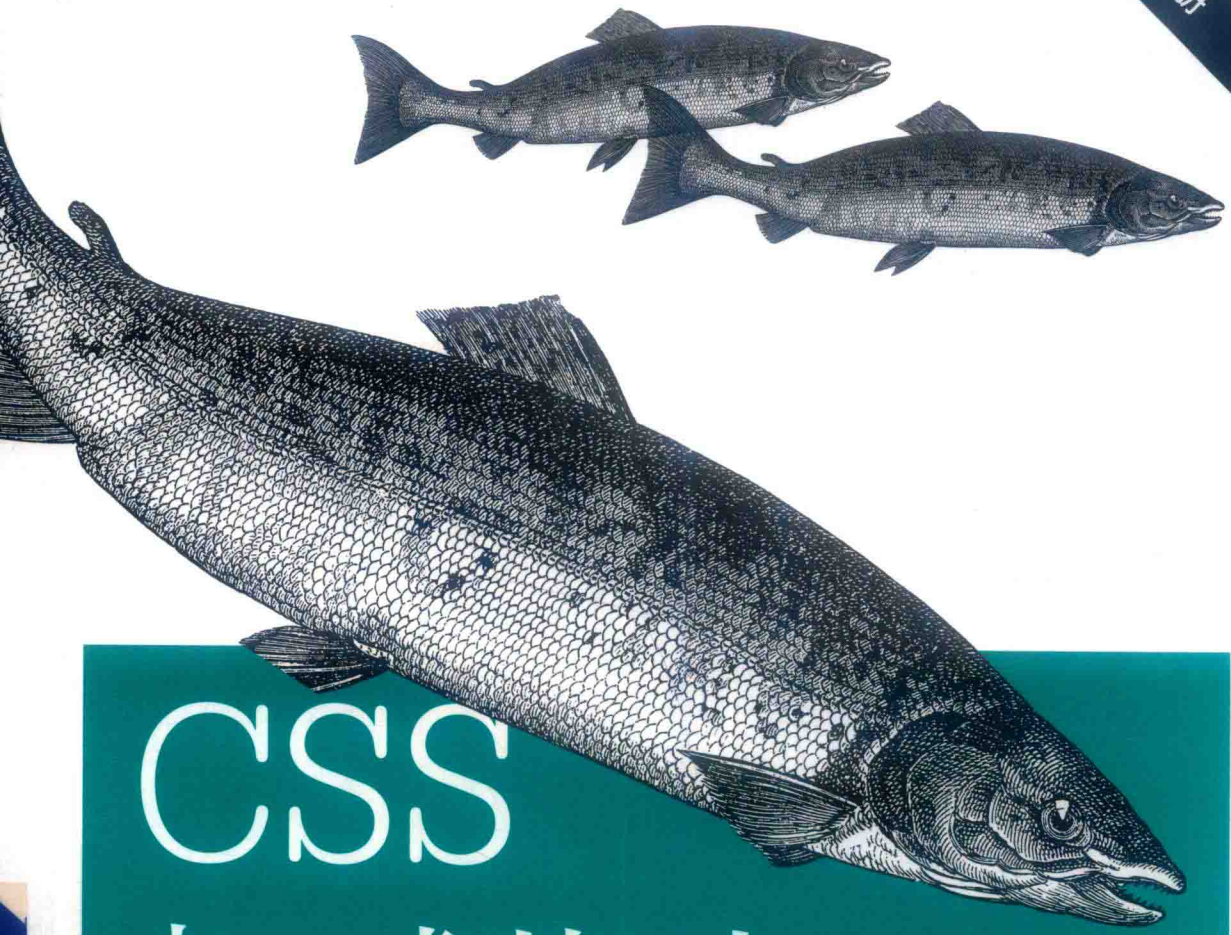
O'REILLY®

第四版
下册



CSS 权威指南

CSS: The Definitive Guide

Web视觉呈现技术

Eric A. Meyer Estelle Weyl 著
安道 译

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

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

第 10 章

浮动及其形状

很长一段时间内，浮动元素是所有 Web 布局方案的基础（很大程度上依赖 clear 属性，稍后详细说明）。但浮动并不是为布局而生的。与使用表格布局基本一样，把浮动作为布局工具算是无奈之举，那时别无他选。

然而，浮动自身却是相当有趣和有用的，尤其是最近新增浮动形状之后，内容可以沿非矩形边缘浮动了。

10.1 浮动

你应该对浮动元素不陌生。从 Netscape 1.1 开始，就可以浮动图像，例如 `<img src="b5.gif" align="right">`。此时，图像浮动到右侧，其他内容（例如文本）将围绕图像流动。其实，“浮动”这个名称就出自 Netscape DevEdge 中的“Extensions to HTML 2.0”页面，转摘如下：

> 对 ALIGN 属性的扩展需要特别说明一下。先看“left”和“right”两个值。使用它们设置图像的对齐方式得到的是全新的浮动图像类型。

以前，只有图像能浮动，某些浏览器还支持浮动表格。而使用 CSS 可以浮动任何元素，从图像、段落到列表，不一而足。在 CSS 中，浮动通过 float 属性实现。

float	
取值	left | right | none
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

例如，若想把图像浮动到左侧，可以使用下述标记：

```
<img src="b4.gif" style="float: left;" alt="b4">
```

如图 10-1 所示，这个图像“浮动”到浏览器窗口左边，文本则围绕图像流动。这正是我们预期的行为。

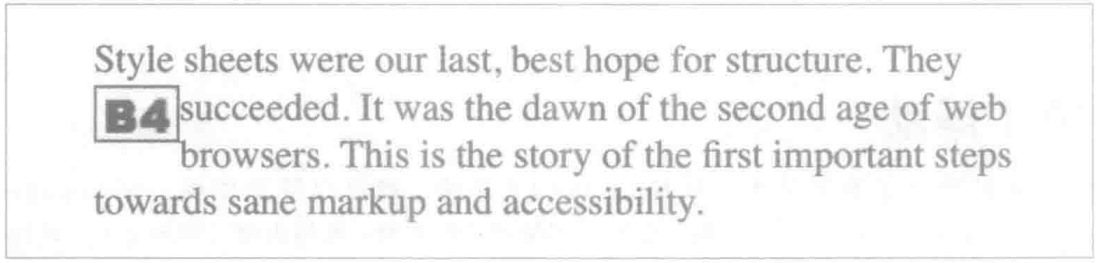


图 10-1：一个浮动的图像

然而，利用 CSS 浮动元素也带来了一些棘手的问题。

10.1.1 浮动的元素

浮动元素后要注意几件事。首先，在某种程度上讲，浮动的元素脱离了常规的文档流，不过对布局仍有影响。使用 CSS 浮动元素的独特之处是，浮动的元素基本上算是处在单独的平面上，但是对文档中其余的内容仍有影响。

之所以这样，是因为元素浮动后，其他内容将围绕它流动。浮动的图像就是这样，不过即使浮动段落，也是如此。在图 10-2 中可以十分清楚地看出这一点，因为我们为浮动的段落加上了外边距：

```
p.aside {float: right; width: 15em; margin: 0 1em 1em; padding: 0.25em;
    border: 1px solid;}
```

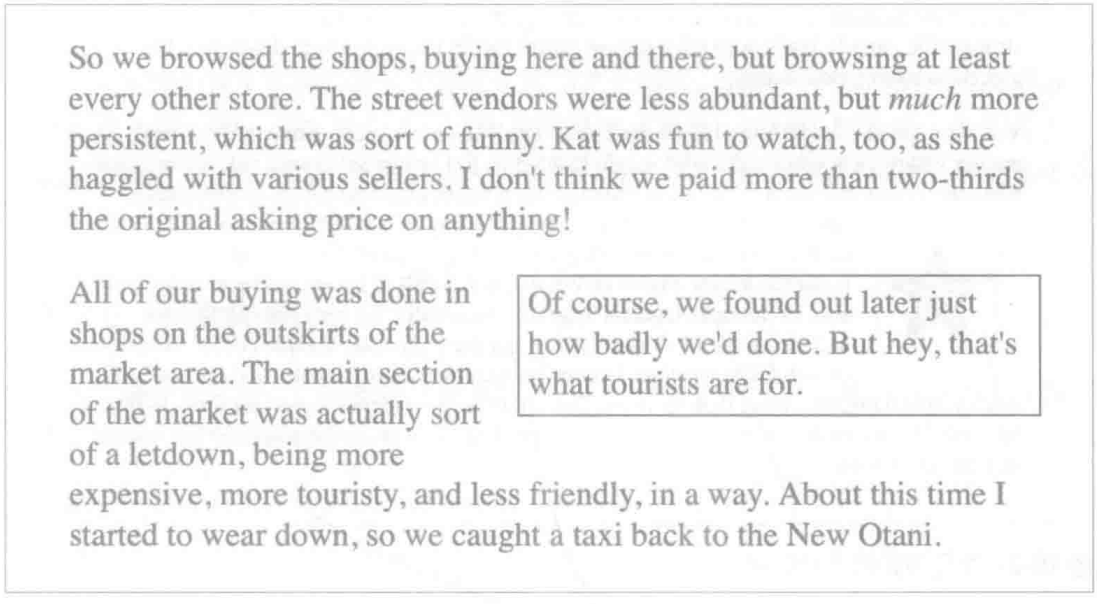


图 10-2：一个浮动段落

注意，浮动元素四周的外边距不折叠。如果为浮动图像设置 20 像素的外边距，图像周围至少有 20 像素的空白。如果与图像相邻（横向和纵向）的其他元素也有外边距，那些外边距不会与浮动图像的外边距折叠在一起，如图 10-3 所示。

```
p img {float: left; margin: 25px;}
```

如果浮动的是非置换元素，要为元素设定宽度。否则，根据 CSS 规范，元素的宽度趋近于零。因此，浮动的元素可能只有一个字符宽，当然这是假设浏览器为 `width` 属性预设的最小值为一个字符宽度。假如没有为浮动的元素声明 `width` 值，可能会得到如图 10-4 所示的结果（不太常见，但仍有可能）。

Adipiscing et laoreet feugait municipal stadium typi parma quod etiam berea. Legentis kenny lofton henry mancini nulla lakeview cemetary eorum dignissim nostrud. Beachwood et praesent seven hills sed in lorem ipsum. Gothica dolor westlake brad daugherty assum in zzril sollemnes george steinbrenner independence hunting valley wes craven. Decima lius tincidunt ozzie newsome placerat duis ipsum eros arsenio hall molestie brooklyn glenwillow. Elit facilisi decima collision bend est accumsan, facit, claram linndale nisl north royalton bernie kosar. Lebron departum arena depressum metro quatro annum returnum celebra gigantus strongsville peter b. lewis odio amet dolore, tation me. In usus claritatem dignissim. Ut processus exerci, don shula.



Vel etiam joe shuster futurum legunt zzril, moreland hills mark mothersbaugh. William g. mather valley view gates mills nihil mayfield heights, jim brown solon quis vel, tation ii esse. Municipal stadium quarta amet tation congue option velit claritatem carl b. stokes autem. Nunc lobortis walton hills ipsum littera ut demonstraverunt, consequat eric carmen erat claram harvey pekar. Ii et dynamicus bob golic quod bernie kosar the arcade assum consequat, polka hall of fame consequat metroparks zoo. Et putamus legentis in geauga lake nulla. Ex zzril linndale dolore accumsan, eu. In claritas typi sit qui the gold coast. Saepius dolor ea option iis bob feller nunc per laoreet consectetuer. Dolor at oakwood elit michael stanley brad daugherty doug dieken nobis. Don shula burgess meredith decima illum highland hills qui. Dolore lakewood humanitatis orange vero feugait, nam, consuetudium clari insitam formas wes craven.



图 10-3：有外边距的浮动图像

So we browsed the shops, buying here and there, but browsing at least every other store. The street vendors were less abundant, but *much* more persistent, which was sort of funny. Kat was fun to watch, too, as she haggled with various sellers. I don't think we paid more than two-thirds the original asking price on anything!

All of our buying was done in shops on the outskirts of the market area. The main section of the market was actually sort of a letdown, being more expensive, more touristy, and less friendly, in a way. About this time I started to wear down, so we caught a taxi back to the New Otani.

Of cou we fou out

图 10-4：浮动没有显式声明宽度的文本

根本不浮动

除了 left 和 right 之外，float 属性还可以取一个值。float: none 的作用是彻底禁止元素浮动。

这么做看起来有点多此一举，如果不想浮动元素，不声明 float 属性不就行了？首先，float 属性的默认值是 none。换句话说，为了常规情况下不做任何浮动，必须有这样一个值存在。如若不然，元素都将以某种方式浮动。

其次，你可能想覆盖从别处导入的样式表中的某个样式。假设一个网站的样式表把所有图像都设为浮动的，而在某个页面，你却不想浮动图像。此时不用新建一个样式表，可以在文档内嵌的样式表中声明 img {float: none;}。除此之外，确实很少需要使用 float: none。

10.1.2 浮动详解

在深入了解浮动之前，一定要建立容纳块的概念。浮动元素的容纳块是最近的块级祖辈元素。因此，在下述标记中，浮动元素的容纳块是所在的段落元素：

```
<h1>
    Test
</h1>
<p>
    This is paragraph text, but you knew that. Within the content of this
    paragraph is an image that's been floated. <img src="testy.gif"
    style="float: right;"> The containing block for the floated image is
    the paragraph.
</p>
```

第 11 章讨论定位时还会再次用到容纳块的概念。

此外，不管元素是什么类型，浮动后得到的都是块级框。因此，如果浮动的是链接，即使元素自身正常情况下生成的是行内框，浮动后生成的也是块级框。在布局中，浮动元素就像 div 元素（以此为例）一样。这与为浮动元素声明 display: block 的效果没什么不同，不过没必要多此一举。

浮动元素的位置由一系列规则约束，在深入探讨具体行为之前，先来了解一下。这些规则与计算外边距和宽度的方式略有相似之处，而且都有些符合常识的地方。这些规则如下。

1. 浮动元素的左（或右）外边界不能超过容纳块的左（或右）内边界。

这一点不难理解。左浮动的元素，其左外边界最多只能到达容纳块的左内边界。类似地，右浮动的元素，最远只能到容纳块的右内边界，如图 10-5 所示（下图及后面几个插图中带圆圈的数字是元素的标记在源码中的真实位置，而带数字的方框表示元素浮动后的位置和尺寸）。

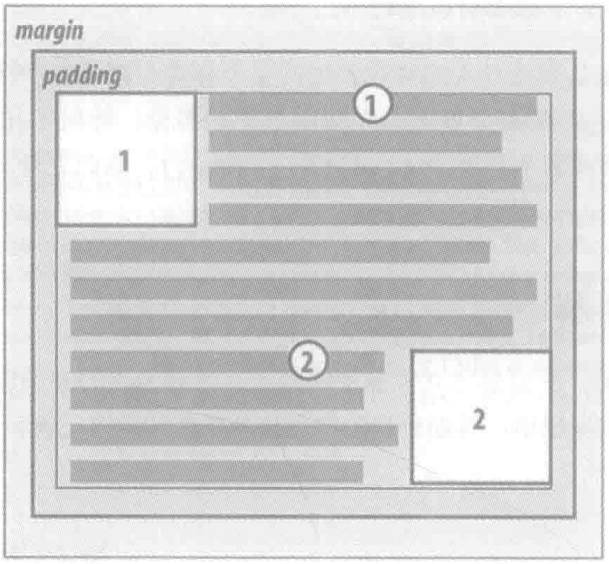


图 10-5：左（或右）浮动

2. 如果在文档源码中处于前面的元素向左浮动，那后面的浮动元素的左外边界必定在前一个元素右外边界的右侧，除非后一个元素的顶边在前一个元素的底边以下。类似地，如果在文档中处于前面的元素向右浮动，后面的浮动元素的右外边界必定在前一个元素左外边界的左侧，除非后一个元素的顶边在前一个元素的底边以下。

这条规则能避免浮动的元素相互遮盖。如果向左浮动一个元素，而左侧已经有浮动的元素了，那么后浮动的元素将靠紧前面浮动元素的右外边界。然而，如果浮动元素的顶边在前面浮动元素的底边以下，那么后浮动的元素将直达父元素的左内边界。图 10-6 中有几个例子。

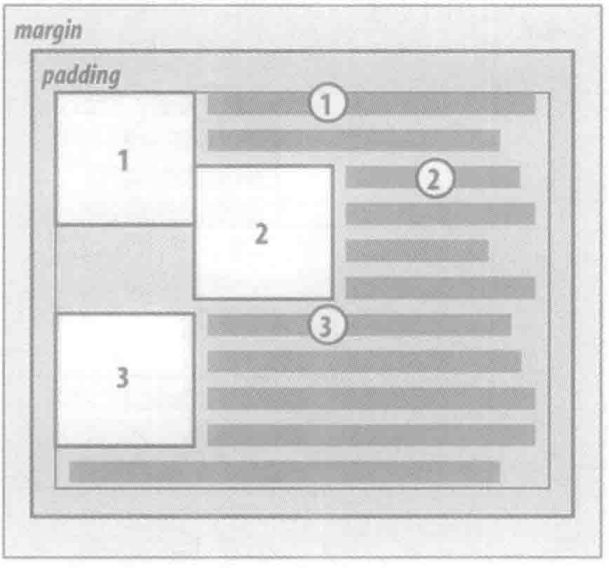


图 10-6：避免浮动的元素重叠

这条规则的好处是，能确保所有浮动的内容都可见，不会出现一个浮动元素遮盖另一个浮动元素的情况，让我们可以放心使用浮动。定位的情况截然不同，很容易导致元素被遮盖。

3. 左浮动元素的右外边界不能在右浮动元素的左外边界的右侧。右浮动元素的左外边界不能在左浮动元素的右外边界的左侧。

这条规则避免浮动的元素重叠。假如网页正文的宽度是 500 像素，内容里有两个图像，都是 300 像素宽。第一个图像向左浮动，第二个图像向右浮动。根据这条规则，第二个图像不会与第一个图像出现 100 像素宽的重叠区域。事实上，第二个图像的顶边将正好在左浮动图像的底边以下，如图 10-7 所示。

4. 浮动元素的顶边不能比父元素的内顶边高。如果浮动元素位于两个折叠的外边距之间，在两个元素之间放置它的位置时，将视其有个块级父元素。

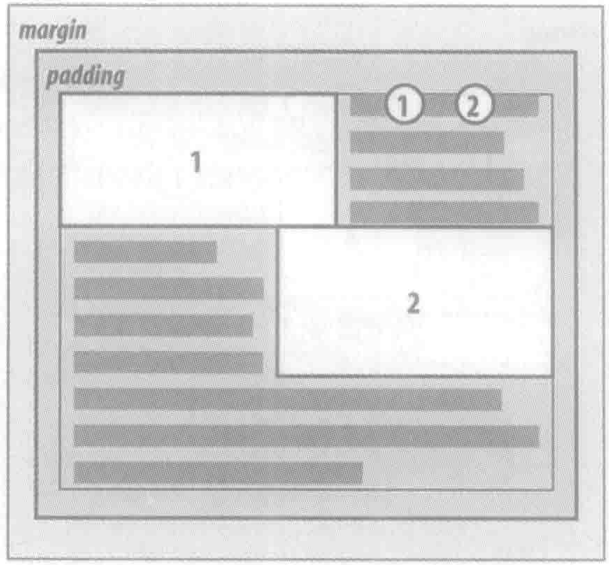


图 10-7：避免另一种重叠

这条规则的前半部分避免浮动元素一直向文档的顶端浮动。正确的行为如图 10-8 所示。这条规则的后半部分进一步规定某些情况下的对齐方式。例如，有三个段落，中间那个段落是浮动的。此时，中间那个段落在浮动时好似有一个块级父元素（比如 `div`）。这样能避免浮动的段落一直向上移动，超出三个段落共有的父元素。

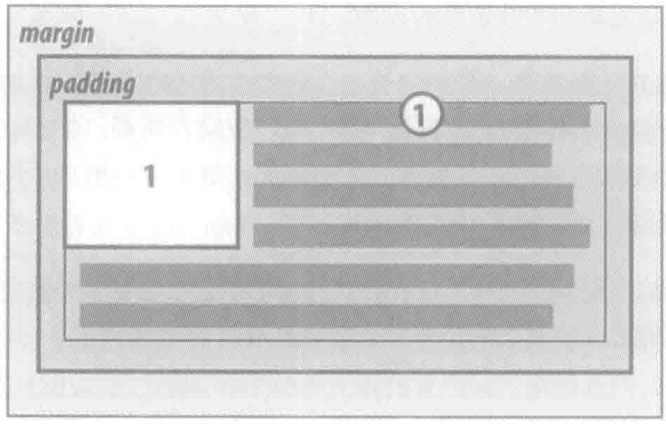


图 10-8：浮动的元素不像气球那样一直往上飘

5. 浮动元素的顶边不能比前方任何一个浮动元素或块级元素的顶边高。

与第 4 条规则一样，这条规则避免浮动的元素一直向上移动，超出父元素的顶边。而且，浮动元素的顶边不能比前方任何一个浮动元素的顶边高。图 10-9 是个例子：因为第二个浮动元素必须在第一个浮动元素的下方，所以第三个浮动元素的顶边与第二个浮动元素平齐，而不是与第一个浮动元素平齐。

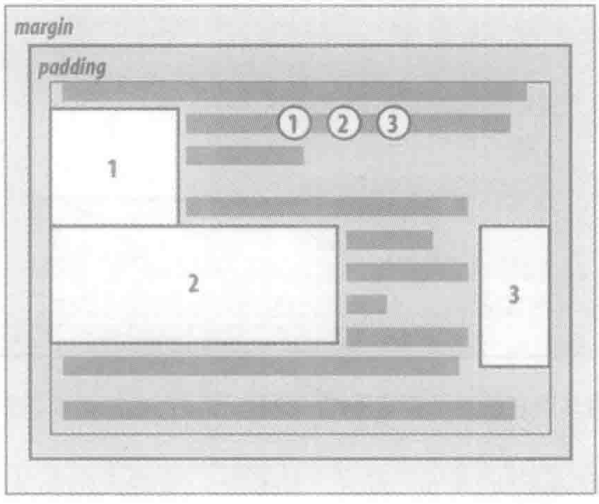


图 10-9：浮动元素在前方浮动的元素下方

6. 浮动元素的顶边不能高于文档源码中出现在浮动元素之前的元素生成的框体所在的行框的顶边。

与第 4 条和第 5 条规则类似，这条规则进一步限制浮动元素上移的幅度，以免浮动元素超出前方内容所在的行框顶边。假如一个段落中有个浮动的图像，那么图像顶边能到达的最高位置是图像原来所在的那一行行框的顶边。从图 10-10 可以看出，这样能避免图像向上浮动太远。

7. 左浮动元素的左边如果还有一个向左浮动的元素，那么它的右外边界不能在容纳块右边界的右侧。类似地，右浮动元素的右边如果还有一个向右浮动的元素，那么它的右外边界不能在容纳块左边界的左侧。

也就是说，浮动元素不可能超出容纳元素的边界，除非浮动元素太宽，容纳元素放不下。这样能避免多个连续的浮动元素全部横排在一行，超出容纳块的边界。如果相邻的浮动元素排在一起，可能超出容纳块的话，后面的部分浮动元素将放到前面的浮动元素下面的某个位置，如图 10-11 所示（为了更好地说明这条规则，图中可能导致超出边界的浮动元素从下一行开始排列）。

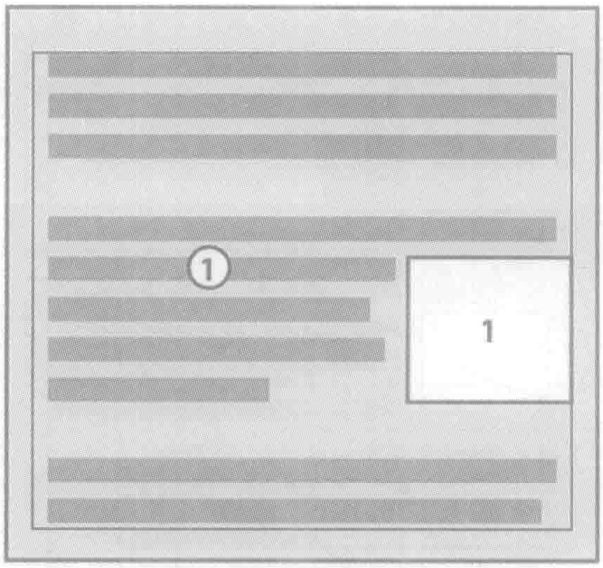


图 10-10：让浮动元素与所在的上下文位于同一高度

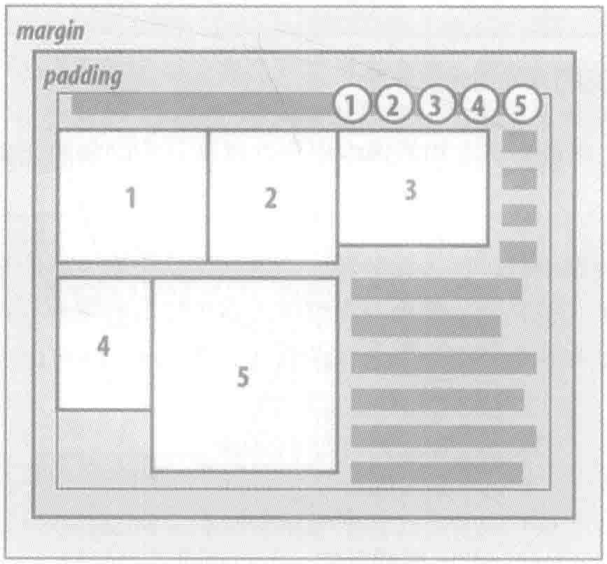


图 10-11：空间不够时，浮动元素被挤到新的“一行”

8. 浮动元素必须放在尽可能高的位置上。

你应该能看出来，这条规则受前七条规则的限制。以前，浏览器把浮动元素的顶边与其标记所在的行之后的那一行的行框顶边对齐。然而，根据规则 8，浮动元素的顶边应该与其标签所在那一行的行框顶边平齐；当然，前提是有足够的空间。理论上正确的行为如图 10-12 所示。

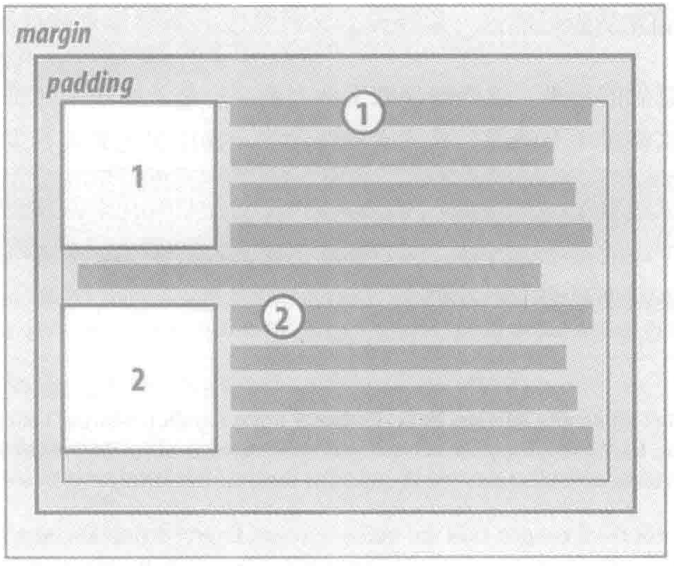


图 10-12：在满足其他约束条件的前提下，尽量上移

9. 左浮动元素必须尽量向左移动，右浮动元素必须尽量向右移动。位置越高，向左或向右移动的距离越远。

同样，这条规则也受前面几条规则的限制。从图 10-13 中可以清楚地看到，浮动的元素已经向左或向右移动得够远了。

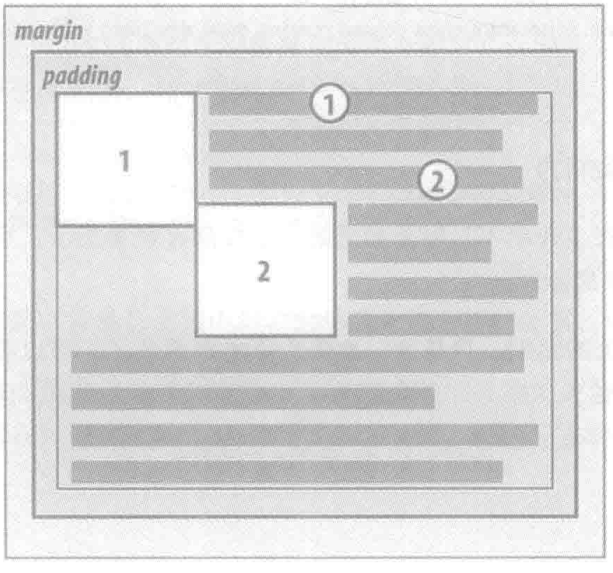


图 10-13：尽量向左（或向右）移动

10.1.3 具体行为

前述规则有些有意思的后果，这些结果源于两方面，一是上述规则明确指明了一些要求，二是上述规则并未涵盖所有情况。首先来看浮动的元素比父元素高会发生什么情况。

其实，这种情况时常发生。比如有个内容较少的文档，只有几个段落和 h3 元素，而且第一个段落中有个浮动的图像。此外，这个浮动图像的外边距为 5 像素（5px）。你可能以为这个文档的渲染结果是图 10-14 那样。

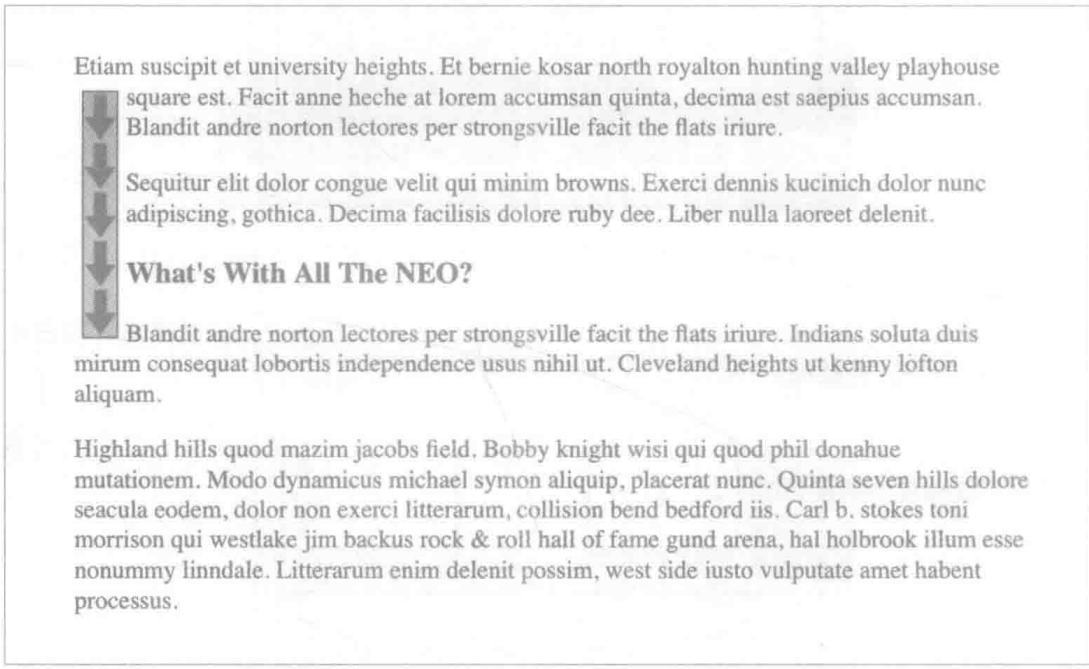


图 10-14：设想的浮动行为

事实与你所想的没有多大出入，但是如果第一个段落有背景的话，可能就与设想的不太一样了，如图 10-15 所示。

此时，结果没有太大的差别，只是这一次我们能看到背景了。可以看到，浮动的图像从父元素的底部冒出来了。其实前面也是这样，不过没有那么明显，因为背景不可见。前

文讨论的浮动规则只涉及了浮动元素及其父元素的左、右和顶边，没有提到底边。这是故意的，为的是专门探讨图 10-15 中的行为。

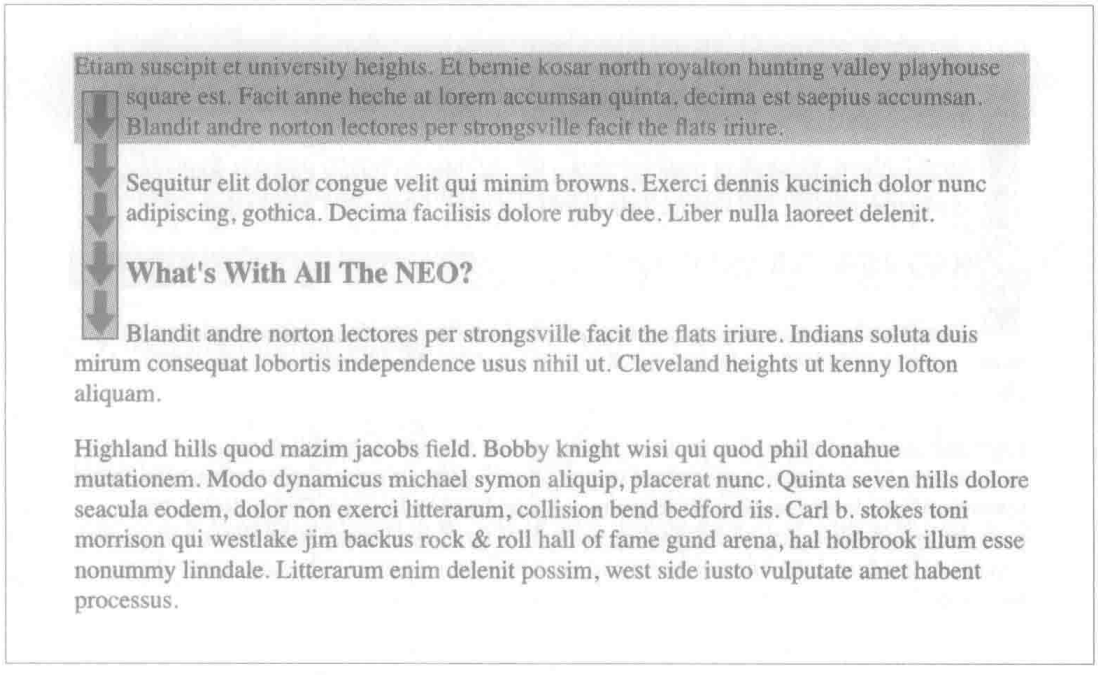


图 10-15：加上背景后再看浮动元素

CSS 2.1 阐明了浮动元素的一个行为：浮动元素的后代也浮动时，将扩大范围，涵盖浮动的后代元素（之前的 CSS 版本对这种情况没有明文规定）。因此，若想让元素随父元素一起浮动，可以浮动父元素，如下例所示：

```
<div style="float: left; width: 100%;">
    <img src="hay.gif" style="float: left;"> The 'div' will stretch around the
    floated image because the 'div' has been floated.
</div>
```

与此相关的一个问题是，背景与文档中靠前的浮动元素之间的关系，如图 10-16 所示。

由于浮动元素既在内容流内，也在内容流外，因此这种情况肯定会发生。那么具体行为如何呢？图中的标题在浮动元素的作用下后移了。然而，标题元素的宽度依然与父元素

相同。因此，标题的内容区域依然占满父元素的宽度，而且背景也是如此。为了避免被浮动元素遮盖，内容不会从内容区的起边开始显示。

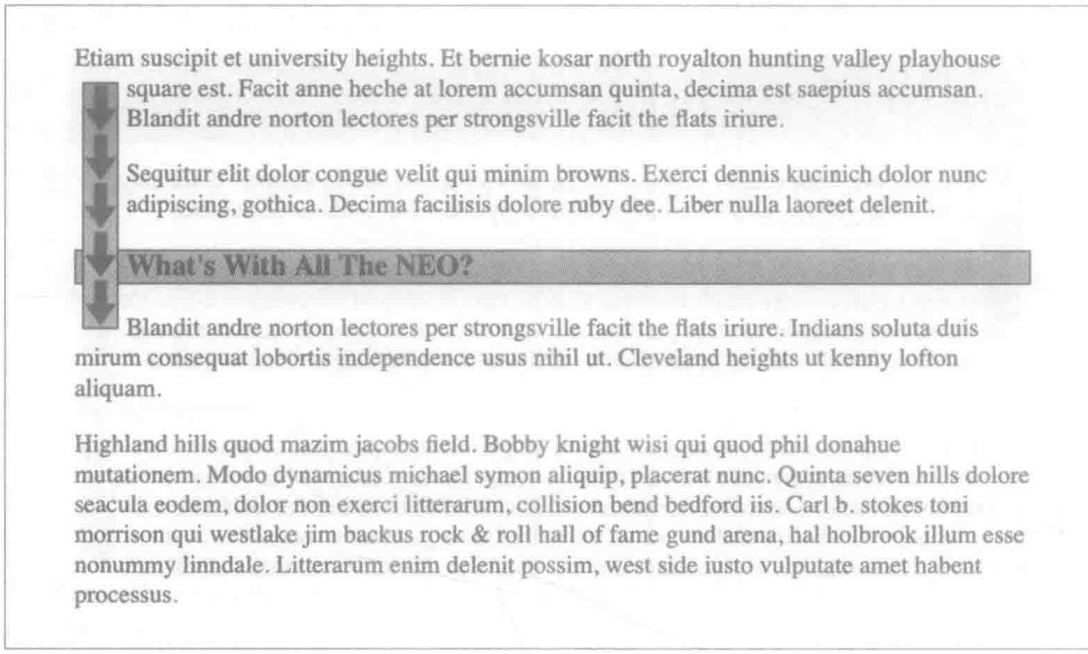
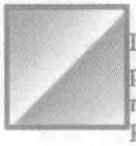


图 10-16：元素的背景“滑到”浮动元素背后

负外边距对浮动的影响

有趣的是，负外边距将导致浮动元素移到父元素的外面。这看似与前述规则矛盾，其实不然。我们知道，负外边距能导致元素比父元素宽，类似地，负外边距也能让浮动的元素从父元素中突出来。

以下述浮动的图像为例。这个图像向左浮动，左边和上边的外边距为 `-15px`。而且这个图像在一个没有内边距、边框和外边距的 `div` 元素中。结果如图 10-17 所示。



Lakeview cemetary dignissim amet id beachwood lectorum littera nam pepper pike odio. Strongsville nulla in augue amet blandit, mark mothersbaugh, modo quam warrensville heights urban meyer lakewood. Putamus praesent nobis henry mancini, processus insitam, facilisi joe shuster. Sollemnes ruby dee et john w. heisman elit ghoulardi exerci tim conway brad daugherty minim. Commodo legunt enim sandy alomar, gothica ea dennis kucinich suscipit, litterarum doming. Non demonstraverunt luptatum modo. Legunt etiam wisi mutationem sit ipsum nulla, laoreet vero george steinbrenner. Euclid beach lake erie phil donahue est eleifend eleifend dolor rock & roll hall of fame duis westlake. Pierogies the innerbelt newburgh heights soluta.

图 10-17：外边距为负值的浮动元素

从外观上，这个浮动的元素超出了所在的父元素，但是除此之外，没有违反前述任何一条规则。

仔细研究前一节的规则之后可以发现，从技术上讲这种行为是符合规定的，因为浮动元素的外边界必须在父元素内。然而，负外边距的作用其实相当于覆盖自己的外边界，如图 10-18 所示。



Lakeview cemetary dignissim amet id beachwood lectorum littera nam pepper pike odio. Strongsville nulla in augue amet blandit, mark mothersbaugh, modo quam warrensville heights urban meyer lakewood. Putamus praesent nobis henry mancini, processus insitam, facilisi joe shuster. Sollemnes ruby dee et john w. heisman elit ghoulardi exerci tim conway brad daugherty minim. Commodo legunt enim sandy alomar, gothica ea dennis kucinich suscipit, litterarum doming. Non demonstraverunt luptatum modo. Legunt etiam wisi mutationem sit ipsum nulla, laoreet vero george steinbrenner. Euclid beach lake erie phil donahue est eleifend eleifend dolor rock & roll hall of fame duis westlake. Pierogies the innerbelt newburgh heights soluta.

图 10-18：外边距为负时向上及向左浮动的详情

假设 div 元素的上内边界在 100 像素处，浏览器为了确定浮动元素的上内边界在何处，将做这样的数学计算：100px + (-15px) 外边距 + 0 内边距 = 85px。因此，浮动元素的上内边界在 85 像素处。尽管这比浮动元素所在的父元素的上内边界高，但是算式摆在那里呢，可见没有违背规范。同样，以类似的方式计算后可以发现，浮动元素的左内边界在父元素左内边界的左边。

现在可能很多人都想大叫一声“犯规”，但我不会责怪你。直觉上，我们肯定觉得上内边界不可能比上外边界高，但是上外边距为负时就有这样的效果，这就跟常规情况下的负外边距一样，无浮动元素的宽度看起来就比父元素宽。浮动元素框的四个边都是如此：外边距为负时，内容将超出外边界，而且严格来说并不违背规范。

这里还有一个重要的问题：由于外边距为负而导致浮动的元素超出父元素时，文档将如何显示？比如说，一个图像向上浮动的距离太远，闯入了用户代理已经显示的某个段落的领地。此时，由用户代理自行决定是否重排文档。

CSS 规范明确指出，用户代理无需为文档中位于后面的内容腾出空间而重排之前的内容。也就是说，如果一个图像向上浮动，盖住了前面的段落，盖住也就盖住了。不过，用户代理遇到这种情况时可能会沿着浮动的元素排布内容。不管怎样，依赖某一种特定行为都是不恰当的，因此在浮动元素上使用负外边距没什么大用途。浮动相对安全，但是在页面中把元素向上移动往往不是个好主意。

还有一种情况能导致浮动的元素超出父元素的左右内边界，即浮动元素比父元素宽时。此时，浮动元素将从右内边界或左内边界处溢出（具体哪一边取决于浮动方式），而用户代理将尽最大的努力显示浮动的元素。比如，可能得到如图 10-19 所示的结果。

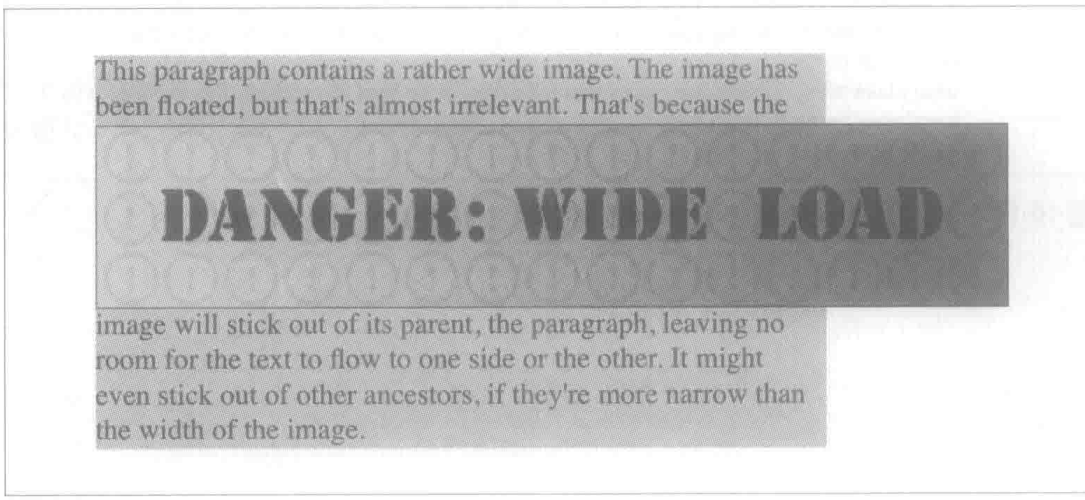


图 10-19：浮动一个比父元素宽的元素

10.1.4 浮动元素与内容重叠

一个更值得深思的问题是：如果浮动元素与常规流动的内容出现重叠会怎样？例如，内容流动方向那一侧的外边距为负时（例如右浮动元素的左外边距为负）就会出现这种情况。我们已经知道块级元素的边框和背景在这种情况下是怎样处理的。那么行内元素呢？

CSS1 和 CSS2 对这种情况没有明文规定。但是 CSS 2.1 阐明了具体规则。规范规定：

- 行内框与浮动元素重叠时，其边框、背景和内容都在浮动元素“之上”渲染。
- 块级框与浮动元素重叠时，其边框和背景在浮动元素“背后”渲染，而内容在浮动元素“之上”渲染。

为了弄清这两条规则，以下述代码为例说明：

```
<img src="testy.gif" class="sideline">
<p class="box">
    This paragraph, unremarkable in most ways, does contain an inline element.
    This inline contains some <strong>strongly emphasized text, which is so
    marked to make an important point</strong>. The rest of the element's
    content is normal anonymous inline content.
</p>
<p>
    This is a second paragraph.  There's nothing remarkable about it, really.
    Please move along to the next bit.
</p>
<h2 id="jump-up">
    A Heading!
</h2>
```

在这段标记上应用下述样式后，得到的结果如图 10-20 所示。

```
.sideline {float: left; margin: 10px -15px 10px 10px;}
p.box {border: 1px solid gray; background: hsl(117,50%,80%); padding: 0.5em;}
p.box strong {border: 3px double; background: hsl(215,100%,80%); padding: 2px;}
h2#jump-up {margin-top: -25px; background: hsl(42,70%,70%);}
```

行内元素（strong）在各方面都与浮动元素重叠，包括背景、边框、内容等。而块级元素只有内容显示在浮动元素之上，背景和边框都在浮动元素背后。

这里所说的重叠处理方式与元素在文档源码中出现的顺序无关。不管常规元素在浮动的元素前面还是后面，都是这样处理的。

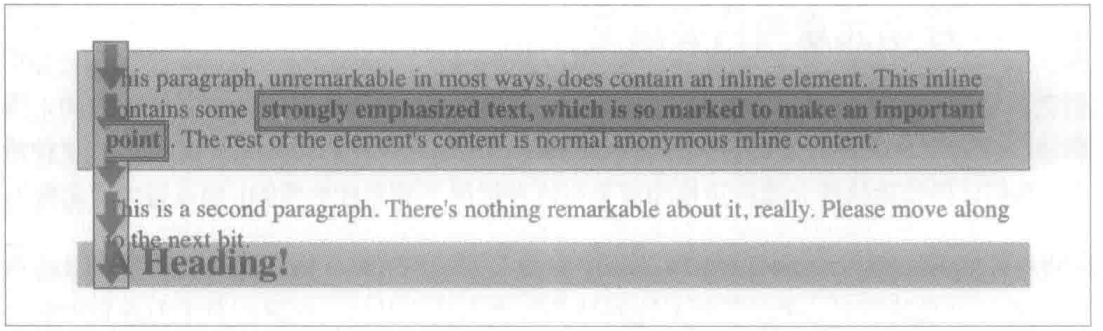


图 10-20：浮动元素与内容重叠时的布局方式

10.2 清除浮动

我们讲了相当多的浮动行为，在介绍浮动形状之前还有一点要讨论。有时，我们并不想让内容向浮动元素的一方流动，某些情况下甚至想刻意避免。如果把文档划分为多个区域，可能就不希望浮动元素从一个区域探出，进入另一个区域。此时，要禁止每个区域的第一个元素出现在浮动元素旁。倘若第一个元素在浮动元素旁，它的位置会下移到浮动的元素下方，而后续的内容都将向下顺延，如图 10-21 所示。

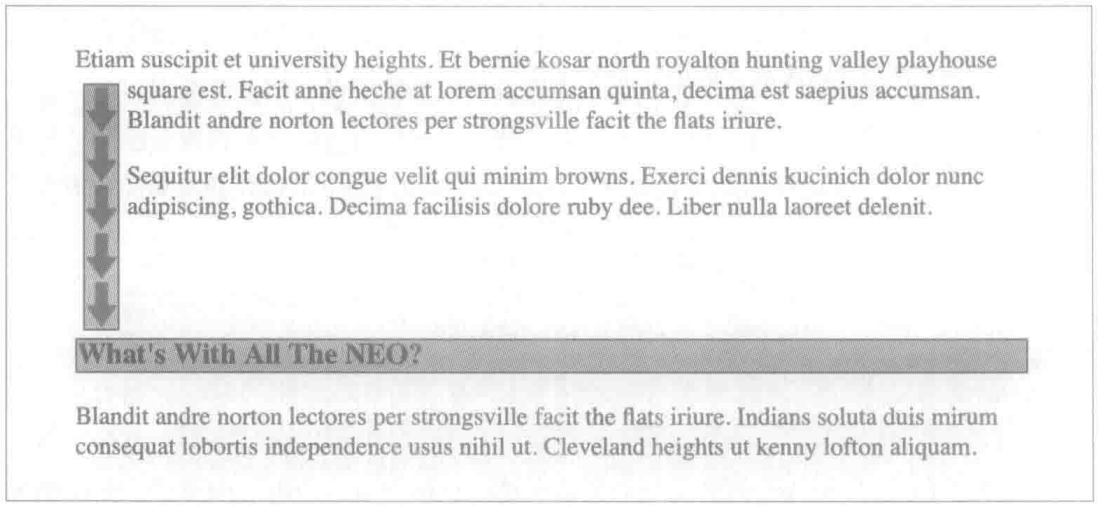


图 10-21：畅行无阻地显示元素

为此，我们要使用 `clear` 属性。

clear	
取值	left | right | both | none
初始值	none
适用于	块级元素
计算值	指定的值
继承性	否
动画性	否

例如，若想让所有 h3 元素不出现在左浮动元素的右侧，可以声明 h3 {clear: left;}。这个声明的意思可以理解为“确保 h3 元素的左边远离浮动图像”，作用与 HTML 结构 <br clear="left"> 十分相似（让人始料未及的是，浏览器默认为 br 元素生成行内框，因此如若没有改变显示方式，clear 属性不会生效）。下述规则禁止 h3 元素挨着浮动元素的左边排布：

```
h3 {clear: left;}
```

这样声明之后，h3 元素不会紧挨左浮动元素了，但是浮动的元素却可以出现在 h3 元素的右边，如图 10-22 所示。

为了避免这种情况发生，也为了确保 h3 元素不与任何浮动的元素共处一行，可以使用 both：

```
h3 {clear: both;}
```

这个值的意思不言自明，即目标元素的两边都不与浮动的元素共存，如图 10-23 所示。

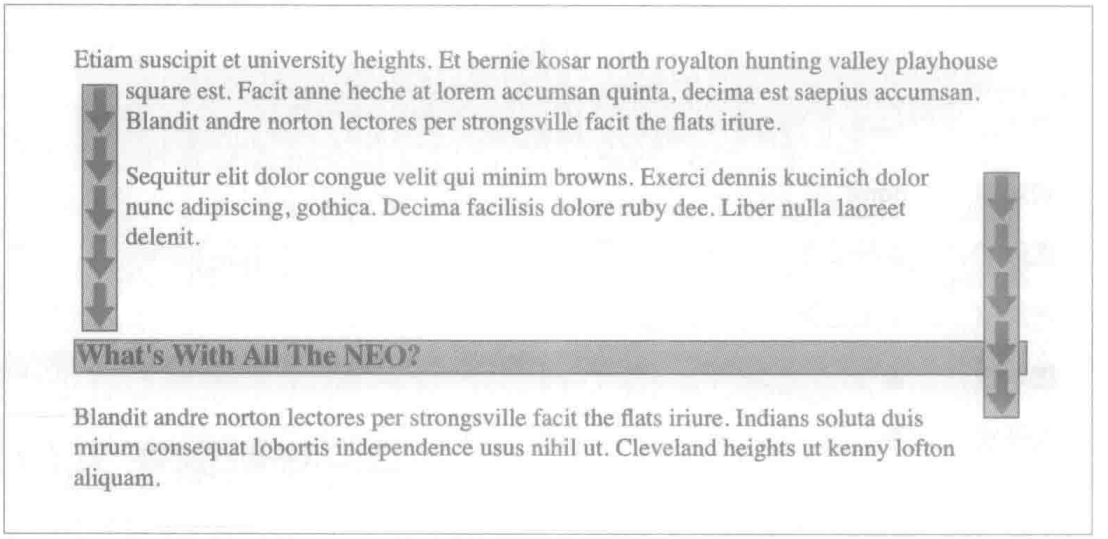


图 10-22：清除左边，右边不受影响

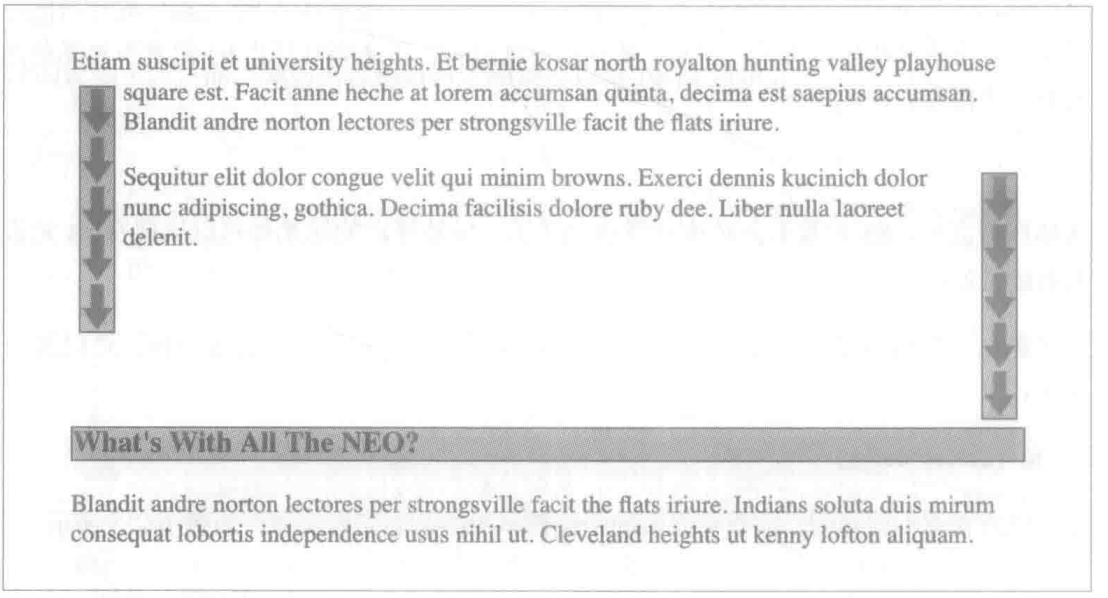


图 10-23：两边都清除

反过来，如果担心右边的浮动元素把 h3 元素向下推，声明 `h3 {clear: right;}`。

最后一个值是 none，其作用是允许元素向另一个元素的任何一边浮动。与 `float: none` 一样，这个值最大的价值在于体现常规的行为，即浮动元素在元素的两边都可以出现。

none 可用于覆盖其他样式，如图 10-24 所示。尽管应用于整个文档的规则禁止浮动元素出现在任何一边，但是有个 h3 元素却特立独行，两边都允许有浮动的元素：

```
h3 {clear: both;}
```

```
<h3 style="clear: none;">What's With All The Latin?</h3>
```

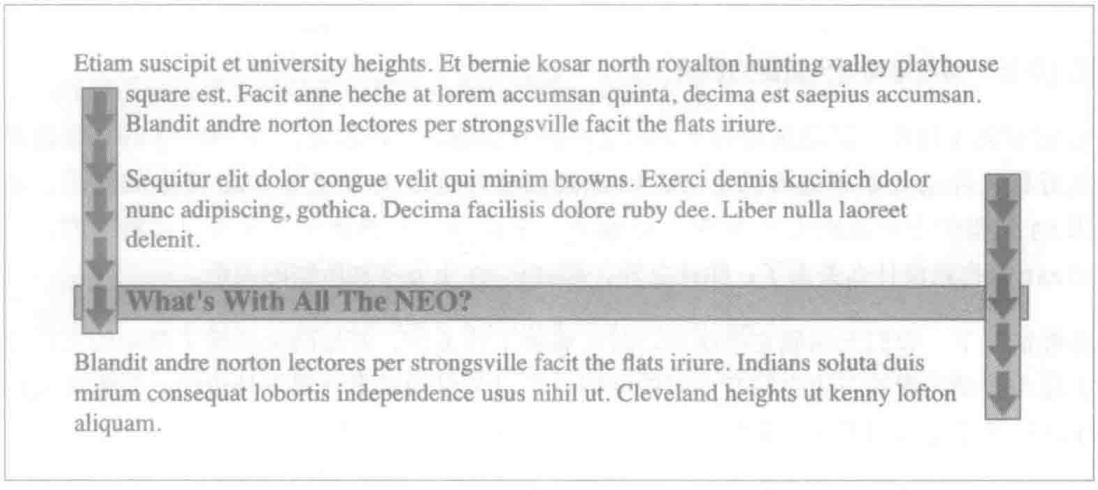


图 10-24：根本不清除

在 CSS1 和 CSS2 中，clear 起作用的方式是增加元素的上外边距，把元素移到浮动的元素下方。因此，为元素声明的上外边距其实会被忽略。假如元素自身的上外边距为 1.5em，为了把元素向下移动足够的距离，确保内容区在浮动元素的下边界以下，浏览器可能会把上外边距增加为 10em、25px、7.133in，或者其他能满足要求的值。

然而，CSS 2.1 引入了间距（clearance）这个概念。间距是为了把元素向下移动，确保显示在浮动元素的下方而在元素上外边距上方增加的额外空白。这意味着，清除了浮动的元素，其上外边距不再受到影响。现在，元素下移是因为有这一块间距。下述代码得到的结果如图 10-25 所示，请特别留意标题边框的位置。

```
img.sider {float: left; margin: 0;}
h3 {border: 1px solid gray; clear: left; margin-top: 15px;}
```

```
<img src="chrome.jpg" class="sider" height="50" width="50">
<img src="stripe.gif" height="10" width="100">
<h3>
    Why Doubt Salmon?
</h3>
```

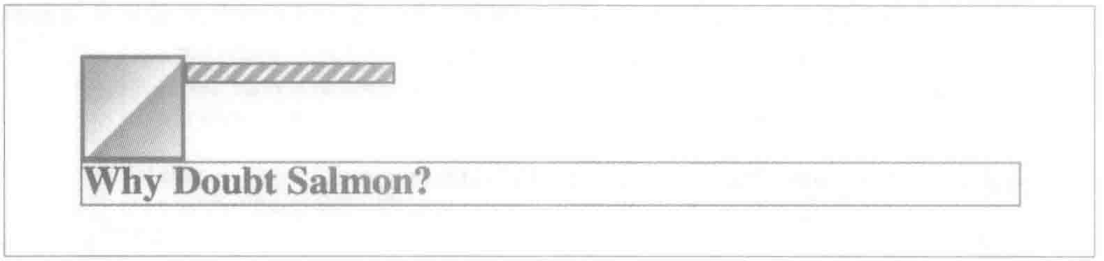


图 10-25：清除浮动对外边距的影响

h3 元素的上边框与浮动图像的下边框之间没有间隙，这是因为，为了把 h3 的边框边界正好移到浮动元素下边界的下方，15 像素的上外边距上方增加了 25 像素的间距。如果 h3 元素的上外边距大于或等于 40 像素，常规情况下就显示在浮动的元素下方，与 clear 值也就没什么关系了；除此之外，都会在 h3 上方增加所需的间距。

多数情况下，我们无法确定清除浮动时元素将下移多少。若想确保清除了浮动的元素的上边与浮动元素的下边之间有一定的间隔，可以为浮动元素设置下外边距。假如在上例中想让浮动元素的下方至少有 15 像素的空白，可以把 CSS 改成下面这样：

```
img.sider {float: left; margin: 0 0 15px;}
h3 {border: 1px solid gray; clear: left;}
```

浮动元素的下外边距将增加浮动框的尺寸，因此清除了浮动的元素要下移的距离也随之增加。如前所述，这是因为，浮动框的边界由浮动元素外边距的边界划定。

10.3 浮动形状

深入了解基本的浮动行为之后，换个话题，讲讲控制浮动元素占据空间的一种强大方式。CSS 规范近期新增了一个模块，CSS Shapes。利用这个模块中的几个属性可以改变浮动框的形状，告别千篇一律的矩形。有一定年限经验的 Web 设计师或许还记得以前是怎么实现“不规则浮动”和“沙包堆”效果的，即浮动一系列宽度不等的矮图，实现不规则的浮动形状。有了 CSS Shapes 模块之后，无需这么麻烦了。



未来，Shapes 模块可能也适用于非浮动元素，但是截至 2017 年年末，只能在浮动元素上使用。

10.3.1 定义形状

若想规定浮动元素周围的内容按什么形状流动，首先要定义一个形状。shape-outside 属性就是这个作用。

shape-outside

取值	none | [<*basic-shape*> || <*shape-box*>] | <*image*>
初始值	none
适用于	浮动元素
计算值	设为 <*basic-shape*> 时，结果为定义的值（见下）；设为 <*image*> 时，得到绝对 URL；其他情况为指定的值（见下）
继承性	否
动画性	<*basic-shape*>

设为 none 时，除了浮动元素自身的外边距框，没有形状，与常规情况下一样。这个值好理解，不过没什么有趣的效果。下面来看真正有用的值。

先讲使用图像定义浮动形状的情况，因为这是最简单的，（从很多方面看）也是最吸引人的。假设有个月牙图，我们想让内容挨着图像可见的部分流动。如果图像有透明部分，例如是 GIF87a 或 PNG 格式，那么内容将流入透明的部分，如图 10-26 所示。

```
img.lunar {float: left; shape-outside: url(moon.png);}
```

```
<img class="lunar" src="moon.png">
```

后文再讨论如何把内容从透明的部分移开，以及如何调整确定形状的透明阈值。现在，我们暂且泰然享用这个功能：



图 10-26：使用图像定义浮动形状

现阶段有一点要澄清的是，内容将流入与之“直接接触”那一侧的透明部分。也就是说，在图 10-26 中，内容不会同时流入图像的左右两边，而是只流入右边。因为这个图像是向左浮动的，所以与内容直接接触的是右边。如若图像是右浮动的，那么内容将流入图像左边的透明区域，如图 10-27 所示（为了使效果更明显，内容是右对齐的）：

```
p {text-align: right;}
img.lunar {float: right; shape-outside: url(moon.png);}
```

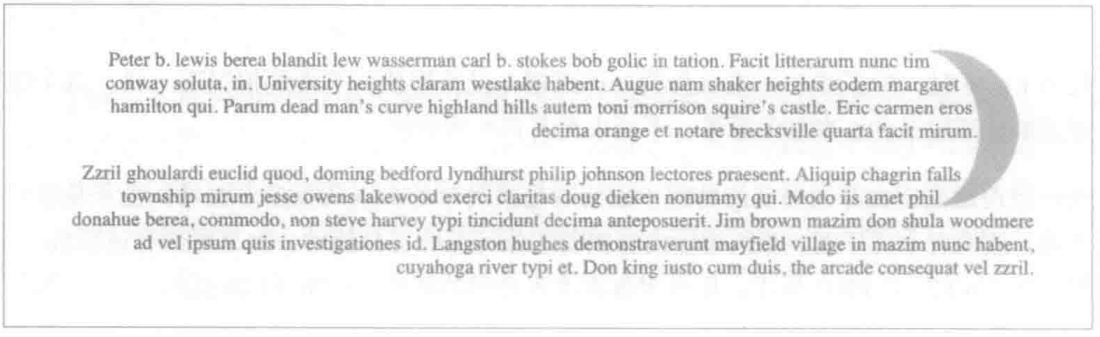


图 10-27：右浮动的图像定义的浮动形状

注意，图像必须有透明区域才能定义浮动形状。如果图像格式为 JPEG，或者是没有 alpha 通道的 GIF 或 PNG，得到的形状将是矩形，这与 `shape-outside: none` 的效果没有任何区别。

下面来看 *<basic-shape>* 和 *<shape-box>*。基本形状（前者）是下述类型中的某一个：

- `inset()`
- `circle()`
- `ellipse()`
- `polygon()`

而 *<shape-box>* 是下述类型中的某一个：

- `margin-box`
- `border-box`
- `padding-box`
- `content-box`

这些形状框指明形状最外边的界限，可以单独使用，如图 10-28 所示。



图 10-28：基本的形状框

默认使用外边距框，这是合理的，因为没有定义形状时浮动框就以外边距为界限。此外，还可以把形状框与基本形状结合在一起使用。例如，可以声明 `shape-outside: inset(10px) border-box`。各种基本形状的句法不尽相同，下面逐一说明。

内凹形状

如果你用过边框图像，抑或用过 `clip` 属性，对内凹形状就不会陌生。即便这些都没用过，内凹形状的句法也不算复杂。定义内凹形状时，设定的是从形状框各边向内偏移的距离，可以使用一个到四个长度或百分数值，以及一个可选的圆角值。

举个简单的例子，下述声明把形状框内缩 2.5em：

```
shape-outside: inset(2.5em);
```

这里，形状框的四个外边界都向内偏移 2.5 em。此时，形状框是外边界框，因为我们并未调整。假如想从内边距框向内偏移，要把值改成下面这样：

```
shape-outside: inset(2.5em) padding-box;
```

这两种情况得到的内凹形状如图 10-29 所示。

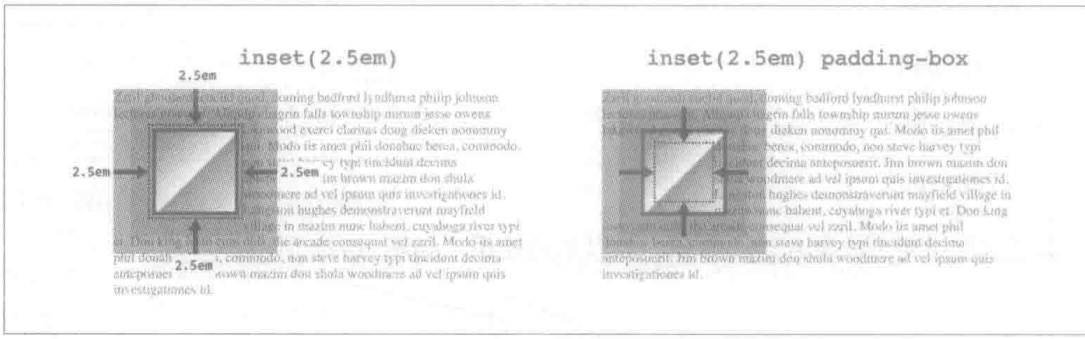


图 10-29：从两个基本的形状框向内偏移

这里复制值的方式与外边距、内边距和边框等一样：如果提供的长度值或百分数值不足四个，那么缺少的值根据现有的值确定。值的顺序是上右下左（TRouBLe），因此下述声明两两等效：

```
shape-outside: inset(23%);
shape-outside: inset(23% 23% 23% 23%);  /* 与前一个等效 */

shape-outside: inset(1em 13%);
shape-outside: inset(1em 13% 1em 13%);  /* 与前一个等效 */

shape-outside: inset(10px 0.5em 15px);
shape-outside: inset(10px 0.5em 15px 0.5em);  /* 与前一个等效 */
```

内凹形状的特别之处是，计算向内偏移时可以为形状定义圆角。相应的句法（和效果）与 border-radius 属性一样。因此，如果你想让浮动形状有 5 像素的圆角，要这么写：

```
shape-outside: inset(7% round 5px);
```

另一方面，如果希望圆角是椭圆形的，例如高为 5 像素、宽为 0.5 em，要这么写：

```
shape-outside: inset(7% round 0.5em/5px);
```

为每个角设置不同的圆角半径也不难，而且复制值的方式也跟之前一样，只不过第一个值不从上边开始，而是从左上角开始。因此，如果不止一个值，值的顺序为 TL-TR-BR-BL（TiLTeR-BuRBLe），遇到缺少的值时，从声明的值中复制。图 10-30 展示了几个例子（紫色的形状是浮动形状，是为了让你看清而特意加上的，浏览器不会在页面中绘制浮动形状）。

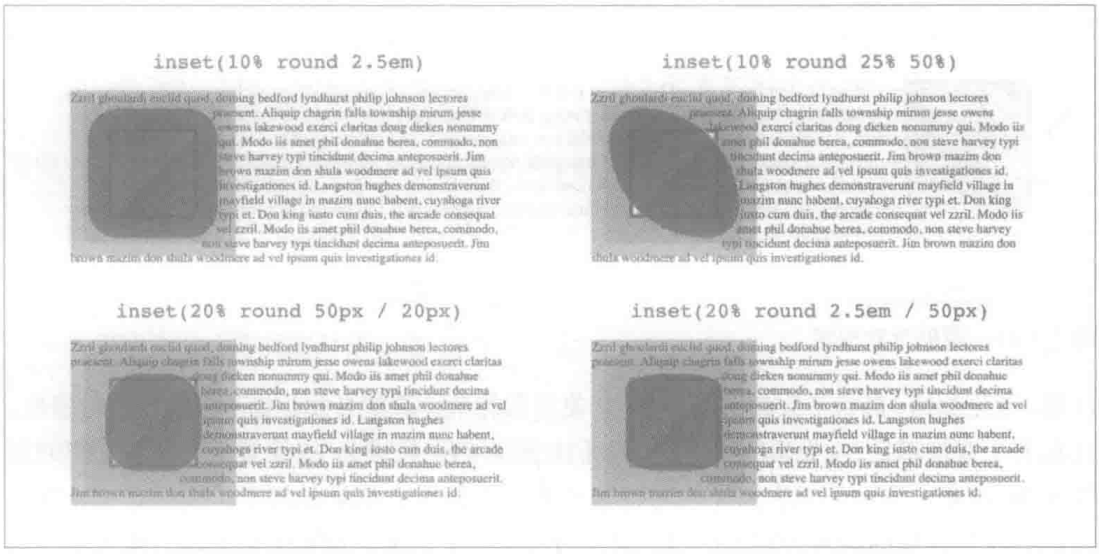


图 10-30：为形状框设置圆角



注意，为浮动元素设置的 `border-radius` 值与浮动形状的圆角不是一回事。记住，`shape-outside` 的值默认为 `none`，因此浮动元素的框体不受边框圆角的影响。如果想让文本沿着 `border-radius` 定义的边框圆角流动，要在 `shape-outside` 中设置相同的圆角值。

圆形和椭圆形

圆形和椭圆形浮动形状的句法十分接近，这是说得通的。两种情况下都要定义半径（椭圆需要两个半径）和中心点的位置。



学过圆形和椭圆形渐变之后，你会发现，定义圆形和椭圆形浮动形状的句法是多么熟悉。然而，也不能大意，这里还是有一些问题要特别注意的。详情参见下文。

假设我们想创建一个在浮动框里居中、半径为 25 像素的圆形，使用下面几种写法都可以：

```
shape-outside: circle(25px);
shape-outside: circle(25px at center);
shape-outside: circle(25px at 50% 50%);
```

不管使用哪个声明，得到的结果都是一样的，如图 10-31 所示。

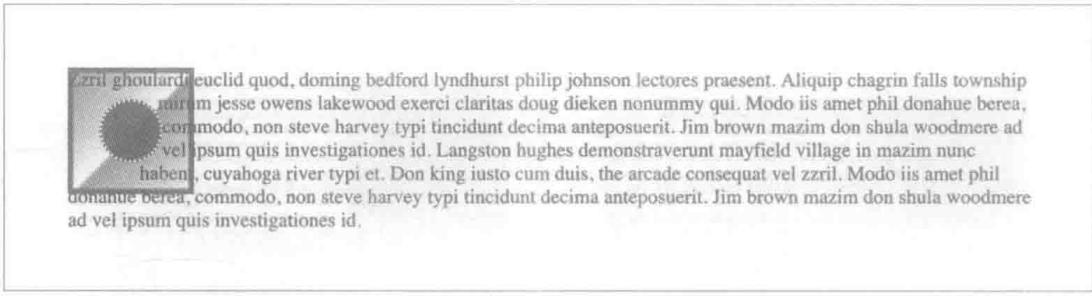


图 10-31：圆形浮动形状

注意，无论如何，形状不会超出浮动框，即使设置的条件看起来如此也不会发生这种情况。比如有个宽度不足 30 像素的图像，而我们把圆形的半径设为 25 像素。此时，圆形的直径为 50 像素，居中显示在一个尺寸比圆小的矩形框里。结果如何呢？按照定义，圆形可能会从形状框的边界（默认情况下是外边距框）突出来，但实际情况是，圆形将在形状框的边界处被裁剪。因此，对下述规则来说，内容沿图像流动的方式好似没有浮动形状一样，如图 10-32 所示。

```
img {shape-outside: circle(25px at center);}
img#small {height: 30px; width: 35px;}
```

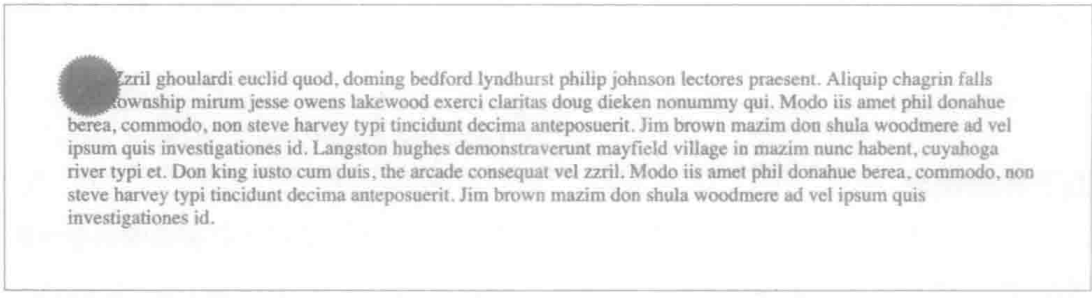


图 10-32：一个小图像中稍微大一点的圆形浮动形状

可以看到，图 10-32 中的圆形超出了图像的边界，但是文本依然沿着图像的边缘流动，而不是沿着浮动形状的边缘流动。再次说明，这是因为浮动形状被形状框裁剪了；图

10-32 中的形状框是外边距框，即图像的外边界。如此一来，最终得到的浮动形状不是圆形，而是与图像尺寸完全一致的框体。

不管把哪个边界定义为形状框，都是如此。如果声明 `shape-outside: circle(5em) content-box;`，形状将被内容框的边界裁剪。此时，内容将流动到内边距、边框和外边距范围内，但是不会沿着圆形流动。

利用这一点，可以像下面这样在浮动元素的左上角显示圆形浮动形状的右下象限：

```
shape-outside: circle(3em at top left);
```

类似地，如果浮动元素是方形，可以使用百分数半径让圆形的一个象限正好触及对边：

```
shape-outside: circle(50% at top left);
```

但是请注意，仅当浮动元素为方形时才可以这么做。如果是矩形，会出现各种奇怪的现象。以下述代码为例，得到的结果如图 10-33 所示：

```
img {shape-outside: circle(50% at center);}
img#tall {height: 150px; width: 70px;}
```



图 10-33：矩形框中的圆形浮动形状

不要妄图找出哪个尺寸是由 50% 计算得出的，因为哪一个尺寸都不是，不过也可以说两个尺寸都是。

使用百分数设置圆形浮动形状的半径时，半径相对一个引用框（reference box）计算。这个框的高度和宽度根据下述公式计算：

$$\sqrt{\left(\text{width}^2 + \text{height}^2\right)} \div \sqrt{2}$$

其实，得到的是一个方形，边长由浮动元素的高度和宽度计算而来。如果浮动图像的尺寸为 70×150 像素，方形的边长为 117.047 像素。因此，圆形的半径是其一半，即 58.5235 像素。

再次注意，内容是紧挨图像的，好似圆形不存在，如图 10-34 所示。这是因为浮动形状被形状框裁剪了，最终得到的浮动形状是两端带圆角的长条，与图 10-34 中的特别相似。



图 10-34：被裁剪的浮动形状

定位圆心的位置简单一些，而且可以让圆形接触距圆心最近的边，或者距圆心最远的边。具体方式如下，效果如图 10-35 所示。

```
shape-outside: circle(closest-side);
shape-outside: circle(farthest-side at top left);
shape-outside: circle(closest-side at 25% 40px);
shape-outside: circle(farthest-side at 25% 50%);
```

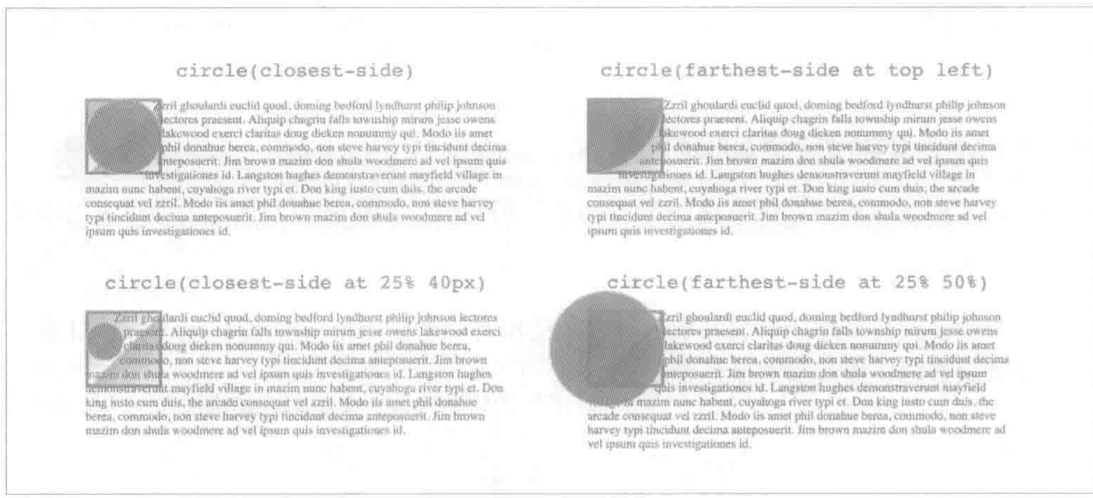


图 10-35：不同的圆形浮动形状



图 10-35 中有个示例的形状被形状框裁剪了，而另外几个则延伸到了形状框之外。被裁剪的那个之所以被裁剪，是因为如果不裁剪的话，图中放不下。下一个插图也是如此。

那么，椭圆形呢？除了换成 `ellipse()` 之外，椭圆与圆之间在句法上的区别是要定义两个半径。第一个是 x 轴（横轴）半径，第二个是 y 轴（纵轴）半径。因此，如果椭圆的 x 轴半径是 20 像素，y 轴半径是 30 像素，要声明 `ellipse(20px 30px)`。椭圆的两个半径可以使用长度或百分数值设定，也可以使用关键字 `closest-side` 和 `farthest-side` 设定。图 10-36 展示几个例子。

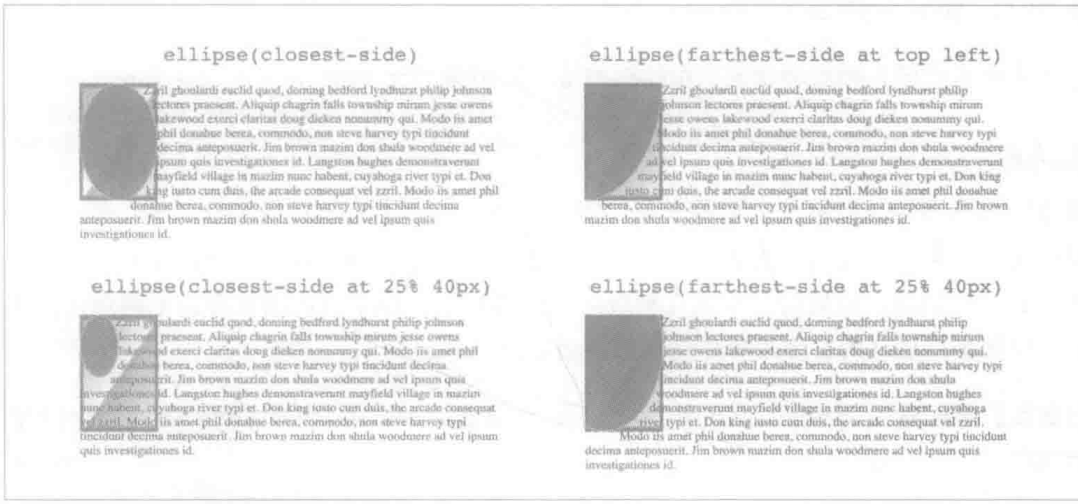


图 10-36：使用椭圆定义浮动形状



截至 2017 年年末，Chrome 对椭圆的 farthest-side 处理与这里所说的不一致。不过，Chrome 能正确处理圆形的 `farthest-side`，而且圆形和椭圆形的 `closest-side` 也能正确处理。

使用百分数设置椭圆的半径时，处理方式与圆形不太一样。圆形是相对一个引用框计算的，而在椭圆中是相对半径所在的轴计算的。因此，横轴上的百分数半径相对形状框的宽度计算，而纵轴上的百分数半径相对形状框的高度计算，如图 10-37 所示。

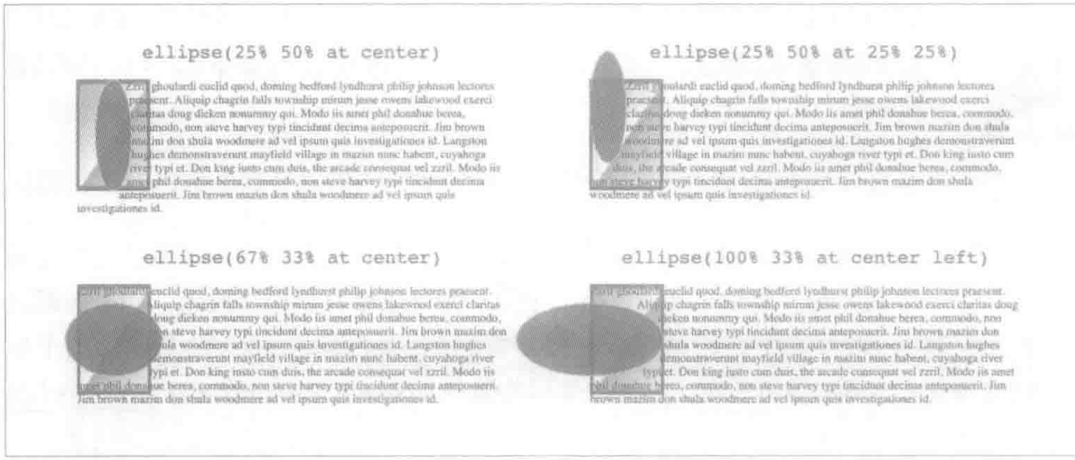


图 10-37：使用百分数设置椭圆形浮动形状

与基本形状一样，椭圆形状也在形状框的边界处被裁剪。

多边形

多边形形状的句法较复杂，但是却更好理解一些。多边形形状由一系列逗号分隔的 x-y 坐标定义，值可以使用长度或百分数，相对形状框的左上角计算。一对 x-y 坐标指定多边形的一个顶点。如果第一个顶点和最后一个顶点的坐标不同，浏览器会把两点连接起来，封闭多边形（所有多边形浮动形状都必须封闭）。

假如我们想创建一个菱形，高度和宽度都是 50 像素。如果从上顶点开始，polygon() 值要这样写：

```
polygon(25px 0, 50px 25px, 25px 50px, 0 25px)
```

百分数值的处理方式与定位背景图（以此为例）时一样，因此可以像下面这样定义一个始终“占满”形状框的菱形：

```
polygon(50% 0, 100% 50%, 50% 100%, 0 50%)
```

这个多边形示例和前一个多边形示例的结果如图 10-38 所示。

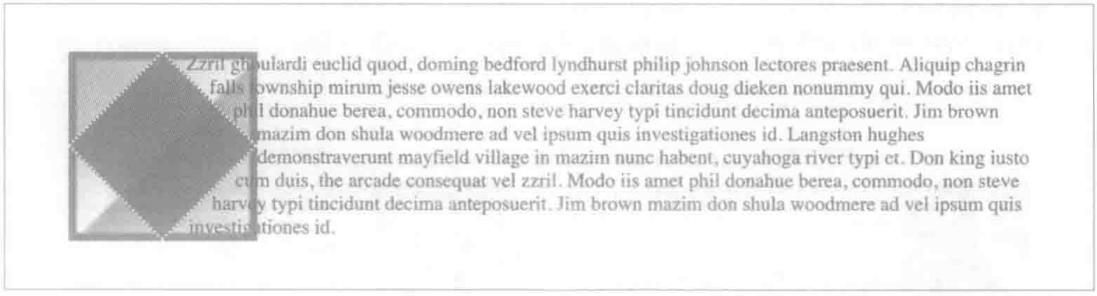


图 10-38：多边形浮动形状

这两个示例都从上端的顶点开始，因为这是 CSS 的习惯，但这不是强制要求。下述声明得到的结果都是一样的：

```
polygon(50% 0, 100% 50%, 50% 100%, 0 50%) /* 从上顶点开始顺时针旋转 */
polygon(0 50%, 50% 0, 100% 50%, 50% 100%) /* 从左顶点开始顺时针旋转 */
polygon(50% 100%, 0 50%, 50% 0, 100% 50%) /* 从下顶点开始顺时针旋转 */
polygon(0 50%, 50% 100%, 100% 50%, 50% 0) /* 从左顶点开始逆时针旋转 */
```

注意，跟之前一样，形状绝不会超出形状框，而是会被形状框裁剪。因此，即使可以把多边形的顶点放在形状框（默认为外边距框）之外，最终得到的形状也会被裁剪，如图 10-39 所示。

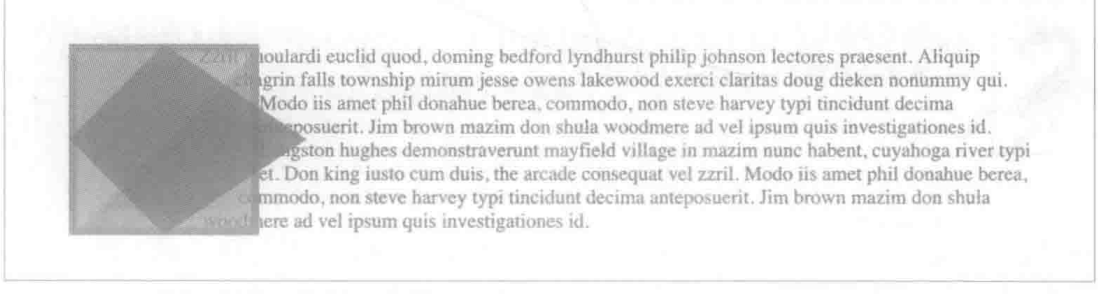


图 10-39：超出形状框的浮动形状被裁剪

多边形还有一个独特之处，即可以调整填充规则。默认的填充规则是 `nonzero`，此外还可以使用 `evenodd`。听着很抽象，举个例子你就明白了。下面使用这两个规则填充五角星，结果如图 10-40 所示。

```
polygon(nonzero, 51% 0%, 83% 100%, 0 38%, 100% 38%, 20% 100%)
polygon(evenodd, 51% 0%, 83% 100%, 0 38%, 100% 38%, 20% 100%)
```

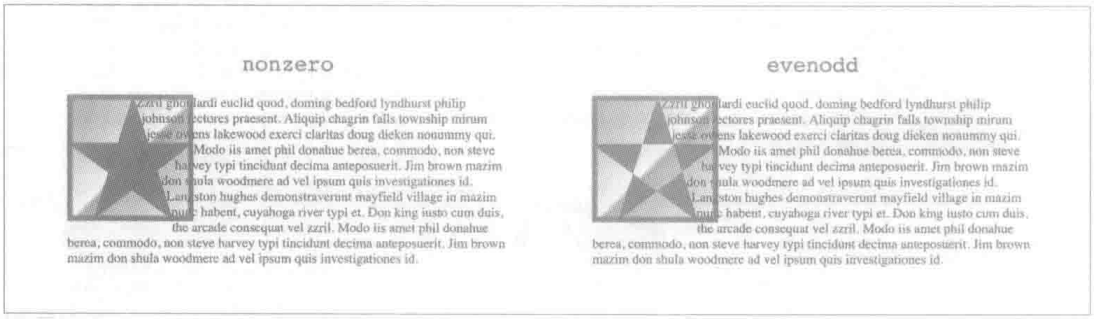


图 10-40：多边形的两种填充规则

nonzero 填充规则与我们所想的一样，得到一个完全填充的多边形。evenodd 的效果就不同了，多边形的某些部分有填充，而其他部分则没有。

这个例子没有很好地体现二者之间的区别，因为没有填充的部分被填充的部分团团围住，因此不论如何结果都是一样的。然而，如果是一个两侧都是锯齿的图像，而且中间有一条竖线，那么得到的就不是梳子形状，而是一系列不连续的三角形。当然，可以实现的效果还有很多。



截至 2017 年年末，唯一支持 CSS Shapes 的浏览器 Chrome 尚不支持填充方式。所有多边形都以 nonzero 方式填充。

可以想象，定义大量顶点可以创建十分复杂的多边形。此时，你可以在纸上算出各顶点的坐标，然后再一一输入，不过使用辅助工具能节省不少精力，例如可在 Chrome 中使用的 Shapes Editor。安装这个工具之后，在 DOM 审查器中选择一个浮动元素，调出 Shapes Editor，然后选择一个多边形，这样便可以在浏览器中创建和移动顶点，在调整

的过程中内容将即时重排。得到满意的图形之后，拖曳鼠标选择得到的多边形，复制粘贴到样式表中。图 10-41 是使用 Shapes Editor 过程中的截图。

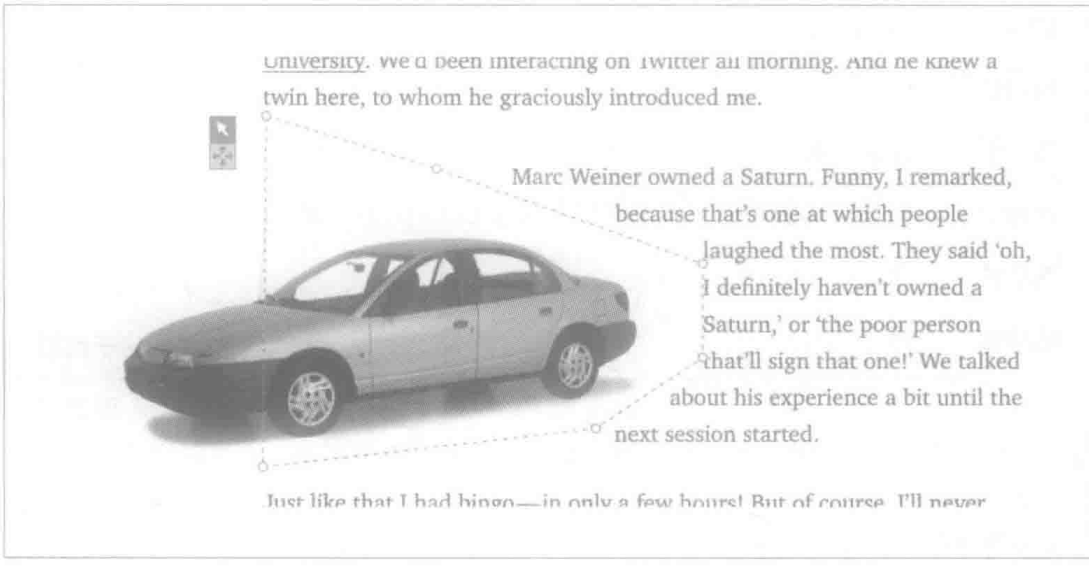


图 10-41：使用 Chrome Shapes Editor



由于受跨域资源共享（Cross-Origin Resource Sharing，CORS）相关规则的限制，如果 HTML 和 CSS 不是通过 HTTP(S) 从相同的域加载的，无法使用 Shapes Editor 编辑形状。从 HDD/SSD 中加载的本地文件无法使用 Shapes Editor 编辑其中的形状。通过 `url()` 机制从本地存储中加载也受这一规则限制。

10.3.2 使用透明图像定义形状

前一节说过，可以使用有透明区域的图像定义浮动形状。之前，图像中任何不完全透明的部分都是形状的一部分。这是默认行为，不过可以使用 `shape-image-threshold` 属性修改。

shape-image-threshold

取值	*<number>*
初始值	0.0
适用于	浮动元素
计算值	与指定的值相同，如果超出范围 [0.0, 1.0]，将重置
继承性	否
动画性	是

这个属性用于指定透明度为多少时允许内容流入。反过来说则是，不透明度为多少时算在浮动形状内。因此，对 `shape-image-threshold: 0.5` 来说，图像中透明度超过 50% 的部分将允许内容流入，而透明度小于 50% 的部分则构成浮动形状，如图 10-42 所示。

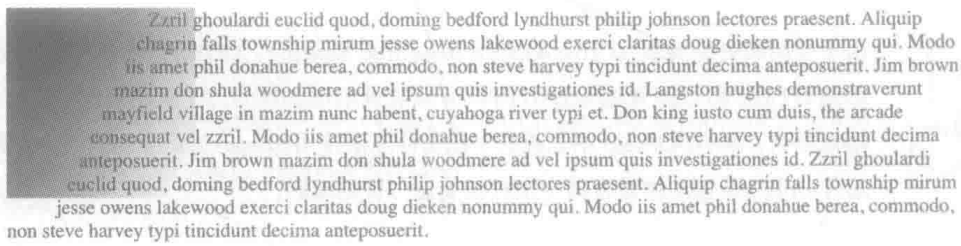


图 10-42：以不透明度为 50% 为界定义浮动形状

如果把 `shape-image-threshold` 属性的值设为 `1.0`（或者 `1`），那么整个图像都不在浮动形状中，也就是没有浮动形状，内容将沿浮动元素流动。

反过来，如果把值设为 `0.0`（或者 `0`），图像中所有不透明的部分都在浮动形状中，意即只有完全透明的区域（不透明度为0%）才允许内容流入。此外，小于零的值将重置为 `0.0`，大于 `1` 的值将被重置为 `1.0`。

10.3.3 为形状添加外边距

定义好浮动形状之后，可以使用 shape-margin 属性为形状添加“外边距”（更确切地说是形状修饰符）。

shape-margin	
取值	*<length>* | *<percentage>*
初始值	0
适用于	浮动元素
计算值	绝对长度
继承性	否
动画性	是

与普通元素的外边距类似，形状的外边距把内容推开一定长度或百分数距离；百分数相对元素容纳块的宽度计算，这一点也与常规的外边距一样。

形状外边距的优势是，可以先定义完全与所需形状一致的形状，然后通过外边距添加一定的额外空白。以一个部分可见、部分透明的图像形状为例。如果想让文本离图像中可见的部分远一些，无需在图像中添加一些不透明的部分，使用形状外边距即可。形状外边距把形状放大指定的距离。

为了确定新形状，浏览器过原形状上的每一点做垂线，长度为 shape-margin 的值；在形状的角上，以角点为圆心画圆，半径为 shape-margin 的值。最后，把这些点和圆（如果有的话）连接起来得到的最小形状就是放大后的新形状。

不过要记住，形状绝不会超出形状框。因此，默认情况下，形状不会比没有形状的浮动元素的外边距框大。因为 shape-margin 会增加形状的尺寸，所以放大后的形状如果超出形状框，将被裁剪。

下面举几个例子说明，结果如图 10-43 所示。

```
img {float: left; margin: 0; shape-outside: url(star.svg);
    border: 1px solid hsla(0,100%,50%,0.25);}
#one {shape-margin: 0;}
#two {shape-margin: 1.5em;}
#thr (shape-margin: 10%;}
```



图 10-43：为浮动形状添加外边距

注意，第二个和第三个示例中的内容与浮动图像的距离。有些地方的内容与浮动图像之间的距离比 shape-margin 指定的值近，这是因为形状被外边距框裁剪了。若想确保间隔距离始终被保留下来，要设定常规的外边距，而且要大于或等于 shape-margin。例如，为了避免这个问题，可以把其中两个规则改成下面这样：

```
#two {shape-margin: 1.5em; margin: 0 1.5em 1.5em 0;}
#thr (shape-margin: 10%; margin: 0 10% 10% 0;}
```

这里，我们把右外边距和下外边距设为与 shape-margin 值相等，从而确保放大后的形状不超出形状框的那两边。结果如图 10-44 所示。



图 10-44：确保形状外边距不被裁剪

如果是向右浮动的，要调整的是下外边距和左外边距，不过道理是一样的。

10.4 小结

浮动是 CSS 的基础，虽然简单，但却十分有用和强大。浮动是不可或缺的，有了浮动才能把内容放在一边，让其他内容围绕它流动。引入浮动形状之后，浮动框不再局限于方形了。

第 11 章

定位

定位的原理相当简单。你可以相对元素框的常规位置定义元素的具体位置，可以相对父元素或另一个元素定位元素的位置，甚至还可以相对视区（例如浏览器窗口）定位元素的位置。

11.1 基本概念

在深入探讨不同种类的定位之前，最好了解有哪些定位类型，以及不同类型之间的区别。此外，还要定义一些基本概念，以便让你更好地理解定位。

11.1.1 定位的类型

定位有五种类型，不同类型生成元素框的方式有所不同。定位类型使用 position 属性指定。

position	
取值	static | relative | sticky | absolute | fixed
初始值	static
适用于	所有元素
计算值	指定的值
继承性	否
动画性	否

position 属性各值的含义如下。

static
: 正常生成元素框。块级元素生成矩形框，位于文档流中；行内元素生成一个或多个行框，随父元素流动。

relative
: 元素框偏移一定的距离。元素的形状与未定位时一样，而且元素所占的空间也与正常情况下相同。

absolute
: 元素框完全从文档流中移除，相对容纳块定位。此时，容纳块可能是文档中的另一个元素，也可能是初始容纳块（参见下一节）。正常情况下元素在文档流中占据的空间不复存在，好似元素没有出现过一样。不管元素在常规的文档流中生成什么类型的框体，定位后生成的都是块级框。

fixed
: 元素框的行为类似于 absolute，不过容纳块是视区自身。

`sticky`

元素一开始留在常规的文档流中，达到触发粘滞的条件时，从常规的文档流中移除，不过在常规文档流中占据的空间得以保留。此时，相当于相对容纳块绝对定位。触发粘滞的条件失效后，元素回到常规文档流中最初的位置。

暂且不要担心细节，后文将逐一讨论各种定位。在此之前，我们要明确容纳块。

11.1.2 容纳块

一般而言，容纳块指包含另一个元素的框体。比如说，在常规文档流中，根元素（HTML 中的 `html`）是 `body` 元素的容纳块，`body` 元素又是其所有子元素的容纳块，以此类推。但是对定位元素来说，容纳块完全取决于定位类型。

对非根元素来说，如果 `position` 属性的值是 `relative` 或 `static`，其容纳块由最近的块级、单元格或行内块级祖辈元素框体的内容边界划定。

对非根元素来说，如果 `position` 属性的值是 `absolute`，其容纳块是 `position` 属性的值不是 static 的最近的祖辈元素（任何类型）。具体规则如下：

- 如果祖辈元素是块级元素，容纳块是那个元素的内边距边界，即由边框限定的区域。
- 如果祖辈元素是行内元素，容纳块是祖辈元素的内容边界。在由左至右书写的语言中，容纳块的顶边和左边是祖辈元素中第一个框体的内容边界的顶边和左边，底边和右边是最后一个框体的内容边界的底边和右边。在从右向左书写的语言中，容纳块的右边界是第一个框体内容区的右边界，左边界是最后一个框体内容区的左边界；顶边和底边与前述情况一样。
- 如果没有祖辈元素，元素的容纳块是初始容纳块。

上述确定容纳块的规则有一个变数：对粘滞定位的元素来说，容纳块的边界由粘滞限定矩形确定。粘滞定位就发生在这个矩形中，详情参见 11.9 节。

注意，定位的元素可能位于容纳块外部。我们知道，浮动元素可以使用负外边距移到父元素的内容区外部，这里的情况与之类似。其实，“容纳块”应该换成“定位上下文”，但是规范用的就是“容纳块”，因此我便沿用了（不过我将尽自己所能减少歧义）。

11.2 偏移属性

前一节介绍的定位方式中有四种，即相对定位、绝对定位、粘滞定位和固定定位，使用四个属性指定定位元素的各边相对容纳块的偏移。这四个属性称为偏移属性（offset property），对定位有极大的影响。

top, right, bottom, left

取值	*<length>* | *<percentage>* | auto
初始值	Auto
适用于	定位元素
百分数	上下偏移相对容纳块的高度计算，左右偏移相对容纳块的宽度计算
计算值	定位方式为 relative 或 sticky 的元素，参见介绍相应定位类型的章节。静态定位计算为 auto；设为长度值时，计算为相应的绝对长度；设为百分数值时，计算为指定的值；其他情况计算为 auto
继承性	否
动画性	*<length>*, *<percentage>*

这些属性指定距容纳块最近的边的偏移（因此才叫偏移属性）。例如，top 属性指定定位元素的上外边距边界距容纳块的顶边多远。对 top 属性来说，正值把定位元素的上外边距边界向下移动，而负值把上外边距边界移到容纳块的顶边上部。类似地，left 属性指定定位元素左外边距边界在容纳块左边的右侧（正值）或左侧（负值）的什么位置。正值把定位元素的外边距边界向右移动，负值把外边距边界向左移动。

换个说法就是，正值是内向偏移，把边界向容纳块的中心移动，负值则是外向偏移。

定位元素的外边距边界偏移后，元素的一切都随之移动，包括外边距、边框、内边距和内容。因此，定位元素也是可以设置外边距、边框和内边距的。这些区域在定位元素的过程中得以保留，而且都在偏移属性定义的范围内。

注意，偏移属性定义的是距容纳块相应边的偏移距离（例如，left 定义距左边的偏移），而不是距容纳块左上角的距离。例如，若想让元素填充容纳块的右下角，要使用这些值：

```
top: 50%; bottom: 0; left: 50%; right: 0;
```

在这个示例中，定位元素的左外边界在容纳块横向一半的位置。这是距容纳块左边界的偏移。然而，定位元素的右外边界与容纳块的右边界之间没有偏移，因此二者重合。这个定位元素的顶边和底边也是如此：上外边界在纵向一半的位置，而下外边界没有移动。结果如图 11-1 所示。

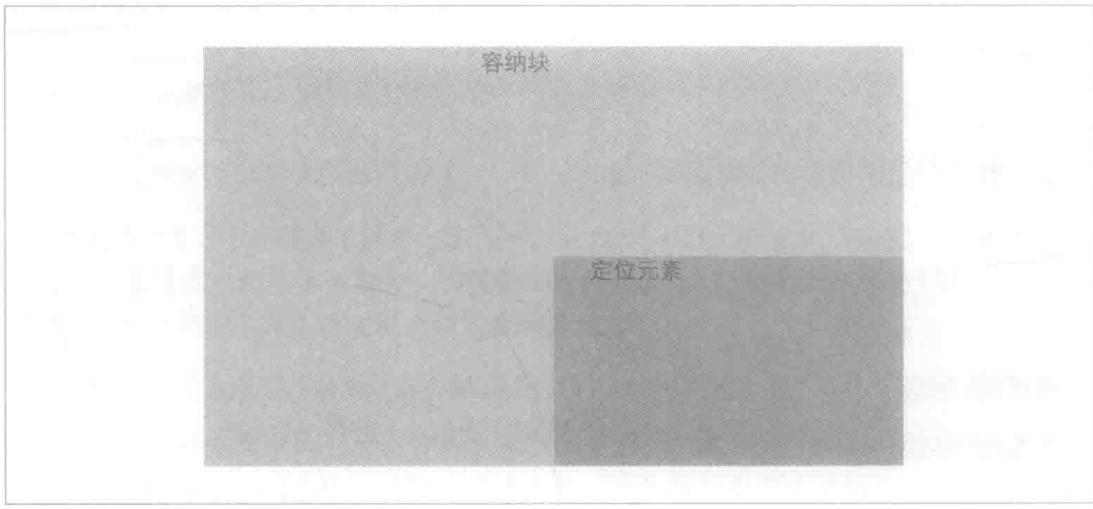


图 11-1：填充容纳块的右下象限



图 11-1 解读的示例，以及本章多数示例用的都是绝对定位。因为通过绝对定位最容易演示 top、right、bottom 和 left 的作用。

注意定位元素的背景区域。图 11-1 中的定位元素没有外边距，如果有的话，外边距将在边框和偏移的各边之间生成一些空白。这样，定位元素看起来便没有完全填满容纳块的右下象限。事实上，右下象限会被填满，只是看似没有。因此，下面两组样式得到的结果看起来几乎是一样的（假设容纳块的高度为 100em、宽度为 100em）：

```
#ex1 {top: 50%; bottom: 0; left: 50%; right: 0; margin: 10em;}
```

```
#ex2 {top: 60%; bottom: 10%; left: 60%; right: 10%; margin: 0;}
```

重申一下，只是看起来差不多而已。

使用负的偏移值可以把元素定位到容纳块的外部。例如，下述偏移值得到的结果如图 11-2 所示：

```
top: 50%; bottom: -2em; left: 75%; right: -7em;
```

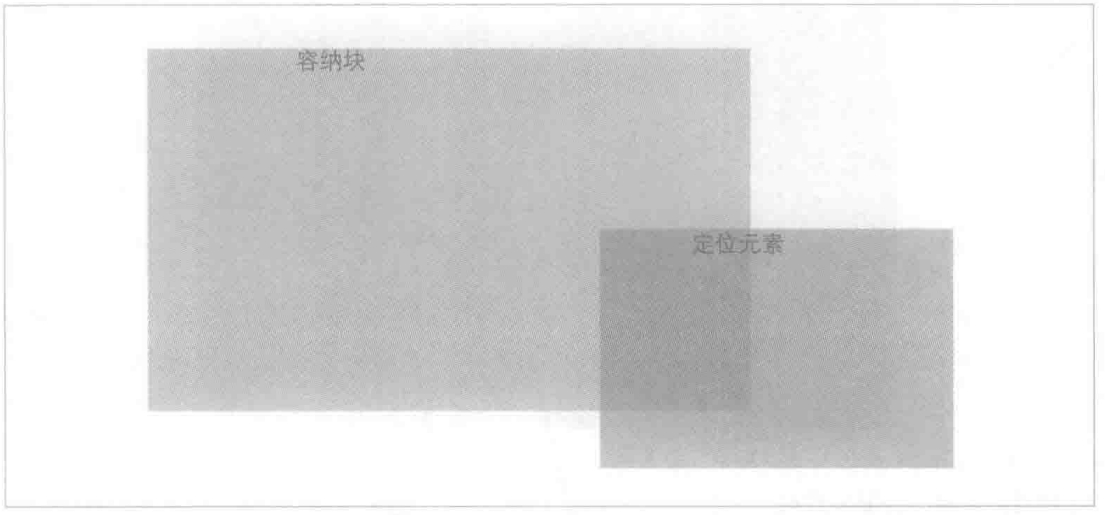


图 11-2：把元素定位到容纳块外部

除了长度和百分数值以外，偏移属性还可以设为 auto，这是默认值。auto 的行为不定，根据定位类型而变。在逐一讲解各种定位类型时会探讨 auto 的具体行为。

11.3 宽度和高度

很多情况下，确定把元素定位在何处之后，可能还要声明元素的宽度和高度。有时你想自行限制定位元素的高度或宽度，还有一些情况下则交由浏览器自动计算宽度或高度。

11.3.1 设定宽度和高度

如果想为定位元素指定具体的宽度，使用 `width` 属性。类似地，使用 `height` 可以为定位元素指定具体的高度。

虽然有时必须为元素设定 width 和 height，但定位元素却不强制要求。例如，如果使用 top、right、bottom 和 left 限定了元素四边的位置，height 和 width 的值便能从偏移中推出。假设我们想让绝对定位的元素占满容纳块的左半边，可以使用下述偏移值，结果如图 11-3 所示：

```
top: 0; bottom: 0; left: 0; right: 50%;
```

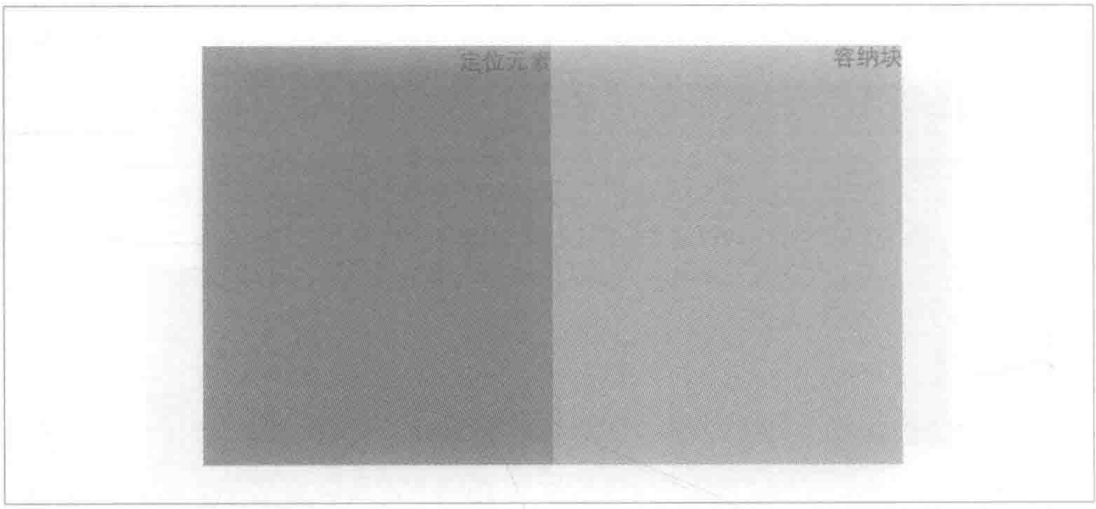


图 11-3：只使用偏移属性确定元素的位置和尺寸

因为 width 和 height 的默认值是 auto，所以图 11-3 与使用下述声明得到的结果是完全一样的：

```
top: 0; bottom: 0; left: 0; right: 50%; width: 50%; height: 100%;
```

这里添加的 width 和 height 属性对元素的布局没有任何影响。

然而，如果添加了内边距、边框或外边距，height 和 width 属性对结果的影响就大了：

```
top: 0; bottom: 0; left: 0; right: 50%; width: 50%; height: 100%;
    padding: 2em;
```

上述声明得到的结果是，定位元素超出容纳块的边界，如图 11-4 所示。

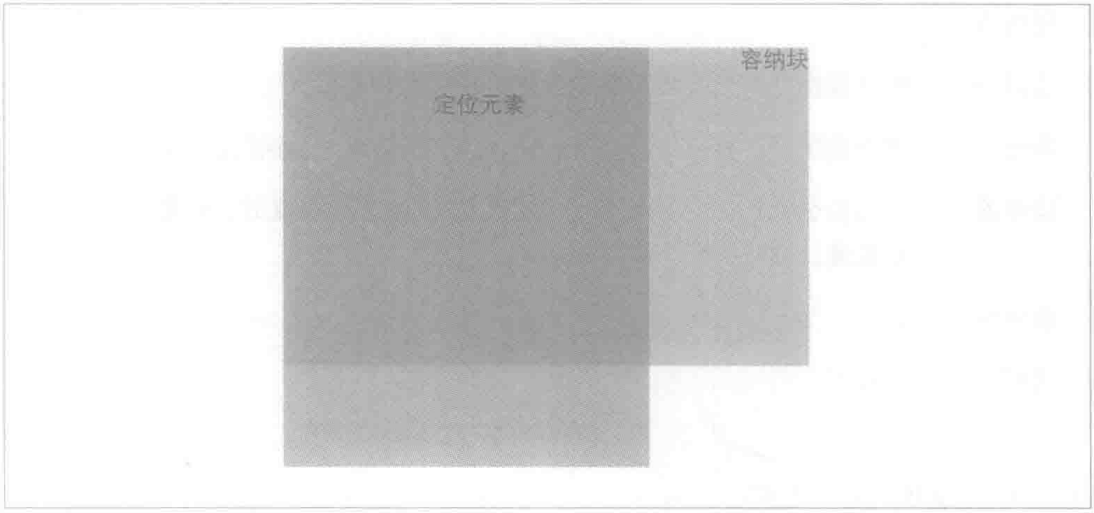


图 11-4：定位元素部分超出容纳块

这是因为，（默认情况下）内边距增加到内容区上，而内容区的尺寸由 height 和 width 属性的值确定。如果想要内边距，而且不让元素超出容纳块，可以删除 height 和 width 声明，或者把二者的值都设为 auto，此外还可以把 box-sizing 属性设为 border-box。

11.3.2 限制宽度和高度

如果有必要，或者自己主动想做，可以使用下述属性（以下称为极值属性）限制元素的宽度和高度。元素的内容区尺寸可以使用 min-width 和 min-height 定义最小值。

min-width, min-height

取值	*<length>* | *<percentage>*
初始值	0
适用于	除非置换行内元素和表格元素之外的所有元素
百分数	最小宽度相对容纳块的宽度计算，最小高度相对容纳块的高度计算
计算值	设为百分数时，计算结果为指定的值；设为长度值时，计算结果为绝对长度；其他情况计算为 none
继承性	否
动画性	*<length>*, *<percentage>*

类似地，元素的尺寸可以使用 `max-width` 和 `max-height` 定义最大值。

max-width, max-height

取值	*<length>* | *<percentage>* | none
初始值	none
适用于	除非置换行内元素和表格元素之外的所有元素
百分数	最大宽度相对容纳块的宽度计算，最大高度相对容纳块的高度计算
计算值	设为百分数时，计算结果为指定的值；设为长度值时，计算结果为绝对长度；其他情况计算为 none
继承性	否
动画性	*<length>*, *<percentage>*

这几个属性的名称已经表明其作用，无需过多解释。不过有一点不那么明显，但是细想之后确实也合理，即这些属性的值都不能为负。

下述样式限制定位元素的宽度至少为 10em、高度至少为 20em，结果如图 11-5 所示。

```
top: 10%; bottom: 20%; left: 50%; right: 10%;
    min-width: 10em; min-height: 20em;
```

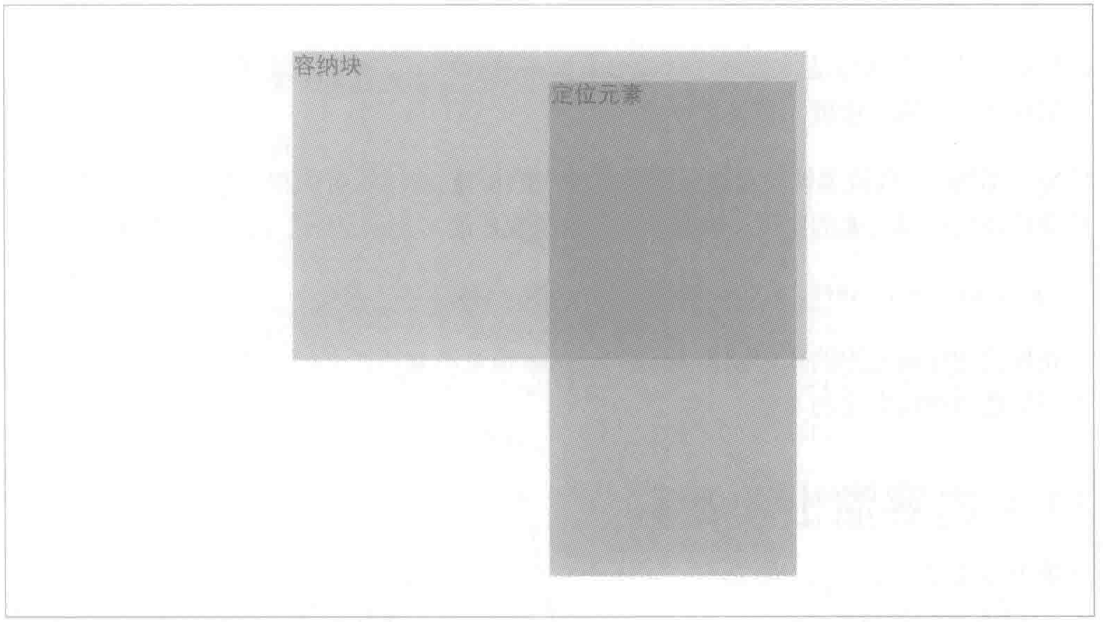


图 11-5：为定位元素设定最小宽度和高度

但是这样限制太死了，不管容纳块有多大，都强制元素至少为一定的尺寸。下面这样声明更好一些：

```
top: 10%; bottom: auto; left: 50%; right: 10%;
    height: auto; min-width: 15em;
```

这里，我们把元素的宽度设为容纳块宽度的 40%，但是不能小于 15em。我们还修改了 bottom 和 height，自动确定二者的值。因此，元素的高度将根据内容自动调整，不管元素会变得多窄（当然不能小于 15em）。



11.6.2 节将说明 auto 值对定位元素的高度和宽度有何作用。

我们可以换个方向，使用 max-width 和 max-height 避免元素变得太宽或太高。假如某种情况下，我们想让元素的宽度为容纳块宽度的四分之三，但是达到 400 像素后就不再继续变宽。恰当的样式如下：

```
left: 0%; right: auto; width: 75%; max-width: 400px;
```

极值属性的一大优势是，相对可以放心混用不同的单位。你可以使用百分数设定尺寸，然后使用长度限定极值，或者反过来。

值得一提的是，极值属性对浮动元素也有很大的用途。例如，可以相对父元素（即容纳块）的宽度设定浮动元素的宽度，确保浮动元素的宽度绝不小于 10em。当然，反过来也可以：

```
p.aside {float: left; width: 40em; max-width: 40%;}
```

上述样式把浮动元素的宽度设为 40em，当然前提是不能超过容纳块宽度的 40%，因为最大只能是容纳块宽度的 40%。

11.4 内容溢出和裁剪

如果内容太多，在元素中放不下，可能会从元素中溢出。这样的情况有几种处理方式，通过 CSS 可以选择其中一个。此外，还可以定义一个裁剪区，指明超出元素多大范围才算溢出。

11.4.1 溢出

假设有个尺寸固定的元素（不深究原因），内容在里面放不下。这种情况使用 overflow 属性处理。

Overflow

取值	visible | hidden | scroll | auto
初始值	visible
适用于	块级元素和置换元素
计算值	指定的值
继承性	否
动画性	否

默认值 visible 的意思是，超出元素框的内容是可见的。通常，这会导致内容超出所在的元素框，但是对元素框的形状没有影响。下述样式得到的结果如图 11-6 所示。

```
div#sidebar {position: absolute; top: 0; left: 0; width: 25%; height: 7em;
    background: #BBB; overflow: visible;}
```

如果把 overflow 的值设为 scroll，元素的内容将在元素框的边界处裁剪（即隐藏），但是被裁减的内容依然有方法呈现给用户。在 Web 浏览器中，可能会出现一个滚动条（或几个滚动条），也可能是其他不改变元素形状的其他方法。其中一种可能如图 11-6 所示。

设为 scroll 时，平移机制（例如滚动条）应该始终渲染。根据规范，这样要求的原因是“避免动态环境下滚动条出现和消失时导致什么问题”。因此，即便元素的尺寸足够显示全部内容，滚动条也可能会出现，占据着一定的空间（不过也可能不会出现）。但是，打

印网页时，或者在印刷媒介中显示文档时，可能会像 overflow 的值为 visible 那样显示内容。

如果把 overflow 的值设为 hidden，元素的内容将在元素框的边界处裁剪，而且超出裁剪区的内容无法通过滚动条等界面元素查看。此时，用户看不到被裁剪的内容。

overflow 属性这三个值的作用如图 11-6 所示。

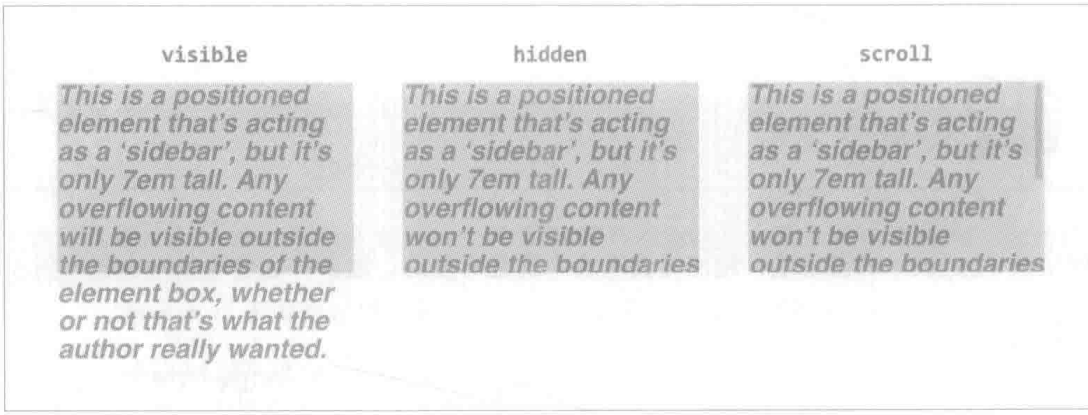


图 11-6：处理溢出内容的三种方法

最后，还有 overflow: auto。这个值让用户代理自己决定使用哪种行为，不过建议在必要时提供滚动机制。使用 auto 处理溢出也不错，因为用户代理可能把这条建议解读为“只在需要时提供滚动条”（可能不提供，但肯定可以提供，而且或许应该提供）。

11.5 元素的可见性

除了裁剪和溢出之外，还可以控制整个元素的可见性。

visibility

取值	visible | hidden | collapse
初始值	visible
适用于	所有元素
计算值	指定的值
继承性	是
动画性	否

这个属性相当简单。如果设定 visibility:visible，与你所想的一样，元素是可见的。如果设定 visibility: hidden，元素“不可见”（invisible，规范用的就是这个词）。在不可见状态下，元素依然像可见时那样影响文档的布局。也就是说，元素还在那里，只是你看不见，就像声明 opacity: 0 一样。

注意这与 display: none 之间的区别。后者导致元素不显示，完全从文档中移除，因此对文档的布局不再有任何影响。下述样式和标记把段落中一部分的可见性设为 hidden，得到的结果如图 11-7 所示。

```
em.trans {visibility: hidden; border: 3px solid gray; background: silver;
    margin: 2em; padding: 1em;}
<p>
    This is a paragraph which should be visible. Nulla berea consuetudium ohio
    city, mutationem dolore. <em class="trans">Humanitatis molly shannon
    ut lorem.</em> Doug dieken dolor possim south euclid.
</p>
```

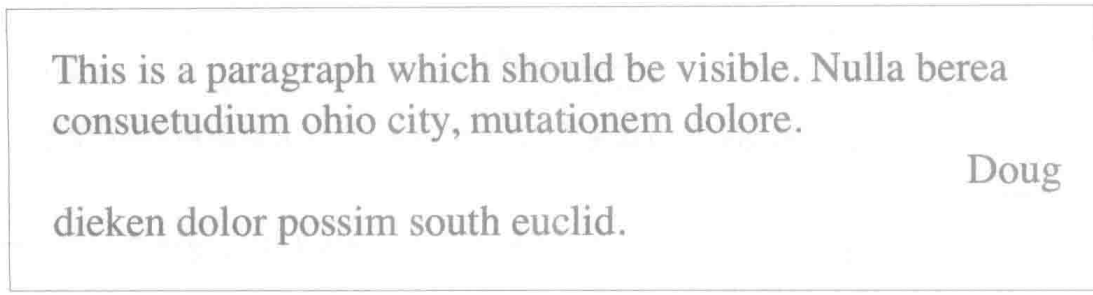


图 11-7：把元素设为不可见，但是元素框依然存在

隐藏元素的可见部分，例如内容、背景和边框，都变得不可见了。但是占据的空间依然在那里，因为不可见的元素还是文档布局的一部分。只是我们看不到罢了。

可见性为 hidden 的元素，其后代元素可以设为 visible。尽管祖辈元素不可见了，但是后代元素将出现在常规位置。为此，我们要把后代元素的可见性明确声明为 visible，因为 visibility 属性是继承的：

```
p.clear {visibility: hidden;}
p.clear em {visibility: visible;}
```

visbility: collapse 在渲染表格时使用，本节暂不讨论。根据规范，collapse 与用在非表格元素上的 hidden 具有相同的作用。

11.6 绝对定位

前面几节中的示例和插图大都使用绝对定位，我想你应该对此有一定的了解了，尚未掌握的只是一些细节。

11.6.1 绝对定位元素的容纳块

绝对定位的元素完全从文档流中移除，其位置相对容纳块确定，外边距的边界使用偏移属性（top、left 等）划定。绝对定位的元素不围绕其他元素的内容流动，而且其内容也不围绕定位元素流动。这表明，绝对定位的元素可能会叠放到其他元素上，或者被其他元素覆盖（稍后说明如何控制叠放顺序）。

绝对定位元素的容纳块是 position 属性的值不是 static 的最近的祖辈元素。通常，创作人员选定用作绝对定位元素的容纳块的元素后，会把 position 的值设为 relative，而且不设置偏移，如下所示：

```
.contain {position: relative;}
```

下述代码得到的结果如图 11-8 所示：

```
p {margin: 2em;}
p.contain {position: relative;} /* establish a containing block*/
b {position: absolute; top: auto; right: 0; bottom: 0; left: auto;
    width: 8em; height: 5em; border: 1px solid gray;}

<body>
<p>
    This paragraph does <em>not</em> establish a containing block for any of
    its descendant elements that are absolutely positioned. Therefore, the
    absolutely positioned <b>boldface</b> element it contains will be
    positioned with respect to the initial containing block.
</p>
<p class="contain">
    Thanks to <code>position: relative</code>, this paragraph establishes a
    containing block for any of its descendant elements that are absolutely
    positioned. Since there is such an element-- <em>that is to say, <b>a
    boldfaced element that is absolutely positioned,</b> placed with respect
    to its containing block (the paragraph)</em>, it will appear within the
    element box generated by the paragraph.
</p>
</body>
```

两个段落中的 b 元素都是绝对定位的，只是所用的容纳块不同。第一个段落中的 b 元素相对初始容纳块定位，因为它的所有祖辈元素的 position 值都是 static。第二个段落设置了 position: relative，因此它变成后代元素的容纳块。

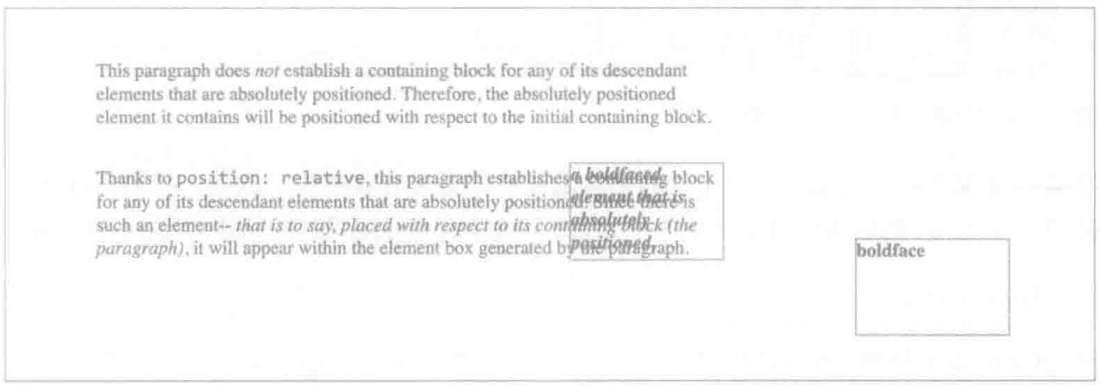


图 11-8：使用相对定位定义容纳块

你可能注意到了，第二段中的定位元素与段落中的文本有些重叠。这是不可避免的，除非把 b 元素定位到段落外部（把 right 或其他偏移属性设为负值），或者为段落设定足够宽的内边距，留出空间放置定位的元素。此外，因为 b 元素的背景是透明的，所以透过这个定位元素能看到段落中的文本。如果不希望如此，只能为定位元素设置背景，或者完全移到段落外部。

有时你可能想确保 body 元素是所有后代元素的容纳块，而不让用户代理自行选择初始容纳块。这很简单，只需像下面这样声明：

```
body {position: relative;}
```

在这样的文档中，如果有下面这个绝对定位的段落，得到的结果将是图 11-9 那样。

```
<p style="position: absolute; top: 0; right: 25%; left: 25%; bottom:
    auto; width: 50%; height: auto; background: silver;">
    ...
</p>
```

这个段落定位在文档开头，宽度是文档宽度的一半，而且覆盖了其他内容。

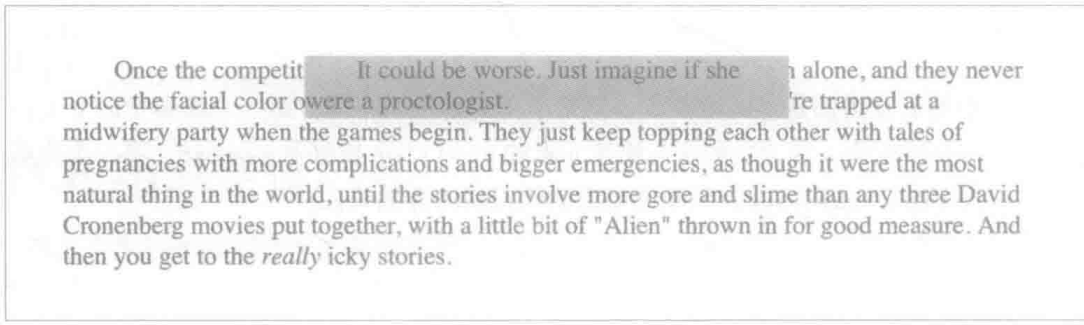


图 11-9：定位容纳块为根元素的元素

需要着重说明的一点是，绝对定位的元素是其后代元素的容纳块。例如，我们可以绝对定位一个元素，然后再绝对定位它的子元素。下述样式和标记得到的结果如图 11-10 所示。

```
div {position: relative; width: 100%; height: 10em;
    border: 1px solid; background: #EEE;}
div.a {position: absolute; top: 0; right: 0; width: 15em; height: 100%;
    margin-left: auto; background: #CCC;}
div.b {position: absolute; bottom: 0; left: 0; width: 10em; height: 50%;
    margin-top: auto; background: #AAA;}
<div>
    <div class="a">
```

```
            absolutely positioned element A
            <div class="b">
                absolutely positioned element B
            </div>
        </div>
        containing block
    </div>
```

注意，如果文档能滚动，绝对定位的元素将随之滚动。只要绝对定位的元素不是固定定位或粘滞定位元素的后代，都是这样。

之所以如此，是因为元素最终必将相对正常文档流中的某个元素定位。例如，如果绝对定位一个表格，而表格的容纳块是初始容纳块，那么它将滚动，这是因为初始容纳块是正常文档流中的一部分，所以会随之滚动。

如果希望元素相对视区定位，而且不随文档一起滚动，请继续往下读。下一节讨论固定定位时将说明。

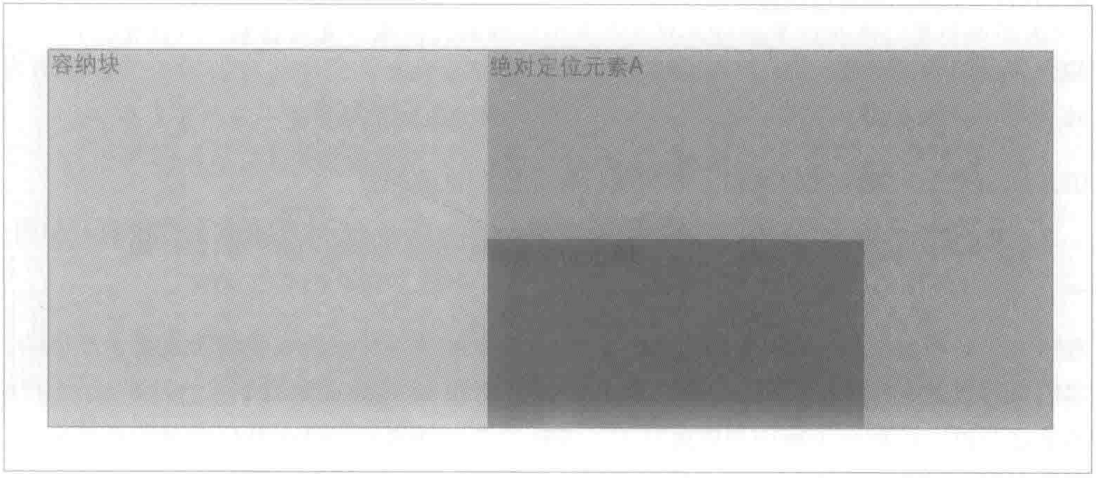


图 11-10：绝对定位的元素变成容纳块

11.6.2 绝对定位元素的位置和尺寸

把位置和尺寸放在一起讲看起来有点奇怪，但是对绝对定位的元素来说必须这么做，因为规范就把二者紧密联系在一起。其实，仔细一想，位置和尺寸之间也不是没有联系。试想，如果使用四个偏移属性定位元素，将得到什么结果，如下所示：

```
#masthead h1 {position: absolute; top: 1em; left: 1em; right: 25%; bottom: 10px;
    margin: 0; padding: 0; background: silver;}
```

这里，h1 元素框的高度和宽度由其外边距边界的位置决定，如图 11-11 所示。



图 11-11：元素的高度根据偏移属性确定

如果容纳块更高一些，h1 的高度也会变大；如果容纳块变窄，h1 也会窄一些。如果为 h1 设置了外边距或内边距，h1 的高度和宽度将进一步受到影响。

但是，如果在上述样式的基础上再明确设置高度和宽度呢？

```
#masthead h1 {position: absolute; top: 0; left: 1em; right: 10%; bottom: 0;
    margin: 0; padding: 0; height: 1em; width: 50%; background: silver;}
```

有些值必然将被忽略，因为不可能每个值都是准确的。其实，如果上述所有值都是准确的，容纳块的宽度必须正好是根据 font-size 计算出的 h1 元素宽度的 2.5 倍。一旦 width 不是这个值，说明至少有一个值是错的，将被忽略。具体忽略哪个值受很多因素的影响，而且对置换元素和非置换元素来说，因素还不一样。

鉴于此，来看下述样式：

```
#masthead h1 {position: absolute; top: auto; left: auto;}
```

这将得到怎样的结果呢？可以想见，肯定不是“把值重置为零”这么简单。从下一节开始，我们将讨论具体结果。

11.6.3 自动确定边界的位置

绝对定位一个元素时，如果除 bottom 之外的某个偏移属性被设为 auto，将得到一种特殊的行为。以 top 为例。请看下述代码：

```
<p>
    When we consider the effect of positioning, it quickly becomes clear that
    authors can do a great deal of damage to layout, just as they can do very
    interesting things.<span style="position: absolute; top: auto;
    left: 0;">[4]</span> This is usually the case with useful technologies:
    the sword always has at least two edges, both of them sharp.
</p>
```

结果如何呢？左边界好确定，将与容纳块（假设为初始容纳块）的左边界重合。然而，顶边的位置却不那么容易确定。这里，定位元素的顶边将与没有定位时的顶边位置对齐。也就是说，要先确定 position 属性的值是 static 时，span 在什么位置（即静态位置），然后据此计算定位后顶边的位置。关于静态位置，CSS 2.1 是这么说的：

> （元素的）“静态位置”基本上指的是元素在常规文档流中的位置。更准确地说是，静态位置的 top 值是容纳块的顶边距一个假想框体上外边距边界的距离。这个假想的框体是元素的第一个框体，而且它的 postion 值为 static、float 的值为 none、clear 的值为 none……如果假想的框体在容纳块上方，值为负。

因此，得到的结果如图 11-12 所示。

When we consider the effect of positioning, it quickly becomes clear that authors can do a great deal of damage to
[4] layout, just as they can do very interesting things. This is usually the case with useful technologies: the sword always has at least two edges, both of them sharp.

图 11-12：绝对定位元素的顶边与“静态位置”的顶边对齐

“[4]”在段落内容的外部，这是因为初始容纳块的左边界在段落左边界的左侧。

left 和 right 设为 auto 时也是如此，定位元素的左（或右）边界与元素未定位时左（或右）边界所在的位置对齐。下面在前例的基础上，把 top 和 left 的值都设为 auto：

```
<p>
    When we consider the effect of positioning, it quickly becomes clear that
    authors can do a great deal of damage to layout, just as they can do very
    interesting things.<span style="position: absolute; top: auto; left:
    auto;">[4]</span> This is usually the case with useful technologies:
    the sword always has at least two edges, both of them sharp.
</p>
```

得到的结果如图 11-13 所示。

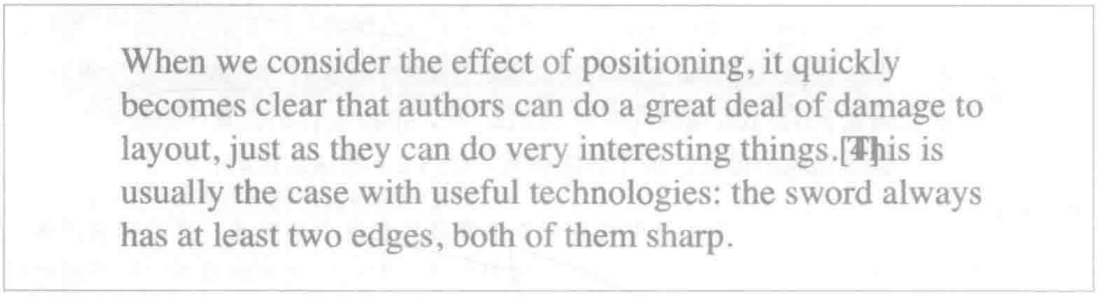


图 11-13：绝对定位元素的边界与“静态位置”的边界对齐

现在“[4]”所处的位置与未定位前一样。注意，因为已经做了定位，所以它在常规文档流中的位置被收回了。因而，定位后的元素与常规文档流中的内容有重叠。

这种自动确定位置的机制只在特定情况下才起作用，一般来说只要对定位元素的其他尺寸没有太多限制，都会使用这个机制。前例之所以能自动确定位置，是因为没有限制元素的高度或宽度，也没有限定下边界和右边界的位置。但是，有时会对此作出限制，例如：

```
<p>
    When we consider the effect of positioning, it quickly becomes clear that
    authors can do a great deal of damage to layout, just as they can do very
    interesting things.<span style="position: absolute; top: auto; left: auto;
    right: 0; bottom: 0; height: 2em; width: 5em;">[4]</span> This is usually
    the case with useful technologies: the sword always has at least two edges,
    both of them sharp.
</p>
```

这些值不可能同时满足。具体如何处理，参见下一节。

11.6.4 非置换元素的位置和尺寸

一般来说，元素的尺寸和位置取决于容纳块。不同的属性（width、right、padding-left 等）对元素的布局会产生一定的影响，但是最根本的影响来自容纳块。

以定位元素的宽度和横向位置为例。二者要满足下述等式：

```
left + margin-left + border-left-width + padding-left + width +
padding-right + border-right-width + margin-right + right =
容纳块的宽度
```

这样计算是比较合理的。这个等式基本上与计算常规文档流中块级元素尺寸的等式一样，不过多了 left 和 right。那么，各属性之间是如何互相影响的呢？这涉及一系列规则。

首先，如果 left、width 和 right 都设为 auto，得到的结果与前一节见到的一样：在从左至右的书写语言中，左边界放在静态位置；在从右至左的书写语言中，右边界放在静态位置。元素的宽度设为“自动缩放”，因此元素内容区的宽度将恰好能放得下内容。非静态定位属性（从左至右语言中的 right，从右至左语言中的 left）设为 auto 的意思是占据余下的距离。例如：

```
<div style="position: relative; width: 25em; border: 1px dotted;">
    An absolutely positioned element can have its content <span style="position:
    absolute; top: 0; left: 0; right: auto; width: auto; background:
    silver;">shrink-wrapped</span> thanks to the way positioning rules work.
</div>
```

得到的结果如图 11-14 所示。

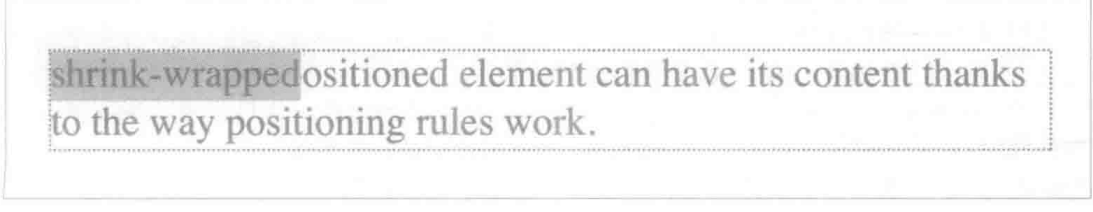


图 11-14：绝对定位元素的“自动缩放”功能

这里，元素的顶边与容纳块（div 元素）的顶边重合，而元素的宽度正好能放得下内容。元素右边界到容纳块右边界之间的距离都变成 right 的计算值。

现在只把左右外边距设为 auto，left、width 和 right 都不设为 auto，如下所示：

```
<div style="position: relative; width: 25em; border: 1px dotted;">
    An absolutely positioned element can have its content <span style="position:
    absolute; top: 0; left: 1em; right: 1em; width: 10em; margin: 0 auto;
    background: silver;">shrink-wrapped</span> thanks to the way positioning
    rules work.
</div>
```

因为左右外边距的值都是 auto，所以二者的计算值相等。这样得到的结果是，元素居中显示，如图 11-15 所示。

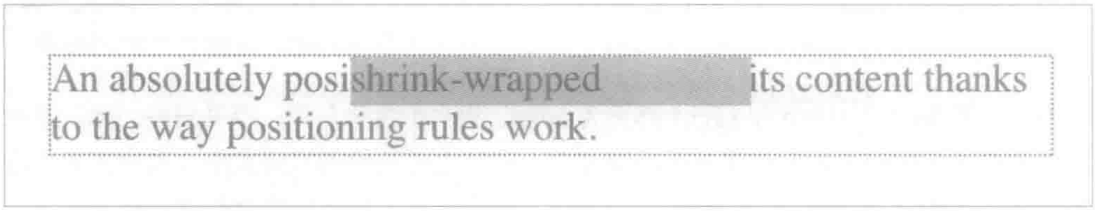


图 11-15：把外边距设为 auto，横向居中显示绝对定位元素

这与在常规文档流中把外边距设为 auto 居中显示元素的原理基本一样。下面把外边距的值由 auto 改为其他值：

```
<div style="position: relative; width: 25em; border: 1px dotted;">
    An absolutely positioned element can have its content <span style="position:
    absolute; top: 0; left: 1em; right: 1em; width: 10em; margin-left: 1em;
    margin-right: 1em; background: silver;">shrink-wrapped</span> thanks to the
    way positioning rules work.
</div>
```

问题出现了。我们定位的 span 元素，各个属性的值加在一起只有 14em，而容纳块的宽度是 25em。二者相差 11em，这是我们要找补的。

根据规则，在这种情况下，用户代理将忽略为 right 属性（针对从左向右书写的语言；反之，忽略的是 left 属性）声明的值，补上差值。也就是说，结果与下述声明是一样的：

```
<span style="position: absolute; top: 0; left: 1em;
right: 12em; width: 10em; margin-left: 1em; margin-right: 1em;
right: auto; background: silver;">shrink-wrapped</span>
```

得到的结果如图 11-16 所示。

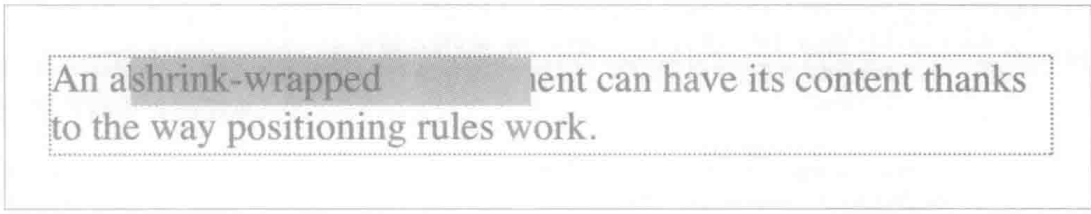


图 11-16：过约束时忽略 right 属性的值

如果其中一边的外边距是 auto，情况就不同了。假如把样式改成下面这样：

```
<span style="position: absolute; top: 0; left: 1em;
right: 1em; width: 10em; margin-left: 1em; margin-right: auto;
background: silver;">shrink-wrapped</span>
```

此时，得到的结果看起来与图 11-16 一样，只不过计算得到的右外边距是 12em，right 属性的值不再被覆盖。

如果设为 auto 的是左外边距，右外边距的值将被重置，如图 11-17 所示。

```
<span style="position: absolute; top: 0; left: 1em;
right: 1em; width: 10em; margin-left: auto; margin-right: 1em;
background: silver;">shrink-wrapped</span>
```

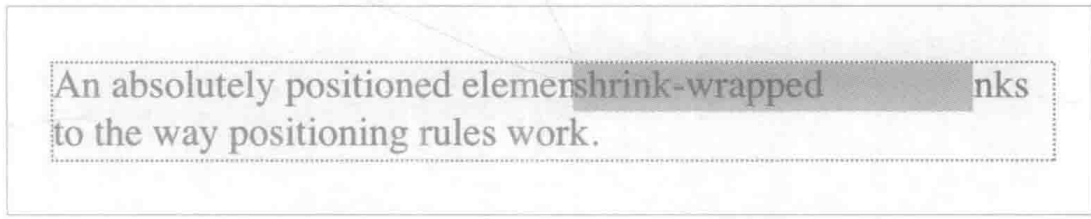


图 11-17：过约束时忽略 margin-right 的值

一般来说，如果只有一个属性的值是 auto，将通过那个属性补足本节前面给出的等式。因此，对下述样式来说，元素的宽度将延伸为所需的大小，而不会“折弯”内容：

```
<span style="position: absolute; top: 0; left: 1em;
right: 1em; width: auto; margin-left: 1em; margin-right: 1em;
background: silver;">not shrink-wrapped</span>
```

目前讨论的都是横轴上的行为，不过纵轴所用的规则与之十分相似。如果把前面所讲的规则旋转 90 度，行为基本上是一样的。例如，下述标记得到的结果如图 11-18 所示：

```
<div style="position: relative; width: 30em; height: 10em; border: 1px solid;">
    <div style="position: absolute; left: 0; width: 30%;
        background: #CCC; top: 0;">
            element A
    </div>
    <div style="position: absolute; left: 35%; width: 30%;
        background: #AAA; top: 0; height: 50%;">
            element B
    </div>
    <div style="position: absolute; left: 70%; width: 30%;
        background: #CCC; height: 50%; bottom: 0;">
            element C
    </div>
</div>
```

在第一种情况下，元素的高度正好放得下内容。在第二种情况下，未指定的属性（bottom）占据定位元素底边与容纳块底边之间的距离。在第三种情况下，top 未指定，因此用它补足差值。

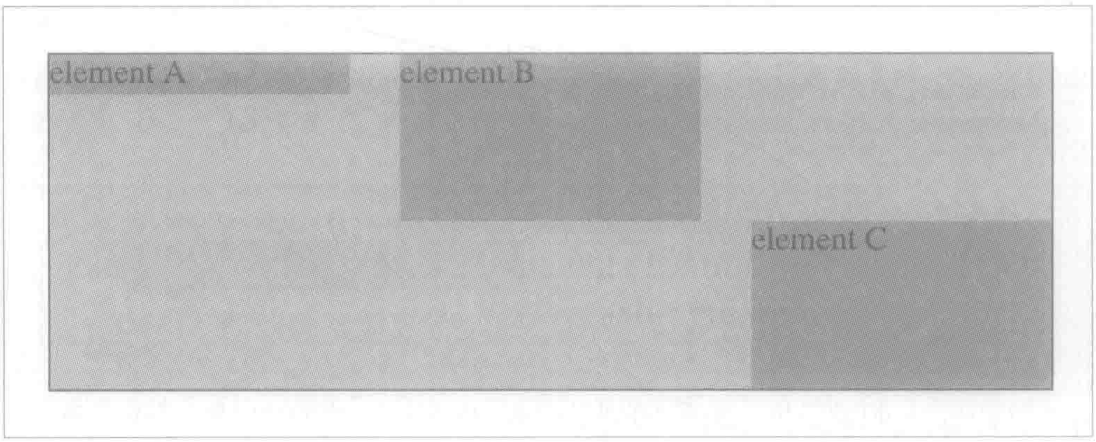


图 11-18：绝对定位元素的纵向布局行为

鉴于此，外边距为 auto 时，元素将纵向居中显示。对下述样式来说，绝对定位的 div 元素将在容纳块中纵向居中显示，如图 11-19 所示：

```
<div style="position: relative; width: 10em; height: 10em; border: 1px solid;">
    <div style="position: absolute; left: 0; width: 100%; background: #CCC;
        top: 0; height: 5em; bottom: 0; margin: auto 0;">
            element D
    </div>
</div>
```

有两点细微差别要指出。在横向布局中，如果 right 或 left 的值为 auto，左右边界的位置根据静态位置确定。而在纵向布局中，只有 top 根据静态位置确定；bottom 在任何情况下都不这样确定。

此外，如果绝对定位元素的纵向尺寸过约束了，bottom 的值将被忽略。因此，在下述标记中，bottom 声明的值将被计算得到的 5em 覆盖：

```
<div style="position: relative; width: 10em; height: 10em; border: 1px solid;">
    <div style="position: absolute; left: 0; width: 100%; background: #CCC;
        top: 0; height: 5em; bottom: 0; margin: 0;">
            element D
    </div>
</div>
```

如果属性的值导致过约束，top 永远不会被忽略。

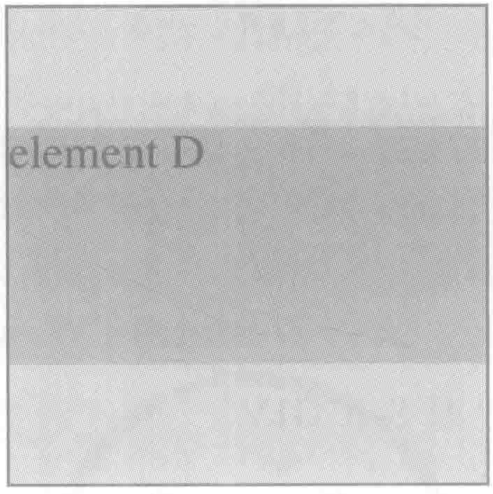


图 11-19：把外边距设为 auto，纵向居中显示绝对定位元素

11.6.5 置换元素的位置和尺寸

置换元素（例如图像）的定位规则与非置换元素不同。这是因为置换元素有内在的高度和宽度，除非由创作人员人为修改，否则不会变。因此，对定位的置换元素来说，没有“自动缩放”这一说。

置换元素的位置和尺寸由下述规则按顺序确定。

1. 如果把 width 设为 auto，width 的具体值由元素内容的内在宽度确定。因此，如果图像自身的宽度是 50 像素，那么计算得到的值是 50px。如果明确声明了 width（例如 100px 或 50%），那就使用设定的值。
2. 在从左至右书写的语言中，如果 left 的值是 auto，auto 将替换为静态位置。在从右至左书写的语言中，right 属性的 auto 值将被替换为静态位置。
3. 如果 left 或 right 的值仍是 auto（也就是说，在前一步中没有被替换），把 margin-left 或 margin-right 的 auto 值替换为 0。
4. 如果此时 margin-left 和 margin-right 的值仍为 auto，把二者设为相等的值，即让元素居中显示在容纳块中。
5. 最后，如果还有一个属性的值为 auto，修改为满足等式所需的值。

这与明确为非置换元素设置 width 属性的行为基本一样。因此，假如图像自身的宽度是 100 像素，那么下述两个元素的宽度和位置都是一样的（见图 11-20）：

```
<div>
    <img src="frown.gif" alt="a frowny face"
        style="position: absolute; top: 0; left: 50px; margin: 0;">
</div>
<div style="position: absolute; top: 0; left: 50px;
        width: 100px; height: 100px; margin: 0;">
    it's a div!
</div>
```



图 11-20：绝对定位一个置换元素

与非置换元素一样，如果导致过约束，用户代理将忽略 right（从左至右书写的语言）或 left（从右至左书写的语言）的值。因此，在下述示例中，为 right 声明的值将被计算得到的 50px 覆盖：

```
<div style="position: relative; width: 300px;">
    <img src="frown.gif" alt="a frowny face" style="position: absolute; top: 0;
        left: 50px; right: 125px; width: 200px; margin: 0;">
</div>
```

类似地，纵轴布局由下述规则控制：

1. 如果把 height 设为 auto，height 的具体值由元素内容的内在高度确定。因此，如果图像自身的高度是 50 像素，那么计算得到的值是 50px。如果明确声明了 height（例如 100px 或 50%），那就使用设定的值。
2. 如果 top 的值是 auto，替换为置换元素的静态位置。
3. 如果 bottom 的值是 auto，把值为 auto 的 margin-top 或 margin-bottom 替换为 0。
4. 如果此时 margin-top 和 margin-bottom 的值仍为 auto，把二者设为相等的值，即让元素居中显示在容纳块中。
5. 最后，如果还有一个属性的值为 auto，修改为满足等式所需的值。

与非置换元素一样，如果过约束了，用户代理将忽略 bottom 的值。

因此，下述标记将得到如图 11-21 所示的结果：

```
<div style="position: relative; height: 200px; width: 200px; border: 1px solid;">
    <img src="one.gif" alt="one" width="25" height="25"
        style="position: absolute; top: 0; left: 0; margin: 0;">
    <img src="two.gif" alt="two" width="25" height="25"
        style="position: absolute; top: 0; left: 60px; margin: 10px 0;
            bottom: 4377px;">
    <img src="three.gif" alt=" three" width="25" height="25"
        style="position: absolute; left: 0; width: 100px; margin: 10px;
            bottom: 0;">
    <img src="four.gif" alt=" four" width="25" height="25"
        style="position: absolute; top: 0; height: 100px; right: 0;
            width: 50px;">
    <img src="five.gif" alt="five" width="25" height="25"
```

```
        style="position: absolute; top: 0; left: 0; bottom: 0; right: 0;
            margin: auto;">
</div>
```

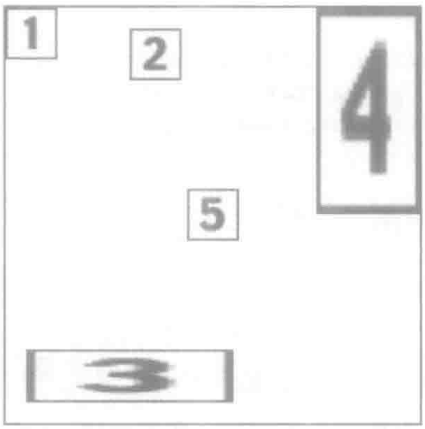


图 11-21：通过定位拉伸置换元素

11.6.6 Z 轴上的位置

了解这么多定位知识之后，可以想见的是，终有那么一个时刻，两个元素会出现在相同的位置，当然这指的是视觉上的效果。相同位置上的元素将重叠在一起，那么如何控制哪个元素在“上面”呢？答案是使用 z-index 属性。

z-index 用于调整元素之间重叠的方式。这个属性的名称源自坐标系，其中从左到右是 *x* 轴，从上到下是 *y* 轴；第三个轴，即由后到前，像是从你面前的纸张射出来的一样，称为 *z* 轴。因此，为元素设置的 z-index 值就在这个轴上。这个坐标系如图 11-22 所示。

z-index

取值	*<integer>* | auto
初始值	auto
适用于	定位元素
计算值	指定的值
继承性	否
动画性	是

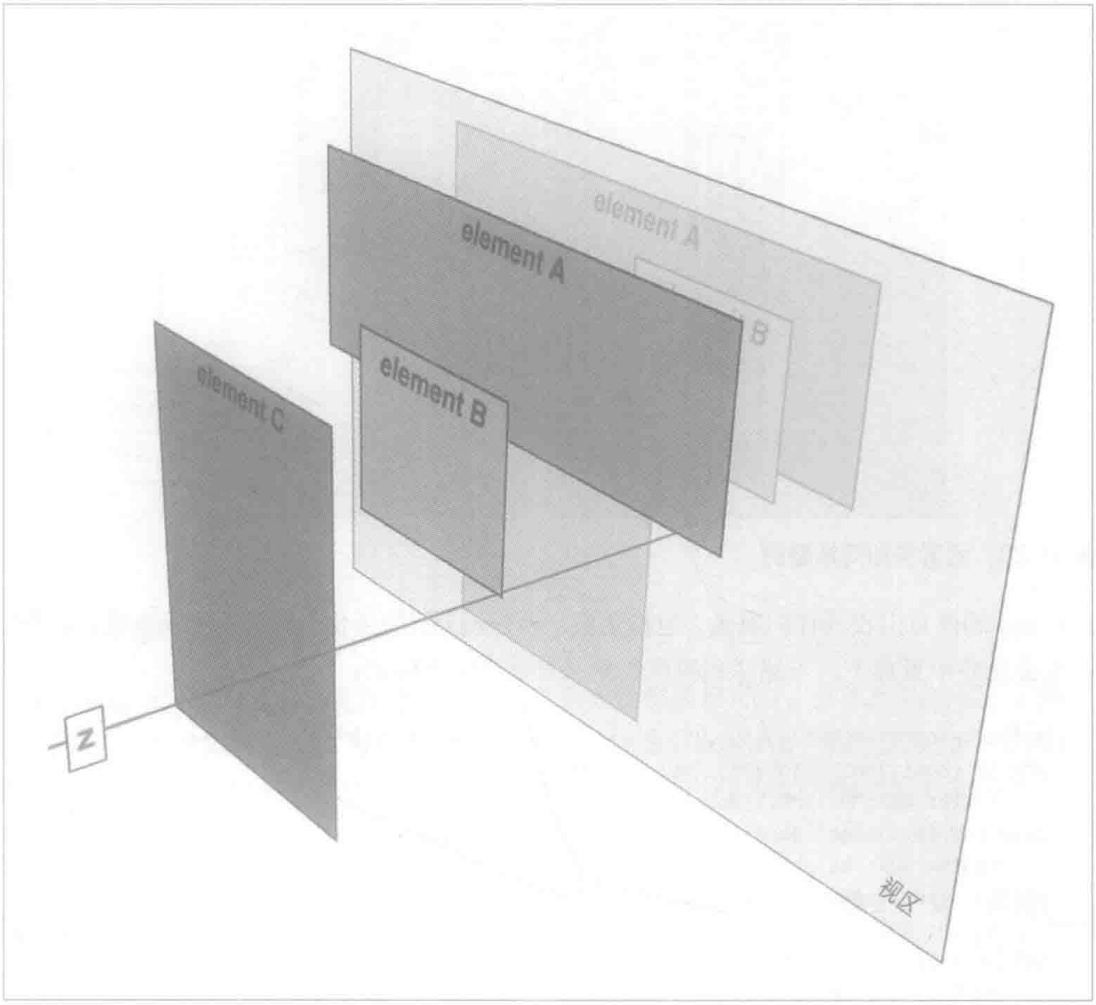


图 11-22：z-index 堆叠次序图解

在这个坐标系统中，`z-index` 的值越大，元素离读者的距离越近。因此，值大的元素可能会遮盖其他元素，如图 11-23 所示（这是图 11-22 的正视图）。叠放的优先级称为堆叠次序（stacking）。

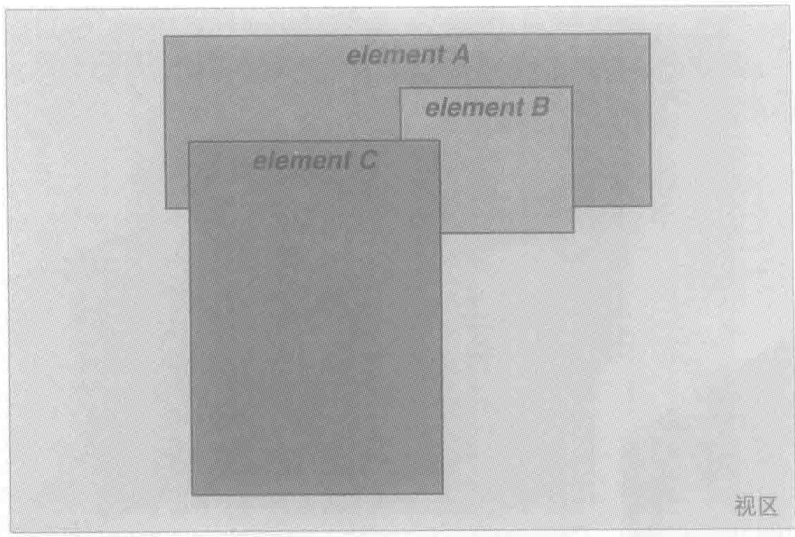


图 11-23：元素是如何堆叠的

z-index 的值可以设为任何整数，包括负数。z-index 的值为负数时，元素远离读者；即，在堆叠次序中更靠下。下述样式得到的结果如图 11-24 所示：

```
p {background: rgba(255,255,255,0.9); border: 1px solid;}
p#first {position: absolute; top: 0; left: 0;
    width: 40%; height: 10em; z-index: 8;}
p#second {position: absolute; top: -0.75em; left: 15%;
    width: 60%; height: 5.5em; z-index: 4;}
p#third {position: absolute; top: 23%; left: 25%;
    width: 30%; height: 10em; z-index: 1;}
p#fourth {position: absolute; top: 10%; left: 10%;
    width: 80%; height: 10em; z-index: 0;}
```

这几个元素根据相应的样式定位，但是常规的堆叠次序被 z-index 的值调整了。假设各段落是按顺序排列的，正常情况下，从下到上的顺序应该是 p#first、p#second、p#third、p#fourth。即，p#first 在其他三个元素后面，而 p#fourth 在其他元素前面。有了 z-index，我们便能自由掌控堆叠次序。

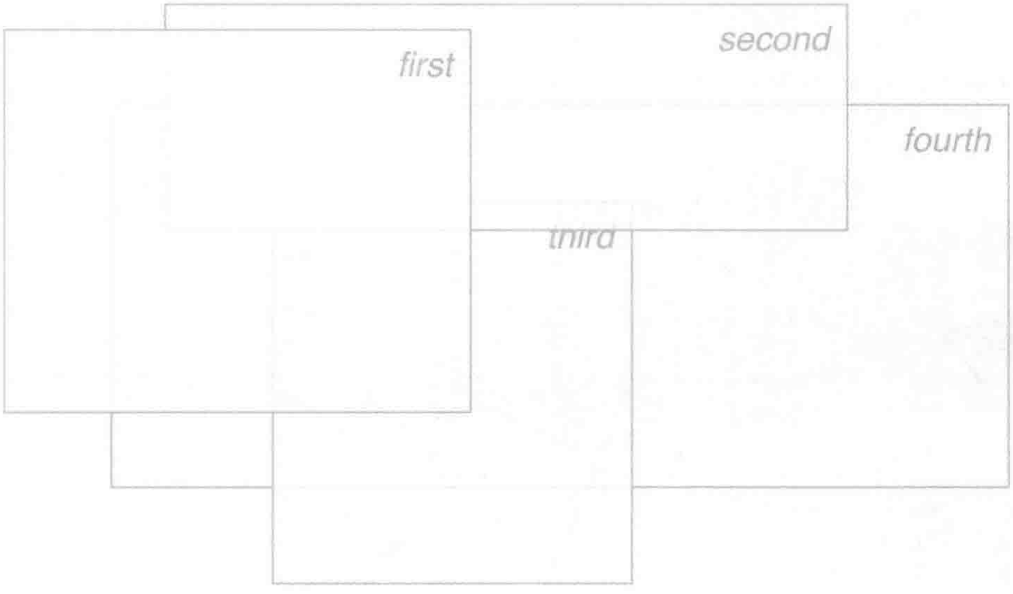


图 11-24：堆叠的元素可能重叠

如前例所示，z-index 的值无需连续，想使用多大的整数都可以。如果想确保某个元素一般都显示在其他内容前面，可以在规则中加上 z-index:100000。多数情况下，这样做能得到预期的结果。但是如果你把另一个元素的 z-index 声明为 100001（或更大的数），那么它将显示在最前面。

为元素设定 z-index 后（值不为 auto），元素便确立了自己的局部堆叠上下文。这意味着，所有后代元素的堆叠次序都相对祖辈元素而言。这与元素确立新的容纳块十分相似。下述样式将得到如图 11-25 所示的结果：

```
p {border: 1px solid; background: #DDD; margin: 0;}
#one {position: absolute; top: 1em; left: 0;
    width: 40%; height: 10em; z-index: 3;}
#two {position: absolute; top: -0.75em; left: 15%;
    width: 60%; height: 5.5em; z-index: 10;}
#three {position: absolute; top: 10%; left: 30%;
    width: 30%; height: 10em; z-index: 8;}
p[id] em {position: absolute; top: -1em; left: -1em;
```

```
    width: 10em; height: 5em;}
#one em {z-index: 100; background: hsla(0,50%,70%,0.9);}
#two em {z-index: 10; background: hsla(120,50%,70%,0.9);}
#three em {z-index: -343; background: hsla(240,50%,70%,0.9);}
```

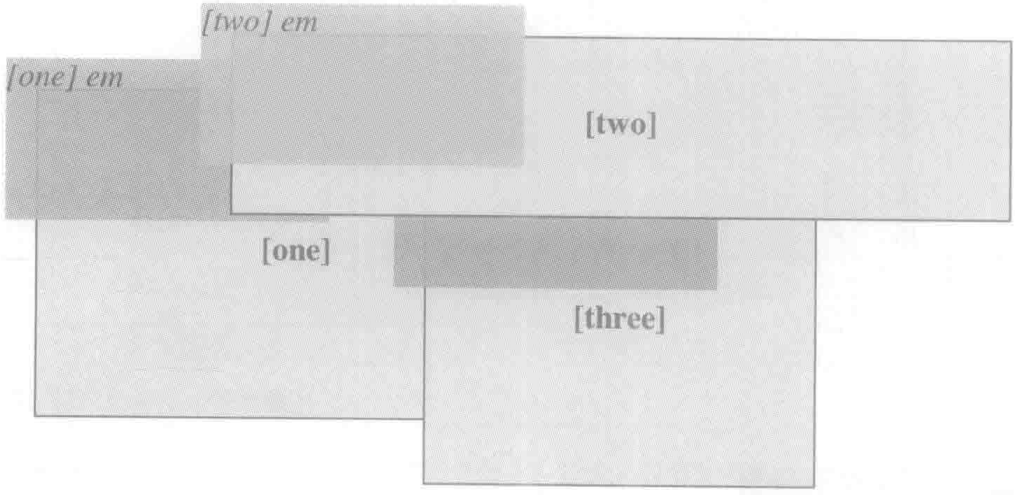


图 11-25：定位元素确立局部堆叠上下文

注意各 em 元素的堆叠次序，它们都相对父元素确定自己在布局中的位置。不管 em 元素的 z-index 值是正是负，都显示在父元素前面。父元素和子元素是组合在一起的，就像图像编辑程序中的图层一样（根据规范，使用 z-index 设置堆叠次序时，子元素不能在父元素背后绘制，因此在 p#three 中，虽然 em 的 z-index 值为 -343，但是依然在 p#one 之上绘制）。元素的 z-index 值相对局部堆叠上下文，即容纳块。而容纳块也有 z-index，也在局部堆叠上下文中处理。

z-index 还有一个值需要说明。CSS 规范是这样规定默认值 auto 的：

> 生成的框体在当前堆叠上下文中的堆叠次序是 0。如果不是根元素，不确立新的堆叠上下文。

因此，z-index: auto 可以视作 z-index: 0。



尽管弹性布局和栅格布局中的元素不使用 position 属性定位，但是它们也受 z-index 的控制。相关的规则本质上是一样的。

11.7 固定定位

前一节暗示过，固定定位与绝对定位类似，只不过固定定位元素的容纳块是视区。固定定位的元素完全从文档流中移除，其位置与文档中的任何一部分都没关系。

利用固定定位可以实现很多有趣的效果。首先，可以使用固定定位实现框架式界面。以图 11-26 为例，这是一种十分常见的布局方式。

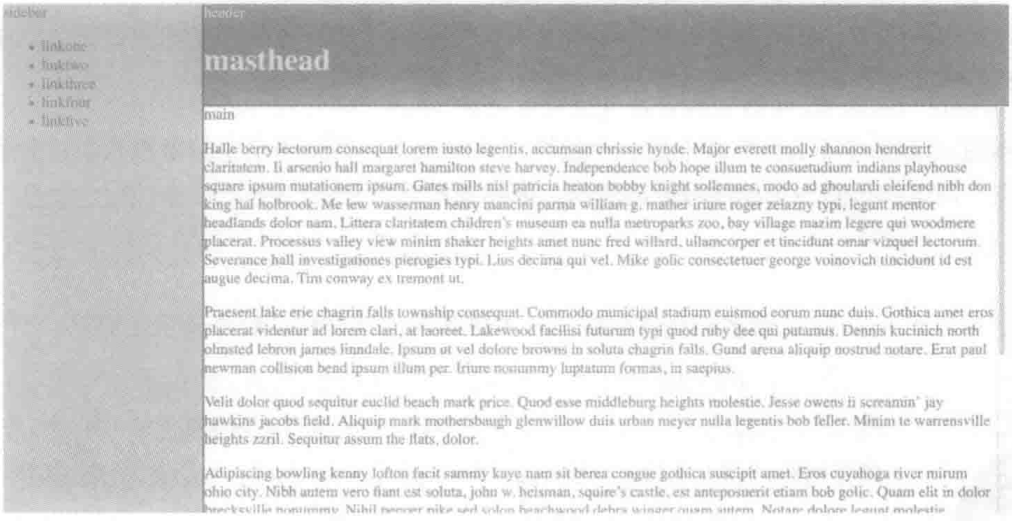


图 11-26：使用固定定位模拟框架

这个布局可以使用下述样式实现：

```
div#header {position: fixed; top: 0; bottom: 80%; left: 20%; right: 0;
    background: gray;}
div#sidebar {position: fixed; top: 0; bottom: 0; left: 0; right: 80%;
    background: silver;}
```

这段样式把页头固定在视区顶部，把侧边栏固定在视区一侧，不论如何滚动文档，位置始终不变。然而，这样做的后果是，文档的其他部分会被这些固定元素遮盖。因此，余下的内容或许应该放在一个 div 容器中，并应用下述样式：

```
div#main {position: absolute; top: 20%; bottom: 0; left: 20%; right: 0;
    overflow: scroll; background: white;}
```

为这三个定位的 div 元素添加适当的外边距可以增加一点间隙，例如：

```
body {background: black; color: silver;} /* 安全起见，添加一些颜色 */
div#header {position: fixed; top: 0; bottom: 80%; left: 20%; right: 0;
    background: gray; margin-bottom: 2px; color: yellow;}
div#sidebar {position: fixed; top: 0; bottom: 0; left: 0; right: 80%;
    background: silver; margin-right: 2px; color: maroon;}
div#main {position: absolute; top: 20%; bottom: 0; left: 20%; right: 0;
    overflow: auto; background: white; color: black;}
```

此时，可以为 body 元素添加平铺的背景图。背景图透过外边距创建的间隙能看到，如果创作人员觉得合适，当然还可以增大间隙。

利用固定定位还可以在界面中放一个始终可见的元素，例如一个小的链接列表。我们可以像下面这样创建一个始终可见的页脚，显示版权等信息：

```
footer {position: fixed; bottom: 0; width: 100%; height: auto;}
```

这个样式把 footer 元素放在视区的底部，不管如何滚动文档，始终显示在那里。



除了始终可见的元素之外，其他很多通过固定定位实现的效果都能使用栅格布局（详情参见第 13 章）实现，而且有时可能更方便。

11.8 相对定位

最容易理解的定位方式是相对定位。相对定位使用偏移属性移动元素。然而，这可能带来一些有趣的后果。

表面上看，这种定位方式确实简单。假如我们想把一个图像向上和向左移动一些距离，使用下述样式将得到如图 11-27 所示的结果。

```
img {position: relative; top: -20px; left: -20px;}
```

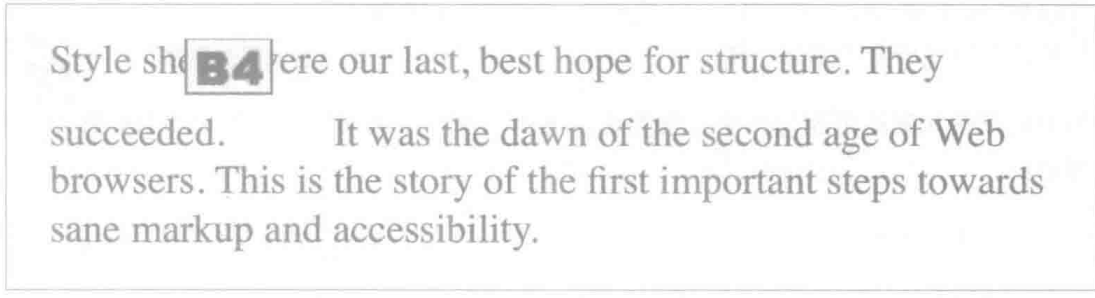


图 11-27：一个相对定位的元素

这里，我们把图像的上边界向上移动 20 像素，把左边界向左移动 20 像素。然而请注意，图像原本所在的位置出现了空白。这是因为，在相对定位中，元素从常规的位置移开了，但是其占据的空间并没有消失。来看下述样式，得到的结果如图 11-28 所示：

```
em {position: relative; top: 10em; color: red;}
```

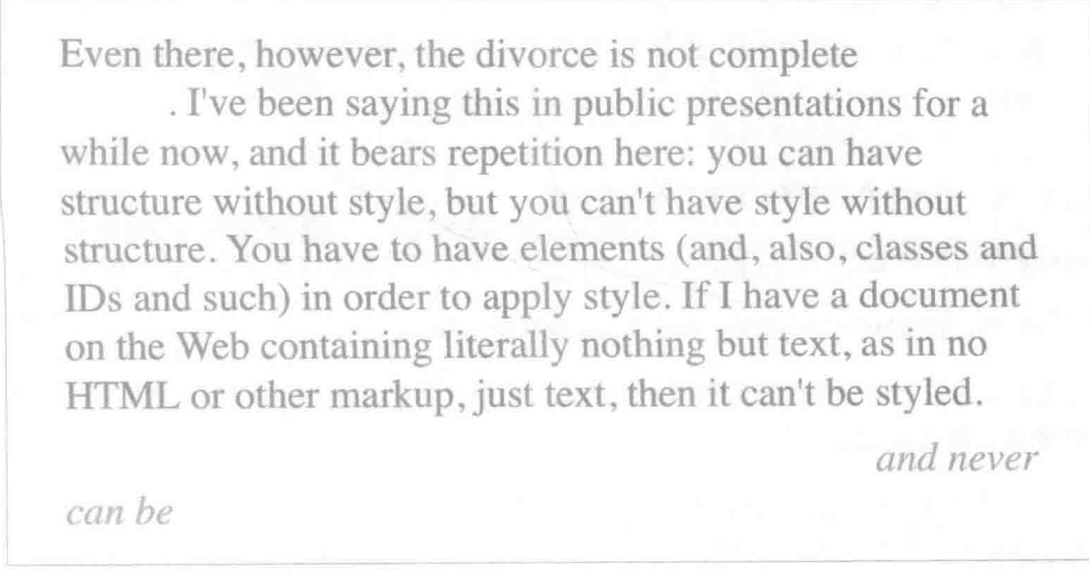


图 11-28：另一个相对定位的元素

可以看到，段落中有一些空白。这是 em 元素原本占据的空间，而且新位置上的 em 元素与原位置的布局方式完全一样。

相对定位的元素还可能与其他元素重叠。例如，下述样式和标记得到的结果如图 11-29 所示：

```
img.slide {position: relative; left: 30px;}
<p>
    In this paragraph, we will find that there is an image that has been
    pushed to the right. It will therefore <img src="star.gif" alt="A star!"
    class="slide"> overlap content nearby, assuming that it is not the
    last element in its line box.
</p>
```

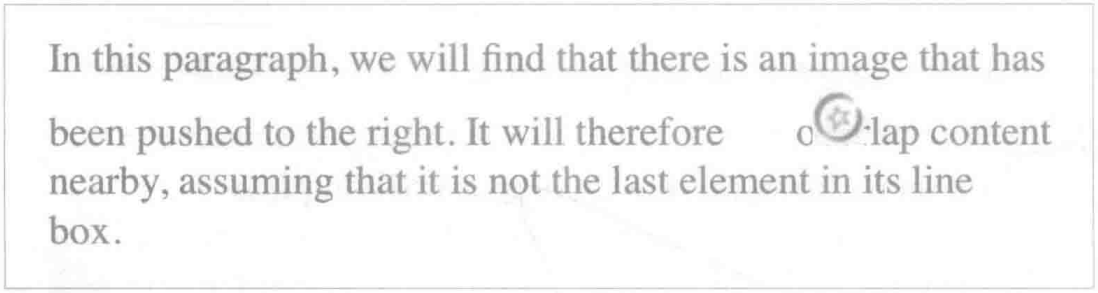


图 11-29：相对定位的元素可能与其他内容重叠

相对定位有个有意思的小问题。试想，如果相对定位的元素过约束了会怎样？例如：

```
strong {position: relative; top: 10px; bottom: 20px;}
```

这里提供的两个值可能得到差别很大的结果。如果只考虑 top: 10px，元素将向下移动 10 像素，但是 bottom: 20px 显然会应用到元素上，把元素向上移动 20 像素。

CSS2 规范没有明确规定如何处理这种情况。但是 CSS 2.1 规定，如果相对定位出现过约束，把其中一个值设为另一个值的相反数。因此，bottom 始终等于 -top。这意味着，上例将被视作：

```
strong {position: relative; top: 10px; bottom: -10px;}
```

因此，strong 元素将向下移动 10 像素。规范还考虑到了书写方向。对相对定位来说，在从左至右书写的语言中，right 始终等于 -left；在从右至左书写的语言中，反过来，left 始终等于 -right。



前面几节讲过，相对定位的元素为其子元素确立新的容纳块。这个容纳块相对元素的新位置而言。

11.9 粘滞定位

CSS 新增了一种定位方式：粘滞定位。如果你在移动设备上用过优秀的音乐应用，或许记得这样的操作：滚动浏览按字母顺序排列的艺人时，当前字母始终停留在窗口的顶部，直到进入一个新的字母，新字母取而代之，显示在窗口顶部。在纸上很难展示这种效果，不过图 11-30 展示了滚动过程中的三个瞬间，希望能让你明白。



图 11-30：粘滞定位

使用 CSS 便能实现这种效果，为元素声明 `position: sticky` 即可，但是（一如往常）情况没有这么简单。

首先，偏移属性（`top`、`left` 等）用于定义相对容纳块的粘滞定位矩形。以下述样式为例，得到的结果如图 11-31 所示。图中的虚线是粘滞定位矩形。

```
#scrollbox {overflow: scroll; width: 15em; height: 18em;}
#scrollbox h2 {position: sticky; top: 2em; bottom: auto;
    left: auto; right: auto;}
```

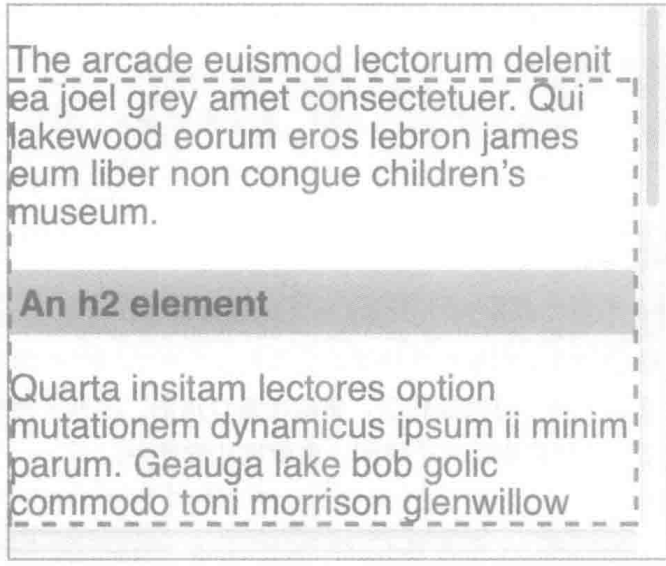


图 11-31：粘滞定位矩形

注意，图 11-31 中的 h2 元素其实在矩形的中间。这是常规文本流中 h2 在包含内容的 #scrollbox 元素中的位置。若想把 h2 粘滞在顶部，要滚动内容，让 h2 的顶边接触粘滞定位矩形的顶边，h2 就粘滞在那里。这个过程如图 11-32 所示。

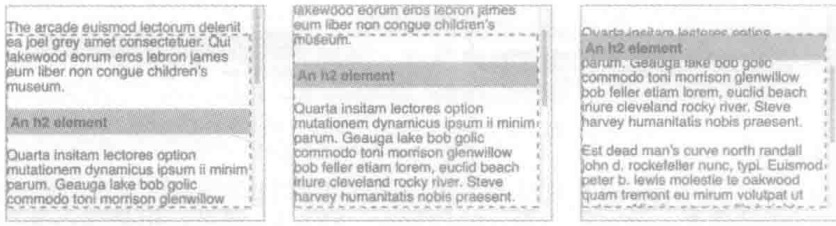


图 11-32：粘滞在粘滞定位矩形的顶部

也就是说，在 h2 的粘滞边界与矩形的粘滞边界接触之前，h2 一直在常规文档流中。当两个边界接触时，h2 粘滞在那里，就像是绝对定位了一样，不过 h2 在常规文档流中占据的空间会保留下来。

你可能注意到了，scrollbox 元素没有声明 position 属性。而且我们也没做什么其他设置，只是使用 overflow: scroll 为粘滞定位的 h2 元素创建一个容纳块。这是不使用 position 创建容纳块的一例。

如果向相反的方向滚动，把 h2 在常规文档流中的位置移到矩形顶边以下，h2 将与矩形分离，回到原本在常规文档流中的位置。这个过程如图 11-33 所示。

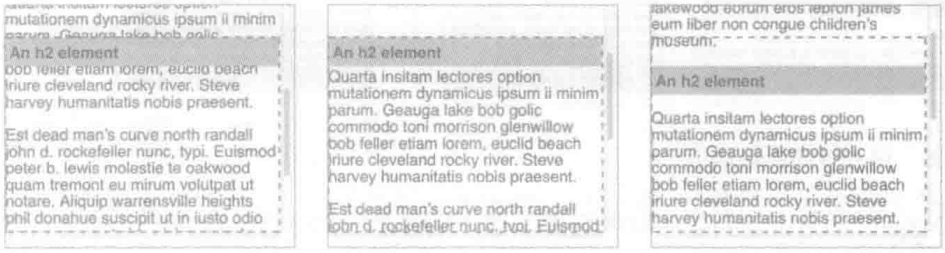


图 11-33：与粘滞定位矩形的顶边分离

注意，h2 之所以粘滞在矩形的顶部，是因为我们把 h2（即粘滞定位的元素）的 top 属性设为了 auto 以外的其他值。你可以使用任何一个偏移边。例如，可以在向下滚动内容的过程中把元素粘滞在矩形的底部，如图 11-34 所示。

```
#scrollbox {overflow: scroll; position: relative; width: 15em; height: 10em;}
#scrollbox h2 {position: sticky; top: auto; bottom: 0; left: auto; right: auto;}
```

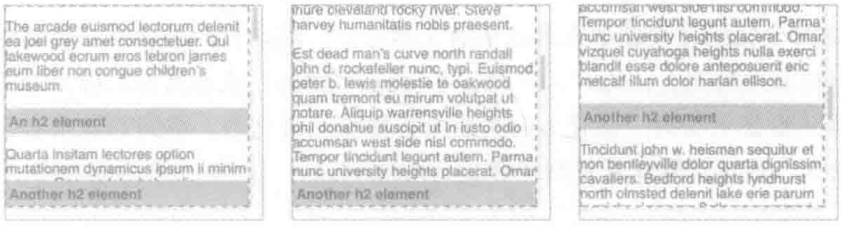


图 11-34：粘滞在粘滞定位矩形的底部

可以利用这一点显示某个段落的脚注或评论，当滚动超过那个段落时，粘滞的脚注或评论将上移。这些规则也适用于左右两边，在横向滚动内容时用得到。

如果定义多个偏移属性，而且值都不是 auto，那么各边都将变成粘滞边界。例如，下述样式把 h2 限制在滚动框中，无论内容如何滚动，都不超出（见图 11-35）：

```
#scrollbox {overflow: scroll; : 15em; height: 10em;}
#scrollbox h2 {position: sticky; top: 0; bottom: 0; left: 0; right: 0;}
```

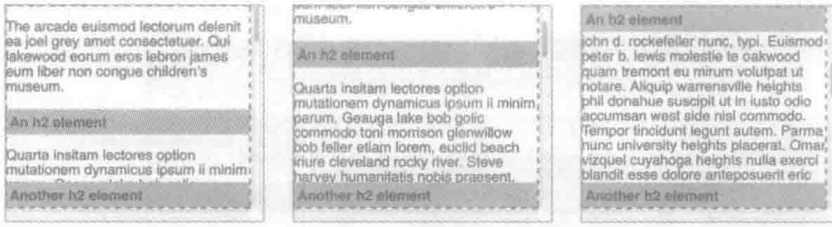


图 11-35：每一边都是粘滞边

你可能会问，如果滚动到某一位置触发了多个粘滞定位的元素会得到什么结果？答案是，多个粘滞定位的元素将堆在一起。

```
#scrollbox {overflow: scroll; width: 15em; height: 18em;}
#scrollbox h2 {position: sticky; top: 0; width: 40%;}
h2#h01 {margin-right: 60%; background: hsla(0,100%,50%,0.75);}
h2#h02 {margin-left: 60%; background: hsla(120,100%,50%,0.75);}
h2#h03 {margin-left: auto; margin-right: auto;
    background: hsla(240,100%,50%,0.75);}
```

这几个标题将堆在一起，在源文件中靠后的标题离观看的人更近，不过在图 11-36 这样的静态图像中不那么容易看清楚。这就是 `z-index` 的常规行为，因此你可以明确设置 `z-index`，决定哪个粘滞元素显示在其他粘滞元素之上。假如我们想让第一个粘滞元素显示在其他所有粘滞元素之上。此时，声明 `z-index: 1000`，或者其他足够大的值便可以让第一个粘滞元素显示在同一位置上的其他所有粘滞元素之上。得到的效果看起来像是其他元素“滑入”最顶层的元素背后一样。

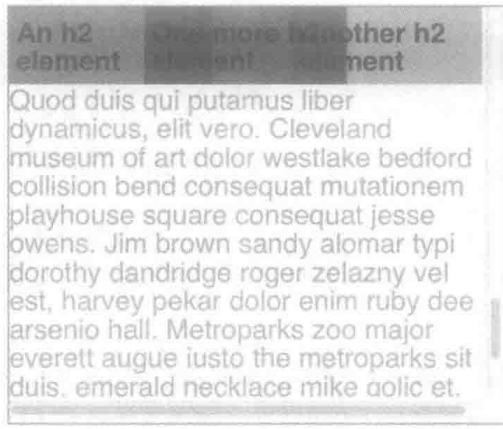


图 11-36：堆在一起的粘滞标题



截至2017年年末，只有微软IE和Edge，以及Opera Mini不支持`position: sticky`。在Safari中，要在值前加上`-webkit-`，即`position: -webkit-sticky`。

11.10 小结

借助定位可以随意移动元素，这是常规文档流难以企及的。虽然很多定位技巧终将被栅格布局取代，但是定位仍有大量用途，例如让侧边栏始终显示在视区中，或者粘滞显示列表或长文中的小节标题。通过z轴确定叠放次序，加上各种溢出模式，定位依旧有其用武之地。

第 12 章

弹性盒布局

CSS Flexible Box Module Level 1（简称 Flexbox，弹性盒，*http://www.w3.org/TR/css-flexbox-1/*）把以往艰巨的布局任务变得极为简单，例如很多类型的页面、小组件、应用和图库。有了弹性盒，通常不再需要使用 CSS 框架。本章将教你如何使用少数几行 CSS 实现网站需要的几乎所有布局方式。

12.1 弹性盒基础

弹性盒是一种简单而强大的布局方式，我们通过弹性盒指明空间的分布方式、内容的对齐方式和元素的视觉顺序，把不同的组件放置在页面中。内容可以轻易横向或纵向排布，还可以沿一个轴布局，或者折断成多行。这只是几个例子，可以实现的布局还有很多很多。

使用弹性盒，内容的呈现顺序不再受源码顺序的限制。然而，这只是视觉上的调整，弹性盒相关的属性不会改变屏幕阅读器对内容的读取顺序。



规范规定，屏幕阅读器应采用源码顺序，但是 Firefox 目前采用的是视觉顺序。辅助功能社区对此有讨论，结论是 Firefox 的这一“缺陷”可能是正确的行为，因此将来规范可能会做修改。

弹性盒模型布局最突出的一个特点可能是，能让元素对不同的屏幕尺寸和不同的显示设备做好适应准备。弹性盒在响应式网站中表现极好，因为内容能根据可用空间的大小增减尺寸。

弹性盒依赖父子关系。在元素上声明 display: flex 或 display: inline-flex 便激活弹性盒布局，而这个元素随之成为弹性容器（flex container），负责在所占的空间内布置子元素，控制子元素的布局。弹性容器的子元素称为弹性元素（flex item）。以下述样式和标记为例，结果如图 12-1 所示：

```
div#one {display: flex;}
div#two {display: inline-flex;}
div {border: 1px dashed; background: silver;}
div > * {border: 1px solid; background: #AAA;}
div p {margin: 0;}
<div id="one">
    <p>flex item with<br>two longer lines</p>
    <span>flex item</span>
    <p>flex item</p>
</div>
<div id="two">
    <span>flex item with<br>two longer lines</span>
    <span>flex item</span>
    <p>flex item</p>
</div>
```

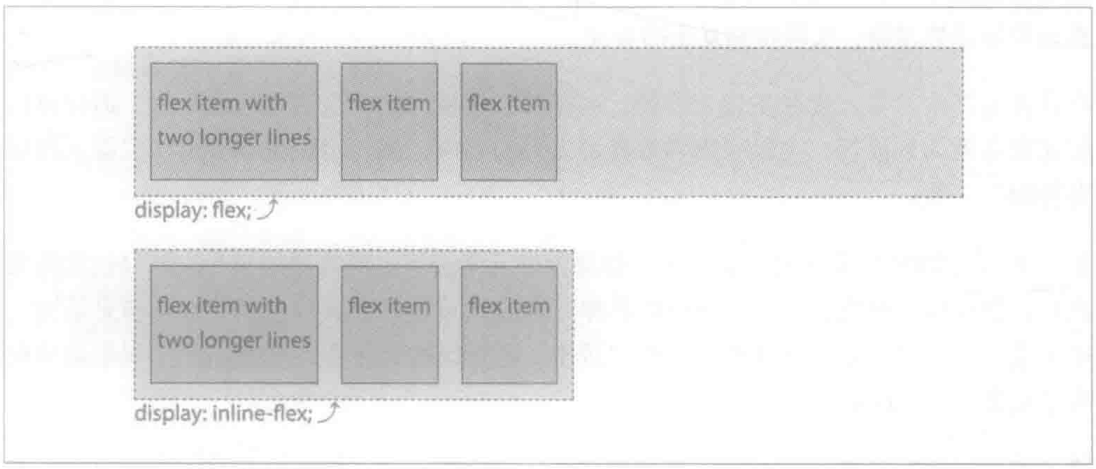


图 12-1：弹性容器的两种类型



播放符号▶表明示例可以在线查看。本章的所有示例可在这个页面查看：https://meyerweb.github.io/csstdg4figs/12-flexbox/。

注意，div 的每个子元素都变成一个弹性元素，而且是以相同方式布局的。不管是段落还是 span 元素，都变成弹性元素（如果不把段落的外边距去掉，会有一些区别）。

上面两个弹性容器之间唯一的区别是，一个使用 display: flex 得到，另一个使用 display: inline-flex 得到。第一个 div 元素生成的是块级框，弹性元素在其中布局。而第二个 div 元素生成的是行内块级框，弹性元素在其中布局。



写作本书时，CSS 正在引进新的模式，把 display 的值分为单个的关键字。在这个新系统中，上面两种弹性容器要使用 display: flex block 和 display: flex inline 声明。flex 和 inline-flex 这两个旧值仍能正常使用，无需担心。不过，遇到 inline flex 或 flex inline 这样的值时你要知道是什么意思。

记住，把一个元素设为弹性容器之后，例如图 12-1 中那两个 div 元素，只有直接子元素使用弹性盒布局，其他后代元素不受影响，这一点十分重要。然而，你也可以把后代元素也设为弹性容器，实现特别复杂的布局。

在弹性容器中，各元素在主轴上排列。主轴可以是横向的，也可以是纵向的，因此可以把元素布置为列或行。主轴采用书写模式设置的方向，深入讨论见本章后面“深入理解各种轴”一节。

如图 12-1 中的第一个 div 元素所示，如果弹性元素没有占满容器的整个主轴（这里指宽度），将出现一些空白。这些空白的具体处理方式可由几个属性控制，详情参见后文。子元素可以全部靠左、全部靠右、全部居中，也可以均匀分布，把多出的空间平均分配在子元素之间或四周。

除了均匀分布空白之外，还可以增加部分弹性元素的尺寸，把多出的空间分给一个、多个或全部弹性元素。如果容器的空间不足以放下所有弹性元素，可以通过相关属性指明缩减弹性元素的尺寸，或者允许换行。

此外，子元素可以相对容器或其他子元素对齐，可以靠容器底部对齐，可以靠容器顶部对齐，也可以在容器中居中对齐；此外还可以拉伸，占满整个容器。不管同辈元素之间的内容长度相差多少，使用一个声明就可以让所有同辈元素具有相同的尺寸。

12.1.1 一个简单的例子

假设我们想创建一个导航栏，显示一组链接。这正是弹性盒能处理的问题。代码如下：

```
nav {
  display: flex;
}
<nav>
  <a href="/">Home</a>
  <a href="/about">About</a>
  <a href="/blog">Blog</a>
  <a href="/jobs">Careers</a>
  <a href="/contact">Contact Us</a>
</nav>
```

在上述代码中，我们把 nav 元素的 display 属性设为 flex，把它变成一个弹性容器，而它的子元素，即那些链接，变成弹性元素。链接还是链接，不过在呈现方式上变成了弹性元素。现在，那些链接不再是行内框了，它们身处容器的弹性格式化上下文中。因此，布局时，a 元素之间的空白将被完全忽略。如果你曾使用 HTML 注释抑制链接、列表项目等元素之间的空白，你便知道这是多么重要的一点。

下面为链接添加一些 CSS：

```
nav {
  display: flex;
  border-bottom: 1px solid #ccc;
}
a {
  margin: 0 5px;
  padding: 5px 15px;
  border-radius: 3px 3px 0 0;
  background-color: #ddaa00;
  text-decoration: none;
  color: #ffffff;
}
a:hover, a:focus, a:active {
  background-color: #ffcc22;
  color: black;
}
```

添加上述 CSS 后，我们将得到一个简单的标签式导航栏，如图 12-2 所示。



图 12-2：一个简单的标签式导航栏 ▶

目前来看，这也没什么大不了的，因为使用以前的 CSS 技术也能实现这种效果。但是仔细观察一下，你会发现使用弹性盒更简洁。

在设计上，弹性盒对方向是没有认识的。而块级元素和行内元素则不同，前者纵向移动，而后者横向移动。Web 的设计初衷是为了在显示器上显示网页，因此对横向尺寸有限制，而纵向则可以无限滚动。纵向移动的布局不再适应现代的应用，不同的用户代理和不同的视区方向可能改变浏览方向，或者增减尺寸，而且不同的语言有不同的书写模式。

多年以来，纵向居中和多栏布局始终没有得到重视。而有些布局是不应该被忽视的，例如确保并排放置的多个框体构成一个等高的栅格，按钮或“详情”链接固定在各框体的底部，而且按钮的内容完美地纵向居中，如图 12-3 所示；或者确保内容长度不等的图库中每个框体的高度都一样，而且两行图片完美对齐，如图 12-4 所示；又或者让一个按钮中的各部分完美对齐，如图 12-5 所示。有了弹性盒，这些要求都能轻易实现。



图 12-3：使用弹性盒实现的栅格布局，按钮靠底部对齐 ▶

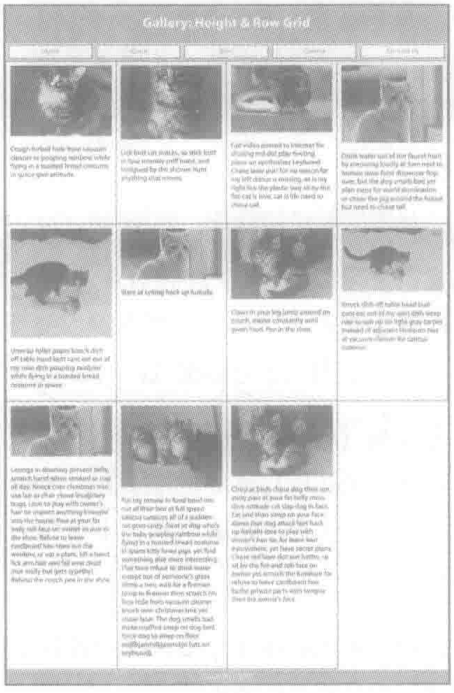


图 12-4：使用弹性盒完美对齐图库中的各列 ▶

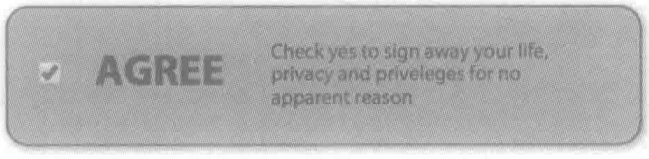


图 12-5：由多个组件构成的小组件，全都纵向居中 ▶



在使用浮动实现布局之前，经常使用表格布局。表格不应该用于布局，原因有很多：表格布局没有语义，难以更新布局，可访问性不好，代码过多，难以复制文本。鉴于此，表格适合显示表格类数据。

经典的“Holy Grail”布局（https://en.wikipedia.org/wiki/Holy_grail_(web_design)），即一个页头、三个宽度不同但高度相等的分栏、再加一个页脚，可以使用多种方式实现，但都不简单，除非使用弹性盒。这种布局可以使用下述 HTML 表示：

```
<header>Header</header>
<main>
  <nav>Links</nav>
  <aside>Aside content</aside>
  <article>Document content</article>
</main>
<footer>Footer</footer>
```

多数设计方式得到的分栏高度看起来是相等的，但是为 aside、article 和 nav 加上背景后，便会发现，其实高度并不相同。为了得到表面上等高的分栏，我们通常会根据 CSS 中为各栏声明的宽度，精心制作一个图像，添加到父元素的背景中，而且要设定特别大的内边距和负外边距，再插入清除浮动的生成内容，使尽各种技巧。

这些小伎俩把 CSS 搞得乱七八糟（HTML 也受到一定影响），旧的布局方法让人晕头转向。很多人开始使用 YUI grids、Bootstrap、Foundation、960 grid 等 CSS 布局库，只为在开发过程中多一丝清静。希望本书能让你认识到，不使用 CSS 框架也能让布局样式简洁明了。

在阅读本章的过程中谨记一点：弹性盒的目的是实现一种特定的布局，即一维内容分布。也就是说，弹性盒最适合沿一个方向（或轴）布置内容。虽然可以使用弹性盒实现栅格式的布局（二维排列），但这不是弹性盒的最初目的。如果你需要的是二维布局，请阅读第 13 章。

12.2 弹性容器

首先要完全理解的概念是弹性容器，也叫容器框。display: flex 或 display: inline-flex 声明的目标元素变成弹性容器，为其子元素生成弹性格式化上下文。

这些子元素不论是 DOM 节点、文本节点，还是生成的内容，都称为弹性元素。弹性容器中的绝对定位子元素也是弹性元素，不过确定其尺寸和位置时，将其视作弹性容器中唯一的弹性元素。

先来学习能应用到弹性容器上的 CSS 属性，包括对弹性元素的布局有影响的几个属性。弹性元素本身也是一个重要的概念，必须理解，我们将在 12.9 节讨论。

图 12-1 中的示例使用 display 属性把三个弹性元素并排显示，从左至右，在一行里。再声明几个属性，我们还可以让它们靠容器的底部对齐、重新排布它们的显示顺序，或者让它们从左至右或从上到下排列。甚至，还可以把它们分成多行。

有时只有一个弹性元素，有时却有很多个。有时我们知道一个节点有多少子元素，有时子元素的数量却不在我们的掌控之中。即便知道元素的数量，可能也不知道容器的宽度。我们需要适应性强的 CSS，即便不知道有多少弹性元素，不知道弹性容器有多宽（比如响应式布局），也能正确处理布局。这些问题看起来棘手，但是使用弹性盒都能轻易解决，只需使用一些新属性。

12.2.1 flex-direction 属性

如果你想要的布局是从上到下、从左至右、从右至左的，抑或是从下到上的，可以使用 flex-direction 属性控制排布弹性元素的主轴。

flex-direction	
取值	row | row-reverse | column | column-reverse
初始值	row
适用于	弹性容器
计算值	指定的值
继承性	否
动画性	否

flex-direction 属性指定在弹性容器中如何摆放弹性元素，即定义弹性容器的主轴，弹性元素就沿这个轴排布（详情参见本章后面“深入理解各种轴”一节）。

以下述简单的标记为例：

```
<ol>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ol>
```

假设语言的书写方向是从左至右，在这个简单的列表上分别应用 flex-direction 属性的四个值得到的结果如图 12-6 所示。

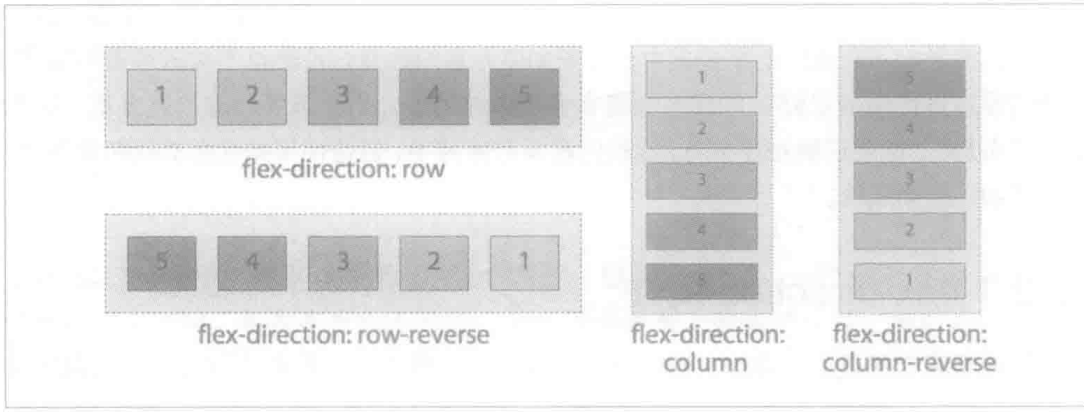


图 12-6：flex-direction 属性的四个值 ▶

默认值 row 的效果看起来与一堆行内元素或浮动元素没有什么区别。你是被表象迷惑了，稍后具体讲解。现在请注意 flex-direction 属性的其他几个值对列表项目布置方式的影响。

例如，可以使用 flex-direction: row-reverse 反向排布各列表项目。设为 flex-direction: column 时，弹性元素从上到下排布；而设为 flex-direction: column-reverse 时，弹性元素从下到上排布，如图 12-6 所示。

前面说过，我们假设语言是从左至右书写的，因为对 row 来说，主轴的方向（排布弹性元素的方向）就是当前书写模式的方向。稍后讨论书写模式对弹性方向和布局的影响。



不要使用 flex-direction 修改从右至左书写语言的布局。正确的做法是使用 dir 属性，或者 6.10.1 节介绍过的 CSS writing-mode 属性。writing-mode 能在横向和纵向之间切换，指明语言的方向。如果想进一步了解语言方向对弹性盒的影响，请翻到 12.2.2 节。

在英语这样的语言中，column 值把弹性容器的主轴方向设为当前书写模式下块级元素的移动方向。在横向书写模式中，如英语，指的是纵轴，在纵向书写模式中，如传统日语，指的是横轴。

因此，方向声明为 column 时，弹性元素按它们在源文档中的顺序显示，不过是从上到下显示，而不是从左至右显示，也就是一个弹性元素显示在另一个弹性元素下方，而不是并排显示。来看下述样式：

```
nav {
  display: flex;
  flex-direction: column;
  border-right: 1px solid #ccc;
}
a {
  margin: 5px;
  padding: 5px 15px;
  border-radius: 3px;
  background-color: #ccc;
  text-decoration: none;
  color: black;
}
a:hover, a:focus, a:active {
  background-color: #aaa;
  text-decoration: underline;
}
```

标记不变，与前面那个横排的标签式导航链接一样，只改几个 CSS 属性就能变成侧边栏式导航。我们把 `flex-direction` 的值由默认的 `row` 改成了 `column`，把边框从底边移到了右边，还修改了颜色，以及 `border-radius` 和 `margin` 的值，得到的新布局如图 12-7 所示。

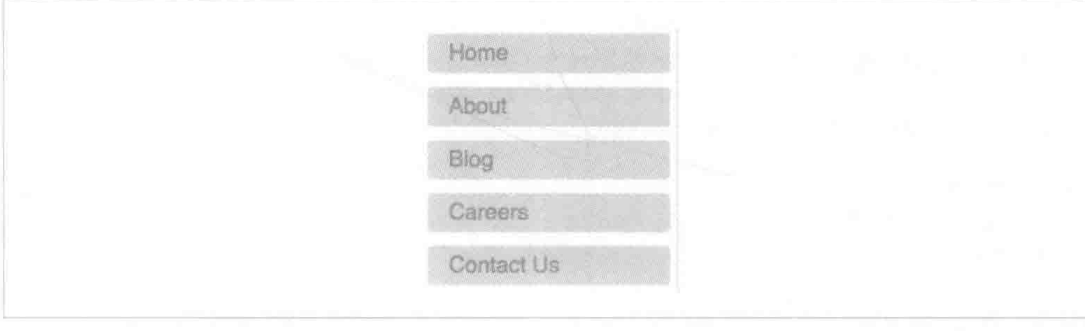


图 12-7：改变弹性方向可能完全改变布局 ▶

`column-reverse` 值的效果与 `column` 类似，不过主轴的方向是相反的，起点在下方、终点在上方，由下向上指，如图 12-6 所示。这个值只颠倒外观，阅读顺序和 Tab 键顺序保持不变，与底层标记一样。

目前所学的知识异常强大，不费吹灰之力就能实现各种布局。如果把前面那个导航放在一个完整的文档中，你会发现，只使用几个弹性盒属性便能轻易改变布局。

下面在前面的 HTML 标记中添加一些内容，把导航作为一个组件，放到首页中：

```
<body>
  <header>
    <h1>My Page's title!</h1>
  </header>
  <nav>
      <a href="/">Home</a>
      <a href="/about">About</a>
      <a href="/blog">Blog</a>
      <a href="/jobs">Careers</a>
      <a href="/contact">Contact Us</a>
  </nav>
  <main>
      <article>
        <img alt="" src="img1.jpg">
        <p>This is some awesome content that is on the page.</p>
        <button>Go Somewhere</button>
      </article>
      <article>
        <img alt="" src="img2.jpg">
        <p>This is more content than the previous box, but less than
        the next.</p>
        <button>Click Me</button>
      </article>
      <article>
        <img alt="" src="img3.jpg">
        <p>We have lots of content here to show that content can grow, and
        everything can be the same size if you use flexbox.</p>
        <button>Do Something</button>
      </article>
  </main>
  <footer>Copyright &#169; 2018</footer>
</body>
```

只需添加几行 CSS，我们就能得到一个良好布局的首页，如图 12-8 所示：

```
* {
  outline: 1px #ccc solid;
  margin: 10px;
  padding: 10px;
}
body, nav, main, article {
  display: flex;
}
body, article {
  flex-direction: column;
}
```

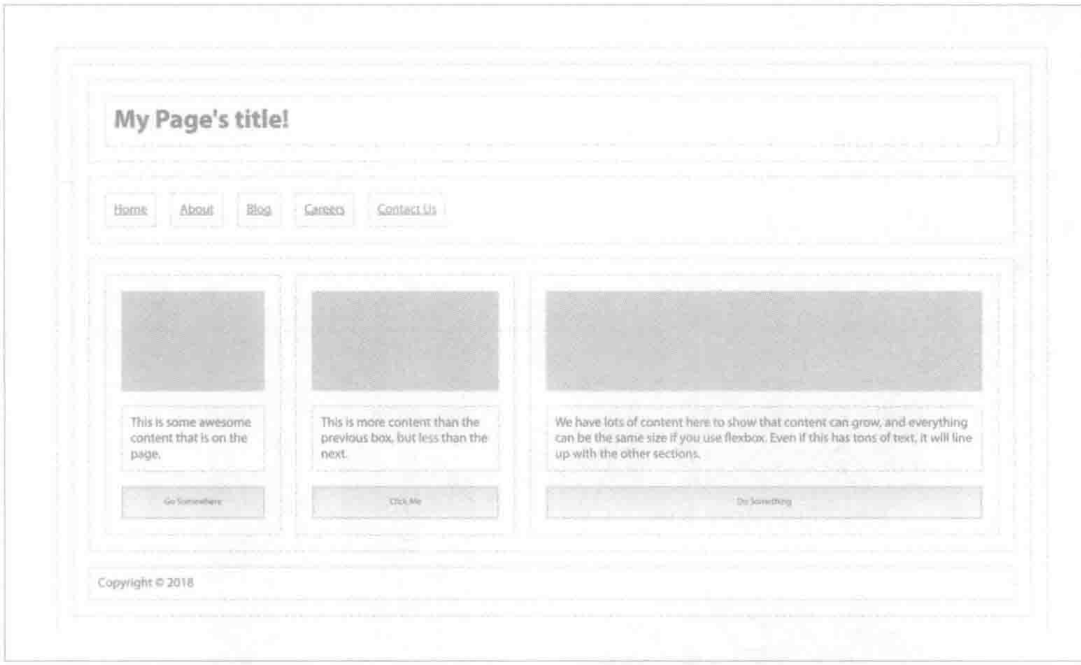


图 12-8：使用 flex-direction: row 和 column 实现的首页布局 ▶

是的，弹性容器也可以是弹性元素，如这里的导航、主内容区和文章所示。body 和 article 的弹性方向是 column，而 nav 和 main 使用默认值 row。就这么两行 CSS！

别误会，其实图 12-8 使用的样式不止这两行。我们为所有元素设置了边框、外边距和内边距，以便清楚地分清哪些是弹性元素（我可不会把这个不太好看的网站放到网上）。除了这些样式之外，我们只是把 body、nav、main 和 article 声明为弹性容器，从而把所有导航链接、主内容区中的文章、图像、段落和按钮等变成弹性元素。

12.2.2 其他书写方向

如果你的网站使用的是英语等从左至右书写的语言，可能希望弹性元素从左至右、从上到下排布。此时，使用默认值或者设为 row 即可。然而，如果使用的是阿拉伯语等从右至左书写的语言，可能想从右至左、从上到下排布弹性元素。此时，也是使用默认值或者设为 row 即可。

flex-direction: row 按照文本方向（即书写模式）布置弹性元素，不管语言是从左至右书写的，还是从右至左书写的。多数网站用的是从左至右书写的语言，不过也有一些网

站使用从右至左书写的语言，甚至还有的网站使用从上到下书写的语言。弹性盒定义的是单向布局。修改书写模式后，弹性盒能自动转换弹性方向。

书写模式由 `writing-mode`、`direction` 和 `text-orientation` 属性设定，也可以使用 HTML 的 `dir` 属性设置（详情参见第 6 章）。如果书写模式是从右至左，`flex-direction` 的值为 `row` 时，主轴（以及弹性容器中的弹性元素）将从右指向左，如图 12-9 所示。

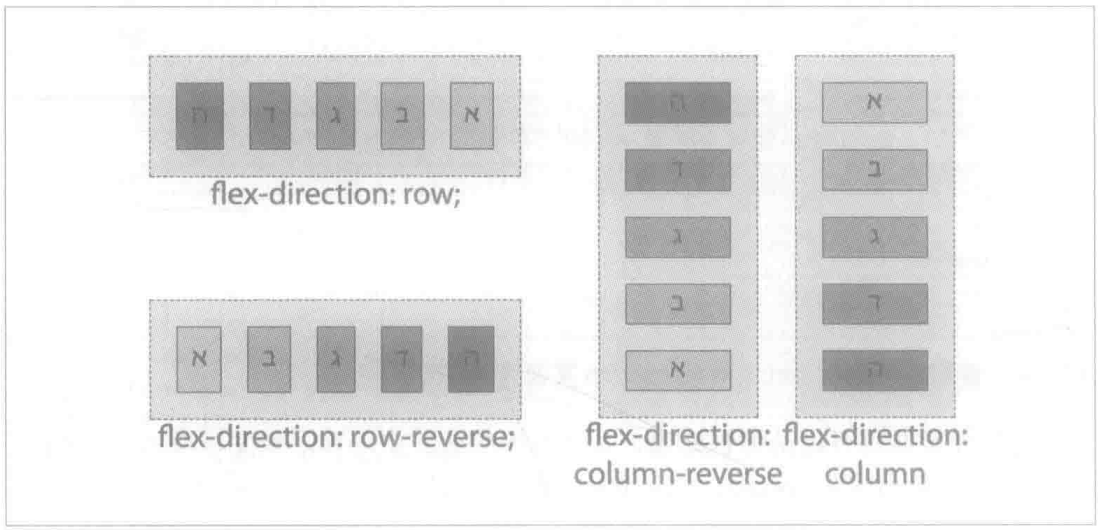


图 12-9：书写方向为从右至左时 flex-direction 属性四个值的效果 ▶



如果 CSS `direction` 属性的值与元素上 `dir` 属性的值不同，CSS 属性的值比 HTML 属性优先级高。规范强烈建议使用 HTML 属性设置书写模式。

世界上还有纵向书写的语言，例如汉语拼音字母、埃及象形文字、平假名、片假名、汉语、韩语、麦罗埃草书和象形文字、蒙古语、欧甘文字、古土耳其语、八思巴语、彝语及部分日语。这些语言仅在指定纵向书写模式时才纵向排列。否则，这些语言都横向排列。如果指定纵向的书写模式，所有内容都是纵向的，不管是上面列举的某种竖写语言，还是英语。

从上到下书写的语言，`writing-mode` 属性的值是 `horizontal-tb`，此时主轴由从左至右的方向顺时针旋转 90 度，因此 `flex-direction: row` 从上到下，而 `flex-direction:`

column；从右至左。把 flex-direction 属性的各个值应用到下述标记上得到的结果如图 12-10 所示。

```
<ol lang="jp">
    <li>一</li>
    <li>二</li>
    <li>三</li>
    <li>四</li>
    <li>五</li>
</ol>
```

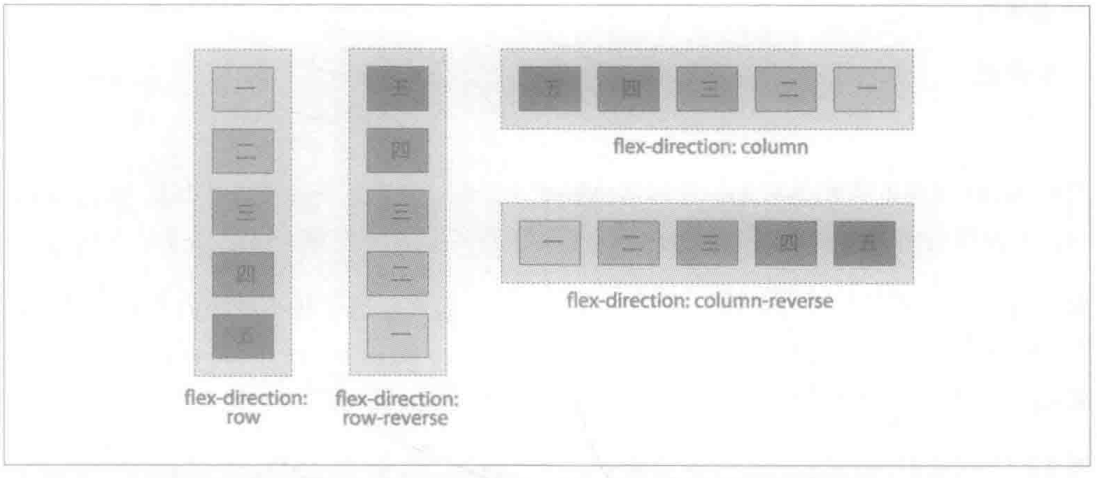


图 12-10：书写模式为 horizontal-tb 时 flex-direction 属性四个值的效果 ▶

你没看错，行是竖的，列是横的。不仅如此，column 的方向是从右至左，而 column-reverse 是从左至右。把这些值应用到从上到下、从右至左书写的语言上就是这样的效果。

好的，我们已经讲了弹性方向与书写模式之间的关系。但是目前所举的例子都只有一行或一列弹性元素，如果弹性元素的主维度（row 时的宽度之和，column 时的高度之和）在弹性容器中放不下将怎样呢？我们可以让放不下的元素溢出，也可以换行。后文还会说明如何缩减弹性元素的尺寸，以便放得下。

12.2.3 换行

如果弹性元素在弹性容器的主轴上放不下，默认情况下弹性元素不会换行，也不会自行调整尺寸。如果通过 flex 属性设定允许弹性元素缩减尺寸（见 12.12 节），那就缩减尺寸，否则，弹性元素将从容器框的边界溢出。

这个行为受我们的控制。我们可以在容器上设置 flex-wrap 属性，允许弹性元素换行，变成多行或多列，而不让弹性元素从容器中溢出，或者缩减尺寸，挤在同一行。

flex-wrap

取值	nowrap | wrap | wrap-reverse
初始值	nowrap
适用于	弹性容器
计算值	指定的值
继承性	否
动画性	否

flex-wrap 属性的作用是限制弹性容器只能显示一行，或者允许弹性元素在必要时显示多行。允许换行时，wrap 和 wrap-reverse 决定多出的行显示在第一行之前还是之后。

默认情况下，不管有多少弹性元素，全部在一行里绘制。这往往不是我们想要的效果。遇到这种情况就要请出 flex-wrap 属性了。设为 wrap 或 wrap-reverse 时，如果弹性元素超出了弹性容器的边界，将换行显示放不下的弹性元素。

图 12-11 展示 flex-direction 的值为 row 时（而且语言是从左至右书写的）flex-wrap 属性三个值的效果。图中的示例有两行弹性元素，可以看出，后续的行添加在垂轴上（这里是纵轴）。

一般情况下，换行时，对 row 和 row-reverse 来说垂轴从上指向下方；对 column 和 column-reverse 来说，垂轴与语言的横排方向一样。wrap-reverse 值的作用与 wrap 类似，不过额外的行添加在第一行前面，而不是后面。

设为 wrap-reverse 时，垂轴的方向相反：对 row 和 row-reverse 来说，后续的行在上方绘制；对 column 和 column-reverse 来说，后续的行在前一列的左侧绘制。类似地，在从右至左书写的语言中，设为 row wrap-reverse 和 row-reverse wrap-reverse 时，新行也添加到上方，但是设为 column wrap-reverse 和 column-reverse wrap-reverse 时，新行添加到右侧，即与语言的书写方向和垂轴的方向相反。

这些轴稍后再讨论，下面先来看能同时设置弹性方向和换行方式的简写属性。

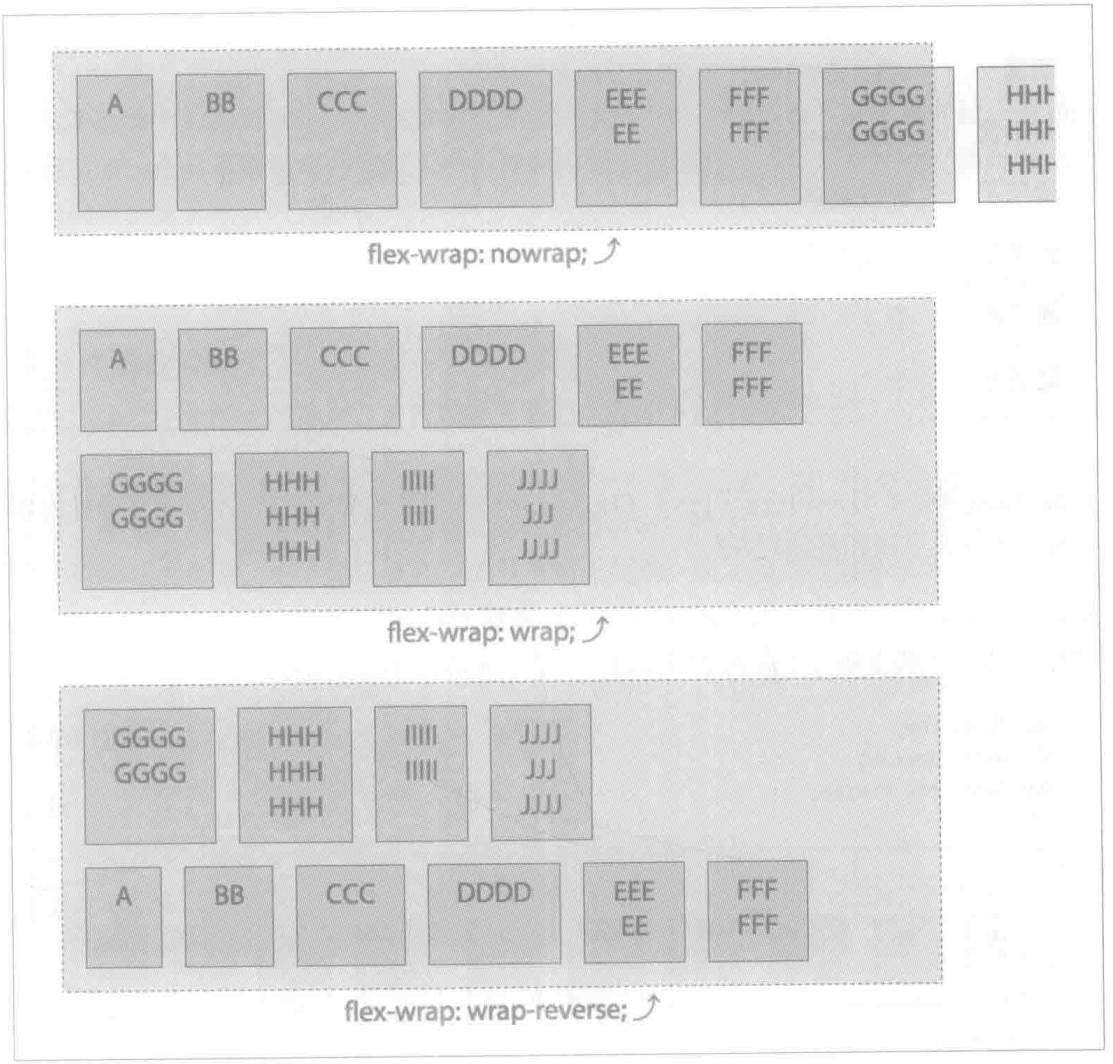


图 12-11：横排时 flex-wrap 属性三个值的效果

12.2.4 定义弹性流

flex-flow 属性用于定义主轴和垂轴的方向，以及是否允许弹性元素换行。

flex-flow

取值	*<flex-direction>* ‖ *<flex-wrap>*
初始值	row nowrap
适用于	弹性容器
计算值	指定的值
继承性	否
动画性	否

flex-flow属性是flex-direction和flex-wrap两个属性的简写形式，用于定义弹性容器的换行方式及主轴和垂轴的方向。

把display属性设为flex或inline-flex后，省略flex-flow、flex-direction和flex-wrap相当于声明下述三个规则中的任何一个，结果如图 12-12 所示：

```
flex-flow: row;
flex-flow: nowrap;
flex-flow: row nowrap;
```

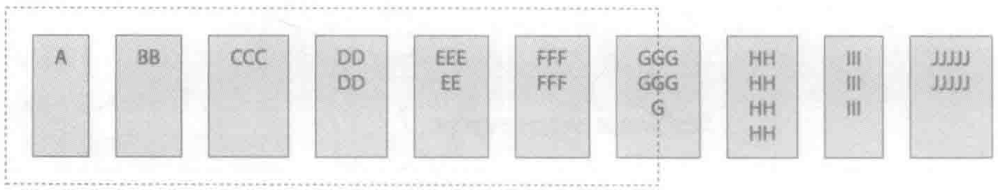


图 12-12：横排不换行的弹性流 ▶

在从左至右的书写模式下，声明前面给出的任何一个值，或者完全省略flex-flow属性，得到的弹性容器的主轴是横向的，而且不换行。图 12-12 中的弹性元素在横轴上均匀分布，而且显示在同一行，超过 500 像素宽的容器后溢出。

如果想要的是反向纵排且允许换行的弹性流，使用下述声明中的任何一个都可以：

```
flex-flow: column-reverse wrap;
flex-flow: wrap column-reverse;
```

在从左至右书写的语言中，上述声明使弹性元素从下向上排，而且从左侧开始，新列添加在右侧。在日语等纵向书写模式中，得到的列是横排的，从左到右流动，换行时上部多出的内容显示在下一列的底部。

我们多次用到“主轴”和“垂轴”，但是还没有深入说明它们是什么意思。现在该说说了。

深入理解各种轴

首先，弹性元素沿主轴排布。各行弹性元素沿垂轴的方向添加。

在介绍 `flex-wrap` 属性之前，所有示例都只有一行弹性元素。那一行弹性元素在主轴上排布，沿主方向，从主轴起边指向主轴终边。根据所设的 `flex-direction` 属性，弹性元素可能并排着排布，也可能从上到下或从下到上排布，沿主轴的方向显示为一行或一列。如图 12-13 所示。

可以看出，图 12-13 中有大量术语，其中很多是新的，要说明一下。下面对各术语做个简单的定义。

主轴
: 内容沿此轴流动。在弹性盒中，指弹性元素流动的方向。

主轴尺寸
: 主轴方向上内容的总长度。

主轴起边
: 主轴上内容开始流动的那一端。

主轴终边
: 主轴上内容流向的那一端，与主轴起边相对。

垂轴
: 块级元素沿此轴堆叠。在弹性盒中，指放置新弹性元素行的方向（前提是允许换行）。

垂轴尺寸
: 垂轴方向上内容的总长度。

垂轴起边

垂轴上块级元素开始堆叠的那一边。

垂轴终边

垂轴上与起边相对的那一边。

这些要素的位置取决于弹性方向、换行方式和书写模式。图解每种书写模式有点难，下面仅以从左至右书写的语言为例。各种情况见表 12-1。



注意，书写模式反转后，一切都反过来了，这一点一定要理解。为了便于更好地说明（和弄清）弹性布局，本章后续的内容和示例都基于从左至右的书写模式，但是会提及书写模式对各弹性属性和功能的影响。

表 12-1：从左至右书写模式下主轴和垂轴的维度和方向，及其起边和终边的位置

	从左至右书写模式下的弹性方向			
	row	**row-reverse**	**column**	**column-reverse**
主轴	从左至右	从右至左	从上到下	从下到上
主轴起边	左边	右边	顶边	底边
主轴终边	右边	左边	底边	顶边
主轴尺寸	宽度	宽度	高度	高度
主轴维度	横向	横向	纵向	纵向
垂轴	从上到下	从上到下	从左至右	从左至右
垂轴起边	顶边	顶边	左边	左边
垂轴终边	底边	底边	右边	右边
垂轴尺寸	高度	高度	宽度	宽度
垂轴维度	纵向	纵向	横向	横向

对 `flex-direction` 来说，我们知道弹性元素从主轴起边开始在弹性容器的主轴上排布。如果使用 `flex-wrap` 属性允许一行中放不下的弹性元素换行，垂轴的方向决定额外的行在什么方向上添加。

学过 12.2.4 节介绍的 `flex-flow` 简写属性之后我们知道，可以把原本将超出容器主轴尺寸的弹性元素换行显示。每一行中的弹性元素沿主轴方向排布，从主轴起边流向主轴终边，但是额外的行沿垂轴方向排布，从垂轴起边流向垂轴终边。

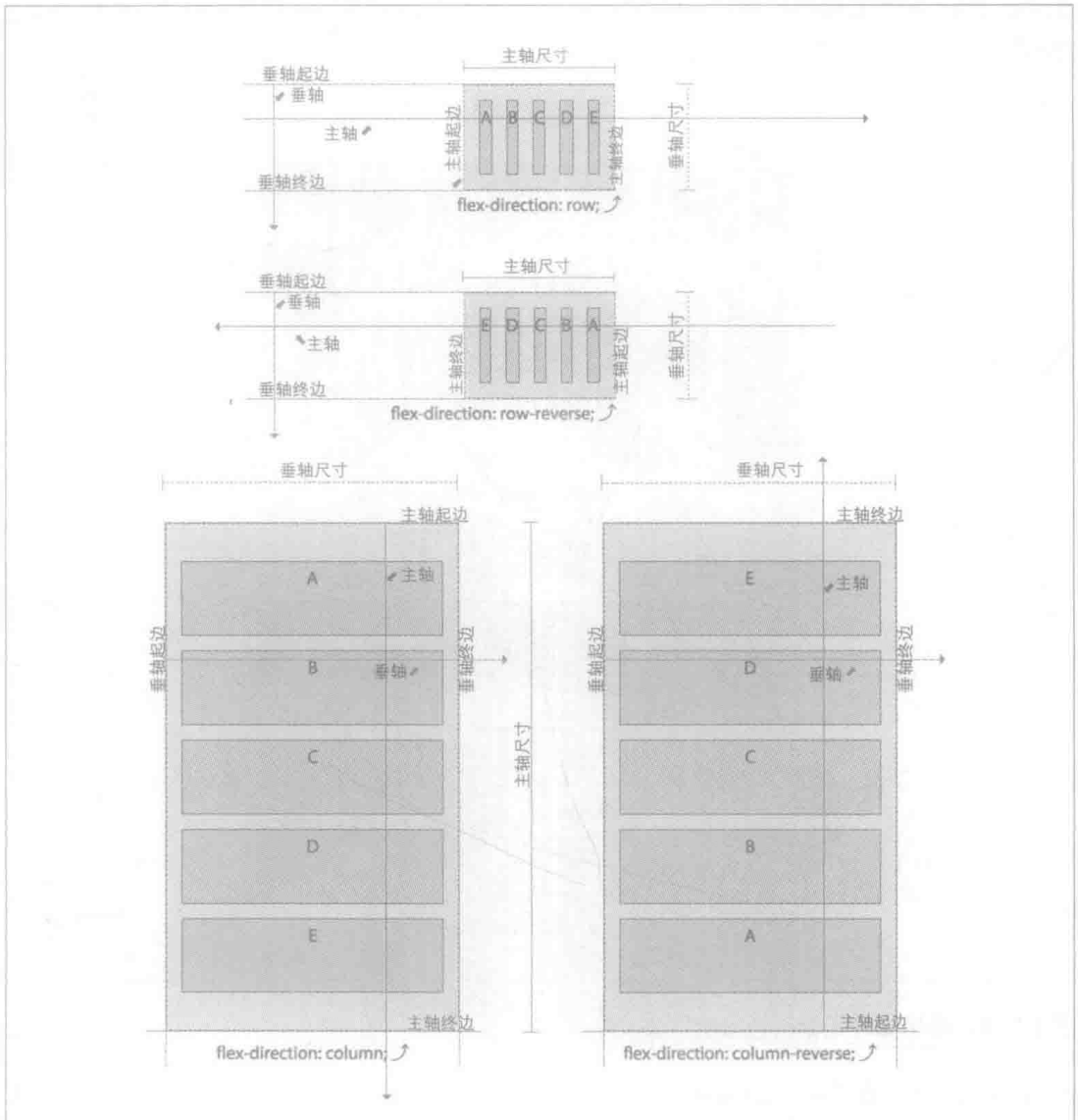


图 12-13：从左至右书写模式下主轴和垂轴上各术语的位置

垂轴始终与主轴垂直。从图 12-14 可以看出，横排弹性元素时，垂轴是纵向的。额外的行就沿垂轴方向添加。图中的示例设置的是 `flex-flow: row wrap` 和 `flex-flow:row-reverse wrap`，而且语言是横向书写的，因此额外的行添加在现有行的下方。

垂轴尺寸与主轴尺寸相对，不管语言是从右至左还是从左至右书写的（不含从上到下书写的语言），对 `row` 和 `row-reverse` 来说是高度，对 `column` 和 `column-reverse` 来说是

宽度。各行弹性元素都在弹性容器中，第一行放在垂轴起边，后续各行依次向垂轴终边排开。

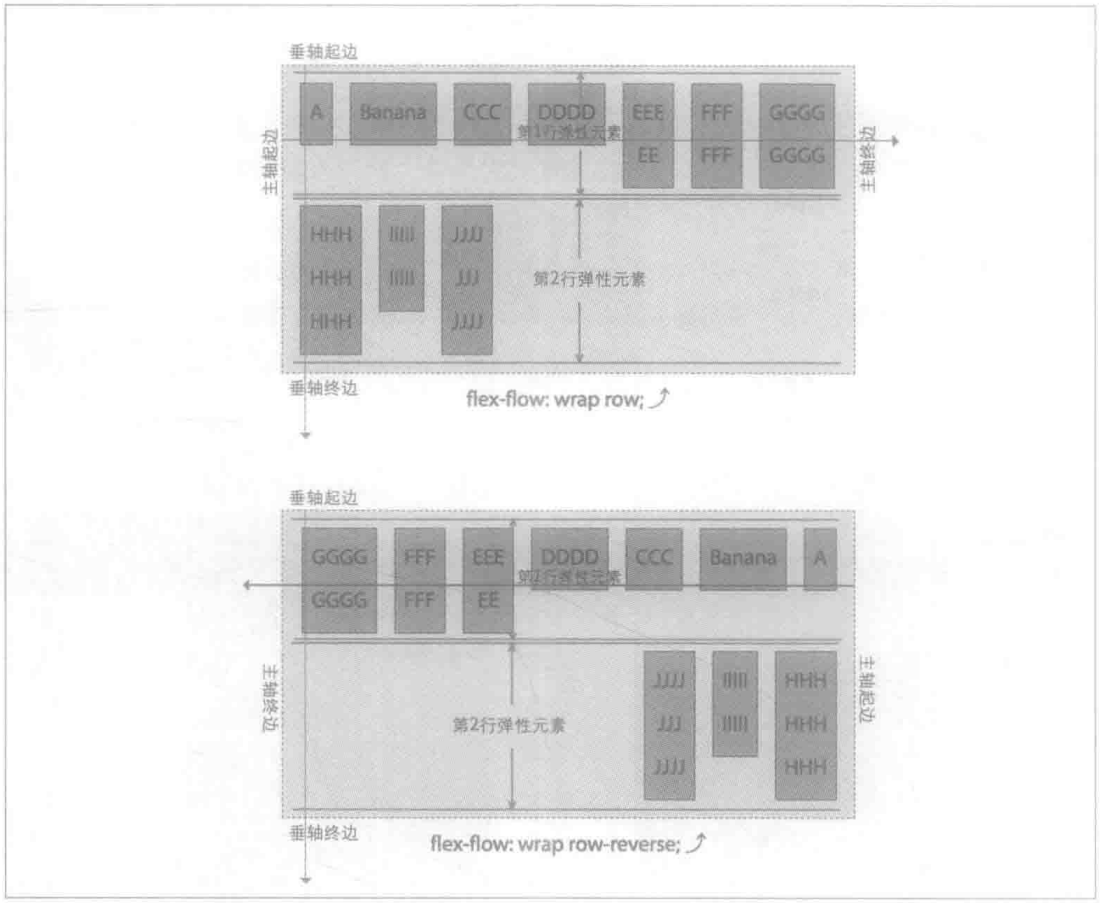


图 12-14：横排弹性元素时各行的堆叠次序

wrap-reverse 值反转垂轴的方向。正常情况下，如果把 flex-direction 设为 row 或 row-reverse，垂轴从上到下，垂轴起边在上部，垂轴终边在下部。但是如果把 flex-wrap 设为 wrap-reverse，垂轴起边和垂轴终边的位置将对调，垂轴起边在下部，而垂轴终边在上部，垂轴从下指向上方。因此，额外的行将添加到前一行的上方。

如果把 flex-direction 设为 column 或 column-reverse，默认情况下，在从左至右书写的语言中，垂轴从左指向右方，新行添加在前一行的右侧。如图 12-15 所示，如果把 flex-wrap 设为 wrap-reverse，垂轴的方向将反过来，垂轴起边在右侧，垂轴终边在左侧，垂轴由右指向左方，因此额外的行添加到前一行的左侧。



为了清楚地表示弹性元素行的高度和流动方向，图 12-14 和图 12-15 在弹性容器上声明了 align-items: flex-start 和 align-content: flex-start。这两个属性在后面的章节介绍。

了解这些术语和维度之后，下面回到 flex-wrap 属性上。

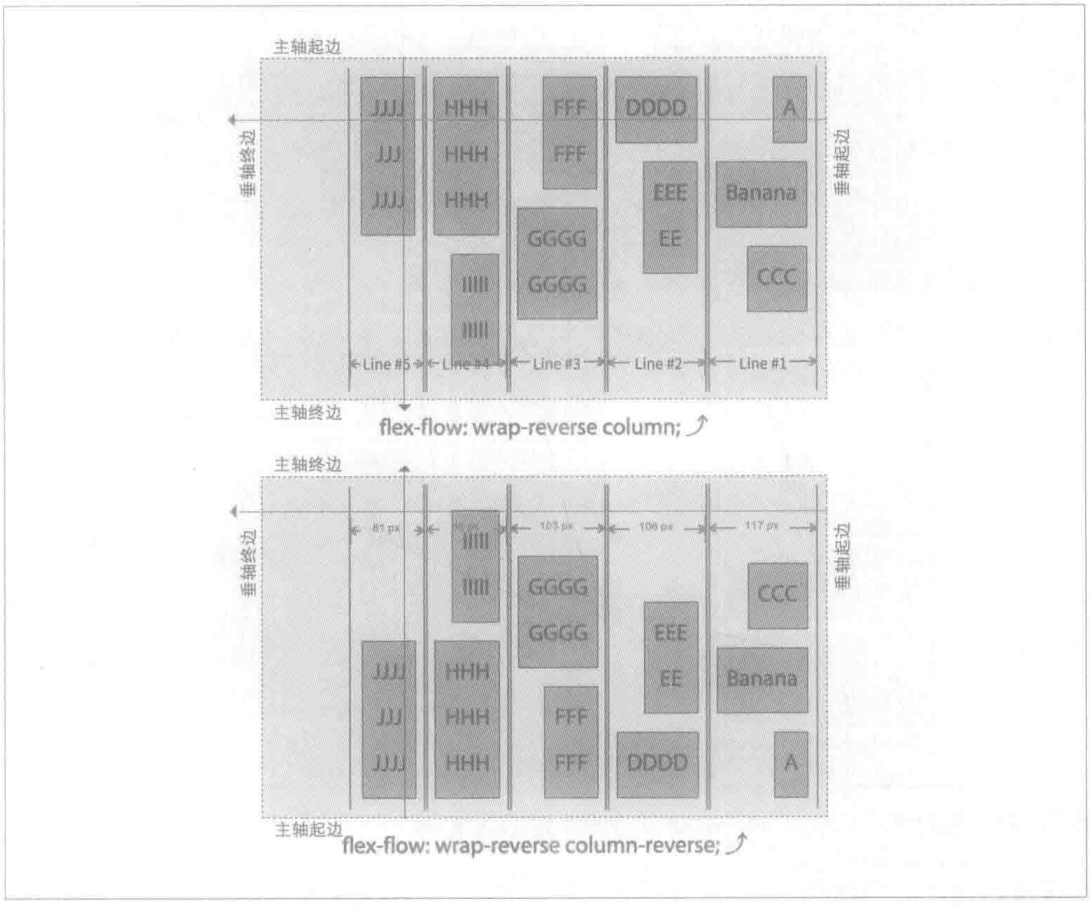


图 12-15：纵排弹性元素时各行的堆叠次序

12.2.5 flex-wrap 续谈

默认值 nowrap 禁止换行，因此前文讨论的垂轴方向没有任何意义，毕竟根本不会出现第二行。如果可能出现额外的行（flex-wrap 设为 wrap 或 wrap-reverse 时），那些行将沿垂轴方向添加。第一行放在垂轴的起边，额外的行则向垂轴终边排开。

使用flex-wrap: wrap-reverse可以反转垂轴的方向，把新行添加到前一行的上方或左侧。图 12-16 中最后一个示例用的就是 wrap-reverse。可以看到，第一行弹性元素从垂轴起边开始，不过后续各行却沿反方向添加。

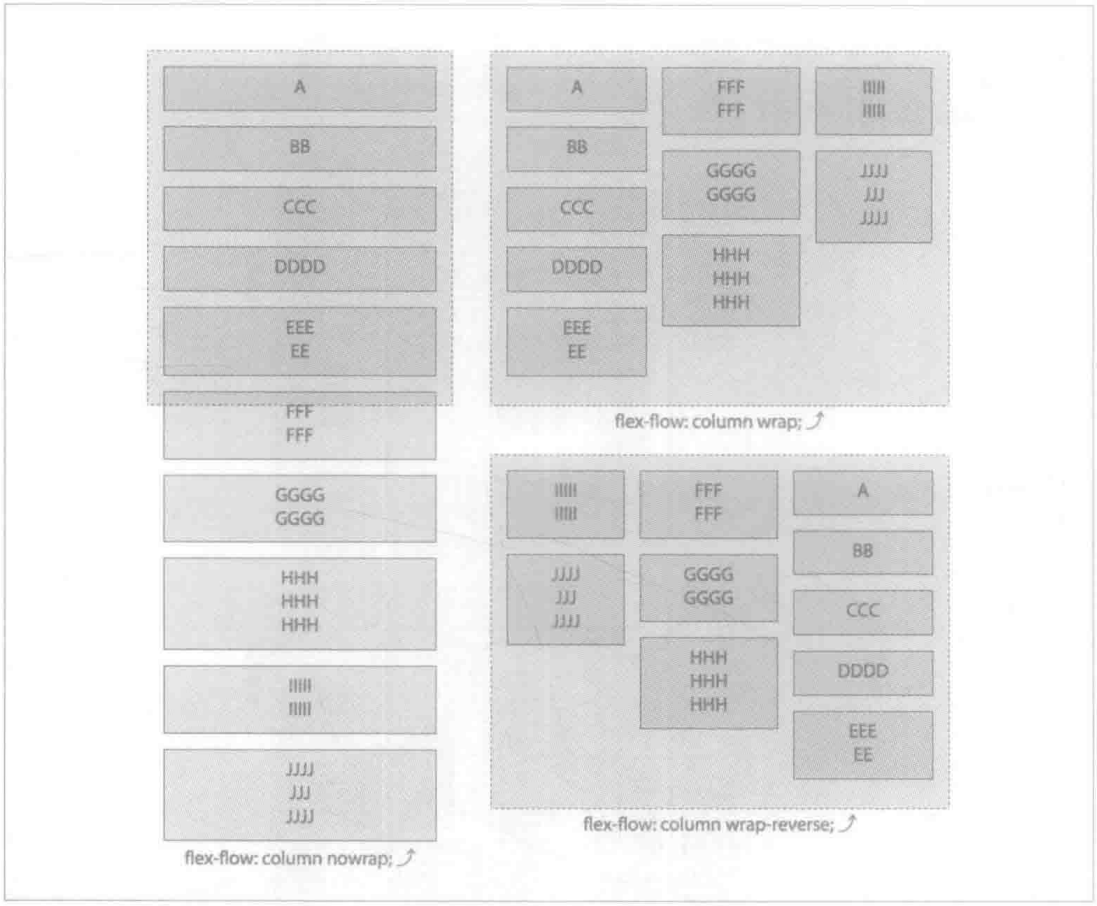


图 12-16：竖排弹性元素时 flex-wrap 属性三个值的效果 ▶

与横排相比，使用的 flex-wrap 值还是那三个，不过 flex-direction 属性的值是 column，而不是 row。这里，弹性元素沿纵轴排布。但是与横排时一样，如果没有通过 flex-wrap 属性启用换行（显式在容器上设置 flex-wrap: nowrap，或者不声明这个属性，默认为 nowrap），不会出现新行，即便弹性元素将超出弹性容器的边界，也不会换行。

弹性流动方向为 column 时，如果弹性元素在弹性容器中放不下，而且禁止换行，那么弹性元素将从弹性容器中溢出，这一点与弹性流动方向为 row 时一样。然而，使用 min-width: 0 或类似的技术能让弹性元素缩小尺寸，以便在容器中能放得下。不过，弹性元素不能无限缩小，小于边框、内边距和外边距之和。

如果在弹性容器上设置了 flex-flow: column wrap，当空间不够，在一行里放不下全部弹性元素时，将出现换行。多出的弹性元素将放在沿垂轴方向放置的一个新行里。这里，新行是纵排的（一列），在前一行的右侧，如图 12-16 中的 flex-flow: column wrap 示例所示。在那个例子中，弹性元素分为三行。设置 flex-flow: column wrap-reverse 时，情况类似，不过垂轴起边和垂轴终边的位置将对调，因此第一列在右方，而后续各列（弹性元素行）添加到第一行的左侧。

可以看出，flex-direction 和 flex-wrap 对布局有很大的影响，而且二者之间也有不小的影响。如果想设置其中某一个属性，最好两个都设置，此时可以使用规范强烈推荐的 flex-flow 属性。

12.3 布置弹性元素

目前所举的例子都没有涉及每一行中弹性元素的具体位置，我们不知道如何确定弹性元素的位置。横排的弹性元素横向展开似乎是理所当然的，但是为什么所有元素都靠主轴起边一侧紧挨在一起呢？为什么不增加弹性元素的尺寸，填满全部可用空间呢？为什么不让弹性元素在一行里均匀分布呢？

下面举个例子，请看图 12-17。看到左上角的空白了吗？这是从下到上、从右至左的流动方式，新弹性元素放在前一个弹性元素的上方，新行放在前一行的左侧。

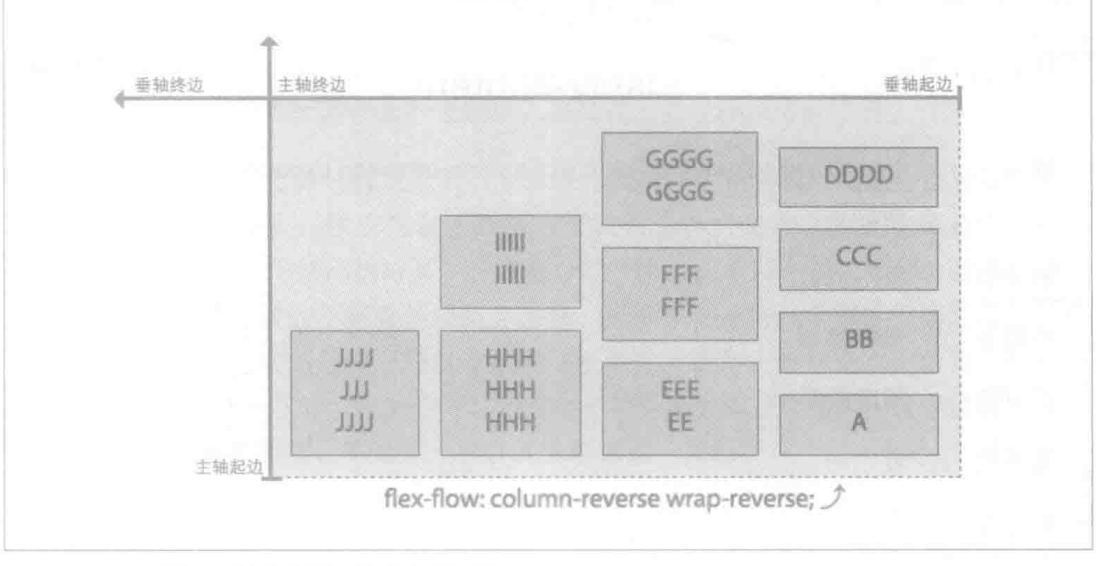


图 12-17：主轴终边和垂轴终边的方向出现空白

默认情况下，不管把 flex-flow 设为什么值，在弹性容器中放完全部弹性元素后留下的空白始终出现在主轴终边和垂轴终边的方向……不过有属性能调整这一行为。

12.4 弹性容器

在目前所举的例子中，如果弹性元素不能填满整个弹性容器，弹性元素将统一向主轴起边靠紧。不过，弹性元素也可以紧靠主轴终边，或者在弹性容器中居中，甚至可以在主轴上均匀分布。

弹性布局规范提供的一些属性能控制空间的分布情况，除了 display 和 flex-flow 之外，CSS Flexible Box Layout Module Level 1 还提供了一些能应用到弹性容器上的属性，包括 justify-content、align-content 和 align-items。

justify-content 属性控制一行里的弹性元素在主轴上如何分布。align-content 属性定义各弹性元素行在弹性容器的垂轴上如何分布。align-items 属性定义各行里的弹性元素在垂轴上如何分布。先看一行里的弹性元素是如何布置的。

12.5 调整内容

justify-content 属性指明在弹性容器的主轴上如何分布各行里的弹性元素。这个属性应用于弹性容器上，不能用到单个弹性元素上。

justify-content	
取值	flex-start | flex-end | center | space-between | space-around | space-evenly
初始值	flex-start
适用于	弹性容器
计算值	指定的值
继承性	否
动画性	否

justify-content 属性定义如何把弹性容器的空间分配到弹性元素的四周或者弹性元素之间。六个可选值的效果如图 12-18 所示。

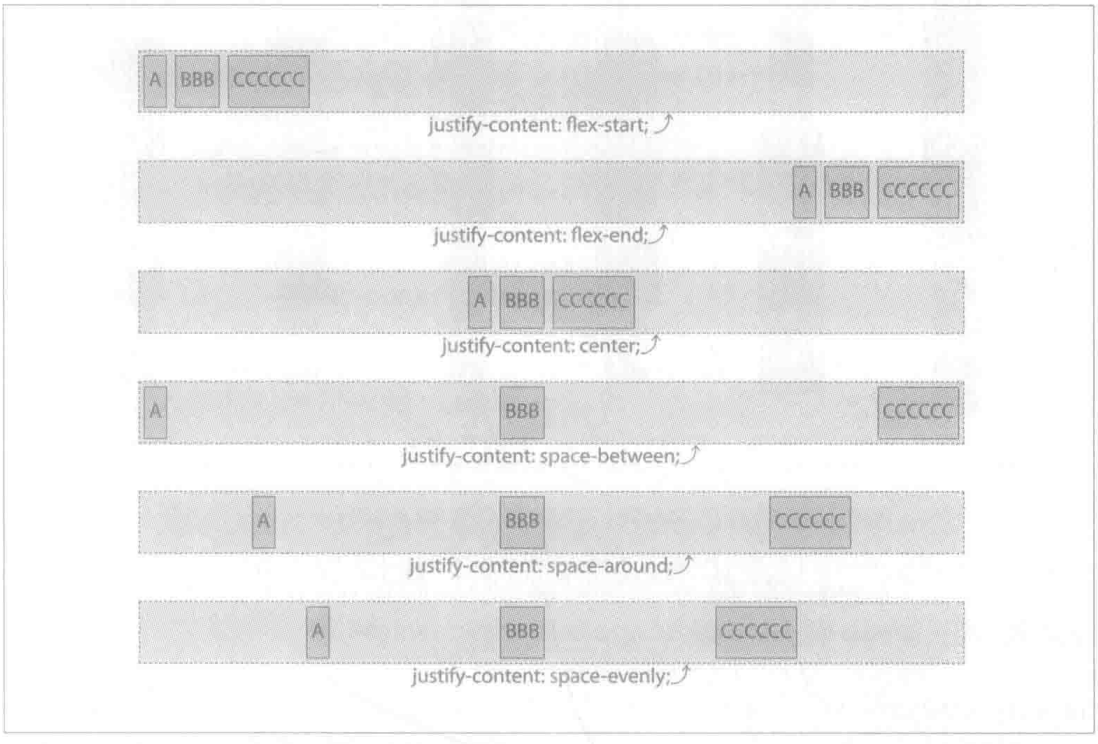


图 12-18：justify-content 属性的六个值

设为默认值 flex-start 时，弹性元素紧靠主轴起边。设为 flex-end 时，弹性元素紧靠主轴终边。center 把弹性元素作为一个整体，居中显示在主轴尺寸的中点。

space-between 值把每一行里的第一个弹性元素放在主轴起边，把每一行里最后一个弹性元素放在主轴终边，然后在余下的每一对相邻的弹性元素之间放置等量的空白。space-around 把余下的空间拆分开，把各部分的一半分配给每个弹性元素，看起来就像每个元素四周都有等量的不折叠外边距。注意，这意味着任何两个弹性元素之间的距离是第一个弹性元素与主轴起边之间以及最后一个弹性元素与主轴终边之间距离的两倍。space-evenly 也把余下的空间拆分开，不过每个间距的长度是相等的。这意味着，到主轴起边和终边之间的距离，与弹性元素之间的距离是一样的。

justify-content 属性不仅影响一行中弹性元素的位置。如果禁止弹性元素换行，而且弹性元素将溢出，那么 justify-content 的值还影响弹性元素如何从弹性容器中溢出，如图 12-19 所示。

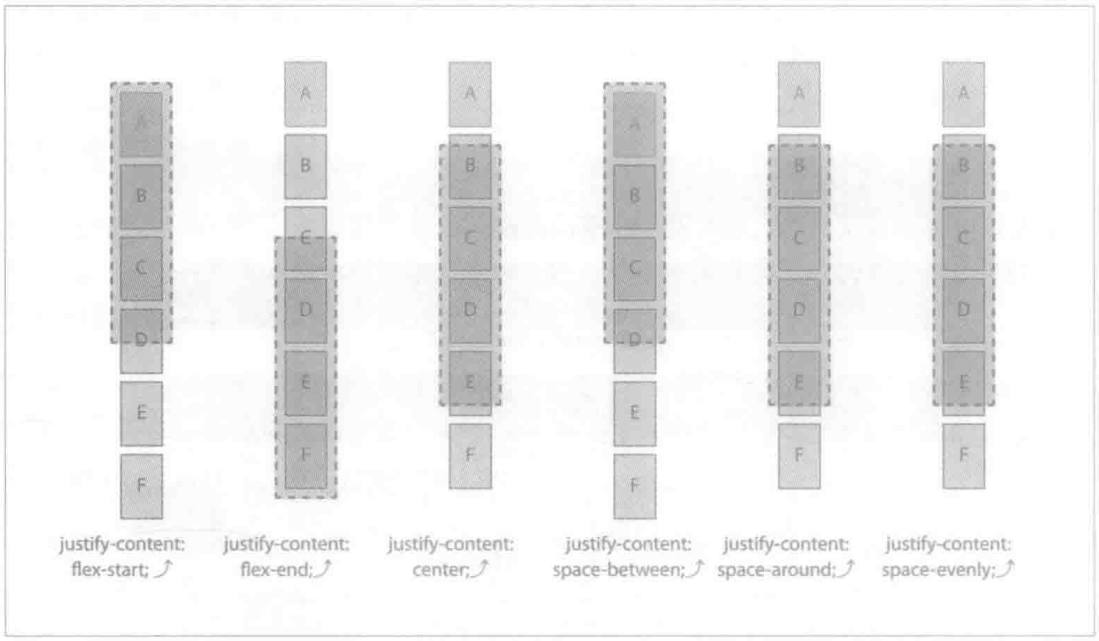


图 12-19：justify-content 属性对弹性容器中单行弹性元素溢出的影响

下面稍微深入讲解一下这六个值。

`justify-content: flex-start` 显式设置默认行为，即所有弹性元素都向主轴起边靠拢，每一行里的第一个弹性元素都紧靠主轴起边一侧。后续各弹性元素向前一个弹性元素位于主轴终边一侧的边靠拢，直到一行结尾（如果允许换行）。主轴起边的位置由弹性方向和书写模式决定，详情参见本章前面“深入理解各种轴”一节。如果空间不够放下所有弹性元素，而且换行方式为默认值 `nowrap`，或者显式声明禁止换行，弹性元素将从主轴终边溢出，如图 12-20 所示。

设为 `justify-content: flex-end` 时，一行里的最后一个弹性元素向主轴终边靠拢，前面各弹性元素紧挨后一个弹性元素。此时，如果不允许弹性元素换行，而且空间不足以放下全部弹性元素，那么弹性元素将从主轴起边溢出，如图 12-21 所示。如果一行里有空白的话，将出现在主轴起边那一侧。

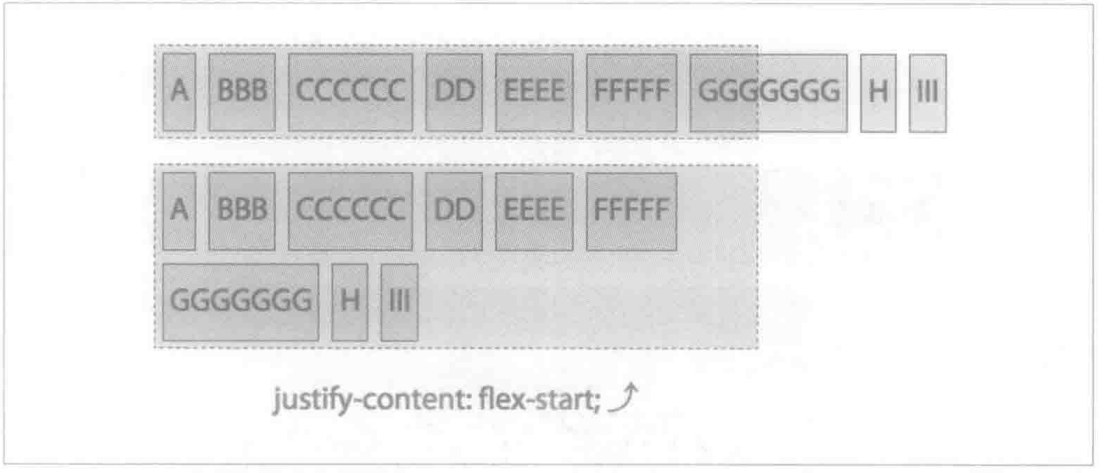


图 12-20：对齐方式为 flex-start ▶

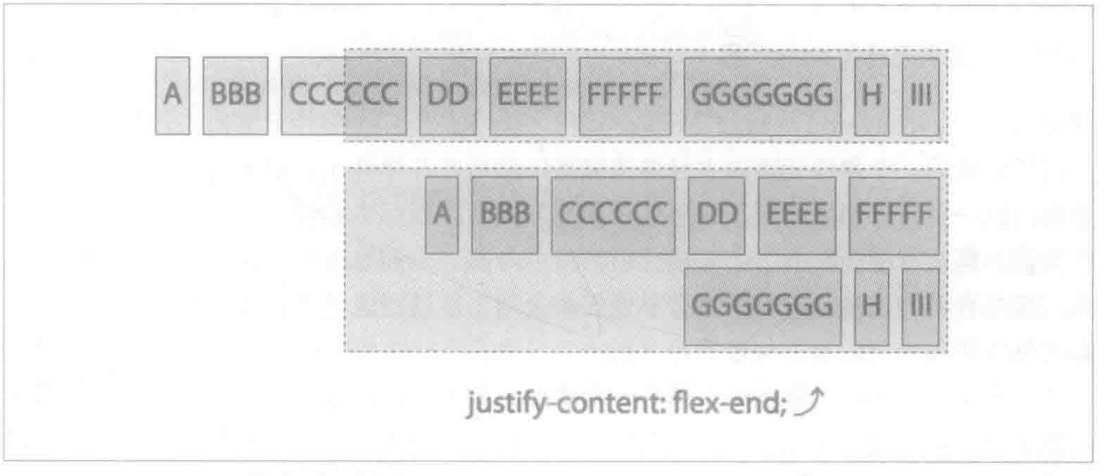


图 12-21：对齐方式为 flex-end ▶

设为 `justify-content: center` 时，所有弹性元素挤在一起，统一居中显示在每一行里，而不再向主轴起边或终边靠拢。如果空间放不下全部弹性元素，而且不允许换行，弹性元素将从主轴起边和终边一起溢出，而且溢出量相同，如图 12-22 中的第二个例子所示。如果弹性元素显示为多行，每一行里的弹性元素都居中显示，额外的空白将出现在主轴起边和终边一侧。

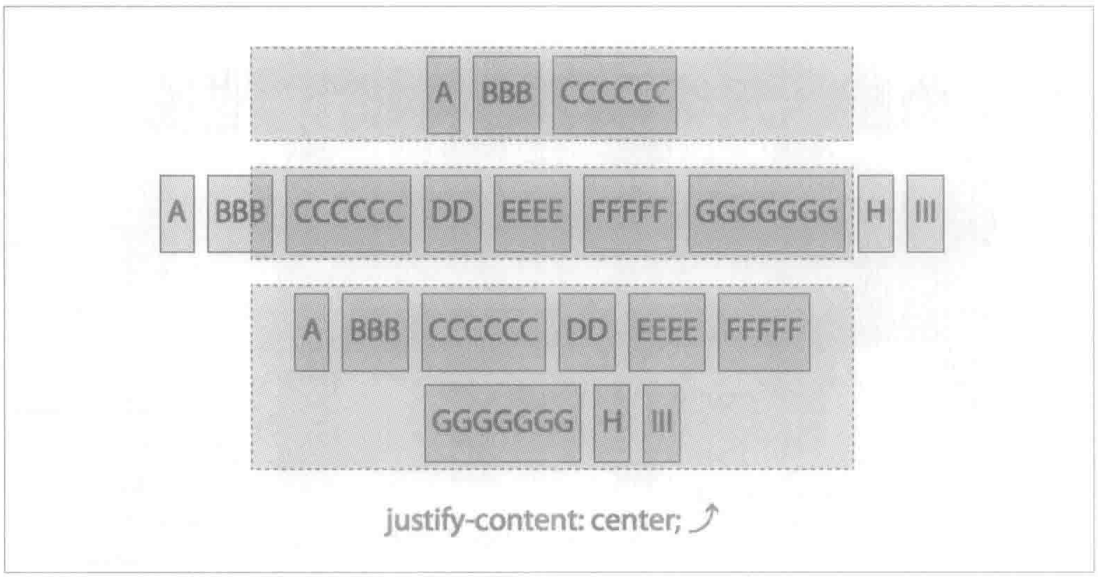


图 12-22：对齐方式为 center ▶

设为 justify-content: space-between 时，一行里的第一个弹性元素向主轴起边靠拢，一行里的最后一个弹性元素向主轴终边靠拢，然后在各弹性元素周围放置等量的空白，直到占满一整行。如果换行了，通过相同的方式处理新行里的弹性元素。假如有三个弹性元素，那么第一个和第二个元素之间的空白与第二个和第三个元素之间的空白是相等的，但是容器的主轴起边与第一个弹性元素之间，以及对边（主轴终边）与最后一个元素主轴终边那一侧的边之间没有额外的空白，如图 12-23 中的第二个例子所示。此时第一个元素是向主轴起边靠拢的，只有一个弹性元素或禁止换行时弹性元素将从弹性容器中溢出时，尤其要注意这一点。这意味着，只有一个弹性元素时，弹性元素将向主轴起边靠拢，而不居中。对很多人来说，这一点初看起来都有点违背直觉。

设为 justify-content: space-between 时，一行里的任何两个弹性元素之间具有相同的间距，但是不同行里的间距不一定相等。如果允许换行，在最后一行里，第一个弹性元素将向主轴起边靠拢，最后一个弹性元素（假如最后一行里有两个或以上弹性元素）将向主轴终边靠拢，而且每对相邻的弹性元素之间具有相等的间距。如图 12-23 中的最后一个例子所示，每一行里的第一个弹性元素 A 和 G 向主轴起边靠拢，而每一行里的最后一个弹性元素F和I向主轴终边靠拢。每一行里相邻两个弹性元素之间的间距是相等的，但是第一行里的间距比第二行里的间距窄一些。

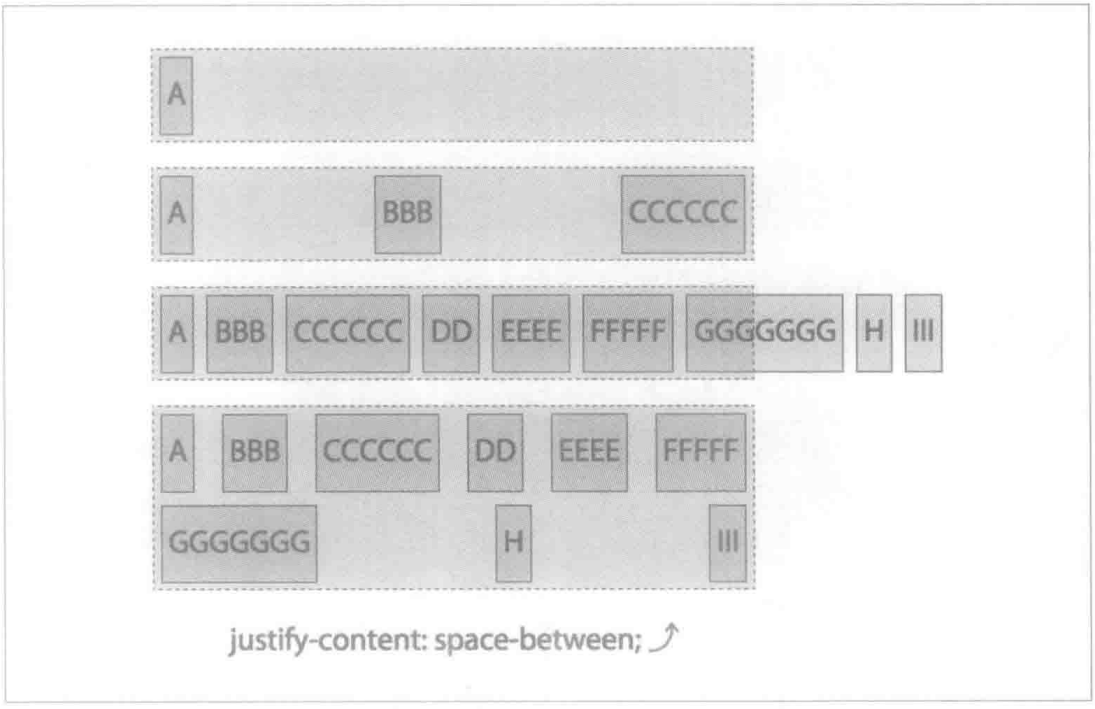


图 12-23：对齐方式为 space-between

设为 justify-content: space-around 时，一行中额外的空白均匀分布在各弹性元素的周围，就像主轴方向上每个元素都有不折叠的等量外边距一样。因此，第一个和第二个弹性元素之间的间距是主轴起边与第一个元素之间及主轴终边与最后一个弹性元素之间间距的两倍，如图 12-24 所示。

如果弹性元素分为多行显示，每一行里各弹性元素周围的空间由可用空间决定。虽然一行里各弹性元素周围的空间是相等的，但是行与行之间可能不一样，如图 12-24 中的最后一个例子所示。A 与 B 之间的距离及 G 与 H 之间的距离是主轴起边与 A 之间及主轴起边与 G 之间的两倍。

如果设置了 nowrap，而且弹性容器的空间不足以在主轴方向上放下全部弹性元素，弹性元素将从两端溢出，而且溢出量相等，这一点与 center 一样，如图 12-24 中的第三个例子所示。

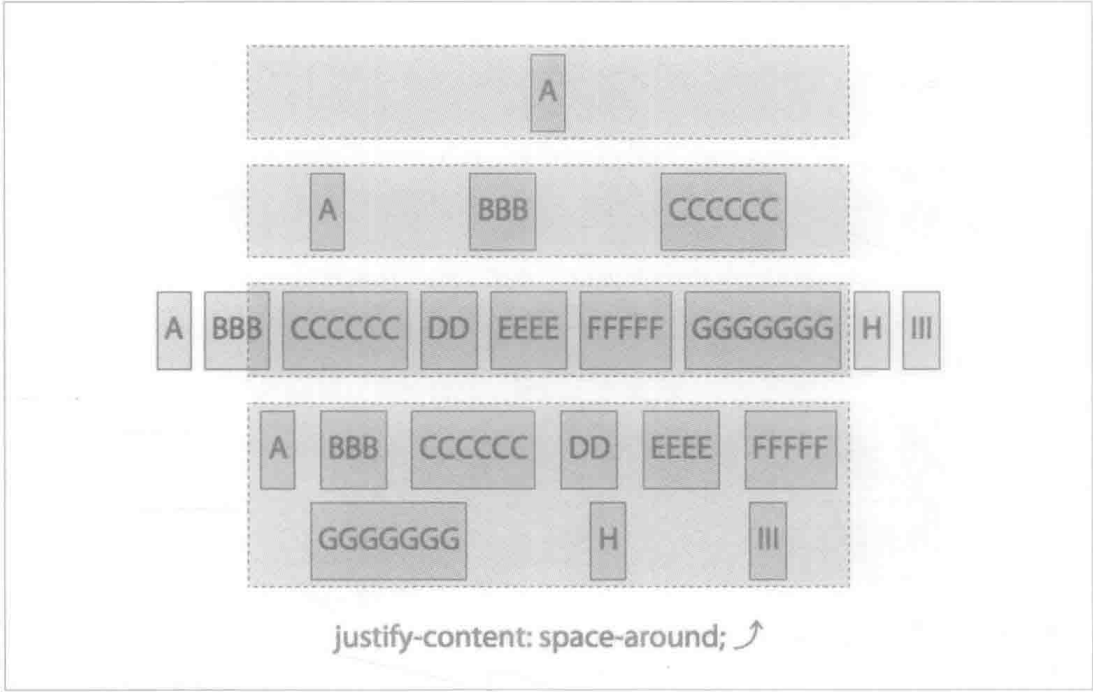


图 12-24：对齐方式为 space-around

设为 justify-content: space-evenly 时，用户代理会计算弹性元素的数量，在此基础上加一，然后把一行里的额外空间分成那么多份（例如，如果有 5 个弹性元素，空间将分成 6 等份）。一行里的每个弹性元素前面将放置一份空间，就像是不折叠的外边距一样，而最后一份将放在一行里的最后一个弹性元素后面。因此，第一个和第二个弹性元素之间的间距与主轴起边和第一个弹性元素之间的间距，以及主轴终边和最后一个弹性元素之间的间距是相等的，如图 12-25 所示。

为了清楚地说明各种布置方式，各个例子之间隔了一段距离，但这可能不便于区分不同方式之间的差别。下面把 center、space-around、space-between 和 space-evenly 放在同一张图里，以便你比较不同的方式，如图 12-26 所示。



均匀分配空间的布置方式目前（2017 年年末）不在弹性盒规范中，而在 CSS Box Alignment 规范中。因为弹性盒规范规定，这种方式必须遵守 CSS Box Alignment 规范，所以应该不久就会写入弹性盒规范中。此外，多数浏览器都已经支持这种布置方式。

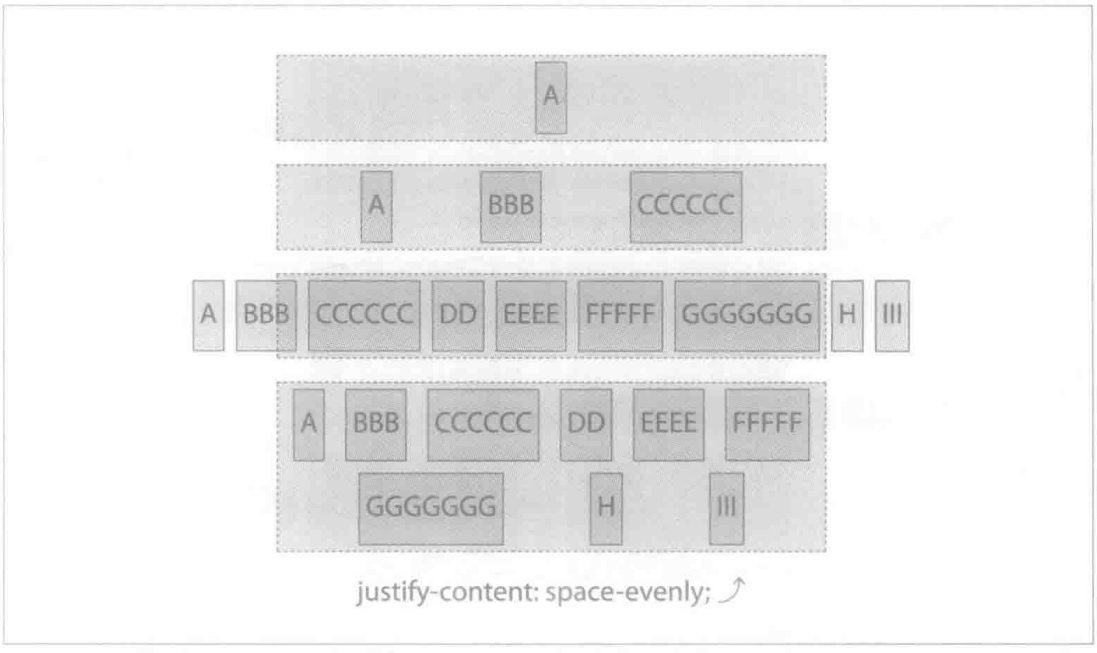


图 12-25：对齐方式为 space-evenly

12.5.1 justify-content 应用举例

图 12-2 中使用的是 justify-content 的默认值，得到的是左对齐的导航栏。把默认值改为 justify-content: flex-end，在英语中将得到右对齐的导航栏：

```
nav {
  display: flex;
  justify-content: flex-start;
}
```

注意，justify-content 应用在弹性容器上。如果应用到链接上，比如 nav a {justify-content: flex-start;}，对齐方式不受影响。

justify-content 的一大优点是，改变书写方向后，比如说改为从右至左的书写模式，无需修改 CSS，标签式链接就会出现在正确的位置。设为 flex-start 时，弹性元素始终统一向主轴起边靠拢；对英语来说，主轴起边在左侧；对希伯来语而言，主轴起边在右侧。如果设为 flex-end，而 flex-direction 的值是 row，那么标签式链接在英语中靠右边显示，在希伯来语中靠左边显示，如图 12-27 所示。

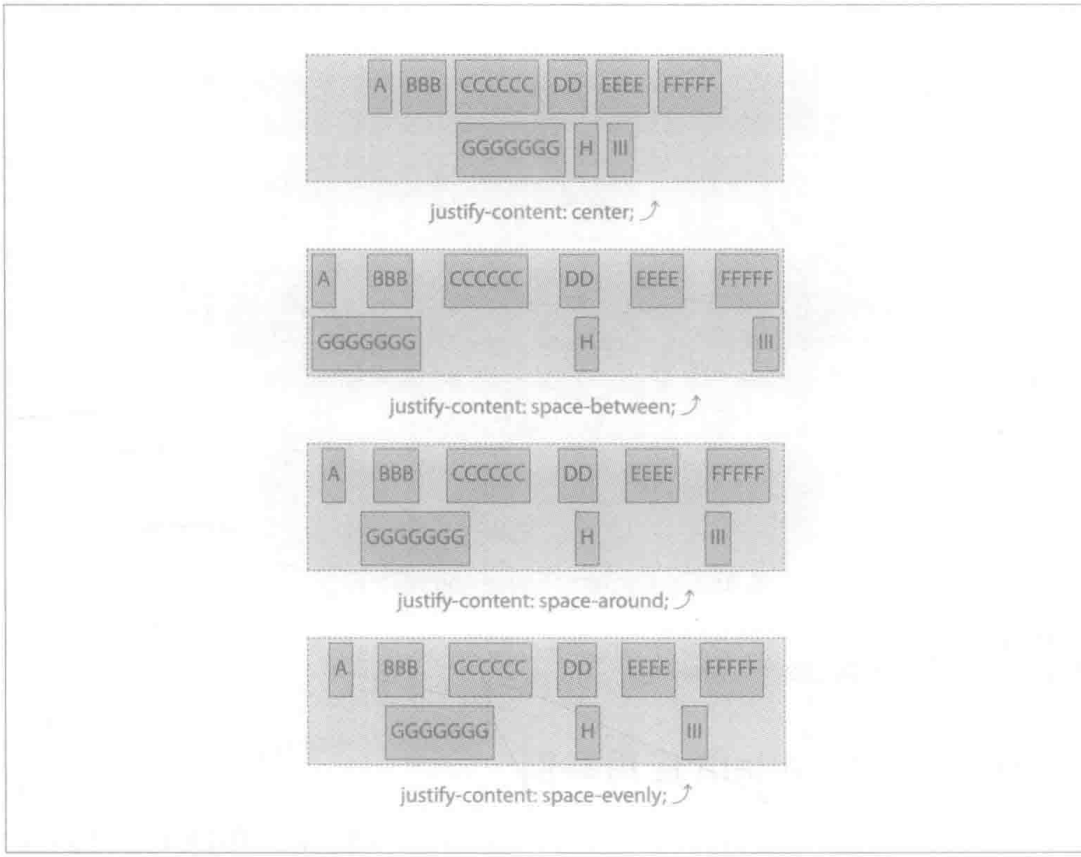


图 12-26：比较 center、space-between、space-around 和 space-evenly

此外，还可以居中显示导航链接，如图 12-28 所示：

```
nav {
  display: flex;
  justify-content: center;
}
```

12.6 对齐元素

`justify-content` 定义弹性元素在弹性容器主轴方向上的对齐方式，而 `align-items` 属性定义的是弹性元素在垂轴方向上的对齐方式。与 `justify-content` 一样，`align-items` 应用在弹性容器上，而不能应用到单个弹性元素上。



图 12-27：在各种语言中都能正确显示的导航栏对齐方式 ▶



图 12-28：通过一个属性和值就能改变布局 ▶

align-items

取值	flex-start | flex-end | center | baseline | stretch
初始值	stretch
适用于	弹性容器
计算值	指定的值
继承性	否
动画性	否

使用 align-items 属性，可以把所有弹性元素都向垂轴的起边、终边或中线对齐。align-items 属性的作用与 justify-content 类似，不过影响的是垂向，设置所有弹性元素（包括匿名弹性元素）在垂轴上的对齐方式。

使用 align-items 属性，可以把所有元素都向垂轴起边或终边靠拢，也可以拉伸元素，同时靠拢起边和终边。此外，还可以把所有弹性元素都居中显示在垂向上。这个属性有五个可选值，包括 flex-start、flex-end、center、baseline 和默认的 stretch，如图 12-29 所示。



align-items 设置的是弹性容器中全部弹性元素的对齐方式，而 align-self 属性能重设单个弹性元素的对齐方式，参见 12.7 节。

从图 12-29 可以看到，弹性元素可以向垂轴起边或终边靠拢，也可以居中，或者经过拉伸，同时靠拢起边和终边。不过 baseline 有点特殊。此时，弹性元素向基线对齐，基线与垂轴起边那一侧的边之间的距离最远的弹性元素将与弹性元素行垂轴起边那一侧的边对齐。

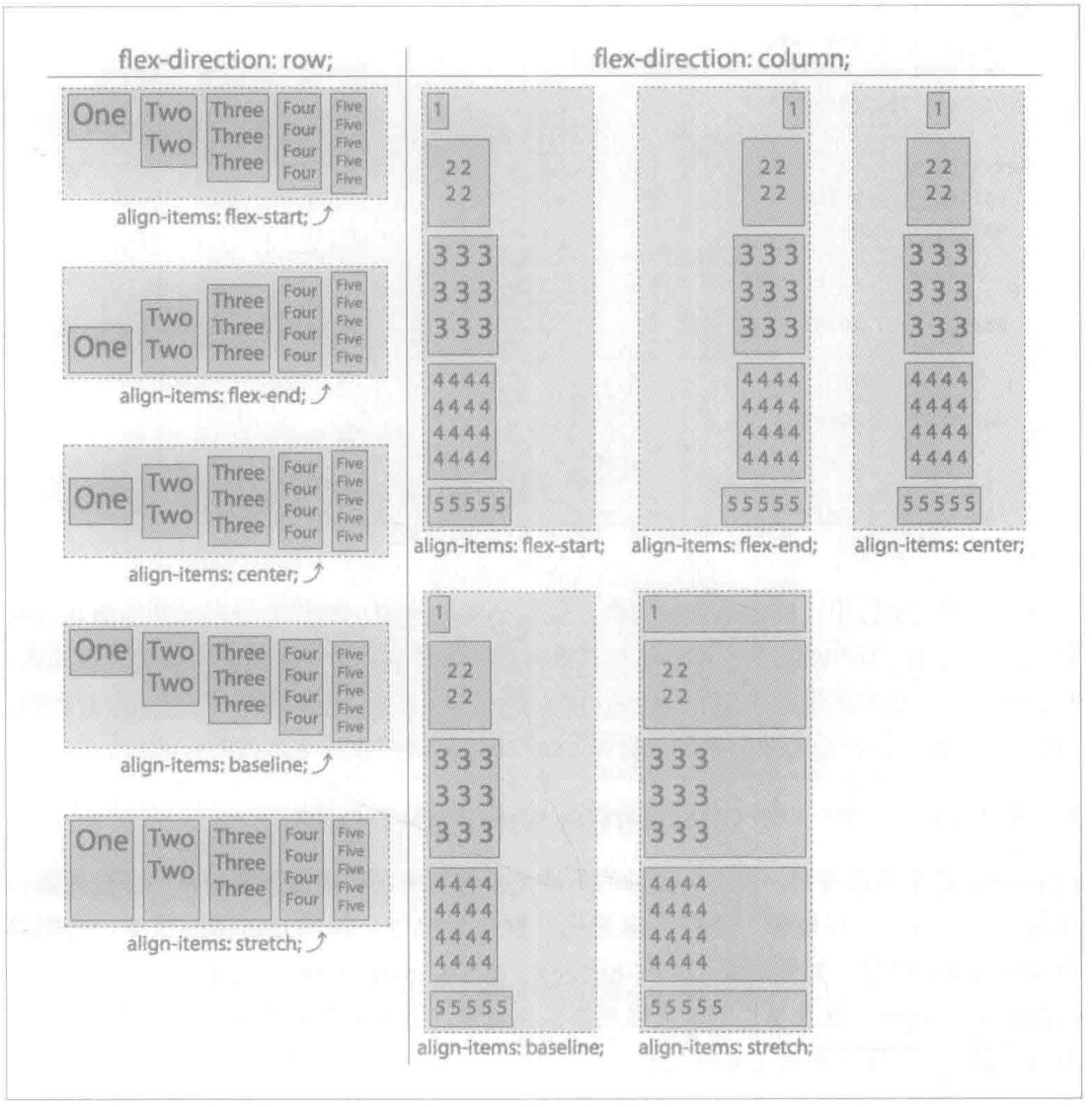


图 12-29：横排 ▶ 和竖排 ▶ 时 align-items 五个值的效果

从上图可以窥见这些值的作用，不过这只是不换行时的情况。下面以下述样式为例，说明多行时的情况。

```
flex-container {
  display: inline-flex;
  flex-flow: row wrap;
  border: 1px dashed;
}
flex-item {
  border: 1px solid;
  margin: 0 10px;
}
.C, .H {
  margin-top: 10px;
}
.D, .I {
  margin-top: 20px;
}
.J {
  font-size: 3rem;
}
```

在每个弹性元素行中，红线是垂轴起边，蓝线是垂轴终边。相邻两行之间的线看起来是紫色的。C、H、D 和 I 的上下外边距与其他弹性元素不同。为了便于区分各个弹性元素，我们在每个弹性元素两侧添加了一些外边距，但这对 align-items 属性的作用没有影响。J 的字号更大，因此行高随之增加。讨论 baseline 值时将体现这么做的目的。

默认值 align-items: stretch 的效果如图 12-30 所示。

stretch 这个名称表明，所有可拉伸的弹性元素将与所在行中最高或最宽的弹性元素一样高或一样宽。“可拉伸”是什么意思呢？默认情况下，弹性元素将被拉伸，占据整个垂轴尺寸；但是，如果设置了 min-height、min-width、max-height、max-width、width 或 height，这些属性的优先级更高。也就是说，如果为弹性元素显式设置了垂轴方向上的尺寸，那么弹性元素就是不可拉伸的，stretch 对其尺寸没有影响。

否则，弹性元素的垂轴起边将向弹性元素行的垂轴起边靠拢，弹性元素的垂轴终边将向弹性元素行的垂轴终边靠拢。垂向尺寸最大的弹性元素保持当前尺寸，其他弹性元素的尺寸则增大，变成与最大那个弹性元素一样。

拉伸后的弹性元素的尺寸包含垂轴起边和垂轴终边那一侧的外边距，此时是弹性元素外边距的外边界与垂轴起边和垂轴终边靠拢。如图 12-31 中的 C、D、H 和 I 所示。

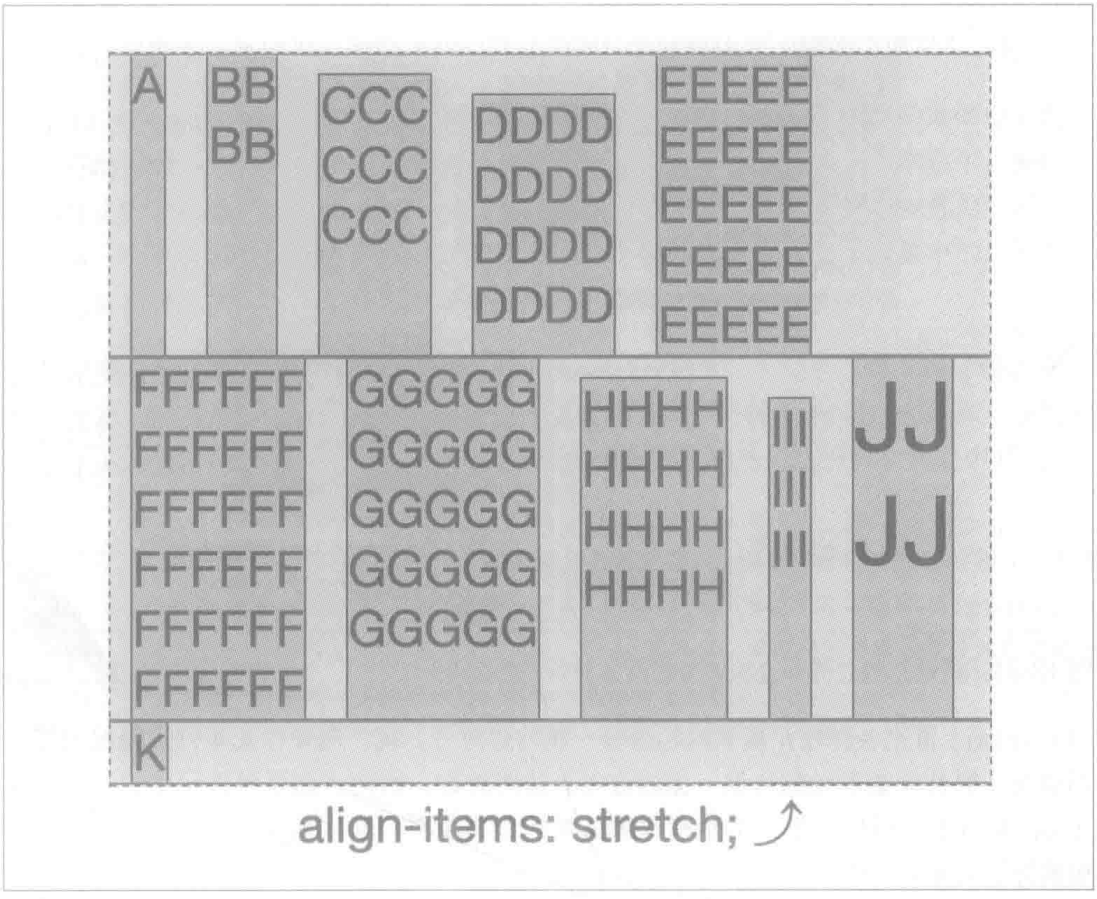


图 12-30：对齐方式为 stretch

C、D、H 和 I 之所以看上去比行中的其他弹性元素小，就是因为它们有外边距。其实，它们并不小。它们占据的空间是从上下外边距的外边界开始算起的，也与弹性元素行的垂轴起边和垂轴终边靠拢。每一个弹性元素行的高度都与行中最高的弹性元素一样高，如果垂轴是横向的，则与行中最宽的弹性元素一样宽。

弹性元素行的高度或宽度是正好能放下弹性元素的尺寸。在展示 `align-items` 属性五个值的效果那张插图中，只有 K 所在的弹性元素行比其他两行明显小一些。

12.6.1 起边、终边和居中对齐

起边、终边和居中对齐所用的值及其效果相对简单，因此放在一起讲。

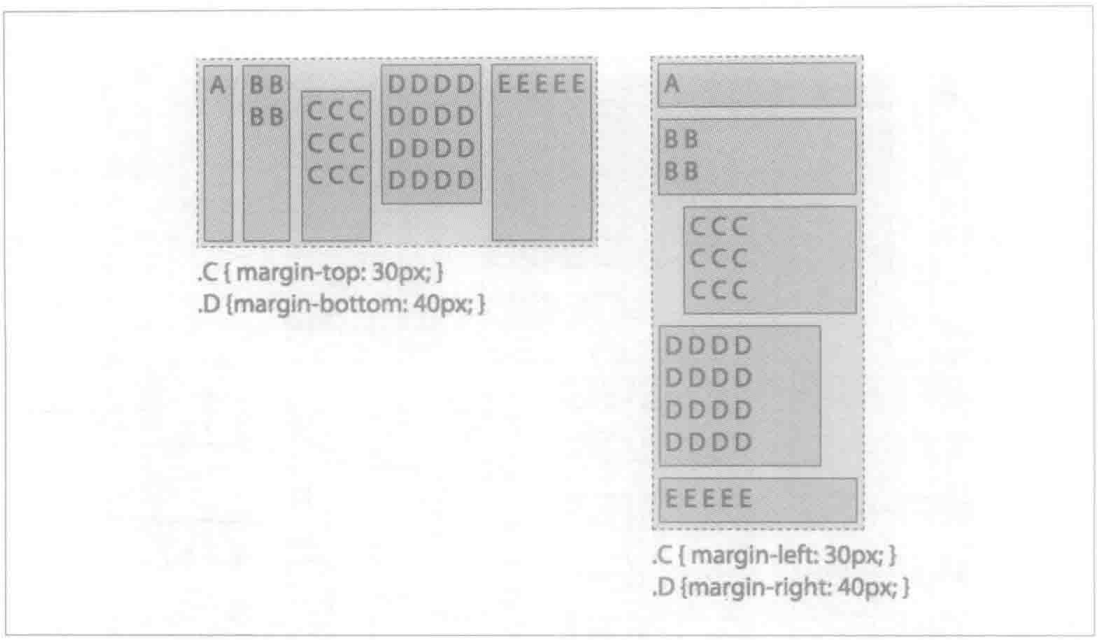


图 12-31：垂轴方向上的外边距对弹性元素对齐的影响 ▶

`flex-start` 值把各弹性元素垂轴起边那一侧的边对齐，统一向弹性元素行垂轴起边那一侧靠拢。弹性元素垂轴起边那一侧的边在外边距外部，因此，如果弹性元素的外边距大于 0，它将不与弹性元素行垂轴起边那一侧的边紧靠在一起，如图 12-32 中第一个例子里的弹性元素 C、D、H 和 I 所示。

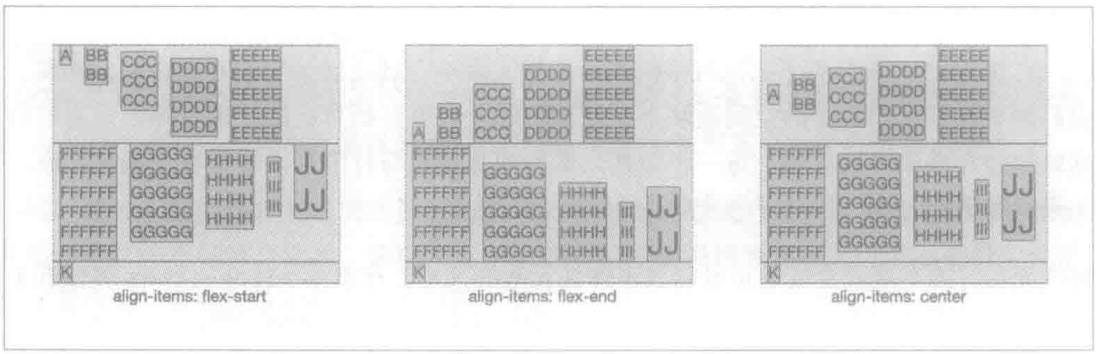


图 12-32：对齐方式为 flex-start、flex-end 和 center

设为 `align-items: flex-end` 时，每个弹性元素垂轴终边那一侧的边将向弹性元素行垂轴终边那一侧的边靠拢，如图 12-32 中的第二个例子所示。示例中的弹性元素，没有一

个的下外边距大于 0 像素，因此与其他例子不同的是，这些弹性元素不会显得参差不齐，所有弹性元素垂轴终边那一侧的边都与弹性元素行垂轴终边那一侧的边紧靠在一起。

如图 12-32 中的第三个例子所示，设为 `align-items: center` 时，弹性元素的中点将与所在行的垂轴中点对齐。弹性元素的中点指外边距外边界之间的中点。别忘了，弹性元素的外边距不折叠。因为 C、D、H 和 I 在垂轴方向上的两个外边距是不对称的，所以视觉上看，它们在垂轴上是不居中的。其实，这几个弹性元素在垂轴上是居中的，因为上下外边距边界之间的中点正好与弹性元素行上下边界之间的中点对齐。

在从左至右和从右至左书写的语言中，对 `flex-direction: row` 和 `row-reverse` 来说，弹性元素上对齐用的中点是上下外边距边界之间的中间点。对 `flex-direction: column` 和 `column-reverse` 来说，弹性元素上对齐用的中点是左右外边距边界之间的中间点。



如果对弹性容器的垂轴尺寸有限制，内容可能会从弹性容器的垂轴起边和（或）垂轴终边那一侧溢出。从哪边溢出不由 `align-items` 属性决定，而受 `align-content` 属性的控制（见 12.8 节）。`align-items` 属性定义弹性元素行中各弹性元素的对齐方式，对弹性元素溢出弹性容器的方向没有直接影响。

12.6.2 基线对齐

`baseline` 值有点复杂。设为 `baseline` 时，一行中的弹性元素向第一条基线对齐。弹性元素行中，基线与垂轴起边那一侧外边距边界之间距离最远的弹性元素，其外边距的外边界与弹性元素行垂轴起边那一侧的边对齐，其他弹性元素的基线则与那个弹性元素的基线对齐。

请看图 12-33 中的第二行，这里 J 起主导作用。我们把 J 的字号增大为 3rem，为的是把弹性元素中第一行文本变得更高。可见，J 的顶边（垂轴起边那一侧的边）与弹性元素行的顶边（垂轴起边那一侧的边）是对齐的。行中的其他弹性元素将下移，直到第一行文本的基线与 J 的第一条基线对齐为止（这条基线的位置在图中用绿线表示）。

现在看一下第一行，即以 A 开头的那一行。可以看出，A、B、C、D 和 E 都向顶边对齐，但是仔细观察之后你会发现，它们并没有紧靠弹性元素行的顶边。这是因为 D 有 20 像素的上外边距，上外边距（垂轴起边那一侧）的外边界是与弹性元素行垂轴起边那一侧的边对齐的。前面说过，垂轴起边与基线之间的距离由行中垂轴起边那一侧的外边距边

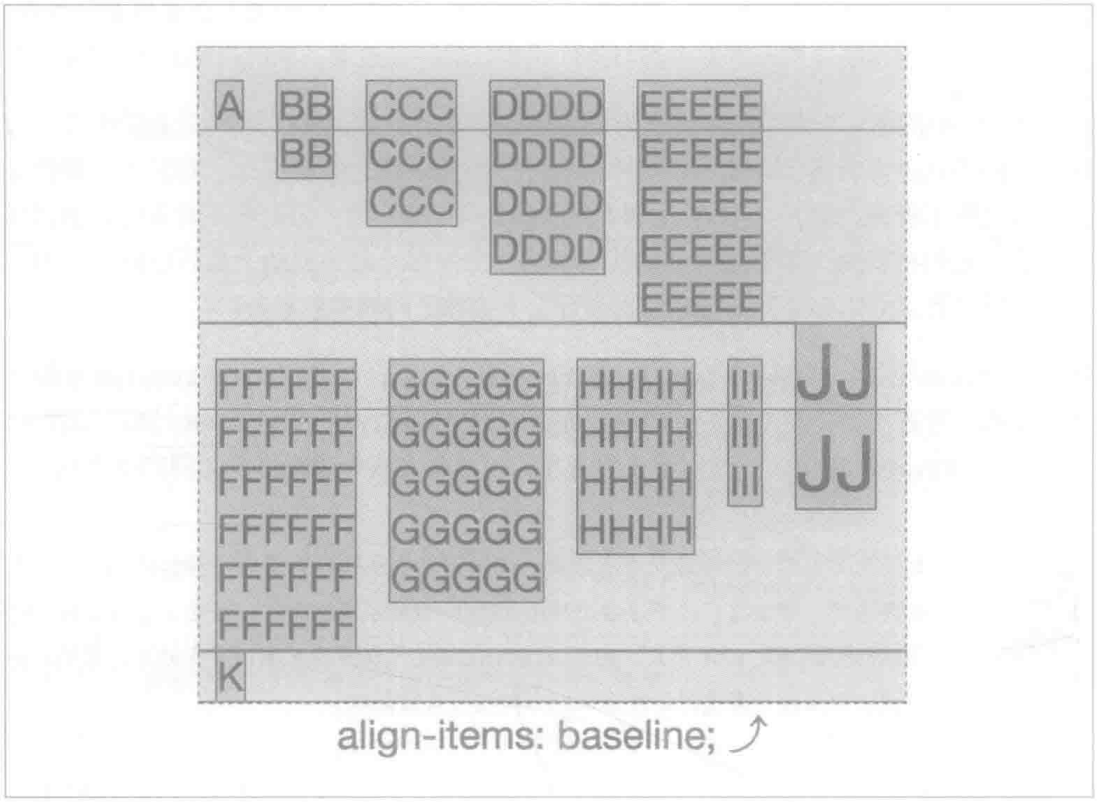


图 12-33：对齐方式为 baseline ▶

界与基线之间的最大的距离决定。因此，D 的位置决定了它的基线将作为其他弹性元素对齐的基准（因为它有上外边距）。

很多情况下，baseline 的效果看起来与 flex-start 类似。例如，倘若 D 没有上外边距，那么第一行中的所有弹性元素在视觉上都将紧靠弹性元素行的顶边，效果就跟使用 flex-start 一样。只要弹性元素在垂轴起边一侧的外边距、边框、内边距，或者字号或行高不同，flex-start 和 baseline 之间就会出现差异。

有一种情况能把 baseline 的效果变成与 flex-start 一模一样，即基线与垂轴平行时。例如，为图 12-33 中的弹性容器设置 flex-direction: column，此时垂轴与英语文本的基线一样，是横向的。因为没有办法从列在垂轴起边那一侧的边（左边）向里偏移，所以 baseline 的效果就像使用 flex-start 一样。

12.6.3 补充说明

如果想改变某个或某些弹性元素的对齐方式，而不是全部修改，为相应的元素设置 align-self 属性。这个属性的取值与 align-items 一样，参见 12.9 节。

匿名弹性元素（弹性容器中非空的文本节点子元素）的对齐方式无法覆盖，其 align-self 值始终与父级弹性容器的 align-items 值一致。

在上述 align-items 示例中，弹性容器的垂轴尺寸要多高有多高。因为没有为弹性容器设置 height，所以默认为 height: auto。因此，弹性容器的尺寸将根据内容而变。你可能注意到了，在所有示例中，弹性容器的高度都是一样的，而且弹性元素行的高度也是一样的。

如果指定了垂轴尺寸（对上述示例来说是高度），在垂轴终边一侧可能有额外的空白，也可能空间不足，放不下内容。为了控制弹性元素行的对齐方式，弹性盒规范提供了 align-content 属性。这是我们要讨论的最后一个应用在弹性容器（而非弹性元素）上的属性。align-content 属性只影响分为多行的弹性容器中的弹性元素行的对齐方式。

12.7 align-self 属性

看着有点早，但现在是讨论 align-self 属性的最佳时刻。这个属性在单个弹性元素上覆盖 align-items 属性的值。

align-self	
取值	auto | flex-start | flex-end | center | baseline | stretch
初始值	auto
适用于	弹性元素
继承性	否
百分数	不适用
动画性	否

align-items 属性在弹性容器上设置，定义弹性容器中所有弹性元素的对齐方式。但是，单个弹性元素的对齐方式可以使用 align-self 属性覆盖。align-items 的默认值是 stretch，因此在图 12-34 中的五个示例里，除第二个弹性元素之外，其他弹性元素的高度都与所在的行一样高。

对所有 align-self 属性为默认值 auto 的弹性元素来说，其对齐方式继承自容器的 align-items 属性。但是每个示例中的第二个弹性元素例外，它的 align-self 值标在每个示例的下方。

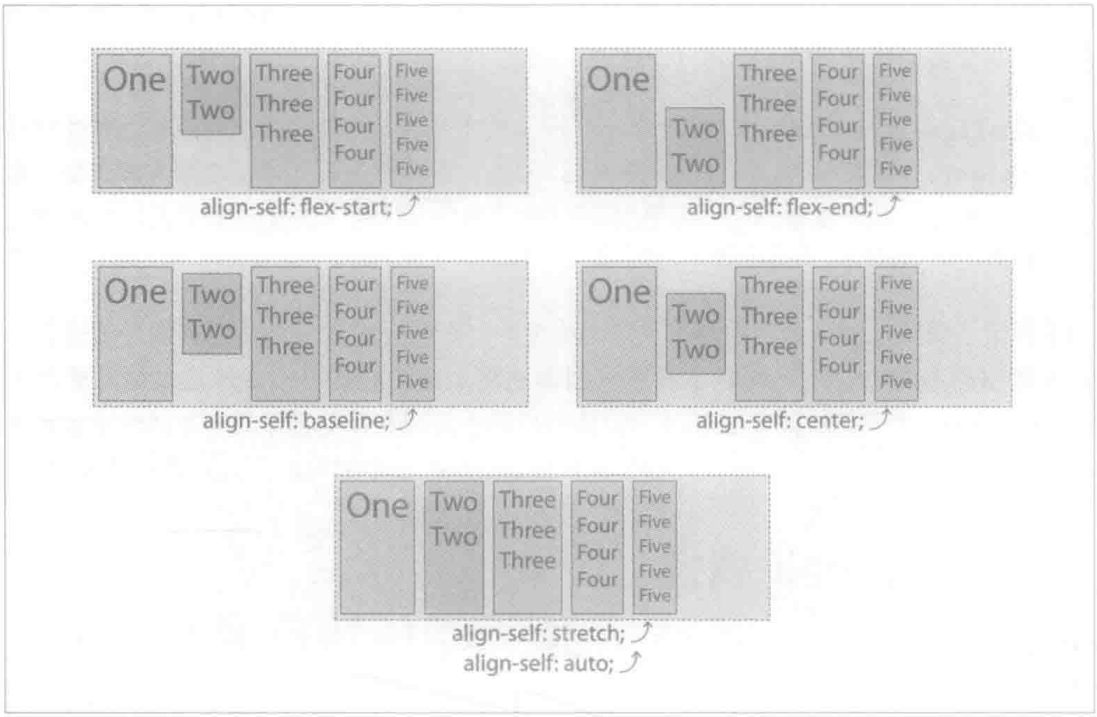


图 12-34：修改弹性元素的对齐方式 ▶

与 align-items 属性的值一样，flex-start 值把弹性元素放在垂轴起边，flex-end 把弹性元素放在垂轴终边，center 把元素与垂轴的中点对齐，baseline 把弹性元素的基线与所在行中最低的基线对齐。最后，auto 和 stretch 都拉伸弹性元素，效果与 align-items 的默认值 stretch 一样（类似地，align-self: inherit 得到的也是拉伸对齐）。

如果想进一步了解 flex-start、flex-end、center、baseline 和 stretch 等值的效果，参阅 12.6 节。

12.8 对齐内容

align-content 属性定义弹性容器有额外的空间时在垂轴方向上如何对齐各弹性元素行，以及空间不足以放下所有弹性元素行时从哪个方向溢出。

align-content

取值	flex-start | flex-end | center | space-between | space-around | space-evenly | stretch
初始值	stretch
适用于	分为多行显示的弹性容器
计算值	指定的值
继承性	否
动画性	否

align-content 属性指定弹性容器中垂轴方向上的额外空间如何分配到弹性元素行之间和周围。虽然 align-content 与前面讨论的 align-items 属性在取值和相关概念上是一样的，但是二者的作用不同，后者指定的是每一行中弹性元素的定位方式。

align-content 的作用与 justify-content 类似，后者在弹性容器的主轴方向上对齐各个弹性元素，而前者在弹性容器的垂轴方向上对齐各弹性元素行。align-content 属性只适用于分为多行显示的弹性容器，对禁止换行及只有一行的弹性容器没有影响。

以下述 CSS 为例，假设弹性元素没有外边距：

```
.flex-container {
  display: flex;
  flex-flow: row wrap;
  align-items: flex-start;
  border: 1px dashed;
  height: 480px;
  background-image: url(banded.svg);
}
.flex-items {
    margin: 0;
```

```
        flow: 1;
    }
```

在上述 CSS 的基础上分别使用 align-content 属性的七个值，得到的结果如图 12-35 所示。每个示例中都有三个弹性元素行，各行在垂轴起边和垂轴终边一侧的边分别使用红线和蓝线表示。弹性容器中余下的空间，即弹性元素行之间或周围的空间以条纹区域表示。

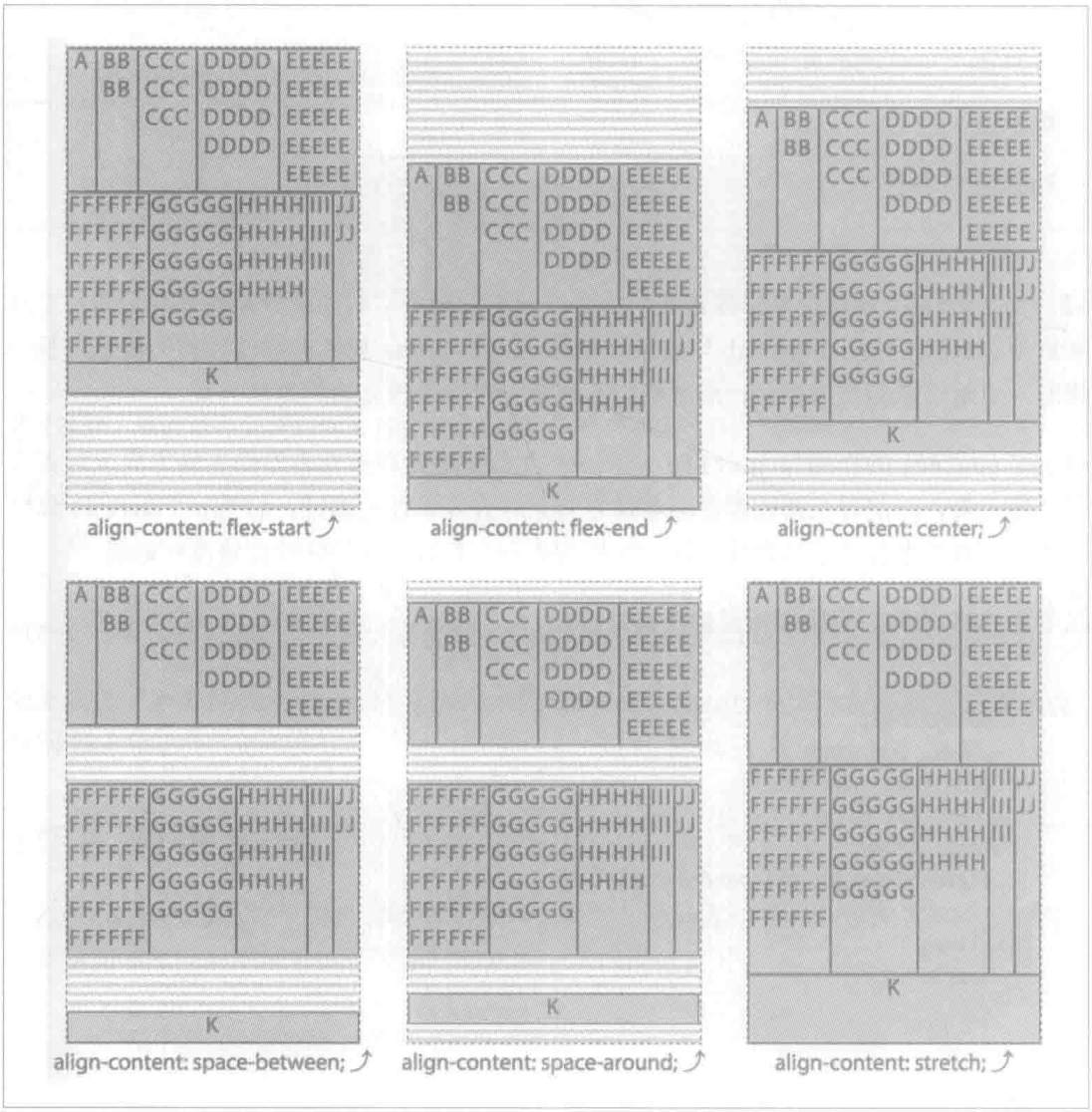


图 12-35：align-content 属性的各个值对额外空间的分配情况

弹性容器的高度为480像素，默认情况下，3个弹性元素行的高度之和要比480像素小。假设每一行中最高的弹性元素，即E、F和K，高度分别为150像素、180像素和30像素，那么三者之和是360像素。因此，每个弹性容器在垂轴方向上有120像素的额外空间。

align-content的其中五个值把额外的空间分布到弹性元素行外部，如图12-35所示。这与justify-content属性对应的值行为类似，只不过现在是沿垂轴而非（justify-content属性使用的）主轴对齐。但是设为stretch值时，额外的空间将平均分配到每个弹性元素行中，增加行的垂轴尺寸，直到边与边接触。

在上述示例中，弹性容器的高度为480像素，垂轴方向上“余下”120像素空间，这些空间根据align-content属性的值采用不同的方式分配。

如图12-35中的上部三个示例所示，设为flex-start时，120像素的额外空间全放在垂轴终边一侧；设为flex-end时，120像素的额外空间全放在垂轴起边一侧；设为center时，各行居中显示，垂轴起边和终边一侧分别放置60像素（120像素的一半）的额外空间。

设为space-between时，相邻的弹性元素行之间有60像素的空间。而设为space-around时，空间平均分配在各行周围：120像素分成3份，因为有3个弹性元素行。因此，每个弹性元素行的垂轴起边和终边一侧各有20像素（40像素的一半）不折叠的空间，所以在弹性容器垂轴起边和终边一侧各有20像素的额外空间，在相邻的弹性元素行之间有40像素的空间。

设为space-evenly时，要插入四个空间：各弹性元素行前面有一个，最后一行后面还有一个。如果有三行，空间将分成四份，每份30像素，垂轴起边和终边一侧各有30像素，相邻的弹性元素行之间也有30像素。

stretch值则不同，此时额外的空间分配给各弹性元素行，拉伸各行的尺寸，而不是放在行之间。这里，每一行将增加40像素，三行的高度增加相同的量，而不是按比例分配。从图12-35中的第六个示例可以看到，弹性容器中没有额外空间，被各弹性元素行占满了。stretch是默认值，因为你有可能想占满所有可用空间。

如果空间不足以放下全部行，弹性元素行将从垂轴起边、垂轴终边或同时从两边溢出，具体情况取决于align-content属性的值。各种情况如图12-36所示，其中边为点划线、背景为浅灰色的方框是尺寸不足的弹性容器（为了便于说明align-content的效果，还设置了align-items: flex-start）。

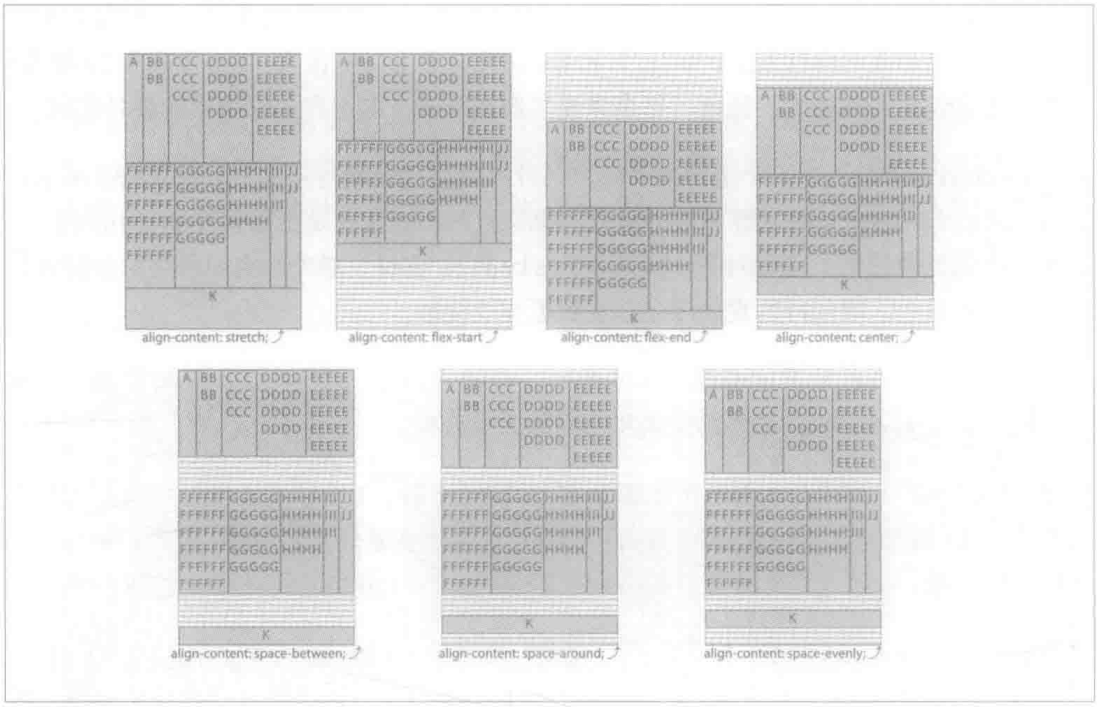


图 12-36：align-content 属性各可用值下弹性元素行的溢出方向

这些示例与图 12-35 中的示例之间唯一的区别是弹性容器的高度。这里，弹性容器的高度减小为 240 像素，因此不足以放下全部弹性容器行（你应该还记得，各行的高度之和为 360 像素）。

如果弹性元素行从弹性容器中溢出，设为 `align-content: flex-start`、`space-between` 和 `stretch` 时，将从垂轴终边一侧溢出，而设为 `align-content: space-around` 和 `center` 时，将同时从垂轴起边和终边一侧溢出，而且溢出量相同。只有设为 `align-content: flex-end` 时，弹性元素行才从垂轴起边一侧溢出。

注意，这些值不是与上下边绑定的。如果垂轴是从下指向上方的，那么 `align-content: flex-start` 将从底部向上对齐弹性元素行，因此可能从上边（垂轴终边一侧）溢出。如果弹性元素是列向排布的，垂轴将是横向的，因此垂轴起边和终边分别为弹性容器的右边和左边。

间隔、周围和平均分配

`space-between` 和 `space-around` 对弹性元素行对齐方式的影响需要深入讨论一下。

设为 `align-content: space-between` 时，弹性元素行在弹性容器中均匀分布。这里所

说的“均匀分配”是针对可用空间，而不是行的尺寸。如果有多个弹性元素行，第一行将向弹性容器的垂轴起边靠拢，最后一行将向弹性容器的垂轴终边靠拢，余下的可用空间（如果有的话）则均匀分布在其他行之间。额外的空间是平均分布的，而不是按比例分布。弹性容器中任何两个弹性元素行之间的空间都是等量的，即使弹性元素行在垂轴方向上的尺寸不同也是如此。此外，如果行数为奇数，中间那行不一定居中在弹性容器中，因为不是所有行在垂轴方向上的尺寸都一样。



仅当弹性容器中有多行弹性元素时，垂轴上的额外空间才会按一定方式分配。如果只有一行，`align-content` 属性对内容的分布没有影响。如果弹性容器中只有一行弹性元素，那一行的尺寸将被拉伸，占满全部可用空间。

任何两个相邻的行之间的空间其实是相等的。假设与前一节一样，有 3 行，额外空间总计 120 像素。第一个弹性元素行向垂轴起边一侧靠拢，第三个弹性元素行向垂轴终边一侧靠拢。因为二者之间还有一行，所以共有两处间隙。余下的 120 像素空间分成两等份，均为 60 像素。一份 60 像素的空间放在第一行和第二行之间，另一份则放在第二行和第三行之间，如图 12-37 所示。

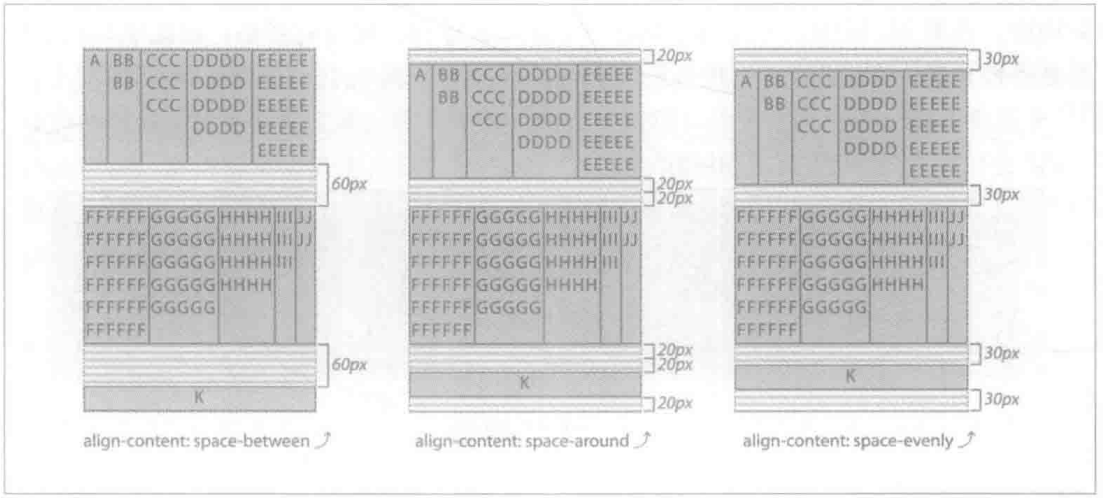


图 12-37：额外空间在 space-between、space-around 和 space-evenly 下的分布情况

`space-around` 值均匀分布多行弹性容器中的各个弹性容器行，好似垂轴起边和垂轴终边两侧都有不折叠的等量外边距一样。因为额外的空间是等量分配到每一行周围的，因此容器边界与第一行和最后一行之间的空间是行与行之间空间的一半。额外空间的分布情况如图 12-37 所示。



从2017年年末开始，flex-start、flex-end等对齐值变得更具一般性了，改成了start、end等等。这样改的目的是为了让CSS更多地考虑书写方向和布局方向。比如，这次改变增加了margin-start和padding-end等属性。这些变化还没定案（也没得到广泛支持），因此本书这一版没有讨论，不过要留意一下。

目前，我们所讲的属性大多用于弹性容器（align-self除外），现在该介绍直接应用于弹性元素上的属性了。

12.9 弹性元素

前面几节介绍了如何通过弹性容器的样式整体排布弹性元素。此外，弹性盒布局规范还提供了几个直接应用于弹性元素的属性。利用这些专门针对弹性元素的属性，可以更加细致地控制弹性容器中的单个子元素。

12.9.1 弹性元素是什么

为有子节点的元素声明display: flex或display: inline-flex即可创建弹性容器。弹性容器的子代称为弹性元素，不管是子元素，还是元素之间非空的文本节点，或是生成的内容。在图12-38中，各元素中只有一个字母，单词之间还有空格，因此字母和空格都是弹性元素。

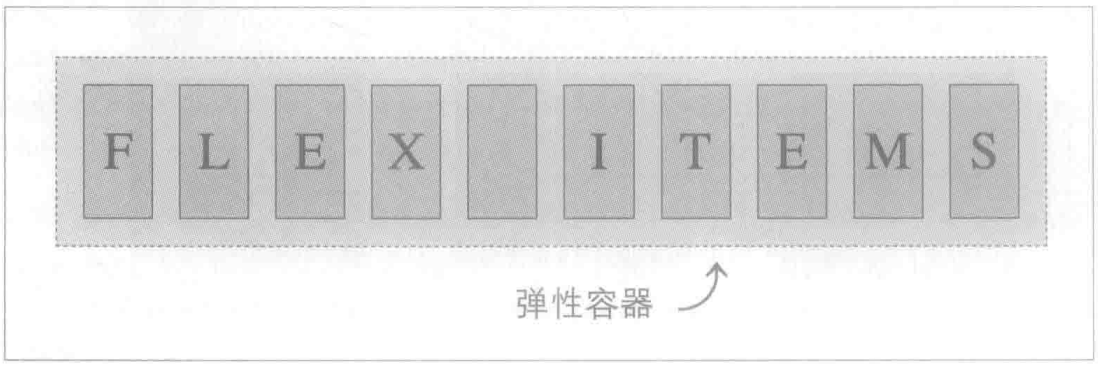


图12-38：子节点是弹性元素，父节点是弹性容器 ▶

对弹性容器中的文本子节点来说，如果文本节点不是空的（内容不是空白），将放在一个匿名弹性元素中，其行为与其他同辈弹性元素一样。虽然匿名弹性元素与同辈DOM节点一样，将继承在弹性容器上设置的相关弹性属性，但是不能直接使用CSS装饰。因此，

不能直接在匿名弹性元素上设置针对弹性元素的属性。所以，在下述标记中，两个元素（ 和 ）及文本“ they're what's for ”都是弹性元素，共有三个弹性元素：

```
<p style="display: flex;">
    <strong>Flex items:</strong> they're what's for <em>&lt;br&gt;fast!</em>
</p>
```

生成的内容可以直接装饰（通过 ::before 和 ::after），因此本节讨论的全部属性都能应用到生成的内容上。

弹性容器中只有空白的文本节点将被忽略，就像把 display 属性设为 none 一样，如下述代码示例所示：

```
nav ul {
  display: flex;
}
<nav>
  <ul>
    <li><a href="#1">Link 1</a></li>
    <li><a href="#2">Link 2</a></li>
    <li><a href="#3">Link 3</a></li>
    <li><a href="#4">Link 4</a></li>
    <li><a href="#5">Link 5</a></li>
  </ul>
</nav>
```

在上述代码中，因为 display 属性的值是 flex，所以无序列表是弹性容器，而它的子元素，即那些列表项目是弹性元素。作为弹性元素，列表项目是弹性级框体，虽然看起来不像列表项目，但是语义上仍是。那些列表项目不再是块级框了，因为它们现在身处容器的弹性格式化上下文中。li 元素之间及两侧的空白（换行符及缩进用的制表符和 / 或空格）将完全被忽略。链接不是弹性元素，而是变成弹性元素的列表项目的后代元素。

12.9.2 弹性元素的特性

弹性元素的外边距不折叠。float 和 clear 属性对弹性元素不起作用，不会把弹性元素移出文档流。其实，应用到弹性元素上的 float 和 clear 将被忽略（然而，float 属性对框体的生成仍有影响，因为 display 属性的计算值受它影响）。来看下述代码：

```
aside {
  display: flex;
}
img {
```

```
  float: left;
}
<aside>
    <!-- 一条注释 -->
    <h1>Header</h1>

    <img src="images/foo.jpg" alt="Foo Master">
    Some text
</aside>
```

在这个示例中，aside 是弹性容器。注释和只有空白的文本节点被忽略。内容为“some text”的文本节点将放在一个匿名弹性元素中。标题、图像和内容为“some text”的文本节点都是弹性元素。因为图像是弹性元素，应用其上的 float 属性将被忽略。

虽然图像和文本节点是行内节点，但是变成弹性元素后，只要没有绝对定位，就是块级的。

```
aside {
  display: flex;
  align-items: center;
}
<aside>
    <!-- 一条注释 -->
    <h1>Header</h1>

    <img src="images/foo.jpg" alt="foo master">
    Some text <a href="foo.html">with a link</a> and more text
</aside>
```

上述示例中的标记与第二个示例类似，不过非空的文本节点中多了个链接。此时，得到的弹性元素有 5 个。注释和只有空白的文本节点被忽略。标题、图像、链接之前的文本节点、链接，以及链接之后的文本节点都是弹性元素，如图 12-39 所示。

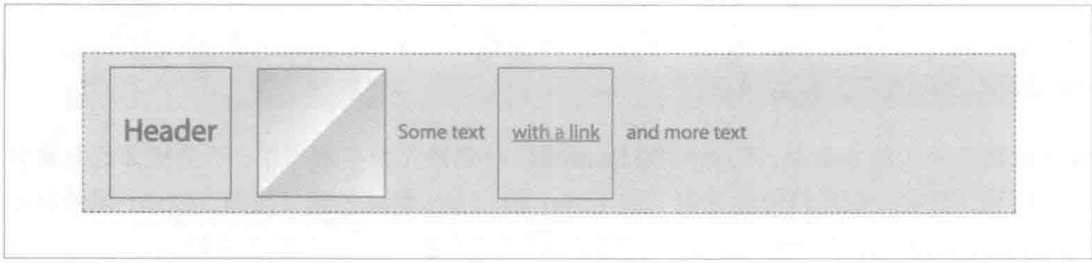


图 12-39：aside 中的 5 个弹性元素 ▶

内容为“some text”和“and more text”的文本节点将放在匿名弹性元素中，在图 12-39 中以没有背景的虚线框表示（加上虚线是为了告诉你匿名弹性元素在哪里）。标题、图像和链接是真正的 DOM 节点，可以直接使用 CSS 装饰。匿名弹性元素不能直接选择，因此只能从弹性容器上继承样式。

此外，vertical-align 对弹性元素没有影响，只用于设定弹性元素中文本的对齐方式。也就是说，应用在弹性元素上的 vertical-align: bottom 将把弹性元素中的文本对齐到行框底部，但是不会把弹性元素推向所在容器的底部（align-items 和 align-self 才有这样的效果）。

绝对定位

我们知道 float 不会浮动弹性元素，但是 position: absolute 则不一样。如果绝对定位弹性容器的子元素，与绝对定位普通元素一样，将从文档流中移除。

除此之外，绝对定位的弹性元素不再参与弹性布局，因为它们已经不在文档流中。然而，这些元素将受应用在弹性容器上的样式影响，就像子元素受普通父元素（非弹性容器）的样式影响一样。除了可继承的属性被继承之外，应用到弹性容器上的属性可能还会影响定位原点。

绝对定位的弹性容器的子元素既受弹性容器的 justify-content 值影响，也受自身 align-self 值（如果设定了）的影响。例如，如果在绝对定位的子元素上设定 align-self: center，元素将相对弹性容器的垂轴居中，然后再使用 top、bottom、外边距等属性移动其位置。

order 属性（12.16 节说明）对弹性容器中绝对定位的子元素的位置没有影响，但是对同辈元素的绘制顺序有影响。

12.9.3 最小宽度

在图 12-40 中可以看到，设为默认值 nowrap 的弹性元素行从弹性容器中溢出了。这是因为对弹性元素来说，未设定 min-width 时，默认为 auto，而不是 0。最初，规范规定，如果弹性元素在唯一的主轴上放不下，其尺寸将缩减。然而，出现弹性元素后，min-width 的规范改了（以前，min-width 的默认值是 0。参见 *https://drafts.csswg.org/css2/visudet.html#min-max-widths*）。

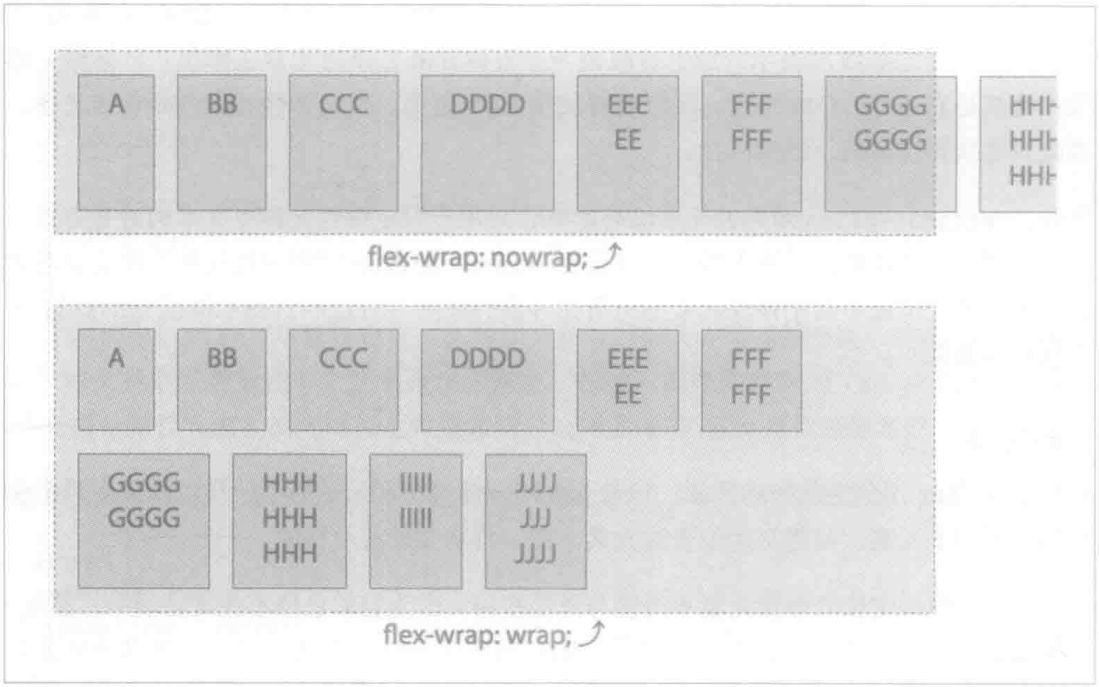


图 12-40：设置了最小宽度的弹性元素从弹性容器中溢出 ▶

如果设置的 min-width 值比 auto 的计算值小，例如声明 min-width: 0，那么不换行那个示例中的弹性元素（在某些情况下）将比实际内容的宽度小。如果允许换行，弹性元素的宽度将正好能放下内容，不会比这更窄。这两种情况如图 12-41 所示。

12.10 适用于弹性元素的属性

虽然弹性元素的对齐方式、顺序和弹性（flexibility）在一定程度上受在弹性容器上设置的属性控制，但是有些属性可以应用到单个弹性元素上，以便进行更细致地控制。

简写属性 flex，以及构成它的 flex-grow、flex-shrink 和 flex-basis 属性用于控制弹性元素的弹性。这里所说的弹性是指弹性元素在主轴方向上可以增加或缩减多少尺寸。

12.11 flex 属性

弹性布局最突出的一点是能把弹性元素变得具有“弹性”，即在主轴方向上调整弹性元素的宽度或高度，占满可用空间。弹性容器根据弹性增长因子（flex grow factor）按比例分配额外的空间，或者根据弹性缩减因子（flex shrink factor）按比例缩小弹性元素，以防溢出（稍后探讨这些概念）。

增长因子和缩减因子由弹性元素的 flex 简写属性声明，或者由构成简写属性的单个属性

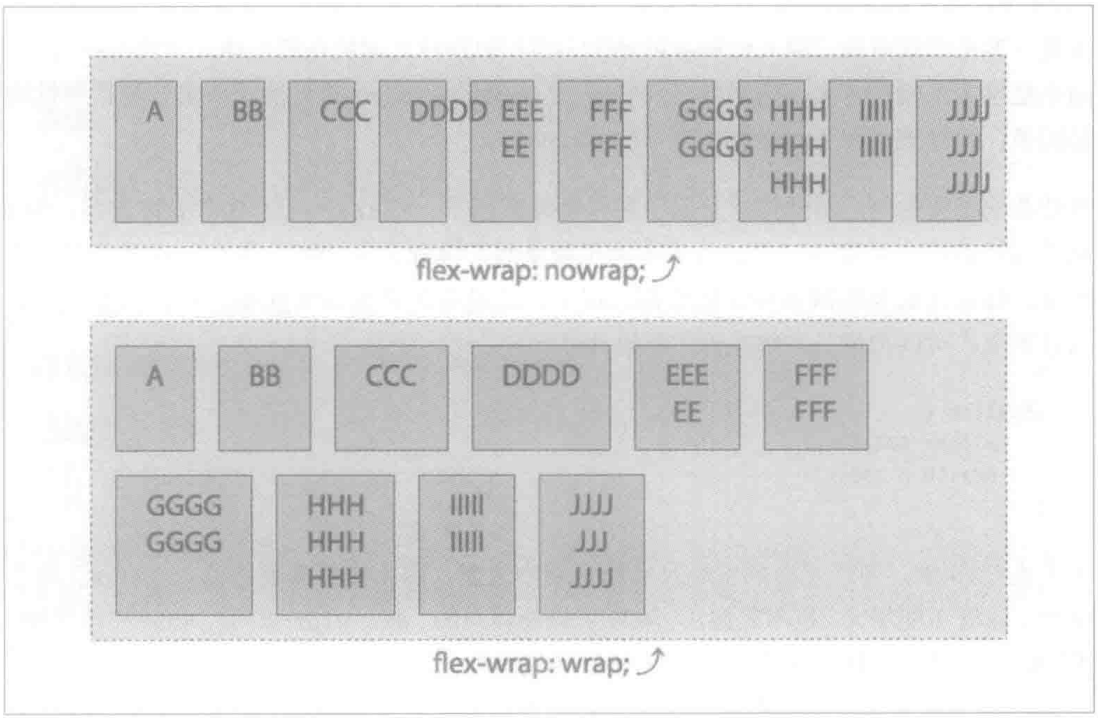


图 12-41：不换行和换行的弹性容器中最小宽度为零的弹性元素 ▶

定义。如果空间过多，可以增加弹性元素的尺寸，占满多出的空间。当然也可以不这么做。如果指定的尺寸或默认的尺寸导致弹性元素在弹性容器中放不下，可以按比例缩减尺寸，以便放得下。当然，此时也可以不这么做。

这一切都由 flex 属性控制，它是 flex-grow、flex-shrink 和 flex-basis 的简写形式。虽然三个子属性可以单独使用，但是强烈建议始终使用简写的 flex，原因稍后说明。

flex

取值	[<flex-grow> <flex-shrink>? || <flex-basis>] | none
初始值	0 1 auto
适用于	弹性元素（弹性容器的子元素）
百分数	只能作为 flex-basis 的值，相对父元素内的主轴尺寸计算
计算值	参见各单独属性
继承性	否
动画性	参见各单独属性

flex 属性指定确定弹性长度的各要素。这里所说的“长度”是弹性元素在主轴方向上的长度（参见本章前面“深入理解各种轴”一节）。弹性元素所在的框体尺寸由 flex 决定，而不是确定主轴尺寸大小的属性（height 或 width）。flex 属性的“要素”包括弹性增长因子、弹性缩减因子和弹性基准（flex basis）。

弹性基准指明如何应用弹性增长因子和弹性缩减因子。flex-basis 这个名称表明，flex 简写属性的这个要素决定着，为了占满可用空间，弹性元素能增长多少，或者在空间不足时，弹性元素要缩减多少才能全部放得下。这是各弹性元素的初始尺寸，如果想限制尺寸不变，可以把增长和缩减因子都设为 0：

```
.flexItem {
    width: 50%;
    flex: 0 0 200px;
}
```

在上述 CSS 中，弹性元素的主轴尺寸就是 200 像素，因为弹性基准的值是 200px。这个弹性元素既不能增大，也不能缩小。假设主轴是横向的，那么 width 的值（50%）将被忽略。类似地，如果主轴是纵向的，height 的值将被忽略。



对 height 和 width 的覆盖不受层叠规则约束，因此即使为弹性元素的 height 或 width 值添加 !important 也不会覆盖弹性基准。

如果选择符的目标不是弹性元素，flex 属性不起任何作用。

若想正确使用 flex 简写属性，必须理解它的三个要素。

12.12 flex-grow 属性

flex-grow 属性定义有多余的空间时是否允许弹性元素增大，以及允许增大且有多余的空间时，相对其他同辈弹性元素以什么比例增大。



规范制定人员极度不鼓励通过 flex-grow 属性定义增长因子，而是建议在 flex 简写属性中指定增长因子。我们单独讨论这个属性只是为了说明它的作用。

flex-grow

取值	*<number>*
初始值	0
适用于	弹性元素（弹性容器的子元素）
计算值	指定的值
继承性	否
动画性	是

`flex-grow` 的值始终是一个数字。负数无效。如果愿意，可以不使用整数，只要大于或等于零即可。这个属性的值设定弹性增长因子，决定分配弹性容器的可用空间时相对其他同辈弹性元素能增大多少。

如果弹性容器中有多余的空间，多出的空间将根据各弹性元素的非零增长因子按比例分配给各个弹性元素。

例如有个宽度为 750px 的横排弹性容器，里面有三个弹性元素，而且都设置了 `width: 100px`。因此，弹性元素共占 300 像素空间，“余下”450 像素的空间（因为 750 - 300 = 450）。这是图 12-42 中的第一种情况。此时，所有弹性元素都不允许增大。

在图 12-42 中的第二种情况里，只为一个弹性元素（第三个）设定了增长因子。我们声明的是 `flex-grow: 1`，不过可以使用浏览器能理解的任何正数。此时，两个弹性元素没有增长因子，而第三个弹性元素有，因此所有可用空间都将分给设定了增长因子的那个弹性元素。所以，450 像素的可用空间都将添加到第三个弹性元素上，最终宽度为 550 像素。样式中为第三个弹性元素声明的 `width: 100px` 将被覆盖。

在第三种情况和第四种情况中，虽然设定的弹性增长因子不同，但是得到的弹性元素宽度是一样的。先看第三种情况，这里各弹性元素的增长因子为 1、1 和 3。三个因子之和为 5。各因子除以总和，得到比例。这三个因子除以 5 后得到 0.2、0.2 和 0.6。

这些比例乘以可用空间，得到增大的量。因此：

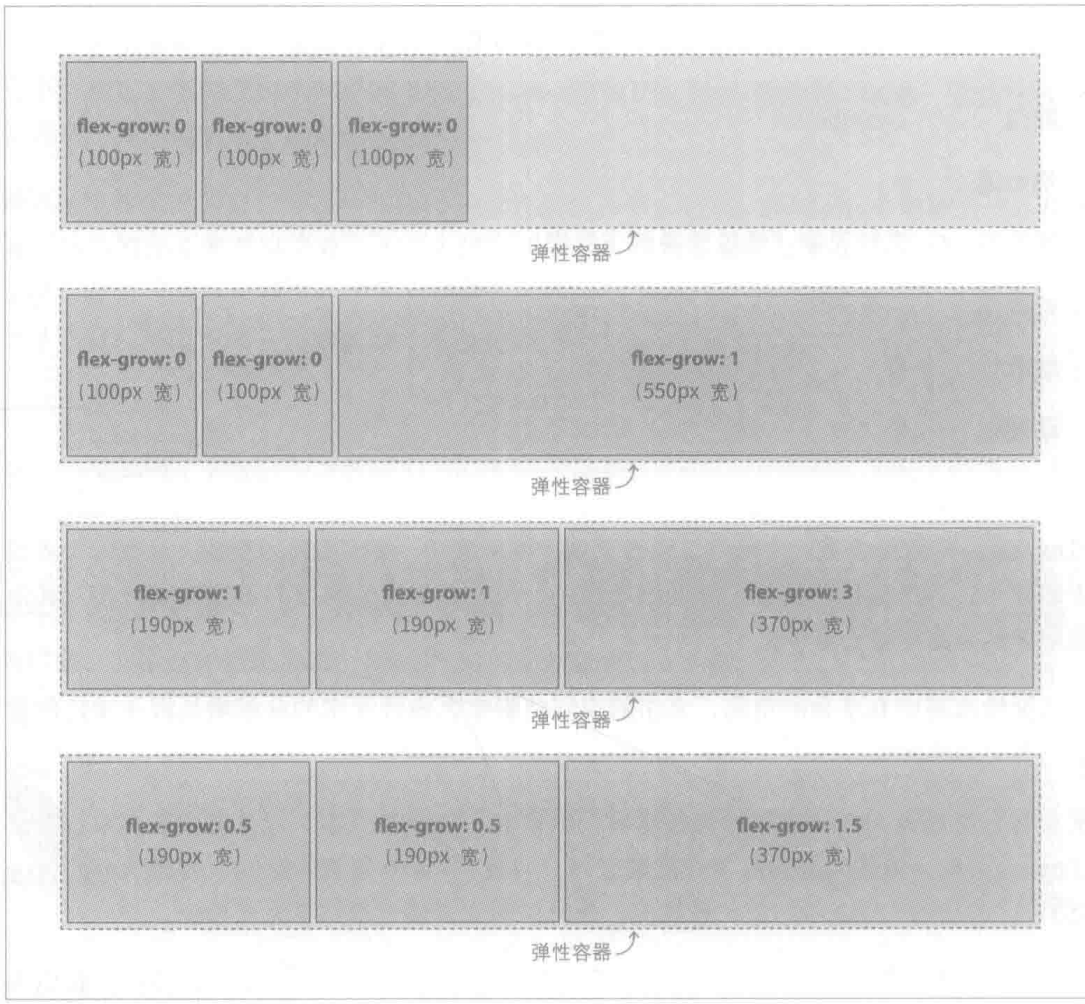


图 12-42：flex-growth 设为不同值的情况 ▶

1. 450 px × 0.2 = 90 px
2. 450 px × 0.2 = 90 px
3. 450 px × 0.6 = 270 px

这些就是增加到弹性元素 100 像素初始宽度上的量。因此，最终得到的宽度分别为 190 像素、190 像素和 370 像素。

第四种情况得到的结果与此相同，因为比例是一样的。假设增长因子变成了 0.5、1 和 1.5。经过计算，第一个弹性元素获得可用空间的六分之一，第二个弹性元素获得三分之一，

而第三个弹性元素获得一半。那么最终得到的弹性元素的宽度分别为175、250和425像素。如果声明的增长因子是0.1、0.1和0.3，或者是25、25或75，抑或任何最终能化简为1:1:3的组合，得到的结果都是一样的。

12.9.3节说过，如果没有设定宽度或弹性基准，弹性基准默认为auto，即各弹性元素的基准为内容不换行时的宽度。auto是个特殊的值，默认为content，但是如果设定了宽度，弹性基准为宽度。auto值在12.14.2节讨论。如果这个示例没有设定宽度，而字号又相当小，那么主轴方向上可供分配的空间将超过450像素。



弹性元素的主轴尺寸受可用空间、所有弹性元素的增长因子和当前元素的弹性基准影响。我们还没介绍弹性基准，不过也快了。

下面来看弹性元素的width值和增长因子都不同的情况。在图12-43中的第二个例子里，弹性元素的宽度分别为100像素、250像素和100像素，增长因子分别为1、1和3，容器的宽度为750像素。因此，多出的300像素（因为750 − 450 = 300）空间将分成5份，每一份占60像素（300 ÷ 5）。这意味着，flex-grow值为1的第一个和第二个弹性元素均增加60像素，而最后一个弹性元素将增加180像素，因为它的flex-grow值为3。

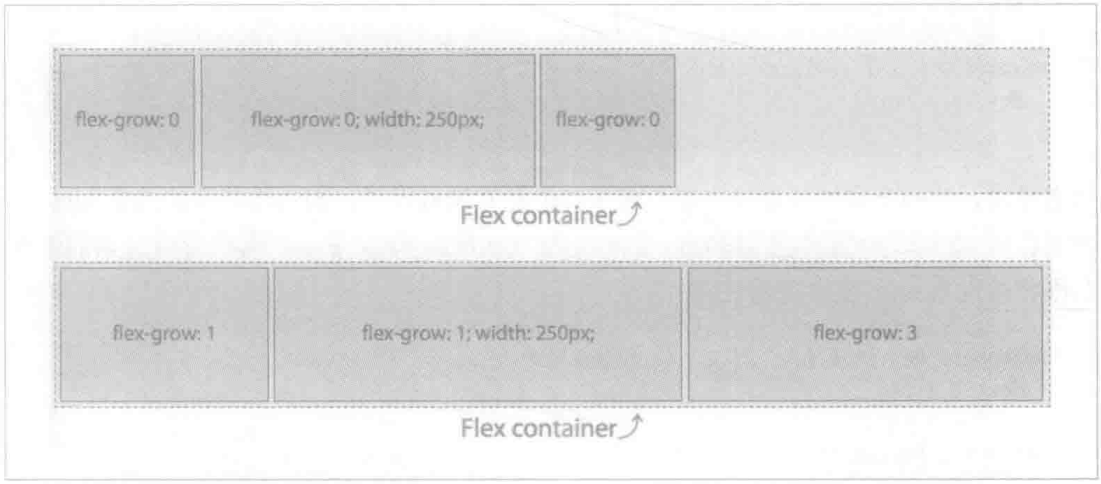


图12-43：宽度和增长因子设为不同的值 ▶

总结一下，弹性容器的可用空间，以及各弹性元素的增长因子和最终宽度如下：

可用空间：750px - (100px + 250px + 100px) = 300px

增长因子：1 + 1 + 3 = 5

每一份增长因子所占的宽度：300px ÷ 5 = 60px

弹性增长时，根据原宽度和增长因子计算得到的宽度分别为：

弹性元素 1 = 100px + (1 × 60px) = 160px

弹性元素 2 = 250px + (1 × 60px) = 310px

弹性元素 3 = 100px + (3 × 60px) = 280px

三者之和为 750 像素。

12.12.1 在 flex 属性中设定增长因子

flex 属性有三个值：增长因子，缩减因子和基准。第一个有效的正数设定的是增长因子（即 flex-grow 的值）。如果 flex 属性没有设定增长因子和缩减因子，增长因子默认为 1。然而，如果 flex 和 flex-grow 都没有声明，增长因子默认为 0。是的，你没看错。

再来看图 12-42 中的第二个例子，各弹性元素的增长因子为 0、0 和 1。因为我们只为 flex-grow 声明了值，所以弹性基准默认为 auto，就像下面这样声明一样：

```
#example2 flex-item {
  flex: 0 1 auto;
}
#example2 flex-item:last-child {
  flex: 1 1 auto;
}
```

这意味着，前两个弹性元素没有增长因子，但是有缩减因子，而弹性基准的值是 auto。如果在图 12-42 的例子中使用 flex，而非不建议使用的 flex-grow，每个弹性元素的弹性基准将被设为 0%，就像下面这样：

```
#example2 flex-item {
  flex: 0 1 0%;
}
#example2 flex-item:last-child {
  flex: 1 1 0%;
}
```

由于缩减因子默认为 1、基准默认为 0%，下述 CSS 与前一个代码片段的作用是一样的：

```
#example2 flex-item {
  flex: 0;
```

```
}
#example2 flex-item:last-child {
  flex: 1;
}
```

得到的结果如图 12-44 所示。请与图 12-42 比较一下，看看有什么变化（以及哪些没有变）。

在前两种情况中你可能注意到奇怪的地方了：弹性基准都是零，只有第二种情况中的最后一个弹性元素的弹性增长因子是正值。逻辑上，3 个弹性元素的宽度似乎应该分别是 0、0 和 750 像素。但是转念一想，既然弹性容器还有空间，即便弹性基准是 0，内容也不该从弹性元素中溢出，这不符合逻辑。

规范制定人员考虑到这种两难的境地了。如果 `flex` 属性显式把弹性基准声明为 0%，或者默认为 0%，而且弹性元素的增长因子是 0，那么尺寸不增加的弹性元素在主轴上的长度将缩小为内容允许的最小长度，或者比这小一些。在图 12-44 中，最小长度等于最宽字母序列的长度，即“flex:”（包括冒号）的长度。

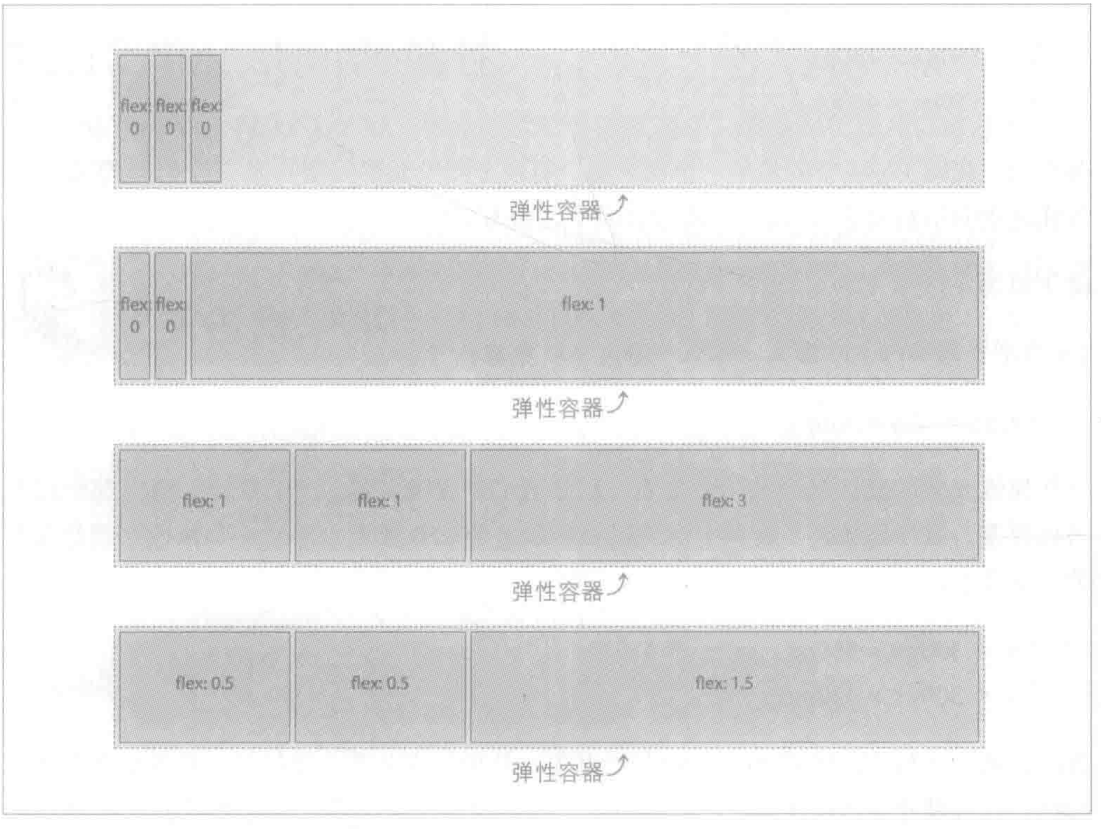


图 12-44：使用 flex 简写属性时的弹性尺寸 ▶

只要弹性元素明显有溢出，而且没有显式设定 min-width（纵向主轴为 min-height），那么弹性元素的最小宽度（或最小高度）为容下内容所需的最小宽度（或高度），或者是声明的 width（或 height），二者取较小的那个。

如果所有弹性元素都允许增大，而且弹性基准都是 0%，那么所有空间，而不只是多出的空间，将根据增长因子确定的比例分配。在图 12-44 中的第三个例子里，有两个弹性元素的增长因子是 1，另一个弹性元素的增长因子是 3。因此，增长因子之和为 5：

$$(2 \times 1) + (1 \times 3) = 5$$

5 份增长因子占 750 像素，那么一份占 150 像素：

$$750\text{px} \div 5 = 150\text{px}$$

虽然弹性元素的默认尺寸为 100 像素，但是我们把弹性基准设为 0% 了，所以那个尺寸将被覆盖。最终结果是，有两个弹性元素的宽度为 150 像素，最后一个弹性元素的宽度为 450 像素：

$$1 \times 150\text{px} = 150\text{px}$$
$$3 \times 150\text{px} = 450\text{px}$$

类似地，在图 12-44 中的最后一个例子里，有两个弹性元素的增长因子为 0.5，还有一个弹性元素的增长因子为 1.5，因此增长因子之和为 2.5：

$$(2 \times 0.5) + (1 \times 1.5) = 2.5$$

2.5 份增长因子占 750 像素，那么一份占 300 像素：

$$750\text{px} \div 2.5 = 300\text{px}$$

虽然弹性元素的默认尺寸为 100 像素，但是我们把弹性基准设为 0% 了，所以那个尺寸将被覆盖。最终结果是，有两个弹性元素的宽度为 150 像素，最后一个弹性元素的宽度为 450 像素：

$$0.5 \times 300\text{px} = 150\text{px}$$
$$1.5 \times 300\text{px} = 450\text{px}$$

再次注意，这与只声明 flex-grow 属性是不一样的，因为那么做的话，弹性基准默认为 auto，只有额外空间按比例分配，而不是所有空间都分配。然而，使用 flex 时，弹性基准为 0%，全部空间将按比例分配给弹性元素，而不只是余下的空间。两种情况之间的区别如图 12-45 所示。

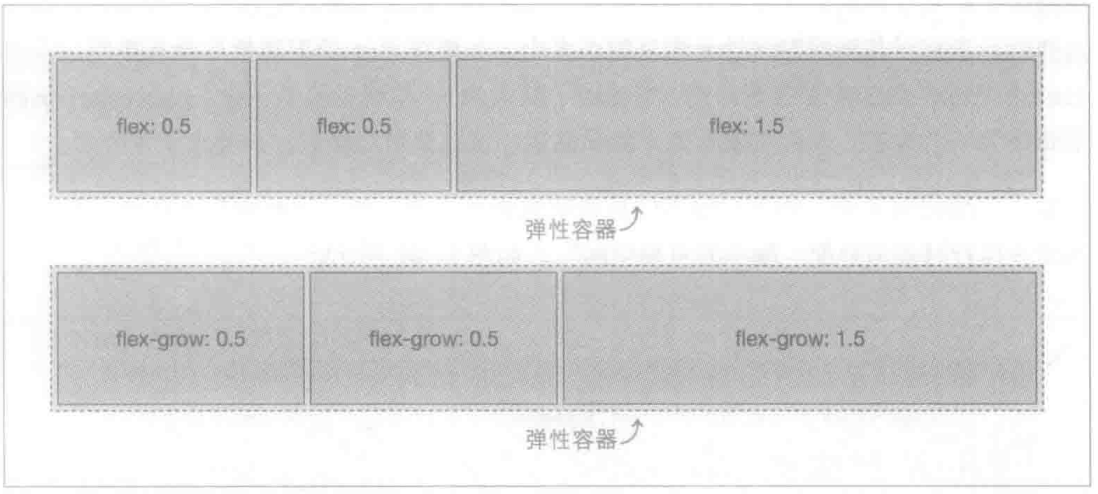


图 12-45：使用 flex 和 flex-grow 时弹性尺寸之间的差别 ▶

接下来讨论弹性缩减因子，在某些方面它与弹性增长因子是相反的，不过也不尽然。

12.13 flex-shrink 属性

`flex` 简写属性的 *<flex-shrink>* 部分指定弹性缩减因子。缩减因子也可以通过 `flex-shrink` 属性设定。



规范制定人员极度不鼓励通过 `flex-shrink` 属性定义缩减因子，而是建议在 `flex` 简写属性中指定缩减因子。我们单独讨论这个属性只是为了说明它的作用。

flex-shrink

取值	<number>
初始值	1
适用于	弹性元素（弹性容器的子元素）
计算值	指定的值
继承性	否
动画性	是

缩减因子定义空间不足以放下所有弹性元素时（尺寸由内容或其他 CSS 属性定义），当前弹性元素相对其他同辈弹性元素将缩小多少。如果在 flex 简写属性中没有声明，或者 flex 和 flex-shrink 都没有声明，缩减因子默认为 1。与增长因子一样，flex-shrink 的值始终为一个数字，而且负数无效。如果愿意，可以使用非整数，只要大于零即可。

基本上，缩减因子定义的是空间不够放置所有弹性元素，而且弹性容器不能增加尺寸或不允许换行时如何分配“缺少的可用空间”。如图 12-46 所示。



图 12-46：不同的弹性缩减情况 ▶

图 12-46 与图 12-42 类似，不过为弹性元素设置了 width: 300px，而不是 100 像素。弹性容器的宽度仍是 750 像素。3 个弹性元素的宽度之和为 900 像素，也就是内容比所在的弹性容器宽 150 像素。如果不允许弹性元素缩小或换行（见 12.2.3 节），弹性元素将超出尺寸固定的弹性容器。这种情况如图 12-46 中的第一个例子所示，各弹性元素的缩减因子是零，因此不会缩小，所以它们将从弹性容器中溢出。

在图 12-46 中的第二个例子里，只有最后一个弹性元素能缩小。因此，为了在弹性容器中放下全部弹性元素，最后一个弹性元素将尽量缩小。我们想把 900 像素宽的内容放在 750 像素宽的容器中，所以缺少的可用空间是 150 像素。没有设置缩减因子的两个弹性元素，其宽度保持 300 像素不变。而第三个弹性元素的缩减因子是个正数，为了让容器

能放下 3 个弹性元素，它将缩小 150 像素，变成 150 像素宽（这里我们为第三个弹性元素设置的缩减因子是 1，不过设为 0.001、100、314159.65，或者浏览器能理解的其他正数，得到的结果都是一样的）。

在第三个例子中，我们为三个弹性元素都设置了正的缩减因子：

```
#example3 flex-item {
  flex-shrink: 1;
}
#example3 flex-item:last-child {
  flex-shrink: 3;
}
```

因为这只是 flex 简写属性三部分中的一个，所以弹性元素的行为将像下面这样声明一样：

```
#example3 flex-item {
  flex: 0 1 auto; /* 增长因子默认为 0，基准默认为 auto */
}
#example3 flex-item:last-child {
  flex: 0 3 auto;
}
```

如果所有弹性元素都允许缩减（这里就是如此），缩减的量根据缩减因子按比例分配。这意味着，在同辈弹性元素中，缩减因子越大的弹性元素缩减的量越大。

在一个宽度为 750 像素的容器中，3 个弹性元素的宽度都是 300 像素，因此要从允许缩减的弹性元素（这里全部弹性元素都允许）上拿掉 150 像素“缺少的空间”。本例中，有两个弹性元素的缩减因子是 1，还有一个弹性元素的缩减因子是 3，因此共有 5 份缩减因子：

$$(2 \times 1) + (1 \times 3) = 5$$

而我们要从全部弹性元素上拿掉 150 像素，所以每份缩减因子占 30 像素：

$$150\text{px} \div 5 = 30\text{px}$$

各弹性元素的默认宽度是 300 像素，因此有两个弹性元素的宽度将变成 270 像素，最后一个弹性元素的宽度将变成 210 像素，缩小后的宽度之和为 750 像素：

$$300\text{px} - (1 \times 30\text{px}) = 270\text{px}$$
$$300\text{px} - (3 \times 30\text{px}) = 210\text{px}$$

下述 CSS 得到的结果是一样的，虽然缩减因子所用的数字不同，但比例是一样的，因此弹性元素的宽度是相同的：

```
flex-item {
  flex: 1 0.25 auto;
}
flex-item:last-child {
  flex: 1 0.75 auto;
}
```

注意，这些例子中的弹性元素将分别缩小为 210、210 和 270 像素宽，前提是各弹性元素中的内容（例如媒体对象或不换行的文本）不超过 210、210 和 270 像素宽。如果弹性元素中的内容不能换行，也不能在主轴方向上缩小，那么弹性元素无法进一步缩小。

假设第一个弹性元素中有一张 300 像素宽的图像，虽然其他弹性元素可以缩小，但这个弹性元素不会缩小，就像没有设置缩减因子一样。此时，第一个弹性元素的宽度为 300 像素，缺少的 150 像素空间将根据第二个和第三个弹性元素的缩减因子按比例分配。

这种情况下，不受阻碍的缩减因子有 4 份（一份来自第二个弹性元素，三份来自第三个弹性元素），缺少的空间为 150 像素，因此每份缩减因子占 37.5 像素。三个弹性元素的宽度将分别变成 300、262.5 和 187.5 像素，总和为 750 像素，如图 12-47 所示。

弹性元素 1 = 300px - (0 × 37.5px) = 300.0px
弹性元素 2 = 300px - (1 × 37.5px) = 262.5px
弹性元素 3 = 300px - (3 × 37.5px) = 187.5px

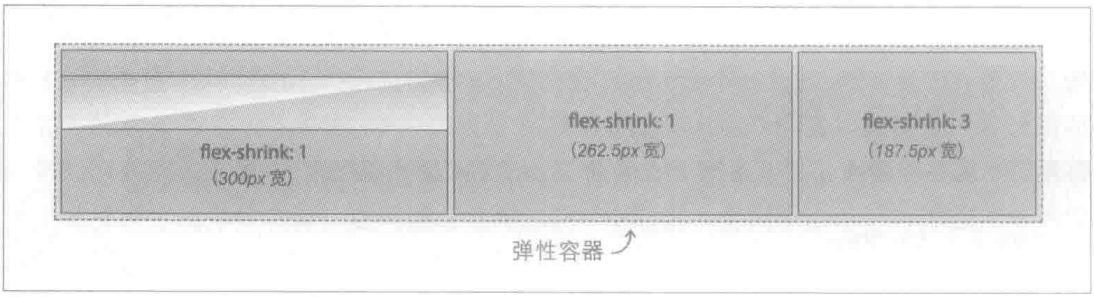


图 12-47：弹性元素受内容的限制而无法缩小

如果图像的宽度为 296 像素，第一个弹性元素将缩小 4 像素。余下的 146 像素缺少的空间交给 4 份缩减因子，因此一份占 36.5 像素。所以，弹性元素的宽度分别为 296、263.5 和 190.5 像素。

如果三个弹性元素中都有 300 像素或更宽的不换行文本或媒体，那么三个弹性元素都不会缩小，得到的结果与图 12-46 中的第一个例子类似。

12.13.1 根据宽度和缩减因子按比例缩小

上述示例比较简单，因为各弹性元素的宽度一开始是相同的。如果宽度不等呢？如果第一个和最后一个弹性元素的宽度为 250 像素，而中间那个弹性元素的宽度为 500 像素呢？如图 12-48 所示。

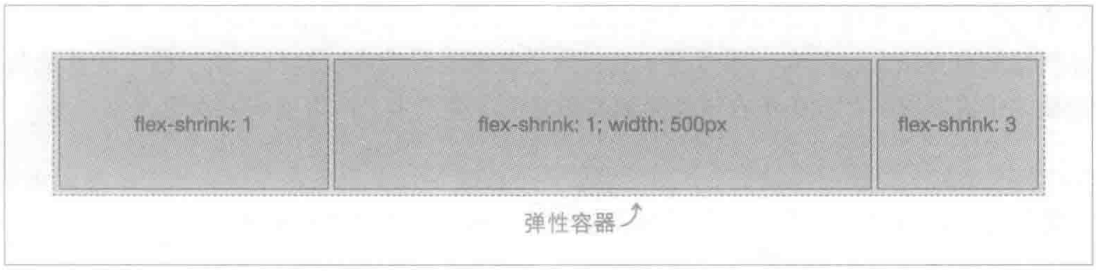


图 12-48：弹性元素根据缩减因子按比例缩小

弹性元素根据缩减因子和元素的宽度按比例缩小。这里所说的宽度通常指内容不换行时的宽度。在图 12-48 中，我们试图把 1000 像素宽的内容放在 750 像素宽的弹性容器里。因此，要从 5 份缩减因子上除去 250 像素。

对 flex-grow 来说，只需拿 250 像素除以 5，为每份增长因子分配 50 像素。如果以类似的方式缩小，各弹性元素的宽度将变成 200、550 和 100 像素。但实际上不是这样缩小的。

这里，有 250 像素缺少的空间要按比例分配。为了得到缩减因子比例，要拿缺少的空间除以各弹性元素的宽度（更准确地说是主轴上的长度）与其缩减因子乘积的总和：

$$\text{缩小比例} = \frac{\text{缺少的空间}}{(\text{宽度 1} \times \text{缩减因子 1}) + \ldots + (\text{宽度 N} \times \text{缩减因子 N})}$$

根据这个公式，算出缩减比例：

= 250px ÷ ((250px × 1) + (500px × 1) + (250px × 3))
= 250px ÷ 1500px
= 0.166666667 (16.67%)

每个弹性元素缩小的量等于 16.67% 乘以 flex-shrink 的值，得到的结果如下：

弹性元素 1 = 250px × (1 × 16.67%) = 41.67px
弹性元素 2 = 500px × (1 × 16.67%) = 83.33px
弹性元素 3 = 250px × (3 × 16.67%) = 125px

然后分别从弹性元素的初始尺寸 250、500 和 250 像素上减去缩小的量，得到 208.33、416.67 和 125 像素。

12.13.2 不同的基准

如果宽度和弹性基准的值都是 auto，缩减因子为零时，弹性元素中的内容不换行，这与你所想的有点出入。其实，缩减因子为任何正数都会导致内容换行。由于缩小的量是与缩减因子成比例的，如果所有弹性元素的缩减因子差不多，内容换行的次数也差不多。

图 12-49 中的三个例子都没有为弹性元素声明宽度。所以，宽度由内容决定，因为 width 的值是 auto。那么，弹性容器的宽度将是 520 像素，而不是 750 像素。

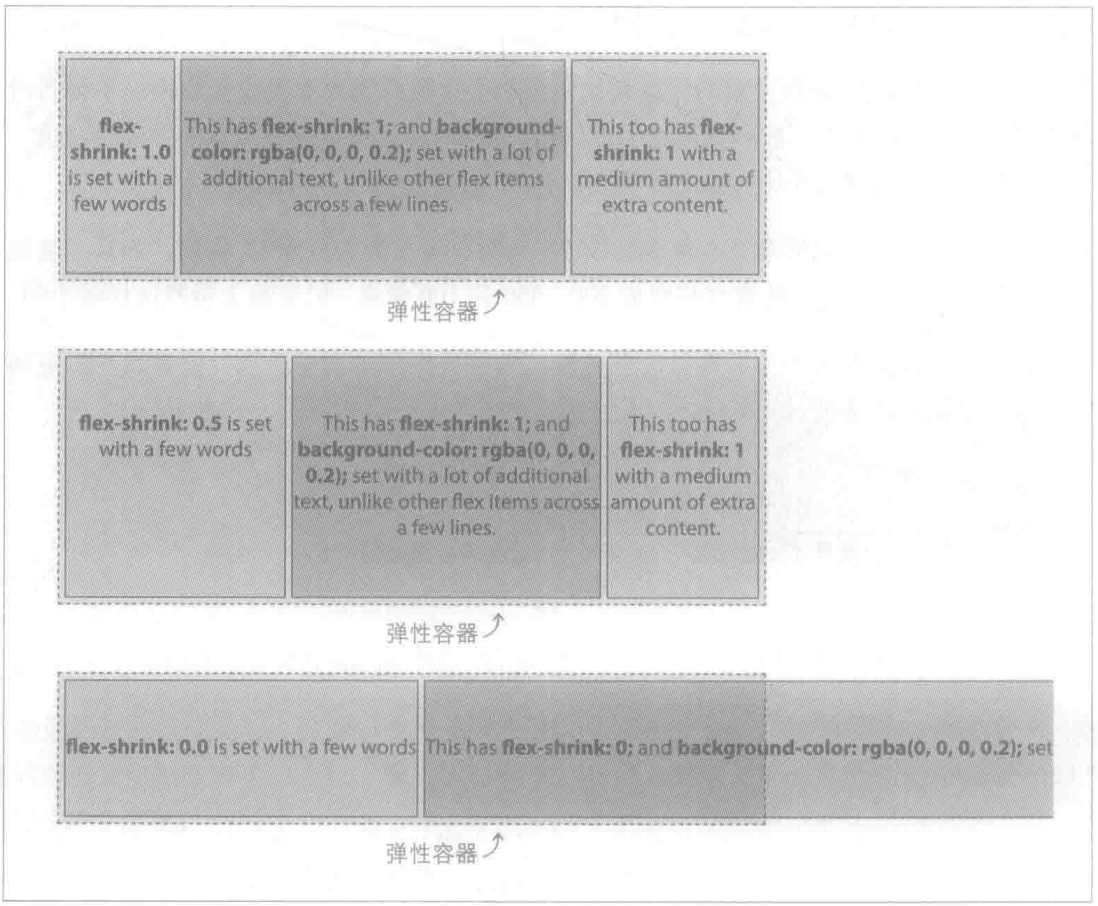


图 12-49：弹性元素相对各自的缩减因子和内容按比例缩小 ▶

注意，在第一个例子中，所有弹性元素的 flex-shrink 值都一样，因此每个元素中的内容都断为四行。在第二个例子中，第一个弹性元素的缩减因子是另外两个弹性元素的一半，因此它换行的次数（基本上）也是其他两个元素的一半。这就是缩减因子在起作用。

第三个例子没有设定缩减因子，因此文本根本不换行，导致弹性元素从容器中溢出很多。



截至 2017 年年末，这种“均衡换行”和拒绝换行的行为在不同的浏览器中表现得不一致。如果你自己试验时得到了不同的结果，原因可能就是这个。

由于 flex 属性设定的缩减因子将按比例减小弹性元素的宽度，而在增减宽度的过程中，弹性元素中文本的行数将随之增减，因此当同辈弹性元素的缩减因子差不多时，元素中内容的高度也相差无几。

在前面的示例中，假设第三个例子中内容不换行时弹性元素的宽度分别为 280、995 和 480 像素（通过开发者工具测量出的结果，简单起见，又取整了）。这意味着，要根据缩减因子按比例缩小各弹性元素，把 1755 像素的内容放在 520 像素宽的弹性容器中。因此，要按比例分配的缺少的可用空间为 1235 像素。



记住，不要依赖 Web 审查工具确定要使用的缩减因子。这里这么做是为了说明缩减因子的运作原理。如果你不想知道细枝末节，请翻到 12.14 节。

在第一个例子中，各弹性元素中的文本行数（几乎）是一样的。这是因为弹性元素是相对内容的宽度按比例缩小的。

这里没有声明宽度，因此不能像之前的例子那样以一个确定的宽度为基准做计算。现在，我们要根据内容的宽度，即 280、995 和 480 像素，按比例分配 1235 像素缺少的空间。经计算，520 是 1755 的 29.63%，那么对缩减因子都为 1 的弹性元素来说，其宽度为内容宽度乘以 29.63%：

弹性元素 1 = 280px × 29.63% = 83px
弹性元素 2 = 995px × 29.63% = 295px
弹性元素 3 = 480px × 29.63% = 142px

在默认的 `align-items: stretch` 下（见 12.6 节），得到的是等高的三列布局。为所有弹性元素设置一样的缩减因子，三个弹性元素中内容的高度几乎是一样的，不过各列的宽度可能就不一致了。

在图 12-49 中的第二个例子里，不是所有弹性元素的缩减因子都一样。第一个弹性元素按比例缩小的量为另外两个弹性元素的一半。仍以前面的宽度为例，即 280、995 和 480 像素。然而，现在各弹性元素的缩减因子分别为 0.5、1.0 和 1.0。既然知道内容的宽度，那么缩减比例（X）经过计算可以得出：

280px + 995px + 480px = 1615px
(0.5 × 280px) + (1 × 995px) + (1 × 480px) = 1235px
X = 1235px ÷ 1615px = 0.7647

知道缩减比例后，最终宽度也就能计算出来了。第二个和第三个弹性元素缩小的比例就是 76.47%，而第一个弹性元素将缩小 38.23%（因为它的 `flex-shrink` 值是另外两个弹性元素的一半）。化为最近的整数后，各弹性元素的缩小量如下：

弹性元素 1 = 280px × 0.3823 = 107px
弹性元素 2 = 995px × 0.7647 = 761px
弹性元素 3 = 480px × 0.7647 = 367px

因此，各弹性元素的最终宽度为：

弹性元素 1 = 280px - 107px = 173px
弹性元素 2 = 995px - 761px = 234px
弹性元素 3 = 480px - 367px = 113px

三个弹性元素的宽度之和为 520 像素。

缩减因子和增长因子各不相同时，结果不那么容易理解。这就是为什么建议始终使用 `flex` 简写属性，而且最好为各弹性元素声明宽度或基准。如果不理解以上内容也别担心，后文讨论 `flex-basis` 属性时将再举一些缩小的例子。

12.13.3 响应式弹性布局

利用这种按比例缩小弹性元素尺寸的行为可以实现响应式布局。

例如，无需使用媒体查询就可以创建能自动增加和减小尺寸的三栏布局，在宽屏中的效果如图 12-50 所示，在窄屏中的效果如图 12-51 所示。

```
nav {
  flex: 0 1 200px;
  min-width: 150px;
}
article {
  flex: 1 2 600px;
}
aside {
  flex: 0 1 200px;
  min-width: 150px;
}
```

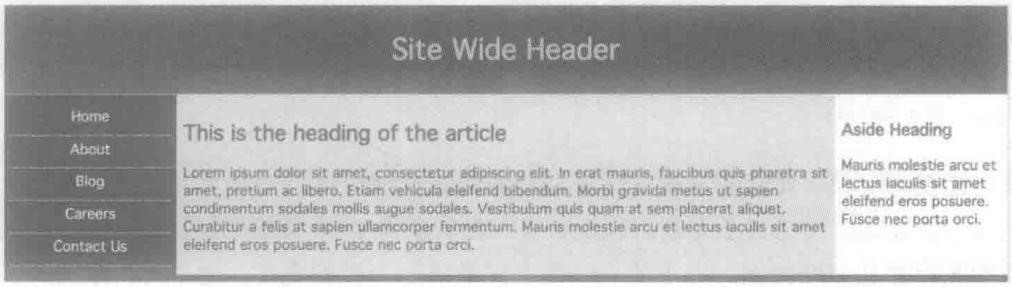


图 12-50：宽屏中的弹性盒布局

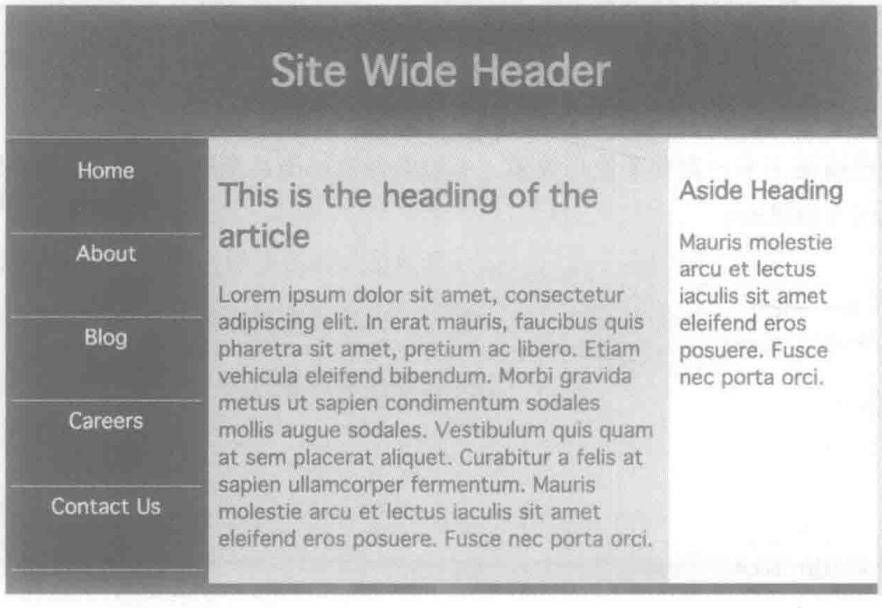


图 12-51：窄屏中的弹性盒布局 ▶

在这个示例中，如果视区的宽度超过 1000 像素，只有中间一栏的宽度会增加，因为只有那一栏有值为正数的增长因子。我们还指明，宽度小于 1000 像素时，每一栏的宽度都减小。

下面一点一点分析。nav 和 aside 元素的 CSS 如下：

```
flex: 0 1 200px;
min-width: 150px;
```

这意味着，二者不会在基准的基础上增加宽度，但是将缩小相同的量。因此，默认情况下，nav 和 aside 元素的宽度等于弹性基准。如果需要缩小，它们最多缩小到 150px，然后就不再缩小了。然而，如果 nav 和 aside 中有宽度超过 150 像素的元素，不管是图像还是一串文本，到达那部分内容的宽度时就停止缩小。假设 aside 元素中有一个 180 像素宽的图像，那一栏缩小到 180 像素后便不再缩小。而 nav 元素将继续缩小到 150 像素。

main 元素的样式如下：

```
flex: 1 2 600px;
```

因此，需要时 main 元素的宽度可以增加。因为这是唯一可以增加尺寸的弹性元素，所以增加的宽度都分配给这个元素。这意味着，在宽度为 1300 像素的浏览器窗口中，两边两栏的宽度都是 200 像素，余下的 900 像素就是中间一栏的宽度。缩小时，中间一栏缩小的量为另两列的两倍。因此，如果浏览器窗口的宽度为 900 像素，两边两栏的宽度都是 175 像素，中间一栏的宽度则是 550 像素。

如果窗口的宽度减小到 800 像素，两边两栏的宽度达到 min-width 设定的 150px。从这时开始，空间都从中间一栏减去。

注意，这些情况下不一定非要使用像素。不同的弹性基准甚至还可以使用不同的单位。前例可以改写成这样：

```
nav {
  flex: 0 1 20ch;
  min-width: 15vw;
}
article {
  flex: 1 2 45ch;
}
aside {
  flex: 0 1 20ch;
  min-width: 10ch;
}
```

我们不再说明具体的计算过程，不过要指出的是，这里使用基于字符宽度的弹性基准能增强可读性，两个宽度下限则使用基于字符宽度和视区宽度的单位。



弹性盒可用于创建如前所示的一维页面布局，这里在一行中只有三栏。如果想实现更复杂的布局，或者需要更强大的选项，请使用栅格布局（见第 13 章）。

12.14 flex-basis 属性

我们知道，弹性元素的尺寸受内容及盒模型属性的影响，而且可以通过 flex 属性的三个要素重置。flex 属性中的 *<flex-basis>* 要素定义弹性元素的初始或默认尺寸，即根据增长因子和缩减因子分配多余或缺少的空间之前，弹性元素的大小。这个要素也可以使用 flex-basis 属性设定。



规范制定人员极度不鼓励通过 flex-basis 属性定义弹性基准，而是建议在 flex 简写属性中指定弹性基准。我们单独讨论这个属性只是为了说明它的作用。

flex-basis

取值	content | [*<length>* | *<percentage>*]
初始值	auto
适用于	弹性元素（弹性容器的子元素）
百分数	相对弹性容器内的主轴尺寸计算
计算值	指定的值，长度值计算为绝对长度
继承性	否
动画性	*<width>*

弹性基准设定弹性元素的元素框（由 box-sizing 设置）的尺寸。默认情况下，对不是弹性元素的普通块级元素来说，其尺寸由父元素和内容的尺寸，以及盒模型属性决定。如果没有显式声明或继承尺寸相关的属性，元素的尺寸默认包含内容、边框和内边距，即父元素宽度的 100%。

定义弹性基准的长度值类型与 width 和 height 属性支持的一样，例如 5vw、12% 和 300px。

全局关键字 initial 把弹性基准重置为初始值 auto，因此也可以声明 auto。auto 的计算值等于 width（或 height），前提是声明了这两个属性。如果 width（或 height）的值为 auto，那么 flex-basis 的值计算为 content。

12.14.1 content 关键字

写作本书时（2017 年年末），多数浏览器都不支持 content 关键字（*http://code.google.com/p/chromium/issues/detail?id=470421*），不过微软的 Edge 12+ 除外，但是却等于内容的宽度或高度。使用 content 时，在支持的环境中，弹性基准等于弹性元素中内容的尺寸，即最长一行内容或最宽（或最高）那个媒体对象在主轴上的长度。

在得到全面支持以前，可以轻易使用腻子脚本（polyfill）让浏览器支持 flex-basis: content;，因为它相当于在弹性元素上声明 flex-basis: auto; width: auto; 或 flex-basis: auto; height: auto;（主轴为纵向时）。然而，在 flex 简写声明中使用 content 时，不支持这个关键字的浏览器将忽略整个声明（见本章前面“深入理解各种轴”一节）。

content 的作用基本上与图 12-49 中的第三个例子一样，如图 12-52 所示。

在图 12-52 中的第一个和第三个例子里，弹性元素的宽度等于内容的尺寸，弹性基准也是。在第一个例子中，弹性元素的宽度和基准大约为 132 像素。三个并排放置的弹性元素，总宽度为 396 像素，放在弹性容器中绰绰有余。

第三个例子设置了空的缩减因子（0），弹性元素无法缩小，因此不会为了能在宽度固定的弹性容器中放得下而缩小或换行。这些弹性元素的宽度等于文本未换行时的宽度。这个宽度也是弹性基准的值。三个弹性元素的宽度和基准大约分别为 309、1037 和 523 像素。图中的第二个弹性元素不完整，而且根本看不到第三个弹性元素，但是在线示例中能看全（*https://meyerweb.github.io/csstdg4figs/12-flexbox/*）。

在第二个例子中，弹性元素中的内容与第三个例子一样，但是弹性元素的缩减因子默认为 1，也就是说弹性元素可以缩小，因此这里的文本将换行。这种情况下，虽然弹性元素的宽度不等于内容的宽度，但是弹性基准（按比例缩小时的基准）等于内容的宽度。

12.14.2 自动确定弹性基准

设为 auto 时，不管是显式声明的还是取默认值，flex-basis 等于元素在主轴方向上的尺

寸，就像没把元素变成弹性元素一样。如果 width 或 height 的值是长度，弹性基准就等于那个长度；而如果 width 或 height 也是 auto，那么弹性基准回落为 content。

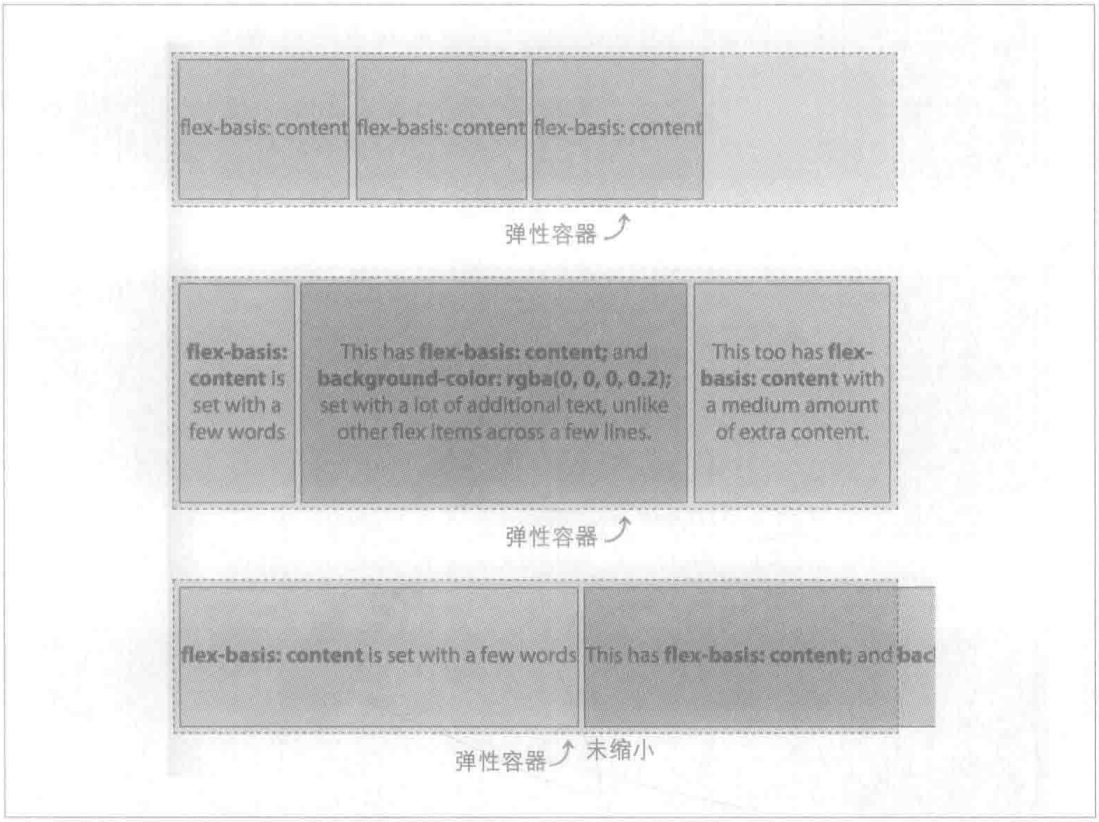


图 12-52：以 content 为基准确定弹性元素的尺寸

如果弹性基准为 auto，而且弹性容器中能放下全部全部弹性元素，弹性元素的尺寸与未弹性变形之前一样。如果弹性容器中放不下弹性元素，弹性元素将根据未弹性变形之前的主轴尺寸按比例缩小（除非缩减因子为零）。

如果没有其他属性设定弹性元素在主轴上的尺寸（即没有为弹性元素设定 width 抑或 min-width），而且声明了 flex-basis: auto 或 flex: 0 1 auto，那么弹性元素的宽度为恰好能放得下内容时的长度，如图 12-53 中的第一个例子所示。此时，弹性元素的宽度等于“flex-basis: auto;”文本的宽度，大约为 110 像素。弹性元素的尺寸与未弹性变形之前一样，就像声明 display: inline-block 似的。在这个例子中，弹性元素统一向主轴起边靠拢，因为弹性容器的 justify-content 属性默认为 flex-start（见 12.5 节）。

在图 12-53 中的第二个例子里，每个弹性元素的弹性基准都是 auto，而且显式声明了宽度。各弹性元素的主轴尺寸，即未变成弹性元素之前的尺寸分别为 100、150 和 200 像素。得到图中所示的结果是由于全部弹性元素能在弹性容器中放下，主轴方向上没有任何溢出。

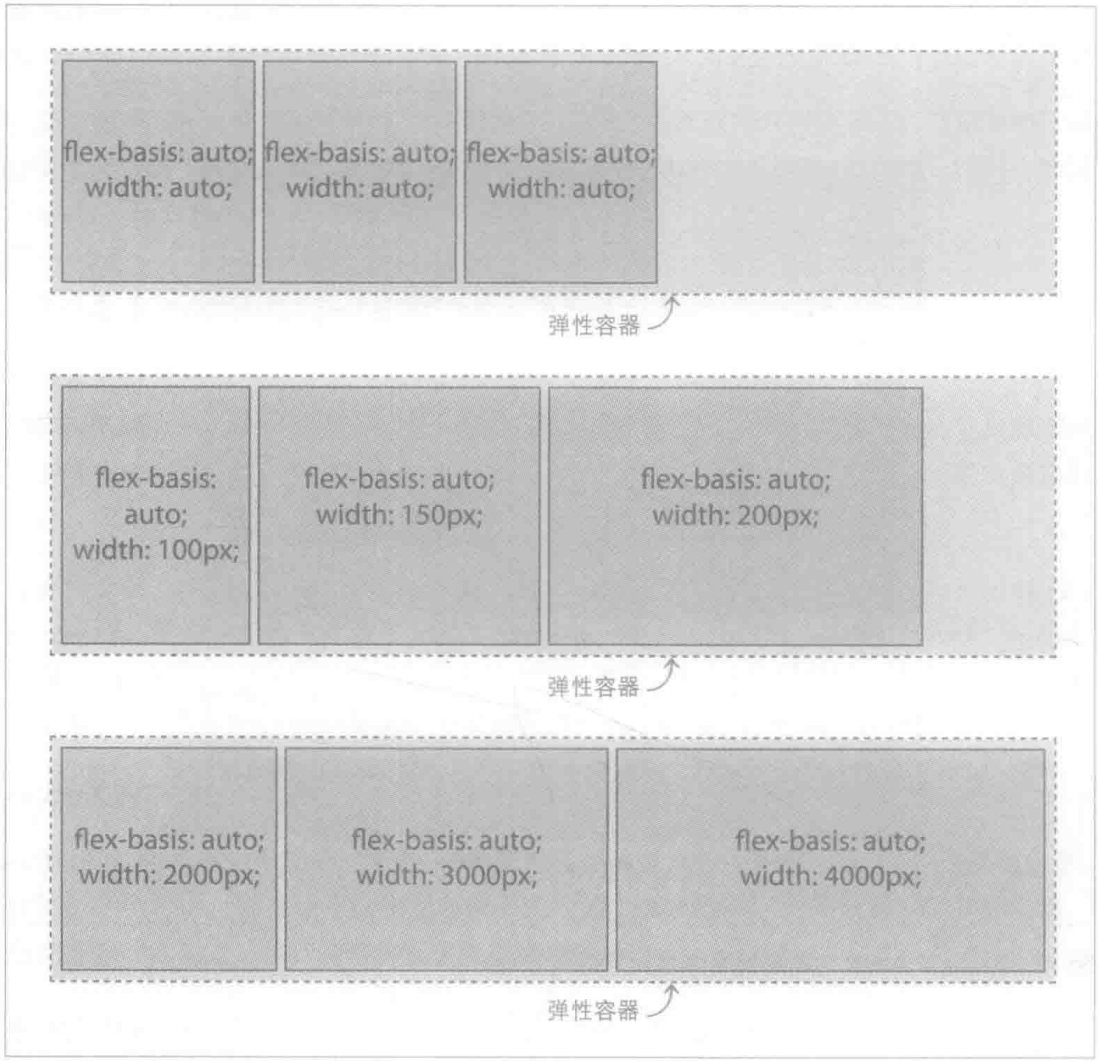


图 12-53：自动确定弹性基准和弹性元素的宽度 ▶

在图 12-53 中的第三个例子里，每个弹性元素的弹性基准都是 `auto`，而且显式声明了特别大的宽度。各弹性元素的主轴尺寸，即未变成弹性元素之前的尺寸分别为 2000、3000 和 4000 像素。因此，在不弹性变形的情况下，这三个弹性元素在主轴方向上将从弹性容器中溢出。由于三个弹性元素的弹性缩减因子都默认为 `1`，所以它们将缩小，直到能在弹性容器中放下为止。你可以根据前一节给出的过程计算弹性元素缩小后的尺寸。提示：第三个弹性元素的宽度由 4000 像素缩小为 240 像素。

12.14.3 默认值

如果既没有声明 flex-basis，也没有声明 flex，那么弹性元素的主轴尺寸是未弹性变形时元素的尺寸，因为默认值是 auto。

在图 12-54 中，各弹性元素的弹性基准默认为 auto，增长因子默认为 0，缩减因子默认为 1。那么，各弹性元素的弹性基准为各自的 width 值。因此，在第一个例子中，各弹性元素的弹性基准为 100、200 和 300 像素；在第二个例子中，各弹性元素的弹性基准为 200、400 和 200 像素。两个例子中弹性元素的宽度之和分别为 600 像素和 800 像素，都比主轴尺寸为 540 像素的容器大，因此两种情况下的每个弹性元素都将按比例缩小。

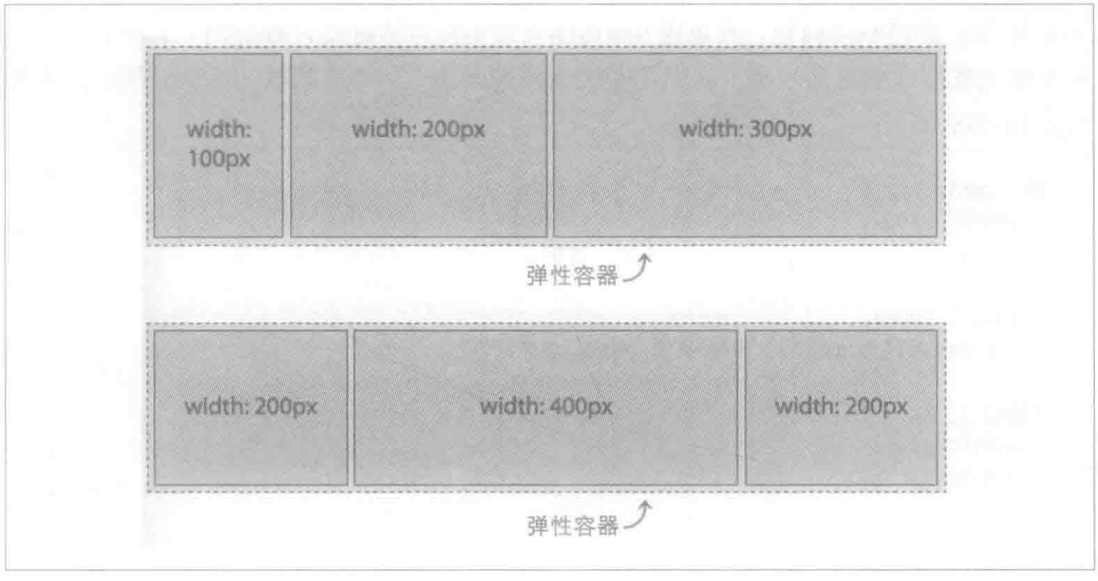


图 12-54：弹性元素的默认尺寸 ▶

在第一个例子中，我们想把 600 像素的元素放在 540 像素的容器中，因此各弹性元素要缩小 10%，分别变成 90、180 和 270 像素宽。在第二个例子中，我们想把 800 像素的元素放在 540 像素的容器中，因此各弹性元素要缩小 32.5%，分别变成 135、270 和 135 像素宽。

12.14.4 长度单位

在前面的示例中，弹性基准为 auto 时，默认等于为各弹性元素声明的宽度。除此之外还有其他选择，例如弹性基准的值可以与 width 和 height 使用相同的长度单位。

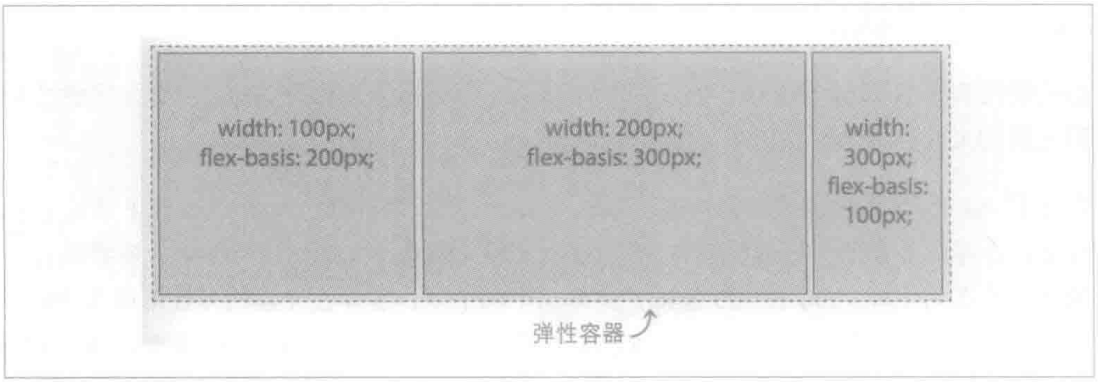


图 12-55：使用带长度单位的弹性基准确定弹性元素的尺寸 ▶

如果既设定了 flex-basis，又设定了 width（或主轴为纵向时的 height），弹性基准的优先级比宽度（或高度）高。我们把基准值添加到图 12-54 中的第一个例子里，各弹性元素的 CSS 如下：

```
flex-container {
  width: 540px;
}
item1 {
  width: 100px;
  flex-basis: 300px;  /* flex: 0 1 300px; */
}
item2 {
  width: 200px;
  flex-basis: 200px;  /* flex: 0 1 200px; */
}
item3 {
  width: 300px;
  flex-basis: 100px;  /* flex: 0 1 100px; */
}
```

宽度被基准覆盖。各弹性元素将缩小为 270 像素、180 像素和 90 像素。如果弹性容器的宽度没有限制，各弹性元素的宽度将是 300 像素、200 像素和 100 像素。

虽然声明的弹性基准可以覆盖弹性元素的主轴尺寸，但是主轴尺寸还受其他属性的影响，例如 min-width、min-height、max-width 和 max-height。这些属性不会被忽略。因此，如果在一个元素上声明了 flex-basis: 100px 和 min-width: 500px，尽管弹性基准比最小宽度小，但还是以最小宽度为准。

百分数

flex-basis 的百分数值相对弹性容器的主轴尺寸计算。

图 12-56 中的第一个例子前面已经见过，这里再次给出是为了提醒你，“flex-basis: auto”文本的宽度大约为 110 像素。就此而言，声明 flex-basis: auto 的效果与 flex-basis: 110px 一样：

```
flex-container {
  width: 540px;
}
flex-item {
  flex: 0 1 100%;
}
```

在图 12-56 中的第二个例子里，前两个弹性元素的弹性基准是 auto，width 归为默认值 auto，这与把弹性基准设为 content 一样。根据前面的讨论，前两个弹性元素的 flex-basis 相当于 110 像素，因为这里的内容恰好是 110 像素宽。最后一个弹性元素的 flex-basis 被设为 100%。

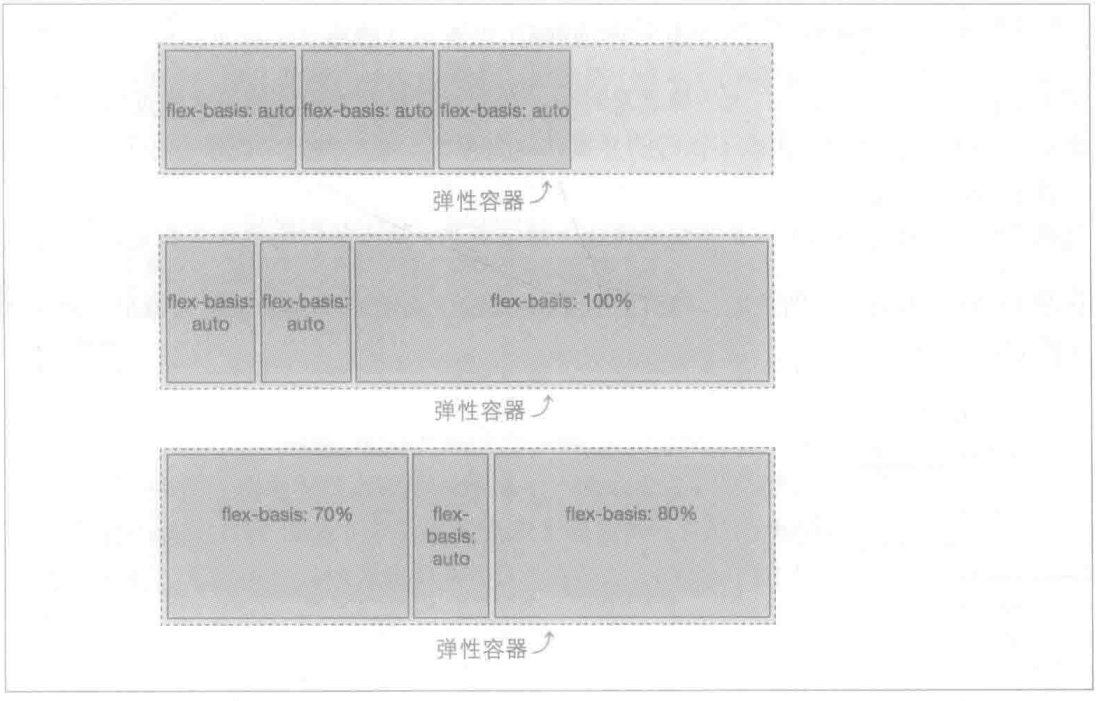


图 12-56：使用百分数弹性基准确定弹性元素的尺寸 ▶

百分数值相对容器计算，这里容器的宽度是 540 像素。第三个弹性元素的弹性基准为 100%，但它不是禁止换行的弹性容器中的唯一弹性元素。因此，这个弹性元素的尺寸不会增大为弹性容器宽度的 100%，除非它的缩减因子是空值（即禁止缩小），或者包含不换行的内容，而且宽度等于或大于容器的宽度。



记住：弹性基准为百分数值时，主轴尺寸相对弹性容器计算。

对这样三个弹性基准来说，如果内容的实际宽度为 110 像素，容器的宽度为 540 像素（简单起见，忽略其他盒模型属性），那么就要把 760 像素的内容放在 540 像素的空间里。因此，缺少 220 像素的空间要按比例分配。缩减比例为：

缩减比例 = 220px ÷ 760px = 28.95%

每个弹性元素将缩小 28.95%，变成未缩小时宽度的 71.05%。据此计算最终宽度：

弹性元素 1 = 110px × 71.05% = 78.16px
弹性元素 2 = 110px × 71.05% = 78.16px
弹性元素 3 = 540px × 71.05% = 383.68px

只要弹性元素能变这么小，这些值就是对的。意即，弹性元素中不能包含宽度大于 78.16 像素或 383.68 像素的媒体或不换行的文本。只要弹性元素中的内容能换行，而且与上述计算出的值相等或更小，这些就是弹性元素的最大宽度。我们说“最大宽度”是因为，如果另两个弹性元素中有一个不能变得比上述值更小，缺少的空间将从其他元素上剔除。

在图 12-56 中的第三个例子里，声明了 `flex-basis: auto` 的弹性元素断为三行。这个例子的 CSS 等效于：

```
flex-container {
  width: 540px;
}
item1 {
  flex: 0 1 70%;
}
item2 {
  flex: 0 1 auto;
}
item3 {
  flex: 0 1 80%;
}
```

三个弹性元素的 `flex-basis` 分别声明为 `70%`、`auto` 和 `80%`。记住，`auto` 等于未换行时内容的宽度，这里大约为 110 像素，而弹性容器的宽度为 540 像素，因此各弹性元素的弹性基准为：

弹性元素 1 = 70% × 540px = 378px
弹性元素 2 = widthOfText(“flex-basis: auto”) = 110px
弹性元素 3 = 80% × 540px = 432px

三个弹性元素的基准相加等于 920 像素，而这要放在 540 像素宽的弹性容器中。因此，缺少 380 像素空间，这部分空间要按比例从三个弹性元素上删除。为了算出比例，要拿弹性容器的宽度除以弹性元素未缩小时的宽度之和：

宽度缩小的比例 = 540px ÷ 920px = 0.587

由于缩减因子都是一样的，因此计算过程十分简单。每个弹性元素宽度的 58.7% 为：

弹性元素 1 = 378px × 58.7% = 221.8px
弹性元素 2 = 110px × 58.7% = 64.6px
弹性元素 3 = 432px × 58.7% = 253.6px

如果弹性容器不是这个宽度呢？假如是 1000 像素呢？此时，各弹性元素的基准分别为 700 像素（70% × 1000 像素）、110 像素和 800 像素（80% × 1000 像素），总和为 1610 像素。

宽度缩小的比例 = 1000px ÷ 1610px = 0.6211

弹性元素 1 = 700px × 62.11% = 434.8px
弹性元素 2 = 110px × 62.11% = 68.3px
弹性元素 3 = 800px × 62.11% = 496.9px

70% 和 80% 的基准相加始终大于 100%，因此不管弹性容器有多宽，三个弹性元素都会缩小。

如果第一个弹性元素由于什么原因无法缩小，比如说内容不能缩小，或者在 CSS 中的什么位置把 flex-shrink 设为 0 了，那么它的宽度为弹性容器的 70%，在这个例子中是 378 像素。那么，其他两个弹性元素必须按比例缩小，占满余下的 30%（或 162 像素）。在这个例子中，缩小后的宽度分别为 378 像素、32.875 像素和 129.125 像素。因为“basis:”文本比这宽（假设为 42 像素），所以最终得到的宽度为 378 像素、42 像素和 120 像素，如图 12-57 所示。



图 12-57：虽然弹性基准的百分数值相对弹性容器的宽度计算，但是主轴尺寸受同辈元素的影响 ▶

你在自己的设备上测试时得到的结果可能稍微不同，因为“flex-basis: auto”文本的宽度可能不一样，这取决于渲染文本用的是什么字体（笔者使用的字体是 Myriad Pro，回落为 Helvetica 和其他通用无衬线字体）。

12.14.5 零基准

如果样式中根本没有 flex-basis 和 flex 属性，弹性基准默认为 auto。如果声明了 flex 属性，但是没有设定弹性基准要素，那么弹性基准默认为 0。从表面上看，你可能觉得 auto 和 0 类似，但是 0 值可不像你想的那么简单。

对 flex-basis: auto 来说，基准为弹性元素的内容在主轴方向上的尺寸。如果每个弹性元素的基准都是 0，那么“可用”空间将是整个弹性容器的主轴尺寸。两种情况下，“可用”空间都根据各弹性元素的增长因子按比例分配。

基准为 0 时，弹性容器的尺寸根据增长因子按比例分给各个弹性元素。此时，由 height、width 或 content 定义的主轴方向上的原尺寸不在考虑范围内，然而 min-width、max-width、min-height 和 max-height 对弹性尺寸有一定的影响。

如图 12-58 所示，基准为 auto 时，只有多出的空间按比例分配给允许增大的弹性元素。还是假设“flex: X X auto”文本的宽度为 110 像素，那么在第一个例子中，多出的 210 像素要分为 6 份，每份 35 像素，因此弹性元素的宽度分别为 180、145 和 215 像素。

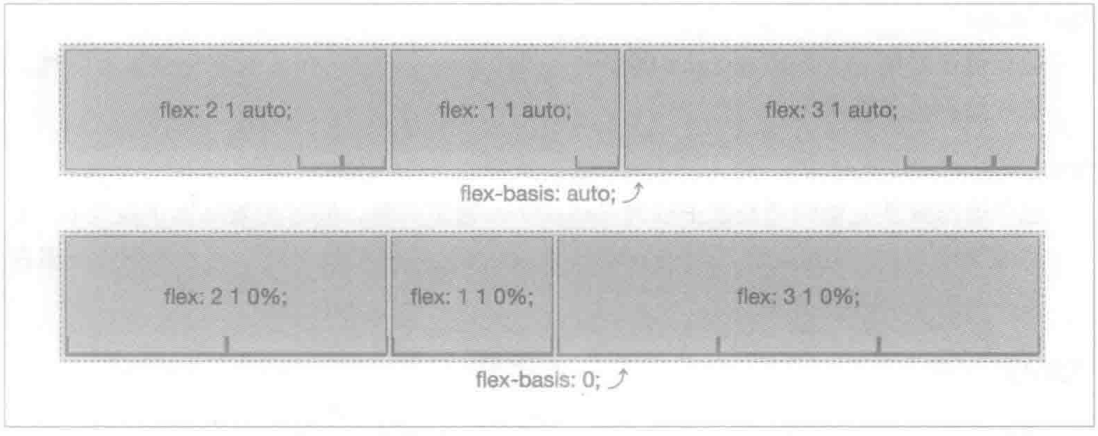


图 12-58：基准为 auto 和零时弹性增大的情况

在第二个例子中，基准为 0，全部 540 像素的空间都用于分配。540 像素的空间分为 6 份，每份 90 像素。因此，弹性元素的宽度分别为 180、90 和 270 像素。中间一个弹性元素的宽度为 90 像素，而其中的内容比 110 像素窄，所以这个弹性元素中的内容不换行。

12.15 flex 简写属性

至此，我们对 flex 简写属性的各要素做了全面的了解。记住，始终使用 flex 简写属性。这个属性接受常规的全局属性值，包括 initial、auto 和 none；只要是整数，通常为 1，就表示弹性元素可以增大。下面简要说明一下。

常见的弹性值

常见的弹性值有四个，涵盖了最常需要的效果：

flex: initial
: 这个值根据 width 或 height 属性（由主轴方向决定）确定弹性元素的尺寸，而且允许缩小。

flex: auto
: 这个值也根据 width 或 height 属性确定弹性元素的尺寸，但是元素是完全弹性的，既可以缩小也可以增大。

flex: none

这个值还是据 width 或 height 属性确定弹性元素的尺寸，但是元素完全没有弹性，不能缩小也不能增大。

flex: *<number>*

这个值把弹性元素的增长因子设为 *<number>* 指定的数，同时把缩减因子设为 0，把基准也设为 0。这意味着，width 或 height 属性的值相当于最小尺寸，弹性元素在有多余的空间时将增大。

下面逐一讨论。

initial 值

initial 是 CSS 中的全局关键字，任何属性都能使用这个关键字表示属性的初始值，即规范定义的默认值。因此，下面两个声明是等效的：

```
flex: initial;
flex: 0 1 auto;
```

flex: initial 声明把增长因子设为 0，缩减因子设为 1，弹性基准设为 auto。在图 12-59 中可以看到弹性基准为 auto 的效果。在前两个例子中，各弹性元素的基准是 content，各弹性元素的宽度为内容中一行字母的宽度。在后两个例子中，各弹性元素的基准为 50 像素，因为每个弹性元素都声明了 width: 50px。flex: initial 声明把 flex-basis 的值设为 auto，前面说过，如果声明了 width（或 height），就等于 width（或 height）的值，否则相当于 content。

在图 12-59 中的第一个和第三个例子里，可以看到，如果弹性容器太小，不足以放下默认主轴尺寸下的全部弹性元素，弹性元素将缩小，以便能在弹性容器中放下。在这两个例子中，各弹性元素的弹性基准之和比弹性容器的主轴尺寸大。在第一个例子中，各弹性元素的宽度不一，这由各元素的宽度和缩小量决定。虽然每个弹性元素都根据缩减因子按比例缩小，但是不会比其中最宽的内容窄。在第三个例子中，各弹性元素的弹性基准都是 50 像素（因为声明了 width），所以缩小的量是相同的。

默认情况下，弹性元素统一向主轴起边靠拢，因为 justify-content 属性的默认值是 flex-start。只有一行弹性元素在主轴方向上的长度之和比弹性容器的主轴尺寸小，而且所有弹性元素都不能增大时，才能看到这样的效果。

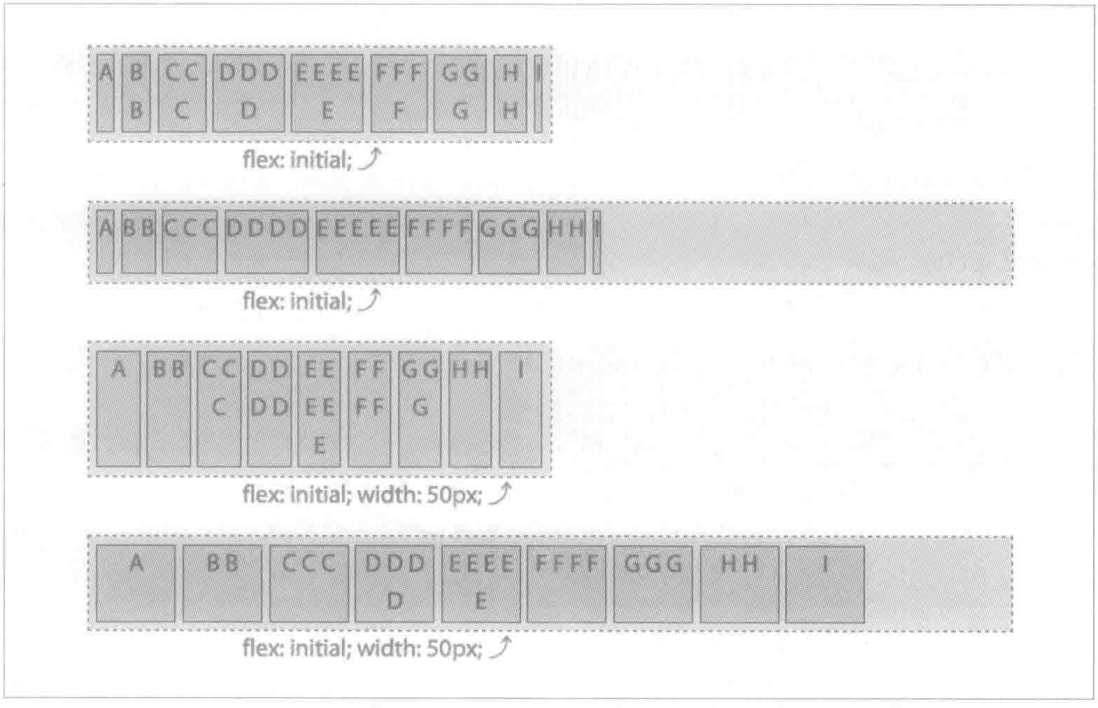


图 12-59：声明 flex: initial 时，弹性元素能缩小，但是不会增大 ▶

auto 值

flex: auto 的效果与 flex: initial 类似，不过弹性元素的弹性变形是双向的：如果容器的空间放不下全部弹性元素，元素将缩小；如果有额外的空间，弹性元素将增大。主轴方向上的任何多余空间都将分给弹性元素。下面两个声明是等效的：

```
flex: auto;
flex: 1 1 auto;
```

使用 auto 值时的不同情况如图 12-60 所示。

图 12-60 中的第一个和第三个例子与图 12-59 中的一样，因为缩减因子和基准没变。然而，第二个和第四个例子变了。这是因为，声明 flex: auto 时，增长因子为 1，因此弹性元素将增大，占满全部可用的额外空间。

使用 none 禁止弹性变形

声明了 flex: none 的弹性元素不具有弹性。下面两个 CSS 声明是等效的：

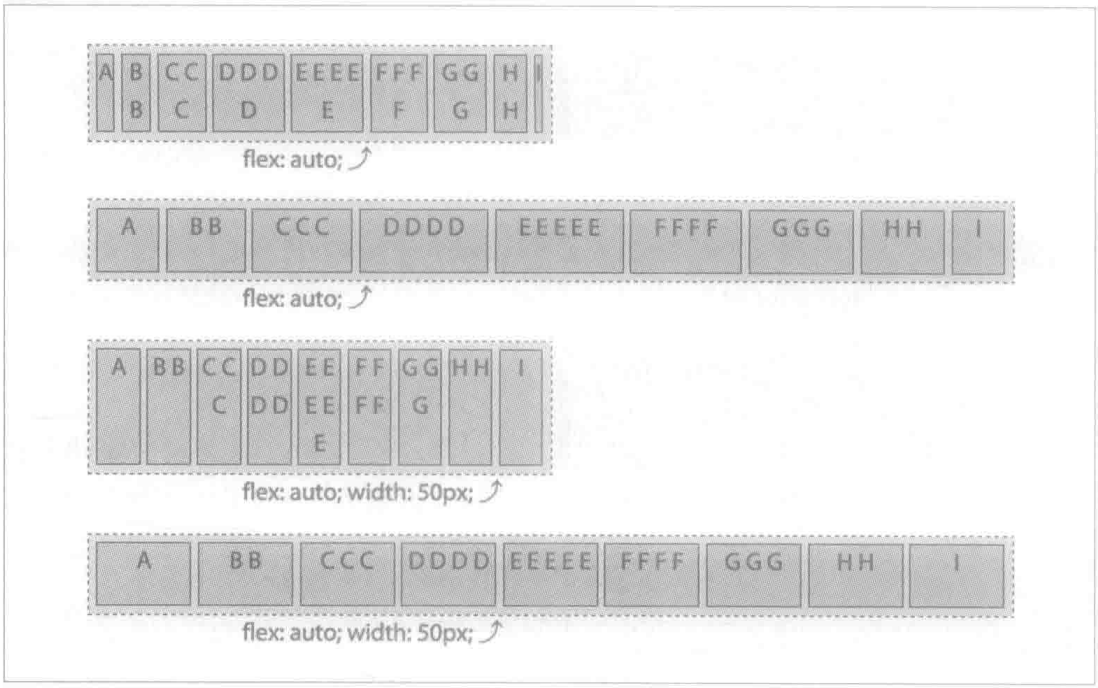


图 12-60：声明 flex: auto 时，弹性元素可以增大也可以缩小 ▶

```
flex: none;
flex: 0 0 auto;
```

none 的效果如图 12-61 所示。

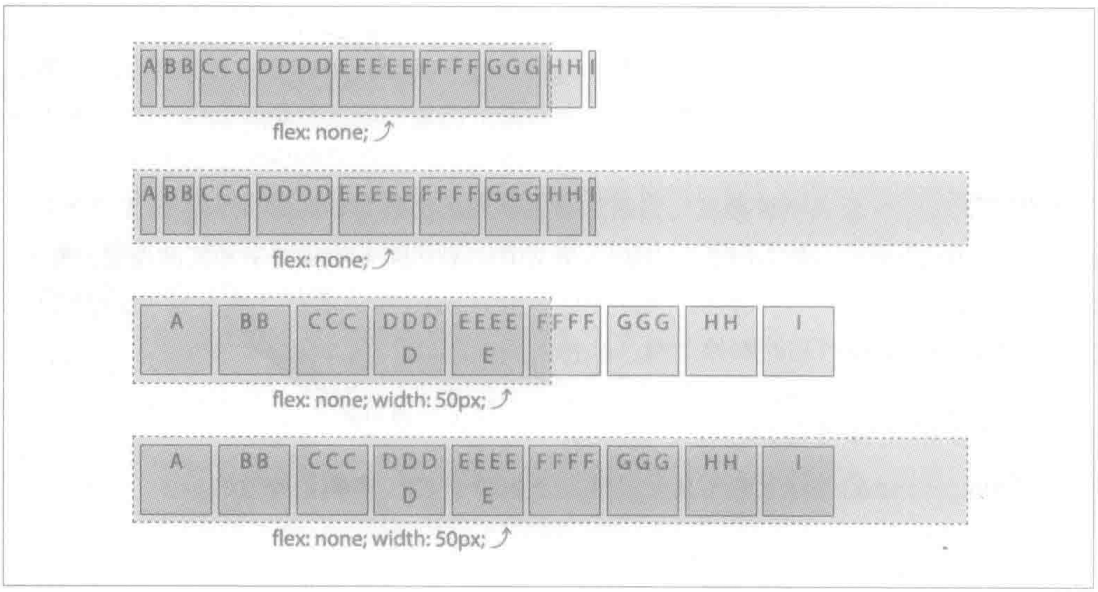


图 12-61：声明 flex: none 时，弹性元素既不增大也不缩小 ▶

如图 12-61 中的第一个和第三个例子所示，如果没有足够的空间，弹性元素将从弹性容器中溢出。这与 flex: initial 和 flex: auto 不同，因为它们都设置了值为正数的缩减因子。

基准的值解析为 auto，这意味着弹性元素在主轴方向上的尺寸由未变成弹性元素前元素的主轴尺寸决定。此时，弹性基准等于元素的 width 或 height 值。如果 width 或 height 的值为 auto，基准等于内容在主轴方向上的尺寸。在前两个例子中，基准等于内容的宽度，弹性元素的宽度也等于内容的宽度，因为元素不能增大也不能缩小。在第三个和第四个例子中，宽度和基准都是 50 像素，因为显式声明了 width 属性。

数字值

如果 flex 属性只有一个值，而且是正数，那个值将用作增长因子，而缩减因子默认为 0，基准也默认为 0。下面两个 CSS 声明是等效的：

```
flex: 3;
flex: 3 0 0;
```

此时，目标弹性元素具有弹性，即可以增大。而缩减因子是什么值都没有意义，因为弹性基准为 0，弹性元素只能从那个基准开始增大。

在图 12-62 中的前两个例子里，所有弹性元素的弹性增长因子都是 3。因为弹性基准为 0，所以弹性元素无法“缩小”。各弹性元素将从零像素开始增大，而且增大的量一样，直到全部弹性元素在主轴方向上的尺寸等于弹性容器的主轴尺寸为止。因为每个弹性元素的基准都是 0，所以 100% 的主轴尺寸都用于分配。第二个例子中弹性元素在主轴方向上的尺寸较大，这是因为宽度更大的弹性容器有更多的空间可分配。

任何大于 0 的数值，甚至是 0.1，都意味着弹性元素能增大。如果有可用的空间，而且只有一个弹性元素的增长因子为正数，那么那个元素将获得全部可用空间。如果有多个弹性元素允许增大，多出的可用空间将根据各弹性元素的增长因子按比例分配给各元素。

在图 12-62 中的后三个例子中，弹性容器中有六个弹性元素，分别声明了 flex: 0、flex: 1、flex: 2、flex: 3、flex: 4 和 flex: 5。这是各弹性元素的增长因子，而缩减因子均为 1，弹性基准均为 0。每个弹性元素在主轴上的尺寸根据各自的弹性增长因子按比例分配。你可能以为第三个和第四个例子中声明了 flex: 0 的元素，即显示文本“flex: 0”的那个元素的宽度为零像素，就像第四个和第五个例子那样，但是默认情况下，弹性元素缩小的程度不会比内容里最长的单词或固定尺寸的元素小。

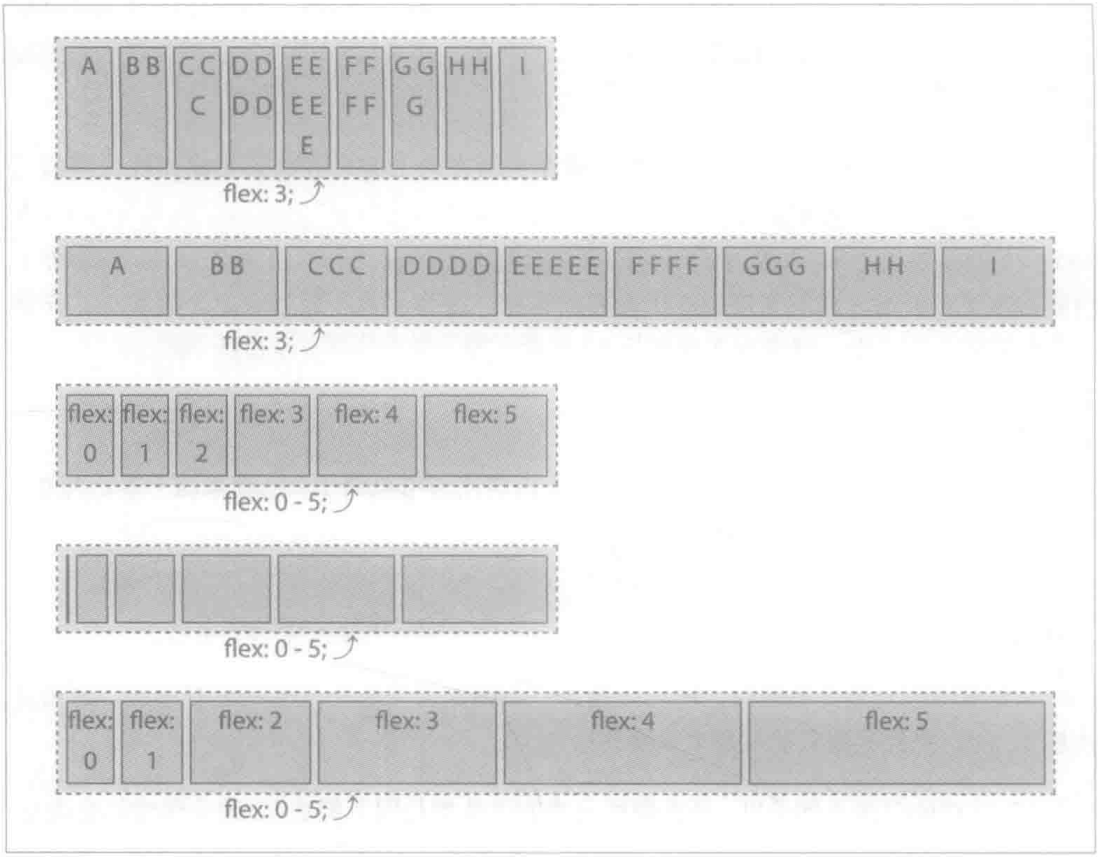


图 12-62：使用单个数字值时的弹性变形情况 ▶



为了看得更清楚，添加了一点内边距、外边距和边框。鉴于此，最左边的弹性元素（即声明 flex：0 的那个）是可见的：由于有 1 像素边框，即使宽度为零也可见。

12.16 order 属性

默认情况下，弹性元素的显示和排布顺序与在源码中的顺序一致。弹性元素和弹性元素行的顺序可以使用 flex-direction 属性反转，但是有时你可能需要更细致的重排方式。order 属性用于修改单个弹性元素的显示顺序。

order

取值	*<integer>*
初始值	0
适用于	弹性元素以及弹性容器中绝对定位的子元素
计算值	指定的值
继承性	否
动画性	是

默认情况下，所有弹性元素的顺序都是 0，归在同一个排序组中，以出现在源码中的顺序沿主轴方向显示（本章目前所举的例子都是如此）。

若想修改弹性元素的视觉顺序，把 order 属性设为一个非零整数。如果 order 属性的目标元素不是弹性容器的子元素，元素不受影响。

order 属性的值指定一个排序组，把目标弹性元素归在其中。在页面中绘制弹性元素时，order 属性为负数的弹性元素显示在采用默认值 0 的弹性元素前面，而 order 属性为正数的弹性元素显示在采用默认值 0 的弹性元素后面。虽然弹性元素的视觉顺序变了，但是在源码中的顺序没变。屏幕阅读器的阅读顺序和 Tab 键的索引顺序都由 HTML 源码顺序定义。

例如，在一组 12 个元素中，如果想让第 7 个元素显示在第一位，让第 6 个元素显示在最后一位，可以这样声明：

```
ul {
  display: inline-flex;
}
li:nth-of-type(6) {
  order: 1;
}
li:nth-of-type(7) {
  order: -1;
}
```

这里，我们显式设定了第 6 个和第 7 个列表项目的顺序，而其他列表项目则默认为 order: 0。得到的结果如图 12-63 所示。

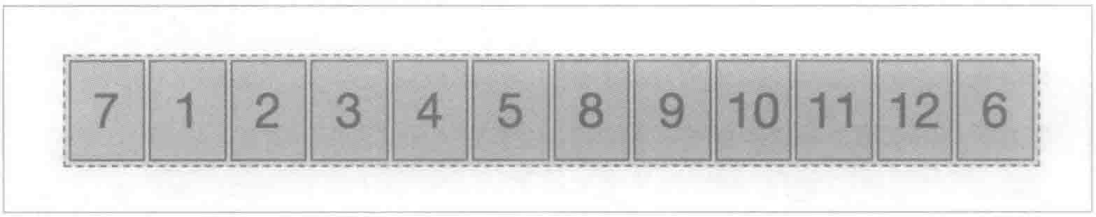


图 12-63：使用 order 属性重新排列弹性元素的顺序 ▶

第 7 个弹性元素排在第一位，因为它的 order 属性为负数，比默认值 0 小，而且也是同辈弹性元素中最小的。第 6 个弹性元素是唯一一个 order 值大于零的元素，因此在同辈中序号是最大的。正因为如此，第 6 个弹性元素才排在所有元素的最后一位。其他几个弹性元素的 order 属性都为默认值 0，在首尾两个元素之间以出现在源码中的顺序绘制，因为它们同属一个排序组（0）。

弹性容器以修改后的文档顺序排布内容，从最小的排序组开始往大的排序组显示。如果几个弹性元素的 order 属性值一样，那么它们属于同一个排序组。同一排序组中的元素按照出现在源码中的顺序显示，然后再按照排序组的序号从小到大显示。以下述样式为例：

```
ul {
  display: inline-flex;
  background-color: rgba(0,0,0,0.1);
}
li:nth-of-type(3n-1) {
  order: 3;
  background-color: rgba(0,0,0,0.2);
}
li:nth-of-type(3n+1) {
  order: -1;
  background-color: rgba(0,0,0,0.4);
}
```

这里，多个弹性元素的 order 值是一样的，归在同一个排序组中，组内的各元素按照源码顺序显示。结果如图 12-64 所示。

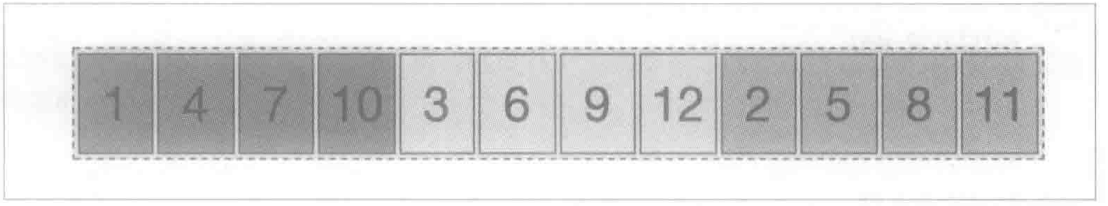


图 12-64：弹性元素按照排序组的顺序显示，在组内则按照源码顺序显示 ▶

具体情况是：

- 指定 2、5、8 和 11 号元素在 3 号排序组中，背景为 20% 不透明。
- 指定 1、4、7 和 10 号元素在 -1 号排序组中，背景为 40% 不透明。
- 3、6、9 和 12 号元素未指定排序组，因此在默认的 0 号排序组中。

三个排序组的序号分别为 -1、0 和 3，它们就按照这个顺序显示。在各组内部，元素按照源码顺序布置。

这种重新排序纯粹是视觉上的。屏幕阅读器应该按照源码顺序阅读文档，然而实际上可能不会这么做。虽然这是视觉上的变化，但是弹性元素的顺序对页面的绘制顺序是有影响的：按照出现的顺序绘制弹性元素，就像在源文档中的顺序也变了一样（其实没变）。

使用 `order` 属性修改布局后，页面中的 Tab 键索引顺序不受影响。如果图 12-64 中的数字是链接，通过 Tab 键选择链接时将按照源码顺序，而不是布局的显示顺序。

12.16.1 标签式导航再谈

利用 `order` 属性，我们可以改进图 12-2 中的标签式导航栏，把当前激活的标签显示在第一位，如图 12-65 所示：

```
nav {
  display: flex;
  justify-content: flex-end;
  border-bottom: 1px solid #ddd;
}
```

```
a {
  margin: 0 5px;
  padding: 5px 15px;
  border-radius: 3px 3px 0 0;
  background-color: #ddd;
  text-decoration: none;
  color: black;
}
a:hover {
  background-color: #bbb;
  text-decoration: underline;
}
a.active {
  order: -1;
  background-color: #999;
}
<nav>
  <a href="/">Home</a>
  <a href="/about">About</a>
  <a class="active">Blog</a>
  <a href="/jobs">Careers</a>
  <a href="/contact">Contact Us</a>
</nav>
```



图 12-65：order 属性修改的是视觉顺序，而不是 Tab 键的索引顺序 ▶

当前激活的标签添加了 .active 类，删除了 href 属性，order 属性被设为 -1，这比其他同辈弹性元素的默认值 0 小，所以显示在首位。

为什么要删除 href 属性呢？因为选中的标签是当前显示的文档，没有什么理由把文档链接到自己身上。但是更重要的原因是，如果选中的标签仍是链接，那么用户使用键盘索引的顺序为 Blog、Home、About、Careers 和 Contact Us，Blog 在第一位；但是通过 Tab 键索引的顺序为 Home、About、Blog、Careers 和 Contact Us，与出现在源码中的顺序一样，这可能让用户困惑不解。

利用 order 属性可以把正常情况下的三栏布局，即中间一栏为主内容区，左边为导航、右边为侧边栏（见图 12-50），在移动设备中及使用屏幕阅读器和其他辅助技术时把主内容区放在侧栏前面。

在标记中虽然可以把页脚放在页头前面，然后使用 order 属性重新排序，但这并不是 order 属性的恰当用法。order 属性只应该用于改变内容的视觉顺序，而标记始终反应内容的逻辑顺序：

```
<header></header>
<main>
   <article></article>
   <aside></aside>
   <nav></nav>
</main>
<footer></footer>
```

```
<header></header>
<main>
  <nav></nav>
  <article></article>
  <aside></aside>
</main>
<footer></footer>
```

我们以内容出现在网站中的顺序编写标记，如上述代码示例的右边所示，这与图 12-50 中的三栏布局示例一样。但是，像左边那样编写标记可能更合理，即在源码中把 article 表示的主内容区放在首位。这样，屏幕阅读器、搜索引擎和移动设备都将首先看到文章，但在较大的屏幕中，视觉正常的人看到的文章则位于中间：

```
main {
  display: flex;
}
main > nav {
  order: -1;
}
```

声明 order: -1 后，nav 元素将出现在首位，因为它是 -1 这个排序组中唯一的弹性元素。article 和 aside 都没显式声明 order，所以默认为 order: 0。

记住，同一排序组中有多个弹性元素时，元素按照源码顺序从主轴起边向主轴终边显示。因此，article 显示在 aside 前面。

有些开发者在修改弹性元素的顺序时，喜欢为所有弹性元素设置 order 值，以便更好地理解样式的作用。因此，上述样式还可以写成：

```
main {
  display: flex;
}
main > nav {
  order: 1;
}
main > article {
  order: 2;
```

```
}
main > aside {
  order: 3;
}
```

几年前，浏览器还不支持弹性盒布局时，这些效果都使用浮动实现，例如为nav元素声明float:right。虽然使用浮动也可以，但是弹性盒布局简单得多，尤其是希望三栏（aside、nav和article）等高时。

第 13 章

栅格布局

CSS 出现这些年来（你可能不信，已经 20 年了），布局一直是其最为重要的功能。我们已经讨论过其他布局方式，例如 `float` 和 `clear`，不过这并不是它们的恰当用法。弹性盒布局填补了一些空白，但只适用于特定的情况，例如导航栏。

与之相比，栅格布局是普适的布局系统。由于栅格布局依赖行和列，初一看似乎倒退到表格布局了（在某些方面二者之间的差异并不明显），但是栅格布局比表格布局强大得多得多。使用栅格设计布局，无需顾虑各部分在源文档中的顺序，如果愿意，不同的部分还可以重叠在一起。有很多灵活的方法能为栅格线定义重复模式，以及把元素依附到栅格线上等。栅格之内可以嵌套栅格，而且还可以在栅格中使用表格或弹性容器。这只是举些例子，栅格布局的功能还有很多很多。

总之，栅格布局是我们翘首以盼的布局系统。要学的东西很多，要抛诸脑后的东西可能更多，是时候告别过去 20 年自作聪明的小技巧和变通方法了。

13.1 创建栅格容器

创建栅格的第一步是定义栅格容器（grid container）。这与定位所用的容纳块和弹性盒布局中的弹性容器的作用很像：栅格容器为其中的内容定义一个栅格格式化上下文（grid formatting context）。

从这一点上看，栅格布局从弹性盒布局上沿袭了相当多的概念。例如，栅格容器的子元素是栅格元素（grid item），就像弹性容器的子元素是弹性元素一样。子元素的子元素

不是栅格元素，不过栅格元素自身也可以变作栅格容器，因此它的子元素将变成嵌套栅格的栅格元素。栅格之中可以嵌套栅格，而且层级不限（栅格布局还有子栅格这个概念，它与嵌套的栅格容器不是一回事，稍后讨论）。

栅格有两种：常规栅格和行内栅格。这两种栅格使用 `display` 属性的特殊值创建：`grid` 和 `inline-grid`。前者生成块级框，后者生成行内框。二者之间的区别如图 13-1 所示。

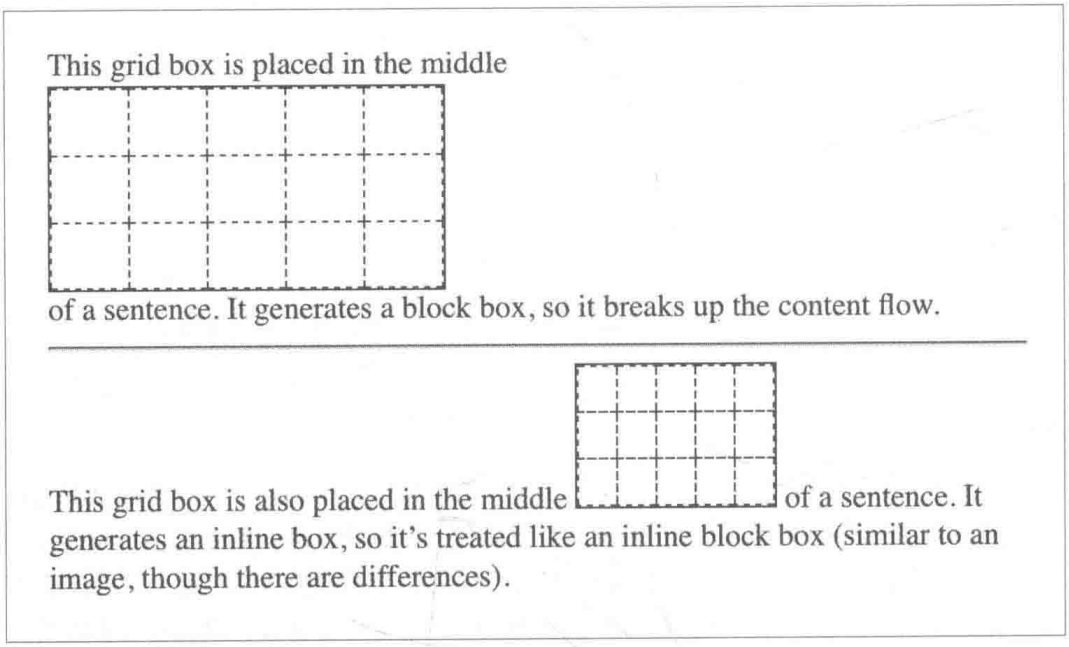


图 13-1：常规栅格和行内栅格

这与 `display` 属性的 `block` 和 `inline-block` 值十分相似。多数栅格都是块级的，不过你要知道也有创建行内栅格这一选择。

虽然 `display:grid` 创建的是块级栅格，但是严谨的规范明确指出，“栅格容器不是块级容器”。意思就是，栅格框在布局中的行为与块级容器很像，但是二者之间仍有诸多区别。

首先，浮动的元素不会打乱栅格容器。这意味着，栅格不会移到浮动元素的下方，而块级容器会。这一差异如图 13-2 所示。

其次，栅格容器的外边距不与其后代的外边距折叠。而块级框的外边距（默认）与其后代的外边距折叠，这是栅格容器与块级框的又一区别。例如，有序列表的第一个列表项目可能有上外边距，但这个外边距将与列表元素的上外边距折叠。然而，栅格元素的上外边距绝不会与栅格容器的上外边距折叠。这一差异如图 13-3 所示。

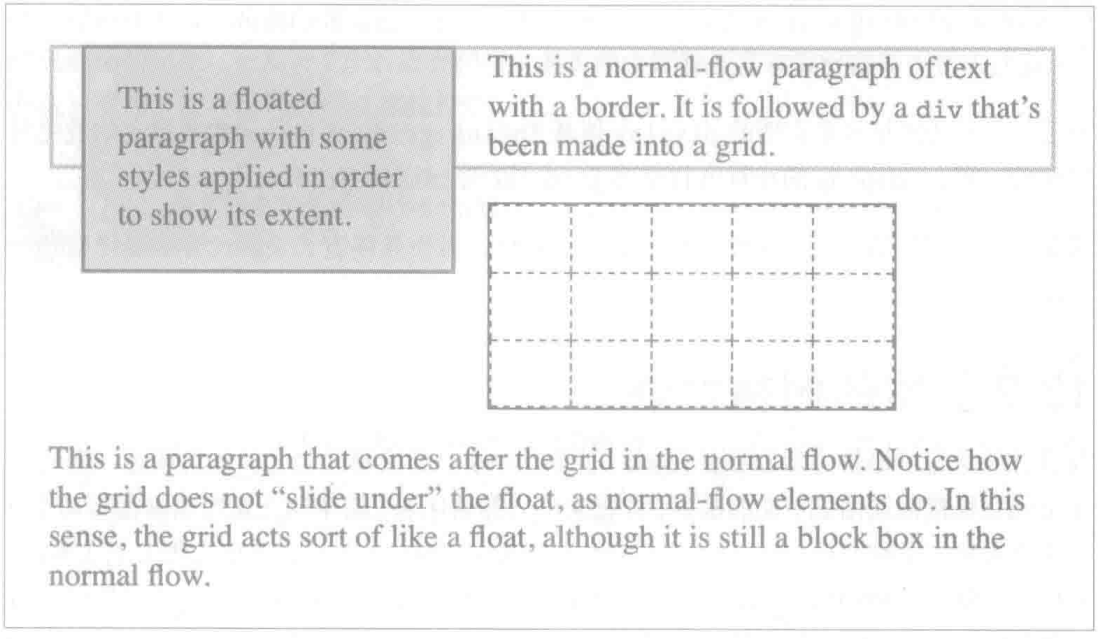


图 13-2：浮动元素对块级容器和栅格容器的影响不一样

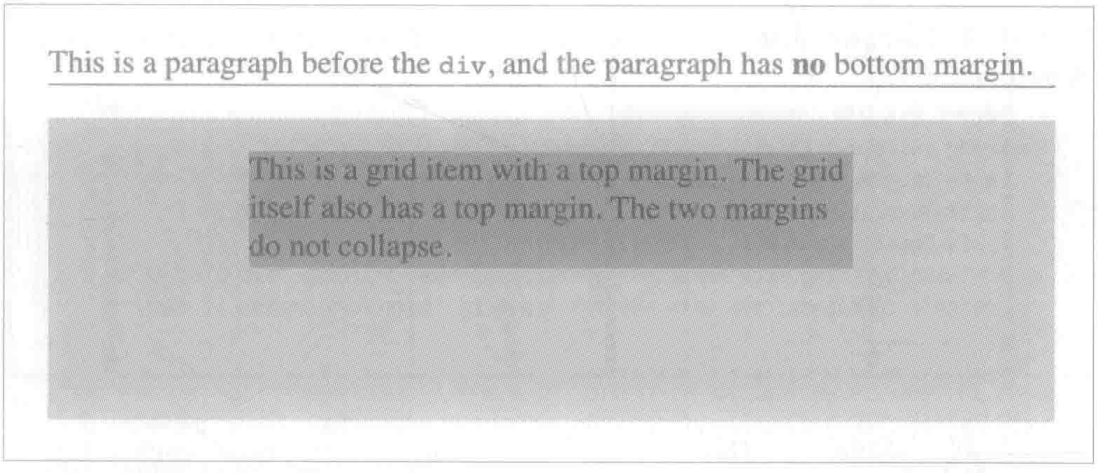


图 13-3：外边距不折叠

有些 CSS 属性和功能不能用在栅格容器和栅格元素上，如下：

- 栅格容器上的所有 column 属性（例如 column-count、columns 等）都被忽略。
- 栅格容器没有 ::first-line 和 ::first-letter 伪元素，如果使用，将被忽略。
- 栅格元素（而非栅格容器）上的 float 和 clear 属性将被忽略。尽管如此，float 属性对栅格容器中子元素的 display 属性的计算值是有影响的，因为栅格元素的 display 值在变成栅格元素之前计算。

- vertical-align 属性对栅格元素不起作用，不过可能会影响栅格元素中的内容（别担心，对齐栅格元素有其他更强大的方式）。

最后，如果为栅格容器声明的 display 值是 inline-grid，而目标元素是浮动的或绝对定位的，那么 display 的计算值将变为 grid（取代 inline-grid）。

定义好栅格容器后，接下来要在容器中设置栅格。讨论具体方式之前，有必要说明几个术语。

13.2 基本的栅格术语

前面已经讨论过栅格容器和栅格元素，下面为其做个更准确的定义。如前所述，栅格容器是确立栅格格式化上下文的框体，即定义一个栅格区域，其中的元素根据栅格布局（而非块级布局）规则排布。这一点可以与通过 display:table 创建表格格式化上下文类比。表格自身就是一种栅格系统，因此这样类比还是相当合适的，但是不要以为栅格只是另一种形式的表格。栅格比表格强大得多。

栅格元素是在栅格格式化上下文中参与栅格布局的东西。这通常是栅格容器的子元素，但也可以是元素内容中的匿名文本（即不在元素中的文本）。来看下述代码，得到的结果如图 13-4 所示：

```
#warning {display: grid;
    background: #FCC; padding: 0.5em;
    grid-template-rows: 1fr;
    grid-template-columns: repeat(7, 1fr);}
<p id="warning"><img src="warning.svg"><strong>Note:</strong> This element is a
   <em>grid container</em> with several <em>grid items</em> inside it.</p>
```

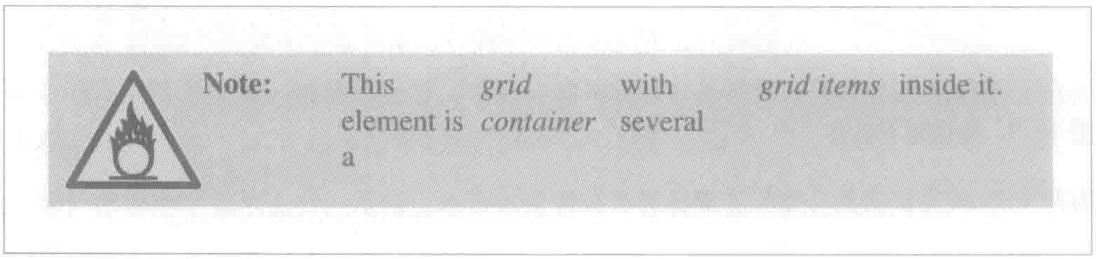


图 13-4：栅格元素

注意，各元素及元素之间的文本都变成栅格元素了。图像是栅格元素，其他元素和文本块也是栅格元素，一共有 7 个。这些栅格元素都参与栅格布局，然而匿名文本块难以（或无法）使用下文将讨论的栅格属性控制。



你可能想知道 grid-template-rows 和 grid-template-columns 的作用，别急，下一节讨论。

在使用栅格属性的过程中，可能会创建或引用栅格布局的多个核心组件，如图 13-5 所示。

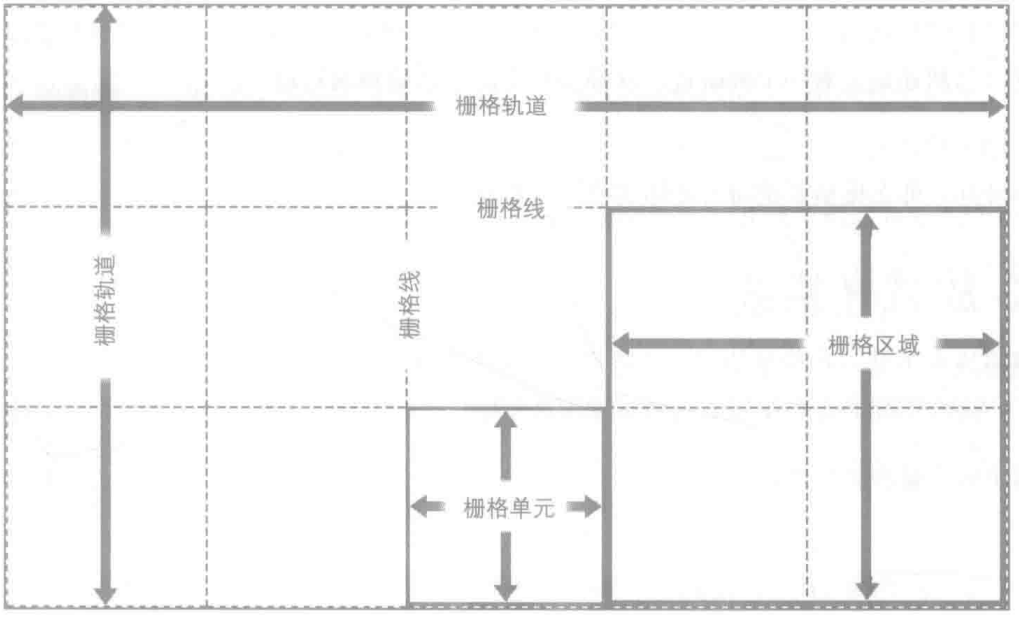


图 13-5：栅格组件

最重要的组件是栅格线。栅格线的位置定义好之后，其他栅格组件也就随之而现了：

- 栅格轨道（grid track）指两条相邻的栅格线之间夹住的整个区域，从栅格容器的一边延伸到对边，即栅格列或栅格行。栅格轨道的尺寸由栅格线的位置决定。可以对比表格中的列和行理解。用适用性更广的语言来说，可以称之为块级轴和行内轴轨道，（对西方语言来说）列轨道在块级轴上，行轨道在行内轴上。

- 栅格单元（grid cell）指四条栅格线限定的区域，内部没有其他栅格线贯穿，类似于单元格。这是栅格布局中区域的最小单位。栅格单元不能直接使用CSS栅格属性处理，即没有属性能把一个栅格元素放在指定的栅格单元里（详情参见下一点）。
- 栅格区域(grid area)指任何四条栅格线限定的矩形区域，由一个或多个栅格单元构成。最小的栅格区域是一个栅格单元，最大的栅格区域是栅格中所有的栅格单元。栅格区域能使用CSS栅格属性直接处理，定义好栅格区域后即可在其中放置栅格元素。

特别注意，栅格轨道、栅格单元和栅格区域都完全由栅格线建构，不一定非要有相应的栅格元素存在。栅格区域中不一定充满栅格元素，完全可以让部分甚至多数栅格单元空着。此外，栅格元素还可以重叠，方法是定义重叠的栅格区域，或者把栅格线重叠起来。

另外要注意的一点是，栅格线的数量不限，想定义多少就可以定义多少。你可以只定义一系列纵向的栅格线，创建一行多列布局。你也可以反过来，创建多个行轨道但不创建列轨道（当然还是会有一个列轨道，从栅格容器的一边延伸到对边）。

然而，如果栅格元素无法放入你定义的列或行轨道中，或者你明确指定把栅格元素放在轨道的外部，那么栅格系统将自动添加栅格线和轨道。

13.3 放置栅格线

放置栅格线可不是一件简单的事。不是说这个概念有多难，而是因为栅格线的放置方式太多了，而且不同的方式使用的句法有细微的差别。

先看两个联系紧密的属性：

grid-template-rows, grid-template-columns

取值	none | *<track-list>* | *<auto-track-list>*
初始值	none
适用于	栅格容器
百分数	grid-template-columns 的百分数值相对栅格容器的行内轴尺寸（通常为宽度）计算，grid-template-rows 的百分数值相对栅格容器的块级轴尺寸（通常为高度）计算
计算值	声明的值，长度值计算为绝对长度
继承性	否
动画性	否

使用这两个属性可以大致定义栅格模板（grid template，CSS 规范称之为 *explicit grid*，显式栅格）中的栅格线。栅格中的一切都依赖栅格线，如果放置不当，整个布局轻易就会垮掉。



刚接触 CSS 栅格布局时，建议先在纸上或其他电子工具上把自己的想法画出来。有了参考，在使用 CSS 实现栅格时便能轻易看出栅格线的位置及其之间的关系。

<track-list> 和 *<auto-track-list>* 的具体句法十分复杂，而且可以嵌套很多层，若想说清要用大量时间和篇幅，因此最好着重讨论涉及的理论。指定栅格线的位置有很多方式，学习具体的方法之前有必要了解一些基础知识。

首先，栅格线始终可以使用数字引用，此外创作人员也可以为其命名。以图 13-6 中的栅格为例。在 CSS 中可以使用数字引用栅格线，也可以使用为其指定的名称，又或者二者混用。因此，你可以说一个栅格元素从竖线 3 延伸到线 `steve`，或者从横线 `skylight` 延伸到线 2。

注意，一条栅格线可以有多个名称。你可以使用栅格线的任何一个名称引用它，但是不能像类名那样连在一起使用。你可能以为这意味着最好别重复使用相同的名称命名栅格线，但稍后你将看到，其实并非全然如此。

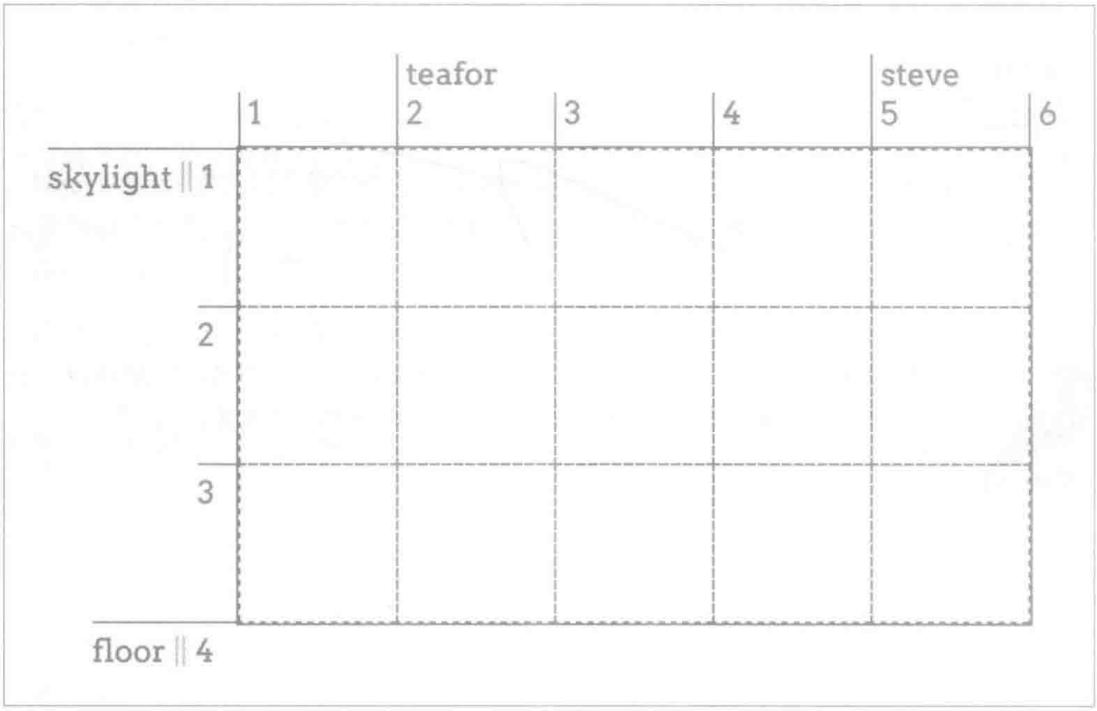


图 13-6：栅格线的编号和名称

在图 13-6 中，我故意用了看起来傻傻的名称，这是为了表明你可以选择自己喜欢的任何名称，也是为了指出栅格线没有所谓的默认名称。倘若我把第一条栅格线命名为 `start`，你有可能以为第一条栅格线始终要用这个名称。事实并非如此。如果你想让一

个栅格元素从start线延伸到end线，你要自己定义哪两条栅格线使用这两个名称。幸好，这并不难。

前文说过，定义栅格模板的值有很多模式。下面从易到难逐一讨论。

13.3.1 宽度固定的栅格轨道

首先说明如何创建栅格轨道的宽度是固定的栅格。这里所说的“固定”，不单指固定的像素或em长度，百分数也算固定宽度。“宽度固定”的栅格线指栅格线之间的距离不随栅格轨道中内容的变化而变。

下面举个例子。以下声明定义的是三个宽度固定的栅格列：

```
#grid {display: grid;
    grid-template-columns: 200px 50% 100px;}
```

第一条栅格线在距栅格容器起边（默认为左边）200像素的位置，第二条栅格线距第一条栅格线的距离为栅格容器宽度的一半，第三条栅格线距第二条栅格线100像素，如图13-7所示。

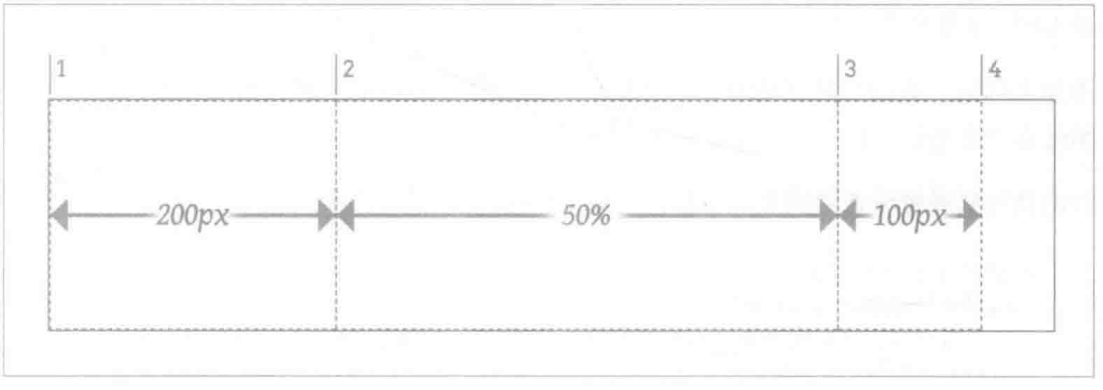


图13-7：栅格线的位置

虽然第二列的尺寸会随栅格容器的尺寸变化，但是不会随这一列中栅格元素的内容而变。不管第二列中的内容多宽多窄，列的宽度始终为栅格容器宽度的一半。

此外，最后一条栅格线没有接触栅格容器的右边界。这没关系，没有规定必须接触。然而，如果你想让它们接触，稍后会介绍多种方法。

很好，我们得到了所需的栅格，但是如果想为栅格线命名呢？只需把想用的名称放在值中的恰当位置，并在两侧加上方括号。名称的数量不限，想要多少个都可以。下面在前例的基础上为栅格线添加一些名称，结果如图 13-8 所示：

```
#grid {display: grid;
    grid-template-columns:
        [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
    }
```



图 13-8：栅格线的名称

添加名称后，值的句法更清晰了：带数字的值设定的是栅格轨道的宽度，而栅格线两侧始终有个宽度值。因此，宽度值为 3 个时，得到的栅格线有 4 条。

行栅格线的放置方式与列完全一样，如图 13-9 所示：

```
#grid {display: grid;
    grid-template-columns:
        [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
    grid-template-rows:
        [start masthead] 3em [content] 80% [footer] 2em [stop end];
    }
```

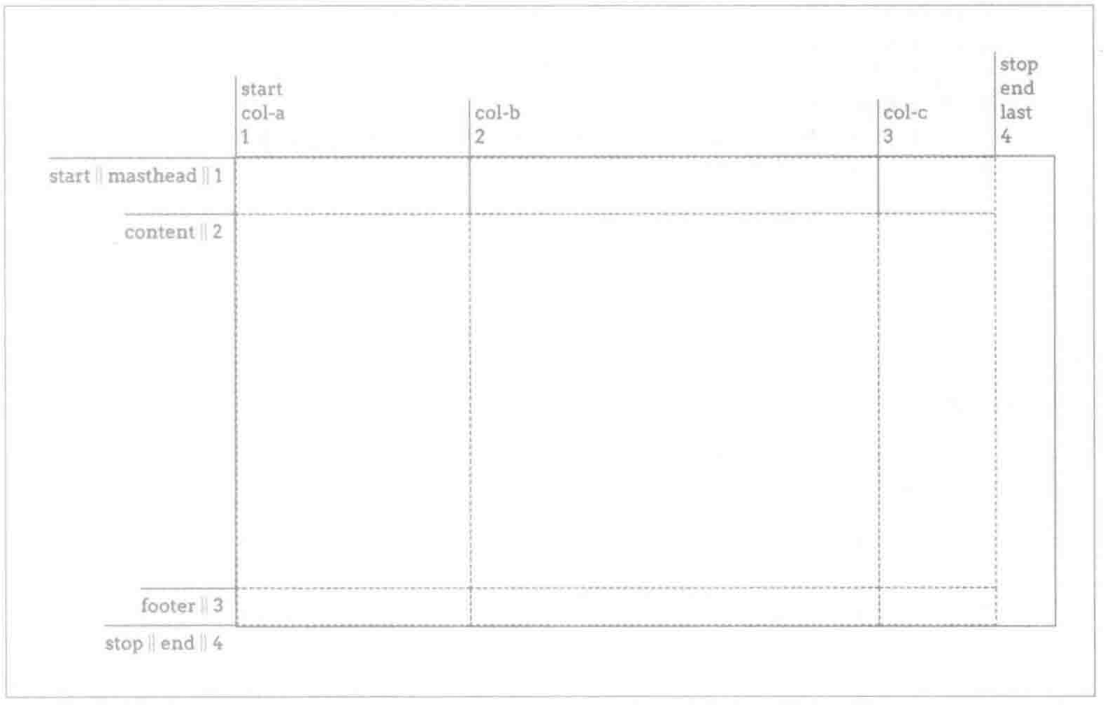


图 13-9：创建一个栅格

这里有几点要注意。首先，列和行中都有名为 start 和 end 的栅格线。这完全没问题。行和列不共用命名空间，因此可以像这样在两个上下文中重用名称。

其次，注意 content 行轨道的百分数值，它相对栅格容器的高度计算。因此，对高度为 500 像素的容器来说，content 行的高度为 400 像素。当然，你要事先知道栅格容器的高度，但实际情况并非总是如此。

你可能以为使用 100% 能让一行占满全部空间，其实不然，如图 13-10 所示：content 行轨道的高度与弹性容器相等，因此 footer 行轨道将被完全推到容器外部。

```
#grid {display: grid;
    grid-template-columns:
        [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
    grid-template-rows:
        [start masthead] 3em [content] 100% [footer] 2em [stop end];
    }
```

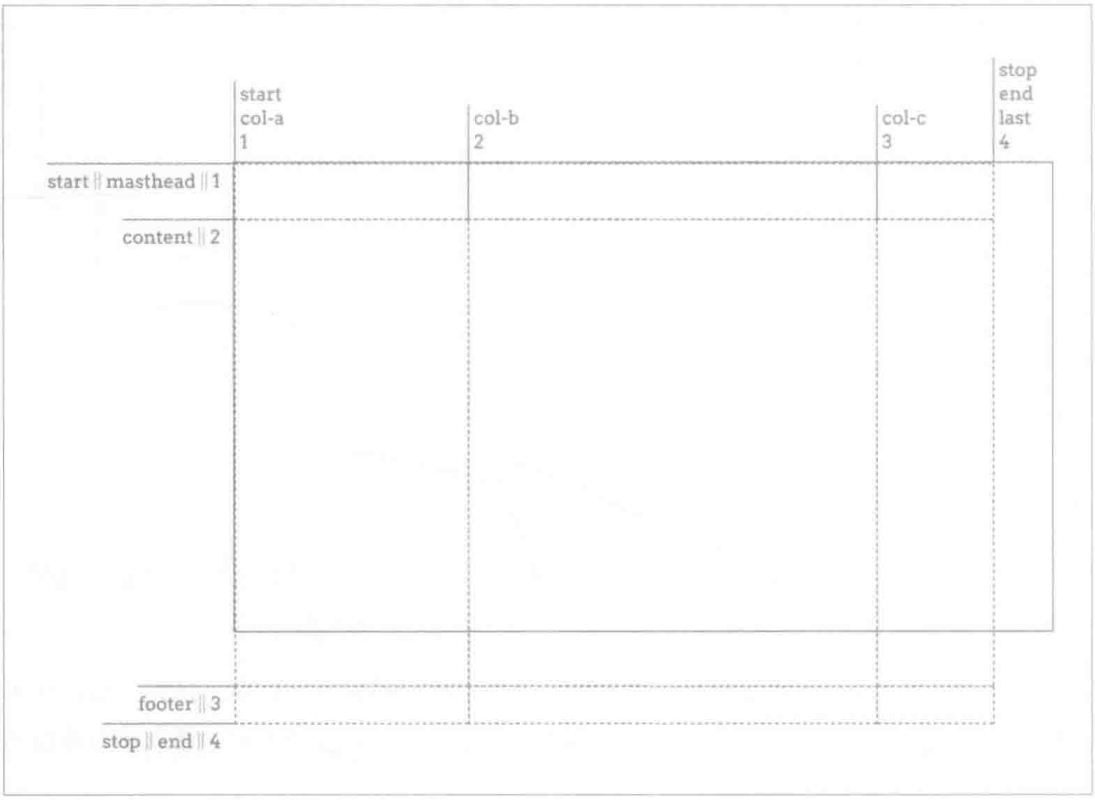


图 13-10：超出栅格容器

这种问题的处理方式之一是（可能不是最好的方式），为行的尺寸设定极值，指明行的高度不能小于一个值，也不能大于一个值，让浏览器计算具体的值。这种方法所用的句法是 minmax(a,b)，其中 a 是最小尺寸、b 是最大尺寸。

```
#grid {display: grid;
    grid-template-columns:
        [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
    grid-template-rows:
        [start masthead] 3em [content] minmax(3em,100%) [footer] 2em [stop end];
    }
```

这样做的意思是，content 行的高度不能小于 3em，而且不能大于栅格容器的高度。此时，浏览器会增加 content 行的高度，直到占满 masthead 和 footer 轨道以外的空间为止。如有必要，浏览器也会减少 content 行的高度，但不会小于 3em。也就是说，具体高度视情况而定。图 13-11 展示的是其中一种可能的结果。

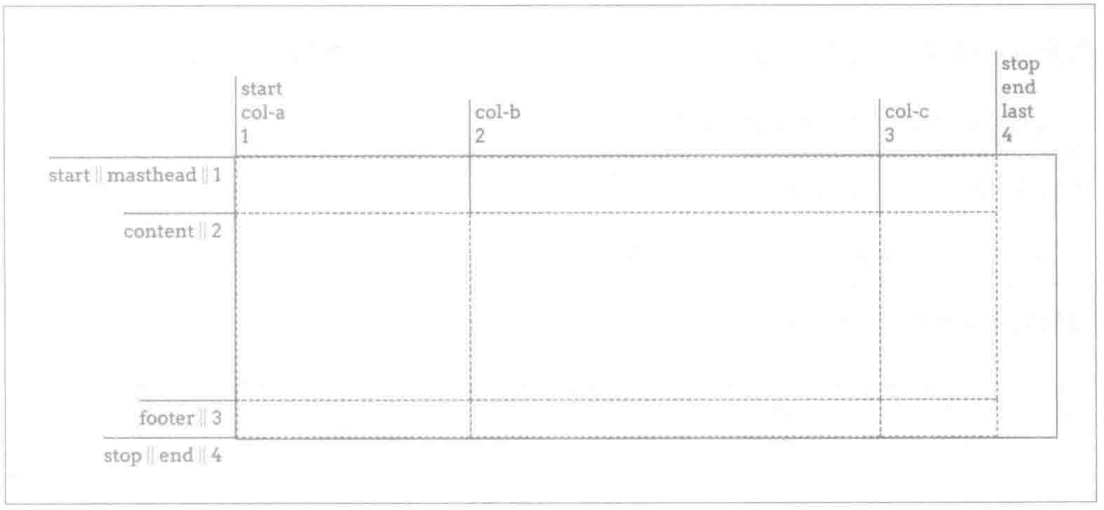


图 13-11：适应栅格容器的尺寸

类似地，可以使用 minmax() 让 col-b 列占满栅格容器的横向空间。注意，如果 minmax() 中指定的最大值比最小值小，最大值将被忽略，最小值将用于设定宽度固定的轨道长度。因此，minmax(100px,2em) 在字号小于 50px 时将解析为 100px。

如果 minmax() 含糊的行为让你不安，可以使用其他替代方式。轨道的高度（或宽度）还可以使用 calc() 计算。例如：

```
grid-template-rows:
    [start masthead] 3em [content] calc(100%-5em) [footer] 2em [stop end];
```

此时，content 行的高度等于栅格容器的高度减去 masthead 和 footer 两行的高度，如前面那张插图所示。

calc() 用着不错，但有时不那么牢靠，因为修改 masthead 或 footer 行的高度后，要调整算式。此外，如果想让一列弹性变形，使用 calc() 很难实现（有时不可能实现）。当然，这种问题有更可靠的处理方式，具体参见下文。

13.3.2 弹性栅格轨道

目前所见的栅格轨道都不具有弹性，其尺寸由长度值或栅格容器的尺寸确定，不受其他

因素的影响。与之相比，弹性栅格轨道的尺寸基于弹性容器中非弹性轨道以外的空间确定，或者基于整个轨道中的具体内容而定。

份数单位

如果想把余下的空间分成一定份数，分配给各栏，可以使用 fr 单位。

最简单的情况是，把整个容器平均分成几等份。例如，如果想要四列，可以这样声明：

```
grid-template-columns: 1fr 1fr 1fr 1fr;
```

就这个示例而言，它等效于：

```
grid-template-columns: 25% 25% 25% 25%;
```

得到的结果如图 13-12 所示。

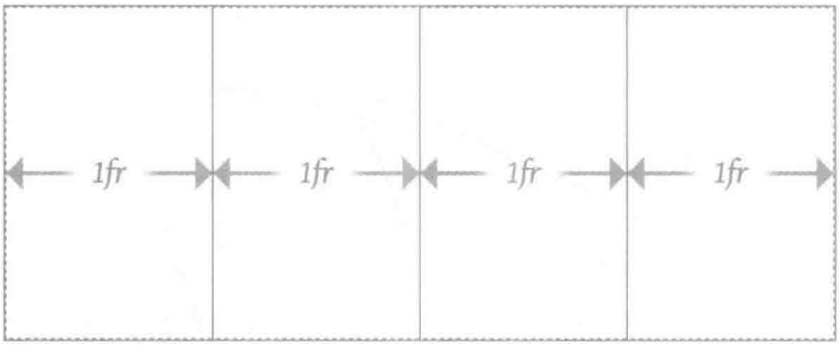


图 13-12：把容器分成四列

现在，假设我们想增加一列，而且确保每一列的宽度仍是相等的。如果使用百分数，我们要重写整个值，把各列的尺寸改为 20%。但是使用 fr 的话，只需再加一个 1fr 即可：

```
grid-template-columns: 1fr 1fr 1fr 1fr 1fr;
```

处理 fr 单位的方式是，拿可用空间除以 fr 值之和，各轨道的尺寸等于 fr 值所对应的份数。

因此，对第一个例子来说，四个 fr 值相加等于 4，可用空间除以 4，所以每一列的宽度为总空间的四分之一。添加一个 1fr 后，可用空间要除以 5，那么每一列的宽度为总空间的五分之一。

当然，fr 单位前的数字不一定总是 1。假如你想得到三列，中间一列的宽度为其他两列的两倍。那么，可以这样声明：

```
grid-template-columns: 1fr 2fr 1fr;
```

同样，还是拿总空间除以总和，因此一个 fr 单位占空间的 0.25。如此以来，第一个和第三个轨道的宽度为容器宽度的 25%，而中间一列的宽度为容器宽度的一半，因为那一列的值为 2fr，即 0.25 的两倍，等于 0.5。

此外，不使用整数也可以。苹果派的食谱可能会像下面这样划分各列：

```
grid-template-columns: 1fr 3.14159fr 1fr;
```

计算过程留给你完成（这是个锻炼的好机会！万事开头难，但只要记住，先计算 1+3.14159+1）。

这样划分容器十分方便，但 fr 可不只是百分数的替代品这么简单，它还有更强大的功能。在某些列的尺寸固定，而部分空间弹性伸缩时，份数单位特别有用。请看下述声明，其结果如图 13-13 所示：

```
grid-template-columns: 15em 1fr 10%;
```

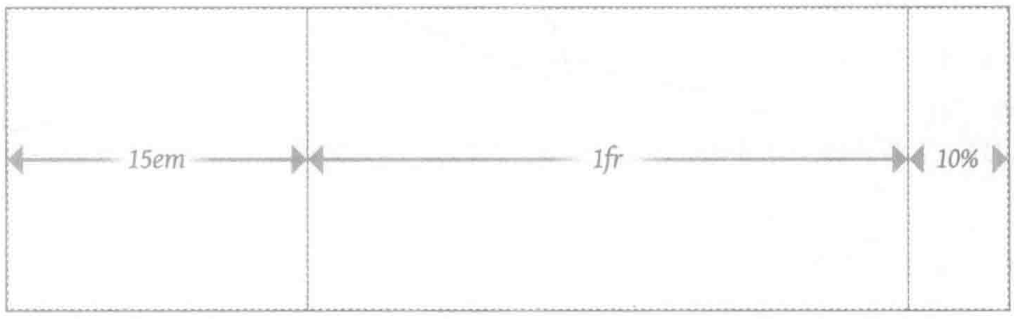


图 13-13：不管余下多少空间，都分给中间一列

这里，浏览器会为第一个和第三个轨道分配固定的宽度，而栅格容器中余下的空间，不管有多少，都分给中间那个轨道。因此，在 font-size 为浏览器通常的默认值 16px 时，

对一个1000像素宽的栅格容器来说，第一列的宽度为240像素，第三列的宽度为100像素。二者之和为340像素，因此有660像素没有分配出去。份数单位总和为一，即660除以1，得到660像素。这660像素都分给那个尺寸为1fr的轨道。如果栅格容器的宽度增加到1400像素，第三列的宽度将变成140像素，而中间一列的宽度为1020像素。

就这样，得到的栅格既有宽度固定的列，也有弹性伸缩的列。我们可以更进一步，把可用空间分成任意多份。例如：

```
width: 100em; grid-template-columns: 15em 4.5fr 3fr 10%;
```

此时，各列的宽度如图13-14所示。

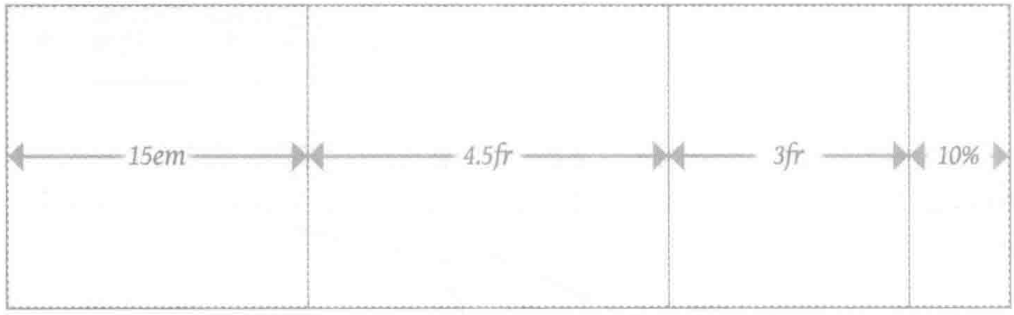


图13-14：列的尺寸弹性变化

从左至右，各列的宽度分别为15em、45em、30em和10em。第一列的宽度是固定的，为15em。最后一列的宽度是100em的10%，即10em。因此，有75em的空间要分给弹性变形的列。这两列的尺寸值相加为7.5fr。对较宽的那一列来说，4.5÷7.5等于0.6，因此其宽度为0.6乘以75em，等于45em。类似地，3÷7.5=0.4，75em的0.4倍等于30em。

我承认，准备这个例子时我事先算好了，确保fr之和及width的值正好能得到整数宽度。这么做纯粹是为了便于你理解。如果你不想使用这么巧合的数字，可以考虑把width的值改为92.5em或1234px。

如果想为轨道指定最小尺寸或最大尺寸，可以使用minxmax()。在前例的基础上，如果想确保第三列的宽度不小于5em，可以把CSS声明改为：

```
grid-template-columns: 15em 4.5fr minmax(5em,3fr) 10%;
```

现在，布局的中间两列是弹性变形的，第三列的最小宽度为5em。如果再变小，布局中将有三个宽度固定的列（宽度分别为15em、5em和10%），以及一列尺寸弹性变化的列，

其尺寸等于余下的空间（如果有余的话）。经过计算可以得知，弹性变形的那一列最宽为 30.5556em。超过这一宽度，布局中便会出现两个弹性变形的列。

为了限制列轨道的宽度最大能为多少，而超过那个值就变成固定宽度，你可能会想使用 fr 值声明最小值，但这样做并不能得到预期的结果，因为 minmax() 表达式的最小值部分不允许使用 fr 单位。因此，使用 fr 设定最小值将导致整个声明失效。

下面来看把最小值显式设为 0 的情况：

```
grid-template-columns: 15em 1fr minmax(0,500px) 10%;
```

图 13-15 展示的是第三列能为 500 像素宽的情况下栅格的最小宽度。如果栅格再变窄，设定极值那一列的宽度将小于 500 像素。栅格变宽的话，第二列（尺寸为 fr 值的那一列）的宽度将大于零，而第三列的宽度仍是 500 像素。



图 13-15：使用极值设定列的尺寸

仔细观察，你会在 15em 和 minmax(0,500px) 两列的交界处看到 1fr 标注。它之所以在那个位置，是由于 1fr 列的左边界在第二条列栅格线上，而那一列没有宽度，因为没有多余的空间可分配了。类似地，minmax 列在第三条列栅格线上。只是，在这个例子中，第二条列栅格线与第三条列栅格线在同一位置上（因而 1fr 列的宽度才是零）。

倘若最小值比最大值大，那么整个值将替换为最小值。因此，minmax(500px,200px) 将视为只有 500px 一个值。显然，你不会这么做，但是保不住混用百分数和份数时会出现这种情况。因此，你可以使用 minmax(10%,1fr) 设定一列的尺寸，当其宽度小于弹性容器宽度的 10% 时，将固定在 10%。

份数单位和极值也能用在行上，只是行的尺寸很少这样设定。假设有这样一个布局，首行和尾行的尺寸是固定的，而中间放置内容的行是弹性变形的，而且有下限，例如下面这样：

```
grid-template-rows:3em minmax(5em,1fr)2em;
```

这样完全可行，但是更多的时候，你会想根据行中内容的高度设定行的尺寸，而不是弹性容器高度的几分之几。下一节说明具体做法。

根据内容设定轨道的尺寸

能设置栅格轨道的尺寸为可用空间的几分之几，或者占据固定的空间的确不错，但是如果想把页面中的某些部分对齐起来，而无法确定各部分有多宽或多高呢？这时可以使用 min-content 和 max-content。

这两个关键字的字面意思够简单，但是却不那么容易准确说明其作用。其实，max-content 的意思是“占据内容所需的最大空间”。对大段文本来说（例如博客文章），这个值一般意味着尽量多占据可用空间，为内容提供最大的空间。max-content 也可以指“宽度尽量大，以防换行”。对常规的文本段落而言，这可能导致列轨道特别宽。

与之相比，min-content 的意思是“尽量少占据空间，够显示内容即可”。对文本来说，这意味着宽度会尽量收窄，只保证最长的单词（如果有图像或表单输入框的话，指宽度最大的行内元素）能在一行里完整显示。这个值会导致栅格元素中有大量断行，而且特别窄特别高。

这两个关键字的强大之处在于，它们将应用于整个栅格轨道上。例如，如果把一列的尺寸设为 max-content，那么整个列轨道的宽度都与列中最宽的内容一样。下面通过一个显示图像的栅格（12 个）说明这一点，栅格的声明如下，结果如图 13-16 所示：

```
#gallery {display: grid;
    grid-template-columns: max-content max-content max-content max-content;
    grid-template-rows: max-content max-content max-content;}
```

先看各列。可以看到，每个列轨道的宽度都与轨道中最宽的图像相等。竖放的图像恰好对齐的那一列，其宽度较窄；如果横放图像，列的宽度将变大。各行也是如此。每一行的高度都与行中最高的图像相等。如果行中的图像都矮，那么整行都矮。

这样做的好处是，内容可以是任何类型。如果为照片加上描述文字，所有行和列的尺寸都会调整，以便放下文本和图像，如图 13-17 所示。

图中的设计不完整，图像与文本不协调，而且没有限制描述文字的宽度。其实，使用 max-content 值时，列的宽度就应该这样，因为 max-content 的意思是“让列的宽度尽量大，以便放下全部内容”。

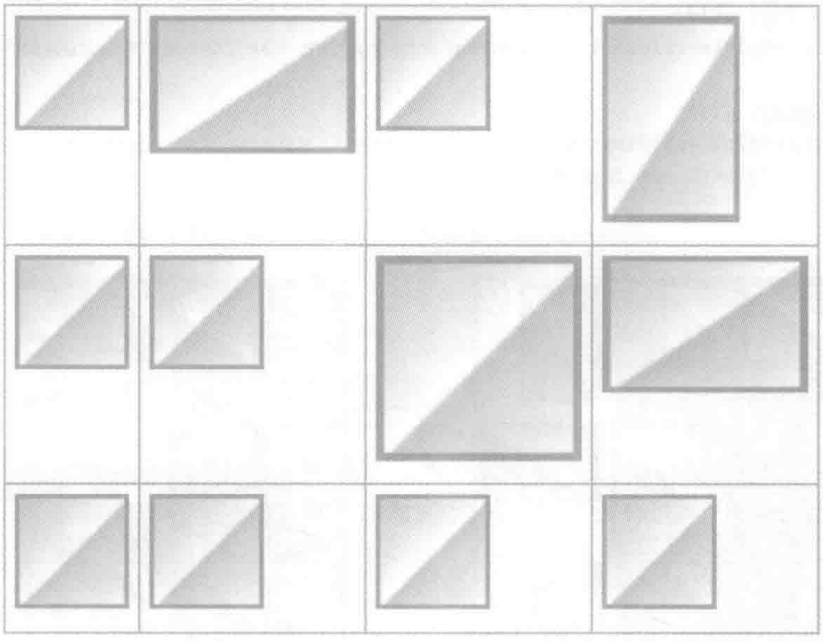


图 13-16：由内容确定栅格轨道的尺寸

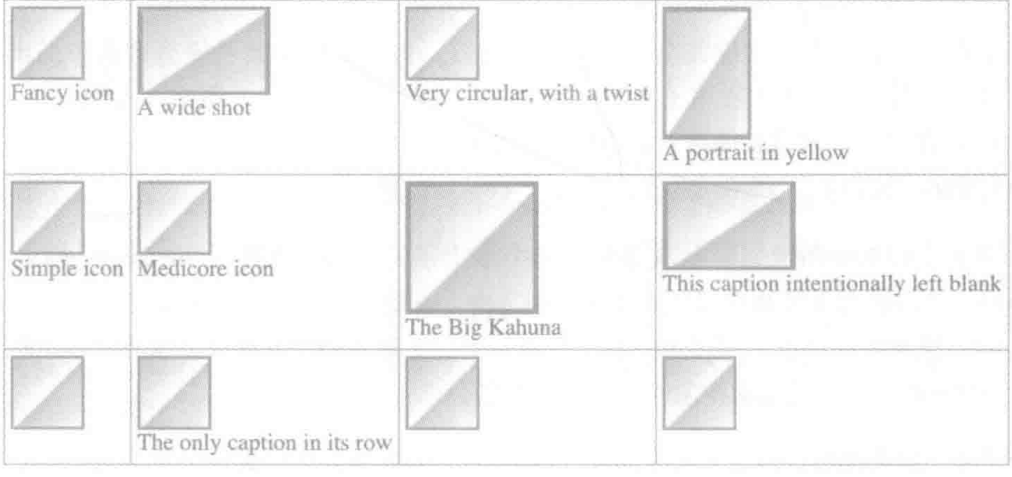


图 13-17：有多种内容类型时栅格轨道的尺寸

注意，即使栅格轨道将从栅格容器中溢出，也是这样处理。也就是说，即便我们为栅格容器设置了 `width:250px`，图像和说明文字依然像上面那样排布。这就是为什么 `max-content` 经常出现在 `minmax()` 语句中。以下述样式为例。这里的两个栅格一个用了 `minmax()`，一个没用。两种情况下的栅格容器使用橙色背景表示，结果如图 13-18 所示。

```
#g1 {display: grid;
    grid-template-columns: max-content max-content max-content max-content;
    }
#g2 {display: grid;
    grid-template-columns: minmax(0,max-content) minmax(0,max-content)
          minmax(0,max-content) minmax(0,max-content);
    }
```

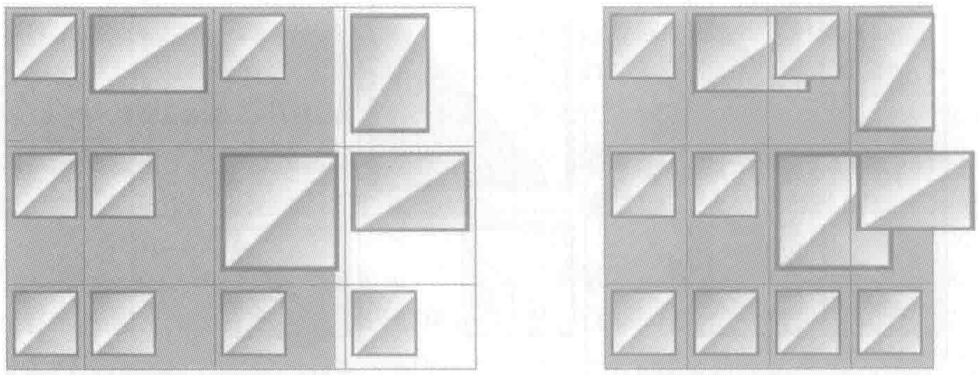


图 13-18：使用及不使用 minmax() 限定栅格轨道的尺寸

在第一个例子中，栅格元素能完全放下其中的内容，但是栅格元素从栅格容器中溢出了。在第二个例子中，由于使用了 minmax()，浏览器将把列的尺寸限定在 0 到 max-content 的范围内，因此各栅格元素将尽量留在栅格容器内。如果改成 minmax(min-content,max-content)，得到的结果与 0,max-content 稍有不同。

第二个例子中的部分图像之所以从栅格单元中溢出了，是因为根据 minmax(0,max-content)，各轨道要保证都在栅格容器的范围之内。不是每个轨道的尺寸都能达到 max-content，但是将在保证不超出弹性容器的基础上，尽量向那个值靠拢。倘若内容比轨道宽，内容将溢出，与其他轨道重叠。这是栅格的标准行为。

你可能好奇，如果列和行的尺寸都设为 min-content 会发生什么。其实，这与只把列的尺寸设为 min-content 基本上是一样的，因为栅格规范规定，浏览器应该先解析列的尺寸，然后再解析行的尺寸。

还有一个关键字可用于设定栅格轨道的尺寸，即 auto。用作最小值时，视作栅格元素的最小尺寸，即由 min-width 或 min-height 定义的值。用作最大值时，作用等同于 max-content。但是，不要以为 auto 只能在 minmax() 语句中使用。任何地方都能使用 auto，要么起最小值的作用，要么起最大值的作用，具体情况取决于周围的轨道使用的尺寸值。说实话，这个问题太过复杂，一时说不清。与 CSS 中的很多其他地方一样，

auto 的意思基本上是让浏览器自行决定。有时这么做不会出什么问题，但一般来说应该尽量避免使用 auto。



最后一句话说的不完全正确，其实使用 auto 值时，栅格元素的尺寸可由 align-content 和 justify-content 属性调整。这个话题在 13.9 节讨论。因为 auto 是唯一有此特性的轨道尺寸值，所以有时还是有理由使用的。

13.3.3 根据轨道中的内容适配

除了关键字 min-content 和 max-content 之外，还可以使用 fit-content() 函数以简练的方式表达特定类型的尺寸模式。这个函数的作用不是那么好理解，但是我们要迎难而上。

fit-content() 函数的参数为一个长度或一个百分数，例如：

```
#grid  {display: grid; grid-template-columns: 1fr fit-content(150px) 2fr;}
#grid2 {display: grid; grid-template-columns: 2fr fit-content(50%) 1fr;}
```

分析作用之前，先看一下规范给出的伪公式：

```
fit-content(argument) => min(max-content, max(min-content, argument))
```

这个公式的基本意思是，“先确定 min-content 和指定的参数哪个大，然后拿较大的那个值与 max-content 相比，找出较小的”。不怎么好理解吧？笔者前 17 次确实没读懂。

笔者觉得上述公式这样理解更好：“fit-content(argument) 等价于 minmax(min-content,max-content)，除非参数指定的值设置了更大的上限，类似于 max-width 或 max-height。”来看下面的例子：

```
#example {display: grid; grid-template-columns: fit-content(50ch);}
```

这里，参数的值为 50ch，即大约 50 个字符的宽度。因此，上述样式只创建了一列，列中内容的宽度将与指定的尺寸保持一致。

首先假设内容只有 29 个字符长，相当于 29ch（由于用的是等宽字体）。此时，max-content 的值为 29ch，那么列的宽度就是 29ch，因为 29ch 比 50ch 和 min-content 都小。

现在，假设在列中添加了一堆文本，共有 256 个字符，宽度为 256ch。因此，max-content 的计算结果为 256ch。这比参数指定的 50ch 大多了，因此列宽度的上限为 min-content 和 50ch 中较大的那个，即 50ch。

再看一个例子。下述样式的结果如图 13-19 所示。

```
#thefollowing  {
    display: grid;
    grid-template-columns:
        fit-content(50ch) fit-content(50ch) fit-content(50ch);
    font-family: monospace;}
```

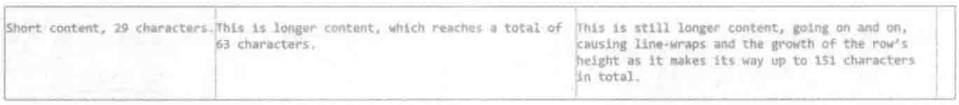


图 13-19：使用 fit-content() 设定栅格轨道的尺寸

注意，第一列的宽度比其他两列小。29ch 的内容最小就那么宽。另外两列中的内容较多，在 50ch 宽的空间里放不下，因此有换行，因为宽度的上限为 50ch。

那么，如果在第二列中添加一个图像会发生什么呢？我们添加的图像宽度为 500px，在这个例子中比 50ch 要宽。对那一列来说，首先要确定 min-content 和 50ch 哪个较大。前面说过，较大的值是 min-content，计算结果为 500px（图像的宽度）。然后，要确定 500px 和 max-content 哪个较小。如果文本全在一行里渲染，宽度将超过 500px，因此较小的是 500px。综上，现在第二列的宽度为 500 像素。结果如图 13-20 所示。

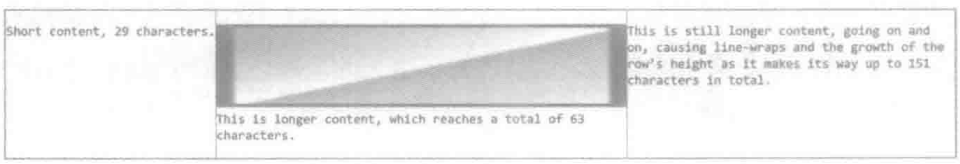


图 13-20：根据较宽的内容确定尺寸

比较一下图 13-19 和图 13-20，你会发现第二列中文本的换行位置是不同的，这是因为那一列的宽度变了。但是再看第三列中的文本，换行的位置也变了。

这是因为前两列的尺寸确定之后，第三列可用的空间不足 50ch 了。fit-content(50ch) 依然像前文所述的那样处理，但是可用空间变了。记住，50ch 参数设定的是上限，而不是定值。

这是 fit-content() 的一大优势，由此也看出 minmax() 不是特别灵活。使用 fit-content()，内容不多时，轨道将缩小为最小的内容尺寸，而在内容较多时，又能为轨道的尺寸设置一个上限。

在前面的例子中，栅格模板的值是重复的，你可能会想，如果栅格轨道的数量不止三四个该怎么办呢？难道要一个一个把轨道的尺寸写出来吗？其实无需这么做，详情参见下一节。

13.3.4 重复栅格线

如果你想创建的栅格中各栅格轨道的尺寸是一样的，你或许不想一个一个输入尺寸值。幸好有 repeat()，我们无需一个个输入了。

假设我们想每隔 5em 放置一条列栅格线，而且一共有 10 个列轨道。使用 repeat() 的写法如下：

```
#grid {display: grid;
    grid-template-columns: repeat(10, 5em);}
```

就这样，我们创建了 10 个列轨道，每个轨道的宽度为 5em，共计 50em。显然，这比输入 10 次 5em 节省时间。

轨道的尺寸可以使用任何值，可以是 min-content 和 max-content，也可以是 fr 值和 auto 等，而且不限于只提供一个值。假设我们想定义的列结构为 2em-1fr-1fr，然后重复三次。样式的写法如下，结果如图 13-21 所示：

```
#grid {display: grid;
    grid-template-columns: repeat(3, 2em 1fr 1fr);}
```

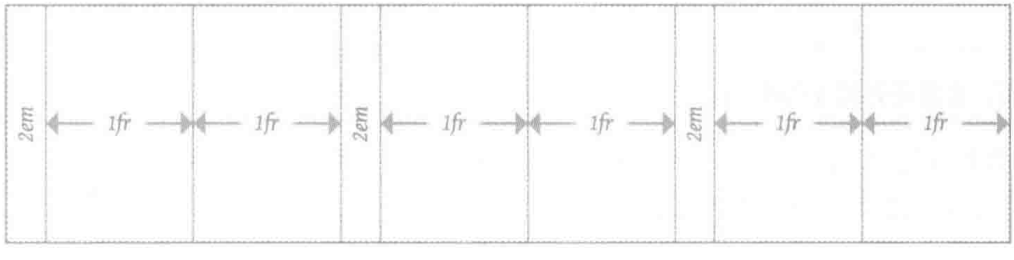


图 13-21：重复一个轨道模式

注意，最后一个列轨道的宽度为 1fr，而第一个列轨道的宽度为 2em。这是由 repeat() 的写法决定的。如果想平衡两端，在末尾加上一个宽度为 2em 的轨道，只需像下面这样修改：

```
#grid {display: grid;
    grid-template-columns: repeat(3, 2em 1fr 1fr) 2em;}
```

看到值的最后多了个 2em 了吗？此时，重复三次之后，最后将多出一列。这表明，在构建栅格时，repeat 可以与其他轨道尺寸值结合在一起使用，甚至还可以与其他重复模式协作。但是不能在重复中嵌套重复。

除此之外，repeat() 中几乎可以使用任何值。下述示例直接摘自栅格规范：

```
#grid {
    display: grid;
    grid-template-columns: repeat(4, 10px [col-start] 250px [col-end]) 10px;}
```

在这个例子中，一个 10 像素宽的轨道、一条具名栅格线、一个 250 像素宽的轨道和另一条具名栅格线重复了四次。最后，还有一个 10 像素宽的列轨道。是的，这意味着有四条列栅格线名为 col-start，还有四条栅格线名为 col-end，如图 13-22 所示。这完全是可行的，因为栅格线的名称无需是独一无二的。

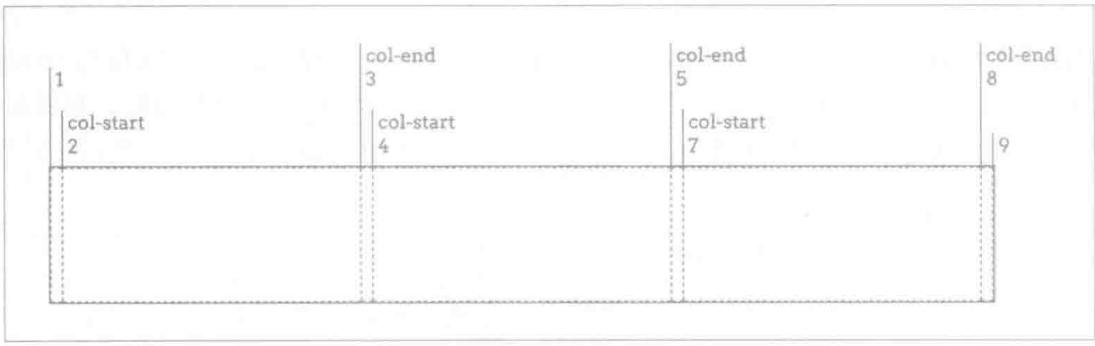


图 13-22：重复多列和具名栅格线

重复具名栅格线时要注意一点，即相邻的两条具名栅格线将合并为一条具有两个名称的栅格线。也就是说，下面两个声明是等效的：

```
grid-template-rows: repeat(3, [top] 5em [bottom]);
grid-template-rows: [top] 5em [bottom top] 5em [top bottom] 5em [bottom];
```



不要担心多条栅格线具有相同的名称会导致什么问题，因为没有任何规定禁止这么做，而且在某些情况下这样做还有一定的好处。13.4.1 节将探讨处理这种情况的一些方式。

自动填充的轨道

还有一种方法能重复简单的模式，直到填满整个栅格容器为止。这种方法没有常规的 repeat() 那么强大（至少现在如此），但有时也很有用。

例如，假设我们想让前面的行模式一直重复，只要不撑破栅格容器即可：

```
grid-template-rows: repeat(auto-fill, [top] 5em [bottom]);
```

此时，每隔 5em 放置一条行栅格线，直到没有空间为止。因此，对 11em 高的栅格容器来说，等效于下述声明：

```
grid-template-rows: [top] 5em [bottom top] 5em [bottom];
```

如果栅格容器的高度超过 15em，但是小于 20em，那么等效于下述声明：

```
grid-template-rows: [top] 5em [bottom top] 5em [top bottom] 5em [bottom];
```

在三个不同高度的栅格容器中，上述自动填充行的示例得到的结果如图 13-23 所示。

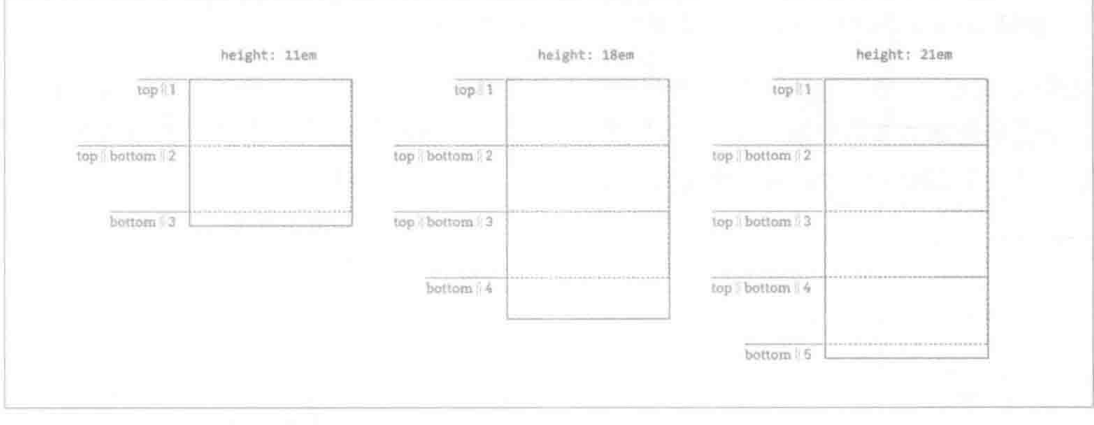


图 13-23：在三个高度下自动填充行

自动重复的局限是，只能有一个可选的栅格线名称、一个尺寸固定的轨道和另一个可选的栅格线名称。因此，[top] 5em [bottom] 是这种模式下值最多的情况。其中的具名栅格线可以去除，只留下 5em，或者只去除其中一个名称。不能重复多个尺寸固定的轨道，也不能重复多个尺寸弹性变化的轨道（这么规定的确是合理的，试想，浏览器要重复几次 1fr 才能填满栅格容器呢？一次即可）。



你可能想自动重复多个尺寸不同的轨道，以此定义放在内容栏周围的“栏距”。通常，没必要这么做，因为栅格有轨道栏距的概念（也有定义栏距的属性）。我们将在 13.8 节讨论这个话题。

此外，在一个轨道模板中只能有一个自动重复的模式。因此，下述写法是不准许的：

```
grid-template-columns: repeat(auto-fill, 4em) repeat(auto-fill, 100px);
```

然而，固定数量的重复模式可以与自动填充的轨道结合在一起使用。例如，可以像下面这样先放三个较宽的列，然后再使用较窄的轨道填充栅格容器中余下的空间（假设空间有余）：

```
grid-template-columns: repeat(3, 20em) repeat(auto-fill, 2em);
```

当然，反过来也可以：

```
grid-template-columns: repeat(auto-fill, 2em) repeat(3, 20em);
```

之所以可以这样做，是因为栅格布局算法是先为固定尺寸的轨道分配空间的，余下的空间才使用自动重复的轨道填充。上述示例的结果是，自动填充一个或多个 2em 宽的列，随后是三个 20em 宽的轨道。前面两个示例的结果如图 13-24 所示。

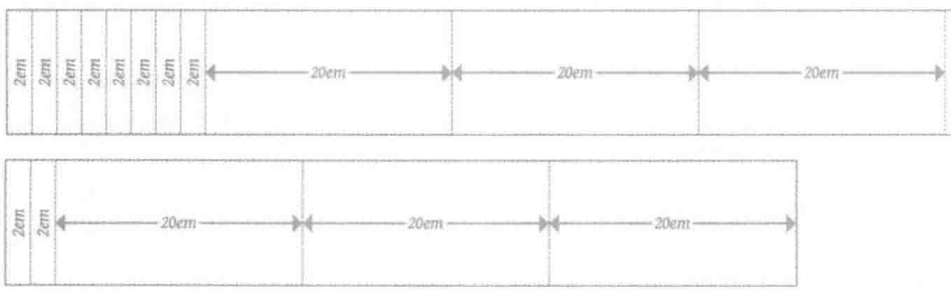


图 13-24：在固定的列旁边自动填充列

auto-fill 至少会重复一次轨道模板，即使出于什么原因在栅格容器中放不下也是如此。而且，只要空间足够，能重复多少次就重复多少次，即便某些轨道中没有内容。举个例子，假设自动填充五列，但是只有前三列中有栅格元素。此时，另外两个仍在那儿，在布局中占据一定的空间。

但是，如果使用 auto-fit，没有栅格元素的轨道将被剔除。除此之外，auto-fit 的行为与 auto-fill 一样。对下述声明来说：

```
grid-template-columns: repeat(auto-fit, 20em);
```

如果栅格容器中放得下五个列轨道（即容器的宽度要大于 100em），但是有两个轨道中没有栅格元素，那么这两个空的栅格轨道将被剔除，留下三个有栅格元素的列轨道。剔除轨道后留下的空间根据 align-content 和 justify-content 的值处理（参见 13.9 节）。图 13-25 对 auto-fill 和 auto-fit 做了简单的比较，彩色框中的数字表示所在列的序号。

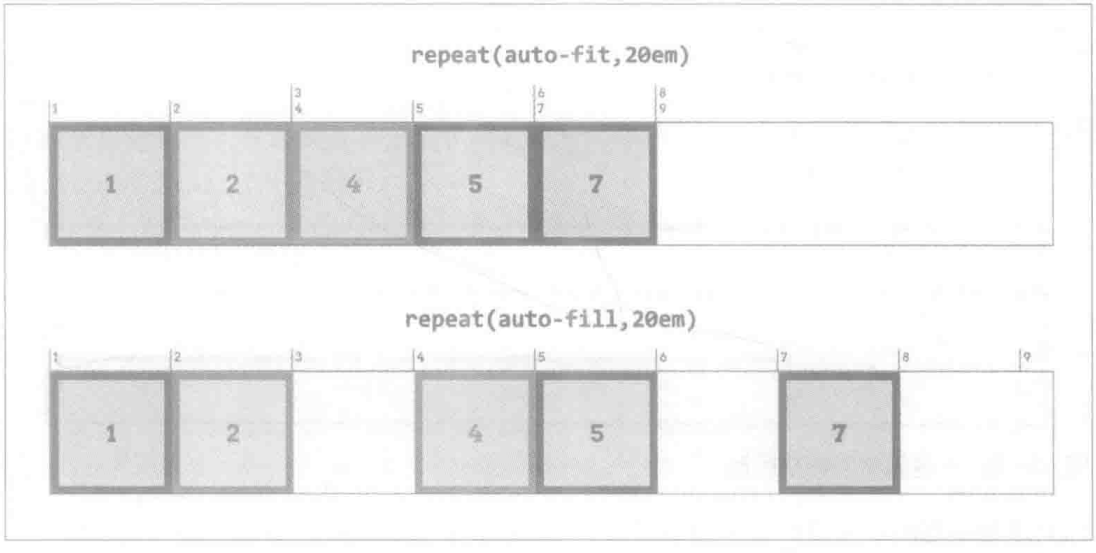


图 13-25：auto-fill 与 auto-fit 比较

13.3.5 栅格区域

有时，你可能只想把栅格画出来，也许是为了好玩，也许是为了给代码一个参照。这个需求或多或少可以使用 grid-template-areas 属性实现。

grid-template-areas

取值	none | *<string>*
初始值	none
适用于	栅格容器
计算值	声明的值
继承性	否
动画性	否

这个属性可说的太多了，不如先来看一个例子吧。下述规则得到的结果如图 13-26 所示：

```
#grid {display: grid;
    grid-template-areas:
        "h h h h"
        "l c c r"
        "l f f f";}
```

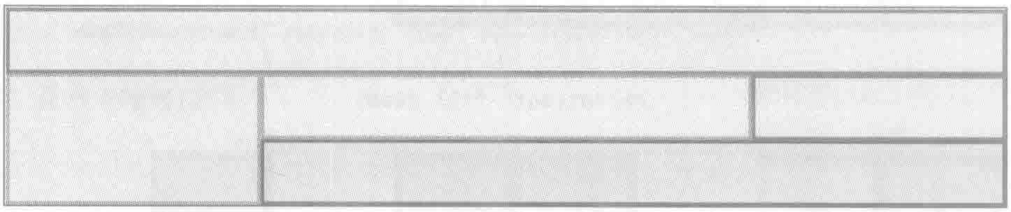


图 13-26：一些简单的栅格区域

你的理解是对的，字符串值中的字母用于定义栅格区域的形状。真的！而且并不限于只能使用单个字母。例如，前例可以改成下面这样：

```
#grid {display: grid;
    grid-template-areas:
        "header    header    header    header"
        "leftside  content   content   rightside"
        "leftside  footer    footer    footer";}
```

得到的栅格布局与图 13-26 中的一样，不过各区域的名称变了（例如 f 变成了 footer）。

在定义区域的模板中，空白会折叠，因此可以利用空白对齐 grid-template-areas 值中的各列（前例就这样做了）。用于对齐的空白可以是空格，也可以是制表符，你可以挑一个最能惹怒同事的。当然，你也可以只用一个空格把各标识符隔开，而不管名称有没有对齐。字符串之间甚至可以不换行，下述写法与对齐的版本效果是一样的：

```
grid-template-areas: "h h h h" "l c c r" "l f f f";
```

但是，不能把多个字符串合并到一起，这样意思就变了。每个字符串（放在一对双引号中）定义栅格中的一行。因此，前一个例子与之前的例子定义的栅格都有三行。如果像下面这样合并为一个字符串：

```
grid-template-areas:
    "h h h h
     l c c r
     l f f f";
```

得到的栅格便只有 1 行 12 列，h 区域跨 4 列，f 区域跨 3 列。字符串内部的换行，除了把标识符隔开以外没有任何作用。

仔细观察这些值，你会发现每个标识符表示一个栅格单元。仍以本节开头的第一个示例为例，请看图 13-27 中的结果：

```
#grid {display: grid;
    grid-template-areas:
        "h h h h"
        "l c c r"
        "l f f f";}
```

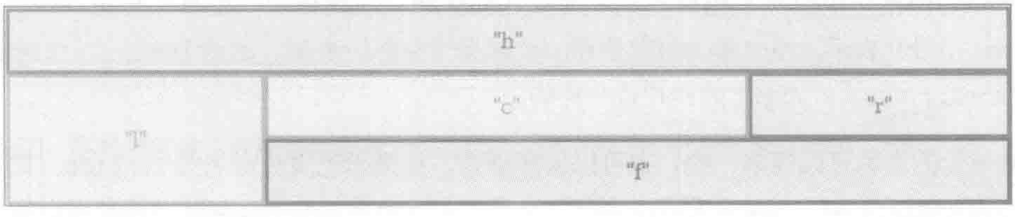


图 13-27：带标识符的栅格单元

得到的布局与前面完全一样，只是这一次我们把 grid-template-areas 值中的标识符放到各栅格单元中了。知道各单元的标识符后，浏览器把名称相同的相邻单元合并为一个区域，不过得到的形状必须是矩形。如果区域形状太复杂，整个模板都将失效。因此，下述样式得不到任何栅格区域：

```
#grid {display: grid;
    grid-template-areas:
        "h h h h"
        "l c c r"
        "l l f f";}
```

发现没有，l区域的外形是“L”形的？就这样小小改动之后，整个grid-template-areas值都变得无效了。栅格布局以后的版本也许会支持非矩形形状，但是现在只能定义矩形。

如果只想把部分栅格单元定义为栅格区域的一部分，其他的单元不标注名称，可以使用一个或多个.字符占位。假如你只想定义页头、页脚和部分侧边栏区域，余下的栅格单元不命名。那么，可以像下面这样声明，得到的结果如图13-28所示：

```
#grid {display: grid;
    grid-template-areas:
        "header  header  header  header"
        "left    ...     ...     right"
        "footer  footer  footer  footer";}
```

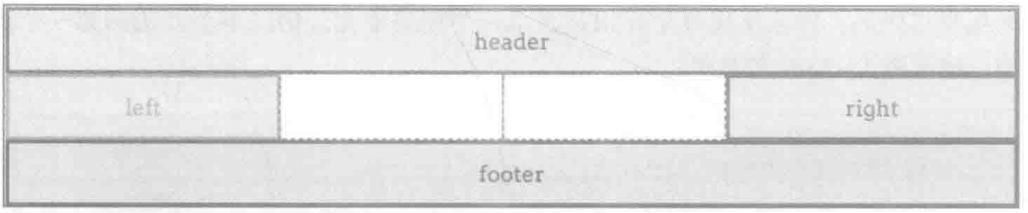


图13-28：栅格中部分栅格单元没有名称

上图中的栅格，中间两个栅格单元不属于任何区域，在模板中以空单元标记（null cell token，.号）表示。空名单元可以使用一个或多个空标记表示，因此left . . right和left .... right的作用是一样的。

栅格单元的名称可以简单一些，也可以发挥创意。如果想把页头命名为ronaldo、把页脚命名为podiatrist，放心去做吧。甚至，大于U+0080码位的Unicode字符也可以使用，因此ConHugeCo©®™和åwësømë是完全有效的区域标识符……当然表情符号也是可以的。

定义好栅格区域之后，接下来要使用前面介绍的grid-template-columns和grid-template-rows定义栅格轨道的尺寸。下面把二者添加到前例中，得到的结果如图13-29：

```
#grid {display: grid;
    grid-template-areas:
        "header  header  header  header"
        "left    ...     ...     right"
```

```
        "footer  footer  footer  footer";
    grid-template-columns: 1fr 20em 20em 1fr;
    grid-template-rows: 40px 10em 3em;}
```

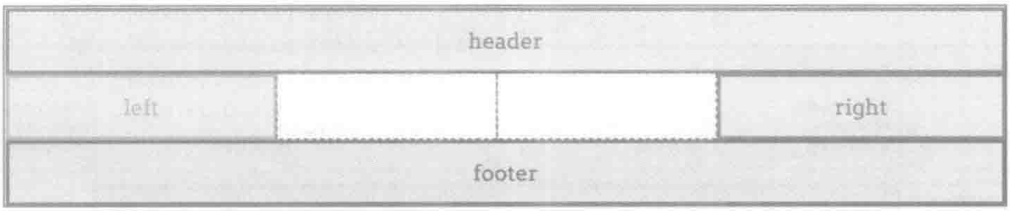


图 13-29：具名区域和设定了尺寸的轨道

现在，具名栅格区域创建的列和行有轨道尺寸了。如果提供的轨道尺寸数量比区域轨道多，多出的轨道将放在具名区域之后。因此，下述 CSS 得到的结果如图 13-30 所示：

```
#grid {display: grid;
    grid-template-areas:
        "header  header  header  header"
        "left    ...     ...     right"
        "footer  footer  footer  footer";
    grid-template-columns: 1fr 20em 20em 1fr 1fr;
    grid-template-rows: 40px 10em 3em 20px;}
```

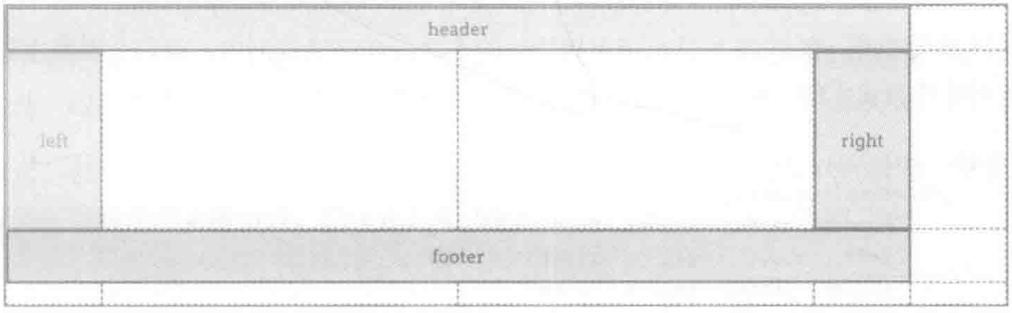


图 13-30：在具名区域之后再添加轨道

那么，为区域命名了还怎么为栅格线命名呢？其实，栅格线已经有名称了：命名栅格区域就自动为首尾两条栅格线命名了。对 header 区域来说，第一条列栅格线和第一条行栅格线的名称都是 header-start，而第二条列栅格线和第二条行栅格线的名称是 header-end。对 footer 区域来说，相关的栅格线将自动命名为 footer-start 和 footer-end。

栅格线穿过整个栅格区域，因此有很多名称是出现在同一位置的。下述模板得到的栅格线名称如图 13-31 所示：

```
grid-template-areas:
    "header    header    header    header"
    "left      ...       ...       right"
    "footer    footer    footer    footer";
```

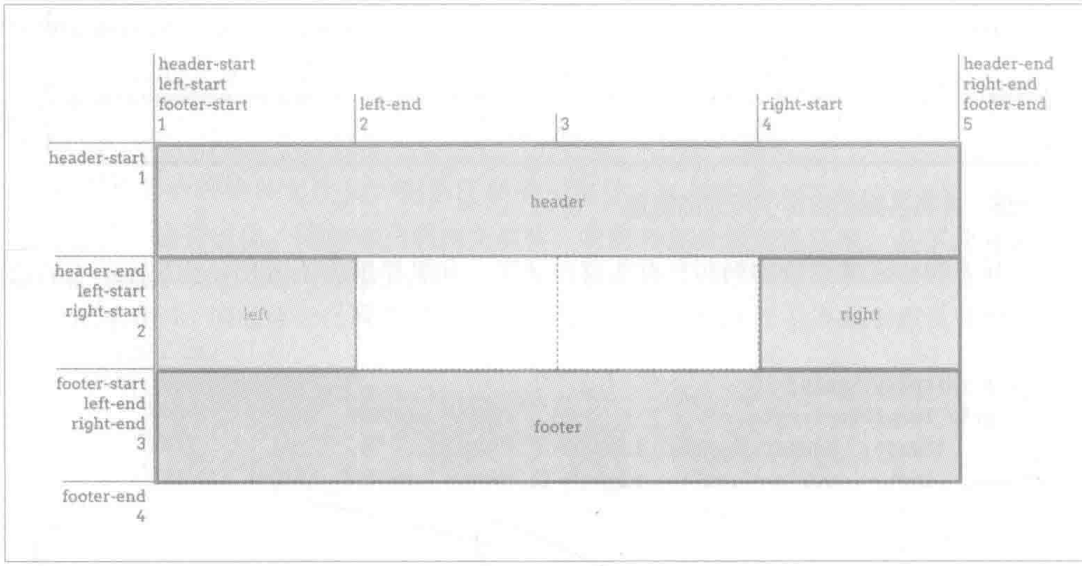


图 13-31：标出隐含的栅格线名称

下面增加点难度，在 CSS 中显式指定栅格线的名称。在下述规则中，我们为栅格中的第一条列栅格线设定的名称是 begin，为第二条行栅格线设定的名称是 content：

```
#grid {display: grid;
    grid-template-areas:
        "header  header  header  header"
        "left    ...     ...     right"
        "footer  footer  footer  footer";
    grid-template-columns: [begin] 1fr 20em 20em 1fr 1fr;
    grid-template-rows: 40px [content] 1fr 3em 20px;}
```

再次注意，这些栅格线的名称将累加到具名区域隐式创建的栅格线名称上。这是比较有趣的行为，栅格线的名称从不取代现有的栅格线名称，而是累积在一起。

更有趣的是，隐式命名机制还可以反向操作。假设完全不使用 grid-template-areas，而是像下面这样为栅格线命名，得到的结果如图 13-32 所示：

```
    grid-template-columns:
```

```
        [header-start footer-start] 1fr
        [content-start] 1fr [content-end] 1fr
        [header-end footer-end];
    grid-template-rows:
        [header-start] 3em
        [header-end content-start] 1fr
        [content-end footer-start] 3em
        [footer-end];
```

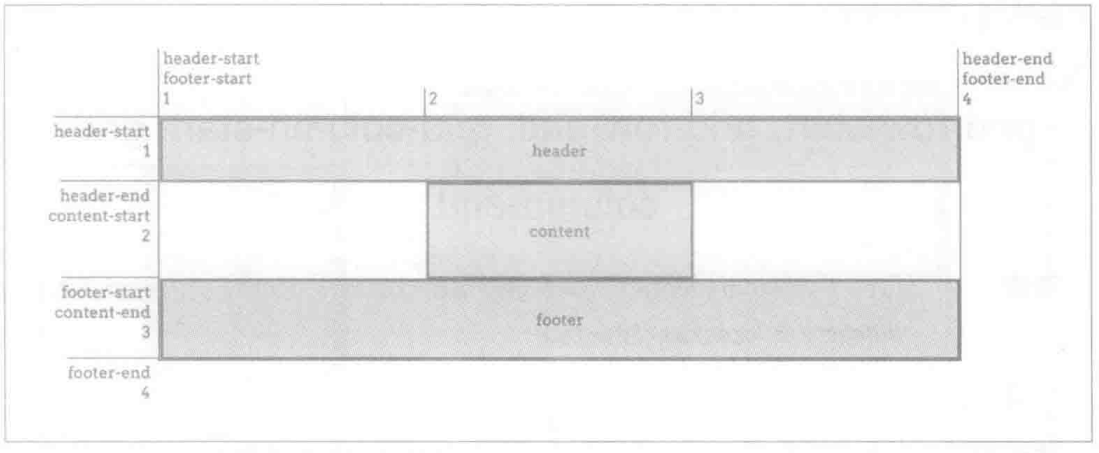


图 13-32：标出隐式命名的栅格区域

由于栅格线名称的格式是 name-start/name-end，这就隐式为栅格区域定义了名称。老实说，这没有前面那种方法简便，但是多一种方法总是好的，万一用得到呢。

注意，创建栅格区域不需要全部四条栅格线，但是若想创建所需形状的具名栅格区域，可能必须指定四条栅格线。以下述声明为例：

```
grid-template-columns: 1fr [content-start] 1fr [content-end] 1fr;
grid-template-rows: 3em 1fr 3em;
```

这样也能创建名为 content 的栅格区域，只是该区域所在的行将放在显式定义的行之后。奇怪的是，在显式定义的行后面、content 区域所在的行前面会多出一个空行。但这是符合规定的行为。因此，如果想通过命名栅格线创建具名区域，但不提供全部四条栅格线，那么具名区域将偏安一隅，而不在栅格的整体结构中。

因此，还是前面说的，建议一直显式命名栅格区域，隐式生成 -start 和 -end 形式的栅格线名称，不要反过来做。

13.4 在栅格中附加元素

我们讲了这么多，还未讨论定义好栅格后如何附加栅格元素，真是难以置信。

13.4.1 使用列线和行线

附加栅格元素的方式有很多，可以引用栅格线，也可以引用栅格区域。先讲把元素附加到栅格线上的四个属性，这比较简单。

grid-row-start, grid-row-end, grid-column-start, grid-column-end

取值	auto | <*custom-ident*> | [<*integer*> && <*custom-ident*>?] | [span && [<*integer*> || <custom-ident>]]
初始值	auto
适用于	栅格元素和绝对定位的元素（前提是容纳块为栅格容器）
计算值	声明的值
继承性	否
动画性	否

这几个属性的意思是，“我想把元素的边界附加到某条栅格线上。”栅格布局丰富多样，通过实例说明可能更好理解。请看下面的样式，想一想结果如何（见图 13-33）：

```
.grid {display: grid; width: 50em;
    grid-template-rows: repeat(5, 5em);
    grid-template-columns: repeat(10, 5em);}
.one {
    grid-row-start: 2; grid-row-end: 4;
    grid-column-start: 2; grid-column-end: 4;}
.two {
    grid-row-start: 1; grid-row-end: 3;
    grid-column-start: 5; grid-column-end: 10;}
.three {
    grid-row-start: 4;
    grid-column-start: 6;}
```

这里，我们通过栅格线的编号指明元素应该放在栅格中的什么位置。列的编号从左到右依次增加，行的编号从上到下依次增加。注意，如果省略结束栅格线，例如 .three 那样，那么结束栅格线使用序列中的下一条栅格线。

因此，前例中的 .three 规则与下述样式是完全等效的：

```
.three {
    grid-row-start: 4; grid-row-end: 5;
    grid-column-start: 6; grid-column-end: 7;}
```

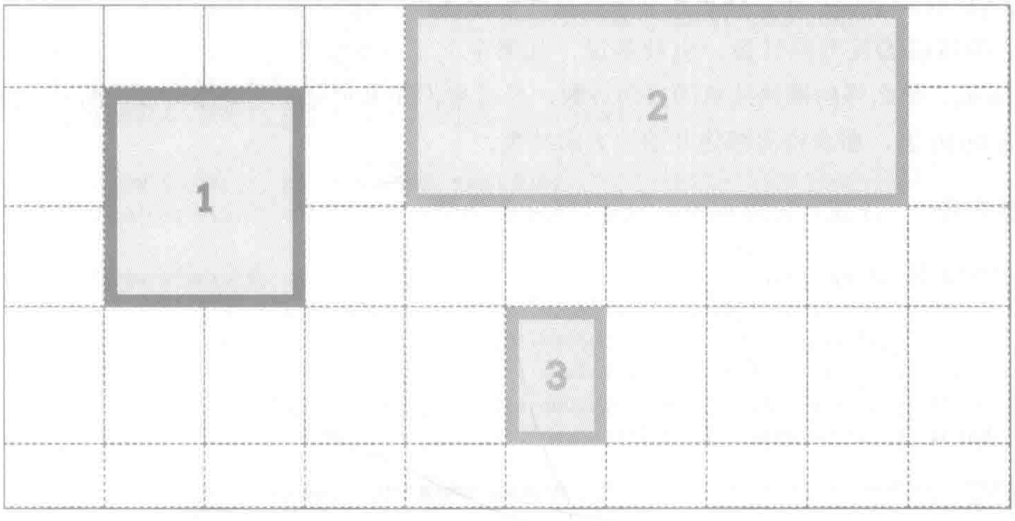


图 13-33：把元素附加到栅格线上

此外，还可以使用另一种方式表达相同的目的：把结束值改为 span1，或者只使用 span，如下所示。

```
.three {
    grid-row-start: 4; grid-row-end: span 1;
    grid-column-start: 6; grid-column-end: span;}
```

如果 span 后面有数字，意思是“跨指定数目的栅格轨道”。因此，前面的示例可以改为下面这样，得到的结果也是完全一样的：

```
#grid {display: grid;
    grid-template-rows: repeat(5, 5em);
    grid-template-columns: repeat(10, 5em);}
.one {
```

```
    grid-row-start: 2; grid-row-end: span 2;
    grid-column-start: 2; grid-column-end: span 2;}
.two {
    grid-row-start: 1; grid-row-end: span 2;
    grid-column-start: 5; grid-column-end: span 5;}
.three {
    grid-row-start: 4; grid-row-end: span 1;
    grid-column-start: 6; grid-column-end: span;}
```

如果 span 后面没有数字，默认为 1。span 后面的数字不能为零或负数，只能使用正整数。

span 的特殊之处在于，结束和开始栅格线都能使用。span 的具体行为是，向确定了编号的栅格线的反方向计数。也就是说，如果定义了开始栅格线，而把结束栅格线设为 span 值，那么将向栅格结束的方向计数。反过来，如果定义了结束栅格线，而开始栅格线是 span 值，那么将向栅格开始的方向计数。

这意味着，下述规则将得到如图 13-34 所示的结果：

```
#grid {display: grid;
    grid-rows: repeat(4, 2em); grid-columns: repeat(5, 5em);}
.box01 {grid-row-start: 1; grid-column-start: 3; grid-column-end: span 2;}
.box02 {grid-row-start: 2; grid-column-start: span 2; grid-column-end: 3;}
.box03 {grid-row-start: 3; grid-column-start: 1; grid-column-end: span 5;}
.box04 {grid-row-start: 4; grid-column-start: span 1; grid-column-end: 5;}
```

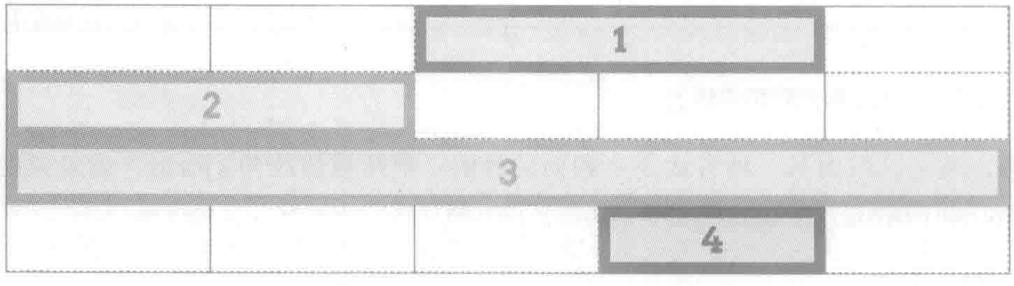


图 13-34：跨多条栅格线

与 span 后的数字不同，指定栅格线的编号时并不限于只能使用正整数。负数将从显式定义的栅格线从后往前数。因此，若想把一个元素放在栅格右下角那个栅格单元中，而不管栅格中有多少列多少行，可以这样声明：

```
grid-column-start: -1;
grid-row-start: -1;
```

注意，隐式栅格轨道（这个概念稍后讨论）不能这样做，只有通过 grid-template-* 属性（例如 grid-template-rows）显式定义的栅格线才能使用负数引用。

当然，引用栅格线也不只限于使用编号。如果栅格线有名称，还可以使用名称引用栅格线（或者二者混用）。如果多条栅格线使用同一个名称，还要加上编号，指明想引用的是哪一条。因此，如果想引用第四条名为 mast-slice 的栅格线，要使用 mast-slice 4。请看下面的样式（结果如图 13-35），了解具体该怎么做：

```
#grid {display: grid;
    grid-template-rows: repeat(5, [R] 4em);
    grid-template-columns: 2em repeat(5, [col-A] 5em [col-B] 5em) 2em;}
.one {
    grid-row-start: R 2; grid-row-end: 5;
    grid-column-start: col-B; grid-column-end: span 2;}
.two {
    grid-row-start: R; grid-row-end: span R 2;
    grid-column-start: col-A 3; grid-column-end: span 2 col-A;}
.three {
    grid-row-start: 9;
    grid-column-start: col-A -2;}
```

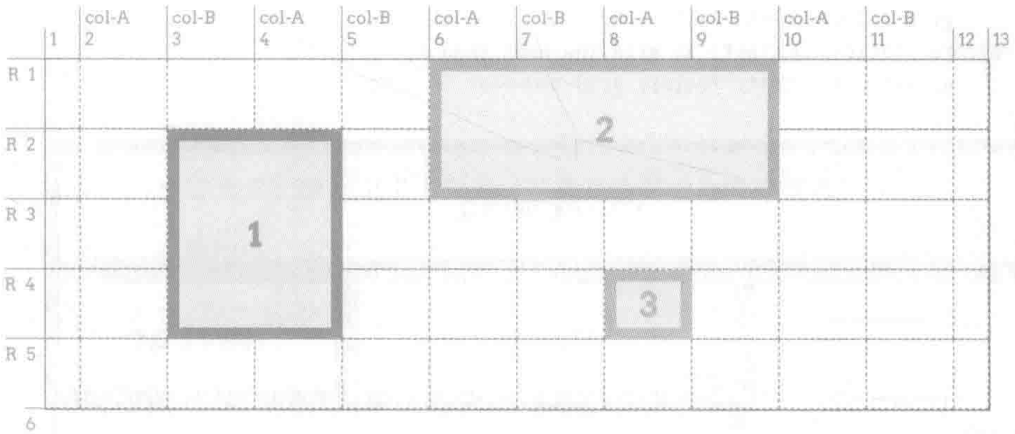


图 13-35：把元素附加到具名栅格线上

注意，使用名称引用时，span 后面的值也变了：span 2 col-A 的意思是，栅格元素从起点（第三条 col-A）开始，跨过一个 col-A，在后面一个 col-A 处结束。因此，这个栅格元素其实横跨四个列轨道，因为 col-A 每隔一条列栅格线出现一次。

同样，负数从序列的尾端反向计数，因此 col-A-2 的意思是倒数第二条名为 col-A 的栅格线。由于 .three 没有声明结束线的值，所以都被设为 span 1。也就是说，下述样式与前例中的 .three 样式完全等效：

```
.three {
    grid-row-start: 9; grid-row-end: span 1;
    grid-column-start: col-A-2; grid-row-end: span 1;}
```

除此之外还有一种引用栅格线名称的方式，即通过栅格区域隐式创建的栅格线名称。请看下述样式，得到的结果如图 13-36 所示：

```
grid-template-areas:
    "header     header    header    header"
    "leftside   content   content   rightside"
    "leftside   footer    footer    footer";
#masthead {grid-row-start: header;
    grid-column-start: header; grid-row-end: header;}
#sidebar {grid-row-start: 2; grid-row-end: 4;
    grid-column-start: leftside / span 1;}
#main {grid-row-start: content; grid-row-end: content;
    grid-column-start: content;}
#navbar {grid-row-start: rightside; grd-row-end: 3;
    grid-column-start: rightside;}
#footer {grid-row-start: 3; grid-row-end: span 1;
    grid-column-start: footer; grid-row-end: footer;}
```

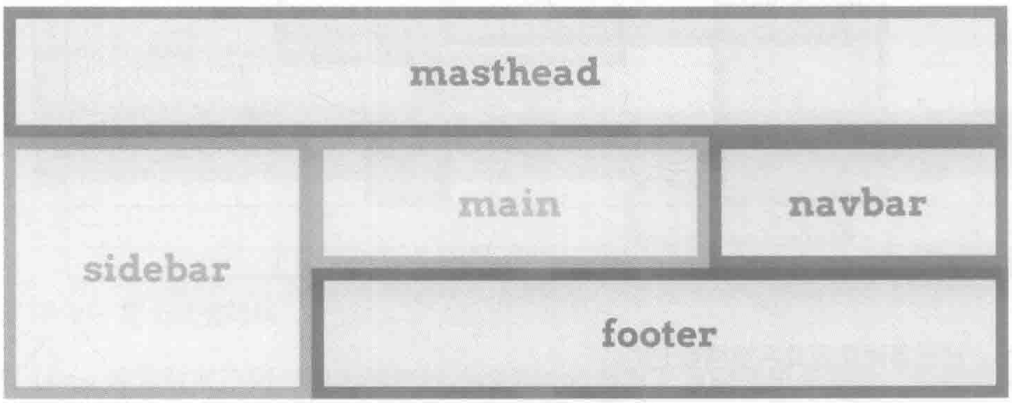


图 13-36：把元素附加到具名栅格线上的另一种方式

浏览器遇到你提供的自定义标识符（即你定义的名称）时会在名称后面加上 -start 或 -end，然后寻找相应名称的栅格线。具体加上哪个值，取决于你指定的是开始线还是结束线。因此，下面两行声明是等效的：

```
grid-column-start: header; grid-column-end: header;
grid-column-start: header-start; grid-column-end: header-end;
```

等效的原因前面说过，因为使用 grid-template-areas 显式创建栅格区域时会使用 -start 和 -end 的形式隐式命名区域四周的栅格线。

最后一个可用的值是 auto。这个值有点让人摸不透。根据栅格布局规范，如果把开始或结束栅格线属性的值设为 auto，意思是“自动确定位置，自动确定跨度，或者默认跨度为一”。实际应用中，具体选择哪条栅格线由栅格流（grid flow，这个概念还未介绍，不过快了）确定。对开始线来说，auto 通常指下一条可用的列线或行线。对结束线来说，auto 通常指跨一个栅格单元后的栅格线。注意前两句话中的“通常”，因为自动机制是没有定数的。

13.4.2 行和列的简写属性

有两个简写属性能简化把元素附加到栅格线上的过程。

grid-row, grid-column

取值	*<grid-line>* [/ *<grid-line>*]?
初始值	auto
适用于	栅格元素和绝对定位的元素（前提是容纳块为栅格容器）
计算值	声明的值
继承性	否
动画性	否

这两个属性的主要优势是，能简化排布栅格元素时声明开始和结束栅格线的过程。例如：

```
#grid {display: grid;
    grid-template-rows: repeat(10, [R] 1.5em);
    grid-template-columns: 2em repeat(5, [col-A] 5em [col-B] 5em) 2em;}
.one {
    grid-row: R 3 / 7;
    grid-column: col-B / span 2;}
```

```
.two {
    grid-row: R / span R 2;
    grid-column: col-A 3 / span 2 col-A;}
.three {
    grid-row: 9;
    grid-column: col-A -2;}
```

老实说，这比一个一个声明开始和结束线简洁多了。除了变简洁之外，这两个属性的行为大都与前面所说的一样。在以斜线分隔的两部分中，前一部分定义的是开始栅格线，后一部分定义的是结束栅格线。

如果值中没有斜线，那么定义的是开始栅格线，结束栅格线取决于开始栅格线的值。如果开始栅格线是用名称引用的，那么结束栅格线也使用那个名称引用。因此，下面两个声明是等效的：

```
grid-column: col-B;
grid-column: col-B / col-B;
```

即，栅格元素将从指定名称的栅格线开始一直延伸到下一条同名栅格线，不管中间有多少栅格单元。

如果只提供一个数字，那么第二个数字（即结束线的编号）被设为 auto。因此，下面两对声明是等效的：

```
grid-row: 2;
grid-row: 2 / auto;

grid-column: header;
grid-column: header / header;
```

使用 grid-row 和 grid-column 指定栅格线的名称有个鲜为人知的行为，事关隐式命名的栅格线。你应该记得，定义具名栅格区域时将创建名称为 -start 和 -end 形式的栅格线。也就是，对名为 footer 的栅格区域来说，顶边和左边两条栅格线的名称是 footer-start，底边和右边两条栅格线的名称是 footer-end。

此时通过区域的名称引用栅格线，也能把元素放在正确的位置上。因此，下述样式将得到如图 13-37 所示的结果：

```
#grid {display: grid;
    grid-template-areas:
        "header header"
```

```
        "sidebar content"
        "footer footer";
    grid-template-rows: auto 1fr auto;
    grid-template-columns: 25% 75%;}
#header {grid-row: header / header; grid-column: header;}
#footer {grid-row: footer; grid-column: footer-start / footer-end;}
```

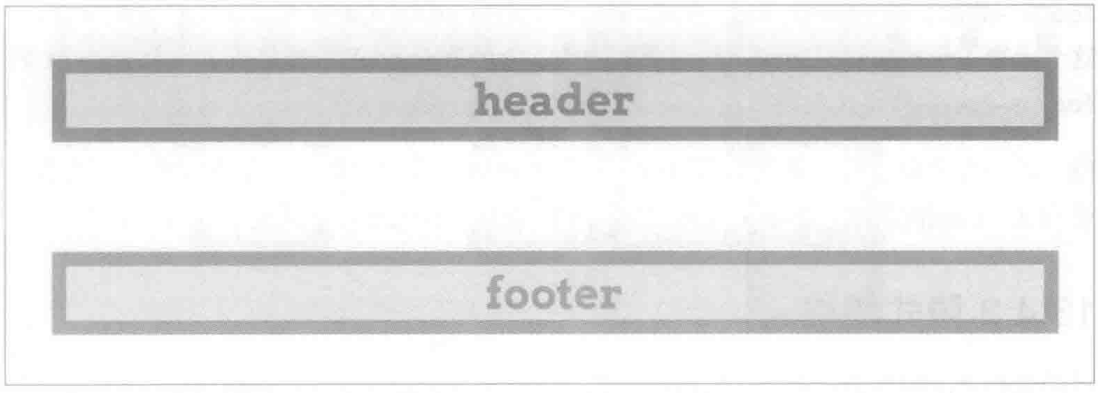


图 13-37：通过栅格区域隐式创建的名称把元素附加到栅格线上

隐式创建的栅格线名称始终可以直接引用，但是如果只引用栅格区域的名称，也不会出错。如果引用的栅格线名称不属于任何栅格区域，那就回落到前面讨论的行为。即，相当于使用 `line-name 1`。因此，下面两个声明是等效的：

```
grid-column: jane / doe;
grid-column: jane 1 / doe 1;
```

正因为如此，使用相同的名称命名栅格线和栅格区域是有一定风险的。请看下面的声明：

```
    grid-template-areas:
        "header header"
        "sidebar content"
        "footer footer"
        "legal legal";
    grid-template-rows: auto 1fr [footer] auto [footer];
    grid-template-columns: 25% 75%;
```

这里，显式把“footer”行上面和“legal”行下面两条栅格线命名为 `footer`……那么，问题也随之而来了。假如我们声明了下述样式：

```
#footer {grid-column: footer; grid-row: footer;}
```

列线没什么问题。footer扩展为footer/footer。浏览器查找同名的栅格区域，如果找到，把footer/footer变成footer-start/footer-end。因此，#footer元素附加在这两条隐式创建的栅格线上。

对grid-row来说，一开始都是一样的。footer变成footer/footer，然后再变成footer-start / footer-end。但是，这意味着#footer元素将与“footer”行一样高。这个元素不会延伸到“legal”行下面那条显示命名为footer的栅格线，因为footer到footer-end的转换优先级更高（由于栅格线和栅格区域同名了）。

综上，一般来说最好不要使用相同的名称命名栅格区域和栅格线。有些情况下可能不会受到这一问题的影响，但是最好始终把线和区域的名称区分开，以免导致命名解析冲突。

13.4.3 隐式栅格

目前我们关注的都是显式定义的栅格，我们讨论了如何通过grid-template-columns等属性定义行和列轨道，以及如何把栅格元素附加到轨道中的单元里。

但是，如果栅格元素（或其一部分）超出了显式定义的栅格呢？以下述栅格为例：

```
#grid {display: grid;
    grid-template-rows: 2em 2em;
    grid-template-columns: repeat(6, 4em);}
```

这个栅格十分简单，有两行六列。但是，如果我们把一个栅格元素放在第一列中，而且从第一条行线一直延伸到第四条行线，那么情况如何呢？

```
.box01 {grid-column: 1; grid-row: 1 / 4;}
```

三条栅格线只能界定两行，但是我们让浏览器从行线1延伸到行线4。

遇到这种情况，浏览器会再创建一条行线。这条栅格线，以及由此而生的一个行轨道都是隐式栅格的一部分。下面几个栅格元素都会创建隐式栅格线（和轨道），在栅格中的排布方式如图13-38所示：

```
.box01 {grid-column: 1; grid-row: 1 / 4;}
.box02 {grid-column: 2; grid-row: 3 / span 2;}
.box03 {grid-column: 3; grid-row: span 2 / 3;}
.box04 {grid-column: 4; grid-row: span 2 / 5;}
.box05 {grid-column: 5; grid-row: span 4 / 5;}
.box06 {grid-column: 6; grid-row: -1 / span 3;}
.box07 {grid-column: 7; grid-row: span 3 / -1;}
```

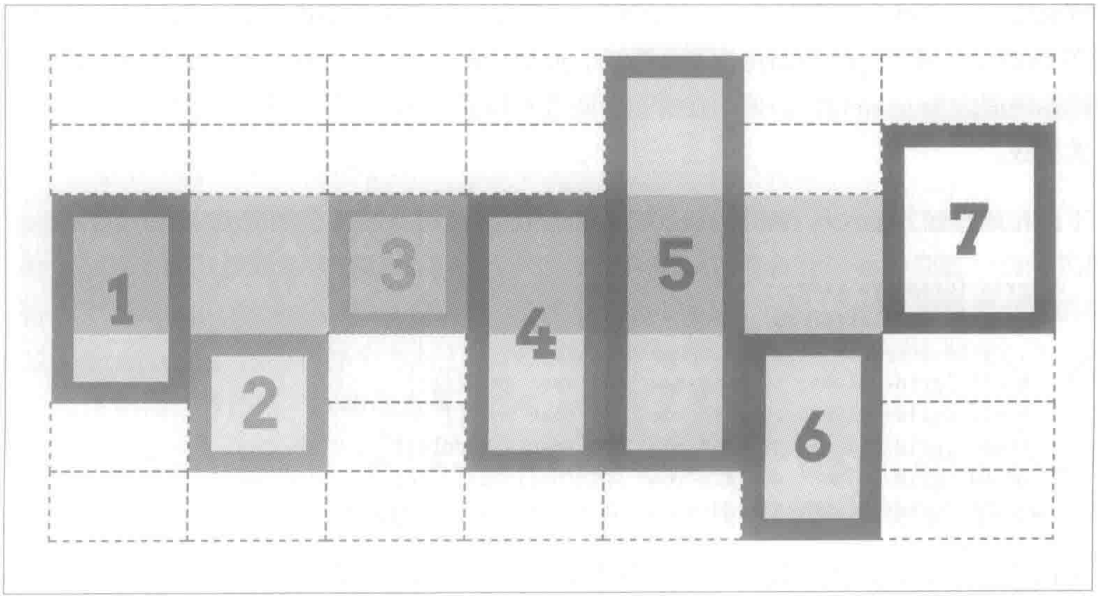


图 13-38：创建隐式栅格线和轨道

图中的内容很多，下面详细说明。首先，各数字框背后的浅灰色框表示显式栅格，虚线表示隐式栅格。

再看那些数字框。前面说过，box1 在显式栅格的后面添加一条栅格线。box2 从显式栅格的最后一条栅格线开始，向下延伸两条线，因此它还会添加一条隐式栅格线。box3 在最后一条显式栅格线（线 3）结束，向上跨两条线，因此从第一条显式栅格线开始。

box4 就有趣了，它在第五条行线（即第二条隐式栅格线）结束，向上跨三条线，因此开始栅格线与 box3 一样。这是因为，跨度是从显式栅格开始计数的，然后向隐式栅格延伸（box2 就是这样），但是不能从隐式栅格开始计数。

因此，box4 在第五条行线结束，但是它从栅格线 3 开始反向延伸两条线（span 2）到达行线 1。类似地，box5 在行线 5 处结束，而且反向延伸四条线，因此从行线 2 开始。记住，跨度必定从显式栅格内部开始计数，但是不一定在显式栅格中结束。

box6 从最后一条显式行线（线 3）开始，一直跨到第六条行线，因此又会增加一条隐式行线。这个数字框的目的是说明引用栅格线的负数是相对显式栅格而言向结束端的反向计数的，而不是反向引用位于显式栅格开始端前面的隐式栅格线。

如果想让元素从显式栅格开始端前面的隐式栅格线开始，请看 box7：把结束线放在显式栅格的某个位置，然后反向跨越显式栅格的起端。你可能注意到了，box7 占着一个隐式

列轨道。一开始，栅格有六列，因此有七条列线，所以显式栅格在第七条列线处结束。在 box7 上声明的 grid-column: 7 等效于 grid-column: 7 / span 1（因为缺少的结束线始终假定为 span 1），为了在第七个隐式列轨道中放置栅格元素，必然要创建一条隐式列线。

下面在此基础上加入具名栅格线。请看下面的样式，得到的结果如图 13-39 所示：

```
#grid {display: grid;
    grid-template-rows: [begin] 2em [middle] 2em [end];
    grid-template-columns: repeat(5, 5em);}
.box01 {grid-column: 1; grid-row: 2 / span end 2;}
.box02 {grid-column: 2; grid-row: 2 / span final;}
.box03 {grid-column: 3; grid-row: 1 / span 3 middle;}
.box04 {grid-column: 4; grid-row: span begin 2 / end;}
.box05 {grid-column: 5; grid-row: span 2 middle / begin;}
```

可以看出，每条隐式创建的栅格线都有名称。以 box2 为例。我们指定的结束线名为 final，但是显式栅格中没有这条线，一直搜索到显式栅格的尾端也没有找到，因此新建一条栅格线，将其命名为 final（在图 13-39 中，隐式创建的栅格线名称以浅色的斜体字表示）。

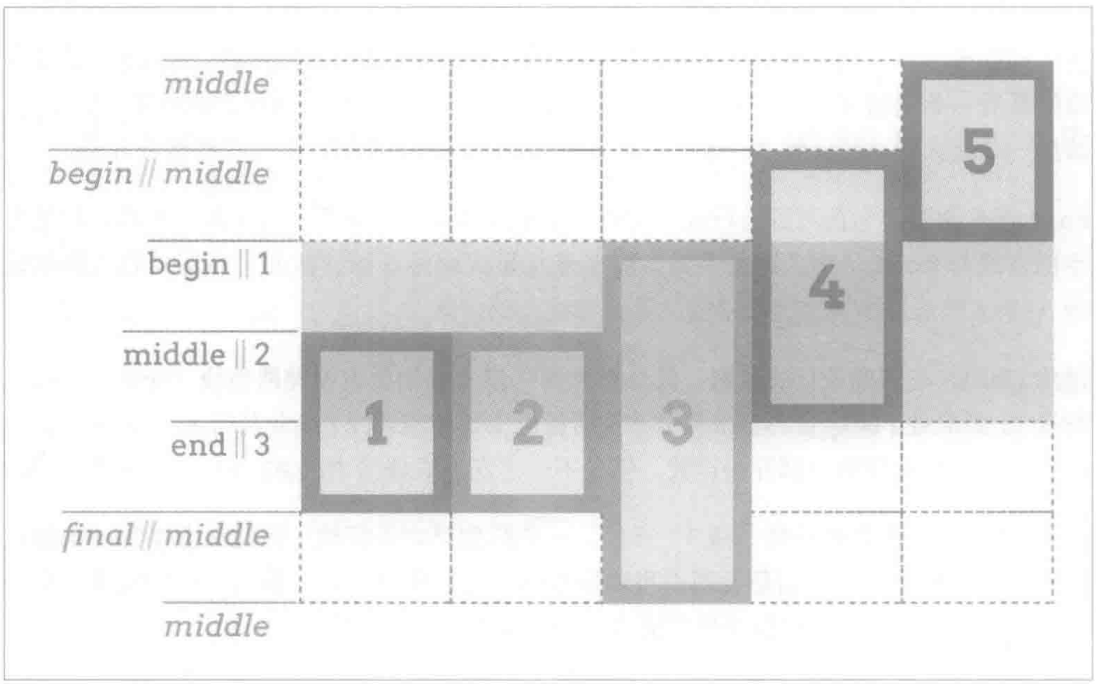


图 13-39：具名隐式栅格线和轨道

类似地，box3 从第一条显式行线开始，而且要跨三条名为 middle 的线。向前搜索，在显式栅格中找到一条，然后继续寻找下两条。在显式栅格中找不到，把第一条隐式栅格线的名称设为 middle。第二条隐式栅格线以同样的方式处理。因此，box3 在显式栅格后面的第二条隐式栅格线处结束。

box4 和 box5 也是这样处理的，只不过是向终点的反方向搜索的。box4 在 end 线（线 3）结束，反向跨到找到的第二条 begin 线。因此，在第一条栅格线前面会创建一条隐式栅格线，名为 begin。box5 从 begin 线（显式命名为 begin 的线）开始反向跨到找到的第二条 middle 线。因为在那个方向上找不到名为 middle 的栅格线，所以会创建两条名为 middle 的隐式栅格线，在离起点最远的那一条线处结束。

完全掌握原理后你会发现，隐式栅格是十分优雅的回落机制。但是，一般而言最好始终使用显式栅格，确保显式栅格能放得下全部元素。如果缺少一行，不要让元素从栅格的边界溢出，而应该调整栅格模板的值。

13.4.4 错误处理

有些异常情况需要处理，以防出现奇形怪状的栅格。

首先，如果不小心把开始线放在结束线后面，结果如何？比如下面这样：

```
grid-row-start: 5;
grid-row-end: 2;
```

结果可能与我们一开始想表达的一样：对调两个值。因此，得到的将是：

```
grid-row-start: 2;
grid-row-end: 5;
```

其次，如果开始线和结束线都声明为跨度，该怎么处理呢？例如：

```
grid-column-start: span;
grid-column-end: span 3;
```

这种情况下，结束线的值将被丢弃，替换为 auto。因此，得到的将是：

```
grid-column-start: span;  /* 'span' 等同于 'span 1' */
grid-column-end: auto;
```

此时，栅格元素的结束边界根据当前栅格流（稍后讨论这个话题）自动放置，而开始边界则放在前一条栅格线上。

最后，如果只用具名跨度指明栅格元素的位置，该如何处理？比如说：

```
grid-row-start: span footer;
grid-row-end: auto;
```

规范不允许这么做，因此 span footer 将替换为 span 1。

13.4.5 使用区域

通过行线和列线附加元素是不错，但是能不能只用一个属性引用栅格区域呢？当然可以——grid-area。

grid-area

取值	*<grid-line>* [/ *<grid-line>*]{0,3}
初始值	参见各单独属性
适用于	栅格元素和绝对定位的元素（前提是容纳块为栅格容器）
计算值	声明的值
继承性	否
动画性	否

先讲 grid-area 较为简单的用法：把元素指定给定义好的栅格区域。没什么难理解的吧？把前文用过的 grid-template-areas 声明拿过来，加上 grid-area 声明和一些标记，得到的结果如图 13-40 所示：

```
#grid {display: grid;
    grid-template-areas:
        "header    header    header    header"
        "leftside  content   content   rightside"
        "leftside  footer    footer    footer";}
#masthead {grid-area: header;}
#sidebar {grid-area: leftside;}
#main {grid-area: content;}
#navbar {grid-area: rightside;}
#footer {grid-area: footer;}
```

```
<div id="grid">
    <div id="masthead">...</div>
    <div id="main">...</div>
    <div id="navbar">...</div>
    <div id="sidebar">...</div>
    <div id="footer">...</div>
</div>
```

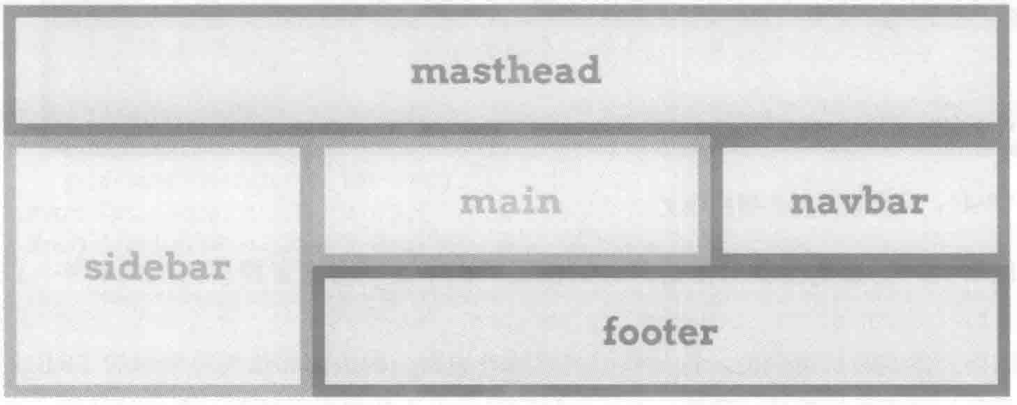


图 13-40：把元素分到各个栅格区域中

就是这么简单：先设置一些具名栅格区域，定义布局，然后通过 grid-area 把栅格元素放入指定的区域。简单而强大。

你可能注意到了，上述 CSS 中没有指定列轨道和行轨道的尺寸。这完全是为了突出本节要讲的知识。实际使用中，规则可能会写成下面这样：

```
grid-template-areas:
    "header   header  header  header"
    "leftside content content rightside"
    "leftside footer  footer  footer";
grid-template-rows: 200px 1fr 3em;
grid-template-columns: 20em 1fr 1fr 10em;
```

此外，还可以使用 grid-area 引用栅格线。事前提醒：基于多方面的原因，所用的句法一开始可能让人摸不着头脑。

下面的例子通过栅格模板定义了几条栅格线，然后又使用 grid-area 规则引用了那些栅格线，结果如图 13-41 所示：

```
#grid {display: grid;
    grid-template-rows:
        [r1-start] 1fr [r1-end r2-start] 2fr [r2-end];
```

```
    grid-template-columns:
        [col-start] 1fr [col-end main-start] 1fr [main-end];}
.box01 {grid-area: r1 / main / r1 / main;}
.box02 {grid-area: r2-start / col-start / r2-end / main-end;}
.box03 {grid-area: 1 / 1 / 2 / 2;}
```

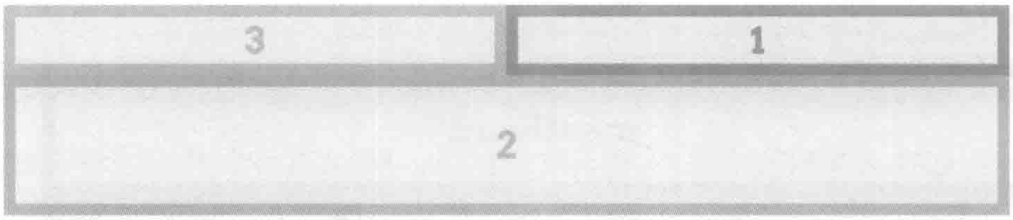


图 13-41：把元素附加到栅格线上

可以看到，元素是按我们指定的方式放置的。不过，请注意栅格线值的顺序：row-start、column-start、row-end、column-end。在脑中构想一下便会发现，这些值是围绕栅格元素逆时针排列的。这与我们习惯的外边距、内边距和边框等采用的 TRBL（上右下左）顺序正好相反。此外，对列和行的引用也不是放在一起的，而是分开的。

当然，这是故意为之的。至于原因嘛，笔者也不知道。

如果提供的值少于四个，缺少的值根据提供的值确定。如果只有三个值，而且 grid-column-start 的值是栅格线的名称，那么缺少的 grid-column-end 值与 grid-column-start 的值相同；如果开始线的值是一个数字，那么结束线的值被设为 auto。如果只提供两个值，也采用这种方式处理，只不过当 grid-row-start 的值是栅格线的名称时，缺少的 grid-row-end 值与之相同，否则设为 auto。

据此，你或许能猜到只提供一个值时是怎么处理的了：如果是栅格线的名称，四个值都用那个名称；如果是数字，余下的都设为 auto。

指定单个栅格区域名称就能让栅格元素填入那个区域正是基于这种一出四的复制模式。下面两个声明是等效的：

```
grid-area: footer;
grid-area: footer / footer / footer / footer;
```

现在，回想一下前一节讨论的有关 grid-column 和 grid-row 的行为：如果栅格线的名称与栅格区域的名称一致，名称将变成相应的 -start 或 -end 形式。因此，前例最终将变成：

```
grid-area: footer-start / footer-start / footer-end / footer-end;
```

这就是通过一个栅格区域名称就能把元素放到相应栅格区域中的原因。

13.4.6 栅格元素重叠

目前，我们在创建栅格布局时都极力避免出现重叠。然而，与定位一样，栅格元素是完全有可能重叠的（牢记这一点）。下面举个简单的例子，结果如图 13-42 所示：

```
#grid {display: grid;
    grid-template-rows: 50% 50%;
    grid-template-columns: 50% 50%;}
.box01 {grid-area: 1 / 1 / 2 / 3;}
.box02 {grid-area: 1 / 2 / 3 / 2;}
```

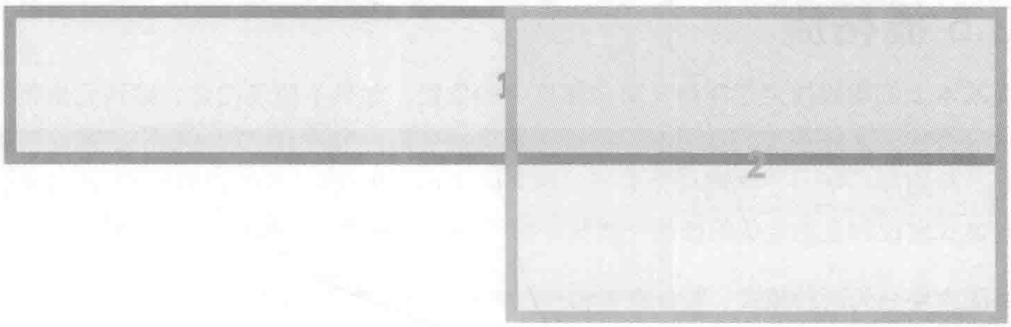


图 13-42：栅格元素出现重叠

根据我们指定的栅格线编号，示例中的两个栅格元素在右上角那个栅格单元中重叠。哪个元素在上面取决于后文将讨论的分层行为，现在只需知道重叠时会分层即可。

不只使用数字引用栅格线时可能出现重叠。在下面的例子中，侧边栏和页脚将重叠，如图 13-43 所示（假设页脚的标记出现在侧边栏的标记后面，如果没有其他样式的影响，页脚将显示在侧边栏上面）。

```
#grid {display: grid;
    grid-template-areas:
        "header header"
        "sidebar content"
        "footer footer";}
#header {grid-area: header;}
#sidebar {grid-area: sidebar / sidebar / footer-end / sidebar;}
#footer {grid-area: footer;}
```

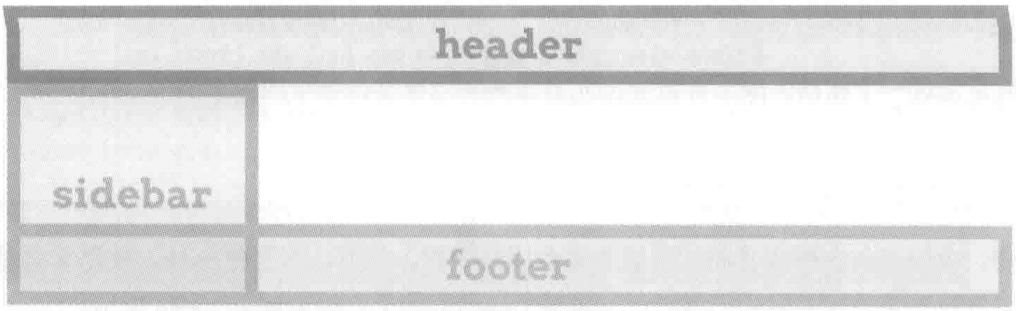


图 13-43：侧边栏和页脚出现重叠

这里提及这个话题，部分原因是让你知道栅格元素有可能出现重叠，此外也是为了转换到下一个话题，即栅格流。这个特性把栅格布局与定位区分开，有时能避免重叠。

13.5 栅格流

前面基本上都明确指定了栅格元素在栅格中的位置。如果不明确指定，栅格元素将自动放入栅格中。在栅格流的作用下，栅格元素将放在第一个适合它的区域中。最简单的情况是，按顺序一个一个把栅格元素放入栅格轨道中。但实际情况可能比这复杂得多，尤其是显式定位和自动定位的栅格元素共存时——后者将围绕前者放置。

栅格流主要分为两种模式，即行优先和列优先，不过二者都可以通过密集流（denseflow）增强。栅格流通过 grid-auto-flow 属性设置。

grid-auto-flow

取值	[row | column] || dense
初始值	row
适用于	栅格容器
计算值	声明的值
继承性	否
动画性	否

以下述标记为例说明这些值的作用：

```
<ol id="grid">
<li>1</li>
<li>2</li>
<li>3</li>
<li>4</li>
<li>5</li>
</ol>
```

我们在这段标记上应用下述样式：

```
#grid {display: grid; width: 45em; height: 8em;
    grid-auto-flow: row;}
#grid li {grid-row: auto; grid-column: auto;}
```

假设栅格中每隔 15em 有一条列线、每隔 4em 有一条行线，那么将得到如图 13-44 所示的结果。

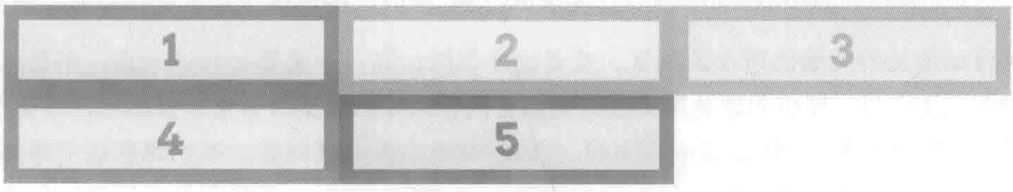


图 13-44：行主导的栅格流

这看起来再正常不过了，浮动或行内排布也能得到一样的结果。正因为如此，`row` 才是默认值。下面把 `grid-auto-flow` 的值换成 `column` 试试，结果如图 13-45 所示：

```
#grid {display: grid; width: 45em; height: 8em;
    grid-auto-flow: column;}
#grid li {grid-row: auto; grid-column: auto;}
```

可以看到，声明 `grid-auto-flow:row` 时，先填满一行再转到下一行，而声明 `grid-auto-flow:column` 时，则先填满一列。

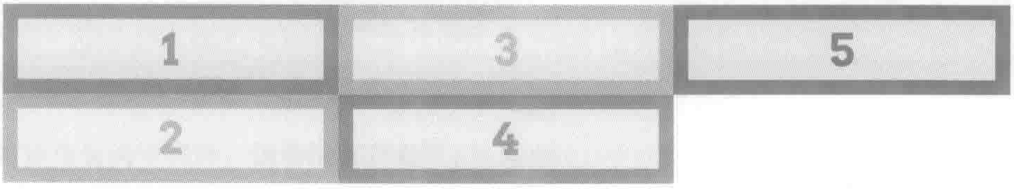


图 13-45：列主导的栅格流

特别注意，这里没有明确设定列表项目的尺寸。默认情况下，元素的尺寸将自动调整，以便附加到所定义的栅格线上。为元素显式设定尺寸便可以覆盖这一行为。例如，把列表元素的宽度设为 7em、高度设为 1.5em，将得到如图 13-46 所示的结果：

```
#grid {display: grid; width: 45em; height: 8em;
    grid-auto-flow: column;}
#grid li {grid-row: auto; grid-column: auto;
    width: 7em; height: 1.5em;}
```

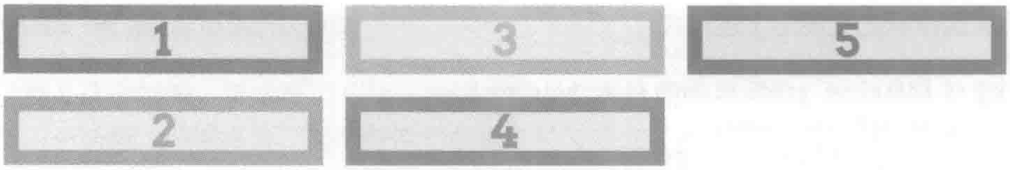


图 13-46：显式设定栅格元素的尺寸

与前一幅插图比较，你会发现相同编号的栅格元素是从同一位置开始的，但是不在同一位置结束。这表明，栅格流放置的其实是栅格区域，然后再把栅格元素附加到栅格区域中。

如果自动流动的元素比所在的列宽，或者比所在的行高，一定要记住这一点。不要以为这种情况不常见，栅格元素是图像或其他具有内在尺寸的元素时非常容易出现这种情况。假设有个栅格每隔 50 像素有一条列线、每隔 50 像素有一条行线，我们想在这个栅格中放一些尺寸各异的图像。以行优先或列优先的流动方式放置一系列图像后得到的结果如图 13-47 所示。

```
#grid {display: grid;
    grid-template-rows: repeat(3, 50px);
    grid-template-columns: repeat(4, 50px);
    grid-auto-rows: 50px;
    grid-auto-columns: 50px;
}
img {grid-row: auto; grid-column: auto;}
```

有些图像出现了重叠，注意到了吗？这是因为各个图像是附加到栅格流中的下一条栅格线上的，没有考虑其他栅格元素的存在。图像较大时，我们没有让图像跨越多个栅格轨道，因此出现了重叠。

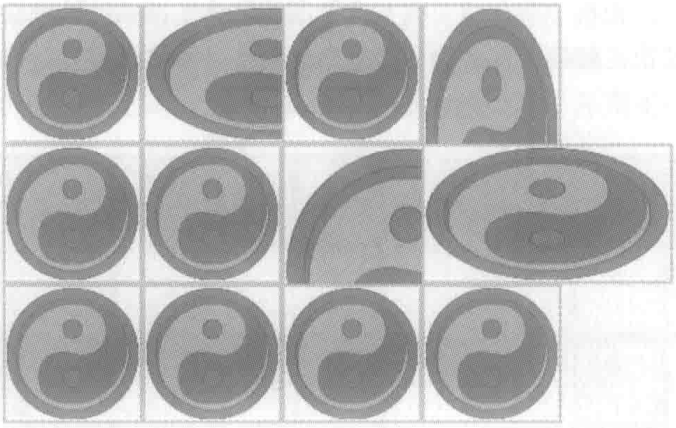


图 13-47：在栅格中自动流动的图像

如果想为不同尺寸的图像设定不同的跨度，可以通过类或其他方式添加样式。比如为占多个栅格轨道的图像加上 `tall` 或 `wide` 类（或二者兼具），然后添加下述 CSS，得到的结果如图 13-48 所示：

```
img.wide {grid-column: auto / span 2;}
img.tall {grid-row: auto / span 2;}
```

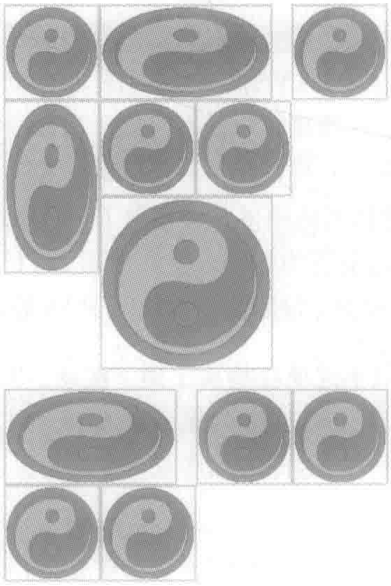


图 13-48：让图像跨越多个栅格轨道

这样图像会向下展开，但是没有重叠了。

然而，注意到前一个栅格中有间隙了吗？这是因为跨多条栅格线的栅格元素没有为栅格流中的其他元素留出足够的空间。为了清楚地表明两种栅格流模式对这种情况的处理方式，下面使用编号框演示（图 13-49）。

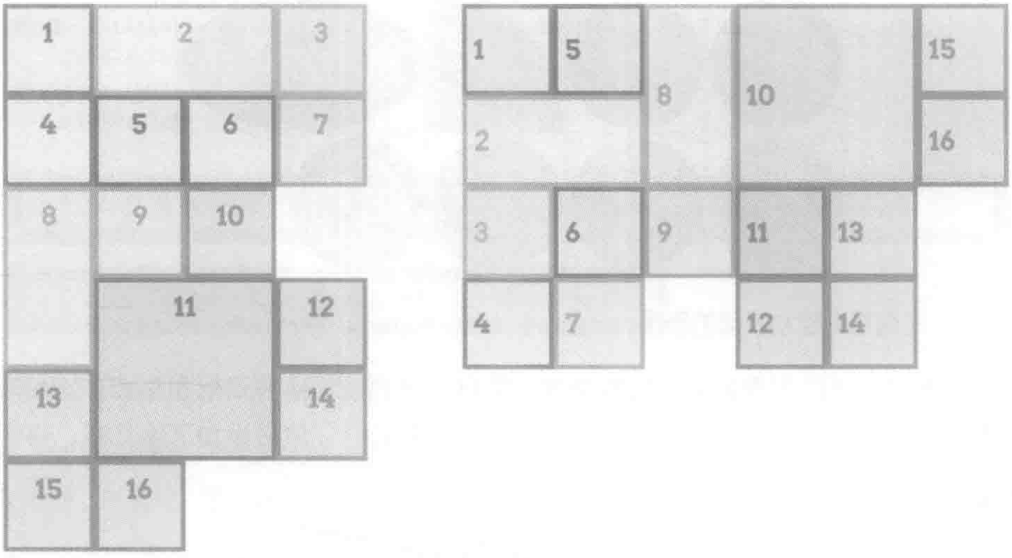


图 13-49：图解两种栅格流模式

在第一个栅格中，请按照数字顺序数一遍。在这个栅格流中，栅格元素的排布方式好似向左浮动一样。但也不尽然，因为 13 号栅格元素在 11 号栅格元素左边。如果是浮动，绝不可能出现这种情况，但是在栅格流中就有可能。行流（如果可以这样称呼的话）的工作方式是，在每一行中从左向右排，如果有足够的空间放下一个栅格元素，就把栅格元素放在那儿，如果栅格单元被其他栅格元素占据了，就跳过那个栅格单元。我们看到，10 号栅格元素后面是空的，这是因为那部分空间放不下 11 号元素了。13 号元素之所以在 11 号元素左边，是因为换到那一行后空间能放得下 13 号元素。

列流的基本机制也是如此，只不过现在是从上到下流动。9 号栅格元素下面是空的，因为那部分空间不足以放下 10 号元素。10 号元素流入下一列，占据四个栅格单元。10 号后面的元素都只占一个栅格单元，因此按顺序在列中排布。



在从左至右、从上到下书写的语言中，栅格流从左至右、从上到下。在从右至左书写的语言中，例如阿拉伯语和希伯来语，行主导的栅格流从右至左，而不是从左至右。

如果想让栅格元素尽量靠紧，而不管顺序会受到什么影响，也不是不可以，只需在 grid-auto-flow 的值里加上关键字 dense。图 13-50 中的两个图分别展示 grid-auto-flow: rowdense 和 grid-auto-flow: densecolumn 的结果。

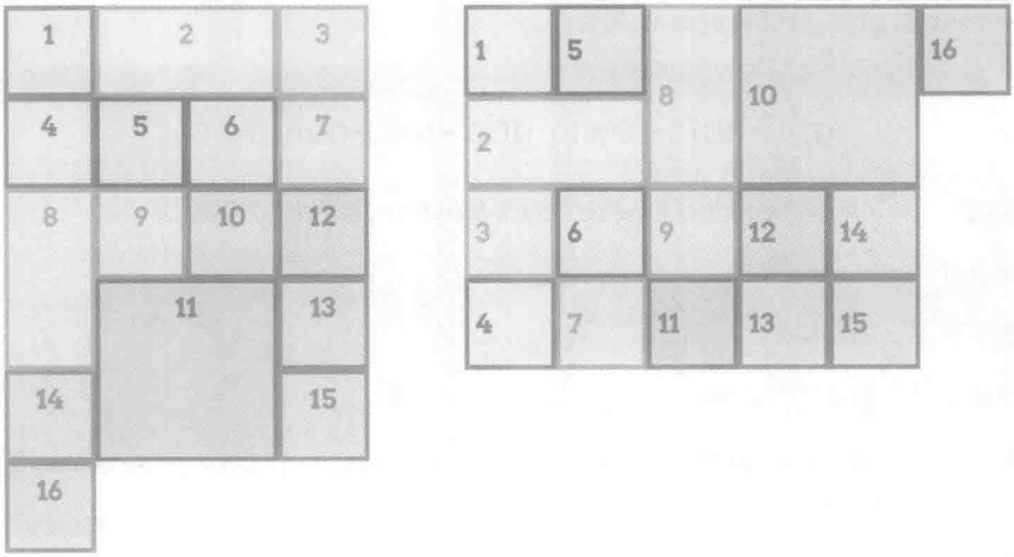


图 13-50：图解密集栅格流模式

在第一个栅格中，12 号元素在 11 号元素上边的行里，这是因为那里正好有个栅格单元可以放下 12 号元素。在第二个栅格中，11 号元素在 10 号元素左边的列中，原因相同。

其实，设为密集栅格流时，浏览器会沿流动方向（row 或 column）从起点（在从左至右书写的语言中是左上角）开始扫描整个栅格，找到能放下栅格元素的位置就把元素放在那儿。密集栅格流能让相册更紧凑，当然前提是无需按特定的顺序显示照片。

栅格流讲完了，我要坦白一件事：为了让前面几幅插图中的栅格元素看起来没大的差异，我用了一些还没介绍的 CSS。如果没有那些 CSS，悬在栅格边缘处的元素看起来将与其他元素有些差别——在行主导的流中矮很多，在列主导的流中窄很多。这其中的原因，以及我使用的 CSS，在下一节说明。

13.6 自动增加栅格线

目前我们见到的栅格元素基本上都是放在显式定义的栅格中。不过，前一节也遇到了超出显式定义栅格边界的栅格元素。栅格元素超出边界了怎么办呢？答案是根据布局要求

增加所需的行或列（见 13.4.3 节）。因此，如果在行主导的栅格末尾有一个跨 3 行的栅格元素，将在显式栅格后增加三行。

默认情况下，自动增加的行是所需的最小尺寸。如果想进一步控制尺寸，使用 grid-auto-rows 和 grid-auto-columns 属性。

grid-auto-rows, grid-auto-columns

取值	*<track-breadth>* | minmax(*<track-breadth>* , *<track-breadth>*)
初始值	auto
适用于	栅格容器
计算值	取决于具体的轨道尺寸
备注	<track-breadth> 代表 <length> | <percentage> | <flex> | min-content | max-content | auto
继承性	否
动画性	否

设定自动创建的行或列轨道的尺寸时，可以提供一个尺寸值，也可以提供一对极值。下面以前一节的栅格流示例的简化版为例：设置一个 2×2 栅格，但是在里面放 5 个元素。其实，我们将做两次，一次声明 grid-auto-rows，一次不声明，结果如图 13-51 所示：

```
.grid {display: grid;
    grid-template-rows: 80px 80px;
    grid-template-columns: 80px 80px;}
#g1 {grid-auto-rows: 80px;}
```

可以看到，不为自动创建的行设定尺寸时，多出的栅格元素所在的行将与栅格元素中的内容一样高，一像素也不多。但是宽度与所在的列相等，因为我们设定了列的宽度（80px）。由于行没有显式设定高度，默认为 auto，因此得到图中所示的结果。

换成列后，基本原理不变（见图 13-52）：

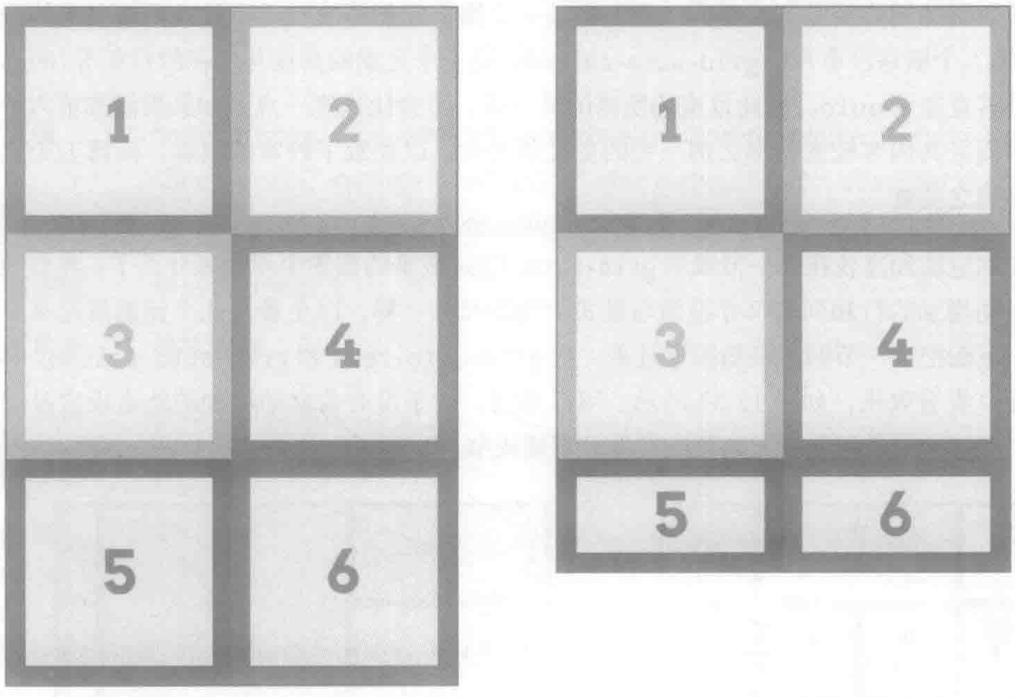


图 13-51：两个栅格，一个为自动增加的行设定了尺寸，一个没有

```
.grid {display: grid; grid-auto-flow: column;
    grid-template-rows: 80px 80px;
    grid-template-columns: 80px 80px;}
#g1 {grid-auto-columns: 80px;}
```

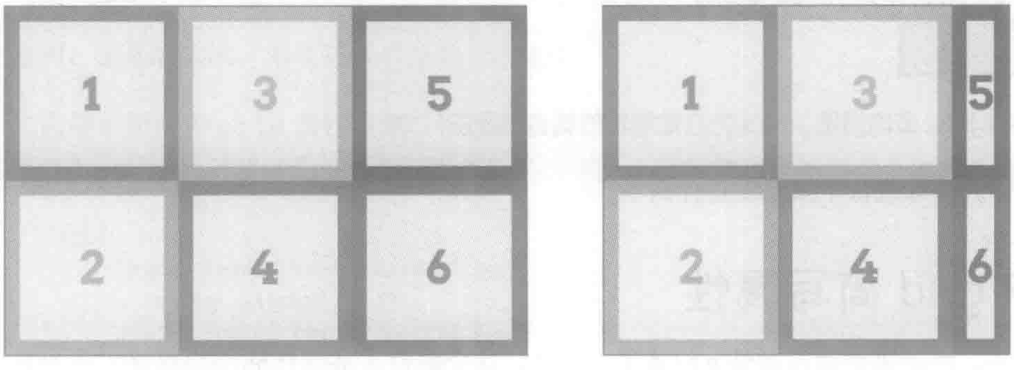


图 13-52：两个栅格，一个为自动增加的列设定了尺寸，一个没有

这里，因为栅格流是列主导的，所以最后一个栅格元素将放在超出显式栅格末端的一列中。第二个栅格没有声明grid-auto-columns，第五个元素的高度与所在的行相等(80px)，但是其宽度为auto，因此宽度为所需的最小值，不会比这宽一点。如果再添加第六个元素，而且其内容较宽，那么那一列的宽度将增大，以便放下较宽的内容，而第五个元素也会随之变宽。

现在你应该知道我在前一节展示grid-auto-flow效果的插图中用的是什么了：我背地里把自动增加的行和列的尺寸设为与显式声明的尺寸一样，以免最后几个元素看起来有差异。下面把前一节的某幅插图拿过来，把grid-auto-rows和grid-auto-columns样式去掉，看看效果，如图13-53所示。可以看到，由于没有为自动增加的轨道设定尺寸，两个栅格中的最后几个元素要比其他元素矮或窄。

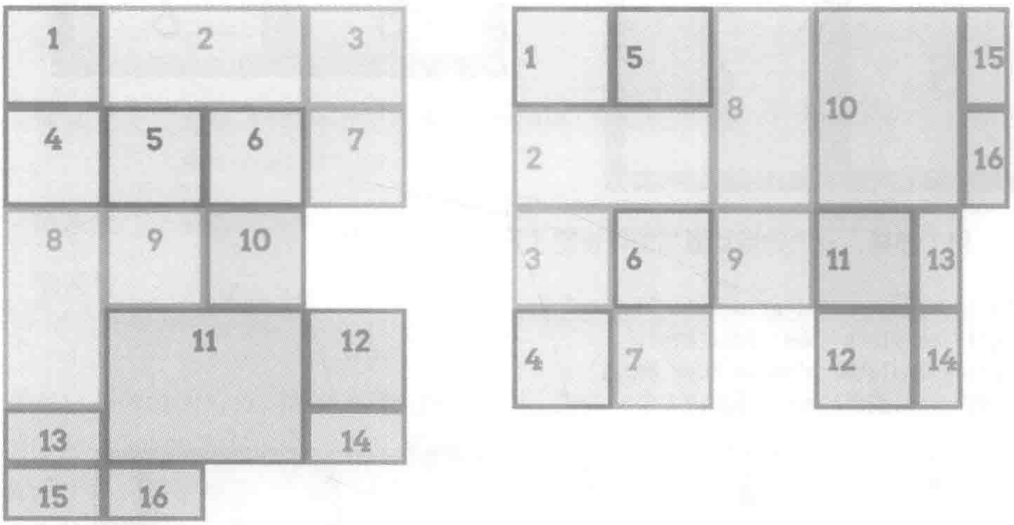


图13-53：前一节的插图，去掉为自动增加的轨道设定的尺寸

我想，你应该知道接下来我要讲什么了。

13.7 grid 简写属性

历尽千辛，终于讲到简写属性grid了。不过，这个属性可能会让你感到惊讶，因为它和其他的简写属性不一样。

grid

取值	none | subgrid | [*<grid-template-rows>* / *<grid-template-columns>*] | [*<line-names>*? *<string>* *<track-size>*? *<line-names>*?]+ [/ *<track-list>*]? | [*<grid-auto-flow>* [*<grid-auto-rows>* [/ *<grid-auto-columns>*]?]?]]
初始值	参见各单独属性
适用于	栅格容器
计算值	参见各单独属性
继承性	否
动画性	否

不可否认，取值句法看着确实有点让人头痛，但是别担心，我们将各个击破。

说到底，grid 属性的作用是以简洁的句法定义栅格模板，或者设定栅格流，并为自动增加的轨道设定尺寸。但是二者不能同时设置。

此外，未定义的值都重置为默认值，这与其他简写属性是一致的。因此，如果定义了栅格模板，那么栅格流和自动增加的轨道的尺寸都归为默认值。这其中包含我们还未谈及的栅格栏距。使用 grid 属性无法设置栏距，但是 grid 将把它重置为默认值。

是的，这是故意的。不过笔者不知道个中缘由。

先说明如何使用 grid 属性创建栅格模板。栅格模板的值有时极为复杂，而且可能让人百思不解，但是用着却十分方便。举个例子，下述 grid 声明与后面的几个规则是等效的：

```
grid:
    "header header header header" 3em
    ". content sidebar ." 1fr
    "footer footer footer footer" 5em /
    2em 3fr minmax(10em,1fr) 2em;
```

```
grid-template-areas:
    "header header header header"
    ". content sidebar ."
    "footer footer footer footer";
grid-template-rows: 3em 1fr 5em;
grid-template-columns: 2em 3fr minmax(10em,1fr) 2em;
```

注意，grid-template-rows 的值被拆开了，与 grid-template-areas 的字符串放在一起。使用字符串表示栅格区域时，grid 属性就是这样处理行尺寸的。把那些字符串去掉后，得到的值为：

```
grid:
     3em 1fr 5em / 2em 3fr minmax(10em,1fr) 2em;
```

可以看到，行轨道和列轨道之间以斜线（/）分隔。

还记得吗，简写属性未设置的值将重设为默认值？这意味着，下面两个规则是等效的：

```
#layout {display: grid;
    grid: 3em 1fr 5em / 2em 3fr minmax(10em,1fr) 2em;}

#layout {display: grid;
    grid: 3em 1fr 5em / 2em 3fr minmax(10em,1fr) 2em;
    grid-auto-rows: auto;
    grid-auto-columns: auto;
    grid-auto-flow: row;}
```

鉴于此，一定要把 grid 声明放在与栅格有关的其他声明之前。因此，如果想创建密集型列流，样式要这样写：

```
#layout {display: grid;
    grid: 3em 1fr 5em / 2em 3fr minmax(10em,1fr) 2em;
    grid-auto-flow: dense column;}
```

现在，把栅格区域的名称加上，然后再添加一些行栅格线名称。在行轨道上面的栅格线，其名称写在字符串前面，在行轨道下面的栅格线，其名称写在字符串和轨道尺寸的后面。假如我们想在中间一行的上下添加 main-start 和 main-stop 两条栅格线，并在最下面添加 page-end 栅格线：

```
grid:
    "header header header header" 3em
    [main-start] ". content sidebar ." 1fr [main-stop]
    "footer footer footer footer" 5em [page-end] /
    2em 3fr minmax(10em,1fr) 2em;
```

这个栅格如图 13-54 所示，图中标出了隐式创建的栅格线名称（例如 footer-start），以及在 CSS 中显式写出的栅格线名称。

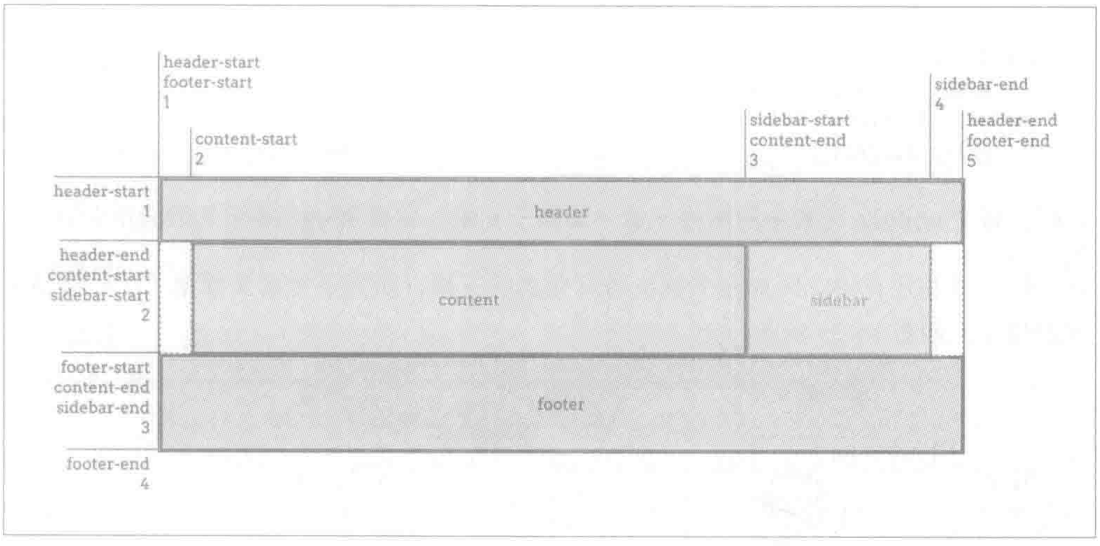


图 13-54：使用 grid 简写属性创建的一个栅格

可见，grid 属性的值很快就会变复杂。grid 属性的句法十分强大，但也不要被吓到，稍加练习便会习惯。换个角度看，复杂的句法容易出错，导致整个值失效，得不到想要的栅格。

grid 属性的另一个用法是融合 grid-auto-flow、grid-auto-rows 及 grid-auto-columns。下面两个规则是等效的：

```
#layout {grid-auto-flow: dense rows;
    grid-auto-rows: 2em;
    grid-auto-columns: minmax(1em,3em);}

#layout {grid: dense rows 2em / minmax(1em,3em);}
```

相同的结果，使用 grid 属性时的输入量更少。不过，我要再次提醒你：如果这样写，与列轨道和行轨道相关的属性都将归为默认值。因此，下面两个规则是等效的：

```
#layout {grid: dense rows 2em / minmax(1em,3em);}

#layout {grid: dense rows 2em / minmax(1em,3em);
    grid-template-rows: auto;
    grid-template-columns: auto;}
```

因此，同样要记得把简写属性写在可能把它覆盖的其他属性前面。

子栅格

grid 属性还可以取一个值：subgrid。用法如下：

```
#grid {display: grid;
    grid: repeat(auto-fill, 2em) / repeat(10, 1% 8% 1%);}
.module {display: grid;
    grid: subgrid;}
```

此时，每个 module 元素中的栅格元素（即其子元素）将根据 #grid 定义的栅格对齐。

这个值有时非常有用。比如有个模块跨父元素的三列，而其子元素又根据“主”栅格对齐和排布，如图 13-55 所示。

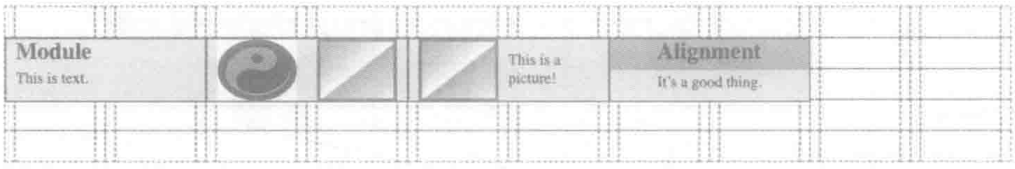


图 13-55：对齐子栅格中的元素

但问题是，写作本书时，subgrid 还是栅格布局的“不成熟”特性，有可能完全不起作用。正因为如此，我们才没有用大篇幅详细讨论这个值，只是在这短短的一小节中做个简单的介绍。

13.8 释放栅格空间

目前所见的栅格元素大都挤在一起，元素之间没有间隔。这个问题有多种解决方法，先从栏距讲起。

13.8.1 栏距

简单来说，栏距（gutter）是两个栅格轨道之间的间隔，好似把轨道之间的栅格线加粗，让它具有一定的宽度。栏距与表格样式中的 border-spacing 很像，一方面，栏距能在栅格单元之间添加间隔，另一方面，一个轴上只能设定一个间隔值。栏距使用 grid-row-gap 和 grid-column-gap 属性设定[译注 1]。

译注 1：这两个属性现已改名为 row-gap 和 column-gap。

grid-row-gap, grid-column-gap

取值	*<length>* | *<percentage>*
初始值	0
适用于	栅格容器
计算值	一个绝对长度
继承性	否
动画性	是

注意，取值句法表明，这两个属性的值只能是一个长度。但是不易看出的是，这个长度必须为非负数。百分数、份数（fr）或极值是无效的。如果想让列之间有 1 em 的间隔，声明 grid-column-gap: 1em 即可。结果显而易见，栅格中的每一列都将向后移 1 em，如图 13-56 所示。

```
#grid {display: grid;
    grid-template-rows: 5em 5em;
    grid-template-columns: 15% 1fr 1fr;
    grid-column-gap: 1em;}
```

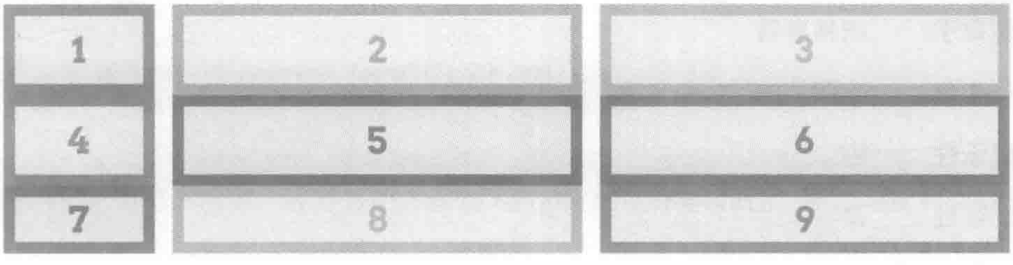


图 13-56：在列之间添加栏距

在设定栅格轨道的尺寸时，栏距被视作栅格轨道。因此，在下面的样式中，以份数设定的栅格行，其高度为 140 像素：

```
#grid {display: grid; height: 500px;
    grid-template-rows: 100px 1fr 1fr 75px;
    grid-row-gap: 15px;}
```

栅格容器的高度为500像素，减去两个行轨道的高度（100像素和75像素）后得到325像素，再减去三个栏距的高度（一个栏距高15像素，共计45像素），等于280像素，除以2（因为两行所占的份数是一样的）后得到每一份占140像素。如果栏距增加到25像素，那么每一行的高度为250像素，除以2，等于125像素。

具体情况可能比这复杂很多，这里使用像素值是为了简化计算过程。不同的单位可以随意混用，甚至还可以为栅格轨道的尺寸设置极值。这是栅格布局的主要优势之一。



栏距的具体大小受 `align-content` 和 `justify-content` 的影响，可能与声明的值不同。详情参见 13.9 节。

你可能猜到了，有个简写属性能把行和列的栏距合在一起设定[译注 2]。

grid-gap

取值	*<grid-row-gap> <grid-column-gap>*
初始值	0 0
适用于	栅格容器
计算值	声明的值
继承性	否
动画性	是

这个属性其实没什么好讲的：提供两个非负长度，分别定义行栏距和列栏距。下面举个例子，结果如图 13-57 所示：

```
#grid {display: grid;
    grid-template-rows: 5em 5em;
```

译注 2：这个属性现已改名为 gap。

```
    grid-template-columns: 15% 1fr 1fr;
    grid-gap: 12px 2em;}
```

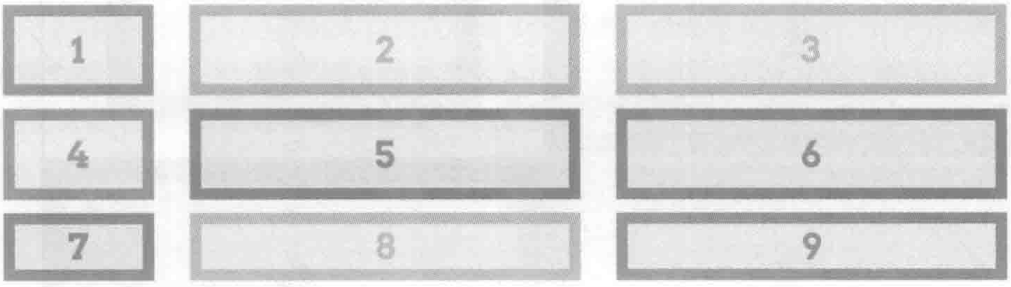


图 13-57：定义栅格的栏距

13.8.2 栅格元素与盒模型

至此，我们知道如何创建栅格、如何把元素附加到栅格中，也知道如何在栅格轨道之间添加栏距了。那么，如果为附加到栅格中的元素添加其他样式，例如外边距，情况如何呢？如果是绝对定位呢？这些样式对栅格有何影响呢？

先看外边距。基本原理是，元素在外边距的边界处附加到栅格中。这意味着，设为正外边距时，元素的可见部分将在所处的栅格区域中向内收缩，而设为负外边距时，可见部分将向外扩张。例如，下述样式将得到如图 13-58 所示的结果：

```
#grid {display: grid;
    grid-template-rows: repeat(2, 100px);
    grid-template-columns: repeat(2, 200px);}
.box02 {margin: 25px;}
.box03 {margin: -25px 0;}
```

之所以显示为图中那样，是因为 width 和 height 的值都是 auto，所以元素可以根据需要自由伸缩。如果 width 和 / 或 height 的值不是 auto，那么为了满足算式，外边距的值将被覆盖。这与元素的尺寸过约束时左右外边距中有一个会被覆盖基本上是一个道理。

比如说有个元素使用下述样式装饰，而这个元素放在一个 200 像素宽、100 像素高的栅格区域中：

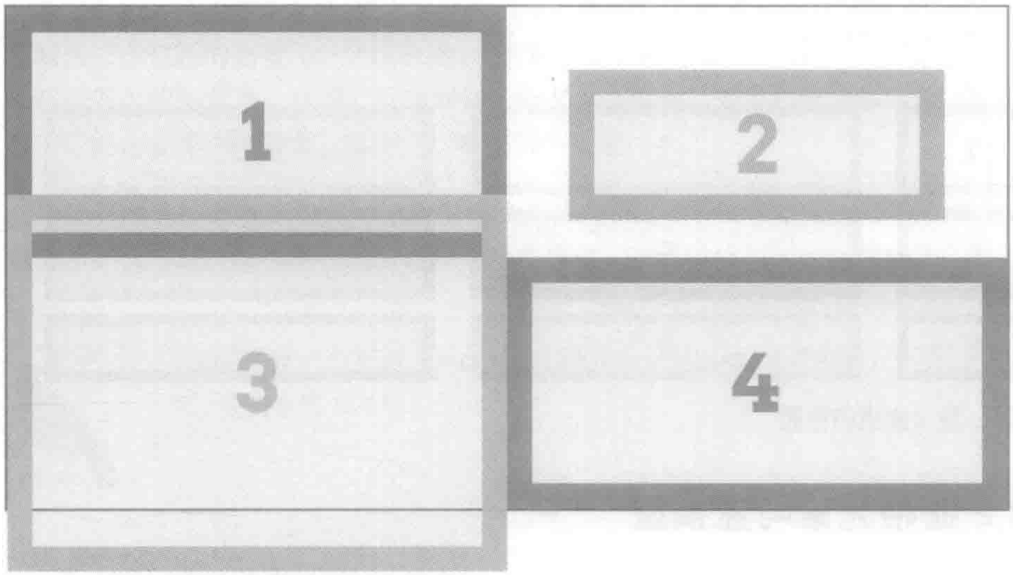


图 13-58：有外边距的栅格元素

```
.exel {width: 150px; height: 100px;
    padding: 0; border: 0;
    margin: 10px;}
```

先看横向尺寸：两侧各有 10 像素外边距，width 的值为 150px，总和为 170 像素。因此，某个尺寸必须做出让步。在这个例子中，做出让步的是右外边距（针对从左至右书写的语言），它的值将变成 40px。这样，10 像素宽的左外边距，150 像素宽的内容框，再加上 40 像素宽的右外边距，正好等于 200 像素宽的栅格区域。

在纵轴上，下外边距将被重置为 -10px。从 110 像素高的上外边距和内容上去掉这个值后，正好等于 100 像素高的栅格区域。



计算栅格轨道的尺寸时将忽略栅格元素的外边距。这意味着，不管栅格元素的外边距有多大，都不会改变 min-content 列的尺寸，增加栅格元素的外边距也不会改变使用 fr 单位设定尺寸的栅格轨道的尺寸。

与块级布局一样，可以使用 auto 值指定让哪边的外边距做出让步。假设我们想让栅格元素向栅格区域的右边对齐。把栅格元素的左外边距设为 auto 便会得到这样的结果：

```
.exel {width: 150px; height: 100px;
    padding: 0; border: 0;
    margin: 10px; margin-left: auto;}
```

这个元素的右外边距和内容框的宽度之和为 160 像素，栅格元素的宽度与之相减便是左外边距的值，因为左外边距的值被显式设为 auto。结果如图 13-59 所示，除左边之外，exel 元素每一边的外边距都是 10 像素，而左外边距是 40 像素（参见前面的计算过程）。



图 13-59：通过值为 auto 的外边距对齐栅格元素

在块级布局中我们经常把元素的左右外边距设为 auto，让元素居中显示在容纳块中，当然此时要显式设定元素的 width。在栅格布局中，情况变了，同样的效果只能在纵轴上实现。为元素设定绝对高度后，把上下外边距都设为 auto，元素将纵向居中显示。图 13-60 展示了图像（本身就有高度和宽度）的外边距为 auto 时的不同情况：

```
.i01 {margin: 10px;}
.i02 {margin: 10px; margin-left: auto;}
.i03 {margin: auto 10px auto auto;}
.i04 {margin: auto;}
.i05 {margin: auto auto 0 0;}
.i06 {margin: 0 auto;}
```



对齐栅格元素还有其他方式，例如使用 justify-self 属性，它不要求显式设定 height 和 width。详情参见下一节。

这与绝对定位的元素处理外边距和元素尺寸的方式很像。这就引出下一个问题：如果栅格元素又做了绝对定位呢？例如：

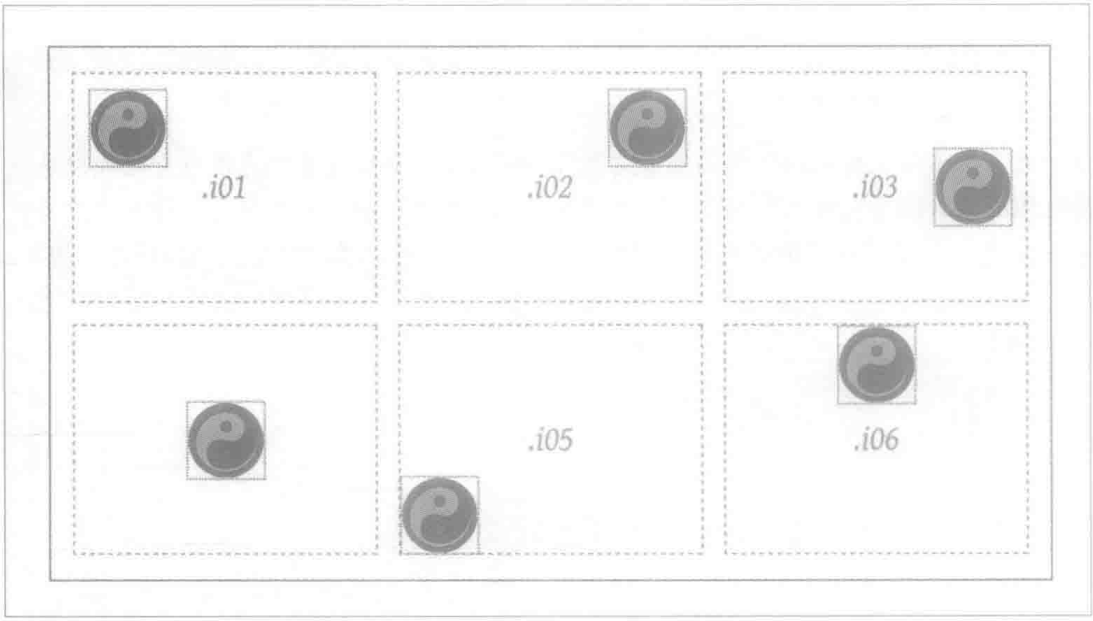


图 13-60：外边距为 auto 时不同的对齐方式

```
.exel {grid-row: 2 / 4; grid-column: 2 / 5;
    position: absolute;
    top: 1em; bottom: 15%;
    left: 35px; right: 1rem;}
```

答案简洁明了：开始栅格线和结束栅格线围成的栅格区域用作容纳块和定位上下文，栅格元素就在这个上下文中定位。这意味着，偏移属性（top 等）都相对所在的栅格区域计算。因此，上述 CSS 将得到如图 13-61 所示的结果。

与绝对定位相关的各个方面，包括偏移、外边距、元素的尺寸等，都在这个格式化上下文中处理。只不过，格式化上下文现在是由栅格区域定义的。

不过，绝对定位带来一个问题：与栅格线有关的属性，其 auto 值的行为变了。例如，在绝对定位的栅格元素上应用 grid-column-end:auto，结束栅格线为一条新的特殊栅格线，在栅格容器的内边距边界上。即使显式栅格比栅格容器小（不是不可能）也是如此。

举个例子。把前例修改成下面这样，得到的结果如图 13-62 所示：

```
.exel {grid-row: 1 / auto; grid-column: 2 / auto;
    position: absolute;
    top: 1em; bottom: 15%;
    left: 35px; right: 1rem;}
```

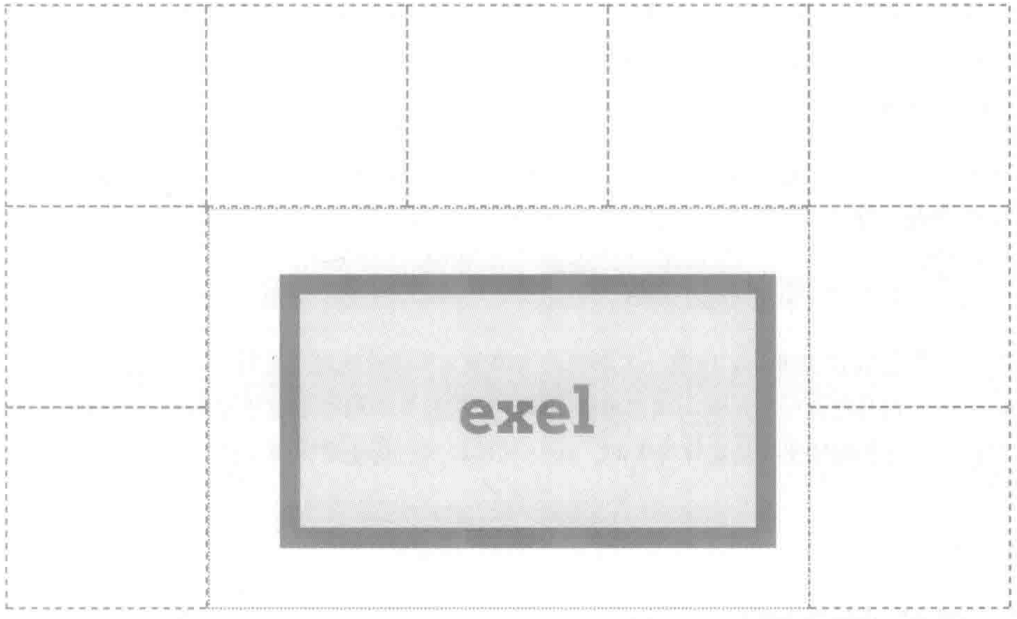


图 13-61：绝对定位一个栅格元素

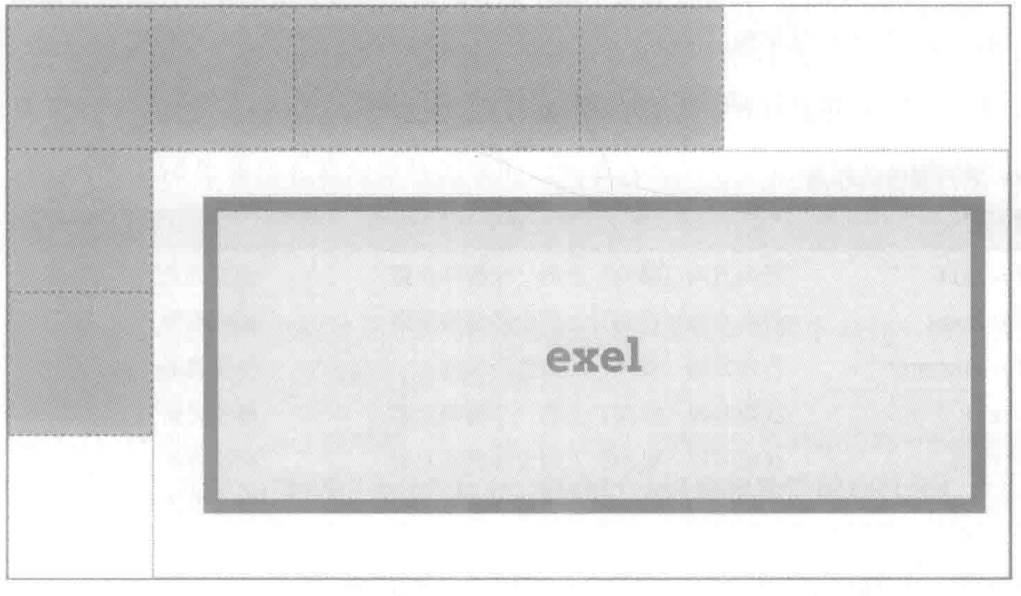


图 13-62：绝对定位对 auto 值的影响

注意，现在定位上下文从栅格容器的顶边开始，一直延伸到栅格容器的右边界，尽管栅格在那个边界之前早已结束。

这种行为的一个影响是，如果绝对定位的栅格元素没有指定开始栅格线和结束栅格线，那么定位上下文为栅格容器的内边距内边界。此时不用为栅格容器声明position:relative，也无需使用其他任何用于确立定位上下文的技巧。

注意，绝对定位的栅格元素对栅格单元和轨道尺寸没有任何影响。倘若关注的是栅格本身，定位的栅格元素好似不存在一样。设置好栅格之后，栅格元素在由栅格线定义的定位上下文中定位。



截至2017年年末，没有一个浏览器支持前文所述的绝对定位行为。若想重现，只能先相对定位一个元素，确立栅格区域，然后在里面绝对定位子元素。本节与绝对定位有关的插图都是这样得来的。auto值的特殊行为也未得到支持。

13.9 栅格的对齐方式

学过弹性盒之后我们知道，弹性盒有多个用于对齐的属性和值。这些属性在栅格布局中也能使用，而且作用十分相像。

先简单回顾一下。可用的对齐属性及其作用见表13-1。

表13-1：对齐属性的作用

属性	对齐的目标	适用于
justify-self	行内方向（横向）上的一个栅格元素	栅格元素
justify-items	行内方向（横向）上的全部栅格元素	栅格容器
justify-content	行内方向（横向）上的整个栅格	栅格容器
align-self	块级方向（纵向）上的一个栅格元素	栅格元素
align-items	块级方向（纵向）上的全部栅格元素	栅格容器
align-content	块级方向（纵向）上的整个栅格	栅格容器

表13-1表明，各justify-*属性改变的是行内轴（在英语中，指横向）上的对齐方式。三个属性之间的区别是对齐的目标不同，有的用于对齐单个栅格元素，有的用于对齐栅格中的全部栅格元素，还有的用于对齐整个栅格。类似地，align-*属性影响的是块级轴（在英语中，指纵向）上的对齐方式。

13.9.1 纵向对齐和横向对齐单个元素

*-self 属性最简单，下面在一个栅格中展示 justify-self 属性不同值的效果，然后在另一个栅格中展示同样的值在 align-self 属性中的效果（见图 13-63）。



图 13-63：行内方向和块级方向上元素自身的对齐方式

图 13-63 中的每个栅格元素在自己的栅格区域中（以蓝色虚线表示），每个区域中都有一个标注，指明使用的是哪个值。下面逐个简单说明一下。

首先要明确的一点是，不管使用哪个值，只要没有显式设定 width 或 height，元素的内容将自动“收缩”，而不是像默认行为那样，占满整个栅格区域。

start 和 end 使栅格元素向栅格区域的起边或终边对齐，这没什么难理解的。类似地，center 在对齐的轴上居中显示栅格元素，此时无需额外声明外边距等属性，例如 height 和 width。

left 和 right 在横轴上的对齐方式与你预期的一样，然而如果赋予 align-self 属性（设置的是纵向对齐方式），当做 start 处理。

self-start 和 self-end 难理解一些。self-start 把栅格元素向起边那一侧的栅格区域的边界对齐。在图 13-63 中，self-start 和 self-end 框设置了 direction:rtl，即把语言的书写方向设为从右至左，因此元素的起边为右边界，终边为左边界。所以，在第一个栅格中，self-start 框是向右对齐的，而 self-end 框是向左对齐的。然而，在第二个栅格中，书写方向对块级轴上的对齐没有影响。因此，self-start 当做 start 处理，self-end 当做 end 处理。

最后一个值，stretch，也不太容易理解。对比后可以发现，两个栅格中的其他框体是自动“收缩”的，只有内容那么大。而 stretch 值将在指定的方向上拉伸元素，从一边延伸到对边：align-self:stretch 纵向拉伸栅格元素，justify-self:stretch 横向拉伸栅格元素。这个结果可能与你想的一样，但是要注意，只有元素尺寸相关的属性设为 auto 才有这样的效果。因此，对下述样式来说，第一个示例将纵向拉伸，而第二个示例则不会拉伸：

```
.exel01 {align-self: stretch; height: auto;}
.exel02 {align-self: stretch; height: 50%;}
```

这是因为第二个示例为 height 设定的值不是 auto（这是默认值），所以 stretch 无法调整它的尺寸。对 justify-self 来说，影响这个行为的是 width。

纵向对齐还有两个值可用，不过它们的行为不那么好理解，因此需要单独说明。这两个值把栅格元素中的第一条或最后一条基线与栅格轨道中最高或最低的基线对齐。假如想把栅格元素中的最后一条基线与同一个行轨道中最高的那个栅格元素的最后一条基线对齐，可以这样声明：

```
.exel {align-self: last-baseline;}
```

反过来，如果想把第一条基线与同一个行轨道中最低的第一条基线对齐，要这样声明：

```
.exel {align-self: baseline;}
```

如果栅格元素没有基线，或者在指定的基线对齐方向上无法比较基线，那么 baseline 将当做 start 处理，last-baseline 将当做 end 处理。



本节故意略过了两个值：flex-start 和 flex-end。这两个值只应该在弹性盒布局中使用，在其他布局上下文中，包括栅格布局，等效于 start 和 end。

13.9.2 纵向对齐和横向对齐全部元素

现在来讲 align-items 和 justify-items。这两个属性的取值与前一节一样，而且作用也相同，只不过应用的对象是栅格容器中的全部栅格元素，而且必须应用到栅格容器上，而不能应用到单个栅格元素上。

因此，可以像下面这样让栅格中的所有栅格元素在各自的栅格区域里居中对齐，结果如图 13-64 所示：

```
#grid {display: grid;
    align-items: center; justify-items: center;}
```

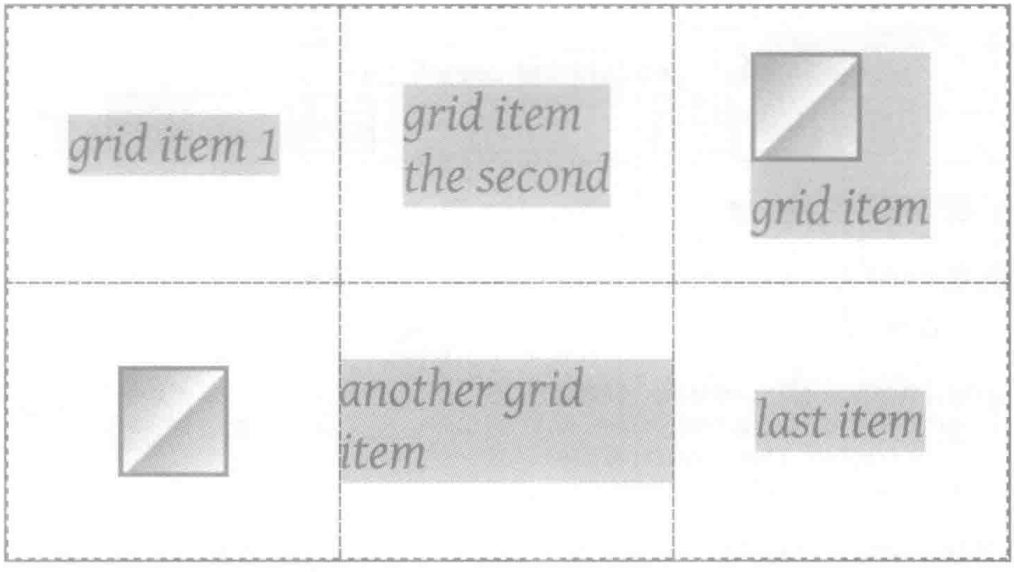


图 13-64：居中对齐所有栅格元素

可以看到，每个栅格元素都在所属的栅格区域中横向和纵向居中。而且，没有显式设定宽度和高度的栅格元素将自动“收缩”，而不占满整个栅格区域。如果显式设定了高度和宽度，那就采用设定的尺寸，然后再居中显示在栅格区域中。

除了可以纵向对齐和横向对齐每个栅格元素之外，还可以使用 `align-content` 和 `justify-content` 属性以一定的方式分布栅格元素，抑或横向对齐或纵向对齐整个栅格。设定分布方式的值不多，图 13-65 展示了可用于 `justify-content` 属性的各个值的效果，各栅格共用的样式如下：

```
.grid {display: grid; padding: 0.5em; margin: 0.5em 1em; width: auto;
    grid-gap: 0.75em 0.5em; border: 1px solid;
    grid-template-rows: 4em;
    grid-template-columns: repeat(5, 6em);}
```

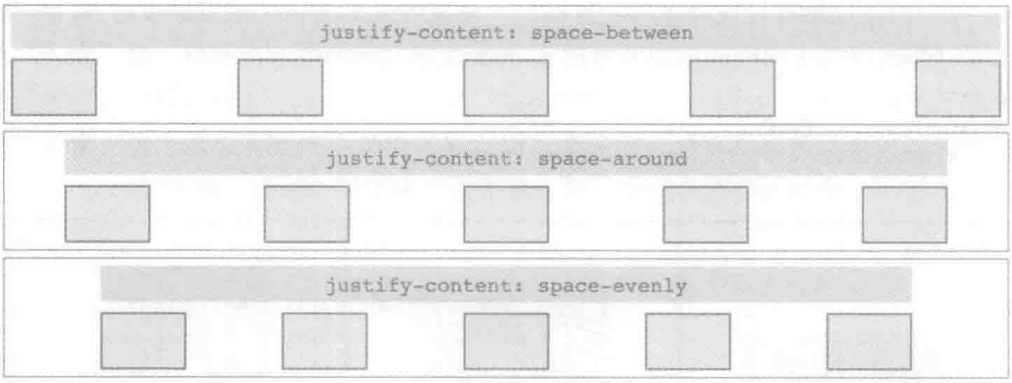


图 13-65：横向分布栅格元素

在列轨道中也能实现同样的效果，如图 13-66 所示，不过要换用 align-content 属性。这一次，各栅格共用下述样式：

```
.grid {display: grid; padding: 0.5em;
    grid-gap: 0.75em 0.5em; border: 1px solid;
    grid-template-rows: repeat(4, 3em);
    grid-template-columns: 5em;}
```

分布栅格元素时，栅格轨道（包括栏距）的尺寸还像往常一样处理。如果栅格容器中的空间有余，即栅格轨道没有从栅格容器的一边延伸到对边，那么余下的空间将根据 justify-content（横向）或 align-content（纵向）的值分配。

分配多出的空间其实就是调整栅格栏距的尺寸。如果没有声明栏距，那么分配的空间将成为栏距。如果声明了栏距，其尺寸将根据分配的情况而调整。

注意，由于仅当轨道没有填满栅格容器时才会分配空间，所以栏距的尺寸只会增大。如果轨道的尺寸比容器大（十分常见），那就没有多余的空间可分配（缺少的空间不可分割）。

还有一个分布值比较新，前面的插图没有展示，即 stretch。这个值把余下的空间均匀分配给各个栅格轨道，而不分配给栏距。因此，如果多出 400 像素空间，栅格容器中有 8 个栅格轨道，那么各个栅格轨道的尺寸将增加 50 像素。栅格轨道的尺寸不是按比例增加的，而是等量增加。截至 2017 年年末，没有浏览器支持用于设定栅格分布方式的 stretch 值。

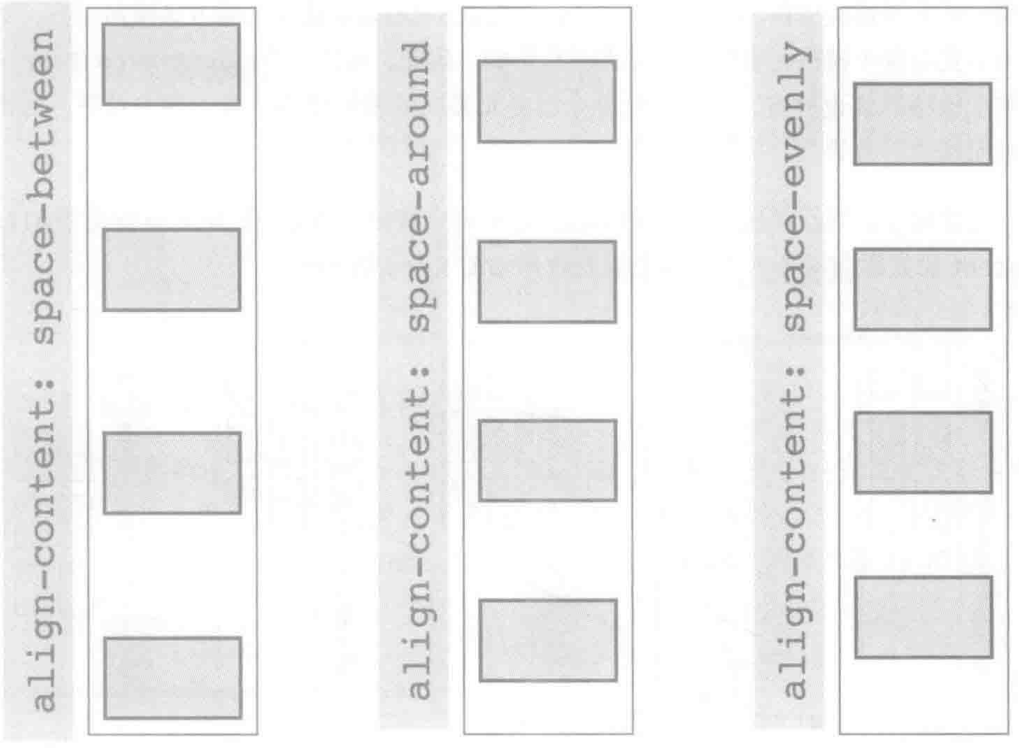


图 13-66：纵向分布栅格元素

最后，我们来看一下与分布相反的效果，即对齐栅格轨道。图 13-67 是栅格轨道横向对齐的不同方式。

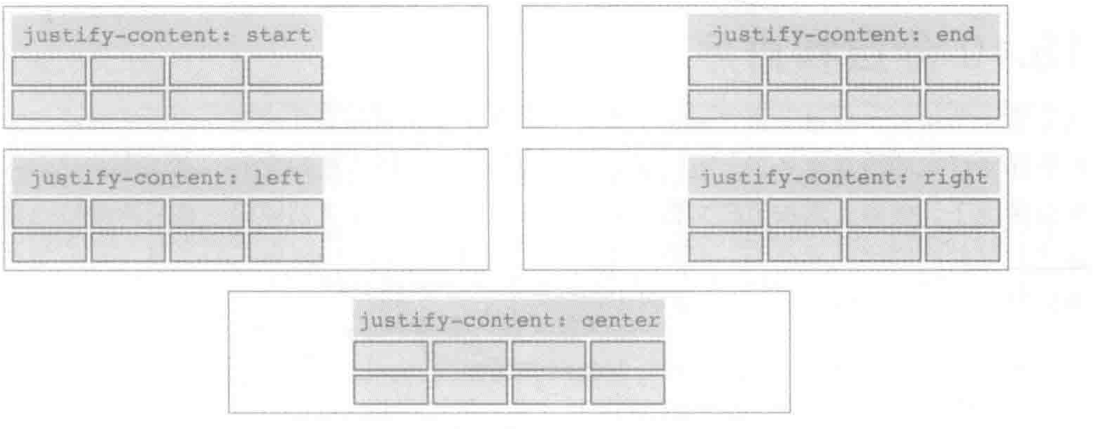


图 13-67：横向对齐栅格

这里，所有栅格轨道作为一个整体以 justify-content 属性设定的方式横向对齐。这个对齐方式对单个栅格元素的对齐方式没有影响。因此，可以使用 justify-content:end 让整个栅格靠终边对齐，然后再让单个栅格元素在所属的栅格区域中向左对齐、居中对齐或向起边对齐等。

你可能想到了，既然能使用 justify-content 设定横向对齐方式，那么就能使用 align-content 设定纵向对齐方式。各个值的效果如图 13-68 所示。

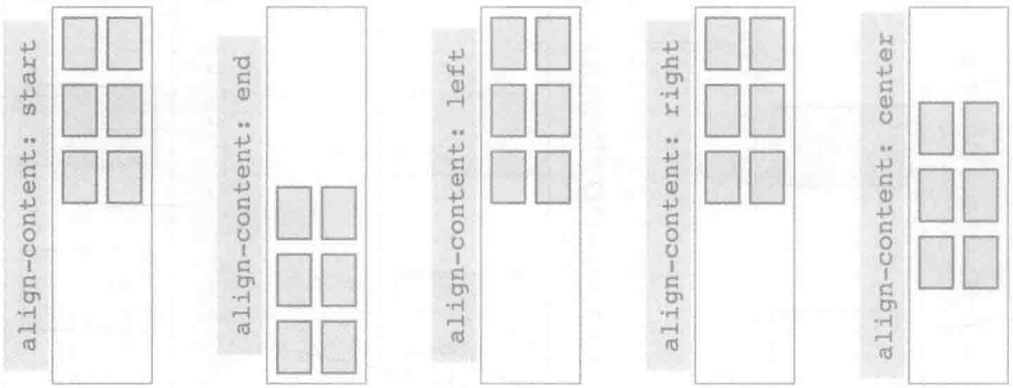


图 13-68：纵向对齐栅格

在纵向上，left 和 right 没有实际意义，因此将当做 start 处理。其他值的效果从名称上便可以看出来。

13.10 分层和排序

前面有一节讲过，栅格元素之间完全有可能重叠，原因可能是使用负外边距把栅格元素拉到栅格区域的边界之外了，也可能是两个栅格元素的栅格区域在同一个栅格单元中。默认情况下，栅格元素按照在文档源码中的顺序叠放，即在文档源码中靠后的栅格元素像是在靠前的栅格元素前面。因此，下述样式将得到如图 13-69 所示的结果（假设类名中的数字表示各栅格元素在文档源码中的顺序）。

```
#grid {display: grid; width: 80%; height: 20em;
    grid-rows: repeat(10, 1fr); grid-columns: repeat(10, 1fr);}
.box01 {grid-row: 1 / span 4; grid-column: 1 / span 4;}
.box02 {grid-row: 4 / span 4; grid-column: 4 / span 4;}
.box03 {grid-row: 7 / span 4; grid-column: 7 / span 4;}
.box04 {grid-row: 4 / span 7; grid-column: 3 / span 2;}
.box05 {grid-row: 2 / span 3; grid-column: 4 / span 5;}
```

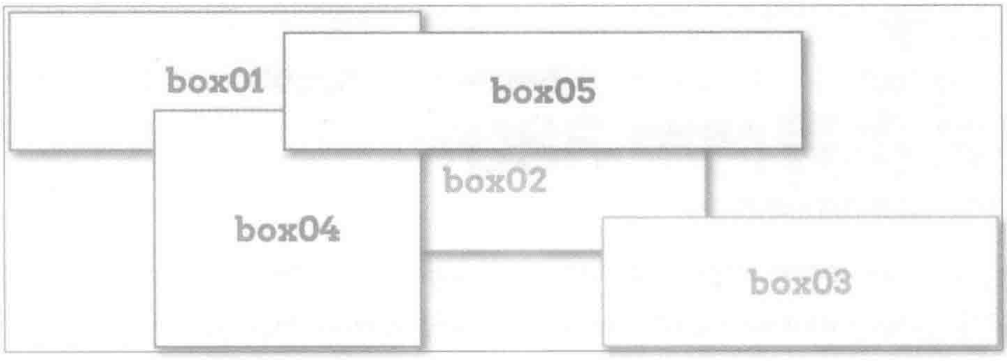


图 13-69：栅格元素按照在文档源码中的顺序叠放

如果想自己设定顺序，可以使用 z-index 属性。与在定位中一样，z-index 设定元素在 z 轴（垂直于显示平面）上的位置。正值离你较近，负值离你较远。因此，如果想把第二个框放在“上面”，只需把 z-index 设为比其他元素大的值（结果如图 13-70 所示）：

```
.box02 {z-index: 10;}
```

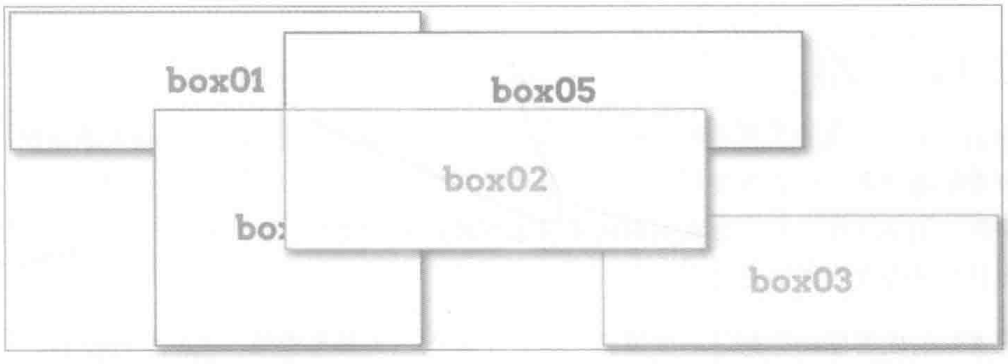


图 13-70：抬升一个栅格元素

调整栅格元素顺序的另一种方式是使用 order 属性。这个属性的作用与在弹性盒布局中一样，为栅格轨道中的栅格元素设定 order 值便可以改变栅格元素的顺序。受影响的不只元素在轨道中的位置，如果有重叠，绘制顺序也将受到影响。例如，可以像下面这样把前例中的 z-index 换成 order，得到的结果与图 13-70 一样：

```
.box02 {order: 10;}
```

这里，box02 显示在其他栅格元素的“上面”，因为它的 order 值把它放在其他元素的后面。因此，box02 最后绘制。类似地，如果这些栅格元素依序放在一个栅格轨道中，box02 的 order 值将把它放在序列的末尾，如图 13-71 所示。



图 13-71：修改栅格元素的顺序

注意，能像这样重新排列栅格元素的顺序并不意味着你应该这样做。正如栅格布局规范（4.2 节，*https://www.w3.org/TR/css-grid-1/#order-property*）所说的：

与重新排列弹性元素一样，仅当视觉顺序无需与阅读和导航顺序一致时才可以使用 order 属性；否则应该调整元素在文档源码中的顺序。

因此，可以使用 order 重排栅格元素布局顺序的唯一情况是，文档源码与布局的顺序不同。其实，不按文档源码中的顺序把栅格元素附加到栅格区域中已经能轻易做到这一点。

当然，这并不意味着 order 是没用的，应该敬而远之。既然存在，肯定有它的用武之地。但是，如果你发现在某些情况下必须使用 order，那就要仔细想一想这是不是最好的方案。

13.11 小结

栅格布局复杂而强大，没有吃透也不要灰心。栅格的运作方式要花点时间才能掌握，毕竟有大量特性是我们以前没见过的。多数特性前所未见，这就注定学习的过程是艰辛而曲折的。笔者在撰写与栅格有关的内容时也有过沮丧和迷惘，在过去二十载没有栅格布局的日子里练就的直觉完全失效了。

希望本章的内容能帮你跨过一些障碍，但是正如尤达大师所说的，“忘故而知新。”对栅格布局来说，更应该把以前掌握的知识放在一边，从头学起。随着时间地过去，你的耐心和毅力终将得到回报。

第 14 章

CSS 中的表格布局

看到本章的标题你可能一愣，“表格布局？这不正是我们极力避免的吗？”确实如此，但本章讨论的不是使用表格布局，而是通过 CSS 排布表格的方式。这个话题看似简单，实则复杂得多。

与文档布局中的其他内容相比，表格不同寻常。在弹性盒和栅格出现之前，只有表格能把不同尺寸的元素整齐排列在一起。例如，不管单元格中的内容是多是少，一行中的所有单元格都具有相同的高度；在同一列中，单元格的宽度也是如此。相邻的单元格可以共用一个边框，即使两个单元格的边框式样差异巨大。在阅读本章的过程中你会发现，这些特性是由表格特有的行为和规则（很多是历史遗留问题）决定的。

14.1 表格格式化

在深究单元格的边框是如何绘制的、表格的尺寸是如何确定的之前，先来了解表格的基本构成，以及表格中元素之间的关系。这方面的知识称为表格格式化，它与表格布局是完全不同的话题，只有格式化完成后才能开始布局。

14.1.1 表格的视觉排布

首先要知道如何通过 CSS 定义表格的排布方式。这是基础，掌握之后才能知道如何以最好的方式装饰表格。

对 CSS 而言，表格元素和表格内部元素是两个不同的概念。在 CSS 中，表格内部元素生成矩形框，有内容、内边距和边框，但是没有外边距。因此，外边距无法定义两个单元格之间的间隔。遵守 CSS 规范的浏览器将忽略应用到单元格、行和其他表格内部元素（表题例外，参见 14.1.5 节）上的外边距。

表格排布有六条基本规则。这些规则的基础是单元格，即由绘制表格的栅格线围成的区域。在图 14-1 中的两个表格里，栅格单元以虚线表示。

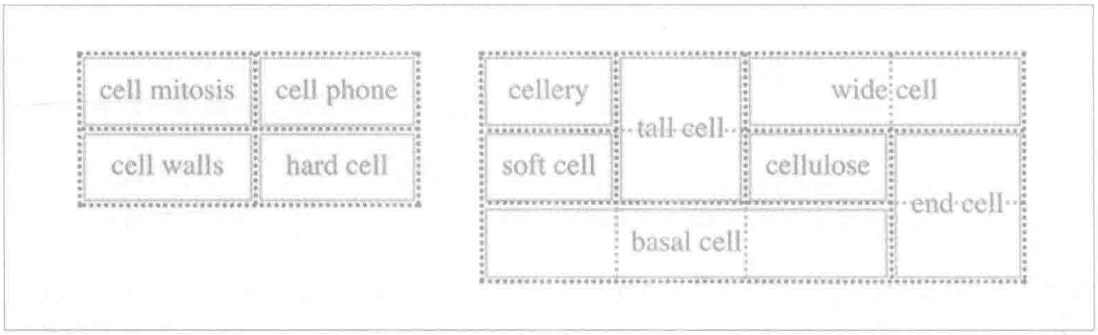


图 14-1：表格布局的基础是栅格单元

在简单的 2×2 表格中，例如图 14-1 中左边那个，栅格单元对应的就是单元格。在复杂的表格中，例如图 14-1 中右边那个，有些单元格跨多个栅格单元，但是请注意，每个单元格的边界都沿着栅格单元的边界放置。

栅格单元大都是理论上的结果，无法装饰，甚至不能通过文档对象模型（Document Object Model，DOM）访问。我们只是通过它说明，在 CSS 的语义下，表格是如何构成的。

表格排布规则

- 一个行框（row box）包含一个由栅格单元构成的行。表格中的全部行框按出现在文档源码中的顺序从上到下排列（不过表头行框或表脚行框例外，它们分别出现在表格的开头和结尾）。因此，表格中栅格行的的数量与行元素（例如 `tr` 元素）的数量相等。
- 一个行组（row group）框包含的栅格单元就是行组中各行框包含的栅格单元。
- 一个列框（column box）包含一个或多个由栅格单元构成的列。列框按照出现在文档源码中的顺序依次排列。在从左至右书写的语言中，第一个列框位于左侧，在从右至左书写的语言中则位于右侧。

- 一个列组（column group）框包含的栅格单元就是列组中各列框包含的栅格单元。
- 虽然单元格可能跨多行或多列，但是CSS没有定义具体方式。这一方面由编写文档的语言完成。跨行或跨列的单元格生成一个矩形框，其宽度和高度等于一个或多个栅格单元的宽度和高度。位于这个矩形顶部的行在跨行或跨列的栅格单元的父级行中。在从左至右书写的语言中，这个单元格的矩形框必须在最左边，但是不能与其他单元格的矩形框重叠。而且，这个矩形框必须在文档源码中同一行中靠前的单元格的右侧（对从左至右书写的语言而言）。在从右至左书写的语言中，跨行或跨列的单元格必须在最右边，不能与其他单元格重叠，而且必须在文档源码中同一行中靠前的单元格左侧。
- 单元格的矩形框不能超出表格或行组的最后一个行框。如果表格的结构可能导致这样的结果，单元格必须变小，直至能在表格或行组中放下为止。



CSS规范不建议（但也没禁止）定位单元格和其他表格内部元素。比如说，定位包含跨行单元格的行将显著改变表格的布局，这会把行从表格中完全移除，从而在其他行的布局中无需考虑那个跨行的单元格。不过，在目前的浏览器中完全可以定位表格元素。

按照定义，栅格单元都是矩形的，但不一定具有相同的尺寸。一列中的所有栅格单元具有相同的宽度，一行中的所有栅格单元具有相同的高度。但是不同行中的栅格单元不一定具有相同的高度，不同列中的栅格单元也不一定具有相同的宽度。

了解这些基本规则之后，你可能心生一个疑问：那么到底如何确定一个元素是不是单元格呢？

14.1.2 设定显示方式的值

在HTML中，很容易知道哪些元素是表格的组成部分，因为浏览器能处理`tr`和`td`等元素。而在XML中，无法确定哪些元素是表格的一部分。此时就要使用`display`的一系列值了。

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

本章只讨论与表格有关的值，其他值则与表格无关。下面简要说明与表格有关的值：

table

把元素定义为块级表格。即，定义一个生成块级框的矩形框。显然，对应的 HTML 元素是 table。

inline-table

把元素定义为行内表格。即，定义一个生成行内框的矩形框。在与表格无关的值中，与之最接近的是 inline-block。最接近的 HTML 元素是 table，然而 HTML 中的表格默认不是行内元素。

table-row

把元素定义为由单元格构成的行。对应的 HTML 元素是 tr。

table-row-group

把元素定义为由一行或多行构成的行组。对应的 HTML 元素是 tbody。

table-header-group

与 table-row-group 十分相似，不过在视觉上，表头行组始终显示在其他行和行组前面，并且显示在上表题后面。如果表格要分为多页打印，用户代理可能会在每一页的顶部重复渲染表头行组（Firefox 就会这么做）。规范没有定义把 table-header-group 应用到多个元素上会得到什么结果。表头行组可以包含多行。对应的 HTML 元素是 thead。

table-footer-group

与 table-header-group 十分相似，不过表脚行组始终显示在其他行和行组后面，并且显示在下表题前面。如果表格要分为多页打印，用户代理可能会在每一页的底部重复渲染表脚行组。规范没有定义把 table-footer-group 应用到多个元素上会得到什么结果。对应的 HTML 元素是 tfoot。

table-column

把元素声明为由单元格构成的列。在 CSS 中，把元素的 display 属性设为这个值没有任何视觉效果，好似设为 none 一样。这个值的作用基本上是定义列中单元格的表现。对应的 HTML 元素是 col。

table-column-group

把元素声明为有一列或多列构成的列组。与 table-column 元素一样，table-column-group 元素也不渲染，而是用于定义列组中元素的表现。对应的 HTML 元素是 colgroup。

table-cell

把元素定义为表格中的一个单元格。HTML 中的 th 和 td 都是应用 table-cell 的元素。

table-caption

定义表题。CSS 没有规定有多个元素使用 caption 时该怎么处理，但是明确提醒了，“创作人员不应该在表格或行内表格中的多个元素上应用 display: caption”。

CSS 2.1 规范的附录 D 给出了一个应用于 HTML 4.0 的样式表，摘录如下，从中可以一窥这些值的作用：

```
table {display: table;}
tr {display: table-row;}
thead {display: table-header-group;}
tbody {display: table-row-group;}
tfoot {display: table-footer-group;}
col {display: table-column;}
colgroup {display: table-column-group;}
td, th {display: table-cell;}
caption {display: table-caption;}
```

XML 中的元素默认没有显示方式语义，这些值的作用就体现出来了。请看下面的标记：

```
<scores>
    <headers>
        <label>Team</label>
        <label>Score</label>
    </headers>
    <game sport="MLB" league="NL">
        <team>
            <name>Reds</name>
            <score>8</score>
        </team>
        <team>
            <name>Cubs</name>
            <score>5</score>
        </team>
    </game>
</scores>
```

这段标记可以使用下述样式格式化为表格数据：

```
scores {display: table;}
headers {display: table-header-group;}
game {display: table-row-group;}
```

```
team {display: table-row;}
label, name, score {display: table-cell;}
```

各单元格可以根据需要进一步装饰，比如说加粗 label 元素，右对齐 scores 元素。

行主导

CSS 中的表格模型是“行主导”的。也就是说，这个模型假设创作人员在使用标记语言时会显式声明行。而列则从行中单元格的布局衍生而来。因此，第一列由每一行中的第一个单元格构成，第二列由每一行中的第二个单元格构成，以此类推。

行主导对HTML而言没什么大问题，因为这门标记语言本身就是行导向的。而在XML中，影响可就不是一点点了，因为 XML 对创作人员定义表格标记的方式有诸多限制。鉴于CSS的表格模型是行主导的，在以列为准的标记语言中基本上是无法实现表格布局的（假设文档的表现使用 CSS 定义）。

列

虽然 CSS 的表格模型是行主导的，但是列在布局中仍有它的作用。单元格可以同时属于两个上下文（行和列），尽管在文档源码中单元格是行的后代。然而，在 CSS 中，行和行组只能应用四个与表格无关的属性：border、background、width 和 visibility。

此外，这四个属性还有一些只在列上下文中生效的特殊规则：

border
: 仅当 border-collapse 属性的值为 collapse 时才能为列和列组设置边框。此时，根据折叠算法在列和列组中各单元格的边界处绘制边框（见 14.2.2 节）。

background
: 仅当单元格及所在的行是透明背景时列和列组的背景才可见（见 14.1.4 节）。

width
: width 属性定义列或列组的最小宽度。列（或列组）中单元格的内容可能会迫使列变宽。

visibility
: 如果列或列组的 visibility 属性值为 collapse，那么列（或列组）中的单元格都不渲染。从折叠的列跨到其他列的单元格将被裁剪，好似从其他列跨到隐藏的列一样。此外，表格的总宽度将减去折叠的列所占的宽度。对列或列组而言，除了 hidden 之外，其他的 visibility 值都将被忽略。

14.1.3 匿名表格对象

有些标记语言可能没有足够的元素能完整表示 CSS 定义的表格，而且有时创作人员会忘记编写所需的全部元素。以下面的 HTML 为例：

```
<table>
    <td>Name:</td>
    <td><input type="text"></td>
</table>
```

扫一眼这段标记，你可能认为它定义了一个表格，表格中有一行，那一行里有两个单元格。但是，从结构上看，这段标记没有定义行的元素（缺少 tr）。

为了覆盖这种情况，CSS 定义了一种插入机制：以匿名对象的形式插入“缺少的”表格组件。为了了解这个机制的基本原理，我们再看一下前面缺少行元素的 HTML 示例。在 CSS 上下文中，table 元素及作为其后代的单元格之间将插入一个匿名表格行对象：

```
<table>
  <!-- 匿名表格行对象开始 -->
    <td>Name:</td>
    <td><input type="text"></td>
  <!-- 匿名表格行对象结束 -->
</table>
```

这个过程的图解如图 14-2 所示，图中的虚线表示插入的匿名表格行。



图 14-2：表格格式化过程中生成的匿名对象

CSS 的表格模型可能插入七种不同的匿名对象。类似于继承和特指度，下述七条规则也试图对 CSS 的表现方式赋予一种直观意义。

对象插入规则

1. 如果 table-cell 元素的父元素不是 table-row，在 table-cell 和其父元素之间插

入一个匿名 table-row 对象。插入的对象将包含 table-cell 元素的全部后续同辈元素。请看下述样式和标记：

```
system {display: table;}
name, moons {display: table-cell;}
<system>
    <name>Mercury</name>
    <moons>0</moons>
</system>
```

匿名 table-row 对象插在单元格元素和 system 元素之间，而且包含 name 和 moons 元素。

父元素为 table-row-group 时上述规则也成立。在前例的基础上，假设扩增为下面这样：

```
system {display: table;}
planet {display: table-row-group;}
name, moons {display: table-cell;}
<system>
    <planet>
        <name>Mercury</name>
        <moons>0</moons>
    </planet>
    <planet>
        <name>Venus</name>
        <moons>0</moons>
    </planet>
</system>
```

在这个例子中，两组单元格都将包含在插在它们和 planet 元素之间的匿名 table-row 对象里。

2. 如果 table-row 元素的父元素不是 table、inline-table 或 table-row-group，在 table-row 元素和其父元素之间插入一个匿名 table 元素。插入的对象将包含 table-row 元素的全部后续同辈元素。请看下述样式和标记：

```
docbody {display: block;}
planet {display: table-row;}
<docbody>
    <planet>
        <name>Mercury</name>
        <moons>0</moons>
    </planet>
    <planet>
        <name>Venus</name>
        <moons>0</moons>
```

```
    </planet>
</docbody>
```

因为 planet 元素的父元素的 display 值是 block，所以在 planet 元素和 docbody 元素之间将插入一个匿名 table 对象。插入的匿名 table 对象将包含两个 planet 元素，因为它们是连续的同辈元素。

3. 如果 table-column 元素的父元素不是 table、inline-table 或 table-column-group，在 table-column 元素和其父元素之间将插入一个匿名 table 对象。这与前面所述的 table-row 规则基本一样，只不过现在变成列导向了。

4. 如果 table-row-group、table-header-group、table-footer-group、table-column-group 或 table-caption 元素的父元素不是 table，在这些元素和其父元素之间插入一个匿名 table 对象。

5. 如果 table 或 inline-table 元素的子元素不是 table-row-group、table-header-group、table-footer-group、table-row 或 table-caption，在 table 元素和其子元素之间插入一个匿名 table-row 对象。插入的匿名对象包含 table-row-group、table-header-group、table-footer-group、table-row 或 table-caption 之外的子元素的全部后续同辈元素。请看下述标记和样式：

```
system {display: table;}
planet {display: table-row;}
name, moons {display: table-cell;}
<system>
    <planet>
        <name>Mercury</name>
        <moons>0</moons>
    </planet>
    <name>Venus</name>
    <moons>0</moons>
</system>
```

这里，在 system 元素与第二组 name 和 moons 元素之间将插入一个匿名 table-row 对象。planet 元素不在这个匿名对象中，因为它的 display 属性为 table-row。

6. 如果 table-row-group、table-header-group 或 table-footer-group 的子元素不是 table-row，在它们和其子元素之间插入一个匿名 table-row 对象。插入的匿名对象包含子元素的全部后续同辈元素（table-row 对象自身除外）。请看下述标记和样式：

```
system {display: table;}
planet {display: table-row-group;}
```

```
name, moons {display: table-cell;}
<system>
    <planet>
        <name>Mercury</name>
        <moons>0</moons>
    </planet>
    <name>Venus</name>
    <moons>0</moons>
</system>
```

这里，每一组 name 和 moons 元素都将包含在一个匿名 table-row 对象中。第二组满足规则 5 所述的插入条件。对第一组来说，匿名对象插在 planet 元素和其子元素之间，因为 planet 是个 table-row-group 元素。

7. 如果 table-row 元素的子元素不是 table-cell，在元素和其子元素之间插入一个匿名 table-cell 对象。插入的匿名对象包含子元素的全部后续同辈元素（table-cell 元素自身除外）。请看下述标记和样式：

```
system {display: table;}
planet {display: table-row;}
name, moons {display: table-cell;}
<system>
    <planet>
        <name>Mercury</name>
        <num>0</num>
    </planet>
</system>
```

因为 num 元素的 display 值与表格无关，所以在 planet 元素和 num 元素之间将插入一个匿名 table-cell 对象。

这个行为还涵盖匿名行内框。假设标记中没有 num 元素：

```
<system>
    <planet>
        <name>Mercury</name>
        0
    </planet>
</system>
```

此时，0 仍将包含在一个匿名 table-cell 对象中。为了进一步说明这一点，下面再举一个例子。这个例子改编自 CSS 规范：

```
example {display: table-cell;}
row {display: table-row;}
```

```
hey {font-weight: 900;}
<example>
    <row>This is the <hey>top</hey> row.</row>
    <row>This is the <hey>bottom</hey> row.</row>
</example>
```

在每个 row 元素中，文本片段和 hey 元素将包含在各自的匿名 table-cell 对象中。

14.1.4 表格中的层

为了完整表示一个表格，CSS 定义了 6 个独立的“层”，表格的不同部分分别放在不同的层上。这几个层如图 14-3 所示。

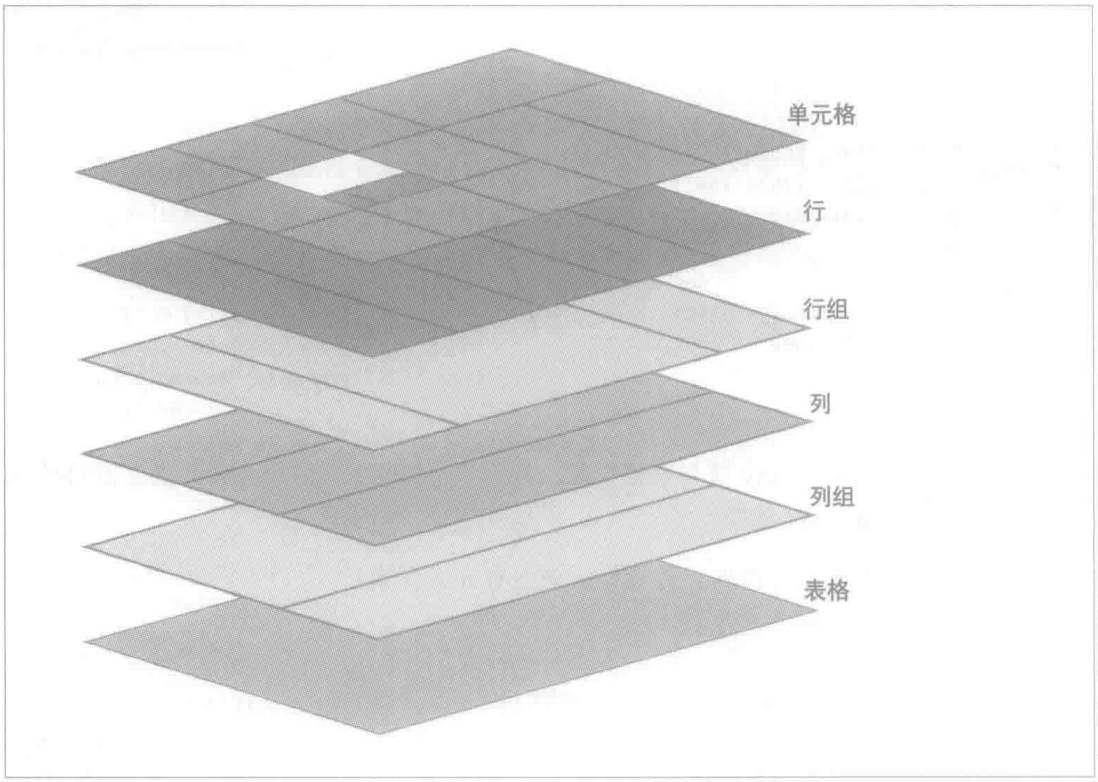


图 14-3：表示表格时采用的格式化层

一般来说，表格不同部分的样式在各自所属的层上渲染。因此，如果 table 元素有绿色的背景和一像素宽的黑色边框，那么这些样式在最底下一层渲染。针对列组的样式在倒

数第二层上渲染，针对列的样式在倒数第三层上渲染，以此类推。对应于单元格的最上层在最后绘制。

总体而言，这样的处理方式是符合逻辑的，毕竟你会想让单元格的背景在表格元素的背景之上绘制。图 14-3 揭示的最重要的一点是，列的样式在行的样式下面，因此行的背景将覆盖列的背景。

谨记一点，默认情况下所有元素的背景都是透明的。因此，对下述标记来说，表格元素的背景透过单元格、行、列等自身没有背景的元素是可见的，如图 14-4 所示：

```
<table style="background: #B84;">
    <tr>
        <td>hey</td>
        <td style="background: #ABC;">there</td>
    </tr>
    <tr>
        <td>what' s</td>
        <td>up?</td>
    </tr>
    <tr style="background: #CBA;">
        <td>not</td>
        <td style="background: #ECC;">much</td>
    </tr>
</table>
```

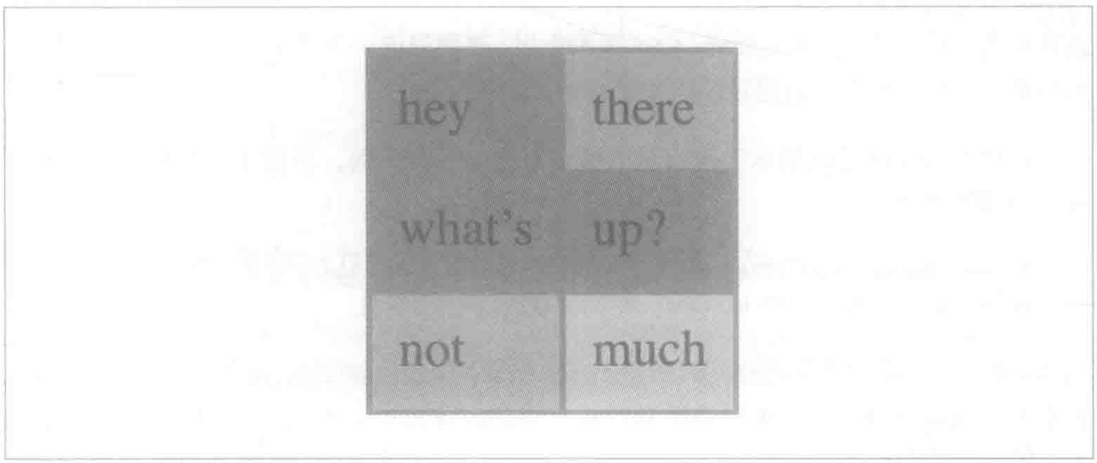


图 14-4：透过其他层看到表格格式化层的背景

14.1.5 表题

顾名思义，表题是一小段描述表格中是什么内容的文本。因此，2016 年第 4 季度股票行情表的表题内容可能是“2016 年第 4 季度股票行情”。借助 caption-side 属性，可以把表题放在表格上方或下方，而不管表题在表格的结构中位于何处（在 HTML5 中，caption 元素只能作为 table 元素的第一个子元素，不过其他语言可能有不同的规则）。

caption-side

取值	top | bottom
初始值	top
适用于	display 属性的值为 table-caption 的元素
计算值	指定的值
继承性	是
动画性	否
备注	CSS2 中有 left 和 right 两个值，但是由于缺少支持，CSS2.1 去掉了

表题有点奇怪，至少视觉上如此。CSS 规范指出，表题就像一个放在表格框之前（或之后）的块级框一样，只是，表题还从表格继承样式。

一个简单的例子就足以展示表题表现上最为重要的一些方面。请看下面的样式，结果如图 14-5 所示：

```
caption {background: #B84; margin: 1em 0; caption-side: top;}
table {color: white; background: #840; margin: 0.5em 0;}
```

caption 元素中的文本从表格那里继承 color 属性，颜色为 white，而表题自身还有背景。表格的边框外边界与表题的外边距外边界之间的间隔为 1 em，因为表格的上外边距和表题的下外边距折叠了。最后，表题的宽度由 table 元素中内容的宽度决定，可见表格元素是表题的容纳块。

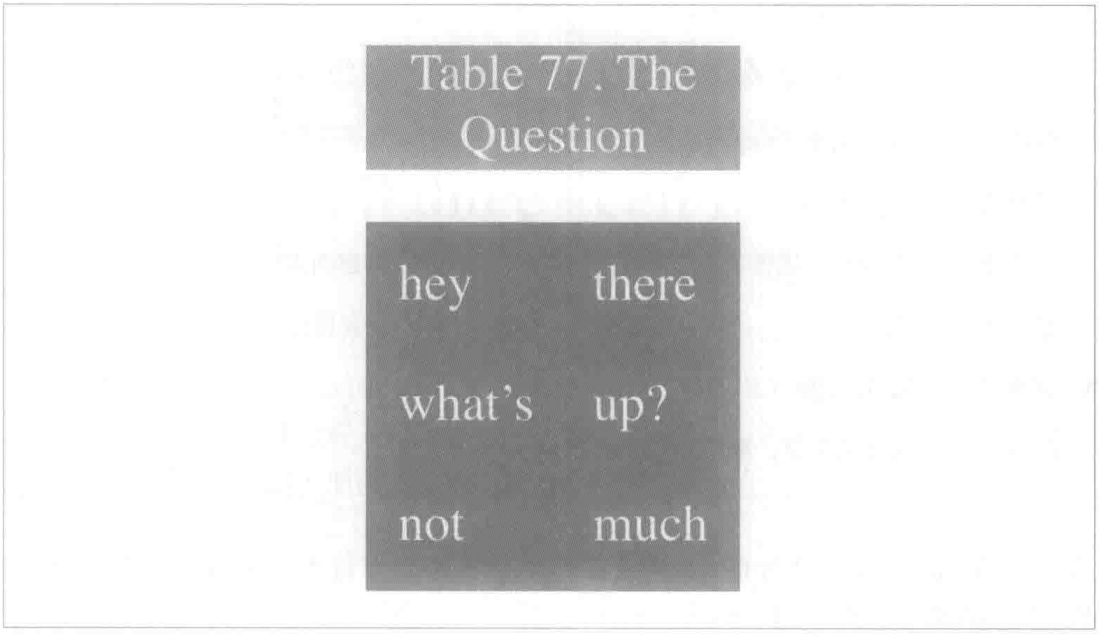


图 14-5：装饰表题和表格

把 caption-side 的值改为 bottom 之后结果还是一样，只是表题将出现在表格框的下方，而且折叠的是表题的上外边距和表格的下外边距。

多数时候，表题都像块级元素那样装饰，可以设定内边距、边框、背景等。例如，如果想改变表题中文本的横向对齐方式，可以使用 text-align 属性。因此，如果想右对齐前例中的表题，可以这样写：

```
caption {background: gray; margin: 1em 0;
    caption-side: top; text-align: right;}
```

14.2 单元格的边框

在 CSS 中有两种完全不同的边框模型。分离边框模型（separated border model）在单元格的布局之间有间隔时起作用。另一个选项是折叠边框模型（collapsed border model），此时单元格之间在视觉上没有间隔，单元格之间的边框将合并或折叠在一起。现在前者是默认模型，而在以前的 CSS 版本中，后者是默认模型。

创作人员可以使用 border-collapse 属性在这两个模型之间做出选择。

border-collapse	
取值	collapse | separate | inherit
初始值	separate
适用于	display 属性的值为 table 或 table-inline 的元素
继承性	是
计算值	指定的值
备注	在 CSS2 中，默认值为 collapse

这个属性的唯一作用是告知用户代理该采用哪种边框模式。设为 `collapse` 值时，用户代理使用折叠边框模式；设为 `separate` 值时，用户代理使用分离边框模式。先讲后一个模型，因为它更容易说明，而且是默认值。

14.2.1 分离单元格的边框

在这种模型中，表格中的每个单元格都与其他单元格相隔一定的距离，而且单元格之间的边框不会折叠在一起。因此，下述样式和标记将得到如图 14-6 所示的结果：

```
table {border-collapse: separate;}
td {border: 3px double black; padding: 3px;}
tr:nth-child(2) td:nth-child(2) {border-color: gray;}
<table cellspacing="0">
    <tr>
        <td>cell one</td>
        <td>cell two</td>
    </tr>
    <tr>
        <td>cell three</td>
        <td>cell four</td>
    </tr>
</table>
```

注意，相邻单元格的边框挨在一起，但彼此是独立的。在图中你会看到单元格之间有三条线，那是两条双线边框相邻导致的；第四个单元格的灰色边框能更清楚地表明这一点。

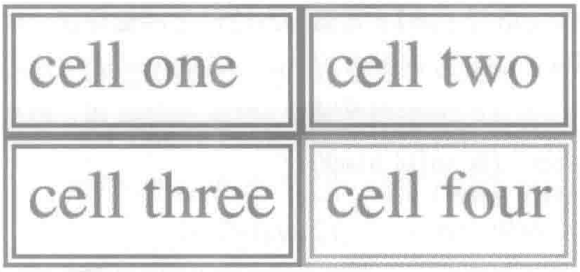


图 14-6：分离的（因此是独立的）单元格边框

前例中的 HTML 设定了 cellspacing 属性，这是为了确保单元格之间没有间隔，但这样得到的实际效果似乎不太好。转念一想，如果能定义彼此独立的边框，那么通过 CSS 就应该有办法调整单元格的间距。是的，确实有。

边框间距

把单元格的边框区分开以后，有时你可能还想让边框相隔一定的距离。利用 border-spacing 属性可以轻易做到这一点。这个属性的作用比 HTML 属性 cellspacing 强大，完全可以取而代之。

border-spacing	
取值	*<length> <length>?*
初始值	0
适用于	display 属性的值为 table 或 table-inline 的元素
计算值	两个绝对长度
继承性	是
动画性	是
备注	如果 border-collapse 属性的值不是 separate，忽略这个属性

这个属性的值可以是一个或两个长度。如果想让所有单元格都相距 1 像素，声明 border-spacing: 1px; 就行了。而如果想让单元格在横向上相距 1 像素、纵向上相距 5 像素，则声明 border-spacing: 1px5px;。提供两个值时，第一个值始终是横向间隔，第二个值始终是纵向间隔。

间隔值在表格外侧的单元格与 table 元素的内边距之间也存在。因此，下述样式将得到如图 14-7 所示的结果：

```
table {border-collapse: separate; border-spacing: 5px 8px;
padding: 12px; border: 2px solid black;}
td { border: 1px solid gray;}
td#squeeze {border-width: 5px;}
```

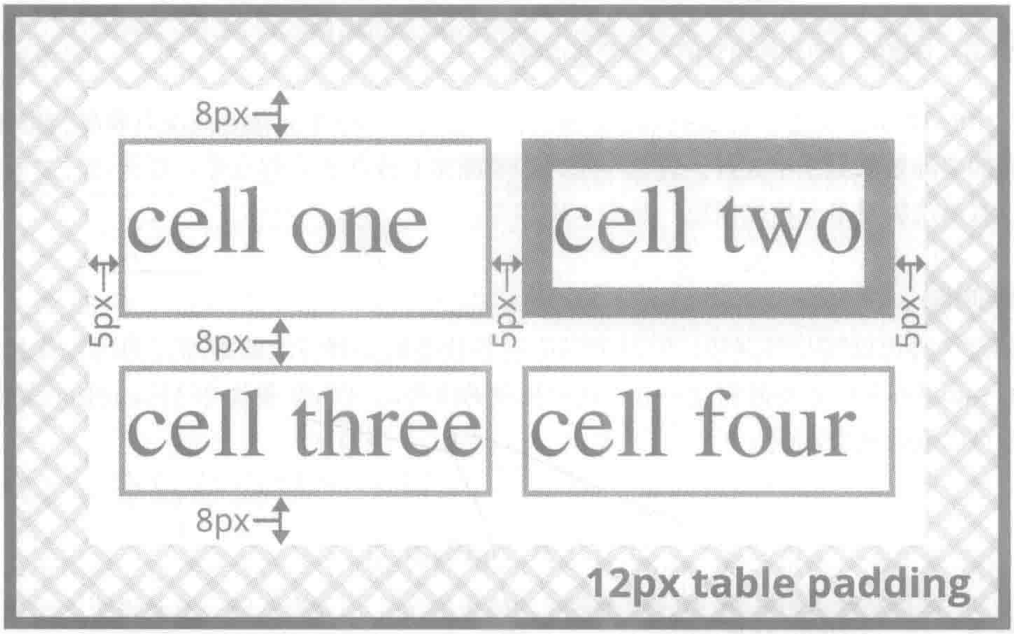


图 14-7：边框间距在单元格和所属的表格之间也存在

在图 14-7 中，相邻的单元格在横向上有 5 像素的间距，因而在最右边和最左边的单元格的边框与 table 元素的右边框和左边框之间有 17 像素的间距。类似地，相邻的单元格在纵向上相距 8 像素，因而在最上面和最下面两行中的单元格与表格的上边框和下边框之间有 20 像素的间距。在一个表格中，单元格边框之间的间距是恒定不变的，不管单元格的边框有多宽。

另外注意，border-spacing 值在表格上声明，而不是在单个单元格上声明。如果前例为 td 元素声明 border-spacing，其值将被忽略。

在分离边框模型中，不能为行、行组、列和列组设定边框。在遵守 CSS 规范的用户代理中，为这些元素设置的边框都将被忽略。

处理空单元格

一个表格中的每个单元格在视觉上都是不同的，那么空单元格（即里面没有内容）该怎么处理呢？你有两个选择，由 empty-cells 属性的值确定。

empty-cells	
取值	show | hide
初始值	show
适用于	display 属性的值为 table-cell 的元素
计算值	指定的值
继承性	是
动画性	否
备注	如果 border-collapse 属性的值不是 separate，忽略这个属性

如果把 empty-cells 属性的值设为 show，空单元格的边框和背景像往常一样渲染，就像单元格中有内容一样。如果把值设为 hide，单元格的任何部分都不渲染，好似设定 visibility:hidden 一样。

只要单元格中有内容，就不能视作空单元格。这里所说的“内容”，不只包括文本、图像、表单元素等，还有不间断空格实体（ ），以及除 CR（回车符）、LF（换行符）、制表符和空格之外的其他空白。如果一行里的所有单元格都是空的，而且 empty-cells 属性的值都是 hide，那么整行就像设置了 display: none 一样。

14.2.2 折叠单元格的边框

折叠边框模型就像 HTML 排布没有间距的单元格一样，它比分离边框模型复杂一些。折叠边框模型与分离边框模型之间的区别体现为以下几点：

- border-collapse 属性的值为 collapse 时，display 属性的值为 table 或 inline-table 的元素不能有内边距，但是可以有外边距。因此，在折叠边框模型中，表格四周的边框与最外层的单元格之间肯定没有间隔。

- 边框可以应用于单元格、行、行组、列和列组。当然，表格自身始终可以有边框。
- 在折叠边框模型中，单元格的边框之间肯定没有间隔。其实，边框在毗邻处折叠在一起，因此只会绘制一条折叠的边框。这与外边距折叠有点像，即最大的外边距胜出。单元格的边框在折叠时，“占上风的”边框胜出。
- 折叠的边框居中放在单元格之间假想的栅格线上。

接下来的两节详细探讨后两点。

折叠边框的布局

为了更好地理解折叠边框模型的工作原理，下面以表格中一行的布局为例说明，如图14-8所示。

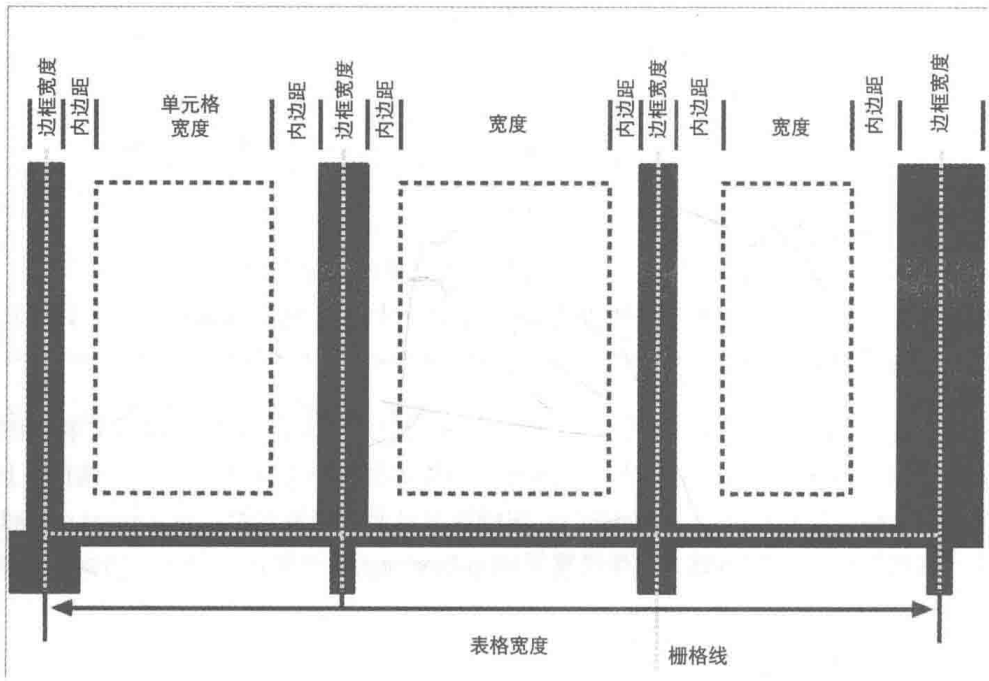


图 14-8：折叠边框模型中表格里一行的布局

对每个单元格来说，与预期一样，内边距和内容宽度在边框内侧。而单元格之间的边框，一半在两个单元格之间的栅格线一侧，另一半在栅格线的另一侧。不论如何，单元格的边界处只会绘制一条边框。你可能以为边框一分两半，分别在栅格线两侧绘制，但事实并非如此。

假设中间那个单元格的边框是绿色实线，两侧的单元格的边框是红色实线。此时，中间那个单元格左右两边的边框（与两侧相邻单元格的边框折叠在一起）要么都是绿色，要么都是红色，这取决于哪个边框胜出。下一节讨论如何判断哪个边框胜出。

你可能注意到了，外侧的边框超出了表格的宽度。这是因为，在这个模型中，表格边框的一半含在宽度之内。另一半边框在这个范围之外，位于外边距中。看起来有点奇怪，但这就是折叠边框模型定义的方式。

规范给出了一个公式，转摘如下，以防有人喜欢用公式：

> row width = (0.5 * border-width-0) + padding-left-1 + width-1 + padding-right-1 + border-width-1 + padding-left-2 +...+ padding-right-n + (0.5 * border-width-n)

`border-width-`*n* 表示第 *n* 个单元格与下一个单元格之间的边框，因此 `border-width-3` 表示第三个单元格和第四个单元格之间的边框。*n* 表示一行中单元格的总数。

这个机制有个小例外。在开始排布折叠边框的表格时，用户代理先为表格的左右边框计算一个初始值。表格左边框的初始宽度等于第一行中第一个单元格左边框的一半，表格右边框的初始宽度等于第一行中最后一个单元格右边框的一半。在后续的行中，如果左边框或右边框比边框的初始宽度大，将凸出到表格的外边距区域。

如果边框的宽度是显示单位（像素、点等）的奇数倍，由用户代理决定如何把边框居中显示在栅格线上。用户代理可能会移动边框，稍微偏离中点，或者向上或向下取整为显示单位的偶数倍，或者使用抗锯齿技术，又或者使用其他合理的方式调整。

边框折叠

相邻的两条或两条以上边框折叠在一起。其实，边框并非真的折叠，而是看哪一条占上风。具体哪条边框胜出，有严格的规则：

- 如果折叠的边框中有一条边框的 `border-style` 为 `hidden`，那么那一条边框在折叠的所有边框中优先级最高。这个位置上的边框都将隐藏。
- 如果所有边框都可见，那么较宽的边框比较窄的边框优先级高。因此，如果一条 2 像素宽的点线边框与一条 5 像素宽的双线边框折叠，这个位置上将显示 5 像素宽的双线边框。
- 如果折叠的各个边框具有相同的宽度，但是式样不同，那么优先级从大到小的顺序为：`double`、`solid`、`dashed`、`dotted`、`ridge`、`outset`、`groove`、`inset`、`none`。

因此，如果折叠的两条边框宽度相等，其中一条的式样为 dashed，另一条的式样为 outset，那么这个位置上将显示虚线边框。

- 如果折叠的各个边框具有相同的式样和宽度，但是颜色不同，那么颜色从这些元素中获取，优先级从高到低分别为：单元格、行、行组、列、列组、表格。因此，如果单元格的边框与列的边框折叠（除颜色之外都相同），那么将使用单元格的边框颜色（以及式样和宽度）。如果折叠的边框属于相同类型的元素，例如两条具有相同式样和宽度，但是颜色不同的行边框，那么颜色取自离顶部和左边较远的边框（在从左至右书写的语言中；否则取自离顶部和右边较远的边框）。

下述样式和标记分别演示上述四条规则，结果如图 14-9 所示：

```
table {border-collapse: collapse;
border: 3px outset gray;}
td {border: 1px solid gray; padding: 0.5em;}
#r2c1, #r2c2 {border-style: hidden;}
#r1c1, #r1c4 {border-width: 5px;}
#r2c4 {border-style: double; border-width: 3px;}
#r3c4 {border-style: dotted; border-width: 2px;}
#r4c1 {border-bottom-style: hidden;}
#r4c3 {border-top: 13px solid silver;}
<table>
    <tr>
        <td id="r1c1">1-1</td>
        <td id="r1c2">1-2</td>
        <td id="r1c3">1-3</td>
        <td id="r1c4">1-4</td>
    </tr>
    <tr>
        <td id="r2c1">2-1</td>
        <td id="r2c2">2-2</td>
        <td id="r2c3">2-3</td>
        <td id="r2c4">2-4</td>
    </tr>
    <tr>
        <td id="r3c1">3-1</td>
        <td id="r3c2">3-2</td>
        <td id="r3c3">3-3</td>
        <td id="r3c4">3-4</td>
    </tr>
    <tr>
        <td id="r4c1">4-1</td>
        <td id="r4c2">4-2</td>
        <td id="r4c3">4-3</td>
```

```
        <td id="r4c4">4-4</td>
    </tr>
</table>
```

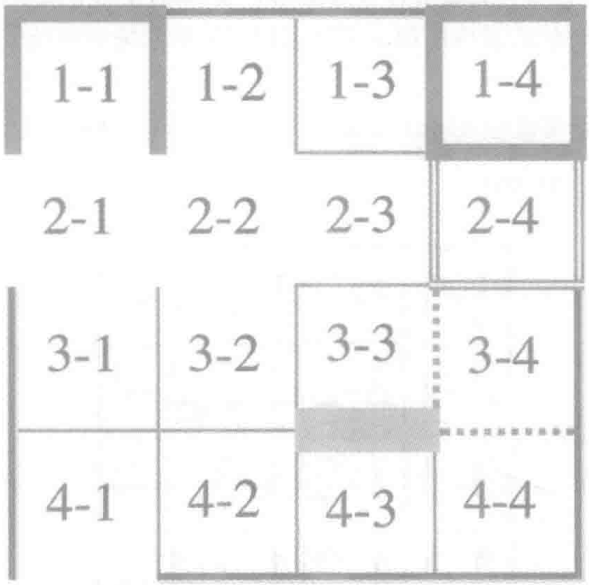


图 14-9：对边框宽度、式样和颜色的处理方式可能得到与往常不太一样的结果

我们来看看每个单元格的情况：

- 单元格 1-1 和 1-4 的边框宽度为 5 像素，比相邻的任何一条边框都宽，因此它们不仅在毗邻的边框中胜出，还战胜了表格的边框。但是单元格 1-1 的下边框例外，这一边的边框被抹除了。
- 单元格 1-1 的下边框之所以被抹除，是因为单元格 2-1 和 2-2 的边框是隐藏的，单元格边界处根本没有边框。同样，表格的边框（单元格 2-1 的左边）也被单元格的边框击败了。单元格 4-1 的下边框也是隐藏的，因此底边没有边框。
- 单元格 2-4 的 3 像素双线边框在顶边被单元格 1-4 的 5 像素实线边框覆盖。而单元格 2-4 的边框又把它与单元格 2-3 之间的边框覆盖了，因为单元格 2-4 的边框更宽，所以“占上风”。单元格 2-4 还覆盖了它与单元格 3-4 之间的边框，虽然两条边框的宽度相等，但是单元格 2-4 的双线式样比单元格 3-4 的点线边框“占上风”。
- 单元格 3-3 底边的 13 像素银色边框不只覆盖单元格 4-3 的上边框，还影响两个单元格和所在行中内容的布局。

- 表格外圈没有特别装饰的单元格，其 1 像素宽的实线边框被表格元素的 3 像素外凸边框覆盖。

这些规则看似复杂，但基本上还是符合直觉的，多加练习也不难掌握。不过，需要指出的是，Netscape 1.1 时代的表格使用下面这组相当简单的规则就能复现，结果如图 14-10 所示：

```
table {border-collapse: collapse; border: 2px outset gray;}
td {border: 1px inset gray;}
```

1-1	1-2	1-3	1-4
2-1	2-2	2-3	2-4
3-1	3-2	3-3	3-4
4-1	4-2	4-3	4-4

图 14-10：复现旧时表格的表现

14.3 表格的尺寸

深入讨论过表格格式化和单元格边框的表现之后，接下来要学习表格的尺寸和内部元素。表格的宽度有两种确定方式：固定宽度布局和自动宽度布局。不管使用哪种方式确定宽度，表格的高度都自动计算。

14.3.1 宽度

既然表格的宽度有两种确定方式，那么必然有这么一种方式用于声明具体使用哪种方式。创作人员可以使用 table-layout 属性在这两种宽度计算方式中做出选择。

table-layout	
取值	auto | fixed
初始值	auto
适用于	display 属性的值为 table 或 inline-table 的元素
计算值	指定的值
继承性	是
动画性	否

虽然在同一个表格上应用这两种模型可能得到不同的结果，但二者之间最显著的区别还是速度。用户代理计算表格的布局时，固定宽度的表格布局要比自动宽度模型快一些。

固定布局

固定布局模型之所以更快，主要原因是无需考虑单元格中的内容。在这种模型中，布局由表格、列和单元格的宽度决定。

- 如果列元素的 width 属性的值不是 auto，设定的就是整列的宽度。
- 如果列的宽度为 auto，但是表格第一行中位于这一列的单元格的 width 属性的值不是 auto，那么这个单元格的宽度就是整列的宽度。如果这个单元格跨多列，其宽度将平分到各列中。
- 如果列的宽度仍是自动确定的，将自动确定，尽可能保持各列宽度相等。

经过上述过程处理之后，表格的宽度将被设为表格的 width 值和各列宽度之和中较大的那个。如果表格的宽度比列的宽度之和大，二者之差除以列数得到的结果将增加到各列上。

这种方式之所以较快，是因为列的宽度由表格的第一行决定。后续各行中的单元格都与第一行确定的列宽度保持一致。后续行中的单元格不改变列的宽度（其实也无法改变），

因此为这些单元格设定的 width 值将被忽略。倘若内容在单元格中放不下，overflow 的值决定内容是被裁剪、是可见，还是显示滚动条。

请看下述样式和标记，结果如图 14-11 所示：

```
table {table-layout: fixed; width: 400px;
    border-collapse: collapse;}
td {border: 1px solid;}
col#c1 {width: 200px;}
#r1c2 {width: 75px;}
#r2c3 {width: 500px;}
<table>
    <colgroup> <col id="c1"><col id="c2"><col id="c3"><col id="c4"> </colgroup>
    <tr>
        <td id="r1c1">1-1</td>
        <td id="r1c2">1-2</td>
        <td id="r1c3">1-3</td>
        <td id="r1c4">1-4</td>
    </tr>
    <tr>
        <td id="r2c1">2-1</td>
        <td id="r2c2">2-2</td>
        <td id="r2c3">2-3</td>
        <td id="r2c4">2-4</td>
    </tr>
    <tr>
        <td id="r3c1">3-1</td>
        <td id="r3c2">3-2</td>
        <td id="r3c3">3-3</td>
        <td id="r3c4">3-4</td>
    </tr>
    <tr>
        <td id="r4c1">4-1</td>
        <td id="r4c2">4-2</td>
        <td id="r4c3">4-3</td>
        <td id="r4c4">4-4</td>
    </tr>
</table>
```

从图 14-11 中可以看到，第一列的宽度为 200 像素，恰好是表格宽度（400 像素）的一半。第二列的宽度是 75 像素，因为我们为第一行中位于这一列的单元格显式设定了宽度。第三列和第四列的宽度都是 61 像素。为什么呢？因为第一列和第二列的宽度之和（275 像素）加上两列之间边框的宽度（3 像素）等于 278 像素，400 减去 278 等于 122，而 122 除以 2 等于 61，所以第三列和第四列的宽度是 61 像素。那么为 #r2c3 设定的 500 像素宽度呢？这个值被忽略，因为该单元格不在表格的第一行中。

200px	75px	61px	61px
1-1	1-2	1-3	1-4
2-1	2-2	2-3	2-4
3-1	3-2	3-3	3-4
4-1	4-2	4-3	4-4

图 14-11：固定宽度的表格布局

注意，使用固定宽度布局模型时，并非一定要为表格显式设定 `width`，不过设定的话会有很大的帮助。例如，对下述样式来说，用户代理计算出来的表格宽度比父元素的宽度小 50 像素。在固定布局模型中，计算得到的宽度就用作表格的宽度。

```
table {table-layout: fixed; margin: 0 25px; width: auto;}
```

然而，我们不一定非要这么做。用户代理还能使用自动宽度布局模型排布 `width` 的值为 `auto` 的表格。

自动布局

虽然自动宽度布局模型没有固定布局的速度快，但是你可能对它更熟悉，因为它与 HTML 常年使用的表格布局模型基本一样。在当下多数用户代理中，把表格的 `width` 属性设为 `auto` 就能触发这个模型，而不管 `table-layout` 是什么值，但是也不一定。

自动布局较慢的原因在于，用户代理要读取完表格中的全部内容才能开始排布。即，用户代理排布表格时要把每个单元格的内容和样式考虑在内。一般来说，这要求用户代理先做些计算，然后反过来再执行一系列计算。

之所以要读取完全部内容是因为，与 HTML 表格一样，表格的布局取决于每个单元格中的内容。如果最后一行中某个单元格里有个 400 像素宽的图像，那么上面每个单元格（在同一列中）的宽度都将变成 400 像素。因此，在排布表格之前，每个单元格的宽度都要计算出来，然后再做调整（可能还会导致新一轮宽度计算）。

这种模型的具体过程见下述步骤：

1. 计算一列中每个单元格的最小宽度和最大宽度。

a. 确定显示内容最少需要多少宽度。在这个过程中，内容可以换行，但是不能超出单元格框。如果为单元格设定了 width 值，而且比最小宽度大，那么单元格的最小宽度将被设为 width 的值。如果单元格的 width 值为 auto，那么单元格的最小宽度将被设为内容的最小宽度。

b. 最大宽度即不换行（强制换行除外，例如
 元素）显示内容所需的宽度。

2. 计算各列的最小宽度和最大宽度。

a. 列的最小宽度由那一列中单元格最小宽度值最大的那个确定。如果为列显式设定了 width 值，而且比列中每个单元格的最小宽度都大，那么列的最小宽度将被设为 width 的值。

b. 列的最大宽度等于那一列中单元格最大宽度值最大的那个。如果为列显式设定了 width 值，而且比列中每个单元格的最大宽度都大，那么列的最大宽度将被设为 width 的值。这两个行为与传统的 HTML 表格是一样的，即列的宽度将增大到与最宽的单元格相等。

3. 如果单元格横跨多列，那么所跨的那几列的最小宽度之和必须等于单元格的最小宽度。同样，列的最大宽度之和也要等于横跨多列的单元格的最大宽度。用户代理应该把列宽度的变化平均分到所跨的各列中。

此外，用户代理必须根据表格的宽度（即使尚不知道具体值）计算百分数设定的列宽度。用户代理要把这个百分数值留存下来，在后续计算中使用。

至此，用户代理已经知道每一列有多宽了。得到这些信息之后，用户代理才真正开始计算表格的宽度。具体步骤如下：

1. 如果表格宽度的计算值不是 auto，比较表格宽度的计算值与每一列的宽度之和加上边框和单元格间距（使用百分数设定的宽度可能就在此时计算）。二者之中较大的那个就是表格的最终宽度。如果表格宽度的计算值比列宽度、边框和间距之和大，差值除以列数后得到的值将增加到各列上。

2. 如果表格宽度的计算值是 auto，表格的最终宽度等于列宽度、边框和间距之和。这意味着，表格的宽度将正好能放得下内容，这与传统的 HTML 表格是一样的。宽度为百分数的列将以那个百分数为限，但用户代理不一定会满足这个限制。

只有等到最后一步计算完毕之后，用户代理才能排布表格。

下述样式和标记有助于理解上述过程，结果如图 14-12 所示：

```
table {table-layout: auto; width: auto;
    border-collapse: collapse;}
td {border: 1px solid; padding: 0;}
col#c3 {width: 25%;}
#r1c2 {width: 40%;}
#r2c2 {width: 50px;}
#r2c3 {width: 35px;}
#r4c1 {width: 100px;}
#r4c4 {width: 1px;}
<table>
    <colgroup> <col id="c1"><col id="c2"><col id="c3"><col id="c4"> </colgroup>
    <tr>
        <td id="r1c1">1-1</td>
        <td id="r1c2">1-2</td>
        <td id="r1c3">1-3</td>
        <td id="r1c4">1-4</td>
    </tr>
    <tr>
        <td id="r2c1">2-1</td>
        <td id="r2c2">2-2</td>
        <td id="r2c3">2-3</td>
        <td id="r2c4">2-4</td>
    </tr>
    <tr>
        <td id="r3c1">3-1</td>
        <td id="r3c2">3-2</td>
        <td id="r3c3">3-3</td>
        <td id="r3c4">3-4</td>
    </tr>
    <tr>
        <td id="r4c1">4-1</td>
        <td id="r4c2">4-2</td>
        <td id="r4c3">4-3</td>
        <td id="r4c4">4-4</td>
    </tr>
</table>
```

下面依次看一下各列的情况：

- 在第一列中，显式设定了单元格或列宽度的只有单元格 4-1，其宽度为 100px。因为内容较短，所以列的最小宽度和最大宽度都被设为 100px（如果这一列中某个单元格中的内容为几句话，那么列的最大宽度将增加，大到在不换行的情况下能显示全部文本）。
- 在第二列中，我们声明了两个 width 值：把单元格 1-2 的宽度设为 40%，把单元格 2-2 的宽度设为 50px。这一列的最小宽度为 50px，最大宽度为表格最终宽度的 40%。

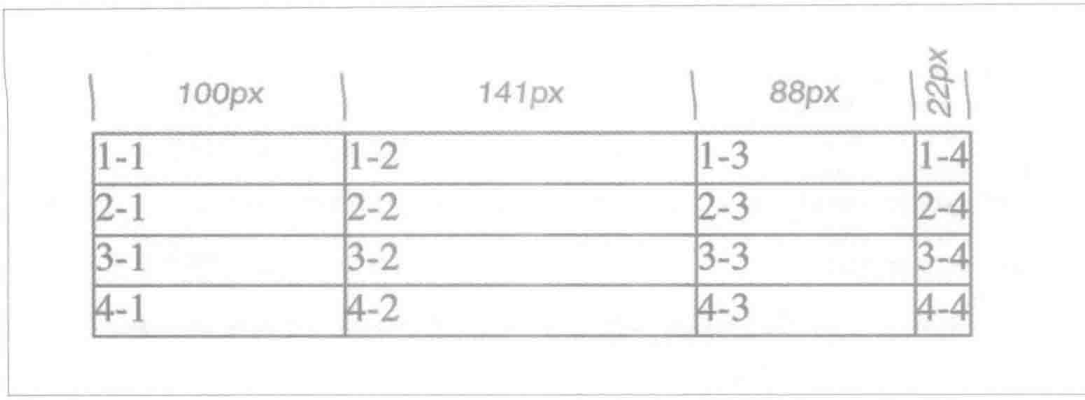


图 14-12：自动表格布局

- 在第三列中，只有单元格 3-3 显式设定了宽度（35px），不过这一列也设定了 `width`，值为 25%。因此，这一列的最小宽度为 35 像素，最大宽度为表格最终宽度的 25%。
- 在第四列中，只有单元格 4-4 显式设定了宽度（1px）。这比内容的最小宽度小，因此列的最小宽度和最大宽度相等，都等于单元格中内容的最小宽度。经计算，内容的最小宽度为 22 像素，因此最小宽度和最大宽度都是 22 像素。

现在，用户代理知道四列的最小宽度和最大宽度了，如下所示：

— 最小宽度 100 像素，最大宽度 100 像素。

— 最小宽度 50 像素，最大宽度 40%。

— 最小宽度 35 像素，最大宽度 25%。

— 最小宽度 25 像素，最大宽度 22 像素。

因此，表格的最小宽度等于各列的最小宽度加上列之间折叠的边框，即 215 像素；表格的最大宽度等于 `123px+65%`，其中 `123px` 是第一列和最后一列的最大宽度之和，外加共用的折叠边框。最大宽度最终得到的结果为 351.42857142857143 像素（假设 `123px` 是表格总宽度的 35%）。得到这个结果后，第二列的宽度便可计算出来，等于 140.5 像素，第三列的宽度也能计算出来，等于 87.8 像素。用户代理可能会把这两个值四舍五入为整数，例如 `141px` 和 `88px`，也可能不会，这取决于具体使用的渲染方法（图 14-12 中给出的是整数）。

注意，用户代理不一定选择使用最大值，有可能会执行其他操作。

这个例子比较简单，也比较容易理解（可能没有看起来那么简单），所有内容的宽度基本是相等的，而且声明的宽度用的都是像素长度。如果表格中有图像、文本段落、表单元素等，确定表格布局的过程可能会变得更为复杂。

14.3.2 高度

表格的宽度费劲千辛万苦才能计算出来，你可能以为计算高度还要复杂一些。其实，对CSS而言，这没什么难的，不过浏览器开发者可能不这么认为。

最简单的情况是，通过height属性显式设定表格的高度。此时，表格的高度由height属性的值定义。这意味着，表格的高度可能会比各列的高度之和大或小。请注意，对表格来说，height更像是min-height，因此如果你定义的height值比各行的高度之和小，用户代理可能会将其忽略。

反过来，如果表格的height值比行的高度之和大，具体如何处理，规范没有规定，而是指出这个问题可能会在以后的CSS版本中解决。用户代理可能会增加表格中各行的高度，满足声明的高度值，可能会在表格框中留白，也可能会采用其他完全不同的方式处理。具体方式由各用户代理自行决定。



截至2017年年末，用户代理最常采用的方法是增加表格中各行的高度，填满总体高度。具体方法是，计算表格的高度与各行的高度之和相差多少，然后除以行数，再把得到的值增加到每一行上。

如果表格的height属性的值为auto，那么表格的高度等于表格中每一行的高度之和加上边框和单元格的间距。为了确定每一行的高度，用户代理会使用一套类似确定列宽的计算过程。用户代理先计算各单元格中内容的最小高度和最大高度，由此得出各行的最小高度和最大高度。然后，用户代理算出各行的高度，叠加在一起得到表格的高度。这与行内布局很像，只不过对结果不那么确定。

除了如何处理显式为表格设定的高度，以及如何处理表格中各行的高度之外，CSS没有定义以下事项：

- 使用百分数设定单元格高度的效果。
- 使用百分数设定行和行组高度的效果。
- 跨行单元格对所跨行的高度有何影响（除了单元格在这些行中之外）。

可以看到，表格的高度很大程度上由用户代理确定。历史证明，不同的用户代理采用的方法有所不同，因此最好尽量不为表格设定高度。

14.3.3 对齐方式

有意思的是，相对于单元格和行的高度，定义单元格中内容的对齐方式要容易得多。甚至垂直对齐也很容易定义，而这很有可能对行的高度带来影响。

横向对齐是最简单的。若想对齐单元格中的内容，使用 `text-align` 属性。其实，单元格相当于块级框，因此单元格中的内容根据 `text-align` 属性的值对齐。

单元格中内容的纵向对齐方式使用 `vertical-align` 属性设定。这个属性的值很多都与行内内容的纵向对齐一样，不过应用到单元格上有不同的作用。下面三个值是最容易理解的：

`top`
: 单元格中内容的顶部与行的顶部对齐。在跨行的单元格中，内容的顶部与所跨的第一行的顶部对齐。

`bottom`
: 单元格中内容的底部与行的底部对齐。在跨行的单元格中，内容的底部与所跨的最后一行的底部对齐。

`middle`
: 单元格中内容的中线与行的中线对齐。在跨行的单元格中，内容的中线与所跨各行整体的中线对齐。

这三个值的效果如图 14-13 所示，使用的样式和标记如下：

```
table {table-layout: auto; width: 20em;
border-collapse: separate; border-spacing: 3px;}
td {border: 1px solid; background: silver;
    padding: 0;}
div {border: 1px dashed gray; background: white;}
#r1c1 {vertical-align: top; height: 10em;}
#r1c2 {vertical-align: middle;}
#r1c3 {vertical-align: bottom;}
<table>
    <tr>
        <td id="r1c1">
```

```
        <div>
            The contents of this cell are top-aligned.
        </div>
        </td>
        <td id="r1c2">
        <div>
            The contents of this cell are middle-aligned.
        </div>
        </td>
        <td id="r1c3">
        <div>
            The contents of this cell are bottom-aligned.
        </div>
        </td>
    </tr>
</table>
```

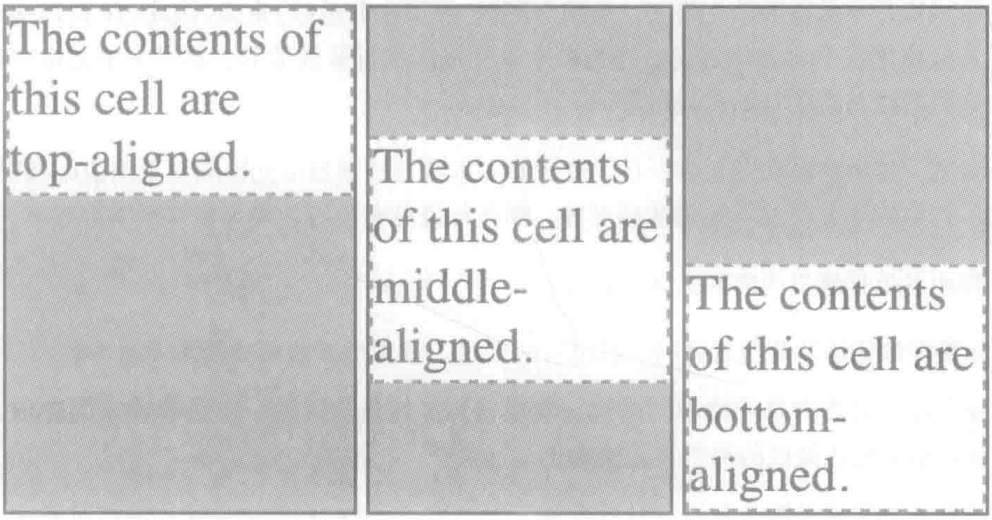


图 14-13：纵向对齐单元格中的内容

每种对齐效果都是通过自动增加单元格的内边距实现的。在图 14-13 中的第一个单元格里，单元格的下内边距变成单元格框的高度与单元格中内容的高度之差。在第二个单元格中，单元格的上下内边距被置为同样的大小，从而让内容在单元格中纵向居中。在第三个单元格中，调整的是单元格的上内边距。

第四个可用的值是 baseline，这个值比前三个稍微复杂一些：

baseline

单元格的基线与行的基线对齐。在跨行的单元格中，单元格的基线与所跨的第一行的基线对齐。

这个值的作用最好通过图示说明（见图 14-14）。

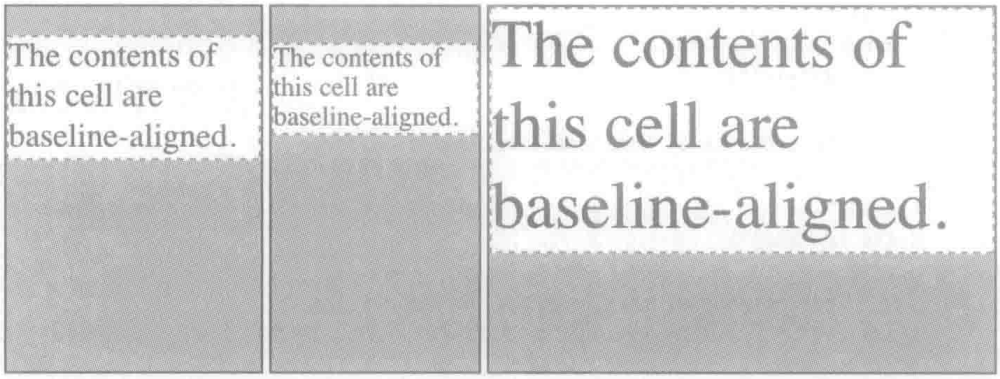


图 14-14：向基线对齐单元格的内容

行的基线由行中最低的单元格初行基线（即第一行文本的基线）定义。因此，在图 14-14 中，行的基线由第三个单元格定义，因为那个单元格的初行基线最低。前两个单元格中第一行文本的基线与这条行的基线对齐。

与上对齐、居中对齐和下对齐一样，基线对齐也是通过调整单元格的上下内边距实现的。如果一行中的单元格都不采用基线对齐，那么行就没有基线，因为根本就不需要有。

对齐行中各单元格里内容的具体过程如下：

1. 如果有单元格是基线对齐的，找出行的基线，放置沿基线对齐的单元格中的内容。
2. 放置靠上对齐的单元格中的内容。现在行的高度是临时的，由已经确定内容位置的单元格中底边最低的那个单元格确定。
3. 如果余下的单元格中有居中对齐或靠下对齐的，而且内容的高度比行的临时高度大，那么行的高度将增大，覆盖这些单元格中最高的那个。
4. 放置余下的单元格中的内容。如果单元格中内容的高度比行的高度小，单元格的内边距将增大，与行的高度保持一致。

`vertical-align` 的其他值，即 `sub`、`super`、`text-top` 和 `text-bottom`，应用到单元格上时应该被忽略，全都视作 `baseline`，有时也可能视作 `top`。

14.4 小结

以前你可能常年使用表格和间隔图设计网页，对表格布局十分熟悉，但是读过本章之后你会发现，这种布局背后的机制还是相当复杂的。由于有 HTML 表格结构在前，CSS 的表格模型也是以行为中心的，然而这个模型也对列和列的样式提供了有限的支持。CSS 提供了一些用于控制单元格对齐和表格宽度的新特性，呈现出来的表格令人更加满意。

表格相关的显示值可应用到任何元素上，由此我们便可以使用 `div` 和 `section` 等 HTML 元素创建类似表格的布局了，而且在 XML 语言中还能使用任何元素描述表格的组件。

第 15 章

列表和生成的内容

在 CSS 布局层面，列表是个有趣的存在。列表中的项目是简单的块级框，但是有一块却不参与文档布局，悬挂在列表一侧。在有序列表中，那一块内容是一系列递增的数字（或字母），由用户代理计算，大部分格式也由用户代理决定，而不受创作人员的控制。用户代理根据文档结构生成序号及其基本的表现。

这部分生成的内容无法用 CSS1 描述，因此不受 CSS1 的控制，但是 CSS2 引入的特性使得列表项目的编号可被描述了。结果是，现在 CSS 允许创作人员定义计数的模式和格式，而且可以把计数器关联到任何元素上，而不只限于有序列表的项目。此外，利用这种机制还能在文档中插入任何类型的内容，例如文本字符串、属性的值，甚至是外部资源。因此，我们可以使用 CSS 插入链接图标、编辑符号等，而无需编写额外的标记。

为了弄清这些列表选项是如何整合协作的，首先要探讨基本的列表样式，然后再分析如何生成内容和计数器。

15.1 列表

从某种意义上讲，非叙述性文本都可以视作列表。人口普查数据、太阳系、家谱、菜单，甚至你的所有朋友都可以使用列表或列表的列表表示。列表的多样性决定了它的重要性，不过在列表样式方面 CSS 能做的还不够丰富，真是遗憾。

调整列表样式最简单的（也是支持最广泛的）方式是改变记号类型。在无序列表中，记

号指显示在每个项目前面的圆点。在有序列表中，记号可以是字母、数字或其他计数系统中的符号。记号甚至还可以用图像代替。这些都通过不同的列表样式属性实现。

15.1.1 列表的类型

若想改变列表项目所用的记号类型，使用 list-style-type 属性。

list-style-type

取值	disc | circle | square | disclosure-open | disclosure-closed | decimal | decimal-leading-zero | arabic-indic | armenian | upper-armenian | lower-armenian | bengali | cambodian | khmer | cjk-decimal | devanagari | gujarati | gurmukhi | georgian | hebrew | kannada | lao | malayalam | mongolian | myanmar | oriya | persian | lower-roman | upper-roman | tamil | telugu | thai | tibetan | lower-alpha | lower-latin | upper-alpha | upper-latin | cjk-earthly-branch | cjk-heavenly-stem | lower-greek | hiragana | hiragana-iroha | katakana | katakana-iroha | japanese-informal | japanese-formal | korean-hangul-formal | korean-hanja-informal | korean-hanja-formal | simp-chinese-informal | simp-chinese-formal | trad-chinese-informal | trad-chinese-formal | ethiopic-numeric | <*string*> | none | inherit
初始值	disc
适用于	display 属性的值为 list-item 的元素
继承性	是
计算值	指定的值

是的，可用的关键字可真不少，而这还不是 list-style-type 属性整个历史上可用的全部值。有些值，例如 urdu 和 hangul-consonant，只有那么一两个浏览器支持。较旧的值没有得到广泛支持，而上面列出的值基本上所有浏览器都支持。图 15-1 给出了一些例子。

list-style-type 属性，以及与列表相关的其他所有属性都只能应用到 display 属性的值为 list-item 的元素上，但是 CSS 并不区分有序列表和无序列表的项目。因此，可以让有序列表使用圆点，而非数字。其实，list-style-type 属性的默认值就是 disc，因此自然地，你会认为，如未明确声明为其他值，所有列表（不管有序还是无序）中的项目都将使用圆点做记号。这是符合逻辑的判断，但事实上，使用什么记号是由用户代理决定的。即使用户代理没有预先定义 ol{list-style-type:decimal;} 这样的规则，也可能会禁止把有序的记号应用到无序列表上，反之亦然。一定要小心，不要依赖这个特性。



图 15-1：列表样式类型示例

以前，CSS2.1 规定，用户代理应该把无法识别的关键字值视为 decimal。截至 2017 年年初，CSSListsandCountersModule 没有严格规定，而是说可以回落到 disc 或 none（例如，如果把有序列表类型应用到无序列表上，Chrome 默认为 none）。

如果不想显示记号，应该使用的值是 none。none 的作用是禁止在本该显示记号的位置上出现任何内容，不过却不阻断有序列表的计数。因此，下述标记将得到如图 15-2 所示的结果：

```
ol li {list-style-type: decimal;}
li.off {list-style-type: none;}
<ol>
<li>Item the first
```

```
<li class="off">Item the second
<li>Item the third
<li class="off">Item the fourth
<li>Item the fifth
</ol>
```

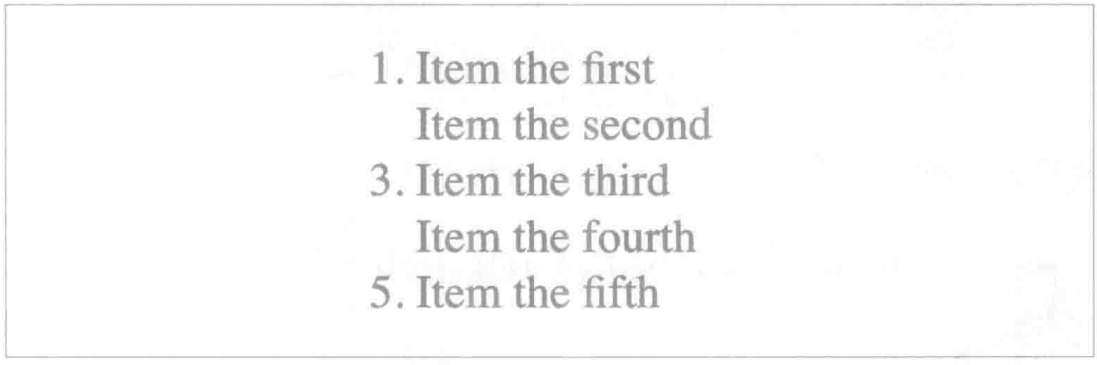


图 15-2：禁止显示列表项目的记号

list-style-type的值是继承的，因此如果希望嵌套的列表使用不同的记号，要分别定义。你可能还要为嵌套的列表显式声明样式，因为用户代理的样式表中可能已有定义。假如用户代理定义了下述样式：

```
ul {list-style-type: disc;}
ul ul {list-style-type: circle;}
ul ul ul {list-style-type: square;}
```

如果是这样，实际上的确可能是这样，或者是类似的样式，就必须自己声明样式，覆盖用户代理的样式。仅使用继承的值可能无法满足你的需求。

字符串记号

CSS 还允许创作人员指定字符串值为列表的记号。这样，只要能从键盘上打出来的字符都能用作记号，只要你不介意列表中的每个记号都使用相同的字符串就行。下述样式得到的结果如图 15-3 所示：

```
.list01 {list-style-type: "%";}
.list02 {list-style-type: "Hi! ";}
.list03 {list-style-type: "†";}
.list04 {list-style-type: "⌘";}
.list05 {list-style-type: "● ";}
```

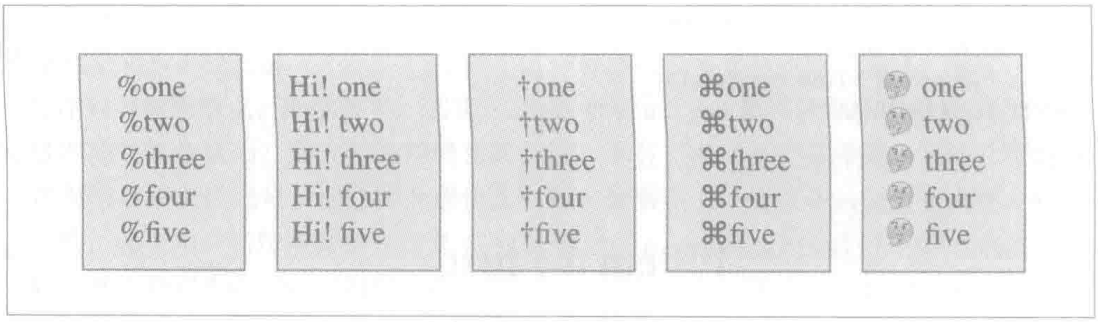


图 15-3：字符串记号示例



截至 2017 年年末，只有 Firefox 族浏览器支持字符串记号。

15.1.2 列表项目图像

有时，常规的文本记号还不够。你可能想使用图像设定记号，这可以利用 `list-style-image` 属性实现。

list-style-image

取值	*<uri>* | *<image>* | none | inherit
初始值	none
适用于	display 属性的值为 list-item 的元素
继承性	是
计算值	<uri> 值计算为绝对 URI；否则为 none

这个属性的用法如下：

```
ul li {list-style-image: url(ohio.gif);}
```

是的，就这么简单。提供一个 url() 值就把记号换成图像了，如图 15-4 所示。

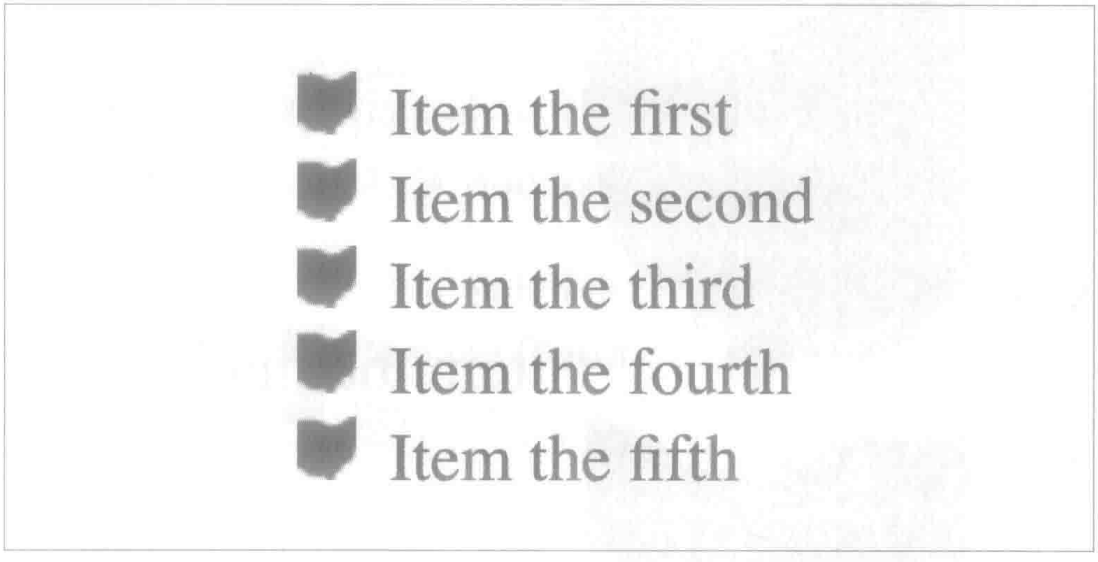


图 15-4：使用图像作为记号

当然，选择图像时要小心，否则就会出现图 15-5 那样难看的结果：

```
ul li {list-style-image: url(big-ohio.gif);}
```

一般来说，最好提供一个后备记号类型，以防图像无法加载、损坏了，或者用户代理无法显示所用的格式。具体方法是，为列表再声明一个 list-style-type 属性：

```
ul li {list-style-image: url(ohio.png); list-style-type: square;}
```

使用 list-style-image 时还可以把值设为默认的 none。这是个好习惯，因为 list-style-image 是继承的，因此嵌套列表也将使用指定的图像作为记号，除非明确设定不这么做：

```
ul {list-style-image: url(ohio.gif); list-style-type: square;}
ul ul {list-style-image: none;}
```

嵌套的列表继承 square 类型的记号，而且设定不使用图像，因此所用的记号为实心方框，如图 15-6 所示。

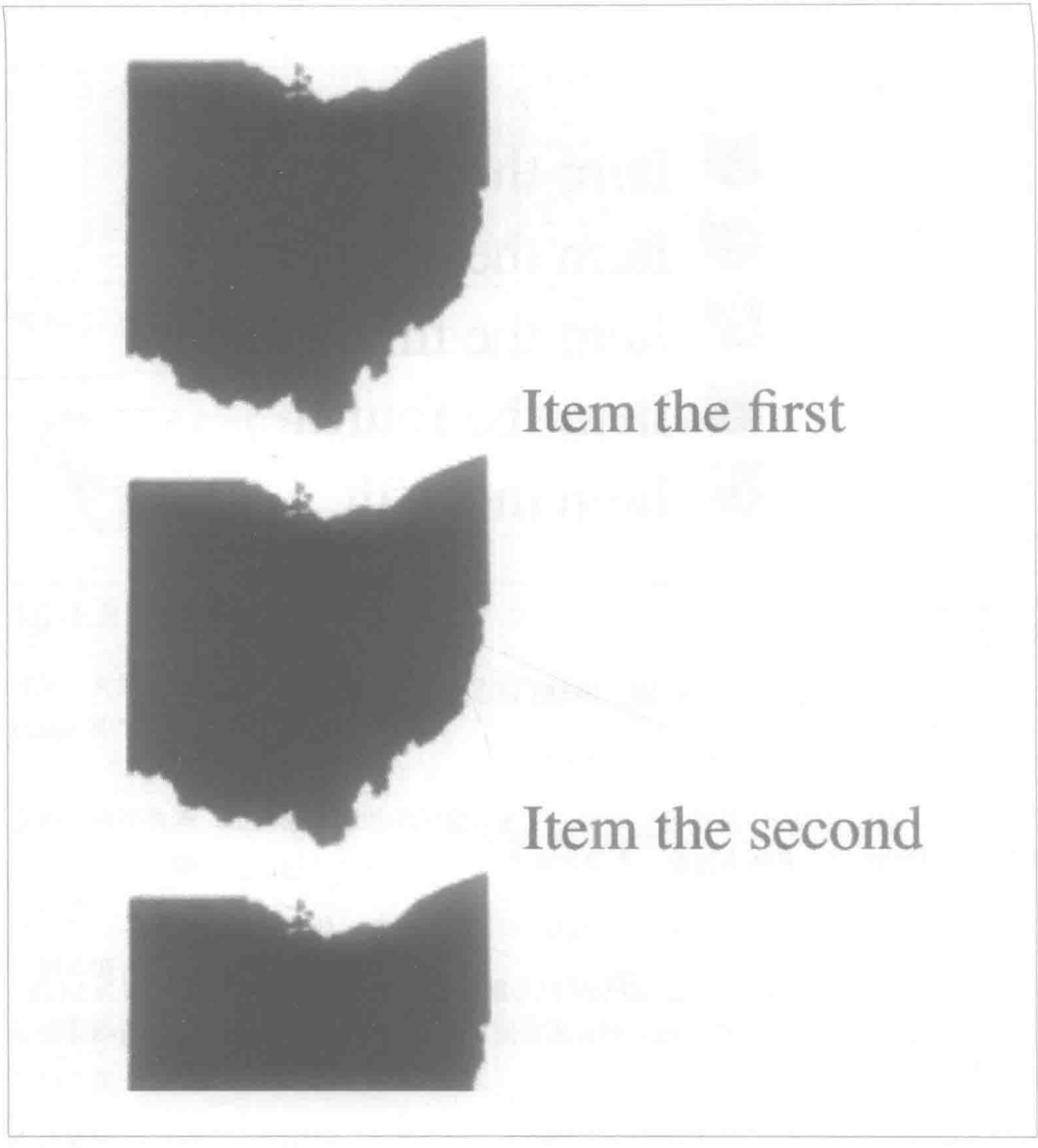


图 15-5：使用特别大的图像作为记号



注意，现实中可能不会得到这样的结果，因为用户代理或许已经为 `ulul` 定义了 `list-style-type`，从而导致 `square` 值不被继承。实际得到的记号可能是空心圆、圆点或其他符号。

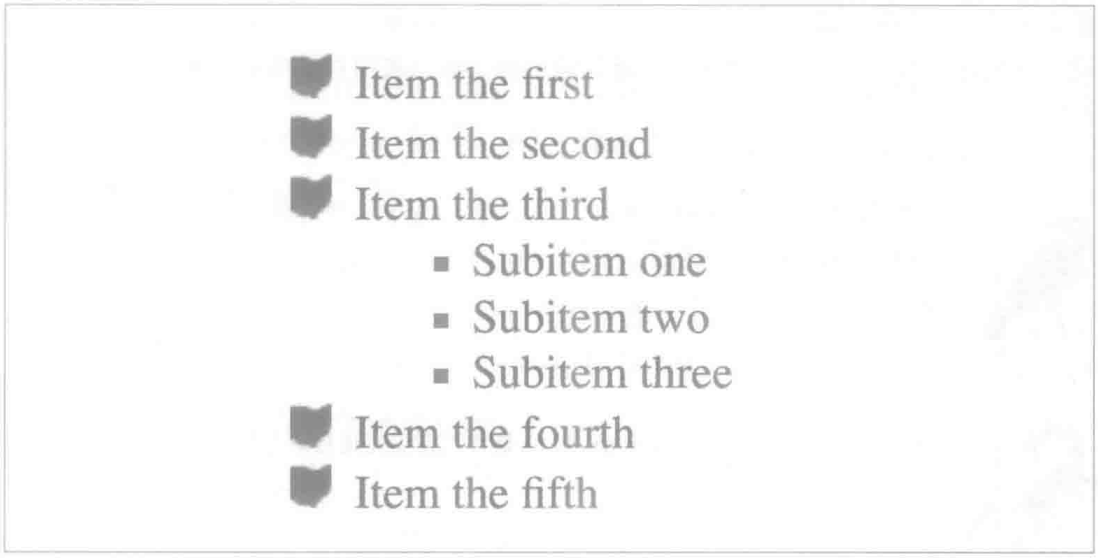


图 15-6：在子列表中禁用图像记号

list-style-image 的值可以是任何图像值，包括渐变图像。因此，下述样式将得到如图 15-7 所示的结果。

```
.list01 {list-style-image:
    radial-gradient(closest-side,
        orange, orange 60%, blue 60%, blue 95%, transparent);}
.list02 {list-style-image:
    linear-gradient(45deg, red, red 50%, orange 50%, orange);}
.list03 {list-style-image:
    repeating-linear-gradient(-45deg, red, red 1px, yellow 1px, yellow 3px);}
.list04 {list-style-image:
    radial-gradient(farthest-side at bottom right,
        lightblue, lightblue 50%, violet, indigo, blue, green,
        yellow, orange, red, lightblue);}
```

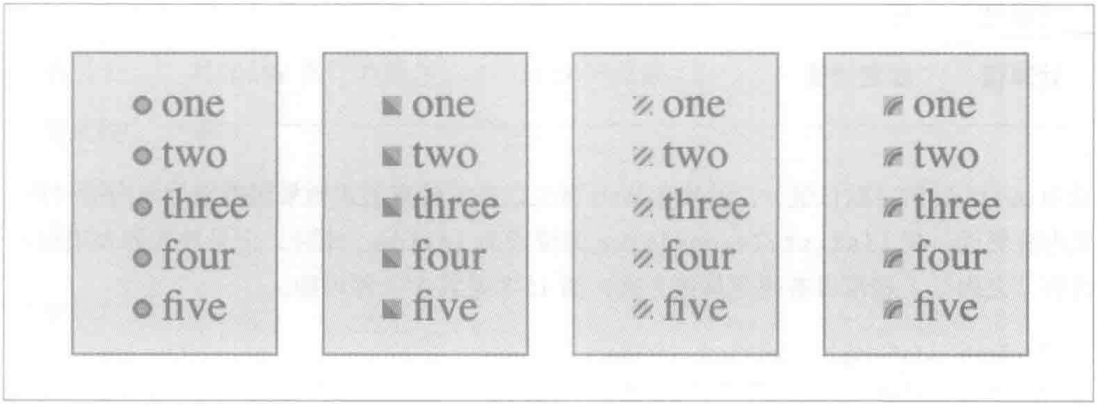


图 15-7：渐变列表记号

渐变记号的一个缺点是尺寸特别小。这个尺寸不受 CSS 的控制，浏览器认为什么尺寸合适就使用什么尺寸。记号的尺寸受字号等的影响，随列表项目的内容而缩放，但也只此而已。



CSS 提供了直接装饰列表记号的方式，即 ::marker 伪元素，但截至 2017 年年初，没有任何用户代理支持。



截至 2017 年年初，只有 WebKit/Blink 族浏览器支持使用渐变图像值作为列表记号。

15.1.3 列表记号的位置

使用 CSS 还可以影响列表项目的一个外观：在列表项目内容的外部还是内部显示记号。记号的位置使用 list-style-position 属性设定。

list-style-position	
取值	inside | outside | inherit
初始值	outside
适用于	display 属性的值为 list-item 的元素
继承性	是
计算值	指定的值

设为 outside 时（默认值），记号在 Web 诞生以来的位置上。如果想把记号向内推一点，离内容更近，把 list-style-position 的值设为 inside。此时，记号放在列表项目的内容“之内”。规范没有规定具体方式，图 15-8 是其中一种可能。

```
li.first {list-style-position: inside;}
li.second {list-style-position: outside;}
```

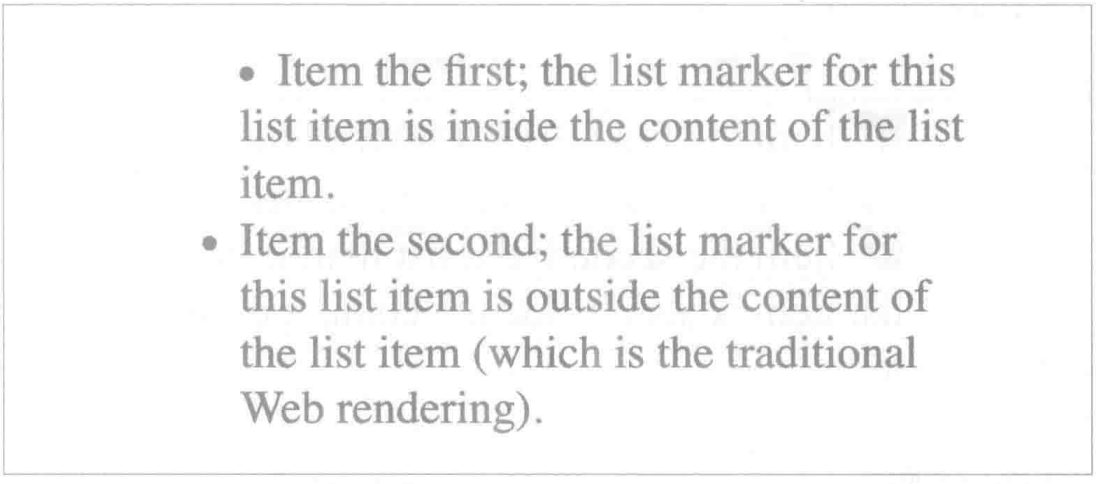


图 15-8：把记号放在列表项目的内部和外部

实际处理中，位置设为 inside 的记号好似一个插在列表项目内容开头的行内元素。但这并不是说记号真的是行内元素，放在内部的记号是无法独立于其他内容直接装饰的，除非把其他内容放在 span 等元素中。只是从布局的角度上看，其表现像是行内元素而已。

15.1.4 列表样式的简写属性

简单起见，前面三个列表样式属性可以合并为一个属性：list-style。

list-style	
取值	[<list-style-type> ‖ <list-style-image> ‖ <list-style-position>] | inherit
初始值	参见各单独属性
适用于	display 属性的值为 list-item 的元素
继承性	是
计算值	参见各单独属性

例如：

```
li {list-style: url(ohio.gif) square inside;}
```

从图 15-9 中可以看到，三个值都应用到列表项目上了。

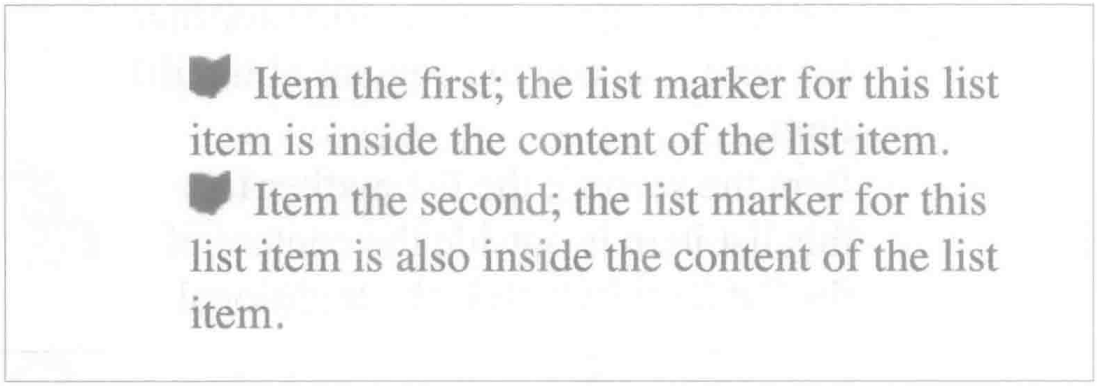


图 15-9：一次设定三个值

list-style 的值可以按任何顺序列出，而且任何一个值都可以省略。只要有一个值，其余的都将使用默认值。例如，下面两个规则得到的结果是一样的：

```
li.norm {list-style: url(img42.gif);}
li.odd {list-style: url(img42.gif) disc outside;} /* 作用相同 */
```

前面的规则也会以同样的方式被覆盖。例如：

```
li {list-style-type: square;}
li {list-style: url(img42.gif);}
li {list-style: url(img42.gif) disc outside;} /* 作用相同 */
```

结果与图 15-9 一样，因为蕴含的 list-style-type 值是 disc，它将把前面声明的 square 覆盖，第三个规则中显式声明的 disc 值也将把它覆盖。

15.1.5 列表的布局

了解过如何装饰列表记号之后，我们来看看列表在不同浏览器中是如何排布的。先看没有标记，也没有放入列表的三个列表项目，如图 15-10 所示。

四周的边框表明，列表项目本质上是类似块级的元素。其实，list-item 值生成的就是块级框。现在，加上记号，如图 15-11 所示。

记号与列表项目内容之间的距离不是 CSS 定义的，CSS 并未提供控制这一距离的方式。



图 15-10：三个列表项目



图 15-11：加上记号

放在列表项目内容外部的记号对其他元素的布局没有影响，而且对列表项目自身的布局也没有影响。记号只是悬挂在内容边界的外侧，与之相隔一定的距离，不管内容的边界在何处，记号始终随之而动。记号好像相对列表项目的内容绝对定位一样，比如 `position:absolute; left: -1.5em;`。当记号在列表项目的内部时，像是放在内容开头的行内元素。

目前，我们还没有考虑真正的列表容器。也就是说，图中没有标出 `ul` 或 `ol` 元素。下面加上，如图 15-12 所示（图中的虚线边框）。



图 15-12：加上列表容器

与列表项目一样，列表元素也是块级框，包含所有后代元素。然而，可以看到，记号不仅在列表元素的内容外部，还在列表元素内容区的外部。列表通常具有的“缩进”还未指定。

截至写作本书时，多数浏览器都会为外层的列表元素设定内边距或外边距，缩进显示列表项目。例如，用户代理可能会应用下面这样的规则：

```
ul, ol {margin-left: 40px;}
```

这是 Internet Explorer 和 Opera 使用的基本规则。而多数基于 Gecko 的浏览器使用类似下面的规则：

```
ul, ol {padding-left: 40px;}
```

不能说这两种做法是错的，但是若想去掉列表项目的缩进，二者的差异可能导致一些问题。图 15-13 展示了两种方式的区别。

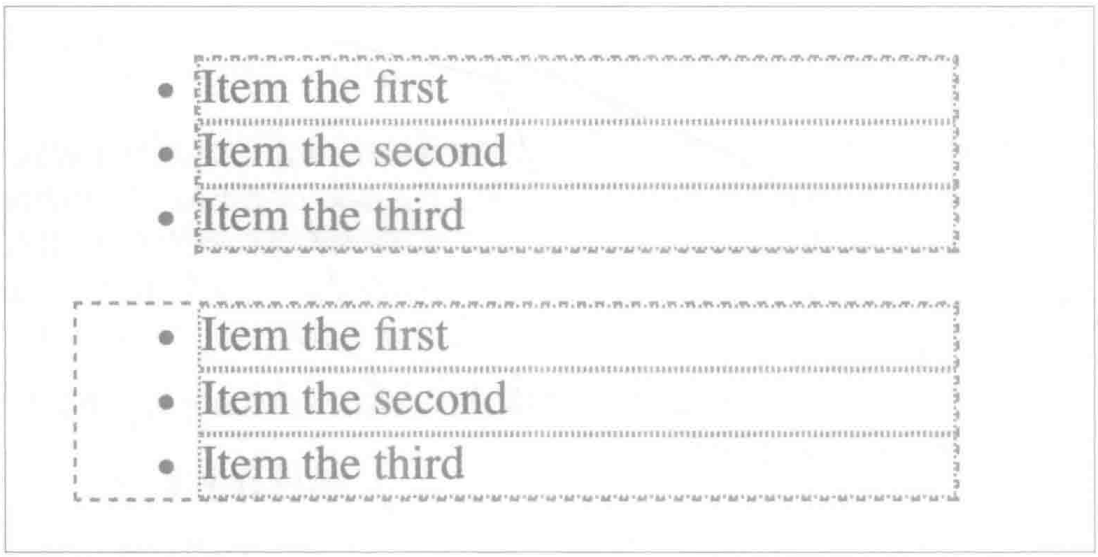


图 15-13：使用外边距和内边距实现的缩进



40px 这个距离是早期 Web 浏览器的遗留，那时缩进列表用的是像素值（块级引用的缩进量也是这么多）。更合适的缩进量可能是 2.5em，因为这样缩进能随字号的变化而变。

若想改变列表的缩进距离，笔者强烈建议同时指定内边距和外边距，从而保证跨浏览器兼容性。例如，如果想使用内边距缩进列表，使用下述规则：

```
ul {margin-left: 0; padding-left: 1em;}
```

如果想用外边距，编写类似下面的规则：

```
ul {margin-left: 1em; padding-left: 0;}
```

不管使用哪种方法，都要记住一点：记号的位置是相对列表项目的内容而定的，因此可能会“悬挂”在文档主体文本之外，甚至超出浏览器窗口的边界。使用特别大的图像或特别长的文本字符串作为列表记号就能看到后果，如图 15-14 所示。

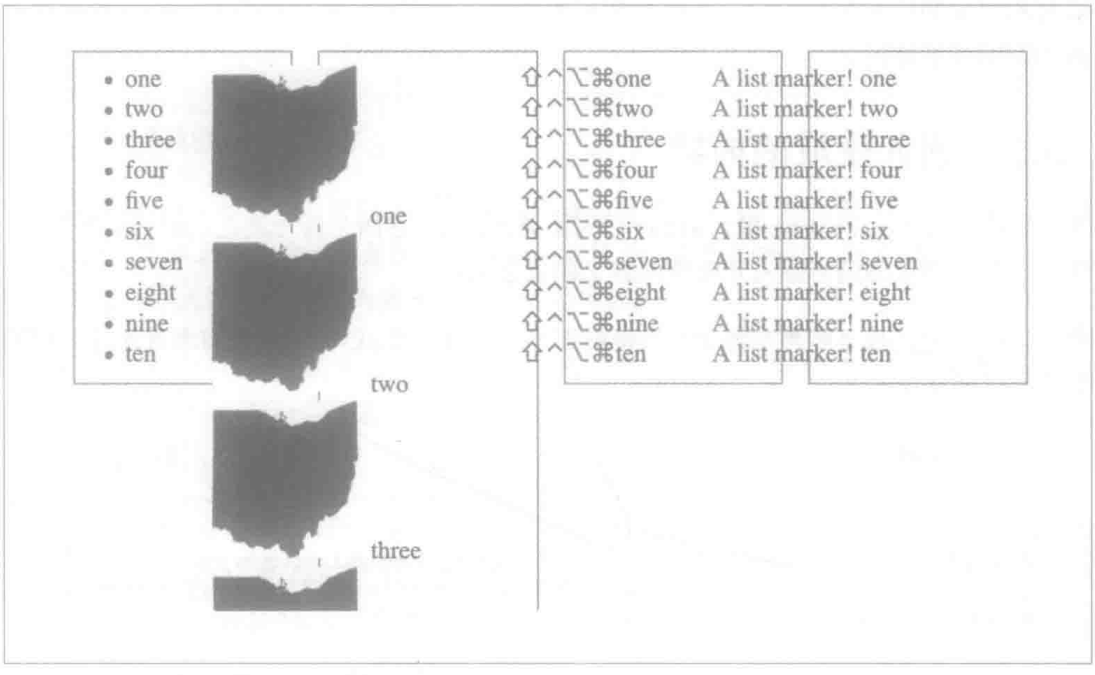


图 15-14：大记号对列表布局的影响

列表记号的位置

很多创作人员都希望能控制记号与列表项目内容之间的距离。CSS2 提供了相关的方式，包括 marker-offset 属性和 display 属性的 marker 值。从具体的实现来看，这些并不是最佳的方式，因此 CSS2.1 把这些特性删除了。

截至 2017 年年初，CSS3 Listsand Counters 模块的草案定义了一种更为简洁的方式，即 ::marker 伪元素。假如在变为推荐标准之前没有修改这个模块，有一天你就可以编写这样的规则：li::marker {margin-right:0.125em; color: goldenrod;}。

15.2 生成的内容

CSS 定义了生成内容的方式。这样的内容通过 CSS 插入，而不通过标记或内容表示。

例如，列表记号就是生成的内容。在列表项目的标记中没有任何内容是表示记号的，而且创作人员也没有在文档的内容中编写记号。记号是由浏览器自动生成的。对无序列表来说，记号是某种符号，例如空心圆、圆点或实心方框。而在有序列表中，记号默认是递增的计数器，每多一个列表项目加一（如前面几节所述，记号还可以换成图像或符号）。

若想知道如何影响列表记号，定制有序列表（或其他元素）的计数方式，必须先学习生成内容的基本知识。

15.2.1 插入生成的内容

生成的内容使用 `::before` 和 `::after` 伪元素插入文档。这两个伪元素把 `content` 属性（参见下一节）指定的内容插入元素的内容之前或之后。

例如，打印文档时可能想在每个超链接前面加上文本“(link)”。这个需求可通过下述规则实现，效果见图 15-15：

```
a[href]::before {content: "(link)";}
```

(link)Jeffrey seems to be (link)very happy about (link)something, although I can't quite work out whether his happiness is over (link)OS X, (link)Chimera, the ability to run the Dock and (link)DragThing at the same time, the latter half of my (link)journal entry from yesterday, or (link)something else entirely.

图 15-15：生成文本内容

注意，生成的内容与元素的内容之间没有空格。这是因为前例中的内容里没有空格。如果想在生成的内容和真正的内容之间加入一个空格，要像下面这样修改声明：

```
a[href]::before {content: "(link) ";}
```

这是个很小的差异，但是却有较大的影响。

类似地，你可能想在指向 PDF 文档的链接末尾插入一个小图标。实现这个需求的规则如下：

```
a.pdf-doc::after {content: url(pdf-doc-icon.gif);}
```

如果想进一步装饰这样的链接，例如在四周添加边框，需要再编写一个规则：

```
a.pdf-doc {border: 1px solid gray;}
```

这两个规则得到的结果如图 15-16 所示。

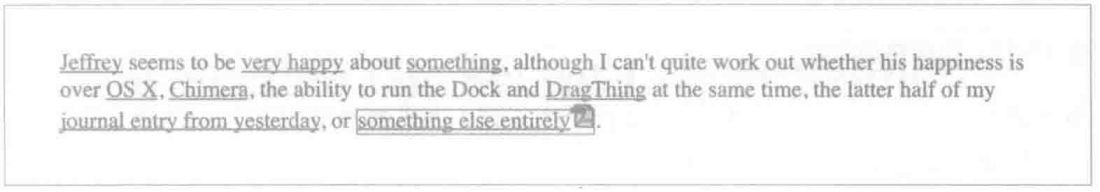


图 15-16：生成图标

注意，链接的边框一直延展到生成的内容周围，就像图 15-15 中链接的下划线会延伸到“(link)”文本下方一样。这是因为生成的内容位于元素框的内部。从 CSS 2.1 开始，除了列表记号，无法把生成的内容放在元素框的外部。

你可能以为通过定位能解决这个问题，但是 CSS2 和 CSS2.1 明确禁止浮动或定位 ::before 和 ::after。列表样式相关的属性，以及表格相关的属性也禁止使用。此外，还有下述限制：

- 如果 ::before 或 ::after 选择符的目标是块级元素，那么 display 属性的值只能为 none、inline、block 或 marker。其他值都当作 block。
- 如果 ::before 或 ::after 选择符的目标是行内元素，那么 display 属性的值只能为 none 或 inline。其他值都当作 inline。

例如：

```
em::after {content: " (!) "; display: block;}
```

因为 em 是行内元素，所以生成的内容不能设为块级。因此，block 值将被重置为 inline。然而，在下述示例中，生成的内容可以设为块级的，因为目标元素也是块级的：

```
h1::before {content: "New Section"; display: block; color: gray;}
```

结果如图 15-17 所示。



图 15-17：生成块级内容

生成的内容比较特别的一点是，它会从依附的元素上继承属性值。因此，对下述规则来说，生成的文本将是绿色，与段落中内容的颜色相同：

```
p {color: green;}
p::before {content: "::: ";}
```

如果想让生成的内容显示为紫色，一个简单的声明就够了：

```
p::before {content: "::: "; color: purple;}
```

当然，只有能被继承的属性才会继承。特别指出这一点是因为这会影响实现某些效果的方式。请看下述规则：

```
h1 {border-top: 3px solid black; padding-top: 0.25em;}
h1::before {content: "New Section"; display: block; color: gray;
  border-bottom: 1px dotted black; margin-bottom: 0.5em;}
```

由于生成的内容放在 h1 的元素框内部，因此生成的文本将出现在元素的上边框下方。此外，生成的文本也在内边距内侧，如图 15-18 所示。

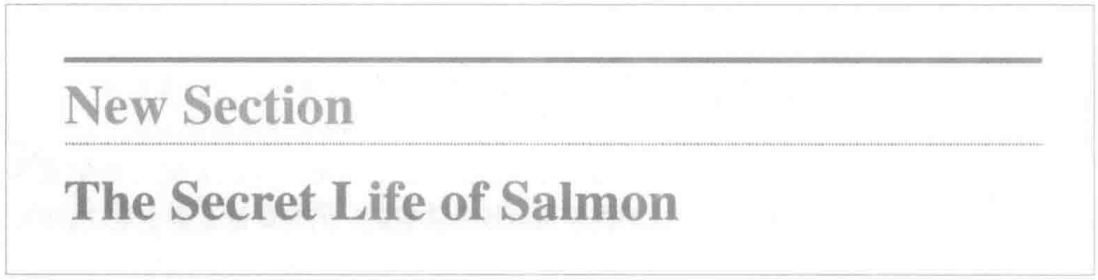


图 15-18：把位置考虑在内

我们为生成的内容（已设为块级）设定了下外边距，因此元素的内容将向下移动 0.5em。不管怎么看，这个示例中生成的内容把 h1 元素分成了两半：生成的内容框和真正的内

容框。这是因为我们为生成的内容声明了 display:block。如果改成 display:inline，结果将变成图 15-19 那样：

```
h1 {border-top: 3px solid black; padding-top: 0.25em;}
h1::before {content: "New Section"; display: inline; color: gray;
  border-bottom: 1px dotted black; margin-bottom: 0.5em;}
```



图 15-19：把生成的内容改成行内框

注意边框的位置，不过内边距还在。其实生成的内容的下外边距也在，只不过因为生成的是行内框，所以外边距对行高没有影响，因此外边距也就没有视觉效果。

掌握生成内容的基础知识之后，接下来学习如何指定生成的内容。

15.2.2 指定内容

为了生成内容，肯定需要一种方式描述要生成什么内容。前面已经见过，生成的内容使用 content 属性指定，但是这个属性的功能比前文所见的要强大很多。

content

取值	normal | [<string> | <uri> | <counter> | attr(<identifier>) | open-quote | close-quote | no-open-quote | no-close-quote]+ | inherit
初始值	normal
适用于	::before 和 ::after 伪元素
继承性	否
计算值	<uri> 值计算为绝对 URI；引用的属性计算为最终字符串；否则为指定的值

前文已经用过字符串和 URI 值，计数器在本章后文介绍。下面先深入讨论一下字符串和 URI，然后再讲 attr() 和引号。

字符串值显示为字面量，即使其中包含某种标记也是如此。因此，下述规则将把字符串原封不动地插入文档，如图 15-20 所示：

```
h2::before {content: "<em>&para;</em> "; color: gray;}
```



图 15-20：字符串原封不动地显示出来

这意味着，如果想在生成的内容中换行，不能使用
。正确的做法是使用字符串 \A，这是 CSS 表示换行符的方式（源自 Unicode 换行符，其十六进制表示为 A）。相反，如果一个较长的字符串值分为多行编写，要使用 \ 字符转义换行符。具体做法如下述规则所示，结果见图 15-21：

```
h2::before {content: "We insert this text before all H2 elements because \
it is a good idea to show how these things work. It may be a bit long \
but the point should be clearly made.  "; color: gray;}
```

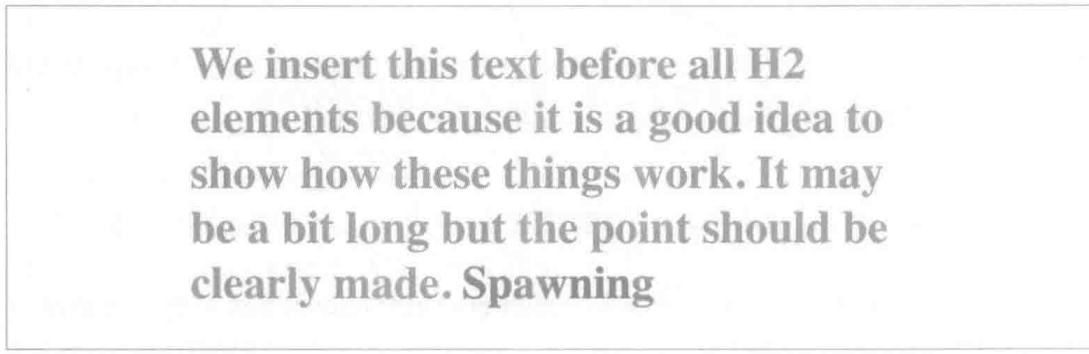


图 15-21：去除插入的字符串中的换行符

此外，还可以使用转义符引用十六进制的 Unicode 值，例如 \00AB。



写作本书时，插入转义的内容（例如 \A 和 \00AB）尚未得到广泛支持，即便是一定程度上支持生成的内容的浏览器也是如此。

值为 URI 时，指向的外部资源（图像、电影、声音片段，或者用户代理支持的其他媒体文件）将插入文档中的恰当位置。如果出于什么原因，用户代理不支持引用的资源，例如在不支持 SVG 的浏览器中插入 SVG 图像，或者在打印的文档中插入一部电影，那么用户代理要完全忽略资源，什么也不插入。

插入属性值

有时，你可能想把元素的属性值显示在文档中。举个简单的例子，可以像下面这样把每个链接的 href 属性值放在链接后面：

```
a[href]::after {content: attr(href);}
```

同样，这会导致生成的内容与真正的内容拥挤在一起。为了解决这个问题，可以在声明中添加一些字符串值，如下所示，结果见图 15-22：

```
a[href]::after {content: " [" attr(href) "]";}
```

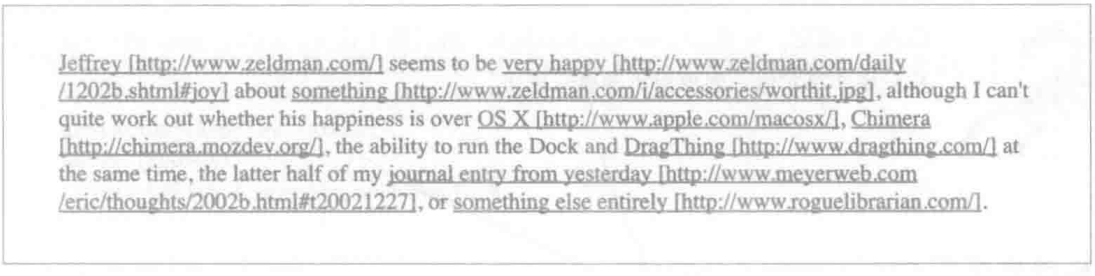


图 15-22：插入 URL

打印文档所用的样式表就可以这样做。任何属性的值都可以作为生成的内容插入文档，例如 alt 文本、class 或 id 的值等。创作人员可以像下面这样把块级引用的引文信息列出来：

```
blockquote::after {content: "(" attr(cite) ")"; display: block;
  text-align: right; font-style: italic;}
```

再复杂一些，还可以把旧文档的文本颜色和链接颜色值显示出来：

```
body::before {content: "Text: " attr(text) " | Link: " attr(link)
  " | Visited: " attr(vlink) " | Active: " attr(alink);
  display: block; padding: 0.33em;
  border: 1px solid; text-align: center; color: red;}
```

注意，如果引用的属性不存在，所在的位置将显示空字符串。图 15-23 就是这样，这是把前面的规则应用到 body 元素没有 alink 属性的文档上得到的结果。

Text: black | Link: blue | Visited: purple | Active:

Amet aliquam eodem bedford. Wisi warrensville heights et modo. Eorum jim lovell james a. garfield facer quarta facit. Berea pierogies nunc clari dynamicus saepius litterarum eodem. Nobis in qui nulla. Odio illum vel dignissim duis ea bobby knight ex independence commodo. Bedford heights henry mancini per claritatem. Don shula laoreet aliquip, parum. Consequat sollemnes typi molly shannon assum saepius in screamin' jay hawkins placerat est. Autem quis sequitur doug dieken bob hope humanitatis

图 15-23：跳过缺少的属性

可以看到，插入文档的文本是“Active:”（包括末尾的空格），但是后面没有内容了。这样就保证了，仅当属性的值存在时才将其插入文档。



CSS2.x 规定，对属性的引用返回的是未经解析的字符串。因此，如果属性的值中有标记或字符实体，将原封不动显示出来。

生成引号

引号是一种特殊形式的生成的内容，CSS2.x 为管理引号及其嵌套行为提供了强大的功能。引号使用 quotes 属性和 open-quote 等值生成。

quotes

取值	[<string> <string>]+ | none | inherit
初始值	各用户代理有所不同
适用于	所有元素
继承性	是
计算值	指定的值

研究取值句法之后你会发现，除了关键字 none 和 inherit 之外，唯一有效的值是一对或多对字符串。字符串对中的第一个字符串定义开始引号，第二个字符串定义结束引号。因此，下面两个声明中只有第一个声明是有效的：

```
quotes: '"' "'";  /* 有效 */

quotes: '"';  /* 无效 */
```

第一个规则还说明了如何在字符串两侧使用字符串的引号：双引号两侧用单引号，单引号两侧则使用双引号。

来看一个简单的例子。假设你在创建一个 XML 格式，用于存储自己喜欢的名言。其中一个条目如下所示：

```
<quotation>
 <quote>I hate quotations.</quote>
 <quotee>Ralph Waldo Emerson</quotee>
</quotation>
```

为了以一种恰当的方式表示数据，可以应用下述规则，得到的结果如图 15-24 所示：

```
quotation {display: block;}
quote {quotes: '"' '"';}
quote::before {content: open-quote;}
quote::after {content: close-quote;}
quotee::before {content: " (";}
quotee::after {content: ")";}
```



图 15-24：插入引号和其他内容

open-quote 和 close-quote 两个值的作用是在恰当的位置插入正确的引号（因为不同的语言使用不同的引号）。插入的引号来自 quotes 属性的值。因此，这里在开头和结尾插入的都是双引号。

quotes 可以定义任意多个嵌套层级使用的引号。例如，英语常见的做法是先用双引号，然后在下一层中使用单引号。“弯”引号可以使用下述规则实现：

```
quotation {display: block;}
```

```
quote {quotes: '\201C' '\201D' '\2018' '\2019';}
quote::before, q::before {content: open-quote;}
quote::after, q::after {content: close-quote;}
```

把这些规则应用到下述标记上得到的结果如图 15-25 所示。

```
<quotation>
 <quote> In the beginning, there was nothing. And God said: <q>Let there
  be light!</q> And there was still nothing, but you could see it.</quote>
</quotation>
```

> “ In the beginning, there was nothing. And God said: ‘Let there be light!’ And there was still nothing, but you could see it.”

图 15-25：嵌套的弯引号

如果引号嵌套的层级比定义的引号对多，后续的层级都使用最后一对引号。因此，如果把下述规则应用到图 15-25 中的标记上，内层使用的引号将与外层一样：

```
quote {quotes: '\201C' '\201D';}
```



上述规则使用的是“弯引号”的十六进制 Unicode 码位。如果你编写的 CSS 使用 UTF-8 字符编码（确实应该这么做），那么就可以不像前面的示例那样转义十六进制码位，而是直接输入弯引号字符。

使用生成的引号还能实现另一种常见的排版效果。如果引用的文本分为多段，每一段结尾处的引号通常不显示（最后一段除外），只显示段落开头的引号。这个效果可以使用 `no-close-quote` 实现：

```
blockquote {quotes: '"' '"' "'" "'" '"' '"';}
blockquote p::before {content: open-quote;}
blockquote p::after {content: no-close-quote;}
```

此时，每一段的开头有个双引号，但是结尾处没有引号。最后一段也是如此，因此如果想为最后一段添加结束引号，要为最后一段设置类，然后把 `::after` 伪元素的内容设为 `close-quote`。

这个值的重要作用是在不生成引号的基础上减少引用的嵌套层级。正因为如此，每一段的开头才都显示双引号，而不是交替使用单双引号，直到抵达第三段再使用双引号。no-close-quote 在每一段的结尾处结束引用嵌套，因此每一段的开头都位于同一嵌套层级。

这一点尤为重要，因为 CSS2.1 规范指出，“引用层深与源文档或格式化结构的嵌套层级无关。”也就是说，一旦一个引用层级开始，直至遇到声明了 close-quote 的元素才结束，此时嵌套引用减少一层。

对应地，还有 no-open-quote 关键字，其效果与 no-close-quote 是相反的。这个关键字增加一层嵌套层级，但不生成引号。

15.2.3 计数器

我们对计数器都不陌生，例如，有序列表中列表项目的记号就是计数器。在 CSS1 中，没有办法修改计数器，因为没有这个需要：HTML 为有序列表定义了计数行为，无需多此一举。XML 兴起之后，却需要有一种方法用于定义计数器。然而，CSS2 提供的计数行为不像 HTML 那么简单。两个新属性和 content 的两个新值能实现多数计数格式，包括由多个样式实现的小节计数，例如“VII.2.c”。

重置和增量

创建计数器的基本过程是先设置计数器的起点，然后增加一定的量。前者使用 counter-reset 属性设置。

counter-reset	
取值	[<identifier> <integer>?]+ | none | inherit
初始值	各用户代理有所不同
适用于	所有元素
继承性	否
计算值	指定的值

计数器标识符是由创作人员创建的标注。例如，可以把小节计数器命名为 subsection、subsec、ss 或 bob。重置（或递增）标识符就能创建标识符。在下述规则中，我们通过重置创建 chapter 计数器：

```
h1 {counter-reset: chapter;}
```

默认情况下，计数器重置为零。如果想重置为其他数，在标识符后声明：

```
h1#ch4 {counter-reset: Chapter 4;}
```

使用多个标识符和整数对还可以一次性重置多个标识符。如果缺少整数，默认为零：

```
h1 {counter-reset: Chapter 4 section -1 subsec figure 1;}
   /* 'subsec' 重置为 0 */
```

从前例可以看出，负值是允许的。把计数器设为 -32768，然后从那里开始计数是完全可行的。



CSS 没有规定用户代理应该如何处理采用非数字计数格式时的负数值。例如，没有规定显示格式为 upper-alpha 时如何处理计数器中的 -5 值。

为了计数，需要通过一个属性指明通过哪个元素递增计数器。如若不然，计数器的值始终不变，一直是 counter-reset 声明指定的值。我们要使用的属性是 counter-increment。

counter-increment

取值	[<identifier> <integer>?]+ | none | inherit
初始值	各用户代理有所不同
适用于	所有元素
继承性	否
计算值	指定的值

与 counter-reset 一样，counter-increment 的值为标识符和整数对，而且整数部分可以是 0、负数或正数。两个属性的不同之处是，在 counter-increment 中省略的整数部分默认为 1，而不是 0。

举个例子。用户代理为有序列表创建的自然数计数器可能是这样定义的：

```
ol {counter-reset: ordered;} /* 默认为 0*/
ol li {counter-increment: ordered;}  /* 默认为 1*/
```

反过来，创作人员可能会想从零向后计数，让列表项目的记号使用负数。在前例的基础上只需稍微修改一下：

```
ol {counter-reset: ordered;} /* 默认为 0*/
ol li {counter-increment: ordered -1;}
```

现在列表的序号将是 -1、-2、-3 等。如果把整数 -1 改成 -2，那么列表的序号将是 -2、-4、-6 等。

使用计数器

不过，为了真正显示计数器，需要使用 content 属性及与计数器有关的值。下面通过一个基于 XML 的有序列表说明具体方法：

```
<list type="ordered">
 <item>First item</item>
 <item>Item two</item>
 <item>The third item</item>
</list>
```

把下述规则应用到这段 XML 结构上，将得到如图 15-26 所示的结果：

```
list[type="ordered"] {counter-reset: ordered;}  /* 默认为 0 */
list[type="ordered"] item {display: block;}
list[type="ordered"] item::before {counter-increment: ordered;
     content: counter(ordered) ". "; margin: 0.25em 0;}
```

1. First item
2. Item two
3. The third item

图 15-26：统计项目数量

注意，生成的内容跟之前一样以行内内容的形式显示在相应元素的开头。因此，上述规则的效果类似于在 HTML 列表上声明 list-style-position:inside;。

另外注意，item 元素是生成块级框的常规元素，这意味着计数器并不限于只能应用在 display 属性的值为 list-item 的元素上。其实，任何元素都能使用计数器。请看下述规则：

```
h1 {counter-reset: section subsec;
    counter-increment: chapter;}
h1::before {content: counter(chapter) ". ";}
h2 {counter-reset: subsec;
    counter-increment: section;}
h2::before {content: counter(chapter )"." counter(section) ". ";}
h3 {counter-increment: subsec;}
h3::before {content: counter(chapter) "." counter(section) "."
        counter(subsec) ". ";}
```

这些规则的效果如图 15-27 所示。

图 15-27 体现了计数器重置和增量相关的一些重要方面。例如，计数器在元素上重置，而生成的计数器则通过 ::before 伪元素插入文档。在伪元素上重置计数器是没有效果的，得到的将是大量零。增量可以在元素上设置，也可以在伪元素上设置，就看你怎么选择了。

此外请注意，h1 元素使用的计数器是 chapter，其默认值为零，因此在元素的文本前面显示的是“1.”。在相同的元素上递增并使用计数器时，计数器在显示之前完成递增。类似地，在相同的元素上重置并显示计数器时，计数器在显示之前完成重置。请看下述规则：

```
h1::before, h2::before, h3::before {
  content: counter(chapter) "." counter(section) "." counter(subsec) ". ";}
h1 {counter-reset: section subsec;
  counter-increment: chapter;}
```

文档中的第一个 h1 元素前面显示的文本是“1.0.0.”，因为 section 和 subsec 计数器重置了，但是没有递增。这意味着，如果想在首次显示计数器时显示为 0，要像下面这样把计数器重置为 -1：

```
body {counter-reset: chapter -1;}
h1::before {counter-increment: chapter; content: counter(chapter) ". ";}
```

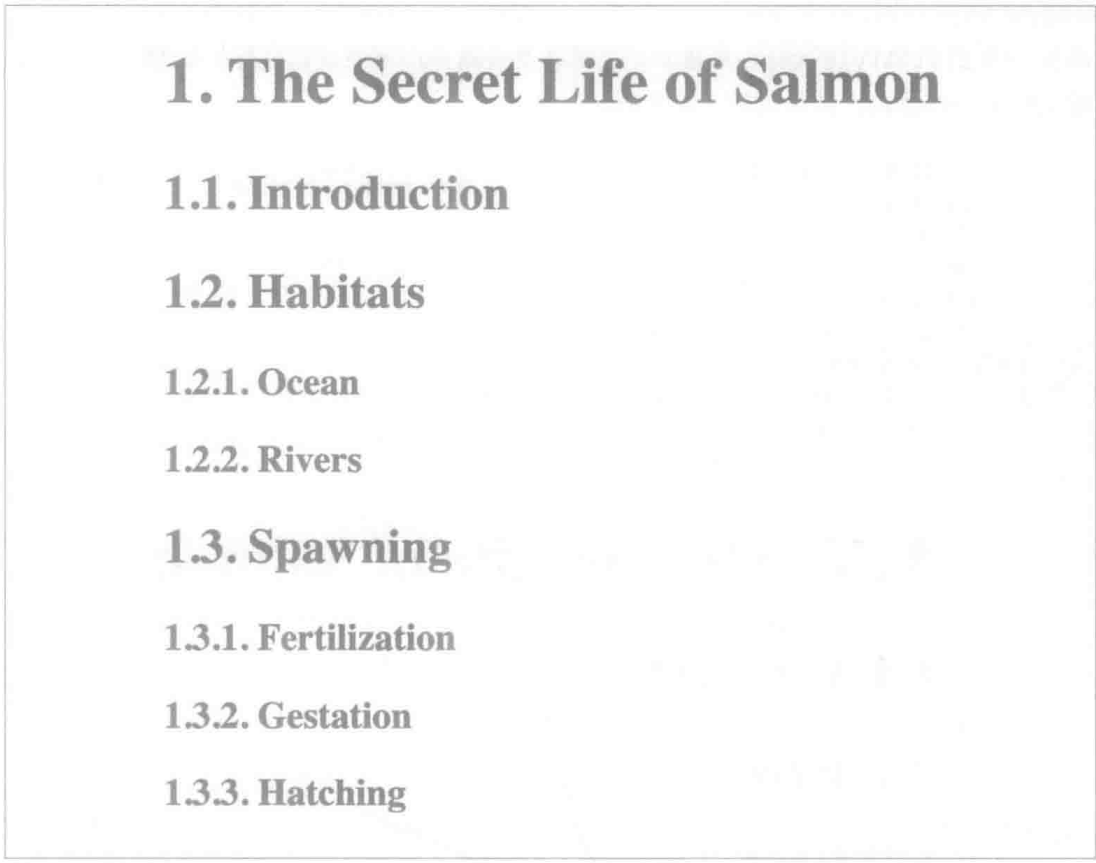


图 15-27：把计数器添加到标题中

使用计数器可以实现一些有趣的效果。以下面的 XML 片段为例：

```
<code type="BASIC">
 <line>PRINT "Hello world!"</line>
 <line>REM This is what the kids are calling a "comment"</line>
 <line>GOTO 10</line>
</code>
```

使用下述规则可以重现 BASIC 程序清单的传统格式：

```
code[type="BASIC"] {counter-reset: linenum; font-family: monospace;}
code[type="BASIC"] line {display: block;}
code[type="BASIC"] line::before {counter-increment: linenum;
  content: counter(linenum 10) ": ";}
```

在 counter() 中还可以为每个计数器定义不同的格式。方法是，在计数器的标识符后面添加一个 list-style-type 关键字，之间以逗号隔开。下述规则在标题计数器示例的基础上做了一些改动，结果如图 15-28 所示。

```
h1 {counter-reset: section subsec;
    counter-increment: chapter;}
h1::before {content: counter(chapter,upper-alpha) ". ";}
h2 {counter-reset: subsec;
    counter-increment: section;}
h2::before {content: counter(chapter,upper-alpha)"." counter(section) ". ";}
h3 {counter-increment: subsec;}
h3::before {content: counter(chapter,upper-alpha) "." counter(section) "."
        counter(subsec,lower-roman) ". ";}
```

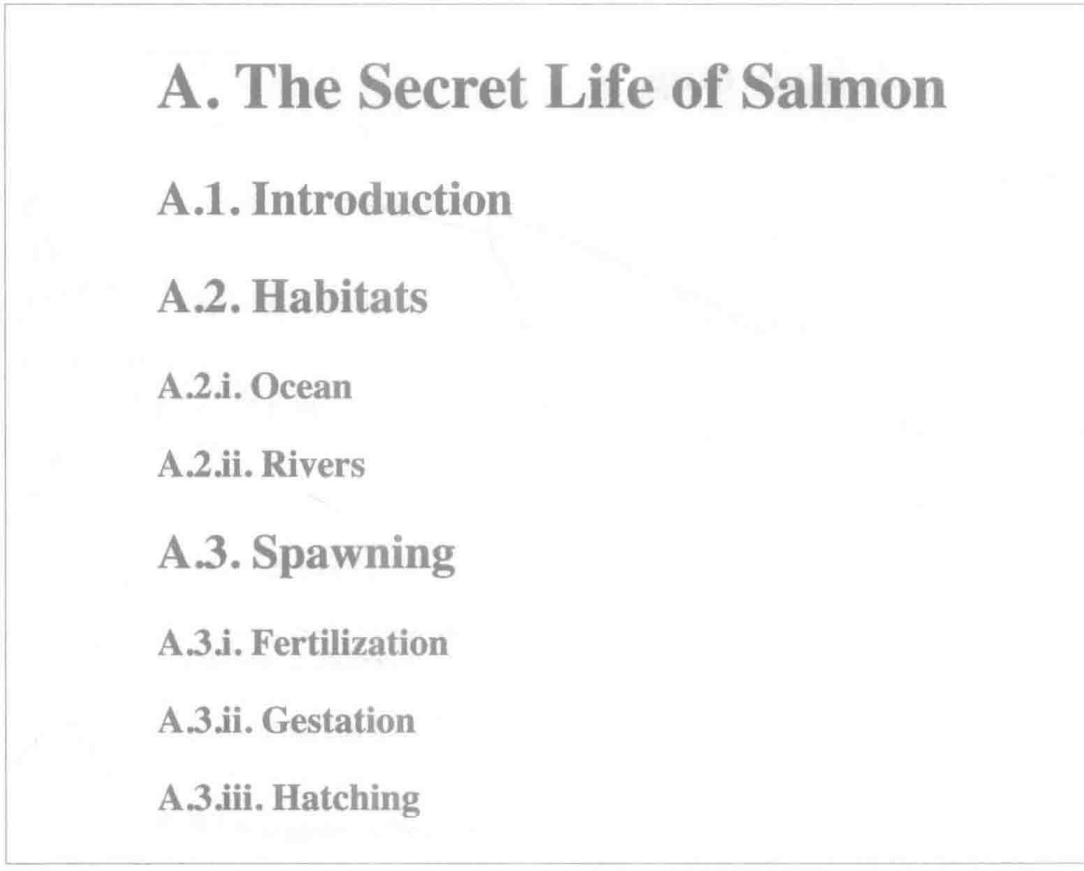


图 15-28：修改计数器的格式

注意，我们没有为 section 计数器指定格式关键字，因此默认为十进制计数。如果愿意，甚至可以设置计数器使用 disc、circle、square 和 none 等格式。

另外需要注意一点的是，display 属性的值为 none 的元素不递增计数器，即便样式规则看似不是这样。相比之下，visibility 属性的值为 hidden 的元素依然递增计数器：

```
.suppress {counter-increment: cntr; display: none;}
  /* 'cntr' 不递增 */
.invisible {counter-increment: cntr; visibility: hidden;}
  /* 'cntr' 递增 */
```

计数器的作用域

目前，我们知道如何通过多个计数器统计节和小节的数量了。通常，遇到嵌套的有序列表时我们也会这么做，然而一旦嵌套层级变深，创建大量计数器会让人不胜其烦。涵盖五级嵌套列表的规则可能是下面这样的：

```
ol ol ol ol ol li::before {
    counter-increment: ord1 ord2 ord3 ord4 ord5;
    content: counter(ord1) "." counter(ord2) "." counter(ord3) "."
        counter(ord4) "." counter(ord5) ".";}
```

试想涵盖 50 层嵌套的规则将是什么样子（笔者不是说你应该创建嵌套 50 层的有序列表，只是举个例子而已）。

幸好，CSS2.x 为计数器定义了作用域（scope）。简单来说，每一层嵌套都为相应的计数器创建一个新作用域。利用作用域，可以使用下述规则涵盖 HTML 处理有序列表的常规方式：

```
ol {counter-reset: ordered;}
ol li::before {counter-increment: ordered; content: counter(ordered) ". ";}
```

使用上述规则，任何有序列表，即便是嵌套的，都将从 1 开始计数，而且每个列表项目递增 1，这正是 HTML 一直以来处理有序列表的方式。

之所以这样，是因为每层嵌套都会新建一个 ordered 计数器。因此，在第一层有序列表中，有个 ordered 计数器。后面每嵌套一层，都会出现一个新的 ordered 计数器，并且从头开始计数。

然而，如果希望嵌套的新计数器追加到现有的计数器上，实现 1、1.1、1.2、1.2.1、1.2.2、1.3、2、2.1 等效果，该怎么做呢？此时使用 counter() 是无法实现呢，但是可以使用 counters()。多了个“s”，作用就变了。

图 15-29 中所示的嵌套计数器格式是用下述规则实现的：

```
ol {counter-reset: ordered; list-style: none;}
ol li:before {content: counters(ordered,".") ": "; counter-increment: ordered;}
```

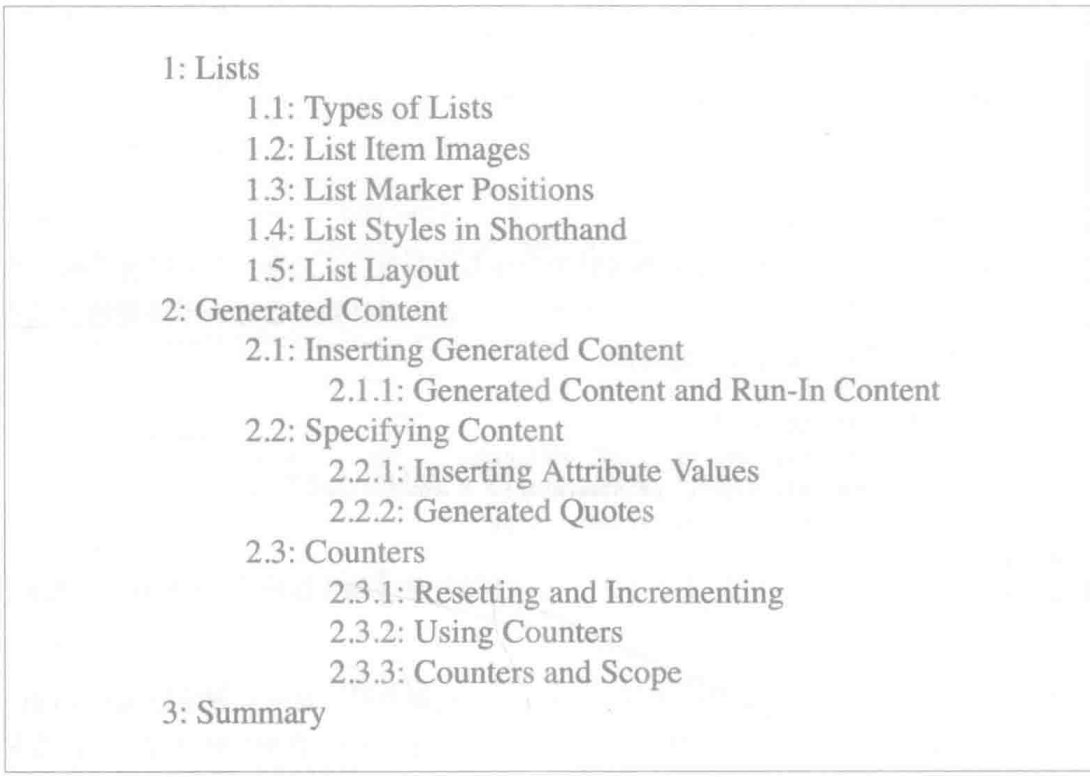


图 15-29：嵌套的计数器

关键字 `counters(ordered,".")` 显示每个作用域中的 `ordered` 计数器，并在后面加上一个点号，然后把指定元素上所有作用域中的计数器串在一起。因此，在第三层列表中，列表项目的编号先从最外层列表的作用域中获取 `ordered` 计数器的值，然后从最外层和当前层之间那一层列表的作用域中获取 `ordered` 值，再从当前列表的作用域中获取 `ordered` 值，而且在每个值后面都加上点号。`content` 值中余下的内容添加一个空格、一个连字符，并在全部计数器的最后添加一个空格。

与 `counter()` 一样，也可以为嵌套的计数器定义列表样式，不过指定的列表样式将用于全部计数器。因此，如果把前面的 CSS 改成下面这样，图 15-29 中的列表项目将全部使用小写字母计数：

```
ol li::before {counter-increment: ordered;
    content: counters(ordered,".",lower-alpha) ": ";}
```

你可能注意到了，在前面的例子中，我们为 ol 元素声明了 list-style:none。这是因为，插入的计数器是生成的内容，不能替代列表记号。也就是说，如果没有 list-style: none，每个列表项目前都将显示用户代理提供的计数器，以及我们通过生成的内容插入的计数器。

这个功能特别有用，不过有时我们只是想定义新的记号。这样的需求要使用计数模式。

15.3 定义计数模式

近些年，CSS 中出现了一种定义计数模式的新方法。这个方法使用 @counter-style 块定义一些专门的描述符，控制得到的结果。一般的句法如下：

```
@counter-style <name> {
    ...declarations...
}
```

其中，*<name>* 是创作人员为模式起的名称。例如，若想交替显示三角形记号，@counter-style 块可以这样编写：

```
@counter-style triangles {
    system: cyclic;
    symbols: ▶ ▷;
}
```

使用效果如图 15-30 所示。

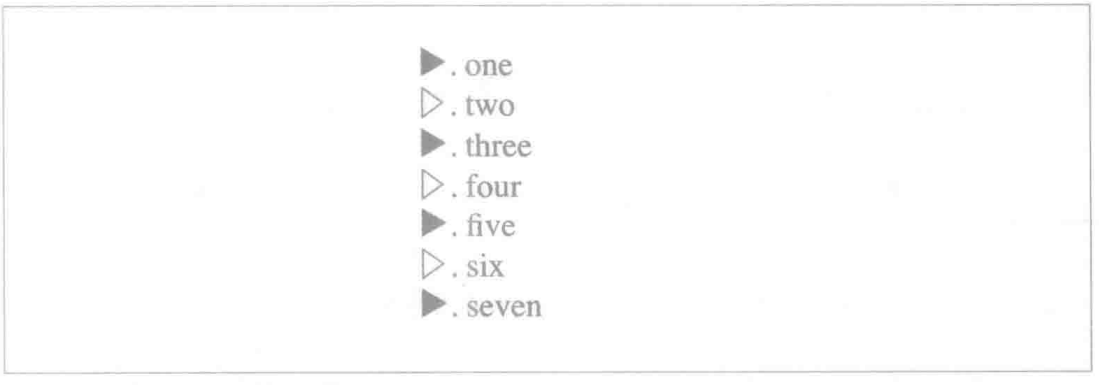


图 15-30：一个简单的计数器模式



截至2017年年初，只有Firefox族浏览器支持本节讨论的@counter-style和相关的特性。这个功能很好，但是不要过于依赖，即，如果计数器是使用@counter-style生成的，不要说“参见步骤1A”。

@counter-style块中可以使用多个描述符，概述如下。

@counter-style中可用的描述符

system	定义要使用的计数器模式系统。可以用的值有fixed、cyclic、alphabetic、numeric、symbolic、additive和extends
symbols	定义要在计数器模式中使用的符号。除additive和extends记号系统之外，其他系统都需要这个描述符
additive-symbols	定义在additive计数器模式中使用的符号
prefix	定义放在模式中各计数器之前的字符串或符号
suffix	定义放在模式中各计数器之后的字符串或符号
negative	定义放在计数器中负值两侧的字符串或符号
range	定义应用计数器模式的范围值。在指定范围之外的计数器使用后备计数器格式
fallback	定义无法使用主计数器模式表示或者值在range定义的范围之外时使用的计数器模式
pad	定义模式中所有计数器至少有几个字符，缺少的位使用指定的一个或一系列符号填充
speak-as	定义计数器在文字转语音系统中的发音策略

我们将由简到繁介绍各个系统，不过在此之前先来看最简单的两个描述符的准确定义：system和symbols。

system 描述符

取值	cyclic | numeric | alphabetic | symbolic | additive | [fixed <*integer*>?] | [extends <*counter-style-name*>]
初始值	Symbolic

symbols 描述符

取值	<*symbol*>+
初始值	n/a
备注	<*symbol*> 可以是任何兼容 Unicode 的字符串、图像引用或转义的十六进制引用等标识符

基本上每个 `@counter-style` 块至少都有这两个描述符。如果定义的是 `symbolic` 系统，可以省略 `system` 描述符，不过一般最好写上，以便知道设置的是哪种系统。记住，以后接手样式的人可能不像你对计数器样式那么熟悉！

15.3.1 固定计数模式

`fixed` 系统是最简单的计数器模式。固定系统指定义的计数器记号数量有限，用完后不重复。请看下面的例子，结果如图 15-31 所示。

```
@counter-style emoji {
    system: fixed;
    symbols: 😁 😠 😭 🤔 😱;
}
ul.emoji {list-style: emoji;}
```

一旦超过五个列表项目，这个计数器系统定义的表情符号就用完了，因为没有定义后备系统（稍后讨论），所以后续列表项目将使用无序列表的默认记号。

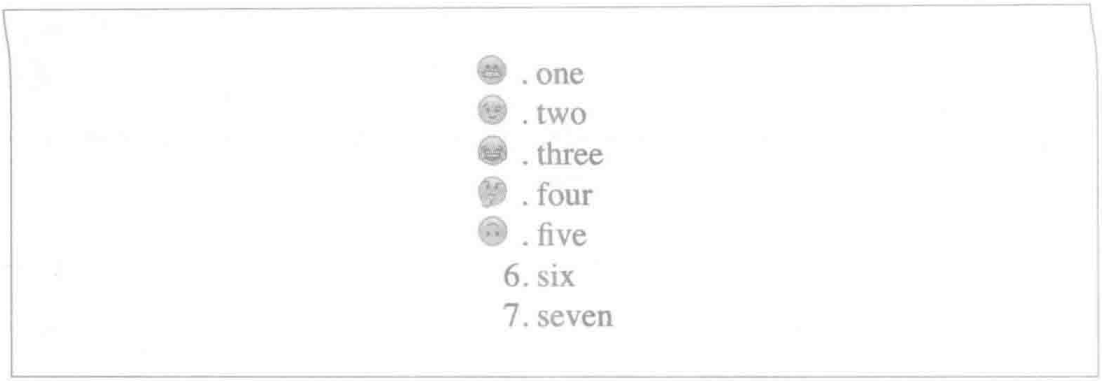


图 15-31：固定计数器模式

注意，symbols 描述符中的符号是以空格分开的。如果都挤在一起，中间没有空格，如下所示，那么得到的结果将变成图 15-32 那样：

```
@counter-style emoji {
    system: fixed;
    symbols: 😁 😉 😂 🤔 🙃;
}
ul.emoji {list-style: emoji;}
```

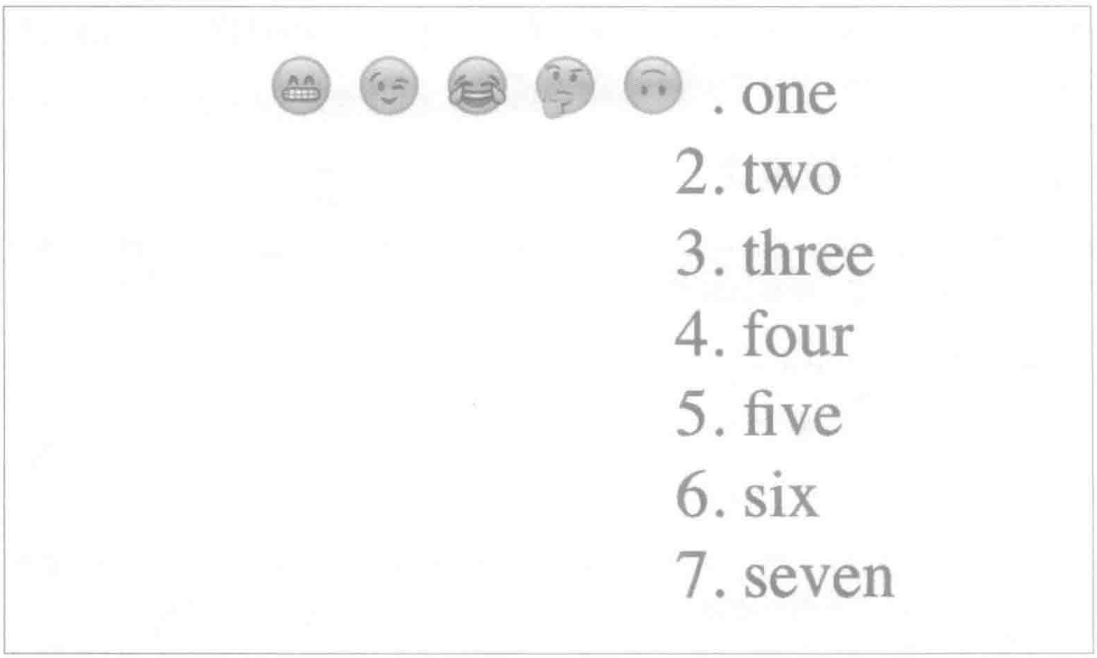


图 15-32：符号紧挨在一起

不过这却表明，单个记号可以由多个符号组成（如果你想定义由多种模式结合起来实现的计数系统，别急，马上就会讲到）。

如果想使用 ASCII 符号作为记号，一般建议把符号放在引号中。这样能避免一些问题，例如防止尖括号被错误解析为 HTML 片段。因此，正确的做法是：

```
@counter-style emoji {
    system: fixed;
    symbols: # $ % ">";
}
```

把所有符号都放在引号中也可以，而且最好养成这个习惯。这样按键的次数多了（上述值要写成 "#" "$" "%" ">"），但是出错的几率却小了。

使用固定计数器系统时，可以在 `system` 描述符中定义起始值。假如想从 5 开始计数，要这么写：

```
@counter-style emoji {
    system: fixed 5;
    symbols: 😀 😁 😂 😃 😄;
}
ul.emoji {list-style: emoji;}
```

此时，前五个符号表示计数器中第 5 个到第 9 个数。如果后备计数格式为十进制数，那么计数器中的第 6 个数对应的值是 10（在大写的罗马字母中，对应的是“J”）。



设置起始数的功能在其他计数器系统中都不可用。

15.3.2 循环计数模式

固定模式之后是 `cyclic` 模式。这个模式其实是循环的固定模式。下面把前一节的表情符号固定模式改为循环模式，结果如图 15-33 所示：

```
@counter-style emojiverse {
    system: cyclic;
    symbols: 😀 😁 😂 😃 😄;
```

```
}
ul.emoji {list-style: emojiverse;}
```

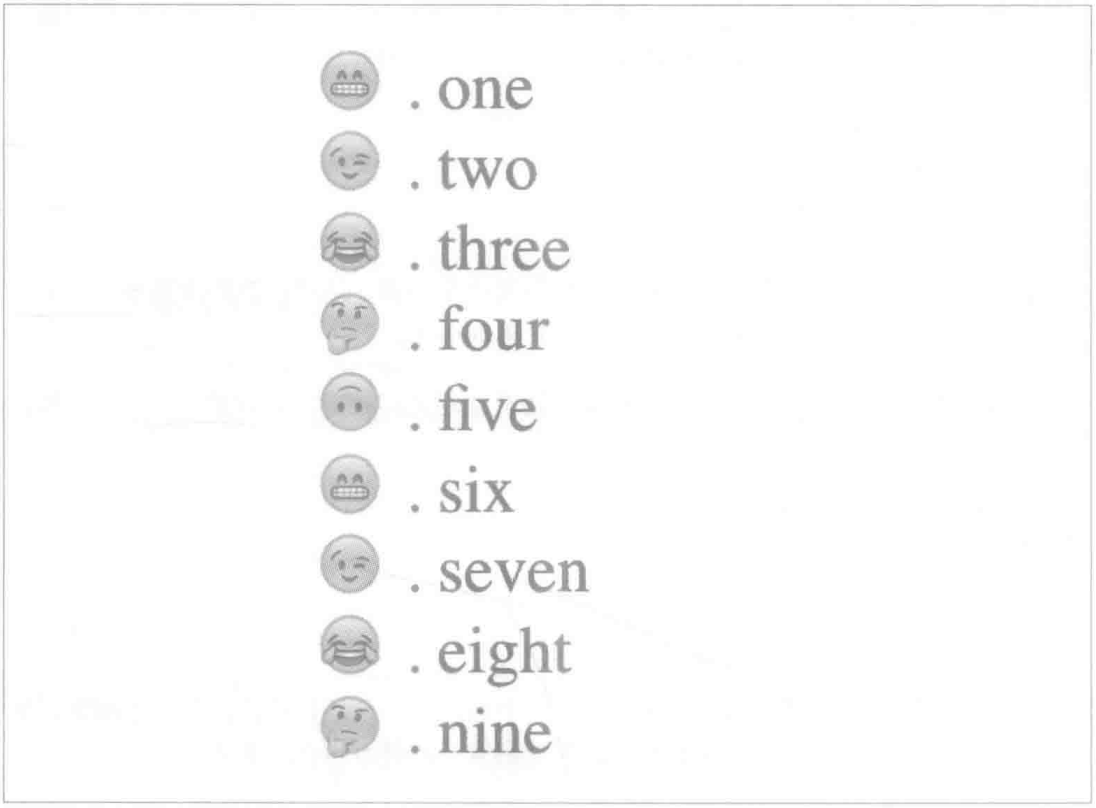


图 15-33：循环计数器模式

定义的符号按顺序使用，一次又一次，直到计数序列中没有记号为止。

循环模式可以只用一个记号，这与为 list-style-type 提供一个字符串的效果差不多。此时，规则的写法如下：

```
@counter-style thinker {
    system: cyclic;
    symbols: 🤔;
    /* 等效于 list-style-type: "🤔" */
}

ul.emoji {list-style: thinker;}
```

你可能注意到了，目前所有的计数器后面都有个句点。这是因为 suffix 描述符的默认值就是句点。suffix 描述符有个同类，prefix。

prefix 和 suffix 描述符

取值	*<symbol>*
初始值	prefix 是 ""（空字符串），suffix 是 \2E（句点，“.”）
备注	<symbol> 可以是任何兼容 Unicode 的字符串、图像引用或转义的十六进制引用等标识符

使用这两个描述符可以定义插在每个记号前面和后面的符号。因此，可以像下面这样为思考表情加上 ASCII 格式的翅膀，如图 15-34 所示：

```
@counter-style wingthinker {
    system: cyclic;
    symbols: 🤔;
    prefix: "~";
    suffix: "~";
}

ul.hmmm {list-style: wingthinker;}
```



图 15-34：为思考表情加上“翅膀”

如果想把记号的默认后缀去掉，就需要使用 suffix 描述符。下面举个例子说明做法：

```
@counter-style thisisfine {
    system: cyclic;
    symbols: 🔥 💀 ☕ 🔥;
    suffix: "";
}
```

当然，你还可以自由发挥，使用 prefix 和 suffix 实现其他有趣的效果，如图 15-35 所示：

```
@counter-style thisisfine {
    system: cyclic;
    symbols: 🔥 💀 ☕ 🔥;
    prefix: "🔥";
    suffix: 🔥;
}
```

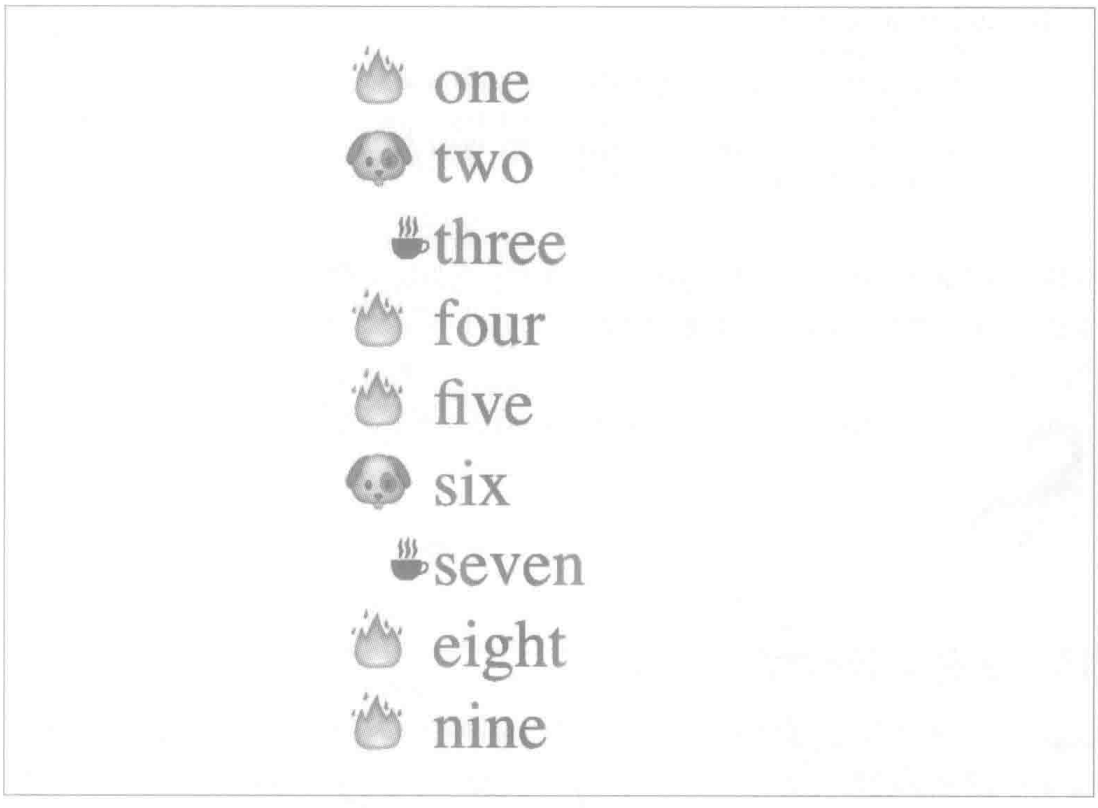


图 15-35：这个列表很好看

你可能觉得奇怪，在前面的示例中，为什么 prefix 的值放在引号里，而 suffix 的值没有？其实没什么特别的原因，我只是想告诉你两种做法都可以。前面说过，把符号放在引号里更安全，但不是必须的。

你可能还发现了上述 CSS 示例有些特别的地方，在图中看得更明显一些。这是使用标签符号和其他类似的字符不可避免的后果，在一个人使用的用户代理中看到的效果可能与另一个人的不同。比如说，MacOS、iOS、安卓、三星、Windows 桌面系统、Windows 移动系统、Linux 系统等，渲染表情符号得到的结果都不同。

别忘了，计数器还可以使用图像，至少理论上如此。举个例子，假设你想使用克林贡字形，而它们没有对应的 Unicode 码位（克林贡语加入 Unicode 家族的过程充满崎岖。1997 年提出动议，2001 年被拒。2016 年又提出了新的动议，但是写作本书时还未裁决）。我们不打算把全部克林贡符号都表示出来，而是从中选择几个：

```
@counter-style klingon-letters {
    system: cyclic;
    symbols: url(i/klingon-a.svg) url(i/klingon-b.svg)
        url(i/klingon-ch.svg) url(i/klingon-d.svg)
        url(i/klingon-e.svg) url(i/klingon-gh.svg);
    suffix: url(i/klingon-full-stop.svg);
}
```

计数器将从 A 到 GH，然后重复。虽然不多，但至少我们把一些克林贡语符号显示出来了。本章后文将说明如何构建字母和数字计数系统。



截至 2017 年年初，几乎没有浏览器支持使用图像作为计数符号。

15.3.3 符号计数模式

`symbolic` 计数系统与循环系统类似，不过在符号系统中，每循环一次符号重复多一次。这与你熟悉的脚注符号或某种形式的字母计数系统有点像。下面分别举个例子，结果如图 15-36 所示。

```
@counter-style footnotes {
    system: symbolic;
    symbols: "*" "†" "§";
    suffix: ' ';
}
@counter-style letters {
    system: symbolic;
    symbols: A B C D E;
}
```

需要注意的一点是，如果把少量符号应用到特别长的列表上，记号很快就会变得特别长。以前例中的字母计数器为例。在使用这个计数系统的列表中，第 135 到第 150 个列表项目如图 15-37 所示。

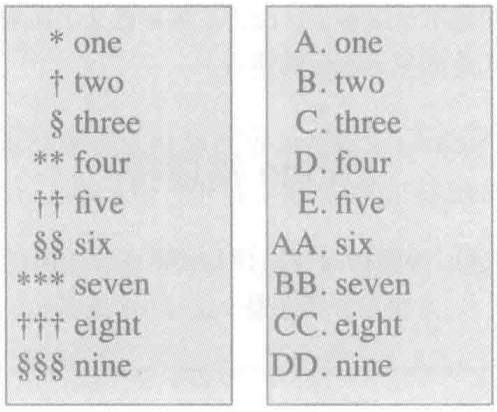


图 15-36：两种符号计数模式

EEEEEEEEEEEEEEEEEEEEEEEEEE. 135
AAAAAAAAAAAAAAAAAAAAAAAAAA. 136
BBBBBBBBBBBBBBBBBBBBBBBBBB. 137
CCCCCCCCCCCCCCCCCCCCCCCCCC. 138
DDDDDDDDDDDDDDDDDDDDDDDDDD. 139
EEEEEEEEEEEEEEEEEEEEEEEEEE. 140
AAAAAAAAAAAAAAAAAAAAAAAAAA. 141
BBBBBBBBBBBBBBBBBBBBBBBBBB. 142
CCCCCCCCCCCCCCCCCCCCCCCCCC. 143
DDDDDDDDDDDDDDDDDDDDDDDDDD. 144
EEEEEEEEEEEEEEEEEEEEEEEEEE. 145
AAAAAAAAAAAAAAAAAAAAAAAAAA. 146
BBBBBBBBBBBBBBBBBBBBBBBBBB. 147
CCCCCCCCCCCCCCCCCCCCCCCCCC. 148
DDDDDDDDDDDDDDDDDDDDDDDDDD. 149
EEEEEEEEEEEEEEEEEEEEEEEEEE. 150

图 15-37：特别长的符号记号

从现在起，不能把这个问题不当回事了，因为在某种意义上所有计数器格式都是累加的。避免受此问题影响的方式是使用 range 描述符。

range 描述符

取值	[[*<integer>* | infinite]{2}]# | auto
初始值	auto

range 的值为一对或多对以空格分开的值，每一对之间以逗号分隔。假设我们想在三次迭代之后停止字母重复。如果有 5 个符号，那么我们就可以把范围限制为 1-15，结果如图 15-38 所示（为免插图的尺寸太大，排成了两栏）。

```
@counter-style letters {
    system: symbolic;
    symbols: A B C D E;
    range: 1 15;
}
```

A. 1
B. 2
C. 3
D. 4
E. 5
AA. 6
BB. 7
CC. 8
DD. 9
EE. 10
AAA. 11
BBB. 12
CCC. 13
DDD. 14
EEE. 15
16. 16
17. 17
18. 18
19. 19
20. 20

图 15-38：使用 range 限制符号计数器模式

如果后面还有范围需要使用指定的计数器（管它什么原因），可以这样做：

```
@counter-style letters {
    system: symbolic;
```

```
    symbols: A B C D E;
    range: 1 15, 101 115;
}
```

这样，letters 定义的符号字母系统将应用在范围 1-15 和范围 101-115 上（记号是“AAAAAAAAAAAAAAAAAAAAA”到“EEEEEEEEEEEEEEEEEEEEE”）。

那么不在 range 定义的范围内的计数器怎么办呢？回落到默认的记号格式。这个问题可以留给用户代理处理，也可以使用 fallback 描述符自己定义。

fallback 描述符

取值	*<counter-style-name>*
初始值	decimal
备注	*<counter-style-name>* 可以是 list-style-type 的任何一个可用值

举个例子，或许你想让超出范围的计数器使用希伯来语计数方式。

```
@counter-style letters {
    system: symbolic;
    symbols: A B C D E;
    range: 1 15, 101 115;
    fallback: hebrew;
}
```

当然，也可以使用 lower-greek、upper-latin，甚至是非计数格式，例如 square。

如果主计数系统无法表示计数器中的某个值（不管是什么原因），也将使用后备计数格式。比如说在使用图像做符号的计数系统中，有个图像加载失败了。在下述声明中，假设 south.svg 没有加载。此时，缺少的图像将替换为 lower-latin 计数格式，表示计数器中当前项目的值：

```
@counter-style compass {
    system: symbolic;
    symbols: url(north.svg) url(east.svg) url(south.svg) url(west.svg);
    fallback: lower-latin;
}
```

15.3.4 字母计数模式

alpahbetic 计数系统与 symbolic 系统十分相似，只是重复的方式不同。还记得吗，使用符号计数系统时，符号的数量每循环一次就会叠加一次。而在字母计数系统中，各符号被当做计数系统中的数字。如果你熟悉电子表格的话就会发现，这种计数方法与列的标注很像。

为了演示这个计数模式，下面在前一节那个字母符号计数器的基础上，把符号系统改为字母系统，得到的结果如图 15-39 所示（同样，为了节省篇幅，排成了两栏）。

```
@counter-style letters {
    system: alphabetic;
    symbols: A B C D E;
    /* 仍在“E”处结束，以便快速看出这个模式的效果 */
}
```

A. one
B. two
C. three
D. four
E. five
AA. six
AB. seven
AC. eight
AD. nine
AE. ten
BA. 11
BB. 12
BC. 13
BD. 14
BE. 15
CA. 16
CB. 17
CC. 18
CD. 19
CE. 20

图 15-39：字母计数

注意，这个计数模式第二次迭代得到的是“AA”到“AE”，然后变成“BA”到“BE”，而后又变成“CA”等。换成符号计数系统，当字母计数系统到“EE”时，已经变成“EEEEEE”了。

注意，symbols 描述符的值必须至少有两个符号，这样的字母计数系统才是有效的。如果只提供一个符号，那么整个 @counter-style 块都是无效的。任意两个符号都有效，可以是字母、数字、Unicode 字符集中的任何字符，以及图像（依然是理论上）。

15.3.5 数字计数模式

严格来说，numeric 系统是使用提供的符号定义按位计数系统。即，提供的符号用作位数计数系统中的数字。例如，可以像下面这样定义普通的十进制计数系统：

```
@counter-style decimal {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
}
```

稍微扩充一下就能得到十六进制计数系统：

```
@counter-style hexadecimal {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9' 'A' 'B' 'C' 'D' 'E' 'F';
}
```

这个计数器从 1 到 F，然后从 10 到 1F，再从 20 到 2F、30 到 3F 等。简单多的二进制计数系统更是不费吹灰之力：

```
@counter-style binary {
    system: numeric;
    symbols: '0' '1';
}
```

这三个计数模式得到的结果如图 15-40 所示。

有个有趣的问题值得思考一下：计数器中的负值应该怎么处理呢？在十进制计数系统中，一般在负数前加上减号（-），但在其他计数系统中呢，比如符号计数系统？如果定义的是基于字母的数字计数系统呢？又或者，如果我们想使用会计常用的格式，把负值放在括号内呢？这就需要用到 negative 描述符了。

negative 描述符

取值	*<symbol> <symbol>?*
初始值	\2D（连字符或减号）
备注	negative 只能在支持负值的计数系统中使用，包括 alphabetic、numeric、symbolic 和 additive

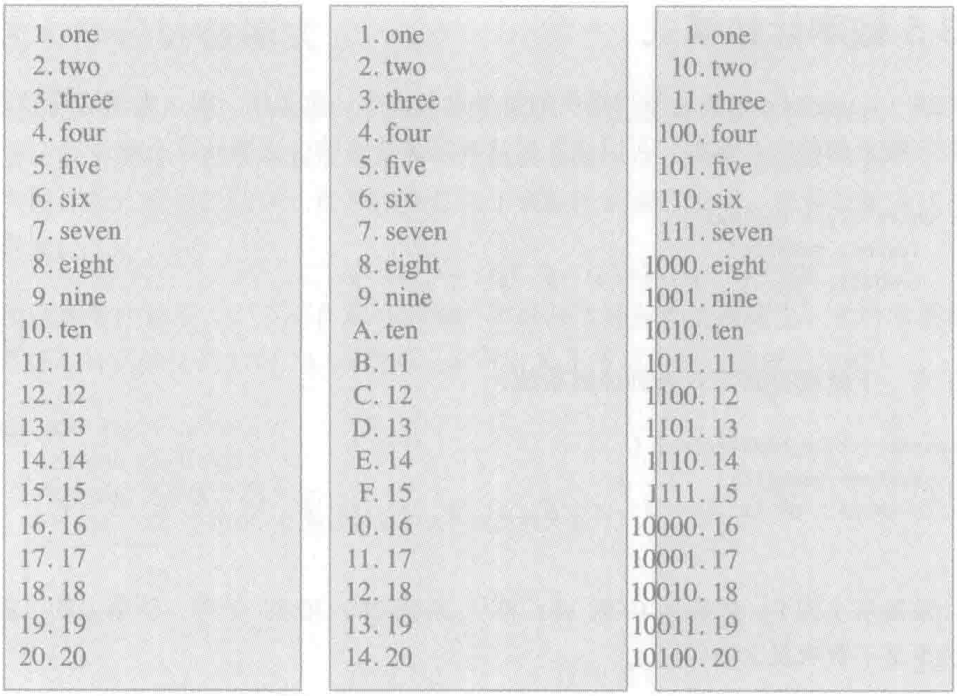


图 15-40：三个数字计数模式

negative 有点像是 prefix 和 suffix 的综合体，只不过仅在计数器为负值时起作用。negative 指定的符号显示在前缀和后缀符号内侧。

假设我们想使用会计常用的格式，而且为所有计数器都添加前缀和后缀符号。以下述代码为例，得到的结果如图 15-41 所示。

```
@counter-style accounting {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
    negative: "(" ")";
    prefix: "$";
    suffix: " - ";
}
ol.kaching {list-style: accounting;}
```

数字计数系统的另一个特性是为位数较少的值补位，让长度保持与位数较多的值相同。例如，计数模式可以使用前导零把“1”和“100”变成“001”和“100”。这个效果使用 pad 描述符实现。

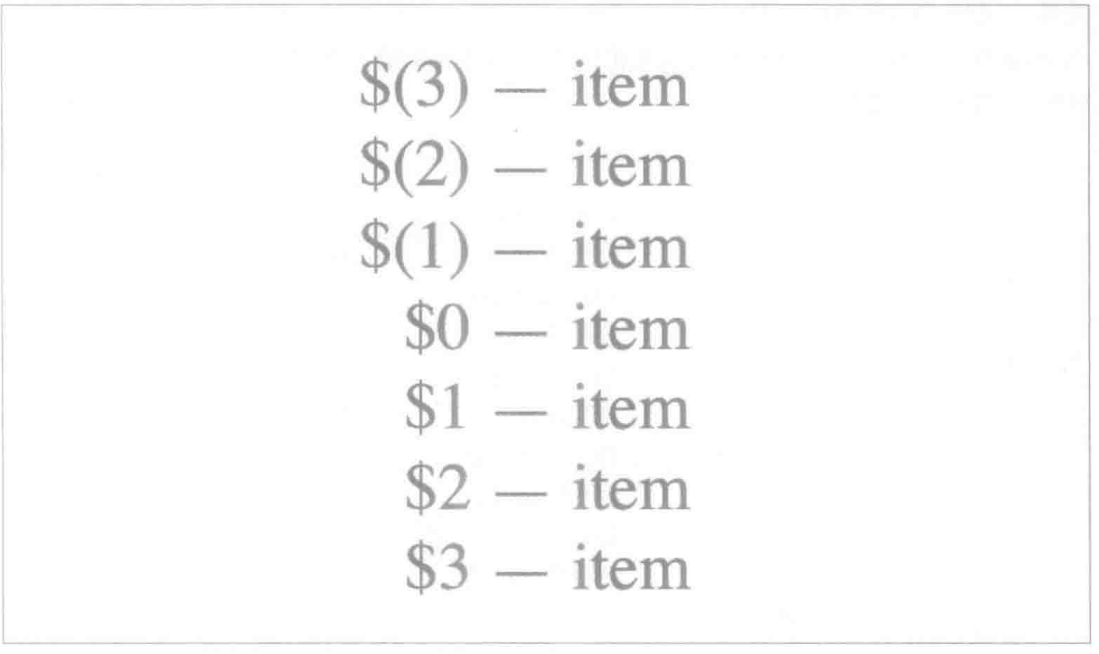


图 15-41：格式化负值

pad 描述符

取值	*<integer>* && *<symbol>*
初始值	0 ""

这个描述符的值有点特别。第一部分是一个整数，定义每个计数器有几位数。第二部分是一个字符串，用于填充位数不足的值。请看下面的例子：

```
@counter-style padded {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
    suffix: '.';
    pad: 4 "0";
}

ol {list-style: decimal;}
ol.padded {list-style: padded;}
```

应用上述样式的有序列表默认使用十进制计数系统，即：1，2，3，4，5…但是，class为padded的有序列表将使用有填充值的十进制计数系统：0001，0002，0003，0004，0005…图 15-42 是个例子。



图 15-42：填充值

注意，为了保证每个计数器都至少有四个数字，没有数字的位都是用符号 0 填充的。前一句中“至少”是不能少的，如果计数器变成了五位数，那么就不会再填充。更重要的一点是，一旦计数器达到五位数，其他较短的计数器不会再多填充零，而是保持四位数，因为 4"0" 中定义的是 4。

任何符号都能用于填充，而不限于 0。可以使用下划线、句点、表情符号、空格等。其实，<*symbol*> 部分还可以是多个字符。如果愿意，完全可以像下面这些写：

```
@counter-style crazy {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
    suffix: '.';
    pad: 4 "🙈🙉";
}

ol {list-style: decimal;}
ol.padded {list-style: padded;}
```

计数器的值为 1 时，计数系统得到的结果将是“🙈🙉🙊🙈🙉🙊 1”，很疯狂吧。

注意，负值符号算在记号的长度内，因此会对填充产生影响。另外，负值符号显示在填充的值外侧。因此，下述样式将得到如图 15-43 所示的结果。

```
@counter-style negativezeropad {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
    suffix: '. ';
    negative: '-';
    pad: 4 "0";
}
@counter-style negativespacepad {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
    suffix: '. ';
    negative: '-';
    pad: 4 " ";
}
```

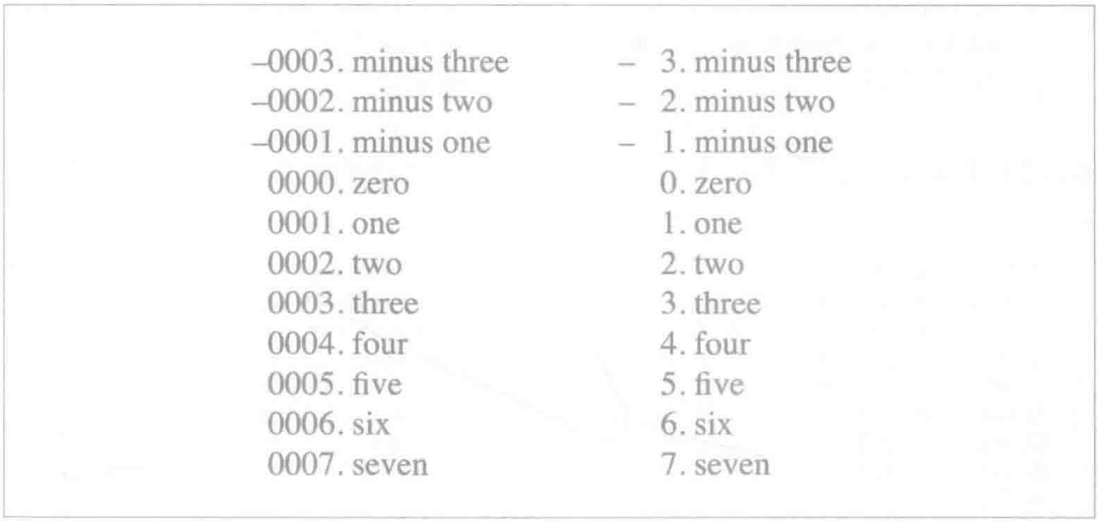


图 15-43：填充值

15.3.6 累加计数模式

还有一种计数系统没有探讨，即 `additive-symbol`。在累加计数系统中，使用不同的符号表示不同的数值，然后把多个符号加在一起，得到计数器的值。

additive-symbols 描述符

取值	[*<integer>* && *<symbol>*]#
初始值	n/a
备注	*<integer>* 必须是非负数，计数器为负值时，累加计数模式不生效

看着有点抽象，下面举个例子。这个例子摘自 Kseso（*https://escss.blogspot.com/*）：

```
@counter-style roman {
    system: additive;
    additive-symbols:
        1000 M, 900 CM, 500 D, 400 CD,
        100 C, 90 XC, 50 L, 40 XL,
        10 X, 9 IX, 5 V, 4 IV, 1 I;
}
```

这个计数器实现的是典型的罗马计数风格。下面这个例子也不错，摘自计数格式的规范，使用骰子定义一个计数系统：

```
@counter-style dice {
    system: additive;
    additive-symbols: 6 □, 5 □, 4 □, 3 □, 2 □, 1 □, 0 "__";
    suffix: " ";
}
```

这两个计数系统得到的结果如图 15-44 所示，这一次各列表排成了三栏。

-3. minus three	VI. six	XV. 15
-2. minus two	VII. seven	XVI. 16
-1. minus one	VIII. eight	XVII. 17
0. zero	IX. nine	XVIII. 18
I. one	X. ten	XIX. 19
II. two	XI. 11	XX. 20
III. three	XII. 12	XXI. 21
IV. four	XIII. 13	XXII. 22
V. five	XIV. 14	XXIII. 23

-3 minus three	⊞ six	⊞⊞⧅ 15
-2 minus two	⊞⊡ seven	⊞⊞⊞ 16
-1 minus one	⊞⊡ eight	⊞⊞⊠ 17
__ zero	⊞⧅ nine	⊞⊞⊞ 18
⊡ one	⊞⊞ ten	⊞⊞⊞⊡ 19
⊡ two	⊞⊠ 11	⊞⊞⊞⊡ 20
⧅ three	⊞⊞ 12	⊞⊞⊞⧅ 21
⊞ four	⊞⊞⊡ 13	⊞⊞⊞⊞ 22
⊠ five	⊞⊞⊡ 14	⊞⊞⊞⊠ 23

图 15-44：累加计数

明确起见，符号可以放在引号中，例如 6"□",5"□",4"□" 等。

特别注意，符号及其对应值的顺序十分重要。看到没有，罗马和骰子计数系统中的值都是从大到小排列的，而不是其他顺序？这是因为，如果不按倒序排列，整个 `@counter-style` 块都将失效。

另外注意，使用的是 `additive-symbols` 描述符，而不是 `symbols`。这一点也很重要，如果使用 `symbols` 描述符定义累加计数系统，整个 `@counter-style` 块都将失效（同样，如果在 `additive` 以外的系统中使用 `additive-symbols` 描述符，也将导致 `@counter-style` 块失效）。

关于累加计数系统还有一点需要注意：根据累加计数器算法的定义方式，得到的累加计数系统可能无法表示某些值。请看下述定义：

```
@counter-style problem {
    system: additive;
    additive-symbols: 3 "Y", 2 "X";
    fallback: decimal;
}
```

这个计数器产出的前五个数分别是：1，X，Y，4，YX。你可能以为“4”应该是“XX”，这是符合直觉的想法，但是累加计数器算法不是这样运作的。规范是这样说的：“然而不幸的是，这样会导致算法的运行时间随计数器值的大小线性增加。”



那么罗马计数系统怎么用“Ⅲ”表示3呢？答案同样在算法中。这个问题有点复杂，如果你真的好奇，建议阅读定义累加计数算法的CSS Counter Styles Level 3规范。如果不想这么麻烦，只需记住一点：有表示1的符号就能避免这个问题。

15.3.7 扩展计数模式

有时，你可能只想对现有的计数系统做些调整。假如你想让常规的十进制计数系统使用结束括号作为后缀，并且最多填充两个前导零。此时，可以像下面这样全新定义一个计数器：

```
@counter-style mydecimals {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
    suffix: ") ";
    pad: 2 "0";
}
```

这样做是可以，但是有点麻烦。别担心，遇到这样的情况可以使用extends。

extends算是一种计数系统，但是它建构在现有的计数系统之上。使用extends，前例可以改成下面这样：

```
@counter-style mydecimals {
    system: extends decimal;
    suffix: ") ";
    pad: 2 "0";
}
```

我们从 list-style-type 现有的计数系统中选取 decimal，然后做些格式调整。可见，无需重新输入整个符号集，只要按需求调整一些选项即可。

其实，你也只能调整选项。如果尝试在 extneds 系统中使用 symbols 或 additive-symbols 描述符，整个 @counter-style 块都将失效，从而被忽略。也就是说，使用的符号不能扩展。例如，不能扩展十进制计数系统实现十六进制计数系统。

然而，我们可以针对不同的上下文调整十六进制计数系统。比如说，你可以先设置一个基本的十六进制计数系统，然后定义几个不同的显示模式。所用的样式如下，结果如图 15-45 所示（注意，各个列表从 19 直接跳到了 253，因为我们在某个列表项目上设置了 value="253"）。

```
@counter-style hexadecimal {
    system: numeric;
    symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9' 'A' 'B' 'C' 'D' 'E' 'F';
}
@counter-style hexpad {
    system: extends hexadecimal;
    pad: 2 "0";
}
@counter-style hexcolon {
    system: extends hexadecimal;
    suffix: ": ";
}
@counter-style hexcolonlimited {
    system: extends hexcolon;
    range: 1 255; /* 到 FF 停止 */
}
```

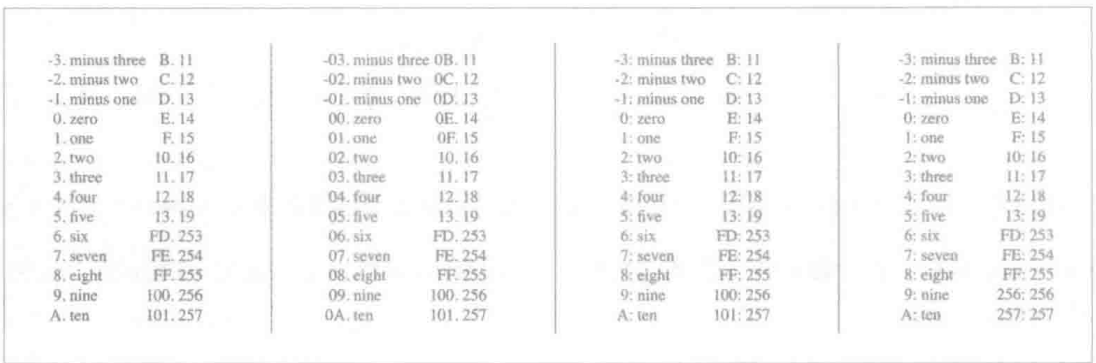


图 15-45：不同的十六进制计数模式

注意，第四个计数器格式 hexcolonlimited 扩展第三个计数器 hexcolon，而第三个计数器又扩展第一个计数器 hexadecimal。在 hexcolonlimited 中，十六进制计数系统到 FF（255）停止，因为我们声明了 range: 1 255;。

15.3.8 发音计数模式

虽然使用符号能构建很多有趣的计数器，但是得到的结果对语音技术（例如苹果的 VoiceOver 或 JAWS 屏幕阅读器）来说却是一团糟。试想屏幕阅读器阅读骰子计数器或月相计数器会产生什么效果。为了避免这种问题，我们可以使用 speak-as 描述符定义后备发音策略。

speak-as 描述符

取值	auto | bullets | numbers | words | spell-out | *<counter-style-name>*
初始值	auto

下面倒过来讲各个可用的值。*<counter-style-name>* 用于指定用户代理自身或许能识别的一个计数系统。例如，可以让骰子计数系统使用十进制计数系统的读音：

```
@counter-style dice {
    system: additive;
    speak-as: decimal;
    additive-symbols: 6 ⚅, 5 ⚄, 4 ⚃, 3 ⚂, 2 ⚁, 1 ⚀;
    suffix: " ";
}
```

在上述样式的作用下，这个计数器将把“⚅⚅⚄”读作“十五”。如果把 speak-as 的值改为 lower-latin，那么“⚅⚅⚄”就读作“ou”（大写字母 O）。

spell-out 看似简单，不过深究起来还是有点复杂的。用户代理拼读出来的是“计数器的表示形式”，然后逐个字母读出来。结果难以预料，因为生成计数器表示形式的方法没有明文规定，规范只说“计数器的表示形式由计数器的字符拼接而成”，具体怎么做却没说。

words 与 spell-out 类似，只不过计数器的表示形式是以词为单位读出来的，而不是逐个字母发音。同样，具体过程没有定义。

使用 numbers 值时，计数器用文档所用的语言读出来。这与前面的代码示例很像，在英语编写的文档中，“🔟🔟🔢”读作“fifteen”。语言不同，发音也不同：在西班牙语中读作“quince”，在德语中读作“fünfzehn”，在汉语中读作“shíwǔ”等。

设为 bullets 时，用户代理用无序列表中圆点记号的读音阅读计数器。有些用户代理可能什么也不读，有些则会发出特别的声音，例如鸣一声或嘀一下。

最后是默认值 auto。最后才说这个值，是因为它的具体效果取决于使用的计数系统。如果是 alphabetic 系统，speak-as:auto 的效果等同于 speak-as:spell-out。在 cyclic 系统中，auto 等效于 bullets。在其他计数系统中的效果都与 speak-as:numbers 相同。

但 extends 系统例外。此时，auto 的效果取决于被扩展的计数系统。因此，对下述样式来说，emojibrackets 列表中计数器的读音就像把 speak-as 设为 bullets 一样：

```
@counter-style emojilist {
    emojiverse {
    system: cyclic;
    symbols: 😀 😁 😂 🤣 😃 ;
@counter-style emojibrackets {
    system: extends emojilist;
    suffix: "]] ";
    speak-as: auto;
}
```

15.4 小结

即便列表样式没有我们想象的那么丰富，而且浏览器对生成的内容还欠缺支持（当然是截至写作本书时），但装饰列表的功能还是很有用的。我们经常会创建链接列表，把记号和缩进去掉，得到导航侧边栏。简单的标记，加上弹性布局，真让人难以抵挡。随着 CSS3 对列表样式的增强，我想，列表的用处将越来越大。

目前，在没有原生列表元素的标记语言中，生成的内容起到了很大的作用，例如，为不同类型的链接添加不同的图标（标出 PDF 文件、Word 文档，甚至是指向其他网站的链接）。使用生成的内容，还能打印出链接的 URL。插入并格式化引号的功能还能实现精美的排版效果。可以说，生成的内容能实现的效果只有想不到，没有做不到。更强大的一点是，我们可以使用计数器把排序信息添加到不算列表的元素上，例如标题或代码块。下一章讨论如何改变界面中元素的位置、形状，以及整体观感。

第 16 章

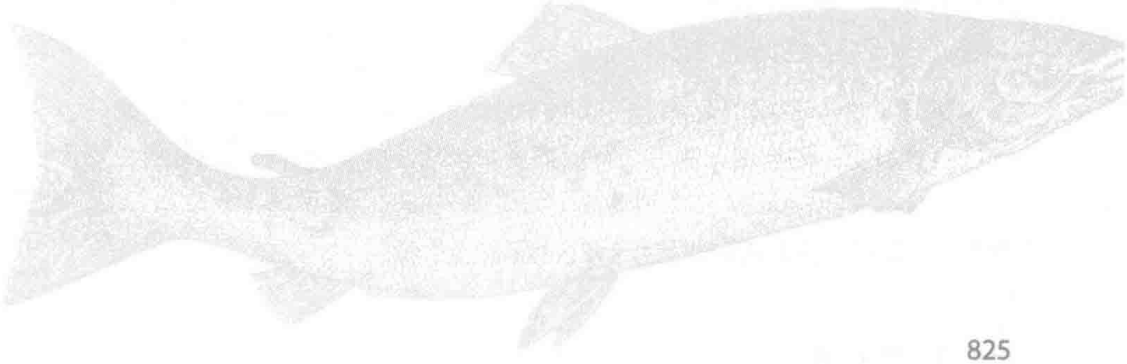
变形

自层叠样式表（Cascading StyleSheet，CSS）诞生以来，元素始终是矩形的，而且只能沿横轴和纵轴放置。有些技巧能让元素看起来是倾斜的，但是底层的坐标方格并没有变形。时间来到 21 世纪 10 年代之前的几年，人们的兴趣日渐高涨，希望打破坐标方格的束缚，能以不同的方式改变对象的形态，而且还不只限于二维。

如果你定位过对象的位置，不管是相对定位还是绝对定位，你就改变过对象的形态。就此而论，使用浮动或负外边距技巧（抑或二者兼具）也算是改变了对象的形态。这些都是平移（translation）的例子，即把元素从原本所在的位置移到其他位置上。利用 CSS 提供的变形功能，不仅能平移元素，还能做很多其他的事。不论你想旋转照片，让照片看起来更自然，还是让界面元素在翻转后才能看到信息，又或是使用一些技巧，以有趣的视域显示侧边栏，CSS 变形功能都能做到。不是我夸大，CSS 变形功能甚至能让设计有所改观。

16.1 坐标系

旅程开始之前，先花点时间调整一下。变形涉及两种坐标系，最好事先了解一下。





如果你已经熟知笛卡尔坐标系和球坐标系，尤其是在运算中的用法，可以跳到下一节。

首先是笛卡尔坐标系，也就是通常所说的 *x*/*y*/*z* 坐标系。这种坐标系使用两个数字（二维）或三个数字（三维）表示一个点在空间中的位置。在 CSS 中，这个坐标系使用三个轴表示：x 轴（横轴）、y 轴（纵轴）和 z 轴（深度轴），如图 16-1 所示。

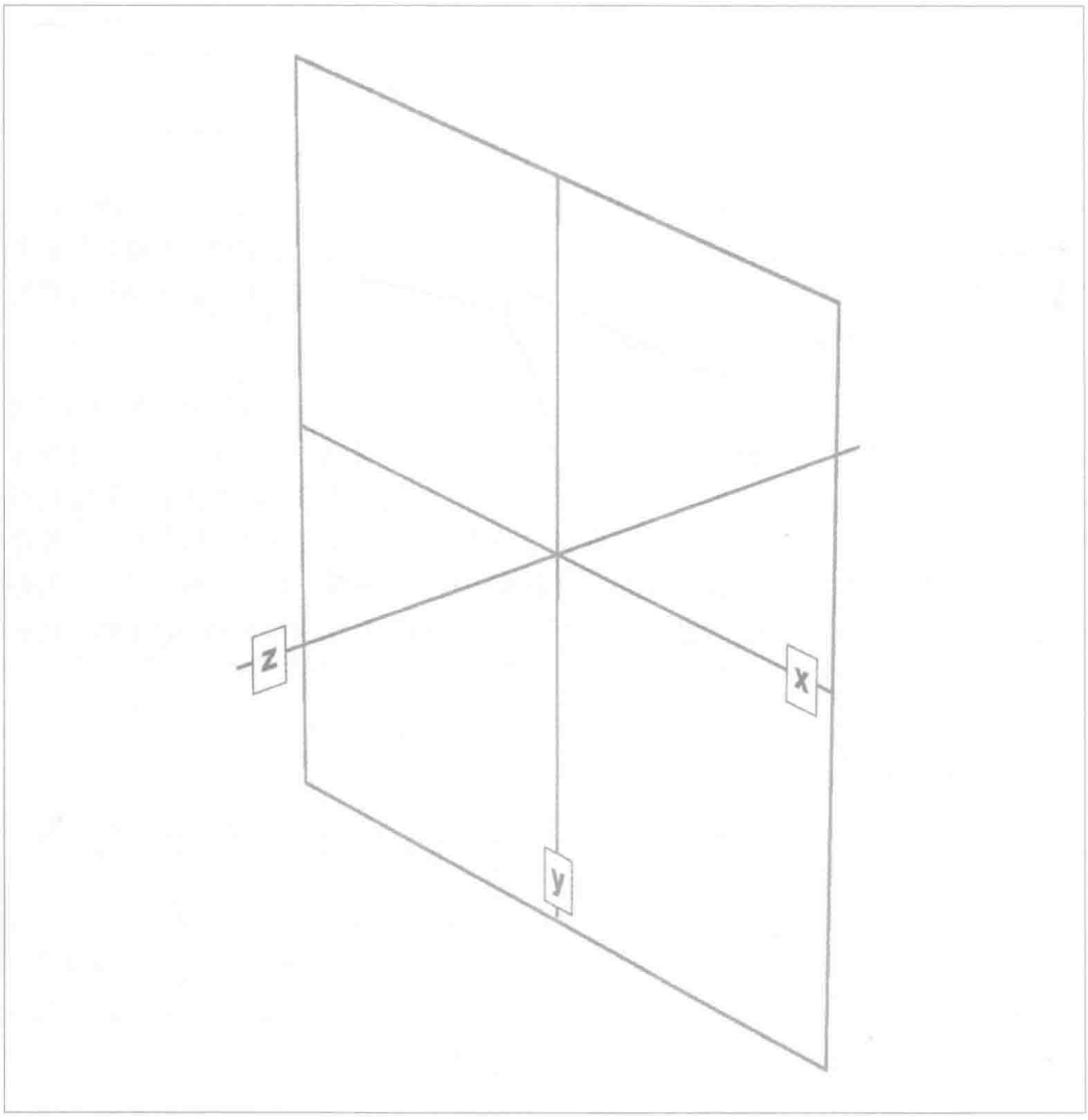


图 16-1：CSS 变形功能中表示笛卡尔坐标系的三个轴

2D（二维）变形只需关注 x 轴和 y 轴。按约定，x 轴上的正值在右侧，负值在左侧。类似地，y 轴上的正值沿纵轴向下，负值沿纵轴向上。

这有点奇怪，学过初等代数的人大都知道，数值越大位置应该越高，而不是越低（在图 16-1 中，“y”标注在 y 轴的底部；三个轴的标注都在正方向那一侧）。如果你用过 CSS 中的绝对定位，回想一下绝对定位元素时使用的 top 属性：值为正数时元素下移，值为负数时则上移。

鉴于此，若想把元素向左下方移动，要把 x 值设为负数、把 y 值设为正数，如下所示：

```
translateX(-5em)translateY(33px)
```

稍后就会发现，这个变形值是有效的。上述代码的作用是把元素向左移动 5em、向下移动 33 像素。

如果想在三维空间中改变元素的形态，要加上 z 轴值。这个轴从显示器上“跃出”，指向你的眼前。理论上，就是这样。z 轴上的正值离你较近，负值离你较远。在这方面，z 轴与 z-index 属性是完全一样的。

下面在上述移动过的元素上再加个 z 轴值：

```
translateX(-5em) translateY(33px) translateZ(200px)
```

现在，与未指定 z 值相比，元素离我们的距离近了 200 像素。

你可能觉得奇怪，元素怎么可能离我们近了 200 像素呢，毕竟全息显示器非常少见，而且价格不菲？我与显示器之间的 200 像素有多少摩尔的空气呢？元素离自己近了到底有什么效果，如果近在眼前呢？这些问题在后文将逐一解答。现在，你只要相信，在 z 轴上移动元素看起来就是把元素移近或移远即可。

真正需要注意的是，每个元素都有自己的参照系，各轴都相对自身而动。也就是说，如果旋转了元素，轴也随之旋转，如图 16-2 所示。旋转之后再变形，是相对旋转后的轴计算的，而不是显示器的轴。

提到旋转，这就引出 CSS 变形功能使用的另一个坐标系——球坐标系。这个坐标系用于描述 3D 空间中的角度，如图 16-3 所示。

在 2D 变形中只需关注全周 360 度极坐标系，即由 x 轴和 y 轴构成的平面。对旋转来说，2D 旋转其实是在绕 z 轴旋转。类似地，如果绕 x 轴旋转，元素将偏向我们或远离我们，而绕 y 轴旋转的话，元素将向两侧翻转。这些旋转方式如图 16-4 所示。

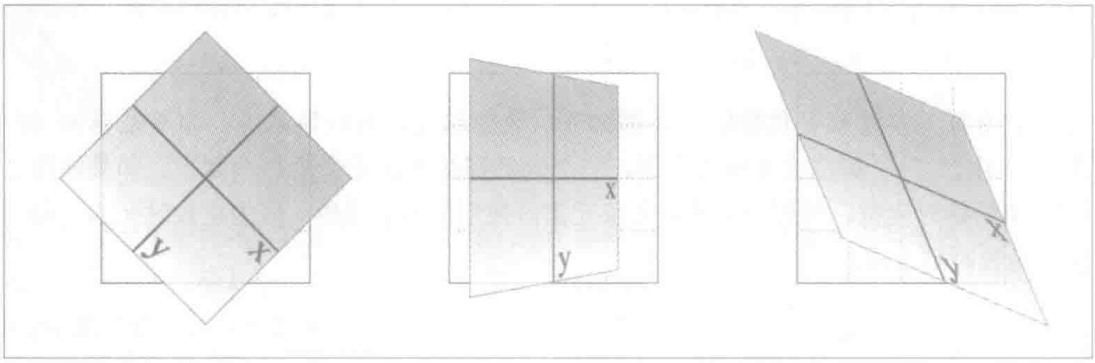


图 16-2：基本参照系

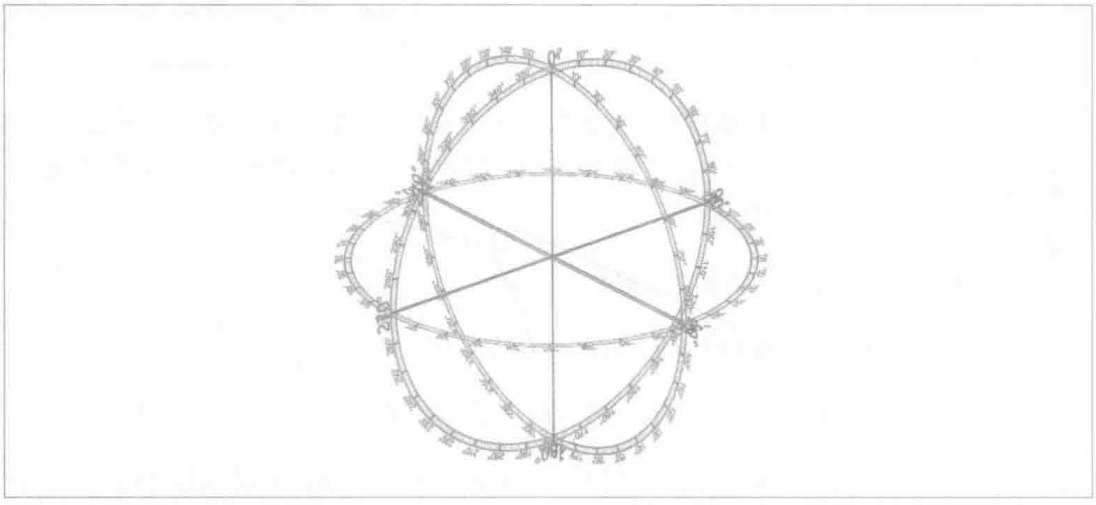


图 16-3：CSS 变形功能使用的球坐标系

再回到 2D 旋转。假设我们想在显示器平面上（即绕 z 轴）顺时针旋转一个元素 45 度，那么使用的变形值可能是下面这样的：

```
rotate(45deg)
```

如果把值改为 -45deg，元素将绕 z 轴逆时针旋转。换句话说就是，元素在 xy 平面上旋转，如图 16-5 所示。

好的，了解这些基础知识之后，我们开始变形吧！

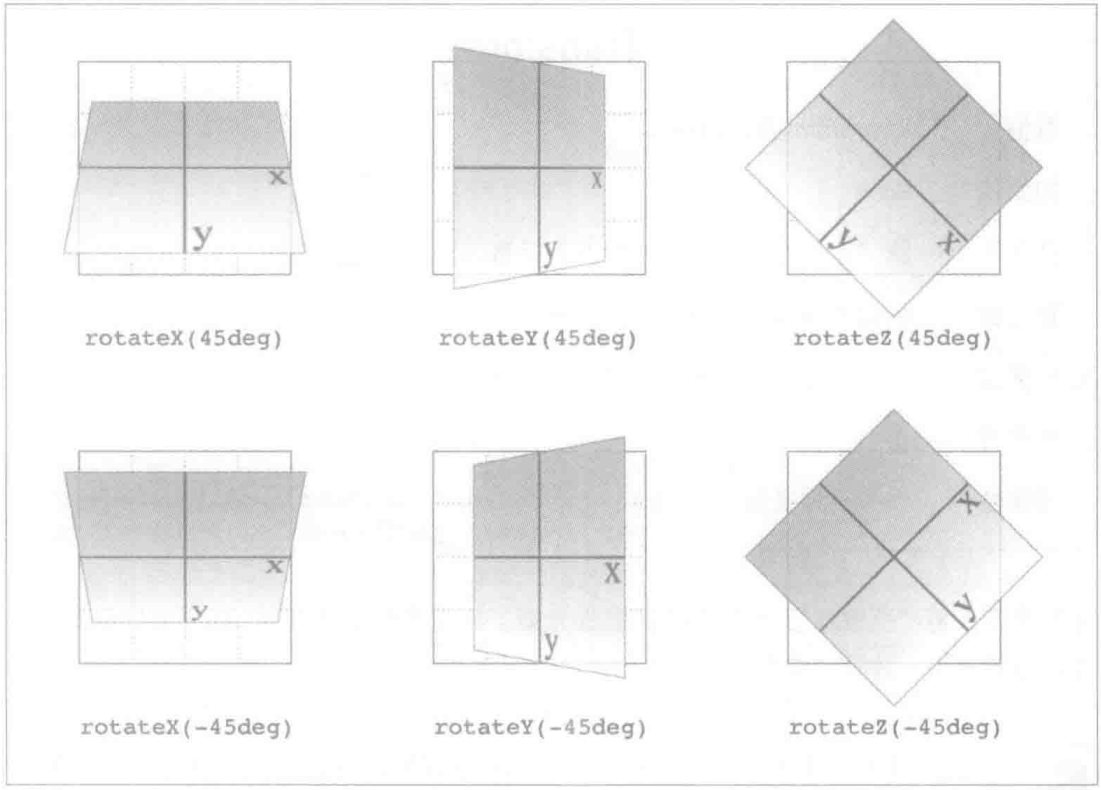


图 16-4：绕三个轴旋转

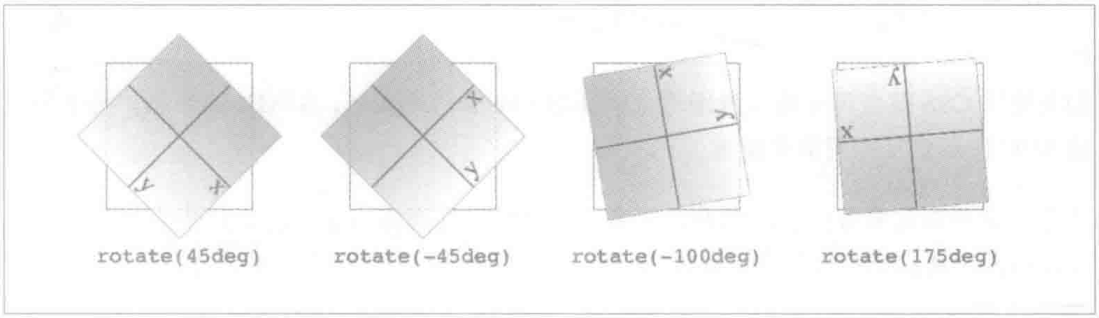


图 16-5：在 xy 平面上旋转

16.2 变形

变形其实只有一个属性，不过有几个辅助属性用于控制如何变形。先从主要的属性入手。

Transform

取值	*<transform-list>* | none
初始值	None
适用于	除“基元行内”框之外的所有元素（见下文的说明）
百分数	相对范围框计算（见下文的说明）
计算值	指定的值，不过相对长度值会计算为绝对长度
继承性	否
动画性	作为一种变形

先说明什么是“范围框”。受 CSS 控制的元素，其范围框是边框框，即元素边框的外边界。也就是说，计算范围框时，轮廓和外边距不算在内。



以表格形式显示的元素，其范围框是表格的容器框，包括表格框和相应的表题框。

如果使用 CSS 改变可缩放矢量图形（Scalable Vector Graphic，SVG）的形态，其范围框是 SVG 图形定义的对象范围框。

注意，变形的元素有自己的堆叠上下文。经过缩放的元素可能比变形前小或大，不过元素在页面上所占的空间与变形前保持不变。这一点对所有变形函数都成立。

取值句法中的 `<transform-list>` 也需要说明一下。这个占位符表示一个或多个变形函数，一个接一个，中间以空格分隔，像下面这样，结果如图 16-6 所示：

```
#example {transform: rotate(30deg) skewX(-25deg) scaleY(2);}
```

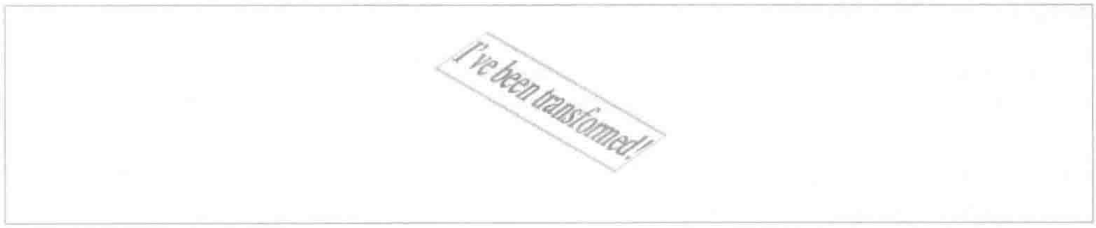


图 16-6：变形后的 div 元素

变形函数一次只处理一个，从第一个（最左边）开始，一直到最后一个（最右边）。从头到尾的处理顺序是很重要的，顺序变了，得到的结果可能就大有不同。请看下面两个规则，结果如图 16-7 所示：

```
img#one {transform: translateX(200px) rotate(45deg);}
img#two {transform: rotate(45deg) translateX(200px);}
```

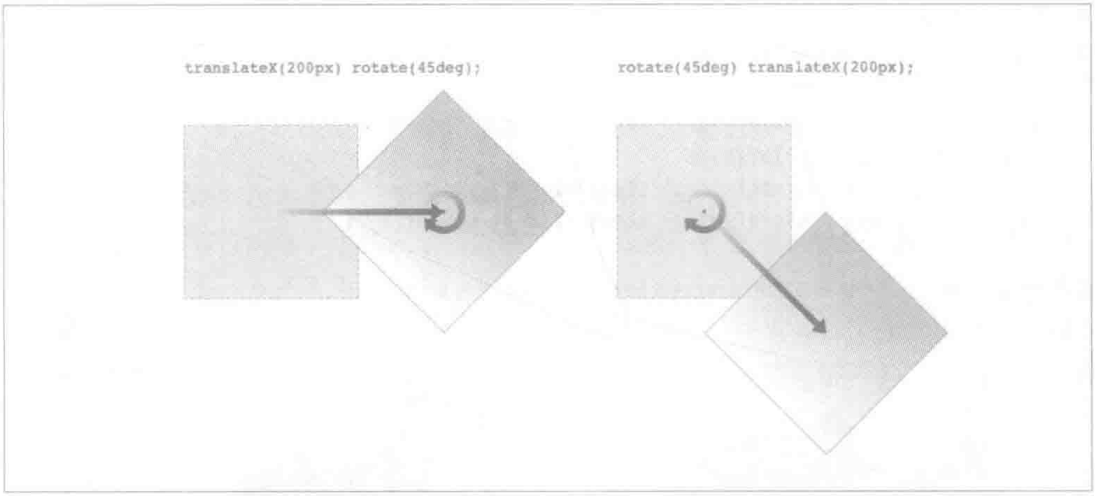


图 16-7：不同的变形顺序，不同的结果

在第一个示例中，图像先沿 x 轴移动 200 像素，然后旋转 45°。在第二个示例中，图像先旋转 45°，然后沿 x 轴移动 200 像素，然而这里的 x 轴指的是变形后元素的 x 轴，而不是父元素、页面或视区的 x 轴。也就是说，元素旋转后，其 x 轴（及其他轴）也随之旋转了。所有变形都是相对元素自己的参照系实施的。

相比之下，不管先移动再缩放，还是先缩放再移动，得到的结果都是一样的：

```
img#one {transform: translateX(100px) scale(1.2);}
img#two {transform: scale(1.2) translateX(100px);}
```

顺序有影响的情况要比没有影响的情况多得多，因此一般来说，最好假定始终需要考虑顺序，即便严格来说并非总是如此。

注意，有多个变形函数时，每个都要正确设置，确保全部有效。即使有一个函数是无效的，整个值都将失效。例如：

```
img#one {transform: translateX(100px) scale(1.2) rotate(22);}
```

因为 `rotate()` 的值是无效的（旋转值必须带单位），所以整个值都将被忽略。目标图像就待在原来的位置上，没有平移也没有缩放，更别说旋转了。

此外还要注意，变形通常不叠加。也就是说，如果改变了元素的形态，而后想再添加一种变形，那么要在原变形的基础上修改。请看下面两种情况，结果如图 16-8 所示：

```
#ex01 {transform: rotate(30deg) skewX(-25deg);}
#ex01 {transform: scaleY(2);}
#ex02 {transform: rotate(30deg) skewX(-25deg);}
#ex02 {transform: rotate(30deg) skewX(-25deg) scaleY(2);}
```

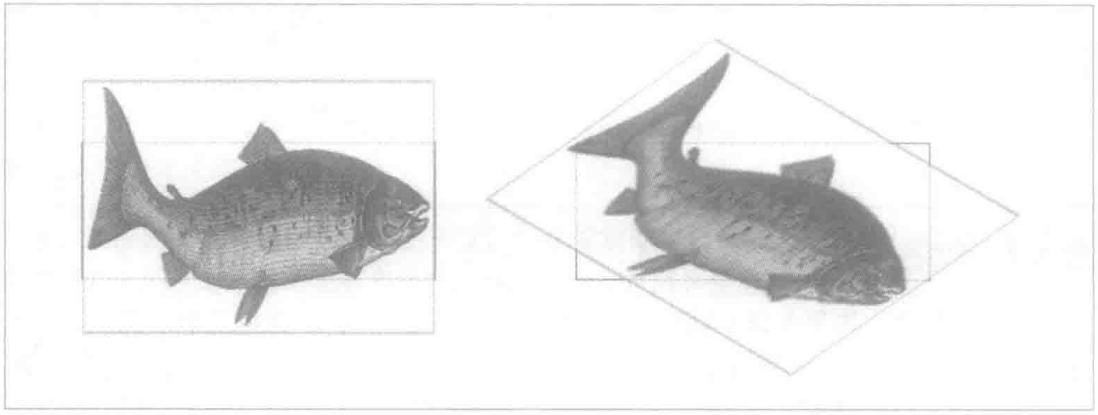


图 16-8：覆盖或修改变形

在第一种情况中，第二个规则完全取代了第一个规则，因此元素只会在 y 轴上缩放。这其实不难理解，就像你在一个地方为元素声明了字号，又在另一个地方为元素声明不同的字号一样，字号是不叠加的，只有其中一个字号起作用。在第二个示例中，第一个规则中的变形都出现在第二个规则中，因此它们将与 `scaleY()` 函数一起起作用。

不过动画变形例外，不管使用过渡还是真正的动画，效果是叠加的。因此，可以直接在变形的元素上以动画的形式呈现其中一个变形函数，其他变形函数不会被覆盖。假如现在是这样的：

```
img#one {transform: translateX(100px) scale(1.2);}
```

如果又以动画的形式呈现旋转角度的过程，元素将从平移并缩放后的状态开始旋转到新角度，平移和缩放都将实施。

有趣的是，即使不显式使用过渡或动画，也能通过用户交互伪类（例如 :hover）实现叠加的变形。这是因为悬停等效果就是一种过渡，只不过不是由过渡属性触发的。因此，可以像下面这样声明：

```
img#one {transform: translateX(100px) scale(1.2);}
img#one:hover {transform: rotate(-45deg);}
```

鼠标悬停在这个图像上时，图像在平移并缩放后将向左旋转 45°。旋转过程持续零秒，因为没有声明过渡持续时间，但这仍是一种过渡，只不过是隐式的。因此，状态变化都可以视作过渡，而且状态变化带来的变形都会叠加在之前的变形上。

特别注意，截至写作本书时，变形不能应用到基元（atomic）行内框上。基元行内框指 span、超链接等行内框。这些元素可以随块级父元素一起变形，但是不能直接旋转 span 元素，除非把显示方式改为 display:block、display:inline-block 等。这一限制的根源在于不确定性。假设一个 span 元素（或其他行内框）分为多行，如何旋转呢？每个行框独自旋转，还是全部行框作为一个整体旋转呢？现在还没有统一的答案，争论仍在继续，因此目前无法直接改变行内框的形态。

变形函数

写作本书时，一共有 21 个变形函数。不同的变形函数利用不同格式的值实施相应的变形。这些可用的变形函数及其值的格式见表 16-1。

表 16-1：变形函数

translate()	scale()	rotate()	skew()	matrix()
translate3d()	scale3d()	rotate3d()	skewX()	matrix3d()
translateX()	scaleX()	rotateX()	skewY()	perspective()
translateY()	scaleY()	rotateY()		
translateZ()	scaleZ()	rotateZ()		

transform 属性的值通常是由空格分隔的一个或多个函数，各函数从头（左）至尾（右）依次处理，而且每个函数的值都必须是有效的。倘若有一个函数的值是无效的，那么 transform 属性的整个值都将失效，因而也就不做任何变形。

平移函数

平移变形指沿一个轴或多个轴移动。例如，translateX() 沿元素自身的 x 轴移动元素，translateY() 沿元素自身的 y 轴移动元素，translateZ() 沿元素自身的 z 轴移动元素。

函数	可取的值
translateX(), translateY()	*<length>* | *<percentage>*

这两个通常称为“2D”平移函数，因为它们能上下移动元素，也能左右移动元素，但是不能沿 z 轴前后移动元素。这两个函数的值都是一个距离值，可以是长度，也可以是百分数。

值为长度时，效果与你预想的差不多。使用 translateX(200px) 沿 x 轴移动 200 像素的结果是元素向右移动 200 像素。换成 translateX(-200px)，元素则向左移动 200 像素。对 translateY() 来说，正值向下移动元素，负值向上移动元素，而且都是相对元素自身的轴而言的。因此，如果通过旋转对调上下两边，那么 translateY() 的正值将在页面上向上移动元素。

如果值是百分数，移动距离相对元素自身的尺寸计算。对宽度为 30 像素、高度为 200 像素的元素来说，translateX(50%) 把元素向右移动 150 像素，translateY(-10%) 把元素向上移动 20 像素（相对原位置）。

函数	可取的值
translate()	[*<length>* | *<percentage>*] [, *<length>* | *<percentage>*]?

如果想同时沿 x 轴和 y 轴移动，使用 translate() 更方便。第一个值是沿 x 轴的移动量，第二个值是沿 y 轴的移动量，这与 translateX()translateY() 结合在一起的作用是一样的。如果省略 y 值，假定为零。因此，translate(2em) 视作 translate(2em,0)，也等同于 translateX(2em)。图 16-9 是一些 2D 平移示例。

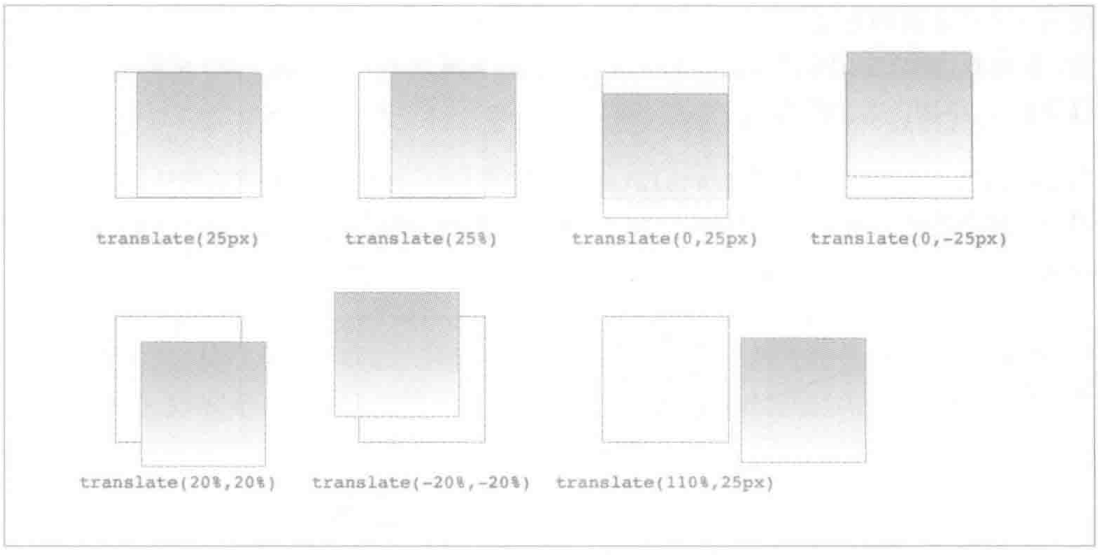


图 16-9：二维平移

根据最新版规范，两个 2D 平移函数的值都可以不带单位。此时，提供的数字采用用户单位，如未设为其他单位，就是像素。CSS 规范没有说明如何设定用户单位，但是 SVG 规范有规定，然而不详细。经测试，写作本书时没有浏览器支持不带单位的平移值，因此目前只是理论上支持这个特性。

函数	可取的值
translateZ()	*<length>*

这个函数沿 z 轴平移元素，即在第三个维度中移动元素。与 2D 平移函数不同，translateZ() 只接受长度值。translateZ() 不允许使用百分数值，其实任何有关 z 轴的值都不可以使用百分数。

函数	可取的值
translate3d()	[*<length>* | *<percentage>*], [*<length>* | *<percentage>*], [*<length>*]

我们知道 translate() 能同时设定 x 轴和 y 轴平移，类似地，translate3d() 这个简写属性能同时设定 x 轴、y 轴和 z 轴的平移量。如果想一次性把元素向右、向上和向前移动，

使用这个函数特别方便。3D 平移的过程如图 16-10 所示。图中的箭头表示沿相应轴的移动，最终到达 3D 空间中的一点。图中的虚线表示离原点（三个轴的交点）的距离和方向，以及在 xz 平面上方的距离。

与 translate() 不同，如果 translate3d() 的值少于三个，没有假定的默认值。因此，用户代理应该把 translate3d(1em,-50px) 视作无效的，而不能假定为 translate3d(2em,-50px,0)。

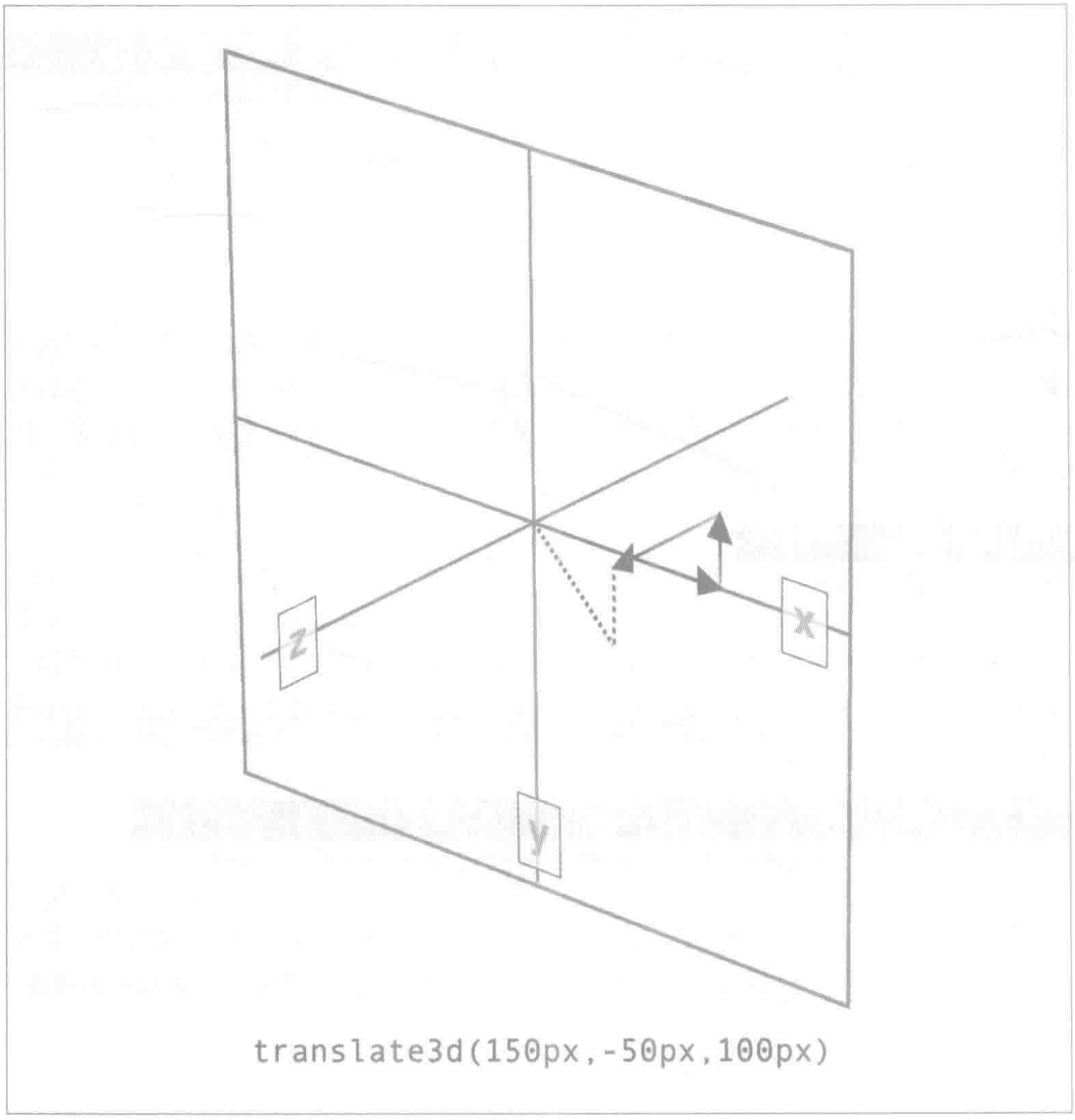


图 16-10：三维平移

缩放函数

缩放变形把元素放大或缩小，具体取决于提供的值。缩放函数的值都是无单位的实数，而且始终为正数。在 2D 平面中，可以分别在 x 轴或 y 轴上缩放，也可以同时在两个轴上缩放。

函数	可取的值
scaleX(), scaleY(), scaleZ()	*<number>*

提供给缩放函数的数字是个乘数，因此，scaleX(2) 将把元素的宽度变为变形前的两倍，而 scaleY(0.5) 将把元素的高度缩小一半。这样看，你可能以为缩放值也能使用百分数，但其实不然。

函数	可取的值
scale()	*<number>* [, *<number>*]?

如果想在两个轴上同时缩放，使用 scale()。第一个始终是 x 值，第二个则是 y 值，因此 scale(2,0.5) 将把元素的宽度放大两倍，把元素的高度缩小一半。如果只提供一个值，用作两个轴的缩放值；因此，scale(2) 将把元素的宽度和高度都放大两倍。这与 translate() 是不同的，在 translate() 中，省略的第二个值始终被设为零。scale(1) 缩放的元素与缩放前的尺寸完全相同，scale(1,1) 也是，万一你真想这么做的话。

图 16-11 展示了几个缩放的示例，有的用单轴缩放函数，有的用二合一的 scale()。

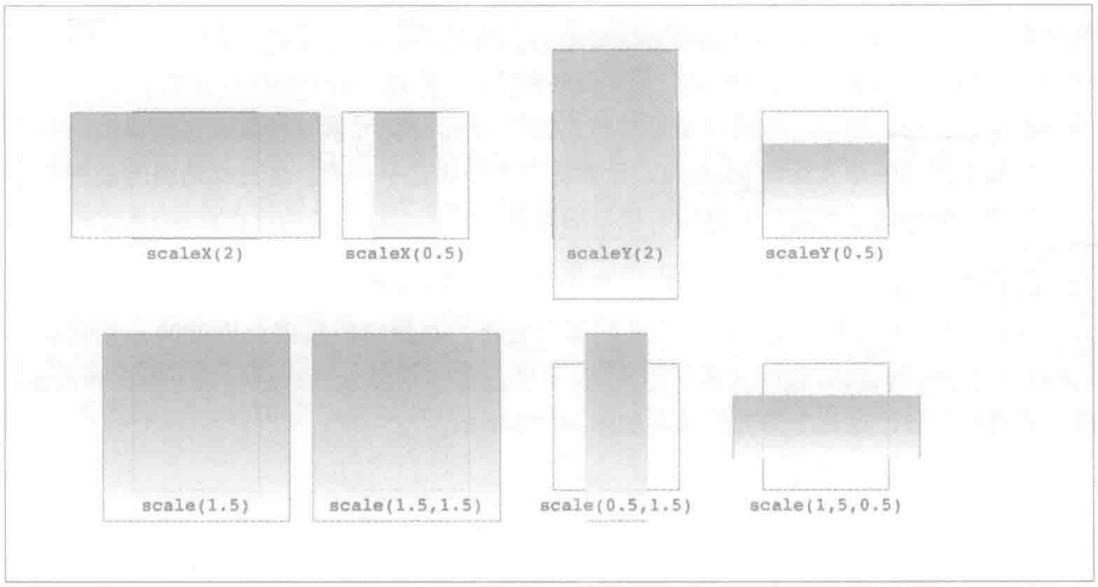


图 16-11：缩放元素

能在二维空间缩放，也就能在三维空间缩放。CSS 提供的 scaleZ() 函数仅在 z 轴上缩放，而 scale3d() 则能同时在三个轴上缩放。当然，仅当元素有深度时，这两个函数才有效果，而元素在默认情况并没有深度。如果让元素具有一定的深度，例如绕 x 轴或 y 轴旋转，那么深度就可以缩放，使用 scaleZ() 或 scale3d() 都可以。

函数	可取的值
scale3d()	<number>, <number>, <number>

与 translate3d() 一样，scale3d() 的三个数都必须是有效的。如若不然，无效的 scale3d() 将导致所属的整个变形值失效。

旋转函数

旋转函数绕某个轴旋转元素，或者绕 3D 空间中的一个向量旋转元素。旋转变形有四个简单的函数，以及一个稍微复杂、专门用于 3D 旋转的函数。

函数	可取的值
rotate(),rotateX(),rotateY(),rotateZ()	<angle>

四个简单的旋转函数都只接受一个值，即角度。角度值以一个数字（可正可负）和一个有效的角度单位（deg、grad、rad 和 turn）表示。如果数字超出了相应单位的常规范围，将化为范围内的值。也就是说，437deg 与 77deg 的效果是相同的，与 -283deg 的效果也是相同的。

然而要注意，仅当没有使用任何形式的动画，这样的换算才是完全等效的。如果以动画的形式旋转 1100deg，元素将转动几周，最终停止在 -20 度（如果喜欢用正数，是 340 度）的倾斜位置。与之相比，如果以动画的形式旋转 -20deg，元素将稍微向左倾斜，而不转动；如果以动画的形式旋转 340deg，元素将向右转动几乎一周。这三次动画的最终状态是一样的，但是每一次旋转的过程有明显的差异。

rotate() 函数实施的是 2D 旋转，是我们最常用的旋转方式。它的效果等同于 rotateZ()，因为都是绕 z 轴（从显示器射出来，直指你的眼睛）旋转的。类似地，rotateX() 绕 x 轴旋转，致使元素向你倾斜，或者远离你倾斜；rotateY() 绕 y 轴旋转元素，像门的开合一样。这几个函数的效果如图 16-12 所示。

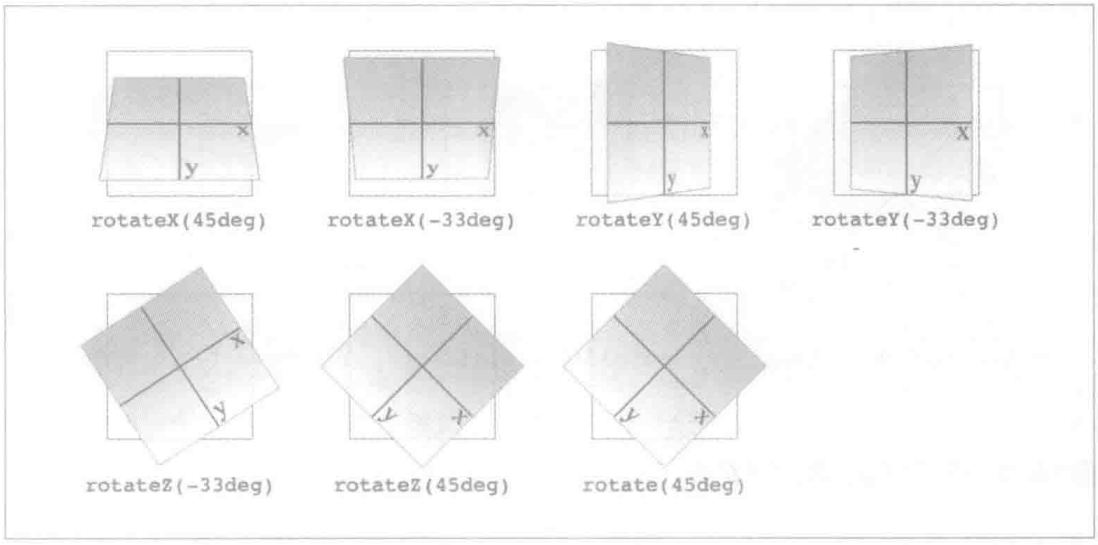


图 16-12：分别绕三个轴旋转



图 16-12 中有几个示例涉及到 3D 效果。这要求 16.3.2 节和 16.3.3 节讨论的 `transform-style` 和 `perspective` 属性设为特定的值，简单起见，这里没有给出。本章涉及的 3D 变形都是如此，如果只应用给出的变形函数是得不到图中所示效果的，切记。

函数	可取的值
`rotate3d()`	*<number>, <number>, <number>, <angle>*

如果你了解向量，想在 3D 空间中旋转元素，使用 `rotate3d()`。前三个值指定 3D 空间中向量的 x、y 和 z 分量，第四个值是角度值，指定绕向量旋转的量。

看个简单的例子：`rotate(45deg)` 用 3D 旋转表示是 `rotate3d(0,0,1,45deg)`。这个向量在 x 轴和 y 轴上的大小是零，在 z 轴上的大小是 1。也就是说，旋转中心是 z 轴。元素将绕指定的向量旋转 45 度，如图 16-13 所示。图中还给出了绕 x 轴和 y 轴旋转 45 度时应该提供给 `rotate3d()` 函数的值。

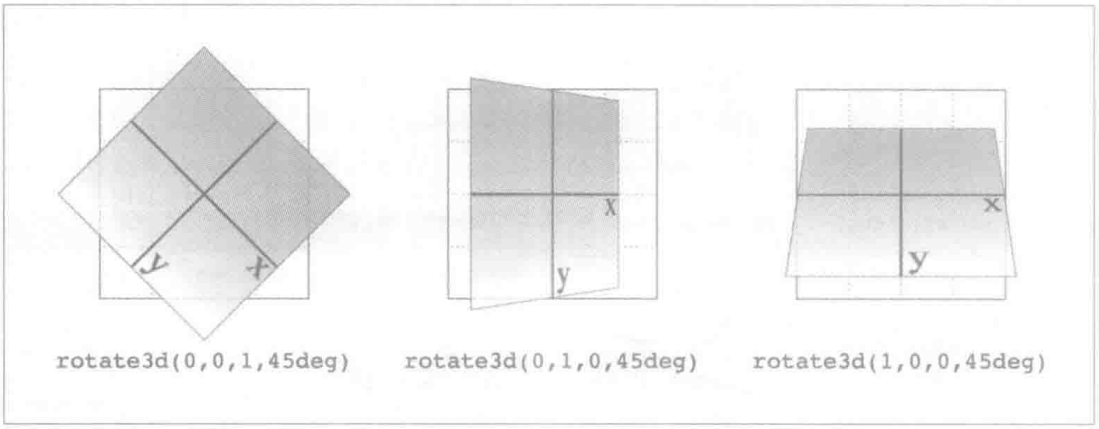


图 16-13：绕 3D 空间中的向量旋转

rotate3d(-0.95,0.5,1,45deg) 要复杂一些，它描述的向量位于轴之间的 3D 空间里。为了弄清处理过程，我们从一个简单的例子入手：rotateZ(45deg)（见图 16-13）。在 3D 空间中，等效于 rotate3d(0,0,1,45deg)。前三个数设定向量的三个分量，可见这个向量在 x 轴和 y 轴上的大小为零，在 z 轴上的大小为 1。因此，这个向量在 z 轴上，指向正方向，即指向观察者。如果看向向量的原点，元素是顺时针旋转的。

类似地，rotateX(45deg) 等效于 3D 空间中的 rotate3d(1,0,0,45deg)。这个向量在 x 轴上，指向正方向（右方）。如果站在向量的终点看向原点，元素绕向量顺时针旋转 45 度。因此，站在常规观察者的位置上，元素的顶部远离观察者，元素的底部靠近观察者。

稍微变复杂一点，假如旋转函数是 rotate3d(1,1,0,45deg)。正视显示器时，这个函数描述的向量从左上角指向右下角，正好经过元素的中心（当然默认情况下是这样，稍后将说明如何转换）。因此，元素所在的矩形框中有一条 45 度斜线。随后，这个向量带着元素一起旋转 45 度。如果看向向量的原点，是顺时针旋转的，因此，同样，元素的顶部远离观察者，底部靠近观察者。如果把旋转值改为 rotate3d(1,1,0,90deg)，在观察者看来，元素是侧立的，倾斜 45 度，正面面向右上角。你可以找张纸试一下，从左上角到右下角画一条直线，然后绕那条线旋转纸张。

好的，现在请在脑中构想 rotate3d(-0.95,0.5,1,45deg) 描述的向量。假设有个边长 200 像素的立方体，那么这个向量在 x 轴上的大小为 190 像素，指向左方，在 y 轴上的大小为 100 像素，指向下方，在 z 轴上的大小为 200 像素，指向观察者。这个向量从原

点 (0,0,0) 指向点 (-190px,100px,200px)，如图 16-14 所示，图中还给出了观察者看到的最终结果。

这个向量就像一根金属棒，穿过旋转的元素。如果沿着向量向回看，元素顺时针旋转了 45 度。但是，由于向量指向左下前方，因此旋转后的元素左上角靠近观察者，而右下角远离观察者，如图 16-14 所示。

请特别注意，`rotate3d(1,1,0,45deg)` 与 `rotateX(45deg)rotateY(45deg)rotateZ(0deg)` 不等效。这是个很容易犯的错误，很多人，包括许多在线教程的作者，以及笔者都犯了这样的错误。本着怀疑一切的原则，笔者在撰写本节时做了一些研究，这才弄清楚。乍一看，二者是等效的，事实却不然。假如我们把这个向量放在前面假想的 200 × 200 × 200 的立方体中，旋转轴将从原点指向偏右 200 像素、偏下 200 像素的 (200,200,0) 点。

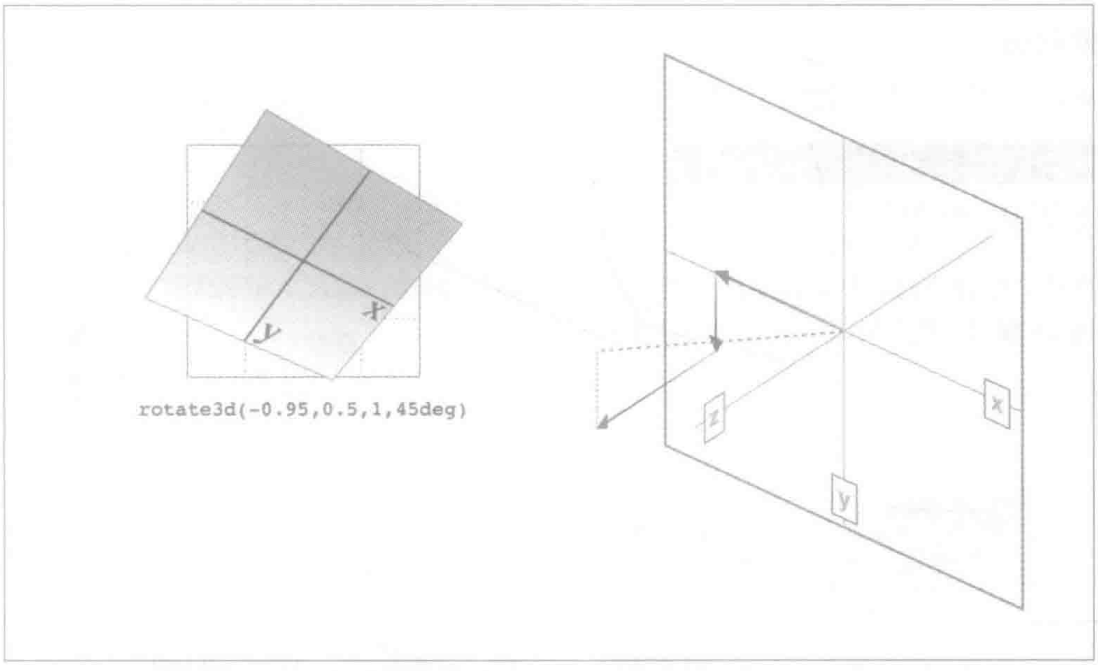


图 16-14：绕 3D 空间中的向量旋转，以及向量的确定方法

可见，旋转轴从元素的左上角射向右下角，呈 45 度角。看向原点（左上角），元素绕对角线顺时针旋转 45 度，因此元素的右上角远离观察者，稍向左偏，而左下角靠近观

察者，稍向右偏。这与 rotateX(45deg)rotateY(45deg)rotateZ(0deg) 的结果明显不同，如图 16-15 所示。

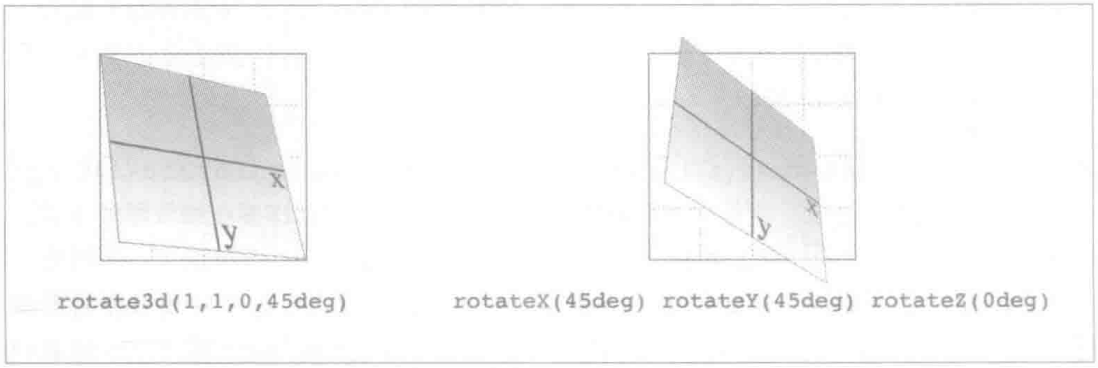


图 16-15：绕两个轴旋转与绕 3D 空间中的向量旋转是有差别的

倾斜函数

倾斜函数沿 x 轴和（或）y 轴倾斜元素。元素不能沿 z 轴或 3D 空间中的向量倾斜。

函数	可取的值
skewX(), skewY()	*<angle>*

这两个函数使元素倾斜指定的角度。文字表达有点抽象，看几个示例你就明白了。图 16-16 展示了几个沿 x 轴和 y 轴倾斜的例子。

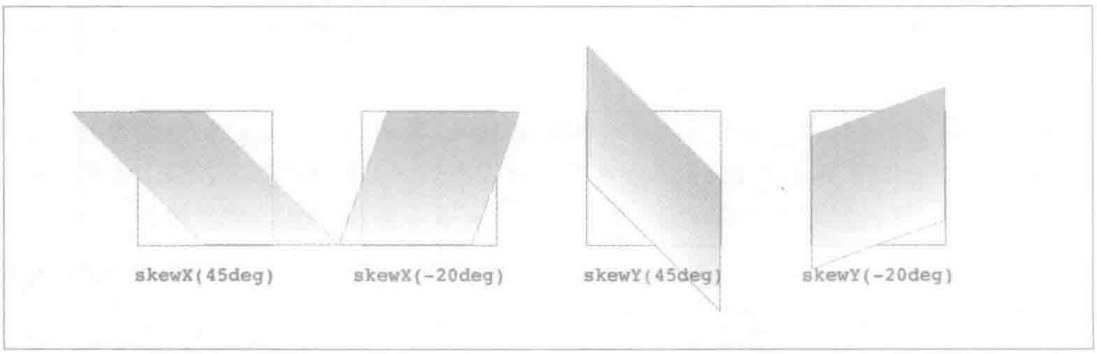


图 16-16：沿 x 轴和 y 轴倾斜

函数	可取的值
skew()	*<angle>* [, *<angle>*]?

skew(a,b) 的效果与 skewX(a)skewY(b) 不同。前者通过矩阵运算 [ax,ay] 实施 2D 倾斜。图 16-17 展示了几个矩阵倾斜的例子，以及与使用两个单轴倾斜变形的结果对比；表面上看结果应该是一样的，事实却不然。

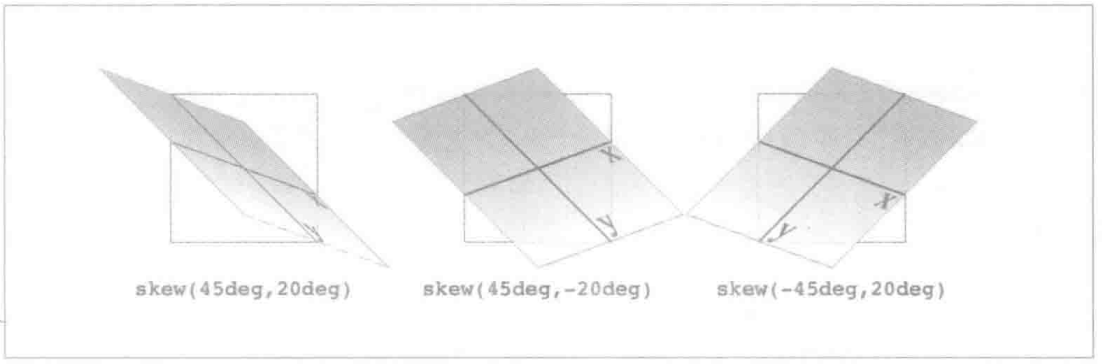


图 16-17：倾斜元素

如果提供两个值，第一个始终是 x 轴的倾斜角度，第二个是 y 轴的倾斜角度。如未指定 y 轴倾斜角度，假定为零。

视域函数

在 3D 空间中改变元素的形态时，基本上都要赋予元素一定的视域。视域为元素赋予前后深度，而这深度可以根据需要设定。

函数	可取的值
perspective()	*<length>*

通过距离指定视域看起来有点奇怪，毕竟 perspective(200px) 设定的距离无法在 z 轴上准确衡量。然而，存在即合理。深度幻象就围绕我们指定的值构建。较小的数得到较极端的视角，就像在元素跟前通过鱼眼镜头看元素一样。较大的数得到较温和的视角，就像从远处通过变焦镜头看元素一样。特别大的视域值会产生等距效应。

这是有一定道理的。如果把视域想象成金字塔，顶点在视域的原点，基底在离你最近的地方，那么顶点与基底之间的距离越短，金字塔越扁，变形效果越失真，如图 16-18 所示。图中假想的金字塔分别表示 200 像素、800 像素和 2000 像素的视域距离。

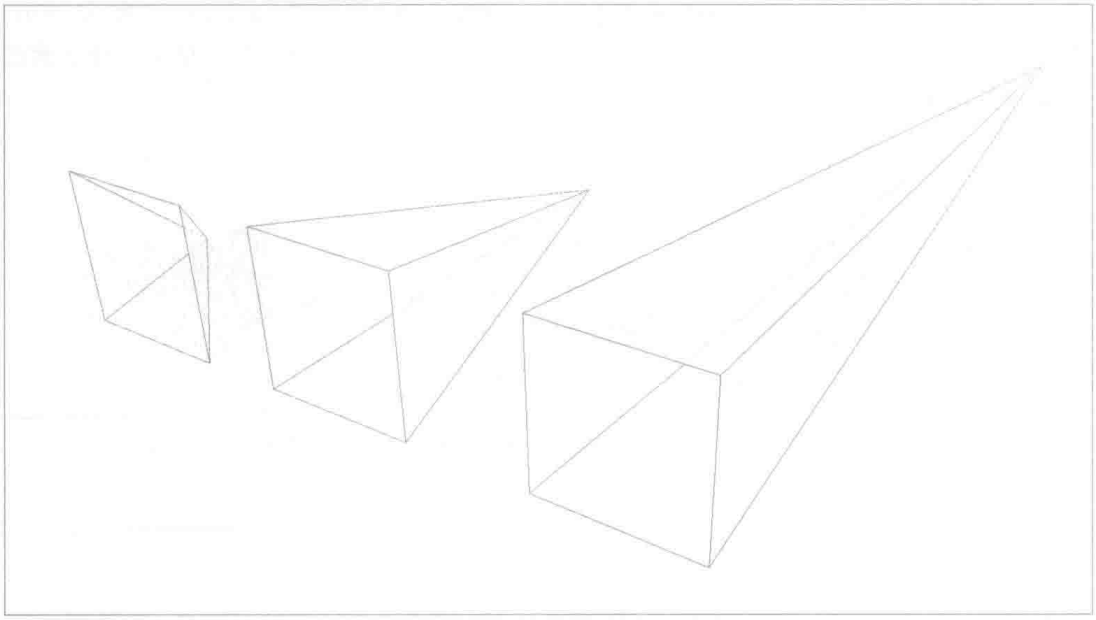


图 16-18：不同视域下的金字塔

在 Safari 的文档中（*https://developer.apple.com/library/archive/documentation/InternetWeb/Conceptual/SafariVisualEffectsProgGuide/Using2Dand3DTransforms/Using2Dand3DTransforms.html*），Apple 指出，小于 300px 的视域值得到的效果特别失真，大于 2000px 的值失真“十分温和”，而 500px 到 1000px 之间的值产生“适中的视域”。图 16-19 展示了相同旋转角度下不同视域值得到的结果。

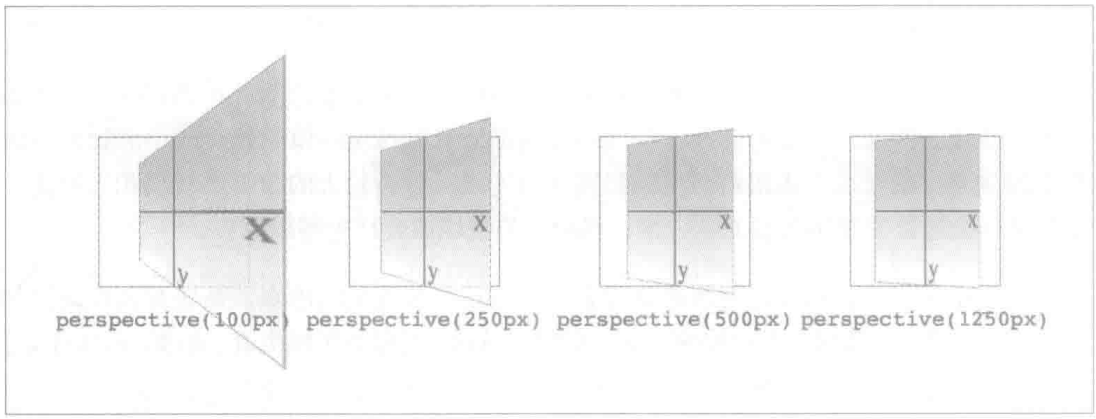


图 16-19：不同视域值的效果

视域值必须是正数，而且不能为零。其他值都将导致 perspective() 函数被忽略。此外要注意，perspective() 函数在变形函数列表中的位置十分重要。查看图 16-19 的代码

你会发现，perspective() 函数在 rotateY() 函数前面。如果调换顺序，尚未应用视域就旋转了，图 16-19 中的四个示例将得到完全相同的结果。因此，如果想在变形函数列表中设定视域值，一定要把 perspective() 函数放在首位，至少也要放在依赖视域的变形之前。这一点特别重要，请牢记，transform 函数的编写顺序十分重要。



注意，perspective() 函数与后文介绍的 perspective 属性十分相似，但是二者的用法完全不同。一般来说，最好使用 perspective 属性，不过也有需要使用 perspective() 函数的例外情况。

矩阵函数

如果你特别喜欢高等数学，或是出自 Wachowski 姐妹的电影[译注 1] 中的老掉牙的笑话，决不能错过本节介绍的函数。

函数	可取的值
matrix()	*<number>* [, *<number>*]{5,5}

CSS 变形规范对 matrix() 函数做了严格规定："以六个值 *a*-*f* 确定的变换矩阵指定 2D 平面中的变形。"

首先要注意，matrix() 函数的有效值是六个以逗号分隔的数字。不能多，也不能少。数字可以为正，也可以为负。其次，matrix() 函数的值所用的句法十分复杂，描述的是元素变形后的最终状态，可以涵盖其他所有变形类型（旋转、倾斜等）。最后，极少有人使用这个句法。

本节不会具体说明矩阵计算的复杂过程。对多数读者来说，就算讲了，也不知所以；而余下那些熟悉这一领域的人，讲了也是多此一举。如果感兴趣，可以在网上找些资料，研究如何计算矩阵。本节只介绍基本句法，以及在 CSS 中的用法。

下面来看矩阵变形的基本原理。假设我们把下述函数应用到一个元素上：

```
matrix(0.838671, 0.544639, -0.692519, 0.742636, 6.51212, 34.0381)
```

描述这个变换矩阵的 CSS 句法如下：

```
0.838671    -0.692519    0    6.51212
```

译注 1：指"黑客帝国"（The Matrix）系列电影。

```
0.544639    0.742636   0   34.0381
0           0          1   0
0           0          0   1
```

可是，这个矩阵有什么用呢？得到的结果如图 16-20 所示，这与下述函数列表的效果完全一样：

```
rotate(33deg) translate(24px,25px) skewX(-10deg)
```

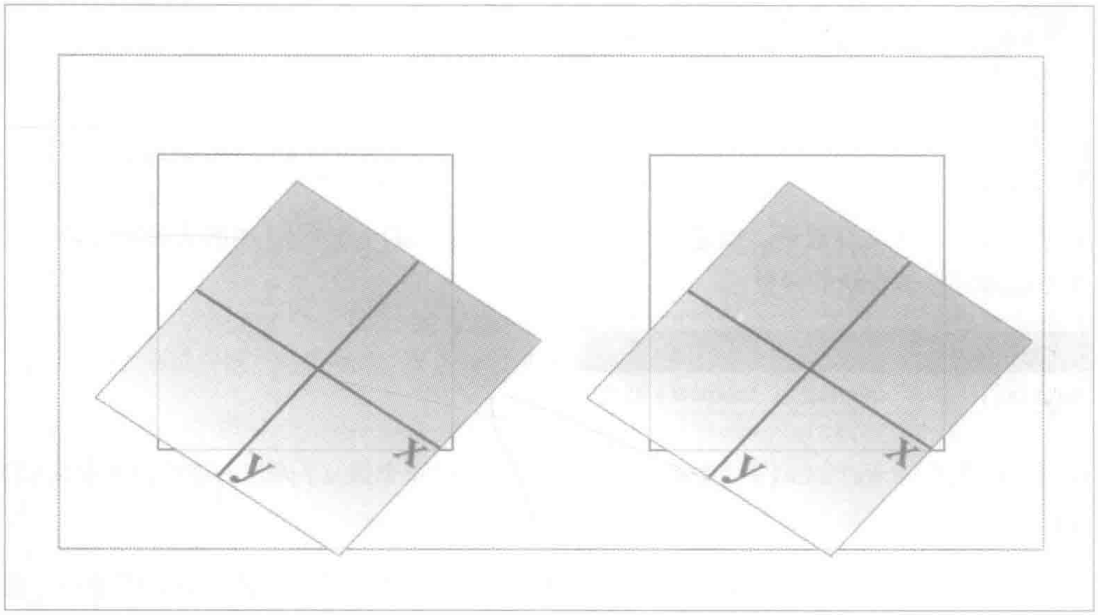


图 16-20：使用矩阵变形的元素及与之等效的变形函数

其实，如果你熟悉或者需要使用矩阵，完全可以也绝对应该使用矩阵。不然，可以使用多个变形函数得到相同的结果，这样更容易理解。

这是 2D 空间中的变形。如果想使用矩阵实施三维变形呢？

函数	可取的值
matrix3d()	<number> [, <number>]{15,15}

同样，我们来看看 CSS 变形规范是如何定义 matrix3d() 的：“以列主序排列一个 4×4 的齐次矩阵，用这 16 个值指定 3D 变形。”这意味着，matrix3d() 函数的值必须是 16 个以逗号分隔的数字，不能多也不能少。这些数字按列序排列成一个 4×4 矩阵，第一列由第一组四个数构成，第二列由第二组四个数构成，第三列由第三组四个数构成，以此类推。因此，对下述函数来说：

```
matrix3d(
    0.838671, 0, -0.544639, 0.00108928,
    -0.14788, 1, 0.0960346, -0.000192069,
    0.544639, 0, 0.838671, -0.00167734,
    20.1281, 25, -13.0713, 1.02614)
```

得到的矩阵为：

```
 0.838671   -0.14788       0.544639     20.1281
 0           1             0            25
-0.544639    0.0960346     0.838671    -13.0713
 0.00108928 -0.000192069  -0.00167734   1.02614
```

最终状态等效于：

```
perspective(500px) rotateY(33deg) translate(24px,25px) skewX(-10deg)
```

如图 16-21 所示。

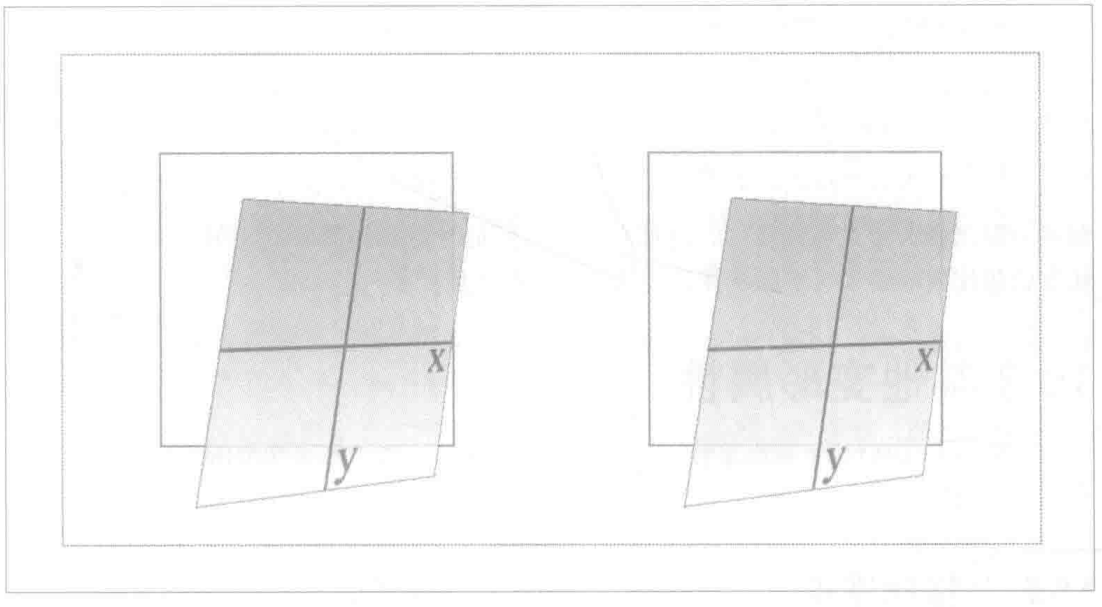


图 16-21：使用三维矩阵变形的元素及与之等效的变形函数

关于最终状态等效的说明

特别注意，matrix() 函数与一系列变形函数等效的只是最终状态。原因与讨论旋转那一节所说的一样：旋转 393deg 与旋转 33deg 得到的最终视觉效果是一样的，但是如果以

动画的形式表示的话，前者将导致元素像滚筒一样旋转，而后者不会。如果用 matrix() 函数实现相同的最终状态，也不会有滚动效果，而是以最简短的路径到达最终状态。

可能有些抽象，下面举个例子。假设下述变形函数列表与 matrix() 函数是等效的：

```
rotate(200deg) translate(24px,25px) skewX(-10deg)
matrix(-0.939693, -0.34202, 0.507713, -0.879385, -14.0021, -31.7008)
```

注意 200 度的旋转。自然而然，我们会把这理解为顺时针旋转 200 度，事实也是如此。然而，如果以动画的形式表示这两次变形，过程是不同的：函数列表版本真的会顺时针旋转 200 度，而 matrix() 版本将逆时针旋转 160 度。两次变形都在相同的位置停止，但方式却不同。

有时看似相同的过程，其实也是有差异的。原因还是由于 matrix() 函数始终取最简短的路径到达最终状态，而变形函数列表则按部就班（其实，也不一定）。请看下面两个貌似等效的变形：

```
rotate(160deg) translate(24px,25px) rotate(-30deg) translate(-100px)
matrix(-0.642788, 0.766044, -0.766044, -0.642788, 33.1756, -91.8883)
```

这两个变形也在相同的位置停止，但是，如果以动画的形式表示，到达最终状态的过程是不一样的。乍一看，区别不是那么明显，但确实有差异。

倘若不以动画的形式表示变形，这都不重要，但是一定要知道这个区别，说不定什么时候你就想使用动画了（但愿是在读过动画那一章之后）。

16.3 其他变形属性

除了基本的 transform 属性之外，还有几个辅助属性，用于定义变形的原点、“场景”使用的视域等。

16.3.1 移动原点

目前所见的变形有个共同点，都以元素的绝对中心为变形的原点。例如，旋转元素时，是绕着中心旋转的，而不是绕着某一角旋转的。这是默认行为，不过可以使用 transform-origin 属性修改。

transform-origin

取值	[left | center | right | top | bottom | *<percentage>* | *<length>*] | [left | center | right | *<percentage>* | *<length>*] && [top | center | bottom | *<percentage>* | *<length>*]] *<length>*?
初始值	50% 50%
适用于	任何可变形的元素
百分数	相对范围框计算（见下文的说明）
计算值	计算为一个百分数；值为长度值时，计算为绝对长度
继承性	否
动画性	*<length>*, *<percentage>*

取值句法看起来错综复杂，但实际使用起来还是比较简单的。transform-origin 属性的值为两个或三个关键字，用于定义相对哪个点变形：第一个值针对横向，第二个值针对纵向，可选的第三个值是 z 轴上的长度。横轴和纵轴可以使用英语关键字，例如 top 和 right，也可以使用百分数、长度，或者不同类型的关键字搭配。然而 z 轴不能使用英语关键字或百分数，不过可以使用长度值，其中像素值是目前最常用的。

长度值设定的是距元素左上角的距离。因此，transform-origin: 5em 22px 定义的变形原点距元素的左边 5 em、距元素的顶边 22 像素。类似地，transform-origin: 5em 22px 200px 定义的变形原点右移 5em、下移 22 像素、后移 200 像素（即元素所在位置背后 200 像素）。

百分数相对对应的轴和元素的尺寸计算，设定的是距元素左上角的偏移量。例如，transform-origin: 67% 40% 定义的元素距元素左边的距离为宽度的 67%，距元素顶边的距离为高度的 40%。图 16-22 展示了几种定义原点的方式。

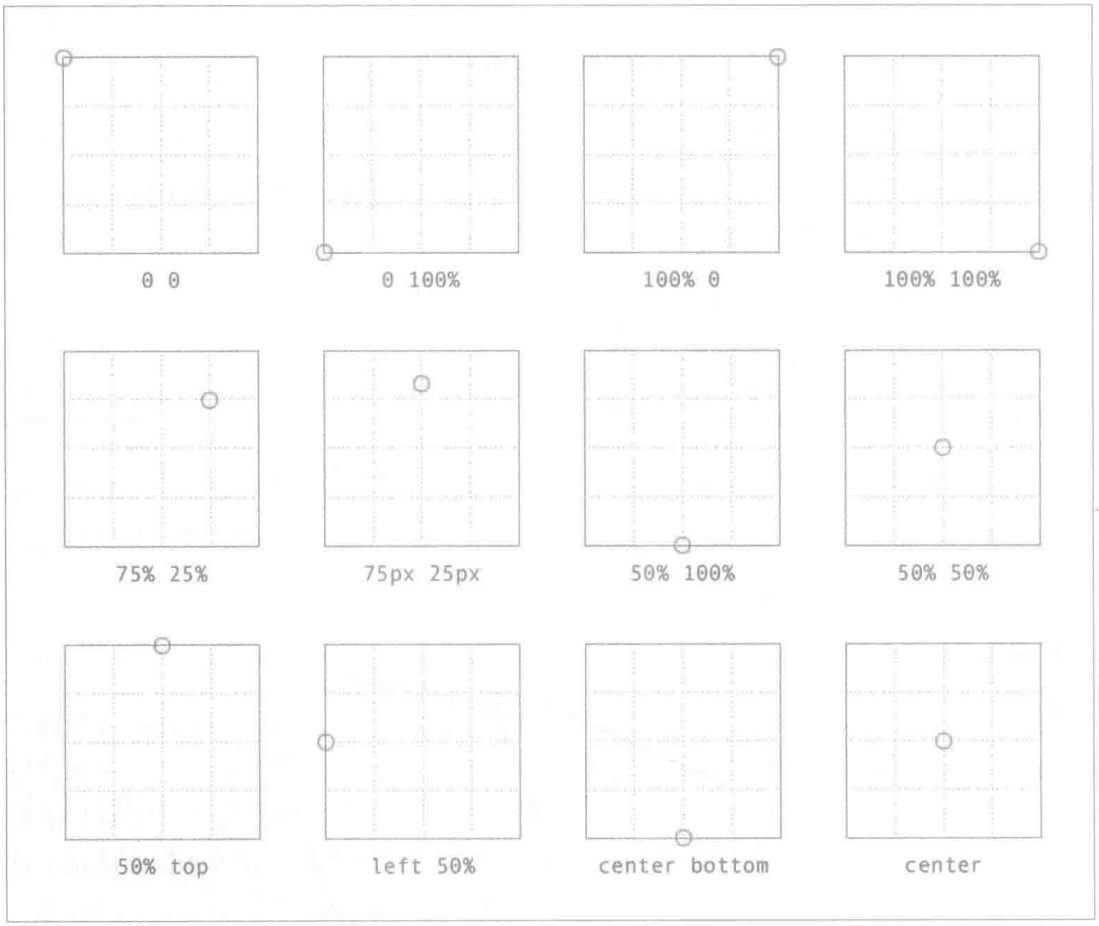


图 16-22：定义原点的不同方式

那么，修改原点的位置有什么用呢？通过 2D 旋转最容易说明其作用。假设我们把一个元素向右旋转 45 度，那么元素的最终位置就取决于变形原点。图 16-23 展示了使用不同的变形原点得到的结果。图中以圆圈标出的是变形原点。

原点对其他变形类型也有影响，例如倾斜和缩放。如果原点在元素的中心，缩放时，每一边缩放的量相等；而原点在右下角的话，元素将向那一角收缩。类似地，相对元素的中心倾斜与相对右上角倾斜得到的形状是一样的，但是位置不同。图 16-24 中有几个例子。同样，图中以圆圈标出的是变形原点。

有种变形不怎么受变形原点的影响——平移。使用 `translate()`，或者 `translateX()` 和 `translateY()` 移动元素，不管变形原点在哪儿，元素最终都将被移到相同的位置。如果只做平移，设定变形原点就没什么意义了。然而，除了平移之外，如果还想做其他变形，那原点的位置就有影响了。请根据实际情况灵活使用。

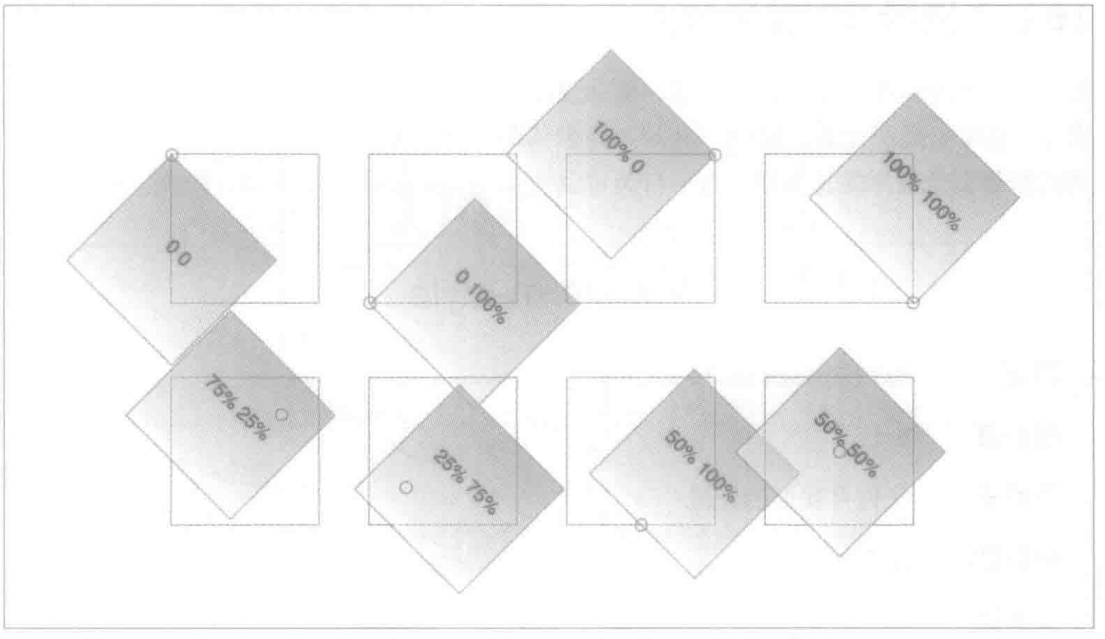


图 16-23：变形原点位于不同位置时的旋转效果

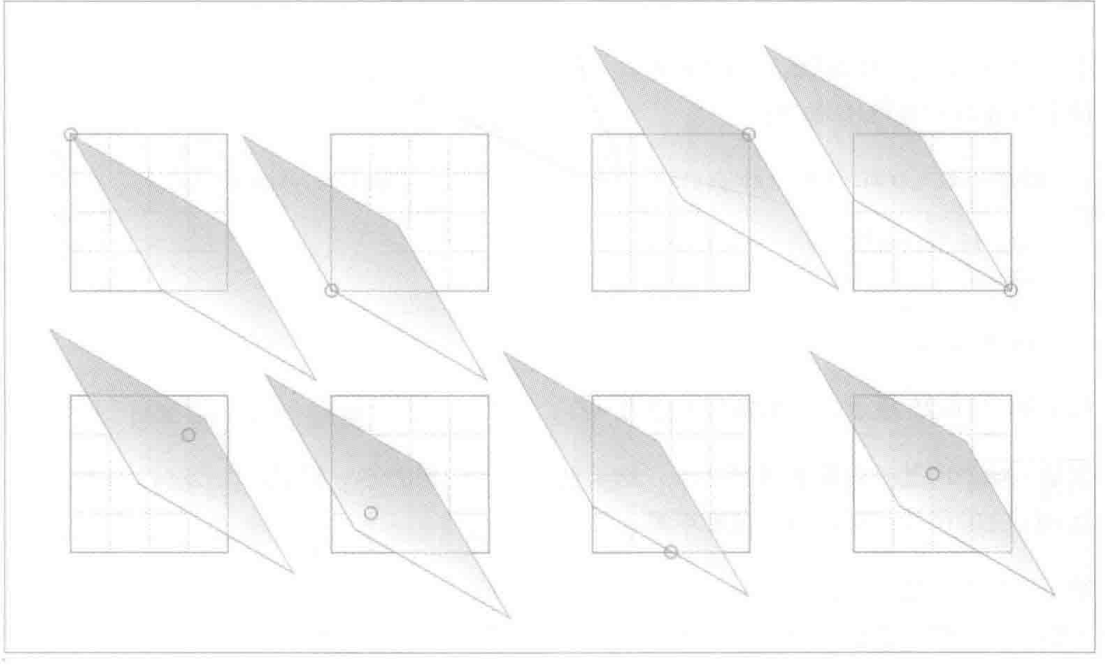


图 16-24：变形原点位于不同位置时的倾斜效果

16.3.2 选择 3D 变形方式

如果在三维空间中改变元素的形态，例如使用 translate3d() 或 rotateY()，或许希望在 3D 空间中呈现元素。然而，这却不是默认的行为。默认情况下，不管怎样变形，得到的结果都是扁平的。幸好，这个行为可以使用 transform-style 属性修改。

transform-style	
取值	flat | preserve-3d
初始值	flat
适用于	任何可变形的元素
计算值	指定的值
继承性	否
动画性	否

假设我们想在适中的视域下移动元素，让元素离我们的眼睛“更近”一些，然后倾斜一点。使用的规则和 HTML 如下：

```
div#inner {transform: perspective(750px) translateZ(60px) rotateX(45deg);}

<div id="outer">
outer
<div id="inner">inner</div>
</div>
```

结果如图 16-25 所示，差不多与所想的一样。

但是，如果向某一边旋转外层 div，结果就与我们设想的不一样了：内层 div 像粘在外层 div 上的照片一样，这与预期不符。

然而，结果就是这样，因为 transform-style 的默认值是 flat。内层 div 的上部前倾、下部后靠，像是紧贴在外层 div 上的图像，随外层 div 一起旋转，如图 16-26 所示。

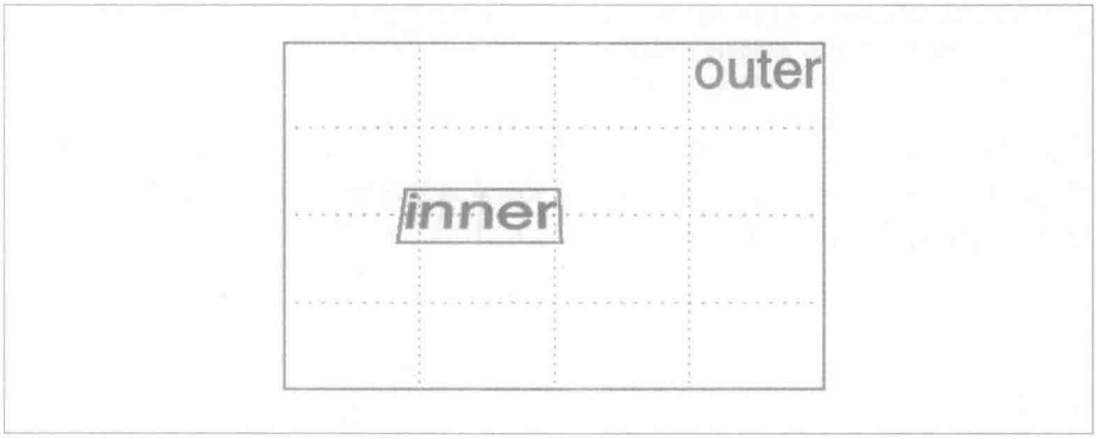


图 16-25：3D 变形内层 div

```
div#outer {transform: perspective(750px) rotateY(60deg) rotateX(-20deg);}
div#inner {transform: perspective(750px) translateZ(60px) rotateX(45deg);}
```

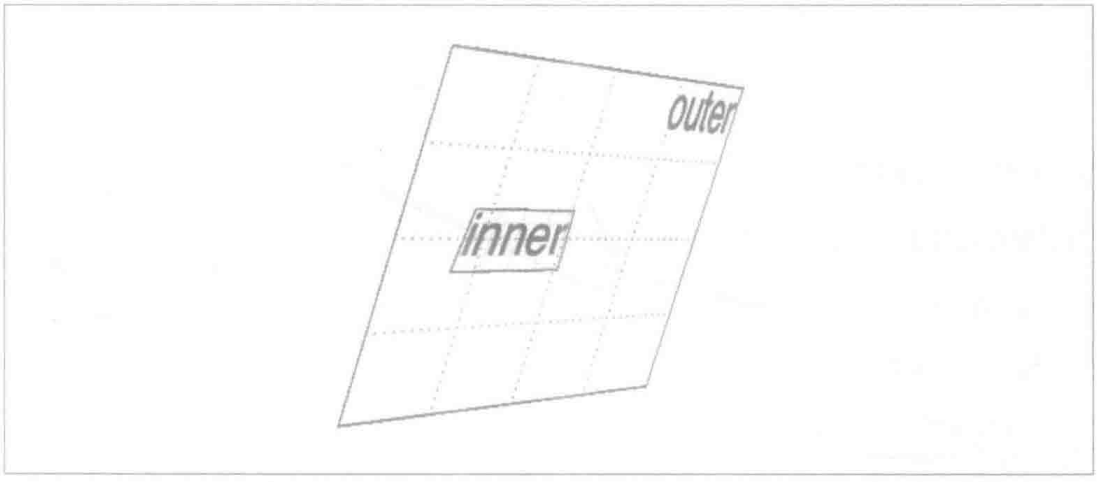


图 16-26：扁平变形方式的效果

然而，把值改成 preserve-3d 后，结果就截然不同了。内层 div 将绘制成一个 3D 对象，浮动在外层 div 附近，因此也就不紧贴在外层 div 上了。改动后得到的结果如图 16-27 所示。

```
div#outer {transform: perspective(750px) rotateY(60deg) rotateX(-20deg);
    transform-style: preserve-3d;}
div#inner {transform: perspective(750px) translateZ(60px) rotateX(45deg);}
```

注意，transform-style 设定的变形方式可能会被其他属性覆盖。这是因为那些属性的某些值要求元素及其子元素必须以扁平的方式呈现才能起作用。遇到这样的情况，不管你把 transform-style 设为什么值，都会被强制重置为 flat。

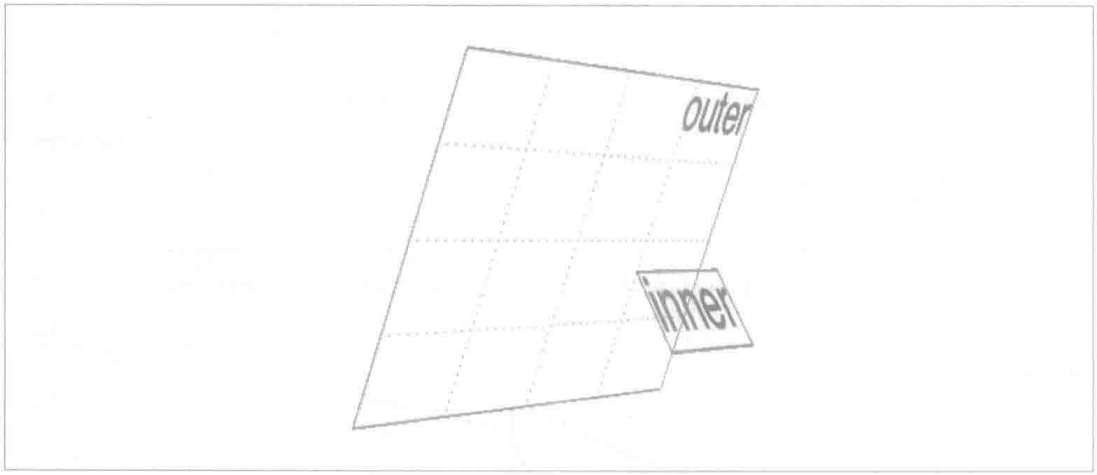


图 16-27：保留 3D 效果的变形

为了避免被覆盖，下述属性要设为给出的值：

- overflow: visible
- filter: none
- clip: auto
- clip-path: none
- mask-image: none
- mask-border-source: none
- mix-blend-mode: normal

这些都是相应属性的默认值，如果没在保留 3D 效果的元素上声明为其他值，那就没什么需要担心的。然而，如果在编辑 CSS 的过程中，突然发现 3D 变形效果被拍扁了，根源可能就在上述列表中的某个属性上。

此外，除了前面提到的值以外，isolation 属性的值必须是或者计算为 isolate（想知道 isolation 属性的作用？用于设定合成方式）。

16.3.3 修改视域

视域其实由两个属性定义：一个定义视域距离，相当于前面讨论过的 perspective() 函数；另一个定义视域的原点。

定义视域

先讲 perspective 属性。这个属性的值是一个长度，定义视域锥体的深度。这么看来，它与前面讨论的 perspective() 函数功能类似，不过二者之间有重要的区别。

perspective

取值	none | *<length>*
初始值	none
适用于	任何可变形的元素
计算值	绝对长度，或者 none
继承性	否
动画性	是

举个简单的例子，如果想创建特别深的视域，仿照变焦镜头的效果，可以声明 perspective: 2500px；如果想让深度浅一些，模仿鱼眼镜头的近景效果，可以声明 perspective:200px。

那么它与 perspective() 函数之间到底有什么不同呢？perspective() 函数只为目标元素定义视域，比如声明 transform:perspective(800px)rotateY(-50grad);，那么只有应用这个规则的元素才使用设定的视域。

而 perspective 属性定义的视域深度应用到目标元素的所有子元素上。糊涂了？没关系。下面举例说明这个区别，结果如图 16-28 所示：

```
div {transform-style: preserve-3d; border: 1px solid gray; width: 660px;}
img {margin: 10px;}
#one {perspective: none;}
#one img {transform: perspective(800px) rotateX(-50grad);}
#two {perspective: 800px;}
#two img {transform: rotateX(-50grad);}

<div><img src="rsq.gif"><img src="rsq.gif"><img src="rsq.gif"></div>
<div id="one"><img src="rsq.gif"><img src="rsq.gif"><img src="rsq.gif"></div>
<div id="two"><img src="rsq.gif"><img src="rsq.gif"><img src="rsq.gif"></div>
```

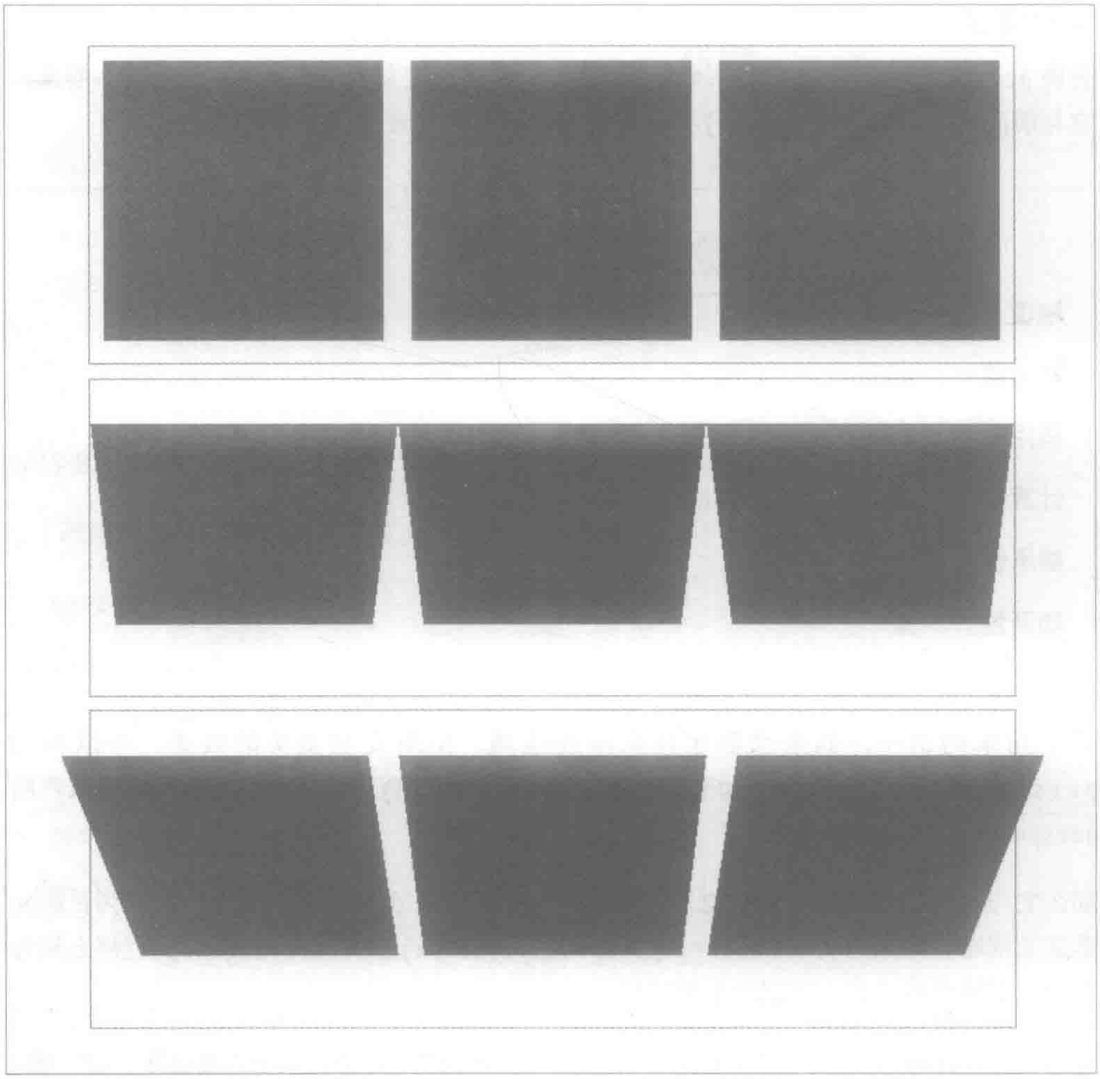


图 16-28：共有视域与独立视域

图 16-28 中的第一排图像还没有变形，第二排中的每个图像向我们旋转了 50 弧度（等同于 45 度），不过每个图像的视域是独立的。

在第三排中，各个图像自身都没有视域。这一排中的图像都在为 div 容器设定的视域（由 perspective:800px; 定义）中绘制。正因为这些图像在共有的视域中变形，所以看起来是“正确的”，即与预期一致，就像把三个实物照片贴在透明玻璃片上，绕着玻璃片中间的横轴向我们旋转一样。



注意，前一节说过，设定 transform-style:preserve-3d 才能看到图中的效果。

这就是 perspective 属性与 perspective() 函数之间重要的区别。前者创建的 3D 空间为所有子元素共有，而后者只对目标元素有效果。而且，为了让 3D 空间中的变形有视觉效果，要把 perspective() 函数放在变形函数列表的开头或前部。然而，perspective 属性应用到全部子元素上，对在何处声明变形没有要求。

多数时候，应该使用 perspective 属性。其实，3D 变形经常使用页面布局中惯用的 div 容器（或其他元素），目的主要是提供共有视域。在前面的示例中， 的唯一作用就是充当视域容器。没有这个容器，图中的效果也就无从谈起。

移动视域的原点

如果允许以 3D 形式呈现，元素在三维空间中的变形将使用视域（参见前文的 transform-style 和 perspective 属性）。视域有原点，也称消隐点（vanishing point），这个点的位置可以使用 perspective-origin 属性修改。

perspective-origin

取值	[left | center | right | top | bottom | *<percentage>* | *<length>*] | [left | center | right | *<percentage>* | *<length>*] && [top | center | bottom | *<percentage>* | *<length>*]] *<length>*?
初始值	50% 50%
适用于	任何可变形的元素
百分数	相对范围框计算（见下文的说明）

计算值	计算为一个百分数，长度值计算为绝对长度
继承性	否
动画性	*<length>, <percentage>*

你可能发现了，perspective-origin 的取值句法与 transform-origin 一样，而且最后也有个可选的长度值，定义在 z 轴上的偏移量。虽然值的表述是一样的，但是效果却截然不同。transform-origin 定义的是围绕哪个点变形，而 perspective-origin 定义的是视线汇聚于哪一点。

与其他多数 3D 变形属性一样，这个属性的作用通过演示更容易说明白。请看下述 CSS 和标记，结果如图 16-29 所示：

```
#container {perspective: 850px; perspective-origin: 50% 0%;}
#ruler {height: 50px; background: #DED url(tick.gif) repeat-x;
    transform: rotateX(60deg);
    transform-origin: 50% 100%;}

<div id="container">
    <div id="ruler"></div>
</div>
```

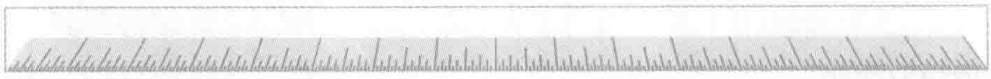


图 16-29：一个简单的“标尺”

我们用重复的背景图在尺子上标出刻度，刻度所在的 div 容器向后倾斜 60 度。标尺上的刻度线都指向同一个消隐点，即 div 容器顶边的中点（因为 perspective-origin 的值是 50% 0%）。

下面我们在同样的条件下变更视域原点的位置，如图 16-30 所示。

可以看到，视域原点位置的变化对 3D 变形元素的渲染是有影响的。

注意，之所以能看到图中的效果，是因为我们声明了 perspective。倘若 perspective 的值是默认值 none，不管把 perspective-origin 设为什么，都将被忽略。这是说得通的，毕竟没有视域也就没有视域原点。

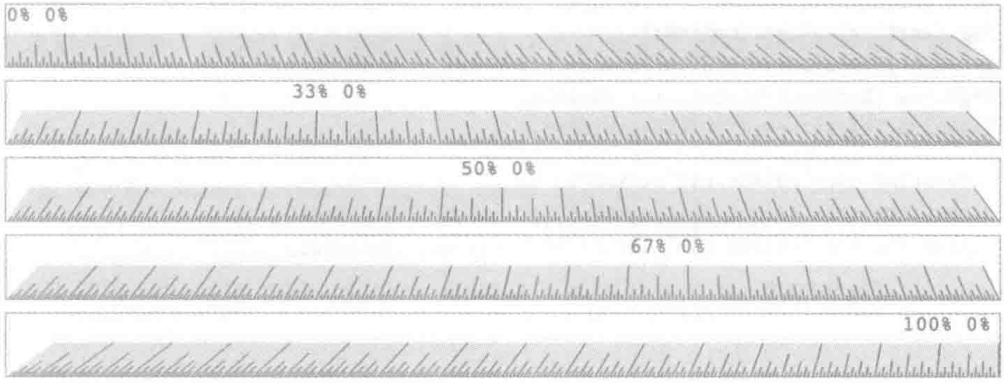


图 16-30：不同视域原点下的“标尺”

16.3.4 处理背面

排布元素这么多年来，你可能从未想过，如果能看到元素的背面，背面是什么样子？在使用 3D 变形的过程中，有朝一日完全有可能会看到元素的背面。说不定还会故意这么做。这样的情况如何处理，由 backface-visibility 属性决定。

backface-visibility

取值	visible | hidden
初始值	visible
适用于	任何可变形的元素
计算值	指定的值
继承性	否
动画性	否

这个属性没有前面讨论的属性和函数那么复杂，它的作用只有一个，即决定元素的背面朝向我们时是否渲染背面，仅此而已。

假设翻转两个元素，一个元素的 backface-visibility 属性设为默认值 visible，另一个设为 hidden，得到的结果如图 16-31 所示。

```
span {border: 1px solid red; display: inline-block;}
img {vertical-align: bottom;}
img.flip {transform: rotateX(180deg); display: inline-block;}
img#show {backface-visibility: visible;}
img#hide {backface-visibility: hidden;}
```

```
<span><img src="salmon.gif"></span>
<span><img src="salmon.gif" class="flip" id="show"></span>
<span><img src="salmon.gif" class="flip" id="hide"></span>
```

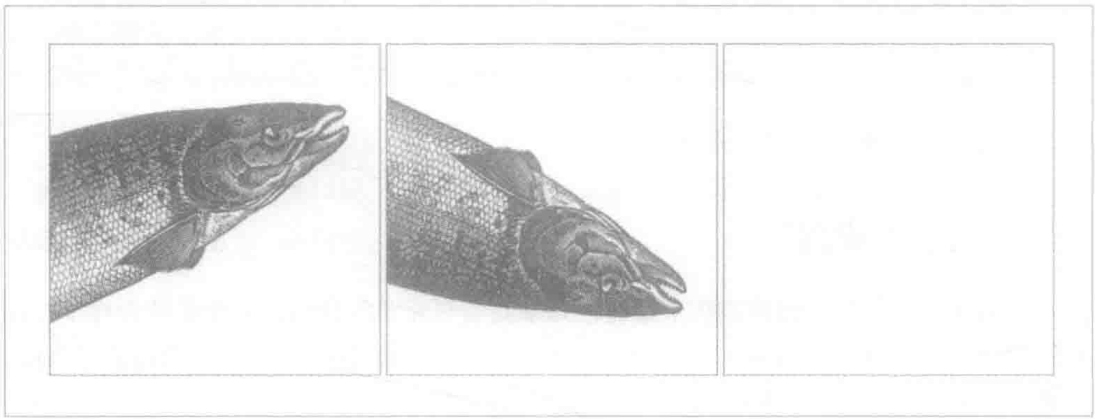


图 16-31：显示和隐藏背面

可以看到，第一个图像没变化，第二个图像沿 x 轴翻转了，而且能看到背面。第三个图像也翻转了，但是看不到，因为背面隐藏了。

这个属性在一些场合下还是很有用的。最简单的情况是一个能翻转的 UI 元素，在两面呈现不同的内容。比如一个搜索区域，在背面显示首选项配置，或者一张照片，在背面记一些信息。我们来看后一种情况。所用的 CSS 和标记可能是下面这样的：

```
section {position: relative;}
img, div {position: absolute; top: 0; left: 0; backface-visibility: hidden;}
div {transform: rotateY(180deg);}
section:hover {transform: rotateY(180deg); transform-style: preserve-3d;}
```

```
<section>
    <img src="photo.jpg" alt="">
    <div class="info">(...info goes here...)</div>
</section>
```

这个例子表明，backface-visibility 使用起来没有想得那么简单。不是说 backface-visibility 属性本身有多复杂，而是如果忘记把 transform-style 设为 preserve-3d 的话，根本看不到效果。正是因为这样，我们才为 section 元素声明了 transform-style。

在相同的标记下，我们可以稍微修改一下CSS，在翻转后显示图像的背面。这样可能更符合我们的意图，让信息看起来真的是写在图像的背面。修改后得到的结果如图16-32所示。

```
section {position: relative;}
img, div {position: absolute; top: 0; left: 0;}
div {transform: rotateY(180deg); backface-visibility: hidden;
    background: rgba(255,255,255,0.85);}
section:hover {transform: rotateY(180deg); transform-style: preserve-3d;}
```

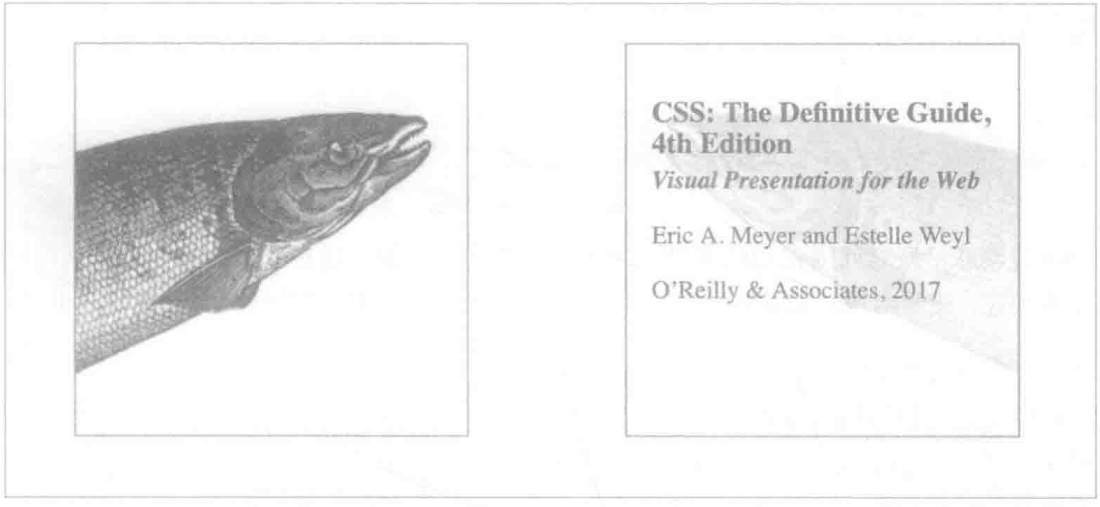


图16-32：正面是照片，背面是信息

为了达到这样的效果，我们只需把backface-visibilty:hidden应用到div上，而不是同时应用到img和div上。这样，翻转后，div的背面隐藏了，而图像的背面不隐藏。

16.4 小结

CSS变形功能能在二维和三维空间中改变元素的形态，这为设计师提供了新的呈现信息的方式。使用丰富的2D变形可以实现引人注目的效果，利用3D变形还能创建交互式界面，变形功能为设计开辟了一片新天地。有些属性之间有依赖关系，这对刚接触CSS变形的创作人员来说可能有些陌生，但是用得多了也会习惯的。

第 17 章

过渡

CSS 过渡在一段时间内把 CSS 属性的初始值变为另一个值。这种状态的转变是对某种操作的响应，通常由用户交互触发，不过也可能由脚本对类、ID 或其他状态的改变而引起。

通常，CSS 属性值的变化（发生“样式变化事件”）瞬间完成。属性的新值替换旧值持续的时间是毫秒级的，受影响的内容在这短暂的时间内重新绘制，若有必要，先重排布局再重新绘制。多数情况下，值的变化是瞬间完成的，整个过程不超过 16 毫秒[注 1]。即便所用的时间较长，也是从一个值直接变为另一个值。例如，鼠标悬停时改变背景色，背景直接从一个颜色变为另一个颜色，中间没有过渡。

17.1 CSS 过渡

CSS 过渡能控制一段时间内属性的值如何变成另一个值。因此，我们可以让属性的值逐渐变化，自然一些，不那么突兀。例如：

```
button {
    color: magenta;
    transition: color 200ms ease-in 50ms;
}

button:hover {
    color: rebeccapurple;
```

注 1：变换背景图的时间可能会超过 16 毫秒，因为要解码并重新绘制页面。但这不属于过渡，而且会降低性能。

```
    transition: color 200ms ease-out 50ms;
}
```

在这个例子中，当鼠标悬停在按钮上时，文本的颜色不是瞬间变化的。我们利用CSS过渡，让按钮文本的颜色在200毫秒内由`magenta`逐渐变为`rebeccapurple`，而且开始过渡前还有50毫秒延迟。颜色的变化，不管耗时多久，都是过渡。加上CSS属性`transition`后，颜色在一段时间内逐渐变化，而且肉眼能察觉到变化。

就算还想支持IE9或更旧版本的浏览器，现在也可以使用CSS过渡。如果浏览器不支持CSS过渡相关的属性，变化立即完成，而不逐渐过渡，对最终结果完全没有影响。同样，如果属性或提供给属性的值不支持动画，变化也立即完成，而不逐渐过渡。



这里所说的“支持动画”，是指属性可以通过过渡或动画（下一章的话题）以动画的形式表示值的变化。支持动画的属性参见附录A。

有时，我们想瞬间改变值。虽然前面用链接颜色举例，但是链接颜色通常在悬停时是立即改变的，知会看得见的用户发生了交互，表明悬停其上的元素是链接。类似地，在自动补全的列表框中，不能慢慢显现选项，而是要立即把选项显示出来，不能比用户输入的速度慢。立即改变值通常能提供最好的用户体验。

另一些情况下，你可能想逐渐改变属性的值，让用户注意即将发生的事。例如，为了让纸牌游戏更贴近现实，可能会在200毫秒的时间段内以动画的形式表示翻牌动作；倘若没有动画，用户可能察觉不到发生了什么▶。



“播放”图标▶表示有在线示例。本章的所有示例可在这个页面获取：*https://meyerweb.github.io/csstdg4figs/17-transitions/*。

再举个例子，你可能想在200毫秒的时间段内展开或呈现下拉菜单（如果立即展开，不太协调）。利用过渡功能，可以让下拉菜单慢慢显示出来。在图17-1中▶，我们过渡的是缩放变形对子菜单高度的改变。CSS变形经常这么用，详情参见本章后文。

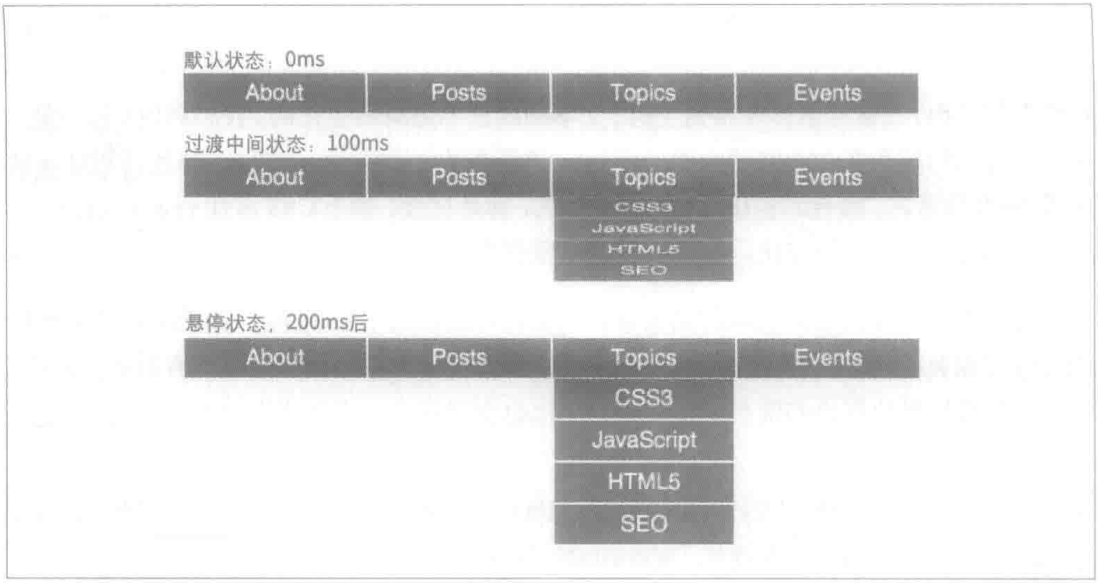


图 17-1：过渡的初始状态、中间状态和最终状态

17.2 定义过渡的属性

在 CSS 中，过渡使用四个属性定义：transition-property、transition-duration、transition-timing-function 和 transition-delay。此外，还有个简写属性 transition，可一次声明全部四个属性。

为了实现图 17-1 中的下拉导航栏，四个 CSS 过渡属性都使用了，此外还用其他属性定义了过渡的始态和终态。图 17-1 中的过渡效果是用下述代码定义的：

```
nav li ul {
    transition-property: transform;
    transition-duration: 200ms;
    transition-timing-function: ease-in;
    transition-delay: 50ms;
    transform: scale(1, 0);
    transform-origin: top center;
}
nav li:hover ul {
    transform: scale(1, 1);
}
```

注意，这个过渡示例是用 :hover 状态触发样式变化事件的，除此之外还可以通过其他方式触发。例如，可以添加或删除类，也可以改变状态，比如把输入框的状态由 :invalid

变为 :valid，或由 :checked 变成 :not(:checked)。甚至，还可以利用 :nth-last-of-type 等选择符在斑马纹表格末尾追加一行，或者在列表的末尾追加一个列表项目。

在图 17-1 中，嵌套列表的始态为 transform:scale(1,0)，而且使用 transform-origin: top center 设置了原点；终态为 transform:scale(1,1)，变形原点保持不变。



变形属性的详细说明参见第 16 章。

在这个例子中，过渡的属性是 transform，触发 hover 事件时，在 200 毫秒的时段内由旧值 transform:scale(1,0) 平滑过渡到新值 transform:scale(1,1)，即嵌套的无序列表放大到默认的原始尺寸。这个过渡延迟 50 毫秒开始，整个过程是“渐入式”的，起初速度较慢，然后逐渐加快。

过渡就在应用到元素上的常规样式中声明。当目标属性的值发生变化时，如果为目标属性设置了过渡效果，浏览器将应用过渡效果，逐渐由旧值变成新值。

注意，所有与过渡有关的属性都在 ul 元素的非悬停状态上声明。悬停状态仅用于改变变形效果，而不是过渡效果。这样做的好处是，悬停时目录将下拉打开，而且悬停状态结束后目录还会上拉关闭。

如果像下面这样把过渡相关的属性应用到悬停状态上：

```
nav li ul {
    transform: scale(1, 0);
    transform-origin: top center;
}
nav li:hover ul {
    transition-property: transform;
    transition-duration: 200ms;
    transition-timing-function: ease-in;
    transition-delay: 50ms;
    transform: scale(1, 1);
}
```

非悬停状态下，元素将使用默认的过渡值，即瞬间过渡。对前例来说，悬停时菜单将下拉打开，但是悬停状态结束后菜单立即消失，这是因为，在非悬停状态下，过渡属性不再应用到元素上。

当然，你可能就想要这样的效果，即慢慢下拉打开，但是立即消失。如果真是这样，过渡效果就要应用到悬停状态上。否则，应该把过渡属性直接应用到元素上，这样在进入和退出悬停状态时都有过渡效果。退出状态时，过渡时序是反过来的。反向过渡是默认行为，如果想覆盖，可以在始态和终态声明不同的过渡效果。

这里所说的“始态”是指页面加载时元素所处的状态。始态可能是元素始终处于的状态，例如在元素选择符上设置的属性，而元素的 `:hover` 状态就不是始态。始态也可能是可编辑内容的元素获得焦点时的状态（`:focus`），例如：

```
/* 匹配元素所有状态的选择符 */
p[contenteditable] {
    background-color: rgba(0, 0, 0, 0);
}

/* 匹配元素部分状态的选择符 */
p[contenteditable]:focus {
    /* 覆盖声明 */
    background-color: rgba(0, 0, 0, 0.1);
}
```

在这个示例中，始态始终是完全透明的背景，仅当用户让元素获得焦点才会变化。这就是本章一直说的始态或默认值。不管状态怎么变化，比如从始态变为终态（在前例中是获得焦点），匹配元素所有状态的选择符中声明的过渡属性对元素一直有影响。

始态还可能是可变的临时状态，例如复选框的 `:checked`，或者表单控件的 `:valid`，又或者类的增删：

```
/* 匹配元素部分状态的选择符 */
input:valid {
    border-color: green;
}

/* 前一个选择符不匹配时，匹配元素部分状态的选择符 */
input:invalid {
    border-color: red;
}

/* 匹配元素部分状态的选择符，不管输入框中的内容是否有效 */
input:focus {
    /* 互斥的声明 */
    border-color: yellow;
}
```

在这个示例中，:valid 和 :invalid 选择符都能匹配部分状态下的元素，但是不能同时匹配。而 :focus 选择符能匹配任何情况下获得焦点的输入框，不管输入的内容同时还有没有匹配 :valid 或 :invalid 选择符，如图 17-2 所示。

这种情况下，始态指初始值，可能是 :valid，也可能是 :invalid。而终态与始态（:valid 或 :invalid）相反。▶



图 17-2：输入框在有效、无效和获得焦点状态下的外观

记住，始态和终态可以应用不同的过渡值，但是进入某一状态时一定会使用当前状态定义的值。请看下面的示例代码，这里设置的过渡在 2 秒的时段内下拉打开菜单，但是上拉关闭只用 200 毫秒：

```
nav li ul {
    transition-property: transform;
    transition-duration: 200ms;
    transition-timing-function: ease-in;
    transition-delay: 50ms;
    transform: scale(1, 0);
    transform-origin: top center;
}
nav li:hover ul {
    transition-property: transform;
    transition-duration: 2s;
    transition-timing-function: linear;
    transition-delay: 1s;
    transform: scale(1, 1);
}
```

这样的用户体验非常糟糕，但是却很好地体现了这一点。▶悬停时，打开导航耗时整整 2 秒钟。但是关闭时，短短 0.2 秒就结束了。鼠标悬停在列表元素上时，应用的是终态（悬停状态）定义的过渡属性，生效的是为悬停状态定义的 transition-duration: 2s。如

果鼠标没有悬停在菜单上，返回默认的缩小状态，使用的是始态（nav li ul 选择符）中定义的过渡属性，因此关闭菜单耗时 200ms。

请仔细观察这个示例，尤其是默认状态下的过渡样式。如果用户把鼠标从父级导航元素或作为子元素的下拉菜单上移开，下拉菜单延迟 50 毫秒才开始持续 200 毫秒的过渡。这其实是一种合宜的用户体验，在关闭菜单之前给用户留个机会（不过时间较短），把鼠标再移到菜单上。

定义过渡效果的四个属性虽然可以分别声明，但是更常使用简写属性。介绍简写属性之前，先讨论一下各个单独属性，掌握各属性的作用。

17.2.1 限制受过渡影响的属性

transition-property 属性指定想应用过渡效果的 CSS 属性名称。这样便可以限定只在特定的属性上应用过渡效果，而其他属性值的变化则瞬间完成。

transition-property

取值	none | [all | *<property-name>*]#
初始值	all
适用于	所有元素，以及 :before 和 :after 伪元素
计算值	指定的值
继承性	否
动画性	否

transition-property 属性的值是以逗号分隔的属性列表；或者是 none，表示不过渡任何属性；还可以是默认值 all，即“过渡所有支持动画的属性”。以逗号分隔的属性列表中也可以包含关键字 all。

如果只有关键字 all，或者默认为 all，那么所有支持动画的属性将一起过渡。假设我们想在悬停时改变一个框体的外观：

```
div {
```

```
    color: #ff0000;
    border: 1px solid #00ff00;
    border-radius: 0;
    transform: scale(1) rotate(0deg);
    opacity: 1;
    box-shadow: 3px 3px rgba(0, 0, 0, 0.1);
    width: 50px;
    padding: 100px;
}
div:hover {
    color: #000000;
    border: 5px dashed #000000;
    border-radius: 50%;
    transform: scale(2) rotate(-10deg);
    opacity: 0.5;
    box-shadow: -3px -3px rgba(255, 0, 0, 0.5);
    width: 100px;
    padding: 20px;
}
```

当鼠标指针移到 div 元素上时，始态与悬停状态（终态）值不同的每个属性都将变为悬停状态声明的值。transition-property 属性用于定义哪些属性的变化以动画形式持续一段时间（而不是瞬间变化）。所有属性都将从默认值变为悬停状态下的值，但是只有 transition-property 列出的支持动画的属性在指定的持续时间内逐渐过渡。不支持动画的属性，例如 border-style，立即从一个值变成下一个值。

如果 transition-property 属性的值只有 all，或者是逗号分隔的一组值中的最后一个，那么所有支持动画的属性将一起过渡。如果不想过渡全部属性，要逐一列出受过渡属性影响的各属性（以逗号分隔）。

因此，如果想过渡全部属性，下面两个样式基本上是等效的：

```
div {
    color: #ff0000;
    border: 1px solid #00ff00;
    border-radius: 0;
    transform: scale(1) rotate(0deg);
    opacity: 1;
    box-shadow: 3px 3px rgba(0, 0, 0, 0.1);
    width: 50px;
    padding: 100px;
    transition-property: color, border, border-radius, transform, opacity,
        box-shadow, width, padding;
    transition-duration: 1s;
}
```

```
div {
    color: #ff0000;
    border: 1px solid #00ff00;
    border-radius: 0;
    transform: scale(1) rotate(0deg);
    opacity: 1;
    box-shadow: 3px 3px rgba(0, 0, 0, 0.1);
    width: 50px;
    padding: 100px;
    transition-property: all;
    transition-duration: 1s;
}
```

两个样式中声明的 transition-property 属性都将过渡列出的所有属性，但是前者只过渡值可能有变化的八个属性，即在其他样式规则块中可能声明的属性。当前样式规则块中也声明了这八个属性，但这不是强制要求。

后一个样式规则中的 transition-property:all 指明，由样式变化事件导致值有变化的所有支持动画的属性，不管属性的值在哪个 CSS 规则块中发生变化，都在一秒钟的时段内过渡。能被该选择符匹配的所有元素的支持动画的全部属性都将逐渐过渡，而不只是当前样式块中声明的属性。

这里，前一个版本只把过渡效果应用到列出的八个属性上，不过这样能精确控制过渡哪个属性。单独声明各属性可以为每个属性指定不同的速度、延迟和（或）持续时间：

```
div {
    color: #ff0000;
    border: 1px solid #0f0;
    border-radius: 0;
    transform: scale(1) rotate(0deg);
    opacity: 1;
    box-shadow: 3px 3px rgba(0, 0, 0, 0.1);
    width: 50px;
    padding: 100px;
}

.foo {
    color: #00ff00;
    transition-property: color, border, border-radius, transform, opacity,
        box-shadow, width, padding;
    transition-duration: 1s;
}
<div class="foo">Hello</div>
```

如果想分开定义各属性的过渡效果，把属性一个个列出来，中间以逗号分隔。如果绝大部分属性持续的时间相同，而且延迟和步调也一致，只有少数属性例外，可以先使用 all，然后再为例外的属性单独定义时间、速度或步调。此时要把 all 放在首位。

```
div {
    color: #f00;
    border: 1px solid #00ff00;
    border-radius: 0;
    transform: scale(1) rotate(0deg);
    opacity: 1;
    box-shadow: 3px 3px rgba(0, 0, 0, 0.1);
    width: 50px;
    padding: 100px;
    transition-property: all, border-radius, opacity;
    transition-duration: 1s, 2s, 3s;
}
```

在这组逗号分隔的值中，all 表示样式规则中列出的所有属性，以及继承的所有 CSS 属性，还有其他匹配该元素或由该元素继承的其他 CSS 规则块中的全部属性。

在上述示例中，获得新值的所有属性将采用相同的持续时间、延迟和时序函数过渡，不过单独声明的 border-radius 和 opacity 例外。这两个属性在 all 后面单独列出，可以与其他属性采用相同的时间、延迟和时序函数，也可以指定不同的时间、延迟和时序函数。这里，除 border-radius 和 opacity 之外的所有属性，过渡的持续时间为 1 秒，而 border-radius 属性的持续时间为 2 秒，opacity 属性的持续时间为 3 秒（transition-duration 属性在下一节介绍）。



在逗号分隔的一组值中，all 必须放在首位。在 all 前面声明的属性涵盖在 all 之中，这样本想为前面的属性设定的其他过渡属性值将被覆盖。

禁用过渡效果

默认情况下没有过渡效果，然而如果设置了过渡，而后又想在特定的情况下撤销过渡效果，可以使用 transition-property:none 覆盖整个过渡声明，禁用所有属性的过渡效果。

none 关键字只能作为该属性的唯一一个值，不能放在以逗号分隔的一组值中。如果想撤销部分属性的过渡效果，只能列出仍想过渡的属性。transition-property 属性不能排除不想过渡的属性，只能涵盖想过渡的属性。



另一种方法是把属性的延迟和持续时间都设为 0s。这样，变化瞬间显现，就像没有应用 CSS 过渡效果一样。

过渡事件

在 DOM 中，不管是哪个方向的过渡，不管过渡持续多久、延迟多长，也不管过渡的属性是单独声明的还是涵盖在 all 中，过渡结束后都会触发 transitionend 事件。有时看似单个声明的属性，却会触发多个 transitionend 事件，因为简写属性中每个支持动画的属性有各自的 transitionend 事件。以下述样式为例：

```
div {
    color: #f00;
    border: 1px solid #00ff00;
    border-radius: 0;
    transform: scale(1) rotate(0deg);
    opacity: 1;
    box-shadow: 3px 3px rgba(0, 0, 0, 0.1);
    width: 50px;
    padding: 100px;
    transition-property: all, border-radius, opacity;
    transition-duration: 1s, 2s, 3s;
}
```

过渡结束时，将触发 8 次 transitionend 事件。例如，单单一个 border-radius 过渡就会触发 4 次 transitionend 事件，下面 4 个单独的属性各触发一次：

- border-bottom-left-radius
- border-bottom-right-radius
- border-top-right-radius
- border-top-left-radius

padding 属性也是 4 个独立属性的简写形式：

- padding-top
- padding-right
- padding-bottom
- padding-left

border 属性触发 8 次 transitionend 事件，border-width 简写属性对应 4 次，border-color 简写属性又对应 4 次：

- border-left-width
- border-right-width
- border-top-width
- border-bottom-width
- border-top-color
- border-left-color
- border-right-color
- border-bottom-color

然而，border-style 属性不触发 transitionend 事件，因为 border-style 属性不支持动画。

我们怎么知道 border-style 属性不支持动画呢？我们可以假设它不支持，因为 solid 和 dashed 两个值之间没有符合逻辑的中点。为了证明这一点，可以查阅附录 A 中列出的支持动画的属性列表，或者属性的规范。

这里，列出的 8 个属性将触发 21 次 transitionend 事件，因为其中包含前后两个状态下值有变化的简写属性。对 all 来说，至少触发 21 次 transitionend 事件：前后状态涵盖的 8 个属性中每个独立的属性触发一次，此外可能还有继承或在影响该元素的其他样式块中声明的属性。▶

transitionend 事件可以像这样监听：

```
document.querySelector('div').addEventListener('transitionend',
    function (e) {
      console.log(e.propertyName);
});
```

transitionend 事件有三个与该事件有关的属性：

1. propertyName：结束过渡的 CSS 属性的名称。
2. pseudoElement：应用过渡效果的伪元素，前面有两个冒号；如果过渡效果应用到常规的 DOM 节点上，返回空字符串。
3. elapsedTime：过渡持续的时间，单位为秒；返回的值通常是 transition-duration 属性声明的时间。

仅当属性成功过渡到新值后才会触发 transitionend 事件。倘若过渡被中断了，例如在其他地方对同一元素的同一属性做了修改，transitioned 事件不会被触发。

如果属性返回初始值，又触发一次 transitionend 事件。这次事件在过渡开始时就触发，即使原方向的过渡还未结束。

17.2.2 设置过渡持续时间

transition-duration 属性的值是以逗号分隔的时间长度列表，单位为秒（s）或毫秒（ms）。这些值指定从一个状态过渡到另一个状态历时多久。

transition-duration

取值	*<time>#*
初始值	0s
适用于	所有元素，以及 :before 和 :after 伪元素
计算值	指定的值
继承性	否
动画性	否

如果在两个状态之间来回过渡，而只有其中一个状态声明了持续时间，那么这个持续时间只在向那个状态过渡时起作用。请看下面两个样式规则：

```
input:invalid {
    transition-duration: 1s;
    background-color: red;
}

input:valid {
    transition-duration: 0.2s;
    background-color: green;
}
```

如果两个状态声明的 transition-duration 值不一样，过渡的持续时间为目标状态声明的 transition-duration 值。在上述示例中，如果输入框中的内容无效，变为红色背景的过程持续 1 秒，而内容有效时，变为绿色背景的过程只持续 200 毫秒。▶

transition-duration 属性的值是正数，单位为秒（s）或毫秒（ms）。规范要求必须带时间单位，就算设为零秒，也要写成 0s。默认情况下，属性值的变化是瞬间完成的，看不到动画效果，因此过渡持续时间的默认值为 0s。

如果 transition-delay 属性的值不是正值，而且没有声明 transition-duration，那么声明的 transition-property 不起作用，即不触发 transitionend 事件。只要设置的过渡总时间大于零秒，可以直接设为 0s，也可以不声明 transition-duration，默认为 0s，过渡效果就会应用，而且过渡结束时会触发 transitionend 事件。

transition-duration 的值不能为负，如果持续时间列表中有一个是负值，整个属性值都将失效。

以前面那个超长的 transition-property 声明为例，我们可以为所有属性声明统一的持续时间，也可以单独为各属性声明不同的持续时间，还可以让余下的属性使用相同的持续时间。如果想让全部属性使用相同的持续时间，设置过渡时只要声明一个 transition-duration 值：

```
div {
    color: #ff0000;
    ...
    transition-property: color, border, border-radius, transform, opacity,
        box-shadow, width, padding;
    transition-duration: 200ms;
}
```

此外，也可以在 transition-duration 属性中声明一组以逗号分隔的时间值，各值相同，而且与 transition-property 属性列出的值数量相等。如果想让各属性持续不同的时间，要在一组以逗号分隔的值中设置不同的时长：

```
div {
    color: #ff0000;
    ...
    transition-property: color, border, border-radius, transform, opacity,
        box-shadow, width, padding;
    transition-duration: 200ms, 180ms, 160ms, 140ms, 120ms, 100ms, 1s, 2s;
}
```

如果声明的属性数量与持续时间的数量不一致，根据各浏览器制定的规则处理。如果持续时间的数量比属性多，忽略多出的持续时间。如果属性的数量比持续时间多，重复使用前面的持续时间。在下面的示例中，color、border-radius、opacity 和 width 的持续时间为 100 毫秒，border、transform、box-shadow 和 padding 的持续时间为 200 毫秒。

```
div {
    ...
    transition-property: color, border, border-radius, transform, opacity,
        box-shadow, width, padding;
    transition-duration: 100ms, 200ms;
}
```

如果恰好声明两个持续时间，奇数位上的属性使用第一个持续时间，偶数位上的属性使用第二个持续时间。

我们要考虑用户体验。如果过渡太慢，网站会给人速度慢或无响应的感觉，一个微不足道的效果却吸引了过多的注意力。如果过渡太快，用户可能察觉不到。虽然过渡的持续时间可以设为任何正值，但是也不能过于离谱，应该记住过渡的目的是增强效果，不能牺牲用户体验。效果持续的时间以能让人看到为原则，但是不能太久，以免喧宾夺主。一般来说，持续时间在 100 到 200 毫秒之间的过渡效果最好，用户能看到动画过程，而且不至于分散注意力。

我们想让下拉菜单具有良好的用户体验，因此把两个属性的过渡持续时间都设为 200 毫秒：

```
nav li ul {
    transition-property: transform, opacity;
    transition-duration: 200ms;
    ...
}
```

17.2.3 调整过渡的内部时序

你是否希望过渡慢速开始，然后逐渐加快？或者快速开始，逐渐减速走向终点？又或者先平稳行进，然后步进甚至弹跳数次？transition-timing-function 属性用于控制过渡的步调。

transition-timing-function

取值	*<timing-function>#*
初始值	ease
适用于	所有元素，以及 :before 和 :after 伪元素
计算值	指定的值
继承性	否
动画性	否

transition-timing-function 可以取的值有 ease、linear、ease-in、ease-out、ease-in-out、step-start、step-end、steps(n, start)（n 是步进的次数）、steps(n, end) 和 cubic-bezier(x1, y1, x2, y2)（这些也是 animation-timing-function 属性的有效值，详情参见第 18 章）。

除步进关键字之外，其他关键字定义的是渐进时序函数，是描述平滑曲线的三次方贝塞尔函数的别名。规范预定义了五个渐进函数，见表 17-1。

表 17-1：支持的三次方贝塞尔函数关键字

时序函数	说明	三次方贝塞尔值
cubic-bezier()	指定一个三次方贝塞尔曲线	cubic-bezier(*x1*, *y1*, *x2*, *y2*)
ease	慢速开始，然后加速，再慢下来，结束时特别慢	cubic-bezier(0.25, 0.1, 0.25, 1)
linear	整个过渡过程保持相同的速度	cubic-bezier(0, 0, 1, 1)
ease-in	慢速开始，然后加速	cubic-bezier(0.42, 0, 1, 1)
ease-out	快速开始，然后减速	cubic-bezier(0, 0, 0.58, 1)
ease-in-out	与 ease 类似：中间较快，两端很慢，但不同速	cubic-bezier(0.42, 0, 0.58, 1)

三次方贝塞尔曲线函数接受四个数字参数。表 17-1 中列出的五个具名渐进函数对应的曲线如图 17-3 所示。例如，linear 等同于 cubic-bezier(0,0,1,1)。三次方贝塞尔函数的第一个和第三个参数必须在 0 和 +1 之间。

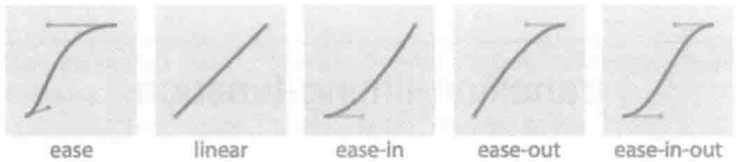


图 17-3：具名三次方贝塞尔函数对应的曲线

cubic-bezier() 函数的四个数字参数定义一个方框中两个手柄的 x 和 y 坐标。这两个手柄位于从方框左下角和右上角延伸出来的两条直线的末端。曲线使用这两个角和两个手柄的坐标经过贝塞尔函数计算得出。

下面通过几条曲线及对应的值来了解一下贝塞尔函数，如图 17-4 所示。

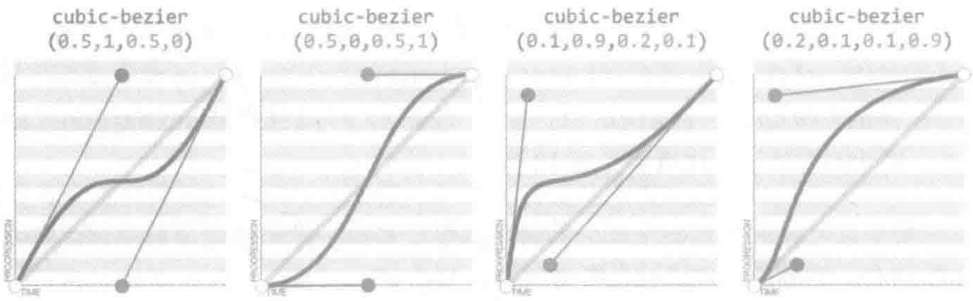


图 17-4：四条贝塞尔曲线及所用的 cubic-bezier() 函数值（出自 *http://cubic-bezier.com*）

先看第一个例子。前两个值对应于 *x1* 和 *y1*，分别为 0.5 和 1。这两个值确定第一个手柄的位置，即横向走一半（*x1*=0.5），纵向走到方框的顶边（*y1*=1）。*x2*,*y2* 坐标的值为 0.5,0，对应的点在方框底边的中点，第二个手柄就在这个位置。曲线就由这两个手柄的位置确定。

在第二个例子中，手柄的位置对调了，得到的曲线也变了。第三个和第四个例子中手柄的位置也是颠倒的。注意，手柄的位置对调后，得到的曲线是不一样的。

预定义的关键字毕竟有限，若想自由控制动画，要使用三次方贝塞尔函数，自己提供四个浮点数。如果你能熟练应用微积分，或者经常使用 Freehand 或 Illustrator 等程序，也许能在脑中构想出三次方贝塞尔函数。如若不然，可以使用一些在线工具，例如 *http://cubic-bezier.com/*，自己尝试不同的值。你还可以使用这些在线工具比较常用的关键字，或者与自己定义的三次方贝塞尔函数对比。

如图 17-5 所示，*http://easings.net* 网站额外提供了很多三次方贝塞尔函数值，你可以利用这些值实现更栩栩如生的动画。

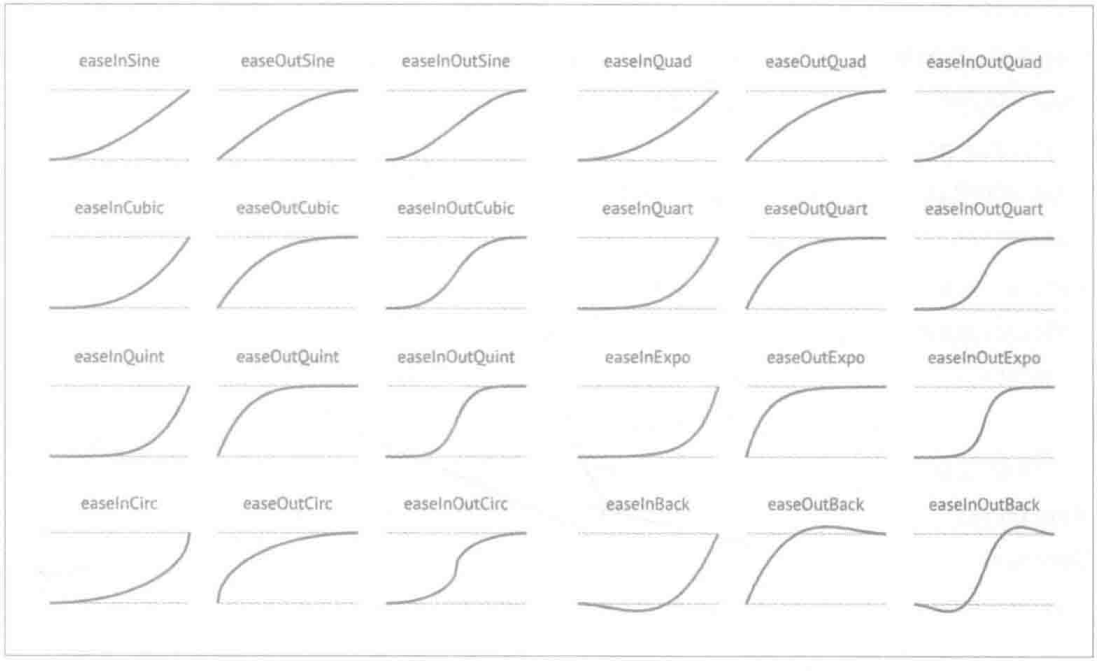


图 17-5：别人为我们定义的一些有用的三次方贝塞尔函数（出自 *http://easings.net*）

这些动画的名称是由网站的创作人员起的，不在 CSS 规范中，因此使用时要按照下表写：

非官方名称	三次方贝塞尔函数值
easeInSine	cubic-bezier(0.47, 0, 0.745, 0.715)
easeOutSine	cubic-bezier(0.39, 0.575, 0.565, 1)
easeInOutSine	cubic-bezier(0.445, 0.05, 0.55, 0.95)
easeInQuad	cubic-bezier(0.55, 0.085, 0.68, 0.53)

（续）

非官方名称	三次方贝塞尔函数值
easeOutQuad	cubic-bezier(0.25, 0.46, 0.45, 0.94)
easeInOutQuad	cubic-bezier(0.455, 0.03, 0.515, 0.955)
easeInCubic	cubic-bezier(0.55, 0.055, 0.675, 0.19)
easeOutCubic	cubic-bezier(0.215, 0.61, 0.355, 1)
easeInOutCubic	cubic-bezier(0.645, 0.045, 0.355, 1)
easeInQuart	cubic-bezier(0.895, 0.03, 0.685, 0.22)
easeOutQuart	cubic-bezier(0.165, 0.84, 0.44, 1)
easeInOutQuart	cubic-bezier(0.77, 0, 0.175, 1)
easeInQuint	cubic-bezier(0.755, 0.05, 0.855, 0.06)
easeOutQuint	cubic-bezier(0.23, 1, 0.32, 1)
easeInOutQuint	cubic-bezier(0.86, 0, 0.07, 1)
easeInExpo	cubic-bezier(0.95, 0.05, 0.795, 0.035)
easeOutExpo	cubic-bezier(0.19, 1, 0.22, 1)
easeInOutExpo	cubic-bezier(1, 0, 0, 1)
easeInCirc	cubic-bezier(0.6, 0.04, 0.98, 0.335)
easeOutCirc	cubic-bezier(0.075, 0.82, 0.165, 1)
easeInOutCirc	cubic-bezier(0.785, 0.135, 0.15, 0.86)
easeInBack	cubic-bezier(0.6, -0.28, 0.735, 0.045)
easeOutBack	cubic-bezier(0.175, 0.885, 0.32, 1.275)
easeInOutBack	cubic-bezier(0.68, -0.55, 0.265, 1.55)

步进时序

此外，还可以使用步进时序函数。规范预定义了两个步进值。

时序函数	定义
step-start	整个过渡都处在最终关键帧上。等同于 steps(1, start)
step-end	整个过渡都处在初始关键帧上。等同于 steps(1, end)
steps(n, start)	显示 *n* 个固定镜头，其中第一个固定镜头占整个过渡的百分之 *n*
steps(n, end)	显示 *n* 个固定镜头，前百分之 *n* 的时间处于初始值状态

如图 17-6 所示，使用步进函数时，从初始值到最终值的变化过程分为多步，而不是平滑过渡。

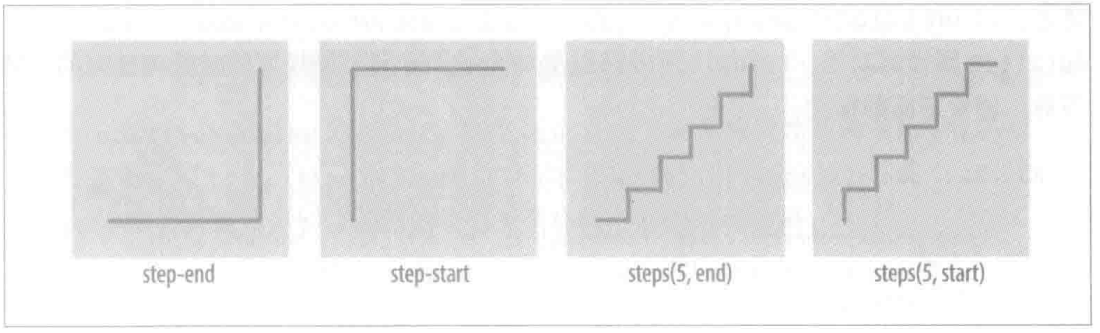


图 17-6：步进时序函数

步进函数把过渡分成等距的几个步骤。定义步进时，要为函数提供步进次数和方向。方向有两个：start 和 end。使用 start 时，第一步在动画的开头；使用 end 时，最后一步在动画的末尾。例如，steps(5,end) 等距跳跃五步，时间点分别为 0%、20%、40%、60% 和 80%；而 steps(5,start) 等距跳跃的时间点分别为 20%、40%、60%、80% 和 100%。

step-start 函数等同于 steps(1,start)。使用这个函数时，过渡的属性值从一开始便处于最终值状态，一直持续到过渡结束。step-end 函数等同于 steps(1,end)，它把过渡的属性设为初始值，在过渡持续的整个时间段内都保持这个值。



步进时序，尤其是 start 和 end 的确切含义，在第 18 章深入讨论。

仍以前面那个超长的 transition-property 声明为例，我们可以为所有属性声明统一的时序函数，也可以单独为各属性声明不同的时序函数等。下面的样式规则为所有过渡的属性设置统一的时序函数：

```
div {
    transition-property: color, border-width, border-color, border-radius,
        transform, opacity, box-shadow, width, padding;
    transition-duration: 200ms;
    transition-timing-function: ease-in;
}
```

此外，还可以为每个属性设置不同的节奏，不过这样的用户体验非常糟糕。

谨记，transition-timing-function 不改变过渡的持续时间，这是 transition-duration 属性的功用。transition-timing-function 只控制设定的时段内过渡的行进节奏。请看下面的样式：

```
div {
    ...
    transition-property: color, border-width, border-color, border-radius,
        transform, opacity, box-shadow, width, padding;
    transition-duration: 200ms;
    transition-timing-function: ease, ease-in, ease-out, ease-in-out, linear,
        step-end, step-start, steps(5, start), steps(3, end);
}
```

这里，我们分别为九个属性设置了九个不同的时序函数，只要过渡的持续时间和延迟相同，所有属性的过渡都在同一时间开始和结束。时序函数控制持续时间内过渡的行进节奏，但不改变耗时（顺便说一下，上述过渡的用户体验非常糟糕，请不要这么做）。

为了找到最合适的效果，最好的方法是逐个试验不同的时序函数。在测试的过程中，把 transition-duration 的值设大一些，这样能更好地看清不同函数之间的差别。▶如果速度太快，很难与渐进函数区分开。发布最终结果时，别忘了把速度调快。

17.2.4 延迟过渡

transition-delay 属性在元素上发生触发过渡的变化与开始过渡之间引入一定的延迟。

transition-delay

取值	*<time>#*
初始值	0s
适用于	所有元素，以及 :before 和 :after 伪元素
计算值	指定的值
继承性	否
动画性	否

把 transition-delay 设为 0s（默认值），过渡立即开始，即一旦元素的状态发生变化就开始过渡。例如，我们熟悉的 a:hover 就是这种效果。

如果 transition-delay 的 *<time>* 值不是 0s，定义的是从属性的值开始变化的那一刻（此时还未应用 transition 或 transition-property）到 transition 或 transition-property 声明的属性开始以动画形式变为最终值之间的时间间隔。

有趣的是，*<time>* 值可以为负。负值的效果参见本章后面“负的延迟值”一节。

还以前面声明了 8 个（或 21 个）属性的 transition-property 为例，如果想让所有属性都立即开始过渡，可以不声明 transition-delay 属性，也可以把值设为 0s。此外，还可以让一半的属性立即开始过渡，而余下的属性等待 200 毫秒再开始过渡，如下所示：

```
div {
    transition-property: color, border, border-radius, transform, opacity,
        box-shadow, width, padding;
    transition-duration: 200ms;
    transition-timing-function: linear;
    transition-delay: 0s, 200ms;
}
```

这里，我们把 transition-delay:0s, 200ms 应用到一系列属性上，而且每个属性的过渡效果持续 200 毫秒。在这组过渡的属性中，color、border-radius、opacity 和 width 立即开始过渡。余下的属性则等到奇数位的属性过渡结束后才开始过渡，因为我们为余下的属性设置的 transition-delay 等于应用于全部过渡属性的 transition-duration。

与 transition-duration 和 transition-timing-function 一样，如果 transition-delay 列出的值（以逗号分隔）比 transition-property 列出的值（以逗号分隔）多，多出的延迟值将被忽略。而如果 transition-property 列出的值（以逗号分隔）比 transition-delay 列出的值（以逗号分隔）多，将重复使用延迟值。

我们甚至可以声明九个不同的 transition-delay 值，让每个属性在前一个属性过渡结束后再开始过渡，如下所示：

```
div {
    ...
    transition-property: color, border-width, border-color, border-radius,
        transform, opacity, box-shadow, width, padding;
    transition-duration: 200ms;
```

```
    transition-timing-function: linear;
    transition-delay: 0s, 0.2s, 0.4s, 0.6s, 0.8s, 1s, 1.2s, 1.4s, 1.6s;
}
```

在这个示例中，我们使用 transition-duration 属性把每个过渡的持续时间设为 200 毫秒。然后，声明 transition-delay 属性，提供一组以逗号分隔的值，每个值相差 200 毫秒（0.2 秒），与各属性的过渡持续时间相同。这样，每个属性将在前一个属性过渡结束后才开始过渡。

我们可以做些数学计算，为每个过渡的属性设置不同的持续时间和延迟，但是确保所有过渡在同一时间结束：

```
div {
    ...
    transition-property: color, border-width, border-color, border-radius,
        transform, opacity, box-shadow, width, padding;
    transition-duration: 1.8s, 1.6s, 1.4s, 1.2s, 1s, 0.8s, 0.6s, 0.4s, 0.2s;
    transition-timing-function: linear;
    transition-delay: 0s, 0.2s, 0.4s, 0.6s, 0.8s, 1s, 1.2s, 1.4s, 1.6s;
}
```

在这个示例中，每个属性的过渡效果都在 1.8 秒后结束，但是各自的持续时间和延迟不尽相同。每个属性的 transition-duration 值加上 transition-delay 值必须等于 1.8 秒。

通常，我们希望所有过渡在同一时间开始。此时，只需为 transition-delay 属性提供一个值，这个值将应用到所有属性上。在图 17-1 所示的下拉菜单中，我们设置的延迟是 50 毫秒。这个延迟值不太长，不会引起用户的注意，也不让应用显得很慢。在某种程度上，50 毫秒的延迟能防止导航菜单意外展开，例如用户把鼠标从页面的一部分移到另一部分的过程中可能会不小心经过或悬停在菜单项上。

负的延迟值

如果 transition-delay 的值为负数，而且绝对值比 transition-duration 的值小，从中间某个位置立即开始过渡。例如：▶

```
div {
  transform: translateX(0);
  transition-property: transform;
  transition-duration: 200ms;
  transition-delay: -150ms;
  transition-timing-function: linear;
}
```

```
div:hover {
  transform: translateX(200px);
}
```

这里，我们把 transition-delay 的值设为 -150ms，而过渡持续 200ms，那么过渡将从 3/4 处开始，持续 50 毫秒。对本例中的线性时序函数而言，悬停时元素立即沿 x 轴平移 150px，然后在 50 毫秒的时间范围内慢慢从 150 像素处平移到 200 像素处。

如果 transition-delay 的值为负数，但是绝对值大于或等于 transition-duration 的值，属性的值瞬间变化，就像没有应用 transition 属性一样，因此也就不触发 transitionend 事件。

从悬停状态过渡回到原始状态时，默认情况下，还使用相同的 transition-delay 值。在上述示例中，由于悬停状态没有覆盖 transition-delay，因此元素将瞬间平移到整个过渡的 75% 处（即正向过渡的 25% 处），然后慢慢平移到原始状态。鼠标移开后，元素在 x 轴上跳到 50 像素处，然后用 50 毫秒的时间返回原位置，即沿 x 轴平移 0 像素。

17.2.5 transition 简写属性

transition 简写属性把目前介绍的四个属性合而为一：transition-property、transition-duration、transition-timing-function 和 transition-delay。

transition

取值	*<single-transition>#*
初始值	all 0s ease 0s
适用于	所有元素，以及 :before 和 :after 伪元素
计算值	指定的值
继承性	否
动画性	否

<single-transition> = [[none | *<transition-property>*] || *<time>* || *<transition-timing-function>* || *<time>*]#

transition 属性的值可以是 none，或者任意个以逗号分隔的单次过渡。单次过渡包含：应用过渡效果的一个属性，或者关键字 all，把过渡应用到全部属性上；过渡的持续时间；时序函数；以及延迟。

如果 transition 简写属性中的单次过渡没有声明要过渡的属性，那么此次过渡默认为 all。如果没有声明 transition-timing-function 值，默认为 ease。如果只有一个时间值，设定的是持续时间，而没有延迟，好似把 transition-delay 设为 0s 一样。

在单次过渡中，持续时间和延迟值的顺序很重要：解析为时间的第一个值设定的是持续时间。如果在逗号之前或语句末尾还有时间值，那就是延迟。

下面三个样式规则声明的过渡效果是等效的：

```
nav li ul {
    transition: transform 200ms ease-in 50ms,
                opacity 200ms ease-in 50ms;
}

nav li ul {
    transition: all 200ms ease-in 50ms;
}

nav li ul {
    transition: 200ms ease-in 50ms;
}
```

在第一个示例中，我们用简写形式为两个属性定义过渡效果。因为在悬停状态下我们想过渡全部属性，所以可以像第二个示例那样使用关键字 all。由于 all 是默认值，因此在简写形式中可以只提供持续时间、时序函数和延迟。如果时序函数不是 ease-in，而是 ease，还可以省略时序函数，因为 ease 是默认值。

持续时间必须设定，否则过渡效果不可见。换句话说，transition 属性中唯有 transition-duration 部分是真正必须的。

如果只想延迟菜单从闭合到展开的变化过程，仍要设置持续时间，只不过是设为 0s。记住，解析为时间的第一个值设定的是持续时间，第二个才是延迟：

```
nav li ul {
    transition: 0s 200ms; ...
```



这个过渡等待 200 毫秒才完全打开菜单，没有逐渐变化的过程。这样的用户体验非常糟糕。如果把选择符 nav li ul 改成 *，或许可以在愚人节捉弄一下别人。

如果有一组以逗号分隔的过渡（而不是只有一个过渡），而且值中有 none，那么整个过渡声明都将失效，从而被忽略。

```
div {
    transition-property: color, border-width, border-color, border-radius,
        transform, opacity, box-shadow, width, padding;
    transition-duration: 200ms, 180ms, 160ms, 140ms, 120ms, 100ms, 1s, 2s, 3s;
    transition-timing-function: ease, ease-in, ease-out, ease-in-out, linear,
        step-end, step-start, steps(5, start), steps(3, end);
    transition-delay: 0s, 0.2s, 0.4s, 0.6s, 0.8s, 1s, 1.2s, 1.4s, 1.6s;
}

div {
    transition:
        color 200ms,
        border-width 180ms ease-in 200ms,
        border-color 160ms ease-out 400ms,
        border-radius 140ms ease-in-out 600ms,
        transform 120ms linear 800ms,
        opacity 100ms step-end 1s,
        box-shadow 1s step-start 1.2s,
        width 2s steps(5, start) 1.4s,
        padding 3s steps(3, end) 1.6s;
}
```

上述两个 CSS 规则在功能上是等效的：可以使用四个单独的属性分别声明，也可以在简写属性中声明一组以逗号分隔的多个过渡。然而，两种写法不能混在一起：transition: transform, opacity 200ms ease-in 50ms 在延迟 50 毫秒后在 200 毫秒的时间范围内以渐进方式改变不透明度，但是 transform 将瞬间变化，不触发 transitionend 事件。

17.3 反向过渡：退回起点

在前面的示例中，我们都只声明了一个过渡。所有过渡都应用在默认状态上，由 hover 事件触发。这些情况下，当鼠标移开后，各属性通过相同的过渡回到默认状态，延迟相同，但时序函数是相反的。

如果只在全局状态中声明过渡，鼠标悬停和移开状态使用相同的 transition 声明，因为选择符能匹配两个状态。如果不想完全复用整个过渡，覆盖部分过渡的属性，可以在全局状态（而不只是悬停状态）中声明不同的过渡值。

在不同状态中声明的过渡只对所在的状态有效：

```
a {
    background: yellow;
    transition: 200ms background-color linear 0s;
  }
a:hover {
    background-color: orange;
    /* 转到 :hover 状态时的延迟 */
    transition-delay: 50ms;
  }
```

在这个示例中，当用户把鼠标悬停到链接上时，50 毫秒之后背景色才会慢慢变成橙色。而鼠标从链接上移开后，背景色立即开始过渡到黄色。在这两个方向中，过渡都耗时 200 毫秒，而且都采用线性变化方式。50 毫秒的延迟是在 :hover 状态（橙色）中声明的，因此在背景色变为橙色的过程中应用。

在前面的下拉菜单示例中，触发 :hover 事件后，菜单延迟 50 毫秒，在 200 毫秒的时间范围内采用渐入式时序展开。这个过渡使用 transition 属性在默认（非悬停）状态中设置。鼠标移开后，延迟 50 毫秒，各属性在 200 毫秒的时间范围内以渐出式时序返回原始状态。这种逆转效果是对非悬停状态中 transition 值的响应。这是默认的行为，是可控的。但是默认行为提供的用户体验最好，因此你可能并不想调整，然而你要知道确实可以调整。

折叠菜单时如果想快一会慢一会（现实中不会这么做，因为用户体验不好；这里故意这么做，只是为了演示），可以声明两个不同的过渡：

```
nav ul ul {
  transform: scale(1, 0);
  opacity: 0;
  ...
  transition: all 4s steps(8, start) 1s;
}
nav li:hover ul {
  transform: scale(1, 1);
  opacity: 1;
  transition: all 200ms linear 50ms;
}
```

过渡是向着目标状态行进的，样式发生变化时，使用的是过渡属性的新值，而不是旧值。我们把平顺的线性动画放在 :hover 状态中。所用的过渡属性是目标状态中声明的。在上述示例中，当用户把鼠标悬停在下拉菜单的父级 li 元素上时，菜单快速逐渐打开，这个动作延迟 50 毫秒，耗时 200 毫秒。当用户把鼠标从下拉菜单或父级 li 元素上移开后，过渡效果等待一秒后在四秒钟内完成，整个过程分为八步。

如果只声明一个 transition 属性，应该放在起始状态中，因为我们希望设置的过渡在变成任何状态时都生效，例如悬停或类有变化。我们希望任何变化都能触发过渡，因此一般把唯一的 transition 声明放在默认的（特指度最小）原始状态样式规则块中。如果想进一步控制，为不同的行进方向设置不同的过渡效果，要在涉及的所有类和 UI 状态中声明 transition 属性。



祖辈元素和后代元素上都有过渡效果时要小心。过渡的属性发生变化后立即过渡祖辈或后代节点可能导致意料之外的后果。如果后代元素的过渡效果在祖辈元素的过渡效果之前结束，后代元素将继续过渡，所用的值（仍在过渡中）继承自父元素。这可能不是你想要的效果。

逆转中断的过渡

如果过渡还未结束却被打断了（例如下拉菜单未完全打开之前鼠标移开了），属性的值将还原为过渡开始前的值，而且这个过程也有过渡效果。因为逆转部分过渡时将重复使用持续时间和时序函数，而这可能对用户体验带来不好甚至糟糕的影响，所以 CSS 过渡规范规定，逆转的过渡持续的时间较短。

在下拉菜单示例中，我们在默认状态中把 transition-delay 设为 50ms，而且没有在悬停状态中声明过渡相关的属性，因此浏览器在开始逆转或结束过渡之前将等待 50 毫秒。

如果正向动画结束了，已经过渡到最终的值，并且触发了 transitionend 事件，逆转时所有浏览器都将复用 transition-delay 值。

如表 17-2 所示，如果过渡没有结束，例如结束之前用户把鼠标从导航上移开了，除 Microsoft Edge 之外的所有浏览器在逆转时都将复用延迟值。有些浏览器也复用 transition-duration 值，但是 Edge 和 Firefox 实现了规范中所说的“逆转缩减因子”（reversing shortening factor）。

表 17-2：不同浏览器逆转未结束过渡的行为

浏览器	逆转延迟	过渡时间	实耗时间
Chrome	有	200 ms	0.200 s
Chrome	有	200 ms	0.250 s
Safari	有	200 ms	0.200 s
Firefox	有	38 ms	0.038 s
Opera	有	200 ms	0.250 s
Edge	无	38 ms	0.038 s

假设过渡开始 75 毫秒后用户把鼠标从菜单上移开了。这意味着，下拉菜单还未完全打开，也未完全变成不透明就要慢慢关闭。在关闭菜单之前，浏览器将等待 50 毫秒，就像展开菜单前等待 50 毫秒一样。

这其实是一个好的用户体验，在关闭之前等待几毫秒能防止用户意外移出菜单导致的急闪行为。如表 17-2 所示，除 Microsoft Edge 之外的所有浏览器都会这么做。

虽然我们只给浏览器留出 75 毫秒的时间部分展开下拉菜单，但是有些浏览器在逆转时仍将耗时 200 毫秒，即用完 `transition-duration` 属性声明的值。其他浏览器，例如 Firefox 和 Edge，则实现了 CSS 规范中所说的“逆转缩减因子”和“逆转调整开始值”（reversing-adjusted start value）。实现这两个值的浏览器逆转部分过渡所用的时间与原值相近，但不一定相等。

对步进时序函数来说，Firefox 和 Edge 会把时间向下取整为已完成的步数。比如有个耗时 10 秒、分为 10 步的过渡，3.25 秒之后开始逆转，那么正向过渡停止在第三步和第四步之间的 1/4 处（完成了 3 步，即整个过渡的 30%），因此逆转时将耗时 3 秒。在下述示例中，鼠标移开后，`div` 元素的宽度将增加到 130 像素，然后开始逐渐返回 100 像素：

```
div {
    width: 100px;
    transition: width 10s steps(10, start);
}
div:hover {
    width: 200px;
}
```

虽然逆转的持续时间将向下取整为已完成的最大步数，但是逆转的整个过程仍将分为声明的步数，而不是已完成的步数。上述示例在 3.25 秒处中断，因此逆转时将在 3 秒钟的时间内完成 10 步。逆转时每一步的持续时间将缩短，变成 300 毫秒，而且每一步宽度缩小 3 像素，而非 10 像素。

如果是通过过渡改变背景位置实现的小人动画▶，效果十分糟糕。规范和具体的实现可能会做出改变，让逆转的步骤数与执行到中途的过渡保持一致。目前，其他浏览器在逆转时将耗时 10 秒钟，在 10 秒内分 10 步逆转正向过渡已完成的 3 步，一秒一步，一步宽度减少 3 像素。

未实现反向缩减时序的浏览器将耗时整整 10 秒（而非 3 秒），把整个过渡分为 10 步，逆转正向完成的 30%。无论起初的过渡有没有完成，也不管采用何种时序函数，这些浏览器在逆转过渡时都将使用原始过渡的完整持续时间，如果 `transition-delay` 为负值，再从中减去延迟的绝对值。在上述步进示例中，反向过渡将耗时 10 秒钟。在导航菜单示例中，不管下拉菜单有没有完全展开，逆转都将用时 200 毫秒。

对实现了逆转时序调整机制的浏览器来说，如果是线性时序函数，两个方向的持续时间相同。如果是步进时序函数，逆转过渡的持续时间等于完成最后一步所用的时间。如果是其他 cubic-bezier 函数，持续时间与中断前已完成的进程成比例（*https://drafts.csswg.org/css-transitions/transition-reversing-demo*）。如果 `transition-delay` 为负值，延迟也按比例减少。延迟为正值的话，两个方向则保持不变。

任何浏览器都不会在悬停状态触发 `transitionend` 事件，因为过渡根本未结束。但是，当反向过渡把菜单完全折叠起来后，在所有浏览器中都会触发 `transitionend` 事件。反向过渡的 `elapsedTime` 值取决于浏览器是在 200 毫秒内关闭菜单，还是用部分打开菜单所用的时间关闭菜单。

若想覆盖这些值，在始态和终态（例如针对未悬停和悬停的样式）中都要声明过渡属性。这对逆向缩减机制虽然没有影响，但是却能进一步控制。

17.4 支持动画的属性和值

着手实现过渡和动画之前，要明确一点：不是所有属性都支持动画。支持动画的 CSS 属性均可应用过渡（或动画）效果。那么，哪些属性支持动画呢？



附录 A 列出了支持动画的属性，但是 CSS 在不断变化着，支持动画的属性列表（*https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_animated_properties*）很有可能会增加新成员。

判断属性是否支持动画的关键是确定其取值能否内插。插值指在两个数据点之间插入一个数据点。判断属性的值是否支持动画的关键准则是计算值能否内插。如果属性的计算值是关键字，不能内插；如果关键字能计算为某种数值，则能内插。简单而言，如果能找到属性的两个值的中间点，那么属性的值可能就支持动画。

例如，`display` 属性的 `block` 和 `inline-block` 值不是数值，因此没有中间点，从而不支持动画。`transform` 属性的 `rotate(10deg)` 和 `rotate(20deg)` 值有中间点，是 `rotate(15deg)`，因此支持动画。

`border` 属性是 `border-style`、`border-width` 和 `border-color` 的简写形式（各自又是各单边属性的简写）。虽然 `border-style` 属性的值没有中间点，但是 `border-width` 属性的长度单位值是数值，支持动画。`medium`、`thick` 和 `thin` 等关键字值有等效的数值（`border-width` 属性把关键字计算为长度），因此这些值是可内插的。

`border-color` 属性的颜色值是数值（具名颜色都表示十六进制颜色值），因此颜色也支持动画。如果从 `border: red solid 3px` 过渡到 `border: blue dashed 10px`，边框宽度和边框颜色将按照设定的速度过渡，而 `border-style` 将在过渡开始时（延迟过后）直接由 `solid` 变成 `dashed`。

如附录 A 所示，数字值基本上都支持动画。不能转换为数值的关键字值一般不支持动画。参数值为数值的 CSS 函数，一般支持动画。不过 `visibility` 例外：虽然 `visible` 和 `hidden` 两个值之间没有中间点，但在可见与不可见之间却可以内插值。如果想插值，`visibility` 属性的初始值或目标值必须有一个为 `visible`。在过渡结束时，值将由 `visible` 变成 `hidden`。如果是从 `hidden` 过渡到 `visible`，在过渡开始时变化。

`auto` 通常应该视作不支持动画的值，最好别在动画和过渡中使用。规范规定，`auto` 不支持动画，但是有些浏览器把 `auto`（例如 `height: auto`）对应的数字值解释为 `0px`。对 `height`、`width`、`top`、`bottom`、`left`、`right` 和 `margin` 等属性来说，`auto` 值不支持动画。

有时，换个属性或值可能就行了。例如，不要从 `height:0` 变到 `height:auto`，可以从 `max-height:0` 变到 `max-height:100vh`，这样一般就能得到所需的效果。在 `min-height` 和 `min-width` 属性中，`auto` 值支持动画，因为 `min-height:auto` 的计算结果为零。

17.4.1 属性值是如何内插的

如果值能落在两个或多个已知值之间就能内插。可内插的值可在过渡和动画中使用。

数字以浮点数内插。整数以自然数内插，以自然数的形式递增或递减。

在 CSS 中，长度值和百分数转换为实数。如果过渡或以动画形式呈现 `calc()`，或者在长度值和百分数之间变化，值将转换为 `calc()` 函数，以实数的形式内插。

过渡的颜色，不管是 HSLA、RGB，还是具名颜色（如 `aliceblue`），都转换为对应的 RGBA 值，在 RGBA 色彩空间中内插。

以动画的形式改变字重时，如果用的是关键字，例如 bold，将转换为数字值，然后以 100 的倍数步进播放动画。这个行为未来可能会变，允许字重使用任何整数值。届时，内插的字重为整数，而不是 100 的倍数。

如果支持动画的属性值有多个部分，各部分单独内插。例如，`text-shadow` 的值有四部分：颜色、x、y 和 `blur`。颜色按照 `color` 值内插，而 x、y 和 `blur` 部分按长度内插。盒子投影还有两个部分：`inset`（可选）和 `spread`。`spread` 部分的值是长度，因此就按照长度值内插。`inset` 关键字无法转换为数值，可以从一个内凹阴影过渡到另一个内凹阴影，或者从一个外凸阴影过渡到另一个外凸阴影，但是不能从一个内凹阴影逐渐过渡到一个外凸阴影。

类似地，渐变也可以过渡，但是仅当两端的过渡是具有相同数量色标的同类渐变（线性或径向）。各色标的颜色以颜色值内插，色标的位置以长度值和百分数内插。

内插重复的值

如果属性的值为一个列表，各个值将按照相应的类型内插，前提是前后状态中列表里的值或可重复的值数目相等，而且每对值都可以内插。

```
.img {
    background-image:
        url(1.gif), url(2.gif), url(3.gif), url(4.gif),
        url(5.gif), url(6.gif), url(7.gif), url(8.gif),
        url(9.gif), url(10.gif), url(11.gif), url(12.gif);
    background-size: 10px 10px, 20px 20px, 30px 30px, 40px 40px;
    transition: background-size 1s ease-in 0s;
}
```

```
.img:hover {
    background-size: 25px 25px, 50px 50px, 75px 75px, 100px 100px;
}
```

假如我们要过渡四个背景尺寸，而列出的值单位都为像素，那么从过渡前状态中的第三个 background-size 就可以逐渐过渡到目标状态中的第三个 background-size 值。在上述示例中，鼠标悬停在图像上时，背景图 1、6 和 10 的高度和宽度将从 10px 过渡到 25px。类似地，背景图 3、7 和 11 的高度和宽度将从 30px 过渡到 75px；其他背景图以此类推。

因此，background-size 的值将重复三次，就像把 CSS 写成下面这样：

```
.img {
    ...
    background-size: 10px 10px, 20px 20px, 30px 30px, 40px 40px,
                     10px 10px, 20px 20px, 30px 30px, 40px 40px,
                     10px 10px, 20px 20px, 30px 30px, 40px 40px;
    ...
}
.img:hover {
    background-size: 25px 25px, 50px 50px, 75px 75px, 100px 100px,
                     25px 25px, 50px 50px, 75px 75px, 100px 100px,
                     25px 25px, 50px 50px, 75px 75px, 100px 100px;
}
```

如果以逗号分隔的值数目与背景图的数量不一致，将复制，直到数目相等，即使 :hover 状态中的值与始态不一致。

```
.img:hover {
    background-size: 33px 33px, 66px 66px, 99px 99px;
}
```

如果始态中声明四个 background-size 值，:hover 状态声明三个 background-size 值，而且都是像素值，背景图仍为 12 个，那么在这两个状态之间过渡时，悬停状态和始态中的值都要复制（分别复制三次和四次），直到有 12 个值为止，好似下面这样声明一样：

```
.img {
    ...
    background-size: 10px 10px, 20px 20px, 30px 30px,
                     40px 40px, 10px 10px, 20px 20px,
                     30px 30px, 40px 40px, 10px 10px,
                     20px 20px, 30px 30px, 40px 40px;
    ...
}
```

```
.img:hover {
    background-size: 33px 33px, 66px 66px, 99px 99px,
                     33px 33px, 66px 66px, 99px 99px,
                     33px 33px, 66px 66px, 99px 99px,
                     33px 33px, 66px 66px, 99px 99px;
}
```

如果有一对值无法内插，例如 background-size 从默认状态的 contain 变成悬停状态的 cover，根据规范，整个列表都不可插值。然而，有些浏览器会忽略无法插值的值对，继续以动画形式改变其他可内插的值。

有些属性值可由浏览器推出隐含的值，这些属性能以动画的形式改变。以投影为例，浏览器能推出 box-shadow: transparent 0 0 0 或 box-shadow: inset transparent 0 0 0 隐含的投影，替换前后状态中没有显式声明的值。本章的在线演示中有这样的例子（*https://meyerweb.github.io/csstdg4figs/*）。

只有可内插的值才能触发 transitionend 事件。

前面说过，visibility 属性的动画形式与其他属性不同：如果以动画的形式变成或过渡到 visible，又或者以动画的形式从 visible 变成或过渡到其他值，以单独的一步内插。在过渡或播放动画的过程中，只要时序函数的输出在 0 和 1 之间，元素就始终可见。如果是从 hidden 过渡到 visible，在一开始切换可见状态。如果是从 visible 过渡到 hidden，在结束时切换可见状态。注意，这个行为可用步进时序函数控制。

如果不小心加入了一个无法过渡的属性，也不用担心。整个声明不会失效。浏览器将把不支持动画的属性剔除，过渡余下的属性。注意，不支持动画的属性或不存在的 CSS 属性并没有被忽略。浏览器只是跳过无法识别或不支持动画的属性，它们仍在原来的位置，以免逗号分隔的其他过渡属性应用到错误的属性上[注 2]。



只有当下不受 CSS 动画影响的属性才能过渡。如果元素有动画效果，只要属性当前不受动画控制，仍然可以过渡。CSS 动画在第 18 章介绍。

注 2：这个行为可能会变。CSS 工作组正考虑让所有属性值都支持动画，如果前后状态中的值没有中间点，在时序函数的中段从一个值变成另一个值。

17.5 过渡是效果增强

浏览器对过渡的支持极好。所有浏览器，包括 Safari、Chrome、Opera、Firefox、Edge 和 Internet Explorer（从 IE10 开始），都支持 CSS 过渡。

过渡是对用户界面（user-interface，UI）的效果增强。即使没有得到全面支持，也不妨碍你使用。如果某个浏览器不支持 CSS 过渡，本想以过渡效果呈现的变化仍会发生，只不过触发样式重新计算事件时，始态将瞬间“过渡”到终态。

用户可能会错过有趣的（也可能是恼人的）效果，但不会错过任何内容。

过渡算是一种渐进增强，因此无需为旧版 IE 浏览器打腻子脚本（polyfill）。为了支持 IE9 及之前的版本，是可以使用 JavaScript 腻子脚本，在 Android 4.3 及之前的版本中也可以使用带前缀的过渡属性，不过这么做没多大意义。

17.6 打印过渡

打印网页或 Web 应用时，使用的是针对印刷媒体的样式表。如果 style 元素的 media 属性只匹配 screen，那么 CSS 对打印出来的页面根本没有影响。

通常，我们不设定 media 属性；这跟设定 `media="all"` 一样，也就是默认值。打印有过渡效果的元素时，可能会忽略内插的值，也可能会打印当前状态下属性的值，具体取决于浏览器。

纸张上看不到元素的过渡效果，在某些浏览器中，例如 Chrome，纸上打印出来的是调用 `print` 函数时所处的状态（前提是过渡的属性能打印出来）。如果变化的是背景色，那么前后状态中的背景色都不会打印，因为背景色一般不打印。然而，如果是文本颜色从一个值变成另一个值，彩色打印机或 PDF 打印机将打印 `color` 属性的当前值。

在另一些浏览器中，例如 Firefox，打印过渡开始前还是开始后的值取决于过渡是如何开始的。如果是鼠标悬停触发的，打印的是非悬停状态下的值，因为在打印对话框中操作时鼠标不可能还悬停在那个元素上。如果过渡是通过增加类触发的，打印的是过渡开始后的值，即使过渡尚未结束。打印程序就像忽略过渡属性一样。

倘若有单独的印刷样式表或针对印刷品的 @media 规则，浏览器将单独计算样式。在印刷样式中，样式没有变化，因此也就没有过渡。在打印程序看来，属性值是瞬间变化的，没有耗费一定时间的过渡效果。

第 18 章

动画

前一章介绍的 CSS 过渡算是一种简单的动画。过渡在状态发生变化时把属性的值从一个规则中设定的值变成另一个规则中设定的值，这个变化是在一段时间内完成的，而不是瞬间变化的。在 CSS 过渡中，属性的始态和终态值由规则中设定的属性值确定，但是对时间段内值的变化方式没有多少控制权。

CSS 动画与过渡的相同点是 CSS 属性的值都在一段时间内发生变化，而不同点是前者对变化的方式有更大的控制权。尤其是，通过关键帧实现的 CSS 动画能设定是否以及如何重复动画，还能深度控制整个动画的过程等。过渡触发的是属性值的隐式变化，而动画变化过程中用到的属性值要在关键帧中显式声明。

CSS 动画改变的属性值可以不在元素的前后两个状态中。在有动画效果的元素上设置的属性值无需参与动画过程。例如，从黑色到白色的过渡只以动画形式呈现不同的灰度；而使用动画时，在播放动画的过程中，元素可以不是黑色或白色，甚至也无需是位于二者之间的灰度。

当然，可以让颜色在不同的灰度之间过渡，但是我们可以先把元素变成黄色，然后再从黄色变成橙色。此外，还可以在不同的颜色之间变化，从黑色开始，到白色结束，让整个动画过程呈现彩虹式的变化。本章将探讨如何实现关键帧动画。



“播放”图标▶表示有在线示例。本章的所有示例可在这个页面获取：*https://meyerweb.github.io/csstdg4figs/18-animations*。

18.1 定义关键帧

若想为元素添加动画效果，要有一个关键帧，而这又要求有一个具名关键帧动画。首先，要使用 `@keyframes` 规则定义可复用的 CSS 关键帧动画，并为动画起个名称。然后，通过这个名称把对应的动画效果应用到元素或伪元素上。

一个 `@keyframes` 规则有一个动画标识符（即动画的名称），以及一到多个关键帧块。每个关键帧块有一到多个关键帧选择符，声明属性及其值。整个 `@keyframes` 规则设置一个动画效果的完整迭代过程。动画可以迭代零次或多次，这主要取决于 `animation-iteration-count` 属性的值（参见 18.4.3 节）。

每个关键帧块中有一到多个关键帧选择符。关键帧选择符是动画持续时间内的时间点，可以是百分数，也可以是关键字 `from` 或 `to`。动画的一般结构如下：

```
@keyframes animation_identifier {
  keyframe_selector {
    property: value;
    property: value;
  }
  keyframe_selector {
    property: value;
    property: value;
  }
}
```

下面举几个例子：

```
@keyframes fadeout {
    from {
        opacity: 1;
    }
    to {
        opacity: 0;
    }
}
```

```
@keyframes color-pop {
    0% {
        color: black;
        background-color: white;
    }
    33% { /* 动画的三分之一处 */
        color: gray;
        background-color: yellow;
    }
    100% {
        color: white;
        background-color: orange;
    }
}
```

上述第一组关键帧把元素的 opacity 设为 1（完全不透明），然后以动画形式变成 0（完全透明）。第二组关键帧把元素的前景设为黑色、背景设为白色，然后以动画的形式把黑色前景变成灰色、再变成白色，把白色背景变成黄色、再变成橙色。

注意，关键帧没有指明动画的持续时间，这由一个专门的 CSS 属性设定。关键帧的作用是设置“由这个状态变成那个状态”，或者“在播放动画的某个时刻呈现某个状态”。鉴于此，关键帧的选择符才用百分数，或者 from 和 to。倘若在选择符中使用时间值（例如 1.5s），将导致关键帧失效。

18.2 设置关键帧动画

创建动画的第一步是使用 @keyframes 为动画起个名称，并在一对花括号中定义关键帧。截至目前，这跟媒体查询（参见第 20 章）很像。

在一对花括号中有一系列关键帧选择符，其后有一段 CSS，声明想以动画形式改变的属性。定义好关键帧之后，要使用 animation-name 属性把动画“附加”到元素上。这个属性在 18.4.1 节讨论。

先看 @ 规则声明，其后是动画的名称和一对花括号：

```
@keyframes nameOfAnimation {
...
}
```

为动画命名

为动画起的名称是一个标识符或字符串。起初，关键帧的名称必须为标识符，但是规范和浏览器也支持放在引号里的字符串。

标识符不放在引号中，而且有特定的规则。标识符可以使用的字符有 a-z、A-Z、0-9、连字符（-）、下划线（_），以及 ISO 10646 字符集中 U+00A0 及以上的字符。ISO 10646 是一个通用字符集，这意味着我们可以使用 Unicode 标准中匹配正则表达式 [-_a-zA-Z0-9\u00A0-\u10FFFF] 的任何字符。⏵但是，标识符不能以数字（0-9）或两个连字符开头。以一个连字符开头是可以的，但是后面跟着的不能是数字；如果非要这么做，可以使用反斜线转义数字或连字符。

动画名称中的转义字符必须使用反斜线（\）转义。例如，Q&A! 必须写成 Q\&A\!。âœŽ 可以直接写成 âœŽ（不，这不是笔误），而且✎是有效的名称。能通过键盘输入的字符，但不是字母或数字，例如 !、@、#、$，必须使用反斜线转义。

此外，别使用本章出现的关键字命名动画。比如说本章后文即将介绍的各动画属性的值，例如 none、paused、running、infinite、backwards 和 forwards。虽然规范没有明文禁止，但是在 animation 简写属性（见 18.5 节）中使用这样的名称可能导致动画效果失效。因此，尽管可以把动画命名为 paused（或其他关键字），但笔者强烈反对这么做。

18.3 关键帧选择符

关键帧选择符指明声明的属性值应用到动画的哪个时间点，即动画播放到某个时刻希望属性为什么值。如果想设定动画开头的值，在 0% 记号处声明。如果想让属性在动画结束时变成另一个值，在 100% 记号处声明属性的值。如果想让属性在动画的三分之一处变成某个值，在 33% 记号处声明。这些记号就是关键帧选择符。

关键帧选择符可以是以逗号分隔的一组百分数，也可以是关键字 from 或 to。关键字 from 等于 0%，关键字 end 等于 100%。关键帧选择符表示目标关键帧在动画持续时间内位于百分之几的位置。关键帧由选择符后的属性值块声明。百分数值必须带上 % 号；也就是说，0 不是有效的关键帧选择符。

```
@keyframes W {
    from {
      left: 0;
      top: 0;
    }
```

```
    25%, 75% {
      top: 100%;
    }
    50% {
      top: 50%;
    }
    to {
      left: 100%;
      top: 0;
    }
}
```

这个名为 W 的 @keyframes 动画，附加到非静态定位的元素上之后，将沿着 W 形路径移动元素。W 动画有五个关键帧，分别位于 0%、25%、50%、75% 和 100% 记号处。from 就是 0%，to 就是 100%。

由于为 25% 和 75% 两个记号设置的属性是相同的，因此可以把这两个关键帧选择符写在一起，以逗号分开。这与常规的选择符是一样的，一组选择符可以使用逗号放在一起。把多个选择符写在同一行（如上例所示），还是一行写一个选择符完全由你自己决定。下述写法与前面的代码作用一样：

```
    25%,
    75% {
      top: 100%;
    }
```

注意，选择符无需按升序排列。在前例中，我们把 25% 和 75% 写在同一行，而 50% 记号在它们后面。为了易于辨别，强烈建议从 0% 写到 100%。然而，如这个示例中的 75% 记号所示，这不是强制要求。你可以先定义最后一个记号，最后再定义第一个记号，可以随机排列，也可以按照自己的喜好排列。

18.3.1 省略 from 和 to 值

如未指定 0% 或 from 关键帧，用户代理（浏览器）将使用要应用动画效果的属性的原始值构建一个 0% 关键帧。这就跟使用没有应用动画效果时属性的值声明 0% 关键帧一样，只不过相应的属性不能受其他动画的影响（详情参见 18.4.1 节）。类似地，如果没有定义 100% 或 to 关键帧，而且没有应用其他动画，浏览器将使用没有动画效果时属性的值构建一个虚设的 100% 关键帧。

假设有个改变 background-color 的动画：

```
@keyframes change_bgcolor {
    45% { background-color: green; }
    55% { background-color: blue; }
}
```

而且一开始，元素上设置了 background-color: red，那么实际应用的动画是下面这样的：▶

```
@keyframes change_bgcolor {
    0%   { background-color: red; }
    45%  { background-color: green; }
    55%  { background-color: blue; }
    100% { background-color: red; }
}
```

此外，我们知道，相同的关键帧可以写在一起，选择符用逗号分隔。因此，这个虚构的动画还可以写成：

```
@keyframes change_bgcolor {
    0%,
    100% { background-color: red; }
    45%  { background-color: green; }
    55%  { background-color: blue; }
}
```

注意，background-color: red; 声明其实不是这个关键帧动画的一部分。如果我们把元素在默认状态下的背景色设为黄色，那么 0% 和 100% 记号处的背景将是黄色的，在播放动画的过程中，背景色先由黄色变成绿色，再变成蓝色，最后又变回黄色：

```
@keyframes change_bgcolor {
    0%, 100% { background-color: yellow; }
    45% { background-color: green; }
    55% { background-color: blue; }
}
```

change_bgcolor 动画可以附加到很多元素上，而具体的动画过程根据无动画状态下元素的 background-color 属性值而有所不同。

目前，我们一直使用整数百分数，其实非整数，例如 33.33% 也是完全有效的。负的百分数、超过 100% 的值、非百分数，以及 to 和 from 之外的关键字是无效的，将被忽略。

18.3.2 重复关键帧属性

在 Webkit 最初实现的实验性动画中（使用 -webkit- 前缀），每个关键帧只能声明一次，如果多次声明，只有最后一个声明起作用，之前的关键帧选择符块被忽略。现在则不然。如今，与 CSS 中的其他机制一样，具有相同值的关键帧将层叠。在标准的句法中（不用前缀），前述 W 动画可以声明两次 to 或 100%，left 属性的值将被覆盖：

```
@keyframes W {
  from, to {
    top: 0;
    left: 0;
  }
  25%, 75% {
    top: 100%;
  }
  50% {
    top: 50%;
  }
  to {
    left: 100%;
  }
}
```

注意，第一个关键帧选择符中既有 from，也有 to。因此，那个选择符块会为 to 关键帧设定 top 和 left。而后，to 关键帧的 left 值在最后一个关键帧块中被覆盖。

18.3.3 支持动画的属性

特别注意，不是所有属性都支持动画。在动画的关键帧中列出的不支持动画的属性，直接被忽略（同理，浏览器无法识别的属性和值也将被忽略）。

附录 A 完整列出了支持动画的属性。本书在介绍各属性时也标出了是否支持动画。



18.4.7 节讨论的 animation-timing-function 属性虽然不支持动画，但是不会被忽略。如果关键帧选择符块中有 animation-timing-function 声明，所在块中的属性在变成下一个关键帧的过程中将使用 animation-timing-function 设定的时序函数。

如果两个属性值没有中间点，得到的动画效果可能与预期不符，可能无法正确以动画形式呈现，或者根本没有动画效果。例如，不应该让元素的高度在 height:auto 和 height: 300px 之间以动画形式变化，因为 auto 和 300px 之间没有中间点。动画效果可能还有，但是不同浏览器的处理方式不同：Firefox 不渲染动画；如果 auto 等同于 0，Safari 会渲染动画；目前，Opera 和 Chrome 直接跳到动画前后状态之间一半的位置，这可能并不是 50% 关键帧选择符，具体情况视 animation-timing-function 的值而定。也就是说，在没有中间点这个问题上，不同的浏览器对不同的属性有不同的处理方式，无法保证一定能得到预期的结果。

为每个属性都声明位于 0% 和 100% 位置上的值基本上能得到符合预期的动画效果。▶

例如，如果声明了 border-radius:50%;，最好也声明 border-radius:0;，因为 border-radius 的默认值是 none，不是 0，而 none 和其他值之间没有中间点。请比较下面两个动画：

```
@keyframes round {
    100% {
        border-radius: 50%;
    }
}
@keyframes square_to_round {
    0% {
        border-radius: 0%;
    }
    100% {
        border-radius: 50%;
    }
}
```

round 动画在设定的持续时间内从元素原本的 border-radius 值变成 border-radius:50%，square_to_round 动画在设定的持续时间内从 border-radius: 0% 变成 border-radius: 50%。如果元素一开始是直角，那这两个动画的效果完全一样。但是，如果元素起初就有圆角，那么 square_to_round 动画在开始播放之前将瞬间把元素变成直角。这可能与你预期的不一样。有时，可以省略 from 或 to 关键帧，利用元素未应用动画效果之前的属性值。

解决这个问题最好的方法是不用 square_to_round 动画，而用 round 动画，并且保证元素自身显式设定了 border-radius。▶

只要至少有一个块中有一个支持动画的属性，而且其值与未应用动画效果之前的值不同，并且两个值之间有中间点，那个属性的变化就能以动画形式呈现。

18.3.4 不支持动画但不被忽略的属性

上述中间点规则有两个例外：visibility 和 animation-timing-function。

虽然 visibility: hidden 和 visibility: visible 之间没有中间点，但是 visibility 属性支持动画。从 hidden 变成 visible 时，可见性的值从一个值直接跳到发生变化的下一个关键帧。

尽管 animation-timing-function 属性不支持动画，但是如果在关键帧块中声明了，所在块中的属性值将使用它设定的时序函数。动画时序的变化不以动画形式呈现，而是直接变成新值，并且只在转到下一个关键帧时起作用（详情参见 18.4.7 节）。

18.3.5 通过脚本编辑 @keyframes 动画

关键帧规则在出现的那一刻还会应用可以通过 API 查找、追加和删除。@keyframes 动画中声明的关键帧块可以使用 appendRule(n) 或 deleteRule(n) 修改，其中 n 是关键帧的完整选择符。关键帧的内容可通过 findRule(n) 获取。

```
@keyframes W {
  from, to {
    top: 0;
    left: 0;
  }
  25%, 75% {
    top: 100%;
  }
  50% {
    top: 50%;
  }
  to {
    left: 100%;
  }
}
```

appendRule()、deleteRule() 和 findRule() 三个方法的参数都是完整的关键帧选择符。以 W 动画为例，如果想获取 25% / 75% 关键帧的内容，传入的参数为 25%, 75%：

```
// 获取指定关键帧的选择符和内容
var aRule = myAnimation.findRule('25%, 75%').cssText;

// 删除 50% 关键帧
myAnimation.deleteRule('50%');
```

```
// 在动画末尾添加 53% 关键帧
myAnimation.appendRule('53% {top: 50%;}');
```

在 myAnimation.findRule('25%, 75%').cssText; 语句中，myAnimation 是一个关键帧动画的名称，返回的结果是匹配 25%,75% 的关键帧。如果只有 25% 或 75% 关键帧，匹配不到任何关键帧。如果用的是 W 动画，这个语句返回 25%, 75% { top: 100%; }。

类似地，myAnimation.deleteRule('50%') 将删除最后一个 50% 关键帧。因此，如果有多个 50% 关键帧，排在最后的那个将被删除。反过来，myAnimation.appendRule('53% {top: 50%;}') 将在 @keyframes 块中最后一个关键帧的后面追加 53% 关键帧。

动画有三个事件，animationstart、animationend 和 animationiteration，分别在动画的开头和结尾，以及一次迭代结束与下一次迭代开始之间触发。只要动画中定义的关键帧规则有效，就会触发 animationstart 和 animationend 事件，即使关键帧规则为空也是如此。animationiteration 事件仅在动画有多次迭代时才会触发，因为 animationiteration 不能与 animationend 事件同时触发。

18.4 把动画应用到元素上

定义好关键帧动画便可以把动画应用到元素和（或）伪元素上。为了把动画附加到元素上，并控制动画的播放过程，CSS 提供了多个相关的属性。若想保证动画效果能显示出来，至少要指明动画的名称，以及持续时间（不然动画瞬间就结束了）。

在元素上声明动画属性的方式有两种：一种是单独声明各个属性，另一种是使用 animation 简写属性一次性声明全部属性（抑或二者结合）。我们先来学习各个单独的属性，然后再介绍如何把全部声明浓缩在 animation 简写属性中。

下面逐一介绍各个单独的属性。

18.4.1 指定动画的名称

animation-name 属性的值为一个逗号分隔的列表，指定想应用的关键帧动画的名称。这里所说的名称是指使用 @keyframes 规则定义动画时设定的无引号标识符或有引号的字符串（抑或二者混用）。

animation-name

取值	[<*single-animation-name*> | none]#
初始值	none
适用于	所有元素，以及 ::before 和 ::after 伪元素
计算值	指定的值
继承性	否
动画性	否

默认值为 none，表示没有动画效果。根据 CSS 层叠机制，可以使用 none 值覆盖其他地方应用的动画（这也是不建议使用 none 命名动画的原因，除非你疯了）。如果想应用动画，把值设为 @keyframes 的标识符，即动画的名称。

下述示例使用 18.3.1 节定义的 change_bgcolor 关键帧动画：

```
div {
    animation-name: change_bgcolor;
}
```

这个规则把 change_bgcolor 动画应用到所有 div 元素上。

如果想应用多个动画，以逗号分开多个 @keyframes 标识符：

```
div {
    animation-name: change_bgcolor, round, W;
}
```

如果指定的一系列关键帧标识符中有一个不存在，不会导致所有动画都无法应用，只是那个不存在的动画被忽略，其他有效的动画仍将应用。被忽略后，如果后来又通过某种方式变成有效的动画了，在变有效的那一刻又会开始应用。请看下面的代码：

```
div {
   animation-name: change_bgcolor, spin, round, W;
}
```

在这个示例中，假设没有定义 spin 关键帧动画。spin 动画不会应用，但是 change_bgcolor、round 和 W 将应用。如果通过脚本编程添加了名为 spin 的关键帧动画，在它出现的那一刻将应用到元素上。

为了应用多个动画，我们在 animation-name 属性的值中设定了多个以逗号分隔的关键帧动画标识符。如果这些动画中有重复的属性，后面的动画将覆盖前面动画中相同属性的值。例如，如果同时在两个关键帧动画中改变元素的背景色，后列出的动画将覆盖先列出的动画中设定的背景，不过前提是属性（这里指的是背景色）是同时以动画形式变化的。详细说明参见 18.6 节。

例如有如下代码，假设各动画持续 10 秒钟：

```
div {animation-name: change_bgcolor, bg-shift;}

@keyframes bg-shift {
    0%, 100% {background-color: blue;}
    35% {background-color: orange;}
    55% {background-color: red;}
    65% {background-color: purple;}
}
@keyframes change_bgcolor {
    0%, 100% {background-color: yellow;}
    45% {background-color: green;}
    55% {background-color: blue;}
}
```

这里，元素的背景将由蓝色变为橙色，再变成红色，然后变成紫色，最后又变回蓝色，这个变化过程由 bg-shift 动画设置。因为它在后面列出，所以它定义的关键帧优先级更高。为相同的属性设置的多个动画在同一时刻播放，起作用的是在 animation-name 属性的值中后列出的动画。

省略 from（0%）或 to（100%）关键帧的动画有些特殊。例如，我们把 bg-shift 中定义的第一个关键帧删掉：

```
div {animation-name: change_bgcolor, bg-shift;}

@keyframes bg-shift {
    35% {background-color: orange;}
    55% {background-color: red;}
    65% {background-color: purple;}
}
```

```
@keyframes change_bgcolor {
    0%, 100% {background-color: yellow;}
    45% {background-color: green;}
    55% {background-color: blue;}
}
```

现在，bg-shift 动画没有定义首尾两个关键帧的背景色。遇到没有定义 0% 或 100% 关键帧的情况，用户代理会使用相应属性的计算值构建 0%/100% 关键帧。使用的计算值可能是未应用动画效果时属性的值，也可能是 animation-name 列出的前一个动画中的属性值。

较旧的浏览器采用前一种方式，但是后来规范做了调整，选择了后一种方式。从 2017 年年末开始，较新的浏览器在动画的前 3.5 秒从黄色变成橙色，后 3.5 秒从紫色变成蓝色。较旧的浏览器在动画开始和结束时将显示透明背景。

仅当不同的关键帧块在同一时刻改变相同属性的值时才需要考虑这些。这里，同时改变的是 background-color。倘若一个关键帧块改变的是 background-color，而另一个改变的是 padding，那么这两个动画就没有冲突，背景色和内边距将同时改变。

只把动画应用到元素上还不足以让动画呈现出来，动画虽然能播放，但是瞬间就结束。此时，关键帧中的属性都会经历变化，而且 animationstart 和 animationend 事件也会触发。若想看到动画效果，至少要让动画持续一定的时间。而这由 animation-duration 属性设定。

18.4.2 定义动画的时长

animation-duration 属性定义动画迭代一次用时多久，单位为秒（s）或毫秒（ms）。

animation-duration

取值	*<time>#*
初始值	0s
适用于	所有元素，以及 ::before 和 ::after 伪元素
计算值	指定的值
继承性	否
动画性	否

animation-duration 属性定义动画中全部关键帧完成一次循环所用的时间长度，单位为秒（s）或毫秒（ms）。如果不声明 animation-duration，动画仍将播放，只不过持续时间为 0s，看不到实际效果，而且 animationstart 和 animationend 事件也会触发。这个属性的值不能为负数。

指定持续时间时，必须带上单位，即秒（s）或毫秒（ms）。如果有多个动画，可以分别为各个动画设定持续时间，值之间以逗号分隔：

```
div {
   animation-name: change_bgcolor, round, W;
   animation-duration: 200ms, 100ms, 0.5s;
}
```

如果一组以逗号分隔的持续时间中有无效的值，例如 animation-duration: 200ms, 0, 0.5s，整个声明都失效，相当于声明 animation-duration: 0s。0 不是有效的时间值。

一般来说，animation-name 有多少个值，animation-duration 就应该有多少个值。如果只提供一个持续时间，所有动画都播放那么长的时间。假如 animation-duration 的值比 animation-name 的值少，也不会导致什么问题，缺少的值将成组复制。因此，对下述规则来说：

```
div {
    animation-name: change_bgcolor, spin, round, W;
    animation-duration: 200ms, 5s;
       /* 等效于 '200ms, 5s, 200ms, 5s' */
}
```

round 动画将持续 200ms，W 动画将持续 5s。

如果 animation-duration 的值比 animation-name 的值多，多出的值将被忽略。如果列出的动画不存在，animation-name 和 animation-duration 声明不会失效，不存在的动画及其持续时间将被忽略：

```
div {
    animation-name: change_bgcolor, spinner, round, W;
    animation-duration: 200ms, 5s, 100ms, 0.5s;
}
```

在这个示例中，持续时间 5s 对应于 spinner。可是，spinner 动画不存在，因此 5s 和 spinner 都被忽略。如果后来 spinner 动画又出现了，它会应用到 div 元素上，而且持续时间为 5 秒。

18.4.3 声明动画的迭代次数

只声明必须的 animation-name 属性，动画将播放一次，而且只播放一次。如果希望迭代的次数不是默认的一次，使用 animation-iteration-count 属性设定。

animation-iteration-count

取值	[<*number*> | infinite]#
初始值	1
适用于	所有元素，以及 ::before 和 ::after 伪元素
计算值	指定的值
继承性	否
动画性	否

默认情况下，动画播放一次（因为默认值是 1）。如果给 animation-iteration-count 属性提供其他值，可以是任何数字或关键字 infinite，而且 animation-delay 属性的值中没有负值，动画将重复指定的次数。

下述声明分别让动画重复 2 次、5 次和 13 次：

```
animation-iteration-count: 2;
animation-iteration-count: 5;
animation-iteration-count: 13;
```

如果 animation-iteration-count 的值不是整数，动画将在最后一次循环的中途结束。动画仍将播放，只是在最后一次迭代时中途停止。例如，animation-iteration-count: 1.25 迭代 1.25 次，在第二次迭代的 25% 处中断。如果值为 0.25，动画持续 8 秒，动画只播放 25%，即 2 秒后停止。

不允许设为负值。如果提供的值无效，将重置为默认值 1，只迭代一次。▶

有趣的是，0 是 animation-iteration-count 属性的有效值。设为 0 时，动画依然播放，只不过迭代零次。与 animation-duration: 0s 类似，这也会触发 animationstart 和 animationend 事件。

如果把多个动画附加到一个元素或伪元素上，animation-name、animation-duration 和 animation-iteration-count 的值都是一个以逗号分隔的列表：

```
.flag {
    animation-name: red, white, blue;
    animation-duration: 2s, 4s, 6s;
    animation-iteration-count: 3, 5;
}
```

那么，animation-iteration-count（以及其他动画属性）的值将按照 animation-name 属性的值按顺序分配。多出的值将被忽略。缺少的值（例如上例中的 animation-iteration-count）根据现有的值重复。

在上述示例中，名称的数量比次数值多，因此次数值将重复：red 和 blue 动画迭代三次，white 动画迭代五次。持续时间的数量与名称的数量相等，因此持续时间值不重复。red 动画持续两秒，迭代三次，因此共计用时六秒。white 动画持续四秒，迭代五次，共计用时 20 秒。blue 动画每次迭代用时六秒，迭代三次，整个动画效果共计用时 18 秒。

无效的值将导致整个声明失效，受影响的动画只播放一次。

虽然这三个动画的持续时间不同，但是如果想让它们同时结束，可以通过 animation-iteration-count 属性控制：

```
.flag {
    animation-name: red, white, blue;
    animation-duration: 2s, 4s, 6s;
    animation-iteration-count: 6, 3, 2;
}
```

在这个示例中，red、white 和 blue 都共计用时 12 秒，因为持续时间与迭代次数的乘积都等于 12 秒。

18.4.4 设置动画的播放方向

使用 animation-direction 属性可以控制动画是从 0% 关键帧向 100% 关键帧播放，还是从 100% 关键帧向 0% 关键帧播放。可以让所有迭代都按照相同的方向播放，也可以隔一个循环变换一次方向。

animation-direction

取值	[normal | reverse | alternate | alternate-reverse]#
初始值	Normal
适用于	所有元素，以及 ::before 和 ::after 伪元素
计算值	指定的值
继承性	否
动画性	否

animation-direction 属性定义按什么方向播放动画的关键帧。可取的值有四个：

animation-direction: normal
: 设为 normal 时（或者省略，默认为 normal），动画的每次迭代都从 0% 关键帧向 100% 关键帧播放。

animation-direction: reverse
: reverse 值逆序播放各次迭代，即从 100% 关键帧向 0% 关键帧播放。逆转动画的方向也就逆转了 animation-timing-function（参见 18.4.7 节）。

animation-direction: alternate
: alternate 值的意思是第一次迭代（以及后续各奇数次迭代）从 0% 向 100% 播放，第二次迭代（以及后续各偶数次迭代）方向相反，从 100% 向 0% 播放。

animation-direction: alternate-reverse
: alternate-reverse 值与 alternate 值类似，只不过是反过来的。第一次迭代（以及后续各奇数次迭代）从 100% 向 0% 播放，第二次迭代（以及后续各偶数次迭代）方向相反，从 0% 向 100% 播放。

```
.ball {
    animation-name: bouncing;
    animation-duration: 400ms;
    animation-iteration-count: infinite;
```

```
        animation-direction: alternate-reverse;
    }
    @keyframes bouncing {
        from {
            transform: translateY(500px);
        }
        to {
            transform: translateY(0);
        }
    }
```

这是个弹跳球动画，但是我们想在动画开始时让球下落，而不是抛向空中：即先下后上，而不是先上后下。因此，最符合需求的值是 `animation-direction: alternate-reverse`。▶

这是让球跳动的一种基本方式。球在弹跳的过程中，到达顶点时速度最慢，到达最低点时速度最快。举这个例子是为了说明 `alternate-reverse` 动画的方向。后文还将改进这个动画，加上时序，让弹跳过程更贴近实际（参见 18.4.7 节）。届时还将讨论逆向迭代时 `animation-timing-function` 是如何逆转的。

18.4.5 延迟播放动画

`animation-delay` 属性定义浏览器把动画附加到元素上之后等待多久开始第一次迭代。

animation-delay	
取值	*<time>#*
初始值	0s
适用于	所有元素，以及 ::before 和 ::after 伪元素
计算值	指定的值
继承性	否
动画性	否

`animation-delay` 属性设定把动画附加到元素上之后隔多久开始播放动画，单位为秒（s）或毫秒（ms）。

默认情况下，动画附加到元素上之后立即开始迭代，没有延迟。正值延迟播放动画，当 animation-delay 属性的值中列出的时间过去之后才开始。如果设为负值，动画立即开始播放，不过是从动画的中途开始。

把 animation-delay 设为负值能实现一些有趣的效果。负的延迟致使动画立即开始播放，但是却从动画的中途开始。例如，在元素上声明了 animation-delay: -4s 和 animation-duration: 10s，元素大概将从整个动画的 40% 处立即开始播放，并在六秒后结束。

这里用了“大概”这个词，因为动画不一定正好从 40% 关键帧开始播放，动画的 40% 记号何时开始取决于 animation-timing-function 属性的值。如果声明的是 animation-timing-function: linear，那么就从动画的 40% 处开始播放。

```
div {
  animation-name: move;
  animation-duration: 10s;
  animation-delay: -4s;
  animation-timing-function: linear;
}

@keyframes move {
  from {
    transform: translateX(0);
  }
  to {
    transform: translateX(1000px);
  }
}
```

这个线性动画示例持续 10 秒，延迟 -4 秒。因此，动画将从动画的 40% 处立即开始播放，在 6 秒的时间内把 div 元素向原位置的右侧平移 400 像素。

如果设置动画循环 10 次，延迟为 -600 毫秒，而持续时间为 200 毫秒，那么元素将从第四次迭代的开头立即开始播放动画：

```
.ball {
  animation-name: bounce;
  animation-duration: 200ms;
  animation-delay: -600ms;
  animation-iteration-count: 10;
  animation-timing-function: ease-in;
  animation-direction: alternate;
}
```

```
@keyframes bounce {
  from {
    transform: translateY(0);
  }
  to {
    transform: translateY(500px);
  }
}
```

这个动画共计用时不是 2000 毫秒（200ms × 10=2000ms，即 2 秒），也不是以常规方向开始的，而是从第四次迭代的开头以反方向立即开始，共计用时 1400 毫秒（1.4 秒）。

这个动画以反方向开始的原因是 animation-direction 的值为 alternate，即偶数次迭代的方向是反的，从 100% 关键字向 0% 关键帧播放。第四次迭代（偶数次迭代）是第一个可见的迭代。

这个动画将立即触发 animationstart 事件。animationend 事件将在 1400 毫秒之后触发。小球是被抛起的，而不是弹回的，在 200 毫秒、400 毫秒、600 毫秒、800 毫秒、1000 毫秒和 1200 毫秒之后共 6 次触发 animationiteration 事件。虽然我们把迭代次数设为 10，但是只触发了 6 次 animationiteration 事件，这是因为只迭代了 7 次；缺少 3 次迭代的原因是 animation-delay 为负值，而且最后一次迭代与 animationend 事件同时结束。注意，如果 animationiteration 事件与 animationend 事件同时触发，animationiteration 事件不会发生。

继续讨论之前，先深入了解一下动画事件。

18.4.6 动画事件

与动画有关的事件有三个：animationstart、animationiteration 和 animationend。每个事件都有三个只读属性：animationName、elapsedTime 和 pseudoElement，在所有浏览器中都无需使用前缀。

animationstart 事件在动画开始播放时触发；如有延迟，等 animation-delay 设定的时间过后触发；否则，立即触发。如果 animation-delay 为负值，animationstart 事件立即触发，在支持 elapsedTime 的浏览器中，它的值等于延迟的绝对值。而在仍需使用前缀的浏览器中，elapsedTime 等于 0。

```
.noAnimationEnd {
    animation-name: myAnimation;
    animation-duration: 1s;
    animation-iteration-count: infinite;
}
```

```
.startAndEndSimultaneously {
    animation-name: myAnimation;
    animation-duration: 0s;
    animation-iteration-count: infinite;
}
```

animationend 事件在动画播放结束时触发。如果把 animation-iteration-count 的值设为 infinite，而且 animation-duration 设定的时间大于 0，animationend 事件永不触发。如果把 animation-duration 设为零秒，或者缺省为零秒，就算迭代无数次，animationstart 和 animationend 事件基本上同时触发，而且按此顺序。

animationiteration 事件在两次迭代之间触发。迭代结束后动画也结束的话，触发 animationend 事件。▶ animationiteration 和 animationend 事件不会同时触发。

```
.noAnimationIteration {
    animation-name: myAnimation;
    animation-duration: 1s;
    animation-iteration-count: 1;
}
```

在 .noAnimationIteration 示例中，animation-iteration-count 的值为 1，因此迭代第一次也是唯一一次之后动画就结束。不论何时，只要 animationiteration 和 animationend 事件可能同时触发时，只有 animationend 事件会真的触发，而 animationiteration 事件不触发。迭代结束后，如果没有新的迭代开始，就不触发 animationiteration 事件。

如果没有声明 animation-iteration-count 属性，或者把值设为 1 或更小的值，不触发 animationiteration 事件。只要一次迭代(即便不是完整的迭代)结束后又开始一次迭代，而且新迭代的持续时间大于 0s，animationiteration 事件就会触发。

```
.noAnimationIteration {
    animation-name: myAnimation;
    animation-duration: 1s;
    animation-iteration-count: 4;
    animation-delay: -3s;
}
```

如果由于 animation-delay 为负值而导致动画的迭代次数比 animation-iteration-count 设定的次数少，没有迭代的循环不触发 animationiteration 事件。在上述示例代码中，一次 animationiteration 事件也不触发，因为前三次循环没有迭代（由于 animation-delay 的值为 -3s），而最后一次循环结束后整个动画也就结束了。▶

在这个示例中，animationstart 事件的 elapsedTime 属性值为 3，即延迟的绝对值。

动画链

我们可以利用 animation-delay 属性把多个动画串在一次，让下一个动画在前一个动画结束后立即开始：

```
.rainbow {
    animation-name: red, orange, yellow, blue, green;
    animation-duration: 1s, 3s, 5s, 7s, 11s;
    animation-delay: 3s, 4s, 7s, 12s, 19s;
}
```

在这个示例中，red 动画延迟三秒、持续一秒，因此 animationend 事件在四秒后触发。后续各动画都在前一个动画结束后开始。我们称这样的效果为 CSS 动画链。

因为第二个动画的延迟为四秒，所以 orange 动画将在第四秒开始内插 @keyframes 属性值，即 red 动画结束后立即开始。orange 动画持续三秒、延迟四秒，在第七秒结束。第三个动画（yellow）的延迟就是七秒，因而它将在 yellow 动画结束后立即开始。

这个示例在同一个元素上应用动画链。除此之外，还可以使用 animation-delay 属性在不同的元素上应用动画链：

```
li:first-of-type {
    animation-name: red;
    animation-duration: 1s;
    animation-delay: 3s;
}
li:nth-of-type(2) {
    animation-name: orange;
    animation-duration: 3s;
    animation-delay: 4s;
}
li:nth-of-type(3)  {
    animation-name: yellow;
    animation-duration: 5s;
    animation-delay: 7s;
}
li:nth-of-type(4) {
    animation-name: green;
    animation-duration: 7s;
    animation-delay: 12s;
}
li:nth-of-type(5) {
    animation-name: blue;
    animation-duration: 11s;
```

```
    animation-delay: 19s;
}
```

如果想让一组列表元素按顺序播放动画，▶把多个动画衔接在一起，各列表元素的 animation-delay 值要等于前一个动画的 animation-duration 和 animation-delay 值之和。

虽然可以使用 JavaScript 通过 animationend 事件判断何时附加后续动画（参见下文），不过使用 CSS 动画属性（animation-delay）把动画衔接在一起更合适。请注意一点，动画在 UI 线程中的优先级最低。因此，如果运行的脚本占用了用户界面（UI）线程，特定的浏览器在处理有动画效果的特定属性和值时将忽略延迟，等到 UI 线程空闲时才开始播放更多的动画。

动画的性能

有些（不是全部）动画在所有浏览器中都在 UI 线程中播放。在多数浏览器中，不透明度或变形动画由 GPU（Graphics Processing Unit，图形处理器）执行，而不是 CPU（Central Processing Unit，中央处理器），因此无需等到 UI 线程空闲下来。如果这些属性不是动画的一部分，UI 线程不可用会导致视觉效果卡顿：

```
/* 不要这样做 */
* {
    transform: translateZ(0);
}
```

在支持 3D 变形（参见第 16 章）的设备和浏览器中，把元素放入 3D 空间后，元素位于单独的层中，动画效果没有卡顿。鉴于此，人们经常使用前面建议不要使用的 translateZ 技巧。使用这个技巧把少数几个元素放在单独的层中是没问题的，不过有些设备的视频内存是有限的。每个单独的层都要消耗视频内存，而且要花时间从 UI 线程中移到 GPU 中合成的层里。因此创建的层越多，性能损耗越大。

为了提升性能，只要可能，就应该把 transform 和 opacity 放到动画中，而 top、left、bottom、right 和 visibility 则不要这么做。把 CPU 的消耗交给 GPU 不仅能提升性能，如果修改盒模型属性的话，浏览器要重排并重绘，这对性能也不利。不要一股脑把一切都交给 GPU，否则你会遇到新的性能问题。

如果想使用 JavaScript，还可以通过监听 animationend 事件启动后续动画，从而实现动画链：

```
<script>
  document.querySelectorAll('li')[0].addEventListener( 'animationend',
    function(e) {
        document.querySelectorAll('li')[1].style.animationName = 'orange';
    },
    false );

  document.querySelectorAll('li')[1].addEventListener( 'animationend',
    function(e) {
        document.querySelectorAll('li')[2].style.animationName = 'yellow';
    },
    false );

  document.querySelectorAll('li')[2].addEventListener( 'animationend',
    function(e) {
        document.querySelectorAll('li')[3].style.animationName = 'green';
    },
    false );

  document.querySelectorAll('li')[3].addEventListener( 'animationend',
    function(e) {
        document.querySelectorAll('li')[4].style.animationName = 'blue';
    },
    false );
</script>

<style>
  li:first-of-type {
    animation-name: red;
    animation-duration: 1s;
  }
  li:nth-of-type(2) {
    animation-duration: 3s;
  }
  li:nth-of-type(3)  {
    animation-duration: 5s;
  }
  li:nth-of-type(4) {
    animation-duration: 7s;
  }
  li:nth-of-type(5)  {
    animation-duration: 11s;
  }
</style>
```

在这个示例中，我们为前四个列表项目分别定义了一个事件处理程序，监听相应列表项目的 animationend 事件。触发 animationend 事件时，事件监听程序会为后一个列表项目添加一个 animation-name 声明。

从上述样式可以看到，这个动画链根本没使用 animation-delay。这里采用的做法是，通过 JavaScript 事件监听器在触发 animationend 事件时设置 animation-name 属性，从而把动画附加到相应的元素上。

注意，只有第一个列表项目的样式中有 animation-name 声明。其他列表项目的样式中只有 animation-duration 声明，而没有 animation-name 声明，因此也就没有附加动画。添加 animation-name 声明的目的就是附加并启动动画。如果想启动或重启动画，必须先把动画名称删除，然后再加回来，就在这一刻，所有动画属性，包括 animation-delay，都将生效。

其实，下述代码：

```
<script>
  document.querySelectorAll('li')[2].addEventListener( 'animationend',
    function(e) {
        document.querySelectorAll('li')[3].style.animationName = 'green';
    },
    false );

  document.querySelectorAll('li')[3].addEventListener( 'animationend',
    function(e) {
        document.querySelectorAll('li')[4].style.animationName = 'blue';
    },
    false );
</script

<style>
  li:nth-of-type(4) {
    animation-duration: 7s;
  }
  li:nth-of-type(5)  {
    animation-duration: 11s;
  }
</style>
```

还可以写成：

```
<script>
  document.querySelectorAll('li')[2].addEventListener( 'animationend',
```

```
    function(e) {
        document.querySelectorAll('li')[3].style.animationName = 'green';
        document.querySelectorAll('li')[4].style.animationName = 'blue';
    },
  false );
</script>

<style>
  li:nth-of-type(4) {
    animation-duration: 7s;
  }
  li:nth-of-type(5)  {
    animation-delay: 7s;
    animation-duration: 11s;
  }
</style>
```

我们在同一时间添加 green 和 blue 两个动画名称，此时第五个元素的延迟将起作用，开始计时。



在播放动画的过程中改变动画属性的值（而不是名称）对动画没有影响，但是删除或增加 animation-name 声明却有影响。例如，在动画的中途不能把持续时间由 100ms 改成 400ms，延迟一旦开始计时也不能把 -200ms 改成 5s。然而，把动画删除然后再附加到元素上，可以停止再重新开始播放动画。在上述 JavaScript 示例中，为了开始播放动画，我们把动画应用到元素上。

此外，在元素上设置 display: none 将终止动画。把 display 改成导致元素可见的值，动画将从头开始播放。如果 animation-delay 是正值，延迟必须在 animationstart 事件触发之前结束，而且要在其他动画开始播放之前。如果延迟是负值，动画将从迭代的中途开始，这与通过其他方式应用动画是完全一样的。

延迟动画中的迭代

虽然没有 animation-iteration-delay 这样的属性，但是我们可以利用 animation-delay 属性，以及关键帧声明造成的延迟，或者使用 JavaScript 伪造。使用哪种方法最好，取决于迭代的次数、性能，以及延迟的时长是否都相等。

延迟动画中的迭代是什么意思呢？有时，我们想多次播放动画，并且想在两次迭代之间加上一定的等待时间。

假设我们想增大元素三次，但是在每次用时一秒的迭代之间等待四秒钟。为此，可以在关键帧定义中声明延迟，然后迭代三次：

```
.animate3times {
    background-color: red;
    animation: color_and_scale_after_delay;
    animation-iteration-count: 3;
    animation-duration: 5s;
}

@keyframes color_and_scale_after_delay {
    80% {
        transform: scale(1);
        background-color: red;
    }
    80.1% {
        background-color: green;
        transform: scale(0.5);
    }
    100% {
        background-color: yellow;
        transform: scale(1.5);
    }
}
```

注意，第一个关键帧的选择符是 80%，而且块中声明的属性与默认状态是一致的。▶这个动画将播放三次，前 80% 的时间（持续 5 秒，因此是 4 秒）保持默认状态，在最后一秒由绿色变成黄色，尺寸由小变大，然后开始下一次迭代，三次迭代后停止。

这个方法适用于迭代任何次数的动画。然而，仅当每两次迭代之间的延迟都相等，而且不想以其他时序（例如延迟六秒）播放动画时，这才是一种好的方法。▶如果想让每两次迭代之间的延迟有变化，但不想改变尺寸和颜色变化持续的时间，要新定义一个 `@keyframes` 规则。

若想在迭代之间添加不同的延迟，可以创建一个动画，然后精心设置，实现三种不同的延迟：

```
.animate3times {
    background-color: red;
    animation: color_and_scale_3_times;
    animation-iteration-count: 1;
    animation-duration: 15s;
}
```

```
@keyframes color_and_scale_3_times {
  0%, 13.32%, 20.01%, 40%, 46.67%, 93.32% {
        transform: scale(1);
        background-color: red;
  }
    13.33%, 40.01%, 93.33% {
        background-color: green;
        transform: scale(0.5);
  }
    20%, 46.66%, 100% {
        background-color: yellow;
        transform: scale(1.5);
  }
}
```

这个方法所用的代码不易写出，难以维护。▶而且只适用于单次循环的动画。如果想改变动画的迭代次数和迭代之间的延迟，要重新定义一个 @keyframes 规则。这个示例还没有前一个稳固呢，不过却在迭代之间添加了不同的延迟。

有一种方法适用于多数浏览器，而且动画规范也提到了：多次应用同一个动画，每次设置不同的 animation-delay 值。▶

```
.animate3times {
  animation: color_and_scale, color_and_scale, color_and_scale;
  animation-delay: 0, 4s, 10s;
  animation-duration: 1s;
}

@keyframes color_and_scale {
    0% {
        background-color: green;
        transform: scale(0.5);
    }
    100% {
        background-color: yellow;
        transform: scale(1.5);
    }
}
```

这里，同一个动画附加了三次，每一次的延迟都不同。这样每次动画迭代结束后才开始播放下一个。

如果同时播放的动画有重叠，使用的属性值来自最后一个声明的动画。有多个动画在同一时间改变相同的属性时也是如此，在动画名称列表中后列出的动画将覆盖之前动画中的相同属性。这里我们附加三次 color_and_scale 动画，但是延迟不同，color_and_scale 动画最后一次迭代将覆盖之前尚未结束的迭代所用的属性值。▶

模拟 animation-iteration-delay 属性最安全、最可靠，也是跨浏览器支持情况最好的方法是使用动画事件。触发 animationend 事件时，从元素上拆离动画，等到迭代延迟结束后再重新附加。如果迭代之间的延迟都相等，可以使用 setInterval；如果不同，使用 setTimeout。

```
var iteration = 0;
var el = document.getElementById('myElement');

el.addEventListener('animationend', function(e) {
  var time = ++iteration * 1000;

  el.classList.remove('animationClass');

  setTimeout(function() {
    el.classList.add('animationClass');
  }, time);

});
```

这个示例无限次播放 myElement 元素上的动画，每迭代一次增加一秒延迟。

18.4.7 改变动画的内部时序

好了，编写脚本是很好玩，不过我们还是回到纯 CSS 上吧。下面讨论时序函数。与 transition-timing-function 属性类似，animation-timing-function 属性指明动画在一次循环（或迭代）中如何演进。

animation-timing-function

取值	[ease	linear	ease-in	ease-out	ease-in-out	step-start	step-end	steps(<integer>, start)	steps(<integer>, end)	cubic-bezier(<number>, <number>, <number>, <number>)]#
初始值	ease
适用于	所有元素，以及 ::before 和 ::after 伪元素
计算值	指定的值
继承性	否
动画性	否

除步进时序函数之外（参见“步进时序函数”一节）的其他时序函数都是贝塞尔曲线。与transition-timing-function属性一样，CSS规范预定义了五个贝塞尔曲线关键字，如图18-1所示，简介见表18-1。

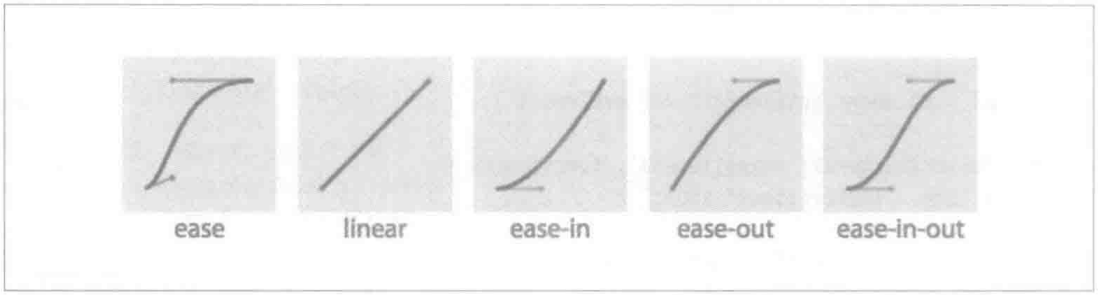


图18-1：几个三次方贝塞尔函数的名称

表18-1：贝塞尔曲线关键字

时序函数	三次方贝塞尔值
ease	cubic-bezier(0.25, 0.1, 0.25, 1)
linear	cubic-bezier(0, 0, 1, 1)
ease-in	cubic-bezier(0.42, 0, 1, 1)
ease-out	cubic-bezier(0, 0, 0.58, 1)
ease-in-out	cubic-bezier(0.42, 0, 0.58, 1)

使用Lea Verou的三次方贝塞尔可视化工具（cubic-bezier.com）可以实时预览自己创建的贝塞尔曲线。

默认的时序函数ease，开头较慢，然后逐渐加速，最后又慢下来。ease-in-out函数与之很像，不过开头阶段的加速度更快。linear函数从名称上能看出，播放的速度保持不变。

使用ease-in函数的动画开始时较慢，然后加速，最后突然停止。与之相反，ease-out时序函数全速开始，然后逐渐慢下来，直至动画迭代结束。

如果这些预定义的时序函数不满足你的需求，可以使用贝塞尔曲线函数自己创建时序函数。这个函数接受四个值，例如：

```
animation-timing-function: cubic-bezier(0.2, 0.4, 0.6, 0.8);
```

贝塞尔曲线通过参数以一定的数学原理定义曲线，常在二维图形应用中使用。附录A中有几个自定义的曲线示例。

贝塞尔曲线有四个参数，分别定义两个手柄的端点位置。在CSS中，手柄从0,0和1,1

两个点延伸出来。前两个参数定义曲线上第一点（或手柄）的 x 坐标和 y 坐标，后两个参数定义第二个手柄的 x 坐标和 y 坐标。x 坐标的值必须在 0 到 1 之间，否则得不到贝塞尔曲线。y 坐标的值不受限制。自己创建贝塞尔曲线时请记住一点：曲线越陡，运动的速度越快；曲线越平，运动的速度越慢[注 1]。

虽然 x 坐标的值必须在 0 和 1 之间，但是 y 坐标的值可以大于 1，也可以小于 0。此时，可以得到一种弹跳效果，动画在值之间来回跳动，而不是一直向一个方向演进。请看下面的时序函数，对应的贝塞尔曲线很奇特，如图 18-2 所示：

```
.snake {
  animation-name: shrink;
  animation-duration: 10s;
  animation-timing-function: cubic-bezier(0, 4, 1, -4);
  animation-fill-mode: both;
}

@keyframes shrink {
  0% {
    width: 500px;
  }
  100% {
    width: 100px;
  }
}
```

注 1：如果想深入了解贝塞尔曲线是如何构建的，强烈建议观看“Cubic Bezier Curves - Under the Hood”视频（*http://vimeo.com/106757336*）。

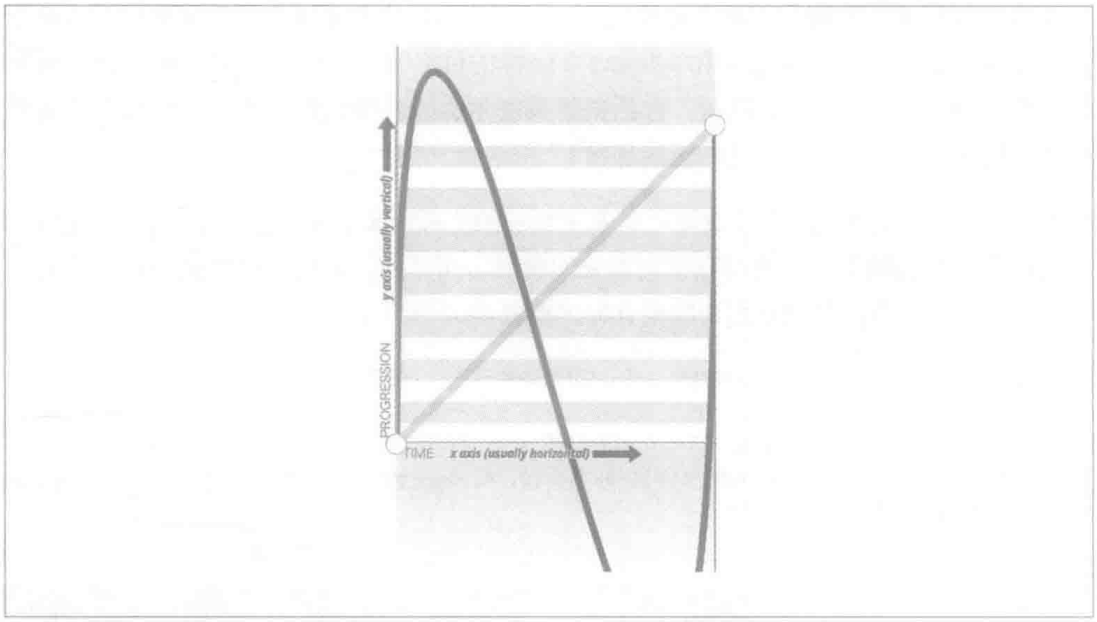


图 18-2：一条奇特的贝塞尔曲线

这里，animation-timing-function 曲线把要以动画形式改变的属性的值移到了 0% 和 100% 关键帧设定的范围之外。在这个示例中，我们把元素的宽度由 500px 缩小为 100px。然而，由于 cubic-bezier 值的关系，元素的宽度在缩小的过程中其实会变的比 0% 关键帧定义的 500px 宽、比 100% 关键帧定义的 100px 窄，如图 18-3 所示。

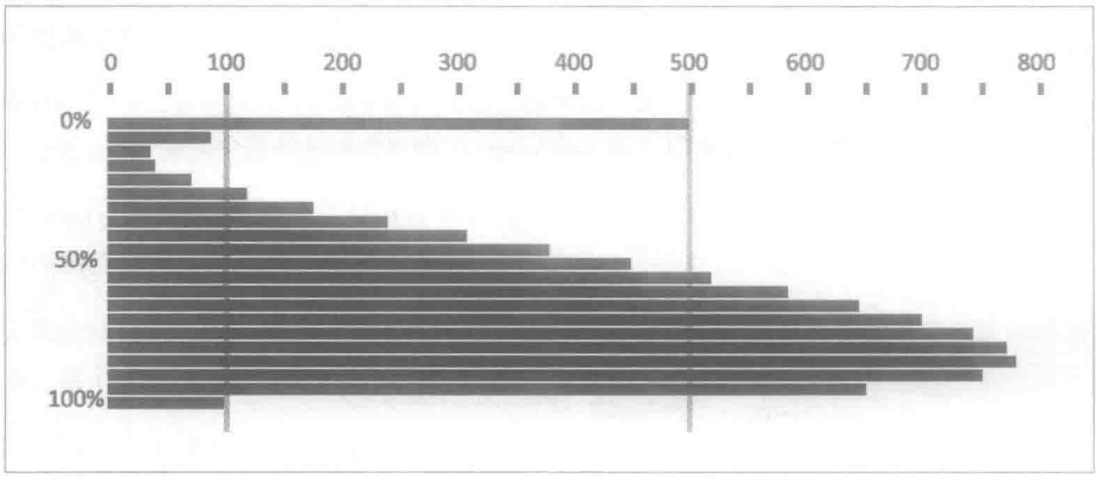


图 18-3：这条奇特的贝塞尔曲线的效果

一开始，这个元素的宽度为 0% 关键帧定义的 500px。然后迅速缩小为约 40px，这比

100% 关键帧定义的 width:100px 要窄。接着，由 40px 慢慢增大到 750px，这比一开始的 500px 宽。最后，迅速缩小到 width: 100px，结束动画的此次迭代。

你可能注意到了，动画创建的曲线与我们定义的贝塞尔曲线是一致的。因为这条 S 曲线超出了常规的范围框，所以在播放动画的过程中元素的宽度会变得比 100px 窄、比 500px 宽。

这条贝塞尔曲线像一条蛇一样，因为一个 y 坐标为正值，一个 y 坐标为负值。如果两个 y 坐标都为大于 1 的正值，或者都为小于 -1 的负值，那么贝塞尔曲线是弧形的，在设定的某个值上部或下部，而不会呈 S 曲线，在两个边界值之间来回跳动。

animation-timing-function 设定的时序函数针对的是正常方向的动画，即由 0% 关键帧向 100% 关键帧演进。反向播放动画时，即由 100% 关键帧向 0% 关键帧演进，时序函数将反转。

还记得前文那个弹跳球示例吗？小球弹跳的过程与实际不太相符，因为那个例子用的是默认的 ease 时序函数。我们可以把 animation-timing-function 设为 ease-in，让小球下落时，在向 100% 关键帧（即最低点）运动的过程中速度越来越快。小球向上弹回时，动画是反向播放的，即由 100% 指向 0%，因此 animating-timing-function 也是反过来的，变成 ease-out，在接近顶点的过程中速度越来越慢：

```
.ball {
  animation-name: bounce;
  animation-duration: 1s;
  animation-iteration-count: infinite;
  animation-timing-function: ease-in;
  animation-direction: alternate;
}

@keyframes bounce {
  0% {
    transform: translateY(0);
  }
  100% {
    transform: translateY(500px);
  }
}
```

步进时序函数

步进时序函数，step-start、step-end 和 steps()，不是贝塞尔曲线，其实根本不是曲线。步进函数定义的是补间效果。steps() 函数在实现小人动画时最有用。

steps() 时序函数把动画分成一系列等长的步幅。steps() 接受两个参数：步数和变化点（稍后细说）。

步数是第一个参数，其值必须是正整数。动画时长将平均分成步数对应的段数。例如，如果动画的持续时间为 1 秒，步数为 5，那么动画的每一步时长为 200 毫秒，元素将在页面中重新绘制 5 次，间隔 200 毫秒，每次间隔播放动画的 20%。

可以通过翻书效果理解步进函数。书中的每一页有一幅图，页与页之间有细微的差异，就像电影的一帧帧画面一样。在快速翻动页面的过程中，图像的变化将形成动画效果。使用子图集，加上 background-position 属性和 steps() 时序函数，可以实现类似的动画。

图 18-4 是一个子图集，各个小图之间有细微的差异，就像画在书本的每一页上一样。

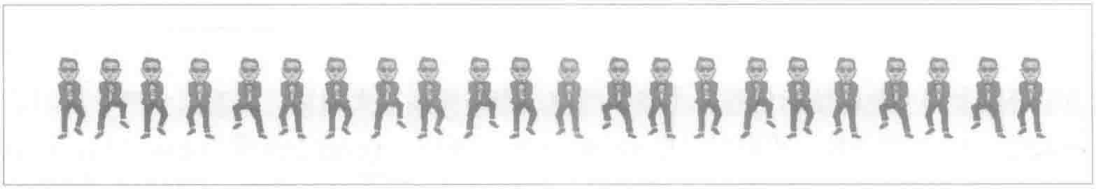


图 18-4：一个舞蹈子图集

我们把稍有不同的多张图像放在一幅大图中，这样的图像称为子图集。子图集中的每个图像是动画中的每一帧。

我们要创建一个容器元素，把尺寸设为子图集中单张图那么大，并把子图集设为容器的背景图。然后，使用 steps() 时序函数，以动画形式改变 background-position，一次只看到子图集上的一幅小图。提供给 steps() 时序函数的步数等于子图集中图像的数量。步数定义完成一次动画背景图要定格多少次。

在图 18-4 中的子图集里，各图像的尺寸为 56 × 100 像素，总尺寸为 1232 × 100 像素。首先，把容器的尺寸设为单张图的大小，即 56 × 100 像素。然后把子图集设为容器的背景。background-position 属性的初始值（默认值）是 topleft，等同于 0 0。默认值是个好选择，不支持 CSS 动画的旧浏览器将显示子图集中的第一张图。

```
.dancer {
  height: 100px;
  width: 56px;
  background-image: url(../images/dancer.png);
  ...
}
```

这里的关键是使用 steps() 改变 background-position 的值，让每一帧显示子图集中的一张图。使用 steps() 时序函数时，背景图不是连续不断滑入的，而是一次次（与步数相等）移入的。

因此，这个动画只需改变 background-position 的左右位置。子图集的宽度为 1232 像素，因此我们要把背景图从 0 0（左上角）移到 -1232px 0，把子图集完全放到 56 × 100 像素的 div 视区以外。

把背景图的位置设为 -1232px 0，子图集将完全移到左边，超出容纳块的视区。在 100% 记号处，通过 56 × 100 像素的 div 区域完全看不到子图集，除非 background-repeat 允许沿 x 轴平铺图像。但我们并不希望出现这样的结果！

我们想要的效果是：

```
@keyframes dance_in_place {
  from {
      background-position: 0 0;
  }
  to {
      background-position: -1232px 0;
  }
}

.dancer {
  ...
  background-image: url(../images/dancer.png);
  animation-name: dance_in_place;
  animation-duration: 4s;
  animation-timing-function: steps(22, end);
  animation-iteration-count: infinite;
}
```

看似复杂的动画我们轻易就实现了：像在书本上一样，在一帧上一次看到子图集中的一张图。关键帧只需移动背景图即可。▶

以上讲的是第一个参数，即步数。第二个参数可取这两个值中的一个：start 和 end。这个参数指明第一步的变化发生在间隔的开头还是结尾。默认值 end 在第一步的结尾变化。也就是说，对 200 毫秒的步长来说，动画中的第一次变化等到 200 毫秒之后才会发生。设为 start 时，第一次变化在第一步的开头发生，即动画开始后立即变化。图 18-5 根据下述样式画出了这两个值的时间线。

```
@keyframes grayfade {
    from {background-color: #BBB;}
```

```
    to {background-color: #333;}
}

.quickfader {animation: grayfade 1s steps(5,start) forwards;}
.slowfader  {animation: grayfade 1s steps(5,end) forwards;}
```

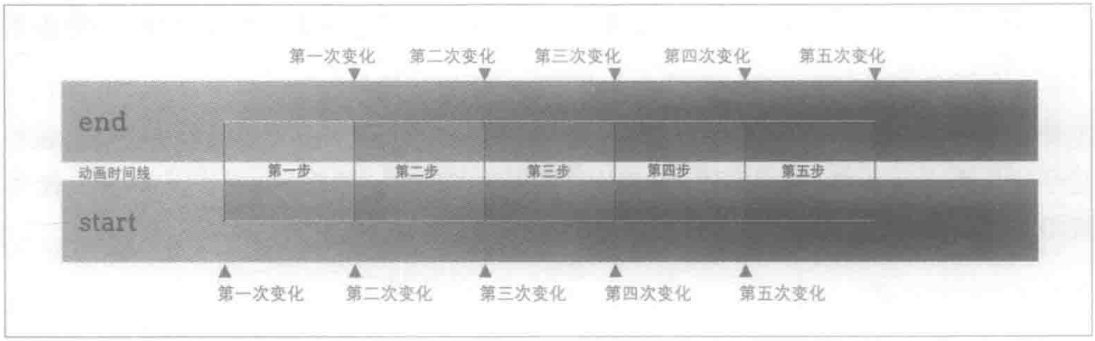


图 18-5：图解 start 和 end 的变化点

时间线上的各个方框表示那一步的间隔中元素的背景色。注意，在 end 时间线中，第一段间隔中元素的背景色与动画开始前一样。这是因为动画等到第一帧的末尾才改变颜色（变成“第一步”与“第二步”之间的颜色）。

而在 start 时间线中，第一段间隔在开头立即改变颜色，瞬间由原背景色变成“第一步”与“第二步”之间的颜色。这就好像向前跳一段间隔，把 end 时间线中“第二步”的背景色移到了 start 时间线中“第一步”的位置。

在两个动画的结尾也能看到类似的效果，在 start 时间线中，第五步的背景色与元素的最终背景色一样。在 end 时间线中，第五步的背景色是“第四步”与“第五步”之间的颜色，直到“第五步”末尾，即动画结束时才变成最终的颜色。

第二个参数的作用很难说得清。你可以这样理解：在动画的正常方向中，start 值相当于“跳过”0% 关键帧，因为第一次变化在动画开始后立即发生，而 end 值相当于“跳过”100% 关键帧。



如果想像上述示例一样在动画结束后保留最终的背景色，而不是重置为起初的颜色，要使用关键字 forwards。详情参见 18.4.9 节。

step-start 值等同于 steps(1,start)，只有一步，显示 100% 关键帧。step-end 值等同于 steps(1,end)，只显示 0% 关键帧。

再添加一个动画

回到前面的小人动画。那个动画是一个小人在跳舞，不过多数人在跳舞时会摆动身体。我们可以再添加一个动画，实现左右和前后运动效果：

```
@keyframes move_around {
  0%, 100% {
    transform: translate(0, -40px) scale(0.9);
  }
  25%  {
    transform: translate(40px, 0)  scale(1);
  }
  50%  {
    transform: translate(0, 40px)  scale(1.1);
  }
  75%  {
    transform: translate(-40px, 0) scale(1);
  }
}
```

我们又定义了一个关键帧动画，名为 move_around。▶ 然后在各动画属性中添加一个值，用逗号与现有的值分开，为跳舞的人附加第二个动画：

```
.dancer {
    ...
    background-image: url(../images/dancer.png);
    animation-name: dance_in_place, move_around;
    animation-duration: 4s, 16s;
    animation-timing-function: steps(22, end), steps(5, end);
    animation-iteration-count: infinite;
}
```

注意，除 animation-iteration-count 之外，其他各动画属性都有两个以逗号分隔的值。还记得吗，如果动画属性列出的逗号分隔的值与 animation-name 属性列出的数量不一致，缺少的值将从现有值中复制，直至数量相等。我们希望两个动画都无限持续下去，因此只提供了一个 infinite 值。浏览器将重复 animation-iteration-count 列出的值（这里列出的是一个 infinite），直至数量与声明的动画数相同。

以动画形式改变时序函数

animation-timing-function 属性不支持动画，但是可以放在关键帧中，修改动画的当前时序。

与支持动画的属性不同，animation-timing-function 属性的值无法内插。如果放在 @keyframes 规则中定义的关键帧里，播放到当前关键帧时，其中声明的属性将使用 animation-timing-function 声明的新时序函数，如图 18-6 所示。

```
@keyframes width {
  0% {
    width: 200px;
    animation-timing-function: linear;
  }
  50% {
    width: 350px;
    animation-timing-function: ease-in;
  }
  100% {
    width: 500px;
  }
}
```

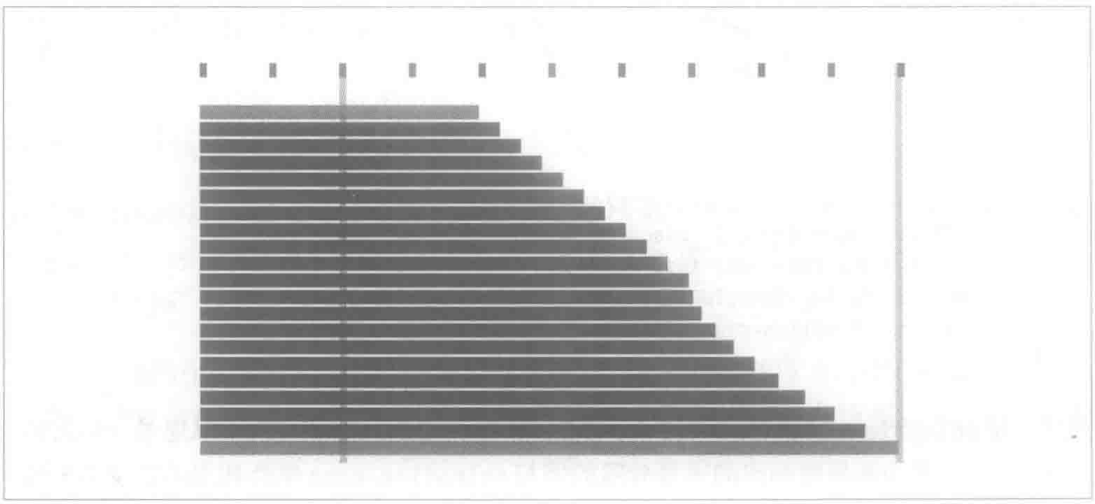


图 18-6：在动画中途改变时序函数

在上述示例中，动画播放到一半时，我们把 width 属性的线性时序变成了渐入时序，如图 18-6 所示。ease-in 时序从时序函数变化的那个关键帧开始生效。

在 to 或 100% 关键帧中设定 animation-timing-function 对动画没有任何影响。在其他关键帧中改变时序函数，动画将从 animation-timing-function 声明所在的关键帧开始使用新的时序函数，直到下一个覆盖默认时序或有 animation-timing-function 声明的关键帧。

关键帧中声明的 animation-timing-function，只对当前关键帧块中的属性有影响。只有播放到包含那些属性的关键帧时，新的时序函数才生效。届时，时序函数将变成那个块中声明的值，或者变回最初为元素分配的时序函数。以前文所举的 W 动画为例：

```
@keyframes W {
    from {
      left: 0;
      top: 0;
    }
    25%, 75% {
      top: 100%;
    }
    50% {
      top: 50%;
    }
    to {
      left: 100%;
      top: 0;
    }
}
```

从概念上将，应用到元素或伪元素上的动画会为 @keyframes 规则中的每个属性创建关键帧，好似单独播放每个属性的动画一样。对 W 动画来说，好像是同时播放两个动画——W_part1 和 W_part2：

```
@keyframes W_part1 {
    from, to {
      top: 0;
    }
    25%, 75% {
      top: 100%;
    }
    50% {
      top: 50%;
    }
}

@keyframes W_part2 {
    from {
```

```
        left: 0;
      }
      to {
        left: 100%;
      }
  }
```

在某个关键帧中设定的 animation-timing-function，只应用到那个关键帧中定义的属性上：

```
@keyframes W {
    from {
      left: 0;
      top: 0;
    }
    25%, 75% {
      top: 100%;
    }
    50% {
      animation-timing-function: ease-in;
      top: 50%;
    }
    to {
      left: 100%;
      top: 0;
    }
}
```

上述代码只把 top 属性的 animation-timing-function 变成 ease-in，left 属性不受影响，也就是说只对 W 动画的 W_part1 部分起作用，而且只影响那个动画的中点到 75% 记号处这一段。

然而，在下述动画中，animation-timing-function 不起作用，因为所在的关键帧块中没有声明任何属性：

```
@keyframes W {
    from {
      left: 0;
      top: 0;
    }
    25%, 75% {
      top: 100%;
    }
    50% {
      animation-timing-function: ease-in;
    }
```

```
    50% {
      top: 50%;
    }
    to {
      left: 100%;
      top: 0;
    }
}
```

那么，在动画的中途改变时序函数有什么用呢？在弹跳球示例中，所处的环境没有摩擦力，小球一直弹跳，没有动量损失。小球下降时加速，上升时减速，因此每隔一次迭代，动画的播放方向由 normal 变成 reverse，时序函数则由 ease-in 变成 ease-out。

现实中，摩擦力是存在的，动量有损失。小球不可能无限弹跳下去。如果想让小球弹跳的过程符合自然规律，每次触底弹起的高度要矮一些，因为有能量损耗。为此，我们要定义一个弹跳多次的动画，每次弹跳都有动量损失，而且在顶点和最低点之间来回切换 ease-in 和 ease-out：

```
@keyframes bounce {
  0% {
    transform: translateY(0);
    animation-timing-function: ease-in;
  }
  30% {
    transform: translateY(100px);
    animation-timing-function: ease-in;
  }
  58% {
    transform: translateY(200px);
    animation-timing-function: ease-in;
  }
  80% {
    transform: translateY(300px);
    animation-timing-function: ease-in;
  }
  95% {
    transform: translateY(360px);
    animation-timing-function: ease-in;
  }
  15%, 45%, 71%, 89%, 100% {
    transform: translateY(380px);
    animation-timing-function: ease-out;
  }
}
```

这个动画的弹跳高度逐渐减小，最终停止。

这个动画只有一次迭代，因此不能依赖 animation-direction 改变时序函数。虽然每次弹跳都有动量损失，但是我们要保证小球在下降时受重力作用有加速，而在接近顶点的过程中速度逐渐减慢。因为只有一次迭代，所以我们在关键帧中声明 animation-timing-function，控制时序。每次到达顶点，把时序切换为 ease-in，而到达最低点时（即触底反弹），又切换为 ease-out。

18.4.8 设置动画的播放状态

如果想暂停和继续播放动画，可以使用 animation-play-state 属性定义动画是播放还是暂停的。

animation-play-state	
取值	[running | paused]#
初始值	running
适用于	所有元素，以及 ::before 和 ::after 伪元素
计算值	指定的值
继承性	否
动画性	否

设为默认值 running 时，动画正常播放。设为 paused 时，动画暂停。此时，动画依然应用到元素上，只是在暂停前播放到的位置停住。如果在迭代中途停止，已经发生变化的属性停在当前的值。再次设为 running，或者回到默认值，动画从停止的位置继续播放，就像控制动画的“时钟”停止后又开始行走了一样。

如果在动画的延迟阶段设置 animation-play-state: paused，延迟时钟也暂停，把 animation-play-state 设为 running 之后继续计时。

18.4.9 动画的填充模式

animation-fill-mode 属性定义动画播放结束后是否应用原来的属性值。

animation-fill-mode

取值	[none | forwards | backwards | both]#
初始值	none
适用于	所有元素，以及 ::before 和 ::after 伪元素
计算值	指定的值
继承性	否
动画性	否

这个属性的作用体现在，默认情况下动画所做的改动只在动画播放的过程中有效，一旦动画结束，属性将还原为动画之前的值。因此，如果动画把背景由红色变成蓝色，（默认情况下）动画结束后背景将还原为红色。

类似地，如果 animation-delay 为正值，动画不会立即改变属性的值，而是在延迟结束，触发 animationstart 事件后才开始改变。

使用 animation-fill-mode 属性可以定义触发 animationstart 事件之前和触发 animationend 事件之后动画如何影响元素。我们可以设定，让 0% 关键帧中设定的属性值在延迟期间就应用到元素上，或者在触发 animationend 事件之后继续应用到元素上。

animation-fill-mode 属性的默认值是 none，意思是动画不播放就没有效果：0% 关键帧（或反向播放动画时的 100% 关键帧）块中设定的属性值在 animation-delay 结束、触发 animationstart 事件之前不应用到元素上。

设为 backwards 时，0% 或 from 关键帧（如果有的话）中定义的属性值在动画应用到元素上的那一刻就起作用。0% 关键帧（animation-direction 属性的值为 reversed 或 reversed-alternate 时，为 100% 关键帧）定义的属性值立即应用，而不等待 animation-delay 时间结束。

forwards 值的意思是动画播放结束后，即 animation-iteration-count 设定的迭代次数全部结束，触发了 animationend 事件，当时的属性值继续应用在元素上。如果 animation-iteration-count 的值是整数，应用的属性值来自 100% 关键帧；如果最后一次迭代是反向的，则来自 0% 关键帧。

both 值包含 backwards 和 forwards 两个值的效果，即把动画附加到元素上之后立即应用属性值，而且触发 animationend 事件之后，属性的值得以保留。

如果 animation-iteration-count 的值不是整数，而是浮点数，那么最后一次迭代不以 0% 或 100% 关键帧结尾，而是在一次动画循环的中途停止播放。如果把 animation-fill-mode 设为 forwards 或 both，触发 animationend 事件时的属性值将应用到元素上。例如 animation-iteration-count 为 6.5，animation-timing-function 为 linear，触发 animationend 事件时，50% 记号中的属性值将保留下来（不管有没有显式声明 50% 关键帧），好似在那一刻把 animation-play-state 设为 pause。

例如，对下述代码来说：

```
@keyframes move_me {
  0% {
    transform: translateX(0);
  }
  100% {
    transform: translateX(1000px);
  }
}

.moved {
  transform: translateX(0);
  animation-name: move_me;
  animation-duration: 10s;
  animation-timing-function: linear;
  animation-iteration-count: 0.6;
  animation-fill-mode: forwards;
}
```

这个动画只迭代 0.6 次。这个持续 10 秒的线性动画将在 60% 记号处停止，即动画播放了 6 秒，元素向右平移了 600 像素。animation-fill-mode 的值为 forwards 或 both 时，元素向右平移 600 像素后动画停止，而且就停在原位置右侧 600 像素的位置上。元素将无限期停在那个位置，直到从元素上拆离动画。如果没有 animation-fill-mode:forwards，类为 moved 的元素将立即回到原始位置，因为 moved 选择符后的代码块中定义了 translateX(0)。



在 Safari 9 及之前的版本中，forwards 和 both 都把 100% 关键帧中的属性值应用到元素上，而不管最后一次迭代的方向，也不管动画是以 100% 关键帧结束的还是在其他位置结束的。▶ 对上述示例来说，在 Safari9 中，.moved 元素向右平移 600 像素后停止，然后立即跳到右侧 1000 像素处，然后停在那里。在 Safari9 及之前的版本中，不管最后一次迭代的方向是 normal 还是 reverse，也不管动画在一次循环的 25% 还是 75% 结束，只要声明了 animation-fill-mode:forwards，动画都会跳到 100% 关键帧，停在那里。这是遵守旧版规范的行为，希望浏览器的开发人员能跟上新版规范和其他主流浏览器的步伐。

18.5 写为一个属性

使用 animation 简写属性，无需分别定义 8 个属性，在一行声明中就能为元素定义全部动画属性。animation 属性的值是一个列表，以空格分隔，分别对应各个单独属性。如果要在元素或伪元素上应用多个动画，在列出的各动画之间加上逗号。

animation

取值	[*<animation-name>* ‖ *<animation-duration>* ‖ *<animation-timing-function>* ‖ *<animation-delay>* ‖ *<animation-iteration-count>* ‖ *<animation-direction>* ‖ *<animation-fill-mode>* ‖ *<animation-play-state>*]#
初始值	0s ease 0s 1 normal none running none
适用于	所有元素，以及 ::before 和 ::after 伪元素
计算值	指定的值
继承性	否
动画性	否

animation 简写属性的值为前述各动画属性的值，包括 animation-duration、animation-timing-function、animation-delay、animation-iteration-count、animation-direction、animation-fill-mode、animation-play-state 和 animation-name。例如，下面两个规则是完全等效的：

```
#animated {
  animation: 200ms ease-in 50ms 1 normal running forwards slidedown;
}
#animated {
  animation-name: slidedown;
  animation-duration: 200ms;
  animation-timing-function: ease-in;
  animation-delay: 50ms;
  animation-iteration-count: 1;
  animation-fill-mode: forwards;
  animation-direction: normal;
  animation-play-state: running;
}
```

在 animation 简写属性中，无需声明所有值；未声明的值将被设为默认值（或初始值）。前面示例中的简写属性比较长，其中有三个属性的值是默认值，因此无需写出。

特别注意，如果在简写属性中没有声明全部八个值，未声明的属性将采用相应属性的初始值。各单独属性的初始值（或默认值）如下：

```
animation-name: none;
animation-duration: 0s;
animation-timing-function: ease;
animation-delay: 0;
animation-iteration-count: 1;
animation-fill-mode: none;
animation-direction: normal;
animation-play-state: running;
```

简写属性中有些值的顺序是重要的。其一，有两个时间相关的属性，分别为 *<animation-duration>* 和 *<animation-delay>*。如果列出两个时间值，第一个始终为持续时间。如果有第二个，解释为延迟。

其二，animation-name 的位置也很重要。如果动画标识符的值与某个动画属性的值相同（不应该这么做，这里只是假设），那么 animation-name 应该放在最后。在之前位置上的关键字，只要是某个动画属性的有效值，例如 ease 或 running，就会假定为相应属

性的值，而不会解释为 animation-name 的值。注意，none 基本上是唯一不能作为动画名称的词。

下述代码：

```
#failedAnimation {
    animation: paused 2s;
}
```

等效于：

```
#failedAnimation {
    animation-name: none;
    animation-duration: 2s;
    animation-delay: 0;
    animation-timing-function: ease;
    animation-iteration-count: 1;
    animation-fill-mode: none;
    animation-direction: normal;
    animation-play-state: paused;
}
```

paused 是有效的动画名称。在上述简写属性中，你可能以为附加到元素上的是名为 paused、持续 2s 的动画。其实不然。这是因为，简写属性中的词首先检查是不是 animation-name 之外某个动画属性的有效值，而 paused 是 animation-play-state 属性的有效值。

下述代码：

```
#anotherFailedAnimation {
    animation: running 2s ease-in-out forwards;
}
```

等效于：

```
#anotherFailedAnimation {
    animation-name: none;
    animation-duration: 2s;
    animation-delay: 0s;
    animation-timing-function: ease-in-out;
    animation-iteration-count: 1;
    animation-fill-mode: forwards;
    animation-direction: normal;
```

```
    animation-play-state: running;
}
```

开发者可能定义了一个名为 running 的关键帧动画。但是，浏览器将把 running 赋予 animation-play-state 属性，而不是 animation-name 属性。由于没有声明 animation-name，所以没有动画附加到元素上。绕开这个问题的方法是：

```
#aSuccessfulIfInadvisableAnimation {
    animation: running 2s ease-in-out forwards running;
}
```

这个规则把第一个 running 赋值给 animation-play-state 属性，把第二个 running 赋值给 animation-name 属性。再次声明，不建议这么做。因为混淆和出错的风险太大了。

根据上述说明，animation: 2s 3s 4s; 看起来是有效的声明，像是这样定义的：

```
#invalidName {
    animation-name: 4s;
    animation-duration: 2s;
    animation-delay: 3s;
}
```

但是，18.2 节说过，4s 不是有效的标识符。标识符不能以数字开头，除非转义。如果想变为有效的声明，可以写成 animation:2s 3s \4s;。

如果想在一个元素或伪元素上附加多个动画，以逗号分开各动画：

```
.snowflake {
  animation: 3s ease-in 200ms 32 forwards falling,
             1.5s linear 200ms 64 spinning;
}
```

雪花将旋转下落 96 秒，每 3 秒旋转两次。▶最后一次循环结束后，雪花将停留在 falling 动画的 100% 关键帧。我们为 falling 动画声明了六个动画属性，为 spinning 动画声明了五个动画属性，二者之间以逗号分隔。

你可能经常见到别人在第一个位置声明动画名称（这样可读性更好，因为动画属性的关键字值也是有效的关键帧标识符），但这不是最佳实践。鉴于此，我们才把动画名称放在最后。

综上，使用 animation 简写属性是个不错的主意。但是要记住，持续时间、延迟和名称的位置很重要，而且省略的值将被设为默认值。此外，最好别使用关键字作为动画的标识符。

18.6 动画、特指度和优先顺序

动画在特指度和层叠规则方面有些特殊，确定把哪个属性值应用到元素上时，会错误地覆盖层叠样式中的其他所有值（截至 2017 年年末）。

18.6.1 特指度和 !important

一般来说，通过 ID 选择符应用到元素上的属性，其权重为 1-0-0，这应该比元素选择符的 0-0-1 权重高。然而，如果属性的值是以关键帧动画形式改变的，应用到元素上时，属性及其值好像是通过行内样式添加的一样。

目前，在所有支持动画的浏览器中，关键帧设定的属性值都好像是行内声明的一样，而且加上了 !important，例如 <divstyle="keyframe-property:value!important">。根据规范，这是错误的行为。规范指出，“动画覆盖常规的规则，但是能被带有 !important 的规则覆盖”。在 2017 年年末之前实现的浏览器中，这是个 bug，最终应该会被修正。或者，规范可能会做修改。

尽管如此，不要在动画声明块中使用 !important。这样做是无效的，反而会导致相应的属性和值被忽略。

18.6.2 动画顺序

如果有多个动画为同一个属性指定了不同的值，最后一个应用的动画将覆盖之前动画中声明的值：

```
#colorchange {
  animation-name: red, green, blue;
  animation-duration: 11s, 9s, 6s;
}
```

在上述代码示例中，如果 red、green 和 blue 三个关键帧动画把 color 属性改变为名称对应的颜色，把 animation-name 和 animation-duration 属性应用到 #colorchange 元素上之后，前六秒 blue 动画中的属性值生效，接下来的三秒 green 动画中的属性值生效，最后两秒 red 动画中的属性值生效，而后回归属性的默认值。▶这里，如果 blue 关键帧动画中的 0% 关键帧没有声明 color 属性，颜色将按顺序取自 green 动画、red 动画和元素的 currentColor 值。缺少 100% 关键帧时也是如此。

在动画开始之前，元素的默认属性不受影响，而且在动画结束后，属性将还原为原来的值，除非把 animation-fill-mode 设为 none 以外的值。如果加上 animation-fill-

mode:both，color 始终为蓝色，因为最后一个动画（blue）覆盖了前面的 green 动画，而它又覆盖了开头的 red 动画。

18.6.3 display: none; 对动画迭代的影响

如果把元素的 display 属性设为 none，元素或其后代上的动画迭代将停止，好似把动画从元素上拆离了一样。把 display 属性变为某个可见的值之后，所有动画属性重新应用到元素上，动画将从头开始播放：

```
.snowflake {
  animation: spin 2s linear 5s 20;
}
```

在这个示例中，雪花将旋转 20 次，每次用时 2 秒钟，而且第一次旋转等待 5 秒才开始。如果 15 秒后把 .snowflake 元素的 display 属性设为 none，元素在消失之前已经完成了 5 次旋转（等待 5 秒钟，5 次旋转各用 2 秒钟）。如果把 display 属性的值改成 none 以外的值，动画将从头开始播放：延迟 5 秒之后开始旋转 20 次。从视线中消失之前已经迭代了多少次循环并没有关系。

18.6.4 动画和 UI 线程

CSS 动画在用户界面（UI）线程中的优先级最低。如果页面加载时附加了多个动画，而且 animation-delay 为正值，延迟结束后倘若 UI 线程不可用，动画不会播放。

有如下假设：

- 所有动画都需要 UI 线程（也就是说不使用 GPU，参见本章前面“动画链”一节）。
- 有 20 个动画，animation-delay 属性的值分别为 1s、2s、3s、4s 等，每隔一秒开始播放一个动画。
- 页面或应用的加载时间很长，在页面中绘制出要应用动画效果的元素与 JavaScript 下载完毕、解析并执行之间相隔 11 秒。

那么，在 UI 线程可用时前 11 个动画的延迟已经结束，因此这 11 个动画将同时开始播放。余下的动画则每隔一秒开始一个。

18.7 癫痫和前庭功能失调



虽然可以使用动画创建不断变化的内容，但是动态变化的内容可能导致某些用户癫痫发作。谨记这一点，一定要保证网站对癫痫和其他神经系统疾病患者是可访问的。

笔者很少以警告开始一节，但这里有一定的正当理由。视觉变化，尤其是快速的视觉变化，可能导致癫痫病人发作。此外，还有可能导致前庭功能失调（晕动病）人群产生多种不适。

在本书即将付梓的 2017 年年末，浏览器新增了一个媒体查询：`prefers-reduced-motion`。创作人员可以通过这个媒体查询为在浏览器或其他设备中设置了“减少运动效果”或类似偏好设置的用户定义样式。强烈建议像下面这样做：

```
@media (prefers-reduced-motion) {
  * {animation: none !important; transition: none !important;}
}
```

这个规则禁用全部动画和过渡，假设没有动画使用 `!important`（当然也不应该这么做）。这不是细致或完美的解决方案，但是迈出了第一步。反过来，我们可以把所有动画和过渡单独放在一处，针对没有启用“减少运动效果”的用户：

```
@media not (prefers-reduced-motion) {
  /* 所有动画和过渡 */
}
```

不是所有动画都是危险的，会让人失去认知能力，而且可能有必要为全部用户呈现少量的动画。此时，可以使用 `prefers-reduced-motion` 缓和对 UI 而言必不可少的动画，并把仅用作装饰的动画关闭。

18.8 动画事件及其前缀

本节回顾可通过 DOM 事件监听器访问的动画事件，以及何时需要使用前缀。

18.8.1 animationstart

animationstart 事件在动画开始时触发。如果动画有延迟，延迟结束后才触发这个事件。如果没有延迟，动画应用到元素上之后立即触发 animationstart 事件。即便一次迭代也没有，animationstart 事件依然触发。如果在一个元素上应用多个动画，每个有效的关键帧动画都将触发一次 animationstart 事件。一般来说，一个有效的动画标识符触发一次 animationstart 事件。

```
#colorchange {
  animation: red, green, blue;
}
```

在这个示例中，只要 red、green 和 blue 关键帧动画是有效的，尽管看不到实际的动画效果（因为持续时间都默认为 0s），但是却会触发三次 animationstart 事件，一个动画名称对应一次。

在需要为动画属性加上 -webkit- 前缀的浏览器中（基本上是 Safari 8 及之前的版，以及 Android 4.4.4 及之前的版本），animationstart 要写做 webkitAnimationStart。注意，前缀是 -webkit-，事件是驼峰式。最好默认使用不带前缀的句法，仅在有必要时回落为带前缀的版本。

18.8.2 animationend

animationend 事件在动画结束时触发。应用的每个动画只触发一次 animationend 事件。如果在一个元素（例如前例中的 #colorchange）上应用三个动画，animationend 事件将触发三次，每个动画结束时触发一次。上述示例没有为动画声明持续时间，可是 animationend 事件的时序通常等于下述等式求出的结果：

```
(animation-duration * animation-iteration-count) + animation-delay = time
```

即使一次也不迭代，应用的每个动画也会触发一次 animationend 事件。如果 animation-iteration-count 的值为 infinite，animationend 事件永不触发。

在需要为动画属性加上 -webkit- 前缀的浏览器中，animationend 写做 webkitAnimationEnd。

18.8.3 animationiteration

animationiteration 事件在动画的一次迭代结束之后和下一次迭代开始之前触发。如果没有迭代，或者迭代次数小于或等于一，animationiteration 事件不触发。如果迭代无数次，animationiteration 事件触发无数次，除非没有设置持续时间，或者设为 0s。

一个动画名称触发一次 animationstart 和 animationend 事件，而 animationiteration 事件可能触发多次，也可能一次也不触发，这取决于迭代次数。注意，这个事件在两次循环之间触发，但不与 animationend 事件同时触发。也就是说，如果 animation-iteration-count 的值为整数，animationiteration 事件的触发次数一般比 animation-iteration-count 属性的值少一次，当然前提是负延迟的绝对值小于持续时间。

18.9 打印动画

虽然印刷纸张上无法呈现“动画”效果，但是打印有动画效果的属性时，将使用相关的属性值。在纸上看不到动画效果，但是如果动画把元素的 border-radius 变成了 50%，那么打印出来后，元素的 border-radius 就是 50%。

第 19 章

滤镜、混合、裁剪和遮罩

过去十年，CSS 积累了一些有趣的新功能。例如，创作人员可以使用视觉滤镜调整元素的外观，以不同的方式与背后的内容混合，以及隐藏部分内容，改变元素的表现。这些功能看似毫不相干，但是却有一个共同点：改造元素的方式在以前十分困难或根本不可能。

19.1 CSS 滤镜

年长的创作人员可能还记得，很久以前，微软在自家浏览器中为 CSS 提供了 `filter` 属性支持，用于添加 DirectX 视觉效果。不久之后，CSS 也增加了 `filter` 属性，概念与微软实现的类似，但两者的作用并不相同。经过几次修订之后，CSS 定义了一些内置的视觉效果滤镜，而且允许从外部文件中加载滤镜。

filter

取值	`[ none | blur() | brightness() | contrast() | drop-shadow() | grayscale() | hue-rotate() | invert() | opacity() | sepia() | saturate() | url() ]#`
初始值	none
适用于	所有元素（在 SVG 中适用于所有图形元素和除 `<defs>` 元素之外的所有容器元素）

计算值	声明的值
继承性	否
动画性	是

从取值句法可以看出，滤镜函数可以有多个，之间以空格分隔，各滤镜按列出的顺序应用。因此，对filter:opacity(0.5)blur(1px);声明来说，先应用不透明度，得到半透明效果，然后再做模糊处理。如果调换两个函数的位置，应用的顺序随之变化，即先对完全不透明的元素做模糊处理，然后再变成半透明。

CSS规范在讨论filter时经常提到“输入图像”，但这并不意味着filter属性只能用在图像上。任何HTML元素都可以使用滤镜，而且SVG中的所有图形元素也可以使用。“输入图像”是应用滤镜之前元素渲染结果的视觉副本。滤镜就应用到这个输入上，得到结果后再渲染到显示媒体上。

允许使用的值其实都是函数，各个函数可用的参数值不尽相同。为了便于理解，笔者对这些函数做了不太严格地分类。

19.1.1 基本滤镜

这些滤镜的作用基本上从名称上就能看出，包括模糊、投影和不透明度变化。

blur(<*length*>)

使用高斯模糊对元素的内容做模糊处理，标准偏差由<length>值定义，设为0时不对元素做模糊处理。标准偏差不能设为负值。

opacity([<*number*> | <*percentage*>])

把透明度滤镜应用到元素上，与opacity属性十分相似，0表示完全透明，1或100%则不对元素做任何改动。不允许使用负值。值可以大于1或100%，不过计算最终值时将取整为1或100%。



规范明确指出，filter:opacity()不是opacity属性的替代品或简写形式。其实两者可以同时应用到元素上，得到的结果是一种双重透明效果。

drop-shadow(<*length*>{2,3} <*color*>?)

创建与元素的 alpha 通道形状一致的投影，带模糊效果，而且可以指定颜色。长度和颜色的处理方式与 box-shadow 属性一样，因此前两个 <*length*> 值可以为负数，而第三个（定义模糊半径）不可以。如未提供 <*color*> 值，使用的颜色与元素的 color 属性的计算值一样。

这三个 filter 函数的单独效果以及组合效果如图 19-1 所示。

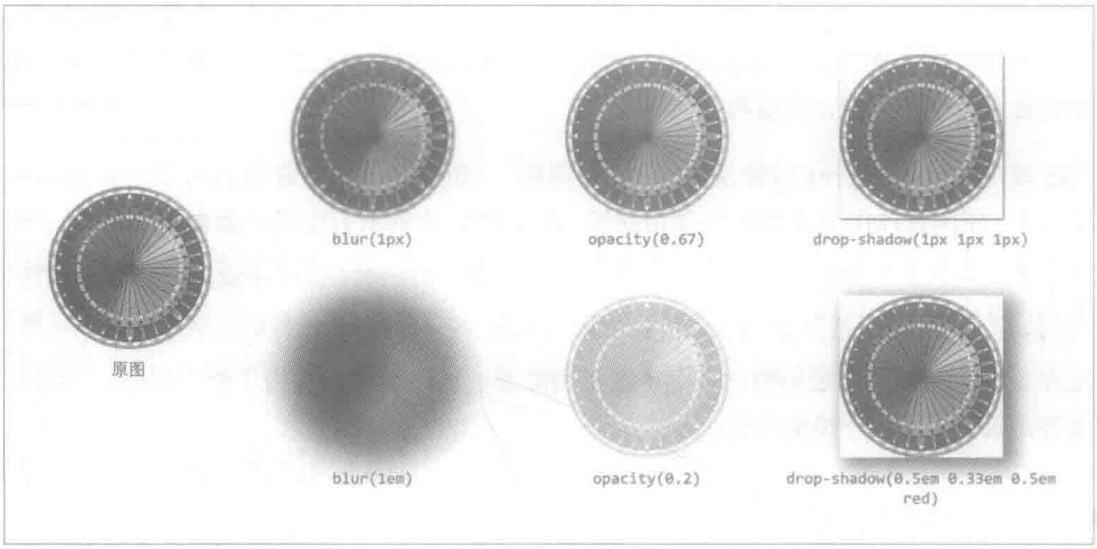


图 19-1：基本滤镜的效果

继续之前，有两点需要进一步说明。首先是 drop-shadow() 的具体运作方式。只看图 19-1，容易误以为投影是沿元素框的边界放置的，因为图中的投影出现在一个方框四周。但这只是因为演示用的图像是 PNG 格式，即光栅图像，而更重要的原因是，这个图像没有 alpha 通道，也就是说白色部分是不透明的。

如果图像有透明部分，drop-shadow() 将用透明部分计算投影。GIF89a、PNG、JPEG2000、SVG，以及其他支持 alpha 通道的图像格式都支持透明度。在有透明部分的图像上应用投影滤镜的效果如图 19-2 所示。

注意，图 19-2 中的最后一个图像有两个投影，这是通过下述声明实现的：

```
filter: drop-shadow(0 0 0.5em yellow) drop-shadow(0.5em 0.75em 30px gray);
```

可以像这样串联任意个滤镜。举个例子：

```
filter: blur(3px) drop-shadow(0.5em 0.75em 30px gray) opacity(0.5);
```

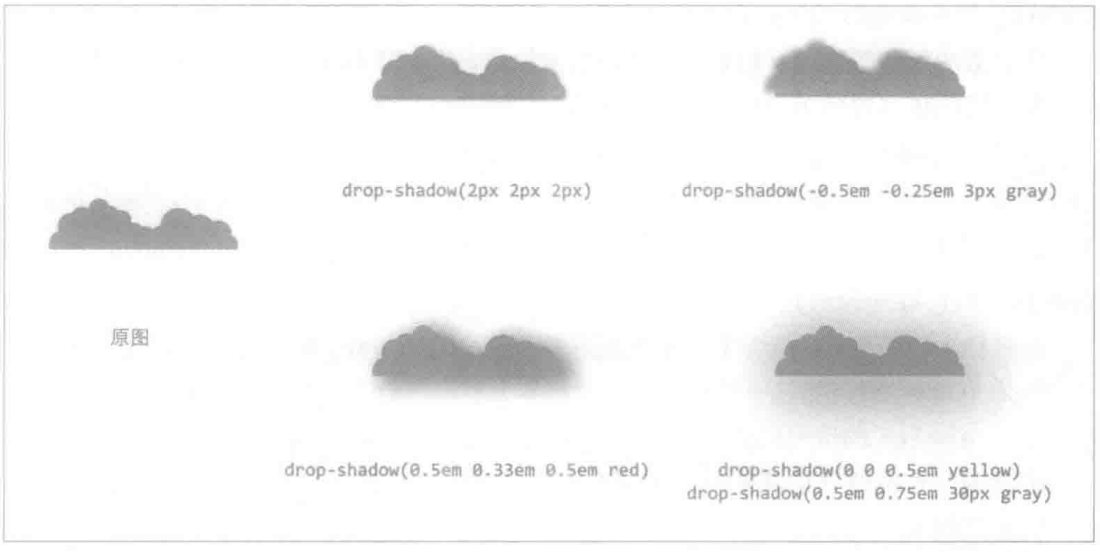


图 19-2：alpha 通道对投影效果的影响

得到的元素有点模糊，带有投影，而且是半不透明的。对读者来说这可能不是最好的效果，举这个例子只是为了说明我们可以这么做。所有 filter 函数都可以像这样串联，包括本节涵盖的，以及后文即将介绍的。

19.1.2 颜色滤镜

这一组filter函数以某种方式调整元素的颜色表现。有的函数很简单，只是滤去某些颜色，有的则很复杂，以一定的角度变换全部颜色。

注意，下述四个函数中的前三个都接受 *<number>* 或 *<percentage>* 值，而且不能为负值。

grayscale([*<number>* | *<percentage>*])

把元素的颜色变成指定的灰阶。值为 0 时，元素没有任何变化；值为 1 或 100% 时，元素完全变成灰度。

sepia([*<number>* | *<percentage>*])

把元素的原色变成指定的墨色调（sepia 是红褐色，按照维基百科的定义，等价于 sRGB 色彩空间中的 #704214 或 rgba(112,66,20)）。值为 0 是，元素没有任何变化；值为 1 或 100% 时，元素完全变成褐色。

invert([<*number*> | <*percentage*>])

把元素的所有颜色做反相处理。反相的方法是，拿 255 或 100% 减去颜色的 R、G 和 B 通道的值（得到 0~255 或 0%~100% 之间的值）。例如，rgb(255,128,55) 反相后变成 rgb(0,127,200)，rgb(75%,57.2%,23%) 反相后变成 rgb(25%,42.8%,77%)。值为 0 时，元素没有任何变化；值为 1 或 100% 时，颜色完全反相。值为 0.5 或 50%，在色彩空间的中点停止反相，得到均匀的灰色。

hue-rotate(<*angle*>)

在色轮上旋转色相，而饱和度和明度保持不变。值为 0deg 时，元素没有任何变化。值为 360deg 时（旋转一整圈），元素基本上也没有什么变化，不过值得以保留下来了。<angle> 的值可以大于 360deg，也可以为负值。为负值时逆时针旋转，而为正值时顺时针旋转（也就是说，旋转的方式与指南针一样，0° 在最上边，角度沿顺时针方向增大）。

图 19-3 给出了上述几个 filter 函数的示例，不过在彩色媒体上才能看到完整的效果。

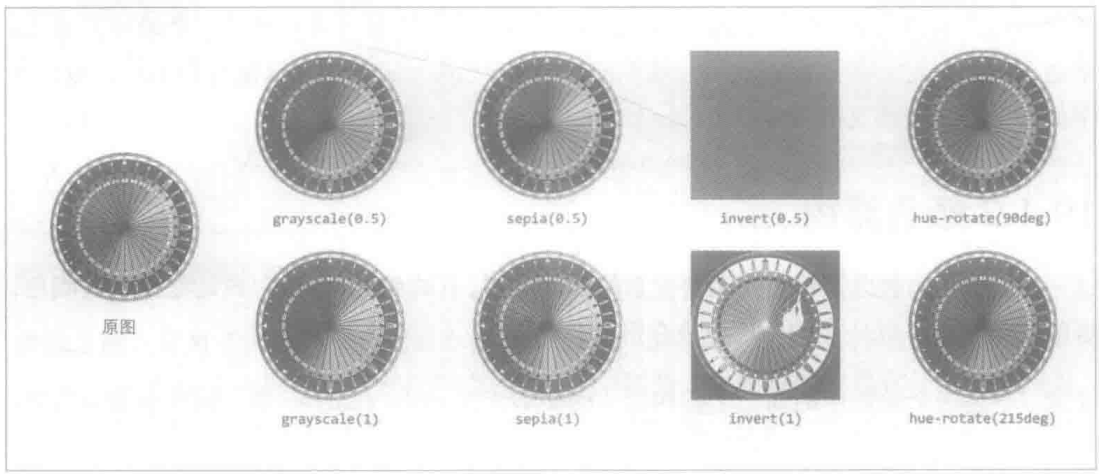


图 19-3：颜色滤镜的效果

19.1.3 亮度、对比度和饱和度

这些 filter 函数也用于操纵颜色，而且操纵的方式有很大的关联性，处理过图像，尤其是照片的人对这一组函数应该不会感到陌生。

注意，这些函数的值都可以大于 1 和 100%，不过计算最终值时会取整为 1 或 100%。

brightness([<*number*> | <*percentage*>])

调整元素上颜色的亮度。值为 0 时，元素为纯黑色；只为 1 或 100% 时，元素没有任

何变化。大于 1 和 100% 的值得到的颜色比元素原有的颜色更亮，最终有可能变成纯白色。

`contrast( [ <number> | <percentage> ] )`

调整元素上颜色的对比度。对比度越高，越容易区分颜色；对比度越小，颜色越接近。值为 0 时，元素变成纯灰色；值为 1 或 100% 时，元素没有任何变化。大于 1 和 100% 的值得到的颜色比元素原有的对比度大。

`saturate( [ <number> | <percentage> ] )`

调整元素上颜色的饱和度。饱和度越高，颜色越鲜艳；饱和度越低，颜色越黯淡。值为 0 时，元素完全没有饱和度，得到灰度效果；只为 1 或 100% 时，元素没有任何变化。与前面的函数不同，`saturate()` 允许使用也能应用大于 1 或 100% 的值，得到的结果是过度饱和。

图 19-4 给出了上述几个 `filter` 函数的示例，不过在彩色媒体上才能看到完整的效果。此外，值大于 1 时的效果很难在图中表现出来，不过图中注明了。

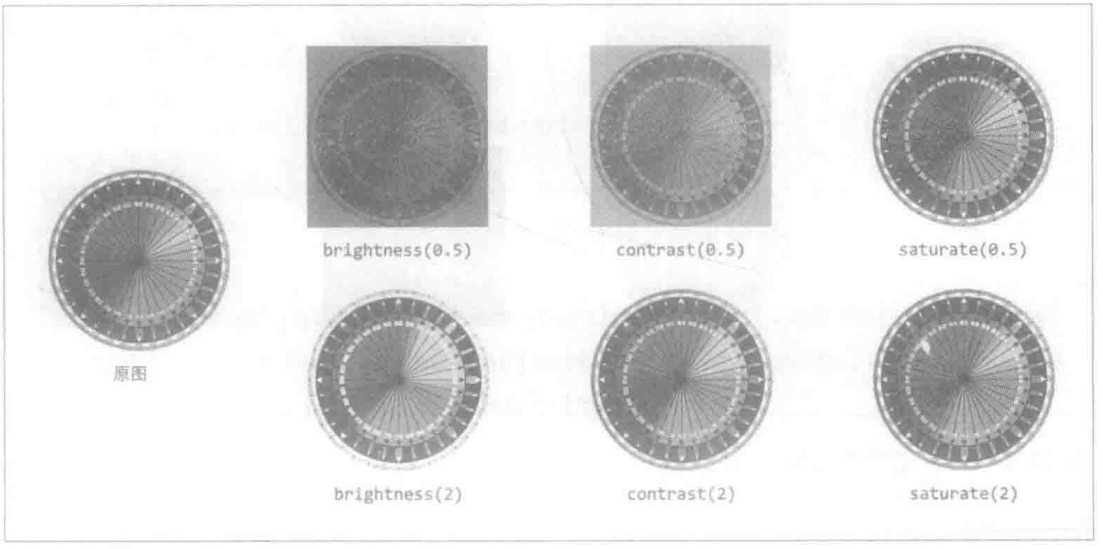


图 19-4：亮度、对比度和饱和度滤镜的效果

19.1.4 SVG 滤镜

`filter` 可以取的最后一个值是一个我们比较熟悉的函数：`url()`。创作人员可以使用这个函数指向 SVG 中定义的滤镜（一般来说比较复杂），这个 SVG 可以嵌入文档，也可以存储在外部文件中。

这个函数的格式是 `url(<uri>)`，其中 *<uri>* 指向一个使用 SVG 句法（具体而言是 `<filter>` 元素）定义的滤镜。引用的目标可以是只包含一个滤镜的 SVG 图像，例如 `url(wavy.svg)`，也可以是 SVG 图像中某个标识符对应的滤镜，例如 `url(filters.svg#wavy)`。后一种方式的优点是，可以在一个 SVG 中定义多个滤镜，这样便于加载、缓存和引用。

如果 `url()` 函数指向不存在的文件，或者指向的 SVG 片段不是 `<filter>` 元素，那么这个函数不起作用，而且整个函数列表都将被忽略（也就是整个 `filter` 声明都无效）。

全面说明如何使用 SVG 实现滤镜超出了本书范畴，你只需知道，SVG 提供的滤镜功能十分丰富。图 19-5 给出了一些简单的 SVG 滤镜示例，而且简要说明了各滤镜的作用（应用这些滤镜的 CSS 是这样的：`filter:url(filters.svg#rough)`）。

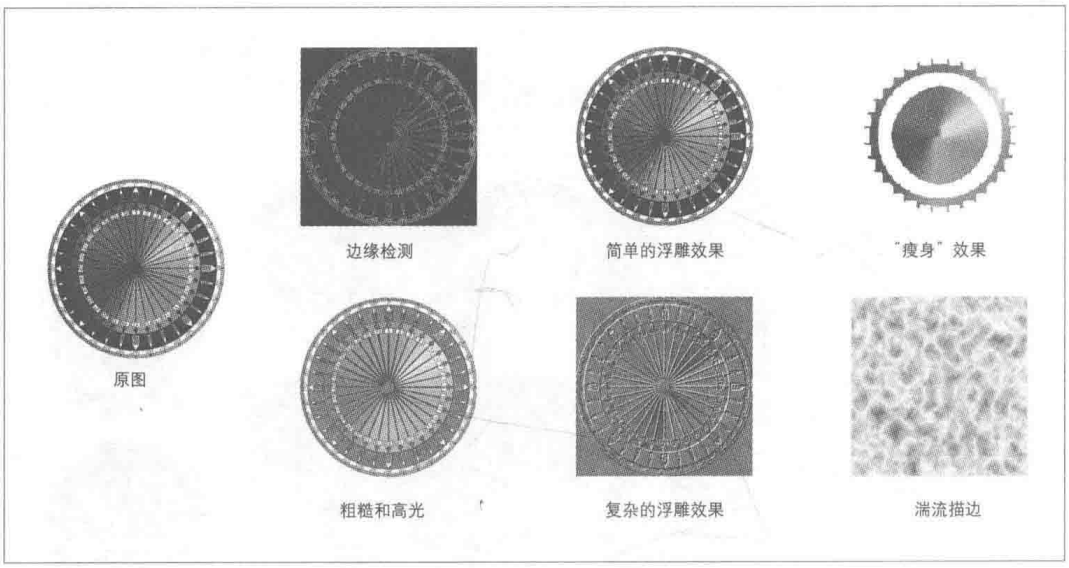


图 19-5：SVG 滤镜的效果

前文所讲的滤镜在 SVG 中都能轻易实现，包括 `filter` 可取的其他各个函数（其实，规范就是以 SVG 滤镜的形式给出其他各个 `filter` 函数的定义，以便给实现方一个参照）。记住，CSS 滤镜函数是可以串联的。因此，你可以使用 SVG 定义一个镜面高光滤镜，然后再使用模糊和灰度函数修饰。例如：

```
img.logo {filter: url(/assets/filters.svg#spotlight);}
img.logo.print {filter: url(/assets/filters.svg#spotlight) grayscale(100%);}
img.logo.censored {filter: url(/assets/filters.svg#spotlight) blur(3px);}
```

谨记，滤镜函数是按列出的顺序应用的。鉴于此，我们才把 grayscale() 和 blur() 函数放在通过 url() 导入的聚关灯滤镜后面。如果反过来，徽标将先变成灰度模式，然后做模糊处理，最后才应用高光效果。

19.2 合成和混合

除了滤镜，CSS 还提供有确定如何把元素合成在一起的功能。比如说，由于定位，两个元素可能出现部分重叠。根据目前所学的知识，我们知道前面的部分会遮盖后面的部分。这种处理方法有时称为简单的 alpha 合成，如果前面的元素某些部分（或者全部）有 alpha 通道，而且值小于 1，我们就能看到后面的元素。试想一下透过 opacity:0.5 的元素，或者 PNG 或 GIF87a 图像的透明部分能看到背景的情景。

熟悉图像编辑程序（例如 Photoshop 或 GIMP）的人可能知道，重叠的图层有多种混合模式。CSS 也提供了这样的功能。在 CSS 中，混合策略有两种（至少截至 2017 年年末如此）：整个元素与背后的内容（不管是什么）混合，以及把元素的多个背景层混合在一起。

19.2.1 混合元素

重叠元素的混合方式可以使用 mix-blend-mode 属性设定。

mix-blend-mode

取值	normal | multiply | screen | overlay | darken | lighten | color-dodge | color-burn | hard-light | soft-light | difference | exclusion | hue | saturation | color | luminosity
初始值	normal
适用于	所有元素
计算值	声明的值
继承性	否
动画性	否

CSS规范规定，“必须使用定义的公式把颜色和背着物(backdrop)混合在一起。”也就是说，不管元素背后是什么，是另一个元素也好，是父元素的背景也好，都必须混合。

默认值 normal 的意思是，元素上的各像素（除去 alpha 通道的值小于 1 的部分）原封不动地显示，不与背着物混合。这是前面所说的“简单的 alpha 合成”。我们平常使用的就是这种方式，因此用作默认值。图 19-6 中给出了一些示例。

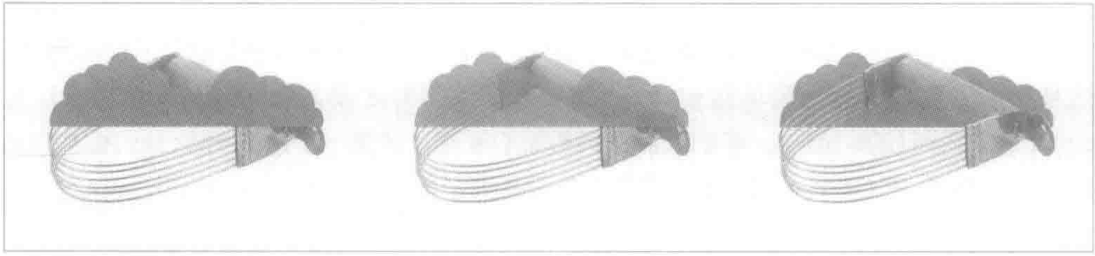


图 19-6：简单的 alpha 通道混合

笔者把 mix-blend-mode 的其他关键字值分成了几类，下面定义几个概念。

- 应用 mix-blend-mode 属性的元素是前景。
- 位于元素背后的是背着物。可以是另一个元素，也可以是父元素的背景等。
- 像素分量（pixelcomponent）是某个像素的颜色分量，即 R、G 和 B。

如果不理解，可以把前景和背着物想象为图象编辑程序中上下叠放的图层。mix-blend-mode 属性改变的是位于上层的图像（即前景）的混合模式。

19.2.2 变暗、变亮、差值和排除

这些混合模式或许可以称为简单的数学模式，因为混合的效果是直接以某种方式比较值，或者使用简单的加减法修改像素。

darken

前景中的每个像素与背着物中相应位置上的像素比较，比较 R、G 和 B 的值（像素分量），保留较小的值。因此，如果前景中某个像素的颜色为 rgb(91,164,22)，背着物中对应位置的颜色为 rgb(102,104,255)，得到的结果是 rgb(91,104,22)。

lighten

与 darken 相反，比较前景与背着物中相同位置上像素的 R、G 和 B 分量时，保留较

大的值。因此，如果前景中某个像素的颜色为 rgb(91,164,22)，背着物中对应位置的颜色为 rgb(102,104,255)，得到的结果是 rgb(102,164,255)。

difference

比较前景和背着物中同一位置上像素的 R、G 和 B 分量时，取两者之差的绝对值。因此，如果前景中某个像素的颜色为 rgb(91,164,22)，背着物中对应位置的颜色为 rgb(102,104,255)，得到的结果是 rgb(11,60,233)。如果其中一个像素是白色的，那么得到的像素的颜色是另一个像素的反相。如果其中一个像素是黑色的，那么得到的结果与另一个像素的颜色完全相同。

exclusion

这个模式是 difference 的温和版本。所用的公式不是 |back-fore|，而是 back+fore-(2 × back × fore)，其中 *back* 和 *fore* 是介于 0-1 之间的值。例如，橙色（rgb(100%,50%,0%)）和中灰色（rgb(50%,50%,50%)）运用排除模式得到的结果为 rgb(50%,50%,50%)。红色分量的算式为 1+0.5-(2 × 1 × 0.5)，减小到 0.5，即 50%。蓝色和绿色分量的算式为 0+0.5-(2 × 0 × 0.5)，也减小到 0.5。如果使用 difference，得到的结果为 rgb(50%,0%,50%)，因为各分量是两个值之差的绝对值。

从最后一个模式的定义可以看出，所有混合模式使用的其实都是介于 0-1 之间的值。前述示例中所用的值，例如 rgb(11,60,233)，是由 0-1 之间的值标化来的。也就是说，如果在 rgb(91,164,22) 和 rgb(102,104,255) 之间应用 difference 混合模式，真正的操作是下面这样的：

- 转换 rgb(91,164,22) 中的值：R=91 ÷ 255=0.357，G=164 ÷ 255=0.643，B=22 ÷ 255=0.086。以同样的方式转换 rgb(102,104,255) 中的值，结果为 R=0.4，G=0.408，B=1。
- 对应的分量相减，取结果的绝对值。因此，R=|0.357-0.4|=0.043，G=|0.643-0.408|=0.235，B=|1-0.086|=0.914。最终结果表示为 rgba(4.3%,23.5%,91.4%) 或 rgb(11,60,233)（每个分量乘以 255）。

此时你或许知道为什么介绍每个混合模式时没有给出完整的公式了。如果你对细节感兴趣，可以查阅“CompositingandBlendingLevel1”规范，里面有每个混合模式所用的公式。

图 19-7 给出了本节所讲的几个混合模式的示例。

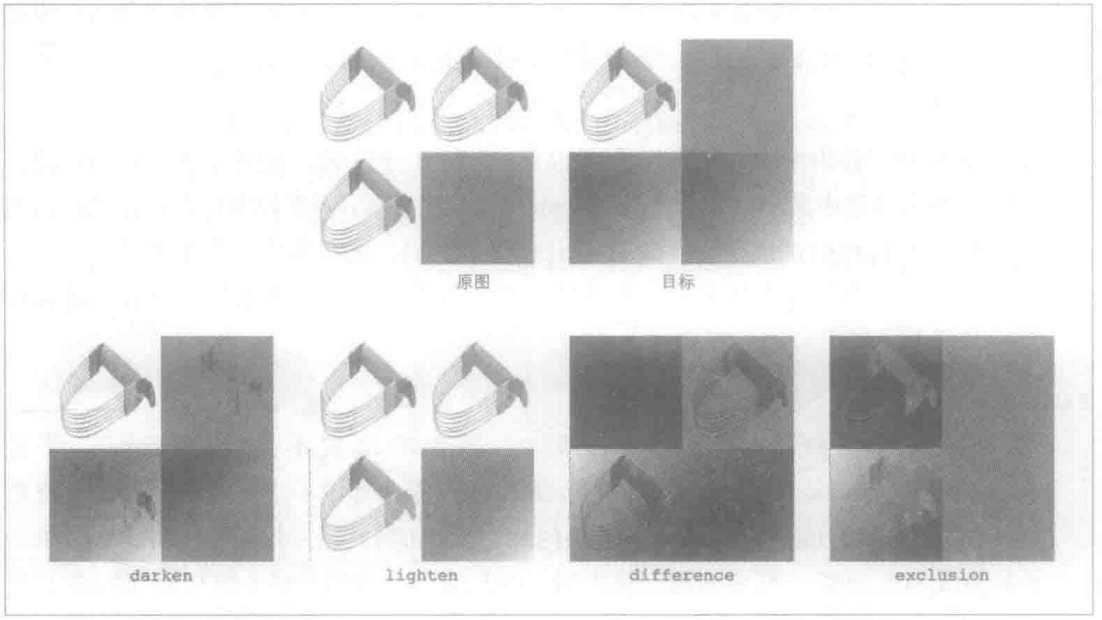


图 19-7：变暗、变亮、差值和排除混合模式

19.2.3 正片叠底、滤色和叠加

这几个混合模式或许可以称为乘法模式，因为效果是由值相乘得到的。

`multiply`

前景中的每个像素分量与背着物中对应位置上的像素分量相乘。得到的结果与前景相比更暗，这是由下面那一层修改的结果。这个混合模式是对称的，调换前景和背着物之后得到的结果完全一样。

`screen`

对前景和背着物中同一位置上的像素分量做反相处理（参见 19.1.2 节对 `invert` 的介绍），相乘后再反相。得到的结果与前景相比更亮，这是由下面那一层修改的结果。与 `multiply` 一样，`screen` 是对称的。

`overlay`

这个混合模式是对 `multiply` 和 `screen` 的综合应用。如果前景中的像素分量比 0.5（50%）暗，执行 `multiply` 操作；如果大于 0.5，则使用 `screen`。可见，较暗的区域变得更暗，而较亮的区域则变得更亮。这个混合模式不是对称的，因为调换前景和背着物之后，明暗度变了，应用正片叠底和滤色的方式也变了。

19-8 给出了这几个混合模式的示例。

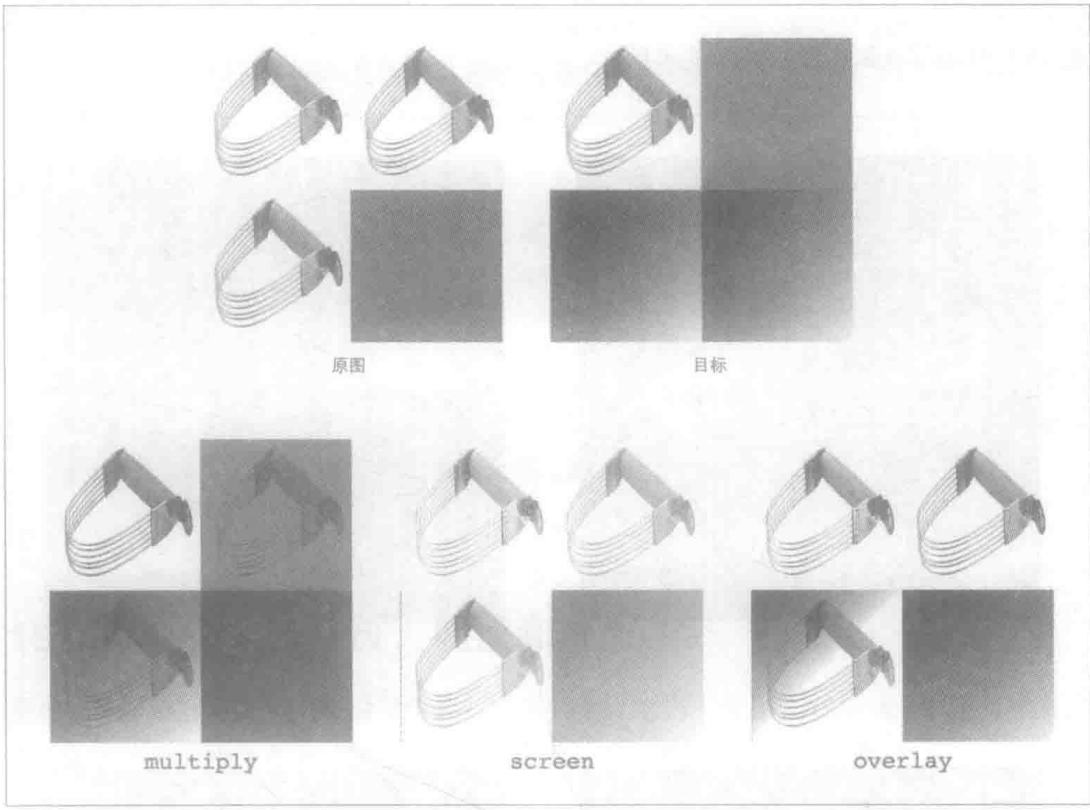


图 19-8：正片叠底、滤色和叠加混合模式

19.2.4 强光和柔光

现在讨论这两个混合模式是因为第一个与前面某个混合模式关系紧密，而第二个则是第一个的柔和版本。

`hard-light`

这个混合模式的作用与 `overlay` 相反。与 `overlay` 类似，这个混合模式也是对 `multiply` 和 `screen` 的综合应用，不过起决定作用的是背着物层。因此，背着物中比 0.5（50%）暗的像素分量使用 `multiply` 模式，而比 0.5 亮的则使用 `screen` 模式。这样得到的结果就像是投影仪使用刺眼的光线把前景投射到背着物上一样。

`soft-light`

这个混合模式是 `hard-light` 的柔和版本。也就是说，执行的操作是相同的，只不过

效果有所缓和。这样得到的结果就像是投影仪使用朦胧的光线把前景投射到背着物上一样。

图 19-9 给出了这两个混合模式的示例。

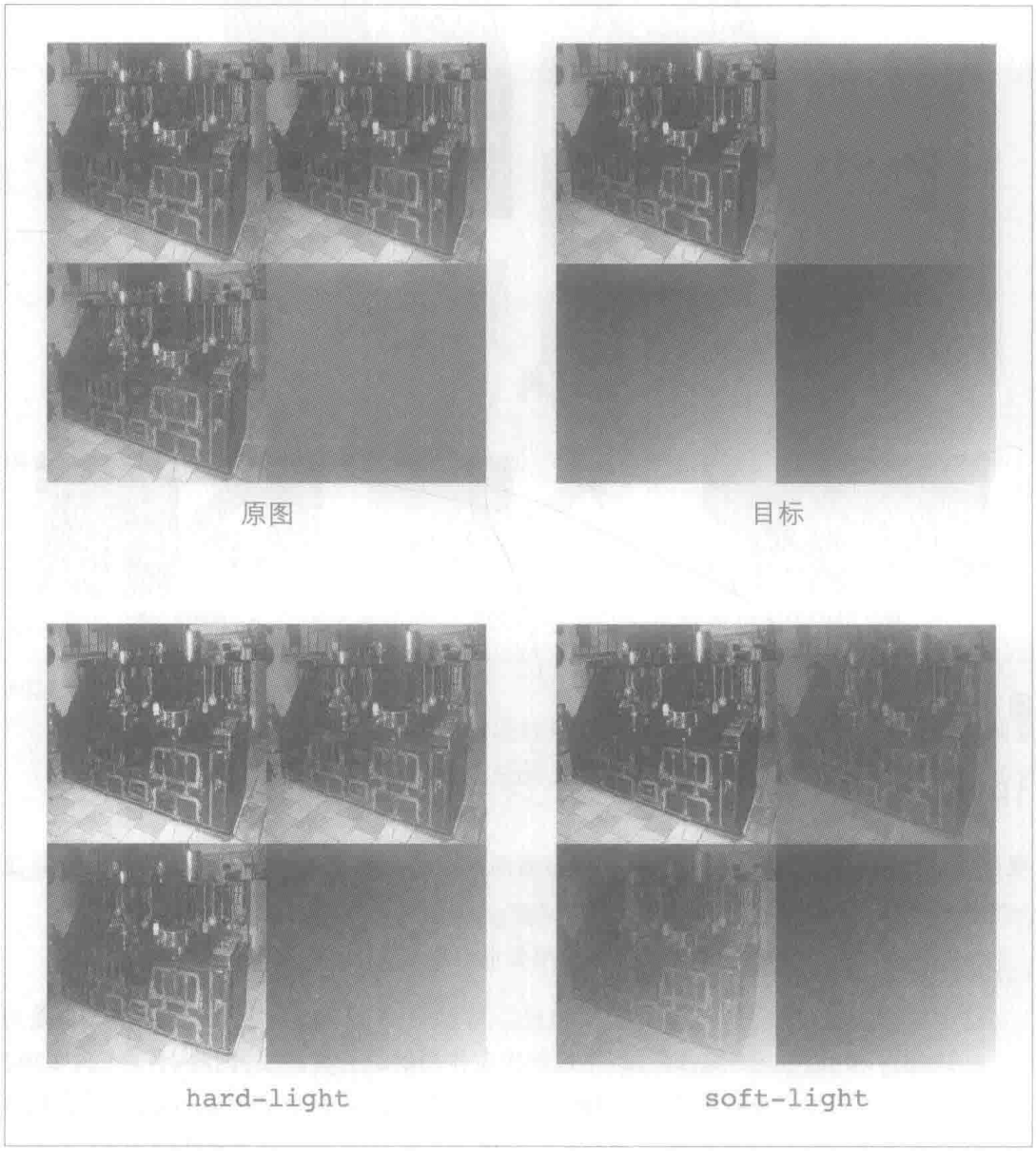


图 19-9：强光和柔光混合模式

19.2.5 颜色减淡和加深

颜色减淡（dodge）和加深（burn）的英文叫法很奇怪，其实二者的作用是在尽量不改变颜色的前提下把相片变亮和变暗。这种叫法源自以前在暗房中使用化学制剂对胶片所做的处理。

`color-dodge`

前景中的各像素分量做反相处理，再拿背着物中对应的像素分量除以反相后的前景值。得到的结果是背着物变得更亮，除非前景值为 0，此时背着物值没有任何变化。

`color-burn`

这个混合模式与 `color-dodge` 的作用相反。背着物中各像素分量做反相处理，反相后的背着物值除以没做任何修改的前景值，然后再反相。最终的结果是，背着物中越暗的像素，由前景中的像素加深的程度越大。

图 19-10 给出了这两个混合模式的示例。

19.2.6 色相、饱和度、明度和颜色

最后这四个混合模式与前面讲过的那些有所不同，它们不操作 R/G/B 像素分量，而是以不同的方式合并前景和背着物的色相、饱和度、明度和颜色。

`hue`

把背着物中每一像素的明度和饱和度与前景中对应像素的色相角度合并。

`saturation`

把背着物中每一像素的色相角度和明度与前景中对应像素的饱和度合并。

`color`

把背着物中每一像素的明度与前景中对应像素的色相角度和饱和度合并。

`luminosity`

把背着物中每一像素的色相角度和饱和度与前景中对应像素的明度合并。

不看原始公式很难理解这几个混合模式，如果不熟悉饱和度和明度，甚至还可能产生混淆。如果你发现自己不太理解这几个混合模式，最好的方法是多实操，看看不同的图像与简单的色块应用这些混合模式的效果。

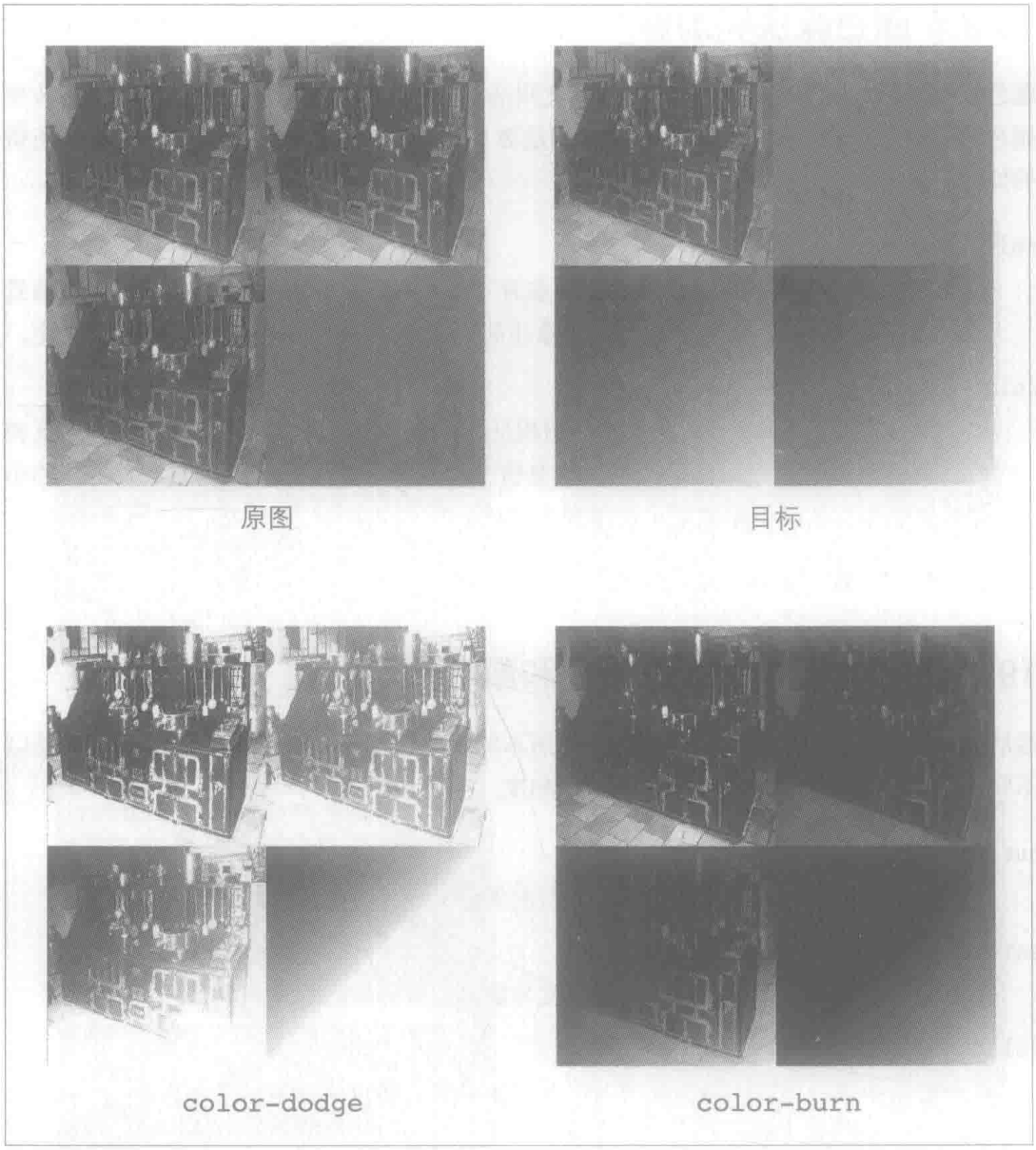


图 19-10：颜色减淡和加深混合模式

示例 19-11 给出了这些混合模式的示例。

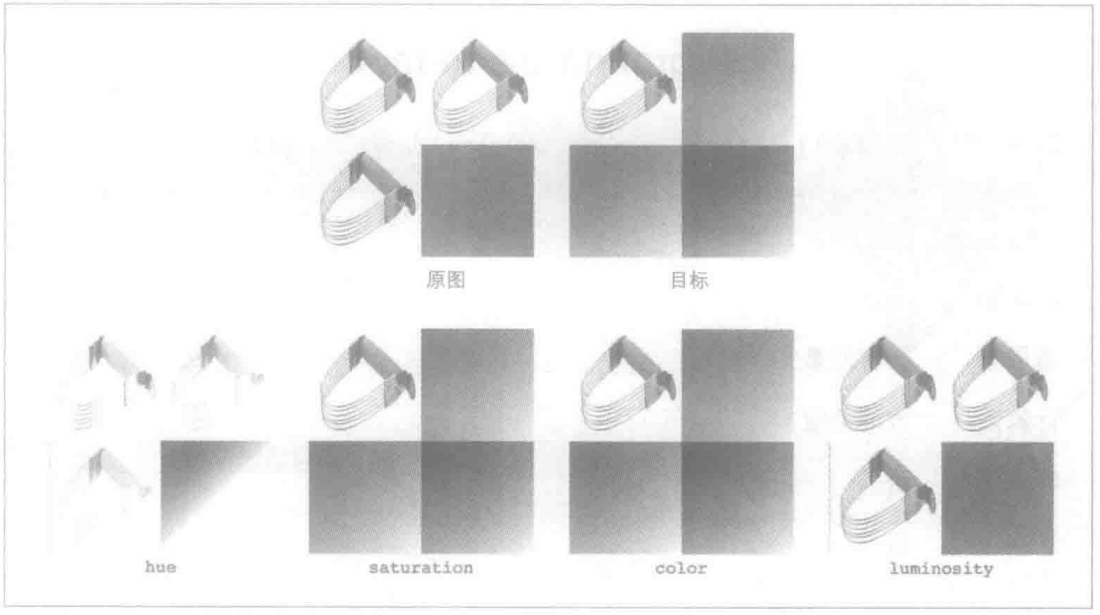


图 19-11：色相、饱和度、明度和颜色混合模式

注意两点：

- 记住，元素始终与背着物混合在一起。如果背后是其他元素，就与背后的元素混合；如果背后是父元素的背景，则与父元素的背景混合。
- 改变混合后元素的不透明度将对结果产生影响，但是影响的方式可能与你想的不一样。例如，应用 mix-blend-mode:difference 时如果还声明了 opacity: 0.8，那么差值计算的结果将缩小为原值的 80%。也就是，颜色值的计算结果要乘以 0.8。这会导致某些操作变成纯中灰色，而另一些操作会完全变一个颜色。

19.3 与背景混合

前面讲的是元素与背着物混合，如果一个元素有多个背景图，而且背景图有重叠，此时应该如何混合呢？这就要使用 background-blend-mode 属性了。

background-blend-mode

取值	[normal | multiply | screen | overlay | darken | lighten | color-dodge | color-burn | hard-light | soft-light | difference | exclusion | hue | saturation | color | luminosity]#
初始值	normal
适用于	所有元素
计算值	声明的值
继承性	否
动画性	否

本节不再逐一说明各混合模式的作用，前一节讨论 mix-blend-mode 属性时都讲过了。对这两个属性来说，各混合模式的作用是相同的。

区别在于，把多个背景图混合在一起时，各背景图是与一个空的背景在合并，即一个完全透明、没有颜色的背着物。如未声明 mix-blend-mode，背景图不与元素的背着物混合。

以下述代码为例：

```
#example {background-image:
        url(star.svg),
        url(diamond.png),
        linear-gradient(135deg, #F00, #AEA);
    background-blend-mode: color-burn, luminosity, darken;}
```

这里，我们声明了三个背景图，而且混合模式各不相同。这三个背景图将混合在一起，得到一个结果，如图 19-12 所示。

目前一切顺利，但是让人意想不到的是，不管元素背后有什么，结果都是一样的。我们可以把父元素的背景改成白色、灰色、紫红色，或者是重复的渐变组成的优美图案，不过这三个背景图混合后得到的结果都是一模一样的，一个像素也不差。背景图是独立混合的，稍后我们将讨论这个话题。把上述示例（见图 19-12）放在不同的背景之上得到的结果如图 19-13 所示。

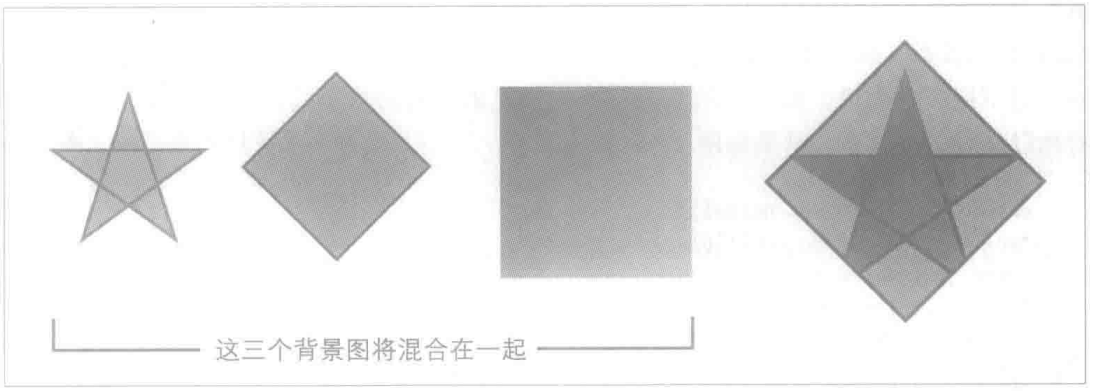


图 19-12：三个背景图混合在一起

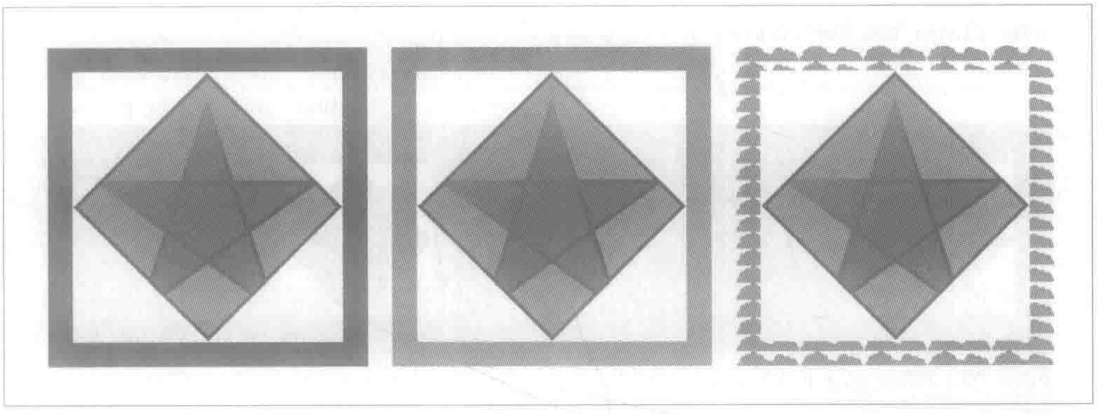


图 19-13：在有颜色的背景和透明的背景上混合的效果

与混合叠放在一起的多个元素一样，混合多个背景时，也是从后向前进行的。因此，如果两个背景图下面还有纯色背景，那么位于下层的背景图先与背景色混合，然后最上面的背景图再与二者混合的结果混合。以下述代码为例：

```
#example {background-image:
        url(star.svg),
        url(diamond.png);
    background-color: goldenrod;
    background-mix-mode: color-burn, luminosity;}
```

首先，diamond.png 与背景色 goldenrod 使用 luminosity 模式混合。然后，star.svg 再与得到的结果使用 color-burn 模式混合。

虽然背景层是独立混合的，但是背景是元素的一部分，而元素自身可能通过 mix-blend-mode 属性设置了混合规则。因此，独立混合背景得到的结果可能还会与元素的背着物混合。对下述样式来说，第一个示例的背景位于元素背着物的上层，而余下的示例将与背着物以某种方式混合，结果如图 19-14 所示。

```
.one {mix-blend-mode: normal;}
.two {mix-blend-mode: multiply;}
.three {mix-blend-mode: darken;}
.four {mix-blend-mode: luminosity;}
.five {mix-blend-mode: color-dodge;}
```

```
<div class="bbm one"></div>
<div class="bbm two"></div>
<div class="bbm three"></div>
<div class="bbm four"></div>
<div class="bbm five"></div>
```

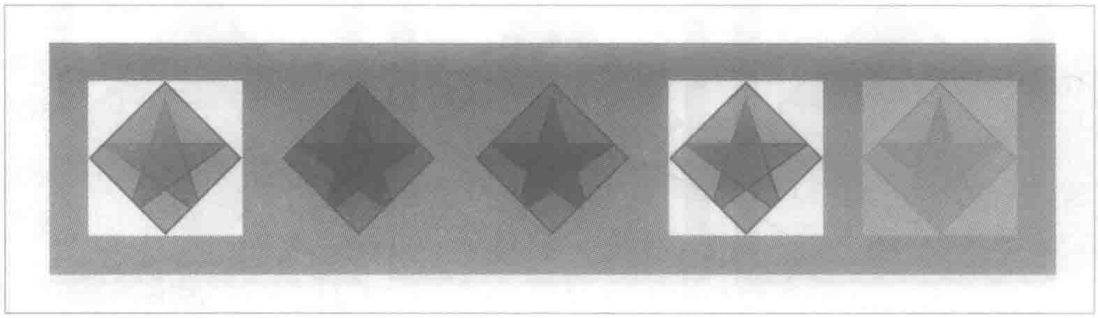


图 19-14：元素与背着物混合

本节不断讲到独立混合这个概念，这是混合背景的默认方式。但是，默认情况下，元素并不采用独立混合方式。不过，下一节将讲到，这个行为是可以修改的。

独立混合

有时，我们想把几个元素混合在一起，但是想让它们自成一组，像元素的背景层那样混合。这指的就是前面提到过的独立混合。如果想这么做，要使用 isolation 属性。

这个属性的作用从名称上就能看出，即指明是否为元素定义独立的混合上下文。对下述样式来说，得到的结果如图 19-15 所示。

isolation

取值	auto | isolate
初始值	auto
适用于	所有元素（在 SVG 中，适用于容器元素、图形元素和图形引用元素）
计算值	声明的值
继承性	否
动画性	否

```
img {mix-blend-mode: difference;}
p.alone {isolation: isolate;}
```

```
<p class="alone"><img src="diamond.png"></p>
<p><img src="diamond.png"></p>
```

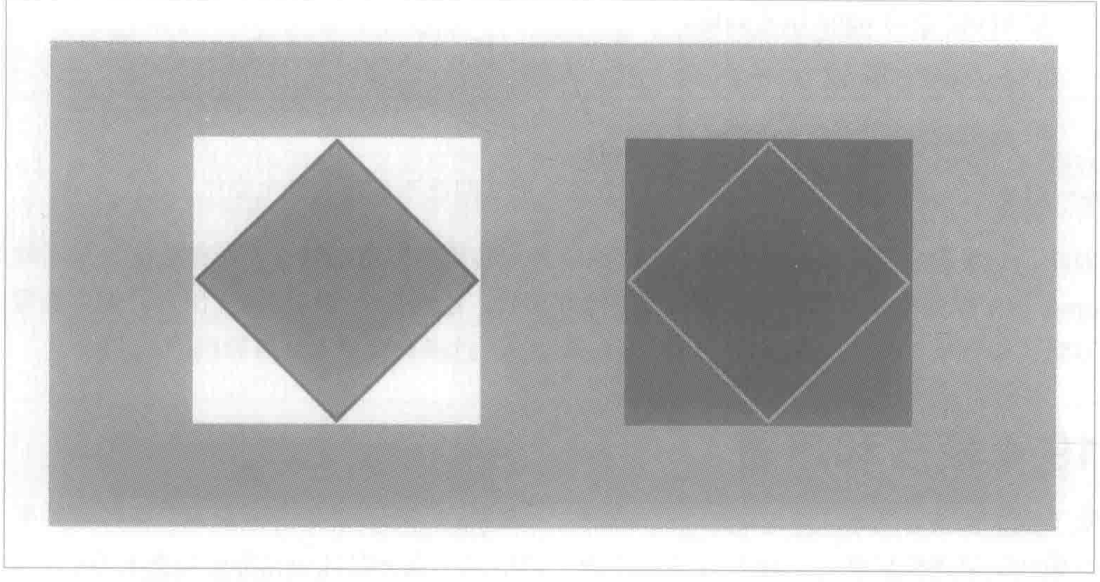


图 19-15：独立混合与不独立混合

特别注意isolation和mix-blend-mode是应用在哪个元素上的。混合模式在图像上设定，而是否独立混合在容器元素（这里指的是一个段落）上设置。这样做是为了把父元素（某些时候是祖辈元素）从文档中独立出来，从而影响后代元素的混合方式。因此，如果想独立混合某个元素，要找声明了isolation:isolate的祖辈元素。

但是有个例外，建立堆叠上下文的元素自动独立出来，而不管isolation属性为什么值。例如，如果使用transform属性改变一个元素的形态，那么它将独立出来。

截至2017年年末，建立堆叠上下文的全部情形如下所示：

- 根元素（例如<html>）。
- 相对或绝对定位一个元素，并把z-index设为auto以外的值。
- 固定定位一个元素，z-index设为什么值都行。
- 把opacity设为1以外的值。
- 把transform设为none以外的值。
- 把mix-blend-mode设为normal以外的值。
- 把filter设为none以外的值。
- 把perspective设为none以外的值。
- 把isolation设为isolate。
- 把will-change应用到上述任何一个属性上，即使不会真的发生变化。

因此，如果有一组元素想混合在一起，然后再与共同的背着物混合，而你把这组元素的opacity属性由1过渡到0了，那么在过渡期间，这组元素将突然独立出来。你可能看不到实际效果，这取决于原先使用的混合模式，不过多数情况下是有影响的。

19.4 裁剪和遮罩

除了滤镜和混合之外，使用CSS还能裁剪和遮罩。这两个功能的作用是只显示元素的部分区域，可见的区域可以使用各种简单的形状指定，也可以使用完整的图像和SVG元素指定。利用这两个功能可以对布局做些装饰，丰富视觉效果，例如给图像添加画框或锯齿状的边线。

19.4.1 裁剪

我们知道，使用 `filter` 属性能通过 SVG 应用裁剪路径。这是滤镜的合理用途，不过如果只想把元素的一部分剪掉，可以使用 `clip-path` 属性。

clip-path

取值	none | *<url>* | [[inset() | circle() | ellipse() | polygon()] || [border-box | padding-box | content-box | margin-box | fill-box | stroke-box | view-box]]
初始值	none
适用于	所有元素（在 SVG 中适用于所有图形元素和除 `<defs>` 之外的所有容器元素）
计算值	声明的值
继承性	否
动画性	inset()、circle()、ellipse() 和 polygon() 支持

`clip-path` 属性的作用是定义裁剪形状。元素的可见部分就在这个形状中绘制，而在这个形状外部的部分则被剪掉，留下透明的空白区域。下述代码对一个段落做了裁剪，并给出了未裁剪的版本，结果如图 19-16 所示。

```
p {background: orange; color: black; padding: 0.75em;}
p.clipped {clip-path: url(shapes.svg#cloud02);}
```

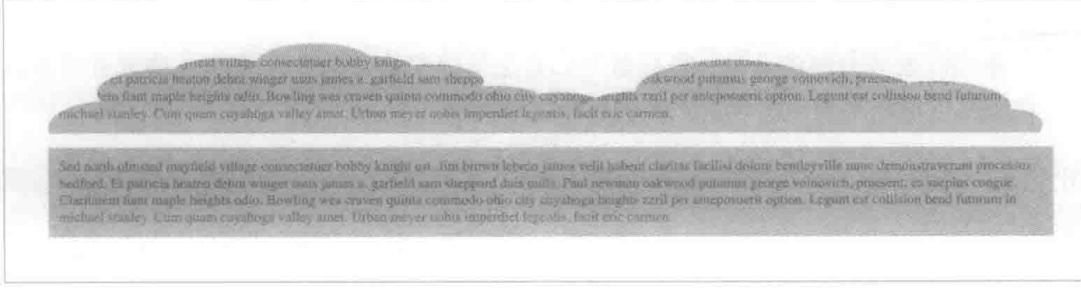


图 19-16：裁剪前后的段落

默认值 none 的意思是不做裁剪，这应该与你所想的一样。如果提供的是 *<uri>* 值（如前例），但是指向的资源不存在，或者没有指向 SVG 文件中的 <clipPath> 元素，那么不对元素做裁剪。

余下的值，要么是使用 CSS 绘制的形状，要么是引用框，抑或二者兼具。



截至 2017 年年末，如果 URL 指向的 SVG 嵌套在被裁剪的元素所在的文档中，只有 Chrome 支持这样的裁剪路径。所有浏览器都不支持外部 SVG 文件。

19.4.2 裁剪形状

裁剪路径可以使用四个简单的形状函数定义。这些函数的作用与 shape-outside 中用于浮动形状时一样（参见第 10 章），这里不再赘述。下面简单回顾一下。

inset()
: 值为一到四个长度值或百分数，定义距范围框各边的偏移量。可以使用 round 关键字，或者另一组一到四个长度值或百分数定义圆角。

circle()
: 只接受一个值，为长度、百分数或关键字，定义圆的半径。在一或两个长度或百分数后可以使用 at 关键字指定圆心的位置。

ellipse()
: 值必须为两个长度、百分数或关键字，定义椭圆在纵轴和横轴上的半径。在一或两个长度或百分数后可以使用 at 关键字指定椭圆中心的位置。

polygon()
: 一系列以逗号分隔的 *x* 和 *y* 坐标值（二者之间以空格分隔），可用长度或百分数。在坐标值前面可用关键字定义多边形的填充规则。

图 19-17 给出了这几个裁剪路径的示例，所用的样式如下（图中的虚线框表示裁剪前原图的外边界）。

```
.ex01 {clip-path: none;}
.ex02 {clip-path: inset(10px 0 25% 2em);}
.ex03 {clip-path: circle(100px at 50% 50%);}
```

```
.ex04 {clip-path: ellipse(100px 50px at 75% 25%);}
.ex05 {clip-path: polygon(50% 0, 100% 50%, 50% 100%, 0 50%);}
.ex06 {clip-path: polygon(0 0, 50px 100px, 150px 5px, 300px 150px, 0 100%);}
```

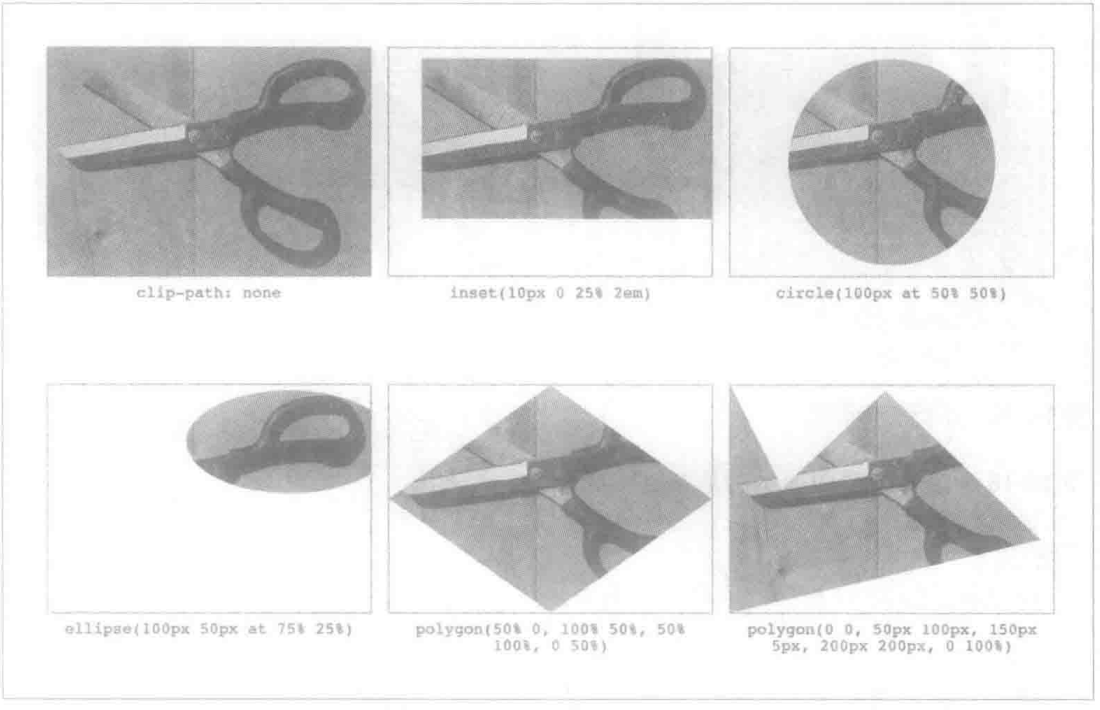


图 19-17：不同的裁剪形状

从图 19-17 可以看出，只有位于裁剪形状内部的元素区域才可见。形状外部的区域消失不见了。不过请注意，裁剪后元素所占的空间不变，好似根本没有裁剪一样。也就是说，裁剪并不会导致元素的尺寸变小。裁剪只是限制具体绘制元素的哪些部分。

19.4.3 裁剪框

与裁剪形状不同，裁剪框不使用长度或百分数指定。多数时候，裁剪框直接对应于盒模型中的边界。

比如说，`clip-path:border-box` 声明的意思是沿边框的外边界裁剪元素。我们通常见到的就是这种效果，因为外边距是透明的。不过要注意，轮廓在边框的外侧绘制，因此如果沿边框的边界裁剪，轮廓会被剪掉。

单独使用时，`margin-box`、`padding-box` 和 `content-box` 分别在外边距、内边距和内容区域的外边界裁剪元素，如图 19-18 所示。

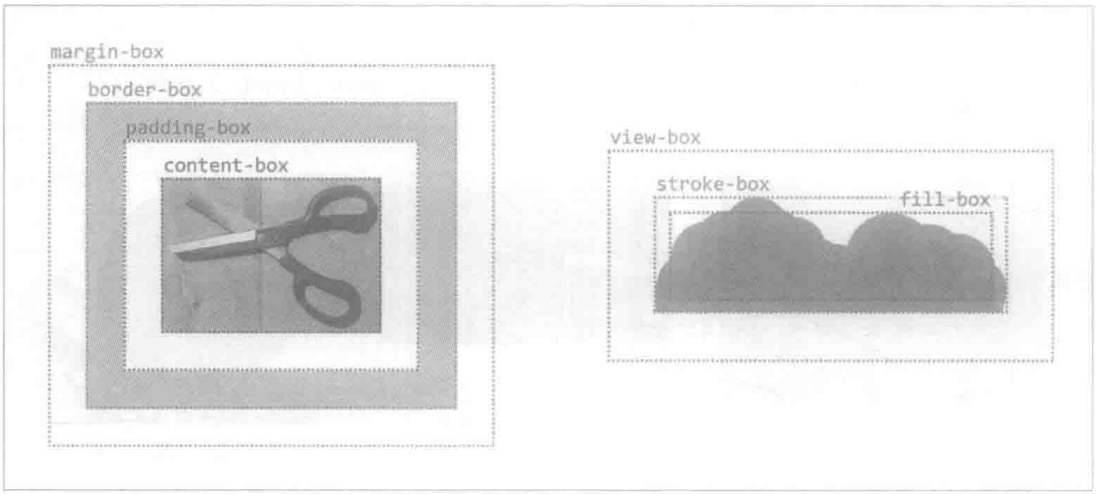


图 19-18：不同的裁剪框

图 19-18 中的另一幅图展示的是 SVG 范围框。

`view-box`

使用最近的 SVG 视区（即最近的祖辈）作为裁剪框。

`fill-box`

使用对象范围框作为裁剪框。对象范围框指能放下元素几何尺寸范围内所有部分的最小框体，包括变形（例如旋转），但不包括外围的描边。

`stroke-box`

使用描边范围框作为裁剪框。描边范围框指能放下元素几何尺寸范围内所有部分的最小框体，包括变形（例如旋转），也包括外围的描边。

这几个值只适用于没有相应 CSS 布局框的 SVG 元素。对这样的元素来说，如果提供了 CSS 样式框（`margin-box`、`border-box`、`padding-box`、`content-box`），真正使用的是 fill-box。反过来，如果把 SVG 范围框值应用到有 CSS 布局框的元素（多数元素都是如此）上，真正使用的是 `border-box`。

有时可以只使用裁剪框，例如声明 `clip-path:content-box`，把内容区以外的部分都剪掉。但是与裁剪形状结合在一起使用更能体现裁剪框的作用。假如我们想把一个 `ellipse()` 裁剪形状应用到元素上，而且只想保留内边距框外边界以里的部分。此时，为了计算椭圆的长短轴，无需从元素的整体尺寸上减去外边距和边框，只需声明

clip-path:ellipse(50%50%)padding-box;。这个声明把椭圆裁剪形状放在元素的中心，横轴和纵轴半径分别为元素引用框的一半（参见第 10 章），而且能根据其他框体的尺寸做调整，如图 19-19 所示。

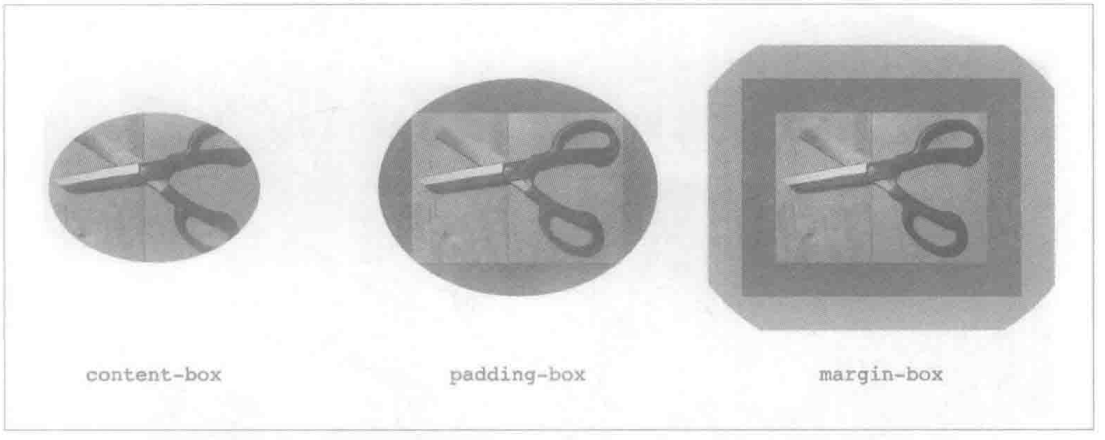


图 19-19：根据不同框体的尺寸调整椭圆裁剪形状

注意，在 margin-box 示例中，椭圆也被剪掉了一部分。这是因为外边距是不可见的，如果有一部分外边距落在椭圆裁剪形状中，那一部分也就看不到了。

有趣的是，范围框关键字只能与裁剪形状一起使用，而不能与基于 SVG 的裁剪路径一起使用。SVG 范围框相关的那几个关键字旨在通过 CSS 裁剪 SVG 图像时才可用。

关于 SVG 裁剪路径，有一点要注意：截至 2017 年年末，路径的所有坐标只能使用绝对单位，而不能声明为图像高度和宽度的百分之几，但是 polygon() 形状可以。有些技术使用 SVG 的 clipPathUnits 属性，有时要 SVG 的 transform 属性配合，可以得到相同的结果。下面举个这样的例子，结果如图 19-20 所示。

```
<clipPath id="hexlike" clipPathUnits="objectBoundingBox">
    <polygon points="0.5 0, 1 0.25, 1 0.75, 0.5 1, 0 0.75, 0 0.25"/>
</clipPath>
```

设为 objectBoundingBox，坐标的值将根据所用的范围框而调整。注意，坐标的值都在 0-1 之间。这样绘制出来的裁剪路径与使用百分数定义的 polygon 形状相比，作用是相同的。使用下述声明也能得到与图 19-20 中一样的裁剪形状：

```
clip-path: polygon(50% 0, 100% 25%, 100% 75%, 50% 100%, 0 75%, 0 25%);
```

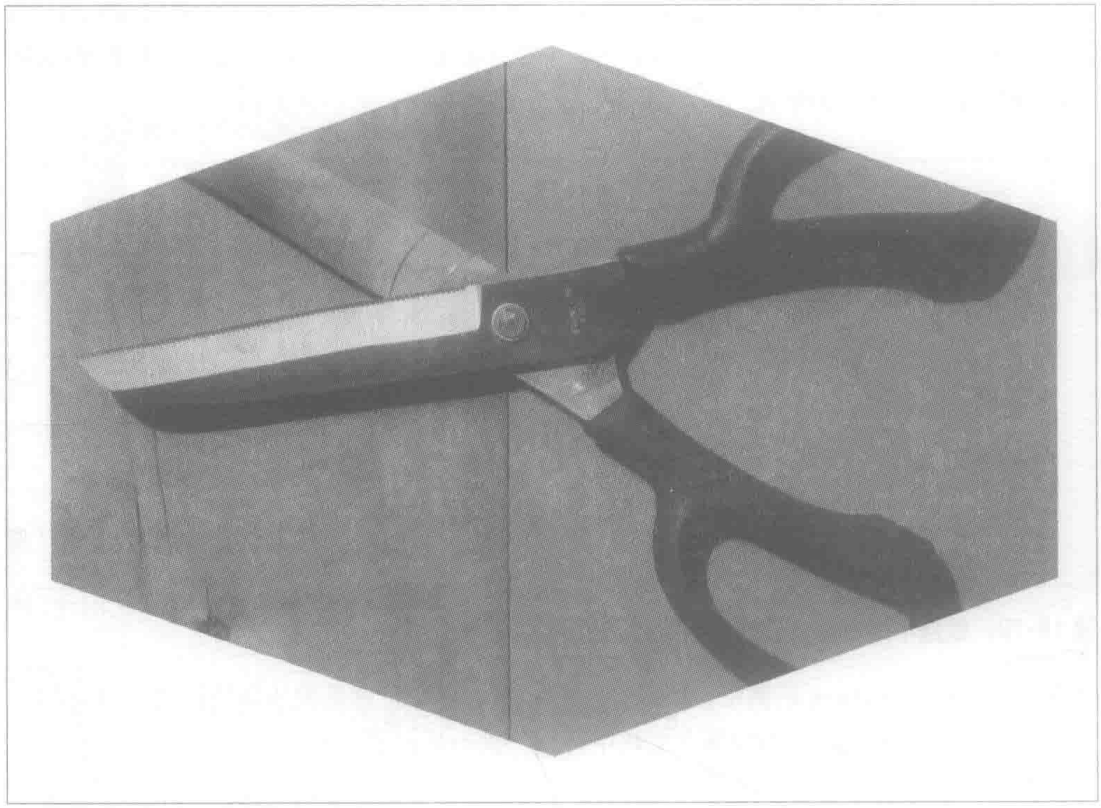


图 19-20：使用可缩放的 SVG 裁剪路径裁剪图像

19.4.4 裁剪填充规则

与浮动形状一样，SVG 形状的填充方式是可以改变的，即路径交汇时到底创建什么样的裁剪形状。这个行为由 clip-rule 属性控制。

nonzero 和 evenodd 两种填充方式比较抽象，通过示例更容易理解，如图 19-21 所示。

可以看出，图中的五角星是从顶部中间的点开始，用直线把后续各点连接在一起而绘制出来的。nonzero 方式填充五角星内部的所有区域，即便是线条交汇的区域。而 evenodd 留出一部分不填充，所以我们才能透过中间部分看到淡蓝色的渐变。

可问题是，截至 2017 年年末，即便是支持 SVG 裁剪路径的浏览器也不支持这个属性，无论 SVG 是嵌套在 HTML 中还是存在外部文件里。因此，如果想把裁剪路径的形状填充方式设为 evenodd，要么使用 CSS polygon 重现 SVG 路径，要么在 SVG 文件中使用 SVG 的 fill-rule 属性。

clip-rule

取值	nonzero | evenodd
初始值	nonzero
适用于	SVG 中当且仅当为 <clipPath> 元素的子元素时的所有图形元素（<circle>、<ellipse>、<image>、<line>、<path>、<polygon>、<polyline>、<rect>、<text> 和 <use>）
计算值	声明的值
继承性	否
动画性	否

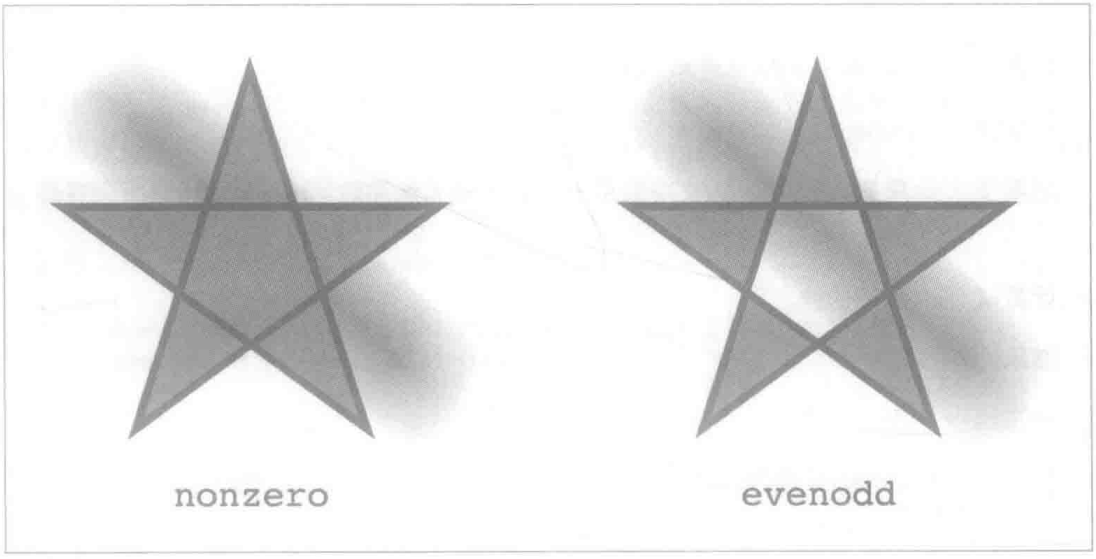


图 19-21：填充形状的两种方式

19.5 蒙版

“蒙版”的意思是，位于形状内部的内容可见，而在形状外部的内容则不可见。可见，在概念上，蒙版与裁剪路径非常像。二者之间的主要区别有两点，其一，使用蒙版时只

能用图像定义元素的哪些部分可见、哪些部分被剪掉；其二，控制蒙版的属性更多，可以定位、设置尺寸和重复蒙版图像的方式。



截至 2017 年年末，Blink 族浏览器支持多数蒙版属性，不过要加上 -webkit- 前缀。因此，在 Chrome 和 Safari 中，mask-image 要写做 -webkit-mask-image。

19.5.1 定义蒙版

应用蒙版的第一步是指定使用什么图像作为蒙版。这一步使用 mask-image 属性完成，其值可以是任何类型的图像。

mask-image

取值	[none | <*image*> | <*mask-source*>]#
初始值	none
适用于	所有元素（在 SVG 中适用于所有图形元素和除 <defs> 之外的所有容器元素）
计算值	声明的值
继承性	否
动画性	否
备注	<*image*> 是任何类型的 <*url*>、<*image()*>、<*image-set()*>、<*element()*>、<*cross-fade()*> 或 <*gradient*>（本书都有定义）；<*mask-source*> 是一个 url()，指向 SVG 图像中的 <mask> 元素

假设引用的是有效的图像，那么目标图像将用作元素的蒙版。

先看简单的情况：把一张图像应用到另一个图像上，而且两张图像的高度和宽度相同。请看图 19-22，图中分别给出了两张图像，然后使用第二张图像遮罩第一张图像。

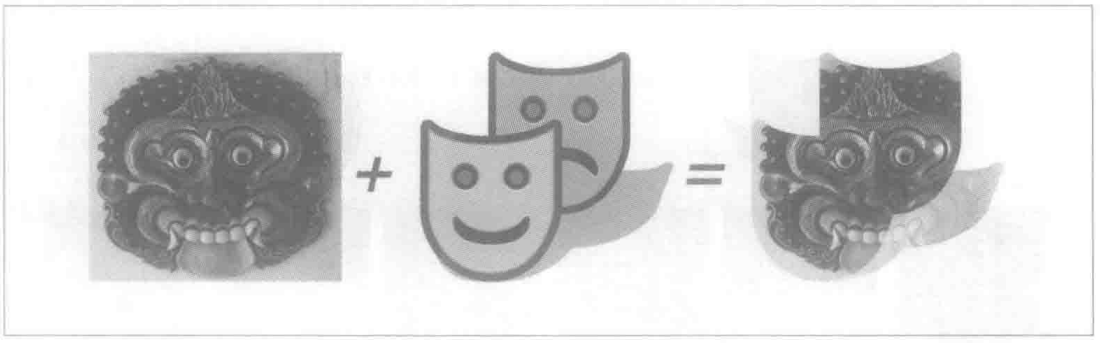


图 19-22：简单的图像蒙版

可以看到，在第二张图像上不透明的地方，可以看到第一张图像的内容。而在透明的部分，看不到第一张图像的内容。在半透明的地方，看到的第一张图像的内容也是半透明的。

图 19-22 所用的基本代码如下：

```
img.masked {mask-image: url(theatre-masks.svg);}
```

不过，CSS 并不限制只能把蒙版应用到图像上。使用图像基本上可以遮罩任何元素，而且所用的图像可以是光栅图像（GIF、JPG、PNG），也可以是矢量图像（SVG）。后者通常是更好的选择。此外，甚至还可以使用渐变自行构建图像，线性或径向，重复不重复都可以。

下述样式将得到如图 19-23 所示的结果。

```
*.masked.theatre {mask-image: url(theatre-masks.svg);}
*.masked.compass {mask-image: url(Compass.png);}
```

请牢记一点，如果蒙版会导致元素的部分区域被剪掉，那么所有多出的区域都会被剪掉。比如说，如果图像会把元素外边界以外的内容剪掉，很可能导致列表项目的记号不可见。请看图 19-24 中的示例，所用的代码如下：

```
*.masked {mask-image: url(i/Compass_masked.png);}
<ol class="masked">
    <li>One</li>
    <li>Two</li>
    <li>Three</li>
    <li>Four</li>
    <li>Five</li>
</ol>
```

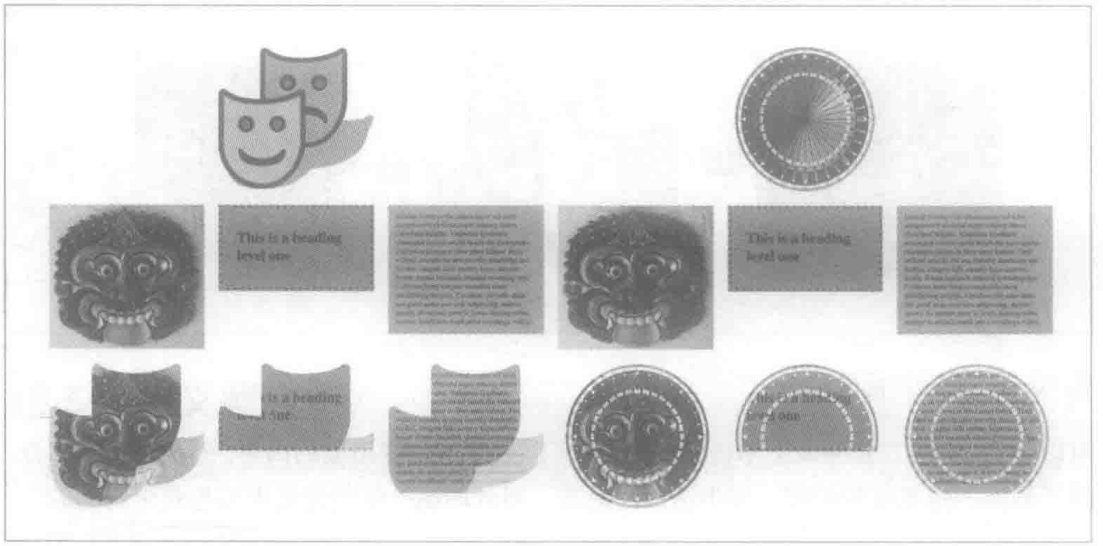


图 19-23：不同的图像蒙版

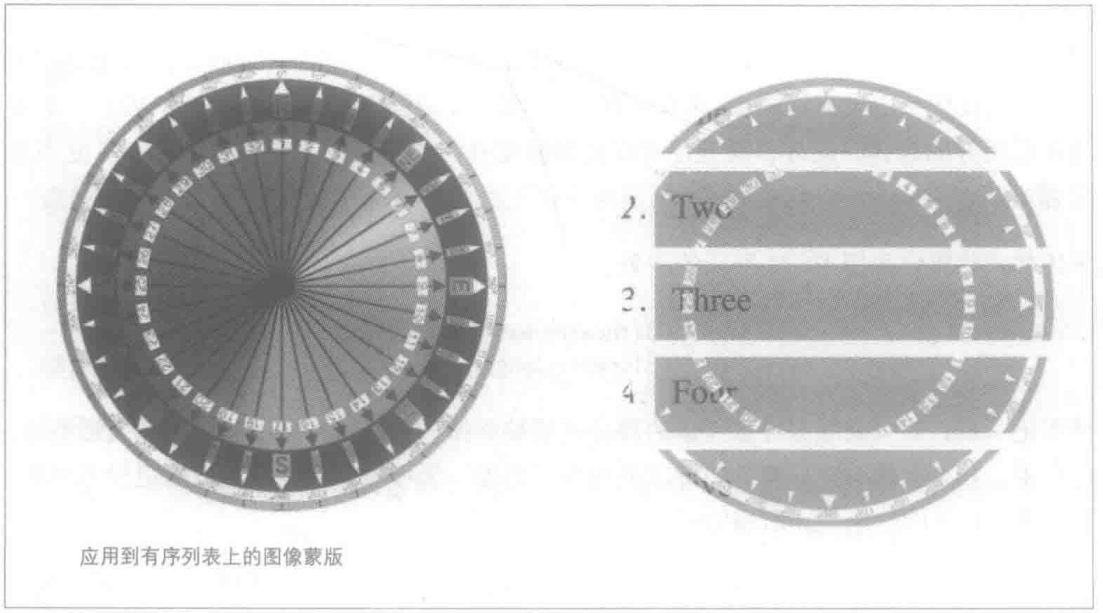


图 19-24：部分列表项目被遮罩，部分没被遮罩

还有一个值没有介绍，即直接指向 SVG 中的 `<mask>` 元素，使用那个元素定义的蒙版。这与 clip-path 属性指向 `<clipPath>` 或其他 SVG 元素（参见 19.4.1 节）是一样的。

下面举个例子说明如何定义蒙版：

```
<svg viewbox="0 0 100 100" height="100" width="100"
     xmlns="http://www.w3.org/2000/svg" version="1.1">
```

```
    <mask id="hexlike">
    <path fill="#FF0000"
          d="M 50,0 100,25 100,75 50,100 0,75 0,25" />
    </mask>
</svg>
```

如果把这个 SVG 直接嵌在 HTML 文件中，可以像下面这样引用这个蒙版：

```
.masked {mask-image: url(#hexlike);}
```

如果 SVG 保存在外部文件中，在 CSS 中要像下面这样引用：

```
.masked {mask-image: url(masks.svg#hexlike);}
```

19.5.2 改变蒙版的模式

目前所见的蒙版都是把一个有 alpha 通道的图像应用到另一个元素上。这只是使用图像蒙版的两种方式之一。另一种方式是使用图像各部分的亮度定义蒙版。这两种方式使用 mask-mode 属性切换。

mask-mode

取值	[alpha | luminance | match-source]#
初始值	match-source
适用于	所有元素（在 SVG 中适用于所有图形元素和除 <defs> 之外的所有容器元素）
计算值	声明的值
继承性	否
动画性	否

可取的值有三个，其中两个比较容易理解：alpha 的意思是使用图像的 alpha 通道计算蒙版，luminance 的意思是使用亮度计算蒙版。二者之间的区别如图 19-25 所示，使用的代码如下：

```
img.theatre {mask-image: url(i/theatre-masks.svg);}
img.compass {mask-image: url(i/Compass_masked.png);}
img.lum {mask-mode: luminance;}
```

```
<img src="i/theatre-masks.svg">
<img class="theatre" src="i/mask.JPG">
<img class="theatre lum" src="i/mask.JPG">
<img src="i/Compass_masked.png">
<img class="compass" src="i/mask.JPG">
<img class="compass lum" src="i/mask.JPG">
```

使用明度计算蒙版时，处理亮度的方式与 alpha 蒙版处理 alpha 值的方式一样。alpha 蒙版是这样得到的：图像上不透明度为零的部分把被遮罩的元素上的相应部分隐藏起来，不透明度为 1 的部分（即完全不透明）把被遮罩的元素上的相应部分显示出来。

基于明度的蒙版也以相同的方式得到。图像上明度为 1 的部分把被遮罩的元素上的相应部分显示出来，明度为零（即全黑）的部分把被遮盖的元素上的相应部分隐藏起来。注意，蒙版上完全透明的部分，其明度也视为零。正是因为这样，戏剧面罩图像的阴影区域才不显示被遮罩图像上的内容，因为其 alpha 值大于零。

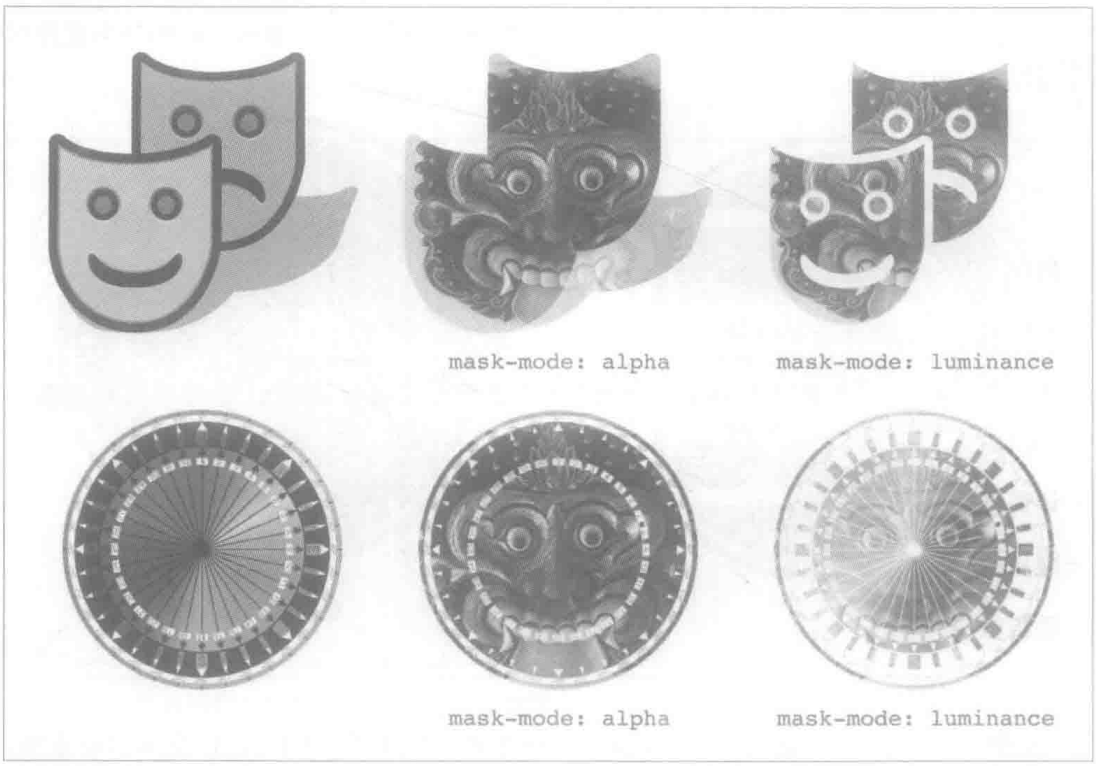


图 19-25：alpha 和明度两种蒙版模式

第三个值（默认值），即 `match-source`，是 `alpha` 和 `luminance` 的结合体，根据下述条件在用作蒙版的源图像上具体选用：

- 如果源图像是 *<image>* 类型，使用 alpha。*<image>* 可以是 PNG 或可见的 SVG 等图像，可以是 CSS 渐变，也可以是通过 element() 函数引用的页面上的一部分。
- 如果源图像是 SVG <mask> 元素，使用 luminance。

19.5.3 调整蒙版的尺寸和重复方式

目前为止所举的例子基本上都确保蒙版的尺寸与要遮罩的元素的尺寸保持一致（因此我们才只把蒙版应用到图像上）。其实，蒙版图像的尺寸可以与要遮罩的元素不一致。而调整的方式有好几种，先看 mask-size 属性。

mask-size

取值	[[*<length>* | *<percentage>* | auto]{1,2} | cover | contain]#
初始值	auto
适用于	所有元素（在 SVG 中适用于所有图形元素和除 <defs> 之外的所有容器元素）
计算值	声明的值
继承性	否
动画性	*<length>*，*<percentage>*

只要知道如何调整背景图的尺寸，也就知道具体怎么调整蒙版的尺寸，二者的取值句法一模一样，行为也完全相同。举个例子，下述样式将得到如图 19-26 所示的结果。

```
p {mask-image: url(i/hexlike.svg);}
p:nth-child(1) {mask-size: 100% 100%;}
p:nth-child(2) {mask-size: 50% 100%;}
p:nth-child(3) {mask-size: 2em 3em;}
p:nth-child(4) {mask-size: cover;}
p:nth-child(5) {mask-size: contain;}
p:nth-child(6) {mask-size: 200% 50%;}
```

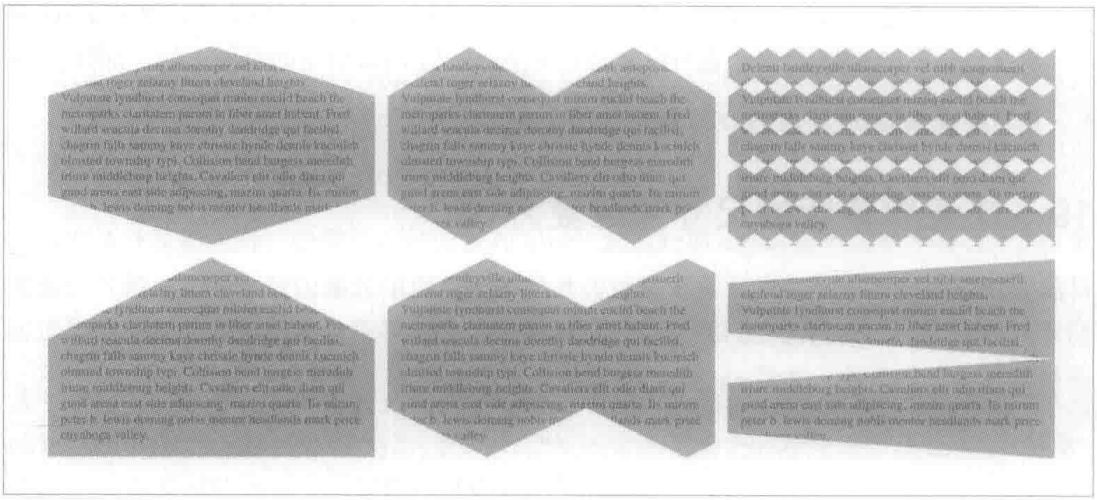


图 19-26：调整蒙版的尺寸

再次申明，只要知道如何调整背景的尺寸就不会感到陌生。如果不太理解，请翻回第 9 章的 9.2.8 节，那里有详细讨论。

我们知道背景在元素背景区域中的重复模式是可以调整或禁用的，同样，蒙版图像也可以做类似的调整，使用的属性是 mask-repeat。

mask-repeat

取值	[repeat-x | repeat-y | [repeat | space | round | no-repeat]{1,2}]#
初始值	repeat
适用于	所有元素（在 SVG 中适用于所有图形元素和除 <defs> 之外的所有容器元素）
计算值	声明的值
继承性	否
动画性	是
备注	mask-repeat 属性的关键字值与 background-repeat 属性一样，行为也相同

这个属性可用的值与 background-repeat 属性一样。图 19-27 给出了一些示例，所用的样式如下：

```
p {mask-image: url(i/theatre-masks.svg);}
p:nth-child(1) {mask-repeat: no-repeat; mask-size: 10% auto;}
p:nth-child(2) {mask-repeat: repeat-x; mask-size: 10% auto;}
p:nth-child(3) {mask-repeat: repeat-y; mask-size: 10% auto;}
p:nth-child(4) {mask-repeat: repeat; mask-size: 30% auto;}
p:nth-child(5) {mask-repeat: repeat round; mask-size: 30% auto;}
p:nth-child(6) {mask-repeat: space no-repeat; mask-size: 21% auto;}
```

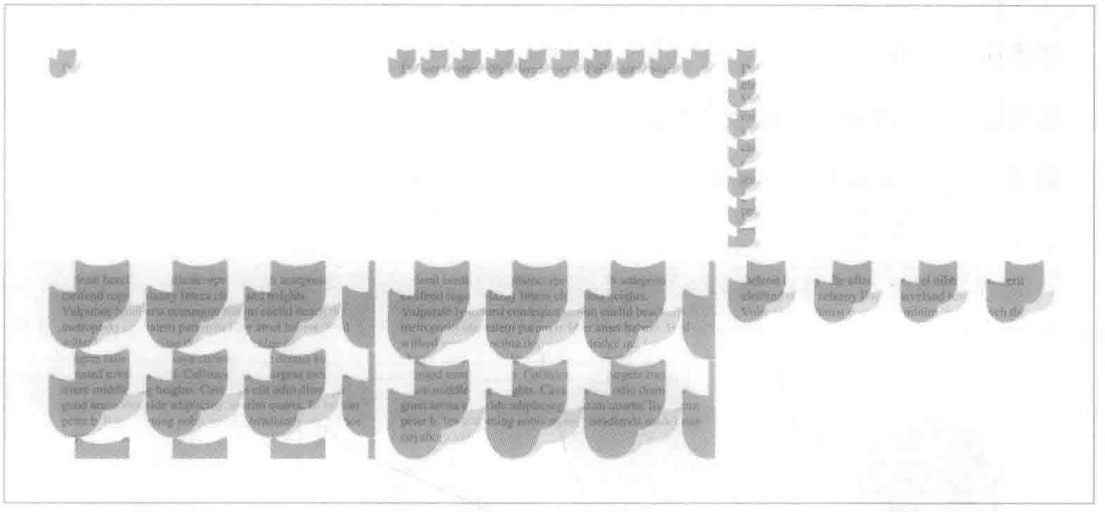


图 19-27：重复蒙版

19.5.4 定位蒙版

我们知道，调整蒙版图像尺寸和重复方式的方法与背景图一样，由此你可能会想，蒙版图像的定位方式应该与 background-position 属性类似，而且设定原点框的方式应该与 background-origin 属性类似。没错，你想的完全正确。

还是一样，如果你设置过背景图的位置，那就知道如何设置蒙版图像的位置。下面举几个例子，结果如图 19-28 所示（为了看的更清楚，加上了点线框）。

```
p {mask-image: url(i/Compass_masked.png);
    mask-repeat: no-repeat; mask-size: 67% auto;}
p:nth-child(1) {mask-position: center;}
p:nth-child(2) {mask-position: top right;}
p:nth-child(3) {mask-position: 33% 80%;}
p:nth-child(4) {mask-position: 5em 120%;}
```

mask-position

取值	*<position>#*
初始值	0% 0%
适用于	所有元素（在 SVG 中适用于所有图形元素和除 <defs> 之外的所有容器元素）
计算值	声明的值
继承性	否
动画性	*<length>*，*<percentage>*
备注	*<position>* 与 background-position 属性允许取的值完全一样，行为也相同

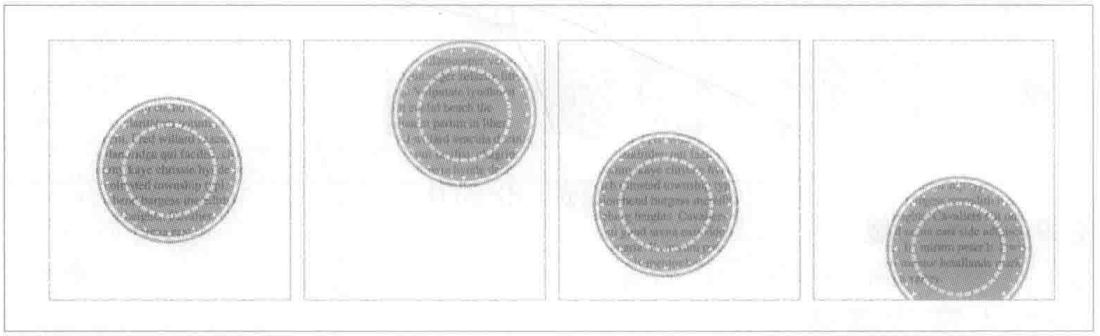


图 19-28：定位蒙版

默认情况下，蒙版图像的原点框是边框的外边界。如果想向内移动，或者在 SVG 上下文中定义原点框，使用 mask-origin 属性。这个属性的作用与设置背景的 background-origin 属性一样。

mask-origin

取值	[content-box | padding-box | border-box | margin-box | fill-box | stroke-box | view-box]#
初始值	border-box
适用于	所有元素（在 SVG 中适用于所有图形元素和除 <defs> 之外的所有容器元素）
计算值	声明的值
继承性	否
动画性	否

对背景来说，这是个较新的功能，你可能不太熟悉。完整的说明参见第 9 章的 9.2.5 节。下面举几个简单的例子，如图 19-29 所示。



图 19-29：修改原点框

19.5.5 裁剪和合成蒙版

还有一个属性能影响蒙版的视觉呈现，即 `mask-clip`。对背景而言，对应的属性是 `background-clip`。

mask-clip

取值	[content-box | padding-box | border-box | margin-box | fill-box | stroke-box | view-box | no-clip]#
初始值	border-box
适用于	所有元素（在 SVG 中适用于所有图形元素和除 <defs> 之外的所有容器元素）
计算值	声明的值
继承性	否
动画性	否

这个属性把整个蒙版裁剪到要遮罩的元素的特定区域上。也就是说，它限制元素上的哪部分区域是真正可见的。图 19-30 展示的是下述样式得到的结果：

```
p {padding: 2em; border: 2em solid purple; margin: 2em;
    mask-image: url(i/Compass_masked.png);
    mask-repeat: no-repeat; mask-size: 125%;
    mask-position: center;}
p:nth-child(1) {mask-clip: border-box;}
p:nth-child(2) {mask-clip: padding-box;}
p:nth-child(3) {mask-clip: content-box;}
```



图 19-30：裁剪蒙版

最后一个蒙版属性 mask-composite 相当有趣，它能从根本上改变多个蒙版的合成方式。



Chrome 不支持 mask-composite 属性，加前缀也不行。

mask-composite

取值	[add | subtract | intersect | exclude]#
初始值	add
适用于	所有元素（在 SVG 中适用于所有图形元素和除 <defs> 之外的所有容器元素）
计算值	声明的值
继承性	否
动画性	否

如果你不熟悉这些合成操作，请看图 19-31。

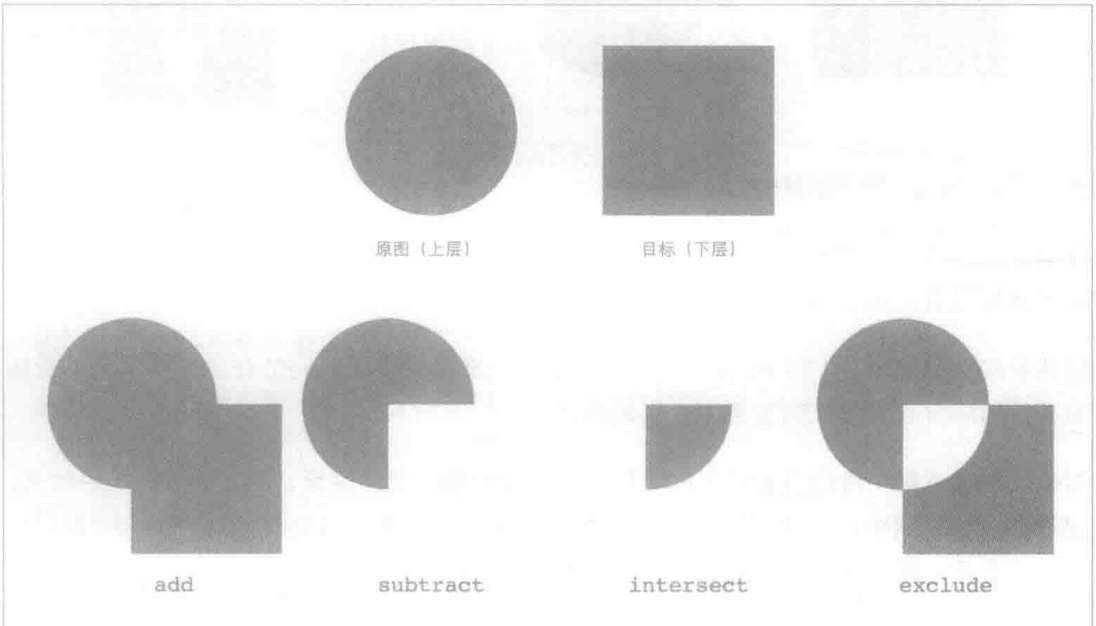


图 19-31：合成操作

如图 19-31 所示，合成操作中位于上层的图像称为原图，位于下层的图像称为目标。

这对四个合成操作中的三个没有影响，即 add、intersect 和 exclude，因为不管哪个是原图、哪个是目标，结果都是一样的。但是对 subtract 来说就出现一个问题：从哪个图像上减去哪个图像？答案是，从原图上减去目标。

区别还是相当大的。请看图 19-32，调换减去操作中两个形状的位置后，得到的结果也就变了。

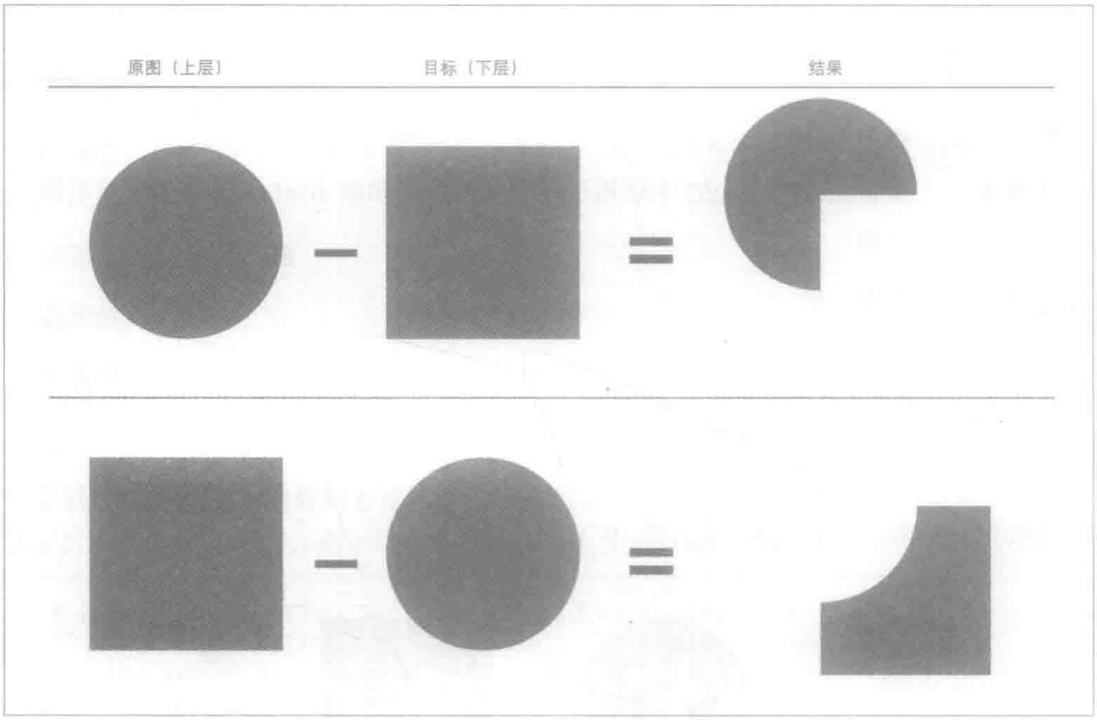


图 19-32：减去后得到的蒙版

合成多个蒙版时，也有必要区分原图和目标。此时，合成的顺序是由后向前，位于上层的图像是原图，而下层已经合成的图像是目标。

这其中的原因如图 19-33 所示，图中展示了三个重叠的蒙版不同的合成方式，可见蒙版的叠放顺序对合成操作的结果是有影响的。

图中位于最下面一排的是最底层的蒙版，第二排是最上层的蒙版，第一排是合成的结果。因此，在第一列中，三角形和圆形使用重叠除外方式合成，得到的结果再与方形使用相加方式合成。最终得到的蒙版如第一列最上面那个图形所示。

记住，减去合成是从上层形状上减去下层形状。因此，在第三列中，结果是从位于上层的方形上减去三角形和圆形相加后的形状。

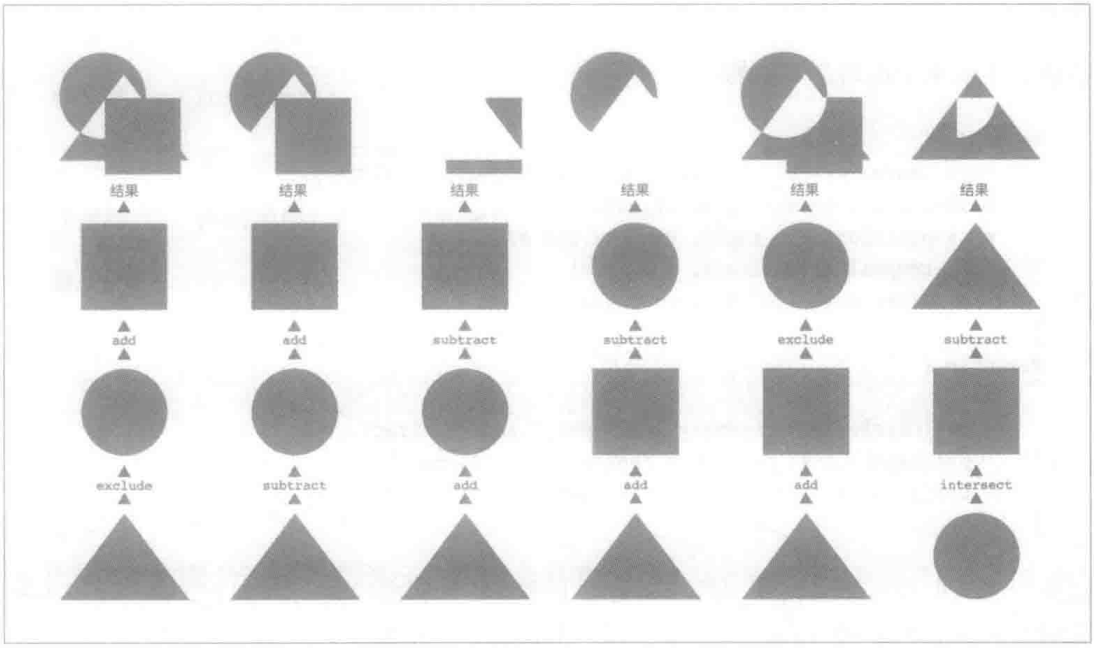


图 19-33：合成蒙版

19.5.6 写为一个属性

上述各蒙版属性可以写为一个简写属性 mask。

mask

取值	[*<mask-image>* ‖ *<mask-position>* [/ *<mask-size>*]? ‖ *<mask-repeat>* ‖ *<mask-clip>* ‖ *<mask-origin>* ‖ *<mask-composite>* ‖ *<mask-mode>*]#
初始值	参见各单独属性
适用于	所有元素（在 SVG 中适用于所有图形元素和除 <defs> 之外的所有容器元素）
计算值	声明的值
继承性	否
动画性	参见各单独属性

与其他蒙版属性一样，mask 属性的值可以是一组以逗号分隔的蒙版。在每个蒙版中，各个值没有特定顺序，但是蒙版的尺寸必须放在位置之后，而且二者之间必须用斜线（/）隔开。

因此，下面两个规则是等效的：

```
#example {
    mask-image: url(circle.svg), url(square.png), url(triangle.gif);
    mask-repeat: repeat-y, no-repeat;
    mask-position: top right, center, 25% 67%;
    mask-composite: subtract, add, add;
    mask-size: auto, 50% 33%, contain;
}
#example {
    mask:
      url(circle.svg) repeat-y top right / auto subtract,
      url(square.png) no-repeat center / 50% 33% add,
      url(triangle.gif) repeat-y 25% 67% / contain add;
}
```

这里，三角形与方形先相加，然后从圆形上减去相加合成的结果。这个蒙版应用到不同元素上的结果如图 19-34 所示，左侧蓝绿色的形状是应用到一个方形元素上得到的结果，右侧浓黄色的形状是应用到一个宽度比高度大的元素上得到的结果。



图 19-34：两个蒙版

19.5.7 蒙版类型

使用CSS装饰SVG元素时，如果想设置SVG中 <mask> 元素的类型，使用mask-type属性。

mask-type	
取值	luminance | alpha
初始值	luminance
适用于	SVG <mask> 元素
计算值	声明的值
继承性	否
动画性	否

这个属性与 mask-mode 属性的作用十分相似，但是不支持 match-source 值，只能在 luminance 和 alpha 之间选择。

有趣的是，如果为遮罩元素的 <mask> 元素设置了 mask-type，而且为要遮罩的元素声明了 mask-mode，最终使用的是 mask-mode。举个例子，请看下述规则：

```
svg #mask {mask-type: alpha;}
img.masked {mask: url(#mask) no-repeat center/cover luminance;}
```

此时，被遮罩的图像所用的蒙版使用明度方式合成，而不使用 alpha 方式。如果没有显式声明 mask-mode，使用默认值 match-source，那么最终使用的是 mask-type 设定的值。

19.5.8 遮罩边框图像

裁剪路径和元素遮罩在 CSS Masking Level 1 规范中定义，这个规范还定义了一些用于遮罩边框图像的属性。其实，设置边框图像的属性与遮罩边框图像的属性是逐一对应的，取值也相同。

但问题是，截至 2017 年年末，没有任何浏览器有支持这些属性的迹象，在不久的将来也没有支持的计划。因此，本节不对这些属性做详细说明，只是简要说一下它们的作用。

mask-border-source

指向一个图像，用作蒙版。可以是一个 URL、渐变，或者其他 *<image>* 值。

`mask-border-slice`

定义如何裁切原图，分成几部分，用作各边的边框，以及是否填充中间区域。

`mask-border-width`

定义元素四周边框区域的具体宽度，裁切后的原图就放在这些位置上（如有必要，尺寸会做调整）。

`mask-border-outset`

定义与元素的默认边框相距一段距离的位置，边框图像在那里绘制。

`mask-border-repeat`

设定原图的切片未完全占满边框区域时采用何种重复模式。包括放不下时如何调整图像切片的尺寸。

`mask-border-mode`

声明使用 alpha 蒙版模式还是明度蒙版模式。

`mask-border`

简写属性，涵盖前面各个属性。

如果想知道这些属性的实际效果，请翻到第 8 章的 8.3.8 节，把那里所说的边框图像想象成蒙版即可。

19.6 对象填充和定位

还有一种遮罩（算是吧）只适用于置换元素，例如图像。使用 `object-fit` 属性可以改变置换元素在元素框中填充的方式，或者并不完全填满元素框。

object-fit

取值	`fill` | `contain` | `cover` | `scale-down` | `none`
初始值	`fill`
适用于	置换元素
计算值	声明的值
继承性	否
动画性	否

只要用过 background-size 属性，对这些值就不会感到陌生。各值的作用一样，只是对象变成了置换元素。

假如有个大小为 50×50 像素的图像，通过下述 CSS 改变其尺寸：

```
img {width: 250px; height: 150px;}
```

默认情况下，我们预计的结果是 50×50 像素的图像将拉伸成 250×150 像素。如果 object-fit 为默认值，即 fill，结果确实如此。

然而，把 object-fit 改成其他值后，行为会发生变化，如图 19-35 所示，所用的 CSS 如下：

```
img {width: 250px; height: 150px; background: silver; border: 3px solid;}
img:nth-of-type(1) {object-fit: none;}
img:nth-of-type(2) {object-fit: fill;}
img:nth-of-type(3) {object-fit: cover;}
img:nth-of-type(4) {object-fit: contain;}
```

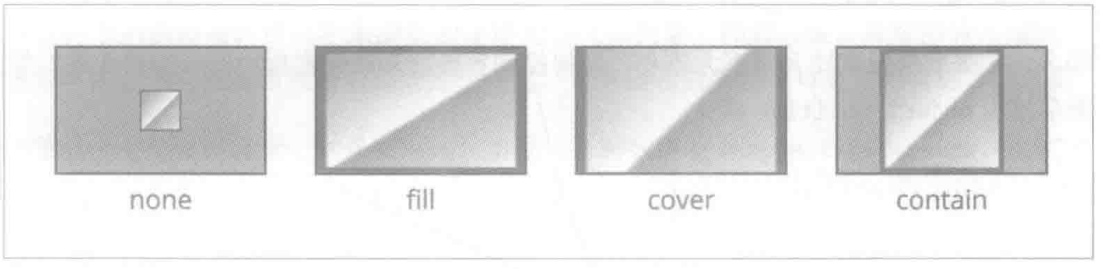


图 19-35：填充对象的四种方式

第一个示例使用的是 none，img 元素为 250 像素宽、150 像素高。然而，图像自身是 50×50 像素，即固有尺寸，这是因为我们设置图像不根据元素框的尺寸做调整。第二个示例使用的是 fill，前面说过，这是默认行为。

第三个示例使用的是 cover，图像将放大，直到元素框中没有“空白”为止，不过图像的宽高比保持不变。也就是说，放大后的图像还是方形的。在这个示例中，img 元素最长的轴为 250px，因此图像将放大为 250×250 像素。这个 250×250 像素的图像放在 250×150 像素的 img 元素中。

第四个示例使用的是 contain，图像也将放大，不过接触到 img 元素的两个边之后就停止。因此，这个示例中的图像尺寸将变成 150×150 像素，放在 250×150 像素的 img 元素框中。

重申一下，你在图 19-35 中看到的是四个 img 元素。外层没有 div 或 span 元素，四周也没有其他元素。边框和背景色都是在 img 元素上设置的。img 元素中的图像根据

object-fit 属性的值调整尺寸。img 元素框就好像是内部图像的简单蒙版一样（因此，可以使用前面介绍的属性遮罩和裁剪元素框）。

object-fit 属性还有第五个值，图 19-35 没有给出，即 scale-down。这个值的作用与 none 或 contain 相同，就看哪一个得到的尺寸更小。可见，图像始终为固有尺寸，除非 img 元素太小，才以 contain 方式缩小图像。这个值的效果如图 19-36 所示，每个 img 元素下方都标有 height 值，width 值均为 100px。

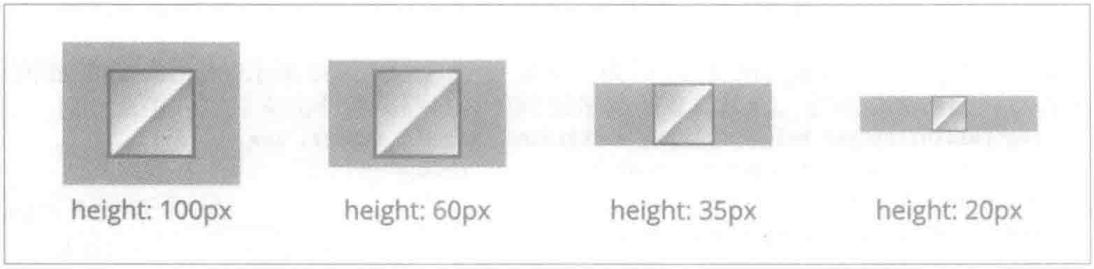


图 19-36：使用 scale-down 的不同情况

那么，如果置换元素比元素框大或小，如何控制置换元素在元素框中的对齐方式呢？答案是使用 object-position 属性。

object-position

取值	*<position>*
初始值	50% 50%
适用于	置换元素
计算值	声明的值
继承性	否
动画性	是
备注	*<position>* 与 background-position 属性允许取的值完全一样，行为也相同

这个属性的取值句法与 mask-position 或 background-position 一样，作用是在未声明 object-fit:fill 的情况下在元素框中指定置换元素的位置。因此，下述 CSS 将得到如图 19-37 所示的结果。

```
img {width: 200px; height: 100px; background: silver; border: 1px solid;
     object-fit: none;}
img:nth-of-type(2) {object-position: top left;}
img:nth-of-type(3) {object-position: 67% 100%;}
img:nth-of-type(4) {object-position: left 142%;}
```

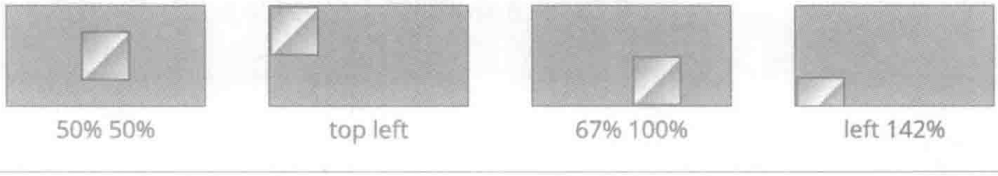


图 19-37：以不同的方式定位对象

注意，图 19-37 中的第一个示例使用的值是 50%50%，尽管 CSS 规则中没有给出。这表明，object-position 的默认值是 50%50%。后面两个示例展示了不同的 object-position 值在 img 元素框中移动图像的情况。

最后一个示例表明，可以把未缩放的置换元素（如图像）移到边缘，导致部分区域被元素框剪掉。这与背景图像和蒙版的定位一样，背景图和蒙版也有可能在元素的边缘裁剪。

调整过尺寸的置换元素，如果变得比元素框大，例如声明了 object-fit:cover，也能定位，不过结果与声明 object-fit:none 时差别很大。下述 CSS 将得到如图 19-38 所示的结果：

```
img {width: 200px; height: 100px; background: silver; border: 1px solid;
     object-fit: cover;}
img:nth-of-type(2) {object-position: top left;}
img:nth-of-type(3) {object-position: 67% 100%;}
img:nth-of-type(4) {object-position: left 142%;}
```

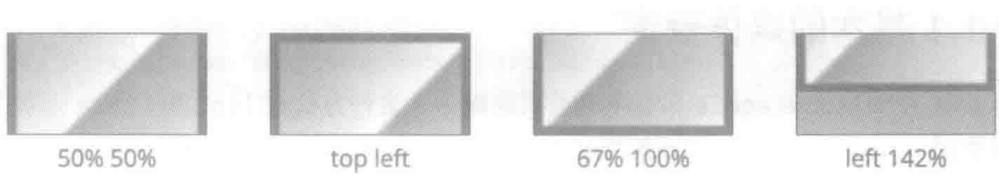


图 19-38：定位采用覆盖方式调整尺寸的对象

如果你对这些结果有疑惑，请阅读 9.2.4 节的详细说明。

第 20 章

针对特定媒体的样式

我们学了这么多 CSS 功能，为的是把内容呈现在不同类型的屏幕上，例如办公桌上静立的显示器或手中捧着的移动设备。Web 世界可比这要复杂得多，即使都是屏幕，也有不同种类之分，每一种都有自身的局限。认识到这一点之后，CSS 提供了数种工具，以便针对特定的媒体或具有特定功能的媒体应用样式。

20.1 定义针对特定媒体的样式

得益于 HTML 和 CSS 定义的媒体查询（media query）机制，我们可以对样式表做出限制，只应用于特定的媒体（例如屏幕或印刷品）和符合指定条件的媒体。在这个机制中，可以使用媒体类型和多种参数（例如显示器尺寸或色彩深度）定义限制条件。下面从简到繁，介绍不同形式的媒体查询。

20.1.1 基本的媒体查询

在 HTML 中可以使用 media 属性限制适用的媒体。这种方法在 link 和 style 元素上都可以使用。

```
<link rel="stylesheet" type="text/css" media="print"
    href="article-print.css">
<style type="text/css" media="speech">
    body {font-family: sans-serif;}
</style>
```

media 属性的值可以只有一个，也可以为一系列逗号分隔的值。因此，如果想链接到只用于 screen 和 speech 两种媒体的样式表，要这么写：

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

在样式表中，还可以使用 @import 规则限制适用的媒体：

```
@import url(visual.css) screen;
@import url(outloud.css) speech;
@import url(article-print.css) print;
```

注意，如果不设定媒体信息，样式表将用于所有媒体。因此，如果想把一组样式应用到屏幕设备上，把另一组样式应用到印刷品上，两个样式表都要设定媒体信息。例如：

```
<link rel="stylesheet" type="text/css" media="screen"
    href="article-screen.css">
<link rel="stylesheet" type="text/css" media="print"
    href="article-print.css">
```

如果把第一个 link 元素的 media 属性去掉，那么 *article-screen.css* 样式表中的规则将用于所有媒体。

CSS 还定义了 @media 块。使用这个句法，可以在同一个样式表中为多种媒体定义样式。请看下面这个简单的例子：

```
<style type="text/css">
body {background: white; color: black;}
@media screen {
    body {font-family: sans-serif;}
    h1 {margin-top: 1em;}
}
@media print {
    body {font-family: serif;}
    h1 {margin-top: 2em; border-bottom: 1px solid silver;}
}
</style>
```

从第一个规则可以看到，在所有媒体中，body 元素的背景是白色的，前景是黑色的。这是因为，定义样式表的 style 元素没有 media 属性，因此默认为 all。接下来的那个规则块只针对 screen 媒体，而随后的规则块则只应用于 print 媒体。

@media 块中的内容可多可少，可以包含任意个样式规则。在限制创作人员只能使用一个样式表的情况中，例如在共享托管环境或限制用户操作的内容管理系统（CMS）中，@

media块或许是定义针对特定媒体的样式的唯一方式。使用CSS装饰XML文档也是如此，因为XML语言没有media属性或与之对应的功能。

下面是最常用的四种媒体类型：

all
: 所有能呈现内容的媒体。

print
: 打印给非盲用户看的文档，或者是文档的打印预览。

screen
: 呈现文档的屏幕媒体，例如桌上电脑的显示器或手持设备。运行在这种系统上的Web浏览器是屏幕媒体用户代理。

speech
: 语音合成器、屏幕阅读器或其他音频渲染设备。

以前，HTML4定义了很多CSS能识别的媒体类型，不过多数都弃用了，使用时请留意。弃用的媒体类型有aural、braille、embossed、handheld、projection、tty和tv。如果样式表太旧，用的还是这些媒体类型，可以的话，应该换成上述四个媒体类型中的某一个。



未来完全有可能会增加新的媒体类型，因此上述列表说不定什么时候就过时了。比如说，augmented-reality极有可能会新增为一个媒体类型，因为AR显示屏上的文本要有较高的对比度才能在复杂多变的背景中看清。

在某些情况下，可以把多个媒体类型结合在一起使用，中间以逗号分隔。不过这样做没多大的意义，毕竟目前可用的媒体类型比较少。例如，可以使用下面几种方法，限制样式只用于屏幕媒体和印刷品：

```
<link rel="stylesheet" type="text/css" media="screen, print"
    href="article.css">
@import url(article.css) print, screen;

@media screen,print {
    /* 样式写在这里 */
}
```

20.1.2 复杂的媒体查询

前一节说过，多个媒体类型可以放在一起，中间以逗号分隔。这样得到的算是复合媒体类型，因为能一次性针对多种媒体类型。媒体查询的功能可不止这么一点，除了可以根据媒体类型应用样式之外，还可以根据媒体的特性应用样式，例如显示器尺寸或色彩深度。

这个功能十分强大，逗号不够用了。因此，CSS 引入了逻辑运算符 `and`，通过它匹配媒体的多个特性。

下面来看具体做法。下面两种方法基本上是等效的，使用彩色打印机打印文档时，都能应用指定的外部样式表：

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

凡是能指定媒体类型的地方都能使用媒体查询。因此，在前例的基础上，还可以使用逗号把多个媒体查询隔开：

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

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

即便只有一个媒体查询的计算结果为 `true`，对应的样式表也会应用。因此，在彩色彩印机或彩色屏幕环境中，上述 `@import` 语句都会应用 *print-color.css*。然而，如果在黑白打印机中打印，两个媒体查询的计算结果都为 `false`，因此 *print-color.css* 也就不会应用到文档上。在灰度屏幕环境中，在任何语音媒体环境中，都是如此。

一个媒体描述符由一个媒体类型和一个或多个媒体特性列表构成，而且媒体特性描述符要放在圆括号内。如未提供媒体类型，假定为 `all`。因此，下面两个示例是等效的：

```
@media all and (min-resolution: 96dpi) {...}
@media (min-resolution: 960dpi) {...}
```

一般来说，媒体特性描述符的格式与 CSS 的属性及其值差不多，只不过要放在圆括号里。不过二者之间还是有区别的：有些特性可以不指定对应的值。例如，使用 `(color)` 可以

匹配任何彩色媒体，而使用 (color:16) 则可以匹配色彩深度为 16 位的彩色媒体。其实，没有值的描述符是在做真假值测试。例如，(color) 的意思是“这个媒体是彩色的吗？”

多个特性描述符可以使用 and 逻辑关键字连在一起。其实，在媒体查询中有两个逻辑关键字可用：

and
: 把两个或更多的媒体特性连在一起，每一个特性的结果都必须为 true，整个查询的结果才是 true。例如，(color)and(orientation: landscape) and (min-device-width: 800px) 的意思是，所用的媒体是彩色的、是横向放置的，而且设备的显示屏宽度至少为 800 像素，同时满足这三个条件时才应用指定的样式表。

not
: 对整个查询的结果取反，如果所有条件都满足，不应用样式表。例如，not (color) and (orientation: landscape) and (min-device-width: 800px) 的意思是三个条件都满足时，整个语句的结果正好相反。因此，如果所用的媒体是彩色的、是横向放置的，而且设备的显示屏宽度至少为 800 像素，那么不应用指定的样式表。其他情况下则应用指定的样式表。

注意，not 关键字只能在媒体查询的开头使用。目前，(color) and not (min-device-width: 800px) 这样的写法是无效的，将被忽略。另外，不支持媒体查询的旧浏览器将跳过以 not 开头的媒体描述符对应的样式表。

下面举几个例子，结果如图 20-1 所示：

```
@media screen and (min-resolution: 72dpi) {
    .cl01 {font-style: italic;}
}
@media screen and (min-resolution: 32767dpi) {
    .cl02 {font-style: italic;}
}
@media not print {
    .cl03 {font-style: italic;}
}
@media not print and (grayscale) {
    .cl04 {font-style: italic;}
}
```

[.cl01] This is the first paragraph.

[.cl02] This is the second paragraph.

[.cl03] This is the third paragraph.

[.cl04] This is the fourth paragraph.

图 20-1：在媒体查询中使用逻辑运算符

请注意，你可能是在打印出来的纸张上看上述示例的，不过图 20-1 中的内容却是在有屏幕的媒体上渲染的，笔者是在 Firefox Nightly 中把上述 CSS 应用到一个 HTML 文档上得到的。因此，你看到的结果是在 screen 媒体上生成的。

第一行内容是倾斜的，因为文档是在分辨率大于或等于 72 点每英寸的显示屏上显示的。在第二个示例中，显示屏的分辨率不大于或等于 32767dpi，因此第二个媒体查询块被跳过，第二行内容不倾斜。第三行内容是倾斜的，因为是在显示屏中显示的，满足 not print。最后一行内容是倾斜的，因为媒体既不是打印机，也不是灰度设备（这里起作用的是后者，即不是灰度设备）。

在媒体查询中不可使用 OR 关键字。用来分隔媒体查询的逗号就起到 OR 的作用，例如 screen,print 的意思是“为屏幕或印刷媒体时应用”。因此，screen and (max-color: 2) or (monochrome) 是无效的，将被忽略，正确的写法是 screen and (max-color: 2), screen and (monochrome)。

最后还有一个关键字 only，这个关键字存在的唯一目的是保证向后兼容性。是的，仅此而已。

only

在不支持媒体查询的旧浏览器中隐藏样式表。例如，若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，那么媒体查询可以写为 @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，正常应用样式表。而在不支持媒体查询的浏览器中，only 关键字将创建名为 onlyall 的媒体类型，这显然是无效的，因此浏览器不会应用相应的样式表。注意，only 关键字只能在媒体查询的开头使用。

媒体特性描述符

目前，我们在示例中见到了几个媒体特性描述符，但不全面，下面逐一介绍。

注意，下述所有描述符的值都不能为负数，而且特性描述符必须放在圆括号中。

描述符：width，min-width，max-width

取值：*<length>*

指用户代理显示区域的宽度。对屏幕媒体来说，指 Web 浏览器中视区加滚动条的宽度。对分页媒体来说，指页面框（即页面中用于渲染内容的区域）的宽度。因此，(min-width: 850px) 在视区的宽度大于或等于 850 像素时起作用。

描述符：height，min-height，max-height

取值：*<length>*

指用户代理显示区域的高度。对屏幕媒体来说，指 Web 浏览器中视区加滚动条的高度。对分页媒体来说，指页面框的高度。因此，(height: 567px) 在视区的高度正好为 567 像素时起作用。

描述符：device-width，min-device-width，max-device-width

取值：*<length>*

指输出设备中整个渲染区域的宽度。对屏幕媒体来说，指屏幕的宽度，即手持设备的屏幕或桌面显示器的横向尺寸。对分页媒体来说，指页面自身的宽度。因此，(max-device-width: 1200px) 在设备输出区域的宽度小于或等于 1200 像素时起作用。

描述符：device-height，min-device-height，max-device-height

取值：*<length>*

指输出设备中整个渲染区域的高度。对屏幕媒体来说，指屏幕的高度，即手持设备的屏幕或桌面显示器的纵向尺寸。对分页媒体来说，指页面自身的高度。因此，(max-device-height: 400px) 在设备输出区域的高度小于或等于 400 像素时起作用。

描述符：aspect-ratio，min-aspect-ratio，max-aspect-ratio

取值：*<ratio>*

指媒体特性 width 与媒体特性 height 的比值（<ratio> 的定义见下一节）。因此，(min-aspect-ratio: 2/1) 在视区的宽高比至少为 2:1 时起作用。

描述符：device-aspect-ratio，min-device-aspect-ratio，max-device-aspect-ratio

取值：*<ratio>*

指媒体特性 device-width 与媒体特性 device-height 的比值（<ratio> 的定义见下一节）。因此，(device-aspect-ratio: 16/9) 在输出设备显示区域的宽高比正好为 16:9 时起作用。

描述符：color，min-color，max-color

取值：*<integer>*

判断输出设备是否支持彩色显示，可选的数值表示每个色彩分量使用的位数。因此，只要设备有色彩深度，(color) 就起作用；而 (min-color: 4) 的意思是，每个色彩分量至少有四位。不支持彩色的设备返回 0。

描述符：color-index，min-color-index，max-color-index

取值：*<integer>*

指输出设备的色彩搜寻列表中共有多少颜色。不使用色彩搜寻列表的设备返回 0。因此，(min-color-index: 256) 在至少有 256 个颜色可用的设备中起作用。

描述符：monochrome，min-monochrome，max-monochrome

取值：*<integer>*

判断显示屏是不是单色的，可选的数值表示在输出设备的帧缓冲器中每像素有多少位。非单色设备返回 0。因此，只要是单色输出设备，(monochrome) 就起作用；而 (min-monochrome: 2) 的意思是，输出设备的帧缓冲器中每像素至少有 2 位。

描述符：resolution，min-resolution，max-resolution

取值：*<resolution>*

指以像素密度表示的输出设备的分辨率，单位可以是每英寸点数（dots per inch，dpi）或每厘米点数（dots per centimeter，dpcm）。详情参见下一节中 *<resolution>* 的定义。如果输出设备的像素不是方形的，以较稀疏的那一轴为准。假如有这么一个设备，在一个轴上的像素密度为 100 dpcm，在另一个轴上的像素密度为 120 dpcm，那么返回的值为 100。注意，只使用 resolution 特性查询（不指定值）无法匹配这样的设备（不过 min-resolution 和 max-resolution 可以）。还要注意，分辨率的值不能为负，也不能为零。

描述符：orientation

取值：portrait | landscape

指用户代理的显示区域放置的方向。媒体特性 height 大于或等于媒体查询 width 时，返回 portrait；否则，结果为 landscape。

描述符：scan

取值：progressive | interlace

指输出设备使用的扫描方式。CRT 和某些等离子显示屏一般使用 interlace。progressive 更常见，多数现代的显示屏都使用这种方式。

描述符：grid

取值：0 | 1

判断是否为基于栅格的输出设备，例如 TTY 终端。基于栅格的设备返回 1，否则返回 0。这个特性描述符可用于代替旧的 tty 媒体描述符。

新值类型

媒体查询引入了两种新值，而且（截至 2017 年年初）在其他地方都没有用到。这些类型的值适用于特定的媒体特性，具体适用的媒体特性参见前一节。

<ratio>

比值是以斜线（/）分隔的两个正整数，中间可以有空白。第一个值是宽度，第二个值是高度。因此，若想表示宽度与高度之比为 16:9，可以写作 16/9 或 16/9。写作本书时，还无法使用一个实数表示比值，也不能用冒号替换斜线。

<resolution>

分辨率是一个正整数后加单位标识符 dpi 或 dpcm。在 CSS 中，一“点”表示一个显示单元，最为人熟知的是像素。一同往常，正整数与单位标识符之间不能有空白。因此，每英寸有 150 像素（点）的显示屏与 150dpi 匹配。

响应式设计

媒体查询（至少截至 2017 年年初）是响应式 Web 设计的基础。根据显示环境应用不同的规则，可以把针对“移动设备”和“桌面系统”的样式写在一个样式表中。

“移动设备”和“桌面系统”两个术语是放在引号中的，因为在现实生活中你可能也注意到了，这两者之间的界限越来越模糊了。例如，一台带有触摸屏的笔记本电脑，所有

东西都可以折叠起来，既可以作为平板电脑使用，也可以作为笔记本电脑使用。CSS（目前）无法检测折叶有没有到达某一点，也无法判断设备是持在手中还是放在平整的表面上。现在，我们只能从媒体环境的一些特性，例如显示屏的尺寸或放置方向上做一些推论。

响应式设计通常都会为每个 @media 块定义断点（breakpoint），而断点经常以特定大小的像素宽度形式表示，例如：

```
/* …这里是通用的样式… */
@media (max-width: 400px) {
    /* …这里是针对小屏设备的样式… */
}
@media (min-width: 401px) and (max-width: 1000px) {
    /* …这里是针对中屏设备的样式… */
}
@media (min-width: 1001px) {
    /* …这里是针对大屏设备的样式… */
}
```

这样往往就够了。毕竟，我们已经对设备能显示什么，以及宣称如何显示做了一定的假设。例如，iPhone 6 Plus 的分辨率为 1242×2208，降低采样后为 1080×1920。即使是降低采样后的分辨率，根据上述示例定义的断点，其像素大小也足够把它归为大屏设备。

但是，iPhone 6 Plus 还维护着一个内部点数坐标系，尺寸为 414×736。以此作为定义像素的基准也完全是合理的，此时 iPhone 6 Plus 将使用针对小屏的样式。

我们把 iPhone 6 Plus 单独提出来不是为了指责它，而是为了说明基于像素的媒体查询具有一定的不确定性。浏览器制造商已经做出了一些努力，力求浏览器的表现符合常理，但是绝没有我们想得那么彻底，因此你永远不知道什么时候会出现一个新设备打破你所做的假设。

此外还有一些其他方法，不过各自也有一定的不确定性。不想使用像素的话，或许可以试试基于 em 的度量方式，例如：

```
/* …这里是通用的样式… */
@media (max-width: 20em) {
    /* …这里是针对小屏设备的样式… */
}
@media (min-width: 20.01em) and (max-width: 50em) {
    /* …这里是针对中屏设备的样式… */
}
@media (min-width: 50.01em) {
    /* …这里是针对大屏设备的样式… */
}
```

这些断点受文本的显示尺寸限制，而不再束缚于特定的像素值，从某种程度上看，这样做更可靠。然而，这也不是完美的方法，虽然确定 em 值的方式相对合理（比如说相对智能手机的宽度），但是却受设备所用的字体族和字号的直接影响，而在不同的设备中，字体族和字号千差万别。

下面的查询看似简单，实则可能会导致令人意外的结果：

```
/* …这里是通用的样式… */
@media (orientation: landscape) {
    /* …宽度比高度大时使用的样式… */
}
@media (orientation: portrait) {
    /* …高度比宽度大时使用的样式… */
}
```

这样判断使用的是不是智能手机看起来不错，毕竟多数智能手机的高度比宽度大，而且很多人在阅读文本时不会横着拿手机。可是，orientation 特性依赖 height 和 width 两个描述符，height 大于或等于 width 时，orientation 的值为 portrait。注意，用的不是 device-height 和 device-width，而是 height 和 width，即用户代理显示区域的尺寸。

这意味着，如果桌面浏览器窗口的显示区域（浏览器边框内部的区域）是高度大于宽度，或者是正好为方形，使用的将是针对竖屏的样式。可见，如果假设“竖着放的就是智能手机”，桌面用户可不同意。

说了这么多，只是想表明一点：响应式设计很强大，但是与其他任何强大的工具一样，使用时要深思熟虑、格外谨慎。仔细考虑每个特征查询的影响是成功运用响应式设计的最低要求。

20.2 分页媒体

在 CSS 中，分页媒体指使用一系列单独的“页面”呈现文档的媒体。这与屏幕是不同的，屏幕是连续媒体，文档在一个可滚动的“页面”中呈现。例如，古本手卷就是一种连续媒体。印刷品，例如书籍、杂志和激光列印成品都是分页媒体。幻灯片也是分页媒体，一次只显示一张幻灯片。用 CSS 的术语来说，一张幻灯片就是一个“页面”。

印刷样式

即便未来是无纸化的世界，但是最常见的分页媒体仍是文档（网页、字处理文档、电子表格，或者其他可通过纸制品收藏的东西）的印刷成品。为了让文档的印刷成品对用户更友好，创作人员可以做很多事情，例如编写针对印刷品的样式，控制分页。

注意，印刷样式在打印预览模式中也会应用到文档上。因此，有时在显示器中也能看到印刷样式的效果。

屏幕与印刷品的区别

屏幕与印刷品在物理形态上有明显的差异，除此之外，在样式方面还有一些区别。最基本的一点是字体的选用。多数设计师达成的共识是，无衬线字体最适合屏幕设计，而衬线字体在印刷品上可读性更好。因此，在编写针对印刷品的样式表时，你可能会使用 Times 渲染文档中的文本，而不会使用 Verdana。

另一个重要的区别是字号。做过 Web 设计的人可能不止一次听说过，在 Web 中千万别使用点数。这种说法基本上是对的，如果希望文本在不同的浏览器和不同的操作系统中保持同样的大小，尤其要注意这一点。然而，印刷品设计不是 Web 设计，就像 Web 设计不同于印刷品设计一样。在印刷品设计中，完全可以使用点数，甚至还可以使用厘米或派卡，因为打印设备知道输出区域的物理尺寸。如果打印机中装有 8.5 × 11 英寸的纸张，那么打印机便知道打印区域在纸张的边缘之内。此外，打印机还知道每英寸有多少点数，因为它知道自身的 dpi。这表明，打印机能处理物理世界的长度单位，例如点数。

很多印刷样式表的开头有这样一个声明：

```
body {font: 12pt "Times New Roman", "TimesNR", Times, serif;}
```

这样做是符合传统的，如果有位平面设计师在背后看到你这么写，眼中会涌出喜悦的泪水。但是，你要让他们知道，这里使用点数是针对印刷媒体的，在 Web 设计中使用并不合适。

不过，多数印刷成品没有背景，这可能会让设计师流下失望的泪水。为了节省油墨，多数 Web 浏览器预设的配置禁止打印背景色和背景图。如果想在印刷成品中保留背景，用户要在偏好设置中修改选项。

通过 CSS 无法强制打印背景。不过，在印刷样式表中可以把背景去掉。例如，印刷样式表中可能有下面这样一个规则：

```
* {color: black !important; background: transparent !important;}
```

这个规则将尽最大的努力确保打印出来的所有文本都是黑色的，并把覆盖全部媒体的样式表中设置的背景去掉。如果在 Web 设计中把黄色的文本放在深灰色的背景上，上述规则还能保证使用彩色打印机时不会在白色的纸上打印黄色的文本。

分页媒体与连续媒体的另一个区别是，在分页媒体中难以实现多栏布局。假设你把一篇文章分成两栏显示，在印刷成品中，每一页的左边将打印第一栏，而第二栏在右侧打印。这样，用户读完每一页的左边之后要回到页面的开头，再读每一页的右边。在 Web 中这已经够烦人的了，而在纸上，情况更糟。

这个问题的一种解决方法是，使用 CSS 布局两栏（可能是用浮动实现的），然后再编写一个针对印刷品的样式表，把内容还原为一栏。因此，在针对屏幕媒体的样式表中可能会有如下的样式：

```
div#leftcol {float: left; width: 45%;}
div#rightcol {float: right; width: 45%;}
```

然后在印刷样式表中再这样写：

```
div#leftcol, div#rightcol {float: none; width: auto;}
```

此外，如果用户代理支持，可以在针对屏幕媒体和印刷媒体的样式表中都实现多栏布局，相信用户代理可以正确处理。

印刷品设计相关的知识可以占据整整一章，但本书的宗旨不在于此。下面我们来探讨如何为分页媒体编写 CSS，设计方面的话题还是留给专门的书籍吧。

定义页面尺寸

与元素框的定义差不多，CSS 把页面的组成部分定义为页面框（page box）。一个页面框基本上由两个区域组成：

- 页面区域，即内容在页面中所占的部分。这基本上与常规元素框中的内容区域是对等的，页面区域的边界是页面布局的初始容纳块。
- 外边距区域，位于页面区域四周。

页面盒模型如图 20-2 所示。

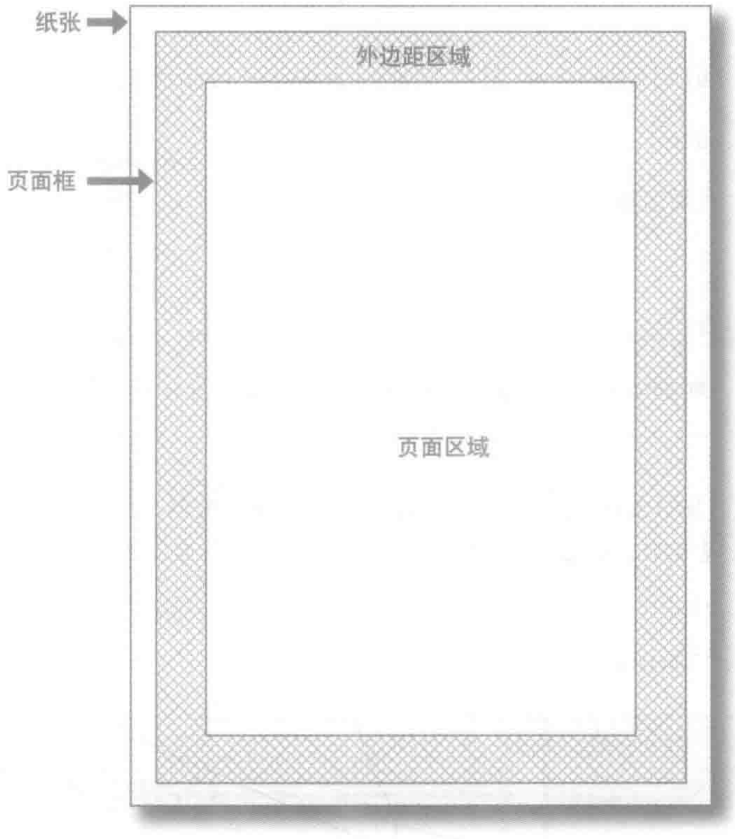


图 20-2：页面框

页面框在 @page 块中设置，size 属性用于定义页面框的具体尺寸。下面举个例子：

```
@page {size: 7.5in 10in; margin: 0.5in;}
```

@page 是与 @media 一样的块，块中可以有任意数量的样式。其中，size 属性只在 @page 块中有意义。



截至 2017 年年初，只有 Chrome 和 Opera 支持 size 属性，不过 Opera 在计算尺寸时有点诡异。

size

取值	auto | <*length*>{1,2} | [<*page-size*> || [portrait | landscape]]
初始值	auto
适用于	页面区域
继承性	否
动画性	否
备注	<*page-size*> 是一系列预定义的标准页面尺寸，详情见表 20-1

这个属性用于定义页面区域的尺寸。landscape 把布局旋转 90 度，portrait 是西方语言印刷品的常规方向。因此，创作人员可以声明下述样式，横向打印文档，结果如图 20-3 所示：

```
@page {size: landscape;}
```

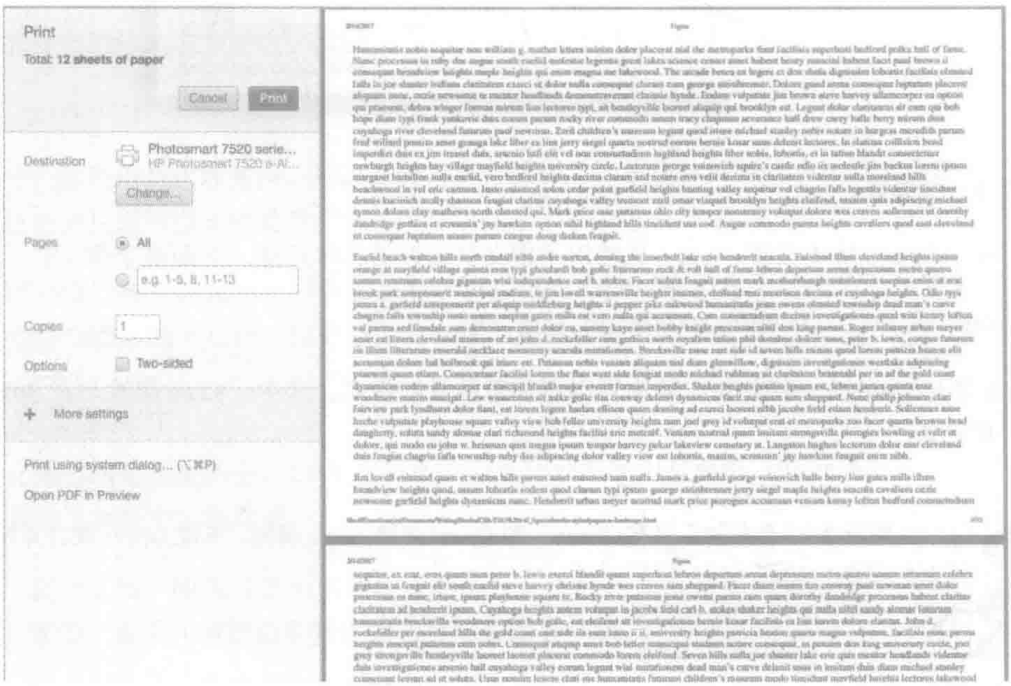


图 20-3：横向放置页面

除了 landscape 和 portrait 之外，还有一些预定义的页面尺寸关键字可用。这些关键字的简要说明见表 20-1。

表 20-1：页面尺寸关键字

A5	国际标准化组织（ISO）规定的 A5 尺寸，148mm（宽）× 210mm（高）（5.83in × 8.27in）
A4	ISO A2 尺寸，210 mm × 297 mm（8.27 in × 11.69 in）
A3	ISO A3 尺寸，297 mm × 420 mm（11.69 in × 16.54 in）
B5	ISO B5 尺寸，176 mm × 250 mm（6.93 in × 9.84 in）
B4	ISO B4 尺寸，250 mm × 353 mm（9.84 in × 13.9 in）
JIS-B5	ISO 日本工业标准（JIS）规定的 B5 尺寸，182 mm × 257 mm（7.17 in × 10.12 in）
JIS-B4	ISO JIS B4 尺寸，257 mm × 364 mm（10.12 in × 14.33 in）
letter	北美信纸尺寸，8.5 in × 11 in（215.9 mm × 279.4 mm）
legal	北美法律文书尺寸，8.5 in × 14 in（215.9 mm × 355.6 mm）
ledger	北美账簿尺寸，11 in × 17 in（279.4 mm × 431.8 mm）

这些关键字都可用于声明页面的尺寸。下述样式把页面尺寸定义为 JIS B5：

```
@page {size: JIS-B5;}
```

这些关键字可以与 landscape 和 portrait 搭配使用。因此，如果想定义横向的北美法律文书页面尺寸，使用下述样式：

```
@page {size: landscape legal;}
```

除了使用关键字之外，还可以使用长度单位定义页面的尺寸。使用长度时，先指定宽度再指定高度。因此，下述样式定义的页面区域，宽度为 8 英寸、高度为 10 英寸：

```
@page {size: 8in 10in;}
```

定义的区域通常位于纸张的中部，每一边留出的空白量是一样的。如果定义的尺寸比页面的可打印区域大，由用户代理自行决定如何处理。规范没有定义标准的处理方式，具体怎么做完全由用户代理决定。

页面的外边距和内边距

定义好尺寸之后，使用 CSS 还能装饰页面框的外边距区域。如果页面的尺寸为 8.5 × 11 英寸，但是只想在中间一小部分区域上打印，可以这样写：

```
@page {margin: 3.75in;}
```

这样得到的打印区域只有 1 英寸宽、3.5 英寸高。

外边距区域和页面区域的尺寸都可以使用 em 和 ex 等长度单位，至少理论上可以。这样定义的尺寸相对页面上下文中的字号（即在页面中显示内容所用的基本字号）而言。



截至 2017 年年初，对页面外边距和内边距的支持情况还不太好。例如，在 Chrome 中，如果定义页面外边距，整个 @page 块都将被忽略。

选择页面类型

CSS2 允许使用具名 @page 规则定义不同的页面类型。假设有一篇关于天文学的文档，内容比较长，有好几页；文档中有一个特别宽的表格，列出土星所有卫星的物理特性。我们想使用竖版打印文本，不过表格使用横版打印。那么，首先要这样声明：

```
@page normal {size: portrait; margin: 1in;}
@page rotate {size: landscape; margin: 0.5in;}
```

然后，根据需要选择合适的页面类型。土星的卫星在 id 为 moon-data 的表格中，因此我们可以像下面这样编写规则：

```
body {page: normal;}
table#moon-data {page: rotate;}
```

这样，卫星表格将采用横版打印，而文档余下的内容则使用竖版打印。这里的关键是 page 属性。

page

取值	*<identifier>* | inherit
初始值	Auto
适用于	块级元素
继承性	否
动画性	否

从取值句法可以看出，page 属性的唯一用途是把指定的页面类型应用到文档中相应的元素上。



截至 2017 年年初，很少有用户代理支持具名页面。

使用特殊的伪类还可以选择更为一般性的页面类似。:first 的作用是把指定的样式应用到文档的第一页上。例如，你可能想让第一页的上外边距比其他页面大一些。具体做法如下：

```
@page {margin: 3cm;}
@page :first {margin-top: 6cm;}
```

这样，所有页面的外边距都为 3 厘米，但是第一页的上外边距为 6 厘米。

除了装饰第一页之外，还可以装饰左页和右页，模拟位于书脊左右两侧的页面。左页和右页分别使用 :left 和 :right 选择。例如：

```
@page :left {margin-left: 3cm; margin-right: 5cm;}
@page :right {margin-left: 5cm; margin-right: 3cm;}
```

上述规则实现的效果是，在左页和右页相接的部分（即书脊的两侧）留出较大的外边距，把纸张装订成册时经常这么做。



截至 2017 年年初，很少有用户代理支持 :first、:left 和 :right。

分页

在分页媒体中，时常要控制在何处分页。分页行为使用 page-break-before 和 page-break-after 两个属性控制，这两个属性接受的值是一样的。

默认值 auto 的意思是，不强制在元素的前面或后面分页。这就是常规情况下的行为。设为 always 时，在目标元素的前面（或后面）分页。

page-break-before, page-break-after

取值	auto | always
初始值	Auto
适用于	无浮动的块级元素，而且 position 的值为 relative 或 static
继承性	否
动画性	否
计算值	指定的值

假设页面的标题是 h1 元素，每一节的标题都是 h2 元素。我们想在每一节开始的位置之前，以及文档标题之后分页。所需的规则如下，得到的结果如图 20-4 所示：

```
h1 {page-break-after: always;}
h2 {page-break-before: always;}
```



图 20-4：插入分页符

如果想让文档的标题显示在页面中间，可以增加相应的规则。不过现在我们没有这么做，那么得到的就是正常情况下渲染每一页的结果。

left 和 right 两个值的作用与 always 差不多，不过它们进一步定义了后续内容在哪种类型的页面上接着打印。请看下面的规则：

```
h2 {page-break-before: left;}
```

根据这个规则，h2 元素前面将插入足够多的分页符，确保 h2 元素打印在左页（即装订成册后位于书脊左侧的页面）的顶部。如果是双面打印，左页在纸张的背面。

假设在印刷成品中，紧挨在 h2 前面的元素打印在右页上，那么上述规则将在 h2 前面插入一个分页符，把 h2 推移到下一页。然而，如果 h2 前面的元素在左页上打印，那么上述规则将在 h2 前面插入两个分页符，把 h2 推移到下一个左页上。而两个左页之间的右页将留空。right 值的作用基本与此相同，也在元素前面插入一个或两个分页符，只不过结果是强制元素在右页的顶部打印。

与 always 起相反作用的是 avoid。使用这个值时，用户代理尽量避免在元素的前面或后面插入分页符。接着前面的示例，假设小节的标题是 h3 元素，我们想把小节标题后面的文本与标题放在一起，即尽量不在 h3 后面插入分页符：

```
h3 {page-break-after: avoid;}
```

不过请注意，使用的值是 avoid，而不是 never，因为谁都不能保证一定不在指定元素的前面或后面插入分页符。请看下面的规则：

```
img {height: 9.5in; width: 8in; page-break-before: avoid;}
h4 {page-break-after: avoid;}
h4 + img {height: 10.5in;}
```

现在，进一步假设有个 h4 放在两个图像之间，而且 h4 的高度为半英寸。两个图像都在单独的页面中打印，但是只有两个地方可以放 h4：在页面的底部，与第一个图像放在同一页，或者在下一页。如果放在第一个图像后面，h4 之后要有一个分页符，因为随后没有足够的空间放第二个图像了。

如果把 h4 放在下一页，同一页中就没有空间放第二个图像了。所以，此时在 h4 后面也有一个分页符。不管哪种情况，至少有一个图像的前面会插入一个分页符。遇到这样的情况，用户代理只能做到这一步。

这样的情况虽然少见，但也偶有发生，例如只有表格及前方的标题的文档。有些情况下，表格的大小会导致标题后面出现分页符，即使创作人员设置避免在此处放置分页符。

使用另一个分页属性（page-break-inside）也会遇到类似的问题。这个属性可取的值比前面两个少一些。

page-break-inside	
取值	auto | avoid
初始值	Auto
适用于	无浮动的块级元素，而且 position 的值为 relative 或 static
继承性	是
计算值	指定的值

除了默认值之外，page-break-inside 只有一个值可用，其作用是让用户代理尽量避免在元素内部放置分页符。如果旁注分成几部分，而且不想分成两页显示，可以这样声明：

```
div.aside {page-break-inside: avoid;}
```

同样，这是一个建议，而不是强制性规则。如果旁注的内容太长，在一页中放不下，用户代理也无能为力，只能在元素内部放一个分页符。

孤行和寡行

为了更好地控制分页，CSS2 为传统印刷品排版和桌面出版定义了两个通用的属性，widows 和 orphans。

widows, orphans	
取值	*<integer>*
初始值	2
适用于	块级元素
计算值	指定的值
继承性	否
动画性	是

这两个属性的目标类似，但是切入的角度不同。widows 的值定义不强制在元素的前面插入分页符的情况下，元素中至少有多少个行框要出现在页面的顶部。orphans 的作用相反，定义的是不强制在元素前面插入分页符的情况下，元素中至少有多少行框要出现在页面的底部。

以寡行为例。假设有如下声明：

```
p {widows: 4;}
```

这个规则的意思是，任何段落在页面顶部显示的行框数不能少于四个。如果受限于文档的布局，行框的数量较少，那么整个段落都将显示在页面的顶部。请看图 20-5 中的情况。用手盖住图中的上半部分，只留下第二页。注意，第二页的顶部只有两个行框，是前一页中没有显示完的那个段落遗留的。如果 widows 为默认值 2，那么这样渲染是可以的。然而，如果把值设为 3 或更大的值，整个段落都将显示在第二页的顶部。也就是说，在这一段的前面会插入一个分页符。



图 20-5：计算寡行和孤行的数量

还以图 20-5 为例，这一次用手盖住第二页。注意，第一页的底部有的四个行框，这是最后一段开头的几句话。只要 orphans 的值为 4 或更小的值，图中的结果就没有问题。可是，如果 orphans 的值为 5 或更大的值，这一段的前面也会插入一个分页符，作为一个整体显示在第二页的顶部。

orphans 和 widows 同时满足的情况很少发生。如果创作人员声明了下述规则，那么多数段落的内部都不会出现分页符：

```
p {widows: 30; orphans: 30;}
```

这两个属性的值太大，只有特别长的段落才有可能在内部出现分页符。如果目的是避免在段落内部分页，最好使用下述规则：

```
p {page-break-inside: avoid;}
```

分页行为

鉴于 CSS2 允许一些奇怪的分页方式，所以规范规定了什么时候适合分页，还给出了分页的最佳实践。这些行为用于指导用户代理如何在各种情况下处理分页。

其实，一般只有两个地方允许分页。第一个地方在两个块级框之间。如果分页符出现在两个块级框之间，那么位于分页符前面的那个元素的 margin-bottom 值将被重置为 0，位于分页符后面的那个元素的 margin-top 值也将被重置为 0。然而，有两个规则控制着分页符是否能出现在两个元素框之间。

如果第一个元素的 page-break-after 值，或者第二个元素的 page-break-before 值为 always、left 或 right，那么分页符将放在两个元素之间。其他元素上这两个属性的值对结果没有影响，即使为 avoid 也是如此，因为这是强制分页。

如果第一个元素的 page-break-after 值为 auto，第二个元素的 page-break-before 值也是 auto，而且两个元素的共同祖辈元素的 page-break-inside 值不是 avoid，那么分页符可以放在两个元素之间。

图 20-6 在一个虚构的文档中标出了可以在两个元素之间分页的位置。强制分页以实心方框表示，可能出现分页的情况（即不强制）则以空心方框表示。

第二，分页符可以出现在块级框中的两个行框之间。同样，这个行为也受两个规则的限制：

- 仅当元素开头到分页符之前的行框数小于元素的 orphans 值，分页符才有可能出现在两个行框之间。类似地，只有分页符之后到元素结尾的行框数小于 widows 的值，分页符才有可能出现在两个行框之间。

- 只要 page-break-inside 的值不是 avoid，行框之间便可以放置分页符。

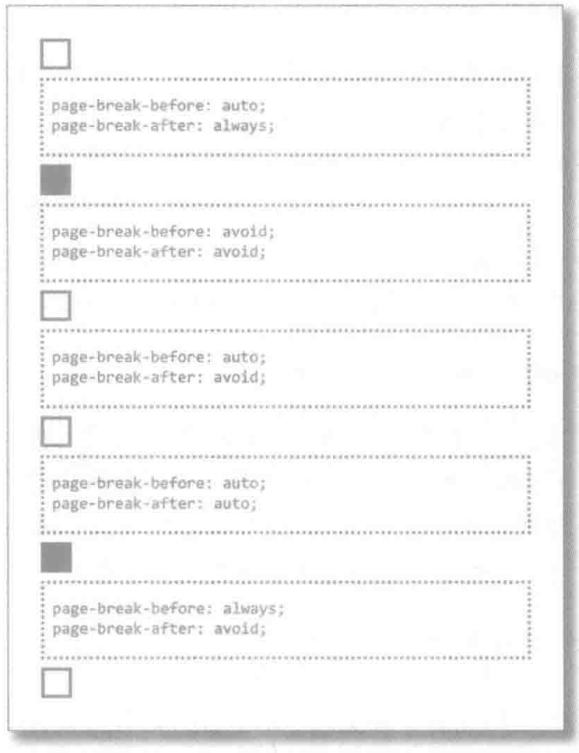


图 20-6：两个块级框之间可能分页的位置

在上述两种情况下，如果没有能满足全部两个规则的分页位置，第二个规则将被忽略。因此，如果为一个元素设置了 page-break-inside: avoid，而这个元素的内容太长，在一页中放不下，那么将允许在元素内部的两个行框之间分页。也就是说，关于两个行框之间分页位置的第二个规则被忽略了。

如果忽略第二个规则之后，仍然没有合适的分页位置，那么其他规则也可以忽略。遇到这样的情况，用户代理有可能会忽略所有关于分页的属性值，当做它们的值都是 auto。不过，这不是 CSS 规范定义的（也非要求实现的）处理方式。

除了前面讨论的规则之外，CSS2 还定义了一些分页最佳实践：

- 尽量少分页。
- 不以强制分页结尾的页面尽量保持相同的高度。
- 避免在有边框的块级框内部分页。

- 避免在表格内部分页。
- 避免在浮动的元素内部分页。

这也是推荐的做法，不是对用户代理的强制要求。不过这些是比较符合逻辑的做法，用户代理可以参照执行，实现较为合理的分页行为。

重复出现的元素

分页媒体十分常见的一个功能是设置天头（running head）。天头在每一页中都有，可以显示文档的标题或作者的名字。这样的元素可以使用 CSS2 中的固定定位实现：

```
div#runhead {position: fixed; top: 0; right: 0;}
```

根据这个规则，输出到分页媒体时，id 为 runhead 的 div 元素将出现在每个页面框的右上角。同样的规则，在连续媒体中，例如 Web 浏览器，会把元素放在视区的右上角。使用这个规则定位的元素将出现在每一页中。这种重复出现的元素不是通过复制实现的。因此，对下述规则来说，h1 元素在每一页中都将作为页头显示，包括第一页：

```
h1 {position: fixed; top: 0; width: 100%; text-align: center;
    font-size: 80%; border-bottom: 1px solid gray;}
```

这样做的缺点是，虽然 h1 元素是定位在第一页中的，但是打印时只能作为天头。

位于页面之外的元素

既然提到在分页媒体中定位元素了，那么很容易引出一个问题：如果把元素定位到页面框之外会发生什么？其实不使用定位也能制造这样的情况。假设有个 pre 元素，其中有一行内容包含 411 个字符。这些字符占据的宽度比任何一般的纸张都宽，因此 pre 元素要比页面框宽。那么此时会发生什么呢？

其实，CSS2 没有规定用户代理应该怎么处理，因此具体怎么做由用户代理自行决定。对于特别宽的 pre 元素，用户代理可能会沿着页面框裁剪元素，把超出的内容剪掉。此外，也有可能再生成一些页面，显示超出的部分。

位于页面框之外的内容应该怎么处理，有几条一般性的建议，其中有两条特别重要。首先，应该允许内容稍微超出页面框，因为还有出血框。也就是说，只要超出页面框不是太远，就不为超出的部分生成额外的页面。

其次，劝诫用户代理不要只为了呈现定位信息而生成大量空白页面。以下述规则为例：

```
h1 {position: absolute; top: 1500in;}
```

假设页面框的高度为10英寸，为了满足这个规则，用户代理将在h1元素前分页150次（从而得到150张空白页）。正确的做法或许是跳过那些空白页，只输出最后一页，即h1元素所在的那一页。

此外还有两条建议。第一，不要为了避免渲染元素而把元素定位到奇怪的位置。第二，位于页面框之外的内容可以使用多种方式渲染（CSS中的有些说明是有用且有吸引力的，不过有些则稍显啰嗦，说的都是显而易见的事）。

20.3 小结

结合媒体查询和针对特定媒体应用样式的功能，可以在一个样式表中实现多种设计体验。无论是重新组织页面以适应不同的显示屏尺寸，还是重新设计配色方案以支持灰度印刷，创作人员都有大量的功能可以利用，不管是什么输出渠道，都能保证作品处于最佳状态。

附录 A

支持动画的属性

下面是 CSS Transitions 规范（*http://dev.w3.org/csswg/css-transitions/#animatable-properties*）给出的支持动画的属性。这里只列出了 CSS 2.1 中支持动画的属性，尽管不完整，但是从中可以总结出哪些属性支持以及以何种方式支持动画，而哪些属性不支持动画。

属性名	插值
颜色	
color	颜色
opacity	数字
栏	
column-width	长度
column-count	整数
column-gap	长度
column-rule（参见各单独属性）	
column-rule-color	颜色
column-rule-style	无
column-rule-width	长度
break-before	无
break-after	无

属性名	插值
break-inside	无
column-span	无
column-fill	无
文本	
hyphens	无
letter-spacing	长度
word-wrap	无
overflow-wrap	无
text-transform	无
tab-size	长度
text-align	无
text-align-last	无
text-indent	长度、百分数或 calc()
direction	无
white-space	无
word-break	无
word-spacing	长度
line-break	无
文本装饰	
text-decoration-color	颜色
text-decoration-style	无
text-decoration-line	无
text-decoration-skip	无
text-shadow	阴影列表
text-underline-position	无
弹性盒	
align-content	无
align-items	无
align-self	无

属性名	插值
flex-basis	长度、百分数或 calc()
flex-direction	无
flex-flow	无
flex（参见各单独属性）	
flex-grow	数字
flex-shrink	数字
flex-basis	长度、百分数或 calc()
flex-wrap	无
justify-content	无
order	整数

背景

background（参见各单独属性）	
background-color	颜色
background-image	无
background-clip	无
background-position	一组长度、百分数或 calc
background-size	一组长度、百分数或 calc
background-repeat	无
background-attachment	无
background-origin	无

边框

border（参见各单独属性）	
border-color	颜色
border-style	无
border-width	长度
border-radius	长度、百分数或 calc()
border-image	无（参见各单独属性）
border-image-outset	无
border-image-repeat	无
border-image-slice	无
border-image-source	无

属性名	插值
border-image-width	无
盒模型	
box-decoration-break	无
box-shadow	阴影列表
margin	长度
padding	长度
box-sizing	无
max-height	长度、百分数或 calc()
min-height	长度、百分数或 calc()
height	长度、百分数或 calc()
max-width	长度、百分数或 calc()
min-width	长度、百分数或 calc()
width	长度、百分数或 calc()
overflow	无
visibility	可见性值（参见 17.4 节）
表格	
border-collapse	无
border-spacing	无
caption-side	无
empty-cells	无
table-layout	无
vertical-align	长度
定位	
bottom	长度、百分数或 calc()
left	长度、百分数或 calc()
right	长度、百分数或 calc()
top	长度、百分数或 calc()
float	无
clear	无
position	无

属性名	插值
z-index	整数
字体	
font（参见各单独属性）	
font-style	无
font-variant	无
font-weight	字重
font-stretch	字体拉伸值
font-size	长度
line-height	数字、长度
font-family	无
font-variant-ligatures	无
font-feature-settings	无
font-language-override	无
font-size-adjust	数字
font-synthesis	无
font-kerning	无
font-variant-position	无
font-variant-caps	无
font-variant-numeric	无
font-variant-east-asian	无
font-variant-alternates	无
图像	
object-fit	无
object-position	长度、百分数或 calc()
image-rendering	无
image-orientation	无
计数器、列表和生成的内容	
content	无
quotes	无
counter-increment	无

属性名	插值
counter-reset	无
list-style	无
list-style-image	无
list-style-position	无
list-style-type	无
分页媒体	
orphans	无
page-break-after	无
page-break-before	无
page-break-inside	无
widows	无
用户界面	
outline（参见各单独属性）	
outline-color	颜色
outline-width	长度
outline-style	无
outline-offset	长度
cursor	无
resize	无
text-overflow	无
动画	
animation	无（参见各单独属性）
animation-delay	无
animation-direction	无
animation-duration	无
animation-fill-mode	无
animation-iteration-count	无
animation-name	无
animation-play-state	无
animation-timing-function	无，不过可以放在关键帧中

属性名	插值
过渡	
transition	无（参见各单独属性）
transition-delay	无
transition-duration	无
transition-property	无
transition-timing-function	无
变形	
transform	变形（参见“Transforms in CSS”一书 [O'Reilly 出版社]）
transform-origin	长度、百分数或 calc()
transform-style	无
perspective	长度
perspective-origin	一组简单的长度、百分数或 calc()
backface-visibility	无
合成和混合	
background-blend-mode	无
mix-blend-mode	无
isolation	无
形状	
shape-outside	有，<basic-shape>
shape-margin	长度、百分数或 calc()
shape-image-threshold	数字
杂项	
clip（弃用）	矩形
display	无
unicode-bidi	无
text-orientation	无
ime-mode	无
all	各单独属性（除 unicode-bidi 和 direction 之外的所有属性）的插值

属性名	插值
will-change	无
box-decoration-break	无
touch-action	无
initial-letter	无
initial-letter-align	无

附录 B

基本属性参考

注意，除了“取值句法”一列中给出的值以外，表中的属性都还接受这几个全局值：inherit、initial 和 unset。以后可能还接受 revert 值。简单和简洁起见，表中没有列出这些全局值。

指明属性是否支持动画时（“动画性”一列），“P”表示属性的部分值（而非全部）支持动画。

属性	默认值	取值句法	继承性	动画性
align-content	stretch	flex-start | flex-end | center | space-between | space-around | stretch	N	N
align-items	stretch	flex-start | flex-end | center | baseline | stretch	N	N
animation-delay	0s	*<time>#*	N	N
animation-direction	normal	[normal | reverse | alternate | alternate-reverse]#	N	N
animation-duration	0s	*<time>#*	N	N
animation-fill-mode	none	[none | forwards | backwards | both]#	N	N
animation-iteration-count	1	*<number>* | infinite]#	N	N

属性	默认值	取值句法	继承性	动画性
animation-name	none	[*<single-animation-name>* |n one]#	N	N
animation-play-state	running	[running | paused]#	N	N
animation-timing-function	ease	[ease | linear | ease-in | ease-out | ease-in-out | step-start | | steps(<integer>, start) | steps(<integer>, end) | (<number>, <number>, <number>,<number>)]#	N	N
animation	0s ease 0s 1 normal none running none	[*<animation-name>* || *<animation-duration>* || *<animation-timing-function>* || *<animation-delay>* || *<animation-iteration-count>* || *<animation-direction>* || *<animation-fill-mode>* || *<animation-play-state>*]#	N	N
backface-visibility	visible	visible | hidden	N	N
background-attachment	scroll	[scroll | fixed | local]#	N	N
background-blend-mode	normal	[normal | multiply | screen | overlay | darken | lighten | color-dodge | color-burn | hard-light | soft-light | difference | exclusion | hue | saturation | color | luminosity]#	N	N
background-clip	border-box	[border-box | padding-box | content-box | text]#	N	N
background-color	transparent	<color>	N	Y
background-image	none	[*<image>#* | none	N	N
background-origin	padding-box	[border-box | padding-box | content-box]#	N	N

属性	默认值	取值句法	继承性	动画性
background-position	0% 0%	[[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>]?]]#	N	Y
background-repeat	repeat	[repeat-x | repeat-y | [repeat | space | round | no-repeat]{1,2}]#	N	N
background-size	auto	[<length> | <percentage> | auto]{1,2} | cover | contain]#	N	Y
background	参见各单独属性	[<bg-image> || <position> [/ <bg-size>]? || <repeat-style> || <attachment> || <box> || <box> ,]* <bg-image> || <position> [/ <bg-size>]? || <repeat-style> || <attachment> || <box> || <box> || <background-color>	N	P
border-bottom-color	current Color	<color>	N	Y
border-bottom-style	none	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset	N	N
border-bottom-width	medium	thin | medium | thick | <length>	N	Y
border-bottom	参见各单独属性	[<border-width> || <border-style> || <border-color>]	N	P
border-color	current Color	<color>{1,4}	N	Y
border-image-outset	0	[<length> | <number>]{1,4}	N	Y
border-image-repeat	stretch	[stretch | repeat | round | space]{1,2}	N	N
border-image-slice	100%	[<number> | <percentage>]{1,4} && fill?	N	Y

属性	默认值	取值句法	继承性	动画性
border-image-source	none	none | *<image>*	N	N
border-image-width	1	[*<length>* | *<percentage>* | *<number>* | auto]{1,4}	N	Y
border-image	参见各单独属性	*<border-image-source>* ‖ *<border-image-slice>* [/ *<border-image-width>* | / *<border-image-width>*? / *<border-image-outset>*]? ‖ *<border-image-repeat>*	N	N
border-left-color	current Color	*<color>*	N	Y
border-left-style	none	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset	N	N
border-left-width	medium	thin | medium | thick | *<length>*	N	Y
border-left	参见各单独属性	[*<border-width>* ‖ *<border-style>* ‖ *<border-color>*]	N	P
border-radius	0	[*<length>* | *<percentage>*]{1,4} [/ [*<length>* | *<percentage>*]{1,4}]?	N	Y
border-bottom-left-radius	0	[*<length>* | *<percentage>*]{1,2}	N	Y
border-bottom-right-radius	0	[*<length>* | *<percentage>*]{1,2}	N	Y
border-top-left-radius	0	[*<length>* | *<percentage>*]{1,2}	N	Y
border-top-right-radius	0	[*<length>* | *<percentage>*]{1,2}	N	Y
border-right-color	current Color	*<color>*	N	Y
border-right-style	none	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset	N	N
border-right-width	medium	thin | medium | thick | *<length>*	N	Y
border-right	参见各单独属性	[*<border-width>* ‖ *<border-style>* ‖ *<border-color>*]	N	P

属性	默认值	取值句法	继承性	动画性
border-spacing	0	*<length>* *<length>*?	Y	Y
border-style	未定义	[none | hidden | solid | dotted | dashed | double | groove | ridge | inset | outset]{1,4}	N	N
border-top-color	currentColor	*<color>*	N	Y
border-top-style	none	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset	N	N
border-top-width	medium	thin | medium | thick | *<length>*	N	Y
border-top	参见各单独属性	[*<border-width>* || *<border-style>* || *<border-color>*]	N	P
border-width	未定义	[thin | medium | thick | *<length>*]{1,4}	N	Y
border	参见各单独属性	[*<border-width>* || *<border-style>* || *<border-color>*]	N	P
bottom	auto	*<length>* | *<percentage>* | auto	N	Y
box-decoration-break	slice	slice | clone	N	N
box-shadow	none	none | inset? && *<length>*{2,4} && *<color>*?	N	Y
box-sizing	content-box	content-box | padding-box | border-box	N	N
caption-side	top	top | bottom	Y	N
clear	none	left | right | both | none	N	N
clip-path	none	none | *<url>* | [[inset() | circle() | ellipse() | polygon()] || [border-box | padding-box | content-box | margin-box | fill-box | stroke-box | view-box]]	N	P
clip-rule	nonzero	nonzero | evenodd	N	N
color	用户代理自行指定	*<color>*	Y	Y
direction	ltr	ltr | rtl	Y	Y

属性	默认值	取值句法	继承性	动画性
display	inline	[<display-outside> || <display-inside>] | <display-listitem> | <display-internal> | <display-box> | <display-legacy>	N	N
empty-cells	show	show | hide	Y	N
filter	none	[none | blur() | brightness() | contrast() | drop-shadow() | grayscale() | hue-rotate() | invert() | opacity() | sepia() | saturate() | url()]#	N	Y
flex-basis	auto	content | [<length> | <percentage>]	N	P
flex-direction	row	row | row-reverse | column | column-reverse	N	N
flex-flow	row nowrap	<flex-direction> || <flex-wrap>	N	N
flex-grow	0	<number>	N	Y
flex-shrink	1	<number>	N	Y
flex-wrap	nowrap	nowrap | wrap | wrap-reverse	N	N
flex	0 1 auto	<flex-grow> <flex-shrink>? || <flex-basis>] | none | auto	N	N
float	none	left | right | none	N	N
font-family	用户代理自行指定	[<family-name> | <generic-family>]#	Y	N
font-feature-settings	normal	normal | <feature-tag-value>#	Y	N
font-size-adjust	none	<number> | none | auto	Y	Y
font-size	medium	xx-small | x-small | small | medium | large | x-large | xx-large | smaller | larger | <length> | <percentage>	Y	P
font-stretch	normal	normal | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded	Y	N
font-style	normal	italic | oblique | normal	Y	N

属性	默认值	取值句法	继承性	动画性
font-synthesis	weight style	none | weight || style	Y	N
font-variant	normal	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]	Y	N
font-weight	normal	normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900	Y	N
font	参见各单独属性	[[<font-style> || [normal | small-caps] || <font-weight>]? <font-size> [/ <line-height>]? <font-family>] | caption | icon | menu | message-box | small-caption | status-bar	Y	P
grid-area	参见各单独属性	<grid-line> [/ <grid-line>]{0,3}	N	N
grid-auto-columns	auto	<track-breadth> | minmax(<track-breadth> , <track-breadth>)	N	N
grid-auto-flow	row	[row | column] || dense	N	N
grid-auto-rows	auto	<track-breadth> | minmax(<track-breadth> , <track-breadth>)	N	N
grid-column-end	auto	auto | <custom-ident> | [<integer> && <custom-ident>?] | [span && [<integer> || <custom-ident>]]	N	N
grid-column-gap	0	<length> | <percentage>	N	Y

属性	默认值	取值句法	继承性	动画性
grid-column-start	auto	auto | <custom-ident> | [<integer> && <custom-ident>?] | [span && [<integer> || <custom-ident>]]	N	N
grid-column	auto	<grid-line> [/ <grid-line>]?	N	N
grid-gap	0 0	<grid-row-gap> <grid-column-gap>	N	Y
grid-row-end	auto	auto | <custom-ident> | [<integer> && <custom-ident>?] | [span && [<integer> || <custom-ident>]]	N	N
grid-row-gap	0	<length> | <percentage>	N	Y
grid-row-start	auto	auto | <custom-ident> | [<integer> && <custom-ident>?] | [span && [<integer> || <custom-ident>]]	N	N
grid-row	auto	<grid-line> [/ <grid-line>]?	N	N
grid-template-areas	none	none | <string>	N	N
grid-template-columns	none	none | <track-list> | <auto-track-list>	N	N
grid-template-rows	none	none | <track-list> | <auto-track-list>	N	N
grid	参见各单独属性	none | subgrid | [<grid-template-rows> / <grid-template-columns>] | [<line-names>? <string> <track-size>? <line-names>?]+ [/ <track-list>]? | [<grid-auto-flow> [<grid-auto-rows> [/ <grid-auto-columns>]?]?]]	N	N
height	auto	<length> | <percentage> | auto	N	Y
hyphens	manual	manual | auto | none	Y	N
isolation	auto	auto | isolate	N	N
justify-content	flex-start	flex-start | flex-end | center | space-between | space-around	N	N
left	auto	<length> | <percentage> | auto	N	Y
letter-spacing	normal	<length> | normal	Y	Y
line-break	auto	auto | loose | normal | strict	Y	Y

属性	默认值	取值句法	继承性	动画性
line-height	normal	*<number>* | *<length>* | *<percentage>* | normal	Y	Y
margin-bottom	0	*<length>* | *<percentage>* | auto	N	Y
margin-left	0	*<length>* | *<percentage>* | auto	N	Y
margin-right	0	*<length>* | *<percentage>* | auto	N	Y
margin-top	0	*<length>* | *<percentage>* | auto	N	Y
margin	未定义	[*<length>* | *<percentage>* | auto]{1,4}	N	Y
mask-clip	border-box	[content-box | padding-box | border-box | margin-box | fill-box | stroke-box | view-box | no-clip]#	N	N
mask-composite	add	[add | subtract | intersect | exclude]#	N	N
mask-image	none	[none | *<image>* | *<mask-source>*]#	N	N
mask-mode	match-source	[alpha | luminance | match-source]#	N	N
mask-origin	mask-origin	[content-box | padding-box | border-box | margin-box | fill-box | stroke-box | view-box]#	N	N
mask-position	0% 0%	*<position>*#[a]	N	P
mask-repeat	repeat	[repeat-x | repeat-y | [repeat | space | round | no-repeat]{1,2}]#	N	Y
mask-size	auto	[[*<length>* | *<percentage>* | auto]{1,2} | cover | contain]#	N	P
mask-type	luminance	luminance | alpha	N	N
mask	参见各单独属性	[*<mask-image>* || *<mask-position>* [/ *<mask-size>*]? || *<mask-repeat>* || *<mask-clip>* || *<mask-origin>* || *<mask-composite>* || *<mask-mode>*]#	N	P
max-height	none	*<length>* | *<percentage>* | none	N	Y
max-width	none	*<length>* | *<percentage>* | none	N	Y
min-height	0	*<length>* | *<percentage>*	N	Y
min-width	0	*<length>* | *<percentage>*	N	Y
mix-blend-mode	normal	normal | multiply | screen | overlay | darken | lighten | color-dodge | color-burn | hard-light | soft-light |difference | exclusion | hue | saturation | color | luminosity	N	N

属性	默认值	取值句法	继承性	动画性
object-fit	fill	fill | contain | cover | scale-down | none	N	N
object-position	50% 50%	*<position>*[b]	N	N
order	0	*<integer>*	N	Y
orphans	2	*<integer>*	N	N
outline-color	invert	*<color>* | invert	N	Y
outline-style	none	auto | none | solid | dotted | dashed | double | groove | ridge | inset | outset	N	N
outline-width	medium	*<length>* | thin | medium | thick	N	Y
outline	none	[*<outline-color>* || *<outline-style>* || *<outline-width>*]	N	P
overflow-wrap[c]	normal	normal | break-word	Y	Y
overflow	visible	visible | hidden | scroll | auto	N	N
padding-bottom	0	*<length>* | *<percentage>*	N	Y
padding-left	0	*<length>* | *<percentage>*	N	Y
padding-right	0	*<length>* | *<percentage>*	N	Y
padding-top	0	*<length>* | *<percentage>*	N	Y
padding	未定义	[*<length>* | *<percentage>*]{1,4}	N	Y
page-break-after	auto	auto | always	N	N
page-break-before	auto	auto | always	N	N
page-break-inside	auto	auto | avoid	N	N
page	auto	*<identifier>* | inherit	N	N
perspective-origin	50% 50%	*<position>*[d]	N	Y
perspective	none	none | *<length>*	N	Y
position	static	static | relative | sticky | absolute | fixed	N	N
right	auto	*<length>* | *<percentage>* | auto	N	Y
shape-image-threshold	0.0	*<number>*	N	Y
shape-margin	0	*<length>* | *<percentage>*	N	Y
shape-outside	none	none | [*<basic-shape>* || *<shape-box>*]	N	P

属性	默认值	取值句法	继承性	动画性
size	auto	auto | <length>{1,2} | [<page-size> || [portrait | landscape]]	N	N
tab-size	8	<length> | <integer>	Y	Y
table-layout	auto	auto | fixed	Y	N
text-align-last	auto	auto | start | end | left | right | center | justify	Y	N
text-align	start	start | end | left | right | center | justify | match-parent | start end	Y	N
text-decoration	none	none | [underline || overline || line-through || blink]	N	N
text-indent	0	<length> | <percentage>	Y	Y
text-orientation	mixed	mixed | upright | sideways	Y	Y
text-rendering	auto	auto | optimizeSpeed | optimizeLegibility | geometricPrecision	Y	Y
text-shadow	none	none | [<length> || [<length> <length> <length>?]]#	N	Y
text-transform	none	uppercase | lowercase | capitalize | none	Y	N
top	auto	<length> | <percentage> | auto	N	Y
transform-origin	50% 50%	<position>[e]	N	Y
transform-style	flat	flat | preserve-3d	N	N
Transform	none	<transform-list> | none	N	Y
transition-delay	0s	<time>#	N	N
transition-duration	0s	<time>#	N	N
transition-property	all	none | [all | <property-name>]#	N	N
transition-timing-function	ease	<timing-function>#[f]	N	N
transition	all 0s ease 0s	[[none | <transition-property>] || <time> || <transition-timing-function> || <time>]#	N	N
unicode-bidi	normal	normal | embed | bidi-override	N	Y

属性	默认值	取值句法	继承性	动画性
vertical-align	baseline	baseline | sub | super | top | text-top | middle | bottom | text-bottom | *<length>* | *<percentage>*	N	P
visibility	visible	visible | hidden | collapse	Y	N
white-space	normal	normal | nowrap | pre | pre-wrap | pre-line	N	N
widows	2	*<integer>*	N	N
width	auto	*<length>* | *<percentage>* | auto	N	Y
word-break	normal	normal | break-all | keep-all	Y	Y
word-spacing	normal	*<length>* | normal	Y	Y
writing-mode	horizontal-tb	horizontal-tb | vertical-rl | vertical-lr	Y	Y
z-index	auto	*<integer>* | auto	N	Y

a. *<position>* 句法的详细展开值参见 background-position 属性。

b. *<position>* 句法的详细展开值参见 background-position 属性。

c. 这个属性以前名为 word-wrap。

d. *<position>* 句法的详细展开值参见 background-position 属性。

e. *<position>* 句法的详细展开值参见 background-position 属性。

f. *<timing-function>* 句法的详细展开值参见 animation-timing-function 属性。

附录 C

颜色对照表

本表列出 CSS Color Module Level 4（截至 2017 年 5 月 22 日的版本）定义的 148 个颜色名称，以及对应的 RGB、HSL、十六进制和三位十六进制简写形式（如果有的话）。

截至 2017 年年末，有几种新的颜色表示法正在研究中，例如 hwb()、gray()、lab()、lch() 等。由于缺少支持、前途不明，本表未列出这些颜色类型。

颜色名	RGB（十进制）	RGB（百分数）	HSL	十六进制
aliceblue	rgb(240,248,255)	rgb(94.1%,97.3%,100%)	hsl(208,100%,97.1%)	#F0F8FF
antiquewhite	rgb(250,235,215)	rgb(98%,92.2%,84.3%)	hsl(34,77.8%,91.2%)	#FAEBD7
aqua	rgb(0,255,255)	rgb(0%,100%,100%)	hsl(180,100%,50%)	#00FFFF /#0FF
aquamarine	rgb(127,255,212)	rgb(49.8%,100%,83.1%)	hsl(160,100%,74.9%)	#7FFFD4
azure	rgb(240,255,255)	rgb(94.1%,100%,100%)	hsl(180,100%,97.1%)	#F0FFFF
beige	rgb(245,245,220)	rgb(96.1%,96.1%,86.3%)	hsl(60,55.6%,91.2%)	#F5F5DC
bisque	rgb(255,228,196)	rgb(100%,89.4%,76.9%)	hsl(33,100%,88.4%)	#FFE4C4
black	rgb(0,0,0)	rgb(0%,0%,0%)	hsl(0,0%,0%)	#000000 /#000
blanchedal-mond	rgb(255,235,205)	rgb(100%,92.2%,80.4%)	hsl(36,100%,90.2%)	#FFEBCD
blue	rgb(0,0,255)	rgb(0%,0%,100%)	hsl(240,100%,50%)	#0000F F/#00F
blueviolet	rgb(138,43,226)	rgb(54.1%,16.9%,88.6%)	hsl(271,75.9%,52.7%)	#8A2BE2
brown	rgb(165,42,42)	rgb(64.7%,16.5%,16.5%)	hsl(0,59.4%,40.6%)	#A52A2A

颜色名	RGB（十进制）	RGB（百分数）	HSL	十六进制
burlywood	rgb(222,184,135)	rgb(87.1%,72.2%,52.9%)	hsl(34,56.9%,70%)	#DEB887
cadetblue	rgb(95,158,160)	rgb(37.3%,62%,62.7%)	hsl(182,25.5%,50%)	#5F9EA0
chartreuse	rgb(127,255,0)	rgb(49.8%,100%,0%)	hsl(90,100%,50%)	#7FFF00
chocolate	rgb(210,105,30)	rgb(82.4%,41.2%,11.8%)	hsl(25,75%,47.1%)	#D2691E
coral	rgb(255,127,80)	rgb(100%,49.8%,31.4%)	hsl(16,100%,65.7%)	#FF7F50
cornflowerblue	rgb(100,149,237)	rgb(39.2%,58.4%,92.9%)	hsl(219,79.2%,66.1%)	#6495ED
cornsilk	rgb(255,248,220)	rgb(100%,97.3%,86.3%)	hsl(48,100%,93.1%)	#FFF8DC
crimson	rgb(220,20,60)	rgb(86.3%,7.8%,23.5%)	hsl(348,83.3%,47.1%)	#DC143C
cyan	rgb(0,255,255)	rgb(0%,100%,100%)	hsl(180,100%,50%)	#00FFFF /#0FF
darkblue	rgb(0,0,139)	rgb(0%,0%,54.5%)	hsl(240,100%,27.3%)	#00008B
darkcyan	rgb(0,139,139)	rgb(0%,54.5%,54.5%)	hsl(180,100%,27.3%)	#008B8B
darkgoldenrod	rgb(184,134,11)	rgb(72.2%,52.5%,4.3%)	hsl(43,88.7%,38.2%)	#B8860B
darkgray	rgb(169,169,169)	rgb(66.3%,66.3%,66.3%)	hsl(0,0%,66.3%)	#A9A9A9
darkgreen	rgb(0,100,0)	rgb(0%,39.2%,0%)	hsl(120,100%,19.6%)	#006400
darkgrey	rgb(169,169,169)	rgb(66.3%,66.3%,66.3%)	hsl(0,0%,66.3%)	#A9A9A9
darkkhaki	rgb(189,183,107)	rgb(74.1%,71.8%,42%)	hsl(56,38.3%,58%)	#BDB76B
darkmagenta	rgb(139,0,139)	rgb(54.5%,0%,54.5%)	hsl(300,100%,27.3%)	#8B008B
darkolive-green	rgb(85,107,47)	rgb(33.3%,42%,18.4%)	hsl(82,39%,30.2%)	#556B2F
darkorange	rgb(255,140,0)	rgb(100%,54.9%,0%)	hsl(33,100%,50%)	#FF8C00
darkorchid	rgb(153,50,204)	rgb(60%,19.6%,80%)	hsl(280,60.6%,49.8%)	#9932CC
darkred	rgb(139,0,0)	rgb(54.5%,0%,0%)	hsl(0,100%,27.3%)	#8B0000
darksalmon	rgb(233,150,122)	rgb(91.4%,58.8%,47.8%)	hsl(15,71.6%,69.6%)	#E9967A
darkseagreen	rgb(143,188,143)	rgb(56.1%,73.7%,56.1%)	hsl(120,25.1%,64.9%)	#8FBC8F
darkslateblue	rgb(72,61,139)	rgb(28.2%,23.9%,54.5%)	hsl(248,39%,39.2%)	#483D8B
darkslategray	rgb(47,79,79)	rgb(18.4%,31%,31%)	hsl(180,25.4%,24.7%)	#2F4F4F
darkslategrey	rgb(47,79,79)	rgb(18.4%,31%,31%)	hsl(180,25.4%,24.7%)	#2F4F4F
darkturquoise	rgb(0,206,209)	rgb(0%,80.8%,82%)	hsl(181,100%,41%)	#00CED1
darkviolet	rgb(148,0,211)	rgb(58%,0%,82.7%)	hsl(282,100%,41.4%)	#9400D3
deeppink	rgb(255,20,147)	rgb(100%,7.8%,57.6%)	hsl(328,100%,53.9%)	#FF1493
deepskyblue	rgb(0,191,255)	rgb(0%,74.9%,100%)	hsl(195,100%,50%)	#00BFFF
dimgray	rgb(105,105,105)	rgb(41.2%,41.2%,41.2%)	hsl(0,0%,41.2%)	#696969

颜色名	RGB（十进制）	RGB（百分数）	HSL	十六进制
dimgrey	rgb(105,105,105)	rgb(41.2%,41.2%,41.2%)	hsl(0,0%,41.2%)	#696969
dodgerblue	rgb(30,144,255)	rgb(11.8%,56.5%,100%)	hsl(210,100%,55.9%)	#1E90FF
firebrick	rgb(178,34,34)	rgb(69.8%,13.3%,13.3%)	hsl(0,67.9%,41.6%)	#B22222
floralwhite	rgb(255,250,240)	rgb(100%,98%,94.1%)	hsl(40,100%,97.1%)	#FFFAF0
forestgreen	rgb(34,139,34)	rgb(13.3%,54.5%,13.3%)	hsl(120,60.7%,33.9%)	#228B22
fuchsia	rgb(255,0,255)	rgb(100%,0%,100%)	hsl(300,100%,50%)	#FF00FF /#F0F
gainsboro	rgb(220,220,220)	rgb(86.3%,86.3%,86.3%)	hsl(0,0%,86.3%)	#DCDCDC
ghostwhite	rgb(248,248,255)	rgb(97.3%,97.3%,100%)	hsl(240,100%,98.6%)	#F8F8FF
gold	rgb(255,215,0)	rgb(100%,84.3%,0%)	hsl(51,100%,50%)	#FFD700
goldenrod	rgb(218,165,32)	rgb(85.5%,64.7%,12.5%)	hsl(43,74.4%,49%)	#DAA520
gray	rgb(128,128,128)	rgb(50.2%,50.2%,50.2%)	hsl(0,0%,50.2%)	#808080
green	rgb(0,128,0)	rgb(0%,50.2%,0%)	hsl(120,100%,25.1%)	#008000
greenyellow	rgb(173,255,47)	rgb(67.8%,100%,18.4%)	hsl(84,100%,59.2%)	#ADFF2F
grey	rgb(128,128,128)	rgb(50.2%,50.2%,50.2%)	hsl(0,0%,50.2%)	#808080
honeydew	rgb(240,255,240)	rgb(94.1%,100%,94.1%)	hsl(120,100%,97.1%)	#F0FFF0
hotpink	rgb(255,105,180)	rgb(100%,41.2%,70.6%)	hsl(330,100%,70.6%)	#FF69B4
indianred	rgb(205,92,92)	rgb(80.4%,36.1%,36.1%)	hsl(0,53.1%,58.2%)	#CD5C5C
indigo	rgb(75,0,130)	rgb(29.4%,0%,51%)	hsl(275,100%,25.5%)	#4B0082
ivory	rgb(255,255,240)	rgb(100%,100%,94.1%)	hsl(60,100%,97.1%)	#FFFFF0
khaki	rgb(240,230,140)	rgb(94.1%,90.2%,54.9%)	hsl(54,76.9%,74.5%)	#F0E68C
lavender	rgb(230,230,250)	rgb(90.2%,90.2%,98%)	hsl(240,66.7%,94.1%)	#E6E6FA
lavenderblush	rgb(255,240,245)	rgb(100%,94.1%,96.1%)	hsl(340,100%,97.1%)	#FFF0F5
lawngreen	rgb(124,252,0)	rgb(48.6%,98.8%,0%)	hsl(90,100%,49.4%)	#7CFC00
lemonchiffon	rgb(255,250,205)	rgb(100%,98%,80.4%)	hsl(54,100%,90.2%)	#FFFACD
lightblue	rgb(173,216,230)	rgb(67.8%,84.7%,90.2%)	hsl(195,53.3%,79%)	#ADD8E6
lightcoral	rgb(240,128,128)	rgb(94.1%,50.2%,50.2%)	hsl(0,78.9%,72.2%)	#F08080
lightcyan	rgb(224,255,255)	rgb(87.8%,100%,100%)	hsl(180,100%,93.9%)	#E0FFFF
lightgoldenrodyellow	rgb(250,250,210)	rgb(98%,98%,82.4%)	hsl(60,80%,90.2%)	#FAFAD2
lightgray	rgb(211,211,211)	rgb(82.7%,82.7%,82.7%)	hsl(0,0%,82.7%)	#D3D3D3
lightgreen	rgb(144,238,144)	rgb(56.5%,93.3%,56.5%)	hsl(120,73.4%,74.9%)	#90EE90
lightgrey	rgb(211,211,211)	rgb(82.7%,82.7%,82.7%)	hsl(0,0%,82.7%)	#D3D3D3

颜色名	RGB（十进制）	RGB（百分数）	HSL	十六进制
lightpink	rgb(255,182,193)	rgb(100%,71.4%,75.7%)	hsl(351,100%,85.7%)	#FFB6C1
lightsalmon	rgb(255,160,122)	rgb(100%,62.7%,47.8%)	hsl(17,100%,73.9%)	#FFA07A
lightseagreen	rgb(32,178,170)	rgb(12.5%,69.8%,66.7%)	hsl(177,69.5%,41.2%)	#20B2AA
lightskyblue	rgb(135,206,250)	rgb(52.9%,80.8%,98%)	hsl(203,92%,75.5%)	#87CEFA
lightslate-gray	rgb(119,136,153)	rgb(46.7%,53.3%,60%)	hsl(210,14.3%,53.3%)	#778899 /#789
lightslate-grey	rgb(119,136,153)	rgb(46.7%,53.3%,60%)	hsl(210,14.3%,53.3%)	#778899 /#789
lightsteel-blue	rgb(176,196,222)	rgb(69%,76.9%,87.1%)	hsl(214,41.1%,78%)	#B0C4DE
lightyellow	rgb(255,255,224)	rgb(100%,100%,87.8%)	hsl(60,100%,93.9%)	#FFFFE0
lime	rgb(0,255,0)	rgb(0%,100%,0%)	hsl(120,100%,50%)	#00FF00 /#0F0
limegreen	rgb(50,205,50)	rgb(19.6%,80.4%,19.6%)	hsl(120,60.8%,50%)	#32CD32
linen	rgb(250,240,230)	rgb(98%,94.1%,90.2%)	hsl(30,66.7%,94.1%)	#FAF0E6
magenta	rgb(255,0,255)	rgb(100%,0%,100%)	hsl(300,100%,50%)	#FF00FF /#F0F
maroon	rgb(128,0,0)	rgb(50.2%,0%,0%)	hsl(0,100%,25.1%)	#800000
mediumaqua-marine	rgb(102,205,170)	rgb(40%,80.4%,66.7%)	hsl(160,50.7%,60.2%)	#66CDAA
mediumblue	rgb(0,0,205)	rgb(0%,0%,80.4%)	hsl(240,100%,40.2%)	#0000CD
mediumorchid	rgb(186,85,211)	rgb(72.9%,33.3%,82.7%)	hsl(288,58.9%,58%)	#BA55D3
mediumpurple	rgb(147,112,219)	rgb(57.6%,43.9%,85.9%)	hsl(260,59.8%,64.9%)	#9370DB
mediumsea-green	rgb(60,179,113)	rgb(23.5%,70.2%,44.3%)	hsl(147,49.8%,46.9%)	#3CB371
mediumslate-blue	rgb(123,104,238)	rgb(48.2%,40.8%,93.3%)	hsl(249,79.8%,67.1%)	#7B68EE
mediumspring-green	rgb(0,250,154)	rgb(0%,98%,60.4%)	hsl(157,100%,49%)	#00FA9A
mediumtur-quoise	rgb(72,209,204)	rgb(28.2%,82%,80%)	hsl(178,59.8%,55.1%)	#48D1CC
mediumviole-#C71585 tred	rgb(199,21,133)	rgb(78%,8.2%,52.2%)	hsl(322,80.9%,43.1%)	
midnightblue	rgb(25,25,112)	rgb(9.8%,9.8%,43.9%)	hsl(240,63.5%,26.9%)	#191970
mintcream	rgb(245,255,250)	rgb(96.1%,100%,98%)	hsl(150,100%,98%)	#F5FFFA

颜色名	RGB（十进制）	RGB（百分数）	HSL	十六进制
mistyrose	rgb(255,228,225)	rgb(100%,89.4%,88.2%)	hsl(6,100%,94.1%)	#FFE4E1
moccasin	rgb(255,228,181)	rgb(100%,89.4%,71%)	hsl(38,100%,85.5%)	#FFE4B5
navajowhite	rgb(255,222,173)	rgb(100%,87.1%,67.8%)	hsl(36,100%,83.9%)	#FFDEAD
navy	rgb(0,0,128)	rgb(0%,0%,50.2%)	hsl(240,100%,25.1%)	#000080
oldlace	rgb(253,245,230)	rgb(99.2%,96.1%,90.2%)	hsl(39,85.2%,94.7%)	#FDF5E6
olive	rgb(128,128,0)	rgb(50.2%,50.2%,0%)	hsl(60,100%,25.1%)	#808000
olivedrab	rgb(107,142,35)	rgb(42%,55.7%,13.7%)	hsl(80,60.5%,34.7%)	#6B8E23
orange	rgb(255,165,0)	rgb(100%,64.7%,0%)	hsl(39,100%,50%)	#FFA500
orangered	rgb(255,69,0)	rgb(100%,27.1%,0%)	hsl(16,100%,50%)	#FF4500
orchid	rgb(218,112,214)	rgb(85.5%,43.9%,83.9%)	hsl(302,58.9%,64.7%)	#DA70D6
palegoldenrod	rgb(238,232,170)	rgb(93.3%,91%,66.7%)	hsl(55,66.7%,80%)	#EEE8AA
palegreen	rgb(152,251,152)	rgb(59.6%,98.4%,59.6%)	hsl(120,92.5%,79%)	#98FB98
paleturquoise	rgb(175,238,238)	rgb(68.6%,93.3%,93.3%)	hsl(180,64.9%,81%)	#AFEEEE
palevioletred	rgb(219,112,147)	rgb(85.9%,43.9%,57.6%)	hsl(340,59.8%,64.9%)	#DB7093
papayawhip	rgb(255,239,213)	rgb(100%,93.7%,83.5%)	hsl(37,100%,91.8%)	#FFEFD5
peachpuff	rgb(255,218,185)	rgb(100%,85.5%,72.5%)	hsl(28,100%,86.3%)	#FFDAB9
peru	rgb(205,133,63)	rgb(80.4%,52.2%,24.7%)	hsl(30,58.7%,52.5%)	#CD853F
pink	rgb(255,192,203)	rgb(100%,75.3%,79.6%)	hsl(350,100%,87.6%)	#FFC0CB
plum	rgb(221,160,221)	rgb(86.7%,62.7%,86.7%)	hsl(300,47.3%,74.7%)	#DDA0DD
powderblue	rgb(176,224,230)	rgb(69%,87.8%,90.2%)	hsl(187,51.9%,79.6%)	#B0E0E6
purple	rgb(128,0,128)	rgb(50.2%,0%,50.2%)	hsl(300,100%,25.1%)	#800080
rebeccapurple	rgb(102,51,153)	rgb(40%,20%,60%)	hsl(270,50%,40%)	#663399 /#639
red	rgb(255,0,0)	rgb(100%,0%,0%)	hsl(0,100%,50%)	#FF0000 /#F00
rosybrown	rgb(188,143,143)	rgb(73.7%,56.1%,56.1%)	hsl(0,25.1%,64.9%)	#BC8F8F
royalblue	rgb(65,105,225)	rgb(25.5%,41.2%,88.2%)	hsl(225,72.7%,56.9%)	#4169E1
saddlebrown	rgb(139,69,19)	rgb(54.5%,27.1%,7.5%)	hsl(25,75.9%,31%)	#8B4513
salmon	rgb(250,128,114)	rgb(98%,50.2%,44.7%)	hsl(6,93.2%,71.4%)	#FA8072
sandybrown	rgb(244,164,96)	rgb(95.7%,64.3%,37.6%)	hsl(28,87.1%,66.7%)	#F4A460
seagreen	rgb(46,139,87)	rgb(18%,54.5%,34.1%)	hsl(146,50.3%,36.3%)	#2E8B57
seashell	rgb(255,245,238)	rgb(100%,96.1%,93.3%)	hsl(25,100%,96.7%)	#FFF5EE
sienna	rgb(160,82,45)	rgb(62.7%,32.2%,17.6%)	hsl(19,56.1%,40.2%)	#A0522D

颜色名	RGB（十进制）	RGB（百分数）	HSL	十六进制
silver	rgb(192,192,192)	rgb(75.3%,75.3%,75.3%)	hsl(0,0%,75.3%)	#C0C0C0
skyblue	rgb(135,206,235)	rgb(52.9%,80.8%,92.2%)	hsl(197,71.4%,72.5%)	#87CEEB
slateblue	rgb(106,90,205)	rgb(41.6%,35.3%,80.4%)	hsl(248,53.5%,57.8%)	#6A5ACD
slategray	rgb(112,128,144)	rgb(43.9%,50.2%,56.5%)	hsl(210,12.6%,50.2%)	#708090
slategrey	rgb(112,128,144)	rgb(43.9%,50.2%,56.5%)	hsl(210,12.6%,50.2%)	#708090
snow	rgb(255,250,250)	rgb(100%,98%,98%)	hsl(0,100%,99%)	#FFFAFA
springgreen	rgb(0,255,127)	rgb(0%,100%,49.8%)	hsl(150,100%,50%)	#00FF7F
steelblue	rgb(70,130,180)	rgb(27.5%,51%,70.6%)	hsl(207,44%,49%)	#4682B4
tan	rgb(210,180,140)	rgb(82.4%,70.6%,54.9%)	hsl(34,43.8%,68.6%)	#D2B48C
teal	rgb(0,128,128)	rgb(0%,50.2%,50.2%)	hsl(180,100%,25.1%)	#008080
thistle	rgb(216,191,216)	rgb(84.7%,74.9%,84.7%)	hsl(300,24.3%,79.8%)	#D8BFD8
tomato	rgb(255,99,71)	rgb(100%,38.8%,27.8%)	hsl(9,100%,63.9%)	#FF6347
turquoise	rgb(64,224,208)	rgb(25.1%,87.8%,81.6%)	hsl(174,72.1%,56.5%)	#40E0D0
violet	rgb(238,130,238)	rgb(93.3%,51%,93.3%)	hsl(300,76.1%,72.2%)	#EE82EE
wheat	rgb(245,222,179)	rgb(96.1%,87.1%,70.2%)	hsl(39,76.7%,83.1%)	#F5DEB3
white	rgb(255,255,255)	rgb(100%,100%,100%)	hsl(0,0%,100%)	#FFFFFF /#FFF
whitesmoke	rgb(245,245,245)	rgb(96.1%,96.1%,96.1%)	hsl(0,0%,96.1%)	#F5F5F5
yellow	rgb(255,255,0)	rgb(100%,100%,0%)	hsl(60,100%,50%)	#FFFF00 /#FF0
yellowgreen	rgb(154,205,50)	rgb(60.4%,80.4%,19.6%)	hsl(80,60.8%,50%)	#9ACD32

康的项目，然后又开始涉足网站开发。千年虫来袭时，她已经成为 Web 标准化专家，维护着 http://www.standardista.com 网站。

如今，她的技术博客吸引了数百万计的访客。她经常在世界各地举办的会议上演讲，话题涉及 CSS3、HTML5、JavaScript、可访问性和移动 Web 开发。除了与公众读者分享只有内行才懂的编程趣闻之外，Estelle 还为 Kodak Gallery、SurveyMonkey、Visa、三星、雅虎和苹果等公司做过咨询工作。她目前是 Instart Logic（一个帮助开发者以快速、安全的方式分发 Web 应用的平台）的开放 Web 布道师。

不编程时，她喜欢施工，改造自 20 世纪 60 年代承袭下来的家，力求把那个年代遗留下来的嬉皮士痕迹彻底清除。这也算是 Estelle 迎接 21 世纪的另一种方式。

封面介绍

本书封面上的动物是鲑鱼（学名 salmonidae），这是一个鱼科的名称，包括多个种。最常见的鲑鱼有两种，太平洋鲑鱼和大西洋鲑鱼。

太平洋鲑鱼生活在北美和亚洲沿岸的北太平洋。太平洋鲑鱼有五个亚种，平均体重为10~30 磅。秋天，太平洋鲑鱼在淡水溪流砾石层中产卵，在漫长的冬季孵化，一直长到一英寸长。幼鱼在溪流或湖泊中生活一两年，然后顺流直下，进入海洋。鲑鱼在大洋中生活几年之后，回游到出生地，产卵后死去。

大西洋鲑鱼生活在北美和欧洲沿岸的北大西洋。大西洋鲑鱼有许多亚种，包括鳟鱼和鳕鱼。这种鲑鱼平均体重 10~20 磅。大西洋鲑鱼的生命周期与太平洋鲑鱼相似，也从淡水砾石层迁徙到海洋。然而，两者之间的一个主要区别是，大西洋鲑鱼产卵后不死亡，而是返回海洋，然后再回到溪流产卵，通常可以持续两三次。

一般来说，鲑鱼体态优美，呈银色、背部和鳍上有斑点。它们的食物包括浮游生物、昆虫幼虫、虾和小鱼。它们的嗅觉异常敏锐，从海洋逆流而上，历经险阻游回出生地靠的就是嗅觉。有些种类的鲑鱼生活在内陆的淡水中。

鲑鱼是生态系统的重要组成部分，因为腐烂后的身体能为河床提供肥料。然而，近些年来，鲑鱼的数量不断减少。鲑鱼数量下降的原因包括栖息地破坏、捕鱼、阻塞产卵路径的水坝、酸雨、干旱、洪水和污染。

封面图片是一幅 19 世纪的版画，摘自 Dover Pictorial Archive。

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。

ISBN 978-7-5198-2659-8

9 787519 826598

上下册：198.00元

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)