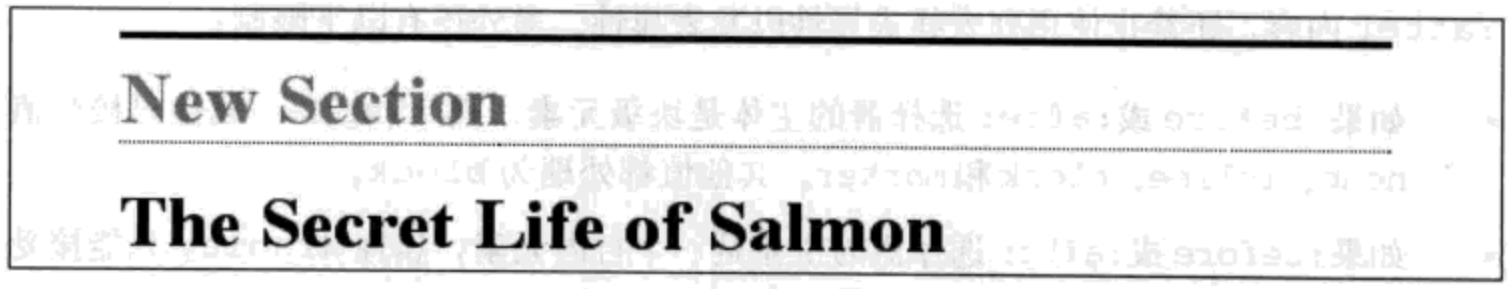

图12-14：考虑生成内容的放置

生成内容（作为块级内容）的下外边距将元素的实际内容向下推了0.5em。不论怎样讲，此例中生成内容的效果就是把h1元素分成了两部分：生成内容框和实际内容框。这是因为生成内容声明为display: block。如果将其修改为display: inline，效果则如图12-15所示：

```
h1 {border-top: 3px solid black; padding-top: 0.25em;}
h1:before {content: "New Section"; display: inline; color: gray;
  border-bottom: 1px dotted black; margin-bottom: 0.5em;}
```

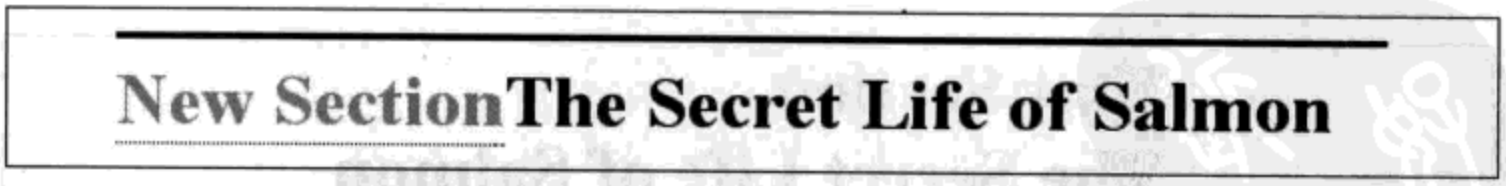

图12-15：将生成内容改为行内内容

注意边框如何放置，还要注意上内边距仍保留。生成内容的下外边距也保留，不过由于生成内容现在是行内内容，而行内元素的外边距不影响行高，所以视觉上外边距不起作用。

有了关于生成内容的基础知识后，下面进一步讨论如何指定具体的生成内容。

指定内容

要生成内容，需要有一种办法来描述生成的内容。前面已经看到，这是利用content属性处理的，不过关于这个属性还有更多内容，前面了解的还远远不够。

content

值：	normal | [<string> | <uri> | <counter> | attr(<identifier>) | open-quote | close-quote | no-open-quote | no-close-quote]+ | inherit
初始值：	normal
应用于：	:before和:after伪元素
继承性：	无
计算值：	对于<uri>值，为绝对URI；对于属性引用，则为结果串；否则，根据指定确定

前面已经看到了串和URI值的使用，计数器将在本章后面介绍。在介绍attr()和quote值之前，我们将更详细地讨论串和URI。

串值会原样显示，即使其中包含某种标记也不例外。因此，以下规则会原样插入到文档中，如图12-16所示：

```
h2:before {content: "<em>&para;</em> "; color: gray;}
```

¶ Spawning

图12-16：串原样显示

这说明，如果你希望生成内容中有一个换行（回车），不能直接使用
，而要使用串\A，这是CSS表示换行的方法（由Unicode换行符得来，其十六进制位置是A）。相反，如果有一个很长的串，需要把它分成多行则要用\符号对换行符转义。以下规则对此做了描述，结果见图12-17所示：

```
h2:before {content: "We insert this text before all H2 elements because \
it is a good idea to show how these things work. It may be a bit long \
but the point should be clearly made. "; color: gray;}
```

还可以使用转义来指示十六进制Unicode值，如\00AB。

We insert this text before all H2 elements because it is a good idea to show how these things work. It may be a bit long but the point should be clearly made. **Spawning**

图12-17：插入和去除换行

警告： 写作本书时，对插入转义内容（如\A和\00AB）的支持并不多，甚至支持生成内容的那些浏览器也很少支持插入转义内容。

利用URI值，只需指向一个外部资源（一个图像、视频、声音剪辑或用户代理支持的其他任何资源），然后插入到文档中适当的位置。如果用户代理出于某种原因不支持所指定的资源，例如你想向浏览器插入一个SVG图像，但这个浏览器无法识别SVG，或者向一个要打印的文档中插入视频，此时就要求用户代理完全忽略这个资源，不插入任何内容。

插入属性值

有些情况下，你可能想取一个元素的属性值，使之作为文档显示的一部分。举一个简单的例子，可以把每个链接的href属性值直接放在链接的后面，如下：

```
a[href]:after {content: attr(href);}
```

这也会导致生成内容与具体内容冲突的问题。为解决这个问题，可以向声明增加一些串值，其结果如图12-18所示：

```
a[href]:after {content: " [" attr(href) "]";}
```

Jeffrey [http://www.zeldman.com/] seems to be very happy [http://www.zeldman.com/daily/1202b.shtml#joy] about something [http://www.zeldman.com/i/accessories/worthit.jpg], although I can't quite work out whether his happiness is over OS X [http://www.apple.com/macosx/], Chimera [http://chimera.mozdev.org/], the ability to run the Dock and DragThing [http://www.dragthing.com/] at the same time, the latter half of my journal entry from yesterday [http://www.meyerweb.com/eric/thoughts/2002b.html#t20021227], or something else entirely [http://www.roguelibrarian.com/].

图12-18：插入URL

例如，这对于打印样式表可能很有用。所有属性值都可以作为生成内容插入：alt文本、class或id值，以及任何属性。创作人员可能会明确显示块引用的引用信息，如下：

```
blockquote:after {content: "(" attr(cite) ")"; display: block;
  text-align: right; font-style: italic;}
```

对此，利用更复杂的规则还可以显示一个法律文档的文本和链接颜色值：

```
body:before {content: "Text: " attr(text) " | Link: " attr(link)
  " | Visited: " attr(vlink) " | Active: " attr(alink);
  display: block; padding: 0.33em;
  border: 1px solid black; text-align: center;}
```

注意，如果一个属性不存在，会在相应位置插入一个空串。正如图 12-19 所示，在此向一个文档应用以上示例规则，该文档中 body 元素没有 alink 属性。

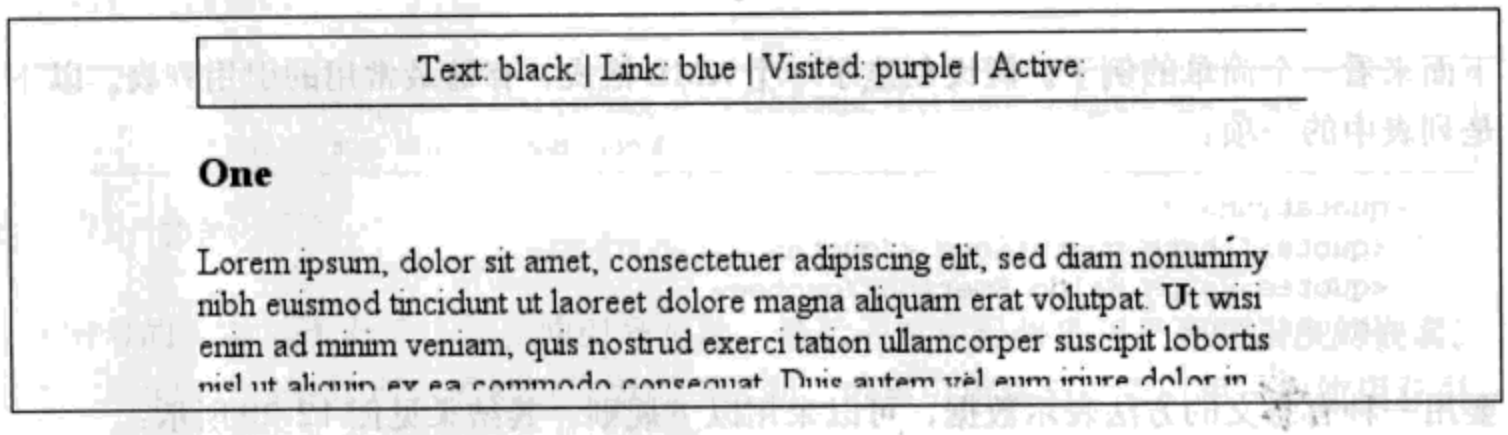

图12-19：没有的属性会被忽略

可以看到，文本“Active:”（包括后面的空格）确实插入到了文档中，不过后面什么也没有。如果只想在属性存在时才插入该属性的值，这就很方便。

警告： CSS2.x定义属性引用的返回值是未解析的串。因此，如果一个属性的值包含标记或字符实体，会原样显示。

生成引号

生成内容有一种特殊形式，即引号，CSS2.x提供了一种有效的方式来管理引号及其嵌套行为。由于提供了 open-quote 等成对的content值以及属性 quotes，使得生成引号的管理成为可能。

quotes

值：	[<string> <string>]+ | none | inherit
初始值：	取决于具体的用户代理
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

研究值的语法时可以发现，除了关键字none和inherit外，唯一合法的值是一对或多对串。在一对串中，前一个串定义了开始引号（open-quote），第二个串定义了结束引号（close-quote）。因此，以下两个声明中仅第一个合法：

```
quotes: '"' "'";  /* valid */
quotes: '"';  /* NOT VALID */
```

串本身是引号时，也可以用引号将其包围，第一个规则就展示了这样一种方法。双引号可以由单引号包围，反之亦然。

下面来看一个简单的例子。假设创建了一个XML格式，存储最常用的引用列表。以下是列表中的一项：

```
<quotation>
 <quote>I hate quotations.</quote>
 <quotee>Ralph Waldo Emerson</quotee>
</quotation>
```

要用一种有意义的方法表示数据，可以采用以下规则，其结果见图12-20所示：

```
quotation: {display: block;}
quote {quotes: '"' '"';}
quote:before {content: open-quote;}
quote:after {content: close-quote;}
quotee:before {content: " (";}
quotee:after {content: ")";}
```

"I hate quotations." (Ralph Waldo Emerson)

图12-20：插入引号和其他内容

值open-quote和close-quote用于插入合适的引号符号（因为不同语言有不同的引号）。它们使用quotes值来确定如何工作。因此，quotation以一个双引号开始，并以一个双引号结束。

如果想使用“弯引号”而不是普通的直引号（这在大多数打印媒体中很常见），quote规则要写作：

```
quote {quotes: '\201C' '\201D';}
```

这里使用了“弯引号”的十六进制Unicode值，如果应用到前面的quotation，Emerson的引用会包围在弯引号中，而不像图12-20中所示那样包围在直引号中。

利用quotes，只要愿意，可以定义任意多层嵌套引用模式。例如，在英语中，一种常用的做法是先以双引号开头，在其中嵌套的引用使用单引号。通过以下规则，就可以利用“弯引号”做到这一点：

```
quotation: display: block;}
quote {quotes: '\201C' '\201D' '\2018' '\2019';}
quote:before, q:before{content: open-quote;}
quote:after, q:after {content: close-quote;}
```

应用到以下标记时，这些规则的效果如图 12-21 所示：

```
<quotation>
  <quote> In the beginning, there was nothing. And God said: <q>Let there
  be light!</q> And there was still nothing, but you could see it.</quote>
</quotation>
```

> "In the beginning, there was nothing. And God said: 'Let there be light!' And there was still nothing, but you could see it."

图12-21：嵌套弯引号

如果引号的嵌套层次大于已定义的引号对数，最后一对引号将重用于更深层次的嵌套。因此，如果向图 12-21 所示的标记应用以下规则，内层引用将与外层引用一样使用双引号：

```
quote {quotes: '\201C' '\201D';}
```

利用生成引号，还能实现另外一种常见的排版效果。如果有多段引用文本，通常会忽略每一段的结束引号（close-quote），而只显示开始引号，只是最后一段例外。可以使用no-close-quote值达到这个效果：

```
blockquote {quotes: '"' '"' "'" "'" '"' '"';}
blockquote p:before   {content: open-quote;}
blockquote p:after  {content: no-close-quote;}
```

利用这个规则，每一段的开始处会有一个双引号，不过没有结束引号。最后一段也是如此，所以如果想为最后一段增加一个结束引号，需要为最后一段指定类（class），并为其:after 内容声明一个 close-quote。

这个值很重要，因为这样可以使引用嵌套层次递减，而不必真正生成一个符号。正因如此，在第三段之前，每个段落都以一个双引号开始，而不是交替使用双引号和单引号。no-close-quote在各段的最后结束了引用嵌套，因此每一段都从相同的嵌套层次开始。

这很重要，正如CSS2.1 规范所指出的，“引用深度不依赖于源文档或格式化结构的嵌套”。换句话说，开始一个引用层次时，所有元素都有相同的嵌套层次，直到遇到一个close-quote，此时引用嵌套层次减 1。

为保证完备，还有一个no-open-quote关键字，其效果与no-close-quote对称。这个关键字会让引用嵌套层次增 1，但不生成符号。

计数器

我们都已经对计数器很熟悉了；例如，有序列表中的列表项标志就是计数器。在CSS1中，没有办法影响这些计数器很大程度上是因为没有这个必要：HTML为有序列表定义了自己的计数行为，这就足够了。随着XML的出现，现在需要提供一种定义计数器的方法，这很重要。不过，CSS2没有满足于只是提供HTML中的简单计数。它增加了两个属性和两个content值，从而可以定义几乎所有计数格式，包括采用多种样式的小节计数器，如“VII.2.c”。

重置和递增

创建计数器的基础包括两个方面，一是能设置计数器的起点，二是能将其递增一定的量。前者由属性counter-reset处理。

counter-reset

值：	[<identifier> <integer>?]+ | none | inherit
初始值：	取决于具体的用户代理
应用于：	所有元素
继承性：	无
计算值：	根据指定确定

计数器标识符只是创作人员创建的一个标签。例如，可以将小节计数器命名为subsection、subsec、ss甚至bob。只要重置（或递增）标识符，就足以使之建立。在以下规则中，计数器chapter就在重置时定义：

```
h1 {counter-reset: chapter;}
```

默认地，计数器重置为0。如果想重置为另一个数，可以在标识符后面声明这个数：

```
h1#ch4 {counter-reset: chapter 4;}
```

还可以在标识符－整数对中一次重置多个标识符。如果少了一个整数，则默认为0：

```
h1 {counter-reset: chapter 4 section -1 subsec figure 1;}
   /* 'subsec' is reset to 0 */
```

从上例可以看到，负值是允许的。将计数器设置为-32768并由此递增是完全合法的。

注意： 如果采用非数值计数样式，CSS没有定义用户代理对于负计数器值该如何处理。例如，如果一个计数器的值是-5，但是其显示样式是upper-alpha，此时该采取什么行为？对此就没有定义。

总之，需要一个属性来指示元素将计数器递增。否则，计数器将永远保持计数器重置声明中指定的值。毫无疑问，所需的这个属性就是counter-increment。

counter-increment	
值：	[<identifier> <integer>?]+ | none | inherit
初始值：	取决于具体的用户代理
应用于：	所有元素
继承性：	无
计算值：	根据指定确定

类似于counter-reset，counter-increment也接受标识符－整数对，其中整数部分可以是0，也可以是负数或正数。与counter-reset的区别在于，如果counter-increment中的标识符－整数对少了一个整数，则默认为1而不是0。

举例来说，以下显示了用户代理如何定义计数器生成有序列表传统的1、2、3计数：

```
ol {counter-reset: ordered;}  /* defaults to 0 */
ol li {counter-increment: ordered;}  /* defaults to 1 */
```

另一方面，创作人员可能希望从0向下计，使列表项使用一种渐进的负数体系。为此只需稍作修改：

```
ol {counter-reset: ordered;}  /* defaults to 0 */
ol li {counter-increment: ordered -1;}
```

列表的计数将是-1、-2、-3等等。如果将整数-1换成-2，列表的计数则是-2、-4、-6等等。

使用计数器

不过，要具体显示计数器，还需要结合使用content属性和一个与计数器有关的值。要了解这是如何做到的，下面以一个基于XML的有序列表为例，如下：

```
<list type="ordered">
 <item>First item</item>
 <item>Item two</item>
 <item>The third item</item>
</list>
```

向采用此结构的XML应用以下规则，可以得到如图12-22所示的结果：

```
list[type="ordered"] {counter-reset: ordered;}  /* defaults to 0 */
list[type="ordered"] item {display: block;}
list[type="ordered"] item:before {counter-increment: ordered;
 content: counter(ordered) ". "; margin: 0.25em 0;}
```

1. First item
2. Item two
3. The third item

图12-22：对列表项计数

注意，与平常一样，生成内容作为行内内容放在相关元素的开始位置。因此，其效果类似于声明了 `list-style-position: inside;` 的HTML列表。

还要注意，item元素是生成块级框的普通元素，这说明计数器并不仅限于display为list-item的元素。实际上，任何元素都可以利用计数器。考虑以下规则：

```
h1:before {counter-reset: section subsec;
  counter-increment: chapter;
  content: counter(chapter) ". ";}
h2:before {counter-reset: subsec;
  counter-increment: section;
  content: counter(chapter )"." counter(section) ". ";}
h3:before {counter-increment: subsec;
  content: counter(chapter) "." counter(section) "." counter(subsec) ". ";}
```

这些规则的效果见图12-23所示。

图12-23展示了有关计数器重置和递增的几个要点。注意h1元素如何使用计数器chapter，该计数器默认为0，但在元素文本前却显示了一个“1.”。计数器由同一个元素递增和使用时，递增发生在计数器显示之前。类似地，如果计数器在同一个元素中重置和显示，重置也在计数器显示之前发生。考虑以下规则：

```
h1:before, h2:before, h3:before {
  content: counter(chapter) "." counter(section) "." counter(subsec) ". ";}
h1 {counter-reset: section subsec;
  counter-increment: chapter;}
```

1. The Secret Life of Salmon

1.1. Introduction

1.2. Habitats

1.2.1. Ocean

1.2.2. Rivers

1.3. Spawning

1.3.1. Fertilization

1.3.2. Gestation

1.3.3. Hatching

图12-23：为标题增加计数器

文档中第一个h1元素前面有文本“1.0.0. ”，因为计数器section和subsec都重置，但没有递增。这意味着如果希望一个递增计数器第一次显示0，只需将该计数器重置为-1，如下：

```
body {counter-reset: chapter -1;}
h1:before {counter-increment: chapter; content: counter(chapter) ". ";}
```

对计数器还可以做一些有意思的事情。考虑以下XML：

```
<code type="BASIC">
 <line>PRINT "Hello world!"</line>
 <line>REM This is what the kids are calling a "comment"</line>
 <line>GOTO 10</line>
</code>
```

可以用以下规则改写BASIC程序清单的传统格式：

```
code[type="BASIC"] {counter-reset: linenum; font-family: monospace;}
code[type="BASIC"] line {display: block;}
code[type="BASIC"] line:before {counter-increment: linenum;
  content: counter(linenum 10) ": ";}
```

还可以为每个计数器定义一个列表样式，作为counter()格式的一部分。为此可以在计

数器的标识符后面增加一个list-style-type关键字，用逗号分隔。以下修改了标题计数器示例，其结果见图12-24所示：

```
h1:before {counter-reset: section subsec;
  counter-increment: chapter;
  content: counter(chapter,upper-alpha) ". ";}
h2:before {counter-reset: subsec;
  counter-increment: section;
  content: counter(chapter,upper-alpha)"." counter(section) ". ";}
h3:before {counter-increment: subsec;
  content: counter(chapter,upper-alpha) "." counter(section) "."
  counter(subsec,lower-roman) ". ";}
```

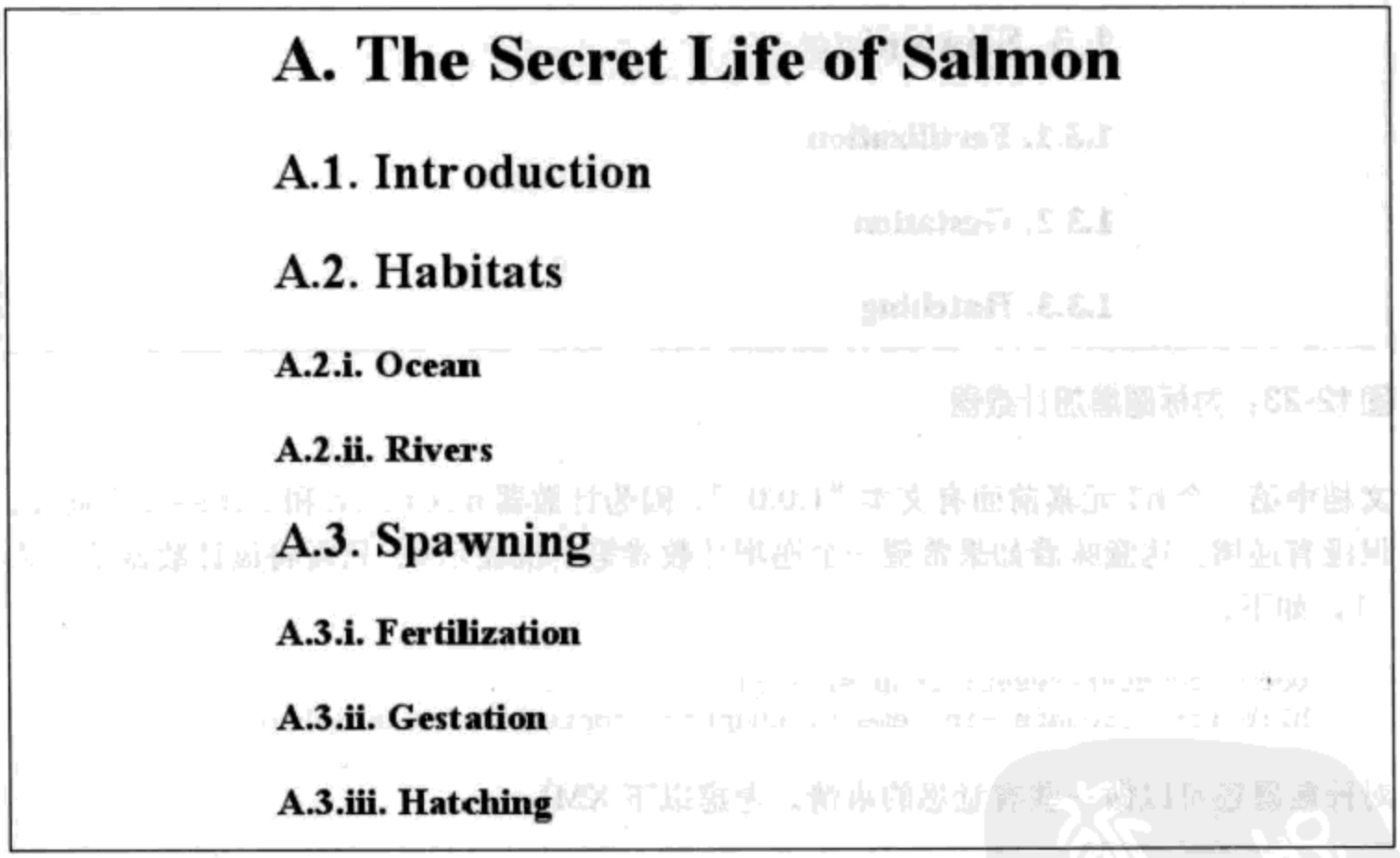

图12-24：改变计数器样式

注意，没有为计数器section指定样式关键字，所以它默认为decimal计数样式。如果愿意，甚至可以将计数器设置为使用样式disc、circle、square和none。

有意思的是，即使规则看上去会让计数器递增，但实际上display为none的元素并不会递增计数器。相反，visibility为hidden的元素确实会递增计数器：

```
.suppress {counter-increment: cntr; display: none;}
  /* 'cntr' is NOT incremented */
.invisible {counter-increment: cntr; visibility: hidden;}
  /* 'cntr' IS incremented */
```

计数器和作用域

至此，我们已经了解了如何把多个计数器串在一起创建一种多级计数。创作人员通常还需要对嵌套有序列表这么处理，不过为了达到很深的嵌套层次，需要创建足够多的计数器，这种做法很快会变得相当笨拙。仅仅是建立5层嵌套列表就需要一大堆的规则，如下：

```
ol ol ol ol ol li:before {counter-increment: ord1 ord2 ord3 ord4 ord5;
  content: counter(ord1) "." counter(ord2) "." counter(ord3) "."
   counter(ord4) "." counter(ord5) ".";}
```

想想看，要建立50层嵌套需要写多少规则！（这并不是说你应当建立50层的嵌套有序列表，即不表示这是合理的，而只是暂且举个例子。）

好在CSS2.x描述了计数器的作用域（scope）概念。简单地说，每层嵌套都会为给定计数器创建一个新的作用域。正是因为有作用域，以下规则才能以常规HTML方式实现嵌套表计数：

```
ol {counter-reset: ordered;}
ol li:before {counter-increment: ordered;
 content: counter(ordered) ". ";}
```

这些规则会使有序列表（甚至嵌套在其他列表中的有序列表）从1开始计数，并且逐项增1，这正是HTML一直以来的做法。

之所以能做到这一点，是因为每层嵌套都为计数器ordered创建了一个新实例。所以，对于第一个有序列表，会创建ordered的一个实例。然后，对于嵌套在第一个列表中的各个列表，又会创建另一个新实例，因此对于每个列表计数总是从头开始。

不过，如果你希望有序列表这样计数，使每层嵌套都创建一个新计数器追加到老计数器上，如：1、1.1、1.2、1.2.1、1.2.2、1.3、2、2.1等。利用counter( )是办不到的，不过可以用counters( )实现。这里的区别就在于一个“s”（counters( )而不是counter( )）。

要创建如图12-25所示的嵌套计数器样式，需要以下规则：

```
ol {counter-reset: ordered;}
ol li:before {counter-increment: ordered;
 content: counters(ordered,".") " - ";}
```

基本说来，关键字counters(ordered,".")会显示各作用域的ordered计数器，并追加一个点号，然后把对应一个给定元素的所有作用域计数器串起来。因此，一个3层嵌套列表中的列表项就会有这样的前缀：最外层列表作用域的ordered值、中间层列

1. - Lists
    1.1. - Types of Lists
    1.2. - List Item Images
    1.3. - List Marker Positions
    1.4. - List Styles in Shorthand
    1.5. - List Layout
2. - Generated Content
    2.1. - Inserting Generated Content
        2.1.1. - Generated Content and Run-In Content
    2.2. - Specifying Content
        2.2.1. - Inserting Attribute Values
        2.2.2. - Generated Quotes
    2.3. - Counters
        2.3.1. - Resetting and Incrementing
        2.3.2. - Using Counters
        2.3.3. - Counters and Scope
3. - Summary

图12-25：嵌套计数器

表作用域的ordered值（中间层列表是最外层列表和当前列表之间的列表），以及当前列表作用域的ordered值，各个ordered值后面都有一个点号。content值的余下部分（" - "）会在所有这些计数器后面增加一个空格、一个连字号以及另一个空格。

与counter( )类似，可以为嵌套计数器定义一个列表样式，不过所有计数器都应用同样的样式。因此，如果如下修改前面的CSS，图12-25中所示的列表项都会使用小写字母而不是数字作为计数器：

```
ol li:before {counter-increment: ordered;
 content: counters(ordered,".",lower-alpha) ": ";}
```

小结

尽管列表样式没有我们希望得那么复杂，而且浏览器对生成内容的支持还有些欠缺（至少在写作本书的时候是这样），不过能够对列表应用样式还是很有用的。一个常见的用途是取一个链接列表，去除其标志和缩进来创建一个导航边栏。一方面要保证简单的标记，另一方面要得到灵活的布局，这很难做到。CSS3在列表样式方面预期有一些改进，基于此，我们期待列表将来变得越来越有用。

至于目前，如果一个标记语言本身没有固有的列表元素，生成内容会有很大帮助，例如，可以插入图标之类的内容指向某种类型的链接（PDF文件、Word文档，甚至另一个网站的链接）。利用生成内容，还能很容易地打印链接URL，由于生成内容能插入引号并

完成格式化，这对于排版非常有利，可以得到很好的效果。完全可以这么说：生成内容的用途只受你想象力的限制，只要想得到，生成内容就能做得到。更好的一点是，利用计数器现在还可以向任何元素（而不只是列表）关联序数信息，如标题或代码块。如果你还希望利用设计支持用户界面方面的一些新特性，模仿用户操作系统的外观，那么请继续读下去。下一章将讨论在CSS设计中如何使用系统颜色和字体。

第13章

用户界面样式

CSS的绝大部分都是关于文档的样式，不过它还提供了很多有用的界面样式工具，而不仅仅面向文档。例如，Mozilla开发人员使用一种名为XUL的语言创建Mozilla浏览器（以及许多Mozilla产品）的界面。XUL就利用CSS和类CSS声明来提供导航按钮、边栏页、对话框、状态框以及chrome本身的其他部分。

类似地，可以重用用户默认环境的某些方面来设置文档的字体和颜色样式；甚至可以对焦点强调和鼠标光标施加影响。CSS2的界面功能会让用户有更愉快的体验，不过如果使用不当，也可能造成用户更混乱。

系统字体和颜色

有时你可能希望自己的文档能尽可能地模仿用户的计算环境。如果你在创建基于Web的应用，目标是让Web组件看上去是用户操作系统中的一部分，这就是一个显而易见的例子。尽管CSS2不允许在自己的文档中重用操作系统外观的任何一个方面，不过有一组丰富的颜色和有限的字体可供选择。

系统字体

假设已经创建了一个元素（例如通过JavaScript实现），其功能相当于按钮。可以让这个控件看上去就像是用户计算环境中的一个按钮，从而满足用户对控件外观的期望，相应地使控件更为有用。

为了达到这个目的，只需编写以下规则：

```
a.widget {font: caption;}
```

这会为class为widget的所有元素设置字体，使之与有标题控件（如按钮）中的文本使用同样的字体系列、字体大小、字体加粗、字体风格以及字体变形。

CSS2定义了6个系统字体关键字。分别描述如下：

caption
: 由标题控件使用的字体样式，如按钮和下拉控件。

icon
: 操作系统图标标签所用的字体样式，如硬盘驱动器、文件夹和文件图标。

menu
: 下拉菜单和菜单列表中文本使用的字体样式。

message-box
: 对话框中文本使用的字体样式。

small-caption
: 由标题小控件的标签使用的字体样式。

status-bar
: 窗口状态条中文本使用的字体样式。

有一点很重要，要认识到这些值只能用于font属性，它们本身就是简写形式。例如，假设一个用户的操作系统将图标标签显示为10像素的Geneva字体，而且没有加粗、没有斜体，也没有小型大写字母效果。这意味着以下3个规则都是等价的，其结果如图13-1所示：

```
body {font: icon;}
body {font: 10px Geneva;}
body {
  font-weight: normal;
  font-style: normal;
  font-variant: normal;
  font-size: 10px;
  font-family: Geneva;
}
```

因此像icon这样一个简单值实际上包含了很多其他的值。这在CSS中很特别，所以使用这些值比平常要稍微复杂一些。

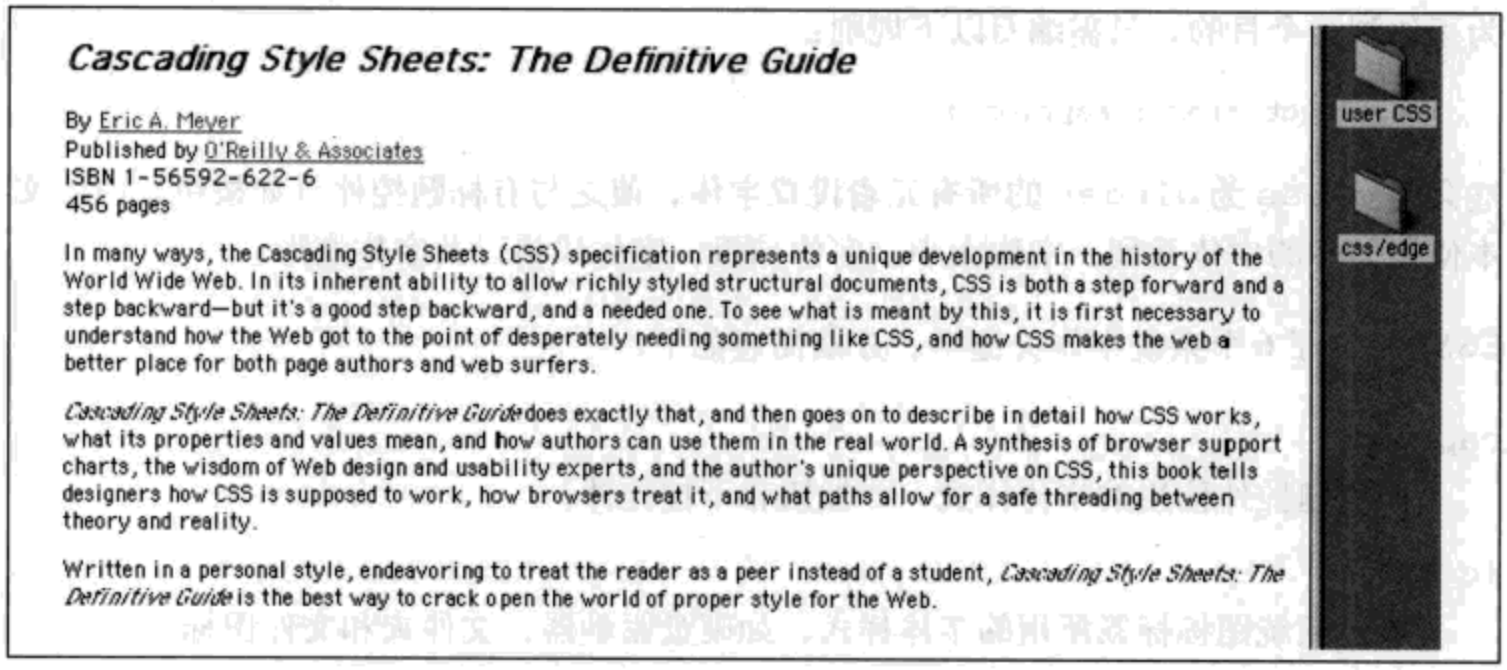

图13-1：使文本看上去像是图标标签

举例来说，假设你希望使用图标标签的字体样式，不过还希望将这个字体加粗，尽管用户系统中的图标标签本身可能不是粗体。为此，需要按以下顺序声明规则：

```
body {font: icon; font-weight: bold;}
```

如果声明按这个顺序写，第一个声明会让用户代理将body元素的字体设置为与图标标签一致，然后利用第二个声明修改该字体的加粗。如果把顺序倒过来，font声明的值会覆盖以上第二个声明的font-weight值（译注1），这就会丢掉加粗声明。简写属性（如font）也是用类似的方式处理。

你可能想知道，为什么这里没有通用字体系列，因为通常都建议创作人员指定字体时要采用类似Geneva, sans-serif;的形式（以防用户浏览器不支持所指定的字体）。CSS不要求“添加”通用字体系列，不过在这种情况下也没有这个必要。如果用户代理想要“抽取”计算环境中显示某种元素所用的字体系列，就绝对能保证这种字体对浏览器一定可用。

如果所需的系统字体样式不可用或者无法确定，用户代理可以在一组近似的字体样式中猜测。例如，可以这样近似small-caption：取caption的字体样式，再缩小字体的大小。如果无法做出这样的猜测，用户代理就应当使用一种“用户代理默认字体”。

译注1： 实际上此时font-weight声明将作为第一个声明。

系统颜色

警告： 写作本书时，CSS3 颜色模块的工作草案废弃了系统颜色关键字，而增加了新的属性 appearance。类似地，CSS2.1 预期到CSS3 中的这些修改，也废弃了这些关键字。强烈建议创作人员不要使用系统颜色，因为在CSS的将来版本中它们很可能不再出现。之所以要强调这一点，原因是当前可用的一些浏览器确实支持系统颜色。

如果你想重用用户操作系统中指定的颜色，CSS2为此定义了一系列系统颜色关键字。只要能用 <color> 值的环境，就可以使用这些值。例如，可以通过以下声明让一个元素的背景与用户的桌面颜色一致：

```
div#test {background-color: Background;}
```

因此，可以如下为文档指定系统的默认文本颜色和背景颜色：

```
body {color: WindowText; background: Window;}
```

通过这种定制，更有可能让用户很好地阅读文档，因为用户应该已经适当地配置了操作系统以支持这些颜色（如果没有，那是他活该！）。

总共有 28 个系统颜色关键字，不过 CSS 没有明确地定义这些关键字。相反，对各个关键字的含义只有一些概括（而且非常简短）的描述。下表描述了所有这28个关键字。如果 Windows 2000 显示控制面板的“外观”页中有选项与之对应，将在以下描述中补充说明。

ActiveBorder

这种颜色应用于活动窗口的外边框（“活动窗口边框”中的第一个颜色）。

ActiveCaption

当前活动窗口标题的背景色（“活动标题栏”中的第一个颜色）。

AppWorkspace

支持多个文档的应用中使用的背景色，例如Microsoft Word中打开文档“后面”的背景色（“应用背景”中的第一个颜色）。

Background

桌面的背景色（“桌面”中的第一个颜色）。

ButtonFace

三维按钮“面”上使用的颜色。

ButtonHighlight

三维显示元素背离虚拟光源的边沿上的亮色。因此，如果虚拟光源位于左上角，这就是显示元素右边界和下边界上使用的亮色。

ButtonShadow

三维显示元素的阴影色。

ButtonText

“按下”按钮上文本的颜色（“3D对象”中的字体颜色）。

CaptionText

标题、大小框中的文本以及滚动箭头框中符号的颜色（“活动标题栏”中的字体颜色）。

GrayText

置灰(禁用)文本。如果当前显示驱动程序不支持纯灰色，这个关键字解释为#000。

Highlight

控件中选中项的颜色（“选中项”中的第一个颜色）。

HighlightText

控件中选中项的文本颜色（“选中项”中的字体颜色）。

InactiveBorder

应用于不活动窗口的外边框的颜色（“不活动窗口边框”中的第一个颜色）。

InactiveCaption

不活动窗口的标题的背景色（“不活动标题栏”的第一个颜色）。

InactiveCaptionText

不活动标题中的文本颜色（“不活动标题栏”的字体颜色）。

InfoBackground

工具提示中的背景色（“工具提示”中的第一个颜色）。

InfoText

工具提示中的文本颜色（“工具提示”中的字体颜色）。

Menu

菜单背景的颜色（“菜单”中的第一个颜色）。

MenuText

菜单中的文本颜色（“菜单”中的字体颜色）。

Scrollbar

滚动条的“灰色区域”。

ThreeDDarkShadow

与三维显示元素的深阴影颜色相同。

ThreeDFace

与三维显示元素的表面颜色相同。

ThreeDHighlight

三维显示元素上的亮色。

ThreeDLightShadow

三维显示元素上的浅色（面向光源边沿上的颜色）。

ThreeDShadow

三维显示元素上的深阴影。

Window

窗口的背景的颜色（“窗口”中的第一个颜色）。

WindowFrame

应用于窗口的框架的颜色。

WindowText

窗口中的文本颜色（“窗口”中的字体颜色）。

CSS2将系统颜色关键字定义为不区分大小写，不过建议还是使用上表所示的混合大小写写法，这样颜色名更可读。可以看到，ThreeDLightShadow乍看上去就比threedlightshadow更容易理解。

系统颜色关键字本质上是含糊的，这有一个明显的缺点，不同的用户代理可能以不同的方式解释这些关键字，即使这些用户代理在同一个操作系统上运行。因此，在使用这些关键字时，不要指望肯定会有一致的结果。例如，要避免这样的文字：“查找颜色与桌面颜色一致的文本。”因为用户可能在默认桌面颜色之上放了一个桌面图像（或“墙纸”）。

光标

用户界面的另一个重要部分是光标（在CSS规范中称之为“指示设备”），它由鼠标、写字板、图形书写板甚至光学读取系统之类的设备控制。在大多数Web浏览器中，光标对于提供交互反馈很有用；例如，光标经过一个超链接时会变成一只食指伸出的小手，这就是一个明显的例子。

改变光标

CSS2允许改变光标图标，这说明创建一个类似于操作系统中桌面应用的Web应用会容易得多。例如，经过一个指向帮助文件的链接时，光标可能会变成一个“帮助”图标（如问号），如图13-2所示。

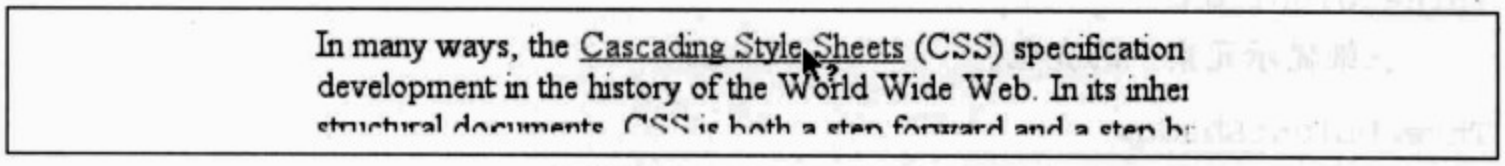

图13-2：改变光标图标

这是利用cursor属性完成的。

cursor

值：	[[<uri>,]* [auto | default | pointer | crosshair | move | e-resize | ne-resize | nw-resize | n-resize | se-resize | sw-resize | s-resize | w-resize| text | wait | help | progress]] | inherit
初始值：	auto
应用于：	所有元素
继承性：	有
计算值：	对于<uri>值，为绝对值；否则，根据指定确定

默认值auto只表示用户代理应当确定最适合当前上下文的光标图标。这与default不同，后者要求图标是操作系统的默认光标。默认光标通常是一个箭头，不过也不一定；这取决于当前的计算环境。

指示和选择光标

值pointer会把光标图标改为与移过超链接时的光标相同。甚至可以为HTML文档描述这种行为：

```
a[href] {cursor: pointer;}
```

利用cursor，可以将任何元素定义为像链接一样改变光标图标。这可能会让用户糊涂，所以建议不要经常这样做。另一方面（可以这么说），利用cursor可以更容易地利用非链接元素创建交互式、脚本驱动的屏幕部件，然后适当地改变图标，如图13-3所示。

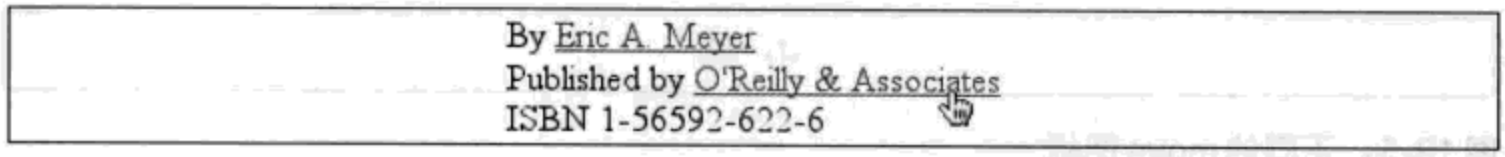

图13-3：指示元素的交互性

注意： Windows平台的Internet Explorer在IE6以前不能识别`pointer`，而是使用值`hand`将光标改变为“指示手”图标。这两个值IE6都能识别。对此常见的建议是按先后顺序使用这两个值，如下：

```
#example {cursor: pointer; cursor: hand;}
```

这并不会验证是否确实这样做，不过这样可以在较新的浏览器和Explorer的较早版本中得到一致的结果。注意，这里的顺序很重要：如果把这两个值的顺序倒过来，就不要指望它还能正常工作。更多详细信息参见 *http://developer.mozilla.org/en/docs/Giving_‘cursor’_a_Hand*。

Web浏览领域中另一个很常见的光标图标是`text`图标，只要用户能选择文本，就会出现这个图标。这往往是一个“I”光标，作为一个视觉提示，指示用户可以拖动－选择光标下的内容。图13-4显示了一段已经选中的文本的最后有一个文本图标。

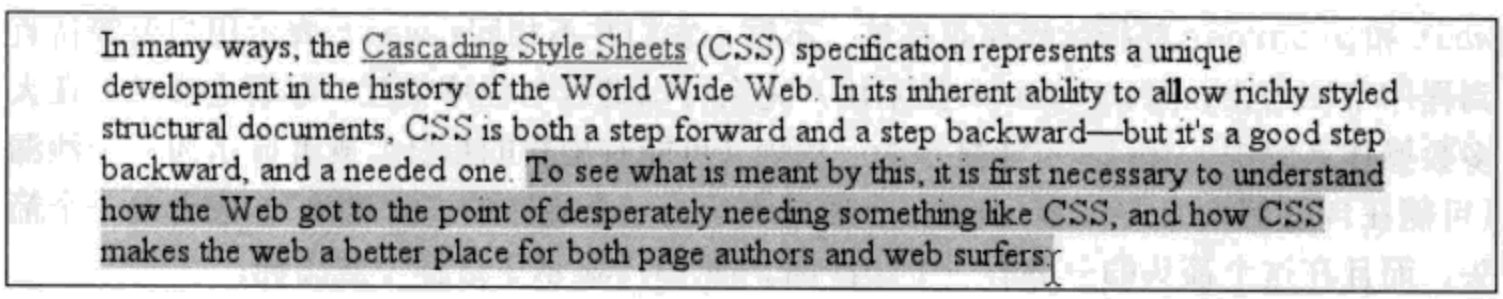

图13-4：可选择文本和文本光标

指示交互性的另一种办法是使用值`crosshair`，显然，crosshair会把光标图标变为一个十字符号。这往往是一对彼此呈直角交叉放置的短线，其中一条垂直，另一条水平，看上去就像一个加号（+）。不过，十字符号也可以类似于乘号（或小写的“x”），甚至是手枪瞄准镜内显示的准星图标。十字符号通常用于屏幕捕捉程序，如果用户想准确地知道正在点击哪一个像素，这就很有用。

移动光标

在很多情况下，`move`值会得到与`crosshair`类似的结果。创作人员需要指示一个屏幕元素可以移动时就会使用move，它通常显示为一个加粗的十字线，线的两端分别有箭头。也可以显示为一个“拳头”，用户点击并按下鼠标按钮时图标中的“手指”是弯曲的。图13-5给出了这两种可能的`move`图标。

图13-5：不同的move图标

还有一些与move相关的cursor值：e-resize、ne-resize等等。Windows和大多数图形化Unix-shell用户会把这些值识别为鼠标光标放在窗口一边或角落时出现的图标。例如，把光标放在窗口的右边界上会出现一个e-resize光标，指示用户可以把窗口的右边来回拖动来改变窗口大小。把光标放在左下角则会显示sw-resize光标图标。有很多不同的方法可以表现这些图标；图13-6显示了一些可能的图标。

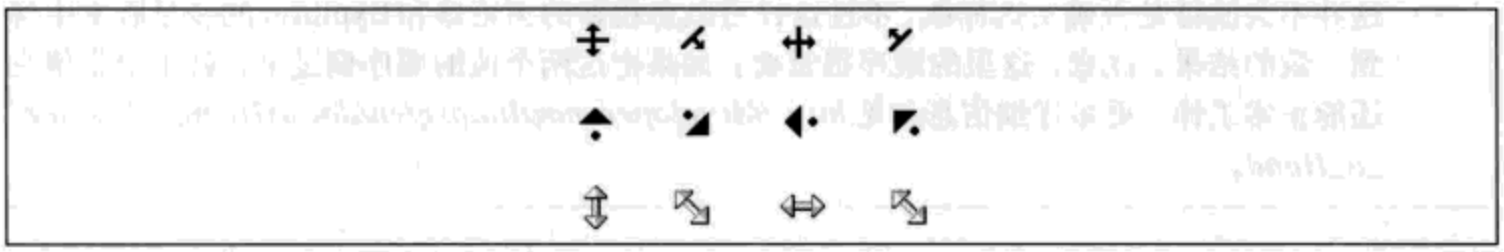

图13-6：选择“大小调整”光标

等待和前进

wait和progress都指示程序正在忙。不过，它们并不相同：wait表示用户要等待直到程序不忙为止，而progress指示用户完全可以继续与程序交互，尽管它很忙。在大多数操作系统中，wait可能显示为一块表（可能有旋转的指针）或者显示为一个沙漏（可能在自己倒来倒去）。progress通常表示为一个旋转的“沙滩球”，或者是一个箭头，而且在这个箭头的一旁有一个小沙漏。图13-7显示了这样一些图标。

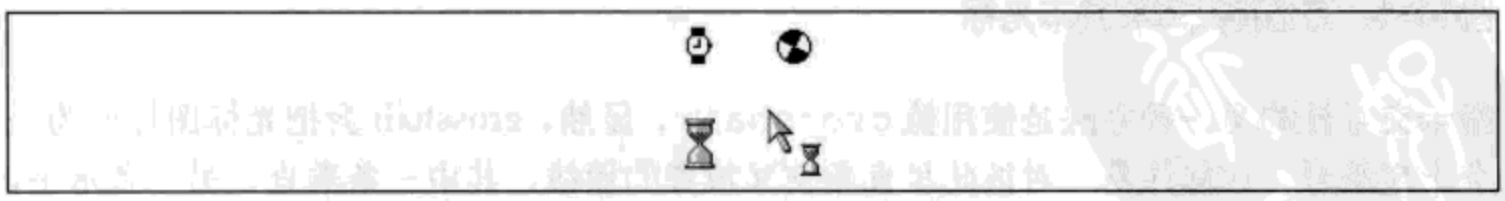

图13-7：等待与前进

注意： 值progress在CSS2.1中引入。

提供帮助

有时创作人员希望指示用户可以得到某种形式的帮助，此时就可以使用值help。help有两种非常常见的表现方式，可能是一个问号；也可能是一个箭头，箭头旁边有一个小

问号。如果已经确定某些链接指向更多信息，或者这些链接指向的信息有助于用户更好地理解网页，help就很有用。例如：

```
a.help {cursor: help;}
```

还可以使用help指示一个元素有“额外”信息，如有title属性的acronym元素。在很多用户代理中，把光标放在一个有标题的缩写词上时，用户代理会在一个“工具提示”中显示title属性的内容。不过，如果用户把光标移动得很快，或者用户的计算机速度很慢，倘若光标没有改变，用户可能不知道还有额外的信息。对于这些用户，以下规则可能很有用，并将会得到如图 13-8 所示的结果：

```
acronym[title] {cursor: help; border-bottom: 1px dotted gray;}
```

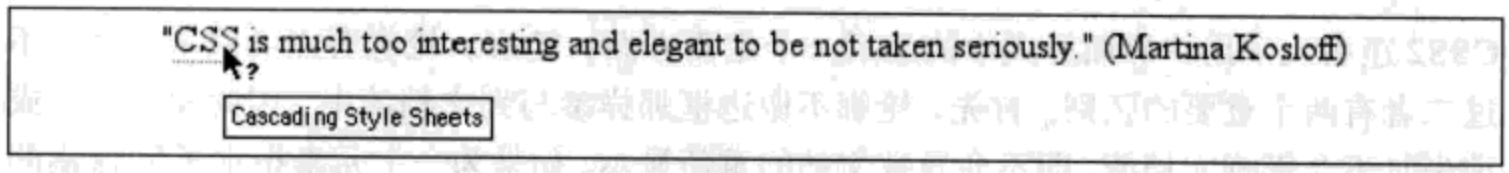

图13-8：显示提供了帮助（更多信息）

图形光标

最后也是最有意思的一点是，还可以指定定制光标。这可以使用一个URL值做到：

```
a.external {cursor: url(globe.cur), pointer;}
```

利用这个规则，要求用户代理加载文件 *globe.cur*，并将其用作光标图标，如图 13-9 所示。

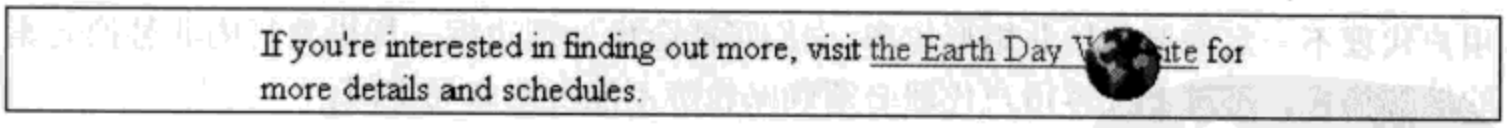

图13-9：使用一个定制图形光标

当然，用户代理必须支持存储 *globe.cur* 所用的文件格式。如果用户代理不支持这种格式，就会转而使用值pointer。注意，在cursor语法定义中，URL必须跟有一个逗号和某个通用关键字。这与属性font-family不同，对于font-family，可以指定一个特定字体系列而不必提供任何后路。实际上，对于可能采用的任何图形光标，cursor都要求有后路。

甚至可以在作为后路的关键字之前指定多个光标文件。例如，可以用不同格式创建同样的基本光标，把它们放在一个规则中，希望用户代理至少支持其中的一个：

```
a.external {cursor: url(globe.svg#globe), url(globe.cur), url(globe.png),
  url(globe.gif), url(globe.xbm), pointer;}
```

用户代理会逐个查看各个URL，直到找到一个可以用作为光标图标的文件。如果用户代理无法找到任何支持的文件，就会使用作为后路的关键字。

注意： 如果用户代理支持动画图形文件来替换光标，就可以实现动画光标。例如，IE6就支持利用*.ani*文件实现这种功能。

轮廓

CSS2还引入了用户界面样式中的最后一个主要方面：轮廓。轮廓有点类似于边框，不过二者有两个重要的区别。首先，轮廓不像边框那样参与到文档流中，因此轮廓出现或消失时不会影响文档流，即不会导致文档的重新显示。如果为一个元素指定了50像素的轮廓，这个轮廓很可能会覆盖其他元素。其次，轮廓可能不是矩形，不过不要高兴得太早，这并不是说可以创建圆形轮廓。相反，这意味着行内元素的轮廓可能不同于该元素的边框。利用轮廓，用户代理可以“合并”部分轮廓，创建一个连续但非矩形的形状。图13-10显示了这样一个例子。

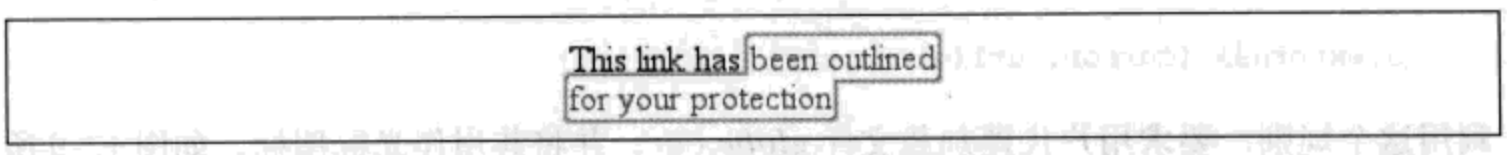

图13-10：轮廓可以有不规则形状

用户代理不一定需要支持非矩形轮廓。它们可能会像处理边框一样设置行内非替换元素的轮廓格式。不过，兼容用户代理必须确保轮廓不能占据布局空间。

轮廓和边框之间还有一个根本的差异：它们不是同一个东西，所以可以在同一个元素上共存。这会导致一些有意思的效果，如图13-11所示。

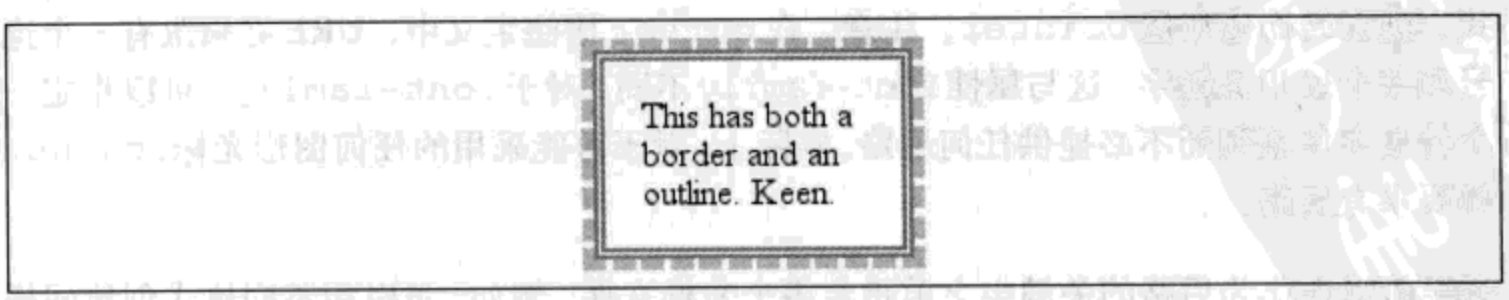

图13-11：边框与轮廓共存

CSS2规范指出："轮廓可以画在边框边界的外面。"注意这句话里的措辞。只是建议用户代理按这句话去做，不过这并不是必须恪守的要求。有些用户代理可能决定把轮廓画在内边框边界内，或者离边框有一点距离的地方。写作本书时，所有支持轮廓的浏览器都会把轮廓画在外边框边界外面，所以谢天谢地这是一致的。

一般认为轮廓是用户界面样式的一部分，这是因为它们最常用于指示当前焦点。如果一个用户使用键盘导航从一个链接跳到另一个链接，那么当前有焦点的链接往往会有一个轮廓。在Windows平台的Internet Explorer中，轮廓应用于用户已选择的任何链接（如果用户使用鼠标，则是用鼠标"点击"的链接），而且会一直留在那里，尽管有时并不希望如此。其他浏览器会向有键盘输入焦点的文本输入应用轮廓，指示用户键入时会从哪里开始输入。

可以看到，轮廓的样式与边框很类似，不过除了前面所提到的，二者之间还有一些重要的差别。下面只是简略地介绍它们的相似性，更多的时间将用来讨论二者的不同。

设置轮廓样式

类似于边框，轮廓最基本的方面就是样式，这要利用`outline-style`设置。

outline-style

值：	none | dotted | dashed | solid | double | groove | ridge | inset | outset | inherit
初始值：	none
应用于：	所有元素
继承性：	无
计算值：	根据指定确定

这组样式关键字与边框样式关键字基本上相同，其视觉效果也是一样的。只是少了一个关键字：`hidden`不是一个合法的轮廓样式，用户代理实际上要把它处理为`none`。这是有道理的，因为即使轮廓可见也不会影响布局。

轮廓与边框之间的另一个区别是，只能为一个`outline-style`值指定一个关键字（而边框可以指定最多4个关键字）。其实际效果是，一个元素周围必然有相同的轮廓样式，而不论轮廓是否是矩形。这可能也是对的，因为对于非矩形轮廓，要想确定如何应用不同的样式可能很困难。

轮廓宽度

一旦建立了轮廓并为之指定样式，接下来可以用outline-width定义轮廓的宽度（这一点不难猜到），这是一个不错的想法。

outline-width

值：	thin | medium | thick | <length> | inherit
初始值：	medium
应用于：	所有元素
继承性：	无
计算值：	绝对长度；如果轮廓的样式是none或hidden，则为0

如果你设置过边框的宽度，对这个关键字列表应该不陌生。outline-width与border-width之间唯一真正的区别是整个轮廓只能声明一个宽度（类似于轮廓样式）。因此，一个值中只允许有一个关键字。

设置轮廓颜色

既然可以设置样式和宽度，可以想见，有一个outline-color属性来指定轮廓的颜色。

outline-color

值：	<color> | invert | inherit
初始值：	invert （或用户代理特定的值；见正文说明）
应用于：	所有元素
继承性：	无
计算值：	根据指定确定

边框与轮廓之间最有意思的差别就出现在这里：轮廓颜色有关键字invert，而且这是默认值。反色轮廓意味着要对轮廓所在的像素完成反色转换。见图13-12所示。

invert导致对轮廓“后面”的像素完成反色转换，这个过程可以确保不论轮廓后面是什么都将可见。如果一个用户代理出于某种原因无法支持反色转换，则会使用元素的color计算值。

图13-12：对轮廓反色

能够对屏幕上的像素反色，这很有意思，特别是对轮廓的宽度没有理论上的限制。所以，如果你愿意，完全可以使用轮廓对文档中的很大一部分完成反色。这并不是使用轮廓的本来目的，不过确实可以这样做，图 13-13 显示了这种做法的一个结果。

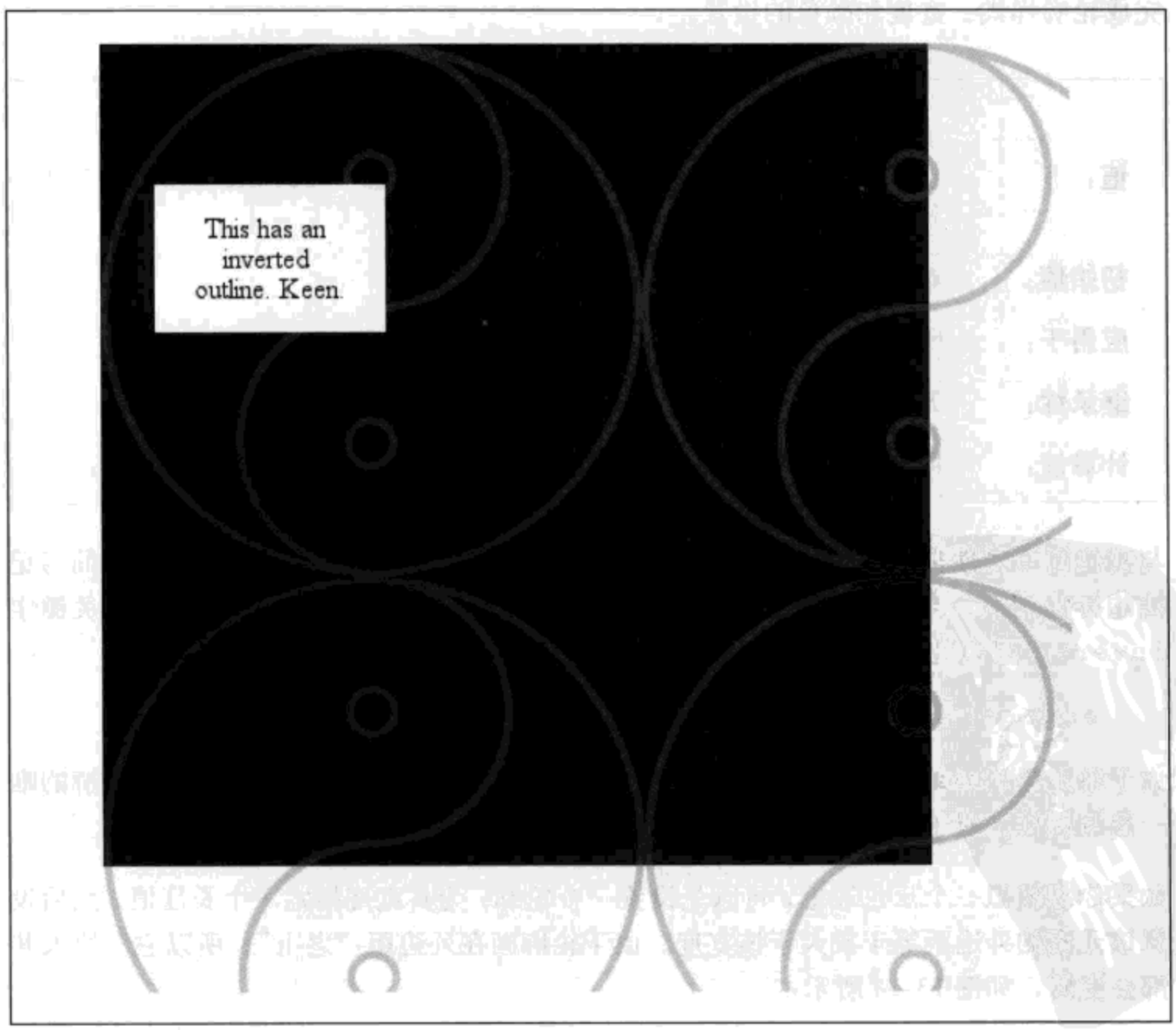

图13-13：大范围反色

当然，如果想为轮廓定义一种特定的颜色，只需使用任何合法的颜色值。以下声明的结果应该很显然：

```
outline-color: red;
outline-color: #000;
outline-color: rgb(50%,50%,50%);
```

这里可能存在一个缺点，轮廓颜色有可能与其周围的像素颜色很接近，这种情况下用户将无法看清。正因如此才定义了 invert。

类似于轮廓样式和宽度，对整个轮廓只能定义一种颜色。

汇总

类似于设置边框样式的 border 属性，轮廓也有一个简写属性 outline，允许一次就完成轮廓样式、宽度和颜色的设置。

outline	
值：	[<outline-color> || <outline-style> || <outline-width>] | inherit
初始值：	对简写属性未定义
应用于：	所有元素
继承性：	无
计算值：	见各个属性（outline-color 等等）

与其他简写属性类似，outline 把多个属性汇总为一个简洁的记法。它与其他简写记法的行为相同，会覆盖先前定义的值。因此，在以下例子中，轮廓会使用颜色关键字 invert，因为这是第二个声明指示的颜色（默认值）：

```
a:focus {outline-color: red; outline: thick solid;}
```

由于给定轮廓必须采用某种统一的样式、宽度和颜色，所以 outline 是关于轮廓的唯一简写属性。对于轮廓没有诸如 outline-top 或 outline-right 之类的属性。

如果你想模拟一个反色边框，可以先设置一个轮廓，为其宽度指定一个长度值，然后设置该元素的外边距等于或大于该宽度。由于轮廓画在外边距“之上”，所以它会填入其部分空间，如图 13-14 所示：

```
div#callbox {outline: 5px solid invert; margin: 5px;}
input:focus {outline: 1em double gray;}
```

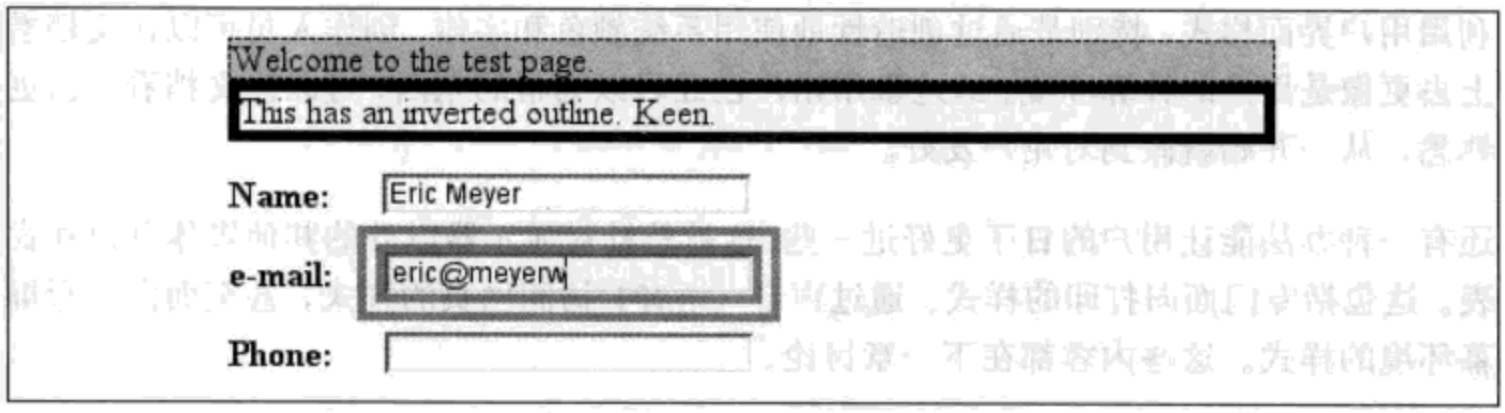

图13-14：不同的轮廓

前面曾提到过，轮廓不会参与到文档流中。因此即使因为焦点的改变使轮廓从一个链接移动到另一个链接上，也不会导致文档重新显示。如果创作人员使用边框指示焦点，文档布局就会不停地变化和跳跃。轮廓可以得到与边框同样的效果，但不会导致跳跃。

之所以轮廓可以做到这一点，原因在于：根据定义，轮廓画在元素框余下部分之上。由于CSS2中轮廓不会覆盖其元素框的可见部分，而只可能覆盖外边距（透明外边距），所以这不算大问题。如果CSS的将来版本允许轮廓向里移，能覆盖边框或元素框的其他可见部分，轮廓的放置就会变得更重要。

遗憾的是，CSS2有一些方面很含糊，其中包括它明确地拒绝定义两个轮廓相互重叠时的行为，也没有定义元素被其他元素部分模糊时对其轮廓会有怎样的影响。这些问题可以用一个例子来说明：

```
div#one {outline: 1em solid invert;}
div#two {outline: 1em solid invert; margin: -2em -2em 0 2em;
  background: white;}
```

现在假设在文档中div#two就在div#one之后。它会与第一个div重叠，其背景将与第一个div轮廓的一部分重叠。这里没有提供此代码块的示意图，因为CSS2规范对于会发生什么情况没有任何说明。是第一个div的轮廓可见，覆盖第二个div的背景和内容吗？这两个反色轮廓会在某些地方相交；那里会发生什么？像素会反色两次吗（相应地最终恢复为其原来的状态）？或者像素是不是只反色一次，然后不再改变？我们无从知道。这里无论怎么讲都不能说对也不能说错，而只是一种可能的结果，它不一定是用户代理最终的实现，也不一定是CSS将来版本所定义的行为。

小结

利用用户界面样式，特别是通过创造性地使用系统颜色和字体，创作人员可以让文档看上去更像是用户的计算环境。通过重用用户已经习以为常的东西，可以让文档看上去更熟悉，从一开始就做到对用户友好。

还有一种办法能让用户的日子更好过一些，这就是针对显示器以外的其他媒体创建样式表。这包括专门面向打印的样式、通过声音（语音）访问网页的样式，甚至面向投影屏幕环境的样式。这些内容都在下一章讨论。

第14章

非屏幕媒体

并不是每一个上网的人都能看到本书里讨论的效果。美国大约有110万人是盲人，与视力健全的人相比，他们对Web有着完全不同的体验。

幸运的是，借助于非视觉访问途径，CSS并不是寂静无声的。CSS2提供了在非屏幕媒体中应用样式的功能。Web的大多数演进都在显示器上发生，也就是在视觉媒体中完成，不过只要用户代理提供了适当的支持，CSS2也可以用于非视觉媒体。

可以将文档设计为既能通过视觉途径表现，又能通过非视觉途径表现，绝对不能小看这样做的好处。如果拿到一个文档，可以使用特定于媒体的不同样式表分别针对屏幕、打印和声音表现等场合重新应用样式，这样可以省去你的很多麻烦。例如，你不需要再链接到页面的一个“打印机友好”版本。不用创建完全不同的标记结构（比如说一个用于屏幕媒体，另一个用于打印媒体），通过重用同一个文档可以让你的网站更有效率。

为此，对于一个包含幻灯片大纲的HTML文档，可以应用样式以便在屏幕上阅读，或者得到简洁可读的打印输出、显示为幻灯片，以及使用屏幕阅读器能解释的某种方式表现。在这一章中，我们将介绍如何完成后三个任务（因为本书前面讨论的都是屏幕表现问题）。

设计特定于媒体的样式表

利用HTML和CSS中定义的机制，可以将样式表限制为仅用于某种特定媒体。对于基于HTML的样式表，可以通过`media`属性对媒体做出限制。在`link`和`style`元素中的用法是一样的：

```
<link rel="stylesheet" type="text/css" media="print"
   href="article-print.css">
<style type="text/css" media="projection">
body {font-family: sans-serif;}
</style>
```

media 属性可以接受一个媒体值或者媒体值表（各媒体值之间用逗号分隔）。因此，要链入一个只用于屏幕和投影媒体的样式表，可以写作：

```
<link rel="stylesheet" type="text/css" media="screen, projection"
   href="visual.css">
```

在样式表本身，还可以在 @import 规则上限制媒体：

```
@import url(visual.css) screen, projection;
@import url(article-print.css) print;
```

要记住，如果没有为样式表增加媒体信息，它会应用于所有媒体。因此，如果你希望一组样式只应用于屏幕媒体，而另外一组样式只应用于打印媒体，就要向这两个样式表显式地增加媒体信息。例如：

```
<link rel="stylesheet" type="text/css" media="screen"
   href="article-screen.css">
<link rel="stylesheet" type="text/css" media="print"
   href="article-print.css">
```

如果从上例中第一个 link 元素去掉 media 属性，样式表 *article-screen.css* 中的规则就会应用到所有媒体，包括打印媒体、投影媒体和所有其他媒体。

CSS2 还定义了 @media 块的语法，允许在同一个样式表中为多个媒体定义样式。考虑以下这个简单的例子：

```
<style type="text/css">
body {background: white; color: black;}
@media screen {
   body {font-family: sans-serif;}
   h1 {margin-top: 1em;}
}
@media print {
   body {font-family: serif;}
   h1 {margin-top: 2em; border-bottom: 1px solid silver;}
}
</style>
```

在此可以看到，所有媒体中的 body 元素都指定有一个白色背景和一个黑色前景。然后单独为屏幕媒体提供了一组规则，接下来的另一组规则只应用于打印媒体。

@media 块的大小不限，可以包含任意多个规则。如果创作人员只能控制一个样式表，@media 块则可能是定义特定媒体样式表的唯一途径。如果使用 CSS 对一个使用 XML

语言的文档应用样式，而这个XML语言中没有media属性或相应属性，在这种情况下，也必须使用 @media 块。

分页媒体

在CSS术语中，分页媒体（paged medium）是把文档表示处理为一系列离散“页面”的媒体。这与屏幕媒体有所不同，屏幕媒体是一种连续型媒体（continuous medium）：文档表示为一个可滚动的“页面”。可以把连续媒体比作是一本手卷。打印资料，如书、杂志和激光打印机输出，都是分页媒体。幻灯片也是分页媒体，因为一次只会显示一系列幻灯片中的一张。每张幻灯片在CSS术语中就是一个“页面”。

打印样式

即使在“无纸化”的将来，最常见到的分页媒体仍是文档的打印输出，如Web页面、字处理文档、电子表格或成张的输出。创作人员可以采取很多措施让用户更喜欢文档的打印输出，如改变分页或者是创建专用于打印的样式。

注意，打印样式也会应用于“打印预览”模式的文档显示。因此，在某些情况下可以在显示器上看到打印样式。

屏幕与打印的区别

除了明显的物理差异外，屏幕和打印设计之间还有很多样式上的不同。最基本的差别是字体选择。大多数设计人员会告诉你最适合屏幕设计的字体是sans-serif字体，但是在打印媒体中serif字体更可读。因此，你可能会建立这样一个打印样式表，对文档中的文本使用Times字体而不是Verdana字体。

另一个主要区别在于字体大小。如果你做过Web设计，可能会一而再、再而三地听人说，点作为Web上的字体大小是一个不好的选择。这种说法基本上是对的，特别是如果你希望不同的浏览器和操作系统上有一致大小的文本。不过，Web设计不是打印设计，同样的，打印设计也并非Web设计。使用点（甚至厘米或派卡）在打印设计中是完全可以的，因为打印设备知道其输出区域的物理大小。如果一个打印机上放了大小为8.5×11的纸张，它就知道打印区域由这张纸的边界界定，还会知道一英寸有多少点，因为它很清楚自己的分辨率（dpi，每英寸点数）。这说明，打印机可以处理物理世界的长度单位，如点。

所以，很多打印样式表可能最前面都有下面的声明：

```
body {font: 12pt "Times New Roman", "TimesNR", Times, serif;}
```

这很常见，可能会让图形设计人员暗自窃喜。不过一定要知道，之所以能接受点只是因为打印媒体特有的性质，对于Web设计来说，点还是不太适用。

另外，大多数打印输出中都没有背景，这可能会让设计人员有些迷惑。为了节省用户的打印油墨，大多数Web浏览器都预先配置为不打印背景色和图像。如果用户确实希望在打印输出中看到这些背景，就必须在首选项中修改一个选项。无法通过CSS强制打印背景。不过，可以使用一个打印样式表保证不打印背景。例如，可以在打印样式表中加入以下规则：

```
* {color: black !important; background: white !important;}
```

这样能确保所有元素都打印为黑色文本，而且会删除在全媒体样式表中指定的所有背景。由于大多数用户的打印机都会这样显示页面，最好遵循同样的方式建立你的打印样式。如果你的Web设计中设置了深灰色背景和黄色文本，以上规则还能确保有彩色打印机的用户不会在白纸上打印黄色文本。

注意：CSS2.x没有提供任何机制来指导如何根据用户的输出设备选择样式表。因此，所有打印机都会使用你定义的打印样式表。CSS3媒体查询模块定义了一些方法，可以向彩色打印机和灰度打印机发送不同的样式表，不过在写作本书时，对媒体查询的支持基本上还不存在。

分页媒体与连续媒体之间还有一个区别：在分页媒体中更难使用多列布局。假设你有一篇文章，其中文档分栏为两列。在打印输出中，各页的左边是第一列，右边包含第二列。这就要求用户先读每页的左边，然后回到打印输出的起始位置，再读每页的右边。在Web中这就很让人恼火，不过打印到纸上后情况更显糟糕。

显然，对此的解决方法是一方面使用CSS建立两列布局（可能通过浮动实现），然后编写一个打印样式表，将内容恢复成单列打印。因此，对于屏幕样式表可能会写以下规则：

```
div#leftcol {float: left; width: 45%;}
div#rightcol {float: right; width: 45%;}
```

然后在打印样式表中再写下面的规则：

```
div#leftcol, div#rightcol {float: none; width: auto;}
```

如果CSS有办法建立多列流式布局，就不需要这样做了。可惜，尽管多年来一直有这种提议，但写作本书时还是未见到任何举动。

关于打印设计的详细内容，可能要用一整章的篇幅才能说清，不过这并不是本书的主要目的。下面来讨论分页媒体CSS的详细内容，有关设计的内容留给其他书来讨论。

定义页面大小

类似于定义元素框的方法，CSS 定义了描述页面组件的页框（page box）。页框基本上由两个区组成：

- 页面区（page area），这是页面中放内容的部分。从某种程度上说，它类似于正常元素框中的内容区，因为页面区的边界相当于页面中布局的初始包含块（关于包含块的详细讨论见第 7 章）。
- 外边距区（margin area），这是围绕页面区的部分。

页框模型如图 14-1 所示。

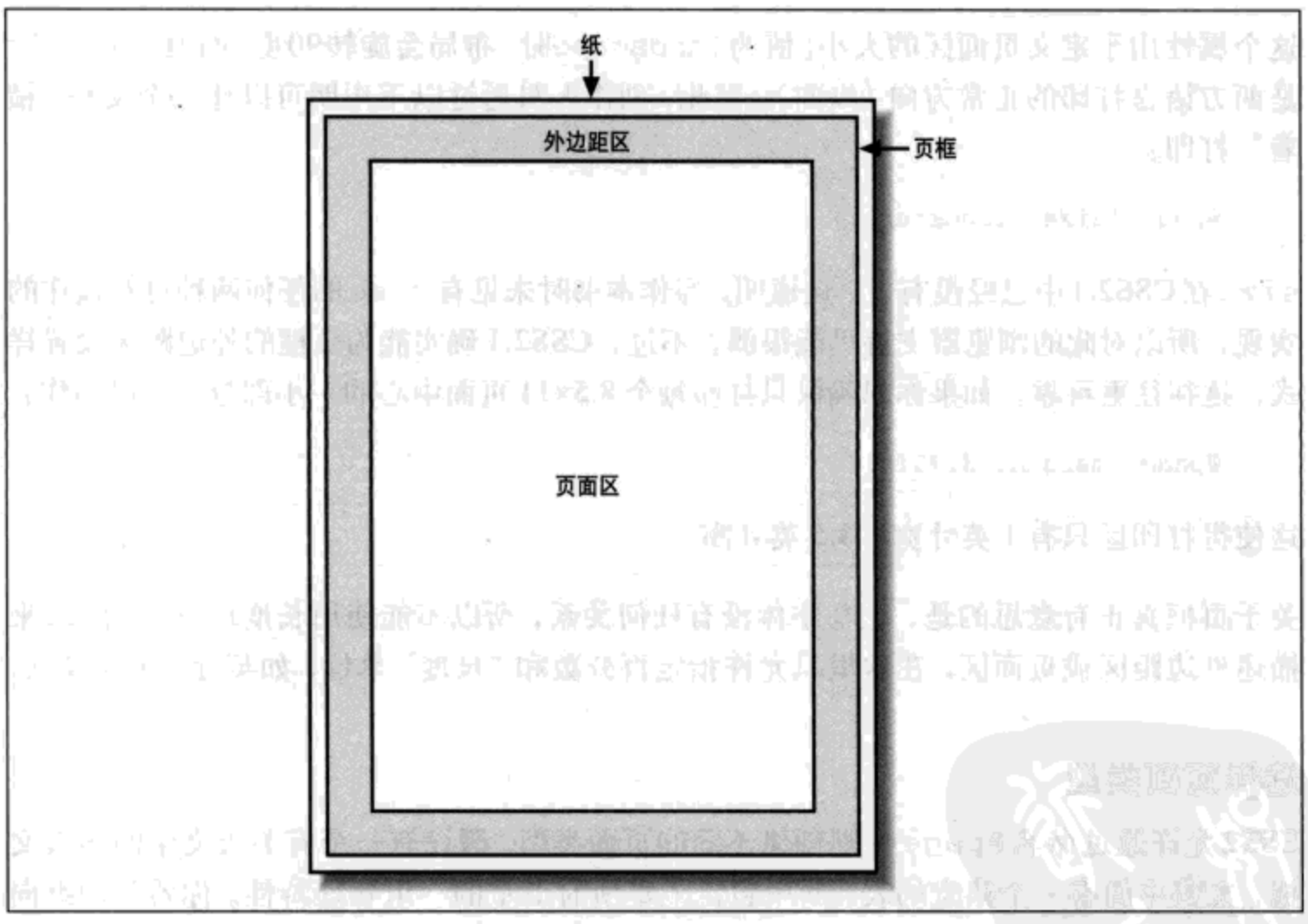

图14-1：页框

在CSS2中，可以定义页框的大小，还可以定义外边距大小。在CSS2.1中，创作人员只能设置外边距区的大小。这两个版本都要使用@page规则来完成设置。以下是一个简单的例子：

```
@page {size: 7.5in 10in; margin: 0.5in;}
```

这是一个CSS2规则，因为它使用了属性size，这个属性已经从CSS2.1中去除，原因是缺乏实现支持。

size	
值：	<length>{1,2} | auto | portrait | landscape | inherit
初始值：	auto
应用于：	页面区
继承性：	无

这个属性用于定义页面区的大小。值为landscape时，布局会旋转90度，而值portrait是西方语言打印的正常方向（纵向）。因此，创作人员通过以下声明可以让一个文档“横着”打印：

```
@page {size: landscape;}
```

size在CSS2.1中已经没有了，这说明，写作本书时未见有size的任何两种可互操作的实现，所以对此的浏览器支持可能很弱。不过，CSS2.1确实能为页框的外边距区设置样式，这往往更可靠。如果你想确保只打印每个8.5×11页面中心的一小部分，可以写作：

```
@page {margin: 3.75in;}
```

这使得打印区只有1英寸宽，3.5英寸高。

关于页框真正有意思的是，它与字体没有任何关系，所以不能使用长度单位em和ex来描述外边距区或页面区。在这里只允许指定百分数和“尺度”单位，如英寸、厘米或点。

选择页面类型

CSS2允许通过命名@page规则创建不同的页面类型。假设有一个有关天文学的多页文档，文档中间有一个很宽的表，其中包含土星所有卫星的一组物理特性。你希望用纵向（portrait）模式打印文本，不过表要采用横向（landscape）模式打印。首先可以写以下规则：

```
@page normal {size: portrait; margin: 1in;}
@page rotate {size: landscape; margin: 0.5in;}
```

现在只需根据需要应用这些页面类型。土星的卫星表的id为moon-data，所以可以写以下规则：

```
body {page: normal;}
table#moon-data {page: rotate;}
```

这会导致土星的卫星表以横向模式打印，不过文档的余下部分都纵向打印。这是通过page属性实现的（page属性也已经从CSS2.1中去除）。

page

值：	<identifier> | inherit
初始值：	auto
应用于：	块级元素
继承性：	有

从值定义可以看出，page的存在就是为了让你向文档中的不同元素指定命名页面类型。

通过特殊伪类，可以处理更多通用页面类型，更好的是，这在CSS2和CSS2.1中都有定义。:first允许向文档中的第一个页面应用特殊样式。例如，可能希望为第一个页面指定比其他页面更大的上外边距。可以如下做到：

```
@page {margin: 3cm;}
@page :first {margin-top: 6cm;}
```

这样一来，所有页面上都会有一个3厘米的外边距，不过第一页例外，它有一个6厘米的上外边距。其效果如图14-2所示。

除了为第一页指定样式，还可以对左页和右页分别设置样式，模拟书脊左右两边的书页。为此可以使用:left和:right指定不同的样式。例如：

```
@page :left {margin-left: 3cm; margin-right: 5cm;}
@page :right {margin-left: 5cm; margin-right: 3cm;}
```

这些规则的效果是在书脊左右两边的左页和右页内容“之间”指定更大的外边距。页面要装订成书时，这是一个很常用的实践做法。上述规则的结果见图14-3所示。

分页

在分页媒体中，对分页和页面放置方式有所影响是一个不错的想法。可以使用属性page-break-before和page-break-after影响分页，它们接受同样的一组值。

图14-2：为第一页指定特殊样式

默认值auto只是说明不要求在元素之前或之后分页。这与正常的打印输出相同。always值导致在设置样式的元素之前（或之后）放一个分页符。

例如，假设遇到以下情况，页面标题是一个h1元素，小节标题都是h2元素。你可能希望在文档的各小节开始之前有一个分页，另外在文档标题之后分页。为此需要以下规则，其结果见图14-4所示：

```
h1 {page-break-after: always;}
h2 {page-break-before: always;}
```

当然，如果希望文档标题在页面上居中，可以增加规则来达到这个效果。既然没有增加这种规则，各页只会采用非常直接的样式表现。

值left和right的做法与always相同，只不过它们更进一步定义了继续打印哪种类型的页面。考虑以下规则：

```
h2 {page-break-before: left;}
```

这会要求所有h2元素的前面有足够的分页符，使得h2在左页的顶部打印；也就是说，如果要装订打印输出，h2所在页面要出现在书脊的左边。如果是双面打印，这意味着在一张纸的“背面”打印。

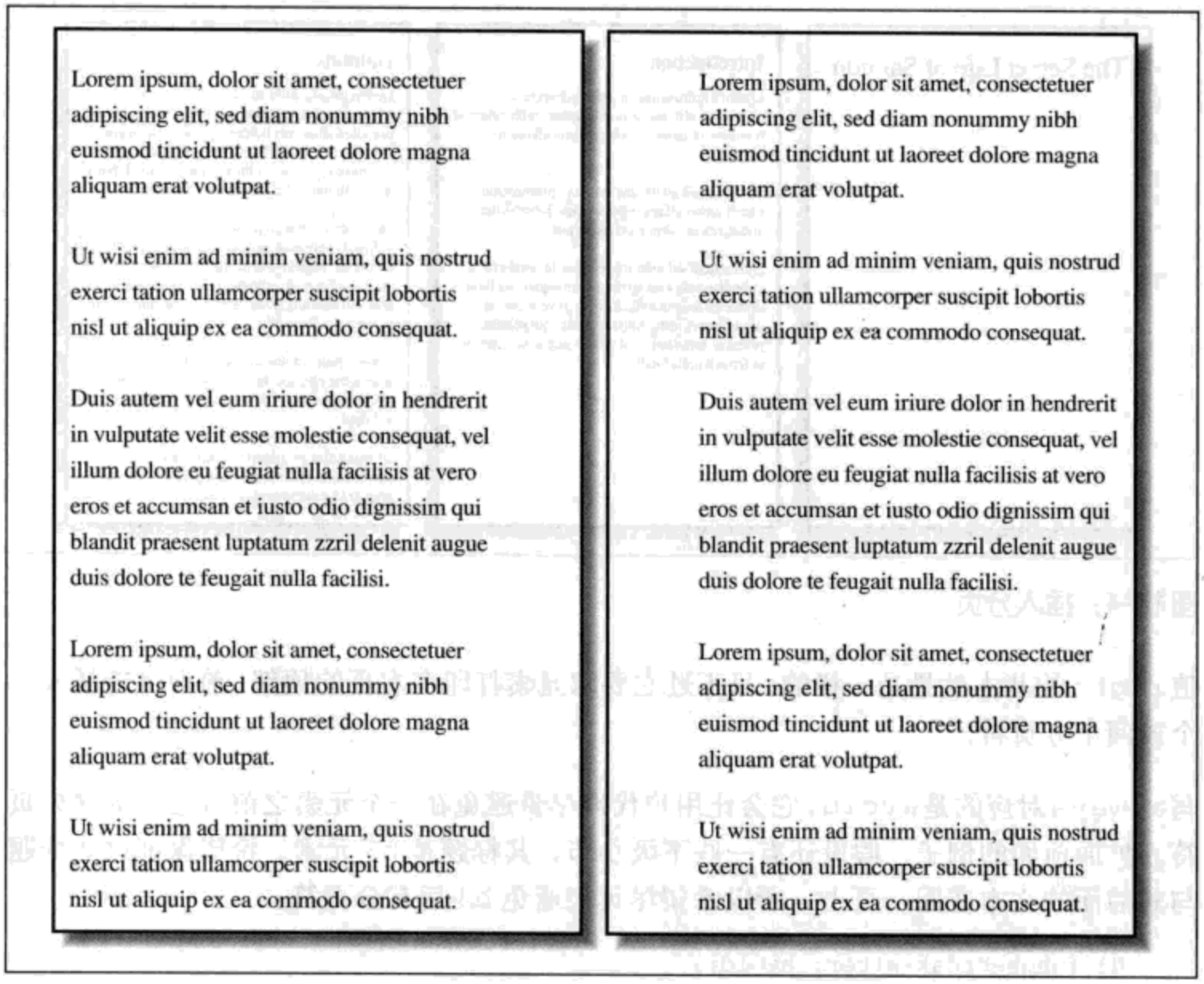

图14-3：为左页和右页设置不同的样式

page-break-before、page-break-after

值：	auto | always | avoid | left | right | inherit
初始值：	auto
应用于：	position值为relative或static的非浮动块级元素
继承性：	无
计算值：	根据指定确定

假设打印时h2之前的元素打印在一个右页上。前面的规则就会在h2之前插入一个分页，将其推到下一页。不过，如果另一个h2前面的元素打印在左页上，这个h2前面就会插入两个分页符，使之放在下一个左页的顶部。两个左页之间的右页故意留作空白页。

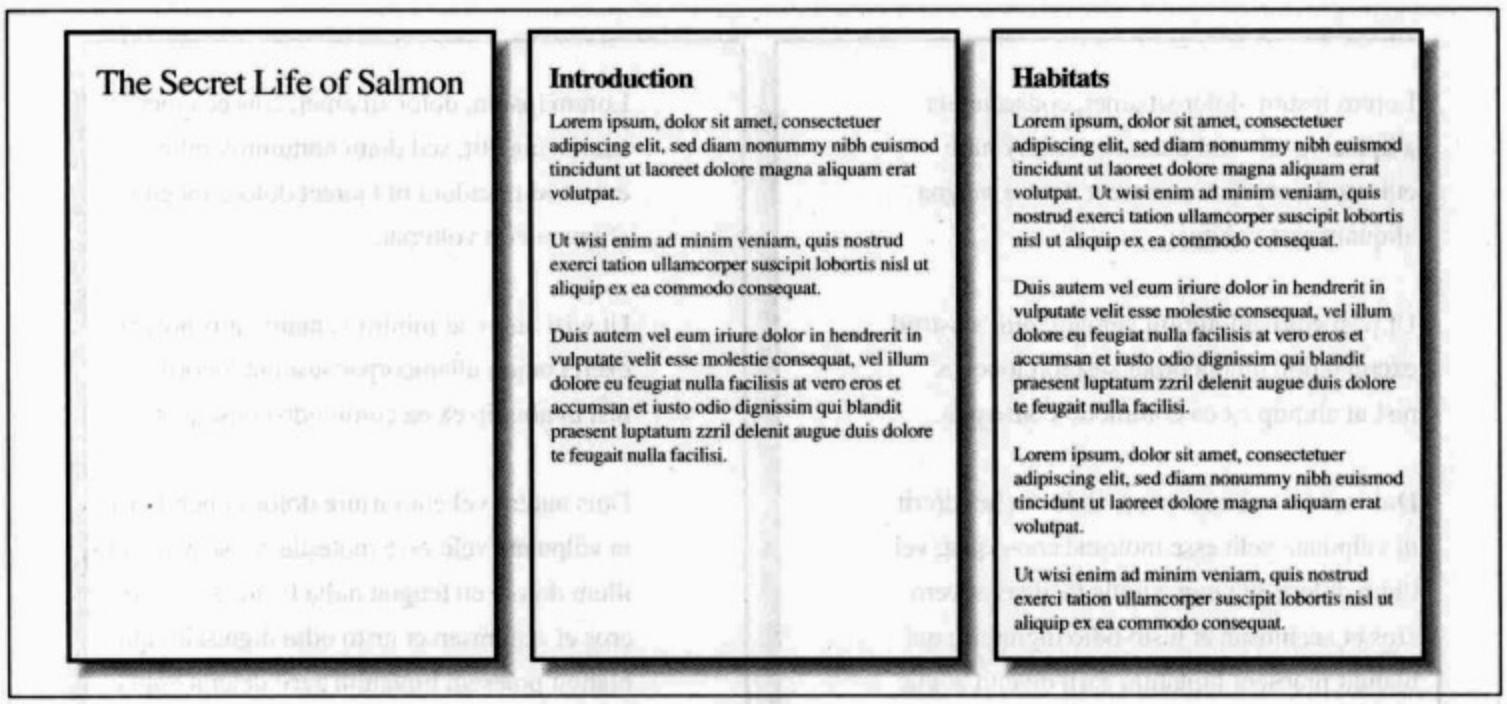

图14-4：插入分页

值right的基本效果是一样的，只不过它要求元素打印在右页的顶部，前面可能插入一个或两个分页符。

与always对应的是avoid，它会让用户代理尽量避免在一个元素之前或之后放置分页符。扩展前面的例子，假设还有一些下级小节，其标题是h3元素。希望保证这些标题与其后面的文本在同一页上，所以希望尽可能避免h3后有分页符：

```
h3 {page-break-after: avoid;}
```

不过，要注意这个值称为avoid而不是never。无法绝对保证一个给定元素之前或之后不插入分页符。考虑以下情况：

```
img {height: 9.5in; width: 8in; page-break-before: avoid;}
h4 {page-break-after: avoid;}
h4 + img {height: 10.5in;}
```

现在，进一步假设有以下情况，一个h4放在两个图像之间，其高度计算为半英寸。每个图像必须在一个单独的页面上打印，h4只可能放在两个位置上：第一个图像元素所在页面的底部，或者放在其后面的页面上。如果它放在第一个图像后面，就必须跟有一个分页符，因为后面再没有空间放第二个图像了，如图14-5所示。

另一方面，如果h4放在第一个图像后面的下一个新页面上，那么这一页上将没有空间放第二个图像。所以同样地，h4之后也会有一个分页符。无论哪一种情况，至少一个图像前面有一个分页符（或者两个图像前面都有分页符）。对于这种情况，用户代理将无能为力。

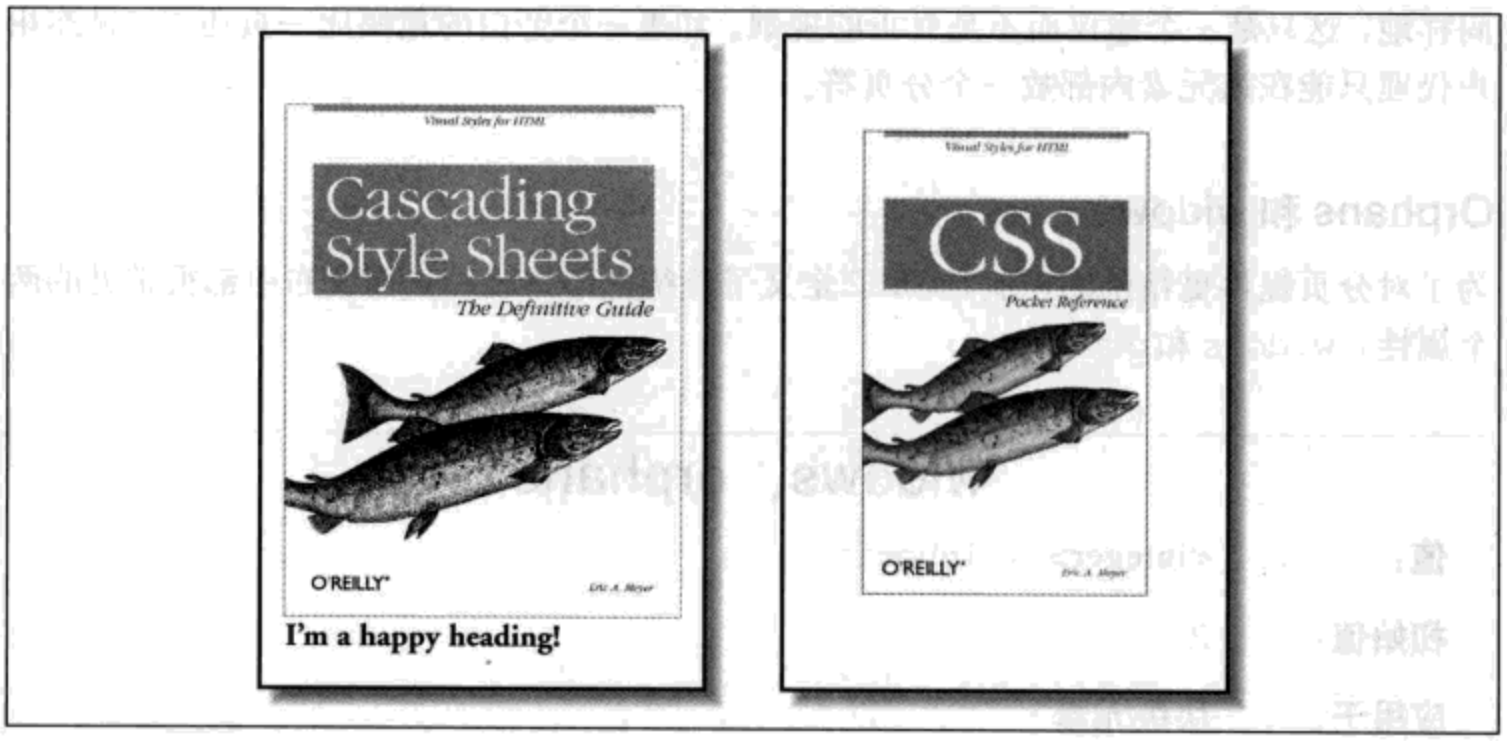

图14-5：必要的分页

显然这种情况很少见，不过它确实存在 —— 例如，如果一个文档中标题的后面只有表。这些表可能就要以特殊的方式打印，从而保证标题元素后面跟有一个分页符（即使创作人员明确要求避免这些分页符，也不起作用）。

另一个分页属性page-break-inside也可能出现同样的问题。这个属性的可取值更有限。

page-break-inside

值：	auto | avoid | inherit
初始值：	auto
应用于：	position值为relative或static的非浮动块级元素
继承性：	有
计算值：	根据指定确定

利用page-break-inside，可以不采用默认做法而另作选择：请求用户代理尽量避免在一个元素内部放置分页符。如果有一系列“旁白”，而且不希望它们分跨两页，可以作以下声明：

```
div.aside {page-break-inside: avoid;}
```

同样地，这只是一个建议而不是真正的规则。如果一个旁白的篇幅比一页还长，显然用户代理只能在该元素内部放一个分页符。

Orphans 和 widows

为了对分页提供更精细的控制，CSS2 定义了传统打印排版和桌面发布中都很常见的两个属性：widows 和 orphans。

widows、orphans

值：	<integer> | inherit
初始值：	2
应用于：	块级元素
继承性：	有
计算值：	根据指定确定

这两个属性可谓殊途同归，目的一样，只是实现角度不同。值widows定义了放在页面顶部的元素在不导致前面增加分页符的前提下所包含的最小行框数。orphans 的作用恰好与之对应：它定义了不会导致元素前增加分页符的前提下可以出现在页面底部的最小行框数。

下面以 widows 为例。假设做了以下声明：

```
p {widows: 4;}
```

这意味着，所有段落都会在页面顶部显示不少于4个行框。如果文档的布局使得行框数小于4，那么整个段落都会放在页面顶部。考虑图 14-6 所示的情况。假设用手盖住图中的上半部分，只能看到第二页。可以注意到，对于上一页开始的段落，从上一页的最后到该段落的结束共有两个行框。如果使用默认的widows值2，这样显示是可以接受的。不过，如果值为 3 或更大，整个段落会作为一整块出现在第二个页面的顶部。这就要求在当前段落之前插入一个分页符。

再来看图 14-6，不过这一次用手把第二页盖住。可以注意到页面底部的 4 个行框，这是最后一个段落的前几行。如果orphans值等于或小于4，这是可以的。但是如果orphans值大于等于5，这个段落前面也会有一个分页符，并作为一整块单独放在第二页的顶部。

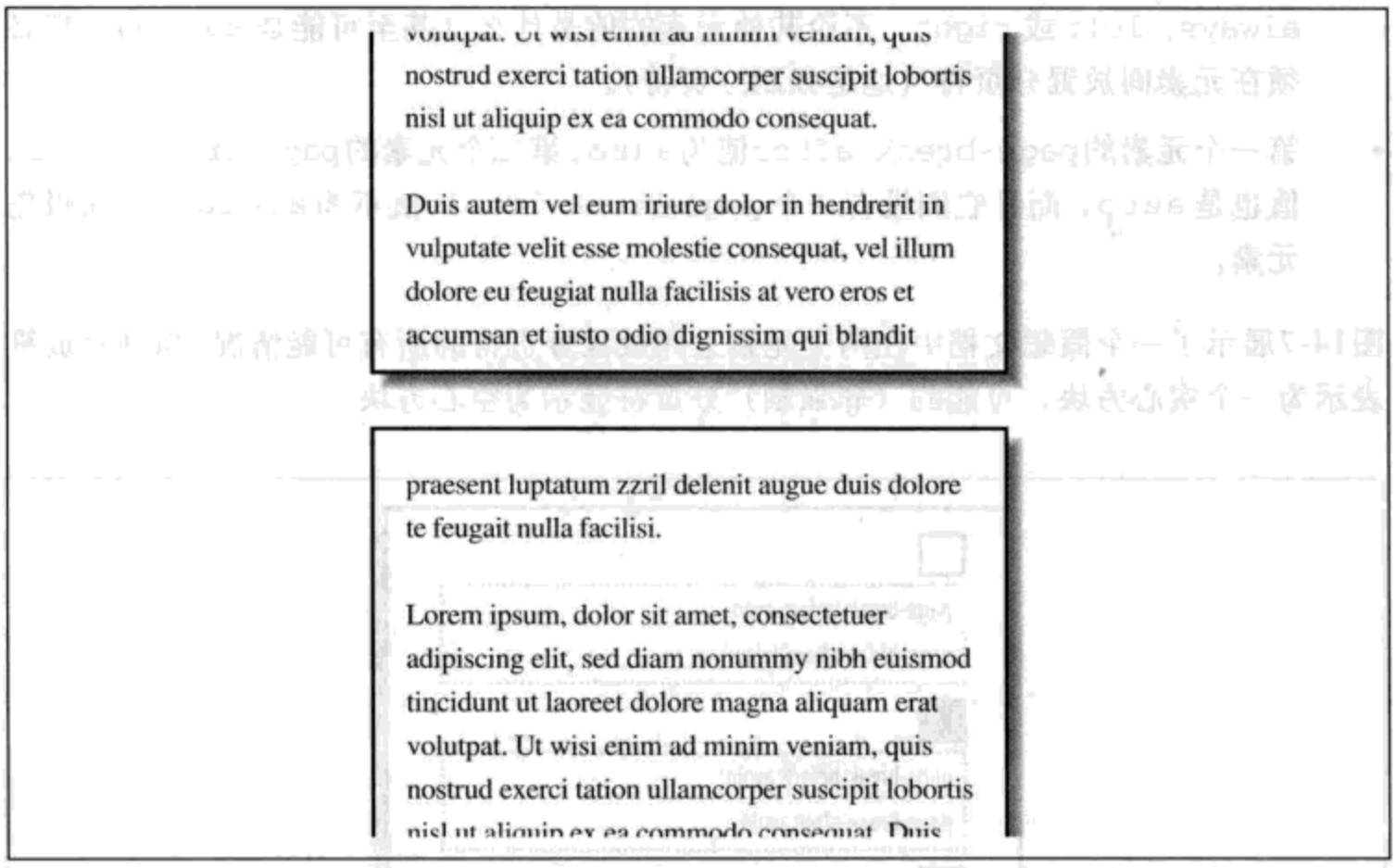

图14-6：利用widows

当然，orphans和widows都必须满足。如果创作人员声明了以下规则，大多数段落内部都不会有分页符：

```
p {widows: 30; orphans: 30;}
```

根据这些值，只有相当长的段落才会有内部分页符。当然，如果只是要避免内部分页，最好用以下规则描述：

```
p {page-break-inside: avoid;}
```

分页行为

由于CSS2允许一些奇怪的分页样式，它关于所允许的分页符和“最佳”分页符定义了一组行为。这些行为可以指导用户代理如何在不同的情况下处理分页。

实际上只有两个通用位置上允许有分页符。第一个是在两个块级框之间。如果分页符在两个块框之间，则分页符之前的元素的margin-bottom值重置为0，分页符后的元素的margin-top值也重置为0。不过，以下两个规则允许在两个元素框之间放置分页符：

- 第一个元素的page-break-after值或第二个元素的page-break-before值是

always、left或right。不论其他元素的值是什么（甚至可能是avoid），都必须在元素间放置分页符（这是强制分页符）。

- 第一个元素的page-break-after值为auto，第二个元素的page-break-before值也是auto，而且它们没有一个page-break-inside值不为avoid的共同祖先元素。

图14-7展示了一个假想文档中在两个元素之间放置分页符的所有可能情况。强制分页符表示为一个实心方块，可能的（非强制）分页符显示为空心方块。

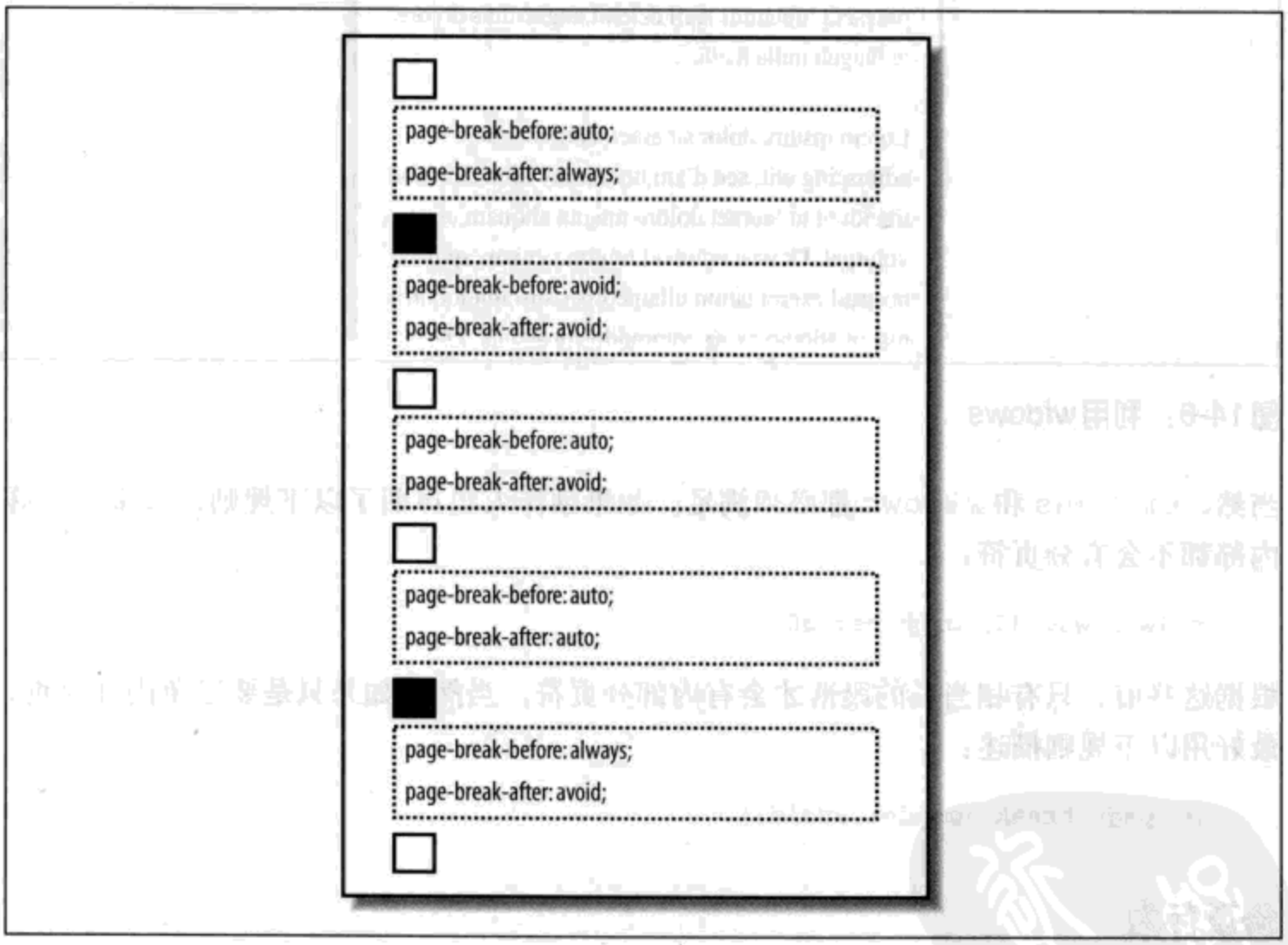

图14-7：块框之间可能放置的分页符

第二个允许放分页符的通用位置是一个块级框中的两个行框之间。这也由两个规则控制：

- 只有当元素开始与分页符之前行框之间的行框数小于该元素的orphans值时，才可能在两个行框之间出现分页符。类似地，只有当分页符之后行框与元素结束处之间的行框数小于widows值时，才可能在两个行框之间插入分页符。
- 如果元素的page-break-inside值不是avoid，可能在行框之间放分页符。

在这两种情况下，倘若无论分页符如何放置都无法使这两个规则同时满足，则会忽略控制分页符放置的第二条规则。因此，给定以下情况，一个元素指定了page-break-inside: avoid，但是它比一整页还长，则允许在该元素内部插入一个分页符。换句话说，会忽略在行框之间放置分页符的第二个规则。

如果忽略各组规则中的第二个规则后仍然不能很好地放置分页符，还可以忽略其他规则。在这种情况下，用户代理很可能忽略所有分页属性值，就好像它们都设置为auto一样，不过这种方法在CSS规范中没有定义（或未做要求）。

除了前面讨论过的规则外，CSS2还定义了一组“最佳”分页行为：

- 尽可能少分页。
- 让所有不是以强制分页符结尾的页面都有相同的高度。
- 避免在有边框的块内部分页。
- 避免在表内部分页。
- 避免在浮动元素内部分页。

这些建议对用户代理来说并不是必要的，不过它们提供了合理的指导，应该有助于得到理想的分页行为。

重复元素

对于使用分页媒体的创作人员，往往要求能有一个页头（running head）。所谓页头是指在每个页面上都出现的一个元素，如文档的标题或创作人员的姓名。在CSS2中可以使用固定定位元素指定页头。例如：

```
div#runhead {position: fixed; top: 0; right: 0;}
```

在文档输出到一个分页媒体时，这会把一个id为runhead的div放在每个页框的右上角。这个规则也会把该元素放在连续媒体视窗的右上角，如Web浏览器。以这种方式定位的元素会出现在每一页上。无法通过“复制”一个元素使之成为重复元素。因此，根据以下规则，h1元素会作为页头出现在每一页上，包括第一页：

```
h1 {position: fixed; top: 0; width: 100%; text-align: center;
   font-size: 80%; border-bottom: 1px solid gray;}
```

其缺点是，h1元素在第一页上定位时只能打印为页头。

页面外的元素

以上讨论的都是分页媒体中元素的定位，这就带来一个有意思的问题：如果元素定位在页框之外会发生什么情况？甚至不用定位就能造成这种情况。考虑一个pre元素，其中包含很宽的一行，共有411个字符。这很可能比所有标准纸张都宽，所以元素将比页框宽。那么会有什么结果呢？

对于这种情况，CSS2没有指出用户代理应该怎么做，所以要由用户代理自己确定解决方案。对于一个非常宽的pre元素，用户代理可能只是将该元素剪裁为适应页框大小，并扔掉余下的内容。它也可能生成额外的页面来显示元素中“多出”的部分。

关于如何处理超出页框的内容，有很多常用的建议，其中有两个非常重要。首先，内容应该能稍稍超出页框，从而允许生成“出血版”（译注1）。这说明，对于超出页框的部分内容，不会生成额外的页面。

其次，用户代理要注意不能只是为了满足定位信息而生成太多空页面。考虑以下情况：

```
h1 {position: absolute; top: 1500in;}
```

假设页框为10英寸高，为了满足这个规则，用户代理必须在h1前面加上150个分页符（相应地有150个空页）。相反，用户代理可以选择跳过这些空页，只是显示最后一页，最后这一页会具体包含h1元素。

还有两个建议指出，用户代理不能只是为了避免显示某些元素而把元素定位到奇怪的位置，另外放在页框之外的内容也能以多种方式表现（CSS的某些元素很有用也很复杂，不过有些看上去则很直接明了）。

投影样式

除了打印页面外，另一种常用的分页媒体是投影（projection），它描述了投影到一个大屏幕上的信息，适合很多观众观看。Microsoft PowerPoint就是当前最知名的投影媒体之一。

写作本书时，只有一个用户代理支持投影媒体CSS：Windows平台的Opera。这种功能称为“OperaShow”，它允许创作人员取任何HTML文档，将其转换成一个幻灯片。下面将介绍这种功能的基础，因为将来这种功能可能还会出现其他用户代理中，另外这提供了一个很有意思的视角，对于了解CSS在屏幕或打印以外的其他媒体中如何使用很有帮助。

译注1：出血版打印时允许超出页面边界。

建立幻灯片

如果把一个文档分成一系列幻灯片，需要一种办法来定义各幻灯片之间的界限。这个工作使用分页属性完成。到底使用page-break-before还是page-break-after，很大程度上取决于文档如何构造。

举例来说，考虑如图14-8所示的HTML文档。这里有一系列h2元素，每个h2后面有一个无序列表。这就构成了幻灯片的“大纲视图”。

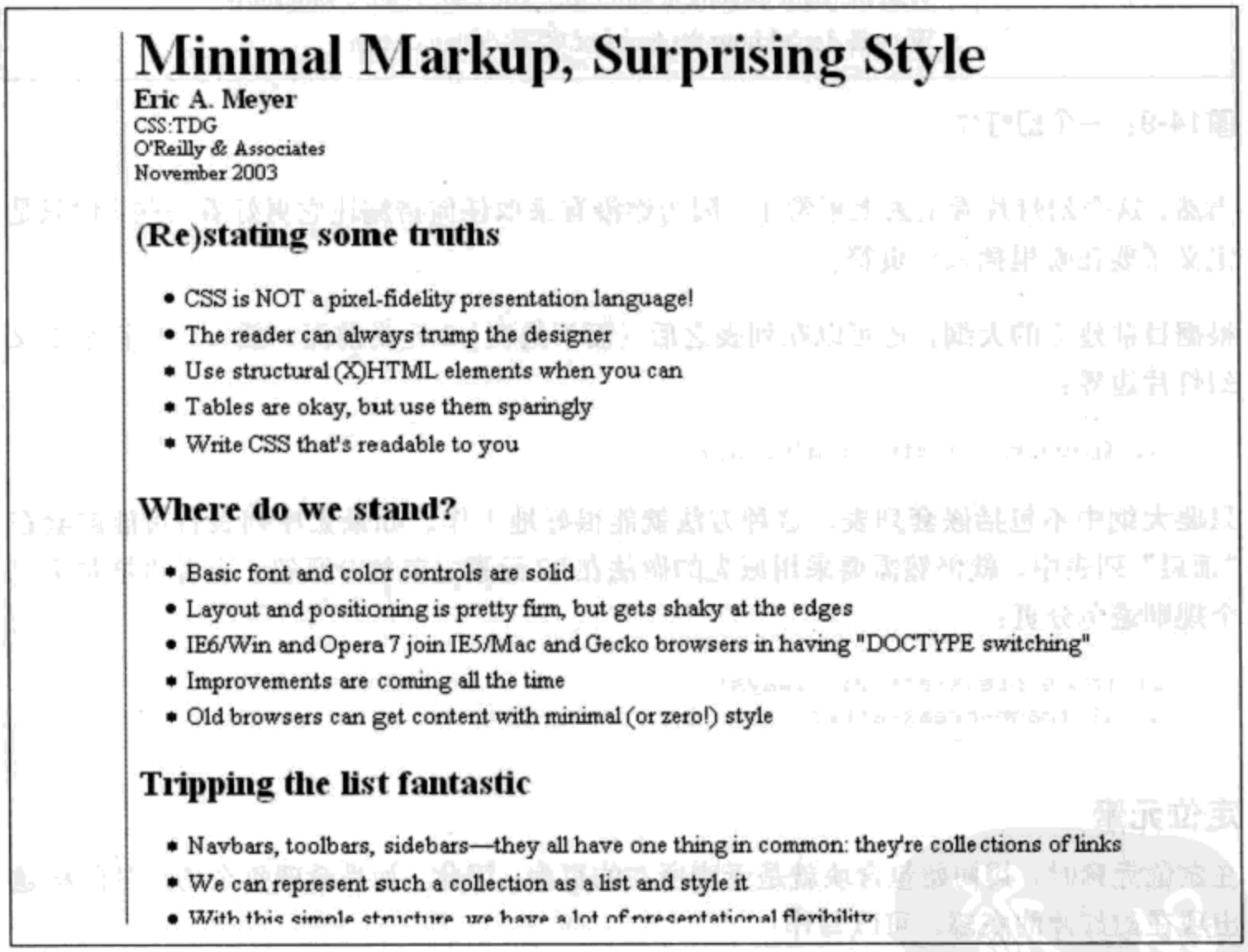

图14-8：使用简单HTML的幻灯片大纲

现在你要做的只是把文档分成幻灯片。由于每个幻灯片都从一个h2元素开始，只需声明：

```
h2 {page-break-before: always;}
```

这会确保每个页面（也就是每个幻灯片）都从一个h2元素开始。由于每个幻灯片的标题都由一个h2表示，所以这很合适：每个幻灯片都以一个h2作为其第一个元素。图14-9给出了一个幻灯片的表现。

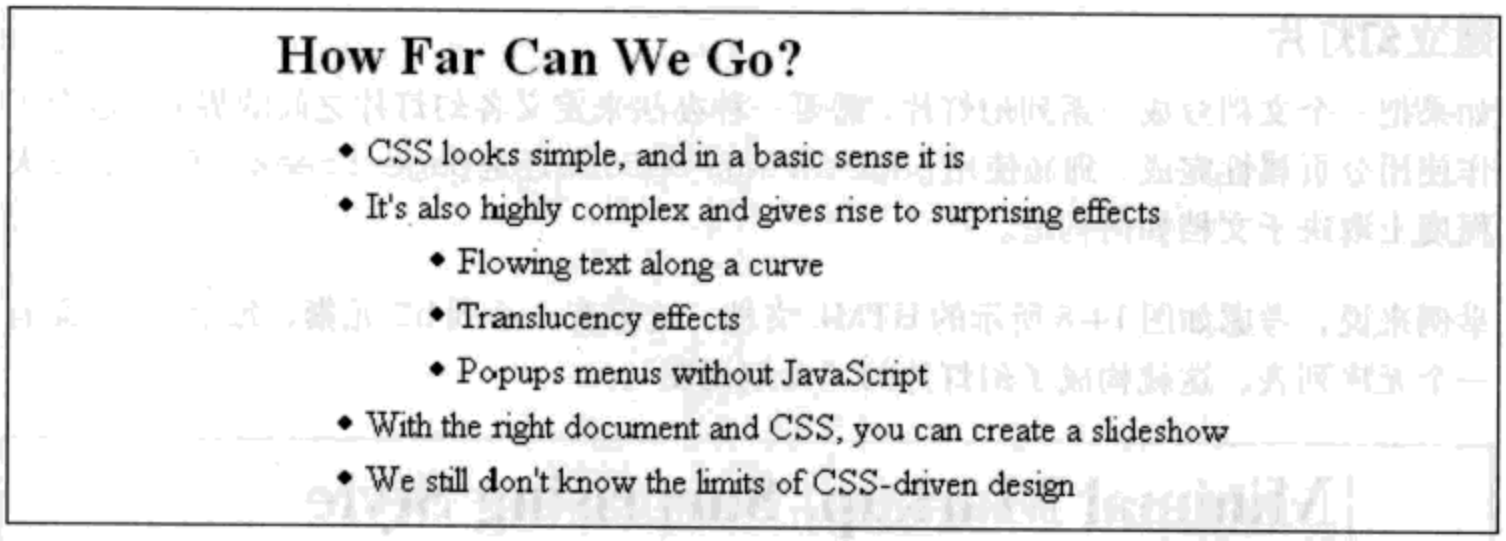

图14-9：一个幻灯片

当然，这个幻灯片看上去太平常了，因为你没有采取任何措施让它更好看一些；你只是定义了要在哪里插入分页符。

根据目前建立的大纲，还可以在列表之后（而不是在h2元素前面）插入分页符来定义幻灯片边界：

```
ul {page-break-after: always;}
```

只要大纲中不包括嵌套列表，这种方法就能很好地工作。如果无序列表有可能嵌套在“顶层”列表中，就仍然需要采用原先的做法在h2元素之前放分页符，或者再增加第二个规则避免分页：

```
ul {page-break-after: always;}
ul ul {page-break-after: auto;}
```

定位元素

在定位元素时，其初始包含块就是元素所在的页框。因此，如果希望每个幻灯片的标题出现在幻灯片的底部，可以写作：

```
h2 {page-break-before: always; position: absolute; bottom: 0; right: 0;}
```

这个规则将把所有h2元素放在其所在页框（幻灯片）的右下角。当然，也可以相对于其他元素来定位元素，而不是相对于页框。有关定位的详细内容见第10章的介绍。

另一方面，类似于打印媒体，固定定位元素会出现在幻灯片中的每一个页框中。这说明，可以将一个元素（如文档标题）放在每一个幻灯片上，如下：

```
h1 {position: fixed; top: 0; right: 0; font-size: 80%;}
```

这种技术可以用于创建每一页上的页脚、图形边栏等。

关于投影的考虑

通常认为Web设计应当灵活，而且应当能适应任何分辨率，这在大多数情况下都是对的。不过，投影样式不是Web样式，所以创建投影样式表时提前考虑一个特定的分辨率是合理的。举一个例子，（写作本书时）大多数投影仪都默认采用1024×768的分辨率。如果你知道要在这种大小上投影，就可以针对这种分辨率建立CSS样式，这是合理的。可以针对目标分辨率对字体大小、元素放置等等进行微调，以创建最佳的视觉体验。

为此，可能会为不同的分辨率创建不同的样式表：一个针对800×600，另一个针对1024×768，第三个针对1280×1024，这些都是最常见的情况。图14-10显示了一个面向1024×768分辨率的幻灯片。

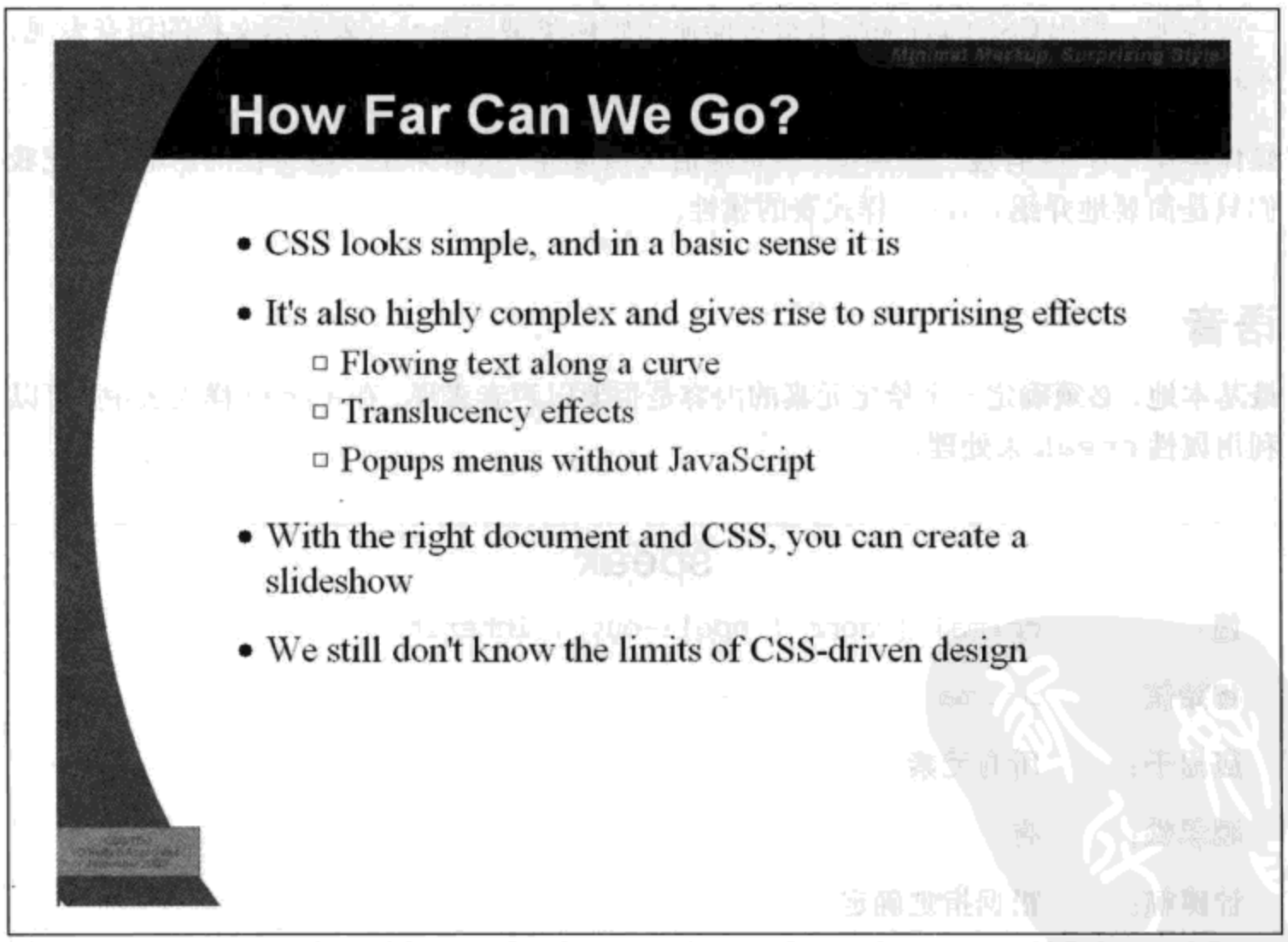

图14-10：一个充分应用样式的幻灯片

还有一点要记住，如果采用对比强烈的颜色，投影文档对观众来说往往更可读。这一点特别有意义，因为有些投影仪灯泡不如另外一些灯泡亮，如果灯泡比较暗，就需要更强烈的颜色对比。这就强调了一个事实，与正常的Web设计（以及其他方面）相比，投影更不能保证颜色的真实性。

声音样式

如果用户眼睛看不到，CSS支持的大多数视觉样式对他就没有任何帮助。对于这些用户，重要的不是加阴影或设置圆角，而是页面的具体文本内容——要让用户理解这些内容就必须用声音表现。Web内容的声音表现并非只对盲人有益。例如，嵌入在汽车里的用户代理可能会使用声音样式让驾驶指令等Web内容（甚至是驾驶员的email）读起来更生动。

为了满足这些用户的需求，CSS2中专门有一部分描述声音样式。写作本书时，有两个用户代理支持（至少在某种程度上支持）声音样式：Emacspeak和Fonix SpeakThis。尽管如此，CSS2.1还是废弃了媒体类型aural以及与之相关的所有属性。当前规范中有一段说明，指出CSS的将来版本很可能使用媒体类型speech来表示文档的语音表现，不过它没有提供任何细节。

媒体类型aural有过一些实现，但是随后又被废弃，这带来了一些奇怪的影响，因此我们只是简要地介绍aural样式表的属性。

语音

最基本地，必须确定一个给定元素的内容是否要以声音表现。在aural样式表中，可以利用属性speak来处理。

speak

值：	normal | none | spell-out | inherit
初始值：	normal
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

默认值normal指示应该将一个元素的内容读出。如果出于某种原因不能读出一个元素的内容，就要使用值none。尽管可以使用none禁止一个元素的声音表现，不过可以在其后代元素中覆盖这个值，相应地后代元素的内容可以用声音表现。在下面的例子中，文本“Navigation:”不会用声音表现，但是文本“Home”会被读出：

```
<div style="speak: none;">
```

```
Navigation:
<a href="home.html" style="speak: normal;">Home</a>
</div>
```

如果一个元素及其后代都不能用声音表现，应当使用display: none。在下面的例子中，div的内容都不能用声音表现（也不能在任何其他媒体中表现）：

```
<div style="display: none;">
Navigation:
<a href="home.html" style="speak: normal;">Home</a>
</div>
```

speak的第三个值是spell-out，这通常与缩略语或应当读出的其他内容结合使用。例如，以下标记片段要用声音表现为T-E-D-S，或读作“tee eee dee ess”：

```
<acronym style="speak: spell-out;" title="Technology Evangelism and
  Developer Support">TEDS</acronym>
```

标点符号和数字

还有两个属性影响元素内容的声音表现。第一个影响标点符号的表现，属性名为speak-punctuation（这个名字很贴切）。

speak-punctuation

值：	code | none | inherit
初始值：	none
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

根据默认值none，标点符号用声音表现时会作为适当长度的停顿，不过CSS并没有定义到底停顿多长。举一个例子，表示句号（即句子结尾）的停顿可能是表示逗号的停顿的两倍长。停顿长度往往取决于具体语言。

根据值code，会把标点符号具体读出，因此以下例子可能读作“avast comma ye scalawags exclamation point”：

```
<p style="speak-punctuation: code;">Avast, ye scalawags!</p>
```

再看另一个例子，以下片段可能用声音表现为“a left bracket href right bracket left curly brace color colon red semicolon right curly brace”：

```
<code style="speak-punctuation: code;">a[href] {color: red;}</code>
```

类似于标点符号的声音表现，speak-numeral定义了如何读出数字。

speak-numeral

值：	digits | continuous | inherit
初始值：	continuous
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

默认值continuous表示数字会被读作一个完整的数，值为digits时会逐个地读出数字。考虑以下例子：

```
<p style="speak-numeral: continuous;">23</p>
<p style="speak-numeral: digits;">23</p>
```

第一个段落的声音表现是“twenty-three”(23)，而第二个段落读作“two three”(2、3)。与标点符号类似，数字表现也依赖于不同的语言，但没有具体定义。

表标题的声音表现

在表的声音表现中，可能很容易失去控制，不知道单元格数据到底是什么意思。如果有一个12行的表，你正处在这个表中的第9行，该行中的第6个单元格是“21.77”，怎么才能记住第6列表示什么？甚至你是否还要记住该行中的数字与什么关联？表标题可以提供这些信息，在视觉上能很容易查看各列或各行的含义。为了在声音媒体中解决这个问题，CSS2引入了speak-header。

speak-header

值：	once | always | inherit
初始值：	once
应用于：	包含表标题信息的元素
继承性：	有
计算值：	根据指定确定

默认地，用户代理在遇到单元格时只会将表标题的内容表现一次。还有一种可能的做法，只要所表现的单元格与标题相关，就总会表现该表标题信息。

下面考虑一个简单的表作为例子：

```
<table id="colors">
<caption>Favorite Color</caption>
<tr id="headers">
<th>Jim</th><th>Joe</th><th>Jane</th>
</tr>
<tr>
<td>red</td><td>green</td><td>blue</td>
<td>
</tr>
</table>
```

如果没有应用任何样式，这个表的声音表现应当是“Favorite Color Jim Joe Jane red green blue”。由此可能可以得出这些是什么意思，不过想象一下，如果一个表包含10个或20个人最喜欢的颜色，你还能搞清楚吗？现在，假设向这个表应用以下样式：

```
#colors {speak-header: always;}
#headers {speak: none;}
```

现在表的声音表现则是“Favorite Color Jim red Joe green Jane blue”。这样理解起来就更容易了，而且不论表可能变得多大都不会影响理解。

注意： 文档语言本身会定义采用什么方法确定一个元素是表标题。标记语言也可能会提供一些办法将标题信息与元素或元素组关联 —— 例如，HTML4 中的 `scope` 和 `axis` 属性。

语速

除了能影响语音样式，CSS 还提供了一个 `speech-rate` 属性，这个属性用于设置用怎样的语速表现内容。

这个属性的值定义如下：

<number>

按每分钟单词数指定语速。不同语言的语速往往不同，因为有些语言要比其他语言说得更快。

`x-slow`

相当于每分钟 80 词。

speech-rate	
值：	<number> | x-slow | slow | medium | fast | x-fast | faster | slower | inherit
初始值：	medium
应用于：	所有元素
继承性：	有
计算值：	绝对数值

slow

相当于每分钟120词。

medium

相当于每分钟180~200词。

fast

相当于每分钟300词。

x-fast

相当于每分钟500词。

faster

将当前语速提高每分钟40词。

slower

将当前语速降低每分钟40词。

以下两个例子展示了语速的显著改变：

```
*.duh {speech-rate: x-slow;}
div#disclaimer {speech-rate: x-fast;}
```

CSS没有定义具体如何改变语速。用户代理可以拉长每一个词，延长词之间的停顿，或者二者兼而有之。

音量

对于声音媒体的表现，最重要的方面之一是用户代理所产生声音的音量。这就引入了一个名字很贴切的属性：volume。

volume	
值：	<number> | <percentage> | silent | x-soft | soft | medium | loud | x-loud | inherit
初始值：	medium
应用于：	所有元素
继承性：	有
计算值：	绝对数值

这个属性的值定义如下：

<number>

提供音量的数值表示。0对应于可听到的最小音量，这与静音不同；100对应于可接受的最大音量。

<percentage>

计算为继承值的一个百分数。

silent

不产生声音，这与数字值0不同。这相当于声音领域的visibility: hidden。

x-soft

对应于数字值0。

soft

对应于数字值25。

medium

对应于数字值50。

loud

对应于数字值75。

x-loud

对应于数字值100。

要注意重要的一点（把这句话速念5遍！），volume值定义的是平均音量，而不是所产生的每个声音的确切音量。因此，如果一个元素指定为volume: 50;，其内容可能并不完全用这个音量表现，有些可能会在此之上，而有些可能在这个水平之下，特别是当声音严重变形或者声音的变化幅度很大时，这种情况更为突出。

数值范围往往由用户配置，因为只有单个用户能确定自己的可听到的最小音量（0）和可接受的最大音量（100）。举一个例子，一个用户可能认为可听到的最小音量是34dB，可接受的最大音量是84dB。这说明，在0~100之间有50dB的范围，值每增1就意味着平均音量有半分贝的提升。换句话说，volume: soft;将被解释为平均音量46.5dB。

百分数值的作用类似于font-size中的百分数：会根据父元素的值增加或降低。例如：

```
div.marine {volume: 60;}
big {volume: 125%;}

<div class="marine">
When I say jump, I mean <big>rabbit</big>, you maggots!
</div>
```

根据前面描述的声音范围，div元素的内容将用64dB的平均音量说出。只有big元素例外，其音量是父元素值60的125%，计算得到75，相当于71.5dB。

如果指定百分数值使元素的音量计算值超出了0~100的范围，这个值会“剪裁”为最接近的边界值（0或100）。假设将前面的样式改为：

```
div.marine {volume: 60;}
big {volume: 200%;}
```

这会使big元素的volumn值计算为120；这个值会“剪裁”为100，在此对应为平均音量84dB。

以这种方式定义音量的好处是，相同的样式表可以在不同环境中应用。例如，图书馆里的0和100设置可能就与汽车里的相应设置不同，不过这些值在各种环境中都对应同样的听觉效果。

指定声音

至此，我们已经讨论了影响声音表现的很多方法，但是目前还没有介绍如何选择表现内容时所用的声音。类似于font-family，CSS定义了一个名为voice-family的属性。

与font-family类似，利用voice-family，创作人员可以提供表现元素内容时使用的声音列表（各声音用逗号分隔）。用户代理在这个表中查找第一个声音，如果可用则选择这个声音。如果不可用，用户代理再在列表中查找下一个声音，依次类推，直到找到某个特定声音，或者查完所有指定的声音。

根据值语法的定义方式，可以按任何顺序提供多个特定或通用声音系列。因此，可以用一个特定系列而不是通用系列结束。例如：

voice-family	
值：	[[<specific-voice> | <generic-voice>],]* [<specific-voice> | <generic-voice>] | inherit
初始值：	取决于用户代理
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

```
h1 {voice-family: Mark, male, Joe;}
```

CSS2.x 没有定义通用系列值，不过注意 male、female 和 child 都是被允许的。因此，可以如下为 XML 文档中的元素指定样式：

```
rosen {voice-family:  Gary, Scott, male;}
guild {voice-family: Tim, Jim, male;}
claud {voice-family: Donald, Ian, male;}
gertr {voice-family: Joanna, Susan, female;}
albert {voice-family: Bobby, Paulie, child;}
```

要表现一个给定元素，所选的具体声音会影响用户对该元素的感知，因为有些声音的音高要高于或低于另外一些声音，或者可能有些单调。CSS 提供了一些方法来影响声音的这些方面。

改变声音

一旦让用户代理使用某种特定声音来表现内容，你可能还想改变它的某些方面。例如，某个声音听上去还不错，不过音高对你来说有些过高。另外一个声音可能有些太“变化”了，不过其他方面倒能满足你的需要。CSS 定义了一些属性来影响声音的所有方面。

改变音高

显然，不同的声音有不同的音高。作为最基本的例子，男声的平均音高大约在 120Hz，而女声的平均音高大约在 210Hz。因此，每个声音系列都有自己的默认音高，CSS 允许创作人员使用属性 pitch 来改变音高。

关键字 x-low 到 x-high 没有明确的定义，所以只能说其中每个关键字对应的音高比其前一个关键字对应的音高要高。这类似于 font-size 关键字 xx-small 到 xx-large，这些关键字也没有明确定义，不过每个关键字对应的字体大小都必须比前一个大。

pitch	
值：	<frequency> | x-low | low | medium | high | x-high | inherit
初始值：	medium
应用于：	所有元素
继承性：	有
计算值：	绝对频率值

频率值则是另一回事。如果定义一个明确的音高频率，声音就会相应调整，使其平均音高与所提供的值一致。例如：

```
h1 {pitch: 150Hz;}
```

如果使用了一个预料之外的声音，其效果可能是戏剧性的。下面考虑一个例子，其中为元素指定了两个 voice-family 和一个音高频率：

```
h1 {voice-family: Jethro, Susie; pitch: 100Hz;}
```

对于这个例子，假设“Jethro”的默认音高是110Hz，“Susie”的默认音高是200Hz。如果“Jethro”被选中，则h1元素会用比正常稍低的声音读出。如果“Jethro”不可用，而是使用了“Susie”，与默认声音相比变化会很显著，甚至可能很怪异。

除了元素表现时使用的音高，还可以使用属性pitch-range来影响音高的变化范围。

pitch-range	
值：	<number> | inherit
初始值：	50
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

pitch-range的目的是加大或减少给定声音的变形。音高范围越低，所有音高与平均音高就越接近，这会得到一种单调的声音。默认值50可以得到“正常”变形。如果比这个值高，声音的“变化”程度（颤抖度）就会增加。

重音与音色

与pitch-range对应的属性是stress，这个属性有助于创作人员减弱或加强语言的重音模式。

stress

值：	<number> | inherit
初始值：	50
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

从某种程度上说，所有人类语言都有重音模式。例如，在英语中，句子里的不同部分要求有不同的重音。前面这个句子（In English, for example, sentences have different parts that call for different stress.）可以写作：

```
<sentence>
  <primary>In English,</primary>
  <tertiary>for example,</tertiary>
  <secondary>sentences have different parts that call for
different stress.</secondary>
</sentence>
```

为了定义句子中各部分的重音级别，样式表可能指出：

```
primary {stress: 65;}
secondary {stress: 50;}
tertiary {stress: 33;}
```

这会使句子中不太重要的部分减弱加重，而比较重要的部分将更为强调。stress值取决于具体语言，所以同一个值可能会得到不同的重音级别和模式。CSS没有定义这种差别（你现在应该不会对此感到奇怪了）。

richness在许多方面与stress很类似。

声音的richness值越高，就越“明亮”，它的“含义”就越丰富。值较低，声音就更轻柔、“更流畅”（引自CSS2规范）。因此，演员独白时可能指定richness: 80;，而在一旁小声说话时可能指定为richness: 25;。

richness	
值：	<number> | inherit
初始值：	50
应用于：	所有元素
继承性：	有
计算值：	根据指定确定

停顿和提示

在视觉设计中，可以为元素指定额外的外边距，使之与其他元素分开，或者通过增加边框让用户特别注意这个元素。这会把用户的目光吸引到这些元素上。在声音表现中，与之最接近的是在元素周围插入停顿和声音提示。

停顿

所有口语都依赖于某种形式的停顿。单词、短语和句子之间的短暂停顿对于含义的理解至关重要，甚至不亚于单词本身。从某种意义上说，停顿就相当于声音领域的外边距，因为停顿和外边距都用于将元素与其周围内容分开。在CSS中，可以用三个属性向文档中插入停顿：pause-before、pause-after和pause。

pause-before、pause-after	
值：	<time> | <percentage> | inherit
初始值：	0
应用于：	所有元素
继承性：	无
计算值：	绝对时间值

采用<time>值格式，可以用秒数或毫秒数来表示停顿长度。例如，假设希望在一个h1元素后整整停顿2秒钟。以下的任何一个规则都可以达到这个效果：

```
h1 {pause-after: 2s;}
h1 {pause-after: 2000ms;}  /* the same length of timé as '2s' */
```

百分数要稍微复杂一些，因为它们要相对于speech-rate的隐含度量值来计算。尽管你不愿意，但确实如此！下面来看这是如何计算的。首先，考虑以下规则：

```
h1 {speech-rate: 180;}
```

这意味着所有h1元素在用声音表现时语速大约为每秒钟3个词。再考虑下面的规则：

```
h1 {speech-rate: 180; pause-before: 200%;}
```

百分数根据平均单词长度计算。在这里，念一个词要用333.33毫秒，所以其200%就是666.66毫秒。换个角度讲，每个h1之前会有大约2/3秒的停顿。如果修改这个规则，让speech-rate值为120，这个停顿则是整整1秒。

简写属性pause将pause-before和pause-after综合在一起。

pause

值：	[[<time> | <percentage>]{1,2}] | inherit
初始值：	0
应用于：	所有元素
继承性：	无
计算值：	见单个属性（pause-before等）

如果只提供了一个值，它会同时作为元素之前和之后的停顿值。如果提供了两个值，则第一个值是元素前的停顿，第二个是元素后的停顿值。因此，以下规则都是等价的：

```
pre {pause: 1s;}
pre {pause: 1s 1s;}
pre {pause-before: 1s; pause-after: 1s;}
```

提示

如果停顿还不足以让人关注一个元素，还可以在元素前后插入声音提示，这在声音领域就相当于边框。类似于停顿属性，共有3个提示属性：cue-before、cue-after和cue。

通过提供一个声音资源的URI，用户代理要加载该资源，并在元素前（或后）播放这个声音。假设希望在文档中每个未访问的超链接之前插入时钟报时声，每个已访问链接之前插入哔哔声。规则如下所示：

```
a:link {cue-before: url(chime.mp3);}
a:visited {cue-before: url(beep.wav);}
```

cue-before、cue-after	
值：	<uri> | none | inherit
初始值：	none
应用于：	所有元素
继承性：	无
计算值：	对于<uri>值，为绝对URI；否则，为none

不难想象简写属性cue的做法。

cue	
值：	[<cue-before> || <cue-after>] | inherit
初始值：	none
应用于：	所有元素
继承性：	无
计算值：	见单个属性（cue-before等）

与pause类似，如果只为cue提供一个值，意味着这个值会同时用于元素前和元素后提示。提供两个值则意味着第一个用于元素前提示，第二个用于元素后提示。因此，以下规则是等价的：

```
a[href] {cue: url(ping.mp3);}
a[href] {cue: url(ping.mp3) url(ping.mp3);}
a[href] {cue-before: url(ping.mp3); cue-after: url(ping.mp3);}
```

停顿、提示和生成内容

停顿和提示都在生成内容“之外”播放。考虑以下规则：

```
h1 {cue: url(trumpet.mp3);}
h1:before {content: "Behold! ";}
h1:after {content: ". Verily!";}

<h1>The Beginning</h1>
```

这个元素的声音表现大致是“(喇叭声) Behold! The Beginning. Verily!(喇叭声)”。

CSS 没有指定停顿在提示“之外”还是提示在停顿“之外”，所以声音用户代理在这方面的行为是无法预测的。

背景声音

视觉元素可以有背景，所以声音元素也应该有背景，这才公平。在声音媒体中，背景是指读出元素的同时播放一个声音。用于完成这个任务的属性是 play-during。

play-during

值：	<uri> [mix || repeat]? | auto | none | inherit
初始值：	auto
应用于：	所有元素
继承性：	无
计算值：	对于 <uri> 值，为绝对 URI；否则，根据指定确定

最简单的例子是在元素的声音表现开始时播放一个声音：

```
h1 {play-during: url(trumpets.mp3);}
```

根据这个规则，读出任何 h1 元素时都会播放声音文件 *trumpets.mp3*。这个声音文件只播放一次。如果它比念出元素内容所需的时间要短，这个声音将在元素念完之前停止。如果比念出元素内容所需的时间长，一旦念完元素的所有内容，这个声音就会立即停止。

如果希望一个声音在读元素期间一直重复，可以增加关键字 repeat。这在声音领域就相当于 background-repeat: repeat:

```
div.ocean {play-during: url(wave.wav) repeat;}
```

类似于可见的背景，默认情况下并没有背景声音。考虑以下情况：

```
a:link {play-during: url(chains.mp3) repeat;}
em {play-during: url(bass.mp3) repeat;}

<a href="http://www.example.com/">This is a <em>really great</em> site!</a>
```

在此，*chains.mp3* 将会在读链接文本时反复播放，但 em 元素的文本除外。读出 em 元素的文本时，听不到 chains 声音，而是会听到 *bass.mp3*。父元素的背景声音是听不到的，这与视觉领域类似，如果两个元素都有可见的背景，将无法看到 em 元素底下的父元素背景。

如果想让这两个声音混合，可以使用关键字mix：

```
a:link {play-during: url(chains.mp3) repeat;}
em {play-during: url(bass.mp3) repeat mix;}
```

现在，*chains.mp3*在读所有链接文本时都能听到，包括em元素的文本。对于em元素，会同时听到*chains.mp3*和*bass.mp3*，这两个声音会混合在一起。

尽管背景声音与可见背景基本类似，但确实有一点不同，这里还有一个值none。这个关键字会去掉所有声音，包括可能属于祖先元素的声音。因此，根据以下规则，em文本将根本没有背景声音——*bass.mp3*和*chains.mp3*都听不到：

```
a:link {play-during: url(chains.mp3) repeat;}
em {play-during: none;}

<a href="http://www.example.com/">This is a <em>really great</em> site!</a>
```

声音定位

如果只有一个人在讲话，声音则从空间中的一点发出，当然除非这个人在到处移动。在一个多人参与的交谈中，每个声音会来自空间中的不同点。

利用一些高端声响系统和3D声音，应该可以在这个空间内定位声音。CSS2.x定义了两个属性来完成这个任务，其中一个属性定义了发声源在水平面上的角度，第二个属性定义了发声源在垂直面上的角度。声音沿水平面的位置由属性azimuth处理。

azimuth

值：	<angle> | [[left-side | far-left | left | center-left | center | center-right | right | far-right | right-side] || behind] | leftwards | rightwards | inherit
初始值：	center
应用于：	所有元素
继承性：	有
计算值：	标准化角度

角度值可以有3种单位：deg（度）、grad（梯度）和rad（弧度）。这些单位类型的可取值范围分别是0~360deg、0~400grad和0~6.2831853rad。允许有负值，不过它们会重新计算为正值。例如，-45deg等价于315deg（360-45），-50grad等同于grad。

大部分关键字都等价于某个角度值。如表14-1所示，这里使用度为单位来描述角度值，图14-11给出了示意图。表14-1的最后一列显示了第一列关键字与behind结合使用时的等价角度值。

表14-1：azimuth关键字和等价的角度

关键字	角度	behind
center	0	180deg -180deg
center-right	20deg -340deg	160deg -200deg
right	40deg -320deg	140deg -220deg
far-right	60deg -300deg	120deg -240deg
right-side	90deg -270deg	90deg -270deg
center-left	340deg -20deg	200deg -160deg
left	320deg -40deg	220deg -140deg
far-left	300deg -60deg	240deg -120deg
left-side	270deg -90deg	270deg -90deg

注意：behind不能与角度值结合使用。它只能与表14-1中所列的某个关键字连用。

除了表14-1中所列的关键字，还有另外两个关键字：leftwards和rightwards。前者的效果是从当前azimuth角度值减去20deg，后者是向当前值增加20deg。例如：

```
body {azimuth: right-side;}  /* equivalent to 90deg */
h1 {azimuth: leftwards;}
```

h1元素的azimuth计算角度值为70deg。现在考虑以下情况：

```
body {azimuth: behind;}  /* equivalent to 180deg */
h1 {azimuth: leftwards;}  /* computes to 160deg */
```

根据这些规则，leftwards的作用是使声音移向右边，而不是左边。这很奇怪，不过CSS2就是这样规定的。类似地，前例中使用rightwards会让h1元素的发声源向左移20度。

elevation与azimuth很类似，不过更简单一些，它指定声音在垂直面上的位置。

类似于azimuth，elevation可以接受度、梯度和弧度角度值。与角度等价的3个关键字是above（90度），level（0）和below（−90度），如图14-12所示。

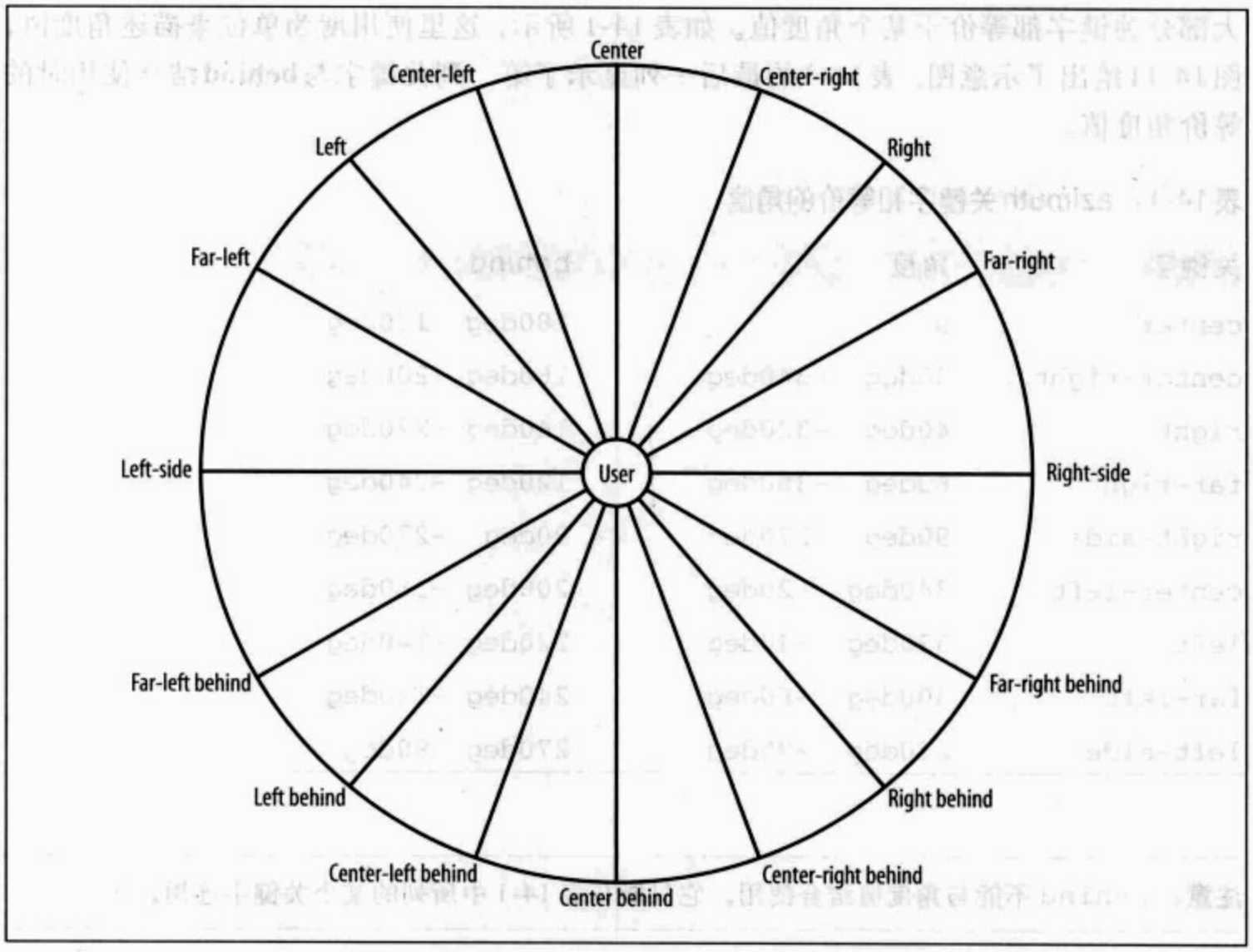

图14-11：水平面下视图

elevation	
值：	<angle> | below | level | above | higher | lower | inherit
初始值：	level
应用于：	所有元素
继承性：	有
计算值：	标准化角度

相对位置关键字higher和lower分别将当前俯仰角增加或减少10度。因此，在下面的例子中，作为body子元素的h1元素会放在水平之上10度的位置：

```
body {elevation: level;}  /* equivalent to 0 */
body > h1 {elevation: higher;}
```

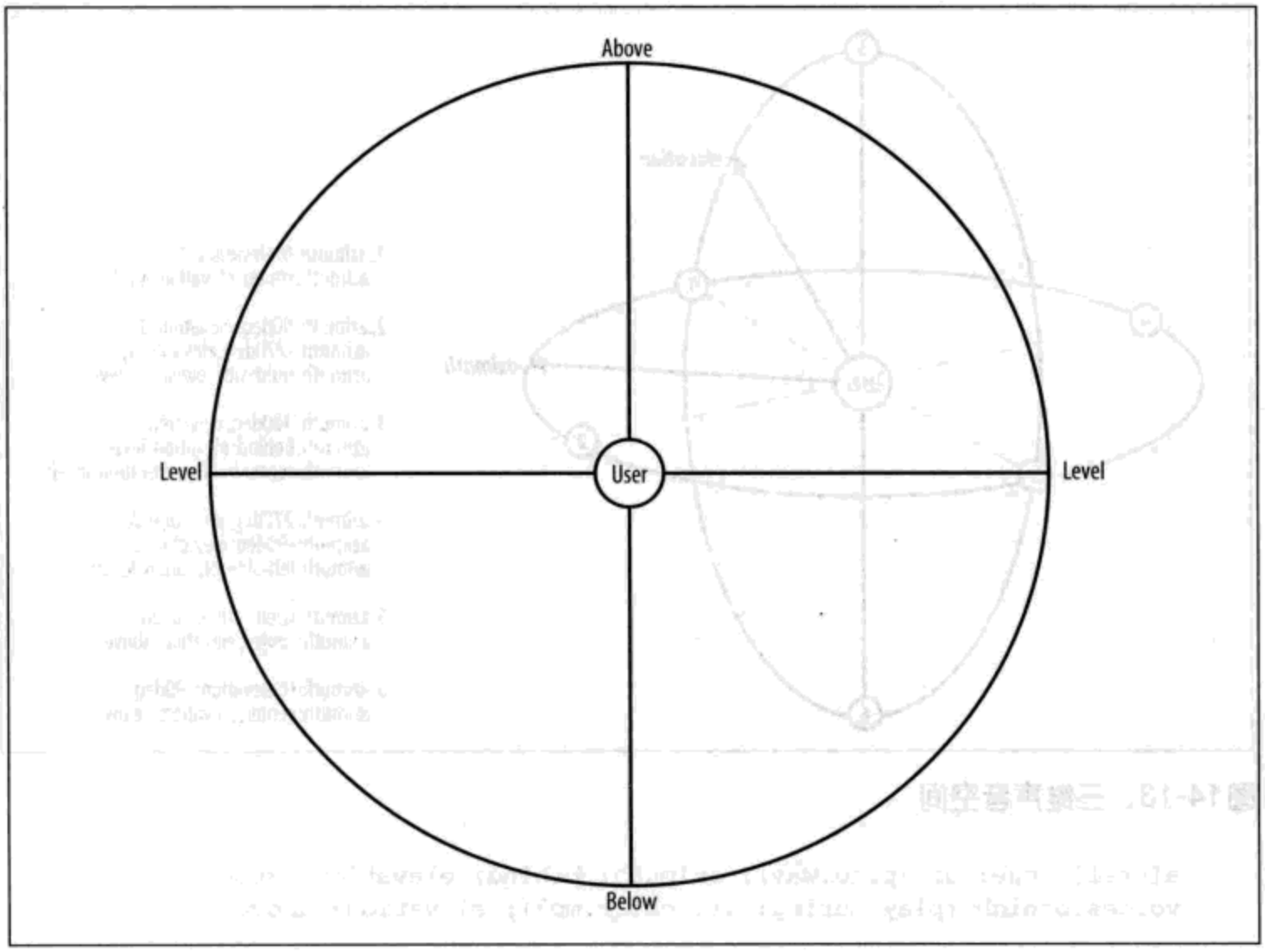

图14-12：垂直面右视图

结合 azimuth 和 elevation

结合azimuth和elevation的值，它们定义了以用户为中心的一个假想球体中的一个点。图 14-13 展示了这样一个球，其中有一些刻度点和值，声音将放在这些位置。

假设你坐在一个椅子上，有一个点在水平面上位于你的正前方和右边的中间（水平 45 度），另外在垂直面上位于水平面与你头顶的中间（垂直 45 度）。这个点可以描述为azimuth: 45deg; elevation: 45deg;。现在假设有一个声源在同样的仰角上，不过在水平面上位于你左边和后面的中间。这个声源可以用以下任何一种方式描述：

```
azimuth: -135deg; elevation: 45deg;
azimuth: 215deg; elevation: 45deg;
azimuth: left behind; elevation: 45deg;
```

定位声音可以帮助用户将提示与其他声音源区分开，或者可能有助于创建单独定位的特殊声音：

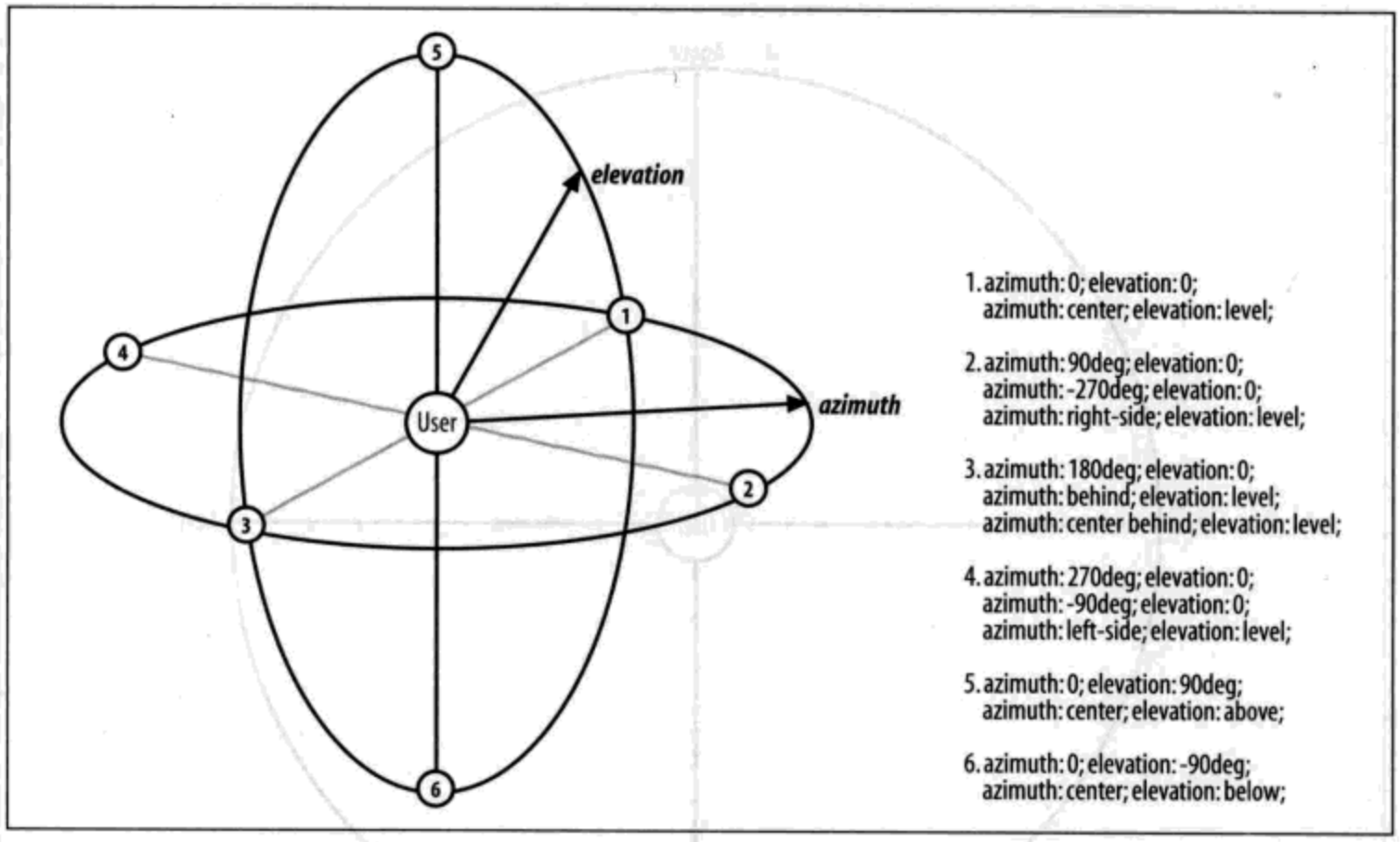

图14-13：三维声音空间

```
a[href] {cue: url(ping.wav); azimuth: behind; elevation: 30deg;}
voices.onhigh {play-during: url(choir.mp3); elevation: above;}
```

小结

尽管Web开发的第一阶段主要关注于视觉领域，不过随后还需要用其他媒体提供Web内容，正因如此，CSS中引入了特定于媒体的样式。对于同一个文档，可以采用最适合不同输出媒体的方式定制文档的表现，这种能力确实非常强大。尽管最常见的应用是为文档创建“打印友好”的样式，不过我们已经看到，还可以利用投影样式使用Opera创建幻灯片。

尽管声音样式对盲人用户非常有用，但写作本书时只有两个程序对此提供支持（而且还只是CSS中有关声音样式的一部分），另外CSS2.x中定义的媒体类型`aural`不会继续在CSS的将来版本中沿用。相反，在文档的声音表现方面，将来可能会采用`speech`媒体类型。

附录 A

属性参考

视觉媒体

background

这是一种简写方法，可以在一个声明中表述所有背景属性。通常建议使用这个属性而不是分别使用单个属性，因为这个属性在较老的浏览器中能得到更好的支持，而且需要键入的字母也更少。

值： [<background-color> || <background-image> || <background-repeat> || <background-attachment> || <background-position>] | inherit

初始值： 根据单个属性

应用于： 所有元素

继承性： 无

百分数： <background-position> 允许的值

计算值： 见单个属性

background-attachment

这个属性定义了文档滚动时背景图像是否随着元素滚动。它可以用于创建“对齐”的背景；有关的更多详细信息参见第 9 章。

值： scroll | fixed | inherit

初始值： scroll

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

background-color

这会为元素的背景设置一种纯色。这种颜色会填充元素的内容、内边距和边框区域，扩展到元素边框的外边界。如果边框有透明部分（如虚线边框），会透过这些透明部分显示出背景色。

值： <color> | transparent | inherit

初始值： transparent

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

background-image

这会在元素的背景中放置一个图像。根据background-repeat属性的值，图像可以无限平铺、沿着某个轴（x轴或y轴）平铺，或者根本不平铺。初始背景图像（原图像）根据background-position属性的值放置。

值： <uri> | none | inherit

初始值： none

应用于： 所有元素

继承性： 无

计算值： 绝对URI

background-position

这个属性设置背景原图像（由background-image定义）的位置；背景图像如果要重复，将从这一点开始。

值： [[<percentage> | <length> | left | center | right] [<percentage> | <length> | top | center | bottom]?] | [[left | center | right] || [top | center | bottom]] | inherit

初始值： 0% 0%

应用于： 块级和替换元素

继承性： 无

百分数： 相对于元素和原图像上的相应点

计算值： 如果指定 <length>，则为绝对长度偏移；否则，为百分数值

---

background-repeat

这定义了背景图像的平铺模式。注意，重复值为axis-related时会沿着两个轴同时重复。将从原图像开始重复，原图像由 background-image 定义，并根据 background-position 的值放置。

值： repeat | repeat-x | repeat-y | no-repeat | inherit

初始值： repeat

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

---

border

这是一个简写属性，定义了一个元素边框的宽度、颜色和样式。注意，这些值都不是必要的，如果忽略边框样式，将不应用任何边框，因为默认边框样式是 none。

值： [<border-width> || <border-style> || <border-color>] | inherit

初始值： 见单个属性

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

---

border-bottom

这个简写属性定义了一个元素下边框的宽度、颜色和样式。与 border 类似，如果忽略边框样式，则不会出现边框。

值： [<border-width> || <border-style> || <border-color>] | inherit

初始值： 对简写属性未定义

应用于： 所有元素

继承性： 无

计算值： 见单个属性（border-width等）

border-bottom-color

这个属性为一个元素下边框的可见部分设置颜色。只能定义纯色，而且只有当边框的样式是一个非none或hidden的值时边框才可能出现。

值： <color> | transparent | inherit

初始值： 元素的color值

应用于： 所有元素

继承性： 无

计算值： 如果没有指定任何值，则使用该元素的color属性计算值；否则，根据指定确定

border-bottom-style

这个属性定义了一个元素下边框的样式。只有当这个值不是none时边框才可能出现。在CSS1中，HTML用户代理只需支持solid和none。

值： none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset | inherit

初始值： none

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

border-bottom-width

这会设置元素下边框的宽度，只有当边框样式不是none时才起作用。如果边框样式是none，边框宽度实际上会重置为0。不允许指定负长度值。

值： thin | medium | thick | <length> | inherit

初始值： medium

应用于： 所有元素

继承性： 无

计算值： 绝对长度；如果边框样式是none或hidden，则为0

---

border-color

这个简写属性会设置一个元素所有边框中可见部分的颜色，或者为4个边分别设置不同的颜色。要记住，边框的样式不能为none或hidden，否则边框不会出现。

值： [<color> | transparent]{1,4} | inherit

初始值： 对简写属性未定义

应用于： 所有元素

继承性： 无

计算值： 见单个属性（border-top-color等）

---

border-left

这个简写属性会定义一个元素左边框的宽度、颜色和样式。与border类似，如忽略边框样式，则不会出现边框。

值： [<border-width> || <border-style> || <border-color>] | inherit

初始值： 对简写属性未定义

应用于： 所有元素

继承性： 无

计算值： 见单个属性（border-width等）

---

border-left-color

这个属性为一个元素左边框的可见部分设置颜色。只能定义纯色，而且只有当边框的样式是一个非none或hidden的值时边框才可能出现。

值： <color> | transparent | inherit

初始值： 元素的color值

应用于： 所有元素

继承性： 无

计算值： 如果没有指定任何值，则取该元素的color属性计算值；否则，根据指定确定

border-left-style

这个属性定义元素左边框的样式。只有当值不是none时边框才可能出现。在CSS1中，HTML用户代理只需支持solid和none。

值： none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset | inherit

初始值： none

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

border-left-width

这个属性设置元素左边框的宽度，只有当边框样式不是none时才起作用。如果边框样式是none，边框宽度实际上会重置为0。不允许指定负长度值。

值： thin | medium | thick | <length> | inherit

初始值： medium

应用于： 所有元素

继承性： 无

计算值： 绝对长度；如果边框样式是none或hidden，则为0

border-right

这个简写属性定义一个元素右边框的宽度、颜色和样式。类似于border，如果忽略边框样式，则不会出现边框。

值： [<border-width> || <border-style> || <border-color>] | inherit

初始值： 对简写属性未定义

应用于： 所有元素

继承性： 无

计算值： 见单个属性（border-width等）

border-right-color

这个属性为一个元素右边框的可见部分设置颜色。只能定义纯色，而且边框的样式必须是一个非none或hidden的值，边框才可能出现。

值： <color> | transparent | inherit

初始值： 元素的color值

应用于： 所有元素

继承性： 无

计算值： 如果没有指定任何值，则取该元素的color属性计算值；否则，根据指定确定

---

border-right-style

这个属性定义一个元素右边框的样式。只有当这个值不是none时边框才可能出现。在CSS1中，HTML用户代理只需支持solid和none。

值： none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset | inherit

初始值： none

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

---

border-right-width

这个属性设置一个元素右边框的宽度，如果当边框样式不是none时才起作用。如果边框样式是none，边框宽度实际上会重置为0。不允许指定负长度值。

值： thin | medium | thick | <length> | inherit

初始值： medium

应用于： 所有元素

继承性： 无

计算值： 绝对长度；如果边框样式是none或hidden，则为0

---

border-style

这个简写属性用于设置元素所有边框的样式，或者单独地为各边设置边框样式。边框的样式值不能是none，否则边框不会显示。

值： [none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset]{1,4} | inherit

初始值： 对简写属性未定义

应用于： 所有元素

继承性： 无

计算值： 见单个属性（border-top-style等）

说明： HTML用户只需支持solid和none；余下的值（除了hidden）都可以解释为solid

---

border-top

这个简写属性定义了一个元素上边框的宽度、颜色和样式。类似于border，如果忽略边框样式，则不会显示边框。

值： [<border-width> || <border-style> || <border-color>] | inherit

初始值： 对简写属性未定义

应用于： 所有元素

继承性： 无

计算值： 见单个属性（border-width等）

---

border-top-color

这个属性设置一个元素上边框可见部分的颜色。只能定义纯色，只有当边框样式不是none或hidden时边框才可能出现。

值： <color> | transparent | inherit

初始值： 元素的color值

应用于： 所有元素

继承性： 无

计算值： 如果没有指定任何值，则取该元素的color属性计算值；否则，根据指定确定

---

border-top-style

这个属性定义一个元素上边框的样式。只有当这个值不是none时边框才可能出现。在CSS1中，HTML用户只需支持solid和none。

值： none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset | inherit

初始值： none

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

---

border-top-width

这个属性会设置元素上边框的宽度，只有当边框样式不是none时才起作用。如果样式为none，宽度实际上会重置为0。不允许指定负长度值。

值： thin | medium | thick | <length> | inherit

初始值： medium

应用于： 所有元素

继承性： 无

计算值： 绝对长度；如果边框样式是none或hidden，则为0

---

border-width

这个简写属性可以用于为元素的所有边框设置宽度，或者单独地为各边边框设置宽度。只有当边框样式不是none时边框宽度才起作用。如果边框样式是none，边框宽度实际上会重置为0。不允许指定负长度值。

值： [thin | medium | thick | <length>]{1,4} | inherit

初始值： 对简写属性未定义

应用于： 所有元素

继承性： 无

计算值： 见单个属性（border-top-style等）

---

bottom

这个属性定义定位元素下外边距边界与其包含块下边界之间的偏移。

值： <length> | <percentage> | auto | inherit

初始值： auto

应用于： 定位元素（即position值不为static的元素）

继承性： 无

百分数： 相对于包含块的高度

计算值： 对于相对定位元素，见说明；对于static元素，为auto；对于长度值，则为相应的绝对长度；对于百分数值，为指定值；否则，为auto。

说明： 对于相对定位元素，如果bottom和top都是auto，其计算值则都是0；如果其中一个为auto，则取另一个值的相反数；如果二者都不是auto，bottom将取top值的相反数。

---

clear

这个属性定义了元素的哪些边上不允许出现浮动元素。在CSS1和CSS2中，这是通过自动为清除元素（即设置了clear属性的元素）增加上外边距实现的。在CSS2.1中，会在元素上外边距之上增加清除空间，而外边距本身并不改变。不论哪一种做法，最终结果都一样，如果声明为左边或右边清除，会使元素的上外边框边界刚好在该边上浮动元素的下外边距边界之下。

值： left | right | both | none | inherit

初始值： none

应用于： 块级元素

继承性： 无

计算值： 根据指定确定

---

clip

这个属性用于定义一个剪裁矩形，对于一个绝对定位元素，在这个矩形内的内容才可见。出了这个剪裁区域的内容会根据overflow的值来处理。剪裁区域可能比元素的内容区大，也可能比内容区小。

值： rect(*top*,*right*,*bottom*,*left*) | auto | inherit

初始值： auto

应用于： 绝对定位元素（在CSS2中，clip应用于块级和替换元素）

继承性： 无

计算值： 对于矩形，计算值为表示剪裁矩形4个边的4个计算长度；否则，根据指定确定

color

这个属性设置一个元素的前景色（在HTML表现中，就是元素文本的颜色）；光栅图像不受color影响。这个颜色还会应用到元素的所有边框，除非被border-color或另外某个边框颜色属性（border-top-color等）覆盖。

值： <color> | inherit

初始值： 特定于用户代理

应用于： 所有元素

继承性： 有

计算值： 根据指定确定

content

这个属性用于定义元素之前或之后放置的生成内容。默认地，这往往是行内内容，不过该内容创建的框类型可以用属性display控制。

值： normal | [<string> | <uri> | <counter> | attr(<identifier>) | open-quote | close-quote | no-open-quote | no-close-quote]+ | inherit

初始值： normal

应用于： :before和:after伪元素

继承性： 无

计算值： 对于<uri>值，为绝对URI；对于属性引用，为结果串；否则，根据指定确定

counter-increment

利用这个属性，计数器可以递增（或递减）某个值，这可以是正值或负值。如果没有提供<integer>，则默认为1。

值： [<identifier> <integer>?]+ | none | inherit

初始值： 取决于具体的用户代理

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

counter-reset

利用这个属性，计数器可以设置或重置为任何值，可以是正值或负值。如果没有提供<integer>，则默认为0。

值： [<identifier> <integer>?]+ | none | inherit

初始值： 取决于具体的用户代理

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

cursor

这个属性定义了鼠标指针放在一个元素边界范围内时所用的光标形状（不过CSS2.1没有定义由哪个边界确定这个范围）。

值： [[<uri>,]* [auto | default | pointer | crosshair | move | e-resize | ne-resize | nw-resize | n-resize | se-resize | sw-resize | s-resize | w-resize| text | wait | help | progress]] | inherit

初始值： auto

应用于： 所有元素

继承性： 有

计算值： 对于<uri>值，为绝对URI；否则，根据指定确定

direction

这个属性指定了块的基本书写方向，以及针对Unicode双向算法的嵌入和覆盖方向。不支持双向文本的用户代理可以忽略这个属性。

值： ltr | rtl | inherit

初始值： ltr

应用于： 所有元素

继承性： 有

计算值： 根据指定确定

display

这个属性用于定义建立布局时元素生成的显示框类型。对于HTML等文档类型，如果使用display不谨慎会很危险，因为可能违反HTML中已经定义的显示层次结构。对于XML，由于XML没有内置的这种层次结构，所以display是绝对必要的。

值： none | inline | block | inline-block | list-item | run-in | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | inherit

初始值： inline

应用于： 所有元素

继承性： 无

计算值： 对于浮动元素、定位元素和根元素可变（见CSS2.1第9.7节），否则，根据指定确定

说明： CSS2中有值compact和marker，不过由于缺乏广泛支持，已经从CSS2.1中去除

float

这个属性定义元素在哪个方向浮动。以往这个属性总应用于图像，使文本围绕在图像周围，不过在CSS中，任何元素都可以浮动。浮动元素会生成一个块级框，而不论它本身是何种元素。浮动非替换元素要指定一个明确的宽度；否则，它们会尽可能地窄。

值： left | right | none | inherit

初始值： none

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

font

这个简写属性用于一次设置元素字体的两个或更多方面。使用icon等关键字可以适当地设置元素的字体，使之与用户计算环境中的某个方面一致。注意，如果没有使用这些关键字，至少要指定字体大小和字体系列。

值： [[<font-style> || <font-variant> || <font-weight>]? <font-size> [/ <line-height>]? <font-family>] | caption | icon | menu | message-box | small-caption | status-bar | inherit

初始值： 根据单个属性

应用于： 所有元素

继承性： 有

百分数： 对于<font-size>，要相对于父元素计算；对于<line-height>，则相对于元素的<font-size>计算

计算值： 见单个属性（font-style等）

---

font-family

这个属性定义用于元素文本显示的字体系列。注意，使用某种特定的字体系列（如Geneva）完全取决于用户机器上该字体系列是否可用；这个属性没有指示任何字体下载。因此，强烈推荐使用一个通用字体系列名作为后路。

值： [[<family-name> | <generic-family>],]* [<family-name> | <generic-family>] | inherit

初始值： 特定于用户代理

应用于： 所有元素

继承性： 有

计算值： 根据指定确定

---

font-size

这个属性设置一个元素的字体大小。注意，实际上它设置的是字体中字符框的高度；实际的字符字形可能比这些框高或矮（通常会更矮）。各关键字对应的字体必须比下一个最小关键字相应字体要高，并且要小于下一个最大关键字对应的字体。不允许指定负长度和百分数值。

值： xx-small | x-small | small | medium | large | x-large | xx-large | smaller | larger | <length> | <percentage> | inherit

初始值： medium

应用于： 所有元素

继承性： 有

百分数： 根据父元素的字体大小计算

计算值： 绝对长度

---

font-style

这个属性设置使用斜体、倾斜或正常字体。斜体文本通常定义为字体系列中的一个单独的字体。理论上讲，用户代理可以根据正常字体计算一个倾斜字体。

值： `italic | oblique | normal | inherit`

初始值： `normal`

应用于： 所有元素

继承性： 有

计算值： 根据指定确定

---

font-variant

这个属性主要用于定义小型大写字母文本。理论上，用户代理可以根据正常字体计算出小型大写字母字体。

值： `small-caps | normal | inherit`

初始值： `normal`

应用于： 所有元素

继承性： 有

计算值： 根据指定确定

---

font-weight

这个属性用于设置显示元素的文本中所用的字体加粗。数字值`400`相当于关键字`normal`，`700`等价于`bold`。每个数字值对应的字体加粗必须至少与下一个最小数字一样细，而且至少与下一个最大数字一样粗。

值： `normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | inherit`

初始值： `normal`

应用于： 所有元素

继承性： 有

计算值： 某个数字值（100等），或者某个数字值再加上一个相对值（bolder或lighter）

---

height

这个属性定义元素内容区的高度，在内容区外面可以增加内边距、边框和外边距。行内非替换元素会忽略这个属性。不允许指定负长度和百分数值。

值： <length> | <percentage> | auto | inherit

初始值： auto

应用于： 块级和替换元素

继承性： 无

百分数： 相对于包含块的高度计算

计算值： 对于auto和百分数值，根据指定确定；否则，是一个绝对长度，除非该属性不适用于当前元素（此时为auto）

---

left

这个属性定义一个定位元素的左外边距边界与其包含块左边界之间的偏移。

值： <length> | <percentage> | auto | inherit

初始值： auto

应用于： 定位元素（即position值不为static的元素）

继承性： 无

百分数： 相对于包含块的宽度

计算值： 对于相对定位元素，见说明；对于static元素，为auto；对于长度值，为相应的绝对长度；对于百分数值，为指定值；否则，为auto

说明： 对于相对定位元素，left的计算值始终等于right

---

letter-spacing

这个属性定义了在文本字符框之间插入多少空间。由于字符字形通常比其字符框要窄，指定长度值时，会调整字母之间通常的间隔。因此，normal就相当于值为0。允许指定负长度值，这会让字母之间挤得更紧。

值： <length> | normal | inherit

初始值： normal

应用于： 所有元素

继承性： 有

计算值： 对于长度值，为绝对长度；否则，是normal

---

line-height

这个属性会影响行框的布局。在应用到一个块级元素时，它定义了该元素中基线之间的最小距离而不是最大距离。line-height和font-size的计算值之差（在CSS中称为“行间距”）分为两半，分别加到一个文本行内容的顶部和底部。可以包含所有这些内容的最小框就是行框。原始数字值指定了一个缩放因子，后代元素会继承这个缩放因子而不是计算值。不允许指定负值。

值： <length> | <percentage> | <number> | normal | inherit

初始值： normal

应用于： 所有元素（不过参见有关替换和块级元素的正文）

继承性： 有

百分数： 相对于元素的字体大小

计算值： 对于长度和百分数值，为绝对值；否则，根据指定确定

---

list-style

这是一个简写属性，涵盖了所有其他列表样式属性。由于它应用到所有display为list-item的元素，所以在普通的HTML和XHTML中只能用于li元素，不过实际上它可以应用到任何元素，并由list-item元素继承。

值： [<list-style-type> || <list-style-image> || <list-style-position>] | inherit

初始值： 见单个属性

应用于： display值为list-item的元素

继承性： 有

计算值： 见单个属性

---

list-style-image

这个属性指定作为一个有序或无序列表项标志的图像。图像相对于列表项内容的放置位置通常使用list-style-position属性控制。

值： <uri> | none | inherit

初始值： none

应用于： display值为list-item的元素

继承性： 有

计算值： 对于<uri>值，为绝对URI；否则，为none

---

list-style-position

这个属性用于声明列表标志相对于列表项内容的位置。外部（outside）标志会放在离列表项边框边界一定距离处，不过这个距离在CSS中未定义。内部（inside）标志处理为好像它们是插入在列表项内容最前面的行内元素一样。

值： inside | outside | inherit

初始值： outside

应用于： display值为list-item的元素

继承性： 有

计算值： 根据指定确定

---

list-style-type

这个属性用于声明表示列表时所用的标志系统的类型。

CSS2.1值： disc | circle | square | decimal | decimal-leading-zero | lower-roman | upper-roman | lower-greek | lower-latin | upper-latin | armenian | georgian | none | inherit

CSS2值： disc | circle | square | decimal | decimal-leading-zero | upper-alpha | lower-alpha | upper-roman | lower-roman | lower-greek | hebrew | armenian | georgian | cjk-ideographic | hiragana | katakana | hiragana-iroha | none | inherit

初始值： disc

应用于： display值为list-item的元素

继承性： 有

计算值： 根据指定确定

margin

这个简写属性设置一个元素所有外边距的宽度，或者设置各边上外边距的宽度。块级元素的垂直相邻外边距会合并，而行内元素实际上不占上下外边距。行内元素的左右外边距不会合并，同样地，浮动元素的外边距也不合并。允许指定负的外边距值，不过使用时要小心。

值： [<length> | <percentage> | auto]{1,4} | inherit

初始值： 未定义

应用于： 所有元素

继承性： 无

百分数： 相对于包含块的宽度

计算值： 见单个属性

margin-bottom

这个属性设置一个元素下外边距的宽度。允许指定负值，不过使用时要小心。

值： <length> | <percentage> | auto | inherit

初始值： 0

应用于： 所有元素

继承性： 无

百分数： 相对于包含块的宽度

计算值： 对于百分数，根据指定确定；对于长度值，则为绝对长度

margin-left

这个属性设置元素左外边距的宽度。允许指定负值，不过使用时要小心。

值： <length> | <percentage> | auto | inherit

初始值： 0

应用于： 所有元素

继承性： 无

百分数： 相对于包含块的宽度

计算值： 对于百分数，根据指定确定；对于长度值，则为绝对长度

margin-right

这个属性设置元素右外边距的宽度。允许指定负值，不过使用时要小心。

值： <length> | <percentage> | auto | inherit

初始值： 0

应用于： 所有元素

继承性： 无

百分数： 相对于包含块的宽度

计算值： 对于百分数，根据指定确定；对于长度值，则为绝对长度

margin-top

这个属性设置元素上外边距的宽度。允许指定负值，不过使用时要小心。

值： <length> | <percentage> | auto | inherit

初始值： 0

应用于： 所有元素

继承性： 无

百分数： 相对于包含块的宽度

计算值： 对于百分数，根据指定确定；对于长度值，则为绝对长度

max-height

这个属性值会对元素的高度设置一个最高限制。因此，元素可以比指定值矮，但不能比其高。不允许指定负值。

值： <length> | <percentage> | none | inherit

初始值： none

应用于： 除了行内非替换元素和表元素之外的所有其他元素

继承性： 无

百分数： 相对于包含块的高度

计算值： 对于百分数，根据指定确定；对于长度值，为绝对长度；否则，为none

max-width

这个属性值对元素的宽度设置一个最大限制。因此，元素可以比指定值窄，但不能比其宽。不允许指定负值。

值： <length> | <percentage> | none | inherit

初始值： none

应用于： 除了行内非替换元素和表元素之外的所有其他元素

继承性： 无

百分数： 相对于包含块的宽度

计算值： 对于百分数，根据指定确定；对于长度值，为绝对长度；否则，为none

min-height

这个属性值对元素的高度设置一个最小限制。因此，元素可以比指定值高，但不能比其矮。不允许指定负值。

值： <length> | <percentage> | inherit

初始值： 0

应用于： 除了行内非替换元素和表元素之外的所有其他元素

继承性： 无

百分数： 相对于包含块的高度

计算值： 对于百分数，根据指定确定；对于长度值，为绝对长度

min-width

这个属性值对元素的宽度设置一个最小限制。所以，元素可以比指定值宽，但不能比其窄。不允许指定负值。

值： <length> | <percentage> | inherit

初始值： 0

应用于： 除了行内非替换元素和表元素的所有其他元素

继承性： 无

百分数： 相对于包含块的宽度

计算值： 对于百分数，根据指定确定；对于长度值，为绝对长度；否则，为none

outline

这个简写属性用于设置一个元素的整个轮廓。轮廓可以是不规则形状，不会改变，也不会影响元素的放置。

值： [<outline-color> || <outline-style> || <outline-width>] | inherit

初始值： 对简写属性未定义

应用于： 所有元素

继承性： 无

计算值： 见单个属性（outline-color等）

outline-color

这个属性设置一个元素整个轮廓中可见部分的颜色。要记住，轮廓的样式不能是none，否则轮廓不会出现。

值： <color> | invert | inherit

初始值： invert（或用户代理的特定值；见正文的介绍）

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

outline-style

这个属性用于设置一个元素整个轮廓的样式。样式不能是none，否则轮廓不会出现。

值： none | dotted | dashed | solid | double | groove | ridge | inset | outset | inherit

初始值： none

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

outline-width

这个属性设置元素整个轮廓的宽度。只有当轮廓样式不是none时，这个宽度才会起作用。如果样式为none，宽度实际上会重置为0。不允许指定负长度值。

值： thin | medium | thick | <length> | inherit

初始值： medium

应用于： 所有元素

继承性： 无

计算值： 绝对长度；如果边框样式是 none 或 hidden，则为 0

---

overflow

这个属性定义溢出元素内容区的内容会如何处理。如果值为 scroll，不论是否需要，用户代理都会提供一种滚动机制。因此，有可能即使元素框中可以放下所有内容也会出现滚动条。

值： visible | hidden | scroll | auto | inherit

初始值： visible

应用于： 块级和替换元素

继承性： 无

计算值： 根据指定确定

---

padding

这个简写属性设置元素所有内边距的宽度，或者设置各边上内边距的宽度。行内非替换元素上设置的内边距不会影响行高计算；因此，如果一个元素既有内边距又有背景，从视觉上看可能会延伸到其他行，有可能还会与其他内容重叠。元素的背景会延伸穿过内边距。不允许指定负内边距值。

值： [<length> | <percentage>]{1,4} | inherit

初始值： 对简写元素未定义

应用于： 所有元素

继承性： 无

百分数： 相对于包含块的宽度

计算值： 见单个属性（padding-top 等）

说明： 内边距绝不能为负

padding-bottom

这个属性设置元素下内边距的宽度。行内非替换元素上设置的下内边距不会影响行高计算，因此，如果一个元素既有内边距又有背景，从视觉上看可能会延伸到其他行，有可能还会与其他内容重叠。不允许指定负内边距值。

值： <length> | <percentage> | inherit

初始值： 0

应用于： 所有元素

继承性： 无

百分数： 相对于包含块的宽度

计算值： 对于百分数值，根据指定确定；对于长度值，则为绝对长度

说明： 内边距不能为负

padding-left

这个属性设置元素左内边距的宽度。行内非替换元素上设置的左内边距仅在元素所生成的第一个行内框的左边出现。不允许指定负内边距值。

值： <length> | <percentage> | inherit

初始值： 0

应用于： 所有元素

继承性： 无

百分数： 相对于包含块的宽度

计算值： 对于百分数值，根据指定确定；对于长度值，则为绝对长度

说明： 内边距不能为负

padding-right

这个属性设置元素右内边距的宽度。行内非替换元素上设置的右内边距仅在元素所生成的最后一个行内框的右边出现。不允许指定负内边距值。

值： <length> | <percentage> | inherit

初始值： 0

应用于： 所有元素

继承性： 无

百分数： 相对于包含块的宽度

计算值： 对于百分数值，根据指定确定；对于长度值，则为绝对长度

说明： 内边距不能为负

---

padding-top

这个属性设置元素上内边距的宽度。行内非替换元素上设置的上内边距不会影响行高计算；因此，如果一个元素既有内边距又有背景，从视觉上看可能会延伸到其他行，有可能还会与其他内容重叠。不允许指定负内边距值。

值： <length> | <percentage> | inherit

初始值： 0

应用于： 所有元素

继承性： 无

百分数： 相对于包含块的宽度

计算值： 对于百分数值，根据指定确定；对于长度值，则为绝对长度

说明： 内边距不能为负

---

position

这个属性定义建立元素布局所用的定位机制。任何元素都可以定位，不过绝对或固定定位元素会生成一个块级框，而不论该元素本身是什么类型。相对定位元素会相对于它在正常流中的默认位置偏移。

值： static | relative | absolute | fixed | inherit

初始值： static

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

---

quotes

这个属性用于确定引号和嵌套引号中使用的引用模式。具体的引号通过属性content插入。

值： [<string> <string>]+ | none | inherit

初始值： 取决于具体的用户代理

应用于： 所有元素

继承性： 有

计算值： 根据指定确定

---

right

这个属性定义定位元素右外边距边界与其包含块右边界之间的偏移。

值： <length> | <percentage> | auto | inherit

初始值： auto

应用于： 定位元素（即position值不为static的元素）

继承性： 无

百分数： 相对于包含块的宽度

计算值： 对于相对定位元素，见说明；对于静态元素，为auto；对于长度值，为相应的绝对长度；对于百分数值，为指定值；否则，为auto

说明： 对于相对定位元素，其left计算值始终等于right

---

text-align

这个属性通过指定行框与哪个点对齐，从而设置块级元素内文本的水平对齐方式。通过允许用户代理调整行内容中字母和字之间的间隔，可以支持值justify；不同用户代理可能会得到不同的结果。

CSS2.1值： left | center | right | justify | inherit

CSS2值： left | center | right | justify | <string> |inherit

初始值： 特定于用户代理，也可能取决于书写方向

应用于： 块级元素

继承性： 有

计算值： 根据指定确定

说明： CSS2引入了一个<string>值，不过由于缺乏广泛支持，已经从CSS2.1中去除

text-decoration

这个属性允许对文本设置某种效果，如加下划线。如果后代元素没有自己的装饰，祖先元素上设置的装饰会“延伸”到后代元素中。不要求用户代理支持blink。

值： none | [underline || overline || line-through || blink] | inherit

初始值： none

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

text-indent

用于定义块级元素中第一个内容行的缩进。这最常用于建立一个“标签页”效果。允许指定负值，这会产生一种“悬挂缩进”的效果。

值： <length> | <percentage> | inherit

初始值： 0

应用于： 块级元素

继承性： 有

百分数： 相对于包含块的宽度

计算值： 对于百分数，根据指定确定；对于长度值，则为绝对长度

text-transform

这个属性会改变元素中的字母大小写，而不论源文档中文本的大小写。如果值为capitalize，则要对某些字母大写，但是并没有明确地定义如何确定哪些字母要大写，这取决于用户代理如何识别出各个“词”。

值： uppercase | lowercase | capitalize | none | inherit

初始值： none

应用于： 所有元素

继承性： 有

计算值： 根据指定确定

top

这个属性定义了一个定位元素的上外边距边界与其包含块上边界之间的偏移。

值： <length> | <percentage> | auto | inherit

初始值： auto

应用于： 定位元素（即position值不为static的元素）

继承性： 无

百分数： 相对于包含块的高度

计算值： 对于相对定位元素，见说明；对于静态元素，为auto；对于长度值，为相应的绝对长度；对于百分数值，为指定值；否则，为auto

说明： 对于相对定位元素，如果top和bottom都是auto，其计算值都为0；如果其中一个为auto，则变成另一个值的相反数；如果二者都不是auto，bottom则取top值的相反数

unicode-bidi

这个属性允许创作人员在Unicode嵌套算法中生成多层嵌套。不支持双向文本的用户代理可以忽略这个属性。

值： normal | embed | bidi-override | inherit

初始值： normal

应用于： 所有元素

继承性： 无

计算值： 根据指定确定

vertical-align

这个属性定义行内元素的基线相对于该元素所在行的基线的垂直对齐。允许指定负长度值和百分数值，这会使元素降低而不是升高。在表单元格中，这个属性会设置单元格框中单元格内容的对齐方式。

值： baseline | sub | super | top | text-top | middle | bottom | text-bottom | <percentage> | <length> | inherit

初始值： baseline

应用于： 行内元素和表单元格

继承性： 无

百分数： 相对于元素的line-height的值

计算值： 对于百分数和长度值，为绝对长度；否则，根据指定确定

说明： 应用到表单元格时，只能识别baseline、top、middle和bottom等值

---

visibility

这个属性指定是否显示一个元素生成的元素框。这意味着元素仍占据其本来的空间，不过可以完全不可见。值collapse在表中用于从表布局中删除列或行。

值： visible | hidden | collapse | inherit

初始值： visible

应用于： 所有元素

继承性： 有

计算值： 根据指定确定

---

white-space

这个属性声明建立布局过程中如何处理元素中的空白符。值pre-wrap和pre-line是CSS2.1中新增的。

值： normal | nowrap | pre | pre-wrap | pre-line | inherit

初始值： normal

应用于： 所有元素（CSS2.1）；块级元素（CSS1和CSS2）

继承性： 无

计算值： 根据指定确定

---

width

这个属性定义了元素内容区的宽度，内容区外面可以增加内边距、边框和外边距。行内非替换元素会忽略这个属性。不允许指定负长度值和百分数值。

值： <length> | <percentage> | auto | inherit

初始值： auto

应用于： 块级和替换元素

继承性： 无

百分数： 相对于包含块的宽度

计算值： 对于auto和百分数值，根据指定确定；否则是绝对长度，除非当前元素无法应用此属性（此时为auto）

word-spacing

这个属性定义元素中字之间插入多少空白符。针对这个属性，“字”定义为由空白符包围的一个字符串。如果指定为长度值，会调整字之间的通常间隔；所以，normal就等同于设置为0。允许指定负长度值，这会让字之间挤得更紧。

值： <length> | normal | inherit

初始值： normal

应用于： 所有元素

继承性： 有

计算值： 对于normal，为绝对长度0；否则，是绝对长度

z-index

这个属性设置一个定位元素沿z轴的位置，z轴定义为垂直延伸到显示区的轴。如果为正数，则离用户更近，为负数则表示离用户更远。

值： <integer> | auto | inherit

初始值： auto

应用于： 定位元素

继承性： 无

计算值： 根据指定确定

表

border-collapse

这个属性定义了表中建立边框布局时使用的布局模型，即单元格、行等等的边框。尽管这个属性只应用于表，不过它可以由表中的所有元素继承。

值： collapse | separate | inherit

初始值： separate

应用于： display 值为 table 或 inline-table 的元素

继承性： 有

计算值： 根据指定确定

说明： 在 CSS2 中，默认值是 collapse

---

border-spacing

这个属性指定分隔边框模型中单元格边界之间的距离。在指定的两个长度值中，第一个是水平间隔，第二个是垂直间隔。除非 border-collapse 被设置为 separate，否则将忽略这个属性。尽管这个属性只应用于表，不过它可以由表中的所有元素继承。

值： <length> <length>? | inherit

初始值： 0

应用于： display 值为 table 或 inline-table 的元素

继承性： 有

计算值： 两个绝对长度

说明： 除非 border-collapse 被设置为 separate，否则将忽略这个属性

---

caption-side

这个属性指定了表标题相对于表框的放置位置。表标题显示为好像它是表之前(或之后)的一个块级元素。

值： top | bottom

初始值： top

应用于： display 值为 table-caption 的元素

继承性： 有

计算值： 根据指定确定

说明： CSS2 中还有值 left 和 right，不过由于缺乏广泛支持，所以已经从 CSS2.1 去除

---

empty-cells

这个属性定义了不包含任何内容的表单元格如何表示。如果显示，就会绘制单元格的边框和背景。除非 border-collapse 被设置为 separate，否则将忽略这个属性。

值： show | hide | inherit

初始值： show

应用于： display 值为 table-cell 的元素

继承性： 有

计算值： 根据指定确定

说明： 除非 border-collapse 被设置为 separate，否则将忽略这个属性

table-layout

这个属性指定了完成表布局时所用的布局算法。固定布局算法比较快，但是不太灵活，而自动算法比较慢，不过更能反映传统的 HTML 表。

值： auto | fixed | inherit

初始值： auto

应用于： display 值为 table 或 inline-table 的元素

继承性： 有

计算值： 根据指定确定

分页媒体

orphans

这个属性指定了元素放在页面底部时所允许的最少文本行数。这可能会影响元素中分页符的放置。

值： <integer> | inherit

初始值： 2

应用于： 块级元素

继承性： 有

计算值： 根据指定确定

page-break-after

这个属性声明一个元素后是否应当放置分页符。尽管可以用 always 强制放上分页符，

但是无法保证避免分页符的插入；创作人员最多只能请求用户代理尽可能避免插入分页符。

值： auto | always | avoid | left | right | inherit

初始值： auto

应用于： position 值为 relative 或 static 的非浮动块级元素

继承性： 无

计算值： 根据指定确定

---

page-break-before

这个属性声明一个元素前是否应当放置分页符。尽管可以用 always 强制放上分页符，但是无法保证避免分页符的插入；创作人员最多只能请求用户代理尽可能避免插入分页符。

值： auto | always | avoid | left | right | inherit

初始值： auto

应用于： position 值为 relative 或 static 的非浮动块级元素

继承性： 无

计算值： 根据指定确定

---

page-break-inside

这个属性声明一个元素内部是否应当放置分页符。由于元素可能比页框还高，所以无法保证避免分页符的插入；创作人员最多只能请求用户代理尽可能避免插入分页符。

值： auto | avoid | inherit

初始值： auto

应用于： position 值为 relative 或 static 的非浮动块级元素

继承性： 有

计算值： 根据指定确定

---

widows

这个属性指定元素放在页面顶部时所允许的最少文本行数。这可能会影响元素中分页符的放置。

值： <integer> | inherit

初始值： 2

应用于： 块级元素

继承性： 有

计算值： 根据指定确定

从 CSS2.1 去除的属性

以下属性出现在CSS2中，但是由于缺乏广泛支持，已经从CSS2.1去除。它们没有计算值信息，因为计算值在CSS2.1中才首次明确定义。

视觉样式

font-size-adjust

这个属性的目的是让创作人员能够完成字体缩放，使得替代字体与创作人员所要的字体不至于差异太大，尽管替代字体可能有完全不同的x-height。注意，这个属性在CSS2.1中未出现。

值： <number> | none | inherit

初始值： none

应用于： 所有元素

继承性： 有

font-stretch

利用这个属性，可以将给定字体的字符字形拉伸得更宽或更窄，通常会选择字体系列中的压缩或拉伸字体。注意，这个属性在CSS2.1中未出现。

值： normal | wider | narrower | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded | inherit

初始值： normal

应用于： 所有元素

继承性： 有

marker-offset

这个属性指定了标志框的最近边框边界与其相应元素框之间的距离。

值： <length> | auto | inherit

初始值： auto

应用于： display 值为 marker 的元素

继承性： 无

说明： 这个属性在 CSS2.1 中已经废弃，而且很可能不会在 CSS3 中再次出现，display 值 marker 也不会再出现；写作本书时，据称可能会用其他机制来达到这些效果

text-shadow

这个属性允许为元素中的文本指定一个或多个“阴影”。阴影定义中的前两个长度值分别设置与元素文本的水平和垂直偏移。第三个长度值定义了模糊半径。注意，这个属性在 CSS2.1 中未出现。

值： none | [<color> || <length> <length> <length>? ,]* [<color> || <length> <length> <length>?] | inherit

初始值： none

应用于： 所有元素

继承性： 无

分页媒体

marks

这个属性定义是否在内容区之外但是在画布的可打印区域内放“十字标志”（也称为登记标志或定位标志）。这种标志具体的放置和显示没有相关定义。注意，这个值在CSS2.1中未出现。

值： [crop || cross] | none | inherit

初始值： none

应用于： 页面上下文

继承性： N/A

page

这个属性与size结合可以指定打印一个元素时所用的特定页面类型。注意，这个属性在CSS2.1中未出现。

值： <identifier> | inherit

初始值： auto

应用于： 块级元素

继承性： 有

size

利用这个属性，创作人员可以声明打印一个元素时所用页框的大小和方向。它可以与page结合使用，不过并不要求一定如此。注意，这个属性在CSS2.1中未出现。

值： <length>{1,2} | auto | portrait | landscape | inherit

初始值： auto

应用于： 页面区

继承性： 无

声音样式

azimuth

这个属性设置了发声源在水平面（也称为地平线）的角度。这个属性与elevation结合使用，将声音放在以用户为中心的一个假想球体中的一点上。

值： <angle> | [[left-side | far-left | left | center-left | center | center-right | right | far-right | right-side] || behind] | leftwards | rightwards | inherit

初始值： center

应用于： 所有元素

继承性： 有

计算值： 标准化角度

cue

这是一个简写属性，允许创作人员定义在元素内容的声音表现之前和之后播放的提示。“提示”相当于听觉领域的图标。

值： [<cue-before> || <cue-after>] | inherit

初始值： none

应用于： 所有元素

继承性： 无

计算值： 见单个属性（cue-before等）

cue-after

这个属性允许创作人员定义在元素内容的声音表现之后播放的提示。

值： <uri> | none | inherit

初始值： none

应用于： 所有元素

继承性： 无

计算值： 对于<uri>值，为绝对URI；否则，为none

cue-before

这个属性允许创作人员定义在元素内容的声音表现之前播放的提示。

值： <uri> | none | inherit

初始值： none

应用于： 所有元素

继承性： 无

计算值： 对于<uri>值，为绝对URI；否则，为none

elevation

这个属性设置发声源在水平面（也称为地平线）之上或之下的垂直角度（俯仰角）。这个属性与azimuth结合使用，将声音放在以用户为中心的一个假想球体中的一点上。

值： <angle> | below | level | above | higher | lower | inherit

初始值： level

应用于： 所有元素

继承性： 有

计算值： 标准化角度

---

pause

这是一个简写属性，允许创作人员定义在元素内容的声音表现之前或之后的停顿。“停顿”是一个时间间隔，在此期间不播放任何内容，不过可能还能听到背景声音。

值： [[<time> | <percentage>]{1,2}] | inherit

初始值： 0

应用于： 所有元素

继承性： 无

计算值： 见单个属性（pause-before等）

---

pause-after

这个属性允许创作人员定义在元素内容的声音表现之后的停顿长度。

值： <time> | <percentage> | inherit

初始值： 0

应用于： 所有元素

继承性： 无

计算值： 绝对时间值

---

pause-before

这个属性允许创作人员定义在元素内容的声音表现之前的停顿长度。

值： <time> | <percentage> | inherit

初始值： 0

应用于： 所有元素

继承性： 无

计算值： 绝对时间值

pitch

指定用声音表现元素的内容时所用语音声音的平均音高（频率）。声音的平均音高很大程度上取决于声音系列。

值： <frequency> | x-low | low | medium | high | x-high | inherit

初始值： medium

应用于： 所有元素

继承性： 有

计算值： 绝对频率值

pitch-range

这个属性指定用声音表现元素的内容时语音所用平均音高的变化。变化幅度越大，声音听上去就会越“颤抖”。

值： <number> | inherit

初始值： 50

应用于： 所有元素

继承性： 有

计算值： 根据指定确定

play-during

这个属性提供的声音将在用声音表现元素的内容时作为背景播放。这个声音可以与其他背景声音混合（使用祖先元素的play-during设置），或者可以在元素的声音表现期间替换其他声音。

值： <uri> | [mix || repeat]? | auto | none | inherit

初始值： auto

应用于： 所有元素

继承性： 无

计算值： 对于 <uri> 值，为绝对 URI；否则，根据指定确定

richness

这个属性设置了用声音表现元素的内容时所用语音的“亮度”(音色)。声音越明亮,“含义”就越丰富。

值: <number> | inherit

初始值: 50

应用于: 所有元素

继承性: 有

计算值: 根据指定确定

speak

这个属性确定元素内容如何用声音表现,或者是否用声音表现。值spell-out通常用于缩写词和简写,如W3C或CSS。如果值为none,就会跳过该元素(不会用声音表现)。

值: normal | none | spell-out | inherit

初始值: normal

应用于: 所有元素

继承性: 有

计算值: 根据指定确定

speak-header

这个属性指定是否在念各个单元格之前读出与这些单元格关联的表标题的内容,或者只是当与一个单元格关联的标题不同于前一个单元格相关标题时才读出这些标题内容。

值: once | always | inherit

初始值: once

应用于: 包含表标题信息的元素

继承性: 有

计算值: 根据指定确定

speak-numeral

这个属性确定声音表现期间如何读出数字。

值： digits | continuous | inherit

初始值： continuous

应用于： 所有元素

继承性： 有

计算值： 根据指定确定

---

speak-punctuation

这个属性确定声音表现期间如何读出标点符号。如果值为code，会原样读出标点符号。

值： code | none | inherit

初始值： none

应用于： 所有元素

继承性： 有

计算值： 根据指定确定

---

speech-rate

这个属性设置用声音表现元素的内容时的语率，即读出单词的平均速率。

值： <number> | x-slow | slow | medium | fast | x-fast | faster | slower | inherit

初始值： medium

应用于： 所有元素

继承性： 有

计算值： 绝对数值

---

stress

这个属性影响语音语调的峰值高度。最高峰值由语言中的重音记号生成。

值： <number> | inherit

初始值： 50

应用于： 所有元素

继承性： 有

计算值： 根据指定确定

voice-family

这个属性用于定义用元素内容的声音表现中可以使用的一组声音系列，对应于font-family。可取的通用声音包括male、female和child。

值： [[<specific-voice> | <generic-voice>],]* [<specific-voice> | <generic-voice>] | inherit

初始值： 取决于具体的用户代理

应用于： 所有元素

继承性： 有

计算值： 根据指定确定

volume

这个属性设置用声音表现内容时声音波形的平均音量。因此，有很大波峰和波谷的波形可能会超过或低于这个属性设置的音量。注意，0不同于silent。

值： <number> | <percentage> | silent | x-soft | soft | medium | loud | x-loud | inherit

初始值： medium

应用于： 所有元素

继承性： 有

计算值： 绝对数

附录 B

选择器、伪类和伪元素参考

选择器

通配选择器

这个选择器与文档语言中的所有元素名匹配。如果规则没有显式指定选择器，则可以认为是通配选择器。

模式：*

示例：

```
* {color: red;}
div * p {color: blue;}
```

类型选择器

这个选择器与文档语言中的一个元素名匹配。该元素名的每一个实例都与之匹配(CSS1称之为元素选择器)。

模式：element1

示例：

```
body {background: #FFF;}
p {font-size: 1em;}
```

后代选择器

后代选择器允许创作人员根据元素的状态（即作为另一个元素的后代）来选择该元素。与之匹配的元素可以是祖先元素的子元素、孙元素、曾孙元素等等（CSS1 称之为上下文选择器）。

模式：element1 element2

示例：

```
body h1 {font-size: 200%;}
table tr td div ul li {color: purple;}
```

子选择器

这种类型的选择器用于根据元素的状态（即作为另一个元素的子元素）来选择该元素。这比后代元素的限制更严格，因为只能匹配子元素。

模式：element1 > element2

示例：

```
div > p {color: cyan;}
ul > li {font-weight: bold;}
```

相邻兄弟选择器

该选择器允许创作人员选择元素的下一个相邻兄弟。两个元素之间的文本会被忽略；只考虑文档树中的元素及其位置。

模式：element1 + element2

示例：

```
table + p {margin-top: 2.5em;}
h1 + * {margin-top: 0;}
```

类选择器

在支持类选择器的语言中（如 HTML、SVG 和 MathML），使用“点记法”的类选择器可以用于选择有某个或某些特定类值的元素。类值名必须跟在点号后面。多个类值可以“串”在一起。如果点号前面没有元素名，这个选择器将与所有包含该类值的元素匹配。

模式：element1.classname element1.classname1.classname2

示例：

```
p.urgent {color: red;}
a.external {font-style: italic;}
.example {background: olive;}
```

ID 选择器

在支持类选择器的语言中（如HTML），使用“# 号记法”的ID选择器可以用于选择有某个或某些特定ID值的元素。ID值名必须紧跟在一个 # 号后面。如果 # 号前面没有元素名，这个选择器将与所有包含该ID值的元素匹配。

模式：element1#idname

示例：

```
h1#page-title {font-size: 250%;}
body#home {background: silver;}
#example {background: lime;}
```

简单属性选择器

这种选择器允许创作人员根据属性的存在来选择元素，而不论该属性的值是什么。

模式：element1[attr]

示例：

```
a[rel] {border-bottom: 3px double gray;}
p[class] {border: 1px dotted silver;}
```

具体属性值选择器

这种选择器允许创作人员根据属性的具体值选择元素。

模式：element1[attr="value"]

示例：

```
a[rel="Home"] {font-weight: bold;}
p[class="urgent"] {color: red;;}
```

部分属性值选择器

这种选择器允许创作人员根据属性值（各值之间用空格分隔）中的一部分选择元素。注意，[class~="value"] 等价于 .value（见以上说明）。

模式：element1[attr~="value"]

示例：

```
a[rel~="friend"] {text-transform: uppercase;}
p[class~="warning"] {background: yellow;}
```

开始子串属性值选择器

这种选择器允许创作人员根据属性值最开始的子串选择元素。

模式：element1[attr^="substring"]

示例：

```
a[href^="/blog"] {text-transform: uppercase;}
p[class^="test-"] {background: yellow;}
```

结束子串属性值选择器

这种选择器允许创作人员根据属性值最后面的子串选择元素。

模式：element1[attr$="substring"]

示例：

```
a[href$=".pdf"] {font-style: italic;}
```

任意子串属性值选择器

这种选择器允许创作人员根据属性值中任意位置上的子串选择元素。

模式：element1[attr*="substring"]

示例：

```
a[href*="oreilly.com"] {font-weight: bold;}
div [class*="port"] {border: 1px solid red;}
```

语言属性选择器

这种选择器允许创作人员选择有一个 lang 属性的元素，lang 属性值是一个用连字号分隔的值列表，而且最前面是该选择器中提供的值。

模式：element1[lang|="lc"]

示例：

```
html[lang|="en"] {color: gray;}
```

伪类和伪元素

:active

这个伪类应用于处于激活状态的元素。最常见的例子就是在HTML文档中点击一个超链接：在鼠标按钮按下期间，这个链接是激活的。还有其他一些方式来激活元素，另外从理论上讲其他元素也可以被激活，不过CSS对此没有定义。

类型：伪类

应用于：被激活的元素

示例：

```
a:active {color: red;}
*:active {background: blue;}
```

:after

这个伪元素允许创作人员在元素内容的最后插入生成内容。默认地，这个伪元素是行内元素，不过可以使用属性 display 改变这一点。

类型：伪元素

生成：一个伪元素，其中包含放在元素内容之后的生成内容

示例：

```
a.external:after {content: " " url(/icons/globe.gif);}
p:after {content: " | ";}
```

:before

这个伪元素允许创作人员在元素内容的最前面插入生成内容。默认地，这个伪元素是行内元素，不过可以使用属性 display 改变这一点。

类型：伪元素

生成：一个伪元素，其中包含放在元素内容前面的生成内容

示例：

```
a[href]:before {content: "[LINK] ";}
p:before {content: attr(class);}
```

:first-child

利用这个伪类，只有当元素是另一个元素的第一个子元素时才能匹配。例如，p:first-child会选择作为另外某个元素第一个子元素的所有 p 元素。一般可能认为这会选择作为段落第一个子元素的元素，但事实上并非如此；如果要选择段落的第一个子元素，应当写作 p > *:first-child。

类型：伪类

应用于：作为另一个元素第一个子元素的所有元素

示例：

```
body *:first-child {font-weight: bold;}
p:first-child {font-size: 125%;}
```

:first-letter

这个伪元素用于指定一个元素第一个字母的样式。所有前导标点符号应当与第一个字母一同应用该样式。某些语言有一些要处理为单个字符的字母组合，如果是这样，用户代理可能会把首字母样式同时应用到这个字母组合。在 CSS2.1 之前，:first-letter 只能与块级元素关联。CSS2.1 则扩大了这个范围，可以与任何元素关联。可以应用到首字母的属性是有限的。

类型：伪元素

生成：包含元素首字母的一个伪元素

示例：

```
h1:first-letter {font-size: 166%;}
a:first-letter {text-decoration: underline;}
```

:first-line

这个伪元素用于设置元素中第一行文本的样式，而不论该行出现多少单词。:first-line只能与块级元素关联。可以应用到首行的属性是有限的。

类型：伪元素

生成：包含元素第一行格式化文本的伪元素

示例：

```
p.lead:first-line {font-weight: bold;}
```

:focus

这个伪类应用于有焦点的元素。例如HTML中的一个有文本输入焦点的输入框，其中出现了文本输入光标；也就是说，在用户开始键入时，文本会输入到这个输入框。其他元素（如超链接）也可以有焦点，不过CSS没有定义哪些元素可以有焦点。

类型：伪类

应用于：有焦点的元素

示例：

```
a:focus {outline: 1px dotted red;}
input:focus {background: yellow;}
```

:hover

这个伪类应用于处于“悬停状态”的元素。悬停定义为用户指示了一个元素但没有将其激活。对此最常见的例子是将鼠标指针移到HTML文档中一个超链接的边界范围内。理论上，其他元素也可以处于悬停状态，不过CSS没有定义究竟是哪些元素。

类型：伪类

应用于：处于悬停状态的元素

示例：

```
a[href]:hover {text-decoration: underline;}
p:hover {background: yellow;}
```

:lang

这个伪类根据元素的语言编码匹配元素。这种语言信息必须包含在文档中，或者与文档关联；不能从CSS指定。:lang的处理与|=属性选择器相同。

类型：伪类

应用于：有相关语言编码信息的元素

示例：

```
html:lang(en) {background: silver;}
*:lang(fr) {quotes: '?' ' ?;}
```

:link

这个伪类应用于URI尚未访问过的链接；也就是说，链接所指的URI尚未出现在用户代理的历史中。这种状态与:visited状态是互斥的。

类型：伪类

应用于：指向另外的尚未访问过的资源的链接

示例：

```
a:link {color: blue;}
*:link {text-decoration: underline;}
```

:visited

这个伪类应用于URI已访问的链接；也就是说，链接指向的URI已经出现在用户代理的历史中。这个状态与:link状态互斥。

类型：伪类

应用于：指向另外的已访问资源的链接

示例：

```
a:visited {color: purple;}
*:visited {color: gray;}
```

附录C

示例HTML 4样式表

以下样式表是从CSS2.1规范的附录D节选而来。需要指出重要的两点。首先，尽管CSS2.1称“建议开发人员使用（这个样式表）作为实现中的默认样式表”，但这不一定可行。例如，对于以下规则：

```
ol, ul, dir, menu, dd
    {margin-left: 40px;}
```

这个规则描述了传统列表缩进40像素的距离，它使用左外边距达到这个效果。不过，有些浏览器会使用40像素的左内边距而不是外边距，而且认为这种解决方案更好（有关详细内容见第12章）。因此，不能指望以此作为所有用户代理的默认样式表。提供这个样式表更多地是为了说明相关内容，这只是一个学习工具。

要注意的第二点是，并非所有HTML元素在这个样式表中都得到了充分描述，因为CSS还没有详细到足以完备而准确地描述所有元素。比较经典的例子是表单元素，如“提交”按钮，这些元素是替换元素，但是有其自己的特殊格式化需求。“提交”按钮是替换元素，所以其元素框的底边应当与基线对齐。不过，创作人员可能希望按钮中的文本与该行中其他文本的基线对齐。这种想法是有道理的，不过（写作本书时）CSS没有能力描述这种行为；因此，只能用以下规则描述这种元素：

```
button, textarea, input, object, select, img {
                    display:inline-block;}
```

这种元素的其他格式化部分则留给用户代理完成。

记住以上这两点，以下是CSS2规范中的样式表（格式上稍做调整）。除了格式调整外，如果还做了其他修改都会在注释中说明。

```
address, blockquote, body, dd, div, dl, dt, fieldset, form,
frame, frameset, h1, h2, h3, h4, h5, h6, noframes,
ol, p, ul, center, dir, hr, menu, pre {
                    display: block;}
li                  {display: list-item;}
head                {display: none;}
table               {display: table;}
tr                  {display: table-row;}
thead               {display: table-header-group;}
tbody               {display: table-row-group;}
tfoot               {display: table-footer-group;}
col                 {display: table-column;}
colgroup            {display: table-column-group;}
td, th              {display: table-cell;}
caption             {display: table-caption;}
th                  {font-weight: bolder; text-align: center;}
caption             {text-align: center;}
body                {padding: 8px; line-height: 1.12em;}
h1                  {font-size: 2em; margin: .67em 0;}
h2                  {font-size: 1.5em; margin: .75em 0;}
h3                  {font-size: 1.17em; margin: .83em 0;}
h4, p, blockquote, ul, fieldset, form, ol, dl, dir, menu {
                     margin: 1.12em 0;}
h5                  {font-size: .83em; margin: 1.5em 0;}
h6                  {font-size: .75em; margin: 1.67em 0;}
h1, h2, h3, h4, h5, h6, b, strong {
                    font-weight: bolder;}
blockquote          {margin-left: 40px; margin-right: 40px;}
i, cite, em, var, address  {
                    font-style: italic;}
pre, tt, code, kbd, samp {
                    font-family: monospace;}
pre                 {white-space: pre;}
button, textarea, input, object, select, img {
                    display:inline-block;}
big                 {font-size: 1.17em;}
small, sub, sup {font-size: .83em;}
sub                 {vertical-align: sub;}
sup                 {vertical-align: super;}
s, strike, del  {text-decoration: line-through;}
hr                  {border: 1px inset;}
ol, ul, dir, menu, dd  {
                    margin-left: 40px;}
ol                  {list-style-type: decimal;}
ol ul, ul ol, ul ul, ol ol {
                    margin-top: 0; margin-bottom: 0;}
u, ins              {text-decoration: underline;}
br:before           {content: "\A";}
center              {text-align: center;}
abbr, acronym   {font-variant: small-caps; letter-spacing: 0.1em;}
:link,:visited {text-decoration: underline;}
:focus              {outline: thin dotted invert;}
/* Begin bidirectionality settings (do not change) */
```

```
BDO[DIR="ltr"]  {direction: ltr; unicode-bidi: bidi-override;}
BDO[DIR="rtl"]  {direction: rtl; unicode-bidi: bidi-override;}
*[DIR="ltr"]    {direction: ltr; unicode-bidi: embed;}
*[DIR="rtl"]    {direction: rtl; unicode-bidi: embed;}
@media print {
  h1            {page-break-before: always;}
  h1, h2, h3, h4, h5, h6 {
                 page-break-after: avoid;}
  ul, ol, dl    {page-break-before: avoid;}
}
@media aural {  /* changed from 'speech' which was not defined in CSS2 */
  h1, h2, h3, h4, h5, h6 {
                voice-family: paul, male; stress: 20; richness: 90;}
  h1            {pitch: x-low; pitch-range: 90;}
  h2            {pitch: x-low; pitch-range: 80;}
  h3            {pitch: low; pitch-range: 70;}
  h4            {pitch: medium; pitch-range: 60;}
  h5            {pitch: medium; pitch-range: 50;}
  h6            {pitch: medium; pitch-range: 40;}
  li, dt, dd    {pitch: medium; richness: 60;}
  dt            {stress: 80;}
  pre, code, tt {pitch: medium; pitch-range: 0; stress: 0; richness: 80;}
  em            {pitch: medium; pitch-range: 60; stress: 60; richness: 50;}
  strong        {pitch: medium; pitch-range: 60; stress: 90; richness: 90;}
  dfn           {pitch: high; pitch-range: 60; stress: 60;}
  s, strike     {richness: 0;}
  i             {pitch: medium; pitch-range: 60; stress: 60; richness: 50;}
  b             {pitch: medium; pitch-range: 60; stress: 90; richness: 90;}
  u             {richness: 0;}
  a:link        {voice-family: harry, male;}
  a:visited     {voice-family: betty, female;}
  a:active      {voice-family: betty, female; pitch-range: 80; pitch: x-high;}
}
```