D3

简介

D3 (或者叫 **D3.js**)是一个基于 web 标准的 JavaScript 可视化库。 D3 可以借助 SVG, Canvas 以及 HTML 将你的数据生动的展现出来。 D3 结合了强大的可视化交互技术以及数据驱动 DOM 的技术， 让你可以借助于现代浏览器的强大功能自由的对数据进行可视化。

如果需要标准条形图、折线图或饼图，应该考虑使用Chart.js等库。但是，如果需要定制图表或有非常精确的需求，则应考虑 D3。

D3 的功能包括：

- 数据驱动的 HTML 和 SVG 元素修改
- 比例函数（scale函数）
- 加载和转换数据（例如 CSV 数据）
- 生成复杂图表的助手，例如树形图、压缩圆、网络图、地图
- 用于在不同图表状态之间制作动画的强大转换系统（过度系统）
- 强大的用户交互支持，包括平移、缩放和拖动

起步

如果使用 `npm`, 则可以通过 `npm install d3` 来安装. 此外还可以下载 最新版 (opens new window), 最新版支持 AMD、CommonJS 以及基础标签引入形式. 你也可以直接从 d3js.org (opens new window), CDNJS (opens new window), 或者 unpkg (opens new window)加载. 比如:

```
<script src="https://d3js.org/d3.v5.min.js"></script>
```

你也可以单独使用 `d3` 中的某个模块, 比如单独使用 d3-selection

```
<script src="https://d3js.org/d3-selection.v1.js"></script>
```

D3基于 ES2015 modules (opens new window)开发. 可以使用 `Rollup`, `webpack` 或者其他你偏爱的打包工具进行构建. 在一个符合 ES2015 的应用中导入 `d3` 或者 `d3` 的某些模块:

```
import {scaleLinear} from "d3-scale";
```

或者导入 `d3` 的全部功能并且设置命名空间 (这里是 `d3`):

```
import * as d3 from "d3";
```

在 Nodejs 环境中:

```
var d3 = require("d3");
```

你也可以导入多个模块然后将这些模块集合到 `d3` 对象中, 此时使用 Object.assign (opens new window):

```
var d3 = Object.assign({}, require("d3-format"), require("d3-geo"), require("d3-geo-projection"));
```

支持环境

`D3` 5+ 支持最新浏览器，比如 Chrome，Edge，Firefox 以及 Safari。D3 4以及之前的版本支持 IE 9 以上的版本。D3 的一部分功能能在旧版的浏览器中运行，因为 D3 的核心功能对浏览器的要求比较低。例如 d3-selection 使用 Level 1 级 Selectors API (opens new window)，但是可以通过预先加载Sizzle (opens new window)来实现兼容。现代浏览器对 SVG (opens new window)和 CSS3 Transition (opens new window)的支持比较好。

本地开发

由于浏览器的安全限制，不能直接读取本地文件。在本地开发的时候，必须要运行一个服务器环境而不是使用`file://`, 推荐使用Nodejs的http-server (opens new window)，安装方法:

```
npm install -g http-server
```

运行:

```
http-server &
```

然后会在当前目录启动一个 http://localhost:8080 的服务。

选择（Selections）

D3 可以选择一些 HTML 或 SVG 元素并更改它们的样式和/或属性：`d3.select`和`d3.selectAll`。

这两个函数都将字符串作为其唯一参数。该字符串指定要选择的元素，并采用 CSS 选择器字符串的形式（例如`div.item`，`#my-chart`或`g:first-child`）。

- **做出选择后，您可以使用以下函数修改其中的元素：**

Name	行为	例子
`.style`	更新样式	`d3.selectAll('circle').style('fill', 'red')`
`.attr`	更新属性	`d3.selectAll('rect').attr('width', 10)`
`.classed`	添加/删除类属性	`d3.select('.item').classed('selected', true)`
`.property`	更新元素的属性	`d3.selectAll('.checkbox').property('checked', false)`
`.text`	更新文本内容	`d3.select('div.title').text('My new book')`
`.html`	更改html内容	`d3.select('.legend').html('<div class="block"></div><div>0 - 10</div>')`

[//]: `.classed` 是一个 boolean.

除了将常量值传递给`.style`, `.attr`, `.classed`, `.property`,之外`.text`， `.html`您还可以传入一个函数。例如:

```
d3.selectAll('circle')
  .attr('cx', function(d, i) {
    return i * 100;
  });
```

该函数接受两个参数，通常命名为`d`和`i`。第一个参数`d`是**连接数据**（或“数据”）。`i`是选择中元素的**索引**。

- **可以使用该方法将事件处理程序添加到选定元素`.on`。**

此方法有两个参数：第一个是指定事件类型的字符串;第二个是触发事件时调用的函数（“回调函数”）。此回调函数有两个参数，通常命名为`e`和`d`。`e`是 DOM 事件对象并且`d`是连接数据。

最常见的事件包括（有关详细信息，请参阅MDN 事件参考）：

活动名称	描述
`click`	元素已被点击
`mouseenter`	鼠标指针已移动到元素上
`mouseover`	鼠标指针已移动到元素或其子元素上
`mouseleave`	鼠标指针已移离元素
`mouseout`	鼠标指针已移离元素或其子元素
`mousemove`	鼠标指针移到元素上

在事件回调函数中，`this`变量绑定到触发事件的 DOM 元素。这使我们能够执行以下操作：

```
d3.selectAll('circle')
  .on('click', function(e, d) {
    d3.select(this)
      .style('fill', 'orange');
  });
```

[//]:这`this`是一个 DOM 元素，而不是 D3 选择，因此如果您想使用 D3 修改它，您必须首先使用`d3.select(this)`.

- **插入和删除元素**

`.append`可以使用 D3和`.insert`方法将元素添加到选择的元素中。可以使用 删除元素`.remove`。

`.append`将一个元素附加到选择中的每个元素。如果元素已经有子元素，则新元素将成为**最后一个子元素**。第一个参数指定元素的类型。

```
d3.selectAll('g.item')
  .append('text')
  .text('A');
```

`.insert`类似于`.append`但它允许我们指定第二个参数，**该参数指定（作为 CSS 选择器）在哪个元素之前插入新元素**。

```
d3.selectAll('g.item')
  .insert('text', 'circle')
  .text('A');
```

`.remove`从页面中**删除选择中的所有元素**。例如，给定一些圆圈，您可以使用以下方法删除它们：

```
d3.selectAll('circle')
  .remove();
```

- **链接**

大多数选择方法的返回值是选择本身。这意味着诸如 和 之类的选择方法`.style`可以`.attr`链接`.on`起来。例如：

```
d3.selectAll('circle')
  .style('fill', '#333')
  .attr('r', 20)
  .on('click', function(d, i) {
    d3.select(this)
      .style('fill', 'orange');
  });
```

- **每个**

该`.each`方法允许您为**选择的每个元素**调用一个函数。

回调函数有两个参数，通常命名为`d`和`i`。第一个参数`d`是**连接数据**。`i`是选择中元素的**索引**。`this`关键字是指选择中的当前 HTML 或 SVG 元素。

这是一个示例，`.each`用于为每个选择的元素调用函数。该函数计算索引是奇数还是偶数，并相应地修改圆：

```
d3.selectAll('circle')
  .each(function(d, i) {
    var odd = i % 2 === 1;

    d3.select(this)
      .style('fill', odd ? 'orange' : '#ddd')
      .attr('r', odd ? 40 : 20);
  });
```

- **调用**

该`.call`方法允许调用一个函数，**选择本身**作为第一个参数传递给该函数。

`.call`在您想要对选择进行操作的可重用函数时很有用。

例如，`colorAll`获取一个选区并将选区元素的填充设置为橙色：

```
function colorAll(selection) {
  selection
    .style('fill', 'orange');
}

d3.selectAll('circle')
  .call(colorAll);
```

- **筛选和排序选择**

您可以使用 D3 的`.filter`方法过滤选择。第一个参数是一个函数，它返回`true`是否应该包含元素。过滤的选择由该`filter`方法返回，因此您可以继续链接选择方法。

在此示例中，您过滤偶数元素并将它们着色为橙色：

```
d3.selectAll('circle')
  .filter(function(d, i) {
    return i % 2 === 0;
  })
  .style('fill', 'orange');
```

通过调用`.sort`和传入比较器函数对选择中的元素进行排序。比较器函数有两个参数，通常是`a`和`b`，它们代表被比较的两个元素的数据。如果比较器函数返回负数，`a`将放在前面`b`，如果是正数，`a`将放在后面`b`。

```
 d3.selectAll('.person')
    .sort(function(a, b) {
      return b.score - a.score;
    });
```

数据连接(Data joins)

数据连接在数据数组和 HTML 或 SVG 元素的选择之间创建对应关系。

将数组加入 HTML/SVG 元素意味着：1.添加或删除 HTML（或 SVG）元素，以便**每个数组元素都有一个对应的 HTML（或 SVG）元素**；2.每个 HTML/SVG 元素都可以根据其对应数组元素的值进行**定位、调整大小和样式**

- **创建数据连接**

创建数据连接的一般模式是：

```
d3.select(container)
  .selectAll(element-type)
  .data(array)
  .join(element-type);
```

1. `container`是一个 CSS 选择器字符串，它指定将包含连接的 HTML/SVG 元素的**单个元素**
1. `element-type`是描述您要加入**的元素类型的**字符串（例如“div”或“circle”）
3. `array`是您要加入**的阵列**的名称

更新连接的元素：

连接的 HTML 或 SVG 元素可以使用`.style`的`.attr`和`.text`方法进行更新。

数据驱动更新：

如果传入一个**函数**`.attr`，或者您可以以**数据驱动的**方式`.style`更新 HTML/SVG 元素。

为selection 中的每个元素调用该函数。它有两个参数，通常命名为`d`和`i`。

第一个参数 (`d`) 表示相应的数组元素（或“连接值”）。第二个参数`i`表示选择中元素的索引。

函数的返回值用于设置属性或样式值。

```
let myData = [40, 10, 20, 60, 30];

d3.select('.chart')
  .selectAll('circle')
  .data(myData)
  .join('circle')
  .attr('cx', function(d, i) {
    return i * 100;
  })
  .attr('cy', 50)
  .attr('r', 40)
  .style('fill', 'orange');
```

- **连接元素数组**

```
var cities = [
  { name: 'London', population: 8674000},
  { name: 'New York', population: 8406000},
  { name: 'Sydney', population: 4293000},
  { name: 'Paris', population: 2244000},
  { name: 'Beijing', population: 11510000}
];

d3.select('.chart')
  .selectAll('circle')
  .data(cities)
  .join('circle')
  .attr('cx', function(d, i) {
    return i * 100;
  })
  .attr('cy', 50)
  .attr('r', function(d) {
```

```
    let scaleFactor = 0.00004;
    return scaleFactor * d.population;
  })
  .style('fill', '#aaa');
```

- **更新函数**

如果您的数据数组发生更改，**您将需要再次执行连接**。（与 Vue.js 等一些框架不同，D3 不会自动为您执行此操作。）

因此我们通常将连接代码放在一个函数中。每当数据发生变化时，我们都会调用这个函数。

我们将数据数组传递到`update`. 每次`update`调用都会执行连接。

```
function getData() {
  let data = [];
  let numItems = Math.ceil(Math.random() * 5);

  for(let i=0; i<numItems; i++) {
    data.push(Math.random() * 60);
  }

  return data;
}

function update(data) {
  d3.select('.chart')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cx', function(d, i) {
      return i * 100;
    })
    .attr('cy', 50)
    .attr('r', function(d) {
      return 0.5 * d;
    })
    .style('fill', function(d) {
      return d > 30 ? 'orange' : '#eee';
    });
}

function updateAll() {
  let myData = getData();
  update(myData);
}

updateAll();
```

```
d3.select("button")
  .on("click", updateAll);
```

- **键函数（Key functions）**

当 D3 执行数据连接时，它将第一个数组元素连接到选择中的第一个元素，将第二个数组元素连接到选择中的第二个元素，依此类推。

但是，如果数组元素的顺序发生变化（由于排序、插入或删除元素），则数组元素可以连接**到不同的 DOM 元素**。

您可以通过将**键函数**传递给`.data`方法来确保每个数组元素保持连接到相同的 HTML/SVG 元素。键函数应该为每个数组元素返回一个**唯一的 id 值。**

```
d3.select('#content')
        .selectAll('div')
        .data(data, function(d) {
            return d;
        })
        .join('div')
        .transition()
        .style('left', function(d, i) {
            return i * 32 + 'px';
        })
        .text(function(d) {
            return d;
        });
```

- **调试**

当 D3 执行数据连接时，它会为`__data__`选择中的每个 DOM 元素添加一个属性，并将连接的数据分配给它。

我们可以在谷歌浏览器中通过右键单击一个元素，选择“检查”并输入：

```
$0.__data__
```

`$0`表示正在检查的元素。

进入、退出和更新

本章解释了如何额外控制 HTML 和 SVG 元素在创建、更新或删除时的行为方式。可以使用这些方法实现过渡效果和一些特定效果（例如元素淡入和淡出）。

（刚刚创建的 HTML/SVG 元素被称为**进入**元素，即将被移除的元素被称为**退出**元素。）

可以通过将函数传递给方法来区别对待进入和退出元素`.join`：

```
.join(
  function(enter) {
    ...
  },
  function(update) {
    ...
  },
  function(exit) {
    ...
  }
)
```

每个函数都有一个参数:

1. enter函数的参数`enter`是表示需要创建的元素的**enter selection**
2. 更新函数的参数`update`是一个包含已经存在的元素的选择（并且没有退出）
3. exit 函数的参数`exit`是**退出选择**，包含需要移除的元素

进入、更新和退出函数**必须返回 selection**。

```
function getData() {
  let data = [];
  let numItems = Math.ceil(Math.random() * 5);

  for(let i=0; i<numItems; i++) {
    data.push(40);
  }

  return data;
}

function update(data) {
  d3.select('.chart')
    .selectAll('circle')
    .data(data)
    .join(
      function(enter) {
        return enter.append('circle')
          .style('opacity', 0.25);
      },
      function(update) {
        return update.style('opacity', 1);
      }
    )
    .attr('cx', function(d, i) {
      return i * 100;
    })
    .attr('cy', 50)
    .attr('r', function(d) {
```

```
      return 0.5 * d;
    })
    .style('fill', 'orange');
}

function updateAll() {
  let myData = getData();
  update(myData);
}

updateAll();

d3.select("button")
  .on("click", updateAll);
```

更多效果：https://www.d3indepth.com/transitions/

比例函数（Scale functions）

Scale 函数是 JavaScript 函数，它们：接受输入（通常是数字、日期或类别）并返回一个值（例如坐标、颜色、长度或半径）。它们通常用于将（或“映射”）数据值转换为视觉变量（例如位置、长度和颜色）。

- **构建尺度**

要创建线性比例，您可以使用：

```
let myScale = d3.scaleLinear();

myScale
  .domain([0, 100])
  .range([0, 800]);

myScale(0);    // returns 0
myScale(50);   // returns 400
myScale(100);  // returns 800
```

- **缩放类型**

D3 有大约 12 种不同的比例类型（scaleLinear、scalePow、scaleQuantise、scaleOrdinal 等），广义上讲它们可以分为 3 组：

1. 具有连续输入和连续输出
2. 具有连续输入和离散输出
3. 具有离散输入和离散输出
    - **具有连续输入和连续输出**
        - scaleLinear：使用线性函数y=mx+c在域和范围内进行插值。

```
let linearScale = d3.scaleLinear()
  .domain([0, 10])
  .range([0, 600]);

linearScale(0);   // returns 0
linearScale(5);   // returns 300
linearScale(10);  // returns 600
```

通常用于将数据值转换为位置和长度。它们在创建条形图、折线图和许多其他图表类型时很有用。

输出范围也可以指定为颜色：

```
let linearScale = d3.scaleLinear()
  .domain([0, 10])
  .range(['yellow', 'red']);

linearScale(0);   // returns "rgb(255, 255, 0)"
linearScale(5);   // returns "rgb(255, 128, 0)"
linearScale(10);  // returns "rgb(255, 0, 0)"
```

- scaleSqrt：对于按面积（而不是半径）确定圆的大小很有用。

```
let sqrtScale = d3.scaleSqrt()
  .domain([0, 100])
  .range([0, 30]);

sqrtScale(0);   // returns 0
sqrtScale(50);  // returns 21.21...
sqrtScale(100); // returns 30
```

- scalePow：是更通用的版本`scaleSqrt`。该比例使用幂函数$y=mx^k+c$进行插值。指数`k`使用`.exponent()`。

```
let powerScale = d3.scalePow()
  .exponent(0.5)
  .domain([0, 100])
  .range([0, 30]);

powerScale(0);   // returns 0
powerScale(50);  // returns 21.21...
powerScale(100); // returns 30
```

- scaleLog：使用对数函数y=m*log(x)+b进行插值，当数据具有指数性质时可能很有用。

```
let logScale = d3.scaleLog()
  .domain([10, 100000])
  .range([0, 600]);

logScale(10);     // returns 0
logScale(100);    // returns 150
logScale(1000);   // returns 300
logScale(100000); // returns 600
```

- scaleTime：类似于，`scaleLinear`除了域表示为日期数组。（在处理时间序列数据时**非常有用。**）

```
timeScale = d3.scaleTime()
  .domain([new Date(2016, 0, 1), new Date(2017, 0, 1)])
  .range([0, 700]);

timeScale(new Date(2016, 0, 1));   // returns 0
timeScale(new Date(2016, 6, 1));   // returns 348.00...
timeScale(new Date(2017, 0, 1));   // returns 700
```

- scaleSequential：用于将连续值映射到由预设（或自定义）**插值器**确定的输出范围。（插值器是一个函数，它接受 0 到 1 之间的输入并输出两个数字、颜色、字符串等之间的插值。）

```
let sequentialScale = d3.scaleSequential()
  .domain([0, 100])
  .interpolator(d3.interpolateRainbow);

sequentialScale(0);   // returns 'rgb(110, 64, 170)'
sequentialScale(50);  // returns 'rgb(175, 240, 91)'
sequentialScale(100); // returns 'rgb(110, 64, 170)'
```

还有一个插件d3-scale-chromatic，它提供了众所周知的ColorBrewer配色方案。

- Clamping：默认情况下当输入值超出域时，`scaleLinear`、`scalePow`、`scaleSqrt`、`scaleLog`、`scaleTime`和`scaleSequential`仍会计算。

```
let linearScale = d3.scaleLinear()
  .domain([0, 10])
  .range([0, 100]);

linearScale(20);  // returns 200
linearScale(-10); // returns -100
```

可以通过使用以下方法**限制缩放**功能，以便输入值保持在域内`.clamp`：

```
linearScale.clamp(true);

linearScale(20);  // returns 100
linearScale(-10); // returns 0
```

您可以使用 `.clamp(false)` 关闭。

- Nice：如果域是根据实际数据自动计算的（例如，通过使用 `d3.extent`），则开始值和结束值可能不是整数。这不一定是个问题，但它可能看起来有点不整洁。因此，D3 `.nice()` 提供了一个刻度函数，它将域四舍五入到“不错”的舍入值。

```
let data = [0.243, 0.584, 0.987, 0.153, 0.433];
let extent = d3.extent(data);

let linearScale = d3.scaleLinear()
  .domain(extent)
  .range([0, 100])
  .nice();
```

请注意，`.nice()` 每次更新域时都必须调用。

- Multiple segments：`scaleLinear`, `scalePow`, `scaleSqrt`, `scaleLog` and `scaleTime` 通常由两个值组成，但如果您提供 3 个或更多值，则会细分为多个段

```
let linearScale = d3.scaleLinear()
  .domain([-10, 0, 10])
  .range(['red', '#ddd', 'blue']);

linearScale(-10);  // returns "rgb(255, 0, 0)"
linearScale(0);    // returns "rgb(221, 221, 221)"
linearScale(5);    // returns "rgb(111, 111, 238)"
```

- Inversion：
- 该方法允许您在给定**输出** `.invert()` 值的情况下确定缩放函数的**输入**值（假设缩放函数具有数值域）：

  ```
  let linearScale = d3.scaleLinear()
    .domain([0, 10])
    .range([0, 100]);

  linearScale.invert(50);   // returns 5
  linearScale.invert(100);  // returns 10
  ```

- **具有连续输入和离散输出**
  - scaleQuantize：接受连续输入并输出由范围定义的多个离散量。

```
let quantizeScale = d3.scaleQuantize()
  .domain([0, 100])
  .range(['lightblue', 'orange', 'lightgreen', 'pink']);

quantizeScale(10);  // returns 'lightblue'
quantizeScale(30);  // returns 'orange'
quantizeScale(90);  // returns 'pink'
```

- scaleQuantile：将连续数字输入映射到离散值。域由**数字数组**定义

```
let myData = [0, 5, 7, 10, 20, 30, 35, 40, 60, 62, 65, 70, 80, 90, 100];

let quantileScale = d3.scaleQuantile()
  .domain(myData)
  .range(['lightblue', 'orange', 'lightgreen']);

quantileScale(0);   // returns 'lightblue'
quantileScale(20);  // returns 'lightblue'
quantileScale(30);  // returns 'orange'
quantileScale(65);  // returns 'lightgreen'
```

- scaleThreshold：将连续数值输入映射到范围定义的离散值。*指定n-1 个*域分割点，其中*n*是范围值的数量（即range的数量）。

```
let thresholdScale = d3.scaleThreshold()
  .domain([0, 50, 100])
  .range(['#ccc', 'lightblue', 'orange', '#ccc']);

thresholdScale(-10);  // returns '#ccc'
thresholdScale(20);   // returns 'lightblue'
thresholdScale(70);   // returns 'orange'
thresholdScale(110);  // returns '#ccc'
```

- **具有离散输入和离散输出**
  - scaleOrdinal：将离散值（由数组指定）映射到离散值（也由数组指定）。域数组指定可能的输入值，范围数组指定输出值。如果范围数组比域数组短，则范围数组将重复。

```
let myData = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep',
'Oct', 'Nov', 'Dec']

let ordinalScale = d3.scaleOrdinal()
  .domain(myData)
  .range(['black', '#ccc', '#ccc']);

ordinalScale('Jan');  // returns 'black';
ordinalScale('Feb');  // returns '#ccc';
ordinalScale('Mar');  // returns '#ccc';
ordinalScale('Apr');  // returns 'black';
```

默认情况下，如果将不在域中的值用作输入，则比例尺将隐式将该值添加到域中。

如果这不是所需的行为，您可以使用以下命令为未知值指定输出值`.unknown()`。

- scaleBand：有助于创建条形图，同时考虑每个条形之间的间隔。域被指定为一组值（每个波段一个值），范围为波段的最小和最大范围（例如条形图的总宽度）。

  实际上`scaleBand`会将范围拆分为*n* 个带区（其中*n*是域数组中的值的数量），并考虑在指定间隔的情况下计算条形图的位置和宽度。

```
let bandScale = d3.scaleBand()
  .domain(['Mon', 'Tue', 'Wed', 'Thu', 'Fri'])
  .range([0, 200]);

bandScale('Mon'); // returns 0
bandScale('Tue'); // returns 40
bandScale('Fri'); // returns 160
```

可以使用`.bandwidth()`访问每个波段的宽度

可以配置两种类型的间隔：1.`paddingInner`它指定（作为带宽的百分比）每个带之间的填充量;2.`paddingOuter`它指定（作为带宽的百分比）第一个带之前和最后一个带之后的填充量

- scalePoint：创建从一组离散值映射到指定范围内等距的点

```
let pointScale = d3.scalePoint()
  .domain(['Mon', 'Tue', 'Wed', 'Thu', 'Fri'])
  .range([0, 500]);

pointScale('Mon');  // returns 0
pointScale('Tue');  // returns 125
pointScale('Fri');  // returns 500
```

可以使用法`.step()`访问点之间的距离

外部填充(间隔)可以指定为填充与点间距的比率。例如，要使外部填充为点间距的四分之一，请使用值0.25

形状（Shapes）

- **SVG**

形状由 SVG `path` 元素组成。它们中的每一个都有一个 `d` 定义路径形状的属性（路径数据）。

路径数据由 `M0,80L100,100L200,30L300,50L400,40L500,80` 描述路径形状的命令列表组成。每个字母例如 `M` 或 `L` 描述一个命令，例如“移动到”和“画一条线到”。

您可以自己创建路径数据，但 D3 提供了为您完成工作的生成器函数。它们有多种形式：

线	为多段线生成路径数据（通常用于折线图）
区域	生成区域的路径数据（通常用于堆叠折线图和流线图）
堆	从多系列数据生成堆栈数据
弧	生成弧的路径数据（通常用于饼图）
饼	从数据数组生成饼角数据
象	为加号、星号、菱形等符号生成路径数据

- **Line generator**

  在给定**坐标数组的情况下生成路径数据字符串**。使用命令创建线生成器 `d3.line()`：

```
var lineGenerator = d3.line();
var points = [
  [0, 80],
  [100, 100],
  [200, 30],
  [300, 50],
  [400, 40],
  [500, 80]
];

var pathData = lineGenerator(points); // returns
"M0,80L100,100L200,30L300,50L400,40L500,80"

d3.select('path')
  .attr('d', pathData);
```

  - **.x and .y methods**：默认情况下，每个数组元素代表一个由二维数组（例如 `[0, 100]`）定义的坐标。但是，可以使用 `.x` 和 `.y` 指定生成器来解释每个数组元素。

```
var data = [
  {value: 10},
  {value: 50},
  {value: 30},
  {value: 40},
```

```
  {value: 20},
  {value: 70},
  {value: 50}
];

var xScale = d3.scaleLinear().domain([0, 6]).range([0, 600]);
var yScale = d3.scaleLinear().domain([0, 80]).range([150, 0]);

lineGenerator
  .x(function(d, i) {
    return xScale(i);
  })
  .y(function(d) {
    return yScale(d.value);
  });
```

x 坐标是使用应用于数组索引的线性比例函数设置的。这导致在 x 方向上等距的点。y 坐标是使用应用于`value`属性的线性比例设置的

- **.defined()**：可以配置丢失数据时的行为。传入一个函数，如果数据定义明确`true`，则返回该函数。如果函数返回`false`，生成器将跳过它：

```
var points = [
  [0, 80],
  [100, 100],
  null,
  [300, 50],
  [400, 40],
  [500, 80]
];

lineGenerator
  .defined(function(d) {
    return d !== null;
  });
```

- **.curve()**：可以配置点的插值方式。尽管有许多不同的曲线类型可用，但它们可以分为两个阵营：通过点的曲线（`curveLinear`、`curveCardinal`、`curveCatmullRom`、`curveMonotone`和）和不通过点的曲线（`curveNatural`和）。`curveStep`、`curveBasis`、`curveBundle`。有关更多信息，请参阅曲线资源管理器。

```
var lineGenerator = d3.line()
  .curve(d3.curveCardinal);
```

- **Rendering to canvas**：默认情况下，形状生成器输出 SVG 路径数据。但是，可以使用`.context()`函数绘制到画布(canvas)元素上。

```
var context = d3.select('canvas').node().getContext('2d');

lineGenerator.context(context);

context.strokeStyle = '#999';
context.beginPath();
lineGenerator(points);
context.stroke();
```

- **Radial line**：与线生成器类似，但点通过**角度**（从 12 点钟方向顺时针工作）和**半径**转换，而不是`x`和`y`

```
var radialLineGenerator = d3.radialLine();

var points = [
  [0, 80],
  [Math.PI * 0.25, 80],
  [Math.PI * 0.5, 30],
  [Math.PI * 0.75, 80],
  [Math.PI, 80],
  [Math.PI * 1.25, 80],
  [Math.PI * 1.5, 80],
  [Math.PI * 1.75, 80],
  [Math.PI * 2, 80]
];

var pathData = radialLineGenerator(points);
```

`radialLine`生成器还具有方法`.angle`， `.radius`,可以将函数传递给这些方法。

```
var points = [
  {a: 0, r: 80},
  {a: Math.PI * 0.25, r: 80},
  {a: Math.PI * 0.5, r: 30},
  {a: Math.PI * 0.75, r: 80},
  ...
];

radialLineGenerator
  .angle(function(d) {
    return d.a;
  })
  .radius(function(d) {
    return d.r;
  });

var pathData = radialLineGenerator(points);
```

- **Area generator**：输出定义两条线之间区域的路径数据。默认情况下，它生成`y=0`由点数组定义的多段线之间的区域。与线生成器一样，可以使用`.curve()`、`.defined()`、`.context()`。

```
var areaGenerator = d3.area();

var points = [
  [0, 80],
  [100, 100],
  [200, 30],
  [300, 50],
  [400, 40],
  [500, 80]
];

var pathData = areaGenerator(points);
```

可以使用以下方法配置基线`.y0`：

```
areaGenerator.y0(150);
```

还可以将函数传递给`.y0`and`.y1`：

```
var yScale = d3.scaleLinear().domain([0, 100]).range([200, 0]);

var points = [
  {x: 0, low: 30, high: 80},
  {x: 100, low: 80, high: 100},
  {x: 200, low: 20, high: 30},
  {x: 300, low: 20, high: 50},
  {x: 400, low: 10, high: 40},
  {x: 500, low: 50, high: 80}
];

areaGenerator
  .x(function(d) {
    return d.x;
  })
  .y0(function(d) {
    return yScale(d.low);
  })
  .y1(function(d) {
    return yScale(d.high);
  });
```

  - **Radial area**：与区域生成器类似，但点由**角度**（从 12 点钟方向顺时针工作）和**半径**转换，而不是`x`和`y`：

```
var points = [
  {angle: 0, r0: 30, r1: 80},
  {angle: Math.PI * 0.25, r0: 30, r1: 70},
  {angle: Math.PI * 0.5, r0: 30, r1: 80},
  {angle: Math.PI * 0.75, r0: 30, r1: 70},
  {angle: Math.PI, r0: 30, r1: 80},
  {angle: Math.PI * 1.25, r0: 30, r1: 70},
  {angle: Math.PI * 1.5, r0: 30, r1: 80},
  {angle: Math.PI * 1.75, r0: 30, r1: 70},
  {angle: Math.PI * 2, r0: 30, r1: 80}
];

var radialAreaGenerator = d3.radialArea()
  .angle(function(d) {
    return d.angle;
  })
  .innerRadius(function(d) {
    return d.r0;
  })
  .outerRadius(function(d) {
    return d.r1;
  });
```

- **Stack generator**：堆栈生成器接受一个对象数组并为每个对象属性生成一个数组。每个数组包含每个数据点的**下限值和上限值**。计算下限值和上限值，以便每个系列都堆叠在前一个系列的顶部。

```
//在这个例子中，我们有一个对象数组。使用d3.stack. 我们使用它的.keys方法来传递我们想要堆叠的属性键。在这种情况下，我们正在堆叠apricots,blueberries和cherries：


var data = [
  {day: 'Mon', apricots: 120, blueberries: 180, cherries: 100},
  {day: 'Tue', apricots: 60,  blueberries: 185, cherries: 105},
  {day: 'Wed', apricots: 100, blueberries: 215, cherries: 110},
  {day: 'Thu', apricots: 80,  blueberries: 230, cherries: 105},
  {day: 'Fri', apricots: 120, blueberries: 240, cherries: 105}
];

var stack = d3.stack()
  .keys(['apricots', 'blueberries', 'cherries']);

var stackedSeries = stack(data);

// stackedSeries = [
//   [ [0, 120],   [0, 60],   [0, 100],    [0, 80],    [0, 120] ],   // Apricots
//   [ [120, 300], [60, 245], [100, 315],  [80, 310],  [120, 360] ], // Blueberries
//   [ [300, 400], [245, 350], [315, 425], [310, 415], [360, 465] ]  // Cherries
// ]
```

堆栈生成器输出的数据可以随心所欲地使用，但通常它会用于生成堆积条形图，或与面积生成器结合使用时，堆叠折线图

- **.order()**：可以使用`.order()`配置堆叠系列的顺序

```
stack.order(d3.stackOrderInsideOut);
```

将每个列相加，然后根据选择的顺序进行排序。可能的顺序是：

stackOrderNone	**（默认）与 .keys() 中指定的顺序相同的系列**
stackOrderAscending	最小的列在底部
stackOrderDescending	最大的列在底部
stackOrderInsideOut	中间最大列
stackOrderReverse	stackOrderNone 的反转

- **.offset()**：默认情况下，堆叠系列的基线为零。但是，可以配置堆栈生成器的偏移量以实现不同的效果。

```
stack.offset(d3.stackOffsetExpand);
```

可用的偏移量是：

stackOffsetNone	**（默认）无偏移**
stackOffset展开	系列之和归一化（值为 1）
stackOffsetSilhouette	堆栈中心位于 y=0
stackOffsetWiggle	层的摆动最小化（通常用于流图）

- **Arc generator**：弧生成器根据角度和半径值生成路径数据。然后可以传递一个包含`startAngle`、`endAngle`和`innerRadius`、`outerRadius`对象来生成路径数据

```
var arcGenerator = d3.arc();

var pathData = arcGenerator({
  startAngle: 0,
  endAngle: 0.25 * Math.PI,
  innerRadius: 50,
  outerRadius: 100
});

// pathData is "M6.123233995736766e-
15,-100A100,100,0,0,1,70.71067811865476,-70.710678
// 11865474L35.35533905932738,-35.35533905932737A50,50,0,0,0,3.061616997868383e-
15,-50Z"
```

- **Configuration**：可以配置`innerRadius`，`outerRadius`，`startAngle`，`endAngle`以便您不必每次都传递它们

```
arcGenerator
  .innerRadius(20)
  .outerRadius(100);

pathData = arcGenerator({
  startAngle: 0,
  endAngle: 0.25 * Math.PI
});

// pathData is "M6.123233995736766e-
15,-100A100,100,0,0,1,70.71067811865476,-70.71067811
// 865474L14.142135623730951,-14.14213562373095A20,20,0,0,0,1.2246467991473533e-
15,-20Z"
```

还可以配置圆角半径（`cornerRadius`）和弧段之间的填充（`padAngle`和`padRadius`）

```
arcGenerator
  .padAngle(.02)
  .padRadius(100)
  .cornerRadius(4);
```

弧形填充有两个参数`padAngle`，`padRadius`。 当它们相乘时，定义了相邻段之间的距离。因此，在上面的示例中，填充距离为`0.02 * 100 = 2`。请注意，计算填充以保持（在可能的情况下）平行的段边界。

- **Accessor functions**：可以为`startAngle`、`endAngle`和定义访问器函数`innerRadius`、`outerRadius`。

```
arcGenerator
  .startAngle(function(d) {
    return d.startAngleOfMyArc;
  })
  .endAngle(function(d) {
    return d.endAngleOfMyArc;
  });

arcGenerator({
  startAngleOfMyArc: 0,
  endAngleOfMyArc: 0.25 * Math.PI
});
```

  - **Centroid**：计算圆弧的质心有时很有用，例如在定位标签时，`.centroid()`可以做到这一点

```
arcGenerator.centroid({
  startAngle: 0,
  endAngle: 0.25 * Math.PI
});
// returns [22.96100594190539, -55.43277195067721]
```

- **Pie generator**：给定一个数据数组，饼图生成器将输出一个对象数组，其中包含由**起始**角度和**结束角度**增加的原始数据，然后，可以使用弧生成器来创建路径字符串

```
var pieGenerator = d3.pie();
var data = [10, 40, 30, 20, 60, 80];
var arcData = pieGenerator(data);

// arcData is an array of objects: [
//   {
//     data: 10,
//     endAngle: 6.28...,
//     index: 5,
//     padAngle: 0,
//     startAngle: 6.02...,
//     value: 10
//   },
//   ...
// ]

var arcGenerator = d3.arc()
  .innerRadius(20)
  .outerRadius(100);

d3.select('g')
  .selectAll('path')
  .data(arcData)
  .enter()
```

```
  .append('path')
  .attr('d', arcGenerator);
```

饼图生成器有许多配置函数，包括`.startAngle()`、`.endAngle()`、`.sort()`、`.padAngle()`。`.startAngle()`和`.endAngle()`配置饼图的起止角度。例如，这允许创建半圆形饼图：

```
var pieGenerator = d3.pie()
  .startAngle(-0.5 * Math.PI)
  .endAngle(0.5 * Math.PI);
```

可以使用`.sort`更改排序顺序

```
var pieGenerator = d3.pie()
  .value(function(d) {return d.quantity;})
  .sort(function(a, b) {
    return a.name.localeCompare(b.name);
  });

var fruits = [
  {name: 'Apples', quantity: 20},
  {name: 'Bananas', quantity: 40},
  {name: 'Cherries', quantity: 50},
  {name: 'Damsons', quantity: 10},
  {name: 'Elderberries', quantity: 30},
];
```

`padAngle()`指定相邻段之间的角度填充

- **Symbols**：为数据可视化中常用的符号生成路径数据(d3.symbolCircle、d3.symbolCross、d3.symbolDiamond、d3.symbolSquare、d3.symbolStar、d3.symbolTriangle、d3.symbolWye)

```
var symbolGenerator = d3.symbol()
  .type(d3.symbolStar)
  .size(80);

var pathData = symbolGenerator();

d3.select('path')
  .attr('d', pathData);
```

轴（Axes）

- **创建轴**

  创建轴需要：1.包含轴的 SVG 元素（通常是`g`元素）；2.D3比例函数。当使用 D3 比例函数定义轴时，比例域确定最小和最大刻度值，范围确定轴的长度。

要创建轴：1.创建一个`d3.axisBottom`、`d3.axisTop`、`d3.axisLeft`、`d3.axisRight`函数；2.选择容器元素并将轴生成器传递到`.call`

```
<svg width="600" height="100">
  <g transform="translate(20, 50)"></g>
</svg>
```

```
let scale = d3.scaleLinear().domain([0, 100]).range([0, 500]);

let axis = d3.axisBottom(scale);

d3.select('svg g')
  .call(axis);
```

- **轴方向**：`d3.axisBottom`, `d3.axisTop`, `d3.axisLeft`和`d3.axisRight`分别用于生成适合图表底部、顶部、左侧和右侧的轴
- **Scale类型**：可以传入任何具有数字输出的比例函数。这包括`scaleLinear`、`scaleSqrt`、`scaleLog`、`scaleTime`、`scaleBand`、`scalePoint`
- **过渡**：如果刻度的域（domain）发生变化，可以通过再次调用`.call(axis)`来更新轴。还可以调用`.transition`使轴有动画

```
d3.select('svg g')
  .transition()
  .call(axis);
```

- **轴配置**：可以通过以下方式配置轴：1.指定刻度数或指定刻度值；2.指定刻度标签的格式（例如，添加百分号）；3.指定刻度大小
  - **刻度数**：.ticks方法指定轴有多少刻度

```
let scale = d3.scaleLinear().domain([0, 100]).range([0, 500]);

let axis = d3.axisBottom(scale);

axis.ticks(20);

d3.select('svg g')
  .call(axis);
```

  - **刻度值**：通过将刻度值数组传递给`.tickValues`方法来指定轴的刻度值

```
let scale = d3.scaleLinear().domain([0, 100]).range([0, 500]);

let axis = d3.axisBottom(scale);

axis.tickValues([0, 25, 50, 75, 100]);

d3.select('svg g')
  .call(axis);
```

- **刻度标签格式**：可以通过两种方式格式化刻度标签：1.使用该`.ticks`方法并传入一个格式字符串作为第二个参数；2.将格式化函数传递给`.tickFormat`方法。该函数接受一个值并输出一个格式化的值。

```
let scale = d3.scaleLinear().domain([0, 100]).range([0, 500]);

let axis = d3.axisBottom(scale);

axis.ticks(4, "$.2f");

d3.select('svg g')
  .call(axis);
```

```
let scale = d3.scaleLinear().domain([0, 100]).range([0, 500]);

let axis = d3.axisBottom(scale);

axis.ticks(4)
  .tickFormat(function(d) {
    return d + "%";
  });

d3.select('svg g')
  .call(axis);
```

格式字符串功能强大，在官方文档的 d3-format 部分有深入描述。

- **刻度尺寸**：使用`.tickSize`方法设置；还可以使用`.tickPadding`设置刻度和刻度标签之间的距离

```
let scale = d3.scaleLinear().domain([0, 100]).range([0, 500]);

let axis = d3.axisBottom(scale)
  .tickPadding(10)
  .tickSize(10);

d3.select('svg g')
  .call(axis);
```

层次结构（Hierarchies）

分析或可视化数据时的一种常用技术是将数据组织成**组**。(分组)

可以将层次结构视为树状结构，其中根项目（或“节点”）拆分为顶级组。每个顶级组拆分为二级组，依此类推。最顶层的项目（或**节点**）称为**根节点**。最底部的项目称为**叶子**或**叶子节点**。

有几种方法可以可视化分层数据，包括**树**、**树图**、**压缩圆图**和**旭日图**。

- **从数据数组创建层次结构**：使用 D3 的`.rollup`函数按任何分类属性对数据进行分组。第一个参数是要分组的数组；第二个参数是一个**reduce**函数，这是一个接受值数组并输出单个值的函数；其余参数是指定要分组的属性的函数。可以使用`.get`检查返回的map对象

```
function sumWorldwideGross(group) {
  return d3.sum(group, function(d) {
    return d.Worldwide_Gross;
  });
}

let groups = d3.rollup(data,
                       sumWorldwideGross,
                       function(d) { return d.Distributor; },
                       function(d) { return d.Genre; }
                      );

// Get Sony Pictures
groups.get('Sony Pictures');  // {"Comedy" => 22498520, "Action" => 315268353, "Drama"
=> 93905424}

// Get Drama within Sony Pictures
groups.get('Sony Pictures').get('Drama');  // 93905424
```

- **d3.层次结构**：通过调用`d3.hierarchy`并传入生成的map对象来创建的`d3.rollup`

```
function sumWorldwideGross(group) {
  return d3.sum(group, function(d) {
    return d.Worldwide_Gross;
  });
}

let groups = d3.rollup(data,
                       sumWorldwideGross,
                       function(d) { return d.Distributor; },
                       function(d) { return d.Genre; }
                      );

let root = d3.hierarchy(groups);
```

`d3.hierarchy`输出的是一个嵌套对象，类似于：

```
{
  data: [undefined, Map(3)],
  children: [
    {
      data: ["Sony Pictures", Map(3)],
      children: [...],
      depth: 1,
      height: 1,
      parent: {...} // this item's parent node
    }.
    {
      data: ["Walt Disney Pictures", Map(2)],
      children: [...],
      depth: 1,
      height: 1,
      parent: {...} // this item's parent node
    }.
    {
      data: ["Warner Bros.", Map(3)],
      children: [...],
      depth: 1,
      height: 1,
      parent: {...} // this item's parent node
    }
  ],
  depth: 0,
  height: 2,
  parent: null
}
```

层次结构中的每个项目（或**节点**）都有属性:`data`, `children`, `depth`, `height` and `parent`.

`data`是传入的关联项目中的map或对象。通常，不需要访问该值，因为层次结构通过其`children`和`value`属性使该数据可用。

`children`是一个包含节点子节点的数组。

`depth`和`height`指示层次结构中节点的深度和高度。

`parent`引用节点的父节点。

- **可视化层次结构**
  - **树形布局**：首先使用创建树布局函数`d3.tree()`；可以命令配置树的大小`.size`；然后，可以调用`treeLayout`，传入定义的层次结构对象`root`

```
var treeLayout = d3.tree();

treeLayout.size([400, 200]);

treeLayout(root);
```

这将在`root`的每个节点上写入`x`和`y`值。

绘制节点：

1. 用于`root.descendants()`获取所有节点的数组
2. 将此数组加入圆（或任何其他类型的 SVG 元素）
3. 使用`x`和`y`定位圆圈

要绘制链接：

1. 用于`root.links()`获取所有链接的数组
2. 将数组连接到行（或路径）元素
3. 使用`x`和`y`链接的`source`和`target`属性来定位线

```
// Nodes
d3.select('svg g.nodes')
  .selectAll('circle.node')
  .data(root.descendants())
  .join('circle')
  .classed('node', true)
  .attr('cx', function(d) {return d.x;})
  .attr('cy', function(d) {return d.y;})
  .attr('r', 4);

// Links
d3.select('svg g.links')
  .selectAll('line.link')
  .data(root.links())
  .join('line')
  .classed('link', true)
  .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;});
```

- **集群布局**：与`tree`布局非常相似，主要区别在于**所有叶节点都放置在相同的深度**。

```
var clusterLayout = d3.cluster()
  .size([400, 200]);

var root = d3.hierarchy(data);

clusterLayout(root);
```

- **树状图布局**：用于直观地表示**每个项目都有关联值的**层次结构。

  通过调用创建树图布局函数`d3.treemap()`;可以配置布局;在将此布局应用于层次结构之前，**必须在层次结构上**运行`.sum()`。调用`treemapLayout`，传入`root`之前定义的层次结构对象.

```
var treemapLayout = d3.treemap();

treemapLayout
  .size([400, 200])
  .paddingOuter(10);

root.sum(function(d) {
  return d.value;
});

treemapLayout(root);
```

树形图布局函数向每个节点添加 4 个属性`x0`、`x1`、`y0`、`y1`。它们指定树形图中每个矩形的尺寸。

```
d3.select('svg g')
  .selectAll('rect')
  .data(root.descendants())
  .join('rect')
  .attr('x', function(d) { return d.x0; })
  .attr('y', function(d) { return d.y0; })
  .attr('width', function(d) { return d.x1 - d.x0; })
  .attr('height', function(d) { return d.y1 - d.y0; })
```

如果想在每个矩形中添加标签，可以将`g`元素加入数组并添加`rect`和`text`元素到每个`g`：

```
var nodes = d3.select('svg g')
  .selectAll('g')
  .data(rootNode.descendants())
  .join('g')
  .attr('transform', function(d) {return 'translate(' + [d.x0, d.y0] + ')'})

nodes
  .append('rect')
  .attr('width', function(d) { return d.x1 - d.x0; })
  .attr('height', function(d) { return d.y1 - d.y0; })

nodes
  .append('text')
  .attr('dx', 4)
  .attr('dy', 14)
  .text(function(d) {
    return d.data.name;
  })
```

`treemap`可以通过多种方式配置布局：

1. 可以使用设置节点子节点周围的填充`.paddingOuter`
2. 兄弟节点之间的填充可以使用`.paddingInner`
3. 可以使用同时设置外部和内部填充`.padding`
4. 外部填充也可以使用,`.paddingTop`、`.paddingBottom`、`.paddingLeft`、`.paddingRight`进行微调

树形图有不止一种排列矩形的策略。D3 有一些内置的：`treemapBinary`，`treemapDice`,`treemapSlice`、`treemapSliceDice`、`treemapSquarify`

- **包布局**：类似于树布局，但用圆圈表示节点。

```
var packLayout = d3.pack();

packLayout.size([300, 300]);

rootNode.sum(function(d) {
  return d.value;
});

packLayout(rootNode);

d3.select('svg g')
  .selectAll('circle')
  .data(rootNode.descendants())
  .join('circle')
  .attr('cx', function(d) { return d.x; })
  .attr('cy', function(d) { return d.y; })
  .attr('r', function(d) { return d.r; })
```

可以通过`g`为每个后代创建元素来添加标签：

```
var nodes = d3.select('svg g')
  .selectAll('g')
  .data(rootNode.descendants())
  .join('g')
  .attr('transform', function(d) {return 'translate(' + [d.x, d.y] + ')'})

nodes
  .append('circle')
  .attr('r', function(d) { return d.r; })

nodes
  .append('text')
  .attr('dy', 4)
  .text(function(d) {
    return d.children === undefined ? d.data.name : '';
  })
```

可以使用以下方式配置每个圆圈周围的填充

```
packLayout.padding(10)
```

- **分区布局**：将一个矩形空间细分为层，每个层代表层次结构中的一个层。对于层中的每个节点，每一层进一步细分

```
var partitionLayout = d3.partition();

partitionLayout.size([400, 200]);

rootNode.sum(function(d) {
  return d.value;
});

partitionLayout(rootNode);

d3.select('svg g')
  .selectAll('rect')
  .data(rootNode.descendants())
  .join('rect')
  .attr('x', function(d) { return d.x0; })
  .attr('y', function(d) { return d.y0; })
  .attr('width', function(d) { return d.x1 - d.x0; })
  .attr('height', function(d) { return d.y1 - d.y0; });

partitionLayout.padding(2);
```

如果您想更改分区布局的方向，以便图层从左到右运行，您可以在定义元素时交换`x0`、`x1`和交换：`y0`、`y1`

```
 .attr('x', function(d) { return d.y0; })
  .attr('y', function(d) { return d.x0; })
  .attr('width', function(d) { return d.y1 - d.y0; })
  .attr('height', function(d) { return d.x1 - d.x0; });
```

还可以将`x`尺寸映射到旋转角度和`y`半径以创建旭日图：

```
var data = {
  "name": "A1",
  "children": [
    {
      "name": "B1",
      "children": [
        {
```

```
          "name": "C1",
          "value": 100
        },
        {
          "name": "C2",
          "value": 300
        },
        {
          "name": "C3",
          "value": 200
        }
      ]
    },
    {
      "name": "B2",
      "value": 200
    }
  ]
};

var radius = 150;

var partitionLayout = d3.partition()
  .size([2 * Math.PI, radius]);

var arcGenerator = d3.arc()
  .startAngle(function(d) { return d.x0; })
  .endAngle(function(d) { return d.x1; })
  .innerRadius(function(d) { return d.y0; })
  .outerRadius(function(d) { return d.y1; });

var rootNode = d3.hierarchy(data)

rootNode.sum(function(d) {
  return d.value;
});

partitionLayout(rootNode);

d3.select('svg g')
  .selectAll('path')
  .data(rootNode.descendants())
  .join('path')
  .attr('d', arcGenerator);
```

弦（Chords）

- **弦图**

弦图可视化一组节点之间的链接（或流），其中每个流都有一个数值。

数据需要采用 n*n 矩阵的形式（其中 n 是项目数）;第一行代表从第一个项目到第一个、第二个和第三个项目.....的流量;

使用`chord()`创建布局;

使用`.padAngle()`（设置相邻组之间的角度）、`.sortGroups()`（指定组的顺序）、`.sortSubgroups()`（在每个组内排序）和`.sortChords()`（确定弦图的 z 顺序）;

返回一个数组。数组的每个元素都是一个具有`source`和`target`属性的对象。每个`source`和`target`具有将定义每个弦形状的属性`startAngle`和`endAngle`

我们使用`ribbon`形状生成器将弦属性转换为路径数据。

```
var data = [
  [10, 20, 30],
  [40, 60, 80],
  [100, 200, 300]
];

var chordGenerator = d3.chord();

var chords = chordGenerator(data);

var ribbonGenerator = d3.ribbon().radius(200);

d3.select('g')
  .selectAll('path')
  .data(chords)
  .join('path')
  .attr('d', ribbonGenerator)
```

力（Force）

D3 的力布局使用基于**物理的模拟器**来定位视觉元素。可以在元素之间设置力，例如：1.所有元素都可以配置为相互排斥;2.元素可以被吸引到重心;3.链接的元素可以设置为固定距离（例如用于网络可视化）;4.元素可以配置为避免相互交叉（碰撞检测）

力布局比其他 D3 布局需要更多的计算量（通常需要几秒钟的时间），并且解决方案是以**逐步（迭代）的**方式计算的。通常 SVG/HTML 元素的位置会随着模拟的迭代而更新，这就是为什么我们看到圆圈挤在一起的原因。

- **设置力模拟**:

设置力模拟有 4 个步骤:

1. 创建对象数组
2. 调用`forceSimulation`，传入对象数组

3. 向系统添加一个或多个力函数（例如`forceManyBody`，`forceCenter`，`forceCollide`）
4. 设置回调函数以在每个刻度后更新元素位置

```
var width = 300, height = 300
var nodes = [{}, {}, {}, {}, {}]

var simulation = d3.forceSimulation(nodes)
  .force('charge', d3.forceManyBody())
  .force('center', d3.forceCenter(width / 2, height / 2))
  .on('tick', ticked);

function ticked() {
  var u = d3.select('svg')
    .selectAll('circle')
    .data(nodes)
    .join('circle')
    .attr('r', 5)
    .attr('cx', function(d) {
      return d.x
    })
    .attr('cy', function(d) {
      return d.y
    });
}
```

- **力中心(forceCenter)**：将元素作为一个整体围绕着中心点居中

```
//可以用中心位置初始化
d3.forceCenter(100, 100)

//或使用配置功能.x()和.y()
d3.forceCenter().x(100).y(100)

//将其添加到系统中
simulation.force('center', d3.forceCenter(100, 100))
```

- **forceManyBody**：使所有元素相互吸引或排斥。可以设置吸引或排斥的强度。`.strength()`正值将导致元素相互吸引，而负值将导致元素相互排斥。默认值为`-30`。

```
simulation.force('charge', d3.forceManyBody().strength(-20))
```

- **力碰撞(forceCollide)**：用于阻止圆形元素重叠。在将圆形"聚集"在一起时特别有用。此函数的第一个参数`d`是连接数据，您可以从中得出半径。

```
var numNodes = 100
var nodes = d3.range(numNodes).map(function(d) {
  return {radius: Math.random() * 25}
})

var simulation = d3.forceSimulation(nodes)
  .force('charge', d3.forceManyBody().strength(5))
  .force('center', d3.forceCenter(width / 2, height / 2))
  .force('collision', d3.forceCollide().radius(function(d) {
    return d.radius
  }))
```

- **forceX 和 forceY**：导致元素**被吸引到**指定的位置。可以对所有元素使用一个中心，也可以在每个元素的基础上施加力。

```
var xCenter = [100, 300, 500];

var simulation = d3.forceSimulation(nodes)
  .force('charge', d3.forceManyBody().strength(5))
  .force('x', d3.forceX().x(function(d) {
    return xCenter[d.category];
  }))
  .force('collision', d3.forceCollide().radius(function(d) {
    return d.radius;
  }));
```

可以使用`forceX`或`forceY`沿轴定位元素

```
var simulation = d3.forceSimulation(nodes)
  .force('charge', d3.forceManyBody().strength(5))
  .force('x', d3.forceX().x(function(d) {
    return xScale(d.value);
  }))
  .force('y', d3.forceY().y(function(d) {
    return 0;
  }))
  .force('collision', d3.forceCollide().radius(function(d) {
    return d.radius;
  }));
```

- **力链接(forceLink)**：将链接的元素推到一个**固定的距离**。它需要一**组链接**来指定您希望将哪些元素链接在一起。每个链接对象指定一个源元素和目标元素，其中值是元素的数组索引：

```
var links = [
  {source: 0, target: 1},
  {source: 0, target: 2},
  {source: 0, target: 3},
```

```
  {source: 1, target: 6},
  {source: 3, target: 4},
  {source: 3, target: 7},
  {source: 4, target: 5},
  {source: 4, target: 7}
]

var simulation = d3.forceSimulation(nodes)
  .force('charge', d3.forceManyBody().strength(-100))
  .force('center', d3.forceCenter(width / 2, height / 2))
  .force('link', d3.forceLink().links(links));

//可以使用.distance()（默认值为 30）和配置链接元素的距离和强度.strength()。
```

地理（Geographic）

D3 的方法不同于所谓的栅格方法，例如Leaflet和 Google Maps。这些**预渲染地图特征为图像瓦片**，它们从网络服务器加载并在浏览器中拼凑在一起形成地图。

通常，D3以 GeoJSON 的形式请求**矢量地理信息，并在浏览器中将其呈现为 SVG 或 Canvas**。

光栅地图通常看起来更像传统的印刷地图，其中可以显示很多细节（例如地名、道路、河流等），而不会影响性能。但是，使用矢量方法更容易实现动画和交互等动态内容。（将这两种方法结合起来也很常见。）

- **D3 映射概念**：三个关键概念

1. **GeoJSON**（用于指定地理数据的基于 JSON 的格式）
2. **投影**（从纬度/经度坐标转换为 x 和 y 坐标的函数）
3. **地理路径生成器**（将 GeoJSON 形状转换为 SVG 或 Canvas 路径的函数）

GeoJSON 是一种使用 JSON 格式表示地理数据的标准，完整的规范位于geojson.org。每个要素都由**几何**（国家的简单多边形和廷巴克图的一个点）和**属性**组成。D3 在渲染 GeoJSON 时会处理大部分细节，因此只需对 GeoJSON 有基本的了解即可开始使用 D3 映射。

投影函数采用经度和纬度坐标（以数组的形式`[lon, lat]`）并将其转换为 x 和 y 坐标。投影数学可以变得相当复杂，但幸运的是 D3 提供了大量的投影函数。

地理路径生成器是一个接受 GeoJSON 对象并将其转换为 SVG 路径字符串的函数。可以使用该方法创建生成器`.geoPath`并使用投影功能对其进行配置。

```
let geoJson = {
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "name": "Africa"
      },
      "geometry": {
        "type": "Polygon",
        "coordinates": [[[-6, 36], [33, 30], ... , [-6, 36]]]
```

```
      }
    },
    ...
  ]
}

let projection = d3.geoEquirectangular();

let geoGenerator = d3.geoPath()
  .projection(projection);

// Join the FeatureCollection's features array to path elements
let u = d3.select('#content g.map')
  .selectAll('path')
  .data(geojson.features)
  .join('path')
  .attr('d', geoGenerator);
```

- **地理JSON**：

GeoJSON 是一种基于 JSON 的结构，用于指定地理数据。通常，它是使用mapshaper、ogr2ogr、shp2json或QGIS等工具从 shapefile 数据（一种广泛用于 GIS 领域的地理空间矢量数据格式）转换而来的。

shapefile 的一个来源是Natural Earth，如果开始，我建议尝试使用mapshaper来导入 shapefile 并导出为 GeoJSON。它还可以按属性过滤（例如，如果您想按大陆过滤国家）。

可以在不详细了解 GeoJSON 规范的情况下创建地图，因为诸如 mapshaper 和 D3 之类的工具可以很好地抽象出细节。

到目前为止，我们已经在示例文件中嵌入了 GeoJSON 对象。实际上，GeoJSON 将位于一个单独的文件中，并使用 ajax 请求加载。但在本章的其余部分，我们将使用以下方式加载 GeoJSON 文件：

```
d3.json('ne_110m_land.json', function(err, json) {
  createMap(json);
})
```

- **预测**：将球体（例如地球）上的点转换（或“投影”）到平面（例如屏幕）上的点的方法有很多。

  简而言之，没有完美的投影，因为每个投影都会扭曲形状、面积、距离和/或方向。选择投影是选择您不想扭曲的属性并接受其他属性会失真的情况（或选择努力平衡方法的投影）。例如，如果准确表示国家的大小很重要，那么选择一个努力保留面积的投影（可能以形状、距离和方向为代价）。

  D3 的核心预测：

  - `geoAzimuthalEqualArea`
  - `geoAzimuthalEquidistant`
  - `geoGnomonic`
  - `geoOrthographic`
  - `geoStereographic`
  - `geoAlbers`
  - `geoConicConformal`

- `geoConicEqualArea`
- `geoConicEquidistant`
- `geoEquirectangular`
- `geoMercator`
- `geoTransverseMercator`

核心投影具有用于设置以下参数的配置功能：

规模	投影的比例因子
中央	投影中心[经度、纬度]
翻译	投影中心的像素 [x,y] 位置
旋转	投影的旋转 [lambda, phi, gamma]（或 [yaw, pitch, roll]）

每个参数的确切含义取决于每个投影背后的数学，但从广义上讲：

- scale 指定投影的**比例因子**。数字越大，地图越大。
- center 指定**投影的中心**（带`[lon, lat]`数组）
- translate 指定**投影中心在屏幕上的位置**（带`[x, y]`数组）
- rotate 指定**投影的旋转**（带`[λ, φ, γ]`数组），其中参数分别对应 yaw、pitch 和 roll

可以使用投影的方法将像素坐标转换为经度/纬度数组

```
let projection = d3.geoAzimuthalEqualArea();

projection( [-3.0026, 16.7666] )
// returns [473.67353385539417, 213.6120079887163]

projection.invert( [473.67353385539417, 213.6120079887163] )
// returns [-3.0026, 16.766]
```

`.fitExtent()`方法设置投影的**比例**和**平移**，以使几何形状适合给定的边界框

```
//参数：边界框的左上点 ( [x, y]) 和边界框的大小 ( [width, height])。

projection.fitExtent([[0, 0], [900, 500]], geojson);
```

- **地理路径生成器**
  - 渲染 SVG：1.将特征数组加入 SVG`path`元素；2.使用地理路径生成器更新每个`path`元素的属性`d`

```
let geoJson = {
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "name": "Africa"
```

```
      },
      "geometry": {
        "type": "Polygon",
        "coordinates": [[[-6, 36], [33, 30], ... , [-6, 36]]]
      }
    },
    {
      "type": "Feature",
      "properties": {
        "name": "Australia"
      },
      "geometry": {
        "type": "Polygon",
        "coordinates": [[[143, -11], [153, -28], ... , [143, -11]]]
      }
    },
    {
      "type": "Feature",
      "properties": {
        "name": "Timbuktu"
      },
      "geometry": {
        "type": "Point",
        "coordinates": [-3.0026, 16.7666]
      }
    }
  ]
}

let projection = d3.geoEquirectangular();

let geoGenerator = d3.geoPath()
  .projection(projection);

// Join the FeatureCollection's features array to path elements
let u = d3.select('#content g.map')
  .selectAll('path')
  .data(geojson.features)
  .join('path')
  .attr('d', geoGenerator);
```

- 渲染到画布：1.将`canvas`DOM 元素传递给生成器的`context`方法;2.开始一个画布路径（使用`context.beginPath()`）并调用`geoGenerator`它将产生必要的画布调用

```
let context = d3.select('#content canvas')
  .node()
  .getContext('2d');

let geoGenerator = d3.geoPath()
  .projection(projection)
  .context(context);

context.beginPath();
geoGenerator({type: 'FeatureCollection', features: geojson.features})
context.stroke();
```

- 直线和圆弧：地理路径生成器可以区分多边形（通常用于地理区域）和点（通常用于经度/纬度位置）特征。它将多边形渲染为线段，将点渲染为弧。

```
let geoGenerator = d3.geoPath()
  .pointRadius(5)
  .projection(projection);
```

- 路径几何：地理路径生成器还可用于计算投影 GeoJSON 特征的**面积**（以像素为单位）、**质心**、**边界框**和**路径长度（以像素为单位）**：

```
let feature = geojson.features[0];

// Compute the feature's area (in pixels)
geoGenerator.area(feature);
// returns 30324.86518469876

// Compute the feature's centroid (in pixel co-ordinates)
geoGenerator.centroid(feature);
// returns [266.9510120424504, 127.35819206325564]

// Compute the feature's bounds (in pixel co-ordinates)
geoGenerator.bounds(feature);
// returns [[140.6588054321928, 24.336293856408275], [378.02358370342165,
272.17304763960306]]

// Compute the path length (in pixels)
geoGenerator.measure(feature);
// returns 775.7895349902461
```

- **形状**：可以向地图添加线和/或圆。

```
//线可以作为 LineString 特征添加，并将被投影到大弧中（即穿过地球表面的最短距离）。
geoGenerator({
  type: 'Feature',
  geometry: {
```

```
    type: 'LineString',
    coordinates: [[0.1278, 51.5074], [-74.0059, 40.7128]]
  }
});


//圆形特征可以使用d3.geoCircle(). 这将创建一个圆生成器，该生成器返回一个表示圆的 GeoJSON 对象。
let circleGenerator = d3.geoCircle()
  .center([0.1278, 51.5074])
  .radius(5);

let circle = circleGenerator();
// returns a GeoJSON object representing a circle

geoGenerator(circle);
// returns a path string representing the projected circle


//可以使用d3.graticule(). 这将创建一个格线生成器，该生成器返回一个表示格线的 GeoJSON 对象。
let graticuleGenerator = d3.geoGraticule();

let graticules = graticuleGenerator();
// returns a GeoJSON object representing the graticule

geoGenerator(graticules);
// returns a path string representing the projected graticule
```

- **球面几何**

d3.geoInterpolate()函数接受 0 到 1 之间的输入并在两个`[lon, lat]`位置之间进行插值：

```
let londonLonLat = [0.1278, 51.5074];
let newYorkLonLat = [-74.0059, 40.7128];
let geoInterpolator = d3.geoInterpolate(londonLonLat, newYorkLonLat);

geoInterpolator(0);
// returns [0.1278, 51.5074]

geoInterpolator(0.5);
// returns [-41.182023242967695, 52.41428456719971] (halfway between the two locations)
```

可以使用`d3.geoContains`接受 GeoJSON 功能和`[lon, lat]`数组并返回布尔值来检查鼠标或触摸事件是否发生在要素边界内(SVG渲染情况下有效)

```
d3.geoContains(ukFeature, [0.1278, 51.5074]);
// returns true
```

请求（Requests）

每当网络浏览器希望请求资源时，无论是 HTML 文件、JPEG 图像还是 CSV 文件，它都会使用**HTTP 请求**。通常，您希望请求的数据（或资源）将具有**URL**（统一资源定位器），例如 `https://assets.codepen.io/2814973/my-csv.csv`.

D3 使请求数据相对简单。它处理 HTTP 请求，还可以将传入的数据转换为有用的格式。例如，它可以请求 CSV文件并将其转换为对象数组。

- **请求 CSV 数据**

```
function update(data) {
  d3.select('#content tbody')
    .selectAll('tr')
    .data(data)
    .join('tr')
    .html(function(d) {
      let html = '<tr>';
      html += '<td>' + d.company + '</td>';
      html += '<td>' + d.industry + '</td>';
      html += '<td>' + d.revenue + '</td>';
      html += '<td>' + d.workers + '</td>';
      html += '</tr>';
      return html;
    });
}

d3.csv('https://assets.codepen.io/2814973/Inc5000+Company+List_2014-top250.csv')
  .then(function(data) {
    update(data);
  });
```

`d3.csv`接受 URL 作为其第一个参数并返回一个**Promise**对象。

一个 promise 对象代表一个**异步**操作。异步操作是其结果是未来某个时间的操作。这意味着您的代码可以在发起请求后立即继续运行。当浏览器接收到请求的数据时，promise 就**完成**了，传递给 promise `.then`方法的函数就会被调用。

完成请求后，D3 将传入的 CSV 文件转换为对象数组。每个对象代表一行数据：

```
[
  {
    "rank": "1",
    "workers": "227",
    "company": "Fuhu",
    "state_l": "California",
    "city": "El Segundo",
    "growth": "158956.9106",
    "revenue": "195640000",
    "industry": "Consumer Products & Services"
```

```
  },
  {
    "rank": "2",
    "workers": "191",
    "company": "Quest Nutrition",
    "state_l": "California",
    "city": "El Segundo",
    "growth": "57347.9246",
    "revenue": "82640563",
    "industry": "Food & Beverage"
  },
  {
    "rank": "3",
    "workers": "145",
    "company": "Reliant Asset Management",
    "state_l": "Virginia",
    "city": "Arlington",
    "growth": "55460.1646",
    "revenue": "85076502",
    "industry": "Business Products & Services"
  },
  ...
]
```

行转换：D3 将 CSV 文件中的数字解释为**字符串**。可以在代码中的任何位置将字符串转换为数字，但我建议直接进行转换。这可以通过将函数`d3.csv`作为第二个参数传入来完成。对每一行数据调用该函数，然后返回一个带有任何适当转换的新对象。

```
function convertRow(d) {
  return {
    rank: +d.rank,
    workers: +d.workers,
    name: d.company,
    state: d.state_l,
    city: d.city,
    growth: +d.growth,
    revenue: +d.revenue,
    sector: d.industry
  }
}

d3.csv('https://assets.codepen.io/2814973/Inc5000+Company+List_2014-top250.csv',
convertRow)
  .then(function(data) {
    console.log(data);
  });
```

- **请求 TSV 数据**：TSV 数据是**制表符分隔值**数据，其处理方式与 CSV 类似。用于`d3.tsv`加载 TSV 数据。

- **请求 JSON 数据**：JSON 是一种密切反映 JavaScript 数组和对象的文件格式。它允许嵌套结构，使其优于表格文件格式。

```
function update(data) {
  d3.select('#content tbody')
    .selectAll('tr')
    .data(data)
    .join('tr')
    .html(function(d) {
      let html = '<tr>';
      html += '<td>' + d.name + '</td>';
      html += '<td>' + d.indicator1 + '</td>';
      html += '<td>' + d.indicator2 + '</td>';
      html += '</tr>';
      return html;
    });
}

d3.json('https://assets.codepen.io/2814973/my-json.json')
  .then(function(data) {
    update(data);
  });
```

当 JSON 文件到达时，D3 将其转换为 JavaScript 数组或对象。与 CSV 数据不同，JSON 数据不一定是对象数组，因此`d3.json`不支持行转换函数。

过度（Transitions）

D3 过渡可以在不同**图表状态之间平滑地制作动画**。

- **创建D3过渡**：在要转换的和方法之前添加一个`.transition()`调用：`.attr`、`.style`

```
function update() {
  d3.select('svg')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cy', 50)
    .transition()
    .attr('cx', function(d) {
      return d.x;
    })
    .attr('r', function(d) {
      return d.r;
    })
    .style('fill', function(d) {
      return d.fill;
    });
}
```

- **持续时间和延迟**：可以通过调用`.duration`来更改转换的持续时间。该`.duration`方法接受一个参数，该参数以毫秒为单位指定持续时间；延迟通常用于将选择中的每个元素延迟不同的量。可以通过将**函数**`.delay`传入并将延迟设置为元素索引的倍数来创建**交错**转换

```
d3.select('svg')
  .selectAll('circle')
  .data(data)
  .join('circle')
  .attr('cy', 50)
  .attr('r', 40)
  .transition()
  .duration(2000)
  .attr('cx', function(d) {
    return d;
  });
```

```
d3.select('svg')
  .selectAll('circle')
  .data(data)
  .join('circle')
  .attr('cy', 50)
  .attr('r', 40)
  .transition()
  .delay(function(d, i) {
    return i * 75;
  })
  .attr('cx', function(d) {
    return d;
  });
```

- **缓动函数**：定义了元素在过渡期间的速度变化。例如，一些缓动函数会导致元素快速启动并逐渐变慢。其他人则相反（开始缓慢并加速），或者是定义特殊效果，例如弹跳。D3 有许多内置的缓动函数。

  一般来说，“in”是指运动的开始，“out”是指运动的结束。因此，`easeBounceOut`导致元素在过渡**结束时反弹。**`easeBounceInOut`使元素在过渡的**开始和结束时反弹。**

```
d3.select('svg')
  .selectAll('circle')
  .data(data)
  .join('circle')
  .attr('cy', 50)
  .attr('r', 40)
  .transition()
  .ease(d3.easeBounceOut)
  .attr('cx', function(d) {
    return d;
  });
```

- **链式转换**：可以通过添加多个调用链接在一起`.transition`。每个过渡都将轮流进行。（当第一个过渡结束时，第二个将开始，依此类推。）

```
function update() {
  d3.select('svg')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cy', 50)
    .transition()
    .attr('cx', function(d) {
      return d.x;
    })
    .transition()
    .duration(750)
    .ease(d3.easeBounce)
    .attr('r', function(d) {
      return d.r;
    });
}
```

- **.tween**：自定义元素所采用的路径（例如沿着大圆的圆周）。需要将一个名称（可以是您喜欢的任何名称）和一个函数传递给`.tween`。该函数会被选择中的每个元素调用一次。它必须返回一个**函数**，该函数将在转换的每个步骤中被调用。`t`将0 到 1 之间的值传递给tween函数。（`t`在过渡开始时为 0，在结束时为 1。）

```
let data = [], majorRadius = 100;

function updateData() {
  data = [Math.random() * 2 * Math.PI];
}

function getCurrentAngle(el) {
  // Compute the current angle from the current values of cx and cy
  let x = d3.select(el).attr('cx');
  let y = d3.select(el).attr('cy');
  return Math.atan2(y, x);
}

function update() {
  d3.select('svg g')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('r', 7)
    .transition()
    .tween('circumference', function(d) {
      let currentAngle = getCurrentAngle(this);
      let targetAngle = d;
```

```
      // Create an interpolator function
      let i = d3.interpolate(currentAngle, targetAngle);

      return function(t) {
        let angle = i(t);

        d3.select(this)
          .attr('cx', majorRadius * Math.cos(angle))
          .attr('cy', majorRadius * Math.sin(angle));
      };
    });
}
```

交互（Interaction）

D3 提供了许多模块来帮助您添加交互性，例如缩放、平移和画笔。

- **四叉树（Quadtrees）**：通过在每次移动鼠标时搜索最接近鼠标指针的项目，可以更轻松地挑选小项目。**这可能是一项昂贵的操作，但使用 D3 的四叉树**模块可以提高效率。四叉树是一种树数据结构，它递归地将一个区域划分为越来越小的区域，并且可以使搜索项目更有效。

```
let data = [], width = 600, height = 400, numPoints = 100;

let quadtree = d3.quadtree()
  .x(function(d) {return d.x;})
  .y(function(d) {return d.y;});

let hoveredId;

function updateData() {
  data = [];
  for(let i=0; i<numPoints; i++) {
    data.push({
      id: i,
      x: Math.random() * width,
      y: Math.random() * height,
      r: 1 + Math.random() * 20
    });
  }
}

function handleMousemove(e) {
  let pos = d3.pointer(e, this);
  let d = quadtree.find(pos[0], pos[1], 20);
  hoveredId = d ? d.id : undefined;
  update();
}
```

```
function initEvents() {
  d3.select('svg')
    .on('mousemove', handleMousemove);
}

function updateQuadtree() {
  quadtree.addAll(data);
}

function update() {
  d3.select('svg')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cx', function(d) { return d.x; })
    .attr('cy', function(d) { return d.y; })
    .attr('r', function(d) { return d.r; })
    .style('fill', function(d) { return d.id === hoveredId ? 'red' : null;});
}

updateData();
updateQuadtree();
update();
initEvents();
```

- **德劳内三角形（Delaunay triangles）**：还可以使用 D3 的 Delaunay 模块来查找最近点。给定一个点数组，Delaunay 三角剖分将所有点与三角形连接起来，从而使碎片最小化。四叉树不同，不支持添加最大距离作为第三个参数。

```
let data = [], width = 600, height = 400, numPoints = 100;
let triangles;
let hoveredId;

function updateData() {
  data = [];
  for(let i=0; i<numPoints; i++) {
    data.push({
      id: i,
      x: Math.random() * width,
      y: Math.random() * height,
      r: 1 + Math.random() * 20
    });
  }
}

function handleMousemove(e) {
  let pos = d3.pointer(e, this);
  let i = triangles.find(pos[0], pos[1]);
```

```
  hoveredId = data[i].id;
  update();
}

function initEvents() {
  d3.select('svg')
    .on('mousemove', handleMousemove);
}

function updateDelaunay() {
  triangles = d3.Delaunay
    .from(data,
        function(d) {return d.x;},
        function(d) {return d.y;}
       );
}

function update() {
  d3.select('svg')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cx', function(d) { return d.x; })
    .attr('cy', function(d) { return d.y; })
    .attr('r', function(d) { return d.r; })
    .style('fill', function(d) { return d.id === hoveredId ? 'red' : null;});
}

initEvents();
updateData();
updateDelaunay();
update();
```

- **拖动（Dragging）**：D3 有一个用于向元素添加拖动行为的模块。D3 的拖动模块也支持触摸手势。使 HTML/SVG 元素可拖动需要三个步骤：
  - 调用`d3.drag()`创建**拖动行为**函数
  - 添加一个在拖动事件发生时调用的事件处理程序。事件处理程序接收一个事件对象，您可以使用它来更新拖动元素的位置
  - 将拖动行为附加到要使其可拖动的元素

  拖动事件对象有几个属性，其中最有用的是：

属性名称	描述
`.subject`	被拖动元素的连接数据
`.x` & `.y`	被拖动元素的新坐标
`.dx` & `.dy`	被拖动元素的新坐标，相对于之前的坐标

```
let data = [], width = 600, height = 400, numPoints = 10;

let drag = d3.drag()
  .on('drag', handleDrag);

function handleDrag(e) {
  e.subject.x = e.x;
  e.subject.y = e.y;
  update();
}

function initDrag() {
  d3.select('svg')
    .selectAll('circle')
    .call(drag);
}

function updateData() {
  data = [];
  for(let i=0; i<numPoints; i++) {
    data.push({
      id: i,
      x: Math.random() * width,
      y: Math.random() * height
    });
  }
}

function update() {
  d3.select('svg')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cx', function(d) { return d.x; })
    .attr('cy', function(d) { return d.y; })
    .attr('r', 40);
}

updateData();
update();
initDrag();
```

- **刷牙（Brushing）**:允许用户**指定一个区域**（通过按下鼠标按钮，移动鼠标，然后释放），例如，选择一组元素。将画笔行为添加到 HTML 或 SVG 元素需要三个步骤：
  - 调用`d3.brush()`创建**画笔行为**函数
  - 添加一个在发生画笔事件时调用的事件处理程序。事件处理程序接收画笔范围，然后可用于选择元素、定义缩放区域等。
  - 将画笔行为附加到一个元素（或多个元素）

```
let brush = d3.brush()
  .on('start brush', handleBrush);

function handleBrush(e) {
  // Use the brush extent e.selection to compute, for example, which elements to
select
}

function initBrush() {
  d3.select('svg')
    .call(brush);
}

initBrush();
```

事件类型：`'brush'`和`'start'`和`'end'`。`'brush'`表示画笔范围已更改。`'start'`表示刷牙已经开始（例如用户按下了鼠标按钮）。`'end'`表示刷牙结束（例如用户已松开鼠标按钮）。

`handleBrush`接收单个参数`e`，该参数是表示画笔事件的对象。画笔事件中最有用的属性`.selection`是将画笔的范围表示为一个数组`[[x0, y0], [x1, y1]]`，其中`x0, y0`和`x1, y1`是画笔的对角。通常`handleBrush`会计算哪些元素在画笔范围内并相应地更新它们。

通过选择元素并将画笔行为传递给`.call`方法，可以将画笔行为附加到元素：

```
//完整例子
let data = [], width = 600, height = 400, numPoints = 100;

let brush = d3.brush()
  .on('start brush', handleBrush);

let brushExtent;

function handleBrush(e) {
  brushExtent = e.selection;
  update();
}

function initBrush() {
  d3.select('svg g')
    .call(brush);
}

function updateData() {
  data = [];
  for(let i=0; i<numPoints; i++) {
    data.push({
      id: i,
      x: Math.random() * width,
      y: Math.random() * height
```

```
    });
  }
}

function isInBrushExtent(d) {
  return brushExtent &&
    d.x >= brushExtent[0][0] &&
    d.x <= brushExtent[1][0] &&
    d.y >= brushExtent[0][1] &&
    d.y <= brushExtent[1][1];
}

function update() {
  d3.select('svg')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cx', function(d) { return d.x; })
    .attr('cy', function(d) { return d.y; })
    .attr('r', 4)
    .style('fill', function(d) {
      return isInBrushExtent(d) ? 'red' : null;
    });
}

initBrush();
updateData();
update();
```

D3 还提供了画笔`d3.brushX`和`d3.brushY`将画笔限制在一个维度上。

画笔行为有两种设置画笔范围`.move`和`.clear`。第一个设置画笔范围，第二个清除画笔。

```
d3.select('svg g')
  .call(brush.move, [[50, 50], [100, 100]]);
```

缩放和平移（Zoom and pan）

向元素添加缩放和平移行为需要三个步骤：

1. 调用`d3.zoom()`创建**缩放行为**函数
2. 添加在发生缩放或平移事件时调用的事件处理程序。事件处理程序接收可应用于图表元素的转换
3. 将缩放行为附加到接收缩放和平移手势的元素

```
let data = [], width = 600, height = 400, numPoints = 100;

let zoom = d3.zoom()
  .on('zoom', handleZoom);
```

```
function handleZoom(e) {
  d3.select('svg g')
    .attr('transform', e.transform);
}

function initZoom() {
  d3.select('svg')
    .call(zoom);
}

function updateData() {
  data = [];
  for(let i=0; i<numPoints; i++) {
    data.push({
      id: i,
      x: Math.random() * width,
      y: Math.random() * height
    });
  }
}

function update() {
  d3.select('svg g')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cx', function(d) { return d.x; })
    .attr('cy', function(d) { return d.y; })
    .attr('r', 3);
}

initZoom();
updateData();
update();
```

可以限制缩放和平移，以便用户只能在指定范围内缩放和平移。`.scaleExtent`可以使用传递数组来限制缩放，`[min, max]`其中`min`是最小比例因子,`max`是最大比例因子;使用`.translateExtent`指定`[[x0, y0], [x1, y1]]`限制用户平移的范围。

```
let zoom = d3.zoom()
  .scaleExtent([1, 5]);

let width = 600, height = 400;

let zoom = d3.zoom()
  .scaleExtent([1, 5])
  .translateExtent([[0, 0], [width, height]]);
```

可以通过编程方式进行缩放和平移。例如，您可以创建单击时缩放图表的按钮。缩放行为具有以下以编程方式设置缩放和平移的方法：

方法名称	描述
`.translateBy`	将给定的`x, y`偏移量添加到当前变换
`.translateTo`	设置变换，使给定`x, y`坐标居中（或定位在给定点上`[px, py]`）
`.scaleBy`	将当前比例因子乘以给定值
`.scaleTo`	将比例因子设置为给定值
`.transform`	将变换设置为给定的变换。（`d3.zoomIdentity`)用于创建缩放变换。）

小结

使用d3.js需要一定的html、css、js基础。此外对于svg和canvas也要有一定的认识。如果有使用过其他图表库，对于图表类型有一定了解，也会有帮助。

相对一些库，d3的自定义性更高，可以按照你的需求，随意地使用函数来实现，而不是想方设法去利用预设的参数尽可能地贴近你的设计。

目前已经到了版本6，新特性?

D3、交互、代码架构、状态管理、样式等方面?

用d3来实现常用的图表类型?