Broadview
www.broadview.com.cn

精通D3.js

• 吕之华 著 •

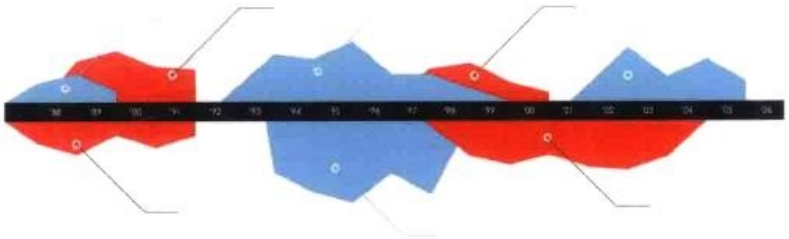
交互式数据可视化高级编程

大数据时代需要可视化工具，
D3是世界最流行的可视化函数库。

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版



大数据时代需要可视化工具，D3是世界最流行的可视化函数库。本书手把手教你学会D3，从零讲起直到高级应用，既是教程，又可作为参考手册，查阅D3各种方法的用法。内容图文并茂，示例丰富，帮助你轻松生成各种漂亮图形。

——阮一峰

• 内容简介 •

本书以当前流行的数据可视化技术D3.js为主要内容，分为三大部分，共计13章。第一部分讲述基础知识，第二部分学习制作各种常见图表，第三部分讲解交互式图表及地图的进阶应用。本书是一个相对完整的D3.js教程，讲解此技术所有重要的知识点，既有基础入门知识，又有相对深入的内容。笔者秉持以下原则：由易到难，循序渐进，图文并茂，清晰易懂。

书中源代码下载网址：

http://www.ourd3js.com/
http://broadview.com.cn/26776

上架建议：计算机 > Web前端

博文视点Broadview

@博文视点Broadview

责任编辑：付　睿
封面设计：侯士卿

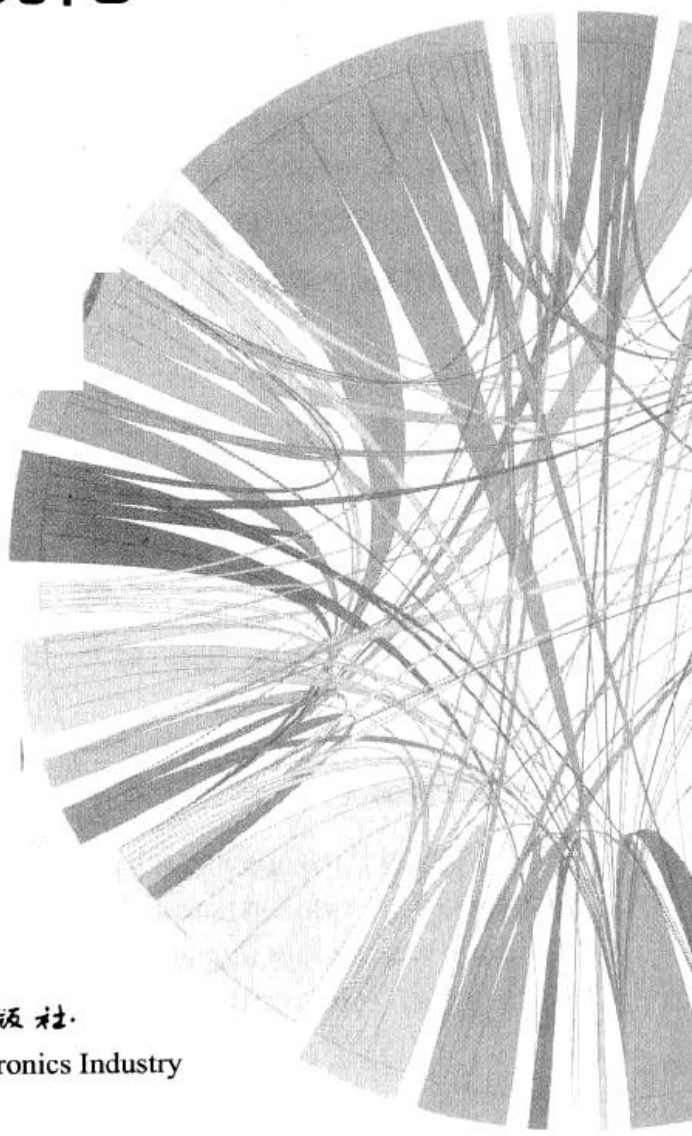
ISBN 978-7-121-26776-5

定价：79.00元

精通D3.js

吕之华 著

交互式数据可视化高级编程



電子工業出版社
Publishing House of Electronics Industry

内 容 简 介

本书以当前流行的数据可视化技术 D3.js 为主要内容，分为三大部分，共计 13 章。第一部分讲述基础知识，第二部分学习制作各种常见图表，第三部分讲解交互式图表及地图的进阶应用。本书作为一个相对完整的 D3.js 教程，讲解此技术所有重要的知识点，既有基础入门知识，又有相对深入的内容。笔者秉持以下原则：由易到难，循序渐进，图文并茂，清晰易懂。

本书适合有一定计算机基础的读者，需要熟悉 C、C++、Java、JavaScript 等至少一门编程语言，能够理解基础的数据结构和算法。

图书在版编目（CIP）数据

精通 D3.js：交互式数据可视化高级编程/吕之华著. —北京：电子工业出版社，2015.8
ISBN 978-7-121-26776-5

Ⅰ. ①精… Ⅱ. ①吕… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2015）第 169494 号

责任编辑：付睿
印　　刷：北京丰源印刷厂
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：25.25　字数：562 千字　彩插：9
版　　次：2015 年 8 月第 1 版
印　　次：2015 年 8 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

目　录

第 1 章

D3 简介

本章内容包括：

D3 是什么

- 数据可视化是什么
- 常见可视化图表的种类
- 学习 D3 的方法

笔者第一次接触到 D3，首先是被绚丽多彩的图表吸引，然后陶醉于这些图表的可交互特性，而且动画流畅简洁、赏心悦目。近年来，可视化越来越流行，许多报刊杂志、门户网站、新闻媒体都大量使用可视化技术，使得复杂的数据和文字变得十分容易理解，有一句谚语“一张图片价值相当于一千个字”，的确是名副其实。对于信息爆炸式增长的今天，我们没有时间一条一条地阅读，希望的是一眼就能找到自己想要的信息。图片不仅容易理解，而且容易记忆，是值得推广的信息传达方式。

1.1 D3 是什么

D3 的全称是 Data-Driven Documents，直译为“数据驱动的文档”。听名字有点抽象，简单概括为一句话：

D3 是一个 JavaScript 的函数库，是用来做数据可视化的。

文档指 DOM，即文档对象模型（Document Object Model）。D3 允许用户绑定任意数据到

DOM，然后根据数据来操作文档，创建可交互式的图表。

JavaScript 文件的后缀名通常为.js，故 D3 也常称为 D3.js。D3 提供了各种简单易用的函数，大大简化了 JavaScript 操作数据生成图表的难度。由于它本质上是 JavaScript，所以用 JavaScript 也是可以实现所有功能的，但 D3 能大大减轻你的工作量，尤其是在数据可视化方面，D3 已经将生成可视化的复杂步骤精简到了几个简单的函数，你只需要输入几个简单的数据，就能够转换为各种绚丽的图形。有过 JavaScript 基础的朋友一定很容易理解它。

1.1.1 D3 简史

大数据时代蓬勃发展的今天，每天都有惊人的数据产生，怎么提取并显示有用的信息，变得越来越重要。在 Web 浏览器中进行可视化也成为迫切的需求，有许多项目以此为目标。2009 年，Mike Bostock、Jeff Heer、Vadim Ogievetsky 共同开发了 Protovis，可以算是 D3 的前身。2011 年，他们停止了 Protovis，使用 JavaScript 开发了一个新的项目，这就是 D3。

JavaScript 诞生于 1995 年，经过近 20 年的发展，如今已经成为 Web 浏览器上事实上的标准语言，其应用范围扩展到服务器端、移动领域等，并且越来越完善。因此，D3 采用 JavaScript 作为开发语言。

2011 年 2 月 18 日，Mike Bostock 发布了 v1.0 版本。

2011 年 8 月 25 日，v2.0 版本发布，功能大幅增强，应用逐渐增多。

2012 年 12 月 22 日，v3.0 版本发布，修复了大量 bug，功能更加稳健。

笔者写作时，最新版本为 v3.4.13。v3.0 版之后差别不大，本书以 v3.x 版为标准。

1.1.2 D3 的优势

可视化的库有很多，基于 JavaScript 开发的库也有很多，D3 有什么优势呢？

1．数据能够与 DOM 绑定在一起

D3 能够将数据与 DOM 绑定在一起，使得数据与图形成为一个整体，即图形中有数据、数据中有图形。那么在生成图形或更改图形时，就可以方便地根据数据进行操作。并且，当数据更改之后，图形的更新也会很方便。

2．数据转换和绘制是独立的

将数据变成图表，需要不少数学算法，一些可视化库的做法是：提供一个函数 drawPie()，输入数据，直接绘制出饼状图。

D3 的做法是：提供一个函数 computePie()，可将数据转换成饼状图的数据，然后开发者使用自己喜欢的方式来绘制饼状图。

看起来，好像 D3 使问题变麻烦了，但是在图表比较复杂的时候，直接绘制的饼状图往往达不到要求，细微的部分没有办法更改。将两者分开，就极大地提高了自由度，以至于开发者甚至可以使用其他的图形库来显示 D3 计算的数据。

3．代码简洁

JQuery 是网页开发中很常用的库，其链式语法被很多人喜爱。D3 也采用了这一语法，能够一个函数套一个函数，使得代码很简洁。

4．大量布局

饼状图、树形图、打包图、矩阵树图等，D3 将大量复杂的算法封装成一个一个“布局”，能够适用于各种图表的制作。

5．基于 SVG，缩放不会损失精度

SVG，是可缩放的矢量图形。D3 大部分是在 SVG 上绘制的，并且提供了大量的图形生成器，使得在 SVG 上绘制图形变得简单。另外，由于 SVG 是矢量图，所以放大缩小不会有精度损失。

1.1.3　D3 的适用范围

D3 开发的应用是显示在网页上的。因此，开发者需将数据置于服务器端，并在网页文件（HTML）中插入 D3 代码。用户通过浏览器请求此网页文件，就会看见开发者希望让用户看到的可视化内容。

Ben Fry 在他的著作《Visualizing Data》中将数据可视化的过程分为七个步骤。

（1）获取——Acquire

（2）分析——Parse

（3）过滤——Filter

（4）挖掘——Mine

（5）表现——Represent

（6）改善——Refine

（7）交互——Interact

前四步不属于 D3 的处理范围，更多的是处理后三步，即表现、改善、交互。

1.2　数据可视化是什么

数据可视化（Data Visualization）起源于 18 世纪，William Playfair 在出版的书籍《The Commercial and Political Atlas》中第一次使用了柱形图和折线图，当时是为了表示国家的进出口

量，在今天依然这么使用。19 世纪初，出版了《Statistical Breviary》一书，里面第一次使用了饼状图，这三种图形都是至今最常用的最著名的可视化图形。19 世纪中叶，数据可视化主要被用于军事用途，用来表示军队死亡原因、军队的分布图等。进入 20 世纪，数据可视化有了飞跃性的发展。1990 年，在人机界面学会上，作为信息可视化原型的技术被发表。1995 年，IEEE Information Visualization 正式创立，信息可视化作为独立的学科被正式确立。近年，随着大数据时代的到来，数据可视化作为大量数据的呈现方式，成为当前重要的课题。

1.2.1 目的

The main goal of data visualization is its ability to visualize data, communicating information clearly and effectively.

数据可视化的目的，是要对数据进行可视化处理，以使得能够明确地、有效地传递信息。

— Vitaly Friedman

比起枯燥乏味的数值，人类对于大小、位置、浓淡、颜色、形状等能够有更好、更快的认识。经过可视化之后的数据能够加深人对于数据的理解和记忆。

例如有以下的数据，请找出最大值：

[321, 564, 1391, 245, 641, 798, 871]

数据量比较小，用肉眼也能找出来，但更好的办法是将数据进行可视化处理，如图 1-1 所示。

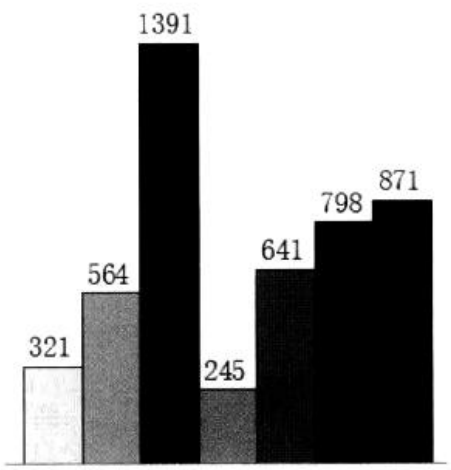


图 1-1　可视化后的柱形图

如图 1-1 所示，很明显，经过可视化之后，数据变得容易理解了。

1.2.2 构成要素

数据可视化的手法很多，其中有一些共通的视觉要素。

- **坐标**。数值的位置被对应到直角坐标系或极坐标系上，如图 1-2 所示。

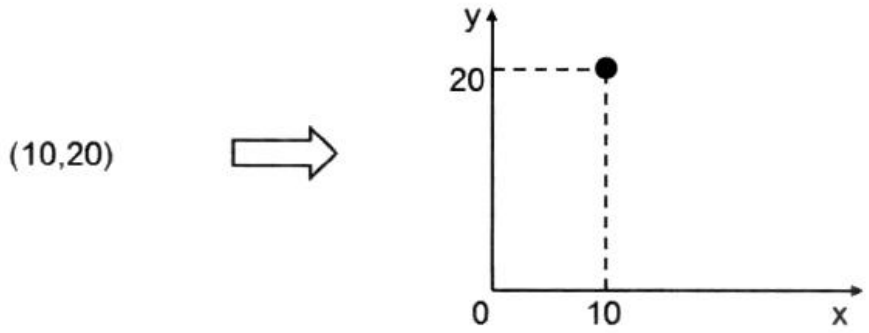


图 1-2　对应到坐标系上

- **大小**。数值的大小被对应到图形的大小上，如图 1-3 所示。

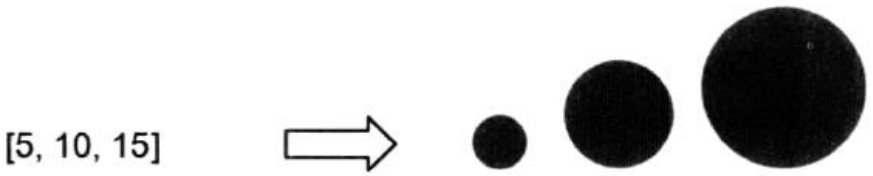


图 1-3　对应到大小上

- **色彩**。数值的分类和界限等对应到不同的颜色上，如图 1-4 所示。

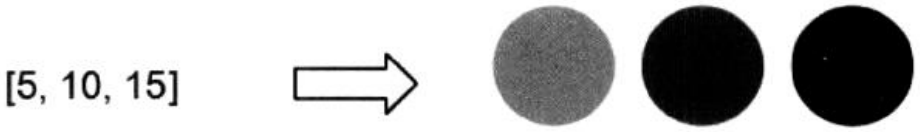


图 1-4　对应到颜色上

- **标签**。数值的特征用标签来标记，如图 1-5 所示。

图 1-5　对应到标签上

- **关联**。数值之间的联系，用关联线条等连接起来，如图 1-6 所示。

如图 1-2 至图 1-6 所示，列举了一些常见的将数据对应到视觉要素的方式。这些方式经过多数人的使用，是最容易被人理解也是最容易制作的，但是视觉要素并非局限于此。

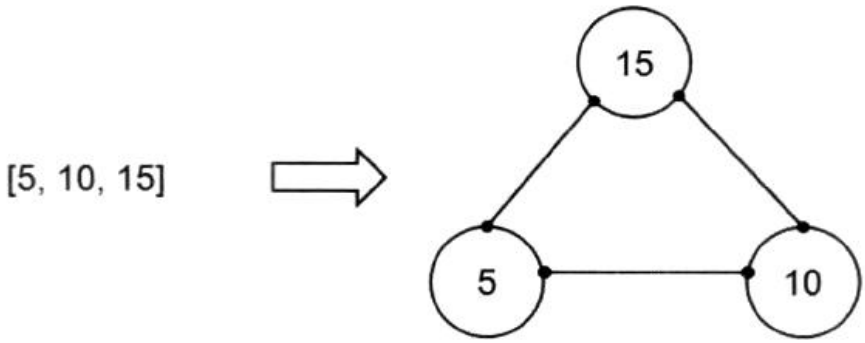


图 1-6　对应到关联线条上

1.2.3　相关概念

数据可视化（Data Visualization）和信息可视化（Information Visualization）很相近，有时几乎可以等同。但严格来说它们是不同的，它们的不同可以总结为一句话：

数据可视化是对数字信息进行可视化，信息可视化是对**数字信息**和**非数字信息**进行可视化。

1.3　图表种类

用于数据可视化的图表种类相当多，这里列举一些常用的图表。实际应用时可单独使用，也可以多种联动。

一般来说，图表要尽可能简单，能用简单的就用简单的。有的人可能会觉得简单的图表太古老、不大气，而追求复杂的图表，这反而有点本末倒置。数据可视化的目的，是要使数据明确地、有效地传递，而简单的图表是能够最快被人认可的。凭感觉自创图表也是可以的，但要注意此图表是否比原来的更简单易懂。

下面列举部分可视化图表。要注意，一种图表对应的汉语名称可能有多种，在本节中使用的名称将作为本书的标准。

1．柱形图

柱形图是最常见、最容易理解的图表，使用矩形的长短来表示数据的大小。数据类型一般是形如“时间—销售额”这样的二维数据集，图表要表现的是“随着时间的变化，销售额的变化情况”（如图 1-7 所示）。

如将图 1-7 的 x 轴和 y 轴替换，得到横向的柱形图，有时也称为条形图，但本质是一样的，都是用柱形的长短来表示数据的大小。此外，还可用矩形的宽窄来表示第三维的数据，使得一个图里的信息量更大，但会加深理解的难度。

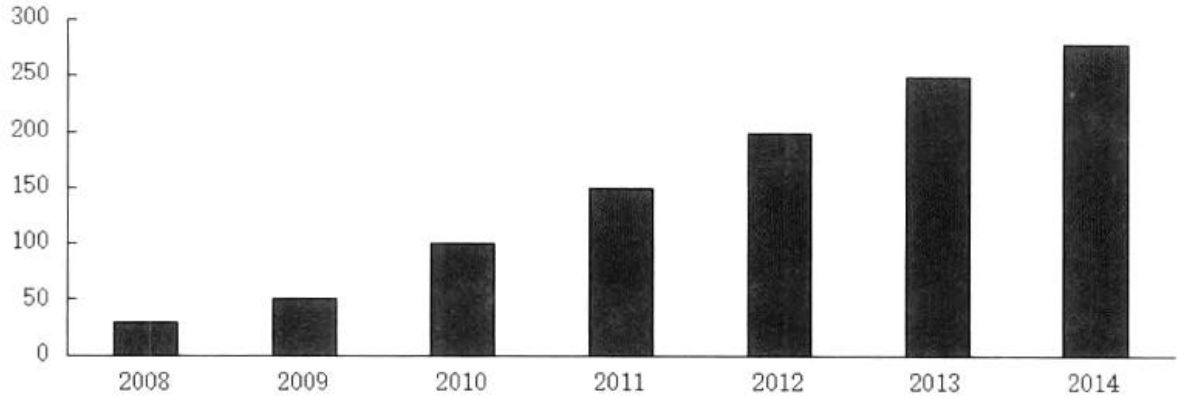


图 1-7　柱形图

2．散点图

散点图使用三维数据集，将其中的二维数据分别对应到 x 轴和 y 轴，再将第三维用点表示，而第三维数据是对应前二维的。其直观表现为在 x 轴和 y 轴的坐标系中分布着很多点，如图 1-8 所示。

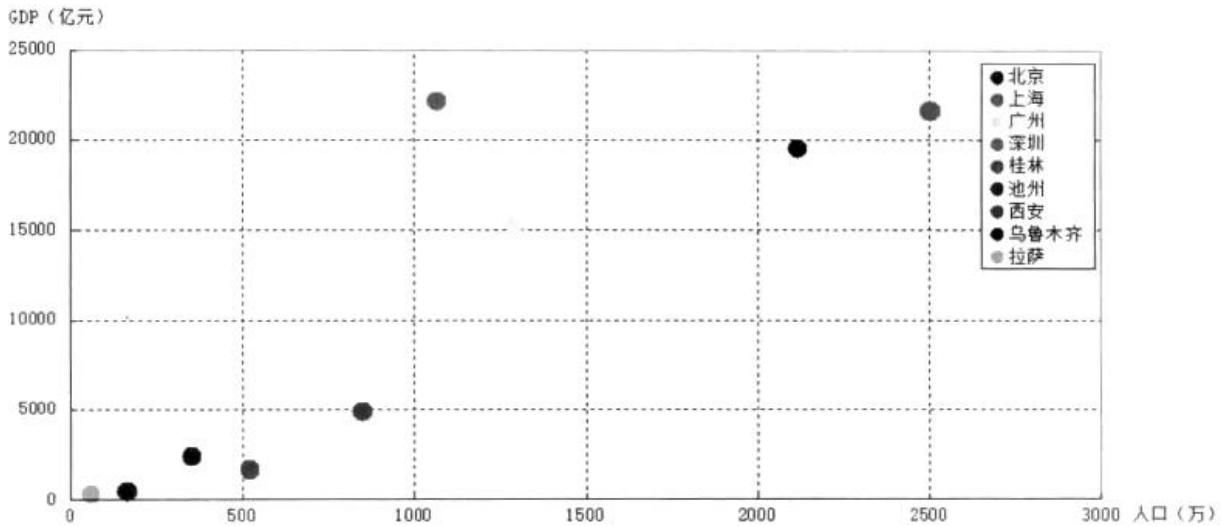


图 1-8　散点图

图 1-8 中 x 轴表示人口，y 轴表示 GDP，第三维数据为城市，所表示的内容为各城市在人口和 GDP 的二维坐标系中的分布。

3．折线图

折线图的目的与柱形图类似，也适合表示在二维数据集中，某一维相对于另一维的变化趋势，不同的是：

- 折线图较适合连续的数据，柱形图较适合离散的数据。

- 折线图较适合大量的数据，柱形图较适合少量的数据。
- 折线图用于表示多个数据集之间的比较时，效果较好。

图 1-9 表示的是中国和日本从 2005 年至 2013 年 GDP 的变化趋势。

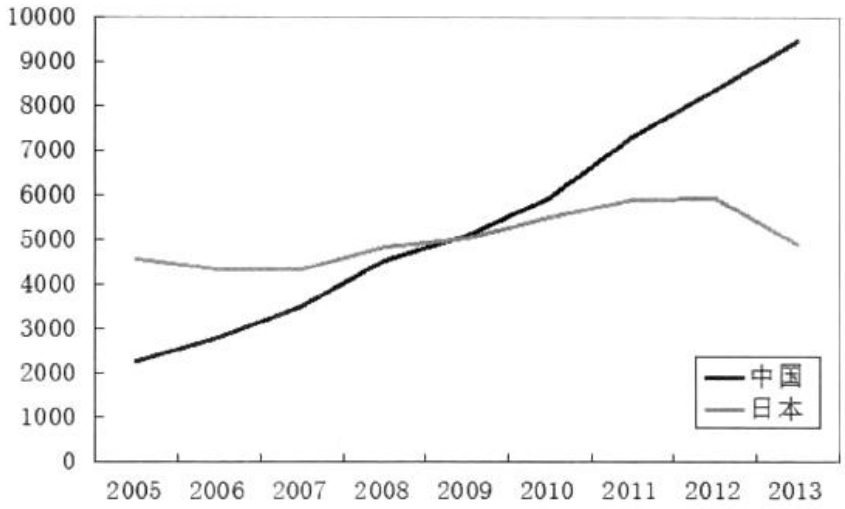


图 1-9　折线图

4．饼状图

饼状图可以用于比较数值的大小，但是有一个缺点：如果数值之间差距不大，肉眼很难分辨。因此，最好用于表示某一个值占全体值的百分比，比如 2014 年各浏览器占市场份额的百分比、各操作系统的份额百分比、各编程语言的使用比等。

如图 1-10 所示，饼状图的每一块都用标签表示出来，也可以用线连接到外部表示。另外，饼状图还有一些变种，如各扇形的半径不同，该半径可表示另一个数据量。

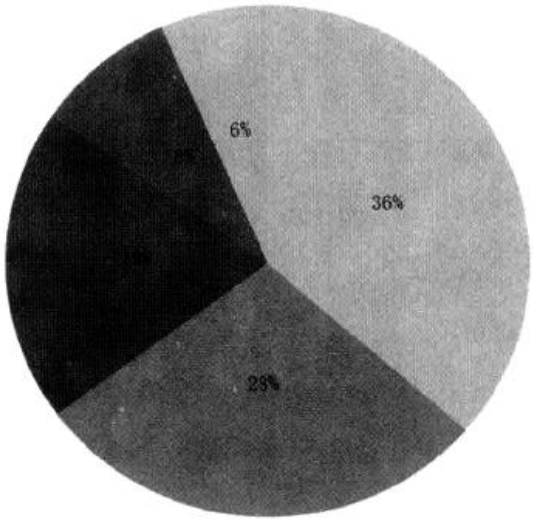


图 1-10　饼状图

5. 弦图

弦图，主要用于表示节点之间的联系。两点之间的连线表示哪两个节点具有联系，线的粗细表示权重。如图 1-11 所示，表示各城市的人口来自于哪些城市，后面章节会有详细介绍，这里只要有个印象即可。

6. 力导向图

力导向图适合描述大量顶点之间的关系，各顶点之间具有相互的作用力。如图 1-12 所示，各顶点之间用线相连，相连的顶点表示具有一定的关系。实际应用时可以赋予顶点和连线各种意义，如可做成人物关系图、力导向地图等，具有很大的扩展性。

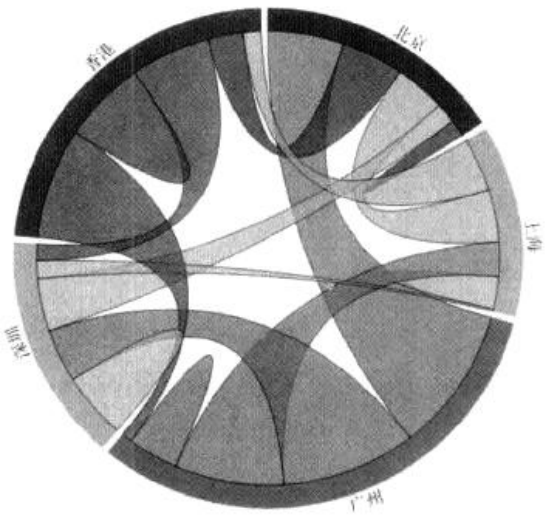


图 1-11　弦图

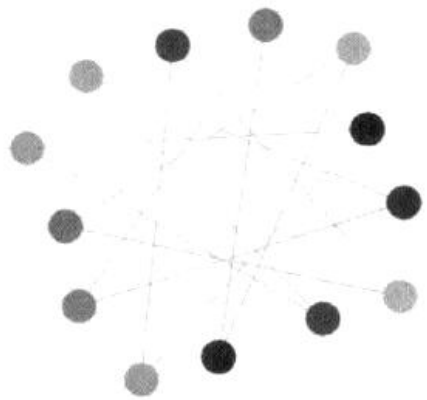


图 1-12　力导向图

7. 树状图

树状图用于表示层级、上下级、包含与被包含关系，与之类似的还有集群图。如图 1-13 所示，像树枝一样展现出省份和城市的包含关系。

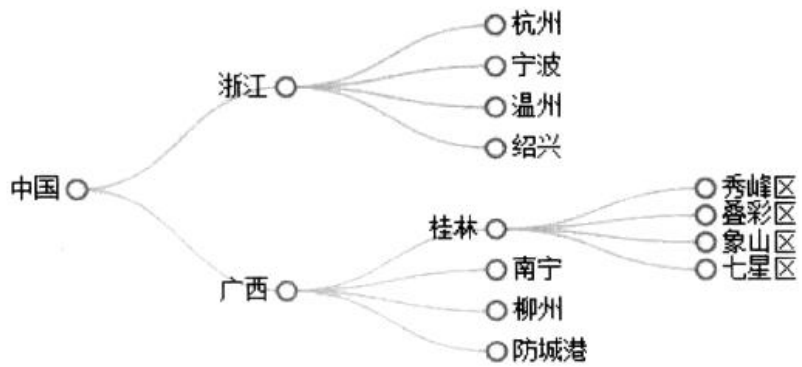


图 1-13　树状图

8. 打包图

打包图，用于表示包含与被包含的关系，也可表示各对象的权重。如图 1-14 所示，圆内套圆表示节点的关系，圆的大小表示节点的权重。

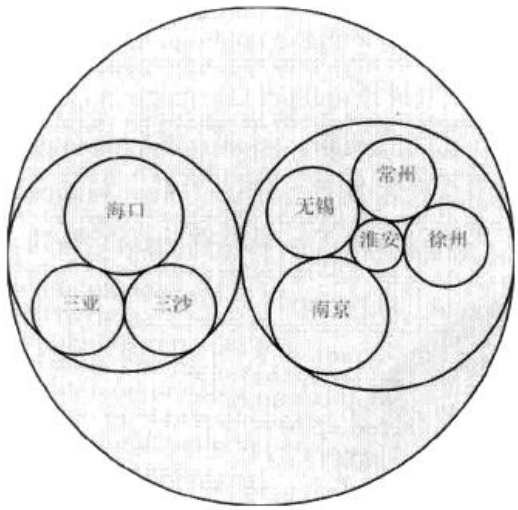


图 1-14　打包图

9. 分区图

分区图用于表示包含与被包含关系，其表现形式很像将硬盘分区，如图 1-15 和图 1-16 所示。

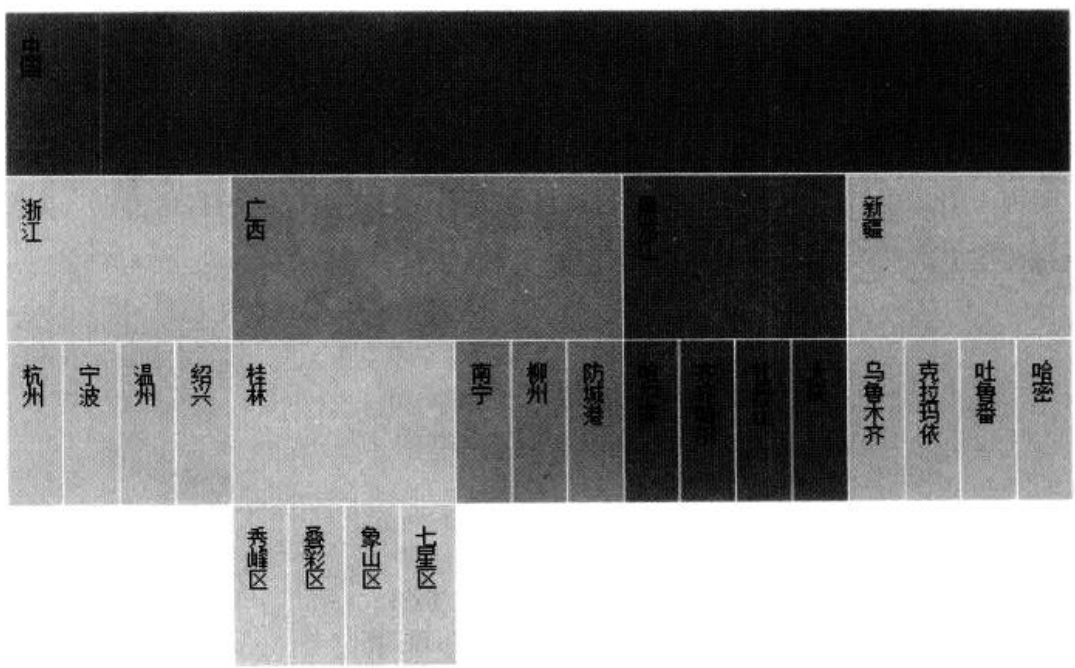


图 1-15　矩形分区图

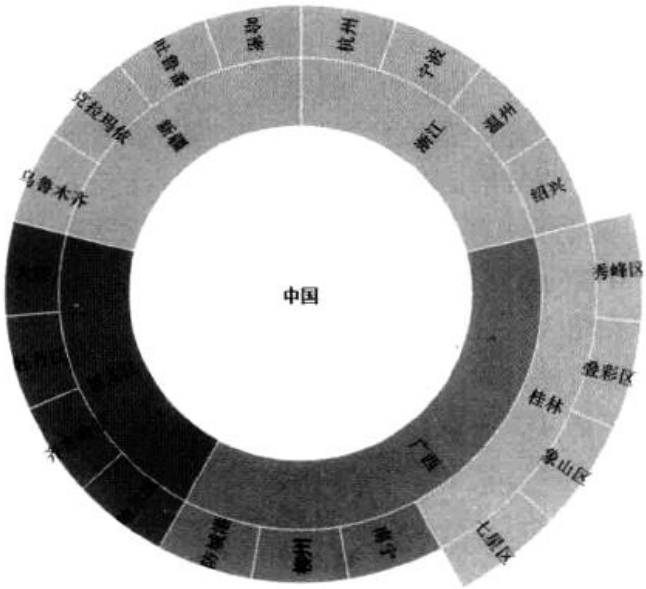


图 1-16　圆形分区图

1.4　学习方法

首先，不要做以下两件事。

- 看到某一个图表，感觉很好，立即复制下来，希望通过简单地更改数据或参数，就能达到自己使用的目的。
- 在完全不了解 JavaScript 的情况下学习 D3。D3 编程不一定用得到高深语法，但基础知识是必要的。

那么，D3 难学吗？

有不少人认为 D3 挺难学的，原因有三。

1．官方文档写得比较难

官网上提供了 API 文档，还有大量的例子。但是，大部分例子只有代码，没有文字说明。API 虽有说明，但是却没有太多针对性的例子，使初学者感觉头大。

2．不好理解数据转换和绘制分开的模式

一个函数，drawPie()，输入数据，输出绘制图形，一般人的思维模式是这样的。但是，D3 偏偏将两者分开了，分开之后能带来极大的自由度，但是也使得初学者理解它有些困难。

3．外语不好

对大部分国人来说，看英文文档还是挺头疼的，而中文资料相对不够丰富。

乍看上去，D3 有些难学，但是一旦掌握了，就能适应各种图表的制作，自由度大，功能极强。有人说，D3 就像是 Photoshop，其他的库就像是 Windows 画图板：前者需要一定的时间学习，学成后在图像处理上所向披靡；后者不需要学习时间，会和不会没有太大的价值。这么比喻可能有点夸张，笔者有一个更好的比喻（灵感来源于辜鸿铭先生的文章）。

D3 就像是写毛笔字，其他的可视化库就像是写钢笔字。钢笔字上手容易，下笔简单、快捷，写出来的东西叫作**文章**。毛笔字需要长期磨炼，上手较难，但是一旦掌握了，便能行云流水，心随念想，可进可退，只在笔尖，写出来的东西叫作**艺术**。

建议初学者从简单图表开始做起，尤其是柱形图、散点图、折线图这三种最基础的图表。通过反复练习基础图表，掌握 D3 各大功能模块的运用方法，把基础打好。如此，在制作复杂图表时才能起到事半功倍的效果。如果急于求成，一开始就着眼于“思维导图”这样的图表，会步履维艰，断不可为。

下面是一些学习 D3 的网站：

- http://d3js.org/

D3 的官方网站，含有 API 和大量示例。

- http://bost.ocks.org/mike/

D3 创始人制作的，有很多说明文档。

- https://www.dashingd3js.com/table-of-contents，

非常简单易懂的教程，文字解释、图片都十分清晰。此站开设的目的就是为了让人迅速而高效地掌握 D3。

- http://www.ourd3js.com/

笔者站点，有 D3 的一系列教程。

本书分为基础知识、常用图表的制作、深度应用三个部分。难度上由浅入深，建议依顺序阅读，在可以选择跳过的部分会有提示。

第 2 章

Web 前端开发基础

本章内容包括：

- 浏览器和服务器
- HTML 和 CSS 基础
- JavaScript 基础
- DOM 基础
- SVG 基础

本章简单介绍 Web 前端开发的基础，是学习 D3 的预备知识，主要针对没有前端基础的读者。有一定基础的读者可选择性阅读，或遇到不明问题时再查询即可。

第 1 节，介绍浏览器和服务器之间是如何进行交互的。理解了这一点，才知道要在什么地方使用 D3。

第 2 节，讲述 HTML 和 CSS 的基础知识。HTML 是用于描述网页内容的，CSS 是用于定义网页样式的，它们相互独立却常一起出现。

第 3 节，学习 JavaScript，一种直译式脚本语言，用于设定网页的行为。D3 就是基于 JavaScript 开发的。

第 4 节，介绍 DOM，即文档对象模型。它是针对结构化文档的一个接口，使用它可以动态地访问和修改 HTML 文档。

第 5 节，学习 SVG（可缩放矢量图形）。SVG 是一种画图板，绘制出来的是矢量图，D3 的图形大部分是绘制在 SVG 上的。

2.1 浏览器和服务器

浏览器（Browser）对我们来说太重要了，每天早上看新闻要用到，工作时查询资料要用到，看网络视频要用到，休闲娱乐也要用到。如果要评选每天使用得最多的软件，恐怕就是浏览器了。那么，当输入网址进入网站后，见到的图片文字是从哪来的，怎么传输过来的，这些信息是存储在哪里的，什么人准备的呢？

如图 2-1 所示，管理员在服务器（Server）存储各种信息，包括主页、图片、音乐、视频等，服务器与因特网（Internet）相连。很多与因特网相连的用户，通过个人电脑中的浏览器，发送请求给服务器，服务器就将其需要的信息（文字图片等）传送给用户，用户就能够在浏览器上享受到各种服务。

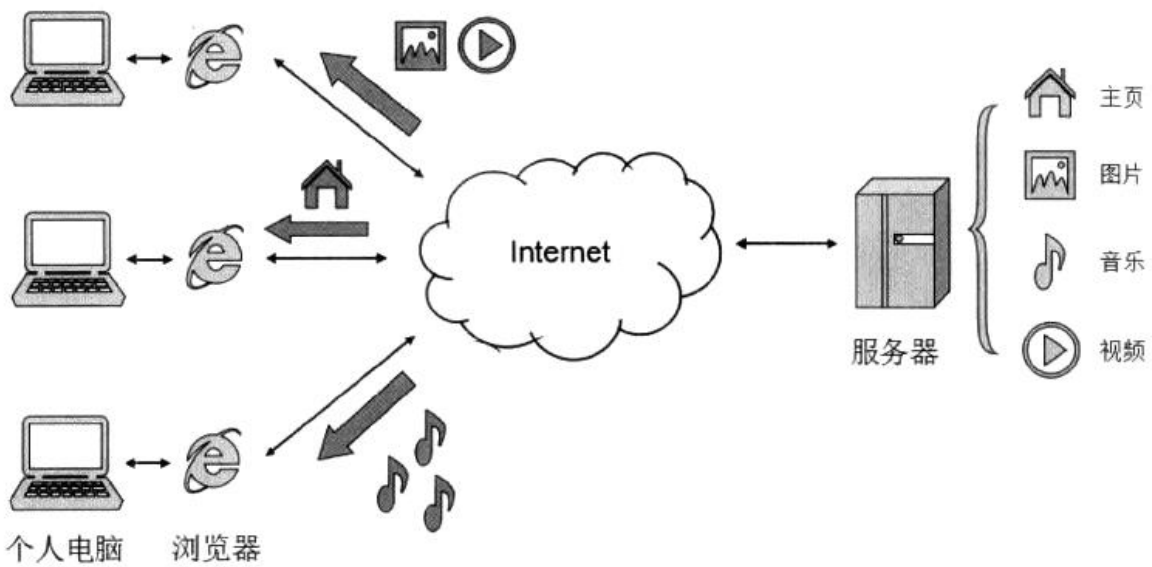


图 2-1 浏览器与服务器的交互

当然，浏览器与服务器的交互实际上要复杂得多，但理解本书的内容不需要知道那么多细节，有上述内容已经足够。

2.1.1 浏览器

浏览器软件有很多，常见的有：

（1）Internet Explorer（简称 IE）是最常见的浏览器，由 Windows 系统自带，发行于 1995 年，使用人数最多。目前的最新版本为 IE 12，还在使用的最古老的版本可能是 IE 6。IE 系列占浏览器市场份额的一半以上，远超其同行。制作网页时必须要考虑到用户所使用的浏览器版本，

有些遗憾的是，D3 对 IE 8 及以下版本支持不好。有人为使 D3 兼容 IE 8 以下版本，做了各种努力，效果却差强人意。不过，使用 IE 8 以下版本的用户目前已不足 17%，并且还在不断下降，旧版本被淘汰是大势所趋，因此不必过于担心此问题。

（2）Firefox（火狐）浏览器是相当受欢迎的一款浏览器，目前已占有全球市场份额的 13%，在某些国家的占有率超过 80%。Firefox 是 Mozilla 基金会开发的开源浏览器，以其稳定的性能、良好的安全性、丰富的组件著称。由于其开源性，大量的程序员、研究员、工程师在上面开发扩展组件，使其几乎无所不能。

（3）Chrome（谷歌）浏览器是由 Google 公司开发的浏览器。其特点是简单、快速，并且不易崩溃。其界面相当简洁，几乎所有空间都用于显示网页，而且不仅浏览网页的速度快，打开软件的速度也快，能明显感觉到与别的浏览器之间的区别。**本书中的所有代码，都是用 Chrome 进行测试的。**

IE、Firefox、Chrome 三款浏览器所占市场份额之和超过 90%，因此使用这三款对网页程序进行测试最重要。其他还有在苹果计算机的系统 Mac OS X 中使用的 Safari 浏览器，Opera 浏览器，符合国人用户习惯的 360、搜狗、傲游等浏览器，种类很多，如图 2-2 所示，有些浏览器的内核是一样的，只是根据特定用户的习惯做了组件的扩展。D3 可运行于 IE 9+、Firefox、Chrome、Safari、Opera 等浏览器。

图 2-2　各种浏览器的图标

2.1.2　服务器

服务器是一个管理资源并为用户提供服务的计算机。有各种规模的服务器，最小的可能与个人电脑配置差不多，但在处理能力、稳定性、安全性等方面还是有差异的。近年来出现了云服务器，只需要支付相当低的价格，即可拥有服务器，并且其管理方式比物理服务器更加简单高效。

要使服务器能够提供各种服务，需要安装服务器软件。如果要提供 Web 信息浏览服务，需要安装 Web 服务器软件，其主要需支持 HTTP 协议，能处理用户发送过来的 HTTP 请求。常见

的 Web 服务器软件有 Apache、Tomcat、IIS 等。

（1）Apache HTTP Server（简称 Apache）是 Apache 软件基金会的一个开放源码的 Web 服务器软件，可以在大多数计算机操作系统中运行。由于其多平台和安全性被广泛使用，是最流行的 Web 服务器端软件之一。

（2）Tomcat 也是 Apache 软件基金会一个著名的服务器软件，其与 Apache 的区别和联系如下。

- Apache 只支持 HTML 静态网页，通过插件可支持 PHP；Tomcat 支持 ASP、JSP、PHP、CGI 等动态网页。
- Apache 是用 C 语言实现的；Tomcat 使用 Java 实现的，更好地支持 Servlet 和 JSP。
- Apache 的稳定性较好。
- Apache 对于静态页面的解析速度比 Tomcat 快。
- Apache 比 Tomcat 早，本质上来说 Tomcat 的功能可以替代 Apache。

（3）Internet Information Services（IIS）是微软公司提供的服务器软件，除了提供 Web 服务之外，还提供 FTP 服务、SMTP 服务等。其使用比较简单，早期版本漏洞较多，但从 IIS 6.0 版本开始做了大量修复，安全性有了大幅增长。

Web 服务器软件并非只是服务器计算机才能安装，在个人电脑上也能安装，这对于开发者测试是十分方便的。

本书中的示例，都使用 Apache HTTP Server 进行测试。安装 Apache 的方法在第 3 章有详细介绍。

2.2 HTML&CSS

HTML（Hyper Text Markup Language）指的是超文本标记语言，CSS（Cascading Style Sheets）指的是层叠样式表。用浏览器打开任意一个网页，右键单击页面，选择“查看网页源代码”命令，即可看到用于描述网页内容的源码信息。最常见到的结构有两种，第一种形如：

```
<!DOCTYPE html>
<html>
    <head>
        <title>宋词</title>
    </head>
<body>
    <h1>一剪梅</h1>
    <p>红藕香残玉簟秋，轻解罗裳，独上兰舟</p>
    <p>云中谁寄锦书来，雁字回时，月满西楼</p>
</body>
</html>
```

另一种代码的样式形如：

```
p {
    color:red;
    font-family:simsun;
    font-size:20px;
}
```

前者就是 HTML 代码，用于定义文档的结构和内容，例如显示什么文字、用段落显示还是表格显示等。后者是 CSS 代码，用于定义 HTML 元素的样式，如字体大小、背景颜色、布局等。

要使用 HTML 和 CSS 来制作网页，只需要在记事本程序里编写即可。新建一个后缀为.txt 的文本文件，然后将后缀改为.html，再用记事本程序打开后编写代码。编写完保存之后，再使用浏览器打开此文件，即可浏览效果。但是为了方便，一般使用功能更强大的记事本软件（例如 Notepad++、Sublime Text），有代码高亮的功能，能加快开发速度。

2.2.1　HTML 元素

HTML 是一种标记语言，每个元素都有一个标记标签（Markup Tag）。上面的代码中，已经使用了<html>、<body>、 等标签，都是用一对尖括号包围的。标签有成对出现的，如 ；也有不成对出现的，如
。对于成对出现的标签，前一个叫作开始标签，不带斜杠；后一个叫作结束标签，带斜杠。

1．文档声明

在上面的代码中首先出现的是<!DOCTYPE>，这是一个声明，不是 HTML 标签，其主要目的是告诉浏览器 HTML 的版本信息。此声明需要在 HTML 的第一行书写。

旧版的 HTML 4.01 中，此声明有三种形式：

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

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

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

其中，第一种是过渡模式；第二种是严格模式；第三种等同于第一种，但允许使用框架集内容。在 HTML5 版本中，此声明写作：

```
<!DOCTYPE html>
```

2014 年 10 月 29 日，万维网联盟宣布 HTML5 标准最终制定完成，在这之前大部分浏览器就已经支持了某些 HTML5 的特性。HTML5 的应用必然会越来越广泛，因此，DOCTYPE 的声明尽可能写成 HTML5 的形式，而且此形式也最简洁。

2．头部

<head>是头部元素的标签，其包含的信息用于告诉浏览器文档的标题、页面的编码、脚本文件和样式文件的引用地址等。

- title

定义文档的标题，此标题包括：浏览器标签中的标题、收藏页面时的默认标题、显示在搜索引擎结果中的标题。使用方法为：

```
<title>宋词</title>
```

- meta

定义元数据的信息。<meta>没有结束标签，其用法是通过 http-equiv 或 name 指定信息的种类，再用 content 来定义此种类的内容，常见的有：

```
<meta http-equiv="content-type"
      content="text/html;charset=gb2312" />

<meta name="description" content="HTML Document" />

<meta name="keywords" content="HTML, CSS, JavaScript, D3" />
```

第一行告诉浏览器文档的类型为 HTML，使用的编码为 GB2312。第二行告诉搜索引擎网页的主要内容是什么。第三行告诉搜索引擎网页的关键词是什么。

- link

用于引用外部资源，如引用样式表：

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

- style

用于在文档中定义 CSS 样式，如果数量很多，则建议写在外部文件中，再用<link>引用。如果数量较少，有时为了便于查看 HTML 元素和 CSS 样式的关系，可将其在<style>中定义。

- script

用于定义客户端脚本，最常见的是 JavaScript 脚本，目前已经成为事实上的标准语言。

3．属性

标签可以拥有属性，表示如下：

```
name="value"
```

属性名称不加引号，属性值通常加双引号，也可以加单引号。前面出现的 rel="stylesheet" 和 name="keywords"都是属性值。

4. 主体

主体由<body>标签定义，<body>里包含着各种元素标签，用于表示文档的内容。大多数 HTML 元素标签被分为**块级元素**和**内联元素**。

块级元素在显示时会以新行表示，常见的如下所示。

- h1、h2、h3、h4、h5、h6

定义标题，h1 是最大的标题，h6 是最小的标题。

- p

定义段落，末尾会自动换行。

- ul、ol、li

定义列表，ul 用于无序列表，ol 用于有序列表，内部再嵌套 li，每行使用一个。无序列表使用小黑圆圈表示，有序列表使用数字表示。定义一个两行的无序列表如下：

```
<ul>
<li>张三</li>
<li>李四</li>
</ul>
```

- table

表格，其中每行由<tr>定义，每列由<td>定义。定义一个两行两列的表格如下：

```
<table border="1">
    <tr>
        <td>第一行，第一列</td>
        <td>第一行，第二列</td>
    </tr>
    <tr>
        <td>第二行，第一列</td>
        <td>第二行，第二列</td>
    </tr>
</table>
```

- div

定义文档中的分区或节，内部包含任意其他元素。

内联元素在显示时不以新行表示，常见的如下所示。

➢ a

超链接的标签，点击后跳转到目标页面，通过 href 来指定 URL 路径。

```
<a href="http://www.ourd3js.com/">数据可视化</a>
```

- img

图片，通过 src 来指定图片 URL 地址，alt 设定当图片无法加载时显示的文字。

```
<img src="dog.jpg" alt="Dog Image"/>
```

- span

用于组合行内元素，添加在行内元素之间。

5. 注释

注释标签在浏览器中不会显示，程序员自己可适当添加以便阅读。添加的方法为使用<!--和-->将内容包含在里面。

```
<!-- <p>This is a dog. -->
```

2.2.2 CSS 选择器

如果要给两个段落定义相同的样式，该如何做呢？

```
<p style="color:red;background-color:yellow;font-size:22px;">
   红藕香残玉簟秋，轻解罗裳，独上兰舟</p>

<p style="color:red;background-color:yellow;font-size:22px;">
   云中谁寄锦书来，雁字回时，月满西楼</p>
```

上述代码将两个段落都设置成红色字体、黄色背景、22px 大小的字体。目的虽然达到了，但十分冗长。如果有更多的段落需设置成这样呢？这样的代码不仅不利于阅读，而且当希望将字体颜色改变成黑色时，需要改很多地方，非常不便。

定义一个 CSS 选择器，然后在段落标签中应用该选择器，即可达到目的。

```
.pstyle {
   color: red;
   background-color: yellow;
   font-size: 22px;
}
```

在<p>元素中应用选择器。

```
<p class="pstyle">红藕香残玉簟秋，轻解罗裳，独上兰舟</p>
<p class="pstyle">云中谁寄锦书来，雁字回时，月满西楼</p>
```

是否感觉心情愉快了许多，将来如果想要修改字体、颜色等，只需在选择器中修改即可，

不需要改动 HTML 代码，极大地提高了效率。要记住，HTML 只应用来描述网页的内容，如“标题是什么”、“段落是什么”；而如何表现这些内容，如“字体颜色”、“字体大小”等都使用 CSS。除非必要，否则不要在 HTML 元素中添加大量的属性来描述如何表现元素。

1. 语法

CSS 由两部分组成：**选择器的名称**和**“属性名称-属性值”**。

```
selector {
    name1: value1;
    name2: value3;
    name3: value3;
}
```

属性名称与属性值之间用冒号隔开，每一对值之间加分号隔开。冒号之后建议加一空格，有利于阅读，空格不会影响效果。如果属性值中间有空格，需要给值加引号：

```
h1 { font-family: "sans serif"; }
```

2. 选择器

一个 CSS 选择器能对应一个元素，也能对应多个元素，也可能多个选择器对应一个元素。要达到不同的对应方式，选择器有如下数种类型。

- 元素选择器

以 HTML 元素的标签作为名称。例如：

```
p { color: blue; }
```

则所有段落的文字都会变为蓝色。

- 选择器分组

如果几个选择器的样式相同，可用逗号分隔：

```
h1, h2, p {
    color: blue;
}
```

则<h1>、<h2>、<p>三个标签都将文字变为蓝色。

- 类选择器

在选择器名称前加一个点（.），表示是类选择器：

```
.important { color: red; }
```

要应用此类，在元素标签里添加一个 class 属性，

```
<p class="important">段落</pan>
```

如果此类选择器会应用在很多标签里，而你只希望在段落的标签里才应用此选择器，可：

```
p.important { color: red; }
```

要注意，p 和.important 之间没有空格，如果加了空格则变为派生选择器。

- ID 选择器

如果希望某个特定的元素具有某种样式，可以使用 ID 选择器。与类选择器不同的是，ID 选择器在文档中只使用一个，但是即便多次使用，很多浏览器也能解读。如此，就变得很像类选择器，建议不要使用多次。

```
#index { font-weight: bold; }
<p id="index">Index 1</p>
```

在选择器名称前加#号，在元素属性中添加 id 属性即可，属性值不需要加井号（#）。

- 派生选择器

派生选择器又分为三种。如果希望 p 中的 span 元素应用样式，可在 p 与 span 之间加一空格，这个称为后代选择器。

```
p span { color: red; }
```

如果只希望选择 p 的直系子元素 span，则在 p 与 span 之间加一个大于号（>），这个称为子元素选择器。

```
p > span { color: red; }
```

如果希望选择紧接在 h1 元素之后的一个 p 元素，且 h1 和 p 拥有共同的父元素，则在 h1 和 p 之间加一个加号（+），这个称为相邻兄弟选择器。

```
h1 + p { color: red; }
```

3. 属性名称和属性值

属性名称和属性值是成对出现的，CSS 中有很多属性名称，最常见的属性如下。

- 尺寸

width：元素宽度。

height：元素高度。

- 背景

background-color：背景颜色。

background-image：背景图像。

background-position：背景图像的起始位置。

background-repeat：背景图像是否及如何重复。

- 文本

color：文本颜色。

line-height：行高。

text-align：对齐元素中的文本。

- 字体

font-family：字体。

font-size：字体尺寸。

font-style：字体风格。

font-weight：字体粗细。

- 边框

padding：内边距。

border：边框。

margin：外边距。

- 定位

position：元素是静态的、相对的、绝对的，还是固定的。

top：到上边界的距离。

right：到右边界的距离。

bottom：到下边界的距离。

left：到左边界的距离。

float：浮动。

clear：清除浮动。

2.2.3　综合运用

HTML 用于描述“内容是什么”，CSS 用于描述“如何表现此内容”。下面举一个例子，综合使用 HTML 和 CSS，来实践前面的内容。制作网页时，首先要明确所需的网页布局，假设布局如图 2-3 所示。

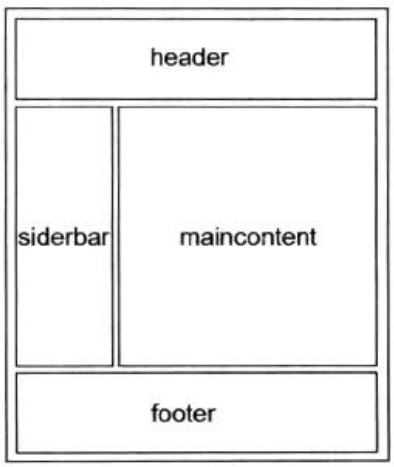


图 2-3　网页布局（各区域名称是 div 的 id 号）

网页被划分为四块区域。可用 div 来定义各区域，HTML 结构如下：

```
<div id="container">
    <div id="header">
    </div>
    <div id="mainbody">
        <div id="siderbar">
        </div>
        <div id="maincontent">
        </div>
    </div>
    <div id="footer">
    </div>
</div>
```

分别用 id 属性来标示各区域，其中 siderbar 和 maincontent 被包含于 mainbody 中，最后再将所有区域块都放入 container 区域块里，方便操控。区域块明确之后，首先在 HTML 里添加 h1、p、ul 等内容标签，然后再为各区域块定义样式。如图 2-4 所示，左图是未添加 CSS 样式之前的模样，可以看到这是纯粹的内容，区域块也没有如图 2-3 所示那样布局。右图是添加了 CSS 样式之后，已经变成了目标布局，文字也美观了不少。

图 2-4　HTML 的内容（左）加上 CSS 样式之后的效果（右）

图 2-4 的布局中，siderbar 和 maincontent 是使用浮动属性布局，这种布局被应用在很多网站上。为 HTML 元素添加内容后，代码如下：

```
<div id="siderbar">
<h3 class="category">作者</h3>
    <ul>
        <li>柳永</li>
        <li>苏轼</li>
```

```
        <li>欧阳修</li>
        <li>李清照</li>
    </ul>
</div>
<div id="maincontent">
    <p>寒蝉凄切。对<span class="important">长亭</span>晚，骤雨初歇。</p>
    <p>都门帐饮无绪，留恋处，<span class="important">兰舟</span>催发。</p>
    <p>执手相看泪眼，竟无语凝噎。</p>
    <p>念去去，千里烟波，暮霭沉沉楚天阔。</p>
</div>
```

使用 h3、ul、li、p、span 添加了内容，要注意这些标签中都没有包含样式属性，样式都要在 CSS 选择器中定义。接下来讲解 siderbar 和 maincontent 这两个 id 的样式。

```
#siderbar{
    width: 25%;
    height: 100%;
    float: left;
    clear: left;
    border: 1px solid gray;
    border-radius: 5px;
}

#maincontent{
    width: 73%;
    height: 100%;
    float: right;
    clear: right;
    text-align: center;
    border: 1px solid gray;
    border-radius: 5px;
    font-size: 20px;
}
```

width 和 height 是区域块的宽度和高度，这里使用百分比，表示父元素宽高的百分之多少。float 分别设定为 left 和 right，表示前者向左浮动，右者向右浮动。clear 表示清除浮动。border 用于定义区域块的边框，三个值依次为宽度、样式、颜色。border-radius 用于定义圆角边框，值为圆角的半径。font-size 用于设定字体大小。

2.3　JavaScript

JavaScript 是 D3 的开发语言，使用 D3 时会涉及很多 JavaScript 的概念，因此基础知识的掌

握是必要的。JavaScript 的语法并不复杂，与 C/C++、Java 很相似，只是记住语法可能只需要半个月，但想要得心应手地使用可能需要数年。本节仅介绍学习 D3 所必需的基础知识。

2.3.1 在 HTML 中使用 JavaScript

在 HTML 文档中通过<script>标签来使用 JavaScript。如果要在 HTML 里写 JavaScript 代码，则：

```
<script type="text/javascript">
console.log("Hello, World");
</script>
```

属性 type 用于设定脚本语言的类型，除了 text/javascript 之外，还可设定为 text/ecmascript、text/vbscript 等脚本语言。但 JavaScript 已成为事实上 Web 浏览器的标准语言，因此可不设定这个值。如果忽略，则默认为 JavaScript 语言。上面的代码输出“Hello, World”字符串，函数 console.log()能够将文本输出到控制台。

如果 JavaScript 是写在一个单独的文件里的，那么引用方法为：

```
<script src="http://d3js.org/d3.v3.min.js"
  charset="utf-8"></script>
```

通过 src 属性来指向外部链接文件，此处指向了 d3.v3.min.js 的源文件，D3 的所有代码就是写在这个文件里的。有一点要注意，浏览器会按照<script>出现的顺序依次加载，一个完了再加载另一个。如果<script>写在 HTML 的<head>里，加载的文件过多过大，则在加载的时间里用户看到的是一片空白。因此，<script>也可以放到<body>中所有的内容元素之后，让浏览器先显示内容，再加载脚本。JavaScript 文件的后缀名通常带有.js 扩展名，但并不是必需的，浏览器不会检查扩展名。

2.3.2 语法

1. 区分大小写

变量名、函数名是区分大小写的，因此 name 和 Name 是不同的。第一个字符必须是字母、下画线（_）或美元符号，后面的字符可以有数字。

2. 注释

注释分为单行注释和多行注释，注释方法与 C/C++一样：

```
// var name = "zhangsan";      单行注释
```

```
/*                                    多行注释
   var a = 10;
   var b = a;
*/
```

3. 分号

每条语句结尾添加分号（;），虽然不添加浏览器也能解读，但容易造成意想不到的错误，因此建议都加上分号。

4. 花括号

花括号的方法建议写成：

```
function getName(person){
   return person.name;
}
```

而不是像 C 语言一样写成：

```
function getName(person)
{
   return person.name;
}
```

虽然两种都不报错，但每种语言都有一些不成文的规范，只有符合规范代码才能被更多人接受。

2.3.3　变量

定义变量时使用 var 操作符，后跟变量名，多个变量名之间用逗号隔开。如下所示：

```
var a;          //undefined
var b,c,d;      //undefined
```

如果定义变量时不为其赋值，则默认值为 **undefined**，表示该值未定义。为防止出现意外后果，通常会在定义时为其赋值。可以在一个 var 后对多个变量赋值，各变量间用逗号隔开：

```
var a = 10;
var b = 20,
    c = 25,
    d = 30;
```

JavaScript 的变量是松散类型的，所有变量的定义都是使用 var，既可以是数值，也可以是布尔型（true 或 false），还可以是字符串。并且，即便一个变量初始值为数值，仍然可以将其他

类型的值赋给它：

```
var a = 26;
var b = false;
var c = "message";
a = "box";
```

a 初始值为 26，但仍可用字符串 box 为其赋值，这与 C 语言不同。虽然如此，变量的类型最好自始至终都是确定的，不建议轻易更改。

2.3.4 数据类型

JavaScript 有五种基本数据类型：undefined、null、boolean、number、string。还有一种复杂数据类型：object。变量属于哪一种数据类型可用 typeof 查看，如下所示：

```
var a = 26;
console.log(typeof a);  //number
```

a 变量中存储的是数值 26，属于数值(number)类型，因此使用 typeof 测定后，输出了 number。typeof 是 JavaScript 的一个关键字。下面分别介绍六种数据类型。

1. undefined

未初始化的变量默认值都是 undefined，表示该值未定义。如果某变量将来要为其赋值为其他基本数据类型之一，那么将其初始化为 undefined 较好：

```
var c = undefined;      //初始为undefined
c = 10;                 //将来将其赋值为number（数值）类型的10
```

2. null

与 undefined 十分类似，不同之处在于：null 表示一个空对象、undefined 表示一个未定义的基本类型的变量。如果某变量将来要为其赋值为 object 类型，将其初始化为 null 较好。

```
var o = null;
o = { name:"zhangsan", age:19 };
```

如果对一个赋值为 null 的变量用 typeof 检测，结果是 object：

```
var o = null;
console.log(typeof o);  //object
```

这其实也证明了 null 适合用于表示空对象，因此，当不需要使用一个对象，将其清空时，可赋值为 null。赋值之后，其类型仍然为 object，表示这个变量还是对象，只是被清空了而已。

```
var e = { name:"lisi", age:20 };
e = null;
console.log(typeof e);  //object
```

3. boolean

布尔类型有两个值：true 和 false，表示“真”和“假”两种状态。在条件语句（如 if）中，其他类型的值会自动转换为布尔型。有五种值会转换为 false：0、NaN、undefined、null、""（空字符串），其他值均转换为 true。那么在进行条件判断时，即可使用如下语句：

```
var a = 10;
if(a){
   console.log("数值不是0或NaN");
}
```

4. number

数值可使用十进制数、八进制数、十六进制数，八进制数以零（0）开头，十六进制数以 0x 开头：

```
var num_10 = 120;       //十进制数120
var num_8 = 020;        //八进制数20，相当于十进制数16
var num_16 = 0x3A;      //十六进制数3A，相当于十进制数58
```

浮点数值用小数点来表示，较大或较小的数可用指数来表示：

```
var f = 3.1415;         //浮点数
var m = 3e5;            //指数形式，表示3 * 10^5
```

JavaScript 的数值类型有范围。最大数值为 Number.MAX_VALUE，这个值通常为 1.7976931348623157e+308；最小数值为 Number.MIN_VALUE，这个值为 5e-324。如果超出此范围，正数则返回 Infinity（正无穷），负数则返回-Infinity（负无穷）。另外，正数除以 0 返回正无穷，负数除以 0 返回负无穷。

数值类型还有一个特殊的值 NaN，是 Not a Number 的意思，表示不是一个数。

5. string

字符串类型可由双引号或单引号表示：

```
var str_1 = "China";
var str_2 = 'America';
```

字符串的长度可用 length 得到：

```
console.log(str_1.length); //5
```

两个字符串拼接可用加号（+）实现：

```
console.log(str_1 + str_2);    //ChinaAmerica
```

6. object

对象 object 是拥有**属性**和**方法**的数据类型。**属性**是与对象相关的值，**方法**是在对象上执行的动作。现实生活中，楼房可以是对象，人也可以是对象。下面的做法就创建了一个对象。

```
var person = new Object();
```

但是，这个对象没有属性，也没有方法。下面为其添加属性和方法：

```
person.name = "WangWu";
person.age = 20;
person.growUp = function(){
    this.age += 1;        //年龄增加1岁
}
```

这段代码增加了两个属性 name 和 age，还有一个方法 growUp()。接下来试着访问这些属性和方法：

```
console.log( person.name );    //WangWu
console.log( person.age );     //20
person.growUp();
console.log( person.age );     //21
```

给方法命名的方式通常有两种。一种是空格之后单词的首字母大写，即 growUp()；另一种是空格用下画线代替（_），字母全用小写，即 grow_up()。

2.3.5 操作符

操作符用于数值的运算、比较、判断等操作，具体有以下几类。

1. 算术操作符

算术操作符即加减乘除等操作，是最常用的操作符。

```
var num1 = 5 + 3;        //加法，结果为8
var num2 = 5 - 3;        //减法，结果为2
var num3 = 5 * 3;        //乘法，结果为15
var num4 = 5 / 3;        //除法，结果为1.6666666666666667
var num5 = 5 % 3;        //求余，结果为2
num1++;                  //递增，结果为9
num2--;                  //递减，结果为1
```

2. 赋值操作符

算术运算符和等号配合使用，能减少代码的书写量：

```
var a = 18;          //赋值，使用等号（=）
a += 3;              //等同于 a = a + 3; 结果为21
a -= 2;              //等同于 a = a - 2; 结果为19
a *= 4;              //等同于 a = a * 4; 结果为76
a /= 2;              //等同于 a = a / 2; 结果为38
a %= 7;              //等同于 a = a % 7; 结果为3
```

3. 布尔操作符

布尔操作符共有三个：非（!）、与（&&）、或（||）。

```
var isDog = false;
var isAnimal = true;
console.log( !isDog );              //true
console.log( isDog && isAnimal ); //false
console.log( isDog || isAnimal ); //true
```

4. 关系操作符

关系操作符即大于（>）、小于（<）、大于等于（>=）、小于等于（<=）、相等（==）、全等（===）、不相等（!=）、不全等（!==）这八种。

```
console.log( 8 > 7 );          //true
console.log( 8 >= 7 );         //true
console.log( 8 < 7 );          //false
console.log( 8 <= 7 );         //false
console.log( "8" == 8 );       //两者转换后相等，结果为true
console.log( "8" === 8 );      //两者转换后不相等，false
console.log( "7" != 7 );       //两者转换后相等，false
console.log( "7" !== 7 );      //两者转换后不相等，true
```

特别要注意“相等”和“全等”的区别，相等操作符会自动对数据类型进行转换，全等则不会。

5. 条件操作符

条件操作符只有一个：

```
var result = 5 > 3 ? true : false;     //true
```

如果问号前的条件为 true，则返回冒号之前的值；如果为 false，则返回冒号之后的值。

2.3.6 语句

语句用于流程控制，实现条件判断、循环等功能。

1. if-else 语句

最常见的条件判断语句，可以单独使用 if，也可以添加 else if 和 else。

```
if( 20 > 33 ){
    console.log("大于");
}else if( 20 === 33 ){
    console.log("等于");
}else{
    console.log("小于");      //符合这个条件
}
```

2. while 和 do-while 语句

while 语句是先测试条件再循环，do-while 语句是先循环再判断条件。

```
var i = 0;
while(i<5){      //先判断
    i++;         //再循环
}

var j = 0;
do{
    j++;         //先循环
}while(j<10)     //再判断
```

3. for 和 for-in 语句

for 是一种功能更强大的循环语句，紧跟的括号里用两个分号分隔成三个部分，第一部分用于值的初始化，第二部分用于条件判断，第三部分用于求新值：

```
for(var i=0;i<5;i++){
    console.log(i);      //0,1,2,3,4
}
```

for-in 主要用于枚举对象属性和方法，如对于以下对象：

```
person.name = "WangWu";
person.age = 20;
person.growUp = function(){
    this.age += 1;       //年龄增加1岁
```

```
}
```

则可写代码如下：

```
for(var prop in person){
    console.log(prop);          //输出name,age,growup
}
```

4. switch 语句

与 if 类似，用于条件判断。switch 适于判断的情况是，如果某变量是这个值的情况下，执行某操作，如果是那个值，执行另一操作。每种情况最后要用 break 结尾，防止继续往下判断。

```
var i = 20;
switch(i){
    case 10:                            //判断过此值，不符合
        console.log(10);
        break;                          //用break跳出switch
    case 20:                            //判断过此值，符合
        console.log(20);                //输出20
        break;                          //用break跳出switch
    case 30:                            //没有判断过此值
        console.log(30);
        break;
    default:                            //如果都不符合，默认怎么处理
        console.log("都不符合");
}
```

5. break、continue 和 label 语句

break 语句可退出循环，不再判定下次执行循环的条件；continue 只是退出本次循环，还会继续判定下次是否执行循环：

```
for(var i=0;i<3;i++){
    if( i === 1)
        break;
    console.log(i);             //只输出0
}

for(var i=0;i<3;i++){
    if( i === 1)
        continue;
    console.log(i);             //输出0和2
}
```

但是，break 和 continue 都只能跳出当前花括号的循环。如果要跳出多重循环，会比较麻烦。

因此，可以配合 label 使用，用于跳出多重循环。label 标记在循环之前：

```
fori: for(var i=0;i<3;i++){
    for(var j=0;j<3;j++){
        if( i == 1 && j == 0 )
            break fori;
        console.log(i+"\t"+j);
    }
}
```

fori 即最外层循环的 label 名称，在 break 之后加上此名称即可指定跳出哪个循环。

2.3.7 函数

JavaScript 的函数使用 function 关键字来声明，后跟参数及函数体。与 C/C++或 Java 等不同，JavaScript 中函数的参数不需要指定数据类型。其基本语法如下：

```
function add(num1, num2){
    return num1 + num2;
}
```

上面定义了一个函数名为 add 的函数，函数体里执行的是两个数相加并返回。然后即可使用此函数名来调用函数：

```
var a = add(3,5);        //a的值为8
```

还有一种定义函数的方法：

```
var add = function(num1,num2){
    return num1 + num2;
}
```

表面上看是将一个无名函数 function 赋值给了一个变量 add，这种方式在后续 D3 的学习中会经常见到。

2.3.8 对象

在 2.3.4 节里说过一些关于对象的内容，提供了一种创建对象的方式。除此之外，还有一些常见的创建对象的方法：

```
var person = {
    name: "WangWu",          //两个属性之间要添加逗号
    age: 20,
```

```
    growUp: function(){
              this.age += 1;
            }              //如果之后还有属性，这里也要添加逗号
}
```

还有一种方法，可制作一个构造函数，然后即可使用 new 来创建对象，对于习惯了 C 或 Java 的人可能会感到很开心：

```
function Person(name,age){
    this.name = name;
    this.age = age;
    this.growUp = function(){
        this.age += 1;
    }
}
var wangwu = new Person("WangWu",18);
```

调用对象的属性有两种方法，都很常用：

```
console.log(wangwu.name);          //小圆点
console.log(wangwu["name"]);       //方括号
```

通常使用小圆点的方式，但方括号的方式也很重要，有一些场合只能使用这种方式。另外，使用方括号的方式调用函数也是可以的：

```
wangwu["growUp"]();
```

2.3.9　数组

数组是常用的数据结构，用一对方括号即可定义。JavaScript 的**数组其实是一种对象**。使用 typeof 检测，可知数组是 object 类型：

```
var a = [1,2,3,4,5];
console.log(typeof a); //object
```

同一数组里还可以是不同类型的值：

```
var a = [1,"apple",false,4,5];
```

还有一种使用 Array 构造函数创建数组的方法：

```
var a = new Array(1,2,3,4);     //长度为4，数组项分别为1,2,3,4的数组
var b = new Array(20);          //长度为20，数组项还没有赋值的数组
```

通过这种方式，更能清楚地看到数组是对象。因此可以想象，数组里的各项其实就是这个

Array 对象里的属性，属性的名称分别为 0、1、2、3（数组的序号）等。那么能否使用如下方式调用呢？

```
console.log(a.1);        //不能得到结果
```

结果出错。前面说过，JavaScript 的变量名是不能以数字开头的，因此数组里的属性名称 0、1、2、3 等可看作是特例，是数组专用的。第 2.3.8 节中提到调用对象的方式除了使用小圆点之外，还可使用方括号的方式：

```
console.log(a["1"]);    //得到正确的结果
```

这样就证明了数组项其实是对象的属性，数组项的序号实际上是属性的名称。这一点对于理解数组是什么非常重要。

另外，数组的长度可用 length 来获取。length 属性是可以赋值的，赋值之后，超出此长度的部分就不能使用了。

```
var city = ["Beijing","Shanghai","Guangzhou"];
console.log(city.length);       //3
console.log(city);              //["Beijing", "Shanghai", "Guangzhou"]
city.length = 1;                //将长度赋值为1
console.log(city);              //["Beijing"]
```

给数组添加项，可以直接给指定序号的项赋值：

```
city[10] = "GuiLin";
console.log(city.length);       //11
```

其实，从序号 3～9 都是没有项的，但是 length 的值却为 11。length 总是比最大的序号大 1（数组序号从 0 开始），与中间有多少项无关。

给数组添加和删除项的方法，还可使用以下函数。

- push：在末尾添加项。
- pop：将末尾项删除并返回。
- shift：将第一项删除并返回。
- unshift：从最前面推入项。

2.4 DOM

DOM，指文档对象模型（Document Object Model），是针对结构化文档的一个接口，它允许程序和脚本动态地访问和修改文档。其中，针对 HTML 的模型称为 HTML DOM。换言之，使用这套模型即可任意访问和修改 HTML 元素。D3 中的函数大量脱胎于 DOM，因此了解 DOM

的基本原理对于理解 D3 有很大的帮助。

2.4.1　结构

DOM 是以树形结构来描述 HTML 文档的，其被称为节点树。每个 HTML 元素都是树上的一个节点，节点之间的关系如图 2-5 所示。

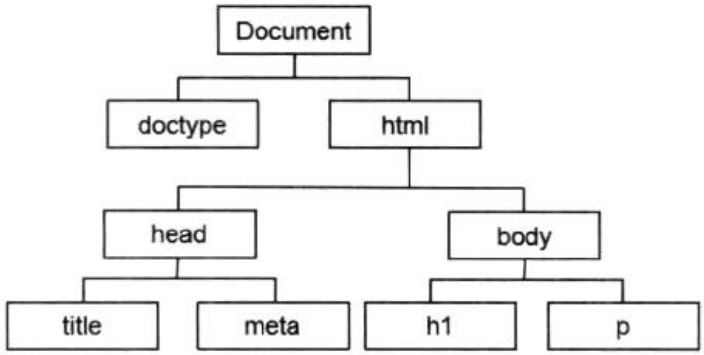


图 2-5　HTML 文档的树形结构

html 是 body 的父节点（parent），body 是 html 的子节点（child），body 是 head 的同胞节点（sibling）。

2.4.2　访问和修改 HTML 元素

DOM 的文档对象保存在 document 中，调用其方法或属性，即可访问 HTML 文档中的任意元素。访问方式有：

```
document.getElementById("myid");                    //返回id为myid的元素
document.getElementsByTagName("p");                 //返回所有标签为p的元素
document.getElementsByClassName("myclass");//返回类为myclass的元素
```

这三种方法返回所指定的元素。下面以 getElementsByTagName 为例来说明其用法：

```
<p>Apple</p>
<p>Banana</p>
<script>
   var para = document.getElementsByTagName("p");
   for( var i=0; i<para.length; i++ ){
       console.log( para[i].innerHTML );
   }
</script>
```

上面的代码使用 getElementsByTagName 获取了文档中所有的 p 元素，返回值保存在变量 para 中。para 是一个数组，保存了所有的 p 节点。然后，在 for 循环中输出了 para 的内容，最终结果是在控制台中输出 Apple 和 Banana 两行文字。这里使用了 innerHTML 属性，这个属性是能够对其赋值的。例如将上述 for 循环改成：

```
for( var i=0; i<para.length; i++ ){
    para[i].innerHTML = "Pear";
}
```

则两个 p 元素的内容均被修改为 Pear 了。在 HTML DOM 中，常用的属性如下。

- innerHTML：元素标签内部的文本，包括 HTML 标签。
- innerText：元素标签内部的文本，但不包括 HTML 标签。
- outerHTML：包括元素标签自身在内的文本，也包括内部的 HTML 标签。
- outerText：包括元素标签自身在内的文本，但不包括 HTML 标签。
- nodeName：节点名称。
- parentNode：父节点。
- childNodes：子节点。
- nextSibling：下一个同胞节点。
- previousSibling：上一个同胞节点。
- style：元素的样式。

使用 style 属性，即可修改元素的样式，如下代码所示：

```
var body = document.getElementsByTagName("body");
console.log(body);
body[0].style.backgroundColor = "blue";
```

这段代码将 body 元素的背景颜色修改成了蓝色。要注意，属性名称 backgroundColor 与 CSS 中的名称 background-color 是不同的，因为变量名中不允许拥有减号（-）。故其对应名称是去掉减号后，将下一个单词的首字母改成大写即可。

2.4.3 添加和删除节点

给文档中添加节点可使用 appendChild()方法，每个元素节点中都有这个方法，能在该元素的末尾添加子节点。为此，首先要创建一个节点，然后才能添加。请看如下代码：

```
var para = document.createElement("p");                    //创建节点p
para.innerHTML = "Hello";                                  //给p的内容赋值
var body = document.getElementById("mybody");              //获取body节点
body.appendChild(para);                                    //给body节点添加子节点
```

还可以删除文档中的节点：

```
<body id="mybody">
    <p id="mypara">Apple</p>
    <p>Banana</p>
    <script>
        var para = document.getElementById("mypara");
        var body = document.getElementById("mybody");
        body.removeChild(para);
    </script>
</body>
```

上面的代码中，首先获取了 id 为 mypara 的元素 p 节点，然后获取了 body 元素节点，再使用 body 中的 removeChild()方法删除掉 p 节点。最终结果是页面中只有一个段落，文字为 Banana。

2.4.4　事件

HTML DOM 通过事件能够与用户进行交互，如鼠标单击、鼠标移入、页面加载等。先看如下代码：

```
<p id="mypara">Click Here</p>
<script>
    var para = document.getElementById("mypara");
    para.onclick = function(){
        this.innerHTML = "Thank you";
    }
</script>
```

上述代码为选择的 p 元素定义了一个事件 onclick，当用户单击该元素时，便调用 onclick 中的无名函数 function。该函数的内容是将段落文字更改为 Thank you。像这样的事件有很多，常用的如下。

- onload：页面或图片加载完成时。
- onclick：鼠标单击。
- ondblclick：鼠标双击。
- onkeydown：键盘某个按键按下。
- onkeypress：键盘某个按键按下并松开。
- onkeyup：键盘某个按键松开。
- onmousedown：鼠标按钮按下。
- onmousemove：鼠标移动。
- onmouseout：鼠标从某元素移开。

- onmouseover：鼠标移到某元素上。
- onmouseup：鼠标松开。

2.5 SVG

SVG，指可缩放矢量图形（Scalable Vector Graphics），是用于描述二维矢量图形的一种图形格式，是由万维网联盟制定的开放标准。SVG 使用 XML 格式来定义图形，除了 IE8 之前的版本外，绝大部分浏览器都支持 SVG，可将 SVG 文本直接嵌入 HTML 中显示。D3 十分适合在 SVG 中绘制图形。

2.5.1 位图和矢量图

位图（Bitmap）是由一系列像素组成的。像素点各自具有自己的颜色（例如 RGB），以一定的排列构成图像。对于需要表现丰富色彩的图像，位图很合适。但是，如果将图像放大，需要产生新的像素，新的像素是以原始像素进行插值计算获得的，会造成失真。例如，放大后，线段看上去会不太整齐，随着放大倍数的增大，会产生强烈的锯齿效果。

矢量图（Vector Graphics）不是由像素定义的，而是由线段和曲线来定义的。相比位图，矢量图具有如下优点。

- 文件小。只需要保存像线条的起点和终点这样的信息，所需空间小。
- 缩放、旋转或改变形状等都不会失真，不影响显示品质，是“分辨率独立”的。
- 线条、颜色十分平滑，锯齿不明显。

如图 2-6 所示，位图放大之后失真现象严重，而矢量图则不受影响。但是要注意，矢量图最后也是要转换成位图来表示的，只不过其内部存储着产生位图的方法，放大时自动用不同的参数来运用这些方法。

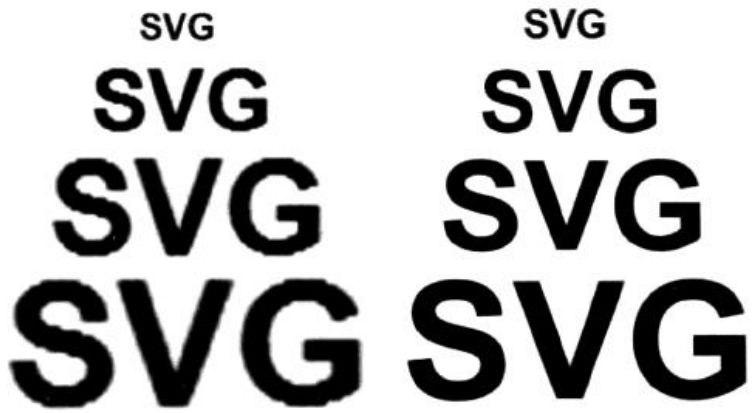


图 2-6 位图（左）和矢量图（右）放大后的效果

但是，矢量图并非就完全优于位图。由于矢量图是用数学方法描述的图，因此不适合表现自然度较高、复杂多变的图。

2.5.2　图形元素

使用 SVG 中的图形元素，可以在 HTML 文件中嵌入，也可直接将文件名改为 xxx.svg 来使用。首先，添加一组<svg>标签：

```
<svg width="300" height="300" version="1.1"
     xmlns="http://www.w3.org/2000/svg">
</svg>
```

其中，width 和 height 分别表示绘制区域的宽度和高度，version 表示 SVG 的版本号，1.1 版本是 2003 年确立的 W3C 标准，也是最新版本，xmlns 表示命名空间。需要绘制的图形都要添加到这一组<svg></svg>之间。SVG 中预定义了七种形状元素，分别为：矩形<rect>、圆形<circle>、椭圆<ellipse>、线段<line>、折线<polyline>、多边形<polygon>、路径<path>。

这些元素中表示形状的参数各有不同，但也有一些共同的属性，如平移的方向和大小、颜色样式等。下面介绍这些元素的参数和示例。

1．矩形

矩形的参数共有 6 个。

- x：矩形左上角的 x 坐标。
- y：矩形左上角的 y 坐标。
- width：矩形的宽度。
- height：矩形的高度。
- rx：对于圆角矩形，指定椭圆在 x 方向的半径。
- ry：对于圆角矩形，指定椭圆在 y 方向的半径。

下面分别绘制一个直角矩形和圆角矩形：

```
<rect x="20" y="20" width="200" height="100"
style=" fill:steelblue; stroke:blue; stroke-width:4; opacity:0.5" />

<rect x="250" y="20" rx="20" ry="30" width="200" height="100"
style=" fill:yellow; stroke:black; stroke-width:4; opacity:0.5" />
```

sytle 用于指定矩形的样式，结果如图 2-7 所示。

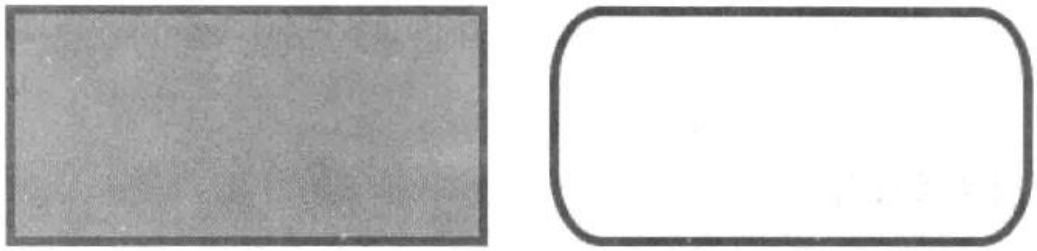


图 2-7　直角矩形（左）和圆角矩形（右）

2. 圆形和椭圆形

圆形的参数是 3 个。

- cx：圆心的 x 坐标。
- cy：圆心的 y 坐标。
- r：圆的半径。

椭圆的参数与圆形类似，只是半径分为水平半径和垂直半径。

- cx：圆心的 x 坐标。
- cy：圆心的 y 坐标。
- rx：椭圆的水平半径。
- ry：椭圆的垂直半径。

下面的代码绘制圆形和椭圆形：

```
<circle cx="150" cy="150" r="80"
style=" fill:yellow; stroke:black; stroke-width:4" />

<ellipse cx="350" cy="150" rx="110" ry="80"
style=" fill:#33FF33; stroke:blue; stroke-width:4" />
```

其结果如图 2-8 所示。

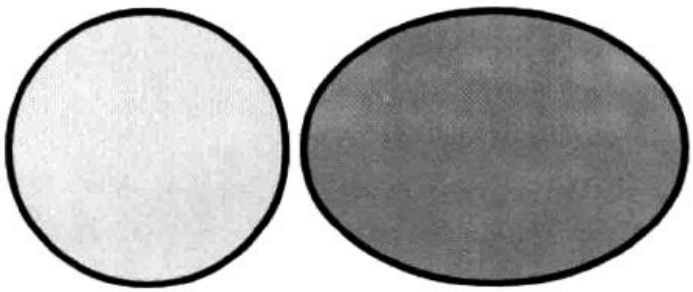


图 2-8　圆形和椭圆形

3. 线段

线段的参数是起点和终点的坐标。

- x1：起点的 x 坐标。
- y1：起点的 y 坐标。
- x2：终点的 x 坐标。
- y2：终点的 y 坐标。

绘制线段的代码为：

```
<line x1="20" y1="20" x2="300" y2="100"
      style=" stroke:black; stroke-width:4" />
```

结果如图 2-9 所示。

图 2-9　线段

4. 多边形和折线

多边形和折线的参数是一样的，都只有一个 points 参数，这个参数的值是一系列的点坐标。不同之处是多边形会将终点和起点连接起来，而折线不连接。下面的代码分别绘制多边形和折线：

```
<polygon points="100,20 20,90 60,160 140,160 180,90"
    style=" fill:LawnGreen; stroke:black; stroke-width:3 "/>

<polyline points="100,20 20,90 60,160 140,160 180,90"
    style=" fill:white; stroke:black; stroke-width:3 "
    transform="translate(200,0)"/>
```

结果如图 2-10 所示。

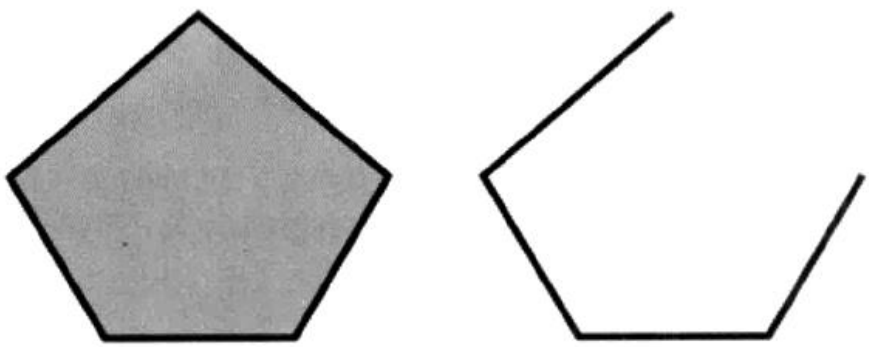


图 2-10　多边形（左）和折线（右）

5. 路径

<path>标签的功能最丰富，前面列举的图形都可以用路径制作出来。与折线类似，也是通过给出一系列点坐标来绘制。在 D3 中绘制地图时，会经常用到此标签。其用法是：给出一个坐标点，在坐标点前面添加一个英文字母，表示是如何运动到此坐标点的。英文字母按照功能可分为五类。

移动类

- M = moveto：将画笔移动到指定坐标。

直线类

- L = lineto：画直线到指定坐标。
- H = horizontal lineto：画水平直线到指定坐标。
- V = vertical lineto：画垂直直线到指定坐标。

曲线类

- C = curveto：画三次贝塞尔曲线经**两个**指定控制点到达终点坐标。
- S = shorthand/smooth curveto：与前一条三次贝塞尔曲线相连，第一个控制点为前一条曲线第二个控制点的对称点，只需输入第二个控制点和终点，即可绘制一个三次贝塞尔曲线。
- Q = quadratic Bézier curveto：画二次贝塞尔曲线经**一个**指定控制点到达终点坐标。
- T = Shorthand/smooth quadratic Bézier curveto：与前一条二次贝塞尔曲线相连，控制点为前一条二次贝塞尔曲线控制点的对称点，只需输入终点，即可绘制一个二次贝塞尔曲线。

弧线类

- A = elliptical arc：画椭圆曲线到指定坐标。

闭合类

- Z = closepath：绘制一条直线连接终点和起点，用来封闭图形。

上述命令都是用大写英文字母表示，表示坐标系中的绝对坐标。也可用小写英文字母，表示的是相对坐标（相对当前画笔所在点）。下面分别介绍其使用方法。

绘制直线：

```
<path d="M30,100 L270,300
         M30,100 H270
         M30,100 V300"
    style="stroke:black; stroke-width:3 "/>
```

上述代码画了三条直线，都使用的是绝对坐标，起点都是(30,100)。第一条是画到(270,300)，第二条是水平画到(270,100)，第三条是垂直画到(30,300)。H 和 V 都只需要一个坐标值，如果输入了两个，则使用后一个值。结果如图 2-11 所示。

绘制三次贝塞尔曲线：

```
<path d="M30,100 C100,20 190,20 270,100
```

```
        S400,180 450,100"
style="fill:white; stroke:black; stroke-width:3 "/>
```

C 后接三个坐标，分别为两个控制点和终点；S 后接两个坐标，分别为第二个控制点和终点。S 会根据之前的曲线自动生成一个控制点。结果如图 2-12 所示，虚线的点是自动添加的控制点。

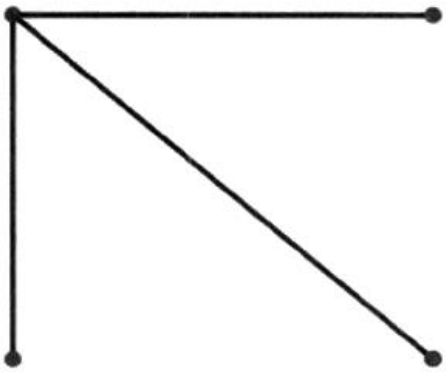


图 2-11　用路径绘制直线

图 2-12　用路径绘制三次贝塞尔曲线

绘制二次贝塞尔曲线：

```
<path d="M30,100 Q190,20 270,100
         T450,100"
   style="fill:white; stroke:black; stroke-width:3 "/>
```

结果如图 2-13 所示。

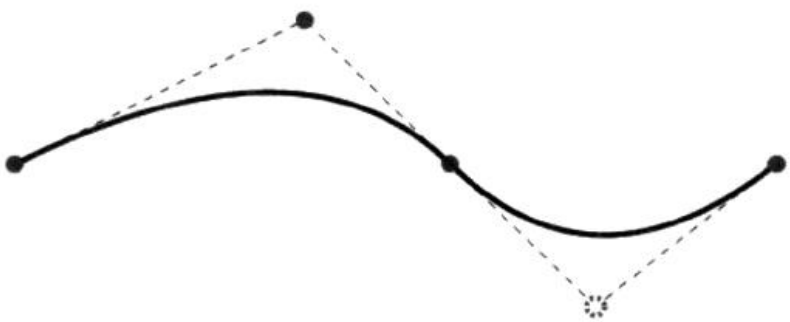


图 2-13　用路径绘制二次贝塞尔曲线

弧线是根据椭圆来绘制的，参数较多：

```
A( rx, ry, x-axis-rotation, large-arc-flag, sweep-flag, x, y )
```

- rx：椭圆 x 方向的半轴大小。
- ry：椭圆 y 方向的半轴大小。
- x-axis-rotation：椭圆的 x 轴与水平轴顺时针方向的夹角。

- large-arc-flag：有两个值（1：大角度弧线、0：小角度弧线）。
- sweep-flag：有两个值（1：顺时针至终点、0：逆时针至终点）。
- x：终点 x 坐标。
- y：终点 y 坐标。

下面的代码绘制一条弧线：

```
<path d="M100,200 a200,150 0 1,0 150,-150 Z"
      fill="yellow" stroke="blue" stroke-width="4" />
```

其中，a 用了小写英文字母，表示相对坐标，当前画笔的位置为(100,200)，所以终点位置为(100 + 150,200 - 150)，即(250,50)。包含弧线的椭圆的 x 和 y 方向的半径分别为 200 和 150，椭圆的 x 轴与水平轴方向夹角为 0°，采用大角度、逆时针方向走向终点。Z 表示闭合回路。结果如图 2-14 所示。

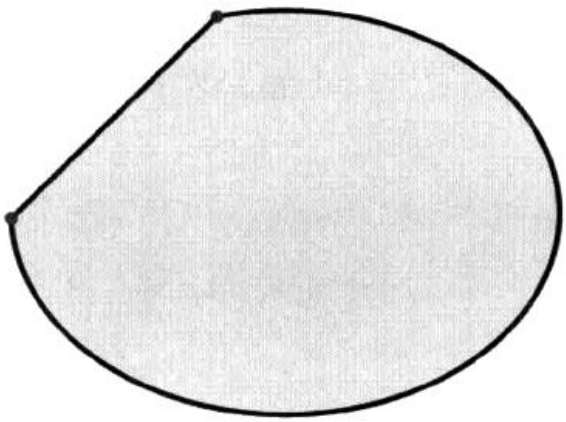


图 2-14　用路径绘制弧线

2.5.3　文字

在 SVG 中可以使用<text>标签绘制文字，其属性如下。

- x：文字位置的 x 坐标。
- y：文字位置的 y 坐标。
- dx：相对于当前位置在 x 方向上平移的距离（值为正则往右，负则往左）。
- dy：相对于当前位置在 y 方向上平移的距离（值为正则往下，负则往上）。
- textLength：文字的显示长度（不足则拉长，足则压缩）
- rotate：旋转角度（顺时针为正，逆时针为负）。

例如有以下代码：

```
<text x="200" y="150" dx="-5" dy="5" rotate="180" textLength="90" >
I love D3
</text>
```

此代码在(200,150)位置，向左偏移 5 个单位，向下偏移 5 个单位，文字顺时针旋转 180°，以文字长度为 90° 的方式绘制了文本“I love D3”。

如果要对文字中某一部分文字单独定义样式，可使用<tspan>标签。

```
<text x="200" y="150" dx="-5" dy="5" textLength="90" >
    I love <tspan fill="red">D3</tspan>
</text>
```

上述代码将<tspan>中的文字设定成红色，结果如图 2-15 所示。

I love D3

图 2-15　绘制文字

2.5.4　样式

前几个代码示例中，已经使用到了一些样式，如 fill、stroke、stroke-width 等。SVG 支持使用 CSS 选择器给元素定义样式。如：

```
.linestyle{
    stroke: red;
    stroke-width: 2;
}
```

那么在使用标签时，指定此样式即可：

```
<line class="linestyle" x1="10" y1="10" x2="100" y2="100" />
```

也可以直接在元素中写样式，有两种写法。

（1）单独写

```
<line fill="yellow" stroke="blue" stroke-width="4"
x1="10" y1="10" x2="100" y2="100" />
```

（2）合并写

```
<line style="fill:white; stroke:black; stroke-width:3
x1="10" y1="10" x2="100" y2="100" />
```

常见的样式有以下这些。

- fill：填充色，改变文字<text>的颜色也用这个。
- stroke：轮廓线的颜色。
- stroke-width：轮廓线的宽度。
- stroke-linecap：线头端点的样式，圆角、直角等。
- stroke-dasharray：虚线的样式。
- opacity：透明度，0.0 为完全透明，1.0 为完全不透明。
- font-family：字体。
- font-size：字体大小。
- font-weight：字体粗细，有 normal、bold、bolder、lighter 可选。
- font-style：字体的样式，斜体等。
- text-decoration：上画线、下画线等。

使用 CSS 选择器可以避免重复编写代码，但也有一些问题，比如将在 SVG 中绘制的图形输出到外部的 xxx.svg 文件时，使用选择器的方式可能会遇到一些问题，在第 9 章会对此进行说明。

2.5.5 标记

标记（marker）是 SVG 中一个重要的概念，能贴附于<path>、<line>、<polyline>、<polygon>元素上。最典型的应用就是给线段添加箭头。标记<marker>写在<defs></defs>中，defs 用于定义可重复利用的图形元素。先来看<marker>的各属性和意义。

- viewBox：坐标系的区域。
- refX, refY：在 viewBox 内的基准点，绘制时此点在直线端点上。
- markerUnits：标记大小的基准，有两个值，即 strokeWidth（线的宽度）和 userSpaceOnUse（线前端的大小）。
- markerWidth, markerHeight：标识的大小。
- orient：绘制方向，可设定为 auto（自动确认方向）和角度值。
- id：标识的 id 号。

各参数的意义如图 2-16 所示。

然后在<marker>里定义图形即可。下次调用这个标记，就会绘制标记里的图形。下面定义一个箭头：

```
<defs>
<marker id="arrow"
    markerUnits="strokeWidth"
```

```
    markerWidth="12"
    markerHeight="12"
    viewBox="0 0 12 12"
    refX="6"
    refY="6"
    orient="auto">
<path d="M2,2 L10,6 L2,10 L6,6 L2,2"
      style="fill: #000000;" />
   </marker>
   </defs>
```

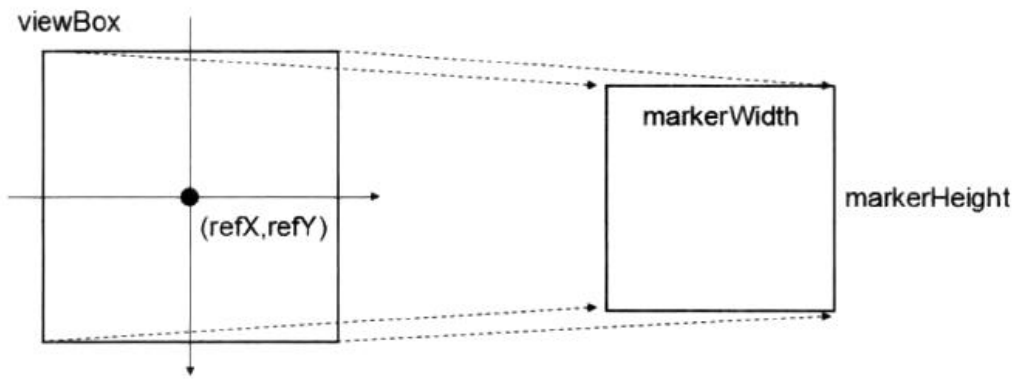


图 2-16　marker 中参数的意义

这个标记里，id 定义为 arrow，接下来在需要使用的地方使用。markerUnit 定义为 strokeWidth，表示此标记的大小以调用此标记的元素的轮廓线的大小为基准。markerWidth 和 markerHeight 分别定义为 12，在 viewBox 坐标系中绘制的图形会被自动伸缩到此尺寸。refX 和 refY 表示调用此标记的直线上的点对应于 viewBox 坐标系中的位置。orient 设定为 auto，表示箭头的方向让其自动判定。接下来在路径<path>绘制了一个箭头，里面使用的坐标，都是在 viewBox 范围内的。

下面分别绘制一条带箭头的直线和曲线：

```
<line x1="0" y1="0" x2="200" y2="50"
      stroke="red" stroke-width="2"
      marker-end="url(#arrow)"/>

<path d="M20,70 T80,100 T160,80 T200,90"
      fill="white" stroke="red" stroke-width="2"
      marker-start="url(#arrow)"
      marker-mid="url(#arrow)"
      marker-end="url(#arrow)"/>
```

其中，#arrow 表示使用 id 为 arrow 的标记。设定位置的属性如下。

- marker-start：路径起点处。
- marker-mid：路径中间端点处。
- marker-end：路径终点处。

使用 marker-mid，箭头将绘制在路径的节点处，故对于只有起点和终点的直线<line>，使用 marker-mid 无效。结果如图 2-17 所示。

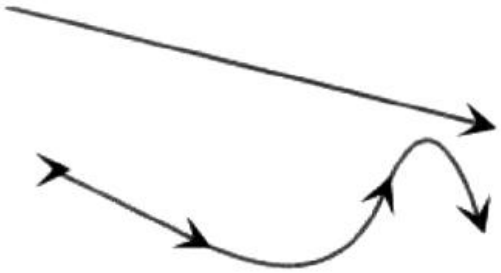
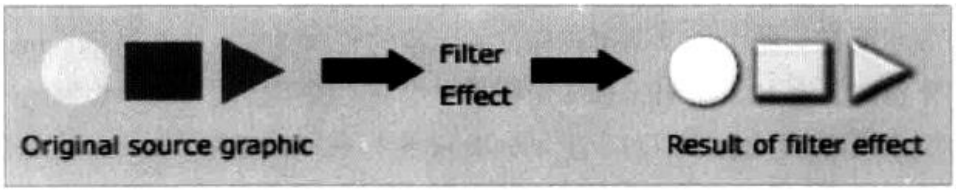


图 2-17　使用标记<marker>绘制箭头

2.5.6　滤镜

滤镜（filter）能使图形更具有艺术效果。对源图形使用滤镜能修改其显示结果。但是，滤镜不会改变源图形的数学参数，只是将其渲染后传送给显示设备，如图 2-18 所示。



* http://www.w3.org/TR/SVG11/images/filters/filters00.png

图 2-18　滤镜的作用过程

滤镜的标签为<filter>，和标记<marker>一样，也是在<defs>中定义的。滤镜的种类很多，例如 feMorphology、feGaussianBlur、feFlood 等，还有定义光源的滤镜 feDistantLight、fePointLight、feSpotLight，都是以 fe 开头的。下面列举其中一个滤镜 feGaussianBlur 的用法。

```
<defs>
<filter id="GaussianBlur">
    <feGaussianBlur in="SourceGraphic" stdDeviation="2" />
</filter>
</defs>
```

滤镜也需要设定一个 id，在图形元素上指定。在<filter>里定义滤镜，上述代码定义了一个

feGaussianBlur，一个高斯模糊的滤镜。in 是使用滤镜的对象，此处是源图形 SourceGraphic。stdDeviation 是高斯模糊唯一的参数，数值越大，模糊程度越高。使用此滤镜的代码如下。

```
<rect x="100" y="100" width="150" height="100"
      fill="blue"  />

<rect x="300" y="100" width="150" height="100"
      fill="blue" filter="url(#GaussianBlur)" />
```

上述代码分别绘制了一个没有使用滤镜的矩形和使用了滤镜的矩形，其中 filter="url(#GaussianBlur)"指定滤镜的 id，GaussianBlur 就是刚才定义的滤镜的 id。结果如图 2-19 所示。

图 2-19　使用滤镜前（左）和使用滤镜后（右）的矩形

2.5.7　渐变

有时需要在一个图形上使用渐变的颜色，渐变表示一种颜色平滑过渡到另一种颜色。SVG 中有线性渐变<linearGradient>和放射性渐变<radialGradient>。下面以线性渐变为例来讲述渐变的用法。

渐变也是定义在<defs>标签中的。给渐变定义一个 id 号，在图形元素上指定此 id 号即可。

```
<defs>
<linearGradient id="myGradient" x1="0%" y1="0%" x2="100%" y2="0%">
   <stop offset="0%" stop-color="#F00" />
   <stop offset="100%" stop-color="#0FF" />
</linearGradient>
</defs>
```

x1、y1、x2、y2 定义渐变的方向，此处是水平渐变。offset 定义渐变开始的位置，stop-color 定义此位置的颜色。然后使用此渐变：

```
<rect fill="url(#myGradient)"
      x="10" y="10" width="300" height="100"/>
```

结果如图 2-20 所示。

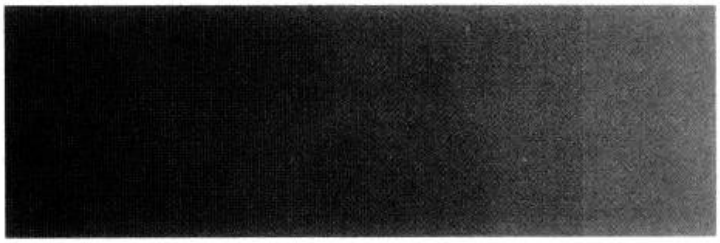


图 2-20　水平线性渐变

若将 x1、y1、x2、y2 分别变为 0%、0%、0%、100%，则可变为垂直渐变，如图 2-21 所示。

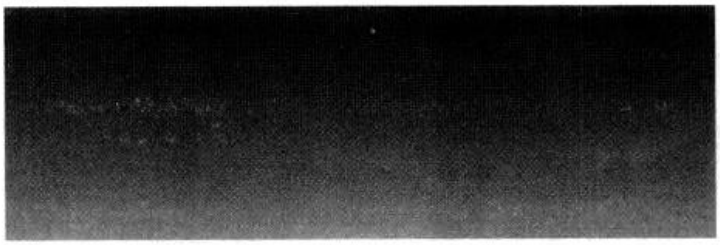


图 2-21　垂直线性渐变

第3章

安装和使用

本章内容包括：

- 安装 D3
- 搭建服务器
- 使用 D3 写一个简单程序
- 使用 D3 绘制简单矢量图

本章介绍如何使用 D3 进行开发。

第1节，讲述安装 D3 的方法。

第2节，学习搭建一个简单的服务器。

第3节，做一个简单的网页，内容是关于如何用 D3 来改变 HTML 元素的。

第4节，绘制一个简单的矢量图，画图板为 SVG。

第5节，讲述如何在浏览器上调试 JavaScript 程序。

3.1 安装

D3 类库的文件名为 d3.js，只有一个文件，所有的对象、函数、变量都写在此文件中。实际上是不需要安装的，只要在<script>中引用此文件即可。可将库文件下载下来，放在工程目录中引用，也可以直接通过网络引用。

3.1.1 下载文件

D3 的官方网站是：

http://d3js.org/

容易找到下载链接，文件名为 d3.zip。解压缩后得到以下三个文件。

- d3.js，开发项目时为了调试方便，可使用此文件。
- d3.min.js，是 d3.js 压缩后的版本，去掉了空格，体积较小，功能完全一样，浏览器的读取速度会快不少，因此发布时使用此文件。
- LICENSE，是版权许可证文件。

现将解压缩后的文件夹置于工程目录下，目录结构为：

```
folder --- d3 --- d3.js
       |      |- d3.min.js
       |--index.html
```

在 index.html 中引用 d3.js，代码如下：

```
<script src="d3/d3.js" charset="utf-8"></script>
```

属性 charset 指定解析文件的编码，选择 utf-8，因为 d3.js 中有一些特殊字符（如 π ）。请注意，指定目录时，地址“d3/d3.js”之前没有斜杠（/）。如果加了斜杠，表示在服务器根目录下的文件夹 d3 中的 d3.js；如果没加斜杠，表示当前 HTML 文件所在目录下的 d3 文件夹。如果要指定当前目录的上一层目录，路径如下：

```
../d3/d3.js
```

3.1.2 网络引用

可以直接通过网络引用文件：

```
<script src="http://d3js.org/d3.v3.min.js" charset="utf-8">
</script>
```

这种方法无须下载文件，很方便，但必须保持网络畅通才可使用。

3.2 搭建服务器

浏览器可以直接打开 HTML 文件，但是有些浏览器会限制 JavaScript 加载本地文件。如果

不搭建服务器，使用 D3 的某些请求文件的函数（如 d3.json、d3.xml 等）时会出现错误提示。经测试，IE 和 Chrome 会有此限制，而 Firefox 不具有此限制。虽然 Firefox 可以解决此问题，但是建议将网页文件都放到 Web 服务器上进行测试。

在本机安装一个简易服务器软件的方法很简单，下面就讲述如何在本机上安装 Apache Http Server。

Apache HTTP Server 可以在 http://httpd.apache.org/下载得到，其安装步骤如下。

（1）有 no_ssl 和 ssl 两种版本，no_ssl 就是用 http://…形式访问的，ssl 是用 https://…形式访问的。HTTPS 是 HTTP 的安全版，用于对安全较敏感的通信，例如网银支付等。一般来说，用 no_ssl 就好了。本次下载的安装包文件名为：

httpd-2.2.25-win32-x86-no_ssl.msi

（2）打开安装包，单击几次“Next”按钮后，出现如图 3-1 所示的界面，要求输入 Network Domain（网络域名）、Server Name（服务器名）和管理员邮箱。随意即可，安装好之后也是可以改的。

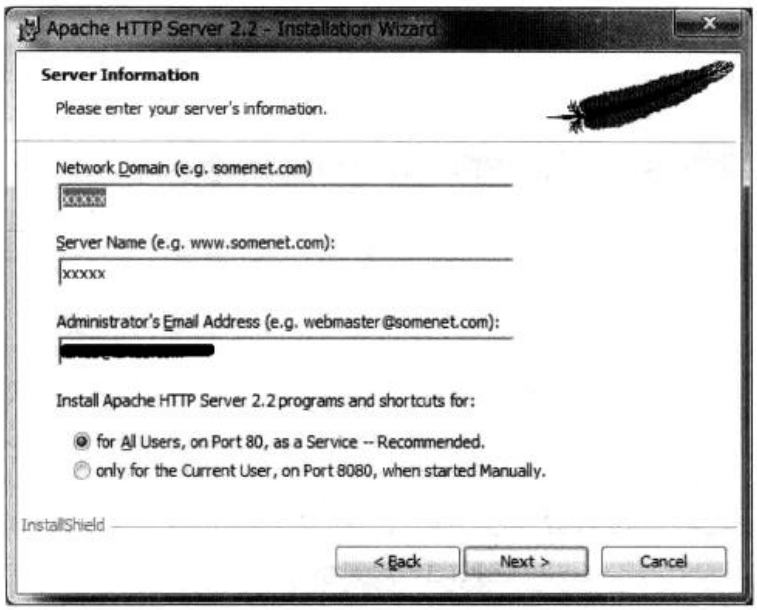


图 3-1　输入域名、服务器名、邮箱地址

（3）然后需要选择“Typical（典型）”还是“Custom（自定义）”安装方式。选择“Custom”安装方式后会出现如图 3-2 所示的窗口，把能安装的都勾选上后单击“Next”按钮。

（4）等待安装结束后，单击“Finish”按钮即可。

（5）安装好后，任务栏的右下角会出现一个图表。右键单击图表，选择“Open Apache Monitor”命令，出现如图 3-3 所示的窗口，可以单击“Start”、“Stop”、“Restart”等按钮，默认已经开启了，如果没有，单击“Start”按钮开启服务。

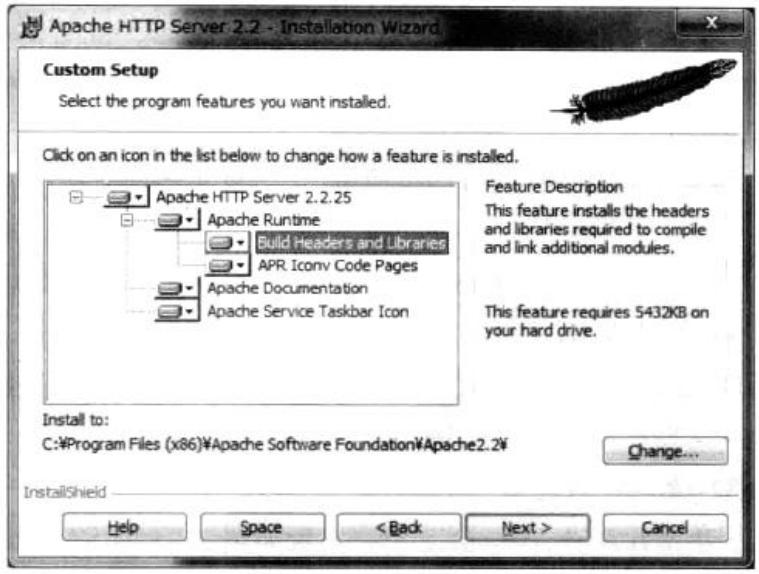


图 3-2 选择安装项

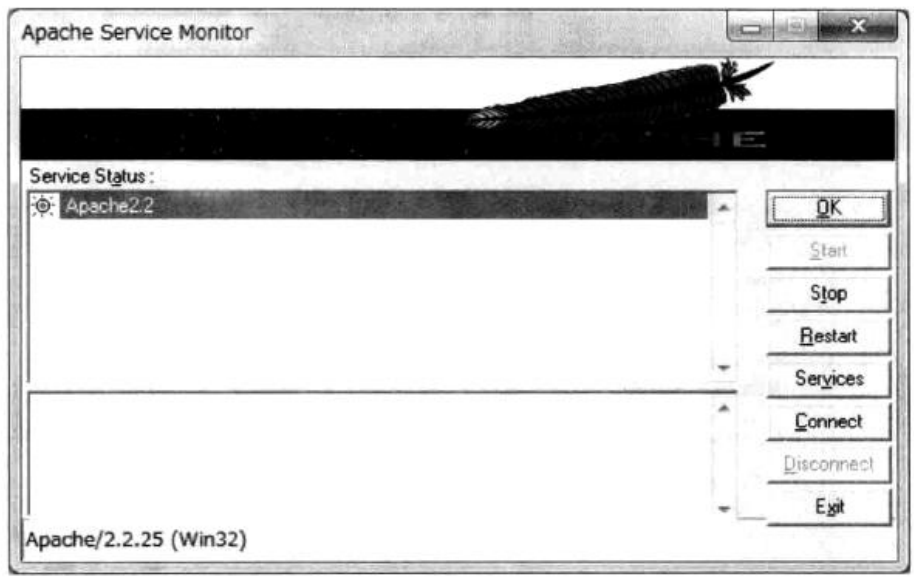


图 3-3 服务控制台

（6）打开安装目录 C:\Program Files (x86)\Apache Software Foundation\Apache2.2，文件目录结构如图 3-4 所示，其中“htdocs”为默认的根目录文件夹，把需要的各种网页文件放到此处即可。如果想要更改此目录的位置，在“conf”文件夹里，有一个名为 httpd.conf 的文件，打开后可以设定包括根目录等选项。

（7）测试一下安装是否成功，打开浏览器，输入：http://127.0.0.1/ 或 http://localhost/。如果出现如图 3-5 所示的页面，就是成功了。

这个页面是位于根目录 htdocs 下的 index.html 文件。开发 D3 项目时，将项目文件夹放入根目录，再在浏览器中的地址栏输入：

```
http://127.0.0.1/d3project/xxx.html
```

d3project 为文件夹名。如此即可解决浏览器限制 JavaScript 加载本地文件的问题。

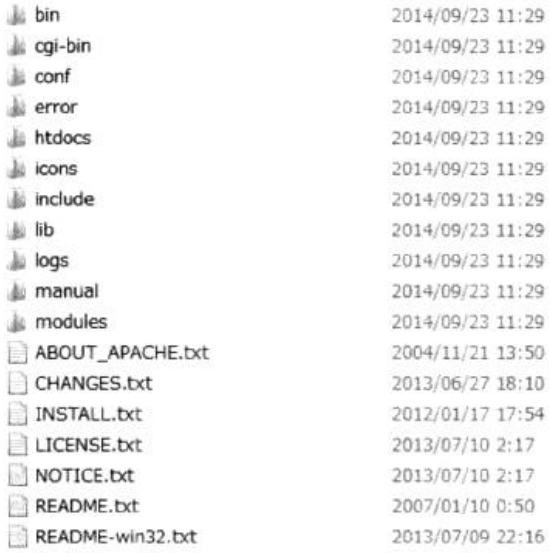


图 3-4　文件目录结构

图 3-5　测试安装结果

3.3　Hello, World

学习编程语言的第一个程序大多数是在屏幕上输出“Hello, World”字符串，因为简洁实用，已成为惯例。不过为了体验 D3 的功能，将会稍微扩展此惯例。首先用 HTML 输出一字符串，然后分别使用 JavaScript 和 D3 将字符串修改为“Hello, World”。

现有的 HTML 代码如下：

```
<body>
<p>Cat</p>
<p>Dog</p>
</body>
```

以上代码的结果，是输出两个段落，段落文字分别为 Cat 和 Dog。用 JavaScript 将两个段落的文字更改为“Hello, World”的代码为：

```
<script>
var paragraphs = document.getElementsByTagName("p");
for (var i = 0; i < paragraphs.length; i++) {
    var paragraph = paragraphs.item(i);
    paragraph.innerHTML = "Hello, World";
}
```

```
</script>
```

用 D3 将两个段落更改为“Hello, World”的代码为：

```
<script src="http://d3js.org/d3.v3.min.js" charset="utf-8">
</script>
<script>
d3.select("body")                    //选择<body>
   .selectAll("p")                   //选择<body>里所有的<p>
   .text("Hello, World");            //将文字更改为Hello, World
</script>
```

记得导入 d3 库文件。可以看到，与 JavaScript 代码的详细相比，D3 代码十分简洁，没有 for 循环，能一次性操作所有元素。接下来，添加些许内容。

改变两段落的样式，将段落文字的颜色设置为红色，大小设置为 72px。

```
var p = d3.select("body")
          .selectAll("p")
          .text("Hello, World");
p.style("color","red");          //将文字颜色更改为红色
p.style("font-size","72px");     //将文字大小更改为72px
```

将选择集赋值给变量 p，于是 p 就代表<body>中的所有<p>。接下来，在修改选择集的样式（如颜色，字体大小）时，就可以直接用变量 p。

3.4 绘制矢量图

一般来说，D3 是在 SVG 画板上作图的。虽然也可以在 Canvas 等上作图，但是 D3 提供了大量 SVG 图形的路径生成器，说 D3 是为 SVG 量身定制的也不为过。

首先，在 HTML 的<body>标签中添加<svg>标签，然后将所有的图形元素都添加在<svg>中。

```
var width  = 400;                              //SVG绘制区域的宽度
var height = 400;                              //SVG绘制区域的高度
var svg = d3.select("body")                    //选择<body>
            .append("svg")                     //在<body>中添加<svg>
            .attr("width", width)              //设定<svg>的宽度属性
            .attr("height", height);           //设定<svg>的高度属性
```

append()用于添加元素，attr()用于给指定属性赋值。append 函数返回一个选择集，上述代码中将该选择集赋值给了变量 svg。然后可在选择集中添加图形元素，例如圆<circle>。

```
svg.append("circle")          //在<svg>中添加<circle>标签
   .attr("cx","50px")         //设置属性cx的值为50px
```

```
    .attr("cy","50px")  //设置属性cy的值为50px
    .attr("r", "50px")  //设置属性r的值为50px
    .attr("fill","red");//设置属性fill的值为red
```

在坐标(50,50)，添加一个半径为 50px、颜色为红色的圆形，结果如图 3-6 所示。按“F12”键弹出的开发者工具窗口中，可以看<svg>里有一组<circle>。

图 3-6　绘制圆

3.5　调试

图 3-6 有一个窗口，其中包含 HTML 页面元素，以及“Console（控制台）”等选项卡。此窗口是为了让开发者调试程序的，在浏览器界面按“F12”键即可弹出。

一般来说，调试的时候需要在代码中加入若干输出信息，以便开发者确认变量中的值是否正确、错误信息是什么等。输出信息的方法，常用的有两种。

1．输出到控制台

函数 console.log()最常用，格式如下：

```
console.log(error);
```

在控制台窗口中会输出变量 error 的内容，如图 3-7 所示。

2．弹出窗口提示错误

函数 alert()的格式如下：

```
alert(error);
```

浏览器会弹出窗口，如图 3-8 所示。

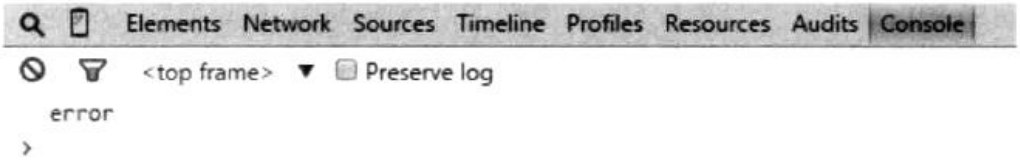


图 3-7　在控制台输出错误信息

图 3-8　弹出窗口显示错误信息

建议使用 console.log，相对来说更方便，可同时查看多个输出。后续章节中，常常要将数据进行转换，而对于开发者来说知道转换后的数据是怎样的后才能使用。因此，在控制台输出错误信息的方式会经常出现。

第 4 章

选择集与数据

本章内容包括：

- 如何选择元素
- 选择集的状态和属性
- 数据绑定
- 选择集的处理
- 操作数组
- 柱形图

选择集和数据，是 D3 最重要的基础。选择集是被选择的元素集合，例如所有的<p>；数据是可视化的来源，D3 允许将数据和选择集绑定在一起，以凭借数据操作选择集。

第 1 节，讲述如何选择元素。

第 2 节，介绍如何查看选择集的状态，以及设定和获取选择集的属性。

第 3 节，讨论对应一个选择集，如何添加、插入和删除元素。

第 4 节，学习如何将数据绑定到选择集上，极为重要。

第 5 节，讨论当数据与选择集绑定在一起后，该如何处理。

第 6 节，介绍数组的处理方法，可视化数据主要是由数组构成的。

第 7 节，制作一个简单的柱形图，以复习本章的知识。

4.1 选择元素

D3 中，选择元素的函数有两个：select 和 selectAll。两种方式都很常用，它们的区别如下。

- select：返回匹配选择器的第一个元素。
- selectAll：返回匹配选择器的所有元素。

选择器是 select 和 selectAll 的参数，指定应当选择文档中的哪些元素。此处的选择器，指的是 CSS 选择器，第 2.2.2 节讲述了选择器的相关内容。知道了选择器的表示方式，即可用如下代码来选择所需元素。

```
d3.select("body");                    //选择body元素
d3.select("#important");              //选择id为important的元素
d3.select(".content");                //选择类为content的第一个元素
```

要注意，符合选择器的可能有多个元素，但 select 只选择第一个。如果要选择符合选择器的所有元素，使用 selectAll，代码如下。

```
d3.selectAll("p");                    //选择所有的p元素
d3.selectAll(".content");             //选择类为content的所有元素
d3.selectAll("ul li");                //选择ul中所有的li元素
```

关于 select 和 selectAll 的参数，除了 CSS 选择器，还可以是已经被 DOM API 选择的元素，代码如下。

```
var important = document.getElementById("important");
d3.select(important);
```

上面的代码中，变量 important 保存的是被 getElementById 选择的元素，将 important 作为参数 select 的参数即可。但是，此方法有一个问题：如果选择的是多个元素，将其作为 select 的参数，不能达到选择其中第一个元素的效果。这种情况下，应将其作为 selectAll 的参数。请看下面的示例。

```
//使用DOM选择类为content的元素集
var content = document.getElementsByClassName("content");

//如果使用select，达不到选择第一个元素的效果
//d3.select(content);

//这种使用方法是正确的
d3.selectAll(content);
```

基本上，使用 getElementById 选择的元素要用 select，使用 getElementsByClassName 选择的元素要使用 selectAll。但是，尽量直接使用 CSS 选择器作为参数，简单易懂。

如果要对选择集中的元素再进行一番选择，例如选择 body 中所有的 p 元素，除了使用 CSS 的派生选择器作为参数之外，还可采用如下手法。

```
d3.select("body").selectAll("p");
```

熟悉 jQuery 的朋友一定对此用法很有好感，D3 也支持这种连续调用函数的方法，称为“连缀语法”。

4.2　选择集

d3.select 和 d3.selectAll 返回的对象称为**选择集**（**selection**），添加、删除、设定网页中的元素，都得使用选择集。

4.2.1　查看状态

查看选择集的状态，有三个函数可供使用。

- selection.**empty**()

如果选择集为空，则返回 true；如果不为空，返回 false。

- selection.**node**()

返回第一个非空元素，如果选择集为空，返回 null。

- selection.**size**()

返回选择集中的元素个数。

以上三个函数，都是对选择集说的，请看如下代码。

```
<p> Paragraph 1 </p>
<p> Paragraph 2 </p>
<p> Paragraph 3 </p>
<script>
    var paragraphs = d3.selectAll("p");
    console.log( paragraphs.empty() );      //false
    console.log( paragraphs.node() );       //<p> Paragraph 1 </p>
    console.log( paragraphs.size() );       //3
</script>
```

d3.selectAll 选择了文档中所有的 p 元素，返回一个选择集，保存到变量 paragraphs。其后，分别调用 empty()、node()和 size()，返回结果输出到控制台。输出结果是三行文字，分别为：“false”、“ Paragraph 1 ”和“3”。

4.2.2　设定和获取属性

假设 HTML 文档中有一个段落元素 ，如下：

```
<p> This is a paragraph </p>
```

如果要给此元素设置一个 id，可直接写在 HTML 元素上，代码如下：

```
<p id="para"> This is a paragraph </p>
```

换成用 D3 来修改元素 p 的属性，只需：

```
d3.select("p").attr("id","para");
```

包括上面的 attr()，设定或获取选择集属性的函数共有六个，以下分别介绍。

- selection.**attr**(name[, value])

设置或获取选择集的属性，name 是属性名称，value 是属性值。如果省略 value，则返回当前的属性值；如果不省略，则将属性 name 的值设置为 value。

应用此函数后，添加的属性在元素标签中以 name="value"的形式出现。即便是不存在的属性，也可以添加，只是没有任何效果。第 3.4 节，给 svg 添加 circle 元素后，也使用了此函数：

```
svg.append("circle")          //在<svg>中添加<circle>标签
   .attr("cx","50px")         //设置属性cx的值为50px
   .attr("cy","50px")         //设置属性cy的值为50px
   .attr("r", "50px")         //设置属性r的值为50px
   .attr("fill","red");       //设置属性fill的值为red
```

其结果是，circle 元素添加了如下属性：

```
<circle cx="50px" cy="50px" r="50px" fill="blue"></circle>
```

如果要获取选择集某属性的值，则省略第二个参数 value 即可：

```
var cx = d3.select("circle").attr("cx");
console.log(cx);          //50px
```

- selection.**classed**(name[, value])

设定或获取选择集的 CSS 类，name 是类名，value 是一个布尔值。布尔值表示该类是否开启。selection.attr()是可以设置类的，但在设置多个类时不方便，例如：

```
d3.select("p")
   .attr("class","red bigsize");   //类名间用空格隔开
```

给选择集设置了 red 和 bigsize 两个类。但是，如果需要根据判断条件来开启和关闭，attr()就办不到了，而 classed 的第二个参数正是一个判断条件：

```
d3.select("p")
   .classed("red",true)              //开启red类
   .classed("bigsize",false);        //不开启bigsize类
```

当布尔值为 true 时，开启对应的类，元素标签添加 class="classname"。当布尔值为 false 时，标签中不会添加任何属性。另外，classed 还有以下形式：

```
.classed({ "red": true , "bigsize":true });    //写在一个对象里
.classed("red bigsize",true);                  //用空格分开写在一起
```

如果省略第二个参数 value，则返回一个布尔值，表示类是否开启：

```
//输出true或false
console.log( d3.select("p").classed("bigsize") );
```

- selection.**style**(name[, value[, priority]])

设定或获取选择集的样式，name 是样式名，value 是样式值：

```
d3.select("p")
    .style("color","red")
    .style("font-size","30px");
```

则元素标签将添加如下属性：

```
<p style="color: red; font-size: 30px;"></p>
```

与 classed 一样，可以用对象的形式，将几个样式写在一起：

```
.style({"color":"red" , "font-size":"30px"});
```

如果只保留第一个参数，则返回该样式的值。

- selection.**property**(name[, value])

设定或获取选择集的属性，name 是属性名，value 是属性值。如果省略 value，则返回属性名。

有部分属性，不能用 attr()设定和获取，最典型的是文本输入框的 value 属性，此属性值不会在标签中显示。例如，有文本框元素：

```
<input id="fname" type="text" name="fullname"/>
```

在文本框中输入文本，例如字符串 ZhangSan。但是，标签中并不会添加 value="ZhangSan"，因此，不能使用 attr()来获取属性值，而要用 property()：

```
d3.select("#fname").property("value");
```

返回结果是字符串 ZhangSan。使用第二个参数，即可给文本框赋值：

```
d3.select("#fname").property("value","LiSi");
```

此外，还有复选框等，都需要用 property()来获取属性。总之，凡是不能用 attr()来处理的属性，都可考虑用 property()。

- selection.**text**([value])

设定或获取选择集的文本内容，如果省略 value，则返回当前的文本内容。文本内容相当于 DOM 的 innerText，不包括元素内部的标签：

```
//HTML中的元素，标签里包含着标签
<p> This <span>is</span> a paragraph </p>

//选择p后，调用text()函数，将返回值在控制台中输出
console.log( d3.select("p").text() );
```

输出结果为 This is a paragraph，不包含其中的 span 标签。

- selection.**html**([value])

设定或获取选择集的内部 HTML 内容，相当于 DOM 的 innerHTML，包括元素内部的标签：

```
<p> This <span>is</span> a paragraph </p>
console.log( d3.select("p").html() );
```

输出结果为 This is a paragraph，包含其中的 span 标签。

4.3 添加、插入和删除

对于选择集，可以添加、插入和删除元素，相关函数的介绍如下。

- selection.**append**(name)

在选择集的末尾添加一个元素，name 为元素名称。

- selection.**insert**(name[, before])

在选择集中的指定元素之前插入一个元素，name 是被插入的元素名称，before 是 CSS 选择器名称。

- selection.**remove**()

删除选择集中的元素。

下面的代码，包含了以上三个函数的用法：

```
<body>
    <!-- body中的三个段落元素 -->
    <p>Car</p>
    <p id="plane">Plane</p>
    <p>Ship</p>

    //JavaScript代码块
    <script>
```

```
        //选择body元素
        var body = d3.select("body");

        //在body中所有元素的末尾处添加一个p元素，内容为Train
        body.append("p").text("Train");

        //在body中id为plane的元素前添加一个p元素，内容为Bike
        body.insert("p","#plane").text("Bike");

        //选择id为plane的元素
        var plane = d3.select("#plane");
        //删除id为plane的元素
        plane.remove();

    </script>
</body>
```

这段代码的结果是在网页上显示四个段落，文字和排列次序分别为：Car、Bike、Ship、Train。添加删除元素的过程如图 4-1 所示。

首先，使用 append()给 body 的末尾添加了一个 p 元素，内容为 Train，这时页面中有四个段落元素，按次序排列分别为：Car、Plane、Ship、Train。

然后，调用 insert()在 Plane 前插入了一个 p 元素，内容为 Bike，这时页面中有五个段落元素，按次序排列分别为：Car、Bike、Plane、Ship、Train。

最后，调用 remove()将 Plane 删除，故最终结果为：Car、Bike、Ship、Train。

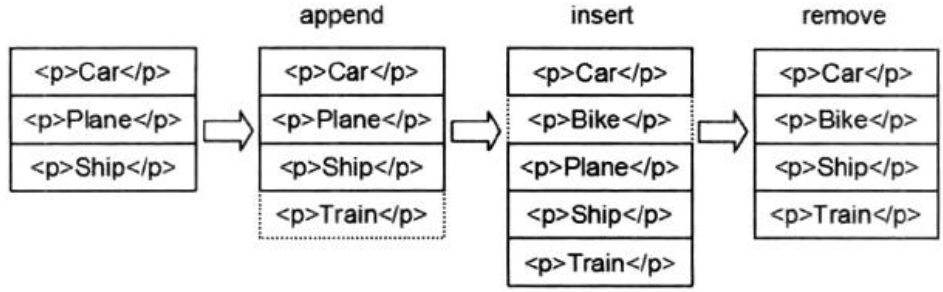


图 4-1　添加删除元素的过程

上面代码中，insert()的第二个参数是#plane，这是 CSS 选择器，任何符合选择器语法的字符串都可作为第二个参数的值。

4.4　数据绑定

将数据绑定到 DOM 上，是 D3 最大的特色。d3.select 和 d3.selectAll 返回元素的选择集，

选择集上是没有数据的。数据绑定，就是使被选择元素里“含有”数据。相关函数有两个。

- selection.**datum**([value])

选择集中的每一个元素都绑定相同的数据 value。

- selection.**data**([values[, key]])

选择集中的每一个元素分别绑定数组 values 的每一项。key 是一个键函数，用于指定绑定数组时的对应规则。

上面的概念可能难以理解，不要着急，接下来会详细剖析 datum()和 data()，来看看数据绑定是如何工作的，两个函数有什么不同之处。不理解原理，直接使用或许也不错，但遇到意料之外的结果时，往往不知道如何调试。因此，大致知道其工作过程是必要的，而且本节内容对于想要学好 D3 太重要了，笔者建议一定要细心阅读。

4.4.1 datum()的工作过程

datum()绑定数据的方法很简单，可能使用得比较少，某些时候还是能派上大用场的。并且，它能帮助理解 D3 是如何绑定数据到选择集上的。请先看下面的代码。

```
<body>
    <!-- 三个段落元素 -->
    <p>Fire</p>
    <p>Water</p>
    <p>Wind</p>

    <script>
        //选择body中所有的p元素，选择集结果赋值给变量p
        var p = d3.select("body").selectAll("p");

        //绑定数值7到选择集上
        p.datum(7);

        //在控制台输出选择集
        console.log(p);
    </script>
</body>
```

代码中，使用 datum()将数值 7 绑定到了选择集上，然后在控制台输出该选择集。在浏览器的控制台，可以看到如图 4-2 所示的输出结果。其中包含有三个 p 元素，正是使用 selectAll 选择的三个段落，还可看到选择集的大小（length）、父节点（parentNode）等信息。

展开任意一个 p 元素，其各属性值如图 4-3 所示。注意画圈的部分，有一个__data__属性，这是被绑定数据之后才会出现的，其数值 7 正是刚才绑定的数据。展开其他的 p 元素，会发现

每一个元素中都多出了一个__data__，并且数值都是 7。

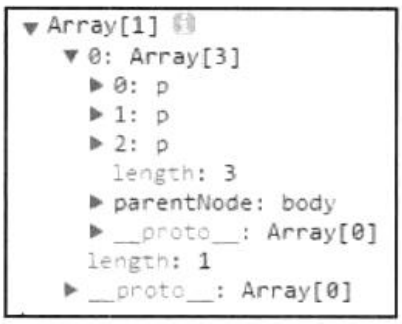


图 4-2　被输出到控制台的选择集

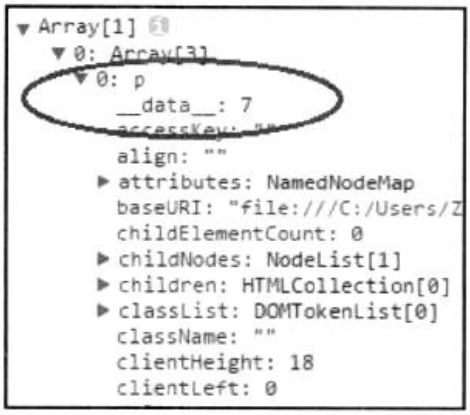


图 4-3　绑定到元素上的数据

那么 datum()的工作过程就再明白不过了，即对于选择集中的每一个元素，都为其增加一个__data__属性，属性值为 datum(value)的参数 value。此处的 value 并非一定要是 number（数值）型，也可以是 string（字符串）、boolean（布尔型）和 object（对象），无论是什么类型，其工作过程都是给__data__赋值。如果使用 undefined 和 null 作为参数，则将不会创建__data__属性。下面来看看 datum()的源代码：

```
d3_selectionPrototype.datum = function(value) {
    return arguments.length
        ? this.property("__data__", value)
        : this.property("__data__");
};
```

由以上源代码可知，datum()是用第 4.2.2 节中提到的 property()函数实现的：如果有参数 value，则调用 property 给当前对象添加一个__data__属性；否则返回__data__属性值。下面来试验一下没有参数的情形。

```
var p = d3.select("body").selectAll("p");
p.datum(7);
console.log( p.datum() );        //在控制台输出被绑定的数据
```

这段代码将在控制台输出数字 7，正是被绑定的数据。现在有一个问题，数据被绑定在选择集上后，该如何使用呢，或者说 D3 希望我们如何使用呢？举一个例子，用被绑定的字符串，替换掉段落原来的字符串。

```
<body>
    <p>Fire</p>
    <p>Water</p>
    <p>Wind</p>
    <script>
```

```
        var p = d3.select("body").selectAll("p");
        p.datum("Thunder")          //绑定字符串Thunder到选择集上
          .text(function(d,i){  //替换内容
                return d + " " + i;
        });
    </script>
</body>
```

datum()绑定了一个字符串 Thunder 到选择集上，然后调用 text()替换字符串。text()可设定选择集的文本内容。在这里，text()的参数是一个无名函数 function(d,i)，这两个参数分别代表**数据（datum）**和**索引（index）**。其实，无名函数两个参数的名称不一定要用 d 和 i，但意义是不变的。按照惯例，建议写成 d 和 i。

最后，无名函数返回了由 d 和 i 结合而成的字符串，中间加一空格。结果如图 4-4 所示，网页中的三个段落元素 p 的字符串分别被替换成：Thunder 0、Thunder 1、Thunder 2。

由结果也可得知，无名函数的两个参数 d 和 i，d 是被绑定的字符串，i 是索引号。索引号是从 0 开始的。D3 中，使用被绑定的数据大多数用上述形式：**定义一个无名函数 function(d,i)，在函数体中使用 d 和 i。**

D3 还有一个特性，能使被绑定的数据传递给子元素。对前一段代码稍做修改：

```
p.datum("Thunder")
    .append("span") //在每一个被选择元素后添加元素span
    .text(function(d,i){
        return " " + d;
    });
```

结果如图 4-5 所示，各段落元素的末尾被添加了 span 元素，内容为 Thunder。

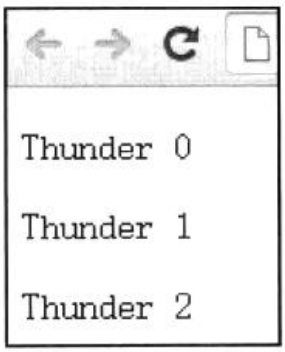


图 4-4　使用被绑定的数据

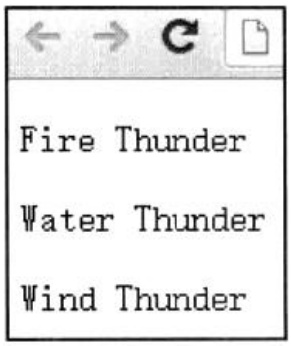


图 4-5　在各段落元素中添加 span 子元素

下面使用 console.log 在控制台输出选择集 p，如图 4-6 所示。可以看到，子元素 span 里也含有属性__data__，属性值仍然是字符串 Thunder。据此可得出一个结论：

在被绑定数据的选择集中添加元素后，新元素会继承该数据。

这一点后续章节中会很常用。

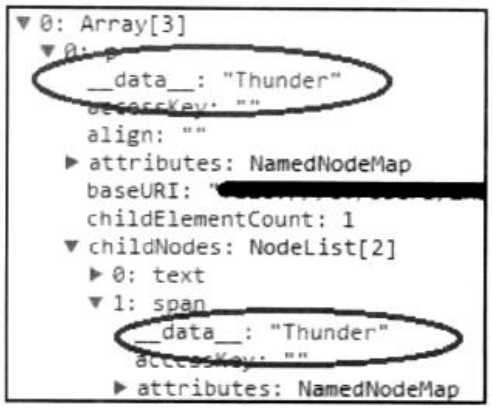


图 4-6　将被绑定的数据传递给子元素

4.4.2　data()的工作过程

data()能将数组各项分别绑定到选择集的各元素上，并且能指定绑定的规则。当数组长度与元素数量不一致时，data()也能够处理。当数组长度大于元素数量时，为多余数据预留元素位置，以便将来插入新元素；当数组长度小于元素数量时，能获取多余元素的位置，以便将来删除。下面剖析 data()是如何绑定数据的，与 datum()有什么不同。

假设 body 中有三个段落元素 p，代码如下：

```
<body>
    <p>Lion</p>
    <p>Tiger</p>
    <p>Leopard</p>
</body>
```

现要将一个数组的各项分别绑定到各元素上。假设，数组为[3,6,9]，令第一个 p 元素绑定 3，第二个绑定 6，第三个绑定 9，这种情况就需要使用 data()。如果是 datum()，则会将数组本身绑定到各元素上，即第一个 p 元素绑定[3,6,9]，第二个绑定[3,6,9]，第三个也是绑定[3,6,9]，data()和 datum()的区别如图 4-7 所示。

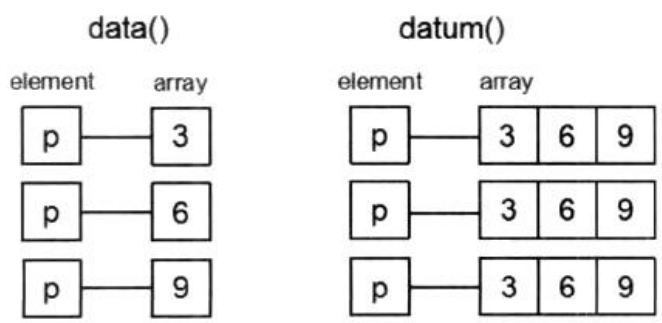


图 4-7　data()和 datum()的区别

使用 data()绑定数据的代码如下：

```
//定义数组
var dataset = [3, 6, 9];

//选择body中的p元素
var p = d3.select("body").selectAll("p");

//绑定数据到选择集
var update = p.data(dataset);

//输出绑定的结果
console.log(update);
```

这段代码中，将数组绑定到选择集并输出结果。如图 4-8 所示，数组的三项分别被绑定到了各元素上，与图 4-7 的预测一致。

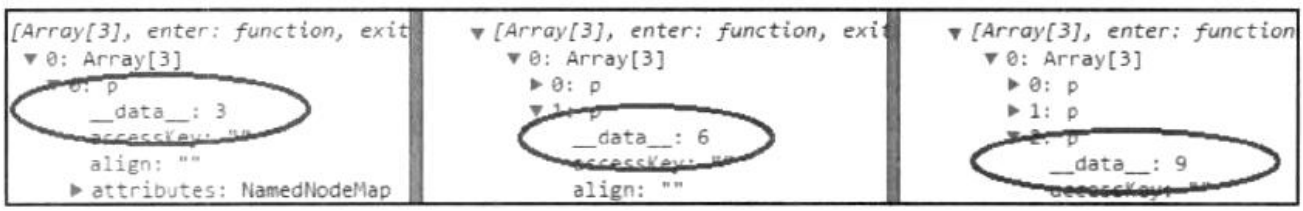


图 4-8　使用 data()将数组分别绑定到各元素

上例中，数组长度与元素数量正好相等。两者也有不相等的情况，如果数组长度为 5，而元素数量为 3，则多出两个数据没有绑定元素。如果数组长度为 1，元素数量为 3，则会有两个元素没有绑定数据。在 D3 中，根据数组长度和元素数量的关系，分别把各种情况归纳如下。

- update：数组长度 = 元素数量。
- enter：数组长度 > 元素数量。
- exit：数组长度 < 元素数量。

这三个单词的含义可能很难理解，update 的原意为“更新”，enter 的原意为“进入”，exit 的原意为“退出”，直译的结果很难表现所需的意思。一般来说，在读入数据进行可视化的过程中，被读入的数据都要绑定到选择集的元素上，没有绑定数据的元素是没有用的。因此，这三个单词可以理解为：

- 如果数组长度大于元素数量，则部分还不存在的元素“即将进入可视化（enter）”。
- 如果数组长度小于元素数量，则多余的元素“即将退出可视化（exit）”。
- 如果数组长度等于元素数量，则绑定数据的元素“即将被更新（update）”。

还是很难理解吗？没关系，请看图 4-9，其中左图表示数组长度为 5、元素数量为 3 的情况，那么有两个数组项没有与之相连的元素，这一部分被称为 enter；其中右图表示数组长度为 1、元素数量为 3 的情况，那么有两个元素没有数组项与之相连，这一部分称为 exit。数组项和元

素相连的部分被称为 update。

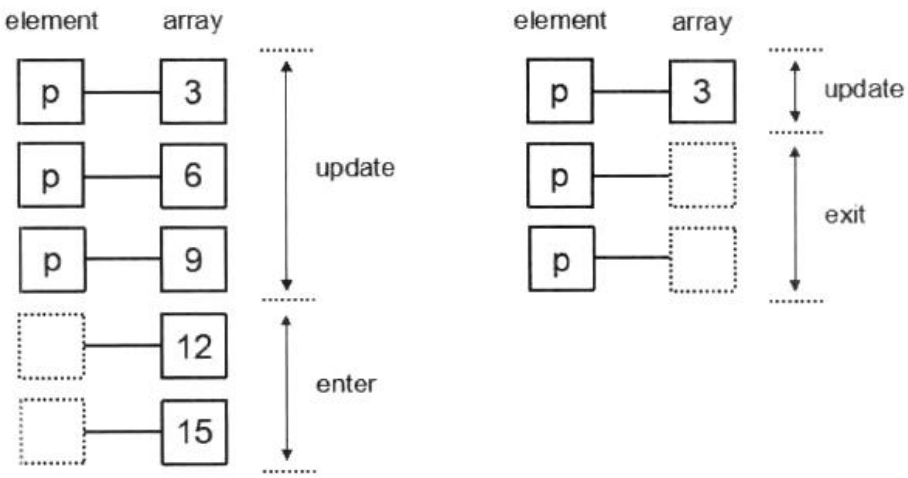


图 4-9　当数组长度大于元素数量时，update 和 enter 所代表的范围（左）；当数组长度小于元素数量时，update 和 exit 所代表的范围（右）

data()返回一个对象，对象里包含 update 部分，还有两个方法：一个是 enter()，返回 enter 部分；一个是 exit()，返回 exit 部分。请看如下代码：

```
var dataset = [3, 6, 9, 12, 15];
var p = d3.select("body").selectAll("p");
var update = p.data(dataset);
console.log(update);
console.log(update.enter());
console.log(update.exit());
```

上面的代码中，数组长度为 5，元素数量为 3，多出两个数组项。其输出结果如图 4-10 和图 4-11 所示。

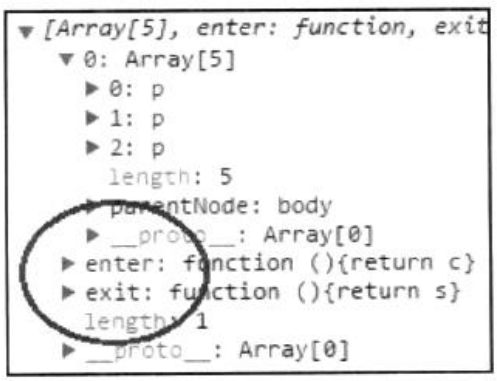


图 4-10　console.log(update)的输出结果

图 4-10 中，可以看到被绑定数据的三个 p 元素。还有 enter()和 exit()两个函数，用于返回

本次绑定的 enter 和 exit 部分。

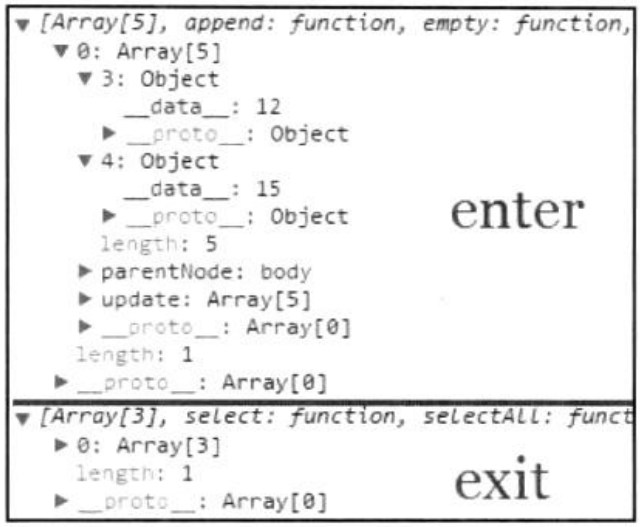


图 4-11　console.log(update.enter())和 console.log(update.exit())的输出结果

由图 4-11 可以看到，enter 部分中，D3 已经为多余的数组项 12 和 15 预留了位置，以备将来添加元素。enter 部分中，还有一个变量 update，指向 update 部分。本次绑定中，没有多余的元素，所以 exit 部分没有内容。如果将数组换成：

```
var dataset = [3];
```

则 exit 部分的输出结果如图 4-12 所示，多出两个 p 元素。

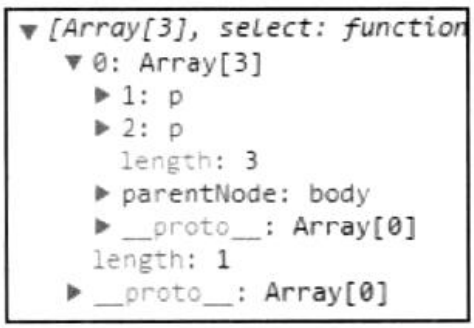


图 4-12　数组长度小于元素数量时的 exit 部分

update、enter、exit 是非常重要的概念，可能一时难以理解，在后续章节中还会大量出现。

4.4.3　绑定的顺序

默认情况下，data()是按索引号顺序绑定的：第 0 个元素绑定数组第 0 项，第 1 个元素绑定数组第 1 项，依此类推。也可以不按此顺序进行，这就要用到 data()的第二个参数。此参数是一个函数，称为**键函数**（key function）。

要注意，**只有在选择集原来已经绑定有数据的情况下，使用键函数才有效果**。请看以下示例。

```
<body>
    <!-- 三个空的p元素 -->
    <p></p>
    <p></p>
    <p></p>
    <script>
    //数据
    var persons = [ { id: 3 , name:"张三" },
                    { id: 6 , name:"李四" },
                    { id: 9 , name:"王五" }];

    //选择body中的所有p元素
    var p = d3.select("body").selectAll("p");

    //绑定数据，并修改p元素的内容
    p.data(persons)
        .text(function(d){
            return d.id + " : " + d.name;
        });
    </script>
</body>
```

这段代码对 p 元素的内容进行了修改，修改之后的 p 元素为：

```
<p>3 : 张三</p>
<p>6 : 李四</p>
<p>9 : 王五</p>
```

下面将 persons 里的数据更新，再绑定一次数据。本次绑定添加键函数：

```
//更新persons里的数据
persons = [     { id: 6 , name:"张三" },
                { id: 9 , name:"李四" },
                { id: 3 , name:"王五" }];

//根据键函数的规则绑定数据，并修改内容
p.data(persons, function(d){ return d.id; })
        .text(function(d){
            return d.id + " : " + d.name;
        });
```

键函数里只有一个语句 return d.id，表示使用数组项的 id 作为**键**。使用本次绑定的数据修改 p 元素的内容后，结果如下：

```
<p>3 : 王五</p>
<p>6 : 张三</p>
<p>9 : 李四</p>
```

可以看到，结果并没有按照新 persons 数组的次序（6：张三、9：李四、3：王五）排列。绑定过程如图 4-13 所示，绑定的顺序不按照索引号绑定，而是使**键**值依次对应。

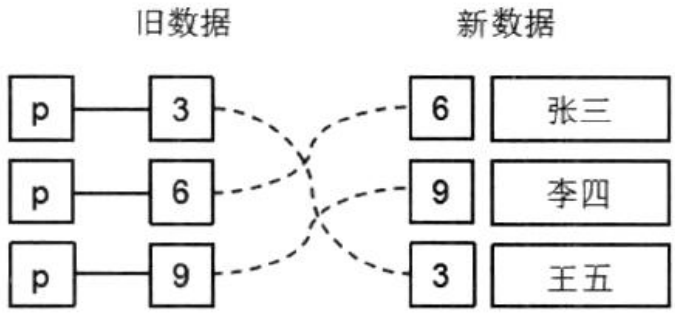


图 4-13　根据键函数绑定数据

4.5　选择集的处理

上一节讲述了绑定数据的原理。当数组长度与元素数量不一致时，有 enter 部分和 exit 部分，前者表示存在多余的数据，后者表示存在多余的元素。本节将介绍如何处理这些多余的东西，然后讲解一个处理模板，此模板包含处理 enter、exit、update 部分的内容。

4.5.1　enter 的处理方法

如果没有足够的元素，那么处理方法就是添加元素。请看下面的代码：

```
<body>
    <p></p>
    <script>
    var dataset = [3, 6, 9];
    var p = d3.select("body").selectAll("p");

    //绑定数据后，分别获取update和enter部分
    var update = p.data(dataset);
    var enter = update.enter();

    //update部分的处理方法是直接修改内容
    update.text( function(d){ return d; } );
```

```
    //enter部分的处理方法是添加元素后再修改内容
    enter.append("p")
        .text(function(d){ return d; });
    </script>
</body>
```

本例中，body 中的 p 元素只有一个，但是数据有三个，因此 enter 部分有多余的两个数据。对多余数据的处理方法就是添加元素（append），使其与多余数据对应。处理之后，body 里有三个 p 元素，内容分别为：

```
<p>3</p>
<p>6</p>
<p>9</p>
```

通常，从服务器读取数据后，网页中是没有与之对应的元素的。因此，有一个很常见的用法：**选择一个空集，然后使用 enter().append()的形式来添加足够数量的元素**。例如，假设 body 里没有 p 元素，请看如下代码。

```
var dataset = [10,20,30,40,50];
var body = d3.select("body");
body.selectAll("p")     //选择body中的所有p，但由于没有p，因此选择了一个空集
    .data(dataset)      //绑定dataset数组
    .enter()            //返回enter部分
    .append("p")        //添加p元素
    .text(function(d){ return d; });
```

上述代码中，selectAll 选择了一个空集，然后绑定数据。由于选择集为空，因此 data()返回的 update 部分为空。然后，调用 enter()返回 enter 部分，enter 部分包含有多余的五个数据。接下来，向 enter 部分添加元素（append），使得每一个数据都有元素 p 与之对应。最后，更改 p 元素的内容。最终，网页中元素结构为：

```
<body>
    <p>10</p>
    <p>20</p>
    <p>30</p>
    <p>40</p>
    <p>50</p>
</body>
```

4.5.2　exit 的处理方法

如果存在多余的元素，没有数据与之对应，那么就需要删除元素。使用 remove()即可删除

元素。请看下面的示例，假设 body 中原有 5 个 p 元素。

```
var dataset = [10, 20, 30];
var p = d3.select("body").selectAll("p");

//绑定数据之后，分别获取update部分和exit部分
var update = p.data(dataset);
var exit = update.exit();

//update部分的处理方法是修改内容
update.text( function(d){ return d; } );

//exit部分的处理方法是删除
exit.remove();
```

这段代码中，对于 exit 部分的处理方法是删除。删除之后，网页中将不会有多余的 p 元素，剩下来的每一个 p 元素都有数据与之对应。

4.5.3 处理模板

经过第 4.5.1 和 4.5.2 节的学习，知道了如何处理多余的数据和元素。但是，有时候不知道是数据多，还是元素多。其实，不必在意数组长度和元素数量。大多数情况下，update 部分都是“更新元素属性”，enter 部分都是“添加元素并赋予初始属性”，exit 部分都是“删除元素”。因此，无论数据和元素如何，其处理方法都是类似的。因此，可归纳成一个处理模板，用户不必理会数组长度和元素数量之间的关系。模板的代码如下。

```
var dataset = [10, 20, 30];
var p = d3.select("body").selectAll("p");

//绑定数据后，分别返回update、enter、exit部分
var update = p.data(dataset);
var enter = update.enter();
var exit = update.exit();

//1.update部分的处理方法
update.text( function(d){ return d; } );

//2.enter部分的处理方法
enter.append("p")
    .text( function(d){ return d; } );

//3.exit部分的处理方法
```

```
exit.remove();
```

如此，则不需要考虑网页中 p 元素是多余还是不足。无论是何种情况，最终的结果必定是一个 p 元素对应数组中的一个项，没有多余的。对于上面的模式，笔者称其为**模板**，在数据需经常更新时很常用。

4.5.4　过滤器

有时需要根据被绑定数据对某选择集的元素进行过滤，如只对 id 大于 100 的职员进行奖励，只选拔身高超过 170cm 的学生等。类似这样的问题，需根据条件获取选择集的一部分子集，该方法称为**过滤器**。代码如下所示。

```
selection.filter(function(d,i){
    if( d > 20 )
        return true;
    else
        return false;
});
```

当所绑定的数据的数值大于 20 时，返回 true；否则返回 false。即只有数值大于 20 的元素才能够存在，其他的被过滤掉。这个函数的返回结果为绑定数据的数值大于 20 的所有元素。

4.5.5　选择集的顺序

sort()可以根据**被绑定数据**重新排列选择集中的元素。sort()的参数是一个**无名函数**，该函数也称作**比较器**。比较器的规则与 JavaScript 的 Array.sort()一样。先看下面的比较函数。

```
function(a,b){
    if( a < b )
        return -1;
    else if( a > b )
        return 1;
    else
        return 0;
}
```

此比较函数有两个参数，当 a 应该位于 b 之前时，则返回一个负数；当 a 应该位于 b 之后时，则返回一个正数；如果 a 与 b 相等，则返回 0。

根据以上规则，可以对选择集重新排序，例如：

```
selection.sort(function(a,b){
```

```
    return b-a;
});
```

如此可使选择集递减排序。如果不指定比较函数，则默认为 d3.ascending，这是 D3 提供的一个递增函数。

4.5.6 each()的应用

each()允许对选择集的各元素分别处理。请看如下代码：

```
var persons = [{ id: 1001 , name:"ZhangSan" },
               { id: 1002 , name:"LiSi" }];

var p = d3.select("body").selectAll("p");

p.data(persons)
    .each(function(d,i){
        d.age = 20;
    })
    .text( function(d,i){
        return d.id + " " + d.name + " " + d.age;
    });
```

这段代码中，被绑定的数据里是没有 age 属性的。通过 each()函数为每一项添加了 age 属性后，在最后的 text()中使用了此新属性。

4.5.7 call()的应用

call()允许将选择集自身作为参数，传递给某一函数。如：

```
d3.selectAll("div").call(myfun);
```

这段代码将选择集作为参数传递给 myfun 函数使用。等同于以下代码：

```
function myfun(selection) {
    //在这里进行相关操作
    selection.attr("name", "value");
}
myfun(d3.selectAll("div"));
```

后续章节中，将会出现拖曳、缩放元素等操作，那时会常用到 call()，这里先有一个印象即可，后文中出现时还会详细介绍其用法。

4.6　数组的处理

数组是一种常用的数据结构，通常是由相同数据类型的项组成集合，拥有数组名，可以凭借数组名和下标来访问数组项。虽然 JavaScript 允许一个数组中存在不同的数据类型，但实际很少这样使用。需要被可视化的数据常以数组的形式存在，虽然 JavaScript 中提供了不少操作数组的方法，但 JavaScript 不是为了数据可视化而存在的。因此，D3 根据数据可视化的需求封装了不少数组处理函数。

4.6.1　排序

第 4.5.5 节中提到，对选择集使用 sort()时，如果不指定比较函数，默认是 d3.ascending。D3 提供了递增和递减两个比较函数。比较函数的规则是：有函数 function(a,b)，

如果要使 a 位于 b 之前，则返回值小于 0；

如果要使 a 位于 b 之后，则返回值大于 0；

如果 a 与 b 相等，则返回值为 0。

- **d3.ascending**(a,b)

递增函数。如果 a 小于 b，返回–1；如果 a 大于 b，返回 1；如果 a 等于 b，返回 0。请看下面的例子：

```
var numbers = [54,23,77,11,34];
numbers.sort(d3.ascending);
console.log(numbers);      //[11, 23, 34, 54, 77]
```

要注意，上面的 sort()是 JavaScript 的数组对象（Array）的方法，而不是 D3 的 selection.sort()。输出结果是经过递增排序后的数组。

- **d3.descending**(a,b)

递减函数。如果 a 大于 b，返回–1；如果 a 小于 b，返回 1；如果 a 等于 b，返回 0。再看一个类似的例子：

```
var numbers = [54,23,77,11,34];
numbers.sort(d3.descending);
console.log(numbers);      //[77, 54, 34, 23, 11]
```

输出结果是经过递减排序后的数组。

4.6.2 求值

求取数组的最大值、最小值、中间值、平均值等，是十分常用的操作。D3 中，这一类函数形如：

```
d3.function( array [, accessor ] );
```

第一个参数 array 是数组，第二个参数 accessor 是可选参数。accessor 是一个函数，指定之后，数组各项首先会调用 accessor，然后再使用原函数 function 进行处理。

- **d3.min**(array[, accessor])

返回数组最小值。

- **d3.max**(array[, accessor])

返回数组最大值。

- **d3.extend**(array[, accessor])

返回数组最小值和最大值。

以上三个函数的参数有两个：必选参数 array 和可选参数 accessor。其中，array 中的 undefined 会自动被忽略。请看下面的例子：

```
//数组定义
var numbers = [30,20,10,50,40];

//求最小值和最大值
var min = d3.min(numbers);
var max = d3.max(numbers);
var extent = d3.extent(numbers);

//输出结果
console.log(min);       //10
console.log(max);       //50
console.log(extent);    //[10, 50]

//使用accessor，在求值前先处理数据
var minAcc = d3.min(numbers, function(d){ return d*3; });
var maxAcc = d3.max(numbers, function(d){ return d - 5; });
var extentAcc = d3.extent(numbers, function(d){ return d%7; });

//输出结果
console.log(minAcc);        //30
console.log(maxAcc);        //45
console.log(extentAcc);     //[1, 6]
```

这段代码中，先是在不指定 accessor 的情况下，调用了最大值和最小值的三个函数，并输出计算结果。然后，在指定 accessor 的情况下，再次调用了三个函数。以 d3.min 为例来讲解 accessor 的用法，函数如下：

```
function(d){
    return d*3;
}
```

对于以上函数，numbers 数组中的每一项都会先调用此函数，即：**每一项都乘以 3**。调用之后数组变为[90, 60, 30, 150, 120]，然后再求此数组的最小值，结果为 30。d3.max()中的 accessor 是将每一项减 5，减去之后数组变为[25, 15, 5, 45, 35]，因此最大值为 45。

d3.extent()相当于分别调用 d3.min()和 d3.max()，返回值是一个数组，第一项是最小值，第二项是最大值。

- **d3.sum**(array[, accessor])

返回数组的总和，如果数组为空，则返回 0。

- **d3.mean**(array[, accessor])

返回数组的平均值，如果数组为空，则返回 undefined。

以上两个函数的参数同样是：必选参数 array 和可选参数 accessor。array 中无效的值 undefined 或 NaN 会被忽略。请看下面的例子：

```
//数组定义
var numbers = [69,11,undefined,53,27,82,65,34,NaN];

//求总和、平均值
var sum = d3.sum(numbers,function(d){ return d/3; });
var mean = d3.mean(numbers);

//输出结果
console.log(sum);       //113.66666666666667
console.log(mean);      //48.714285714285715
```

这段代码的数组中，有两项的值为 undefined 和 NaN，但是对于函数的使用是不受影响的。这里有一个问题：求平均值的时候，平均值等于数组各项的总和除以数组长度，但是由于有 undefined 和 NaN 等值的存在，数组长度会大于实际需要的长度，那么 d3.mean()会怎么处理呢？其实，d3.mean()并非等同于：

```
d3.sum(array)/array.length;
```

而是使用去除掉无效数值之后的有效长度。

- **d3.median**(array[, accessor])

求数组的中间值，如果数组为空，则返回 undefined。

- **d3.quantile**(numbers, p)

求取 p 分位点的值，p 的范围为[0,1]。数组需先递增排序。

d3.median()的参数为：数组 array 和可选参数 accessor。与 d3.sum()和 d3.mean()一样，会忽略掉 undefined 和 NaN。如果数组的有效长度为奇数，则中间值为数组经过递增排序之后位于正中间的值；如果有效长度为偶数，则中间值为经过递增排序后位于正中间的两个数的平均值。

d3.median()其实是使用 d3.quantile()实现的。d3.quantile()接受两个参数：第一个是经过递增排序后的数组；第二个是分位点，范围是[0,1]。请先看看 d3.quantile()是如何使用的：

```
var numbers = [3,1,10];
numbers.sort(d3.ascending);
d3.quantile(numbers,0);          //返回1
d3.quantile(numbers,0.25);       //返回2
d3.quantile(numbers,0.5);        //返回3
d3.quantile(numbers,0.75);       //返回6.5
d3.quantile(numbers,0.9);        //返回8.599999999999998
d3.quantile(numbers,1.0);        //返回10
```

这段代码的数组，1 位于 0 分位处，3 位于 0.5 分位处，10 位于 1 分位处。d3.median()其实相当于是将数组中的无效值（undefined 和 NaN）去掉之后，再使用 d3.quantile()获取 0.5 分位处的值，即：

```
d3.quantile( numbers.sort(d3.ascending), 0.5);
```

下面来看看使用 d3.median()的例子。

```
var numbers1 = [3,1,7,undefined,9,NaN];
d3.median(numbers1);          //返回5
var numbers2 = [3,1,7,undefined,9,10,NaN];
d3.median(numbers2);          //返回7
```

这段代码表示数组中无效值被忽略了，只计算有效项的中间值。

- **d3.variance**(array[, accessor])

求方差。

- **d3.deviation**(array[, accessor])

求标准差。

方差和标准差用于度量随机变量和均值之间的偏离程度，在概率统计中经常用到。其中标准差是方差的二次方根。这两个值越大，表示此随机变量偏离均值的程度越大。这两个函数的参数为必选参数 array 和可选参数 accessor，并且都会忽略数组 array 中的 undefined 和 NaN。请看下面的代码。

```
var numbers1 = [1,9,7,9,5,8,9,10];
```

```
d3.mean(numbers1);                    //返回平均值：7.25
d3.variance(numbers1);                //返回方差：约等于8.79
d3.deviation(numbers1);               //返回标准差：约等于2.96

var numbers2 = [7,8,6,7,7,8,8,7];
d3.mean(numbers2);                    //返回平均值：7.25
d3.variance(numbers2);                //返回方差：约等于0.50
d3.deviation(numbers2);               //返回标准差：约等于0.71
```

这段代码中，数组 numbers1 和 numbers2 的平均值都是 7.25，但是前者的方差和标准差分别为 8.79 和 2.96，后者的方差和标准差分别为 0.50 和 0.71，表明数组 numbers1 中的值偏离平均值 7.25 的程度较大。

- **d3.bisectLeft()**

获取某数组项左边的位置。

- **d3.bisect()**

获取某数组项右边的位置。

- **d3.bisectRight()**

和 bisect()一样。

有时候需要对数组中指定的位置插入项，因此需要获取指定的位置。在 JavaScript 中，要向某数组插入项，可使用 splice()，而 bisectLeft()、bisect()、bisectRight()可配合 splice()使用。首先来看看 splice()是怎样插入数组项的。

```
var countries = ["China","America","Japan","France"];

//在数组索引为1的位置处，删除0个项后，插入字符串Germany
countries.splice(1,0,"Germany");

//输出 ["China", "Germany", "America", "Japan", "France"]
console.log(countries);

//在数组索引为3的位置处，删除一个项后，插入两个字符串Britain和Russia
countries.splice(3,1,"Britain","Russia");

//输出["China", "Germany", "America", "Britain", "Russia", "France"]
console.log(countries);
```

splice()可用于删除数组项，也可用于插入数组项。例如，splice(1,3)表示删除从下标 1 开始的 3 项；splice(1,0,"Germany")则表示在索引为 1 的位置处插入字符串 Germany（删除 0 个项）。下面来看看 bisectLeft()是怎么使用的：

```
var numbers = [10,13,16,19,22,25];

//iLeft的值为2
```

```
var iLeft = d3.bisectLeft(numbers.sort(d3.ascending),16);

//在iLeft位置处，删除0个项后，插入77
numbers.splice(iLeft,0,77);

//输出[10, 13, 77, 16, 19, 22, 25]
console.log(numbers);
```

这段代码中，将 numbers 排序后，再使用 bisectLeft()获取了 16 左边的位置。bisectLeft()所使用的数组必须经过递增排序。第二个参数用于指定某项的位置，如果此项在数组中存在，则返回此位置的左边；如果此项在数组中不存在，则返回第一个大于此项的值的左边。请看下面的例子：

```
var numbers = [10,13,16,19,22,25];

//18在数组中不存在，返回介于16和19之间的位置，返回值为3
var iLeft = d3.bisectLeft(numbers.sort(d3.ascending),18);
numbers.splice(iLeft,0,77);

//输出[10, 13, 16, 77, 19, 22, 25]
console.log(numbers);
```

bisect()和 bisectRight()是一样的，和 bisectLeft()类似，只是获取的是指定项右边的位置。

4.6.3 操作数组

D3 提供了将数组洗牌、合并等操作，使用起来是很方便的。

- **d3.shuffle**(array[, lo[, hi]])：随机排列数组。

“shuffle”有“洗牌”的意思，将数组作为参数使用后，能将数组随机排列。请看下面的例子。

```
var numbers = [10,13,16,19,22,25];
d3.shuffle(numbers);

//结果随机，本次试验为[16, 25, 13, 10, 22, 19]
console.log(numbers);
```

数组被重新随机排列了。

- **d3.merge**(arrays)：合并两个数组。

将两个数组合并时使用：

```
// returns [1, 2, 3]
```

```
d3.merge([ [1], [2, 3] ]);
```

数组[1]和[2,3]被合并成了一个数组[1,2,3]。

- **d3.pairs**(array)：返回邻接的数组对。

以第 *i* 项和第 *i*–1 项为对返回，请看下面的例子：

```
var colors = ["red","blue","green"];

//colors不变，结果保存在pairs中
var pairs = d3.pairs(colors);

//结果为[ ["red","blue"] , ["blue","green"] ]
console.log(pairs);
```

使用 pairs()后，原数组是不变的。

- **d3.range**([start,]stop[, step])：返回等差数列。

参数有三个：start、stop、step。返回的等差数列为：

```
[start, start + step, start + 2 * step, ...]
```

如果 stop 为正，则最后的值小于 stop；如果 stop 为负，则最后的值大于 stop。start 和 step 如果省略，则默认值分别为 0 和 1。请看下面的例子。

```
//start为0，stop为10，step为1
var a = d3.range(10);

//输出[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
console.log(a);

//start为2，stop为10，step为1
var b = d3.range(2,10);

//输出[2, 3, 4, 5, 6, 7, 8, 9]
console.log(b);

//start为2，stop为10，step为2
var c = d3.range(2,10,2);

//输出[2, 4, 6, 8]
console.log(c);
```

要注意省略参数的使用方法，根据参数个数的不同，结果是不同的。range()在生成数组时经常使用。

- **d3.permute**(array, indexes)：根据指定的索引号数组返回排列后的数组。

此函数可以用某个由索引号组成的数组，返回根据索引号排列后的新数组，原数组不变。请看下面的例子：

```
var animals = ["cat","dog","bird"];

//根据[2,1,0]将数组animals重新排列，但是原数组animals不变，结果保存在返回值中
var newAnimals = d3.permute(animals,[2,1,0]);

//输出["bird", "dog", "cat"]
console.log(newAnimals);
```

要注意数组索引号是从 0 开始的，如果有超出范围的索引号，该位置会以 undefined 代替。

- **d3.zip**(arrays...)：用多个数组来制作数组的数组。

参数是任意多个数组，输出是数组的数组。例如：

```
   var zip = d3.zip(   [1000,1001,1002],
                       ["Zhangsan","Lisi","Wangwu"],
                       [true,false,true] );

//输出为[  [1000, "Zhangsan", true] ,
       [1001, "Lisi", false] ,
       [1002, "Wangwu", true] ]
console.log(zip);
```

可以看到，参数中每个数组的第 *i* 项组成了新数组的第 *i* 项。zip()可以被用来求向量的内积，请看下面的例子：

```
var a = [10,20,5];
var b = [-5,10,3];
var ab = d3.sum( d3.zip(a,b) , function(d){ return d[0]*d[1]; } );
console.log(ab);   //165
```

这段代码中，使用 d3.zip(a,b)得到的结果为：

```
[ [10,-5], [20,10], [5,3] ];
```

然后，这个数组会被 function(d)先处理，处理的结果为：

```
[ -50, 200, 15 ];
```

最后再被 d3.sum()求和，结果即向量 a 和 b 的内积。

- **d3.transpose**(matrix)：求转置矩阵。

将矩阵的行换成相应的列，得到的矩阵即转置矩阵。请看下面的例子：

```
var a = [ [1,2,3] , [4,5,6] ] ;
```

```
//转置后，原数组a不变，结果保存在返回值中
var t = d3.transpose(a);

//输出[ [1,4],[2,5],[3,6] ]
console.log(t);
```

4.6.4　映射（Map）

映射（Map）是十分常见的一种数据结构，由一系列**键（key）**和**值（value）**组成。每个 key 对应一个 value，根据 key 可以获取和设定 value，也可以根据 key 来查询 value。图 4-14 展示了一个映射，该映射以每个值的 id 作为键，每个键对应一个值。

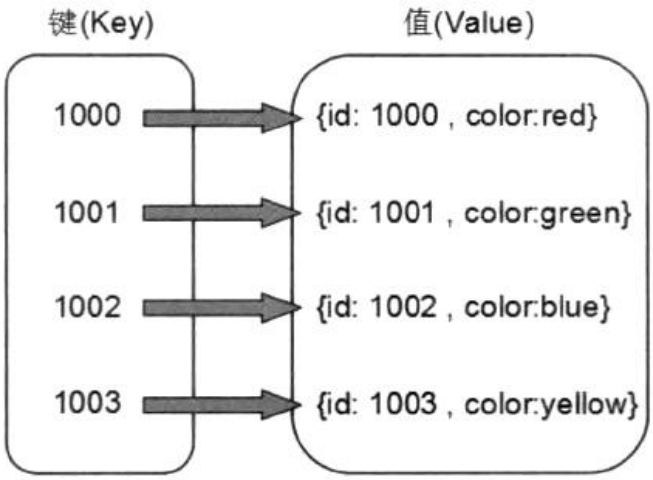


图 4-14　映射（map）的键（key）和值（value）

d3.map()能构建映射，包括以下方法。

- **d3.map**([object][, key])

构造映射。第一个参数是源数组，第二个参数用于指定映射的 key。

- **map.has**(key)

如果指定的 key 存在，则返回 true；反之，则返回 false。

- **map.get**(key)

如果指定的 key 存在，则返回该 key 的 value；否则，返回 undefined。

- **map.set**(key, value)

对指定的 key 设定 value，如果该 key 已经存在，则新 value 会覆盖旧 value；如果该 key 不存在，则会添加一个新的 value。

- **map.remove**(key)

如果指定的 key 存在，则将此 key 和 value 删除，并返回 true；如果不存在，则返回 false。

- **map.keys()**

以数组形式返回该 map 所有的 key。

- **map.values()**

以数组形式返回该 map 所有的 value。

- **map.entries()**

以数组形式返回该 map 所有的 key 和 value。

- **map.forEach**(function)

分别对该映射中的每一项调用 function 函数，function 函数传入两个参数：key 和 value。分别代表每一项的 key 和 value。

- **map.empty()**

如果该映射为空，返回 true；否则，返回 false。

- **map.size()**

返回该映射的大小。

下面参照图 4-14 来构建一个映射，并分别调用上述函数。请看下面的代码。

```
//用于构建映射的数组
var dataset = [ { id: 1000, color:"red" },
                { id: 1001, color:"green" },
                { id: 1002, color:"blue" } ];

//以数组dataset构建映射，并以其中各项的id作为键
var map = d3.map(dataset, function(d){ return d.id; });

map.has(1001);      //返回true
map.has(1003);      //返回false

map.get(1001);      //返回{ id: 1001, color:"green" }
map.get(1003);      //返回undefined

//将1001键的值设置为 { id: 1001, color:"yellow" }
map.set(1001, { id: 1001, color:"yellow" } );

//将1003键的值设置为 { id: 1003, color:"white" }
map.set(1003, { id: 1003, color:"white" } );

map.remove(1001);  //删除键为1001的键和值

map.keys();         //返回["1000", "1002", "1003"]
map.values();       //返回所有的值
map.entries();      //返回所有的键和值
```

```
//该循环会进行三次，键依次为1000、1002、1003
map.forEach(function(key,value){
    console.log(key);
    console.log(value);
});

map.empty();    //返回false
map.size();     //返回3
```

映射还是很好理解的。唯一要注意的是，以上代码中，**值（value）**是包含**键（key）**的，即值是{id: 1001, color:"yellow"}，而不是{color: "yellow"}，使用 set()设定值的时候尤其要留意。键（key）是在构建映射时，由 function(d){ return d.id; }指定的，当然也可以指定别的不相关值，不一定要使用 id。

4.6.5　集合（Set）

集合（Set）是数学中常用的概念，表示具有某种特定性质的事物的总体。集合里的项叫作**元素**。集合的相关方法如下。

- **d3.set**([array])

使用数组来构建集合，如果数组里有重复的元素，则只添加其中一项。

- **set.has**(value)

如果集合中有指定元素，则返回 true；如果没有，返回 false。

- **set.add**(value)

如果该集合中没有指定元素，则将其添加到集合中，并返回该元素；如果有，则不添加。

- **set.remove**(value)

如果该集合中有指定元素，则将其删除并返回 true；否则，返回 false。

- **set.values**()

以数组形式返回该集合中所有的元素。

- **set.forEach**(function)

对每一个元素都调用 function 函数，函数里传入一个参数，即该元素的值。

- **set.empty**()

如果该集合为空，则返回 true；否则，返回 false。

- **set.size**()

返回该集合的大小。

与映射相比，集合比较简单，没有键的概念。下面通过一段代码，来实践上述函数的使用方法：

```
//源数组
var dataset = ["tiger","dragon","snake","horse","sheep"];

//构建一个集合，将其保存在变量set中
var set = d3.set(dataset);

set.has("tiger");        //返回true
set.add("monkey");       /添加monkey，并返回monkey

set.remove("snake");     //删除snake

//返回["tiger","dragon","horse","sheep","monkey"]
   set.values();

//集合中的每一个元素都将调用function函数，该函数的内容为输出各元素
set.forEach(function(value){
   console.log( value );
});

set.empty();    //返回false
set.size();     //返回5
```

有一点要注意，集合中不允许出现相同的元素，如果有相同的元素，则只会保留其中一项。这一点主要表现在构造集合和添加元素时：

```
var dataset = ["tiger","tiger","dragon","snake","horse","sheep"];
var set = d3.set(dataset);
set.add("dragon");

//输出["tiger", "dragon", "snake", "horse", "sheep"]
console.log( set.values() );
```

由于数组 dataset 中的 tiger 元素有两个，因此结果只能保留一个。在使用 add()添加元素时，dragon 已经在集合中存在了，因此也没有添加。

4.6.6 嵌套结构（Nest）

嵌套结构能够使用**键（key）**对数组中的大量对象进行**分类**，多个键一层套一层，使得分类越来越具体，索引越来越方便。假设现有数组，如图 4-15 所示。

图 4-15　嵌套结构的源数组

对于这样一个数组，可以使用各 value 中的某些数据作为键（key），例如如果要在几千个职员数据中查找其中一个职员的信息，但只知道其出生地和年龄分别为**北京**和 **22 岁**，一般来说这么查找比较简单：

先查找在**北京**的职员，再在其中查找 **22 岁**的职员。

如此查找可一步步缩小范围。那么**出生地**和**年龄**即可作为嵌套结构的**键（key）。**如图 4-16 所示，经过分类之后，要查找某一个元素时，即可首先根据键缩小范围。

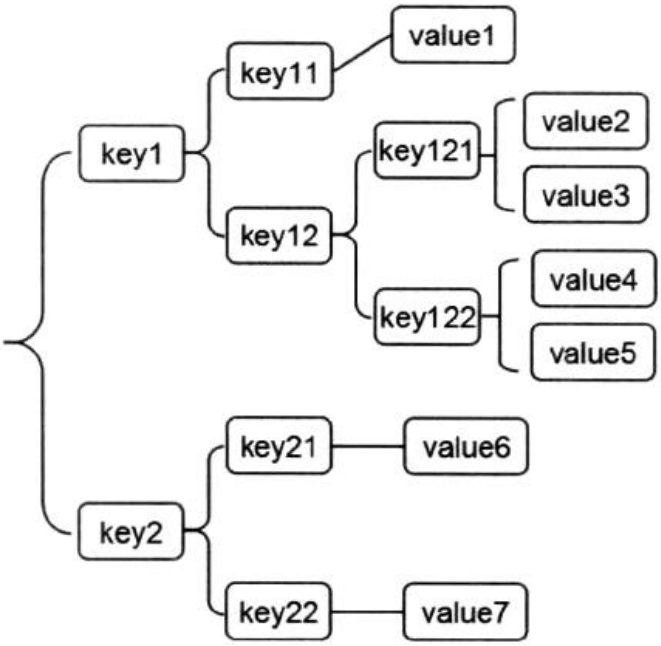


图 4-16　使用键对数组分类

下面先看一个简单的例子：

```
var persons = [
    {id:100, name:"张某某", year:1989, hometown:"北京"},
    {id:101, name:"李某某", year:1987, hometown:"北京"},
    {id:102, name:"王某某", year:1988, hometown:"上海"},
    {id:103, name:"赵某某", year:1987, hometown:"广州"},
    {id:104, name:"孙某某", year:1989, hometown:"上海"}
];

var nest = d3.nest()
              //将year作为第一个键
              .key(function(d) { return d.year; })
              //将hometown作为第二个键
              .key(function(d) { return d.hometown; })
```

```
                //指定将应用嵌套结构的数组为persons
                .entries(persons);

    console.log(nest);
```

以上代码涉及三个方法，意义如下。

- **d3.nest()**

该函数没有任何参数，表示接下来将会构建一个新的嵌套结构。其他函数需要跟在此函数之后一起使用。

- **nest.key**(function)

指定嵌套结构的键。

- **nest.entries**(array)

指定数组 array 将被用于构建嵌套结构。

上述代码分别指定 year 和 hometown 为嵌套结构的键，要注意它们出现的顺序是会影响到结果的。最终输出的是用数组表示的类似图 4-16 的结构：

```
[{key: 1987, values: [
    {key: "北京", values: [
        {id:101, name:"李某某", year:1987, hometown:"北京"}
    ]},
    {key: "广州", values: [
        {id:103, name:"赵某某", year:1987, hometown:"广州"}
    ]}
]},
{key: 1988, values: [
    {key: "上海", values: [
        {id:102, name:"王某某", year:1988, hometown:"上海"}
    ]}
]},
{key: 1989, values: [
    {key: "北京", values: [
        {id:100, name:"张某某", year:1989, hometown:"北京"}
    ]},
    {key: "上海", values: [
        {id:104, name:"孙某某", year:1989, hometown:"上海"}
    ]}
]}]
```

嵌套结构还有一些方法，介绍如下。

- **nest.sortKeys**(comparator)

按照键对嵌套结构进行排序，接在 nest.key()后使用。

```
d3.nest()
    .key(function(d) { return d.year; })
    .sortKeys(d3.descending)    //按照键year进行排序
    .key()  //其他键的定义
```

- **nest.sortValues**(comparator)

按照值对嵌套结构进行排序。

- **nest.rollup**(function)

对每一组叶子节点调用指定的函数 function，该函数含有一个参数 values，是当前叶子节点的数组。

- **nest.map**(array[, mapType])

以映射的形式输出数组。

下面实践一下上述方法，假设有以下数据：

```
var persons = [
    {sex:"男", age:48 , name:"张某某"},
    {sex:"男", age:42 , name:"李某某"},
    {sex:"男", age:45 , name:"王某某"},
    {sex:"女", age:33 , name:"赵某某"},
    {sex:"女", age:31 , name:"孙某某"}
];
```

如果用 **sortValues()**将数组按 age 排序并输出成嵌套结构，可使用如下代码。

```
var nest = d3.nest()
                .key(function(d) { return d.sex; })
                .sortValues(function(a,b){
                    return d3.ascending(a.age,b.age);
                })
                .entries(persons);
```

则以 sex 为键的每一个分组的元素，都将按照 age 进行递增排序。其输出结果为：

```
[{key: "男", values: [
    {sex:"男", age:42 , name:"李某某"},
    {sex:"男", age:45 , name:"王某某"},
    {sex:"男", age:48 , name:"张某某"}
]},
{key: "女", values: [
    {sex:"女", age:31 , name:"孙某某"}
    {sex:"女", age:33 , name:"赵某某"}
]}]
```

rollup()的参数是一个无名函数 function()，设定之后，各元素分组都会调用。在上面代码的

基础上，添加 rollup()之后，嵌套结构的定义如下所示：

```
var nest = d3.nest()
              .key(function(d) { return d.sex; })
              .rollup(function(values){ return values.length; })
              .entries(persons);
```

输出结果如下：

```
[{key: "男", values: 3 },
 {key: "女", values: 2 }]
```

values 的原值是该分组的数组，在这里通过 rollup()将其变为了该分组元素的数量。

如果想将结果输出为映射形式，可以使用 **nest.map()**，代码如下所示：

```
var map= d3.nest()
           .key(function(d) { return d.sex; })
           .map(persons,d3.map);
```

使用映射的方式输出后，其结果为：

```
{
   "女":[
       {sex:"女", age:33 , name:"赵某某"},
       {sex:"女", age:31 , name:"孙某某"}
   ] ,
   "男":[
       {sex:"男", age:48 , name:"张某某"},
       {sex:"男", age:42 , name:"李某某"},
       {sex:"男", age:45 , name:"王某某"}
   ]
}
```

可以看到，使用映射的方式输出时，其结果的最外层是一个花括号，而不是中括号。即它是一个对象，而不是一个数组。内部的形式也有很多不同之处，请注意区别。

4.7 柱形图的制作

本章讲解了很多概念，但还没有关于实际制作图表的内容。为了将本章的知识融会贯通，加深理解，本节制作一个柱形图。

柱形图（Bar Chart）是使用柱形的长短来表示数据变化的图表，也是最简单的图表之一。一般情况下，柱形图包括：**矩形**、**坐标轴**和**文字**。坐标轴会在下一章中单独介绍，暂不理会，

此处只绘制矩形和文字。第 2.5 节中提到，SVG 矩形元素的标签是<rect>，其中属性 width 和 height 可用于设定矩形的长短宽窄；文字元素的标签是<text>，用于表示矩形图中的文字。

4.7.1　矩形和文字

既然柱形图是用矩形的长短来表示数据的，那么可定义一个数组，数组项直接就是矩形的长短：

```
var dataset = [50, 43, 120, 87, 99, 167, 142];
```

上面定义了一个数组 dataset，数组的长度就是矩形的数目，数组项的大小表示矩形的高度，单位为像素（px）。例如，50 表示在网页中矩形的高度是 50 个像素。以后会讲到，“**50 直接对应 50 个像素**”的方法并不理想。但是，为了使问题尽可能简单，先这么做着。定义一块 SVG 的绘制区域：

```
var width  = 400;  //SVG绘制区域的宽度
var height = 400;  //SVG绘制区域的高度

var svg = d3.select("body")                    //选择<body>
           .append("svg")                      //在<body>中添加<svg>
           .attr("width", width)               //设定<svg>的宽度属性
           .attr("height", height);            //设定<svg>的高度属性
```

<body>中添加了一个<svg>元素标签，其宽度和高度都被设置为 400 像素。柱形图的图形元素都将被添加到<svg>里，例如<rect>和<text>。在此之前，先定义三个变量：

```
//定义上下左右的边距
var padding = { top: 20 , right: 20, bottom: 20, left: 20 };

//矩形所占的宽度（包括空白），单位为像素
var rectStep = 35;

//矩形所占的宽度（不包括空白），单位为像素
var rectWidth = 30;
```

三个变量的含义如图 4-17 所示，padding 表示在 svg 绘制区域内一段空白的宽度，这样做可以防止某些图形元素绘制到 svg 区域的外部。rectStep 表示前一个矩形开始位置到后一个矩形开始位置的距离，此部分包含一段空白，为了与下一个矩形做区分。rectWidth 是矩形实际所占的宽度，此部分是要填充颜色的。rectStep 大于 rectWidth。

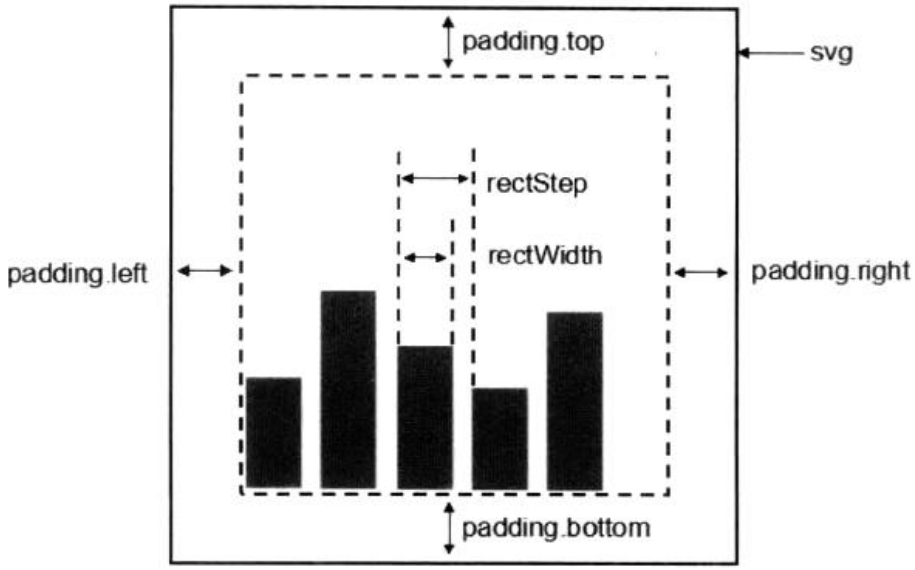


图 4-17　柱形图各参数的含义

了解了矩形的绘制位置之后，在<svg>中添加<rect>元素。此时，<svg>中没有任何元素，只有数据 dataset。第 4.5.1 节提到，可用 selectAll()选择一个空集，然后再用 enter().append()添加元素，最终使得<rect>元素的数量与数据的数量一致。

```
var rect = svg.selectAll("rect")
           .data(dataset)        //绑定数据
           .enter()              //获取enter部分
           .append("rect") //添加rect元素，使其与绑定数组的长度一致
           .attr("fill","steelblue")       //设置颜色为steelblue
           .attr("x", function(d,i){       //设置矩形的x坐标
               return padding.left + i * rectStep;
           })
           .attr("y", function(d){     //设置矩形的y坐标
               return height- padding.bottom - d;
           })
           .attr("width",rectWidth)        //设置矩形的宽度
           .attr("height",function(d){//设置矩形的高度
               return d;
           });
```

数组 dataset 的长度为 7，<rect>也需要有七个。添加了元素之后，需要为其设置颜色、位置和大小。颜色设置为钢蓝色（steelblue）。x 和 y 坐标的位置最重要，由于坐标的设置与矩形的索引号有关，因此需要使用无名函数 function()，而且函数里需要放入两个参数 d 和 i，前面说过，d 表示数据（datum），i 表示索引号（index）。这里有七个矩形，分别绑定了数组 dataset 中的七个项，那么 d 和 i 的值分别为：

```
d: 50, 43, 120, 87, 99, 167, 142
i: 0, 1, 2, 3, 4, 5, 6
```

经过上述代码的计算，各矩形的 x 和 y 坐标分别为：

```
x: 20, 55, 90, 125, 160, 195, 230
y: 330, 337, 260, 293, 281, 213, 238
```

x 和 y 坐标是矩形的左上角顶点，其坐标是相对于 svg 绘制区域来说的。svg 区域的左上角为坐标原点(0,0)，越往右 x 值越大，越往下 y 值越小，如图 4-18 所示。

矩形的宽度即变量 rectWdith 的值，高度即数组 dataset 中的各数值。矩形的绘制结果如图 4-19 所示。

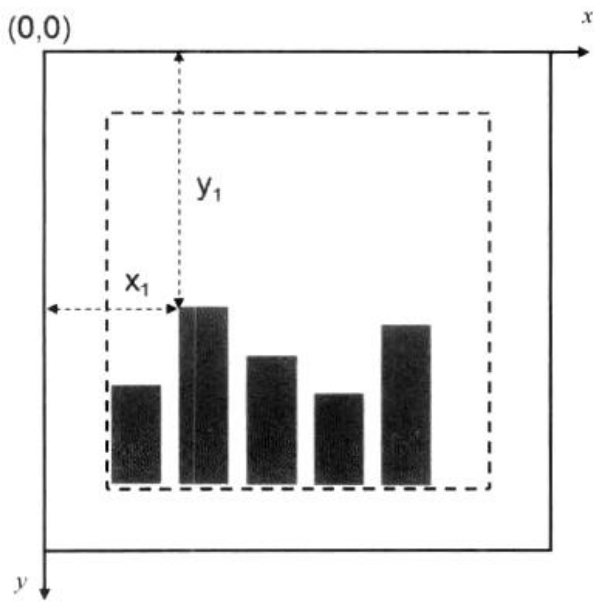


图 4-18　矩形的起始坐标

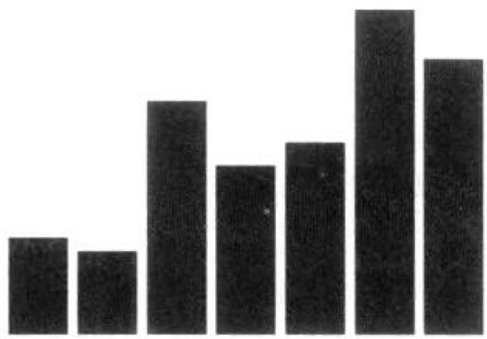


图 4-19　矩形的绘制结果

接下来为矩形添加标签文字，方法与添加<rect>一样，先用 selectAll()选择一个空集，然后再为选择集的 enter 部分添加足够数量的<text>元素。

```
var text = svg.selectAll("text")
          .data(dataset)          //绑定数据
          .enter()                //获取enter部分
          .append("text")         //添加text元素，使其与绑定数组的长度一致
          .attr("fill","white")
          .attr("font-size","14px")
          .attr("text-anchor","middle")
          .attr("x", function(d,i){
              return padding.left + i * rectStep;
          })
          .attr("y", function(d){
```

```
            return height- padding.bottom - d;
        })
        .attr("dx",rectWidth/2)
        .attr("dy","1em")
        .text(function(d){
            return d;
        });
```

这段代码中，将文字的颜色设置成白色，因为要添加到矩形之上，所以要与矩形的颜色不同，以示区别。字体大小被设置为 14px，结果如图 4-20 所示。

为了使每一个<text>元素正好显示在每个矩形的正中心，代码中设置了元素的 text-anchor、x、y、dx、dy 五个属性。其意义如图 4-21 所示，x 和 y 属性和矩形是一样的，不必再说。dx 和 dy 是相对于(x,y)平移的大小，默认单位是 px（像素）。在上面的代码中，dy 被设置成 1em。em 单位表示的是当前文字所占的一行的宽度，在表示文字相对位置时经常使用。经过平移设置之后，文本会从(x+dx,y+dy)位置开始显示，暂时把此位置称为**起始位置**。一般来说，此时有三种需求。

（1）文字的第一个字符位于起始位置的右方。

（2）文字的中心位于起始位置上。

（3）文字的最后一个字符靠近起始位置。

选择哪一种，是通过 text-anchor 来指定的。text-anchor 有三个值：start、middle、end，分别对应上述三种情况。在本次绘制中，使用的是 middle。

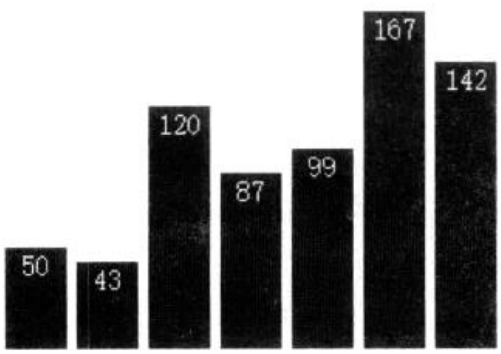


图 4-20　添加文字

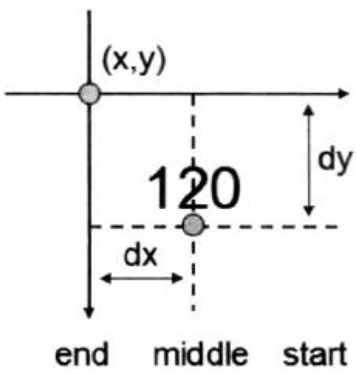


图 4-21　文字的位置属性

柱形图的基本构成部分绘制完成，读者可能注意到，两次添加元素，都使用了一种形式的代码：

```
d3.selectAll().data()
.enter().append()
```

此形式以后会经常使用，其用法在第 4.5.1 节中讲解过，不再复述。但是，本例的代码中，

使用 selectAll()选择元素时，都使用的是元素本来的名称，如 selectAll("rect")，但如果页面内原来就有<rect>元素，直接使用名称的话已有的元素也会被选上，这不是我们所希望的。我们需要的是选择一个空集。因此，可使用 CSS 选择器来代替元素名称，例如：

```
d3.selectAll(".myrect");
d3.selectAll(".mytext");
```

使用 myrect 和 mytext 两个类名。虽然这两个类可能并不存在，但是没关系，我们需要的仅仅是选择一个空集。

4.7.2　更新数据

有时候需要更新数据，更新后柱形图也得跟着变化。例如将数据排序、增加新的数据等，都需要柱形图跟着变化。本节将制作一个能够随着数据的更新实时变化的柱形图。

对上一节的代码做部分修改，使其能够在数据更改时，自动添加或删除矩形，并更新矩形的属性。这就要使用到第 4.5.3 节介绍过的处理模板，分别考虑 update、enter、exit 三个部分的处理方法。请看下面的代码：

```
function draw(){
    //获取矩形的update部分
    var updateRect = svg.selectAll("rect")
                        .data(dataset);

    //获取矩形的enter部分
    var enterRect = updateRect.enter();

    //获取矩形的exit部分
    var exitRect = updateRect.exit();

    //1. 矩形的update部分的处理方法

    //2. 矩形的enter部分的处理方法

    //3. 矩形的exit部分的处理方法

    //获取文字的update部分
    var updateText = svg.selectAll("text")
                        .data(dataset);
```

```
    //获取文字的enter部分
    var enterText = updateText.enter();

    //获取文字的exit部分
    var exitText = updateText.exit();

    //1. 文字的update部分的处理方法

    //2. 文字的enter部分的处理方法

    //3. 文字的exit部分的处理方法
}
```

将绘制图形的代码写在一个函数 draw()里，当数据发生更新时，再次调用此函数即可。模板中，分别获取矩形和文字的 update、enter、exit 三个部分，然后再分别处理。以矩形元素<rect>为例，来讲解如何分别处理。

```
//1. 矩形的update部分的处理方法
updateRect.attr("fill","steelblue")    //设置颜色为steelblue
        .attr("x", function(d,i){      //设置矩形的x坐标
            return padding.left + i * rectStep;
        })
        .attr("y", function(d){        //设置矩形的y坐标
            return height- padding.bottom - d;
        })
        .attr("width",rectWidth)       //设置矩形的宽度
        .attr("height",function(d){    //设置矩形的高度
            return d;
        });

//2. 矩形的enter部分的处理方法
enterRect.append("rect")
        .attr("fill","steelblue")      //设置颜色为steelblue
        .attr("x", function(d,i){      //设置矩形的x坐标
            return padding.left + i * rectStep;
        })
        .attr("y", function(d){    //设置矩形的y坐标
            return height- padding.bottom - d;
        })
        .attr("width",rectWidth)       //设置矩形的宽度
        .attr("height",function(d){//设置矩形的高度
            return d;
        });
```

```
//3. 矩形的exit部分的处理方法
exitRect.remove();
```

update 部分的处理方法是更新属性，enter 部分的处理方法是添加元素后再赋予其相应的属性，exit 部分的处理方法则是删除掉多余的元素。如此处理后，对于原数组 dataset，无论是排序还是增减数据，图表都能够自动发生相应的变化。下面在 HTML 中分别添加两个按钮，一个用于排序，一个用于增加数据。在 HTML 中的<body>里添加以下两个按钮：

```
<button type="button" onclick="mysort()">排序</button>
<button type="button" onclick="myadd()">增加数据</button>
```

给两个按钮定义了两个事件函数，但单击事件发生时，分别调用 mysort()和 myadd()。这两个函数的实现很简单，只要将数据处理后，再调用 draw()重绘一次即可：

```
function mysort(){
    dataset.sort(d3.ascending);//排序
    draw();
}

function myadd(){
    dataset.push( Math.floor(Math.random() * 100) );    //添加一个项
    draw();
}
```

添加按钮之后，结果如图 4-22 所示。

分别单击两个按钮，数组 dataset 发生变化，图形也能够自动对应数据发生相应的变化。结果如图 4-23 和图 4-24 所示。单击“排序”按钮后，柱形图从小到大进行了排序。再单击三次“增加数据”按钮后，在柱形图的右边增加了三个矩形。

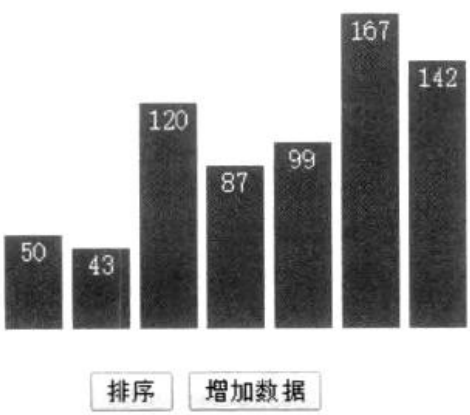


图 4-22　添加按钮后的结果

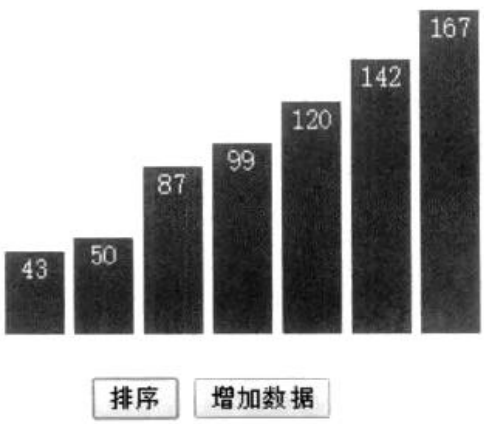
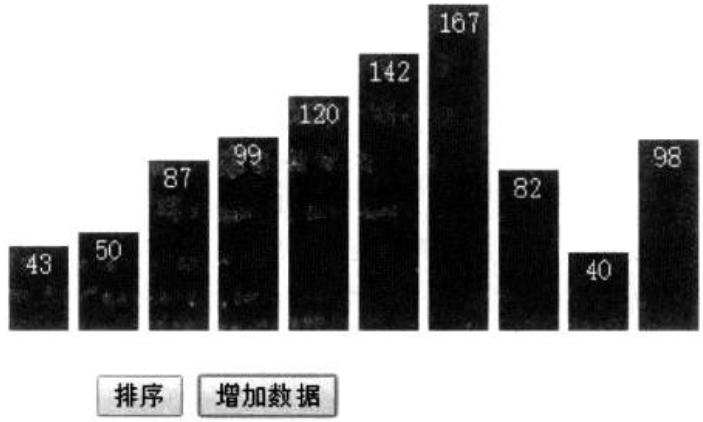


图 4-23　单击“排序”按钮后的结果



排序　增加数据

图 4-24　单击“排序”按钮后，再单击“增加数据”按钮三次的结果

当数据需要更新时，本例中所使用的模板会经常用到，而且 update、enter、exit 三个部分的处理方法通常也是类似的，即分别为更新属性、添加元素并设置属性、删除多余元素。希望读者能够理解并牢记本例中所使用的模板。

第 5 章

比例尺和坐标轴

本章内容包括：

- 定量比例尺（定义域是连续的）
- 序数比例尺（定义域是不连续的）
- 坐标轴
- 添加柱形图的坐标轴
- 制作散点图

比例尺能将“一个区间”的数据对应到“另一个区间”。例如：

（1）将[0, 1]对应到[0,300]。当输入 0.5 时，输出 150。

（2）将[0, 1, 2]对应到["red","green","blue"]。当输入 2 时，输出 blue。

第 1 节，讲述定量比例尺，上面的（1）就是定量比例尺，其定义域是连续的。

第 2 节，讲述序数比例尺，上面的（2）就是序数比例尺，其定义域是不连续的。

第 3 节，介绍如何应用比例尺来制作坐标轴。

第 4 节，完成第 4 章没有完成的问题：给柱形图添加坐标轴。

第 5 节，制作一个散点图，作为本章内容的复习。

5.1 定量比例尺

数学上有**函数**的概念，不是编程中所说的函数，如线性函数、指数函数、对数函数等，而

指的是一个量随着另一个量的变化而变化。例如有以下线性函数：

$$y=2x+1$$

该函数在二维坐标系中绘制出来的图形是一条直线，如果限制 x 的范围为[0,2]，则可计算得到 y 的范围为[1,5]。x 的范围[0,2]称为该函数的**定义域**，y 的范围称为该函数的**值域**；根据 x 计算得到 y 的方法称为**对应法则**。定义域、值域、对应法则称为函数的三要素。

在数据可视化中，常需要像上述函数一样，将一个量转换为另一个量。D3 提供了这样的转换方法，称为**比例尺**（scale）。本节中所说的**定量比例尺**，是指当定义域是连续的情况。从 0～2 之间的所有值，称为连续的值；类似 0、1、2 这样独立的值，称为离散的值。

为什么要用比例尺呢？假设现在要为某汽车公司做数据可视化，要将每月的汽车销量用柱形图表示。假设一月份销售了 100 辆车，用 100 个像素长度的柱子来表示；那么，二月份销售了 500 辆车，就可以用 500 个像素表示。如果三月份销售了 3000 辆车呢，不可能用 3000 个像素，浏览器页面的长度可能也不够，这时候就要用到比例尺了。

在第 4 章制作柱形图的时候，就曾提到过使用 50 个像素来表示数值 50 不好。实际上，观看者在看柱形图的时候，并不关心像素的多少，关心的是各柱形之间长度的比较，以及文字标签。例如，对于数组[1000,1500,2000]，没有必要分别用 1000、1500、2000 个像素来表示长度，用其十分之一 100、150、200 即可，文字标签可使用原数值。

D3 中提供了很多比例尺，适用于各种计算。每个比例尺都需要指定一个 domain（定义域）和 range（值域）。最常用的是线性比例尺，其用法如下：

```
var linear = d3.scale.linear()          //创建一个线性比例尺
                .domain([0,500])        //定义域
                .range([0,100]);        //值域

console.log( linear(50) );      //输出10
console.log( linear(250) );     //输出50
console.log( linear(450) );     //输出90
```

这段代码创建了一个线性比例尺，其定义域被设置为[0,500]，值域被设置为[0,100]。该比例尺相当于数学中的：

$$y=\frac{1}{5}x, 0\leqslant x\leqslant 500$$

定义了这个比例尺后，就能够很方便地根据 x 计算 y。上面的代码中，分别给 linear 传了 50、250、450 三个数，结果分别为 10、50、90。从这里可以看出，D3 中的比例尺可当作**函数**（这里指编程中的函数）使用，调用时需要传入参数。

5.1.1 线性比例尺

线性比例尺（Linear Scale）是常用比例尺，与线性函数类似，计算线性的对应关系。相关

方法的介绍如下。

- d3.scale.**linear**()

创建一个线性比例尺。

- **linear**(x)

输入一个在定义域内的值 x，返回值域内对应的值。

- linear.**invert**(y)

输入一个在值域内的值，返回定义域内对应的值。

- linear.**domain**([numbers])

设定或获取定义域。

- linear.**range**([values])

设定或获取值域。

- linear.**rangeRound**([values])

代替 range()使用的话，比例尺的输出值会进行四舍五入的运算，结果为整数。

- linear.**clamp**([boolean])

默认被设置为 false，当该比例尺接收一个超出定义域范围内的值时，依然能够按照同样的计算方法计算得到一个值，这个值可能是超出值域范围的。如果设置为 true，则任何超出值域范围的值，都会被收缩到值域范围内。

- linear.**nice**([count])

将定义域的范围扩展成比较理想的形式。例如，定义域为[0.500000543, 0.899995435234]，则使用 nice()之后，其定义域变为[0.5, 0.9]。对于[0.500000543, 69.99997766]这样的定义域，则自动将其变为[0, 70]。

- inear.**ticks**([count])

设定或获取定义域内具有代表性的值的数目。count 默认为 10，如果定义域为[0,70]，则该函数返回[0, 10, 20, 30, 40, 50, 60, 70]。如果 count 设置为 3，则返回[0, 20, 40, 60]。该方法主要用于选取坐标轴刻度。

- linear.**tickFormat**(count,[format])

用于设置定义域内具有代表性的值的表示形式，如显示到小数点后两位，使用百分比的形式显示，主要用于坐标轴上。

以上方法中，linear(x)、invert()、domain()、range()是基础方法，使用形式如下所示：

```
var linear = d3.scale.linear()
           .domain([0,20])          //设置定义域为[0,20]
           .range([0,100]);         //设置值域为[0,100]

console.log( linear(10) );          //输出50
console.log( linear(30) );          //输出150
```

```
console.log( linear.invert(80) ); //输出16
```

前文出现过类似的示例，还是很好理解的。用 linear()计算的结果是，输出都是输入值的 5 倍；而使用 linear.invert()时，输出都是输入值的五分之一。要注意的是，倒数第二行，linear()接收了一个超出定义域范围的值 30，结果还是正常输出其乘以 5 之后的值 150。此输出值也超出了值域的范围，如果不希望其超出范围，可使用 **clamp()**，代码如下所示：

```
linear.clamp(true);
console.log( linear(30) ); //输出100
```

将 clamp()设置为 true 后，超出值域的值会取值域的上下限作为输出。对于输出数值，如果希望对其进行四舍五入，要使用 **rangeRound()**来设置：

```
linear.rangeRound([0,100]);
console.log( linear(13.33) );  //输出67
```

如果不用 rangeRound()重新设置值域，则输出值为 66.64999999999999，其四舍五入后值为 67。如果定义域中有无穷小数，可用 **nice()**，代码如下所示：

```
linear.domain([0.12300000,0.4888888888]).nice();
console.log( linear.domain() );    //输出[0.1, 0.5]

linear.domain([33.611111,45.97745]).nice();
console.log( linear.domain() );    //输出[33, 46]
```

应用 nice()后，定义域变成了比较工整的形式，但是并不是四舍五入。最后讲解 **ticks()**和 **tickFormat()**的用法，它们主要是用在坐标轴上的。请看下面的代码：

```
var linear = d3.scale.linear()
          .domain([-20,20])
          .range([0,100]);

var ticks = linear.ticks(5);
console.log(ticks);    //输出[-20, -10, 0, 10, 20]

var tickFormat = linear.tickFormat(5,"+");

for(var i=0;i<ticks.length;i++){

   //ticks数组中的每一个值，都使用一次tickFormat()函数
   ticks[i] = tickFormat(ticks[i]);
}
console.log(ticks);    //输出["-20", "-10", "+0", "+10", "+20"]
```

这段代码中，比例尺的定义域为[-20, 20]，调用 ticks(5)之后返回一个数组，分别是该定义

域内具有代表性的值。然后，调用 tickFormat()，返回值是一个函数，因此调用时要使用函数的调用方式。最终结果是，ticks 变成了设定的格式。此处设定的格式为"+"：表示如果是正数，则在前面添加一个加号，负数则添加一个减号。除此之外，常用的格式还有"%"、"$"等，遵循迷你语言格式规范。

比例尺的 domain()和 range()最少放入两个数，可以超过两个数，但两者的数量必须相等。以下代码是放入三个数的情况：

```
var scale = d3.scale.linear();
scale.domain([0, 20, 40])
     .range([0, 100, 150]);
var result = scale(30);    //返回值为125
```

这表示有两个线性函数，当输入的值为 30 时，属于 domain 的 20～40 范围，那么输出为 100～150 范围。这里的 30 对应的值为 125，所以 result 的值为 125。

5.1.2　指数和对数比例尺

指数比例尺（Power Scale）和**对数比例尺**（Log Scale）中的很多方法和线性比例尺是一样的，例如 domain()、range()、rangeRound()、clamp()、nice()、ticks()、tickFormat()等，名称和作用都是相同的。但是，指数比例尺多一个 exponent()，用于指定指数；对数比例尺多一个 base()，用于指定底数。两者的用法和原理是一样的，下面以指数比例尺来说明使用方法，先看一段代码：

```
//设置指数比例尺的指数为3
var pow = d3.scale.pow().exponent(3);
console.log( pow(2) ); //输出8
console.log( pow(3) ); //输出27

//设置指数比例尺的指数为0.5，即平方根
pow.exponent(0.5);
console.log( pow(2) ); //输出1.414
console.log( pow(3) ); //输出1.732
```

这样使用的话，效果和 JavaScript 的 Math.pow()是一样的，即为是 *N* 次方运算。既然如此，这样的运算为何要指定定义域 domain 和值域 range 呢。再来看一段代码：

```
var pow = d3.scale.pow()
            .exponent(3)
            .domain([0,3])
            .range([0,90]);
```

```
console.log( pow(1.5) );          //输出11.25
```

这段代码输出 11.25，指数为 3，输入为 1.5。可是 1.5 的 3 次方为 3.375，这是怎么计算的呢？其实，指数比例尺内部调用了线性比例尺，而且把这个线性比例尺定义域的边界变为了其指定次方。在这段代码中，实际上相当于定义了一个线性比例尺，定义域为[0, 27]，值域为[0, 90]。当计算 1.5 的 3 次方得到结果 3.375 之后，再对这个结果应用线性比例尺，最终得到 11.25。为了验证这一点，请看如下代码：

```
//指数比例尺
var pow = d3.scale.pow()
          .exponent(3)
          .domain([0,3])
          .range([0,90]);

//线性比例尺
var linear = d3.scale.linear()
          .domain([0,Math.pow(3,3)])
          .range([0,90]);

//输出11.25
console.log( pow(1.5) );

//输出11.25
console.log( linear(Math.pow(1.5,3)) );
```

和预想的一样，最终输出的两个数值相等。指数比例尺的定义域和值域默认都为[0,1]，因此如果不指定的话，和 Math.pow()没什么区别。

5.1.3 量子和分位比例尺

量子比例尺（Quantize Scale）的定义域是连续的，值域是离散的，根据输入值的不同，结果是其对应的离散值。例如：

定义域：[0,10]

值域：["red","green","blue","yellow","black"]

使用量子比例尺后，定义域将被分隔成 5 段，每一段对应值域的一个值。[0,2)对应 red，[2,4)对应 green，依此类推。请看下面的代码。

```
var quantize = d3.scale.quantize()
               .domain([0,10])
               .range(["red","green","blue","yellow","black"]);
```

```
console.log( quantize(1) );              //red
console.log( quantize(3) );              //green
console.log( quantize(5.9999) );         //blue
console.log( quantize(6) );              //yellow
```

因此，量子比例尺很适合处理类似“数值对应颜色”的问题。例如，要表示中国各省份的GDP，数值越大，就用越深的颜色来表示。此时，可以使用量子比例尺。下面做一个图，图中有几个圆，圆的半径越小，颜色越深。

```
//定义量子化比例尺
var quantize = d3.scale.quantize()
            .domain([50,0])
            .range(["#888","#666","#444","#222","#000"]);

//定义圆的半径
var r = [45,35,25,15,5];

//给body中添加一个svg元素
var svg = d3.select("body").append("svg")
            .attr("width",400)
            .attr("height",400);

//给svg里添加圆
svg.selectAll("circle")
   .data(r)
   .enter()
   .append("circle")
   .attr("cx",function(d,i){ return 50 + i * 30; })
   .attr("cy",50)
   .attr("r",function(d){ return d; })
   .attr("fill",function(d){ return quantize(d); });
```

结果如图 5-1 所示。绘制了五个紧挨着的圆，颜色分别为"#888"、"#666"、"#444"、"#222"、"#000"。

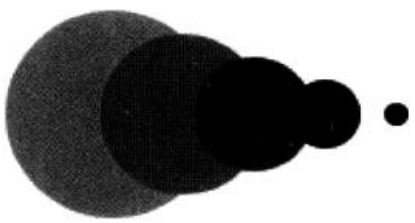


图 5-1　圆的半径越小，颜色越深

与量子比例尺十分类似的是**分位比例尺**（Quantile Scale），也是用于将连续的定义域区域分

成段，每一段对应到一个离散的值上。下面通过一段代码来看看它们的不同之处：

```
//量子比例尺
var quantize = d3.scale.quantize()
                    .domain([0,2,4,10])
                    .range([1,100]);

//分位比例尺
var quantile = d3.scale.quantile()
                    .domain([0,2,4,10])
                    .range([1,100]);

console.log( quantize(3) );    //输出1
console.log( quantile(3) );    //输出100
```

这段代码中，两个比例尺的定义域和值域都是一样的，同样输入 3；但是一个对应到离散数值 1 上，另一个对应到 100 上，这是**两者的分段位置不同**导致的。量子比例尺的分段值为 5，而分位比例尺的分段值为 3。

```
console.log( quantize(4.99) ); //量子比例尺，输出1
console.log( quantize(5) );    //量子比例尺，输出100
console.log( quantile(2.99) ); //分位比例尺，输出1
console.log( quantile(3) );    //分位比例尺，输出100
```

从此处可看出，两者的分段处的值是不同的。量子比例尺的分段值**只与定义域的起始值和结束值有关**，其中间有多少其他数值都没有影响，分段值取其算数平均值。而分位比例尺的分段值**与定义域中存在的数值都有关**。使用 quantile.quantiles()可以查询分位比例尺的分段值。

5.1.4 阈值比例尺

阈值（Threshold）又叫临界值，是指一个效应能够产生的最低值或最高值。**阈值比例尺**（Threshold Scale）是通过设定阈值，将连续的定义域映射到离散的值域里，与量子比例尺和分位比例尺相似。下面来看一个例子：

```
var threshold = d3.scale.threshold()
          .domain([10,20,30])
          .range(["red","green","blue","black"]);

console.log( threshold(5) );   //输出red
console.log( threshold(15) );  //输出green
console.log( threshold(25) );  //输出blue
console.log( threshold(35) );  //输出black
```

这段代码的阈值比例尺定义了三个阈值：10、20、30；则空间被这三个阈值分为四段，分别为：负无穷到 10、10 到 20、20 到 30、30 到正无穷。值域设定了四个离散的值，则定义域的四段分别对应到这四个值上。因此，最终的输出结果分别为 red、green、blue、black。阈值比例尺可以使用 invertExtent()通过值域求定义域。下面来看看定义域是否如所预想的一样。

```
//输出[undefined, 10]
console.log( threshold.invertExtent("red") );

//输出[10, 20]
console.log( threshold.invertExtent("green") );

//输出[20, 30]
console.log( threshold.invertExtent("blue") );

//输出[30, undefined]
console.log( threshold.invertExtent("black") );
```

阈值比例尺、量子比例尺、分位比例尺，三者十分相似，都是将连续的定义域映射到离散的值域里，开发者要根据需求选择使用。

5.2　序数比例尺

定量比例尺的定义域都是连续的，值域有连续的也有离散的。**序数比例尺**（Ordinal Scale）的定义域和值域都是离散的。现实中会有这样的需求，通过输入一些离散的值（如名称、序号、ID 号等），要得到另一些离散的值（如颜色、头衔等），这种时候就要考虑序数比例尺。

- d3.scale.**ordinal**()

构建一个序数比例尺。

- **ordinal**(x)

输入定义域内一个离散值，返回值域内一个离散值。

- ordinal.**domain**([values])

设定或获取定义域。

- ordinal.**range**([values])

设定或获取值域。

- ordinal.**rangePoints**(interval[, padding])

代替 range()设定值域。接收一个连续的区间，然后根据定义域中离散值的数量将其分段，分段值即作为值域的离散值。

- ordinal.**rangeRoundPoints**(interval[, padding])

和 rangePoints()一样，但是会将结果取整。

- ordinal.**rangeBands**(interval[, padding[, outerPadding]])

代替 range()设定值域。与 rangePoints()一样，也是接收一个连续的区间，然后根据定义域中离散值的数量将其分段，但是其分段方法是不同的。

- ordinal.**rangeRoundBands**(interval[, padding[, outerPadding]])

和 rangeBands()一样，但是会将结果取整。

- ordinal.**rangeBand**()

返回使用 rangeBands()设定后每一段的宽度。

- ordinal.**rangeExtend**()

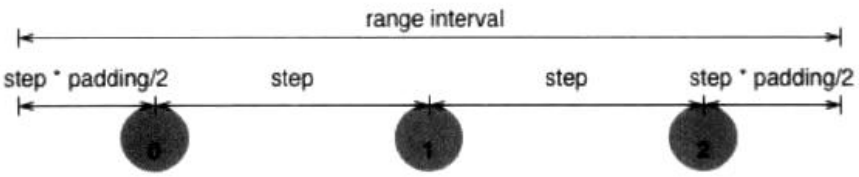
返回一个数组，数组里存有值域的最大值和最小值。

首先来看一个简单的例子。下面定义一个序数比例尺，定义域设置为[1, 2, 3, 4, 5]五个离散值，值域设置为[10, 20, 30, 40, 50]五个离散值。其输入结果如下：

```
var ordinal = d3.scale.ordinal()
            .domain([1,2,3,4,5])
            .range([10,20,30,40,50]);

console.log( ordinal(1) ); //输出10
console.log( ordinal(3) ); //输出30
console.log( ordinal(5) ); //输出50
console.log( ordinal(8) ); //输入值不在定义域中，输出10
```

由此可见，1 对应 10，3 对应 30，5 对应 50，与定义域和值域的排列次序一致。但是最后一行输入值为 8，不在定义域内，输出值为 10；先不管其输出值有没有道理，总之不要输入超出定义域的值。

但是，如果一个一个地设定值域的值，使其对应到定义域上，比较麻烦。D3 提供了 rangePoints()和 rangeRoundPoints()用于解决此问题：只要接收一个连续的区间，即可自动计算出相应的离散值。这两个方法都有两个参数：interval 和 padding。interval 是区间，padding 是边界部分留下的空白，可省略，默认为 0。其意义如图 5-2 所示。



* https://github.com/mbostock/d3/wiki/Ordinal-Scales

图 5-2　rangePoints()和 rangeRoundPoints()中参数的含义

图 5-2 中，range interval 就是 rangePoints()的第一个参数 interval 的值，是一个范围，如[0,100]；padding 是第二个参数；step 是根据定义域的数值计算得到的值。图中圆圈所代表的点，就是计

算得到的离散值。请看下面的代码：

```
var ordinal = d3.scale.ordinal()
            .domain([1,2,3,4,5])
            .rangePoints([0,100]);

console.log( ordinal.range() );//输出值域[0, 25, 50, 75, 100]
console.log( ordinal(1) );     //输出0
console.log( ordinal(3) );     //输出50
console.log( ordinal(5) );     //输出100
```

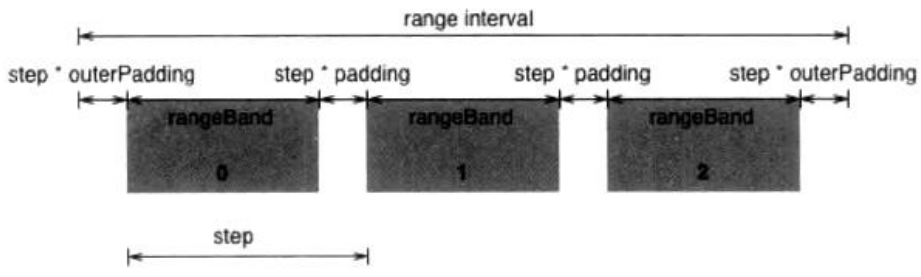
上面代码的序数比例尺中，rangePoints()的第一个参数为[0, 100]，第二个参数省略。对应到图 5-2 中的话，则 range interval 等于[0, 100]，padding 等于 0，step 等于 25。因此，得到了上面输出的 5 个离散值，与定义域的 5 个值是一一对应的。下面来看看 padding 有设定值的情况。

```
ordinal.rangePoints([0,100],5);

//输出[27.77777, 38.88888, 50, 61.11111, 72.22222]
console.log( ordinal.range() );
```

这样设定后，padding 等于 5，step 等于 11.11111。则 step * padding/2 等于 27.77777，正是输出数组的第一个值。这样，有时输出的数组是无穷小数，如果希望其都是整数，可用 rangeRoundPoints()，代码如下。

```
ordinal.rangeRoundPoints([0,100],5);

//输出[28, 39, 50, 61, 72] ，结果被四舍五入取整了
console.log( ordinal.range() );
```

D3 还提供了 rangeBands()和 rangeRoundBands()。与 rangePoints()稍有不同的是，其接收三个参数：interval、padding、outerPadding。各参数的意义如图 5-3 所示，range interval 是范围，padding 和 outerPadding 分别是内部和边界的空白的参数，默认为 0。方框中的 rangeBand，表示一个区间；而该区间的起点，就是得到的离散值。



* https://github.com/mbostock/d3/wiki/Ordinal-Scales

图 5-3　rangeBands()中参数的含义

rangeBand()可返回 rangeBand 的值。要注意函数名的区别：rangeBands()后面有 s，用于设置值域；rangeBand()后面没有 s，用于返回图 5-3 中的 rangeBand 的值。下面看一个简单的例子：

```
var bands = d3.scale.ordinal()
            .domain([1,2,3,4,5])
            .rangeBands([0,100]);

//输出[0, 20, 40, 60, 80]
console.log( bands.range() );

//输出20
console.log( bands.rangeBand() );
```

这段代码中，padding 和 outerPadding 都没有设定，默认为 0。计算可得，rangeBand 为 20，值域有五个离散的值，分别是每一个 rangeBand 区域的起点，即[0, 20, 40, 60, 80]。下面再测试设定了空白的情况：

```
bands.rangeBands([0,100],0.5,0.2);

//输出[4.08163, 24.48979, 44.89795, 65.30612, 85.71428]
//（小数点后有手动省略）
console.log( bands.range() );

//输出10.20408
console.log( bands.rangeBand() );
```

这段代码中，padding 为 0.5，outerPadding 为 0.2。对应到图 5-3 中，step 计算的值约等于 20，因此左右边界的空白 step * outerPadding 约等于 4，即输出数组的第一个值。step * padding 约等于 10，即每个 rangeBand 之间的空白长度。

D3 提供了几个获取颜色的序数比例尺。制作图表时，常需要设定各图形元素的颜色，每次都要手动设定很麻烦，如果对颜色没有特殊要求直接使用这些**颜色比例尺**即可。并且，它们的颜色都经过精心的色彩搭配，相当美观。颜色比例尺共有 4 个。

- d3.scale.**category10**()：10 种颜色。
- d3.scale.**category20**()：20 种颜色。
- d3.scale.**category20b**()：20 种颜色。
- d3.scale.**category20c**()：20 种颜色。

这四个都是序数比例尺，输入离散值后返回值也是离散值。例如 category10()提供了 10 种颜色，分别是：#1f77b4、#ff7f0e、#2ca02c、#d62728、#9467bd、#8c564b、#e377c2、#7f7f7f、

#bcbd22、#17becf。请看下面的代码：

```
var color = d3.scale.category10();

//输出#1f77b4
console.log( color(1) );

//输出#ff7f0e
console.log( color("zhangsan") );
```

可以看到，无论输入值是什么样的离散值，该比例尺都按照颜色顺序返回：先返回了#1f77b4，再返回#ff7f0e，如果后面还有则继续返回后面的值。使用这四个比例尺来设定颜色以后会经常见到，例如像下面这样的应用：

```
var width = 600;
var height = 600;
var dataset = d3.range(5); //返回[0,1,2,3,4,5]

//定义表示颜色的序数比例尺
var color = d3.scale.category10();

var svg = d3.select("body").append("svg")
                    .attr("width",width)
                    .attr("height",height);

//绘制圆
var circle = svg.selectAll("circle")
                .data(dataset)
                .enter()
                .append("circle")
                .attr("cx",function(d,i){
                    return 30 + i * 80;
                })
                .attr("cy",100)
                .attr("r",30)
                .attr("fill",function(d,i){
                    比例尺
                    return color(i);
                });
```

d3.range()返回一个等差数列，但是此处仅使用其长度，不使用数组的各项值。以上代码绘制了 5 个圆，在给每个圆设置颜色的时候，使用了 color(i)；**color 是颜色比例尺**，i 是被绑定数

据的索引号，被当作 color 的参数使用。但是，不一定非得使用索引号，别的离散值也可以，颜色都会按顺序返回。如图 5-4 所示，为方便读者了解颜色值，每个圆的下方都加了用 16 进制数表示的 RGB 值。

图 5-4　使用序数比例尺设定颜色

5.3　坐标轴

坐标轴（Axis）在很多图表中都可见到，例如柱形图、折线图、散点图等。坐标轴由一组线段和文字组成，坐标轴上的点由一个坐标值确定。但是，如果使用 SVG 的直线和文字一笔一画地绘制坐标轴，工作量很大。D3 提供了坐标轴的制作方法，需要配合本章所说的比例尺一起使用。开发者仅仅需要几行代码，就能够生成各式各样的坐标轴。与坐标轴相关的方法介绍如下。

- d3.svg.**axis**()

创建一个默认的新坐标轴。

- **axis**(selection)

将此坐标轴应用到指定的选择集上，该选择集需要包含有<svg>或<g>元素。

- axis.**scale**([scale])

设定或获取坐标轴的比例尺。

- axis.**orient**([orientation])

设定或获取坐标轴的方向，有四个值：top、bottom、left、right。top 表示水平坐标轴的刻度在直线下方，bottom 表示水平坐标轴的刻度在直线上方，left 表示垂直坐标轴的刻度在直线右方，right 表示垂直坐标轴的刻度在直线左方。

- axis.**ticks**([argument...])

设定或获取坐标轴的分隔数，默认为 10。例如，设定为 5，则坐标轴上的刻度数量为 6，分段数为 5。这个函数会调用比例尺的 ticks()。

- axis.**tickValues**([values])

设定或获取坐标轴的指定刻度。例如，参数为[1,2,3,6,7,8]，则在这几个值上会有刻度。

- axis.**tickSize**([inner, outer])

设定或获取坐标轴的内外刻度的长度。默认都为 6。

- axis.**innerTickSize**([size])

设定或获取坐标轴的内刻度的长度。内刻度指不是两端的刻度。

- axis.**outerTickSize**([size])

设定或获取坐标轴的外刻度的长度。外刻度指两端的刻度。

- axis.**tickFormat**([format])

设定或获取刻度的格式。

5.3.1　绘制方法

在第 2 章介绍过，SVG 中有<path>、<line>、<text>元素，D3 所绘制的坐标轴就是由这三种元素组成的。其中，坐标轴的主直线是由<path>绘制的，刻度是由<line>绘制的，刻度文字是由<text>所绘制的。先看一个简单的例子：

```
var width = 600;
var height = 600;

var svg = d3.select("body").append("svg")
           .attr("width",width)
           .attr("height",height);

//用于坐标轴的线性比例尺
var xScale = d3.scale.linear()
               .domain([0,10])
               .range([0,300]);

//定义坐标轴
var axis = d3.svg.axis()
           .scale(xScale)           //使用上面定义的比例尺
           .orient("bottom");  //刻度方向向下

//在svg中添加一个包含坐标轴各元素的g元素
var gAxis = svg.append("g")
       .attr("transform","translate(80,80)"); //平移到(80,80)

//在gAxis中绘制坐标轴
axis(gAxis);
```

坐标轴的所有图形元素需放入<svg>或<g>里，建议新建一个 g 元素来控制，而不要直接放在<svg>里，因为<svg>中通常还包含有其他的图形元素。上述代码中，先在<body>中添加了<svg>，然后再在<svg>中添加了<g>，坐标轴就绘制在这个<g>中。绘制之后，HTML 的元素结构如图 5-5 所示。class 为 tick 的<g>元素就是刻度，每一个刻度里都包含有<line>和<text>。坐标轴的主直线是最下方的<path>，其 class 是 domain。

```
▼<svg width="600" height="600">
 ▼<g transform="translate(80,80)">
  ▼<g class="tick" transform="translate(0,0)" style="opacity: 1;">
     <line y2="6" x2="0"></line>
     <text dy=".71em" y="9" x="0" style="text-anchor: middle;">0</text>
   </g>
  ▶<g class="tick" transform="translate(30,0)" style="opacity: 1;">…</g>
  ▶<g class="tick" transform="translate(60,0)" style="opacity: 1;">…</g>
  ▶<g class="tick" transform="translate(90,0)" style="opacity: 1;">…</g>
  ▶<g class="tick" transform="translate(120,0)" style="opacity: 1;">…</g>
  ▶<g class="tick" transform="translate(150,0)" style="opacity: 1;">…</g>
  ▶<g class="tick" transform="translate(180,0)" style="opacity: 1;">…</g>
  ▶<g class="tick" transform="translate(210,0)" style="opacity: 1;">…</g>
  ▶<g class="tick" transform="translate(240,0)" style="opacity: 1;">…</g>
  ▶<g class="tick" transform="translate(270,0)" style="opacity: 1;">…</g>
  ▶<g class="tick" transform="translate(300,0)" style="opacity: 1;">…</g>
   <path class="domain" d="M0,6V0H300V6"></path>
  </g>
 </svg>
```

图 5-5　组成坐标轴的元素结构

xScale 是一个线性比例尺，其定义域为[0, 10]，这是坐标轴刻度值的范围。值域为[0, 300]，这是坐标轴实际的像素长度。定义坐标轴时，使用 scale(xScale)指定比例尺。代码的最后一行，axis(gAxis)表示的是在 gAxis 选择集中绘制坐标轴，gAxis 是在<svg>中新添加的分组元素。绘制结果如图 5-6 所示。

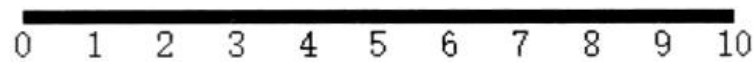


图 5-6　原始坐标轴

看到图 5-6 可能会很吃惊，这个坐标轴确实太丑了，最主要是刻度直线都没有显示出来。这是因为还没有为坐标轴定制样式，样式如下：

```
.axis path,
.axis line{
    fill: none;
    stroke: black;
    shape-rendering: crispEdges;
}
```

```
.axis text {
    font-family: sans-serif;
    font-size: 11px;
}
```

令坐标轴设置成以上样式。绘制坐标轴前，添加下面一行代码：

```
gAxis.attr("class","axis");
axis(gAxis);
```

添加样式后，如图 5-7 所示。

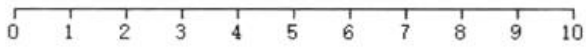


图 5-7　为坐标轴添加样式后

这样就美观多了。上面使用了 axis(gAxis)的方式来指定绘制的位置，除此之外，还有一种常用的方式，即使用第 4.5.7 节的 call()：

```
gAxis.call(axis);
```

这种方式的调用很常见，需要牢记。

5.3.2　刻度

说到坐标轴的属性，基本上是在说刻度，例如刻度的方向、间隔、长度、文字格式等。第 5.3.1 节的坐标轴，设置了刻度的方向 orient("bottom")，因此刻度在直线的下方。如果要设置在什么值上标出刻度，使用 ticks()和 tickValues()。请看下面的例子：

```
var axisLeft = d3.svg.axis()
            .scale(scale)
            .orient("left")
            .ticks(5);

var axisRight = d3.svg.axis()
            .scale(scale)
            .orient("right")
            .tickValues([3,4,5,6,7]);
```

比例尺 scale 的定义域为[0,10]。这段代码定义了两个坐标轴，刻度分别位于左边和右边，刻度值分别用 **ticks()**和 **tickValues()**来指定，结果如图 5-8 所示，请注意刻度的区别。

上面绘制的坐标轴，刻度的直线都是相同长度的；有时候也需要不同长度的，最常见的是首尾两个刻度的长度比内部要长。此时需用到 tickSize()，代码如下：

```
var axisTop = d3.svg.axis()
              .scale(scale)
              .orient("top")
              .ticks(5)
              .tickSize(2,4);
```

tickSize()的第一个参数是内部刻度的直线长度，第二个参数是首尾两个刻度的直线长度；也可以用 **innerTickSize()**和 **outerTickSize()**分别进行设置。如图 5-9 所示，两端的刻度线比内部的要长。

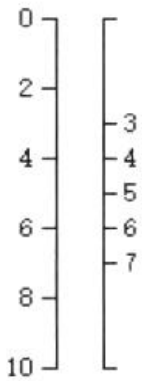


图 5-8　刻度的方向和刻度值

图 5-9　刻度线的长度

刻度文字的格式通过 **tickFormat()**设置，此处还需用到 d3.format()，它返回的对象作为 tickFormat()的参数。在 d3.format()的参数里，可指定刻度文字的格式。例如：

```
.tickFormat(d3.format("$0.1f"));
```

结果如图 5-10 所示。

图 5-10　文字的格式

文字格式的规则遵循迷你语言格式规范。

5.3.3　各比例尺的坐标轴

坐标轴必须要设置一个比例尺，根据比例尺的不同可以得到不同的坐标轴。使用得最多的是线性比例尺。下面来看看随着比例尺的不同，坐标轴的刻度是怎样变化的。

```
var linear = d3.scale.linear()
               .domain([0,1])
               .range([0,500]);

var pow = d3.scale.pow()
            .exponent(2)
            .domain([0,1])
            .range([0,500]);

var log = d3.scale.log()
            .domain([0.01, 1])
            .range([0, 500]);
```

以上代码分别定义了线性、指数、对数比例尺。将它们绘制成坐标轴后，如图 5-11 所示。

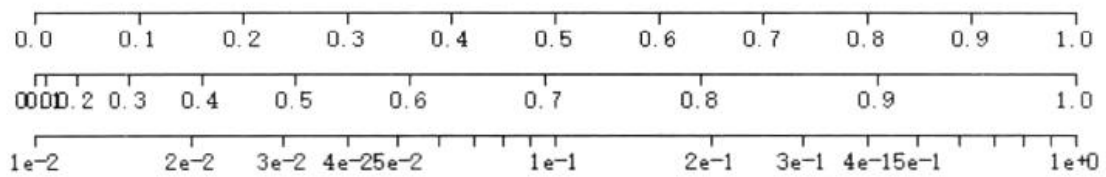


图 5-11　从上至下，分别对应线性、指数、对数比例尺

5.4　柱形图的坐标轴

第 4 章制作的柱形图没有添加坐标轴。另外，当时使用数值的大小来直接表示像素的多少，这种方式并不好。本节为其添加坐标轴，并且使用比例尺使得柱形的长度可以自由伸缩，即数值 50 不直接用 50 个像素来表示。先为柱形图定义比例尺：

```
//x轴宽度
var xAxisWidth = 300;

//y轴宽度
var yAxisWidth = 300;

//x轴比例尺（序数比例尺）
var xScale = d3.scale.ordinal()
              .domain(d3.range(dataset.length))
              .rangeRoundBands([0,xAxisWidth],0.2);
```

```
//y轴比例尺（线性比例尺）
var yScale = d3.scale.linear()
           .domain([0,d3.max(dataset)])
           .range([0,yAxisWidth]);
```

首先定义两个变量，用来保存 *x* 和 *y* 轴的宽度。*x* 轴用于表示柱形的序号，可见定义域是离散的，因此使用序数比例尺。*y* 轴用于伸缩柱形的长度，使用线性比例尺。*x* 轴的序数比例尺中，定义域为 d3.range(dataset.length)，这相当于一个等差数列[0, 1, 2, 3, ...]，即可用于表示柱形的序号。值域使用了 rangeRoundBands()，其说明在本章中介绍过，第一个参数是范围，第二个参数是间隔。这么设置之后，可用 rangeBand()返回的值作为柱形的宽度。因此，rangeRoundBands()非常适合制作柱形图。

有了比例尺之后，矩形的位置、长度等都要用比例尺来计算。这么做之后，数据和绘制就能够完全分开，而且只需要修改比例尺，即可将图表自由伸缩。下面是修改后所设置矩形参数的代码。同样，矩形文字的代码也需要改变，修改方法是类似的，这里不做叙述。

```
.attr("x", function(d,i){        //设置矩形的x坐标
      return padding.left + xScale(i);
})
.attr("y", function(d){          //设置矩形的y坐标
      return height- padding.bottom - yScale(d);
})
.attr("width",xScale.rangeBand()) //设置矩形的宽度
.attr("height",function(d){      //设置矩形的高度
      return yScale(d);
});
```

这样绘制出来的矩形，不仅高度能够按比例伸缩，各柱形之间也会拥有空格，这是由 rangeBand()做到的，请参考第 5.2 节的内容来理解。绘制好矩形和文字后，要定义坐标轴：

```
//x轴
var xAxis = d3.svg.axis()
              .scale(xScale)
              .orient("bottom");

//重新设置y轴比例尺的值域，与原来的相反
yScale.range([yAxisWidth,0]);

//y轴
var yAxis = d3.svg.axis()
              .scale(yScale)
              .orient("left");
```

本例中，y 轴是从下往上的，越往上，值越大。因此，y 轴比例尺的值域需要设置为相反的值。坐标轴定义完之后，在<svg>添加两个<g>，分别用来放 x 轴和 y 轴的元素。然后，平移到目标位置，添加 x 轴和平移的代码如下所示，y 轴与此类似：

```
svg.append("g")
    .attr("class","axis")
    .attr("transform","translate(" + padding.left + "," +
            (height - padding.bottom) +  ")")
    .call(xAxis);
```

x 轴被平移到左下角，y 轴从左上角开始绘制。如图 5-12 所示，一个功能齐全的柱形图做好了。

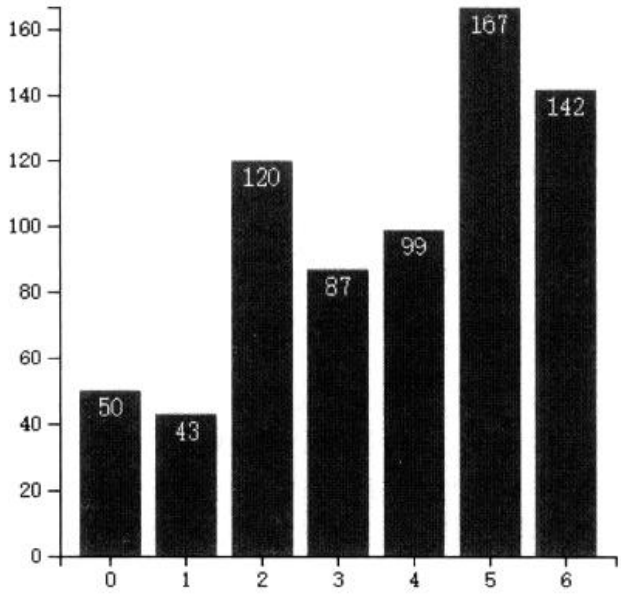


图 5-12　添加了坐标轴的柱形图

5.5　散点图的制作

为了熟练掌握比例尺和坐标轴的使用，本节再制作一种简单图表。**散点图**（Scatter Chart），通常是一横一竖两个坐标轴，数据是一组二维坐标，分别对应两个坐标轴，与坐标轴对应的地方打上点（圆）。由概念容易猜到，需要的元素包括 circle（圆）和 axis（坐标轴）。

需要进行可视化的数据如下：

```
//圆心数据
var center = [[0.5,0.5],[0.7,0.8],[0.4,0.9],
```

```
            [0.11,0.32],[0.88,0.25],[0.75,0.12],
            [0.5,0.1],[0.2,0.3],[0.4,0.1],[0.6,0.7]];
```

数组中的每一项都是一个数组，子数组的第一项表示 x 值，第二项表示 y 值。实际应用中，x 轴和 y 轴可能对应着不同的意义，单位也可能不同，例如人口-GDP、烟龄-肺癌率等。这些数据都不可能直接用像素作为单位来绘制，因为类似（0.5, 0.5）、（0.7, 0.8）这样的位置，即便绘制了也会看到圆都挤到一块，分不清彼此。因此，要先使用比例尺将它们放大。与第 5.4 节的柱形图类似，定义如下比例尺：

```
//x轴比例尺
var xScale = d3.scale.linear()
              .domain([0, 1.2 * d3.max(center,function(d){
                      return d[0];
              })])
              .range([0,xAxisWidth]);

//y轴比例尺
var yScale = d3.scale.linear()
                  .domain([0, 1.2 * d3.max(center,function(d){
                          return d[1];
                  })])
                  .range([0,yAxisWidth]);
```

xAxisWidth 和 yAxisWidth 可以根据需要设定。要注意，两个比例尺都是线性比例尺，在设定定义域 domain 时，使用了 d3.max()，这是一个求数组最大值的函数，相关解释在第 4.6.2 节中有阐述。对于 x 轴的比例尺来说，这里的意思是，对于 center 数组的每一项，返回其子数组的第一项（d[0]）组成一个新数组，然后再求最大值。最大值前面乘了一个 1.2，这是为了使得散点图不会有某一点存在于坐标轴的边缘上。下面在 svg 中绘制图形，先绘制圆，代码如下：

```
//外边框
var padding = { top: 30 , right: 30, bottom: 30, left: 30 };

//绘制圆
var cirlce = svg.selectAll("circle")
          .data(center)          //绑定数据
          .enter()               //获取enter部分
          .append("circle")      //添加circle元素，使其与绑定数组的长度一致
          .attr("fill","black")                //设置颜色为black
          .attr("cx", function(d){             //设置圆心的x坐标
              return padding.left + xScale(d[0]);
          })
```

```
        .attr("cy", function(d){          //设置圆心的y坐标
            return height- padding.bottom - yScale(d[1]);
        })
        .attr("r", 5 );
```

注意粗体字部分，分别使用 x 轴和 y 轴的比例尺放大数据。其余的绘制坐标轴的部分和第 5.4 节的柱形图完全一样，结果如图 5-13 所示。

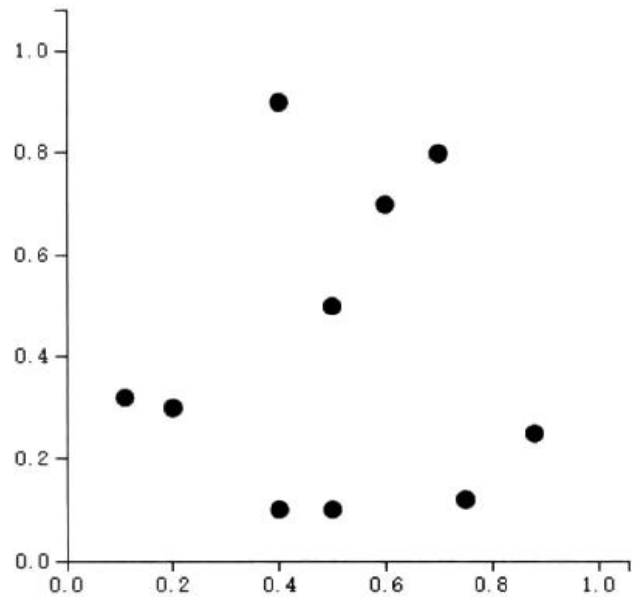


图 5-13　散点图

第 6 章

绘制

本章内容包括：

- 颜色
- 各种路径生成器
- 制作折线图

D3 本身没有作图的功能，它只能为我们计算出作图所需的数据。因此，实际作图是需要指定一个画板的，这个画板就是 SVG（可缩放的矢量图形），在前几章已经使用过了。虽然 D3 不是必须在 SVG 上作图，但 SVG 一定是最适合 D3 的。

SVG 的图形元素包括矩形<rect>、圆形<circle>、线段<line>、路径<path>等，其中路径是最强大的，可以表示其他所有图形。但是，路径元素<path>的路径值比较复杂，如果手动计算必定要花很大工夫。为此，D3 提供了数量众多的路径生成器，以完成这一复杂工作。

第1节，讲述 RGB 和 HSL 颜色，以及颜色插值函数。

第2节至第 7 节，分别介绍线段生成器、区域生成器、弧生成器、符号生成器、弦生成器、对角线生成器，共六个生成器的用法。

第 8 节，制作折线图，用于复习本章的知识。

6.1 颜色

计算机中的颜色，常用的标准有 RGB 和 HSL。

RGB 色彩模式是通过对红（Red）、绿（Green）、蓝（Blue）三个颜色通道相互叠加来得到各式各样的颜色。三个通道的值的范围都为 0～255，因此总共能表示 16777216（256×256×256）种，即一千六百多万种颜色。几乎包括了人类所能识别的所有颜色，是应用最广泛的色彩模式。

HSL 色彩模式是通过对色相（Hue）、饱和度（Saturation）、明度（Lightness）三个通道的相互叠加来得到各种颜色的。其中，色相的范围为 0°～360°，饱和度的范围为 0～1，明度的范围为 0～1。色相的取值是一个角度，每个角度可以代表之中的一种颜色，需要记住的是 0° 或 360° 代表红色，120° 代表绿色，240° 代表蓝色。饱和度的数值越大，颜色越鲜艳，灰色越少。明度值用于控制色彩的明暗变化，值越大，越明亮，越接近于白色；值越小，越暗，越接近黑色。

RGB 颜色和 HSL 颜色可以相互转换。

6.1.1 RGB

D3 中提供了 RGB 颜色的创建、调整明暗、转换为 HSL 模式的方法，介绍如下。

- d3.**rgb**(r, g, b)

分别输出 r、g、b 值来创建颜色，范围都为[0, 255]。

- d3.**rgb**(color)

输入相应的字符串来创建颜色，例如：

  - RGB 的十进制值："rgb(255, 255, 255) "。
  - HSL 的十进制值："hsl(120, 0.5, 0.5) "。
  - RGB 的十六进制值："#ffeeaa"。
  - RGB 的十六进制值的缩写形式："#fea"。
  - 颜色名称："red"、 "white"。

- rgb.**brighter**([k])

颜色变得更明亮。RGB 各通道的值乘以 0.7 ^ -k。如果 k 省略，k 的值为 1。只有当某通道的值的范围在 30～255 之间时，才会进行相应的计算。

- rgb.**darker**([k])

颜色变得更暗。RGB 各通道的值乘以 0.7 ^ k。

- rgb.**hsl**()

返回该颜色对应的 HSL 值。

- rgb.**toString**()

以字符串形式返回该颜色值，如"#ffeeaa"。

brighter()和 **darker()**返回一个新的颜色对象，不会改变当前颜色对象。**hsl()**返回当前颜色对应的 HSL 值，也是一个新的对象。请看以下示例：

```
var color1 = d3.rgb(40,80,0);
```

```
var color2 = d3.rgb("red");
var color3 = d3.rgb("rgb(0,255,255)");

//将color1的颜色变亮，返回值的颜色为 r: 81, g: 163, b:0
console.log( color1.brighter(2) );
//原颜色值不变，依然是 r: 40, g: 80, b:0
console.log( color1 );

//将color2的颜色变亮，返回值的颜色为 r: 124, g: 0, b:0
console.log( color2.darker(2) );
//原颜色值不变，依然是 r: 255, g: 0, b:0
console.log( color2 );

//输出color3颜色的HSL值，h: 180, s: 1, l: 0.5
console.log( color3.hsl() );

//输出#00ffff
console.log( color3.toString() );
```

要注意，brighter()、darker()、hsl()返回的都是对象，不是字符串。

6.1.2 HSL

HSL 颜色的创建和使用与 d3.rgb 几乎一样，只是各颜色通道的意义不同。

- d3.**hsl**(h, s, l)

根据 h、s、l 的值来创建 HSL 颜色。

- d3.**hsl**(color)

根据字符串来创建 HSL 颜色。

- hsl.**brighter**([k])。

变得更亮。

- hsl.**darker**([k])

变得更暗。

- hsl.**rgb**()

返回对应的 RGB 颜色。

- hsl.**toString**()

以 RGB 字符串形式输出该颜色。

对 HSL 颜色来说，**brighter()**和 **darker()**更好理解，因为 HSL 的“L”就是明亮度。也就是说，应用 brighter()或 darker()后，只有 h、s、l 中的第三个颜色通道“l”发生变化。请看以下代码：

```
var hsl = d3.hsl(120,1.0,0.5);
```

```
//返回的对象中，h:120, s:1.0, l:0.714
console.log( hsl.brighter() );

//返回的对象中，h:120, s:1.0, l:0.35
console.log( hsl.darker() );

//返回的对象中，r:0, g:255, b:0
console.log( hsl.rgb() );

//输出#00ff00
console.log( hsl.toString() );
```

一般来说，编程人员喜欢使用 RGB 颜色，比较好理解。美术人员更喜欢使用 HSL 颜色，方便调整饱和度和亮度。

6.1.3　插值

如果要计算介于两个颜色之间的颜色，需要用到**插值**（Interpolation）。D3 提供了 d3.interpolateRgb()来处理 RGB 颜色之间的插值运算，d3.interpolateHsl()来处理 HSL 颜色之间的插值运算。更方便的是用 d3.interpolate()，它会自动判断颜色的类型。d3.interpolate()也可以处理数值、字符串等之间的插值。请看下面的例子：

```
var a = d3.rgb(255,0,0);   //红色
var b = d3.rgb(0,255,0);   //绿色

var compute = d3.interpolate(a,b);

console.log( compute(0) );     //输出#ff0000
console.log( compute(0.2) );   //输出#cc3300
console.log( compute(0.5) );   //输出#808000
console.log( compute(1) );     //输出#00ff00

console.log( compute(2) );     //输出#00ff00
console.log( compute(-1) );    //输出#ff0000
```

这段代码里，定义了两个 RGB 颜色：**红（255, 0, 0）**和**绿（0, 255, 0）**。然后，以这两个颜色对象作为 **d3.interpolate(a, b)**的参数。d3.interpolate 返回一个函数，保存在变量 **compute** 里。于是，compute 可当作函数使用，参数是一个数值。

当数值为 0 时，返回红色；

当数值为 1 时，返回绿色。

当数值为0～1之间的值时，返回介于红色和绿色之间的颜色。
如果数值超出1，则返回绿色；数值小于0，则返回红色。

6.2 线段生成器

在SVG区域里，线段元素是：

```
<line x1="20" y1="20" x2="300" y2="100" />
```

添加元素时，可以使用append()，再设置属性，例如：

```
svg.append("line")
   .attr("x1",20)
   .attr("y1",20)
   .attr("x2",300)
   .attr("y2",100);
```

还有一种添加直线的方法：使用路径元素<path>。路径元素也是可以绘制直线的。例如，相同的直线可用：

```
<path d="M20,20L300,100" />
```

添加。换成D3添加元素的方式，代码如下所示：

```
svg.append("path")
   .attr("d","M20,20L300,100");
```

这样做虽然是成功绘制了该路径，但有一个问题：类似M20,20L300,100的字符串，是怎么得到的呢？此处只有两个点，可以手动输入，如果有成百上千个点，极不方便。因此，D3中引入了**路径生成器**（Path Generator）的概念，能够自动根据数据生成路径。用于生成线段的程序，叫作**线段生成器**（Line Generator）。

线段生成器由**d3.svg.line()**创建，先看一个简单例子。

```
//线段的点数据，每一项是一个点的x和y坐标
var lines = [[80,80],[200,100],[200,200],[100,200]];

//创建一个线段生成器
var linePath = d3.svg.line();

//添加路径
svg.append("path")
   .attr("d",linePath(lines)) //使用了线段生成器
   .attr("stroke","black")
```

```
    .attr("stroke-width","3px")
    .attr("fill","none");
```

线段的数据有四个点，保存在变量 lines 里。线段生成器保存在变量 linePath 里，如此 linePath 可当作函数使用：linePath(lines)根据数据 lines 生成路径。结果是一段折线，如图 6-1 所示。

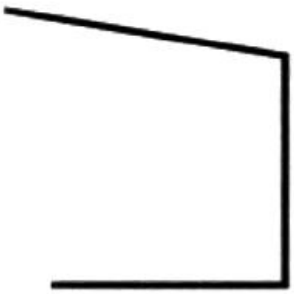


图 6-1　线段生成器

有一个问题：线段生成器是如何从数据生成路径的，又是怎么知道在 lines 数组中，每一项（例如[200, 100]）的第一个值是 x 坐标，第二个值是 y 坐标。如果希望第二个值是 x 坐标，第一个值是 y 坐标，该怎么设定呢。下面介绍与线段生成器相关的方法。

- d3.svg.**line**()

创建一个线段生成器。

- **line**(data)

使用线段生成器绘制 data 数据。

- line.**x**([x])

设置或获取线段 x 坐标的访问器，即使用什么数据作为线段的 x 坐标。

- line.**y**([y])

同上，设置或获取 y 坐标的访问器。

- line.**interpolate**([interpolate])

设置或获取线段的插值模式，共有 13 种。

- line.**tension**([tension])

设置或获取张力系数，当插值模式为 cardinal、cardinal-open、cardinal-closed 的时候有效。

- line.**defined**([defined])

设置或获取一个访问器，用于确认线段是否存在，只有判定为存在的数据才被绘制。

line.x()和 **line.y()**可制定规则：如何从数据中获取 x 和 y 坐标。这种方法称为**访问器**，以后还会经常见到。line.x()和 line.y()默认为：

```
function x(d) {
    return d[0];     //第一个值是x坐标
}
```

```
function y(d) {
    return d[1];     //第二个值是y坐标
}
```

因此，在图 6-1 的例子中，对于数组 lines 的某一项，例如[200, 100]，第一个值 200 被看作是 *x* 坐标，第二个值 100 是 *y* 坐标。下面对 x 和 y 访问器做一些修改，代码如下：

```
var lines = [80,120,160,200,240,280];

var linePath = d3.svg.line()
                .x(function(d){ return d; })
                .y(function(d,i){ return i%2==0 ? 40 : 120; });
```

如此，调用 linePath(lines)所得的路径中，各顶点为：(80, 40)、(120, 120)、(160, 40)、(200, 120)、(240, 40)、(280, 120)。以第一个点为例：

```
x = 80 （d等于80）
y = 0%2 == 0 ? 40 : 120 （i等于0，最终表达式的值为40）
```

依此类推，可计算得到各点坐标。绘制此路径，结果如图 6-2 所示。

图 6-2　x 和 y 的访问器

图 6-2 的路径元素输出到控制台的结果，如图 6-3 所示。可以看到，路径是 M80,40L120,120...，字符串很长。手动输入该路径是很费劲的，但使用路径生成器仅需数行代码，如此开发者就能够专注于任务的逻辑。

```
▼<svg width="500" height="500">
   <path d="M80,40L120,120L160,40L200,120L240,40L280,120" stroke="black" stroke-width="3px"
   fill="none"></path>
 </svg>
```

图 6-3　线段生成器生成的元素

线段生成器是有插值模式的，根据模式的不同，线段的形式（路径）会发生很大的变化。共有 13 种模式，如 linear、linear-closed、step、basis、bundle、cardinal、monotone。使用的时候可自行查询 API 手册。设置方法如下所示：

```
var linePath = d3.svg.line()
```

```
.interpolate("linear-closed");
```

插值模式，指的是对于线段两个端点之间的值如何计算。默认的插值模式是 linear，即线性的。interpolate 的参数，除预定义的 13 种模式之外，还可以是一个自己定义的函数。下面来看看线性插值模式（linear）的实现方法。

```
function interpolateLinear(points) {
      return points.join("L");
}
```

意思是：**对于所有的顶点，在顶点数值之间加入一个 L 作为分隔**。因此，如图 6-3 所示，可以看到结果是“M80,40L120,120L160,40...”，各顶点坐标之间都被加了一个 L。其他所有的模式都是使用类似的手法实现的。图 6-4 展示了不同插值模式下线段的形态，虽然有些已经不是“线段”了，但都是根据线段的顶点生成的。当插值模式为 cardinal、cardinal-open、cardinal-closed 的时候，可以使用 tension()设置张力系数。

如果希望选择性地使用顶点数据，可用 defined()，示例如下：

```
var lines = [80,120,160,200,240,280];

var linePath = d3.svg.line()
                 .x(function(d){ return d; })
                 .y(function(d,i){ return i%2==0 ? 40 : 120; })
                 .defined(function(d){ return d<200; });
```

意思是，对于数组 lines 的每一项，如果其值小于 200，则将其添加进路径，否则不添加。结果如图 6-5 所示，只有 80、120、160 三个数据被绘制了。

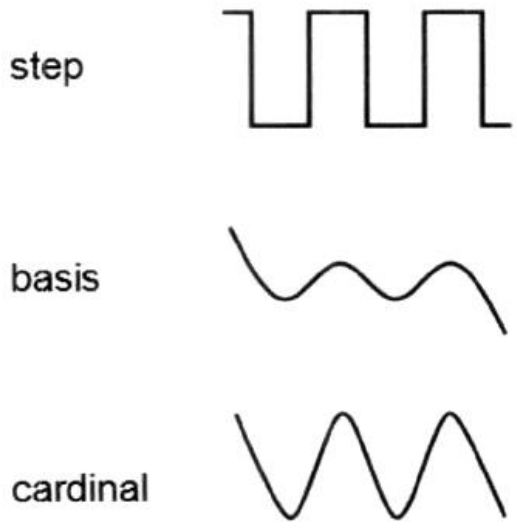


图 6-4　对图 6-2 同样的数据应用不同的插值模式

图 6-5　使用 defined()之后

6.3 区域生成器

区域生成器（Area Generator）用于生成一块区域，使用方法与线段生成器类似。数据访问器有 x()、x0()、x1()、y()、y0()、y1()六个，数量很多，但不需要全部使用。请看下面的例子：

```
var dataset = [80,120,130,70,60,90];

//创建一个区域生成器
var areaPath = d3.svg.area()
                    .x(function(d,i){ return 50 + i * 80; })
                    .y0(function(d,i){ return height/2; })
                    .y1(function(d,i){ return height/2 - d; });

//添加路径
svg.append("path")
   .attr("d",areaPath(dataset))
   .attr("stroke","black")
   .attr("stroke-width","3px")
   .attr("fill","yellow");
```

上述代码定义了一个数组 dataset 和一个区域生成器 areaPath。此区域生成器定制了三个访问器：**x()、y0()、y1()**。将 areaPath 当作函数使用，areaPath(dataset)返回的字符串直接作为<path>元素的 d 的值使用。结果如图 6-6 所示。

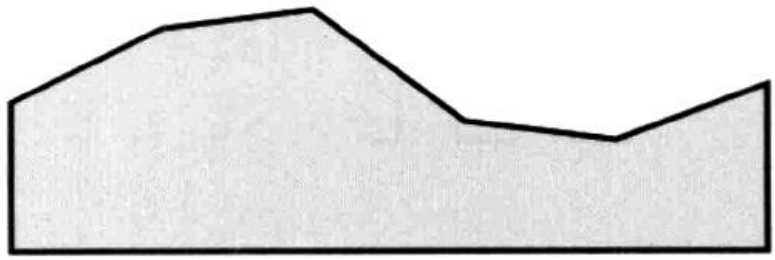


图 6-6　区域生成器

图 6-6 中，上边界的折线是数组 dataset 中各值的反映。根据访问器的设定，可得以下数值（height 的值是 500）。

当 d = 80，i = 0 时，x 等于 50，y0 等于 250，y1 等于 170。

当 d = 120，i = 1 时，x 等于 50 + 1 * 80，y0 等于 250，y1 等于 130。

依此类推。

图 6-7 展示了各访问器的意义，x 是各段的 *x* 坐标，y0 是区域的下限坐标，y1 是区域的上限坐标。如果不定制访问器，默认是：x 为 d[0]，y0 为 0，y1 为 d[1]。此外，还有 **x0()**、**x1()**、**y()**三个访问器，意义类似，如果要制作图 6-6 的横向图需要用到。

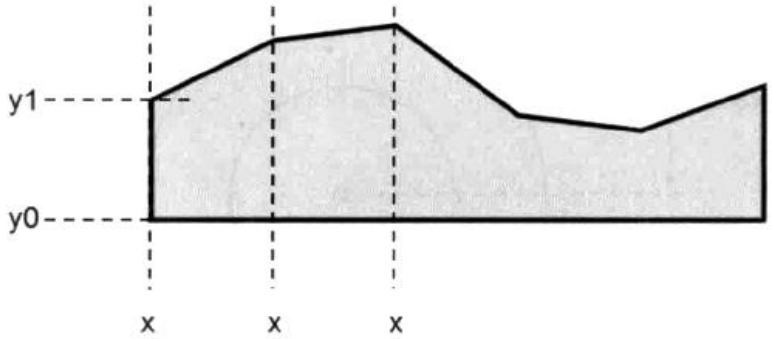


图 6-7　数据访问器 x()、y0()、y1()的意义

与线段生成器类似，区域生成器也有 interpolate()、tension()、defined()这些方法，意义相同。通过设定 interpolate，两点之间的插值会发生相应变化，规则与线段生成器一样。但是，某些模式是区域生成器里没有的，如 linear-closed。因为区域本身就是闭合的，所以就不需要了。图 6-8 展示了 step 和 basis 两种模式的形态。

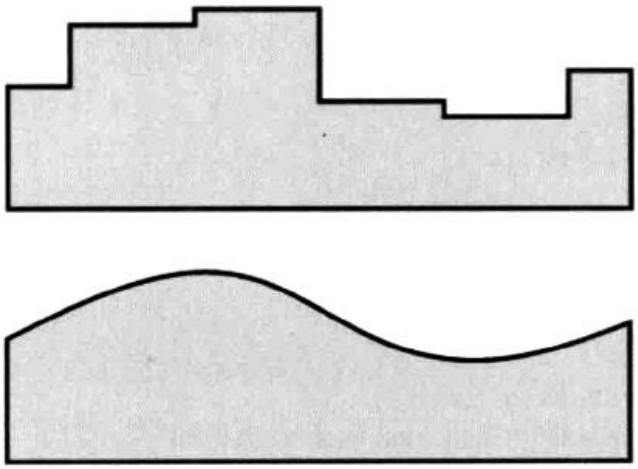


图 6-8　上图为 step 模式，下图为 basis 模式

在需要生成折线或曲线下方的面积的时候，可以考虑使用区域生成器。

6.4　弧生成器

弧生成器（Arc Generator）可凭借起始角度、终止角度、内半径、外半径等，生成弧线的

路径，因此在制作饼状图、弦图等图表时很常用。

有四个访问器需要谨记：内半径访问器 **innerRadius()**、外半径访问器 **outerRadius()**、起始角度访问器 **startAngle()**、终止角度访问器 **endAngle()**。各参数的意义如图 6-9 所示。

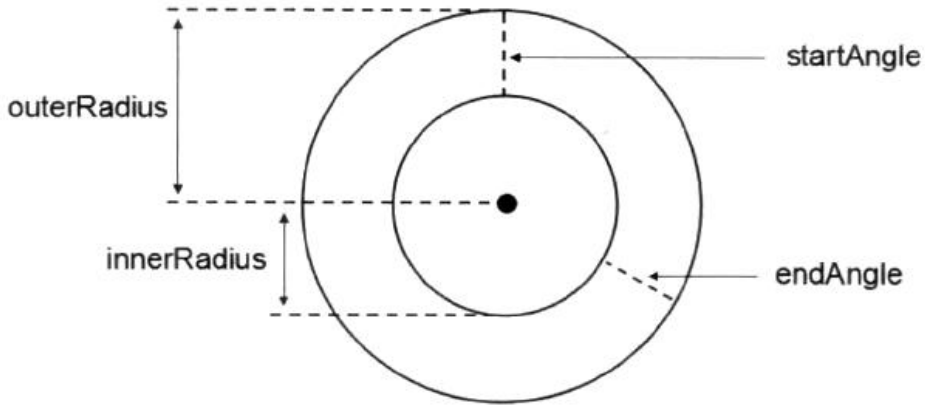


图 6-9　弧生成器中各参数的意义

startAngle 和 endAngle 的单位是弧度，即：0° 要用 0，180° 要用 3.1415926（π）。0° 的位置在“时钟零点”处，终止角度是按照顺时针旋转的。outerRadius 表示外弧半径，innerRadius 表示内弧半径，内弧之内的部分不属于弧的一部分。下面请看一段代码：

```
var dataset = { startAngle: 0 , endAngle: Math.PI * 0.75 };

//创建一个弧生成器
var arcPath = d3.svg.arc()
                .innerRadius(50)
                .outerRadius(100);

//添加路径
svg.append("path")
   .attr("d",arcPath(dataset))
   .attr("transform","translate(250,250)")
   .attr("stroke","black")
   .attr("stroke-width","3px")
   .attr("fill","yellow");
```

dataset 是一个对象，成员变量有两个：startAngle 和 endAngle。然后，创建一个弧生成器 arcPath，设置其内半径和外半径的访问器，分别为常量 50 和 100。没有设置 startAngle 和 endAngle 的访问器，默认使用目标对象中名称为 startAngle 和 endAngle 的变量。最后，添加一个路径元素，将弧生成器计算所得的路径作为属性 d 的值，结果如图 6-10 所示。

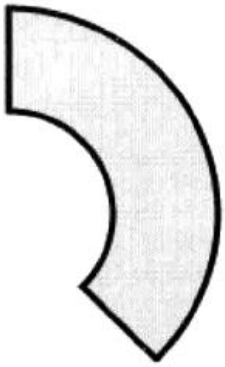


图 6-10　绘制一段弧

看见结果图 6-10，有人可能会联想到这是饼状图的一部分。不错，饼状图正是用弧生成器绘制的。只需在数据中多添加几段弧，令其 startAngle 和 endAngle 成首尾相连的形式，合计 360° 即可。定义饼状图的数据如下：

```
var dataset = [{ startAngle: 0 , endAngle: Math.PI * 0.6 },
               { startAngle: Math.PI * 0.6 , endAngle: Math.PI },
               { startAngle: Math.PI  , endAngle: Math.PI * 1.7 },
               { startAngle: Math.PI * 1.7 , endAngle: Math.PI * 2 }];
```

有四段弧，首尾相连，从 0 到 2π。然后，插入足够数量的路径元素<path>，分别用弧生成器计算路径：

```
//创建一个弧生成器
var arcPath = d3.svg.arc()
                .innerRadius(0)
                .outerRadius(100);

var color = d3.scale.category10();

//添加路径
svg.selectAll("path")
   .data(dataset)
   .enter()
   .append("path")
   .attr("d",function(d){ return arcPath(d); })
   .attr("transform","translate(250,250)")
   .attr("stroke","black")
   .attr("stroke-width","2px")
   .attr("fill",function(d,i){ return color(i); });
```

第 5.2 节提到，color 是一个序数比例尺，按序列返回颜色。此处是为了给饼状图的每段弧上不同的颜色。弧生成器的内半径被设置为 0，这是一个中间没有孔的饼状图，如图 6-11 所示。

还需要给每一段弧添加一个标签文字，先确定文字的位置：**arc.centroid()**可计算弧的中心位置。

```
svg.selectAll("text")
    .data(dataset)
    .enter()
    .append("text")
    .attr("transform",function(d){
        return "translate(250,250)" +
            "translate(" + arcPath.centroid(d) + ")";  //弧的中心位置
    })
    .attr("text-anchor","middle")
    .attr("fill","white")
    .attr("font-size","18px")
    .text(function(d){
    return Math.floor((d.endAngle - d.startAngle)*180/Math.PI)
            + "° ";
});
```

注意粗体字的部分，centroid()的参数是弧对象，返回值是一个二维坐标，其位置是相对于圆心而言的。添加文字之后，结果如图 6-12 所示，添加的文字是每段弧对应的角度。

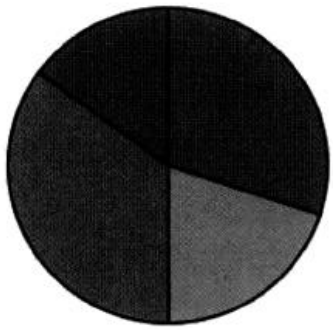


图 6-11　四段弧组成饼状图

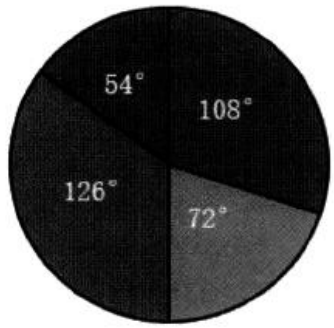


图 6-12　添加文字

如果知道 startAngle 和 endAngle，可以按以上方法作图。但是，一般不会知道，只会知道更原始的数据，如 10、15、20。要将其转换成 startAngle 和 endAngle，才能使用弧生成器来绘制。D3 提供了用于进行这种数据转换的方法，称为布局。布局的内容将在第 10 章进行详细介绍。

6.5　符号生成器

符号生成器（Symbol Generator）能够生成三角形、十字架、菱形、圆形等符号，相关方法

介绍如下。

- d3.svg.**symbol**()

创建一个符号生成器。

- **symbol**(datum[, index])

返回指定数据 datum 的路径字符串。

- symbol.**type**([type])

设定或获取符号的类型。

- symbol.**size**([size])

设定或获取符号的大小，单位是像素的平方。例如设定为 100，则是一个宽度为 10，高度也为 10 的符号。默认是 64。

- d3.svg.**symbolTypes**

支持的符号类型。

type()和 **size()**是访问器，其参数可以是函数，也可以是常数。d3.svg.symbolTypes 是一个数组，里面存有各种符号的字符串，请看下面的代码：

```
console.log(d3.svg.symbolTypes);
```

输出结果为：

```
["circle", "cross", "diamond", "square", "triangle-down", "triangle-up"]
```

符号生成器的类型共有六种：圆形（circle）、十字架（cross）、菱形（diamond）、正方形（square）、下三角形（triangle-down）、上三角形（triangle-up）。下面定义一个数组，数组的每一项是一个对象，对象中有 size 和 type 成员。

```
//数组长度
var n = 30;

//数组
var dataset = [];

//给数组添加元素
for(var i=0;i<n;i++){
    dataset.push( {
        //符号的大小
        size: Math.random() * 30 + 200,
        //符号的类型
        type: d3.svg.symbolTypes[ Math.floor(
              Math.random() * d3.svg.symbolTypes.length )]
    });
}
```

符号的大小和类型都使用随机数来生成。数组 dataset 的每一项是一个对象，其中的变量包

括：size 表示大小，type 表示类型。因此，定制符号生成器时，size 和 type 访问器也要使用此名称，代码如下：

```
//创建一个符号生成器
var symbol = d3.svg.symbol()
                .size(function(d){ return d.size; })
                .type(function(d){ return d.type; });
```

size()的函数里，返回 d.size，表示对于传入的对象，以其名称为 size 的变量作为符号的大小。接下来，添加足够数量的路径元素，代码如下：

```
var color = d3.scale.category20b();

//添加路径
svg.selectAll()
   .data(dataset)
   .enter()
   .append("path")
   .attr("d",function(d){ return symbol(d); })
   .attr("transform",function(d,i){
       var x = 100 + i%5 * 20;
       var y = 100 + Math.floor(i/5) * 20;
       return "translate(" + x + "," + y + ")";
   })
   .attr("fill",function(d,i){ return color(i); });
```

symbol(d)的返回值是一个字符串，构成一个符号。如图 6-13 所示，共 30 个符号，每行显示 5 个，符号的位置是通过设定属性 transform 确定的。

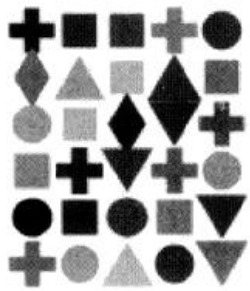


图 6-13　符号生成器

6.6　弦生成器

弦生成器（Chord Generator）根据两段弧来绘制弦，共有五个访问器，分别为 **source()**、

target()、**radius()**、**startAngle()**、**endAngle()**，默认都返回与函数名称相同的变量。如果都使用默认的访问器，则要绘制一段弧，其数据组成应该形如：

```
{
   source:{
      startAngle: 0.2 ,
      endAngle: Math.PI * 0.3 ,
      radius: 100
   },
   target:{
      startAngle: Math.PI * 1.0 ,
      endAngle: Math.PI * 1.6 ,
      radius:100
   }
}
```

其中，source 为起始弧，target 为终止弧，而 startAngle、endAngle、radius 则分别是弧的起始角度、终止角度和半径；也可以更改访问器，使变量具有别的名称，或是使用常量。例如：

```
var chord = d3.svg.chord()
              .source(function(d){ return d.startArc; })
              .target(function(d){ return d.endArc; })
              .radius(200)
              .startAngle(function(d){ return d.start; })
              .endAngle(function(d){ return d.end; });
```

这段代码中，弦生成器的起始弧被设定为 startArc，终止弧为 endArc，半径为常量 200，起始角度为 start、终止角度为 end。因此，数据格式需要修改为：

```
{
   startArc:{
       start: 0.2 ,
       end: Math.PI * 0.3 ,
       radius: 100
   },
   endArc:{
       start: Math.PI * 1.0 ,
       end: Math.PI * 1.6 ,
       radius:100
   }
}
```

如果应用上面定义的弦生成器，由于半径的访问器被设置为常量，因此数据中不必再有半径。

弦生成器的五个访问器所代表参数的意义如图 6-14 所示，弦生成器由两段弧连接而成，右上角的 source 弧带有三个参数，左下角的 target 带有三个参数。

下面绘制一段弦，使用本节开头的数据。先定义一个弦生成器，访问器全部使用默认的，然后在 SVG 中添加路径，再以数据作为生成器的参数返回路径字符串。代码如下所示：

```
//创建一个弦生成器
var chord = d3.svg.chord();

//添加路径
svg.append("path")
    .attr("d",chord(dataset)  )
    .attr("transform","translate(200,200)")
    .attr("fill","yellow")
    .attr("stroke","black")
    .attr("stroke-width",3);
```

结果如图 6-15 所示。弦生成器可用于制作弦图，详细制作方法将在第 10 章进行讲解。

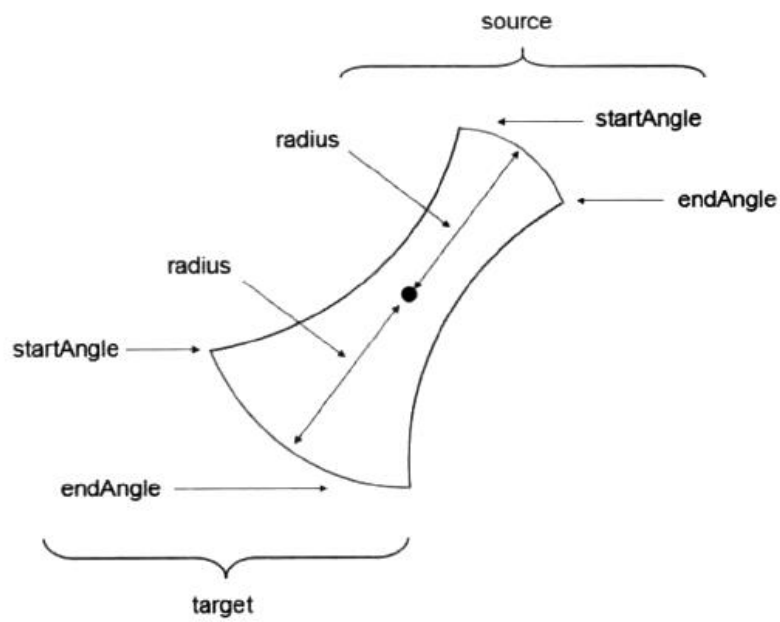


图 6-14　弦生成器各参数的意义

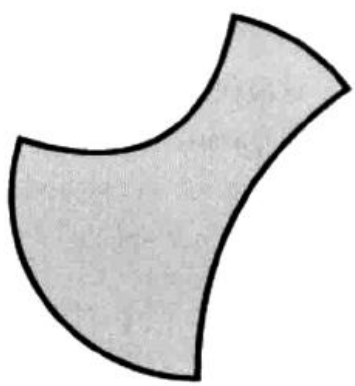


图 6-15　弦生成器

6.7　对角线生成器

弦生成器用于将两段弧连接起来，**对角线生成器**（Diagonal Generator）用于将两个点连接起来，连接线是三次贝赛尔曲线。该生成器使用 **d3.svg.diagonal()**创建，有两个访问器，**source()** 和 **target()**，还有一个投影函数 **projection()**，用于将坐标进行投影。现有如下数据：

```
var dataset = {    source:{ x: 100 , y: 100 },
                   target:{ x: 300 , y: 200 }};
```

source 是起点，target 是终点，其中包含的是 x 坐标和 y 坐标。下面要将这两个点用三次贝塞尔曲线连接起来。先定义一个对角线生成器，访问器都使用默认的；然后添加<path>元素，再使用生成器得到所需要的对角线路径。代码如下所示：

```
//创建一个对角线生成器
var diagonal = d3.svg.diagonal();

//添加路径
svg.append("path")
       .attr("d",diagonal(dataset)  )
       .attr("fill","none")
       .attr("stroke","black")
       .attr("stroke-width",3);
```

结果如图 6-16 所示，左上角的是 source 起点，右下角的是 target 终点，中间的曲线有两个弯；在第 2.5.2 节中介绍过，三次贝塞尔曲线是有两个弯的，与结果符合。

图 6-16　对角线生成器

使用 projection()可以定制具有投影的生成器。投影用于将坐标进行变换，定义了之后，起点和终点坐标都会首先调用该投影进行坐标转换，然后再生成路径。例如，有以下生成器：

```
var diagonal = d3.svg.diagonal()
                 .projection(function(d){
                     var x = d.x * 1.5;
                     var y = d.y * 1.5;
                     return [x,y];
                 });
```

这样，对于每个起点和终点坐标，x 坐标和 y 坐标都会放大 1.5 倍，起点坐标变为(150, 150)，终点坐标变为(450, 300)。但是，原数据并不会更改，只是在绘制的时候使用投影后的坐标。

6.8　折线图的制作

柱形图、散点图、折线图是最简单的三种图表。第 4 章讲述了没有坐标轴的柱形图的制作，

第 5 章为其添加了坐标轴。第 5 章还制作了散点图。本节使用路径生成器绘制一个**折线图**（Line Chart）。

改革开放以来，国民生产总值 GDP 成为中国经济的重要指标，将历年的 GDP 用折线图表现出来是常用的手段。以中国和日本的 GDP 为例，有如下数据：

```
var dataset = [
    {
        country: "china",
        gdp: [[2000,11920],[2001,13170],[2002,14550],
              [2003,16500],[2004,19440],[2005,22870],
              [2006,27930],[2007,35040],[2008,45470],
              [2009,51050],[2010,59490],[2011,73140],
              [2012,83860],[2013,103550]]
    },
    {
        country: "japan",
        gdp: [[2000,47310],[2001,41590],[2002,39800],
              [2003,43020],[2004,46550],[2005,45710],
              [2006,43560],[2007,43560],[2008,48490],
              [2009,50350],[2010,54950],[2011,59050],
              [2012,59370],[2013,48980]]
        }
];
```

dataset 是一个数组，每一项是一个对象，对象里有两个成员，国家名称 country 和国民生产总值 gdp。gdp 也是一个数组，其中[2000, 11920]表示 2000 年的 GDP 为 11920 亿美元。首先，定义 x 轴和 y 轴的比例尺，x 轴表示年份，y 轴表示 GDP 值。定义比例尺之前，要明确绘制区域和 GDP 的最大值：

```
//外边框
var padding = { top: 50 , right: 50, bottom: 50, left: 50 };

//计算GDP的最大值
    var gdpmax = 0;
    for(var i = 0; i < dataset.length; i++){
        var currGdp = d3.max( dataset[i].gdp , function(d) {
                                            return d[1];
                                });
    if( currGdp > gdpmax )
        gdpmax = currGdp;
}
```

padding 是到 SVG 画板上下左右各边界的距离，单位为像素。gdp 的最大值保存在 gdpmax

变量中。使用 d3.max()可以很方便地求数组中的最大值。接下来，凭借 padding 和 gdpmax 定义比例尺的定义域和值域：

```
var xScale = d3.scale.linear()
           .domain([2000,2013])
           .range([ 0 , width - padding.left - padding.right ]);

var yScale = d3.scale.linear()
           .domain([0,gdpmax * 1.1])
           .range([ height - padding.top - padding.bottom , 0 ]);
```

x 轴的定义域是 2000～2013，此处为了代码简洁手动指定了，实际应用时应从数据中获取。y 轴的定义域是 0～gdpmax * 1.1，乘以 1.1 是为了使得图形不在坐标轴的边界绘制。接下来，根据数据定义一个线段生成器：

```
//创建一个直线生成器
var linePath = d3.svg.line()
                 .x(function(d){ return xScale(d[0]); })
                 .y(function(d){ return yScale(d[1]); });
```

该直线生成器的访问器 x 为 xScale(d[0])，y 为 yScale(d[1])。接下来要传入的数据是 gdp 数组，如 d 为[2000, 11920]这样的值；那么 d[0]就是年份，d[1]是国民生产总值。对这两个值都使用比例尺变换，则输入的数据会自动按照比例尺伸缩后再生成直线路径。

定义两个 RGB 颜色，分别用于两条折线的着色。然后，添加与数组 dataset 长度相同数量的<path>元素，并设置为线段生成器计算的路径。代码如下所示：

```
//定义两个颜色
var colors = [ d3.rgb(0,0,255) , d3.rgb(0,255,0) ];

//添加路径
svg.selectAll("path")  //选择<svg>中所有的<path>
   .data(dataset)      //绑定数据
   .enter()            //选择enter部分
   .append("path")     `//添加足够数量的<path>元素
   .attr("transform","translate(" + padding.left + "," +
                                    padding.top  +")")
   .attr("d", function(d){
       return linePath(d.gdp);     //返回线段生成器得到的路径
   })
   .attr("fill","none")
   .attr("stroke-width",3)
   .attr("stroke",function(d,i){
```

```
        return colors[i];
    });
```

添加元素的形式“selectAll().data().enter().append()”，相信读者已经很熟悉了。给属性transform赋予适当的值，令折线平移到指定位置。在<path>元素的d属性中，使用线段生成器计算路径，注意linePath()的参数是d.gdp。此处的线段生成器是按照数组gdp的格式来设定访问器的，因此一定要以d.gdp，而不是以d作为参数。

下面再为折线图添加坐标轴：

```
//x轴
var xAxis = d3.svg.axis()
                .scale(xScale)
                .ticks(5)
                .tickFormat(d3.format("d"))
                .orient("bottom");

//y轴
var yAxis = d3.svg.axis()
                .scale(yScale)
                .orient("left");

//添加一个<g>元素用于放x轴
svg.append("g")
    .attr("class","axis")
    .attr("transform","translate(" + padding.left + "," +
                (height - padding.bottom) +  ")")
    .call(xAxis);

//添加一个<g>元素用于放y轴
svg.append("g")
    .attr("class","axis")
    .attr("transform","translate(" + padding.left + "," + padding.top +  ")")
    .call(yAxis);
```

由于*x*轴的刻度是年份的意思，而数据里的数据类型却是整数类型；所以如果直接在坐标轴上显示，2000年会显示成2,000，2002年会显示成2,002，多一个逗号。因此，粗体字部分加了一条tickFormat()，其中，d3.format("d")表示刻度的数字都用字符串表示。设定之后，年份之间的逗号就会消失。然后，将两个坐标轴分别放到两个<g>元素里。

结果如图6-17所示，如果想要使一段一段的直线看起来更光滑一些，可以使用直线生成器

的插值函数，第 6.2 节介绍了 13 种插值模式。例如，设置为 basis 模式后，线段变为曲线，如图 6-18 所示。

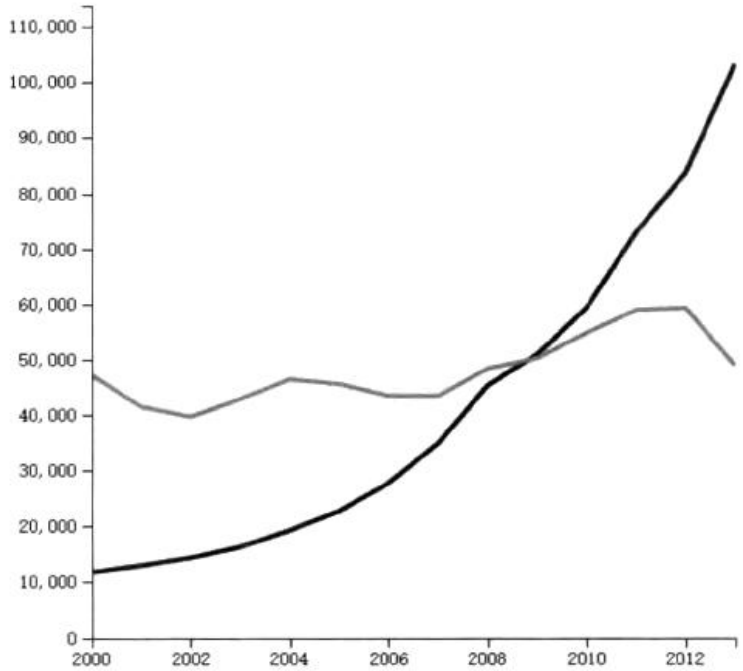


图 6-17　折线图

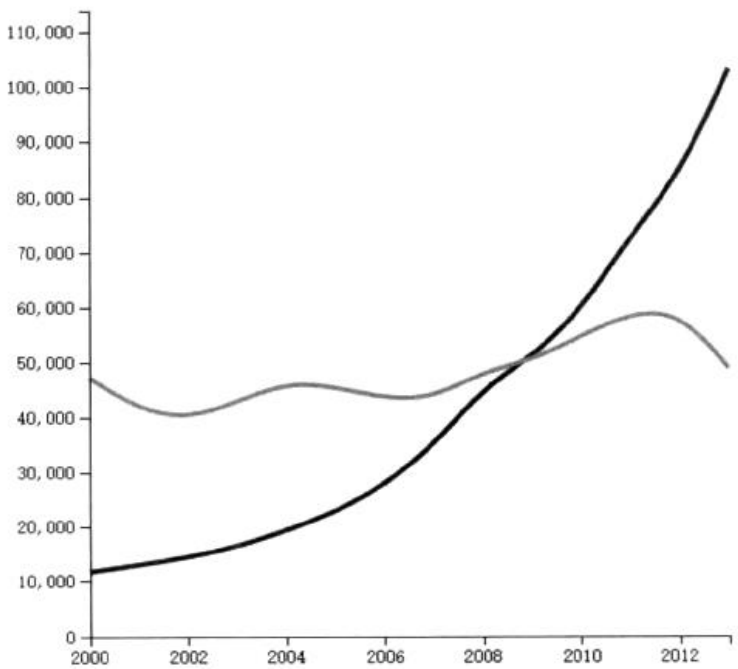


图 6-18　使用插值之后的折线图

上面的折线图还缺少一个标记，用户不知道哪条直线是中国的GDP，哪条是日本的GDP。可添加两个矩形，分别填充为相应的颜色。矩形边上添加表示国家名称的文字。如图6-19所示，一个完整的折线图做好了。

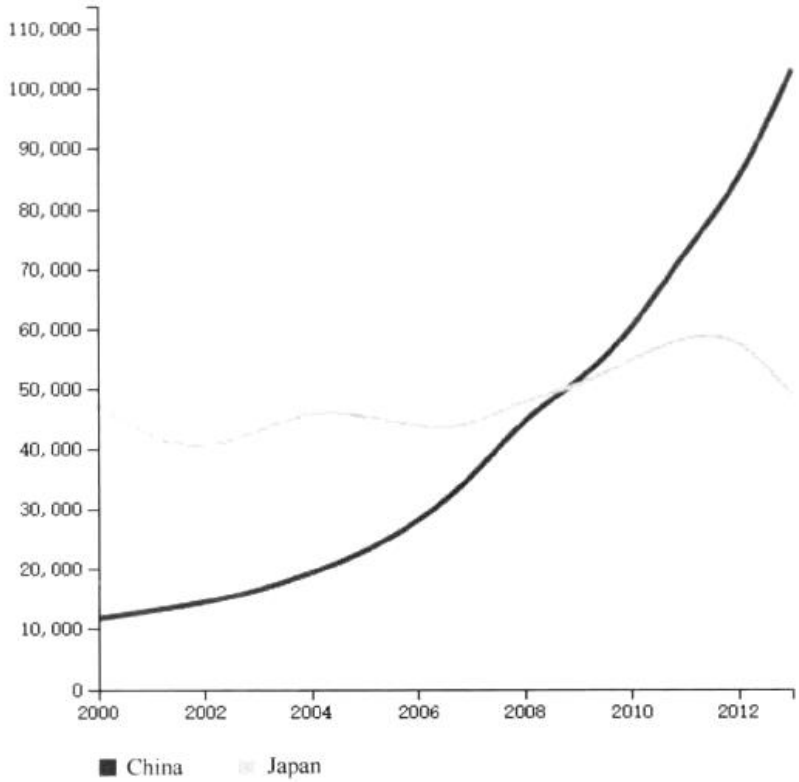


图6-19　添加上标记的折线图

第 7 章

动画

本章内容包括：

- 过渡效果
- 制作动画

过渡和动画都是动态效果，但它们是有区别的：过渡效果的起始状态和目标状态都很明确，指定之后元素会从起始状态缓缓变为目标状态，时间是确定的；动画的起始状态和目标状态不明确，通常不用指定，并且时间通常是不确定的。

第 1 节，讲述关于过渡的知识。

第 2 节，学习如何制作动画。

第 3 节，以散点图为例，讲述在什么场合可以使用过渡。

第 4 节，制作两个动画：时钟和小球运动。

7.1 过渡效果

前三章中绘制的柱形图、散点图、折线图都是直接添加到 SVG 上的，用户直接看到的就是图表的最终结果。第 4.7.2 节中，当单击“排序”按钮后，柱形图的变化也是瞬间发生的。所有的变化都在瞬间完成，不一定友好，有时希望某些变化不是瞬时的，这样可以给用户反应的时间。

例如，当单击“排序”按钮时，所有柱形都缓慢地伸缩到目标位置；当元素的颜色发生变

化时，过渡性地变化到目标颜色。

图形元素从一种状态变化到另一种状态，叫作**过渡**。被 d3.select()或 d3.selectAll()选择的选择集都有一个方法：transition()，能将选择集变为过渡对象。

7.1.1 过渡的启动

启动过渡效果，与以下四个方法相关。

- d3.**transition**([selection],[name])

创建一个过渡对象，参数是选择集。但是，由于每个选择集中都有 transition()方法，可用 d3.select("rect").transition()的方式来创建过渡，因此一般不直接用 d3.transition()。

- transition.**delay**([delay])

设定延迟的时间。过渡会经过一定时间后才开始发生，单位是毫秒。

- transition.**duration**([duration])

设定过渡的持续时间（不包括延迟的时间），单位为毫秒。如.duration(2000)，是持续 2000ms，即 2s。

- transition.**ease**(value[, arguments])

设定过渡样式，例如线性过渡、在目标处弹跳几次等方式。

下面制作一个过渡效果：

在 SVG 区域中添加一个矩形，矩形的初始宽度为 100，经过过渡后，最终为 300。代码如下所示：

```
svg.append("rect")
        .attr("fill","steelblue")
        .attr("x",10)
        .attr("y",10)
        .attr("width",100)
        .attr("height",30)
        .transition()
        .attr("width",300);
```

上述代码只使用了 transition()。默认情况下，延迟（delay）为 0ms，持续时长（duration）为 250ms。请注意过渡前后的状态：

过渡前，矩形的宽度（width）为 100。

过渡后，矩形的宽度为 300。

过渡的前后状态必须是不同的，才能看到变化。不同状态，可以是形状、颜色、位置、透明度等。如图 7-1 所示，矩形从在 0ms 时刻的宽度渐变到 250ms 时刻的宽度。

图 7-1　0 ms 时的矩形（上）和 250 ms 时的矩形（下）

如果没有调用 transition()，append()返回该元素的**选择集对象**。如果调用了 transition()，返回的就不是**选择集对象**，而是一个**过渡对象**。选择集对象和过渡对象是完全不同的概念，其成员变量和方法有所不同。关于其中区别，请看以下代码：

```
var rect = svg.append("rect")
              .attr("fill","steelblue")
              .attr("x",10)
              .attr("y",10)
              .attr("width",100)
              .attr("height",30);

console.log(rect); //rect是选择集

var rectTran = rect.transition(); //启动过渡效果

console.log(rectTran); //rectTran是一个过渡对象
```

rect 是一个选择集对象，rectTran 是一个过渡对象。控制台的输出结果如图 7-2 所示，可以看到，它们是完全不同的两种对象。

```
▶ [Array[1], select: function, selectAll: function, attr: function, classed: function, style:

▶ [Array[1], namespace: "__transition__", id: 1, call: function, empty: function, node: funct
```

图 7-2　变量 rect 和 rectTran 在控制台的输出结果

如图 7-3 所示，左图为选择集对象的方法列表，可以找到 data()、sort()、append()等前面介绍过的方法。右图是过渡对象的方法列表，没有 data()、datum()等方法。因此，过渡对象是不能绑定数据的。

调用 transition()，得到过渡对象。之后，一般会跟着 delay()、duration()、ease()，用于设置延迟、过渡时间、过渡样式。请看下面的例子：

```
var rect = svg.append("rect")
              .attr("fill","steelblue")
              .attr("x",10)
              .attr("y",10)
              .attr("width",100)
              .attr("height",30);
```

```
var rectTran = rect.transition()
                   .delay(500)          //延迟500ms再开始
                   .duration(1000)      //过渡时长为1000ms
                   .ease("bounce")      //过渡样式
                   .attr("width",300);  //目标属性
```

这段代码的过渡效果总时长为 1500ms（延迟 500ms+过渡时长 1000ms），在目标属性处会弹跳几次，这是由于过渡样式被设置为 bounce。

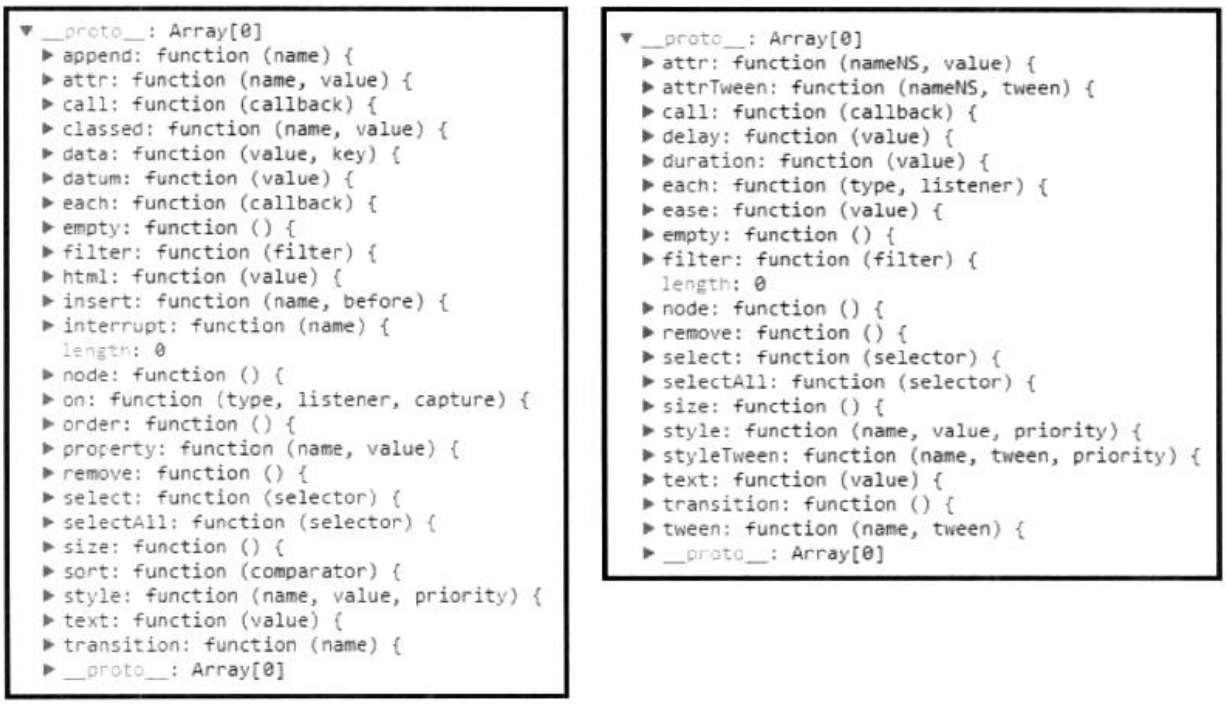


图 7-3　变量 rect（左）和 rectTran（右）的成员函数

另外，transition()可多次调用。每一次都会产生一个新的过渡，可以连续使用。请看下面的代码。

```
var rectTran = rect.transition()          //开始一个过渡
                   .attr("width",300)     //目标宽度为300
                   .transition()          //开始一个过渡
                   .attr("height",300)    //目标高度为300
                   .transition()          //开始一个过渡
                   .attr("width",100)     //目标宽度为100
                   .transition()          //开始一个过渡
                   .attr("height",100);   //目标高度为100
```

这段代码中，第一个过渡是将矩形的宽度转变到 300，第二个是将高度转变到 300，第三个是将宽度转变到 100，第四个是将高度转变到 100。由于没有设定延迟和过渡时间，都使用默认值。

7.1.2　过渡的属性

过渡的属性，是指元素的状态。由于过渡的前后状态不同，因此需要指定过渡前后元素的不同属性。

- transition.**attr**(name, value)

将属性 name 过渡到目标值 value。value 可以是一个函数。

- transition.**attrTween**(name, tween)

将属性 name 使用插值函数 tween()进行过渡。

attr()的使用很好理解，例如：

```
.attr("width",100)
.transition()
.attr("width",300)
```

初始宽度为 100，目标宽度为 300，该过渡会在 250ms（默认时间）内将宽度属性从 100 变为 300，属性变化的中间值是由默认的插值函数计算的。如果要手动设置插值函数，要使用 attrTween()，请看下面的例子：

```
var rect = svg.append("rect")
                .attr("fill","steelblue")
                .attr("x",10)
                .attr("y",10)
                .attr("width",100)
                .attr("height",30);

var rectTran = rect.transition()
                .duration(2000)
                .attrTween("width",function(d, i, a) {
                        return function(t){
                            return Number(a) + t * 300;
                        }
                    }
                );
```

attrTween()的第一个参数是属性名，第二个参数是一个无名函数 function(d, i, a)，d 是被绑定数据，i 是索引号，a 是属性 width 的初始值。该函数返回的 function(t)，就是插值函数。其参数 t 的范围是[0, 1]，0 表示变化的起始，1 表示变化的结束。此处，初始值 a 等于 100，当 t 等于 0 时，function(t)返回 100；当 t 等于 1 时，function(t)返回 400。因此，该过渡将 width 属性从

100 过渡到 400，所用时间为 2000ms（2s）。

- transition.**style**(name, value[, priority])

将 CSS 样式的 name 属性过渡到目标值 value。priority 是可选参数，表示 CSS 样式的优先级，只有 null 和 important 两个值。

- transition.**styleTween**(name, tween[, priority])

将 CSS 样式的属性 name 使用函数 tween 进行过渡。与 attrTween()类似。

假设在 SVG 中有如下元素：

```
<rect x="10" y="10"
width="100" height="30"
style="fill: rgb(70, 130, 180);">
</rect>
```

attr()和 attrTween()操作的是 x、y、width、height 这样的属性。style()和 styleTween()操作的是 style 里的样式，例如：

```
.style("fill","steelblue")
.transition()
.style("fill","red")
```

这段代码会将 style 里的 fill 的值从 rgb(70, 130, 180)变成 rgb(255, 0, 0)，即从 steelblue 变为 red，过渡的时间为 250ms。

- transition.**text**(value)

过渡开始时，将文本设置为 value 值。

- transition.**tween**(name, factory)

将属性 name 按照函数 factory 进行过渡。attrTween()和 styleTween()都是用此函数实现的。

对文字进行过渡要用到 tween()。前面在介绍 attrTween()的时候对矩形的宽度实现了过渡，现添加如下要求：

矩形上添加文字，用来标识矩形的宽度；同时，矩形的宽度在变化的时候，文字也要跟着变化。代码如下所示：

```
var text = svg.append("text")
              .attr("fill","white")
              .attr("x",100)
              .attr("y",10)
              .attr("dy","1.2em")
              .attr("text-anchor","end")
              .text(100);

var initx = text.attr("x");
var initText = text.text();
```

```
var textTran = text.transition()
            .duration(2000)
            .tween("text",function(){
                return function(t){
                    d3.select(this)
                        .attr("x",Number(initx) + t * 300)
                        .text(Math.floor(Number(initText) + t * 300));
                }
            });
```

tween()的第二个参数返回的是 function(t)，t 的范围也是[0, 1]。当 t 为 0 时，函数体里的操作是：

```
d3.select(this)
    .attr("x",100 + 0 * 300)
    .text(Math.floor(100 + 0 * 300));
```

当 t 为 1 时，函数体里的操作是：

```
d3.select(this)
    .attr("x",100 + 1 * 300)
    .text(Math.floor(100 + 1 * 300));
```

过渡的结果如图 7-4 所示。随着时间的推移，文字的内容和文字的位置都是变化的。

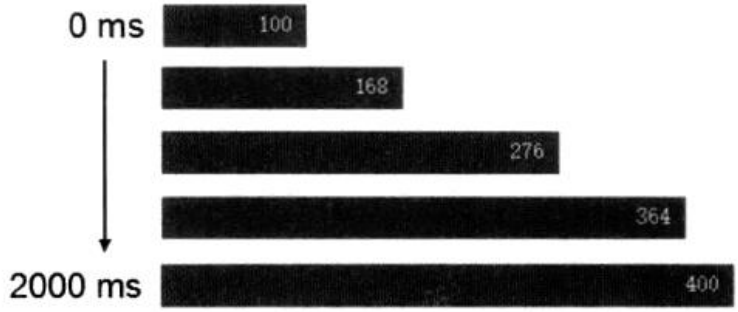


图 7-4　随着时间的推移矩形长度和文字的变化

- transition.**remove**()

过渡结束后，删除被选择元素。

当元素淡出的时候需要用到。例如：

```
rect.transition()
.attr("width",0)
.remove();
```

当矩形过渡的目标完成，即 width 变为 0 的时候，就删除该元素。此外，还可以据此做成：

元素颜色慢慢变淡，最后删除。

7.1.3 子元素

在使用 selection.transition()的时候，该过渡是对于选择集自身的元素来说的，选择集里的子元素不受影响。例如，有 SVG 元素如下：

```
<svg>
    <g>
        <rect></rect>
        <text></text>
    </g>
</svg>
```

如果对<g>元素启动过渡效果。

```
var g = d3.select("g");

g.transition()
    .attr("x",50);
```

结果是<g>元素经过过渡后最终变为<g x="50">。没有任何意义，<g>的子元素<rect>和<text>的 x 属性并没有变。

如果需要选择子元素，参见如下方法。

- transition.**select**(selector)

选择符合选择器的第一个子元素进行过渡。

- transition.**selectAll**(selector)

选择符合选择器的所有元素进行过渡。

- transition.**filter**(selector)

过滤器，与 selection.filter()类似。

选择过渡对象的子元素，与选择集选择子集类似。见下面这个例子：有三个矩形，初始长度都是 100。代码如下：

```
var dataset = [100, 100, 100];

var g = svg.append("g");

var rect = g.selectAll("rect")
            .data(dataset)
            .enter()
            .append("rect")
            .attr("fill","steelblue")
```

```
            .attr("id",function(d,i){ return "rect" + i; })
            .attr("x",10)
            .attr("y",function(d,i){ return 10 + i * 35; })
            .attr("width",function(d,i){ return d; })
            .attr("height",30);
```

以上代码在<g>中添加了三个<rect>，id 分别为 rect0、rect1、rect2。结果如图 7-5 所示，SVG 中有三个矩形元素。

然后，根据 id 选择第二个矩形元素进行过渡操作。代码为：

```
g.transition()
    .select("#rect1")          //选择g中id为rect1的元素
    .attr("width",300);
```

过渡效果的结果如图 7-6 所示。此外，先选择子元素，再调用过渡函数，也是可以的。例如：

```
g.select("#rect1")
    .transition()
    .attr("width",300);
```

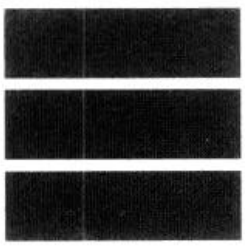


图 7-5　三个相同长度的矩形

图 7-6　选择第二个子元素进行过渡

如果要选择<g>中所有的矩形元素，代码可写成：

```
g.transition()
    .selectAll("rect")
    .attr("width",300);
```

同样，也可以先选择再过渡，如 g.selectAll("rect").transition()。结果如图 7-7 所示。

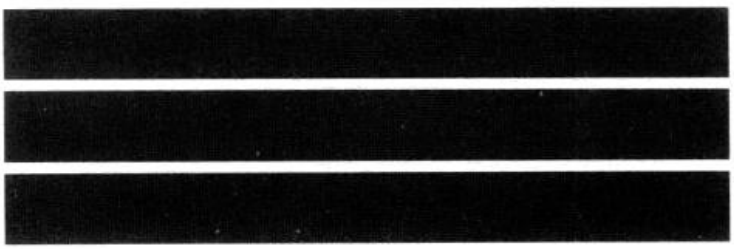


图 7-7　选择所有子元素

第 4 章中介绍过选择集的过滤器，过渡的过滤器与其类似。例如，要将 g 中序号小于 1 的元素过滤掉，代码为：

```
g.transition()
    .selectAll("rect")
    .filter(function(d,i){ return i >= 1; })
    .attr("width",300);
```

选择集中的每一个元素都将调用过滤器函数，当返回值为 true 时，执行过滤操作，否则不执行。结果如图 7-8 所示，由于三个矩形的序号分别为 0、1、2，因此，只有后两个矩形元素满足过滤器的要求。

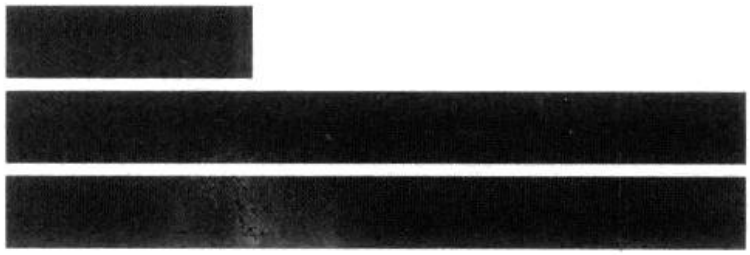


图 7-8　过滤掉序号小于 1 的元素

7.1.4　each()和 call()

选择集里有 each()，过渡里也有 each()，但稍有不同，过渡对象里的 each()支持对事件的响应。

- transition.**each**([type,]listener)

type 表示事件的类型，有 start（开始）、end（结束）、interrupt（打断）三个值。当对应的事件发生时，调用监听器 listener，监听器是一个函数。type 也可以省略，如果省略了，那么和选择集的 each()几乎一样。

```
g.transition()
    .duration(2000)
    .selectAll("rect")
    .each("start",function(d,i){
        console.log("start");
    })
    .each("end",function(d,i){
        console.log("end");
    })
    .attr("width",300);
```

过渡开始时，在控制台输出 start；过渡结束时，输出 end。开始和结束的事件很好理解，但是 **interrupt**（打断）事件在什么情况下才会发生呢？

当某过渡在进行中，该元素又在别处被调用一个新的过渡，这时候就会发生打断事件。

请看下面的例子：

```
g.transition()
    .duration(2000)
    .selectAll("rect")
    .each("interrupt",function(d,i){
        console.log("interrupt");
    })
    .attr("width",300);

setTimeout(function(){
    g.transition()
        .selectAll("rect")
        .attr("width",10);
},1000);
```

这段代码定义了一个过渡，时间为 2s，包含打断事件。此外又调用了 setTimeout()，在经过指定时间后执行指定的函数，此处是在 1s 后执行 function()。function()中调用了相同元素的过渡，而且调用的时候之前的过渡还没结束，因此发生了打断事件。事件发生后，在控制台里输出 interrput。

如果当过渡还在进行的时候，用户可能会执行操作。此时，就要考虑设定打断事件。

- transition.**call**(function[, arguments...])

以过渡对象本身为参数调用 function。对坐标轴进行过渡操作时需要用到，请看下面的例子：

```
var xScale = d3.scale.linear()
                .domain([0,10])
                .range([0,300]);

var xAxis = d3.svg.axis()
                .scale(xScale)
                .orient("bottom");

var g = svg.append("g")
           .attr("class","axis")
           .attr("transform","translate(50,200)")
           .call(xAxis);

//坐标轴的定义域发生了变化
```

```
xScale.domain([0,50]);

//定义一个过渡，事件为2000ms，令坐标轴的变化缓缓发生
g.transition()
    .duration(2000)
    .call(xAxis);
```

如果坐标轴的定义域或值域需要发生变化，就可以采用过渡方法。以上代码中，坐标轴的定义域由[0, 10]变成[0, 50]。要更新坐标轴，再调用一次 call()即可，call()前面要加上 transition()。

7.1.5 过渡样式

过渡效果的原理是：给定一个插值函数 function(t)，t 的范围为[0, 1]；在一段时间内，不断地取[0, 1]内的值，来调用 function(t)，因此得到了某属性变化过程中的值。根据如何取 t 在[0, 1]内的值，可以有不同的**过渡样式**。D3 预定义了一些样式，介绍如下。

- linear

线性地变化，随时间的增长以稳定速度增加。

- cubic

默认的方式，逐渐加快速度。

- elastic

像弹簧似地接近终点。

- back

先往回缩一点，再冲到终点。

- bounce

在终点处弹跳几次，笔者最爱的方式。

上述方式可以加上一个减号（-）与 in、out 联合使用，有四种方式。

- in

按正方向运动。

- out

按相反方向运动。

- in-out

前半段按 in 方式运动，后半段按 out 方式运动。

- out-in

前半段按 out 方式运动，后半段按 in 方式运动。

例如，可以使用 linear-in、elastic-out-in、bounce-out、back-in-out 等组合，默认的组合是 cubic-in-out。

7.2 定时器

动画是由一张张连续的图片组成的，每张图片称为一帧。如果 1s 之内播放了 60 帧，那么此动画的帧率为 60 FPS，FPS 意为每秒显示的帧数（Frames Per Second）。制作动画，就是要制作出一帧一帧连续的图进行显示。常用的有三种方法。

- setInterval(code, millisec)

以指定的周期来执行代码，直到 clearInterval()被调用或窗口被关闭。

- setTimeout(code, millisec)

经过指定的时间后执行代码。

- d3.timer(function[, delay[, time]])

相对指定的绝对时间 time 延迟 delay 时长后，调用 function()，内部实现时使用的是 requestAnimationFrame。

这两个函数都能够实现周期性地调用某函数，来实现动画的效果。setInterval 和 setTimeout 是传统方法，当浏览器显示频率大于 setInterval 或 setTimeout 设定的频率的时候，可能会导致动画的过渡绘制而**失帧**。为了解决此问题引入的 requestAnimationFrame，能与浏览器的绘制时间间隔完全一致，节省了资源。但是，绝不是说 requestAnimationFrame 就完全优于另两个，有时候与浏览器显示频率保持一致会使动画显得过慢。至于应该选择哪一个使用，没有一定的标准，根据自己的要求试验几次，选择自己认为最优的即可。幸运的是，无论哪一种方式都非常容易改写成另一种。

7.2.1 setInterval 和 setTimeout

使用 setInterval 制作动画的代码如下：

```
//每隔10ms调用一次draw
setInterval(draw, 10);

function draw(){
    //重绘
    ...
}
```

重绘函数是 draw，使用 setInterval 设定成每隔 10ms 调用一次 draw，则浏览器就会连续不断地重绘，形成动画；另外，也可以使用 setTimeout，将此函数写在 draw 里，以达到反复重绘的目的。代码如下所示：

```
function draw(){
    //重绘
    ...

    //经过指定的延迟时间10ms再调用draw
    setTimeout(draw,10);
}
```

setInterval 和 setTimeout 是浏览器对象 window 的成员：window.setInterval 和 window.setTimeout。两个函数都会返回一个 id 值，将此 id 值传给 clearInterval 和 clearTimeout 可用于解除此定时器。例如：

```
var id = setInterval(draw, 10);

clearInterval(id);
调用clearInterval后，动画暂停。
```

7.2.2 d3.timer

d3.timer 是使用 requestAnimationFrame 实现的。requestAnimationFrame 的用法与 setTimeout 类似，只是不用指定延迟的时间而已：

```
function draw(){
    //重绘
    ...

    requestAnimationFrame(draw);
}
```

requestAnimationFrame 会自动按照浏览器的显示频率来计算。将 requestAnimationFrame 封装后的 d3.timer，可以实现类似 setInterval 似的调用方式：

```
d3.timer(draw);

function draw(){
    //重绘
    ...
}
```

不用像 setInterval 一样指定调用周期，因为 d3.timer 的后两个参数可用于指定延迟一段时间后再执行。其中，第二个参数 delay 是相对于绝对时间的延迟时间；第三个参数是绝对时间，默认值为 Date.now()，即当前时间。例如：

```
d3.timer(draw, 1000);
```

```
d3.timer(draw, 500, +new Date(2015,1,1,15,21,30));
```

第一句是相当于当前实现延后 1000ms 后再不断地调用 draw，第二句是相当于 2015 年 2 月 1 日 15 时 21 分 30 秒后再延迟 500ms。要注意，这里在 new Date 前有一个加号（+），用于将时间转换为毫秒。另外，Date 的第二个参数月份是从 0 开始的，0 表示一月，1 表示二月。

```
var date = new Date(2015,1,1,15,21,30);

console.log(date);      //输出Sun Feb 01 2015 15:21:30
console.log(+date);     //输出1422771690000
console.log(date.getTime());   //输出1422771690000
```

+date 等同于 date.getTime()，输出一个毫秒数，这个毫秒数是从 1970 年 1 月 1 日算起的。

7.3　应用过渡的场合

过渡是使元素缓缓地发生变化，因此在以下三种场合可能会用到过渡。

- 元素刚开始出现的时候。
- 元素被更新的时候（用户进行了交互操作）。
- 元素被删除的时候。

元素刚开始出现时，为了告知用户图形绘制的顺序或侧重点，可能用到；元素被更新时，为了让用户能更清楚地看到哪些数据发生了变化，可能会用到；元素被删除的时候，将元素缓缓淡出，能使用户感到友好。

下面以第 5 章制作的散点图为例，来说明如何将上述三种场合写成 D3 代码。由于本例涉及数据的更新，请读者回想一下第 4.5.3 节的处理模板和第 4.7.2 是如何更新柱形图的。

1. 数据

有以下数据：一组圆心坐标，作为散点。代码如下：

```
var center = [[0.5,0.5],[0.7,0.8],[0.4,0.9],[0.11,0.32],
              [0.88,0.25],[0.75,0.12],[0.5,0.1],[0.2,0.3],
              [0.4,0.1],[0.6,0.7]];
```

2. SVG 区域

添加一个 SVG 区域，并定义外边框。代码如下：

```
var width  = 500;  //SVG绘制区域的宽度
var height = 500;  //SVG绘制区域的高度

var svg = d3.select("body")              //选择<body>
```

```
                .append("svg")              //在<body>中添加<svg>
                .attr("width", width)       //设定<svg>的宽度属性
                .attr("height", height);//设定<svg>的高度属性

//外边框
var padding = { top: 30 , right: 30, bottom: 30, left: 30 };
```

3. 比例尺

定义比例尺，在绘制散点和坐标时用到。代码如下：

```
//x轴宽度
var xAxisWidth = 300;

//y轴宽度
var yAxisWidth = 300;

//x轴比例尺
var xScale = d3.scale.linear()
                .domain([0, 1])
                .range([0,xAxisWidth]);

//y轴比例尺
var yScale = d3.scale.linear()
                .domain([0, 1])
                .range([0,yAxisWidth]);
```

4. 绘制散点（应用过渡）

将绘制散点的代码都写在 drawCircle()函数里，其中包含 update、enter、exit 三部分的处理办法，分别对应更新、添加、删除三种处理数据的方式。当数据发生这三种变化时，每一种处理办法都是采用了过渡的；即图形的变化不是一蹴而就，而是缓缓变化的。注意代码中粗体字的部分使用了 transition()，过渡时间都设定为 500ms。本例中，更新、添加、删除三个部分的处理方法分别如下。

- 添加新点时，从坐标原点处过渡到目标位置，如图 7-9 所示。

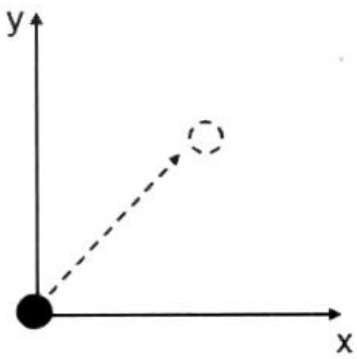


图 7-9　添加新点时的过渡操作

- 更新点时，坐标系中的散点过渡到新的位置，如图 7-10 所示。
- 点被删除时，点慢慢变成白色，最后删除，如图 7-11 所示。

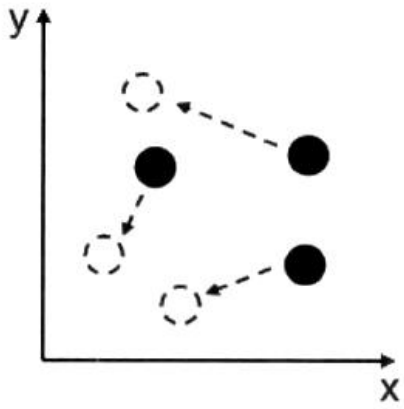


图 7-10　更新的过渡操作

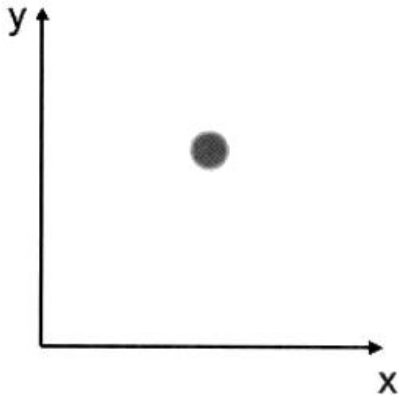


图 7-11　删除点时的过渡操作（淡出）

将上述三种情况，都封装在 drawCircle()里，代码如下：

```
function drawCircle(){

//绑定数据，获取update部分
var circleUpdate = svg.selectAll("circle")
                      .data(center);        //绑定数据

//获取enter部分
var circleEnter = circleUpdate.enter();

//获取exit部分
var circleExit = circleUpdate.exit();

//1. update部分的处理办法
//使用过渡的方式，缓缓移动到新坐标位置
circleUpdate.transition()                   //更新数据时启动过渡
            .duration(500)
            .attr("cx", function(d){        //新的x坐标
                return padding.left + xScale(d[0]);
            })
            .attr("cy", function(d){        //新的y坐标
                return height- padding.bottom - yScale(d[1]);
            });

//2. enter部分的处理办法
//插入圆到坐标原点，然后再过渡到目标点
circleEnter.append("circle")                //添加元素
```

```
            .attr("fill","black")
            .attr("cx", padding.left)                    //过渡前的x坐标
            .attr("cy", height- padding.bottom)          //过渡前的y坐标
            .attr("r", 7 )
            .transition()                    //启动添加元素时的过渡
            .duration(500)                   //设定过渡时间
            .attr("cx", function(d){         //过渡后的x坐标
                return padding.left + xScale(d[0]);
            })
            .attr("cy", function(d){         //过渡后的y坐标
                console.log(d[1] + " " + yScale(d[1]));
                return height- padding.bottom - yScale(d[1]);
            });

//3. exit部分的处理办法
//慢慢变成白色，最后删除
circleExit.transition()                  //删除数据时启动过渡
            .duration(500)               //时间为500ms
            .attr("fill","white")        //设定过渡目标
            .remove();
}
```

5. 绘制坐标轴

坐标轴的绘制方法与前几章一样，只是要注意的是，绘制 y 轴时更改过的比例尺要在绘制完成之后再变回去。代码如下：

```
function drawAxis(){

//x轴的生成器
var xAxis = d3.svg.axis()
                .scale(xScale)
                .orient("bottom")
                .ticks(5);

//重定义y轴比例尺的值域
yScale.range([yAxisWidth,0]);

//y轴的生成器
var yAxis = d3.svg.axis()
                .scale(yScale)
                .orient("left")
                .ticks(5);

//绘制x轴
```

```
svg.append("g")
        .attr("class","axis")
        .attr("transform","translate(" + padding.left + "," +
                                    (height - padding.bottom) +  ")")
        .call(xAxis);

//绘制y轴
svg.append("g")
        .attr("class","axis")
        .attr("transform","translate(" + padding.left + "," +
                        (height - padding.bottom - yAxisWidth) +  ")")
        .call(yAxis);

//绘制完坐标轴后将值域变回去
yScale.range([0,yAxisWidth]);

}
```

6. 更新数据

添加三个函数，分别在更新、添加、删除时调用：

```
//更新
function update(){
    for(var i=0;i<center.length;i++){   //对于每一个点
        center[i][0] = Math.random();   //更新x坐标
        center[i][1] = Math.random();   //更新y坐标
    }
    drawCircle();   //重绘
}

//添加
function add(){
    center.push( [ Math.random() , Math.random() ] );  //添加新点
    drawCircle();
}

//删除
function sub(){
    center.pop();   //删除center数组中的最后一个点
    drawCircle();   //重绘
}
```

然后，在<body>中添加三个按钮，单击后分别调用上述三个函数：

```
<body>
```

```
<button type="button" onclick="update()"> 更新 </button>
<button type="button" onclick="add()"> 增加 </button>
<button type="button" onclick="sub()"> 减少 </button>
</body>
```

7. 结果

结果如图 7-12 所示，初始图为左上图，图中的散点最初都是由坐标原点经过 500ms 过渡而来的。单击“更新”按钮后，所有散点经过 500ms 过渡到新位置，此为右上图。单击数次“增加”按钮后，成为左下图，新增加的顶点都是从左下角飞出来的。单击数次“减少”按钮后，最终称为右下图，删除顶点时顶点慢慢变白（淡出），最后消失。

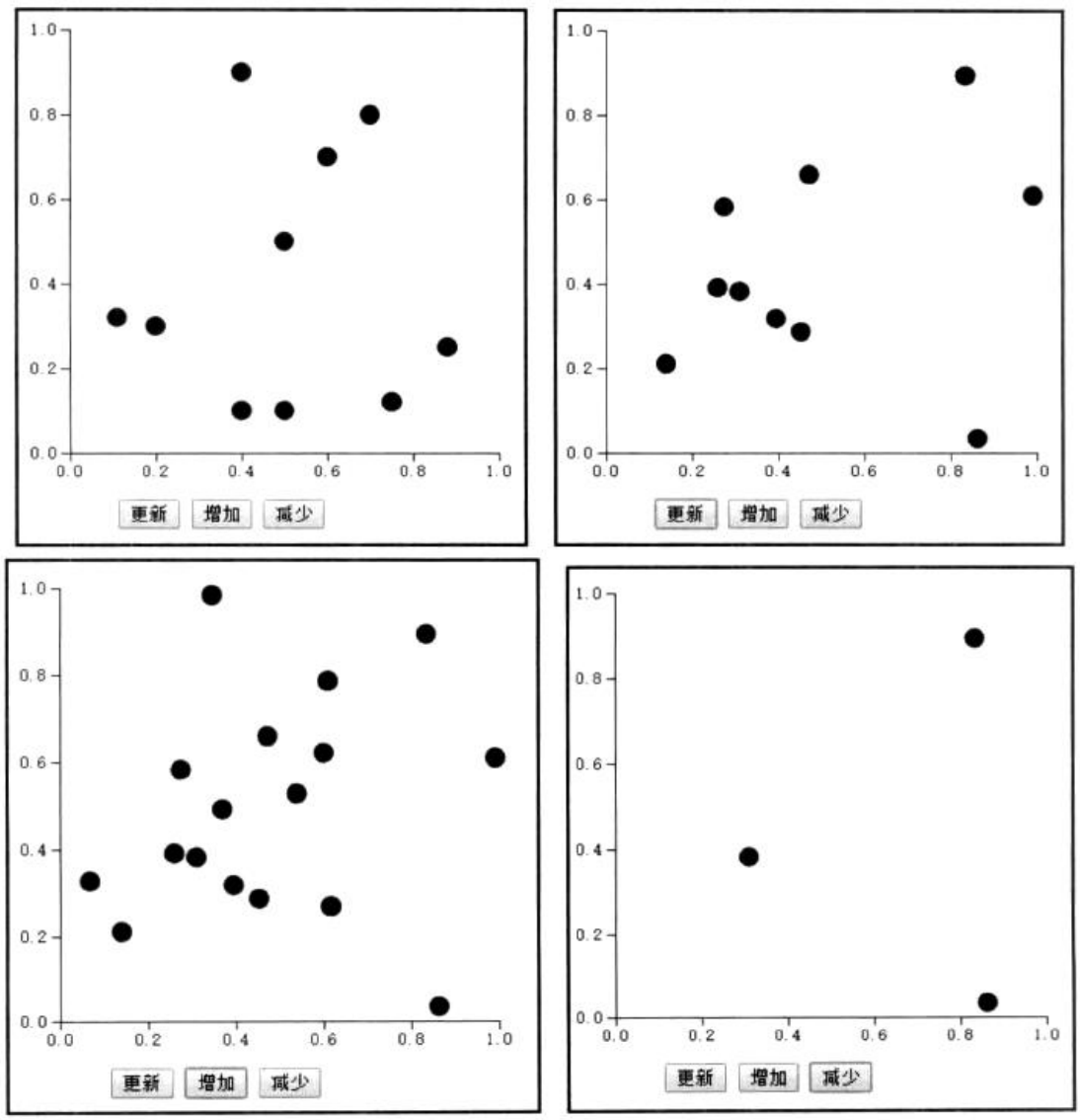


图 7-12　初始图（左上），单击“更新”按钮后（右上），单击数次“增加”按钮后（左下），单击数次“减少”按钮后（右下）

本节以散点图为例，讲述了当更新、添加、删除操作发生时，如何应用过渡。由于图片不容易展示过渡效果变化的过程，细致的感受请运行源代码。

7.4 简单的动画制作

本节制作两个动画：时钟和小球运动。时钟使用 setInterval 制作，小球运动使用 d3.timer 制作。

7.4.1 时钟

时钟本身就是用来测量时间的，自然会随着时间的变化而变化。获取本地的当前时间可调用 Date()。下面写一个函数：getTimeString，获取当前时间的时分秒，并以 21:12:30 的形式返回。代码如下：

```
function getTimeString(){
    var time = new Date(); //获取本地当前时间

    var hours = time.getHours();                    //获取小时
    var minutes = time.getMinutes();                //获取分钟
    var seconds = time.getSeconds();                //获取秒数

    //如果时分秒为1、2、3这样的数，则将其变为01、02、03
    hours = hours < 10 ? "0" + hours : hours;
    minutes = minutes < 10 ? "0" + minutes : minutes;
    seconds = seconds < 10 ? "0" + seconds : seconds;

    //返回最终字符串
    return hours + ":" + minutes + ":" + seconds;
}
```

接下来，在SVG区域中添加一段文本，使用append()添加<text>元素。位置设置为(100,100)，再将文本的class属性设置为time：

```
var timeText = svg.append("text")
                  .attr("x",100)
                  .attr("y",100)
                  .attr("class","time")
                  .text(getTimeString());
```

在类time里主要设定字体、颜色、粗细等。代码如下：

```
.time {
    font-family: Cursive;
    font-size: 40px;
    stroke: black;
    stroke-width: 2;
}
```

接下来使用 setInterval 制作动画，每 1000ms 调用一次绘制函数，更改 timeText 的内容即可，代码如下：

```
//每1000ms调用一次updateTime()
setInterval(updateTime,1000);

function updateTime(){
    //更新timeText的内容
    timeText.text(getTimeString());
}
```

结果如图 7-13 所示，该时钟每 1s 更新一次。

图 7-13　时钟

7.4.2　小球运动

物理现象常需用动画来描述，例如小球的运动，随着时间的推移，小球的速度和位置会发生变化。下面制作一个小球的动画：

在 2D 空间中，小球仅受到重力的作用。假设碰到墙壁后没有能量损失。

定义一个函数 updateBall，用于更新小球的速度和位置，并重新绘制小球。然后，调用 d3.timer()，以 updateBall 作为参数。代码如下：

```
var t0 = Date.now();    //前一时刻
var t1 = Date.now();    //当前时刻
var dt = t1 - t0;       //时间间隔

//循环调用updateBall
d3.timer(updateBall);

//更新小球的速度和位置，并重绘
function updateBall(){
```

```
    //计算当前时刻
    t1 = Date.now();

    //计算当前时刻与前一时刻的间隔（乘以0.001变单位为秒）
    dt = (t1 - t0)*0.001;

    //计算下一时刻小球的坐标
    ...

    //更新图形
    ball.attr("cx",function(d){ return d.x; })
        .attr("cy",function(d){ return d.y; });

    //记录当前时刻到t0，为下一次计算使用
    t0 = t1;
}
```

要计算小球的物理运动，需要知道每次重绘之间的间隔是多少。时间间隔可用 Date.now() 计算，以当前时刻减去前一时刻即可得到。结果如图 7-14 所示，展示了小球在某一个时间段的运动轨迹。

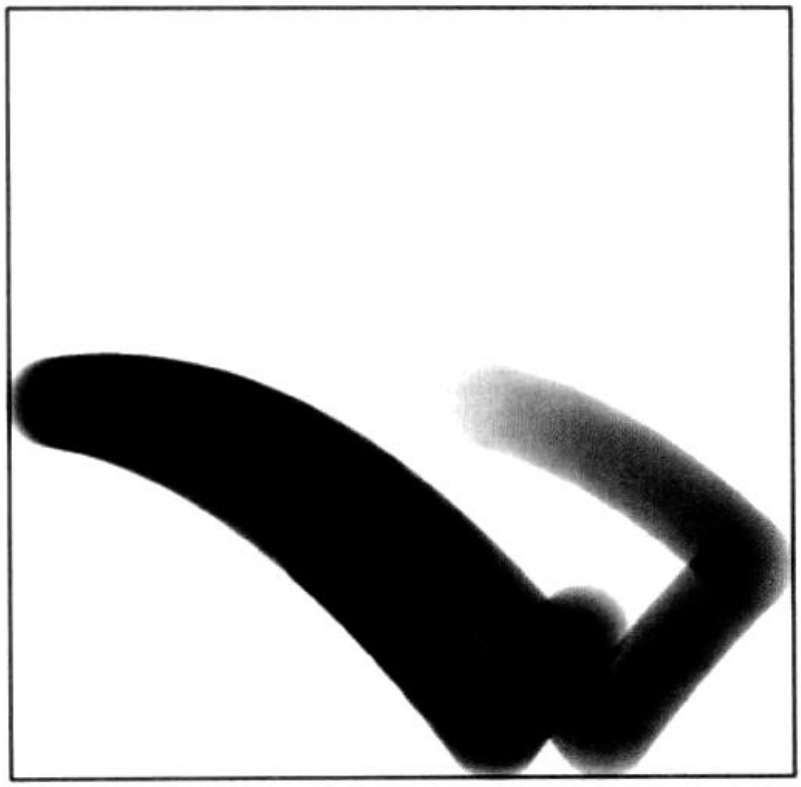


图 7-14 小球的运动轨迹

第8章

交互

本章内容包括：

- 交互式的入门：鼠标、键盘、触屏
- 事件
- 行为：拖曳和缩放

D3 能创建交互式图表，选择集有一个方法 on()，用来设定事件的监听器。当某事件被触发时，此事件的属性保存在 d3.event 里。

第1节，讲述三种交互式手段：鼠标、键盘、触屏。

第2节，介绍事件对象 d3.event。

第3节，学习创建两种行为：拖曳和缩放。

8.1 交互式入门

与图表的交互，指在图形元素上设置一个或多个监听器，当事件发生时，做出相应的反应。如果直接用 DOM API 给网页元素添加监听器，例如给一个段落元素<p>添加点击事件，代码如下：

```
<p id="mypara">Click Here</p>
<script>
    var para = document.getElementById("mypara");
    para.onclick = function(){
```

```
        this.innerHTML = "Thank you";
    }
</script>
```

click 是事件名称，function()是监听器函数。当点击<p>时，调用 function()将<p>的内容更改为“Thank you”。以上是 DOM API 的写法，在 D3 中更简单，请看以下代码：

```
d3.select("#mypara")
    .on("click",function(){
        d3.select(this).text("Thank you");
    });
```

select 选择段落元素，再通过 on()添加事件和监听器。结果同样是当鼠标点击段落时，文字更改为“Thank you”。

不同浏览器所能捕捉到的事件不同，有一部分标准事件是各浏览器间通用的，关于事件种类，可查询以下链接：

https://developer.mozilla.org/en-US/docs/Web/Events#Standard_events

所有选择集对象都有 on()方法，设定事件监听器的代码如下：

```
selection.on("click", function(){
    ...
})
```

如果某元素已经有了监听器，新的监听器会覆盖旧的。另外，如果给同一事件添加多个监听器，需修改名称：**事件名后加一个点，然后再输入一个名称**。代码如下所示：

```
.on("click.first",function(){
    console.log("First Click");
})
.on("click.second",function(){
    console.log("Second Click");
});
```

如此设定后，当 click 事件发生时，两个监听器都会被调用。如果需要删除某元素上的监听器，代码如下：

```
.on("click",null);
```

第一个参数为事件名，第二个参数为 null，如此可删除监听器。但是，将 click 设置为 null，click.first 和 click.second 并不会被删除，必须要指定全称。

有一个问题很容易出错：过渡对象是没有 on()的，不能对过渡对象设置监听器。因此，下面的代码是不正确的。

```
svg.select("circle")
```

```
    .transition()
    .on("click",function(){      //错误，过渡对象没有on()
    ...
});
```

浏览器会报错：**使用了没有定义的函数**。由于使用了 transition()，返回的就是过渡对象。因此，如果要设定监听器，需要在调用 transition()之前。

```
svg.select("circle")
    .on("click",function(){ //正确，选择集对象里有on()
        ...
    })
    .transition();
```

8.1.1 鼠标

鼠标事件是用于响应鼠标操作的，如单击、双击、移动等，在浏览器上绝大多数交互式操作是使用鼠标的。常用的事件介绍如下。

- click

鼠标单击某元素时，相当于 mousedown 和 mouseup 组合在一起。

- mouseover

光标放在某元素上。

- mouseout

光标从某元素上移出来时。

- mousemove

鼠标被移动的时候。

- mousedown

鼠标按钮被按下。

- mouseup

鼠标按钮被松开。

- dblclick

鼠标双击。

下面举一个应用 mouseover 和 mouseout 的例子：为第 5.4 节的柱形图添加事件。当光标移入时，柱形变为黄色；当光标移出时，缓缓变回原来的铁蓝色。请看下面的代码：

```
var rect = svg.selectAll("rect")
           .data(dataset)        //绑定数据
           .enter()              //获取enter部分
           .append("rect") //添加rect元素，使其与绑定数组的长度一致
```

```
.attr("fill","steelblue")          //设置颜色为steelblue
.attr("x", function(d,i){          //设置矩形的x坐标
    return padding.left + xScale(i);
})
.attr("y", function(d){      //设置矩形的y坐标
    return height- padding.bottom - yScale(d);
})
.attr("width",xScale.rangeBand())   //设置矩形的宽度
.attr("height",function(d){         //设置矩形的高度
    return yScale(d);
})
.on("mouseover",function(d,i){
    //当光标移到元素上时，将此元素变为黄色
    d3.select(this)
        .attr("fill","yellow");
})
.on("mouseout",function(d,i){
    //当光标移出元素时，将此元素变回原来的铁蓝色
    d3.select(this)
        .transition()          //开启一个过渡
        .duration(500)         //过渡的时间为500ms
        .attr("fill","steelblue");
});
```

d3.select(this)，表示选择当前的元素，this 就是事件被触发的元素。要改变响应事件的元素时，常会用到 d3.select(this)。此外，为使鼠标移出元素时产生颜色渐变效果，调用 transition() 创建了一个过渡，过渡的目标是矩形原来的颜色，如图 8-1 所示。

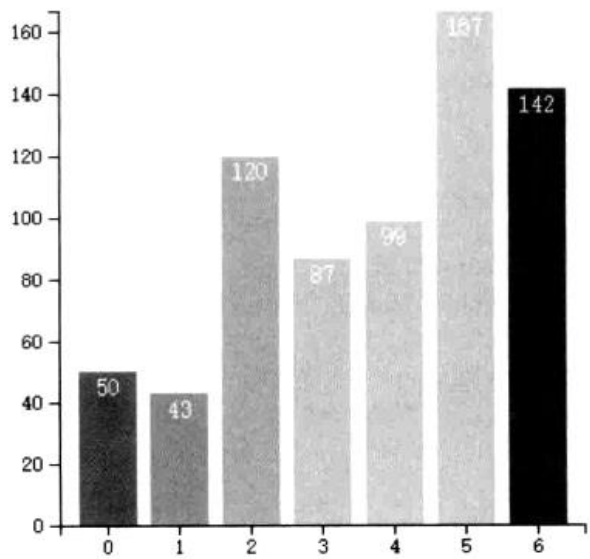


图 8-1 鼠标移入和移出

8.1.2 键盘

键盘事件有以下三种。

- keydown

当用户按下任意键时触发，按住不放会重复触发此事件。该事件不会区分字母的大小写，例如“A”和“a”被视为一致。

- keypress

当用户按下字符键（大小写字母、数字、加号、等号、回车等）时触发，按住不放会重复触发此事件。该事件区分字母的大小写。

- keyup

当用户释放键时触发，不区分字母的大小写。

键盘的事件相对较少，但是要注意区分 keydown 和 keypress 的不同。keydown 用于任意键的事件，keypress 用于字符键。如果只需要处理字母数字类的响应，或是要对大小写字母分别处理的时候，使用 keypress；如果要处理上下左右（↑ ↓ ←→）、Shift、Ctrl 等特殊键的输入，使用 keydown。

下面看一个例子，响应键盘上的 A、S、D、F 四个键，并分别将这几个键的矩形和文字在 SVG 上绘制出来。

```
//四个字母
var characters = ["A","S","D","F"];

//绘制四个矩形
var rects = svg.selectAll("rect")
               .data(characters)
               .enter()
               .append("rect")
               .attr("x",function(d,i){ return 10 + i * 60; })
               .attr("y",150)
               .attr("width",55)
               .attr("height",55)
               .attr("rx",5)
               .attr("ry",5)
               .attr("fill","black");

//绘制四个文字
var texts = svg.selectAll("text")
               .data(characters)
               .enter()
```

```
            .append("text")
            .attr("x",function(d,i){ return 10 + i * 60; })
            .attr("y",150)
            .attr("dx",10)
            .attr("dy",25)
            .attr("fill","white")
            .attr("font-size",24)
            .text(function(d){ return d; });
```

绘制结果如图 8-2 所示。

图 8-2 键盘上的四个键

接下来要添加监听事件，键盘的监听事件添加到<body>中，使用 d3.select("body")选择该元素后，再使用 on()来添加监听器：

```
d3.select("body")
    .on("keydown",function(){

      //keydown事件的监听器
      rects.attr("fill",function(d){

          if( d == String.fromCharCode(d3.event.keyCode) ){
            //如果所按下的键与此rect元素上绑定的数据d相同，则返回黄色
              return "yellow";
            }else{
             //否则返回黑色
              return "black";
            }

          });
      })
      .on("keyup",function(){

      //keyup事件的监听器
      rects.attr("fill","black");

  });
```

被按下的键会保存在 d3.event.keyCode 里，以 ASCII 码保存。使用 String.fromCharCode()

可以将 ASCII 码转换为字符。结果如图 8-3 所示，当 S 键被按下时，调用 keydown 事件的监听器，S 所属的矩形变成了黄色。当 S 键松开时，调用 keyup 事件的监听器，S 所属的矩形又会变成黑色。

图 8-3　S 键被按下

使用键盘时，通常是按下一个键就立即放开（敲击一个键），此时三种事件都会发生，发生的顺序是 keydown、keypress、keyup。

8.1.3　触屏

触摸屏近些年发展迅速，已不是什么新颖的技术，用户很可能使用 iPad 或 iPhone 等设备浏览网页。因此，需要考虑响应触屏事件的监听器，常用的触屏事件有以下三种。

- touchstart

当触摸点被放在触摸屏上时。

- touchmove

当触摸点在触摸屏上移动时。

- touchend

当触摸点从触摸屏上拿开时。

举一个例子：画一个矩形作为参照物，矩形不动，再画一个圆形，矩形和圆形开始都是蓝色。另外，在圆形元素上添加触屏的三个事件，要达到的效果是：

（1）当用户触摸到圆上时，圆变为黄色。

（2）用户可通过触摸拖动圆形。

（3）用户不再触摸后，圆变回蓝色。

代码如下所示：

```
var circle = svg.append("circle")
            .attr("cx",150)
            .attr("cy",200)
            .attr("r",50)
            .attr("fill","blue")
            .on("touchstart",function(){        //touchstart事件的监听器
                d3.select(this).attr("fill","yellow");
            })
```

```
.on("touchmove",function(){      //touchmove事件的监听器
    var pos = d3.touches(this)[0];      //获取第一个触摸点
    d3.select(this)
        .attr("cx",pos[0])              //触摸点的x坐标
        .attr("cy",pos[1]);             //触摸点的y坐标
})
.on("touchend",function(){       //touchend事件的监听器
    d3.select(this).attr("fill","blue");
});
```

当开始触摸（touchstart）事件发生时，通过改变填充属性将圆变为黄色。当触摸移动（touchmove）事件发生时，先使用 d3.touches(this)[0]获取第一个触摸点，再更改圆心坐标来达到拖动的目的。当触摸结束（touchend）事件发生时，再将圆的填充色改为蓝色。

使用 d3.touches()可以从被触摸的对象里获取触摸点的坐标。由于可能不止有一个触摸点，因此其返回值为一个数组：[[x0, y0], [x1, y1], ...]。其中，[x0, y0]是一个触摸点，[x1, y1]是第二个触摸点，依此类推。d3.touches(this)[0]是获取当前被触摸元素的第一个触摸点。

结果如图 8-4 所示，其中圆形用于捕捉触屏事件，矩形是作为参照物存在的。

当触摸到圆时，圆的颜色变为黄色，如图 8-5 所示。

图 8-4　初始状态

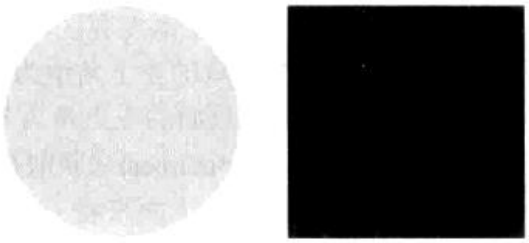


图 8-5　触摸到圆

将圆拖动到别的位置，如图 8-6 所示。

如图 8-7 所示，离开触摸屏之后，圆变回蓝色。

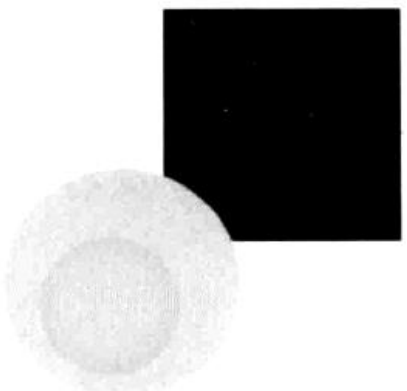


图 8-6　通过触摸移动圆形

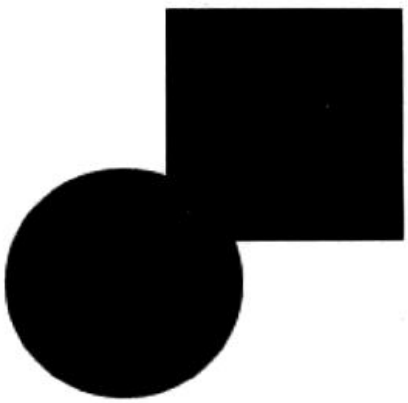


图 8-7　离开触摸屏

8.2 事件

上一节使用了 d3.event.keyCode，其中 keyCode 是对象 d3.event 的成员变量。任意事件发生时，其事件的信息都会保存在 d3.event 里。因此，每次事件发生后，d3.event 都会更新一次。根据事件种类的不同，d3.event 保存的对象名称和属性是不同的。图 8-8 展示了当 click 事件、keydown 事件、touchstart 事件发生时，d3.event 的变化：由上至下分别是 MouseEvent、KeyboardEvent、TouchEvent。

```
▶ MouseEvent {dataTransfer: null, toElement: body, fromElement: null, y: 138, x: 5
▶ KeyboardEvent {repeat: false, metaKey: false, altKey: false, shiftKey: false, ct
▶ TouchEvent {metaKey: false, altKey: false, shiftKey: false, ctrlKey: false, chan
```

图 8-8 在控制台输出 d3.event

当事件对象为 MouseEvent 时，其成员变量有 screenX、screenY、clientX、clientY、pageX、pageY。其中，

screenX 和 **screenY** 的参照点是显示器屏幕的左上角。

clientX 和 **clientY** 以浏览器内容区域的左上角为参照点，此参照随着滚动条的拖动而变动（即无论如何拖动，内容区域的左上角恒为原点）。

pageX 和 **pageY** 是以页面的左上角为参照点，参照点不会随着滚动条的移动而移动。

D3 提供了 d3.mouse(container)来获取相对于容器 container 的坐标，使用此函数可轻松获取点击处相对于<svg>的坐标。下面来看一个例子，先在<body>中添加一个 元素，其具有内边框：

```
<div style="padding:50 ; background-color: gray"></div>
```

然后在 里加入<svg>元素，在<svg>元素里再加入一个矩形<rect>元素：

```
var width  = 400;  //SVG绘制区域的宽度
var height = 400;  //SVG绘制区域的高度

var svg = d3.select("div")          //选择<body>
        .append("svg")            //在<body>中添加<svg>
        .style("background-color","yellow")  //设定背景颜色为黄色
        .attr("width", width)   //设定<svg>的宽度属性
        .attr("height", height);//设定<svg>的高度属性

svg.append("rect")
    .attr("x",200)
    .attr("y",100)
    .attr("width",100)
```

```
    .attr("height",100);
```

绘制结果如图 8-9 所示。如果对矩形元素<rect>添加点击事件，那么有一点必须要考虑：当点击事件发生时，获取的鼠标坐标是以 的左上角为原点，还是以<svg>的左上角为原点呢？

很多时候，我们希望其坐标是相对于<svg>来说的。如此可以只关心画板之内，而不用关心画板外的世界。

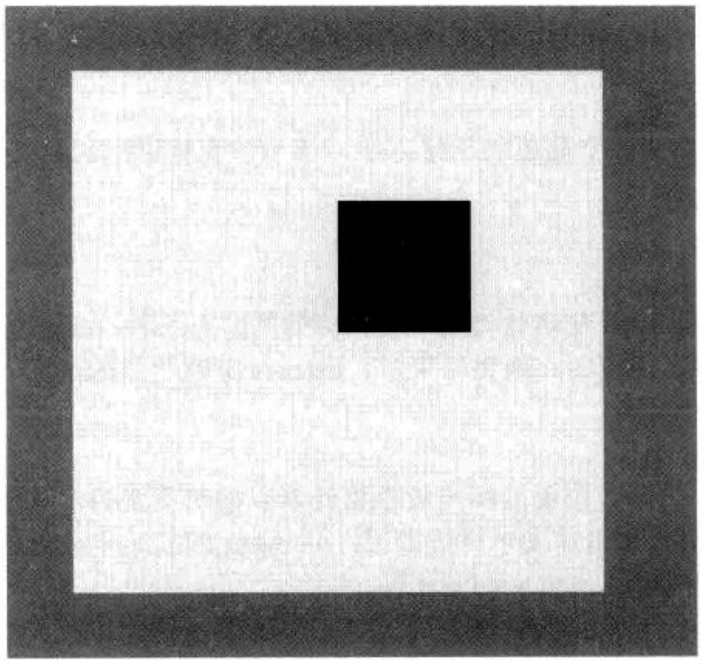


图 8-9　最外层为 div 元素，中间层为 svg，最里层为 rect 元素

给矩形元素添加下面一句代码：

```
.on("click",function(){
    console.log(d3.mouse(this));    //this是当前被点击的元素
})
```

添加之后，如果点击矩形的左上角，将会输出(200, 100)。很明显，这是以<svg>的左上角为参照点的。因为 有 50 个像素的内边距，如果是以 为参照点的，输出结果应当为(250, 150)。

当事件对象为 TouchEvent 时，在第 8.1.3 节时曾用过 d3.touches()来获取坐标值。这与 d3.mouse()是类似的，也能获取相对于画板<svg>的坐标值。

8.3　行为

拖曳（drag）和缩放（zoom）是较高级的交互式操作，D3 中提供了 d3.behavior.drag 和 d3.behavior.zoom 来创建这两种行为。

8.3.1 拖曳

拖曳（drag），是指使用鼠标或触屏将元素从一个位置移到另一个位置。在某图形元素上，鼠标拖曳的操作顺序是：

（1）按下鼠标键。

（2）鼠标移动。

（3）放开鼠标键。

D3 为拖曳行为提供了一个简单的创建方法，该方法支持鼠标和触屏的拖曳。

- d3.behavior.**drag**()

创建一个拖曳行为。

- drag.**on**(type[, listener])

设定事件的监听器。type 是事件类型，被支持的类型有三种：dragstart、drag、dragend。它们分别表示拖曳开始时、拖曳中、拖曳结束时。listener 是监听器的函数，如果省略该参数，则返回当前指定事件的监听器。

- drag.**origin**([origin])

设定拖曳的起点。此函数可**使鼠标与被平移元素以相对不变的偏移量移动**。如果设定了起点，此函数会在 mousedown 事件发生时被调用，d3.event 的初始坐标就是此起点坐标。该函数的返回值是一个对象，对象里包含 x 和 y 属性。

下面做一个例子：绘制两个圆，颜色为黑色，能够通过鼠标和触屏将其拖曳移动。首先，定义两个圆的数据：

```
var circles = [ { cx: 150, cy:200, r:30 },
                { cx: 250, cy:200, r:30 }];
```

其中，cx 和 cy 为圆心坐标，r 为半径。然后，在<svg>中添加足够数量的<circle>元素：

```
svg.selectAll("circle")
        .data(circles)          //绑定数据
        .enter()
        .append("circle")
        .attr("cx",function(d){ return d.cx; })
        .attr("cy",function(d){ return d.cy; })
        .attr("r",function(d){ return d.r; })
        .attr("fill","black")
        .call(drag);            //调用drag函数
```

这段代码的最后一行，调用了 call()，参数为 drag()。drag()是接下来要定义的拖曳行为，使用 d3.behavior.drag 创建，代码如下所示。

```
var drag = d3.behavior.drag()       //创建一个拖曳行为
        .origin(function(d,i){      //设置起点坐标
```

```
        //起点坐标为被拖动物体的圆心坐标
        return {x: d.cx ,y: d.cy }
    })
    .on("dragstart", function(d){   //dragstart的监听器
        console.log("拖曳开始");
    })
    .on("dragend", function(d){     //dragend的监听器
        console.log("拖曳结束");
    })
    .on("drag", function(d){        //drag的监听器
        d3.select(this)     //选择当前被拖曳的元素
            //将d3.event.x赋值给被绑定的数据，再将cx属性设置为该值
            .attr("cx", d.cx = d3.event.x )
            //将d3.event.y赋值给被绑定的数据，再将cy属性设置为该值
            .attr("cy", d.cy = d3.event.y );
    });
```

这段代码中，origin()里返回的是包含 x 和 y 属性的对象，x 的值为 d.cx，y 的值为 d.cy。拖曳的起点坐标是要被写入到 d3.event 里的，那么 d3.event 最初始的值就是被绑定数据里保存的圆心坐标。

在 dragstart 和 dragend 的监听器中分别输出“拖曳开始”和“拖曳结束”两个字符串。在 drag 事件的监听器中，将当前被选择元素<circle>的属性 cx 和 cy 更改为 d3.event.x 和 d3.event.y。

d3.event.x 和 d3.event.y 是当前的拖曳坐标。但是，**拖曳坐标不一定是鼠标所在的坐标，这取决于 origin()所设定的起点位置**。当 origin()设定为 null 的时候，鼠标坐标就是拖曳坐标。

结果如图 8-10 和图 8-11 所示。在拖曳之后，圆的位置发生了变化，而且控制台输出了两行文字，这是分别在拖曳开始和拖曳结束时输出的，证明调用了 dragstart 和 dragend 的监听器。另外，不仅用鼠标可以拖曳圆，用触屏也可以。

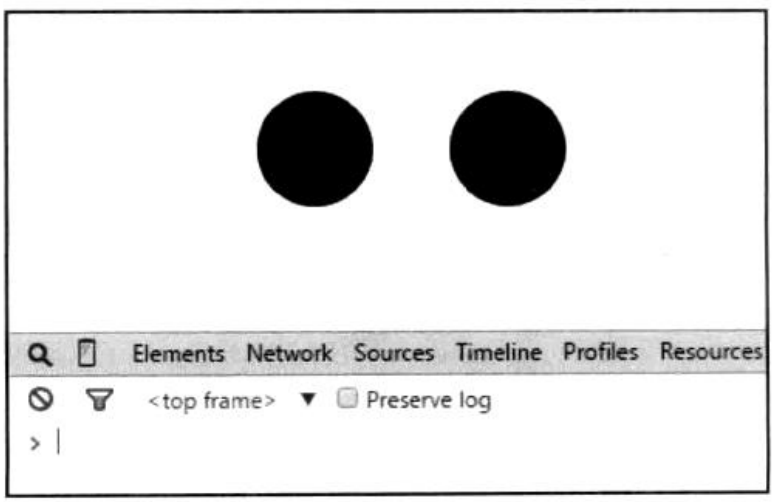


图 8-10　初始状态，被绘制的两个圆

如果希望鼠标坐标和被拖曳元素不在一个位置，而是保持一定的偏移量，可以改变 origin 的值。例如，将上例的代码稍做修改：

```
.origin(function(d,i){
    return {x: 300 ,y:.300 }
})
```

则每次拖曳时，圆都会先跑到(300, 300)的位置，然后与鼠标保持一定的偏移量移动，如图 8-12 所示。

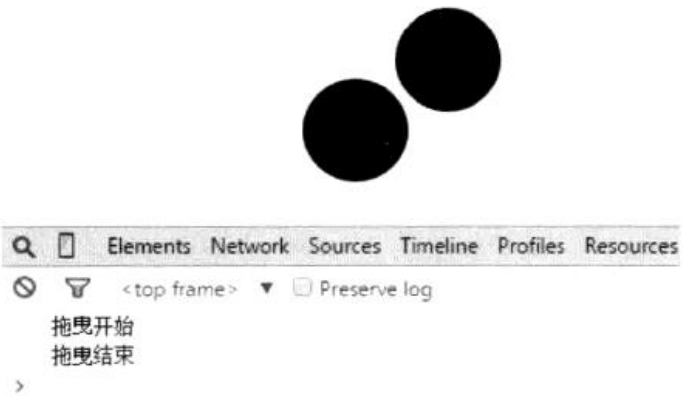


图 8-11　将圆拖曳到其他位置后

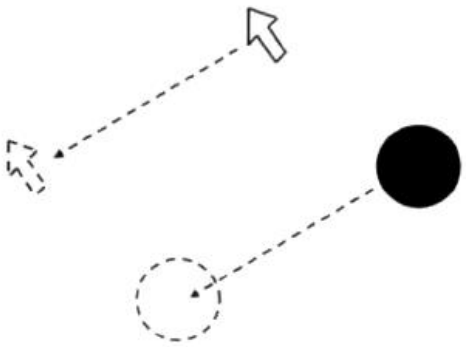


图 8-12　鼠标与元素以一定的偏移量移动

有时候会看到 origin()里的代码是这样的：function(d){ return d; }。这种情况一定是被绑定数据里有 x 和 y 属性存在。

当 drag 事件被触发时，d3.event 对象如图 8-13 所示。其中，x 和 y 表示拖曳坐标，dx 和 dy 表示相对上一次被触发的偏移量。

```
▼Object {type: "drag", x: 176, y: 156, dx: 1, dy: 0…}
    dx: 1
    dy: 0
  ▶ sourceEvent: MouseEvent
  ▶ target: function n(){this.on("mousedown.drag",u).on("touchstart.drag",i)}
    type: "drag"
    x: 176
    y: 156
  ▶ __proto__: Object
```

图 8-13　drag 事件被触发时的 d3.event 对象

8.3.2　缩放

缩放（zoom）在地图中的应用最多，例如在谷歌地图或百度地图里，用户能够将鼠标移动到一点，通过滚轮可以放大或缩小地图。D3 提供了 d3.behavior.zoom()用于构建缩放行为。另外，

在百度或谷歌地图中，用户可通过按住鼠标左键不放来移动地图，这称为**缩放平移（pan）**。d3.behavior.zoom()同时支持鼠标和触屏。

- d3.behavior.**zoom**()

创建一个缩放行为。

- **zoom**(selection)

应用此行为到选择集 selection。

- zoom.**translate**([translate])

设定当前的缩放平移向量，默认为[0, 0]。如果省略，则返回当前的平移向量。

- zoom.**scale**([scale])

设定初始的放大（缩小）量，默认为 1。如果省略，则返回当前的缩放量。

- zoom.**scaleExtent**([extent])

设定放大（缩小）的最小值和最大值，默认为[0, 无穷大]。

- zoom.**center**([center])

设定缩放的中心，默认为鼠标当前位置。

- zoom.**x**([x])

设定一个 x 方向的比例尺，该比例尺会随着放大缩小自动改变其定义域。

- zoom.**y**([y])

设定一个 y 方向的比例尺，该比例尺会随着放大缩小自动改变其定义域。

- zoom.**on**(type, listener)

设置事件类型和监听器。事件类型有三种：zoomstart、zoom、zoomend，分别表示缩放开始时、缩放中、缩放结束时。

下面制作一个例子：绘制四个圆，并添加缩放行为。圆的数据如下所示：

```
var circles = [ { cx: 150, cy:200, r:30 },
                { cx: 220, cy:200, r:30 },
                { cx: 150, cy:270, r:30 },
                { cx: 220, cy:270, r:30 }];
```

cx 和 cy 是圆心坐标，r 是半径。然后，添加一个<g>元素，将所有<circle>都添加在<g>里：

```
var g = svg.append("g")
           .call(zoom);

g.selectAll("circle")
   .data(circles)
   .enter()
```

```
    .append("circle")
    .attr("cx",function(d){ return d.cx; })
    .attr("cy",function(d){ return d.cy; })
    .attr("r",function(d){ return d.r; })
    .attr("fill","black");
```

g 调用了 call(zoom)，zoom 是缩放行为。本例中，是为 g 添加了缩放行为，而不是在某一个圆上添加。因此，当对一个圆进行缩放时，其他圆也一起缩放。拖放行为的定义如下。

```
var zoom = d3.behavior.zoom()  //创建一个缩放行为
        .scaleExtent([1, 10])        //设置缩放的范围
        .on("zoom", function(d){
            d3.select(this).attr("transform",
                "translate(" + d3.event.translate + ")" +  //平移量
                "scale(" + d3.event.scale + ")");  //缩放量
        });
```

scaleExtent()定义了缩放的范围，最小值为 1，表示最小只能缩放到原始尺寸，最大值为 10，表示最大能放大到原来的 10 倍。

zoom 的监听器内容是：当 zoom 事件被触发时，选择当前元素，更改其 transform 属性。transform 属性的值里，translate(x, y)用于平移，scale(s)用于缩放。而且，zoom 的事件对象 d3.event 里正好有变量 translate 和 scale，这正是缩放事件产生的平移和缩放量。

结果如图 8-14 和图 8-15 所示。

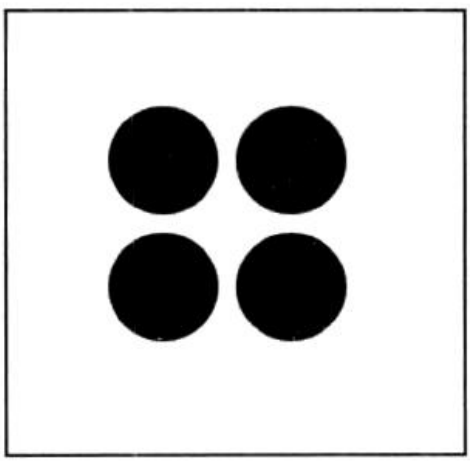


图 8-14　初始图

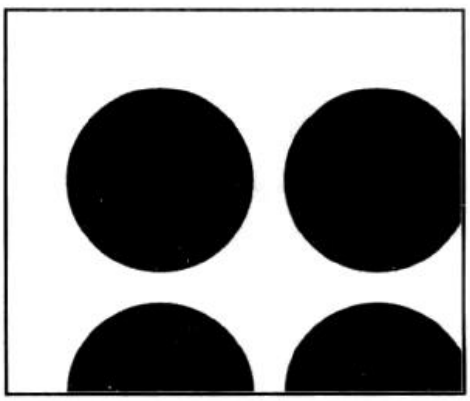


图 8-15　使用滚轮放大后

图 8-16 展示了 d3.event 里保存的值。可以看到，scale 是一个浮点型值，translate 是一个数组，里面有两个值，分别是平移量的 x 坐标和 y 坐标。

```
▼Object {type: "zoom", scale: 3.031433133020798, translate: Array[2], sourceEvent: Wh
    scale: 3.031433133020798
  ▶ sourceEvent: WheelEvent
  ▶ target: function n(n){n.on(z,s).on(Xa+".zoom",h).on("dblclick.zoom",g).on(T,f)}
  ▼ translate: Array[2]
      0: -317.0426465431546
      1: -382.0485067998202
      length: 2
    ▶ __proto__: Array[0]
    type: "zoom"
  ▶ __proto__: Object
```

图 8-16　zoom 事件发生时 d3.event 的值

如果在定义缩放行为的时候，使用 x()和 y()定义了比例尺，则随着缩放的变化，比例尺的定义域也会发生变化。例如，对上述代码添加两个比例尺：

```
var x = d3.scale.linear()
    .domain([0, width])     //width的值为400
    .range([0, width]);

var y = d3.scale.linear()
    .domain([0, height])        //height的值为400
    .range([0, height]);
```

初始定义域分别为[0, width]和[0, height]。然后，将缩放行为的代码做如下修改。

```
var zoom = d3.behavior.zoom()       //创建一个缩放行为
        .x(x)
        .y(y)
        .scaleExtent([1, 10])   //设置缩放的范围
        .on("zoom", function(d){

            console.log("x 的定义域: " + x.domain());
            console.log("x 的值域: " + x.range());
            console.log("y 的定义域: " + y.domain());
            console.log("y 的值域: " + y.range());

            d3.select(this).attr("transform",
                "translate(" + d3.event.translate + ")" +  //平移量
                "scale(" + d3.event.scale + ")");  //缩放量
        });
```

当 zoom 事件被触发时，会输出比例尺的定义域和值域。如图 8-17 所示，当放大到某一程度时，x 比例尺的定义域变为[104, 236]，y 比例尺的定义域变为[127, 259]，值域不变，还是[0, 400]。

```
x 的定义域：104.37596438890509,236.32675546619444
x 的值域：0,400
y 的定义域：127.39499699773893,259.3457880750283
y 的值域：0,400
```

图 8-17　放大后比例尺的定义域和值域

此变化可以理解为：经过放大之后，对于 x 轴来说，原来位于第 104 个像素的图形，现在位于 0 处，原来位于第 236 个像素的图形，现在位于 400 处，参见概念图 8-18。可以想象，原来的第 0 个像素变化到了某一负值，已经不在画板之内。定义域的自动变化，**可用于更新原来应用这个比例尺的图形。**

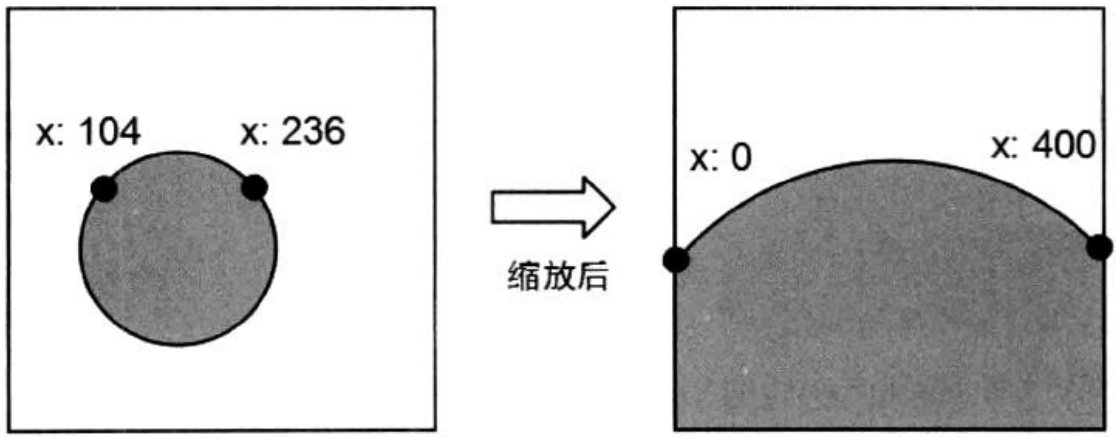


图 8-18　缩放后比例尺的变化

第 9 章

导入和导出

本章内容包括：

- 导入文件
- 导出文件

第 1 节，讲述如何读取 JSON、CSV、XML、TXT 文件。

第 2 节，学习如何将可视化图表输出成 SVG、PDF、PNG 等文件。

9.1 文件导入

数据可视化的数据一般都不写在 HTML 代码里，而是保存在外部文件中。外部文件位于服务器端，因此，必须向服务器端请求数据。

AJAX（Asynchronous JavaScript and XML，异步的 JavaScript 和 XML），是在后台与服务器进行数据交换的比较好的方式。AJAX 可以使网页实现异步更新，这意味着请求文件的同时没有必要更新全部网页。其中，XMLHttpRequest 是 AJAX 的基础，所有现代浏览器均支持 XMLHttpRequest。

D3 中提供了一个方法 d3.xhr()，是基于 XMLHttpRequest 实现的，用起来比 XMLHttpRequest 简单。但是，一般不直接使用 d3.xhr()，D3 在此方法之上又封装了一层，对应于不同类型的文件。例如，

d3.json()：请求 JSON 文件。

d3.csv()、d3.tsv()：请求 CSV、TSV 表格文件。

d3.xml()：请求 XML 文件。

d3.html()：请求 HTML 文件。

d3.text()：请求 TXT 文本文件。

以上方法，是使用 D3 进行编程时经常用到的。以 d3.json 为例，其使用格式如下。

```
d3.json("example.json",function(error,data){
    /data为服务器返回的数据
});
```

其中，example.json 为文件名，function 是一个无名函数，包含两个参数：error 和 data。error 是错误信息，data 就是文件 example.json 中的数据。

但是，需要注意的是，由于这些方法都是**异步**请求的（基于 AJAX），因此，如果有以下情况，

```
var mydata;
d3.json("example.json",function(error,data){
mydata = data;
});
funA(mydata);        //错误，因此d3.json是异步的，此函数调用时，
                     mydata不一定有值
```

请求 example.json 后，将数据赋值给 mydata，然后在 funA()中使用。乍看起来没有问题，但是由于 d3.json 是异步读取的，有可能文件还没有读取完就已经调用 funcA 了，此时 mydata 不一定有值。应该将 funA 放到函数体里，代码如下：

```
d3.json("example.json",function(error,data){
    funA(data);
});
```

这样就能保证 funA 的调用是在读取到数据后。

9.1.1 JSON

JSON（JavaScript Object Notation）是一种轻量级的数据交换语言，以文字为基础，且易于阅读。尽管 JSON 是 JavaScript 的一个子集，但 JSON 是独立于语言的文本格式，并且采用了类似于 C 语言家族的一些习惯。

JSON 格式的规则如下。

- **对象**：写在花括号（{}）之内。
- **名称/值**：对象里由一系列名称/值组成。名称写在双引号里。

- **值**：名称/值中的值，一个值可以是字符串、数值、布尔值、数组、对象和 null。

例如：

```
{
    "name":"中国"
}
```

这条语句建立了一个对象，对象里有一个名称/值对，名称为 name，值为字符串“中国”。值也可以是数组，例如：

```
{
    "name":"中国",
    "children":
    [
        "name":"浙江" ,
        "name":"广西"
    ]
}
```

children 的值是一个数组，数组里有两个名称/值对。

现假设有一文件名为 city.json，内容为：

```
{
    "name":"中国",
    "children":
    [{
        "name":"浙江" ,
        "children":
        [
            {"name":"杭州" },
            {"name":"宁波" },
            {"name":"温州" },
            {"name":"绍兴" }
    ]
 },
{
    "name":"广西" ,
    "children":
    [
        {"name":"桂林"},
        {"name":"南宁"},
        {"name":"柳州"},
        {"name":"防城港"}
```

```
        ]
    }]
}
```

这是按照 JSON 的格式书写的，下面使用 d3.json()来读取该文件：

```
d3.json("city.json",function(error,data){

    //有错误则输出错误信息
    if (error)
        return console.error(error);

    //输出数据
    console.log(data);
});
```

d3.json()的第一个参数是文件地址，此处是 city.json，表示在本网页文件的同一目录下寻找；第二个参数是一个 function(error, data)函数，其中 data 里保存的是读取到的数据。输出结果如图 9-1 所示。

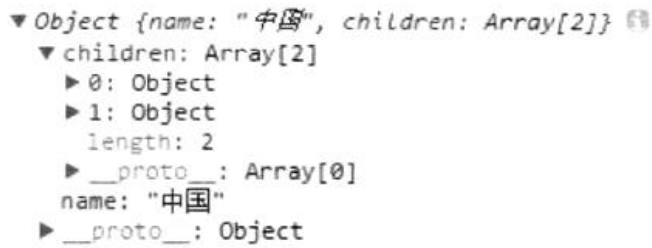


图 9-1　读取 city.json 的结果

从图 9-1 中可以看到，最外层的对象里，有一个属性 name 的值是字符串“中国”，另一个属性 children 的值是一个数组，长度为 2。

要注意，大多数浏览器不能直接读取本地文件，将网页的 HTML 文件与 JSON 文件放到本地目录下是不能正确读取的。第 3.2 节，讲述了使用 Apache 搭建简易服务器的方法。测试本章的用例时，需将 HTML 文件与 JSON 文件放在服务器的根目录下。

假如根目录为 htdocs，在文件夹里新建一个文件夹 project，将网页文件 readjson.html 和 city.json 放在 project 里。然后，启动服务器。之后，打开浏览器，在地址栏输入：

http://127.0.0.1/project/readjson.html

即可。d3.json()、d3.csv()、d3.xml()等都要这么做，否则大部分浏览器会报错（Firefox 除外）。

9.1.2　CSV

Microsoft Excel 或 OpenOffice Calc 等办公软件能够制作表格，简单易懂，对普通用户来说

很方便。Microsoft Excel 默认保存为 xls 格式，OpenOffice Calc 默认保存为 ods 格式。要读取这些格式比较复杂，D3 没有提供这样的方法。但是，表格软件都能够另外保存为 csv 格式，这是一种简单、通用的表格文件。

CSV（Comma Separated Values，**逗号分隔值**），是以纯文本形式存储表格数据，每个单元格之间用逗号（Comma）分隔的表格文件。CSV 格式没有一个通用标准，一般使用的是 RFC 4180 中所定的规则。

CSV 的文本格式如下：

```
省份,人口,GDP
山东,9000,50000
浙江,5000,20000
```

理解起来非常简单，每一个单元格之间用逗号隔开。如果想在单元格里输入逗号怎么办呢？用双引号框起来就行，如下：

```
省份,人口,GDP
山东,"9,000","50,000"
浙江,"5,000","20,000"
```

有些软件在将文件保存为 CSV 格式时，会让用户选择使用什么符号（逗号、分号等）来分隔单元格，请尽量选择逗号。

下面以 OpenOffice 为例，讲述如何编辑和保存表格为 CSV 文件。

（1）首先，打开 OpenOffice Calc。就像 Microsoft Office 中有 Word、Excel、PowerPoint 一样，OpenOffice 中编辑表示使用的是 Calc。打开后，输入单元格的内容，如图 9-2 所示。

（2）选择“文件”→“另存为”命令。在弹出的对话框中文件类型设置为“CSV 文本”，底下再勾选“编辑筛选设置”复选框，如图 9-3 所示。

	A	B	C
1	name	sex	age
2	张三	男	24
3	李四	女	33
4	王五	男	28

图 9-2　编辑表格

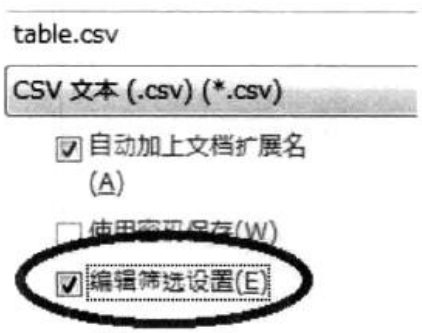


图 9-3　表格另存为

（3）在弹出的对话框中，选择编码（建议用 UTF-8），字段分隔符选择“逗号”，文字分隔符选择“分号”，最后单击“确定”按钮，如图 9-4 所示。

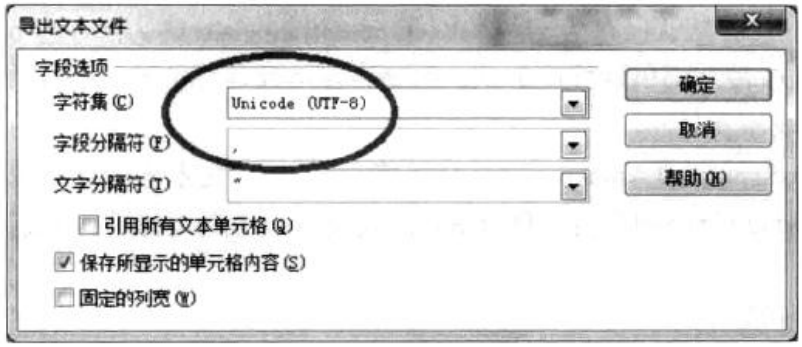


图 9-4　选择编码

（4）保存成功后，用记事本程序打开文件，结果如图 9-5 所示。

name,sex,age
张三,男,24
李四,女,33
王五,男,28

图 9-5　文件内容

要注意，d3.csv()只支持用**逗号**分隔符保存的文件。d3.csv()读取文件的代码如下：

```
d3.csv("table.csv",function(error,csvdata){
    if(error){
        console.log(error);
    }
    console.log(csvdata);
});
```

读取数据后，输出结果如图 9-6 所示。

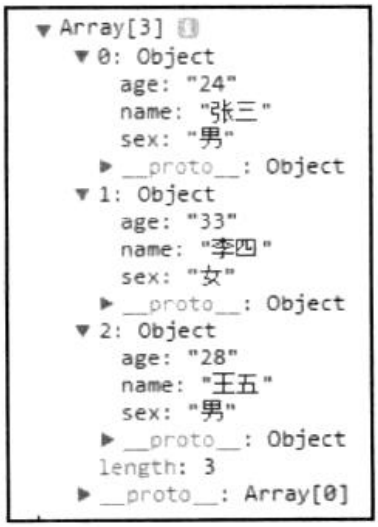


图 9-6　CSV 文件的输出结果

可以看到，变量 csvdata 是一个数组，数组中的每个元素都一个对象，每个对象里都有 age、name、sex 三个成员变量。这三个成员变量正是所编辑表格头一排的三个单元格。如此，就可以在代码中这样调用了：

```
for( var i=0; i<csvdata.length; i++ ){
    var name = csvdata[i].name;
    var sex = csvdata[i].sex;
    var age = csvdata[i].age;
    console.log( "name: " + name + "\n" +
                 "sex: " + sex + "\n" +
                 "age: " + age  );
}
```

CSV 文件是以**逗号**作为单元分隔符的表格文件。除此之外，还有以**制表符 Tab** 作为单元分隔符的 TSV 文件，另外还有其他分隔符的表格文件。

TSV（Tab Separated Values，制表分隔值），和 CSV 文件仅仅是分隔符不一样。它的格式如下：

```
name    age
张三    22
李四    24
```

读取 TSV 文件的方法和读取 CSV 文件的方法是一样的，只要更改一下函数名即可：

```
d3.tsv("table.tsv",function(error,tsvdata){
    console.log(tsvdata);
});
```

其实，d3.csv()和 d3.tsv()本质上是一个方法，下面是它们在源码中的定义：

```
d3.csv = d3.dsv(",", "text/csv");
d3.tsv = d3.dsv("  ", "text/tab-separated-values");
```

可以看到，它们都是 d3.dsv()。d3.dsv()可以读取以**任意字符或字符串**作为分隔符的表格文件。接下来试试用 d3.dsv()读取以**分号**作为分隔符的表格文件。假设有如下文件：

```
name;age
张三;22
李四;24
```

读取的代码如下：

```
var dsv = d3.dsv(";", "text/plain");
dsv("table.dsv",function(error,dsvdata){
    if(error)
        console.log(error);
```

```
    console.log(dsvdata);
});
```

调用 d3.dsv()，第一个参数是分号，即分隔符。d3.dsv 返回一个函数，保存在变量 dsv 上。然后，就可以将 dsv 当作是 d3.csv 和 d3.tsv 一样使用。但是，要注意不再用 d3.dsv，而是直接使用变量 dsv 即可。

使用 d3.csv()读取 CSV 文件时，有时会出现乱码问题。例如，使用 d3.csv()读取 xxx.csv 文件，如果 xxx.csv 文件使用的是 UTF-8 编码，不会有什么问题。但是，如果 xxx.csv 文件是 UTF-8 编码，用 Microsoft Excel 打开可能会出现乱码。因为国内的 Excel 默认使用 GB2312 打开，而且在打开的时候不能选择编码（OpenOffice 没有这个问题）。

GB2312 和 GB18030 是国内常用的编码，如果 CSV 文件用这两种编码保存，那么用 Excel 可直接打开而不出现乱码。但是，如果使用 GB2312 和 GB18030，在用 d3.csv()读取的时候，又会在可视化的时候出现乱码，这正是问题所在。

因此，需要在调用函数的时候设定编码。

d3.csv()和 d3.tsv()两个函数实质上都是 d3.dsv 函数，在定义的时候，第二个参数是可以设置编码的。可重定义 d3.csv()和 d3.tsv()如下：

```
var csv = d3.dsv(",", "text/csv;charset=gb2312");
var tsv = d3.dsv(" ", "text/tab-separated-values;charset=gb2312");
```

然后直接使用变量 csv 和 tsv 读取文件：

```
csv("xxx.csv",function(error,csvdata){

}

tsv("xxx.tsv",function(error,tsvdata){

}
```

如此即可解决乱码问题。

9.1.3 XML

XML（Extensible Markup Language，可扩展标记语言），是一种类似 HTML 的标记语言，设计宗旨是传输数据。现有如下内容：

```
<note>
<title>XML Text</title>
<date>2015.2.17</date>
<body>Hello, XML</body>
```

```
</note>
```

XML 的标签需要自行定义。D3 提供了 d3.xml()来读取 XML 文件。假设包含上述内容的 XML 文件的文件名为 example.xml，则：

```
d3.xml("example.xml",function(error,xmlDocument){
    if (error)
        return console.error(error);

    console.log( xmlDocument.getElementsByTagName("title")[0]
                            .innerHTML );
    console.log( xmlDocument.getElementsByTagName("date")[0]
                            .innerHTML );
    console.log( xmlDocument.getElementsByTagName("body")[0]
                            .innerHTML );
});
```

xmlDocument 可像 HTML 中的 document 一样，用 getElementsByTagName、getElementById 等方法来获取文件中的标签元素。读取 XML 文件后在控制台的输出结果如图 9-7 所示。

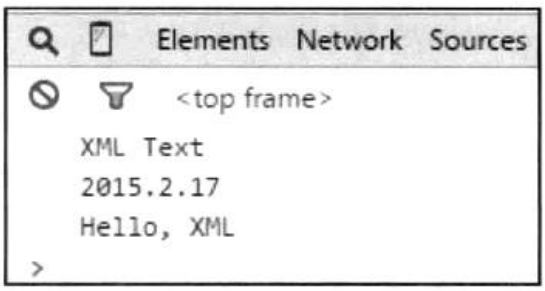


图 9-7　读取 XML 文件后在控制台的输出结果

9.1.4 TEXT

文本文件使用 d3.text()读取。现有文件 note.txt，内容如下：

```
This is a text file.
```

读取该文件的代码如下：

```
d3.text("note.txt",function(error,txtdata){
    if (error)
        return console.error(error);

    d3.select("body")
        .append("p")
```

```
        .text(txtdata);
});
```

读取之后，文本内容保存在 txtdata 里。这段代码在 body 中新建了一个 p 元素，其内容为 txtdata，结果如图 9-8 所示。

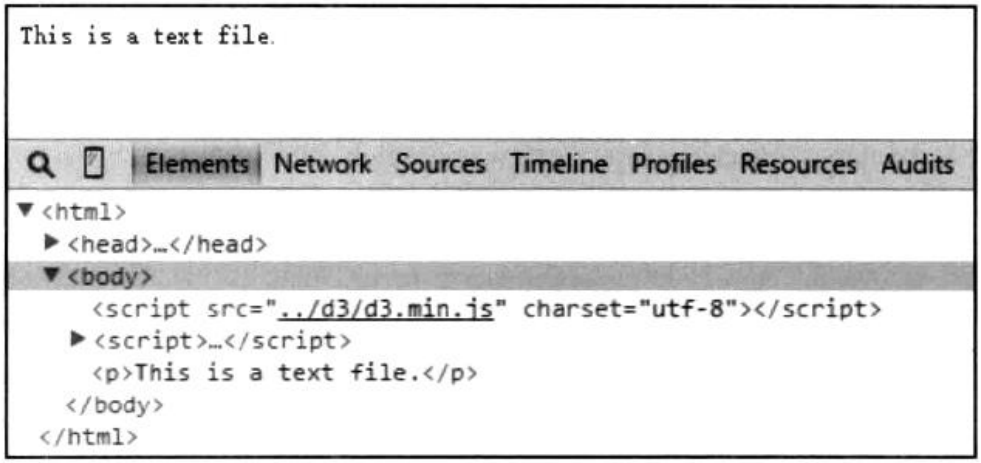


图 9-8　txt 文件的输出结果

9.2　文件导出

用户可能会希望将在网页上看到的可视化图形保存为一个文件，以便用于发表、传输，或在其他软件中编辑。

9.2.1　导出为 SVG 文件

D3 大多数是在 SVG 区域里绘制图形的。那么，怎样才能将此区域输出成一个单独的.svg 文件呢？这里提供两种方法：SVG Crowbar 和 d3-downloadable。

1. SVG Crowbar

SVG Crowbar 是一个谷歌浏览器的书签式插件，官方网页如下：

http://nytimes.github.io/svg-crowbar/

用谷歌浏览器打开该网页，找到如图 9-9 所示图标。

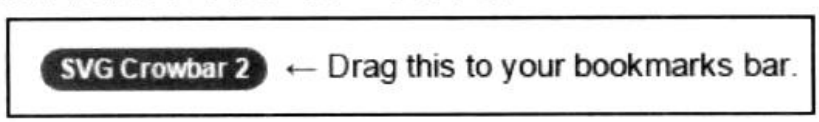


图 9-9　SVG Crowbar2

将其拖到书签栏中。如果浏览器上方没有显示书签栏，可在浏览器的设置（chrome://settings）中，勾选如图 9-10 所示的复选框。

将如图 9-9 所示的 SVG Crowbar 2 拖到书签栏中，书签栏中会多出一个书签，如图 9-11 所示。

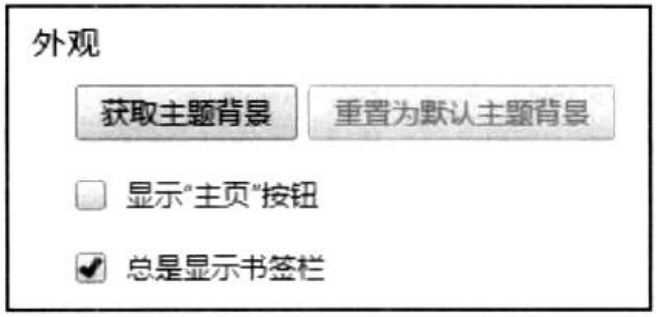


图 9-10　显示书签栏

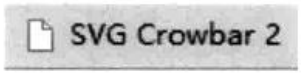


图 9-11　将 SVG Crowbar 2 拖到书签栏中

好了，现在对于网页中的 SVG 区域，可以单击 SVG Crowbar2 来保存。例如，在浏览器中打开第 6 章制作的折线图，点击书签后，如图 9-12 所示。

左上角出现了一个图标，上有“Download”按钮。单击该按钮即可保存文件，该文件的后缀名为.svg，可直接用浏览器打开。

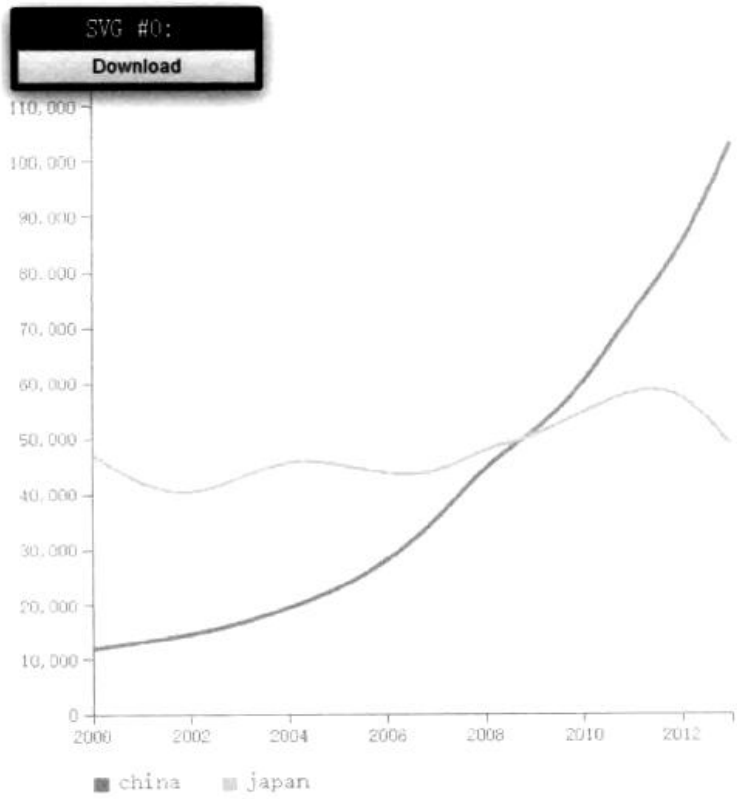


图 9-12　保存为 SVG 文件

但是，这个方法只支持于谷歌浏览器（Chrome）。

2．d3-downloadable

d3-downloadable 是使用 JavaScript 写的，可以做到让用户单击 SVG 区域后，显示一个选择框，能够保存为 SVG、PNG、JPG 三种格式。在下面的网址：

https://github.com/likr/d3-downloadable

可以下载。需要的文件有两个：d3-downloadable.js 和 d3-downloadable.css。将它们放在一个文件夹 d3-downloadable 里，再将文件夹复制到工程目录里。使用时，在<head></head>中加入：

```
<link rel="stylesheet"
      href="d3-downloadable/d3-downloadable.css"/>
```

再引用 js 文件：

```
<script src="d3-downloadable/d3-downloadable.js"></script>
```

下面看一个例子：

```
var width  = 500;  //SVG区域的宽度
var height = 500;  //SVG区域的高度

var svg = d3.select("body")
            .append("svg")
            .attr("width", width)
            .attr("height", height);

//添加一个矩形
svg.append("rect")
   .attr("x",100)
   .attr("y",100)
   .attr("width",100)
   .attr("height",100)
   .style("fill","red");

//添加一个圆
svg.append("circle")
   .attr("cx",300)
   .attr("cy",150)
   .attr("r",50)
   .style("fill","blue");

//调用d3-downloadable，指定宽度、高度和文件名
svg.call(d3.downloadable({
```

```
    width: width,
    height: height,
    filename: "filename"
}));
```

结果如图 9-13 所示。右键单击 SVG 区域后，弹出一个选择框，能够保存为 SVG、PNG、JPG 三种格式。

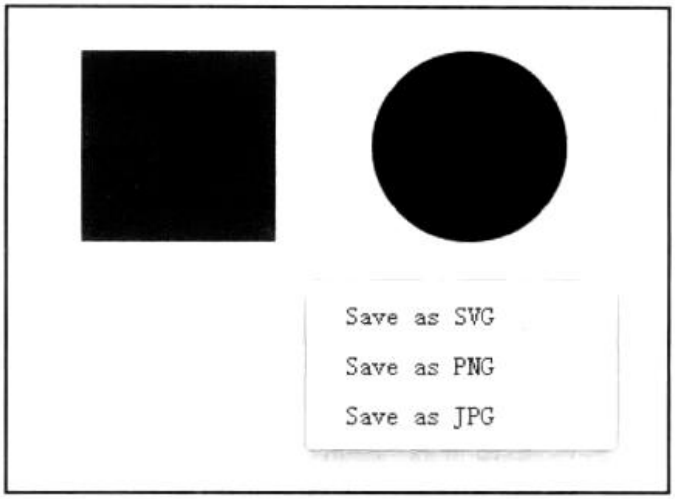


图 9-13　右键单击 SVG 区域后，弹出选择框

但是，d3-downloadable 不支持外部链接的图形元素样式，即如果使用下述方式：

```
<style>
    svg rect{
        fill: red;
    }
</style>
```

该样式不会被保存到 svg 的 rect 元素中。因此，需要在添加元素时直接赋予其属性：

```
svg.append("rect")
    .style("fill","red");
```

9.2.2　编辑矢量图

将可视化图形保存为 SVG 文件后，可以在其他软件中编辑、修改，也可以输出成别的文件格式，如 PNG、PDF 等。这里介绍一个软件 inkscape。

inkscape 是一个矢量图形编辑器，是开源软件，与 Illustrator、Freehand、CorelDraw、XaraX 等软件很类似，且具有跨平台特性。inkscape 可以导入各种矢量图文件，经过编辑后，可以导出成 PNG、PDF 等格式。inkscape 的界面如图 9-14 所示。

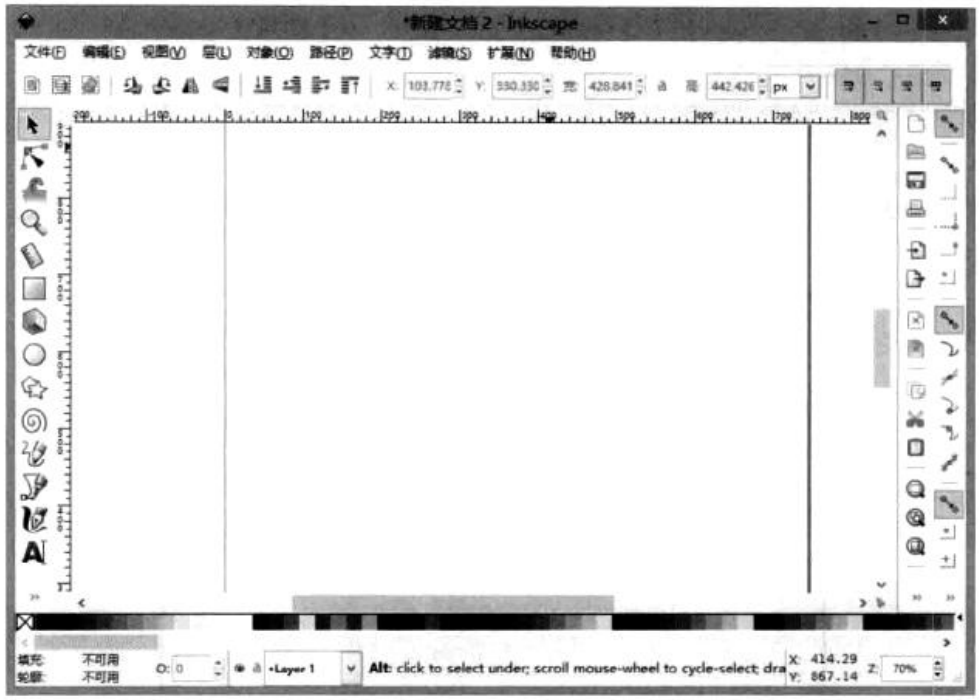


图 9-14　inkscape 的界面

inkscape 可在以下网址免费下载：

https://inkscape.org/

安装后，运行软件，界面如图 9-14 所示。选择“文件”→“另存为”命令，可以在弹出的对话框中看到能够导出的文件类型。如图 9-15 所示，其中包括 SVG、PDF、PNG、EPS、EMF 等常用格式。

图 9-15　保存类型

例如，将图 9-13 的图形保存为 SVG 文件后，将其拖入到 inkscape 中，添加一些图形后，如图 9-16 所示。

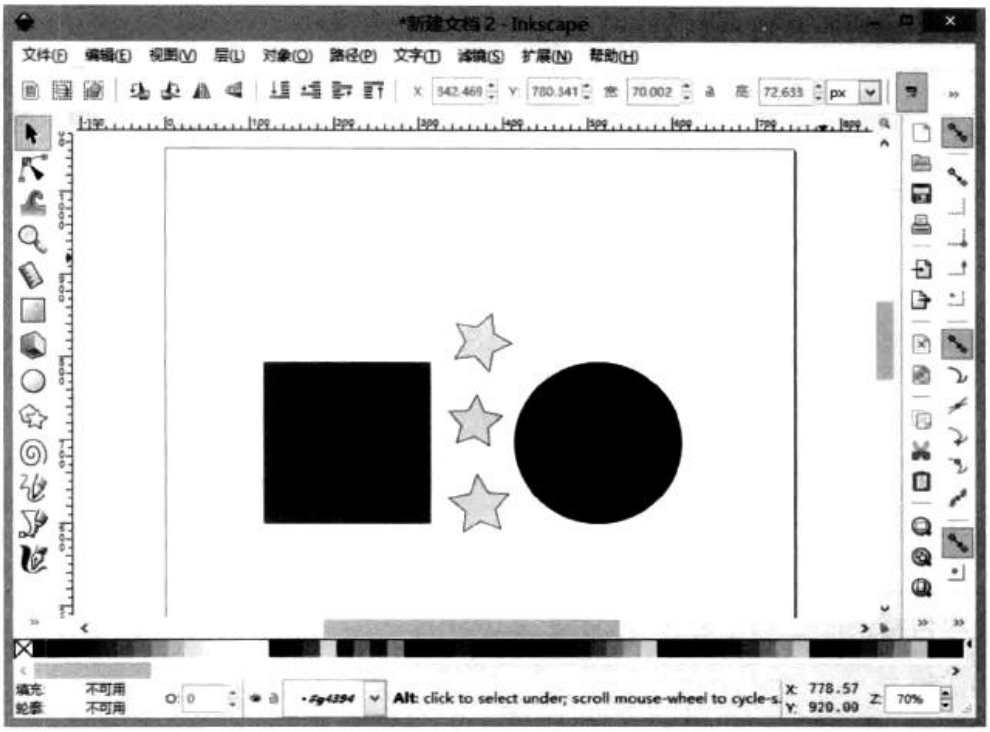


图 9-16　编辑图形

将其另存为 PDF 文件，用 Adobe Reader 打开该文件，如图 9-17 所示。

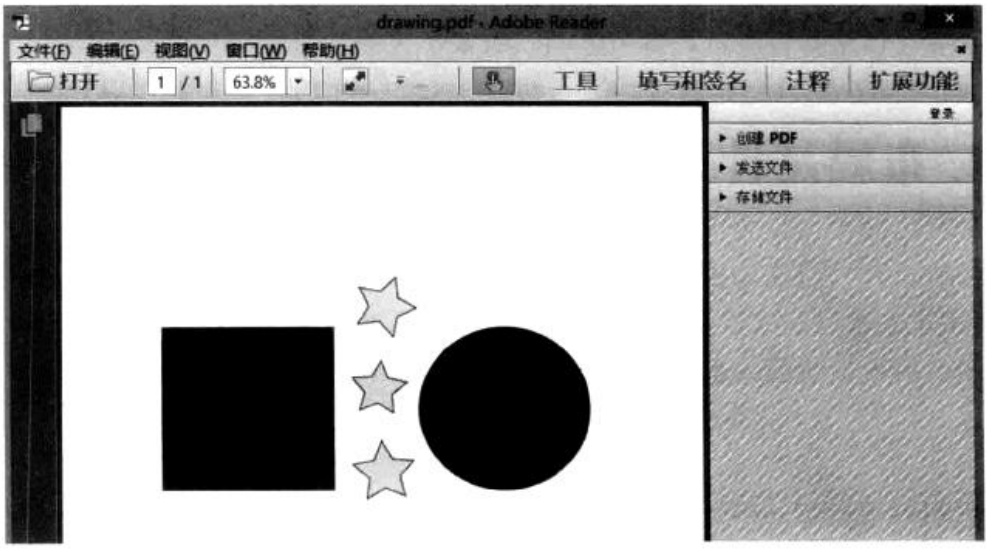


图 9-17　在 Adobe Reader 中打开 PDF 文件

第 10 章

布局

本章内容包括：

- 布局的概念
- 各种布局的用例

第 1 节，讲述布局是什么。

第 2 节到第 12 节，分别介绍各布局的用法，共 11 个布局，每一个都配有示例。关于此部分的内容，读者可以选择性阅读，各节之间没有太多关联性。

10.1 布局是什么

布局（Layout），是 D3 中很重要的内容，使用布局能够轻松制作很多图表。从字面来理解“布局”有些困难，容易让人误以为其功能是“绘制”：在这里画圆、在那里画矩形。其实，布局是为了绘图，但不是绘图。本章中，为了方便初学者理解布局，从布局的表面作用出发，将布局称呼为：

数据转换。

什么意思呢？假设有一个数据 [10, 20, 30, 40]。如果要将此数组绘制成饼状图，直接使用数组中的数据是不行的，需要将数据“转换”成起始角度和终止角度。例如，将 10 转换成 0（起始）～π/5（终止）。有了起始角度和终止角度，才能够绘制饼状图。布局的意义就在于**计算出方便绘图的数据**。至于用什么来绘制，怎样绘制，在 SVG 还是在 Canvas 里绘制，却与布局无

关。当然，你也可以按原字面“布局”来理解，也可理解为“计算”，只要知道意思即可。

在前面的章节中，制作了柱形图、散点图、折线图。D3 没有提供这三种图表的布局，因为这三种图表足够简单，不需要进行复杂的数据转换。需要使用布局来制作的图表，会涉及一些数学运算，而开发者一般不希望为此去查阅数学书籍。有了布局，一切都会变得简单。

D3 总共提供了 12 个布局：饼状图（Pie）、力导向图（Force）、弦图（Chord）、树状图（Tree）、集群图（Cluster）、捆图（Bundle）、打包图（Pack）、直方图（Histogram）、分区图（Partition）、堆栈图（Stack）、矩阵树图（Treemap）、层级图（Hierarchy）。

12 个布局中，层级图（Hierarchy）不能直接使用。集群图、打包图、分区图、树状图、矩阵树图是由层级图扩展来的。如此一来，能够使用的布局是 11 个（有 5 个是由层级图扩展而来）。这些布局的作用都是将某种数据转换成另一种数据，而转换后的数据是利于可视化的。

布局的使用遵循以下顺序。

（1）确定初始数据。

（2）转换数据。

（3）绘制。

初始数据是指**转换前的数据**，转换前的数据要符合一定的格式要求，才能被布局函数转换。**转换数据**指的是布局根据初始数据计算绘图所需数据。**绘制**是使用转换后的数据在 SVG 画板上绘图，很可能会用到第 6 章介绍的生成器。本章的第 2 节到第 12 节，每一节都讲解一个布局，每个布局都会按照以上顺序制作一个示例。

10.2　饼状图

饼状图（Pie Chart），简称**饼图**，通过将圆形划分为几个扇形，来描述数量或百分比的关系。扇形的大小与数量的大小成比例，所有扇形正好组成一个完整的圆。

饼状图布局（Pie Layout），能根据“一系列数值”计算出“一系列对象”，每一个对象都包含起始角度和终止角度等绘制饼状图所需的数据。

- d3.layout.**pie**()

创建一个饼状图布局。

- **pie**(values[, index])

转换数据，转换后的每一个对象中都包含有起始角度和终止角度。

- pie.**value**([accessor])

设定或获取值访问器，即如何从接收的数据中提取初始值。如果省略参数，则返回当前的值访问器。

- pie.**sort**([comparator])

设定或获取比较器，用于排序，例如 d3.ascending 和 d3.descending。如果省略，则返回当前的比较器。

- pie.**startAngle**([angle])

设定或获取饼状图的起始角度，默认为 0（弧度）。

- pie.**endAngle**([angle])

设定或获取饼状图的终止角度，默认为 2π（弧度）。

下面来制作一个饼状图，要求如下。

有 2014 年各大智能手机厂商在中国的出货量数据，将此数据进行可视化反映到饼状图上。其中，饼状图的每一段弧上都标注出该厂商的出货量占总数的百分比，弧外标注出厂商名称。

1. 确定初始数据

有如下数据：

```
var dataset = [ ["小米",60.8], ["三星",58.4], ["联想",47.3],
                ["苹果",46.6], ["华为",41.3], ["酷派",40.1] ,
                ["其他",111.5] ];
```

这是 2014 年各大智能手机厂商在中国的出货量。例如，["小米", 60.8]，表示小米的出货量为 60.8，单位是百万台。现要将此数据反映到饼状图上。

2. 转换数据

代码如下：

```
var pie = d3.layout.pie()       //创建饼状图布局
          .value(function(d){ return d[1]; });    //值访问器

//dataset为初始数据，piedata为转换后的数据
var piedata = pie(dataset);

console.log(piedata);       //输出转换后的数据
```

首先，使用 d3.layout.pie()创建一个布局，并设定值访问器如下：

```
function(d){
   return d[1];
}
```

值访问器，就是如何从传入的数组里提取需要转换数据的方法。因此，要根据所接收的数组的格式，来设定值访问器。设定之后，数组的每一项都会调用该访问器，并返回一个值，此返回值就是接下来要用于计算的数值。在这里，数组是 dataset，其各项元素是：

```
["小米",60.8], ["三星",58.4], ["联想",47.3] ...
```

每一项都调用值访问器，分别返回：

```
60.8, 58.4, 47.3 ...
```

这就是实际使用的数组。创建好布局后，将其赋值给变量 pie，那么变量 pie 就可以当函数使用，参数是初始数据，返回值是转换后的数据。

调用 pie(dataset)，参数为 dataset，返回值保存到 piedata。然后，在控制台输出转换后的数据，如图 10-1 所示。

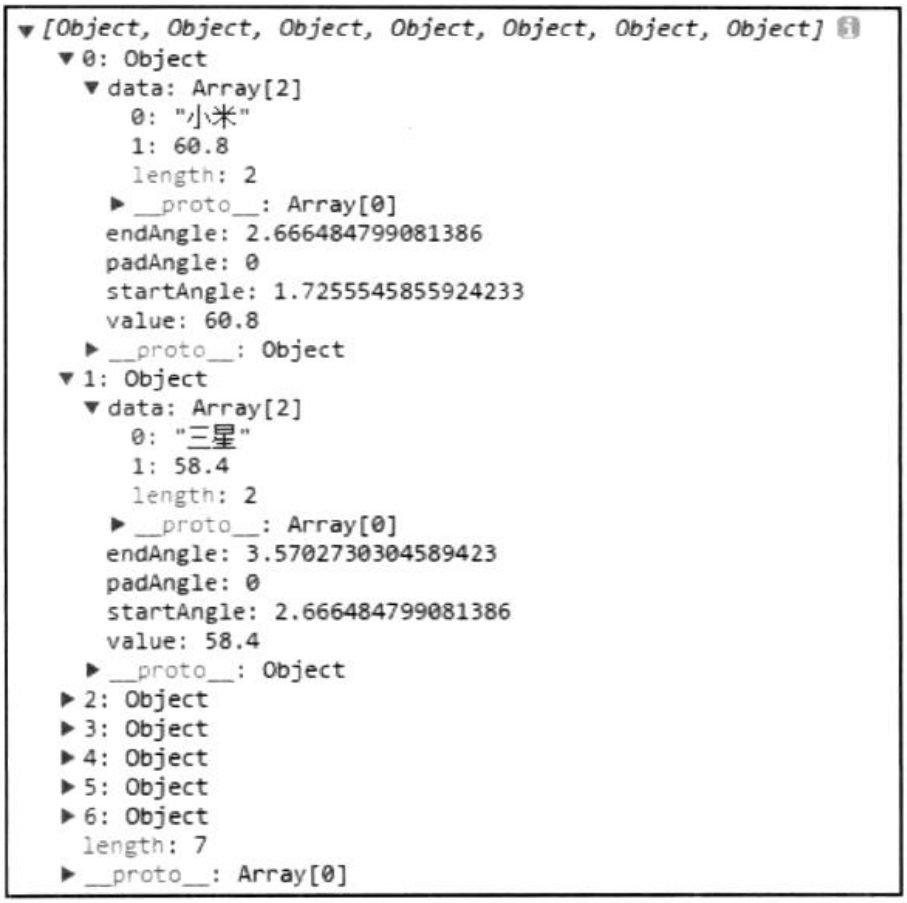


图 10-1　被饼状图布局转换后的数据

由图 10-1 可以看到，转换后的数据也是一个数组，数组的每一项是一个对象 object。各对象里的变量如下。

- data：转换前的数据项。
- value：从 data 数据项中提取出来的值（取决于 value 访问器）。
- startAngle：这段弧的起始角度（弧度）。
- endAngle：这段弧的终止角度（弧度）。

有了转换后的数据，就可以绘制图表了。

3. 绘制

绘制饼状图要使用第 6.4 节提到的弧生成器，可以发现，弧生成器绘制图形所需的参数也是 startAngle 和 endAngle。因此，饼状图布局和弧生成器是配套使用的。首先，创建一个弧生成器，代码如下：

```
//外半径和内半径
var outerRadius = width / 3;
var innerRadius = 0;

//创建弧生成器
var arc = d3.svg.arc()
            .innerRadius(innerRadius)
            .outerRadius(outerRadius);

var color = d3.scale.category20();
```

outerRadius 和 innerRadius 分别是弧的外半径和内半径，width 是 SVG 的宽度，值是 400。最后一行，定义了一个颜色比例尺，保存在变量 color 里。

其次，在<svg>里添加几个<g>元素，每一个<g>用于包含一段弧。弧由三部分组成：路径<path>、文字<text>、直线<line>。代码如下：

```
//添加对应数目的弧组，即<g>元素
var arcs = svg.selectAll("g")
              .data(piedata)        //绑定转换后的数据piedata
              .enter()
              .append("g")
              .attr("transform",
                 "translate("+( width/2 )+","+ ( height/2 ) +")");
```

这段代码返回一个<g>元素的选择集，<g>的数量与数组 piedata 的长度一致。其中，所有<g>元素都添加了 transform 属性，被平移到 SVG 画板的中心(width/2, height/2)。选择集保存在变量 arcs 中，如此一来，只要用 arcs.append()就可以同时为所有<g>添加元素。接下来按顺序向<g>中添加元素。

先向<g>里添加路径元素<path>：

```
//添加弧的路径元素
arcs.append("path")
       .attr("fill",function(d,i){
           return color(i);     //设定弧的颜色
       })
       .attr("d",function(d){
           return arc(d);       //使用弧生成器获取路径
       });
```

弧的颜色用颜色比例尺返回。路径元素<path>的属性 d 是路径值，前面定义的弧生成器就是为了在这里生产路径值用的。函数体中的变量 d，就是被绑定的数组的各项元素，每一项都是一个对象，由图 10-1 可以知道各对象的值。

添加路径元素后，饼状图的主体部分就有了，如图 10-2 所示，一共有七段弧。由于内半径设定为 0，因此中心处没有空白。

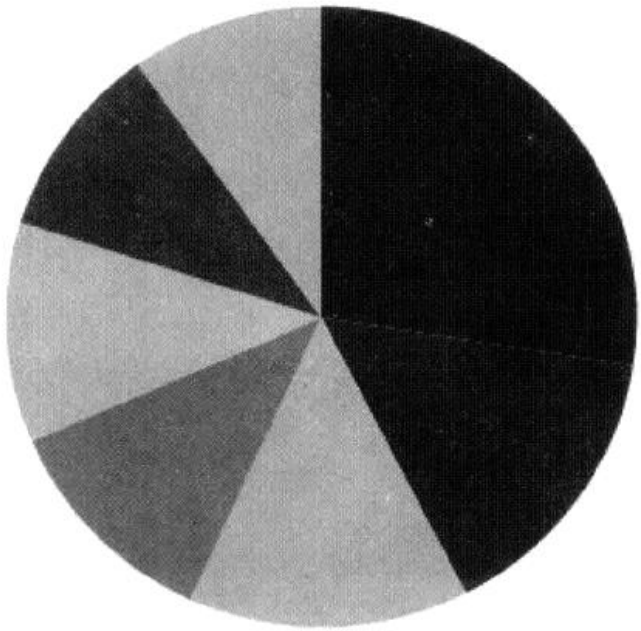


图 10-2　绘制弧

再向<g>中添加弧内文字，文字的内容是各厂商在中国的市场份额的百分比：

```
//添加弧内的文字元素
arcs.append("text")
    .attr("transform",function(d){
        var x = arc.centroid(d)[0] * 1.4;  //文字的x坐标
        var y = arc.centroid(d)[1] * 1.4;  //文字的y坐标
        return "translate(" + x + "," + y + ")";
    })
    .attr("text-anchor","middle")
    .text(function(d){
        //计算市场份额的百分比
        var percent = Number(d.value) /
                d3.sum(dataset,function(d){ return d[1]; }) * 100;

        //保留1位小数点，末尾加一个百分号返回
        return percent.toFixed(1) + "%";
    });
```

计算文字的坐标时，使用了 arc.centroid()，此方法能计算弧的中心坐标。arc.centroid(d)返回一个数组[x, y]，分别表示 x 和 y 方向的坐标。但是，弧的中心是相对于圆心来说的。例如，如果某段弧的中心坐标为(50, 50)，不是说弧的中心在 SVG 画板的(50, 50)处，而是在以圆心为坐标原点的(50, 50)处。

如图 10-3 所示，弧中心坐标是 arc.centroid(d)，弧边缘的中心坐标可用 arc.centroid(d)乘以 2 来计算。也就是说，使用“arc.centroid(d)乘以一个系数的方法”可以获取圆心和弧中心所在直线上的任意点。

添加了弧内文字后，如图 10-4 所示。

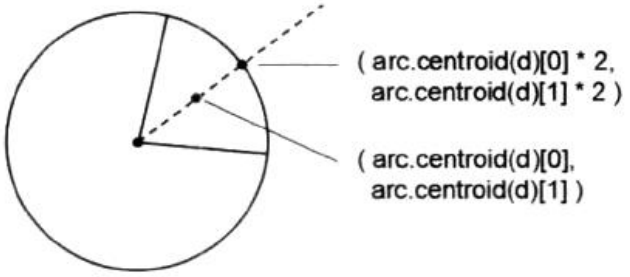


图 10-3　弧中心的计算

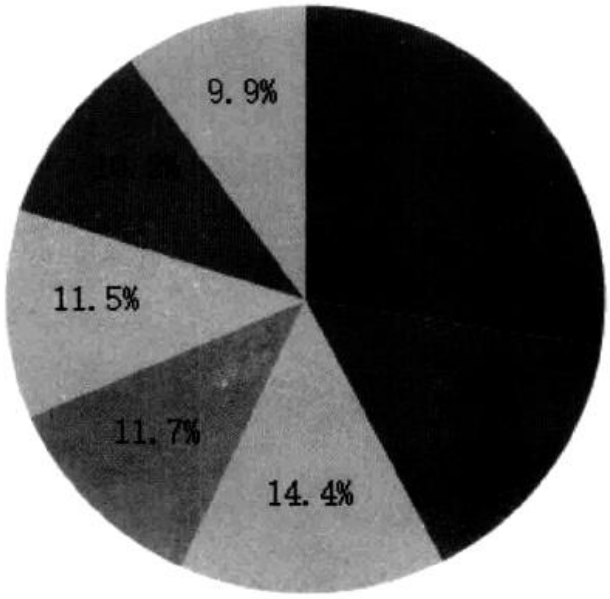


图 10-4　添加了弧内文字

接下来，还需添加弧外文字，用于标注弧是属于哪一家公司的。从弧的边缘绘制一条直线，直线外端有文字。这里也要用到 arc.centroid()，代码如下：

```
//添加连接弧外文字的直线元素
arcs.append("line")
    .attr("stroke","black")
    .attr("x1",function(d){ return arc.centroid(d)[0] * 2; })
    .attr("y1",function(d){ return arc.centroid(d)[1] * 2; })
    .attr("x2",function(d){ return arc.centroid(d)[0] * 2.2; })
    .attr("y2",function(d){ return arc.centroid(d)[1] * 2.2; });

//添加弧外的文字元素
arcs.append("text")
        .attr("transform",function(d){
            var x = arc.centroid(d)[0] * 2.5;
```

```
        var y = arc.centroid(d)[1] * 2.5;
        return "translate(" + x + "," + y + ")";
    })
    .attr("text-anchor","middle")
    .text(function(d){
        return d.data[0];
    });
```

结果如图 10-5 所示。

默认情况下，饼状图是由 0～2π 组成一个完整的圆。但是，也可以在定义布局时设定其范围，例如更改布局为：

```
var pie = d3.layout.pie()
            .startAngle( Math.PI * 0.2 )
            .endAngle( Math.PI * 1.5 )
            .value(function(d){ return d[1]; });
```

范围被设定为 Math.PI * 0.2～Math.PI * 1.5，即 36° 到 270°，如此可得如图 10-6 所示的饼状图。

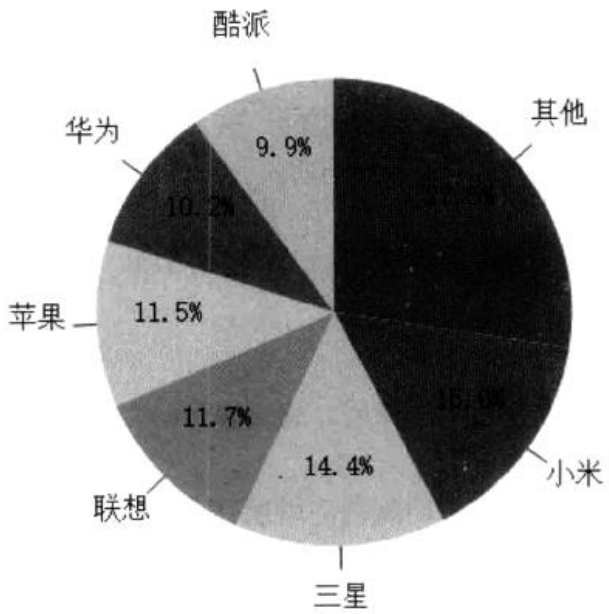


图 10-5　弧外的文字

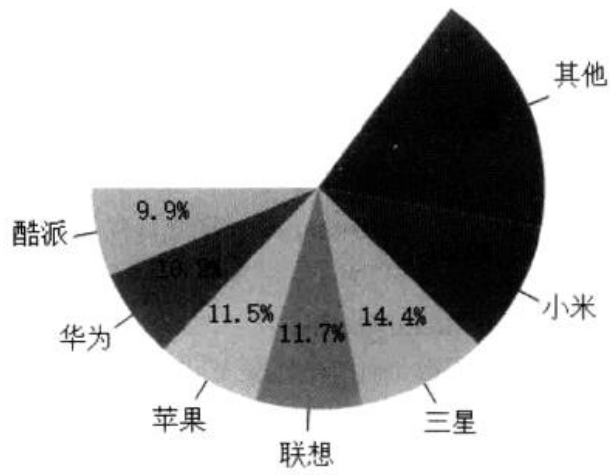


图 10-6　设定范围的饼状图

10.3　力导向图

力导向图（Force-Directed Graph），是绘图的一种算法。在二维或三维空间里配置**节点**，节

点之间用线连接，称为**连线**。各连线的长度几乎相等，且尽可能不相交。节点和连线都被施加了力的作用，力是根据节点和连线的相对位置计算的。根据力的作用，来计算节点和连线的运动轨迹，并不断降低它们的能量，最终达到一种能量很低的安定状态。

力导向图布局（Force Layout）所包含的方法甚多，其中有与节点、连线相关的，也有与动画相关的，还有与交互式操作相关的，是所有布局中数量最多的。

- d3.layout.**force**()

创建一个力导向图布局。

- force.**size**([size])

设定或获取力导向图的作用范围，使用方法为 force.size([x, y])，此函数用于指定两件事。

  - 重力的重心位置为（x/2，y/2）。
  - 所有节点的初始位置限定为[0, x]和[0, y]之间。

如果不指定，默认为[1, 1]。

节点相关

- force.**nodes**([nodes])

设定或获取力导向图布局的节点数组。如果省略参数 nodes，则返回当前的节点数组。

- force.**friction**([friction])

设定或获取摩擦系数，值的范围是[0, 1]，默认为 0.9。但是这个值其实并非物理意义上的摩擦，其实际意义更接近速度随时间产生的损耗，这个损耗是针对节点的。值为 1，表示没有速度的损耗；值为 0，表示速度的损耗最大。如果省略参数，则返回当前值。

- force.**charge**([charge])

设定或获取节点的电荷数。该参数决定是排斥还是吸引，默认值为–30。值为正，则相互吸引，绝对值越大吸引力越大。值为负，则相互排斥，绝对值越大排斥力越大。

- force.**chargeDistance**([distance])

设定或获取引力的作用距离，超过这个距离，则没有引力的作用。默认值为无穷大。

- force.**theta**([theta])

设定或获取限制值。如果节点数过多，力导向图布局的计算时间就会增加（复杂度为 $O(n \log n)$）。theta 就是为了限制这个计算而存在的，默认值为 0.8。这个值越小，就能把计算限制得越紧。

- force.**gravity**([gravity])

设定或获取重力。以 size()设定的中心产生重力，各节点都会向中心运动，默认值为 0.1。如果设定为 0，则没有重力的作用。

连线相关

- force.**links**([links])

设定或获取力导向图布局的连线数组。如果省略参数 links，则返回当前的连线数组。

- force.**linkDistance**([distance])

设定或获取连线的长度，默认为 20。

- force.**linkStrength**([strength])

设定或获取连线的坚硬度，值的范围为[0, 1]，值越大越坚硬。其直观感受是：

- 值为 1，则拖动一个节点 A，与之相连节点会与 A 保持 linkDistance()设定的距离运动。
- 值为 0，则拖动一个节点 A，与之相连的节点不会运动，连线会被拉长。

动画相关

- force.**alpha**([value])

设定或获取动画的冷却系数，运动过程中该系数会不断减小，直到小于 0.005 为止，此时动画停止。

- force.**start**()

将 alpha 设定为 0.1 后，开始计算。

- force.**stop**()

等价于 alpha(0)，停止动画。

- force.**resume**()

等价于 alpha(0.1)，重新启动动画。但是，这个函数通常不直接使用，在 start()和 drag()的内部自动调用。

- force.**tick**()

动画的计算进入到下一步。

- force.**on**(type, listener)

type 是事件类型，有三种事件：start、tick、end。listener 是监听器。

交互式相关

- force.**drag**()

用于拖曳操作的函数。

下面制作一个力导向图，要求如下。

有一系列节点和连线，令其产生力导向作用。节点用圆表示，连线用线段表示。每一个节点都有一个名称，名称要绘制出来，并且力导向图要支持拖曳。

1．确定初始数据

初始数据是节点数组 nodes 和连线数组 edges，如下：

```
var nodes = [ { name: "0" },
              { name: "1" },
              { name: "2" },
              { name: "3" },
```

```
                { name: "4" },
                { name: "5" },
                { name: "6" }];

var edges = [ { source : 0  , target: 1 } ,
              { source : 0  , target: 2 } ,
              { source : 0  , target: 3 } ,
              { source : 1  , target: 4 } ,
              { source : 1  , target: 5 } ,
              { source : 1  , target: 6 } ];
```

nodes 的对象只有一个变量 name，表示该节点的名称。edges 的对象里有两个变量 source 和 target，分别表示连线两端的节点，节点的序号从 0 开始。要注意，此处必须使用 source 和 target 这两个名称。

2．转换数据

创建一个力导向图的布局，将 nodes 和 edges 作为布局被转换的数据，并设定连线的距离为 90，节点的电荷数为–400，其他参数使用默认值：

```
var force = d3.layout.force()
                .nodes(nodes)              //设定节点数组
                .links(edges)              //设定连线数组
                .size([width,height])      //设定作用范围
                .linkDistance(90)          //设定连线的距离
                .charge(-400);             //设定节点的电荷数

force.start(); //开启布局计算

console.log(nodes);     //输出转换后的节点数组
console.log(edges);     //输出转换后的连线数组
```

调用 force.start()后，布局开始计算数据。力导向图与其他布局不同的是：转换数据时，源数组是会变化的。转换后的节点数组如图 10-7 所示。

可以看到，每一个节点里保有原来的 name 属性不变，另外还多出了一大堆属性。其中：

- index：节点的索引号。
- x：当前 x 坐标。
- y：当前 y 坐标。
- px：上一时刻的 x 坐标。
- py：上一时刻的 y 坐标。
- fixed：节点是否固定。
- weight：节点的权重（有多少连线与之相连）。

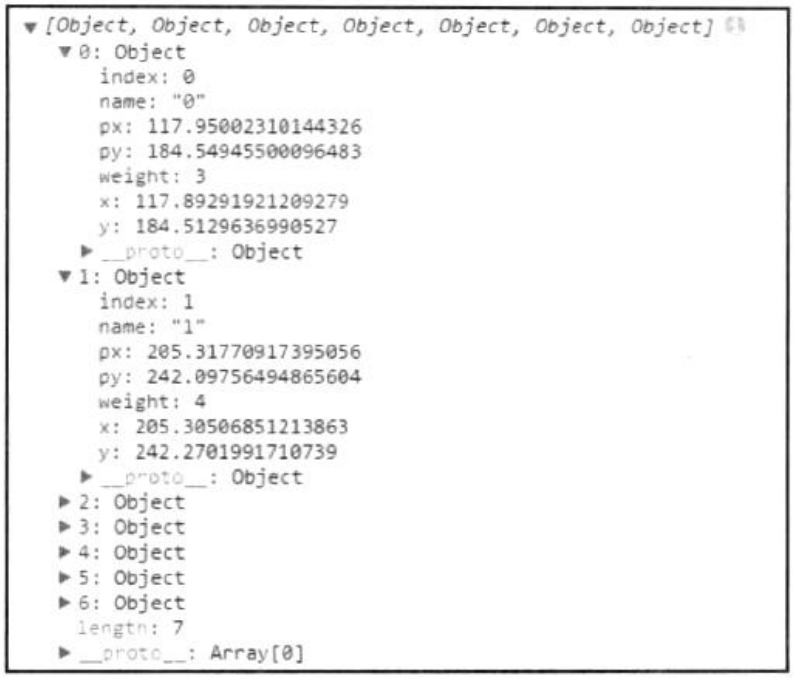


图 10-7　转换后的节点数组

转换后的连线数组如图 10-8 所示。连线数组中的每一个对象，还是有两个变量 source 和 target，只是内容变成了节点对象，而不是刚开始的索引号。

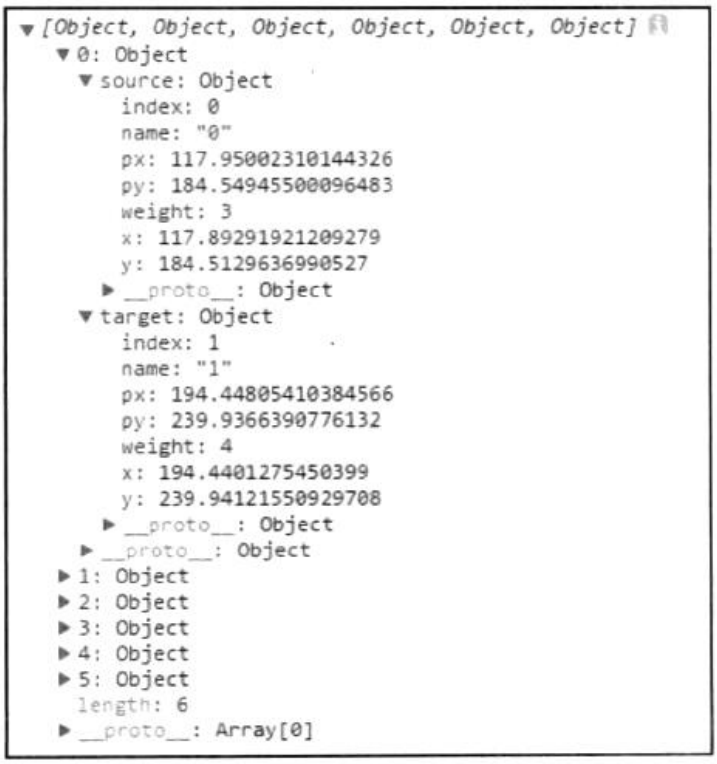


图 10-8　转换后的连线数组

3. 绘制

绑定数组 nodes 和 edges，分别添加节点的元素<circle>和连线的元素<line>。另外，还要添加文字元素<text>，以标注节点的名称。其中，各元素的 CSS 样式名称分别为：forceCircle、forceText、forceLine。代码如下所示：

```
var color = d3.scale.category20();

//绘制连线
var lines = svg.selectAll(".forceLine")
                .data(edges)
                .enter()
                .append("line")
                .attr("class","forceLine");

//绘制节点
var circles = svg.selectAll(".forceCircle")
                .data(nodes)
                .enter()
                .append("circle")
                .attr("class","forceCircle")
                .attr("r",20)
                .style("fill",function(d,i){
                    return color(i);
                })
                .call(force.drag); //允许拖动

//绘制文字
var texts = svg.selectAll(".forceText")
                .data(nodes)
                .enter()
                .append("text")
                .attr("class","forceText")
                .attr("x",function(d){ return d.x; })
                .attr("y",function(d){ return d.y; })
                .attr("dy", ".3em")
                .text(function(d){ return d.name; });
```

节点的选择集调用了 call(force.drag)，如此可让节点支持鼠标拖曳。如此一来，画板上的图形元素就具备了。但是，力导向图是时刻更新的，更新之后节点的坐标会变化，图形元素当然也需要更新。因此，还需设置一个监听器，当力导向图更新时，更新所有的图形元素。

force.on()可为三种事件设定监听器：start、tick、end。其中，start 是刚开始运动，end 是运动停止，tick 是表示运动的每一步。因此，只需为 tick 事件设置监听器即可：

```
//tick事件的监听器
force.on("tick", function(){
    //更新连线的端点坐标
    lines.attr("x1",function(d){ return d.source.x; });
    lines.attr("y1",function(d){ return d.source.y; });
    lines.attr("x2",function(d){ return d.target.x; });
    lines.attr("y2",function(d){ return d.target.y; });

    //更新节点坐标
    circles.attr("cx",function(d){ return d.x; });
    circles.attr("cy",function(d){ return d.y; });

    //更新节点文字的坐标
    texts.attr("x",function(d){ return d.x; });
    texts.attr("y",function(d){ return d.y; });
});
```

选择集 lines、circles、texts 上都绑定了数据，当每一次 tick 事件发生时，被绑定的数据被更新。那么，function(d)中的 d 也都更新了。因此，对于 lines、circles、texts，再用其被绑定的数据来设定属性值即可。

结果如图 10-9 所示。

拖动节点 1 后，其他节点在力导向的作用下跟着运动，如图 10-10 所示。

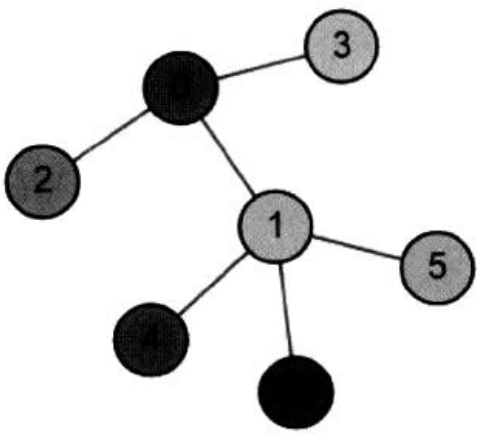


图 10-9　绘制结果

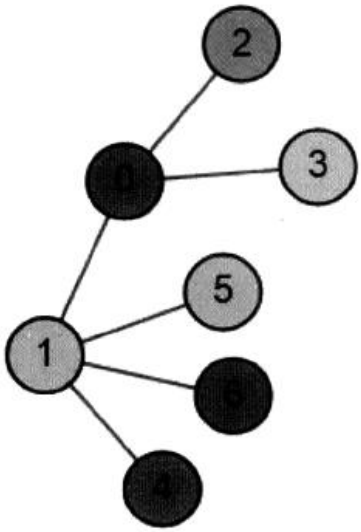


图 10-10　拖动节点

除了 tick 事件，还可以为 start 和 end 事件添加监听器，例如在运动开始和结束时在控制台输出文字，这样可以更好地理解力导向图什么时候开始作用，什么时候停止作用：

```
//力学图运动开始时
force.on("start", function(){
```

```
    console.log("运动开始");
});

//力学图运动结束时
force.on("end", function(){
    console.log("运动结束");
});
```

除了力导向图整体的事件之外，还有一个重要的事件，这就是拖曳。上面的代码使用了force.drag()，这是力导向布局的拖曳行为。对于拖曳行为，可以定义三种事件的监听器：dragstart、dragend、drag，分别对应拖曳开始时、拖曳结束时、拖曳进行中。

现添加一条要求：**节点被拖曳后自动固定，不再受力的作用，且拖曳时节点变成黄色，拖曳结束后恢复**。定义以下拖曳行为：

```
var drag = force.drag()
                .on("dragstart",function(d){
                    //拖曳开始后设定被拖曳对象为固定
                    d.fixed = true;
                })
                .on("dragend",function(d,i){
                    //拖曳结束后变为原来的颜色
                    d3.select(this).style("fill",color(i));
                })
                .on("drag",function(d){
                    //拖曳中对象变为黄色
                    d3.select(this).style("fill","yellow");
                });
```

力导向图中的节点都有一个 fixed 属性，当值为 true 时，即可将该节点设置为固定。拖曳行为保存在变量 drag 中，接下来，对被拖曳的元素使用：

```
.call(drag)
```

即可。结果如图 10-11 所示，图中所有的节点都被拖曳过，因此都固定住了，5 号节点正在被拖曳，故颜色变成了黄色。

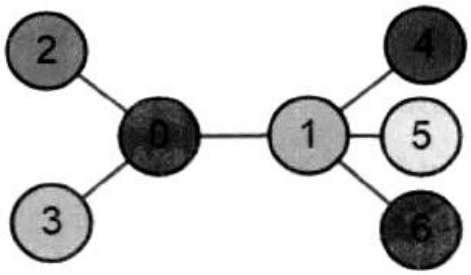


图 10-11　所有的节点都被拖曳过，5 号节点正在被拖曳

10.4　弦图

弦图（Chord Diagram），用于表示一组元素之间的联系。

弦图各区域的含义如图 10-12 所示。弦图分为两部分：外部的**节点**和内部的**弦**。源数据是一个方块矩阵（行数和列数相等，$N \times N$），矩阵中的 A、B、C 等在弦图的外圆上用相互分隔的几段弧来表示，即**节点**。**节点**的长度为该元素所在行的总和，例如 A 弧的长度为(A, A)、(A, B)、(A, C)、(A, D)、(A, E)的和。

从一个元素到另一个元素绘制弧，表示两个元素相关，弧的宽度表示权重，这就是**弦**。例如，C 与 D 的关系由两组值表示：(C, D)和(D, C)。在 C 与 D 之间的弦，与 C 节点相接的弧长是(C, D)的值，与 D 节点相接的弧长是(D, C)的值。

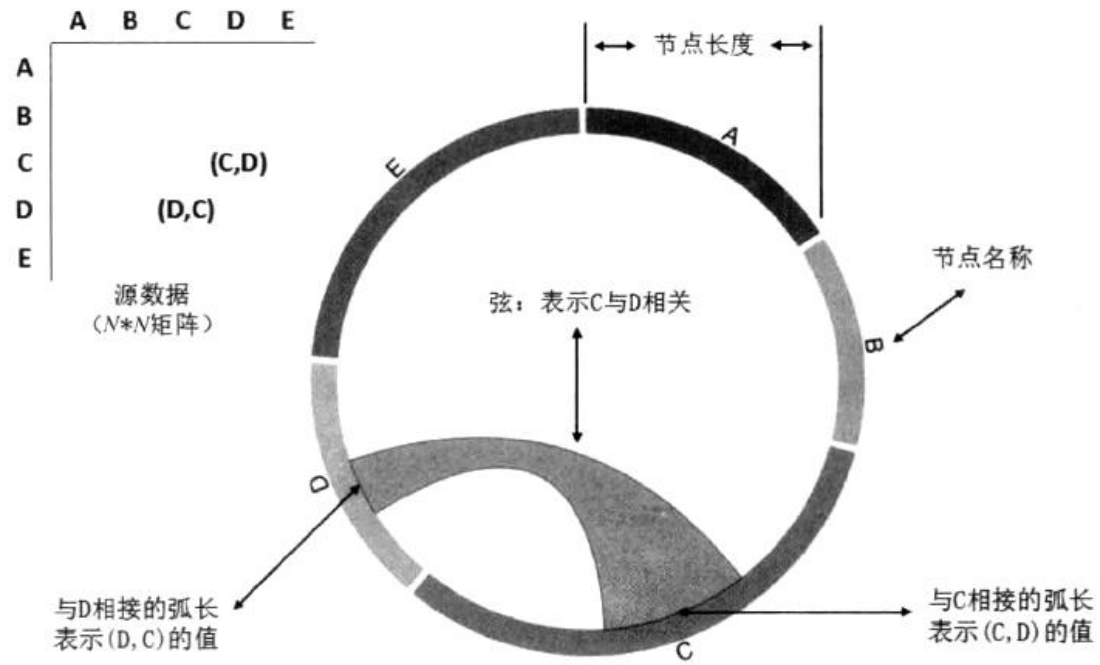


图 10-12　弦图各部分的含义

矩阵里还有(A, A)、(B, B)等值，这些值反映到弦图里，是一段弧从某元素节点出来，最后又回到此节点，如图 10-13 所示。

弦图布局（Chord Layout），能将一个方块矩阵转换成可用于绘制**节点**和**弦**的数据。下面介绍与弦图相关的方法。

- d3.layout.**chord**()

创建一个弦图布局。

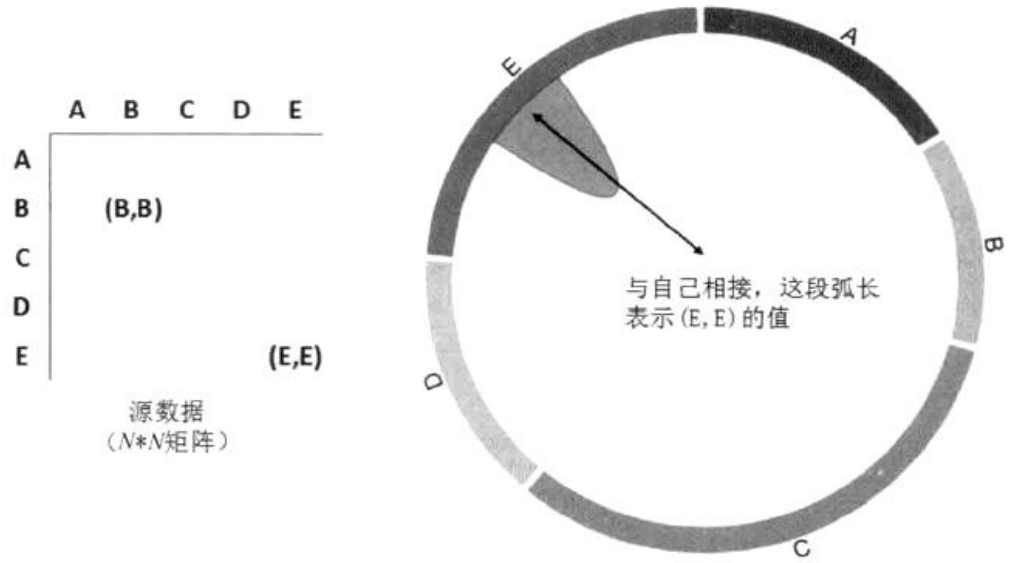


图 10-13　元素自身的弦

- chord.**matrix**([matrix])

设定或获取矩阵，必须是方块矩阵，即行列数相等。

- chord.**padding**([padding])

设定或获取元素节点之间的间距，默认为 0。

- chord.**sortGroups**([comparator])

对节点进行排序。comparator 是比较函数，如 d3.ascending 和 d3.descending。

- chord.**sortSubgroups**([comparator])

对各节点的所在行的数据进行排序，comparator 为比较函数。

- chord.**sortChords**([comparator])

对弦进行排序。

- chord.**chords**()

返回弦数组。

- chord.**groups**()

返回节点数组。

下面制作一个弦图，要求是：有五大洲的人口组成数据，以弦图的形式做可视化。

1．确定初始数据

初始数据如下：

```
var continent = [ "亚洲" , "欧洲" , "非洲" , "美洲" , "大洋洲" ];

var population = [
[ 9000,  870   , 3000  , 1000 , 5200 ],
```

```
    [ 3400,  8000  , 2300  , 4922 , 374  ],
    [ 2000,  2000  , 7700  , 4881 , 1050 ],
    [ 3000,  8012  , 5531  , 500  , 400  ],
    [ 3540,  4310  , 1500  , 1900 , 300 ]
];
```

continent 是各洲名称，population 一个方块矩阵，矩阵里的数字表示人口的来源。例如，

	亚洲	美洲
亚洲	9000	1000
美洲	7000	500

里面的数字表示：
亚洲的人口，有 9000 是本地人，有 1000 人是来自美洲的移民，总人口为 9000 + 1000。
美洲的人口，有 500 是本地人，有 7000 人是来自亚洲的移民，总人口为 500 + 7000。
现在要将这样的数据进行可视化。

2. 转换数据

创建一个弦图布局，将各节点之间的间隔设定为 0.03，节点所在行的数据要排序。

```
var chord = d3.layout.chord()
              .padding(0.03)
              .sortSubgroups(d3.ascending)
              .matrix(population);

console.log(chord.groups());
console.log(chord.chords());
```

chord.groups()返回节点数组，chord.chords()返回弦数组。转换之后，原来的数据 population 是不变的。图 10-14 是**节点数组**的输出结果，每一个节点都包含有如下属性。

- index：行索引号，行号从 0 开始。
- startAngle：弧的起始角度。
- endAngle：弧的终止角度。
- value：该行所有数值的和。

看到 startAngle 和 endAngle，要立即想到用弧生成器（d3.svg.arc）来画弧。

图 10-15 是**弦数组**的输出结果。每一段弦都有变量 source 和 target，分别代表起始弧和目标弧。source 和 target 对象里，包含有如下属性。

- index：行号，i。
- subindex：列号，j。
- startAngle：弧的起始角度。

- endAngle：弧的终止角度。
- value：矩阵单元(i, j)的值。

弦的绘制要使用弦生成器（d3.svg.chord），而弦生成器所需的数据格式，恰巧就是弦图布局转换后的数据格式。

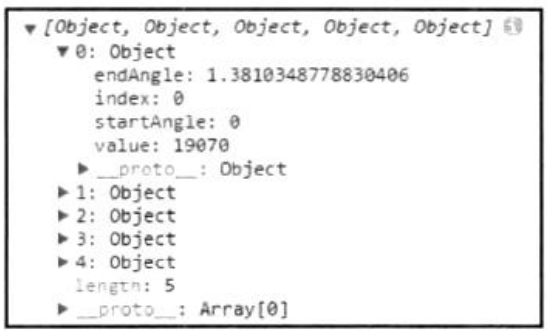


图 10-14　节点数组

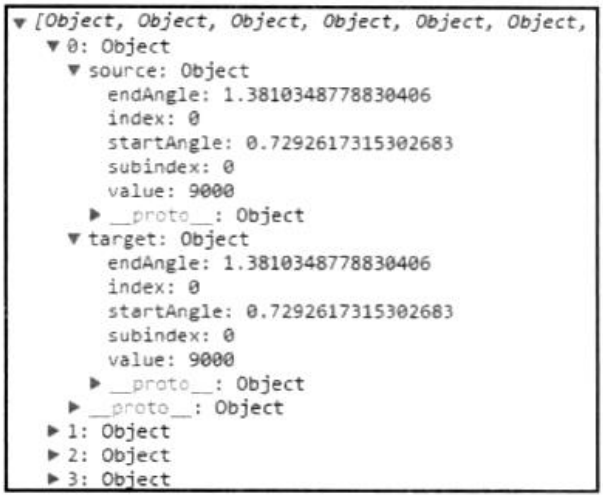


图 10-15　弦数组

3．绘制

有了转换后的数据，可以开始绘图了。首先，在 svg 中添加几个分组元素<g>，分别用来装节点和弦。可使用如下结构：

```
<svg>
    <g>                 //包含节点和弦
        <g> </g>        //节点
        <g> </g>        //弦
    </g>
</svg>
```

在<svg>里添加一个<g>元素，用于设置平移的属性（确定弦图的中心）。然后，在<g>元素里再添加两个<g>，一个用来装节点，一个用来装弦。代码如下：

```
//弦图的<g>元素
var gChord = svg.append("g")
                .attr("transform",
                      "translate(" + width/2 + "," + height/2 + ")");

//节点的<g>元素
var gOuter = gChord.append("g");

//弦的<g>元素
```

```
var gInner = gChord.append("g");
```

弦图的<g>元素被设定了 transform 属性，平移到 svg 的中心（width/2, height/2）。弦图的所有节点添加到 gOuter 里，所有弦添加到 gInner 里。

```
创建一个弧生成器，并为其设定内半径和外半径。
//颜色比例尺
var color20 = d3.scale.category20();

//绘制节点
var innerRadius = width/2 * 0.7;
   var outerRadius = innerRadius * 1.1;

//弧生成器
var arcOuter =  d3.svg.arc()
                      .innerRadius(innerRadius)
                      .outerRadius(outerRadius);
```

在 gOuter 里添加路径元素<path>，路径值使用弧生成器 arcOuter 计算，代码如下：

```
gOuter.selectAll(".outerPath")
            .data(chord.groups())          //绑定节点数组
            .enter()
            .append("path")
            .attr("class","outerPath")
            .style("fill", function(d) { return color20(d.index); })
            .attr("d", arcOuter );         //使用弧生成器
```

结果如图 10-16 所示，5 个节点被绘制出来了。

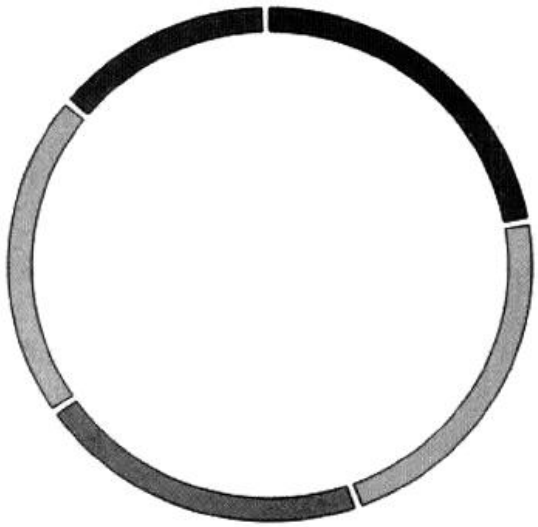


图 10-16　节点的绘制

还需要为节点添加文字<text>，也放到 gOuter 的分组元素里。在添加文字元素时，要对文字进行平移、旋转等，以使其显示到适当的位置。在 SVG 中，平移旋转等是由 transform 属性来完成的，transform 的值可以是：

- **translate(50, 100)**：*x* 方向平移 50，*y* 方向平移 100。
- **rotate(180)**：旋转 180°，参数是角度。

代码如下：

```
gOuter.selectAll(".outerText")
    .data(chord.groups())
    .enter()
    .append("text")
    .each( function(d,i) {       //为被绑定的数据添加变量
        d.angle = (d.startAngle + d.endAngle)/2;    //弧的中心角度
        d.name = continent[i]; //节点名称
    })
    .attr("class","outerText")
    .attr("dy",".35em")
    .attr("transform", function(d){     //设定平移属性的值

        //先旋转d.angle°（将该值转换为角度）
         var result = "rotate(" + ( d.angle * 180 / Math.PI ) + ")";

        //平移到外半径之外
        result += "translate(0,"+ -1.0 * ( outerRadius + 10 ) +")" ;

        //对位于弦图下方的文字，翻转180°，为了防止其是倒着的
        if( d.angle > Math.PI * 3 / 4 &&  d.angle < Math.PI * 5 / 4 )
            result += "rotate(180)";

        return result;
    })
    .text(function(d){
        return d.name;
    });
```

each()是对选择集中的元素都调用其参数的 function(d,i)函数，在这里是为了给被绑定的数据添加两个变量。添加文字后，结果如图 10-17 所示。

还剩最后一步，只要添加上内部的弦，就大功告成了。创建一个弦生成器，为其设置半径。在 gInner 的分组元素里添加对应数目的路径元素<path>，再使用弦生成器来计算弦的路径。

```
var arcInner =  d3.svg.chord()
                    .radius(innerRadius);
```

```
gInner.selectAll(".innerPath")
        .data(chord.chords())          //绑定弦数组
        .enter()
        .append("path")
        .attr("class","innerPath")
        .attr("d", arcInner )          //使用弦生成器
        .style("fill", function(d) {
    return color20(d.source.index);
 });
```

结果如图 10-18 所示，一个弦图就绘制完成了。

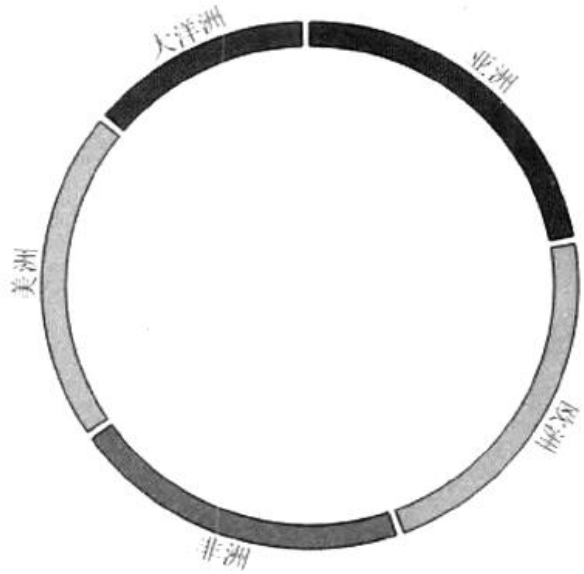


图 10-17　添加节点文字

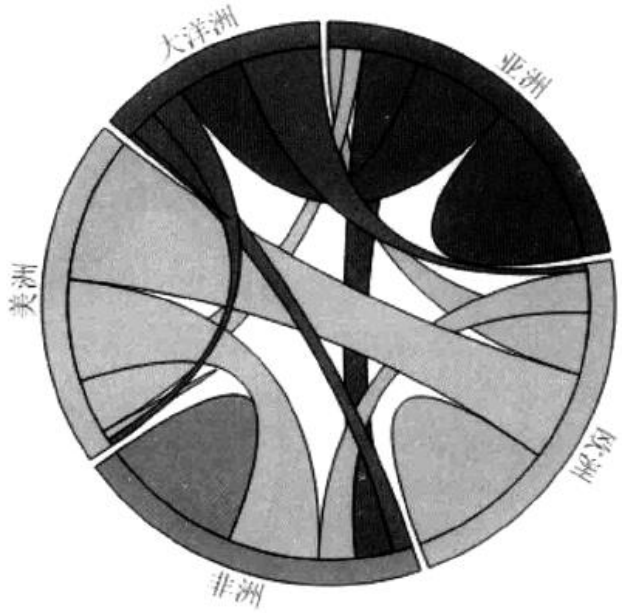


图 10-18　添加弦

弦图的内部比较凌乱，有时无法看清楚与每一个节点相连接的弦。可以添加一些交互式的操作解决此问题。例如，当鼠标移到某节点上时，只有与该节点相接的弦才能显示，其他的都要隐身。要做到这点很容易：

```
gOuter.selectAll(".outerPath")
      .on("mouseover",fade(0.0))       //鼠标放到节点上
      .on("mouseout",fade(1.0));       //鼠标从节点上移开
```

接下来要写一个 fade()函数，参数是不透明度（1.0 为完全不透明，0.0 为完全透明）。fade()函数要返回一个无名函数，作为监听器。其实现为：

```
function fade(opacity){
    //返回一个function(g, i)
    return function(g,i){
```

```
    gInner.selectAll(".innerPath") //选择所有的弦
        .filter( function(d) {  //过滤器
            //没有连接到鼠标所在节点的弦才能通过
            return d.source.index != i && d.target.index != i;
        })
        .transition()   //过渡
        .style("opacity", opacity);//透明度
    }
}
```

这样，当鼠标移到节点上时，其他的弦会缓缓地隐身，当移开时又会缓缓地出现，结果如图 10-19 所示。

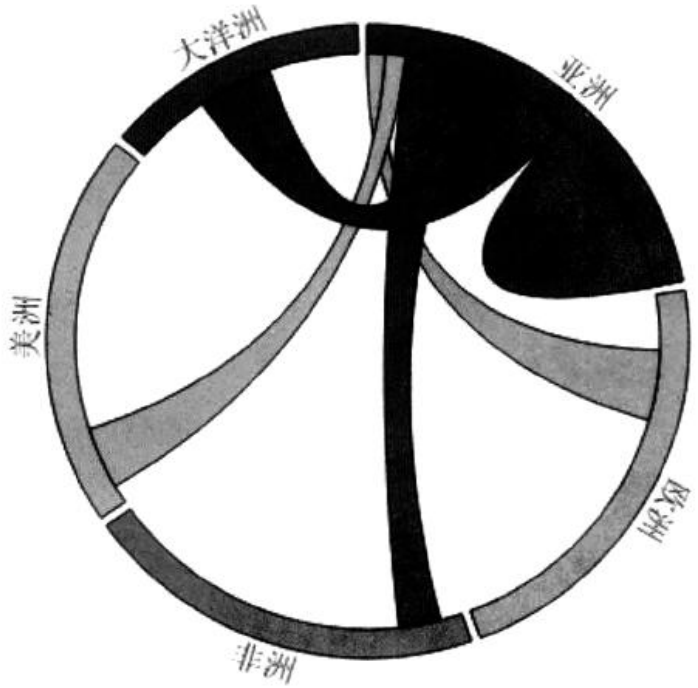


图 10-19　鼠标移到“亚洲”节点上

10.5　树状图

树状图（Tree Diagram），在区域内按规则布置**节点（node）**，再用**连线（link）**将节点连接起来，最终的图表形似一棵树的枝干。

树状图布局（Tree Layout）的相关方法介绍如下。

- d3.layout.**tree**()

创建一个树状图布局。

- tree.**size**([size])

设定或获取尺寸，参数 size 是只有两个元素的数组，分别表示宽和高，例如[x, y]。

- tree.**nodeSize**([nodeSize])

设定或获取各节点的大小，以数组的形式：[x, y]。

- tree.**value**([value])

设定或获取值访问器，默认为 null。如果使用了此函数，各节点会多一个 value 属性。

- tree.**children**([children])

设定或获取子节点访问器。默认情况下，认为当前节点的变量 children 是子节点。

- tree.**sort**([comparator])

设定或获取排序的比较器。

- tree.**separation**([separation])

设定或获取相邻节点之间的间隔。

- tree.**nodes**(root)

根据 root 进行计算，获取节点数组。

- tree.**links**(nodes)

根据 nodes 进行计算，获取连线数组。

下面制作一个树状图，要求如下。

有表示“中国—省份—城市”的数据，表示各节点之间的嵌套关系。用树状图布局将其可视化。

1. 确定初始数据

数据写在一个 JSON 文件里，文件名为 city.json。节点里有 name 和 children 属性，name 是该节点的名称，children 是子节点。如果没有 children，表明该节点是叶子节点。以下是文件内容，限于篇幅，这里只给出其中一部分：

```
{
"name":"中国",
"children":
[
   {
       "name":"浙江" ,
       "children":
       [
           {"name":"杭州" },
           {"name":"宁波" },
           {"name":"温州" },
           {"name":"绍兴" }
```

```
        ]
    },
    {
        "name":"广西" ,
        "children":
        [
            {
                "name":"桂林",
                "children":
                [
                    {"name":"秀峰区"},
                    {"name":"叠彩区"},
                    {"name":"象山区"},
                    {"name":"七星区"}
                    ]
                },
                {"name":"南宁"},
                {"name":"柳州"},
                {"name":"防城港"}
            ]
        },
    ]
    }
```

2. 转换数据

创建一个树状图布局，并设定布局的尺寸和节点之间的间隔。

```
var tree = d3.layout.tree()
                .size([width, height-200])
                .separation(function(a, b) {
                    return (a.parent == b.parent ? 1 : 2) ;
                });
```

width 和 height 是 SVG 区域的宽和高。seperation()的参数是 function(a, b)，a 和 b 表示两个相邻节点，这段代码的意思是：如果 a 和 b 节点的父节点是相同的，a 和 b 的间隔是 1，否则是 2，其实它也是 seperation()的默认值。

接下来，使用 d3.json()来读取数据。

```
d3.json("city.json", function(error, root) {
    var nodes = tree.nodes(root);
    var links = tree.links(nodes);

    console.log(nodes);
```

```
    console.log(links);
}
```

读取的文件内容，保存在变量 root 里，使用 tree.nodes(root)计算得到节点数组，保存在变量 nodes 里，使用 tree.links(nodes)得到连线数组，保存在 links 里。

输出 nodes 的结果如图 10-20 所示，每一个节点对象都包含如下内容。

- parent：父节点。
- children：子节点。
- depth：节点的深度。
- x：节点的 x 坐标。
- y：节点的 y 坐标。

输出 links 的结果如图 10-21 所示，每一个连线对象都包含如下内容

- source：前端节点。
- target：后端节点。

图 10-20　节点数组

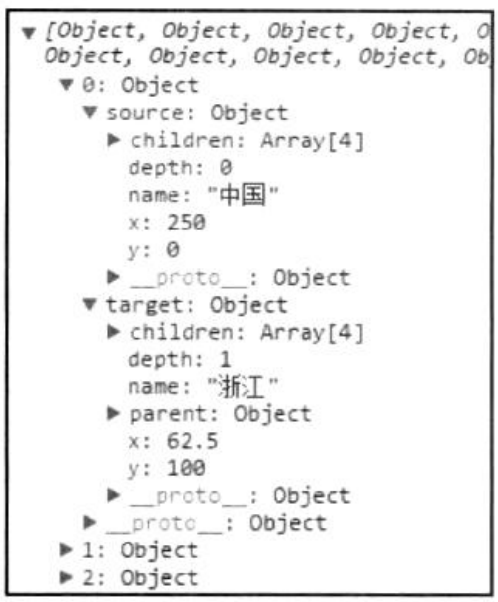


图 10-21　连线数组

3．绘制

首先，确定用什么图形元素来绘图：节点用<circle>，节点的名称用<text>，连线用<path>。其中，连线画成曲线，使用对角线生成器来做。

创建一个对角线生成器，并设定其投影。树状图布局默认是将各节点坐标安排成一棵由上到下逐渐展开的树，现在要将其绘制成由左至右的：

```
var diagonal = d3.svg.diagonal()
            .projection(function(d) { return [d.y, d.x]; });
```

projection()作用于每一个节点，也就是说其 x 坐标和 y 坐标将被对调，这样就能够制成一个横向的树状图。

其次，在 function(error, root)的函数体里写代码，添加树状图的连线：

```
var link = gTree.selectAll(".link")
                .data(links)
                .enter()
                .append("path")
            .attr("class", "link")
            .attr("d", diagonal);  //使用对角线生成器
```

结果如图 10-22 所示。要注意，在向<svg>添加元素时，如果两个元素在同一位置，那么后添加的元素会挡住之前添加的元素。因此，先绘制连线还绘制先绘制节点有讲究的，要根据需求决定。

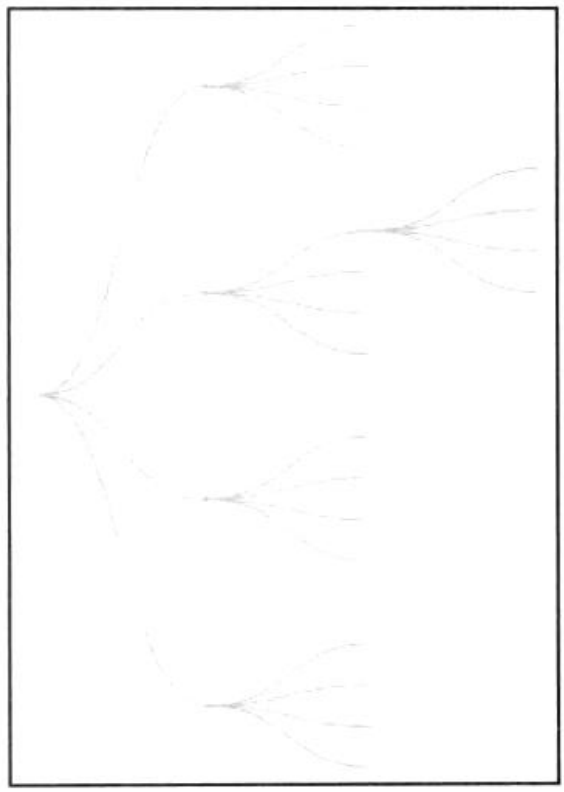


图 10-22　绘制连线

再次，为添加节点做准备。由于每一个节点都包含<circle>和<text>元素，可以将这两个元素放到<g>里。下面添加足够数量的<g>，并平移到指定位置：

```
var node = gTree.selectAll(".node")
            .data(nodes)
            .enter()
```

```
            .append("g")
            .attr("class", "node")
            .attr("transform", function(d) {
                return "translate(" + d.y + "," + d.x + ")";
            });
```

这段代码中，要注意与对角线生成器的投影函数保持一致，把 d.y 作为 x 方向平移量，把 d.x 作为 y 方向的平移量。因为本例中是要生成一个从左到右逐渐展开的树。

然后，分别向分组元素<g>中添加圆形元素<circle>和文字元素<text>：

```
node.append("circle")
    .attr("r", 4.5);

node.append("text")
    .attr("dx", function(d) { return d.children ? -8 : 8; })
    .attr("dy", 3)
    .style("text-anchor", function(d) {
       return d.children ? "end" : "start";
    })
    .text(function(d) { return d.name; });
```

结果如图 10-23 所示。

图 10-23　树状图结果

有两句代码可能较难理解，分别是给<text>设置属性 dx 和样式 text-anchor 的部分。该部分可解释为：**如果某节点是叶子节点，则节点文字在圆圈的右边，否则在左边**。如图 10-24 所示，节点“桂林”的 text-anchor 为 end，并且向左（-8）平移，节点“哈尔滨”的 text-anchor 为 start，并向右（8）平移。关于 text-anchor 的内容，在第 4.7.1 节时叙述过。

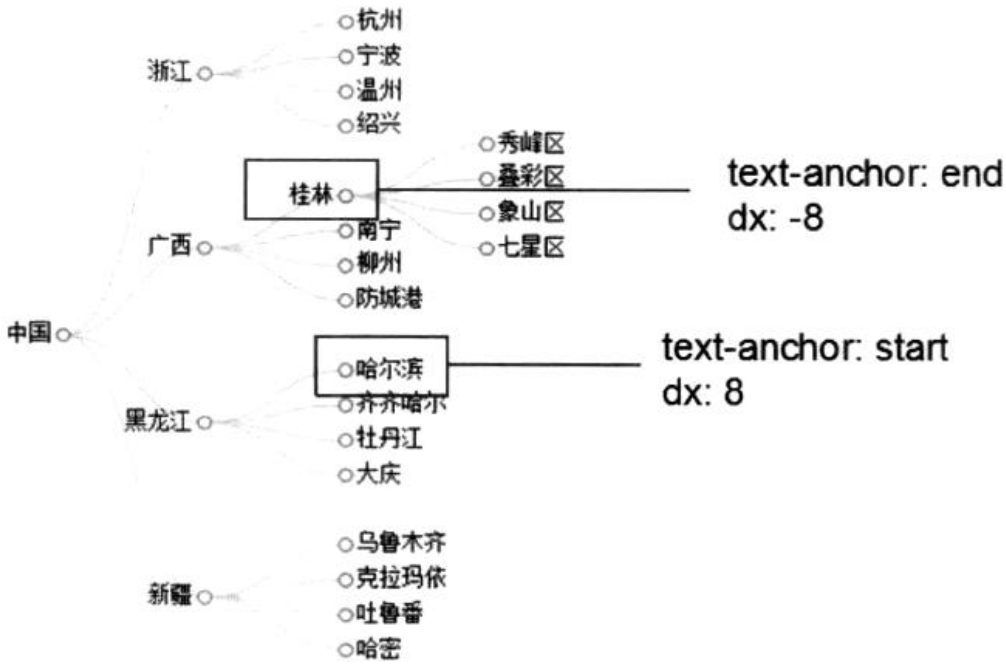


图 10-24　树状图结果

10.6　集群图

集群图（Cluster Diagram），与树状图很相似，其布局中各方法的意义和用法与树状图也完全一样。与树状图相比，集群图只有一点不同：

所有叶子节点都被放置在相同的深度。

什么意思？将上一节树状图的代码稍做修改，改成使用集群图的布局：

```
var cluster = d3.layout.cluster()
                .size([width, height-200])
                .separation(function(a, b) {
                    return (a.parent == b.parent ? 1 : 2) ;
                });
```

布局被保存在变量 cluster 中。然后，将上一节的代码中的 tree 都改为 cluster，就可变成集群图，结果如图 10-25 所示。对比图 10-23，可以看出，树状图的叶子节点不一定都在同一深度

位置上，而集群图里无论中间进过几层，所有的叶子节点都在同一深度位置。

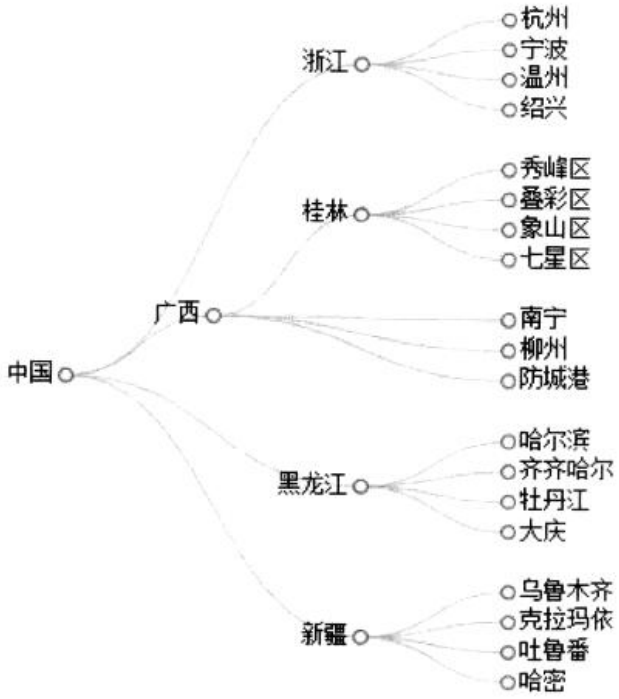


图 10-25　集群图结果

集群图和树状图的布局都是通过 size()来设定尺寸的，如果使用宽和高：

```
size([width, height])
```

那么在节点数组中，x 表示宽度，y 表示高度。如果使用角度和半径：

```
size([angle, radius])
```

那么节点的 x 表示角度，y 表示半径，如图 10-26 所示。
下面制作一个圆形的集群图。

1. 确定初始数据

还是使用表示城市包含关系的数据，但是向其中添加更多的省名和城市名。

2. 转换数据

首先，创建集群图的布局：

```
var cluster = d3.layout.cluster()
            .size([360, width/2 - 100])
            .separation(function(a, b) {
                return (a.parent == b.parent ? 1 : 2) / a.depth;
            });
```

size()的第一个参数为360，表示所有节点在360°范围内分布，半径为300（width的值为800）。在separation()里，后面除以了一个a.depth，能使节点之间的间隔成比例地减少，在圆形集群图和树状图中常用。

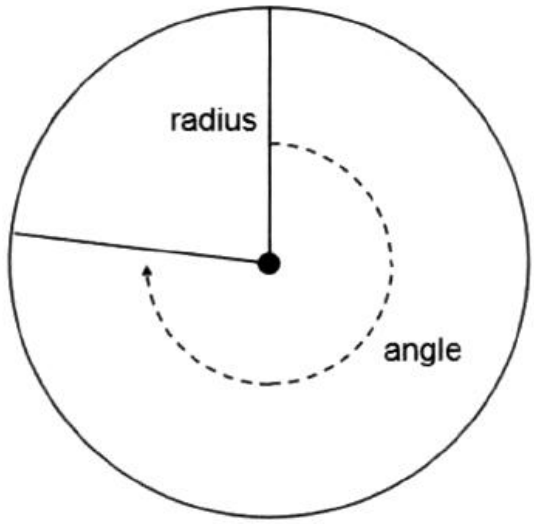


图10-26　size([angle, radius])参数的意义

转换数据的方法与树状图完全一样，转换后的节点数组和连线数组中各变量的名称也与树状图一致。只是在这里我们要绘制圆形的，因此，各节点对象中，x代表角度，y代表半径。

3. 绘制

创建一个放射式对角线生成器，用法与普通的对角线生成器类似：

```
var diagonal = d3.svg.diagonal.radial()
    .projection(function(d) {
        var radius = d.y,          //d.y是半径
            angle  = d.x / 180 * Math.PI;       //d.x是角度，
                                                //将角度换算成弧度
        return [ radius , angle ];
    });
```

投影函数projection中，d.y是半径，d.x是角度，将d.x换算成弧度，最终返回[radius, angle]。然后，可以凭此计算放射性对角线的路径值了，代码与树状图一样：

```
var link = gCluster.selectAll(".link")
            .data(links)
            .enter()
            .append("path")
            .attr("class", "link")
            .attr("d", diagonal);      //使用放射式对角线生成器
```

连线绘制好后，再绘制节点：

```
var node = gCluster.selectAll(".node")
              .data(nodes)
              .enter()
              .append("g")
              .attr("class", "node")
              .attr("transform", function(d) {
    return "rotate(" + (d.x- 90) + ")translate(" + d.y + ")";
        });
```

这里尤其要注意节点的定位方式：先旋转(d.x–90)，再平移 d.y。我们知道 d.x 是角度，那么为何要减去 90。这是因为 rotate 是以水平方向 x 轴的正方向为旋转起始点的，而布局计算的 d.x 是以 y 轴的负方向为旋转起点，如图 10-27 所示。二者的旋转都是顺时针的（正为顺时针，负为逆时针）。因此，二者相差 90°，用 rotate 定位顶点的角度时要用(d.x–90)°。在 translate 平移方面，平移对应的半径大小即可，即 d.y。

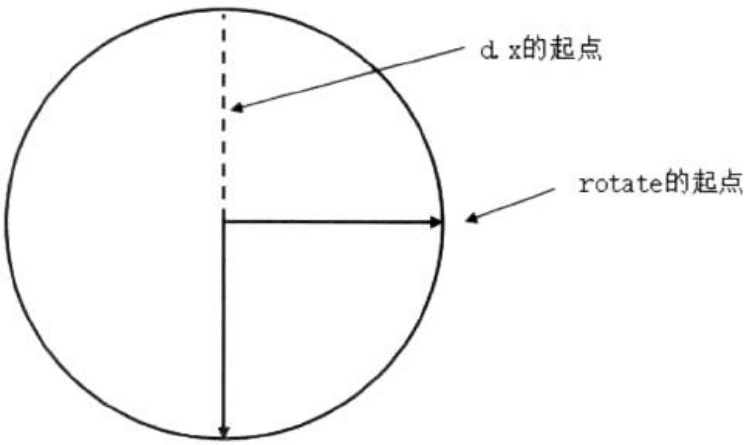


图 10-27　旋转的起点

然后可以添加圆圈和文字，要注意文字的位置。

```
node.append("circle")
    .attr("r", 4.5);

node.append("text")
    .attr("transform",function(d){
        return d.x < 180 ? "translate(8)" : "rotate(180)translate(-8)";
    })
    .attr("dy", ".3em")
    .style("text-anchor", function(d) {
        return d.x < 180 ? "start" : "end";
})
```

```
.text(function(d) { return d.name; });
```

粗体字部分要留意，如果文字的角度超过 180°，要将文字旋转 180°，这是为了防止一部分文字是倒的。

绘制结果如图 10-28 所示，所有叶子节点都在同一半径位置。

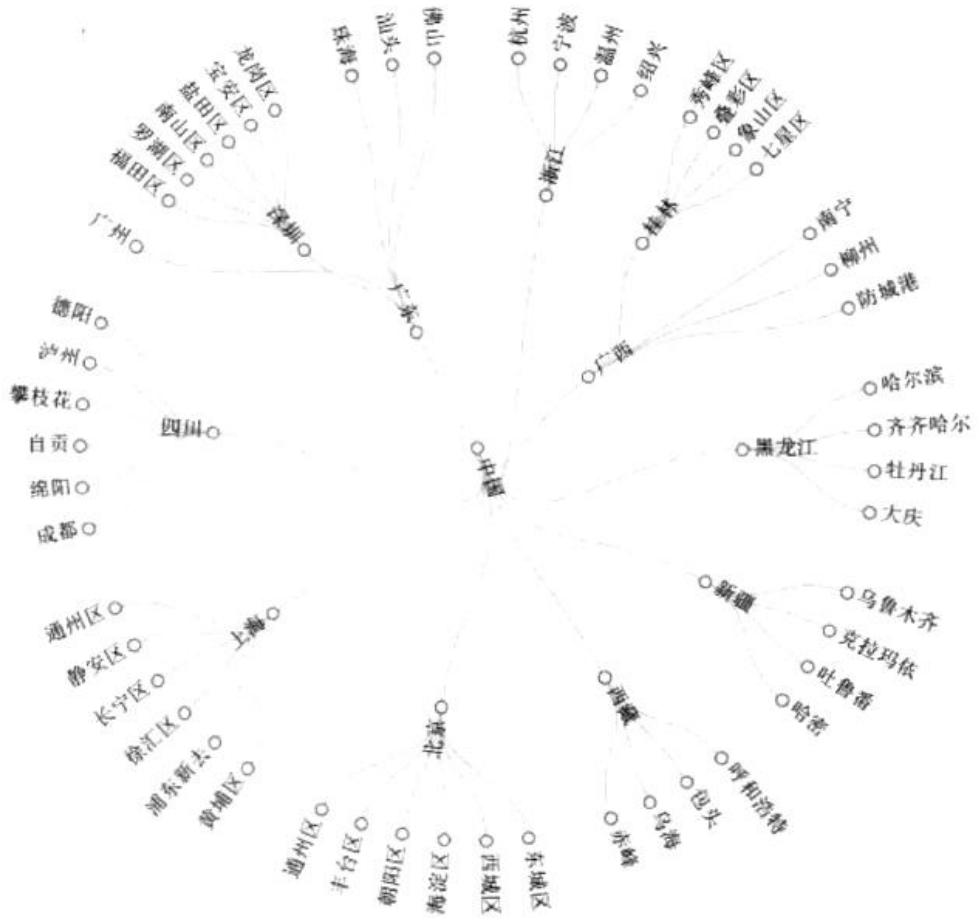


图 10-28　圆形集群图

10.7　捆图

捆图布局（Bundle Layout）是比较奇特的一个布局，只有两个方法。

- d3.layout.**bundle**()

创建一个捆图布局。

- **bundle**(links)

根据数组 links 的 source 和 target，计算路径。

捆图的方法之所以少，是因为它需要与其他层级布局一起使用。所谓层级布局，是指采用嵌套结构（父子节点关系）来描述节点信息的布局。由层级布局扩展而来的布局是：集群图、打包图、分区图、树状图、矩阵树图。其中，最常见的是与集群图一起使用，使用集群图布局计算节点的位置，再用捆图布局计算连线路径。也就是说，捆图布局只干一件事：

计算连线的路径。

下面举一个例子：

中国的高铁已经在很多城市开通，如北京到上海，北京到桂林等。现制作一个捆图来表示经过哪一座城市的高铁最密集。

1. 确定初始数据

定义节点数组，有九座城市：

```
var cities = {
        name: "",
        children:[
            {name: "北京"},{name: "上海"},{name: "杭州"},
            {name: "广州"},{name: "桂林"},{name: "昆明"},
            {name: "成都"},{name: "西安"},{name: "太原"}
        ]
    };
```

这九座城市所属的节点有一个公共的父节点，父节点名称为空，稍后并不绘制此父节点。然后，定义连线数组，即连接各城市高铁的线路：

```
var railway = [
            {source: "北京", target: "上海"},
            {source: "北京", target: "广州"},
            {source: "北京", target: "杭州"},
            {source: "北京", target: "西安"},
            {source: "北京", target: "成都"},
            {source: "北京", target: "太原"},
            {source: "北京", target: "桂林"},
            {source: "北京", target: "昆明"},
            {source: "北京", target: "成都"},
            {source: "上海", target: "杭州"},
            {source: "昆明", target: "成都"},
            {source: "西安", target: "太原"}
]; //（该数据为假设，并没有经过调查）
```

source 和 target 分别表示高铁的两端。

2. 转换数据

前面提到，捆图布局要与其他布局联合使用。因此，首先分别创建一个集群图布局和一个

捆图布局：

```
var cluster = d3.layout.cluster()          //集群图布局
              .size([360, width/2 - 50])
              .separation(function(a, b) {
                  return (a.parent == b.parent ? 1 : 2) / a.depth;
              });

var bundle = d3.layout.bundle();           //捆图布局
```

由集群图布局的方法 separation 可以看出，节点将被分布成圆形。捆图布局没有参数可以设置，只创建即可，保存在变量 bundle 中。

先使用集群图布局计算节点：

```
var nodes = cluster.nodes(cities);
console.log(nodes);
```

节点数组保存在变量 nodes 中，输出此数组。如图 10-29 所示，第一个节点有 9 个子节点，其他的节点都有且只有一个父节点，没有子节点。这就是**捆图**要使用的节点数组，但是却是用集群图布局计算而来的。

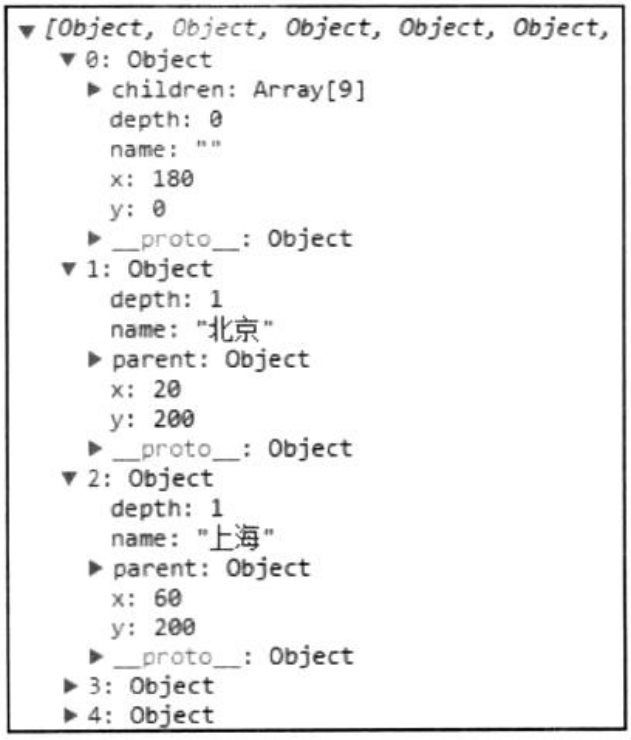


图 10-29　捆图的节点数组

下一步是重点，要使用数组 railway。由于 railway 中存储的 source 和 target 都只有城市名称，因此要将它换成 nodes 中的节点对象。写一个函数，按城市名将 railway 中的 source 和 target

替换成节点对象：

```
function map( nodes, links ){
    var hash = [];
    for(var i = 0; i < nodes.length; i++){
        hash[nodes[i].name] = nodes[i];
    }
    var resultLinks = [];
    for(var i = 0; i < links.length; i++){
        resultLinks.push({      source: hash[ links[i].source ],
                                target: hash[ links[i].target ]
                            });
    }
    return resultLinks;
}
```

使用该函数返回的数组，即可作为捆图布局 bundle 的参数使用：

```
var oLinks = map(nodes, railway); //将连线两端换成节点对象
console.log(oLinks);

var links = bundle(oLinks);       //调用捆图布局，转换数据
console.log(links);
```

map()返回的结果保存在 oLinks，bundle()返回的结果保存在 links 中。在控制台的输出结果分别如图 10-30 和图 10-31 所示，这是连线数组转换前后的情况。

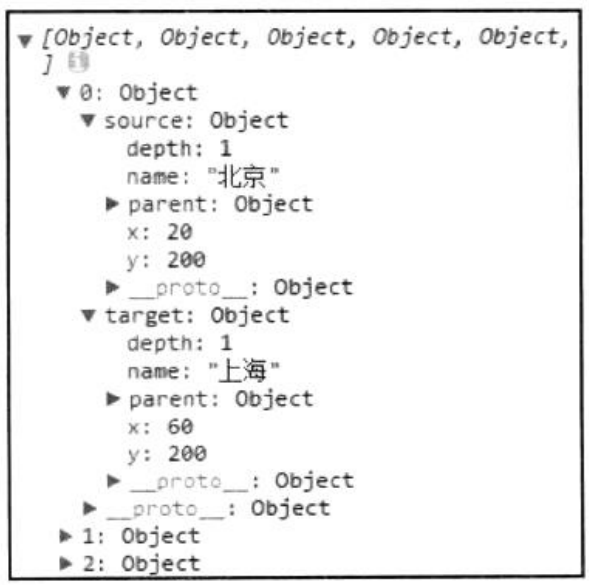


图 10-30　捆图的连线数组

如图 10-30 所示，转换前，连线数组的每一项都只有两个变量：source 和 target，内容是节

点对象。对于第一个连线，是从“北京”到“上海”。如图 10-31 所示，转换之后，source 和 target 不见了，取而代之的是 0、1、2，变成了一个数组。很明显，该数组的第 0 项和 source 的内容一样，第 2 项和 target 的内容一样，但中间多出了一项（图 10-31 的红框内）。多出的这一项是根据 source 和 target 公共的父节点计算出来的。于是，该数组表示了一条路径。

其实，捆图布局根据各连线的 source 和 target 为我们计算了一条条连线路径，据此可以把捆图布局的作用简单地解释为：**根据节点数组生成一系列曲线，这些曲线能更好地表示“经过哪座城市的高铁最多”。**

图 10-31　捆图的连线数组

3. 绘制

经捆图布局转换后的数据很适合用 d3.svg.line()和 d3.svg.line.radial()来绘制，前者是线段生成器，后者是放射式线段生成器。第 6.2 节中提到，在 line.interpolate()所预定义的插值模式中，有一种就叫作 bundle，正是为捆图准备的。由于本例中用集群图布局计算节点数组使用的是圆形，故要用放射式的线段生成器。

首先，创建一个放射式线段生成器：

```
var line = d3.svg.line.radial()
             .interpolate("bundle")
             .tension(.85)
             .radius(function(d) { return d.y; })
             .angle(function(d) { return d.x / 180 * Math.PI; });
```

其次，添加一个分组元素<g>，与捆图相关的元素都放在此分组里：

```
gBundle = svg.append("g")
            .attr("transform",
                "translate(" + (width/2) + "," + (height/2) + ")");

var color = d3.scale.category20c();    //颜色比例尺
```

再次，在 gBundle 中添加连线路径：

```
var link = gBundle.selectAll(".link")
          .data(links)
          .enter()
          .append("path")
          .attr("class", "link")
          .attr("d", line); //使用线段生成器
```

对于连线的 CSS 样式，添加透明度能够在连线汇聚处更能显示出“捆”的效果。例如样式设定为：

```
.link {
    fill: none;
    stroke: black;
    stroke-opacity: .5;
    stroke-width: 8px;
}
```

连线的绘制结果如图 10-32 所示。

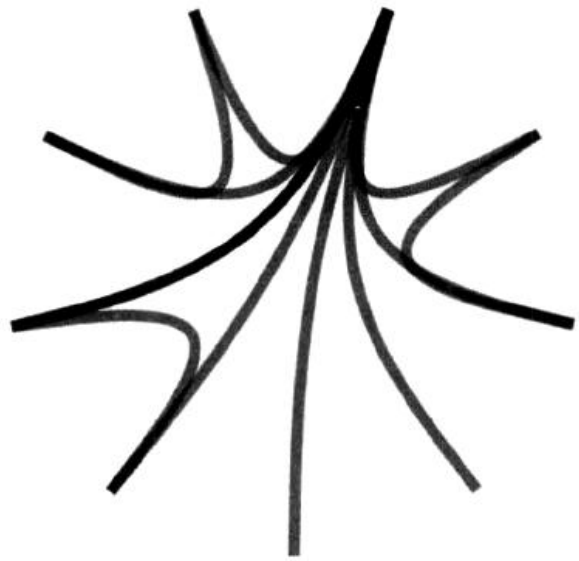


图 10-32　绘制捆图的连线

然后，向图中添加节点。节点用一个带名称的圆来表示：

```
var node = gBundle.selectAll(".node")
           .data( nodes.filter(function(d) { return !d.children; }) )
           .enter()
           .append("g")
           .attr("class", "node")
           .attr("transform", function(d) {
           return "rotate(" + (d.x- 90) + ")translate("
                  + d.y + ")" + "rotate("+ (90 - d.x) +")";
           });
```

要注意，被绑定的数组是经过过滤后的 nodes 数组。此处的 filter 是 JavaScript 数组对象自身的方法，粗体字部分的意思是：**只绑定没有子节点的节点**。也就是说九座城市的公共父节点不绘制。

最后，在该分组元素<g>中分别加入<circle>和<text>。

```
node.append("circle")
           .attr("r", 20)
           .style("fill",function(d,i){ return color(i); });

   node.append("text")
           .attr("dy",".2em")
           .style("text-anchor", "middle")
           .text(function(d) { return d.name; });
```

结果如图 10-33 所示。由于经过北京的高铁线路最多，连线在北京的圆圈处最密集，就好像将很多条绳子“捆”在这里一样。

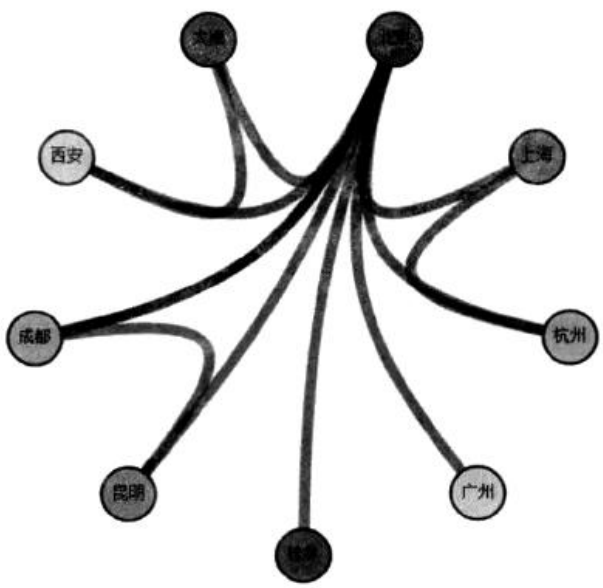


图 10-33　捆图的最终结果

10.8　打包图

打包图（Pack Diagram），是**层级布局**的一个扩展，可以用来表示包含于被包含的关系。打包图使用**圆**来表示节点，**圆的半径**表示节点的大小，圆里套圆表示节点之间的父子关系。打包图布局与其他层级图布局的使用方法类似。

- d3.layout.**pack**()

创建一个打包图布局。

- pack.**size**([size])

设定布局的尺寸，参数 size 是只有两个元素的数组，分别表示宽和高，例如[x, y]。

- pack.**radius**([radius])

设定或获取叶子节点的半径。

- pack.**padding**([padding])

设定或获取节点的间距。

- pack.**children**([children])

设定或获取子节点访问器。默认是认为当前节点中的变量 children 里的子节点。

- pack.**value**([value])

设定或获取值访问器。

- pack.**sort**([comparator])

设定或获取节点排序的比较器。

- pack.**nodes**(root)

根据 root 进行计算，获取节点数组。

- pack.**links**(nodes)

根据 nodes 进行计算，获取连线数组。

下面制作一个例子，要求如下：对第 10.5 节树状图的数据，使用打包图进行可视化。

1．确定初始数据

与第 10.5 节树状图的数据一样。

2．转换数据

创建一个打包图布局，设定尺寸为[width, height]，即 SVG 画板的大小。叶子节点的半径设定为 30，节点间距设定为 5。在这里，节点半径和间距都被设定成常数，但是写成函数的形式也是可以的：

```
var pack = d3.layout.pack()
```

```
            .size([ width, height ])
            .radius(30)
            .padding(5);
```

之后，用 d3.json()读取数据，再用打包图布局 pack 来计算节点数组和连线数组：

```
d3.json("city.json", function(error, root) {
    var nodes = pack.nodes(root);          //计算节点数组
    var links = pack.links(nodes);         //计算连线数组

    console.log(nodes);
    console.log(links);
}
```

与树状图、集群图的代码十分相似。下面来看看节点数组和连线数组在控制台的输出结果。图 10-34 是节点数组的输出结果，可以看到，节点对象的属性中包含有如下内容。

- parent：父节点。
- children：子节点。
- value：节点的值，由 value 的访问器决定。
- depth：节点的深度。
- x：节点的 x 坐标。
- y：节点的 y 坐标。
- r：节点的半径。

图 10-35 展示了连线数组，每个连线对象包含有 source 和 target，分别表示连线的前端和后端。

图 10-34　打包图的节点数组

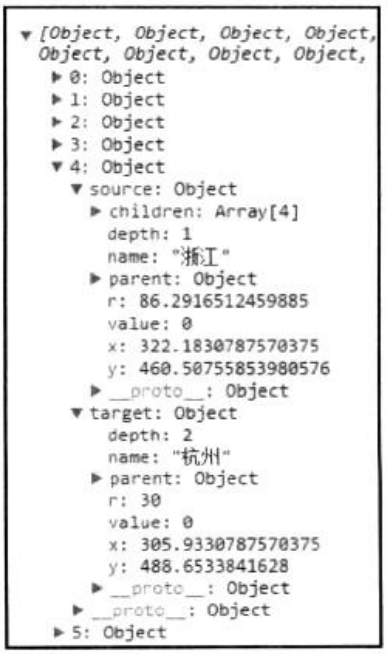


图 10-35　打包图的连线数组

无论用什么布局来制图，在转换数据后，都要先输出到控制台，观察数据的格式之后，再决定如何使用它来添加图形元素。

3. 绘制

首先，明确用什么图形元素来表示：节点圆<circle>，节点文字<text>。本例中，不绘制连线。

然后，添加足够数量的圆形元素<circle>：

```
svg.selectAll("circle")
        .data(nodes)
        .enter()
        .append("circle")
        .attr("class",function(d) {
            return d.children ? "node" : "leafnode";
        })
        .attr("cx",function(d){ return d.x; })
        .attr("cy",function(d){ return d.y; })
        .attr("r",function(d){ return d.r; });
```

这段代码的粗体字部分表示：如果该节点不是叶子节点，则应用 node 样式；如果是叶子节点，则应用 leafnode 样式。cx、cy、r 分别是圆的坐标和半径，由于绑定的数组为 nodes，对应使用节点的 d.x、d.y、d.r 即可。

最后，添加节点的文字元素<text>：

```
svg.selectAll("text")
        .data(nodes)
        .enter()
        .append("text")
        .attr("class","nodeText")
        .style("fill-opacity",function(d){
            return d.children ? 0 : 1;
        })
        .attr("x",function(d){ return d.x; })
        .attr("y",function(d){ return d.y; })
        .attr("dy",".3em")
        .text(function(d){ return d.name; });
```

如果节点不是叶子节点，则不显示文字（设定其透明度为 0）。如此，只有叶子节点会显示

文字。最终结果如图 10-36 所示。

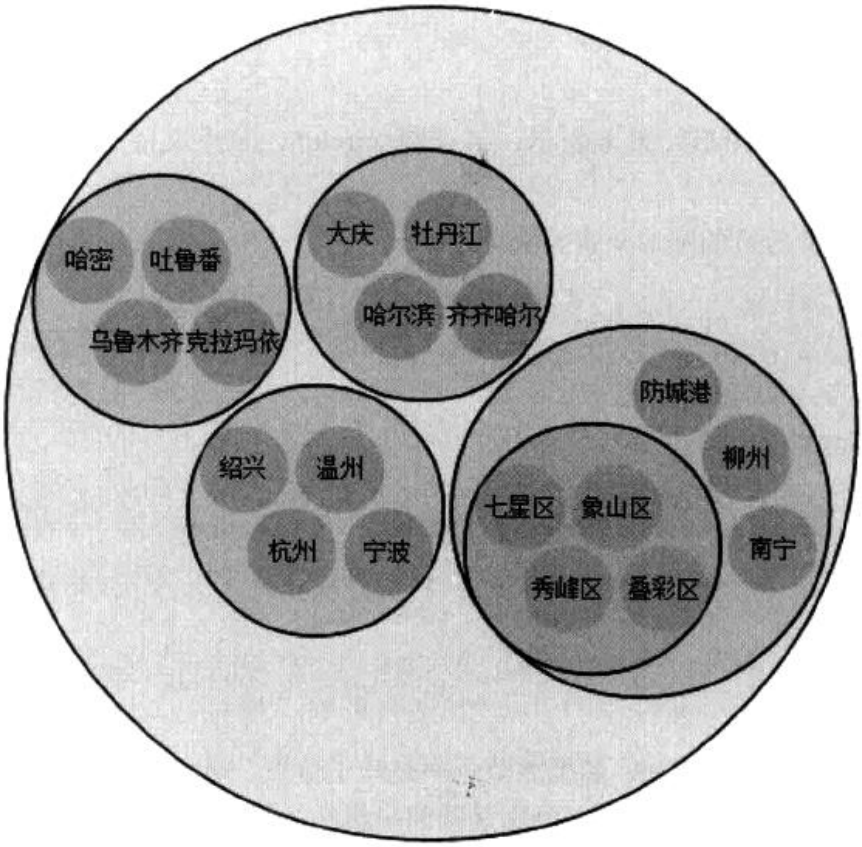


图 10-36　打包图的最终结果

10.9　直方图

直方图（Histogram），是用于表示数据分布的图表。

何为**数据分布**？

假设，某学校高三学生共有 100 人。这 100 人里：

- 身高在 150cm～159cm 的有 5 人。
- 身高在 160cm～169cm 的有 25 人。
- 身高在 170cm～179cm 的有 45 人。
- 身高在 180cm～189cm 的有 22 人。
- 身高在 190cm～199cm 的有 3 人。

也就是说，100 个人的身高**数据**分布在上述五个区间里。将以上分布表现出来的图表，就是直方图。光看“直方图”这三个字，很容易让人联想到柱形图，但两者的意义完全不同。

下面介绍直方图的相关方法。

- d3.layout.**histogram**()

创建一个直方图布局。

- histogram.**value**([accessor])

设定或获取值访问器。

- histogram.**range**([range])

设定数据分布的范围，通过一个只有两项的数组来指定，分别表示上下限，如[min, max]。

- histogram.**bins**()
  - histogram.**bins**(count)
  - histogram.**bins**(thresholds)
  - histogram.**bins**(function)

设定 bins 的个数或长度，即数据的分布区间。

如果没有参数，则返回当前设定的值。

如果以整数 count 作为参数，则 count 代表 bins 的数量，bins 的区间长度则被均匀地分隔。

如果以数组 thresholds 作为参数，则可以指定任意长度的区间，例如：

[140, 160, 165, 170, 175, 180, 200]

如果以函数 function 作为参数，该函数必须返回一个 thresholds 数组。

- histogram.**frequency**([frequency])

如果 frequency 的值为 true，则统计的是数量；如果为 false，则统计的是概率。

下面制作一个直方图，要求如下。

按正态分布随机生成 100 个人的身高，统计在身高范围[130, 210]内的分布情况，该区分要均等分为 20 段。最终分别以矩形和曲线来表现此正态分布。

1．确定初始数据

D3 中提供有 d3.random.normal()，用于生成正态分布，该函数的第一个参数是平均值，第二个是标准差，本例中分别设置为 170 和 10，即这 100 个人的平均身高是 170cm。请看下面的代码：

```
//创建一个生成正态分布的函数，其中平均值为170，标准差为10
var rand = d3.random.normal(170,10);

//定义一个数组，用于保存正态分布生成的数值
var dataset = [];

//根据正态分布生成100个随机数，插入到数组dataset中
```

```
for(var i=0; i<100; i++){
    dataset.push( rand() );
    }

//输出数组到控制台
console.log(dataset);
```

按正态分布生成的 100 个数值被保存在数组 dataset 里。dataset 的部分数据如图 10-37 所示。

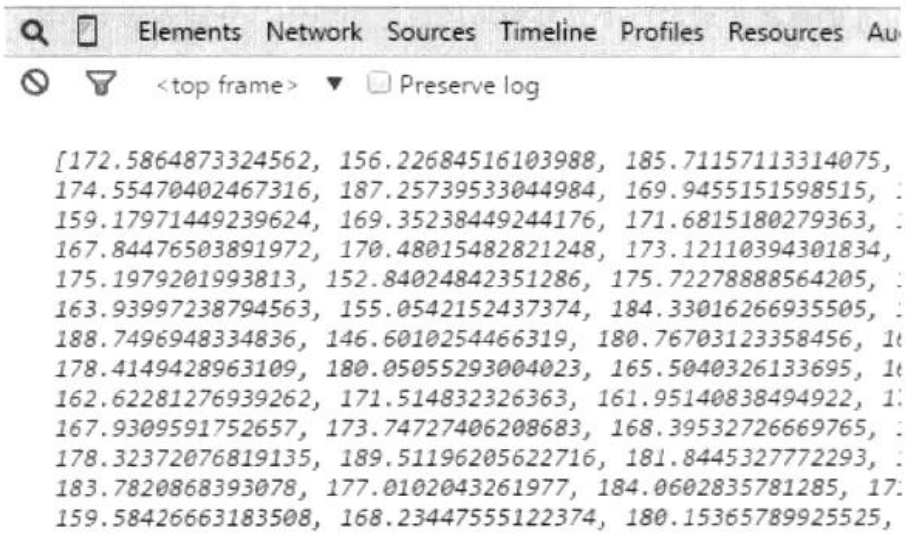


图 10-37　打包图的最终结果

2．转换数据

创建一个直方图布局，设定 bins 数为 20，数据分布的范围为 130cm～210cm，统计方式为数量统计：

```
var binNum = 20,
    rangeMin = 130,
    rangeMax = 210;

var histogram = d3.layout.histogram()
                  .range([rangeMin, rangeMax])     //数据分布的范围
                  .bins(binNum)                    //bins的数量
                  .frequency(true);                //按照数量统计的方式

var hisData = histogram(dataset);                  //转换数据
console.log(hisData);
```

转换后的数据保存在变量 hisData 中，在控制台的输出结果如图 10-38 所示。可以看到 hisData 变成一个长度为 20 的数组，数组的每一项代表一个分布区间。每一项里含有落到此区

间的数值，例如落到第六区间的数值有：

156.22，157.23，154.14，155.05

此外，还有三个变量，意义如下。

- x：区间的下限值。
- dx：区间的长度。
- y：落到此区间的数量（frequency 为 true）或概率（frequency 为 false）。

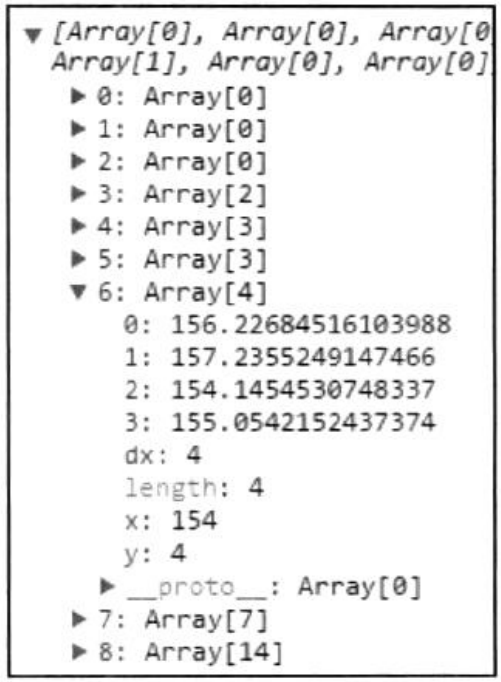


图 10-38　经直方图布局转换后的数组

3．绘制

由转换后的数据可知，x 轴表示区间，y 轴表示此区间的数量。例如，身高在 158cm～162cm（x 轴）区间内的人数为 14 人（y 轴）。我们知道，不能用 14 个像素来代表 14 个人，那样太短了。因此，需要用到比例尺，x 轴使用序数比例尺，y 轴使用线性比例尺。

首先，定义 x 轴的比例尺：

```
var xAxisWidth = 450,
    xTicks = hisData.map( function(d){ return d.x; } );

    var xScale = d3.scale.ordinal()
                   .domain(xTicks)
                   .rangeRoundBands([0, xAxisWidth],0.1);
```

xAxisWidth 是坐标轴的实际长度，单位为像素，作为比例尺值域的上限使用。xTicks 是一个数组，保存有 x 轴的刻度，其值为：

```
[130, 134, 138, 142, 146, 150, 154, 158, 162, 166, 170, 174, 178, 182, 186,
190, 194, 198, 202, 206]
```

这是使用 hisData.map()求得的。map()是数组对象 Array 的一个方法，在 map 中定义一个无名函数 function(d)，如此一来，hisData 的每一项元素都会调用此无名函数，最终返回一个数组。xTicks 作为 *x* 轴比例尺的定义域使用。

其次，绘制 *x* 轴，令 *x* 轴的刻度在轴的下方。

```
//外边框
var padding = { top: 30 , right: 30, bottom: 30, left: 30 };

//绘制x轴
var xAxis = d3.svg.axis()
          .scale(xScale)
          .orient("bottom")
          .tickFormat(d3.format(".0f"));

svg.append("g")
       .attr("class","axis")
       .attr("transform","translate(" + padding.left + "," +
                         (height - padding.bottom) + ")")
       .call(xAxis);
```

结果如图 10-39 所示。坐标轴的每一个刻度表示一个区间，例如 130 表示身高在 130cm～134cm 之间。

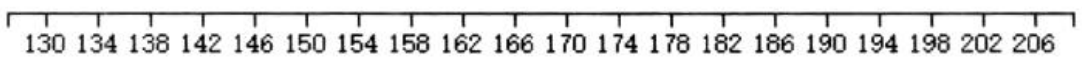


图 10-39　直方图的 *x* 轴

然后，添加 *y* 轴的比例尺。虽然本例不将 *y* 轴画出来，但是需要比例尺来伸缩长度，因为刚才提到，不可能用 14 个像素来表示 14 个人。代码如下：

```
var yAxisWidth = 450;

var yScale = d3.scale.linear()
          .domain([ d3.min(hisData, function(d){ return d.y; }),
                    d3.max(hisData, function(d){ return d.y; }) ])
          .range([5, yAxisWidth]);
```

这是一个线性比例尺，要用于计算矩形和曲线的 *y* 坐标，其值域被设定为[5, 450]，也就是

说，d.y 中的最小值对应到 5，最大值对应到 450。

最后，添加矩形元素。绑定数组 hisData，添加对应数量的矩形，再用比例尺分别计算矩形的 x、y、width、height：

```
//绘制矩形
var gRect = svg.append("g")
              .attr("transform","translate(" + padding.left + "," +
                                        ( -padding.bottom ) +  ")");

gRect.selectAll("rect")
       .data(hisData)
       .enter()
       .append("rect")
       .attr("class","rect")
       .attr("x",function(d,i){                    //x坐标
           return xScale(d.x);
       })
       .attr("y", function(d,i){                   //y坐标
           return height - yScale(d.y);
       })
       .attr("width", function(d,i){               //宽度
           return xScale.rangeBand();
       })
       .attr("height", function(d){                //高度
           return yScale(d.y);
       });
```

要注意比例尺 xScale 和 yScale 与其参数的使用，以下三个值的意义尤为重要。

xScale(d.x)

d.x 是各区间的下限值，即 130、134、138 等。xScale 是序数比例尺，将离散值对应到离散值。xScale 的值域为 6、28、50 等一系列离散的值。那么，当 d.x 是 130 时，返回 6；当 d.x 是 134 时，返回 28。这与 *x* 坐标轴正好是对应的，也就是所需的矩形的 *x* 坐标。

yScale(d.y)

d.y 是落到各区间的数值的数量。yScale 是线性比例尺，将 d.y 按比例伸缩。

xScale.rangeBand()

序数比例尺每一段的宽度，在这里用作矩形的宽度。

用矩形来表示正态分布的结果如图 10-40 所示。经过可视化后，可以一眼看出高三年级处于哪一身高段的人多。

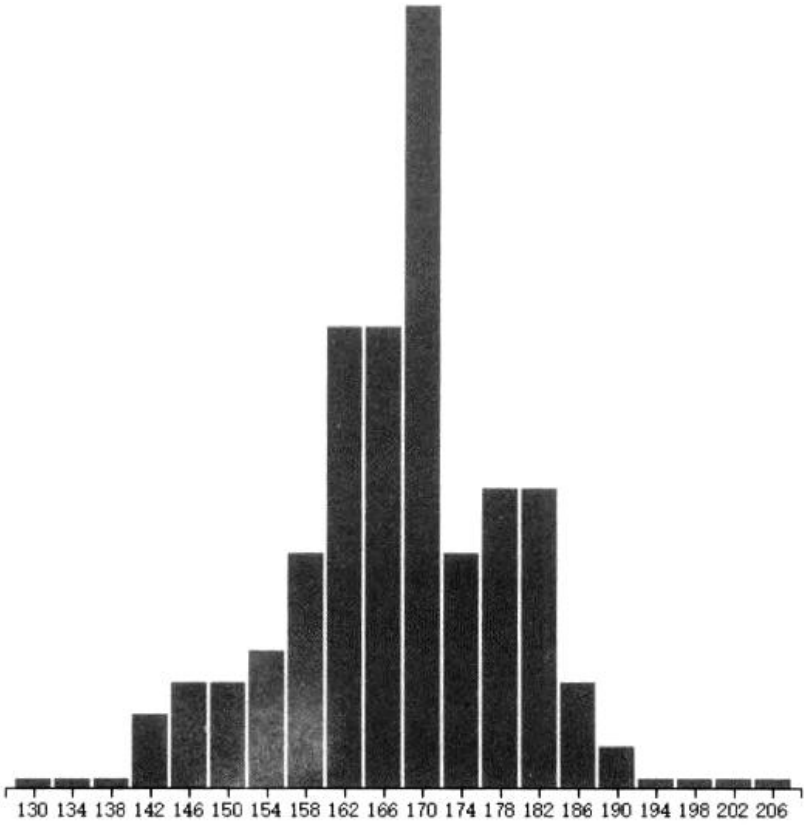


图 10-40　矩形直方图

也可以用曲线表示正态分布：

```
//绘制曲线
var lineGenerator = d3.svg.line()
                      .x(function(d){ return xScale(d.x); })
                      .y(function(d){ return height - yScale(d.y); })
                      .interpolate("basis");

var gLine = svg.append("g")
               .attr("transform","translate(" + padding.left + "," +
                                    ( -padding.bottom ) +  ")")
                   .style("opacity",0.0);

gLine.append("path")
    .attr("class","linePath")
    .attr("d",lineGenerator(hisData));
```

第 6 章介绍的线段生成器 d3.svg.line 可轻易实现。线段生成器的 x 访问器即相当于矩形的 x 坐标，y 访问器相当于矩形的 y 坐标，二者的计算方法一样。插值模式选择 basis，使用贝塞尔曲线。

曲线的绘制结果如图 10-41 所示，如此也可以表示正态分布。这里可体会到：布局只是用来转换数据（或者说计算数据）的，只能得到坐标、数量等信息，至于要用什么图形来表示该数据，是用矩形，还是圆形，或是线段，那是另一回事。

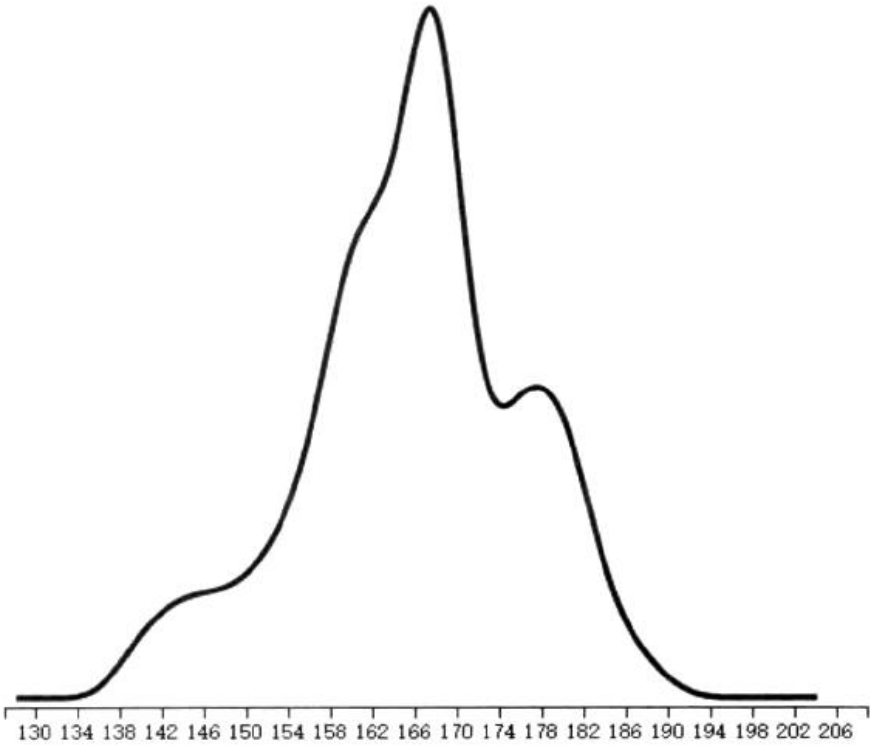


图 10-41　曲线直方图

10.10　分区图

分区图（Partition Diagram），也是层级布局的扩展，可以做成矩形形态和圆形形态。分区图中的每一个元素的尺寸大小都等于其子元素的尺寸大小之和。

- d3.layout.**partition**()

创建一个分区图布局。

- partition.**size**([size])

设定或获取布局的尺寸，参数 size 是只有两个元素的数组，分别表示宽和高。

- partition.**children**([children])

设定或获取子节点访问器。默认情况下，当前节点对象的变量 children 是子节点。

- partition.**value**([value])

设定或获取值访问器。

- partition.**sort**([comparator])

设定或获取节点排序的比较器。

- partition.**nodes**(root)

根据 root 进行计算，获取节点数组。

- partition.**links**(nodes)

根据 nodes 进行计算，获取连线数组。

下面制作一个分区图，要求如下：使用第 10.5 节树状图的数据，分别制作矩形形态的分区图和圆形形态的分区图。

1. 确定初始数据

使用第 10.5 节的 city.json 文件。

2. 转换数据

首先，创建一个分区图布局，设定尺寸为[800, 500]，排序比较器为 null。对于矩形分区图，用矩形的宽和高来设置尺寸即可：

```
var partition = d3.layout.partition()
            .sort(null)
            .size([800,500])          //矩形的宽和高
            .value(function(d) { return 1; });
```

然后，调用 d3.json()请求 city.json 文件。获取到文件后，用 partition.nodes()计算节点数组，用 partition.links()计算连线数组：

```
d3.json("city.json", function(error, root) {

    if(error)
        console.log(error);
    console.log(root);

    var nodes = partition.nodes(root);
    var links = partition.links(nodes);

    console.log(nodes);
}
```

在控制台输出节点数组 nodes，如图 10-42 所示。每一个节点对象所包含变量的意义如下。

- x：节点的 *x* 坐标。
- y：节点的 *y* 坐标。
- dx：节点的宽度。
- dy：节点的高度。

- depth：节点的深度（从 0 开始）。
- children：子节点数组。
- parent：父节点数组。
- value：节点的值，由 value 访问器指定，且父节点的 value 值必定等于其子节点的 value 值之和。

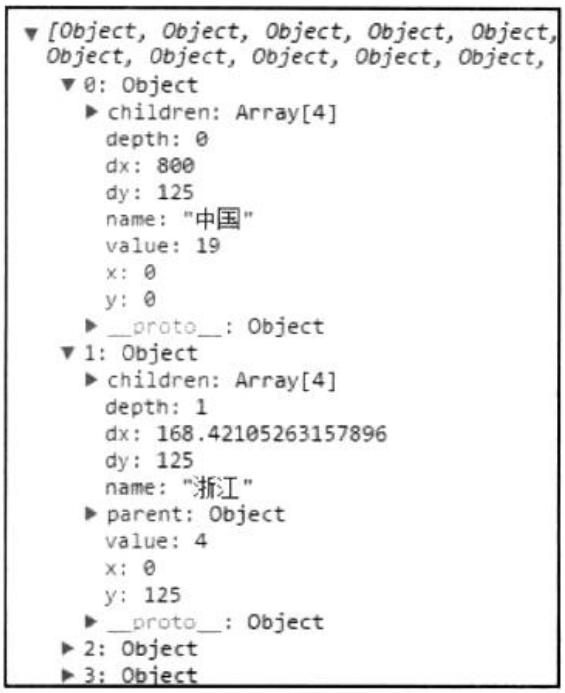


图 10-42　分区图的节点数组

3. 绘制

首先，明确用什么图形元素：节点使用矩形元素<rect>，节点文字使用文字元素<text>。各节点的<rect>和<text>可视为一个整体，放到<g>里。因此，绑定节点数组并据此添加足够数量的<g>元素：

```
var gRects = svg.selectAll("g")
                .data(nodes)
                .enter()
                .append("g");
```

然后，依次向 gRects 里添加<rect>和<text>：

```
gRects.append("rect")
    .attr("x", function(d) { return d.x; })
    .attr("y", function(d) { return d.y; })
    .attr("width", function(d) { return d.dx; })
    .attr("height", function(d) { return d.dy; })
```

```
    .style("stroke", "#fff")
    .style("fill", function(d) {
        return color((d.children ? d : d.parent).name);
    });

gRects.append("text")
    .attr("class","nodeText")
    .attr("x",function(d){ return d.x; })
    .attr("y",function(d){ return d.y; })
    .attr("dx", 20 )
    .attr("dy", 20 )
    .text(function(d,i) { return d.name; });
```

节点对象的 x、y、dx、dy 正好对应矩形的四个属性。此外，对于 SVG 的文字元素<text>，可在其 style 里设置 writing-mode，令其变为竖排显示：

```
.nodeText {
    writing-mode: tb;
    font-family: simsun;
    font-size: 16px;
}
```

矩形分区图的结果如图 10-43 所示。

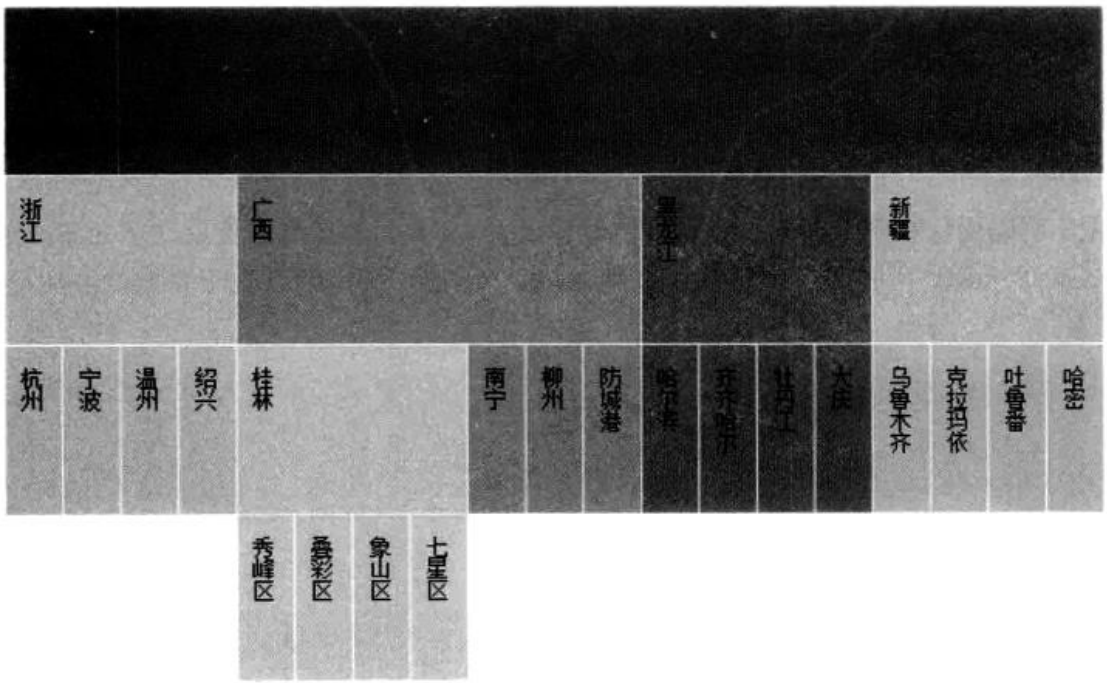


图 10-43　矩形分区图

如果要将分区图变成圆形，需要重新定义分区图布局，主要是设定布局的尺寸：

```
var partition = d3.layout.partition()
                  .sort(null)
                  .size([2 * Math.PI, radius * radius])
                  .value(function(d) { return 1; });
```

如此设置后，经过布局计算得到的节点对象里，x 和 dx 代表角度，y 和 dy 代表半径，这是因为 size 数组的第一项为 2*Math.PI，第二项为 radius*radius。

然后，创建一个弧生成器，将起始角度、终止角度、内半径、外半径的访问器分别按如下方式设定：

```
var arc = d3.svg.arc()
        .startAngle(function(d) { return d.x; })
        .endAngle(function(d) { return d.x + d.dx; })
        .innerRadius(function(d) { return Math.sqrt(d.y); })
        .outerRadius(function(d) { return Math.sqrt(d.y + d.dy); });
```

在创建矩形分区图的时候，d.x 代表矩形的 x 坐标，d.dx 代表矩形的宽。现在 d.x 代表角度了，那么起始角度自然为 d.x，终止角度为 d.x + d.dx。关于内半径和外半径的设定，d.y 和 d.dy 的意义也是如此，但是由于 d.y 和 d.dy 都是“半径的平方（radius*radius）”，因此要加上 Math.sqrt() 求平方根。

最后，添加各分区的元素，与矩形分区图类似，先添加足够数量的<g>，再在<g>里添加<path>和<text>，其中<path>用于绘制弧形：

```
var gArcs = svg.selectAll("g")
                .data(nodes)
                .enter()
                .append("g");

gArcs.append("path")
        .attr("display", function(d) {      //圆中心的弧不绘制
            return d.depth ? null : "none";
        })
        .attr("d", arc) //使用弧生成器
        .style("stroke", "#fff")
        .style("fill", function(d) {
            return color((d.children ? d : d.parent).name);
        });

gArcs.append("text")
        .attr("class","nodeText")
```

```
        .attr("dy",".5em")
        .attr("transform",function(d,i){
            if( i !== 0 ){        //除圆中心的文字外，都进行平移旋转
            var r = d.x + d.dx/2;   //旋转角度

            //超过180° 的文字调整旋转角度，防止文字是倒的
            var angle = Math.PI/2;
            r += r < Math.PI ? ( angle * -1 ) : angle ;
            r *= 180 / Math.PI;

            return  "translate(" + arc.centroid(d) + ")" +
                    "rotate(" + r + ")";
        }
    })
    .text(function(d,i) { return d.name; });
```

弧生成器用于为每一个<path>元素生成路径。文字<text>的位置通过设置 transform 属性使其平移旋转到指定位置，由于分区图是圆形，要防止出现倒文字。

圆形分区图的结果如图 10-44 所示。

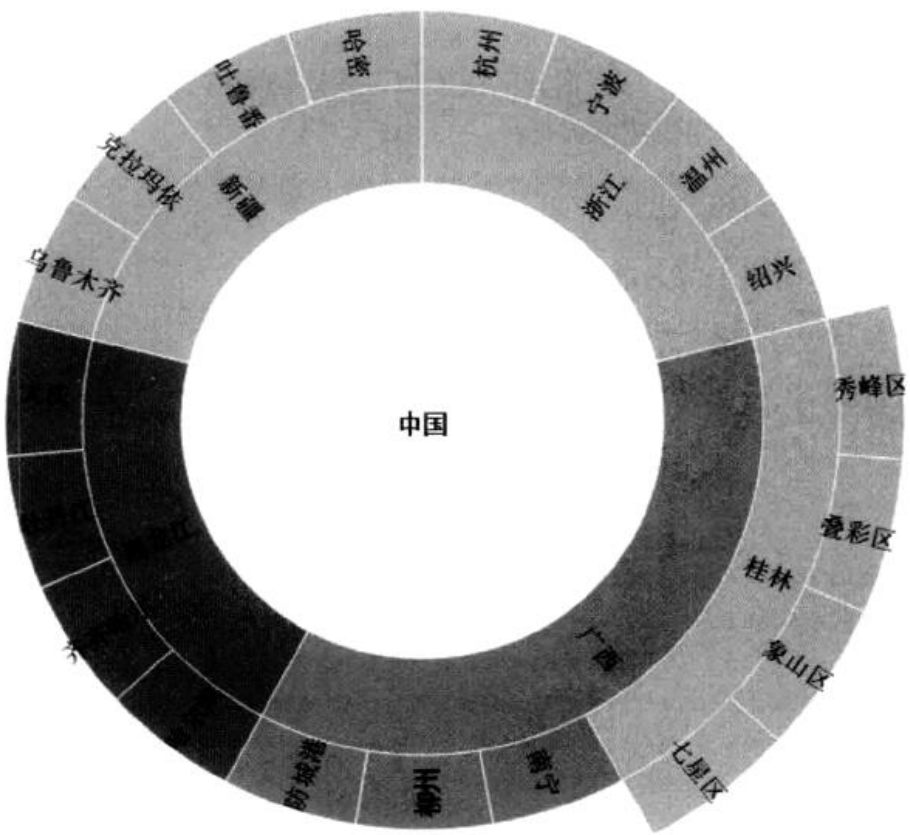


图 10-44　圆形分区图

10.11　堆栈图

堆栈图布局（Stack Layout）能够计算二维数组每一数据层的基线，以方便将各数据层叠加起来。例如，有如下情况。

某公司，销售三种产品：个人电脑、智能手机、软件。

2005 年，三种产品的利润分别为 3000 万元、2000 万元、1100 万元。

2006 年，三种产品的利润分别为 1300 万元、4000 万元、1700 万元。

计算可得，2005 年总利润为 6100 万元，2006 年总利润为 7000 万元。

如果要将 2005 年的利润用柱形表示，那么应该画三个矩形，三个矩形堆叠在一起。这时候就有一个问题：**每一个矩形的起始 *y* 坐标是多少，高应该是多少？**

输入数组，直接为上述问题求解的，就是堆栈图布局。

堆栈图布局包含以下方法。

- d3.layout.**stack**()

创建一个堆栈图布局。

- stack.**values**([accessor])

设定或获取各层的值访问器，不指定的话默认为 function(d){ d.values }。

- stack.**offset**([children])

设定或获取堆栈图的堆叠算法，有四个值可供选择。

  - **zero**：以 0 为基准，高与 y 值成比例，最常用。
  - **silhouette**：高度与 zero 相同，但是各层都居中，不以 0 为基准。
  - **expand**：合计为 1，以比例计算。
  - **wiggle**：减小各层的倾斜。

- stack.**order**([order])

设定或获取各层的顺序，有三个值可供选择。

  - **default**：使用输入数组默认的顺序。
  - **reverse**：使用与输入数组相反的顺序。
  - **inside-out**：使用 stack(layers[, index])的 index 指定的顺序

- stack.**x**([accessor])

设定或获取 x 访问器，用于决定 *x* 坐标。

- stack.**y**([accessor])

设定或获取 y 访问器，用于决定 *y* 坐标和高 y0。

- stack.**out**([setter])

设定或获取计算后数据的保存方法，一般使用默认即可。

下面制作一个堆栈图，要求如下。

有某公司从2005年到2009年销售个人电脑、智能手机、软件的数据，用堆栈图布局将其可视化。

1. 确定初始数据

某公司销售个人电脑、智能手机、软件的数据如下：

```
var dataset = [
    { name: "PC" ,
      sales: [  { year:2005, profit: 3000 },
                { year:2006, profit: 1300 },
                { year:2007, profit: 3700 },
                { year:2008, profit: 4900 },
                { year:2009, profit: 700 }] },
      { name: "SmartPhone" ,
      sales: [  { year:2005, profit: 2000 },
                { year:2006, profit: 4000 },
                { year:2007, profit: 1810 },
                { year:2008, profit: 6540 },
                { year:2009, profit: 2820 }] },
    { name: "Software" ,
      sales: [  { year:2005, profit: 1100 },
                { year:2006, profit: 1700 },
                { year:2007, profit: 1680 },
                { year:2008, profit: 4000 },
                { year:2009, profit: 4900 }] }
    ];
```

dataset是一个数组，数组的每一项是一个对象，对象里含有name和sales。name是产品名、sales是销售情况。sales也是一个数组，每一项也是对象，对象里包含有year表示年份、profit表示利润。

2. 转换数据

首先，创建一个堆栈图布局，并设定x访问器和y访问器：

```
var stack = d3.layout.stack()
              .values(function(d){ return d.sales; })
              .x(function(d){ return d.year; })
              .y(function(d){ return d.profit; });
```

values访问器指定的是d.sales，表示接下来接收的数组，要计算的数据是数组每一项的变量sales。x访问器指定的是d.year，y访问器指定的是d.profit，都是相对于values访问器指定的对象说的，即数组sales各项的变量year和profit。

然后，以dataset为布局stack的参数，返回被转换后的数据：

```
var data = stack(dataset);
console.log(data);
```

要注意，对堆栈图布局来说，转换之后原数据也会改变，因此 dataset 和 data 的值是一样的。data 的输出值如图 10-45 所示。可以看到，sales 的每一项都多了两个值：y0 和 y。y0 即该层起始坐标，y 是高度。*x* 坐标就是 year。

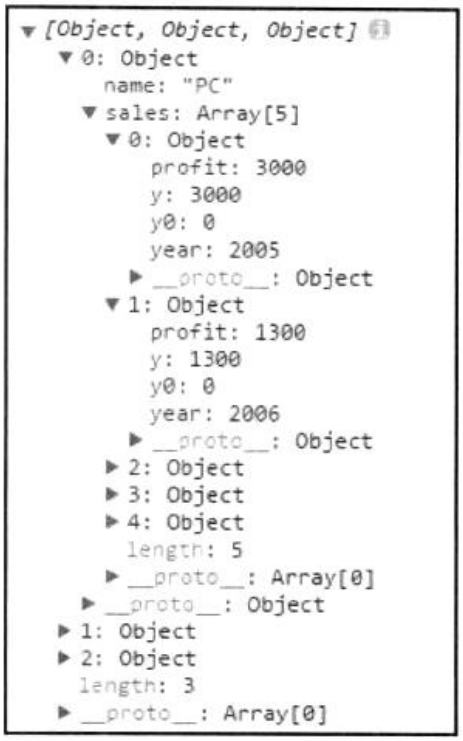


图 10-45　堆栈图布局的数组

3. 绘制

首先，创建 *x* 轴和 *y* 轴比例尺，在添加图形元素和坐标轴的时候都要用到。由于绘制坐标轴，需要给坐标轴的刻度留出一部分空白。因此，先定义一个外边框：

```
var padding = { left:50, right:100, top:30, bottom:30 };
```

右边部分留出的空白较多，是为了在后面添加标签。*x* 轴比例尺的定义如下：

```
var xRangeWidth = width - padding.left - padding.right;

    var xScale = d3.scale.ordinal()
                  .domain( data[0].sales.map(function(d){
                        return d.year;
                  }))
                    .rangeBands([0, xRangeWidth],0.3);
```

本例中 x 轴代表年份，2005 年、2006 年、2007 年等，是离散的，也就是说比例尺的定义域是离散的。从第 5 章的内容可知，序数比例尺 d3.scale.ordinal 的定义域是离散的。上面代码将定义域设定成：

```
[2005, 2006, 2007, 2008, 2009]
```

值域是根据 rangeBands()计算的，实际是：

```
[31, 134, 238, 342, 446]          //（省略了小数点）
```

因此，在 2005 年处堆叠的矩形的 x 坐标为 31。再创建 y 轴的比例尺，代码如下：

```
//最大利润（定义域的最大值）
var maxProfit = d3.max(data[data.length-1].sales, function(d){
                    return d.y0 + d.y;
                });

//最大高度（值域的最大值）
var yRangeWidth = height - padding.top - padding.bottom;

var yScale = d3.scale.linear()
                .domain([0, maxProfit])     //定义域
                .range([0, yRangeWidth]);   //值域
```

这段代码中，求最大利润时，是对 data 数组中的最后一项 data[2]求取 sales 的最大值。这是因为 data[2]代表着最高的层，如图 10-46 所示，data[2].sales 中的各项 y0+y，必定比 data[1]和 data[0]的大。因此，只要用 d3.max()求取 data[2].sales 中的最大值即可。值域是 SVG 的高度减去外边框的上下宽度。然后为 d3.scale.linear()设定定义域和值域即可。

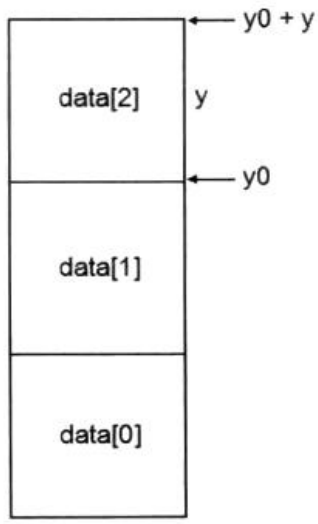


图 10-46　data 数组中的每一项所代表的层

其次，添加足够数量的分组元素<g>，每一个分组代表一种产品，每一种产品都用一种颜色来标识。

```
//颜色比例尺
var color = d3.scale.category10();

//添加分组元素
var groups = svg.selectAll("g")
               .data(data)
               .enter()
               .append("g")
               .style("fill",function(d,i){ return color(i); });
```

现在添加了三个分组，分别代表 PC、SmartPhone、Software，且每一个分组元素的 fill 都设定了颜色。

再次，为每个分组<g>添加矩形元素<rect>：

```
//添加矩形
var rects = groups.selectAll("rect")
              .data(function(d){ return d.sales; })
              .enter()
              .append("rect")
              .attr("x",function(d){ return xScale(d.year); })
              .attr("y",function(d){
                  return yRangeWidth - yScale( d.y0 + d.y );
              })
              .attr("width",function(d){
                  return xScale.rangeBand();
              })
              .attr("height",function(d){ return yScale(d.y); })
              .attr("transform","translate(" + padding.left + ","
                                             + padding.top + ")");
```

每一个分组元素里还要绑定数组 sales，以添加足够数量（每个分组 5 个）的矩形。然后再使用比例尺为矩形的 x、y、width、height 属性赋值。

然后，添加坐标轴，结果如图 10-47 所示。

但是，什么颜色代表什么产品从图里看不出来。解决此问题最常用的方法是：在图表旁边添加几个图形标志，并加上文字，告诉用户某种颜色对应的是什么。

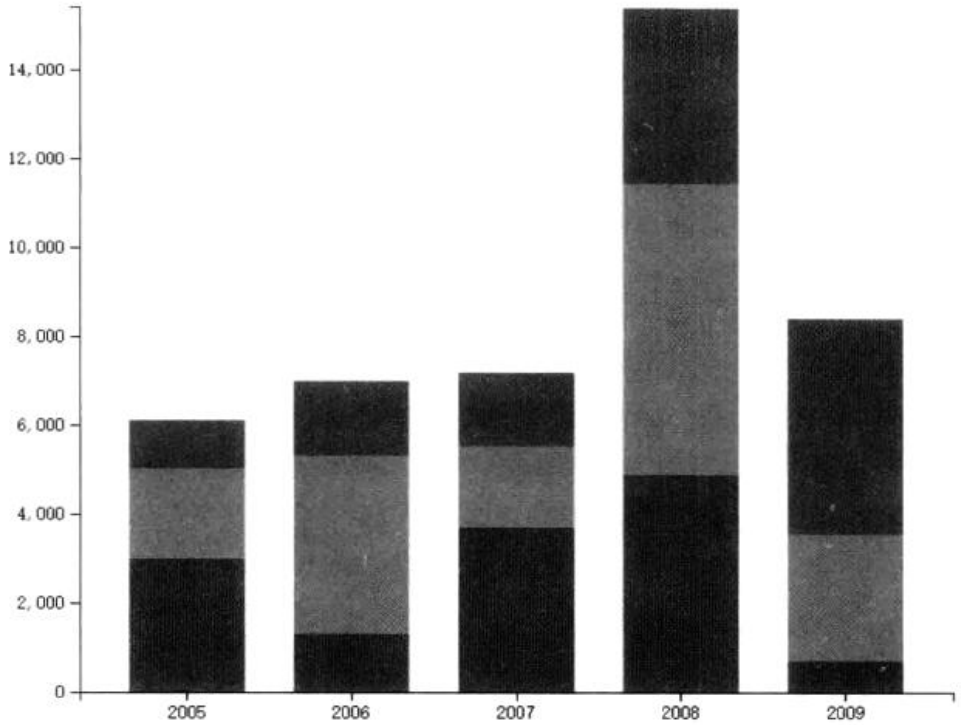


图 10-47　堆栈图结果

最后，在分组<g>里添加用于标志的图形元素：<circle>和<text>，代码如下：

```
var labHeight = 50;
var labRadius = 10;

var labelCircle = groups.append("circle")
                        .attr("cx",function(d){
                            return width - padding.right*0.98;
                        })
                        .attr("cy",function(d,i){
                            return padding.top * 2 + labHeight * i;
                        })
                        .attr("r",labRadius);

var labelText = groups.append("text")
                      .attr("x",function(d){
                          return width - padding.right*0.8;
                      })
                      .attr("y",function(d,i){
                          return padding.top * 2 + labHeight * i;
```

```
                    })
                    .attr("dy",labRadius/2)
                    .text(function(d){ return d.name; });
```

将标志放到图表的右上角，最终结果如图 10-48 所示。

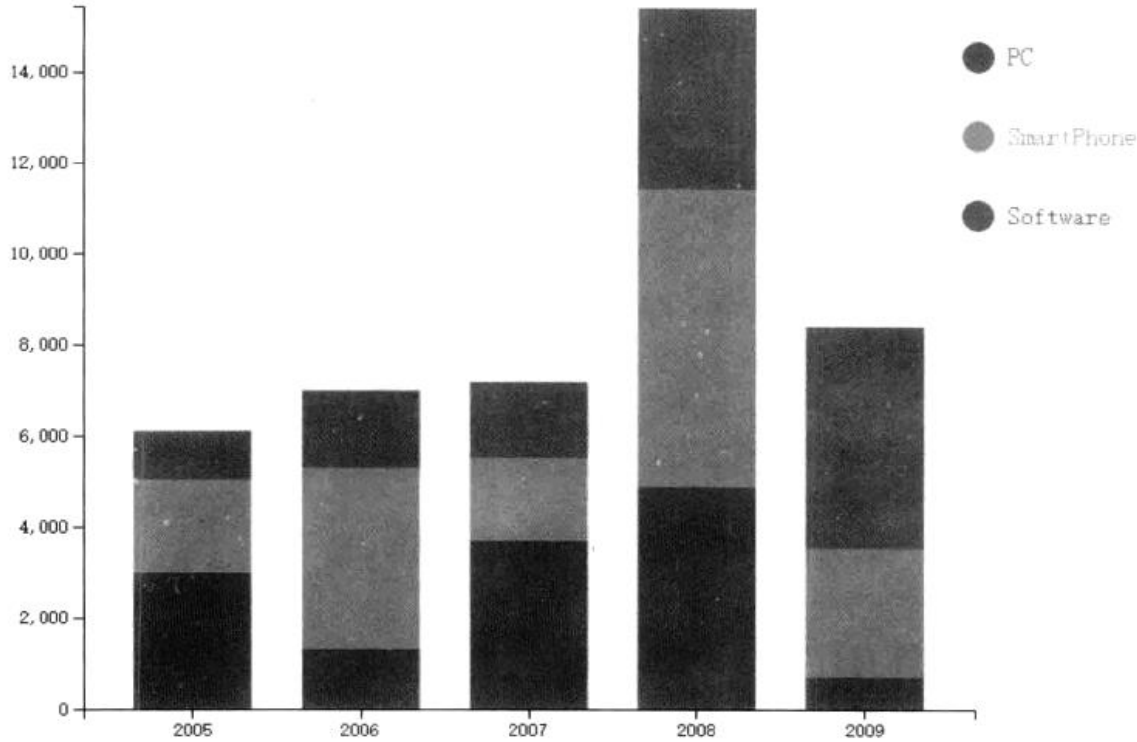


图 10-48　添加标签文字后的堆栈图

除了用矩形表示堆叠效果，另一种常见的是使用面积区域。第 6 章提到的区域生成器 d3.svg.area 正好可以用上。创建一个区域生成器如下：

```
var area = d3.svg.area()
             .x( function(d){
                 return xScale(d.year) + xScale.rangeBand()/2;
             })
             .y0( function(d){
                 return yRangeWidth - yScale(d.y0);
             })
             .y1( function(d){
                 return yRangeWidth - yScale(d.y0+d.y);
             })
             .interpolate("basis");
```

则堆栈图变成如图 10-49 所示的模样。

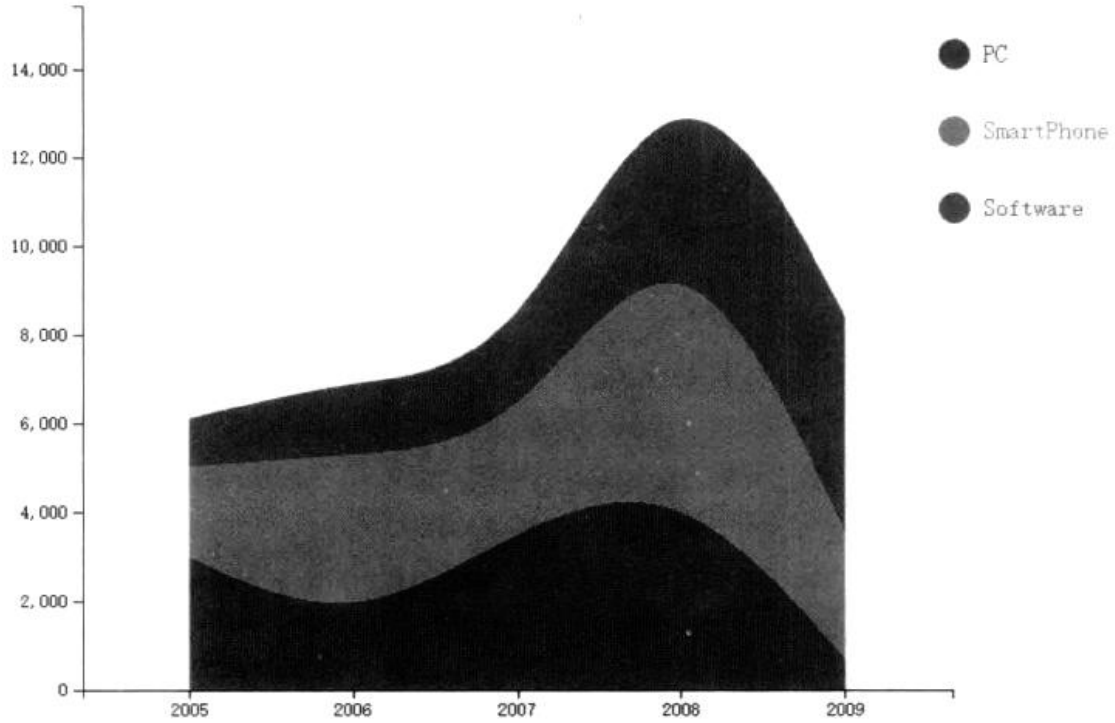


图 10-49　使用区域生成器绘制的堆栈图

10.12　矩阵树图

矩阵树图（Treemap），是层级布局的扩展，根据数据将区域划分为矩形的集合。矩形的大小和颜色，都是数据的反映。许多门户网站都能见到类似图 10-50 的图。将照片以不同大小的矩形排列的情形，这正是矩阵树图的应用。

由于矩阵树图也是层级布局的扩展，因此与树状图、集群图、打包图、分区图很像，部分方法的意义是相同的。

- d3.layout.**treemap**()

创建一个矩阵树图布局。

- treemap.**nodes**(root)

根据 root 进行计算，获取节点数组。

- treemap.**links**(nodes)

根据 nodes 进行计算，获取连线数组。

- treemap.**children**([children])

设定或获取子节点访问器。默认情况下，当前节点对象的变量 children 是子节点。

- treemap.**sort**([comparator])

设定或获取节点排序的比较器。

http://www.texastribune.org/2010/10/07/treemap-reveals-campaign-ad-trends/

图 10-50　矩阵树图的用例

- treemap.**value**([value])

设定或获取值访问器。

- treemap.**size**([size])

设定或获取布局的尺寸，参数 size 是只有两个元素的数组，分别表示宽和高。

- treemap.**padding**([padding])

设定或获取矩形单元之间的间隔，单位为像素。

- treemap.**round**([round])

设定或获取是否对计算结果进行四舍五入。

- treemap.**sticky**([sticky])

设定或获取矩阵树图是否是“黏性”的。

- treemap.**ratio**([ratio])

设定或获取布局的比例。

- treemap.**mode**([mode])

设定布局的模式，即生成矩阵的算法，不同算法结果有很大差异。有四个值可供选择。

squarify：按矩形分割，矩形受目标比例（ratio）的控制。是默认值。该模式如图 10-51 所示。

slice：水平分割。该模式如图 10-52 所示。

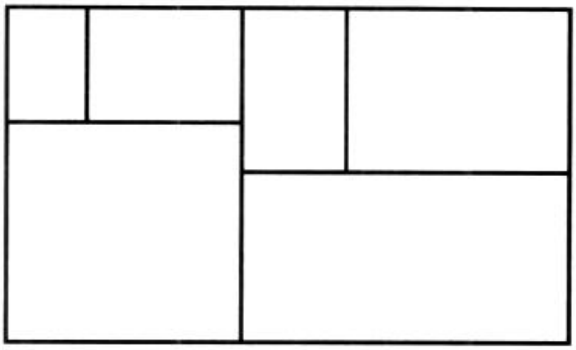


图 10-51　squarify 模式

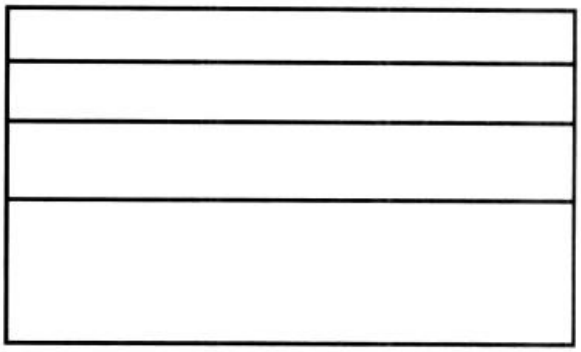


图 10-52　slice 模式

dice：垂直分割。该模式如图 10-53 所示。

slice-dice：水平和垂直交替分割。该模式如图 10-54 所示。

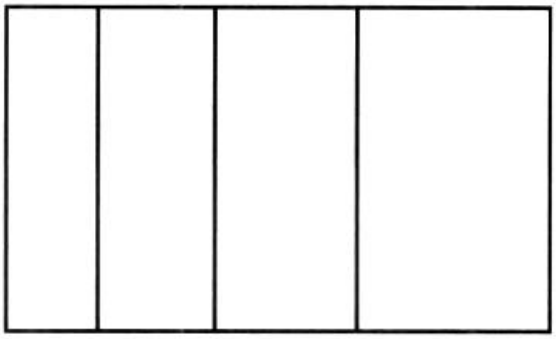


图 10-53　dice 模式

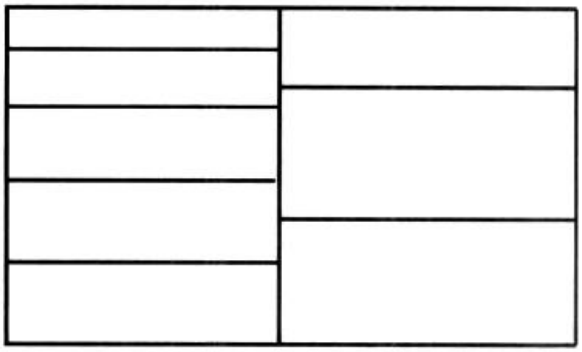


图 10-54　slice-dice 模式

下面制作一个矩阵树图，要求如下。

以浙江、广西、江苏三省份 2013 年的 GDP 作为数据，以 GDP 大小作为节点的权重将其制作成矩阵树图。

1. 确定初始数据

新建一个 citygdp.json 文件，内容如下：

```
{
   "name": "中国",
   "children":
   [
       {
           "name": "浙江",
           "children":
           [
               {"name":"杭州", "gdp":8343},
```

```
                {"name":"宁波", "gdp":7128},
                {"name":"温州", "gdp":4003},
                {"name":"绍兴", "gdp":3620},
                {"name":"湖州", "gdp":1803},
                {"name":"嘉兴", "gdp":3147},
                {"name":"金华", "gdp":2958},
                {"name":"衢州", "gdp":1056},
                {"name":"舟山", "gdp":1021}
            ]
        },
 ***************
    省略部分数据
 ***************
    ]
}
```

每一个叶子节点都包含有 name 和 gdp，name 是节点名称，gdp 是节点大小。省略部分的数据还包含有广西和江苏两省的城市。

2. 转换数据

首先，创建一个矩阵树图布局，尺寸设置为[width, height]，即 SVG 画板的尺寸。值访问器设定为 d.gdp，代码如下：

```
var treemap = d3.layout.treemap()
                .size([width, height])
                .value(function(d){ return d.gdp; });
```

这样设定值访问器后，每个节点都将拥有变量 value，且其值为 d.gdp 的值。如果一个节点存在子节点，则其 gdp 值为所有子节点的 value 的和。例如，节点“浙江”的 gdp 是省内各城市的 gdp 的和。

然后，用 d3.json 请求文件，再转换数据：

```
d3.json("citygdp.json", function(error, root) {
    var nodes = treemap.nodes(root);
    var links = treemap.links(nodes);

    console.log(nodes);
    console.log(links);
    }
```

转换数据后，节点数组的输出结果如图 10-55 所示。其中，节点对象的属性包括如下内容。

- parent：父节点。
- children：子节点。

- depth：节点的深度。
- value：节点的 value 值，由 value 访问器决定。
- x：节点的 x 坐标。
- y：节点的 y 坐标。
- dx：x 方向的宽度。
- dy：y 方向的宽度。

连线数组的输出如图 10-56 所示，各连线对象都包含有 source 和 target，分别是连线的两端。

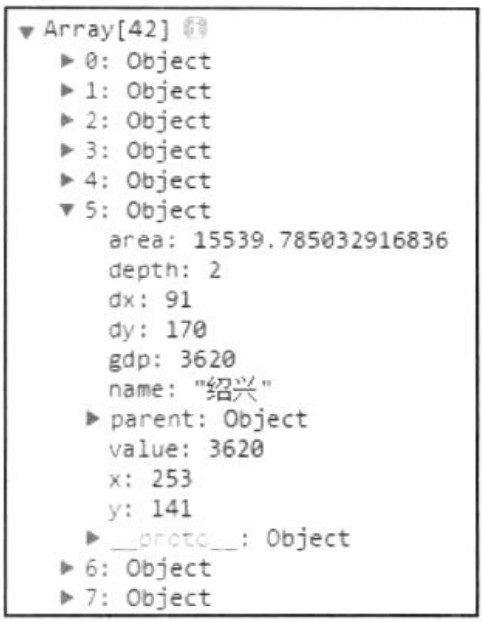


图 10-55　矩阵树图的节点数组

图 10-56　矩阵树图的连线数组

3. 绘制

本例不绘制连线，只使用节点数组。节点的绘制很简单，按节点数目添加足够的分组元素<g>，再在<g>里添加<rect>和<text>：

```
var groups = svg.selectAll("g")
        .data(nodes.filter(function(d){ return !d.children; }))
        .enter()
        .append("g");
```

```
var rects = groups.append("rect")
                .attr("class","nodeRect")
                .attr("x",function(d){ return d.x; })
                .attr("y",function(d){ return d.y; })
                .attr("width",function(d){ return d.dx; })
                .attr("height",function(d){ return d.dy; })
                .style("fill",function(d,i){
                    return color(d.parent.name); });

var texts = groups.append("text")
                .attr("class","nodeName")
                .attr("x",function(d){ return d.x; })
                .attr("y",function(d){ return d.y; })
                .attr("dx","0.5em")
                .attr("dy","1.5em")
                .text(function(d){
                    return d.name + " " + d.gdp;
                });
```

结果如图 10-57 所示。

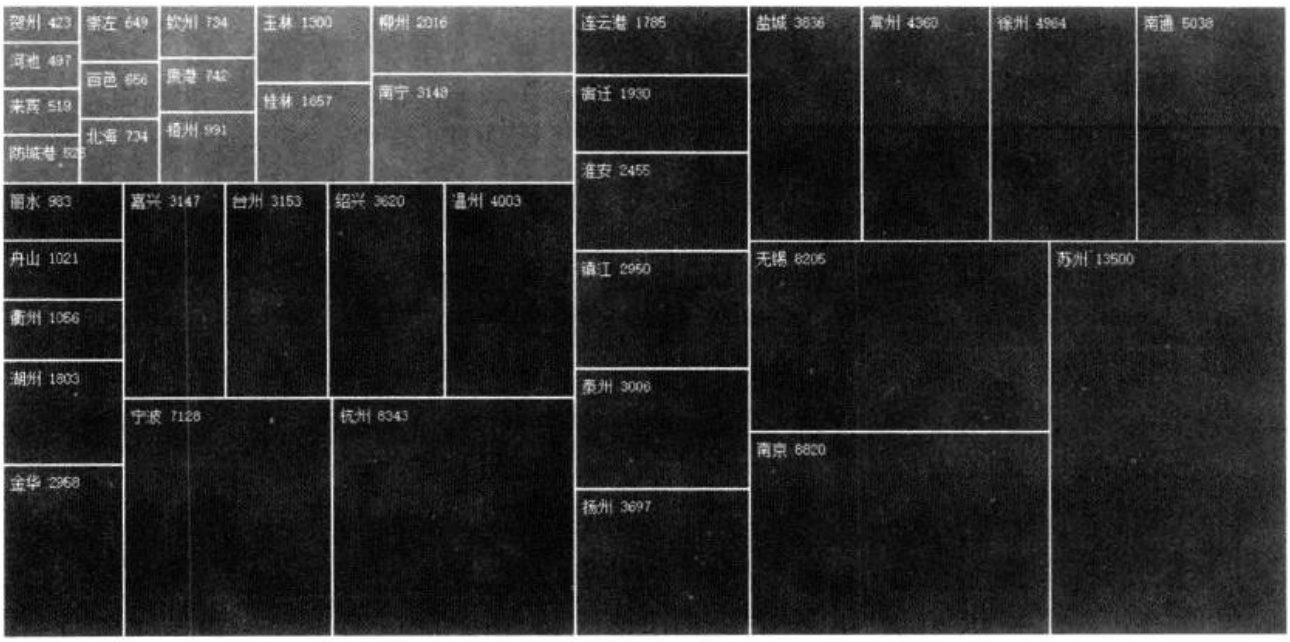


图 10-57　矩阵树图

第 11 章

地图

本章内容包括：

- 地图的数据格式
- 制作中国地图
- 地理路径生成器和形状生成器
- 投影
- 球面数学

第 1 节，讲述如何获取地理数据和简化数据，并介绍两种地图文件的格式：GeoJSON 和 TopoJSON。

第 2 节，基于 GeoJSON 和 TopoJSON 文件，分别制作中国地图。

第 3 节，介绍地理路径生成器和形状生成器，前者用于生成地图的路径，后者用于生成地图上的经纬度网格。

第 4 节，讲述地图的投影。

第 5 节，介绍 D3 提供的与球面数学相关的方法。

11.1 地图的数据

地图数据一般保存为 JSON 格式，D3 常用的有两种：GeoJSON 和 TopoJSON。其中，GeoJSON 是描述地理信息的一种基本格式，TopoJSON 是由 D3 的作者 Mike Bostock 制定的格式，它们都

符合 JSON 的规范。本节依次讲述获取地图数据、简化数据的方法，然后介绍分别详述 GeoJSON 和 TopoJSON 格式。

11.1.1　获取数据

全世界的地理信息都可以在 Natural Earth 下载到。Natural Earth 是由许多志愿者共同创建的，并且得到了 NACIS（北美制图信息学会）的支持。所有地理数据都可以自由免费应用到工程中。Natural Earth 的主页地址如下：

http://www.naturalearthdata.com/

打开网址后，单击页面中的“Downloads”标签，跳转到下载页面。下载页面里包含如图 11-1 所示的内容。有三种尺寸可供下载。

- 大尺寸（Large scale data），比例尺 1:10,000,000，1cm = 100km。
- 中尺寸（Medium scale data），比例尺 1:50,000,000，1cm = 500km。
- 小尺寸（Small scale data），比例尺 1:110,000,000，1cm = 1100km。

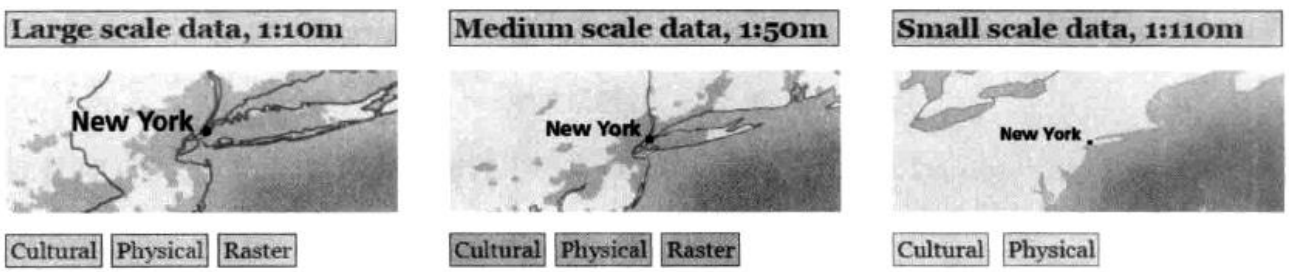


图 11-1　各种尺寸的地图

各尺寸里有“Cultural”、“Physical”、“Raster”三个选项。其中，Cultural 里包含具有“文化性”的地理信息，例如按国家和地区边界划分的地图、按行政省划分的地图、包含有飞机场港口的地图等。Physical 里包含的是“物理性”的地理信息，例如海岸线、陆地、海洋、河流等。Raster 里包含栅格地图。

如果要下载包含中国各省边界的地图，可单击“Large scale data, 1:10m”下的“Cultural”，在新页面中找到“Admin 1 - States, Provinces”，如图 11-2 所示。

单击“Download states and provinces”按钮，下载后，得到文件：

ne_10m_admin_1_states_provinces.zip

将该文件解压缩，得到的文件列表如图 11-3 所示。其中，最重要的文件是：

ne_10m_admin_1_states_provinces.shp

这里面包含有全球各国家和地区的地理信息，但是不能直接使用。接下来要从此 shp 文件里提取出需要的地理信息，并保存为 JSON 格式。

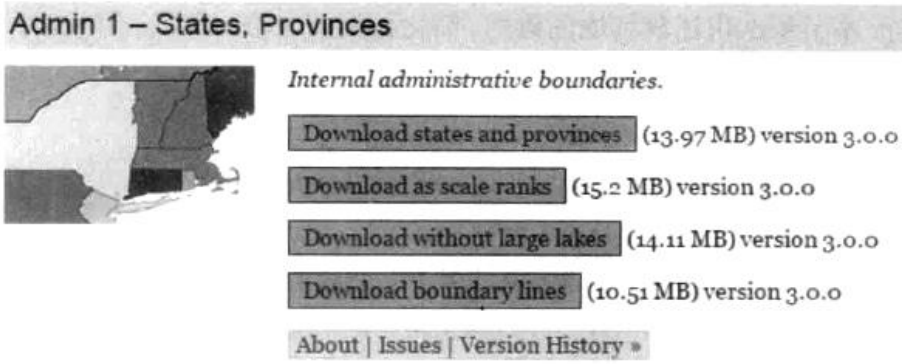


图 11-2　包括国家和省份的地图

名称	修改日期	类型	大小
ne_10m_admin_1_states_provinces.cpg	2013/10/8 15:35	CPG 文件	1 KB
ne_10m_admin_1_states_provinces.dbf	2013/10/8 15:35	OpenOffice.org ...	14,910 KB
ne_10m_admin_1_states_provinces.prj	2013/10/8 15:34	PRJ 文件	1 KB
ne_10m_admin_1_states_provinces.README.ht...	2013/6/2 20:10	Chrome HTML D...	29 KB
ne_10m_admin_1_states_provinces.shp	2013/10/8 15:49	SHP 文件	21,107 KB
ne_10m_admin_1_states_provinces.shx	2013/10/8 15:49	SHX 文件	37 KB
ne_10m_admin_1_states_provinces.VERSION.txt	2013/6/2 20:10	文本文档	1 KB

图 11-3　地图源文件

这里需要一个工具 ogr2ogr，它能按需提取 shp 文件中的地理信息，以及转换为 JSON 格式。但是，ogr2ogr 需要使用命令行操作，不太方便。有一个基于 ogr2ogr 开发的图形化软件：ogr2gui。这是一个免费开源的软件，基于 GPL 协议。下载地址为：

http://www.ogr2gui.ca/en/index.php

打开网址，可以选择 ogr2gui_0.7x32.zip 和 ogr2gui_0.7x64.zip 下载，分别对应 32 位和 64 位系统，根据自己的系统选择即可。下载文件解压缩后即可使用，软件界面如图 11-4 所示。

图 11-4 的软件界面中，在“Source”区域里选择源文件，在“Target”区域里选择输出的目标文件，其中在“Format”下拉列表框中可以选择输出格式，包括 JSON。在“Options”文本框中可输入可选命令。最下方的文本框里是自动生成的最终命令。ogr2gui 为转换数据提供了方便，不需要记忆过多的 ogr2ogr 命令。

下面试着转换 shp 文件，步骤如下。

（1）单击“Source”区域里的“Open”按钮，选择 shp 文件。

（2）在“Target”区域里的“Format”下拉列表框里选择 GeoJSON 格式。

（3）单击“Save”按钮，选择要保存的目标文件。

（4）在“Options”文本框里，输入：-where "ADM0_A3 IN ('CHN')"。

（5）单击“Execute”按钮运行，可获得一个 GeoJSON 文件。

图 11-4　ogr2gui 软件界面

如图 11-5 所示，按上述（1）～（4）步设定好后，最下方的文本框自动生成了 ogr2ogr 命令。单击“Execute”按钮运行后，将得到一个文件：**china.geojson**。它的格式符合 GeoJSON 规范，也可以将后缀改为 json，对使用没有影响。

上面提取了中国地图的数据。在“Options”文本框里输入的“ADM0_A3”是一种国名标准。此标准的全称是 ISO 3166-1 alpha-3，它允许用三个拉丁文字来表示国名，例如 CHN（中国）、USA（美国）、JPN（日本）、GBR（英国）、FRA（法国）等。

如要提取多个国家的地图数据，可用逗号将多个名称分隔。例如，提取中国、美国、英国到一个文件里，命令如下：

```
-where "ADM0_A3 IN ('CHN','USA','GBR')"
```

另外，在不少网站上可直接下载各种地图的 GeoJSON 文件，读者可在 Google、GitHub 等网站搜索下载，可省去不少麻烦。如果找不到满意的地图数据，再使用以上方法制作即可。Natural Earth 上的地理数据据说是最全面的。

图 11-5　使用 ogr2gui 获取 GeoJSON 文件

11.1.2　简化数据

原始的地图数据通常很大，例如上一节的 shp 文件有 20MB，其中包含有地图上细微的边界变化。我们需要这么多信息吗？

不需要。

只需要一个大概的边界，能让用户明白这是什么地形，是哪个国家即可。文件过大，读取要花很长时间。因此，要对数据进行简化。本书推荐一款在线的简化工具，网址为：

http://mapshaper.org/

打开后，出现如图 11-6 所示的对话框。该网站支持 SHP、GeoJSON、TopoJSON 三种能格式。

单击“select”按钮，选择上一节下载的 shp 文件，结果显示如图 11-7 所示。可以看到 shp 文件里所保存的世界地图，以及每个国家按省划分的边界线。

图 11-6　选择需要简化的文件

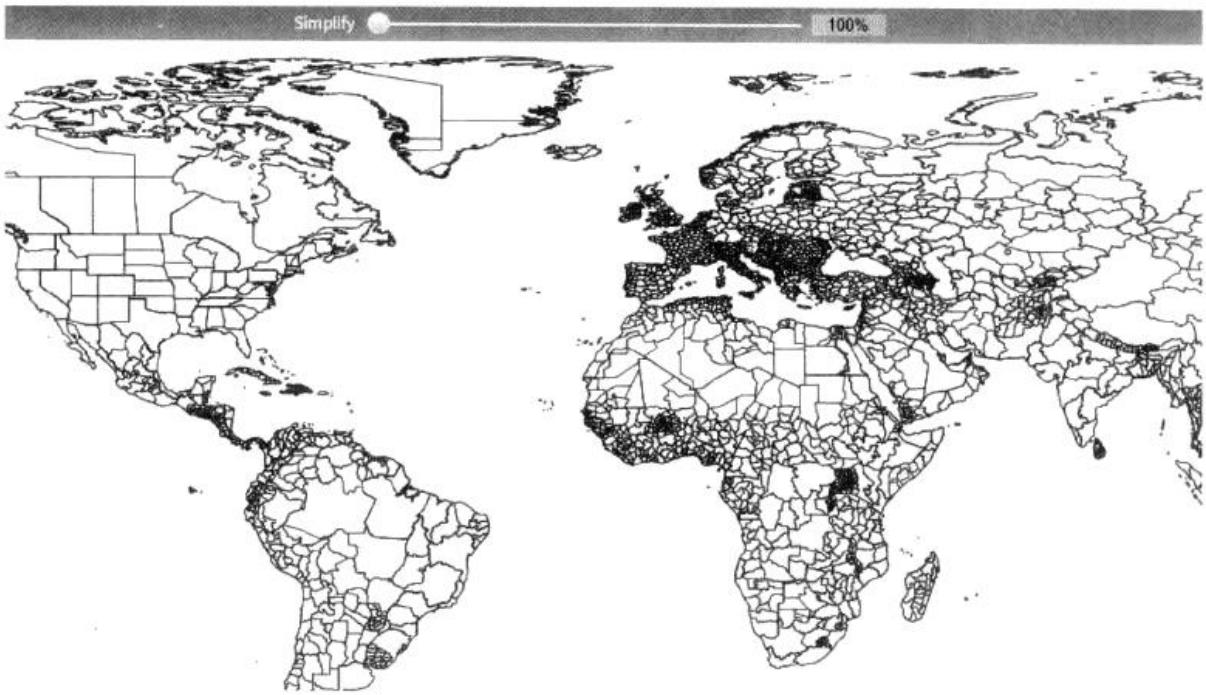


图 11-7　shp 文件里的世界地图

在网页的上方有一个“Simplify”设置，可选择简化的比例。将其滑动至 1%，则地图变为图 11-8 所示形状。

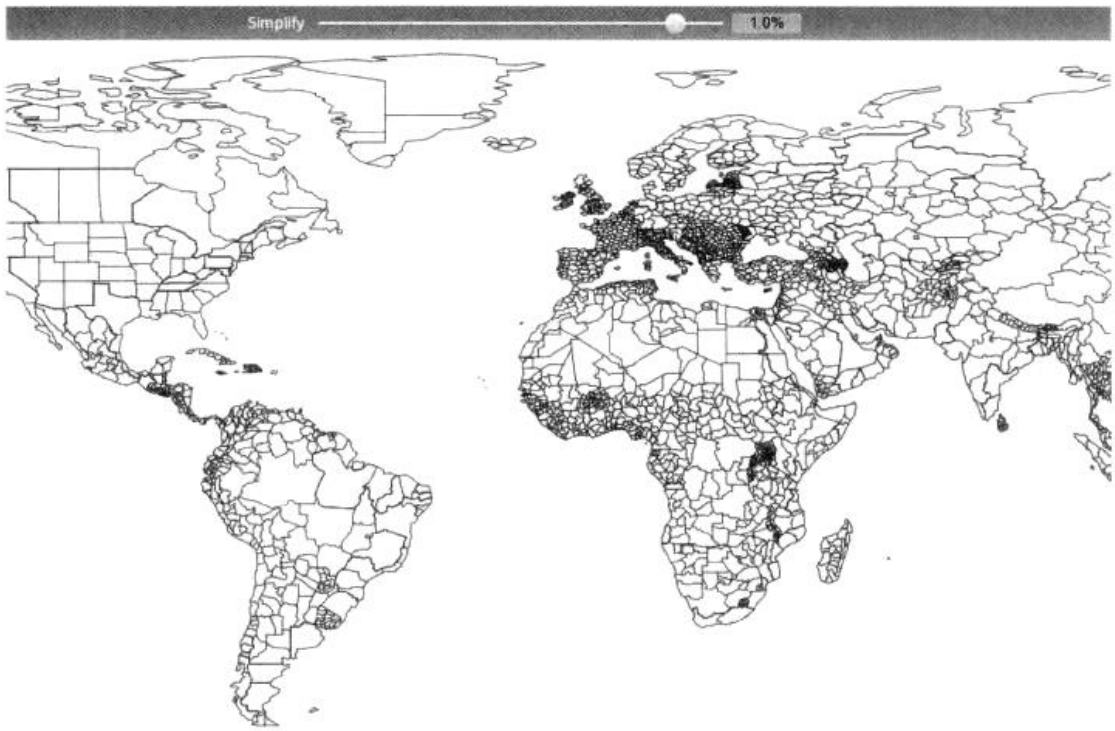


图 11-8　简化为原来的 1%

网页的右上方，有三个按钮，如图 11-9 所示，可选择不同的输出格式。

图 11-9　选择输出格式

可以先在 mapshaper.org 简化 shp 文件，再用上一节的方法提取局部的地图（例如中国）；也可以先用上一节的方法提取局部的地图，再在 mapshaper.org 上简化。最后，保存为 GeoJSON 或 TopoJSON 格式。

11.1.3　GeoJSON

GeoJSON 是用于描述地理空间信息的数据格式。GeoJSON 不是一种新的格式，其语法规范是符合 JSON 格式的，只不过对其名称进行了规范，专门用于表示地理信息。GeoJSON 里的对象也是由**名称/值对**的集合构成，名称总是字符串，值可以是字符串、数字、布尔值、对象、数组、null（参见第 9.1.1 节关于 JSON 格式的说明）。

GeoJSON 的最外层是一个单独的对象（object）。这个对象可表示：

- 几何体（Geometry）。
- 特征（Feature）。
- 特征集合（FeatureCollection）。

最外层的 GeoJSON 里可能包含有很多子对象，每一个 GeoJSON 对象都有一个 type 属性，表示对象的类型，type 的值必须是下面之一。

- Point：点。
- MultiPoint：多点。
- LineString：线。
- MultiLineString：多线。
- Polygon：面。
- MultiPolygon：多面。
- GeometryCollection：几何体集合。
- Feature：特征。
- FeatureCollection：特征集合。

下面举几个例子。

点对象：

```
{
    "type": "Point",
    "coordinates": [ -105, 39 ]
}
```

线对象：

```
{
    "type": "LineString",
    "coordinates": [[-105, 39 ], [-107, 38 ]]
}
```

面对象：

```
{
    "type": "Polygon",
    "coordinates":[[ [30, 0], [31, 0], [31, 5], [30, 5], [30, 0] ]]
}
```

由以上格式可以发现，每一个对象都有一个成员变量 coordinates。如果 type 的值为 Point、MultiPoint、LineString、MultiLineString、Polygon、MultiPolygon 之一，则该对象必须有变量 coordinates。

如果 type 的值为 GeometryCollection（几何体集合），那么该对象必须有变量 geometries，其值是一个数组，数组的每一项都是一个 GeoJSON 的几何对象。例如：

```
{
    "type": "GeometryCollection",
    "geometries": [
        {
            "type": "Point",
            "coordinates": [100, 40]
        },
        {
            "type": "LineString",
            "coordinates": [ [100, 30], [100, 35] ]
        }
    ]
}
```

如果 type 的值为 Feature（特征），那么此特征对象必须包含有变量 geometry，表示几何体，geometry 的值必须是几何体对象。此特征对象还包含有一个 properties，表示特性，properties 的值可以是任意 JSON 对象或 null。例如：

```
{
    "type": "Feature",
    "properties": {
        "name": "北京"
    },
    "geometry": {
        "type": "Point",
```

```
        "coordinates": [ 116.3671875, 39.977120098439634]
    }
}
```

如果 type 的值为 FeatureCollection（特征集合），则该对象必须有一个名称为 features 的成员。features 的值是一个数组，数组的每一项都是一个特征对象。第 11.1.1 节制作的 china.geojson，就是一个特征集合，其内容形如（**为使其容易理解，修改省略了部分内容，只保留其大致结构**）：

```
{
    "type": "FeatureCollection",
    "features": [
    {
        "type": "Feature",
        "properties": { "id": 1 , "name": "甘肃" },
        "geometry": {"type": "Polygon", "coordinates": [ ... ]}
    },
    {
        "type": "Feature",
        "properties": { "id": 2 , "name": "青海" },
        "geometry": {"type": "Polygon", "coordinates": [ ... ]}
    },
    {
        "type": "Feature",
        "properties": { "id": 3 , "name": "广西" },
        "geometry": {"type": "Polygon", "coordinates": [ ... ]}
    },
    /*** 以下省略 ***/
    ]
}
```

这样就很清楚了，china.geojson 地图文件里，只有一个对象，该对象的 type 值为特征集合（FeatureCollection）。其成员 features 的每一项描述一个省的地理信息，每一项都是一个特征对象（Feature），特征对象的 properties 里包含有该省的名称、id 号等，geometry 里包含有此省份的地理信息。

如果从 Natural Earth 上下载的数据不能符合要求，可以手动制作 GeoJSON 文件，但是手动输入很烦琐，还必须查询经纬度等信息。这时候，可以求助于一些在线生成 GeoJSON 的工具。例如，打开下面网址：

http://geojson.io/

打开后，如图 11-10 所示，可以在地图上查找需要的区域。地图右上角有工具栏，工具栏右方是自动生成的 GeoJSON 对象。例如，选择工具栏最下方的“Draw a marker”工具，在上海的位置点一下，会自动生成以下的 GeoJSON 对象：

```
{
    "type": "FeatureCollection",
    "features": [
    {
        "type": "Feature",
        "properties": {},
        "geometry": {
        "type": "Point",
        "coordinates": [
            121.46484375,
            31.31610138349565
        ]
        }
        }
    ]
}
```

在自动生成的文本的基础上修改，能省去不少时间，也能防止格式上的错误。另外，该对象没有添加“特性”属性（properties 的值），需要手动添加。

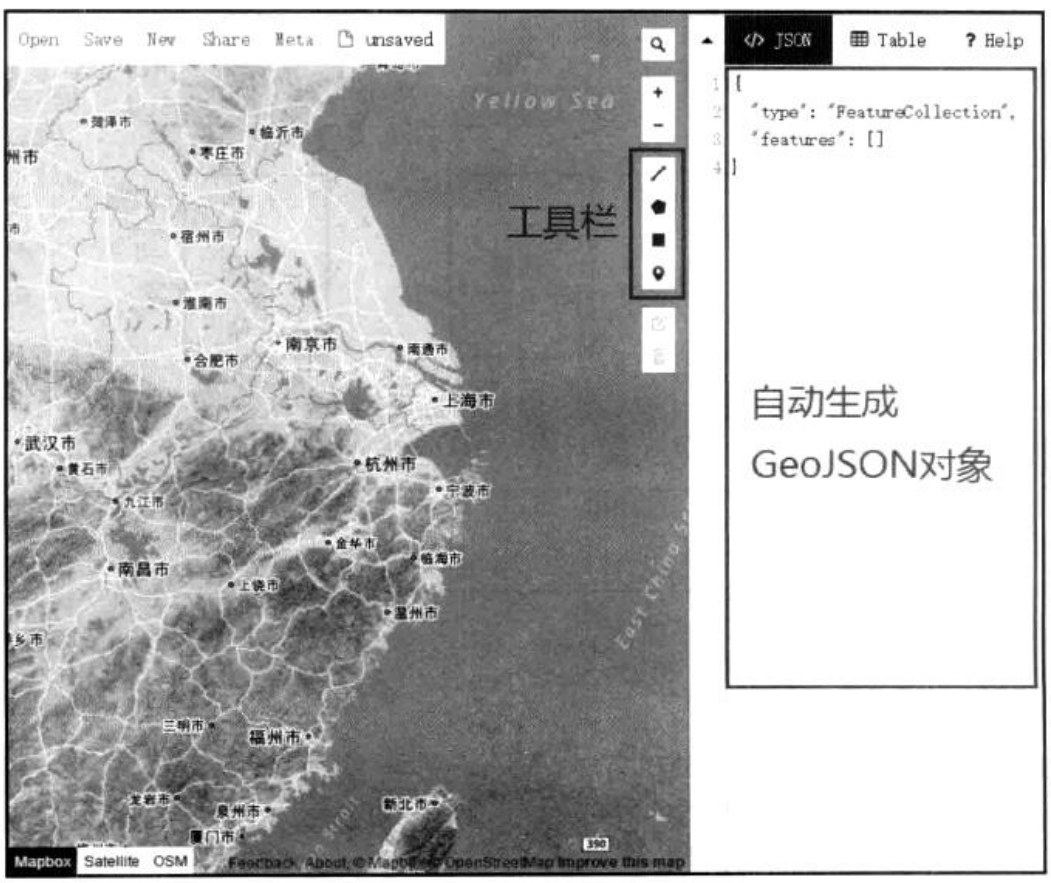


图 11-10　geojson.io 自动生成 GeoJSON 对象

11.1.4 TopoJSON

TopoJSON 是 GeoJSON 按拓扑学编码后的扩展形式，是由 D3 的作者 Mike Bostock 制定的。相比 GeoJSON 直接使用 Polygon、Point 之类的几何体来表示图形的方法，TopoJSON 中的每一个几何体都是通过将**共享边（被称为 arcs）**整合后组成的。

TopoJSON 消除了冗余，文件大小缩小了 80%，因为：

- 边界线只记录一次（例如广西和广东的交界线只记录一次）。
- 地理坐标使用整数，不使用浮点数。

图 11-11 展示了 TopoJSON 格式中的几何体是如何形成的。TopoJSON 里有一个名称为 arcs 的公共数组，保存有地图里所有需要使用的边。地图中的几何体“西藏”、“新疆”、“甘肃”都分别从 arcs 里提取自己需要的边来组成自己的边界。“西藏”和“新疆”是接壤的，因此共用 arcs 数组下标为 1 的项，“甘肃”和“新疆”也是接壤的，因此共用下标为 4 的项。通过这种方式，TopoJSON 大大缩减了数据量。

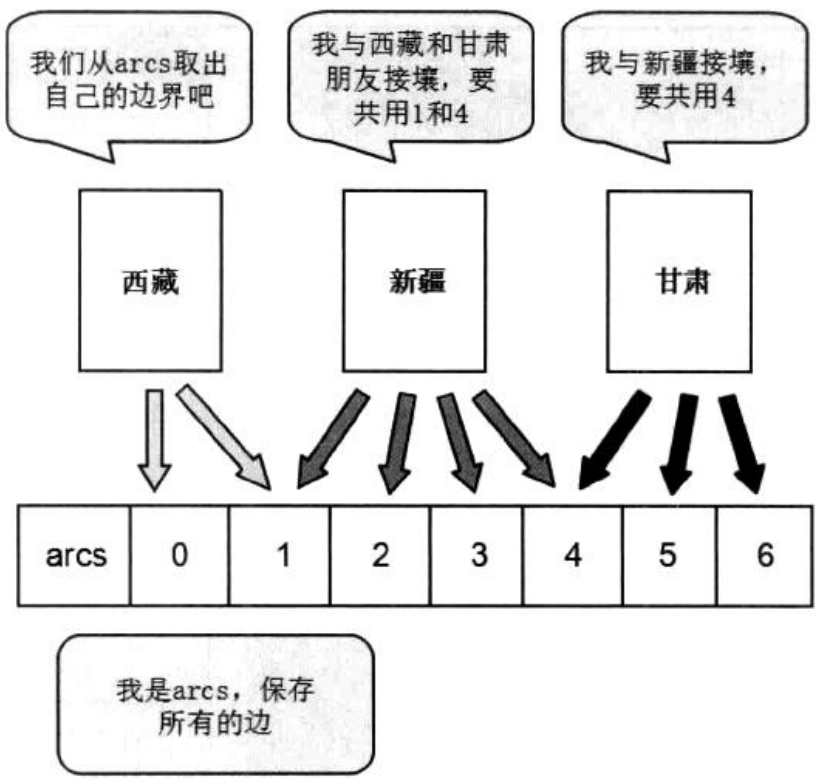


图 11-11　TopoJSON 格式中几何体的构成方法

下面看一个 TopoJSON 文件，该文件描述了加勒比海地图的一个岛屿阿鲁巴（aruba）的地形：

```
{
    "type": "Topology",
    "transform": {
        "scale": [0.036003600360036005, 0.017361589674592462],
        "translate": [-180, -89.99892578124998]
    },
    "objects": {
        "aruba": {
            "type": "Polygon",
            "arcs": [[0]],
            "id": 533
        }
    },
    "arcs": [[[3058, 5901], [0, -2], [-2, 1], [-1, 3], [-2, 3], [0, 3], [1,
1], [1, -3], [2, -5], [1, -1]]]
}
```

type 的值是 Topology，表示文件类型。transform 用于描述缩放量和平移量，分别用一个只有两项的数组来表示。objects 里存有几何体模块，此处只有阿鲁巴岛（aruba）。aruba 里的 type 和 GeoJSON 一样表示几何体类型，arcs 表示如何从最外层的数组 **arcs** 里提取地形。

11.2　中国地图

基于 GeoJSON 格式和 TopoJSON 格式都可以绘制地图。由于 TopoJSON 只相当于 GeoJSON 的 20%大小，因此在 D3 的应用中尽可能使用 TopoJSON。但是，由于 TopoJSON 的标准只是由 D3 的作者制定的，目前还不是世界范围内承认的标准。

通过第 11.1 节的方法，制作两个中国地图的 JSON 文件。

- china.geojson，250KB
- china.topojson，41KB

分别是 GeoJSON 格式和 TopoJSON 格式的，本节将分别使用这两个文件来制作地图。

11.2.1　基于 GeoJSON

中国地图通常是这样表示的：在一个矩形区域内，画上中国大陆及港澳台，矩形的右下角添加一个小框，里面绘制南海诸岛。由于南海诸岛绘制在一个矩形框内，需要单独绘制。可使用以下两种方法。

- 将南海诸岛保存成一张图片，然后贴到 SVG 里。
- 将南海诸岛的地理信息写成 SVG 元素，保存在一个 svg 文件里。

本例使用第二种方法，好处是：SVG 是矢量图，将其进行放大缩小不会损失质量。那么，接下来要使用以下两个文件。

- china.geojson，中国大陆及港澳台。
- southchinasea.svg，表示南海诸岛的矩形。

其中，json 文件使用 d3.json 读取，svg 文件使用 d3.xml 读取（参见第 9.1 节）。

首先，定义**地图的投影和地理路径生成器**：

```
//定义地图的投影
var projection = d3.geo.mercator()
                   .center([107, 31])
                   .scale(600)
                   .translate([width/2, height/2]);

//定义地理路径生成器
var path = d3.geo.path()
                .projection(projection);    //设定投影
```

这段代码中，d3.geo.mercator()是一种投影方法，关于投影的详细介绍在第 11.4 节。center()设定地图的中心位置，107 是经度，31 是纬度。scale 和 translate 分别设置缩放量和平移量。定义好投影之后，在 d3.geo.path()中应用此投影。如此，在使用地理路径生成器时，每一个坐标都会先调用此投影函数，然后才生产路径值。

然后，通过 d3.json 请求文件 china.geojson，并添加足够数量的<path>，每一个<path>用于绘制一个省的路径。

```
//颜色比例尺
var color = d3.scale.category20();

//请求china.geojson
d3.json("china.geojson", function(error, root) {
   if(error)
       return console.error(error);
   console.log(root);       //所读取到的地图信息保存在root里

   var groups = svg.append("g");

   groups.selectAll("path")
           .data( root.features )
           .enter()
           .append("path")
           .attr("class","province")
           .style("fill", function(d,i){
               return color(i);
           })
           .attr("d", path );  //使用路径生成器
});
```

china.geojson 文件的内容保存在变量 root 里，在控制台输出后，结果如图 11-12 所示。root 里有一个数组 features，数组的每一项就是一个省的地理信息。

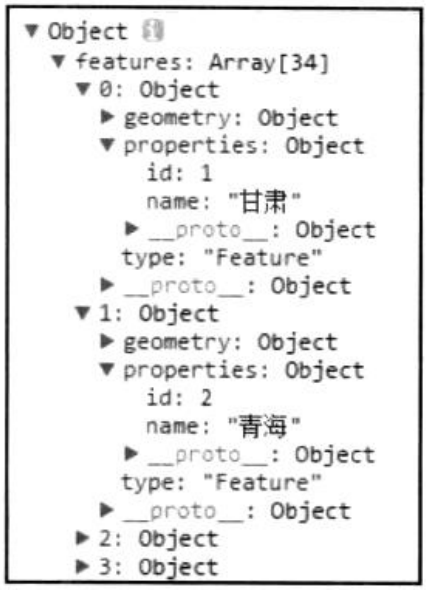


图 11-12　root 的结构

本例所绘制的中国地图，其 SVG 结构如下所示。

```
<g>
    <path>甘肃的路径</path>
    <path>青海的路径</path>
    /*** 省略 ***/
</g>
```

一个最外层的分组元素<g>，里面包含若干<path>，每一个<path>代表一个省，故要让<path>的选择集绑定数组 root.features。因此，要使用如下的代码：

```
var groups = svg.append("g");

groups.selectAll("path")
        .data( root.features )
        .enter()
        .append("path");
```

最后，绘制中国南海诸岛，通过 d3.xml 请求文件 southchinasea.svg，代码如下：

```
//请求southchinasea.svg
d3.xml("southchinasea.svg", function(error, xmlDocument) {
    svg.html(function(d){
        return d3.select(this).html() +
              xmlDocument.getElementsByTagName("g")[0].outerHTML;
    });

    d3.select("#southchinasea")
        .attr("transform","translate(540,410)scale(0.5)")
```

```
        .attr("class","southchinasea");
});
```

由于 southchinasea.svg 里直接保存的就是 SVG 元素，所以只需要在当前 svg 对象的 html 属性后，添加 southchinasea.svg 里的内容即可。即将以下两者的字符串相加即可。

- d3.select(this).html()

当前 SVG 文档的 innerHTML，即内部 HTML 内容。

- xmlDocument.getElementsByTagName("g")[0].outerHTML

请求的 svg 文件中第一个<g>元素的 outerHTML，即外部 HTML 内容。

添加了南海诸岛的 svg 元素后，再将其移动到合适的位置，并添加样式。各省份和南海诸岛的样式如下：

```
.province {
    stroke: black;
    stroke-width: 1px;
}

.southchinasea {
    stroke: black;
    stroke-width: 1px;
    fill: red;
}
```

中国地图的制作结果如图 11-13 所示。

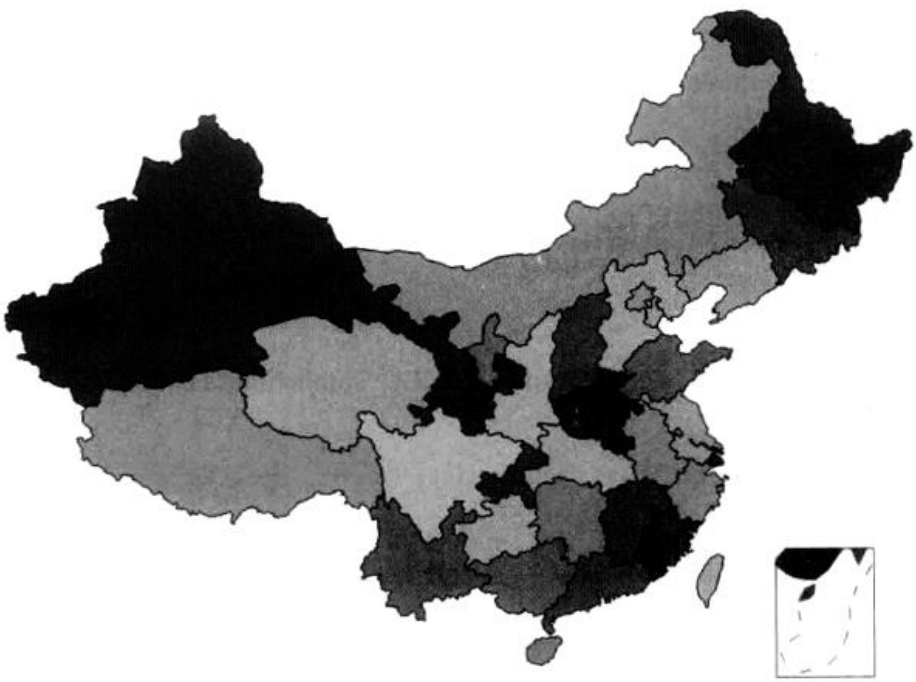


图 11-13　请求 GeoJSON 文件后绘制的中国地图

HTML 的文档结构如图 11-14 所示，<svg>里有两个<g>，分别用来容纳南海诸岛和中国各省的图形元素。

```
▼<html>
  ▶<head>…</head>
  ▼<body>
     <script src="../d3/d3.min.js"></script>
    ▶<script>…</script>
    ▼<svg width="700" height="700">
      ▶<g xmlns="http://www.w3.org/2000/svg" id="southchinasea"
      class="southchinasea">…</g>
      ▶<g>…</g>
      </svg>
    </body>
  </html>
```

图 11-14　HTML 的文档结构

11.2.2　基于 TopoJSON

Mike Bostock 提供了解析 TopoJSON 格式的 API，有两部分：**客户端应用**和**服务端应用**。客户端 API 支持将 TopoJSON 转换为 GeoJSON，以便于显示在浏览器上；服务端 API 支持将其他格式转换为 TopoJSON。本节只讲述客户端 API。

下面的网址可以有下载链接：

https://github.com/mbostock/topojson

下载到 topojson.js 文件后，再在 HTML 中的<script>处引用。还可以直接通过网络引用：

<script src="http://d3js.org/topojson.v1.min.js"></script>

与 d3.js 一样，topojson.js 不支持 IE 8 以下（包括 IE8）的浏览器。

1. 绘制中国地图

下面制作一个与图 11-13 相同的中国地图，要求使用 TopoJSON 格式的地图文件。将要使用的文件有两个：

- china.topojson，由 china.geojson 转换而来。
- southchinasea.svg，表示南海诸岛的矩形。

本节所需的投影函数和地理路径生成器，与上一节相同，不再重复。

首先，通过 d3.json()请求文件 china.topojson，并对文件进行相关处理，代码如下：

```
d3.json("china.topojson", function(error, toporoot) {
    if (error)
        return console.error(error);
```

```
    //输出china.topojson的对象
    console.log(toporoot);

    //将TopoJSON对象转换成GeoJSON，保存在georoot中
    var georoot = topojson.feature(toporoot,toporoot.objects.china);

    //输出GeoJSON对象
    console.log(georoot);
});
```

topojson.feature()是 topojson.js 中的一个方法，相关说明如下。

- topojson.**feature**(topology, object)

返回 GeoJSON 的特征（Feature）或特征集合（FeatureCollection），第一个参数 topology 是 TopoJSON 文件的对象，第二个参数 object 是 TopoJSON 对象中表示几何体的一个对象。

在上面的代码中，在控制台分别输出了 toporoot 和 georoot，后者是前者经过 topojson.feature() 转换后的结果。输出结果如图 11-15 所示，上半部分是对象 toporoot，下半部分是 georoot。可以看到，TopoJSON 对象中，有数组 arcs（保存共享边），对象 objects（保存几何体对象），objects 中有一个对象 china，保存中国地图的几何体。因此，topojson.feature()的参数被设定为：

```
topojson.feature(toporoot, toporoot.objects.china);
```

该函数返回的 GeoJSON 对象保存到变量 georoot 里，在图 11-15 可以看到 georoot 对象是一个特征集合（FeatureCollection）。

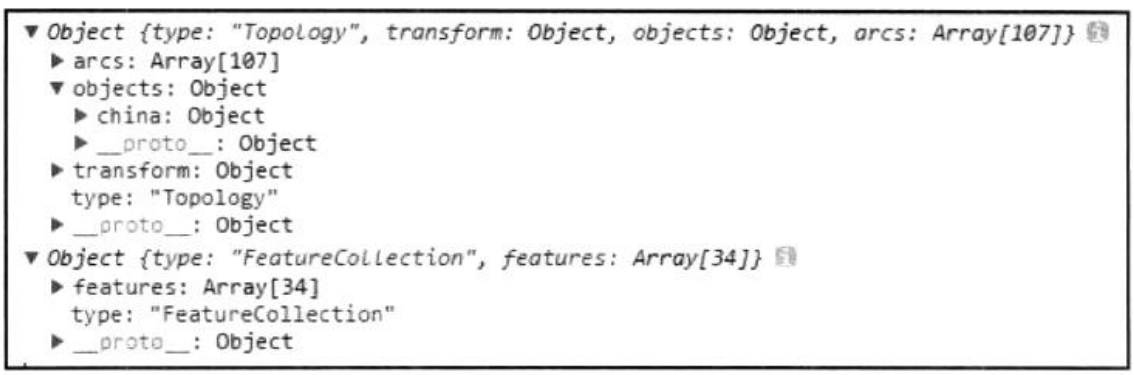


图 11-15　变量 toporoot 和 georoot 的输出

因此实际上，在绘制地图时，还是使用了 GeoJSON 对象。之后的代码与上一节相同，在 d3.json()里添加如下代码：

```
var groups = svg.append("g");

groups.selectAll("path")
          .data( georoot.features )
          .enter()
```

```
        .append("path")
        .attr("class","province")
        .style("fill", function(d,i){
            return color(i);
        })
        .attr("d", path );
```

绘制结果如图 11-16 所示，与图 11-13 是一样的。虽然结果一致，但是由于 TopoJSON 文件比 GeoJSON 文件小很多（缩减了 80%），因此使用 d3.json()请求文件的时间会大大缩短。在地图文件比较大、比较多的时候是非常有利的。

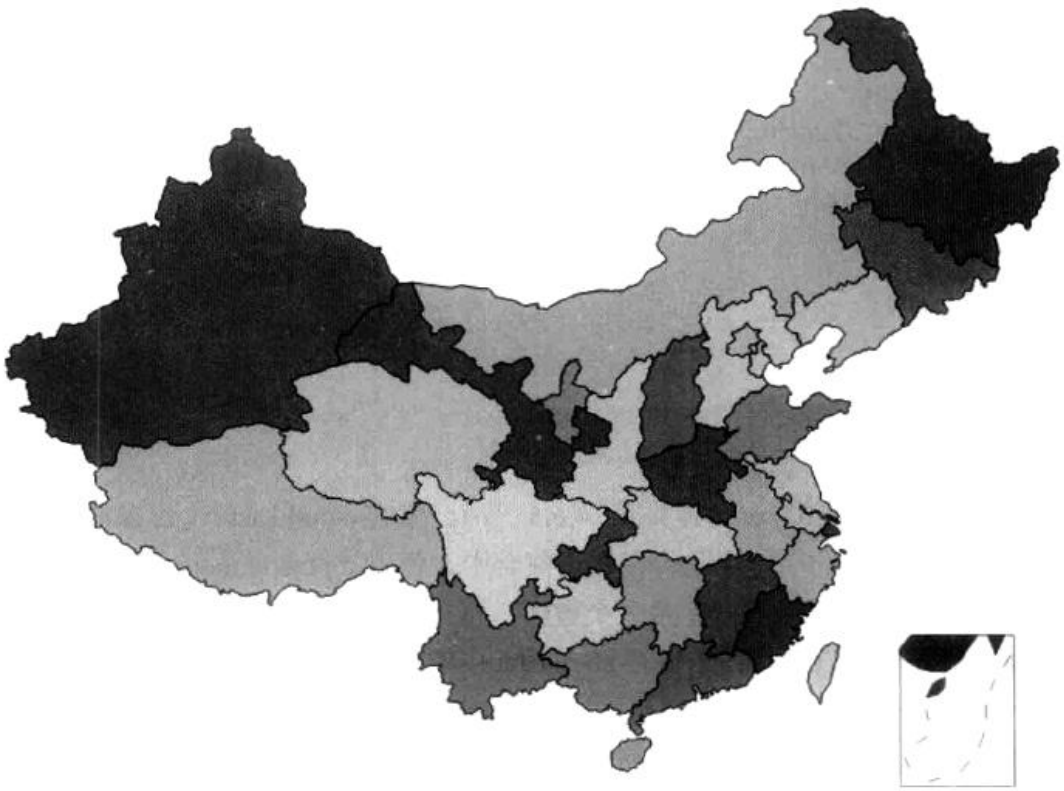


图 11-16　请求 TopoJSON 文件绘制的中国地图

2. 合并地区

除了读取速度比较快之外，使用 TopoJSON 还可以很方便地实现许多有趣的功能。例如，有下列方法。

- topojson.**merge**(topology, objects)

返回 GeoJSON 的类型为 MultiPolygon 的几何对象。objects 是一个数组，包含 Polygon 和 MultiPolygon 对象。此函数能将 objects 中的集合体组合而成一个整体。

现在要制作一幅地图，将中国的“东南地区”作为一个整体表示。将东南地区各省份合并

后，用蓝色表示，其他部分用灰色表示。

请求 china.topojson 文件，代码如下：

```
d3.json("china.topojson", function(error, toporoot) {

    /** 调用topojson.merge将TopoJSON格式的对象（topojson）
    转换成GeoJSON格式的对象（georoot）。 **/

    //东南各省名称的集合
    var southeast = d3.set([
        "广东","海南","福建","浙江","江西",
        "江苏","台湾","上海","香港","澳门"
    ]);

    //合并东南各省
    var mergedPolygon = topojson.merge(toporoot,
            toporoot.objects.china.geometries.filter(
                function(d){
                    return southeast.has(d.properties.name);
                })
            );

    //输出合并结果
    console.log(mergedPolygon);
});
```

这段代码使用了集合 d3.set（详见第 4.6.5 节）。集合 southeast 包含东南各省的名称，southeast.has()能检查传入的字符串是否存在于集合中。使用 topojson.merge()合并东南各省时，由于 toporoot.objects.china.geometries 是一个数组，每一项保存一个省的几何信息。然后通过数组对象的 filter()函数，使只有存在于集合 southeast 中的省份才能留下来。合并后的几何体对象保存在变量 mergedPolygon 里。在控制台输出 mergedPolygon，结果如图 11-17 所示，可见几何体的类型是 MultiPolygon。不必理会几何体对象的具体内容是什么，图形的生成 d3.geo.path()会为我们完成。

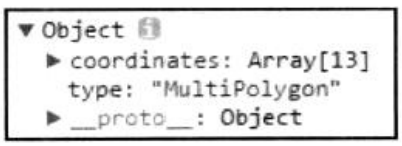


图 11-17　合并后的几何体对象

接下来分别绘制“**东南地区**”和“**其他省份**”。为便于观察，将东南地区设定成蓝色，其余部分设置为灰色。对于东南地区之外的部分，使用数组 georoot.features，其每一项代表一个省份，要用 filter()过滤，只保留“其他省份”，绑定数据时使用 **data()**。对于东南各省，使用合并后的几何体对象 mergedPolygon，要用 **datum()**绑定数据，因为只需要一个路径元素<path>即可，

因为“东南地区”的数据是一个整体。

```
//绘制除了东南各省之外的中国地图，颜色为灰色
var groups = svg.append("g");

groups.selectAll("path")
      .data( georoot.features.filter(function(d){
            return !southeast.has(d.properties.name);
      }))
      .enter()
      .append("path")
      .attr("class","province")
      .style("fill", "#ccc")
      .attr("d", path );

//绘制东南各省，颜色为蓝色
svg.append("path")
       .datum( mergedPolygon )
       .attr("class","province")
       .style("fill","blue")
       .attr("d", path );
```

效果如图 11-18 所示，当需要将某些地区作为一个整体来看的时候，可考虑使用 topojson.merge()。

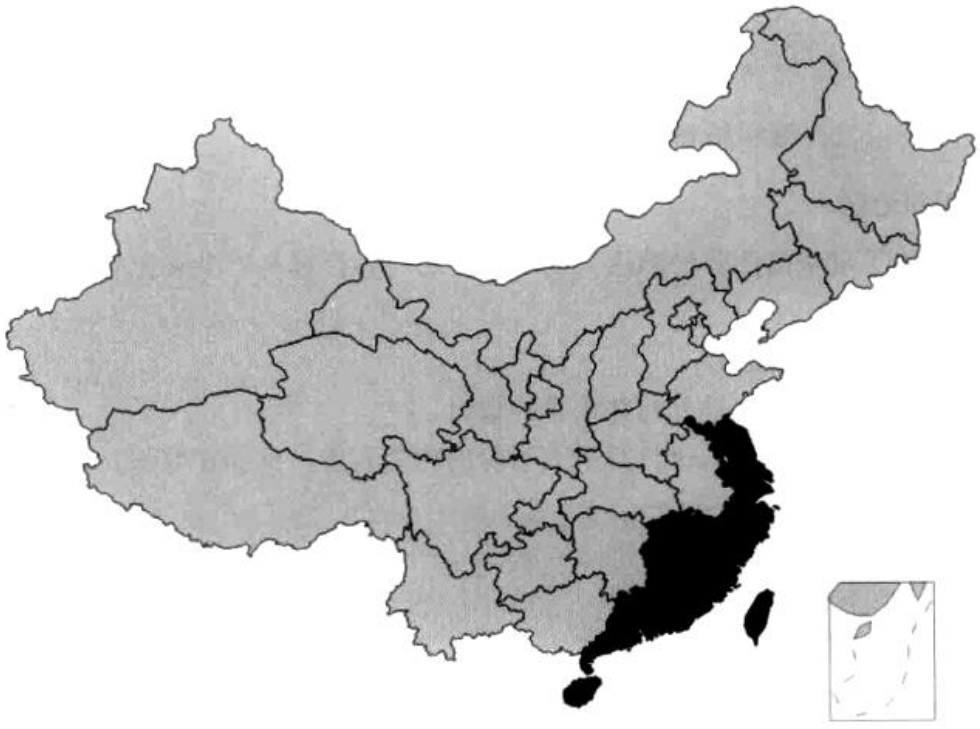


图 11-18　合并东南各省

3．绘制边界线

使用 TopoJSON 可以获取相邻地区的边界线，请看以下方法。

- topojson.mesh(topology, object, [filter])

返回 GeoJSON 的 MultiLineString 的几何体。可选参数 filter 是一个过滤器，该过滤器是一个函数 function(a,b)，有两个参数 a 和 b，分别代表两个几何体。

例如，要获取西藏和新疆的边界线，并在地图上标示，代码如下：

```
var boundary = topojson.mesh(toporoot, toporoot.objects.china,
                    function(a, b) {
                        return a.properties.name === "西藏" &&
                    b.properties.name === "新疆";
                     });

console.log(boundary);
```

当几何体 a 的名称为“西藏”，b 为“新疆”时，提取边界线。边界线的结果保存在变量 border 里，这是一个类型为 MultiLineString 的几何体。

在<svg>里添加一个路径元素<path>，用 datum()绑定变量 boundary，再调用地理路径生成器获取路径即可，代码如下：

```
svg.append("path")
      .datum(boundary)
      .attr("class", "boundary")
      .attr("d", path);
```

如图 11-19 所示，西藏和新疆的交界线被用粗线标示出来了。

4．查找相邻地区

TopoJSON 除了可获取两省份的边界线之外，还可以计算与一个省份相邻的省份，这就要用到以下方法。

- topojson.neighbors(objects)

获取相邻几何体，objects 是几何体集合的数组。

首先，通过 topojson.neighbors 计算所有省份的相邻省份，保存在数组 neighbors 里：

```
var neighbors = topojson.neighbors(
                    toporoot.objects.china.geometries);

console.log(neighbors);
```

数组 neighbors 保存有各省份的邻省序号，在控制台的输出如图 11-20 所示。例如，neighbors[0]的内容为序号 0 的省份的邻省，由图可知分别为 1、10、12、13、17、29，这些都是与 0 相邻的省份的序号。

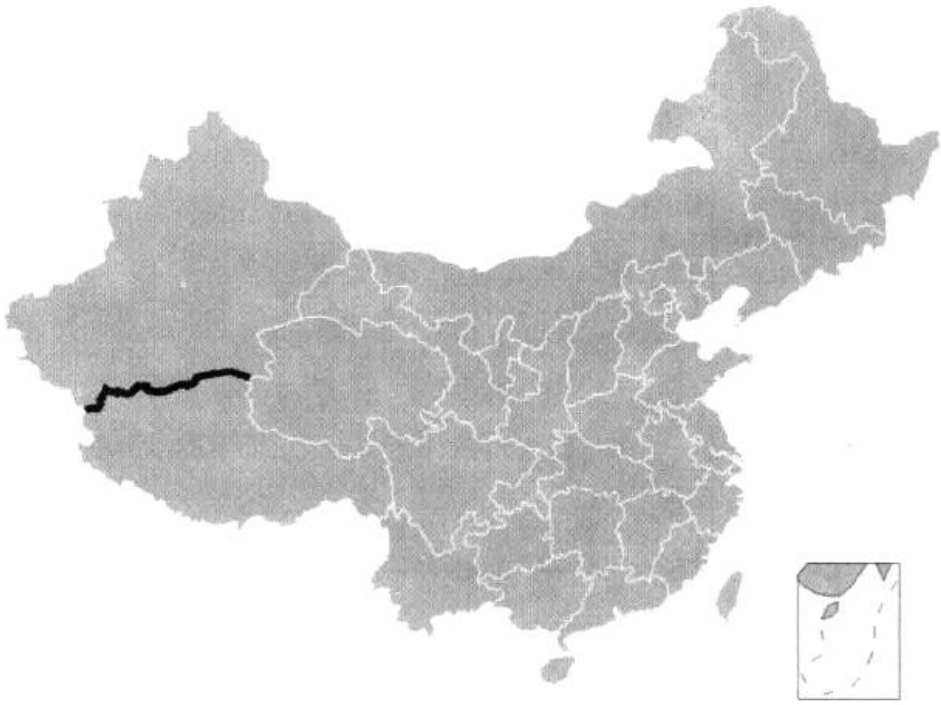


图 11-19　西藏和新疆的交界线

```
▼ [Array[6], Array[4], Array[4], Array[5], Array[5],
  Array[8], Array[4], Array[6], Array[6], Array[7],
  Array[2], Array[5], Array[3], Array[8], Array[2],
   ▼ 0: Array[6]
       0: 1
       1: 10
       2: 12
       3: 13
       4: 17
       5: 29
       length: 6
     ▶ __proto__: Array[0]
   ▶ 1: Array[4]
   ▶ 2: Array[4]
   ▶ 3: Array[5]
   ▶ 4: Array[5]
   ▶ 5: Array[2]
   ▶ 6: Array[3]
   ▶ 7: Array[6]
   ▶ 8: Array[5]
   ▶ 9: Array[4]
   ▶ 10: Array[3]
   ▶ 11: Array[0]
```

图 11-20　数组 neighbors 的输出

然后，绘制地图，并加入交互式操作：令当鼠标移动到某一省份时，该省份的颜色变为红色，相邻省份变为蓝色，鼠标移出时变回原来的颜色。代码如下：

```
var groups = svg.append("g");

var paths = groups.selectAll("path")
                  .data( georoot.features )
```

```
                    .enter()
                    .append("path")
                    .attr("class","province")
                    .style("fill", "#ccc")
                    .attr("d", path );

paths.each(function(d,i){
        //为每一个元素添加相邻省份的选择集
        d.neighbors = d3.selectAll(
            neighbors[i].map(function(j) {
                return paths[0][j];
            })
        );
    })
    .on("mouseover",function(d,i){
        //鼠标移入后，变色
        d3.select(this).style("fill","red");
        d.neighbors.style("fill","steelblue");
    })
    .on("mouseout",function(d,i){
        //鼠标移出后，变回原来的颜色
        d3.select(this).style("fill","#ccc");
        d.neighbors.style("fill", "#ccc");
    });
```

结果如图 11-21 所示，鼠标移到“湖北”上，周围的六个省份“陕西”、“河南”、“安徽”、“江西”、“湖南”、“重庆”都变成了蓝色。

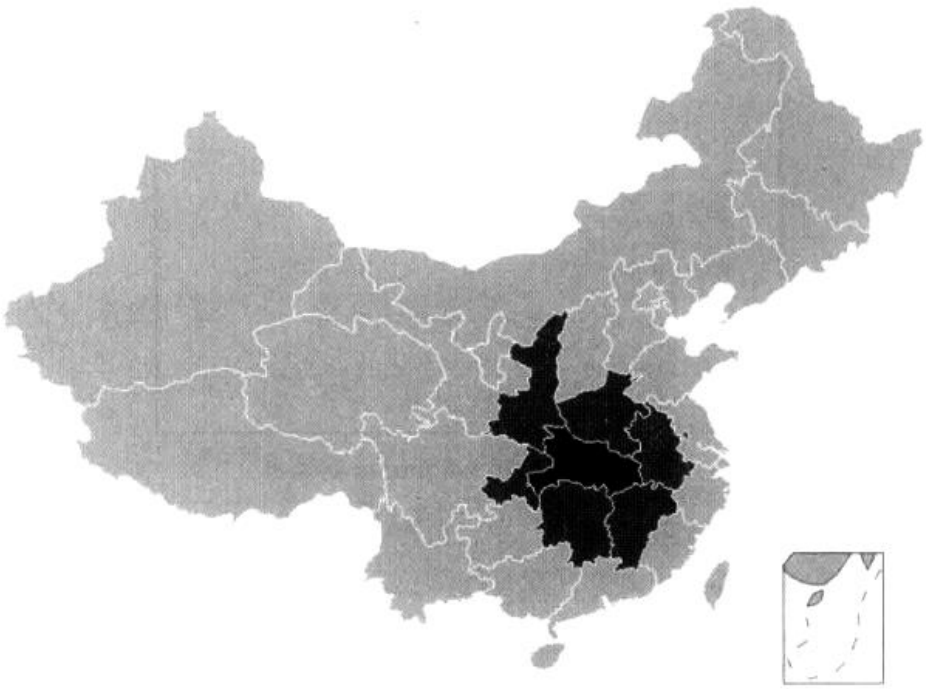


图 11-21　相邻省份

11.3　地理路径

第 6 章提到过生成器的概念，当时介绍了线段生成器、区域生成器、弧生成器等。要绘制地图也要用到生成器，因为地图也是使用路径元素<path>。前面已多次使用的 d3.geo.path()是**地理路径生成器**，是绘制地图的核心组件。此外，还有两种形状生成器：d3.geo.graticule()和 d3.geo.circle()，前者用于绘制经线和纬线，后者用于以某一地点为中心绘制圆形网格。

11.3.1　地理路径生成器

地理路径生成器（Geographic Path Generator），能通过 GeoJSON 文件（或 TopoJSON 文件）生成地图的路径值，将该路径赋值给 SVG 中的<path>元素，即可绘制地图。

与地理路径生成器相关的方法，分为三种。

第一，创建和生成

- d3.geo.**path**()

创建一个地理路径生成器，默认使用 albersUsa 投影，点半径为 4.5 像素。

- **path**(feature[, index])

根据指定的特征（参数 feature）返回路径字符串，feature 可以是任意 GeoJSON 的特征或几何体对象，其中包括 Point、MultiPoint、LineString、GeometryCollection、Feature、FeatureCollection 等。

第二，设置

- path.**projection**([projection])

设定投影方式，默认为 albersUsa。

- path.**context**([context])

设定渲染的上下文。如果不设定，默认为 null，则该生成器返回 SVG 路径字符串。如果设定，则路径生成器会根据指定的上下文来渲染几何图形。例如可以将 Canvas 的上下文传给此函数，则可以直接在 canvas 上作图。

- path.**pointRadius**([radius])

设定点半径，在显示 Point 和 MultiPoint 时使用。

第三，计算

- path.**area**(feature)

计算指定几何体的投影面积，单位是“平方像素”。如果 feature 的类型为 Point、MultiPoint、LineString、MultiLineString，则面积为 0；如果类型为 Polygon 和 MultiPolygon，则先计算外部

环的面积，然后减去内部的空洞。

- path.**centroid**(feature)

计算几何体的中心，单位是像素。在给地图加标签时很有用。

- path.**bounds**(feature)

计算几何体的边界框，单位是像素。

d3.geo.path()的工作过程如图 11-22 所示。地图的源文件如果是 GeoJSON，则直接使用，如果是 TopoJSON 文件，则先用 topojson.js 转换成 GeoJSON 后再使用。d3.geo.path()提取 GeoJSON 文件的信息，并计算路径，获取的路径直接放到：

```
<path  d="获取的路径"></path>
```

即可。另外，d3.geo.path()可以设置投影、上下文、点半径。其中，投影有关地图的显示方式。上下文通常不用设定，除非需要在 Canvas 上作图。点半径会影响 GeoJSON 对象中的 Point 和 MultiPoint 元素，默认是 4.5 个像素。如果是在 SVG 下作图，则创建生成器的代码形如：

```
var projection = d3.geo.mercator()
                   .center([107, 31])
                   .scale(600)
                   .translate([width/2, height/2]);

var path = d3.geo.path()
              .projection(projection)
              .pointRadius(5);
```

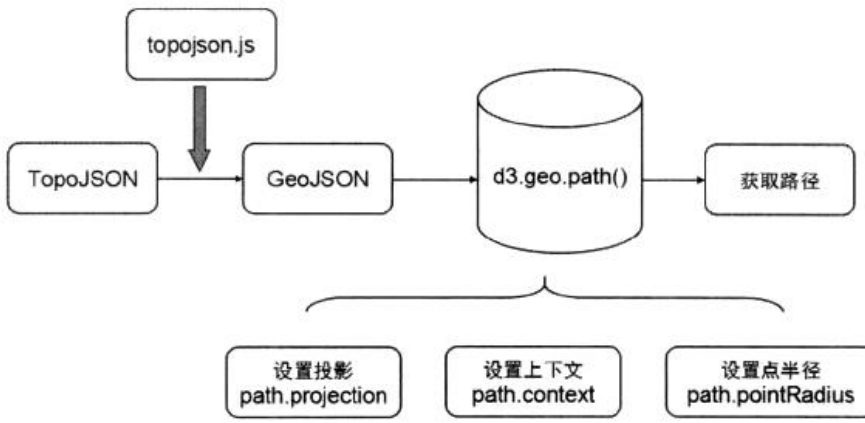


图 11-22　d3.geo.path()的工作过程

d3.geo.path()还另有三个**用于计算**的方法，分别用于计算面积、中心、边界框。下面制作一个示例，来讲解这三个方法。在图 11-13 的基础上修改，要求做到：

鼠标点击某一省份后，显示出该省份的中心和边界框，并在控制台输出面积、中心、边界框的信息。

为使中心和边界框易于观察，将各省份的填充颜色设定为灰色，边框为白色。中心用<circle>元素表示，填充为绿色。边界框使用<rect>元素，不填充，线条为蓝色。代码如下：

```
var groups = svg.append("g");

groups.selectAll("path")
          .data( georoot.features )
          .enter()
          .append("path")
          .attr("class","province")
          .attr("d", path )
          .on("click",function(d){

              //计算面积、中心、边界框
              var area = path.area(d);
              var centroid = path.centroid(d);
              var bounds = path.bounds(d);

              //输出到控制台
              console.log( "省份: " + d.properties.name );
              console.log( "面积: " + area );
              console.log( "中心: " + centroid );
              console.log( "边界框: ");
              console.log( bounds );

              //显示中心
              svg.append("circle")
                  .attr("class","centroid")
                  .attr("cx", centroid[0] )
                  .attr("cy", centroid[1] )
                  .attr("r", 8);

              //显示边界框
              svg.append("rect")
                  .attr("class","boundingbox")
                  .attr("x", bounds[0][0] )
                  .attr("y", bounds[0][1] )
                  .attr("width", bounds[1][0] - bounds[0][0] )
                  .attr("height", bounds[1][1] - bounds[0][1] );
      });
```

要注意 path.centroid(d)的参数，这里的 d 是某个省份的几何体对象，另外，也是被绑定的数组 georoot.features 的其中一项。当然，也是可以像这样调用的：

```
path.centroid( georoot.features[0] );
```

```
path.bounds( geoRoot.features[1] );
```

结果如图 11-23 所示，鼠标分别点击“青海”、“山西”、“湖南”、“吉林”之后，这四个省份都显示中心和边界框。

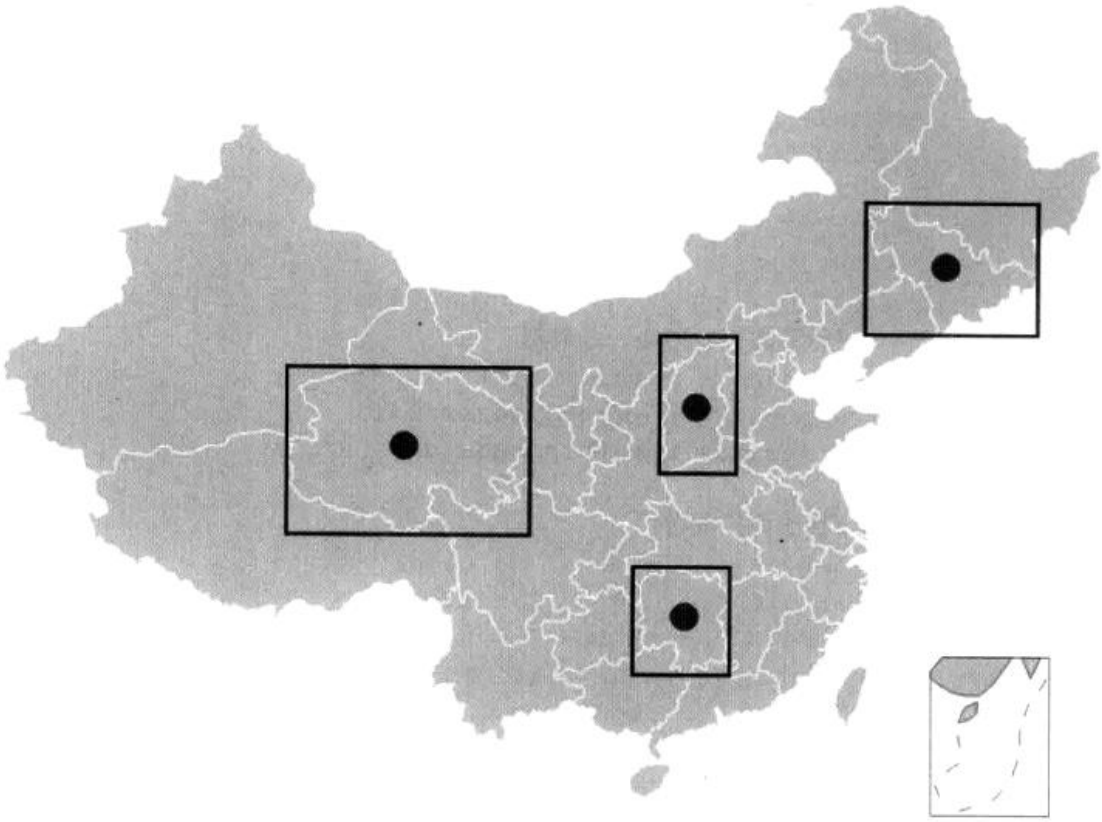


图 11-23　分别点击“青海”、“山西”、“湖南”、“吉林”后显示的中心和边界框

在控制台分别输出了四个省份的相关信息，其中“青海”的内容如图 11-24 所示。面积值显示为 9632.9147，这个值的单位是“平方像素”，即按照当前的投影方式所算出的青海省的面积。中心点坐标是约为[234, 289]。边界框是一个二维数组，各项的意义如下：

[[左上角 x 坐标，左上角 y 坐标],

[右下角 x 坐标，右下角 y 坐标]]

```
省份：青海
面积：9632.914762220687
中心：234.84038533422,289.63206203091204
边界框：
▼ Array[2]
   ▶ 0: Array[2]
   ▶ 1: Array[2]
     length: 2
   ▶ __proto__: Array[0]
```

图 11-24　在控制台输出“青海”的相关信息

11.3.2　形状生成器

形状生成器（Shape Generators）能够在地图上绘制网格，D3 中提供了两种：d3.geo.graticule() 和 d3.geo.circle()。前者生成经线和纬线，后者能够以某一地点为中心生成圆形网格。

经度和纬度不仅是地图的标识，也能带来美观效果。d3.geo.graticule()能够自动生成 GeoJSON 对象，再调用 d3.geo.path，即可简单生成经纬线网格。另外，如果地图和经纬线都使用同一个地理路径生成器，网格和地图能够完美切合。

- d3.geo.**graticule**()

创建一个经纬线网格的生成器。

- **graticule**()

返回一个类型为 MultiLineString 的几何体对象，用于显示所有的经线和纬线。

- graticule.**lines**()

返回一个数组，每一项为 LineString 类型的几何体对象，表示经线或纬线。

- graticule.**outline**()

返回一个类型为 Polygon 的对象，表示网格的轮廓。

- graticule.**extent**([extent])

设定网格的范围，如果不指定，默认值为[[-180, -80],[180, 80]]，即经度从–180°～+180°，纬度从–80°～+80°。

- graticule.**step**([step])

设定网格的步幅，默认为[10, 10]，即经度和纬度都是每隔 10° 就算一次。

下面制作一个例子，要求是：**在世界地图的后面平铺上经纬线网格。**

首先，创建一个网格生成器，并生成网格。

```
var eps = 1e-4;

//创建一个网格生成器，经度范围是-180° ～180° ，纬度从-90° ～90°
var graticule = d3.geo.graticule()
                .extent([[-180, -90],[180+eps, 90]])
                .step([10,10]);

var grid = graticule();     //生成网格数据

console.log(grid);          //输出网格数据
```

设置网格范围时需要在边界处加一个极小值（eps），这是为了防止网格没有边界线。在控制台输出 grid 的结果，如图 11-25 所示，这是一个类型为 MultiLineString 的 GeoJSON 对象。要

注意，graticule()只是**生成数据**，并不绘制，不是说调用 graticule()之后就直接在画板上作图了。第 10 章讲解布局时也是如此，D3 处处都体现了一种思想：**生成数据和绘图是分开的**。

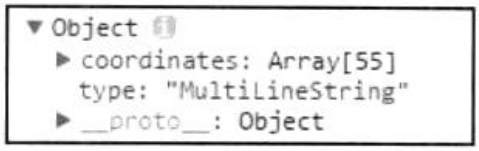


图 11-25　网格数据

其次，定义投影函数和地理路径生成器：

```
var projection = d3.geo.mercator()
                .center([0, 0])
                .scale(60)
                .translate([width/2, height/2]);

var path = d3.geo.path()
                .projection(projection);
```

然后，向<svg>添加一个<path>，并用地理路径生成器生成路径：

```
svg.append("path")
    .datum(grid)
    .attr("class","graticule")
    .attr("d",path);
```

经纬度网格如图 11-26 所示。

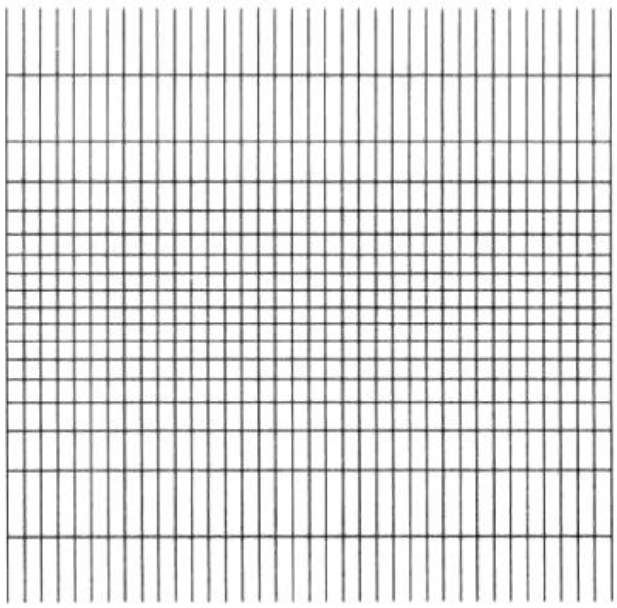


图 11-26　经纬度网格

最后，将世界地图铺在网格上面，如果 11-27 所示。

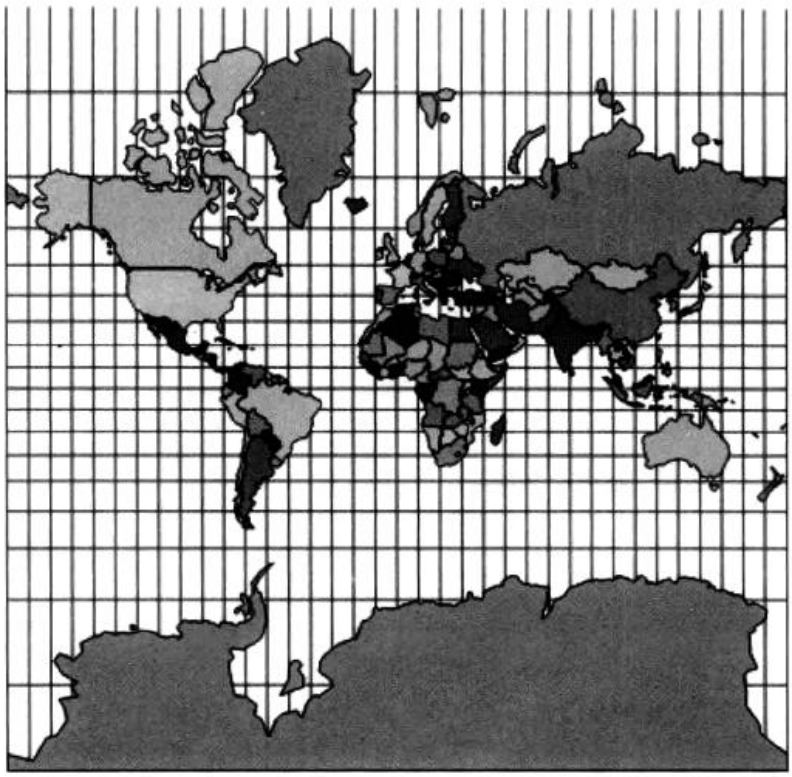


图 11-27　带经纬度网格的世界地图

如果要绘制局部地区，例如中国的经纬线的网格，只需要更改 extent()即可。显示局部地区后，可能需要增加网格的密度（调用 step），否则地图上的网格太稀疏。例如：

```
var graticule = d3.geo.graticule()
                  .extent([[71, 16],[137, 54]])
                  .step([5,5]);
```

网格范围限制在经度 71° ～137° ，纬度 16° ～54° 。经度和纬度都每隔 5° 画一条线。结果如图 11-28 所示。

经纬线网格是使用 d3.geo.path()绘制的，根据投影函数的不同，网格表现出来的形态也不同，可能有圆形、扇形、方形等。但是，只要绘制地图和经纬线网格的 d3.geo.path()是同一个，地图和网格都会自动对应。

另一种形状生成器的介绍如下，它用于制作圆形网格。

- d3.geo.**circle**()

创建一个圆形网格的生成器。

- **circle**(arguments...)

返回一个类型为 Polygon 的 GeoJSON 对象。

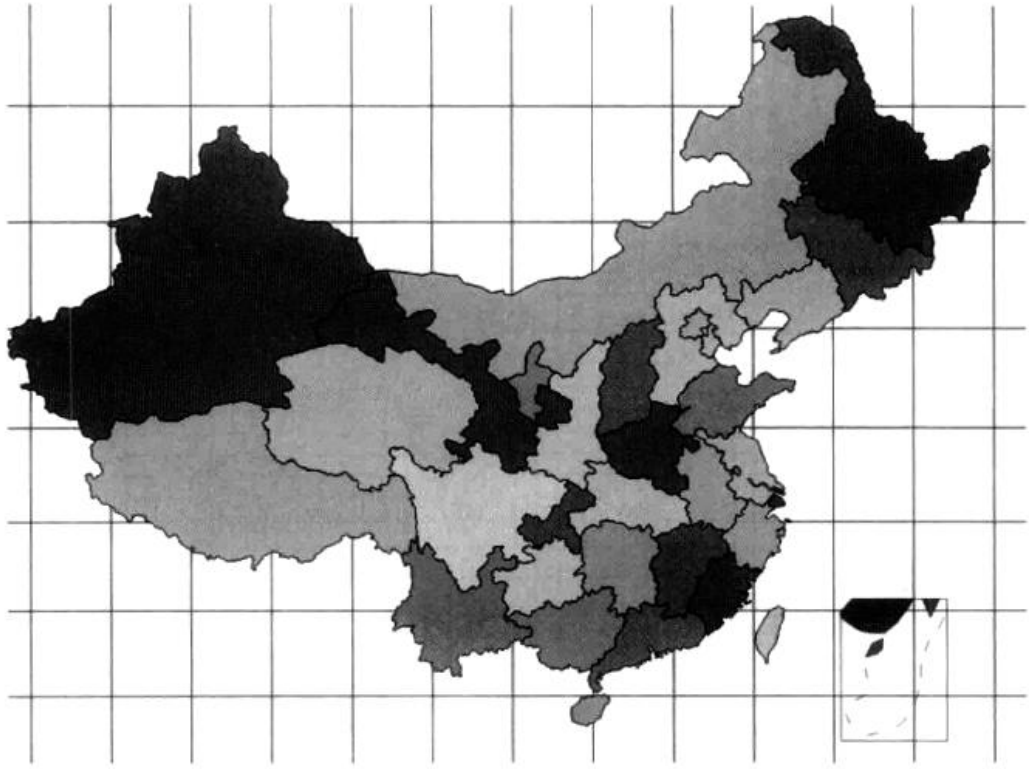


图 11-28　带经纬度网格的中国地图

- circle.**origin**([origin])

设定原点，origin 是一个数组，表示经纬度，默认为[0, 0]。

- circle.**angle**([angle])

设定角半径，默认为 90° 。

下面制作一个示例，要求是：**分别在平面地图和球体地图上展示圆形网格。**

首先，创建一个圆形网格生成器，原点设置在印度洋中部，经度为 77° ，纬度为–19° 。以原点为中心，每隔 5° 绘制一个圆，网格包含整个地球。

```
var angles = d3.range(0,180,5);

var geocircle = d3.geo.circle()
                .origin([77,-19]);
```

这里使用d3.range()创建了一个等差数列，从0～180，每隔5取一个值，结果为[0, 5, 10, 15, ... , 175]，长度为36。

然后，在绘制网格时依次设定不同的角度，并生成圆形网格的路径。

```
var geocircle = d3.geo.circle()
                .origin([77,-19]);
```

```
svg.append("g")
    .selectAll(".geocircle")
    .data(angles)
    .enter()
    .append("path")
    .attr("class", "geocircle")
    .attr("d", function(d){
        var circle = geocircle.angle(d);          //设定角度
        return path( circle() );                  //生成网格的GeoJSON并获取路径
    });
```

这段代码添加了 36 个<path>，分别绑定数组 angles 的各项元素。计算路径时，先用 geocircle.angle()设定角半径，然后用 circle()生成圆形网格的 GeoJSON 对象，再通过地理路径生成器获取路径。

结果如图 11-29 所示，从印度洋中心开始，圆形网格扩散到全球。

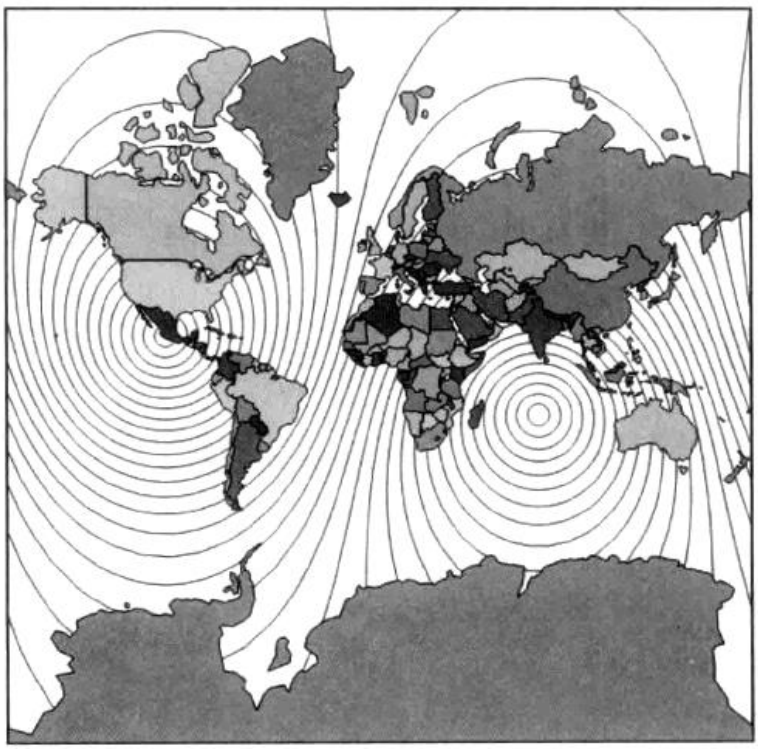


图 11-29　平面地图上显示圆形网格

为了能更好地理解圆形网格是怎样的，更改一下投影函数，使地图显示为地球仪模式。定义投影函数如下：

```
var projection = d3.geo.orthographic()
                   .center([0, 0])
                   .scale(300)
```

```
                .rotate([-50,0])
                .clipAngle(90)
                .translate([width/2, height/2]);
```

结果如图 11-30 所示。

图 11-30　球形地图上显示圆形网格

11.4　投影

使用 d3.geo.projection()能够创建投影。但是，除非需要一个很特殊的投影，否则用不到 d3.geo.projection()。本章前几节已经多次使用了投影，例如 d3.geo.mercator()和 d3.geo.orthographic()，这些投影都是通过 d3.geo.projection 创建的，并且已经被设置好了，适用于绝大部分地图。下面介绍与投影相关的方法。

- d3.geo.**projection**(raw)

以点函数 raw 创建一个投影。

- **projection**(location)

将球面坐标系的坐标（单位为度）投影到笛卡儿坐标系的坐标（单位为像素）。参数 location 是一个数组，形式为[经度, 纬度]。返回值是一个数组，表示像素的位置[x, y]。

- projection.**invert**(point)

将笛卡儿坐标系的坐标（单位为像素）反投影到球面坐标系的坐标（单位为度）。参数 point 是一个数组[x, y]，返回值也是一个数组，表示[经度, 纬度]。

- projection.**rotate**([rotation])

设置旋转，参数 rotation 是一个数组[yaw, pitch, roll]，分别表示偏航角度（绕 y 轴旋转）、俯仰角度（绕 x 轴旋转）、翻滚角度（绕 z 轴旋转）。默认为[0, 0, 0]。

- projection.**center**([location])

设定投影的中心，location 是一个数组，形式为[经度, 纬度]，默认为[0, 0]。

- projection.**translate**([point])

设置投影的平移属性，point 是一个数组，形式为[x, y]，默认为[480, 250]。

- projection.**scale**([scale])

设置投影的缩放因子，默认为 150。

- projection.**clipAngle**(angle)

设定投影的裁剪角度，单位是度。

- projection.**clipExtent**(extent)

设定一个矩形的裁剪框，extent 的格式为[[x0, y0],[x1, y1]]，x0 是视窗的左边界，y0 是上方，x1 是右方，y1 是下方。

上面说到，使用 d3.geo.projection()手动创建一个投影的情况很少。那是因为 D3 已经预定义了很多种投影。

1．墨卡托投影：d3.geo.mercator()

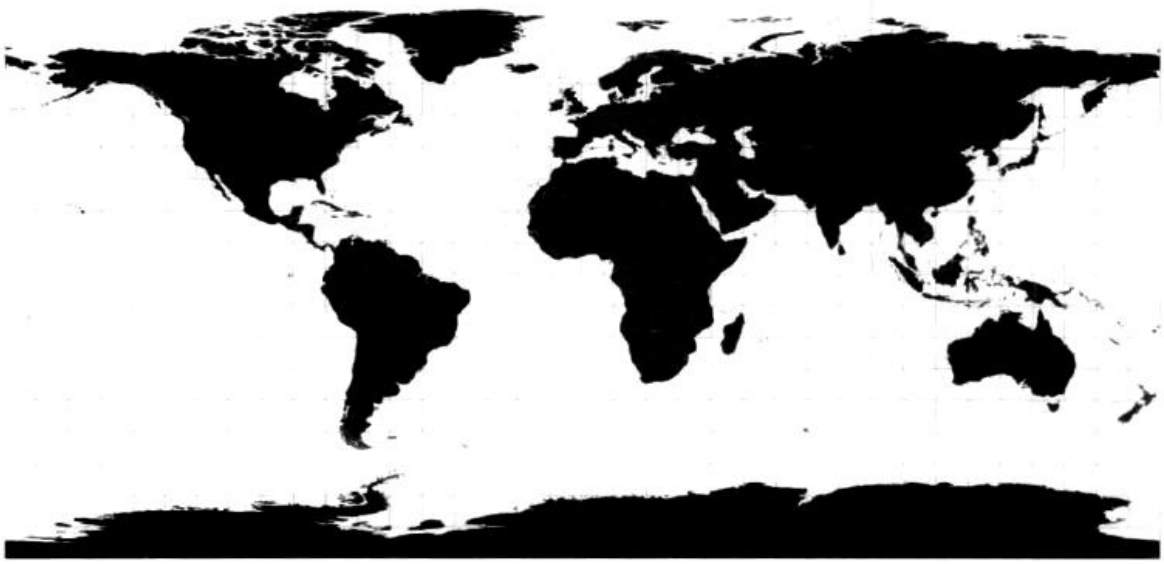
地理学家墨卡托于 1569 年发表的一幅长 202cm、宽 124cm 的世界地图里所用的投影法。在此投影法的地图上，经纬线于任何位置垂直相交，使得地图可绘制在长方体上。墨卡托投影如图 11-31 所示。

http://bl.ocks.org/mbostock/3757132

图 11-31　墨卡托投影

2．等距圆筒投影：d3.geo.equirectangular()

该投影的经线和纬线是相互垂直且等距的，如果赤道的长度为 2，那么南极到北极的距离为 1，展开之后可得一个宽为高度 2 倍的长方形。等距圆筒投影如图 11-32 所示。



http://bl.ocks.org/mbostock/3757119

图 11-32　等距圆筒投影

3．正射投影：d3.geo.orthographic()

正射投影是一种透视投影，适合显示半球，可用于制作地球仪。正射投影如图 11-33 所示。

http://bl.ocks.org/mbostock/3757125

图 11-33　正射投影

4．球极平面投影：d3.geo.stereographic()

一种以平面来看球面的方法，也是一种透视投影。球极平面投影如图 11-34 所示。

http://bl.ocks.org/mbostock/3757137

图 11-34　球极平面投影

5．心射极平投影：d3.geo.gnomonic()

将球面上的大圆投影成直线的投影。心射极平投影如图 11-35 所示。

http://bl.ocks.org/mbostock/3757349

图 11-35　心射极平投影

6．扇形投影：d3.geo.conicEquidistant()

将地图以扇形的方式显示。扇形投影如图 11-36 所示。

http://bl.ocks.org/mbostock/3734317

图 11-36　扇形投影

下面以等距圆筒投影 d3.geo.equirectangular()来讲解设定中心 center()、缩放 scale()、平移 translate()、投影 projection()和反投影 invert()，以及矩形裁剪 clipExtent()的用法。

如图 11-37 所示，平移由 translate([x, y])来提供 ***x* 方向的平移量**和 ***y* 方向的平移量**，平移之后，原点坐标为(x, y)。但是，将地图的哪一点作为原点呢，是北京，还是上海？这是由 center([cx, cy])设定的。地图的缩放比例由 scale()设定，默认的缩放因子为 150。缩放因子不要设置为 1 这样的值，会小得看不见地图。

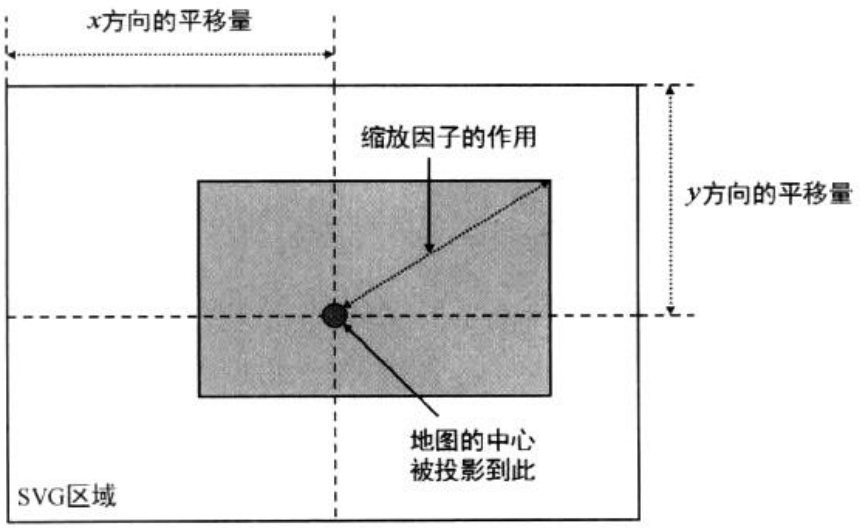


图 11-37　设定中心、平移、缩放的参数意义

例如，定义如下投影，中心设定为北京的经纬度，缩放因子为 80，通过平移使坐标原点位于 SVG 区域的中心：

```
var projection = d3.geo.equirectangular()
        .center([116.38, 39.93])                    //中心设置为北京
        .scale(80)                                  //缩放因子
        .translate([width/2, height/2]);            //以SVG区域的中心为坐标原点
```

结果如图 11-38 所示。

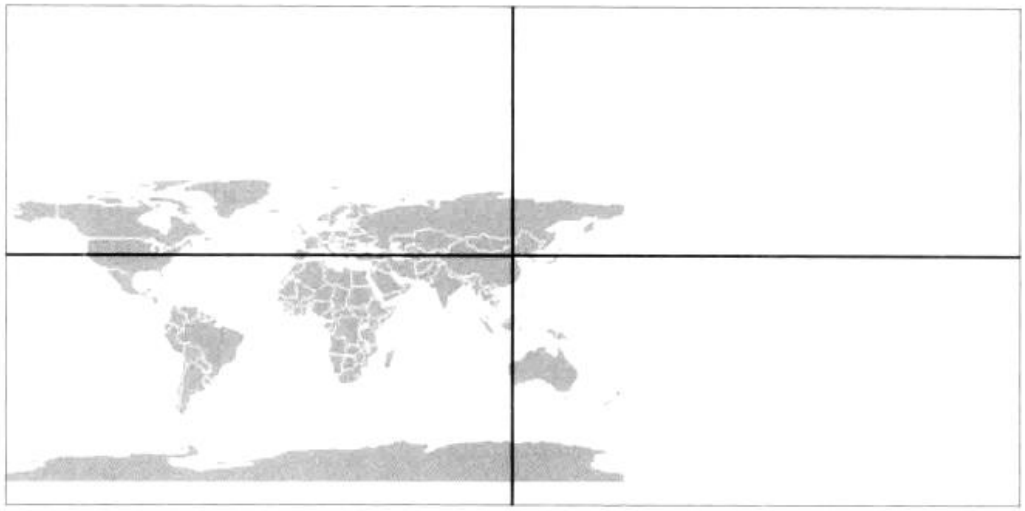


图 11-38　北京位于坐标原点

如果修改投影的参数如下：

```
var projection = d3.geo.equirectangular()
                   .center([0, 51.5])       //中心设置为伦敦
                   .scale(100)              //缩放因子
                   .translate([600, 200]); //以(600,200)处为坐标原点
```

结果如图 11-39 所示。

图 11-39　伦敦位于坐标原点，原点位于 SVG 区域的(600,200)坐标处

下面利用投影函数，在图 11-38 上加一个红点，标出美国首都华盛顿的位置。在 projection() 里传入华盛顿的经纬度作为参数，即可返回投影之后的坐标，然后在此位标处添加一个圆：

```
var washington = projection([-77.04,38.91]);  //获取华盛顿的投影坐标

svg.append("circle")
   .attr("cx",washington[0])
   .attr("cy",washington[1])
   .attr("r",10)
   .style("fill","red");
```

结果如图 11-40 所示，华盛顿位置被标出了一个红点。如果使用 invert()可通过平面上的坐标得到经纬度，代码如下：

```
var pos = projection.invert(washington);

console.log(pos);  //输出[-77.04, 38.91]
```

输出的数组中，第一项是经度，第二项是纬度。

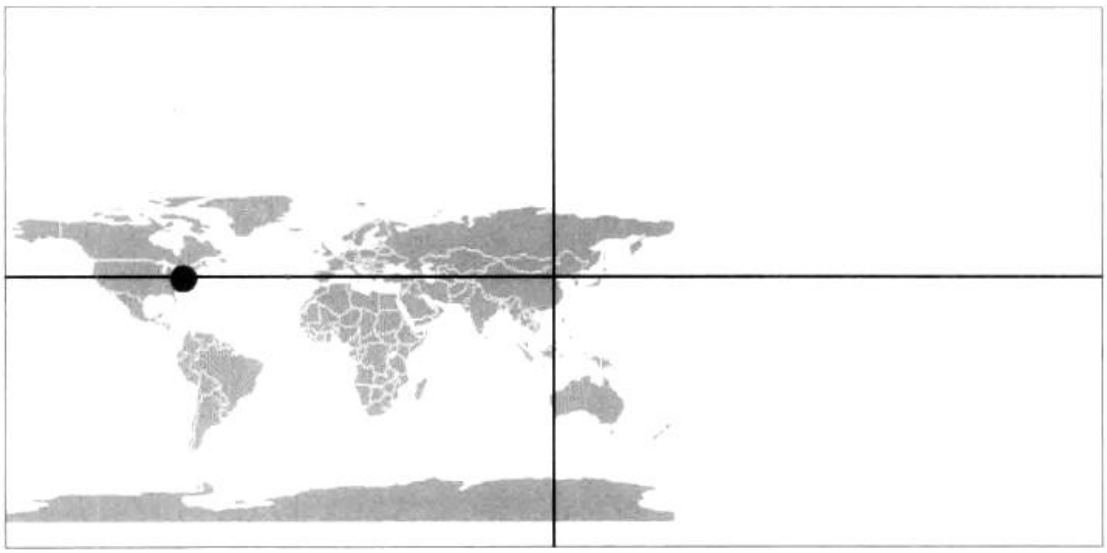


图 11-40　标记出华盛顿的位置

如果要将图 11-40 的右半部分裁剪掉，将 clipExtent()的参数设置为：

```
.clipExtent([[0,0],[width/2,height]])
```

即可，结果如图 11-41 所示。

接下来，以正射投影 d3.geo.orthographic()为例来看旋转和裁剪的用法，定义以下投影：

```
var ortho = d3.geo.orthographic()
              .scale(130)
```

```
        .translate([width/2, height/2])
        .rotate([0,0,0])
        .clipAngle(90);
```

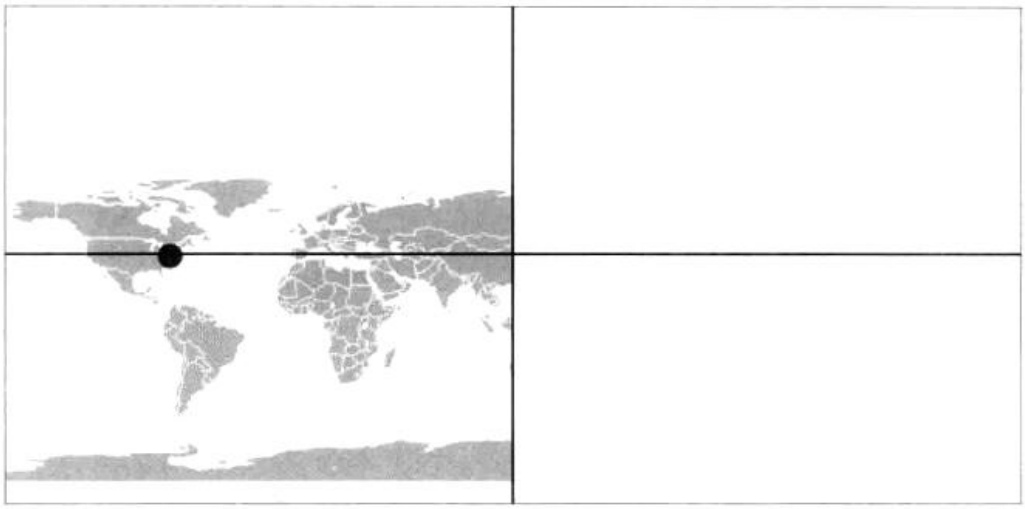


图 11-41　裁剪掉右半部分

旋转函数 rotate()的三个角度都是 0，裁剪角度 clipAngle()为 90°，结果如图 11-42 的左图所示，如果将 rotate()的参数改为[60, 0, 0]，则结果如图 11-42 的右图所示。要注意数值的正负与旋转的关系问题。

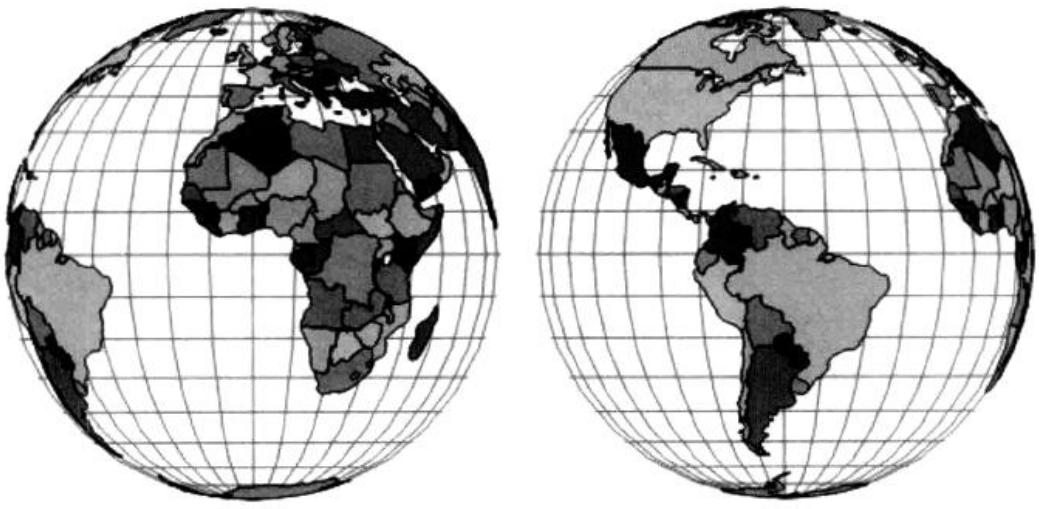


图 11-42　rotate()的参数为[0, 0, 0]的状态（左），参数为[60, 0, 0]的状态（右）

上面的投影还有一个裁剪函数 clipAngle()，角度被设置成了 90°，这是什么意思呢？先看将角度分别设置为 30°、60°、90°、180° 的结果。如图 11-43 所示，随着角度的增加，显示的部分逐渐增多。裁剪角度的意义可能较难理解，其含义如图 11-44 所示，当 angle 为 30° 时，实

际上“可见区域”有 60° 的范围，这就是图 11-43 的左上图所显示的内容。因此，要显示半球的情况下，应将裁剪角度设置为 90° ，即图 11-43 左下的情况。如果角度超过 90° ，例如 180° ，则会变成图 11-43 右下的情况，地球背面的一部分地形与前面的重合，这不是理想的半球地图。

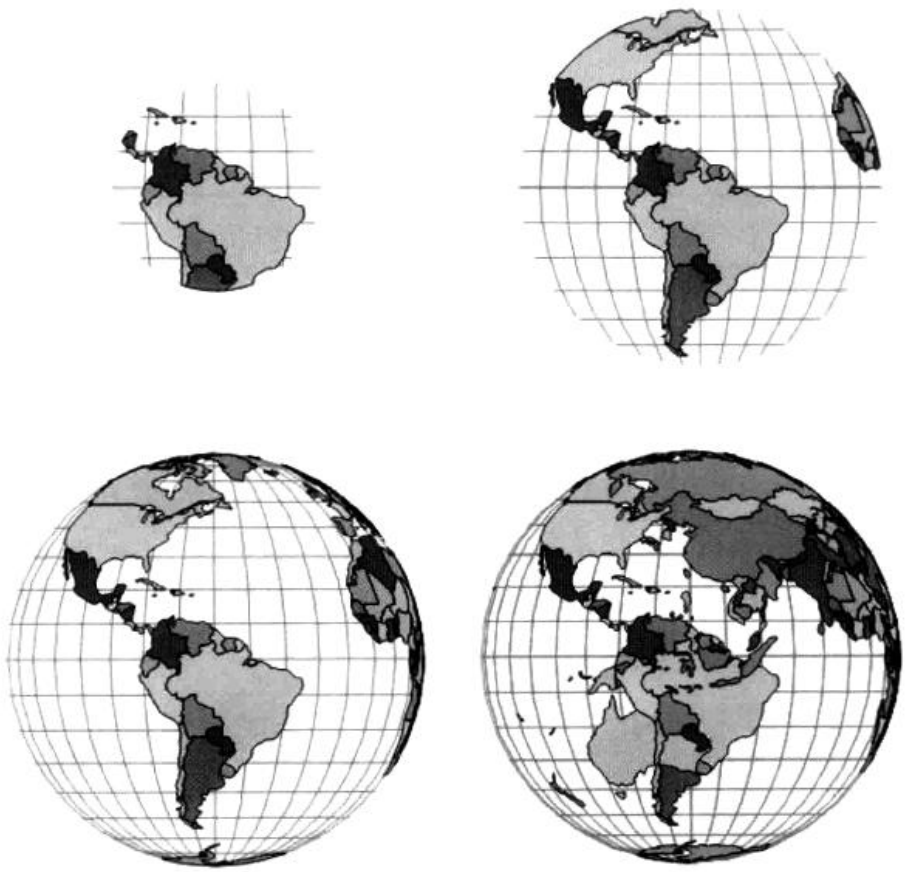


图 11-43　clipAngle()的参数分别为 30° （左上）、60° （右上）、90° （左下）、180° （右下）

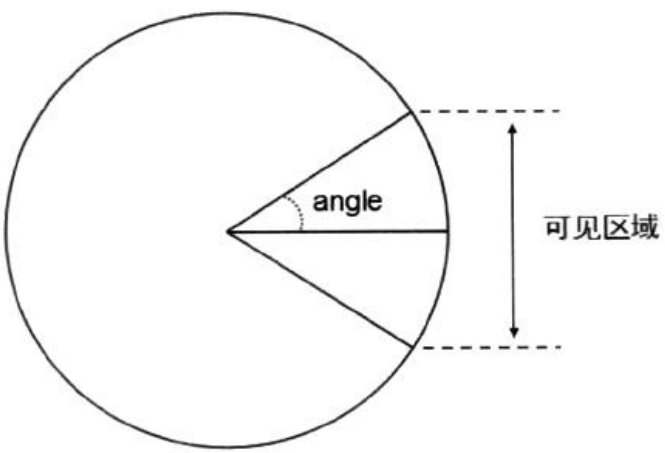


图 11-44　裁剪角度的意义

11.5　球面数学

第 11.3.1 节提到了一些计算地图的面积、中心、边框的方法，单位是像素或像素平方，这些值都是经过投影后的值。但是，地图数据原本是保存在 GeoJSON（或 TopoJSON）中的，而且是以“经度，纬度”的形式保存的。有时候需要根据经纬度进行计算，例如计算几何体的中心。

为此，D3 提供了一些方法，能够以 GeoJSON 格式的数据进行球面数学的计算，相关介绍如下。

- d3.geo.**area**(feature)

返回几何体的立体角（Solid Angle），单位为球面度（Steradian）。立体角类似于二维上圆的弧长。

- d3.geo.**centroid**(feature)

返回几何体的中心，形式为[经度, 纬度]。

- d3.geo.**bounds**(feature)

返回几何体的边界框，形式为[[left, bottom], [right, top]]，其中 left 是最小经度，bottom 是最小纬度，right 是最大经度，top 是最大纬度。

例如，要计算甘肃的立体角、中心和边界框，代码如下：

```
var gansu = root.features[0]; //甘肃的GeoJSON对象

var area = d3.geo.area(gansu);
var centroid = d3.geo.centroid(gansu);
var bounds = d3.geo.bounds(gansu);
```

要注意，area 计算的是立体角 Ω，并非面积。由立体角计算表面积的公式为：$S = R^2 \Omega$，其中 R 是球体的半径。

- d3.geo.**distance**(a, b)

返回 a 和 b 两点的弧长，单位是弧度。a 和 b 都是数组，形式为[经度, 纬度]。

- d3.geo.**length**(feature)

返回几何体边界的长度，单位是弧度。

- d3.geo.**interpolate**(a, b)

返回一个插值函数，对 a 和 b 两点间的部分进行插值。a 和 b 都是数组，形式为[经度, 纬度]。返回的插值函数，在使用时需要传入一个参数 t，范围为 0～1。当 t 等于 0 时，返回 a 点；当 t 等于 1 时，返回 b 点。

请看下面例子。

```
var beijing = [116.4, 39.9];          //北京的经纬度
var shanghai = [121.5, 31.2];         //上海的经纬度

var dis = d3.geo.distance(beijing, shanghai);
var length = d3.geo.length(province);
var interpolator = d3.geo.interpolate(beijing, shanghai);

console.log( interpolator(0) );       //输出北京的经纬度
console.log( interpolator(0.5) );     //输出北京—上海的中点的经纬度
console.log( interpolator(1) );       //输出上海的经纬度
```

由于 d3.geo.distance()和 d3.geo.length()的返回值都是弧度，如果要计算弧长，要用弧度乘以半径来计算。

- d3.geo.**rotation**(rotate)

返回一个旋转器函数。rotate 是一个数组：[经度, 纬度, 原点]。原点可以省略。

- **rotation**(location)

返回 location 旋转后的经纬度。

- rotation.**invert**(location)

反旋转指定位置。

请看以下代码，定义一个经度旋转 30° 的旋转器，并将北京的经纬度进行旋转。

```
var beijing = [116.4, 39.9];                          //北京的经纬度

var rotation = d3.geo.rotation([30,0]);               //创建一个旋转器
var posRotated = rotation(beijing);                   //使用旋转器

console.log( posRotated );                            //输出旋转后的经纬度
console.log( rotation.invert(posRotated) );           //输出北京的经纬度
```

这段代码输出结果为：

```
[146.4, 39.9]
[116.4, 39.9]
```

北京的经度经过旋转器变换后，变为[116.4 + 30, 39.9]。反旋转后又得到了北京的经纬度。

第12章

友好的交互

本章内容包括：

- 提示框
- 坐标系中的焦点
- 元素组合
- 区域选择
- 开关

交互式操作，是指用户与图表之间的信息交换，例如用户点击图表，图表展现出某种变化。交互式操作有很多，本章介绍五种典型的手法，每一种手法都制作一两个图表。

第 1 节，讲述如何制作提示框。提示框，是当用户鼠标移到图表的某一元素上时弹出的，包含提示文字的框。

第 2 节，介绍坐标系中的焦点，当图表中有坐标轴存在时经常会用到。

第 3 节，学习如何组合图表的元素，以饼状图为例做说明。

第 4 节，讲述如何在 SVG 画板中进行区域选择，涉及一个新的概念：brush（刷子）。

第 5 节，学习如何制作开关，并据此制作一个思维导图。

12.1 提示框

一般来说，图表中不宜存在过多文字。但是，有时需要一些文字来描述某些图形元素。那

么，可以实现一种交互：**当用户鼠标滑到某图形元素时，出现一个提示框，里面写有描述文字。**这是一种简单、普遍的交互式，几乎适用于所有图表。通过给提示框定制外观，能给用户带来很好的体验。

提示框，就是“文字”加“边框”。前面章节中，如果要在 SVG 区域中显示文字，都使用<text>元素。但是，有两个问题：

（1）如果字符串过长，<text>元素不能自动换行，虽然可以通过<text>的子元素<tspan>来模拟自动换行的功能，但是很麻烦。

（2）<text>是 SVG 的元素。也就是说，<text>是“图形”而非“文字”，它与 SVG 中的<circle>、<line>、<path>等元素本质上是一样的。那么，当输出 SVG 图形时，<text>也会作为图形的一部分输出。

因此，SVG 的<text>元素不适合制作提示框。

有一种简单的方法：div + css。div 是 HTML 的元素，在样式中设定其定位方法为**绝对定位**：

```
.tooltip{
    position: absolute;
    width: 120;
    height: auto;
}
```

然后，当监听到鼠标事件时，用鼠标的坐标为提示框定位即可，代码形如：

```
element.on("mouseover",function(d){
            tooltip.style("left", (d3.event.pageX) + "px")
                   .style("top", (d3.event.pageY) + "px");
        })
```

实际应用中，为使提示框美观，还需为 div 设置更多的样式。

12.1.1 饼状图的提示框

以饼状图为例，制作一个简单的提示框。将第 10.2 节的饼状图稍微修改后，得到图 12-1。此饼状图的数据是**各大厂商的名称和出货量**：

```
var dataset = [ ["小米",60.8], ["三星",58.4], ["联想",47.3],
                ["苹果",46.6], ["华为",41.3], ["酷派",40.1] ,
                ["其他",111.5] ];
```

为了显示各厂商所占市场份额的百分比，在图形中显示的数据是 9.9%、10.2%这样的数值。但是，从图中不能了解到各大厂商的出货量。如果将出货量的文字也添加到图中，会显得很乱。这种时候就可以考虑添加提示框。

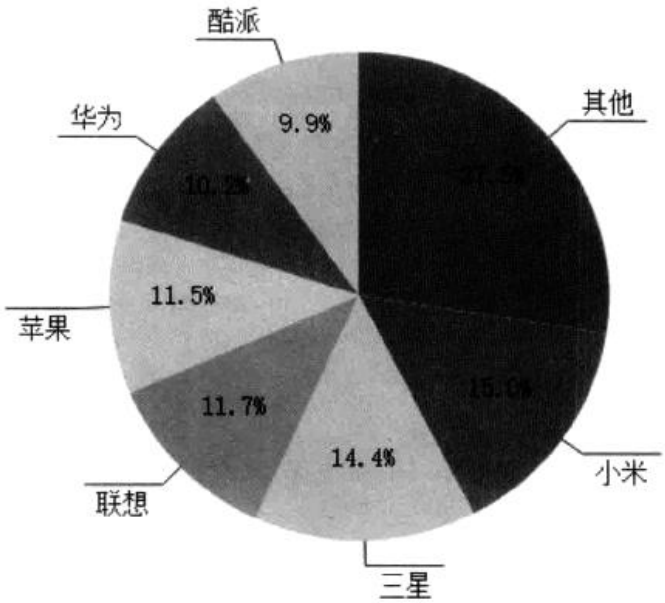


图 12-1　饼状图

首先，在<body>中添加一个 ，透明度设定为 0，即完全透明，div 的类设定为 tooltip。

```
var tooltip = d3.select("body")
                .append("div")
                .attr("class","tooltip")
                .style("opacity",0.0);
```

然后，定义一个 tooltip 样式，并将其定位方式设置为**绝对定位**，这一步是重点。其他的属性是关于边框外观和文字显示方式的，此处定义一个简单的。

```
.tooltip{
    position: absolute;
    width: 120;
    height: auto;
    font-family: simsun;
    font-size: 14px;
    text-align: center;
    border-style: solid;
    border-width: 1px;
    background-color: white;
    border-radius: 5px;
}
```

最后，为饼状图的各图形元素定制鼠标事件的监听器，其中包括：鼠标放到图形上时（mouseover）、鼠标在图形上移动时（mousemove），鼠标移出时（mouseout）。

```
arcs.on("mouseover",function(d){
        /*
```

```
        鼠标移入时，
        （1）通过 selection.html() 来更改提示框的文字
        （2）通过更改样式 left 和 top 来设定提示框的位置
        （3）设定提示框的透明度为1.0（完全不透明）
        */

        tooltip.html(d.data[0] + "的出货量为" + "<br />" +
                     d.data[1] + " 百万台")
               .style("left", (d3.event.pageX) + "px")
               .style("top", (d3.event.pageY + 20) + "px")
               .style("opacity",1.0);
    })
    .on("mousemove",function(d){
        /* 鼠标移动时，更改样式 left 和 top 来改变提示框的位置 */

    tooltip.style("left", (d3.event.pageX) + "px")
           .style("top", (d3.event.pageY + 20) + "px");
    })
    .on("mouseout",function(d){
        /* 鼠标移出时，将透明度设定为0.0（完全透明）*/

        tooltip.style("opacity",0.0);
    })
```

d3.event.pageX 和 d3.event.pageY 是当前鼠标相对于浏览器页面的坐标，而对于处于绝对定位状态的 元素来说，其样式 left 和 top 也是相对于浏览器页面来说的。赋值的时候，令 top 的值为 d3.event.pageY + 20，使提示框稍微显示在鼠标位置的下方，这么做能够防止鼠标在提示框上移动导致不触发事件的问题。

结果如图 12-2 所示，当鼠标移动到“联想”上时，出现了提示框。

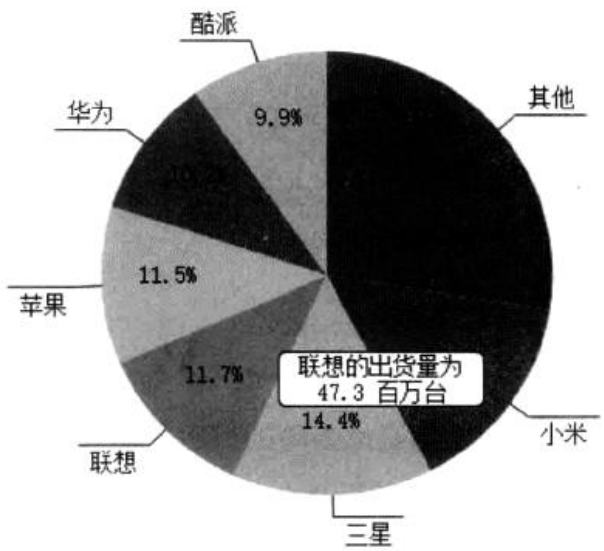


图 12-2　鼠标滑动到“联想”上时出现的提示框

12.1.2　提示框的样式

上一节的提示框很简陋。稍微修改一下 CSS 样式，为其定制一个美观的边框：

```
.tooltip{
    position: absolute;
    width: 120px;
    height: auto;
    text-align: center;
    font-family: simsun;
    font-size: 14px;
    color: white;
    background-color: black;
    border-width: 2px solid black;
    border-radius: 5px;
}
.tooltip:after {
    content: '';
    position: absolute;
    bottom: 100%;
    left: 20%;
    margin-left: -8px;
    width: 0;
    height: 0;
    border-bottom: 12px solid #000000;
    border-right: 12px solid transparent;
    border-left: 12px solid transparent;
}
```

提示框的背景和边框都设定为黑色，字体设定为白色，并且通过 tooltip:after 样式为提示框上方添加了一个小箭头，结果如图 12-3 所示。

还可以为提示框添加一个颜色标识，以标识其指向的图形元素的颜色，例如在提示框的右边添加一个小阴影框，如图 12-4 所示。

这需要使用到 CSS3 的 box-shadow 属性，其语法如下：

```
box-shadow: h-shadow v-shadow blur spread color inset;
```

其中，h-shadow 是水平阴影、v-shadow 是垂直阴影的位置，其他是可选参数，形如：

```
box-shadow: 10px 0px 0px red;
```

了解了 box-shadow 的用法，即可在鼠标事件被触发时，将被选择图形的颜色值作为 box-shadow 的参数使用。

```
arcs.on("mouseover",function(d,i){
        tooltip.html(d.data[0] + "的出货量为" + "<br />" +
```

```
                    d.data[1] + " 百万台")
                .style("left", (d3.event.pageX) + "px")
                .style("top", (d3.event.pageY + 20) + "px")
                .style("opacity",1.0);

        // color(i) 为被选择图形的颜色
        tooltip.style("box-shadow", "10px 0px 0px " + color(i) );
})
```

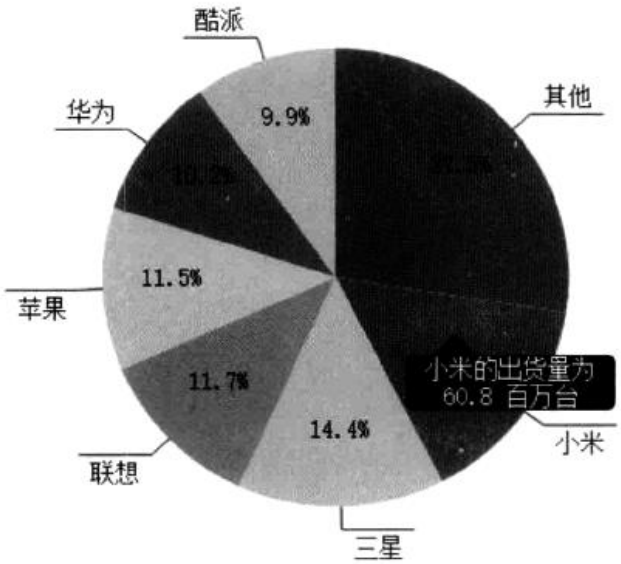


图 12-3　带有小箭头的提示框

图 12-4　添加阴影框

当鼠标移动到某图形元素上时，将其颜色作为阴影框的颜色。如此有利于告诉用户是哪一个图形元素被选中。鼠标滑动到“华为”上的效果如图 12-5 所示。

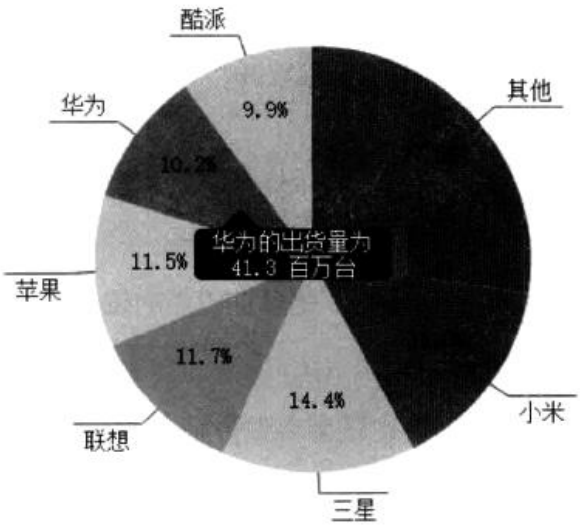


图 12-5　鼠标滑动到“华为”上

此外，还可以使用外部图片为提示框定制外观。但是，由于可视化图表应当追求简洁清晰，不宜使用特别华丽的边框。

12.2　坐标系中的焦点

坐标轴是普遍使用的度量工具，对于柱形图、折线图、散点图之类的图表来说更是基本组成单元。以折线图为例，有时会有这样的交互需求：

随着鼠标在坐标系中滑动，显示出与鼠标的 *x* 值对应的折线图中的焦点，并连接两条到坐标轴的虚线，以表明该点到坐标轴的距离。

图 12-6 是该交互的示意图。不仅折线图，有坐标轴的图表都可考虑类似的交互式需求。本节将在折线图上显示的点称为**焦点**，与坐标轴垂直的虚线称为**对齐线**。

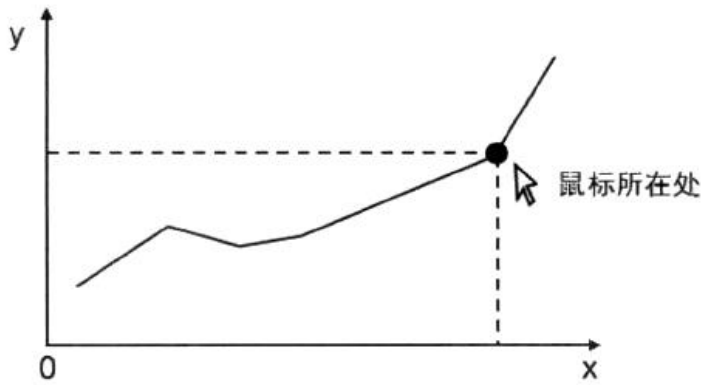


图 12-6　折线图中的焦点，以及对应到坐标轴的虚线

12.2.1　折线图的焦点

第 6 章制作了一个反映从 2000 年到 2013 年中日两国 GDP 变化的折线图，下面在此图的基础上，使用户能在坐标系中滑动鼠标的同时显示出对应的焦点和对齐线。为了使问题简单，只保留中国一条折线，如图 12-7 所示。

首先，请思考两个问题：

（1）怎样捕捉鼠标事件。

（2）怎样确定焦点的位置。

先看第一个问题。对某个选择集监听鼠标事件的代码如下：

```
selection.on("mouseover",function(d){
    //监听器代码
})
```

selection 是被监听的对象，当鼠标移动到该选择集的图形元素上时即触发监听器函数的内容。但是，这个**选择集**是指什么呢？

折线吗？不可能。因为折线是用线段生成器生成的，只有一个<path>元素，如果对它设置鼠标监听，那么只有当鼠标滑到折线上时才会触发时间，在其他位置滑动则不会。而且折线很细，鼠标滑动到折线上很困难，会影响交互性。

坐标轴吗？也不可能。如果用户需要在坐标轴上滑动，更会影响交互性。

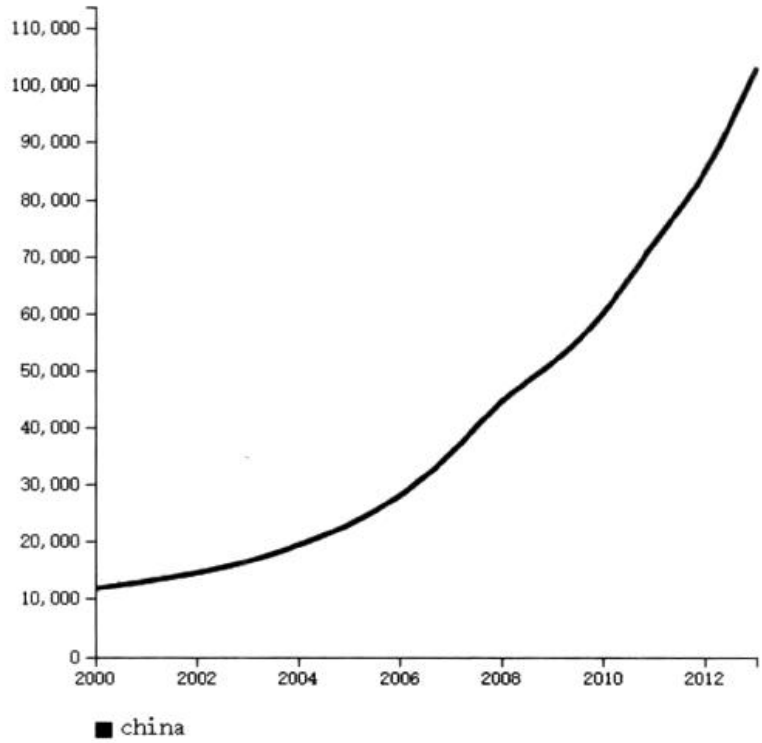


图 12-7　中国 2000 年到 2013 年 GDP 变化折线图

除了折线和坐标轴，在坐标系中已经没有图形元素了。那么，没有元素不能设置监听器。因此，**需要在坐标系中添加一个透明矩形，该矩形不代表任何图形元素，只是为了捕捉鼠标事件**，代码如下：

```
svg.append("rect")
    .attr("class","overlay")
    .attr("x", padding.left)
    .attr("y", padding.top)
    .attr("width",width - padding.left - padding.right)
    .attr("height",height - padding.top - padding.bottom)
    .on("mouseover", function(){
        //监听鼠标移入事件
    })
    .on("mouseout", function(){
```

```
        //监听鼠标移出事件
    })
    .on("mousemove", function(){
        //监听鼠标滑动事件
    });
```

该矩形的长为 x 轴的长度，宽为 y 轴的长度，正好覆盖坐标系中的所有区域，当然也包括折线图。如此，无论鼠标滑动到坐标轴的什么位置，程序都能捕捉得到。然后，再给该矩形元素设置样式：

```
.overlay {
    fill: none;
    pointer-events: all;
}
```

填充方式设置为 none，即不填充。pointer-events 设置为 all，这个属性很重要，它表示即便有其他元素覆盖于矩形之上，矩形也能够捕捉到事件。图 12-8 展示了透明矩形的位置。

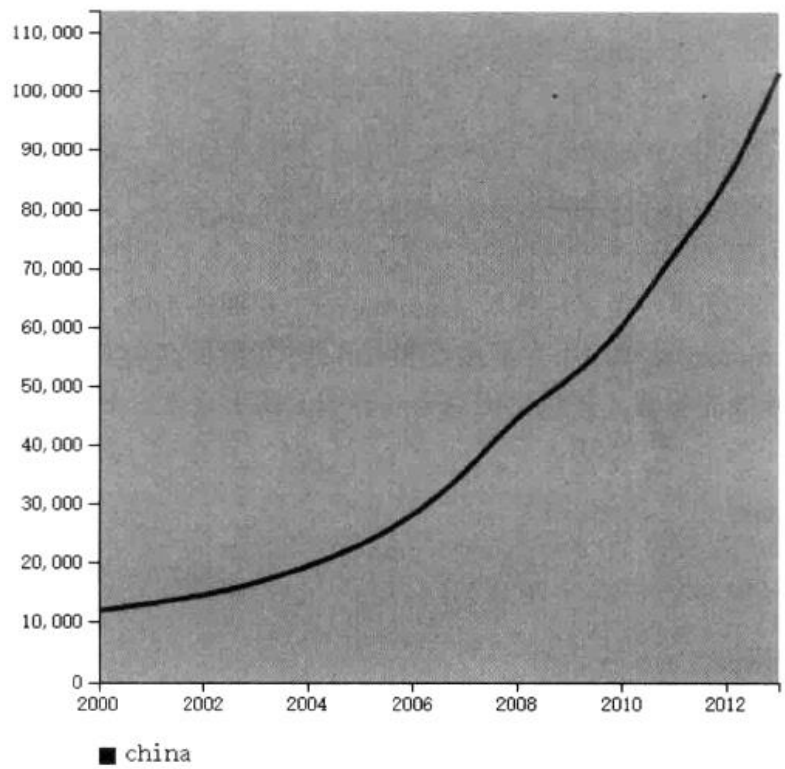


图 12-8　覆盖在折线图上的透明矩形（为了表明矩形的位置，此图中染了色）

然后看本节开始提出的第二个问题。确定焦点的位置，也就是计算焦点的 x 坐标和 y 坐标，焦点必定是位于折线上的。在生成折线时，计算过折线图上各点的坐标，当时是通过 x 轴和 y 轴的比例尺，将数据转换成坐标系中的点。那么，现在能够凭借透明矩形捕获坐标系中的点，

当然也能够使用**比例尺的反函数**求取折线上的点。

想明白上面两个问题，就很容易写成代码了。

首先，创建焦点和对齐线所需的图形元素，暂时不设定它们的位置属性，并且将 display 属性设置为 none，令其不显示出来：

```
//焦点的元素
var focusCircle = svg.append("g")
      .attr("class", "focusCircle")
      .style("display", "none");

focusCircle.append("circle")
      .attr("r", 4.5);

focusCircle.append("text")
      .attr("dx", 10)
      .attr("dy", "1em");

//对齐线的元素
var focusLine = svg.append("g")
                   .attr("class", "focusLine")
                   .style("display", "none");

var vLine = focusLine.append("line");
var hLine = focusLine.append("line");
```

然后，设定交互式操作：当鼠标移入（mouseover）透明矩形时，显示焦点和对齐线；当鼠标从透明矩形移出（mouseout）时，不显示焦点和对齐线；当鼠标在透明矩形上滑动（mousemove）时，计算焦点和对齐线的位置，并赋予焦点和对齐线的图形元素。对透明矩形添加监听器，代码如下：

```
.on("mouseover", function() {
    focusCircle.style("display", null);
    focusLine.style("display", null);
})
 .on("mouseout", function() {
     focusCircle.style("display", "none");
     focusLine.style("display", "none");
 })
 .on("mousemove", mousemove);
```

最重要的是 mousemove 事件的监听器，每次触发该事件时，都需要重新计算焦点的位置。在本例中，x 轴表示年份（2000 年、2001 年、2002 年等），y 轴表示 GDP 值（11920 亿美元、13170 亿美元、14550 亿美元等），每一对值（2000 年—11920 亿美元）都是折线图上的一个点，

折线图就是连接这些点而形成的。现在，需要根据当前鼠标的位置，来判断应该显示折线图上的哪一个点。其计算流程如下：

（1）捕获鼠标位置，得到一个位置(166, 238)。

（2）根据比例尺的反函数计算年份和 GDP 值，计算得到(2005.395, 46131.525)，即 2005.395 年的 GDP 值为 46131.525 亿美元。但是，这个点是根据比例计算得到的，折线图中没有这个点。

（1）将 2005.395 进行四舍五入，得到 2005 年。

（2）在原数组中查找 2005 年的中国的 GDP 值，得到 22870 亿美元。

（3）使用比例尺，计算(2005, 22870)在折线图上的点坐标，该点即为焦点。

（4）根据焦点位置绘制对齐线。

最后，根据上面的流程，写成 mousemove 事件的监听器，代码如下：

```
function mousemove() {
    /* 当鼠标在透明矩形内滑动时调用 */

    //折线的源数组
    var data = dataset[0].gdp;

    //获取鼠标相对于透明矩形左上角的坐标，左上角坐标为(0,0)
    var mouseX = d3.mouse(this)[0] - padding.left;
    var mouseY = d3.mouse(this)[1] - padding.top;

    //通过比例尺的反函数计算原数据中的值，例如x0为某个年份，y0为GDP值
    var x0 = xScale.invert( mouseX );
    var y0 = yScale.invert( mouseY );

    //对x0四舍五入，如果x0是2005.6，则返回2006；如果是2005.2，则返回2005
    x0 = Math.round(x0);

    //查找在原数组中x0的值，并返回索引号
    var bisect = d3.bisector( function(d) { return d[0]; }).left;
    var index = bisect(data, x0) ;

    //从数据中获取年份和GDP值
    var x1 = data[index][0];
    var y1 = data[index][1];

    //分别用x轴和y轴的比例尺，计算焦点的位置
    var focusX = xScale(x1) + padding.left;
    var focusY = yScale(y1) + padding.top;

    //通过平移，使焦点移动到指定位置
    focusCircle.attr("transform","translate(" + focusX + "," +
```

```
                    focusY + ")");

    //设定焦点的文字信息
    focusCircle.select("text").text( x1 + "年的GDP： " +
                                     y1 + "亿美元");

    //设定垂直对齐线的起点和终点
    vLine.attr("x1", focusX)
        .attr("y1",focusY)
        .attr("x2",focusX)
        .attr("y2",height - padding.bottom);

    //设定水平对齐线的起点和终点
    hLine.attr("x1", focusX)
        .attr("y1",focusY)
        .attr("x2",padding.left)
        .attr("y2",focusY);
}
```

如图 12-9 所示，当鼠标的 x 坐标在 2006.5 到 2007.5 之间时，显示出了图中的焦点和对齐线，焦点右边有文字描述："2007 年的 GDP：35940 亿美元"。

滑动鼠标，焦点和对齐线会随着鼠标的位置变化自动计算新的位置。要注意，每次只需更新焦点和对齐线的坐标即可，不需要重新插入新的图形元素。

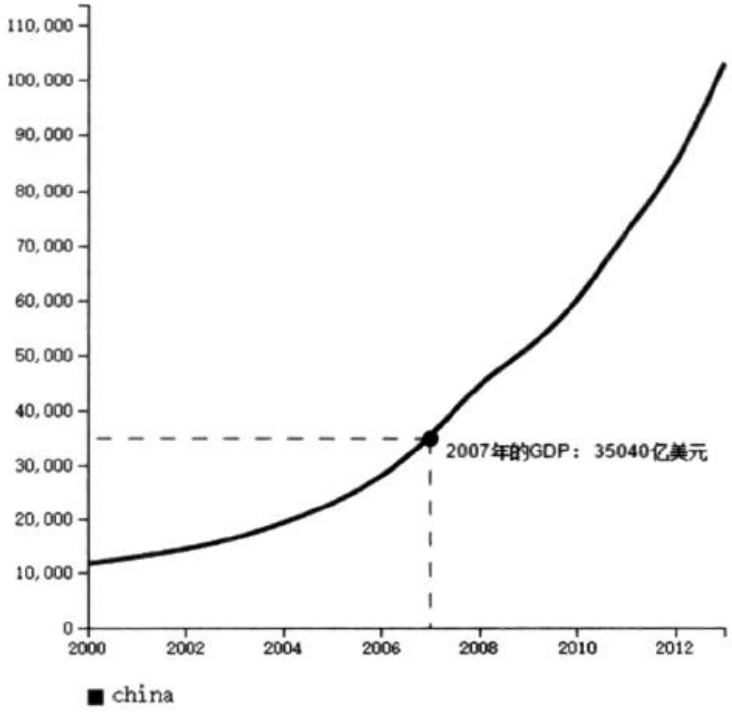


图 12-9　鼠标滑动到 2007 年附近时显示出的焦点和对齐线

12.2.2　为折线图添加提示框

第 12.1 节中的提示框可以作为显示焦点文字的容器，这样能够容纳更多的信息，而且变得更友好。对图 12-9 的折线图，将其交互方式做一些修改，实现以下功能。

（1）鼠标滑动时显示一条垂直于 x 轴的对齐线。

（2）鼠标滑动时显示一个提示框，跟随鼠标移动，内容显示折线图中焦点的值。

提示框的模型如图 12-10 所示，最上面一行显示“年份”，下面分为数行，每一个焦点显示一行，每一行从左至右分别表示折线的颜色、国家名称、GDP 值。

图 12-10　折线图的提示框

最外层的提示框当然是 元素，里面的部分也都用 元素，再分别添加 CSS 样式即可。提示框的结构如图 12-11 所示，写成 HTML 元素的代码如下：

```
<div class="tooltip">
    <div class="title"></div>
    <div>
        <div class="desColor"></div>
        <div class="desText"></div>
    </div>
    <div>
        <div class="desColor"></div>
        <div class="desText"></div>
    </div>
</div>
```

最外层 的类名为 tooltip，其样式使用第 12.1 节的 tooltip 即可，重点还是要将 position 设置为 absolute。

再设定另外三个 div 的样式：title 使用 border-bottom 属性增加一个下画线，文字居中；为 desColor 设置定量的宽度和高度，并向左浮动，且通过 margin 与周边元素保持间距，但是不设置背景颜色，留着在程序中设定；desText 设置为行内元素。代码如下：

```
.tooltip .title{
```

```
    border-bottom: 1px solid #000;
    text-align: center;
}

.tooltip .desColor{
    width: 10px;
    height: 10px;
    float: left;
    margin: 9px 8px 1px 8px;
}

.tooltip .desText{
    display: inline;
}
```

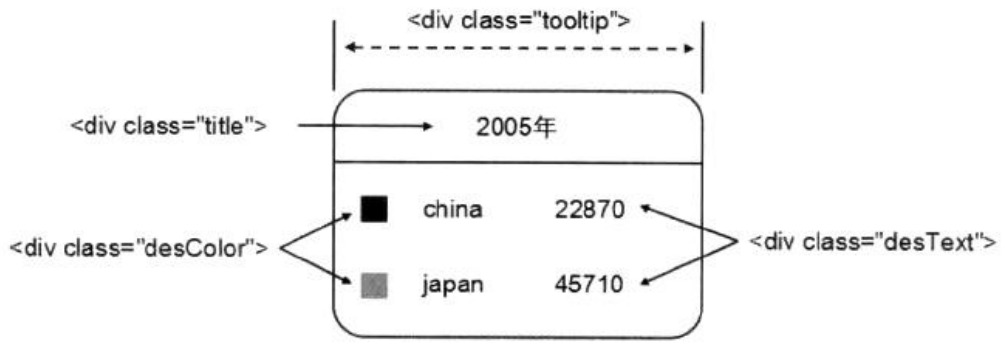


图 12-11　提示框的 div 结构

首先，添加提示框的元素，并分别设定属性 class。添加元素的时候不需要设定位置，留着在鼠标事件触发时再设定。代码如下：

```
//添加一个提示框
var tooltip = d3.select("body")
                .append("div")
                .attr("class","tooltip")
                .style("opacity",0.0);

var title = tooltip.append("div")
                .attr("class","title");

var des = tooltip.selectAll(".des")
                .data(dataset)
                .enter()
                .append("div");

var desColor = des.append("div")
```

```
                    .attr("class","desColor");

var desText = des.append("div")
                    .attr("class","desText");
```

其次，添加一条垂直于 x 轴的对齐线。

```
//添加垂直于x轴的对齐线
var vLine = svg.append("line")
               .attr("class", "focusLine")
               .style("display", "none");
```

然后，参照上一节的步骤，添加一个透明矩形用于捕获鼠标事件，并分别为其添加鼠标移入、鼠标移出、鼠标滑动的监听器。鼠标移入时，显示元素；鼠标移出时，隐蔽元素。

```
.on("mouseover", function() {
   tooltip.style("left", (d3.event.pageX) + "px")
          .style("top", (d3.event.pageY + 20) + "px")
          .style("opacity",1.0);
   vLine.style("display",null);
})
.on("mouseout", function() {
   tooltip.style("opacity",0.0);
   vLine.style("display","none");
})
.on("mousemove", mousemove);
```

其中最重要的仍然是 mousemove 的监听器，该事件触发时要**更新提示框和对齐线的位置**。最后，添加 mousemove 事件的监听器，代码如下：

```
function mousemove() {

/************************************

   与第12.2.1节的代码相同，
   故省略部分代码。

   根据鼠标事件的坐标，计算焦点对应的年份

   ************************************/

   //获取年份和gdp数据
   var year = x0;
   var gdp = [];
```

```
    for(var k=0; k<dataset.length; k++ ){
        gdp[k] = { country: dataset[k].country,
                   value: dataset[k].gdp[index][1]};
    }

    //设置提示框的标题文字（年份）
    title.html("<strong>" + year + "年</strong>");

    //设置颜色标记的颜色
    desColor.style("background-color",function(d,i){
            return colors[i];
    });

    //设置描述文字的内容
    desText.html( function(d,i){
        return gdp[i].country + "\t" + "<strong>" +
                gdp[i].value + "</strong>";
    });

    //设置提示框的位置
    tooltip.style("left", (d3.event.pageX) + "px")
            .style("top", (d3.event.pageY + 20) + "px");


    //获取垂直对齐线的x坐标
    var vlx = xScale(data[index][0]) + padding.left;

    //设定垂直对齐线的起点和终点
    vLine.attr("x1", vlx)
        .attr("y1",padding.top)
        .attr("x2",vlx)
        .attr("y2",height - padding.bottom);
}
```

上面的代码分为两部分。第一部分是设置提示框的属性：先用计算得到的年份和 GDP 值，分别为提示框的各部分赋值（标题、颜色标记、描述文字），然后再设置提示框的位置。第二部分是设置对齐线的坐标。

结果如图 12-12 和图 12-13 所示，分别是鼠标移动到 2005 年和 2009 年的区域时出现的提示框和对齐线，可以看到友好度得到了明显提升。此外，还可以为对齐线与折线的交点添加<circle>元素，能够明示出焦点的位置，这里就不阐述了。

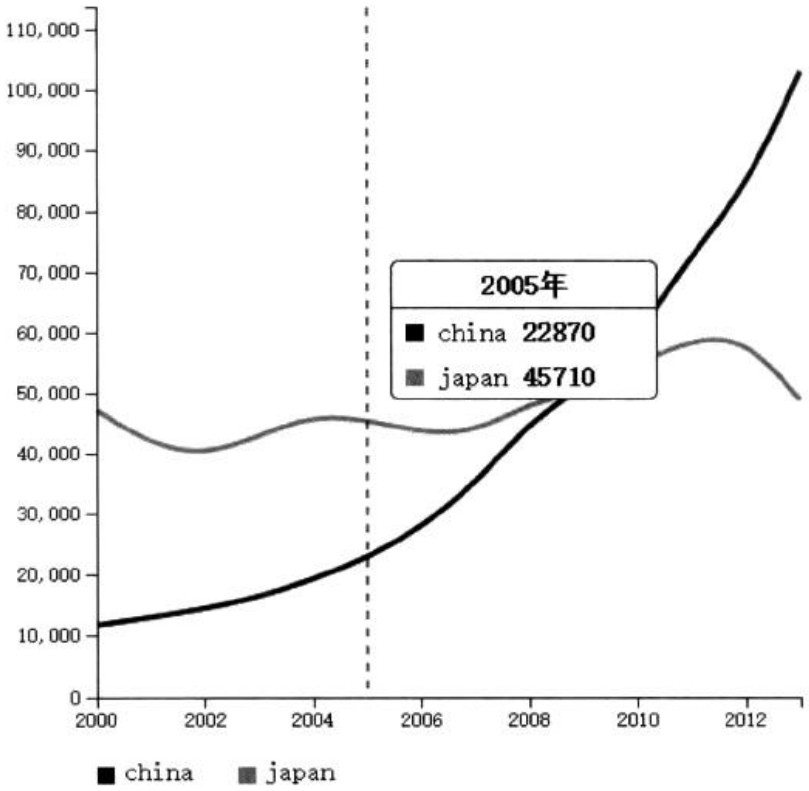


图 12-12　鼠标移到“2005 年对应的区域”附近

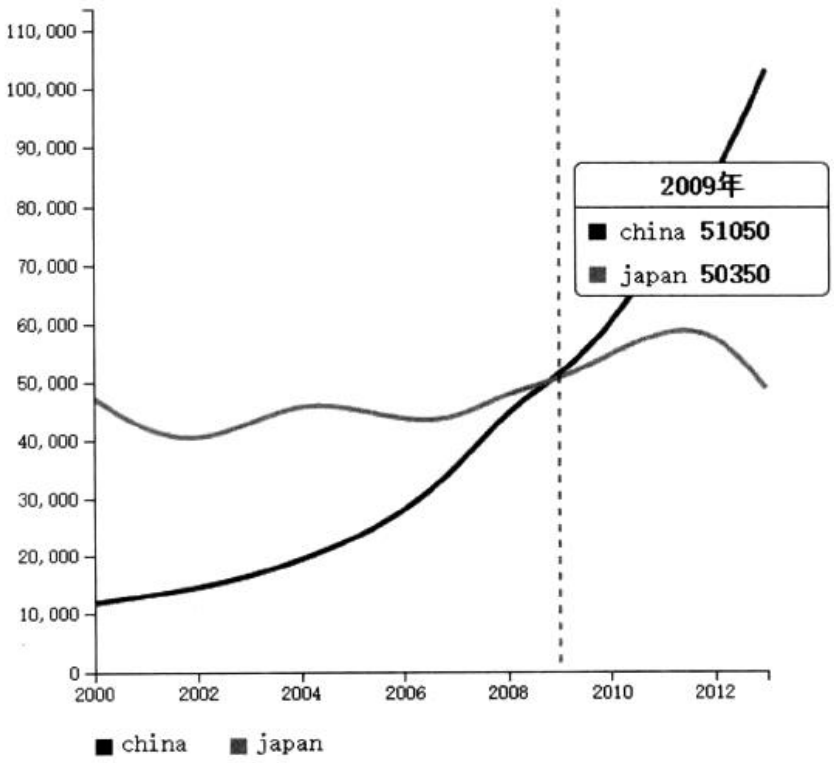


图 12-13　鼠标移到“2009 年对应的区域”附近

12.3 元素组合

在一个图表里，有时需要减少一个元素，有时需要将两个元素合并起来，元素经过组合后数值和图表发生相应的变化。以饼状图为例，图 12-14 展示了将饼状图中的某一个元素移出饼状图的情况，图 12-15 展示了将 A 和 B 合并在一起的情况。

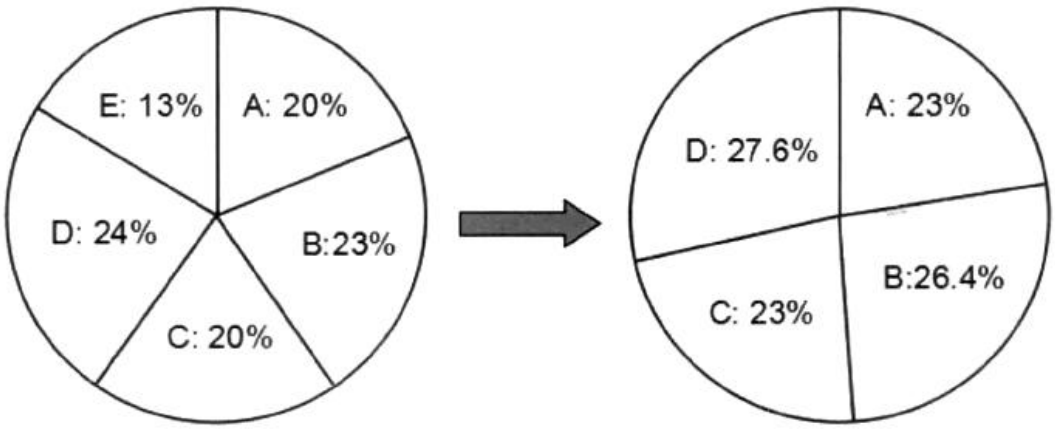


图 12-14 将左图的“E”元素从饼状图中移出去，变为右图

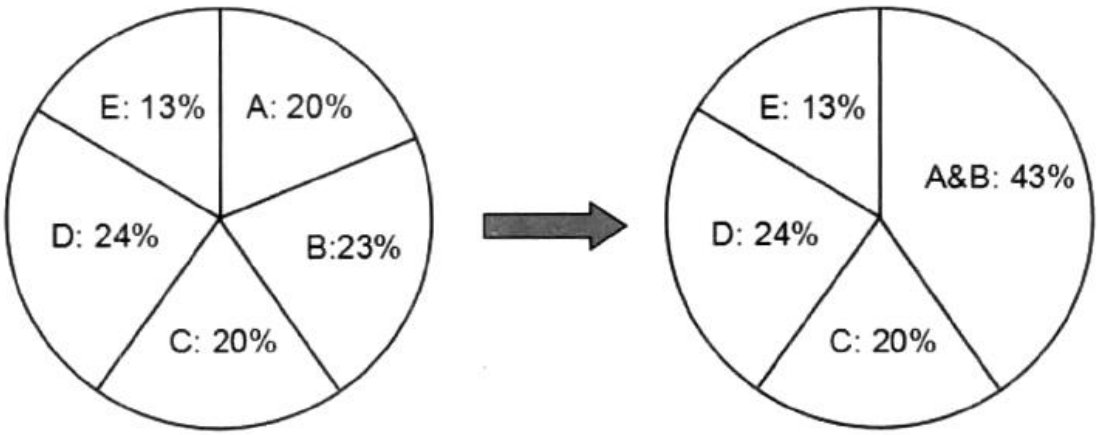


图 12-15 将 A 和 B 合并在一起

使用鼠标的拖曳功能来完成图形元素的组合是比较好的选择，也可以同时考虑使用触摸屏的事件。本节将依次讲解三种手法。

（1）如何拖曳饼状图中的元素。

（2）如何将饼状图的某个元素移出饼状图，并令饼状图发生相应的变化。

（3）如何将饼状图的两个元素合并起来。

12.3.1　饼状图的拖曳

使用拖曳行为 d3.behavior.drag 能完成图形元素的拖曳移动。以图 12-1 的饼状图为基础添加内容。

首先，添加弧的分组元素的时候，为每一个分组都添加两个属性：dx 和 dy，代表弧的平移量，并应用到 transform 属性上。代码如下：

```
var arcs = svg.selectAll("g")
              .data(piedata)        //绑定转换后的数据piedata
              .enter()
              .append("g")
              .each(function(d){
                  d.dx = width/2;
                  d.dy = height/2;
              })
              .attr("transform","translate("+( width/2 )+","+
                                  ( height/2 ) +")");
```

其次，定义拖曳行为，origin 设置为 null，表示拖曳点与元素之间没有偏移量。当 drag 事件触发时，调用监听器 dragmove()。每次 dragmove 事件触发时，都将鼠标的偏移量分别加到 d.dx 和 d.dy 中，得到弧元素新的偏移量。

然后，给被触发事件的图形元素的 transform 属性设置新的 dx 和 dy，即可将弧平移到新的位置。代码如下：

```
var drag = d3.behavior.drag()
             .origin(null)
             .on("drag", dragmove);

function dragmove(d) {
   d.dx += d3.event.dx;          //加鼠标的x方向偏移量
   d.dy += d3.event.dy;          //加鼠标的y方向偏移量
   d3.select(this)
          .attr("transform","translate("+d.dx+","+d.dy+")");
}
```

最后，让弧的选择集通过 call()调用 drag 行为：

```
arcs.call(drag);
```

结果如图 12-16 所示，“其他”被向右上方拖曳了一段距离。有一件事要明白：饼状图的每一条弧的位置都是由 transform 属性来确定平移方向和大小的。本例为每一条弧添加了两个变量

dx 和 dy，用来代表平移量。鼠标拖曳某元素后，某元素看起来移动了，其实只是 transform 属性的值改变了，并不是改变了饼状图布局计算出来的数据（即角度、半径等）。例如，此次平移后，各段弧的 transform 属性分别为（属性值为假设）：

酷派，transform= 300, 300　（初始平移值）

其他，transform= 330, 270　（新平移值）

小米，transform= 300, 300　（初始平移值）

因此，本例的拖曳其实是通过更改 transform 属性实现的。

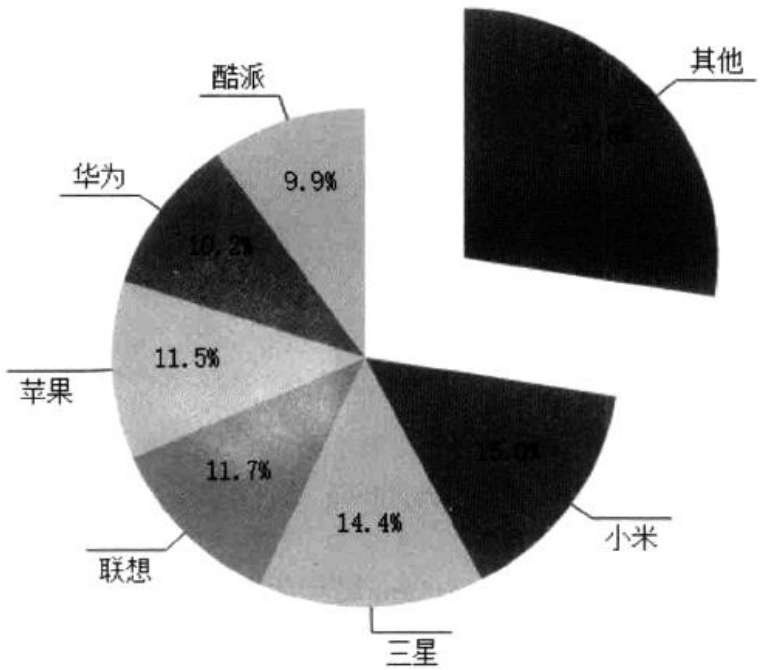


图 12-16　将“其他”元素向右上拖曳一段距离

12.3.2　移入和移出

无论是将饼状图的元素移入还是移出图表，原图表中的数据是发生了更改的，请回想第4.5.3节的处理模板，当选择集中的数据发生更新时应该怎么做。

由于移入和移出元素涉及数据的更新，故每次数据发生更新时，都要调用饼状图布局重新计算饼状图的位置。因此，需要定义一个重绘函数 redraw，使其能够适应当数据发生变化（增加或删除）时，自动更新图形元素的属性。程序的流程如下：

原数组→调用布局绘制饼状图→移出元素→减少原数组元素→调用重绘函数→得到新的饼状图

重绘函数中，先分别获取弧的 update、enter、exit 部分，然后再依次处理这三个部分。其中，update 部分的操作是更新元素的属性，enter 部分的操作是添加弧元素，exit 部分的操作是

删除元素。代码如下：

```
function redraw(){ //重绘函数

    //获取update部分，piedata为饼状图的数组
    var arcsUpdate = svg.selectAll(".arcGroup")
                .data(piedata,function(d){  return d.data[0]; });

    //获取enter部分
    var arcsEnter = arcsUpdate.enter();

    //获取exit部分
    var arcsExit = arcsUpdate.exit();

    //1. update部分的处理办法，设定新的属性
    arcsUpdate.call(setAttributes);

    //2. enter部分的处理办法
    var newArcs = arcsEnter.append("g")
                           .attr("class","arcGroup");

    //添加弧
    newArcs.append("path")
           .attr("class","arcPath");

    //添加弧内文字
    newArcs.append("text")
           .attr("class","percent");

    //添加弧外文字
    newArcs.append("text")
           .attr("class","company");

    //添加连接文字的直线1
    newArcs.append("line")
           .attr("class","conLine1");

    //添加连接文字的直线2
    newArcs.append("line")
           .attr("class","conLine2");

    //设定属性
    newArcs.call(setAttributes);

    //3. exit部分的处理办法，删除元素
```

```
    arcsExit.remove();
}
```

每一段弧的分组元素<g>里，包含的元素包括：

- 路径元素<path>，有宽度的一段弧。
- 两个文字元素<text>，分别显示“百分比”和“厂商”。
- 两个线段元素<line>，连接弧和文字。

由于处理 enter 部分时，添加元素后要为属性赋值，处理 update 部分时也要为元素的属性赋值，因此为属性赋值的功能可以写在一个 setAttributes()函数里。因此，上面的代码有以下两条语句：

```
arcsUpdate.call(setAttributes);     //为已更新的弧的属性赋值
newArcs.call(setAttributes);        //为新添加的弧的属性赋值
```

这种 call 的用法已经说过，等同于：

```
setAttributes(arcsUpdate);
```

exit 部分的处理办法很简单，就是删除元素。下面来看 setAttributes()的内容，代码如下：

```
function setAttributes(arcs){  //设置弧的属性

    /*
```

为每一段弧添加三个数据：

```
circle——饼状图所在的圆，其中包含cx、cy、r三个属性
dx——x方向的相对偏移量，拖曳事件触发时使用
dy——y方向的相对偏移量，拖曳事件触发时使用
*/
arcs.each(function(d){
    d.circle = piecircle;        //piecircle是定义饼状图所在圆的变量
    d.dx = 0;
    d.dy = 0;
});

//将饼状图平移到指定位置
arcs.attr("transform",function(d){
    return "translate("+ d.circle.cx +","+ d.circle.cy +")";
});

//绘制弧
arcs.select(".arcPath")
```

```
    .attr("fill",function(d,i){
        return color(d.data[0]);    //设定弧的颜色
    })
    .attr("d",function(d){
        return arc(d);              //使用弧生成器
    });


/*************************
省略为其他元素赋值的代码
*************************/

}
```

其中，变量 piecircle 的定义如下：

```
var piecircle = {
    cx: width/2,
    cy: height/2,
    r: outerRadius
};
```

定义这个变量的目的是为了方便判定某个元素是否移入或移出了饼状图，这一点后面会做说明。接下来，定义拖曳行为，部分内容与第 12.3.1 节类似，只是这一次要添加 dragend 事件的监听器，它监听的内容是：

当某一段弧被拖出到饼状图之外时，做两件事：**一是更新原来的饼图，使其达到图 12-14 的效果；二是将被拖出去的弧变成一个圆，使其显示在饼状图之外。**

拖曳行为的定义如下所示：

```
//定义拖曳行为
var drag = d3.behavior.drag()
                .origin(null)
                .on("drag", dragmove)
                .on("dragend",dragend);

//正在拖曳事件（drag）被触发时的监听器
function dragmove(d) {
    d.dx += d3.event.dx;    //加鼠标x方向的偏移量
    d.dy += d3.event.dy;    //加鼠标y方向的偏移量

    //为被拖曳的弧的平移属性设定新的值
```

```
    d3.select(this)
        .attr("transform","translate("+ ( d.dx + d.circle.cx ) +
                          "," + ( d.dy + d.circle.cy ) +")");
}

//拖曳结束事件（dragend）被触发时的监听器
function dragend(d,i) {

        //计算被拖曳的元素到饼状图圆心距离的平方
        var dis2 = d.dx * d.dx + d.dy * d.dy;

        if( dis2 > d.circle.r * d.circle.r ){
        //如果被拖到了饼状图之外

         /*
           删除原数组dataset中的第i个元素（即被拖曳元素）
           被删除的元素保存在movedData中
         */
        var movedData = dataset.splice(i,1);

        //重新调用布局计算dataset数组，结果保存在piedata中
        piedata = pie(dataset);

        //添加一个圆，用于表示被移出的弧
        appendCircle(movedData[0]);

        //重绘
        redraw();
    }
}

//所有弧调用上面定义的拖曳行为
arcs.call(drag);
```

拖曳结果事件发生时，除了更新原来的饼状图之外，还要将被移出去的弧变成一个圆，如图 12-17 所示。另外，还有一个问题，某段弧被移出去了，饼状图数组中的数据就少了一项，但是这个数据是需要保存下来的，因为以后还有可能将移出去的弧再移进来。在上面的代码里，被删除的数据保存在 movedData 里，而 movedData 又作为 appendCircle()的参数使用了，也就是说在 appendCircle()里必须实现将 movedData 与新添加的元素绑定起来。

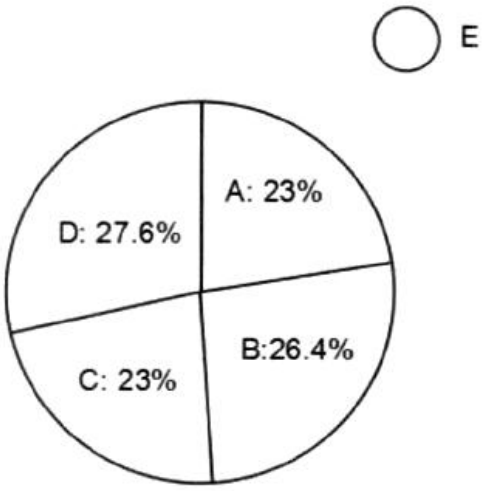


图 12-17　移出去的弧显示成一个圆

接下来讲解 appendCircle()里的代码，要添加一个圆和文字来表示移出去的元素，SVG 结构如下：

```
<g>     //数据绑定到这
<circle></circle>
<text></text>
</g>
```

一个分组元素里带有<circle>和<text>，数据绑定到<g>上，以便将来再次移入饼状图的时候使用。单个元素绑定单个数据，使用 datum()即可，代码如下：

```
//在svg里添加一个<g>，用于容纳所有移出来的弧
var circleGroups = svg.append("g");

function appendCircle(data){

    //为拖曳出来的图形增加新元素
    gCircle = circleGroups.append("g")
                .datum(data)     //绑定被移出的数据
                .attr("class","movedArc")
                .attr("transform","translate(" +
                                    d3.event.sourceEvent.offsetX + ","
                                  + d3.event.sourceEvent.offsetY +")")
                .call(dragCircle);

    //添加一个圆
    gCircle.append("circle")
           .attr("cx", 0)
           .attr("cy", 0)
```

```
            .attr("r", 20)
            .style("fill",function(d){
                return color(d[0]);
            });

    //添加文字
    gCircle.append("text")
            .attr("dx", "22px")
            .attr("dy", ".4em")
            .text(function(d){
                return d[0];
            });
}
```

如图 12-18 所示，将“小米”移出去后，饼状图发生了变化，反映出了在没有“小米”的情况下，其他各厂商的市场份额。

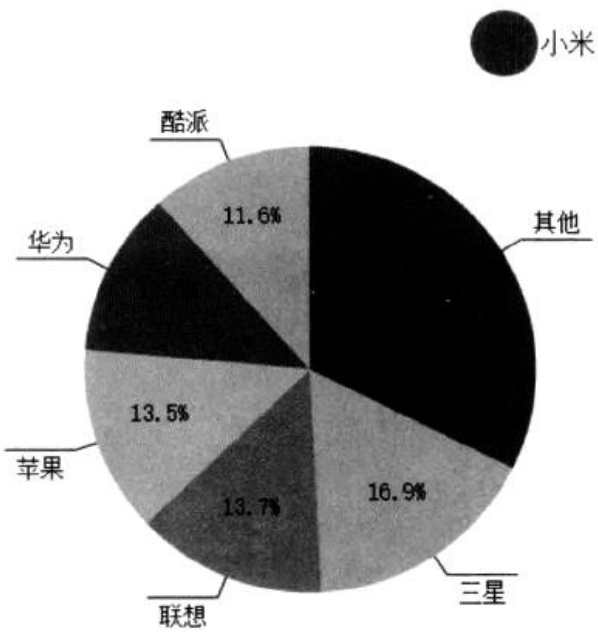


图 12-18　将“小米”从饼状图移出去后

图 12-19 展示了依次将“小米”、“三星”、“联想”、“苹果”、“其他”移出后，只剩下“酷派”和“华为”的饼状图，这就反映了“酷派”和“华为”的对比情况。也就是说，通过添加这样的交互式操作，可视化的内容增加了。

有“移出”，就有“移入”，用户可能需要将移出的弧再次移入到饼状图中。在 appendCircle() 函数里有这样一行代码：

```
gCircle.call(dragCircle);
```

给移出去的图形元素也添加了一个拖曳行为，可以想象，该行为里也要定义 drag 事件的监听器，用于移动元素，以及 dragend 事件的监听器，用于判断当拖曳结束且元素位于饼状图之内时，要重新绘制饼状图。这一点和移出来时定义的拖曳行为是很像的。

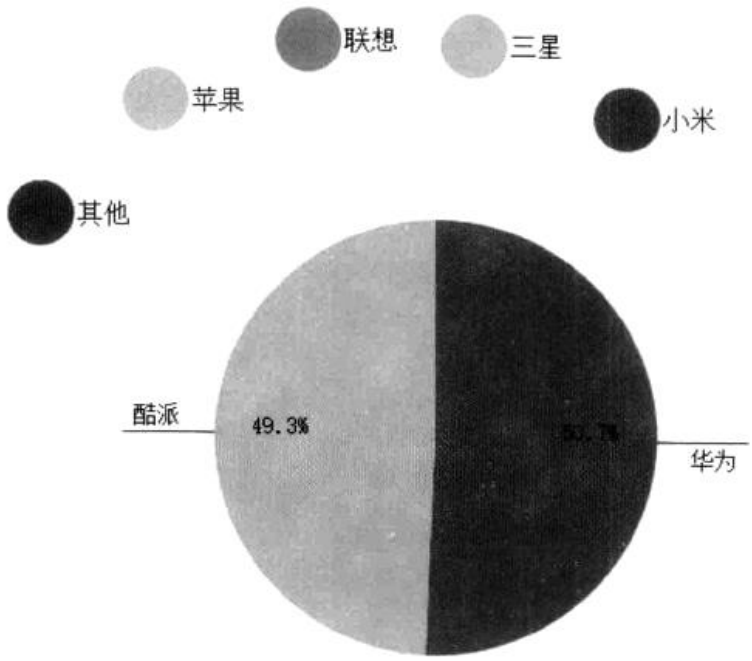


图 12-19　依次移出“小米”、“三星”、“联想”、“苹果”、“其他”之后

关于被移出元素的拖曳行为，主要 dragend 的监听器。在这里面需要判断当拖曳结束时，该元素是否处在饼状图的内部，如果是，就重新绘制饼状图，否则，继续保持原样，这一点与将它移出来的时候是类似的。通过计算与圆心的距离，即可判断是否位于圆内。代码如下所示：

```
//已经被移出去的元素的拖曳行为
var dragCircle = d3.behavior.drag()
                        .origin(null)
                        .on("drag", dragCircleMove)
                        .on("dragend",dragCricleEnd);

//正在被拖曳时的监听器
function dragCircleMove(d) {
    d.x = d3.event.sourceEvent.offsetX;
    d.y = d3.event.sourceEvent.offsetY;

    d3.select(this)
        .attr("transform","translate("+ d.x + "," + d.y +")");
}

//拖曳结束时的监听器
```

```
function dragCricleEnd(d,i) {
    //计算到圆心距离的平方
    var dis2 = ( d.x - piecircle.cx ) * ( d.x - piecircle.cx ) +
               ( d.y - piecircle.cy ) * ( d.y - piecircle.cy );

    if( dis2 < piecircle.r * piecircle.r ){
        //如果拖曳结束时元素在饼状图之内

        //将绑定的数据重新添加到dataset里
        dataset.push([d[0],d[1]]);

        //使用布局重新计算饼状图
        piedata = pie(dataset);

        //删除移出去时添加的圆和文字
        d3.select(this).remove();

        //调用重绘函数
        redraw();
    }
}
```

如此一来，被移出元素可以自由地经拖曳移入饼状图，如图 12-20 所示。

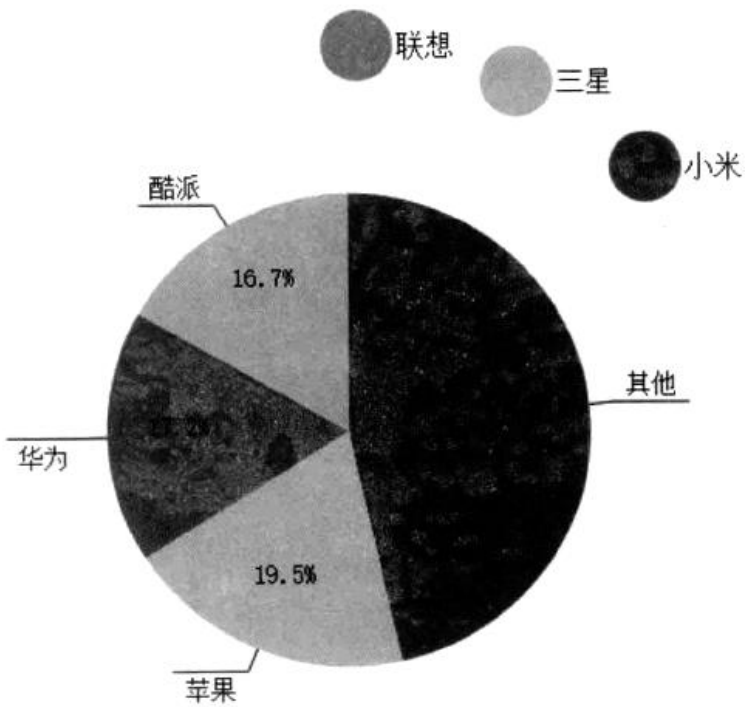


图 12-20　将“其他”和“苹果”再次移入饼状图之后

打开控制台可以看到此时的 HTML 文档结构如图 12-21 所示。class 为 arcGroup 的元素，就是饼状图的弧，此时有四个，即“酷派”、“华为”、“苹果”、“其他”。class 为 moveArc 的就是被移出去的元素，此时有三个，即“小米”、“三星”、“联想”。

图 12-21　HTML 的文档结构

12.3.3　合并

本节要实现图 12-15 的功能，将两段弧合并在一起。上一节将移出去的元素再次移入饼状图，当拖曳结束时判断了元素的位置是否在饼状图之内，是则移入。本节要讲述的合并方法与其类似，首先要判断移入的元素落在哪一段弧的范围内，然后将数据与那一段弧合并，再调用布局重新计算饼状图。因此，只需要在上一节的代码中修改 dragCricleEnd()即可，代码如下：

```
function dragCricleEnd(d,i) {
    //计算到圆心距离的平方
    var dis2 = ( d.x - piecircle.cx ) * ( d.x - piecircle.cx ) +
               ( d.y - piecircle.cy ) * ( d.y - piecircle.cy );

    if( dis2 < piecircle.r * piecircle.r  ){
        //如果拖曳结束时元素在饼状图之内

        //计算与y轴的夹角
        var vec = { x: d.x - piecircle.cx , y: d.y - piecircle.cy };
        var zerov = { x: 0.0 , y: -1.0 };
        var costheta = ( vec.x * zerov.x + vec.y * zerov.y ) /
        ( norm(vec) * norm(zerov) );
        var theta = Math.acos(costheta);
        theta = d.x < piecircle.cx ? 2*Math.PI - theta : theta;

        //通过比较theta和startAngle、endAngle来判断落到哪一段弧上
```

```
            //弧的索引号保存在index里
            var index;
            for (var j = 0; j < piedata.length; j++ ) {
                if( theta >= piedata[j].startAngle &&
                    theta <= piedata[j].endAngle ){
                    index = j;
                    break;
                }
            }

            //给原数组中增加数据
            dataset[index][0] += " & " + d[0];
            dataset[index][1] += d[1];

            //调用布局重新计算饼状图
            piedata = pie(dataset);

             //删除移出时添加的圆和文字
            d3.select(this).remove();

             //重绘
            redraw();
        }
    }
```

结果如图 12-22 所示，将“小米”从饼状图移出去之后，再次将其移入时，即可与某段弧合并，这一次移到了“华为”上，变成了“华为 & 小米”。如此一来，就可以看到小米和华为加在一起的市场份额。

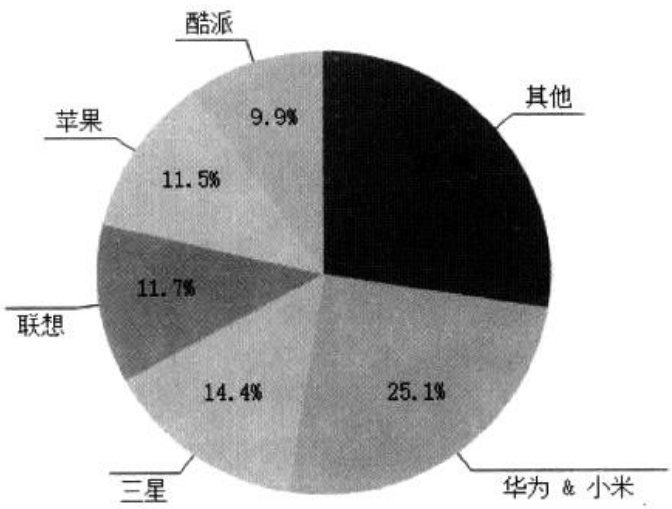


图 12-22　合并“华为”和“小米”

12.4　区域选择

在 Windows 操作系统的画图板程序里，有“选择框”工具。按下鼠标左键，拖动鼠标，显示出一个选择框，该矩形框随着鼠标的移动而改变大小。放开鼠标左键后，矩形位置和大小就确定了，将鼠标移到矩形的右下角，显示出一个两端带有箭头的线段，再次按下鼠标左键即可改变矩形的大小。

像这样，选择一个区域，对区域内的元素进行操作，是一种常见的交互式行为，在可视化图表中，也可以为用户添加此操作。这种功能如果手动实现很复杂，很幸运地，D3 提供了 brush（刷子）控件，应用起来就非常简单了。

- d3.svg.**brush**()

创建一个刷子。

- **brush**(selection)

在 selection 里添加与刷子相关的元素，selection 通常是<g>的选择集。

- brush.**x**([scale])

设定或获取 *x* 轴的比例尺。

- brush.**y**([scale])

设定或获取 *y* 轴的比例尺。

- brush.**extent**([values])

设定或获取刷子的范围，如果设置了 values 值，则指定一个范围，否则返回当前的范围。如果只设置了 *x* 轴的比例尺，则 values 的格式为[x0, x1]，x0 和 x1 分别是 *x* 方向的下限和上限；如果只设置了 *y* 轴的比例尺，则 values 的格式为[y0, y1]，y0 和 y1 分别是 *y* 方向的下限和上限；如果两个比例尺都设置了，则 values 的格式为[[x0, y0], [x1, y1]]。

- brush.**clear**()

清空当前刷子的范围（extent）。

- brush.**empty**()

如果刷子的范围是空，则返回 true，否则返回 false。

- brush.**on**(type[, listener])

设定刷子的监听器，有三种事件类型。

  - brushstart——鼠标键按下时。
  - brush——鼠标键按下拖动时。
  - brushend——鼠标键放开时。

12.4.1 在SVG画板里选择一块区域

首先，要学会如何在SVG里创建选择框（刷子）。其实选择框就是用SVG中的元素和事件模拟出来的，并非什么新的东西。下面制作一个例子。

首先，给HTML的<body>里添加一个<svg>，再添加一个圆<circle>和矩形<circle>作为参照物：

```
var width = 500;
var height = 500;

var svg = d3.select("body")
            .append("svg")
            .attr("width", width)
            .attr("height", height);

svg.append("circle")
   .attr("cx",100)
   .attr("cy",100)
   .attr("r",30)
   .style("fill","black");

svg.append("rect")
   .attr("x",150)
   .attr("y",70)
   .attr("width",70)
   .attr("height",60)
   .style("fill","black");
```

由刷子的函数介绍可知，定义刷子需要 x 轴和 y 轴的比例尺，虽然图表中不一定有“坐标轴”，但是两个比例尺是需要的：

```
var xScale = d3.scale.linear()
               .domain([0, width])
               .range([0, width]);

var yScale = d3.scale.linear()
               .domain([0, height])
               .range([0, height]);
```

比例尺的值域（range）可以理解为刷子的范围，从上述比例尺的定义可知，刷子的范围为：x 方向从0到width，y 方向从0到height，即覆盖整个SVG的范围。那么比例尺的定义域（domain）与刷子有什么关系呢？我们知道，比例尺的作用是使得数据可以按比例缩放，假如将 x 轴比例尺的定义域和值域设置为：

```
domain: [0, 10]
range:  [0, 100]
```

拖出一个选择框，x 方向的范围为[30, 70]，但实际输出的范围是[3, 7]。也就是说，根据 x 轴比例尺的反函数计算出了一个定义域内的范围。

然后，创建一个刷子，并将前面的比例尺作为参数：

```
var brush = d3.svg.brush()
              .x(xScale)
              .y(yScale)
              .extent([[0, 0], [100, 100]])
              .on("brush",brushed);

function brushed(){
   var extent = brush.extent();
   console.log( "x方向的下限:  " + extent[0][0] );
   console.log( "x方向的上限:  " + extent[1][0] );
   console.log( "y方向的下限:  " + extent[0][1] );
   console.log( "y方向的上限:  " + extent[1][1] );
}
```

brush 的初始范围被设置为一个宽和高都是 100 的正方形。当 brush 事件触发时，调用 brushed 函数，在这里仅仅是输出刷子的范围。

创建刷子后，还需要在<svg>中添加刷子的相关元素，一般是将所有元素放到一个分组<g>里。在 SVG 里 append 一个 g，通过 call 调用刷子：

```
svg.append("g")
   .call(brush)
   .selectAll("rect")
   .style("fill-opacity",0.3);
```

这段代码还选择了<g>中的<rect>元素，然后设置其透明度。由于刷子就是由一系列 SVG 的图形元素组成的，因此可以为其设置样式。

结果如图 12-23 所示，左上角的矩形是刷子的初始范围，中间是一圆一方两个参照物。

鼠标在 SVG 区域的任意范围内移动（由两个比例尺确定的有效范围），光标都会变成十字形。在空白处单击鼠标键可取消当前选择框，按下拖动可产生新的选择框。如图 12-24 所示，在圆和方形之间拖出了一个新的选择框。此时，控制台的输出如下，即刷子的范围：

x 方向的下限：51

x 方向的上限：189

y 方向的下限：47

y 方向的上限：164

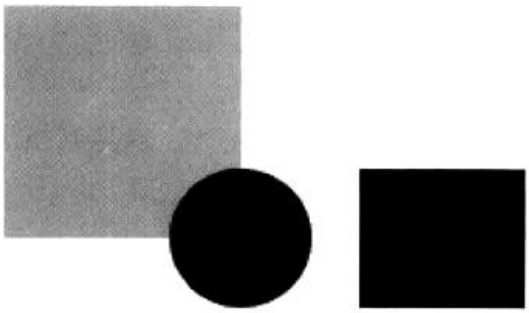


图 12-23　刷子的初始范围

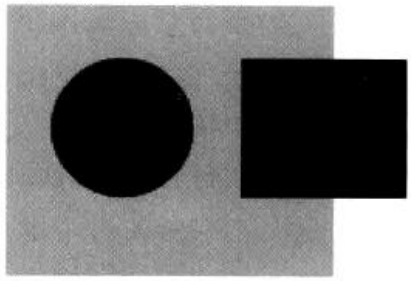


图 12-24　用鼠标拖出一个新的选择框

前面提到，选择框是用 SVG 的元素构成的。如图 12-25 所示，可以看到，在上面添加的分组<g>里，添加了一大堆元素，有两个<rect>和八个<g>，<g>里面还有<rect>。不必理会刷子具体是怎样构成的，只要知道要修改选择框的样式时，需要选中刷子元素中所有的<rect>即可。

```
▼<svg width="500" height="500">
   <circle cx="100" cy="100" r="30" style="fill: rgb(0, 0, 0);"></circle>
   <rect x="150" y="70" width="70" height="60" style="fill: rgb(0, 0, 0);"></rect>
  ▼<g style="pointer-events: all; -webkit-tap-highlight-color: rgba(0, 0, 0, 0);">
     <rect class="background" x="0" width="500" y="0" height="500" style="visibility: hidden;
     cursor: crosshair; fill-opacity: 0.3;"></rect>
     <rect class="extent" x="51" width="138" y="47" height="117" style="cursor: move; fill-
     opacity: 0.3;"></rect>
    ▶<g class="resize n" transform="translate(51,47)" style="cursor: ns-resize;">…</g>
    ▶<g class="resize e" transform="translate(189,47)" style="cursor: ew-resize;">…</g>
    ▶<g class="resize s" transform="translate(51,164)" style="cursor: ns-resize;">…</g>
    ▶<g class="resize w" transform="translate(51,47)" style="cursor: ew-resize;">…</g>
    ▶<g class="resize nw" transform="translate(51,47)" style="cursor: nwse-resize;">…</g>
    ▶<g class="resize ne" transform="translate(189,47)" style="cursor: nesw-resize;">…</g>
    ▶<g class="resize se" transform="translate(189,164)" style="cursor: nwse-resize;">…</g>
    ▶<g class="resize sw" transform="translate(51,164)" style="cursor: nesw-resize;">…</g>
     </g>
 </svg>
```

图 12-25　组成刷子的 SVG 元素

12.4.2　散点图的区域选择

知道了刷子是怎样工作的，本节将以散点图为例，讲述如何修改选择框选中的元素。

首先，定义两个比例尺，此次不将定义域和值域设置成相同值：

```
//边界空白
var padding = {left: 50, right: 50, top: 50, bottom: 50};

//x轴的比例尺
var xScale = d3.scale.linear()
               .domain([0, 10])
               .range([padding.left, width - padding.right]);

//y轴的比例尺
```

```
var yScale = d3.scale.linear()
                .domain([10, 0])
                .range([padding.top, height - padding.bottom]);
```

以此比例尺定义的刷子所输出的范围，x 方向在[0, 10]，y 方向在[10, 0]内。

其次，向<svg>中随机添加散点，并分别添加 x 轴和 y 轴，添加坐标轴的代码省略：

```
//散点图的数据
var dataset = [];

for(var i=0; i<150; i++){
    dataset.push([Math.random()*10, Math.random()*10]);
}

//添加散点
var circles = svg.selectAll("circle")
                 .data(dataset)
                 .enter()
                 .append("circle")
                 .attr("cx",function(d){
                   return xScale(d[0]);
                 })
                 .attr("cy",function(d){
                   return yScale(d[1]);
                 })
                 .attr("r",5)
                 .style("fill","black");
```

然后，定义一个刷子，并设定：当 brush 事件触发时，调用 brushed 函数。函数的内容是：对散点图中的所有点，如果其坐标值在选择框的范围内，则设置其填充样式为红色，否则保持黑色。代码如下：

```
var brush = d3.svg.brush()
              .x(xScale)
              .y(yScale)
              .extent([[0, 0], [0, 0]])
              .on("brush",brushed);

function brushed(){

    //选择框的范围
    var extent = brush.extent();
    var xmin = extent[0][0];
    var xmax = extent[1][0];
    var ymin = extent[0][1];
    var ymax = extent[1][1];
```

```
    circles.style("fill",function(d){
        //如果散点的坐标在选择框范围内，变为红色，否则为黑色
        if( d[0] >= xmin && d[0] <= xmax &&
            d[1] >= ymin && d[1] <= ymax ){
            return "red";
        }else{
            return "black";
        }
    });
}
```

最后，添加一个<g>，将所有与刷子相关的元素都放到<g>里，并设置刷子里的所有<rect>的填充透明度为 0.3。

```
svg.append("g")
    .call(brush)
    .selectAll("rect")
    .style("fill-opacity",0.3);
```

结果如图 12-26 所示，只能够在两个坐标轴之间的区域拖曳出选择框，这是由比例尺的值域决定的。两坐标轴之间有很多散点，原为黑色，拖出一个选择框后，框内的元素变成了红色，外部的仍为黑色。

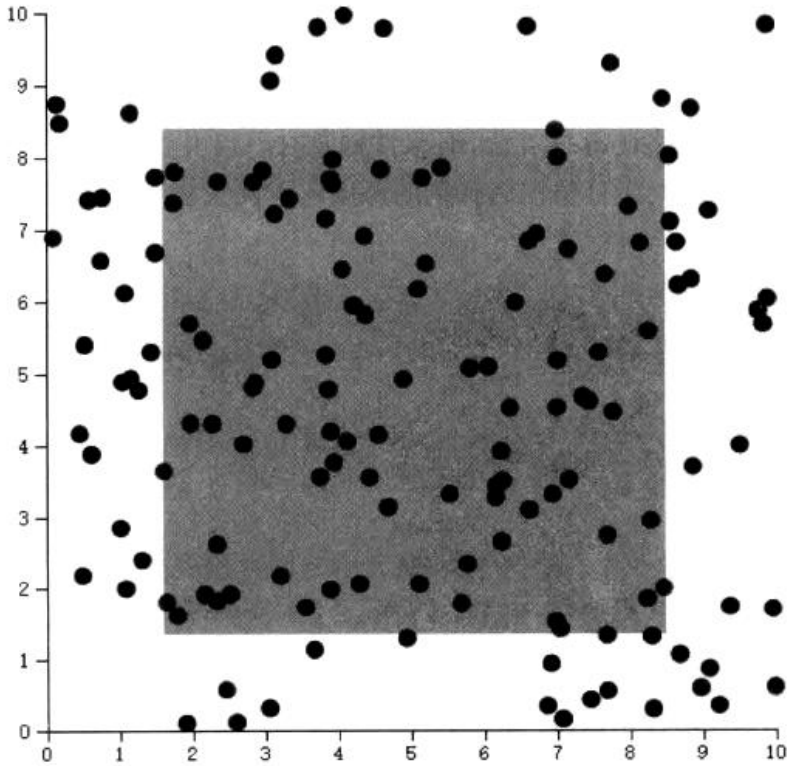


图 12-26　选择框内的散点变成了红色

12.5　开关

在图表中，有这样一种交互式操作：点击某个元素，会出现新的元素，再点击一次该元素，刚出现的新元素又会被隐藏。这种交互很像一个**开关**，其优势如下。

（1）由于一部分元素可被隐藏，能在一个图表中装载更多的信息。

（2）让用户具有启发式学习的效果。

思维导图，是一种简单而有效的表达发散性思维的图。思维导图的制作需要用到开关，因为在表达节点之间的相互隶属关系时，具有子节点的节点可以作为开关来使用，用户需要点击开关节点才能看到子节点，再点击一次又能够将子节点隐藏。图 12-27 是一个简单的思维导图，点击“如何学习 D3”，能看到“预备知识”、“安装”、“入门”、“进阶”。点击“入门”能看到其三个子节点。

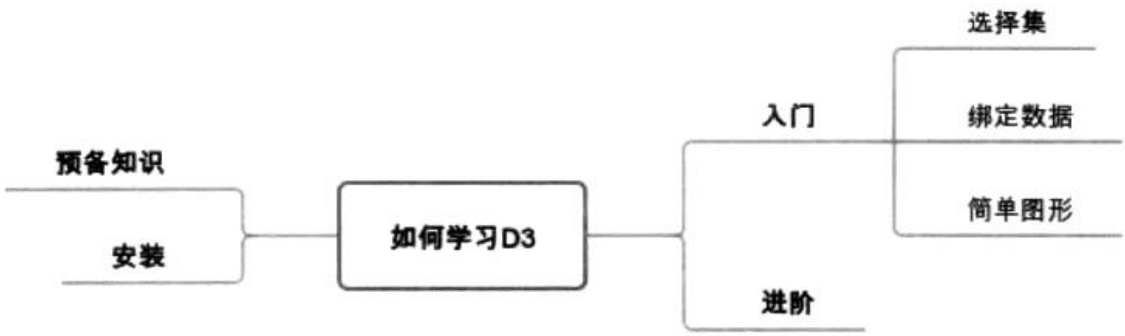


图 12-27　思维导图

12.5.1　思维导图的构造思路

思维导图的节点具有层级关系和隶属关系，很像枝叶从树干伸展开来的形状。在第 10 章讲解布局的时候，曾提到有五个布局是由层级布局扩展来的，容易发现，其中的树状图（tree layout）和集群图（cluster layout）布局制作出来的图具有“树形”，那么可以凭借这两种布局来制作思维导图。

树状图布局，将一个具有层级关系的对象 root 转换成节点数组 nodes 时，情况如下。有一个 root 对象：

```
{
    name: "node1",
    children:
    [
```

```
        { name: "node2" },
        { name: "node3" }
    ]
}
```

经树状图布局转换后，得到的节点数组 nodes 如下：

```
[
    {
        name: "node1",
        children:
        [
            { name: "node2" },
            { name: "node3" }
        ]
    },
    { name: "node2" },
    { name: "node3" }
]
```

图 12-28 为以上节点数组的示意图。由于 node1 具有子节点，可作为开关使用，点击 node1 才会展现 node2 和 node3。

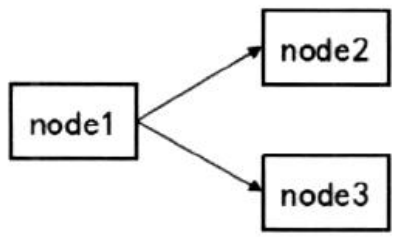


图 12-28　只有三个节点的树状图

问题是：怎样制作一个“开关”，使得点击树状图中的某个节点时，树状图更新并显示出被点击节点的子节点。

我们知道，树状图的层级关系是由每一个对象的 children 属性决定的（当然，也可以通过 tree.children()修改这一点），也就是说，如果某一个节点的 children 值为空，则再次用布局计算时，其子节点就不会进入节点数组 nodes 了。例如，将 root 改为：

```
{
    name: "node1",
    children: null
}
```

则得到的节点数组 nodes 里将没有 node2 和 node3 节点。也就是说，“开关”只要将被点击，节点的 children 设置为 null 即可。但是，由于将来可能还要用到 children 节点，可设一临时变量

_children 保存此值，例如：

```
{
    name: "node1",
    children: null
    _children:      /* 临时变量 */
    [
        { name: "node2" },
        { name: "node3" }
    ]

}
```

树状图布局不会认为_children 是保存子节点的变量，只把它看作是一般的变量而保存下来，因此节点数组 nodes 里只有一个节点。根据上面的思路，写一个开关切换函数如下：

```
//切换开关，d 为被点击的节点
function toggle(d){

    if(d.children){
        //如果有子节点
        d._children = d.children; //将该子节点保存到_children
        d.children = null;  //将子节点设置为null

    }else{
    //如果没有子节点
        d.children = d._children; //从_children取回原来的子节点
        d._children = null; //将_children设置为null
    }
}
```

每次开关状态切换时，都要重新调用布局重新计算节点的位置，也就是说，要有一个重绘函数能够处理数据发生更新的情况。这就又要用到第 4.5.3 节的处理模板，重绘函数的部分代码如下，其中尤其要注意开关函数是如何被使用的：

```
//重绘函数
function redraw(source){

    //重新计算节点和连线
    var nodes = tree.nodes(root);
    var links = tree.links(nodes);

    //获取节点的update部分
    var nodeUpdate = svg.selectAll(".node")
                        .data(nodes, function(d){ return d.name; });
```

```
    //获取节点的enter部分
    var nodeEnter = nodeUpdate.enter();

    //在给enter部分添加新的节点时，添加监听器，应用开关切换函数
    nodeEnter.append("g")
            .on("click", function(d) {
                toggle(d);
                redraw(d);
             });

/***************
        省略
***************/
}
```

每一个被新添加的节点，都会响应 click 事件。当某个节点被点击时，如果它具有子节点，则在开关切换函数的作用下，root 对象被修改了，然后调用重绘函数后，新的树状图将被绘制。如此一来，树状图具有开关功能，也就可以当作思维导图使用。

12.5.2 思维导图的制作

首先，准备一个具有层级关系的 JSON 文件，文件名为 learn.json，保存有关如何学习 D3 的方法，文件的部分内容如下：

```
{
    "name":"如何学习D3",
    "children":
    [
        {
            "name":"预备知识" ,
            "children":
            [
                {"name":"HTML & CSS" },
                {"name":"JavaScript" },
                {"name":"DOM" },
                {"name":"SVG" }
            ]
        },
         /***** 省略 *****/
    ]
}
```

其次，创建一个树状图布局和对角线生成器：

```
var tree = d3.layout.tree()
            .size([height, width]);

var diagonal = d3.svg.diagonal()
            .projection(function(d) { return [d.y, d.x]; });
```

再次，通过 d3.json 请求 learn.json 文件：

```
d3.json("learn.json",function(error,root){

    //给第一个节点添加初始坐标x0和x1
    root.x0 = height / 2;
    root.y0 = 0;

    //以第一个节点为起始节点，重绘
    redraw(root);
}
```

然后，实现最关键的重绘函数，函数声明如下：

```
function redraw(source)
```

只有一个参数 source，这是被点击的节点，如果该节点原来为闭合状态，点击后其子节点将显现，如果原来为打开状态，点击后其子节点将隐藏。函数体的实现，分为四个步骤。

1．调用布局，计算节点和连线数组

树状图布局的 tree.nodes()返回节点数组，tree.links()返回连线数组。其中，对节点的 y 坐标重新计算，使其只与节点的深度有关，由于后期绘制节点和连线时要将 x 坐标和 y 坐标对调，因此这里重计算的实际上是水平方向的坐标。代码如下：

```
//应用布局，计算节点和连线
var nodes = tree.nodes(root);
var links = tree.links(nodes);

//重新计算节点的y坐标
nodes.forEach(function(d) { d.y = d.depth * 180; });
```

之所以重新计算 y 坐标，是为了当数据更新（用于点击节点）时，保证树状图的结构不要发生太大的变化，如此看起来比较自然。

2．分别处理节点的 update、enter、exit 三部分

在 svg 里选择当前所有的节点，使其与节点数组 nodes 绑定，绑定时要设定一个键函数。键函数里直接返回 d.name，当节点数组发生更新时，新节点要与旧节点在名称上相对应。代码

如下：

```
//获取节点的update部分
var nodeUpdate = svg.selectAll(".node")
                .data(nodes, function(d){ return d.name; });

//获取节点的enter部分
var nodeEnter = nodeUpdate.enter();

//获取节点的exit部分
var nodeExit = nodeUpdate.exit();
```

先处理 enter 部分，即添加节点。节点的构成为：分组元素里有一个圆表示节点，还有一个文字元素表示节点的名称。元素结构如下：

```
<g>
   <circle></circle>
   <text></text>
</g>
```

本例中，每一个新添加的节点都将缓慢地过渡到自己本身的位置，如此更具有友好性。因此，新节点的初始位置都设定在 source 节点处，确切地说是重回之前 source 节点的位置，该坐标是保存在 source.x0 和 source.y0 里的。另外，对于每一个新节点，设置<circle>的半径为 0，设置<text>为完全透明，接下来在处理 update 部分的时候会将这些新节点过渡到正常状态。图 12-29 展示了处理 enter 部分和 update 部分时节点的位置是如何确定和过渡的。

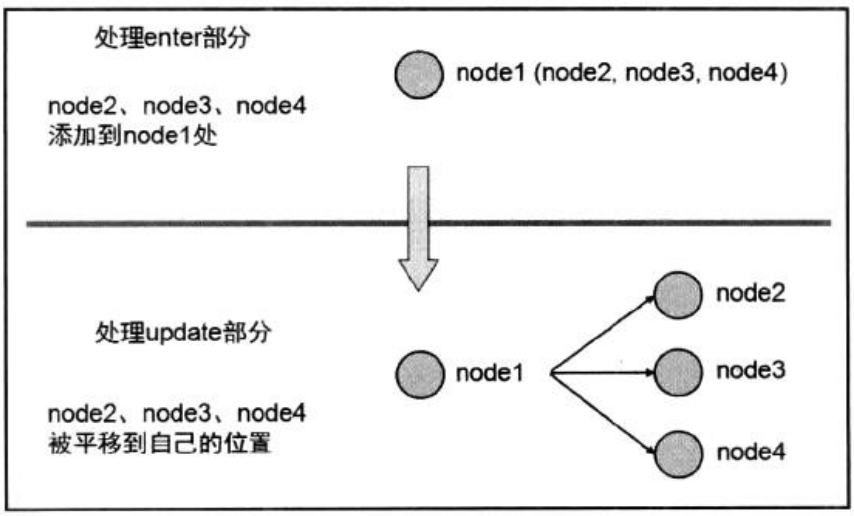


图 12-29　处理 enter 部分和 update 部分时，关于节点位置的过渡方法

处理 enter 部分的代码如下：

```
//1. 节点的enter部分的处理办法
```

```
var enterNodes = nodeEnter.append("g")
      .attr("class","node")
      .attr("transform", function(d) {
          return "translate(" + source.y0 + "," + source.x0 + ")";
       })
      .on("click", function(d) {
          toggle(d);
          redraw(d);
       });

enterNodes.append("circle")
      .attr("r", 0)
      .style("fill", function(d) {
          return d._children ? "lightsteelblue" : "#fff";
       });

enterNodes.append("text")
       .attr("x", function(d) {
          return d.children || d._children ? -14 : 14;
       })
       .attr("dy", ".35em")
       .attr("text-anchor", function(d) {
          return d.children || d._children ? "end" : "start";
       })
       .text(function(d) { return d.name; })
       .style("fill-opacity", 0);
```

然后处理 update 部分，将所有节点（包括在 enter 部分新添加的节点）都缓缓过渡到新的位置。由于新的节点数组是与节点选择集绑定在一起的，因此 d.x 和 d.y 里保存的就是新的坐标值。此外，还要将<circle>的半径过渡到正常半径，<text>过渡到完全不透明。代码如下：

```
//2. 节点的 update 部分的处理办法
var updateNodes = nodeUpdate.transition()
            .duration(500)
            .attr("transform", function(d) {
               return "translate(" + d.y + "," + d.x + ")";
            });

updateNodes.select("circle")
       .attr("r", 8)
```

```
        .style("fill", function(d) {
            return d._children ? "lightsteelblue" : "#fff";
        });

updateNodes.select("text")
        .style("fill-opacity", 1);
```

最后处理 exit 部分，需要删除的节点的位置缓缓过渡到其父节点处，另外，为使其达到缓缓消失的效果，将节点的圆元素<circle>的半径过渡到 0，文字元素<text>过渡成完全透明，最终使用 remove 删除节点。代码如下：

```
//3. 节点的exit部分的处理办法
var exitNodes = nodeExit.transition()
        .duration(500)
        .attr("transform", function(d) {
            return "translate(" + source.y + "," + source.x + ")";
        })
        .remove();

exitNodes.select("circle")
        .attr("r", 0);

exitNodes.select("text")
        .style("fill-opacity", 0);
```

3. 分别处理连线的 update、enter、exit 三部分

在 svg 中选择所有的连线，绑定连线数组 links，由此可获得连线的 update、enter、exit 部分。

```
//获取连线的update部分
var linkUpdate = svg.selectAll(".link")
        .data(links, function(d){ return d.target.name; });

//获取连线的enter部分
var linkEnter = linkUpdate.enter();

//获取连线的exit部分
var linkExit = linkUpdate.exit();
```

对于连线的 enter 部分，是插入路径元素<path>，路径由对角线生成器获取，对角线的起点和终点坐标都是(source.x0, source.y0)。

对于连线的 update 部分，将所有的连线的位置（对角线的起点和终点）更新到新的位置，

即目前绑定的数组 links 里保存的位置。

对于连线的 exit 部分，令其缓缓过渡到当前的 source 点，再移除。代码如下：

```
//1. 连线的enter部分的处理办法
linkEnter.insert("path",".node")
        .attr("class", "link")
        .attr("d", function(d) {
           var o = {x: source.x0, y: source.y0};
           return diagonal({source: o, target: o});
        })
        .transition()
        .duration(500)
        .attr("d", diagonal);

//2. 连线的update部分的处理办法
linkUpdate.transition()
      .duration(500)
      .attr("d", diagonal);

//3. 连线的exit部分的处理办法
linkExit.transition()
        .duration(500)
        .attr("d", function(d) {
          var o = {x: source.x, y: source.y};
          return diagonal({source: o, target: o});
        })
        .remove();
```

4. 保存当前的节点坐标

当用户点击节点后，数据发生更新，即每个节点的坐标要发生更新。但是，在对节点和连线进行过渡操作的时候，需要使用到更新前的数据（source.x0 和 source.y0）。因此，每一次调用重绘函数，都要将当前节点的位置保存下来：

```
nodes.forEach(function(d) {
     d.x0 = d.x;
     d.y0 = d.y;
});
```

x 坐标和 y 坐标分别保存在 x0 和 y0 中，在调用 redraw(source)时，被点击的节点作为参数传到了重绘函数里，因此 source.x0 和 source.y0 里保存的是被点击之前节点的坐标。

结果如图 12-30 和图 12-31 所示，后者是前者再次展开部分节点之后的结果。

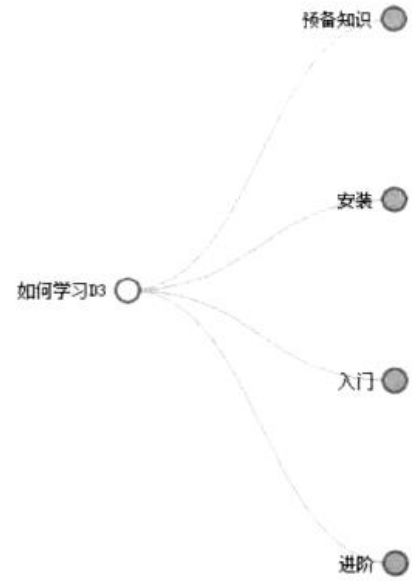


图 12-30　思维导图，只展开最初的四个节点

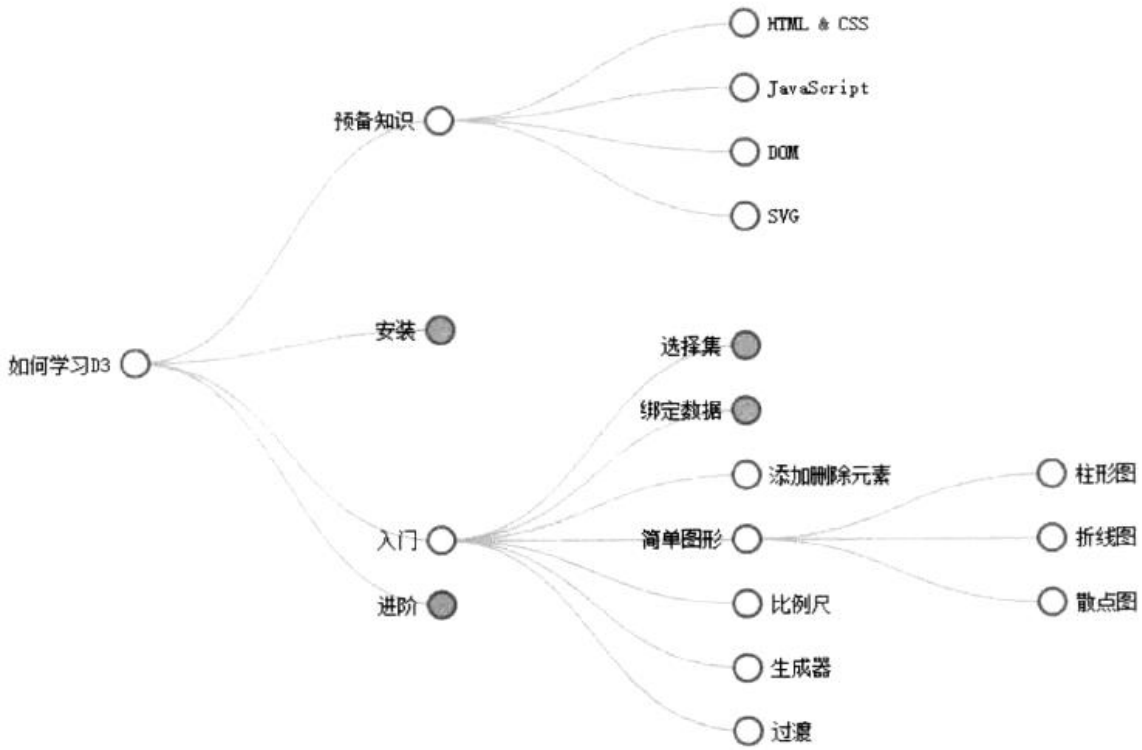


图 12-31　思维导图，展开“预备知识”和“入门”的部分节点后

第 13 章

地图进阶

本章内容包括：

- 值域的颜色
- 标注
- 标线
- 拖曳和缩放
- 力导向地图

地图的可视化，是网络上常见的，而且相比其他图表比较令人心动。本章讲述关于地图的进阶应用。

第 1 节，讲述将一段值域映射到颜色，并应用到地图上。例如，中国各省的 GDP。

第 2 节，介绍如何制作标注，标注是地图上只需一个坐标的元素。

第 3 节，介绍如何制作标线，标线是地图上需要两个坐标以上的元素。

第 4 节，学习如何对地图拖曳和缩放。

第 5 节，制作一个力导向的中国地图，并对其预备知识 Voronoi 图和 Delaunay 三角剖分进行讲解。

13.1 值域的颜色

第 11 章的中国地图，各省份的颜色值都是随意的。如果要将一些值反映在地图上，可以利

用颜色的变化来表示值的变化。例如，有值域的范围为：

```
[10, 500]
```

现希望 10 用浅绿表示，500 用深绿表示，10～500 之间的值用浅绿和深绿之间的颜色表示。显然，此处需要一个函数，传入的参数是 10～500 之间的值，返回值是浅绿到深绿之间的颜色值。使用 D3 的颜色插值函数可以实现这一效果，先用 d3.rgb 创建浅绿和深绿两个 rgb 颜色对象，然后以它们作为 d3.interpolate 的参数。

```
var palegreen = d3.rgb(66,251,75);      //浅绿
var darkgreen = d3.rgb(2,100,7);        //深绿

var color = d3.interpolate(a,b);        //颜色插值函数
```

最后一行的 color 可作为函数使用，参数的范围为[0, 1]。当参数为 0 时，返回浅绿；当参数为 1 时，返回深绿。但是，我们需要的值域是[10, 500]，而不是[0, 1]。因此，要定义一个线性比例尺，将[10, 500]按线性关系映射到[0, 1]。代码如下：

```
var linear = d3.scale.linear()
               .domain([10, 500])
               .range([0, 1]);
```

如此一来，便可结合比例尺来使用颜色插值函数。

```
color( linear(10) );        //返回浅绿RGB(66,251,75)
color( linear(250) );       //返回浅绿和深绿之间的值
color( linear(500) );       //返回深绿RGB(2,100,7)
```

在地图上给各区域赋颜色值的时候，可使用上述手法将值域转换成颜色值。

下面制作一个例子，要求如下。

有一组中国各省份旅游业产值的数据（虚构），将其在中国地图上进行可视化，产值低的省份用浅色填充，产值高的用深色填充。

首先，虚构一组数据，内容是中国各省份旅游业的产值，文件名为 tourism.json。部分内容如下所示，其中，数组 provinces 的每一项是一个省份的值，name 是省份名称，value 是该省份旅游业的产值。代码如下：

```
{
    "name": "中国",
    "provinces":
    [
        {"name": "北京", "value":   14149  },
        {"name": "天津", "value":   2226.41 },
        {"name": "河北", "value":   1544.94 },
        {"name": "山西", "value":   3720.24 },
```

```
        {"name": "内蒙古", "value": 2771.96 }
        /***** 省略部分数据  *****/
    ]
}
```

然后，通过 d3.json 请求 tourism.json 文件，并用线性比例尺结合颜色插值的方法求取各省份的颜色值。代码如下：

```
d3.json("tourism.json", function(error, valuedata){
//将读取到的数据存到数组values，令其索引号为各省的名称
    var values = [];
    for(var i=0; i<valuedata.provinces.length; i++){
        var name = valuedata.provinces[i].name;
        var value = valuedata.provinces[i].value;
        values[name] = value;
    }

    //求最大值和最小值
    var maxvalue = d3.max(valuedata.provinces, function(d){
            return d.value;
});
    var minvalue = 0;

    //定义一个线性比例尺，将最小值和最大值之间的值映射到[0, 1]
    var linear = d3.scale.linear()
                        .domain([minvalue, maxvalue])
                        .range([0, 1]);

    //定义最小值和最大值对应的颜色
    var a = d3.rgb(0,255,255);          //浅蓝色
    var b = d3.rgb(0,0,255);            //蓝色

    //颜色插值函数
    var computeColor = d3.interpolate(a,b);

    //设定各省份的填充色
    provinces.style("fill", function(d,i){
        var t = linear( values[d.properties.name] );
        var color = computeColor(t);
        return color.toString();
    });
});
```

这段代码的步骤为：

（1）将文件中的数据整理到一个数组 values 中，以省份的名称作为索引号。例如，**values["河北"] = 1544.94**。

（2）求旅游业产值的最大值和最小值，并以此作为比例尺定义域的最大值和最小值。

（3）定义颜色比例尺。

（4）各省份填充颜色。

颜色插值函数以浅蓝色和深蓝色作为边界，也就是说，某省份的旅游业产值越大，用越深的蓝色来表示，结果如图 13-1 所示。

但是，图 13-1 缺少一个用于告诉用户什么颜色对应什么数值的标志。常用的方法有两种：一种是添加多个矩形，每个矩形用一种颜色填充，并在矩形旁边添加对应的数值；另一种是添加一个线性渐变的矩形，并在旁边添加对应的数值。前一种方法比较简单，后一种方法需要用到渐变填充。本例采用第二种方法。

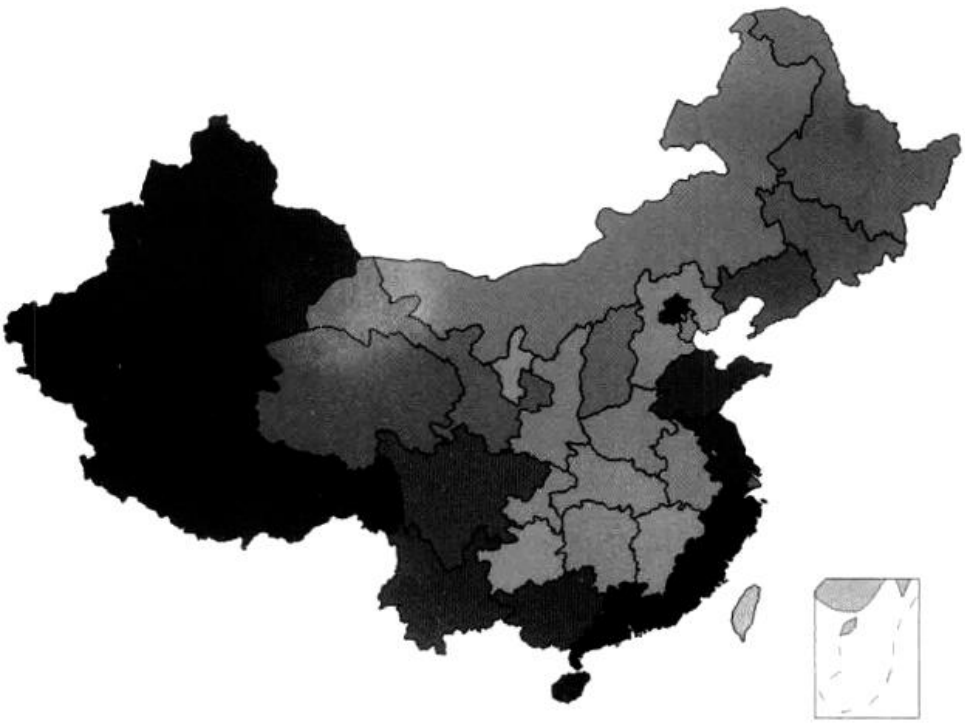


图 13-1　颜色越深，表示旅游业的产值越大，没有值的地区用灰色填充

SVG 中的线性渐变<linearGradient>是定义在<defs>标签中的。给渐变定义一个 id 号，在图形元素上指定此 id 号即可使用（第 2.5.7 节）。参照此元素结构，用 D3 代码定义一个线性渐变如下：

```
var defs = svg.append("defs");

var linearGradient = defs.append("linearGradient")
                         .attr("id","linearColor")
                         .attr("x1","0%")
                         .attr("y1","0%")
                         .attr("x2","100%")
```

```
                                .attr("y2","0%");

var stop1 = linearGradient.append("stop")
                          .attr("offset","0%")
                          .style("stop-color",a.toString());

var stop2 = linearGradient.append("stop")
                          .attr("offset","100%")
                          .style("stop-color",b.toString());
```

最后，在适当的位置添加一个矩形，并设定 fill 属性的值为 url(#linearColor)，其中 linearColor 为线性渐变的 id 号。代码如下：

```
var colorRect = svg.append("rect")
            .attr("x", 20)
            .attr("y", 490)
            .attr("width", 140)
            .attr("height", 30)
            .style("fill","url(#" + linearGradient.attr("id") + ")");
```

在渐变矩形上方添加文字后，最终结果如图 13-2 所示。

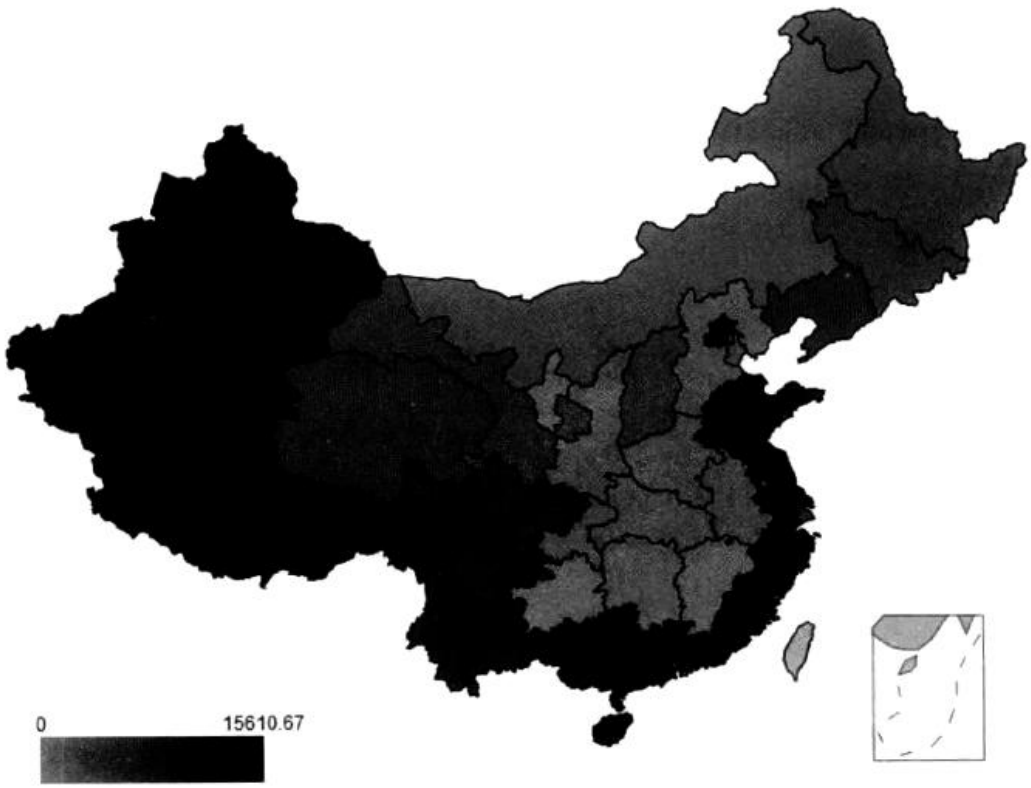


图 13-2　在左下角添加了渐变填充的矩形

13.2 标注

标注，是指地图上只需要一个坐标即可表示的元素。例如，在经纬度(116, 39)处画一个圆，在(108, 30)处画一个符号，这些都属于标注，也可以将标注理解为“点元素”。

我们只知道经纬度是不能直接在地图上作图的，需要先用投影函数将其转换成像素坐标。例如，如果要在中国地图上标出“北京”的位置，但是不知道北京的像素坐标。北京的经纬度可通过查询得知是(116.3, 39.9)，将此值作为投影函数的参数即可得到像素坐标。其实，GeoJSON文件的地理信息也是经纬度，也是经过投影函数转换后得到了像素坐标。因此，如果使用同一个投影函数，那么转换后的北京坐标即可直接在地图上绘制。

首先，定义一个投影函数如下：

```
var projection = d3.geo.mercator()
                       .center([107, 31])
                       .scale(600)
                       .translate([width/2, height/2]);
```

其次，使用此投影定义地理路径生成器，用于绘制地图：

```
var path = d3.geo.path()
              .projection(projection);
```

然后，以北京的经纬度作为投影的参数，得到北京的像素坐标：

```
var peking = [116.3, 39.9];
var proPeking = projection(peking);
```

最后，用上面得到的像素坐标绘制一个圆，该圆就正好位于北京的位置：

```
svg.append("circle")
   .attr("class","point")
   .attr("cx",proPeking[0])
   .attr("cy",proPeking[1])
   .attr("r",8);
```

凭借以上思路，可以在地图上标注很多有意思的信息。

13.2.1 标注地点

标注一种常见的用法是指出某地点的位置。下面制作一个例子，要求如下。

在中国地图上标注出五个城市的位置，并在各标注处添加一张该城市的图片，五个城市分别是北京、上海、桂林、乌鲁木齐、拉萨。

首先，搜集五个城市的经纬度和图片，在网上可查到。将图片保存在网页 HTML 文件同一目录下的 img 文件夹里，然后写一个 JSON 文件，将经纬度信息和图片路径信息汇集起来。JSON 文件内容如下：

```
{
    "name": "地点",
    "location":
    [
        {
            "name":"北京",
            "log":"116.3",
            "lat":"39.9",
            "img":"img/beijing.png"
        },
        {
            "name":"上海",
            "log":"121.4",
            "lat":"31.2",
            "img":"img/shanghai.png"
        },
        /***** 省略 *****/
    ]
}
```

图片的数据不存在 JSON 文件里，仅仅保存路径即可。绘制完地图后，调用 d3.json 请求 places.json 文件，并通过绑定数组 location 添加足够数量的分组元素<g>，每个分组代表一个城市。利用分组元素<g>的 transform 属性可将标注点平移到指定位置，平移量可通过投影函数计算城市的经纬度得到。

然后，向<g>里分别添加圆形<circle>和图片<image>。<image>是 SVG 的图片元素，只需要五个属性就够了：

```
<image xlink:href="image.png" x="200" y="200" width="100" height="100">
</image>
```

其中，

- **xlink:href**：图片名称或图片网址。
- **x**：图片左上角 x 坐标。

- **y**：图片左上角 y 坐标。
- **width**：图片宽度。
- **height**：图片高度。

请求文件及插入标注点的代码如下：

```
d3.json("places.json", function(error, places ) {

    //插入分组元素
    var location = svg.selectAll(".location")
                .data(places.location)
                .enter()
                .append("g")
                .attr("class","location")
                .attr("transform",function(d){
                    //计算标注点的位置
                    var coor = projection([d.log, d.lat]);
                    return "translate("+ coor[0] + "," + coor[1] +")";
                });

    //插入一个圆
    location.append("circle")
            .attr("r",7);

    //插入一张图片
    location.append("image")
            .attr("x",20)
            .attr("y",-40)
            .attr("width",90)
            .attr("height",90)
            .attr("xlink:href",function(d){
                return d.img;
            });
    });
```

结果如图 13-3 所示。

13.2.2 夜光图

利用标注可制作夜光图，现有要求如下。

夜晚各城市都会有灯光，越发达的城市肯定越亮，将中国所有城市的灯光使用圆标注在地图上，越亮的越接近黄色，越不亮的越接近白色。

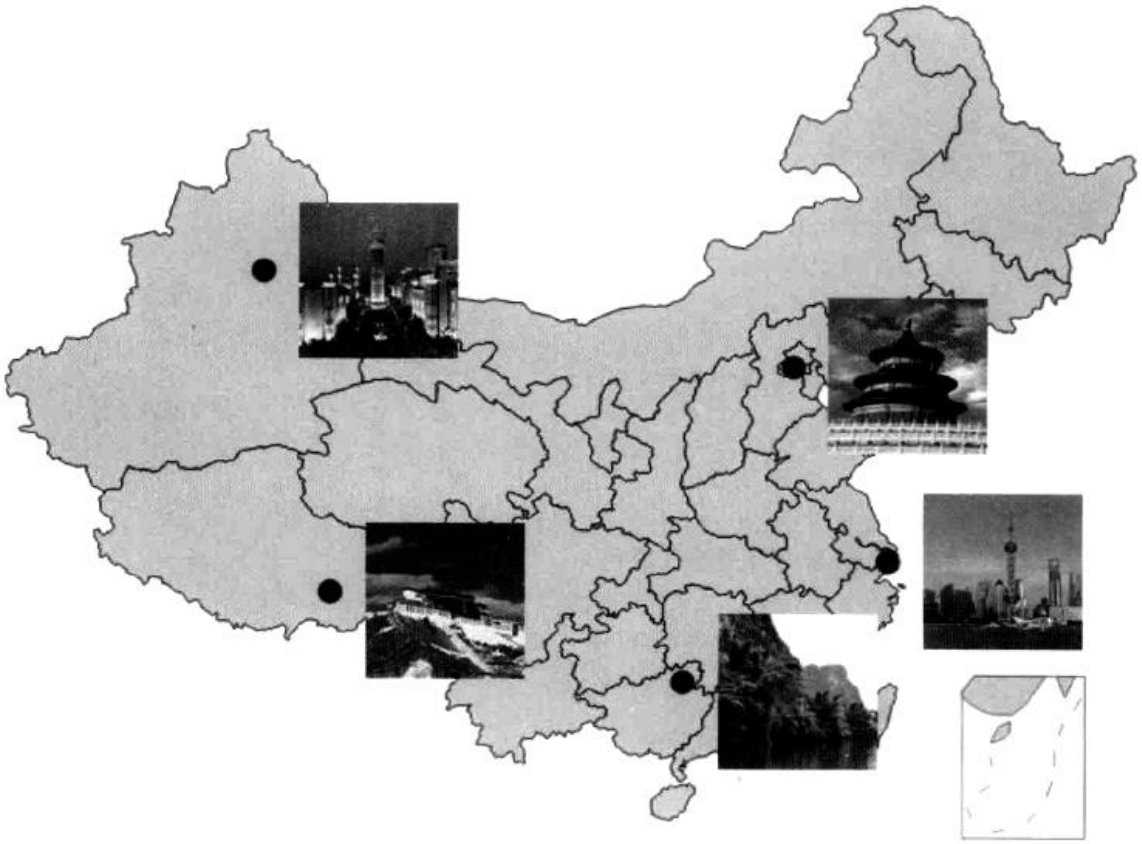


图 13-3　标注地点

首先，要准备数据：**中国所有城市的经纬度和夜晚的灯光数据**。但是，所有城市的经纬度容易找到，灯光数据不易找到。因此，本例使用随机数生成虚构的灯光数据。

Math.random()返回随机数，范围为[0, 1]，约定：当随机数为 0 时，颜色为白色；随机数为 1 时，颜色为黄色。这里要定义一个颜色插值函数，以白色和黄色为边界，Math.random()返回的数值传给插值函数得到的值，就作为某标注点的颜色。

请求文件，并添加所有城市的标注点的代码如下所示：

```
d3.json("cities.json", function(error, chinadata){

    /**************************************************
        整理chinadata中的经纬度数据，保存在数组cities里
    **************************************************/

    //定义最小值和最大值对应的颜色
    var a = d3.rgb(255,255,255);          //白色
    var b = d3.rgb(255,255,0);            //黄色

    //颜色插值函数
```

```
    var computeColor = d3.interpolate(a,b);

    var points = svg.selectAll("circle")
            .data(cities)          //绑定数组
            .enter()
            .append("circle")
            .attr("class","point")
            .attr("cx",function(d){
                return projection([d.log, d.lat])[0];  //设定x坐标
            })
            .attr("cy",function(d){
                return projection([d.log, d.lat])[1];  //设定y坐标
            })
            .attr("r",2.5)         //标注点半径
            .style("fill",function(d){
                //计算灯光强度（颜色）
                var color = computeColor(Math.random());
                return color.toString();
            })
            //设定过滤器
            .style("filter","url(#"+ gaussian.attr("id") +")");
    });
```

对于数组 cities 的每一项，变量 log 为经度，lat 为纬度，因此 projection([d.log, d.lat])可返回某城市的像素坐标。另外，为使得标注出来的圆更具有“灯光”的效果，为圆添加过滤器，在第 2.5.6 节提到了模糊滤镜的概念，SVG 中的元素结构为：

```
<defs>
<filter id="GaussianBlur">
   <feGaussianBlur in="SourceGraphic" stdDeviation="2" />
</filter>
</defs>
```

参照此结构，用 D3 在 SVG 中添加相应的元素构成滤镜，代码如下：

```
var defs = svg.append("defs");

var gaussian = defs.append("filter")
               .attr("id","gaussian");

gaussian.append("feGaussianBlur")
   .attr("in","SourceGraphic")
   .attr("stdDeviation","1");
```

如此，则可对某元素设定该模糊滤镜，即：

```
.style("filter","url(#"+ gaussian.attr("id") +")");
```

结果如图 13-4 所示。要注意，本例中由于数据是虚构的，不能代表实际的灯光效果。此外，实际应用时灯光的颜色也要详加考虑，本例只提供一个示范。

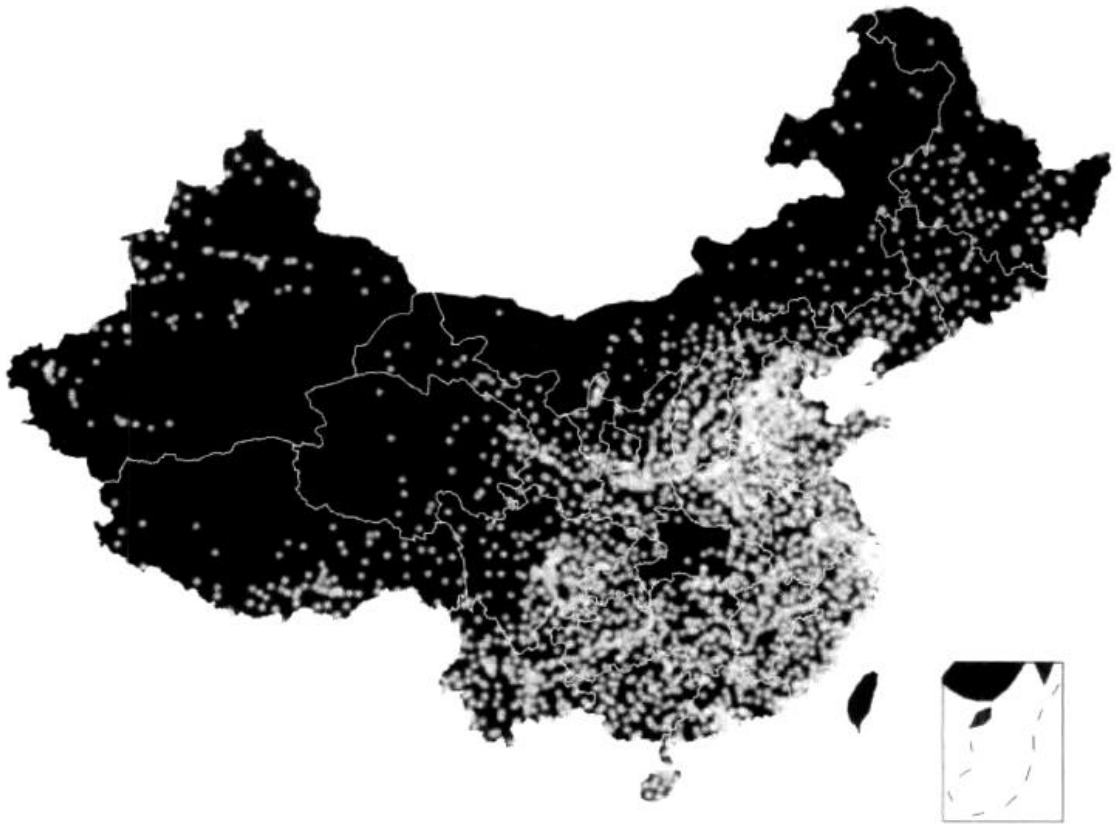


图 13-4　城市夜光图

13.3　标线

标线，是指地图上需要两个坐标以上才能表示的元素。例如，北京和上海之间连线。用于绘制标线的图形元素有两种：线段元素<line>和路径元素<path>。如果是在平面地图上，且不要求两点之间有曲线，<line>已足够；如果是在球面地图上，或对于平面地图上的曲线，则需使用<path>。标线有时带有箭头，表示方向。

13.3.1　带有箭头的标线

如果需要表示标线的方向，则可以在末端加箭头。第 2.5.5 节，提到了给 SVG 定义标记，

从而为<line>或<path>添加箭头的方法。箭头的标记如下：

```
<defs>
<marker id="arrow"
        markerUnits="strokeWidth"
        markerWidth="12"
        markerHeight="12"
        viewBox="0 0 12 12"
        refX="6"
        refY="6"
        orient="auto">
<path d="M2,2 L10,6 L2,10 L6,6 L2,2"
        style="fill: #000000;" />
</marker>
</defs>
```

标记是定义在<defs>中的。其中，<marker>是标记的主体，<marker>中的<path>是标记的图形，此处是箭头的路径，也可用其他图形，如圆形、矩形等。参照此结构，使用 D3 的代码添加一个箭头标记的代码如下：

```
var defs = svg.append("defs");

var arrowMarker = defs.append("marker")
                      .attr("id","arrow")
                      .attr("markerUnits","strokeWidth")
                      .attr("markerWidth","12")
                      .attr("markerHeight","12")
                      .attr("viewBox","0 0 12 12")
                      .attr("refX","6")
                      .attr("refY","6")
                      .attr("orient","auto");

var arrow_path = "M2,2 L10,6 L2,10 L6,6 L2,2";

arrowMarker.append("path")
        .attr("d",arrow_path)
        .attr("fill","#000");
```

对于需要添加箭头的线段，设定其 marker-end 属性为 url(#arrow)即可添加箭头，arrow 是箭头标记的 id 号。

下面在平面的中国地图上添加一个带箭头的标线，表示“从桂林到北京”的路径。对于平面地图上两点之间连线，用<line>元素即可。

根据两座城市的经纬度分别计算其像素坐标，并添加一个<line>，设置属性 marker-end 的

值为 url(#arrow)。代码如下：

```
var peking = projection([116.3, 39.9]);
var guilin = projection([110.3, 25.3]);

svg.append("line")
    .attr("class","route")
    .attr("x1",guilin[0])
    .attr("y1",guilin[1])
    .attr("x2",peking[0])
    .attr("y2",peking[1])
    .attr("marker-end","url(#arrow)");
```

如此一来，标线的末尾会带一个箭头，结果如图 13-5 所示。

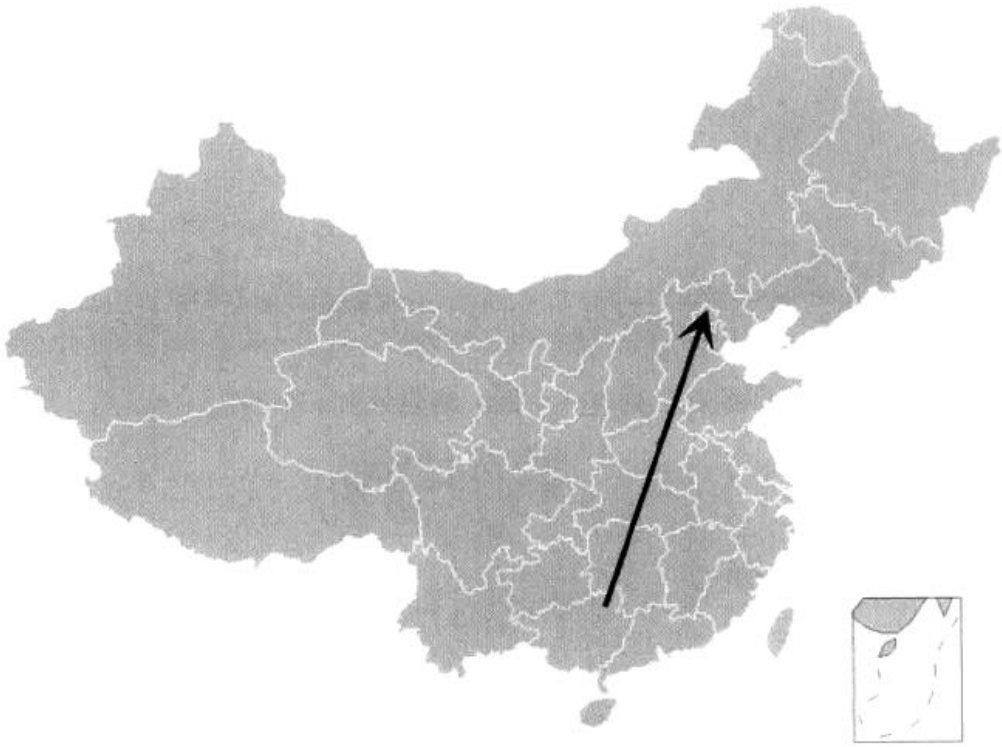


图 13-5　带箭头的标线

上面的箭头是添加到线段终点的。此外，可定义一个新的标记，添加到线段的起点。例如，起点显示一个圆。

定义一个新的标记，代码如下：

```
var startMarker = defs.append("marker")
                      .attr("id","startPoint")
                      .attr("markerUnits","strokeWidth")
```

```
                        .attr("markerWidth","12")
                        .attr("markerHeight","12")
                        .attr("viewBox","0 0 12 12")
                        .attr("refX","6")
                        .attr("refY","6")
                        .attr("orient","auto");

startMarker.append("circle")
           .attr("cx",6)
           .attr("cy",6)
           .attr("r",2)
           .attr("fill","#000");
```

此标记的 id 号是 startPoint，用其为线段的 marker-start 赋值即可。将添加线段元素的代码修改为：

```
svg.append("line")
   .attr("class","route")
   .attr("x1",guilin[0])
   .attr("y1",guilin[1])
   .attr("x2",peking[0])
   .attr("y2",peking[1])
   .attr("marker-end","url(#arrow)")        //终点处添加箭头
   .attr("marker-start","url(#startPoint)"); //起点处添加圆
```

结果如图 13-6 所示，标线的起点处有一个圆，终点处有一个箭头。

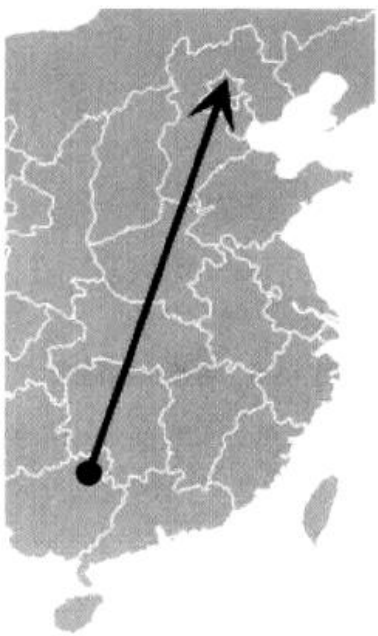


图 13-6　起点处有圆，终点处有箭头

13.3.2　球状地图的标线

对于平面地图，可以用投影求出两点的像素坐标，再用<line>连接。如果是球面地图，则需要用<path>绘制一条连接球面两点的路径。例如，球面地图上，从北京到华盛顿的标线，不可能用<line>绘制。问题是，<path>的路径值怎么计算呢？

绘制地图的时候，使用了地理路径生成器 d3.geo.path，将保存于 GeoJSON 文件里的数据变成了很多路径元素<path>。因此，只要使用同一个地理路径生成器，即可计算出球面两点之间连线的路径。

首先，创建一个 GeoJSON 格式的对象，描述北京到华盛顿的连线。类型为 LineString，即直线。坐标数组 coordinates 里只有两项，分别是北京和华盛顿的经纬度，如下：

```
var pekingToWashington = {
    type: "LineString",
    coordinates: [[116.4, 39.9], [-77.0, 38.9]]
};
```

然后，将此 GeoJSON 对象作为参数传给地理路径生成器，即可得到两点之间的路径值。添加一个<path>元素并设定路径值，起点、终点处分别设置原点和箭头，代码如下：

```
svg.append("path")
    .attr("class","route")
    .attr("d",path(pekingToWashington))          //计算路径值
    .attr("marker-end","url(#arrow)")
    .attr("marker-start","url(#startPoint)");
```

结果如图 13-7 所示。如果要更改标线在地图上的行走方向，只要增加 pekingToWashington 的坐标值即可。例如更改为：

```
var pekingToWashington = {
    type: "LineString",
    coordinates: [[116.4, 39.9], [146.4, 39.7], [176.4, 39.5],
                  [-150.4, 39.3],[-110.4, 39.1],[-77.0, 38.9]]
};
```

则结果如图 13-8 所示，北京到华盛顿的路线变成了从太平洋穿过。

图 13-7　球面地图上，北京到华盛顿的标线

图 13-8　从北京穿过太平洋到达华盛顿

13.4　拖动和缩放

使用百度地图或谷歌地图的时候，用户可以通过鼠标拖动地图以观看其他地方，也可以滑动滚轮以放大或缩小地图，此功能也可能会出现在地图的可视化里。在 D3 中，可通过修改投影函数来实现。绘制地图时有如下代码：

```
var projection = d3.geo.mercator()
                   .center([0, 0])
                   .scale(260)
                   .translate([width/2, height/2]);

var path = d3.geo.path()
              .projection(projection);
```

投影里有 scale()，用于控制缩放量；有 translate()，用于控制平移量；有 rotate()，用于控制旋转量。因此，通过修改投影，然后重绘，便可拖动和缩放地图。

13.4.1　平面地图

绘制一份平面的世界地图，包含两部分：经纬线网格和地理边界。

首先，绘制经纬度网格，代码如下：

```
var graticule = d3.geo.graticule()
                   .extent([[-180, -90],[180+eps, 90]])
```

```
                    .step([10,10]);

var grid = graticule();

var gridPath = svg.append("path")
            .datum(grid)
            .attr("class","graticule")
            .attr("d",path);
```

网格元素的选择集保存在变量 gridPath 里，当事件发生时，需要更新网格，那时需要用到此选择集。

其次，读取 GeoJSON 文件，绘制各国地理边界：

```
var groups = svg.append("g");

var countries = groups.selectAll("path")
                    .data( root.features )
                    .enter()
                    .append("path")
                    .attr("class","country")
                    .style("fill", function(d,i){
                        return color(i);
                    })
                    .attr("d",path);
```

各国的路径元素选择集保存在 countries 里，后面更新时同样要用到。

再次，从投影对象中获取初始的平移量和缩放量。之后如果事件触发，更新平移量和缩放量时，需要加到初始值上。

```
var initTran = projection.translate();
var initScale = projection.scale();
```

然后，定义一个 zoom 行为。当 zoom 事件触发时，该事件的具体信息被写入 d3.event。此时，有两个属性：d3.event.scale 和 d3.event.translate，前者可作为投影函数 scale 的参数，后者可作为 translate 的参数。代码如下所示：

```
var zoom = d3.behavior.zoom()
    .scaleExtent([1, 10])
    .on("zoom", function(d){
        //更新投影函数的平移量
        projection.translate([
            tTran[0] + d3.event.translate[0],
            initTran[1] + d3.event.translate[1] ]
        ;

        //更新投影函数的缩放量
```

```
        projection.scale( initScale * d3.event.scale );

        //重绘地图
        countries.attr("d",path);

        //重绘经纬度网格
        gridPath.attr("d",path);
    });
```

最后，在SVG中添加一个透明的矩形，专门用来捕捉事件：

```
svg.append("rect")
    .attr("class","overlay")
    .attr("x", 0)
    .attr("y", 0)
    .attr("width",width)
    .attr("height",height)
    .call(zoom);
```

其class设置为：

```
.overlay {
    fill: none;
    pointer-events: all;
}
```

结果如图13-9所示，这是地图的初始状态。按住鼠标键不放，拖曳地图后，如图13-10所示。使用滚轮放大地图某部分后，如图13-11所示。

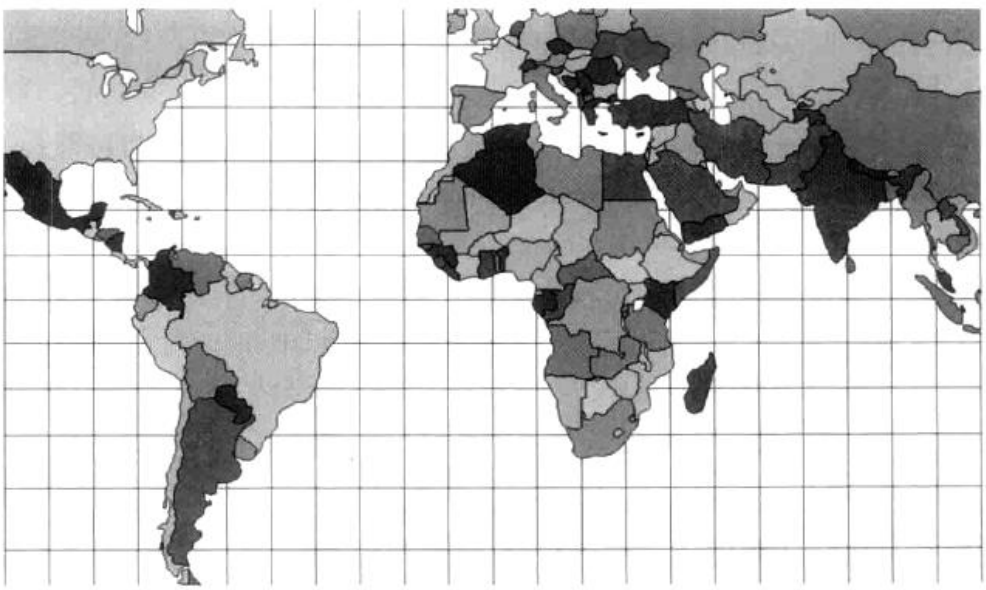


图13-9　世界地图

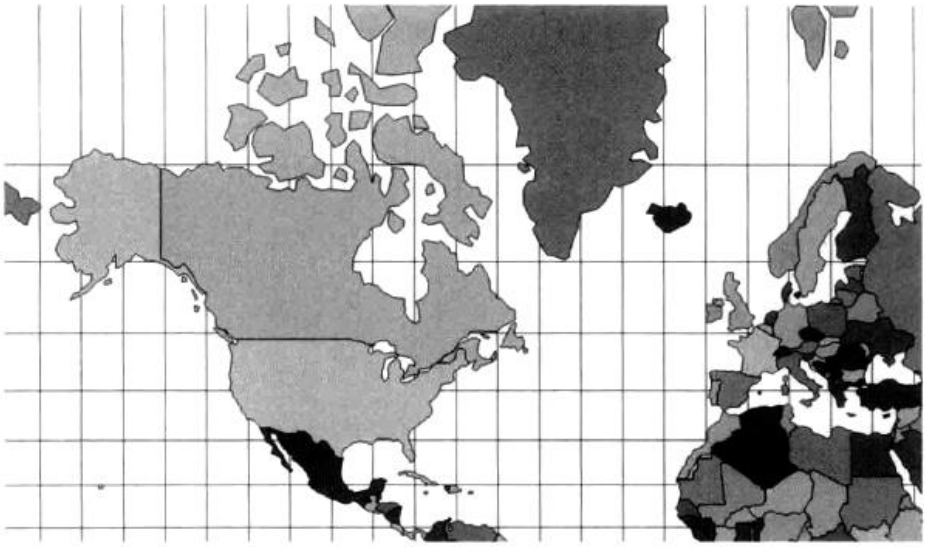


图 13-10　拖曳之后，平移到别处

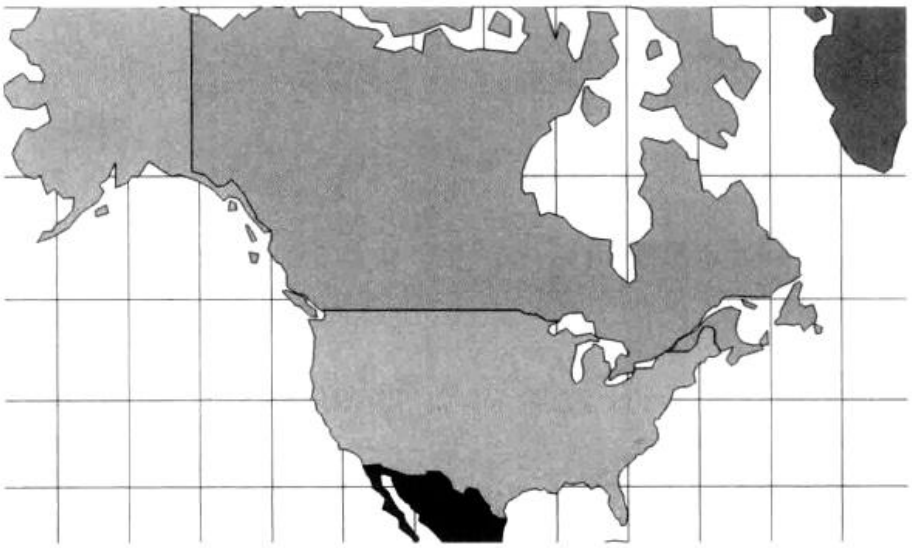


图 13-11　放大美国和加拿大部分的地图

13.4.2　球面地图

球面地图的拖动表现为旋转地球，例如有正射投影 d3.geo.orthographic，其定义形如：

```
var ortho = d3.geo.orthographic()
              .scale(180)
              .translate([width/2, height/2])
              .rotate([60,0,0])
              .clipAngle(90);
```

其中，rotate()是设定球体旋转角度的。当 zoom 事件触发时，可通过更改 rotate()的参数达到旋转球体的目的。缩放球体与平面地图一样，更改 scale()即可。参考上一节的内容，球面地图的 zoom 行为定义如下：

```
//投影初始平移量和缩放量
var initRotate = ortho.rotate();
var initScale = ortho.scale();

var zoom = d3.behavior.zoom()
           .scaleExtent([1, 10])
           .on("zoom", function(d){

           //更改投影的旋转角度
           ortho.rotate([
               initRotate[0] + 180 * d3.event.translate[0] / width ,
               initRotate[1] - 180 * d3.event.translate[1] / height ,
               initRotate[2]
           ]);

           //更新投影的缩放量
           ortho.scale( initScale * d3.event.scale );

           //重绘地图
           countries.attr("d",path);

           //重绘经纬度网格
           gridPath.attr("d",path);
       });
```

结果如图 13-12 所示。

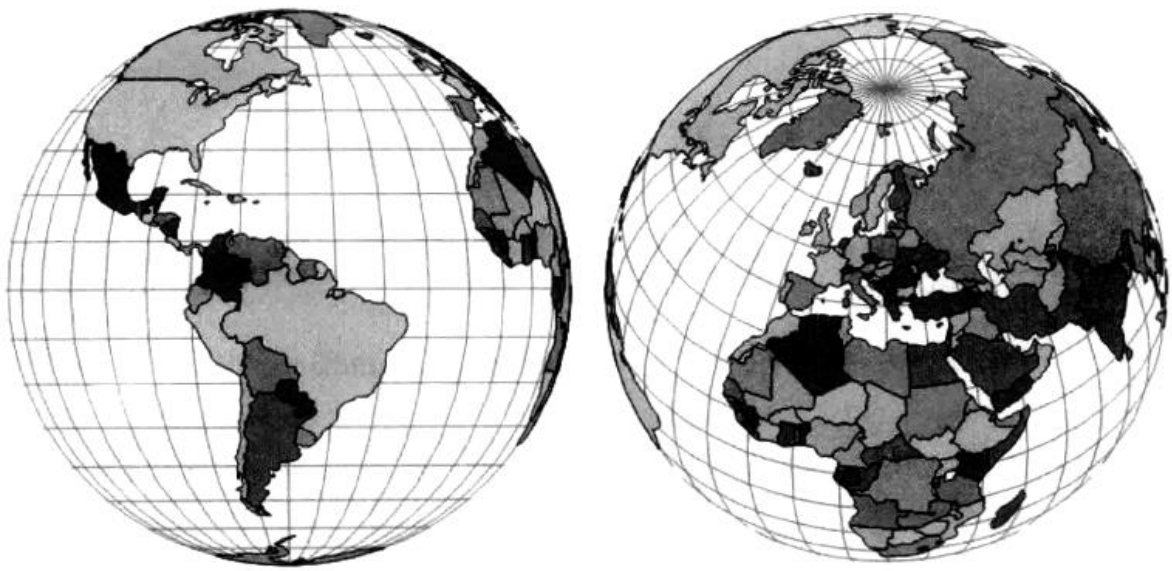


图 13-12　初始球面（左）和旋转后球面（右）

13.5　力导向地图

力导向地图是指结合力导向图和地图，使得地图的各区域（省、市、州等）作为力导向图的各顶点，并按照力导向图的运动方式改变位置，且用户可以拖曳地图的各区域。

第 10.3 节讲解了如何制作力导向图，首先需要的数据是**节点和连线**。节点容易知道，就是地图的各区域，例如陕西省、江西省、湖南省等。但是连线是不知道的，即哪个省与哪个省相连不知道。D3 提供了 Voronoi 图和 Delaunay 三角剖分的相关方法，能够根据节点坐标计算连线。

13.5.1　Voronoi 图和 Delaunay 三角剖分

Voronoi 图，由连接两邻点直线的垂直平分线组成的连续多边形组成。将 Voronoi 图中共享一条边的点连接起来得到一系列三角形，就是 Delaunay 三角剖分。本书不深入探讨其数学本质，只关注其应用方法，因此读者只需要知道：

空间中有 *N* 个点，将其进行 Delaunay 三角剖分后，各顶点将以三角形的形式连接起来。

D3 有 d3.geom.voronoi 用于计算 Voronoi 图，其相关函数简介如下。

- d3.geom.**voronoi**()

创建一个 Voronoi 图的运算器。

- **voronoi**(data)

返回一个多边形数组。

- voronoi.**x**([x])

设定或获取 *x* 方向坐标的访问器。

- voronoi.**y**([y])

设定或获取 *y* 方向坐标的访问器。

- voronoi.**clipExtent**([extent])

设定作用范围，extent 的形式为[[x0, y0], [x1, y1]]，其中 x0 和 x1 分别代表左右边界、y0 和 y1 分别代表上下边界。

- voronoi.**triangles**(data)

返回一个三角形数组，每一个三角形都是 Delaunay 三角形。

- voronoi.**links**(data)

返回一个连线数组，所有连线组成的就是 Delaunay 三角剖分的结果。

下面制作一个 Voronoi 图。首先，在 SVG 区域中随机添加 100 个节点，所有节点都保存在

数组 nodes 里，每个节点都是一个对象，具有两个属性 x 和 y，坐标值随机生成。代码如下：

```
//添加节点
var nodes = [];

for (var i = 0; i < 100; i++) {
    nodes.push({
        x: Math.random() * width,
        y: Math.random() * height
    });
};
```

某次测试中，节点的随机分布状态如图 13-13 所示。

然后，创建一个 Voronoi 图的**运算器**（在 D3 的 API 中，不叫运算器，而叫布局（layout），但为了与第 10 章的布局区分开来，这里称为运算器），代码如下：

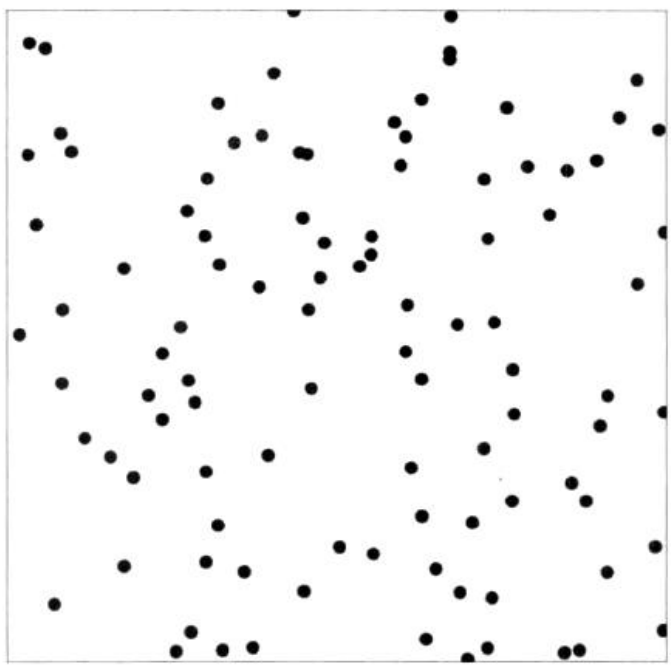


图 13-13　节点的随机分布状态

```
//创建voronoi图的运算器
var voronoi = d3.geom.voronoi()
                .x(function(d){
                    return d.x;
                })
                .y(function(d){
                    return d.y;
```

```
        });
```

分别设置了 x 坐标和 y 坐标的访问器，即传入对象的 x 属性代表 x 坐标，y 属性代表 y 坐标。据此即可计算多边形数组：

```
var polygons = voronoi(nodes);
```

polygons 是一个数组，在控制台输出后的结果如图 13-14 所示。可以看到，该数组的每一项代表一个多边形，共 100 个。每一个多边形也是一个数组，由于各多边形的顶点数不同，各子数组的长度不同（第 0 项多边形有 5 个顶点，第 1 项有 4 个）。

图 13-14　多边形数组

最后，绘制这些多边形。为了方便，可以将多边形的边从数组 polygons 里提取出来，存到一个数组 edges 里，代码如下：

```
//根据多边形数组获取多边形的边
var edges = [];

polygons.forEach(function(d,i){
    for(var i=0; i<d.length; i++){
        if( i !== d.length - 1 )
            edges.push( edge( d[i] , d[i+1] ) );
        else
            edges.push( edge( d[i] , d[0] ) );
    }
});

function edge(a, b) {
    return {
        source: a,
```

```
        target: b
    };
}
```

有了节点数组 nodes 和连线数组 edges，将其绘制出来就很简单了，结果如图 13-15 所示。

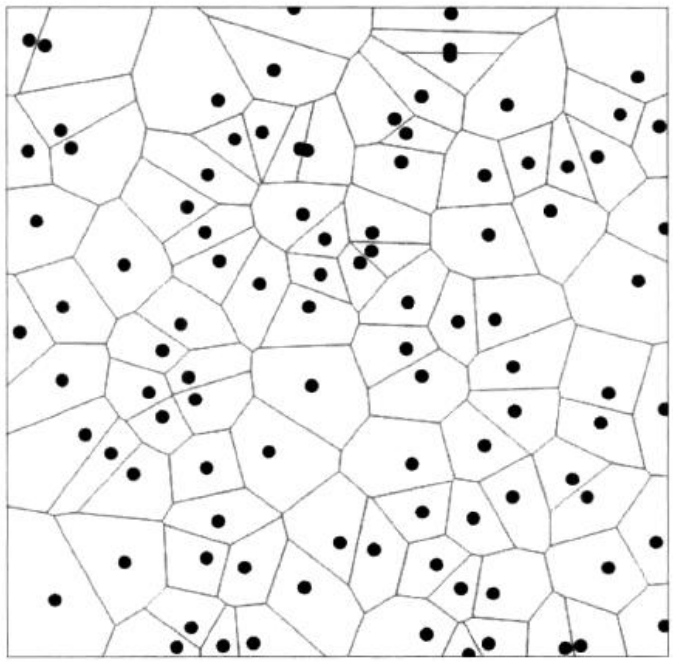


图 13-15　Voronoi 图

接下来再制作一个 Delaunay 三角剖分的图，大部分步骤与 Voronoi 图是一样的。上面提到，将 Voronoi 图相邻多边形共享边的相关点连接起来后，就可得到三角剖分的图。voronoi.trangles() 返回的就是三角剖分得到的三角形数组。因此，在添加节点后，计算连线时，将代码改为：

```
var triangles = voronoi.triangles(nodes);

triangles.forEach(function(d,i){
    links.push( edge( d[0] , d[1] ) );
    links.push( edge( d[1] , d[2] ) );
    links.push( edge( d[2] , d[0] ) );
});
```

其中，triangles 是三角形数组，与多边形数组类似，只不过每一项只有三个节点。得到三角形数组后，再将数组里的所有连线提取出来，保存在数组 links，以便绘制。此外，还有一个更简单的方法，可用于计算三角剖分的所有连线，代码如下：

```
links = voronoi.links(nodes);
```

voronoi.links 可直接得到三角剖分的连线数组，结果如图 13-16 所示。

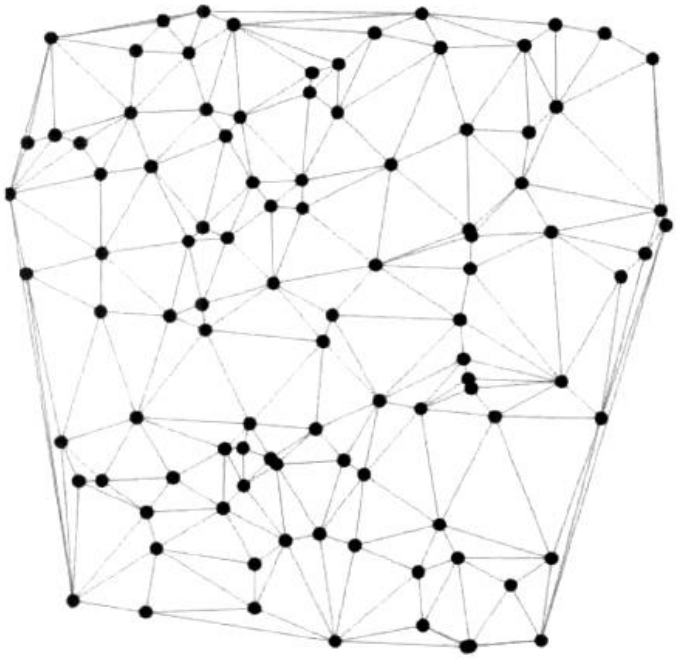


图 13-16　三角剖分图

13.5.2　力导向的中国地图

以中国地图为例，要求如下。

制作一个力导向的地图，支持鼠标拖曳。中国的省份作为节点，连线用三角剖分生成。

1. 定义布局、投影和地理路径生成器

创建一个力导向布局，一个墨卡托投影和一个地理路径生成器，代码如下：

```
var force = d3.layout.force()
                .size([width, height]);

var projection = d3.geo.mercator()
                    .center([107, 31])
                    .scale(850)
                    .translate([width/2, height/2]);

var path = d3.geo.path()
                .projection(projection);
```

2. 请求 GeoJSON 文件

文件名为 china.geojson，保存了中国各省的地理信息。通过 d3.json 读入，代码如下：

```
d3.json("china.geojson", function(error, root) {

    if (error)
        return console.error(error);
    console.log(root.features);
});
```

地理数据保存在 root 里。

3. 计算节点数组和连线数组

先计算节点数组 nodes：

```
var nodes = [];

root.features.forEach(function(d, i) {
    //计算省份的中心坐标
    var centroid = path.centroid(d);

    //定义两个变量x和y，保存中心坐标
    centroid.x = centroid[0];
    centroid.y = centroid[1];

    //将地理特征保存在对象里
    centroid.feature = d;

    //添加到节点数组中
    nodes.push(centroid);
});
```

也就是说，每一个节点（省份）的属性如下：

```
{
    x: 中心的x坐标,
    y: 中心的y坐标,
    feature: 地理信息
}
```

其中，中心坐标是用于供三角剖分以及力导向布局使用的，feature 是绘制时用于计算省份路径的。有了节点数组，连线数组 links 可用如下方式得到：

```
var voronoi = d3.geom.voronoi()
                .x(function(d){
                    return d.x;
                })
                .y(function(d){
                    return d.y;
```

```
                    });

var links = voronoi.links(nodes);
```

这样的用法在上一节已经提到过了。

4. 设定力导向布局的属性

设置力导向布局的属性如下，最重要的两节点之间的距离，为了使得在力导向作用时，地图的各节点能保持适当的位置，可根据各省份中心位置动态求取两节点的距离。

```
force.gravity(0)
    .charge(0)
    .linkDistance(function(d){
        var dx = d.source.x - d.target.x;
        var dy = d.source.y - d.target.y;
        return Math.sqrt( dx*dx + dy*dy );
    })
    .nodes(nodes)
    .links(links)
    .start();
```

数组 nodes 和 links 也分别传给力导向布局。

5. 绘制节点和连线

分别绘制节点和连线，其中节点的元素结构为：

```
<g>
    <path></path>
</g>
```

连线直接使用<line>元素：

```
var nodeGroups = svg.selectAll("g")
    .data(nodes)    //绑定节点数组
    .enter().append("g")
    .attr("transform", function(d) {
        return "translate(" + -d.x + "," + -d.y + ")";
    })
    .call(force.drag)        //调用力导向图的拖曳行为
    .append("path")
    .attr("transform", function(d) {
        return "translate(" + d.x + "," + d.y + ")";
    })
    .attr("stroke","#000")
    .attr("stroke-width",1)
```

```
    .attr("fill", function(d,i){
        return color(i);
    })
    .attr("d", function(d){
        return path(d.feature); //使用地理路径生成器计算路径
    });

var lines = svg.selectAll("line")
    .data(links)    //绑定连线数组
    .enter()
    .append("line")
    .attr("class","link")
    .attr("x1",function(d) { return d.source.x; } )
    .attr("y1",function(d) { return d.source.y; } )
    .attr("x2",function(d) { return d.target.x; } )
    .attr("y2",function(d) { return d.target.y; } );
```

上面代码的粗体字部分是两个完全相反的平移，这里之所以“移过去，再移回来”，是为力导向地图更新 d.x 和 d.y 的时候做准备。此时元素的属性为：

```
<g transform="translate(-d.x, -d.y)">
    <path transform="translate(d.x, d.y)"></path>
</g>
```

当 d.x 和 d.y 更新时，只需要更新节点<path>的 transform 即可，<g>的 transform 不更改。也就是说，当力导向图作用，各省份运动时，计算所得的最终平移量为**相对于初始位置的平移量**，假设 x0 和 y0 为初始中心坐标，x1 和 y1 为新的中心坐标，则：

```
<g transform="translate(-d.x0, -d.y0)">
    <path transform="translate(d.x1, d.y1)"></path>
</g>
```

6. 运动更新

当力导向发生作用时，每一帧的操作为：

```
force.on("tick", function() {
    //更新连线
    lines.attr("x1", function(d) { return d.source.x; })
        .attr("y1", function(d) { return d.source.y; })
        .attr("x2", function(d) { return d.target.x; })
        .attr("y2", function(d) { return d.target.y; });

    //更新节点
    nodeGroups.attr("transform", function(d) {
```

```
        return "translate(" + d.x + "," + d.y + ")";
    });
});
```

结果如图 13-17 所示。各相邻省份之间有线相连，这是以各省份的中心点为节点数组进行三角剖分的结果。去掉省份的地形，单纯的三角剖分的结果如图 13-18 所示。

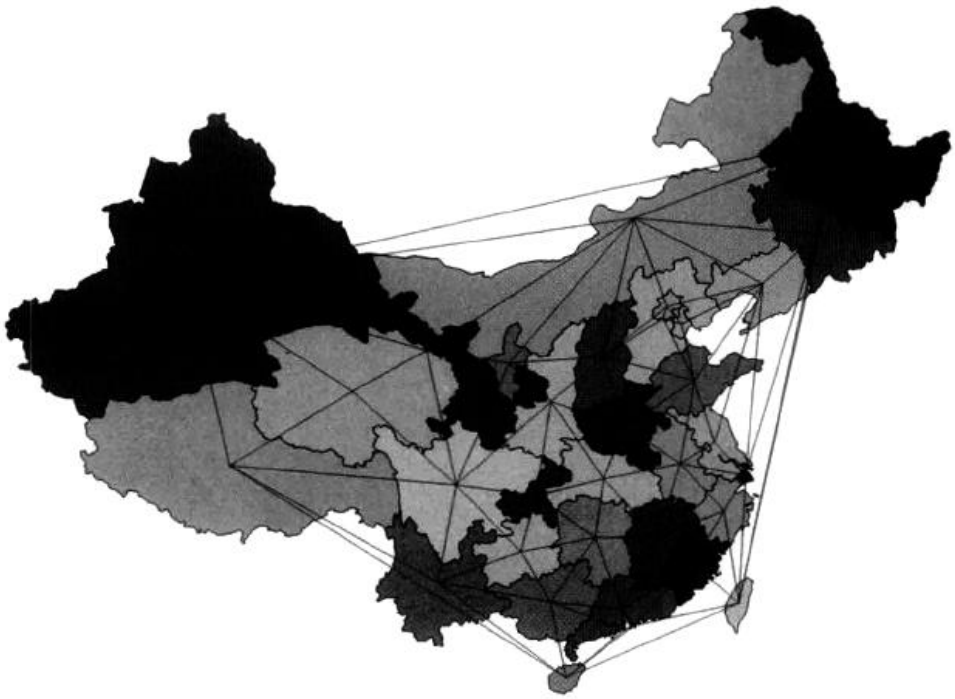


图 13-17　地图的初始状态，各省份之间的连线由三角剖分计算得到

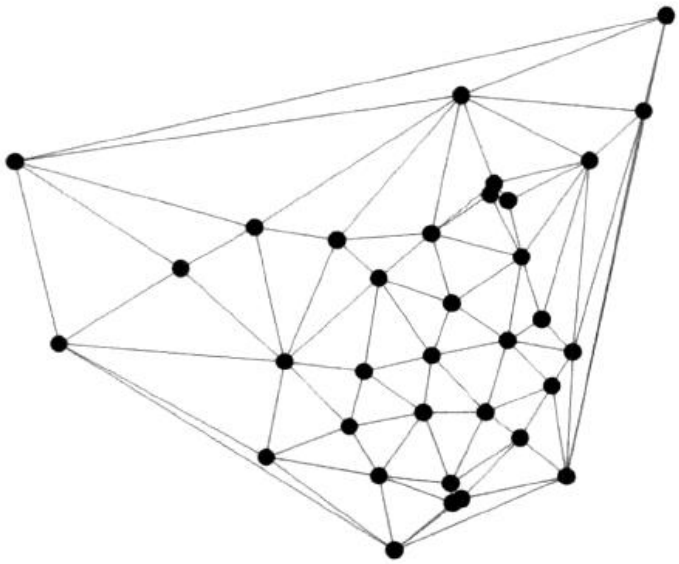


图 13-18　各省份中心点的三角剖分图

所有省份都可用鼠标拖动，拖动后各节点的位置按力导向的关系发生变化，如图 13-19 所示。

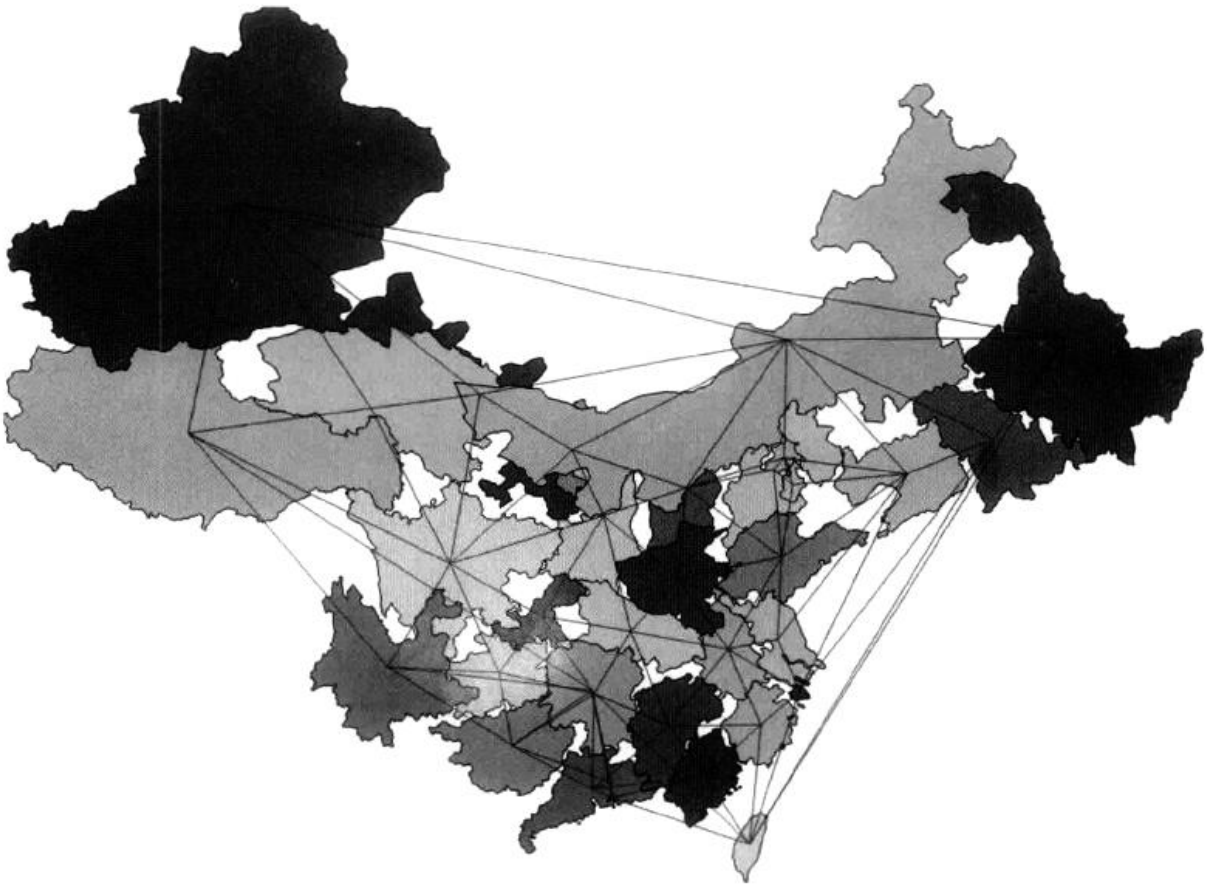


图 13-19　力导向地图

附录A

彩色插图

第2章

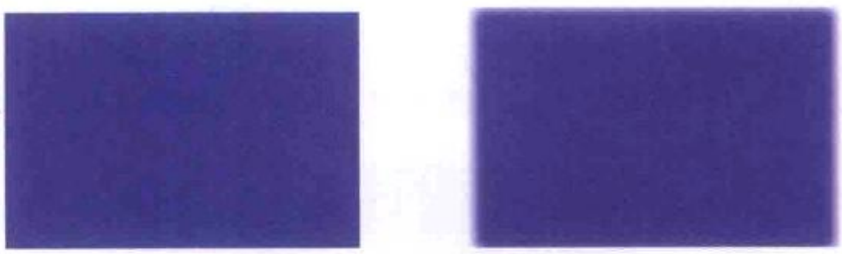


图 2-19　使用滤镜前（左）和使用滤镜后（右）的矩形

图 2-20　水平线性渐变

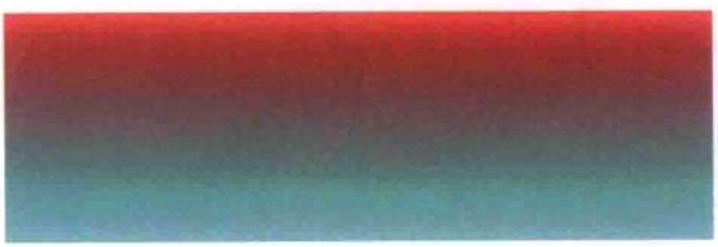


图 2-21　垂直线性渐变

第 10 章

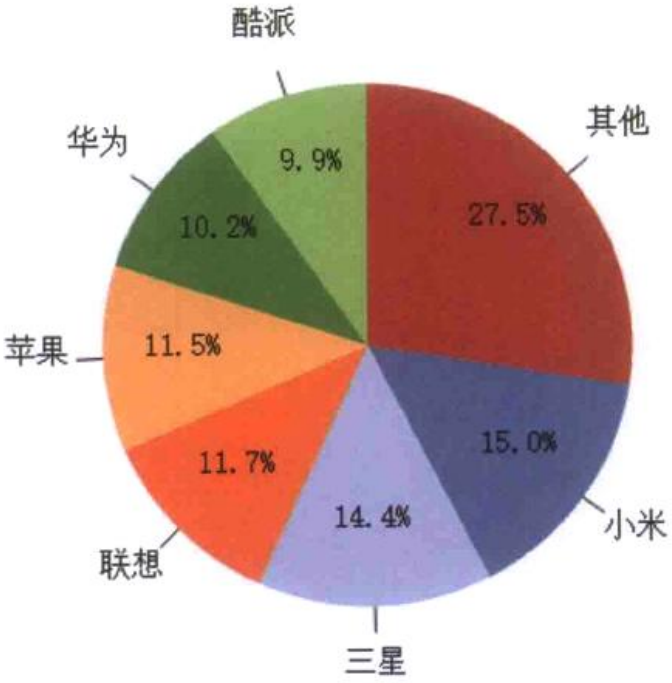


图 10-5　弧外的文字

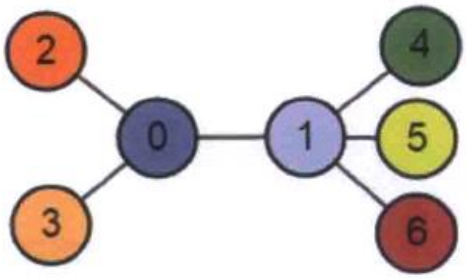


图 10-11　所有的节点都被拖曳过，5 号节点正在被拖曳

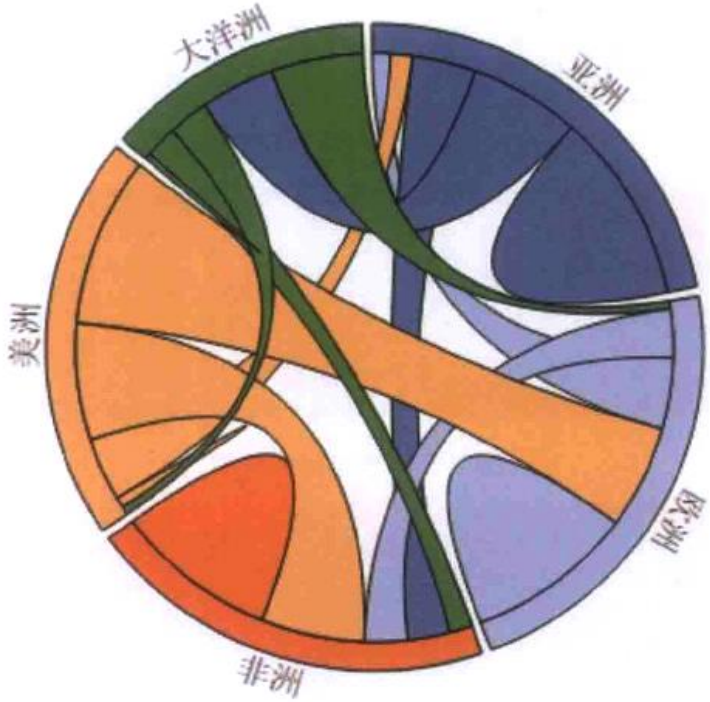


图 10-18　添加弦

图 10-23　树状图结果

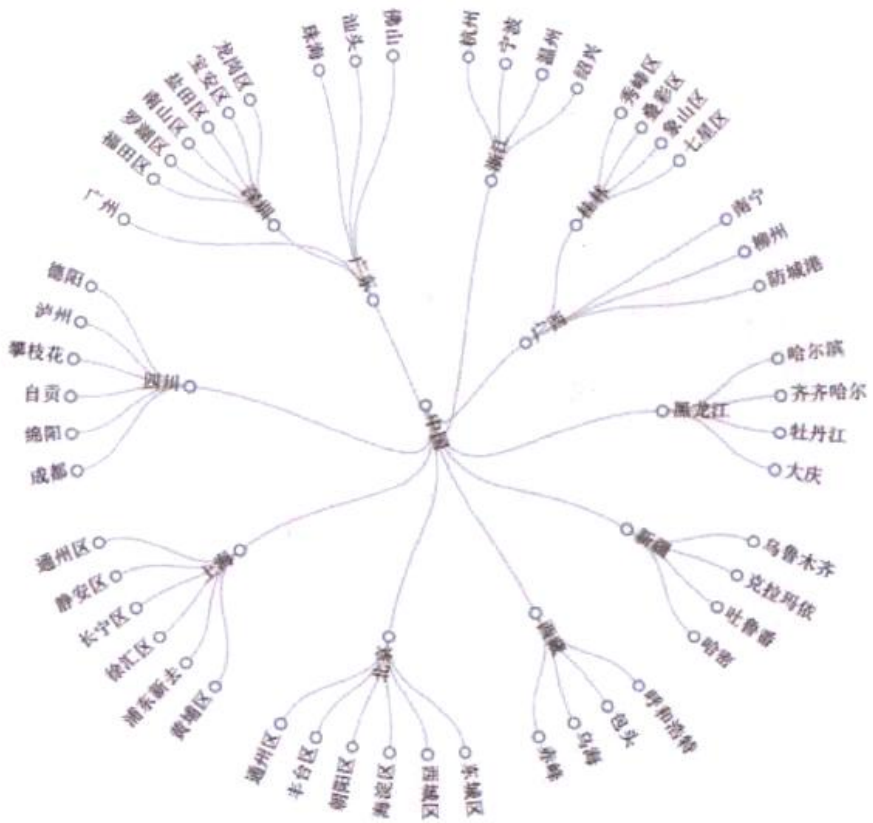


图 10-28　圆形集群图

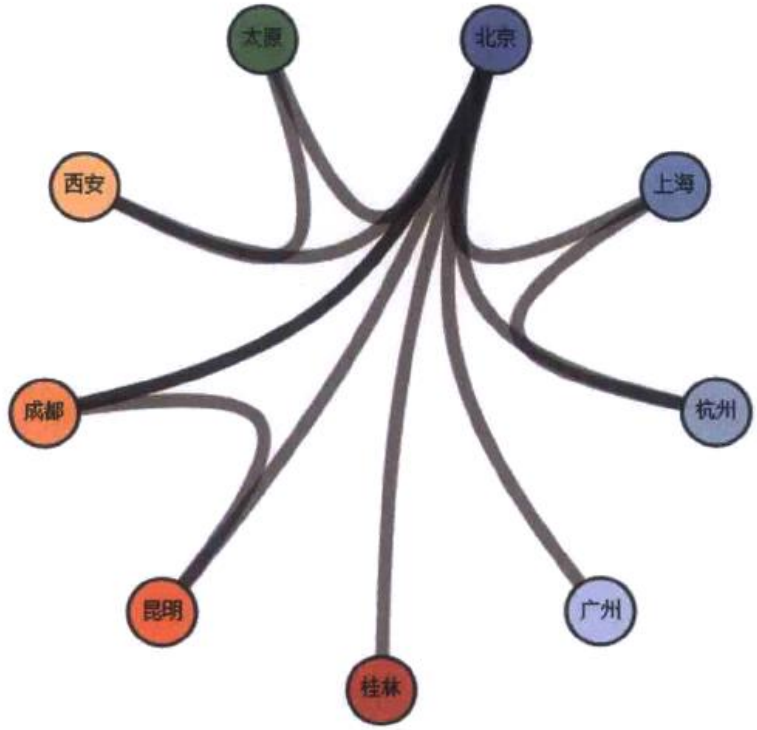


图 10-33　捆图的最终结果

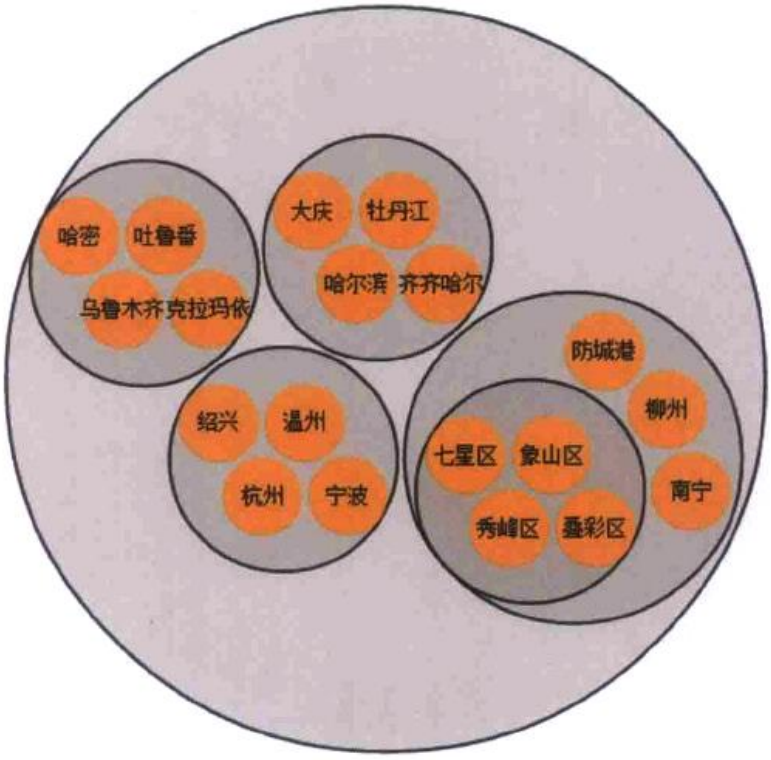


图 10-36　打包图的最终结果

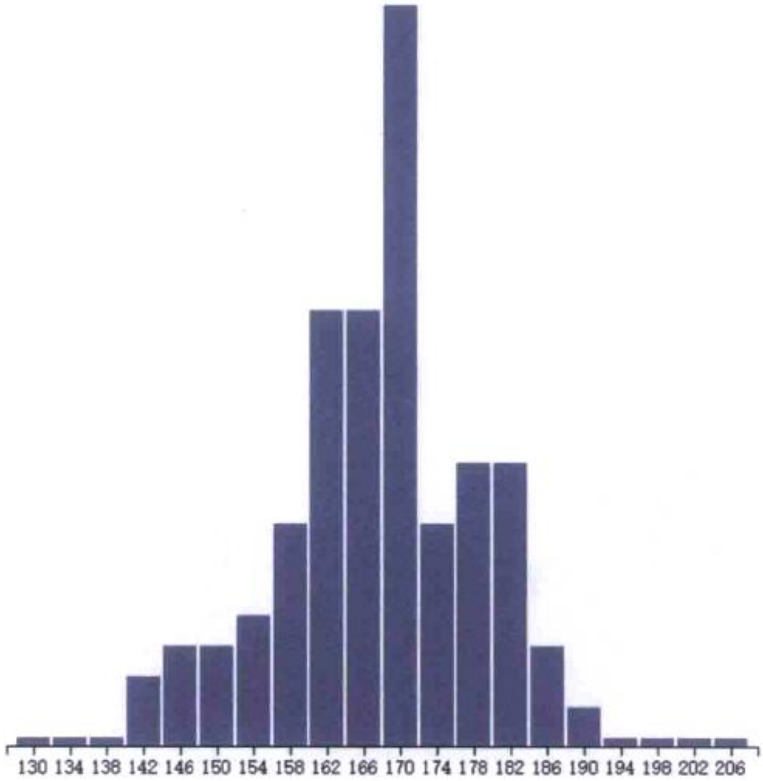


图 10-40　矩形直方图

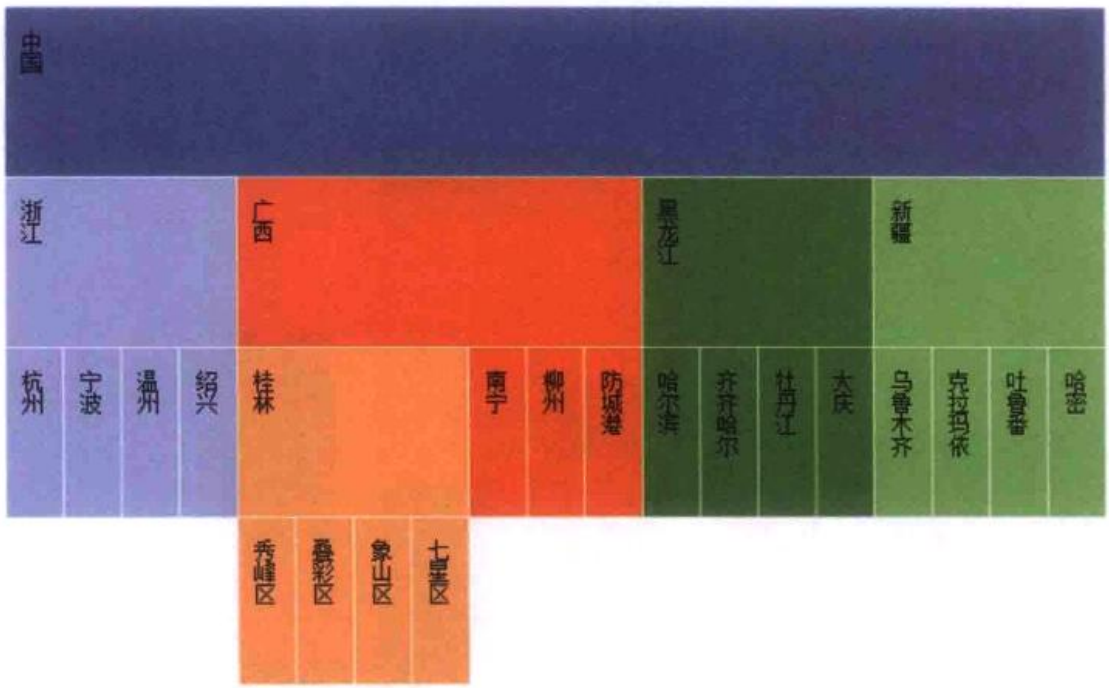


图 10-43　矩形分区图

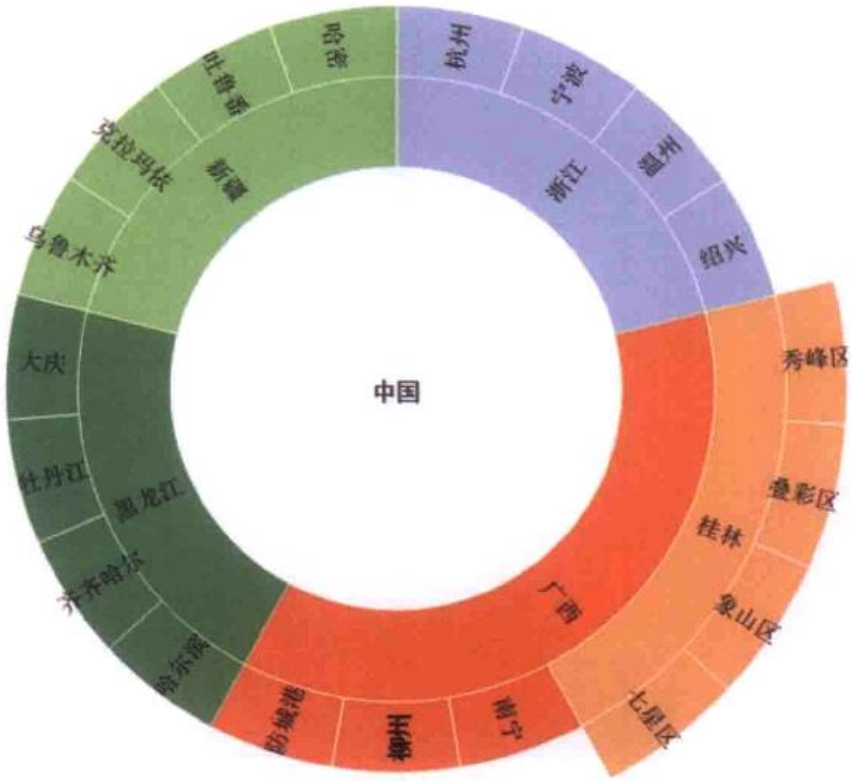


图 10-44　圆形分区图

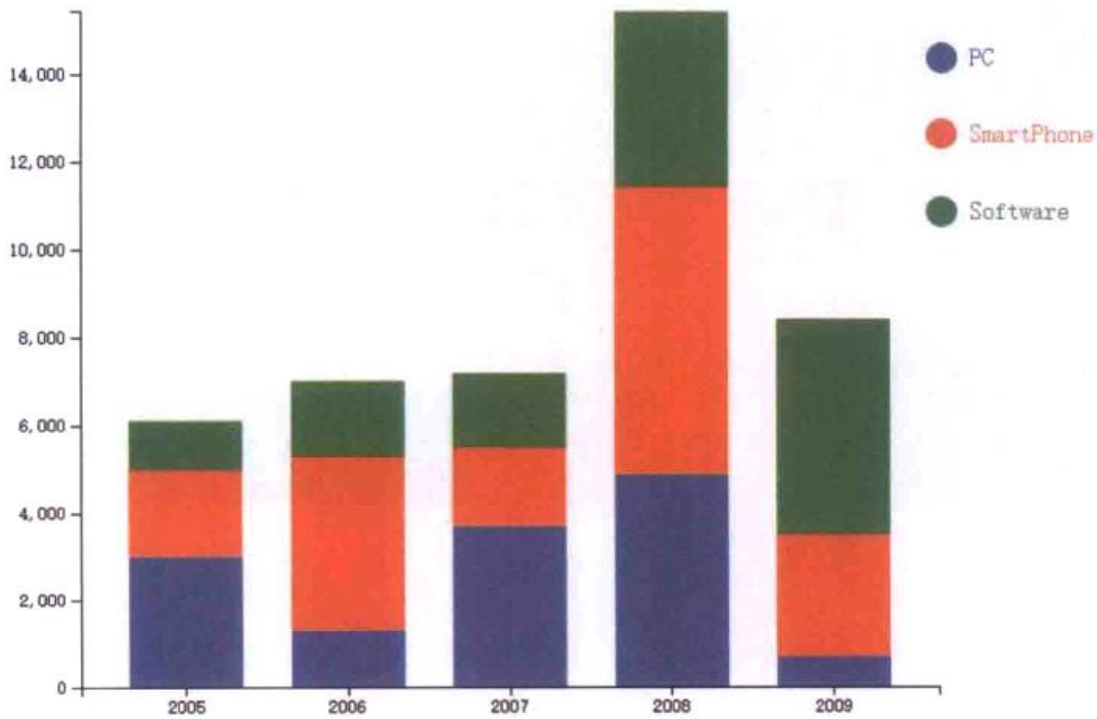


图 10-48　添加标签文字后的堆栈图

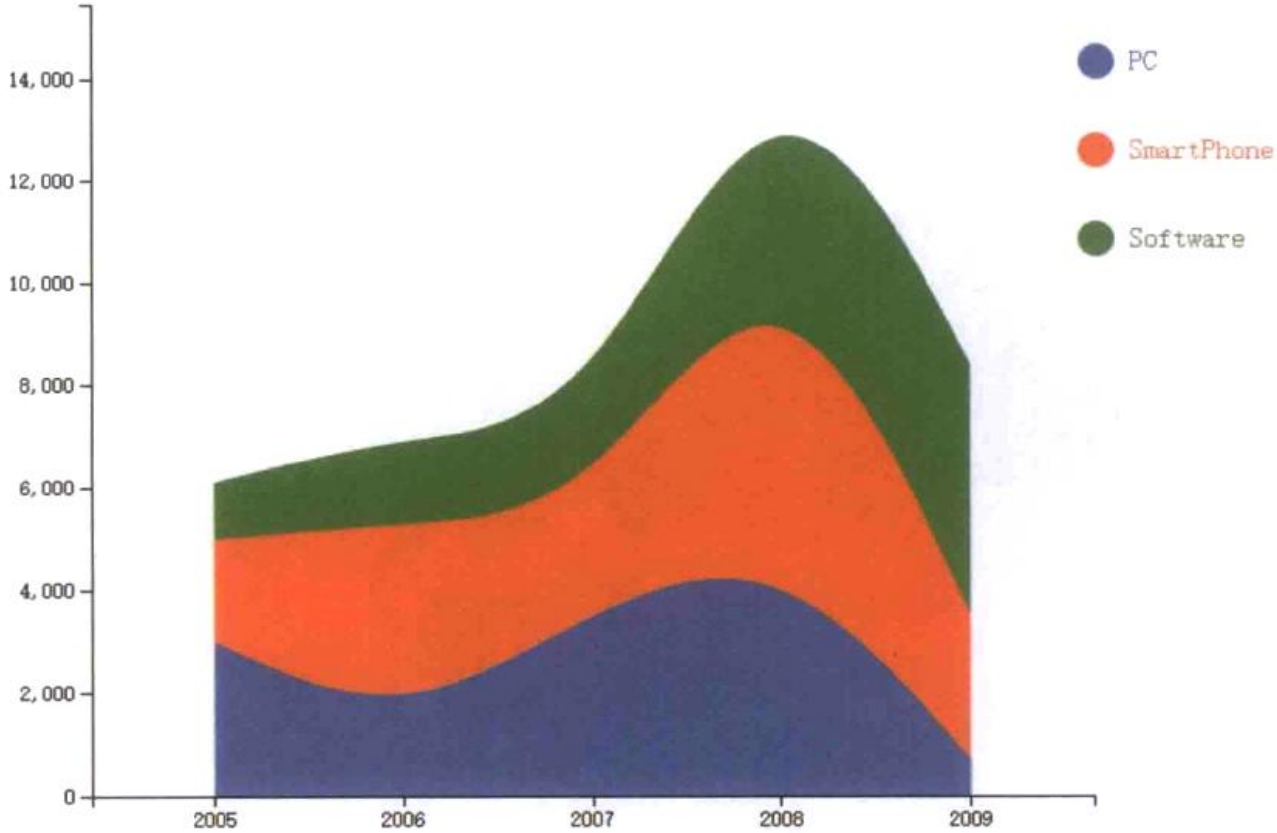


图 10-49 使用区域生成器绘制的堆栈图

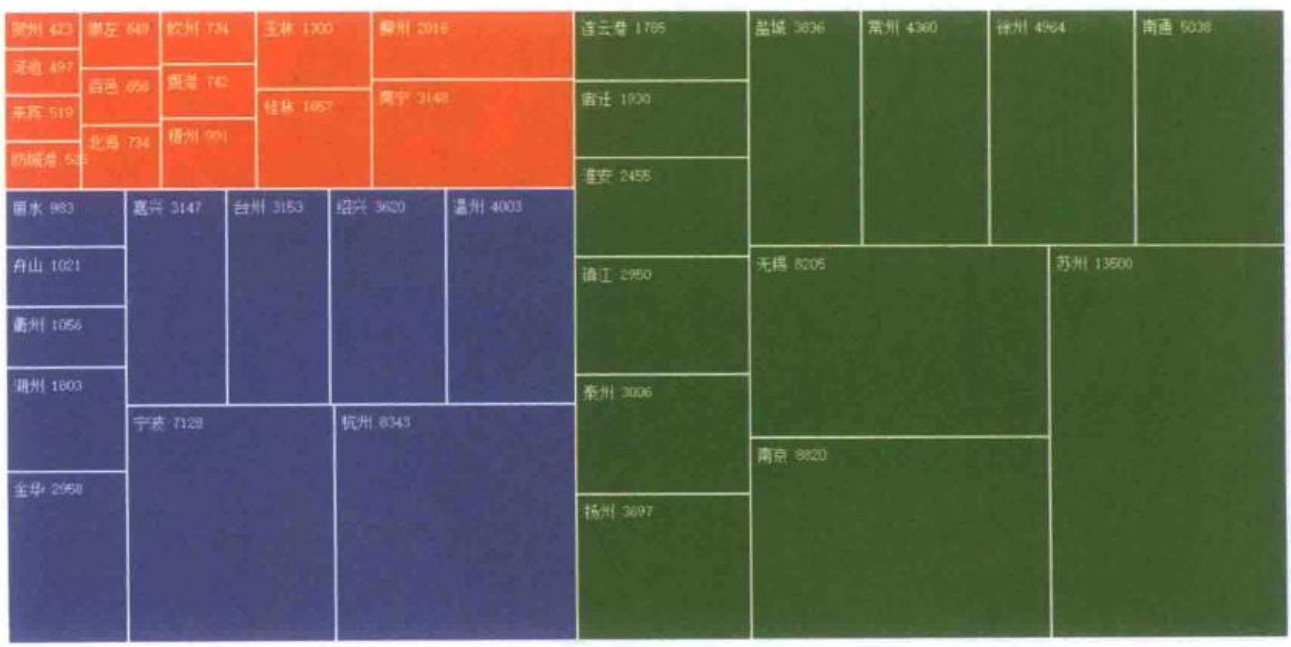


图 10-57 矩阵树图

第 11 章

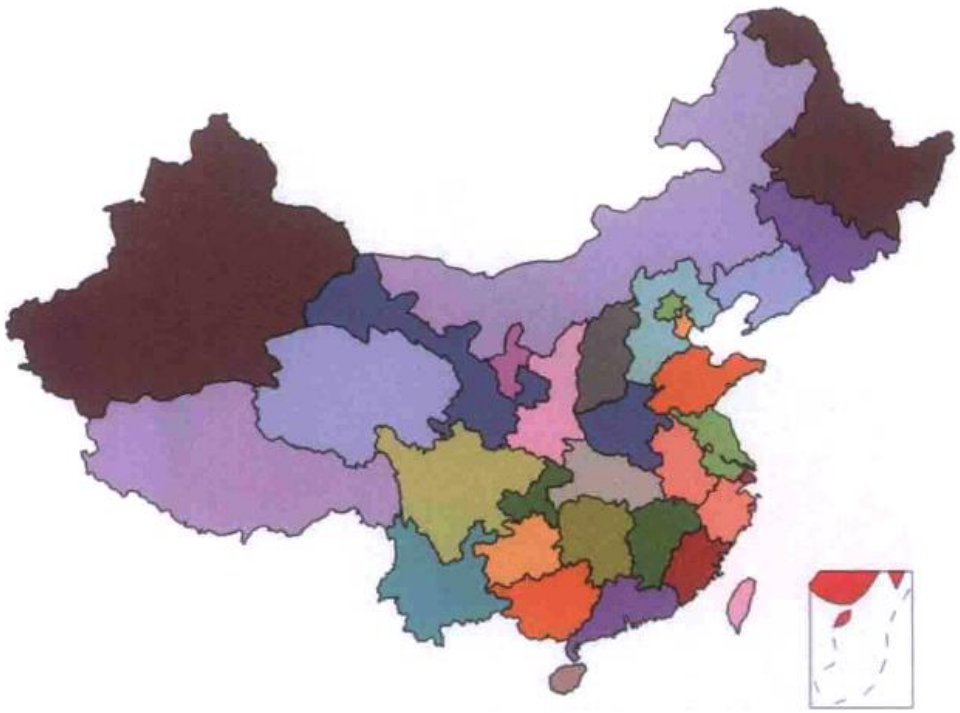


图 11-13　请求 GeoJSON 文件后绘制的中国地图

图 11-18　合并东南各省

图 11-19　西藏和新疆的交界线

图 11-21　相邻省份

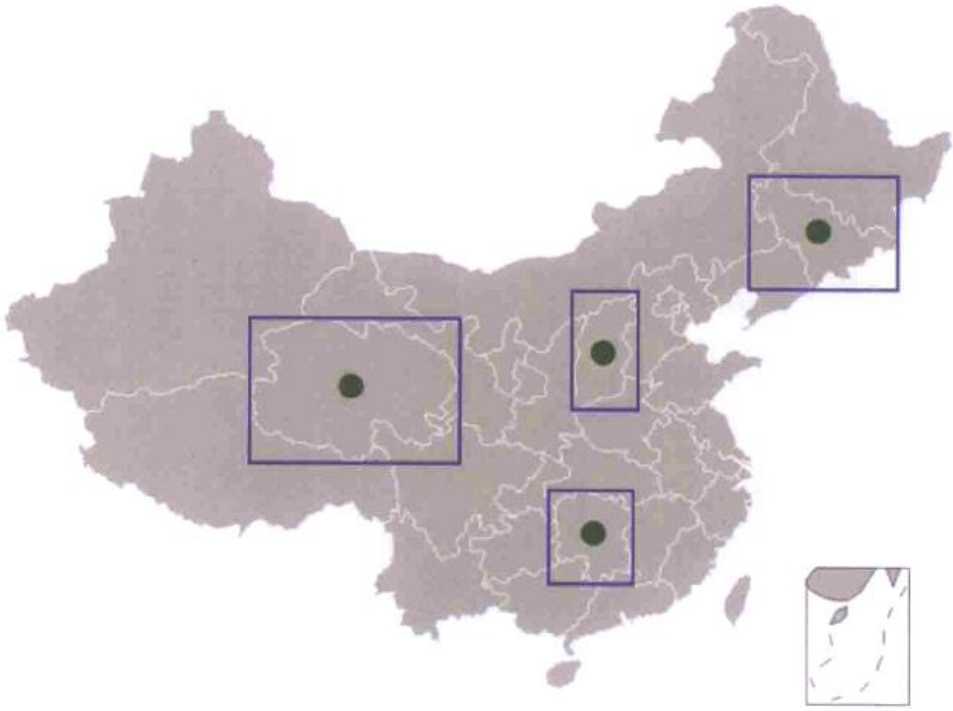


图 11-23　分别点击“青海”、“山西”、“湖南”、“吉林”后显示的中心和边界框

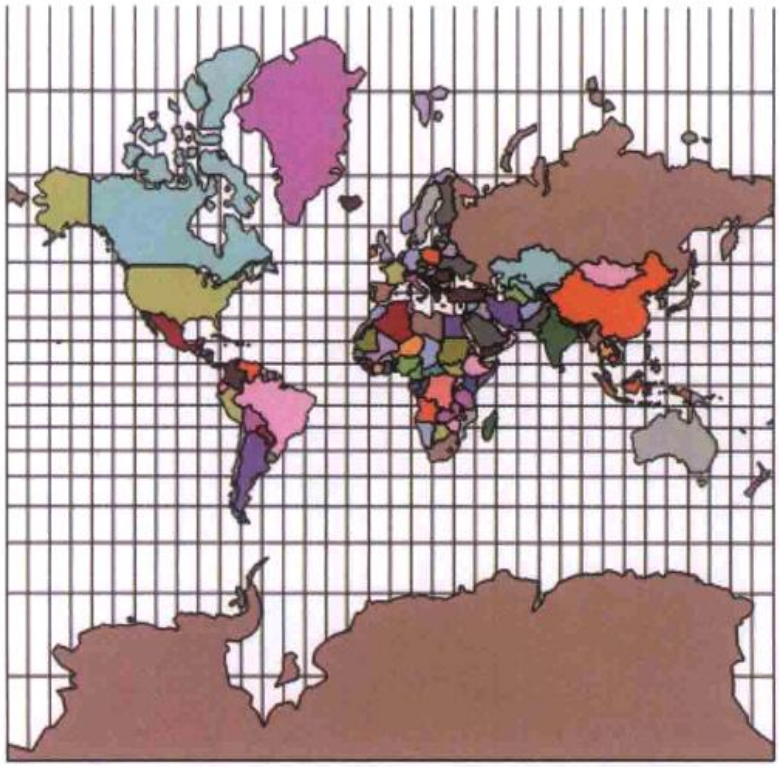


图 11-27　带经纬度网格的世界地图

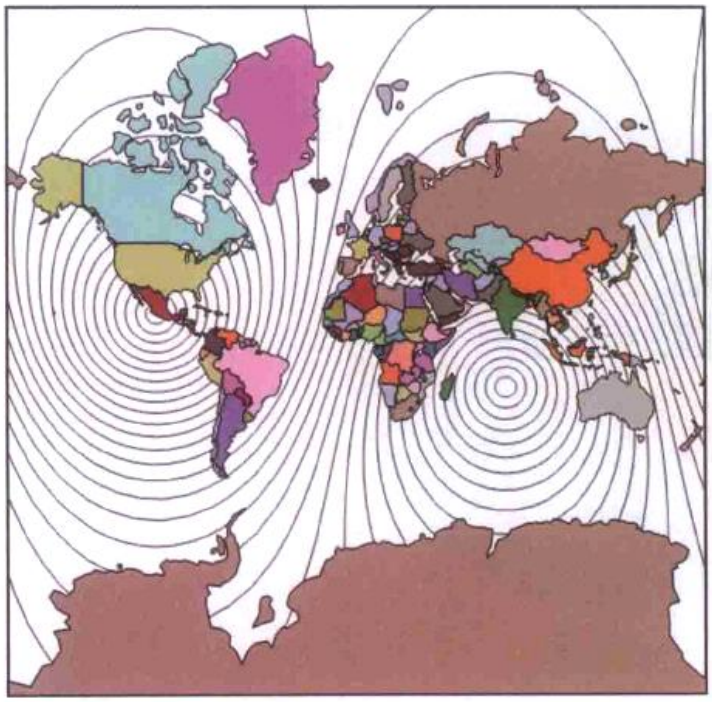


图 11-29　平面地图上显示圆形网格

图 11-30　球形地图上显示圆形网格

第 12 章

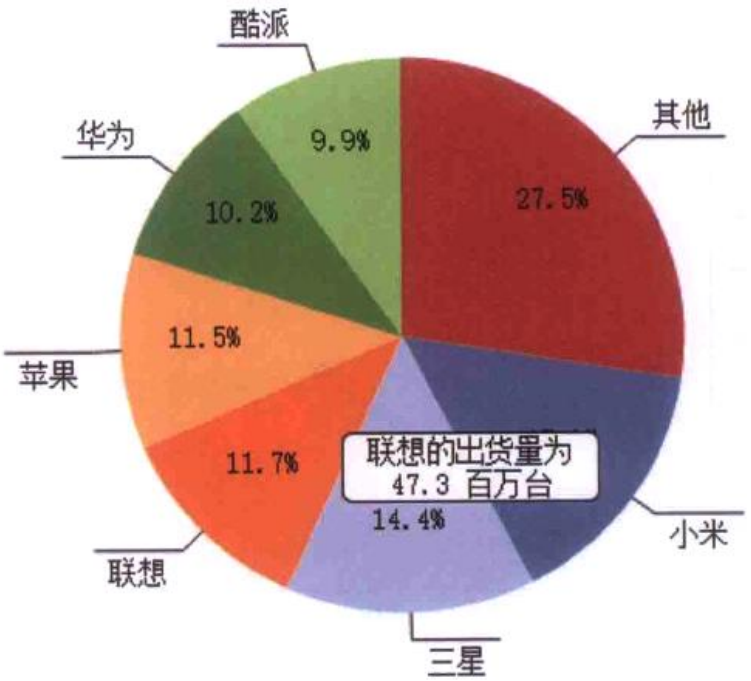


图 12-2　鼠标滑动到“联想”上时出现的提示框

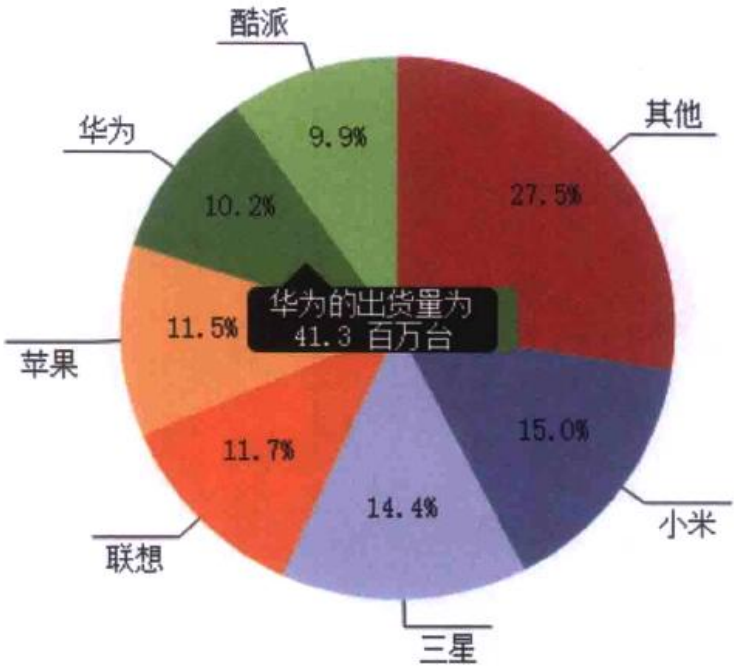


图 12-5　鼠标滑动到“华为”上

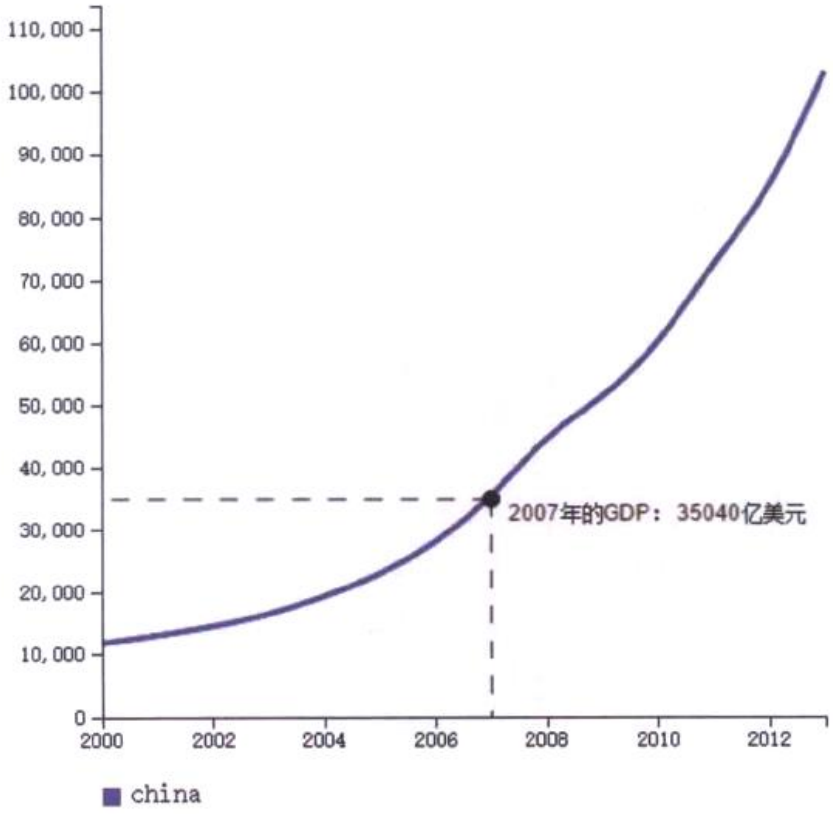


图 12-9　鼠标滑动到 2007 年附近时显示出的焦点和对齐线

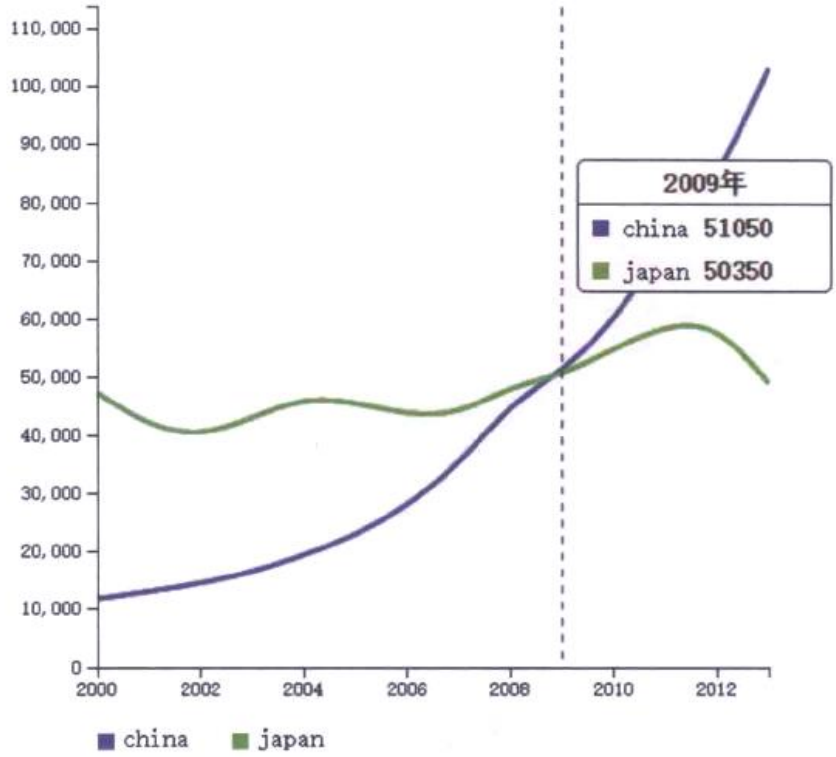


图 12-13　鼠标移到“2009 年对应的区域”附近

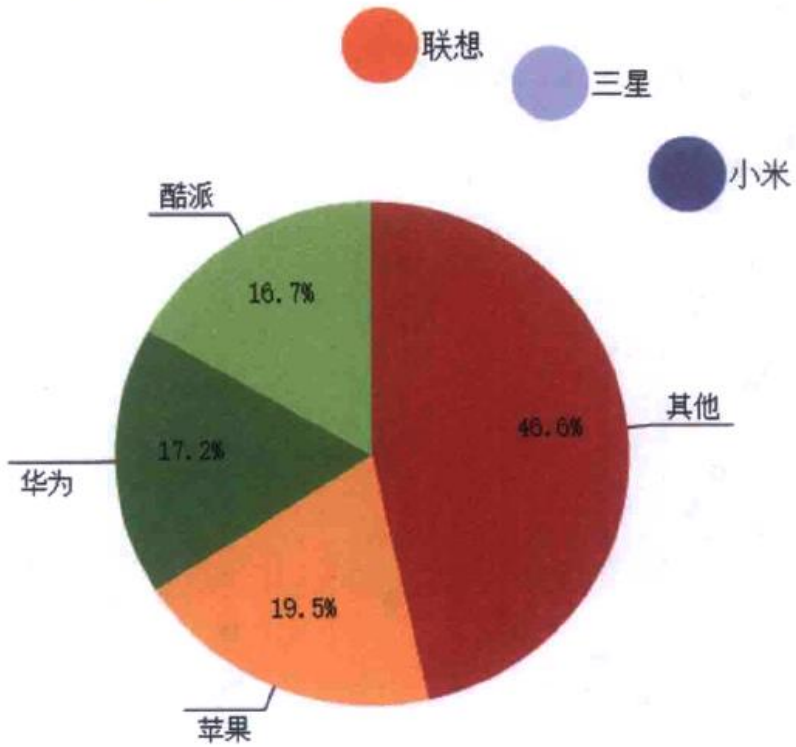


图 12-20　将“其他”和“苹果”再次移入饼状图之后

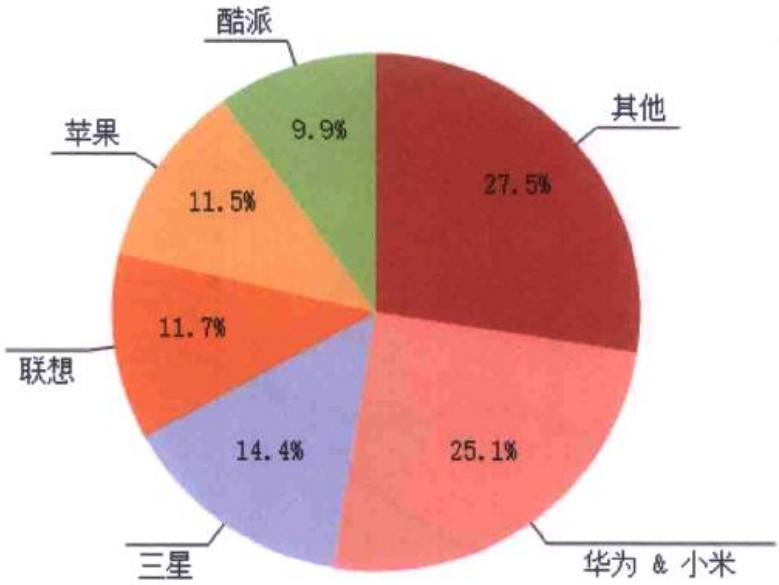


图 12-22　合并“华为”和“小米”

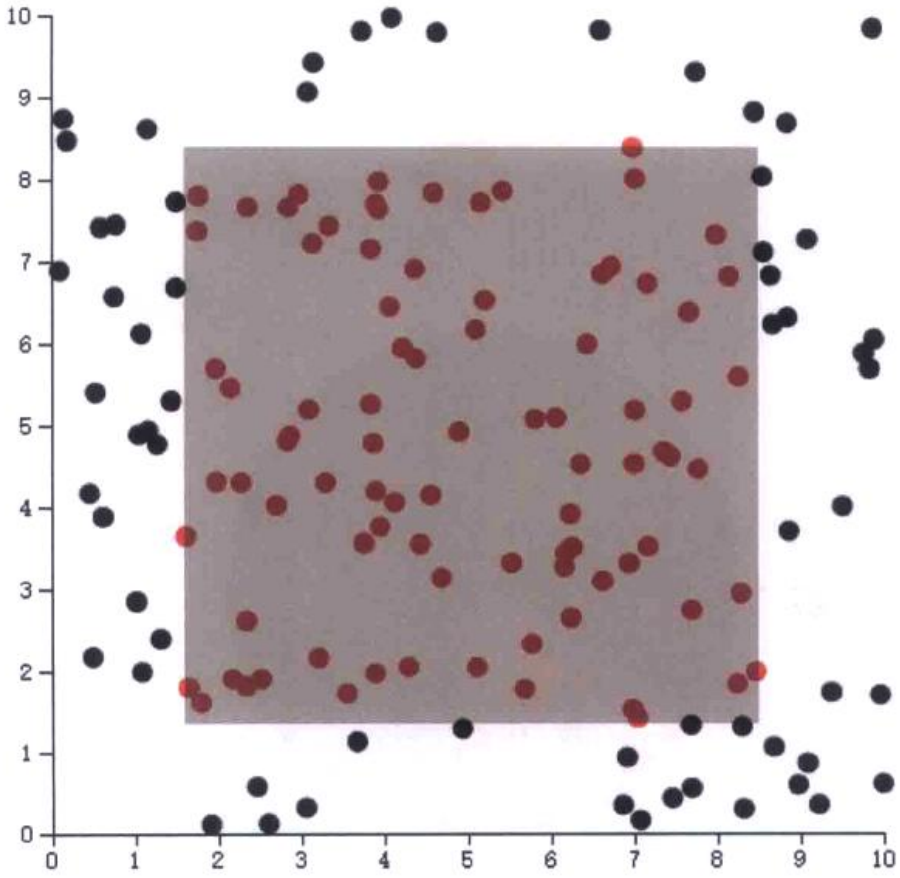


图 12-26　选择框内的散点变成了红色

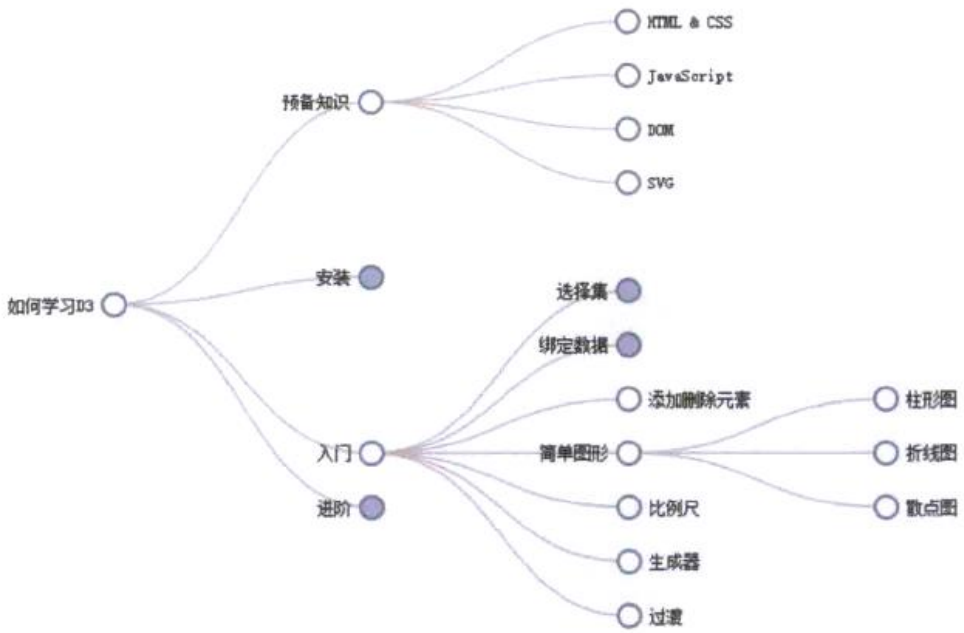


图 12-31　思维导图，展开“预备知识”和“入门”的部分节点后

第 13 章

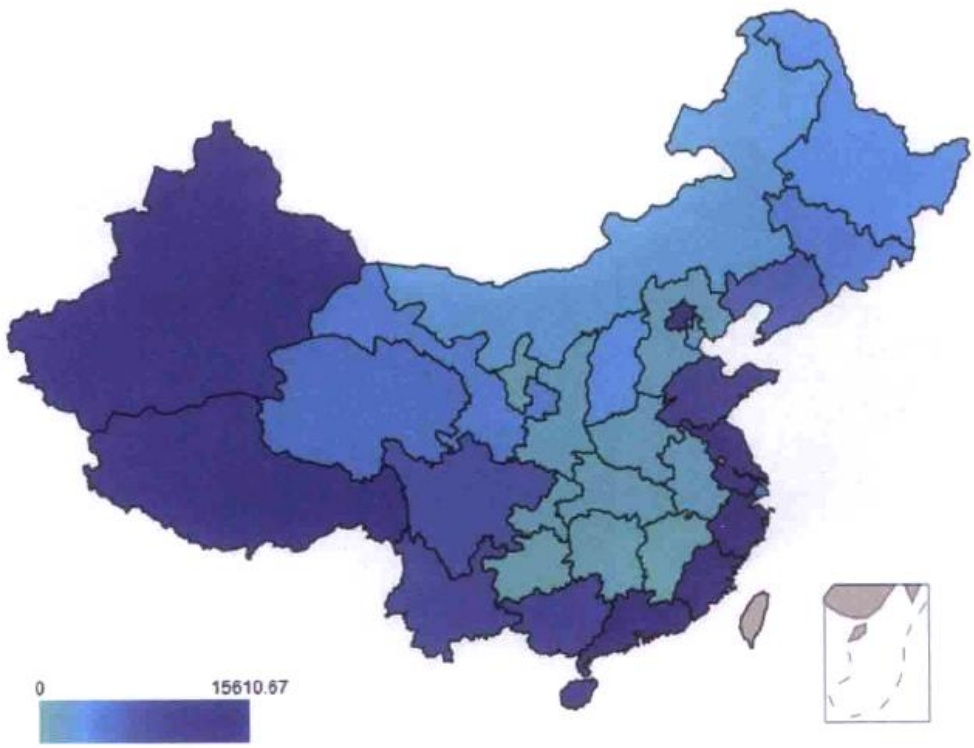


图 13-2　在左下角添加了渐变填充的矩形

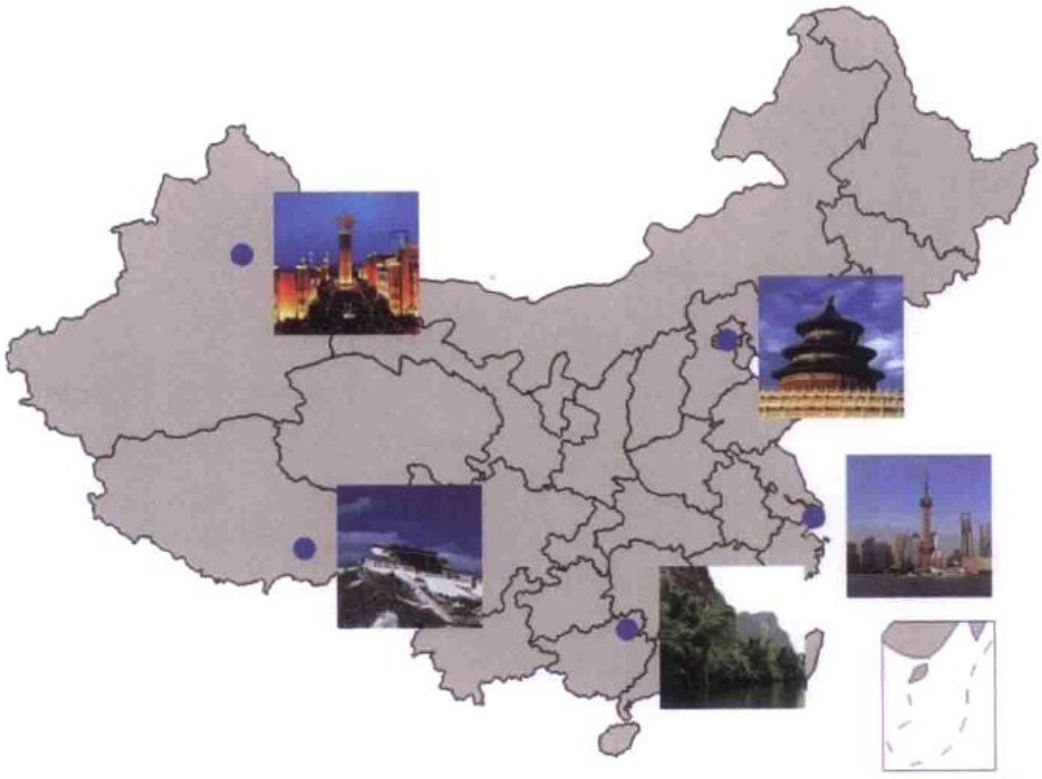


图 13-3　标注地点

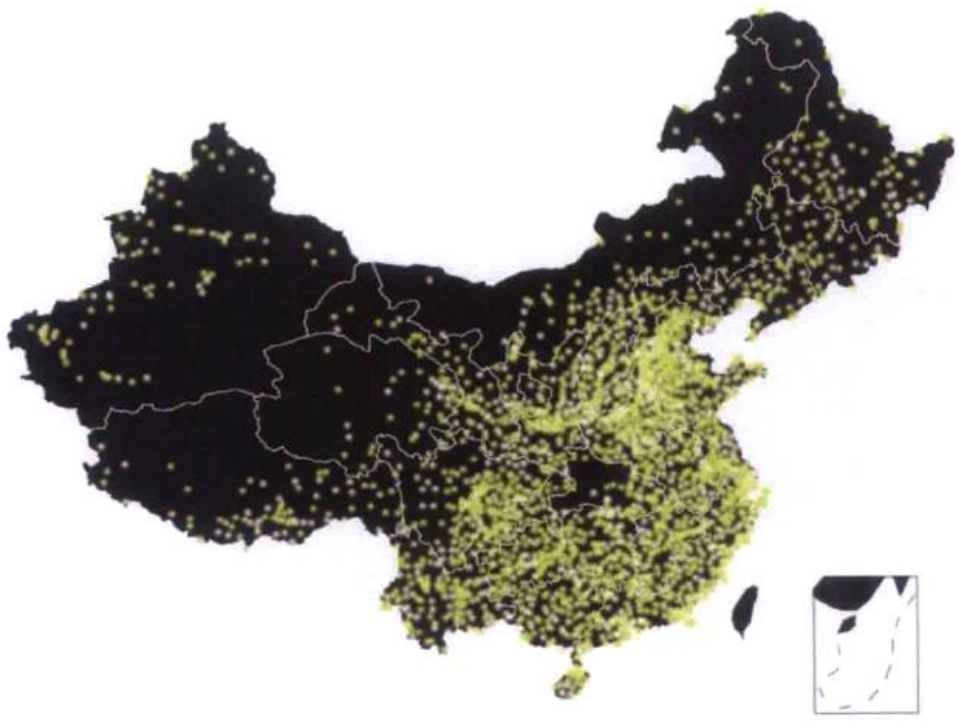


图 13-4　城市夜光图

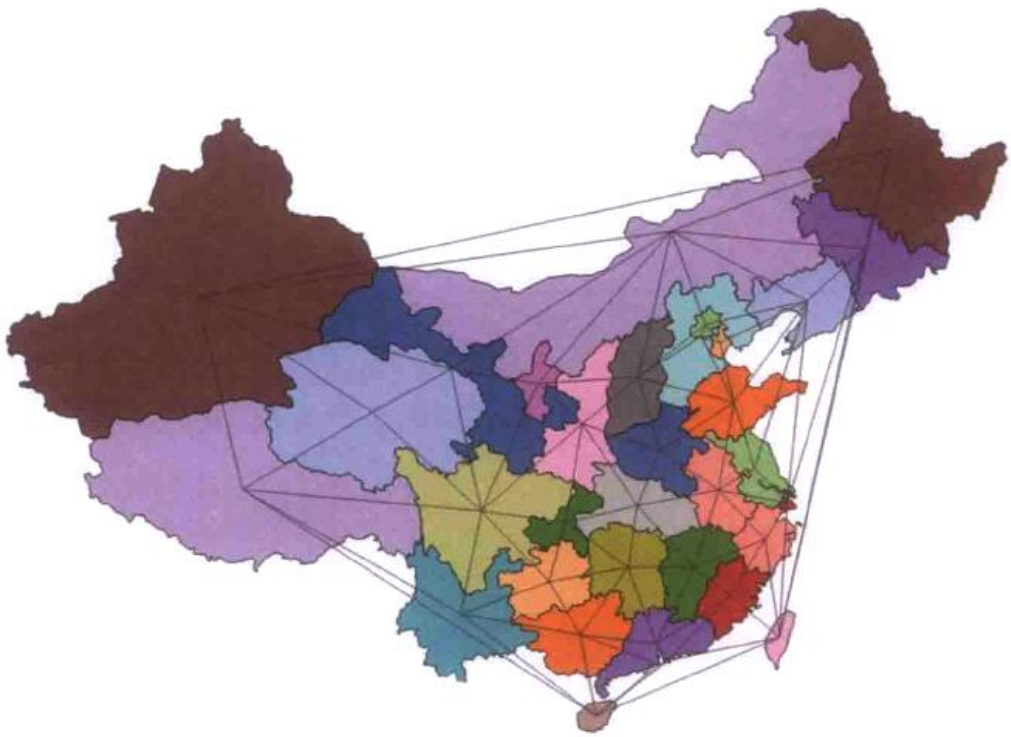


图 13-17　地图的初始状态，各省份之间的连线由三角剖分计算得到

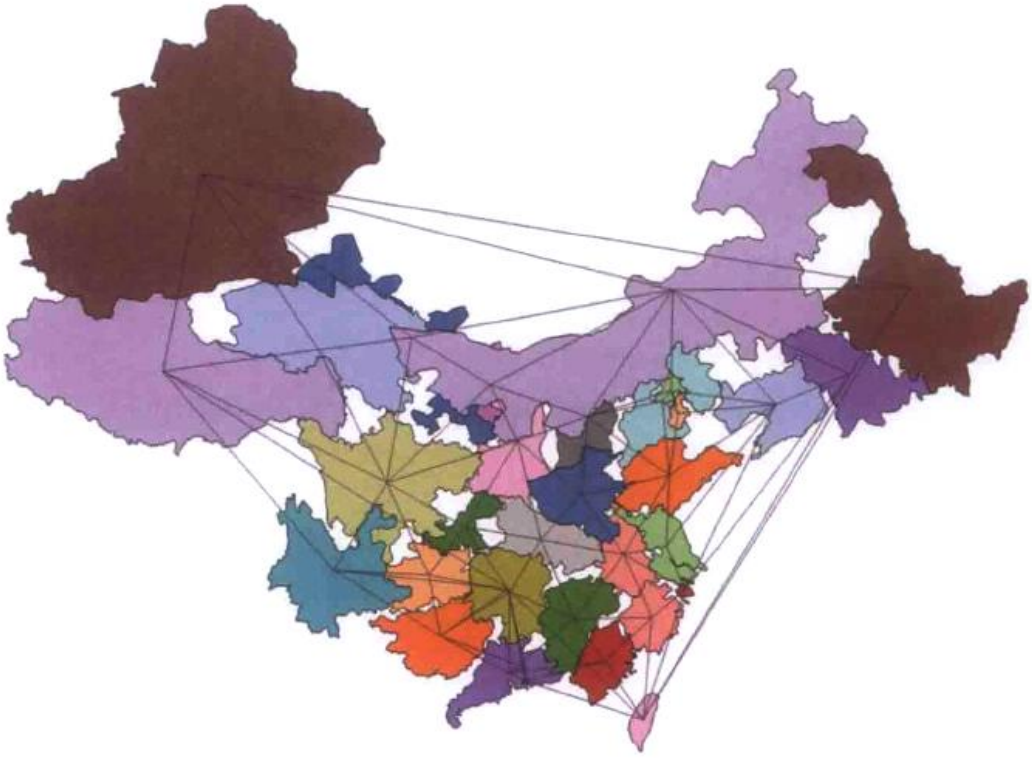


图 13-19　力导向地图

附录 B

参考文献

[1] Fry.B.Visualizing Data. O'Reilly Media, 2007.

[2] 斯考特·玛瑞. 数据可视化实战：使用 D3 设计交互式图表. 李林峰，译. 人民邮电出版社 2013.

[3] 森藤大地. データ可視化「実践」入門.あんちべ，2014.

[4] D3 官方网站. http://d3js.org/.

[5] Mike Bostock 的博客. http://bost.ocks.org/mike/.

[6] Sebastian Gutierrez. https://www.dashingd3js.com/table-of-contents.

[7] 阮一峰的博客. http://www.ruanyifeng.com/.

[8] 张天旭的博客. http://blog.csdn.net/tianxuzhang.

[9] 清水正行的博客. http://shimz.me/blog/.

[10] W3school 在线教程. http://www.w3school.com.cn/.

[11] SVG 官方文档. http://www.w3.org/TR/SVG11/.

[12] Mike Bostock. How selections Work. http://bost.ocks.org/mike/selection/.

[13] Common Format and MIME Type for Comma-Separated Values (CSV) Files. https://tools.ietf.org/html/rfc4180.

[14] USING CIRCOS TO VISUALIZE TABLES. http://circos.ca/guide/tables/.

[15] svg 要素の基本的な使い方まとめ. http://www.h2.dion.ne.jp/~defghi/svgMemo/svgMemo_20.htm.

[16] https://docs.python.org/release/3.1.3/library/string.html#formatspec.

[17] 关于 GeoJSON 的定义. http://geojson.org/.

[18] 关于 TopoJSON 的定义. https://github.com/mbostock/topojson/wiki.