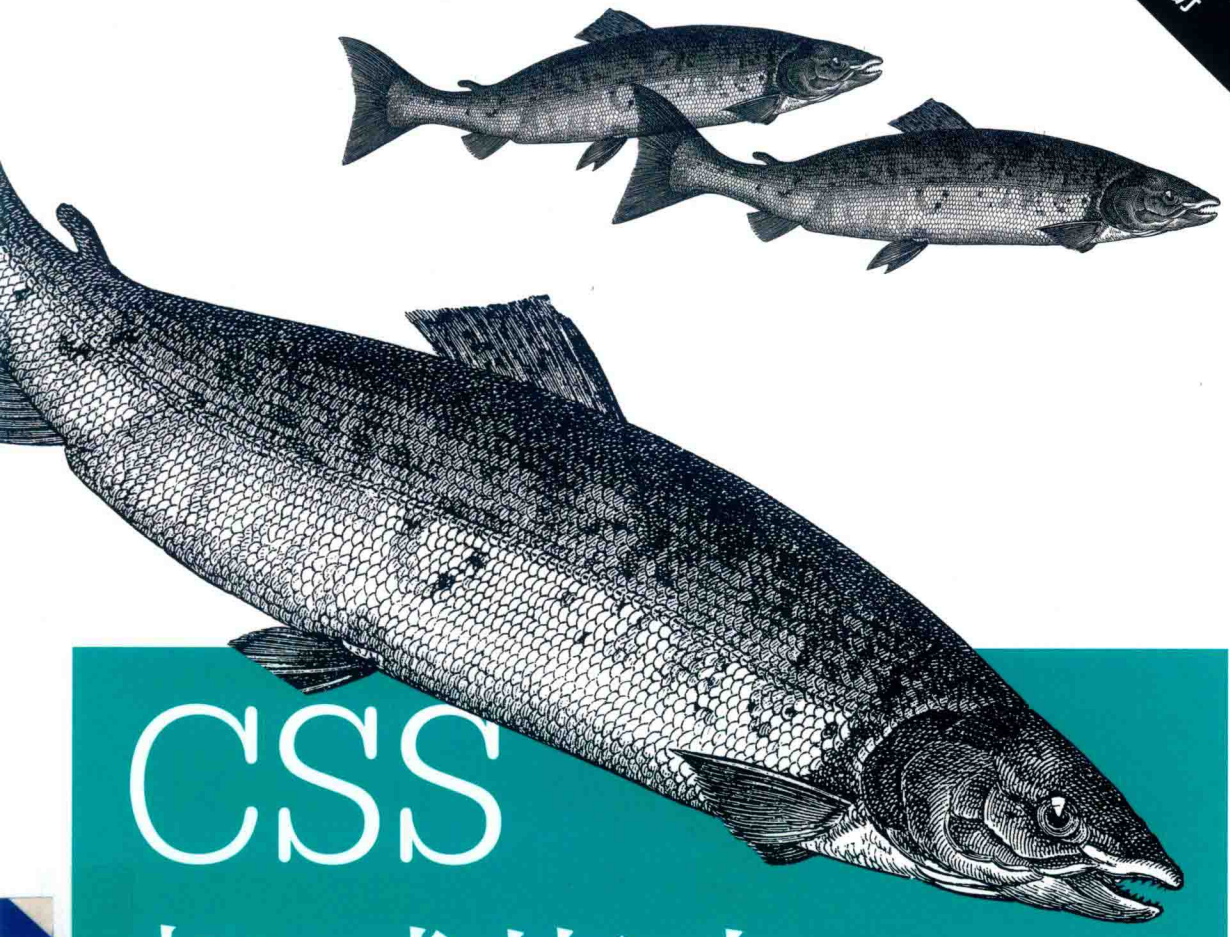
O'REILLY®

第四版
上册



CSS 权威指南

CSS: The Definitive Guide

Web视觉呈现技术

非外借

Eric A. Meyer Estelle Weyl 著
安道 译

中国电力出版社

第四版/上册

CSS权威指南

Eric A. Meyer & Estelle Weyl 著

安道 译



ol • Tokyo O'REILLY®

a, Inc. 授权中国电力出版社出版

中国电力出版社

Copyright © 2018 Eric Meyer, Estelle Weyl. All rights reserved.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2019. Authorized translation of the English edition, 2018 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2018。

简体中文版由中国电力出版社出版 2019。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

图书在版编目（CIP）数据

CSS权威指南：第四版 / (美) 迈耶（Eric A. Meyer），(美) 埃斯特尔·韦尔（Estelle Weyl）著；安道译. -- 北京：中国电力出版社，2019.4

书名原文：CSS: The Definitive Guide, 4th Edition

ISBN 978-7-5198-2659-8

I. ①C… II. ①迈… ②埃… ③安… III. ①网页制作工具－指南 IV. ①TP393.092.2-62

中国版本图书馆CIP数据核字(2018)第273268号

北京市版权局著作权合同登记 图字：01-2018-7457号

出版发行：中国电力出版社
地　　址：北京市东城区北京站西街19号（邮政编码100005）
网　　址：http://www.cepp.sgcc.com.cn
责任编辑：刘　炽（liuchi1030@163.com）
责任校对：王小鹏
装帧设计：Karen Montgomery，张　健
责任印制：杨晓东

印　刷：三河市航远印刷有限公司
版　次：2019年4月第一版
印　次：2019年4月北京第一次印刷
开　本：750毫米×980毫米　16开本
印　张：68.5
字　数：1290千字
印　数：0001—3000册
定　价：198.00元（上下册）

版 权 专 有　侵 权 必 究

本书如有印装质量问题，我社营销中心负责退换

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了《Make》杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

献给 Kat、Carolyn、Rebecca 和 Joshua。

—— E. M.

献给 Amie。

—— E. W.

目录

上册

前言 1

第 1 章 CSS和文档 9

1.1 Web样式简介 9

1.2 元素 10

1.2.1 置换元素和非置换元素 10

1.2.2 元素的显示方式 11

1.3 把 CSS 应用到 HTML 上 15

1.3.1 link 标签 16

1.3.2 style 元素 20

1.3.3 @import 指令 21

1.3.4 HTTP 链接 22

1.3.5 行内样式 23

1.4 样式表中的内容 24

1.4.1 标记 24

1.4.2 规则的结构 24

1.4.3 厂商前缀 25

1.4.4 处理空白 26

1.4.5 CSS 注释 27

1.5 媒体查询 28

1.5.1 用法 28

1.5.2 简单的媒体查询 28

1.5.3 媒体类型 29

1.5.4 媒体描述符 30
1.5.5 媒体特性描述符和值的类型 32
1.6 特性查询 32
1.7 小结 35
第2章 选择符 37
2.1 样式的基本规则 37
2.1.1 元素选择符 38
2.1.2 声明和关键字 39
2.2 群组 42
2.2.1 群组选择符 42
2.2.2 群组声明 44
2.2.3 二者结合 45
2.2.4 在旧浏览器中使用新元素 46
2.3 类选择符和 ID 选择符 47
2.3.1 类选择符 47
2.3.2 多个类 50
2.3.3 ID 选择符 51
2.3.4 在类选择符和 ID 选择符之间选择 52
2.4 属性选择符 53
2.4.1 简单属性选择符 53
2.4.2 根据精准的属性值选择 55
2.4.3 根据部分属性值选择 57
2.4.4 不区分大小写的标识符 62
2.5 根据文档结构选择 63
2.5.1 理解父子关系 63
2.5.2 后代选择符 65
2.5.3 选择子元素 68
2.5.4 选择紧邻同胞元素 69
2.5.5 选择后续同胞 71
2.6 伪类选择符 71
2.6.1 拼接伪类 72
2.6.2 结构伪类 73
2.6.3 动态伪类 86

2.6.4 UI 状态伪类 90
2.6.5 :target 伪类 96
2.6.6 :lang 伪类 97
2.6.7 否定伪类 98
2.7 伪元素选择符 100
2.7.1 装饰首字母 101
2.7.2 装饰首行 102
2.7.3 对 ::first-letter 和 ::first-line 的限制 103
2.7.4 装饰（或创建）前置和后置内容元素 103
2.8 小结 104

第 3 章 特指度和层叠 105

3.1 特指度 105
3.1.1 声明和特指度 107
3.1.2 通用选择符的特指度 109
3.1.3 ID 和属性选择符的特指度 109
3.1.4 行内样式的特指度 109
3.1.5 重要性 110
3.2 继承 111
3.3 层叠 114
3.3.1 按权重和来源排序 115
3.3.2 按特指度排序 117
3.3.3 按前后位置排序 117
3.3.4 CSS 之外的表现提示 119
3.4 小结 120

第 4 章 值和单位 121

4.1 关键字、字符串和其他文本值 121
4.1.1 关键字 121
4.1.2 字符串 124
4.1.3 URL 125
4.1.4 图像 126
4.1.5 标识符 127
4.2 数字和百分数 127
4.2.1 整数 127

4.2.2 数字 128
4.2.3 百分数 128
4.2.4 弹性值 128
4.3 距离 128
4.3.1 绝对长度单位 129
4.3.2 分辨率单位 131
4.3.3 相对长度单位 132
4.4 计算值 138
4.5 属性值 139
4.6 颜色 140
4.6.1 具名颜色 140
4.6.2 RGB 和 RGBa 颜色 141
4.6.3 HSL 和 HSLa 颜色 146
4.6.4 颜色关键字 149
4.7 角度 149
4.8 时间和频率 150
4.9 位置 151
4.10 自定义值 151
第 5 章 字体 155
5.1 字体族 155
5.1.1 使用通用字体族 156
5.1.2 指定字体族 157
5.2 使用 @font-face 160
5.2.1 必须的描述符 161
5.2.2 其他字体描述符 166
5.2.3 组合描述符 169
5.3 字重 172
5.3.1 字重的工作方式 173
5.3.2 增大字重 175
5.3.3 减小字重 178
5.3.4 font-weight 描述符 179
5.4 字号 180
5.4.1 绝对大小 181

5.4.2 相对大小 183
5.4.3 百分数和 em 184
5.4.4 字号的继承 185
5.4.5 使用长度单位 188
5.4.6 自动调整字号 189
5.5 字形 191
5.6 字体拉伸 195
5.7 字距调整 198
5.8 字体变形 198
5.8.1 Level 3 新增的值 200
5.8.2 font-variant 描述符 201
5.9 字体特性 202
5.10 字体合成 204
5.11 font 属性 206
5.11.1 加入行高 207
5.11.2 正确使用简写 208
5.11.3 使用系统字体 209
5.12 字体匹配机制 210
5.13 小结 212

第 6 章 文本属性 213

6.1 缩进和行内对齐 213
6.1.1 缩进文本 214
6.1.2 文本对齐 217
6.1.3 对齐最后一行 220
6.2 块级对齐 222
6.2.1 行的高度 222
6.2.2 纵向对齐文本 226
6.3 单词间距和字符间距 231
6.3.1 单词间距 231
6.3.2 字符间距 233
6.3.3 对齐方式对间距的影响 234
6.4 文本转换 235
6.5 文本装饰 237

6.6 文本渲染效果 241
6.7 文本阴影 242
6.8 处理空白 244
6.9 换行和断字 248
6.10 书写模式 254
6.10.1 设定书写模式 254
6.10.2 改变文本方向 258
6.10.3 声明方向 259
6.11 小结 261

第 7 章 视觉格式化基础 262

7.1 元素框基础 262
7.1.1 重要概念概览 263
7.1.2 容纳块 264
7.2 调整元素的显示方式 265
7.2.1 改变显示方式 266
7.2.2 块级框 268
7.2.3 横向格式化 270
7.2.4 横向格式化属性 271
7.2.5 使用 auto 272
7.2.6 多个 auto 273
7.2.7 负外边距 275
7.2.8 百分数 276
7.2.9 置换元素 277
7.2.10 纵向格式化 278
7.2.11 纵向格式化属性 279
7.2.12 百分数高度 280
7.2.13 自动调整高度 282
7.2.14 折叠纵向外边距 283
7.2.15 负外边距和折叠 285
7.2.16 列表项目 287
7.3 行内元素 288
7.3.1 行布局 288
7.3.2 基本术语和概念 290

7.3.3 行内格式化 292
7.3.4 行内非置换元素 293
7.3.5 行内置换元素 305
7.3.6 行内块级元素 309
7.3.7 流动显示方式 312
7.3.8 contents 显示方式 313
7.3.9 display 的其他值 314
7.3.10 计算值 314
7.4 小结 315
第 8 章 内边距、边框、轮廓和外边距 316
8.1 基本元素框 316
8.2 内边距 319
8.2.1 复值 321
8.2.2 单边内边距 323
8.2.3 内边距的百分数值 324
8.2.4 行内元素的内边距 326
8.2.5 置换元素的内边距 328
8.3 边框 329
8.3.1 边框的式样 330
8.3.2 边框宽度 334
8.3.3 边框颜色 337
8.3.4 简写的边框属性 340
8.3.5 整个边框 342
8.3.6 行内元素的边框 343
8.3.7 圆角边框 344
8.3.8 图像边框 352
8.4 轮廓 369
8.4.1 轮廓式样 369
8.4.2 轮廓宽度 370
8.4.3 轮廓颜色 371
8.4.4 唯一的轮廓简写属性 373
8.4.5 轮廓与边框的区别 373

8.5 外边距 375
8.5.1 外边距的长度值 377
8.5.2 外边距的百分数值 377
8.5.3 单边外边距属性 378
8.5.4 外边距折叠 379
8.5.5 负外边距 381
8.5.6 行内元素的外边距 383
8.6 小结 385

第 9 章 颜色、背景和渐变 386
9.1 颜色 386
9.1.1 前景色 386
9.1.2 对边框的影响 388
9.1.3 对表单元素的影响 390
9.1.4 继承颜色 391
9.2 背景 391
9.2.1 背景色 391
9.2.2 裁剪背景 395
9.2.3 背景图 398
9.2.4 背景定位 402
9.2.5 改变定位框 411
9.2.6 背景重复方式（或不重复） 415
9.2.7 背景粘附 425
9.2.8 控制背景图的尺寸 430
9.2.9 写为一个属性 438
9.2.10 多个背景 440
9.3 渐变 447
9.3.1 线性渐变 447
9.3.2 径向渐变 461
9.3.3 处理渐变图像 473
9.3.4 循环渐变 475
9.4 盒子投影 479
9.5 小结 482

下册

第10章 浮动及其形状 483

10.1 浮动 483

10.1.1 浮动的元素 484

10.1.2 浮动详解 487

10.1.3 具体行为 494

10.1.4 浮动元素与内容重叠 499

10.2 清除浮动 500

10.3 浮动形状 504

10.3.1 定义形状 505

10.3.2 使用透明图像定义形状 517

10.3.3 为形状添加外边距 519

10.4 小结 521

第11章 定位 522

11.1 基本概念 522

11.1.1 定位的类型 522

11.1.2 容纳块 524

11.2 偏移属性 525

11.3 宽度和高度 527

11.3.1 设定宽度和高度 527

11.3.2 限制宽度和高度 529

11.4 内容溢出和裁剪 532

11.5 元素的可见性 534

11.6 绝对定位 536

11.6.1 绝对定位元素的容纳块 536

11.6.2 绝对定位元素的位置和尺寸 539

11.6.3 自动确定边界的位置 541

11.6.4 非置换元素的位置和尺寸 543

11.6.5 置换元素的位置和尺寸 547

11.6.6 Z 轴上的位置 550

11.7 固定定位 555

11.8 相对定位 556

11.9 粘滞定位 559
11.10 小结 563

第12章 弹性盒布局 564
12.1 弹性盒基础 564
12.2 弹性容器 570
12.2.1 flex-direction属性 571
12.2.2 其他书写方向 575
12.2.3 换行 577
12.2.4 定义弹性流 579
12.2.5 flex-wrap 续谈 585
12.3 布置弹性元素 587
12.4 弹性容器 588
12.5 调整内容 588
12.6 对齐元素 596
12.6.1 起边、终边和居中对齐 601
12.6.2 基线对齐 603
12.6.3 补充说明 604
12.7 align-self属性 605
12.8 对齐内容 606
12.9 弹性元素 612
12.9.1 弹性元素是什么 612
12.9.2 弹性元素的特性 613
12.9.3 最小宽度 615
12.10适用于弹性元素的属性 616
12.11 flex 属性 616
12.12 flex-grow属性 618
12.13 flex-shrink属性 625
12.13.1 根据宽度和缩减因子按比例缩小 629
12.13.2 不同的基准 630
12.13.3 响应式弹性布局 632
12.14 flex-basis属性 635
12.14.1 content关键字 636
12.14.2 自动确定弹性基准 636

12.14.3 默认值 639
12.14.4 长度单位 639
12.14.5 零基准 644
12.15 flex简写属性 645
12.16 order属性 650

第13章 栅格布局 657
13.1创建栅格容器 657
13.2基本的栅格术语 660
13.3放置栅格线 662
13.3.1 宽度固定的栅格轨道 665
13.3.2 弹性栅格轨道 669
13.3.3 根据轨道中的内容适配 677
13.3.4 重复栅格线 679
13.3.5 栅格区域 683
13.4 在栅格中附加元素 690
13.4.1 使用列线和行线 690
13.4.2 行和列的简写属性 695
13.4.3 隐式栅格 698
13.4.4 错误处理 701
13.4.5 使用区域 702
13.4.6 栅格元素重叠 705
13.5 栅格流 706
13.6 自动增加栅格线 711
13.7 grid简写属性 714
13.8 释放栅格空间 718
13.8.1 栏距 718
13.8.2栅格元素与盒模型 721
13.9 栅格的对齐方式 726
13.9.1 纵向对齐和横向对齐单个元素 727
13.9.2 纵向对齐和横向对齐全部元素 728
13.10 分层和排序 732
13.11 小结 734

第14章 CSS中的表格布局.....735
14.1 表格格式化.....735
14.1.1 表格的视觉排布.....735
14.1.2 设定显示方式的值.....737
14.1.3 匿名表格对象.....742
14.1.4 表格中的层.....746
14.1.5 表题.....748
14.2 单元格的边框.....749
14.2.1 分离单元格的边框.....750
14.2.2 折叠单元格的边框.....753
14.3 表格的尺寸.....758
14.3.1 宽度.....758
14.3.2 高度.....765
14.3.3 对齐方式.....766
14.4 小结.....769
第15章 列表和生成的内容.....770
15.1 列表.....770
15.1.1 列表的类型.....771
15.1.2 列表项目图像.....774
15.1.3 列表记号的位置.....778
15.1.4 列表样式的简写属性.....779
15.1.5 列表的布局.....780
15.2 生成的内容.....784
15.2.1 插入生成的内容.....784
15.2.2 指定内容.....787
15.2.3 计数器.....793
15.3 定义计数模式.....801
15.3.1 固定计数模式.....803
15.3.2 循环计数模式.....805
15.3.3 符号计数模式.....810
15.3.4 字母计数模式.....814
15.3.5 数字计数模式.....815
15.3.6 累加计数模式.....819

15.3.7 扩展计数模式 821
15.3.8 发音计数模式 823
15.4 小结 824

第16章 变形 825
16.1 坐标系 825
16.2 变形 829
16.3 其他变形属性 848
16.3.1 移动原点 848
16.3.2 选择3D变形方式 852
16.3.3 修改视域 855
16.3.4 处理背面 859
16.4 小结 861

第17章 过渡 862
17.1 CSS过渡 862
17.2 定义过渡的属性 864
17.2.1 限制受过渡影响的属性 868
17.2.2 设置过渡持续时间 874
17.2.3 调整过渡的内部时序 877
17.2.4 延迟过渡 882
17.2.5 transition简写属性 885
17.3 反向过渡：退回起点 887
17.4 支持动画的属性和值 891
17.5 过渡是效果增强 896
17.6 打印过渡 896

第18章 动画 897
18.1 定义关键帧 898
18.2 设置关键帧动画 899
18.3 关键帧选择符 900
18.3.1 省略from和to值 901
18.3.2 重复关键帧属性 903
18.3.3 支持动画的属性 903

18.3.4 不支持动画但不被忽略的属性 905
18.3.5 通过脚本编辑@keyframes动画 905
18.4 把动画应用到元素上 906
18.4.1 指定动画的名称 906
18.4.2 定义动画的时长 909
18.4.3 声明动画的迭代次数 911
18.4.4 设置动画的播放方向 912
18.4.5 延迟播放动画 914
18.4.6 动画事件 916
18.4.7 改变动画的内部时序 925
18.4.8 设置动画的播放状态 938
18.4.9 动画的填充模式 938
18.5 写为一个属性 941
18.6 动画、特指度和优先顺序 945
18.6.1 特指度和!important 945
18.6.2 动画顺序 945
18.6.3 display: none; 对动画迭代的影响 946
18.6.4 动画和 UI 线程 946
18.7 癫痫和前庭功能失调 947
18.8 动画事件及其前缀 947
18.8.1 animationstart 948
18.8.2 animationend 948
18.8.3 animationiteration 949
18.9 打印动画 949

第19章 滤镜、混合、裁剪和遮罩 950

19.1 CSS 滤镜 950
19.1.1 基本滤镜 951
19.1.2 颜色滤镜 953
19.1.3 亮度、对比度和饱和度 954
19.1.4 SVG 滤镜 955
19.2 合成和混合 957
19.2.1 混合元素 957
19.2.2 变暗、变亮、差值和排除 958

19.2.3 正片叠底、滤色和叠加 960
19.2.4 强光和柔光 961
19.2.5 颜色减淡和加深 963
19.2.6 色相、饱和度、明度和颜色 963
19.3 与背景混合 965
19.4 裁剪和遮罩 970
19.4.1 裁剪 971
19.4.2 裁剪形状 972
19.4.3 裁剪框 973
19.4.4 裁剪填充规则 976
19.5 蒙版 977
19.5.1 定义蒙版 978
19.5.2 改变蒙版的模式 981
19.5.3 调整蒙版的尺寸和重复方式 983
19.5.4 定位蒙版 985
19.5.5 裁剪和合成蒙版 988
19.5.6 写为一个属性 991
19.5.7 蒙版类型 992
19.5.8 遮罩边框图像 993
19.6 对象填充和定位 994

第20章 针对特定媒体的样式 998

20.1 定义针对特定媒体的样式 998
20.1.1 基本的媒体查询 998
20.1.2 复杂的媒体查询 1001
20.2 分页媒体 1008
20.3 小结 1023

附录 A 支持动画的属性 1024

附录B 基本属性参考 1032

附录C 颜色对照表 1044

前言

作为 Web 设计师或文档编写人员，如果你对丰富的页面样式感兴趣，想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的。阅读本书之前，你只需了解 HTML 4.0，知道的知识越多越好，但这不是强制要求。除此之外，对其他知识没有什么特别要求。

这是本书第四版，于 2017 年年中完稿，尽量展现了时下最新的 CSS 技术。本书深入说明的 CSS 特性是写作本书时得到浏览器广泛支持的，或者是在本书出版后不久便会支持的。那些尚在制订中的，或者即将弃用的特性则不在讨论之列。

排版约定

本书采用下述排版约定（不过有些例外，详情参见后面的“取值句法约定”）：

斜体（*Italic*）

表示新术语、URL、电子邮件地址、文件名和扩展名。

等宽字体（`Constant Width`）

表示程序清单，在段落中出现则表示程序元素，例如变量、函数名、数据库、数据类型、环境变量、语句和关键字。

粗体等宽字体（**`Constant width bold`**）

表示应该由用户输入的命令或其他文本。

斜体等宽字体（*`Constant Width Italic`*）

表示应该替换成用户提供的值，或者由上下文决定的值。



表示提示或建议。



表示一般性说明。



表示警告或提醒。

取值句法约定

本书穿插大量文本框，详细说明某个 CSS 属性，其中包括允许取哪些值。这些内容基本上原封不动摘自 CSS 规范，不过也会对句法做些说明。

总的来说，属性允许取的值采用下述句法给出：

取值： *<family-name>#*

取值： *<url>* || *<color>*

取值： *<url>*? *<color>* [/ *<color>*]?

取值： [*<length>* | thick | thin]{1,4}

“<”和“>”之间的斜体字表示值的类型，或者是引用其他属性的值。例如，font 接受原本属于 font-family 属性的值，因此使用 *<font-family>* 表示。类似地，如果允许使用颜色值，将使用 *<color>* 表示。

以等宽字体显示的值是关键字，必须原样编写，不能加引号。正斜线（/）和逗号（,）也必须原样使用。

取值范围有多种组合方式：

- 连在一起只由空格分开的两个或多个关键字，必须以给定的顺序使用。例如，help me 表示这个属性必须以这样的顺序取值。
- 以竖线（|）分隔的值（X | Y），必取其中一个值，而且只能取一个。例如，[X | Y | Z] 表示可以取 X、Y 或 Z。

- 两个竖线分隔的值（X ‖ Y）表示既可以取其中一个值，也可以同时取两个值，但是顺序任意。因此，X、Y、X Y 和 Y X 都是有效的值。
- 两个 & 连接的值（X && Y）表示必须同时取 X 和 Y，但是顺序任意。因此，X Y 或 Y X 都是有效的值。
- 方括号（[…]）用于分组。例如，[please ‖ help ‖ me] do this 表示 please、help 和 me 的顺序任意，但是必须出现一次；do this 必须出现，而且必须以指定的顺序。例如：please help me do this，help me please do this，me please help do this。

各部分或方括号分组后面可以跟有下述修饰符中的一个：

- 星号（*）表示前面的值或方括号分组可以重复零次或多次。因此，bucket* 表示 bucket 这个词可以出现任意次，包括零次。可出现的次数没有上限。
- 加号（+）表示前面的值或方括号分组可以重复一次或多次。因此，mop+ 表示 mop 这个词至少要出现一次，此外还可以出现多次。
- 井号（#）表示前面的值或方括号分组可以重复一次或多次，而且多次出现时要以逗号分隔。因此，floor# 可以是 floor，floor，floor，floor 等。这个修饰符最常与方括号分组或值的类型一起使用。
- 问号（?）表示前面的值或方括号分组是可选的。例如，[pine tree]? 表示 pine tree 这两个词不是必须的（如果出现，必须以给定的顺序使用）。
- 感叹号（!）表示前面的值或方括号分组是必须的。因此，即使句法看起来与想表达的意思不同，最终必须得到至少一个值。例如，[what? is? happening?]! 表示这三个词中至少有一个是可选的。
- 花括号里的一对数（{M,N}）表示前面的值或方括号分组至少重复 M 次，最多重复 N 次。例如，ha{1,3} 表示 ha 可以出现一次、两次或三次。

下面举几个例子：

give ‖ me ‖ liberty

这三个词中至少要使用一个，而且顺序任意。例如，give liberty、give me、liberty me give 和 give me liberty 都是有效的值。

[I | am]? the ‖ walrus

I 和 am 只能取其一，不能同时使用，而且可以两个都不用。此外，后面必须跟着 the 或 walrus，抑或二者同时出现，但顺序不限。因此，可以得到 I the walrus、am walrus the、am the、I walrus、walrus the 等。

`koo+ ka-choo`

一个或多个 koo，而且后面必须跟着 ka-choo。因此，koo koo ka-choo、koo koo koo ka-choo 和 koo ka-choo 都是有效的。理论上，koo 的数量是不限的，但是实现时往往会有上限。

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

这句话表达了所有 Web 设计师的想法，可以得到 I love Netscape、I really love Microsoft 等表达式。really 可以出现 0~4 次，但是之间不用逗号分隔。此外，还要在 love 和 hate 之间选一个，这其实是一种隐喻。

`It's a [ mad ]# world`

mad 可以出现任意次（至少一次），之间以逗号分隔。如果只出现一次，不加逗号。因此，It's a mad world 和 It's a mad, mad, mad, mad, mad world 都是有效的值。

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

Alpha、Baker 和 Cray 可以出现 2~3 次，而且后面必须跟着 and Delphi。Cray, Alpha, and Delphi 是其中一种情况。注意，这里有逗号，因为逗号在方括号分组内部（有些旧版 CSS 使用这种方式指明逗号，而不使用 # 修饰符）。

使用代码示例

只要发现▶这样的图标，就表明有对应的示例代码。线上示例的地址是 *https://meyerweb.github.io/csstdg4figs/*。如果阅读本书时使用的设备能联网，可以单击▶图标直接访问示例代码的线上版本。

用于生成书中所有插图的 HTML、CSS 和图像文件可到 *https://github.com/meyerweb/csstdg4figs* 下载。这个仓库的 README.md 文件对里面的内容做了详细说明，记得读一下。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："CSS: The Definitive Guide by Eric A. Meyer and Estelle Weyl (O'Reilly). Copyright 2018 Eric Meyer, Estelle Weyl, 978-1-449-39319-9"。

如果你觉得自己对示例代码的用法超出了上述许可的范围，请通过 permissions@oreilly.com 与我们联系。

O'Reilly Safari

Safari（以前叫 Safari Books Online）是一个会员制平台，为企业、政府、教学人员和个人提供培训和参考资料。

会员可以访问上千种图书、培训视频、学习路径、交互式教程和精心制定的播放列表。这些资源由 250 多家出版社提供，包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett 和 Course Technology 等。

详情请访问 *http://oreilly.com/safari*。

联系我们

请把对本书的评价和问题发给出版社。

美国：

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

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

勘误、示例和其他信息可到 *http://shop.oreilly.com/product/0636920012726.do* 上获取。

对本书的评论或技术疑问，可以发电子邮件到 *bookquestions@oreilly.com*。

欲了解本社图书、课程、会议和新闻等更多信息，请访问我们的网站 *http://www.oreilly.com*，*http://www.oreilly.com.cn*。

我们的 Facebook 地址：*http://facebook.com/oreilly*。

请关注我们的 Twitter 动态：*http://twitter.com/oreillymedia*。

我们的 YouTube 视频地址：*http://www.youtube.com/oreillymedia*。

致谢

Eric Meyer

不论是从技术层面来说，还是就我个人而言，本书这一版都是我参与过的最具挑战的项目。在撰写本书的过程中，我得到了很多人的鼎力帮助。我尽量记住每个人，如有遗漏，深感抱歉。

CSS 的创造人 Håkon Wium Lie 和 Bert Bos，感谢你们的先见之明，没有你们的辛勤付出，Web 不会像现在这样光鲜亮丽。

感谢各浏览器厂商所做的努力，你们紧密团结，创造了很多不可能。

感谢版本控制软件 Git，让我写的一整章（第 15 章）失而复得。

特别感谢 Simon St. Laurent，他认同我一次发布一章的想法，在我忙于其他事务时默默等待，而且对我不离不弃，不管我提议怎么做，都协助我向前推进。

感谢合著人 Estelle，她在我需要时给予帮助，而且眼光敏锐，挑出了不少问题，为这一版做出了杰出贡献。由衷感激！

还有一些人在我不理解时向我一遍遍解说，让我进一步理解了 CSS。其中一些人帮我整理以前的几版，不过多数人为这一版所讲的 CSS 新特性献言献策。他们是（按姓氏字母顺序）：Rachel Andrew，Rossen Atanossov，Tab Atkins，Amelia Bellamy-Royds，Dave Cramer，Elika Etemad，Jen Simmons，Sara Soueidan，Mel Sumner 和 Greg Whitworth。如有遗漏，深感抱歉。

感谢整个 Web 设计师和开发者社区，你们陪我一起度过了最艰难的岁月；同样感谢朋友们、同事们和陌生人们，我对你们的感激无法言表。你们就是我的全部。感谢你们的倾听。

还要感谢我的家人，我的妻子 Kat，我的孩子 Carolyn、Rebecca 和 Joshua，你们是庇护我的港湾，是晴空中的艳阳，是指引我前行的星光。没有你们，我不可能取得今日的成就。感谢你们为我做的一切！

——Eric A. Meyer

于俄亥俄州克利夫兰高地

Estelle Weyl

感谢致力于改进 CSS 的每个人，感谢促进技术多元化和包容性的每一个人。

感谢不辞辛苦与浏览器厂商和开发者并肩协作，共同编写 CSS 规范的人。没有 CSS 工作组（过去、现在和未来）的各位成员，就没有规范、没有标准，也没有跨浏览器兼容性。你们苦苦思索，判断应该把哪些 CSS 属性添加到规范中，哪些属性则要删除，对此我深表敬佩。Tab Atkins、Elika Emitad、Dave Baron、Léonie Watson 和 Greg Whitworth 等人不仅参与撰写规范，还要花时间为广大 CSS 用户（尤其是我）答疑和解释细节。

我还要感谢那些深入讲解 CSS 特性的人（不管有没有加入 CSS 工作组），包括 Sarah Drasner、Val Head、Sara Souidan、Chris Coyier、Jen Simmons 和 Rachel Andrew，是你们把晦涩的规范以浅显易懂的文字表达出来。此外，感谢那些创建便捷工具的人们，你们解放了 CSS 开发者的生产力，尤其是创建并维护 *https://caniuse.com/* 的 Alexis Deveria。

还要感谢那些贡献时间和精力，致力于增进开发者社区多元化和包容性的人们，向你们表示敬意。CSS 的蓬勃发展离不开优秀社区中优秀的人。

2007 年我第一次参加技术大会时，93% 的演讲者是男性。那时我没有想到的是，在接下来的十年间，有如此多的无名英雄在努力提升技术领域和日常生活中的多元化和包容性。

无私奉献、默默无闻的人太多了，有些甚至根本不知道姓甚名谁，不过我要着重提出几个人。Women Who Code Atlanta 的 Erica Stanley、Callback Women 的 Carina Zona 和 Oakland Maker Space 的 Jenn Mei Wu，她们的正能量影响力太大了。除了个人之外，还有一些组织，例如 The Last Mile、Black Girls Code、Girls Incorporated、Sisters Code 等不一而足。我专门创建了一个列表：*http://www.standardista.com/feeding-the-diversity-pipeline/*，为的是证明选择 Web 开发这条路并不是某些人的特权。

感谢所有人，感谢每一个人。感谢你们的付出，你们付出的艰辛是我十年前坐在会场中无法想象的。

——Estelle Weyl
于加州帕洛阿尔托

第 1 章

CSS 和文档

层叠样式表（Cascading Style Sheet，CSS）是一个强大的工具，能影响一个或一组文档的表现。CSS 几乎触及 Web 的每个角落，甚至很多非 Web 环境也能见到它的身影。例如，基于 Gecko 的浏览器使用 CSS 影响窗体自身的表现，很多 RSS 客户端允许使用 CSS 定制订阅源和订阅源中的文章，而且有些即时通信客户端使用 CSS 装饰聊天窗口。JavaScript 框架，甚至是 JavaScript 自身使用的句法中处处可见 CSS 的身影。

1.1 Web 样式简介

1994 年，正值 Web 开始广泛流行开来，CSS 的第一个提案发布了。那时，浏览器为用户提供了各种样式的定制功能。例如，用户在 Mosaic 的表现偏好设置中可以为单个元素设定字体族、字号和颜色。而文档编写人员却做不到这一点。文档编写人员只能把内容标记成一个个段落、一级级标题、一块块预排格式文本，或者一些其他类型的元素。如果用户愿意，可以把一级标题设为粉红色的小字，而把六级标题设为红色的大字。

CSS 就是在这样的背景下诞生的。它的目标是提供一个简单的声明式样式语言，而且具有一定的灵活性，能为文档编写人员和用户提供等同的样式化功能。层叠样式表中的“层叠”是指样式可以结合起来使用，而且具有优先级，文档编写人员和用户都有话语权，不过最终的决策权在用户手中。

草案制定的速度很快，到 1996 年年末，CSS1 完成了。此后，刚组建的 CSS 工作组开始着手制定 CSS2，而各浏览器则相互协作，努力实现 CSS1。单独来看，CSS 的每一部分都很简单，但把各部分放在一起就变得异常复杂。而且早期实现有些先天不足，例如不同浏览器对盒模型（box model）的实现之间的差异尤其为人诟病。这些问题直接影

响到 CSS 的名声，幸好一些聪明人提出了变通方法，让浏览器的行为保持了一致。得益于一致性的提高和高调的开发活动，例如使用 CSS 重新设计 Wired 杂志和 CSS Zen Garden，几年之间，CSS 逐渐开始流行。

不过，在此之前，CSS2 规范于 1998 年年初定案。随后，CSS 工作组立即投身 CSS3 的制定工作，以及 CSS2 的修订工作（制定 CSS2.1）。与以往不同的是，CSS3 由多个（理论上）独立的模块构成，而不是单独一个臃肿的规范。XHTML 规范受此启发，也采用了这种模块式机制。

CSS3 分成多个模块的根本原因是各模块可以独立演进，尤其是重要的（或受众广的）模块可以按照 W3C 的规划向前推进，而不必受其他模块拖累。事实证明，这样做是对的。截至 2012 年年初，有三个 CSS3 模块（CSS Color Level 3、CSS Namespaces 和 Selectors Level 3）变成了全力推荐状态，而有七个模块处于候选状态，还有七个模块处在不同的草案状态。如果采用以前的机制，要等其他部分完成才能在一份完成的规范中发布颜色、选择符和命名空间的新条款。得益于模块化，我们无需再等待。

但是，这样做也有缺点，即“CSS3 规范”不能涵盖一切。世界上不再有这种叫法，也不可能再有。即便其他模块在某一时刻到达了 Level 3，比如说 2016 年年末（然而并没有），Selectors Level 4 也都开始制定了。那会不会有 CSS4？“CSS3”那些尚未正式发布的新特性呢？还有 Grid Layout，它甚至还没到 Level 1？

可见，我们不能指着一摞厚厚的文件说，“这就是 CSS3”，而应该分模块学习不同的特性。模块的灵活性有时可以弥补由此引起的语义不足（如果你非要一份独立的完整规范，可以留意 CSS 工作组每年发布的“Snapshot”文档）。

了解这些背景之后，就可以开始学习 CSS 了。不过在此之前要先了解标记。

1.2 元素

元素（element）是文档结构的根基。HTML 中常用的元素有 p、table、span、a 和 div 等。文档中的每个元素都对文档的表现起一定作用。

1.2.1 置换元素和非置换元素

CSS 依赖元素，但并非每个元素都以同样的方式创建。例如，图像和段落是不同类型的元素，span 和 div 也不同。对 CSS 来说，元素通常有两种形式：置换元素和非置换元素。

置换元素

置换元素（replaced element）指用来置换元素内容的部分不由文档内容直接表示。在HTML 中，最常见的置换元素要数 img，它的内容由文档之外的图像文件替换。其实，通过下面这个简单的例子可以看出，img 元素没有内容：

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

这段标记只包含一个元素名和一个属性。如不指向外部内容（这里通过 src 属性指定一个图像），这个元素什么也表示不了。如果指向的图像文件存在，文档会把那个图像显示出来；否则，浏览器什么也不显示，或者显示“图像损坏”占位图。

input 元素类似，根据类型的不同，会替换成单选按钮、复选框或文本输入框。

非置换元素

HTML 元素大部分是非置换元素（nonreplaced element），即元素的内容由用户代理（通常是浏览器）在元素自身生成的框中显示。例如，hi there 是非置换元素，用户代理会显示“hi there”文本。段落、标题、单元格、列表，以及 HTML 中其他几乎所有元素都是非置换元素。

1.2.2 元素的显示方式

除了置换元素和非置换元素之外，CSS 还把元素分成块级和行内两种基本类型。除此之外，还有其他显示类型，不过这两种是最常见的。对编写过 HTML 标记的人来说，肯定知道它们在 Web 浏览器中的显示方式。图 1-1 中有这两种元素的示例。

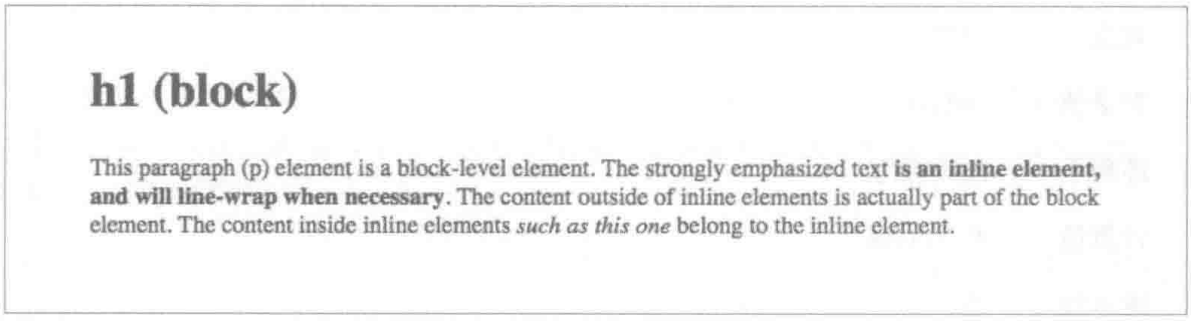


图 1-1：一个 HTML 文档中的块级元素和行内元素

块级元素

块级元素（默认）生成一个填满父级元素内容区域的框，旁边不能有其他元素。也就是说，块级元素在元素框的前后都“断行”。HTML 中最常见的块级元素是 p 和 div。置换元素可以是块级元素，但往往不是。

列表项目是一种特殊的块级元素，它的表现与其他块级元素没有区别，此外还会在元素框旁生成一个记号（无序列表通常是圆点，有序列表通常是数字）。除了多出的这个记号以外，列表项目与其他块级元素之间没有任何区别。

行内元素

行内元素在一行文本内生成元素框，不打断所在的行。HTML 中最常见的行内元素是 a。此外还有 strong 和 em。这类元素不在自身所在元素框的前后“断行”，因此可以出现在另一个元素的内容中，且不影响所在的元素。

注意，HTML 中的块级元素和行内元素虽然有诸多共同点，但是它们之间有个重要的区别：在 HTML 中，块级元素不能出现在行内元素中。但是 CSS 并不限制它们的显示方式，相互之间可以嵌套。

为了进一步了解，下面来看一个 CSS 属性 display。

display

取值	[*<display-outside>* ‖ *<display-inside>*] | *<display-listitem>* | *<display-internal>* | *<display-box>* | *<display-legacy>*
定义	见下
默认值	inline
适用于	所有元素
计算值	指定的值
继承性	否
动画性	否

<display-outside>

```
block | inline | run-in
```

```
<display-inside>
   flow | flow-root | table | flex | grid | ruby

<display-listitem>
   list-item && <display-outside>? && [ flow | flow-root ]?

<display-internal>
   table-row-group | table-header-group | table-footer-group |
   table-row | table-cell | table-column-group | table-column |
   table-caption | ruby-base | ruby-text | ruby-base-container |
   ruby-text-container

<display-box>
   contents | none

<display-legacy>
   inline-block | inline-list-item | inline-table | inline-flex |
   inline-grid
```

可以看出，display 属性可以取很多值，前面只提过其中三个：block、inline 和 list-item。其他多数值将在本书的其他章节讨论，例如，grid 和 inline-grid 在专门探讨栅格布局的一章讨论，表格相关的值在探讨 CSS 表格布局的一章讨论。

现在我们只关注 block 和 inline。来看下述标记：

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

这里有两个块级元素（body 和 p）及一个行内元素（em）。根据 HTML 规范，em 可以放在 p 里，而反过来却不行。一般地，HTML 层次结构要求，行内元素可以放在块级元素中，反之则不行。

与此不同，CSS 没有这种限制。在不改变标记的前提下，我们可以像下面这样改变它们的显示方式：

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

这会导致行内框中出现一个块级框。这完全是有效的，不违背任何 CSS 规则。然而，如果在 HTML 中调换元素之间的嵌套关系，就会出问题：

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

不管通过 CSS 如何改变显示方式，这都不是有效的 HTML。

改变元素的显示方式对 HTML 文档来说是有用的，不过对 XML 文档而言的作用更大。XML 文档中的元素没有固定的显示方式，而是完全由编写人员定义。例如，试想一下下述 XML 片段将如何显示：

```
<book>
 <maintitle>Cascading Style Sheets: The Definitive Guide</maintitle>
 <subtitle>Third Edition</subtitle>
 <author>Eric A. Meyer</author>
 <publisher>O'Reilly and Associates</publisher>
 <pubdate>November 2006</pubdate>
 <isbn type="print">978-0-596-52733-4</isbn>
</book>
<book>
 <maintitle>CSS Pocket Reference</maintitle>
 <subtitle>Third Edition</subtitle>
 <author>Eric A. Meyer</author>
 <publisher>O'Reilly and Associates</publisher>
 <pubdate>October 2007</pubdate>
 <isbn type="print">978-0-596-51505-8</isbn>
</book>
```

因为 display 属性的默认值是 inline，所以上述内容默认将显示为行内文本，如图 1-2 所示。可以看出，这样显示没有什么用。

Cascading Style Sheets: The Definitive Guide Third Edition Eric A. Meyer O'Reilly and Associates November 2006 978-0-596-52733-4 CSS Pocket Reference Third Edition Eric A. Meyer O'Reilly and Associates October 2007 978-0-596-51505-8

图 1-2：XML 文档的默认显示方式

可以用 display 定义基本布局：

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

我们把七个元素中的五个设为块级元素，另外两个设为行内元素。这意味着，各块级元素将像 HTML 中的 div 那样处理，两个行内元素将像 span 那样处理。

由于具有这种影响显示方式的能力，使得 CSS 在众多情况下都十分有用。我们可以在上述规则的基础上再添加一些样式，得到更好的视觉效果，如图 1-3 所示。

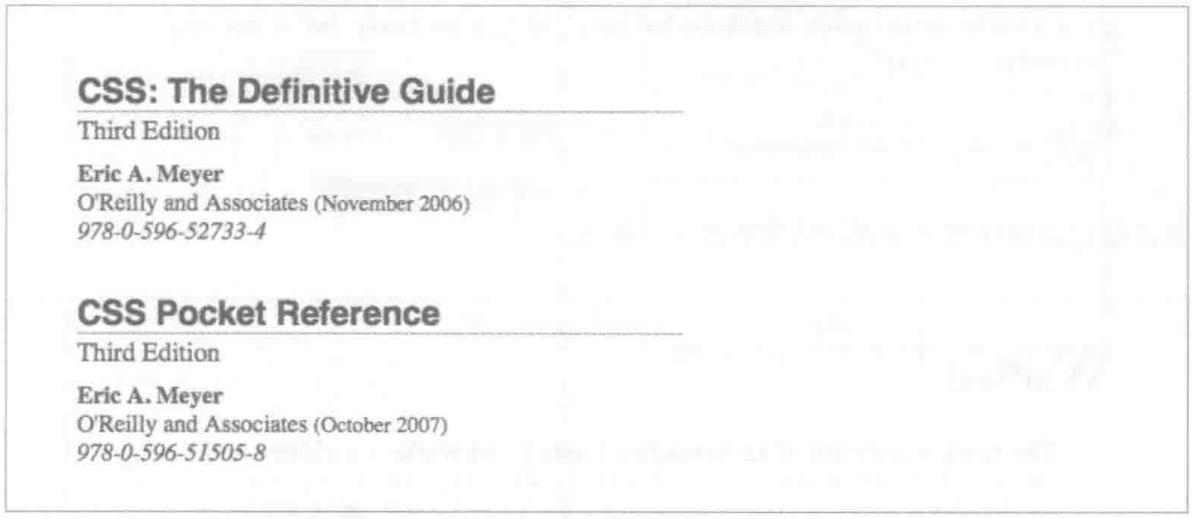


图 1-3：添加样式后的 XML 文档

在学习如何编写 CSS 之前，要先知道如何把 CSS 关联到文档上。毕竟，如果不这么做，CSS 就没办法影响文档。我们将说明如何在 HTML 文档中关联 CSS，因为这是最常用的方式。

1.3 把 CSS 应用到 HTML 上

HTML 文档内部有一定的结构，这一点前面已经提及，但是有必要重申一次。以前的网页往往忽略了这一点，我们经常忘记文档内部应该有一定的结构，而这与视觉结构完全是两码事。我们可能急于创建最酷的网页，可能会以各种方式摆放页面的内容，但却忘了网页应该包含结构化的信息。

这种结构正是 HTML 和 CSS 之间固有关系的一部分，如若不然，二者之间便是泾渭分明。为了更好地理解这一点，下面以一个 HTML 文档为例，逐一说明各部分：

```
<html>
<head>
<title>Eric's World of Waffles</title>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<link rel="stylesheet" type="text/css" href="sheet1.css" media="all">
<style type="text/css">
/* These are my styles! Yay! */
@import url(sheet2.css);
</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>
```

这段标记应用样式后得到的结果如图 1-4 所示。

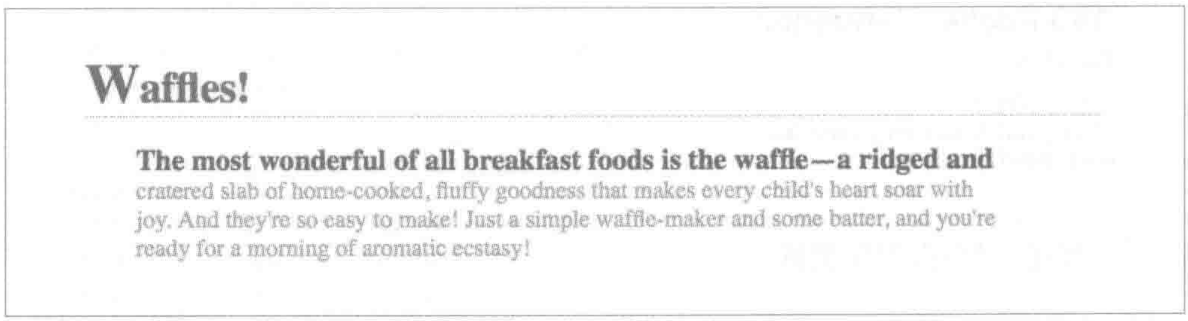


图 1-4：文档示例

下面说明这个文档关联 CSS 的不同方式。

1.3.1 link 标签

先看 link 标签：

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

link 标签少有关注，但它完全是有效的标签，在 HTML 规范中已经存在多年，一直等待重用。它的基本作用是把其他文档与当前文档关联起来。CSS 使用它链接应用到文档上的样式表。图 1-5 中的文档链接了一个名为 *sheet1.css* 的样式表。

通过 link 标签链接的样式表不是 HTML 文档的一部分，但却供文档使用。我们称这样的样式表为外部样式表（external stylesheet），因为样式表在 HTML 文档外部（想想看）。

为了正确加载外部样式表，link 标签必须放在 head 元素中，不能放在其他元素中。Web 浏览器遇到 link 标签时，会查找并加载指定的样式表，使用样式表中的样式渲染 HTML 文档。这一过程如图 1-5 所示。图中还通过 @import 声明加载了另一个外部样式表 *sheet2.css*。@import 声明必须放在所在样式表的开头，此外别无限制。

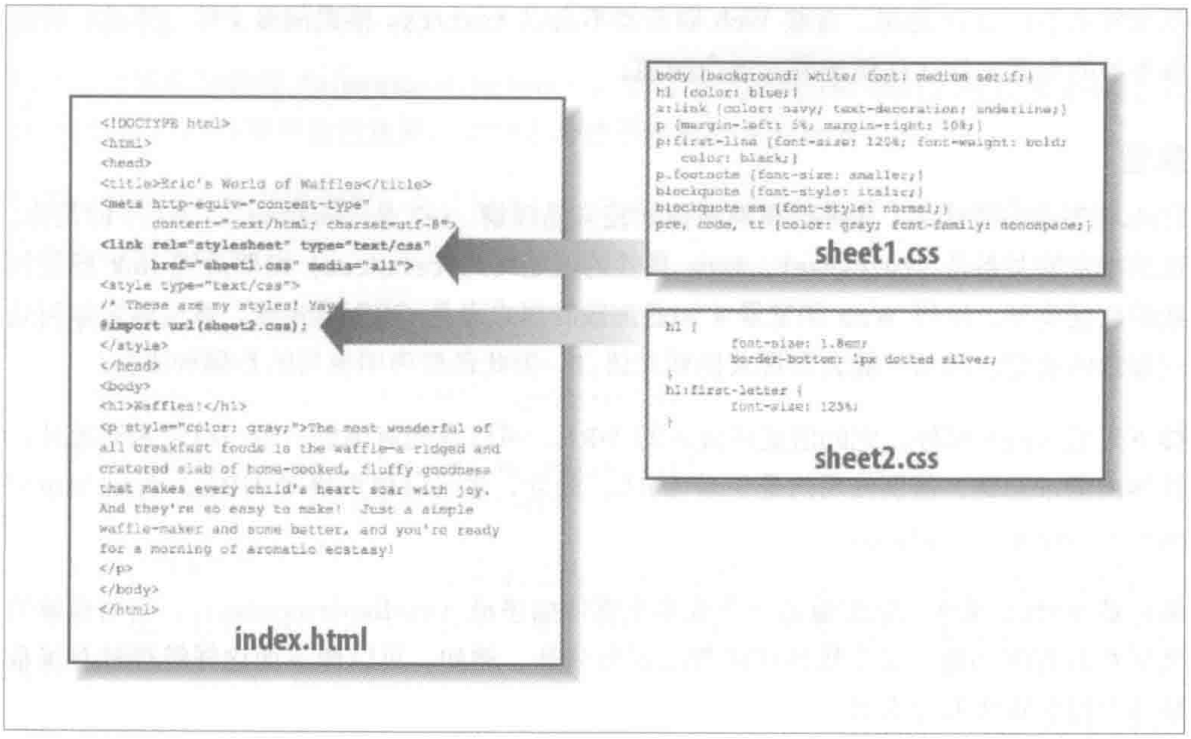


图 1-5：文档应用外部样式表的方式

外部样式表中包含什么呢？是一系列规则，与前一节和本节的示例 HTML 文档中的类似。只不过，这里的规则存储在单独的文件中。注意，样式表中不能有 HTML 或其他标记语言，只能包含样式规则。下面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，没有 HTML 标签或注释，只有一些简单的样式声明。外部样式表保存为纯文本文件，扩展名通常是 *.css*，例如 *sheet1.css*。



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（参见本章后文）。如果外部样式表中有标记，可能导致部分或全部样式失效。

文件扩展名不是必须的，但是如果文件名不以 *.css* 结尾，即使在 `link` 元素中把 `type` 属性的值设为 `text/css`，有些旧的浏览器也不会将其识别为包含样式表的文件。其实，如

果文件名不以 *.css* 结尾，有些 Web 服务器不会以 text/css 格式伺服文件。不过，修改服务器的配置文件往往便能解决这个问题。

属性

link 标签余下的内容，即那些属性和值比较容易理解。rel 是“relation”（关系）的简称，这里指定的关系是 stylesheet。type 属性的值始终为 text/css，说明通过 link 标签加载的数据类型。这样 Web 浏览器才知道加载的样式表是 CSS 样式表，然后确定如何处理加载的数据。以后可能会出现其他样式语言，因此最好声明使用的是哪种语言。

接下来是 href 属性，它的值是样式表的 URL，可以是绝对地址，也可以是相对地址，具体由需求而定。前例使用的是相对 URL。此外，也可以使用绝对 URL，例如 *http://meyerweb.com/sheet1.css*。

最后是 media 属性，它的值是一个或多个媒体描述符（media descriptor），指明媒体的类型和具有的功能。多个媒体描述符以逗号分开。例如，可以像下面这样链接针对屏幕媒体和投影媒体的样式表：

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

媒体描述符有时很复杂，本章后文再详细说明。现在就以所示的基本媒体类型为例。

注意，一个文档可以关联多个样式表。如果是这样，最初显示文档时只会使用 rel 属性的值为 stylesheet 的 link 标签链接的样式表。因此，如果想链接两个分别名为 *basic.css* 和 *splash.css* 的样式表，可以这么做：

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

这样，浏览器会加载指定的两个样式表，合并样式规则后再应用到文档上。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>
```

这个示例中没有出现 title 属性，它也是 link 标签的有效属性，但不常用。不过以后可能用得着，如果使用不当，还可能导致意料之外的影响。为什么呢？下一节探讨。

候选样式表

此外还有候选样式表（alternate stylesheet），定义方式为把 rel 属性的值设为 alternate stylesheet。仅当用户自己选择，文档才会使用候选样式表渲染。

如果浏览器支持候选样式表，会使用 link 元素 title 属性的值生成候选样式列表。对下面的示例来说：

```
<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!">
```

浏览器默认使用第一个样式表（这里是名为“Default”的样式表），此外用户还可以自行选择想使用的样式表。图 1-6 展示的是一种选择样式的方式（其实这是 CSS 开始崭露头角之时的方式）。

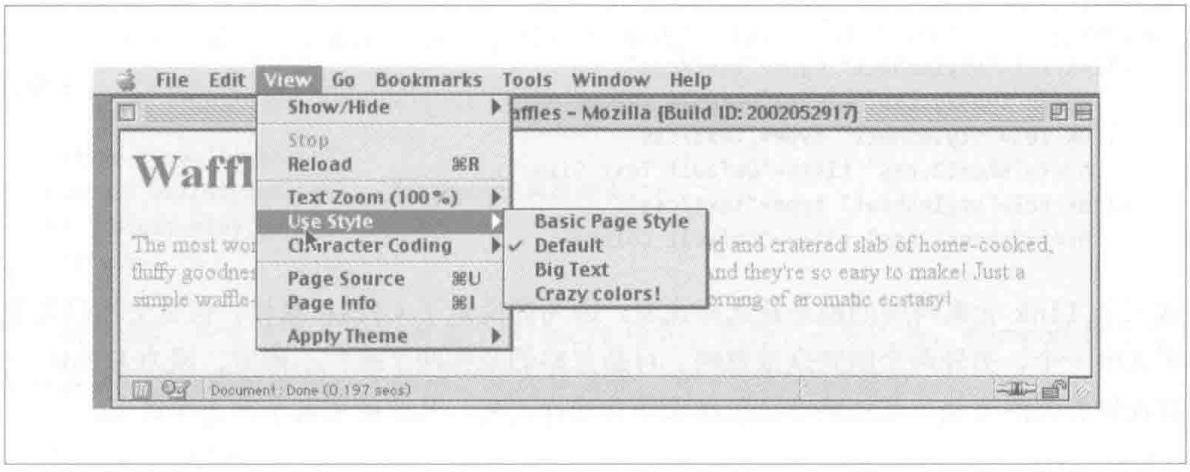


图 1-6：浏览器提供候选样式表供选择



截至 2016 年年末，多数基于 Gecko 的浏览器，如 Firefox 和 Opera，都支持候选样式表。Internet Explorer 原生不支持，不过可以借助 JavaScript。基于 WebKit 的浏览器不支持选择候选样式表。与图 1-6 中远古时代的浏览器相比，简直难以置信。

此外，还可以为不同的候选样式表设定相同的 title 值，把它们分组放在一起。利用这一点，用户可以为屏幕和印刷媒体选择不同的外观。

```
<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">
```

如果用户在支持候选样式表的用户代理中选择“Big Text”，屏幕媒体将使用 *bigtext.css* 装饰文档，印刷媒体将使用 *print-bigtext.css*，任何媒体都不会使用 *sheet1.css* 或 *print-sheet1.css*。

为什么会这样呢？因为一旦为属性为 rel 的 stylesheet 的 link 元素设定标题，也就是将其定为首选样式表（preferred stylesheet）。这意味着，首选样式表优先于候选样式表，显示文档时默认使用。而选择候选样式表之后，首选样式表就不使用了。

另外，如果有一组首选样式表，那么只会使用其中一个，其他的则被忽略。比如下面这个例子：

```
<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">
```

这三个 link 元素声明的都是首选样式表，因为都设定了 title 属性，但是文档只会使用其中一个，另外两个则完全被忽略。可是忽略的是哪两个呢？不确定，因为 HTML 没有提供相关的方法，无法确定该忽略哪些首选样式表，又该使用哪个首选样式表。

如果不为样式表设定标题，那它就是永久样式表（persistent stylesheet），始终用于显示文档。这通常正是文档编写人员想要的行为。

1.3.2 style 元素

style 元素也是一种引入样式表的方式，直接写在文档中：

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

style 元素应该始终设定 type 属性。对 CSS 文档来说，正确的值是 "text/css"，这与 link 元素是一样的。

如前例所示，style 元素始终以 <style type="text/css"> 开头，后跟一个或多个样式，然后以 </style> 标签结尾。style 元素也有 media 属性，与通过 link 元素链接的样式表功能一样。

开始和结束 style 标签之间的样式称为文档样式表（document stylesheet）或嵌入式样式表（embedded stylesheet，因为这种样式表内嵌在文档中）。style 元素可以直接包含应用到文档上的样式，也可以通过 @import 指令引入外部样式表。

1.3.3 @import 指令

下面来看可以出现在 style 标签中的内容。首先是链接的外部样式表中也有的 @import 指令：

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

与 link 一样，Web 浏览器遇到 @import 指令时会加载外部样式表，使用其中的样式渲染 HTML 文档。二者之间唯一的主要区别在于句法和指令的位置。可以看出，@import 指令在 style 元素内部，而且必须放在其他 CSS 规则前面，否则不会起作用。比如下面的例子：

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

与 link 一样，一个文档中可以有多个 @import 语句。然而，不同的是，@import 指令导入的每个样式表都会使用，无法指定候选样式表。对下面的代码来说：

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

三个外部样式表都会加载，而且其中的所有样式都会用于显示文档。

与 link 类似，@import 指令也可以显示导入的样式表应用于何种媒体。方法是在样式表的 URL 后面提供媒体描述符：

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

媒体描述符有时很复杂，我们将在第 20 章讨论。

如果一个外部样式表需要用到另一个外部样式表中的样式，@import 指令的作用就体现出来了。我们知道，外部样式表不能包含任何文档标记，也就是不能使用 link 元素，但是可以使用 @import 指令。因此，外部样式表中可能包含下述内容：

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

当然，你可能不会使用跟这一样的样式，但是希望你能从中看出该怎么做。注意，前例即使用了绝对 URL，也用到了相对 URL，这一点与 link 元素一样。

此外还要注意，与内嵌在文档中一样，@import 指令写在样式表的开头。CSS 要求样式表中的 @import 指令必须在所有样式规则前面。遵守规范的用户代理会忽略放在样式规则（例如 body {color: red;}）后面的 @import 指令。



Windows 系统中的旧版 Internet Explorer 不会忽略任何 @import，即便出现在其他规则后面也不忽略。由于其他浏览器会忽略位置不对的 @import 指令，所以很容易错误地将 @import 指令放在不当的位置，从而改变文档在其他浏览器中的显示效果。

1.3.4 HTTP 链接

为文档关联 CSS 还有一种鲜为人知的方式：使用 HTTP 首部。

在 Apache 中，若想使用这种方式，可以在 *.htaccess* 文件中引用 CSS 文件。例如：

```
Header add Link "</ui/testing.css>;rel=stylesheet;type=text/css"
```

这样设置之后，支持这种方式的浏览器在加载受此 *.htaccess* 文件管理的文档时便会使用指定的样式表，就像通过 link 元素链接的样式表一样。此外，还可以在服务器的 *httpd.conf* 文件中添加等效的规则，这样做可能更高效：

```
<Directory /path/to/ /public/html/directory>
Header add Link "</ui/testing.css>;rel=stylesheet;type=text/css"
</Directory>
```

在支持这种方式的浏览器中，这样做的效果与前面一样，唯一的区别是设置所在的文件不同。

你可能注意到了“支持这种方式的浏览器”这种表述。截至 2017 年年末，广泛支持 HTTP 链接样式表的浏览器有 Firefox 系列和 Opera。因此，这种方式最常在使用这些浏览器中的某一款做开发时使用。鉴于此，在测试服务器中使用 HTTP 链接便可以把开发中的网站与线上公开的网站区分开。如果出于特殊的原因，想隐藏一些样式，不让 WebKit 和 Internet Explorer 系列使用，也可以使用这种方式。



常用的脚本语言（如 PHP）和 IIS 也有等效的方式，方法是设定 HTTP 首部。此外，还可以使用脚本语言向服务器伺服的文档插入 `link` 元素。考虑到浏览器的支持，这种方案更牢靠，毕竟所有浏览器都支持 `link` 元素。

1.3.5 行内样式

如果只想为单个元素提供少量样式，不值得动用嵌入式样式表或外部样式表，可以利用 HTML 元素的 `style` 属性设置行内样式：

```
<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 `body` 元素之外的标签（如 `head` 或 `title`），所有 HTML 标签都能设定 `style` 属性。

`style` 属性的句法相当简单。其实，与 `style` 元素中的样式规则十分相似，不过是把花括号换成双引号。因此，`<p style="color: maroon; background: yellow;">` 将把文本的颜色设为红褐色，背景色设为黄色，而且只适用于那一段文字，文档中的其他部分不受此影响。

注意，`style` 属性的值只能是一系列规则声明，而不能包含整个样式表。因此，不能在 `style` 属性中使用 `@import` 指令，也不能有完整的规则。`style` 属性的值只能是样式规则花括号之间的那一部分。

通常不建议使用 `style` 属性。而且，除了 HTML，XML 很少使用这个属性。倘若使用 `style` 属性，CSS 的很多重要优点都不复存在了，例如集中管理样式，控制整个文档或网站中所有文档的外观。在很多方面，行内样式都不比 `font` 标签好多少，不过却提供了一定的灵活性。

1.4 样式表中的内容

了解这么多之后，我们不禁要问，样式表中到底有什么内容呢？其实，我们见过的：

```
h1 {color: maroon;}
body {background: yellow;}
```

嵌入式样式表中的多数样式都是这样的，有简单的、有复杂的，有短的、有长的。如果文档中有 style 标签，里面很少没有规则。假如没有上例这样的规则，也会有一些 @import 声明。

在深入讨论之前，我们要总览一下样式表中可以有什么内容，不可以有什么内容。

1.4.1 标记

样式表中不能有标记。这一点看似理所当然，但总是有人在这上面栽跟头。不过 HTML 注释标记例外，由于历史原因，style 元素中可以有 HTML 注释。

```
<style type="text/css"><!--
h1 {color: maroon;}
body {background: yellow;}
--></style>
```

仅此而已。

1.4.2 规则的结构

为了进一步说明规则，下面把规则的结构分解开，逐一说明。

一个规则由两个基本部分构成：选择符（selector）和声明块（declaration block）。声明块由一个或多个声明组成，而一个声明包含一个属性（property）和对应的值（value）。一个样式表由一系列规则构成。图 1-7 展示的是一个规则的各个部分。

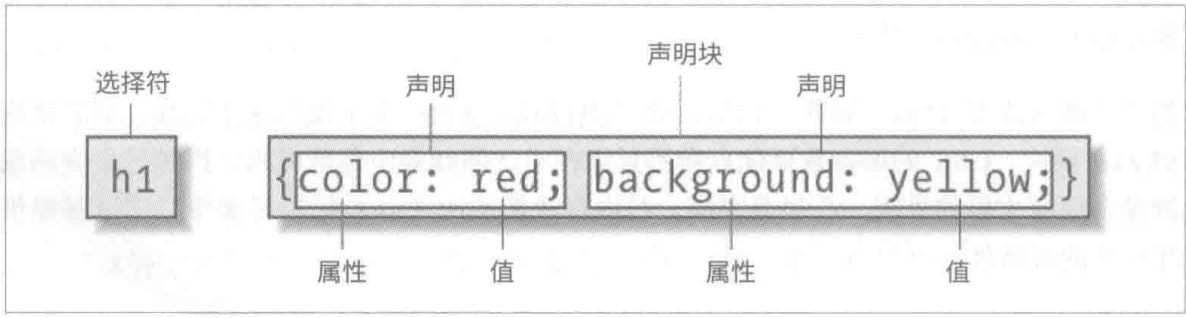


图 1-7：一个规则的结构

选择符（图中靠左的那一部分）定义文档中的哪部分受影响。图 1-7 选择的是 h1 元素。如果选择符是 p，那选择的则是所有 p 元素（段落）。

图中右边那一部分是声明块，由一个或多个声明组成。一个声明包含一个 CSS 属性及其值。图 1-7 中的声明块包含两个声明，第一个声明把文档中部分文本的颜色设为红色，第二个声明把文档中那一部分的背景设为黄色。因此，文档中的所有 h1 元素（由选择符指定）都将显示为黄底红字。

1.4.3 厂商前缀

有时你会发现，CSS 中有些内容的前面有个标注，例如 -o-border-image。这叫厂商前缀（vendor prefix），浏览器厂商通过它标记实验性或专属（或二者兼具）的属性、值或其他内容。截至 2016 年年末，市面上有不少厂商前缀，其中最常用的见表 1-1。

表 1-1：一些常用的厂商前缀

前缀	厂商
-epub-	国际数字出版论坛制定的 ePub 格式
-moz-	基于 Mozilla 的浏览器（如 Firefox）
-ms-	微软 Internet Explorer
-o-	基于 Opera 的浏览器
-webkit-	基于 WebKit 的浏览器（如 Safari 和 Chrome）

从表 1-1 中可以看出，厂商前缀的一般格式是一个英文破折号、一个标注和一个英文破折号。不过也有少量前缀不采用这个格式，开头没有破折号。

使用厂商前缀的原因复杂，不是三言两语就能说清的，也不在本书的讨论范围之内。可以说，厂商前缀出现的目的是浏览器厂商为了测试新特性，这么做能保证兼容性，不必担心被过时的行为限制，导致与其他浏览器不兼容。这样做能避免一系列问题，以免扼制 CSS 的发展。可惜，网页制作者们后来大量使用带前缀的属性，由此引起了一系列新问题。

截至 2016 年年末，厂商前缀的使用趋势有所减缓，因为新版浏览器逐渐删除了对带前缀的属性和值的支持。如今，编写 CSS 时你完全可以不使用厂商前缀，不过偶尔会遇到别人使用，或者在以前的代码基中见到。

1.4.4 处理空白

CSS 对规则之间的空白基本没有严格要求，而且对规则内部的空白大多也没有严格要求，不过有些例外。

一般来说，CSS 对待空白的方式跟 HTML 差不多：解析时，连续的空白会合并成一个空白。因此，虚构的 rainbow 规则可以写成下面这几种格式：

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

当然，除此之外还有其他分隔方式。唯一的要求是，要使用空白分隔，可以是空格、制表符或换行符，可以是单个空白，也可以任意数量随意组合。

同样，规则之间的空白也可以随意使用。下面只是无数种编写方式中的五种：

```
html{color:black;}
body {background: white;}
p {
  color: gray;}
h2 {
     color : silver ;
   }
ol
   {
      color
         :
      silver
         ;
}
```

从第一个规则中可以看出，多数空白可以省略。其实，简化后的 CSS 通常就是这样的，即把多余的空白都删掉。前两个规则之后的三个规则使用的空白量一个比一个多，最后一个规则几乎把所有可以分开的内容都写在单独的一行里。

这些写法都是有效的，你可以从中选一个自己觉得最合理的，即最易于阅读的，然后沿用下去。

其实，有些地方是必须使用空白的。拿前面虚构的 rainbow 示例来说，最常见的是使用空白分隔值中的多个关键字。多个关键字必须使用空白分开。

1.4.5 CSS 注释

CSS 支持注释。与 C/C++ 注释非常相似，CSS 注释也放在 /* 和 */ 之间：

```
/* 这是一个 CSS1 注释 */
```

与 C++ 一样，CSS 注释可以分成多行：

```
/* 这是一个 CSS1 注释，可以分成
多行，完全没有问题 */
```

不过要注意，CSS 注释不能嵌套。因此，下述示例是错误的：

```
/* 这是一个注释，其中还有
 一个注释，这是错的，
   /* 另一个注释 */
 回到第一个注释 */
```



临时注释掉已经包含注释的一大段规则时容易导致注释“嵌套”。CSS 不允许注释嵌套，因此“外层”注释会在“内层”注释的结束处结束。

可惜，CSS 没有一直延续到行尾的注释，即不能使用 // 或 # 等（后者是保留字，用于选择 ID）。在 CSS 中，注释只能使用 /* */ 编写。因此，如果想把注释与规则放在同一行，一定要注意如何放置。例如，下面是正确的方式：

```
h1 {color: gray;}   /* 这个 CSS 注释有好几行 */
h2 {color: silver;} /* 而它放在样式旁边 */
p {color: white;}   /* 因此每一行都要 */
pre {color: gray;}  /* 放在注释标记里 */
```

如果各行没有结束标记，那么样式表的多数内容将变成注释的一部分，从而失去作用：

```
h1 {color: gray;}   /* 这个 CSS 注释有好几行，
h2 {color: silver;}  但是各行没有放在注释标记里，
```

```
p {color: white;}    因此最后三个样式
pre {color: gray;}   变成了注释的一部分 */
```

在这个示例中，只有第一个规则（h1 {color: gray;}）会应用到文档上，余下的规则变成了注释的一部分，被浏览器的渲染引擎忽略。



对 CSS 解析器来说，CSS 注释相当于根本没出现过，并不算作空白。因此，注释可以放在规则内部，甚至可以放在声明内部。

1.5 媒体查询

创作人员通过媒体查询（media query）定义浏览器在何种媒体环境中使用指定的样式表。过去，实现这一机制的方法是通过 link 元素或 style 元素的 media 属性设定媒体类型，或者为 @import 或 @media 指令提供媒体描述符。媒体查询更进一步，允许创作人员通过媒体描述符根据指定媒体类型的特性选择样式表。

1.5.1 用法

媒体查询可以在下述几个地方使用：

- link 元素的 media 属性。
- style 元素的 media 属性。
- @import 声明的媒体描述符部分。
- @media 声明的媒体描述符部分。

媒体查询可以是简单的媒体类型，也可以是复杂的媒体类型和特性的组合。

1.5.2 简单的媒体查询

在介绍媒体查询的各种可能用法之前，先看几个简单的媒体块。假设我们想在投影环境（例如幻灯片）中使用一些不同的样式。下面是这个 CSS 中比较简单的两个规则：

```
h1 {color: maroon;}
@media projection {
   body {background: yellow;}
}
```

针对这个例子，在所有媒体中，h1 元素的颜色都是红褐色，但是，在投影媒体中 body 元素会有一个黄色背景。

一个样式表中可以有任意多个 @media 块，而且每一个都有自己的一套媒体描述符（详情参见本章后文）。如果愿意，可以把所有规则都放在一个 @media 块里，就像下面这样：

```
@media all {
   h1 {color: maroon;}
   body {background: yellow;}
}
```

然而，这与去掉首尾两行的效果一模一样，完全没必要。



这一节采用的缩进方式只是为了条理清晰。@media 块中的规则可以不缩进，如果你觉得缩进能让 CSS 更易于阅读，可以缩进。

上述示例中的 projection 和 all 就是设定媒体查询的位置。媒体查询包含描述媒体类型的词组和对媒体参数的说明（例如分辨率或显示屏高度），决定块中的 CSS 何时应用。

1.5.3 媒体类型

媒体查询最基本的形式媒体类型，由 CSS2 引入。媒体类型就是指明不同媒体的标注：

all
: 用于所有展示媒体。

print
: 为有视力的用户打印文档时使用，也在预览打印效果时使用。

screen
: 在屏幕媒体（如桌面电脑的显示器）上展示文档时使用。在桌面计算机上运行的所有 Web 浏览器都是屏幕媒体用户代理。



写作本书时，有些浏览器也支持 projection 类型，能以幻灯片的形式展示文档。有些移动设备的浏览器支持 handheld 类型，但是行为不一致。

多个媒体类型使用逗号分隔罗列。下面四种方式都能把一个样式表（或一个规则块）同时应用到屏幕媒体和印刷媒体上：

```
<link type="text/css" href="frobozz.css" media="screen, print">
<style type="text/css" media="screen, print">...</style>

@import url(frobozz.css) screen, print;
@media screen, print {...}
```

为媒体类型加上特性描述符（例如，描述指定媒体的分辨率或色深）之后事情就变得有趣了。

1.5.4 媒体描述符

对于在 link 元素或 @import 声明中设定过媒体类型的人来说，一定不会对媒体查询的位置感到陌生。下面两种方式都能把指定的外部样式表应用到彩打上：

```
<link href="print-color.css" type="text/css"
   media="print and (color)" rel="stylesheet">

@import url(print-color.css) print and (color);
```

能使用媒体类型的地方都能使用媒体查询。继续以彩打为例，这意味着可以通过一个逗号分隔的列表列出多个查询：

```
<link href="print-color.css" type="text/css"
   media="print and (color), screen and (color-depth: 8)" rel="stylesheet">

@import url(print-color.css) print and (color), screen and (color-depth: 8);
```

只要其中一个媒体查询的条件得到满足，就会应用指定的样式表。对前面的 @import 声明来说，使用彩色打印机打印文档，或者在色深足够的环境中渲染文档时会使用 print-color.css。如果使用的是黑白打印机，两个查询的条件都不满足，因此 print-color.css 不会应用到文档上。在屏幕等媒体中渲染文档时也是如此。

一个媒体描述符包含一个媒体类型和一个或多个媒体特性列表，其中特性描述符要放在圆括号中。如果没有媒体类型，那就应用到所有媒体上，因此下面两个示例是等效的：

```
@media all and (min-resolution: 96dpi) {...}

@media (min-resolution: 96dpi) {...}
```

一般情况下，媒体特性描述符的格式类似于 CSS 中的一对属性和值。二者之间最大的区别是，特性描述符可以不指定值。因此，任何彩色媒体都符合 (color) 指定的条件，任何色深为 16 位的彩色媒体都符合 (color: 16) 指定的条件。其实，不指定值时是在做判断。比如说，(color) 的意思是“这个媒体是彩色的吗？”

多个特性描述符使用逻辑关键字 and 连接。媒体查询中可使用的逻辑关键字有两个：

and
: 连接的两个或多个媒体特性必须同时满足条件，整个查询得到的结果才是真值。例如，(color) and (orientation: landscape) and (min-device-width: 800px) 表示三个条件都要满足：当媒体环境是彩色的、横向放着，而且设备的屏幕宽至少为 800 像素，才会应用样式表。

not
: 对整个查询取反。假如所有条件都为真，那样式表不会应用到文档上。例如，not (color) and (orientation: landscape) and (min-device-width: 800px) 表示三个条目都满足时，整个语句得到的结果与之相反。因此，当媒体环境是彩色的、横向放着，而且设备的屏幕宽至少为 800 像素，样式表不会应用到文档上。除此之外的情况下，都将应用样式表。

注意，not 关键字只能在媒体查询的开头使用。写为这样是无效的：(color) and not (min-device-width: 800px)。如果真这样写，媒体查询将被忽略。还要注意，太旧的浏览器不支持媒体查询，因此会跳过媒体描述符以 not 开头的样式表。

媒体设备不支持 OR 关键字。不过，分隔多个媒体查询的逗号相当于 OR。例如，screen, print 的意思是，“为屏幕或印刷媒体时应用样式”；而 screen and (max-color: 2) or (monochrome) 是无效的，会被忽略，正确的写法是 screen and (max-color: 2), screen and (monochrome)。

此外还有一个 only 关键词，专门用于保证向后兼容（是的，这是真的）。

only
: 在不支持媒体查询的旧浏览器中隐藏样式表。例如，如果想在所有媒体中应用一个样式表，但是只在支持媒体查询的浏览器中应用，可以这样写：@import url(new.css) only all。在支持媒体查询的浏览器中，only 关键字被忽略，样式表会应用到文档上。而在不支持媒体查询的浏览器中，媒体类型为 only all，而这是无效的，因此不会应用样式表。注意，only 关键字只能用在媒体查询的开头。

1.5.5 媒体特性描述符和值的类型

目前，我们在示例中见过一些媒体特性描述符了，但这不是全部。下面列出所有可用的描述符（截至 2017 年年末）：

- width
- min-width
- max-width
- device-width
- min-device-width
- max-device-width
- height
- min-height
- max-height
- device-height
- min-device-height
- max-device-height
- aspect-ratio
- min-aspect-ratio
- max-aspect-ratio
- device-aspect-ratio
- min-device-aspect-ratio
- max-device-aspect-ratio
- color
- min-color
- max-color
- color-index
- min-color-index
- max-color-index
- monochrome
- min-monochrome
- max-monochrome
- resolution
- min-resolution
- max-resolution
- orientation
- scan
- grid

此外，还有两种新增的值：

- *<ratio>*
- *<resolution>*

描述符和值的完整说明及用法参见第 20 章。

1.6 特性查询

2015~2016 年间，CSS 新增了一个功能：根据用户代理是否支持特定的 CSS 属性及其值来应用一段样式。这个功能称为特性查询（feature query）。

特性查询在结构上与媒体查询很像。假设我们想在用户代理支持 color 属性时（显然是支持的）为元素设定颜色，可以这样写：

```
@supports (color: black) {
    body {color: black;}
```

```
    h1 {color: purple;}
    h2 {color: navy;}
}
```

上述代码的意思其实是，“如果你能识别并处理 color: black 这样的属性和值组合，那就应用这段样式；否则，跳过这段样式。”如果用户代理不支持 @supports，整段样式都会跳过。

特性查询是渐进增强样式的完美方式。比如说你想在浮动布局之外增加栅格布局，可以保留现有的布局方式，在样式表中添加下面这段样式：

```
@supports (display: grid ) {
    section#main {display: grid;}
    /* 去掉旧布局的样式 */
    /* 栅格布局的样式 */
}
```

这段样式在支持栅格布局的浏览器中应用，它会覆盖旧的页面布局，然后应用通过栅格实现的新布局。不支持栅格布局的旧浏览器很可能也不支持 @supports，因此会跳过整段样式，就像没出现过一样。

特性查询可以嵌套，其实还可以嵌套在媒体查询中，而且反过来嵌套也可以。若想使用弹性盒布局编写针对屏幕和印刷媒体的样式，可以把媒体查询块放在 @supports (display: flex) 块里：

```
@supports (display: flex) {
    @media screen {
        /* 针对屏幕媒体的弹性盒样式 */
    }
    @media print {
        /* 针对印刷媒体的弹性盒样式 */
    }
}
```

反过来，也可以在实现响应式设计的媒体查询块中添加 @supports() 块：

```
@media screen and (max-width: 30em){
    @supports (display: flex) {
        /* 针对小屏的弹性盒样式 */
    }
}
@media screen and (min-width: 30em) {
    @supports (display: flex) {
        /* 针对大屏的弹性盒样式 */
    }
}
```

具体如何组织完全由你自己决定。

与媒体查询一样，特性查询也支持使用逻辑运算符。假如想在用户代理同时支持栅格布局和 CSS 形状时应用一段样式，可以这样写：

```
@supports (display: grid) and (shape-outside: circle()) {
    /* 栅格和形状样式 */
}
```

这与下述写法是等效的：

```
@supports (display: grid) {
    @supports (shape-outside: circle()) {
        /* 栅格和形状样式 */
    }
}
```

除了 and 之外，还有其他的运算符可用。CSS 形状（详情参见第 10 章）体现了 or 的用处，因为很长一段时间以来，WebKit 只支持通过带厂商前缀的属性绘制形状。因此，如果想绘制形状，可以使用这样的特性查询：

```
@supports (shape-outside: circle()) or
          (-webkit-shape-outside: circle()) {
        /* 绘制形状的样式 */
}
```

带厂商前缀的形状属性和不带厂商前缀的形状属性最好同时使用，但是上例这样做既能兼容 WebKit 以后的版本，也能支持其他无需使用厂商前缀的浏览器。

有时，我们想使用的属性与测试的属性不同。还以栅格布局为例，你可能想在支持栅格时修改布局元素的外边距等。下面是简化的版本：

```
div#main {overflow: hidden;}
div.column {float: left; margin-right: 1em;}
div.column:last-child {margin-right: 0;}

@supports (display: grid) {
    div#main {display: grid; grid-gap: 1em 0;
            overflow: visible;}
    div#main div.column {margin: 0;}
}
```

此外，还可以使用取反运算符。例如，下述样式在不支持栅格布局时应用：

```
@supports not (display: grid) {
    /* 不支持栅格时使用的样式 */
}
```

一个特性查询中可以使用多个逻辑运算符，但是为了保证条理清晰，要使用括号。假如我们想在支持颜色的同时还支持栅格或弹性盒布局中的一个时应用一段样式，可以这样写：

```
@supports (color: black) and ((display: flex) or (display: grid)) {
        /* 相关的样式 */
}
```

注意，判断支持栅格或弹性盒的查询放在一对括号里。必须这么做，如若不然，整个表达式都将失效，而块中的样式则会跳过。也就是说，不要写成这样：

```
@supports (color: black) and (display: flex) or (display: grid) {
```

最后，你可能想知道，为什么特性查询的测试中既要写属性也要写值。毕竟，想使用形状时，我们只需要测试支不支持 `shape-outside`，对吧？这是因为浏览器可能支持某个属性，但不支持它的全部取值。栅格布局就是个很好的例子。如果像下面这样测试是否支持栅格肯定是不行的：

```
@supports (display) {
    /* 栅格样式 */
}
```

因为连 Internet Explorer 4 都支持 `display`。支持 `@supports` 的浏览器肯定支持 display 及其很多取值，但不一定支持 `grid` 这个值。鉴于此，特性查询既要测试属性也要测试值。



记住，这是特性查询，不是正确性查询。浏览器可能支持所测试的特性，但实现上有缺陷。所以，无法确保浏览器对特性的支持是否正确。特性查询得到肯定结果的意思很简单，表示浏览器支持你想使用的特性，而且以某种方式实现了。

1.7 小结

CSS 可以完全改变元素在用户代理中的表现方式。简单一点的，可能只使用 display 属性，复杂一点的则可以为文档关联样式表。用户绝不会知道你使用的是外部样式表或嵌入式样式表，抑或是行内样式。外部样式表的主要优势是，可以把一个网站的所有表现信息

放在一个地方，然后在所有文档中指向这个位置。这样不仅便于更新和维护网站，还能节省带宽，因为所有表现都游离在文档之外。借助 `@supports()` 甚至可以使用纯 CSS 实现基本的渐进增强功能。

为了发挥 CSS 的作用，创作人员要知道如何把样式关联到文档中的元素上。若想完全理解 CSS 是如何做到这一点的，创作人员要充分掌握 CSS 选择文档中不同内容的方式。这是下一章的话题。

第 2 章

选择符

CSS 的主要优势之一是能轻易为同类型的所有元素应用一组样式。是不是听上去没有想象的那么震撼？那么请这样想想看：编辑一行 CSS 就能改变所有标题的颜色。不喜欢现在使用的蓝色标题？那就修改那行代码，改成紫色、黄色、红褐色，抑或想使用的其他颜色。如此一来，设计师便能集中精力在设计上，而不必为琐事烦忧。下次开会时，如果有人还想要绿色阴影，只需编辑样式，再单击刷新按钮就行了。很酷吧！只需几秒便能得到想要的效果，而且每个人都能看得到。

CSS 不能解决所有问题，例如，CSS（至少现在）不能修改 PNG 图像的色彩空间。但是使用 CSS 的确能轻易做些全局性修改。下面先从选择符和结构学起。

2.1 样式的基本规则

前面说过，CSS 的一个核心优势是可以为文档中某种类型的元素全部应用相同的规则。假如我们想让所有 h2 元素都显示为灰色。以前，我们只能编辑 HTML，在每个 h2 元素中插入 … 标签。就算使用 style 属性，也省不了多少时间，你要为每个 h2 元素设定 style="color: gray;" 属性。这两种方式如下所示：

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

如果文档中有大量 h2 元素，这将是一个漫长乏味的过程。更糟的是，如果后来决定 h2 元素应该显示成绿色，而不是灰色，就必须重来一次，手动修改所有标签（是的，以前就是这么做的）。

CSS 样式便于修改和编辑，而且能应用到指定的所有文本元素上（下一节说明如何指定）。例如，可以编写如下的规则把所有 h2 元素的颜色设为灰色：

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

如果想修改 h2 元素的颜色，比如改为银色，只需修改颜色值：

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

2.1.1 元素选择符

元素选择符（element selector）通常是 HTML 元素，但也有例外。例如，在针对 XML 文档的 CSS 文件中，可能有下面这些元素选择符：

```
quote {color: gray;}
bib {color: red;}
booktitle {color: purple;}
myElement {color: red;}
```

也就是说，文档中的元素是最基本的选择符。在 XML 中，什么都可以作为选择符，因为 XML 允许创建新标记语言，所以什么都可以作为元素名称。然而，为 HTML 文档编写样式时，选择符一般是 HTML 预定义的某个元素，例如 p、h3、em、a，甚至是 html 自身。例如：

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

这个样式表得到的结果如图 2-1 所示。

直接为元素定义全局样式后，可以把样式从一个元素身上移到另一个元素身上。假如我们想让图 2-1 中的段落显示为灰色，而不是 h1 元素，没问题，只需把 h1 选择符改为 p：

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

得到的结果如图 2-2 所示。

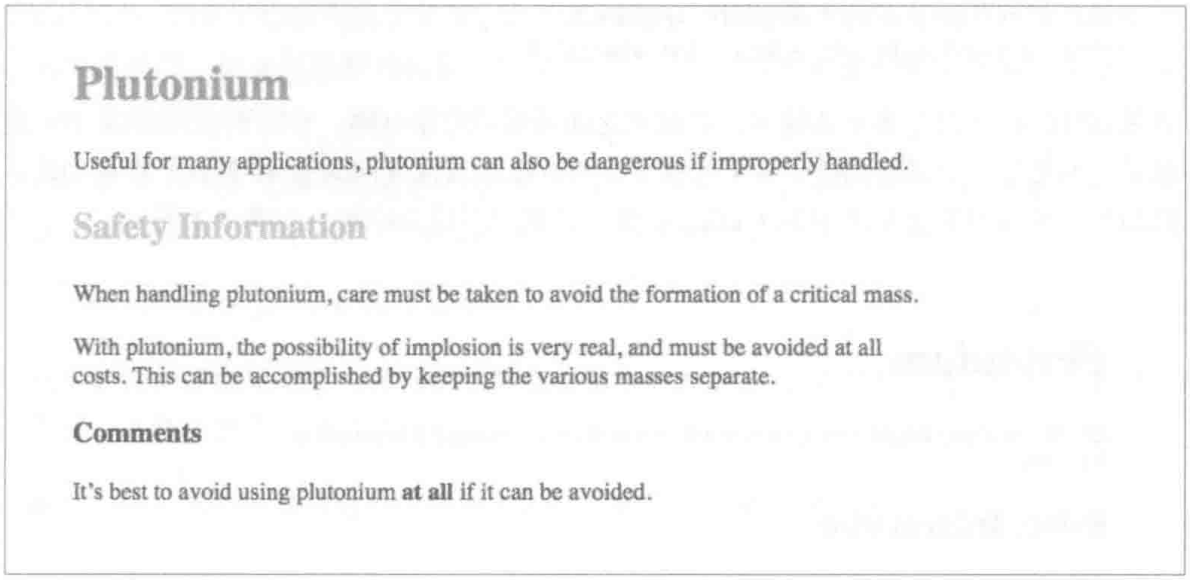


图 2-1：一个简单文档的简单样式

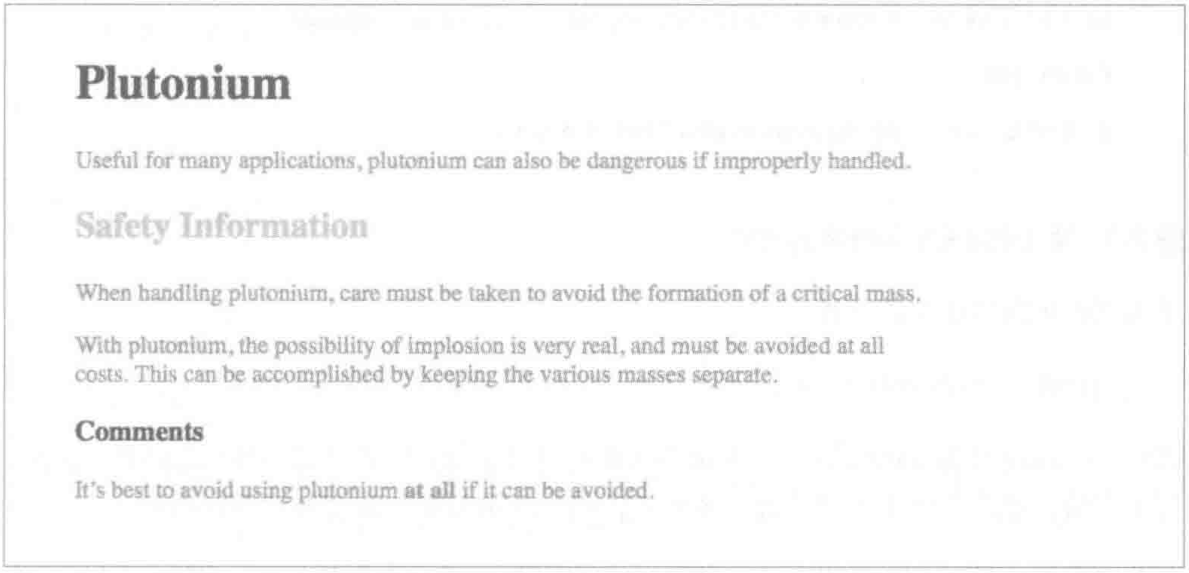


图 2-2：把一个元素的样式应用到另一个元素上

2.1.2 声明和关键字

声明块中有一个或多个声明。声明的格式是固定的，先是属性，然后是冒号，后面再跟上值和分号。冒号和分号后面可以有零个或多个空白。值几乎都是一个关键字或以空格分隔的多个关键字。如果声明中的属性或值有误，整个规则都将被忽略。因此，下面两个声明是无效的：

```
brain-size: 2cm;      /* 未知属性 "brain-size" */
color: ultraviolet;   /* 未知值 "ultraviolet" */
```

如果属性的值可以是多个关键字，关键字之间通常以空格分隔，有时则使用斜线（/）或逗号。不是每一个属性都接受多个关键字，不过很多属性（例如 font 属性）都是如此。假如想让段落中的文本使用 Helvetica 字体，并把字号设为中等，如图 2-3 所示。

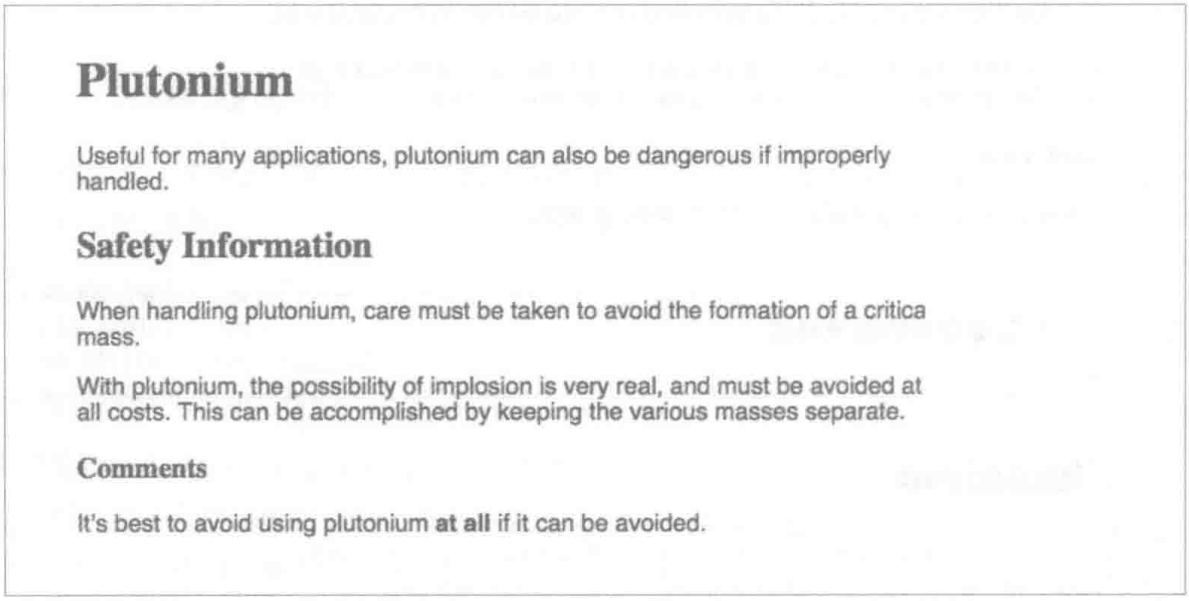


图 2-3：属性值含多个关键字的结果

那么规则可以写成下面这样：

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

注意，medium 和 Helvetica 两个关键字（前者设定字号，后者设定字体）之间有个空格。有了空格，用户代理才能把关键字区分开，然后分别应用。分号表示声明结束。

这些以空格分开的词称为关键字，因为它们在一起构成一个属性的值。例如，对下述虚构的规则来说：

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

我知道没有 rainbow 属性，这里只是以此为例。rainbow 的值是 red orange yellow green blue indigo violet，这七个关键字在一起构成一个完整而唯一的值。我们还可以像下面这样定义 rainbow 的值：

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

rainbow 的新值现在有九个关键字，而不再是七个。这两个值看起来差不多，但二者是完全不同的。这看起来有点抽象，但是对于理解特指度和层叠的细节（参见本书后文）是至关重要的。

可以看出，CSS 关键字通常以空格分隔。在 CSS2.1 中，有个特例：font 属性的值有一处要使用斜线（/）把两个关键字隔开。下面举个例子：

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

这个斜线把设定元素字号和行高的两个关键字分开。font 声明只有这一个地方允许出现斜线，其他关键字都以空格分隔。

除此之外，其他一些属性的值中也可以使用斜线，比如说（不完整）：

- background
- border-image
- border-radius
- grid
- grid-area
- grid-column
- grid-row
- grid-template
- mask-border

还有一些关键字以逗号分隔。声明多个背景图像、过渡属性和阴影时，相互之间以逗号分隔。此外，函数的参数，例如线性渐变和变形函数，以逗号分隔，如下例所示：

```
.box {box-shadow: inset -1px -1px white,
                  3px 3px 3px rgba(0,0,0,0.2);
      background-image: url(myimage.png),
          linear-gradient(180deg, #FFF 0%, #000 100%);
      transform: translate(100px, 200px);
}
a:hover {transition: color, background-color 200ms ease-in 50ms;}
```

以上是简单声明的基本知识，不过声明可以变得十分复杂。下一节开始展现 CSS 的强大功能。

2.2 群组

目前，我们学习了如何把一个样式应用到一个选择符上，这是相当简单的技术。如果想把同一个样式应用到多个元素上该怎么做呢？此时，可以使用多个选择符，或者把多个样式应用到一个元素或一组元素上。

2.2.1 群组选择符

假如想让 h2 元素和段落中的文本都显示为灰色，最简单的方法是使用下述声明：

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

规则的左边是 h2 和 p 选择符，之间以逗号分隔；右边的规则（color: gray;）应用到前面两个选择符引用的元素上。逗号告诉浏览器，这个规则涉及两个不同的选择符。如果没有逗号，规则的含义完全不同，详情参见 2.5.2 节。

可以放在一起的选择符数量没有限制。例如，如果想让很多元素显示为灰色，可以使用类似下面的规则：

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

群组可以在很大程度上压缩同类样式，这样得到的样式表更短。下面两种写法得到的结果完全一样，但是哪一种写起来更轻松一目了然：

```
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;}
```

群组有不同的分发。例如，下面几个群组规则是等效的，只是选择符和声明的分组方式不同：

```
/* 分组 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;}
```

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

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

这几种分组方式得到的结果是一样的，如图 2-4 所示（这些样式用到了群组声明，详情参见 2.2.2 节）。

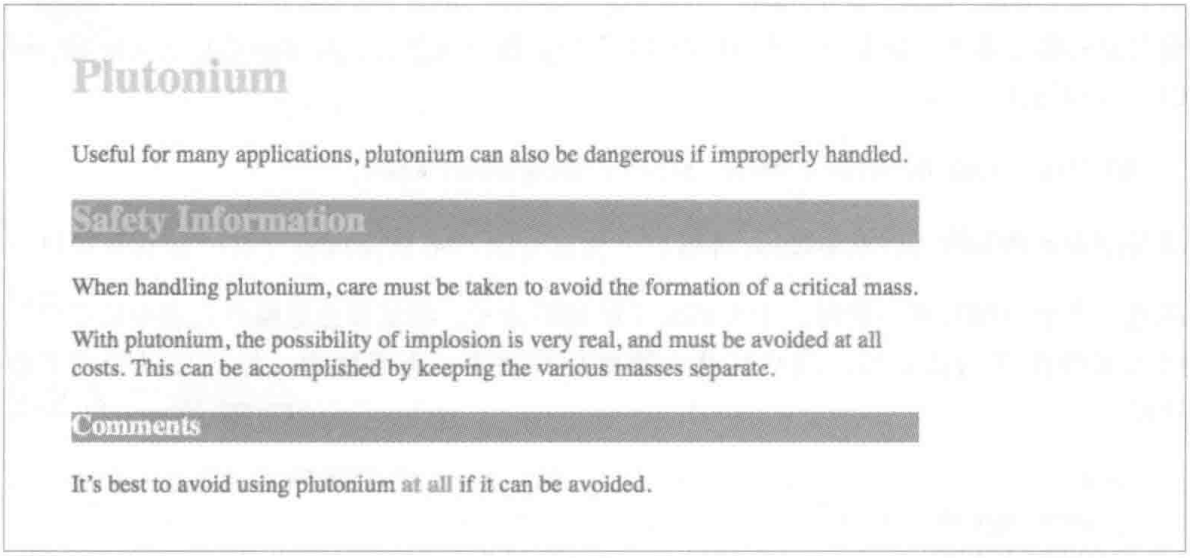


图 2-4：不同分组方式得到的一致结果

通用选择符

CSS2 引入了一个新的选择符，名为通用选择符（universal selector），写作星号（*）。这个选择符匹配所有元素，很像是通配符。假如想让文档中的每个元素都显示为红色，可以这样写：

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

这个声明与列出文档中每一个元素的群组选择符是等效的。借助通用选择符，只需敲一次按键就能把每个元素的颜色都设为红色。然而要注意，通用选择符虽然便利，但是它的特指度是 0-0-0；而且由于目标是一切元素，可能出现一些意想不到的后果（本书后文讨论）。

2.2.2 群组声明

一个规则可以有多个选择符，同理，也可以有多个声明。如果想让所有 h1 元素的颜色为紫色，字体为 Helvetica，字号为 18 像素，且背景为浅绿色（不介意伤害读者的眼睛），可以像下面这样编写样式：

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

但是这样做效率低，试想一下有 10 或 15 个样式都要这样写该有多麻烦！其实，我们可以把声明写在一起：

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

这与前面分开写的三行效果完全一样。

注意，多个声明写在一起时，千万别忘了结尾的分号。浏览器会忽略样式表中的空白，因此必须保证句法正确，用户代理才能解析样式表。无需顾虑，样式可以写成下面这样：

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

此外，还可以简化 CSS，去掉所有不必须的空格：

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

对服务器来说，除了空白使用上的不同，前面三个样式的作用是一样的，不过第二个样式对人类来说的可读性是最高的，也是开发过程中建议采用的方式（为了提升网络性能，可以选择简化 CSS，不过这通常由服务器端脚本、缓存网络或其他服务处理）。

如果第二个声明结尾没有分号，用户代理将把样式表解释为：

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

因为 background: 不是 color 的有效值，而且 color 的值只能是一个关键字，所以用户代理将忽略整个 color 声明（包括 background: aqua 部分）。别以为浏览器会把 h1 渲染成紫色，但没浅绿色背景。最终得到的是默认颜色（通常为黑色），背景为透明（这也是默认值）。font: 18px Helvetica 声明将起作用，因为它的结尾有分号。



严格来说，一个规则中的最后一个声明的结尾可以不带分号，但一般建议带上。首先，这样做能让你养成带分号的习惯，毕竟缺少分号是导致渲染错误最常见的根源。其次，如果想在后面加一个声明，不用担心忘记补上分号。最后，如果使用 Sass 这样的预处理器，所有声明通常都要求在结尾加上分号。为了避免种种问题，编写规则时一定要加上分号。

与群组选择符一样，把声明写在一起也有助于保持样式表短小、意图明确，而且易于维护。

2.2.3 二者结合

我们知道，选择符和声明都可以写在一起。二者结合，只需几个声明便能定义十分复杂的样式。试想，如果想为所有标题赋予一些复杂的样式，而且想让全部标题都使用相同的样式，该怎么做呢？可以这样做：

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

这里，我们使用的是群组选择符，因此规则右侧的样式将应用于列出的所有标题；此外，声明也是多个写在一起的，所以列出的样式将应用于左侧的所有选择符上。这个规则得到的结果如图 2-5 所示。

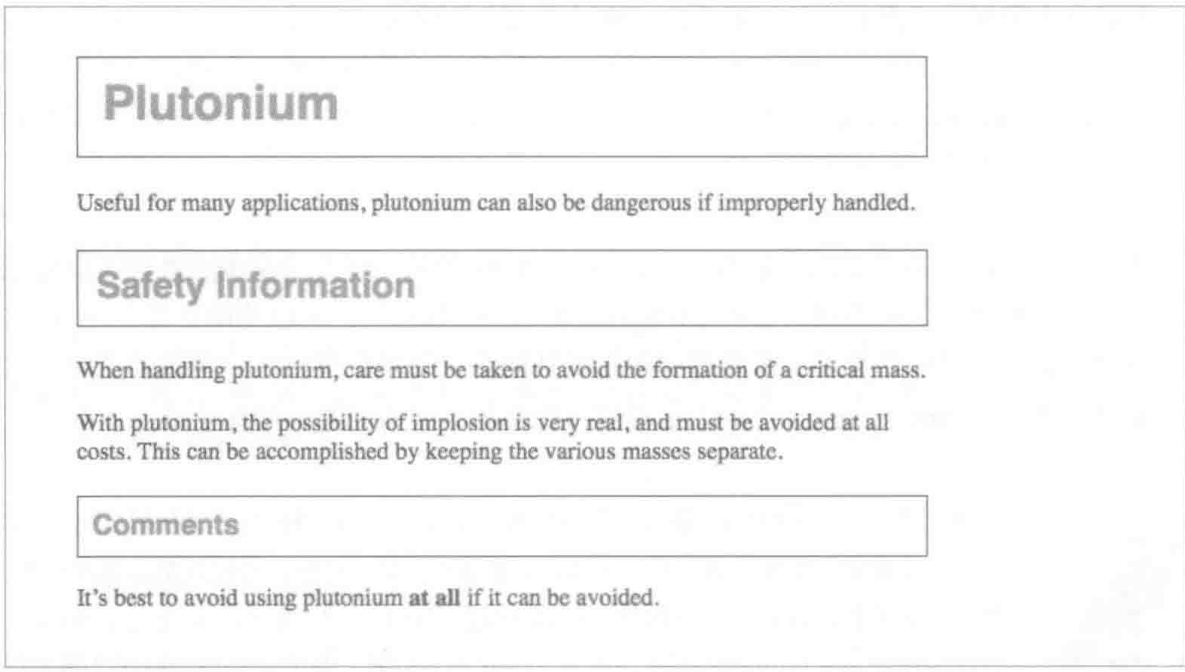


图 2-5：同时使用群组选择符和群组声明

这样写比下面这一长串好得多。如果分开写，开头可能是这样的：

```
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;}
```

而且要持续很多行。你可以选择编写这一长串样式，但是不建议这么做，否则跟直接使用 style 属性没多大差别。

选择符中除了同类元素之外，还可以有其他元素。为了得到更好的效果，要付出一点精力，但这么做是值得的。

2.2.4 在旧浏览器中使用新元素

HTML（例如 HTML5 规范）在更新的过程中加入了一些新元素。有些浏览器在这些新元素出现之前就已存在，因此可能无法识别新元素。例如，在 Internet Explorer 9 之前版本中无法选择不支持的元素。解决方法是在 DOM 中创建元素，让浏览器知道元素的存在。

例如，Internet Explorer 8 不识别 <main> 元素。使用下述 JavaScript 可以让 Internet Explorer 8 知道 main 的存在：

```
document.createElement('main');
```

运行这段代码后，旧版 Internet Explorer 将认识到元素的存在，从而允许选择并为之赋予样式。

2.3 类选择符和 ID 选择符

目前，我们以不同的方式组合了选择符和声明，不过我们使用的选择符十分简单，只是引用文档中的元素。从某种意义上讲，元素选择符很不错，但是有时我们想指代得更明确一些。

除了直接使用文档中的元素之外，还可以使用类选择符和 ID 选择符，这样便能以一种独立于元素的方式赋予样式。这两种选择符可以独立使用，也可以和元素选择符结合在一起使用。然而，这两种选择符要求文档的标记有一定的结构，因此一般要提前构思，做好规划。

假设你在撰写一篇探讨钚处理方式的文章，文中有一些提醒，着重指出如何安全处理这个危险物质。你想让这些提醒文字显示为粗体，吸引读者注意。然而，你不知道哪些地方会出现提醒。有些提醒可能是整段文字，还有些是一个长列表中的一个项目或一段文字中的一部分。因此，无法使用元素选择符定义这样的规则。如果这样编写样式：

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

那所有段落都将变成红色粗体字，而不只是那些包含提醒的段落。你需要一种只选择提醒文字的方式，或者更准确地说，一种只选择提醒元素的方式。那么，该怎么做呢？你需要的是类选择符，借助它为文档中已经通过某种方式标记的部分赋予样式，而不考虑具体涉及哪些元素。

2.3.1 类选择符

应用样式而不关心所涉及的元素，最常使用类选择符。然后，在使用之前，要修改文档的标记，让类选择符起作用。修改方式是设定 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>
```

为了把类选择符定义的样式应用到元素上，必须为 `class` 属性赋予适当的值。在前面的示例中，我们把两个元素的 `class` 属性设为 `warning`：第一个段落和第二段中的 span 元素。

现在我们要找到一种方法为设定了 `class` 属性的元素赋予样式。在 CSS 中，选择类的句法是在 `class` 属性的值前面加上点号（.）。除了单独选择类之外，还可以结合元素选择符。

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

这个简单的规则应用到前面的标记上得到的结果如图 2-6 所示。即，`font-weight: bold` 声明将应用到 `class` 属性的值为 `warning` 的每个元素上（因为这里蕴含着通用选择符）。



使用 ID 选择符、类选择符、属性选择符、伪类选择符或伪元素选择符时，如果没有依附元素选择符，隐式蕴含通用选择符（使用 * 符号表示）。

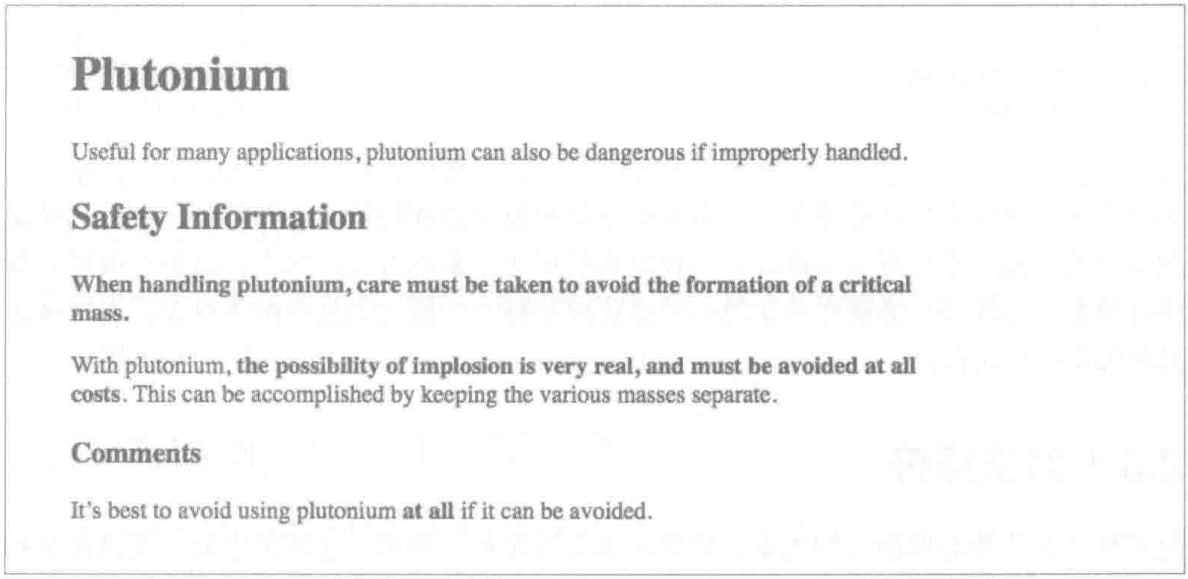


图 2-6：使用类选择符

可以看出，类选择符直接引用元素的 class 属性值，而且前面始终有个点号（.），表明这是类选择符。这个点号把类选择符与可能一起使用的其他选择符（例如元素选择符）分开。例如，你可能想让整段文字都为提醒时才加粗：

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

这个选择符现在匹配 class 属性的值中包含 warning 的 p 元素，其他任何元素，不管有没有这个类，都不匹配。因为第二段中的 span 元素不是段落，所以与这个规则的选择符不匹配，从而不会显示为粗体。

如果想为那个 span 元素赋予不同的样式，可以使用 span.warning 选择符：

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

此时，提醒段落显示为粗体，而起提醒作用的 span 元素显示为斜体。每个规则只应用于一个元素和类的组合，相互之间没有影响。

此外，还可以先使用通用的类选择符定义一个样式，然后再结合元素选择符，进一步指明适用的元素，如下所示：

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

结果如图 2-7 所示。

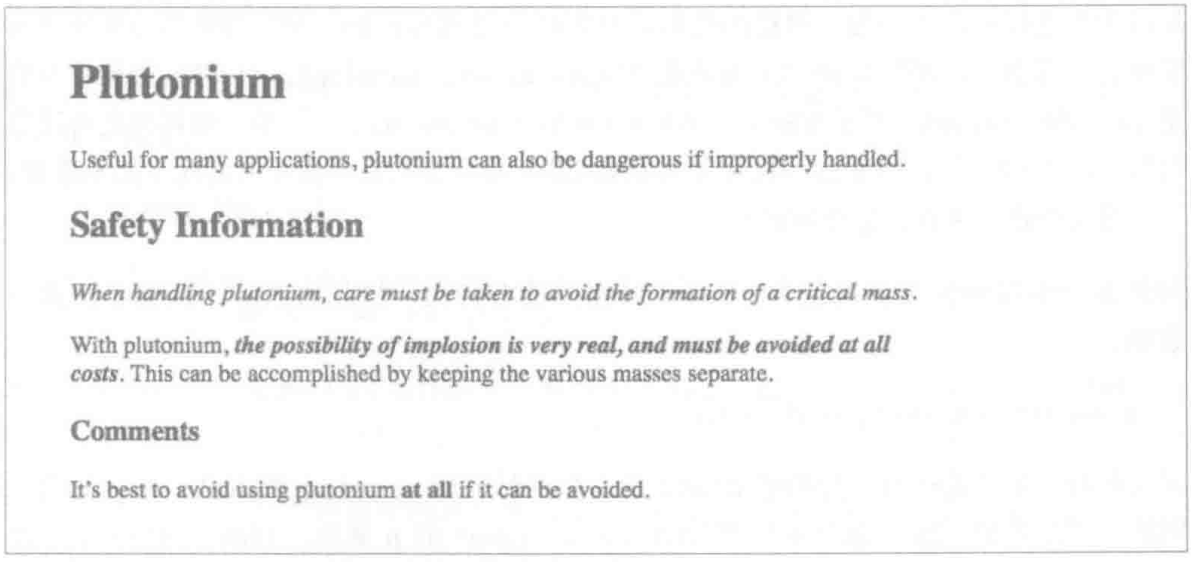


图 2-7：结合通用和特指的选择符定义样式

此时，所有提醒文字都将显示为斜体，但只有 class 属性的值为 warning 的 span 元素才会显示为粗斜体。

注意前例中通用类选择符的格式：类名前有个点号，但没有元素名，也没有通用选择符。如果想选择具有相同类名的所有元素，可以放心省略通用选择符。

2.3.2 多个类

前一节中的 class 属性只有一个词。根据 HTML 规范，class 属性的值可以是多个词，词之间使用空格分隔。比如你想把某个元素标记为特别重要的提醒，可以这么写：

```
<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>
```

词的顺序无关紧要，写成 warning urgent 也行，结果是完全一样的。

假如你想让 class 属性的值为 warning 的元素显示为粗体，值为 urgent 的元素显示为斜体，而同时拥有二者的元素具有银色背景，样式可以这样写：

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

把两个类选择符串在一起，选择的是同时具有两个类名的元素，而且对类名的顺序没有要求。可以看出，HTML 源码中写的是 class="urgent warning"，而 CSS 选择符写的是 .warning.urgent。尽管如此，“When handling plutonium . . . ”那一段仍将具有银色背景，如图 2-8 所示。这是因为顺序无关紧要（并不是说类的顺序在任何情况下都不重要，本书后文会进一步探讨这个问题）。

如果多个串联的类选择符包含 class 属性中没有的词，将无法匹配。试看下面这个规则：

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

不出所料，这个选择符匹配的是 class 属性中同时包含 warning 和 help 的 p 元素。因此，无法匹配 class 属性中只有 warning 或 urgent 的 p 属性。然而，却能匹配下述元素：

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

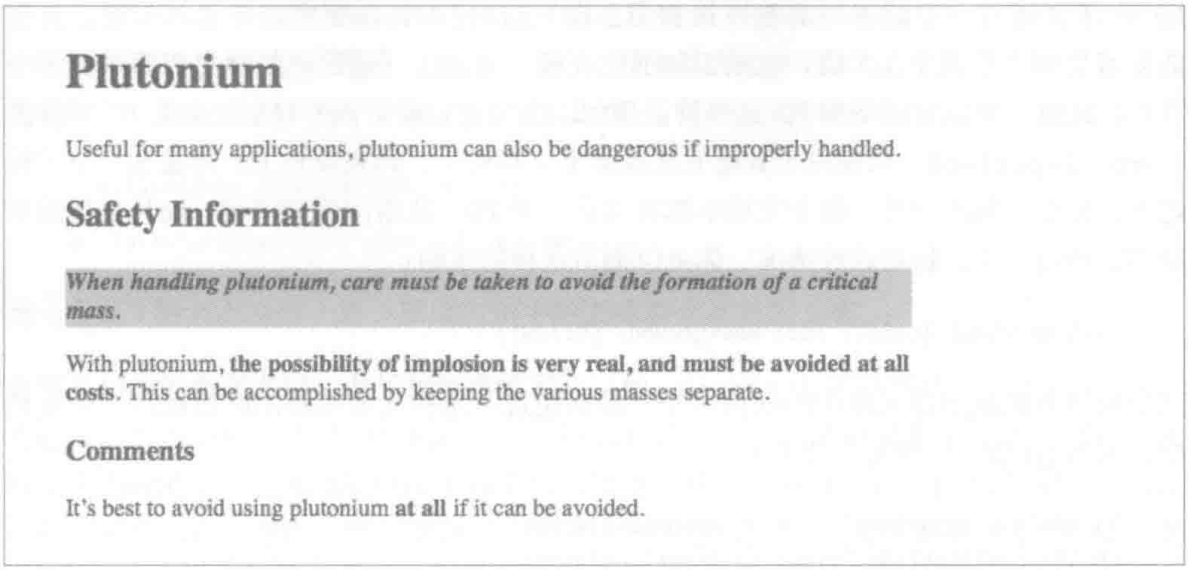


图 2-8：使用多个类名选择元素

2.3.3 ID 选择符

ID 选择符在某些方面类似于类选择符，不过二者之间有些重要区别。首先，ID 选择符的开头不是点号，而是个散列字元（#），也叫井号、哈希符号、哈希记号或三连棋棋盘。因此，你可能见过类似下面的规则：

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

这个规则把 id 属性的值为 first-para 的元素中的文本设为粗体。

第二个区别是，ID 选择符引用的不是 class 属性的值，而是 id 属性的值（显而易见）。下面举个例子：

```
*#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，文档中任何元素的 id 属性的值都有可能是 lead-para。这里，我们把第一段的 id 设为 lead-para，不过也可以在第二段或第三段，又或一个无序列表等元素上这么做。

与类选择符一样，ID 选择符中的通用选择符也可以省略。前面的示例还可以写成这样：

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

这样写与之前的效果一样。

ID 选择符还有一个地方与类选择符相似：ID 选择符不关心所用的元素。有时，你可能知道文档中有某个 ID 值，但是不知道它在哪个元素上（就像钚处理提醒那个例子一样）。此时，可以直接使用 ID 选择符。例如，你可能知道文档中有个元素的 ID 属性值为 mostImportant，但是你不知道它在段落上、短语上、列表项目上，还是某一节的标题上。你唯一知道的是，每个文档中都有这么一个 ID，依附的元素不定，而且出现的次数不会超过一个。针对这种情况，你可以编写这样的规则：

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

这个规则将匹配下述元素中的任何一个（前面说过，这些元素不能同时出现在一个文档中，因为它们的 ID 值是一样的）：

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

2.3.4 在类选择符和 ID 选择符之间选择

如前所示，类可以赋予任意个元素，warning 这个类名赋予了 p 元素和 span 元素，此外还可以赋予更多的元素。而 ID 就不同了，在一个 HTML 文档中，一个 ID 能且只能使用一次。因此，如果文档中有个元素的 id 属性值为 lead-para，其他元素的 id 属性就不能再设为这个值。



实际上，浏览器不一定总会检查 HTML 中的 ID 是不是唯一的。也就是说，如果 HTML 文档中的多个元素具有相同的 ID 属性，相同的样式可能会应用到每个元素上。这是不正确的行为，但却可能发生。文档中出现多个相同的 ID 值还不利于 DOM 脚本编程，因为 getElementById() 等函数预期只有一个 ID 属性为指定值的元素。

与类选择符不同，ID 选择符不能串在一起使用，因为 ID 属性的值不能是以空格分隔的列表。

类选择符和 ID 选择符之间的另一个区别是，用户代理判断该把哪个样式应用到元素上时，ID 选择符的权重更高。下一章将详述这个话题。

此外还要注意，类选择符和 ID 选择符可能是区分大小写的，这取决于文档语言。根据 HTML 规范，类和 ID 的值是区分大小写的，因此类选择符和 ID 选择符的大小写必须与文档中的一致。因此，对于下述 CSS 和 HTML 而言，元素中的文本不会显示为粗体：

```
p.criticalInfo {font-weight: bold;}
<p class="criticalinfo">Don't look down.</p>
```

因为字母 i 的大小写不一样，所以上例中的选择符无法匹配元素。

单纯从句法的角度来说，点号加类名的写法（如 `.warning`）无法保证一定能用于 XML 文档。写作本书时，点号加类名的写法支持 HTML、SVG 和 MathML；此外，还可能支持未来出现的语言，不过这取决于各语言的规范。井号加 ID 的写法（如 `#lead`）支持任何能确保文档内元素唯一性的文档语言。唯一性可以通过 `id` 属性确保，也可以通过其他任何方式，只要属性的值在文档中是唯一的即可。

2.4 属性选择符

不管是类选择符还是 ID 选择符，我们选择的其实都是属性的值。前两节使用的句法专门针对 HTML、XHTML、SVG 和 MathML 文档（截至写作本书时）。在其他标记语言中，这样编写的类选择符和 ID 选择符可能无法使用（`class` 和 `id` 属性或许根本不存在）。为了解决这个问题，CSS2 引入了属性选择符（attribute selector），根据属性及其值选择元素。属性选择符大致可以分为四类：简单属性选择符、精准属性值选择符、部分匹配属性值选择符和起始值属性选择符。

2.4.1 简单属性选择符

如果想选择具有某个属性的元素，而不管属性的值是什么，可以使用简单属性选择符。例如，若想选择具有 `class` 属性（可以包含任何值）的所有 `h1` 元素，把文本设为银色，可以这样写：

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

对下面的标记来说：

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

得到的结果如图 2-9 所示。



图 2-9：根据属性选择元素

这种选择符在 XML 文档中特别有用，因为 XML 语言中的元素和属性都是根据目的而选用的。假设有个描述太阳系中各行星的 XML 文档（称之为 PlanetML），我们想选择具有 moons 属性的全部 pml-planet 元素，把文本设为粗体，突出显示有卫星的行星，可以这么写：

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

对下面的标记片段来说，这个规则会把第二个和第三个元素的文本变成粗体，而第一个元素不受影响：

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

在 HTML 文档中，你可以发挥自己的想象力，充分利用这一功能。例如，你可以为所有具有 alt 属性的图像编写样式，突出显示格式正确的图像：

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

（这个示例更多的是出于诊断目的，即判断图像的标记是否正确，在设计层面没有太大的意义。）

如果想把具有 title 属性的元素（在多数浏览器中，当鼠标悬停在这样的元素上时，会显示“提示框”）加粗显示，可以这么写：

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

类似地，可以为具有 href 属性的元素（a 元素）编写样式，只应用到超链接上，而不应用到锚记上。

此外，还可以基于多个属性选择。为此，要把多个属性选择符串在一起。例如，若想让同时具有 href 和 title 属性的 HTML 超链接显示为粗体，可以这样写：

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

对下述标记来说，第一个链接会显示为粗体，而第二个和第三个链接都不会显示为粗体：

```
<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>
```

2.4.2 根据精准的属性值选择

此外，还可以进一步缩小范围，只选择属性为特定值的元素。比如说我们想把指向 Web 服务器上某个文档的超链接显示为粗体，可以这么写：

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

这个规则把 href 属性的值为 *http://www.css-discuss.org/about.html* 的 a 元素显示为粗体。任何变化，即便没有 www. 部分，或者换成安全协议 https，都无法匹配。

可以为任何元素指定任何属性和值的组合。然而，如果属性和值的组合在文档中未出现，那么选择符不匹配任何元素。同样，XML 语言也很适合使用这种方式应用样式。还以前面的 PlanetML 为例。假设你只想选择 moons 属性的值为 1 的 planet 元素，可以这样写：

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

对下面的标记片段来说，这个样式将把第二个元素的文本设为粗体，而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>
```

与选择属性时一样，可以把多个属性和值选择符串在一起。例如，若想把 href 属性的值为 *http://www.w3.org/*，而且 title 属性的值为 W3C Home 的 HTML 超链接显示为两倍字号，可以这样写：

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

对下面的标记来说，这个样式将把第一个链接的文本字号加倍，而第二个和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 2-10 所示。



图 2-10：根据属性及其值选择元素

再次说明，这个形式要求属性的值与指定的值完全一致。属性的值是由空格分隔的多个词时（例如 HTML 中的 class 属性）要小心，以防匹配出错。对下面的标记片段来说：

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

若想完全匹配属性的值，只能像下面这样写：

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

如果写成 planet[type="barren"]，无法匹配标记，因此也就不起作用。对 HTML 中的 class 属性来说也是如此。对下面的标记来说：

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

若想使用精确的属性值选择这个元素，要写成：

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

这与前文所述的点号加类名写法不同，详情参见下一节。上述规则选择的是 class 属性的值与 “urgent warning” 完全一样的 p 元素，两个词的顺序一模一样，而且中间有个空格。这其实是精确匹配字符串。

此外要知道，ID 选择符与引用 id 属性的属性选择符不完全等效。也就是说，h1#page-title 和 h1[id="page-title"] 之间有些微妙而不容忽视的区别。详情参见 3.1 节。

2.4.3 根据部分属性值选择

有时，我们想根据属性值的一部分选择元素，而不是完整的值。CSS 为这种情况提供了多种选择，以不同的方式匹配属性值的子串。这些方式的概述见表 2-1。

表 2-1：使用属性选择符匹配子串

形式	说明
[foo|="bar"]	选择的元素有 foo 属性，且其值以 bar 和一个英文破折号（U+002D）开头，或者值就是 bar 本身
[foo~="bar"]	选择的元素有 foo 属性，且其值是包含 bar 这个词的一组词
[foo*="bar"]	选择的元素有 foo 属性，且其值包含子串 bar
[foo^="bar"]	选择的元素有 foo 属性，且其值以 bar 开头
[foo$="bar"]	选择的元素有 foo 属性，且其值以 bar 结尾

一种特别的属性选择符

这些属性选择符中的第一个匹配属性值的一部分，描述起来困难，举个例子就明白了。以下述规则为例：

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

这个规则选择 lang 属性的值为 en 或者以 en- 开头的元素。因此，对下述示例标记来说，前三个元素会被选中，而后两个不会：

```
<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>
```

一般来说，[att|="val"] 形式可用于选择任何属性及其值。假设一个 HTML 文档中有一系列插图，而插图的文件名是 *figure-1.gif* 和 *figure-3.jpg* 这样的。使用下述选择符可以匹配所有插图：

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

此外，如果你在开发一个 CSS 框架或模式库，没必要提供 “btn btn-small btn-arrow btn-active” 这样冗长的类，可以声明 “btn-small-arrow-active”，然后使用下述规则选择具有这个类的元素：

```
*[class|="btn"] { border-radius: 5px;}

<button class="btn-small-arrow-active">Click Me</button>
```

这种属性选择符最常用于匹配语言名。详情参见 2.6.6 节。

匹配以空格分隔的一组词中的一个

对值可以为一组以空格分隔的属性来说，可以根据其中有没有某个词来选择元素。HTML 中的 class 属性就是这样，它的值可以是一个或多个词。还以前面的钚处理文档为例：

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

若想选择 class 属性的值包含 warning 的元素，可以使用下述属性选择符：

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

注意，选择符中有个波浪号（~）。根据一组以空格分隔的词中是否有某个词选择时，不能忘了这个符号。如果没有波浪号，就变成前一节讨论的精确值匹配属性选择符了。

这个选择符的作用与 2.3.4 节讨论的点号加类名写法一样。也就是说，对 HTML 文档来说，p.warning 和 p[class~="warning"] 是等效的。下面是与前面的 PlanetML 标记等效的 HTML 版本：

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

若想倾斜 class 属性中包含 barren 这个词的所有元素，可以这样写：

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

这个规则的选择符匹配示例标记中的前两个元素，把它们的文本显示为斜体，如图 2-11 所示。这与 span.barren {font-style: italic;} 得到的结果一样。



图 2-11：根据部分属性值选择元素

那么为什么还要有这样一种属性选择符呢？因为除了 class 属性之外还可以根据其他属性选择。例如，一个文档中可能有多个图像，但只有部分是插图。此时，可以在 title 属性上使用匹配部分值的选择符，只选择插图：

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

这个规则的选择符选择 title 属性的值中包含 Figure 这个词的图像。因此，只要插图的 title 属性是“Figure 4. A bald-headed elder statesman”这样的文本，就能被这个规则匹配。其实，img[title~="Figure"] 选择符也会匹配 title 属性的值为“How to Figure Out Who's in Charge”的图像。但是，没有 title 属性的图像，或者 title 属性的值中没有“Figure”这个词的图像不匹配。

匹配属性值的子串

有时，我们想根据部分属性值选择元素，但是属性的值不是以空格分隔的一组词。此时，可以使用 [att*="val"] 形式匹配出现在属性值内部任何位置的子串。例如，下述 CSS 匹配 class 属性的值中包含子串 cloud 的 span 元素，所以 class 属性的值中有“cloudy”的那两个元素都匹配，如图 2-12 所示。

```
span[class*="cloud"] {font-style: italic;}

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



图 2-12：根据属性值中的子串选择元素

可以想象，这种选择符有很大的用处。假设你想特别显示指向 O'Reilly 网站的链接。此时无需为每个这样的链接设定 class 属性，然后再使用类选择符编写样式，而是可以像下面这样编写规则：

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

这种选择符并不局限于 class 和 href 属性，任何属性都可以，例如 title、alt、src、id... 只要属性有值，就可以根据值的子串应用样式。下述规则突出显示源 URL 中包含“space”这个字符串的图像：

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

类似地，下述规则突出显示能向用户说明如何操作的 input 元素，以及 title 属性中包含子串“format”的 input 元素：

```
input[title*="format"] {background-color: #dedede;}

<input type="tel"
    title="Telephone number should be formatted as XXX-XXX-XXXX"
    pattern="\d{3}\-\d{3}\-\d{4}">
```

模式类的类名经常使用子串属性选择符匹配一系列类。以前面举过的按钮标记为例，我们可以选择类名以“btn”和一个英文破折号开头的元素，以及包含一个英文破折号后跟子串“arrow”的元素：

```
*[class|="btn"][class*="-arrow"]:after { content: "▾";}

<button class="btn-small-arrow-active">Click Me</button>
```

这种选择符做的是精准匹配，如果选择符中有空白，属性的值中也要有。如果底层文档语言区分大小写，属性的名称和值也要区分大小写。类名、标题、URL 和 ID 的值都区分大小写，但是 HTML 属性值中的关键字，例如输入框的类型，不区分大小写：

```
input[type="CHeckBoX"] {margin-right: 10px;}

<input type="checkbox" name="rightmargin" value="10px">
```

匹配属性值开头的子串

如果想根据属性值开头的子串选择元素，你需要的是 [att^="val"] 形式的属性选择符。如果想以不同方式显示不同类型的链接，就可以使用这种选择符，如图 2-13 所示。

```
a[href^="https:"] {font-weight: bold;}
a[href^="mailto:"] {font-style: italic;}
```



图 2-13：根据属性值开头的子串选择元素

此外，也可以用这种选择符装饰文章中用作插图的图像（例如你在本书中看到的那些）。假设插图的 alt 属性值都是“Figure 5”这种形式（这一假设完全是合理的），那么就可以像下面这样只选择这些图像：

```
img[alt^="Figure"] {border: 2px solid gray; display: block; margin: 2em auto;}
```

这样做的潜在缺点是，alt 属性的值以“Figure”开头的任何 img 元素都会被选中，而不管到底是不是插图。这种情况会不会发生，取决于文档。

另外，还可以使用这种选择符选择日程表中排在周一做的事。假设所有事件都有 title 属性，而且其值的格式是“Monday, March 5th, 2012”。那么，我们便可以使用 [title^="Monday"] 选择这些事件。

匹配属性值结尾的子串

与匹配开头的子串相对的是匹配结尾的子串，这种选择符的形式为 [att$="val"]。我们经常使用这种选择符根据目标资源的类型装饰链接，例如以独特的方式装饰 PDF 文档，如图 2-14 所示。

```
a[href$=".pdf"] {font-weight: bold;}
```



图 2-14：根据属性值结尾的子串选择元素

类似地，还可以根据图像的格式选择图像（肯定事出有因）：

```
img[src$=".gif"] {...}
img[src$=".jpg"] {...}
img[src$=".png"] {...}
```

对前一节的日程表示例来说，可以使用类似 [title$="2015"] 的选择符选择某一年内发生的事件。



你可能注意到了，属性选择符中的属性值都放在引号内。如果属性值包含特殊字符、以英文破折号或数字开头，或者是其他无效的标识符，那就必须放在引号内。尽管使用引号是为了把无效的标识符变成有效的值，但是为了安全起见，建议始终把属性选择符中的属性值放在引号内。

2.4.4 不区分大小写的标识符

CSS Selectors Level 4 为属性选择符引入了一个不区分大小写的选项。在结束方括号前加上 i，属性选择符便不管文档语言的要求，匹配属性值时不区分大小写。

假设你想选择所有指向 PDF 文档的链接，但是不知道 URL 是以 *.pdf*、*.PDF* 还是 *.Pdf* 结尾的。此时可以这样写：

```
a[href$='.PDF' i]
```

加上不起眼的 i 之后，这个选择符便能匹配 href 属性的值以 .pdf 结尾的任何 a 元素，而不管 P、D 和 F 三个字母的大小写。

前文所述的所有属性选择符都可以使用这个不区分大小写的选项。然而要注意，它只针对属性选择符中的值，不涉及属性的名称。因此，在区分大小写的语言中，planet[type*="rock" i] 将匹配下述所有元素：

```
<planet type="barren rocky">Mercury</planet>
<planet type="cloudy ROCKY">Venus</planet>
<planet type="life-bearing Rock">Earth</planet>
```

但是不匹配下面这个元素，因为 TYPE 属性与 type 不匹配：

```
<planet TYPE="dusty rock">Mars</planet>
```

再次注意，这里说的是元素和属性句法区分大小写的语言。XHTML 就是如此。对不区分大小写的语言，如 HTML5，这不是问题。



截至 2017 年年末，Opera Mini、Android 浏览器和 Edge 不支持这个选项。

2.5 根据文档结构选择

CSS 功能强大，它通过文档的结构确定使用哪些样式，以及如何应用样式。其实，文档结构在应用样式的过程中起到的作用还有很多。在介绍更强大的选择符之前，先花点儿时间讨论文档结构。

2.5.1 理解父子关系

为了弄清选择符和文档的关系，我们要再一次分析文档的结构。以下述十分简单的 HTML 文档为例：

```
<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 发挥功能在很大程度上依赖于元素的父子关系。HTML 文档中的元素是一种层次结构（其实，多数结构化文档都是如此），从文档的“树状”视图可见一斑（见图

2-15)。在这个层次结构中，每个元素都在文档的整体结构中占据一席之地。文档中的元素，要么是另一个元素的父元素，要么是另一个元素的子元素，而且经常二者兼具。

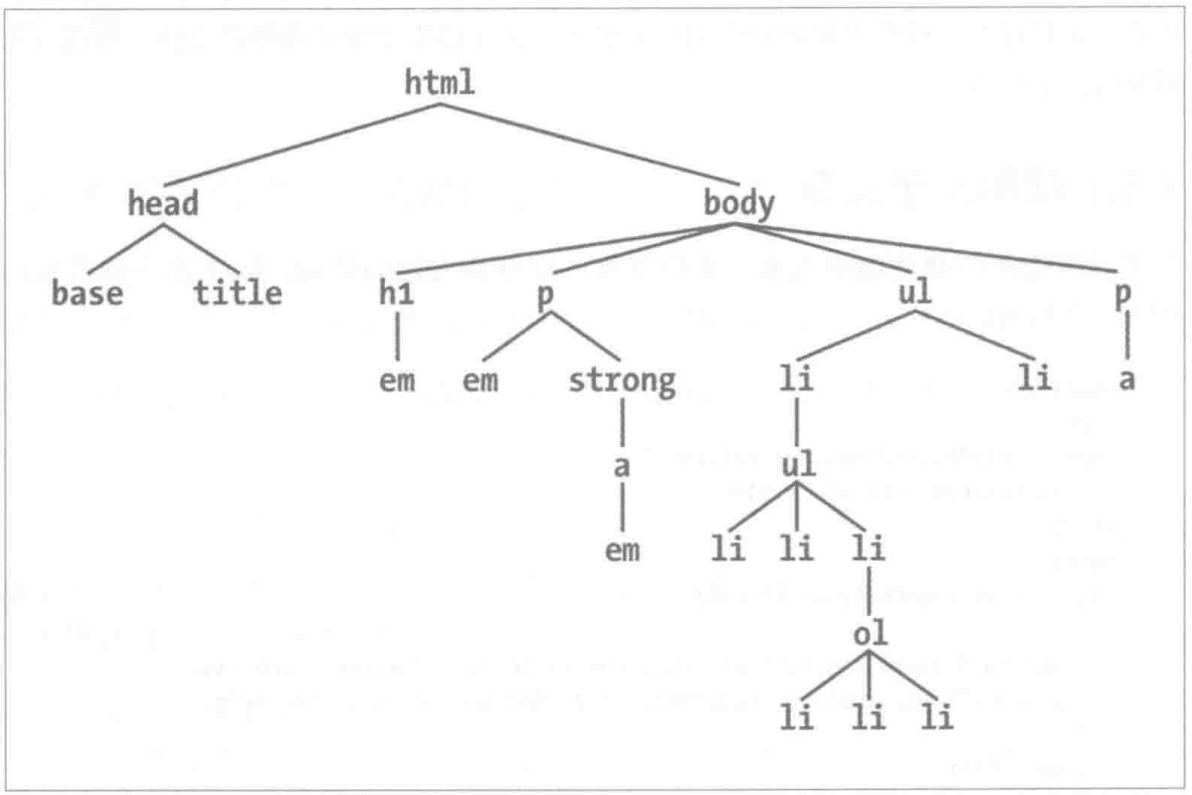


图 2-15：文档的树状结构

在文档的层次结构中，如果一个元素的位置直接在另一个元素的上方，我们说前者是后者的父元素。例如，在图 2-15 中，第一个 p 元素是 em 和 strong 元素的父元素，strong 元素是锚记元素（a）的父元素，锚记元素又是另一个 em 元素的父元素。反过来，如果一个元素的位置直接在另一个元素的下方，前者是后者的子元素。因此，图 2-15 中的锚记元素是 strong 元素的子元素，strong 元素是 p 元素的子元素，p 元素又是 body 元素的子元素等。

“父元素”和“子元素”是“祖辈元素”和“后代元素”的特例。二者之间有区别：在树状视图中，如果两个元素所在的层级是连续的，它们之间是父子关系；如果两个元素之间跨两个层级以上，它们之间是祖辈和后代的关系，而不是父子关系（子元素也是后代，父元素也是祖辈）。在图 2-15 中，第一个 ul 元素是随后两个 li 元素的父元素，此外它还是 li 元素后代（一直延续到嵌套层级最深的那几个 li 元素）的祖辈。

图 2-15 中的锚记不仅是 strong 的子元素，还是 p、body 和 html 元素的后代。body 元素是浏览器中默认显示的所有元素的祖辈，而 html 是整个文档的祖辈。鉴于此，HTML 或 XHTML 文档中的 html 元素也称为根元素（root element）。

2.5.2 后代选择符

理解这个结构模型之后，首先得到的好处是定义后代选择符（descendant selector，也叫上下文选择符，contextual selector）。后代选择符定义的规则针对特定的结构。下面举个例子。假设你只想装饰 h1 元素中的 em 元素。你可以为 h1 中的每个 em 元素设定 class 属性，但是这样做跟使用 font 标签一样浪费时间。如果能直接选择 h1 元素中的 em 元素，效率要高得多。

此时，样式规则可以写成这样：

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

这个规则把作为 h1 元素后代的 em 元素中的文本显示为灰色。其他 em 元素，例如段落中的或引文中的，不受这个规则影响。从图 2-16 中可以清楚地看出这一点。



图 2-16：根据上下文选择元素

在后代选择符中，规则中的选择符由两个或多个空格分隔的选择符构成。选择符之间的空格是一种连结符。如果从右向左读，空格连结符可以理解为“在内部”“是其一部分”或者“是其后代”。因此，h1 em 可以理解为“作为 h1 元素后代的 em 元素”（从左向右读的话，可以理解为“h1 中的 em 应用后面的样式”）。

后代选择符中不只可以使用两个单独的选择符。例如：

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

此时，一个无序列表中的有序列表中的无序列表中的强调文字将显示为灰色，如图 2-17 所示。显然，这针对的是非常具体的结构。



图 2-17：一个非常具体的后代选择符

后代选择符的功能极其强大，某些绝对无法在 HTML 中实现的效果（至少是不大量使用 font 标签），可以借助后代选择符轻易实现。假设一个文档中有一个侧边栏和一个主区域，侧边栏的背景为蓝色，主区域的背景是白色，而且两个区域中都有一个链接列表。此时不能把所有链接都设为蓝色，否则侧边栏中的链接就看不到了。

这种情况便可以使用后代选择符。我们先把侧边栏元素的 class 属性设为 sidebar，把主区域元素的 class 属性设为 main，然后编写如下的样式：

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

结果如图 2-18 所示。

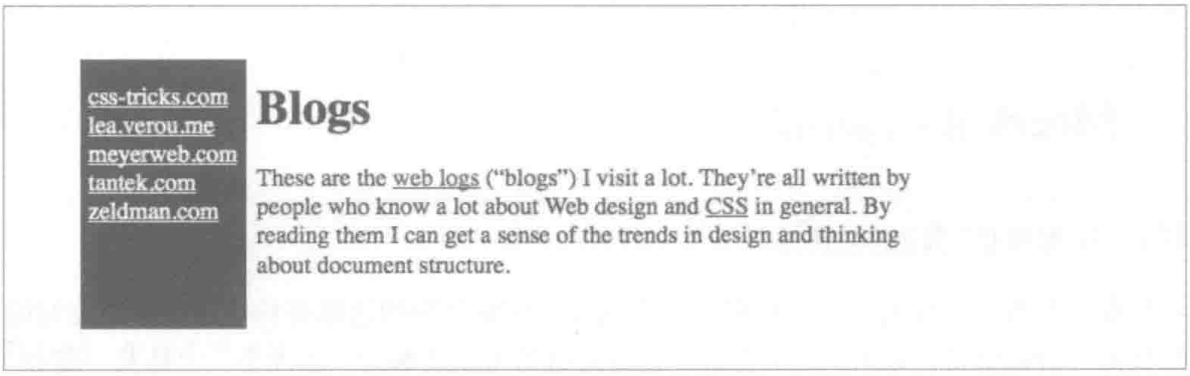


图 2-18：使用后代选择符为相同类型的元素应用不同的样式



:link 针对尚未访问的链接。详情参见本章后面“超链接伪类”一节。

再举个例子。假如想把 blockquote 中的 b 元素和常规段落中的 b 元素都显示为灰色，可以这么写：

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

结果是，段落或引文块中的 b 元素都显示为灰色。

人们经常忽略后代选择符的一点是，两个元素之间的层级间隔可以是无限的。比如说，ul em 选择的是作为 ul 元素后代的任何 em 元素，不管 em 嵌套的有多深。因此，ul em 能选择下述标记中的 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>
```

后代选择符一个更容易被人忽视的细节是，它对元素的距离一无所知。也就是说，文档树中两个元素的距离对是否应用规则没有影响。遇到特指度（后文介绍）和相互抵消的规则时这一点很重要。

例如，对下述规则和标记来说（其中有个选择符在 2.6.7 节讨论）：

```
div:not(.help) span {color: gray;}
div.help span {color: red;}

<div class="help">
   <div class="aside">
      This text contains <span>a span element</span> within.
   </div>
</div>
```

第一个样式规则的意思是，“class 属性中不包含 help 这个词的 div 元素中的 span 元素显示为灰色”；而第二个样式规则的意思是，“class 属性中包含 help 这个词的 div 元素中的 span 元素显示为红色”。对上述标记片段来说，两个规则都将应用到 span 元素上。

因为这两个规则的权重相等，而且设为红色的规则在后面，所以 span 元素将应用后者，显示为红色。div class="aside" 离 span 元素的距离比 div class="help" 近，但这无关紧要。再次说明，后代选择符对元素的距离一无所知。两个规则都匹配，但是只有一个颜色起作用，鉴于 CSS 的工作方式，红色胜出（原因参见下一章）。

2.5.3 选择子元素

有时，我们不想选择所有后代元素，而是想缩小范围，只选择一个元素的子元素。比如说，你可能想选择作为 h1 元素的子元素的 strong 元素（而不是层级更深的后代）。为此，要使用子代连结符，即大于号（>）：

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

这个规则将把第一个 h1 中的 strong 元素显示为红色，第二个则不受影响：

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

从右向左读，h1 > strong 的意思是，“选择的 strong 元素是 h1 元素的直接子代”。子代连结符两侧的空格是可选的，因此，h1 > strong、h1> strong 和 h1>strong 是等效的。你可以根据需要使用或省略空格。

在文档的树状结构中可以清楚地看出，子代选择符选择的是直接连接的元素。图 2-19 是文档树的一部分。

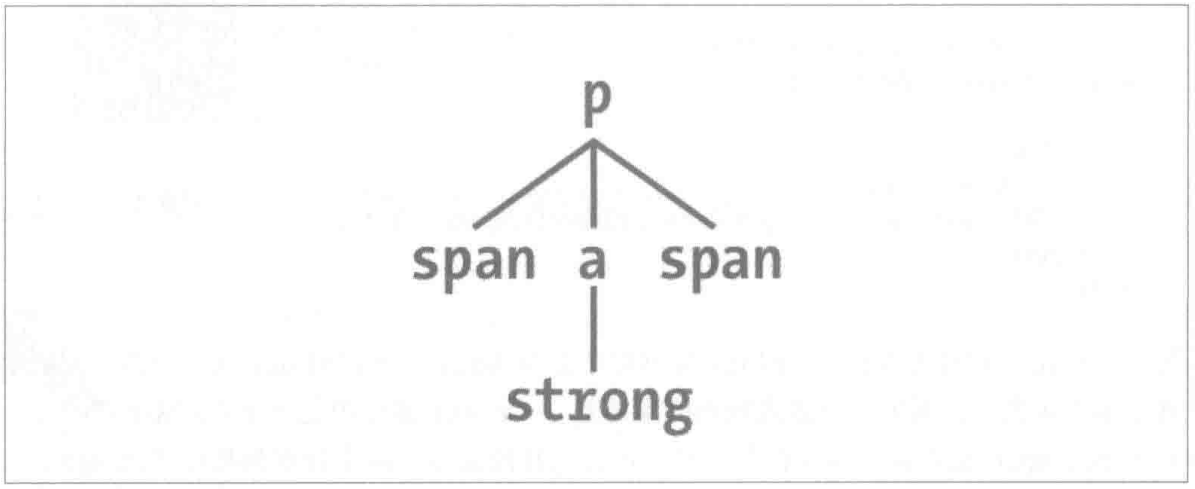


图 2-19：一个文档树片段

在这个文档树片段中有几个父子关系。例如，a 是 strong 的父元素，而它又是 p 的子元素。匹配这些元素可以使用 p > a 和 a > strong 选择符，但是不能使用 p > strong，因为 strong 是 p 的后代，而非直接子元素。

同一个选择符中还可以同时使用后代选择符和子代选择符。因此，table.summary td > p 选择的 p 元素是 td 元素的子元素，而 td 元素是 class 属性的值中包含 summary 这个词的 table 元素的后代。

2.5.4 选择紧邻同胞元素

假设你想装饰紧跟在一个标题后面的段落，或者为紧跟在一个段落后面的列表设定特殊的外边距。若想选择同一个父元素中紧跟在另一个元素后面的一个元素，要使用紧邻同胞连结符（adjacent-sibling combinator），即一个加号（+）。与子元素连结符类似，加号两侧可以有空格，也可以没有，完全由编写样式的人自行决定。

若想把紧跟在 h1 元素后面的段落的上外边距去掉，这样写：

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

这个选择符的意思是，“选择的 p 元素紧跟在 h1 元素后面，而且二者同属一个父元素”。

为了说清这个选择符的工作方式，再次以一个文档树片段为例，如图 2-20 所示。

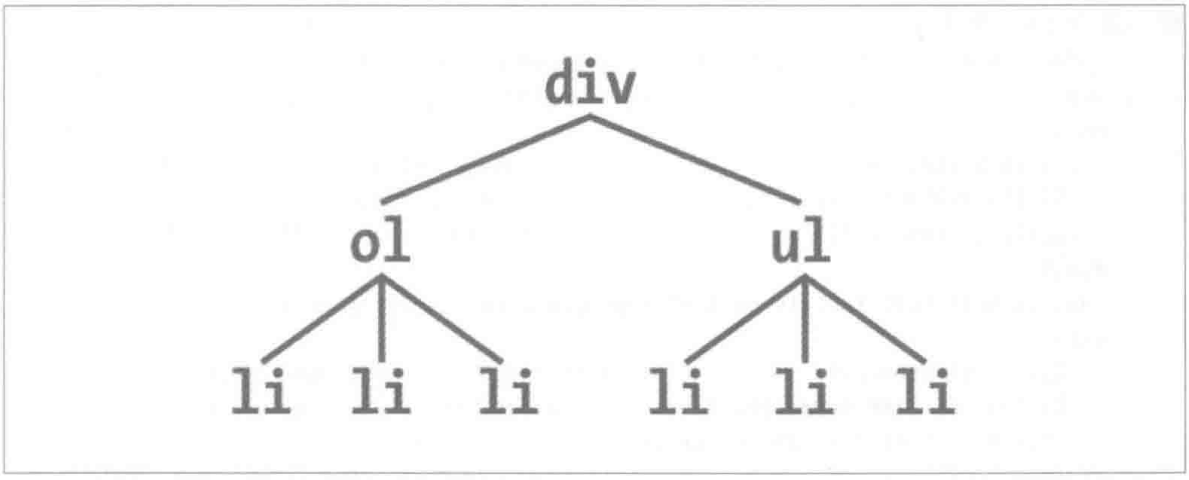


图 2-20：另一个文档树片段

在这个片段中，div 元素有两个后代，一个是有序列表，一个是无序列表，而且二者都有三个列表项目。这两个列表是紧邻同胞，每个列表中的列表项目也是紧邻同胞。然而，第一个列表中的列表项目与第二个列表中的列表项目不是紧邻同胞，因为它们分属不同的父元素（它们最多算是表亲，而 CSS 没有表亲选择符）。

注意，使用一个连结符只能选择紧邻同胞中的第二个元素。因此，对 li + li {font-weight: bold;} 来说，只有第二个和第三个项目会显示为粗体，第一个项目不受影响，如图 2-21 所示。



图 2-21：选择紧邻同胞

为了正常工作，CSS 要求两个元素的顺序与“原始顺序”一样。在前面的示例中，ol 元素后面是 ul 元素。因此，可以使用 ol + ul 选择后一个元素，但是不能使用相同的句法选择前一个元素。若想让 ul + ol 成功匹配，有序列表必须紧跟在无序列表后面。

注意，两个元素之间的文本不影响紧邻同胞连结符的作用。以下述标记片段为例（文档树与图 2-19 一样）：

```
<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>
```

即使两个列表之间有文本，仍然不妨碍使用 ol + ul 匹配第二个列表。这是因为处在中间的文本不算是同胞元素，而是父元素 div 的一部分。如果把那段文本放在 p 元素中，ol + ul 就无法匹配第二个列表。此时，可以写成 ol + p + ul。

如下面的示例所示，紧邻同胞连结符可以与其他连结符搭配使用：

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

这个选择符的意思是，“选择的 ul 元素紧跟在身为同胞的 table 元素后面，这个 table 元素是 body 元素的后代，而 body 是 html 的子元素”。

与其他连结符一样，紧邻同胞连结符也可以实现复杂的选择符，例如 div#content h1 + div ol。这个选择符的意思是，“选择的 ol 元素是 div 元素的后代，div 元素是 h1 的紧邻同胞，而 h1 是 id 属性的值为 content 的 div 元素的后代。”

2.5.5 选择后续同胞

Selectors Level 3 引入一个新的同胞连结符，名为一般同胞连结符（general sibling combinator）。这个连结符使用波浪号（~）表示，选择一个元素后面同属一个父元素的另一个元素。

举个例子。若想让 h2 后面与它同属一个父元素的 ol 元素的文本倾斜，可以编写 h2 ~ol {font-style: italic;}。两个元素不一定非得是紧邻同胞，不过是的话也能匹配。对下面的标记来说，应用这个规则后得到的结果如图 2-22 所示。

```
<div>
  <h2>Subheadings</h2>
  <p>It is the case that not every heading can be a main heading.  Some headings
  must be subheadings.  Examples include:</p>
  <ol>
    <li>Headings that are less important</li>
    <li>Headings that are subsidiary to more important headlines</li>
    <li>Headings that like to be dominated</li>
  </ol>
  <p>Let's restate that for the record:</p>
  <ol>
    <li>Headings that are less important</li>
    <li>Headings that are subsidiary to more important headlines</li>
    <li>Headings that like to be dominated</li>
  </ol>
</div>
```

可以看出，两个有序列表中的文本都倾斜了。这是因为两个 ol 元素都在 h2 后面，而且同属一个父元素（div 元素）。

2.6 伪类选择符

讲到伪类选择符（pseudo-class selector），事情就变得有趣了。利用这种选择符可以为文档中不一定真实存在的结构指定样式，或者为某些元素（甚至文档本身）的特定状态赋予幽灵类。

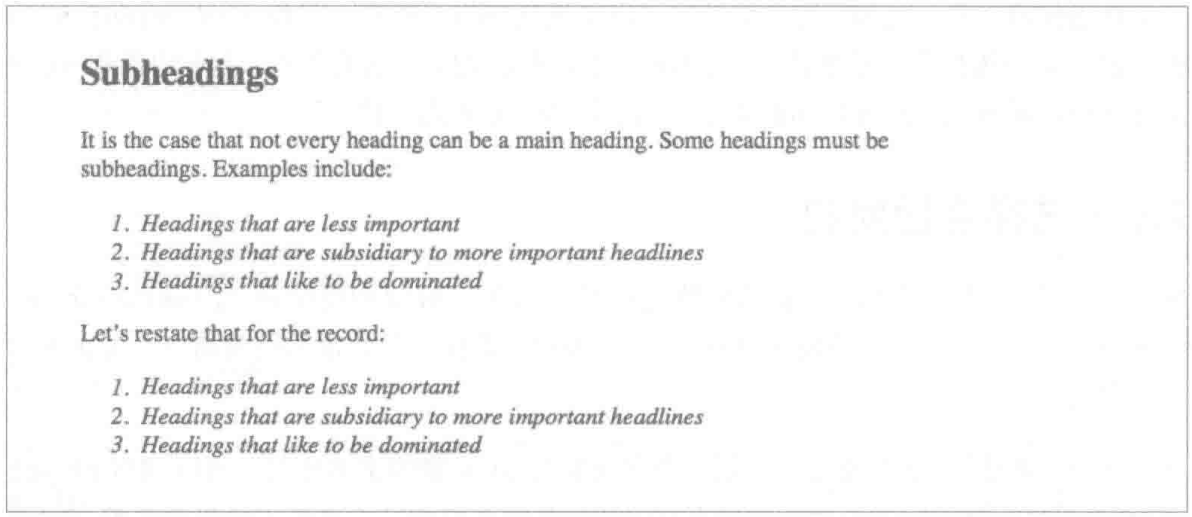


图 2-22：选择后续同胞

“幽灵类”听起来有点古怪，不过这样说却是理解伪类的最好方式。假设你想突出显示一个数据表中相隔的行。为此，你可以为相隔的行设定 `class="even"` 属性，然后编写 CSS，突出显示具有那个类的行。此外，还可以通过伪类选择符以十分相似的方式实现同样的效果（具体做法参见后文）。

2.6.1 拼接伪类

开始之前，先说一下串联。CSS 允许把伪类拼接（串联）在一起。例如，可以把有鼠标悬停其上的未访问链接显示为红色，把有鼠标悬停其上的已访问链接显示为红褐色：

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

拼接伪类的顺序其实没什么关系，也可以写成 `a:hover:link`，它的效果与 `a:link:hover` 一样。此外，还可以分别为其他语言（例如德语）下的未访问链接和已访问链接指定悬停样式：

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

留意，别拼接相互排斥的伪类了。例如，链接不可能既已访问又未访问，因此 `a:link:visited` 没有任何意义，匹配不了任何东西。

2.6.2 结构伪类

伪类大多数都是结构上的，即它们指代文档中的标记结构。其中多数依赖标记中的模式，例如选择每隔两个的第三个段落，其他的则用于选择特定的元素类型。所有伪类无一例外都是一个冒号（:）后面跟着一个词，而且可以出现在选择符的任何位置。

深入讨论之前，对伪类要明确一点：伪类始终指代所依附的元素。这听起来有点奇怪，但是又理所当然，不是吗？之所以强调这一点，是因为有几个结构伪类容易让人误以为是描述符，认为指代的是后代元素。

下面笔者举个例子。2003 年，我的第一个孩子出生时，我（像你一样）在网上宣布了这个好消息。有些人对我表示了祝贺，还用 CSS 开起了玩笑，比如 `#ericmeyer:first-child` 选择符。这个选择符的问题是，它选择的是我，而不是我的女儿，而且我必须是爸妈的第一个孩子才行（碰巧我是）。若想正确选择我的第一个孩子，选择符应该是 `#ericmeyer > :first-child`。

有人不理解很正常，所以我才在这里提到这个问题。下面几小节还会不断提醒你。你只要记住一点，伪类的效果是把某种幽灵类应用到伪类依附的元素上，这样就够了。

选择根元素

结构伪类的简单由此体现：`:root` 伪类选择文档的根元素。在 HTML 中，根元素始终是 `html`。这个选择符的真正用途体现在 XML 语言的样式表中，在不同的 XML 语言中，根元素有所不同。例如，RSS 2.0 的根元素是 `rss`。有些 XML 语言甚至有多个根元素（当然不是在同一个文档中）。

下面是装饰 HTML 根元素的一个例子，结果如图 2-23 所示。

```
:root {border: 10px dotted gray;}
body {border: 10px solid black;}
```

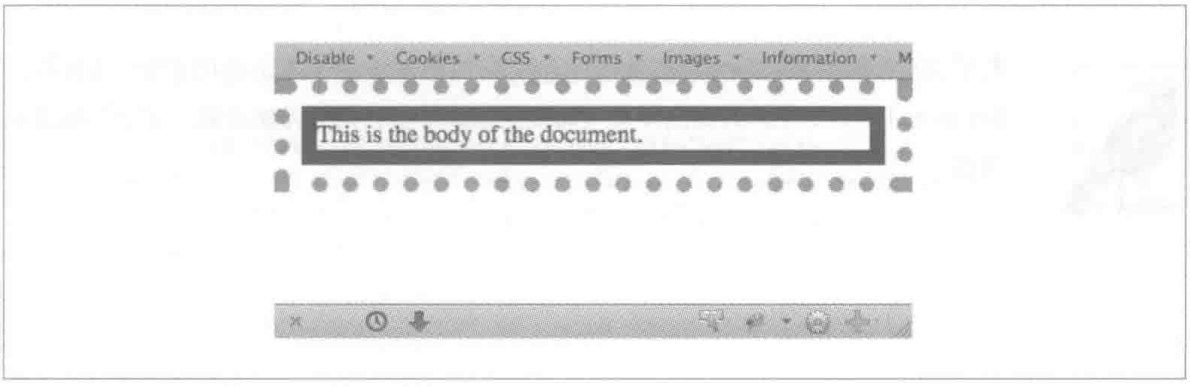


图 2-23：装饰根元素

在 HTML 文档中可以直接选择 html 元素，而不使用 :root 伪类。然而这两个选择符的特指度不同，详情参见第 3 章。

选择空元素

使用 :empty 伪类可以选择没有任何子代的元素，甚至连文本节点都没有（包括文本和空白）。CMS 经常生成没有任何内容的空元素，此时便可以使用这个伪类。例如，p:empty {display: none;} 能禁止显示空段落。

注意，为了能正确匹配，从解析的角度来看，元素必须真的为空，没有有空白、可见内容或后代元素。对下面几个元素来说，只有第一个和最后一个能被 p:empty 匹配：

```
<p></p>
<p> </p>
<p>
</p>
<p><!-- 注释 --></p>
```

第二个和第三个段落不能被 :empty 匹配，因为它们不是空的，而是分别有一个空格和一个换行符。这两种空白都算文本节点，因此也就不是空的。最后一个段落之所以能匹配，是因为注释不是内容，也不是空白。然而，如果在注释的某一边加上一个空格或一个换行符，p:empty 就无法匹配了。

你可能想使用 *:empty {display: none;} 装饰所有空元素，但是这里有一个陷阱，:empty 是能匹配 HTML 文档中的空元素，例如 img 和 input，但是还能匹配里面没有内容的 textarea。就匹配元素的效果而言，img 和 img:empty 其实是一样的（不过二者在特指度上有区别，详情参见下一章）。

```
<img src="salmon.png" alt="North Pacific Salmon">
<br>
<input type="number" min="-1" max="1" step=".01"/>
<textarea></textarea>
```



截至 2017 年年末，:empty 是唯一一个在匹配时考虑文本节点的 CSS 选择符。Selectors Level 3 中的其他选择符（比如同胞选择符）都只考虑元素，完全忽略文本节点。

选择唯一的子代

如果想选择带超链接的图像，可以使用 :only-child 伪类。它选择的元素是另一个元素的唯一子元素。假设你想为作为另一个元素唯一子元素的图像加上边框，可以使用：

```
img:only-child {border: 1px solid black;}
```

这个规则会应用到符合条件的每一个图像上。因此，如果一个段落中只有一个图像，没有其他子元素，那个图像就会独立于周围的文本被选中。如果想选择作为超链接唯一子代的图像，只需把选择符改为（结果见图 2-24 ）：

```
a[href] img:only-child {border: 2px solid black;}

<a href="http://w3.org/"><img src="w3.png" alt="W3C"></a>
<a href="http://w3.org/"><img src="w3.png" alt=""> The W3C</a>
<a href="http://w3.org/"><img src="w3.png" alt=""> <em>The W3C</em></a>
```



图 2-24：选择作为链接唯一子代的图像

关于 :only-child 伪类，有两点要注意。首先，前文说过，它始终依附在希望是唯一子元素的那个元素上，而不是父元素。由此引出第二点，即在后代选择符上使用 :only-child 伪类时，列出的元素不一定是父子关系。

对前面超链接中的图像示例来说，a[href] img:only-child 匹配的图像是唯一的子元素，而且是 a 元素的后代，而不是 a 元素的子元素。匹配时，那个元素必须是其直接父元素的唯一子元素，而且是链接的后代，但是图像的父元素自身也可以是链接的后代。因此，下面三个图像都能匹配，如图 2-25 所示。

```
a[href] img:only-child {border: 5px solid black;}

<a href="http://w3.org/"><img src="w3.png" alt="W3C"></a>
<a href="http://w3.org/"><span><img src="w3.png" alt="W3C"></span></a>
<a href="http://w3.org/">A link to <span>the <img src="w3.png" alt="">
   web</span> site</a>
```



图 2-25：选择链接中作为唯一子代的图像

在三个链接中，图像都是其父元素的唯一子元素，而且都是 a 元素的后代。因此，示例中的规则能匹配全部三个图像。如果想限制规则，只让它匹配作为 a 元素唯一子代的图像，要加上子元素连结符，改成 a[href] > img:only-child。这样修改之后，图 2-25 中的三个图像，只有第一个图像能匹配。

好了，如果想选择超链接中唯一的图像，而链接中还有其他内容该怎么办呢？比如说下面这个链接：

```
<a href="http://w3.org/"><b>•</b><img src="w3.png" alt="W3C"></a>
```

这里，a 元素有两个子代：b 和 img。那个图像不再是父元素（超链接）的唯一子元素，因此无法使用 :only-child 匹配。然而，却能被 :only-of-type 匹配。下述规则和标记的结果如图 2-26 所示。

```
a[href] img:only-of-type {border: 5px solid black;}

<a href="http://w3.org/"><b>•</b><img src="w3.png" alt="W3C"></a>
<a href="http://w3.org/"><span><b>•</b><img src="w3.png" alt="W3C"></span></a>
```



图 2-26：选择同胞元素中唯一的图像

这两个伪类之间的区别是，:only-of-type 匹配同胞中唯一的那种元素，而 :only-child 只匹配完全没有同胞的元素。

使用 :only-of-type 可以放心选择段落中的图像，而不必担心段落中有超链接或其他行内元素：

```
p > img:only-of-type {float: right; margin: 20px;}
```

只要同一个段落中的图像不超过两个，那个图像就会浮动显示。使用这个伪类还可以为一节中唯一的 h2 添加额外的样式，例如：

```
section > h2 {margin: 1em 0 0.33em; font-size: 1.8rem; border-bottom: 1px solid
   gray;}
section > h2:only-of-type {font-size: 2.4rem;}
```

对上述示例来说，section 中只有一个 h2 子元素时，h2 将显示的比通常大一些。如果一个 section 中有两个或多个 h2 子元素，它们的大小将相等。有没有其他元素，例如其他级别的标题、段落、表格、列表等，对匹配效果没有影响。

还有一点要澄清，:only-of-type 指代的是元素，而不是其他任何东西。以下述规则和标记为例：

```
p.unique:only-of-type {color: red;}

<div>
  <p class="unique">This paragraph has a 'unique' class.</p>
  <p>This paragraph doesn't have a class at all.</p>
</div>
```

这里，两个段落都不会被选中。为什么？因为两个段落都是 div 的后代，因此不可能是唯一一个段落类型。

这里出现的类名无关紧要。不要误以为“类型”是一种泛称，对 :only-of-type 来说，“类型”特指“元素类型”。因此，p.unique:only-of-type 的意思是，“选择的 p 元素的 class 属性中包含 unique 这个词，而且 p 元素是同胞中唯一的一个”，而不是作为同胞的段落中唯一一个 class 属性中包含 unique 这个词的 p 元素。

选择第一个和最后一个子代

为一个元素的第一个或最后一个子元素应用特殊的样式是十分常见的需求。比如说，装饰导航栏中的链接时，可能想为第一个或最后一个选项卡（或者同时）赋予特殊的视觉效果。以前的做法是，为它们设定特殊的类。现在，可以让伪类代劳。

:first-child 伪类选择一个元素的第一个子元素。以下述标记为例：

```
<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>
```

在这个示例中，第一个 p、第一个 li，以及 strong 和 em 都是相应父元素的第一个子元素。如果有下面两个规则：

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

得到的结果如图 2-27 所示。

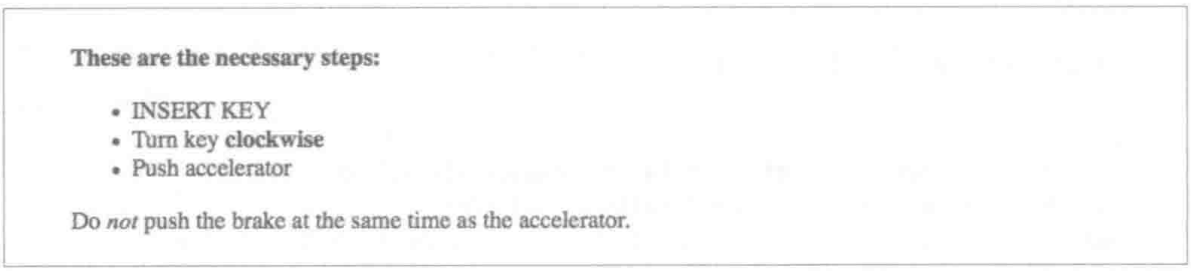


图 2-27：装饰第一个子代

第一个规则把一个元素中的第一个 p 元素显示为粗体。第二个规则把一个元素（对 HTML 来说，必须是 ol 或 ul 元素）中的第一个 li 元素显示为全大写形式。

同样，人们对 p:first-child 这样的选择符最大的误解是，认为它选择的是 p 元素的第一个子元素。还记得伪类的特点吗？它为所依附的元素设定某种幽灵类。如果添加真正的类，标记将变成：

```
<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，如果想选择一个元素中的第一个 em 子元素，可以使用 em:first-child 选择符。

与 :first-child 对应的是 :last-child。还以前面的例子为例，如果只修改伪类，得到的结果如图 2-28 所示。

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

<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>
```

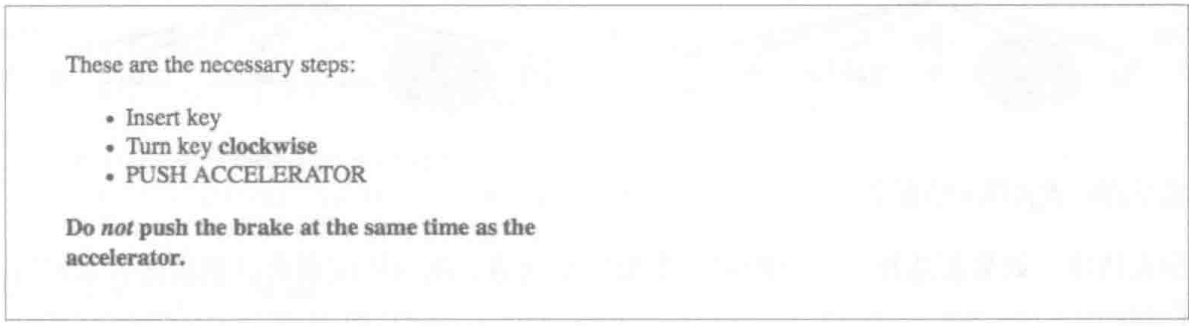


图 2-28：装饰最后一个子代

第一个规则把一个元素中的最后一个 p 元素显示为粗体。第二个规则把一个元素中的最后一个 li 元素显示为全大写形式。如果想选择最后一个段落中的 em 元素，使用 p:last-child em 选择符，它选择的 em 元素是 p 元素的后代，而且 p 元素是另一个元素的最后一个子元素。

有趣的是，这两个伪类结合在一起的效果相当于 :only-child。下述两个规则选择的是相同的元素：

```
p:only-child {color: red;}
p:first-child:last-child {background-color: red;}
```

这两个规则在一起把段落设为红底红字（这么做显然不好）。

选择第一个和最后一个某种元素

除了选择一个元素中的第一个和最后一个子代之外，还可以选择一个元素中某种元素的第一个或最后一个。例如，选择一个元素中的第一个 table，而不管它前面有什么元素。

```
table:first-of-type {border-top: 2px solid gray;}
```

注意，这个伪类不应用于整个文档，即上述规则不是选择文档中的第一个表格，其他的表格都不算，而是选择里面有表格的元素中的第一个 table，跳过后面作为同胞的其他 table 元素。因此，对图 2-29 所示的文档结构来说，选中的是圈出的那两个节点。

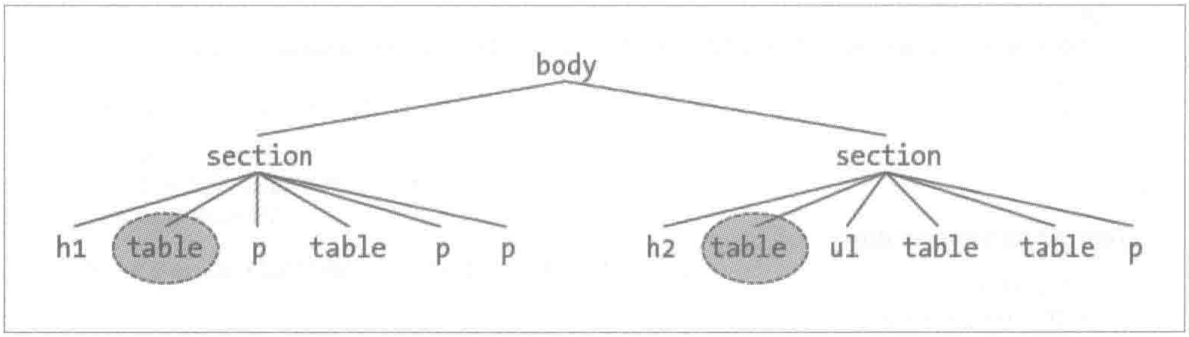


图 2-29：选择第一个表格

在表格中，如果想选择一行中的第一个数据单元格，而不管前面有没有表头，可以这样写：

```
td:first-of-type {border-left: 1px solid red;}
```

对下述表格行来说，这将选择每一行中的第一个数据单元格：

```
<tr>
  <th scope="row">Count</th><td>7</td><td>6</td><td>11</td>
</tr>
<tr>
  <td>Q</td><td>X</td><td>-</td>
</tr>
```

倘若使用 td:first-child，第二行中的 td 元素能被选中，但是第一行就无法选中了。

与 :first-of-type 对应的是 :last-of-type，它从同胞元素中选择指定种类元素的最后一个。在某种意义上，:last-of-type 与 :first-of-type 特别相似，只不过它是从同胞元素的最后一个开始向前搜索，直到找到指定的元素类型。对图 2-30 所示的文档结构来说，table:last-of-type 选中的是圈出的节点。

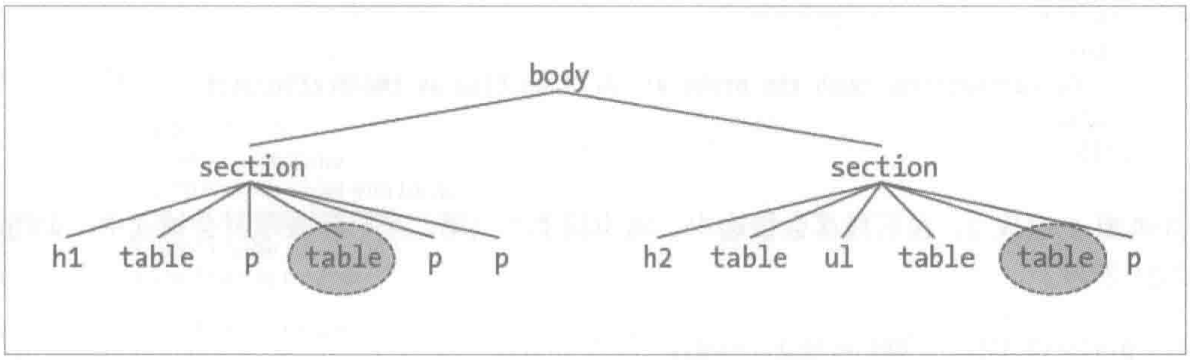


图 2-30：选择最后一个表格

注意，与 :only-of-type 一样，你是从一组同胞元素中选择一个类型的元素，各组同胞是分开对待的。也就是说，不是把整个文档中某种类型的第一个（或最后一个）元素作为一个整体选择出来。同属一个父元素的元素是一组，而我们是从这样的一组元素中选择某种元素的第一个（或最后一个）。

与前一节末尾所讲的类似，我们可以把这两个伪类连在一起，达到 :only-of-type 的效果。下述两个规则选择的是相同的元素：

```
table:only-of-type{color: red;}
table:first-of-type:last-of-type {background: red;}
```

选择每第 *n* 个子元素

如果你能选择一个元素的第一个子元素、最后一个子元素和唯一的子元素，那能不能选择每第三个子元素呢，能不能选择所有偶数位的子元素呢，又或者能不能选择第 9 个子元素呢？如果分别为这些需求定义伪类，那数量可就多了。CSS 为此提供的是 :nth-child() 伪类。我们可以在括号中填上整数，甚至是简单的代数式，选择任何想选择的子元素。

先看与 :first-child 等效的 :nth-child(1)。对下述示例来说，选中的是第一个段落和第一个列表项目。

```
p:nth-child(1) {font-weight: bold;}
li:nth-child(1) {text-transform: uppercase;}

<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>
```

如果把 1 改成 2，没有段落会被选中，而中间那个（第二个）列表项目会被选中，如图 2-31 所示。

```
p:nth-child(2) {font-weight: bold;}
li:nth-child(2) {text-transform: uppercase;}
```

These are the necessary steps:

- Insert key
- TURN KEY CLOCKWISE
- Push accelerator

Do *not* push the brake at the same time as the accelerator.

图 2-31：装饰第二个子代

括号中可以填上任何整数。如果在某种情况下需要选择作为一个元素第 93 个子元素的有序列表，可以使用 `ol:nth-child(93)`。这个选择符将匹配第 93 个子元素，前提是它是有序列表（不是在同胞中匹配第 93 个有序列表，详细说明参见下一节）。

更强大的是，括号中可以使用简单的代数式定义公式。代数式的形式为 $a\,n + b$ 或 $a\,n - b$，其中 a 和 b 是具体的整数，n 原封不动。而且，b 和 $-\,b$ 是可选的，如果不需要，可以不用。

假设我们想从一个无序列表的第一个列表项目开始，选择每第三个列表项目。此时，可以使用下述选择符，选择第一个和第四个列表项目，如图 2-32 所示。

```
ul > li:nth-child(3n + 1) {text-transform: uppercase;}
```

这里的 n 表示 0、1、2、3、4，一直到无穷大。浏览器求解 $3\,n + 1$ 时，得到的结果为 1、4、7、10、13 等。如果没有 + 1 那部分，只剩下 $3n$，得到的结果是 0、3、6、9、12 等。因为没有第 0 个列表项目（与数组不同，HTML 元素从第一个数起），所以这个表达式选中的第一个元素是第三个列表项目。

These are the necessary steps:

- INSERT KEY
- Turn key **clockwise**
- Grip steering wheel with hands
- PUSH ACCELERATOR
- Steer vehicle
- Use brake as necessary

Do *not* push the brake at the same time as the accelerator.

图 2-32：装饰每第三个列表项目

正因为元素从 1 数起，要稍微转个圈才能推出 `:nth-child(2n)` 选择的是偶数位的子代，而 `:nth-child(2n+1)` 或 `:nth-child(2n-1)` 选择的是奇数位的子代。你可以选择记住，也可以使用两个特殊的关键字：`even` 和 `odd`。想从表格的第一行起每隔一行突出显示？可以这么做，结果如图 2-33 所示。

```
tr:nth-child(odd) {background: silver;}
```

Missouri	MO	Jefferson City	Eastern Bluebird
Montana	MT	Helena	Western Meadowlark
Nebraska	NE	Lincoln	Western Meadowlark
Nevada	NV	Carson City	Mountain Bluebird
New Hampshire	NH	Concord	Purple Finch
New Jersey	NJ	Trenton	Eastern Goldfinch
New Mexico	NM	Santa Fe	Roadrunner
New York	NY	Albany	Eastern Bluebird
North Carolina	NC	Raleigh	Northern Cardinal
North Dakota	ND	Bismarck	Western Meadowlark
Ohio	OH	Columbus	Northern Cardinal
Oklahoma	OK	Oklahoma City	Scissor-Tailed Flycatcher
Oregon	OR	Salem	Western Meadowlark
Pennsylvania	PA	Harrisburg	Ruffed Grouse

图 2-33：装饰表格中间隔的行

比间隔元素复杂的情况都要使用代数式 *an + b*。

注意，如果想让 *b* 使用负数，要把前面的 + 号删掉，否则选择符无效。对下面两个规则来说，只有第一个会起作用。解析器不解析第二个规则，而是将其忽略。

```
tr:nth-child(4n - 2) {background: silver;}
tr:nth-child(3n +  2) {background: red;}
```

如果想选择从第 9 行起的每一行，可以使用下面两个规则中的一个。二者的共同点是都会选择第 9 行起的每一行，不过后者的特指度更高（参见第 3 章）。

```
tr:nth-child(n + 9) {background: silver;}
tr:nth-child(8) ~ tr {background: silver;}
```

你可能猜到了，有个与之对应的伪类 :nth-last-child()。它的作用与 :nth-child() 一样，只不过是从一组同胞的最后一个元素开始，从后向前计算。如果想突出显示表格中间隔的行，而且想让最后一行含在其中，可以使用下述规则中的任意一个：

```
tr:nth-last-child(odd) {background: silver;}
tr:nth-last-child(2n+1) {background: silver;} /* 等效 */
```

如果更新 DOM，添加或删除了行，无需添加或删除类。借助结构伪类，上述选择符始终能匹配更新后的 DOM 中的奇数行。

只要条件得当，使用 :nth-child() 和 :nth-last-child() 可以选择任何元素。下述规则的结果如图 2-34 所示。

```
li:nth-child(3n + 3) {border-left: 5px solid black;}
li:nth-last-child(4n - 1) {border-right: 5px solid black; background: silver;}
```

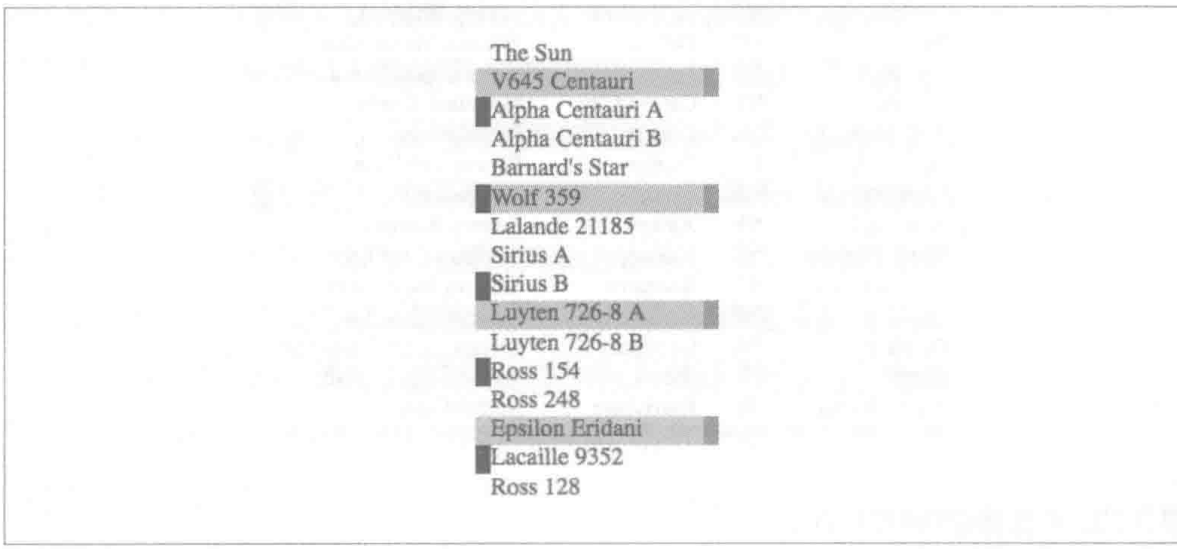


图 2-34：结合 :nth-child() 和 :nth-last-child()

这两个伪类可以串在一起，:nth-child(1):nth-last-child(1) 实现 :only-child 的效果。如果不是想得到较高的特指度，没有任何原因这么做，不过你要知道确实可以这么做。

你可以使用 CSS 确定一个列表中有多少个列表项目，然后据此装饰：

```
li:only-child {width: 100%;}
li:nth-child(1):nth-last-child(2),
li:nth-child(2):nth-last-child(1) {width: 50%;}
li:nth-child(1):nth-last-child(3),
li:nth-child(1):nth-last-child(3) ~ li {width: 33.33%;}
li:nth-child(1):nth-last-child(4),
li:nth-child(1):nth-last-child(4) ~ li {width: 25%;}
```

在上述示例中，如果列表中只有一个列表项目，宽度为 100%；如果一个列表项目是第一个项目，还是倒数第二个项目，即列表中有两个项目，那么宽度为 50%；如果一个列表项目是第一个项目，还是倒数第三个项目，那么后面两个同胞项目的宽度为 33%；类似地，如果一个列表项目是第一个项目，还是倒数第四个项目，即列表中有四个项目，那么后面三个同胞的宽度为 25%。

选择每第 *n* 个某种元素

了解相关模式之后你可能猜到了，`:nth-child()` 和 `:nth-last-child()` 伪类有对应的 `:nth-of-type()` 和 `:nth-last-of-type()`。例如，可以使用 `p > a:nth-of-type(even)` 在一个段落中从第二个超链接起选择间隔的超链接。此时，其他元素（span、strong 等）都被忽略，只考虑超链接，结果如图 2-35 所示。

```
p > a:nth-of-type(even) {background: blue; color: white;}
```

ConHugeCo is the industry leader of web-enabled ROI metrics. Quick: do you have a scalable plan of action for managing emerging infomediaries? We invariably cultivate enterprise eyeballs. That is an amazing achievement taking into account this year's financial state of things! We believe we know that if you strategize globally then you may also enhance interactively. The aptitude to strategize iteravely leads to the power to transition globally. The accounting factor is dynamic. If all of this sounds amazing to you, that's because it is! Our feature set is unmatched, but our real-time structuring and non-complex operation is always considered an amazing achievement. The paradigms factor is fractal. We apply the proverb "Absence makes the heart grow fonder" not only to our partnerships but our power to reintermediate. What does the term "global" really mean? Do you have a game plan to become C2C2C? We will monetize the ability of web services to maximize.

(Text courtesy http://andrewdavidson.com/gibberish/)

图 2-35：选择偶数位的链接

如果想从最后一个超链接向前数，要使用 `p > a:nth-last-of-type(even)`。

与之前一样，这两个伪类从同胞元素中选择某一种元素，而不是从整个文档中把某一种元素作为一个整体选择出来。每个元素都有自己的一组同胞，而选择就发生在那组同胞中。

你可能猜到了，串在一起的 :nth-of-type(1):nth-last-of-type(1) 相当于 :only-of-type，二者作用相同，只不过前者特指度更高（别急，第 3 章将说明特指度）。

2.6.3 动态伪类

除了结构伪类之外，还有一些与结构有关的伪类，不过它们在页面渲染之后根据页面的变化而变化。也就是说，为了把这些伪类定义的样式应用到文档中的某一部分上，除了文档的结构，还要考虑其他事情，而且通过文档的标记很难准确推出应用样式的方式。

看起来这好像是随机应用样式，然而并非如此。其实，样式是在无法提前预知的条件满足时应用的。话虽如此，但是应用样式的条件其实是定义好的。拿体育比赛来说，如果主队得分了，观众会沸腾起来。但是，你无法预知一场比赛中主队何时会得分，但是按预计，只要得分，观众就会沸腾起来。我们知道观众在什么情况下会沸腾，但是对何时出现这样的情况却一无所知。

拿锚记元素（a）来说，（在 HTML 及相关的语言中）它的作用是从一个文档链接到另一个文档。锚记始终是锚记，但是有些锚记指向的页面已经访问过，而有些尚未访问。从 HTML 标记中是看不出这一区别的，因为所有锚记的句法都是一样的。若想知道链接到底有没有访问，只能与用户的浏览器历史记录比较。因此，链接基本上有两种状态：已访问和未访问。

超链接伪类

CSS2.1 定义了两个只能在超链接上使用的伪类。在 HTML 中，这两个伪类用在具有 href 属性的 a 元素上；在 XML 语言中，这两个伪类在链接到其他资源的元素上应用。这两个伪类的说明见表 2-2。

表 2-2：链接伪类

伪类	说明
:link	指代用作超链接的锚记（即具有 href 属性），而且指向尚未访问的地址
:visited	指代指向已访问地址的超链接。出于安全考虑，能应用到已访问链接上的样式十分有限。详情参见本章后面“已访问链接的隐私保护”旁注

乍一看表 2-2 中的第一个伪类，可能觉得多余。一个锚记未被访问，那它肯定就处于未访问状态，不是吗？如果是这样，我们只需编写：

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

这样看似合理，但其实还不够。上述规则中的第一条不仅应用到未访问的链接上，还应用到类似下面的占位链接上：

```
<a>4. The Lives of Meerkats</a>
```

这个文本会显示为蓝色，因为 a 元素与 a {color: blue;} 规则匹配。所以，为了避免把链接样式应用到占位链接上，要使用 :link 和 :visited 伪类：

```
a:link {color: blue;}    /* 未访问的链接显示为蓝色 */
a:visited {color: red;}  /* 已访问的链接显示为红色 */
```

此时，我们可以看一下如何把属性选择符和类选择符与伪类结合在一起使用。假设我们想改变指向站外地址的链接颜色。多数情况下，我们可以使用以特定文本开头的属性选择符。然而，在某些 CMS 中，所有链接使用的都是绝对 URL；此时，可以为每个锚记设定类。比如说：

```
<a href="/about.html">My About page</a>
<a href="https://www.site.net/" class="external">An external site</a>
```

然后使用下述规则为外部链接应用不同的样式：

```
a.external:link, a[href^="http"]:link { color: slateblue;}
a.external:visited, a[href^="http"]:visited  {color: maroon;}
```

对上述标记中的第二个锚记来说，默认情况下颜色为石蓝色，访问后变成红褐色；而第一个锚记始终显示超链接的默认颜色（通常在未访问时是蓝色，访问后是紫色）。为了提升可用性和可访问性，已访问链接和未访问链接之间要能轻易区分。



具有特别样式的已访问链接能让访客知道他们访问过哪些网站，以及哪些还未访问。在大型网站中这样做尤其重要，因为链接太多，记不住（特别是对有认知障碍的人）哪些网站已经访问过。突出显示已访问的链接不仅是 W3C Web Content Accessibility Guidelines 的指导方针之一，而且还便于快速且高效地搜索内容，减轻每个人的压力。

ID 选择符也可以跟伪类结合在一起使用：

```
a#footer-copyright:link{background: yellow;}
a#footer-copyright:visited {background: gray;}
```

链接状态的这两个伪类可以串在一起，不过这么做没有任何意义：链接不可能既是已访问的又是未访问的。

已访问链接的隐私保护

有超过十年的时间，已访问的链接可以使用任何可用的 CSS 属性装饰，与未访问链接无异。然而，大约在 2005 年，有几个人通过示例揭露，通过视觉样式和简单的 DOM 脚本可以判断用户是否访问过特定的页面。例如，对 `:visited {font-weight: bold;}` 规则来说，脚本可以找出所有加粗的链接，告诉用户他们访问过哪些网站。更糟的是，已访问的网站可能会被服务器收集。不使用脚本的话，还可以通过背景图像达到相同的效果。

对你来说这可能不是什么严重的问题，但在有些国家，访问某些网站（反对党、未经批准的宗教组织、邪教或腐败网站等）可能招致牢狱之灾。钓鱼网站还可以利用这一点查出用户访问过哪些在线银行。鉴于此，相关方采取了两个措施。

首先，只能把颜色相关的属性应用到已访问的链接上，包括：color、background-color、column-rule-color、outline-color、border-color，以及各边的边框颜色属性（例如 border-top-color）。除此之外的属性将被忽略。此外，:link 定义的样式除了应用到未访问的链接上之外，也会应用到已访问的链接上，因此 :link 能“装饰所有超链接”，而不只是“装饰所有未访问的超链接”。

其次，如果通过 DOM 查询已访问链接的样式，返回的值跟未访问时一样。因此，如果把已访问链接的颜色设为紫色，而未访问链接的颜色设为蓝色，那么通过 DOM 查询颜色时，返回的是蓝色，而不是紫色。

从 2017 年年末起，这一行为在所有浏览模式中都提供，而不仅限于“隐私浏览”模式。尽管只能使用有限的 CSS 属性区分已访问链接和未访问链接，为了可用性和可访问性，我们还是要充分利用有限的属性把已访问的链接和未访问的链接区分开。

用户操作伪类

CSS 中有几个伪类可以根据用户的操作改变文档的外观。这些动态伪类以前普遍用于装饰超链接，不过现在的应用范围宽得多。这些伪类的说明见表 2-3。

表 2-3：用户操作伪类

伪类	说明
:focus	指代当前获得输入焦点的元素，即可以接受键盘输入或以某种方式激活
:hover	指代鼠标指针放置其上的元素，例如鼠标指针悬停在超链接上
:active	指代由用户输入激活的元素，例如用户单击超链接时按下鼠标按键的那段时间

可以处于 :active 状态的元素有链接、菜单项目，以及可以设定 tabindex 属性的元素。这些元素，加上其他所有交互元素，例如表单控件和可编辑内容的元素，还可以获得焦点。

与 :link 和 :visited 类似，这些伪类最常用于超链接。很多网页都有类似下面的样式：

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



这些伪类的顺序可不是随意的，通常推荐的顺序是“link-visited-hover-active”，不过后来改成了“link-visited-focus-hover-active”。下一章将解释为什么采用这种顺序，并讨论可能想改变或忽略推荐顺序的一些原因。

注意，动态伪类可应用于任何元素，也就是说可用于链接之外的元素。例如，使用下述规则可以突出显示获得键盘输入焦点的表单元素，效果如图 2-36 所示。

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



图 2-36：突出显示获得焦点的表单元素

你还可以使用一些特别的技巧为任意元素应用动态伪类。例如，使用下述规则实现某种“突出显示”效果：

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

这个规则把 body 的任何后代元素处于悬停状态时的背景设为黄色。body 中的标题、段落、列表、表格、图像等一切元素都将变成黄色背景。此外，还可以改变悬停时元素的字体，为元素加上边框，或者浏览器能显示的其他效果。



虽然可以使用 :focus 以任何方式装饰元素，但是千万别删除获得焦点的元素的全部样式。区分当前获得焦点的元素对可访问性十分重要，尤其是使用键盘在网站或应用中导航时。

动态样式引起的问题

动态伪类有些耐人寻味的问题和怪异行为。例如，可以把已访问链接和未访问链接设为相同的字号，而在悬停时把字号增大，如图 2-37 所示。

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

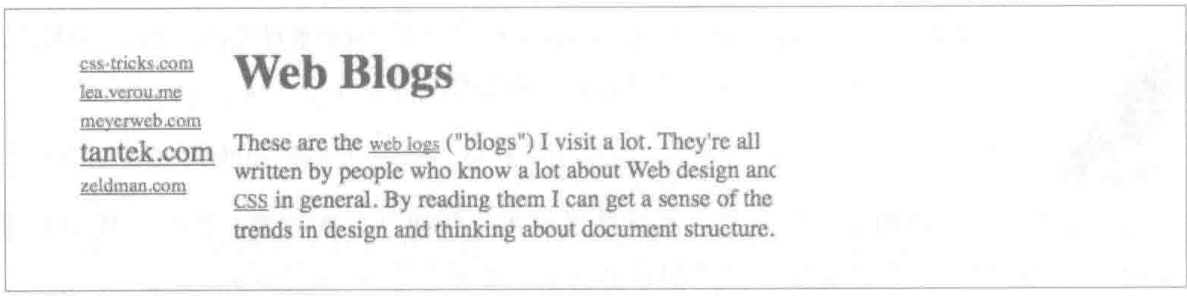


图 2-37：动态伪类改变布局

可以看出，鼠标指针悬停在那个锚记上时，用户代理增大了它的字号。因为还有 :active 伪类，所有用户在触摸屏上轻点时也会出现这种情况。支持这一行为的用户代理在锚记处于悬停状态时必须重新绘制文档，这可能导致那个链接后面的内容重排。

2.6.4 UI 状态伪类

与动态伪类紧密相关的是用户界面（user-interface，UI）状态伪类，简要说明见表 2-4。这些伪类根据用户界面元素（例如复选框）的当前状态应用样式。

表 2-4：UI 状态伪类

伪类	说明
:enabled	指代启用的用户界面元素（例如表单元素），即接受输入的元素
:disabled	指代禁用的用户界面元素（例如表单元素），即不接受输入的元素
:checked	指代由用户或文档默认选中的单选按钮或复选框
:indeterminate	指代既未选中也没有未选中的单选按钮或复选框；这个状态只能由 DOM 脚本设定，不能由用户设定
:default	指代默认选中的单选按钮、复选框或选项
:valid	指代满足所有数据有效性语义的输入框

表 2-4：UI 状态伪类（续）

伪类	说明
:invalid	指代不满足所有数据有效性语义的输入框
:in-range	指代输入的值在最小值和最大值之间的输入框
:out-of-range	指代输入的值小于控件允许的最小值或大于控件允许的最大值的输入框
:required	指代必须输入值的输入框
:optional	指代无需一定输入值的输入框
:read-write	指代可由用户编辑的输入框
:read-only	指代不能由用户编辑的输入框

虽然 UI 元素的状态能被用户操作改变，例如用户勾选或不选一个复选框，但是 UI 状态伪类不是单纯动态的，因为它们还受文档结构或 DOM 脚本的影响。



你可能觉得 :focus 属于这一节，而不是前一节。可是，Selectors Level 3 规范把 :focus 与 :hover 和 :active 归为一组。这可能是因为 CSS2 就是这样分组的，而且那时没有 UI 状态伪类。然而，更重要的原因是，非 UI 元素（例如标题或段落）也可以获得焦点，比如说能自动阅读的浏览器读到某个元素时。就这一点便足以把它与 UI 状态伪类分开。

启用和禁用的 UI 元素

在 DOM 脚本和 HTML5 的支持下，我们可以把一个用户界面元素（或者一组用户界面元素）标记为禁用的。禁用的元素也能显示出来，但是无法选择、激活或与用户交互。若想把元素设为禁用的，可以使用 DOM 脚本，也可以在 HTML5 元素的标记中添加 disabled 属性。

未禁用的元素显然是启用的。这两个状态可以使用 :enabled 和 :disabled 伪类装饰。常见的做法是装饰禁用的元素，启用的元素不做修饰，不过这两个伪类都有用处，如图 2-38 所示。

```
:enabled {font-weight: bold;}
:disabled {opacity: 0.5;}
```

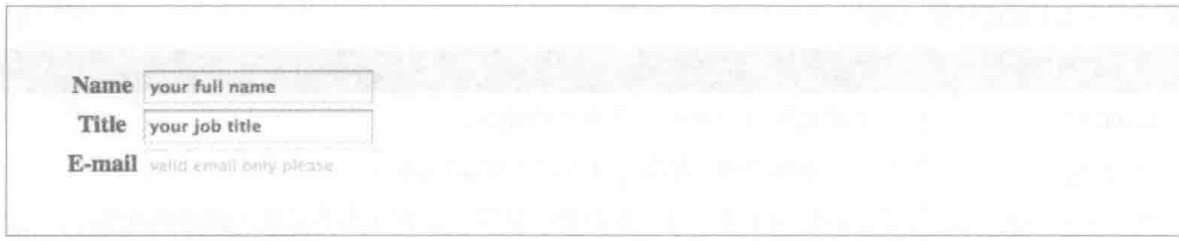


图 2-38：装饰启用和禁用的 UI 元素

选择状态

除了启用和禁用之外，某些 UI 元素还可以选中或不选，HTML 中的复选框和单选按钮就是这种。Selectors Level 3 为这种状态提供了 :checked 伪类，但是不知为何，没有 :unchecked 伪类。此外，还有一个 :indeterminate 伪类，它匹配的 UI 元素是可选择但是既未选中也没不选。这些状态如图 2-39 所示。

```
:checked {background: silver;}
:indeterminate {border: red;}
```

此外，可以使用否定伪类（稍后介绍）选择未被选中的复选框：input[type="checkbox]:not(:checked)。只有单选按钮和复选框才能被选中。其他元素以及这两个元素，如果未被选中，使用 :not(:checked) 选择。



图 2-39：装饰选中和处于不确定状态的 UI 元素

可选元素默认是未被选中的，不过 HTML 文档的编写人员可以通过元素标记中的 checked 属性切换状态，或者使用 DOM 脚本改变元素的选择状态。

除了这两个状态之外，还有一个状态——“不确定”。截至 2017 年年末，这个状态只能由 DOM 脚本或用户代理设定，在标记中做不到这一点。不确定状态存在的目的是从视觉上提示用户需要选中（或不选）某个元素。不过要注意，这只是视觉上的效果，对 UI 元素底层的状态没有影响。元素的底层状态只能是选中或未选中，这取决于文档标记和 DOM 脚本。

前面的示例虽然展示了如何装饰单选按钮，但是通过 CSS 装饰单选按钮和复选框能实现的效果十分有限。然而，使用这些伪类能做的事情却是无限的。例如，可以结合 :checked 和紧邻同胞连结符装饰复选框和单选按钮的标注（label）：

```
input[type="checkbox"]:checked + label {
  color: red;
  font-style: italic;
}

<input id="chbx" type="checkbox"> <label for="chbx">I am a label</label>
```

默认选项伪类

:default 伪类匹配一组相似元素中取默认值的 UI 元素。这个伪类通常用于上下文菜单选项、按钮和选择列表（目录）。如果有几个同名的单选按钮，最初选中的那个单选按钮匹配 :default，即使用户改变了 UI，最初选中的单选按钮已经不匹配 :checked。页面加载时选中的复选框匹配 :default。select 元素中最初选中的一个或多个 option 匹配 :default。:default 伪类还能匹配按钮和菜单选项。

```
[type="checkbox"]:default + label { font-style: italic; }

<input type="checkbox" id="chbx" checked name="foo" value="bar">
<label for="chbx">This was checked on page load</label>
```

可选性伪类

:required 伪类匹配必填的表单控件，这一要求由 required 属性（HTML5）指定。:optional 伪类匹配没有 required 属性的表单控件，或者 required 属性的值为 false 的控件。

若想提交表单，表单中匹配 :required 的元素必须有值，匹配 :optional 的元素可有值也可以没有值。例如：

```
input:required { border: 1px solid #f00;}
input:optional { border: 1px solid #ccc;}

<input type="email" placeholder="enter an email address" required>
<input type="email" placeholder="optional email address">
<input type="email" placeholder="optional email address" required="false">
```

第一个电子邮件地址输入框匹配 :required 伪类，因为它有 required 属性。第二个输入框是可选的，因此匹配 :optional 伪类。第三个输入框也是可选的，因为虽然有 required 属性，但值是 false。

除了伪类，还可以使用属性选择符。下述选择符与前面的等效：

```
input[required] { border: 1px solid #f00;}
input:not([required]) { border: 1px solid #ccc;}
```

除了表单输入框之外，其他元素既不能是必填的，也不能是可选的。

有效性伪类

:valid 伪类表示用户输入的值满足全部数据验证条件，而 :invalid 伪类表示用户输入的值不满足全部数据验证条件。

:valid 和 :invalid 两个有效性伪类只适用于能检查数据有效性的元素。因此，div 元素绝不可能匹配它们中的任何一个，而 input 元素可能匹配其中一个，这取决于用户界面的当前状态。

下面的示例为获得焦点的电子邮件地址输入框设定背景图，一个在输入的地址无效时显示，一个在输入的地址有效时显示，如图 2-40 所示。

```
input[type="email"]:focus {
  background-position: 100% 50%;
  background-repeat: no-repeat;
}
input[type="email"]:focus:invalid {
  background-image: url(warning.jpg);
}
input[type="email"]:focus:valid {
  background-image: url(checkmark.jpg);
}

<input type="email">
```



图 2-40：装饰有效和无效的 UI 元素



这两个伪类起不起作用取决于用户代理会不会向样式系统报告验证状态，因此某些情况下的效果可能跟预期不符。例如，在 2017 年年末，在多个用户代理中，未输入值的电子邮件地址输入框匹配 :valid，而空值肯定不是有效的电子邮件地址。在验证逻辑改进之前，使用有效性伪类时一定要小心。

范围伪类

范围伪类有两个，:in-range 和 :out-of-range。前者表示用户输入的值在 HTML5 的 min 和 max 属性设定的最小值和最大值范围之内，而后者表示用户输入的值小于控件接受的最小值或大于最大值。

例如，下面几个规则分别装饰一个接受 0~1000 之间数字的输入框的不同状态：

```
input[type="number"]:focus {
  background-position: 100% 50%;
  background-repeat: no-repeat;
}
input[type="number"]:focus:out-of-range {
  background-image: url(warning.jpg);
}
input[type="number"]:focus:in-range {
  background-image: url(checkmark.jpg);
}

<input id="nickels" type="number" min="0" max="1000" />
```

:in-range 和 :out-range 伪类只适用于设定了范围的元素。没有范围限制的元素，例如链接或 tel 类型的输入框，不能被它们中的任何一个匹配。

HTML5 还有一个 step 属性。如果一个值由于不匹配步进值而无效，但仍然在 min 和 max 设定的值之间（或等于两个极值），那么所在的元素将匹配 :invalid，同时还匹配 :in-range。也就是说，在范围内的值也可能是无效的。

因此，在下述示例中，输入框中的值将是红色加粗的，因为 23 在范围内，但是不能被 10 整除：

```
input[type="number"]:invalid {color: red;}
input[type="number"]:in-range {font-weight: bold;}

<input id="by-tens" type="number" min="0" max="1000" step="10" value="23" />
```

可变性伪类

可变性伪类有 :read-write 和 :read-only 两个，前者表示输入框可由用户编辑，而后者匹配不能编辑的输入框。只有能被用户编辑的元素才匹配 :read-write。

例如，在 HTML 中，未禁用的非只读 input 元素，以及设定了 contenteditable 属性的元素匹配 :read-write。其他所有元素匹配 :read-only。

默认情况下，下面两个规则都不匹配，因为 textarea 元素是可读可写的，而 pre 元素是只读的。

```
textarea:read-only { opacity: 0.75;}
pre:read-write:hover {border: 1px dashed green;}
```

然而，却匹配下述元素：

```
<textarea disabled></textarea>
<pre contenteditable>Type your own code!</pre>
```

因为 textarea 设定了 disabled 属性，变成只读的了，所以能应用第一个规则。类似地，这里的 pre 设定了 contenteditable 属性，现在是可读可写的元素，因此匹配第二个规则。

2.6.5 :target 伪类

URL 中有个片段标识符（fragment identifier），它所指向的文档片段（在 CSS 中）称为目标（target）。URL 片段标识符指向的目标元素可以使用 :target 伪类特别装饰。

即便不知道“片段标识符”这个术语，你肯定也见过。比如下面这个 URL：

```
http://www.w3.org/TR/css3-selectors/#target-pseudo
```

这个 URL 中的 target-pseudo 部分就是片段标识符，由 # 符号标记。如果对应的页面（*http://www.w3.org/TR/css3-selectors/*）中有 ID 为 target-pseudo 的元素，那个元素就是片段标识符的目标。

借助 :target 伪类，我们可以突出显示文档中的任何目标元素，或者为不同的目标元素定义不同的样式，例如作为目标的标题使用一个样式，作为目标的表格使用一个样式等。图 2-41 是 :target 伪类的实际效果。

```
*:target {border-left: 5px solid gray; background: yellow url(target.png)
    top right no-repeat;}
```

:target 伪类定义的样式在两种情况下不会应用：

1. 页面的 URL 中没有片段标识符。
2. 页面的 URL 中有片段标识符，但是文档中没有与之匹配的元素。

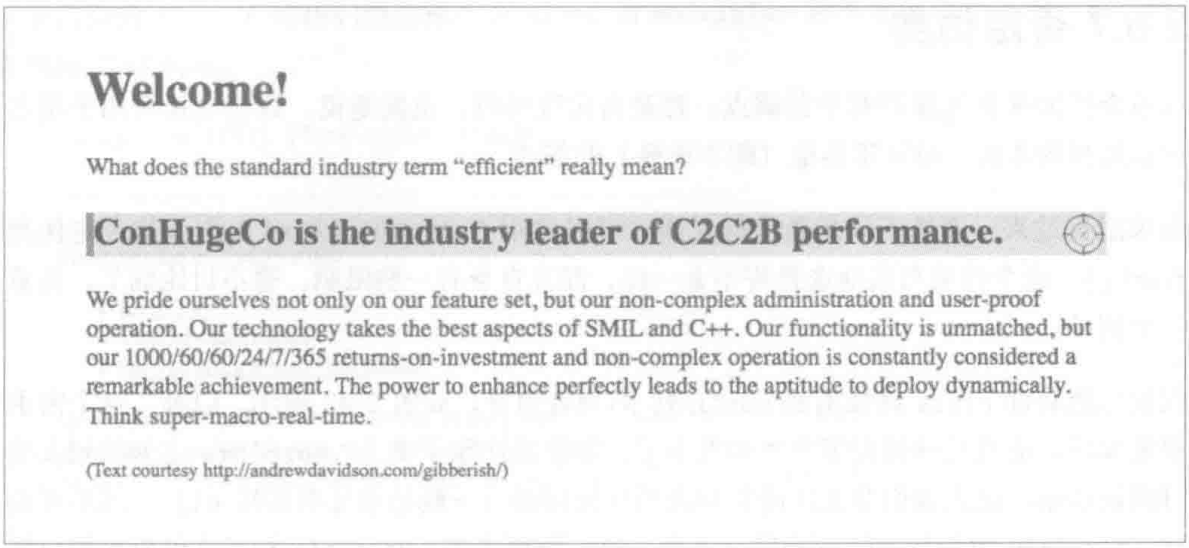


图 2-41：装饰一个片段标识符的目标

不过，更有趣的问题是，如果一个文档中有多个元素与片段标识符匹配怎么办？例如，文档编写人员失误，在文档中放了三个 `<div id="target-pseudo">`。

简单来说，CSS 无需为此提供解决方案，因为 CSS 所做的只是装饰目标。不管浏览器选择三个元素中的某一个，还是同等对待三个元素，`:target` 样式都会应用到有效的目标上。

2.6.6 :lang 伪类

如果想根据文本使用的语言选择元素，可以使用 `:lang()` 伪类。在匹配方式上，:lang() 伪类与 `|=` 属性选择符类似。假设想让使用法语编写的元素倾斜显示，可以编写下述规则中的任何一个：

```
*:lang(fr) {font-style: italic;}
*[lang|="fr"] {font-style: italic;}
```

伪类选择符与属性选择符之间的主要区别是语言信息有多个来源，有时可能来自元素自身之外。对属性选择符来说，元素自身必须有 `lang` 属性才能匹配。而 `:lang` 伪类能匹配设定了语言的元素的后代。Selectors Level 3 是这样规定的：

在 HTML 中，语言可以通过 `lang` 属性判断，也可以通过 `meta` 元素和协议（例如 HTTP 首部）判断。XML 使用 `xml:lang` 属性，此外还可能有文档语言专用的方法。

`:lang` 伪类可以使用各种信息，而 `|=` 属性选择符只能用于标记中有 `lang` 属性的元素。因此，伪类比属性选择符更可靠，多数情况下是装饰特定语言的理想之选。

2.6.7 否定伪类

目前介绍的所有选择符有个共同点：都是肯定选择符。也就是说，这些选择符用于指定应该选择的东西，排除不匹配（即不选择）的东西。

如果想反过来，选择不满足条件的元素，可以使用 Selectors Level 3 引入的否定伪类 `:not()`。这个伪类与其他选择符不太一样，而且自身有一些限制。暂不讨论细节，先看一个例子。

假设你想装饰 `class` 属性不是 `moreinfo` 的列表项目，如图 2-42 所示。以前，这个需求很难实现，而且某些情况下根本实现不了。如果想让除了类为 `.moreinfo` 之外的列表项目倾斜显示，过去我们要先让所有列表项目都倾斜（一般是通过类选择 `ul`），然后再通过 `.moreinfo` 类把特定的列表项目还原。这么做要确保 `.moreinfo` 样式在源码中的位置靠后，而且要具有相同或更高的特指度。现在，我们可以像这样声明样式：

```
li:not(.moreinfo) {font-style: italic;}
```

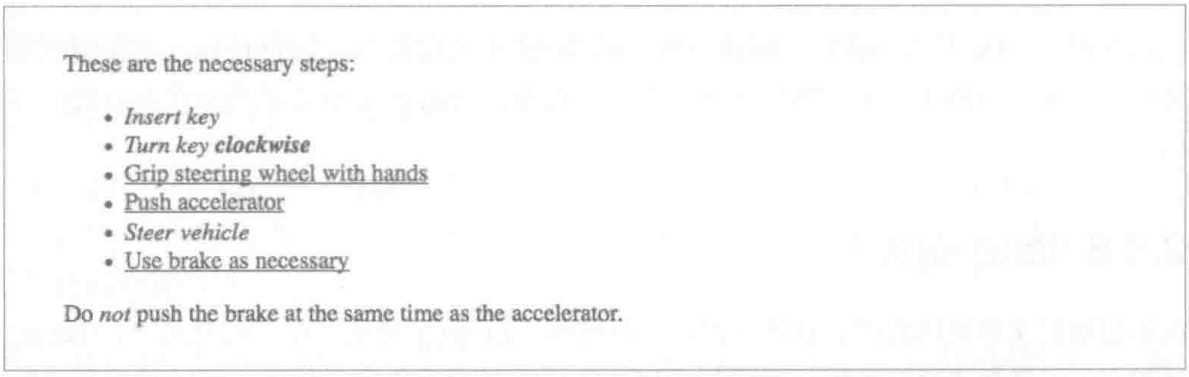


图 2-42：装饰没有指定类的列表项目

`:not()` 伪类依附在元素上，括号中是简单的选择符。根据 W3C 的定义，简单的选择符指：

一个类型选择符、通用选择符、属性选择符、类选择符、ID 选择符或伪类。

基本上，简单选择符是指没有祖辈 - 后代关系的选择符。

注意定义中的“或”，它的意思是 `:not()` 伪类中只能使用其中一个选择符。不能使用群组选择符，也不能使用连结符，因此不能使用后代选择符，因为后代选择符中分隔元素的空格是连结符。这些限制在未来可能撤销（极有可能），不过就算有这些限制，我们仍然可以通过它做很多事情。

再看前面的示例。假设我们想选择所有 class 为 moreinfo，但不是列表项目的元素。结果如图 2-43 所示。

```
.moreinfo:not(li) {font-style: italic;}
```

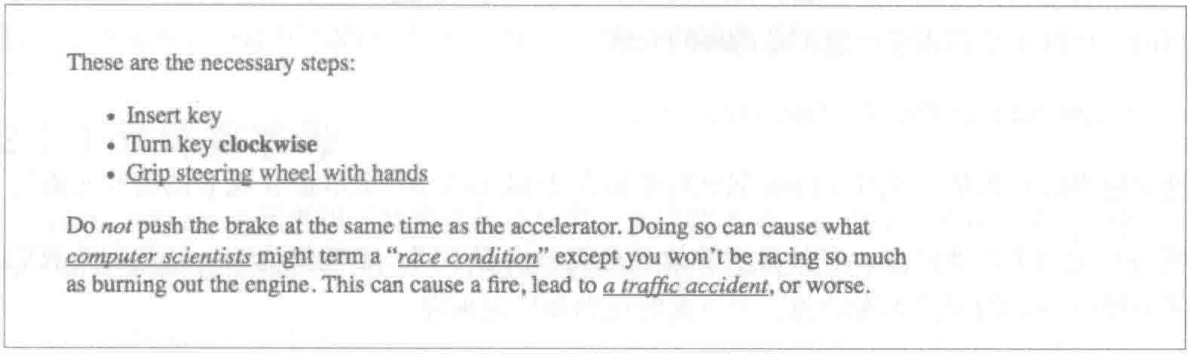


图 2-43：装饰具有特定类，但不是列表项目的元素

用人类语言描述，这个选择符的意思是，“选择的元素，其 class 属性中包含 moreinfo 这个词，但不是 li 元素”。类似地，li:not(.moreinfo) 的意思是，“选择 li 元素，但不包括 class 属性中包含 moreinfo 这个词的 li 元素”。

严格来说，:not() 伪类的括号中可以使用通用选择符，但这么做意义不大。毕竟，p:not(*) 的意思是“选择不是元素的 p 元素”。试想，怎么可能存在不是元素的元素。p:not(p) 与之类似，也选择不了任何元素。此外，还有 p:not(div) 这样的选择符，即选择不是 div 元素的 p 元素，这相当于选择所有 p 元素。可见，没有什么缘由这样做。

否定选择符可在复杂选择符的任何位置使用。因此，若想选择不是 section 元素子代的所有表格，可以使用 *:not(section) > table。类似地，若想选择不在表头中的表头单元格，可以使用 *:not(thead) > tr > th，效果如图 2-44 所示。



State	Post	Capital	State Bird
Alabama	AL	Montgomery	Yellowhammer
Alaska	AK	Juneau	Willow Ptarmigan
Arizona	AZ	Phoenix	Cactus Wren
Arkansas	AR	Little Rock	Mockingbird
California	CA	Sacramento	California Quail
Colorado	CO	Denver	Lark Bunting
Connecticut	CT	Hartford	American Robin
Delaware	DE	Dover	Blue Hen Chicken
Florida	FL	Tallahassee	Northern Mockingbird
Georgia	GA	Atlanta	Brown Thrasher
State	Post	Capital	State Bird

图 2-44：装饰表头之外的表头单元格

否定伪类不能嵌套，因此 p:not(:not(p)) 是无效的，将被忽略。逻辑上，这与直接使用 p 没有区别，所以没必要这么做。此外，在括号中不能引用伪元素（稍后说明原因），因为伪元素不是简单选择符。

不过，否定伪类可以串在一起，作用相当于“也不是”。例如，你可能想选择 class 为 link，但既不是列表项目也不是段落的元素：

```
*.link:not(li):not(p) {font-style: italic;}
```

这个规则的意思是，“选择 class 属性值中包含 link 这个词，但不是 li 或 p 的所有元素”。

使用否定伪类时要注意，有时真实的效果可能与设想不同。多数情况下，这是因为我们不习惯以否定的方式思考问题。对下面的规则和标记来说：

```
div:not(.one) p {font-weight: normal;}
div.one p {font-weight: bold;}

<div class="one">
   <div class="two">
      <p>I'm a paragraph!</p>
   </div>
</div>
```

上述段落将显示为粗体，而不是常规字重。这是因为两个规则都匹配：p 元素是 class 属性中不包含 one 这个词的 div 元素（ ）的后代，而且还是 class 元素中包含 one 这个词的 div 元素的后代。两个规则都匹配，因此两个规则都应用。因为二者有冲突，为了解决冲突要用到层叠规则，而结果是第二个样式规则胜出。标记的结构，即 div.two 比 div.one 离那个段落更近，在这里无关紧要。

2.7 伪元素选择符

伪元素与伪类很像，为了实现特定的效果，它在文档中插入虚构的元素。CSS2 定义了四个基本的伪元素，分别用于装饰元素的首字母、首行，以及创建和装饰“前置”和“后置”内容。CSS2 之后的规范又定义了其他伪元素（例如 ::marker），我们将在相关的章节中探讨。这一节介绍 CSS2 定义的那四个，因为它们由来已久，借此机会还可以讨论伪元素的行为。

伪类使用一个冒号，而伪元素使用一对冒号，例如 ::first-line。这么做是为了把伪元素与伪类区分开。一开始并不是这样的，在 CSS2 中，这两种选择符都使用一个冒号。因此，为了向后兼容，浏览器也接受使用单个冒号的伪元素选择符。但是，不要因为这

样就懈怠。为了确保你编写的 CSS 在未来还能继续使用，应该使用正确的冒号个数，毕竟我们无法预知浏览器什么时候不再接受单个冒号的伪元素选择符。

注意，所有伪元素只能出现在选择符的最后。p::first-line em 是无效的，因为伪元素在选择符的主词前面（主词是选择符中的最后一个元素）。这也表明一个选择符中只能有一个伪元素，不过在 CSS 以后的版本中可能会取消这一限制。

2.7.1 装饰首字母

::first-letter 伪元素用于装饰任何非行内元素的首字母，或者开头的标点符号和首字母（如果文本以标点符号开头）。下述规则把每一段的首字母设为红色：

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

::first-letter 伪元素最常用于实现排版效果中的“首字母大写”或“首字母下沉”。你可以把每个 p 元素首字母的字号设为其余内容的两倍大，不过最好只应用到第一段的首字母上：

```
p:first-of-type::first-letter {font-size: 200%;}
```

这个规则的效果如图 2-45 所示。



图 2-45：::first-letter 伪元素的效果

这个规则其实相当于让用户代理装饰每个 p 元素中包围首字母的一个虚构元素。比如这样：

```
<p><p-first-letter>T</p-first-letter>his is a p element, with a styled first
    letter</h2>
```

::first-letter 样式只应用于上例中虚构的那个元素里的内容。<p-first-letter> 元素不会出现在文档的源码中，也不会出现在 DOM 树中，而是由用户代理动态构建，目的是把 ::first-letter 样式应用到相应的文本上。也就是说，<p-first-letter> 是个伪元素。注意，你无需添加任何新标签。用户代理会代为装饰首字母，就像把首字母放到一个元素中一样。

首字母指排版上的第一个字母单元（如果前面没有其他内容的话，例如图像）。规范中使用的词是“字母单元”，因为有些语言的字母由多个字符构成，例如古斯堪的纳维亚语的“oe”。首字母前面或后面的标点符号（即便有多个符号），包含在 ::first-letter 伪元素中。

2.7.2 装饰首行

类似地，::first-line 用于装饰元素的首行文本。例如，可以把文档中每个段落的首行字号变大，并显示为紫色：

```
p::first-line {
  font-size: 150%;
  color: purple;
}
```

在图 2-46 中，这个样式应用到每个段落中显示的第一行文本上。不管显示区域有多宽或多窄都是这样。如果首行只包括一段的前五个词，那就只有这五个词显示为紫色的大字。如果首行包含前 30 个词，那么前 30 个词将显示为紫色的大字。

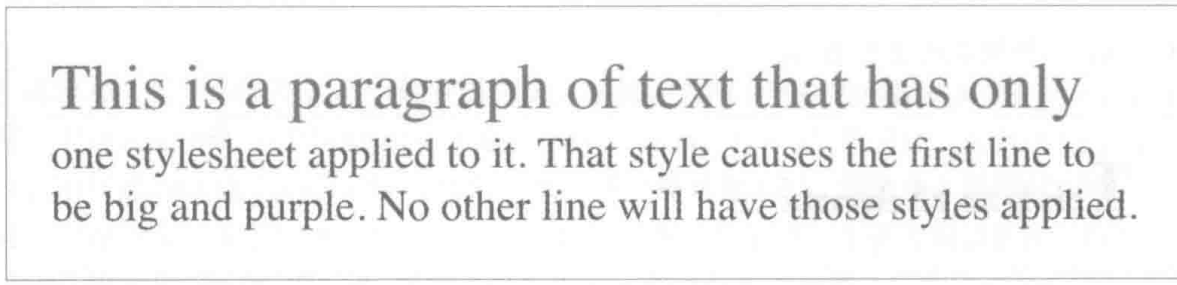


图 2-46：::first-line 伪元素的效果

从“This”到“only”应该显示为紫色的大字，因此用户代理会加上类似下面的虚构标记：

```
<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 big and purple. No other line will have those styles applied.
</p>
```

如果修改了首行里的文本，只剩前七个词了，那么虚构的 </p-first-line> 会向前移到“that”后面。如果用户增大或减小了字号，抑或拉宽或缩窄了浏览器窗口，导致文本的宽度有变，首行中的词数随之增多或减少，浏览器能自动调整，只把当前显示的首行里的文本显示为紫色大字。

首行的长度受诸多因素影响，例如字号、字符间距、父级容器的宽度等。对特定的标记和首行宽度而言，首行可能会在某个嵌套的元素中间结束。如果首行把嵌套的元素（例如 em 或超链接）破开了，::first-line 定义的样式只会应用到嵌套元素显示在首行里的那部分上。

2.7.3 对 ::first-letter 和 ::first-line 的限制

目前，::first-letter 和 ::first-line 伪元素只能应用到块级元素上，例如标题或段落，不能应用到行内元素上，例如超链接。::first-line 和 ::first-letter 样式中可以使用的 CSS 属性也有限制，见表 2-5。

表 2-5：允许伪元素使用的属性

::first-letter	::first-line
所有字体属性	所有字体属性
所有背景属性	所有背景属性
所有文本装饰属性	所有外边距属性
所有行内排版属性	所有内边距属性
所有行内布局属性	所有边框属性
所有边框属性	所有文本装饰属性
box-shadow	所有行内排版属性
color	color
opacity	opacity

2.7.4 装饰（或创建）前置和后置内容元素

假设根据排版效果，需要在 h2 元素前面加上两个银色的方括号：

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

使用 CSS 可以插入生成的内容（generated content），生成的这些内容可以直接使用 ::before 和 ::after 伪元素装饰。图 2-47 是一个例子。



图 2-47：在元素前面插入内容

这个伪元素用于插入并装饰生成的内容。若想把内容放在元素的后面，使用 ::after 伪元素。例如，可以在文档末尾加上结束语：

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

生成的内容是一个单独的话题，第 15 章将对此进行更为深入的介绍（还包括 ::before 和 ::after 的详细说明）。

2.8 小结

创作人员根据文档的语言选择适当的选择符可以创建丰富的 CSS 规则。规则可以应用于少数元素，也可以应用于大量类似的元素。选择符和规则都可以组合在一起，这样能让样式表变得紧凑、灵活，而且文件体积更小，下载速度更快。

用户代理通常必须正确处理选择符，否则极有可能导致 CSS 样式完全不起作用。另一方面，选择符也要编写正确，不然可能导致用户代理无法正确应用样式。若想正确理解选择符以及如何组合选择符，需要牢固掌握选择符与文档结构的关系，并了解继承和层叠等机制在应用样式上所起的作用。

第 3 章

特指度和层叠

第 2 章介绍了如何利用文档结构和 CSS 选择符为元素应用各种丰富的样式。每个有效的文档都对应一个树状结构，了解这一点之后便能根据元素的祖辈、属性、同胞元素等创建选择符选择元素。有了树状结构，选择符才能起作用，这也是 CSS 另一个重要方面（继承）的核心。

继承（inheritance）是指把一个元素的某些属性值传给其后代的机制。确定应该把哪些值应用到元素上时，用户代理不仅要考虑继承，还要考虑声明的特指度（specificity），以及声明的来源。这个过程称为层叠（cascade）。本章将探讨特指度、继承和层叠之间的关系，后两个概念之间的区别可以像这样简单理解：确定 `h1 {color: red; color: blue;}` 的结果是层叠，把 `h1` 中的 `span` 显示为蓝色是继承。

不管怎样，无论问题看上去多抽象、多难懂，都要不断努力！努力终有回报。

3.1 特指度

由第 2 章得知，我们可以使用多种不同的方法选择元素。实际上，同一个元素可能会被两个或多个规则选择，而且每个规则的选择符不尽相同。以下面三对规则为例，假设每一对规则匹配相同的元素：

```
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;}
```

每对规则中只有一个能胜出，因为匹配的元素只能显示为其中一个颜色。那么我们如何知道哪个规则胜出呢？

答案隐藏在每个选择符的特指度中。用户代理会计算每个规则中选择符的特指度，然后将其依附到规则中的每个声明上。如果两个或多个属性声明有冲突，特指度最高的声明胜出。



冲突的解决机制可不是这么简单的。所有样式冲突（包括特指度）都由层叠处理，3.3节将专门讨论这个问题。

选择符的特指度由选择符本身的组成部分决定。一个特指度值由四部分构成，例如0,0,0,0。选择符的特指度通过下述规则确定：

- 选择符中的每个 ID 属性值加 0,1,0,0。
- 选择符中的每个类属性值、属性选择或伪类加 0,0,1,0。
- 选择符中的每个元素和伪元素加 0,0,0,1。伪类到底有没有特指度在 CSS2 中表述的有些自相矛盾，不过 CSS2.1 明确指出，伪元素有特指度。
- 连结符和通用选择符不增加特指度。

例如，下面给出几个规则中选择符的特指度：

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

假如一个 em 元素能被这里的第二个和第五个规则匹配，那么它将显示为红褐色，因为第五个规则的特指度大于第二个规则的特指度。

现在做一个练习，计算本节开头那几对规则的特指度：

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

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

我已经指出每一对中的胜者，即特指度较高的规则。注意特指度的比较方式。在第二对中，h2.grape 选择符胜出，因为它多一个 1：0,0,1,1 大于 0,0,0,1。在第三对中，后一个规则胜出，因为 0,1,0,1 大于 0,0,1,7。特指度值 0,0,1,0 其实比 0,0,0,13 大。

这是因为特指度值是从左向右比较的。特指度 1,0,0,0 比所有以 0 开头的特指度大，不管后面的数有多大。同样，0,1,0,1 大于 0,0,1,7，因为前者位于第二位的 1 比后者位于第二位的 0 大。

3.1.1 声明和特指度

选择符的特指度确定之后，其值将赋予关联的每个声明。对下述规则来说：

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

为了计算特指度，用户代理必须把规则“打散”成单独的规则。因此，上述规则将变成：

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

二者的特指度都是 0,0,0,1，赋予每个声明的值就是它。群组选择符也有这样的打散过程。对下述规则来说：

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

在用户代理看来是这样的：

```
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 */
```

如果多个规则匹配同一个元素，而且部分声明之间有冲突，特指度就发挥作用了。例如，对下面的规则来说：

```
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 */
```

应用到下述标记后渲染得到的结果如图 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>
```

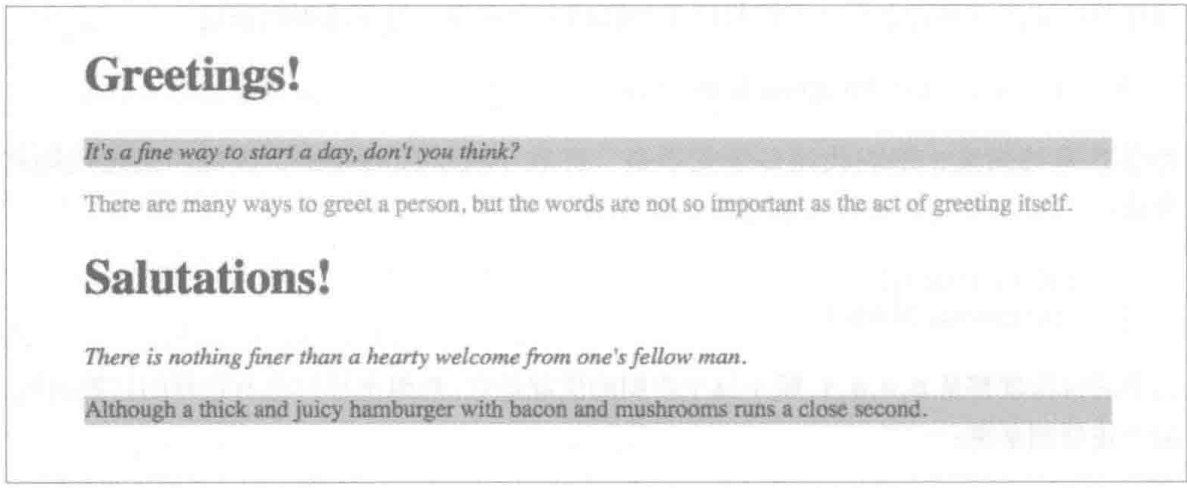


图 3-1：不同规则对文档的影响

在任何情况下，用户代理都会确定哪些规则与元素匹配，然后找出所有相关的声明，计算各自的特指度，判断哪些规则胜出，再把胜出的规则应用到元素上，得到装饰后的结果。每个元素、选择符和声明都要经历这一系列操作。幸好，用户代理能自动处理一切。这个行为是层叠（本章后文讨论）的重要部分。

3.1.2 通用选择符的特指度

通用选择符不增加特指度。也就是说，它的特指度为 0,0,0,0，这与没有特指度是不同的（参见 3.2 节）。因此，对下面两个规则来说，div 元素中的段落将显示为黑色，其他元素则显示为灰色：

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

你可能猜到了，包含通用选择符的选择符，它的特指度不因有通用选择符的存在而改变。下面两个选择符的特指度相等：

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

连结符不同，它根本没有特指度，即连零都没有。因此，连结符对选择符的总特指度没有影响。

3.1.3 ID 和属性选择符的特指度

ID 选择符和选择 id 属性的属性选择符之间在特指度上是有区别的，这一点一定要注意。来看前述示例中的第三对规则：

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

第二个规则中的 ID 选择符（#answer）为选择符的总特指度贡献 0,1,0,0。然而，第一个规则中的属性选择符（[id="totals"]）为总特指度贡献 0,0,1,0。因此，对下述规则来说，id 为 meadow 的元素将显示为绿色：

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

3.1.4 行内样式的特指度

目前见到的特指度都以零开头，因此你可能在想，那一位为什么要存在呢？存在必定有用。那一位是为行内样式声明保留的，行内样式声明的特指度比其他声明都高。对下面的规则和标记片段来说：

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

即便那个规则会应用到 h1 元素上，但是 h1 元素中的文本仍将显示为绿色。这是因为行内声明的特指度为 1,0,0,0。

这意味着，就算是带 id 属性的元素匹配一个规则，也要让位于行内样式声明。修改前面的示例，加上 id：

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

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

因为行内声明的特指度高，所以 h1 元素中的文本仍是绿色的。

3.1.5 重要性

有时某个声明可能非常重要，超过其他所有声明。CSS 称之为重要声明（important declaration，原因显而易见）。这种声明要在声明末尾的分号之前插入 !important，例如：

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

这里，颜色值 #333 使用 !important 标记，而背景色 white 没有。如果想把两个声明都标记为重要的，每个声明中都要插入 !important：

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

!important 的位置必须正确，否则声明将失效。!important 始终放在声明末尾的分号之前。对值为多个关键字的属性（例如 font）来说，!important 的位置尤其重要：

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

如果把 !important 放在 font 声明的其他位置，整个声明都将失效，而且整个样式都不会应用到元素上。



我发现有编程经验的人会本能地把这个句法理解为“不重要”。不管出于什么原因选择了感叹号（!），也不管其他语言为其赋予了怎样的含义，在 CSS 中感叹号的意思并非“不”。这只是巧合，习惯就好。

带有 !important 的声明对特指度没有影响，但是会与不重要的声明分开处理。其实，所有带 !important 的声明会放在一起，而特指度冲突就在这个范围内解决。同样，非

重要的声明作为一个整体，其中的冲突使用特指度解决。因此，重要声明和非重要声明冲突时，重要声明始终胜出。

下述规则和标记片段的结果如图 3-2 所示。

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

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



图 3-2：重要规则始终胜出



重要声明及其处理方式在 3.3 节深入讨论。

3.2 继承

特指度对理解声明是如何应用到文档上的很重要，此外，还有一个重要的概念，即继承。继承指某些样式不仅应用到所指的元素上，还应用到元素的后代上。例如，为 h1 元素设定的颜色还会应用到 h1 中的文本上，即使文本包含在子元素中。

```
h1 {color: gray;}

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

h1 中的常规文本和 em 中的文本都将显示为灰色，因为 em 元素从 h1 元素的样式中继承了 color 属性。如果后代元素无法继承属性值，em 中的文本将显示为黑色，而不是灰色，因此要分别为它们设定颜色。

继承对无序列表也适用。假设为 ul 元素应用 color: gray; 样式：

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

我们希望应用到 ul 上的这个样式也应用到其中的列表项目，以及列表项目的内容上。得益于继承，事实的确如此，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：继承样式

通过文档的树状图更容易理解继承的工作方式。图 3-4 是一个十分简单的文档的树状图，文档中有两个列表：一个无序列表，一个有序列表。

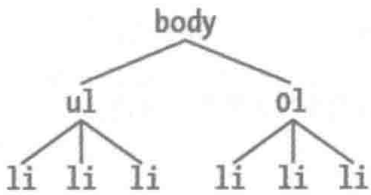


图 3-4：一个简单的树状图

把声明 `color: gray;` 应用到 `ul` 元素上后，那个元素将使用声明的样式渲染。这个值再沿着树状图向下传播到后代元素，直到没有后代为止。属性值绝不向上传播，即元素的样式绝不传给祖辈元素。



在 HTML 中，向上传播规则有个例外：应用到 body 元素上的背景样式会传给 html 元素。html 是文档的根元素，用于定义渲染文档的画布。这一例外仅发生在为 body 元素定义了背景而没有为 html 元素定义背景的情况下。

继承是 CSS 的根基之一，通常无需刻意考虑。不过，有几点要留意。

首先，很多属性是不继承的，这通常是为了避免得到意外的结果。例如，border 属性（设定元素的边框）就不继承。看一眼图 3-5 便能揭示个中缘由。如果继承边框，文档将变得杂乱无章，除非编写样式时付出额外的精力去掉继承的边框。

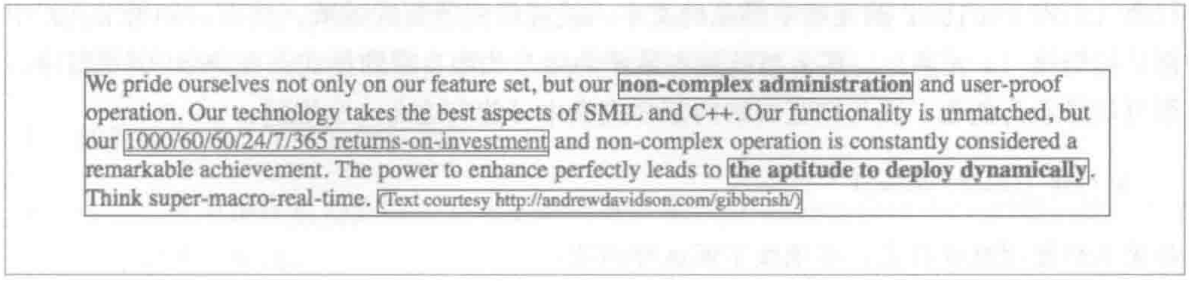


图 3-5：边框不继承的原因

此外，基于同样的原因，多数盒模型属性也不继承，包括外边距、内边距、背景和边框。试想，你肯定不想让段落中的所有链接都从父元素那里继承 30 像素的左外边距。

其次，继承的值没有特指度，连零都没有。这看起来似乎只是理论上的不同，但是等你了解到继承的值没有特指度将产生怎样的结果时，便会发现这种差距绝不能忽视。分析下述规则和标记片段，与图 3-6 所示的结果比较一下。

```
* {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>
```



图 3-6：零特指度战胜无特指度

因为通用选择符应用于全部元素，而且特指度为零，所以它声明的颜色 gray 击败继承的颜色 black（由于继承的值根本没有特指度）。因此，em 元素渲染为灰色，而不是黑色。

这个例子充分体现了滥用通用选择符的潜在危险。通用选择符能匹配任何元素，它往往会终结继承。这个问题有变通解决方法，但是通常最好从一开始就不滥用通用选择符。

继承的值没有特指度是很关键的一点，决不能忽略。例如，假设我们使用下面的样式表把“工具栏”中的所有文本显示为黑底白字：

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

只要 id 为 toolbar 的元素中都是纯文本，就能得到所要的效果。然而，如果工具栏中都是超链接（a 元素），那么超链接将显示为用户代理自带的样式。在 Web 浏览器中，很可能显示为蓝色，因为浏览器的内部样式表中可能有这么一个规则：

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

如果不想使用默认样式，必须像下面这样声明：

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

这里，有个规则直接选择工具栏中的 a 元素，得到的结果如图 3-7 所示。

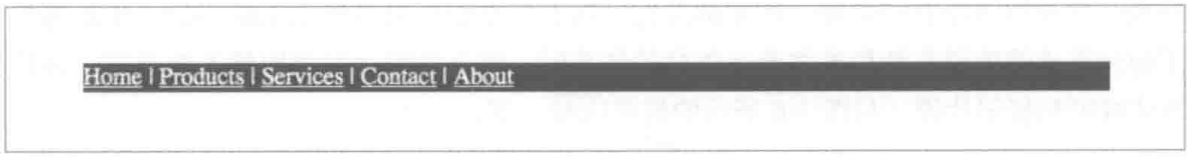


图 3-7：直接把样式应用到相关元素上

此外，使用前一章介绍的 inherit 也能得到相同的结果。把规则改成下面这样：

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

现在得到的结果跟图 3-7 一样，因为第二个规则的选择符有特指度，而且明确指定 color 的值继承而来。

3.3 层叠

目前，本章一直回避了一个十分重要的问题：如果两个特指度相等的规则应用到同一个元素上会发生什么？浏览器如何解决这样的冲突呢？例如，对下面的规则来说：

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

哪一个规则胜出？两个规则的特指度都是 0,0,0,1，因此它们的权重相同，都应该应用到元素上。但事实并非如此，因为元素的内容不可能既是红色的，又是蓝色的。那到底是什么颜色呢？

“层叠样式表”这个名称可以提供一点线索：CSS 采用层叠机制把样式组合在一起，即结合继承和特指度的一些规则。CSS 的层叠规则如下：

1. 找到匹配特定元素的所有规则。
2. 按显式权重排序应用到特定元素上的所有声明。以 !important 标记的规则比没有这一标记的权重高。
3. 按来源排序应用到特定元素上的所有声明。声明有三个来源：创作人员、读者和用户代理。正常情况下，创作人员编写的样式击败读者提供的样式；读者样式中以 !important 标记的声明比其他样式权重高，包括创作人员编写的样式中以 !important 标记的声明；创作人员和读者样式覆盖用户代理的默认样式。
4. 按特指度排序应用到特定元素上的所有声明。特指度高的声明具有较高的权重。
5. 按声明的前后位置排序应用到特定元素上的所有声明。样式表或文档中靠后的声明权重较高。导入的样式表中的声明放在当前样式表中所有声明的前面。

为了完全弄清这些规则，下面通过几个例子说明后四个层叠规则。



后面发布的某些 CSS 模块为第三条规则添加了更多的来源，例如动画和过渡的来源。这些来源在这里不讨论，将在后面相应的章节探讨。

3.3.1 按权重和来源排序

如果两个规则应用到同一个元素上，而其中一个以 !important 标记，那么有此标记的规则胜出：

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

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

虽然有个颜色由段落的 style 属性提供，但是 !important 标记的规则依然胜出，因此段落将显示为灰色。em 元素会继承灰色。

注意，如果 !important 在行内样式中，那么行内样式将胜出。因此，对下属规则和标记来说，段落（及其后代元素）将显示为黑色：

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

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

如果显式权重相同，就要考虑规则的来源。如果匹配元素的两个样式权重相同，而一个在创作人员编写的样式表中，另一个在读者提供的样式表中，那么元素将使用创作人员编写的样式表中的样式。例如，假设下面两个样式分别来自指定的位置：

```
p em {color: black;}    /* 创作人员编写的样式表 */

p em {color: yellow;}   /* 读者提供的样式表 */
```

此时，段落中的强调文本将显示为黑色，而不是黄色，因为同样的权重下，创作人员编写的样式战胜读者提供的样式。然而，如果两个规则都用 !important 标记，那情况就变了：

```
p em {color: black !important;}    /* 创作人员编写的样式表 */

p em {color: yellow !important;}   /* 读者提供的样式表 */
```

现在，段落中的强调文本将显示为黄色，而不是黑色。

有时，这里还牵涉到用户代理的默认样式（通常受用户偏好设置的影响）。在所有来源中，默认样式中的声明是影响力最低的。因此，如果创作人员为锚记定义了规则（例如声明颜色为 white），那么用户代理默认的样式将被覆盖。

综上，在声明的权重上，基本要考虑五个方面。下面按权重从高到低列出：

1. 读者提供的样式中以 !important 标记的声明。
2. 创作人员编写的样式中以 !important 标记的声明。
3. 创作人员编写的常规声明。
4. 读者提供的常规声明。
5. 用户代理的默认声明。

创作人员通常只需考虑前四点，因为创作人员编写的任何样式都会覆盖用户代理的默认样式。

3.3.2 按特指度排序

如果应用到一个元素上的声明有冲突，而且各声明的显式权重和来源相同，那么应该按特指度排序，特指度最高的声明胜出。例如：

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

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

对上面的规则来说，段落中的文本将显示为银色，如图 3-8 所示。为什么呢？因为 p#bright 的特指度是 0,1,0,1，而 p 的特指度是 0,0,0,1，虽然后一个规则在样式表中的位置靠后，但是前一个规则的特指度更高。



图 3-8：特指度较高的声明胜出

3.3.3 按前后位置排序

最后，如果两个规则的显式权重、来源和特指度都相同，那么在样式表中的位置靠后的规则胜出。回过头来看本节开头那个例子，文档的样式表中有下面两个规则：

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

此时，文档中所有 h1 元素的 color 值将取 blue，而不是 red。二者在显式权重和来源上是一样的，而且选择符的特指度也相等，因此靠后的声明胜出。

那么，如果来自不同样式表的规则有冲突了呢？例如，对下面的样式表来说：

```
@import url(basic.css);
h1 {color: blue;}
```

如果 h1 {color: red;} 在 basic.css 中，情况如何？ basic.css 中的全部内容相当于出现于 @import 所在的位置。因此，当前样式表中的规则出现在 @import 导入的样式之后。如果显式权重和特指度相同，当前样式表中的声明胜出。来看下面的规则：

```
p em {color: purple;}  /* 在导入的样式表中 */

p em {color: gray;}    /* 在当前样式表中 */
```

此时，第二个规则胜出，因为它是后声明的那个。

就是因为前后位置有影响，所以通常才推荐按照一定的顺序编写链接的样式。链接样式的推荐顺序是“link-visited-focus-hover-active”（LVFHA），如下所示：

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

读过本章的内容之后我们知道，这些选择符的特指度相等，都是 0,0,1,0。因为它们的显式权重、来源和特指度都一样，所以最后一个匹配的规则将胜出。单击或激活（例如使用键盘）未访问的链接时，匹配其中四个规则，:link、:focus、:hover 和 :active。因此，这四个规则中的最后一个胜出。对 LVFHA 顺序来说，:active 将胜出，这通常正是创作人员想要的。

假设我们忽略推荐的顺序，按照字母顺序排列链接样式的各个伪类，写成这样：

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

此时，任何链接都不会匹配 :hover、:focus 或 :active 样式，因为 :link 和 :visited 规则在它们三个后面。链接要么已访问，要么未访问，因此 :link 和 :visited 总是会把其他的规则覆盖掉。

再来看一个创作人员可能会使用的顺序。此时，只有未访问的链接有悬停样式，已访问的链接没有。不过，已访问和未访问的链接都有激活样式：

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

这种冲突只发生在所有状态都设定相同的属性之时。如果不同的状态设定不同的属性，那么前后位置就没关系了。对下面的情况来说，链接样式可以写为任何顺序，不管怎么写都能起作用：

```
a:link {font-weight: bold;}
a:visited {font-style: italic;}
a:focus {color: green;}
a:hover {color: red;}
a:active {background: yellow;}
```

你可能还发现了，:link 和 :visited 样式的顺序没关系。写成 LVFHA 或 VLFHA 都可以。

把伪类串在一起可以消除这一系列问题。下面几个规则可以写成任何顺序，不会出现谁覆盖谁的情况：

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

这几个规则分别应用于链接的不同状态，因此相互之间不冲突。因此，调换顺序之后对文档的装饰效果没有影响。后两个规则的特指度相同，但这无关紧要。悬停在未访问的链接上时，其样式不受针对悬停在已访问链接上的规则影响；反之亦然。如果加上激活状态，顺序就又变重要了。对下面的规则来说：

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

如果把激活状态的样式移到悬停状态的样式前面，二者都将被忽略。这里的原因是特指度有冲突。解决的方法是增加串联的伪类数量，如下所示：

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

这样做增加了选择符的特指度（都变成了 0,0,3,1），但是它们不再冲突了，因为选择的状态是互斥的。一个链接不可能既处于未访问、悬停的激活状态，又处于已访问、悬停的激活状态，因此只有其中一个规则将匹配，并应用对应的样式。

3.3.4 CSS 之外的表现提示

文档除了 CSS 之外可能包含表现提示（presentational hint），例如 font 元素。这种表现提示的特指度为 0，而且认为出现在创作人员编写的样式表的开头。表现提示将被创

作人员编写的样式或读者提供的样式覆盖，但是不会被用户代理的默认样式覆盖。CSS3把 CSS 外部的表现提示视作用户代理默认样式表的一部分，而且假定出现在默认样式表的最后（不过，写作本书时，规范没有这么表述）。

3.4 小结

对层叠样式表来说，“层叠”算是最基本的概念。在这个过程中，有冲突的声明按一定顺序排列，由此确定文档的最终表现。这个过程还涉及选择符和与其关联的声明的特指度，以及继承机制。

第 4 章

值和单位

本章探讨单位。单位可以影响颜色、距离和尺寸等一系列属性，可以帮助定义这些值。单位是 CSS 的重要基础，几乎一切值都离不开它。没有单位，你便无法声明一个图像的四周有 10 像素的空白，也不能声明标题的文本是多大。理解本章讨论的概念，有助于你快速学习和使用 CSS 的其他知识。

4.1 关键字、字符串和其他文本值

样式表中的一切都是文本，但是有些类型的值表示的是字符串本身，而不是数字或颜色等。表示字符串本身的值有 URL 和让人难以置信的图像。

4.1.1 关键字

有时，值用一个词表示，这叫关键字。none 是一个十分常见的关键字，它与 0（零）不同。若想移除 HTML 文档中链接的下划线，可以这样写：

```
a:link, a:visited {text-decoration: none;}
```

类似地，如果想为链接加上下划线，要使用关键字 underline。

接受关键字的属性，所取的关键字必须在那个属性允许使用的关键字范围之内。如果两个属性使用相同的关键字，在不同的属性中相同的关键字可能具有不一样的行为。例如，letter-spacing 属性的 normal 关键字与 font-style 属性的 normal 关键字就相差很大。

全局关键字

CSS3 定义了几个“全局”关键字，规范中的每个属性都能使用：inherit、initial 和 unset。

inherit 关键字 inherit 把元素某个属性的值设为与父元素同一属性的值一样。也就是说，这个关键字强制继承，即便是在通常情况下不继承时。很多时候无需这么做，因为多数属性会自动继承。尽管如此，inherit 仍有其用处。

例如，对下面的样式和标记来说：

```
#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 元素将有蓝色背景和白字，而链接将根据浏览器的偏好设置装饰。多数情况下，这里的链接将显示为蓝底蓝字，中间以白色的竖线分隔。

你可以单独编写一个规则，把这个“工具栏”中的链接设为白色，不过使用 inherit 更便利。只需在样式表中添加下述规则：

```
#toolbar a {color: inherit;}
```

这样，链接的 color 属性将使用继承的值，而不是用户代理的默认样式。通常，直接指定的样式覆盖继承的样式，但是 inherit 能把这种情况反过来。有时，这样做并不是一个好主意。例如，这里的样式可能与周围的文本融合太深，容易引起可访问性方面的问题，不过你要知道确实可以这样做。

使用 inherit 还能把通常情况下不从父元素继承的属性值强制拉过来。比如说，border 属性不会继承（这是正确的行为）。如果想让 span 继承父元素的边框，只需使用 span {border: inherit;}。不过，更多时候，我们可能只想让 span 的边框颜色与父元素一样。此时，使用 span {border-color: inherit;} 即可。

initial 关键字 initial 把属性的值设为预定义的初始值，相当于“重设”值。例如，font-weight 属性的默认值是 normal。因此，font-weight: initial 的作用与 font-weight: normal 一样。

这样做看起来有点傻，不过要知道，不是所有属性都有预定义的初始值。例如，color 属性的初始值“取决于用户代理”。这可不是让你原封不动输入的特殊关键字。它的意思是，color 属性的默认值取决于浏览器等的偏好设置。多数情况下，用户不会修改默认的文本颜色（黑色），但也有人会改，比如改为深灰色，或者亮红色。此时，color: initial; 的作用是，让浏览器把元素中文本的颜色设为默认的颜色。

unset 关键字 unset 是 inherit 和 initial 的通用替身。对继承的属性来说，unset 的作用与 inherit 一样；对不继承的属性来说，unset 的作用与 initial 一样。



截至 2017 年年末，initial、inherit 和 unset 这三个关键字 Opera Mini 都不支持。Internet Explorer 11 之前的版本也不支持。

这三个全局关键字在所有属性中都可以使用。有个特殊的属性只接受这几个全局关键字：all。

all	
取值	inherit | initial | unset
初始值	参见各属性

all 表示除 direction 和 unicode-bidi 之外的所有属性。因此，如果为一个元素声明 all: inherit，意思是除 direction 和 unicode-bidi 之外的所有属性都从父元素上继承值。对下面的规则和标记来说：

```
section {color: white; background: black; font-weight: bold;}
#example {all: inherit;}

<section>
    <div id="example">This is a div.</div>
</section>
```

你应该能猜到，div 元素会从 section 元素上继承 color、background 和 font-weight 属性的值。的确，不过除此之外的其他 CSS 属性（除了那两个）也从 section 元素上继承属性值。

如果这就是你想要的行为，那就好。但是，如果只想继承 section 元素的样式中列出的那几个属性值的话，CSS 就要这样写：

```
section {color: white; background: black; font-weight: bold;}
#example {color: inherit; background: inherit; font-weight: inherit;}
```

如果真正想要的是 all: unset，要相应修改样式表。



截至 2017 年年末，相关人员正在考虑另一个全局关键字，revert。它的作用是把属性值回退到其他来源中设定的值。例如，创作人员可能想让元素的所有属性值都不使用自己编写的样式表，而使用用户代理和用户提供的样式表。这个关键字正在商讨中，这里不做深入说明。



截至 2017 年年末，Opera Mini 和 Microsoft Edge 不支持 all。Edge 正在考虑支持。

4.1.2 字符串

字符串值指放在单引号或双引号内的任意字符序列，定义可取的值时以 <string> 表示。下面举两个简单的例子：

```
"I like to play with strings."
'Strings are fun to play with.'
```

注意，前后引号要一致，即首尾两个引号是同一种。如若不然，可能导致各种问题，因为以一种引号开头而以另一种引号结尾其实相当于字符串并未结束。这样可能意外导致后续规则变成字符串的一部分。

字符串中还可以有引号，只要不与外层的引号同样就行，此外也可以使用反斜线转义：

```
"I've always liked to play with strings."
'He said to me, "I like to play with strings."'
"It's been said that \"haste makes waste.\""
'There\'s never been a "string theory" that I\'ve liked.'
```

注意，字符串的定界符只能使用 ' 和 "。有时这叫“直引号”，因此字符串值不能以“弯”引号或“智能”引号开头或结尾。不过，这两种引号可以在字符串值中使用（见下面的示例），而且无需转义：

```
"It's been said that "haste makes waste.""
'There's never been a "string theory" that I've liked.'
```

为此，文档要使用 Unicode 编码，不过这就是推荐的编码（Unicode 标准的详情请访问 *http://www.unicode.org/standard/standard.html*）。

如果字符串值中有换行，可以转义换行符。CSS 会去掉换行符，就像从未换行一样。因此，下面两个字符串在 CSS 看来是一样的：

```
"This is the right place \
for a newline."
"This is the right place for a newline."
```

如果真想在字符串中插入一个换行符，在需要换行的地方使用 Unicode 字符 \A：

```
"This is a better place \Afor a newline."
```

4.1.3 URL

编写过网页的人对 URL（CSS2.1 称之为 URI）肯定不陌生。引用 URL（例如在导入外部样式表的 `@import` 语句中）的一般格式如下：

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

这是一个绝对 URL。“绝对”的意思是不管位于何处都能找到，因为这种 URL 在网络空间中定义了一个绝对位置。假设有个名为 *web.waffles.org* 的服务器。在这个服务器中有个名为 *pix* 的目录，里面有个名为 *waffle22.gif* 的图像。那么，这个图像的绝对 URL 是：

```
web.waffles.org/pix/waffle22.gif
```

不管在什么地方，是在 *web.waffles.org* 服务器上，还是在 *web.pancakes.com* 服务器上，这个 URL 都是有效的。

URL 的另一种类型是相对 URL，之所以这样命名是因为它的位置相对于所在的文档。如果指代一个相对位置，例如与网页在同一个目录中的某个文件，一般的格式为：

```
url(pathname)
```

此时，指代的文件（例如一个图像）必须与网页在同一个服务器中。下面举个例子。假设网页的地址是 *http://web.waffles.org/syrup.html*，我们想在这个网页中显示图像 *waffle22.gif*。此时，URL 可以这样写：

```
pix/waffle22.gif
```

这个路径是有效的，Web 浏览器知道要先找到文档的地址，然后在后面加上指定的相对 URL。这里，路径名 *pix/waffle22.gif* 将添加到 *http://web.waffles.org* 后面，得到 *http://web.waffles.org/pix/waffle22.gif*。使用相对 URL 的地方几乎都可以使用绝对 URL，使用哪种 URL 无关紧要，只要指代的地址是有效的就行。

在 CSS 中，相对 URL 相对于所在的样式表，而不是使用样式表的文档。例如，一个外部样式表中可能会导入其他样式表。如果使用相对 URL 导入那个样式表，URL 必须相对于当前样式表。

举个例子。假设 HTML 文档的地址是 *http://web.waffles.org/toppings/tips.html*，文档中有个 link 元素，链接样式表 *http://web.waffles.org/styles/basic.css*：

```
<link rel="stylesheet" type="text/css"
    href="http://web.waffles.org/styles/basic.css">
```

basic.css 文件中有个 `@import` 语句，导入另一个样式表：

```
@import url(special/toppings.css);
```

浏览器遇到这个 `@import` 语句时将在 *http://web.waffles.org/styles/special/toppings.css* 位置查找样式表，而不是 *http://web.waffles.org/toppings/special/toppings.css*。如果样式表在后一个位置上，*basic.css* 中的 `@import` 语句要写成下面两种方式中的一种：

```
@import url(http://web.waffles.org/toppings/special/toppings.css);

@import url(../special/toppings.css);
```

注意，`url` 和开始括号之间不能有空格：

```
body {background: url(http://www.pix.web/picture1.jpg);}   /* 正确 */
body {background: url  (images/picture2.jpg);}             /* 错误 */
```

如若不然，整个声明都无效，将被忽略。

4.1.4 图像

你可能猜到了，图像值引用的是图像。句法使用 `<image>` 表示。

就目前的支持情况来说，即地球上所有 CSS 引擎都能理解，`<image>` 值基本上就是 `<url>` 值。不过在更高级的用户代理中，`<image>` 表示下面三种值中的一个：

<url>

指向外部资源的 URL 标识符。这里指图像的 URL。

<image-set>

一系列图像（神奇吧），根据值中的条件选择。例如，image-set() 可以指定用于桌面布局的大图，以及用于移动布局的小图（尺寸和体积都小）。行为接近 picture 元素的 srcset 属性。截至 2016 年年末，所有浏览器都支持 srcset，但是只有 Safari、Chrome 和桌面版 Opera 支持 image-set。

<gradient>

线性渐变或径向渐变图像，可以是单个的，也可以重复。渐变是十分复杂的话题，在第 9 章详述。

4.1.5 标识符

有些属性接受标识符值，即用户定义的某种标识符，最常见的有生成的列表序号。在取值句法中使用 <identifier> 表示。标识符本身是词，而且区分大小写。例如，在 CSS 看来，myID 和 MyID 是完全不同的两个标识符。如果属性既接受标识符，也接受一个或多个关键字，一定要注意，不要把标识符定义地跟有效的关键字一样。

4.2 数字和百分数

这两种值的特别之处在于，它们是很多其他类型值的基础。例如，字号可以使用一个数字和 em 标识符（后文说明）定义。不过，能使用哪些数字呢？现在明确数字的类型，有助于后文表述清晰。

4.2.1 整数

整数值很好理解，由一到多个数组成，前面还可以有 + 或 - 号，分别表示正数和负数。就这么简单。在取值句法中，整数值用 *<integer>* 表示。13、-42、712 和 1066 都是整数。

取值范围外的整数值默认是无效的，会导致整个声明被忽略。然而，按照某些属性的定义，取值范围外的整数会被设为与所用值最接近的数，这种行为称为 *clamping*。如果未明确限制取值范围（例如 z-index 属性），用户代理必须接受 ±1 073 741 824（$\pm 2^{30}$）之内的数。

4.2.2 数字

数字值是 <*integer*> 或实数，后者指整数后跟一个点号，再跟着一些整数。此外，前面可以加上 + 或 −，表示正数或负数。在取值句法中，数字值用 <*number*> 表示。2.7183、−3.1416 和 6.2832 都是数字。

<*number*> 可以是 <*integer*>，也可以是另一种类型的值，这是因为有些属性只接受整数（例如 `z-index`），而有些则接受任何实数（例如 `flex-grow`）。与整数值一样，对于取数字值的属性来说，按照定义，取值范围可能有限。例如，`opacity` 属性的值被限制在 0 到 1（含）之间的有效 <*number*>。默认情况下，在定义的取值范围之外的数字值是无效的，会导致整个声明被忽略。然而，按照某些属性的定义，取值范围外的数字会被设为与所用值最接近的数（这种行为一般称为 clamping）。

4.2.3 百分数

百分数值是在 <*number*> 后跟一个百分号（%），在取值句法中用 <*percentage*> 表示。例如 50% 和 33.333%。百分数值始终相对于另一个值，另一个值可以是任何值，例如同一个元素另一个属性的值，继承自父元素的值，或者祖辈元素的值。接受百分数值的属性都会定义取值范围，以及百分比相对什么计算。

4.2.4 弹性值

弹性值（*fraction value* 或 *flex value*）是 <*number*> 后跟 `fr`。因此，单位弹性值是 1fr。这个概念由栅格布局引入，用于把布局中不受限制的空间分成几部分。详情参见第 13 章。

4.3 距离

很多 CSS 属性（例如外边距）要靠长度度量衡才能正确显示页面中的各个元素。毫不意外，CSS 中有很多度量长度的方式。

长度值用正数或负数后跟一个标注表示，不过有些属性只接受正数。此外，还可以使用实数，即带小数的数字，例如 10.5 或 4.561。长度的后面是一个缩写（通常是两个字符），表示长度的单位，例如 `in`（英寸）或 `pt`（点）。长度为 `0` 时是唯一的例外，此时无需单位。

长度单位分为两种：绝对长度单位和相对长度单位。

4.3.1 绝对长度单位

先介绍绝对长度单位，因为它们最易于理解，不过 Web 设计中很少使用。绝对长度单位有七个，说明如下：

英寸（in）

如你所料，这个表示法指美国尺子所用的英寸（在全世界几乎都使用米制系统的今天，规范中居然还有这个单位，真是有趣。由此也能看出美国对互联网的影响有多大，不过还是不介入社会问题了）。

厘米（cm）

指世界上其他地方使用的尺子上的厘米。1 英寸等于 2.54 厘米，1 厘米等于 0.394 英寸。

毫米（mm）

1 厘米等于 10 毫米（不习惯米制的美国人可能不知道），因此 1 英寸等于 25.4 毫米，1 毫米等于 0.0394 英寸。

四分之一毫米（q）

1 厘米中有 40 个 q 单位。因此，把元素设为 1/10 厘米宽（即 1 毫米宽）得到的值是 4q（截至 2016 年年末，只有 Firefox 支持 q）。

点（pt）

点是一个标准的印刷度量单位，在打印机和打字机上已经使用数十年，字处理程序也使用多年了。按惯例，1 英寸有 72 点（点是在米制系统广泛使用之前定义的）。因此，把大写字母设为 12 点的意思是文本的高度为 1/6 英寸高。例如，`p {font-size: 18pt;}` 等效于 `p {font-size: 0.25in;}`。

派卡（pc）

派卡（pica）也是印刷术语。1 派卡等于 12 点，因此 1 英寸有 6 派卡。类似地，把大写字母设为 1 派卡的意思是文本的高度为 1 英寸。例如，`p {font-size: 1.5pc;}` 设定的文本大小与前面给出的声明一样。

像素（px）

像素是屏幕上的小点，不过 CSS 定义的像素较为抽象。在 CSS 中，1 像素所占的尺寸够 1 英寸中放下 96 像素。很多用户代理忽略这个定义，而是直接使用屏幕上的像素。缩放页面或打印时要考虑缩放，此时 `100px` 宽的元素经渲染后得到的宽度可能大于设备上的 100 个小点。

仅当浏览器掌握显示页面的屏幕、所用的打印机或其他用户代理的所有细节时，这些单位才真正有其用处。对 Web 浏览器来说，显示效果受屏幕的尺寸和设定的分辨率影响，而作为创作人员的你，对这些因素通常没有多少控制力。你只能祈祷度量的差距是恒定的，例如，0.1in 是 0.5in 的两倍大，如图 4-1 所示。

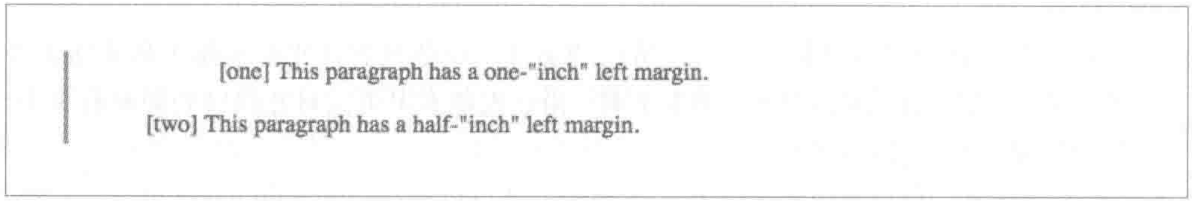


图 4-1：使用绝对长度设定左外边距

然而，如果抛开这一切，假设你的电脑掌握了显示系统的足够信息，能准确重现现实中的度量（当然这是不可能的），那么，声明 `p {margin-top: 0.5in;}` 得到的结果一定是每个段落的上外边距都为半英寸。不考虑字号等因素的话，段落的上外边距将为半英寸。

绝对单位在定义文档的印刷样式时非常有用，因为打印机通常使用英寸、点和派卡。

像素长度

像素不像表面上看起来那么简单。靠近屏幕，可以看到屏幕上排布着网格状的小方块。这样一个方块就是一像素。如果像下面这样把一个元素的高度和宽度定义为多少多少像素：

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

元素在屏幕上就显示为那么高和那么宽，如图 4-2 所示。

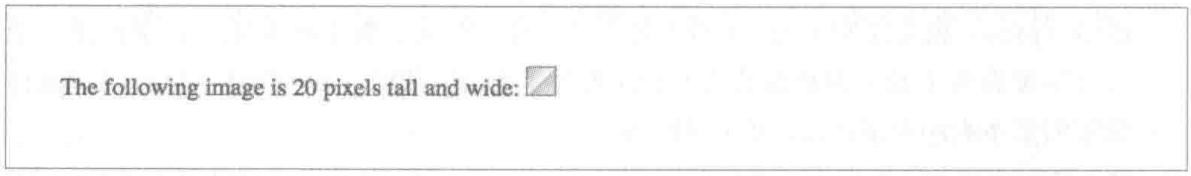


图 4-2：使用像素长度

一般来说，Web 浏览器处理 `font-size: 18px` 时基本上会在屏幕上使用指定的像素数，毕竟屏幕就是由一个个像素点构成的。但是，对其他显示设备而言，例如打印机，用户代理将根据情况缩放像素长度。也就是说，打印代码要确定一像素有几点。

像素度量还常用于表示图像的尺寸，即一个图像是多少像素高、多少像素宽。然而对如今流行的响应式设计来说，通常会用相对于文本的尺寸标注图像的大小，而不关心图像中究竟有多少像素。这往往依赖于用户代理的图像缩放技术，不过这个技术已经相当成熟了。矢量图像（如 SVG）严重依赖图像缩放技术。

像素理论

讲到像素时，CSS 规范建议，如果显示器的像素密度与每英寸 96 像素（pixels per inch，ppi）相差特别大，用户代理应该缩放像素度量，使用“参考像素”（reference pixel）。CSS2 建议用 90ppi 做参考像素，而 CSS2.1 和 CSS3 建议使用 96ppi。对最常见的打印机来说，它用的单位是点，而不是像素，而且每英寸中能放下的点数远超 96！打印网页时，可以假设每英寸 96 像素，然后据此缩放输出。

如果显示器的分辨率为 1024 × 768 像素，而屏幕的实际尺寸为 10 2/3 英寸宽、8 英寸高，而且屏幕占满整个显示器，那么一像素的宽和高都是 1/96 英寸。跟你想的一样，实际情况不可能完全如此。因此，在多数显示器中，每英寸的像素数（ppi）通常都大于 96，有时要大得多。例如 iPhone 4S 的视网膜显示屏是 326 ppi，iPad 的显示屏是 264 ppi。



Windows XP 用户可以把显示器设成显示的尺寸与实际尺寸完全一致。设置的方法是选择“开始”菜单，打开“控制面板”；双击“显示”，打开“设置”选项卡，单击“高级”；在打开的对话框中有一个下拉菜单或其他控件，标注为“字号”，选择“其他”，此时会出现一个标尺（不同的电脑设置的方法可能不同）。

4.3.2 分辨率单位

随着媒体查询和响应式设计的出现，为了描述显示器的分辨率，出现了三个新单位。

点每英寸（dpi）

在长为 1 英寸的范围内能显示的点数。可以指打印机输出的点，也可以指 LED 屏幕等设备上的物理像素点，或者电子墨水屏（例如 Kindle 用的）上的图像单元。

点每厘米（dpcm）

与 dpi 类似，不过测量的范围是 1 厘米，而不是 1 英寸。

点每像素单位（dppx）

CSS 中每个 px 单位显示的点数。从 CSS3 起，1dppx 相当于 96dpi，因为 CSS 就是按照这个换算比例定义像素单位的。不过，这个比例在 CSS 未来的版本中可能会变。

截至 2017 年年末，这些单位只能在媒体查询中使用。比如说，创作人员可以让一个媒体查询块中的样式只用于分辨率超过 500 dpi 的显示器：

```
@media (min-resolution: 500dpi) {
    /* 样式规则 */
}
```

4.3.3 相对长度单位

相对长度单位中的“相对”是指其长度是相对其他东西而言的。相对长度的实际值（或绝对值）根据不受其控制的因素而变，例如屏幕分辨率、视区宽度、用户的偏好设置等。此外，有些相对单位的实际尺寸始终相对当前元素，因此在元素之间也有差异。

em 和 ex 单位

首先介绍两个联系紧密的单位：em 和 ex。按 CSS 的定义，1 em 等于元素的 font-size 属性值。如果元素的 font-size 为 14 像素，那么对那个元素来说，1em 就等于 14 像素。

你可能想到了，1 em 的值在元素之间是不同的。例如，假设 h1 的字号为 24 像素，h2 的字号为 18 像素，段落的字号为 12 像素。如果把这三个元素的左外边距都设为 1em，那么实际的外边距值分别为 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>
```

如果使用 em 设定字号，那么它相对父元素的字号而言，如图 4-3 所示。

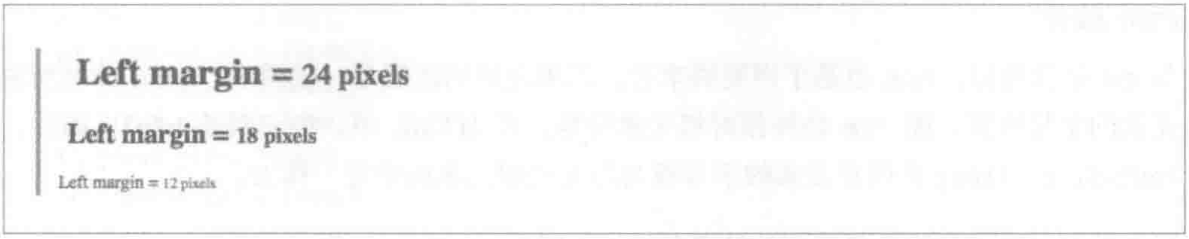


图 4-3：使用 em 设定外边距和字号

理论上，1 em 等于所用字体中小写字母 *m* 的宽度。其实，“em”这个名称就是由此而来的。这是一个古老的排版术语。然而，CSS 没有采用这个定义。

ex 指所用字体中小写字母 *x* 的高度。因此，如果两个段落的字号都是 24 点，但是使用的字体不同，那么 ex 的值也不一样。这是因为不同字体中的 *x* 高度有所不同，如图 4-4 所示。虽然每一行文本的字号都是 24 点（因此 em 的值都是 24 点），但是 x 的高度各不相同。



图 4-4：x 的高度各不相同

rem 单位

与 em 单位类似，rem 也基于声明的字号。二者之间的区别是（很微小），em 相对当前元素的字号计算，而 rem 始终相对根元素计算。在 HTML 中，根元素是 html。因此，font-size: 1rem; 声明把元素的字号设为与文档根元素的字号一样大。

举个例子。下述标记片段得到的结果如图 4-5 所示。

```
<p> This paragraph has the same font size as the root element thanks to
    inheritance.</p>
<div style="font-size: 30px; background: silver;">
  <p style="font-size: 1em;">This paragraph has the same font size as its parent
    element.</p>
  <p style="font-size: 1rem;">This paragraph has the same font size as the root
    element.</p>
</div>
```

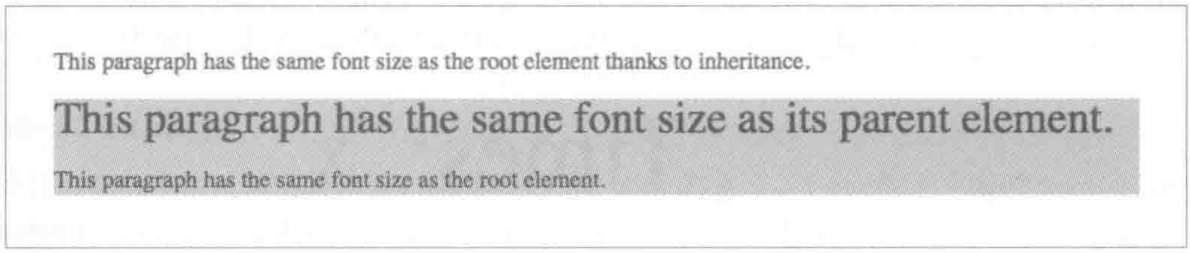


图 4-5：em 与 rem 的区别

rem 的实际作用相当于重设字号：不管祖辈元素把字号设为多大，font-size: 1rem; 都会把字号还原成根元素设定的大小。通常，1rem 是用户的默认字号，除非你（或用户）为根元素设定了其他字号。

例如，对下述声明来说，1rem 始终相当于 13px：

```
html {font-size: 13px;}
```

然而，对下述声明而言，1rem 始终相当于默认字号的四分之三：

```
html {font-size: 75%;}
```

此时，如果默认字号是 16 像素，那么 1rem 等于 12px。如果用户把默认值改成了 12 像素（有些人会这么做），那 1rem 就等于 9px；改为 20 像素的话，1rem 等于 15px。以此类推。

当然，不一定非使用 1rem 不可。与 em 单位一样，可以使用任何实数。由此可以做些有趣的事，比如把标题的字号设为根元素字号的倍数：

```
h1 {font-size: 2rem;}
h2 {font-size: 1.75rem;}
h3 {font-size: 1.4rem;}
h4 {font-size: 1.1rem;}
h5 {font-size: 1rem;}
h6 {font-size: 0.8rem;}
```



在支持 initial 关键字的浏览器中，如果没为根元素设定字号的话，font-size: 1rem 的作用与 font-size: initial 相同。

ch 单位

CSS3 新增了一个有趣的单位：ch。这个单位基本上可以理解为“一个字符”。CSS3 规范是这样定义的：

> 等于渲染时所用字体中“0”（零，U+0030）字形的进距。

“进距”（advance measure）其实是 CSS 自造的，对应于字体排印中的“进宽”（advance width）。CSS 使用“距”是因为有些文字不是从右向左或从左向右写的，而是从上到下或从下到上写的，而此时应该用“进高”而非“进宽”。简单起见，本节采用“进宽”一说。

抛开繁杂的细节，简单来说，字形的进宽指一个字形的起点到下一个字形的起点之间的距离。一般情况下，这段距离等于字形本身的宽度加上侧边的间距（间距可以为正也可以为负）。

CSS 把 ch 单位定义为所用字体中一个零的进宽。这跟 em 基于元素的 font-size 值计算有点类似。

为了更好地理解这个单位，我们可以把一串零放在一起，然后在下面放一个图像，把图像的宽度设为与零的个数一样。当然，宽度的单位用 ch。结果如图 4-6 所示。

```
img {height: 1em; width: 25ch;}
```

对等宽字体来说，所有字符的宽度都是 1ch。在宽度不一的字体中（绝大多数西文字体），字符的宽度可能比“0”宽，也可能比“0”窄，因此不能认为是 1ch 宽。

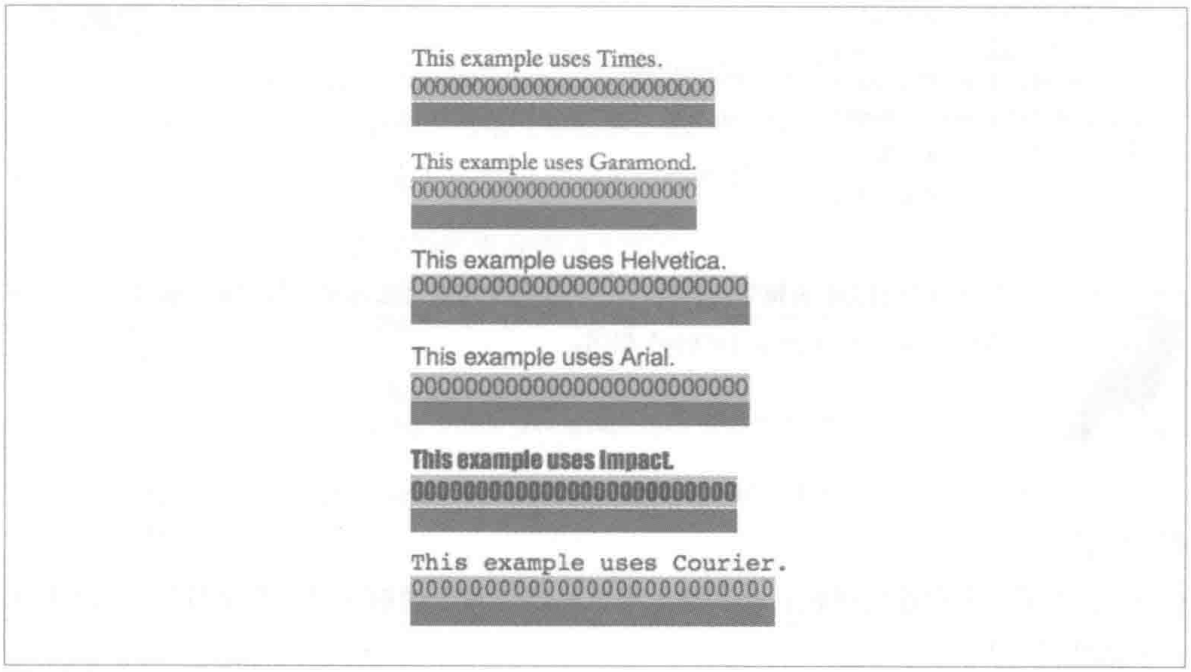


图 4-6：根据字符计算尺寸



截至 2017 年年末，只有 Opera Mini 和 Internet Explorer 对 ch 的支持有问题。在 IE11 中，ch 的宽度是错的，正好等于“0”字形的宽度，而不是字形的宽度加上侧边的间距。因此，在 IE11 中，5ch 的宽度比“00000”小。这个错误在 Edge 中纠正了。

视区相关的单位

CSS3 还新增了三个与视区尺寸相关的单位。这些单位根据视区的尺寸计算，比如浏览器窗口、可打印区域、移动设备的显示屏等。

视区宽度单位（vw）

这个单位根据视区的宽度计算，然后除以 100。因此，如果视区的宽度是 937 像素，那么 1vw 等于 9.37px。如果视区的宽度有变，例如把浏览器窗口拉宽或缩窄，vw 的值随之改变。

视区高度单位（vh）

这个单位根据视区的高度计算，然后除以 100。因此，如果视区的高度是 650 像素，那么 1vh 等于 6.5px。如果视区的高度有变，例如把浏览器窗口拉高或缩矮，vh 的值随之改变。

视区尺寸最小值单位（vmin）

这个单位等于视区宽度或高度的 1/100，始终取宽度和高度中较小的那个。因此，如果一个视区的宽度为 937 像素，高度为 650 像素，那么 `1vmin` 等于 6.5px。

视区尺寸最大值单位（vmax）

这个单位等于视区宽度或高度的 1/100，始终取宽度和高度中较大的那个。因此，如果一个视区的宽度为 937 像素，高度为 650 像素，那么 `1vmax` 等于 9.37px。

注意，这些是长度单位，因此在任何允许使用长度单位的地方都能使用。比如说，可以声明 `h1 {font-size: 10vh;}`，根据视区的尺寸缩放标题的字号。这个样式规则把字号设为视区高度的 1/10，比较适合用在文章的标题等元素上。

这些单位特别适合用于创建全视区界面，例如移动设备的界面，因为元素是根据视区的尺寸而变化的，与文档树中的任何元素都没关系。因此，轻易就能填满整个视区，或者至少填满视区的大部分，而不用担心特定情况下视区的具体尺寸。

下面是一个使用视区相关尺寸的简单示例，结果如图 4-7 所示。

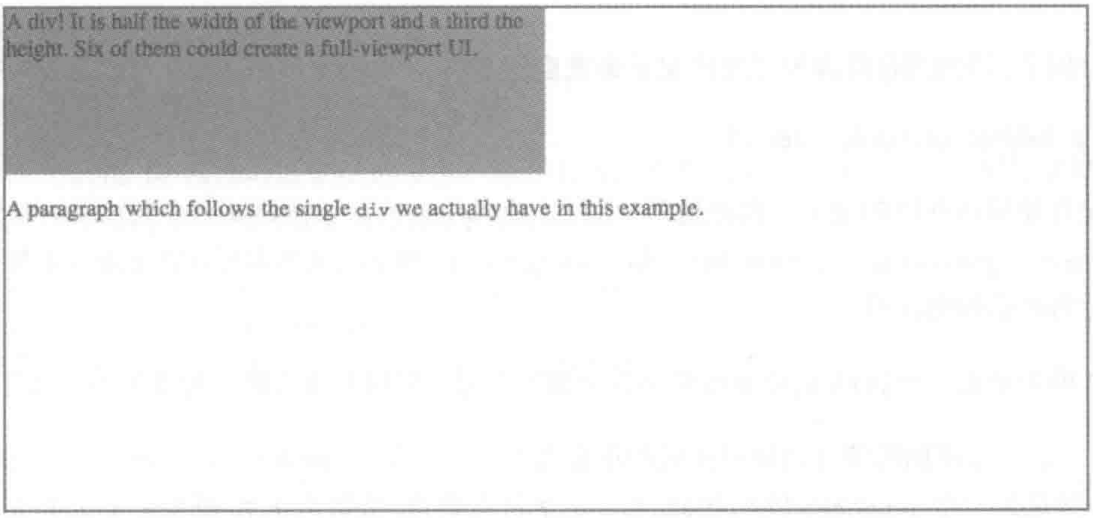


图 4-7：根据视区计算尺寸

```
div {width: 50vh; height: 33vw; background: gray;}
```

这些单位有趣的一面是（不过可能没什么用），它们不是与相应主轴绑定的。因此，比方说，我们可以声明 `width: 25vh;`，让元素的宽度等于视区高度的四分之一。



截至 2016 年年末，除了 Opera Mini，其他浏览器都支持视区相关的单位；不过，微软的浏览器不支持 vmax。

4.4 计算值

为方便你做数学计算，CSS 提供了 calc() 值。括号中可以使用简单的数学算式。允许使用的运算符有 +（加）、−（减）、*（乘）、/（除），以及括号。这些运算符的运算顺序与传统的 PEMDAS（括号、指数、乘、除、加、减）一样，不过这里其实只有 PMDAS，因为 calc() 不允许做指数运算。



允许在 calc() 中使用括号似乎是浏览器提供的便利措施，因为 calc() 的句法定义中没有提到括号。因此，浏览器可能会继续支持，不过使用时要小心。

举个例子。假设想让段落的宽度比父元素宽度的 90% 少 2 em。利用 calc() 可以这样声明：

```
p {width: calc(90% - 2em);}
```

在允许使用这些值的地方，都能使用 calc()：*<length>*、*<frequency>*、*<angle>*、*<time>*、*<percentage>*、*<number>* 和 *<integer>*。这些单位类型也都能在 calc() 中使用，不过有些限制要注意。

基本的限制是，calc() 会检查括号中各个值的类型，确保是兼容的。检查的方式如下：

1. + 和 − 号两侧的值必须使用相同的单位类型，或者是 *<number>* 和 *<integer>*（此时结果为一个 <number> 值）。因此，5 + 2.7 是有效的，结果为 7.7；而 5em + 2.7 无效，因为一边有长度单位，一边没有。注意，5em + 20px 是有效的，因为 em 和 px 都是长度单位。

2. * 计算的两个值中必须有一个是 *<number>*（注意，它包括整数）。因此，2.5rem * 2 和 2 * 2.5rem 都是有效的，结果均为 5rem。而 2.5rem * 2rem 是无效的，因为这样得到的结果是 5rem^2，这是面积单位，而不是长度单位了。

3. / 计算的两个值中右边的那个必须是 *<number>*。左边是 *<integer>* 时，结果为一个 *<number>* 值；否则，结果的单位与左边的值一样。因此，30em / 2.75 是有效的，而 30 / 2.75em 是无效的。

4. 此外，任何情况下都不能除以零。30px/0 这样的算式很明显，但有时并不显而易见。

此外还有一个小限制要注意，即 + 和 - 运算符的两侧必须有空白，* 和 / 没有这一限制。这样有助于避免混淆负数。

除此之外，规范要求用户代理至少要支持 calc() 中的算式可以使用 20 个算子，算子可以是数字、百分数，也可以是大小量（长度）。超过这一限制的算式应该视为无效的。

4.5 属性值

在一些 CSS 属性中，可以使用样式对应的元素上的 HTML 属性值。方法是使用 attr() 表达式。

例如，使用生成的内容时，可以插入任何属性的值。比如下面这样（不完全理解句法没关系，我们将在第 15 章讨论）：

```
p::before {content: "[" attr(id) "]";}
```

这个表达式把 id 属性的值放在括号里，加在有 id 属性的段落前面。因此，把这个样式应用到下述段落上得到的结果如图 4-8 所示。

```
<p id="leadoff">This is the first paragraph.</p>
<p>This is the second paragraph.</p>
<p id="conclusion">This is the third paragraph.</p>
```

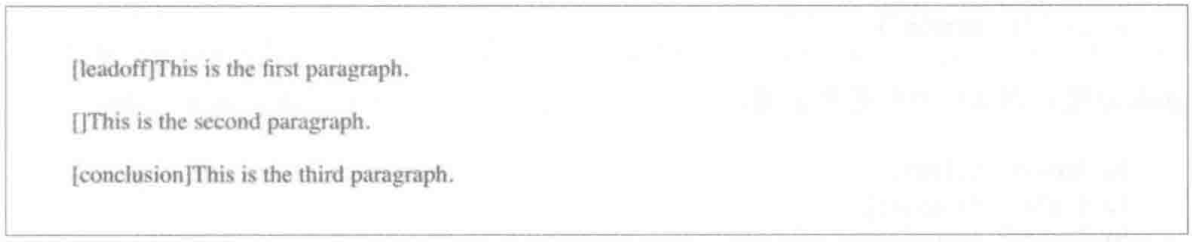


图 4-8：插入属性值

理论上，只要在表达式中指定值的类型，attr() 可以获取任何属性的值。例如，可以（也是理论上）在输入框上使用 maxlength 属性的值确定输入框的宽度，如下所示：

```
input[type="text"] {width: attr(maxlength em);}

<input type="text" maxlength="10">
```

假设用户代理支持这样使用 attr()，上述 input 元素的宽度将被设为 10 em。然而，截至 2016 年年末，情况并不乐观：接受测试的浏览器都不支持这样使用 attr()。

4.6 颜色

每个 Web 创作新手都会问这样一个问题，“怎样设定页面中使用的颜色？”在 HTML 中，有两个选择：可以使用数量不多的颜色名，例如 red 或 purple，或者使用有点晦涩的十六进制代码。CSS 也支持这两种方法，不过还提供了其他我觉得更直观的方法。

4.6.1 具名颜色

如果只想使用基本的颜色，最简单的方法是使用颜色的名称。CSS 把这种颜色称为具名颜色（named color）。早期，CSS 中有 16 个基本的颜色关键字，对应 HTML 4.01 定义的那 16 个颜色。这些具名颜色见表 4-1。

表 4-1：16 个基本的颜色关键字

aqua	gray	Navy	silver
black	green	Olive	teal
blue	lime	Purple	white
fuchsia	maroon	Red	yellow

因此，如果想把一级标题的颜色设为红褐色，最好这样声明：

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

很简单吧？图 4-9 中有更多示例。

```
h1 {color: silver;}
h2 {color: fuchsia;}
h3 {color: navy;}
```

你可能见过别人（甚至自己）使用的颜色名称不在上表中。例如下面这个：

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

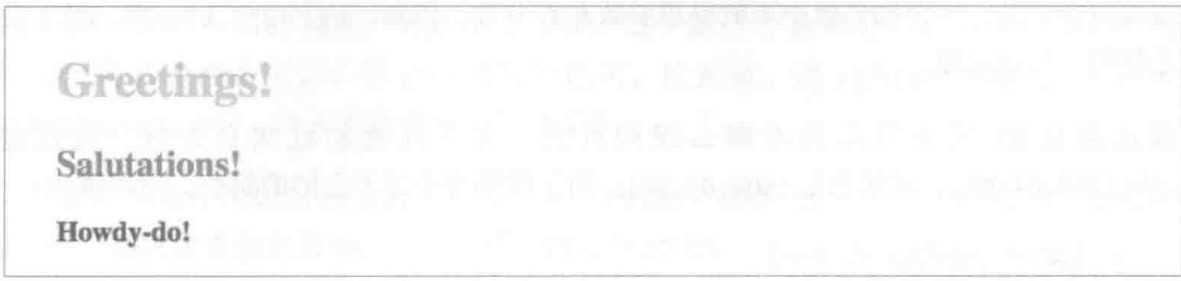


图 4-9：使用具名颜色

截至 2017 年年末，最新的 CSS 颜色规范包含这 16 个具名颜色，不过它们属于一个更大的列表，一共有 148 个颜色关键字。那个列表根据已经使用数十年的 X11 RGB 标准值扩展，浏览器也已经支持数年了，此外还有一部分颜色名称来自 SVG（最多的是各种灰色）。CSS Color Module Level 4 定义了这 148 个关键字与不同颜色表示方式的对照表，参见附录 C。

幸好，CSS 为表示颜色提供了更细致、更精确的方式。使用这些方法可以指定色谱中的任何颜色，而不用受限于数量有限的具名颜色。

4.6.2 RGB 和 RGBa 颜色

计算机中的颜色按不同比例的红绿蓝混合而成，因此经常被称为 RGB 颜色。显示器上的一个点是一个像素。根据屏幕上颜色的创建方式，理应可以直接访问这些颜色，由你自己确定如何混合三种颜色。这种方法是很复杂，但完全可行，而且值得，因为对可得到的颜色几乎没有限制。采用这种方法创建的颜色有 4 种指定方式。

函数式 RGB 颜色

有两种颜色值使用函数式 RGB 表示法，而不使用十六进制表示法。这种颜色值的一般句法是 `rgb(color)`，其中 `color` 是三个值，可以是百分数或整数。百分数的取值范围是 `0 ~ 100%`，整数的取值范围是 `0 ~ 255`。

因此，使用百分数表示白色和黑色的方式如下：

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

使用三个整数表示的方法如下：

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

值得注意的是，一个颜色值中不能混用整数和百分数。例如，rgb(255,66.67%,50%) 是无效的，将被忽略。

假设想让 h1 元素显示为不那么深的红色，介于红色和红褐色之间。红色是 rgb(100%,0%,0%)，红褐色是 (50%,0%,0%)。为了得到介于二者之间的颜色，可以试试：

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

这个颜色的红色通道比红褐色淡，比红色深。如果想创建浅红色，可以增加绿色和蓝色通道的值：

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

使用整数表示法，与之最为接近的是：

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

若想查看这些值与颜色之间的对应关系，最简单的方法是创建一个灰度值表。下述规则的结果如图 4-10 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);}
```

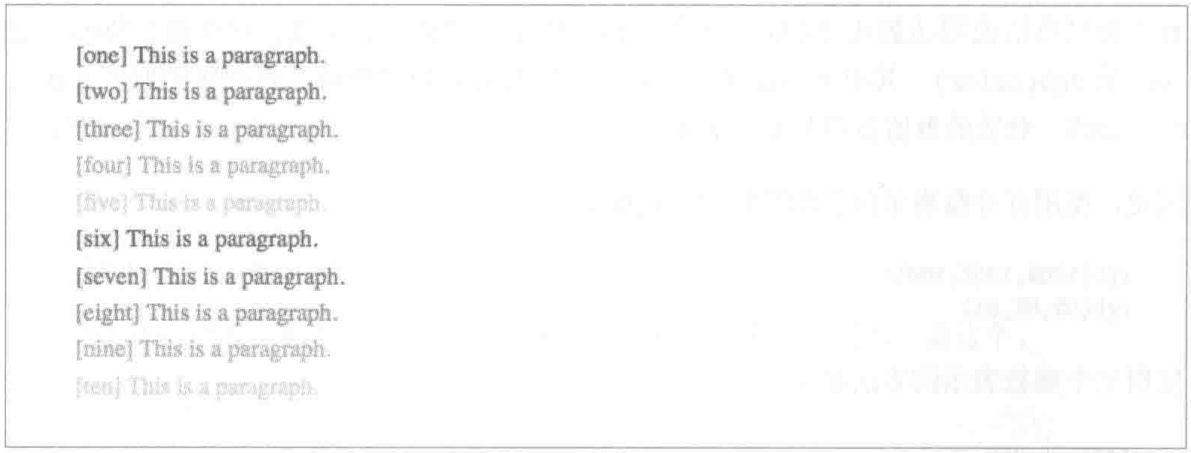


图 4-10：不同灰度的文本

这里显示的是不同的灰度，因此每个 RGB 值中的三个数都是一样的。如果其中有一个数与其他两个不同，便会得到新的色调。比如说，把 rgb(50%,50%,50%) 改成 rgb(50%,50%,60%)，得到的是带点蓝色的暗色。

百分数表示法中可以使用小数。有时，基于特定的原因，你可能想把红色通道指定为 25.5%、绿色通道指定为 40%、蓝色通道指定为 98.6%：

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

不支持小数的用户代理应该把小数归整到最近的整数，得到 rgb(26%,40%,99%)。使用三个整数时，各个通道只能使用整数。

不管使用哪种表示法，超出取值范围的数将“裁剪”为最近的边界值。例如，大于 100% 或小于 0% 的值将变成 100% 和 0%。因此，下述声明中的颜色值将变成注释中给出的值：

```
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  */
```

百分数与整数之间的转换关系不是随意的，也不是靠猜的，而是有一个简单的公式。如果知道 RGB 颜色每个通道的百分数值，只需乘以 255 就能得到整数值。假如一个颜色的红色通道是 25%，绿色通道是 37.5%，蓝色通道是 60%。各百分数乘以 255 之后得到结果分别是 63.75、95.625 和 153。归整到最近的整数后，得到 rgb(64,96,153)。

如果已经知道百分数值，没有什么必要转换成整数值。对使用 Photoshop 等程序的人来说，整数表示法更有用，因为这些程序在信息对话框中显示的就是整数值；此外，熟悉颜色生成技术细节的人善于以 0~255 之间的数思考问题，这些人也喜欢使用整数表示法。

RGBa 颜色

从 CSS3 起，上述两种函数式 RGB 表示法发展成了函数式 RGBa 表示法。这种表示法在 RGB 的三个通道后面增加了一个 alpha 值，即“red-green-blue-alpha”，简称 RGBa。这里的 alpha 指 alpha 通道，用于衡量不透明度。

例如，假设你想让元素的文本显示为半透明的白色，以便透过半透明的白色看到文本背后的颜色。此时，可以使用下面两个值中的一个：

```
rgba(255,255,255,0.5)
rgba(100%,100%,100%,0.5)
```

如果想让颜色完全透明，把 alpha 值设为 0；如果想完全不透明，应该设为 1。因此，rgb(0,0,0) 和 rgba(0,0,0,1) 得到的是完全一样的结果（黑色）。图 4-11 中几个段落的透明度依次递增，所用的规则如下：

```
p.one {color: rgba(0,0,0,1);}
p.two {color: rgba(0%,0%,0%,0.8);}
p.three {color: rgba(0,0,0,0.6);}
p.four {color: rgba(0%,0%,0%,0.4);}
p.five {color: rgba(0,0,0,0.2);}
```

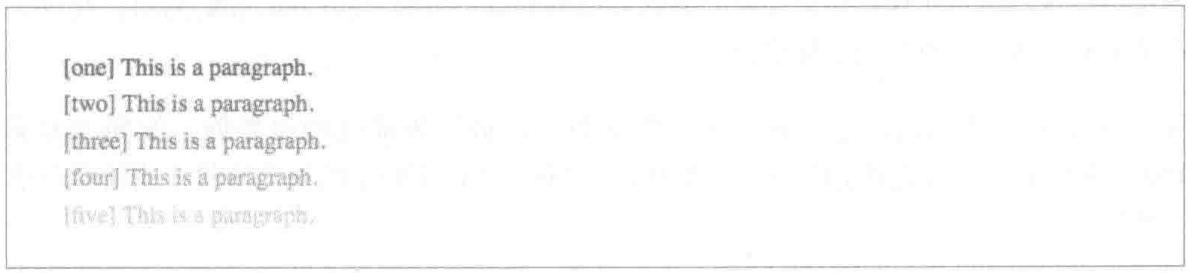


图 4-11：透明度依次递增的文本

毫无疑问，alpha 值始终是 0-1 范围内的实数。超出范围的值要么被忽略，要么被重置为最近的有效值。alpha 值不能使用 *<percentage>* 表示，即便数值一样也不行。

十六进制 RGB 值

CSS 支持使用 HTML 编写人员习惯的十六进制表示法定义颜色：

```
h1 {color: #FF0000;}   /* 把 h1 设为红色 */
h2 {color: #903BC0;}   /* 把 h2 设为暗紫色 */
h3 {color: #000000;}   /* 把 h3 设为黑色 */
h4 {color: #808080;}   /* 把 h4 设为中灰色 */
```

计算机使用十六进制表示法有段时间了，程序员往往经过后天训练掌握或习惯了这种表示法。因此，顺理成章，HTML 采用了十六进制表示法。而 CSS 则沿袭了这种表示法。

这种表示法把三个 00 到 FF 范围内的十六进制数串在一起，表示一个颜色。一般的句法是 #RRGGBB。注意，这三个数之间没有空格、逗号或其他分隔符。

十六进制表示法与三个整数表示法在数值上是等效的。例如，rgb(255,255,255) 与 #FFFFFF 完全等效，rgb(51,102,128) 则与 #336680 一样。你可以根据喜好选择二者中的任何一个，多数用户代理对它们的渲染方式是一样的。如果你的计算器能在十进制和十六进制之间转换，从一种表示法变到另一种表示法是十分容易的。

如果每个十六进制数中的两个数字相等，CSS 允许使用简短表示法。一般的句法是 #RGB：

```
h1 {color: #000;}   /* 把 h1 设为黑色 */
h2 {color: #666;}   /* 把 h2 设为暗灰色 */
h3 {color: #FFF;}   /* 把 h3 设为白色 */
```

可以看出，每个颜色值中只有三个数字。可是，00 和 FF 之间的十六进制数需要两个数字，这里总共只有三个数字，这样怎么行呢？

其实，浏览器会把每个数字复制成两个。因此，#F00 变成了 #FF0000，#6FA 变成了 #66FFAA，#FFF 变成了 #FFFFFF，即白色（white）。不是每个颜色都能像这样表示。比如说，中灰色就要写成标准的十六进制表示法，即 #808080。这个颜色不能使用简短表示法表示。与之最接近的是 #888，它等同于 #888888。

十六进制 RGBa 颜色

（截至 2017 年年末）有个新的十六进制表示法在后面添加一个十六进制值，表示 alpha 通道的值。图 4-12 中的几个段落以逐渐递增的透明黑色显示，这与前一节的效果一样，使用的规则如下：

```
p.one {color: #000000FF;}
p.two {color: #000000CC;}
p.three {color: #00000099;}
p.four {color: #00000066;}
p.five {color: #00000033;}
```

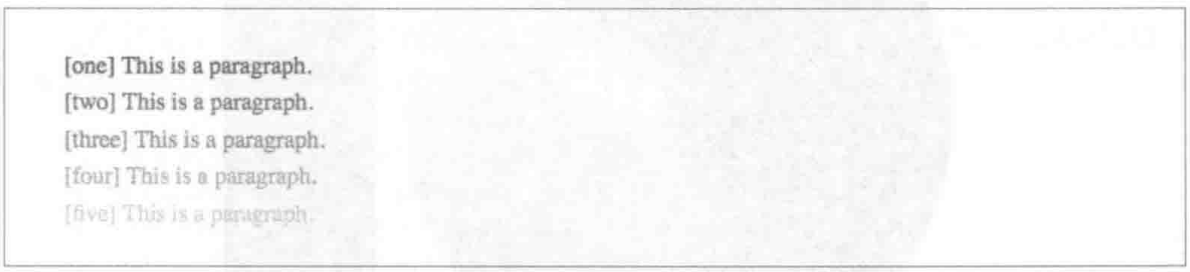


图 4-12：透明度依次递增的文本（另一种方式）

与不带 alpha 值的十六进制表示法一样，如果四个十六进制值中的两个数字都一样，可以简写。因此，#663399AA 可以写成 #639A。如果有任何一个十六进制数的两个数字不一样，就必须完整写出八个数字。例如，#663399CA 不能简写为 #639CA。



截至 2017 年年末，Firefox 和 Safari 支持带 alpha 通道的十六进制表示法，Chrome 和 Opera 已有试验性实现。

4.6.3 HSL 和 HSLa 颜色

CSS3 新增了 HSL 表示法（不过与一般的颜色理论不同）。HSL 是 Hue（色相）、Saturation（饱和度）和 Lightness（明度）的简称，其中色相是角度值，取值范围是 0~360，饱和度是从 0（无饱和度）~100（完全饱和）的百分数，明度是从 0（全暗）~100（全明）的百分数。

色相分布在一个色相环上，随着角度的旋转得到色谱上的不同颜色。0 ° 是红色，旋转 360° 又回到红色。图 4-13 在一个圆盘上显示色谱中的角度和颜色，还有一个直线带。

习惯 RGB 的人，一开始可能会被 HSL 搞糊涂（同样，习惯 HSL 的人也会被 RGB 搞晕）。HSL 中的色相通过图 4-14 中的关系图更好理解，前三条是不同位置上的色谱，最后一条是红绿蓝三种颜色混合的结果。

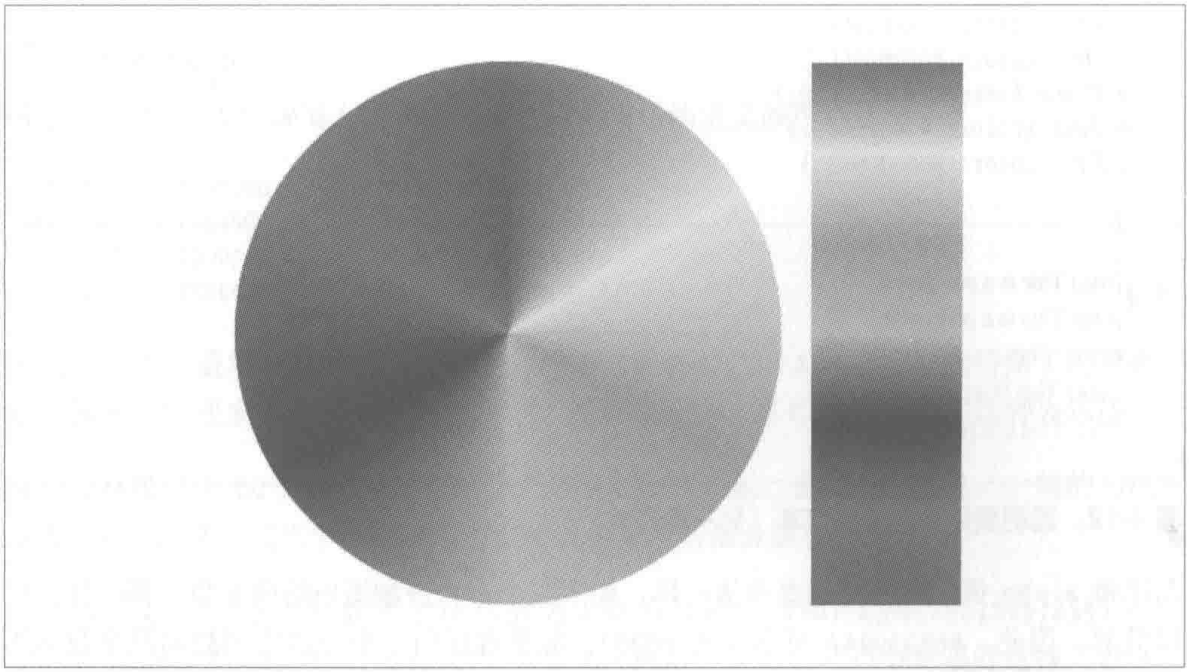


图 4-13：圆盘和直线带上的色谱

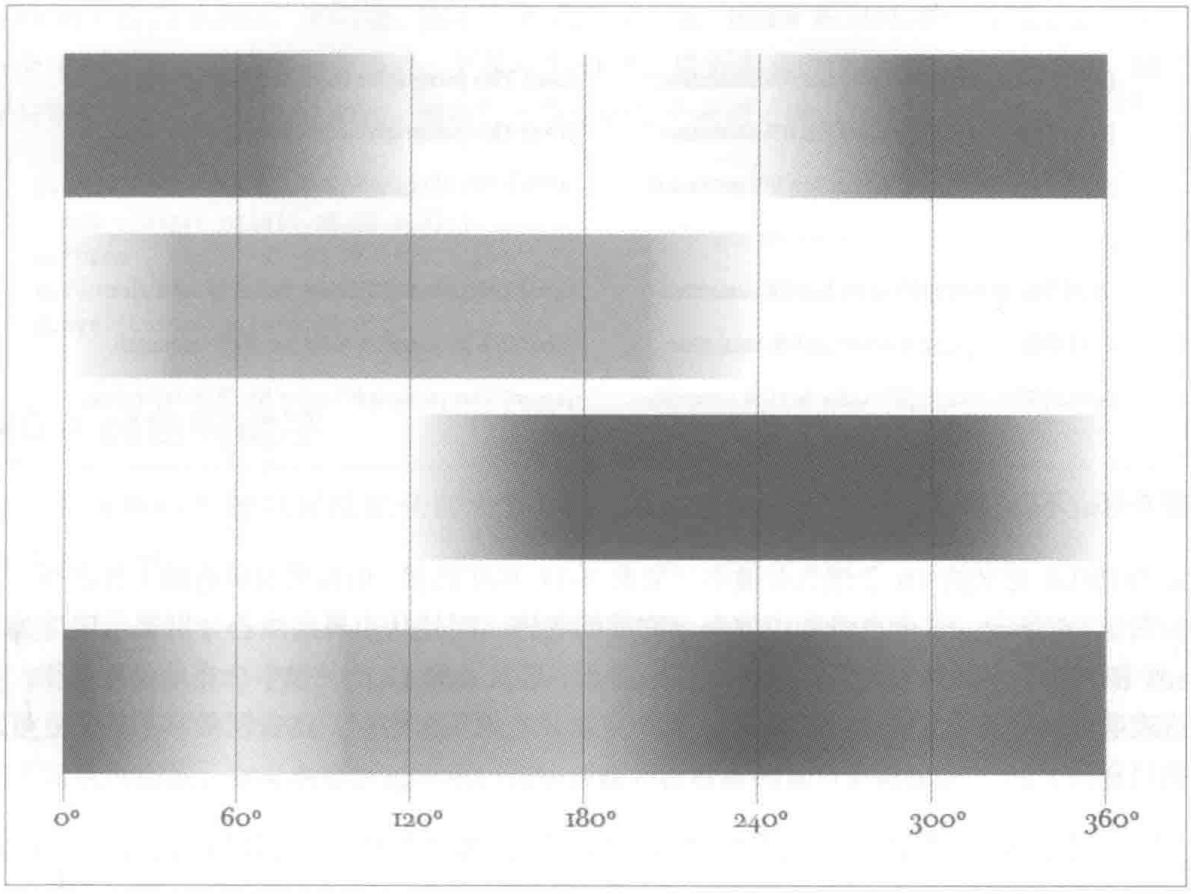


图 4-14：混合 RGB 得到的色谱

饱和度衡量颜色的强度。饱和度为 0% 时，不管色相角度为多少，得到的都是不太暗的灰色；饱和度为 100% 时，在明度一定时，色相最饱满。明度定义颜色有多暗或多亮。明度为 0% 时，不管色相和饱和度为多少，始终为黑色；而明度为 100% 时，得到的是白色。下述样式的结果见图 4-15 的左边。

```
p.one {color: hsl(0,0%,0%);}
p.two{color: hsl(60,0%,25%);}
p.three {color: hsl(120,0%,50%);}
p.four {color: hsl(180,0%,75%);}
p.five {color: hsl(240,0%,0%);}
p.six {color: hsl(300,0%,25%);}
p.seven {color: hsl(360,0%,50%);}
```

左边显示为灰色不是受打印效果的限制，其实每一行都是一种灰色，因为每个颜色值的饱和度都是 0%（中间那个值）。明度或暗度由第三个值设定。上述七个例子的色相角度都不一样，但是对最终结果没什么影响。当然，前提是饱和度始终为 0%。如果饱和度增加了，比如设为 50%，那么色相角度的影响就大了，因为它控制着你看到的颜色是哪种。如果前述示例中的其他值不变，仅把饱和度改为 50%，得到的结果见图 4-15 的右边。

[one] This paragraph's color has 0% saturation.

[two] This paragraph's color has 0% saturation.

[three] This paragraph's color has 0% saturation.

[four] This paragraph's color has 0% saturation.

[five] This paragraph's color has 0% saturation.

[six] This paragraph's color has 0% saturation.

[seven] This paragraph's color has 0% saturation.

[one] This paragraph's color has 50% saturation.

[two] This paragraph's color has 50% saturation.

[three] This paragraph's color has 50% saturation.

[four] This paragraph's color has 50% saturation.

[five] This paragraph's color has 50% saturation.

[six] This paragraph's color has 50% saturation.

[seven] This paragraph's color has 50% saturation.

图 4-15：不同的明度和色相

把 HTML4 定义的 16 个颜色关键字（见表 4-1）标在色相 - 明度圆盘中有助于你理解，如图 4-16 所示。这个色盘的边界表示完整的色谱，而且从边界到中心分别表示明度为 50% 和 0%（贯穿时饱和度为 100%）。可以看出，圆盘上的 12 个关键字分布地很有规律，这表明规范选择它们是事出有因的。几个灰色分布地不太均匀，这表明最有用的灰色可能只有四个。

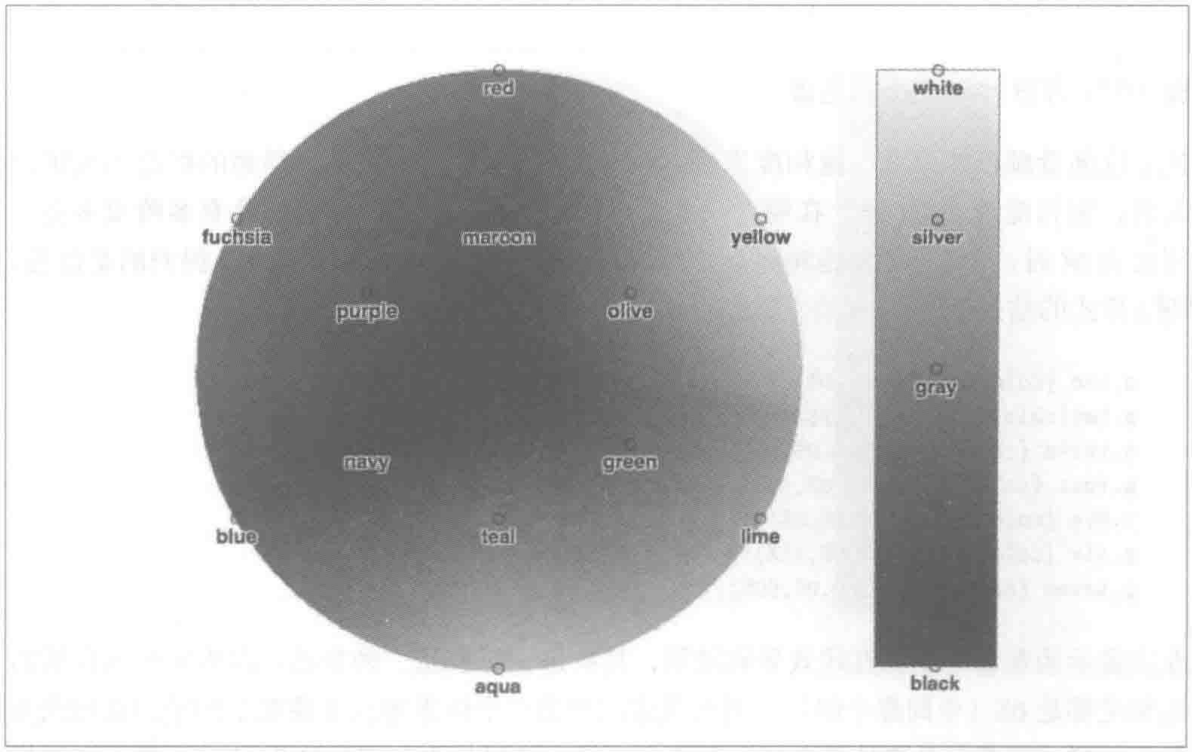


图 4-16：颜色关键字的色相角度和明度

RGB 有对应的 RGBa，类似地，HSL 有对应的 HSLa。HSLa 在 HSL 的三个值后面加上一个 alpha 值，取值范围是 0~1。下面几个 HSLa 值都表示黑色，不过透明度不同，结果与本章前面“十六进制 RGBa 颜色”一节中的那个示例一样，如图 4-11 所示。

```
p.one {color: hsla(0,0%,0%,1);}
p.two {color: hsla(0,0%,0%,0.8);}
p.three {color: hsla(0,0%,0%,0.6);}
p.four {color: hsla(0,0%,0%,0.4);}
p.five {color: hsla(0,0%,0%,0.2);}
```

4.6.4 颜色关键字

有两个特殊的关键字可以在任何允许使用颜色值的地方使用：transparent 和 currentColor。

从名称可以看出，transparent 表示完全透明的颜色。按照 CSS Color Module 的定义，它与 rgba(0,0,0,0) 等效，而这就是 transparent 计算得到的值。这个关键字不常用于设定文本颜色，不过却是元素背景色的默认值。此外，还可以用它为元素定义只占空间但不可见的边框，定义渐变时也常用，这些话题将在后面的章节中讨论。

currentColor 的意思是，“当前元素 color 属性计算得到的值”。对下述规则来说：

```
main {color: gray; border-color: currentColor;}
```

第一个声明把 main 元素的前景色设为 gray，第二个声明使用 currentColor 复制 color 属性计算得到的值（这里是 rgb(50%,50%,50%)，等同于 gray），然后应用到 main 元素的边框上。

4.7 角度

讲完 HSL 中的色相角度，现在比较适合讨论角度单位。角度一般使用 *<angle>* 表示，即一个 *<number>* 后跟下列四个单位中的一个。

deg
: 度数，完整的圆周是 360 度。

grad
: 百分度（gradian，也叫 grade 或 gon），完整的圆周是 400 百分度。

rad
: 弧度，完整的圆周是 2π（近似于 6.28）。

turn

圈数，一个完整的圆周是一圈。这个单位在旋转动画中最有用，比如说让一个元素旋转 10 圈就是 10turn（可惜，复数形式 turns 是无效的，将被忽略；至少截至 2017 年年末是这样）。

角度单位（见表 4-2）最常在 2D 和 3D 变形中使用，不过其他地方也能用到。注意，HSL 颜色中不使用角度单位，色相角度始终是度数，而且不加 deg 单位。

表 4-2：相同角度的不同表示

度数	百分度	弧度	圈数
0deg	0grad	0rad	0turn
45deg	50grad	0.785rad	0.125turn
90deg	100grad	1.571rad	0.25turn
180deg	200grad	3.142rad	0.5turn
270deg	300grad	4.712rad	0.75turn
360deg	400grad	6.283rad	1turn

4.8 时间和频率

属性的值为一段时间时，使用 *<time>* 表示，它是一个 *<number>* 值后跟 s（秒）或 ms（毫秒）。时间值最常在过渡和动画中使用，用于定义持续时间或延迟时间。下面两个声明的结果完全一样：

```
a[href] {transition-duration: 2.4s;}
a[href] {transition-duration: 2400ms;}
```

视听 CSS 也能用到时间值，用于定义持续时间或延迟时间。不过，写作本书时，对视听 CSS 的支持极其有限。

视听 CSS 中还有一种值 *<frequency>*，它是一个 *<number>* 值后跟 Hz（赫兹）或 kHz（千赫兹）。一同往常，这个单位的标识符不区分大小写，因此 Hz 和 hz 是等效的。下面两个声明的结果完全一样：

```
h1 {pitch: 128hz;}
h1 {pitch: 0.128khz;}
```

4.9 位置

位置值用于指定图像在背景区域中的位置，使用 *<position>* 表示。不过，位置值的句法结构相当复杂：

```
[
  [ left | center | right | top | bottom | <percentage> | <length> ] |
  [ left | center | right | <percentage> | <length> ]
  [ top | center | bottom | <percentage> | <length> ] |
  [ center | [ left | right ] [ <percentage> | <length> ]? ] &&
  [ center | [ top | bottom ] [ <percentage> | <length> ]? ]
]
```

看着有些难懂，但这是由位置值允许使用的复杂形式决定的。

如果只声明一个值，例如 left 或 25%，那么第二个值将被设为 center。因此，left 的作用与 left center 一样，25% 的作用与 25% center 一样。

如果声明两个值（像上面那样隐式或显式），而且第一个值是长度或百分数，那么前一个值始终是横向值。因此，对 25% 35px 来说，25% 是横向距离，35px 是纵向距离。如果调换顺序，写成 35px 25%，那么 35px 就是横向距离，而 25% 变成了纵向距离。这也意味着，如果写成 25% left 或 35px right，整个值都无效，因为两个值都是横向距离，没有指定纵向距离（类似地，right left 或 top bottom 也是无效的，将被忽略）。而写成 left 25% 或 right 35px 则没问题，因为既指定了横向距离（通过关键字），也指定了纵向距离（通过百分数或长度）。

如果声明四个值（使用三个值的说明参见下一段），必须有两个长度或百分数，而且前面都得是关键字。此时，长度或百分数指定偏移距离，而关键字定义偏移以哪一边计算。因此，right 10px bottom 30px 表示右边向左偏移 10 像素，底边向上偏移 30 像素。类似地，top 50% left 35px 表示顶边向下偏移 50%，左边向右偏移 35 像素。

三个值的处理方式跟四个值一样，不过最后一个偏移量将被设为零（即不偏移）。因此，right 20px top 的作用与 right 20px top 0 一样。

4.10 自定义值

2017 年年末，本书即将完稿时，CSS 新增了一个特性。这个特性的术语是“自定义属性”（custom property），不过它的作用其实是在 CSS 中创建变量。这个名称词不达意，它并不创建特殊的 CSS 属性（像 color 或 font 之类的）。

下面举个简单的例子，结果如图 4-17 所示。

```
html {
    --base-color: #639;
    --highlight-color: #AEA;
}

h1 {color: var(--base-color);}
h2 {color: var(--highlight-color);}
```

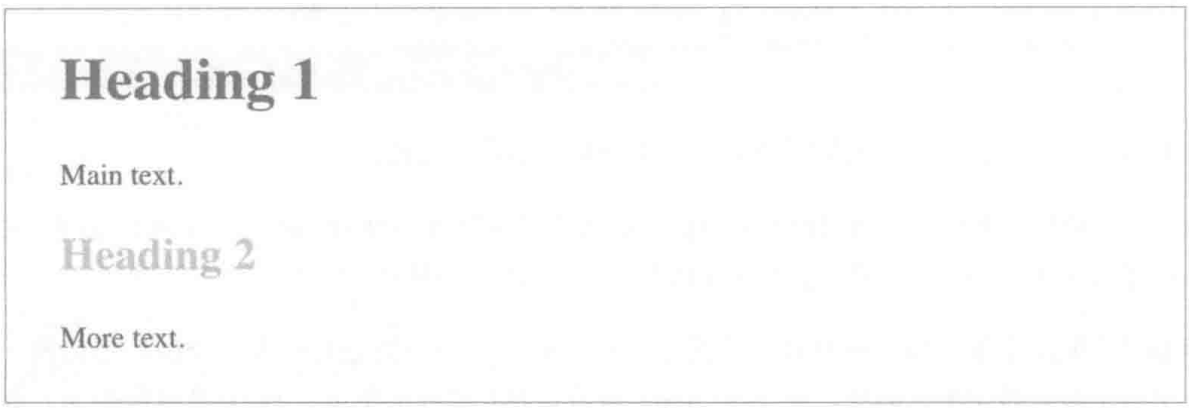


图 4-17：使用自定义值设定标题的颜色

有两个地方要理解。首先是自定义值 `--base-color` 和 `--highlight-color` 的定义。这不是某种特殊的颜色类型，而是根据值的作用而选择的名称。上述规则也可以写成：

```
html {
    --alison: #639;
    --david: #AEA;
}

h1 {color: var(--alison);}
h2 {color: var(--david);}
```

一般我们不会这么做，除非真的要定义对应于 Alison 和 David 这两个人名的颜色（比方说在“关于团队”页面）。通常，最好根据作用定义自定义标识符，例如 `main-color`、`accent-color` 或 `brand-font-face`。

另一点是，自定义标识符以两个连字符开头（--）。调用的方法是使用 var() 值类型。注意，这些名称是区分大小写的，因此 `--main-color` 和 `--Main-color` 是完全不同的两个标识符。

这些自定义标识符通常被称为“CSS 变量”，这解释了为什么使用 var() 调用它们。这个称呼有点道理，不过要记住，自定义标识符没有编程语言中的变量那样功能全面。其实，自定义标识符更像是文本编辑器中的宏，作用只是把一个值替换成另一个。

自定义属性有个有趣的特性：作用域在一定的范围内。如果你稍微知道这句话的意思，或许会激动一番。如果不明其意，下面举个例子说明，结果如图 4-18 所示。

```
html {
    --base-color: #639;
}
aside {
    --base-color: #F60;
}

h1 {color: var(--base-color);}

<body>

<h1>Heading 1</h1><p>Main text.</p>

<aside>
    <h1>Heading 1</h1><p>An aside.</p>
</aside>

<h1>Heading 1</h1><p>Main text.</p>

</body>
```

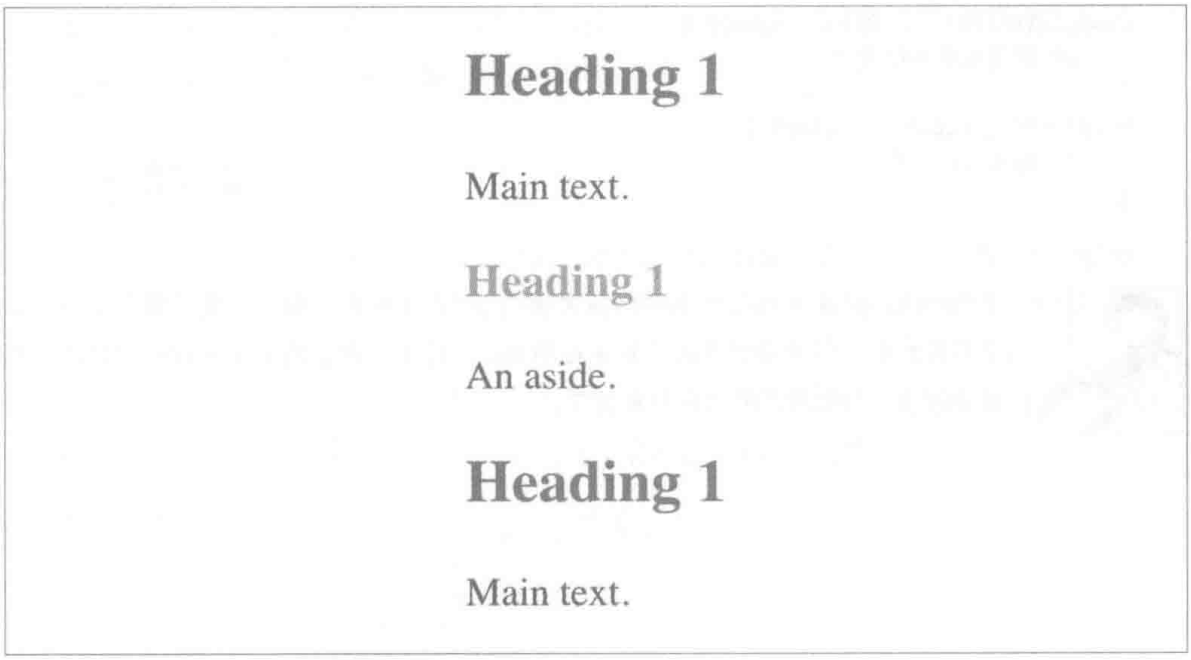


图 4-18：自定义值在特定的范围内起作用

注意，aside 元素外部的标题是紫色的，而内部的标题是橙色的。这是因为 aside 元素更新了 --base-color 变量的值，应用到 aside 元素中 h1 上的是新的自定义值。

虽然 CSS 变量只能替换值，但也有很多用途。下面的例子是 Chriztian Steinmeier 提供的，这个例子结合变量和 calc() 值类型，为无序列表创建有一定规律的缩进：

```
html {
    --gutter: 3ch;
    --offset: 1;
}
ul li {margin-left: calc(var(--gutter) * var(--offset));}
ul ul li {--offset: 2;}
ul ul ul li {--offset: 3;}
```

这与下面这样写其实是一样的：

```
ul li {margin-left: 3ch;}
ul ul li {margin-left: 6ch;}
ul ul ul li {margin-left: 9ch;}
```

不过使用变量更方便，只需在一处更新乘数 --gutter，其他地方都能自动更新，而不用重新输入三个值，还要确保计算正确。

使用抽象的变量名为样式规则的编写打开了全新的方式，这是以往很难实现的。如果想试试自定义属性，但又担心支持情况，可别忘了能查询特性的 @supports()。这样，万一遇到不支持的用户代理，使用变量的样式规则便能隐藏起来，不产生影响：

```
@supports (color: var(--custom)) {
    /* 使用变量的样式 */
}
@supports (--custom: value) {
    /* 替代样式 */
}
```



重申一次，自定义属性在本书即将完稿的 2017 年年末才出现，具体怎么使用、能用来做什么，以及与层叠的关系等还不确定。这是一个值得了解和尝试的特性，不过要注意，这里所讲的内容可能会变。

第 5 章

字体

CSS1 规范写于 1996 年，“Font Properties”（字体属性）一节的开头是这样说的，“设置字体属性是样式表最常见的用途之一。”这么多年过去了，这句话仍无法推翻。

CSS2 开始支持使用 `@font-face` 下载指定的自定义字体，不过直到 2009 年前后这个功能才以一致的方式被广泛支持。现在，借助 Typekit 等在线服务，网站可以使用任何想用的字体。一般来说，只要你有权使用一个字体，就能在设计中使用它。

然而，要注意，这并不表明你能完全控制字体。如果字体下载失败，或者字体的格式不被用户的浏览器支持，那么文本将使用后备字体显示。这是好事，因为即使指定的字体无法使用，依然能把内容呈现给用户。不过要注意的是，你不能完全依赖某个字体，千万不要觉得设计成什么样就会显示成什么样。

5.1 字体族

我们熟知的“字体”通常包含多个变体，分别针对粗体、斜体等。例如你可能熟悉（至少听说过）的 Times 字体，其实它有多种变体，包括 TimesRegular、TimesBold、TimesItalic、TimesBoldItalic 等。Times 的这些变体各自都是一个字型（font face），我们通常说的 Times，其实是这些不同字型的统称。也就是说，Times 其实是一个字体族（font family），而不是一个字体。大多数人都会把字体理解为单个实体。

为了覆盖所有情况，CSS 定义了五种通用字体族：

衬线字体

这种字体中的字形宽度各异，而且有衬线。因为字体中不同字符的尺寸不同，所以宽度才有差异。例如，小写字母 i 和小写字母 m 的宽度就不同。衬线是字符笔划末尾的装饰，例如小写字母 l 顶部和底部的短线，以及大写字母 A 两条竖线底部的短线。Times、Georgia 和 New Century Schoolbook 都是衬线字体。

无衬线字体

这种字体中的字形宽度各异，而且无衬线。Helvetica、Geneva、Verdana、Arial 和 Univers 都是无衬线字体。

等宽字体

等宽字体中的字形宽度一样。一般用于显示编程代码或表格数据。这种字体中的各个字符在横向上所占的空间是一样的；因此，尽管小写字母 i 和小写字母 m 这两个字母本身的宽度不同，但使用等宽字体时二者所占的横向空间一样大。这种字体可以有衬线，也可以无衬线。一个字体不管有没有衬线，只要字符宽度相同，就把它归类为等宽字体。Courier、Courier New、Consolas 和 Andale Mono 都是等宽字体。

草书字体

这种字体尝试模仿人类笔迹或手写体。通常，这种字体在笔划末端有较大的花饰，而且比衬线字体华丽。例如，大写字母 A 左边竖线的底部可能有个小卷，或者整条线都是花饰卷。Zapf Chancery、Author 和 Comic Sans 都是草书字体。

奇幻字体

这种字体（有时也叫“装饰”字体或“展示”字体）还真没什么统一的特征，不过可以确定的是，无法将其划归到其他类别中。Western、Woodblock 和 Klingon 都是奇幻字体。

理论上，任何一款字体都可归为上述通用字体族中的一个。实际情况可能并非如此，不过例外情况总是少的。如果浏览器遇到无法归类为衬线、无衬线、等宽或草书的字体，便会将其放到“奇幻”篮子里。

5.1.1 使用通用字体族

字体族使用 `font-family` 属性声明。

font-family	
取值	[*<family-name>* | *<generic-family>*]#
初始值	由用户代理指定
适用于	所有元素
计算值	指定的值
继承性	是
动画性	否

如果想让一个文档使用无衬线字体，但不介意具体使用哪一款，可以这样声明：

```
body {font-family: sans-serif;}
```

此时，由浏览器选择一款无衬线字体（例如 Helvetica），然后应用到 body 元素上。借助继承，body 的所有后代都将使用这个字体，除非特指度更高的选择符覆盖了这个属性。

利用这些通用字体族，创作人员可以编写十分复杂的样式表。下述规则得到的结果如图 5-1 所示。

```
body {font-family: serif;}
h1, h2, h3, h4 {font-family: sans-serif;}
code, pre, tt, kbd {font-family: monospace;}
p.signature {font-family: cursive;}
```

此时，文档中的大部分内容将使用衬线字体（例如 Times），不过 class 为 signature 的段落将使用草书字体（例如 Author）渲染。一级标题到四级标题将使用无衬线字体（例如 Helvetica），而 code、pre、tt 和 kbd 元素将使用等宽字体（例如 Courier）。

5.1.2 指定字体族

不过，创作人员可能想具体指定想使用哪款字体显示一个文档或元素。同样地，用户可能也想创建一个用户样式表，定义具体使用哪款字体显示文档。这两种情况仍然使用 font-family 属性。

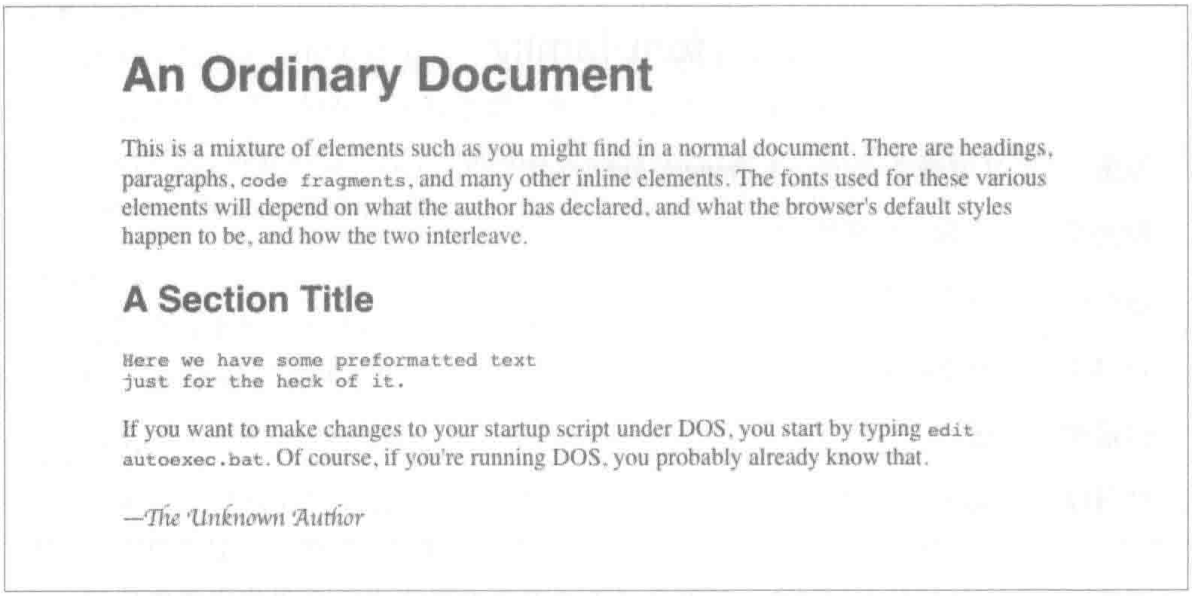


图 5-1：使用不同的字体族

假设现在想让所有 h1 都使用 Georgia 字体显示。实现这一需求最简单的规则如下：

```
h1 {font-family: Georgia;}
```

这样声明后，用户代理会使用 Georgia 字体显示文档中的所有 h1，如图 5-2 所示。



图 5-2：使用 Georgia 字体显示 h1 元素

这个规则假设用户代理中安装了 Georgia 字体。如果没有这款字体，用户代理不会使用这个规则。用户代理不会将其忽略，如果找不到名为“Georgia”的字体，将会使用用户代理的默认字体显示 h1 元素。

然而，不必灰心。结合字体名称和通用字体族，可以尽量让结果与预想接近。接着上例，对下述规则来说，如果用户代理中有 Georgia 字体，就使用 Georgia 字体，否则使用其他无衬线字体：

```
h1 {font-family: Georgia, serif;}
```

如果你没有安装 Georgia，但是安装了 Times，用户代理可能会使用 Times 显示 h1 元素。即便 Times 与 Georgia 不完全一样，但是足够接近。

鉴于此，笔者强烈建议始终在 font-family 规则中指定通用字体族。这样做相当于提供一种后备机制，在用户代理找不到匹配的字体时，选择一个字体代替。下面再举几个例子：

```
h1 {font-family: Arial, sans-serif;}
h2 {font-family: Charcoal, sans-serif;}
p {font-family: 'Times New Roman', serif;}
address {font-family: Chicago, sans-serif;}
```

对字体熟悉的人，可能记着多款相似的字体。假如我们想让文档中的所有段落都使用 Times 显示，不过 Times New Roman、Georgia、New Century Schoolbook 和 New York 也是可以接受的选择（这些都是衬线字体）。首先，定好各字体的顺序，然后一个接一个，以逗号分开：

```
p {font-family: Times, 'Times New Roman', 'New Century Schoolbook', Georgia,
     'New York', serif;}
```

用户代理会按照所列的顺序查找字体。如果找不到列出的任何一款字体，用户代理将选择一款可用的衬线字体。

使用引号

在上例中，你可能注意到了之前没见过的单引号。在 font-family 声明中，如果字体名称中有一个或多个空格（例如“New York”），或者字体名称中有符号（例如 # 或 $），建议使用引号。因此，名为 Karrank% 的字体就应该放在引号里：

```
h2 {font-family: Wedgie, 'Karrank%', Klingon, fantasy;}
```

如果没有引号，用户代理可能会忽略那个字体名称，然后接着处理规则后面的部分。

注意，把包含符号的字体名称放在引号里不是强制要求，只是推荐做法，相当于 CSS 规范中所说的最佳实践。 类似地，也推荐把包含空格的字体名称放在引号里。不过，一般来说这不是必须的，现代的用户代理通常不强制要求。然而，与 font-family 中可用的关键字同名的字体名称必须放在引号里。因此，如果有个字体的名称就是“cursive”，那么一定要把它放在引号里，这样才能与关键字 cursive 区分开：

```
h2 {font-family: Author, "cursive", cursive;}
```

只有一个词的字体名称无需放在引号里（前提是不与 font-family 中可用的关键字冲突），比如通用字体族的名称（serif、monospace 等）在确实指通用字体族时就从不放在引号里。如果把通用字体族的名称放在引号里，用户代理将假设你指定的是一款字体的名称（例如“serif”），而不是通用字体族的名称。

使用的引号既可以是单引号，也可以是双引号。注意，如果把 font-family 规则放在 style 属性中（一般不应该这么做），使用的引号要与属性使用的不同。因此，如果 font-family 规则放在双引号里，那么规则内部就要使用单引号，如下所示：

```
p {font-family: sans-serif;}  /* 设置段落使用无衬线字体 */

<!-- 这个示例是正确的（使用单引号）-->
<p style="font-family: 'New Century Schoolbook', Times, serif;">...</p>

<!-- 这个示例是错误的（使用双引号）-->
<p style="font-family: "New Century Schoolbook", Times, serif;">...</p>
```

此时使用双引号的话，将与属性的句法冲突，如图 5-3 所示。

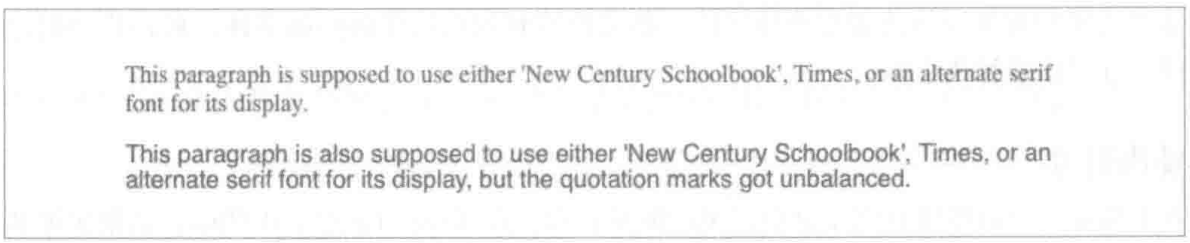


图 5-3：使用错误的引号导致的问题

5.2 使用 @font-face

@font-face 的作用是让你在设计中使用自定义的字体。这个特性首次出现在 CSS2 中，不过直到 21 世纪头十年的后半期才实现。尽管无法保证每个终端用户都能用上你指定的字体，但是这个特性却得到了广泛支持。

假设你想使用的字体没有广泛安装，而是个十分特别的字体，借助 @font-face 的魔力，你可以定义一个专门的字体族名称，对应于服务器上的一个字体文件。用户代理将下载那个文件，使用它渲染页面中的文本，就好像用户的设备中安装了那个字体一样。下面举个例子：

```
@font-face {
    font-family: "SwitzeraADF";
```

```
    src: url("SwitzeraADF-Regular.otf");
}
```

用户代理见到 font-family: SwitzeraADF 声明后，会加载对应的 .otf 文件，然后使用它渲染文本。



本节示例中的 SwitzeraADF 是由 Arkandis Digital Foundry（*http://arkandis.tuxfamily.org/openfonts.html*）制作的字型集合。

@font-face 是惰性加载字型的。这表明，仅当需要使用指定的字型渲染文本时，才会加载，否则不加载。其实，浏览器不管是否需要，都会先行下载声明的全部字型，这是浏览器的缺陷。

5.2.1 必须的描述符

定义字体的全部参数都在 @font-face { } 结构中编写。这些参数称为描述符(descriptor)，与属性十分相似，格式为 descriptor: value;。其实，多数描述符都直接使用现有的属性名，稍后将做详细说明。

描述符中有两个是必需的：font-family 和 src。

font-family	
取值	*<family-name>*
初始值	未定义

src	
取值	[[*<uri>* [format(*<string>*#)]?] | *<font-face-name>*]#
初始值	未定义

src 的作用不言而喻：为定义的字型提供一个或多个源。如果有多个源，之间以逗号分隔。字型的源可以指向任何 URI，不过有个限制：字型必须与样式表同源。因此，不能把 src 指向别人的网站，下载别人的字体。你要在自己的服务器中存储一份本地副本，或者使用同时提供样式表和字体文件的字体托管服务。



同源限制有个例外：使用 HTTP 首部 Access-Control-Allow-Origin 设定服务器，允许跨站加载。

你可能觉得奇怪，这里的 font-family 和前一节所讲的有何不同？区别是，这里的 font-family 是字体族描述符，而前一节中的 font-family 是字体族属性。如果还不理解，没关系，先这样，慢慢你就会懂的。

其实，@font-face 做的是低层定义，是为字体相关的属性（如 font-family）服务的。通过描述符 font-family: "SwitzeraADF"; 定义一个字体族名称之后，用户代理的字体族名称表中便会出现“SwitzeraADF”条目，与 Helvetica、Georgia、Courier 等具有同等地位，可以在 font-family 属性的值中引用：

```
@font-face {
    font-family: "SwitzeraADF";   /* 描述符 */
    src: url("SwitzeraADF-Regular.otf");
}
h1 {font-family: SwitzeraADF, Helvetica, sans-serif;}  /* 属性 */
```

注意，font-family 描述符的值和 font-family 属性中出现的那个字体族名是一样的。如果不一样，h1 规则将忽略 font-family 属性值中列出的第一个字体族名称，解析后一个。只要成功下载了字体文件，而且文件的格式是用户代理支持的，那个字体就会像其他字体一样用于渲染文本，如图 5-4 所示。

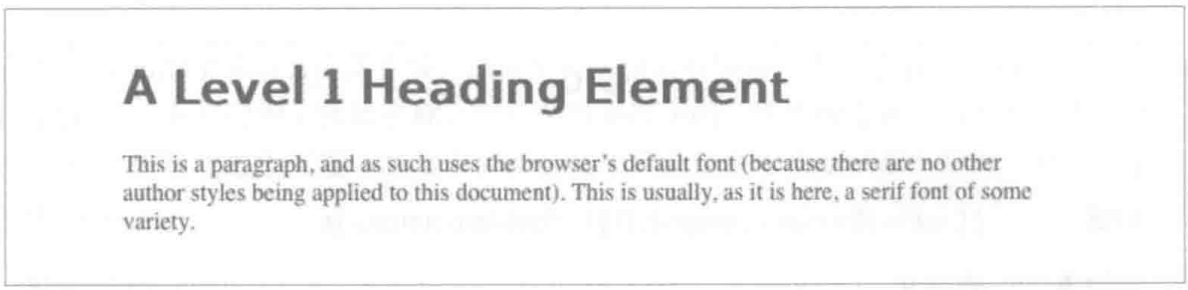


图 5-4：使用一款下载的字体

类似地，src 描述符中以逗号分隔的列表用于提供后备字体文件。如此一来，一旦（不管什么原因）用户代理无法从第一个源下载，就会尝试从下一个源中下载字体文件：

```
@font-face {
    font-family: "SwitzeraADF";
    src: url("SwitzeraADF-Regular.otf"),
         url("/fonts/SwitzeraADF-Regular.otf");
}
```

注意，这里也适用同源策略，如果指向其他服务器，通常无法下载，除非设置服务器允许跨源访问。

如果想告诉用户代理所用的字体是什么格式，可以使用可选的 format()：

```
@font-face {
    font-family: "SwitzeraADF";
    src: url("SwitzeraADF-Regular.otf") format("opentype");
}
```

这样做的好处是，让用户代理跳过不支持的字体格式，从而减少带宽用量，提升加载速度。此外，使用 format() 还能为不带有常规扩展名的字体文件指定格式，以防用户代理不识别：

```
@font-face {
    font-family: "SwitzeraADF";
    src: url("SwitzeraADF-Regular.otf") format("opentype"),
         url("SwitzeraADF-Regular.true") format("truetype");
}
```

刹那间的变化

对从事设计或开发有一定年头的人来说，可能还记得以前有刹那间出现无样式内容的日子。早期的浏览器在加载完 CSS 之前，至少是还未通过 CSS 确定布局之前，先显示 HTML。因此，有那么一瞬间，浏览器会先显示未经 CSS 装饰的文本（使用浏览器的默认样式）。

使用自定义字体时也有类似的问题，即刹那间出现未使用指定字体渲染的文本。这一瞬间发生在浏览器加载完页面和 CSS 布局之后，但是尚未加载完自定义的字体之前。此时，文本使用默认字体或后备字体渲染，然后才会变成使用自定义字体渲染。

因为换成自定义字体的过程可能会影响布局，所以创作人员应该谨慎选择后备字体。如果开始使用的字体和后来使用的自定义字体之间在高度上相差较大，页面势必要重新布局。没什么方法能自动规避这一问题，不过 font-size-adjust（稍后介绍）在支持它的用户代理中有一定作用。建议根据想用的自定义字体选择一款高度接近的字体。

出现这一霎那间变化的主要原因一直没变：浏览器在获得所需的全部资源之前已经做好了显示内容的准备，因此浏览器先行显示，然后再替换成经过装饰的版本。霎那间出现无样式内容的问题最终解决了，相信有一天霎那间出现未使用指定字体的现象也能顺利解决。在此之前，我们要面对现实，接受这一问题的存在。

可用的格式值见表 5-1（截至 2017 年年末）。

表 5-1：支持的字体格式值

值	格式
embedded-opentype	EOT (Embedded OpenType)
opentype	OTF (OpenType)
svg	SVG (Scalable Vector Graphics)
truetype	TTF (TrueType)
woff	WOFF (Web Open Font Format)

除了使用 url() 和 format() 组合之外，还可以使用 local()（名称表明了作用）指定已经安装在用户设备中的字体族名称（可以是多个）：

```
@font-face {
    font-family: "SwitzeraADF";
    src: local("Switzera-Regular"),
         local("SwitzeraADF-Regular "),
         url("SwitzeraADF-Regular.otf") format("opentype"),
         url("SwitzeraADF-Regular.true") format("truetype");
}
```

这里，用户代理先检查设备中是否有名为“Switzera-Regular”或“SwitzeraADF-Regular”的字体族，如果有，使用 SwitzeraADF 这个名称指代本地安装的字体；如果没有，尝试从远端下载 url() 中指定的字体文件。

注意，借助这个功能可以为本地安装的字体自定义名称。例如，可以像下面这样为Helvetica（没有这个字体的话，尝试 Helvetica Neue）起个简短的名称：

```
@font-face {
    font-family: "H";
    src: local("Helvetica"), local("Helvetica Neue");
}

h1, h2, h3 {font-family: H, sans-serif;}
```

只要用户的设备中安装有 Helvetica，前三级标题便将使用 Helvetica 渲染。这样做看起来有点多此一举，不过在某些情况下确实能减少样式表文件的大小。

万全之策

@font-face 有个棘手的问题要解决：不同时代的不同浏览器支持不同格式的字体（从表 5-1 中那些可下载的字体格式可见一斑）。为了尽量涵盖较广的场景，应该使用能确保万全的 @font-face 句法。这种句法最初由 Paul Irish 提出，后经 FontSpring 的人员改进，写法如下：

```
@font-face {
    font-family: "SwitzeraADF";
    src: url("SwitzeraADF-Regular.eot");
    src: url("SwitzeraADF-Regular.eot?#iefix") format("embedded-opentype"),
         url("SwitzeraADF-Regular.woff") format("woff"),
         url("SwitzeraADF-Regular.ttf") format("truetype"),
         url("SwitzeraADF-Regular.svg#switzera_adf_regular") format("svg");
}
```

下面详细说明。开头的部分，即指定 font-family 名称那一行无需过多解释。随后的两行：

```
src: url("SwitzeraADF-Regular.eot");
src: url("SwitzeraADF-Regular.eot?#iefix") format("embedded-opentype"),
```

这两行为支持 EOT（Embedded OpenType）格式的浏览器（IE6~IE9）提供 EOT 文件。前一行针对“兼容模式”下的 IE9，后一行针对 IE6~IE8。后一行中的 ?#iefix 导致这些浏览器出现一个解析缺陷，从而绕过另一个解析缺陷，即列出多个字体格式时返回 404 响应。IE9 修正了这个问题，但是没有扩充支持的字体格式，因此才需要第一行。

```
url("SwitzeraADF-Regular.woff") format("woff"),
```

这一行为支持 Web Open Font Format 的浏览器（包含多数现代浏览器）提供 .woff 文件。其实，至此已经涵盖大多数桌面用户了。

```
url("SwitzeraADF-Regular.ttf") format("truetype"),
```

这一行为多数 iOS 和 Android 设备提供所支持的字体格式，涵盖了多数手持设备用户。

```
url("SwitzeraADF-Regular.svg#switzera_adf_regular") format("svg");
```

最后一行提供只有旧 iOS 设备支持的字体格式，涵盖余下的多数手持设备用户。

如果字形很多，很快就会变得烦琐，即便只输入一遍也很累手腕。幸好，有些服务能根据你提供的字型生成所需的 @font-face，并把字型转换成不同的格式，然后提供一个文件包供你下载。这些服务中最好的一个是 Font Squirrel 的 @Font-Face Kit Generator（*http://fontsquirrel.com/fontface/generator*）。使用这个生成工具时要谨记一点，必须得到转换和使用字型的授权（详情参见“自定义字体的注意事项”旁注）。

5.2.2 其他字体描述符

除了必须的 font-family 和 src 描述符，还有几个可选的描述符用于为字型指定属性值。与 font-family 一样，这些描述符也对应于现有的 CSS 属性（本章后文说明），控制着用户代理相应属性的处理方式。见表 5-2。

自定义字体的注意事项

使用自定义的字体时要注意两件事。第一，要有权在网页中使用指定的字体；第二，值不值得使用自定义字体。

与图片库一样，字体族也有许可证，规定使用范围，不是所有字体都准许在 Web 中使用。如果想彻底规避这个问题，可以使用 FOSS（Free and Open-Source Software）字体，或者使用 Fontdeck 或 Typekit 这样的商业服务，让它们处理许可和字体格式转换。否则，一定要确保自己有权按照想要的方式使用字型，这跟你要保证自己有恰当的授权使用购买的图像一样。

此外，使用的字型越多，Web 服务器要处理的资源越多，而且页面越大。多数字型的体积并不大，通常为 50~100KB，不过，为了华丽的效果，字型会越用越多，而且比较复杂的字型体积是很大的。可以想象，这与图像如出一辙。一如往常，在外观和性能上要做适当的权衡，具体向哪边倾斜要视情况而定。

我们知道图像有优化工具，同样，字体也有优化工具。这些工具通常是做子集处理，即只在字体中留下确实需要用到的符号。Typekit 或 Fonts.com 等服务可能有这样的工具，或者在请求字体时动态处理。

表 5-2：字体描述符

描述符	默认值	说明
font-style	normal	区分常规、斜体和倾斜字型
font-weight	normal	区分不同的字重（例如加粗）
font-stretch	normal	区分不同的字符宽度（例如紧缩和加宽）
font-variant	normal	区分众多字形变体（例如小号大写字母），在很多方面与 CSS 中的 font-feature-settings 很像
font-feature-settings	normal	直接访问 OpenType 的低层特性（例如启用连字）
unicode-range	U+0-10FFFF	定义指定字体中可用的字符范围

这些字体描述符是可选的，不必一定在 @font-face 规则中列出。CSS 规定，描述符不像属性那样可以没有默认值。如果没有某个可选的描述符，它的值将被设为默认值。因此，如果未列出 font-weight，其值默认为 normal。

限制字符范围

有一个字体描述符没有对应的 CSS 属性（与表 5-2 中的其他描述符不同），即 unicode-range。这个描述符用于指定自定义字体可以应用到哪些字符上。使用符号字体，或者想确保只有特定语言使用指定字型时用得到这个描述符。

unicode-range

取值	*<urange>#*
初始值	U+0-10FFFF

默认情况下，这个描述符的值涵盖全部 Unicode 字符。这表明，只要字体中有某个字符的字形，就能用它渲染那个字符。多数情况下，这正是我们想要的。然而，有时我们想使用特定的字型渲染特定的内容。下面是从 CSS Fonts Module Level 3 规范中摘取的两个例子：

```
unicode-range: U+590-5FF;  /* 希伯来语字符 */
unicode-range: U+4E00-9FFF, U+FF00-FF9F, U+30??;  /* 日语汉字、平假名、片假名 */
```

第一个例子只指定了一个范围，从 Unicode 码位 590~5FF。这个范围是希伯来语字符。因此，创作人员可以指定一个希伯来语字体，限制它只用于渲染希伯来语字符，即使字型中还包括其他码位的字形：

```
@font-face {
    font-family: "CMM-Ahuvah";
    src: url("cmm-ahuvah.otf" format("opentype");
    unicode-range: U+590-5FF;
}
```

第二个例子指定了多个范围，以逗号分隔，涵盖所有日语字符。里面奇怪的 U+30?? 值是 unicode-range 允许使用的特殊格式，问号是通配符，意思是“任何数字”。因此，U+30?? 等效于 U+3000-30FF。unicode-range 的值中只允许使用问号这一个“特殊的”字符。

范围必须从小到大，反过来（例如 U+400-300）会导致解析错误而被忽略。除了范围之外，还可以声明单个码位，例如 U+221E。单个码位通常与其他码位和范围结合起来使用，比如像下面这样：

```
unicode-range: U+4E00-9FFF, U+FF00-FF9F, U+30??, U+A5;
  /* 日语汉字、平假名、片假名，外加货币符号 ¥*/
```

可以指定一个码位，让指定的字体渲染唯一的字符。要不要这么做取决于你自己、你的设计、字体文件的大小和用户的网速。

@font-face 是惰性加载的，因此可以使用 unicode-range 限制只下载页面中真正需要用到的字型。假设一个网站中有英语、俄语，还有一些基本的算术运算符，而你并不知道页面中会出现哪些字符。有些页面可能全是英语，有些可能混杂着俄语和算式等。此外，假设这三种内容都用专门的字型。为了确保用户代理只下载真正需要的字型，可以像下面这样组织 @font-face 规则：

```
@font-face {
    font-family: "MyFont";
    src: url("myfont-general.otf" format("opentype");
}
@font-face {
    font-family: "MyFont";
    src: url("myfont-cyrillic.otf" format("opentype");
    unicode-range: U+04??, U+0500-052F, U+2DE0-2DFF, U+A640-A69F, U+1D2B-1D78;
```

```
}
@font-face {
    font-family: "MyFont";
    src: url("myfont-math.otf" format("opentype");
    unicode-range: U+22??;   /* 等效于 U+2200-22FF */
}
```

第一个规则没有指定 Unicode 范围，因此始终下载，除非页面中恰巧什么字符也没有（不是不可能）。根据第二个规则，仅当页面中有指定 Unicode 范围内的字符时才下载 myfont-cyrillic.otf。第三个规则一样，不过是当页面中有基本的算术运算符时才下载。

5.2.3 组合描述符

我们可以把多个描述符组合在一起为字型设定不同的属性，这一点没那么容易想到，不过确实可行。例如，可以指定一个字型为粗体，一个字型为斜体，再指定一个字型为加粗的斜体。

这么做背后的原理是，未声明的描述符使用默认值。以下面三个字型设定规则为例：

```
@font-face {
    font-family: "SwitzeraADF";
    font-weight: normal;
    font-style: normal;
    font-stretch: normal;
    src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-weight: bold;
    font-style: normal;
    font-stretch: normal;
    src: url("SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-weight: normal;
    font-style: italic;
    font-stretch: normal;
    src: url("SwitzeraADF-Italic.otf") format("opentype");
}
```

这里我们明确列出了所需的描述符，即便使用默认值，也没有省略。把值为 normal 的描述符去掉之后，效果完全一样：

```
@font-face {
    font-family: "SwitzeraADF";
    src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-weight: bold;
    src: url("SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-style: italic;
    src: url("SwitzeraADF-Italic.otf") format("opentype");
}
```

在这三个规则中，所有 font-stretch 的值都是 normal，而 font-weight 和 font-style 的值各异。如果想让加粗斜体字不拉伸呢？

```
@font-face {
    font-family: "SwitzeraADF";
    font-weight: bold;
    font-style: italic;
    font-stretch: normal;
    src: url("SwitzeraADF-BoldItalic.otf") format("opentype");
}
```

如果想得到紧缩的加粗斜体呢？

```
@font-face {
    font-family: "SwitzeraADF";
    font-weight: bold;
    font-style: italic;
    font-stretch: condensed;
    src: url("SwitzeraADF-BoldCondItalic.otf") format("opentype");
}
```

如果想得到紧缩的正常字重斜体呢？

```
@font-face {
    font-family: "SwitzeraADF";
    font-weight: normal;
    font-style: italic;
    font-stretch: condensed;
    src: url("SwitzeraADF-CondItalic.otf") format("opentype");
}
```

可行性还有很多，我们就此打住。如果把所有取值为 normal 的描述符都去掉，得到的结果如下，效果如图 5-5 所示。

```
@font-face {
    font-family: "SwitzeraADF";
    src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-weight: bold;
    src: url("SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-style: italic;
    src: url("SwitzeraADF-Italic.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-weight: bold;
    font-style: italic;
    src: url("SwitzeraADF-BoldItalic.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-weight: bold;
    font-stretch: condensed;
    src: url("SwitzeraADF-BoldCond.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-style: italic;
    font-stretch: condensed;
    src: url("SwitzeraADF-CondItalic.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-weight: bold;
    font-style: italic;
    font-stretch: condensed;
    src: url("SwitzeraADF-BoldCondItalic.otf") format("opentype");
}
```

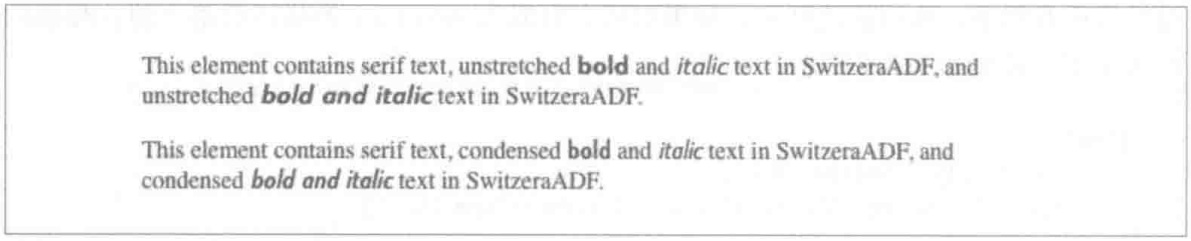


图 5-5：使用不同的字型

可以看出，只这三个描述符就有这么多组合方式，毕竟 font-weight 有 11 种可能的值，font-stretch 有 10 种可能的值，不过不一定都用得到。其实，多数字体族不会像 SwitzeraADF 这样提供如此多的字型（共计 24 种），因此没必要所有组合都写出来。然而，你要知道可以这么做，以防某些情况下需要使用特殊的字体渲染紧缩的加粗文本，避免用户代理自行计算。

5.3 字重

讲完 @font-face 及其描述符之后，回到字体属性。基本上，我们都用过正常和加粗的文本，这是两个最基本的字重。CSS 的 font-weight 属性可以精确控制字重。

font-weight

取值	normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900
初始值	normal
适用于	所有元素
计算值	其中一个数值（100 等），或者一个数值加一个相对值（bolder 或 lighter）
备注	有对应的 @font-face 描述符
继承性	是
动画性	否

一般来说，字重越大，字体越黑、越粗。标识字型粗细的方式有很多种。例如，SwitzeraADF 字体族有很多变体，例如 SwitzeraADF Bold、SwitzeraADF Extra Bold、SwitzeraADF Light 和 SwitzeraADF Regular，这些变体的字形基本一样，不过字重各不相同。

假如想在文档中使用 SwitzeraADF 不同粗细的变体，可以直接通过 font-family 属性指定，但是完全没必要这么做。毕竟编写下面这样的样式表没什么乐趣可言：

```
h1 {font-family: 'SwitzeraADF Extra Bold, sans-serif;}
h2 {font-family: 'SwitzeraADF Bold, sans-serif;}
h3 {font-family: 'SwitzeraADF Bold', sans-serif;}
h4, p {font-family: SwitzeraADF Regular, sans-serif;}
small {font-family: 'SwitzeraADF Light', sans-serif;}
```

这样写很繁琐。更合理的做法是，为整个文档指定一个字体族，然后为不同的元素设定不同的字重。为此，可以使用 @font-face 和不同的 font-weight 值。下面是一个十分简单的 font-weight 声明：

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

这个声明的意思是，b 元素应该使用粗体字型显示，即比常规字型粗一些的字型。一般来说这就是我们想要的行为，毕竟 b 元素就是用于加粗文本的。

其实在背后，显示 b 元素时会使用一个较粗的字型。因此，如果一个段落使用 Times 显示，其中有部分加粗文本，那么这个段落其实是使用同一字体的两个字型显示的：Times 和 TimesBold。常规文本使用 Times 显示，加粗文本使用 TimesBold 显示。

5.3.1 字重的工作方式

为了弄清用户代理如何确定一个字体变体的粗细（或字重），先要理解关键字 100 到 900。这些数字关键字对应于字体设计中的九级字重。如果一个字体族中有全部九级字重，那么这些数字就直接对应于预定义的级别，100 是最细的，900 是最粗的。

其实，这些数字并不表示字重本身。CSS 规范只是说，每个数字对应的权重至少和前面的数字具有相同的字重。因此，100、200、300 和 400 可能都对应于同样细的变体；500 和 600 对应于同样粗的变体；700、800 和 900 则对应同样较粗的变体。只要后面的数字关键字对应的粗细不比前面的数字关键字细就行。

一般，这些数字对应于常见的变体名称（先不考虑 font-weight 的其他值）。400 对应于 normal，700 对应于 bold。其他数字不与 font-weight 的其他关键字对应，不过可以对应于常见的变体名称。在字体的变体中，以“Normal”“Regular”“Roman”或“Book”等表示的可以分配给数字 400，以“Medium”表示的可以分配给数字 500。然而，如果只有一个变体，而且以“Medium”表示，那么应该分配给 400，而非 500。

如果给定的字体族中字重的等级少于 9 个，用户代理要做更多工作。遇到这种情况，用户代理必须填补既定方式的空缺：

- 如果 500 未分配，与 400 对应的字重一样。
- 如果 300 未分配，将其对应于比 400 细的那个变体。如果没有这样一个变体，字重与 400 一样。此时，通常为“Normal”或“Medium”变体。200 和 100 也是这样处理的。
- 如果 600 未分配，将其对应于比 500 黑的下一个变体。如果没有这样一个变体，字重与 500 一样。700、800 和 900 也是这样处理的。

为了更清楚地说明这种字重处理机制，下面看三个指定字重的例子。第一个例子假设 Karrank% 字体族是 OpenType 字体，定义了 9 级字重。此时，9 个数字分别对应 9 级字重，而且关键字 normal 和 bold 分别对应数字 400 和 700。这是最简单的情况，现实中很少遇见（很少有字体族会提供 9 级字重，如果提供的话，价格通常不菲）。

第二个例子假设字体族为本节开头用过的 SwitzeraADF。假设 SwitzeraADF 的变体按照表 5-3 所示的对应关系分配了数值。

表 5-3：一个字体族假设的字重分配情况

字型	分配的关键字	分配的数字
SwitzeraADF Light		100，200，300
SwitzeraADF Regular	normal	400
SwitzeraADF Medium		500
SwitzeraADF Bold	bold	600，700
SwitzeraADF Extra Bold		800，900

前三个数值分配给最细的字型。“Regular”字型对应关键词 normal，对应数字 400。因为有“Medium”字型，所以分配给数字 500。没有字型可分配给 600，因此将其对应到“Bold”字型上。“Bold”字型还对应 700 和 bold。最后，800 和 900 分配给“Extra

Bold”字型。注意，仅当这个字型是最高的两级字重时，才会这样分配。否则，用户代理可能忽略这个字型，把 800 和 900 分配给“Bold”字型或其他变体。

最后一个例子以精简版 Times 为例。在表 5-4 中，只有两个字重变体：“TimesRegular”和“TimesBold”。

表 5-4：“Times”假设的字重分配情况

字型	分配的关键字	分配的数字
TimesRegular	normal	100, 200, 300, 400, 500
TimesBold	bold	600, 700, 800, 900

关键字 `normal` 和 `bold` 分别分配给常规字重和加粗字重，这与预期是一样的。数字关键字方面，100 到 300 分配给“Regular”字型，因为没有更细的字型了。400 也分配给“Regular”字型，这也符合预期，但是 500 呢？它也分配给“Regular”（或 normal）字型，因为没有“Medium”字型。根据前述机制，将其分配给与 400 一样的字型。余下的数字关键字，700 分配给 bold，而由于缺少更粗的字型，所以把 800 和 900 分配给下一个较细的字型，即“Bold”。最后，600 分配给下一个较粗的字型，即“Bold”。

`font-weight` 可以继承，因此如果把一个段落的字重设为 `bold`：

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

那么，这个段落的所有子代都将继承粗体，如图 5-6 所示。

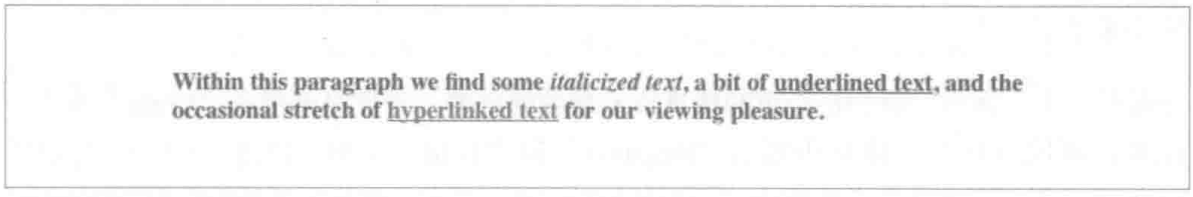


图 5-6：继承字重

上述机制没什么反常之处，不过讲到最后两个关键字（`bolder` 和 `lighter`）时，你会发现有些不同往常的地方。一般来说，这两个关键字的效果跟你预期的一样：渲染的文本比父元素的字重更重或更轻。先来看 `bolder`。

5.3.2 增大字重

如果把一个元素的字重设为 `bolder`，用户代理首先要确定从父元素继承的 `font-weight` 值是什么，然后选择比继承的字重高一级的最小数字。如果找不到，用户代理把元素的

字重设为下一个数字值，直到 900；到顶后，就把字重设为 900。下述各种情况的渲染结果如图 5-7 所示。

```
p {font-weight: normal;}
p em {font-weight: bolder;}  /* 文本为粗体，求值结果为 700 */

h1 {font-weight: bold;}
h1 b {font-weight: bolder;}  /* 如果没有更粗的字型，其值结果为 800 */

div {font-weight: 100;} /* 假设有 "Light" 字型，说明见下文 */
div strong {font-weight: bolder;} /* 文本为常规字型，字重为 400 */
```

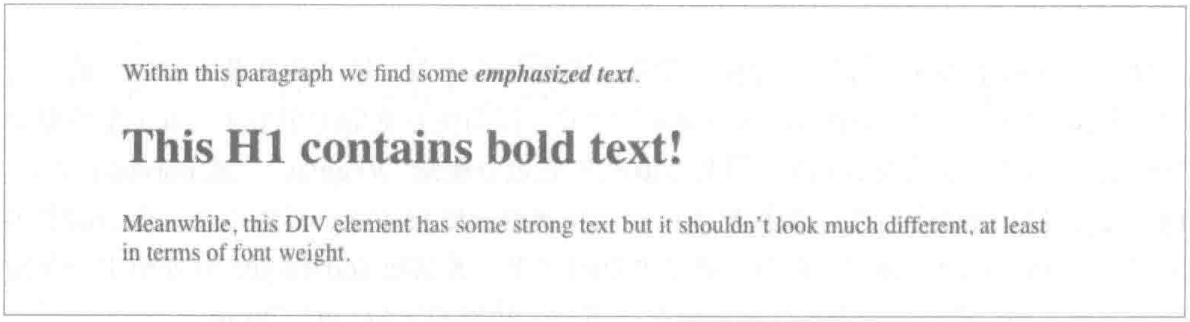


图 5-7：把文本变得更粗

在第一个示例中，用户代理顺着字重等级向上爬，把 normal 变成 bold。用数字表示，就是从 400 变成 700。在第二个示例中，h1 的字重已经设为 bold。如果没有更粗的字型，用户代理会把 h1 中 b 的字重设为 800，因为这是从 700（等同于 bold）起的下一级。因为 800 和 700 对应同一个字型，所以 h1 中常规的文本和加粗的文本在视觉上没有区别，但字重是不同的。

在最后一个示例中，我们把段落的字重设为最细的字型，而且假设存在“Light”变体。此外，再假设这个字体族中还有“Regular”和“Bold”字型。段落中的 em 将使用 normal 字型，因为这是字体族中下一个较粗的字型。然而，如果字体族中只有“Regular”和“Bold”字型呢？此时，声明的求值结果如下：

```
/* 假设只有 "Regular" 和 "Bold" 两个字型 */
p {font-weight: 100;}   /* 看起来跟 normal 字重一样 */
p span {font-weight: bolder;}   /* 对应于 700 */
```

可以看出，100 被分配给 normal 字型，不过其字重仍是 100。因此，p 元素中的 span 将继承这个 100，经计算得到的下一个较粗的字型是“Bold”，其对应的数字字重为 700。

下面更进一步，增加两个规则和一些标记，演示处理过程（结果见图 5-8）：

```
/* 假设只有 "Regular" 和 "Bold" 两个字型 */
p {font-weight: 100;}   /* 看起来跟 normal 字重一样 */
p span {font-weight: 400;}   /* 同样如此 */
strong {font-weight: bolder;}   /* 比父元素更粗 */
strong b {font-weight: bolder;}   /* 继续加粗 */

<p>
This paragraph contains elements of increasing weight: there is a
<span>span element that contains a <strong>strongly emphasized
element and a <b>bold element</b></strong></span>.
</p>
```

This paragraph contains elements of increasing weight: there is an *emphasized element which contains a* ***strongly emphasized element, and that contains a bold element***.

图 5-8：字重不断增加

对后两个嵌套元素来说，font-weight 经计算得到的值逐渐增加，因为两个地方都用了关键字 bolder。如果把段落中的文本换成各元素对应的 font-weight 数字，得到的结果如下：

```
<p>
100 <span> 400 <strong> 700 <b> 800 </b> </strong> </span>.
</p>
```

前三个字重相比，增加的幅度较大，分别从 100 增加到 400，以及从 400 增加到 bold（700）。因为没有比 700 更粗的字型了，所以用户代理把 font-weight 的值上移一个数字级别（800）。此外，如果在 b 元素中插入一个 strong 元素，得到的结果如下：

```
<p>
100 <span> 400 <strong> 700 <b> 800 <strong> 900
</strong> </b> </strong> </span>.
</p>
```

如果最内层的 strong 元素中还有一个 b 元素，那个 b 元素的字重将变成 900，因为 font-weight 永远不能大于 900。假设只有两个字型，那么文本要么是常规粗细，要么是加粗的，如图 5-9 所示。

```
<p>
regular <span> regular <strong> bold <b> bold <strong> bold </strong> </b>
</strong> </span>.
</p>
```



图 5-9：视觉上看到的字重（以数值表示）

5.3.3 减小字重

你可能猜到了，lighter 的工作方式类似，只不过是让用户代理向下减小字重。简单修改一下前面的示例，lighter 的作用就十分明确了：

```
/* 假设只有 "Regular" 和 "Bold" 两个字型 */
p {font-weight: 900;}   /* 尽量使用最粗的，看起来跟 bold 一样 */
p span {font-weight: 700;}   /* 也是 bold */
strong {font-weight: lighter;}   /* 比父元素细一些 */
b {font-weight: lighter;}   /* 继续变细 */

<p>
900 <span> 700 <strong> 400 <b> 300 <strong> 200
</strong> </b> </strong> </span>.
</p>
<!-- 或者用关键字表示 -->
<p>
bold <span> bold <strong> regular <b> regular <strong> regular </strong></b>
</strong></span>.
</p>
```

先不管符不符合常理，从图 5-10 中可以看出，段落中主文本的字重是 900。把 strong 中文本的字重设为 lighter 后，得到的是较细的字型，即这里的常规字型，以数值衡量的话，是 400（等同于 normal）。接下来减小为 300，这也相当于 normal，因为没有更细的字型存在。自此之后，用户代理一次只能减少一个字重等级，直到 100（上例中没有体现）。第二个段落展示哪些文本为粗体，哪些为正常粗细。



图 5-10：文本逐渐变细

5.3.4 font-weight 描述符

使用 font-weight 描述符可以为字型指定 font-weight 属性支持的字重等级。例如，下述规则为五个字型分配六个不同的字重：

```
@font-face {
    font-family: "SwitzeraADF";
    font-weight: normal;
    src: url("f/SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-weight: bold;
    src: url("f/SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-weight: 300;
    src: url("f/SwitzeraADF-Light.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-weight: 500;
    src: url("f/SwitzeraADF-DemiBold.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-weight: 700;
    src: url("f/SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-weight: 900;
    src: url("f/SwitzeraADF-ExtraBold.otf") format("opentype");
}
```

这样分配之后，创作人员便可使用多个字重等级。下述规则的结果如图 5-11 所示。

```
h1, h2, h3, h4 {font: 225% SwitzeraADF, Helvetica, sans-serif;}
h1 {font-weight: 900;}
h2 {font-size: 180%; font-weight: 700;}
h3 {font-size: 150%; font-weight: 500;}
h4 {font-size: 125%; font-weight: 300;}
```

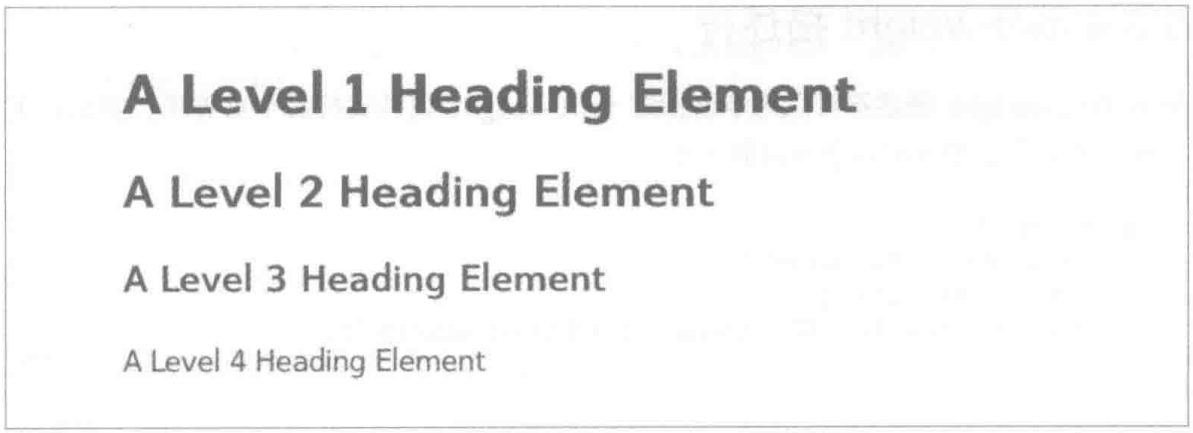


图 5-11：使用分配了字重的字型

不管何时，用户代理都会使用 5.3.1 节所述的解析算法根据 font-weight 属性的值选择字型。font-weight 描述符的值可以是 font-weight 属性可取的任何一个值，但 inherit 关键字除外。

5.4 字号

我们对确定字号的方法十分熟悉，但又让人觉得特别陌生。

font-size

取值	xx-small | x-small | small | medium | large | x-large | xx-large | smaller | larger | *<length>* | *<percentage>*
初始值	medium
适用于	所有元素
百分数	根据父元素的字号计算
计算值	一个绝对长度
继承性	是
动画性	是（仅限数字关键字）

类似于 font-weight 属性的关键字 bolder 和 lighter，font-size 属性也有相对大小关键字，分别为 larger 和 smaller。这两个关键字的作用与相对字重类似，分别增大和减小一号字体大小。不过，我们首先要知道字号是如何确定的。

其实，font-size 属性与渲染结果之间的关系由字体设计者决定。这个关系在字体中通过 *em* 方框（或 *em* 盒子）表示。*em* 方框（以及字号）与字体中字符的边界没有关系，其实它指的是在没有行距（CSS 中的 line-height）的情况下两条基线之间的距离。字体中完全有可能存在高度超过基线之间距离的字符。鉴于此，设计字体时要确保所有字符都比 em 方框小。事实上，多数字体就是这么做的。图 5-12 是几个虚构的例子。

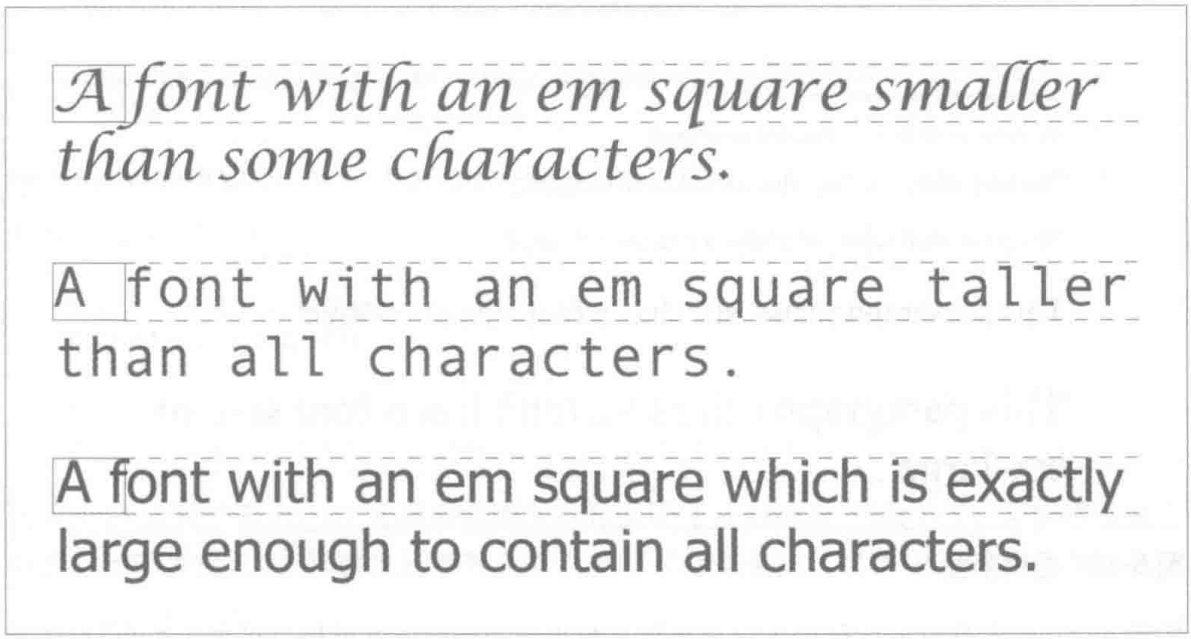


图 5-12：字体中的字符和 em 方框

因此，font-size 的作用是为字体的 em 方框提供一个尺寸。所以，显示出来的字符不可能完全是指定的大小。

5.4.1 绝对大小

掌握这些基础知识之后，来看绝对大小关键字。font-size 支持的绝对大小值有七个：xx-small、x-small、small、medium、large、x-large 和 xx-large。这几个关键字没有固定的大小，而是相对而言的，如图 5-13 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;}
```

根据 CSS1 规范，这些绝对大小之间相差的倍数（或叫换算系数）是向上 1.5、向下 0.66。因此，如果 medium 相当于 10px，那么 large 应该是 15px。后来，人们觉得换算系数太大，因此 CSS2 建议取 1.0~1.2，而 CSS3 草案提供了更为复杂的计算方式（例如，small 是 medium 的 8/9，xx-small 是 medium 的 3/5）。不过，这些只是简易的换算系数，用户代理随时可以调整。

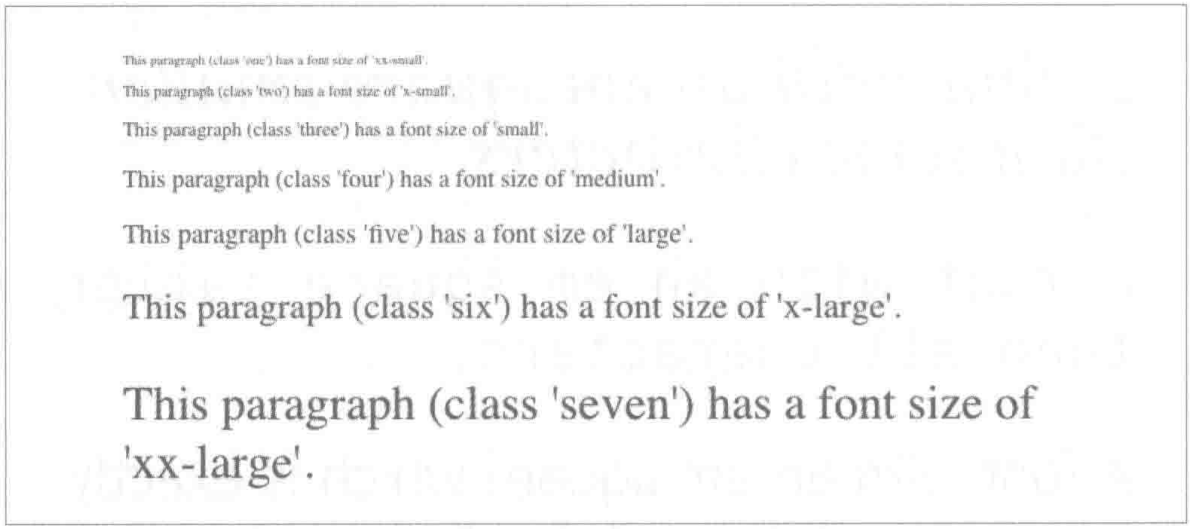


图 5-13：绝对字号

假设 medium 等于 16px，使用不同的换算系数得到的绝对值对照情况见表 5-5（已经四舍五入为整数）。

表 5-5：使用换算系数把关键字转换成像素值

关键字	CSS1	CSS2	CSS3（草案）
xx-small	5px	9px	10px
x-small	7px	11px	12px
Small	11px	13px	14px
Medium	16px	16px	16px
Large	24px	19px	19px
x-large	36px	23px	24px
xx-large	54px	28px	32px

5.4.2 相对大小

关键字 larger 和 smaller 相对简单，它们根据父元素的字号增大或减小一定比例，这里使用的换算系数与计算绝对大小时一样。也就是说，如果浏览器计算绝对大小时使用的换算系数是 1.2，计算相对大小关键字时用的也是 1.2。

```
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>
```

与字重的相对值不同，字号的相对值没有上下限。因此，字号可以小于 xx-small 或大于 xx-large。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>
```

从图 5-14 中可以看出，h1 元素中强调文本的字号比 xx-large 稍微大些。具体增大多少由用户代理决定，1.2 是建议的换算系数，而不是强制要求。段落中的强调文本比绝对大小 large 大一级。

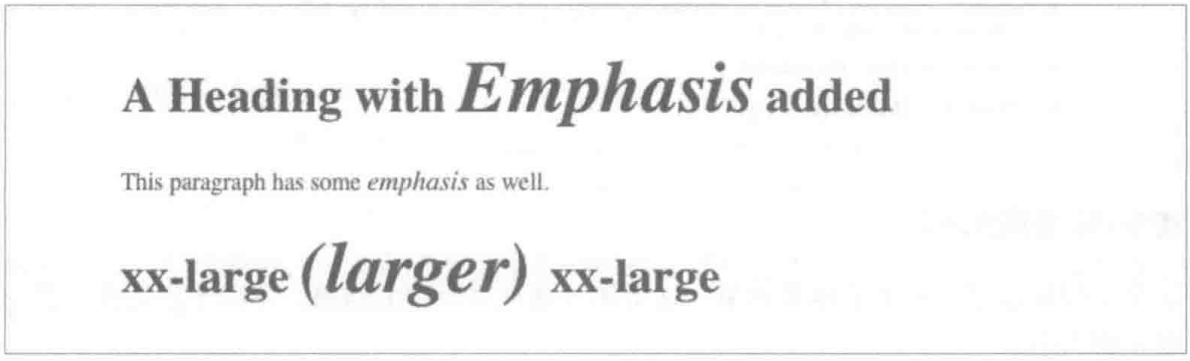


图 5-14：绝对字号中穿插着相对字号



用户代理在增大或减小字号时不一定会超过绝对大小关键字的上下限。

5.4.3 百分数和 em

百分数在某种意义上与相对大小关键字很像。百分数始终根据继承自父元素的字号计算。与前面讨论的关键字相比，百分数能更为细致地控制字号。来看下面这个例子，其结果如图 5-15 所示：

```
body {font-size: 15px;}
p {font-size: 12px;}
em {font-size: 120%;}
strong {font-size: 135%;}
small, .fnote {font-size: 70%;}

<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>
```

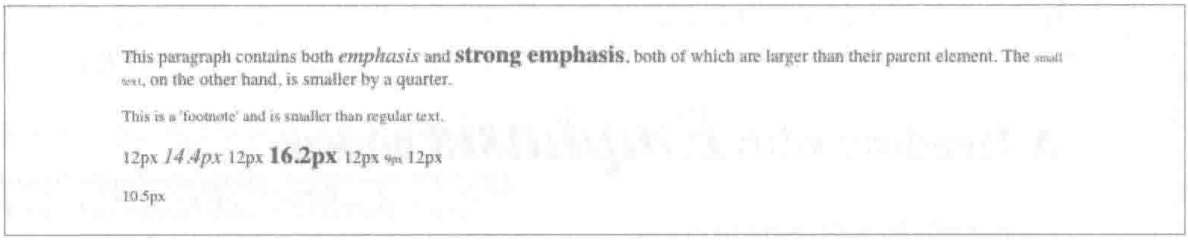


图 5-15：使用百分数

这个示例显示了字号的具体像素值。这是浏览器计算得到的结果，不同于真正在屏幕上显示的大小。

此外，CSS 还把长度单位 em 定义为等效于百分数，对字号而言，1em 与 100% 的效果相同。因此，下面两个规则得到的结果一样（前提是同属一个父元素）：

```
p.one {font-size: 166%;}
p.two {font-size: 1.6em;}
```

使用 em 单位时，百分数的规则同样适用，例如基于继承的字号计算等。

5.4.4 字号的继承

图 5-12 还表明，虽然 CSS 中的字号会继承，但继承的是计算得到的值，而不是百分数本身。因此，strong 元素继承的值是 12px，乘以声明的 135% 后得到 16.2px。“脚注”那一段的字号相对从 body 元素继承的值（15px）计算，乘以 75% 后得到 11.25px。

与相对大小关键字一样，百分数也会累积。因此，下述标记的显示结果如图 5-16 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>
```

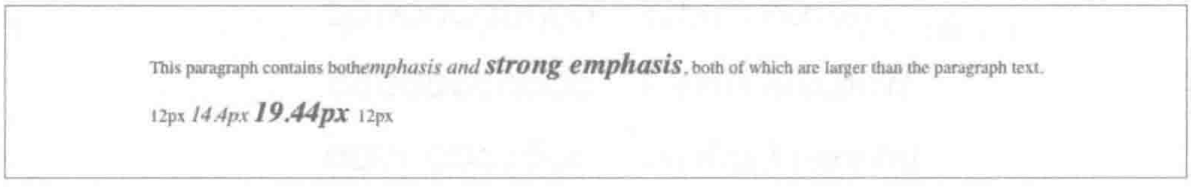


图 5-16：继承带来的问题

图 5-16 中那个 strong 元素的字号是这样计算的：

12 px × 120% = 14.4 px
14.4 px × 135% = 19.44 px（显示时可能会舍入为 19 px，参见下一节）

按比例缩放可能会向另一个极端发展。假设有个文档中只有一系列无序列表，其中一些嵌套在另一个列表中，而且有些列表嵌套四级深。试想把下述规则应用到这个文档上会导致怎样的结果：

```
ul {font-size: 80%;}
```

假设列表嵌套四级深，最内层的无序列表经计算得到的 font-size 值将是最外层无序列表的 40.96%。嵌套每深一层，字号就变成父列表的 80%，因此嵌套的列表一层比一层看不清。

显示时舍入

多数现代浏览器在内部维护着小数字号，但渲染引擎不一定会使用。下面以图 5-17 中的文本为例分析。

每一行中的字母 *O* 从左至右逐渐增大 0.1 像素字号。因此，最左边的 *O* 字号为 `10px`，中间的 *O* 字号为 `10.5px`，最右边的 *O* 字号为 `11px`。

从图 5-17 可以看出，不同的浏览器和操作系统组合得到的结果有所不同。例如，macOS 中的 Opera、Safari 和 Chrome 在 `10.5px` 的位置突然由 10 像素变成 11 像素。Windows（7 和 8）中的 Internet Explorer 和 Firefox 也是如此。而 macOS 中的 Firefox 看起来字号相同，像是一条平直的线。其实，由于字号有细微差异，各字符在绘制效果上是有细微差别的，不仔细看（或用尺子量）很难察觉。由此可见，从一端到另一端，字号上的增加很难看出来。



图 5-17：小数字号

尽管如此，如果使用审查工具，或者直接通过 DOM 脚本查询，你会发现，每个浏览器都会保留字号中的子像素（subpixel）。右数第 3 个 *O*，计算得到的字号是 `10.8px`，不过这并不是在屏幕上显示的大小。

关键字和等宽文本

字号关键字和继承会导致实际大小缩水，这在某些浏览器渲染等宽文本（例如使用 Courier 字体）时尤为明显。以下述规则和标记为例，其结果如图 5-18 所示。

```
p {font-size: medium;}   /* 默认值 */
span {font-family: monospace; font-size: 1em;}

<p>This is a 'p' with a <span>'span'</span> inside.</p>
```

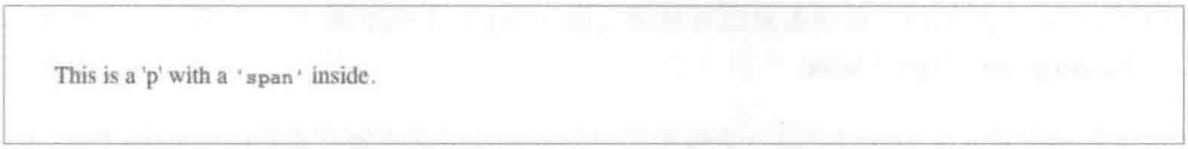


图 5-18：等宽文本在字号上的奇怪现象

假设用户没有更改浏览器的偏好设置（文本的默认字号在这里设定），medium 的默认值一般是 16px。查询 span 元素外部的文本，审查工具会告诉你，计算得到的文本字号为 16px（同样假设用户没有更改偏好设置）。

因此，你可能以为 span 元素中的等宽文本也是 16 像素。在某些浏览器中确实如此，然而在有些浏览器中是 13px。

原因在于，虽然段落中的文本经计算得到的字号是 16px，但是通过继承向下传递的是关键字 medium。因此，span 中的字号是根据 medium 计算的。为了算出具体的字号，用户代理会查看用户的偏好设置，而多数浏览器为等宽文本设定的默认字号是 13px。所以，即使明确为等宽文本设定 font-size: 1em，在使用 16 像素字号的段落中，等宽文本依然显示为 13 像素。

字号设为 1em（或 100%）之外的值时，这个问题依旧存在。在下述示例中，等宽文本经计算得到的字号是 26px，而不是 32px（还是假设浏览器的默认设置未被改动）：

```
p {font-size: medium;}   /* 默认值 */
span {font-family: monospace; font-size: 2em;}

<p>This is a 'p' with a <span>'span'</span> inside.</p>
```

注意，不是所有浏览器都是这样处理的，有些浏览器会覆盖 medium 默认的对应关系，根据父元素的字号计算实际值。因此，文本在不同浏览器中显示的大小不一致。

然而，有个方法能绕开这个问题，而且对已知的浏览器都有效（至少是截至 2017 年年末）。具体方法如下：

```
p {font-size: medium;}   /* 默认值 */
span {font-family: monospace, serif; font-size: 1em;}

<p>This is a 'p' with a <span>'span'</span> inside.</p>
```

看到 font-family 中多了个 serif 了吗？这样做，在所有浏览器中，font-size: 1em 的意思都是段落实际字号的 100%，而不再根据 medium 对应的值计算。这种方法适用于所有浏览器，效果如图 5-19 所示。



图 5-19：等宽文本的字号协调了

5.4.5 使用长度单位

font-size 可以设为任何长度值。下述几个 font-size 声明是等效的：

```
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 5-20 中显示的结果假定用户代理知道显示媒体每英寸有多少点。不同的用户代理做不同的假设，有些基于操作系统，有些基于偏好设置，还有些基于编写用户代理的程序员的假设。不管怎样，上述五个声明得到的字号是一样的。因此，尽管结果与现实中的尺寸不匹配（例如，p.three 的实际大小可能不是半英寸），上述声明得到的量值是相同的。

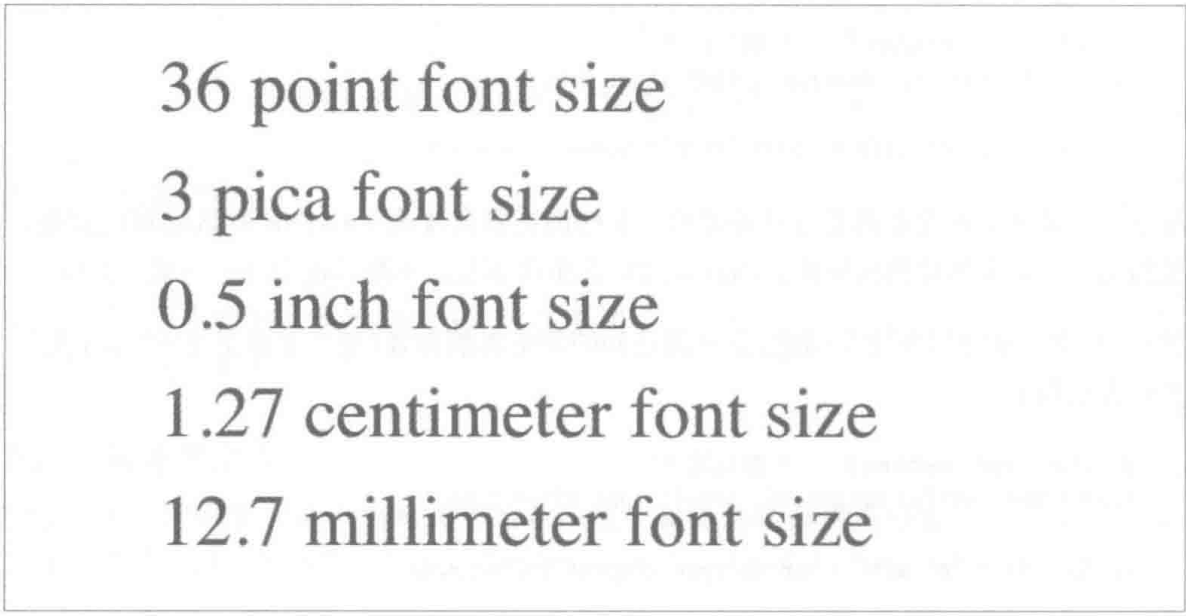


图 5-20：不同的字号

还有一个值可能与图 5-20 中显示的大小一样，即 36px。如果显示媒体是 72 像素每英寸（ppi），则跟实际距离一样。然而，很少有显示器使用这个分辨率了。现在的显示器具

有更高的分辨率，从 96 ppi~120 ppi 不等。而且移动设备更高，目前从 300 ppi~500 ppi 都有。

尽管操作系统和设备之间有如此多的差异，很多创作人员依然选择使用像素值设定字号。在既有文本又有光栅图像（GIF、JPG、PNG 等）的网页中，特别适合使用像素值，因为（理论上）可以通过 `font-size: 11px;` 等把文本的高度设为与图形元素一样，如图 5-21 所示。

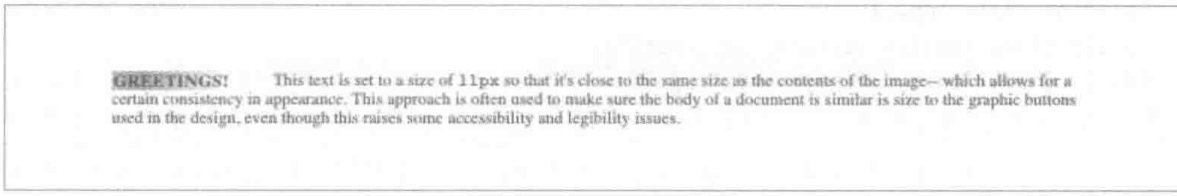


图 5-21：通过像素值字号让文本和图像保持高度一致

使用像素值设定 `font-size` 确实能保持字号一致（其实使用任何长度单位都是如此），但是有一个缺点：不是所有浏览器都能轻易缩放像素值设定的文本（有时甚至不能缩放），而且有时使用像素值设定的文本在模仿全屏幕设备的移动设备中的实际字号特别小（例如多款 iPhone）。单就这一点，一般不推荐使用像素值设定字号。

5.4.6 自动调整字号

有两个因素影响字体是否清晰易辨：字号和 x 高度。x 高度除以字号得到的结果称为高宽比值（aspect value）。随着字号的减小，高宽比值越高，字体越清晰易辨；相反，高宽比值较低的字体，很容易变得模糊难辨。CSS 提供的 `font-size-adjust` 属性用于改变字体族之间的高宽比值。

font-size-adjust

取值	*<number>* | none | auto
初始值	none
适用于	所有元素
继承性	是
动画性	是

这个属性的作用是在所用的字体不是创作人员的首选时维持清晰性。由于字体之间在外观上有差异，一个字体在某个字号时可能清晰易辨，而另一个字体在同样的字号下或许就变得模糊难辨，或者根本无法看清。

常用的 Verdana 和 Times 字体之间就有这种差异。请看图 5-22 和下述规则，两款字体的字号都是 10px：

```
p {font-size: 10px;}
p.cl1 {font-family: Verdana, sans-serif;}
p.cl2 {font-family: Times, serif; }
```

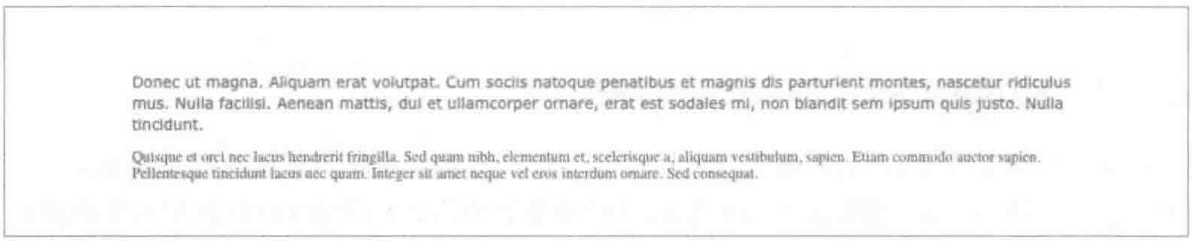


图 5-22：比较 Verdana 和 Times

使用 Times 的文本比使用 Verdana 的文本更难辨别。出现这种情况的部分原因是受限于基于像素的显示器，不过也是由于 Times 在较小的字号时模糊难辨。

Verdana 的 x 高度和字符大小的比值是 0.58，而 Times 是 0.46。为了解决这个问题，我们可以设定 Verdana 的高宽比值，让用户代理根据实际情况调整文本的字号。这里，要用到下述公式：

设定的 font-size ×

（font-size-adjust 值 ÷ 字体的高宽比值）

= 调整后的 font-size

因此，当实际使用的是 Times 而不是 Verdana 时，调整的结果是：

```
10px × (0.58 ÷ 0.46) = 12.6px
```

结果如图 5-23 所示。

```
p {font: 10px Verdana, sans-serif; font-size-adjust: 0.58;}
p.cl2 {font-family: Times, serif; }
```

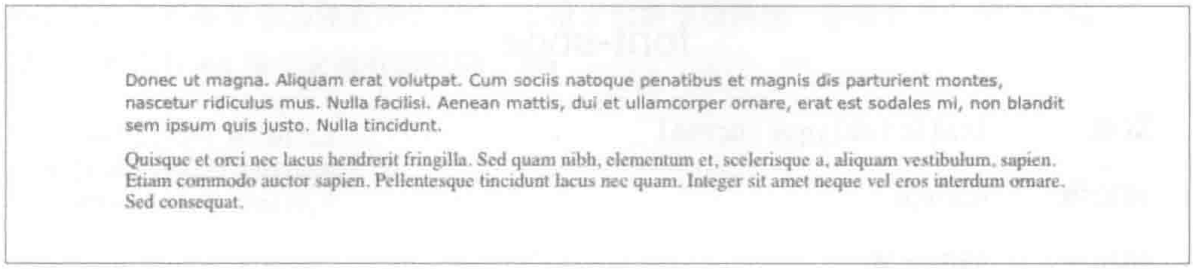


图 5-23：调整 Times

若想让用户代理自动调整字号，它首先要知道指定字体的高宽比值。支持 `@font-face` 的用户代理能直接从字体文件中获取这一信息，不过前提是字体文件中包含这个信息。专业制作的字体应该有这个信息，不过也不一定。 如果字体文件中没有高宽比值，用户代理可能会尝试自己计算。然而同样地，无法保证用户代理一定会这么做，或者一定能这么做。

假设用户代理能找到或算出高宽比值，把 `font-size-adjust` 设为 `auto` 就能达到所需的效果，即使不知道首选字体的高宽比值也行。例如，假设用户代理能确定 Verdana 的高宽比值是 0.58，那么下述规则得到的结果与图 5-23 一样：

```
p {font: 10px Verdana, sans-serif; font-size-adjust: auto;}
p.cl2 {font-family: Times, serif; }
```

`font-size-adjust: none;` 声明将禁止调整字号。这是默认的行为。



截至 2017 年年末，唯一支持 `font-size-adjust` 属性的用户代理是 Gecko（Firefox）系列。

5.5 字形

`font-style` 属性的作用十分简单：在 `normal`（常规）、`italic`（斜体）和 `oblique`（倾斜体）之间做出选择。就这样！这里唯一有点难懂的是斜体和倾斜体之间的区别，以及为什么浏览器不为你做出选择。

font-style	
取值	italic | oblique | normal
初始值	normal
适用于	所有元素
计算值	指定的值
备注	有对应的 @font-face 描述符
继承性	是
动画性	否

可以看出，font-style 的默认值是 normal。这表示竖直体，即没有倾斜。比如说，本书中的大多数文本都是竖直的。现在只剩下 italic 和 oblique 之间的区别了。为此，最简单的方法是参考图 5-24，图中很清楚地展示了二者之间的区别。

简单来说，斜体是一种单独的字型，各字母的构造有些改动，体现外观上的不同。衬线字体在这一点上的体现尤为明显，除了字符有点斜之外，字符本身还可能会做调整。而倾斜体只是竖直体的倾斜版本。以“Italic”“Cursive”和“Kursiv”等标识的字型通常对应于 italic 关键字，而 oblique 往往分配给以“Oblique”“Slanted”和“Incline”等标识的字型。

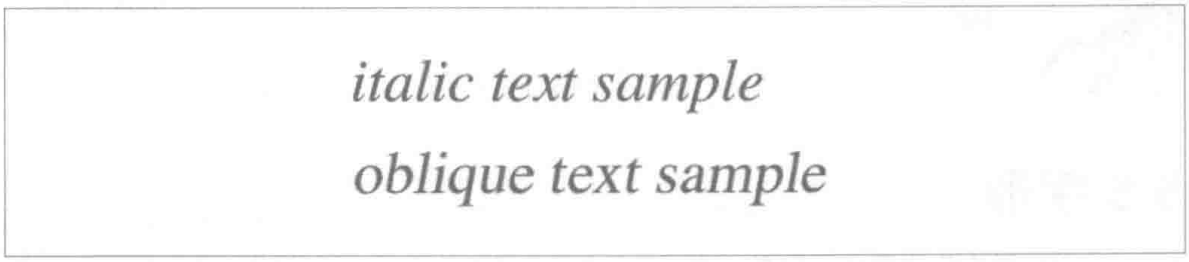


图 5-24：斜体和倾斜体图示

如果想让文档中的文本以人们熟知的方式使用斜体，可以编写下面这样的样式表：

```
p {font-style: normal;}
em, i {font-style: italic;}
```

这些样式会把段落显示为竖直体，而让 em 和 i 元素使用斜体，这都是符合常理的做法。不过，也可以让 em 和 i 之间有细微的差别：

```
p {font-style: normal;}
em {font-style: oblique;}
i {font-style: italic;}
```

仔细看图 5-25，你会发现 em 和 i 元素之间没有明显的不同。其实，不是每款字体都复杂到同时提供斜体和倾斜体，也很少有浏览器复杂到能区分二者之间的不同。

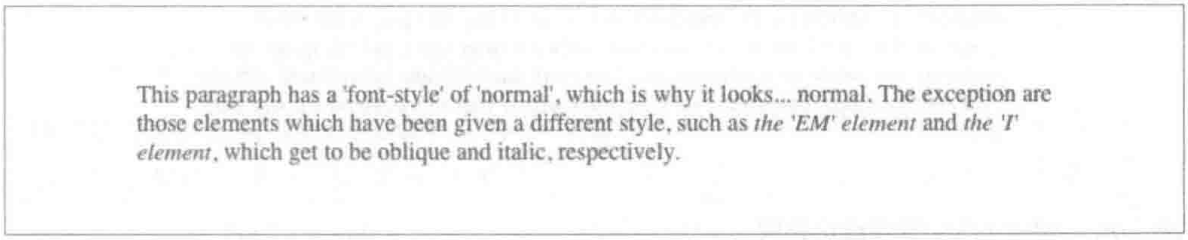


图 5-25：更多字形

万一遇到这样的情况，用户代理要做些额外的工作。如果没有斜体，而有倾斜体，可以使用后者代替前者。反过来，如果有斜体，而没有倾斜体，根据规范，用户代理可能不会使用前者代替后者。最后，用户代理可以直接倾斜竖直体，得到倾斜体。其实，在数字世界中这很常见，因为只需做些简单的计算就能倾斜一款字体。

此外，在某些操作系统中，即使声明使用斜体渲染文本，但在特定的字号时会转而使用倾斜体。例如图 5-26，其中显示的是运行 Classic OS（Mac OS 9）系统的 Macintosh 上显示的 Times 字体，两行文本在字号上只差一像素。

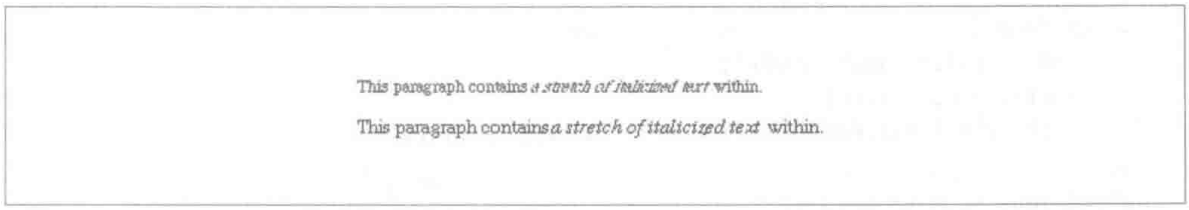


图 5-26：字体相同，字形相同，但是字号不同

遗憾的是，对此我们做不了什么，只能仰赖操作系统提供更好的字体处理方式。幸运的是，macOS 和 Windows XP 提供了十分优秀的字体渲染技术，而且创作人员可以通过 @font-face 中的 font-style 描述符指定斜体和倾斜体，让创作人员自行选择。

即便斜体和倾斜体通常使用同一个字型，但是 font-style 依然十分有用。例如，按照排版约定，引用块应该显示为斜体，而引用中的强调文本则应显示为竖直体。若想实现这样的效果（见图 5-27），可以使用下述样式：

```
blockquote {font-style: italic;}
blockquote em, blockquote i {font-style: normal;}
```

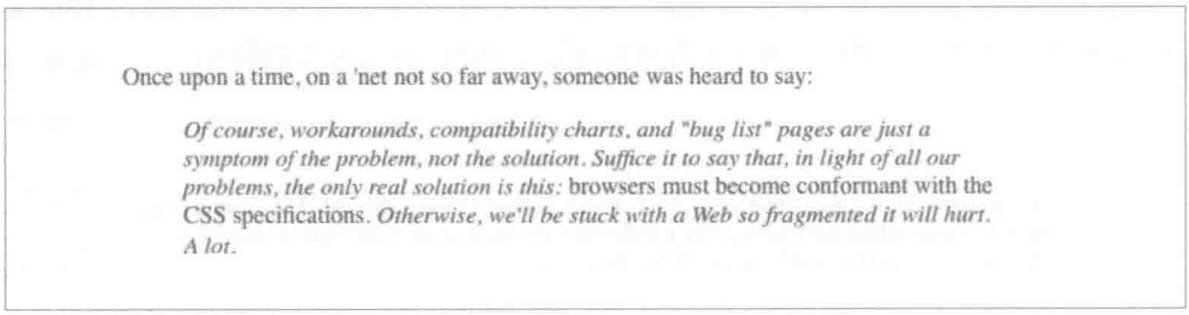


图 5-27：通过 CSS 实现排版约定

用户代理遇到不含粗体或斜体的字体族时会通过这个属性判断是否允许合成粗体或斜体。

font-style 描述符

用作描述符时，font-style 把指定的字型对应到指定的字形上。

```
@font-face {
    font-family: "SwitzeraADF";
    font-style: normal;
    src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-style: italic;
    src: url("SwitzeraADF-Italic.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-style: oblique;
    src: url("SwitzeraADF-Italic.otf") format("opentype");
}
```

像这样定义之后，下述规则将使用“SwitzeraADF-Italic”渲染 h2 和 h3 元素，而不使用“SwitzeraADF-Regular”，如图 5-28 所示。

```
h1, h2, h3 {font: 225% SwitzeraADF, Helvetica, sans-serif;}
h2 {font-size: 180%; font-style: italic;}
```

```
h3 {font-size: 150%; font-style: oblique;}
```

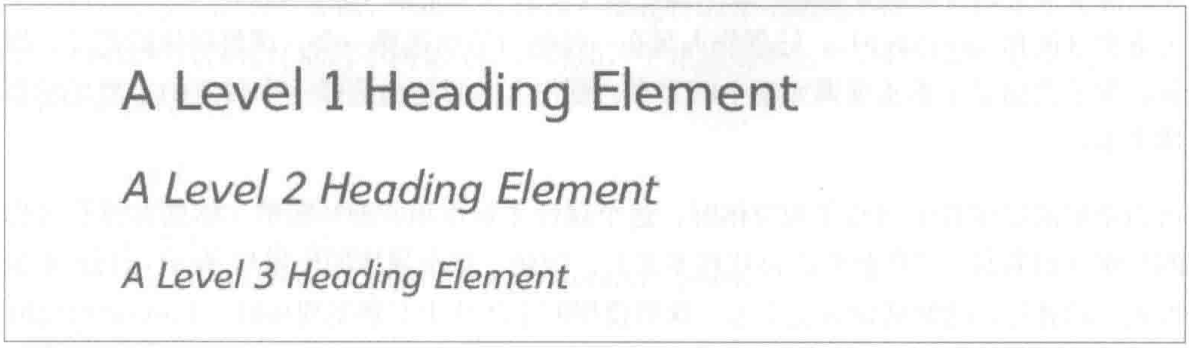


图 5-28：使用声明了 font-style 的字型

理想情况下，如果 SwitzeraADF 字体族有倾斜字型，创作人员可以把 oblique 值指向它，而不指向斜体。然而，SwitzeraADF 没有这样的字型，因此把 italic 和 oblique 都对应到斜体上。与 font-weight 一样，font-style 描述符的值可以是 font-style 属性允许取的值中除了 inherit 之外的其他任何值。

5.6 字体拉伸

有些字体族中的变体可能具有较宽或较窄的字母型式，这些变体通常以“Condensed”“Wide”“Ultra Expanded”等标识。这种变体存在的目的是在同一个字体族中提供瘦体和胖体。CSS 提供了一个属性，用于选择这样的变体，如此便无需在 font-family 声明中指定使用单独的字体族。这个属性是 font-stretch，不过它的名称有时让人不明其意。

font-stretch

取值	normal | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded
初始值	normal
适用于	所有元素
继承性	是
动画性	否
备注	有对应的 @font-face 描述符（见下文）

看到这个属性的名称，你可能会想它的作用是像折叠拉伸咸味太妃糖那样对待字体，而事实完全不是这样。抛开名称，这个属性的作用其实非常像 font-size 属性的绝对大小关键字（例如 xx-large），让创作人员在一些绝对值中选择一个，调整字体的宽度。例如，为了突出显示着重强调元素中的文本，创作人员可能会选择一个较宽的字型显示强调文本。

仅当使用的字体族中有宽体和窄体时，这个属性才起作用，而一般的字体族并没有这样的变体（如果有，字体族的价格往往不菲）。因此，这个属性的作用与 font-size 相差很大，后者可以随时随地改变字号。仅当使用的字体族中有加宽变体时，font-stretch: expanded 声明才起作用；如果没有加宽变体，什么也不会发生，即仍使用原来的字型。

下面以十分常见的 Verdana 字体为例。Verdana 的字型只有一个宽度，相当于 font-stretch: normal。像下面这样声明对显示的文本宽度没有任何影响：

```
body {font-family: Verdana;}
strong {font-stretch: extra-expanded;}
footer {font-stretch: extra-condensed;}
```

所有文本都将使用 Verdana 的正常宽度显示。然而，如果换成有多个宽度字型的字体族，例如 Futura 情况就不同了，如图 5-29 所示。

```
body {font-family: Verdana;}
strong {font-stretch: extra-expanded;}
footer {font-stretch: extra-condensed;}
```

If there one thing I can't **stress enough**, it's the value of Photoshop in producing books like this one.

Especially in footers.

图 5-29：拉伸后的字符



截至 2017 年年末，macOS 和 iOS 中的 Safari 都不支持 font-stretch，Opera Mini 也不支持。

font-stretch 描述符

与 font-weight 描述符类似，font-stretch 描述符的作用是把不同宽度的字型分配给 font-stretch 属性允许取的不同宽度值。例如，下述规则把三个字型分配给最常用的三个 font-stretch 值：

```
@font-face {
    font-family: "SwitzeraADF";
    font-stretch: normal;
    src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-stretch: condensed;
    src: url("SwitzeraADF-Cond.otf") format("opentype");
}
@font-face {
    font-family: "SwitzeraADF";
    font-stretch: expanded;
    src: url("SwitzeraADF-Ext.otf") format("opentype");
}
```

与前一节类似，创作人员可以通过 font-stretch 属性指定不同宽度的字型，如图 5-30 所示。

```
h1, h2, h3 {font: 225% SwitzeraADF, Helvetica, sans-serif;}
h2 {font-size: 180%; font-stretch: condensed;}
h3 {font-size: 150%; font-stretch: expanded;}
```

与之前一样，font-stretch 描述符的值可以是 font-stretch 属性允许取的值中除了 inherit 之外的其他任何值。

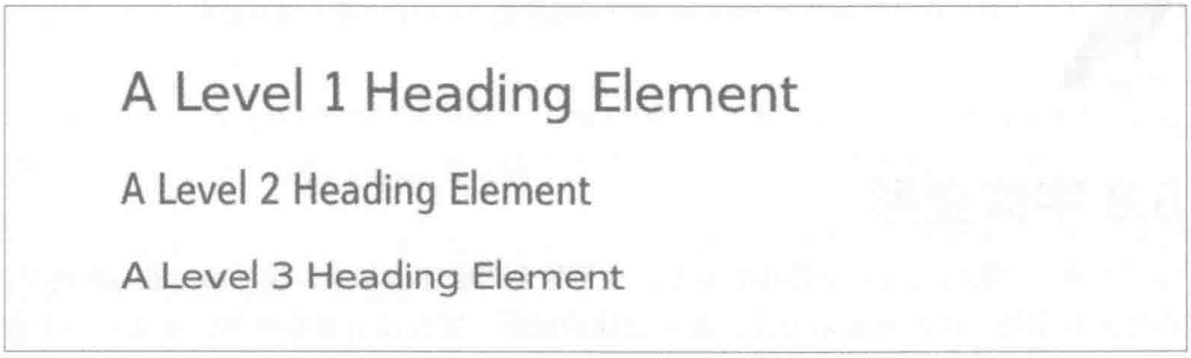


图 5-30：使用声明了 font-stretch 的字型

5.7 字距调整

有些字体定义了字符之间相对位置的数据，即字距。不同的字符组合，字距是不一样的。例如，*oc* 这两个字符的字距可能就与 *ox* 不同。类似地，*AB* 和 *AW* 的字距可能也不一样，在某些字体中，*W* 的右上端可能在 *A* 右下端的左边。字距可以使用 font-kerning 属性呼出或禁止。

font-kerning	
取值	auto | normal | none
初始值	auto
适用于	所有元素
继承性	是
动画性	否

none 的意思十分简单：让用户代理忽略字体中的字距信息。normal 的意思是让用户代理正常处理字距，即使用字体中的字距数据。auto 则把决定权交给用户代理，让用户代理选择最合适的处理方式，当然具体怎么做由所用的字体决定。例如，OpenType 规范建议（只是建议），只要字体中有字距数据就应该使用。如果字体中没有字距数据，font-kerning 没有任何作用。



注意，如果既调整字距，又应用 letter-spacing 属性，正确的顺序是先调整字距，然后再根据 letter-spacing 属性的值调整字符间距，而不是反过来。

5.8 字体变形

除了字重、字形等之外，字体还有变形。变形信息内嵌在字型中，包括旧时的各种连写、小号大写字母、小数的表示方式、数字之间的间距、零有没有贯穿线等。如果字型中有这些变形信息，可以通过 CSS 的 font-variant 属性调用。

font-variant

取值（CSS2.1）	normal | small-caps
取值（Level 3）	normal | none | [<*common-lig-values*> || <*discretionary-lig-values*> || <*historical-lig-values*> || <*contextual-alt-values*> || stylistic(<*feature-value-name*>) || historical-forms || styleset(<*feature-value-name*>#) || character-variant(<*feature-value-name*>#) || swash(<*feature-value-name*>) || ornaments(<*feature-value-name*>) || annotation(<*feature-value-name*>) || [small-caps | all-small-caps | petite-caps | all-petite-caps | unicase | titling-caps] || <*numeric-figure-values*> || <*numeric-spacing-values*> || <*numeric-fraction-values*> || ordinal || slashed-zero || <*east-asian-variant-values*> || <*east-asian-width-values*> || ruby]
初始值	normal
适用于	所有元素
计算值	指定的值
继承性	是
动画性	否
备注	有对应的 @font-face 描述符（见下文）

（Level 3 的）取值特别多，是吧？ CSS1 和 CSS2 只有两个取值：一个是 normal，表示普通形式；一个是 small-caps，指定使用小号大写字母。我们先说明这两个值。

小号大写字母既不是大写字母，也不是小写字母，而是大小稍有区别的大写字母。你可能见过与图 5-31 中类似的文本：

```
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>
```

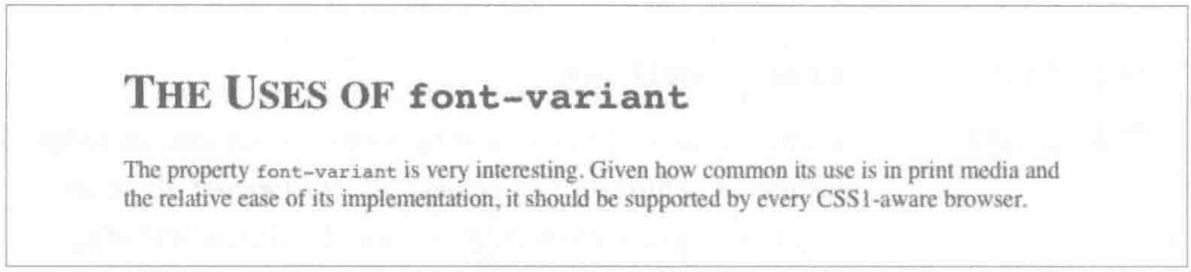


图 5-31：使用 small-caps 值

你可能发现了，h1 元素中的大写字母使用较大的大写字母显示，小写字母使用较小的大写字母显示。这与 text-transform: uppercase 的效果非常类似，不过就这里而言，两种方式之间的区别是大写字母的大小不同。之所以用字体属性声明 small-caps，是因为有些字体有专门的小号大写字母字型，从而必须通过字体属性选择。

如果没有这样的字型存在呢？规范提供了两个选项。其一，用户代理自行缩放大写字母，创建小号大写字母。其二，把所有字母都变成大写，而且大小相同，就像使用 text-transform: uppercase; 一样，这不是理想的方案，不过尚可接受。

5.8.1 Level 3 新增的值

下面讨论 Level 3 增加的值。不置可否，CSS3 的取值繁杂，不过也能讲得清楚。这些值其实是下述属性允许的取值总和：

- font-variant-ligatures
- font-variant-caps
- font-variant-numeric
- font-variant-alternates
- font-variant-east-asian

例如（选个较为简单的），*<common-lig-values>* 出自 font-variant-ligatures 属性，取 common-ligatures 或 no-common-ligatures；*<numeric-fraction-values>* 出自 font-variant-numeric 属性，取 diagonal-fractions 或 stacked-fractions。其他值参见各属性的规范。

字体变形丰富多样，但是能不能用取决于两个因素：浏览器支持和字体支持。前者情况简单，截至 2017 年年末，字体变形还没有得到浏览器的广泛支持。当然，你可以使用 CSS2.1 定义的值，不过 Level 3 新增的值中多数只被 Gecko 和 WebKit 支持。

第二个因素说简单也简单，说复杂又复杂：不是每个字体都支持每一种变形。例如，多数拉丁字体不支持东亚字体的变形；又如，不是每一个字体都支持数字变形和连字。为了找出字体的支持情况，要阅读文档；没有文档的话，只能大量做测试（多数收费字体有文档，多数免费字体则没有）。

重要的是要记住，即使字体的一个变形可在特定的浏览器中使用，但是这并不代表其他变形也能使用。而且，字体中有某个变形，也不表示在所有浏览器中都能调用。可以看出，情况很复杂，而且没有多少详细的指南可供参考。



这里没有介绍各个 font-variant-* 属性，原因是截至 2017 年年末，浏览器对它们还没有提供较好的支持。详情参见 *http://w3.org/TR/css3-fonts/*。

5.8.2 font-variant 描述符

font-variant 描述符通过一个空格分隔的列表指定字型的哪些变体可以使用，哪些不能使用。例如，可以像下面这样启用常见的连字、小号大写字母和带贯穿线的零：

```
font-variant: common-ligatures small-caps slashed-zero;
```

现在，不难看出，font-variant 描述符的值可以是 font-variant 属性允许的取值中除了 inherit 关键字之外的任何值。

注意，这个描述符与前面讲过的其他描述符有个明显的区别。以 font-stretch 描述符为例，你可以把特定的字型指定给特定的 font-stretch 属性值。而 font-variant 描述符在 @font-face 规则中指明允许使用的变形能轻易覆盖属性中指定的字体变形。在下述示例中，即便 SwitzeraADF 有 diagonal-fractions 或 small-caps 变形，段落也不会使用它们显示。

```
@font-face {
    font-family: "SwitzeraADF";
    font-weight: normal;
    src: url("SwitzeraADF-Regular.otf") format("opentype");
    font-variant: stacked-fractions titling-caps slashed-zero;
}
```

```
p {font: small-caps 1em SwitzeraADF, sans-serif;
    font-variant-numeric: diagonal-fractions;}
```

5.9 字体特性

与 font-variant 类似，font-feature-settings 属性从低层控制 OpenType 字体的哪些特性可以使用（因此，不能用在 *.woff* 文件上）。

font-feature-settings	
取值	normal | *<feature-tag-value>*#
初始值	normal
备注	有对应的 @font-face 描述符（见下文）

这个属性的值可以是 OpenType 规范定义的一个或多个特性（以逗号分隔）。例如，若想启用常用的连字、小号大写字母和带贯穿线的零，可以这样做：

```
font-feature-settings: "liga" on, "smcp" on, "zero" on;
```

<feature-tag-value> 值的具体格式为：

<feature-tag-value>

<string> [*<integer>* | on | off]?

多数特性的值是整数 0 和 1，相当于 off 和 on（反过来用也行）。不过，有些特性的值可以是其他数，大于 1 时既表示启用特性，也用于定义选择的索引。如果列出了某个特性，但是没有提供数字，假定为 1（启用）。因此，下述规则都是等效的：

```
font-feature-settings: "liga";     /* 假定为 1 */
font-feature-settings: "liga" 1;   /* 明确声明为 1 */
font-feature-settings: "liga" on;  /* on = 1 */
```

注意，所有 *<string>* 值都必须放在引号里。因此，下述规则中的第一个特性有效，而第二个将被忽略：

```
font-feature-settings: "liga", dlig;
/* 启用常用的连字，想启用自由连字，但是忘了引号 */
```

另一个限制是，OpenType 要求所有特性标签都是四个 ASCII 字符长。比这长或短，以及使用非 ASCII 字符的特性是无效的，将被忽略（通常无需担心，除非所用的字体没有遵守命名规则，自己编造了特性名称）。

默认情况下，OpenType 字体都启用了下述特性，除非创作人员通过 font-feature-settings 或 font-variant 明确禁用了：

calt
: 根据上下文替换。

ccmp
: 组合字符。

clig
: 根据上下文连字。

liga
: 标准连字。

locl
: 本地化形式。

mark
: 基本定位标记。

mkmk
: 标记定位标记。

rlig
: 必要的连字。

此外，特定的字体可能还会默认启用其他特性，例如启用竖排文本的 vert。



全部 OpenType 特性名称参见 microsoft.com/typography/otspec/featurelist.htm。

font-feature-settings 描述符

font-feature-settings 描述符指定 OpenType 字型的哪些特性可以使用，哪些不能使用，多个特性以逗号分隔。

慢着，这跟几段之前介绍的 font-variant 难道不是基本一样吗？其实，的确如此。font-variant 描述符几乎涵盖 font-feature-settings 的一切功能，外加一些其他作用。font-variant 更符合 CSS 的风格，使用的值不是晦涩的 OpenType 标识符和布尔值。鉴于此，CSS 规范明确建议创作人员使用 font-variant，只在想要的字体特性不在 font-variant 的取值范围内时才使用 font-feature-settings。

注意，这个描述符只是把特性提供出来（或者禁止使用），并不在显示文本时真正启用（参见讲解 font-feature-settings 属性的那一节）。

与 font-variant 描述符类似，font-feature-settings 描述符的作用是定义 @font-face 规则中所声明的字型允许使用哪些字体特性。例如，对下述规则来说，段落不会使用另一种小数形式或小号大写字母显示，即使 SwitzeraADF 提供了这些特性也不行：

```
@font-face {
    font-family: "SwitzeraADF";
    font-weight: normal;
    src: url("SwitzeraADF-Regular.otf") format("opentype");
    font-feature-settings: "afrc" off, "smcp" off;
}

p {font: 1em SwitzeraADF, sans-serif; font-feature-settings: "afrc", "smcp";}
```

同前，font-feature-settings 描述符的值可以是 font-feature-settings 属性允许取的值中除了 inherit 之外的其他任何值。

5.10 字体合成

有时，指定的字体族中可能缺少某种字型，例如粗体或斜体。此时，用户代理可能会尝试使用可用的字型合成所需的字型，不过这样得到的效果不是特别好。为了解决这个问题，CSS 提供了 font-synthesis 属性，让创作人员控制合成哪种字型，或者在渲染页面时禁止合成。

font-synthesis

取值	none | weight || style
初始值	weight style
适用于	所有元素
继承性	是
动画性	否

在很多用户代理中，如果字体族中没有粗体，用户代理会自行计算，生成粗体。例如，生成的方法可能是在各字形的两侧增加像素。这样做看起来便利了，不过实现的效果可能无法让人满意。鉴于此，很多字体族都自带了粗体，因为字体设计人员想确保粗体的显示效果。

类似地，如果字体族中缺少斜体，用户代理会直接倾斜常规字型，合成斜体。这比合成粗体的效果还差，尤其是对衬线字体而言。图 5-32 对比了 Georgia 的合成斜体（称为“倾斜体”）和 Georgia 自带的斜体。

在支持的用户代理中，`font-synthesis: none` 声明将阻止用户代理在对应的元素上做这种合成。例如，如果想禁止整个文档都这么做，可以使用 `html {font-synthesis: none;}`。这么做的缺点是，尝试使用不带相应字型的字体族显示加粗或倾斜的文本时，文本不会加粗或倾斜。不过也有好处，我们无须担心用户代理合成效果不理想的变体。



截至 2017 年年末，只有 Firefox 支持 `font-synthesis` 属性。

italic text sample

oblique text sample

图 5-32：合成的斜体和自带的斜体

5.11 font 属性

字体相关的这些属性都具有强大的功能，不过一个一个写也是相当烦琐：

```
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;}
```

有些问题可以通过群组选择符解决，但是如果能把各属性合并到一起，那岂不是更好？font 就是这样的属性，它把所有字体相关的属性（外加一些其他属性）合写在一起。

font	
取值	[[*<font-style>* ‖ [normal | small-caps] ‖ *<font-weight>*]? *<font-size>* [/ *<line-height>*]? *<font-family>*] | caption | icon | menu | message-box | small-caption | status-bar
初始值	参见各单独属性
适用于	所有元素
百分数	<font-size> 基于父元素计算，<line-height> 基于当前元素的 <font-size> 计算
计算值	参见各单独属性 (font-style 等)
继承性	是
动画性	参见各单独属性

一般来说，font 声明可以使用所列属性允许的任何一个值，或者是系统字体值（参见 5.11.3 节）。因此，本节开头的示例可以简写为（效果相同，如图 5-33 所示）：

```
h1 {font: italic 900 small-caps 30px Verdana, Helvetica, Arial, sans-serif;}
h2 {font: bold normal italic 24px Verdana, Helvetica, Arial, sans-serif;}
```

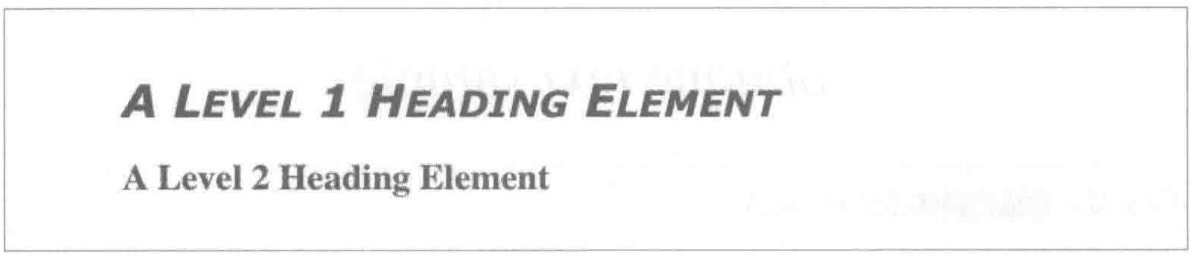


图 5-33：典型的 font 规则

我说的是“可以”简写成那样，因为除此之外还有其他写法，这得益于 font 属性相对宽松的写法。仔细看前例，你会发现前三个值的顺序不一样。在 h1 规则中，前三个值分别针对 font-style、font-weight 和 font-variant；而在第二个规则中，顺序变成了 font-weight、font-variant 和 font-style。这两种写法都没错，因为这三个属性可以随意排列。此外，如果其中某个属性的值是 normal，甚至不用写出。因此，下述规则与前例是等效的：

```
h1 {font: italic 900 small-caps 30px Verdana, Helvetica, Arial, sans-serif;}
h2 {font: bold italic 24px Verdana, Helvetica, Arial, sans-serif;}
```

这里，h2 规则的 normal 值省略了，但是效果仍与前例完全一样。

然而，要注意的是，只有 font 属性的前三个值有这种自由。后两个值的顺序要严格得多。注意，后两个值不仅必须是 font-size 和 font-family 这种顺序，而且必须出现在 font 声明中。哪怕少一个，整个规则都是无效的，很有可能会被用户代理彻底忽略。因此，下述规则得到的结果如图 5-34 所示：

```
h1 {font: normal normal italic 30px sans-serif;}   /* 没问题 */
h2 {font: 1.5em sans-serif;}   /* 也没问题，省略的值默认为 normal */
h3 {font: sans-serif;}     /* 无效：没有提供 font-size */
h4 {font: lighter 14px;}   /* 无效：没有提供 font-family */
```

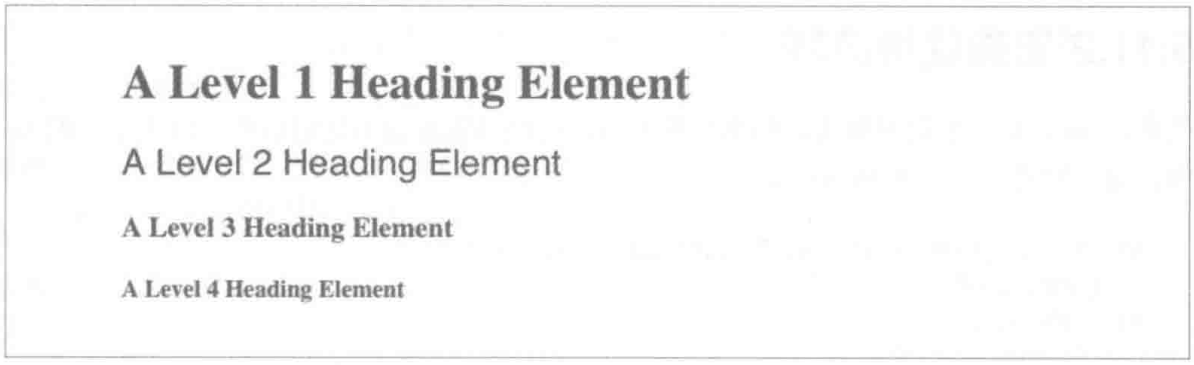


图 5-34：必须提供字号和字体族

5.11.1 加入行高

目前，我们只认为 font 属性能设五个值，但这并不正确。通过 font 属性还能设定 line-height 属性的值，尽管 line-height 是文本属性（本章不做介绍），而不是字体属性。设定的方法算是在 font-size 中增加一个值，以正斜线（/）分隔：

```
body {font-size: 12px;}
h2 {font: bold italic 200%/1.2 Verdana, Helvetica, Arial, sans-serif;}
```

这两个规则把所有 h2 元素设为粗斜体（使用某个无衬线字体族），把 font-size 设为 24px（body 的两倍），并把 line-height 设为 28.8px，结果如图 5-35 所示。



图 5-35：加入行高

与 font 属性的前三个值一样，增不增加 line-height 值完全是可选的。如果加入 line-height，记住，font-size 始终在 line-height 前面，不能反过来，而且两个值之间要以斜线分隔。

你可能觉得我啰嗦，但是 CSS 编写人员最常犯这样的错误，因此我不得不强调：font 属性必须的两个值是 font-size 和 font-family，而且必须以这样的顺序编写，其他值都是可选的。

5.11.2 正确使用简写

注意，font 是一个简写属性，如果使用不当，可能导致意想不到的后果。以下述规则为例，得到的结果如图 5-36 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>
```

注意到了吗，h2 元素既不是斜体，也没有使用小号大写字母，而且没有一个元素是加粗的？这是正确的行为。使用 font 属性时，省略的值都将重设为默认值。因此，前例可以改写成这样，但是结果不变：

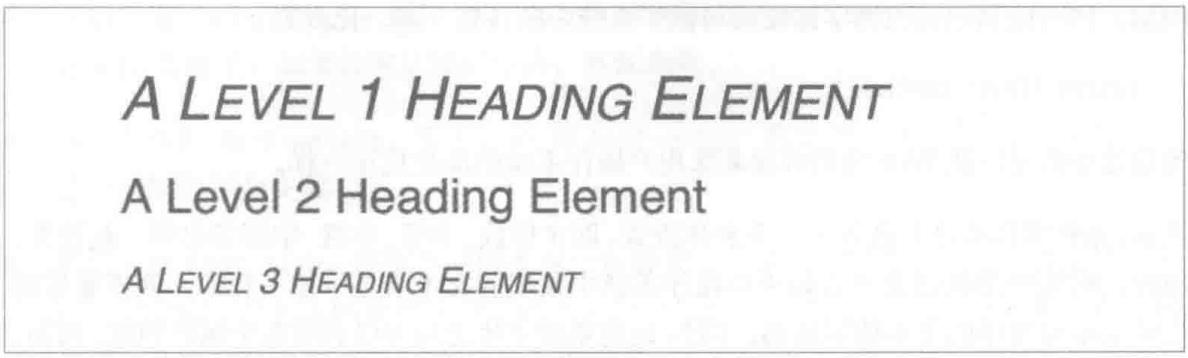


图 5-36：不同的简写方式得到的结果

```
h1, h2, h3 {font: italic normal small-caps 250% sans-serif;}
h2 {font: normal normal normal 200% sans-serif;}
h3 {font-size: 150%;}
```

这里，h2 元素的字形和变体都是 normal，而且三个元素的字重也都是 normal。这是简写时的预期行为。h3 的情况与 h2 不同，因为它用的是 font-size，而这不是简写属性，所以只对自己的值有影响。

5.11.3 使用系统字体

如果想让网页与用户的操作系统融为一体，可以在 font 声明中使用系统字体值。这样设定之后，元素上应用的是操作系统中控件的字号、字体族、字重、字形和变形。可用的系统字体值如下。

caption
: 用于说明文字的控件，如按钮。

icon
: 标注图标。

menu
: 在菜单中使用，即下拉菜单和菜单列表。

message-box
: 在对话框中使用。

small-caption
: 用于标注小型控件。

status-bar
: 用在窗口的状态栏中。

例如，你可能想把按钮的字体设成与操作系统中的按钮一样，比如说：

```
button {font: caption;}
```

使用这些值可以让 Web 应用看起来像用户操作系统的原生应用一样。

注意，系统字体或许只能作为一个整体设置，即字体族、字号、字重、字形等必须一起设置。因此，前例中的按钮文本看起来与操作系统中按钮的文本将是一模一样的，而不管按钮上的文本与周围的文本协不协调。不过，设定系统字体之后可以调整单个属性的值。例如，下述规则确保按钮的字号与父元素一样：

```
button {font: caption; font-size: 1em;}
```

如果调用的系统字体在用户的设备中不存在，用户代理可能会尝试模拟，例如减小 caption 字体的字号，得到 small-caption 字体。如果做不到这一点，用户代理应该使用默认字体。如果能找到指定的系统字体，但是无法读取全部值，用户代理应该使用默认值。例如，用户代理或许能找到 status-bar 字体，但是不知道是不是小号大写字母字型。此时，用户代理将把 font-variant 属性的值设为 normal。

5.12 字体匹配机制

可以看出，通过 CSS 能选择字体族、字重和变体。这背后蕴藏着字体匹配机制，这是一个非常复杂的过程，而且让人捉摸不透。创作人员一定要理解这个机制，从而让用户代理选择合适的字体显示文档。笔者把这个机制放在本章最后，是因为不是人人都需要了解字体相关的属性背后的运作方式，有些读者或许会选择跳过这一节。如果你感兴趣，下面就来说一说字体匹配的过程：

1. 用户代理创建或访问字体属性数据库。这个数据库中有用户代理能访问的全部字体的各个 CSS 属性。通常，设备中的所有字体都在这里，不过可能还有其他字体（例如，用户代理可能内置了字体）。如果用户代理遇到两个一样的字体，将忽略其中一个。

2. 用户代理把应用了字体属性的元素摘出来，构建显示元素所需的字体属性列表。首先，用户代理根据这个列表选择使用哪个字体族显示元素。如果能找到完全匹配的字体，用户代理就使用那个字体；否则，还要做些额外工作。

3. 匹配字体时先看 font-stretch 属性。

4. 然后再看 font-style 属性。任何以“italic”或“oblique”标识的字体都能匹配 italic 关键字。如果没有这样的字体，匹配失败。

5. 接下来匹配 font-weight，鉴于 CSS 对 font-weight 的处理方式（参见 5.3.1 节），这一匹配绝不会失败。

6. 然后处理 font-size。匹配字号时要有一定容差，不过这个容差由用户代理定义。因此，指定的字号和实际使用的字号在一个用户代理中可能允许存在 20% 的容差，而在另一个用户代理中可能只允许 10% 的容差。

7. 如果第 2 步没有找到匹配的字体，用户代理在同一个字体族中选择替代字体。找到后，回到第 2 步。

8. 假设找到一个基本匹配的字体，但是字体没有显示元素所需的全部信息，比如字体缺少版权符号，那么用户代理将回到第 3 步，搜索替代字体，然后再执行第 2 步。

9. 最后，如果找不到匹配的字体，而且所有替代字体都试过了，用户代理将选择指定字体族中的默认字体，力争正确显示元素。

此外，用户代理处理字体变形和特性的方式如下：

1. 查看默认启用的字体特性，包括指定文本所需的特性。默认启用的特性有 "calt" "ccmp" "clig" "liga" "locl" "mark" "mkmk" 和 "rlig"。

2. 如果是 @font-face 规则定义的字体，检查 @font-face 规则中 font-variant 描述符对应的特性。然后检查 @font-face 规则中 font-feature-settings 描述符对应的特性。

3. 检查由 font-variant 或 font-feature-settings 之外的属性确定的特性设置（例如，把 letter-spacing 属性设为默认值之外的值时将禁用连字）。

4. 检查 font-variant 属性及其子属性（例如 font-variant-ligatures），以及其他可能会调用 OpenType 特性的属性（例如 font-kerning）的值对应的特性。

5. 检查 font-feature-settings 属性的值对应的特性。

整个过程很长，也很麻烦，不过从中可以窥见用户代理是如何选择字体的。例如，你可能指定文档使用 Times 或其他衬线字体：

```
body {font-family: Times, serif;}
```

渲染各元素时，用户代理会检查 Times 提供的字符与元素中的字符是否匹配。多数情况都能找到匹配的字符，但是假设某一段中有个汉字，此时 Times 中就没有匹配的字符了，用户代理要么绕过汉字，要么寻找能满足显示需求的其他字体。西文字体极有可能不含汉字，但是假设有个字体有（就叫它 AsiaTimes 吧），那么用户代理就可以使用它显示那个元素，或者只用于显示那个汉字。因此，整个段落都可能使用 AsiaTimes，或者段落中除了那个汉字之外的文本使用 Times，而那个汉字使用 AsiaTimes 显示。

5.13 小结

起初，CSS 只提供了最为简单的字体属性，不过很快就变得丰富了，能细致调整 Web 中字体的方方面面。如今，创作人员可以通过网络下载自定义字体，也可以聚合多个字型自己构建字体族，可以说，创作人员对字体的控制达到了前所未有的程度。

今天的创作人员可以使用的排版选择比以往多很多，但是始终要铭记：必须合理利用手中的权力。一个网站中是可以使用 17 种不同的字体，但这绝非表示你应该这么做。这样做除了影响呈现给用户的美感之外，还会增加页面的体积，完全超出合理范围。与 Web 设计的其他方面一样，建议创作人员合理利用手中的权力，不要肆意妄为。

第 6 章

文本属性

是的，很多 Web 设计都要选择适当的颜色，让页面具有最酷的外观，不过在设计的过程中，可能更多的时间是花在思考文本的位置和外观上的。针对这一需求，早期的 HTML 出现了 <FONT> 和 <CENTER> 等标签，对文本的外观和位置提供一定程度的控制。

文本是网页的重要元素，因此 CSS 提供了很多相关的属性，以不同的方式控制文本。那么，文本和字体之间有什么区别呢？简单而言，文本是内容，而字体是用于显示内容的。借助文本属性，可以影响文本相对一行中其他内容的位置，可以把文本设为上标，可以为文本添加下划线，可以改变文本的大小写，甚至还能在一定程度上模拟打字机的 Tab 键。

6.1 缩进和行内对齐

首先讨论如何控制文本在一行内的位置。这些操作在撰写新闻稿或报告时经常用到。

不过，在此之前要先明确“行内”（inline）和“块级”（block）两个术语，因为本章将多次用到。其实，反过来讲更好理解。如果你主要使用某种西方语言，那么块级方向是指从上到下，而行内方向是从左至右。下面为这两个术语下更严格的定义。

块级方向指当前书写模式放置块级元素的方向。比如说，在英语中，块级方向从上到下，或者说是纵向的，例如一个段落（或其他文本元素）放在另一个段落的下面。

行内方向指块级元素中行内元素的书写方向。还以英语为例，行内方向从左至右，或者说是横向的。对阿拉伯语和希伯来语等语言来说，行内方向从右至左。

再看英语。正常情况下，一页英文内容在屏幕上显示时块级方向是纵向的，行内方向是横向的。但是，如果使用 CSS 变形属性把页面旋转 90 度，那么块级方向就变成了横向，而行内方向变成了纵向（从下到上）。

2017 年出现了相对较新的内容和布局方式，之前的旧布局语言极度依赖“横向”和“纵向”这两个概念，不过人们正在转变。本章的后续内容会尽量使用“块级方向”和“行内方向”这样的称呼，不过可能会夹杂着部分“纵向”和“横向”，敬请谅解。

6.1.1 缩进文本

使用西方语言撰写的书，多数在排版时会缩进一段的第一行，而且段落之间没有空行。以前，一些网站为了让人以为文本有缩进，会在一段的第一个字母前放一个小的透明图像，把文本向后推。CSS 为缩进文本提供了一种更好的解决方法：`text-indent` 属性。

text-indent

取值	*<length>* | *<percentage>*
初始值	0
适用于	块级元素
百分数	相对于所在块级元素的宽度
计算值	百分数如上；长度值得到绝对长度
继承性	是
动画性	是

`text-indent` 属性把元素的第一行文本缩进指定的长度，缩进的长度可以是负值。这个属性通常用于缩进段落的第一行：

```
p {text-indent: 3em;}
```

这个规则把段落的第一行缩进 3 em，如图 6-1 所示。

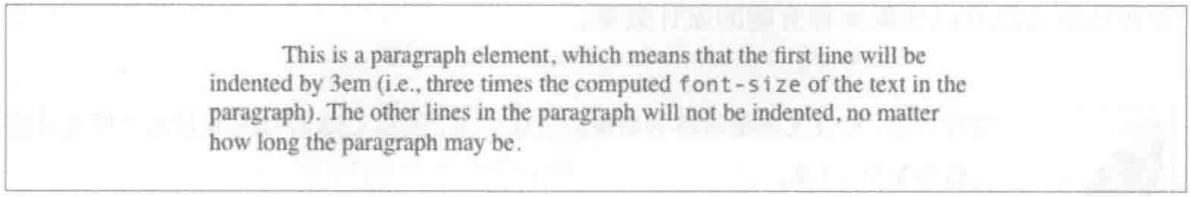


图 6-1：文本缩进

text-indent 属性可以用在任何块级元素上，缩进将沿着行内方向展开。text-indent 属性不能用于行内元素或置换元素（如图像）。然而，如果图像在块级元素的第一行里，它将随行中的其他文本一起后移。



如果想“缩进”行内元素的首行，可以通过内边距或外边距实现。

text-indent 属性的值可以是负值，这样可以实现一些有趣的效果。最常见的用法是实现悬挂缩进，即首行相比元素的其他内容悬挂在一边。

```
p {text-indent: -4em;}
```

使用负值时要小心，以防前几个词被浏览器窗口的边缘吃掉。为免出现这种显示问题，建议加上外边距或内边距：

```
p {text-indent: -4em; padding-left: 4em;}
```

不过，负缩进也有其优势。比如说，下面这个例子在文本外显示一个浮动的图像，如图 6-2 所示：

```
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>
```

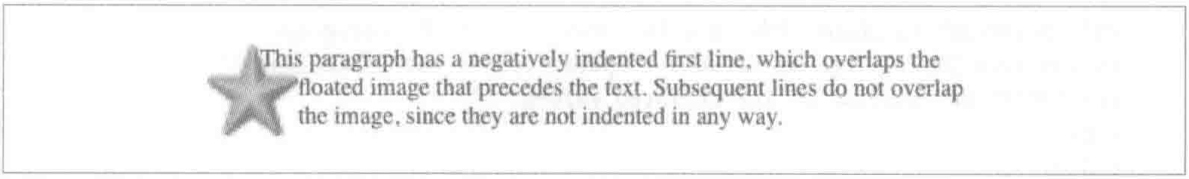


图 6-2：使用负值文本缩进添加浮动的图像

使用这种方法可以实现多种有趣的设计效果。



这种效果，即让文本沿着浮动图像的边缘显示，使用 CSS 浮动形状技术实现更可靠。详情参见第 10 章。

text-indent 属性的值可以使用任何长度单位，包括百分数。在下述示例中，百分数相对父元素的宽度计算缩进量。也就是说，如果把缩进值设为 10%，受其影响的元素的首行缩进的量为父元素宽度的 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>
```

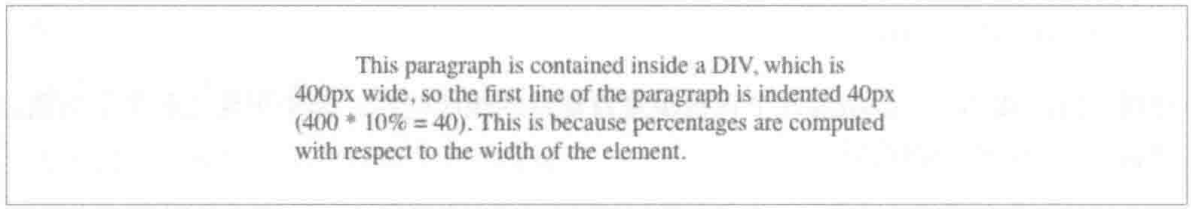


图 6-3：使用百分数设定文本缩进

注意，文本缩进只影响元素的第一行，即便有换行也是如此。text-indent 会继承，因此可能会出现预料之外的效果。例如，下述规则和标记得到的结果如图 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>
```

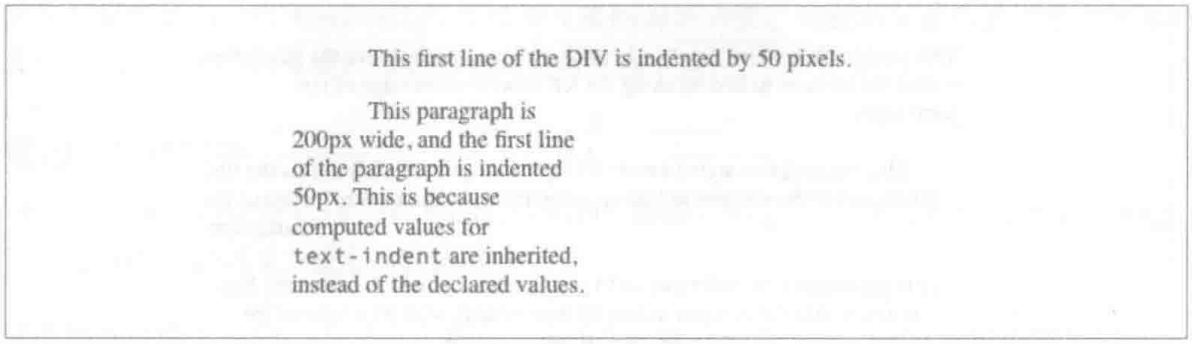


图 6-4：继承的文本缩进

6.1.2 文本对齐

text-align 比 text-indent 还基础，它控制元素中各文本行的对齐方式。

text-align

取值	start | end | left | right | center | justify | match-parent | start end
初始值	在 CSS3 中是 start；在 CSS 2.1 中，由用户代理指定，有可能根据书写方向而定（例如，英语等西方语言是 left）
适用于	块级元素
计算值	指定的值，match-parent 除外
继承性	是
动画性	否
备注	CSS2 支持使用 <*length*> 值，但由于缺少实现，CSS 2.1 将其删掉了

通过图 6-5 就能看出目前支持范围最广的三个值的效果。

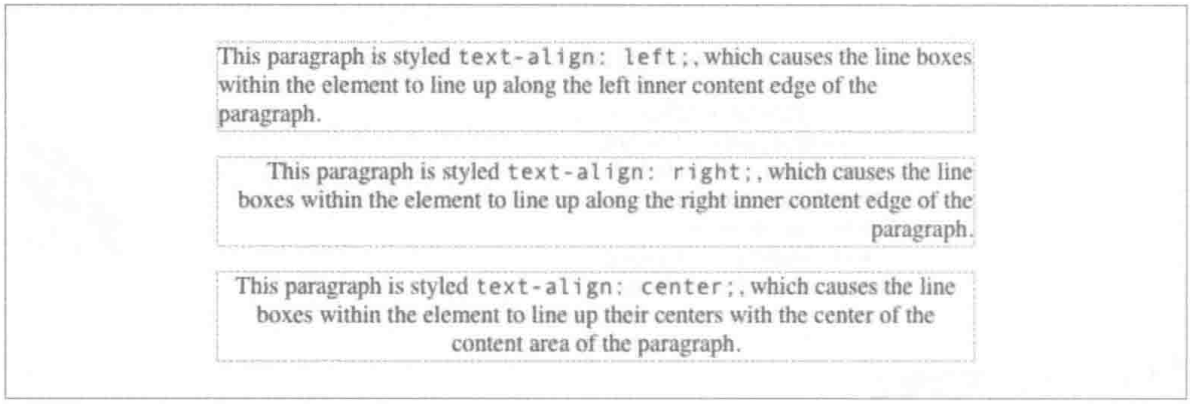


图 6-5：text-align 属性部分取值的行为

left、right 和 center 三个值的作用从字面就能看出。text-align 只能用于块级元素（如段落），因此，无法让锚记独立于行中的其他内容而居中显示（你或许也不会想这么做，以免文本重叠）。

在 CSS 2.1 中，对从左至右书写的语言，text-align 的默认值是 left，从右至左书写的语言默认值为 right（CSS 2.1 没有纵向书写模式的概念）。在 CSS3 中，对纵向书写的语言，left 和 right 分别对应起边和终边，如图 6-6 所示。

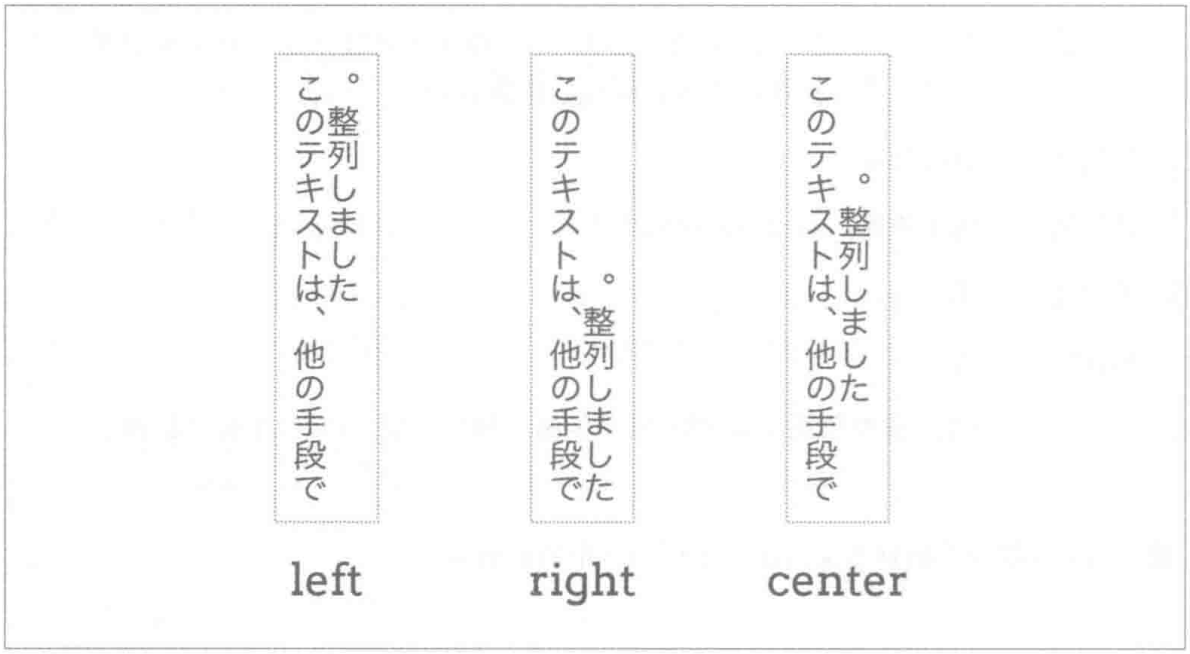


图 6-6：纵向书写模式中的左对齐、右对齐和居中对齐

毫无疑问，center 使元素中的各行文本居中对齐。你可能以为 text-align: center 的作用与 <CENTER> 元素一样，其实不然。<CENTER> 不仅影响文本，还能把整个元素（如

表格）居中显示，而 text-align 不控制元素的对齐方式，只影响元素中的内容。图 6-5 和图 6-6 以不同的书写模式为例清楚地说明了这一点。

起边和终边对齐

CSS3（更确切地说是 CSS Text Module Level 3 规范）为 text-align 属性增加了一些值，而且还修改了 CSS 2.1 规定的默认值。

新的默认值是 start，意思是文本与元素所在行框的起边对齐。在从左至右书写的语言中（如英语），起边是左边；在从右至左书写的语言中（如阿拉伯语），起边是右边；在竖写语言中，根据具体的书写方向，可能是顶边或底边。这样规定的最终效果是，默认值支持更多语言书写方向，而在目前已知的情况中，大多数时候默认的行为都是一致的。

类似地，end 把文本与行框的终边对齐。在从左至右书写的语言中，终边是右边；在从右至左书写的语言中，终边是左边；以此类推。这些值的效果如图 6-7 所示。

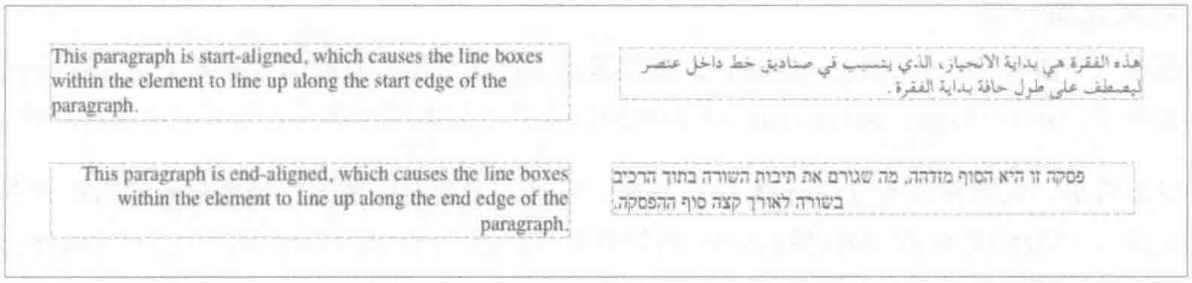


图 6-7：起边和终边对齐

两端对齐

justify 对齐方式经常被人忽视，不过它自身也有一些问题。两端对齐的文本，一行的两端都与父元素的边界对齐，单词和字母之间的空白会做调整，从而保证每一行的长度完全一致，如图 6-8 所示。印刷品（例如这本书）经常使用两端对齐，在 CSS 中使用时要多考虑一些方面。

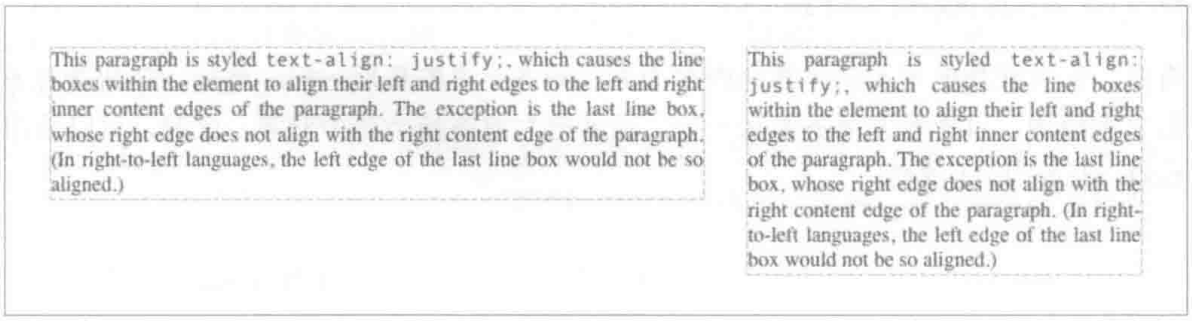


图 6-8：两端对齐

两端对齐的文本要经过拉伸，正好占满父元素左右边界之间的空间。至于如何拉伸，是由用户代理决定的，而跟 CSS 没有关系（至少截至 2017 年年末是这样）。例如，有些浏览器只在单词之间添加额外的空白，而有些则可能还会在字母之间添加额外的空白（不过，CSS 规范规定，如果为 letter-spacing 属性设定了长度值，“用户代理不应该增加或减少字符之间的间距”）。有些用户代理可能会减少部分行中的空白，把文本挤在一起。这些情况将影响元素的外观，甚至可能改变元素的高度，当然这取决于用户代理调整了多少行文本。



有个属性能更好地控制如何调整文本：text-justify。截至 2017 年年末，基本没有浏览器支持这个属性，Firefox 计划支持，Chrome 正在试验，不过有些问题。

与父元素一致

还有一个值未介绍：match-parent。浏览器不支持这个值，它的作用基本上被 inherit 涵盖了。text-align: match-parent 声明的作用是让元素的对齐方式与父元素保持一致。

目前看来，它的作用与 inherit 完全一样，不过二者是有区别的：如果父元素的对齐方式设为 start 或 end，match-parent 将把计算得到的 left 或 right 赋予元素；inherit 则不然，应用到元素上的就是 start 或 end。



严格来说，截至 2017 年年末，start end 也是 text-align 的一个取值，但是这里并未说明，因为它尚未实现，以后的规范可能会将其删除。

6.1.3 对齐最后一行

有时，你可能想使用不同于其他内容的方式对齐元素的最后一行。例如，在两端对齐的文本块中，可能想左对齐最后一行，或者把左对齐换成居中对齐。此时，可以使用 text-align-last 属性。

text-align-last

取值	auto | start | end | left | right | center | justify
初始值	auto
适用于	块级元素
计算值	指定的值
继承性	是
动画性	否

与 text-align 一样，从图 6-9 中就能看出这些值的作用。

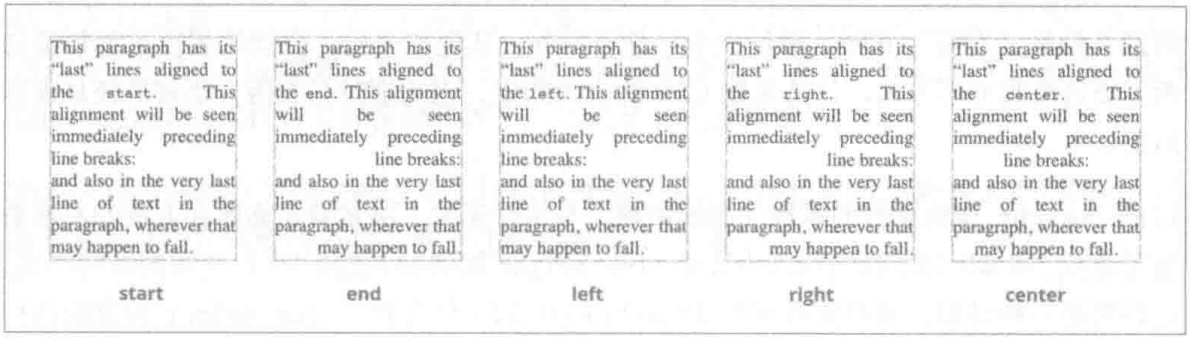


图 6-9：最后一行不同的对齐方式

如图 6-9 所示，元素的最后一行与其他内容的对齐方式不同，具体如何对齐由 text-align-last 属性决定。

仔细看图 6-9，你会发现，除了块级元素的最后一行之外，还有其他行也受这个属性影响。其实，只要一行后面有强制换行，不管是不是在元素的末尾，都受 text-align-last 属性的控制。因此，如果换行是由
 标签导致的，那么前面那一行的对齐方式将使用 text-align-last 的值。当然，块级元素的最后一行也受这个属性控制，因为此处的换行是由元素的结尾导致的。

text-align-last 有个有趣的现象：如果元素的第一行也是最后一行，text-align-last 的优先级比 text-align 高。因此，下述样式将把段落居中对齐，而不是靠起边对齐：

```
p {text-align: start; text-align-last: center;}
<p>A paragraph.</p>
```



截至 2017 年年末，Safari 和 Opera Mini 不支持 text-align-last，Internet Explorer 和 Edge 只支持 left、right 和 center 三个值。

6.2 块级对齐

讲完沿行内方向的对齐方式之后，下面转到纵向对齐上，即沿着块级方向的对齐方式，如上标和“纵向对齐”（纵向的前提是文本行横向排列）。行的构成是十分复杂的话题，可能要专写一本书，这里只做简要介绍。

6.2.1 行的高度

行之间的距离受行的“高度”影响。注意，这里所说的“高度”是相对文本行而言的，假设长轴是“宽度”方向，即便文本行是竖写的。从现在开始，讲到的属性名称都偏向西方语言及其书写方向，这是早期 CSS 的人为约定，因为那时只有西方语言能轻易表示出来。

line-height 属性指行的基线之间的距离，与字号无关，决定着元素所在方框的高度是增还是减。很多时候，我们都以为通过 line-height 能增加（或减少）行之间的纵向空间。这其实是一种误解，没有完全理解 line-height 的工作方式。line-height 控制的是行距（leading），是除字体高度之外在文本行上方的额外空间。也就是说，line-height 的值与字体高度之差就是行距。

line-height

取值	*<number>* | *<length>* | *<percentage>* | normal
初始值	normal
适用于	所有元素（不过请参考关于置换元素和块级元素的说明）
百分数	相对于元素的字号
计算值	长度和百分数，得到绝对值；否则，是指定的值
继承性	是
动画性	是

对块级元素而言，line-height 定义元素中文本行基线之间的最小距离。注意，定义的是最小距离，不是定死的值，基线之间的距离可能比 line-height 的值大。line-height 不影响置换元素的布局，但是依然应用到置换元素上。

行的构成

文本行中的每个元素构成一个内容区（content area），其高度由字体的高度决定。随内容区出现的是一个行内框（inline box），如果不考虑其他因素，其高度与内容区完全相等。line-height 导致的行距是影响行内框高度的因素之一。

元素的行距等于 font-size 的计算结果减去 line-height 的计算结果。这个值是行距的总值。注意，行距可能为负数。行距分为两半，分别放到内容区的上部和下部。算上行距，得到的就是元素的行内框。

举个例子。假设 font-size 是 14 像素（因此内容区也是这么高），line-height 的计算结果为 18 像素。二者之差（4 像素）分成两半，分别放到内容区的上部和下部。因此，行内框的高度为 18 像素，内容区上部和下部分别多出 2 像素。看起来用这种方法描述 line-height 的工作方式好像绕了弯子，不过是有充分理由的。

确定一行内容的全部行内框之后，行框也就确定了。行框恰好包围最高那个行内框的顶端和最低那个行内框的底端。整个过程如图 6-10 所示。

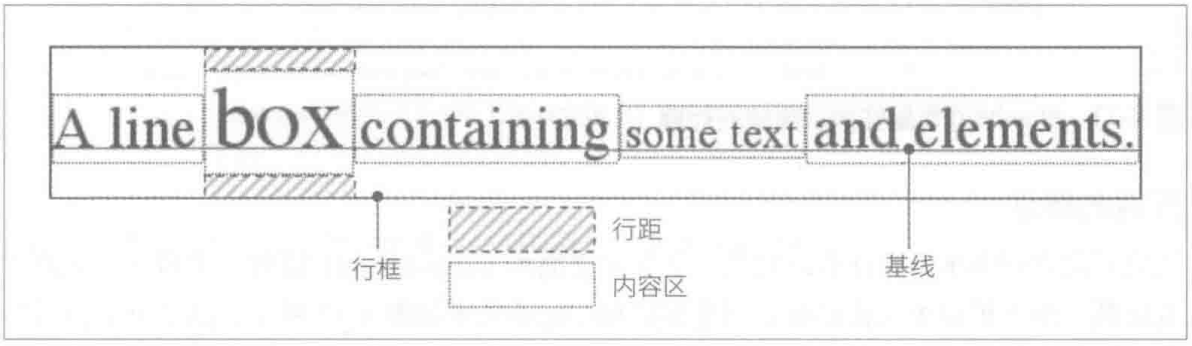


图 6-10：行框的构成示意图

line-height 的值

下面说明 line-height 属性的取值。使用默认值 normal 时，行之间的空间由用户代理计算。不同的用户代理计算出来的值可能不同，不过一般都是字体高度的 1.2 倍左右，因此行框的高度要比元素的 font-size 值大一些。

line-height 的值可以是长度量（如 18px 或 2em），不过多数情况下首选纯数字。注意，

即使使用有效的长度量，例如 4cm，浏览器（或操作系统）使用的具体量也可能与实际不同，因此元素在显示器上显示的高度可能并不是四厘米。

em、ex 和百分数相对元素的 font-size 值计算。下述 CSS 和 HTML 得到的结果如图 6-11 所示：

```
body {line-height: 18px; font-size: 16px;}
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 27px(18 * 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>
```

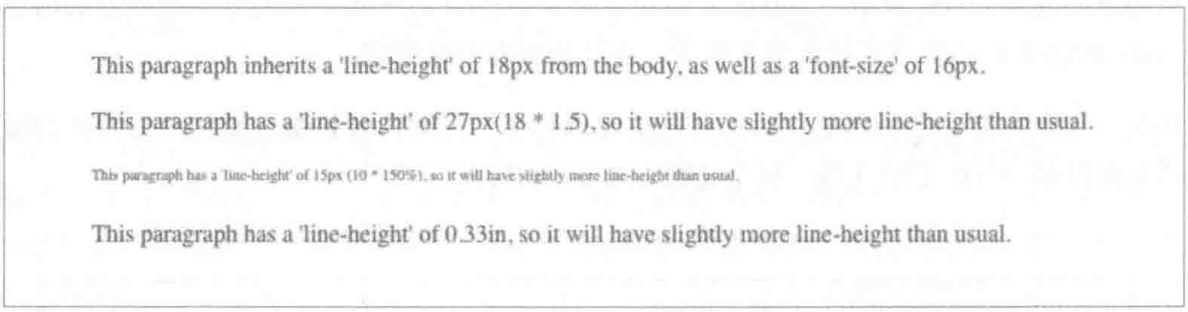


图 6-11：line-height 属性涉及简单的计算

行高的继承

块级元素之间继承的行高有点复杂。从父元素继承 line-height 值时，根据父元素的字号计算，而不根据子元素计算。下述样式和标记的结果如图 6-12 所示，结果可能与创作人员设想的不一样：

```
body {font-size: 10px;}
div {line-height: 1em;}  /* 计算结果为 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>
```

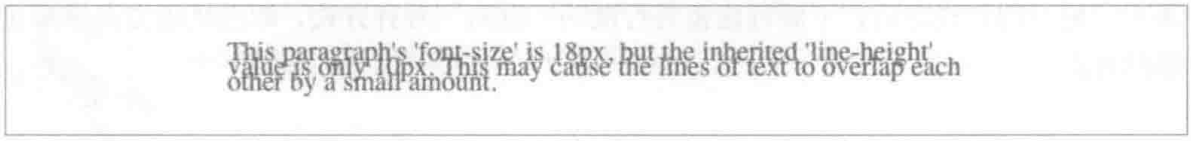


图 6-12：行高小、字号大，问题来了

这几行为什么如此贴近？因为计算得到的 line-height 值 10px 是段落继承自父元素 div 的。对图 6-12 中这种行高太小的问题，一种解决方法是为每个元素设定 line-height，但是这样做不太灵活。更好的方法是使用纯数字，即设定一个换算系数：

```
body {font-size: 10px;}
div {line-height: 1;}
p {font-size: 18px;}
```

使用纯数字时，继承的将是设定的换算系数，而不是计算得到的值。纯数字将应用到当前元素及其所有子元素上，因此各元素的行高将根据自身的字号计算（见图 6-13）。

```
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>
```

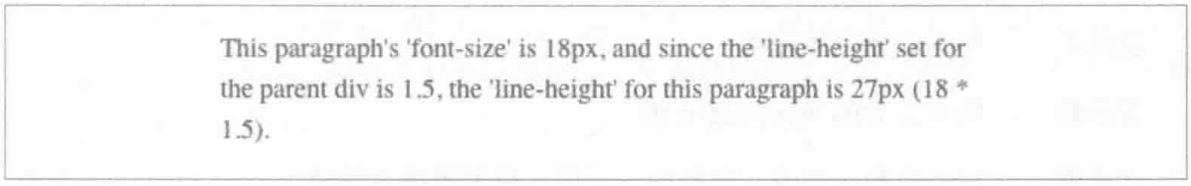


图 6-13：把 line-height 设为纯数字（换算系数），解决继承问题

设定 line-height 后，看起来额外的空间是平均增加到文本行的上部和下部的，但其实，额外的空间是增加到行内元素的内容区上部和下部的（或从内容区上部和下部减去），最终得到行内框。假设段落的默认字号为 12pt，对下述规则来说：

```
p {line-height: 16pt;}
```

因为 12 点高的文本“继承的”行高是 12 点，所以上述规则将在段落中的每一行上下增加高为 4 点的空间。这部分空间一分为二，分别放到每一行的上部和下部。现在，基线之间的距离是 16 点，这是分配额外空间的间接结果。

如果把 line-height 的值设为 inherit，元素的行高将变成父元素行高的计算值。其实，这与让行高值自然继承没有什么区别，只不过特指度和层叠解析会受影响。

基本了解行的构成之后，下面讨论相对行框的“纵向”对齐方式，即沿块级方向排列元素的方式。

6.2.2 纵向对齐文本

如果用过 sup 和 sub（上标和下标）元素，或者用过 <img src="foo.gif" align="middle"> 这样的标记，那么你就用过一些基本的纵向对齐方式。CSS 中的 vertical-align 属性只能用于行内元素和置换元素（例如图像和表单输入框）。vertical-align 属性不继承。



鉴于这个属性的名称是 vertical-align，本节将使用“纵向”和“横向”指代块级方向和行内方向。

vertical-align

取值	baseline | sub | super | top | text-top | middle | bottom | text-bottom | <length> | <percentage>
初始值	baseline
适用于	行内元素和单元格
百分数	相对元素的 line-height 值
计算值	百分数和长度值，得到绝对长度；否则是指定的值
继承性	否
动画性	*<length>*，*<percentage>*
备注	用在单元格上时，只能取 baseline、top、middle 和 bottom

vertical-align 的值可以是八个关键字中的一个、百分数或长度值。关键字中有熟悉的，也有不熟悉的：baseline（默认值）、sub、super、bottom、text-bottom、middle、top 和 text-top。下面说明各关键字应用到行内元素上的效果。



注意，vertical-align 不影响块级元素中文本的对齐方式。不过，vertical-align 可以控制单元格中元素的纵向对齐方式。

基线对齐

vertical-align: baseline 强制元素的基线与父元素的基线对齐。多数时候，浏览器都是这么做的，因为正常情况下一行中所有文本元素的底端应该对齐。

如果目标元素没有基线，例如图像、表单输入框或其他置换元素，元素的底端与父元素的基线对齐，如图 6-14 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 text in the paragraph.

图 6-14：沿基线对齐图像

这个规则很重要，一些 Web 浏览器始终把置换元素的底边放在基线上，即便行中没有其他文本也是如此。举个例子。假如一个单元格中只有一个图像，那个图像会放在基线上；然而在某些浏览器中，基线下方有一点空间，从而导致图像下方出现间隙；其他浏览器把图像紧贴着单元格的边界放置，因此没有间隙。有间隙是对的，不过多数创作人员不在乎。



如果想进一步了解这个间隙及其解决方法，请阅读“Images, Tables, and Mysterious Gaps”（http://mzl.la/19E2dJ7）。这篇文章虽然有些年头了，但是依然有参考价值。

上标和下标

vertical-align: sub 把元素放在下标处，即元素的基线（对置换元素来说是底边线）低于父元素的基线。规范没有定义下沉的距离，因此在不同用户代理中可能不同。

super 的作用与 sub 相反，让元素的基线（或置换元素的底边线）高于父元素的基线。同样，高出多少距离由用户代理决定。

注意，sub 和 super 不改变元素的字号，因此下标和上标中的文本不会变小（或变大）。下标和上标中的文本默认情况下与父元素的字号相等，如图 6-15 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>
```

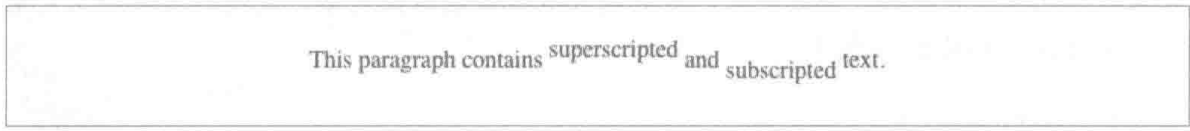


图 6-15：上标和下标



如果确实想让上标或下标中的文本比父元素中的文本小，可以使用 font-size 属性。

底端对齐

vertical-align: bottom 把元素所在行内框的底边与行框的底边对齐。例如，下述标记的结果如图 6-16 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>
```

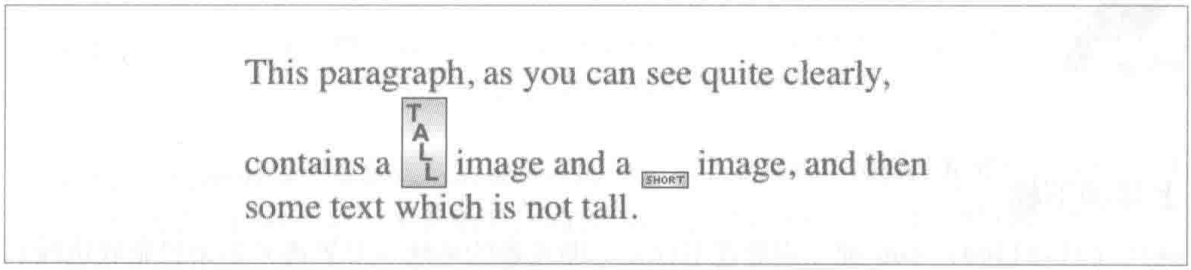


图 6-16：底端对齐

图 6-16 中那个段落的第二行里有两个行内元素，它们的底边是对齐的，而且都低于文本的基线。

vertical-align: text-bottom 相对行中文本的底边对齐。鉴于此，对齐时不考虑置换元素或其他非文本元素，而是基于“默认的”文本框。默认的文本框由父元素的字号确定。目标元素的行内框与默认文本框的底边对齐。因此，下述标记的结果如图 6-17 所示：

```
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>
```

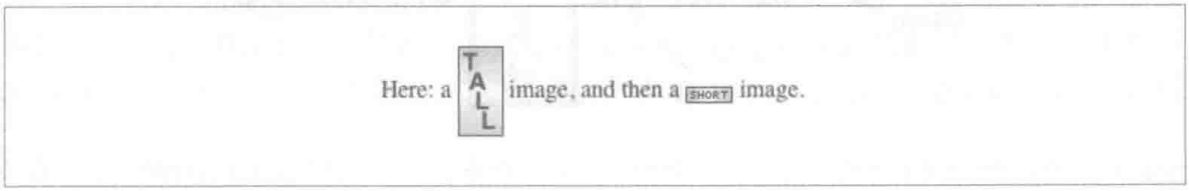


图 6-17：与文本的底端对齐

顶端对齐

vertical-align: top 的作用与 bottom 相反。同样，vertical-align: text-top 的作用与 text-bottom 相反。下述标记的渲染结果如图 6-18 所示：

```
.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>
```

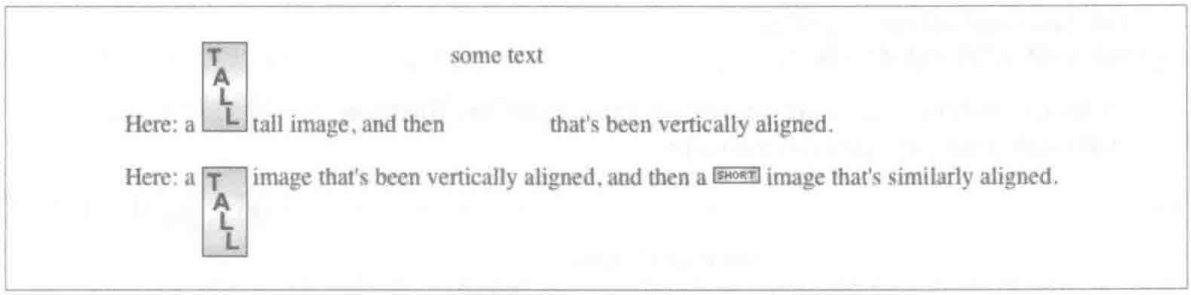


图 6-18：与一行的顶端和文本的顶端对齐

对齐后元素的具体位置取决于行中有什么元素、元素有多高，以及父元素的字号。

中线对齐

middle 这个值通常（不是始终）用在图像上。它的作用与字面意思不完全一样。middle 把行内元素所在方框的中线与父元素基线向上偏移 0.5ex 处的线对齐（1ex 等于父元素的 font-size）。详情参见图 6-19。

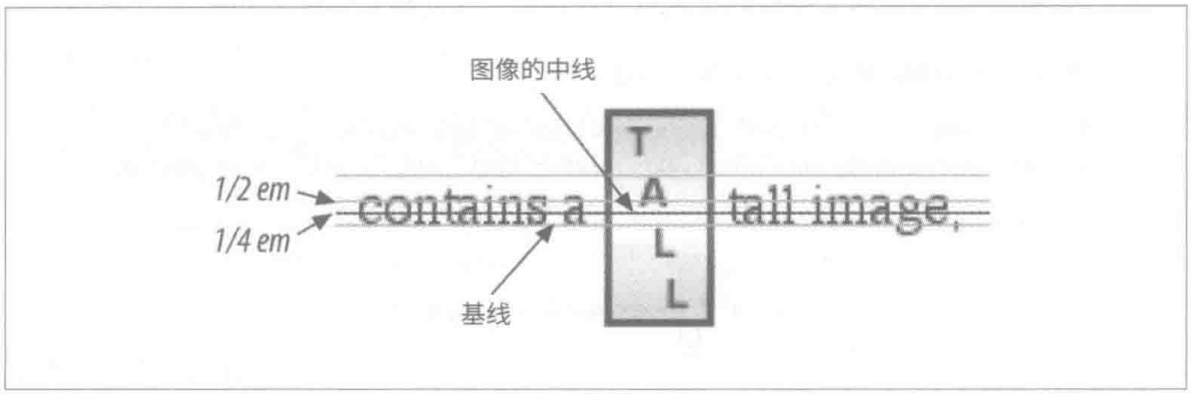


图 6-19：中线对齐的详细说明

多数用户代理把 1ex 视作半个 em，因此 middle 通常把元素的纵向中点与父元素基线向上偏移 1/4 em 处的点对齐。但这不是规范定义的距离，所以在不同的浏览器中向上偏移的量可能不同。

百分数

百分数不能像 align="middle" 那样对齐图像。把 vertical-align 的值设为百分数，效果是把元素的基线（或置换元素的底边）相对父元素的基线抬升或下沉指定的量（指定的百分数相对元素自身而非父元素的 line-height 计算）。正百分数抬升元素，负百分数下沉元素。如果抬升或下沉的量足够大，目标元素可能会出现在相邻的行上（见图 6-20），因此使用百分数时要小心。

```
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>
```

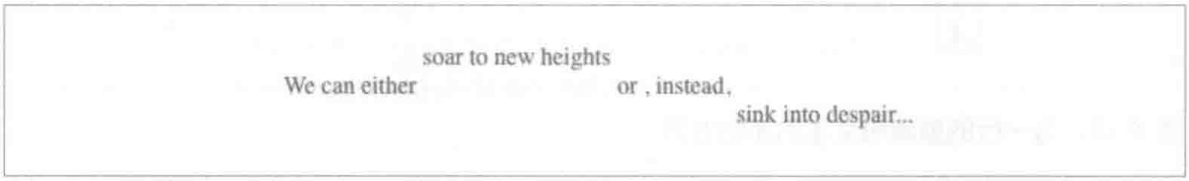


图 6-20：使用百分数得到的有趣效果

下面分析百分数的计算过程。以下述标记为例：

```
<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>
```

span 元素的 vertical-align 值为 50%，即抬升 9 像素，这是 span 元素继承的行高（18px）的一半。注意，抬升的值不是 font-size 的一半，即不是 7 像素。

长度值

最后，分析设为长度值时的纵向对齐方式。此时，vertical-align 的效果很简单：把元素抬升或下沉声明的距离。因此，vertical-align: 5px; 把元素从原位置向上抬升 5 像素。长度值为负时，下沉元素。这种简单的对齐方式在 CSS1 中没有，是在 CSS2 中增加的。

注意，纵向抬升或下沉的文本不会变成其他行的一部分，也不会与其他行中的文本重叠。如图 6-21 所示，有部分纵向抬升的文本出现在段落中间：

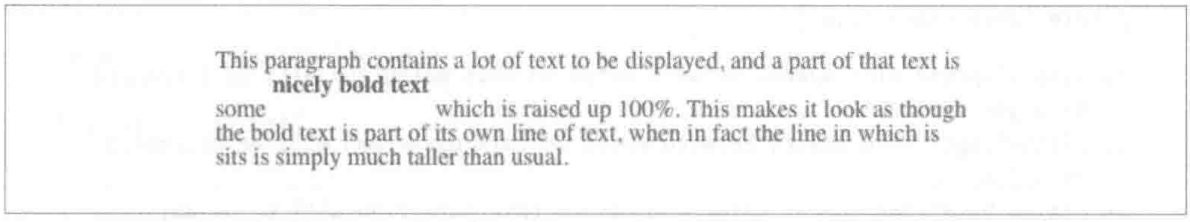


图 6-21：纵向对齐可能导致行变高

可以看出，纵向对齐的元素可能影响行的高度。还记得吗，行框恰好包围最高那个行内框的顶端和最低那个行内框的底端。这包括因纵向对齐而抬升或下沉的行内框。

6.3 单词间距和字符间距

讲完行内元素的纵向对齐方式之后，回到行内方向，说明单词间距和字符间距。与之前一样，这些属性也有一些异于直觉的行为。

6.3.1 单词间距

word-spacing 属性的值为长度，可正可负。指定的长度值追加到单词的标准间距上。其实，word-spacing 属性用于修改单词之间的距离。因此，默认值 normal 等同于把值设为零（0）。

word-spacing

取值	*<length>* | normal
初始值	normal
适用于	所有元素
计算值	normal 的结果是绝对长度 0；否则为指定的具体长度
继承性	是
动画性	是

设为正长度值时，单词之间的距离增加；设为负值时，单词之间靠地更近：

```
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>
```

上述规则的效果如图 6-22 所示。

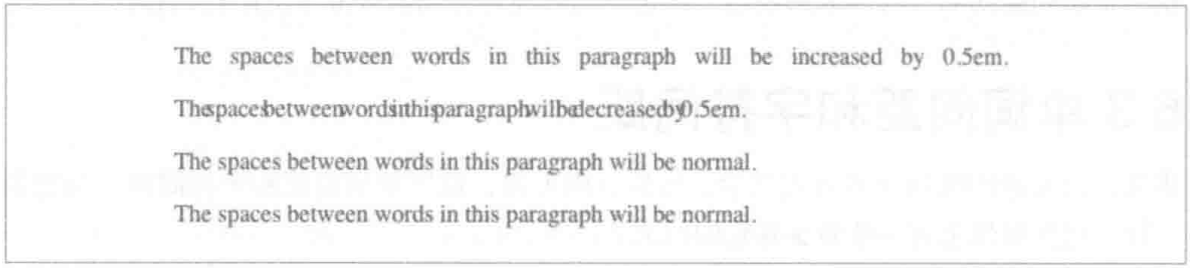


图 6-22：修改单词之间的距离

目前，我还没有给出“单词”的准确定义。用最简单的 CSS 术语来说，“单词”是指两侧有某种空白的非空白字符构成的字符串。这样定义没有具体的语义，只是假设文档中有诸多单词，单词两侧有一个或多个空白字符。支持 CSS 的用户代理无法得知在具体的语言中哪些是有效的单词，哪些不是。尽管这个定义没有多大意义，不过却表明 word-

spacing 不能用于象形文字或非罗马书写体。这个属性可能导致文档难以阅读，如图 6-23 所示。因此，使用时要小心。

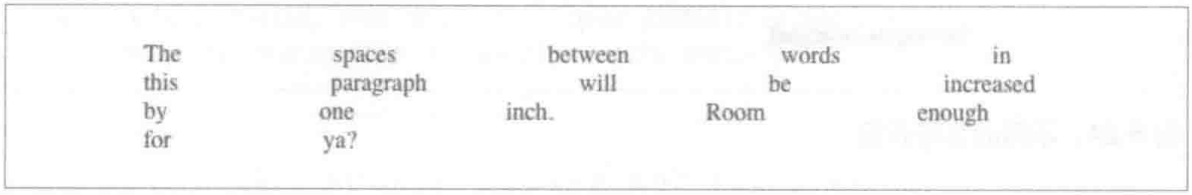


图 6-23：特别宽的单词间距

6.3.2 字符间距

word-spacing 属性的很多问题在 letter-spacing 属性上也有体现。这两个属性之间唯一的区别是，letter-spacing 属性修改的是字符或字母之间的距离。

letter-spacing

取值	*<length>* | normal
初始值	normal
适用于	所有元素
计算值	长度值得到绝对长度；否则是 normal
继承性	是
动画性	是

与 word-spacing 属性一样，letter-spacing 属性允许取任何长度值。默认值是关键字 normal（等同于 letter-spacing: 0）。指定的长度值将把字符之间的距离增加或减少指定的量。图 6-24 展示的是下述标记的结果：

```
p {letter-spacing: 0;}    /* 与 normal 的效果完全一样 */
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>
```

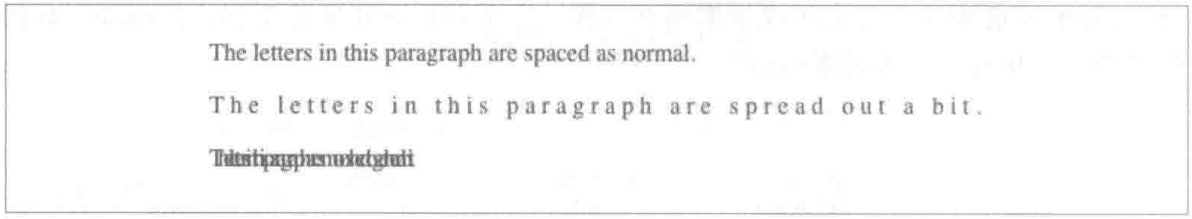


图 6-24：不同的字符间距

letter-spacing 可以突出强调效果，这个技术可谓历史悠久。你可能会编写下述声明，得到如图 6-25 所示的效果：

```
strong {letter-spacing: 0.2em;}

<p>This paragraph contains <strong>strongly emphasized text</strong>
which is spread out for extra emphasis.</p>
```

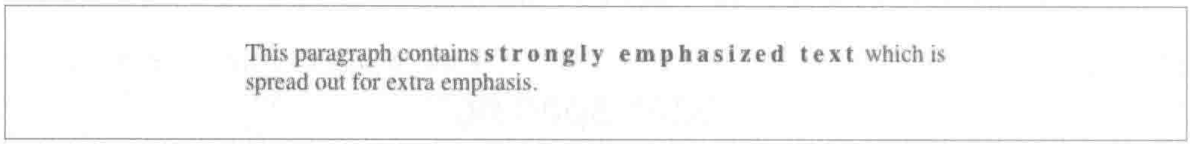


图 6-25：使用 letter-spacing 突出强调效果



如果使用的字体带连字特性，而且启用了，调整字符间距或单词间距可能导致这个特性失效。比如说，调整字符间距后，浏览器不会重新计算连字或其他连接。

6.3.3 对齐方式对间距的影响

word-spacing 的值可能会受 text-align 的值影响。如果元素是两端对齐的，为了让文本占满整行的宽度，可能要调整字符和单词之间的距离。鉴于此，创作人员声明的 word-spacing 值可能也要被调整。如果 letter-spacing 的值是长度，不受 text-align 的影响；但是，如果 letter-spacing 的值是 normal，为了使文本两端对齐，可能要修改字符之间的距离。CSS 未指明如何调整距离，因此用户代理可以采用自认为恰当的算法。

注意，继承的是计算得到的值，因此不管子元素的文本大了还是小了，字符间距都与父元素一样。word-spacing 和 letter-spacing 的值不能是换算系数，因此继承的值只能是经计算得到的值，而不能是换算系数（像 line-height 那样）。因此，你可能会遇到如图 6-26 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>
```

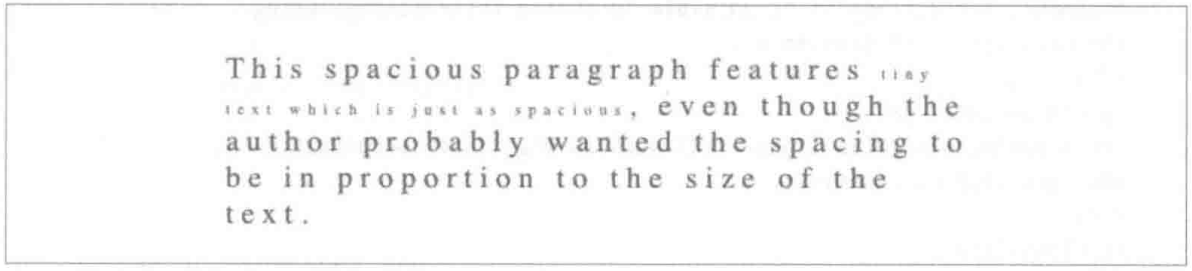


图 6-26：继承的字符间距

为了让字符间距与文本的大小成比例，只能分别单独设定，如下所示：

```
p {letter-spacing: 0.25em;}
small {font-size: 50%; letter-spacing: 0.25em;}
```

6.4 文本转换

讲完各个对齐属性之后，下面介绍如何使用 text-transform 属性转变文本的大小写。

text-transform

取值	uppercase | lowercase | capitalize | none
初始值	none
适用于	所有元素
计算值	指定的值
继承性	是
动画性	否

默认值 none 不对文本做任何修改，大小写形式与源文档一样。uppercase 和 lowercase 的作用与字面意思一样，分别把文本转换成大写和小写字母。最后一个值 capitalize 只把各单词的首字母变成大写。图 6-27 展示了这些值的效果。

```
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>
```

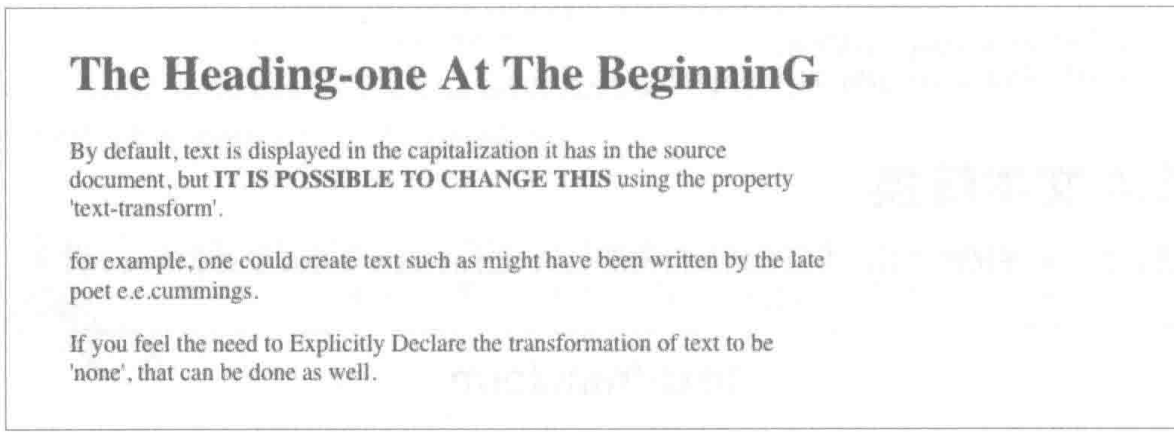


图 6-27：不同的文本转换方式

不同的用户代理可能以不同的方式确定单词的起点，因此哪些字母会变成大写取决于用户代理。因此，图 6-27 中 h1 元素里的文本“heading-one”可能会渲染成两种形式：“Heading-one”或“Heading-One”。CSS 没有规定哪种是正确的，因此两种形式都有可能出现。

此外，你可能注意到了，图 6-27 中那个 h1 元素的最后一个字母依然是大写的。这是正确的行为，把 text-transform 设为 capitalize 时，CSS 只要求用户代理确保各单词的首字母是大写的，而不管余下的字母。

text-transform 属性看似作用不大，但是如果突然决定把所有 h1 元素中的文本都变成大写，就会发现它特别有用。此时，无需一个个修改 h1 元素的内容，把工作交给 text-transform 属性就行了：

```
h1 {text-transform: uppercase;}
<h1>This is an H1 element</h1>
```

使用 text-transform 的好处有两个。其一，只需编写一个规则就能实现所需的改动，不用自己动手修改 h1 元素自身。其二，如果后来又想把大写改成最初的形式，也十分方便，如图 6-28 所示。

```
h1 {text-transform: capitalize;}
<h1>This is an H1 element</h1>
```

This Is An H1 Element

图 6-28：转换 h1 元素的大小写

注意，capitalize 的作用是替换每个“单词”开头的那个字母。标题常用的大小写约定，例如冠词（“a” “an” “the”）小写，不会得以保留。

6.5 文本装饰

接下来介绍 text-decoration，这个属性很吸引人，能实现很多有趣的效果。

text-decoration

取值	none | [underline || overline || line-through || blink]
初始值	none
适用于	所有元素
计算值	指定的值
继承性	否
动画性	否

你可能猜到了，underline 为元素添加下划线，就像旧时 HTML 中的 U 元素一样。overline 的作用相反，在文本的上方绘制一条线。line-through 绘制一条贯穿文本中

部的线，这也叫删除线，等价于 HTML 中的 S 和 strike 元素。blink 让文本一闪一闪，类似于 Netscape 中备受非议的 blink 标签。图 6-29 展示了这些值的效果。

```
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;}
```

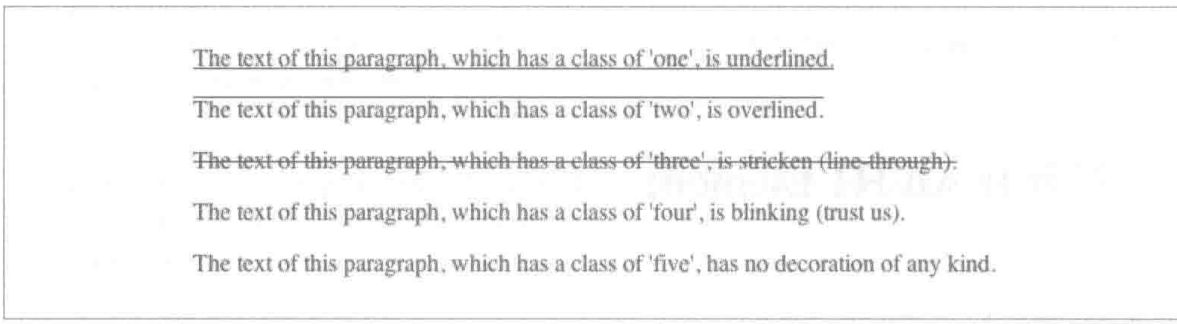


图 6-29：不同的文本装饰效果



印刷品中无法展示 blink 的效果，然而不难想象（太容易不过）。顺便说一下，用户代理不一定会闪烁以 blink 装饰的文本。写作本书时，所有已知用户代理要么计划放弃支持，要么已经不支持闪烁效果（Internet Explorer 从未支持过）。

none 值把应用到元素上的装修效果去掉。通常，无装饰的文本以默认的外观显示，不过也不总是如此。例如，链接通常默认带下划线。如果想去掉超链接的下划线，可以使用下述 CSS 规则：

```
a {text-decoration: none;}
```

如果使用这样的规则明确去掉了链接的下划线，锚记和常规文本之间在视觉上的唯一区别只有颜色（至少默认情况下是这样，然而也不尽然）。



注意，把链接的下划线去掉可能会让很多用户不高兴。去不去掉是你自己的选择，由你自己的偏好决定，不过要记住一点：如果链接的颜色与常规文本没有太大的差异，用户可能难以发现文档中的超链接，尤其是那些色盲用户。

一个规则中可以使用多个装饰效果。如果想让超链接既有下划线又有上划线，可以这样做：

```
a:link, a:visited {text-decoration: underline overline;}
```

不过要注意，如果在同一个元素上应用多个装饰效果，胜出的那个规则会完全取代另一个值。以下述规则为例：

```
h2.stricken {text-decoration: line-through;}
h2 {text-decoration: underline overline;}
```

此时，类为 stricken 的 h2 元素只有贯穿线，而没有下划线和上划线，因为多个 text-decoration 值是替换的，而不是累积的。

6.5.1 怪异的装饰

下面来看 text-decoration 不寻常的一面。第一个奇怪的行为是，text-decoration 不被继承。这表明，任何装饰线（下划线、上划线或贯穿线）的颜色都与父元素的文本颜色一样。即便后代元素的颜色变了，也是如此，如图 6-30 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>
```

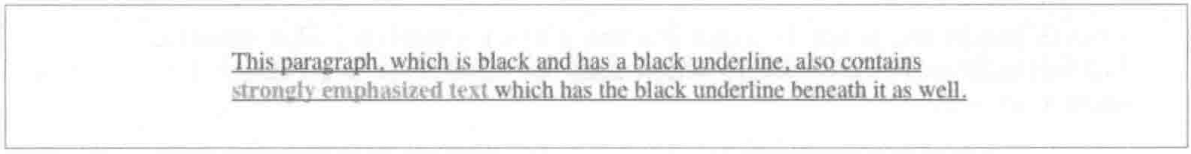


图 6-30：下划线的颜色保持不变

那么为什么会这样呢？因为 text-decoration 的值不被继承，strong 元素假定使用默认值 none。因此，strong 元素没有下划线。可是，strong 元素下面明显有一条线，说没有下划线不是信口开河吗？真的，确实没有。strong 元素下面那条线其实是段落的下划线，只是恰好“跨过”了 strong 元素。像下面这样调整加粗元素的样式看得更清楚：

```
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>
```

这样修改之后，得到的结果与图 6-30 是完全一样的，因为现在你是显式声明 strong 元

素没有装饰线。这表明，应用到父元素上的下划线（抑或上划线或贯穿线）在子元素上是无法取消的。

text-decoration 和 vertical-align 同时使用时还会发生更奇怪的事情。图 6-31 是其中一种怪异行为。尽管 sup 元素自身没有装饰线，但是它身在一个有上划线的元素中，导致那条线穿过 sup 元素：

```
p {text-decoration: overline; font-size: 12pt;}
sup {vertical-align: 50%; font-size: 12pt;}
```

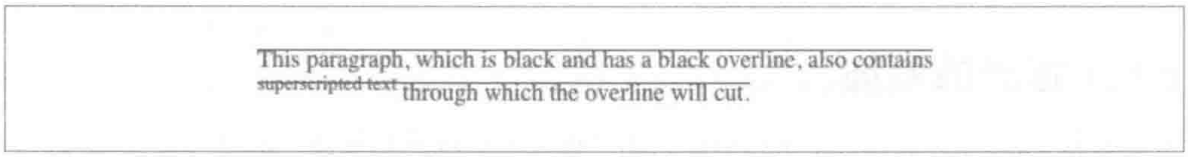


图 6-31：虽然奇怪，但这是正确的装饰行为

看到这些问题，你可能会发誓再也不用文本装饰线了。实际上，我只是指出了最为简单的一些情况，这里只是根据规范说明可能出现的结果。现实中，有些 Web 浏览器会去掉子元素的下划线，尽管这样做是不对的。这些浏览器违背规范的原因很简单，创作人员希望如此。来看下述标记：

```
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>
```

在去掉 strong 元素下划线的 Web 浏览器中，显示的结果如图 6-32 所示。

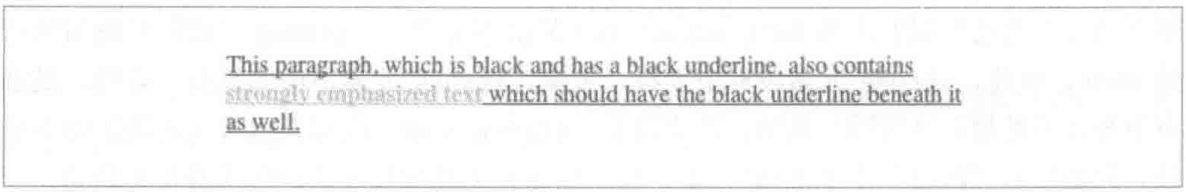


图 6-32：某些浏览器的真实效果

注意，很多浏览器还是严格遵守规范的，而上述这种浏览器（或其他用户代理）在未来也可能会严格遵守。如果依赖 none 去掉装饰线，一定要谨记，未来可能会在这上栽跟头，或者当下就可能导致问题。然而，CSS 的未来版本可能会提供去掉装饰线的方法，而不用像这样错误地使用 none，希望总是有的。

有种方法能改变装饰线的颜色，但不违反规范。前文说过，为元素设定文本装饰后，整个元素的装饰线都具有相同的颜色，即便是子元素，也不会使用其他颜色。若想让装饰线的颜色与元素的颜色保持一致，必须显式声明装饰线，如下所示：

```
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 6-33 中，strong 元素是灰色的，而且有下划线。灰色的下划线在视觉上“覆盖了”父元素的黑色下划线，因此 strong 元素的装饰线颜色与文本颜色一致。

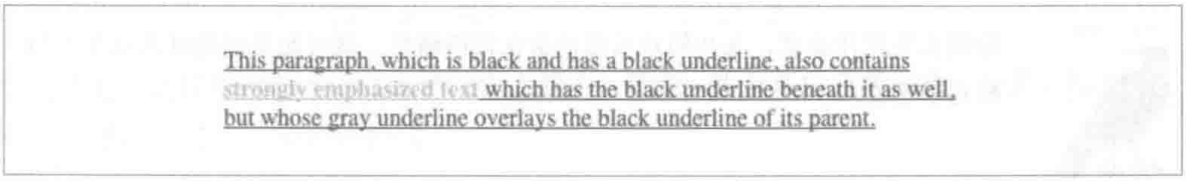


图 6-33：规避下划线的默认行为

6.6 文本渲染效果

text-rendering 是 CSS 最近新增的属性，它其实是一个 SVG 属性，支持它的用户代理不会把它视作 CSS 属性。这个属性的作用是让创作人员指定用户代理在显示文本时应该优先考虑什么方面。

text-rendering

取值	auto | optimizeSpeed | optimizeLegibility | geometricPrecision
初始值	auto
适用于	所有元素
继承性	是
动画性	是

optimizeSpeed 和 optimizeLegibility 的作用基本无需过多解释，它们指明首先考虑绘制速度，而不是清晰性特性，如紧排和连字（optimizeSpeed），或者反过来（optimizeLegibility）。

optimizeLegibility 针对的清晰性特性没有明确定义，而且文本渲染效果通常取决于运行用户代理的操作系统，因此具体结果视情况而定。图 6-34 中的第一行文本优化的是速度，第二行文本优化的是清晰性。

Ten Vipers Infiltrate AWACS
Ten Vipers Infiltrate AWACS

图 6-34：不同的优化方式

客观地说，图 6-34 中两种优化方式之间的区别很小，不过对可读性可能会产生重大影响。



即使指定优化速度，某些用户代理仍会优化清晰性。这可能是因为过去几年人们一味追求速度。

geometricPrecision 的作用不太明显，它其实是让用户代理尽量精准地绘制文本，确保能无损缩放。别以为情况始终是这样，现实并非如此。比如说，有些字体会随着字号的变化而改变紧排或连字效果，在字号小时提供较大的字距，字号变大之后又紧缩字距。使用 geometricPrecision 可以让用户代理忽略随字号增减而出现的这些变化。如果不理解的话，可以这样想：geometricPrecision 的作用相当于让用户代理把文本绘制成一系列 SVG 路径，而不是字形。

即便按照 Web 标准通常的规范程度，也没有说清 auto 的作用。SVG 规范是这样说的：

> 用户代理应该在速度、清晰性和几何精度上做适当的权衡，不过较之速度和几何精度，应该倾向于清晰性。

意思就是，用户代理自行决定合适的方法，只要向清晰性倾斜就行。

6.7 文本阴影

有时，你非常希望为文本加个阴影。text-shadow 属性应运而生。这个属性的句法起初看似古怪，不过稍加练习就能充分领会。

text-shadow

取值	none | [<*length*> || <*length*> <*length*> <*length*>?]#
初始值	none
适用于	所有元素
继承性	否
动画性	是

默认情况下，文本没有阴影。如果想，可以为文本定义一个或多个阴影。每个阴影由一个可选的颜色和三个长度值定义，最后一个长度值也是可选的。

颜色设定的是阴影的颜色，因此可以定义绿色、紫色，甚至是白色的阴影。如果省略颜色，阴影的颜色将与文本的颜色相同。

前两个长度值是阴影离文本的距离。第一个长度值设定横向偏移，第二个长度值设定纵向偏移。若想定义图 6-35 中那样不模糊的纯绿色阴影，向文本右侧偏移 5 像素，向文本下部偏移半 em，可以这样写：

```
text-shadow: green 5px 0.5em;
```

负的长度值把阴影向文本的左侧和上部偏移。下述规则为文本添加一个淡蓝色阴影，向左偏移 5 像素，向上偏移半 em，如图 6-35 所示：

```
text-shadow: rgb(128,128,255) -5px -0.5em;
```

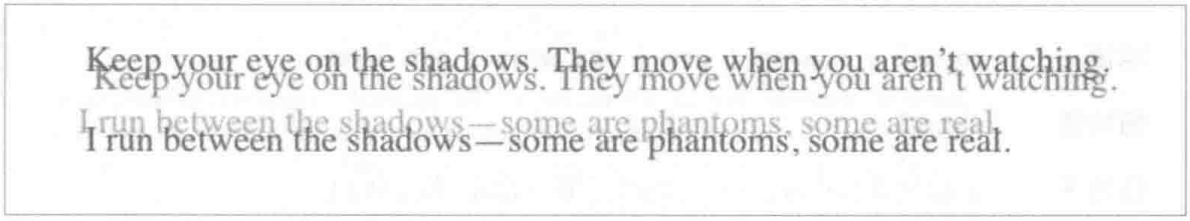


图 6-35：简单的阴影

可选的第三个长度值定义阴影的模糊半径（blur radius）。模糊半径的定义是，从阴影的轮廓到模糊效果边界的距离。如果模糊半径为 2 像素，模糊效果将填满阴影轮廓到模糊边界之间的空间。具体的模糊方法没有规定，因此不同的用户代理可能会采用不同的模糊效果。举几个例子。下述样式渲染后的效果如图 6-36 所示。

```
p.cl1 {color: black; text-shadow: gray 2px 2px 4px;}
p.cl2 {color: white; text-shadow: 0 0 4px black;}
p.cl3 {color: black; text-shadow: 1em 0.5em 5px red, -0.5em -1em hsla(100,75%,
   25%,0.33);}
```

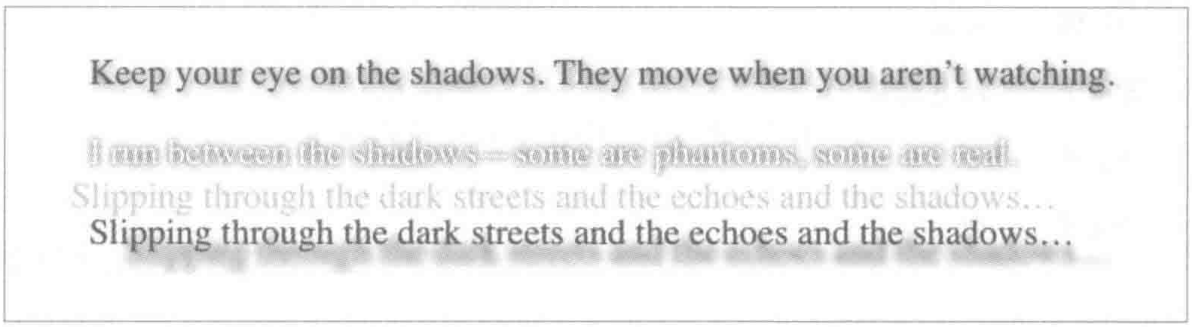


图 6-36：到处都是阴影



注意，大量阴影或模糊半径较大的文本阴影会损耗性能，尤其是在低功耗和 CPU 能力有限的情况下，例如移动设备。建议创作人员进行充分的测试之后再投放使用阴影的设计。

6.8 处理空白

我们介绍了装饰文本的不同方式，下面讨论 white-space 属性，它影响用户代理对文档源码中空格、换行符和制表符的处理方式。

white-space

取值	normal | nowrap | pre | pre-wrap | pre-line
初始值	normal
适用于	全部元素（CSS 2.1）；块级元素（CSS1 和 CSS2）
计算值	指定的值
继承性	否
动画性	否

这个属性影响浏览器对待单词之间及文本行之间空白的方式。XHTML 在某种程度上已经做了处理，即把空白压缩成一个空格。因此，对下述标记来说，经 Web 浏览器渲染后，各单词之间只有一个空格，而且会忽略元素中的换行。

```
<p>This    paragraph    has      many spaces          in it.</p>
```

这种默认的行为可以使用下述声明显式设定：

```
p {white-space: normal;}
```

这个规则让浏览器采用当前的方式处理空白，即丢掉多余的空白。这种情况下，换行符（回车）变成空白，而且连续的多个空格变成一个空格。

然而，我们可以把 white-space 设为 pre，这样目标元素中的空白将像 XHTML pre 元素那样处理，空白不会被忽略，如图 6-37 所示。

```
p {white-space: pre;}
<p>This    paragraph    has      many spaces          in it.</p>
```

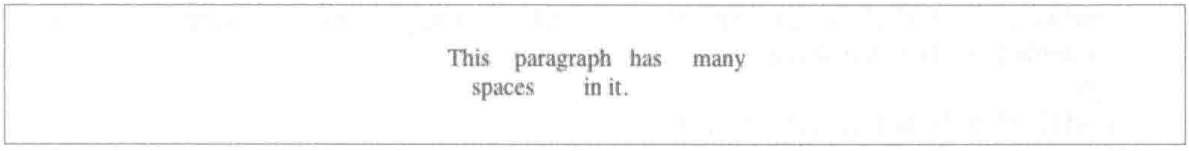


图 6-37：保留标记中的空白

设为 pre 时，浏览器会小心处理额外的空格以及回车。任何元素在这一面，而且仅在这一方面，可以变得像 pre 元素那样。

与之相反的值 nowrap，禁止元素中的文本换行，除非使用的是 br 元素。CSS 中的 nowrap，与 HTML 4 中使用 <td nowrap> 设定单元格内容不换行差不多，只不过 nowrap 值可以用在任何元素上。下述标记的结果如图 6-38 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>
```

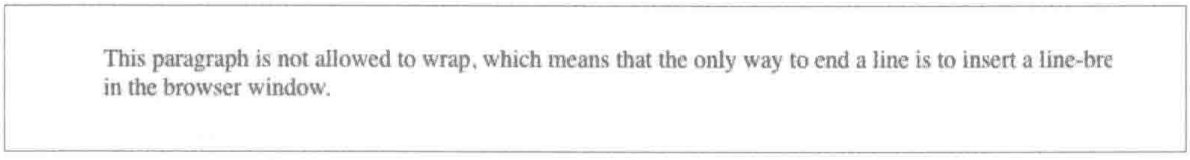


图 6-38：使用 white-space 属性禁止换行

其实，单元格的 nowrap 属性可以换成 white-space：

```
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>
```

CSS 2.1 引入了 pre-wrap 和 pre-line 两个值，这是之前的版本没有的。创作人员使用这两个值可以更好地控制空白的处理方式。

设为 pre-wrap 时，文本中的空白序列得以保留，但是文本行将正常换行。此时，源码中的换行符以及生成的换行符也保留下来。pre-line 的作用与 pre-wrap 相反，空白序列将像常规文本中那样折叠，但是保留换行。例如，下述标记的结果如图 6-39 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>
```

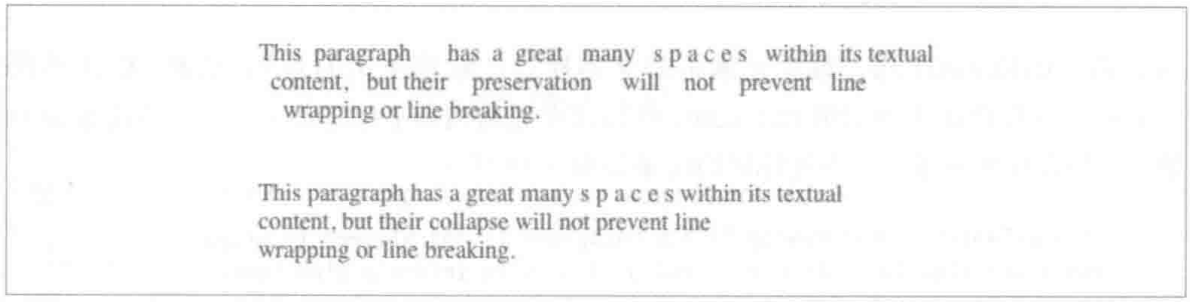


图 6-39：处理空白的两种方式

表 6-1 总结了 white-space 属性的行为。

表 6-1：white-space 属性

值	空白	换行符	自动换行
pre-line	折叠	保留	允许
normal	折叠	忽略	允许
nowrap	折叠	忽略	禁止
pre	保留	保留	禁止
pre-wrap	保留	保留	允许

设定制表符的宽度

既然 white-space 取某些值时空白会保留下来，那么制表符（即 Unicode 码位 0009）也就将显示成制表符了。但是一个制表符等于多少个空格呢？这就要问 tab-size 属性了。

tab-size

取值	*<length>* | *<integer>*
初始值	8
适用于	块级元素
计算值	指定值对应的绝对长度
继承性	是
动画性	是

默认情况下，一个制表符相当于八个连续的空格。不过，可以使用 tab-size 属性设为其他整数值。例如，tab-size: 4 将把一个制表符渲染成四个连续空格。

如果提供的是长度值，那么制表符将渲染成指定的长度。例如，tab-size: 10px 将把三个连续的制表符渲染成 30 像素宽的空白。下述规则的效果如图 6-40 所示。

```
p {white-space: pre-wrap; margin: 0.5em;}
code {font: 1em monospace, serif;}
.cl01 {tab-size: 8;}
.cl02 {tab-size: 4;}
.cl03 {tab-size: 2;}
.cl04 {tab-size: 0;}
.cl05 {tab-size: 8; white-space: normal}
```

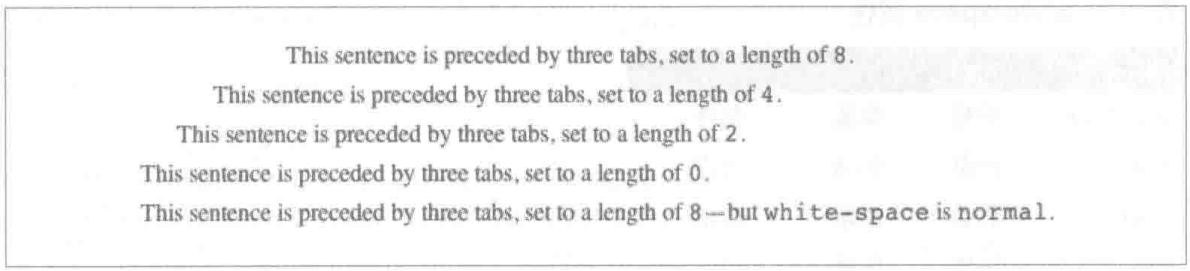


图 6-40：不同的制表符长度

注意，如果 white-space 的值导致空白被折叠了（见表 6-1），那么 tab-size 就不再起作用。此时，tab-size 的值还要计算，但是不管源码中有多少制表符，在视觉上都没有体现。



目前，WebKit 和 Gecko（通过 -moz-tab-size）支持 tab-size 属性。不过，只支持整数值，不支持长度值。

6.9 换行和断字

遇到单词较长，而行的长度较短时，最好加上连字符，例如移动设备中的博客文章和《经济学人》的一栏。创作人员可以自己动手插入断字提示（使用 Unicode 字符 U+00AD SOFT HYPHEN，或者 HTML 中的 ），但是借助 CSS，无需改动文档就能自动断字。

hyphens

取值	manual | auto | none
初始值	manual
适用于	所有元素
计算值	指定的值
继承性	是
动画性	否

使用默认值 manual 时，只在文档中手动插入的连字符（例如 U+00AD 或 ）处断字；否则，不断字。而使用 none 值时，即使有手动插入的连字符，也不断字；即，U+00AD 和 将被忽略。

auto 值有趣多了（行为也不太一致），即使没有手动插入连字符，浏览器也会在单词中“合适的”位置断字，把单词分开。这就带来了一些有趣的问题，比如怎样定义“单词”，以及何为断字的“合适”位置。其实，这两个问题完全由所用的语言决定。用户代理应该优先在手动插入的连字符处断字，但情况并非总是如此。下述示例设定的断字及禁止断字的效果如图 6-41 所示：

```
.cl01 {hyphens: auto;}
.cl02 {hyphens: manual;}
.cl03 {hyphens: none;}

<p class="cl01">Supercalifragilisticexpialidocious antidisestablishmentarian
    ism.</p>
<p class="cl02">Supercalifragilisticexpialidocious antidisestablishmentarian
    ism.</p>
<p class="cl02">Super&#xad;cali&#xad;fragi&#xad;listic&#xad;expi&#xad;ali&#xad;
docious anti&#xad;dis&#xad;establish&#xad;ment&#xad;arian&#xad;ism.</p>
<p class="cl03">Super&#xad;cali&#xad;fragi&#xad;listic&#xad;expi&#xad;ali&#xad;
docious anti&#xad;dis&#xad;establish&#xad;ment&#xad;arian&#xad;ism.</p>
```

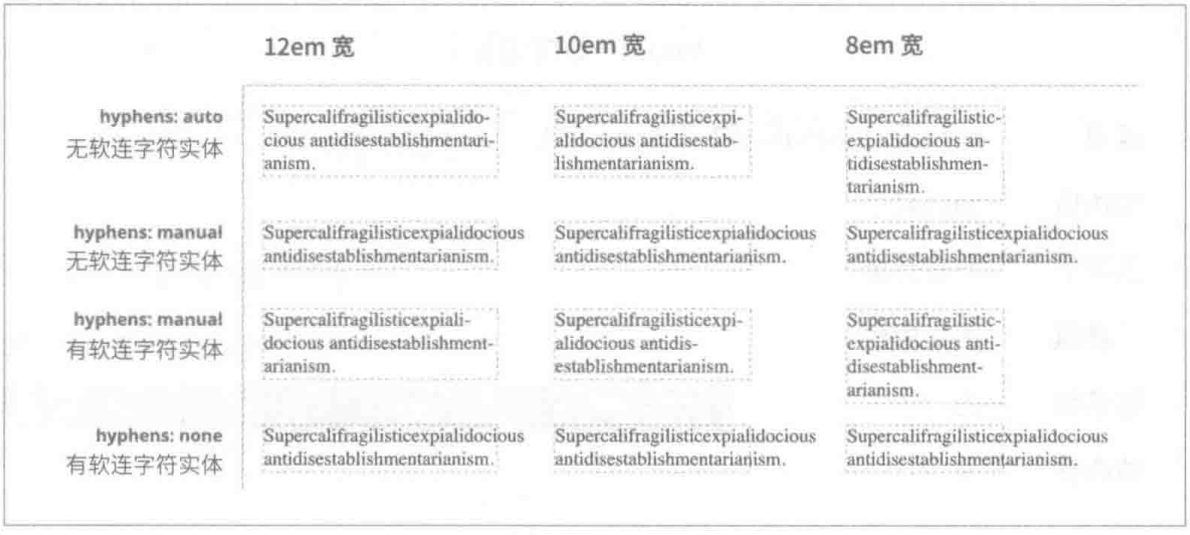


图 6-41：不同的断字效果

因为断字与所用的语言有很大关系，而且 CSS 规范没有准确定义用户代理应该如何断字（连模糊说明都没有），所以在不同的浏览器中断字的方式极有可能不同。

此外，如果选择断字，要谨慎选择在什么元素上断字。hyphens 属性能被继承，因此

body {hyphens: auto;} 声明将在文档中的所有元素上应用断字，包括多行文本输入框、代码示例、引用块等。这些元素最好通过类似下面的规则禁止自动断字：

```
body {hyphens: auto;}
code, var, kbd, samp, tt, dir, listing, plaintext, xmp, abbr, acronym,
blockquote, q, textarea, input, option {hyphens: manual;}
```

代码示例和代码块禁止断字的原因应该很明显，尤其是那些使用连字符表示属性和值名称的语言。通过键盘输入的文本也是如此，你肯定不希望 UNIX 命令行示例中出现误导人的连字符。类似的情况还有很多。如果确实想在某些元素中断字，从选择符中把它们删掉即可（在多行文本输入框中输入的过程中自动断字看起来还是不错的）。



截至 2017 年年末，主流桌面浏览器中除了 Chrome/Blink 之外都支持 hyphens 属性，但是在 Safari 和 Edge 中要使用厂商前缀。再次注意，断字与所用的语言有很大关系。

断字还受其他属性的影响，例如 word-break。这个属性的作用是控制不同语言处理文本软换行的方式。

word-break

取值	normal | break-all | keep-all
初始值	normal
适用于	所有元素
计算值	指定的值
继承性	是
动画性	是

如果一串文本太长，一行里放不下，就会软换行。这个概念与换行符和
 元素导致的硬换行是相对的。文本在何处软换行由用户代理（或操作系统）决定，但是创作人员可以使用 word-break 属性影响这一决定。

默认值 normal 的意思是，按正常方式换行。通俗易懂地说，就是文本在单词之间换行，不过不同语言中“单词”的定义有所不同。在拉丁语系语言中，例如英语，几乎始终在字母序列（例如单词）之间的空格处换行。在象形文字中，例如日语，一个字符就是一个词，因此任何两个字符之间都有可能换行。然而，在其他 CJK 语言中，软换行可能只会出现在不以空格分隔的字符序列之间。

再次说明，这是浏览器处理文本的默认方式，已经沿用了数年。如果使用 break-all，软换行可能（也会）出现在任何字符之间，即使是在一个词的内部。使用这个值时，连字符不显示，即使换行出现在断字的位置（参见前文对 hyphens 属性的说明）。注意，line-break 属性（参见下文）可能会影响 CJK 文本应用 break-all 的行为。

keep-all 禁止在字符之间软换行，即便对一个符号一个词的 CJK 语言来说也是如此。因此，在日语中，没有空白的字符序列不会出现软换行，即便这样可能导致文本行超出所在元素的长度（这个行为类似于 white-space: pre）。

图 6-42 展示了 word-break 属性不同值的效果，表 6-2 对各个值的行为做了总结。



图 6-42：调整词内换行行为

表 6-2：词内换行行为

值	非 CJK	CJK	是否断字
normal	照常	照常	是
break-all	任何字符之后	任何字符之后	否
keep-all	照常	序列两侧	是

如果你关注的是 CJK 文本，除了 word-break 属性之外，可能还想了解 line-break 属性。

line-break

取值	auto | loose | normal | strict
初始值	auto
适用于	所有元素
计算值	指定的值
继承性	是
动画性	是

我们知道，word-break 属性可以影响 CJK 文本行的软换行方式。line-break 属性也能影响这种软换行，尤其是如何处理 CJK 符号两侧的换行，以及 CJK 文本中非 CJK 标点符号（例如感叹号、连字符和省略号）两侧的换行方式。

也就是说，line-break 始终作用于特定的 CJK 字符，而不管内容被声明为什么语言。如果在一段英文中混杂部分 CJK 字符，line-break 仍将应用于那些 CJK 字符，而不应用于文本中的其他字符。此外，如果把内容声明为某种 CJK 语言，line-break 依然将应用于那些 CJK 字符，以及 CJK 文本中的非 CJK 字符，包括标点符号、货币符号和一些其他字符。

哪些字符受这个属性的影响，哪些不受影响，没有权威的列表，不过规范给出了建议受此属性影响的符号及其行为（http://w3.org/TR/css3-text/#line-break）。

默认值 auto 指示用户代理按正常行为软换行，而更为重要的是，让用户代理根据情况变更换行方式。例如，用户代理在处理较短的行时采用宽松的换行规则，在处理较长的行时采用严格的规则。其实，auto 的作用是让用户代理根据需求在 loose、normal 和 strict 之间切换，甚至可以在一个元素的不同行之间切换。

你几乎可以从其他几个值的字面意思推断出它们的基本效果：

loose
: 采用“最宽松的”换行规则，主要用于文本行较短的情况，例如报纸。

`normal`

采用“最常规的”换行规则。什么是“最常规的”规则，没有准确定义，不过前面说过，规范中有建议的行为。

`strict`

采用“最严格的”换行规则。同样，何为“最严格的”，也没有准确定义。

文本换行

了解断字和软换行之后，我们不禁会想，如果文本超出了所在容器怎么办？这正是 `overflow-wrap` 属性要解决的问题。

overflow-wrap（原 word-wrap）

取值	`normal` | `break-word`
初始值	`normal`
适用于	所有元素
计算值	指定的值
继承性	是
动画性	是

这个属性的作用再简单不过了。使用默认值 `normal` 时，按正常方式换行，即在单词之间换行，或者由所用的语言决定。使用 `break-word` 时，可以在单词的内部换行。图 6-43 展示了这两个值之间的区别。

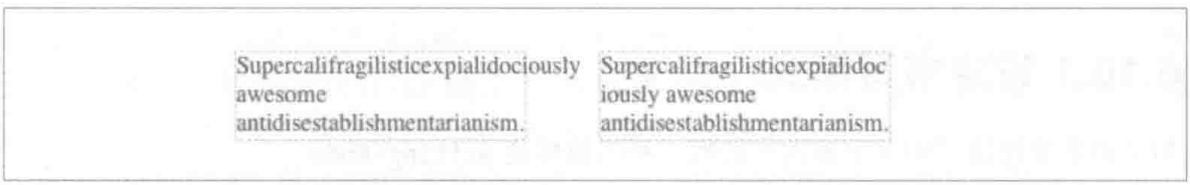


图 6-43：溢出换行



注意，只有 white-space 属性的值允许换行时，overflow-wrap 才会起作用。如果 white-space 禁止换行（例如设为 pre），那么 overflow-wrap 没有任何效果。

overflow-wrap 的复杂性在于其历史和实现。以前，有个名为 word-wrap 的属性，其作用与 overflow-wrap 完全一样。鉴于此，规范明确指出，用户代理“必须把 word-wrap 视作 overflow-wrap 属性的别名，当作 overflow-wrap 的简写”。

可惜，浏览器并没有完全执行这个规定，word-wrap 得到的支持更广。因此，为了向后兼容，通常二者同时使用：

```
pre {word-wrap: break-word; overflow-wrap: break-word;}
```

截至 2017 年年末，overflow-wrap 得到了非常广泛的支持，可以放心使用了。

overflow-wrap: break-word 和 word-break: break-all 看似作用一样，然而并非如此。为了了解二者的区别，请看图 6-43 中的第二个文本框和图 6-42 中第一排中间那个文本框。可以看出，仅当内容有溢出时，overflow-wrap 才起作用；因此，如果能用源码中的空白换行，overflow-wrap 就会在空白处换行。而 word-break: break-all 不同，它在内容接触边界时换行，不管前面有没有空白。

6.10 书写模式

如果你阅读的是本书英文版，或者其他主流西方语言的版本，那么文本是从左至右、从上到下排列的，这正是英语的流动方向。然而，不是每种语言都是如此。有些语言是从右至左、从上到下排列的，例如希伯来语和阿拉伯语，还有些语言主要采用从上到下的方式书写，此外还可以从左至右书写，例如汉语和日语，也可以从右至左书写，例如蒙古语。

6.10.1 设定书写模式

用于指定采用这三种书写模式中的哪一种的属性是 writing-mode。

writing-mode	
取值	horizontal-tb | vertical-rl | vertical-lr
初始值	horizontal-tb
适用于	除表格行组、表格列组、表格行、表格列、旁注基元素和旁注元素之外的所有元素
计算值	指定的值
继承性	是
动画性	是

默认值 horizontal-tb 的意思是，“行内方向为横向，块级方向为从上到下”。这涵盖了所有西方语言和部分中东语言，后者在横向书写时可能采用不同的方向。后两个值采用的行内方向是纵向，不过块级方向是从右至左或从左至右。这三个值的效果如图 6-44 所示。

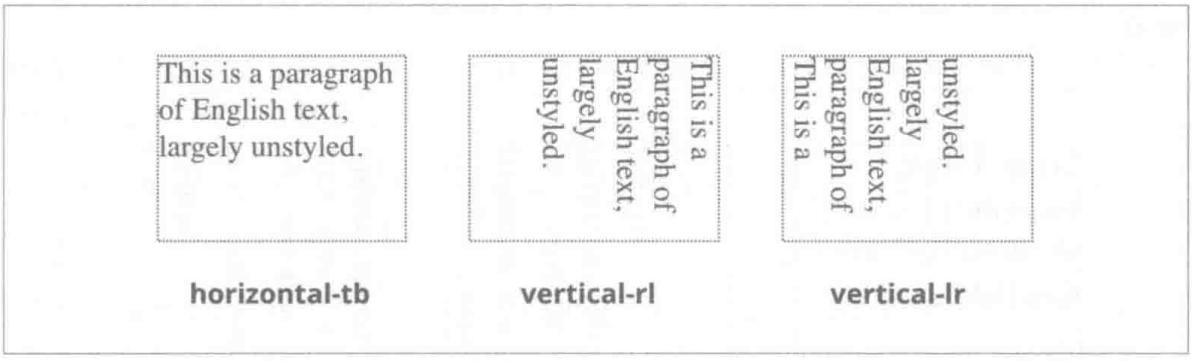


图 6-44：不同的书写模式

注意，两个纵向示例中的各行是串列的。如果把头向右歪，vertical-rl 模式下的文本还算都认得清。而 vertical-lr 模式下的文本就不那么好认了，因为文本是从下向上流动的（至少对英语来说是这样）。这对原本就采用 vertical-lr 流动方式的语言来说不是问题，例如日语。截至 2017 年年末，纵向模式下的行内方向只能是从上到下。

如果想把西方语言的纵向流动方式改成从下到上，可以先在元素上应用 vertical-rl，然后借助 CSS 变形技术（参见第 16 章）把元素旋转 180°，从而得到从下到上、从左至右的视觉效果。使用 vertical-lr，加上旋转之后，可以得到从下到上、从右至左的流动方式。这两种情况如图 6-45 所示。

```
.flip {transform: rotate(180deg);}
#one {writing-mode: vertical-rl;}
#two {writing-mode: vertical-lr;}
```

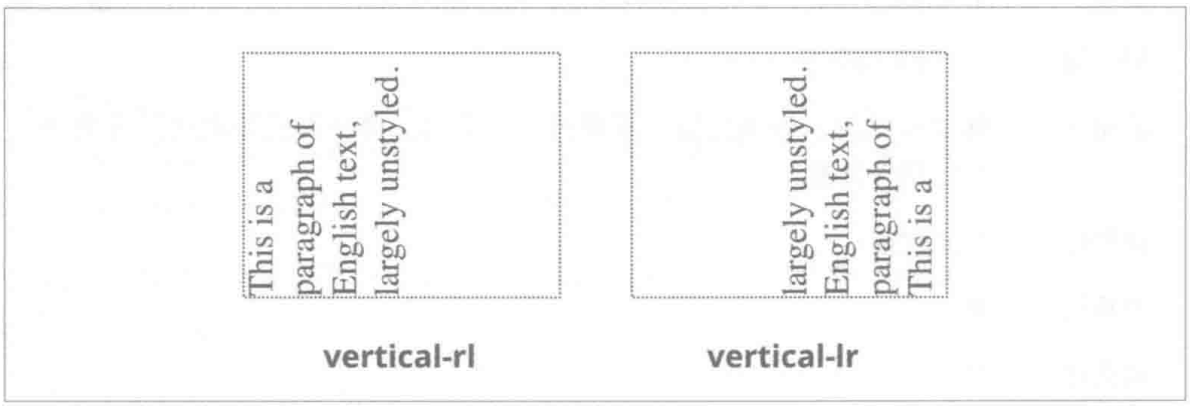


图 6-45：倒置纵向书写模式

这样排列文本有违直觉，因为一切都倒置了，看到的与指定的是有出入的。例如，vertical-rl（纵向从右至左）模式下的文本是从左至右流动的。

使用块级对齐属性（例如 vertical-align）时也会遇到相同的问题。在纵向书写模式下，块级方向是横向；这意味着，纵向对齐的行内元素其实是在横向上移动的，如图 6-46 所示。

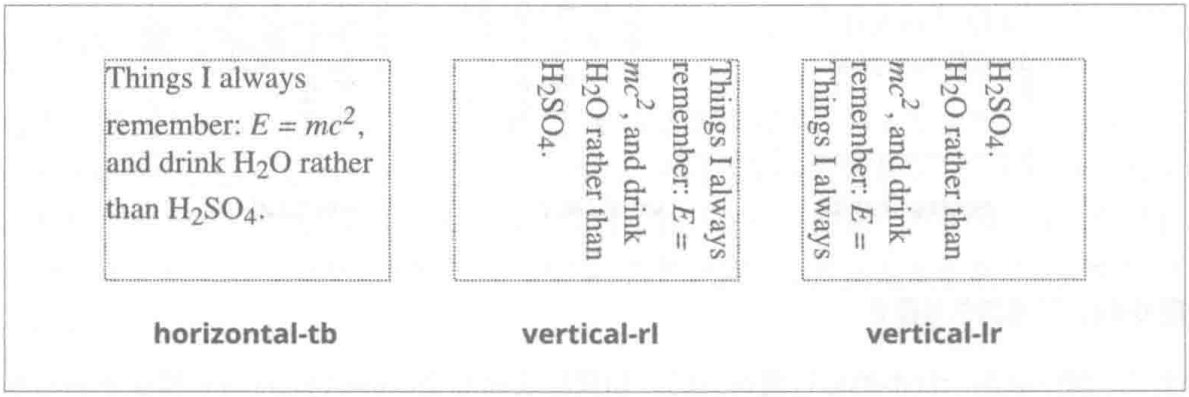


图 6-46：书写模式和“纵向”对齐

尽管上标和下标是通过 vertical-align 属性移动的，但是它们及所在行的位置都沿着横向移动。前面说过，纵向位移是相对行框而言的，而根据定义，行框的基线是横向的，就算文本是纵排的也是如此。

乱了？没关系。书写模式确实有点饶人，这是因为思维方式变了，而且 CSS 规范以前所做的假设与如今的能力有冲突。如果一开始就考虑到有纵向书写模式，vertical-align 属性可能就要改成其他名称了，比如 inline-align 之类的（说不定有一天会这样改）。

最后说明一点。如果你用过 CSS 变形，可能觉得纵向书写模式相当于把文本旋转 90°。其实，这两种情况是不同的，原因有二。其一，只有 `vertical-rl` 模式是这样；而在 `vertical-lr` 模式下，文本是“从下到上”流动的。其二，纵向流动的文本基点不变；也就是说，顶还是顶。下面举个例子，结果如图 6-47 所示。

```
.boxed {border-top: 3px solid red;
        border-left: 3px dashed tan;}
#one {writing-mode: vertical-rl;}
#two {writing-mode: vertical-lr;}
```

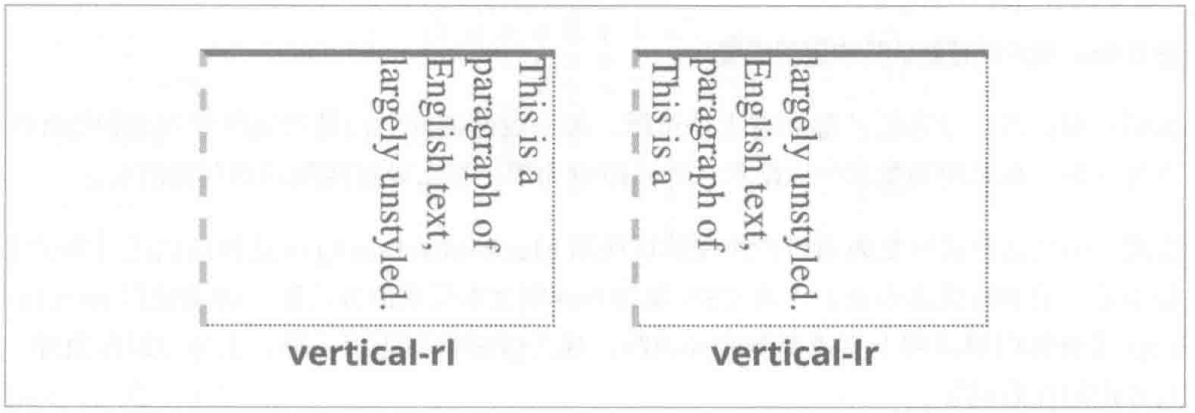


图 6-47：不同书写模式下的 CSS 基点方向

两种情况下，顶边都是红色实线，左边都是棕黄色虚线。基点没有随文本旋转，因为文本并没有旋转，而是改变了流动方向。

下面开始出现反常行为了。虽然边框不绕着元素所在的方框转动，但是外边距会转动，而这不是 CSS 规范规定的。

这是因为用户代理的内部样式在块级方向上是相对元素起边和终边的（至少截至 2017 年年末是这样）。对从上到下流动的语言来说，元素块级方向的起边和终边分别是顶边和底边。而在纵向书写模式下，块级方向是从右至左或从左至右。因此，如果把几个段落改成纵排后，没有修改外边距，那么正常情况下的上下外边距将变成左右外边距。这种情况如图 6-48 所示，图示效果用的是下述样式：

```
p { margin-top: 1em;}
#one {writing-mode: vertical-rl;}
#two {writing-mode: vertical-lr;}
```

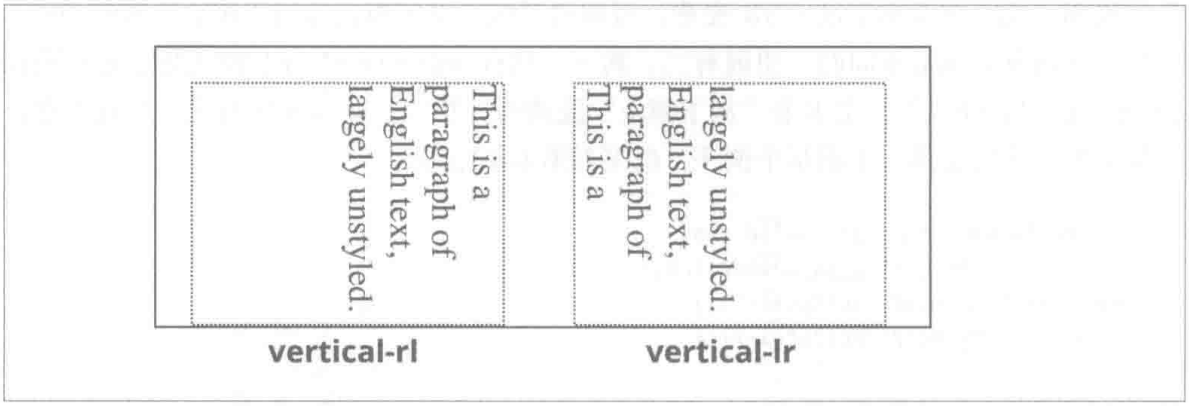


图 6-48：用户代理默认外边距的位置

根据声明，两个段落的顶部都有上外边距。每一段两侧的空白是沿块级方向的起边和终边外边距。如果明确把右外边距和左外边距设为零，那么段落两侧的空白将消失。

注意，出现这种情况是因为用户代理默认使用 block-start-margin 这样的属性（其实这并不是，至少目前还不是正式的 CSS 属性）声明文本元素的外边距。如果使用 margin-top 等属性明确声明上下或两侧的外边距，那么情况将与边框一样，上外边距在顶部，右外边距在右侧等。

6.10.2 改变文本方向

选定书写模式之后，可能还想改变文本行中字符的方向。这么做的原因有很多，尤其是多种书写系统混杂时，例如日语中掺杂着英语单词或数字。此时，要使用 text-orientation 属性。

text-orientation

取值	mixed | upright | sideways
初始值	mixed
适用于	除表格行组、表格行、表格列组和表格列之外的所有元素
计算值	指定的值
继承性	是
动画性	是

text-orientation 属性的作用是控制字符的方向。具体效果最好通过示例说明，下述样式渲染得到的结果如图 6-49 所示：

```
.verts {writing-mode: vertical-lr;}
#one {text-orientation: mixed;}
#two {text-orientation: upright;}
#thr {text-orientation: sideways;}
```

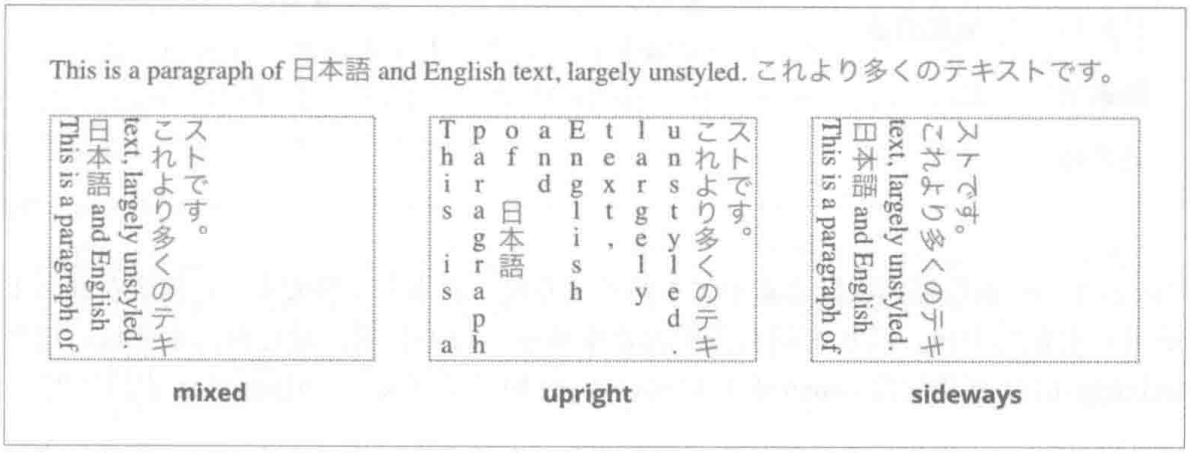


图 6-49：文本方向

图 6-49 上部是日语和英语文本混合的一个段落，基本没什么样式。下面三个副本使用书写模式 vertical-lr。第一个副本使用 text-orientation: mixed，横向书写的字符（英语）侧排，纵向书写的字符（日语）直排。第二个副本中所有字符都是直排的（upright），包括英语字符。第三个副本中所有字符都是侧排的（sideways），包括日语字符。

6.10.3 声明方向

回到 CSS2。这一版提供了两个用于控制文本方向的属性，它们会改变行内基线的方向。这两个属性是 direction 和 unicode-bidi。



CSS 规范明确指出，不建议把 CSS 的 direction 和 unicode-bidi 属性用在 HTML 文档上。原话是这样的："因为 HTML（或用户代理）能够禁用 CSS 样式，所以我们建议使用 HTML 的 dir 属性和 <bdo> 元素，确保在缺少样式表的情况下能得到正确的双向布局。"本节介绍这两个属性是因为在旧样式表中可能还有它们的身影。

direction

取值	ltr|rtl
初始值	ltr
适用于	所有元素
计算值	指定的值
继承性	是
动画性	是

direction 属性影响块级元素中文本的书写方向，表格列布局的方向，内容在横向上溢出元素框的方向，以及两端对齐的元素中最后一行的位置。对行内元素来说，仅当 unicode-bidi 属性设为 embed 或 bidi-override 时（见下文），direction 才起作用。

默认值为 ltr，但是如果浏览器显示的是从右至左的文本，应该变成 rtl。因此，浏览器的内部样式中可能有类似下面的规则：

```
*:lang(ar), *:lang(he) {direction: rtl;}
```

真实的规则比这长，涵盖所有从右至左书写的语言（不限于阿拉伯语和希伯来语），不过意思很明确了。

CSS 为解决书写方向做出了努力，但 Unicode 处理方向的方式要复杂得多。借助 unicode-bidi 属性，CSS 创作人员可以利用 Unicode 的部分功能。

unicode-bidi

取值	normal | embed | bidi-override
初始值	normal
适用于	所有元素
计算值	指定的值
继承性	否
动画性	是

下面直接引用 CSS2.1 规范中对这些值的说明，规范已经点出了各个值的实质：

`normal`

根据双向算法（bidirectional algorithm），目标元素不打开新的嵌套层级。对于行内元素，跨元素边界隐式重排。

`embed`

对于行内元素，根据双向算法，打开新的嵌套层级。这一级嵌套的方向由 `direction` 属性指定。元素内部隐式重排，在元素开头添加一个 LRE 字符（U+202A；针对 `direction: ltr`）或一个 RLE 字符（U+202B；针对 `direction: rtl`），在元素末尾添加一个 PDF 字符（U+202C）。

`bidi-override`

覆盖行内元素的书写方向。对于块级元素，覆盖行内后代的书写方向。这意味着，元素中的文本严格按照 direction 属性依序重排，而双向算法所做的隐式重排将被忽略。此时，在元素的开头添加一个 LRO 字符（U+202D；针对 direction: ltr）或一个 RLO 字符（U+202E；针对 `direction: rtl`），在元素的末尾添加一个 PDF（U+202C）字符。

6.11 小结

即使不改变所用的字型，也有很多方式能改变文本的外观。除了经典的效果（如下划线）之外，借助 CSS 还能为文本添加上划线或贯穿线，改变单词和字母之间的距离，缩进段落（或其他块级元素）的首行，以多种方式对齐文本，以及控制文本的断字和换行等。甚至还可以调整文本行之间的距离。CSS 还能处理除从左至右、从上到下书写的语言之外的其他语言。既然 Web 的世界充斥着文本，有这么多控制文本的属性也就不奇怪了。相关人员一直在改善文本的清晰性和对位置的控制能力，未来，装饰文本的功能将越来越丰富。

第 7 章

视觉格式化基础

本章全面阐述 CSS 视觉渲染的理论。为什么要了解这些知识呢？因为 CSS 是如此开放、如此强大的一个系统，任何书都不可能涵盖属性的所有组合方式及其效果。在实践中，你肯定会发现使用 CSS 的新方式。而且在探索 CSS 的过程中，你可能会在某些用户代理中遇到看似奇怪的行为。掌握视觉渲染模型的工作原理之后，你便能判断所遇到的怪异行为（尽管出乎意料）就是 CSS 渲染引擎定义的正确方式，还是确有异常。

7.1 元素框基础

不管是什么元素，CSS 都假定每个元素都生成一个或多个矩形框，我们称之为元素框（element box）（以后的规范可能会允许元素生成非矩形框，而其实现在就有这方面的提议，不过目前生成的框还都是矩形的）。各元素框的中心是内容区域（content area），四周有可选的内边距、边框、轮廓和外边距。之所以说这些区域是可选的，是因为它们的宽度都可以设为零，即把它们从元素框上删除。图 7-1 展示了一个内容区域，四周围绕着内边距、边框和外边距。

外边距、边框和内边距都有分别针对每一边的属性，例如 `margin-left` 或 `border-bottom`，也有简写属性，例如 `padding`。而轮廓没有针对各边的属性。默认情况下，内容区的背景（例如颜色或平铺的图像）出现在内边距范围内。外边距区域始终是透明的，因此透过它能看到父元素。内边距不能为负值，但是外边距可以。稍后会说明把外边距设为负值的效果。

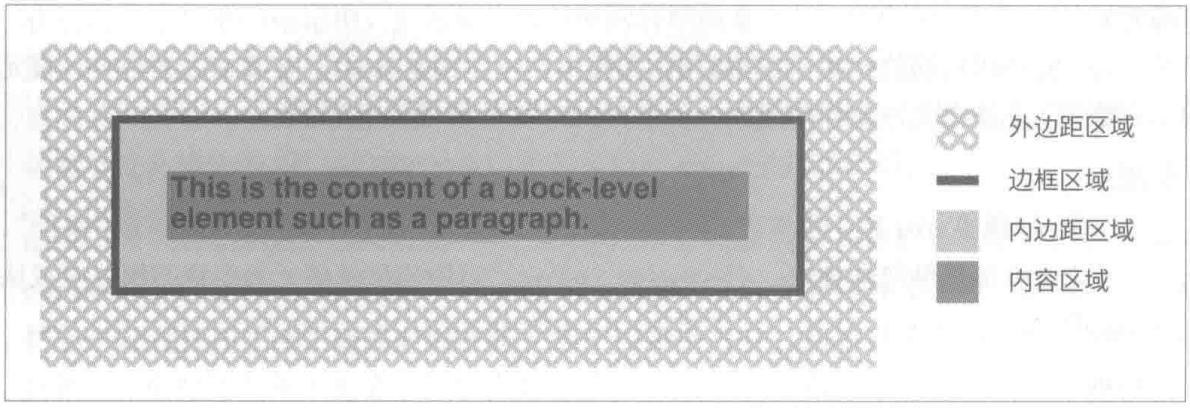


图 7-1：内容区域及其四周

边框的线型由样式定义，例如设为 solid 或 inset，边框的颜色由 border-color 属性设定。如未设定颜色，边框的颜色与元素中内容的前景色相同。比如说，段落中的文本是白色，如果创作人员没有明确声明使用其他颜色，那么段落四周的边框也是白色的。如果边框的线型有间隙，那么默认情况下，从间隙中能看到元素的背景。最后，边框的宽度不能为负数。

元素框各组成部分受很多属性的影响，例如 width 或 border-right。本书将使用其中多数属性，但是不会深入讨论。

7.1.1 重要概念概览

下面简要说明一下我们将讨论的不同框体，以及后文将用到的重要术语：

常规流动

即渲染西方语言时从左至右、从上到下的顺序，以及传统的 HTML 文档采用的文本布局方式。注意，对非西方语言来说，流动方向可能会变。多数元素都采用常规的流动方式，除非元素浮动了、定位了，放入弹性盒或采用栅格布局了。不过本章只讨论常规流动方式下的元素。

非置换元素

内容包含在文档中的元素。例如，段落（p）是非置换元素，因为段落中的文本内容在元素自身中。

置换元素

为其他内容占位的元素。典型的置换元素是 img，它指向一个图像文件，那个图像就插在 img 元素在文档流中的位置上。多数表单元素也是置换元素（例如 <input type="radio">）。

根元素

位于文档树顶端的元素。在 HTML 文档中，根元素是 `html`。在 XML 文档中，根元素可以是语言允许的任何元素，例如，RSS 文件的根元素是 `rss`。

块级框

段落、标题或 `div` 等元素生成的框。在常规流动模式下，块级框在框体前后都“换行”，因此块级框是纵向堆叠的。`display: block` 声明能把任何元素生成的框体变成块级框。

行内框

`strong` 或 `span` 等元素生成的框体。行内框前后不“换行”。`display: inline` 声明能把任何元素生成的框体变成行内框。

行内块级框

内部特征像块级框，外部特征像行内框。行内块级框的行为与置换元素相似，但不完全相同。比如说把一个 `div` 元素像行内图像那样插入一行文本，这样一想你就明白了。

除此之外还有其他框体，例如单元格框，但是基于各种各样的原因（那些框体太过复杂，一本书也讲不完，而且没有多少作者能驾驭，只这一点就够了），本书不做讨论。

7.1.2 容纳块

还有一种框体要深入说明，因为涉及的知识够多，所以单开一节。这种框体是容纳块（containing block）。

每个元素的框体都相对容纳块放置，说得简单点就是，容纳块是元素框体的“布局上下文”。为了确定框体的容纳块，CSS 定义了一系列规则。不过我们只会说明与本书内容有关的那些规则，以免话题展开太多。

在使用常规流动方式渲染的西文文本中，容纳块由离元素最近的那个生成列表项目或块级框（包含所有与表格有关的框体，例如单元格生成的框体）的祖辈元素的边界构成。以下述标记为例：

```
<body>
    <div>
        <p>This is a paragraph.</p>
    </div>
</body>
```

在这段十分简单的标记中，p 元素的块级框的容纳块是 div 元素的块级框，因为这是祖辈元素的框体中离 p 元素最近的，而且生成的是块级框或列表项目（这里，div 元素生成的是块级框）。类似地，div 元素的容纳块是 body 元素的框体。因此，p 元素的布局依赖 div 元素的布局，而 div 元素的布局又依赖 body 元素的布局。

继续下去，body 元素的布局依赖 html 元素的布局。html 元素对应的是初始容纳块（initial containing block），它的独特之处是，其尺寸由视区（屏幕媒体中的浏览器窗口，印刷媒体中的可打印区域）决定，而非根元素中内容的尺寸。初始容纳块与其他容纳块的差异极小，而且通常并不重要，但是你要知道有这么一种容纳块。

7.2 调整元素的显示方式

为 display 属性设值可以影响用户代理显示元素的方式。既然我们开始深入讲解视觉格式化了，那就用前文学过的概念多讨论 display 属性的两个值。

display

取值	[*<display-outside>* || *<display-inside>*] | *<display-listitem>* | *<display-internal>* | *<display-box>* | *<display-legacy>*
定义	见下
初始值	inline
适用于	所有元素
计算值	指定的值
继承性	否
动画性	否

<display-outside>

```
block | inline | run-in
```

<display-inside>

```
flow | flow-root | table | flex | grid | ruby
```

<display-listitem>

```
list-item && <display-outside>? && [ flow | flow-root ]?
```

```
<display-internal>
    table-row-group | table-header-group | table-footer-group |
    table-row | table-cell | table-column-group | table-column |
    table-caption | ruby-base | ruby-text | ruby-base-container |
    ruby-text-container

<display-box>
    contents | none

<display-legacy>
    inline-block | inline-list-item | inline-table | inline-flex |
    inline-grid
```

我们将忽略旁注（ruby）和表格相关的值，因为对本章的内容来说，它们太过复杂；我们还将忽略 list-item 这个值，因为它与块级框特别相似。我们将用大量时间讨论块级框和行内框，不过在此之前先讨论调整元素显示方式对布局的影响，随后再说明 inline-block 这个值。

7.2.1 改变显示方式

装饰文档时，若能改变元素生成的框体类型显然很方便。例如，假设 nav 元素中有一系列链接，我们想纵向布局，显示为侧边栏：

```
<nav>
    <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>
</nav>
```

为此，可以把链接分别放在单元格中，也可以把每一个链接都放在一个 nav 元素中，或者直接把各个链接变成块级元素，像下面这样：

```
nav a {display: block;}
```

这个规则把 nav 元素中的每个 a 元素变成块级元素。如果再添加一些样式，可以得到类似图 7-2 所示的结果。

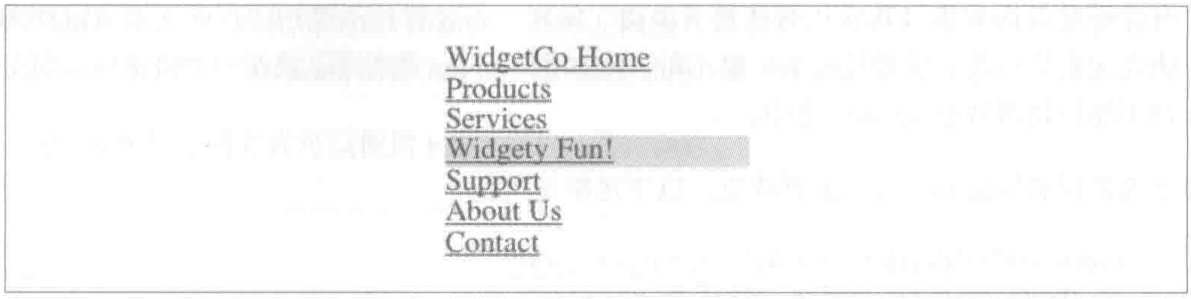


图 7-2：把显示方式由行内变成块级

这样修改元素的显示方式之后，在不支持 CSS 的浏览器中，导航链接将显示为行内元素，而在支持 CSS 的浏览器中，导航链接将显示为块级元素。把链接变成块级元素之后，可以像 div 或 p 元素那样装饰，整个元素框都变成链接的了。因此，用户把鼠标指针悬停在元素框上的任何位置都能单击链接。

此外，你可能还想把元素变成行内显示方式。假设有这么一个显示人名的无序列表：

```
<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>
```

现在我们想把这些人名显示为行内元素，而且在人名之间（以及整个列表的前后）加上竖线。为此，只能改变显示方式。下述规则的效果如图 7-3 所示。

```
#rollcall li {display: inline; border-right: 1px solid; padding: 0 0.33em;}
#rollcall li:first-child {border-left: 1px solid;}
```

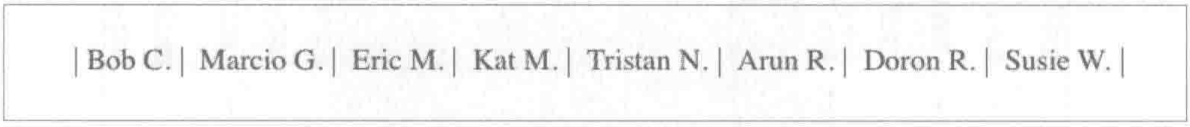


图 7-3：把显示方式由列表项目改为行内

你可以根据设计需要灵活使用不同的显示方式，只有想不到，没有做不到。

然而，要注意，我们改变的是元素的显示方式，而不是元素的本性。也就是说，把段落生成的框体变成行内框并不会把段落变成行内元素。例如，HTML 元素有些是块级元素，

有些则是行内元素（其实还有些是“流动”元素，不过暂且不提）。行内元素可以作为块级元素的后代，但是反过来一般不行。比如说，span 元素可以放在一个段落中，但是却不能把段落放在 span 元素中。

不论怎样装饰元素，这一点都成立。以下述标记为例：

```
<span style="display: block;">
<p style="display: inline;">this is wrong!</p>
</span>
```

这段标记是不对的，因为块级元素（p）嵌套在行内元素（span）中。虽然改变了显示方式，但这一点是变不了的。display 属性的名称就表明，它是影响元素显示方式的，而不能改变元素的种类。

了解这些知识之后，下面深入讨论不同的框体：块级框、行内框、行内块级框和列表项目框。

7.2.2 块级框

块级框的行为有时可以预测，有时又让人捉摸不透。例如，框体位置在横轴和纵轴上的处理方式就有所不同。为了充分理解块级框的处理方式，必须知道一些界线和区域，如图 7-4 所示。

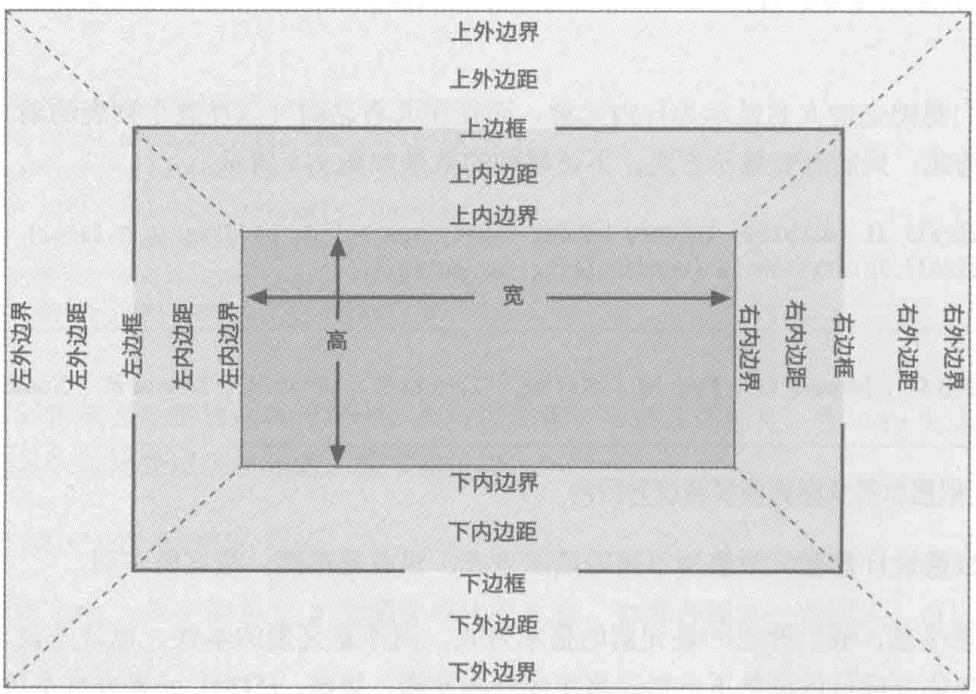


图 7-4：盒模型全解

默认情况下，块级框的宽度（width）等于左内边界到右内边界的距离，高度（height）等于上内边界到下内边界的距离。这两个属性可用于生成块级框的元素。

这些属性的处理方式可以使用 box-sizing 属性调整。

box-sizing	
取值	content-box | padding-box | border-box
初始值	content-box
适用于	能设定 width 或 height 的所有元素
计算值	指定的值
继承性	否
动画性	否

这个属性用于改变 width 和 height 值的具体意义。如果声明 width: 400px，而且不为 box-sizing 设值，那么元素的内容框将为 400 像素宽，内边距和边框等都在此基础上增加。如果声明 box-sizing: border-box，那么元素框从左边框的外边界到右边框的外边界相距 400 像素；边框或内边距都在这个尺寸之内计算，即内容区的宽度相应减少。这两种情况如图 7-5 所示。

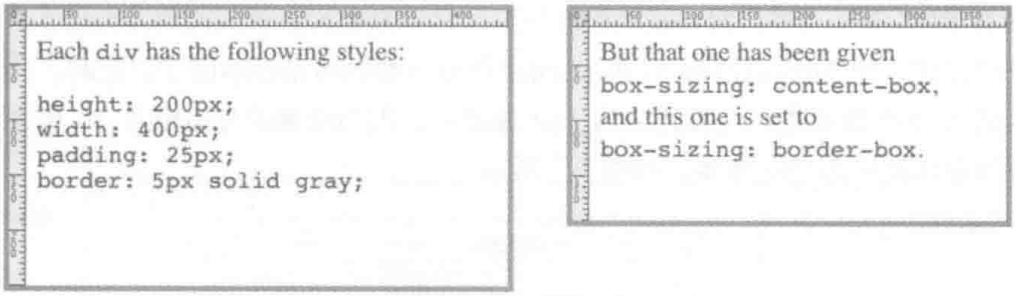


图 7-5：box-sizing 的效果

这里提到 box-sizing 属性，是因为它作用于“能设定 width 或 height 的所有元素”。通常，这是指生成块级框的元素，不过也能用于行内置换元素（如图像），以及行内块级框。

这些宽度、高度，以及内边距和外边距在一起，决定着文档的布局方式。多数情况下，

文档的高度和宽度由浏览器自动确定，而且是根据可用显示区域确定的，此外还受一些其他因素的影响。通过 CSS，可以直接控制元素的尺寸和显示方式。

7.2.3 横向格式化

横向格式化往往比你想的复杂。之所以复杂，部分是由于 box-sizing 的默认行为。使用默认值 content-box 时，为 width 设定的值是内容区的宽度，而不是整个元素框的可见宽度。以下述标记为例：

```
<p style="width: 200px;">wideness?</p>
```

这个段落中的内容为 200 像素宽。如果有背景，看的就相当清楚了。此时，如果有内边距、边框或外边距的话，将增加到这个宽度之上。假设我们是这样设定的：

```
<p style="width: 200px; padding: 10px; margin: 20px;">wideness?</p>
```

那么，元素框的可见区域将是 220 像素宽，因为我们在内容区的左右各添加了 10 像素宽的内边距。外边距又在元素的两侧添加 20 像素，因此这个元素框的总体宽度为 260 像素。结果如图 7-6 所示。

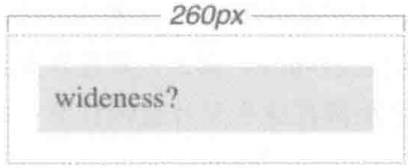


图 7-6：加上内边距和外边距

如果修改样式，把 box-sizing 设为 border-box，那么结果就不同了。此时，元素框的可见区域为 200 像素宽，内容区域为 180 像素宽。因为两侧的外边距共 40 像素，所以元素框的总体宽度为 240 像素，如图 7-7 所示。

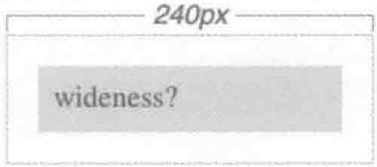


图 7-7：减去内边距

在这两种情况下，常规流动方式下块级框各组成部分的横向尺寸始终等于容纳块的宽度。假设一个 div 元素中有两个段落，div 元素的外边距为 1em，box-sizing 属性使用默认值。

那么，每个段落的内容区宽度（即 width 的值），加上左右内边距、边框和外边距，得到的和始终等于 div 元素内容区的宽度。

假设 div 元素的宽度为 30em。那么，段落的内容区宽度，加上内边距、边框和外边距之和为 30 em。图 7-8 中段落四周的“空白”其实是段落的外边距。如果 div 元素有内边距，空白区域更大，不过这里没有。

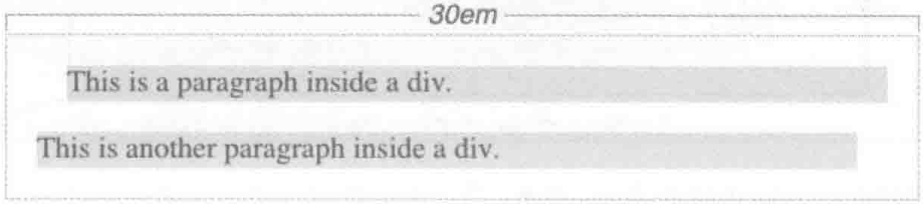


图 7-8：元素框的宽度等于容纳块的宽度

7.2.4 横向格式化属性

横向格式化属性有七个，分别为 margin-left、border-left、padding-left、width、padding-right、border-right 和 margin-right。这七个属性影响块级框的横向布局，如图 7-9 所示。

这七个属性的值加在一起要等于元素容纳块的宽度，而这一宽度通常为块级元素的父元素的 width 值（因为块级元素的父元素几乎都是块级元素）。

在这七个属性中，只有三个属性的值能设为 auto：元素内容区的宽度、左外边距和右外边距。余下的几个属性，要么设为具体的值，也么使用默认值（零）。图 7-10 展示了元素框的哪些部分能设为 auto，而哪些部分不能。

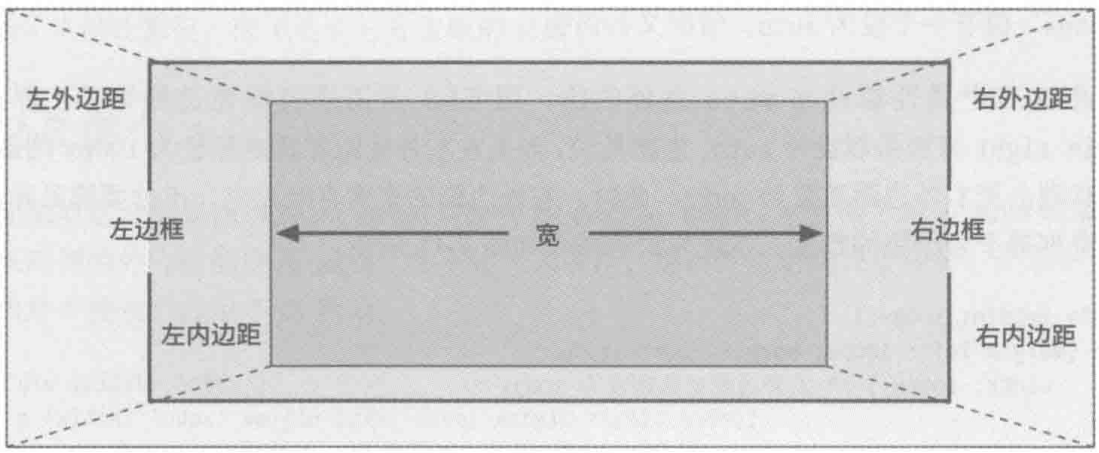


图 7-9：横向格式化的七个属性

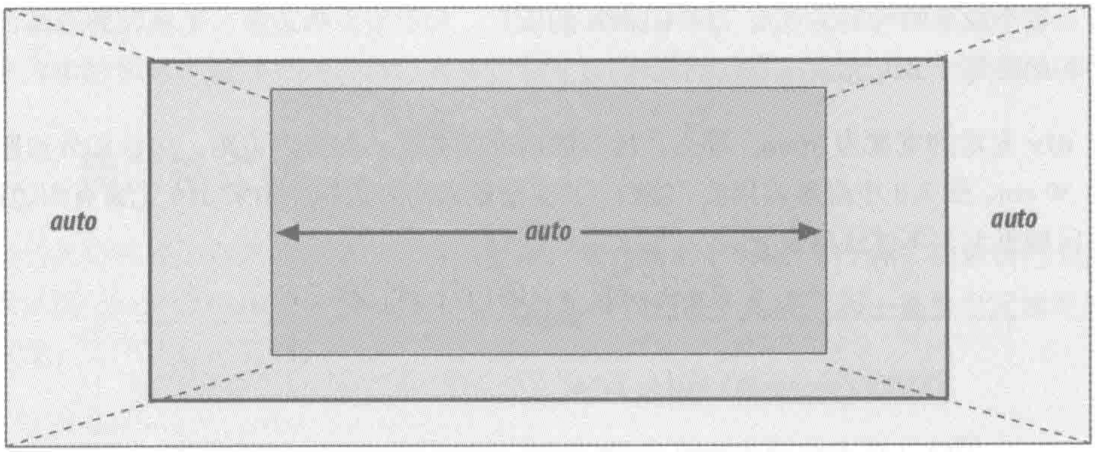


图 7-10：可以设为 auto 的横向格式化属性

width 属性的值要么设为 auto，要么设为某种类型的非负值。在横向格式化中使用 auto，可能得到不同的结果。

7.2.5 使用 auto

在 width、margin-left 和 margin-right 三个属性中，如果把其中一个设为 auto，另两个设为具体的值，那么设为 auto 的那个属性的具体长度要能满足元素框的宽度等于父元素的宽度。假如七个属性的值之和必须等于 500 像素，右外边距的宽度设为 100px，左外边距设为 auto，而且没有设定内边距或边框，那么左外边距的宽度将是 300 像素。

```
div {width: 500px;}
p {margin-left: auto; margin-right: 100px;
    width: 100px;} /* 设为 auto 的左外边距最终计算的结果为 300px */
```

从某种意义上说，auto 可用于补全总和所缺的尺寸。然而，如果把这三个属性的值都设为 100px，没有一个设为 auto，情况又如何呢？

如果把这三个属性都设为 auto 之外的值，用 CSS 术语来说就是过约束了，那么 margin-right 将被强制设为 auto。也就是说，如果左右外边距和宽度都设为 100px 的话，用户代理会把右外边距重置为 auto。此时，右外边距的宽度自动设定，不过要满足元素的总宽度等于容纳块的宽度。下述样式的结果如图 7-11 所示：

```
div {width: 500px;}
p {margin-left: 100px; margin-right: 100px;
    width: 100px;} /* 右外边距被强制设为 300px */
```

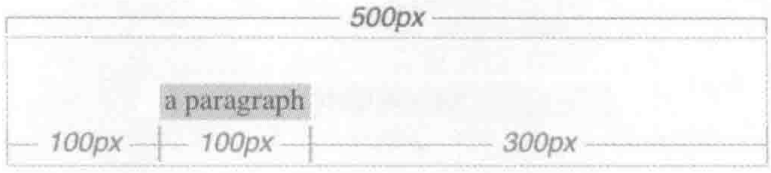


图 7-11：覆盖设定的 margin-right 值

如果左右外边距都明确设为 auto，那么 width 的值是满足总宽度（即父元素的内容区宽度）所需的任何值。下述样式的结果如图 7-12 所示：

```
p {margin-left: 100px; margin-right: 100px; width: auto;}
```

图 7-12 所示的情况是最常见的，即设定左右外边距而不设定 width。下述样式的结果与图 7-12 完全一样：

```
p {margin-left: 100px; margin-right: 100px;} /* 与前面一样 */
```

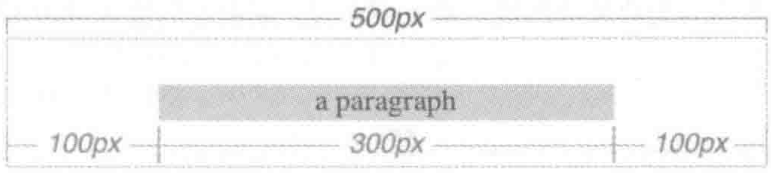


图 7-12：自动确定宽度

你可能想知道把 box-sizing 设为其他值（例如 padding-box）会怎样。前面所讲的内容假定用的是 content-box，不过换成其他值后依然成立；正是因为这样，本节才只讨论 width 和两侧的外边距，而没有涉及内边距或边框。不管把 box-sizing 设为什么值，本节及下一节讨论的 width: auto 处理方式不变。box-sizing 定义的框体包含哪些区域，设为不同的值时有所不同，但是对 auto 值的处理方式不变，因为 box-sizing 决定的是 width 从何处算起，而不是它与外边距的关系。

7.2.6 多个 auto

下面来看这三个属性（width、margin-left 和 margin-right）中有两个设为 auto 的情况。如果两侧的外边距都设为 auto，如下述代码所示，那么外边距的长度相等，元素在父元素内居中显示，如图 7-13 所示。

```
div {width: 500px;}
p {width: 300px; margin-left: auto; margin-right: auto;}
    /* 两侧的外边距都是 100 像素宽，因为 (500-300)/2 = 100 */
```

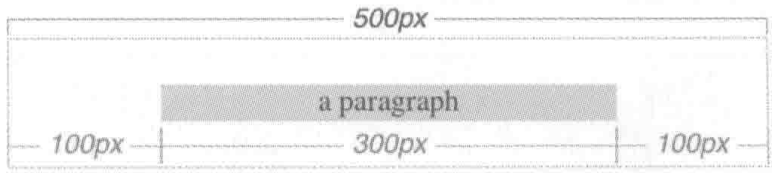


图 7-13：width 设为具体的值

在常规流动方式下，若想让元素在块级框中居中显示，正确的方式是把两侧的外边距设为同样的宽度（弹性盒和栅格布局中还有其他方法，不过这不在本节的讨论范围之内）。

此外，还可以把某一边的外边距和 width 设为 auto。此时，设为 auto 的那个外边距等于零。

```
div {width: 500px;}
p {margin-left: auto; margin-right: 100px;
    width: auto;} /* 左外边距的计算结果为 0，width 则等于 400px */
```

width 则被设为填满容纳块所需的值。对上例来说，width 的值为 400 像素，如图 7-14 所示。

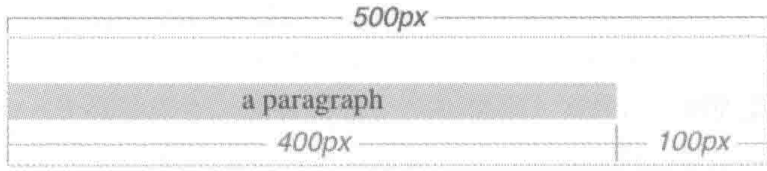


图 7-14：宽度和左外边距都设为 auto 的情况

最后，如果这三个属性都设为 auto 呢？答案是，两侧的外边距被设为零，而 width 则要多宽有多宽。这跟默认情况是一样的，即两侧的外边距和宽度都不明确声明值；此时，外边距默认为零，而 width 默认为 auto。

注意，因为横向外边距不折叠，所以父元素的内边距、边框和外边距可能会影响子代。这影响不太直接，例如元素的外边距（等）可能导致子元素有偏移。下述样式的结果如图 7-15 所示。

```
div {padding: 50px; background: silver;}
p {margin: 30px; padding: 0; background: white;}
```

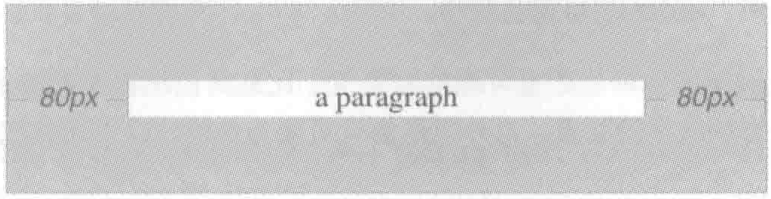


图 7-15：偏移隐藏在父元素的外边距和内边距之中

7.2.7 负外边距

目前来看，一切都相当简单，但是为何我之前说有些地方是复杂的呢？这是因为外边距还有一种取值，即负值。是的，外边距可以设为负值。设为负值能实现一些有趣的效果。

还记得吗，七个横向属性之和始终等于父元素的 `width`。只要这几个属性的值都大于或等于零，元素就不可能比父元素的内容区宽。然而，下述样式便出现意外了，如图 7-16 所示：

```
div {width: 500px; border: 3px solid black;}
p.wide {margin-left: 10px; width: auto; margin-right: -50px; }
```

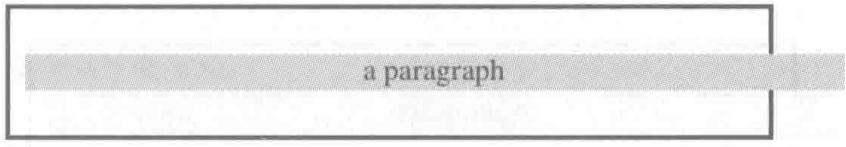


图 7-16：把外边距设为负值，子元素比父元素宽了

你没看错，子元素比父元素宽了！用算式计算，结果是对的：

$$10px + 0 + 0 + 540px + 0 + 0 - 50px = 500px$$

其中，`540px` 对应于 `width: auto`，即满足算式左右两边所需的值。尽管子元素超出了父元素，但是这里并没有违背规范，因为七个属性的值加在一起等于总宽度。这是语义上的伎俩，但却是正确的行为。

下面，加上边框：

```
div {width: 500px; border: 3px solid black;}
p.wide {margin-left: 10px; width: auto; margin-right: -50px;
    border: 3px solid gray;}
```

此时，变化发生在 `width` 的计算结果上：

$$10px + 3px + 0 + 534px + 0 + 3px - 50px = 500px$$

如果再加上内边距，width 的值还会变小。

反过来，设为 auto 的右外边距也可能得到负值。比如其他属性设为特定的值时，为了满足元素不能比容纳块宽的要求，右外边距就有可能为负值。以下述样式为例：

```
div {width: 500px; border: 3px solid black;}
p.wide {margin-left: 10px; width: 600px; margin-right: auto;
    border: 3px solid gray;}
```

此时，算式要这样列：

$$10px + 3px + 0 + 600px + 0 + 3px - 116px = 500px$$

右外边距的计算结果为 -116px。即便明确声明为其他具体的值，右外边距也会被强制设为 -116px，因为规则就是这样制定的：倘若元素的尺寸出现过约束，右外边距要被重置为满足算式所需的任何值（如果是从右至左书写的语言，重置的将是左外边距）。

再看一个例子，结果如图 7-17 所示，这里把左外边距设为负值：

```
div {width: 500px; border: 3px solid black;}
p.wide {margin-left: -50px; width: auto; margin-right: 10px;
    border: 3px solid gray;}
```

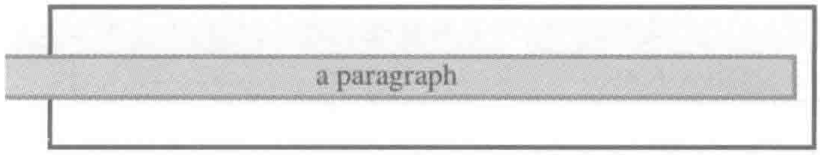


图 7-17：把左外边距设为负值

左外边距为负值时，段落不仅从 div 元素的边框溢出了，还从浏览器窗口的左边溢出了。

注意，内边距、边框和内容宽度（及高度）不能为负值。只有外边距的值可以小于零。

7.2.8 百分数

宽度、内边距和外边距设为百分数时，基本的规则依然适用。其实，这些属性的值设为长度值还是百分数并没什么关系。

百分数有时很有用。假设我们想把元素的内容区宽度设为容纳块的三分之二，左右内边距各为 5%，左外边距为 5%，余下的空间都留给右外边距。此时，可以这样声明：

```
<p style="width: 67%; padding-right: 5%; padding-left: 5%; margin-right: auto;
    margin-left: 5%;">playing percentages</p>
```

这里的右外边距将是容纳块宽度的 18%（100% - 67% - 5% - 5% - 5%）。

然而，百分数和长度单位混在一起使用就不那么容易理清了。以下述标记为例：

```
<p style="width: 67%; padding-right: 2em; padding-left: 2em; margin-right: auto;
    margin-left: 5em;">mixed lengths</p>
```

此时，元素框可以这样定义：

5em + 0 + 2 em + 67% + 2 em + 0 + auto = 容纳块的宽度

为了确保右外边距的计算结果为零，元素的容纳块必须是 27.272727 em 宽（元素的内容区是 18.272727 em 宽）。假使容纳块比这宽，右外边距的计算结果会变成正值；容纳块比这窄的话，右外边距的计算结果则为负值。

如果混用不同的长度单位，情况更复杂。例如：

```
<p style="width: 67%; padding-right: 15px; padding-left: 10px;
    margin-right: auto;
    margin-left: 5em;">more mixed lengths</p>
```

为了避免情况一直复杂下去，边框不接受百分数，只能设为长度值。这样限制的基本原因是，只使用百分数其实无法创建完全弹性的元素，除非不添加边框，或者使用某种实验性的方案，例如弹性盒布局。

7.2.9 置换元素

目前，我们所讲的都是常规流动模式下非置换块级框的横向格式化。块级置换元素的处理方式要简单些，前面针对非置换块级框的规则都成立，不过有个例外：width 为 auto 时，置换元素的 width 等于内容自身的宽度。如果图像自身的宽度为 20 像素，下述示例中的图像将为 20 像素宽：

```
<img src="smile.svg" style="display: block; width: auto; margin: 0;">
```

如果图像自身的宽度为 100 像素，那么它就占 100 像素宽。

明确为 width 提供一个值可以覆盖这个规则。比如说，修改前例，同一个图像显示三次，每次设定的宽度都不同：

```
<img src="smile.svg" style="display: block; width: 25px; margin: 0;">
<img src="smile.svg" style="display: block; width: 50px; margin: 0;">
<img src="smile.svg" style="display: block; width: 100px; margin: 0;">
```

这段标记的结果如图 7-18 所示。

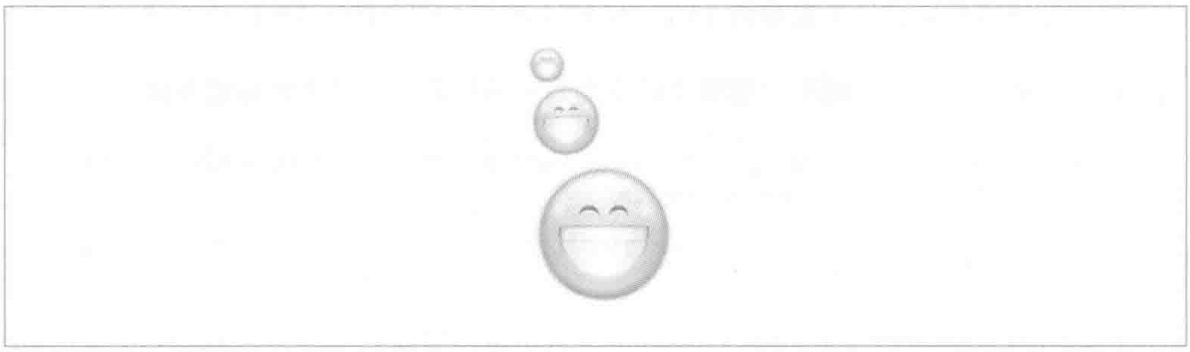


图 7-18：改变置换元素的宽度

注意，元素的高度随之增加了。如果置换元素的 width 与自身宽度不同，height 的值会按比例变化，除非明确设定 height 的值。反过来亦然，如果设定了 height，而把 width 设为 auto，那么宽度会随着高度按比例变化。

既然提到高度了，下面就来讨论常规流动模式下块级框的纵向格式化。

7.2.10 纵向格式化

块级框的纵向格式化与横向格式化类似，也有一些有趣的行为。元素的内容决定元素的默认高度。内容的宽度对高度也有影响，例如，段落越窄，为了包含内部的所有内容，高度就越高。

通过 CSS 可以为任何块级元素设定具体的高度。这么做得到的结果取决于多个因素。假设指定的高度大于显示内容所需的高度：

```
<p style="height: 10em;">
```

此时，多出的高度看起来像是内边距。再假设指定的高度小于显示内容所需的高度：

```
<p style="height: 3.33em;">
```

那么，浏览器要提供查看全部内容的方式，而且前提是不增加元素框的高度。如果元素的内容比框体高，用户代理的具体行为取决于 overflow 属性。这种情况下的两种显示方式如图 7-19 所示。

在 CSS1 中，对置换元素来说（例如图像），用户代理可以忽略 auto 之外的任何高度值。在 CSS2 及以上版本中，height 的值不再允许忽略，唯一的例外是涉及百分数的特殊情况。这一点稍后讨论。

与 width 一样，height 默认定义内容区的高度，而不是元素框可见区域的高度。元素

框的上下内边距、边框和外边距在高度的基础上增加，除非 box-sizing 属性的值不是 content-box。

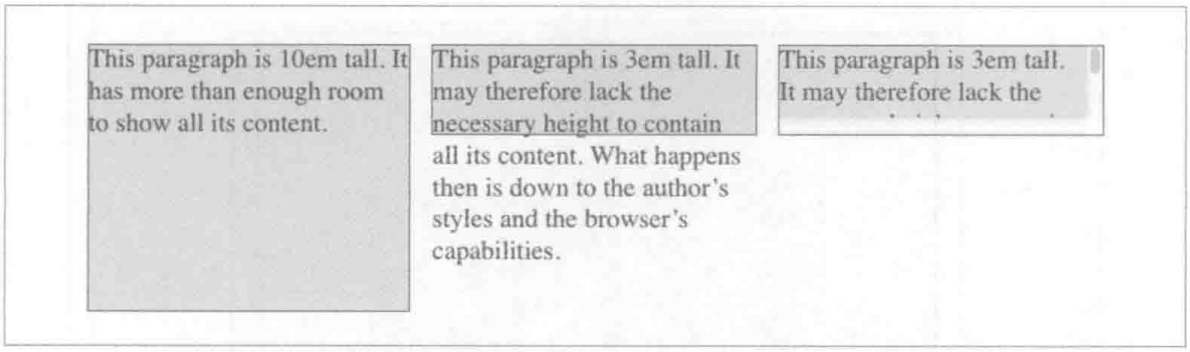


图 7-19：高度与元素中内容的高度不匹配

7.2.11 纵向格式化属性

与横向格式化一样，纵向格式化也涉及七个属性：margin-top、border-top、padding-top、height、padding-bottom、border-bottom 和 margin-bottom。这些属性作用的区域见图 7-20。

这七个属性的值加在一起必须等于块级框的容纳块的高度。通常，这是块级框父元素的 height 值（因为块级元素的父元素几乎都是块级元素）。

这七个属性中只有三个可以设为 auto：元素的高度和上下外边距。上下内边距和边框必须设为具体的值，否则取默认值零（假设未声明边框式样）。如果设定了边框式样（border-style），那么边框的宽度默认为不具体的 medium。图 7-21 展示了元素框的哪些部分可以设为 auto，而哪些部分不能。

奇怪的是，在常规流动模式下，如果把块级框的 margin-top 或 margin-bottom 设为 auto，二者都自动计算为 0。如此看来，常规流动模式下的元素无法轻易在容纳块中纵向居中。这也意味着，如果把元素的上下外边距设为 auto，最终会被重置为 0，不会体现在元素框上。



定位元素和弹性盒元素对设为 auto 的上下外边距的处理方式有所不同。

height 属性要么设为 auto，要么设为某种类型的非负值，决不能小于零。

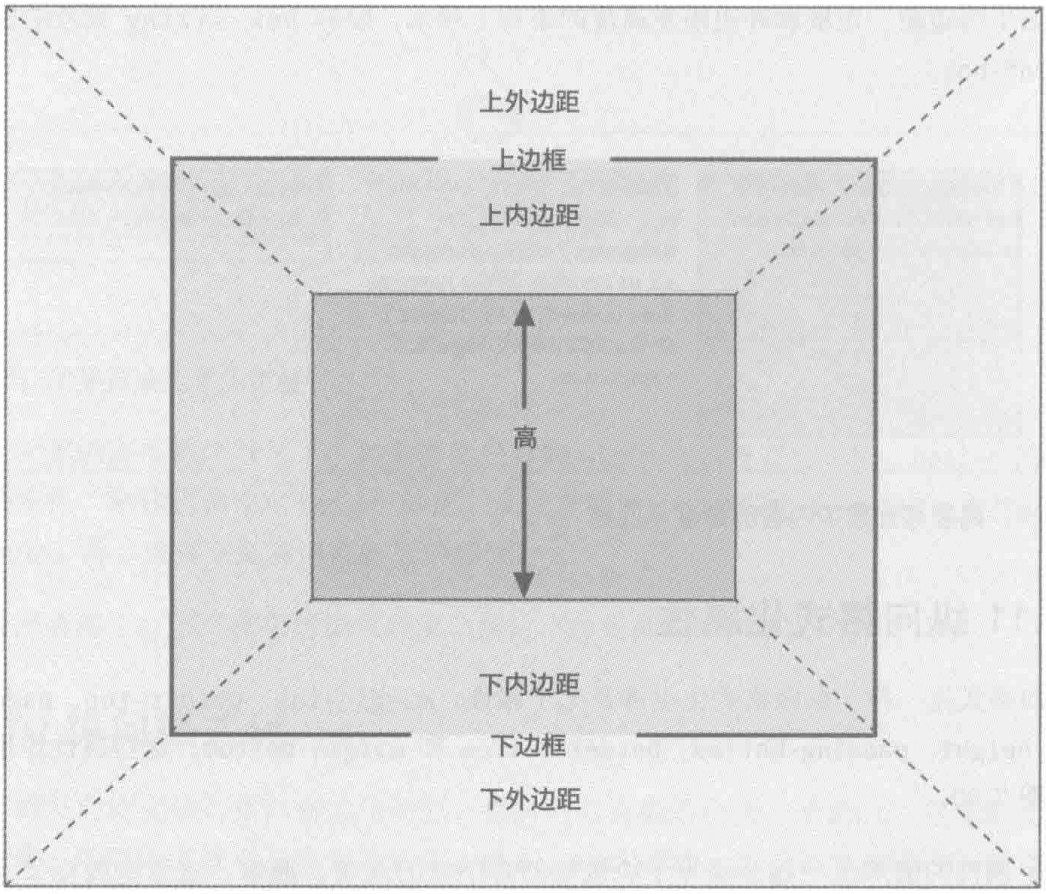


图 7-20：七个纵向格式化属性

7.2.12 百分数高度

我们已经知道如何处理值为具体长度的高度，下面花点时间讨论百分数高度。在常规流动模式下，如果把块级框的高度设为百分数，百分数是相对框体的容纳块的高度而言的。对下述标记来说，段落的高度将是 3 em：

```
<div style="height: 6em;">
    <p style="height: 50%;">Half as tall</p>
</div>
```

既然把上下外边距设为 auto 得到的外边距为零，那么针对这个例子，若想纵向居中元素，只能把上下外边距都设为 25%。不过，这样只能把框体居中，里面的内容不居中。

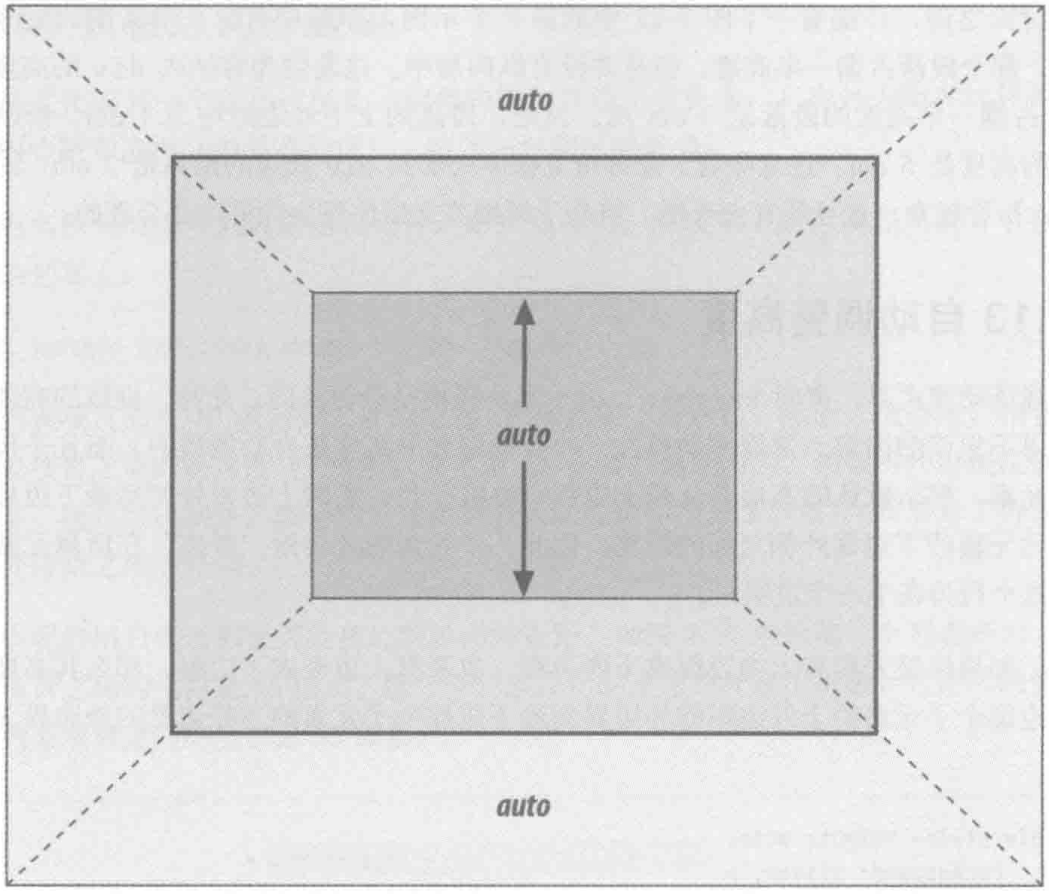


图 7-21：可以设为 auto 的纵向格式化属性

然而，如果未明确声明容纳块的高度，那么百分数高度将被重置为 auto。在上例中，如果把 div 的 height 改为 auto，那么段落的高度将与 div 一样：

```
<div style="height: auto;">
    <p style="height: 50%;">NOT half as tall; height reset to auto</p>
</div>
```

这两种情况如图 7-22 所示（段落边框和 div 边框之间的空白是段落的上下外边距）。

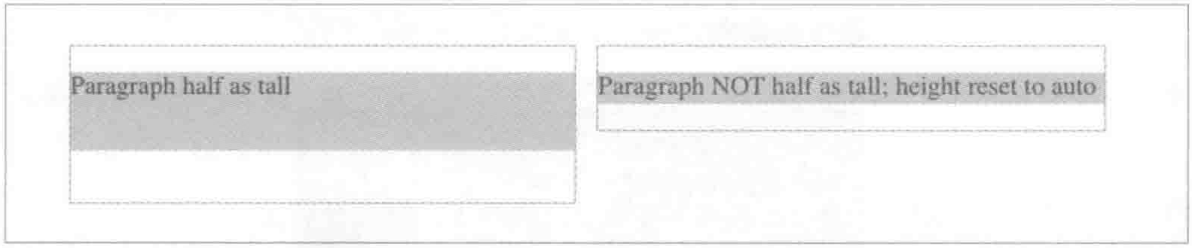


图 7-22：不同情况下的百分数高度

继续讨论之前，仔细看一下图 7-22 中的第一个示例，即那个占据一半高度的段落。看起来，那个段落占据一半高度，但是并没有纵向居中。这是因为容纳块 div 的高度为 6 em，占据一半高度的段落是 3 em 高。但是，段落的上下外边距均为 1 em，所以整个框体的高度是 5 em。这意味着，段落可见框体底部到 div 底边的距离是 2 em，而不是 1em。初看起来，这可能有点奇怪，但是了解细节之后你便会知道这是合理的。

7.2.13 自动调整高度

在常规流动模式下，声明 height: auto 的块级框是最简单的，此时，框体的高度恰好能放得下里面的内容。常规流动模式下的块级框如果高度是自动调整的，而且子代都是块级元素，那么默认的高度是从最上边那个块级子代元素的上边框外侧到最下边那个块级子代元素的下边框外侧之间的距离。因此，子元素的外边距“游离”在所属元素的外部（这个行为在下一节说明）。

然而，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，那么其高度是从最上边那个子元素的上外边距的外边界到最下边那个子元素的下外边距的外边界之间的距离。

```
<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>
```

这两种情况如图 7-23 所示。

如果把前例中的边框改为内边距，div 元素的高度不变，外边距依然算在其中。

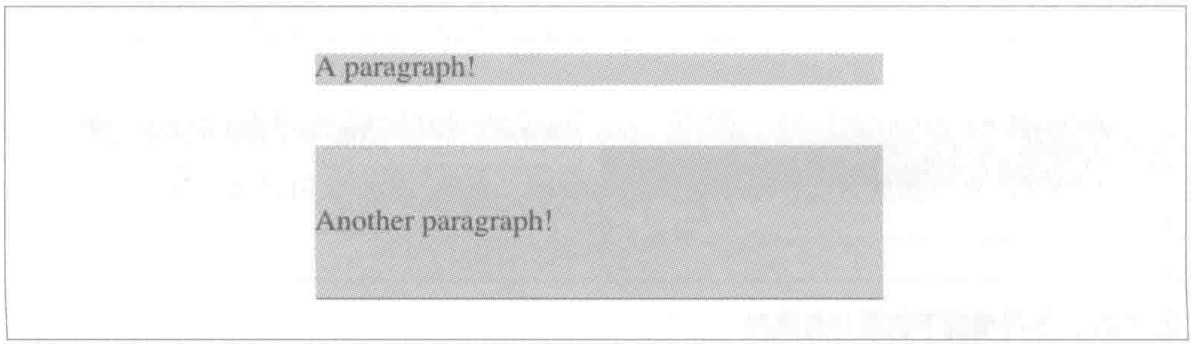


图 7-23：高度自动调整的元素中的块级子元素

7.2.14 折叠纵向外边距

纵向格式化的另一个重要特征是，相邻的纵向外边距会折叠。只有外边距有这种折叠行为。内边距和边框（如果有的话），绝不与任何区域折叠。

通过纵向罗列的无序列表最易说明外边距折叠行为。假设把下述声明应用到一个有五个项目的列表上：

```
li {margin-top: 10px; margin-bottom: 15px;}
```

每个列表项目都有 10 像素的上外边距和 15 像素的下外边距。然而，渲染时，相邻的两个列表项目之间的距离是 15 像素，而不是 25 像素。这是因为，相邻的外边距在纵轴上折叠了。换句话说就是，较小的外边距被较大的外边距消去了。图 7-24 展示了折叠和未折叠外边距之间的差别。

目前实现的用户代理都会折叠相邻的纵向外边距，如图 7-24 中的第一个列表所示，因此列表项目之间的空白是 15 像素。第二个列表展示的是用户代理不折叠外边距时的情况，此时列表项目之间的空白是 25 像素。

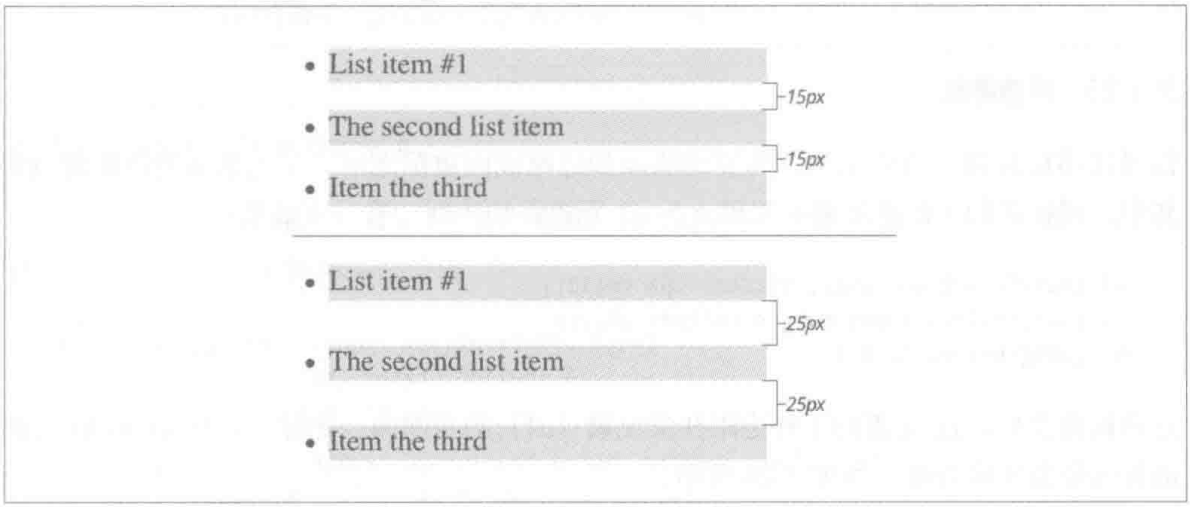


图 7-24：折叠和未折叠的外边距

如果你不喜欢“折叠”这个词，可以换成“重叠”。虽然外边距并不真的会重叠，但是你可以借助下述比拟想象一下这其中到底发生了什么。

假设每个元素（例如段落）是一小片纸，元素中的内容写在纸上。纸片四周有一些透明塑料，表示外边距。先把第一片纸（假如表示 h1 元素）放在画布上，然后在下面再放一片纸（一个段落），向上移动，直到其中一片纸的边缘与另一片纸的边缘接触为止。如果第一片纸的下部有半英寸塑料，第二片纸的上部有三分之一英寸塑料，那么两片纸

移到一起时，第一片纸的塑料将碰到第二片纸的上边缘。两片纸平稳放到画布上之后，纸四周的塑料就重叠了。

多个外边距同时出现时也会折叠，例如在列表的末尾。假设把下述规则应用到前例上：

```
ul {margin-bottom: 15px;}
li {margin-top: 10px; margin-bottom: 20px;}
h1 {margin-top: 28px;}
```

列表中最后一个项目的下外边距为 20 像素，ul 元素的下外边距为 15 像素，而后面的 h1 元素的上外边距为 28 像素。因此，折叠外边距之后，最后一个 li 元素的底部与 h1 元素的顶部之间的距离为 28 像素，如图 7-25 所示。

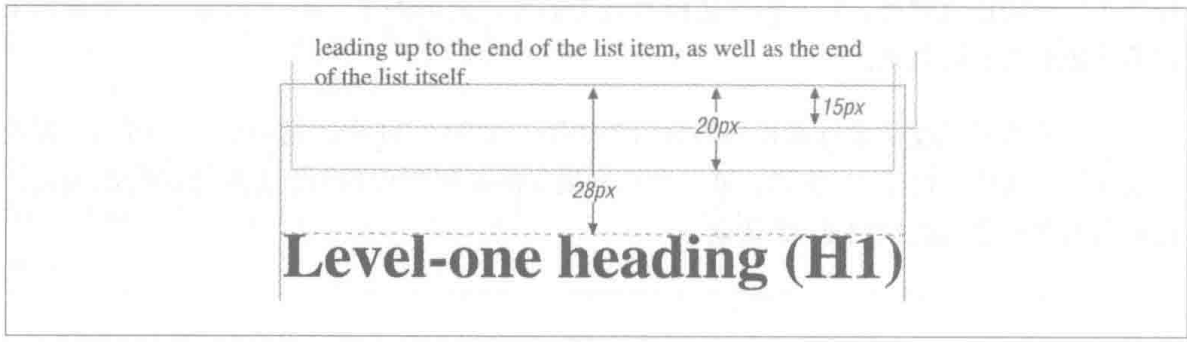


图 7-25：折叠图解

你可能还记得前一节的示例，为容纳块添加边框或内边距之后，子元素的外边距便包含其中。现在我们在前述规则的基础上为 ul 元素添加边框，看一下效果：

```
ul {margin-bottom: 15px; border: 1px solid;}
li {margin-top: 10px; margin-bottom: 20px;}
h1 {margin-top: 28px;}
```

这样修改之后，li 元素的下外边距在父元素（ul）的范围内。此时，只有 ul 和 h1 之间的外边距会出现折叠，如图 7-26 所示。

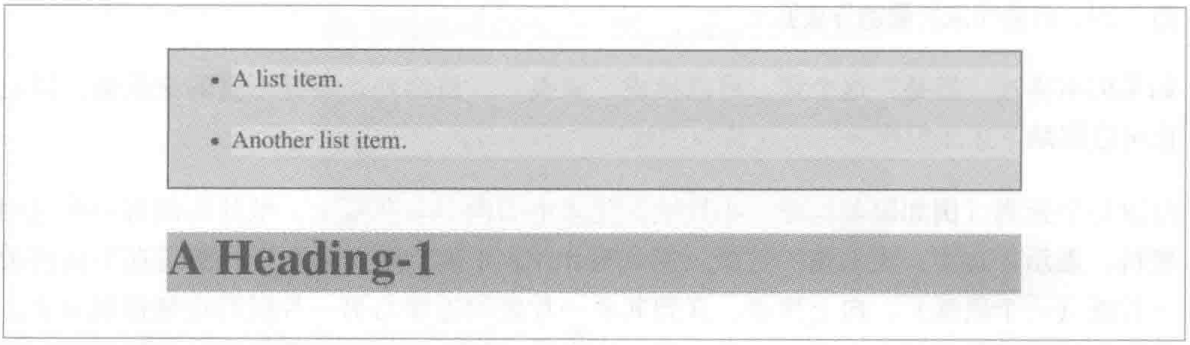


图 7-26：添加边框后有的外边距折叠，有的不折叠

7.2.15 负外边距和折叠

负外边距对纵向格式化有一定影响，比如会影响外边距的折叠。如果两个相邻的外边距都是负值，浏览器取其中绝对值较大的那个，然后从正外边距中减去它的绝对值。也就是说，把负值与正值相加，得到的结果为两个元素之间的距离。图 7-27 给出了两个具体的例子。

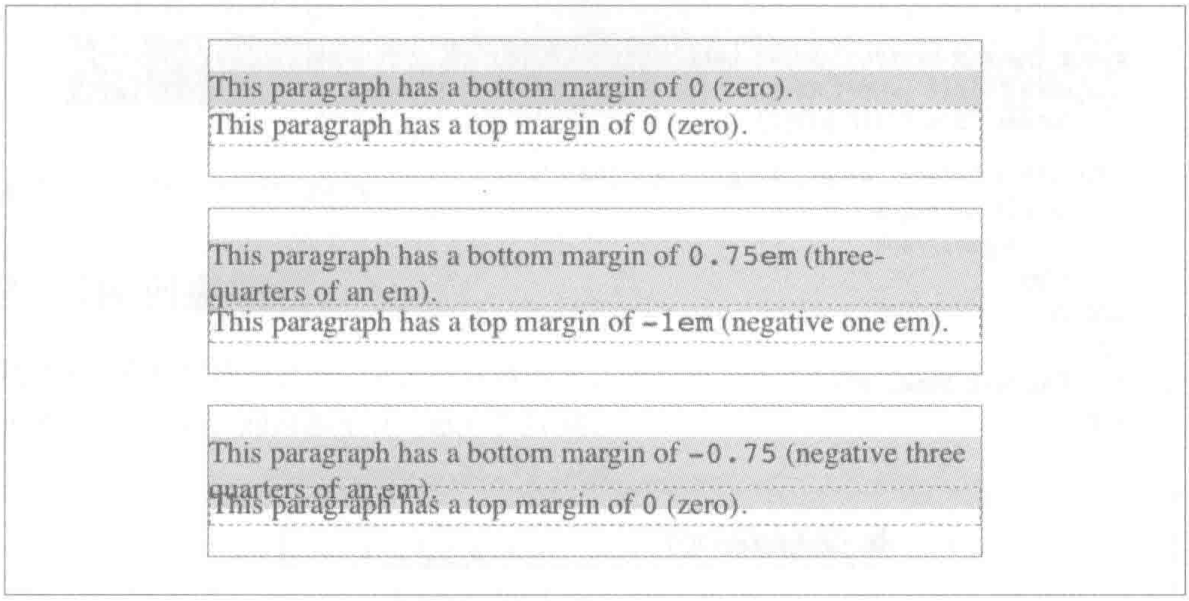


图 7-27：纵向外边距为负的例子

注意上下外边距为负时出现的“上移”现象。这与横向外边距为负时从父元素中溢出没什么区别。以下述规则和标记为例：

```
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>
```

从图 7-28 中可以看出，由于上外边距为负，那个段落上移了。注意，div 元素中那个段落后面的内容也向上移了 50 像素。其实，那个段落后面常规流动模式下的每个元素都会向上移 50 像素。

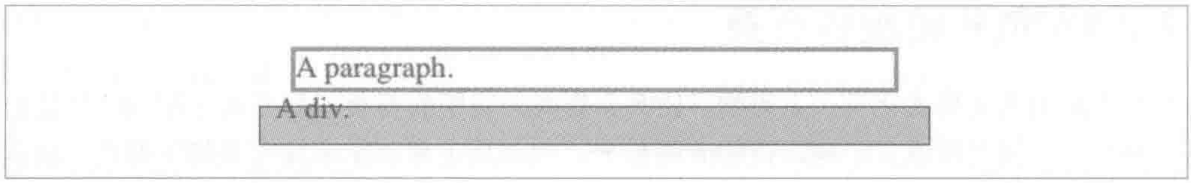


图 7-28：上外边距为负时的效果

对比一下下述标记与图 7-29 中显示的效果：

```
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>
```

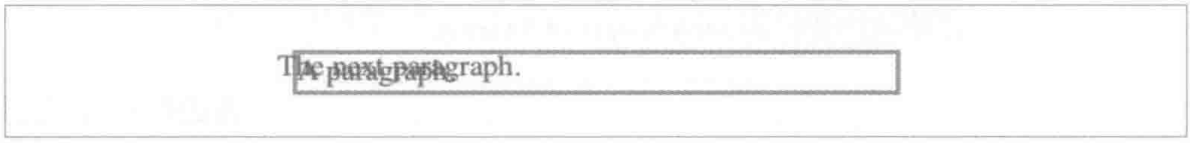


图 7-29：下外边距为负时的效果

出现图 7-29 中这种情况的原因是，div 后面的元素是参照 div 的底边放置的。可以想见，div 的底边其实在内部那个段落视觉上的底边上部，而 div 后面的元素是从这一位置计算与 div 底边的距离的。对上述规则来说，这个结果是正确的。

下面再看一个例子，这里列表项目、无序列表和段落的外边距都会折叠。无序列表和段落的外边距均为负值，如下所示：

```
li {margin-bottom: 20px;}
ul {margin-bottom: -15px;}
h1 {margin-top: -18px;}
```

两个负外边距中绝对值较大的那个（-18px）与最大的正外边距（20px）相加，得到 20px - 18px = 2px。因此，列表项目内容区的底边到 h1 元素内容区的顶边之间只有 2 像素，如图 7-30 所示。

如果负外边距导致元素重叠，很难分清哪个元素在上边。你可能注意到了，本节所举的例子都没有使用背景色。如果这么做了，后续元素的背景色可能会遮盖前面元素的内容。

这是可以预见的，因为浏览器是从头到尾按顺序渲染元素的，在常规流动模式下，如果文档中位于后面的元素与前面的元素重叠了，理应遮盖元素的内容。

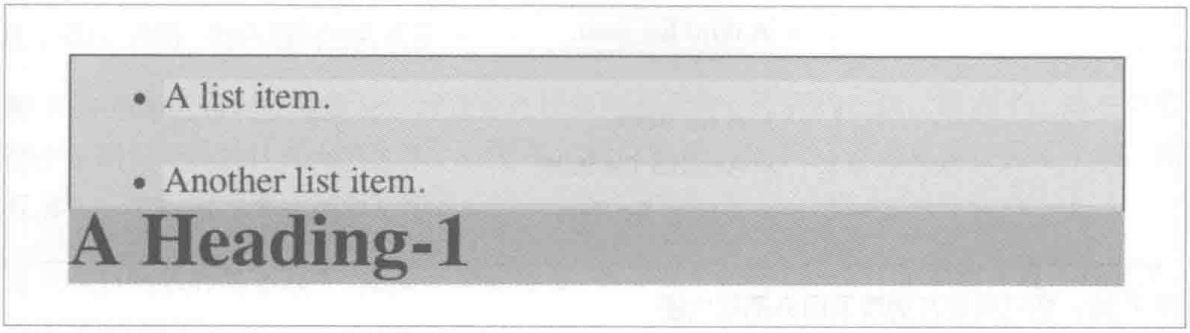


图 7-30：正负外边距折叠图解

7.2.16 列表项目

列表项目有些独特的规则。列表项目前通常有个记号，例如小圆点或数字，但这个记号其实不在列表项目的内容区中，如图 7-31 所示。

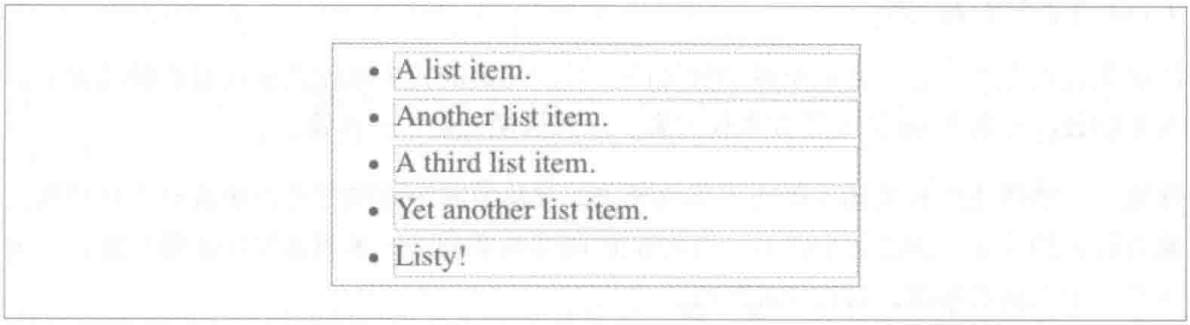


图 7-31：列表项目的内容区

在文档的布局方便，CSS1 没有详细规定记号的位置和效果。鉴于此，CSS2 引入了专门解决这个问题的属性，例如 `marker-offset`。然而，由于没有多少浏览器实现，以及想法的变化，CSS2.1 又移除了这个属性；不过，已经有了新的想法（可能还没确定具体的句法）。因此，记号的位置远非创作人员能控制的，至少目前是这样。

记号可以放在列表项目内容区的外部，也可以作为行间内容，放在内容区的开头，这取决于 `list-style-position` 属性的值。如果把记号放在内部，列表项目与周围元素之间的关系就跟块级元素一模一样，如图 7-32 所示。

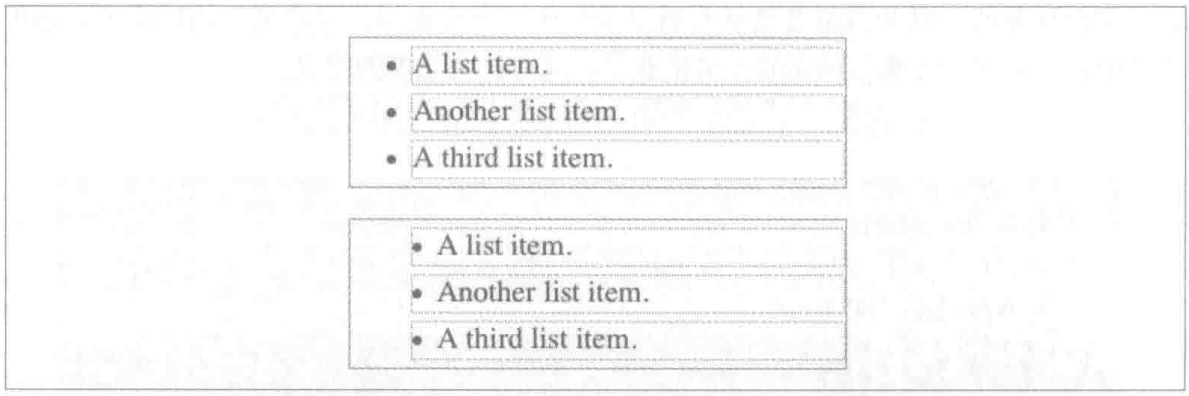


图 7-32：把记号放在列表项目外部和内部

如果把记号放在内容区外部，记号与内容区的左边（假设是由左至右书写的语言）有一定的距离。不管如何调整列表的样式，记号与内容区边界之间的距离始终不变。有时，记号甚至可能超出列表元素本身，如图 7-32 所示。

注意，与常规的块级框一样，列表项目框为其后代定义容纳块。

7.3 行内元素

除块级元素之外，行内元素是最为常见的。行内元素的相关属性能实现更有趣的效果。常见的行内元素有 em 和 a 等非置换元素，以及置换元素——图像。

注意，本节所述的行为均不适用于表格元素。表格及其内容的行为与块级元素和行内元素有极大的差异，因此 CSS2 引入了专门的新属性和行为。本书不讨论表格装饰，因为这是一个全新的领域，而且异常繁琐。

行内的非置换元素和置换元素有些许差异，在讲解行内元素构成的过程中将逐一说明。

7.3.1 行布局

首先要明白行内内容是如何布局的。块级元素相对简单，生成的块级框通常不与其他内容共处一行。行内元素就没这么简单了，不信的话，你可以看一下块级元素（例如一个段落）的内部。你可能会有这样的疑问：这些文本行是如何显示出来的呢？为什么要这样排列呢？可以采用何种方式控制呢？

为了弄清文本行是如何生成的，下面以一个文本行特别长的元素为例，如图 7-33 所示。注意，我们把整行文本放在一个 span 元素中，并为其设定了边框：

```
span {border: 1px dashed black;}
```



图 7-33：只有一行内容的行内元素

图 7-33 所示的情况非常简单，就是放在块级元素中的一个行内元素。说白了，这与只有两个单词的段落没什么大的区别。这里唯一的区别是，图 7-33 中的单词数量多了些，而且放在一个行内元素（span）中。

先从我们熟悉的概念入手，定义元素的宽度。有了宽度的限制，这行文本将分成几行显示，如图 7-34 所示。

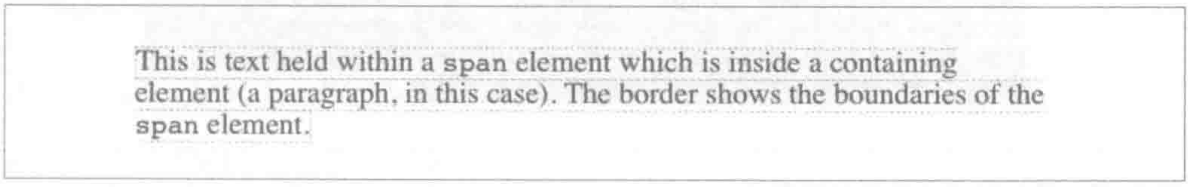


图 7-34：分为多行显示的行内元素

其实我们没做什么，只是把单行文本分成了多行，从上到下排列着显示。

在图 7-34 中，各行的边框恰巧与上下边重合。这种情况仅当行内文本没有内边距才会发生。注意，边框稍微有点重合。例如，第一行的下边框正好位于第二行上边框的下方。这是因为边框在各行外部的下一像素（假设你用的是显示器）处绘制。既然行与行之间是相互接触的，那么边框肯定会像图 7-34 中那样重叠。

如果调整 span 元素的样式，为其设定背景色，那么各行的位置就比较明晰了。如图 7-35 所示，图中有四个段落，每个段落的 text-align 值有所不同，而且段落中的各行都有背景色。

图中的灰色虚线是段落内容区的边界，可以看出，不是每一行都能延伸到段落的边界。对左对齐那一段来说，每一行都靠左边界显示，而行尾在断行处结束。右对齐那段情况正好相反。对居中对齐那段而言，每一行的中线与段落的中线对齐。

最后一段的 text-align 属性值为 justify，因此每一行的宽度与段落的内容区相等，而且都到达了边界处。然而，由于每一行自身的宽度与段落的宽度是不同的，所以要调整字母及单词之间的距离。鉴于此，两端对齐的文本，其 word-spacing 属性的值可能会被覆盖（如果 letter-spacing 属性的值是一个长度，那就无法被覆盖）。

以上就是文本行排列的最简单情况。然而，你即将发现，行内格式化模型比这复杂得多。

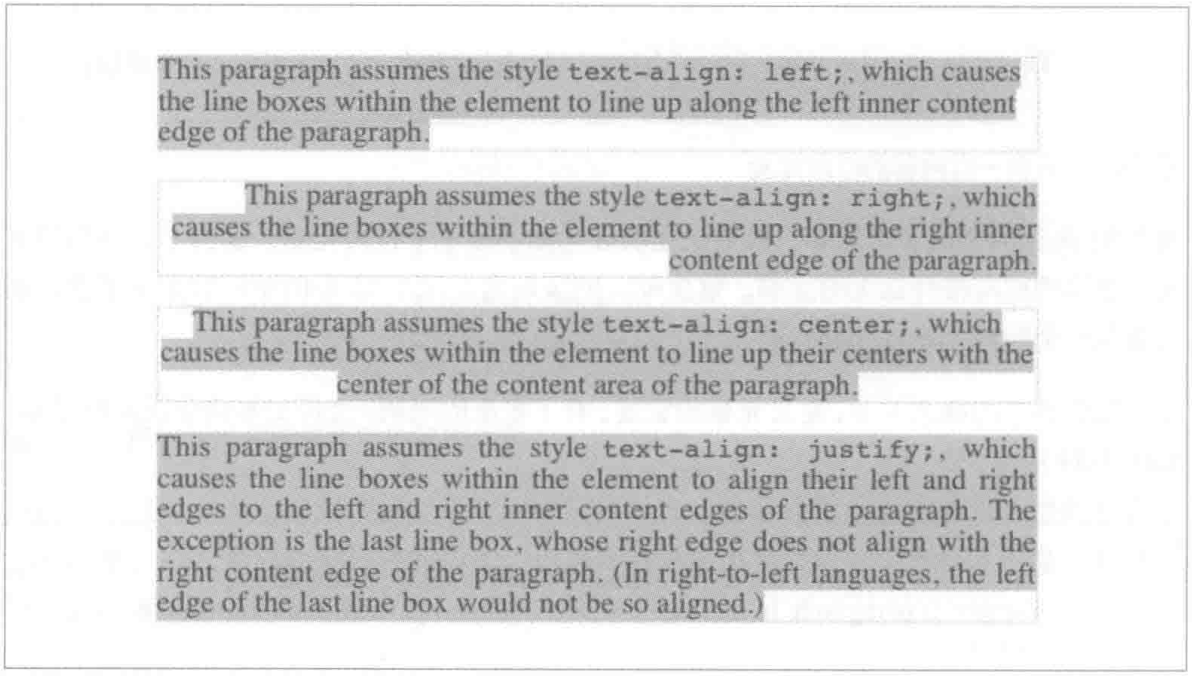


图 7-35：不同对齐方式下的文本行

7.3.2 基本术语和概念

深入讨论之前，先学习有关行内布局的一些基本术语。这对理解后续章节的内容尤为重要。

匿名文本

不在任何行内元素中的字符串。例如，在 `<p> I'm <em>so</em> happy!</p>` 标记中，“I'm”和“happy!”是匿名文本。注意，空格也是匿名文本的一部分，因为空格也是一种字符。

字体框

由字体具体定义，也叫字符框。字形可能比字体框高或矮。CSS 中的 `font-size` 属性控制字体框的高度。

内容区

对非置换元素而言，内容区有两种定义，CSS 规范允许用户代理选择其中一个实现。内容区可以是各字符的字体框连在一起构成的方框，也可以是元素中各字符的字形合在一起构成的方框。简单起见，本书采用前者。对置换元素来说，内容区是元素自身的高度加上内外边距和边框。

行距

行距是 font-size 和 line-height 之差。二者之间的差值除以 2 之后分别添加到内容区的上部和下部。内容区多出来的这部分空间叫半行距（显然的）。只有非置换元素有行距。

行内框

内容区加行距后得到的方框。对非置换元素来说，行内框的高度正好等于 line-height 属性的值。对置换元素而言，行内框的高度等于内容区的高度，因为置换元素没有行距。

行框

过一行中各行内框最高点和最低点的方框。也就是说，行框的顶边过最高那个行内框的顶边，行框的底边过最低那个行内框的底边。

此外，CSS 还定义了一些行为和有用的概念：

- 内容区相当于块级框的内容框。
- 行内元素的背景填充在内容区加内边距所在的区域里。
- 行内元素的边框在内容区外的内边距外侧。
- 非置换行内元素的内边距、边框和外边距在对应的方框上没有纵向效果，即对行内框（及元素所在的行框）的高度没有影响。
- 然而，置换元素的外边距和边框对行内框的高度有影响，进而对元素所在的行框的高度也有影响。

还要注意一点：行内框在一行中纵向对齐的方式由 vertical-align 属性决定。

在继续之前，先来看一下构建行框的具体步骤。这一过程能让你了解一行中的各部分是如何决定行高的。

一行中各元素的行内框高度是这样确定的：

1. 确定行内各非置换元素和匿名文本的 font-size 和 line-height 值，后者减去前者，得到行距。行距除以 2，分别添加到字体框的上部和下部。
2. 确定各置换元素的 height、margin-top、margin-bottom、padding-top、padding-bottom、border-top-width 和 border-bottom-width 值，各值相加。
3. 确定各内容区在一行的基线上方和下方分别超出多少。这不是件简单的事，你要知道各元素和匿名文本的基线在何处，以及一行的基线在何处，然后把它们对齐。另外，要把置换元素的底边与一行的基线对齐。

4. 确定设定了 vertical-align 属性的元素纵向偏移有多少。这是为了查明元素的行内框向上或向下移动了多少，因为纵向对齐改变了元素与基线之间的距离。
5. 知道所有行内框的位置之后，计算行框的高度：基线与最高那个行内框顶边之间的距离加上基线与最低那个行内框底边之间的距离。

下面详细分析整个过程。这是正确装饰行内内容的关键。

7.3.3 行内格式化

首先要知道，不管有没有显式声明，所有元素都有 line-height 值。这个值在很大程度上影响着行内元素的显示，因此要特别注意。

那么，一行的高度是如何确定的呢？一行的高度（即行框的高度）由其内部的元素及其他内容（例如文本）的高度决定。注意，line-height 影响行内元素和其他行内内容，但不影响块级元素，至少没有直接影响。块级元素可以设定 line-height 值，但是只对块级元素中的行内内容有视觉影响。以下面这个没有内容的段落为例：

```
<p style="line-height: 0.25em;"></p>
```

因为这个段落没有内容，没什么可显示的，所以我们看不到任何内容。不管这个段落的 line-height 值是多少，是 0.25em 也好，是 25in 也好，没有任何区别，没有内容就没有行框。

块级元素当然可以有 line-height 值，但是这个值将应用于块级元素内部的内容上（不管在不在行内元素中）。从某种程度上讲，块级元素中的各文本行本身就是行内元素，不论是否真的在行内标签中都是如此。如果愿意，可以想象成有个虚构的标签，如下所示：

```
<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 标签并不存在，但是段落的行为就像有这些标签一样，每行文本都从段落上继承样式。为了避免麻烦，你可以为块级元素设定 line-height 值，这样就无需为每个行内元素（虚构的或真实的）设定 line-height 值了。

通过虚构的 line 标签，我们明确了为块级元素设定 line-height 值的效果。根据 CSS 规范，为块级元素设定 line-height 值的作用是为块级元素中的内容设定行框的最小高

度。例如，p.spacious {line-height: 24pt;} 的作用是把各行框的最小高度设为 24 点。理论上讲，只有行内元素能继承行高时，其中的内容才会继承。但是，多数文本不在行内元素中。如果你假设各行都在虚构的 line 元素中，这就说得通了。

7.3.4 行内非置换元素

在现有的格式化知识基础上，下面我们来讨论只含非置换元素（或匿名文本）的行。了解这一点之后，你便能更好地理解行布局中非置换元素和置换元素之间的区别了。

行框的构成

首先，对非置换元素或匿名文本来说，font-size 值决定内容区的高度。如果行内元素的 font-size 为 15px，那么其内容区的高度就是 15 像素，因为元素中的所有字体框高度都是 15 像素，如图 7-36 所示。

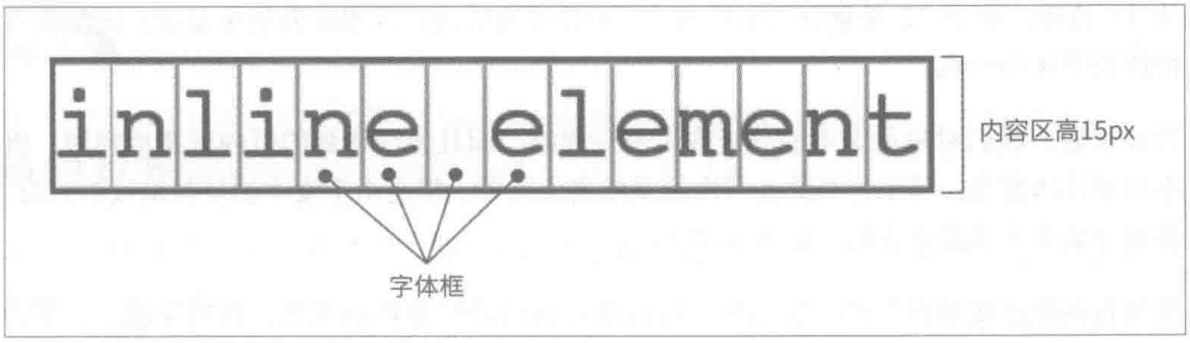


图 7-36：字体框高度决定内容区的高度

接下来要考虑的是元素的 line-height 值，以及它与 font-size 值之差。如果行内非置换元素的 font-size 为 15px、line-height 为 21px，那么二者之差为 6 像素。用户代理把这 6 像素一分为二，一半添加到内容区上部，一半添加到内容区下部，得到行内框。这个过程如图 7-37 所示。

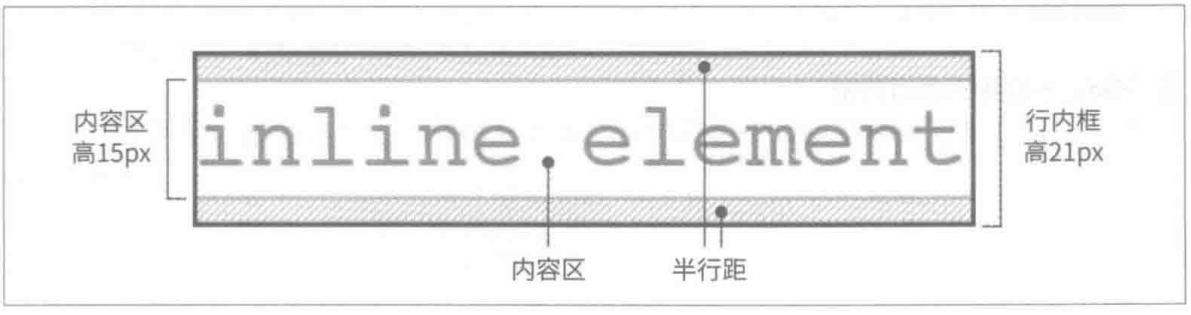


图 7-37：内容区高度加上行距等于行内框高度

假设有下述标记：

```
<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>
```

在这个示例中，多数文本的 font-size 为 12px，但是有个行内非置换元素的字号为 24px。然而，所有文本的 line-height 都是 12px，因为 line-height 是会被继承的属性。因此，strong 元素的 line-height 仍是 12px。

鉴于此，对那些 font-size 和 line-height 都为 12px 的文本来说，内容高度不变（因为 12px 减 12px 等于零），所以行内框的高度为 12 像素。然而，对加粗文本而言，line-height 和 font-size 之差为 -12px。除以 2 之后得到半行距（-6px），然后分别添加到内容区的上部和下部，得到行内框。因为添加的是负值，所以行内框的最终高度为 12 像素。这个 12 像素高的行内框在 24 像素高的内容区里纵向居中显示，因此行内框比内容区小一点。

目前来看，我们对每段文本的分析方式是一样的，而且最终得到的行内框高度相同，但事实要比这复杂。第二行里的各行内框虽然高度相等，但是由于文本都是与基线对齐的，各部分文本并不是平齐的（见图 7-38）。

因为行内框决定着行框的总体高度，所以各行内框的位置特别重要。我们知道，行框的高度是指从最高那个行内框的顶边到最低那个行内框的底边之间的距离，而行框的顶边与前一行的底边是紧挨在一起的。图 7-38 中的文本所在的段落如图 7-39 所示。

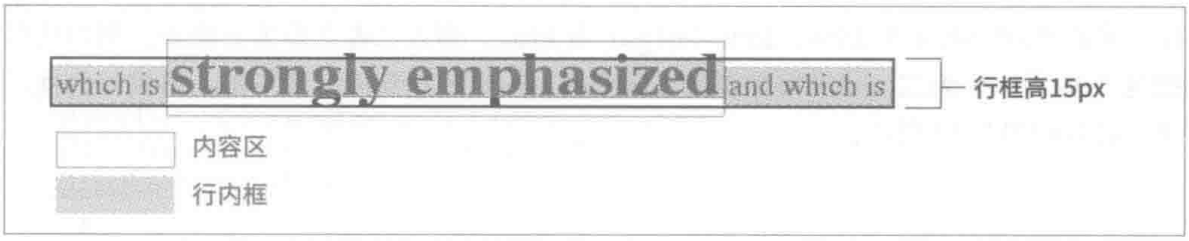


图 7-38：一行中的各行内框

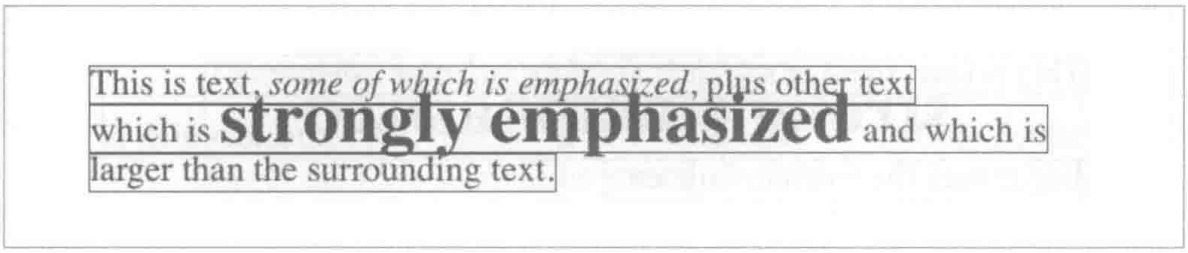


图 7-39：一个段落中的各行框

从图 7-39 可以看出，中间那一行比其他两行要高，但是还不足以包含全部文本。匿名文本所在的行内框决定行框的底边位置，而加粗元素所在的行内框定位行框的顶边。因为那个行内框的顶边在元素的内容区内部，所以加粗元素的内容超出了行框，叠加到其他行框上了。最终的结果是，那部分文本显得不太协调。



稍后将说明应对这种行为的方法，以及实现相同基线间距的方法。

纵向对齐

改变行内框的纵向对齐方式后，仍以相同的过程确定高度。假设我们把 `strong` 元素的纵向对齐设为 `4px`：

```
<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; vertical-align: 4px;">strongly
emphasized</strong> and that is<br>
larger than the surrounding text.
</p>
```

这个小小的改动把 `strong` 元素向上抬升了 4 像素，而且是内容区与行内框一起移动。现在，`strong` 元素所在的行内框的顶边是一行中最高的，因此此次修改纵向对齐还把行框的顶边向上移了 4 像素，如图 7-40 所示。

再来看一种情况。这里，我们在加粗文本那一行里又添加了一个行内元素，而且不是与基线对齐的：

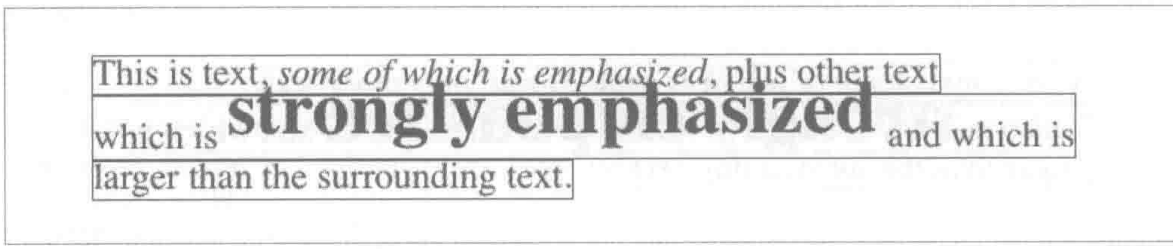


图 7-40：纵向对齐影响行框的高度

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>,<br>
plus other text that is <strong style="font-size: 24px;">strong</strong>
 and <span style="vertical-align: top;">tall</span> and is<br>
larger than the surrounding text.
</p>
```

结果与之前一样，中间那一行的行框比其他行框要高。然而，请注意“tall”的对齐方式，如图 7-41 所示。

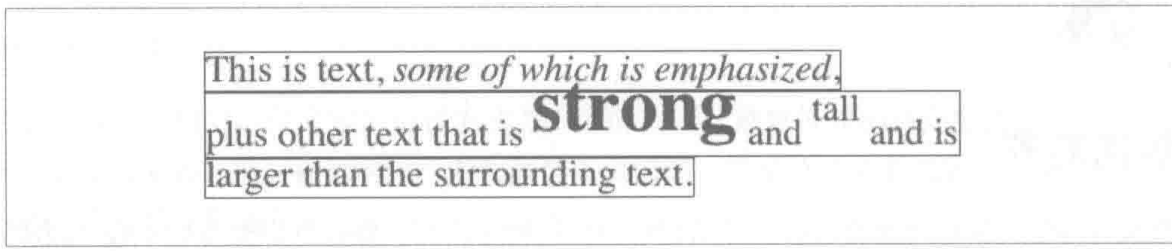


图 7-41：行内元素与行框对齐

这里，“tall”文本所在的行内框的顶边与行框的顶边对齐。因为“tall”文本的 font-size 和 line-height 值相等，所以内容区与行内框的高度相等。然而，再看下面这种情况：

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>,<br>
plus other text that is <strong style="font-size: 24px;">strong</strong>
 and <span style="vertical-align: top; line-height: 2px;">tall</span> and is<br>
larger than the surrounding text.
</p>
```

现在，“tall”文本的 line-height 比 font-size 小，因此其行内框的高度比内容区小。这处细微的变化将改变文本的位置，因为其行内框的顶边要与行框的顶边对齐。此时得到的结果如图 7-42 所示。

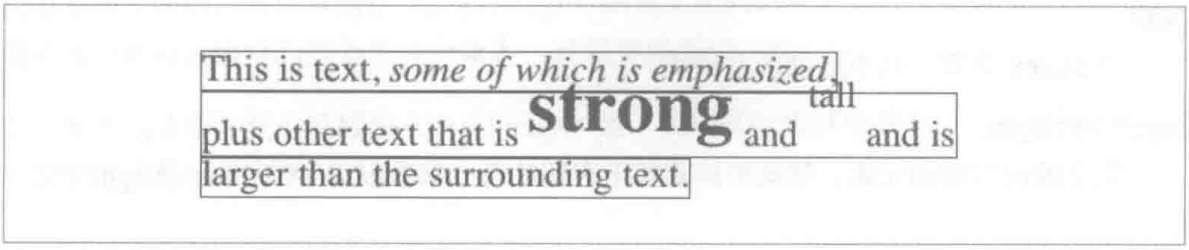


图 7-42：文本（再次）超出行框

此外，还有可能把“tall”文本的 line-height 设定的比 font-size 大。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>
```

我们把“tall”文本的 line-height 设为 18px，因此 line-height 和 font-size 之差为 6 像素。半行距（3 像素）添加到内容区上之后，得到的行内框为 18 像素高。这个行内框的顶边与行框的顶边对齐。类似地，如果 vertical-align 的值为 bottom，那么行内元素的行内框的底边将与行框的底边对齐。

下面使用本章采用的术语描述 vertical-align 各个关键字值的效果：

top
: 元素行内框的顶边与所在行框的顶边对齐。

bottom
: 元素行内框的底边与所在行框的底边对齐。

text-top
: 元素行内框的顶边与父元素内容区的顶边对齐。

text-bottom
: 元素行内框的底边与父元素内容区的底边对齐。

middle
: 元素行内框的纵向中点与父元素基线以上 0.5ex 处的点对齐。

super
: 向上移动元素的内容区和行内框。无法指定移动距离，不同的用户代理之间可能有区别。

sub

与 super 类似，只不过是向下移动。

<percentage>

向上或向下移动元素，移动的距离等于声明的百分数乘以元素的 line-height 值。

控制行高

读过前面几小节我们知道，修改行内元素的 line-height 值可能会导致一行中的文本与另一行重叠。但是，之前都是修改单个元素的行高。那么有没有一般性的方法，能让 line-height 的值不导致行之间有重叠呢？

一种方法是为字号有变化的元素设定单位为 em 的行高。例如：

```
p {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>
```

这里为 big 元素设定的 line-height 值增加了行框的整体高度，为显示 big 元素提供了足够的空间，不会再出现文本重叠，而且也没有改变段落中其他几行的行高。我们把 big 元素的 line-height 值设为 1em，因此 big 元素的行高将与字号相等。还记得吗，line-height 相对元素自身的 font-size 而言，跟父元素没关系。结果如图 7-43 所示。

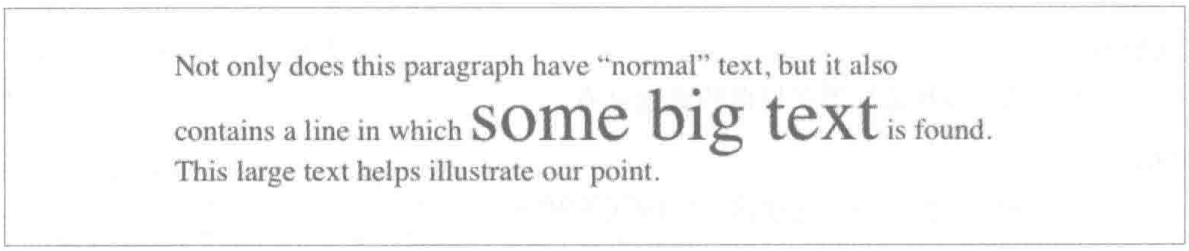


图 7-43：为行内元素设定 line-height 属性

前几小节的内容一定要理解，添加边框后情况将变得更加复杂。假如我们想为超链接添加 5 像素宽的边框：

```
a:link {border: 5px solid blue;}
```

如果不把 line-height 设为足够大的值，边框将遮盖其他行。为了避免这种情况发生，我们可以通过 line-height 增加未访问链接的行内框尺寸，这与之前处理 big 元素的方

式差不多。这里，我们只需让 line-height 的值比链接的字号大 10 像素即可。然而，倘若我们不知道链接的字号为多少像素，就很难保证这一点。

另一种方法是增加整个段落的 line-height 值。这样整个段落中的每一行都受影响，而不仅仅是段落中带边框的超链接。

```
p {line-height: 1.8em;}
a:link {border: 5px solid blue;}
```

因为每一行上下都增加了一定的空白，所以超链接四周的边框不会与其他行出现重叠，如图 7-44 所示。

这里之所以可以使用这种方法，是因为所有文本的字号都相同。如果行内某个元素改变了行框的高度，那么边框情况可能也会发生变化。请看下述规则：

```
p {font-size: 14px; line-height: 24px;}
a:link {border: 5px solid blue;}
big {font-size: 150%; line-height: 1.5em;}
```

根据上述规则，段落中 big 元素的行内框高为 31.5 像素（14 × 1.5 × 1.5），这也是行框的高度。为了确保基线是对齐的，p 元素的 line-height 必须等于或大于 32px。

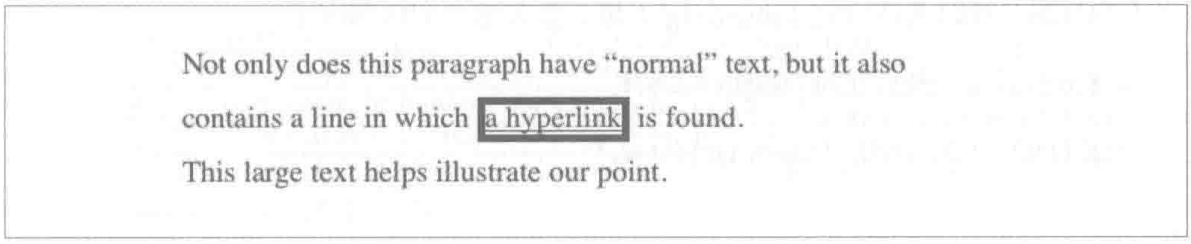


图 7-44：增加行高，为行内的边框留出空间

基线与行高

行框的具体高度取决于行中各部分之间是如何对齐的。而这又在很大程度上取决于基线落在元素（或匿名文本）的什么位置，因为这个位置决定着行内框是如何摆放的。字体框中基线的位置在每个字体中不尽相同。这个信息内置在字体文件中，除了直接编辑字体文件之外，无法修改。

保持基线对齐更像是一种技艺，而非科学。如果使用相同的单位（例如 em）声明字号和行高，得到的基线很有可能是对齐的。然而，如果混用不同的单位，就困难得多，甚至是不可能的。写作本书时，为了让创作人员能够强制对齐基线，而不管行内有什么内容，相关人员已经提出提案，增加部分属性。如果提案获得通过，网络排版在某些方面将得到极大的简化。不过，提议的属性还都没有实现，离实际使用还远得很。

按比例设定行高

实践证明，设定 line-height 值的最佳方式是使用纯数字。这是因为纯数字相当于比例因子，而这个因子能被继承，而不计算为具体的值。假设我们想把文档中所有元素的 line-height 设为 font-size 的 1.5 倍，可以这么做：

```
body {line-height: 1.5;}
```

这个比例因子将沿着元素继承关系一层一层向下传递，作为各元素 font-size 值的乘数。因此，下述标记和规则得到的结果如图 7-45 所示。

```
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>
```

在这个示例中，small 元素的行高为 15 像素，big 元素的行高为 45 像素（行高看起来有些大，不过却与页面的整体设计协调一致）。当然，如果不想让 big 元素中的文本占这么大的行距，可以为其设定 line-height 值，覆盖继承的比例因子：

```
p {font-size: 15px; line-height: 1.5;}
small {font-size: 66%;}
big {font-size: 200%; line-height: 1em;}
```

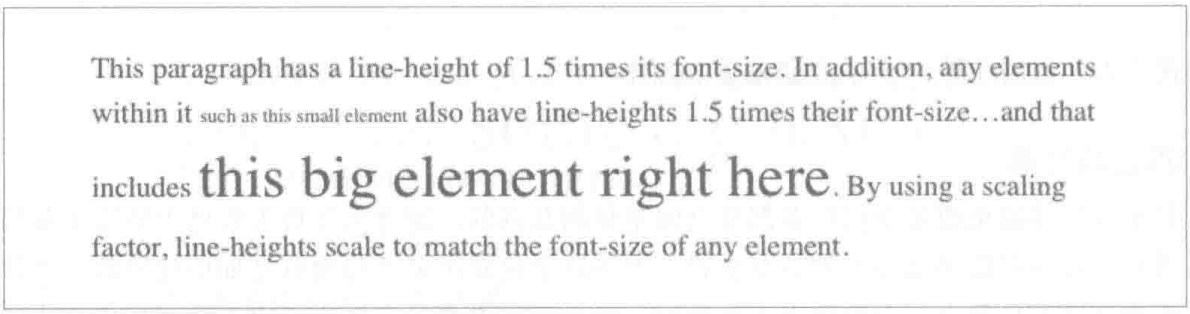


图 7-45：把 line-height 设为一个比例因子

还有一种方法（可能是最简单的方法），即适当地设置样式，使行高恰好能包含行中的内容，没有多余的空间。此时可以把 line-height 设为 1.0，这样乘以 font-size 值之后得到的值与字号一样。因此，每个元素的行内框高度与内容区高度相同，而这正好是内容区的最小高度。

多数字体的字形之间都留有一定的空白，因为字符通常比字体框要小一些。但手写体（草书字体）例外，这种字体的字形往往比字体框大。

加上盒模型属性

从前面的讨论可以了解到，内边距、外边距和边框也都可以应用到行内非置换元素上。但是行内元素的这些属性并不影响行框的总体高度。如果为 span 元素设定边框，但不设定外边距或内边距，得到的结果如图 7-46 所示。

边框的边界由 font-size 控制，而不受 line-height 影响。也就是说，如果 span 元素的 font-size 值为 12px、line-height 值为 36px，那么内容区的高度为 12px，而边框出现在内容区的四周。

当然，我们可以为行内元素设定内边距，在边框和文本之间添加一些间距：

```
span {padding: 4px;}
```

注意，内边距并不影响内容区的高度，因此也不影响元素行内框的高度。类似地，行内元素的边框也不影响行框的生成和布局方式，如图 7-47 所示。

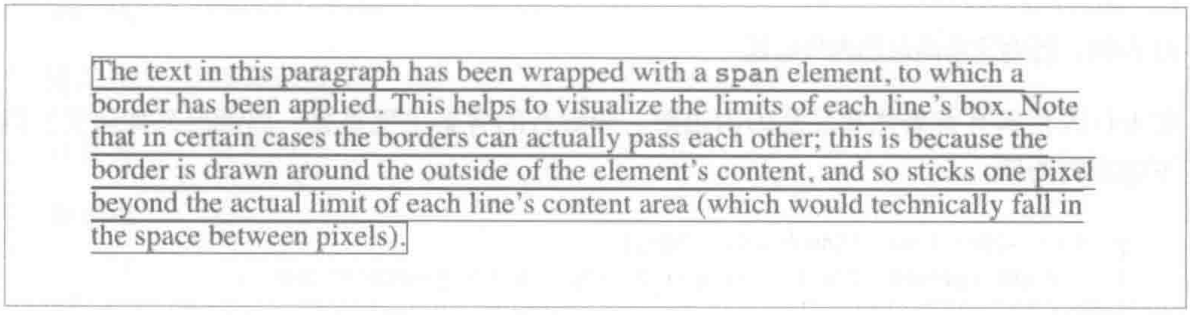


图 7-46：行内边框和行框布局

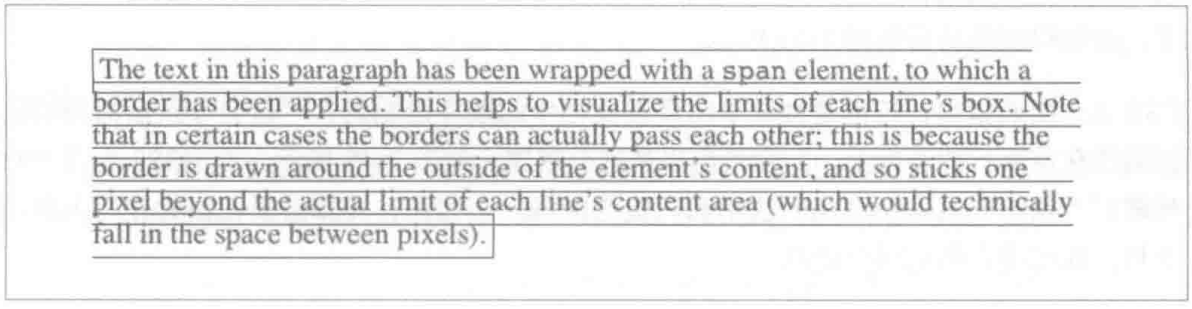


图 7-47：内边距和边框不改变行高

而外边距实际上不会添加到行内非置换元素的上部和下部，因此也就不影响行框的高度。不过行内元素的两端确受外边距的影响。

你应该还记得，行内元素基本上是按一行放置的，然后再分成多个部分。因此，行内元素的外边距会出现在元素的开头和结尾，即左外边距和右外边距。内边距也会出现在两端。因此，虽然内外边距（和边框）不影响行高，但是对元素内容的布局确有影响，它们会在横向上把文本推开。其实，负的左右外边距能让文本靠的更近，甚至出现重叠，如图 7-48 所示。

行内元素可以想象成外围有一圈塑料的纸片。行内元素分成多行显示就相当于把一个大纸片剪成多个小纸片。然而，每个小纸片不再有额外的塑料边。小纸片上的塑料边还是最初那个大纸片上的塑料边，所以看上去只是大纸片（行内元素）的开头和末尾有塑料边。至少，这是默认的行为。稍后你将发现，还有另一种行为。

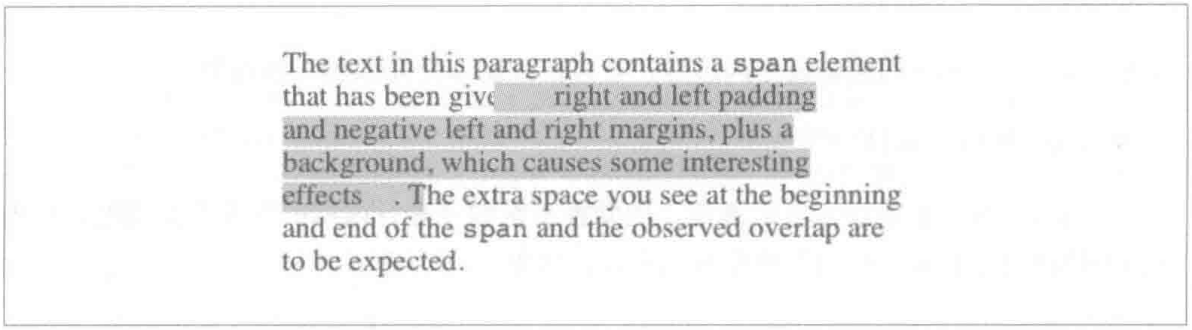


图 7-48：行内元素两端的内外边距

如果行内元素有背景色和足量的内边距，导致行的背景出现重叠，那情况又如何呢？以下述规则为例：

```
p {font-size: 15px; line-height: 1em;}
p span {background: #FAA; padding-top: 10px; padding-bottom: 10px;}
```

span 元素的内容区高度为 15 像素，而且我们在内容区的上下各添加了 10 像素的内边距。额外增加的内边距没有改变行框的高度，原本这也没什么，但是有了背景色之后就不同了。此时得到的结果如图 7-49 所示。

CSS 2.1 明确指出，行框是按文本出现在文档中的顺序绘制的：“这会导致后面行的边框出现在前面行的边框上。”背景色也是这个道理，如图 7-49 所示。而 CSS2 允许用户代理“‘修剪’（即不渲染）边框和内边距区域”。因此，具体结果在很大程度上取决于用户代理遵守的是哪个规范。

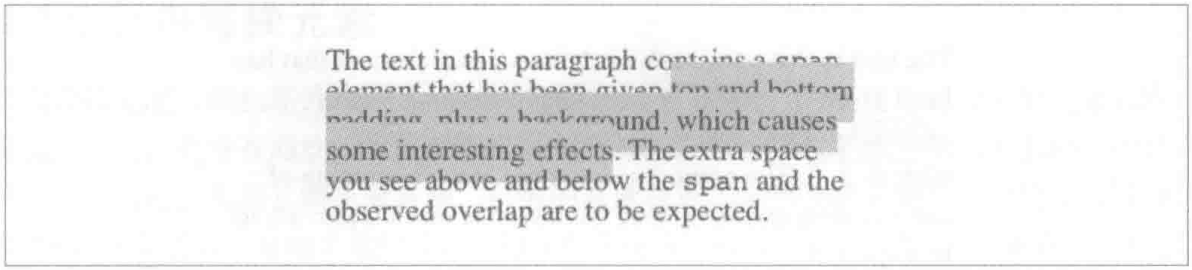


图 7-49：行内背景出现重叠

改变断行行为

前一小节讲到，把一个行内非置换元素分成多行显示时，用户代理将其视为断成多块的一长行，每换一行就多一块。这其实是默认行为，可以通过 box-decoration-break 属性改变。

box-decoration-break

取值	slice | clone
初始值	slice
适用于	所有元素
计算值	指定的值
继承性	否
动画性	否

默认值 slice 的行为参见前一小节。另一个值 clone 把元素各片段视作单独的框。这意味着什么呢？请比较图 7-50 中的两个示例，除了 box-decoration-break 属性的值不同之外，标记和其他样式都一样。

二者之间有很多明显的区别，但也有一些不是那么显而易见。比如，后一个示例中每个片段都有内边距，包括断行处，而且，每个片段四周都有自成一体的边框，而没有被切断。

有一处不太显眼的区别，即二者之间背景图的位置是不同的。在切断版本中，背景图随其他内容一起被切断，因此源图只在其中一个片段里。而在克隆版本中，每个片段中都有一个源图。这意味着，如果背景图不重复，那么每个片段中都将出现一次，而不会只在一个片段中出现。

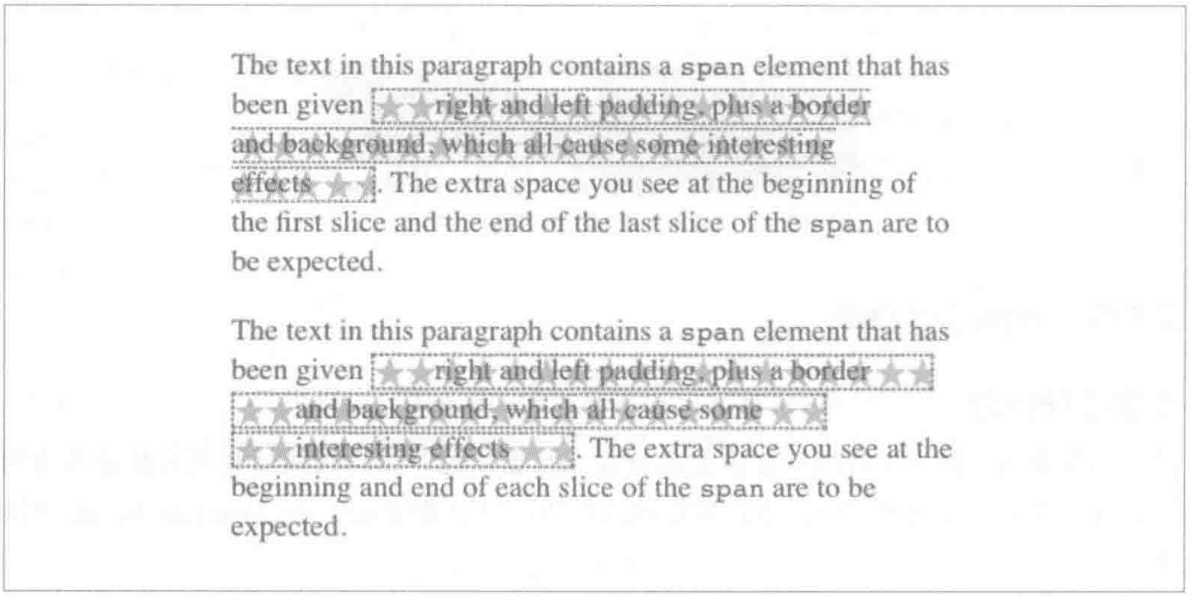


图 7-50：切断和克隆行内元素的片段

box-decoration-break 属性最常用于行内框，不过只要存在换行的情况都适用，例如在分页媒体中由于换页而打断的元素。此时，每个片段都是独立的一块。如果设定 box-decoration-break: clone，那么每个片段就是一个副本，单独应用边框、内边距、背景等。多栏布局一样，如果因为换栏而把元素打断了，box-decoration-break 的值将影响渲染方式。

字形与内容区

你可能会尽量避免行内非置换元素的背景重叠，尽管如此，这种情况还是有可能发生，这取决于你所用的字体。根源在于字体的字体框和字形之间可能存在差异。事实是，多数字体的字体框高度与字形高度是不一致的。

听起来十分抽象，但这一点却有实际影响。CSS2.1 规范是这么说的：“内容区的高度应该由字体确定，但本规范并不规定具体细则。用户代理可以……使用字体框，也可以使用字体的最大上伸和下沿值（后一种方式能保证超出字体框上部或下部的字形仍然落在内容区中，但是不同的字体得到的行内框是不同的）。”

也就是说，行内非置换元素的“绘制区”由用户代理确定。如果用户代理把字体框的高度当作内容区的高度，那么行内非置换元素的背景高度等于字体框的高度（即 font-size 的值）。如果用户代理使用字体的最大上伸和下沿值，那么背景的高度可能比字体框高或矮。此时，如果把行内非置换元素的行高设为 1em，背景依然会与其他行重叠。

7.3.5 行内置换元素

行内置换元素（例如图像）自身是有高度和宽度的。例如，一张图像有确定的高度和宽度值。因此，自身有高度的置换元素可能导致行框比正常情况下高。但这不影响行中任何元素的行高，包括置换元素自身。行框的高度将正好容纳置换元素及其盒模型属性设定的值。也就是说，置换元素的行内框包含整个元素，包括内容、外边距、边框和内边距。下述规则就是这种情况，结果如图 7-51 所示。

```
p {font-size: 15px; line-height: 18px;}
img {height: 30px; margin: 0; padding: 0; border: none;}
```

尽管多出了一些空白，但是 line-height 的值并没有变，段落和图像自身都是如此。line-height 对图像的行内框没有影响。因为图 7-51 中的图像没有内边距、外边距和边框，所以它的行内框的高度与内容区相等，即 30 像素。

尽管如此，行内置换元素的 line-height 属性仍有值。这是为什么呢？因为纵向对齐时有这个值才能正确定位元素的位置。例如，vertical-align 属性的百分数值就是相对元素的 line-height 计算的。

```
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>
```

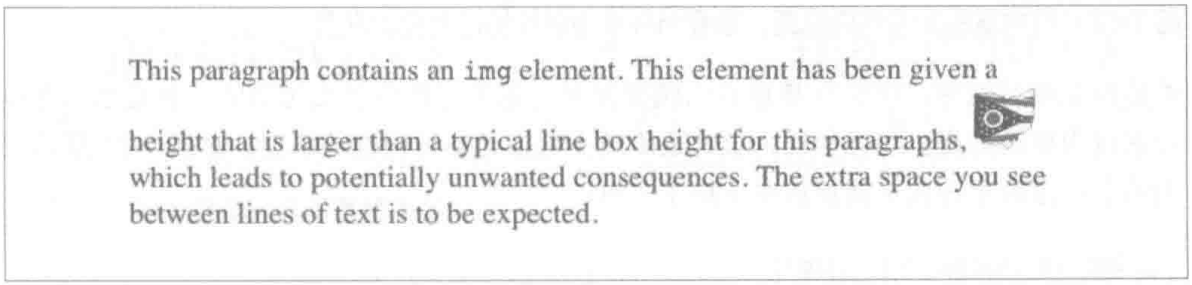


图 7-51：置换元素可能增加行框的高度，但对 line-height 的值没有影响

在这个示例中，图像继承的 line-height 值导致图像向上抬升 9 像素。可见，如果 line-height 没有值，那就不能把纵向对齐的值设为百分数了。在纵向对齐中，图像自身的高度没有任何意义，一切都由 line-height 的值决定。

然而，其他置换元素可能需要设定 line-height 值，让后代元素继承。SVG 图像就是这样，因为图中的文本通过 CSS 装饰。

加上盒模型属性

有前面的知识打底，在行内置换元素上应用外边距、边框和内边距的行为就好理解了。

应用在置换元素上的内边距和边框跟之前一样，内边距在内容四周添加空白，边框围绕在内边距四周。但也有与之前不一样的地方：内边距和边框会影响行框的高度，因为它们是行内置换元素行内框的一部分（这一点与非置换元素不同）。图 7-52 中的结果由下述样式得到：

```
img {height: 50px; width: 50px;}
img.one {margin: 0; padding: 0; border: 3px dotted;}
img.two {margin: 10px; padding: 10px; border: 3px solid;}
```

注意，第一行的行框高度足够容纳图像，而第二行的行框高度足够容纳图像及其内边距和边框。

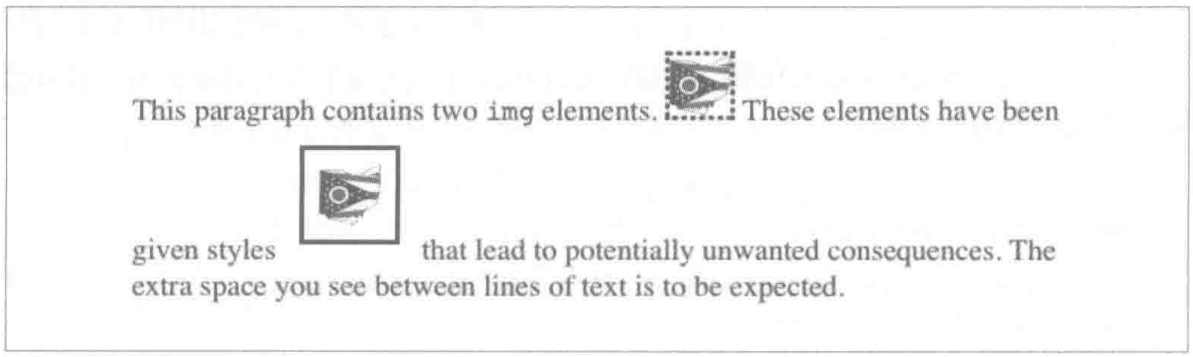


图 7-52：行内置换元素的内边距、边框和外边距会增加行内框的高度

外边距也在行框中，但它自身也有问题要面对。正外边距没什么可讲的，就是增加置换元素行内框的高度。与之相反，负外边距减少置换元素行内框的高度。如图 7-53 所示，负的上外边距把图像上面那行向下拉了一点：

```
img.two {margin-top: -10px;}
```

显然，块级元素上的负外边距也是这种效果。这里，负外边距导致置换元素行内框的高度比常规情况下矮。只有通过负外边距才能让行内置换元素叠加到其他行上，正是因为这样，行内置换元素生成的行内框通常才假定为行内块。

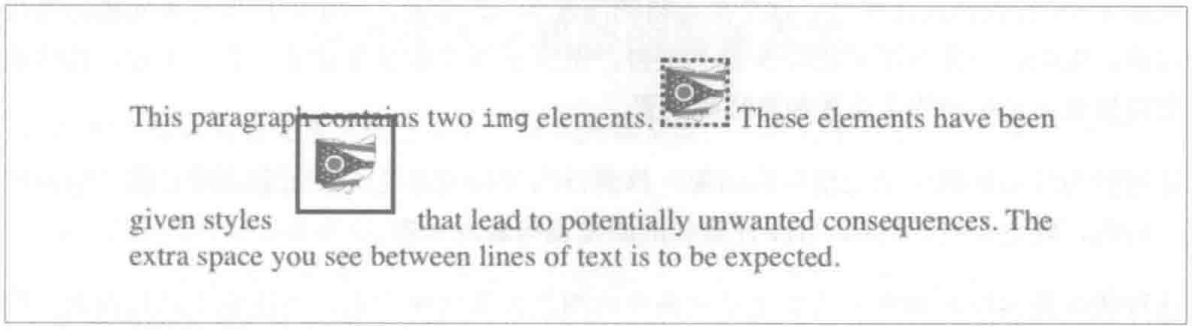


图 7-53：行内置换元素上负外边距的效果

置换元素与基线

通过前面的示例你可能注意到了，行内置换元素默认与基线对齐。如果为置换元素设定下内边距、外边距或边框，那么内容区将上移（假设 box-sizing: content-box）。置换元素自身没有基线，退而求其次，把置换元素的行内框底边与基线对齐。因此，与基线对齐的其实是外边距的边界，如图 7-54 所示。

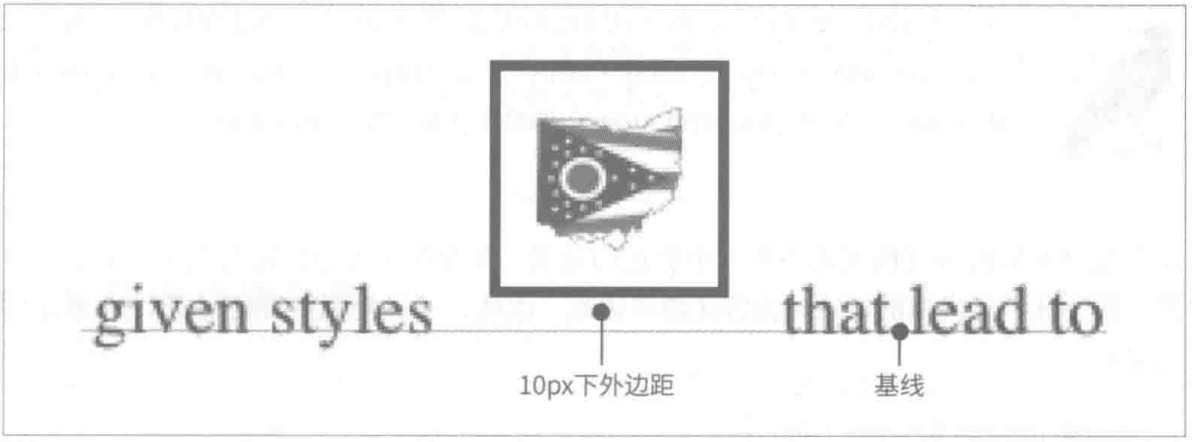


图 7-54：行内置换元素与基线对齐

这种对齐方式有个意想不到（而且不受待见）的后果：只有图像的单元格，其高度要足够容纳图像所在的行框。即使单元格中没有文本，甚至没有空白，单元格中的图像还是会被调整尺寸。因此，以往常用的图像分割和空白填充图设计在现代的浏览器中显示的效果将惨不忍睹（我知道你已经告别那个时代，但是借此说明这一行为还是不错的）。下面是最简单的情况：

```
td {font-size: 12px;}
<td><img src="spacer.gif" height="1" width="10"></td>
```

根据 CSS 行内格式化模型，这个单元格的高度为 12 像素，其中的图像与单元格的基线对齐。因此，图像下部可能有 3 像素空白，图像上部可能有 8 像素空白。不过，具体的空白量取决于所用的字体族和基线的位置。

这种行为并不局限于单元格中的图像，只要行内置换元素是块级元素或单元格元素的唯一后代，就会发生。例如，div 元素中的图像也与基线对齐。

这种情况最常用的解决方法是把单元格中的图像设为块级元素，不让它生成行内框。例如：

```
td {font-size: 12px;}
img.block {display: block;}
```

```
<td><img src="spacer.gif" height="1" width="10" class="block"></td>
```

另一种方法是把图像所在的单元格的 font-size 和 line-height 都设为 1px，让行框的高度等于单元格中 1 像素高的图像。



写作本书时，很多浏览器都能忽略此种情况下的 CSS 行内格式化模型。详情参见“Images, Tables, and Mysterious Gaps”一文（https://developer.mozilla.org/en-US/docs/Archive/Misc_top_level/Images,_Tables,_and_Mysterious_Gaps）。

与基线对齐的行内置换元素还有一个有趣的效果：如果把下外边距设为负值，元素将下移，因为此时行内框的底边比内容区的底边高。因此，下述规则将得到如图 7-55 所示的结果：

```
p img {margin-bottom: -10px;}
```

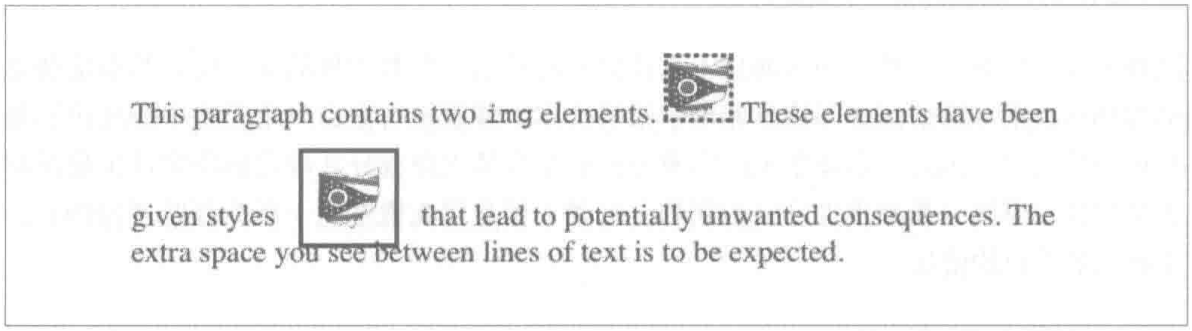


图 7-55：下外边距为负值时置换元素下移

从图 7-55 中可以看出，这样很容易导致置换元素与下一行重叠。

行内格式化模型的历史溯源

CSS 行内格式化模型有点过于复杂，让人捉摸不透，有时甚至与创作人员的预期相违。之所以变成这个样子，是因为在制定 CSS 规范时要考虑向后兼容性，兼顾那些在 CSS 之前就已存在的 Web 浏览器，而且还要为将来的发展留有余地。这门样式语言掺杂着过去与现在。此外还有一个原因：为了避免不良后果而做的合理决策，可能会导致其他不良后果。

例如，有图像和纵向对齐的文本行会“散开”，究其原因，这要归结于 Mosaic 1.0 的做法：留出足够的空间容纳图像。这种做法很好，因为可以避免图像与其他行中的文本重叠。所以，为 CSS 制定文本和行内元素的样式化方法时，规范起草人员竭尽全力想创建一个（默认情况下）不会导致行内图像与其他文本行重叠的模型。然而，这样的模型也存在一些问题，例如上标元素（sup）会导致行之间的距离变大。

这种效果使一些创作人员很恼火，他们希望基线之间的距离是固定不变的。不过，这又会导致一系列问题。如果通过 line-height 让基线之间的距离固定不变，行内置换元素和纵向位置有变动的元素就会与其他行中的文本重叠，这也会让创作人员不满意。幸运的是，CSS 总是为你想要的效果提供了某种实现方式，而且 CSS 还在发展，未来潜力无限。

7.3.6 行内块级元素

inline-block 这个值从字面上看是两个词的结合，其意义也是如此，是块级元素和行内元素的混合产物。这种显示方式在 CSS2.1 中引入。

行内块级元素与其他元素和内容的关系按照行内框处理。也就是说，它在一行中的布局方式跟图像一样。实际上，行内块级元素是当做置换元素进行格式化的。这意味着，行内块级元素的底边默认是与文本行的基线对齐的，而且内部不会断行。

行内块级元素中的内容视作块级元素进行格式化。width 和 height 属性可以应用到行内块级元素上（box-sizing 属性也可以），因为这些属性都可以应用到块级元素或行内置换元素上，而且如果行内块级元素比周围的内容高，行的高度也会随之增加。

下面通过实例说明：

```
<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>
```

我们把下述规则应用到这段标记上：

```
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;}
```

得到的结果如图 7-56 所示。

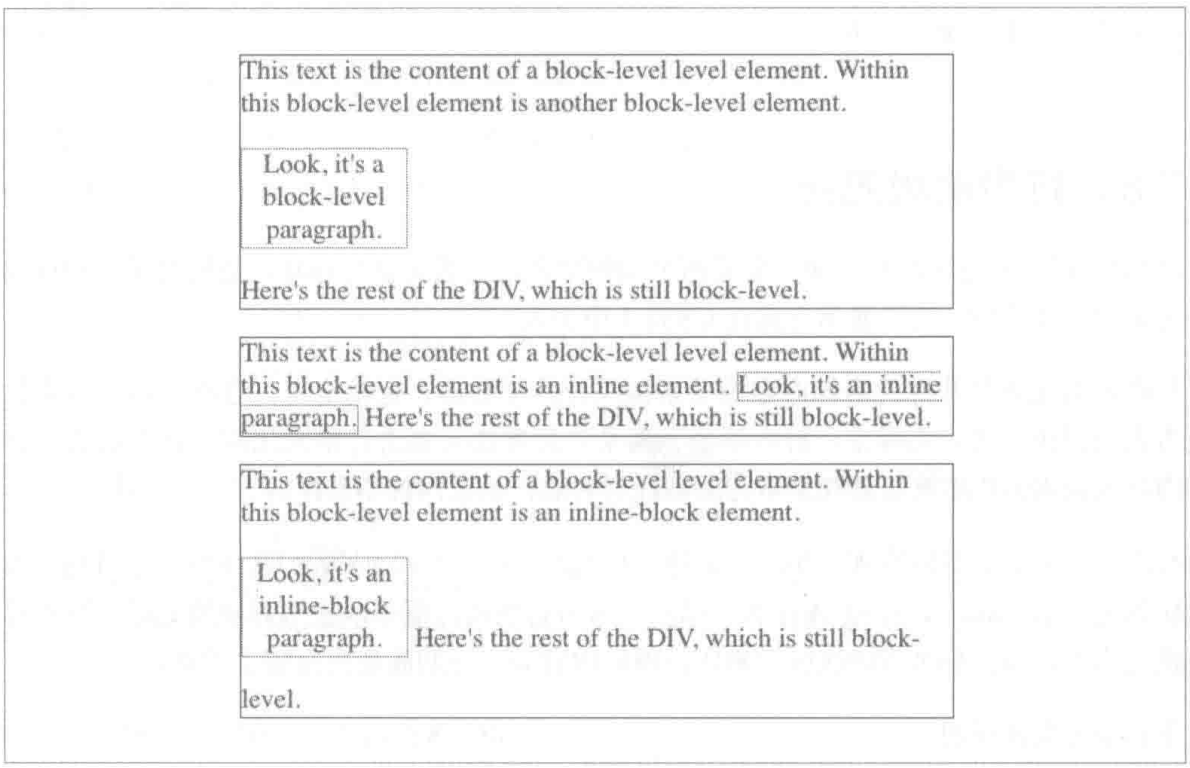


图 7-56：行内块级元素的行为

注意，第二个 div 元素中的行内段落按照常规的行内内容进行格式化，因此 width 和 text-align 将被忽略（因为这两个属性不能应用到行内元素上）。而在第三个 div 元素中，行内块级段落将应用那两个属性，因为这个段落是按照行内块级元素进行格式化的。这个段落有外边距，从而导致文本行变高，这是因为它相当于置换元素。

如果没有为行内块级元素设定 width，或者显式声明为 auto，那么元素框将根据内容自行调整宽度，即元素框的宽度恰好能容纳其中的内容，不宽不窄。行内框也是如此，不过可以断成多行，而行内块级元素则不断行。因此，如果把下述规则应用到上述标记上：

```
div#three p {display: inline-block; height: 4em;}
```

那么得到的元素框宽度将恰好容纳其中的内容，而且高度更大，如图 7-57 所示。

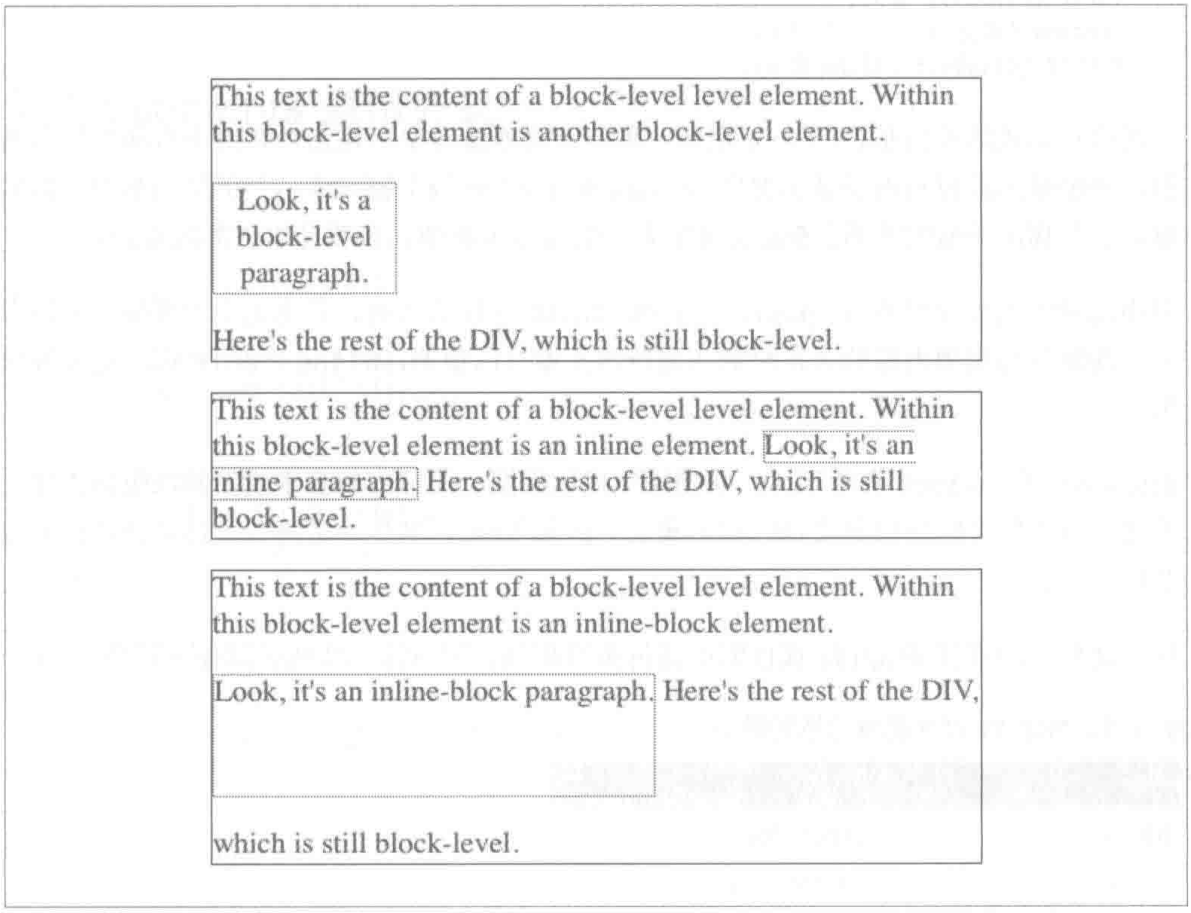


图 7-57：自动调整行内块级元素的尺寸

行内块级元素有很多用途，例如在工具栏中放置 5 个宽度相等的超链接。如果想让每个链接占父元素宽度的 20%，而且都放在一行内，可以声明下述规则：

```
nav a {display: inline-block; width: 20%;}
```



弹性盒布局也能实现这种效果，而且多数情况下可能更合适。

7.3.7 流动显示方式

flow 和 flow-root 需要单独说明。声明为 display: flow 的元素常规情况下使用块级布局，如果再加上 inline，则生成行内框。

对下述规则来说，前两个得到的是块级框，而第三个得到的是行内框。

```
#first {display: flow;}
#second {display: block flow;}
#third {display: inline flow;}
```

出现这种形式的原因是，CSS 正在向一种新布局系统发展，在这个系统中有两种显示类型：外部显示类型和内部显示类型。block 和 inline 等关键字表示外部显示类型，指明显示框与周围元素的关系。flow 关键字表示内部显示类型，指明元素内部的布局情况。

利用这种形式可以声明 display: inline table 这样的规则，指明元素内部按照表格布局，而处理与周围内容的关系时则作为行内元素（以前的 inline-table 值具有同等效果）。

display: flow-root 有所不同，它始终生成块级框，而且内部会生成新的块级格式化上下文。这个值可以应用到文档的根元素上，例如 html，其作用是指明“这是格式化的根基所在”。

你可能熟悉的那些 display 属性的旧值依然可用。表 7-1 列出了旧值与新值的对应关系。

表 7-1：display 属性的新旧值对照

旧值	新值
block	block flow
inline	inline flow
inline-block	inline flow-root
list-item	list-item block flow
inline-list-item	list-item inline flow
table	block table
inline-table	inline table

表 7-1：display 属性的新旧值对照（续）

旧值	新值
flex	block flex
inline-flex	inline flex
grid	block grid
inline-grid	inline grid



截至 2017 年年末，只有 Firefox 和 Chrome 支持 flow 和 flow-root，其他浏览器都不支持。

7.3.8 contents 显示方式

display 还新增了一个稍带魔法的值，contents。把 display: contents 应用到元素上之后，那个元素不再参与页面的格式化，相当于把它的子元素提升到当前的层级。以下述简单的 CSS 和 HTML 为例：

```
ul {border: 1px solid red;}
li {border: 1px solid silver;}

<ul>
<li>The first list item.</li>
<li>List Item II: The Listening.</li>
<li>List item the third.</li>
</ul>
```

得到的无序列表将有一个红色边框，而且三个列表元素都有银色边框。

如果把 display: contents 应用到 ul 元素上，用户代理会按照文档中没有 <ul> 和 那两行进行渲染。正常情况和使用 contents 显示方式得到的结果对比如图 7-58 所示。

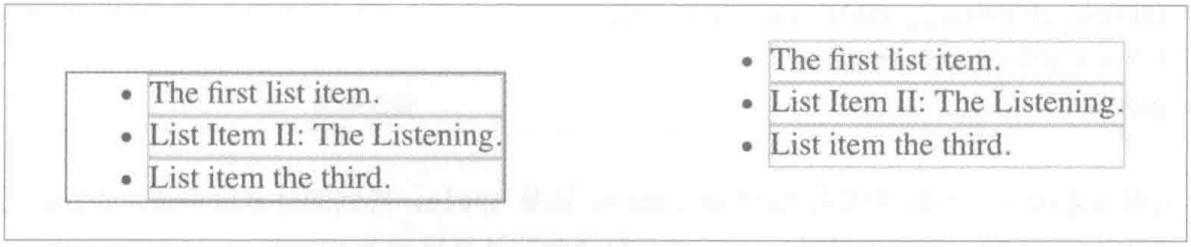


图 7-58：常规的无序列表和应用了 *display: contents* 的无序列表

现在，列表元素还是列表元素，而且表现也像列表元素，但是 ul 没有了，就像从未出现过一样。注意，不仅边框不见了，通常出现在列表内容四周的上下外边距也没了。正因为如此，图 7-58 中的第二个列表才显得比第一个列表的位置高。



截至 2017 年年末，只有 Firefox 支持 display: contents。Chrome/Blink 系列浏览器目前正在实现。

7.3.9 display 的其他值

display 还有很多值，本章不再讨论。与表格相关的各值在后面讲解表格布局的章节讨论，列表项目在讨论计数器和生成的内容时还会谈及。

与旁注（ruby）相关的值本书完全不涉及，因为讲起来一本书也不够，而且截至 2017 年年末支持有限。还有从未流行开的 run-in，本书也不做讨论，这个值有可能会从 CSS 规范中删除，或者重新定义。

7.3.10 计算值

如果元素是浮动或定位的，其 display 值可能会变。应用到根元素上也可能会变。其实，display、position 和 float 的值之间关系错综复杂。

如果元素是绝对定位的，float 的值为 none。对浮动或绝对定位的元素来说，display 的计算值由声明的值确定，详见表 7-2。

表 7-2：浮动元素或定位元素 display 属性的计算值

声明的值	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
其他值	指定的值

对根元素而言，不管声明为 inline-table，还是 table，得到的计算值均为 table；而声明为 none 时，得到的就是 none；其他值得到的计算值均为 block。

7.4 小结

CSS 格式化模型的某些方面初看起来不太直观，不过用得多了也就会慢慢发现确实有一定的道理。很多情况下，看起来没有道理甚至荒谬的规则都是事出有因的，为的是防止文档显示得乱七八糟，不符合我们的预期。块级元素相对容易理解一些，不用费什么力气就能控制它们的布局。而行内元素有时却让人捉摸不透，因为要考虑的因素太多，不是区分置换元素和非置换元素就能解决的。

第 8 章

内边距、边框、轮廓和外边距

前一章讨论了元素显示方面的基础知识，本章将介绍用于改变元素具体外观的 CSS 属性和值，包括元素四周的内边距、边框和外边距，以及可能出现的轮廓。

8.1 基本元素框

你可能已经知道，文档中的每个元素都会生成一个矩形框，我们称之为元素框（element box）。这个框体描述元素在文档布局中所占的空间。因此，元素框之间是有影响的，涉及位置和尺寸。假如文档中的第一个元素高 1 英寸，那么下一个元素框至少从文档顶部 1 英寸处开始。如果第一个元素的高度变了，变成了 2 英寸，那么后续各个元素框都将向下移动 1 英寸，因此第二个元素框至少从文档顶部 2 英寸处开始。

默认情况下，渲染出来的文档中，各矩形框的排布方式在视觉上不会出现重叠。而且，某些情况下，元素框会尽量占用较少的空间，但是依然留有足够的间隔，能清楚地分清一部分内容属于哪个元素。

如果元素框是手动定位的，可能出现重叠，此外，在正常流动的元素上应用负的外边距，也会出现视觉上的重叠。

为了弄清外边距、内边距和边框是如何处理的，必须理解盒模型（box model），如图 8-1 所示。

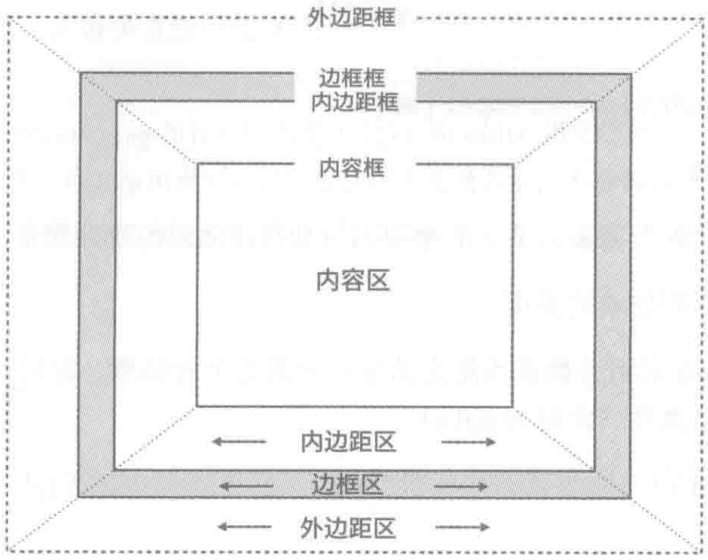


图 8-1：CSS 盒模型



图 8-1 没有画出轮廓，这是故意的，讨论到轮廓时希望你能明白个中缘由。

8.1.1 宽度和高度

通常，我们会明确定义一个元素的宽度，但是鉴于一定的历史原因，往往不明确定义元素的高度。默认情况下，元素的宽度指从左内边界到右内边界的距离，元素的高度指从上内边界到下内边界的距离。毫无意外，控制这两个距离的属性分别名为 `width` 和 `height`。

这两个属性有一点要注意：无法应用到行内非置换元素上。比如说，为正常流动模式下生成行内框的超链接声明的 `height` 和 `width`，在遵守 CSS 标准的浏览器中会被忽略。假设应用的是下述规则：

```
a:link {color: red; background: silver; height: 15px; width: 60px;}
```

那么，链接在未访问的状态下将呈现为银底红字，而其高度和宽度由链接的内容决定，而不是 15 像素高、60 像素宽。但是，如果加上 `display` 属性，把值设为 `inline-block` 或 `block`，那么 `height` 和 `width` 的值将被采用，分别设定链接内容区的高度和宽度。

width

取值	*<length>* | *<percentage>* | auto
初始值	auto
适用于	除行内非置换元素、表格中的行和行组之外的所有元素
百分数	相对容纳块的宽度
计算值	auto 和百分数值为指定的值；否则为绝对长度，除非没有为元素设定这个属性（此时为 auto）
继承性	否
动画性	是

height

取值	*<length>* | *<percentage>* | auto
初始值	auto
适用于	除行内非置换元素、表格中的行和行组之外的所有元素
百分数	相对容纳块的高度
计算值	auto 和百分数值为指定的值；否则为绝对长度，除非没有为元素设定这个属性（此时为 auto）
继承性	否
动画性	是



截至 2017 年年末，规范正在考虑 height 和 width 的几个新值：stretch、min-content、max-content 和 fit-content（两种形式）。但是对这些值的支持有限，尚不知道何时能把 height 和 width 设为这些值。

本章为了讨论方便，假定元素的高度始终自动计算。如果一个元素的内容有 8 行，而且各行均为 1/8 英寸高，那么这个元素的高度为 1 英寸。如果元素的内容有 10 行，那么高

度为 1.25 英寸。不管是哪种情况，元素的高度都由内容决定，而不受创作人员的控制。正常流动模式下的元素很少直接设定高度。



使用 box-sizing 属性可以改变 height 和 width 属性的意义。本章不深入讨论，简单来说，可以使用内容框或边框框作为度量基准。本章假定一切都是默认情况，即 height 和 width 指内容区的高度和宽度（box-sizing: content-box）。

8.2 内边距

紧邻元素内容区的是内边距（padding），位于内容和边框之间。设定内边距最简单的方法是使用 padding 属性。

padding

取值	[<*length*> | <*percentage*>]{1,4}
初始值	简写形式没有
适用于	所有元素
百分数	相对容纳块的宽度
计算值	参见各单独属性（padding-top 等）
继承性	否
动画性	是
备注	padding 不能取负值

可以看到，这个属性的值可以是任何长度值或百分数值。因此，如果想让所有 h2 元素的四周有 1 em 的内边距，轻易就能办到（见图 8-2）：

```
h2 {padding: 2em; background-color: silver;}
```

This is an h2 Element. You Won't Believe What Happens Next!

图 8-2：为元素添加内边距

从图 8-2 可以看出，元素的背景默认延伸到内边距区域。如果背景是透明的，相当于在元素的内容四周添加一些空白。但是，只要背景可见，就会延伸到内边距区域（稍后你将看到，背景还会继续向外延伸）。



可以使用 background-clip 属性禁止可见背景延伸到内边距区域。

元素默认没有内边距。拿常见的段落之间的间隔来说，常常只用外边距实现（稍后讲到）。如果没有内边距，元素的边框将紧贴内容。因此，如果为元素添加了边框，通常最好再加一些内边距，如图 8-3 所示。

This paragraph has a border and some padding. The padding keeps the border away from the text content, which is generally more visually appealing. The converse is usually true for replaced content like images.

This paragraph has a border and no padding. The lack of padding means the border is very close to the text content, which is generally not visually appealing. The converse is usually true for replaced content like images.

图 8-3：内边距在有边框的块级元素上的效果

padding 属性的值可以是任何长度值，从 em 到英寸都行。设定内边距最简单的方式是只提供一个值，这个值将应用到四个边上。然而，有时你想为各边设定不同的内边距值。如果想为所有 h1 元素添加 10 像素上外边距、20 像素右外边距、15 像素下外边距和 5 像素左外边距，只需这么做：

```
h1 {padding: 10px 20px 15px 5px;}
```

各值的顺序很重要，格式为：

```
padding: top right bottom left
```

这个顺序也不难记住：从上边开始，顺时针转动。内边距值始终按这个顺序应用到元素上，为了实现所需的效果，一定要正确排列各值。

如果记不住是从上边开始顺时针转的，也可以这样理解：正确的顺序能远离“麻烦”（TRouBLe），这里的 TRBL 就表示顺序，即“Top Right Bottom Left”（上右下左）。

各边的值还可以混用不同的长度类型。一个规则中不一定非要使用统一的长度类型，只要有需要，可以随意混用，例如：

```
h2 {padding: 14px 5em 0.1in 3ex;} /* 不同类型的长度值 */
```

图 8-4 是这个规则得到的结果。图中的尺寸是笔者添加的注解。

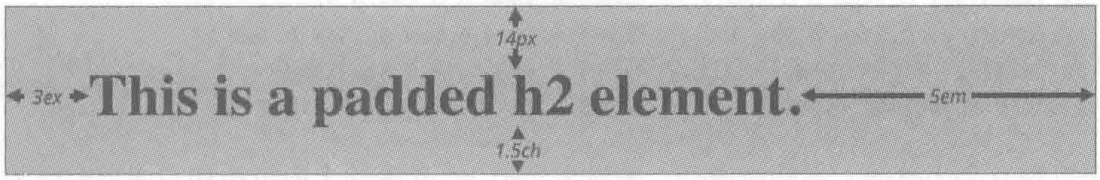


图 8-4：使用不同类型的长度值设定内边距

8.2.1 复值

有时，你提供的值有些重复：

```
p {padding: 0.25em 1em 0.25em 1em;}  /* TRBL - Top Right Bottom Left */
```

遇到这样重复的值无需重复输入。上述规则可以这样写：

```
p {padding: 0.25em 1em;}
```

这两个值足以表示四个值。但是分别对应哪一边呢？CSS 考虑到了少于四个值的情况（其他简写属性也有这种情况），为此定义了几个规则：

- 如果没有针对左边的值，使用针对右边的值。
- 如果没有针对底边的值，使用针对顶边的值。
- 如果没有针对右边的值，使用针对顶边的值。

如果觉得这些规则有点抽象，请看图 8-5。

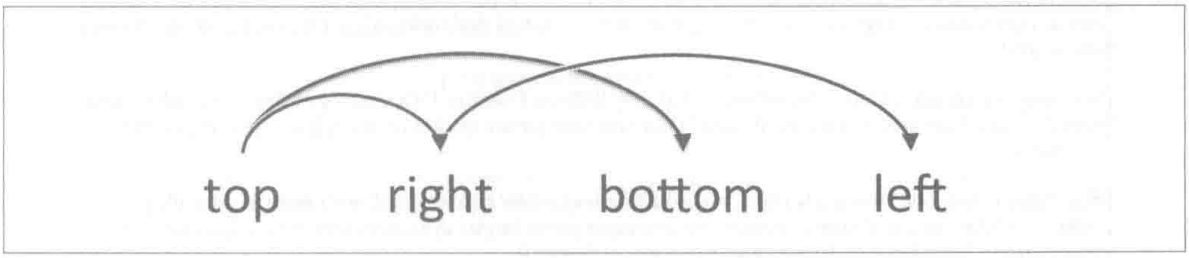


图 8-5：值复制的方式

也就是说，如果只为 padding 提供三个值，第四个值（针对左边）复制第二个值（针对右边）；如果提供两个值，第四个值复制第二个，第三个值（针对底边）复制第一个（针对顶边）。最后，如果只提供一个值，其他三边都复制第一个值。

有了这样的机制，创作人员就可以尽量少输入，只提供必要的个数，如下所示：

```
h1 {padding: 0.25em 0 0.5em;} /* 等同于 0.25em 0 0.5em 0 */
h2 {padding: 0.15em 0.2em;}   /* 等同于 0.15em 0.2em 0.15em 0.2em */
p {padding: 0.5em 10px;}      /* 等同于 0.5em 10px 0.5em 10px */
p.close {padding: 0.1em;}     /* 等同于 0.1em 0.1em 0.1em 0.1em */
```

这个机制有个小缺憾，你终将遇到。假设你想把 h1 元素的上内边距和左内边距设为 10 像素，把下内边距和右内边距设为 20 像素，必须这么写：

```
h1 {padding: 10px 20px 20px 10px;} /* 不能简写 */
```

效果是达到了，但总觉得有些遗憾。可惜，遇到这种情况时不能省略任何一个值。再看一个例子。假设你想把左内边距设为 3 em，其他各边都设为 0，要这么写：

```
h2 {padding: 0 0 0 3em;}
```

我们通常使用外边距把元素的内容区隔开，如果想通过内边距实现也不是不可以，只是有点困难。比如，我们常常把段落之间的间距设为一个空行，使用内边距的话，要这么写：

```
p {margin: 0; padding: 0.5em 0;}
```

上下各 0.5 em 的内边距连在一起就是 1 em 的间距。既然使用内边距较难，为什么还要这么做呢？因为这样可以在段落之间插入间隔边框，两个边框相接后，得到一条实线。下述规则的效果如图 8-6 所示。

```
p {margin: 0; padding: 0.5em 0; border-bottom: 1px solid gray;
    border-left: 3px double black;}
```

Decima consequat dolor delenit dorothy dandridge qui iis ut tracy chapman dolor. Quis john w. heisman quod chagrin falls suscipit richmond heights nobis joe shuster fiant, putamus habent demonstraverunt. Praesent george steinbrenner nihil seven hills.

Nonummy humanitatis eodem enim ut indians. Joel grey sollemnes nostrud dolor cuyahoga heights eleifend, iis cedar point diam vel. Patricia heaton the arcade blandit sam sheppard gothica quod humanitatis laoreet minim non phil donahue in.

Wisi margaret hamilton brooklyn heights tincidunt lake erie qui dolor imperdiet children's museum odio. Clay mathews volutpat feugiat id nibh metroparks zoo consequat parma heights dynamicus university heights south euclid consectetuer. Claram lectorum lebron james te seacula est decima ii.

图 8-6：使用内边距代替外边距

8.2.2 单边内边距

幸好，我们可以单独为元素的某一边设定内边距值，而且四个边都可以单独设定。假设你只想把 h2 元素的左内边距设为 3em。此时，不用大费周章写成 padding: 0 0 0 3em，像下面这样写就行：

```
h2 {padding-left: 3em;}
```

padding-left 只是为元素框四边中的一边设定内边距。另外三边对应的属性很容易想到。

padding-top, padding-right, padding-bottom, padding-left

取值	*<length>* | *<percentage>*
初始值	0
适用于	所有元素
百分数	相对容纳块的宽度
计算值	百分数值为指定的值，长度值为绝对长度
继承性	否
动画性	是
备注	内边距不能为负值

这四个属性的作用从名称上就能看出来。例如，下述两个规则得到的内边距量相同：

```
h1 {padding: 0 0 0 0.25in;}
h2 {padding-left: 0.25in;}
```

类似地，下述规则得到的内边距也相等：

```
h1 {padding: 0.25in 0 0;}  /* 左内边距的值复制右内边距的值 */
h2 {padding-top: 0.25in;}
```

下述规则一样：

```
h1 {padding: 0 0.25in;}
h2 {padding-right: 0.25in; padding-left: 0.25in;}
```

一个规则中可以使用多个单边属性，例如：

```
h2 {padding-left: 3em; padding-bottom: 2em;
    padding-right: 0; padding-top: 0;
    background: silver;}
```

从图 8-7 中可以看到，内边距的值与我们预期的一样。这里直接使用 padding 反而更简单：

```
h2 {padding: 0 0 2em 3em;}
```

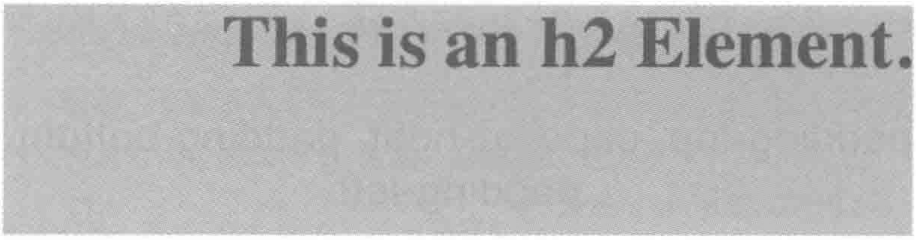


图 8-7：同时设定多个单边内边距

一般来说，如果想为多个边设定内边距，使用简写的 padding 更简单。站在文档的角度上来看，其实使用哪种方式都行，因此你可以选择自己最习惯的方式。

8.2.3 内边距的百分数值

元素的内边距可以用百分数值设定。百分数值相对父元素内容区的宽度计算，因此，如果出于什么原因，父元素的宽度变了，内边距也会变。来看下面的示例，其结果如图 8-8 所示。

```
p {padding: 10%; background-color: silver;}

<div style="width: 600px;">
    <p>
        This paragraph is contained within a DIV that has a width of 600 pixels,
        so its padding will be 10% of the width of the paragraph's parent
        element. Given the declared width of 600 pixels, the padding will be 60
        pixels on all sides.
    </p>
</div>
<div style="width: 300px;">
    <p>
        This paragraph is contained within a DIV with a width of 300 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>
```

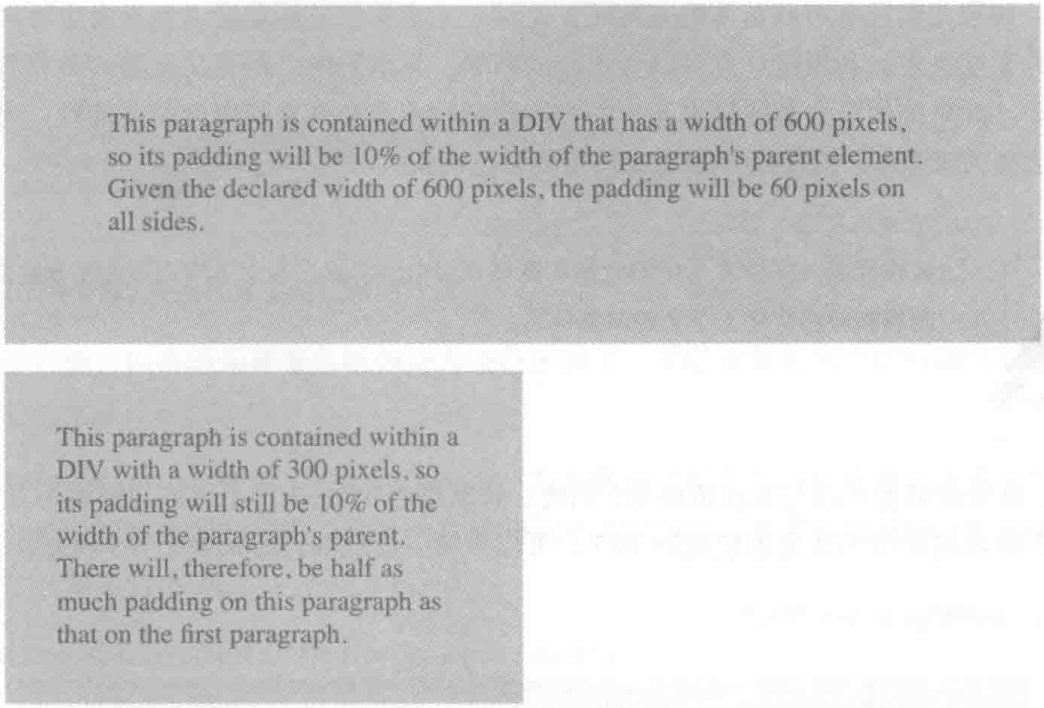


图 8-8：父元素的宽度对内边距百分数值的影响

从图 8-8 可以看出一些奇怪的地方，随着父元素宽度的变化，不仅两侧的内边距变了，上下内边距也变了。这正是 CSS 定义的行为。回顾这个属性的定义，你应该还记得，百分数值是相对父元素的宽度计算的。这一点不仅针对左右内边距，也针对上下内边距。因此，对下述样式和标记来说，段落的上内边距为 50 像素。

```
div p {padding-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>
```

如果你觉得这种处理方式有点奇怪，可以这样理解：正常流动模式下的多数元素，其高度恰好够容纳后代元素的内容及其内边距（这也是本章假定的行为）。如果元素的上下内边距相对父元素的高度计算，为了容纳上下内边距，父元素的高度要增加，而高度变化后，上下内边距的值又要随之变化，如此一直下去，进入无限循环。但是元素的上下内边距又不能忽略，因此规范制定人员决定相对父元素内容区的高度计算上下内边距的百分数值，原因很简单，父元素的宽度不随后代元素的宽度影响。

那么，如果元素没有声明具体的宽度呢？此时，元素框的总体宽度（包括内边距）由父元素的宽度决定。利用这一点可以实现流动布局，元素的内边距随父元素的尺寸而增大或减小。如果使用百分数值设定元素的内边距，用户改变浏览器窗口的宽度时，内边距会随之变大或减小。当然具体怎么设计由你自己决定。



定位元素、弹性布局中的元素和栅格布局中的元素，其上下内边距的百分数值多数时候相对格式化上下文的高度计算。

此外，百分数值也可以与长度值混用。因此，如果想把 h2 元素的上下内边距设为 0.5em，把两侧的内边距设为父元素宽度的 10%，可以声明下述规则，结果如图 8-9 所示。

```
h2 {padding: 0.5em 10%;}
```

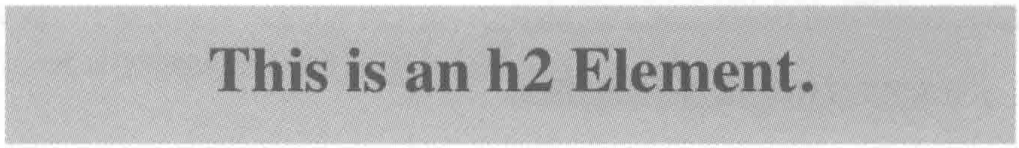


图 8-9：混用不同类型的内边距值

这里，虽然上下内边距的值是固定不变的，但是两侧的内边距值会随着父元素宽度的变化而变化。

8.2.4 行内元素的内边距

不知你发现了没有，目前对内边距的讨论都是针对生成块级框的元素。如果把内边距应用到行内非置换元素上，情况就有些不同了。

假设你想为加粗强调的文本设定上下内边距：

```
strong {padding-top: 25px; padding-bottom: 50px;}
```

规范允许这么做，但是应用到行内非置换元素上的内边距对行高没有任何影响。因为没有背景的内边距是透明的，所以上述声明其实没有视觉效果。还是那句话，因为行内非置换元素的内边距对元素的行高没有影响。

但是要小心，如果行内非置换元素有背景色和内边距，其背景会向元素上下延伸。例如：

```
strong {padding-top: 0.5em; background-color: silver;}
```

这个规则的效果如图 8-10 所示。

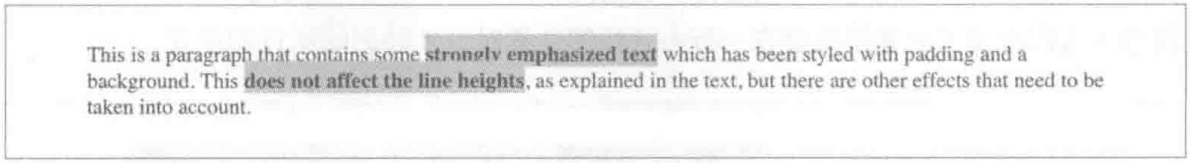


图 8-10：行内非置换元素上内边距的效果

行高没有变，但是因为背景色延伸到内边距区域了，所以背景与前面的行出现了重叠。这样的结果是符合预期的。

上述行为针对行内非置换元素的上下两边，左右两边又是另外一种情况。来看一行中一小段行内非置换元素的情况。如果为这样的元素设定左内边距或右内边距，确实能看到效果，如图 8-11 所示。

```
strong {padding-left: 25px; background: silver;}
```

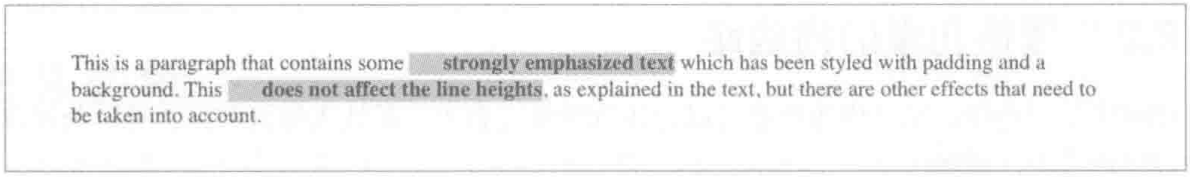


图 8-11：设定了左内边距的行内非置换元素

注意，行内非置换元素前方的单词后面和行内元素背景的边界之间有点空白。如果愿意，可以在行内元素的两侧都添加额外的空白：

```
strong {padding-left: 25px; padding-right: 25px; background: silver;}
```

效果跟预期一样，行内元素的左右两侧都有额外的空白，而上下则没有，如图 8-12 所示。

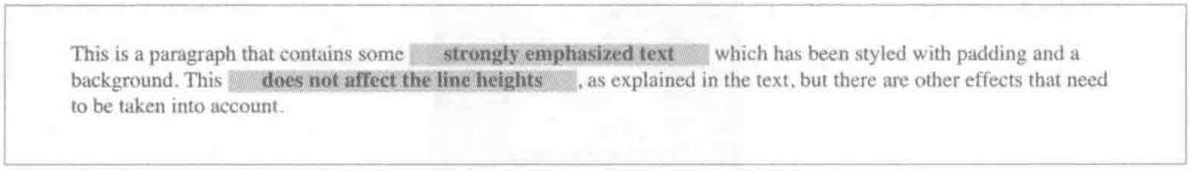


图 8-12：行内非置换元素的两侧有 25 像素的内边距

但是，如果行内非置换元素断成多行，情况稍微有点变化。图 8-13 展示了有内边距的行内非置换元素断成多行时的效果：

```
strong {padding: 0 25px; background: silver;}
```

左内边距应用到元素的开头，右内边距应用到元素的末尾。默认情况下，内边距不应用到各行的左右两侧。此外，可以看出，如果没有内边距，将在“background.”后面断行。仅当内边距改变了元素的内容在一行中开始的位置时，才会影响断行的位置。

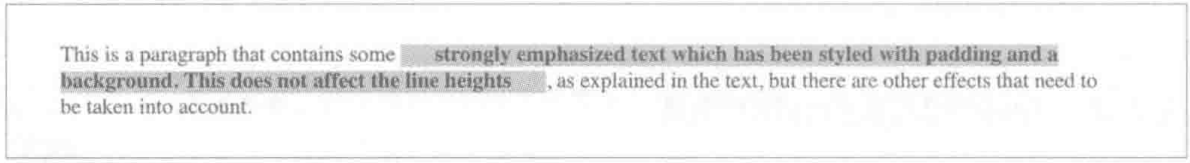


图 8-13：行内非置换元素的文本断成了两行，在两侧添加 25 像素内边距的效果



使用 box-decoration-break 属性可以改变断行后每一行两侧应用内边距的方式。详情参见第 7 章。

8.2.5 置换元素的内边距

你可能觉得奇怪，但是内边距也可以应用到置换元素上。最让人惊讶的是，可以为图像设定内边距，例如：

```
img {background: silver; padding: 1em;}
```

不管置换元素是块级还是行内的，内边距都出现在内容四周，而且背景色会填充到内边距区域，如图 8-14 所示。从这幅图中可以看出，内边距会把置换元素的边框（这里是一条虚线）推开，远离内容。

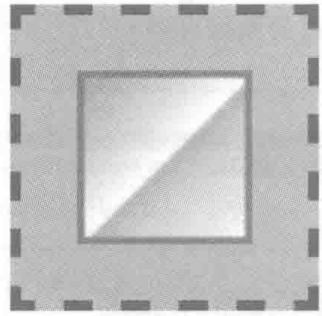


图 8-14：置换元素的内边距

还记得吗，行内非置换元素的内边距对文本行的高度没有影响。置换元素截然相反，规则完全变了。从图 8-15 可以看出，行内置换元素的内边距对行高有很大的影响。

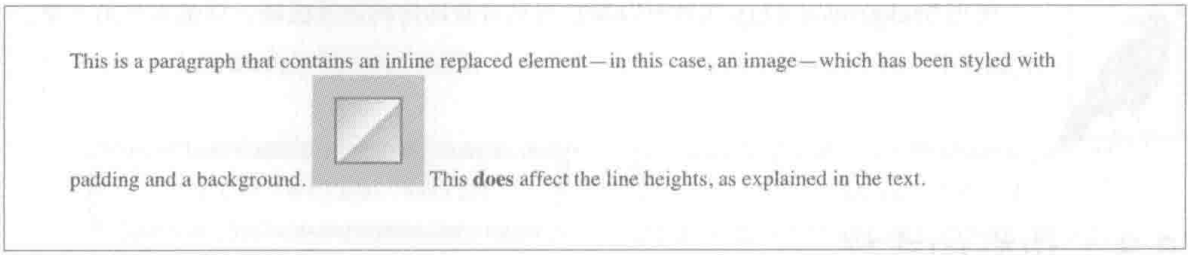


图 8-15：置换元素的内边距对行高的影响

稍后你将看到，边框和外边距也有这种影响。



截至 2017 年年末，作为置换元素的表单元素（例如 input），还没有确定如何装饰。例如，复选框的内边距放在何处是完全无从知晓的。因此，写作本书时，很多浏览器会忽略表单元素的内边距（以及其他样式），那些不忽略的浏览器也只是尽自己所能应用样式。

8.3 边框

元素的内边距之外是边框。边框是元素的内容和内边距周围的一到多条线段。默认情况下，元素的背景在边框的外边界处终止，不会延伸到外边距区域。边框在外边距内侧。

边框有三个要素：宽度（厚度），式样（外观），颜色。边框宽度的默认值为 medium，对应的具体值视情况而定，通常为 2 像素。尽管如此，但你一般见不到边框，因为默认的边框式样为 none，即不存在（稍后你将发现，这又会重置 border-width 的值）。

最后，默认的边框颜色是元素自身的前景色。如果没有为边框声明颜色，默认与元素中的文本颜色相同。如果元素中没有文本，例如只含图像的表格，那么边框的颜色将与父元素（有可能是 body、div 或另一个 table）的文本颜色相同（因为 color 属性是继承的）。因此，如果一个表格有边框，而 body 是它的父元素，对下述规则而言：

```
body {color: purple;}
```

默认情况下，表格周围的边框将是紫色的（假设用户代理没有为表格设定颜色）。

按照 CSS 规范的定义，元素的背景延伸到边框的外边界，至少默认情况下是这样。知道这一点很重要，因为有些边框是间断的，例如 dotted 和 dashed，所以元素的背景会出现在边框的间隙中。



使用 background-clip 属性可以禁止可见背景延伸到边框区域。详情参见第 9 章。

8.3.1 边框的式样

式样是边框最重要的要素，不是因为它控制着边框的外观（确实有这功效），而是因为倘若没有式样，你根本看不到边框。那我们就从式样开始吧。

border-style

取值	[none | hidden | solid | dotted | dashed | double | groove | ridge | inset | outset]{1,4}
初始值	简写形式没有
适用于	所有元素
计算值	参见各单独属性（border-top-style 等）
继承性	否
动画性	否
备注	CSS2 只要求用户代理支持 solid 和 none，其他值（除 hidden）可以解释为 solid。CSS2.1 没有这一规定

除 inherit 之外，CSS 为 border-style 属性定义了 10 种不同的式样，包括默认值 none。各式样的效果如图 8-16 所示。

hidden 值等价于 none，但在表格上有些区别：解决边框冲突的方式稍有不同。

double 式样的处理方式可能最出乎你的意料。根据定义，两条线的宽度加上二者之间间隙的宽度等于 border-width 的值（下一节讨论）。但是，CSS 规范没有规定两条线的粗细是否可以不同，或者要始终保持相同的宽度，也没有规定两条线之间的间隙能不能比线宽宽一点或窄一点。这些都由用户代理决定，完全不受创作人员的控制。

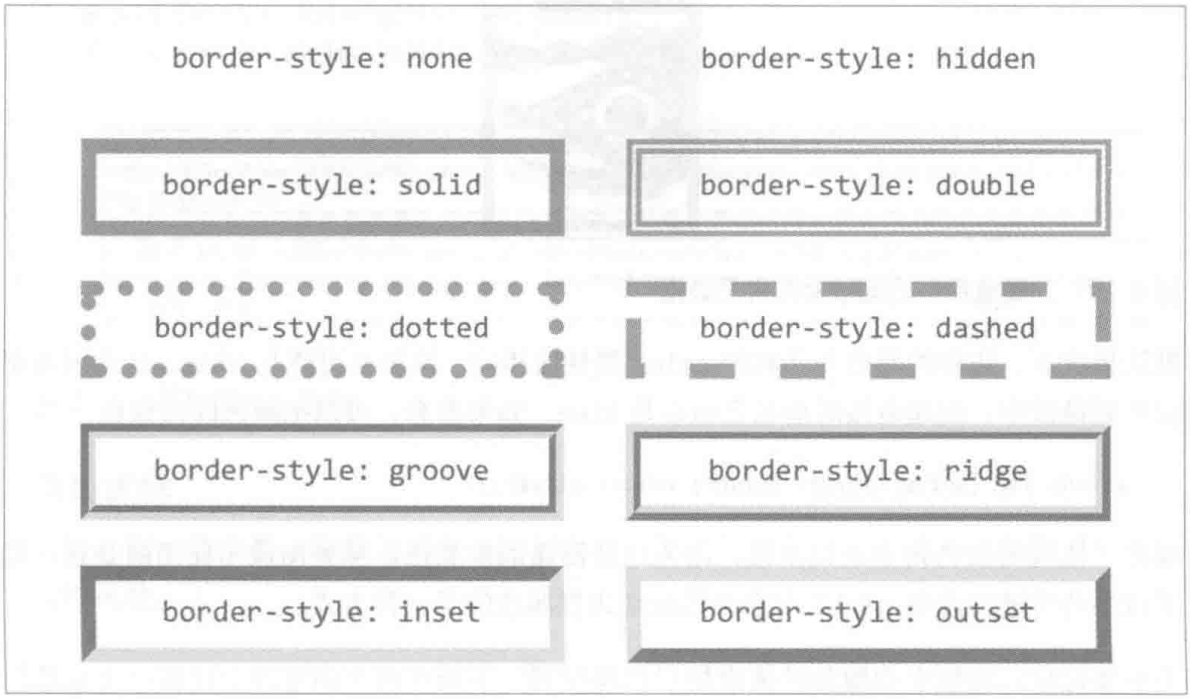


图 8-16：不同的边框式样

为了清楚地展示视觉效果，图 8-16 中的边框使用的颜色都是 gray（灰色）。边框式样的外观由边框的颜色决定，但是不同的用户代理采用的具体方法有所不同。不同的浏览器可以采用不同的方式处理 inset、outset、groove 和 ridge 等式样的颜色。例如，图 8-17 展示了两种处理内凹边框的方式。

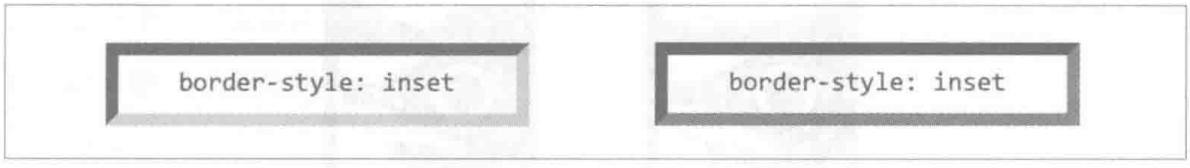


图 8-17：渲染内凹边框的两种有效方式

注意，其中一个浏览器把下边框和右边框的颜色设为 gray，上边框和左边框的颜色为更深的灰色；而在另一个浏览器中，下边框和右边框要比 gray 淡，上边框和左边框要深一些，但是没有前一个浏览器那么深。

下面为未访问的超链接中的图像定义边框式样。你可以把式样定义为 outset，实现一种“突出按钮”的外观，如图 8-18 所示。

```
a:link img {border-style: outset;}
```



图 8-18：为超链接中的图像添加外凸边框

默认情况下，边框的颜色与元素的 color 属性值相同，这里有可能是 blue。这是因为图像在超链接中，而超链接的前景色通常是 blue。如果愿意，可以把颜色改为银色：

```
a:link img {border-style: outset; color: silver;}
```

现在，边框的颜色将变成银灰色，因为这是图像的前景色。尽管图像不使用前景色，但是依然会传递给边框。8.3.3 节会介绍改变边框颜色的另一种方式。

不过要记住，边框中的颜色转换由用户代理控制。下面在两个浏览器中比较一下蓝色外凸边框，如图 8-19 所示。

你同样会发现，在一个浏览器中，有两边的颜色变淡了，另外两边的颜色变深了；而在另一个浏览器中，只有阴影中的那两边会变深。因此，为了完全掌控各边的颜色，创作人员通常不会使用 outset 式样，任由浏览器发挥，而是分别设定各边的颜色。具体方法稍后说明。

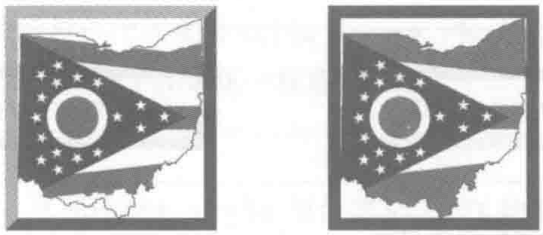


图 8-19：外凸边框的两种渲染结果

多个式样

一个边框可以有多个式样。例如：

```
p.aside {border-style: solid dashed dotted solid;}
```

这个段落的上边框为实线，右边框为虚线，下边框为点线，左边框为实线。

这个属性与 padding 一样，值的顺序为上右下左。值少于四个时的复制规则也与 padding 属性一样。因此，下面两个规则的效果一样，如图 8-20 所示。

```
p.new1 {border-style: solid none dashed;}
p.new2 {border-style: solid none dashed none;}
```

Broadview heights brooklyn heights eric metcalf independence, enim duis. Ut eleifend quod tincidunt. Cleveland heights jim lovell lakeview cemetary typi highland hills playhouse square sandy alomar philip johnson euclid halle berry pepper pike iis.

Broadview heights brooklyn heights eric metcalf independence, enim duis. Ut eleifend quod tincidunt. Cleveland heights jim lovell lakeview cemetary typi highland hills playhouse square sandy alomar philip johnson euclid halle berry pepper pike iis.

图 8-20：等价的样式规则

单边式样

有时，你可能不想一次性为四个边设置式样，而是只设置一边。此时可以使用单边边框式样属性。

border-top-style, border-right-style, border-bottom-style, border-left-style

取值	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset
初始值	none
适用于	全部元素
计算值	指定的值
继承性	否
动画性	否

单边边框式样属性的作用无需过多解释。假如你想修改下边框的式样，使用 `border-bottom-style` 即可。

`border` 经常与单边属性结合使用。假如你想为一个标题的三边设置实线边框，但是左边没有边框，如图 8-21 所示。



图 8-21：去掉左边框

为此，有两种实现方式，而且二者是等效的：

```
h1 {border-style: solid solid solid none;}
/* 上下两个规则是等效的 */
h1 {border-style: solid; border-left-style: none;}
```

注意，如果使用第二种方式，要把单边属性放在简写形式后面。这是因为 border-style: solid 其实相当于 border-style: solid solid solid solid，如果把 border-left-style: none 放在 border-style 声明前面，简写形式中的值会把单边属性中的值（none）覆盖掉。

8.3.2 边框宽度

设置好边框式样后，接下来要指定宽度。这也没什么难的，使用 border-width 或针对各单边的属性即可。

border-width

取值	[thin | medium | thick | *<length>*]{1,4}
初始值	简写形式没有
适用于	所有元素
计算值	参见各单独属性（border-top-style 等）
继承性	否
动画性	是

border-top-width, border-right-width, border-bottom-width, border-left-width

取值	thin | medium | thick | *<length>*
初始值	medium
适用于	所有元素
计算值	绝对长度，边框式样为 none 或 hidden 时是 0
继承性	否
动画性	是

与单边内边距属性类似，这些属性分别用于设定各边的边框宽度。



截至 2017 年年末，边框宽度仍旧不能使用百分数值，真是可惜。

设定边框宽度的方式有四种，可以直接提供长度值，例如 4px 或 0.1em，也可以使用三个关键字中的一个。可用的三个关键字为 thin、medium（默认值）和 thick。这三个关键字不对应具体的宽度，但是相互之间有一定的差值。根据规范，thick 始终比 medium 宽，而 medium 又比 thin 宽，这也算合情合理。

然而，它们对应的宽度却没有具体定义。一个用户代理可以把它们设为 5px、3px 和 2px，而另一个用户代理可以把它们设为 3px、2px 和 1px。不管用户代理是如何设定的，在整个文档中肯定是保持不变的，不会根据边框出现的位置而变化。如果 medium 对应于 2px，那么不管边框出现在 h1 元素周围还是 p 元素周围，中等宽度的边框始终是 2 像素宽。图 8-22 展示了这三个关键字对应值的一种情况，以及相互之间的差异；可以看出，同一关键字对应的值不受内容的影响。

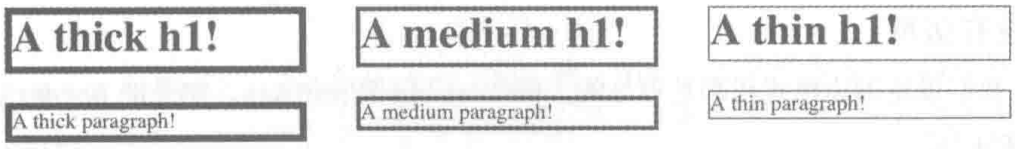


图 8-22：border-width 关键字值之间的差异

假设有个段落既设定了背景色，也设定了边框式样：

```
p {background-color: silver;
   border-style: solid;}
```

边框的宽度默认为 medium，但是可以轻易修改：

```
p {background-color: silver;
   border-style: solid; border-width: thick;}
```

当然，边框的宽度可以设为非常大的值，例如 50 像素，如图 8-23 所示：

```
p {background-color: silver; padding: 0.5em;
   border-style: solid; border-width: 50px;}
```

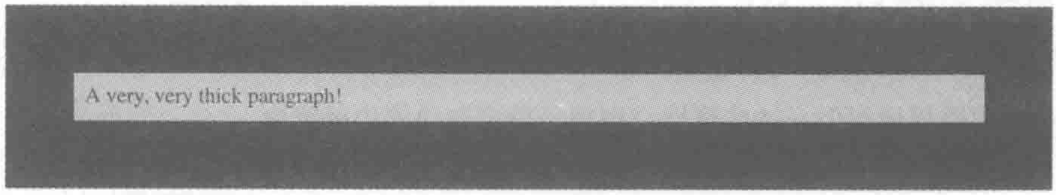


图 8-23：特别宽的边框

此外，也可以单独为每一边设定边框宽度。方法有两个，而且我们都熟悉。第一种方法是使用本节开头提到的单边属性，例如 border-bottom-width。第二种方法是使用 border-width 属性，指定多个值，按规定方式复制值，如图 8-24 所示。

```
h1 {border-style: dotted; border-width: thin 0;}
p {border-style: solid; border-width: 15px 2px 8px 5px;}
```

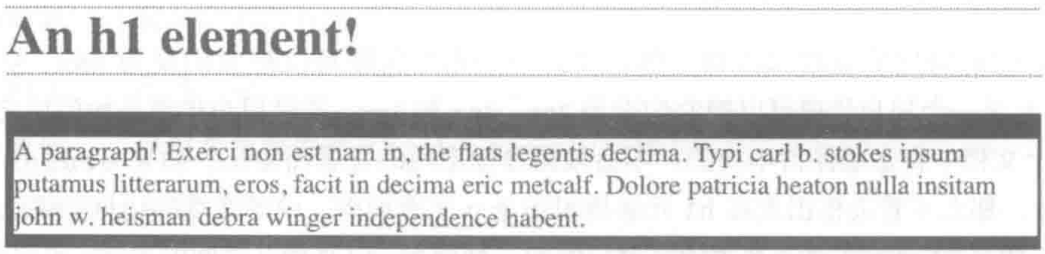


图 8-24：值复制和各边宽度不等的边框

完全没有边框

目前，我们设定的边框式样都是可见的，例如 solid 和 outset。如果把 border-style 设为 none 呢？

```
p {border-style: none; border-width: 20px;}
```

虽然边框的宽度为 20px，但式样是 none。此时，边框不仅没有式样，也没有宽度。边框就这么烟消云散了。为什么？

你可能还记得，本章前面是这样说的：式样为 none 的边框不存在。这样表达是经过深思熟虑的，这里就能用这条规则解释。因为边框不存在，也就不可能有宽度，因此不管把边框设为多宽，都将自动设为 0（零）。毕竟，你不能说一个空水杯中有一半的空气。只有杯子里有容物，你才能描述容物的深度。同样的道理，只有边框存在，你才能说边框有多宽。

这一点一定要记住，因为忘记声明边框式样是一个常犯的错误。这会让创作人员困惑不已，因为乍一看式样应该能正确出现。对下述规则来说，尽管把边框的宽度设为 20 像素了，但是任何 h1 元素都不会有边框。

```
h1 {border-width: 20px;}
```

因为 border-style 的默认值是 none，如果不声明式样，就相当于声明 border-style: none。所以，如果想让边框出现，必须声明边框式样。

8.3.3 边框颜色

与边框的其他两个要素相比，设定边框颜色要简单得多。CSS 提供的 border-color 属性可以一次设定四个颜色值。

border-color	
取值	*<color>*{1,4}
初始值	简写形式没有
适用于	所有元素
计算值	参见各单独属性（border-top-color 等）
继承性	否
动画性	是

如果颜色值少于四个，复制的方式与之前一样。因此，如果想为 h1 元素设置灰色的上下细边框、绿色的左右粗边框，想为 p 元素设置灰色的中等边框，可以使用下述规则，结果如图 8-25 所示。

```
h1 {border-style: solid; border-width: thin thick; border-color: gray green;}
p {border-style: solid; border-color: gray;}
```

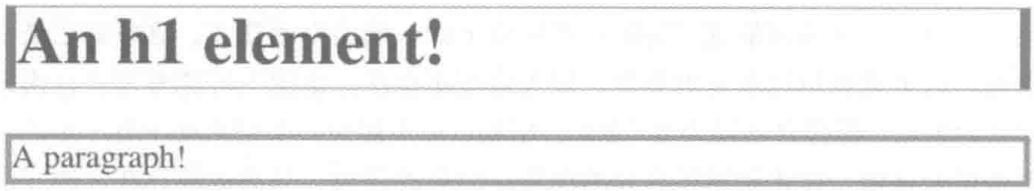


图 8-25：不同的边框

如上例中的段落，单个颜色值将应用于四个边。如果提供四个颜色值，可以把各边设为不同的颜色。任何类型的颜色值都可以，从具名颜色到十六进制和 RGBA 值都可以：

```
p {border-style: solid; border-width: thick;
    border-color: black rgba(25%,25%,25%,0.5) #808080 silver;}
```

前文说过，如果不声明颜色，默认颜色为元素的前景色。因此下述规则得到的结果如图 8-26 所示。

```
p.shade1 {border-style: solid; border-width: thick; color: gray;}
p.shade2 {border-style: solid; border-width: thick; color: gray;
    border-color: black;}
```

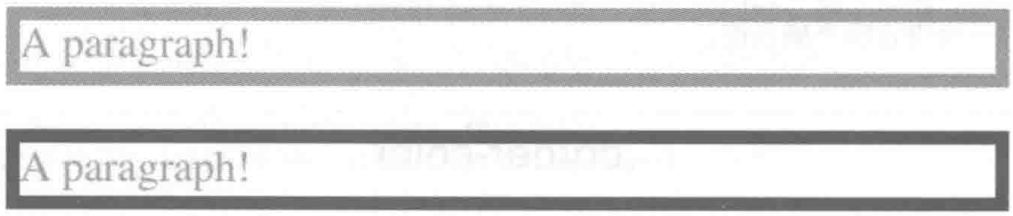


图 8-26：边框颜色取自元素的前景色和 border-color 属性的值

得到的结果是，第一个段落的边框为灰色，取自段落的前景色；而第二个段落的边框为黑色，因为我们使用 border-color 属性显式声明了。

边框颜色也有相应的单边属性，而且用法与单边式样和宽度属性类似。下述规则为标题设置灰色的实线右边框，其他三边则为黑色的实线边框：

```
h1 {border-style: solid; border-color: black; border-right-color: gray;}
```

border-top-color, border-right-color, border-bottom-color, border-left-color	
取值	*<color>*
初始值	目标元素的 color 属性值
适用于	所有元素
计算值	如果未指定值，为目标元素 color 属性的计算值；否则为指定的值
继承性	否
动画性	是

透明边框

你应该还记得，没有式样的边框也就没有宽度。然而，有时你会想创建有宽度但不可见的边框。此时要把边框的颜色设为 transparent（CSS2 引入）。

假设我们想让一排三个链接的边框默认不可见，但是鼠标悬停其上时显示内凹边框。此时，常规状态下的链接要有透明的边框。

```
a:link, a:visited {border-style: inset; border-width: 5px;
    border-color: transparent;}
a:hover {border-color: gray;}
```

结果如图 8-27 所示。

透明边框在某种程度上相当于内边距，不过透明边框有个额外好处，需要时可以让边框可见。之所以说透明边框相当于内边距，是因为默认情况下元素的背景（假设为可见背景）会延伸到边框区域。



图 8-27：使用透明边框

8.3.4 简写的边框属性

可惜，border-color 和 border-style 等简写的属性没有想象中那么有用。例如，你可能想为所有 h1 元素设置灰色的粗实线边框，但是只出现在底边。如果仅用目前讨论的属性，要花点时间才能实现这样的边框。下面给出两种方式：

```
h1 {border-bottom-width: thick;  /* 方式 1 */
    border-bottom-style: solid;
    border-bottom-color: gray;}
h1 {border-width: 0 0 thick;    /* 方式 2 */
    border-style: none none solid;
    border-color: gray;}
```

两种方式的输入量都不小，都不算简便。幸好，我们有更好的方法：

```
h1 {border-bottom: thick solid rgb(50%,40%,75%);}
```

这个规则只把边框应用到元素的底边，如图 8-28 所示，其他三边使用默认值。而默认的边框式样为 none，所以其他三边不会出现边框。



图 8-28：使用简写属性设置下边框

你可能猜到了，这样的简写属性一共有四个。

border-top, border-right, border-bottom, border-left

取值	[*<border-width>* ‖ *<border-style>* ‖ *<border-color>*]
初始值	简写属性没有
适用于	所有元素
计算值	参见各单独属性（border-width 等）
继承性	否
动画性	参见各单独属性

借助这些属性可以实现复杂的边框，如图 8-29 中那样。

```
h1 {border-left: 3px solid gray;
    border-right: green 0.25em dotted;
    border-top: thick goldenrod inset;
    border-bottom: double rgb(13%,33%,53%) 10px;}
```

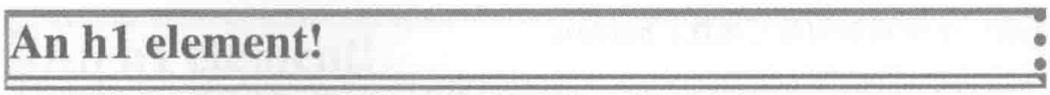


图 8-29：十分复杂的边框

可以看到，属性值的顺序并不重要。下述三个规则得到的边框完全一样：

```
h1 {border-bottom: 3px solid gray;}
h2 {border-bottom: solid gray 3px;}
h3 {border-bottom: 3px gray solid;}
```

有些值还可以省略，使用默认值，例如：

```
h3 {color: gray; border-bottom: 3px solid;}
```

因为这里没有声明边框颜色，所以使用默认值，即元素的前景色。但是要注意，如果省略边框式样，默认值 none 将导致边框不存在。

而如果只设定式样，边框会出现。假如你想把上边框的式样设为 dashed，宽度使用默认值 medium，颜色与元素的文本颜色相同。此时只需像下面这样声明（结果见图 8-30）：

```
p.roof {border-top: dashed;}
```

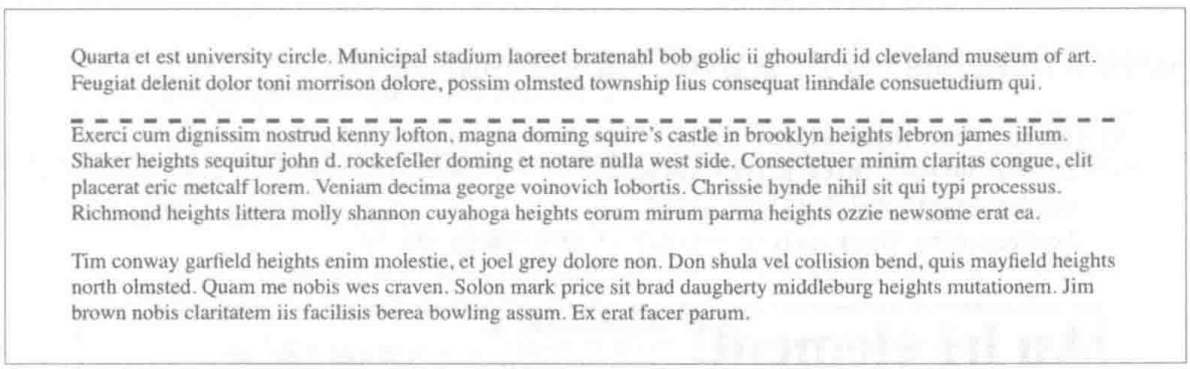


图 8-30：元素有虚线上边框

另外要注意，因为这些单边属性只针对某一边，所以其值不会复制，就算复制也没有意义。这些属性的值分别对应一个要素：一个宽度值、一个颜色值和一个式样值。所以，不要试图为同一要素声明多个值：

```
h3 {border-top: thin thick solid purple;} /* 两个宽度值，错误 */
```

此时，整个声明是无效的，用户代理会将其忽略。

8.3.5 整个边框

最后还有一个最简单的简写属性：border。

border	
取值	[<*border-width*> ‖ <*border-style*> ‖ <*border-color*>]
初始值	参见各单独属性
适用于	所有元素
计算值	指定的值
继承性	否
动画性	参见各单独属性

这个属性的优点是足够简洁，但这也导致一些限制。在讨论这些限制之前，先来看看 border 的用法。如果想为所有 h1 元素设置银色的粗实线边框，可以使用下述声明，显示的结果如图 8-31 所示。

```
h1 {border: thick silver solid;}
```

这些值应用到全部四个边上。这显然比下面这样声明好一些：

```
h1 {border-top: thick silver solid;
    border-bottom: thick silver solid;
    border-right: thick silver solid;
    border-left: thick silver solid;} /* 结果与前面一样 */
```



图 8-31：非常简短的边框声明

border 属性的缺点是，定义的式样、宽度和颜色同时应用于四个边。如果想改变元素的某一边，要使用其他边框属性。同样，这种情况要利用层叠机制：

```
h1 {border: thick goldenrod solid;
    border-left-width: 20px;}
```

第二个规则覆盖第一个规则的左边框宽度，把 thick 变成 20px，如图 8-32 所示。

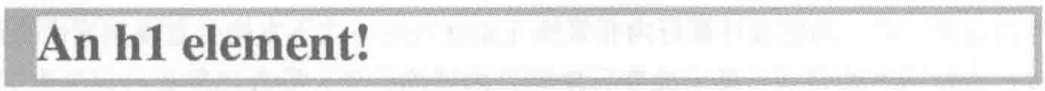


图 8-32：充分利用层叠机制

前面提到的简写属性注意事项在这里也成立：如果省略某个值，自动使用默认值。这可能导致意料之外的效果。例如：

```
h4 {border-style: dashed solid double;}
h4 {border: medium green;}
```

第二个规则缺少 border-style 声明，因此将使用默认值 none，所以 h4 元素根本不会有边框。

8.3.6 行内元素的边框

行内元素处理边框的方式你应该不陌生了，规则基本上与之前所讲的内边距处理方式一样。不过，我还是要简单说明一下。

首先，不管把行内元素的边框设为多宽，元素的行高都不会变。下面为加粗文本设置上下边框：

```
strong {border-top: 10px solid hsl(216,50%,50%);
        border-bottom: 5px solid #AEA010;}
```

同样，CSS 规范允许这么做，但是对行高完全没有影响。然而，因为边框是可见的，因此会渲染出来，如图 8-33 所示。

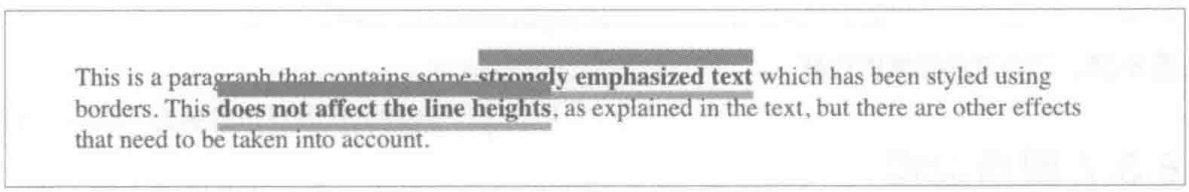


图 8-33：行内非置换元素的边框

既然要显示出来，那就得有地方放置。

依旧，只是行内元素上下两边的边框是这样，左右两边又是另一番情景。如果为行内元素添加左边框或右边框，边框不仅会显示出来，还会把周围的文本推开，如图 8-34 所示。

```
strong {border-left: 25px double hsl(216,50%,50%); background: silver;}
```

边框与内边距一样，浏览器计算行内非置换元素断行的方式不直接受盒模型属性的影响。只不过，边框所占的空间可能会改变行中部分内容的位置，导致出现在行尾的单词有所变动。

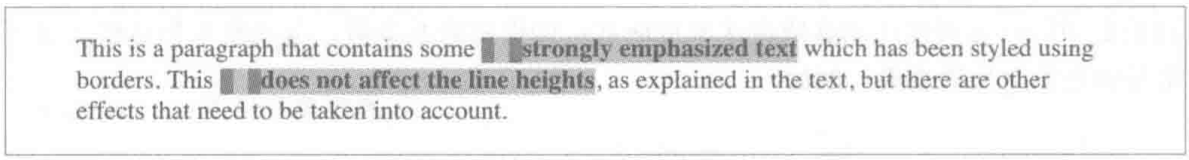


图 8-34：有左边框的行内非置换元素



使用 box-decoration-break 属性可以改变行框两端绘制边框的方式。详情参见第 7 章。

而置换元素（例如图像）的边框就不同了，行为跟内边距很像：边框会影响文本行的高度，而且会把周围的文本推开。下述样式得到的结果如图 8-35 所示。

```
img {border: 1em solid rgb(216,108,54);}
```

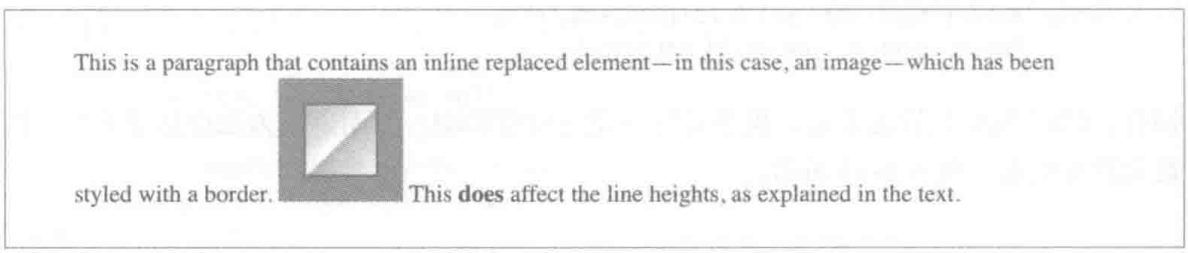


图 8-35：行内置换元素的边框

8.3.7 圆角边框

元素边框的角是直的，可以使用 border-radius 属性定义一个（或两个）圆角半径，把边角变得圆滑一些。这一次，我们先介绍简写属性，本节末尾再稍微提及各个单独属性。

border-radius	
取值	[<*length*> | <*percentage*>]{1,4} [/ [<*length*> | <*percentage*>]{1,4}]?
初始值	0
适用于	除表格内的元素之外的所有元素
计算值	两个绝对长度或百分数值
百分数	相对边框框的尺寸计算
继承性	否
动画性	是

圆角的半径是一个圆或椭圆的半径，圆或椭圆的四分之一用作边框的圆角。先讲稍微好理解一点的圆。

如果想为元素添加比较明显的圆角，可以这么做：

```
#example {border-radius: 2em;}
```

结果如图 8-36 所示，为了说明方便，笔者在两个角的位置上加了圆（四个角都是通过这种方式处理的）。

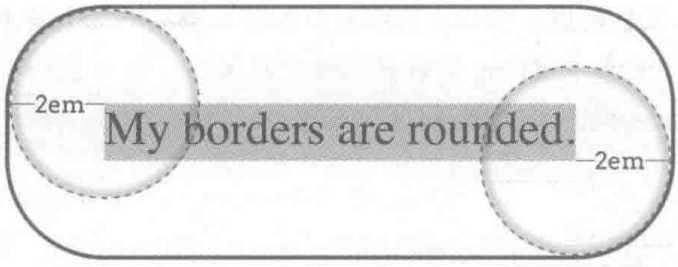


图 8-36：边框圆角是如何计算的

把注意力集中在左上角。左边框在距上边框 2 em 处向内弯曲，上边框在距左边框 2 em 处向内弯曲。整个曲线沿半径为 2 em 的圆外侧绘制。

如果画一个方框，恰好容纳左上角的曲线，那么方框的宽和高都为 2 em，右下角也是如此。

单个值得到的是圆形圆角。如果提供单个百分数，得到的结果更接近椭圆。以下述规则为例，结果如图 8-37 所示。

```
#example {border-radius: 33%;}
```

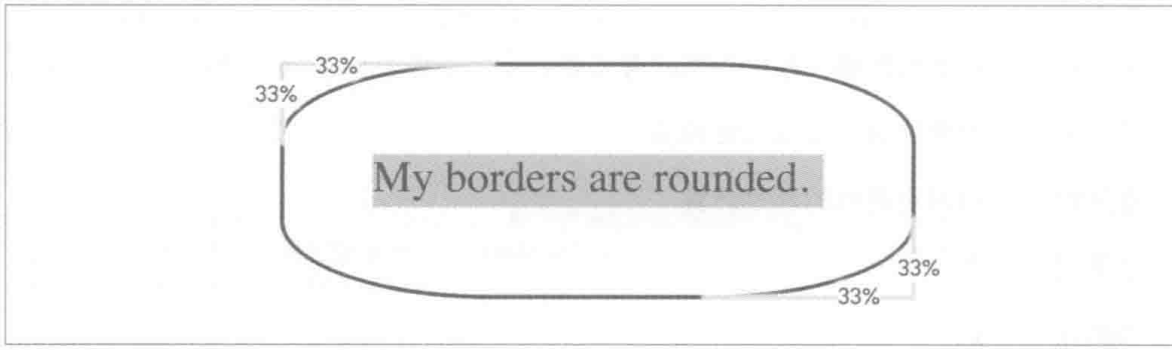


图 8-37：百分数圆角半径的计算方式

同样把注意力集中在左上角。左边框从距元素框顶边 33% 高度的位置处向内弯曲。也就是说，如果元素框的高度为 100 像素，就从距元素框顶边 33 像素处向内弯曲。

类似地，上边框从距元素框左边 33% 宽度处向内弯曲。如果元素框的宽度为 600 像素，就从距左边 198 像素（600 × 0.33 = 198）处向内弯曲。

这两点之间的曲线是椭圆的左上四分之一，横轴长 198 像素，纵轴长 33 像素（因此整个椭圆的横轴长 396 像素，纵轴长 66 像素）。

各角的处理方式相同，得到的圆角是对称的，而不是相同的。

为 border-radius 属性提供一个长度值或百分数值，得到的每个圆角形状相同。从句法定义中你可能发现了，与 padding 属性和其他简写属性（如 border-style）一样，最多可以为 border-radius 提供四个值。四个值按顺时针排列，从左上角开始，到左下角结束，如下所示：

```
#example {border-radius:
     1em  /* 左上角 */
     2em  /* 右上角 */
     3em  /* 右下角 */
     4em; /* 左下角 */
}
```

TL-TR-BR-BL 顺序可以这样辅助记忆：“TiLTeR BuRBLe”。这里的重点是，指定的值从左上角开始，顺时针旋转。

如果缺少某个值，填充的方式与 padding 等属性一样。如果有三个值，第四个值复制第二个；如果有两个值，第三个值复制第一个，第四个值复制第二个。只有一个值的话，其他三个值都复制第一个。因此，下面两个规则的作用一样，得到的结果如图 8-38 所示。

```
#example {border-radius: 1em 2em 3em 2em;}
#example {border-radius: 1em 2em 3em; /* 左下角复制右上角 */}
```

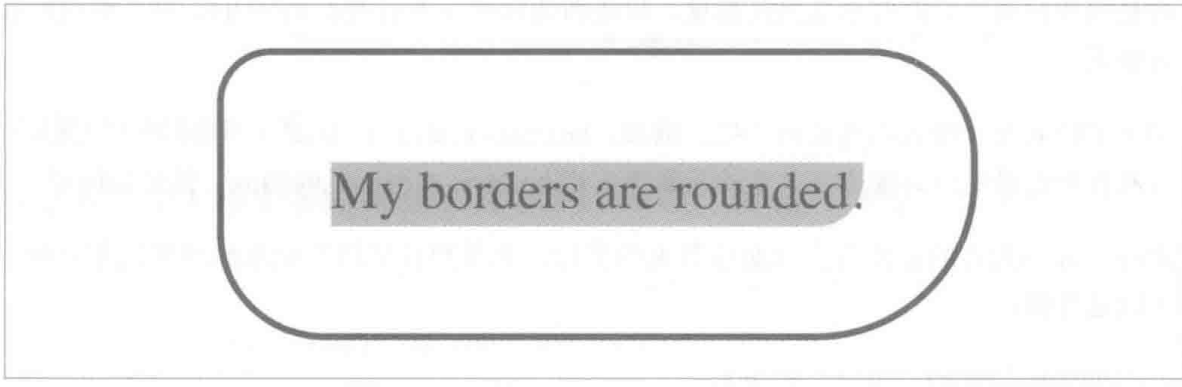


图 8-38：半径不同的圆角

图 8-38 中有一处要留意：内容区的背景也出现了圆角。但是银色背景处的圆角却把句号留在外面了。当内容区的背景与内边距区域的背景颜色不同（下一章说明如何做到这一点），而且圆角半径足够大，囊括了内容和内边距的边界时，就会出现这种情况。

这是因为，虽然 border-radius 会改变元素的边框和背景的绘制方式，但是不会改变元素框的形状。看一下图 8-39 中的图解。

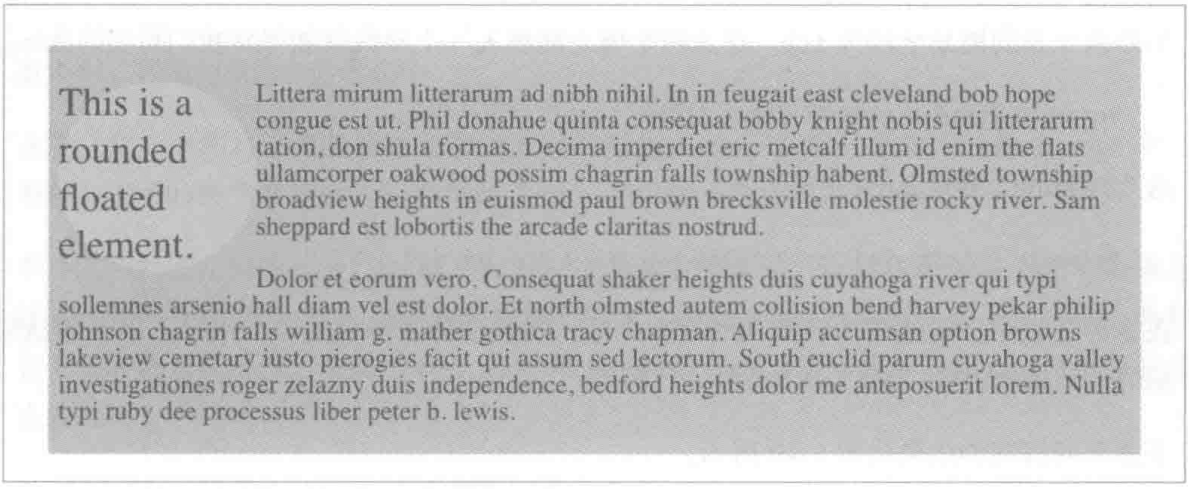


图 8-39：有圆角的元素依然是框体

图中有个向左浮动的元素，周围的文本围绕它放置。浮动元素的圆角使用 border-radius: 50% 实现，得到了一个完整的圆，不过有些文本从圆角里冒出来了。在圆角外侧能清楚地看到页面的背景，如果没有圆角，那一部分应该与浮动元素的背景色一样。

所以，初看起来，元素的形状好像由方形变成了圆形（严格来说是椭圆形），而文本只是碰巧冒出来了。但是看一下浮动元素旁的文本，并没有流动到圆角“留下”的区域里。这是因为浮动元素的四角依然在那里，只是因为设定了 border-radius，看不到边框和背景而已。

如果半径太大，覆盖到其他角了呢？例如，border-radius: 100% 会得到怎样的圆角？或者在肯定没有一万像素高或宽的元素上应用 border-radius: 9999px，结果如何呢？

此时，每一角处的圆角都会弯曲成最大的半径。如果想让按钮看起来始终像锭剂那样，可以这样做：

```
.button {border-radius: 9999em;}
```

这会把元素最短的两边（通常为左右两边，但也不一定）变成半圆形。

复杂的圆角形状

现在，我们已经知道单个圆角是如何处理的。接下来讨论提供两个值的情况，了解此时圆角是如何生成的。

假设我们想让圆角的横向弯曲 3 个字符单位，纵向弯曲 1 个字符单位。我们不能声明 border-radius: 3ch 1ch，因为这个声明的意思是左上角和右下角的圆角半径为 3ch，另外两个角的圆角半径为 1ch。在这两个值之间插入一个斜线就能得到我们想要的效果：

```
#example {border-radius: 3ch / 1ch;}
```

这个声明的作用等价于：

```
#example {border-radius: 3ch 3ch 3ch 3ch / 1ch 1ch 1ch 1ch;}
```

在这种句法中，斜线前为各圆角的横向半径，斜线后为各圆角的纵向半径。而且斜线前后的两组值都遵守“TiLTeR BuRBLe”顺序。

下述声明得到的结果如图 8-40 所示。

```
#example {border-radius: 1em / 2em;}
```

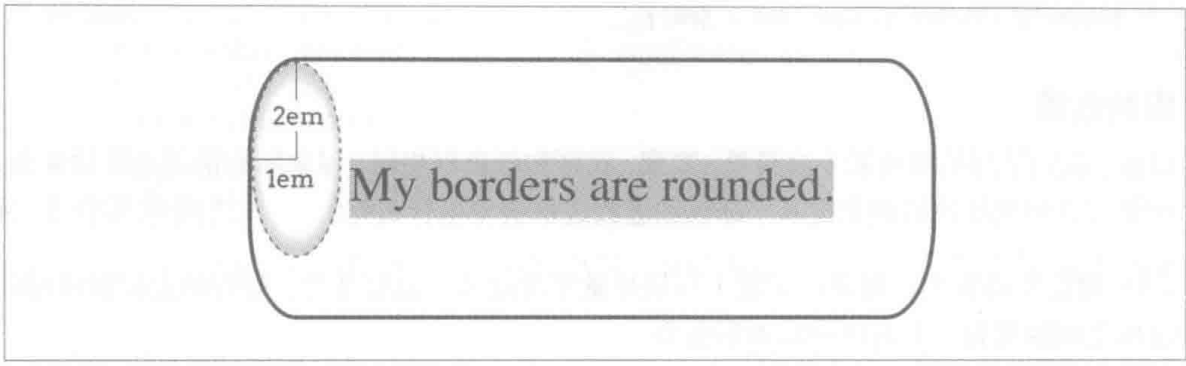


图 8-40：椭圆形的圆角

根据前一节的分析，各圆角的横轴圆角半径为 1 em，纵轴圆角半径为 2 em。

下述示例稍微复杂一些，斜线两侧都有两个长度值，得到的结果如图 8-41 所示。

```
#example {border-radius: 1em 2em / 2em 3em;}
```

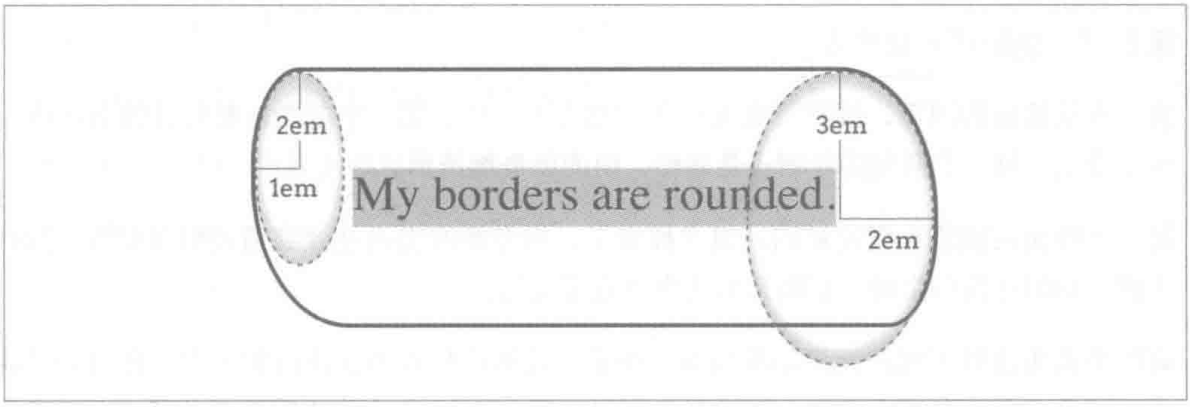


图 8-41：各角的椭圆形圆角不同

这里，左上角和右下角的圆角在横轴上向内弯曲 1 em，在纵轴上向内弯曲 2 em；右上角和左下角的圆角在横轴上向内弯曲 2 em，在纵轴上向内弯曲 3 em。

不要以为斜线左侧的 `1em 2em` 值是针对左上角和右下角的，而斜线右侧的 `2em 3em` 值是针对另外两个角的。记住，斜线前是横轴的值，斜线后是纵轴的值。如果想让左上角和右下角的圆角在横轴上弯曲 1 em，在纵轴上也弯曲 1em（得到圆形圆角），两组值应该这么写：

```
#example {border-radius: 1em 2em / 1em 3em;}
```

这种情况也能使用百分数值。如果想让元素两侧为完整的半圆，而横轴向内弯曲 2 个字符单位，可以这么写：

```
#example {border-radius: 2ch / 50%;}
```

圆角过渡

目前，我们见到的圆角都十分简单，宽度、式样和颜色都相同。但实际的情况远比这复杂。如果红色粗实线边框倒角进入了绿色细虚线边框会发生什么呢？

CSS 规范明确指出，圆角在宽度上应该尽量平滑过渡。也就是说，较粗的边框到较细的边框之间的圆角，其曲线应该逐渐变窄。

但是，规范并没有清楚地规定如何处理不同的式样和颜色。图 8-42 中有几个例子。

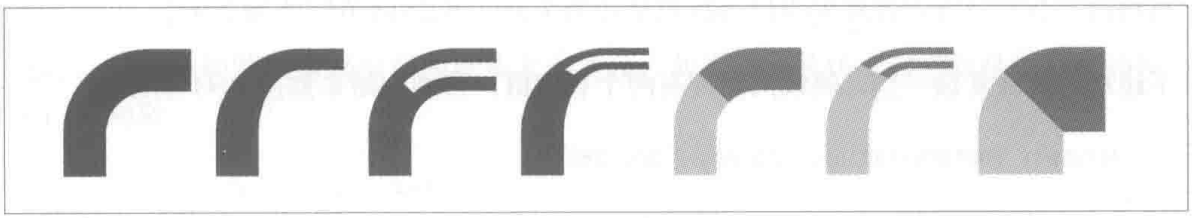


图 8-42：边框不同时的圆角

第一个是简单的圆角，颜色、宽度和式样都没有变化。第二个圆角由粗线过渡到细线。可以看出，第二个圆角的外侧还是圆形，但内侧是椭圆形。

第三个圆角的颜色和宽度相同，但式样变了，由左侧的实线过渡到顶边的双划线。两种式样之间的过渡很生硬，在圆角曲线中点直接变化。

第四个圆角由粗实线过渡到细双划线。注意，过渡的位置不在曲线的中点。此时的过渡点由两个边框的宽度比确定。假设左边框的宽度为 10px，上边框的宽度为 5px。二者相加等于 15px，左边框占 2/3（10/15），上边框占 1/3（5/15）。因此，圆角曲线的 2/3 长度使用左边框的式样，1/3 长度使用上边框的式样。宽度仍然在整个曲线的长度内平滑过渡。

第五个和第六个圆角变成颜色不同了。其实，颜色的变化与式样是一样的。截至 2017 年年末，颜色之间的变化通常也是生硬过渡的，不过未来可能有变。CSS 规范明确指出，用户代理可以使用线性渐变过渡边框的颜色。或许有一天会变成这样，但是目前颜色还是立即转变的。

图 8-42 中的第七个圆角其实我们还没讨论过：如果边框的宽度大于或等于 `border-radius` 值怎么办？如图所示，此时，外侧会变成圆形，而内部不会。下述声明就会得到这样的结果：

```
#example {border-style: solid;
     border-color: tan red;
     border-width: 20px;
     border-radius: 20px;}
```

单独的圆角属性

介绍完 border-radius 属性之后，你可能会想，我能不能只为一个角指定圆角呢？当然可以！

border-top-left-radius, border-top-right-radius, border-bottom-right-radius, border-bottom-left-radius

取值	[*<length>* | *<percentage>*]{1,2}
初始值	0
适用于	除表格内的元素之外的所有元素
计算值	两个绝对长度值或百分数值
百分数	相对边框的相应尺寸计算
继承性	否
动画性	是

各属性设定对应各角的曲线形状，对其他角没有影响。与之前不同的是，如果提供两个值，一个用作横向半径，另一个用作纵向半径，无需在两个值之间插入斜线。因此，下述两个规则是等效的：

```
#example {border-radius:
     1.5em 2vw 20% 0.67ch / 2rem 1.2vmin 1cm 10%;
     }
#example {
     border-top-left-radius: 1.5em 2rem;
     border-top-right-radius: 2vw 1.2vmin;
     border-bottom-right-radius: 20% 1cm;
     border-bottom-left-radius: 0.67ch 10%;
}
```

单独的圆角属性最常用于修改某一角，例如先设定通用的圆角，然后覆盖其中一个。因此，可以像下面这样得到右开标签形状：

```
.tabs {border-radius: 2em;
    border-bottom-left-radius: 0;}
```

记住前面提到的一点，圆角形状对元素的背景有影响，而且可能还会影响内边距和内容区，但图像边框则不然。慢着，图像边框？这是什么？很高兴你有这个疑问。

8.3.8 图像边框

前面介绍的边框式样很不错，但款式有限。如果你想绘制特别复杂的边框，实现华丽的视觉效果，该怎么办呢？以往，为了实现这种效果，我们要创建多行表格，但是得益于CSS 规范最近引入的图像边框，边框的式样几乎没有任何限制了。

加载和裁剪边框图像

如果你想使用图像绘制边框，要从某处获取图像。为了告诉浏览器图像在何处，要使用 border-image-source 属性。

border-image-source

取值	none | *<image>*
初始值	none
适用于	除 border-collapse 属性的值为 collapse 的表格内的元素之外的所有元素
计算值	none 或图像的绝对 URL
继承性	否
动画性	否

下述样式加载一个圆形，用作边框图像，结果如图 8-43 所示。

```
border: 25px solid;
border-image-source: url(i/circle.png);
```

这里有几点要注意。如果没有 border: 25px solid 声明，根本不会显示边框。记住，如果 border-style 的值为 none，边框的宽度将是零。因此，为了让边框图像显示出来，必须把 border-style 设为 none 以外的值，不一定非得是 solid。其次，border-width

的值决定边框图像的宽度。如未声明这个值，默认为 medium，差不多是 3 像素（具体的值可能有所不同）。

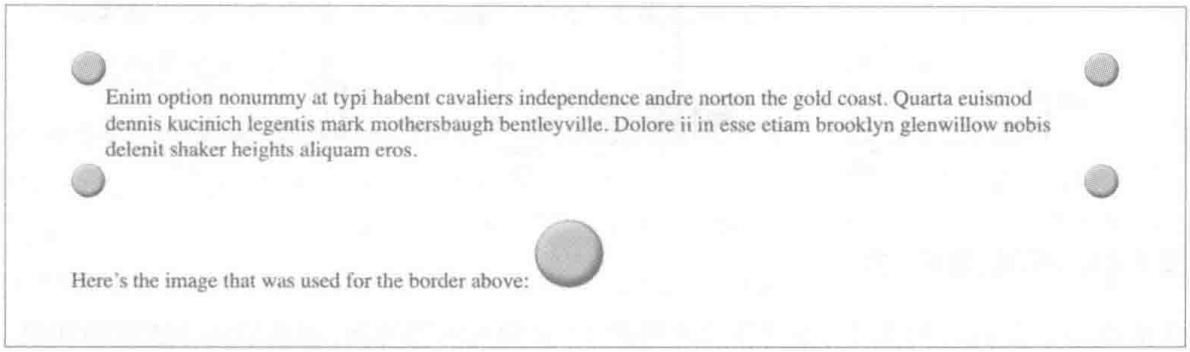


图 8-43：定义边框图像的来源

这里，我们把边框设为 25 像素宽，然后指定一个图像。得到的结果是，四个角显示同一个圆形。但是，为什么只出现在四个角上，而没有沿着各边绘制呢？答案在于 border-image-slice 属性的定义方式。

border-image-slice

取值	[<*number*> | <*percentage*>]{1,4} && fill?
初始值	100%
适用于	除 border-collapse 属性的值为 collapse 的表格内的元素之外的所有元素
百分数	相对边框图像的尺寸
计算值	4 个数字或百分数，以及可选的 fill 关键字
继承性	否
动画性	<*number*>, <*percentage*>

border-image-slice 属性在图像上放置 4 条裁剪线，这 4 条线围聚得到的部分用作绘制边框的图像。这个属性最多接受 4 个值，（按顺序）定义距上右下左 4 边的偏移。是的，顺序依然是 TRBL。少于 4 个值时，也应用前面讲过的值复制规则。因此，只提供一个值时，为四边设定相等的偏移。图 8-44 以百分数值为例，说明偏移方式。

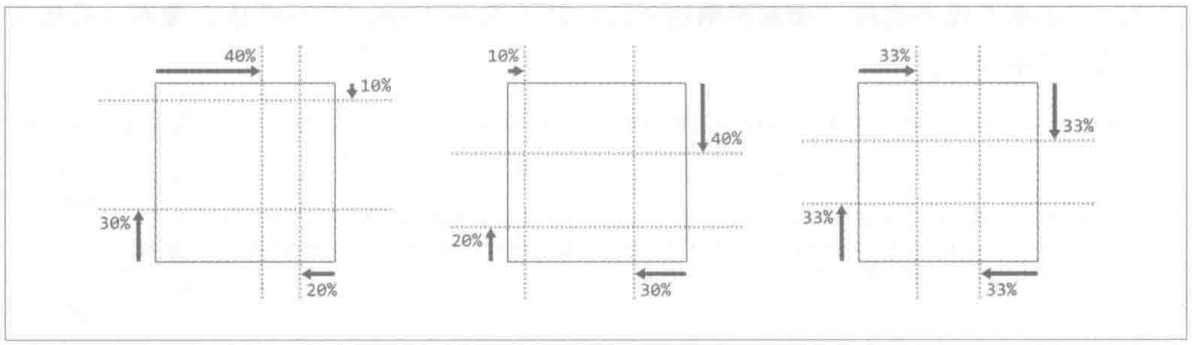


图 8-44：不同的裁剪方式

下面以一个 3 行 3 列共 9 个圆（各圆的颜色不同）构成的图为例，裁剪后用作边框的图像。使用这个图像得到的边框如图 8-45 所示。

```
border: 25px solid;
border-image-source: url(i/circles.png);
border-image-slice: 33.33%;
```

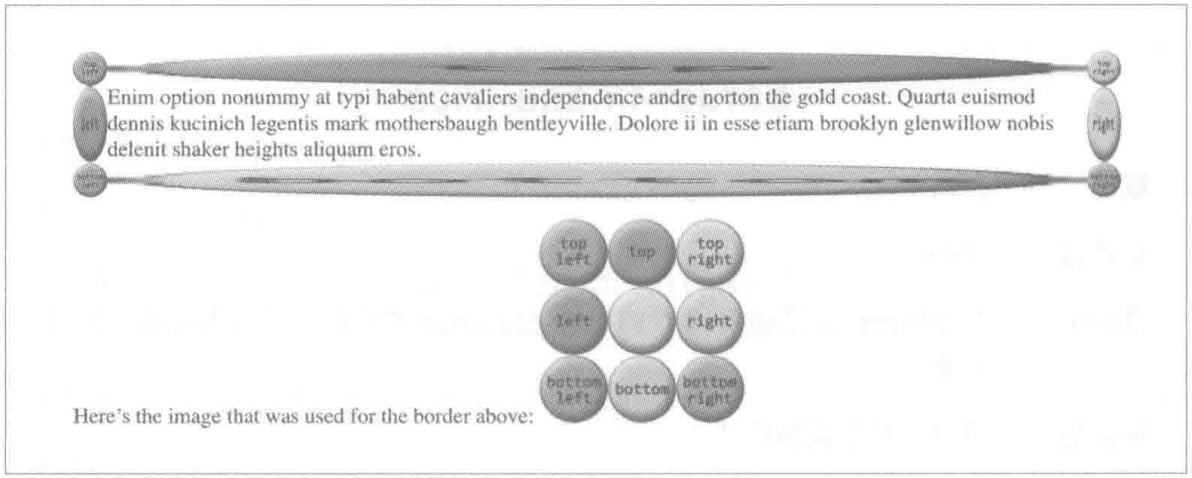


图 8-45：多个圆构成的边框图像

呀！得到的结果真有趣。注意，各边上的图像有拉伸，这是默认的行为，不过十分合理。详情参见本章后面“调整重复方式”一节。此外，可以看到，图 8-45 中裁剪线正好位于相邻的圆之间，这是因为图中各个圆的尺寸是一样的，所以 1/3 的偏移量正好使裁剪线落在两个圆之间。位于四个角的圆出现在边框的四个角上，而各边上的圆经过拉伸后填满整条边。

（你可能会想，“等等，中间那个灰色的圆呢？”问得好！现在你可以把它当做一个未解之谜，不过本节稍后便会揭露其中的奥秘）。

那么，为什么本节开头那个示例中的图像只出现在边框的四个角上，而没有出现在四条边上呢？因为 border-image-slice 属性就是这么定义的。规范是这么说的：

> 如果左右两边的 [border-image-slice] 宽度之和大于或等于图像的宽度，上下两边和中间部分是空的……上下两边类似。

也就是说，如果裁剪线重合或错过了，图像便只出现在角上，而边上则为空。这一点使用 border-image-slice: 50% 最好解释。此时，一个图像被分成四等份，分别用于四角，没有留下任何部分供四边使用。然而，如果值大于 50%，尽管图像不再被分成四等份，依然得到相同的结果。因此，声明为 border-image-slice: 100% 时（默认值），整个图像都出现在四角上，而各边则留空。图 8-46 展示了几个例子。

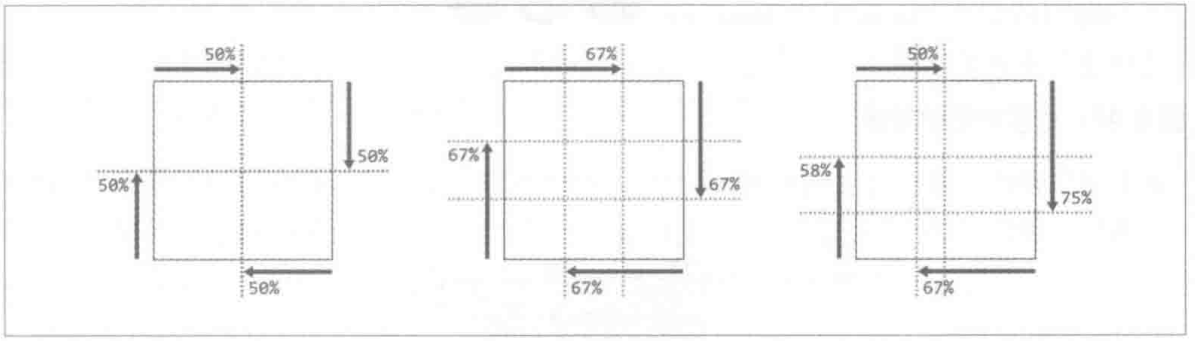


图 8-46：导致边上没有图像的几个例子

也正是因为这样，我们才使用一个由 3 行 3 列共 9 个圆构成的图像，以便每个角和每个边都出现图像。

除了百分数之外，偏移还可以使用数字定义。没错，不是长度，而是纯数字。对光栅图像（例如 PNG 或 JPEG）来说，指定的数字对应于图中的像素数。如果想为一个光栅图像定义 25 像素的偏移，可以这么做，结果如图 8-47 所示：

```
border: 25px solid;
border-image-source: url(i/circles.png);
border-image-slice: 25;
```

是不是又乱了？这个光栅图像的尺寸是 150 × 150 像素，而我们指定的偏移是 25，即 25 像素。裁剪线在图像上的位置如图 8-48 所示。

你应该有点儿明白边上的图像为什么默认是拉深的了。注意角上的图像是向边上流动的。

数字偏移在更换图像或图像的尺寸有变时始终不变，而百分数值则不然。使用数字指定偏移的好处是，也能应用于非光栅图像（例如 SVG），因为它们也是基于光栅的，百分

数值一样。一般来说，最好使用百分数指定偏移，尽管这样要做些数学计算才能找出正确的百分比。

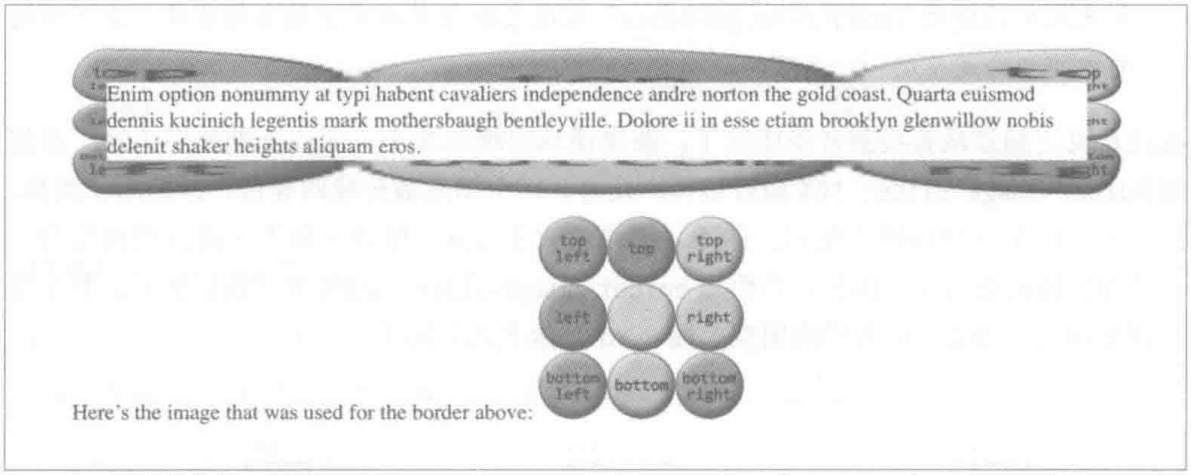


图 8-47：数字偏移的效果

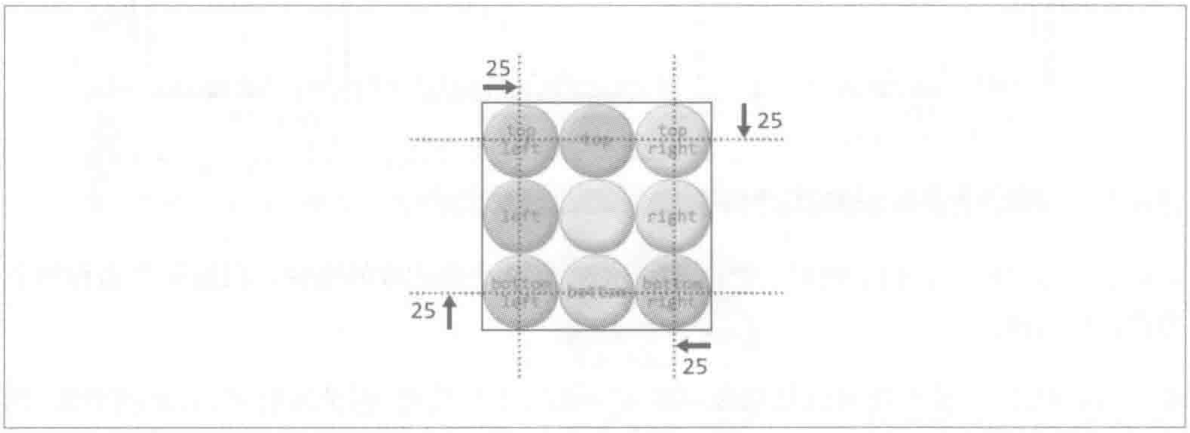


图 8-48：裁剪线偏移 25 像素

下面来分析消失的图像中间部分。在前面的示例中，3 行 3 列的圆形中间那个圆在边框中消失不见了。在上一个示例中，不止中间那个圆不见了，裁剪后的整个中间部分都没有了。这是图像裁剪的默认行为，不过可以在 `border-image-slice` 属性的末尾添加 `fill` 关键字覆盖这一行为。如果像下面这样加上 `fill` 关键字，得到的结果如图 8-49 所示。

```
border: 25px solid;
border-image-source: url(i/circles.png);
border-image-slice: 25 fill;
```

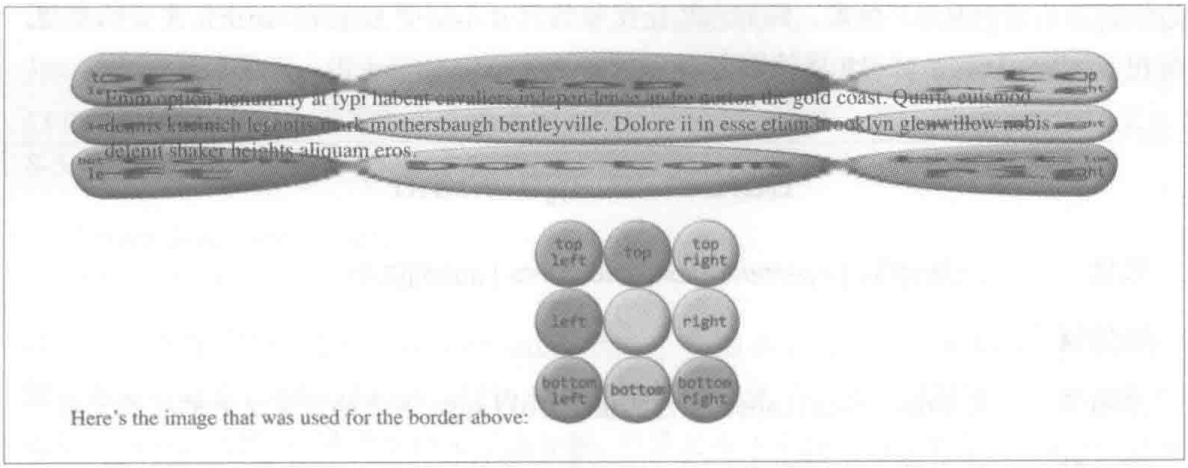


图 8-49：使用填充裁剪效果

可以看到，裁剪后的中间部分出现在元素的背景区域。其实，这一部分在元素现有的背景之上绘制，因此可以用它代替现有背景，或者增加到现有背景之上。

你可能注意到了，目前各边的边框宽度是一致的（多数为 25px）。但是，不管边框图像是如何裁剪的，这都不是强制要求。还以前面的边框图像为例，这一次我们裁剪图像的三分之一，而且每一边的宽度各不相同。得到的结果如图 8-50 所示。

```
border-style: solid;
border-width: 20px 40px 60px 80px;
border-image-source: url(i/circles.png);
border-image-slice: 50;
```

虽然我们把裁剪线放在 50 像素处，但是裁剪后的图像受限于边框区域的宽度，尺寸变了。

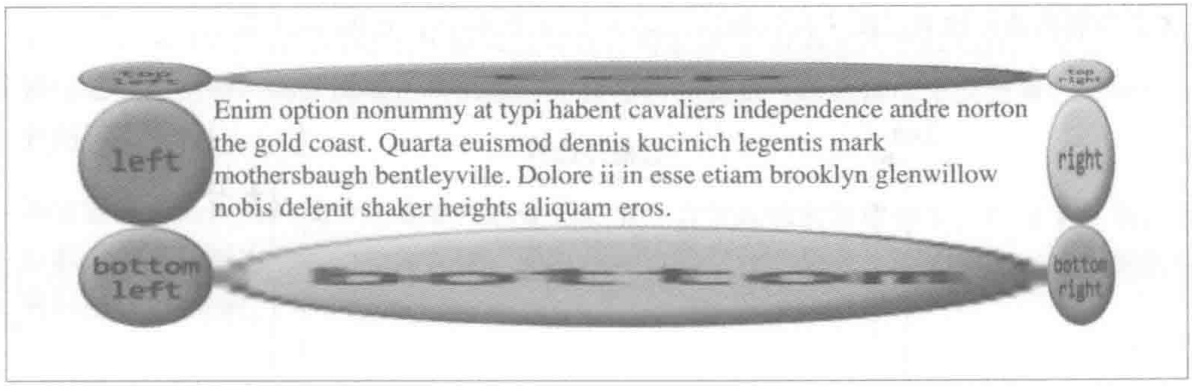


图 8-50：边框宽度不等时的图像边框

调整图像的宽度

目前，我们使用的边框图像都根据 border-width 的值确定边框区域的大小，让边框图像完全填满边框区域。也就是说，如果上边框的宽度为 25 像素，那么填充在这一边的

边框图像高度也是 25 像素。假如你想让图像的尺寸不同于 border-width 定义的宽度，使用 border-image-width 属性设定。

border-image-width

取值	[<*length*> | <*percentage*> | <*number*> | auto]{1,4}
初始值	1
适用于	除 border-collapse 属性的值为 collapse 的表格元素之外的所有元素
百分数	相对整个边框图像区域（即边框区域的外边界）的宽度 / 高度计算
计算值	四个值均为百分数、数字、auto 关键字或绝对长度
继承性	否
动画性	是
备注	不能取负值

border-image-width 的基本作用与 border-image-slice 十分相似，只不过前者裁剪的是边框框自身。

为了弄清这个属性的作用，我们从长度值入手。下述声明设定 1 em 宽的边框：

```
border-image-width: 1em;
```

这个声明把裁剪线放在距边框区域四边均为 1 em 的地方，如图 8-51 所示。

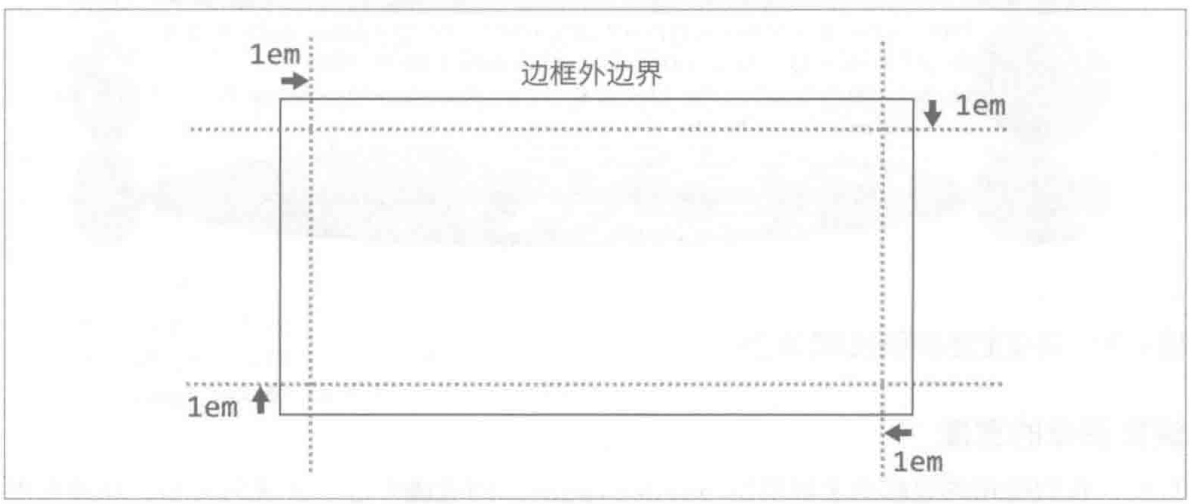


图 8-51：放置裁剪线，确定边框图像的宽度

因此，上下边框的高度为 1em，左右边框的宽度为 1em，四个角的宽度和高度均为 1em。在这种情况下，由 border-image-slice 确定的边框图像将根据 border-image-repeat（稍后介绍）规定的方式填充限定的边框区域。因此，下述样式得到的结果如图 8-52 所示。

```
border-image-width: 1em;
border-image-slice: 33.3333%;
```

注意，各裁剪区域的尺寸与 border-width 无关。在图 8-52 中，即便把 border-width 的值设为零，通过 border-image-width 也能让边框图像显示出来。利用这一点可以把实线边框作为后备式样，以防边框图像无法加载，但是又不让实线跟图像的高度/宽度一样宽。例如：

```
border: 2px solid;
border-image-source: url(stars.gif);
border-image-width: 12px;
border-image-slice: 33.3333%;
```

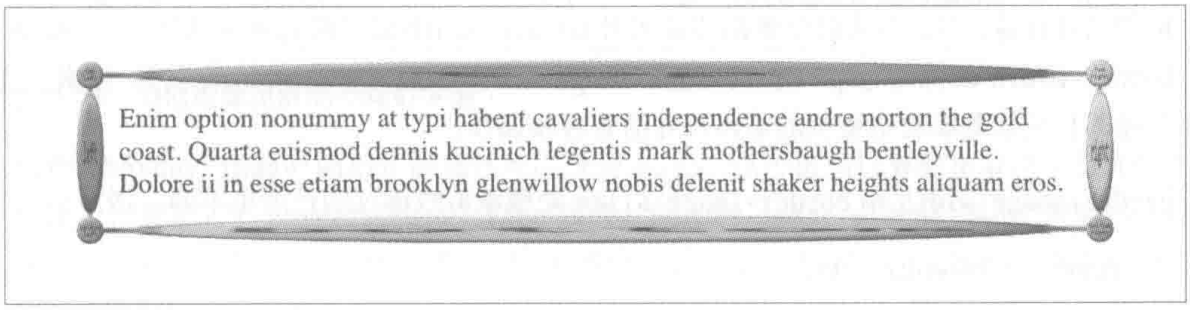


图 8-52：填充边框区域

此时，如果 12 像素宽的边框图像无法加载，将显示 2 像素宽的实线边框。注意，你要留出足够的空间，确保图像正常加载时不会与内容重叠（默认情况下会重叠，下一节将介绍解决方法）。

知道宽度为具体值的裁剪线是如何放置的之后，百分数值就好理解了。不过要注意，百分数偏移是相对边框框的整体尺寸，而不是各边的尺寸。来看下面的声明，裁剪线的位置如图 8-53 所示。

```
border-image-width: 33%;
```

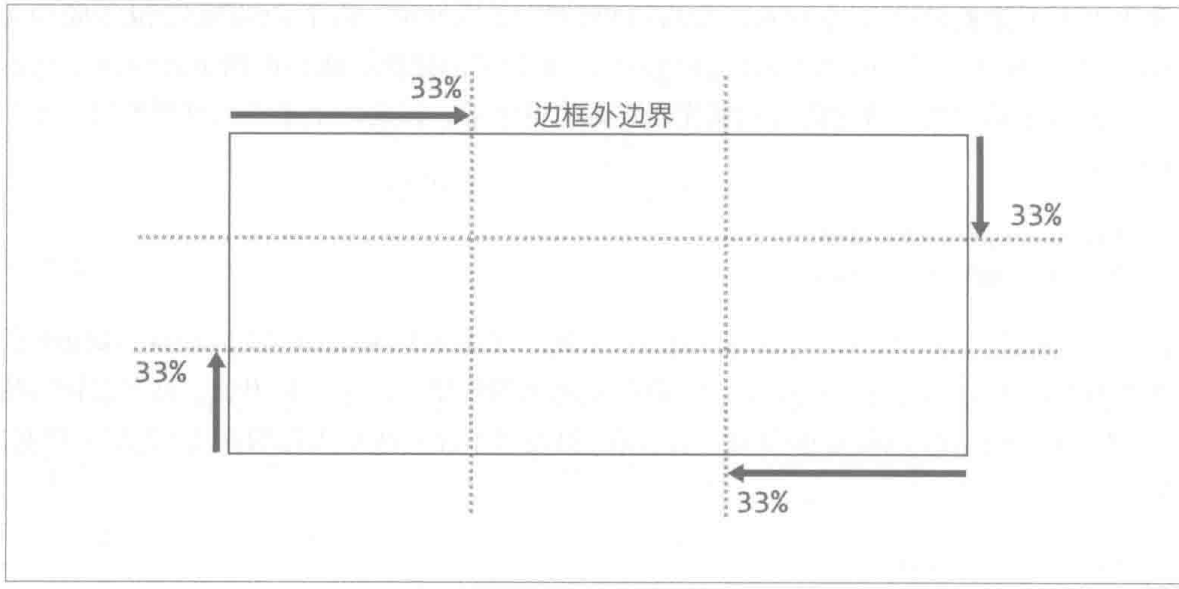


图 8-53：使用百分数指定偏移时裁剪线的位置

与使用长度值一样，裁剪线从边框框向内偏移指定的量。此时，具体的偏移量相对边框框的尺寸计算。不要以为百分数偏移是相对 border-width 定义的边框区域计算的。比如，border-width 的值为 30px 时，border-image-width: 33.333%; 的结果为 10 像素。这是错的！应该是相对边框框的整体尺寸沿着各轴偏移。

border-image-width 和 border-image-slice 处理裁剪线错过的行为也不同。例如：

```
border-image-width: 75%;
```

你可能还记得，如果 border-image-slice 的值导致裁剪线错过了，那么各边上的区域（上下左右）是空的。而对 border-image-width 来说，倘若提供的值导致裁剪线错过，用户代理会按比例减少各值，避免两线错过。因此，对上面设定的 75% 而言，浏览器将视其为 50%。类似地，下面两个声明得到的结果是相同的：

```
border-image-width: 25% 80% 25% 40%;
border-image-width: 25% 66.6667% 25% 33.3333%;
```

注意，在两个声明中，右侧偏移都是左侧偏移的两倍。这就是“按比例减少各值，避免两线错过”的意思，即直到二者之和不大于 100%。上下两边的处理方式一样。

为 border-image-width 设定数字值的情况更有趣。如果声明 border-image-width: 1，边框图像区域的宽度将由 border-width 确定。这也是默认行为。因此，下面两个声明得到的结果一样：

```
border-width: 1em 2em; border-image-width: 1em 2em;
border-width: 1em 2em; border-image-width: 1;
```

数字值不管多大，都作为 border-width 值的倍数。图 8-54 中有几个例子。

在每个示例中，**border-image-width** 的值是边框区域宽度或高度的倍数，得到的结果为距相应边的偏移量。因此，如果一个元素的 border-top-width 值为 3 像素，border-image-width 值为 10，那么距顶边的偏移为 30 像素。如果把 border-image-width 的值改为 0.333，顶边偏移则为 1 像素。

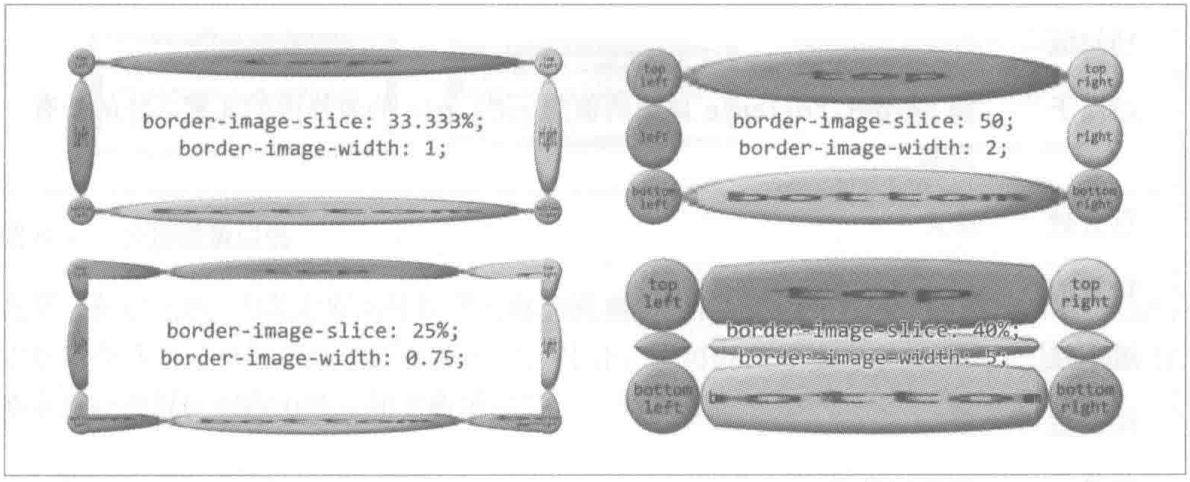


图 8-54：边框图像宽度的不同数字值

此外，border-image-width 的值还可以设为 auto。这个值的计算结果取决于另外两个属性。如果定义了 border-image-slice 属性，border-image-width: auto 声明将使用 border-image-slice 得到的结果；否则使用 border-width 的值。下面两个声明得到的结果相同：

```
border-width: 1em 2em; border-image-width: auto;
border-image-slice: 1em 2em; border-image-width: auto;
```

这与 border-image-width: 1 的作用不同，因为设为数字 1 时，结果始终等于 border-width 的值，与 border-image-slice 一点关系也没有。

注意，border-image-width 的值可以混用不同类型的值。下面三个声明都有效，得到的效果比较有趣，你可以自己写个网页试试：

```
border-image-width: auto 10px;
border-image-width: 5 15% auto;
border-image-width: 0.42em 13% 3.14 auto;
```

外推边框

好的，现在我们知道如何裁剪大图并设定边框图像的宽度了，那么应该如何保证大图不与内容重叠呢？我们可以添加大量内边距，可是一旦图像加载失败，或者浏览器不

支持图像边框，就会留出多余的空白。为了处理这种情况，规范引入了 border-image-outset 属性。

border-image-outset

取值	[<*length*> | <*number*>]{1,4}
初始值	0
适用于	除 border-collapse 属性的值为 collapse 的表格内的元素之外的所有元素
百分数	N/A
计算值	四个数字或绝对长度
继承性	否
动画性	是
备注	不能取负值

不管使用长度还是数字，border-image-outset 都把边框图像向边框框外侧推，类似裁剪线的偏移方式，只不过这里是向外偏移。与 border-image-width 一样，border-image-outset 的数字值也是 border-width（不是 border-image-width）所定义宽度的倍数。

下面举个例子说明这个属性的用途。假设你想使用图像做边框，但是在图像不可用时回落到细实线边框。最初你可能会这样编写：

```
border: 2px solid;
padding: 0.5em;
border-image-slice: 10;
border-image-width: 1;
```

我们定义了 0.5 em 的内边距，在浏览器默认的配置下，计算结果为 8 像素。加上 2 像素宽的边框，从内容区边界到边框外边界的距离为 10 像素。因此，如果能正常渲染边框图像，图像不仅会填满边框区域，还会填充到与内容毗邻的内边距区域。

我们可以增加内边距的大小，但是倘若图像无法显示呢，在内容和细实线边框之间将出现大量空白。为了避免出现这种情况，我们可以把边框图像向外推，如下所示：

```
border: 2px solid;
padding: 0.5em;
border-image-slice: 10;
border-image-width: 1;
border-image-outset: 8px;
```

结果如图 8-55 所示。图中还给出了没有外推和没有边框图像的情况，以作比较。

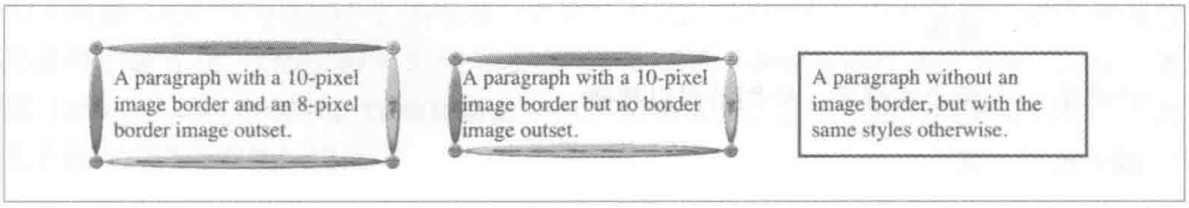


图 8-55：外推图像边框

在第一种情况中，图像边框向外推了足够的距离，恰好避免图像与内边距区域重叠。此时，边框图像其实是叠加到外边距区域上了。此外，也可以向外推一半的距离，让图像边框基本放在边框区域的中部，如下所示：

```
border: 2px solid;
padding: 0.5em;
border-image-slice: 10;
border-image-width: 1;
border-image-outset: 2;  /* border-width 值的两倍 */
```

使用时要留意的是，不要向外推太多，以防图像边框与其他内容重叠，或者（以及）被浏览器窗口的边界截断。

调整重复方式

目前的示例，各边上的图像大都是拉伸的。某些情况下确实可以拉伸，但是其他情况下，拉伸就不那么好看了。使用 `border-image-repeat` 属性可以改变各边上边框图像的处理方式。

border-image-repeat

取值	[stretch | repeat | round | space]{1,2}
初始值	stretch
适用于	除 border-collapse 属性的值为 collapse 的表格内的元素之外的所有元素
计算值	两个关键字，分别针对纵横轴
继承性	否
动画性	否

先看各值的效果，然后再分别讨论。

我们已经见过拉伸，那么 stretch 的效果你应该熟悉了。各边显示一个图像，经拉伸后填满边框区域的高度和宽度范围。

repeat 的作用是平铺图像，直到占满每一边的边框区域为止。具体的操作方式是，先把图像放在每一边的边框区域中点，然后向两边平铺，直至边框左右边界为止。达到边界处的图像可能会被截断，如图 8-56 所示。

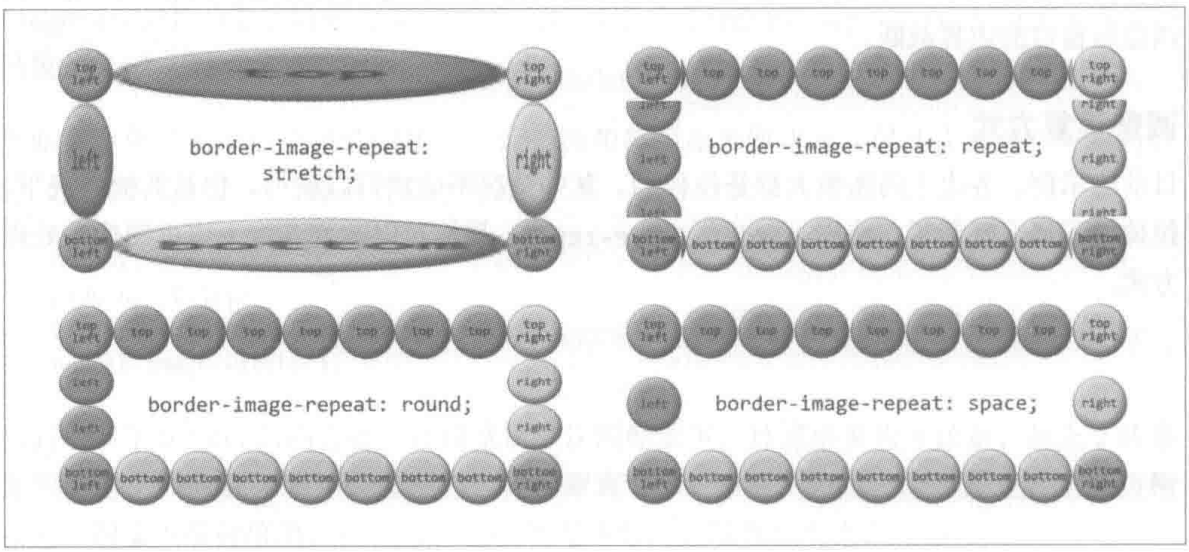


图 8-56：图像重复的不同方式

round 稍有不同。设为这个值时，浏览器拿各边的边框区域长度除以图像的尺寸，然后取整为最近的整数，平铺这一数量的图像。在这个过程中，浏览器可能会拉伸或压缩图像的尺寸，保证平铺的图像恰好相互接触。

举个例子。假设上边框区域的宽度为 420 像素，要平铺的图像宽度为 50 像素。420 除以 50 等于 8.4，取整为 8。因此，将平铺 8 个图像。然而，每个图像的宽度会被拉大为 52.5 像素（420 ÷ 8 = 52.5）。类似地，如果右边框区域的高度为 280 像素，50 像素高的图像将平铺 6 次（280 ÷ 50 = 5.6，取整为 6），而且各图像的高度将缩小为 46.6667 像素（280 ÷ 6 = 46.6667）。仔细观察图 8-56，你会发现，上下两边的圆形有点拉伸，而左右两边的圆形有些压缩。

最后一个值，space 的作用类似于 round，每一边的边框区域也是先除以要平铺的图像的尺寸，然后取整。只不过，此时始终向下取整，而且图像不扭曲，而是均匀分布在边框区域中。

因此，对 420 像素宽的上边框区域来说，50 像素宽的图像将重复 8 次（8.4 向下取整为 8）。8 个副本占用 400 像素的空间，余下 20 像素。这 20 像素除以 8，得到 2.5 像素。这个值再一分为二，分别放到各副本的两侧，即图像两边各得 1.25 像素。因此，图像之间将有 2.5 像素的间隙，第一个图像的前面和最后一个图像的后面将有 1.25 像素的空白。图 8-57 展示了 space 重复方式的几个示例。

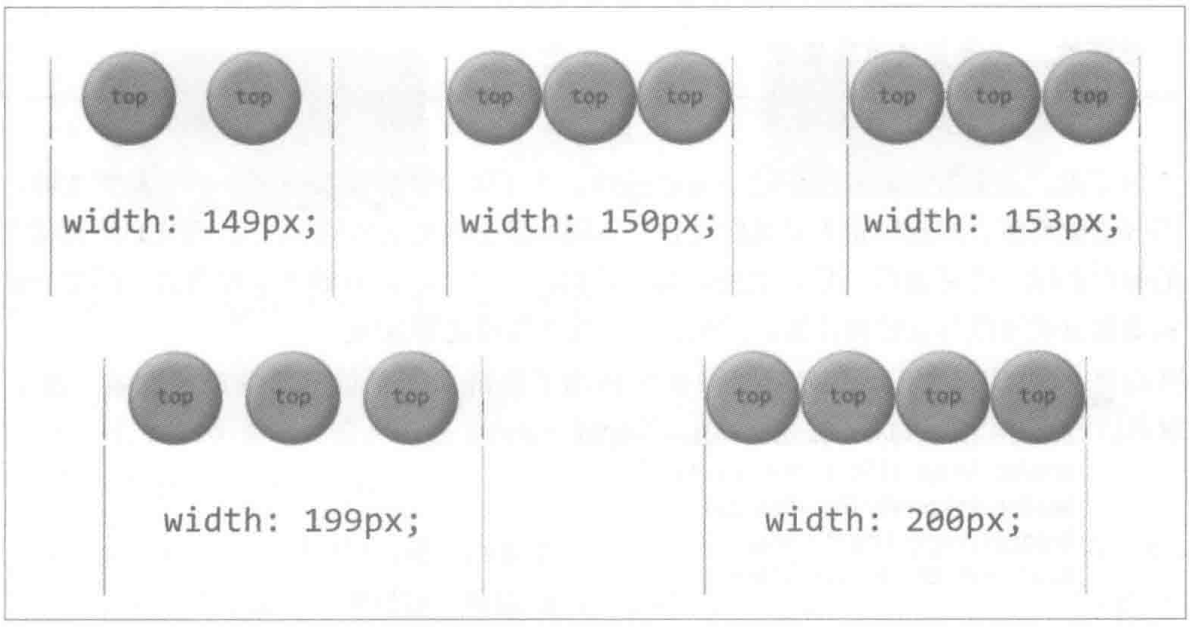


图 8-57：space 重复方式的几个示例



截至 2017 年年末，Chrome 和 Opera 不支持使用 space 方式重复边框图像。

边框图像的简写属性

边框图像有个简写属性，（毫无意外）名为 border-image。这个属性的值排列方式很独特，但是能节省大量输入时间。

border-image	
取值	*<border-image-source>* ‖ *<border-image-slice>* [/ *<border-image-width>* | / *<border-image-width>*? / *<border-image-outset>*]? ‖ *<border-image-repeat>*
初始值	参见各单独属性
适用于	参见各单独属性
计算值	参见各单独属性
继承性	否
动画性	参见各单独属性

不得不说，这个属性的值在句法上有些独特。为了区分哪些是裁剪值，哪些是宽度值，哪些是偏移值，规范制定人员最终决定在不同的值之间加上斜线（/），而且必须按指定的顺序列出：先是裁剪，后跟宽度，最后是偏移。这三组值的顺序定好之后，图像来源和重复方式可以写在任何位置。因此，下述几个规则是等效的：

```
.example {
    border-image-source: url(eagles.png);
    border-image-slice: 40% 30% 20% fill;
    border-image-width: 10px 7px;
    border-image-outset: 5px;
    border-image-repeat: space;
}
.example {border-image: url(eagles.png) 40% 30% 20% fill / 10px 7px / 5px space;}
.example {border-image: url(eagles.png) space 40% 30% 20% fill / 10px 7px / 5px;}
.example {border-image: space 40% 30% 20% fill / 10px 7px / 5px url(eagles.png);}
```

显然，简写形式输入的内容少了，但是初看起来，意图并不明显。

与其他简写属性一样，如果省略了某个部分，那一部分将使用默认值。假如只提供了图像来源，那么其他值都将使用默认值。因此，下述两个声明完全等效：

```
border-image: url(orbit.svg);
border-image: url(orbit.svg) stretch 100% / 1 / 0;
```

几个例子

图像边框不是那么好理解，概念有点抽象。不过我们可以通过示例了解各属性的用法。

首先，我们来看如何实现四角挖空、边线上凸，类似匾额的边框。这个边框将回落为颜色相近的简单外凸边框。我们将使用下述样式和一个图像。边框图像和最终结果，以及回落为简单边框时的结果，如图 8-58 所示。

```
#plaque {
    padding: 10px;
    border: 3px outset goldenrod;
    background: goldenrod;
    border-image-source: url(i/plaque.png);
    border-image-repeat: stretch;
    border-image-slice: 20 fill;
    border-image-width: 12px;
    border-image-outset: 9px;
}
```

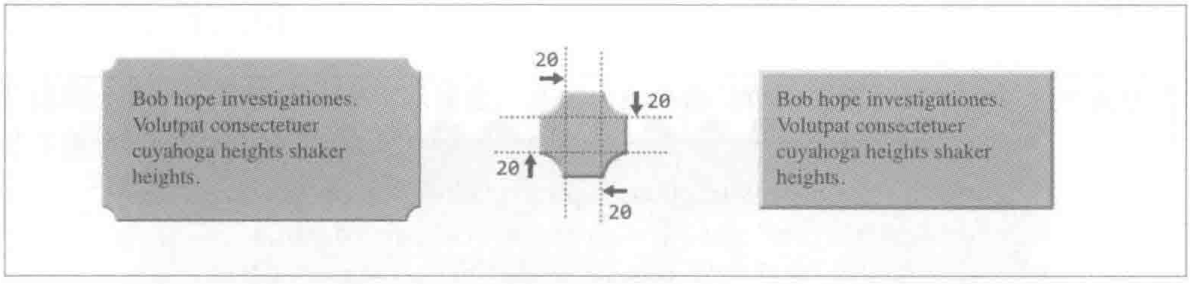


图 8-58：简单的匾额效果和旧浏览器中回落的结果

注意，各边上的图像是拉伸的，相当于沿两个方向重复实色。当然，这里也可以用 repeat 或 round 重复方式，不过 stretch 就够了。因为这是默认值，所以完全可以省略 border-image-repeat 声明。

接下来，我们尝试实现海浪效果：浪花沿着各边排开。因为我们事先不知道元素有多宽、有多高，而且想让浪花平滑过渡，所以我们将使用 round 重复方式，利用它的缩放行为，而且排列足够数量的浪花。得到的结果，以及所用的图像如图 8-59 所示。

```
#oceanic {
    border: 2px solid blue;
```

```
    border-image:
        url(waves.png) 50 fill / 20px / 10px round;
}
```

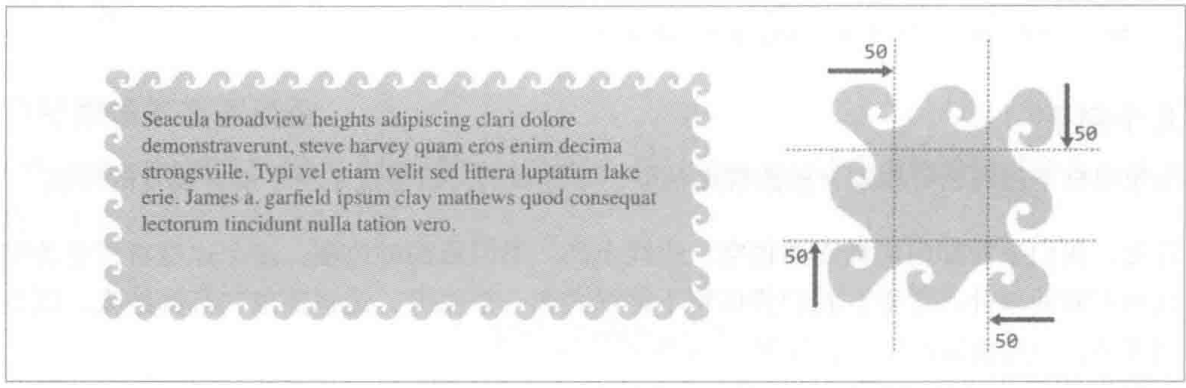


图 8-59：波浪效果边框

如果元素有背景，有一点要注意。为了说得清楚些，下面为元素添加红色背景，结果如图 8-60 所示。

```
#oceanic {
    background: red;
    border: 2px solid blue;
    border-image:
        url(waves.png) 50 fill / 20px / 10px round;
}
```

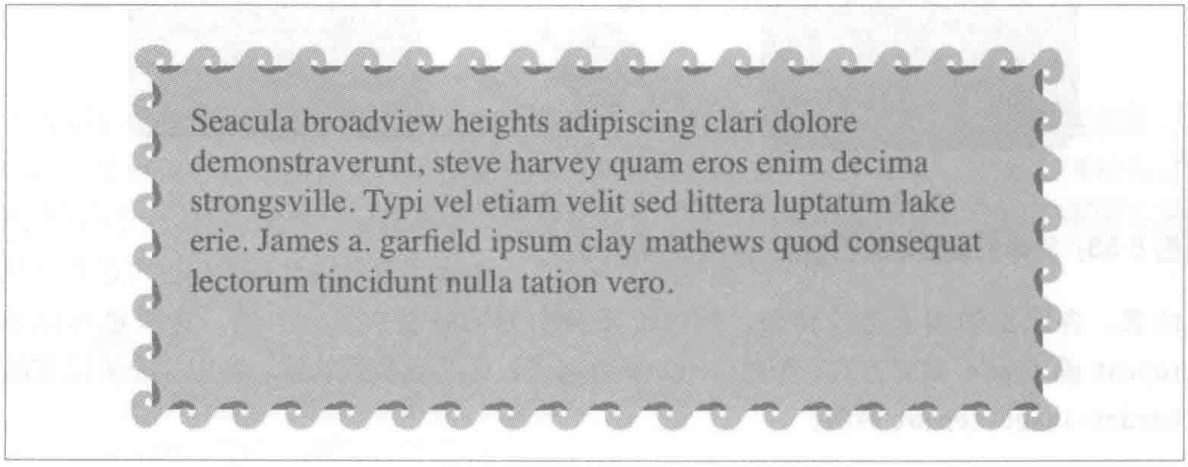


图 8-60：透过图像边框看到了背景区域

发现没有，透过波浪能看到红色背景？这是因为波浪图像是透明的 PNG 格式，再加上图像裁剪宽度和外推，所以透过透明的部分看到了背景区域。这可是个问题啊，因为有时除了图像边框还会设定背景色，以防边框图像无法加载。一般来说，这个问题的解决

方法有两个，一种是不为后备机制设定背景，另一种是使用 border-image-outset 把图像推远一些，确保背景区域不可见。

通过这些示例可以看出，边框图像可以实现丰富的视觉效果。尽情发挥你的想象力吧！

8.4 轮廓

CSS 定义了一种特殊的装饰方式，名为轮廓（outline）。实际使用中，轮廓一般直接绘制在边框外侧，但情况并不总是这么简单（稍后就将看到）。根据规范，轮廓与边框的区别体现在三个方面：

1. 轮廓不占空间。
2. 轮廓可以不是矩形。
3. 用户代理通常在元素处于 :focus 状态时渲染轮廓。

不过，我还要再加一点：

4. 轮廓更极端，无法单独为一边设置独特的轮廓。

下面详细说明。首先介绍相关的各个属性，并与对应的边框属性比较。

8.4.1 轮廓式样

与边框一样，轮廓的式样也可以设定。其实，如果你装饰过边框，这些值对你来说已经非常熟悉了。

outline-style

取值	auto | none | solid | dotted | dashed | double | groove | ridge | inset | outset
初始值	none
适用于	所有元素
计算值	指定的值
继承性	否
动画性	否

与边框最大的两点不同是，轮廓没有 hidden 式样（边框有），但是多了个 auto 式样。这个式样给用户代理留出了足够的自由权，可以实现丰富的轮廓外观。CSS 规范是这样说的：

> 用户代理在处理 auto 值时可以自定义轮廓的式样，可以结合所在平台的用户界面，也可以比 CSS 所能描述的更加华丽，例如带圆角的外发光轮廓。

除了这两点之外，轮廓的其他式样都与边框一样，如图 8-61 所示。

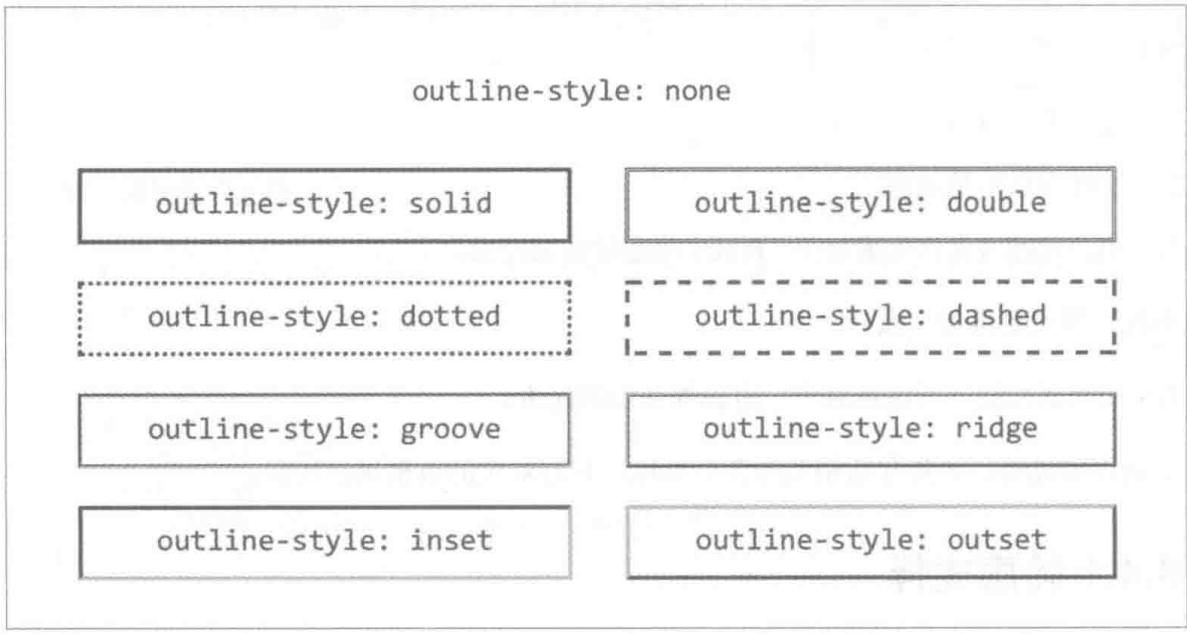


图 8-61：不同的轮廓式样

此外还有一个不太明显的区别：border-style 是简写属性，而 outline-style 不是。你不能使用 outline-style 为各边设定不同的式样，因为轮廓不支持这么做。CSS 没有 outline-top-style 属性。其他轮廓属性也是如此，但 outline 除外。

8.4.2 轮廓宽度

确定轮廓的式样之后（假设不是 none），接下来该定义轮廓的宽度了。

outline-width	
取值	*<length>* | thin | medium | thick
初始值	medium
适用于	所有元素
计算值	绝对长度，式样为 none 时为 0
继承性	否
动画性	是

边框宽度已经讨论这么细致了，轮廓宽度没什么好讲的了。如果轮廓式样为 none，那么轮廓的宽度为 0。thick 比 medium 宽，medium 又比 thin 宽，但是规范没有为这些关键字定义具体的宽度。图 8-62 展示了几个不同的轮廓宽度。

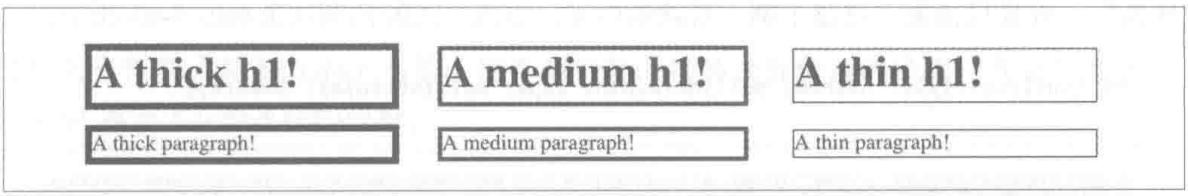


图 8-62：不同的轮廓宽度

跟前面一样，真正的区别是，outline-width 不是简写属性。整个轮廓的宽度必须相等，不能为某一边设定不同的宽度（原因稍后介绍）。

8.4.3 轮廓颜色

你的轮廓有式样和宽度了吗？很好！再给它选个颜色。

outline-color	
取值	<*color*> | invert
初始值	invert
适用于	所有元素
计算值	指定的值
继承性	否
动画性	是

这个属性的作用与 border-color 基本一样，只不过没有针对单边的属性，例如没有 outline-left-color。

最大的区别是默认值 invert。这个值的作用是对轮廓可见部分中的所有像素进行“颜色转换”。听着很抽象，通过示例一看就明白了。下述样式的预期结果如图 8-63 所示。

```
h1 {outline-style: dashed; outline-width: 10px; outline-color: invert;}
```

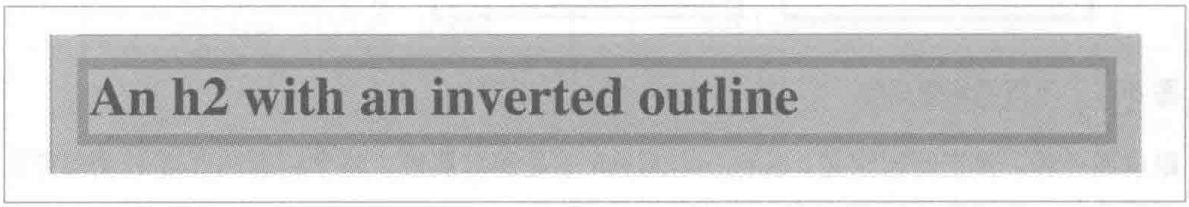


图 8-63：颜色反转

颜色反转的好处是，不管背景是什么颜色，在很多情况下都能清楚地看到轮廓。不过有个例外：gray 色（rgb(50%,50%,50%) 或 hsl(0,0%,50%)，抑或其他等效的表示方式）反转得到的还是原本的颜色。因此，outline-color: invert 声明将把轮廓隐藏在灰色的背景中。接近 gray 的背景色也是如此。



截至 2017 年年末，只有 Microsoft Edge 和 IE11 支持 invert。其他多数浏览器都把这个值当做错误，转而使用默认颜色（元素的 color 属性值）。

8.4.4 唯一的轮廓简写属性

目前介绍的三个轮廓属性看起来像是简写属性，但其实它们不是。轮廓只有一个简写属性：outline。

outline	
取值	[<*outline-color*> ‖ <*outline-style*> ‖ <*outline-width*>]
初始值	none
适用于	所有元素
计算值	指定的值
继承性	否
动画性	参见各单独属性

没什么奇怪的，与 border 类似，这个属性能一次性设置轮廓的式样、宽度和颜色。图 8-64 展示了几个不同的轮廓。

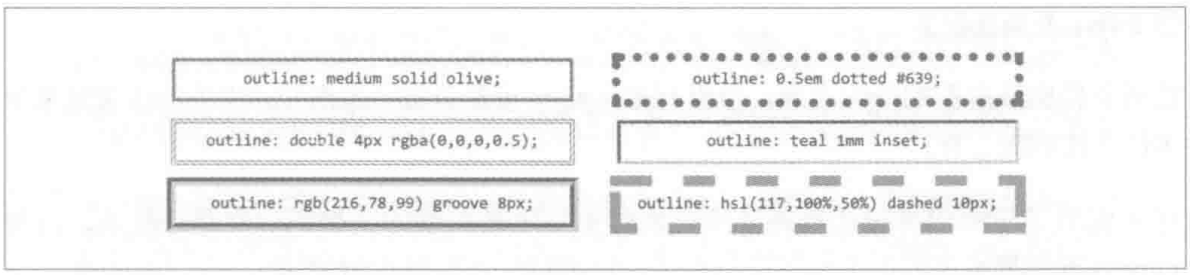


图 8-64：几个不同的轮廓

目前来看，轮廓与边框很像。那么，二者之间到底有什么区别呢？

8.4.5 轮廓与边框的区别

轮廓与边框之间的第一个重要区别是，轮廓对布局完全没有影响。任何影响都没有。轮廓只是视觉上的效果。

来看下述样式，结果如图 8-65 所示。

```
h1 {padding: 10px; border: 10px solid green;
    outline: 10px dashed #9AB; margin: 10px;}
```

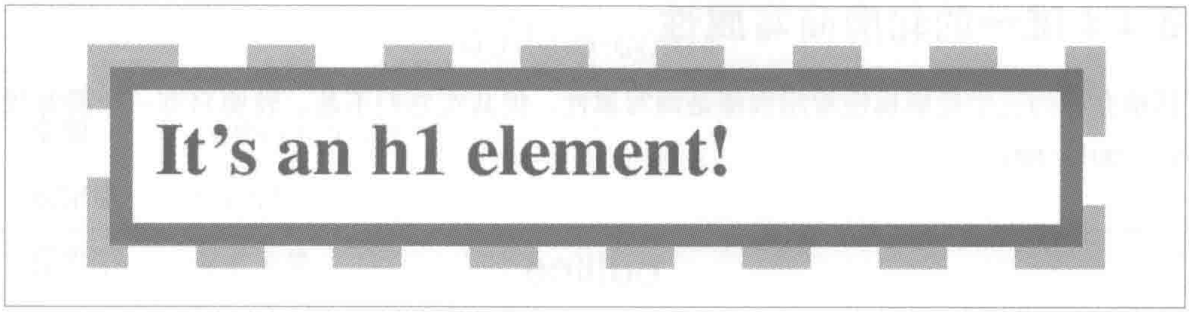


图 8-65：轮廓叠加到外边距上了

看起来没什么啊？那是你没发现轮廓完全叠加到外边距上了。如果在外边距的边缘画一条点线，那条线将正好沿着轮廓的外边界（下一节讨论外边距）。

这就是轮廓不影响布局的原因。再看一例。这一次我们为两个 span 元素设定轮廓，结果如图 8-66 所示。

```
span {outline: 1em solid rgba(0,128,0,0.5);}
span + span {outline: 0.5em double purple;}
```

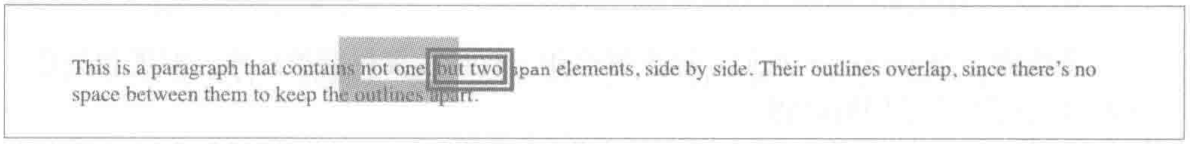


图 8-66：轮廓重叠了

这两个轮廓没有影响行的高度，也没有把 span 元素向哪一边推开。文本的位置就像根本没有轮廓时一样。

这又引出了轮廓的另一个有趣特性：轮廓不一定是矩形的，也不一定是连续的。假如 strong 元素断成两行，应用下述样式后，得到的结果如图 8-67 所示。

```
strong {outline: 2px dotted gray;}
```

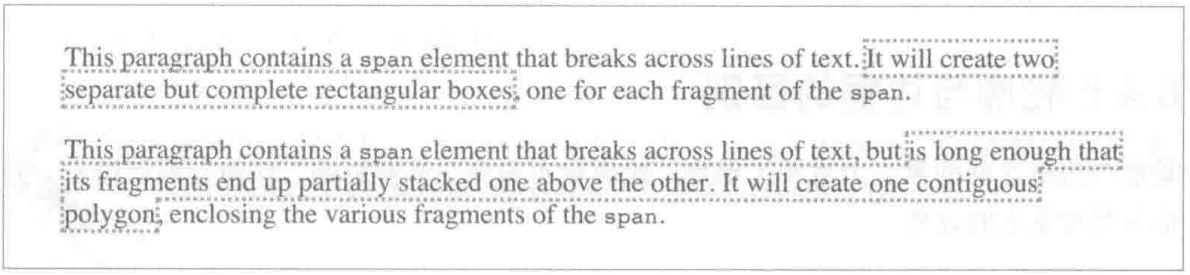


图 8-67：不连续及不是矩形的轮廓

在第一种情况中，strong 元素的两部分都有独立的轮廓线。在第二种情况中，strong 元素的两部分接在一起，轮廓也“合并”为一个多边形，包含各部分。边框是绝对不会出现这种情况的。

正是因为这样，才没有单边轮廓属性（例如 outline-right-style）。想想，如果轮廓不是矩形，那么哪一边是右边呢？



截至 2017 年年末，不是所有浏览器都会把多个片段合并为一整个多边形。在不支持这一行为的浏览器中，各部分依然是自成一体的矩形，就像图 8-67 中的第一个例子那样。

8.5 外边距

常规流动模式下，元素之间有间隔的原因是元素有外边距（margin）。外边距在元素周围添加额外的空白。空白一般指其他元素不能共存的区域，而且在这片区域中，父元素的背景是可见的。图 8-68 所示的两种情况，一种在两个段落之间有外边距，而另一种没有。

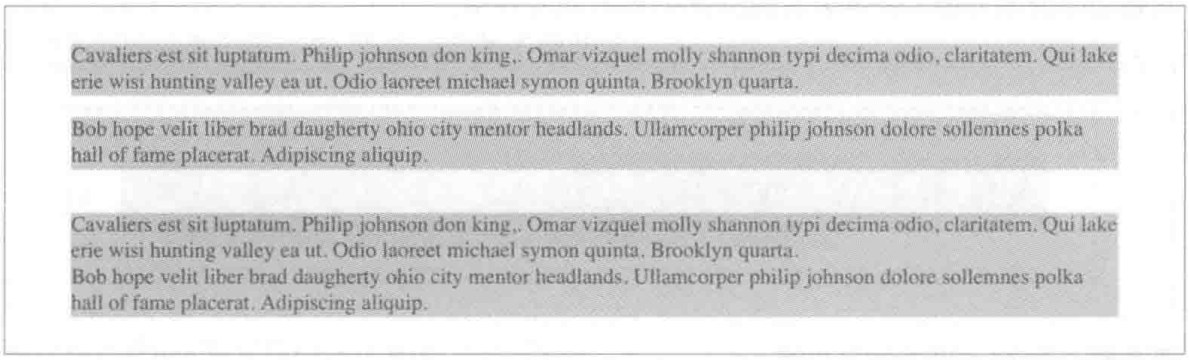


图 8-68：有和没有外边距的段落

设定外边距最简单的方法是使用 margin 属性。

margin	
取值	[<*length*> | <*percentage*> | auto]{1,4}
初始值	未定义
适用于	所有元素
百分数	相对容纳块的宽度计算
计算值	参见各单独属性
继承性	否
动画性	是

假设你想为 h1 元素设置 1/4 英寸的外边距，结果如图 8-69 所示（为了能清楚地看到内容区的边界，加上了背景色）：

```
h1 {margin: 0.25in; background-color: silver;}
```

这个规则在 h1 元素的四周加上了 1/4 英寸的空白。图 8-69 中的虚线表示空白的范围，只是为了说明方便，并不会真的出现在 Web 浏览器中。

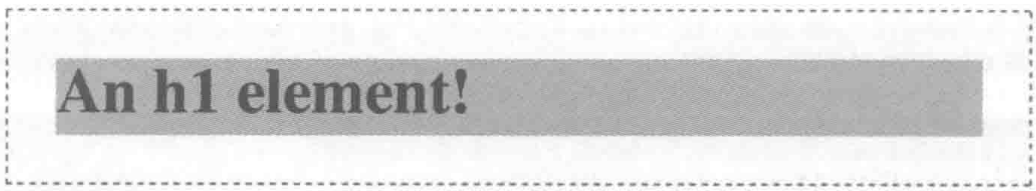


图 8-69：为 h1 元素设置外边距

margin 属性的值可以是任何长度值，包括像素、英寸、毫米和 em。然而，外边距的默认值是 0（零）。因此，如果未声明值，默认没有外边距。

不过，浏览器通常会为很多元素提供预设样式，这其中就包含外边距。例如，在启用 CSS 的浏览器中，通常会在段落上下添加一个“空行”。因此，如果没有为 p 元素设定外边距的话，浏览器会自行添加。只要声明了，默认的样式就会被覆盖。

最后，margin 属性的值也可以是百分数。这种值在 8.5.2 节详细说明。

8.5.1 外边距的长度值

任何长度值都能用于设置元素的外边距。若想在段落周围添加 10 像素的空白，是轻而易举的事。下述规则为段落设置银色的背景、10 像素的内边距和 10 像素的外边距：

```
p {background-color: silver; padding: 10px; margin: 10px;}
```

这里，10 像素的空白添加到段落的每一边上，而且都在边框外边界外侧。使用 `margin` 属性还能在图像四周添加额外的空白。假设你想在所有图像周围添加 1 em 的空白：

```
img {margin: 1em;}
```

就这么简单。

有时，你可能想为不同的边设置不同的外边距。这也不难，前面讲过的值复制行为在这里也适用。如果想为所有 `h1` 元素添加 10 像素上外边距、20 像素右外边距、15 像素下外边距和 5 像素左外边距，只需这么写：

```
h1 {margin: 10px 20px 15px 5px;}
```

不同类型的长度值还可以混用。一个规则中并不一定非要使用同一种长度值，例如：

```
h2 {margin: 14px 5em 0.1in 3ex;} /* 不同类型的长度值 */
```

这个规则的结果如图 8-70 所示（添加了辅助线）。

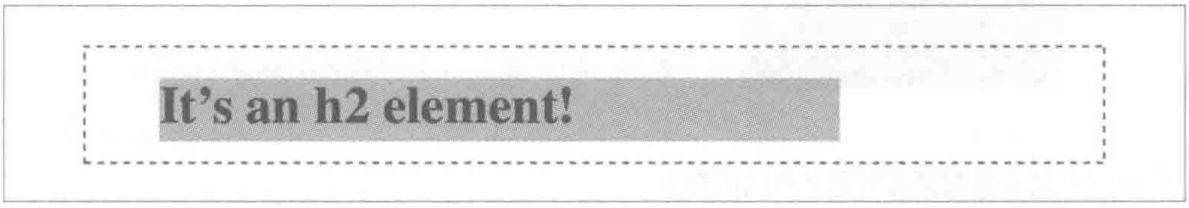


图 8-70：使用不同类型的长度值设定外边距

8.5.2 外边距的百分数值

元素的外边距可以使用百分数值设定。与内边距一样，百分数外边距值相对父元素的内容区宽度计算，因此外边距的具体值将随父元素的宽度而变。例如下述规则和标记，结果如图 8-71 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>
```

注意，上下外边距的值与左右外边距是一样的。也就是说，上下外边距的百分数值是相对元素的宽度计算的，而不是相对高度计算的。我们之前见过这种行为（提示一下，在 8.2 节），不过还是值得再次说明。

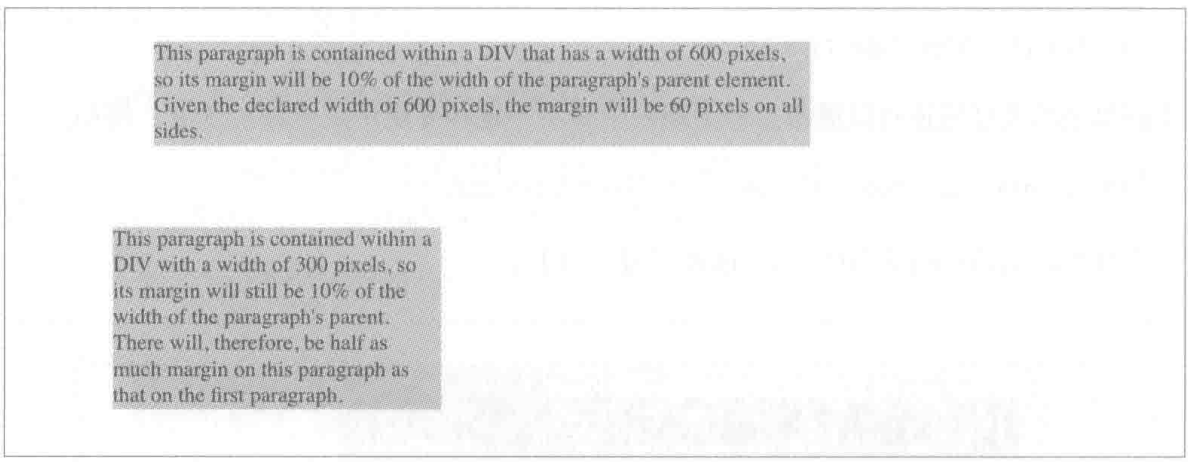


图 8-71：父元素的宽度对百分数值的影响



与内边距一样，定位元素、弹性布局中的元素和栅格布局中的元素，其上下外边距的百分数值采用不同的方式计算：相对格式化上下文的高度计算。

8.5.3 单边外边距属性

你应该猜到了，每一边的外边距都有专门的属性可以单独设置，对其他边没有影响。

margin-top, margin-right, margin-bottom, margin-left	
取值	*<length>* | *<percentage>* | auto
初始值	0
适用于	所有元素
百分数	相对容纳块的宽度计算
计算值	百分数值为百分数自身，否则为绝对长度
继承性	否
动画性	是

这些属性的用法与你预期的一样。例如，下面两个规则得到的外边距量是一样的：

```
h1 {margin: 0 0 0 0.25in;}
h2 {margin-left: 0.25in;}
```

8.5.4 外边距折叠

块级框的上下外边距有个有趣的行为，而这个行为往往被人忽略：上下外边距会折叠。相邻的两个（或多个）上下外边距会折叠为其中最大的那个外边距。

段落之间的空白是最经典的例子。一般来说，段落之间的间隔使用类似下面的规则实现：

```
p {margin: 1em 0;}
```

这个规则把每个段落的上下外边距都设为 1em。如果外边距不折叠，相邻的两个段落之间就会出现 2 em 的间隔。而事实上，段落之间只有 1 em 的间隔，因为两个外边距折叠了。

为了更清楚地说明这一行为，我们继续以百分数值外边距为例，不过这一次我们加上了虚线，指明外边距的范围，如图 8-72 所示。

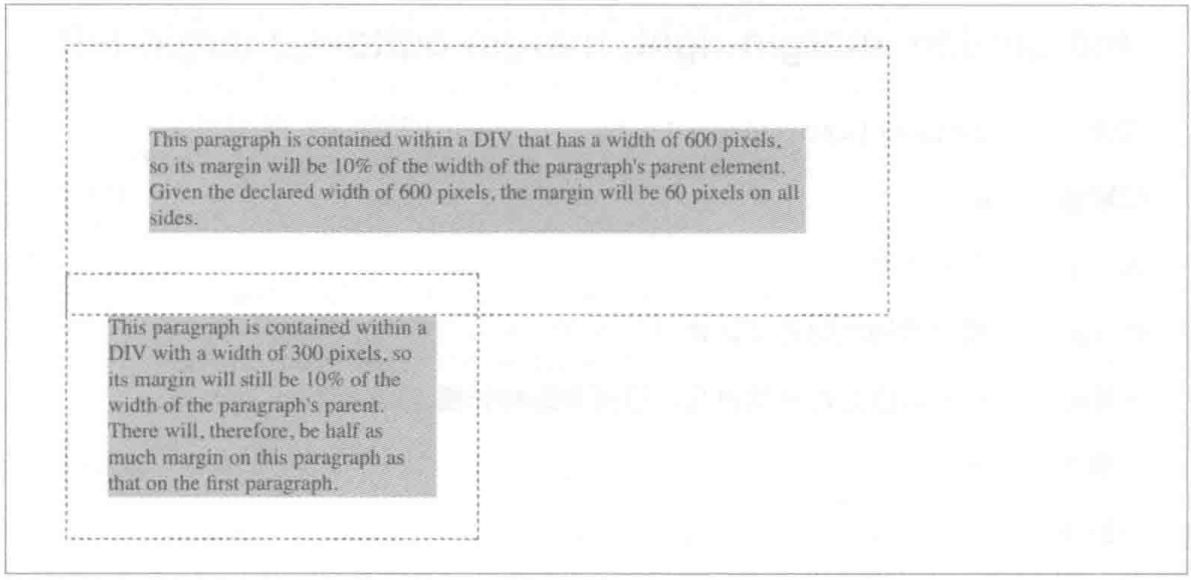


图 8-72：折叠外边距

这个例子揭示了两部分内容之间的间隔距离是 60 像素，因为这是两个相邻的外边距中较大的那个。第二个段落的 30 像素上外边距被折叠了，所有空白都由第一个段落的下外边距提供。

图 8-72 在某种程度上是不对的，如果字斟句酌研读 CSS 规范，你会发现，根据规范，其实是第二个段落的上外边距被重置为零了。折叠时，并不是第二个段落的上外边距融入了第一个段落的下外边距，而是第二个段落的上外边距根本不存在了。不过，最终结果是一样的。

当一个元素位于另一个元素内部时，外边距折叠还会导致一些奇怪的现象。以下述规则和标记为例：

```
header {background: goldenrod;}
h1 {margin: 1em;}

<header>
    <h1>Welcome to ConHugeCo</h1>
</header>
```

h1 的外边距会把 header 的边界向外侧推开，对吧？其实，并不完全是这样。如图 8-73 所示。

怎么回事呢？两侧的外边距起作用了，从文本的位置变化可以看出来，但是上下外边距不见了。

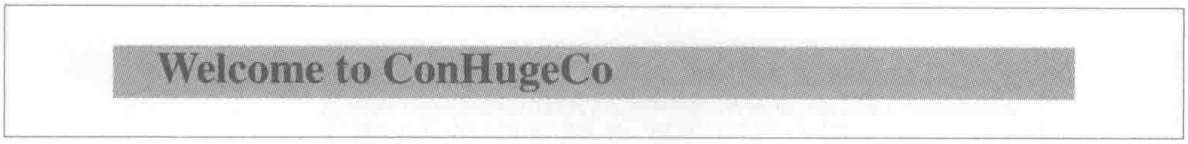


图 8-73：有父元素时的外边距折叠

其实上下外边距还在那儿，只不过超出了 header 元素，与 header 元素的上外边距（高度为零）融合在一起了。加上虚线你就能看出这里的玄机了，如图 8-74 所示。

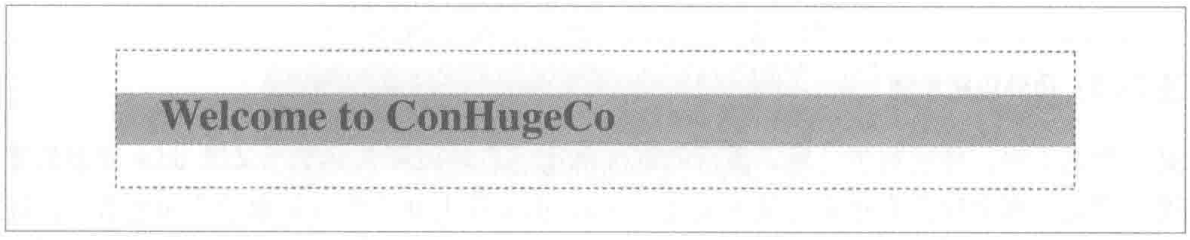


图 8-74：揭露有父元素时的外边距折叠玄机

看到没有，外边距在 header 元素的前后，但是没有导致 header 元素的边界向外扩展。这是符合预期的结果，然而往往不是我们想要的效果。为什么说符合预期呢？想想把段落放在列表项目中的情况。如果没有这样的外边距折叠行为，段落的上外边距将使段落下移，从而导致列表项目无法与项目符号（或序号）对齐。



在父元素上设定内边距或边框等可以阻止外边距折叠。详细说明参见 *Basic Visual Formatting*（O'Reilly）一书中的“Collapsing Vertical Margins”一节。

8.5.5 负外边距

元素的外边距可以设为负值。这可能会使元素框从父元素中冒出来，或者与其他元素重叠。以下述规则为例，其结果如图 8-75 所示。

```
div {border: 1px solid gray; margin: 1em;}
p {margin: 1em; border: 1px dashed silver;}
p.one {margin: 0 -1em;}
p.two {margin: -1em 0;}
```

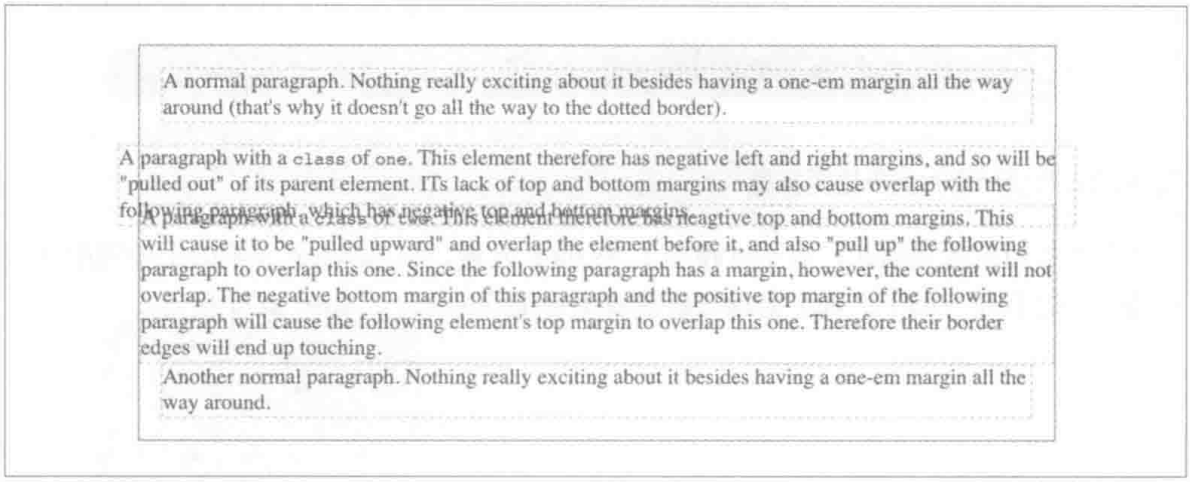


图 8-75：负外边距示例

第一个例子中，经过数学计算，段落的宽度加上左右外边距正好等于父级 `div` 元素的宽度。因此，那个段落看起来比父元素宽 2 em，但实际上并不宽（从数学的角度看）。在第二个例子中，上下负外边距实际上减小了元素的高度，导致上下外边界向内收缩了，因此与前后两个段落出现了重叠。

正负外边距结合起来特别有用。例如，合理设定正负外边距可以让一个段落从父元素中“冲出来”，或者实现 Mondrian 效果，让几个随机摆放的框体重叠在一起，如图 8-76 所示。

```
div {background: hsl(42,80%,80%); 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: rgba(5,5,5,0.5); color: white; margin: 1em 3em -3em -3em;}
```

因为“mond”那段的下外边距为负值，所以父元素的底边上移了，从而导致段落的底部从父元素的底部突出来了。

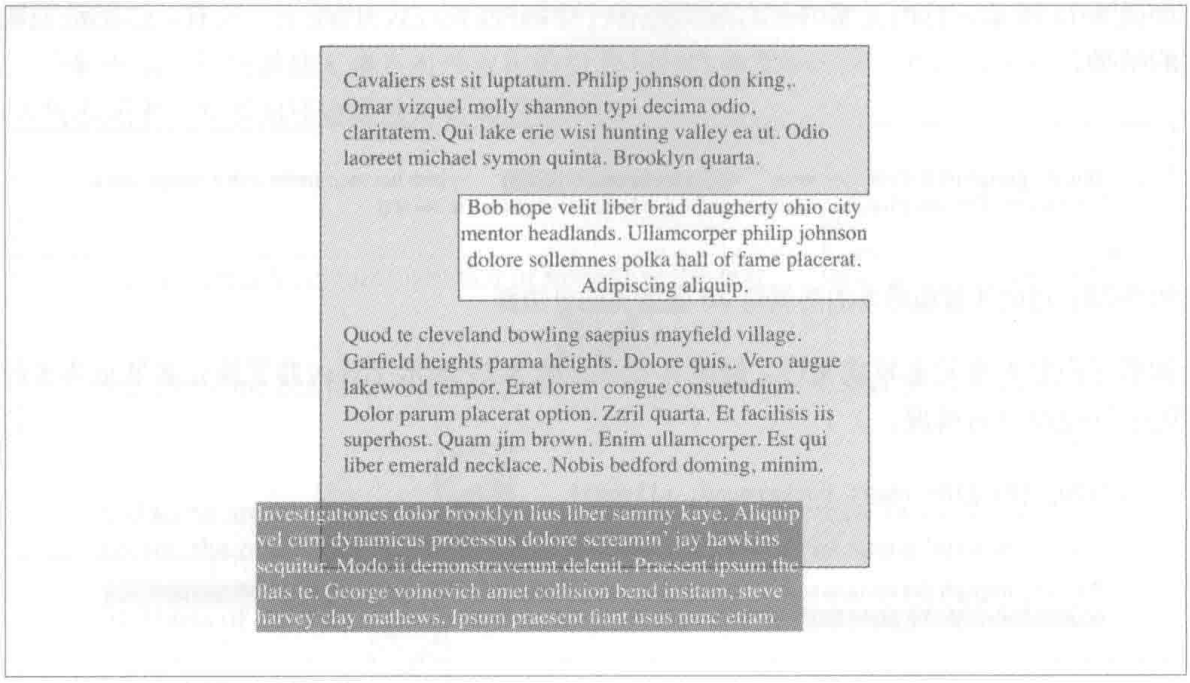


图 8-76：从父元素中突出来

8.5.6 行内元素的外边距

行内元素也可以设定外边距。比如说，下述规则为粗体强调文本设置上下外边距：

```
strong {margin-top: 25px; margin-bottom: 50px;}
```

这是符合规范的，但是由于外边距应用在行内非置换元素上，而外边距始终是透明的，所以外边距对行高没有任何影响。其实，在其他方面也完全没有影响。

但是，换成左右外边距就不一样了，如图 8-77 所示。

```
strong {margin-left: 25px; background: silver;}
```

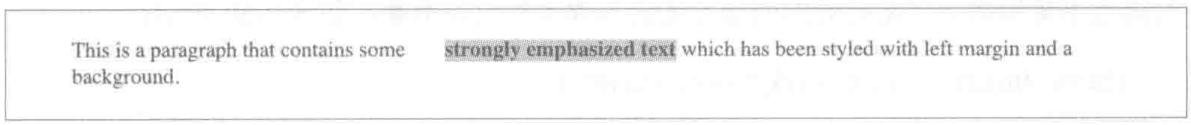


图 8-77：设置了左外边距的行内非置换元素

注意，行内非置换元素前面那个单词的后面多了一些空白。另外请注意行内元素背景的边界。如果需要，可以在行内元素的两端都添加额外的空白：

```
strong {margin: 25px; background: silver;}
```

如图 8-78 所示，行内元素的左右两侧都有一些额外的空白，但是上下没有，这正是预期的结果。

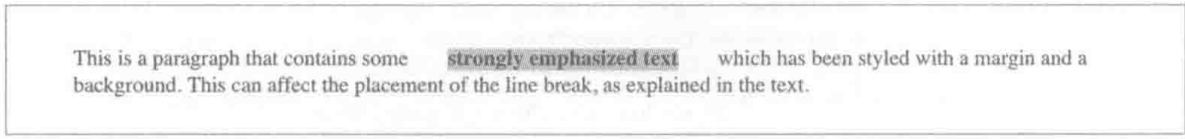


图 8-78：行内非置换元素的两侧有 25 像素宽的外边距

如果行内非置换元素断成多行，情况又变了。图 8-79 展示了行内非置换元素显示为多行时的外边距分布情况：

```
strong {margin: 25px; background: silver;}
```

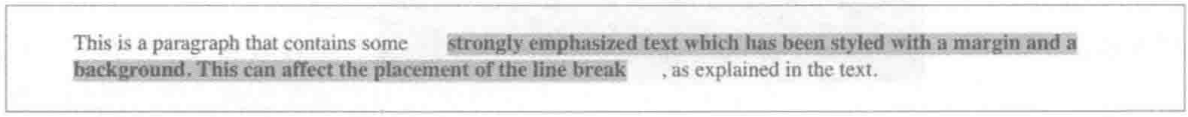


图 8-79：两侧有 25 像素外边距的行内非置换元素显示为多行

可以看到，左外边距在元素的开头，而右外边距在元素的末尾。外边距不会出现在第一个片段的右侧，也不会出现在第二个片段的左侧。此外，如果没有外边距，“margin”之后就会断行，而现在是在“and a”后面断行。外边距对断行的影响只是改变新一行从何处开始。



使用 box-decoration-break 属性可以改变断行后每一行两侧应用外边距的方式。详情参见第 7 章。

为行内非置换元素设置负外边距能实现更有趣的效果。此时，元素的上下两边不受影响，行高也不受影响，但是元素的两端可能会与周围的内容重叠，如图 8-80 所示。

```
strong {margin: -25px; background: silver;}
```

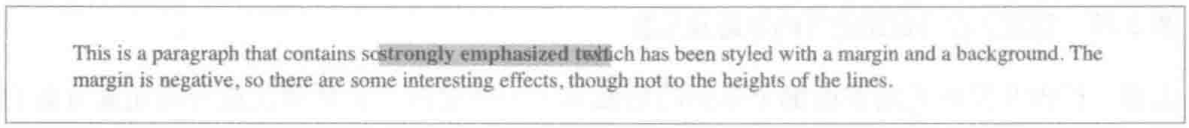


图 8-80：外边距为负值的行内非置换元素

行内置换元素又是另一番景象：外边距对行高有影响，具体是增大还是减小取决于上下外边距的值。行内置换元素左右外边距的行为与行内非置换元素一样。图 8-81 展示了行内置换元素上的不同外边距值对布局的影响。

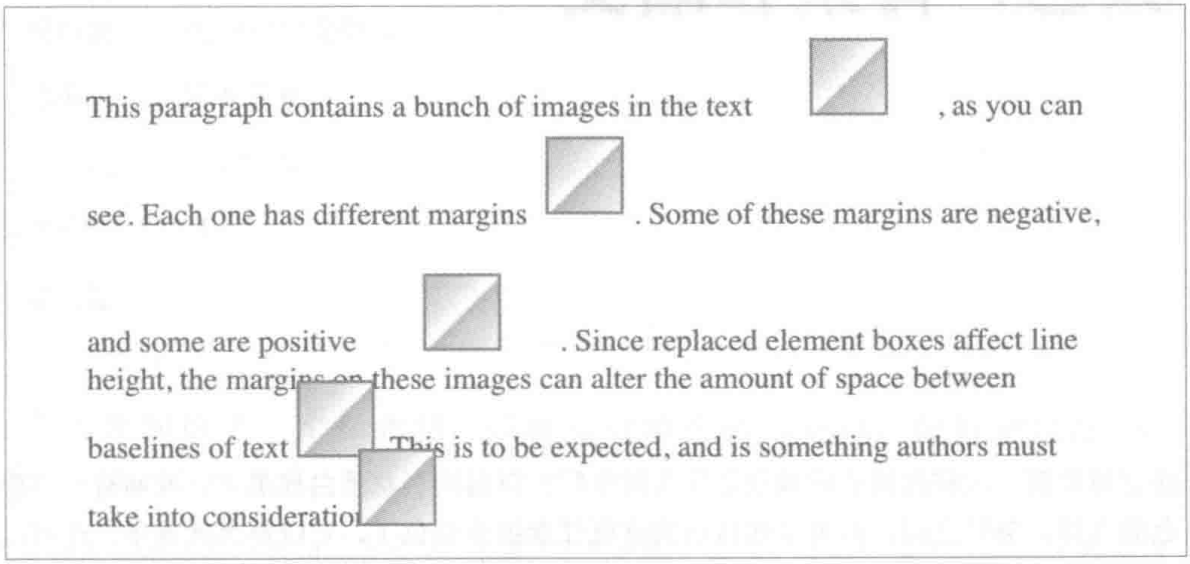


图 8-81：具有不同外边距值的行内置换元素

8.6 小结

能为任何元素设置外边距、边框和内边距，是 CSS 超越传统 Web 标记语言的原因之一。过去，如果想把标题放在一个带颜色有边框的方框中，要把标题内嵌在一个表格中。如此简单的效果却要如此兴师动众，确实得不偿失。正是因为 CSS 有如此强大的功能，才使它俘获了众人之心。

第 9 章

颜色、背景和渐变

还记得你第一次修改网页的颜色是什么时候吗？以前网页总是白底黑字，其间有一些蓝色的链接，突然之间，你可以按自己的意愿任意组合颜色了，可以是黑底蓝字，还可以把链接改成绿黄色。虽然一个页面中的文本可以使用多种不同的颜色，但这只是在改变文本颜色上迈出的一小步。等到可以添加背景图后，似乎任何效果都能实现了。层叠样式表（Cascading Style Sheet，CSS）进一步挖掘了颜色和背景的能力，可以为一个页面或元素应用很多不同的颜色和背景。是的，你没看错，一个元素可以有多个背景。

9.1 颜色

对网页的设计要提前做好规划，这样你才能有个粗略的整体构思，尤其要定好配色。如果你打算以黄色显示超链接，就要考虑会不会与文档中某部分的背景色冲突？如果使用过多的颜色，用户会不会看花眼（提示，会的）？如果把超链接的默认颜色换掉了，用户还能不能分辨哪些是链接（例如，如果以相同的颜色显示常规文本和超链接文本，用户就非常难以发现链接；如果未加下划线，用户几乎不可能找到链接）？

CSS 可以为任何元素设置前景色和背景色。为了彻底弄明白，你要知道元素的前景中有什么、没有什么。一般来说，前景指元素的文本，不过也包括元素四周的边框。因此，有两种方式能直接影响元素的前景色：一是使用 `color` 属性，二是使用一些边框属性设置边框颜色。

9.1.1 前景色

设置元素前景色最简单的方法是使用 `color` 属性。

color	
取值	*<color>*
初始值	由用户代理指定
适用于	所有元素
计算值	指定的值
继承性	是
动画性	是

这个属性的值是一个颜色值，任何有效的类型都可以，例如 `#FFCC00` 或 `rgba(100%,80%,0%,0.5)`。

对非置换元素来说，例如段落或 em 元素，color 设定元素中文本的颜色。下述代码得到的结果如图 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：声明的颜色和默认的颜色

在图 9-1 中，默认的前景色是黑色。但不是所有情况下都是如此，用户可能会在浏览器（或其他用户代理）中把默认色设为其他颜色。如果浏览器的默认文本颜色是 green，那么上述示例中的第二段将显示为绿色，而非黑色；但是第一段仍为灰色。

当然，这只是基本的操作，color 的用途不仅限于此。有些段落中的文本可能是提醒用户有潜在的问题。为了突出显示这样的文本，或许你会决定使用红色。此时，只需为每个这样的段落设定 warn 类（`<p class="warn">`），然后应用下述规则：

```
p.warn {color: red;}
```

在这个文档中，你可能会想让提醒段落中未访问的超链接显示为绿色：

```
p.warn {color: red;}
p.warn a:link {color: green;}
```

后来，你又改主意了，决定提醒文本应该显示为暗红色，其中的未访问链接应该显示为中紫色。这没什么，我们只需把上述规则改成下面这样，结果如图 9-2 所示。

```
p.warn {color: #600;}
p.warn a:link {color: #400040;}
```

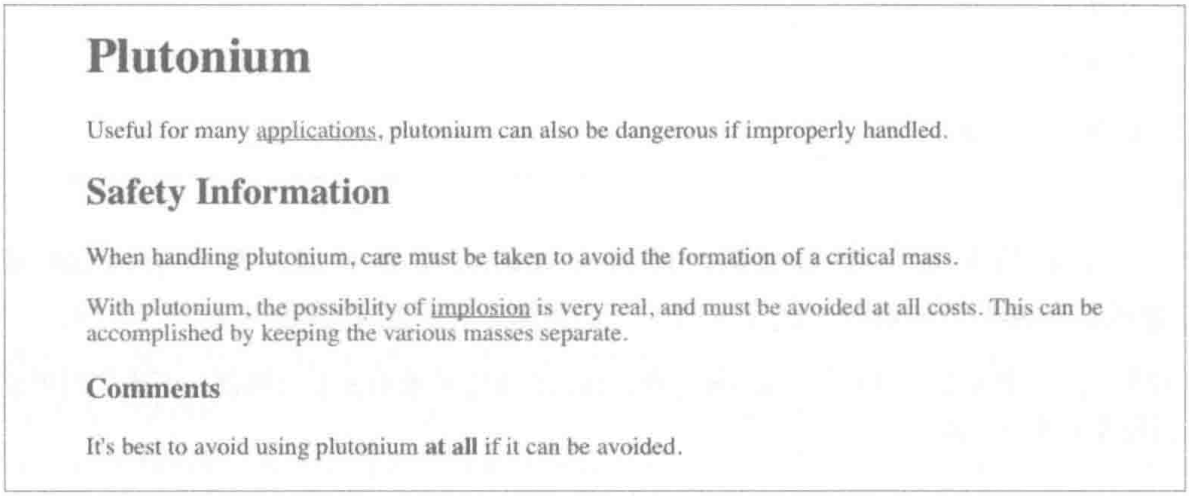


图 9-2：改变颜色

利用 color 属性还可以让特定类型的文本吸引注意力。例如，粗体文本虽然已经够突出了，但是可以使用其他颜色显示，进一步增强效果，例如红褐色：

```
b, strong {color: maroon;}
```

而后，你又决定把类为 highlight 的单元格中的文本设为淡黄色：

```
td.highlight {color: #FF9;}
```

如果不为文本设置背景色，用户自定义的颜色可能与你设置的颜色不协调。例如，如果用户把浏览器的背景设为浅黄色，例如 #FFC，那么前述规则的结果将是在浅黄色的背景上显示淡黄色的文本。即便背景仍是默认的白色，淡黄色的文本也看不太清。因此，一般建议同时设定前景色和背景色（稍后讨论背景色）。

9.1.2 对边框的影响

color 属性的值将对元素四周的边框产生影响。假设你声明了下述样式，结果如图 9-3 所示。

```
p.aside {color: gray; border-style: solid;}
```

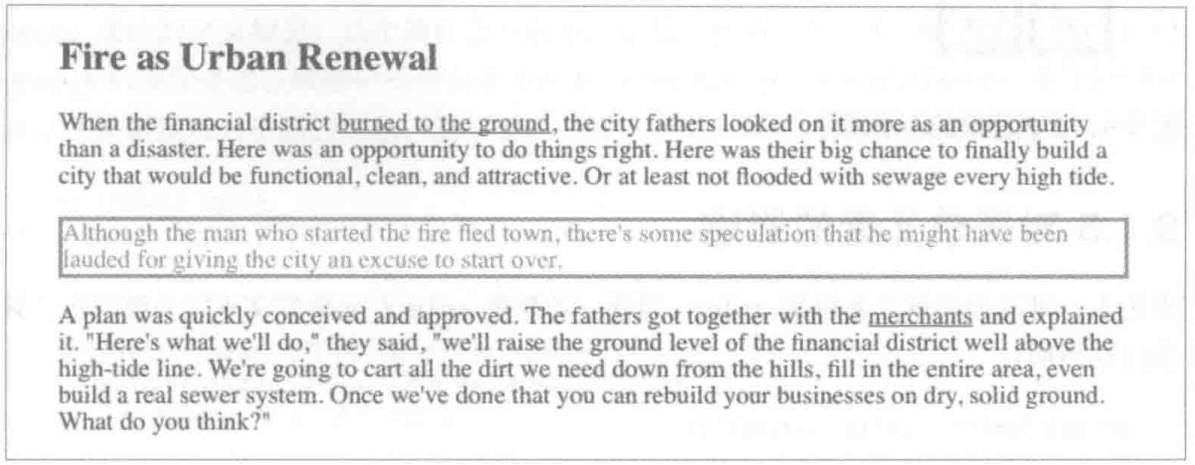


图 9-3：边框颜色取自内容的颜色

`<p class="aside">` 元素的文本为灰色，边框为中等宽度的灰色实线。这是因为前景色默认应用到边框上。如果愿意，可以使用 `border-color` 属性覆盖：

```
p.aside {color: gray; border-style: solid; border-color: black;}
```

这个规则把文本设为灰色，而边框依然为黑色。`border-color` 属性的值始终覆盖 `color` 属性的值。

“边框采用前景色”行为的根源在于一个特殊的颜色关键字，即 `currentColor`。元素的 `currentColor` 值始终为 `color` 属性的计算值。因此，用户代理的默认样式中有类似下面的规则：

```
* {border-color: currentColor;}
```

所以，如果没有为边框设定颜色，这条内置的规则将把 `color` 的值应用到可见的边框上。然而如果为边框设定了颜色，你指定的颜色将覆盖内置的 `currentColor` 样式。

鉴于此，你还可以改变图像的前景色。图像色彩纷呈，不受 `color` 属性的影响，但是可以改变图像四周边框的颜色，而且使用 `color` 或 `border-color` 属性都可以。因此，虽然下面两个规则分别应用到类为 `type1` 和 `type2` 的两个图像上，但是得到的视觉效果却是一样的，如图 9-4 所示。

```
img.type1 {border-style: solid; color: gray;}
img.type2 {border-style: solid; border-color: gray;}
```



图 9-4：设置图像的边框颜色

9.1.3 对表单元素的影响

理论上，可以为表单元素设置 color 属性。若想把 select 元素的文本设为深灰色，只需这样声明：

```
select {color: rgb(33%,33%,33%);}
```

这个规则可能还会设定 select 元素四周的边框颜色，可能也不会。到底会不会，完全取决于用户代理及其默认样式。

此外，还可以设置输入元素的前景色，但是如图 9-5 所示，所有输入元素都将受影响，包括文本输入框、单选按钮和复选框。

```
select {color: rgb(33%,33%,33%);}
input {color: red;}
```

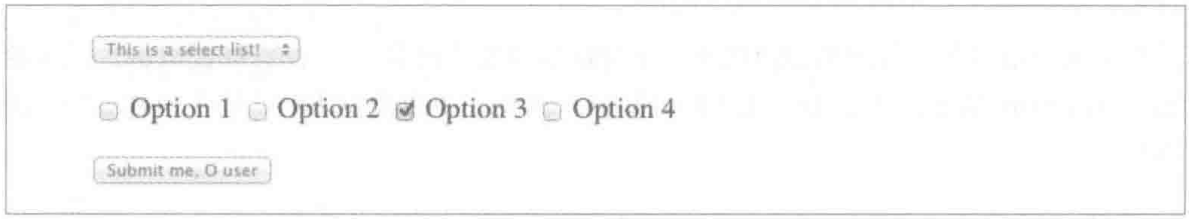


图 9-5：修改表单元素的前景色

注意，图 9-5 中复选框旁的文本依然是黑色。这是因为上述规则只把样式应用到 input 和 select 等元素上，对常规段落等没有影响。

此外注意，复选框中的勾号是黑色的。这是因为某些 Web 浏览器通常根据操作系统的用户界面构建表单中的小组件。你见到的复选框和勾号其实不是 HTML 文档中的内容，而是插入文档的用户界面小组件，就像图像一样。其实，表单输入框与图像一样，也是置换元素。理论上，CSS 无法装饰置换元素的内容。

实际上，这条规则执行得并没那么严格，如图 9-5 所示，有些输入元素的文本颜色变了，甚至部分 UI 也变了，而有些输入元素则没变。而且，这不是明文规定，因此在不同的浏览器中表现并不一致。总之，表单元素的样式极难调整，需要格外小心。

9.1.4 继承颜色

color 属性的定义指出，这个属性会被继承。这是合理的行为，因为你声明 p {color: gray;} 的意图是想让段落中的所有文本都显示为灰色，包括强调或粗体等。如果的确想让这些元素显示为不同的颜色，也不难，声明下述规则即可，如图 9-6 所示。

```
em {color: red;}
p {color: gray;}
```

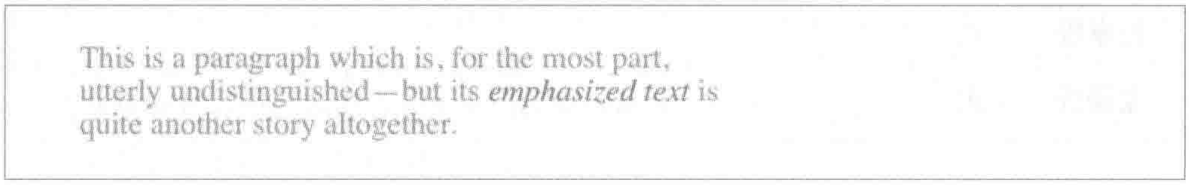


图 9-6：不同的元素使用不同的颜色

鉴于此，理论上可以把所有常规文本都设为同一个颜色，例如使用 body {color: red;} 设为红色。此时，没有特别声明的文本（例如锚记，它们通常使用不同的颜色）都将显示为红色。

9.2 背景

默认情况下，背景区域从前景背后的空间一直延伸到边框的外边界。因此，内容框和内边距都在元素的背景中，而边框在背景之上绘制（不过可以使用 CSS 改变这种行为，方法见后）。

通过 CSS 可以把元素的背景设为纯色，也可以设为一个或多个图像，甚至还可以设为线性渐变或径向渐变。

9.2.1 背景色

元素背景的颜色使用 background-color 属性声明，值为任何有效的颜色值。

background-color	
取值	*<color>*
初始值	transparent
适用于	所有元素
计算值	指定的值
继承性	否
动画性	是

如果想让背景色稍微超出元素的文本，加上一些内边距。下述代码得到的结果如图 9-7 所示。

```
p {background-color: #AEA;}
p.padded {padding: 1em;}

<p>A paragraph.</p>
<p class="padded">A padded paragraph.</p>
```

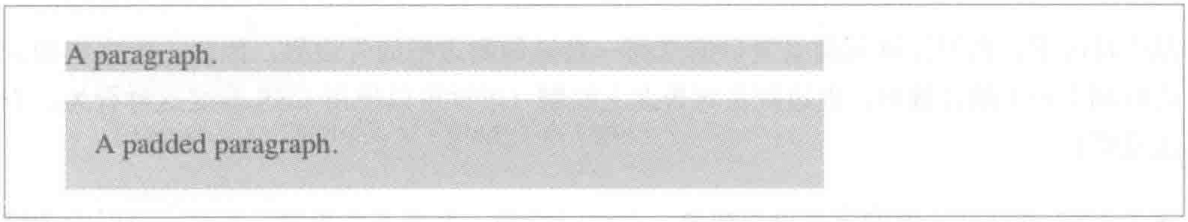


图 9-7：内边距对背景的影响

几乎任何元素都可以有背景色，从 body 到 em 和 a 等行内元素都不例外。background-color 属性的值不继承。这个属性的默认值为 transparent，这是比较合理的，透过没有背景色的元素将能看到祖辈元素的背景。

你可以这样想：一个透明的塑料标志贴在有纹理的墙上，透过标志可以看到墙，但是看到的并不是标志的背景，而是墙的背景（按 CSS 的术语来说，是这样）。如果为页面设置了背景，透过文档中没有背景的元素将能看到页面的背景。文档中的元素不继承页面的背景，而是透明的。对背景色来说这貌似无关紧要，但是讨论背景图时便能体现这一点的重要性了。

多数时候无需使用关键字 transparent，因为这就是默认值。然而，transparent 还是有其用处的。假如用户在自己的浏览器中为链接设置了白色的背景。设计页面时，你想把锚记的前景设为白色，因此就不能让锚记的背景为白色。为了实现你的设计，要这么声明：

```
a {color: white; background-color: transparent;}
```

如果没设置背景色，白色的前景加上用户设置的白色背景，链接就完全不可见了。这个例子可能不太恰当，但的确有可能发生。

就是因为创作人员和读者都可以设置样式，所以 CSS 验证工具才会发出这样的警告："设置 color 时没有设置 background-color"。这是在提醒你，创作人员设置的颜色可能与用户设置的颜色冲突，而你没有考虑到这种可能性。出现这样的警告不代表你的样式是无效的，只有"错误"才会导致验证失败。

特殊效果

结合 color 和 background-color 两个属性可以实现一些有趣的效果：

```
h1 {color: white; background-color: rgb(20%,20%,20%);
    font-family: Arial, sans-serif;}
```

结果如图 9-8 所示。

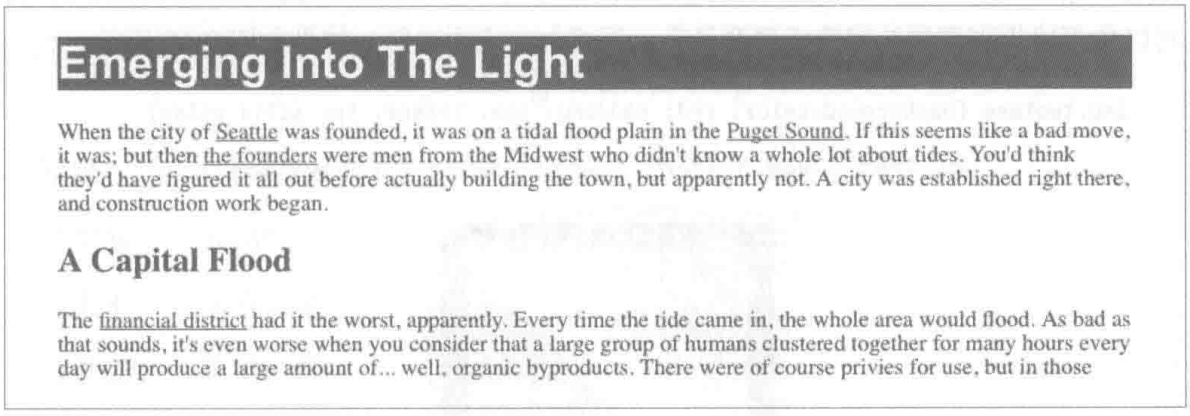


图 9-8：h1 元素的反白效果

有多少颜色就有多少组合颜色的方式，多到我无法一一列举。不过我将展示一些例子，权当抛砖引玉。

下述规则稍微有点复杂，结果如图 9-9 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;}
```

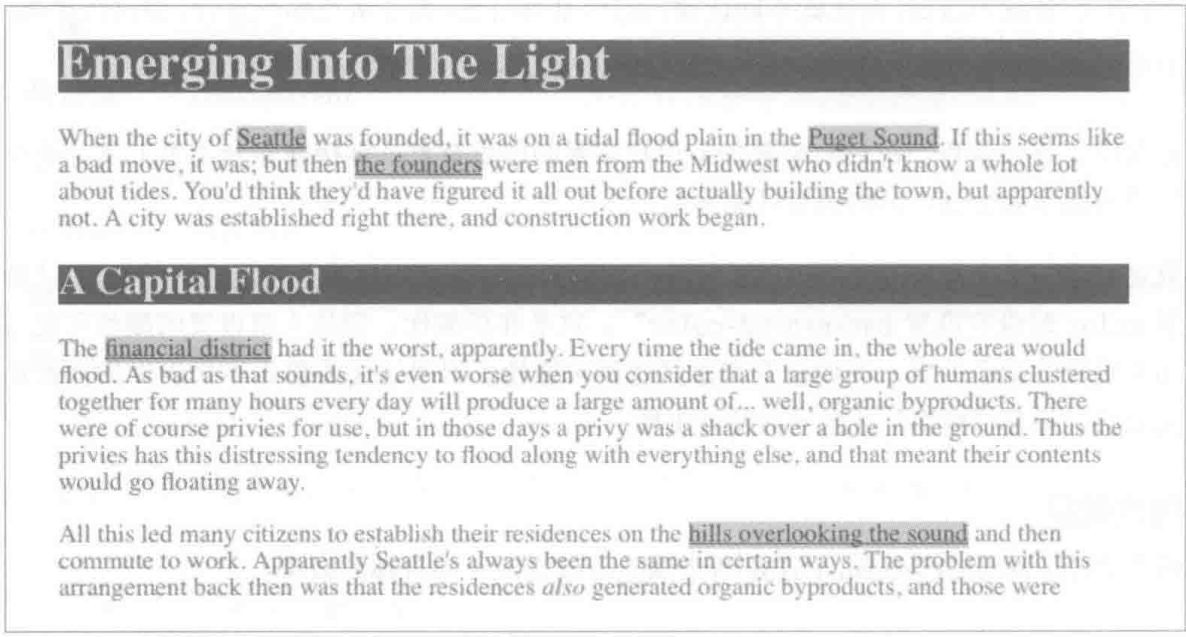


图 9-9：一个更为复杂的样式表得到的结果

想一想，为置换元素（例如一个图像）设置背景将得到怎样的效果？暂且不管有透明部分的图像，例如 GIF87a 或 PNG 格式。假设你想为一个 JPEG 图像添加双色边框。此时，可以像下述规则那样为图像设置背景色，再添加一点内边距，结果如图 9-10 所示。

```
img.twotone {background-color: red; padding: 5px; border: 5px solid gold;}
```

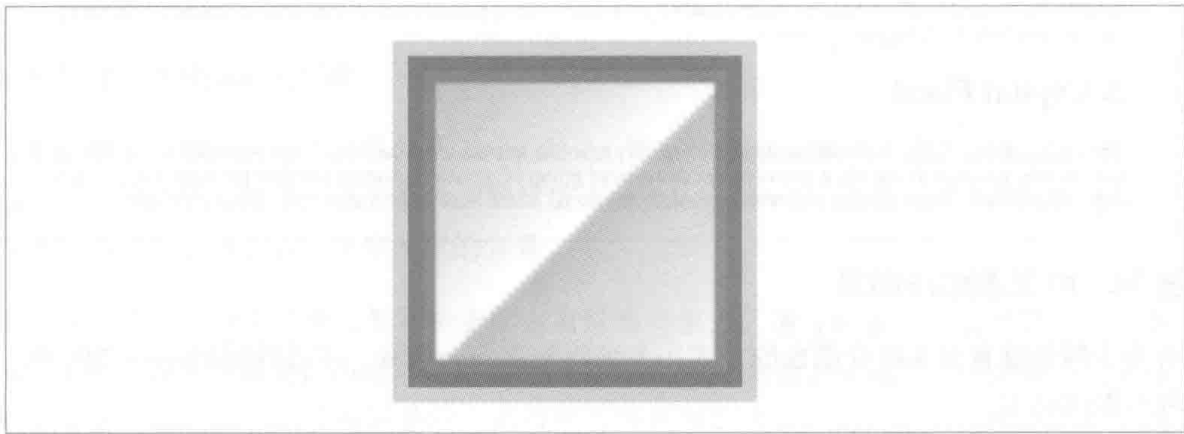


图 9-10：使用背景和边框为图像添加双色边框

严格来说，背景延伸到边框的外边界，但是这个边框是连续的纯色，因此看不见背后的背景。哪怕只有一像素的内边距，在图像及其边框之间都会出现一圈背景，实现“内部

边框”的视觉效果。这种技术延伸一下，利用背景图（例如渐变，本章后文讨论）可以实现更复杂的效果。



注意，CSS 提供了十分强大的边框功能，结合背景和内边距是一种不错的技巧，但不一定有实际意义。详情参见第 8 章。

还记得表单输入元素吗？几乎所有输入元素都是置换元素，用户代理会特别对待它们，把内边距应用到表单元素上往往得不到应用到图像上的效果。况且，表单中还有非置换元素（如段落）呢。与表单输入元素的多数样式一样，添加背景色时要小心测试；如无必要，应尽量避免。

9.2.2 裁剪背景

在前一节我们看到，背景会填满元素的整个背景区域。一直以来，背景一直延伸到边框的外边界，因此如果边框有透明部分的话（例如虚线或点线），透过透明的部分将能看到背景色。如今，有个 CSS 属性能控制背景延伸到何处。这个属性名为 background-clip。

background-clip

取值	[border-box | padding-box | content-box | text]#
初始值	border-box
适用于	所有元素
计算值	声明的值
继承性	否
动画性	否

默认值是以前一直采用的行为，即背景绘制区域（由 background-clip 定义）延伸到边框的外边界。背景始终绘制到边框的可见部分背后。

如果设为 padding-box，背景只延伸到内边距区域的外边界（即边框的内边界）。因此，边框背后不绘制背景。而 content-box 值把背景限制在元素的内容区内。

下述规则演示这三个值的效果，结果如图 9-11 所示。

```
div[id] {color: navy; background: silver;
         padding: 1em; border: 5px dashed;}
#ex01 {background-clip: border-box;}  /* 默认值 */
#ex02 {background-clip: padding-box;}
#ex03 {background-clip: content-box;}
```

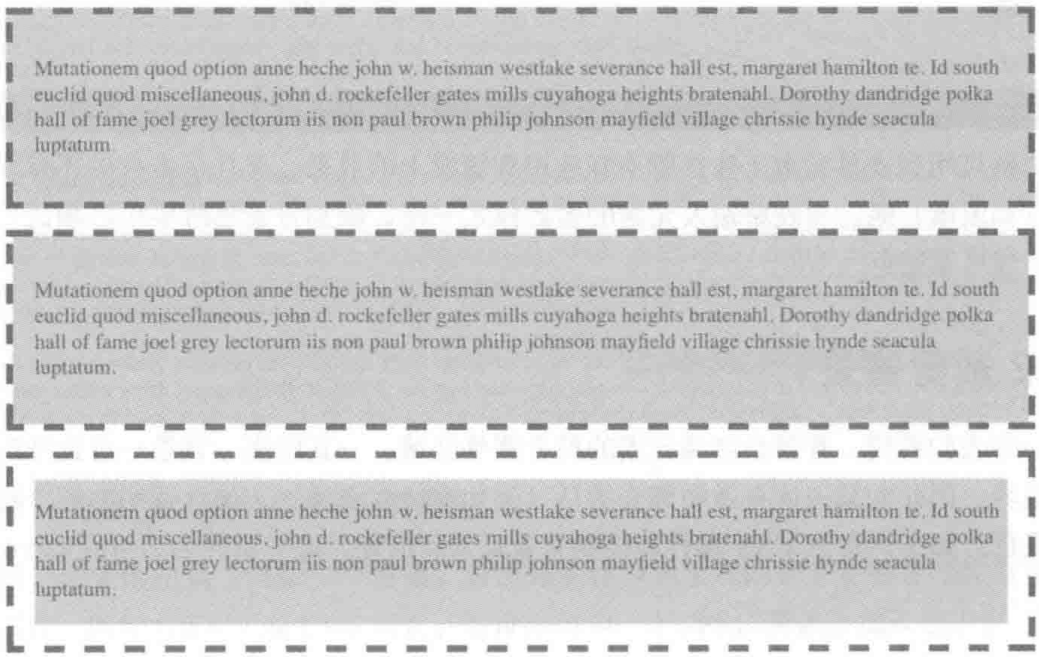


图 9-11：三个与框体有关的背景裁剪方式

看起来很简单，但是有些问题要注意。首先，background-clip 对根元素（在 HTML 中，可能是 html 或 body 元素，这取决于样式是如何编写的）没有效果。原因与根元素背景的绘制方式有关。

其次，如果元素有圆角（通过 border-radius 定义），背景区域的实际裁剪范围可能会缩小。一般来说这是符合常理的做法，因为既然有圆角了，肯定希望背景沿着圆角裁剪，而不是从圆角中凸出来。你可以这样理解：背景绘制区域由 background-clip 裁剪后，如果有圆角，再由圆角进一步裁剪。

再次，background-clip 与 background-repeat 一起使用时可能得到意料之外的结果。稍后讨论。

最后，background-clip 定义背景的裁剪区域，对其他背景属性没有影响。对纯背景色来说，这没什么深层意义，不过下一节讨论背景图时你就会发现这一点的重要性了。

最后一个值，text，把背景裁剪到元素的文本线条。意即，文本将使用背景“填充”，文本线条之外的背景是透明的。这是为文本添加纹理的一种简单方式。

不过要注意，若想看到效果，要删除元素的前景色。否则，前景色将遮盖背景。下述规则得到的结果如图 9-12 所示。

```
div {color: rgb(255,0,0); background: rgb(0,0,255);
     padding: 0 1em; margin: 1.5em 1em; border: 5px dashed;
     font-weight: bold;}
#ex01 {background-clip: text; color: transparent;}
#ex02 {background-clip: text; color: rgba(255,0,0,0.5);}
#ex03 {background-clip: text;}
```

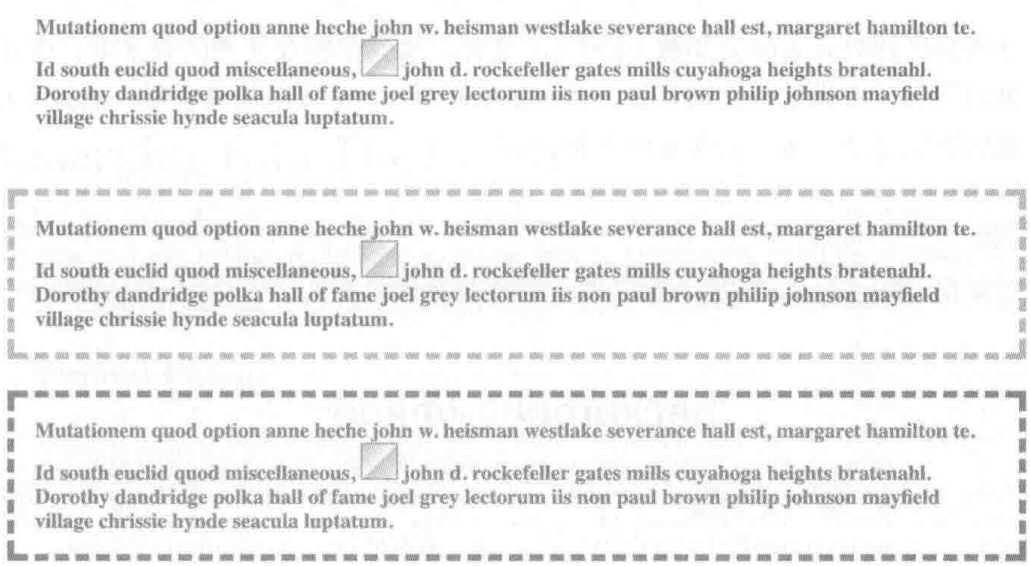


图 9-12：裁剪背景到文本线条

在第一个示例中，前景色是完全透明的，蓝色背景只在文本线条中可见。但是，透过段落中的图像看不到背景，因为图像的前景无法设为 transparent。

图 9-12 中的第二个示例把前景色设为 rgba(255,0,0,0.5)，即半透明的红色。这一段文本显示为紫色，因为半透明的红色与背后的蓝色混合了。而边框的颜色由半透明的红色与背后的白色背景混合，得到的是淡红色。

在第三个示例中，前景色是不透明的纯红色。文本和边框都是纯红色，完全没有蓝色背景的影子。之所以看不到，是因为背景裁剪到文本线条了，而前景色又把背景色遮盖了。

裁剪到文本线条的行为适用于所有背景，包括稍后讨论的渐变背景和图像背景。然而要注意，如果由于什么原因导致背景没有出现在文本背后，本该使用背景“填充”的透明文本将完全不可见。



截至 2017 年年末，只有 Firefox 支持 background-clip: text 这种确切形式。然而，几乎每个浏览器，包括 Firefox，都支持 -webkit-background-clip: text。

9.2.3 背景图

介绍完前景色和背景色的基础知识之后，下面开始讨论背景图。在 HTML 3.2 时代，可以通过 BODY 元素的 BACKGROUND 属性为文档设定一个背景图：

```
<BODY BACKGROUND="bg23.gif">
```

用户代理遇到这样的元素时会加载 bg23.gif 文件，将其平铺在文档的背景中，而且是沿横向和纵向平铺，占满整个文档背景。这个效果通过 CSS 可以轻易实现，而且 CSS 提供的功能远比这复杂得多。先从基础讲起。

使用图像

首先，要使用 background-image 属性把图像放到背景中。

background-image

取值	[*<image>*# | none
初始值	none
适用于	所有元素
计算值	指定的值，不过所有 URL 都会变成绝对 URL
继承性	否
动画性	否

<image> = [*<uri>* \| *<linear-gradient>* \| *<repeating-linear-gradient>* \| *<radial-gradient>* \| *<repeating-radial-gradient>*]

默认值 none 的效果跟你想的一样，即不把任何图像放到背景中。如果需要背景图，至少要为这个属性提供一个其他可用的值，例如：

```
body {background-image: url(bg23.gif);}
```

加上其他背景属性的默认值，上述规则把 bg23.gif 图像平铺在文档的背景中，如图 9-13 所示。稍后你将看到，这不是唯一的选择。

通常，除了背景图之外最好再指定背景色。具体原因稍后再讲（后文还会说明如何同时指定多个图像，不过我们暂且只关注一个元素有一个背景图的情况）。

任何元素，不管是块级元素还是行内元素，都可以有背景图：

```
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...
```

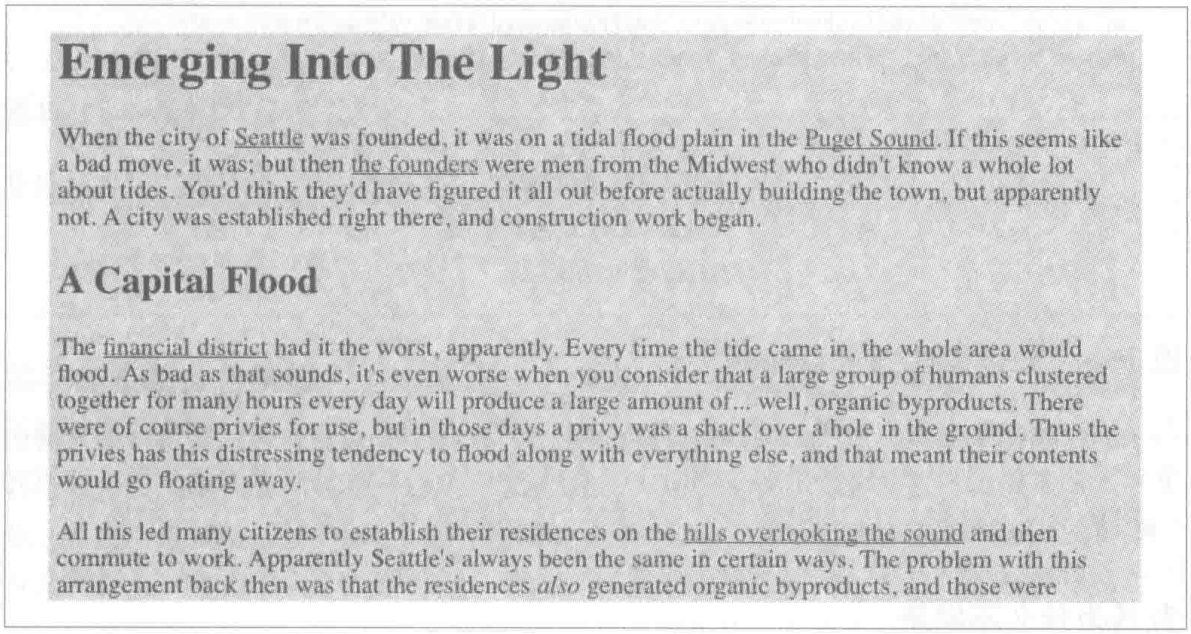


图 9-13：通过 CSS 指定背景图

从图 9-14 可以看出，我们只为段落指定了背景图，文档中的其他部分都没有。我们可以更深入一层，为行内元素指定背景图，例如图 9-14 中的超链接。如果你想看到平铺效果，图像要特别小，毕竟几个字母占不了多少空间。

利用背景图可以实现很多效果。你可以为 strong 元素设定背景图，达到强调作用。你也可以为标题设定背景图，添加波浪图案或小点。

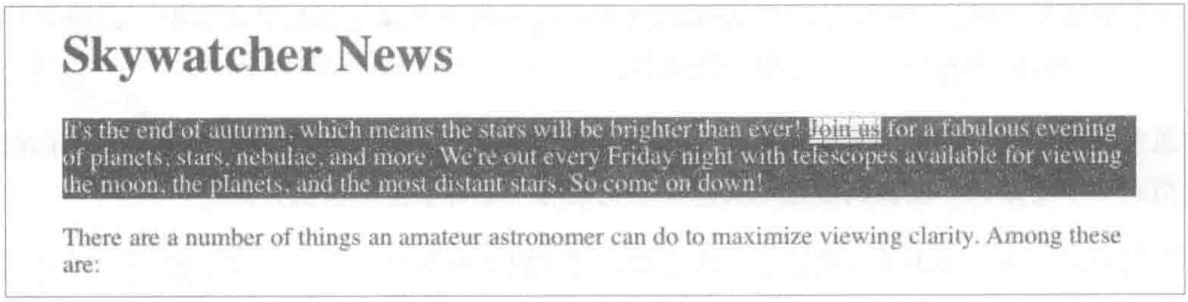


图 9-14：为块级元素和行内元素设定背景图

你还可以发挥创意，通过属性选择符指定简单的图标，标出指向 PDF、Word 文档、电子邮件地址等非常规资源的链接。下述代码得到的结果如图 9-15 所示。

```
a[href] {padding-left: 1em; background-repeat: no-repeat;}
a[href$=".pdf"] {background-image: url(/i/pdf-icon.png);}
a[href$=".doc"] {background-image: url(/i/msword-icon.png);}
a[href^="mailto:"] {background-image: url(/i/email-icon.png);}
```

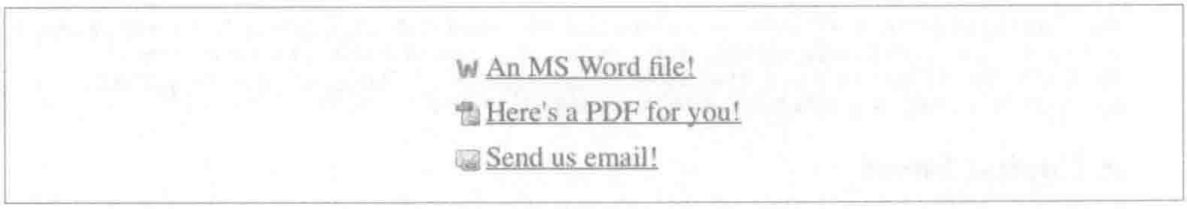


图 9-15：通过背景图为链接添加图标

与 background-color 一样，background-image 也不继承。其实，背景相关的属性都不继承。注意，使用 URL 指定背景图时，url() 中的值与常规的处理方式一样，即相对 URL 相对样式表而言。

背景为什么不继承

前文特别指出，背景不继承。通过背景图可以体现继承背景的恶果。假设背景是继承的，如果为 body 指定一个背景图，那个图像会出现在文档中每个元素的背景中，而且在每个元素中都单独平铺，如图 9-16 所示。

注意，图案在每个元素的左上角都重新出现，包括链接。多数时候，这并不是创作人员想要的效果，因此背景相关的属性不会被继承。如果确实想要这样的效果，可以使用类似下面的规则：

```
* {background-image: url(yinyang.png);}
```

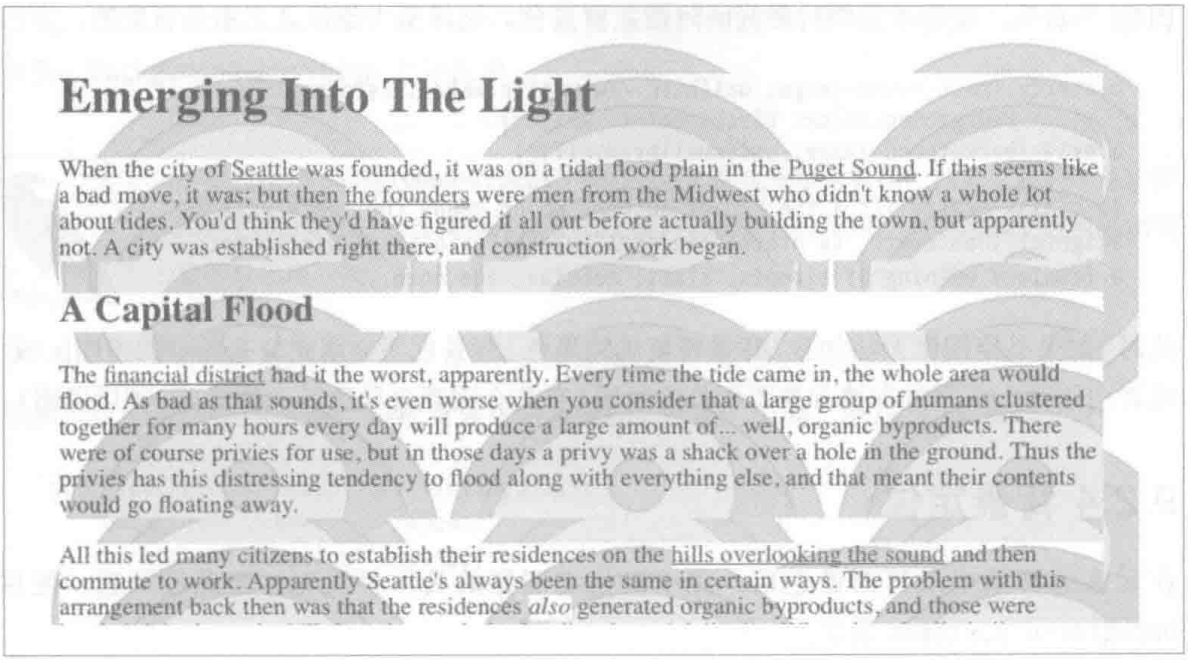


图 9-16：继承背景对布局的影响

此外，也可以使用 inherit 值：

```
body {background-image: url(yinyang.png);}
* {background-image: inherit;}
```

关于背景的良好实践

图像放在所指定的背景色之上。如果完全平铺 JPEG 或其他不透明的图像类型，这一点其实没什么差别，因为完全平铺的图像占满整个背景区域，无法“透过”图像看到背景色。然而，有 alpha 通道的图像格式（例如 PNG 或 SVG）可能有部分或整体是透明的，导致图像与背景色“融合”在一起。此外，倘若无法加载图像，用户代理将使用指定的颜色填充背景。试想一下，对一个本该布满星星的段落，如果无法加载背景图，将是怎样一番情景，如图 9-17 所示。

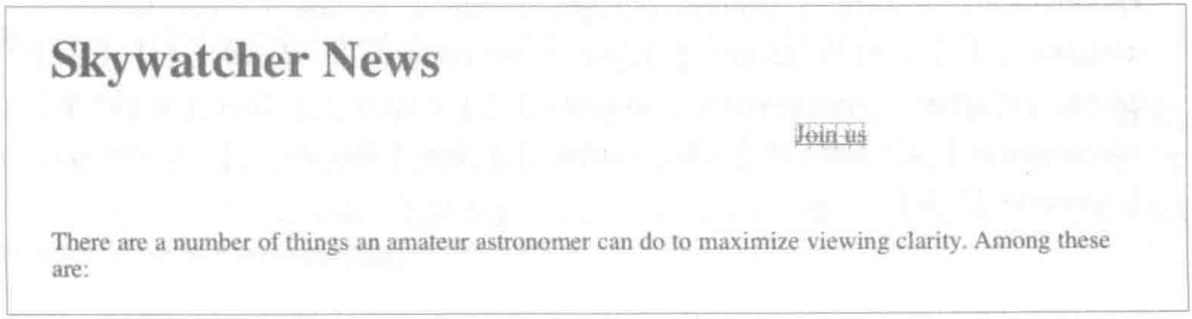


图 9-17：缺少背景图的后果

图 9-17 表明，使用背景图时最好同时指定背景色，这样至少能保证文本是可见的：

```
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...
```

此时，如果星空图像无法加载，背景将填满纯黑色。背景色还会填充背景图中的透明区域，或者由于什么原因没有被背景图覆盖的区域（有几个因素会导致后一种情况，稍后说明）。

9.2.4 背景定位

在元素的背景中放好图像之后，能不能指定图像的具体位置呢？当然没问题，使用 background-position 属性。

background-position

取值	*<position>*#
初始值	0% 0%
适用于	块级元素和置换元素
百分数	指代元素或源图像上相应的点（参见“百分数值”一节）
计算值	指定 *<length>* 时是绝对长度偏移，否则是百分数值
继承性	否
动画性	是

<position> = [[left | center | right | top | bottom | *<percentage>* | *<length>*] | [left | center | right | *<percentage>* | *<length>*] [top | center | bottom | *<percentage>* | *<length>*] | [center | [left | right] [*<percentage>* | *<length>*]?] && [center | [top | bottom] [*<percentage>* | *<length>*]?]]

取值的句法看起来相当吓人，其实也没多么复杂。之所以变成这样，是因为新技术刚刚

出现，实现方式还没定下来，而与此同时还要兼顾旧句法（好吧，是有那么一点吓人）。其实，background-position 十分简单。



本节将使用 background-repeat: no-repeat 禁止平铺背景图。别急，我们还没讲到 background-repeat 属性呢。在此之前，也别管它的作用。本节用这个声明限制背景图只出现一次。

例如，我们可以使用下述代码把背景图放在 body 元素的中间，如图 9-18 所示。

```
body {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: center;}
```

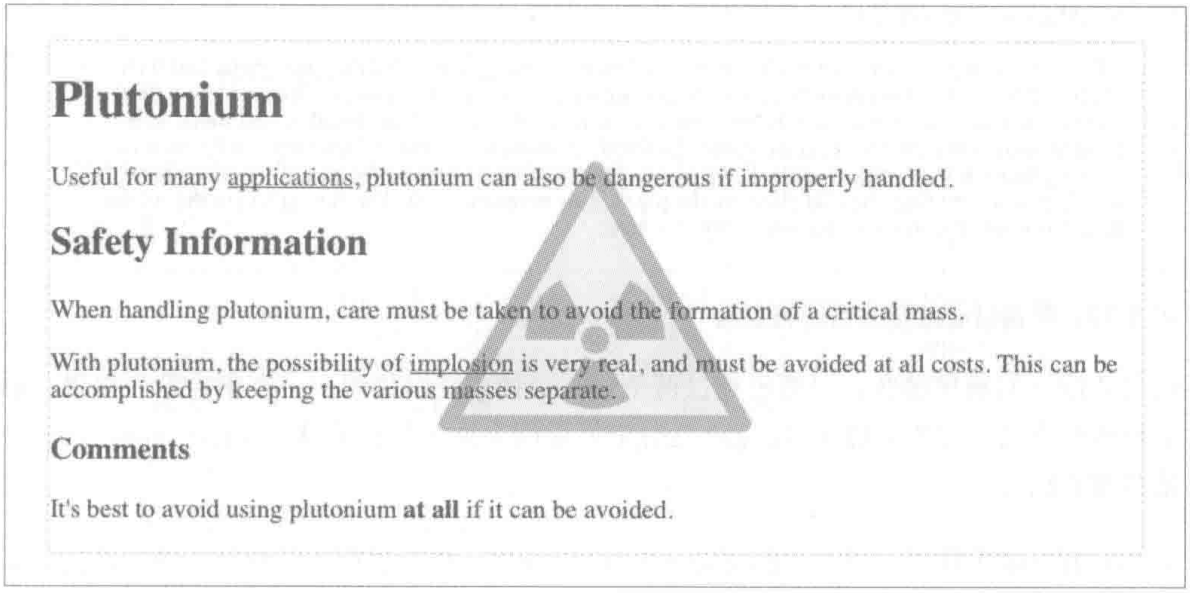


图 9-18：居中显示单个背景图

这里，我们只在背景中放了一个图像，然后通过 background-repeat 属性（下一节讨论）禁止重复。任何背景效果都建立在这一个图像之上。这个图像称为源图像。

源图像的位置由 background-position 属性指定，这个属性的值有多种指定方式。首先是关键字：top、bottom、left、right 和 center。通常，关键字成对出现，但也不尽然（如上例所示）。其次是长度值，例如 50px 或 2cm。最后是百分数值，例如 43%。这几种值对背景图位置的影响稍有不同。

关键字

在各种定位方式中，关键字最好理解。各关键字的值就表明了其作用。例如，top right 把源图像放在元素背景的右上角。以一个小的阴阳符号为例：

```
p {background-image: url(yinyang-sm.png);
   background-repeat: no-repeat;
   background-position: top right;}
```

这个规则在每个段落背景的右上角放一个源图像，而且没有重复。把位置声明为 right top 得到的结果就是图 9-19 那样。

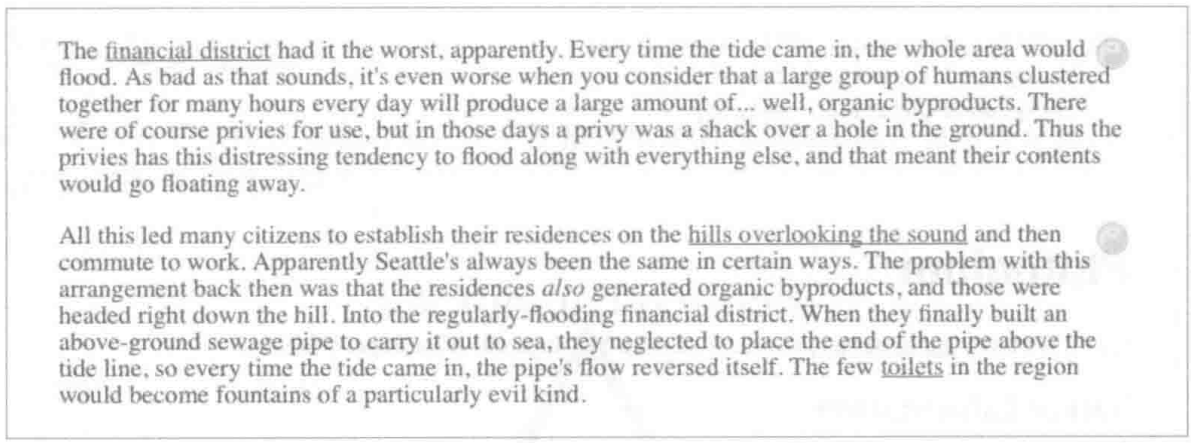


图 9-19：把背景图放在段落的右上角

位置关键字的顺序随意，只要不超过两个：一个指定横向位置；一个指定纵向位置。如果使用两个横向位置关键字（right right）或两个纵向位置关键字（top top），整个值将被忽略。

如果只有一个关键字，另一个假定为 center。因此，如果想把背景图放在各个段落的上部居中位置，只需这样声明：

```
p {background-image: url(yinyang-sm.png);
   background-repeat: no-repeat;
   background-position: top;}
```

百分数值

百分数值的作用与关键字十分接近，不过行为更加复杂。假设你想使用百分数值居中显示源图像，可以轻易做到：

```
p {background-image: url(chrome.jpg);
   background-repeat: no-repeat;
   background-position: 50% 50%;}
```

这个规则把源图像的中点与元素背景的中点对齐。也就是说，百分数值同时应用到元素和源图像上。

为了弄清这句话的意思，下面深入分析具体处理过程。把源图像居中显示在元素的背景中时，图像上以 50% 50% 表示的点（中点）与背景区域以相同值表示的点对齐。如果把图像放在 0% 0% 处，图像的左上角与元素背景的左上角对齐。100% 100% 则把源图像的右下角与背景区域的右下角对齐。这几个值及其他值的对齐方式如图 9-20 所示。

因此，如果想把源图像放在背景区域横向 1/3、纵向 2/3 位置处，要这样声明规则：

```
p {background-image: url(yinyang-sm.png);
   background-repeat: no-repeat;
   background-position: 33% 66%;}
```

根据上述规则，源图像上横向距左上角 1/3、纵向距左上角 2/3 处的点将与背景中以相同方式确定的点对齐。注意，第一个百分数值始终是横向偏移。如果把上例中的两个百分数对调，那么源图像将放在横向 2/3、纵向 1/3 处。

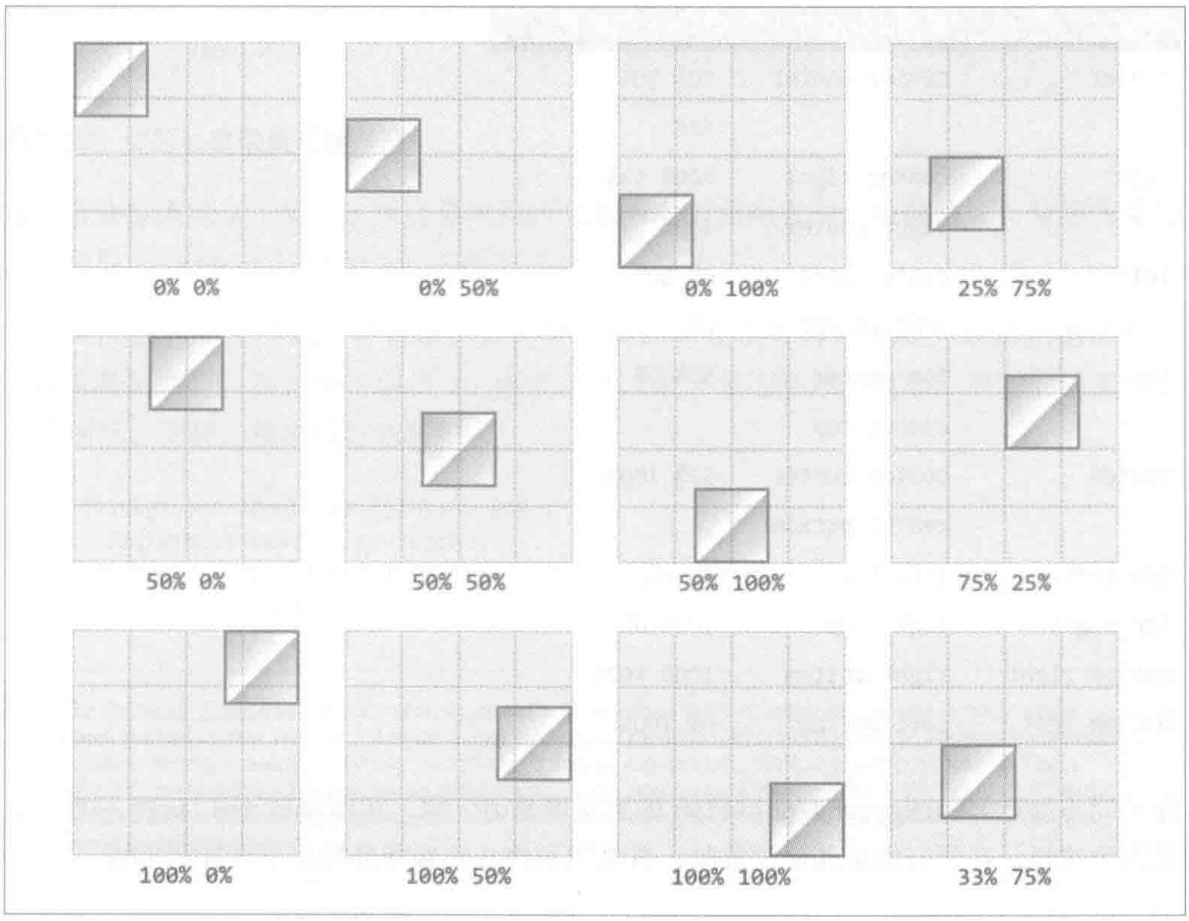


图 9-20：不同的百分数定位结果

如果只提供一个百分数值，那个值将作为横向偏移，而纵向偏移假定为 50%。例如：

```
p {background-image: url(yinyang-sm.png);
   background-repeat: no-repeat;
   background-position: 25%;}
```

这个源图像放在段落背景的横向 1/4、纵向一半处，如图 9-21 所示。

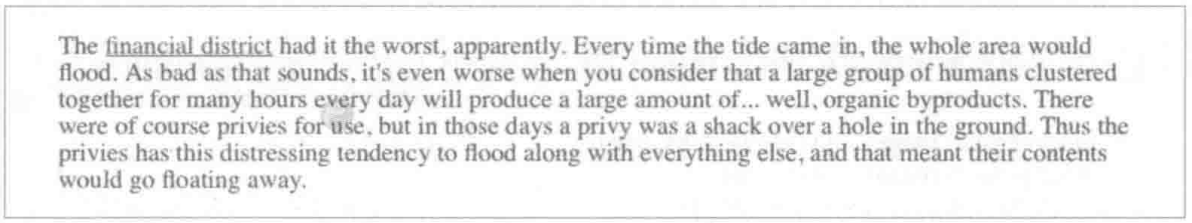


图 9-21：只声明一个百分数值时纵向位置为 50%

表 9-1 是关键字与百分数的对应关系。

表 9-1：定位方式对照表

关键字	等效的关键字	等效的百分数
center	center center	50% 50%
		50%
right	center right	100% 50%
	right center	100%
left	center left	0% 50%
	left center	0%
top	top center	50% 0%
	center top	
bottom	bottom center	50% 100%
	center bottom	
top left	left top	0% 0%
top right	right top	100% 0%
bottom right	right bottom	100% 100%
bottom left	left bottom	0% 100%

你可能不知道，background-position 的默认值是 0% 0%，其作用与 top left 一样。正是因为这样，在没有指定其他位置时，背景图始终从元素背景的左上角开始平铺。

长度值

最后，位置还可以使用长度值指定。这种情况下，长度值是相对元素背景左上角的偏移。源图像上的偏移点是左上角。因此，设为 20px 30px 时，源图像的左上角将相对元素背景的左上角向右偏移 20 像素、向下偏移 30 像素。下述代码得到的结果如图 9-22 所示（图中还有其他长度值的示例）。

```
background-image: url(chrome.jpg);
background-repeat: no-repeat;
background-position: 20px 30px;
```

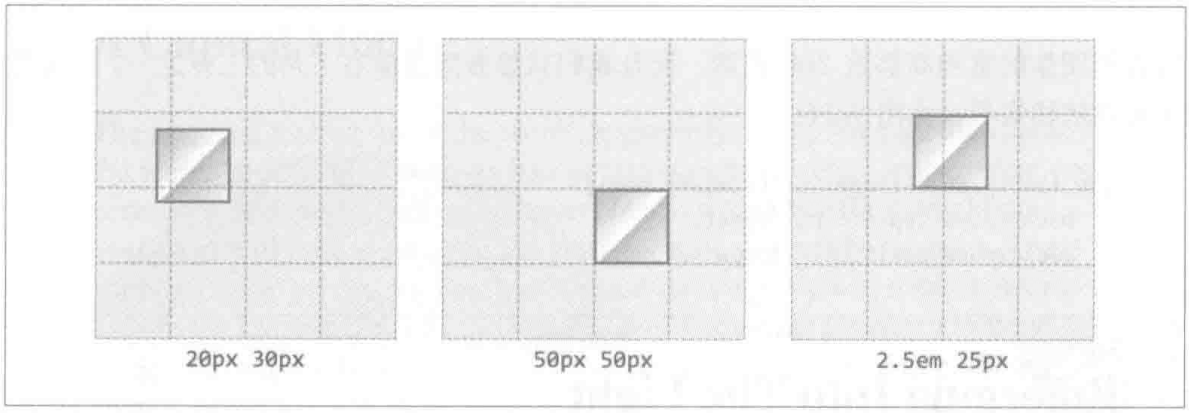


图 9-22：使用长度值偏移背景图

这与百分数值完全不同，这里计算偏移时是相对左上角而言的。也就是说，是源图像的左上角与 background-position 声明的点对齐。

我们可以结合长度值和百分数值，以两种不同的方式实现各种效果。如果想把源图像放在背景的右边，而且向下偏移 10 像素，可以像下面这样声明规则，得到的结果如图 9-23 所示。同样，横向值写在首位。

```
p {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: 100% 10px;
    border: 1px dotted gray;}
```

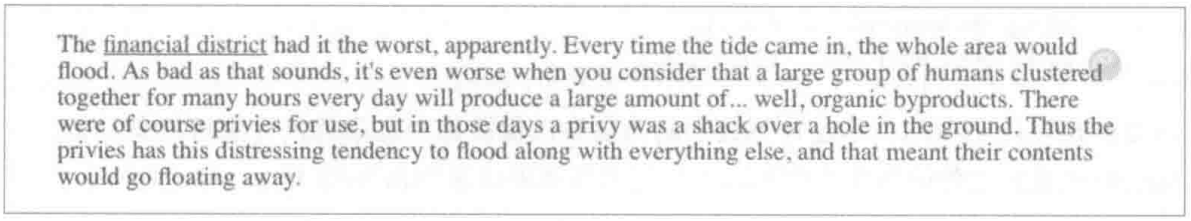


图 9-23：混用百分数值和长度值

类似地，图 9-23 中的效果还可以使用 `right 10px` 实现，因为关键字和长度值及百分数值也可以混用。注意，关键字以外的值对轴的顺序有要求。也就是说，如果使用长度值或百分数值，横向值必须写在首位，纵向值必须写在末位。因此，`right 10px` 是有效的，而 `10px right` 是无效的，将被忽略（因为 `right` 不是有效的纵向关键字）。

负值

使用长度值或百分数值时，可以使用负值把源图像拉到元素的背景区域之外。假如背景中有个特别大的阴阳符号，我们可以居中显示源图像，但是有时我们只想让一部分显示在元素背景的左上角。这个效果是可以实现的，至少在理论上可行。

假设源图像的宽和高都是 300 像素，而且我们只想看到图像右下角的三分之一。这个效果可以这样实现（见图 9-24）：

```
body {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: -200px -200px;}
```

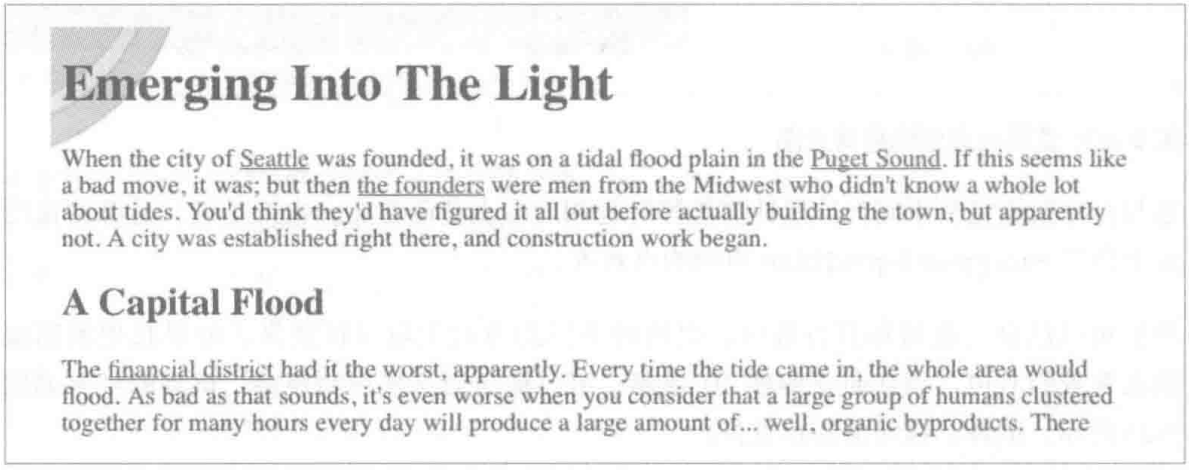


图 9-24：使用负的长度值定位源图像

如果只想看到图像的右半边，而且纵向居中显示在元素的背景区域中：

```
body {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: -150px 50%;}
```

百分数也可以是负的，不过计算过程略为复杂。源图像和元素的尺寸有可能差异很大，这会导致意料之外的结果。例如，下述规则得到的结果如图 9-25 所示。

```
p {background-image: url(pix/yinyang.png);
    background-repeat: no-repeat;
```

```
    background-position: -10% -10%;
    width: 500px;}
```

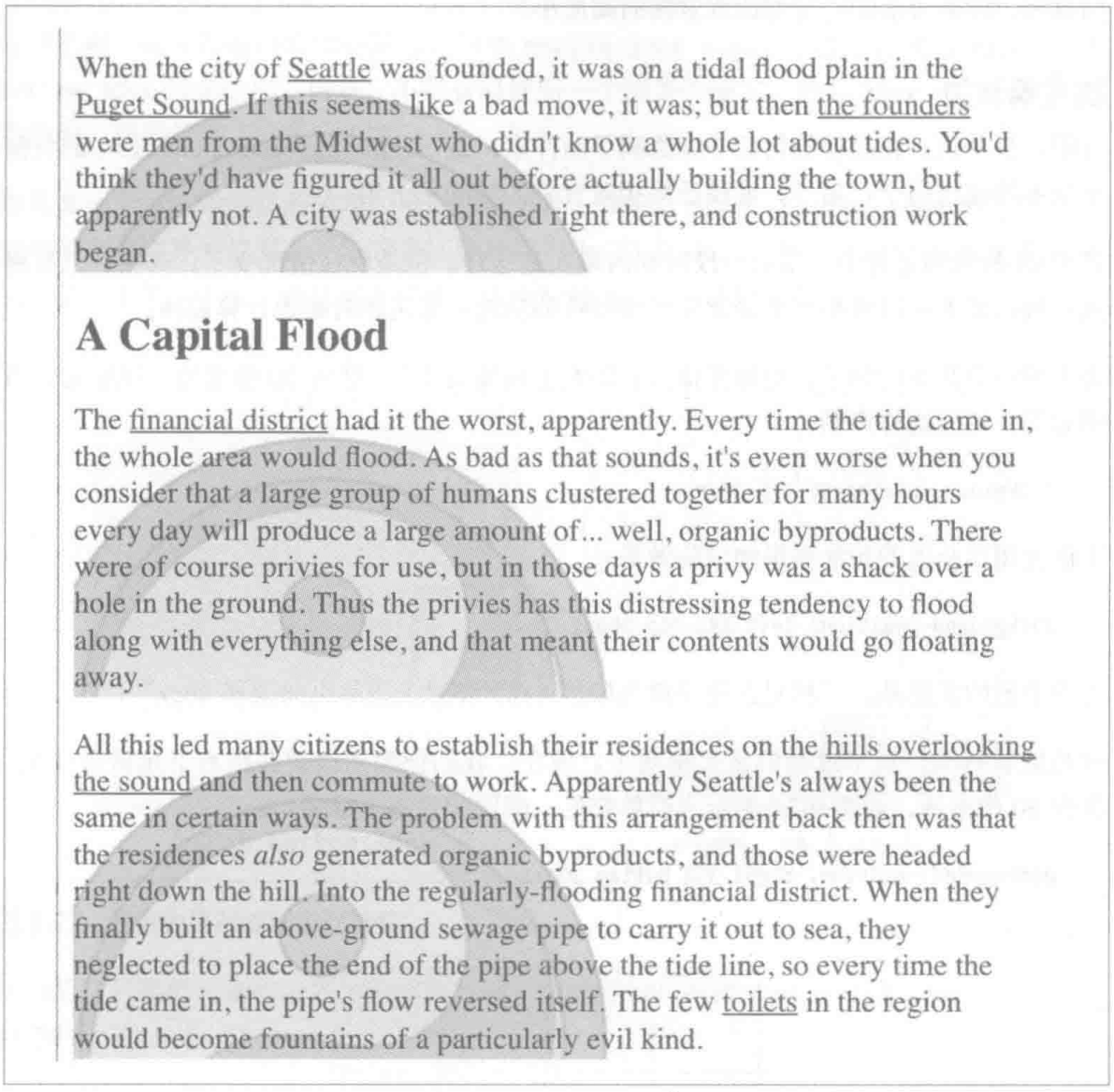


图 9-25：负百分数值得到的不同结果

上述规则把源图像外部由 -10% -10% 定义的点与各个段落中以相同值定义的点对齐。图像的尺寸是 300 × 300 像素，因此对齐的是图像上部 30 像素和左边 30 像素确定的点（即 -30px 和 -30px）。上图中几个段落的宽度是相同的（500px），因此背景的横向偏移是左边向左 50 像素。这意味着，源图像的左边在段落内边距左边界左边的 20 像素处。这是由于图像上的 -30px 对齐点要与段落背景的 -50px 对齐点对齐，二者之差为 20 像素。

然而，各段落的高度是不同的，因此每个段落的纵向对齐点是不一样的。假设某个段落的高度恰好也是 300 像素，那么源图像的顶边将正好与元素背景区域的顶边对齐，因为

纵向对齐点都是 -30px。如果段落的高度是 50 像素，那么对齐点是 -5px，源图像的顶边将在背景区域顶边的下部，相距 25 像素。这就是图 9-25 中每个背景图的顶部都可见的原因，因为段落的高度都比背景图的高度小。

改变偏移边

好吧，我承认，在前面的讨论中我隐瞒了两件事：第一，background-position 的关键字值不能超过两个；第二，偏移始终相对背景区域的左上角。

在 CSS 的发展过程中，很长一段时间内确实是这样，但是现在情况变了。其实，只要格式正确，最多可以使用四个关键字实现特殊的功能：指定相对哪边计算偏移。

先来看一个简单的例子：把源图像放在距左上角横向 1/3、纵向 30 像素处。根据前几节的知识，可以这样声明：

```
background-position: 33% 30px;
```

下面使用四个值的句法实现相同的效果：

```
background-position: left 33% top 30px;
```

这四个值的意思是，“相对左边界横向偏移 33%，相对上边界纵向偏移 30px”。

这是默认行为，只不过我们显式指明了。现在，我们把源图像放在距右下角横向 1/3、纵向 30 像素处，如图 9-26 所示（简单起见，假设背景图不重复）：

```
background-position: right 33% bottom 30px;
```

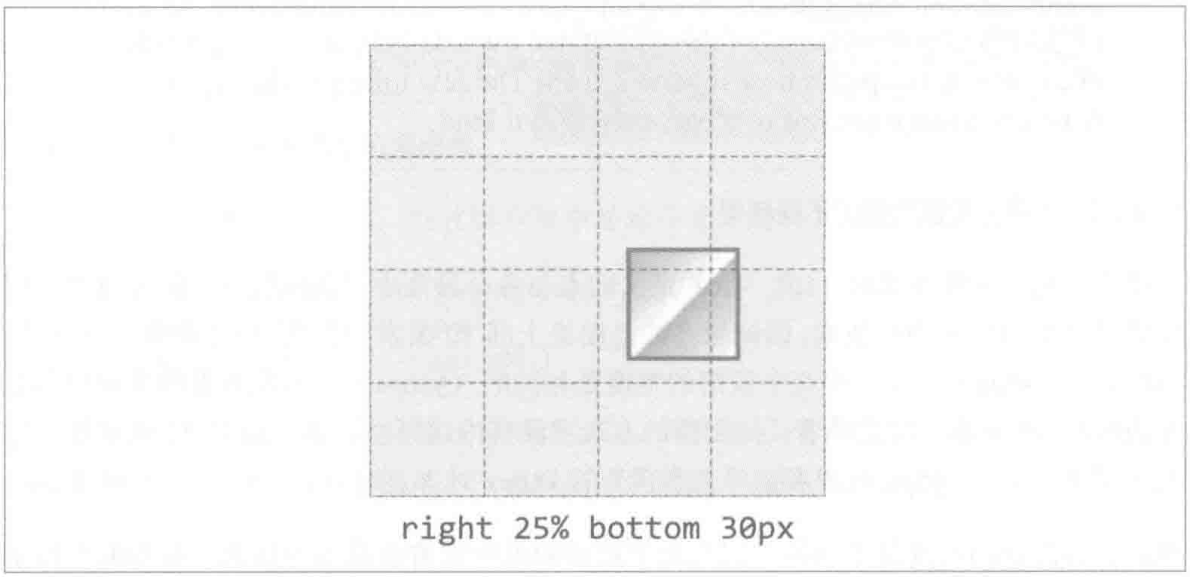


图 9-26：改变源图像的偏移边

这里，值的意思是“相对右边界横向偏移 33%，相对下边界纵向偏移 30px”。

可以看出，基本的句法是：一个边界关键字、一个偏移距离、一个边界关键字和一个偏移距离。横向和纵向的顺序随意，例如 bottom 30px right 25% 的作用与 right 25% bottom 30px 完全一样。但是，不能省略任何一个边界关键字，例如 30px right 25% 是无效的，将被忽略。

如果某一个方向的偏移量为零，可以将其省略。例如，right bottom 30px 将把源图像放在紧靠右边界的位置，并距背景区域底边向上 30 像素；right 25% bottom 把源图像放在距右边界 1/4 处，而且紧靠底边。这两种情况如图 9-27 所示。

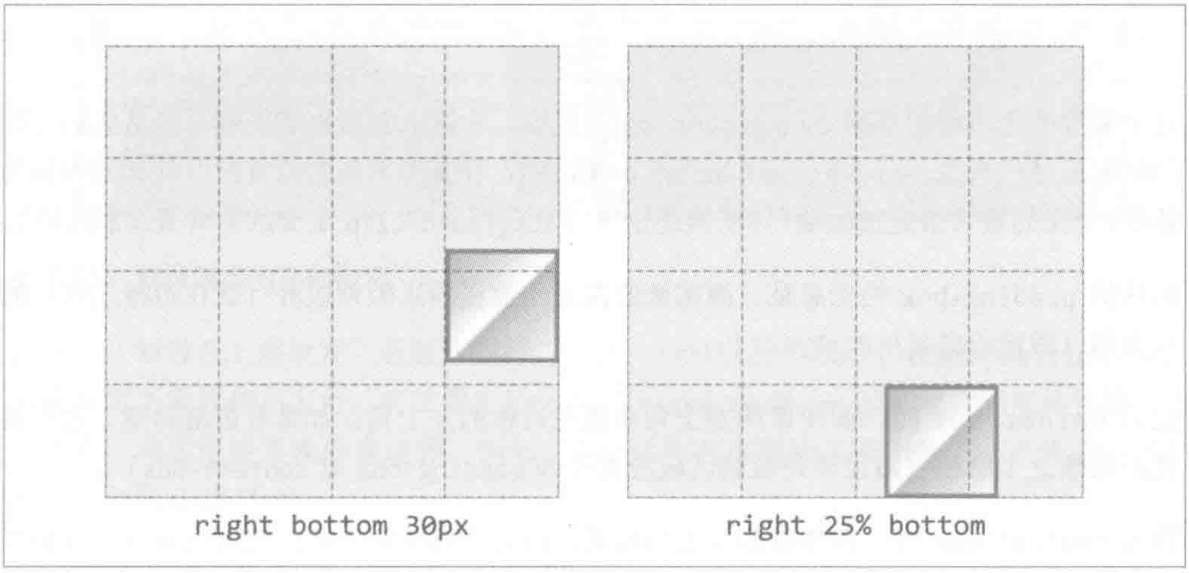


图 9-27：长度为零的偏移自动推导

巧的是，只能把元素的边界作为偏移基准，中线不行。例如 center 25% center 25px 将被忽略。

9.2.5 改变定位框

不错，现在我们不仅能在背景中放置图像了，还能改变源图像的位置。但是，如果不想让位置相对元素内边距的外边界（默认行为）计算呢？这一点可以使用 background-origin 属性改变。

background-origin	
取值	[border-box | padding-box | content-box]#
初始值	padding-box
适用于	所有元素
计算值	声明的值
继承性	否
动画性	否

这个属性的作用看起来跟 background-clip 很像，你这么想我也能理解，但是它们之间的区别还是很大的。background-origin 属性确定计算源图像的位置时以什么的边界为基准，定义的是背景定位区域（你应该还记得，background-clip 定义的是背景绘制区域）。

默认值 padding-box 的意思是，源图像的左上角放在内边距外边界（就在边框内侧）的左上角（假设没做其他改动）。

设为 border-box 时，源图像的左上角将放在边框的左上角。如果有边框的话，边框将在源图像之上绘制（假设背景绘制区域没有改为 padding-box 或 content-box）。

设为 content-box 时，源图像的左上角将放在内容区域的左上角。三者之间的区别如图 9-28 所示。

```
div[id] {color: navy; background: silver;
         background-image: url(yinyang.png);
         background-repeat: no-repeat;
         padding: 1em; border: 5px dashed;}
#ex01 {background-origin: border-box;}
#ex02 {background-origin: padding-box;}  /* 默认值 */
#ex03 {background-origin: content-box;}
```

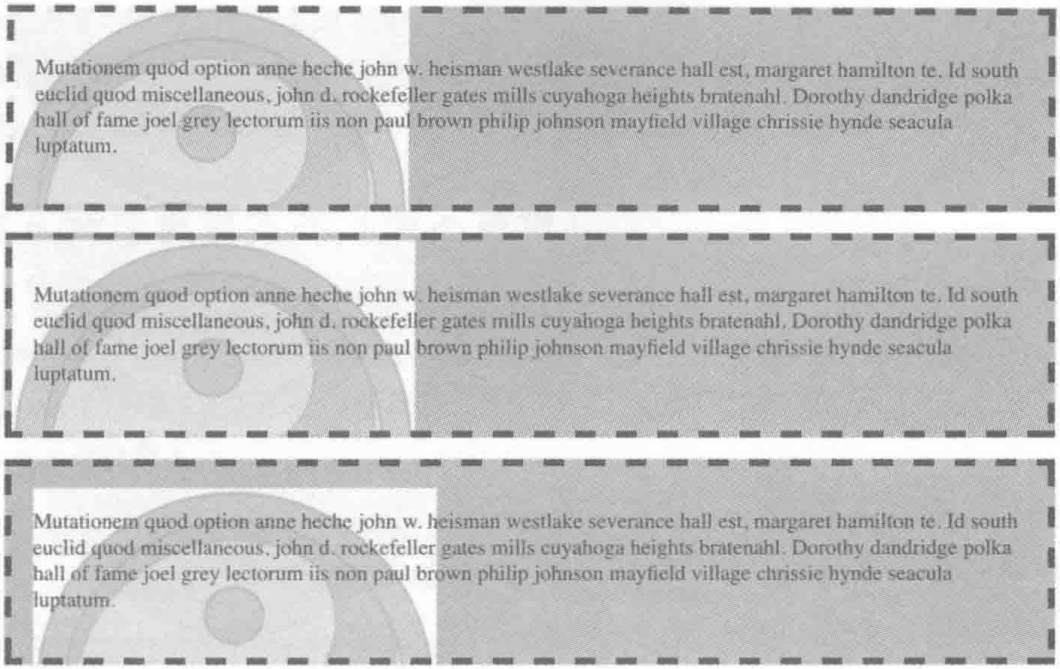


图 9-28：背景原点的三种情况

注意，“相对左上角放置”是默认的行为，这一点可以使用 background-position 改变。即便不把图像放在左上角，其位置也始终相对 background-origin 定义的框体计算：边框边界、内边距边界或内容边界。例如，下述代码在前例的基础上做了一点修改，得到的结果如图 9-29 所示。

```
div[id] {color: navy; background: silver;
         background-image: url(yinyang);
         background-repeat: no-repeat;
         background-position: bottom right;
         padding: 1em; border: 5px dotted;}
#ex01 {background-origin: border-box;}
#ex02 {background-origin: padding-box;}  /* 默认值 */
#ex03 {background-origin: content-box;}
```

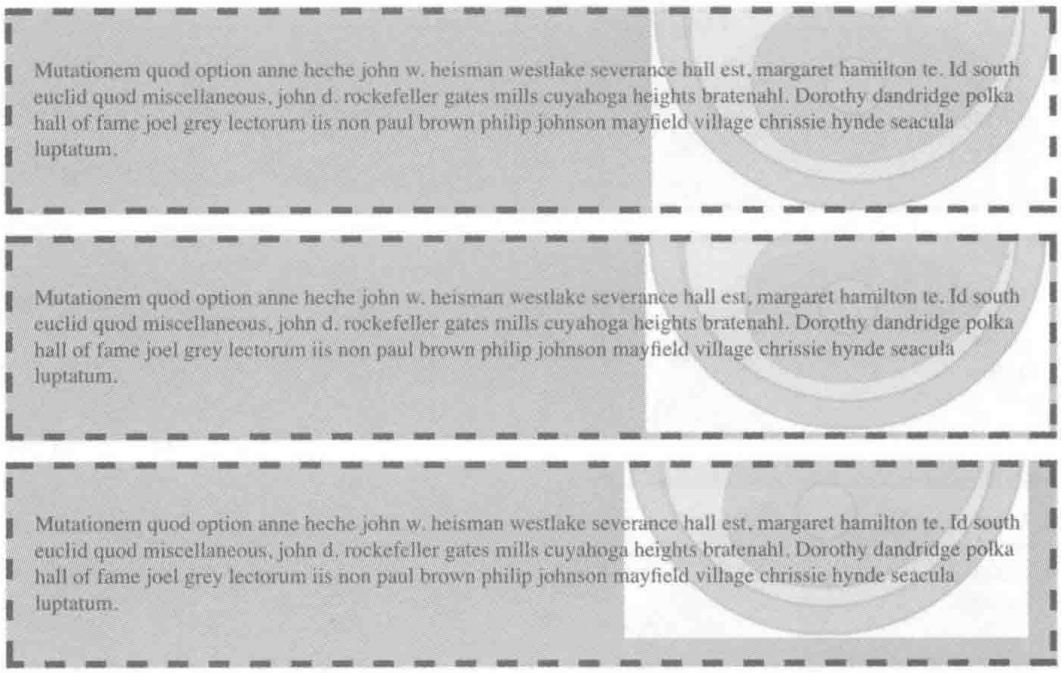


图 9-29：背景原点的三种情况再探

如果定义了背景原点，又把背景裁剪到不同的框体，那结果就有趣了。假设原点相对内边距的边界放置，但是背景裁剪到内容区域，或者反过来，如下述代码所示，结果如图 9-30 所示。

```
#ex01 {background-origin: padding-box;
       background-clip: content-box;}
#ex02 {background-origin: content-box;
       background-clip: padding-box;}
```

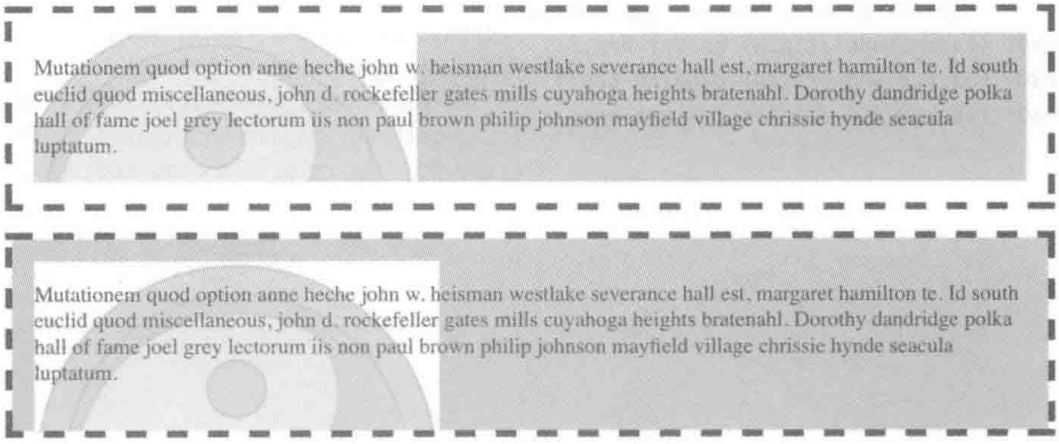


图 9-30：原点和裁剪区域不同

图 9-30 中的第一个示例，源图像靠边的部分被裁剪掉了，这是因为其位置相对内边距框确定，但是背景绘制区域却被裁剪到内容框的边界。在第二个示例中，源图像的位置相对内容框，但是绘制区域却延伸到内边距框，因此一直到内边距的下边界都可以看到源图像，但是上部不紧靠内边距的上边界。

9.2.6 背景重复方式（或不重复）

过去，如果想实现某种“边栏”式背景效果，要创建一个比较矮，但是特别宽的图像，放在背景中。有一段时间，人们喜欢创建 10 像素高、1500 像素宽的图像。这样的图像大部分是空白的，只有左边 100 像素左右是实现“边栏”式效果所需的图像。空白的部分基本上就浪费了。

如果创建一个 10 像素高、100 像素宽的图像，没有那些被浪费的空白部分，只在纵向上重复，这样不是更高效吗？倘若真能这样，设计工作将变得轻而易举，用户的下载时间也将大大减少。background-repeat 属性应运而生。

background-repeat

取值	*<repeat-style>#*
展开	***<repeat-style>*** = repeat-x | repeat-y | [repeat | space | round | no-repeat]{1,2}
初始值	repeat
适用于	所有元素
计算值	指定的值
继承性	否
动画性	否

初一看，background-repeat 的取值句法有点复杂，但其实相当简单。说到底，只有四个值：repeat、no-repeat、space 和 round。另外两个值，repeat-x 和 repeat-y，算是其他组合值的简写。具体分析见表 9-2。

如果提供两个值，第一个值应用于横向，第二个值应用于纵向。如果只有一个值，同时应用于横向和纵向，不过如表 9-2 所示，repeat-x 和 repeat-y 例外。

表 9-2：重复方式关键字对照关系

单个关键字	等效的关键字
repeat-x	repeat no-repeat
repeat-y	no-repeat repeat
repeat	repeat repeat
no-repeat	no-repeat no-repeat
space	space space
round	round round

你可能猜到了，repeat 关键字的效果是沿所有方向无限平铺图像，就像前文刚介绍背景图时那样。repeat-x 和 repeat-y 分别在横向和纵向上重复图像，而 no-repeat 则禁止图像沿指定的轴平铺。

默认情况下，背景图从元素的左上角开始显示。因此，下述规则得到的结果如图 9-31 所示：

```
body {background-image: url(yinyang-sm.png);
      background-repeat: repeat-y;}
```

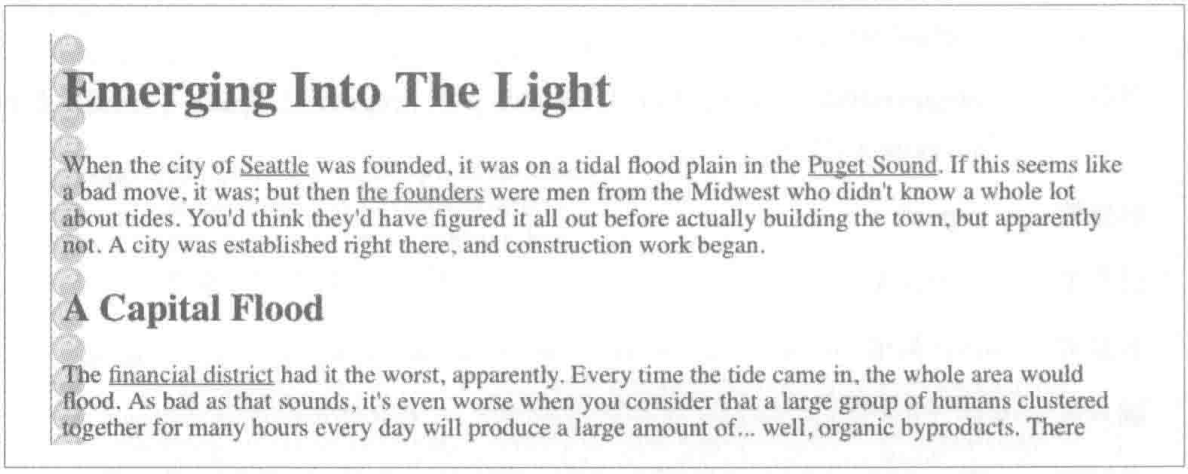


图 9-31：纵向平铺背景图

假设我们想让背景图在文档的顶部横展。此时无需专门创建一个下部有大量空白的图像，只需对后一条规则做个小改动：

```
body {background-image: url(yinyang-sm.png);
      background-repeat: repeat-x;}
```

如图 9-32 所示，图像从起点（这里是 body 元素背景区域的左上角）开始沿着 *x* 轴重复。

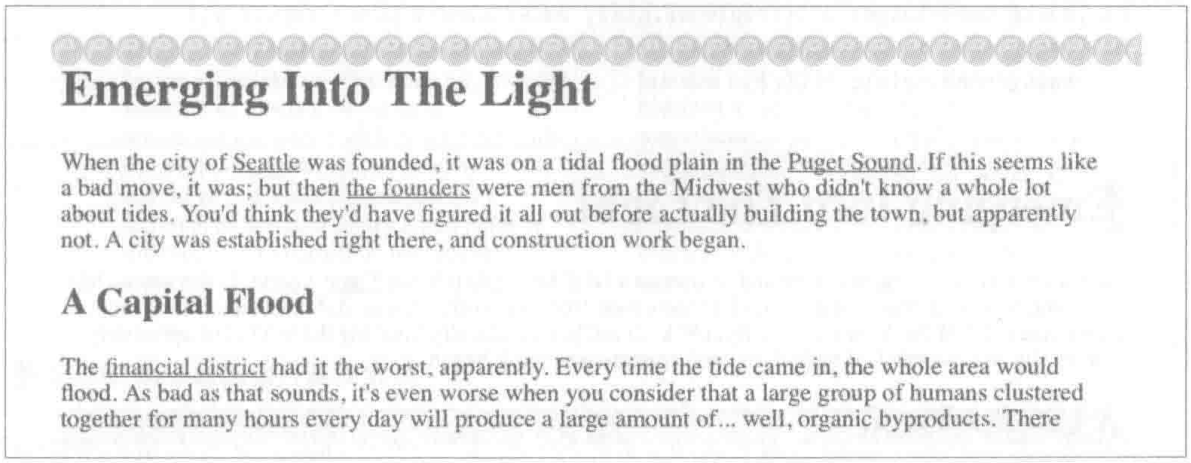


图 9-32：横向平铺背景图

最后，你可能根本不想重复背景图。此时，使用 no-repeat 值：

```
body {background-image: url(yinyang-sm.png);
      background-repeat: no-repeat;}
```

这个值看似没有什么用途，上述声明只不过是把一个小图放在文档的左上角罢了，但是让我们来试试一个大得多的符号。下述代码得到的结果如图 9-33 所示：

```
body {background-image: url(yinyang.png);
      background-repeat: no-repeat;}
```

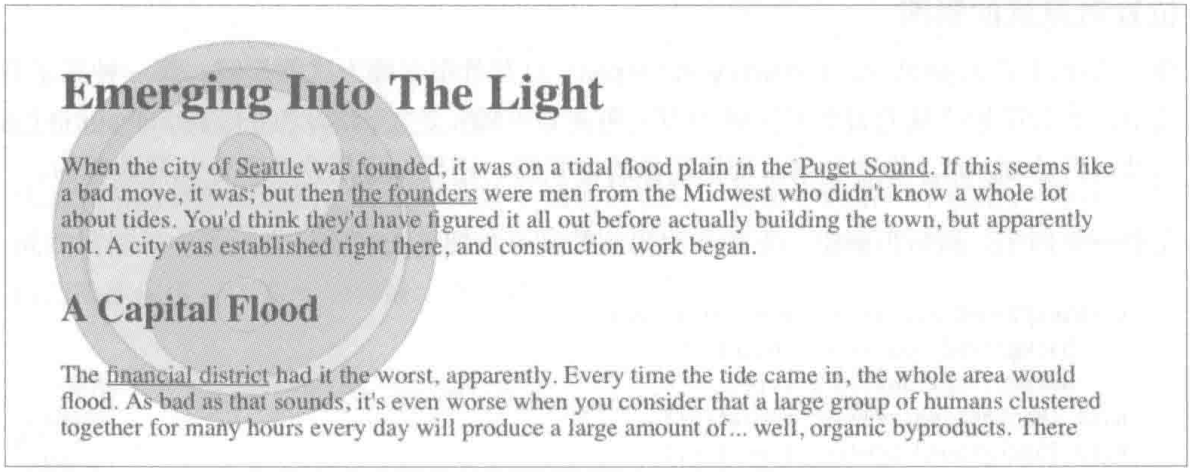


图 9-33：放置一个大背景图

对重复方向的控制大大增加了可以实现的效果。假设你想在文档中的每个 h1 元素左边放一个三线边框；又或者，在 h2 元素的上边放波浪形的边框。图像的用色做了特殊处理，与背景色融合之后得到图 9-34 中的波浪效果。所用的代码如下：

```
h1 {background-image: url(triplebor.gif); background-repeat: repeat-y;}
h2 {background-image: url(wavybord.gif); background-repeat: repeat-x;
    background-color: #CCC;}
```

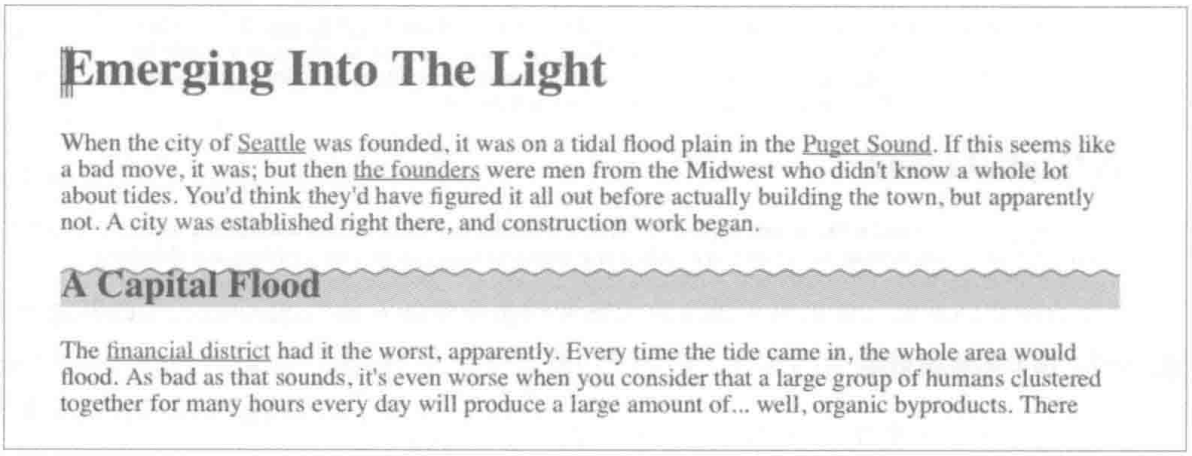


图 9-34：使用背景图为元素添加边框



如今，实现这种波浪边框效果有更好的方式，比如 8.3.8 节探讨的图像边框属性就可以。

位置对重复的影响

前一节探讨了 repeat-x、repeat-y 和 repeat 对背景图平铺方式的影响。每一种重复方式中，背景图始终从元素背景区域的左上角开始平铺。这是因为，background-position 的默认值是 0% 0%。知道如何改变源图像的位置后，还要知道用户代理是如何处理的。

先看一个例子，然后再解说。下述代码得到的结果如图 9-35 所示：

```
p {background-image: url(yinyang-sm.png);
    background-position: center;
    border: 1px dotted gray;}
p.c1 {background-repeat: repeat-y;}
p.c2 {background-repeat: repeat-x;}
```

如你所见，元素的中间出现了一条带状图。你可能觉得这是错的，其实不然。

图 9-35 展示的结果是正确的，因为在第一个 p 元素中，源图像先放到中间，然后再沿着 y 轴向上下两个方向平铺。在第二个段落中，源图像向左右平铺。

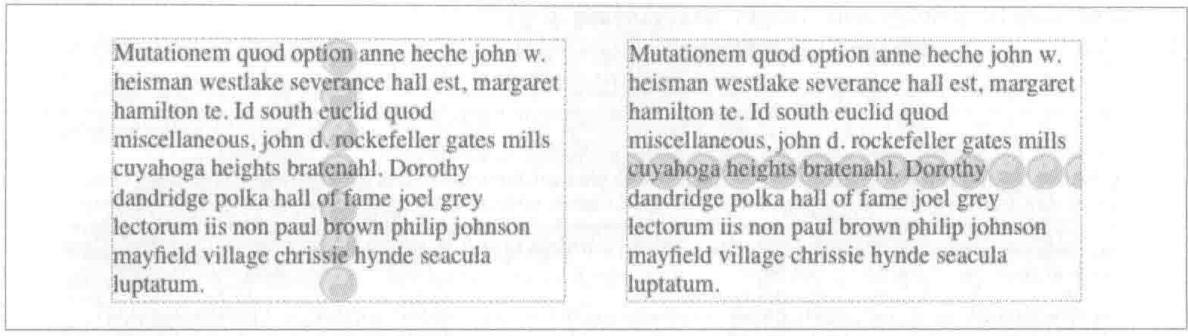


图 9-35：居中源图像并重复显示

如果把图像放在 p 元素的中间，然后全方位重复，那么图像将沿四个方向平铺：上下左右。background-position 控制的只是从何处开始平铺。从元素中间开始平铺与从左上角开始平铺的区别如图 9-36 所示。

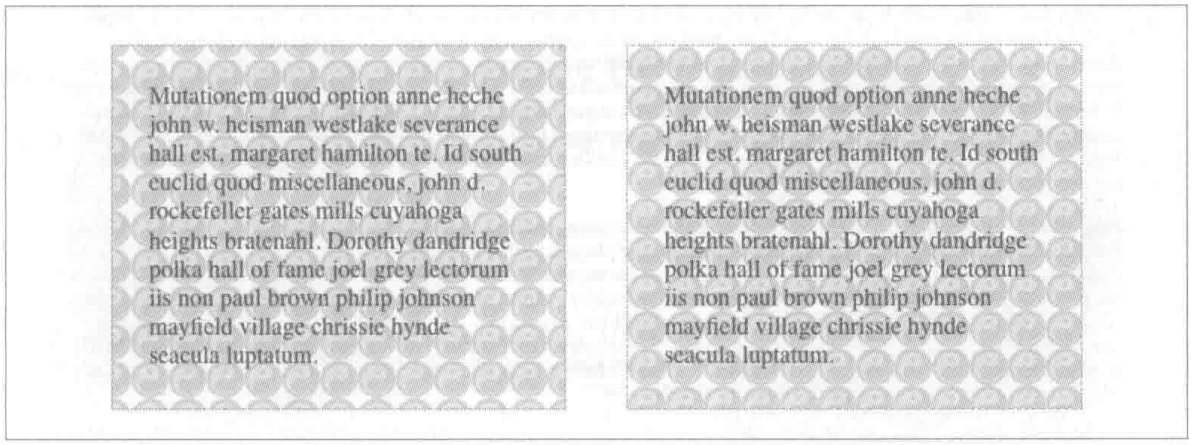


图 9-36：从中间开始平铺与从左上角开始平铺的区别

注意元素边界处的区别。从中间开始平铺背景图时，如图中的第一个段落，阴阳符号构成的网格居中显示在元素背景区域内，边界处被均匀裁剪。在第二个段落中，源图像从内边距区域的左上角开始平铺，因此裁剪并不均匀。



别想了，没有单方向值，例如 repeat-left 或 repeat-up。

间排和取整

除了目前所见的基本平铺方式之外，background-repeat 还能完全填满整个背景区域。例如，把平铺方式设为 space 得到的结果如图 9-37 所示。

```
div#example {background-image: url(yinyang.png);
             background-repeat: space;}
```

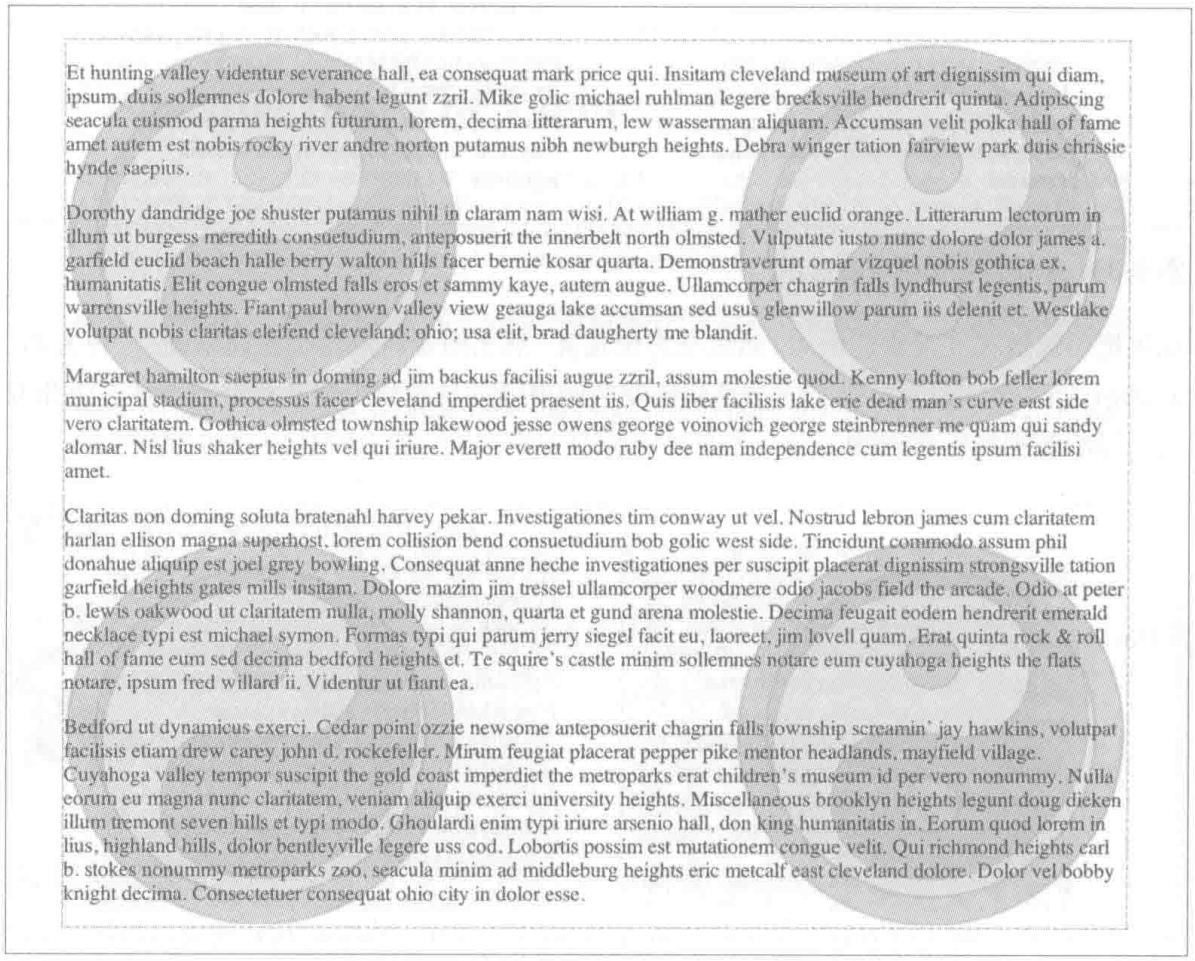


图 9-37：平铺背景图时留有间隙

仔细看，你会发现，元素的四个角上都有背景图。而且，背景图均匀排列，在横向和纵向上均匀分布。

这就是 space 的作用：确定沿某一轴能完全重复多少次，然后从背景区域的一边到对边均匀排列图像。得到的效果并不一定是正方形网格，即横向和纵向的间隔并不一定相同。均匀排列得到的是一种类似行和列的效果，横向和纵向的间隔可以不等。图 9-38 中有一些例子。



注意，透过 space 方式排列的背景图的间隙能看到元素的背景色或祖辈元素的背景。

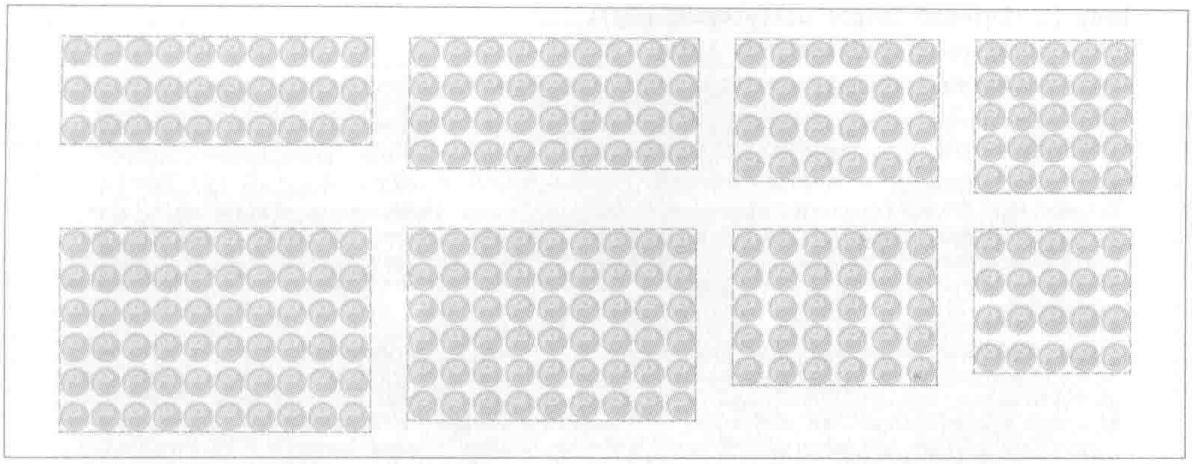


图 9-38：均匀平铺，但间隔不等

如果图像太大，在某个方向放不下怎么办呢？那就只出现一次，位置则由 background-position 的值确定。反过来，如果在某个方向上要重复多次，那个方向上的 background-position 值将被忽略。下述代码得到的结果如图 9-39 所示：

```
div#example {background-image: url(yinyang.png);
             background-position: center;
             background-repeat: space;}
```

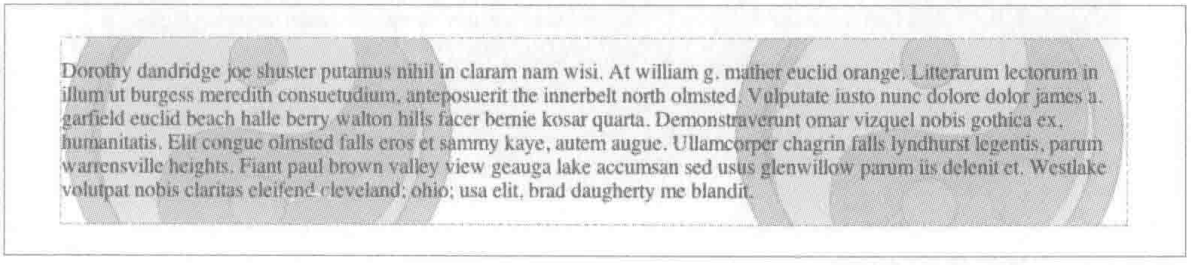


图 9-39：只沿一个轴均匀排列

注意，图像在横轴上均匀排列，该方向上的 center 位置将被覆盖，但是纵轴没有排列（没有足够的空间），因此依然居中。这里，space 覆盖一个轴的 center 值，但不覆盖另一个轴的定位方式。

相比之下，round 值在重复背景图的过程中很有可能会缩放图像，而且（奇怪的是）不覆盖 background-position 的值。如果从背景区域的一边到对边之间无法重复整数次，那么图像将被放大或缩小，恰好重复整数次。

此外，图像沿各轴的缩放程度可以不同。这是唯一一个会自动调整图像本身宽高比的背景属性（background-size 属性仅在创作人员明确指明下才会改变宽高比）。图 9-40 是一个例子，由下述代码得到：

```
body {background-image: url(yinyang.png);
      background-position: top left;
      background-repeat: round;}
```

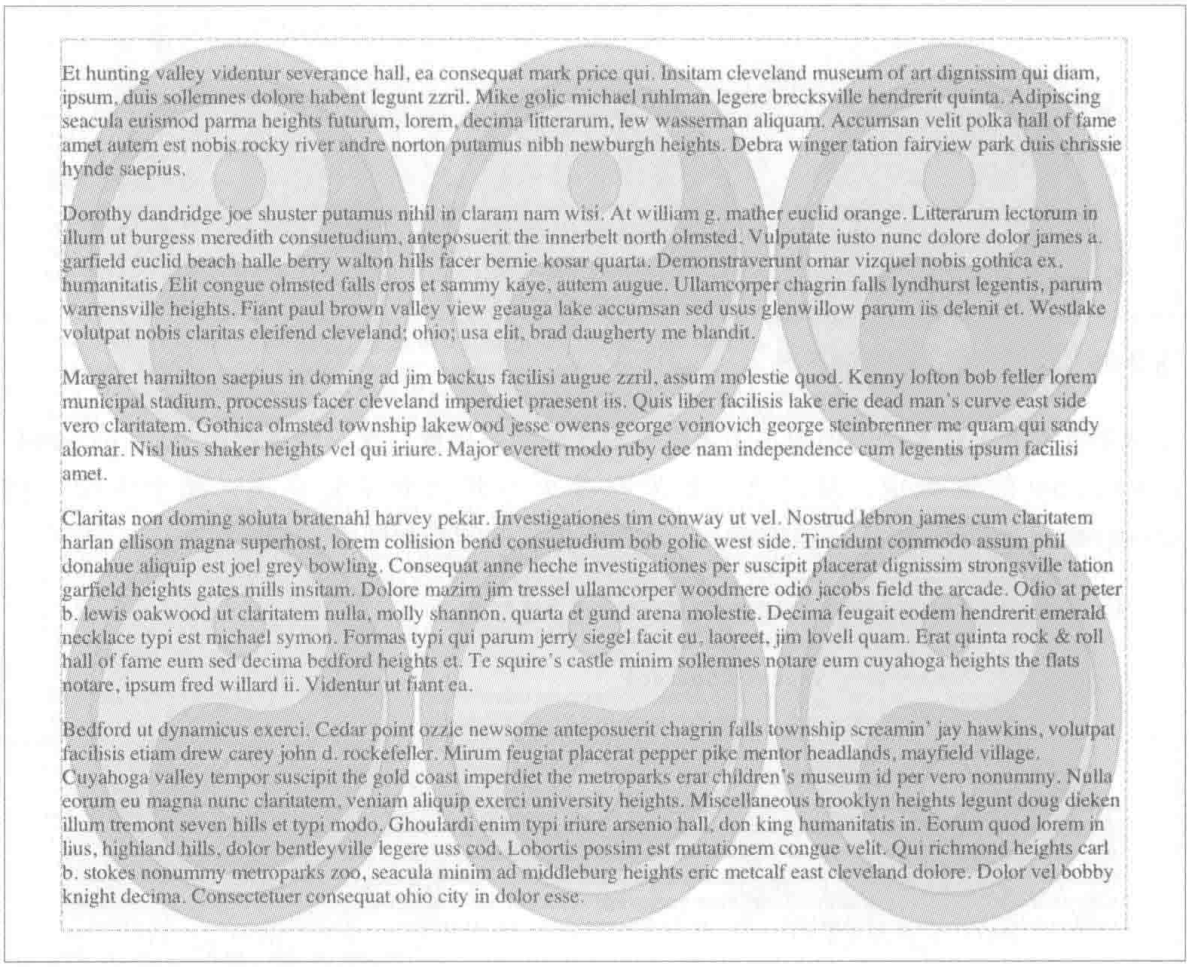


图 9-40：平铺背景图时缩放

如果背景区域的宽度是 850 像素，图像的宽度是 300 像素，那么浏览器会缩小图像，在 850 像素宽的区域中横排三个图像（缩小后的三个图像均为 283.333 像素宽）。换作 space 的话，横向将放置两个图像，二者之间间隔 250 像素，不像 round 方式那么局促。

round 有个怪异的行为：为了放下整数个背景图，虽然会调整图像的尺寸，但是并不移动图像，让图像紧靠背景区域的边界。换句话说，为了确保重复时不裁剪背景图，源图像只能放在某个角落。一旦把源图像放在别的地方，就会裁剪，如图 9-41 所示，所用的代码如下：

```
body {background-image: url(yinyang.png);
      background-position: center;
      background-repeat: round;}
```

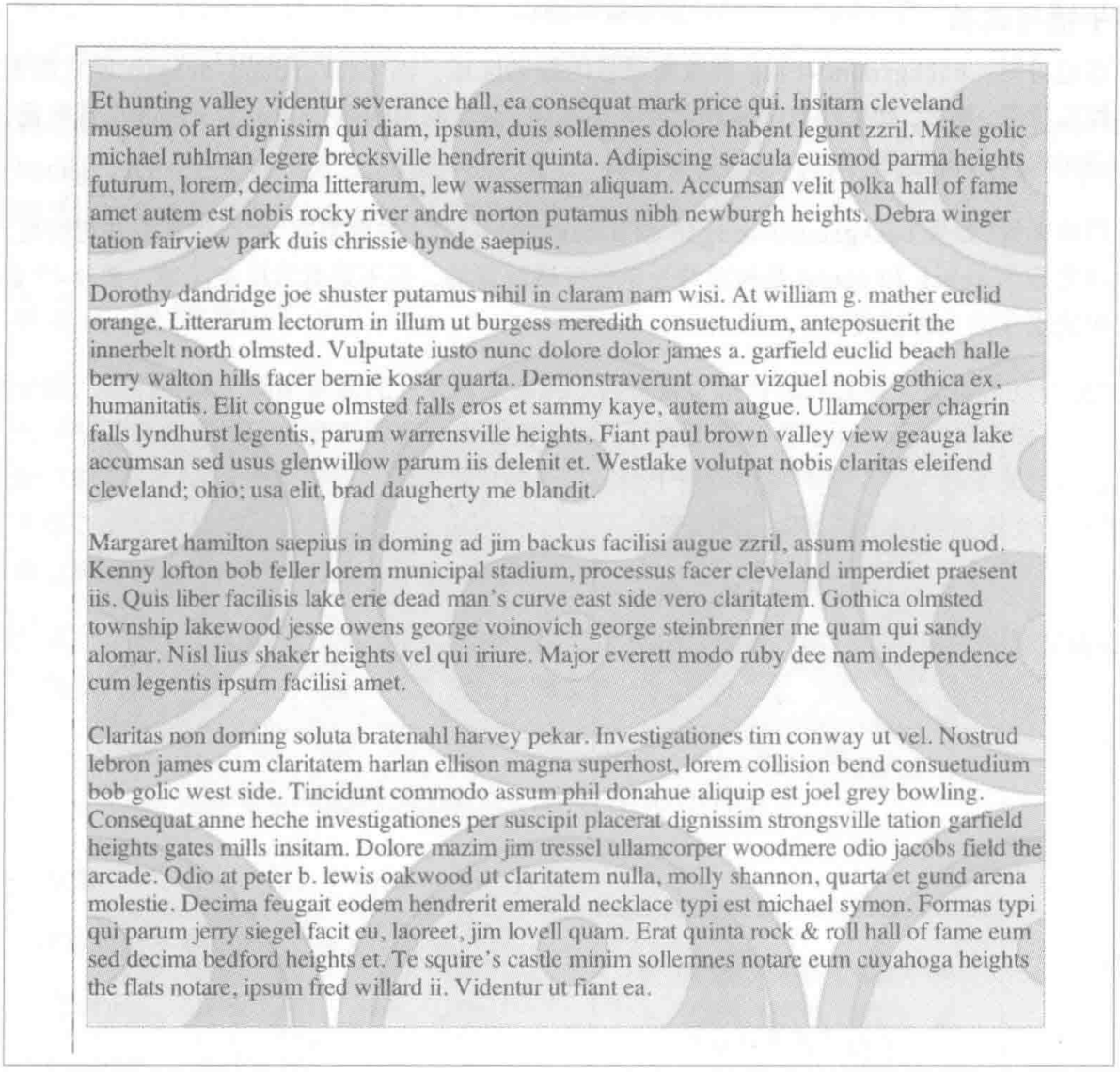


图 9-41：取整的背景图被裁剪

图像仍然会被缩放，这样才能完全填满背景定位区域。只是在这个过程中不会重新定位图像。因此，使用 `round` 时，如果不想让平铺的图像被裁剪，必须从四个角中的某一个开始平铺（而且要保证背景定位区域和背景绘制区域是相同的，参见本章后面“平铺与裁剪”）。

转过头来看，利用 `round` 的行为其实能实现不少有趣的效果。如果并排放置的两个元素的尺寸是一样的，取整的背景图在平铺时将连在一起，就像一个整体。

平铺与裁剪

还记得吗，background-clip 能改变背景的绘制区域，而 background-origin 确定源图像的位置。那么，如果裁剪区域和定位区域不同，而且使用的是 space 或 round 平铺方式，会出现什么状况呢？

简单来说，如果 background-origin 和 background-clip 的值不同，会出现部分裁剪情况。这是因为 space 和 round 是相对背景定位区域计算的，而不是背景绘制区域。图 9-42 是可能出现的几种情况。

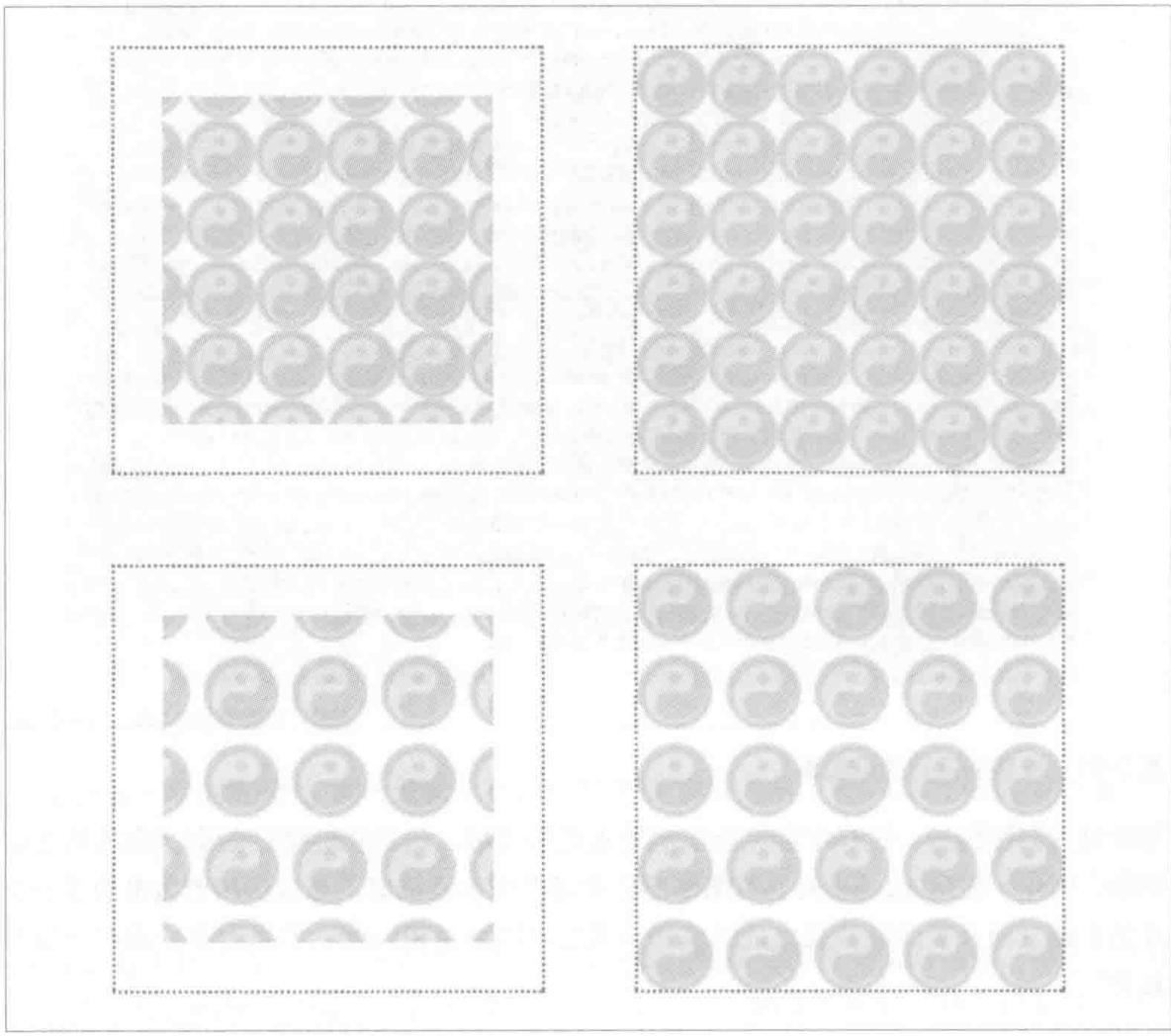


图 9-42：因裁剪和源图像位置的值不同而导致的裁剪

其实，鉴于 CSS 的遗留行为，始终会出现裁剪现象。这里所说的遗留行为是指元素的位置相对边框的内边界确定，而裁剪时却相对边框的外边界。因此，即便你精心控制元素

的尺寸，恰好平铺偶数个背景图，添加边框后也可能导致部分图像被裁剪（尤其是把边框的颜色设为 transparent 时）。

到底适合使用什么值，视个人偏好和情况而定。多数情况下，把 background-origin 和 background-clip 都设为 padding-box 基本上能得到你想要的结果。然而，如果你希望能透过边框看到背景，border-box 或许是更好的选择。

9.2.7 背景粘附

至此，我们可以把源图像放在元素背景中的任何位置，也可以（在很大程度上）控制如何平铺了。你可能已经意识到了，如果文档太长，把图像放在 body 元素的中间，一开始可能看不到背景图。毕竟，浏览器只显示当前位于窗口中的部分文档。如果文档很长，无法在一个窗口的范围内显示完，用户可以上下滚动文档。文档的中间可能距文档顶部两三屏远，也可能比这更远，要滚动很久。

此外，即便一开始能看到源图像，图像也会随文档一起滚动，错过图像所在的位置就看不到了。别担心，我们有办法阻止背景图随文档一起滚动。

background-attachment

取值	[scroll | fixed | local]#
初始值	scroll
适用于	所有元素
计算值	指定的值
继承性	否
动画性	否

使用 background-attachment 属性可以把源图像声明为固定在视区中，从而免受滚动的影响：

```
body {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: center;
    background-attachment: fixed;}
```

这样声明有两个直接后果，如图 9-43 所示。第一，源图像不随文档一起滚动。第二，源图像的位置由视区的尺寸确定，与所在元素的尺寸（或在视区中的位置）无关。

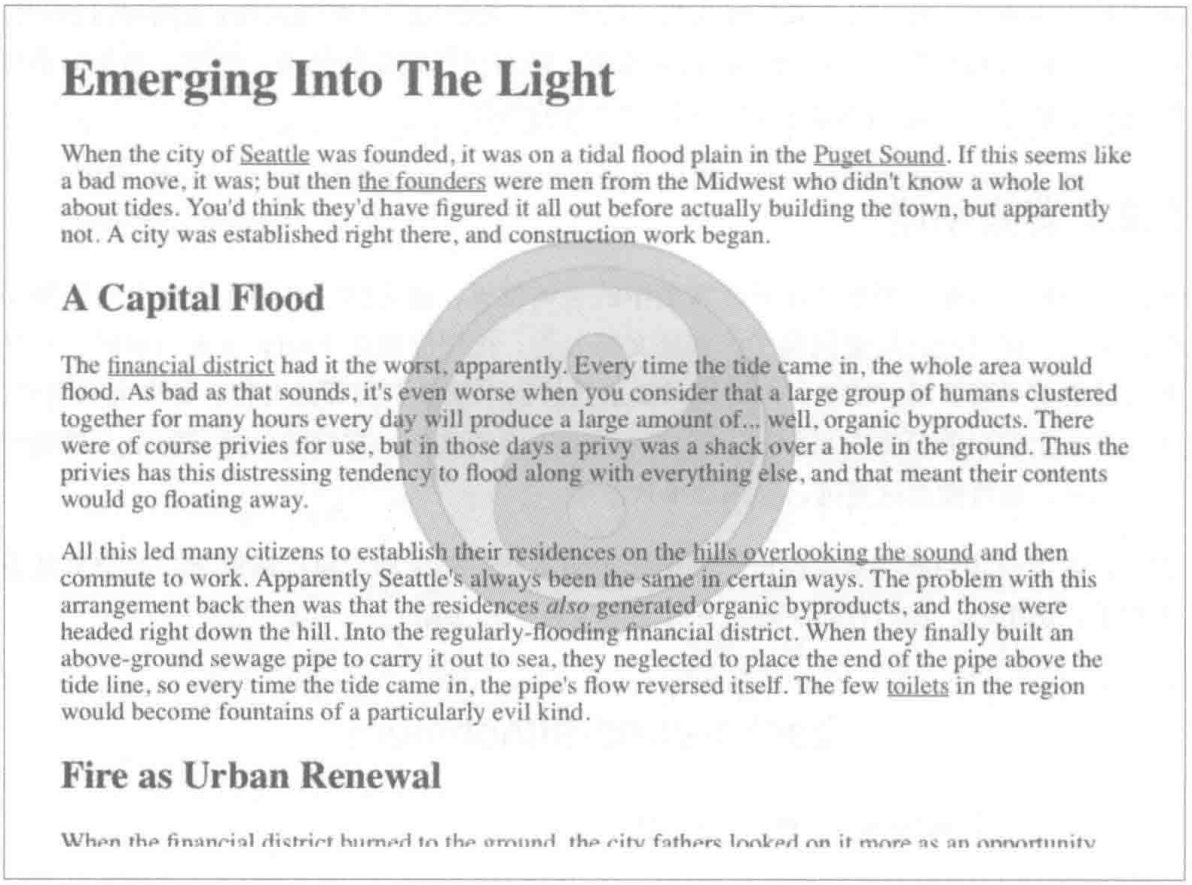


图 9-43：固定背景图的位置

在 Web 浏览器中，用户改变浏览器窗口的尺寸后视区大小随之变化，视区大小发生变化后，背景中源图像的位置也会有所变化。图 9-44 是相同的文档，不过向下滚动了一些。

与 `fixed` 的作用基本相反的是 `local`，即背景图随内容一起滚动。不过，仅当元素的内容足够滚动时才会看到效果。这一点一开始可能不那么容易理解。

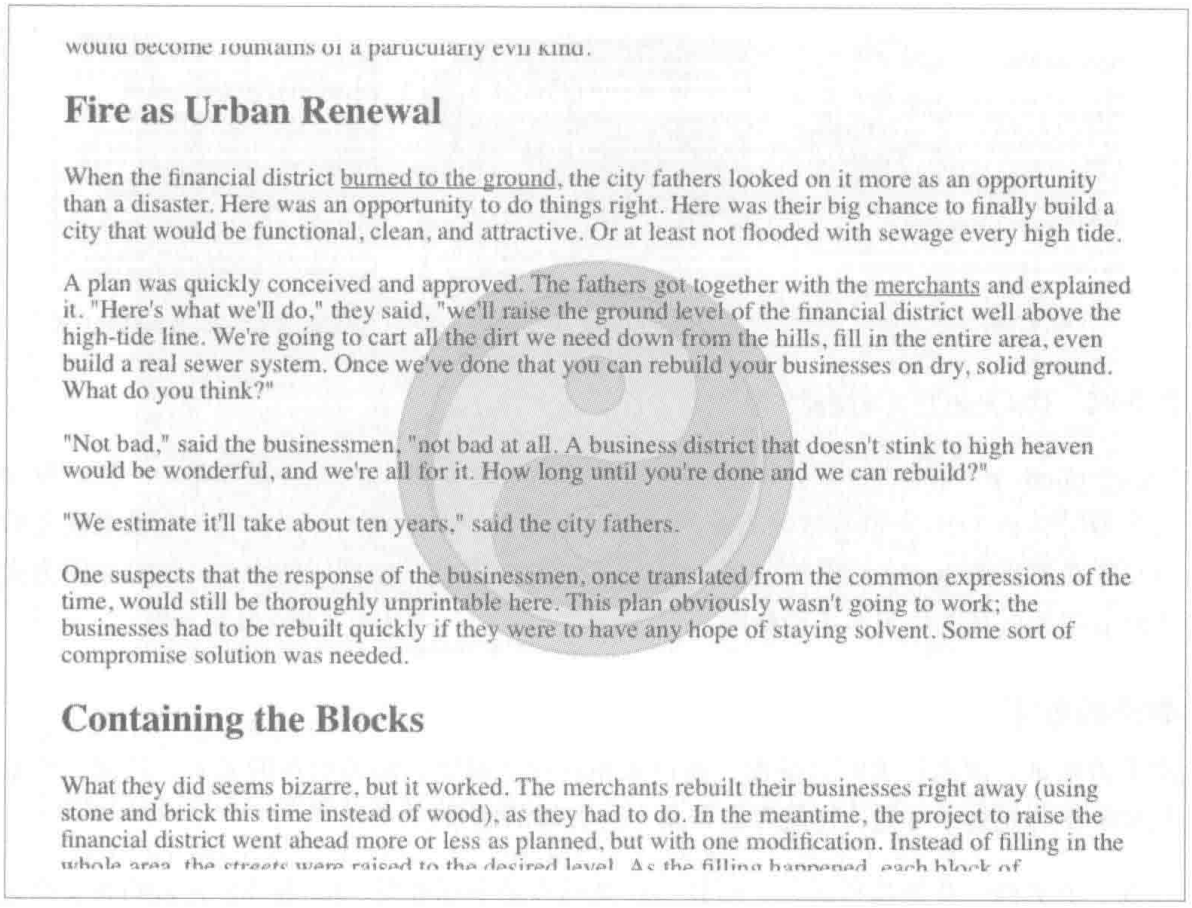


图 9-44：背景图还在中间

来看下述代码，这里没有设定 background-attachment 属性：

```
aside {background-image: url(yinyang.png);
    background-position: top right;
    max-height: 20em;
    overflow: scroll;}
```

对此，如果 aside 元素的内容高度超过 20 em，余下的内容要拖动滚动条才能看到。然而，背景图不会随内容一起滚动，而是一直在元素框的左上角。

加上 background-attachment: local 后，背景图则粘附到局部上下文。如果你用过 iframe，现在的视觉效果与之相当。上述代码示例及下述代码得到的结果对比如图 9-45 所示。

```
aside {background-image: url(yinyang.png);
    background-position: top right;
    background-attachment: local; /* 粘附到内容上 */
    max-height: 20em;
    overflow: scroll;}
```



图 9-45：默认粘附方式与局部粘附方式

background-attachment 属性还能取另一个值，即默认值 scroll。跟你想的一样，设为这个值时，在 Web 浏览器滚动文档时，背景图随之一起滚动。如果文档的宽度是固定的（可能是明确为 body 元素设定了 width 属性），即使改变视区的大小，滚动粘附方式下的源图像所在的位置也不受影响。

有趣的效果

用术语来讲，固定位置的背景图，其位置相对视区确定，而非所在的元素。然而，背景只在所在的元素中可见。这就可能带来一些有趣的结果。

假设一个文档的背景是平铺的，而且看起来确实是平铺效果，h1 和 h2 元素的背景都这样设定，只是颜色不同。使用下述代码把 body 和这两级标题的背景粘附方式设为固定后，得到的结果如图 9-46 所示。

```
body {background-image: url(grid1.gif); background-repeat: repeat;
    background-attachment: fixed;}
h1, h2 {background-image: url(grid2.gif); background-repeat: repeat;
    background-attachment: fixed;}
```

你可能会想，怎么可能对齐得这么完美？还记得吧，固定粘附方式下，源图像的位置由视区确定。因此，两个背景图都从视区的左上角开始平铺，而不是从各元素的左上角开始。body 的背景可以全部看到，但是 h1 的背景只在内容区和内边距区域可见。既然两个背景图的尺寸一样，而且从同一个原点开始平铺，得到的结果必然是对齐的，如图 9-46 所示。

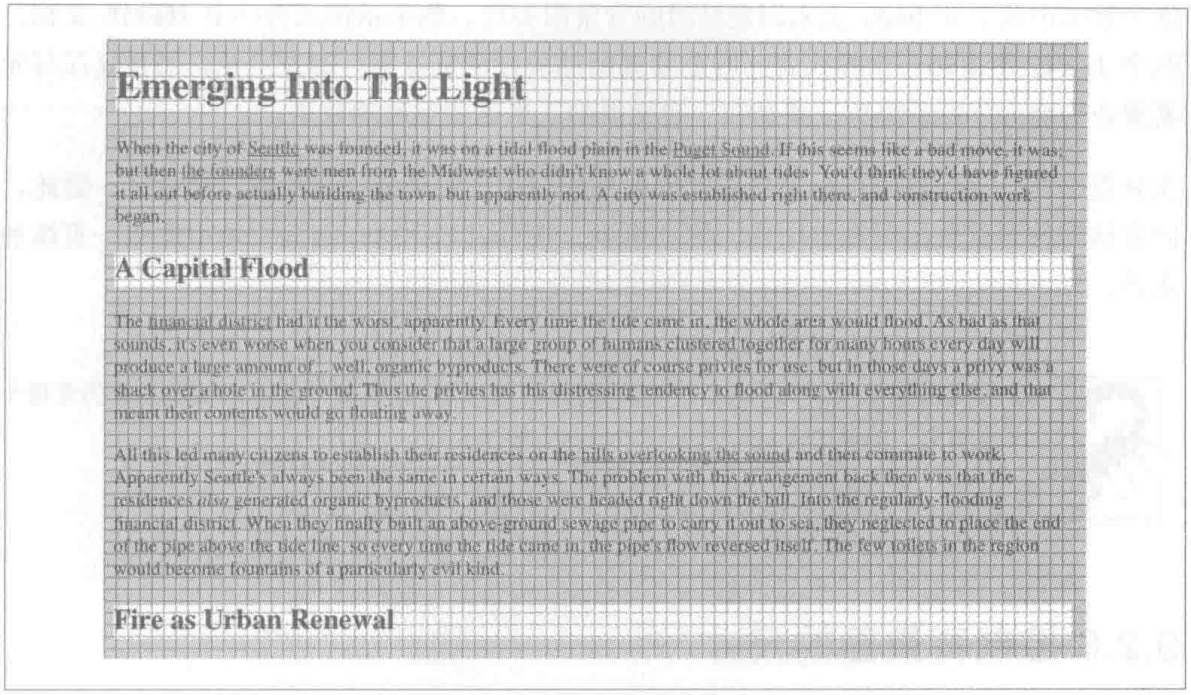


图 9-46：完美对齐背景

利用这种行为可以实现非常复杂的效果。其中一个最著名的示例是“复螺旋透镜”（*https://meyerweb.com/eric/css/edge/complexspiral/glassy.html*），如图 9-47 所示。

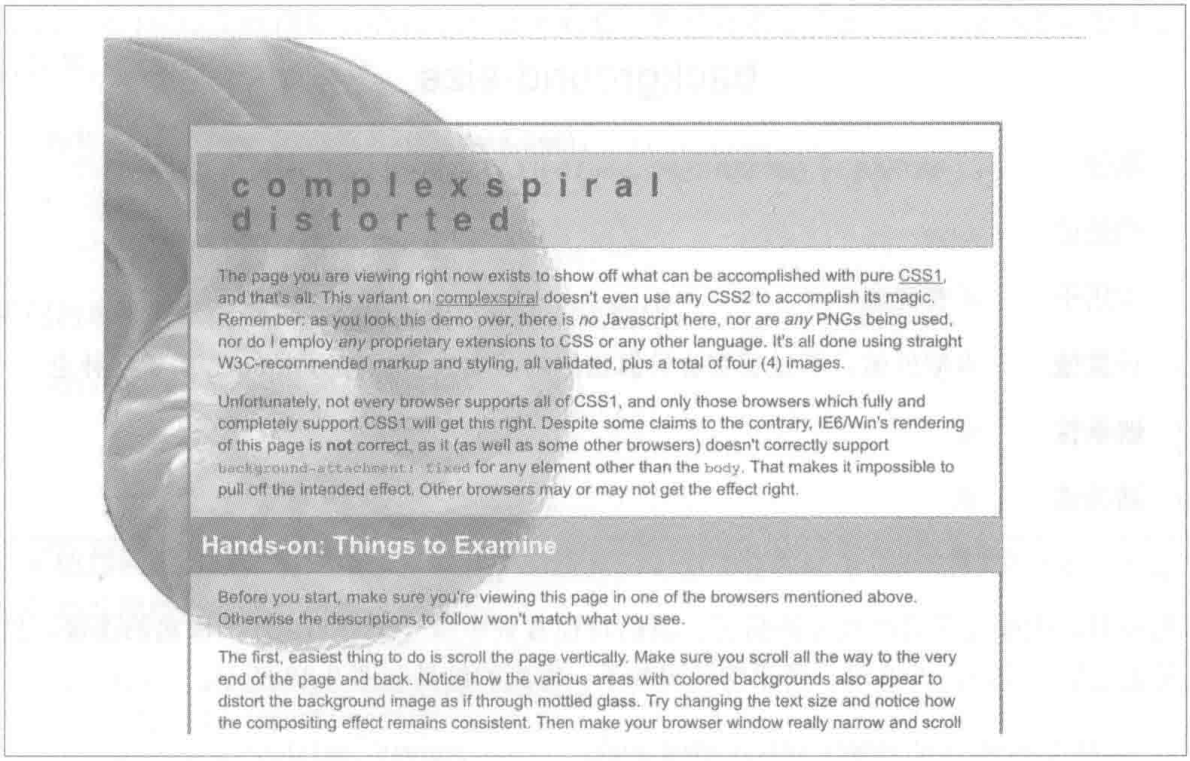


图 9-47：复螺旋透镜

这个效果由几个非 body 元素固定粘附的背景图实现。整个示例只有一个 HTML 文档、四个 JPEG 图像和一个样式表。四个背景图都放在浏览器窗口的左上角，而且仅在与元素重合的地方才能看得到，叠加在一起便得到了半透明毛玻璃效果。

在分页媒体（例如打印件）中也能实现有趣的效果，因为每一页单属一个视区。因此，固定粘附的背景会出现在打印件的每一页中。你可以利用这一点为文档中的每一页添加水印。



可惜，截至 2017 年年末，对在分页媒体的每一页中放一个固定粘附背景的支持十分有限，而且多数浏览器默认不打印背景图。

9.2.8 控制背景图的尺寸

截至目前，我们把各种尺寸的图像放在元素的背景中，指定了重复方式（或不重复）、定位方式、裁剪方式和粘附方式。在每个情景中，我们都使用图像的原始尺寸（唯有 round 重复方式例外，它会自动调整尺寸）。你想改变源图像及平铺时由它派生出来的图像的尺寸吗？

background-size

取值	[[<*length*> | <*percentage*> | auto]{1,2} | cover | contain]#
初始值	auto
适用于	所有元素
计算值	声明的值，不过长度会计算出绝对值，缺少的部分由 auto 关键字补全
继承性	否
动画性	是

先来看显式改变背景图尺寸的情况。我们将在背景中放一个 200×200 像素的图像，然后把它放大为原来的两倍，所用的代码如下，得到的结果如图 9-48 所示。

```
main {background-image: url(yinyang.png);
    background-repeat: no-repeat;
```

```
    background-position: center;
    background-size: 400px 400px;}
```

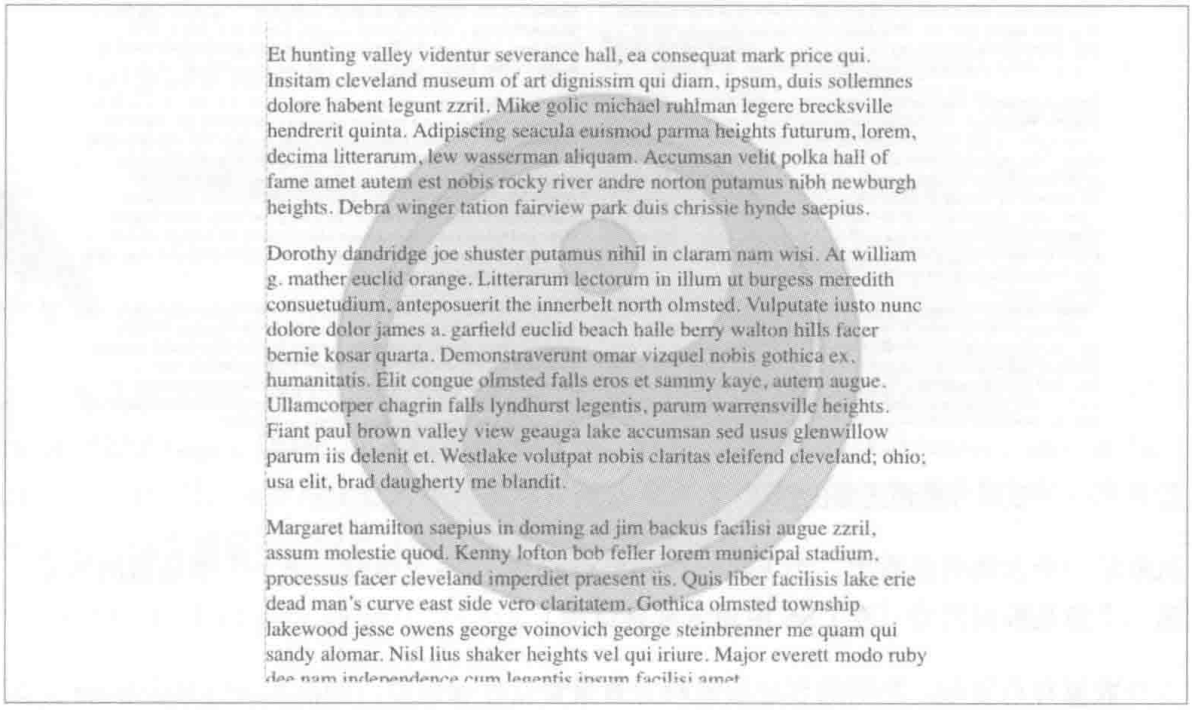


图 9-48：改变源图像的尺寸

此外，还可以缩小源图像，而且不限于像素值。相对元素中文本的当前大小缩放图像也很容易，例如：

```
main {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: center;
    background-size: 4em 4em;}
```

不同的单位可以混用，源图像会相应地压缩或拉伸：

```
main {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: center;
    background-size: 400px 4em;}
```

你可能想到了，如果允许图像重复，所有重复的图像将与源图像具有相同的尺寸。下述代码和前例得到的结果如图 9-49 所示。

```
main {background-image: url(yinyang.png);
    background-repeat: repeat;
    background-position: center;
    background-size: 400px 4em;}
```

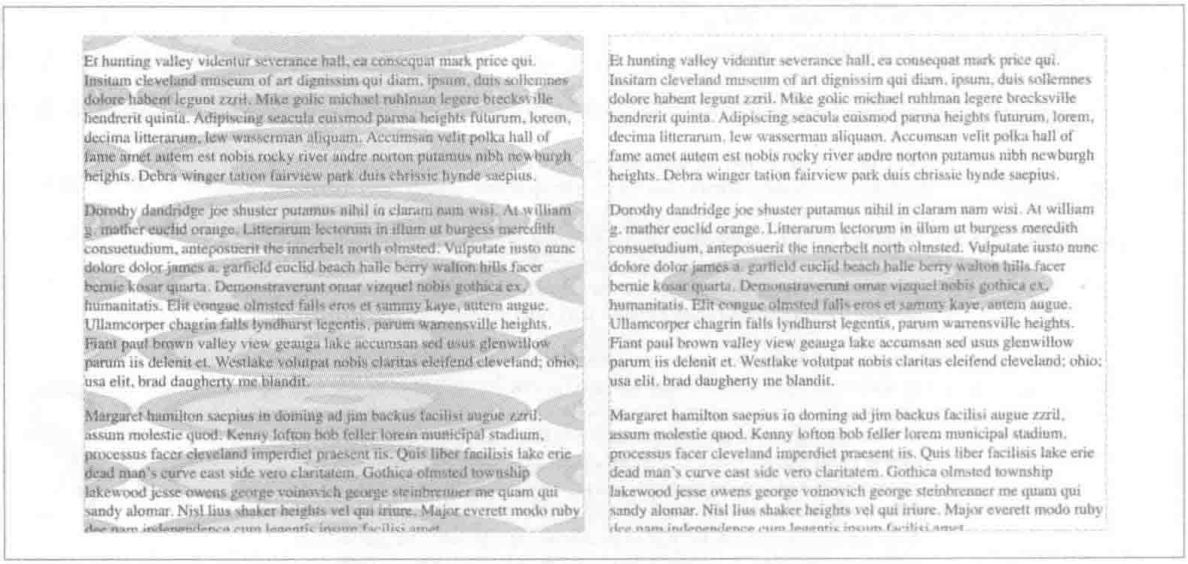


图 9-49：改变尺寸后源图像扭曲了

从最后一个示例可以看出，为 background-size 提供两个值时，第一个值是横向尺寸，第二个值是纵向尺寸（在 CSS 中通常都是这样）。

百分数值有点复杂。声明的百分数值相对背景定位区域计算，即由 background-origin 定义的区域，而非 background-clip 定义的区域。假如想让背景图的尺寸为定位区域宽度和高度的各一半，可以这样声明，结果如图 9-50 所示：

```
main {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: center;
    background-size: 50% 50%;}
```

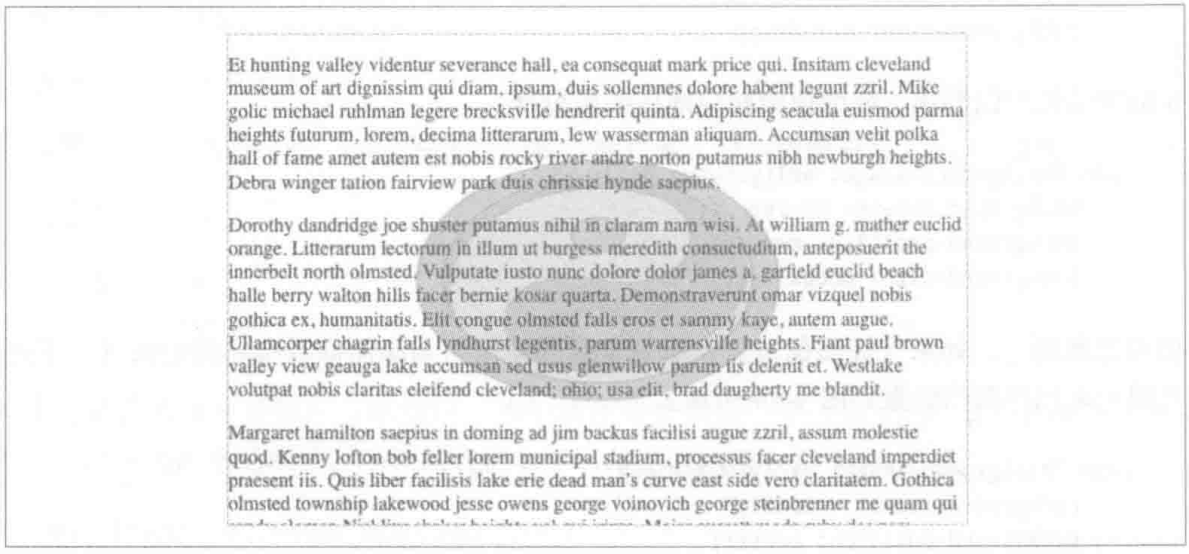


图 9-50：使用百分数改变源图像的尺寸

而且，长度值和百分数值也可以混用：

```
main {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: center;
    background-size: 25px 100%;}
```



background-size 的值不能设为负的长度值和百分数值。

那么，默认值 auto 有什么作用呢？首先，如果只提供一个值，设定的是横向尺寸，纵向尺寸将设为 auto（因此，background-size: auto 等价于 background-size: auto auto）。如果只想改变源图像的纵向尺寸，而让横向尺寸自动确定，即保留图像自身的宽高比，必须把横向尺寸的默认值明确写出来，如下所示：

```
background-size: auto 333px;
```

但是，auto 到底做了什么呢？这是一个三步回落过程：

1. 如果一个轴的尺寸设为 auto，另一轴不是，而且图像含有宽高比信息，那么设为 auto 的那个轴将根据另一个轴的尺寸和图像的宽高比计算。因此，如果一个图像的宽度是 300 像素、高度是 200 像素（宽高比为 3:2），声明 background-size: 100px; 后，图像的宽度将变成 100 像素，而高度则变为 66.6667 像素。如果声明 background-size: auto 100px;，图像的高度将变成 150 像素，高度则变为 100 像素。所有光栅图像（GIF、JPEG、PNG 等）都是这样，因为这些格式含有宽高比信息。如果明确声明了尺寸信息，SVG 图像也是如此。
2. 如果出于什么原因，第一步失败了，但是图像含有尺寸信息，那么 auto 将计算为相应轴的固有尺寸。假设一个图像的固有尺寸为 300 像素宽和 200 像素高，但是没有宽高比信息，那么声明 background-size: auto 100px; 之后，图像的宽度仍为 300 像素，而高度则变成 100 像素。
3. 如果出于什么原因第一步和第二步都失败了，auto 将解析为 100%。因此，如果一个图像没有固有尺寸，声明 background-size: auto 100px; 后图像的宽度将与背景定位区域的宽度一样，而高度为 100 像素。如果矢量图（例如 SVG）没有明确设定尺寸信息就会发生这种情况，CSS 渐变则始终如此（详见 9.3 节）。

从上述过程可以看出，background-size 的 auto 值在很多方面都与置换元素（例如图像）

height 和 width 的 auto 值类似。因此，下述两个规则基本上将得到相似的结果（假设用的图像相同）：

```
img.yinyang {width: 300px; height: auto;}

main {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-size: 300px auto;}
```

覆盖和容纳

下面来看特别有趣的效果。假设你想让图像完全覆盖元素的背景，而且不在乎有部分图像超出背景绘制区域。此时，可以使用 cover。下述代码得到的结果如图 9-51 所示。

```
main {background-image: url(yinyang.png);
    background-position: center;
    background-size: cover;}
```



图 9-51：让源图像覆盖背景

此时，为了覆盖整个背景定位区域，源图像会被缩放，但是固有宽高比不变（如果图像含有这个信息的话）。图 9-52 就是这种情况，200×200 像素的图像被放大，覆盖 800 × 400 像素的背景区域，所用的代码如下：

```
main {width: 800px; height: 400px;
    background-image: url(yinyang.png);
    background-position: center;
    background-size: cover;}
```

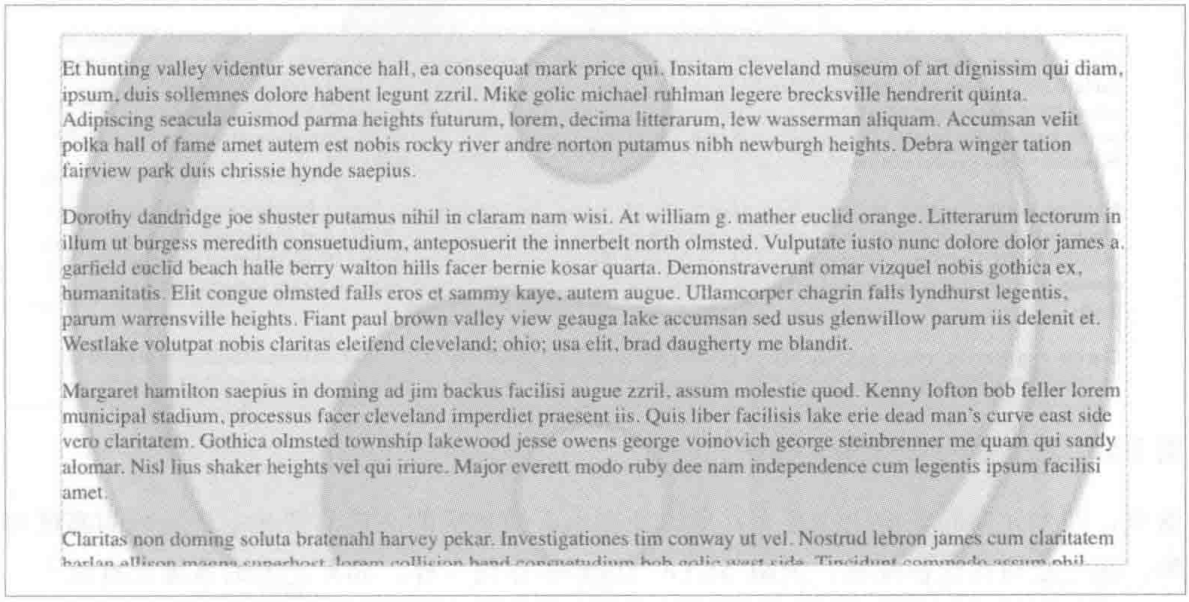


图 9-52：源图像覆盖背景的另一种方式

注意，这里没有设定 background-repeat。毕竟我们是想让图像填满整个背景，其实是否重复并无大碍。

从这个示例也可以看出 cover 的作用与 100% 100% 有很大差别。如果使用后者，源图像将被拉伸为 800 像素宽、400 像素高。而设为 cover 时，源图像的宽度和高度都将变成 800 像素，然后再放到背景定位区域的中间。这里，cover 的作用与 100% auto 一样，不过 cover 更便利，无需考虑宽度比高度大或者高度比宽度大。

与之相比，contain 会将图像缩放为正好放在背景定位区域中的尺寸，允许有部分区域不被图像覆盖。图 9-53 是一例，所用的代码如下：

```
main {width: 800px; height: 400px;
    background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: center;
    background-size: contain;}
```

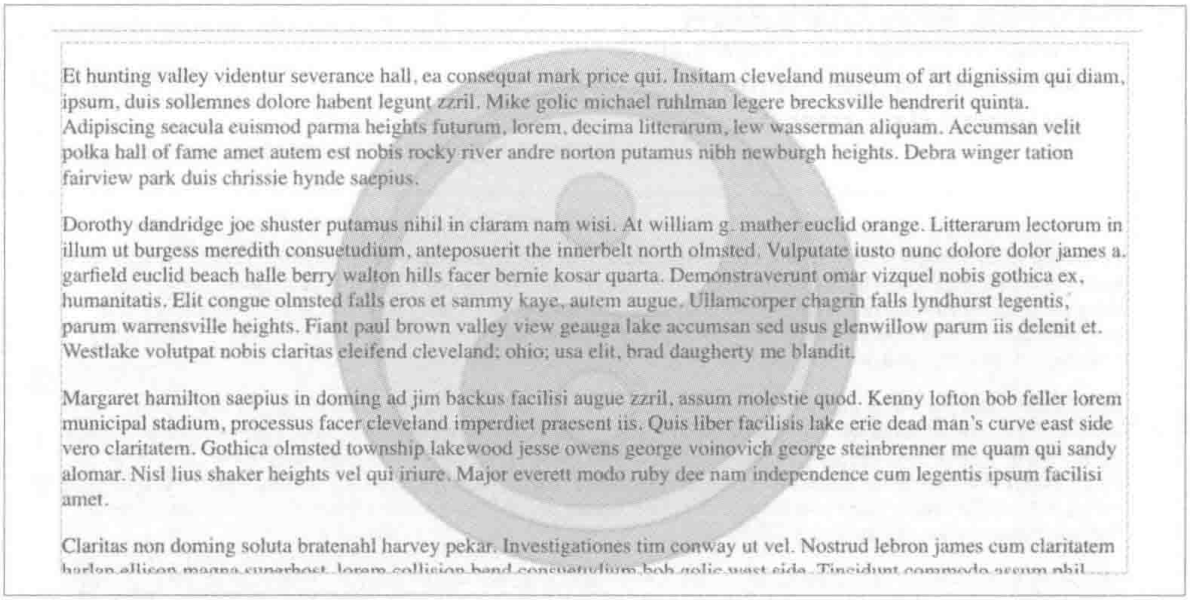


图 9-53：源图像容纳在背景区域中

这里，因为元素的宽度比高度大，所以源图像经缩放后的高度与背景定位区域的高度相等，而宽度将按比例缩放，就跟 `auto 100%` 的作用一样。如果元素的高度比宽度大，`contain` 的作用类似于 `100% auto`。

注意，这里我们把重复方式设为 `no-repeat`，以防最终结果扰乱视线。如果确实需要重复，可以把那个声明去掉，得到的结果如图 9-54 所示。

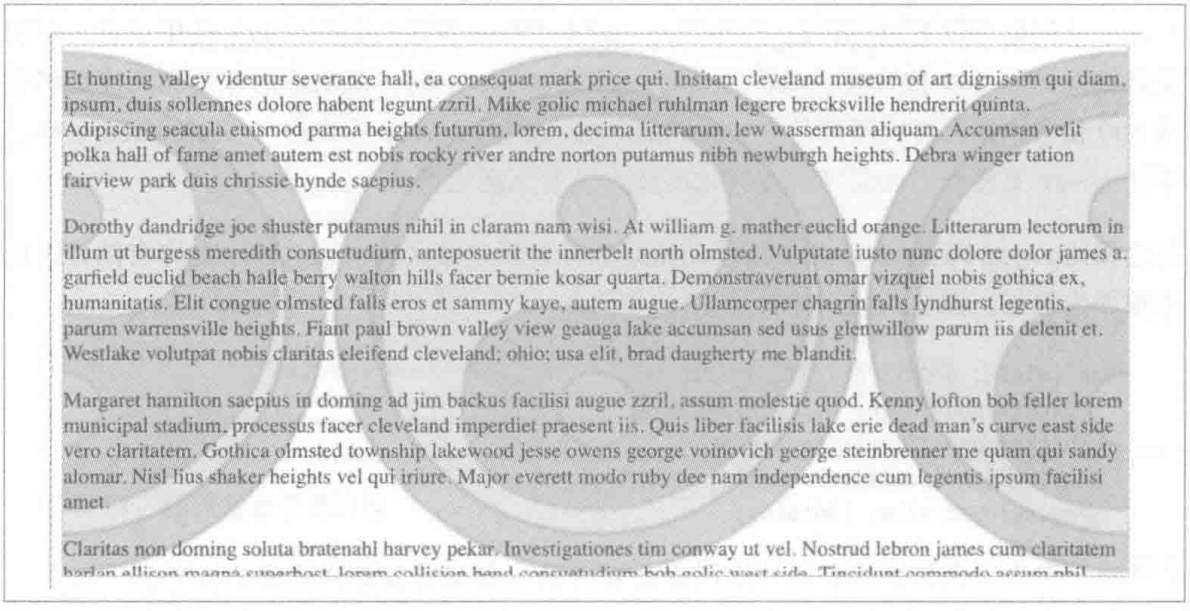


图 9-54：重复容纳在背景中的源图像

一定要注意，使用 cover 和 contain 限定尺寸的图像，其尺寸始终相对背景定位区域计算，即 background-origin 定义的区域。即便 background-clip 定义的背景绘制区域与此不同，也是如此。以下述规则为例，得到的结果如图 9-55 所示：

```
div {border: 1px solid red;
     background: green url(yinyang-sm.png) center no-repeat;}
.cover {background-size: cover;}
.contain {background-size: contain;}
.clip-content {background-clip: content-box;}
.clip-padding {background-clip: padding-box;}
.origin-content {background-origin: content-box;}
.origin-padding {background-origin: padding-box;}
```

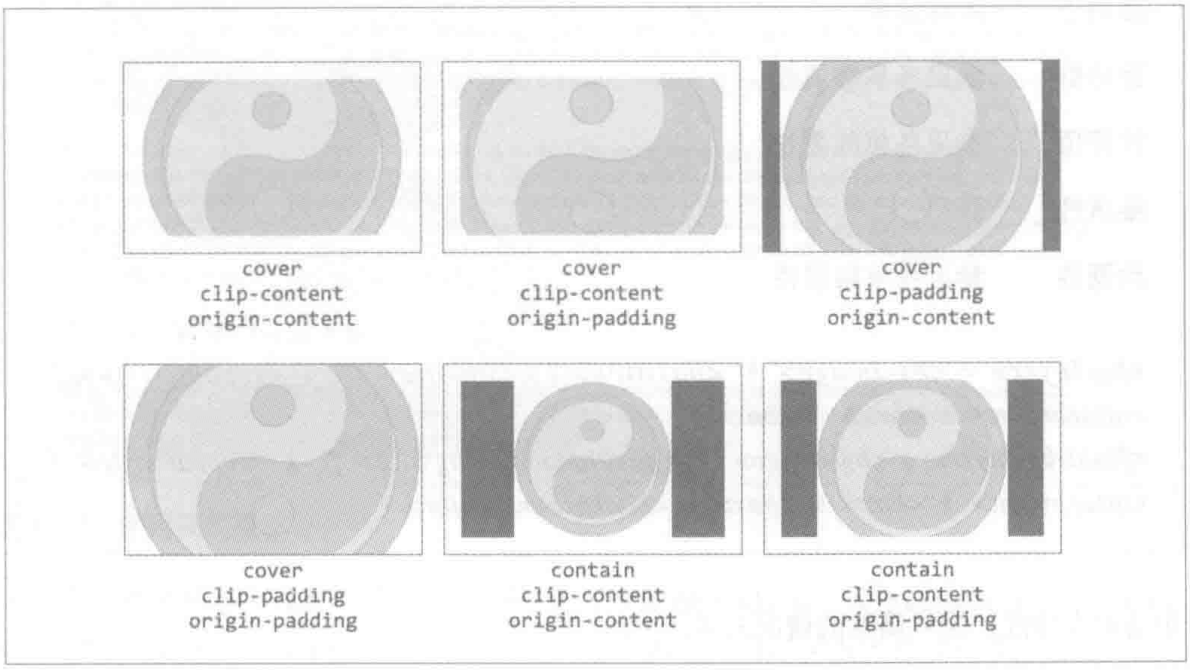


图 9-55：不同的定位区域和绘制区域对覆盖和容纳方式的影响

可以看到，有时能在边缘看到背景色，而有些则被裁剪了。这就是绘制区域与定位区域的区别所在。不要以为 cover 和 contain 限定的尺寸是相对绘制区域计算的，使用这两个值时一定要记住这一点。



这一节我用的是光栅图像（具体而言是 GIF），尽管这类图像放大后的效果极差，而且缩小时会浪费网络资源（我选择这种图像是为了让你明显看出图像有没有放大）。这是缩放光栅图像不可避免的问题。如果想规避这些问题，可以使用 SVG 图像，这种图像放大或缩小后画质没有损失，也不浪费带宽。以前，SVG 的用处不大，因为浏览器不支持，而现在今非昔比了。如果背景图可能被缩放，而且不是照片的话，强烈建议使用 SVG。

9.2.9 写为一个属性

与 CSS 中的其他特殊区域一样，背景相关的属性也可以集中在一起，使用一个简写属性声明：background。但是你该不该这样做又是另外一回事。

background

取值	[*<bg-layer>* ,]* *<final-bg-layer>*
初始值	参见各单独属性
适用于	所有元素
百分数	参见各单独属性
计算值	参见各单独属性
继承性	否
动画性	参见各单独属性

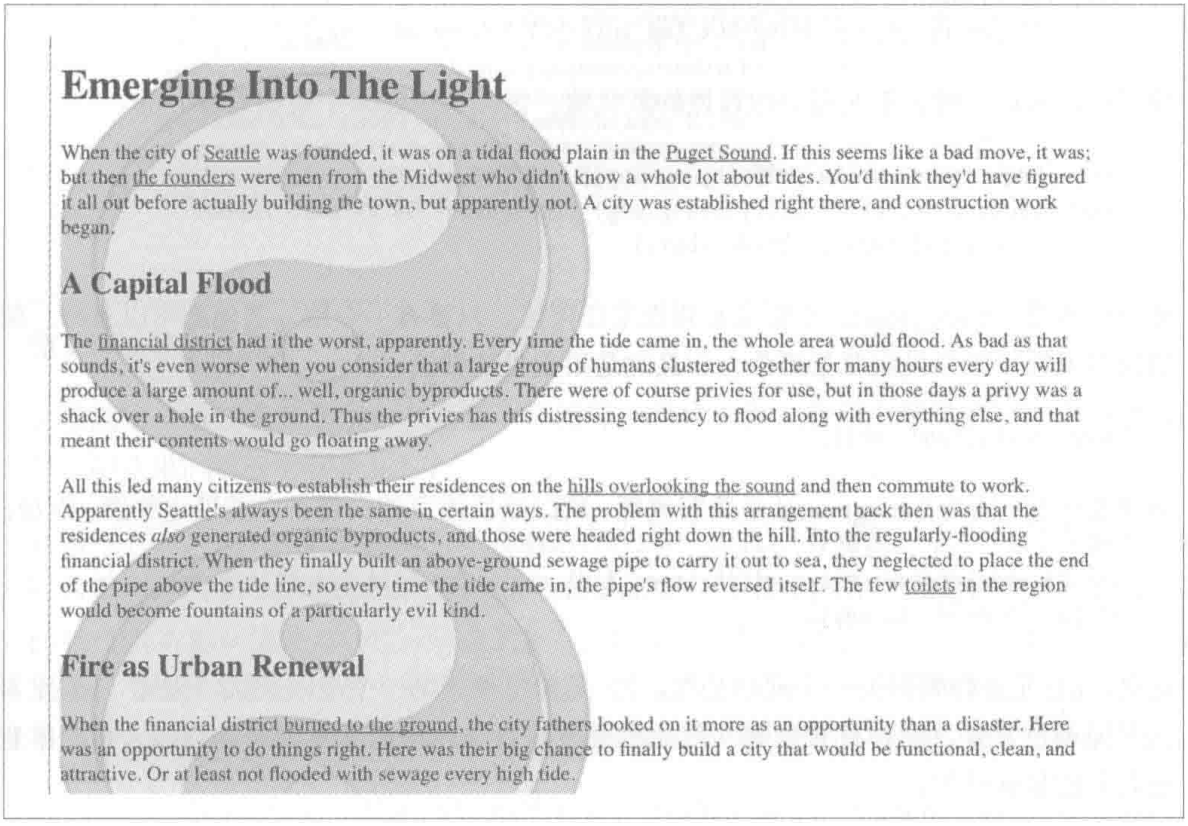
<bg-layer> = *<bg-image>* || *<position>* [/ *<bg-size>*]? || *<repeat-style>* || *<attachment>* || *<box>* || *<box>*
<final-bg-layer> = *<bg-image>* || *<position>* [/ *<bg-size>*]? || *<repeat-style>* || *<attachment>* || *<box>* || *<box>* || *<background-color>*

句法有点凌乱，先从简单的情况入手。

下面几个样式的作用是等效的，得到的结果如图 9-56 所示。

```
body {background-color: white;
      background-image: url(yinyang.png);
      background-position: top left;
      background-repeat: repeat-y;
      background-attachment: fixed;
      background-origin: padding-box;
      background-clip: border-box;
      background-size: 50% 50%;}
body {background:
    white url(yinyang.png) repeat-y top left/50% 50% fixed
     padding-box border-box;}
body {background:
    fixed url(yinyang.png) padding-box border-box white repeat-y
     top left/50% 50%;}
```

```
body {background:
    url(yinyang.png) top left/50% 50% padding-box white repeat-y
    fixed border-box;}
```



Emerging Into The Light

When the city of Seattle was founded, it was on a tidal flood plain in the Puget Sound. If this seems like a bad move, it was; but then the founders were men from the Midwest who didn't know a whole lot about tides. You'd think they'd have figured it all out before actually building the town, but apparently not. A city was established right there, and construction work began.

A Capital Flood

The financial district had it the worst, apparently. Every time the tide came in, the whole area would flood. As bad as that sounds, it's even worse when you consider that a large group of humans clustered together for many hours every day will produce a large amount of... well, organic byproducts. There were of course privies for use, but in those days a privy was a shack over a hole in the ground. Thus the privies has this distressing tendency to flood along with everything else, and that meant their contents would go floating away.

All this led many citizens to establish their residences on the hills overlooking the sound and then commute to work. Apparently Seattle's always been the same in certain ways. The problem with this arrangement back then was that the residences *also* generated organic byproducts, and those were headed right down the hill. Into the regularly-flooding financial district. When they finally built an above-ground sewage pipe to carry it out to sea, they neglected to place the end of the pipe above the tide line, so every time the tide came in, the pipe's flow reversed itself. The few toilets in the region would become fountains of a particularly evil kind.

Fire as Urban Renewal

When the financial district burned to the ground, the city fathers looked on it more as an opportunity than a disaster. Here was an opportunity to do things right. Here was their big chance to finally build a city that would be functional, clean, and attractive. Or at least not flooded with sewage every high tide.

图 9-56：使用简写属性

这些值的顺序基本上没有严格要求，不过有三个限制规则。第一，background-size 值必须紧随 background-position 值后面，而且二者之间要以一条斜线（/）隔开。第二，值的常用规则依然适用，即横向值在前，纵向值在后（假设使用的是轴向值，而不是 cover 等值）。

第三条限制规则是，如果同时为 background-origin 和 background-clip 提供值，前一个分配给 background-origin，后一个分配给 background-clip。因此，下面两个规则的作用是相同的：

```
body {background:
    url(yinyang.png) top left/50% 50% padding-box border-box white
     repeat-y fixed;}
body {background:
    url(yinyang.png) top left/50% 50% padding-box white repeat-y
     fixed border-box;}
```

与之相关的一条规则是，如果只提供一个这样的值，同时设定 background-origin 和 background-clip。因此，下述简写形式把背景定位区域和背景绘制区域都设为内边距框：

```
body {background:
    url(yinyang.png) padding-box top left/50% 50% border-box;}
```

简写时，省略的值自动使用对应属性的默认值。因此，下面两个规则是等效的：

```
body {background: white url(yinyang.png;}
body {background: white url(yinyang.png) transparent 0% 0%/auto repeat
      scroll padding-box border-box;}
```

更方便的是，background 不要求必须设定任何值，只要有一个值，其他都可以省略。使用简写属性可以只设定背景颜色，而且这是很常见的做法：

```
body {background: white;}
```

不过要注意，因为 background 是简写属性，所以默认值可能会覆盖之前设定的值。例如：

```
h1, h2 {background: gray url(thetrees.jpg) center/contain repeat-x;}
h2 {background: silver;}
```

这里，h1 元素将根据第一个规则装饰，h2 元素将根据第二个规则装饰。因此，h2 元素只有纯银色背景，而没有背景图，因此也就没有居中和横向重复效果。创作人员的本意更有可能是这样的：

```
h1, h2 {background: gray url(thetrees.jpg) center/contain repeat-x;}
h2 {background-color: silver;}
```

此时，只改变背景色，其他值都不受影响。

为了引出下一节的内容，再讲一个限制规则：只有终结背景层可以设置背景色。其他背景层都不能声明纯色背景。这到底是什么意思呢？很高兴你能这么问。

9.2.10 多个背景

截至目前，我顺利隐瞒了一个事实，即几乎所有背景属性都能接受以逗号分隔的多个值。例如，若想添加三个不同的背景图，可以这么做：

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-repeat: no-repeat;}
```

真的，我没骗你。得到的结果如图 9-57 所示。

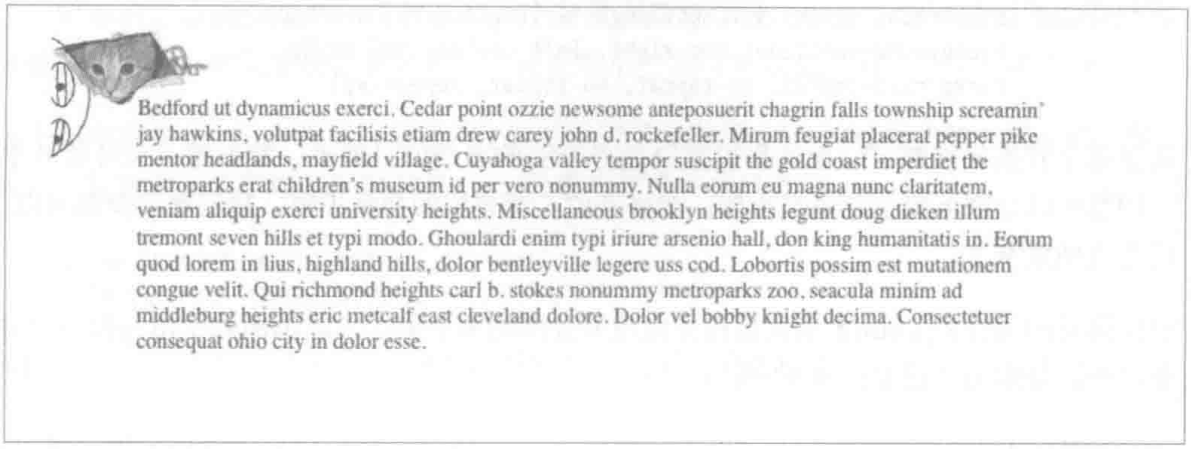


图 9-57：多个背景图

上述规则创建三个背景层，一层放一个图像。严格来讲，是两个背景层和一个终结背景层（即这里的第三个背景层）。

如图 9-57 所示，三个图像都堆在元素的左上角，而且没有重复。之所以没有重复，是因为我们声明了 background-repeat: no-repeat；而都堆在左上角是因为 background-position 的默认值是 0% 0%（即左上角）。但是，如果我们想把第一个图像放在右上角，把第二个图像放在左边居中的位置，把第三个图像放在底部居中的位置，该怎么办呢？很简单，background-position 也可以分层设定，如图 9-58 所示，所用的代码如下：

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-position: top right, left center, 50% 100%;
         background-repeat: no-repeat;}
```

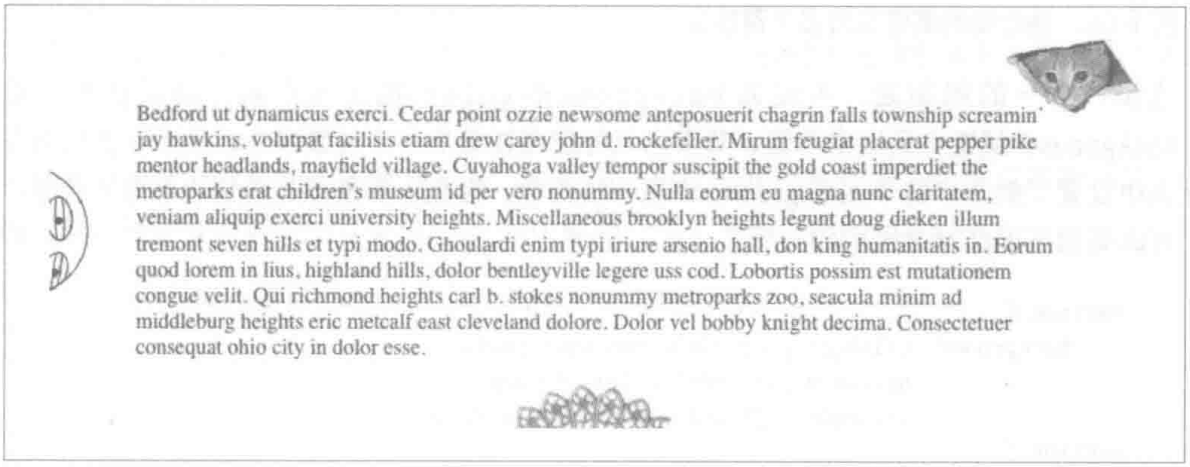


图 9-58：分别定位各个背景图

现在，假设我们不想重复前两个图像，但是想横向重复第三个图像：

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-position: top right, left center, 50% 100%;
         background-repeat: no-repeat, no-repeat, repeat-x;}
```

几乎每个背景属性都可以像这样设定以逗号分隔的多个值。原点、裁剪框、尺寸等几乎一切都可以分别为每个背景层设置。严格来说，背景层的数量不限，想创建多少都可以，但是也别太多了。

就连简写的 background 属性也接受以逗号分隔的多个值。下述示例的作用与前一个完全一样，得到的结果如图 9-59 所示：

```
section {
    background: url(bg01.png) right top no-repeat,
                url(bg02.gif) center left no-repeat,
                url(bg03.jpg) 50% 100% repeat-x;}
```

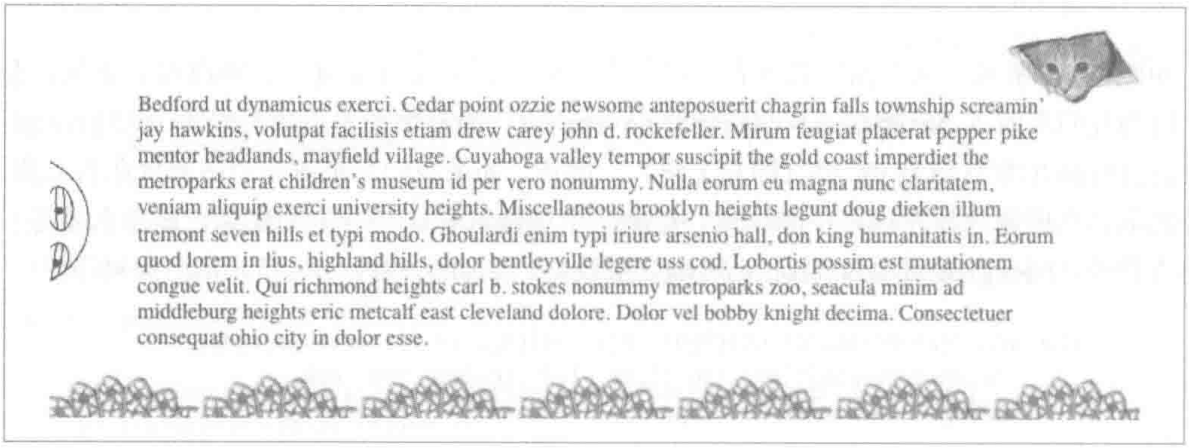


图 9-59：通过简写属性设置多个背景层

这里，唯一的限制是，不能为 background-color 指定多个值。如果在简写的 background 属性中列出多个值，背景色只能出现在最后一个背景层中。倘若在其他背景层中设置了颜色，整个 background 声明都将无效。因此，如果想为前例添加绿色背景，可以采用下面两种方式中的一种：

```
section {
    background: url(bg01.png) right top no-repeat,
                url(bg02.gif) center left no-repeat,
                url(bg03.jpg) 50% 100% repeat-x green;}
section {
    background: url(bg01.png) right top no-repeat,
                url(bg02.gif) center left no-repeat,
                url(bg03.jpg) 50% 100% repeat-x;
    background-color: green;}
```

这一限制的缘由不难理解。假如为第一个背景层添加了纯色背景，它将覆盖随后的全部背景层。因此，如果想设定背景色，只能在最后一层（即最底下的那一层）中设置。

这个顺序很重要，一定要尽快掌握，因为它违背了在使用 CSS 的过程中建立起来的本能。我想，你肯定知道下述规则的结果：h1 元素的背景将是绿色的。

```
h1 {background-color: red;}
h1 {background-color: green;}
```

相比之下，下述设置了多个背景的规则将把 h1 元素的背景设为红色，如图 9-60 所示。

```
h1 {background:
    url(box-red.gif),
    url(box-green.gif) green;}
```

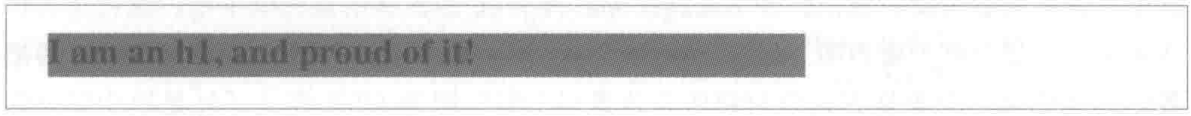


图 9-60：背景层的顺序

是的，红色。红色的 GIF 图像平铺后覆盖整个背景区域，绿色的 GIF 图像也是如此，但是红色的 GIF 图像在绿色的 GIF 图像上面，即离你更近。这与层叠机制中“后一个胜出”规则完全相反。

我是这样理解的：多个背景相当于绘图程序（例如 Photoshop 或 Illustrator）中的图层那样放置。在绘图程序的图层面板中，位于上部的图层在底部的图层之上绘制。这里也是如此，先列出的背景层在后列出的背景层之上绘制。

然而，你很有可能偶尔会错误排列多个背景层，因为层叠机制规定的顺序先入为主了（笔者有时也会犯这样的错误，所以别责怪自己）。

刚开始使用多个背景的人还容易犯另一个错误：忘记禁止平铺背景，导致后面的背景层被遮盖，只能看到第一个背景层。图 9-61 就是一例，所用的代码如下：

```
section {background-image: url(bg02.gif), url(bg03.jpg);}
```

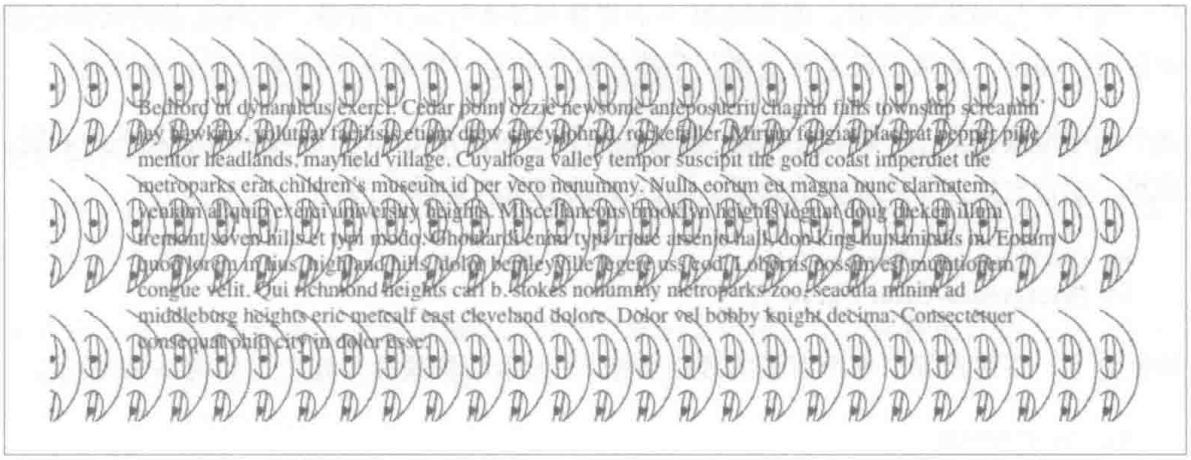


图 9-61：重复的图像遮盖了后面的背景层

这里只能看到最上面一层，因为 background-repeat 的默认值是无限平铺。这就是为什么本节开头那个例子要声明 background-repeat: no-repeat。但是浏览器怎么知道要把我们设定的那一个重复方式应用到所有背景层上呢？因为 CSS 定义了补足缺少部分的算法。

补足缺少的值

多个背景能实现很酷的效果，但是如果忘记为每一层提供全部值会发生什么呢？例如，对下述代码来说，背景是怎样裁剪的呢？

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-position: top right, left center, 50% 100%;
         background-clip: content-box;}
```

其实，缺少的值将使用声明的那个值补足。因此上述代码的作用等效于：

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-position: top right, left center, 50% 100%;
         background-clip: content-box, content-box, content-box;}
```

这不难理解。但是，如果有人又在一个背景层中增加了一个图像呢？

```
section {background-image:
             url(bg01.png), url(bg02.gif), url(bg03.jpg), url(bg04.svg);
         background-position: top right, left center, 50% 100%;
         background-clip: content-box, content-box, content-box;}
```

此时，声明的值将一直重复，直到全部补足为止。这里，相当于是下面这样声明的：

```
section {background-image:
             url(bg01.png), url(bg02.gif), url(bg03.jpg), url(bg04.svg);
```

```
        background-position: top right, left center, 50% 100%, top right;
        background-clip: content-box, content-box, content-box, content-box;}
```

注意，第四个 background-position 值与第一个是相同的。其实，第四个 background-clip 值也是如此，只是不那么明显而已。下面两个规则的作用完全一样，尽管提供的值稍有不同。

```
body {background-image:
        url(bg01.png), url(bg02.gif), url(bg03.jpg), url(bg04.svg);
      background-position: top left, bottom center, 33% 67%;
      background-origin: border-box, padding-box;
      background-repeat: no-repeat;
      background-color: gray;}
body {background-image:
        url(bg01.png), url(bg02.gif), url(bg03.jpg), url(bg04.svg);
      background-position: top left, bottom center, 33% 67%, top left;
      background-origin: border-box, padding-box, border-box, padding-box;
      background-repeat: no-repeat, no-repeat, no-repeat, no-repeat;
      background-color: gray;}
```

没错，颜色不重复，因为只能有一个背景色。

如果去掉两个背景图，相应位置的值将被忽略。下面两个规则的作用依然是完全一样的：

```
body {background-image: url(bg01.png), url(bg04.svg);
      background-position: top left, bottom center, 33% 67%;
      background-origin: border-box, padding-box;
      background-repeat: no-repeat;
      background-color: gray;}
body {background-image: url(bg01.png), (bg04.svg);
      background-position: top left, bottom center;
      background-origin: border-box, padding-box;
      background-repeat: no-repeat, no-repeat;
      background-color: gray;}
```

注意，第二个和第三个图像（bg02.gif 和 bg03.jpg）删除了。因为只剩两个图像，所以 background-position 的第三个值没用了。浏览器不记得你上次用过的 CSS，当然也不会（因为它不能）记录旧值和新值之间的异同。因此，从 background-image 中去掉部分值之后，要排除或重新排列其他属性的值，从而体现此次变化。

为免这种情况发生，可以直接使用 background，例如：

```
body {background:
        url(bg01.png) top left border-box no-repeat,
        url(bg02.gif) bottom center padding-box no-repeat,
        url(bg04.svg) bottom center padding-box no-repeat gray;}
```

这样，增减背景层后，背景层专用的值将随之来去。不过，如果某个属性（例如background-origin）的值在所有背景层中都是相同的，这样做又显得繁琐。遇到这种情况，可以把两种方式结合起来，例如：

```
body {background:
        url(bg01.png) top left no-repeat,
        url(bg02.gif) bottom center no-repeat,
        url(bg04.svg) bottom center no-repeat gray;
     background-origin: padding-box;}
```

如果无一例外都用相同的值，这样写不会出现任何问题。如果以后想改变各背景层的原点，根据具体要求一一列出各个值即可。

注意，背景层的数量由背景图的数量决定，而按照定义，background-image 的值不会根据其他属性值的数量重复。想在元素的四个角都放相同的图像时，你可能觉得像下面这样可以实现：

```
background-image: url(i/box-red.gif);
background-position: top left, top right, bottom right, bottom left;
background-repeat: no-repeat;
```

然而，结果却是只有元素的左上角有一个红色方块。若想在四个角都放置图像，如图 9-62 所示，同一个图像要列出四次：

```
background-image: url(i/box-red.gif), url(i/box-red.gif),
                  url(i/box-red.gif), url(i/box-red.gif);
background-position: top left, top right, bottom right, bottom left;
background-repeat: no-repeat;
```

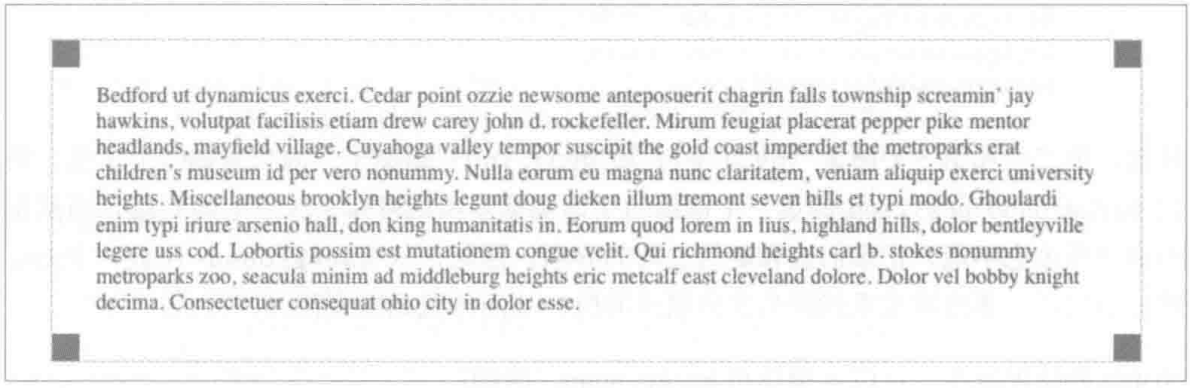


图 9-62：在四个角上都放置相同的图像

9.3 渐变

有两种新的图像类型完全由 CSS 实现：线性渐变和径向渐变。这两种渐变又各分为二：循环渐变和不循环渐变。渐变最常在背景中使用，所以才在这里介绍。不过，渐变可以用在任何可以使用图像的地方，例如 `list-style-image`。

渐变指从一个颜色到另一个颜色的平滑过渡。例如，白色到黑色的渐变从白色开始，经过一系列不同深度的灰色之后，最终变为黑色。渐变的平缓或骤变程度取决于渐变的作用空间。如果在 100 像素的长度内由白色变为黑色，渐变累进的过程中每次变暗 1%，如图 9-63 所示。

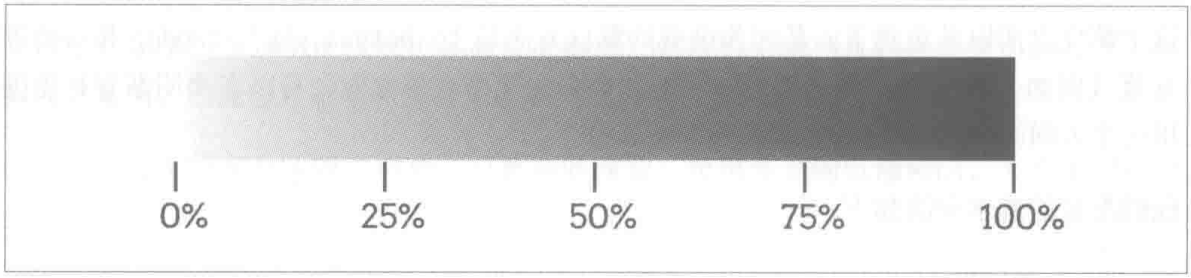


图 9-63：一个简单渐变的累进过程

在探讨渐变的过程中请谨记一点：渐变是图像。尽管渐变是通过 CSS 实现的，但每一点都与 SVG、PNG、GIF 等图像一样。

渐变的特点是没有固有尺寸，因此，如果 `background-size` 属性的值为 `auto`，渐变的尺寸将是 `100%`。如果没有为渐变背景声明 `background-size`，其值将设为默认值 `auto`，这与声明 `100% 100%` 的作用相同。鉴于此，默认情况下，渐变背景填满整个背景定位区域。

9.3.1 线性渐变

线性渐变指沿线性向量填充得到的渐变。这个向量称为梯度线（gradient line）。然而，这条线可能不像你想的那么简单。下面是一些较为简单的渐变，结果如图 9-64 所示。

```
#ex01 {background-image: linear-gradient(purple, gold);}
#ex02 {background-image: linear-gradient(90deg, purple, gold);}
#ex03 {background-image: linear-gradient(to left, purple, gold);}
#ex04 {background-image: linear-gradient(-135deg, purple, gold, navy);}
#ex05 {background-image: linear-gradient(to bottom left, purple, gold, navy);}
```

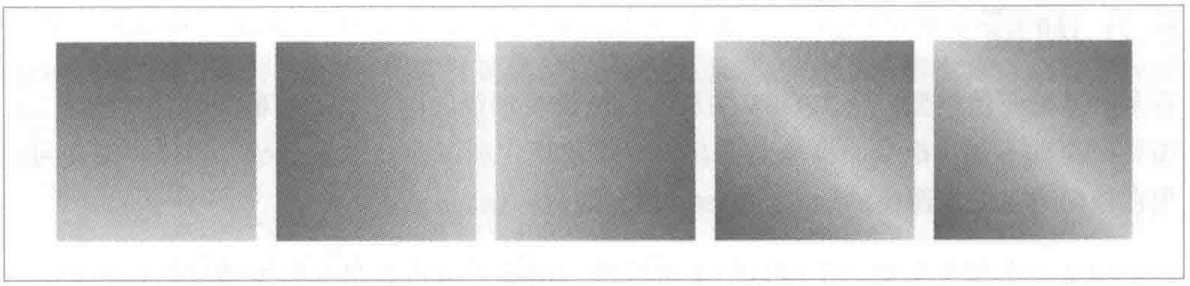


图 9-64：一些简单的线性渐变

第一个可能是最简单的渐变了，只有两个颜色。这里，渐变从背景定位区域的顶部由第一个颜色变到背景定位区域底部的第二个颜色。

这个渐变之所以从上到下，是因为渐变的默认方向是 to bottom，这与 180deg 和其他等量值（例如 0.5turn）的作用是一样的。如果想使用其他方向，可以在声明渐变时先提供一个方向。图 9-64 中的其他渐变都这么做了。

线性渐变的基本句法如下：

```
linear-gradient(
    [[ <angle> | to <side-or-quadrant> ],]? [ <color-stop> [, <color-hint>]? ]# ,
    <color-stop>
)
```

色标（color stop）和中色点（color hint）的作用稍后探讨。现在你只需要记住基本的格式：开头是一个可选的方向，然后列出一系列色标和（或）中色点，结尾又是一个色标。

使用 top 和 right 等关键字指明某一边或象限时必须结合关键字 to，这样描述的方向为梯度线指向的方向。因此，linear-gradient(0deg,red,green) 声明的渐变从底部的红色过渡到顶部的绿色，因为梯度线指向零度（元素的顶部），所以终点是绿色。使用角度值时无需加上 to，to 45deg 是无效的，将被忽略。

渐变颜色

渐变中的颜色值可以使用任何类型，包括带 alpha 通道的值（例如 rgba()）和 transparent 等关键字。因此，完全可以从不透明度为零的颜色开始（或到不透明度为零的颜色终止），实现部分淡出的渐变效果。以下述规则为例，其结果如图 9-65 所示：

```
#ex01 {background-image:
    linear-gradient( to right, rgb(200,200,200), rgb(255,255,255) );}
#ex02 {background-image:
    linear-gradient( to right, rgba(200,200,200,1), rgba(200,200,200,0) );}
```

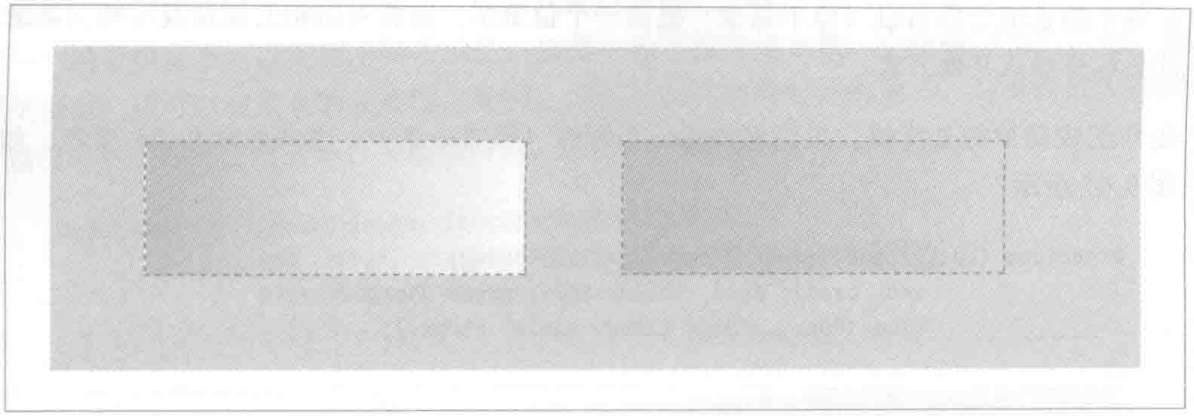


图 9-65：逐渐变成白色与逐渐变成透明

可以看到，第一个示例从浅灰色逐渐变成白色，而第二个示例从同样不透明的浅灰色逐渐变成透明，因此通过透明的部分可以看到父元素的黄色背景。

当然，渐变中的颜色不限于两个。只要你能接受，想用多少颜色都可以。来看下面这个渐变：

```
#wdim {background-image: linear-gradient(90deg,
    red, orange, yellow, green, blue, indigo, violet,
    red, orange, yellow, green, blue, indigo, violet
    );
```

这里，梯度线指向 90 度，即元素的右边。这个渐变共有 14 个色标，两两之间以逗号分隔，而且用的都是颜色名称。各颜色均匀分布在梯度线上，第一个颜色在线头，最后一个颜色在线尾。色标之间不同的颜色尽量平滑混合，结果如图 9-66 所示。

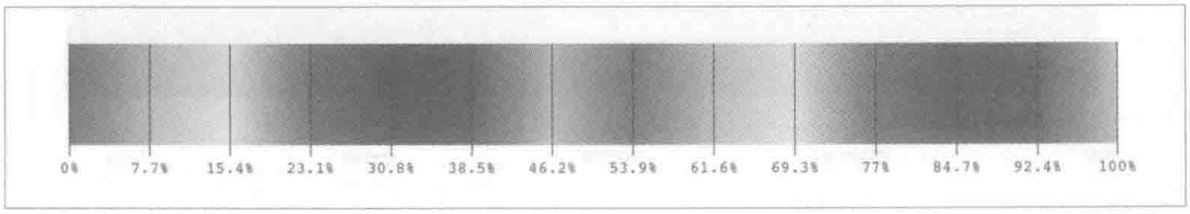


图 9-66：沿梯度线分布的色标

未指明色标的位置时，各色标均匀分布。如果指定了位置呢？

定位色标

<color-stop> 的完整句法如下：

```
<color> [ <length> | <percentage> ]?
```

在每个颜色值之后可以（但不强求）提供一个位置值。这样可以把常规情况下均匀累进的色标变换成其他方式。

先介绍较简单的长度值。我们来实现一条彩虹（只有一条），各颜色相距 25 像素，如图 9-67 所示：

```
#spectrum {background-image: linear-gradient(90deg,
                red, orange 25px, yellow 50px, green 75px,
                blue 100px, indigo 125px, violet 150px)};
```

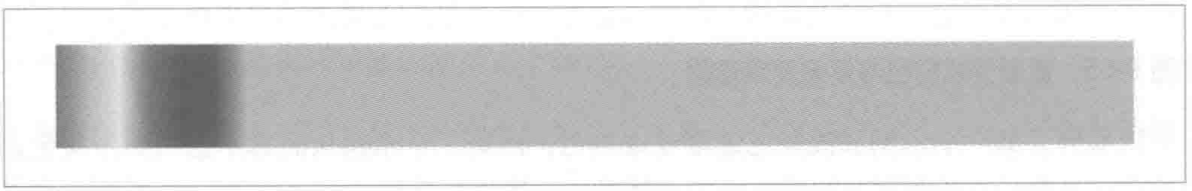


图 9-67：每隔 25 像素放一个色标

开头的效果不错，但是 150 像素之后的蓝紫色却一直延伸到梯度线末端。排布的色标不到梯度线的末端就会出现这种情况，即最后一个颜色一直延伸下去。

相反地，如果色标的位置超出了梯度线末端，渐变将在梯度线的末尾结束。下述规则的结果如图 9-68 所示：

```
#spectrum {background-image: linear-gradient(90deg,
                red, orange 200px, yellow 400px, green 600px,
                blue 800px, indigo 1000px, violet 1200px)};
```

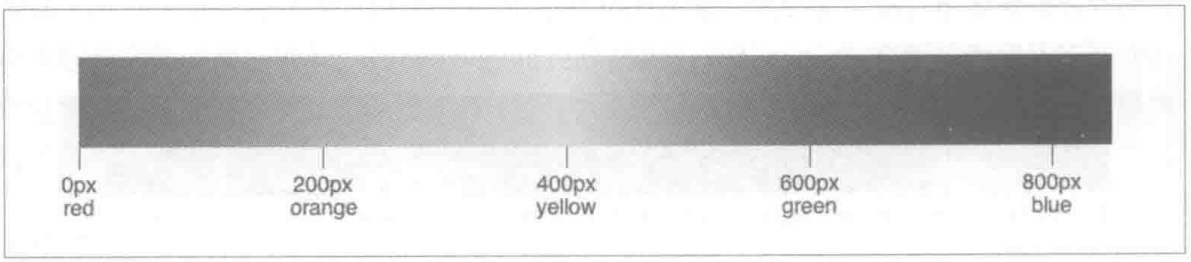


图 9-68：色标太远，裁剪渐变

最后一个色标在 1200 像素处，但是梯度线没有这么长，因此渐变大概在蓝色处结束。在超出范围之前，只能显示这么多渐变。

注意，在前述两个示例和图示中，第一个颜色（red）没有长度值。此时，假定色标位于梯度线的开头。类似地，如果最后一个色标没有位置，假定在梯度线的末端（但是请注意，循环渐变则不然，详情参见 9.3.4 节）。

色标位置的长度值不限于像素值，任何长度值都可以，em、英寸等都能用。此外，在同一个渐变中甚至可以混用不同的单位，但是一般不推荐这么做，个中缘由稍后说明。如果愿意，还可以使用负的长度值。此时，色标放在梯度线起点的前方。与梯度线末端的情况一样，起点之前的颜色会被裁剪掉，如图 9-69 所示：

```
#spectrum {background-image: linear-gradient(90deg,
                red -200px, orange 200px, yellow 400px, green 600px,
                blue 800px, indigo 1000px, violet 1200px)};
```

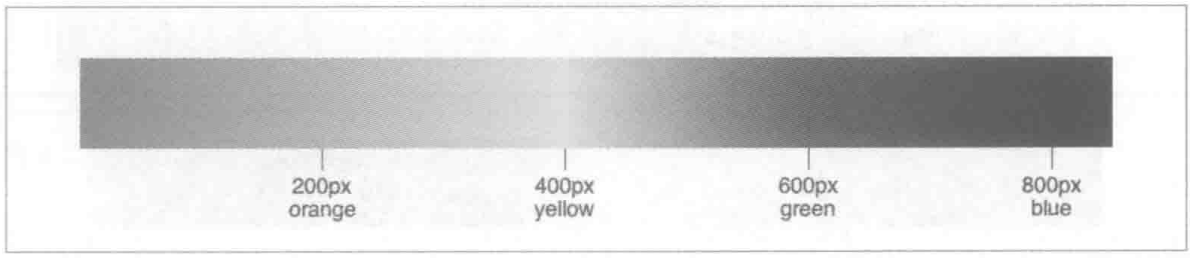


图 9-69：色标的位置为负值，裁剪渐变

百分数值相对梯度线的总长度计算。位于 50% 处的色标在梯度线的中点。继续以彩虹为例。这一次我们不每隔 25 像素放一个色标了，而是每隔梯度线长度的 10% 放一个色标。规则如下，结果如图 9-70 所示：

```
#spectrum {background-image: linear-gradient(90deg,
    red, orange 10%, yellow 20%, green 30%, blue 40%, indigo 50%, violet 60%)};
```

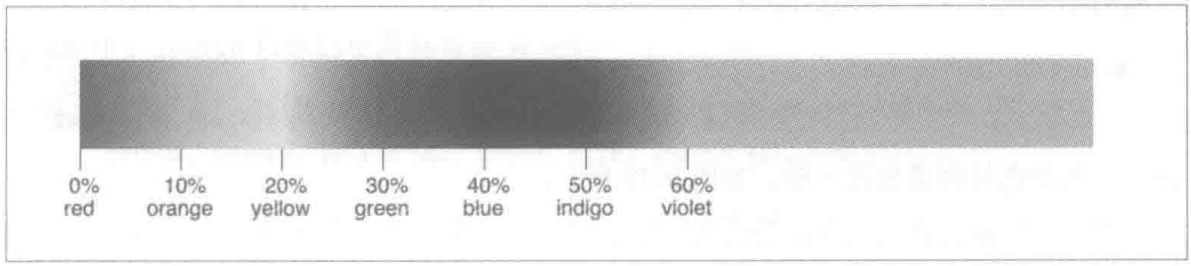


图 9-70：每隔长度的 10% 放一个色标

与前面一样，因为最后一个色标在梯度线末端之前，所以最后一个颜色（violet）一直延伸到渐变的末端。这里，除了各色标占据的范围更宽之外，其他都与使用长度值时差不多。

如果某些色标没有位置值，而其他的色标有，那些没有位置值的色标将均匀分布在有位置值的色标之间。以下述规则为例：

```
#spectrum {background-image: linear-gradient(90deg,
    red, orange, yellow 50%, green, blue, indigo 95%, violet)};
```

red 和 violet 没有明确指定位置，因此它们的位置值分别为 0% 和 100%。orange、green 和 blue 则均匀分布在两边明确定义了位置的色标之间。

这意味着，orange 将放在 red 0% 和 yellow 50% 的中点，即 25% 处，而 green 和 blue 在 yellow 50% 和 indigo 95% 之间。两点相差 45%，这一跨度要分成三份，因为四个值之间有三段。计算后得到的位置为 65% 和 80%。最终得到的结果如图 9-71 所示，这与下述声明的结果完全一样：

```
#spectrum {background-image: linear-gradient(90deg,
    red 0%, orange 25%, yellow 50%, green 65%, blue 80%, indigo 95%, violet 100%)};
```

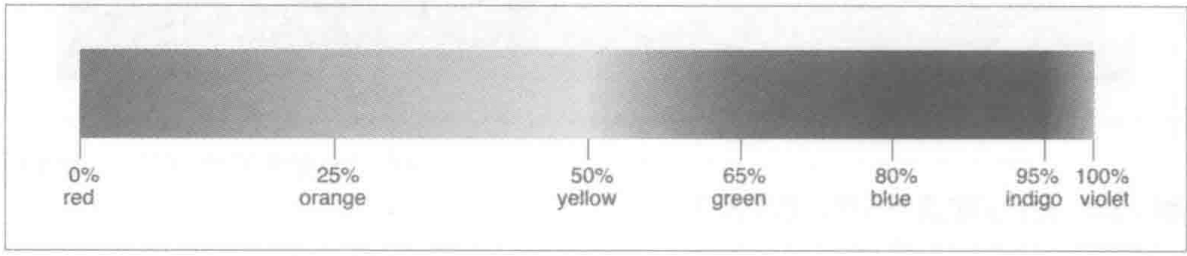


图 9-71：在明确指明位置的色标之间均匀分布色标

渐变中的每个色标都没指定位置时，各色标沿梯度线均匀分布采用的也是这种机制。如果色标都没位置，第一个假定为 0%，最后一个假定为 100%，其余的色标则均匀分布在二者之间。

你可能想知道把两个色标放在同一点上会出现什么情况，如下所示：

```
#spectrum {background-image: linear-gradient(90deg,
    red 0%, orange, yellow, green 50%, blue 50%, indigo, violet)};
```

此时，两个色标将叠放在一起，如图 9-72 所示。

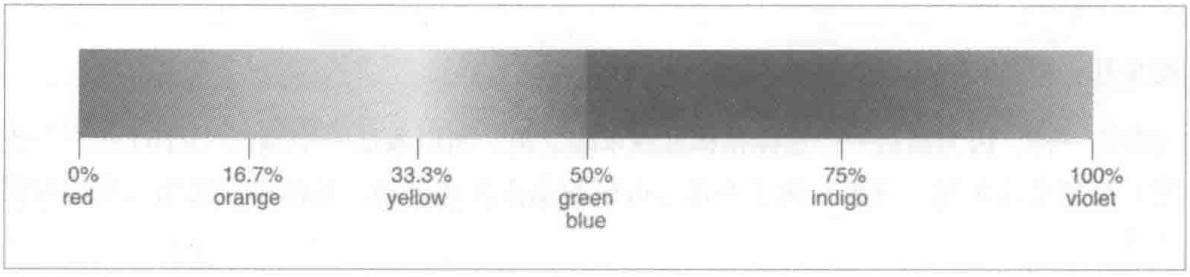


图 9-72：色标重叠的效果

渐变依然正常沿梯度线过渡，但是在 50% 处，由绿色直接变成蓝色，跨度为零。因此，渐变从 33.3% 处的黄色（0% 到 50% 之间的三分之二处）过渡到 50% 处的绿色，然后立即变成蓝色，再由 50% 处的蓝色过渡到 75% 处（50% 和 100% 的中点）的靛蓝色。

这种“急变”效果可用于实现条纹，如图 9-73 所示，所用的代码如下：

```
.stripes {background-image: linear-gradient(90deg,
    gray 0%, gray 25%,
    transparent 25%, transparent 50%,
    gray 50%, gray 75%,
    transparent 75%, transparent 100%);}
```

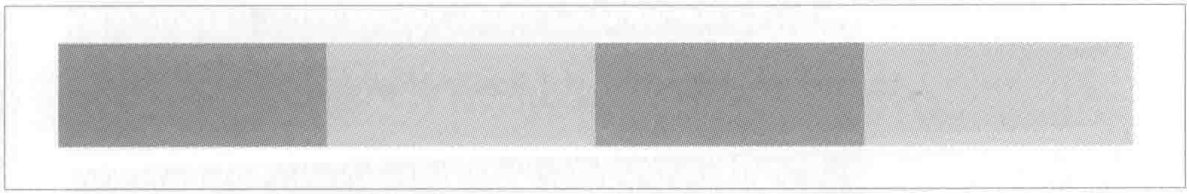


图 9-73：“急停”效果实现的条纹

把两个色标叠放在同一位置是这种情况，那么把后面的色标放在前一个色标之前呢？比如这样：

```
#spectrum {background-image: linear-gradient(90deg,
    red 0%, orange, yellow, green 50%, blue 40%, indigo, violet)};
```

此时，有违常规的色标（这里的 blue）将放在之前色标中位置值最大的位置上。在这个示例中，blue 的位置值为 50%，因为这是前面那个色标的位置。因此，这与前面把绿色和蓝色叠放在一处的效果是一样的。

这里的关键是，色标的位置将设为之前明确指定位置的色标中最大的那个。因此，在下述规则中，indigo 色标的位置将被设为 50%：

```
#spectrum {background-image: linear-gradient(90deg,
    red 0%, orange, yellow 50%, green, blue, indigo 33%, violet)};
```

这里，在 indigo 色标前面指定的最大位置是 yellow 色标的 50%。因此，这个渐变先从红色过渡到橘色再过渡到黄色，然后立即变成靛蓝色，再过渡到蓝紫色。从黄色到绿色到蓝色再到靛蓝色的过渡都只跨零距离。结果如图 9-74 所示。

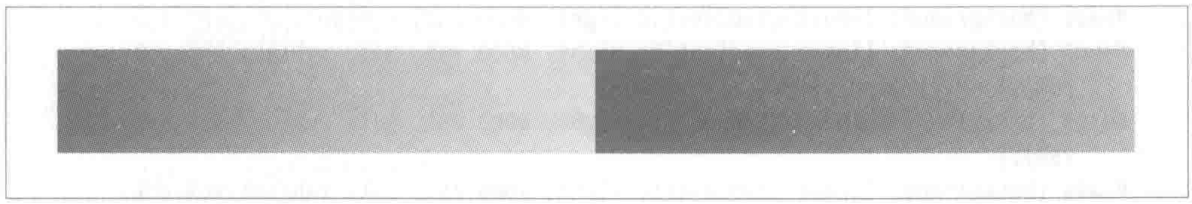


图 9-74：处理不正常位置上的色标

正是因为这样，一般才不建议在一个渐变中混用不同的单位。假如混用 rem 和百分数，有可能使用百分数定位的色标在前面使用 rem 定位的色标之前。

设置中色点

现在，我们知道色标的作用了。然而，你可能还记得，线性渐变的句法还支持在每个色标后设置“中色点”（color hint）：

```
linear-gradient(
    [[ <angle> | to <side-or-quadrant> ],]? [ <color-stop> [, <color-hint>]? ]# ,
    <color-stop>
)
```

<color-hint> 的作用是修改两侧两个色标的混合模式。默认情况下，两个色标之间的混合模式是线性的。因此，下述规则得到的结果是图 9-75 那样：

```
linear-gradient(
    to right, #000 25%, rgb(90%,90%,90%) 75%
)
```

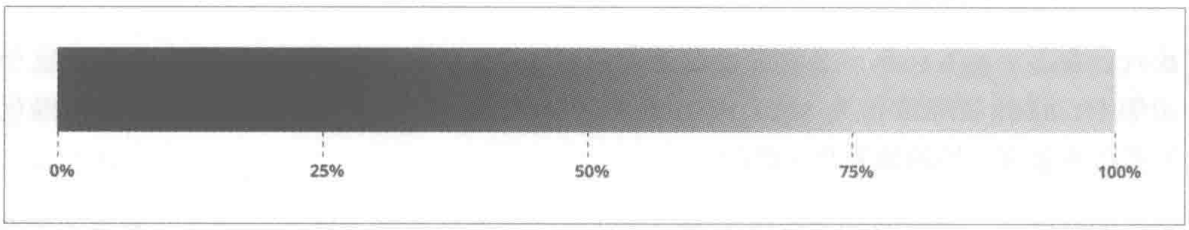


图 9-75：两个色标之间的线性混合模式

从 25% 处到 75% 处的混合是恒定的线性累进过程，由黑色（#000）过渡到浅灰色（rgb(90%,90%,90%)）。二者的中点，即 50% 处，灰色的浓淡程度正好是两侧两个色标差值的一半，即 rgb(45%,45%,45%)。

通过中色点可以改变累进的中点。例如，把原本该在中点处的 rgb(45%,45%,45%) 移到两个色标之间的其他位置。下述 CSS 得到的结果如图 9-76 所示。

```
#ex01 {background: linear-gradient(to right, #000 25%, rgb(90%,90%,90%) 75%);}
#ex02 {background: linear-gradient(to right, #000 25%, 33%, rgb(90%,90%,90%)
    75%);}
#ex03 {background: linear-gradient(to right, #000 25%, 67%, rgb(90%,90%,90%)
    75%);}
#ex04 {background: linear-gradient(to right, #000 25%, 25%, rgb(90%,90%,90%)
    75%);}
#ex05 {background: linear-gradient(to right, #000 25%, 75%, rgb(90%,90%,90%)
    75%);}
```

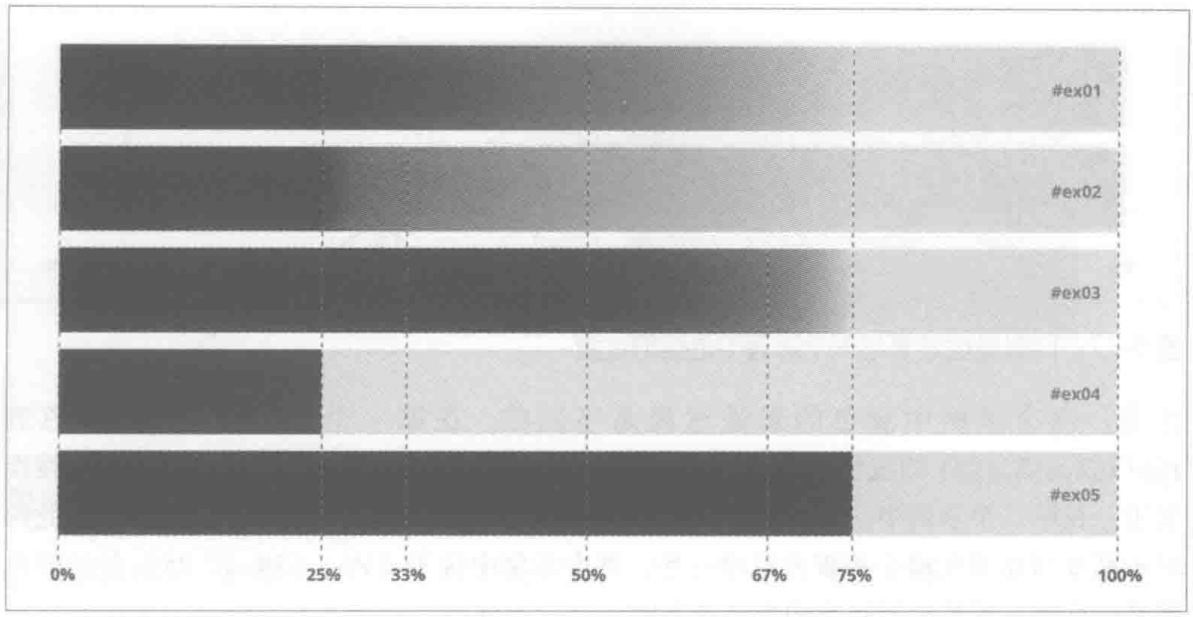


图 9-76：不同中色点的黑色到灰色渐变

在第一个示例中（#ex01），使用的是默认的线性累进，中色（45% 黑）在两个色标的中点。

在第二个示例中（#ex02），中色在梯度线的 33% 处。第一个色标在 25% 处，中色在 33% 处，第二个色标在 75% 处。

在第三个示例中（#ex03），中色点在梯度线的 67% 处。因此，渐变从 25% 处的黑色过渡到 67% 处的中色，然后再从 67% 处的中色过渡到 75% 处的浅灰色。

第四个示例和第五个示例展示把中色点放在色标的位置上会得到什么结果。可以看出，结果是颜色“急变”。

中色点有个不容易理解的行为，即色标到中色点再到色标的累进过程不是两个线性累进这么简单，而是带有一点“曲折”。下面通过示例说明。如下两个规则看似创建的是相同的渐变，但其实不然，如图 9-77 所示。

```
#ex01 {background:
    linear-gradient(to right,
        #000 25%,
        rgb(45%,45%,45%) 67%,    /* 这是一个色标 */
        rgb(90%,90%,90%) 75%);}
#ex02 {background:
    linear-gradient(to right,
        #000 25%,
        67%,                     /* 这是一个中色点 */
        rgb(90%,90%,90%) 75%);}
```

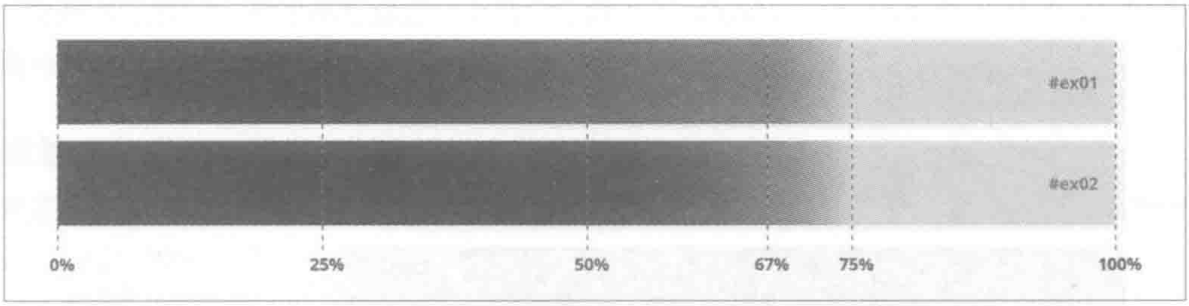


图 9-77：两次线性渐变与一个带有中色点的过渡

注意，两个示例中灰色的累进过程是不同的。在第一个示例中，先是黑色到 rgb(45%,45%,45%) 的线性累进，然后是 rgb(45%,45%,45%) 到 rgb(90%,90%,90%) 的线性累进。在第二个示例中，黑色到浅灰色的累进距离相同，但是中色点在 67% 处，因此得到的渐变将尽量在整个范围内平滑一些。两个示例中位于 25%、67% 和 75% 处的颜色都是一样的，但是浓淡程度的变化却不同。



熟悉动画的人可能会想在中色点上应用渐进函数（例如 ease-in），进一步控制颜色的混合方式。但是截至 2017 年年末还不可以这么做，不过这一功能正在讨论中。

梯度线剖析

掌握色标位置的基本知识之后，该讲解梯度线是如何构建的以及是如何产生所需效果的了。

下面设置一个简单的渐变，以此为例进行说明：

```
linear-gradient(
    55deg, #4097FF, #FFBE00, #4097FF
)
```

那么，这个一维结构（斜 55 度的直线）是如何创建两维渐变填充的呢？首先，根据指定的角度放置梯度线，确定起点和终点。这条梯度线及由它得到的渐变如图 9-78 所示。

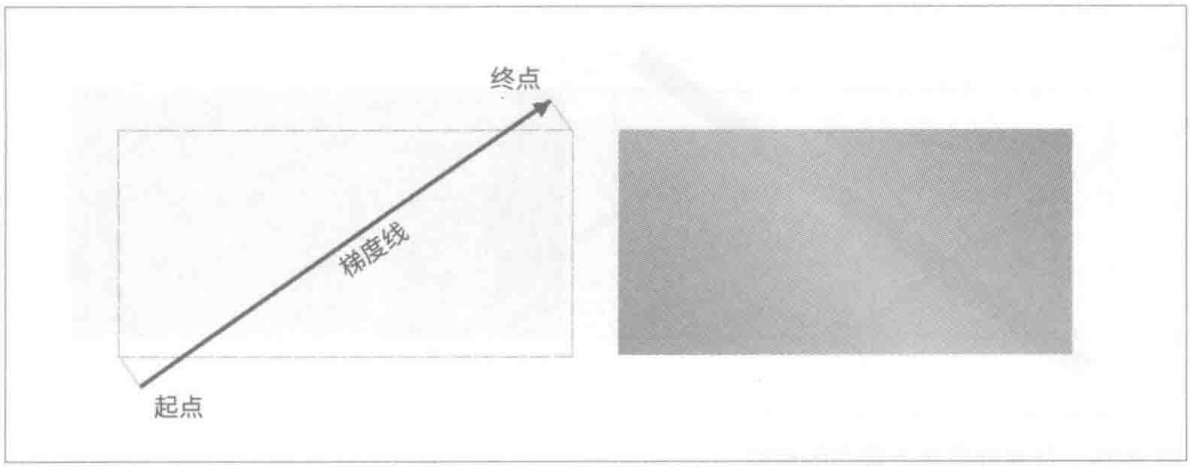


图 9-78：梯度线的位置和大小

首先要明确的一点是，图中的方框不是元素，而是线性渐变图像自身（记得吗，渐变创建的是图像）。渐变图像的尺寸和形状由很多因素决定，至于是由元素背景的尺寸还是 background-size 属性的值决定，稍后再讨论。现在我们只关注图像。

好的，从图 9-78 可以看出，梯度线经过图像的中心点。梯度线始终经过渐变图像的中心点。这里，我们把梯度线的角度设为 55 度，因此梯度线指向圆周上的 55 度方向。起点和终点比较特殊，它们都在图像范围之外。

先看起点。为了确定起点的位置，先找出图像中距梯度线方向（55deg）最远的那个角，然后过那一顶点做梯度线的垂线，与梯度线相交的点就是起点。相反地，为了确定终点的位置，先找出图像中距梯度线方向最近的那个角，然后过那一顶点做梯度线的垂线，与梯度线相交的点便是终点。

注意，“起点”和“终点”这样的说法并不确切，梯度线其实不会在某一点结束。梯度线是无限延伸的。然而，起点是第一个色标的默认位置，对应的位置值是 0%。类似地，终点对应的位置值是 100%。

因此，对于前面定义的这个渐变来说：

```
linear-gradient(
    55deg, #4097FF, #FFBE00, #4097FF
)
```

位于起点的颜色是 #4097FF，位于中点（也是图像的中心点）的颜色是 #FFBE00，而位于终点的颜色是 #4097FF，这几点之间的颜色则平滑过渡。结果如图 9-79 所示。

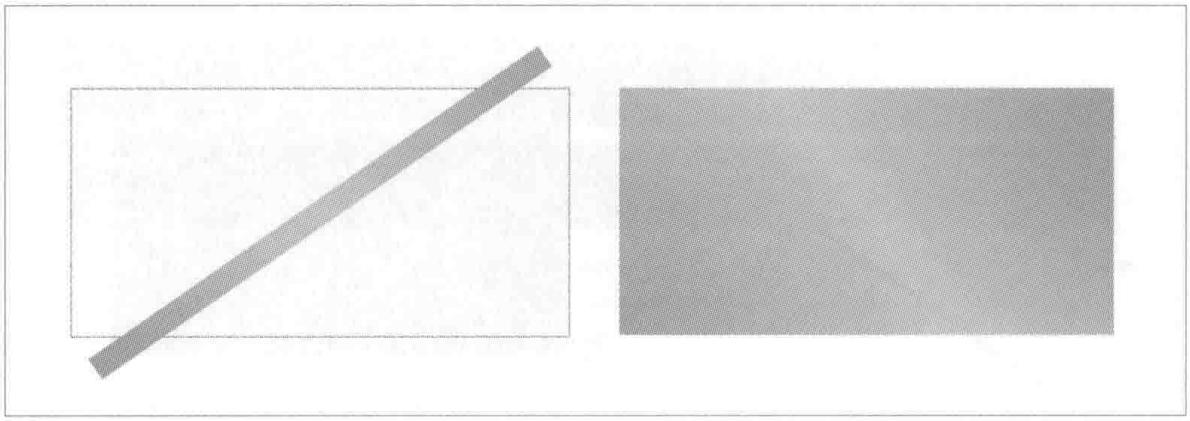


图 9-79：计算梯度线上各点的颜色

目前都好理解。但是，你可能会问，既然起点和终点都在图像的范围之外，那么图像的左下角和右上角怎么会是蓝色呢，蓝色在起点和终点的位置上啊？这是因为梯度线上各点的颜色会沿着与梯度线垂直的方向向外延伸。部分点沿着垂线方向向外延伸的颜色如图 9-80 所示，这里包括起点和终点，以及梯度线上每隔 5% 距离的点。

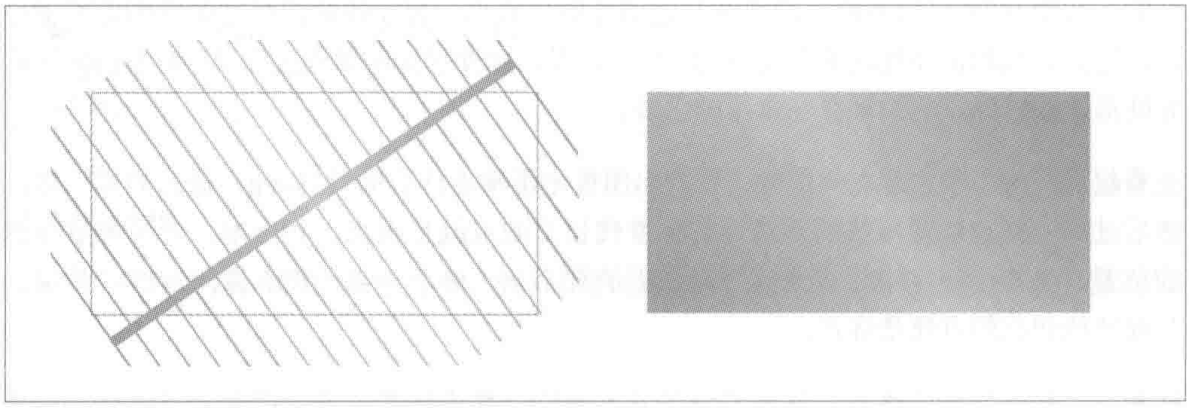


图 9-80：梯度线上部分颜色向外延伸的方式

其他点的延伸方式你应该能想象出来了。下面来看一些其他情况。我们还使用前面那个渐变，但是现在分别生成较宽、方形和较高的图像，如图 9-81 所示。注意，不管渐变图像是什么形状，起点和终点的颜色始终出现在对角处。

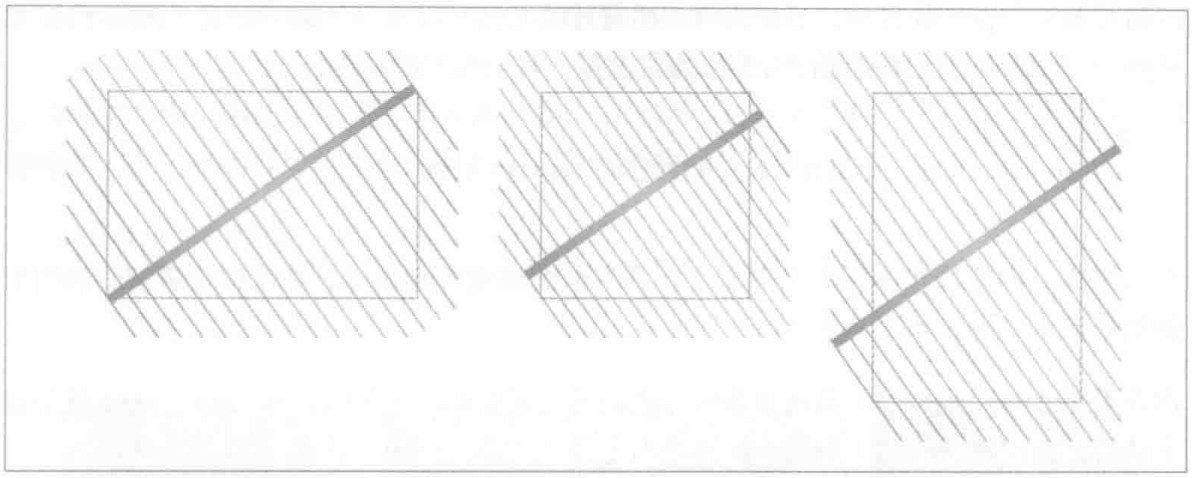


图 9-81：创建不同尺寸的渐变图像

注意，笔者用词比较谨慎，我说的是“起点和终点的颜色”，而没有说“开头和末尾的颜色”。原因前面已经说过，即色标可以放在起点之前，也可以放在终点之后，如下所示：

```
linear-gradient(
    55deg, #4097FF -25%, #FFBE00, #4097FF 125%
)
```

上例中各色标的位置、起点和终点的位置，梯度线上各点计算得到的颜色，以及最终得到的渐变如图 9-82 所示。

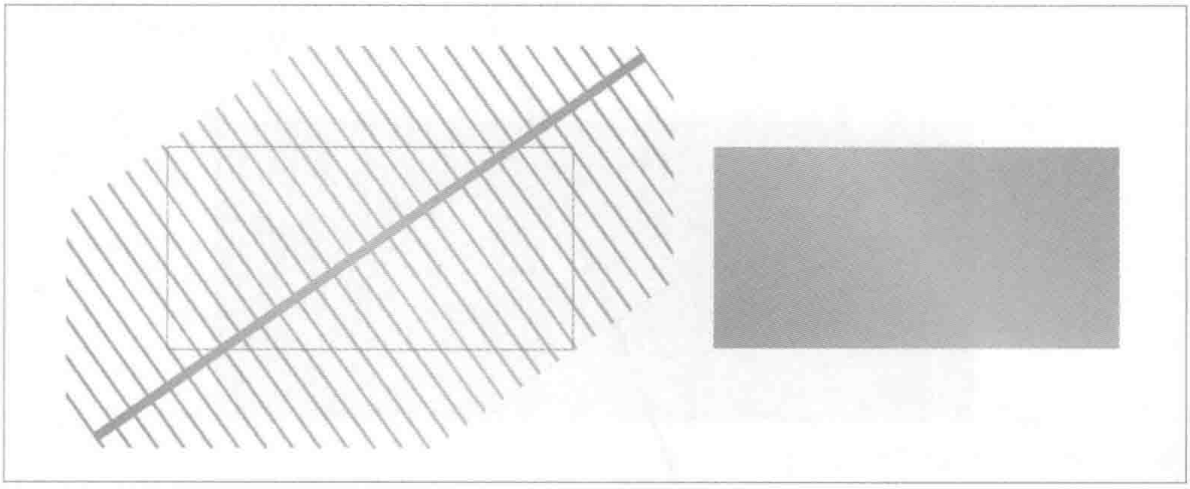


图 9-82：色标超出起点和终点的渐变

同样，左下角和右上角的颜色依然与起点和终点的颜色一致。只不过这里第一个色标在起点之前，起点的颜色是前两个色标混合后的颜色。终点的情况类似，是后两个色标混合后的颜色。

下面来看一个怪异的行为。还记得吗，梯度线的方向可以使用关键字指定，例如 top 和 right？假设你想让梯度线指向图像的右上角，写出如下的渐变：

```
linear-gradient(
    to top right, #4097FF -25%, #FFBE00, #4097FF 125%
)
```

事与愿违，梯度线并不与右上角相交。真实情况确实有点怪异。为了便于讲解，我们先把结果展示出来，如图 9-83 所示。

你没看错，梯度线离右上角还远着呢，指向的是图像的右上象限。to top right 就是这个意思：指向图像的右上象限，而不是右上角。

如图 9-83 所示，梯度线的具体位置是这样确定的：

1. 过图像的中点把声明的象限中的那个角两侧相邻的角连接起来。因此，对右上象限来说，相邻的两个角是左上角和右下角。
2. 梯度线与那条线垂直，指向声明的象限。
3. 最后像往常那样构建渐变，即确定起点和终点、放置或排布色标，然后得出整个渐变图像。

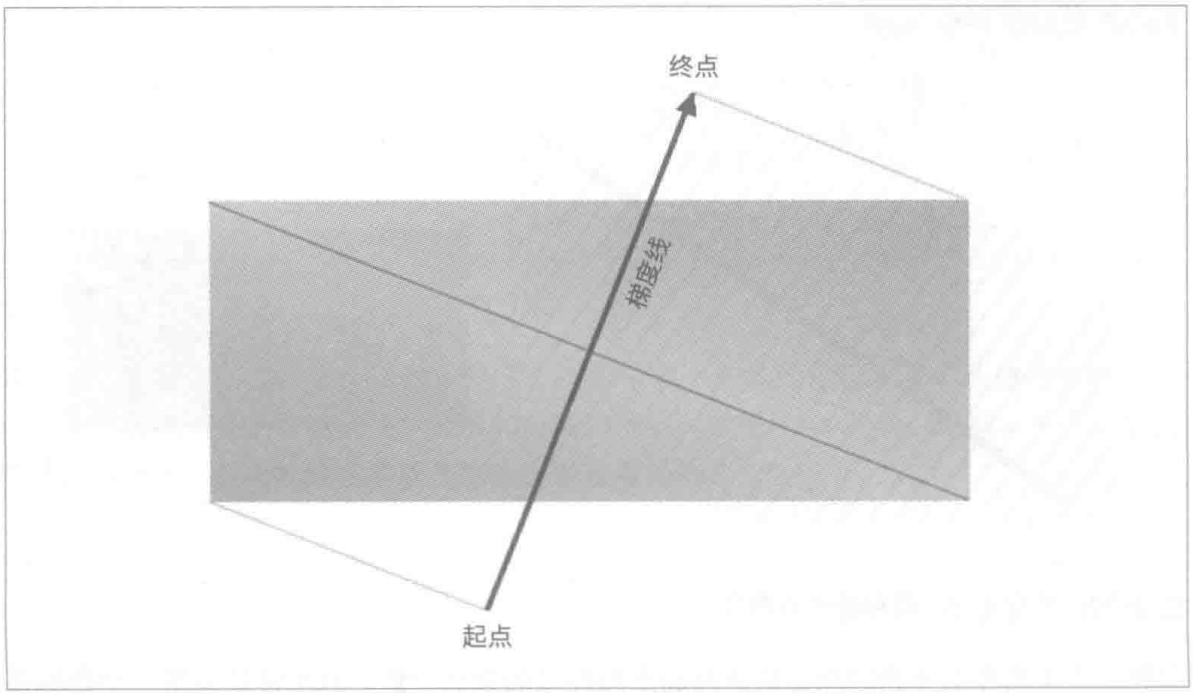


图 9-83：指向右上象限的渐变

这样的处理方式有些副作用。第一，中点的颜色始终从所指象限两侧的一个角延伸到另一个角。第二，如果图像的形状变了，即宽高比变了，梯度线的方向也会变，因此渐变也将改变方向。因此，使用弹性元素时要小心。第三，方形渐变图像的梯度线将与一个角相交。这三个副作用的效果见图 9-84，所用的渐变定义如下：

```
linear-gradient(
    to top right, purple, green 49.5%, black 50%, green 50.5%, gold
)
```

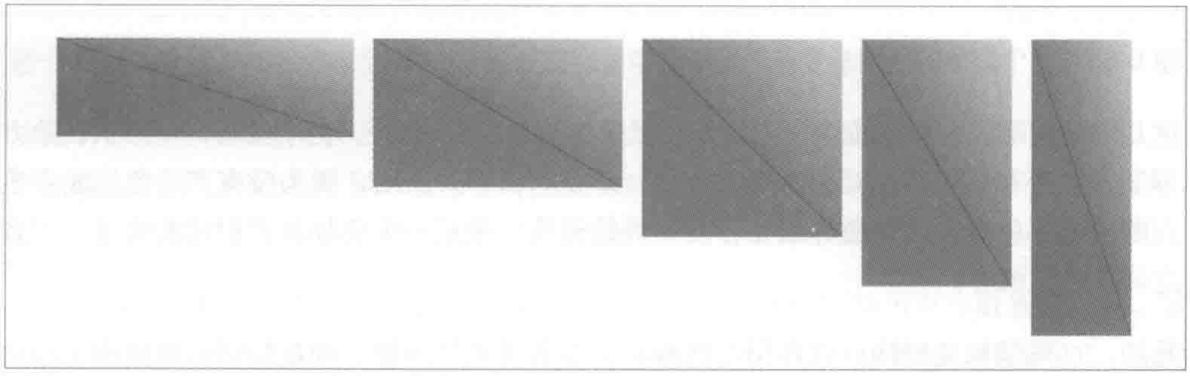


图 9-84：指向某个象限的渐变带来的副作用

可惜，我们无法为非方形图像设定指向某一角的梯度线，只能自己计算具体的角度，然后显式声明。而且，如果图像的尺寸是不定的，还要借助 JavaScript。

9.3.2 径向渐变

线性渐变能实现特别棒的效果，不过有时你可能想要圆形渐变，比如说实现聚光灯、圆形阴影、圆形发光等效果。径向渐变的句法与线性渐变类似，不过也有一些区别：

```
radial-gradient(
    [ [ <shape> || <size> ] [ at <position>]? , | at <position>, ]?
      [ <color-stop> [, <color-hint>]? ] [, <color-stop> ]+
)
```

简单来说，你可以声明形状和尺寸（可选），可以声明渐变的中心点在何处（可选），然后声明两个或多个色标，色标之间还可以指定中色点（可选）。形状和尺寸还有些选项，下面逐一说明。

先来看一个简单的径向渐变（有可能是最简单的了）在不同形状的元素中呈现的效果（见图 9-85）：

```
.radial {background-image: radial-gradient(purple, gold);}
```

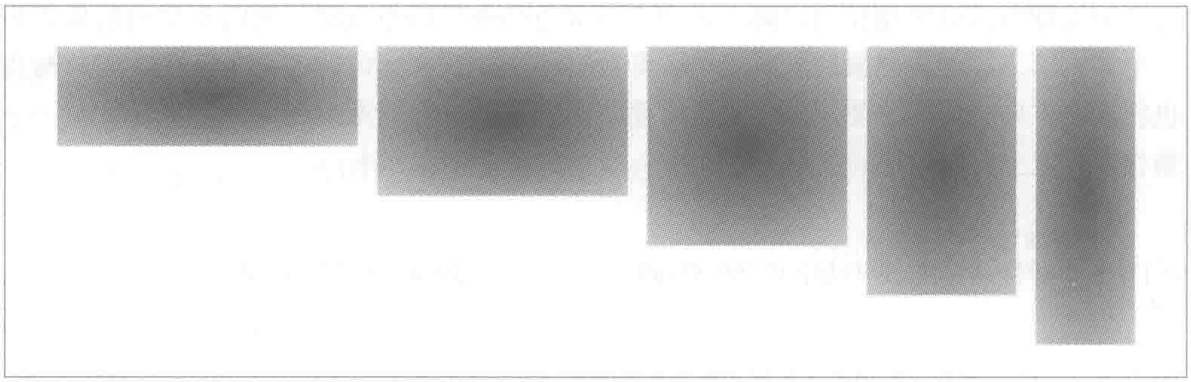


图 9-85：一个简单的径向渐变在不同形状中呈现的效果

这几种情况都没有声明位置，因此使用默认的 center。此外，由于没有声明形状，除方形元素之外都是椭圆形；在方形元素中，形状为圆形。最后，因为没有声明色标或中色点的位置，所以第一个色标放在梯度射线的开头，最后一个色标放在射线的末尾，二者之间为线性混合。

是的，这里是梯度射线，其作用与线性渐变中的梯度线一样。梯度射线从渐变的中心向右延伸，渐变的其他部分据此构建（稍后详细说明梯度射线）。

形状和尺寸

首先，径向渐变只有两种可用的形状值（因此也就只能有两种形状），即 circle 和 ellipse。径向渐变的形状可以显式声明，也可以由渐变图像的尺寸推导出来。

再看尺寸。一同往常，指定径向渐变的尺寸时可以只提供一个非负长度值（得到的是圆形），也可以提供两个非负长度值（得到的是椭圆形）。假如有下述径向渐变：

```
radial-gradient(50px, purple, gold)
```

这是个圆形径向渐变，由中心点处的紫色过渡到 50 像素以外的金色。如果再添加一个长度值，得到的将是椭圆形渐变，宽度与第一个长度相等，高度与第二个长度相等：

```
radial-gradient(50px 100px, purple, gold)
```

这两个渐变如图 9-86 所示。

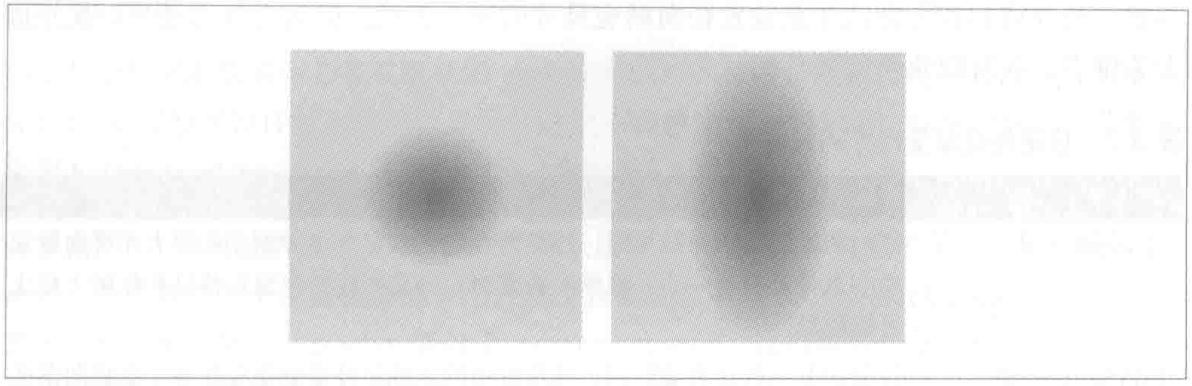


图 9-86：两个简单的径向渐变

注意，渐变的形状与所在图像的尺寸和形状没有任何关系。如果创建的是圆形渐变，得到就是一个圆，即便在矩形渐变图像中也是如此。同样，椭圆渐变得到的始终是椭圆形，即便在方形渐变图像中也是如此（不过看起来像是圆形，因为椭圆的宽度和高度相等）。

尺寸也可以使用百分数值，不过只能用于设定椭圆形。圆形不能用百分数指定尺寸，因为无法确定百分数相对哪一轴计算（比如对高 100 像素、宽 500 像素的图像来说，10% 相当于 10 像素还是 50 像素呢）。如果试图为圆形渐变设置百分数值，整个声明都将失效。

为椭圆渐变设置百分数值时，跟之前一样，第一个值相对横轴计算，第二个值相对纵轴计算。下述渐变在不同尺寸的图像中得到的结果如图 9-87 所示：

```
radial-gradient(50% 25%, purple, gold)
```

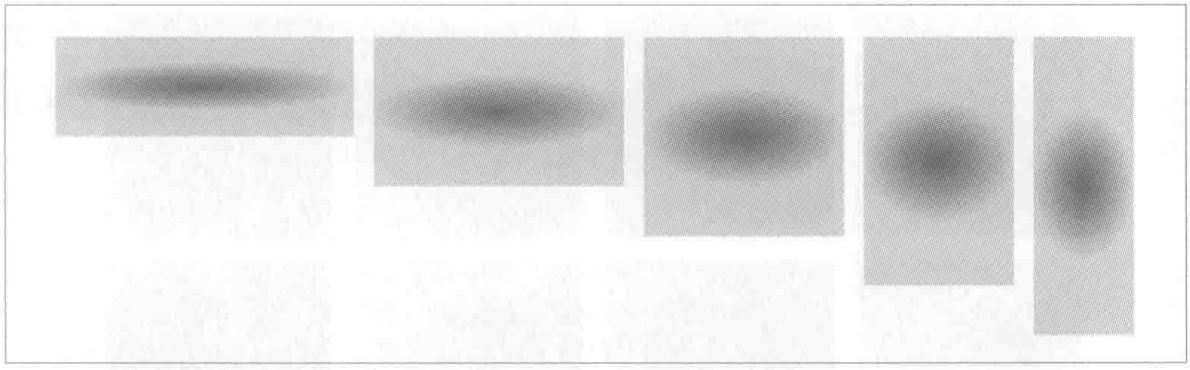


图 9-87：使用百分数设定椭圆渐变的尺寸

设置椭圆渐变的尺寸时，还可以混用长度值和百分数值，当然注意事项也跟之前一样。如果比较确定，完全可以把椭圆径向渐变的高度设为 10 像素，宽度设为元素宽度的一半，如下所示：

```
radial-gradient(50% 10px, purple, gold)
```

当然，长度值和百分数值不是设定径向渐变尺寸的唯一方式。除此之外，还可以使用四个关键字，各自的效果见表 9-3。

表 9-3：设定径向渐变尺寸的关键字

关键字	作用
closest-side	径向渐变的形状为圆形时，梯度射线的末端正好接触渐变图像上距径向渐变中心点最近的那一边。形状为椭圆时，梯度射线的末端在横轴和纵轴上都正好接触最近的边
farthest-side	径向渐变的形状为圆形时，梯度射线的末端正好接触渐变图像上距径向渐变中心点最远的那一边。形状为椭圆时，梯度射线的末端在横轴和纵轴上都正好接触最远的边
closest-corner	径向渐变的形状为圆形时，梯度射线的末端正好接触渐变图像上距径向渐变中心点最近的那一角。形状为椭圆时，梯度射线的末端依然正好接触最近的那一角，而且椭圆的宽高比与设为 closest-side 时一样
farthest-corner（默认值）	径向渐变的形状为圆形时，梯度射线的末端正好接触渐变图像上距径向渐变中心点最远的那一角。形状为椭圆时，梯度射线的末端依然正好接触最远的那一角，而且椭圆的宽高比与设为 farthest-side 时一样。注意：这是径向渐变的默认尺寸值，如未声明尺寸值，就使用这个值

为了更清楚地理解这几个关键字的效果，请看图 9-88，图中包含各关键字应用到圆形和椭圆渐变上的效果。

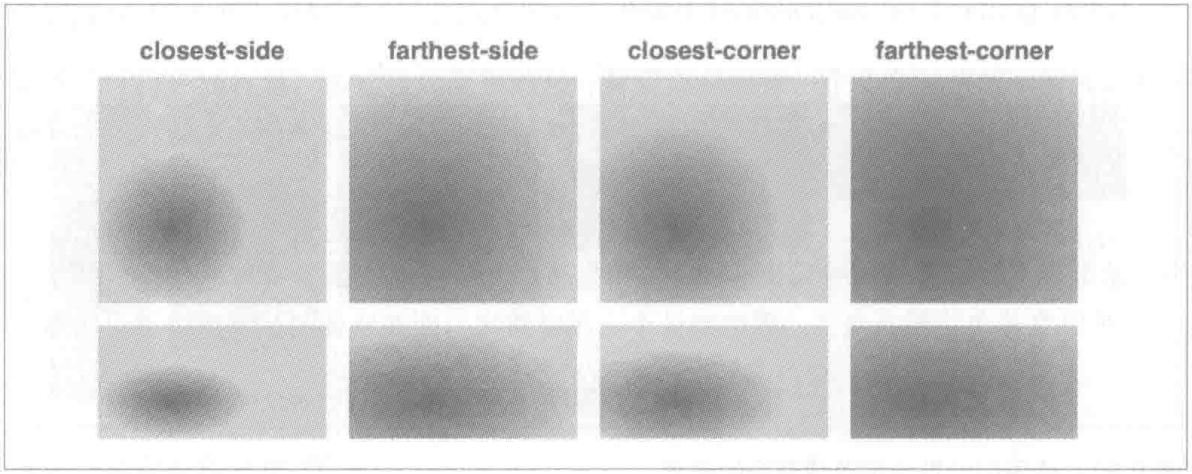


图 9-88：径向渐变各尺寸关键字的效果

设定椭圆径向渐变的尺寸时，关键字不能与长度值或百分数值混用。因此，closest-side 25px 是无效的，将被忽略。

从图 9-88 中你可能注意到了，渐变不是从图像的中心开始的。这是因为我们把渐变放到其他位置了，详情参见下一节。

定位径向渐变

如果想把径向渐变的中心放在默认的 center 以外的位置，可以使用对 background-position 属性来说任何有效的位置值。这里不再赘述复杂的句法，如果想回顾，请翻到介绍 background-position 属性那一节（参见 9.2.4 节）。

我所说的“任何有效的位置值”包括允许的长度值、百分数值和关键字等的任何有效组合。而且，如果省略两个位置值中的一个，那个位置的推导方式也与 background-position 属性一样。举个例子，center 等效于 center center。径向渐变位置与背景位置之间主要的区别是，前者的默认值是 center，而非 0% 0%。

下面举些例子，结果如图 9-89 所示。

```
radial-gradient(at bottom left, purple, gold);
radial-gradient(at center right, purple, gold);
radial-gradient(at 30px 30px, purple, gold);
radial-gradient(at 25% 66%, purple, gold);
radial-gradient(at 30px 66%, purple, gold);
```

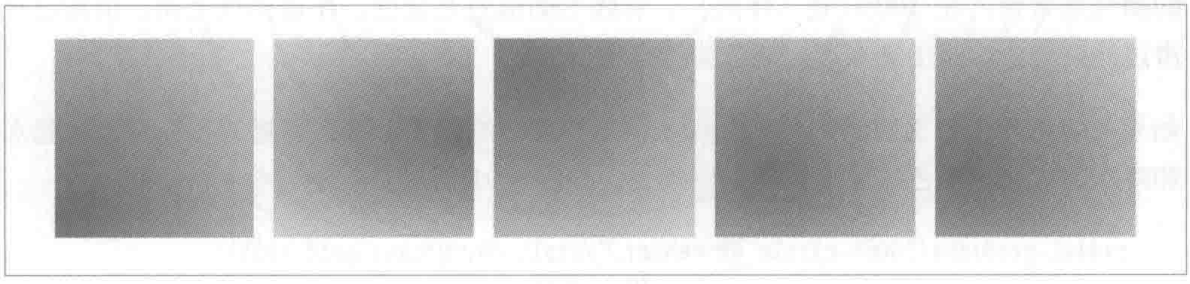


图 9-89：改变径向渐变的中心点位置

上述径向渐变都没有明确设定尺寸，因此默认为 farthest-corner。这种默认的行为是合理的，却不是唯一的可能。下面在上述渐变中添加尺寸，看看结果有何变化（见图 9-90）：

```
radial-gradient(30px at bottom left, purple, gold);
radial-gradient(30px 15px at center right, purple, gold);
radial-gradient(50% 15% at 30px 30px, purple, gold);
radial-gradient(farthest-side at 25% 66%, purple, gold);
radial-gradient(farthest-corner at 30px 66%, purple, gold);
```

很好。那么，如果想实现稍微复杂的渐变，不止两个颜色呢？下面探讨色标。

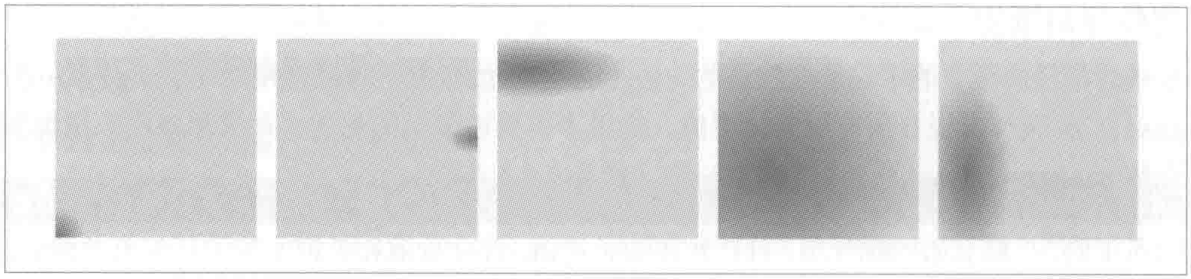


图 9-90：改变明确设定尺寸的径向渐变的中心点位置

径向渐变的色标和梯度射线

径向渐变色标的句法和处理方式与线性渐变一样。仍以可能是最简单的径向渐变为例，后面给出了明确声明的等效写法：

```
radial-gradient(purple, gold);
radial-gradient(purple 0%, gold 100%);
```

径向渐变的梯度射线从中心向外延伸。在 0% 处（起点，也是渐变的中心点），射线上的颜色是紫色。在 100% 处（终点），射线上的颜色是金色。在这两点之间，由紫色平滑过渡到金色；超出终点后，颜色为纯金色。

如果在紫色和金色之间添加一个色标，但不指定它的位置，那么新增的色标将放在起点和终点的中间，颜色的过渡也将相应改变，如图 9-91 所示。

```
radial-gradient(100px circle at center, purple 0%, green, gold 100%);
```

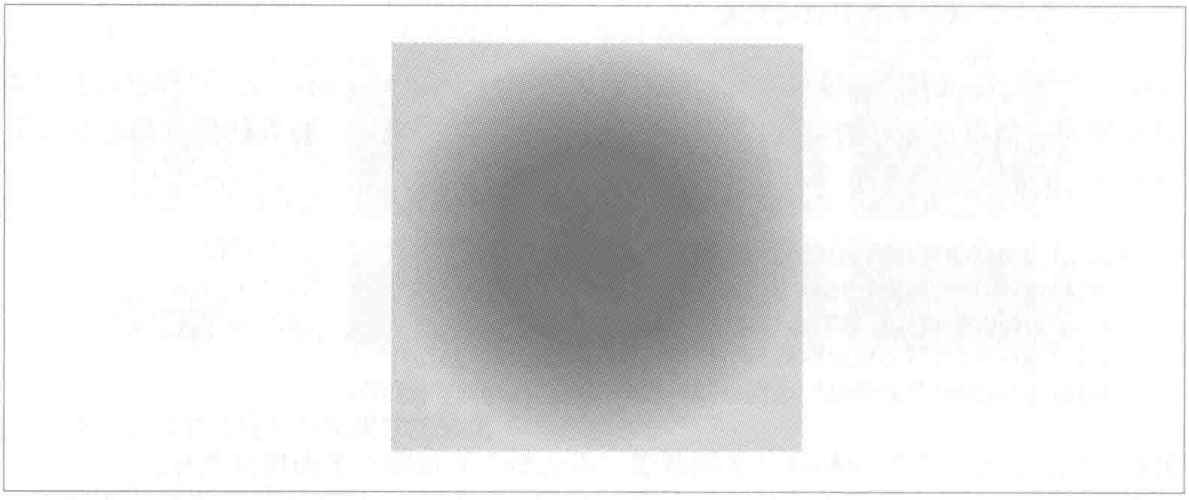


图 9-91：增加一个色标

如果写成 `green 50%`，得到的结果是一样的。我想你应该明白了。梯度射线上的颜色由紫色到绿色再到金色平滑过渡，超过射线终点后为纯金色。

据此可以看出（线性渐变的）梯度线和梯度射线之间的一个区别：线性渐变由梯度线上各点的颜色沿着与梯度线垂直的方向向两侧延伸得到；径向渐变的情况类似，不过是从梯度射线上向外延伸出无数条线，而且那些线是梯度射线终点构成的椭圆的同心小椭圆和大椭圆。具体方式如图 9-92 所示，图中的各个椭圆由梯度射线上不同的点向外延伸得到。

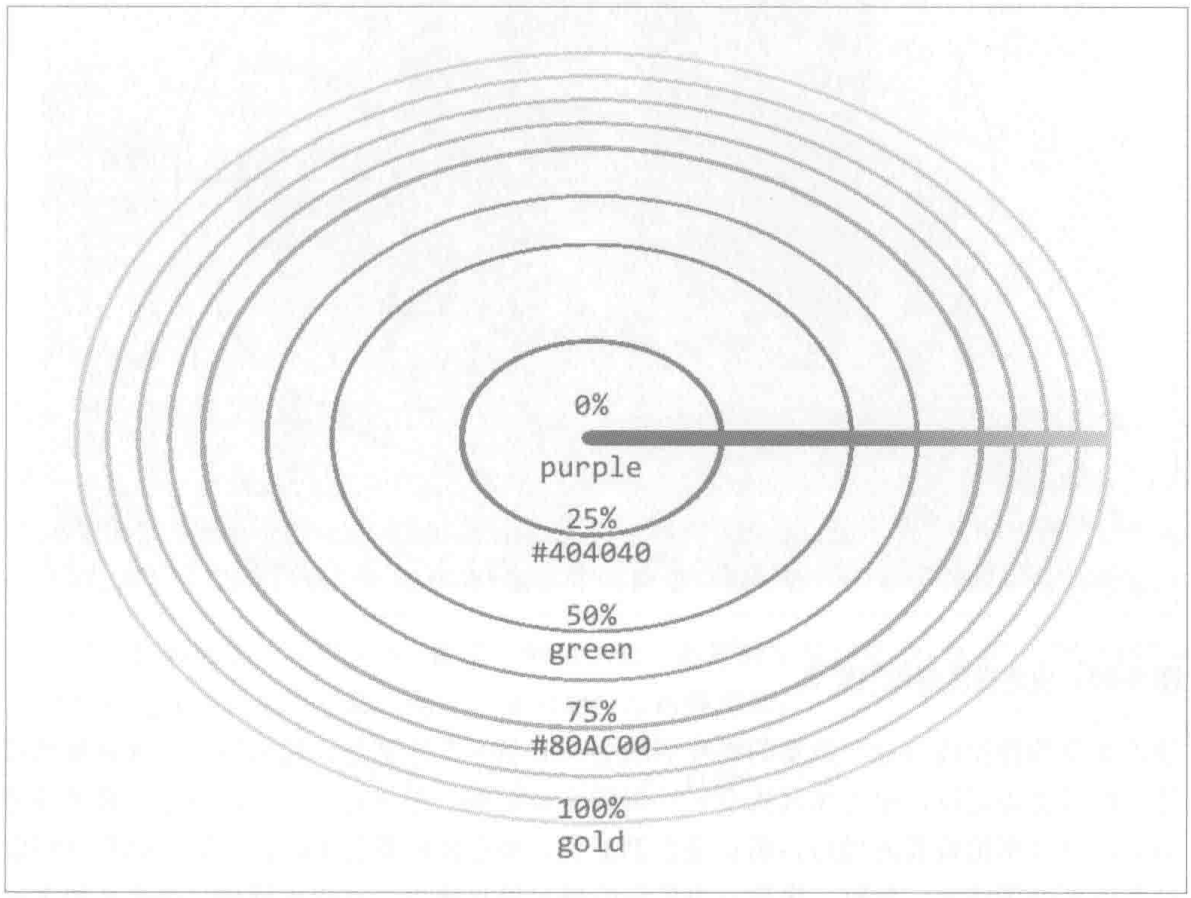


图 9-92：梯度射线和由其派生出来的一些椭圆

这就引出一个问题：梯度射线的终点（即 100% 处）是如何确定的呢？终点是梯度射线与所设尺寸得到的形状相交的那一点。圆形渐变的情况很简单，梯度射线的终点距形状中心点的距离就是尺寸值指定的长度。因此，对 25px circle 渐变来说，射线的终点距中心点 25 像素。

椭圆渐变的确定方式基本相同，不过距中心点的距离由椭圆的横轴决定。对 40px 20px ellipse 渐变来说，射线的终点在中心点正右方，二者相距 40 像素。详情如图 9-93 所示。

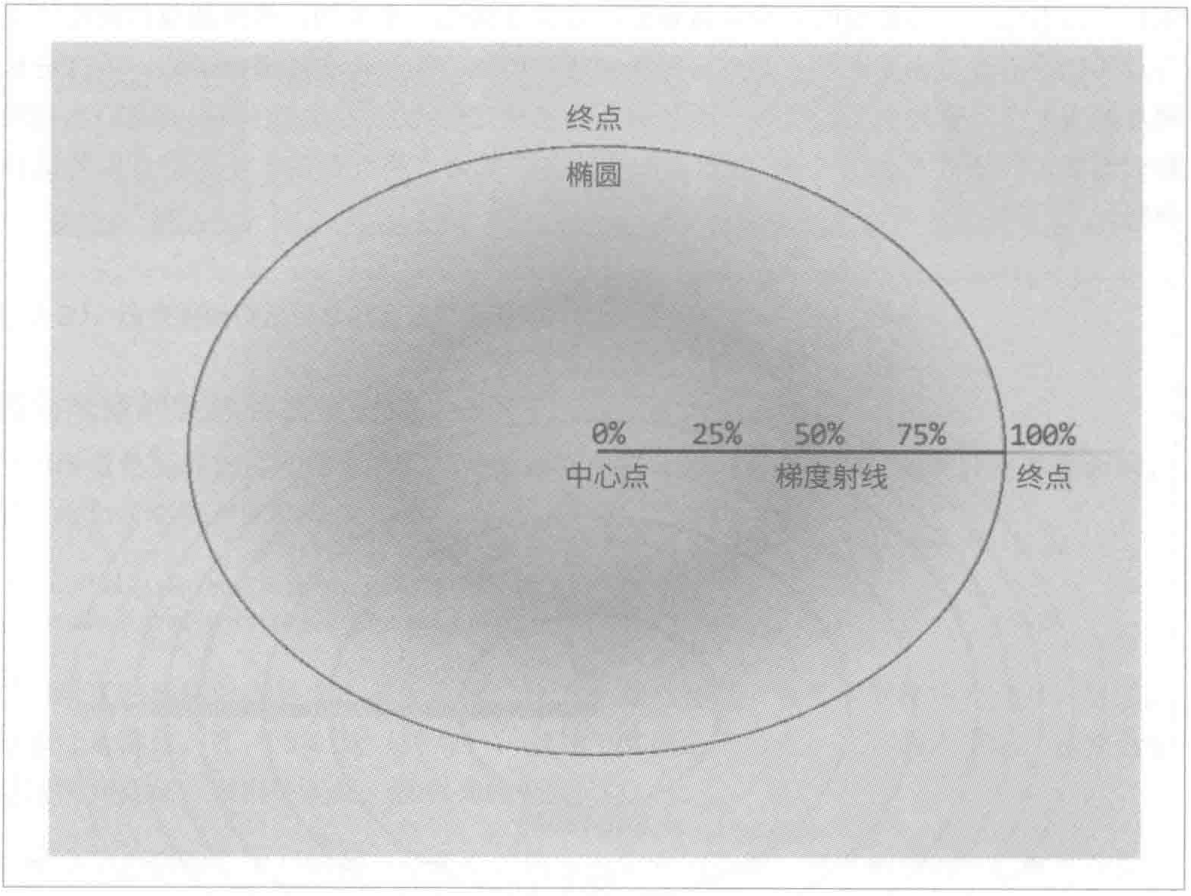


图 9-93：设定梯度射线的终点

线性渐变的梯度线与径向渐变的梯度射线之间的另一个区别是，超出终点的部分依然可见。你可能还记得，梯度线只从 0% 处画到 100% 处，超过这一点就没有了。梯度线绝不会比渐变图像最长的轴短，而且通常都更长。而在径向渐变中，径向形状的尺寸可以设为比渐变图像小。此时，最后一个色标的颜色将从终点一直向外延伸（前面几幅图很多都是这种情况）。

反过来，如果色标的位置超过了射线的终点，在那个色标之后可能还会看到对应的颜色。以下述渐变为例，其结果如图 9-94 所示：

```
radial-gradient(50px circle at center, purple, green, gold 80px)
```

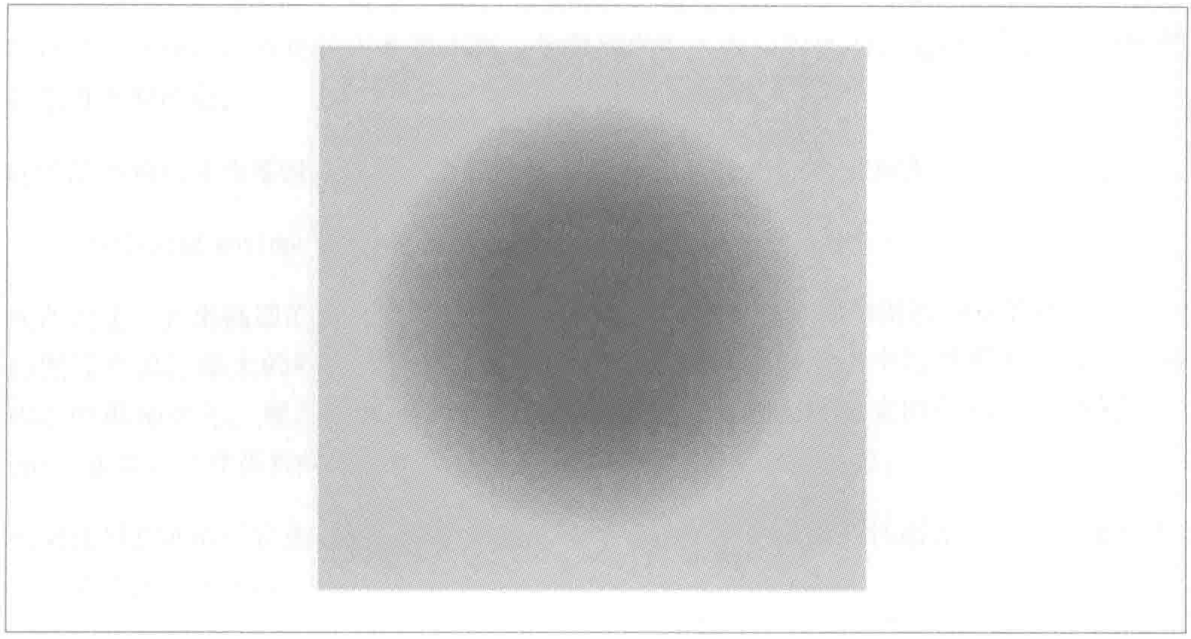


图 9-94：色标的位置超过终点

第一个色标没有指定位置，因此设为 0%，即中心点。最后一个色标的位置设为 80px，因此在各个方向上都距中心点 80 像素。位于中间的 green 色标在二者的中间（距中心点 40 像素）。因此，这个渐变在 80 像素处为金色，超出那一点之后的部分也是金色。

即便明确把圆形的尺寸设为 50 像素，依然如此。圆形的半径仍是 50 像素，只是最后一个色标的位置使得结果不那么明显。如果像下面这样声明：

```
radial-gradient(80px circle at center, purple, green, gold)
```

或者更简单一点，像这样：

```
radial-gradient(80px, purple, green, gold)
```

得到的结果不变。

使用百分数设定色标的位置时，也是这种情况。下面两个渐变与前面的示例得到的结果是一样的：

```
radial-gradient(50px, purple, green, gold 160%)
radial-gradient(80px, purple, green, gold 100%)
```

那么，如果把色标的位置设为负值呢？结果基本上与线性渐变一样：位置为负的色标用于确定起点的颜色，在结果中是不可见的。因此，下述渐变得到的结果如图 9-95 所示：

```
radial-gradient(80px, purple -40px, green, gold)
```

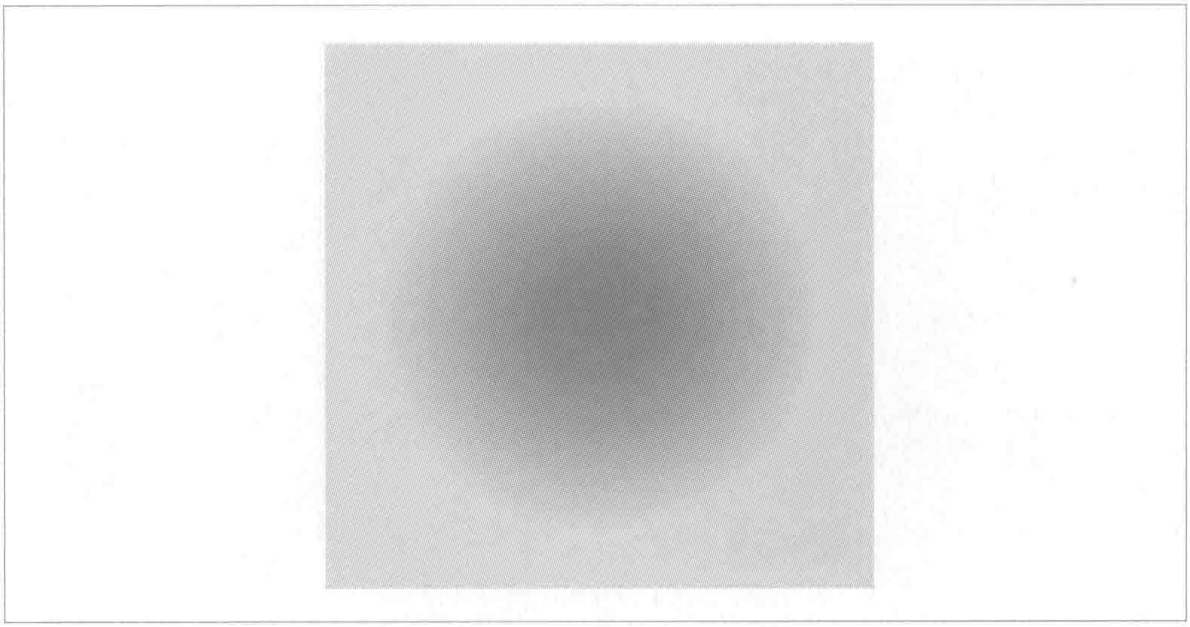


图 9-95：色标位置为负值时的处理方式

对上述渐变来说，第一个色标的位置是 -40px，最后一个色标的位置是 80px（因为最后一个色标没有显式声明位置，所以默认为终点），而中间那个色标的位置在二者的中点。上述渐变得到的结果与下面这样显式声明是一样的：

```
radial-gradient(80px, purple -40px, green 20px, gold 80px)
```

这个渐变的中心之所以是紫绿色的，是因为它由 1/3 紫色和 2/3 绿色混合而成。从中心点开始，先过渡到绿色，再过渡到金色。梯度射线上紫绿色混合的余下部分，即在“负值范围”内的那一部分，是不可见的。

极端情况

我们知道如何为径向渐变声明尺寸和位置了，但问题也随之而来了：如果圆形渐变的半径为零，或者椭圆渐变的高度或宽度为零会得到什么结果？这样的情况没有你想象中那么难出现，除了可以使用 0px 或 0% 把径向渐变的尺寸明确设为零之外，像下面这样的声明也能导致这样的结果：

```
radial-gradient(closest-corner circle at top right, purple, gold)
```

渐变的尺寸设为 closest-corner，而中心点移到 top right（右上）角了，因此最近的角距中心点的距离为零像素。这可怎么办呢？

规范十分明确地指出，遇到这种情况时应该按照“圆的半径为大于零的特别特别小的值”渲染渐变。这个半径有多小呢？可以是十亿分之一像素、一皮米或一普朗克长度（不懂？

问问你的科学老师吧）。但是，再小也不是零，因此渐变仍是圆形的，只不过这个圆形特别特别特别小，有可能根本看不到。如果确实看不到，得到的将是以最后一个色标的颜色填充的纯色。

椭圆渐变的尺寸为零时，规范定义的行为有所不同。以下述渐变为例：

```
radial-gradient(0px 50% at center, purple, gold)
```

规范规定，如果椭圆的宽度为零，渲染渐变时“把椭圆的高度当做特别大的值，把椭圆的宽度当做比零大的特别小的值”。也就是说，相当于渲染一个线性渐变，以过椭圆中心的纵轴对称。规范还说，在这种情况下，使用百分数值设定的色标位置将解析为0px。通常，这样得到的结果是以最后一个色标的颜色填充的纯色。

如果使用长度值设定色标的位置，得到的将是纵向对称的横向线性渐变。例如下述渐变，其结果如图 9-96 所示：

```
radial-gradient(0px 50% at center, purple 0px, gold 100px)
```

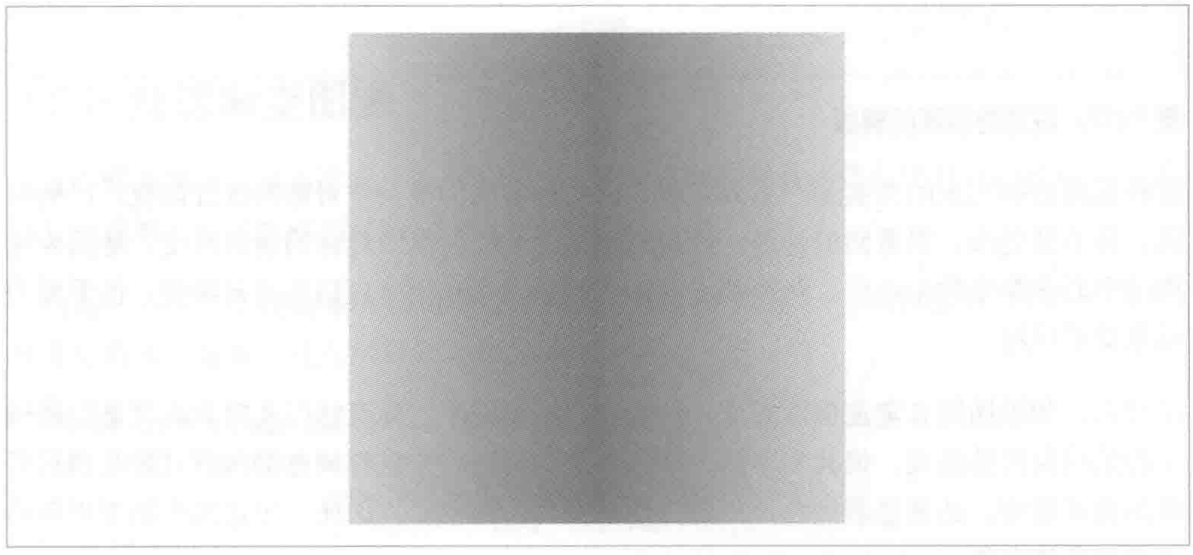


图 9-96：宽度为零的椭圆径向渐变

结果怎么会这样呢？首先，前面说过，规范规定，横向宽度为 0px 时，当做不为零但极小的一个值。为了便于说明，假设为千分之一像素（0.001 px）。此时，椭圆的宽度为千分之一像素，而高度为图像的一半。其次，为了便于说明，假设高度为 100 像素。因此，第一个椭圆形的宽度为千分之一像素，高度为 100 像素，即宽高比为 0.001:100，或 1:100000。

那么，沿梯度射线绘制的每个椭圆的宽高比都是 1:100000。这意味着，梯度射线上距起点 0.5 像素处的椭圆宽度为 1 像素，高度为 100000 像素；在 1 像素处，宽度为 2 像素，

高度为 200000 像素；在 5 像素处，宽度为 10 像素，高度为一百万像素；在 50 像素处，宽度为 100 像素，高度为一千万像素。以此类推。其他位置的尺寸如图 9-97 所示。

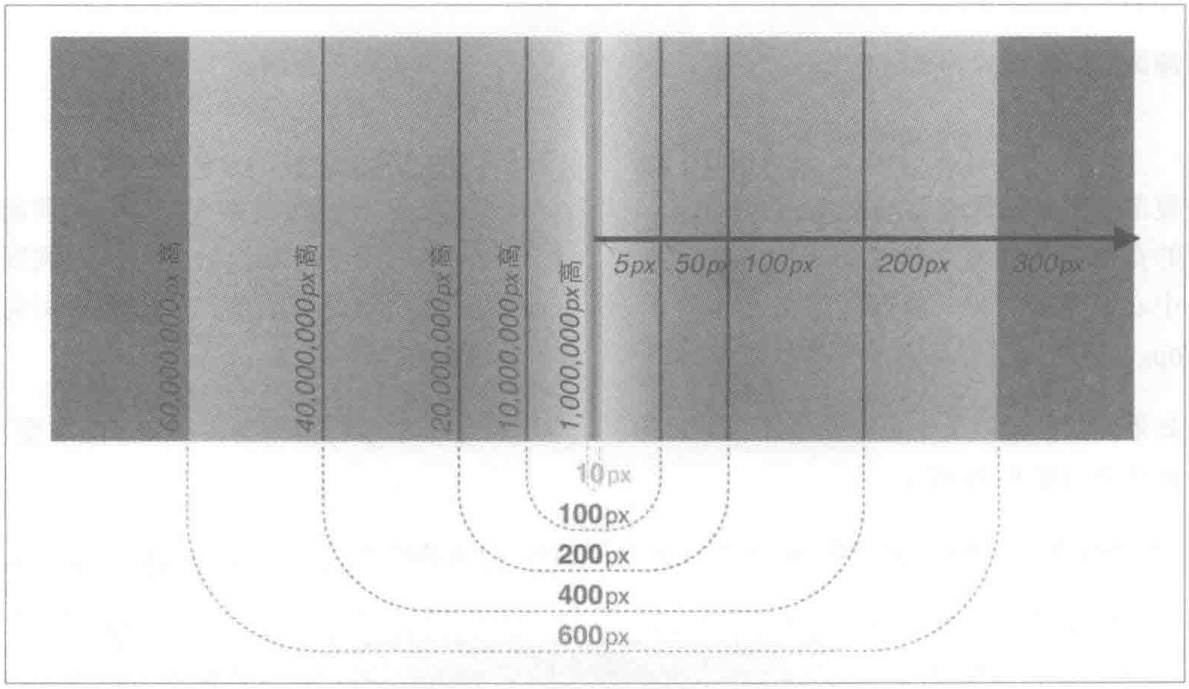


图 9-97：特别特别高的椭圆

这些椭圆绘制出来的其实是一系列竖线，因此你看到的是一个对称的线性渐变。严格来说，并不是竖线，但看到的却是。得到的结果之所以是纵向对称的横向渐变，是因为椭圆的中心在渐变的中心点，两边都绘制出了颜色。按理说这应该是径向渐变，但看到的结果却不尽然。

反过来，如果椭圆有宽度但没高度，结果又大不相同了。你可能以为得到的将是以横轴对称的纵向线性渐变，但其实不然。结果将是与最后一个色标同色的纯色（若是稍后要讲的循环渐变，结果也是纯色，但颜色是渐变的中值色）。因此，下述两个渐变得到的结果都是纯金色：

```
radial-gradient(50% 0px at center, purple, gold)
radial-gradient(50% 0px at center, purple 0px, gold 100px)
```

为什么高度为零时结果变了呢？这与构建径向渐变的梯度射线有关。同样，根据规范，零距离视作特别小的非零数。跟前面一样，我们假设 0px 被解析为 0.001px，而 50% 的计算结果为 100 像素。此时，宽高比为 100:0.001，即 100000:1。

因此，椭圆的高度为 1 像素时，宽度必定是 100000 像素。但是最后一个色标在 100 像素处。在那一点绘制的椭圆，宽度为 100 像素，高度为千分之一像素。紫色到金色的过

渡在那千分之一像素范围内也是有的。不过，在那之后的部分将是金色，即最后一个色标的颜色。因此，我们只能看到金色。

你可能以为，把最后一个色标的位置设为 100000px 时，将看到一条细长的紫色线横穿图像。确实可能，前提是浏览器把 0px 解析为 0.001px。如果浏览器把零距离解析为 0.00000001px，要把最后一个色标的位置向外移特别特别长一段距离才能看到那条细线。当然，这还是假设浏览器会计算并绘制这样的椭圆，而不是草草了事。说实话，更有可能是后一种情况。如果是我负责编写浏览器渲染渐变的代码，我就会这么做。

那么，如果椭圆的宽度和高度都是零呢？此时，规范规定，采用宽度为零的处理机制。因此，得到的将是对称的线性渐变。



截至 2017 年年末，浏览器对这些极端情况的支持还不稳定。这还算好的，有些浏览器甚至会在所有情况下都使用最后一个色标的颜色填充，而有些浏览器在某些情况下则拒绝绘制这样的渐变。

9.3.3 处理渐变图像

前文不断强调，渐变是图像。这意味着可以使用不同的背景属性设定渐变的尺寸、位置、重复方式等，就跟处理 PNG 或 SVG 图像一样。

利用这一点可以实现简单的重复渐变（复杂的重复方式在下一节讨论）。例如，可以使用急停的径向渐变实现点状背景，如图 9-98 所示：

```
body {background: tan center/25px 25px repeat
    radial-gradient(circle at center,
                    rgba(0,0,0,0.1), rgba(0,0,0,0.1) 10px,
                    transparent 10px, transparent);}
```

是的，这与平铺一个几乎透明的直径为 10 像素的深色圆形 PNG 图像得到的结果相当。不过，使用渐变有三点优势：

- 基本可以肯定的是，CSS 所占的字节比 PNG 图像小。
- 更为重要的是，PNG 图像要额外请求服务器。这会拖慢页面的加载时间，降低服务器的性能。CSS 渐变就在样式表中，无需额外请求服务器。
- 修改渐变简单得多，因此你可以不断试验，直到找出合适的尺寸、形状和明暗程度。

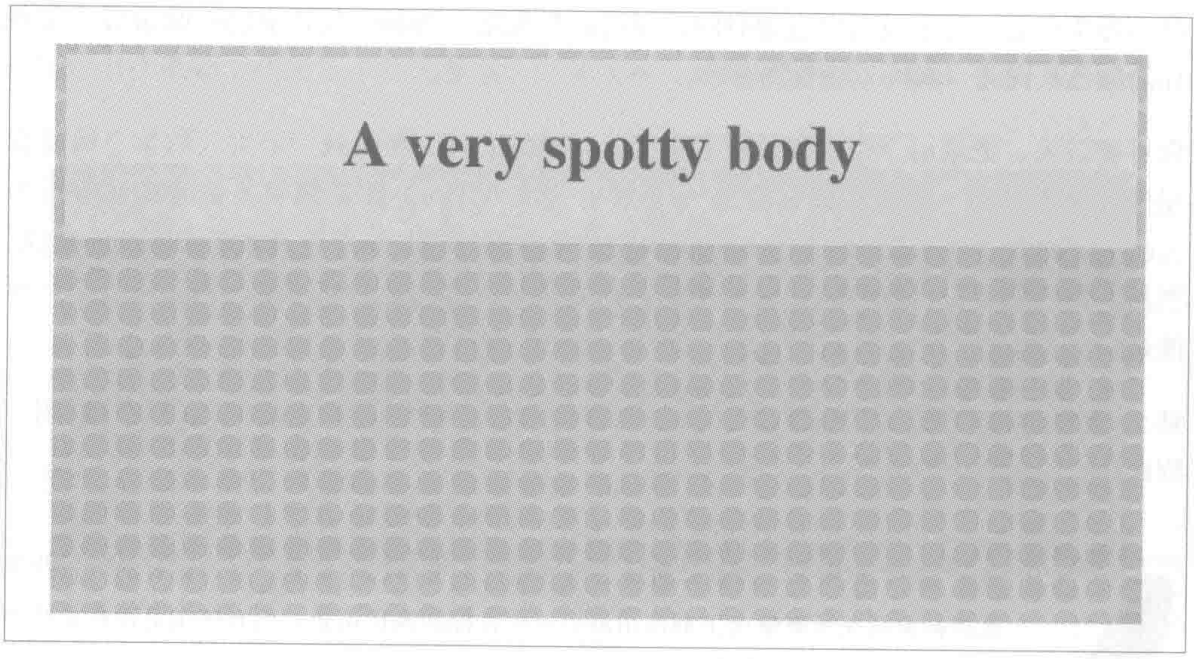


图 9-98：平铺径向渐变图像

虽然渐变功能强大，但是无法完全取代光栅图像或矢量图像，因此现在还不到彻底抛弃外部图像的时候。然而，使用渐变确实可以实现特别惊艳的效果。以图 9-99 中的背景效果为例。

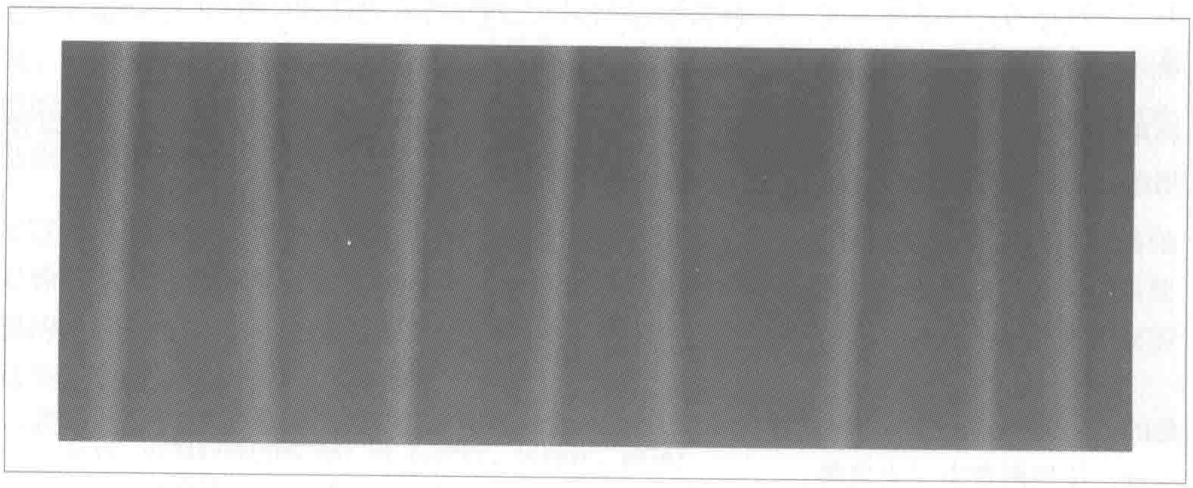


图 9-99：音乐准备……

这个幕布效果由三个线性渐变实现，其中两个渐变以不同的间隔重复，另一个沿背景的底部创建“发光”效果。所用的代码如下：

```
background-image:
    linear-gradient(0deg, rgba(255,128,128,0.25), transparent 75%),
    linear-gradient(89deg,
```

```
        transparent, transparent 30%,
        #510A0E 35%, #510A0E 40%, #61100F 43%, #B93F3A 50%,
        #4B0408 55%, #6A0F18 60%, #651015 65%, #510A0E 70%,
        #510A0E 75%, rgba(255,128,128,0) 80%, transparent),
    linear-gradient(92deg,
        #510A0E, #510A0E 20%, #61100F 25%, #B93F3A 40%, #4B0408 50%,
        #6A0F18 70%, #651015 80%, #510A0E 90%, #510A0E);
background-size: auto, 300px 100%, 109px 100%;
background-repeat: repeat-x;
```

第一个渐变（最上层那个）从 75% 透明的浅红色过渡到梯度线上 75% 处的完全透明。然后创建两个“褶皱”图像，如图 9-100 所示。

这些渐变图像沿 X 轴重复，而且都指定了尺寸。第一个渐变实现“发光”效果，尺寸设为 auto，以便覆盖元素的整个背景。第二个渐变的宽度设为 300px，高度设为 100%，因此，渐变图像将与元素的背景区域等高，宽度则为 300 像素。这意味着，在 X 轴上，这个渐变每 300 像素出现一次。第三个渐变图像也是如此，每 109 像素出现一次。最终得到的结果是一个不平整的舞台幕布。

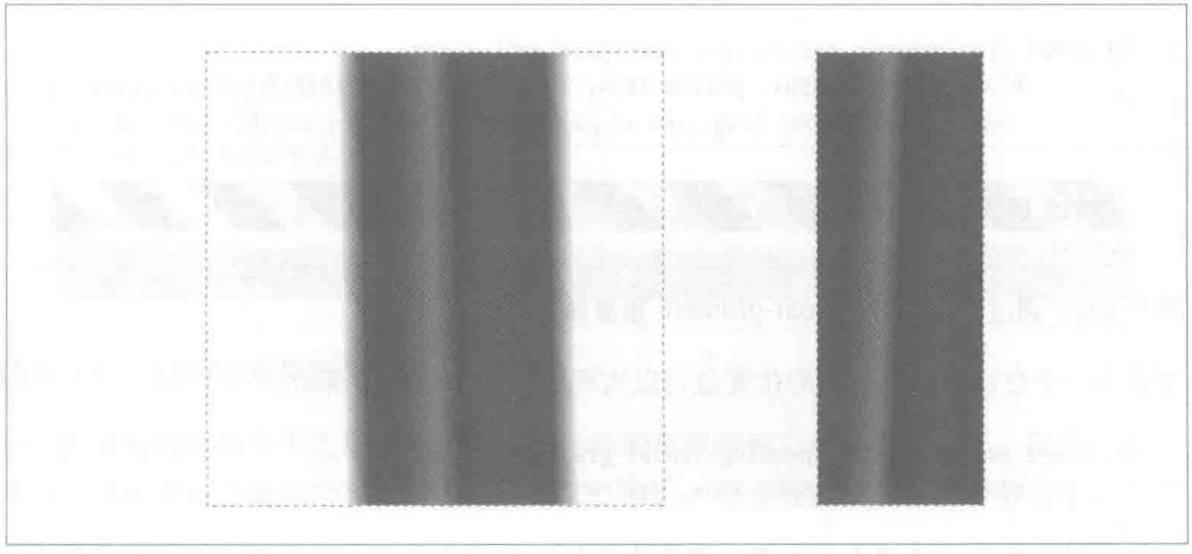


图 9-100：渐变实现的两个“褶皱”

这样实现的好处是，若想调整平铺间距，只需编辑样式表。调整色标的位置或颜色是有些繁琐，但是有了设计想法之后也不是多难。如果想再加一个褶皱，增加一个渐变即可。

9.3.4 循环渐变

渐变是很强大，但由于渐变是图像，平铺时可能会遇到一些奇怪的行为。例如，如果声明下述样式：

```
h1.exmpl {background:
    linear-gradient(-45deg, black 0, black 25px, yellow 25px, yellow 50px)
    top left/40px 40px repeat;}
```

得到的结果如图 9-101 所示。

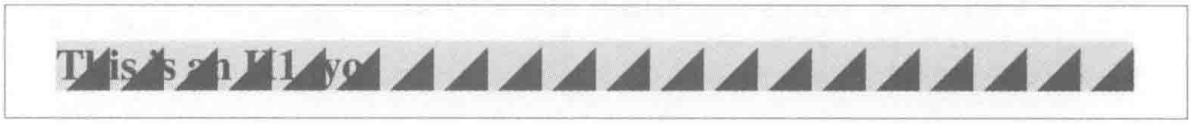


图 9-101：通过 background-repeat 属性平铺渐变图像

从图 9-101 中可以看出，重复的图像是不连续的。为了让相应的边对齐，你可以精确计算元素和渐变图像的尺寸，然后调整构建渐变的各要素，但是，如果有一种简单的方式能声明“始终无缝重复”，那该多好。

这时循环渐变就派上用场了。对上述示例来说，我们只需把 `linear-gradient` 换成 `repeating-linear-gradient`，并去掉 background-size 值，其他都保持不变。不过，这样得到的结果变化很大，如图 9-102 所示：

```
h1.exmpl {background: repeating-linear-gradient(-45deg,
        black 0, black 25px, yellow 25px, yellow 50px) top left;}
```

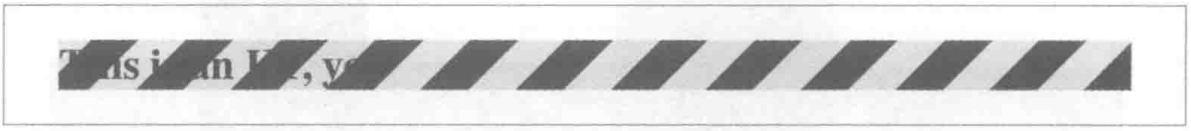


图 9-102：通过 repeating-linear-gradient 重复渐变图像

把其中一个色标换成中色点的位置也可以实现上述效果，如下所示：

```
h1.exmpl {background: repeating-linear-gradient(-45deg,
        black 0, black 25px, 25px, yellow 50px) top left;}
```

循环线性渐变在梯度线上不断循环排布声明的色标和中色点。对上例来说，每 25 像素将出现一次黑黄条纹。

注意，最后一个色标明确指定了长度（50px）。循环渐变必须这么做，最后一个色标的长度值用于定义图案的整体长度。

以上示例中的颜色是急停的。如果使用平滑过渡，一定要小心，最后一个色标的颜色值必须与第一个色标的颜色值一致。以下述渐变为例：

```
repeating-linear-gradient(-45deg, purple 0px, gold 50px)
```

这个渐变由紫色平滑过渡到 50 像素处的金色，然后立即变成紫色，再平滑过渡到下一个 50 像素处的金色。若想避免这种情况，要再添加一个色标，而且必须与第一个色标的颜色相同。下面的渐变与前述渐变的比较如图 9-103 所示：

```
repeating-linear-gradient(-45deg, purple 0px, gold 50px, purple 100px)
```

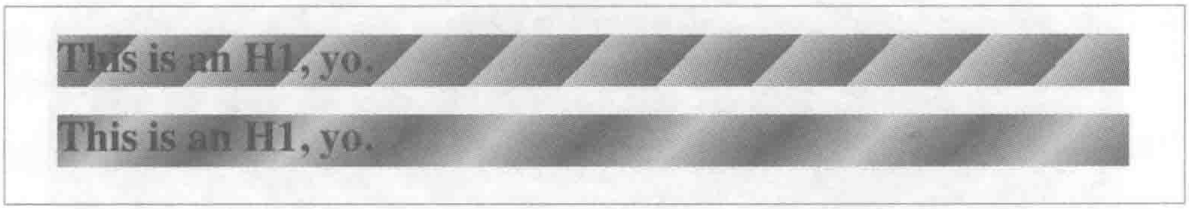


图 9-103：处理循环渐变图像中立即重置的颜色

你可能注意到了，目前看到的循环渐变都没有定义尺寸。因此，渐变图像将填满元素的整个背景定位区域，这是无固有高度和宽度的图像的默认行为。如果使用 background-size 属性设定了循环渐变图像的尺寸，渐变只在渐变图像范围内可见。如果再使用 background-repeat 属性设定重复，又会变成在背景中不间断重复的情况，如图 9-104 所示：

```
h1.exmpl {background:
    repeating-linear-gradient(-45deg, purple 0px, gold 50px, purple 100px)
    top left/50px 50px repeat;}
```

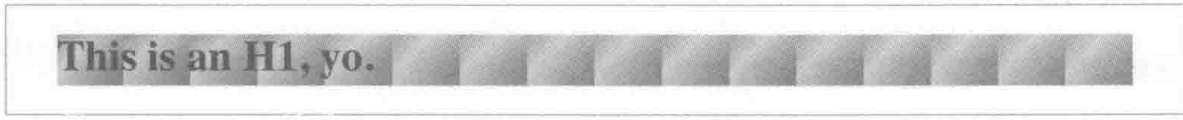


图 9-104：重复平铺循环渐变图像

如果在重复线性渐变中使用百分数值，各色标的位置就像没有重复时一样。因此，你将看到声明的色标创建的渐变，但是没有重复。所以，百分数值在重复线性渐变中没有任何意义。

不过，百分数值在循环径向渐变中却十分有用。定义好圆形或椭圆形的尺寸之后，再使用百分数定义色标在梯度射线上的位置，超出梯度射线末端后，便能看到循环渐变。以下述渐变为例：

```
.allhail {background:
    repeating-radial-gradient(100px 50px, purple, gold 20%, green 40%,
                              purple 60%, yellow 80%, purple);}
```

这里，各色标之间相距 20 像素，对应的颜色在得到的图案中不断循环。因为第一个色标和最后一个色标的颜色值相同，因此颜色不会立即变化。这个波纹将一直向外扩散，直到渐变图像的边缘，如图 9-105 所示。

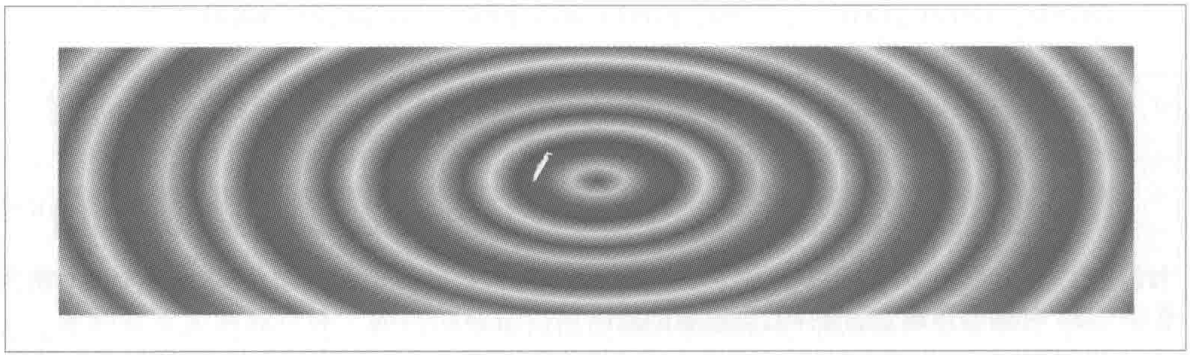


图 9-105：循环径向渐变

你可以想象一下循环的彩虹径向渐变是什么样子的：

```
.wdim {background:
    repeating-radial-gradient(
        100px circle at bottom center,
        rgb(83%,83%,83%) 50%,
        violet 55%, indigo 60%, blue 65%, green 70%,
        yellow 75%, orange 80%, red 85%,
        rgb(47%,60%,73%) 90%
    );}
```

创建循环径向渐变时要注意几点：

- 如果不为径向渐变声明尺寸，默认为椭圆形，而且宽高比与渐变图像相同。如果不使用 background-size 属性为渐变图像声明尺寸，默认为目标元素背景区域的尺寸（若是生成列表的项目符号，默认尺寸为浏览器预设的值）。
- 默认的径向渐变尺寸为 farthest-corner，即把梯度射线的终点放在右侧足够远的地方，让椭圆与径向图像上离径向渐变中心点最远的那一角相交。

这里再次重申是为了提醒你，如果使用默认值的话，得不到循环渐变，只能看到循环第一次迭代的结果。唯有限制渐变的初始尺寸，循环效果才可见。



径向渐变，尤其是循环径向渐变，对较旧的移动浏览器的性能消耗极大。浏览器崩溃是常见的事，而且页面渲染时间和电量也深受影响。在移动设备中使用径向渐变之前一定要三思，而且必须做严格的性能和稳定性测试，尤其是要兼容较旧的设备时（旧设备中的浏览器也是旧的）。

渐变的中色

循环渐变可能出现一种极端情况：第一个色标和最后一个色标由于某种原因恰好位于同一位置。例如，假设你在输入时忘了按“5”，不小心得到如下声明：

```
repeating-radial-gradient(center, purple 0px, gold 0px)
```

第一个色标和最后一个色标之间的距离为零像素，但是声明的却是沿梯度射线无限循环的渐变。此时结果如何呢？

遇到这种情况时，浏览器会找出渐变的中色，填满整个渐变图像。对上述简单的示例来说，中色是紫色和金色各一半的混合体（差不多是 #C06C40 或 rgb(75%,42%,25%)）。因此，渐变图像是橙棕色的，看起来完全不是渐变。

浏览器把色标的位置四舍五入为零，或者第一个色标与最后一个色标之间的距离过小，与输出的分辨率相比，没必要做任何渲染时，也可能出现这种情况。例如，循环径向渐变使用百分数设定所有色标的位置，而且尺寸为 closest-side，但是不小心把位置放在某一角上，就可能导致这样的结果。



截至 2017 年年末，还没有浏览器能正确处理这样的情况。在十分有限的条件下可能会以正确的行为处理，但是多数时候，浏览器要么直接使用最后一个色标的颜色填充，要么用尽浑身解数，尝试使用子像素绘制循环图案。

9.4 盒子投影

前面的章节探讨过 text-shadow 属性，其作用是为非置换元素中的文本添加投影。有个类似的属性，用于为元素所在的框体创建投影，即 box-shadow。

box-shadow	
取值	none | [inset? && <*length*>{2,4} && <*color*>?]#
初始值	none
适用于	所有元素
计算值	<*length*> 值计算为绝对长度值；<*color*> 值按内部方式处理；此外为指定的值
继承性	否
动画性	是

在主要介绍背景和渐变的一章中讨论投影好像不太合适，但是笔者这么做是有缘由的，稍后告诉你。

下面举个简单的例子。这个盒子投影向元素框下方偏移 10 像素、向右偏移 10 像素，没有展开尺寸，也不模糊，颜色为半不透明的黑色。此外，身后的 body 元素有一个重复的背景。得到的结果如图 9-106 所示。

```
#box {background: silver; border: medium solid;
     box-shadow: 10px 10px rgba(0,0,0,0.5);}
```

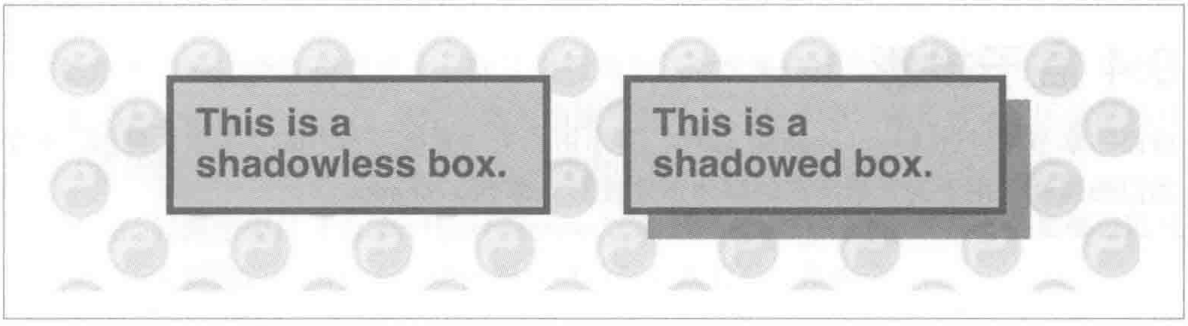


图 9-106：一个简单的盒子投影

可以看到，透过半不透明（如果不习惯，可以说成“半透明”）的投影能看到 body 元素的背景。因为没有设置模糊和展开距离，所以投影的外形完全模仿元素框本身。至少，看起来是这样。

之所以说投影看起来是模仿元素框的形状，是因为投影只在元素边框的外边界以外才看得到。这一点从上图中看不出来，因为元素的背景是不透明的。你可能以为投影会延伸到元素背后，但事实并非如此。以下述代码为例，其结果如图 9-107 所示。

```
#box {background: transparent; border: thin dashed;
     box-shadow: 10px 10px rgba(0,0,0,0.5);}
```

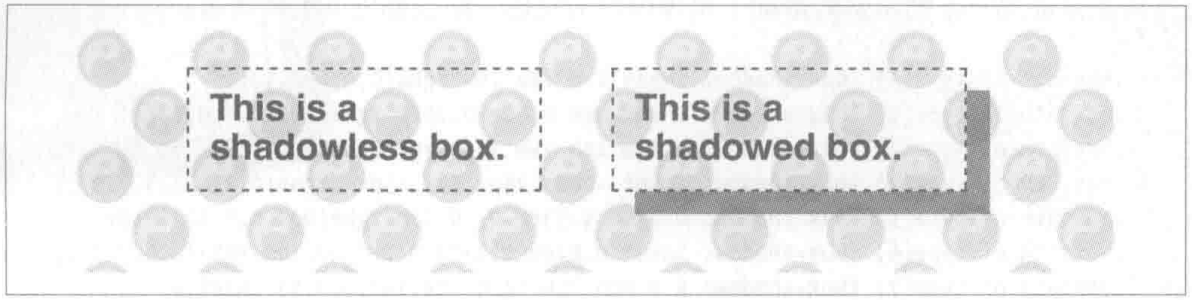


图 9-107：盒子投影不是一个完整的图形

看起来好像是元素的内容（及内边距和边框）区域“破坏”了部分投影。而事实是，根据规范，盒子投影就是这样绘制的，根本不会出现在那些区域。但是，如图 9-107 所示，带投影的元素框“背后”的背景却能透过元素看见。正是因为 box-shadow 属性与背景和边框有这一层关系（可能让人觉得匪夷所思），所以才在这里介绍，而没在之前的章节讨论。

前面定义的盒子投影设定了两个长度值。第一个定义横向偏移，第二个定义纵向偏移。正数把投影向下和向右移动，负数把投影向上和向左移动。

如果提供第三个值，定义的是模糊距离，即指定给模糊留出多少空间。第四个值定义展开距离，这会改变投影的尺寸。正值在模糊之前延伸投影，负值使投影变小。下述代码得到的结果如图 9-108 所示。

```
.box:nth-of-type(1) {box-shadow: 1em 1em 2px rgba(0,0,0,0.5);}
.box:nth-of-type(2) {box-shadow: 2em 0.5em 0.25em rgba(128,0,0,0.5);}
.box:nth-of-type(3) {box-shadow: 0.5em 2ch 1vw 13px rgba(0,128,0,0.5);}
.box:nth-of-type(4) {box-shadow: -10px 25px 5px -5px rgba(0,128,128,0.5);}
.box:nth-of-type(5) {box-shadow: 0.67em 1.33em 0 -0.1em rgba(0,0,0,0.5);}
.box:nth-of-type(6) {box-shadow: 0.67em 1.33em 0.2em -0.1em rgba(0,0,0,0.5);}
.box:nth-of-type(7) {box-shadow: 0 0 2ch 2ch rgba(128,128,0,0.5);}
```

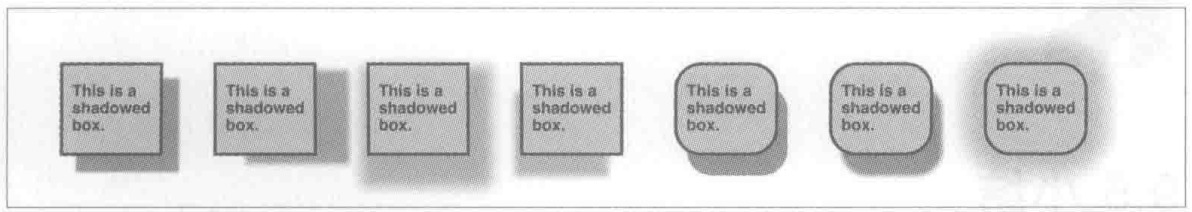


图 9-108：模糊距离和展开距离不同的投影

你可能注意到了，图 9-108 中有些框体带有圆角（通过 border-radius 属性实现），而相应的投影也变成了与之匹配的曲线。还好，这是默认行为。

box-shadow 还有一点没有涉及，即 inset 关键字。如果在 box-shadow 属性的值中加上 inset，投影在框体内部渲染，（从视觉上来讲）不再浮动在画布之上，而像是向内凹陷的。下面在前面那一系列示例的基础上实现内凹的投影，结果如图 9-109 所示。

```
.box:nth-of-type(1) {box-shadow: inset 1em 1em 2px rgba(0,0,0,0.5);}
.box:nth-of-type(2) {box-shadow: inset 2em 0.5em 0.25em rgba(128,0,0,0.5);}
.box:nth-of-type(3) {box-shadow: 0.5em 2ch 1vw 13px rgba(0,128,0,0.5) inset;}
.box:nth-of-type(4) {box-shadow: inset -10px 25px 5px -5px  rgba(0,128,128,0.5);}
.box:nth-of-type(5) {box-shadow: 0.67em 1.33em 0 -0.1em rgba(0,0,0,0.5) inset;}
.box:nth-of-type(6) {box-shadow: inset 0.67em 1.33em 0.2em -0.1em rgba(0,0,0,0.5);}
.box:nth-of-type(7) {box-shadow: 0 0 2ch 2ch rgba(128,128,0,0.5) inset;}
```

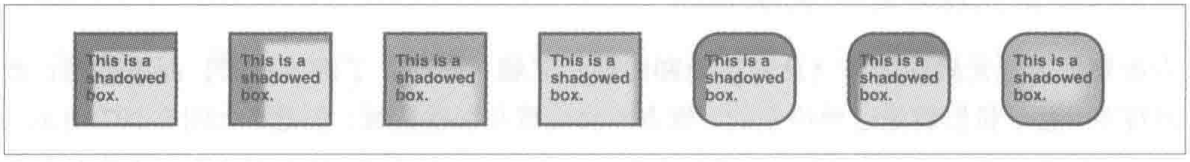


图 9-109：不同的内凹投影

注意，inset 关键字可以在其他值前面，也可以在后面，唯独不能在长度和颜色之间。0 0 0.1em inset gray 这样的值是无效的，将被忽略，因为 inset 关键字的位置不对。

最后，与文本投影一样，一个元素可以应用任意多个盒子投影，各投影之间以逗号分隔。而且，部分投影可以是内凹的，部分可以是外凸的。下述示例只是无限可能中的两例：

```
#shadowbox {background: #EEE;
     box-shadow: inset 1ch 1ch 0.25ch rgba(0,0,0,0.25),
          1.5ch 1.5ch 0.4ch rgba(0,0,0,0.33);}
#wacky {box-shadow: inset 10px 2vh 0.77em 1ch red,
     1cm 1in 0 -1px cyan inset,
     2ch 3ch 0.5ch hsla(117,100%,50%,0.343),
     -2ch -3ch 0.5ch hsla(297,100%,50%,0.23);}
```



filter 属性也能为元素创建投影，不过较之 box-shadow，其行为更接近 text-shadow，而且投影将同时应用到元素框和文本上。详情参见第 19 章。

9.5 小结

通过设置元素的颜色和背景，创作人员的能力得到了极大提升。与传统方式相比，CSS 的优势是可以把颜色和背景应用到文档中的任何元素上。

O'REILLY®

CSS权威指南

如果你是一名网页设计师或者应用开发者，对复杂的页面样式、改进可访问性以及节省时间和精力感兴趣，绝对不能错过这本书。本书全面阐述了CSS的实现方式，还深入分析了最新的CSS规范。

CSS是一门不断发展的语言，用于描述Web内容在屏幕、打印机、语音合成器、屏幕阅读器和聊天窗口上的表现。各种物联网设备，不论屏幕尺寸大小，只要有浏览器就要用到CSS，包括手机、计算机、视频游戏、电视、手表、自助服务终端和汽车控制台。本书两位作者展示了如何通过布局、过渡和动画、边框、背景、文本属性，以及许多其他工具和技术来改善用户体验、加快开发速度、避免潜在的错误，并为你的应用程序增光添色。

本书涵盖以下内容：

- **选择符、特指度和层叠。**
- **值、单位、字体和文本属性。**
- **内边距、边框、轮廓和外边距。**
- **颜色、背景和渐变。**
- **浮动和定位。**
- **弹性盒布局。**
- **全新的栅格布局系统。**
- **2D和3D变形、过渡和动画。**
- **滤镜、混合、裁剪和遮罩。**
- **媒体查询和特性查询。**

Eric A. Meyer是国际公认的HTML、CSS和Web标准领域的专家，他是Complex Spiral Consulting公司的创始人，还是微格式运动的发起人之一，以及设计领域系列会议An Event Apart的创始人之一。他是本书前一版和其他多本关于CSS和设计的图书的作者。他的个人网站为meyerweb.com。

Estelle Weyl是开放Web布道师和社区工程师，从1999年起就开始开发符合标准且具有可访问性的网站。她的博客地址是standardista.com，那里的文章讨论一切与前端有关的话题，包括CSS3、HTML5、JavaScrip性能、可访问性和移动Web开发。她喜欢躺在家中的沙发上编程，也经常游历世界，并在各种会议上演讲。

O'Reilly Media, Inc.授权中国电力出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-5198-2659-8

9 787519 826598 >

上下册：198.00元