



下载APP



## 38 | 实战（二）：如何使用数据驱动框架绘制常用数据图表？

2020-09-28 胡光

跟月影学可视化

[进入课程 >](#)**讲述：月影**

时长 13:55 大小 12.76M



你好，我是月影。

上一节课，我们使用图表库实现了一些常用的可视化图表。使用图表库的好处是非常简单，基本上我们只需要准备好数据，然后根据图形需要的数据格式创建图形，再添加辅助插件，就可以将图表显示出来了。

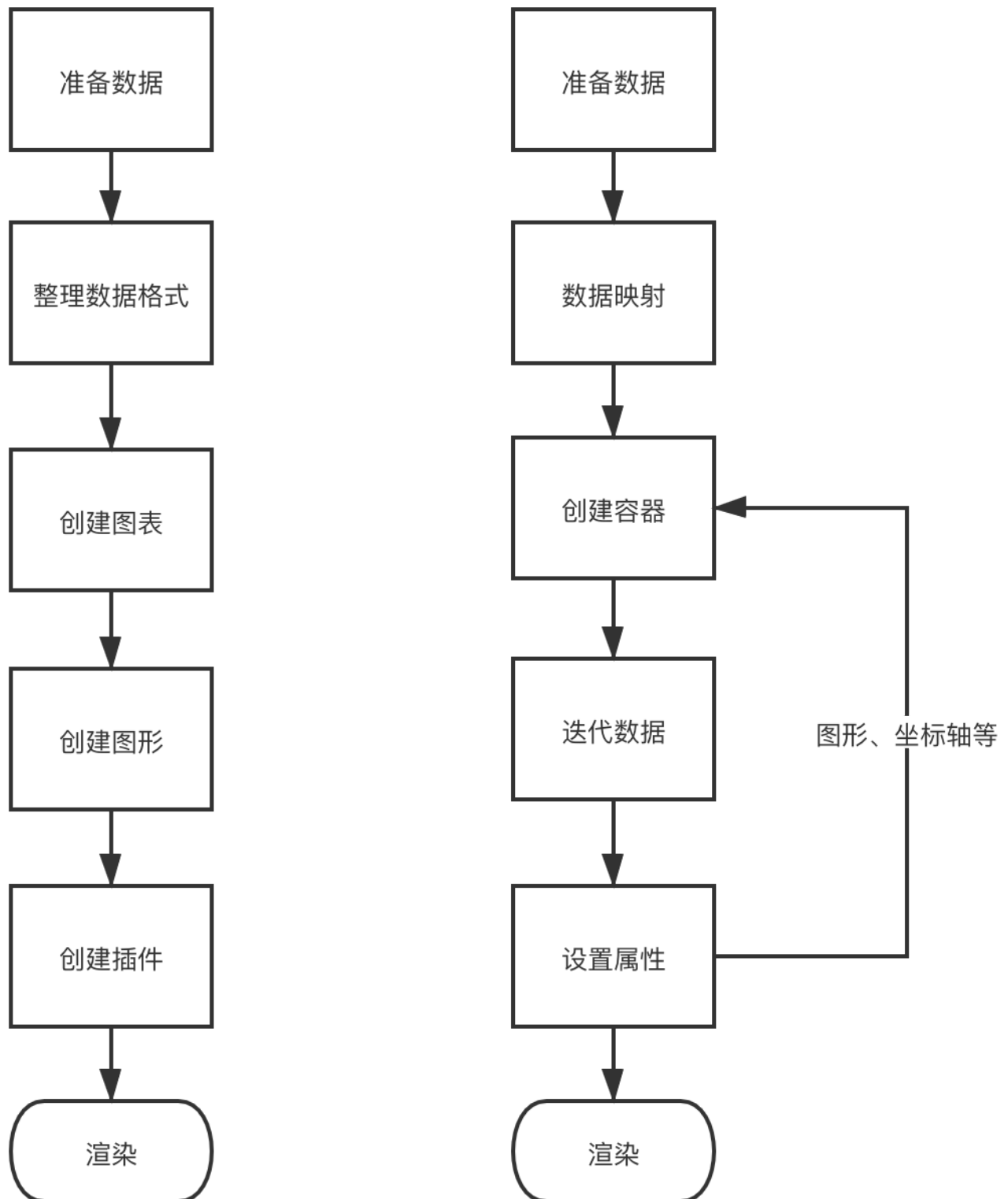
图表库虽然使用上简单，但灵活性不高，对数据格式要求也很严格，我们必须按照各个图表的要求来准备数据。而且，图形和插件的可配置性，完全取决于图表库设计者开放的 API，给开发者的自由度很少。



今天，我们就来说说，使用数据驱动框架来实现图表的方式。这类框架以 D3.js 为代表，提供了数据处理能力，以及从数据转换成视图结构的通用 API，并且不限制用户处理视图

的最终呈现。所以它的特点是更加灵活，不受图表类型对应 API 的制约。不过，因为图表库只要调用 API 就能展现内容，而数据驱动框架需要我们自己去完成内容的呈现，所以，它在使用上就没有图表库那么方便了。

使用图表库和使用数据驱动框架的具体过程和差别，我这里准备了一个对比图，你可以看一下。




使用图表库（左）和使用数据驱动框架（右）渲染图表的流程对比

不过这么讲还是比较抽象，接下来，我们还是通过绘制条形图和力导向图，来体会用数据驱动框架和用图表库构建可视化图表究竟有什么区别。

## 课前准备

与上一节课差不多，我们还是需要使用 SpriteJS，只不过今天我们将 QCharts 换成 D3.js。

 复制代码

```
1 <script src="https://unpkg.com/spritejs/dist/spritejs.min.js"></script>
2 <script src="https://d3js.org/d3.v6.js"></script>
3
```

使用上面的代码，我们就能加载 SpriteJS 和 D3.js，用它们来完成常用图表的绘制了。

## 使用 D3.js 绘制条形图

我们先来绘制条形图，条形图与柱状图差不多，都是用图形的长度来表示数据的多少。只不过，横向对比的条形图，更容易让我们看到各个数据之间的大小，而纵向的柱状图可以同时比较两个变量之间的数据差别。

用 D3.js 绘制图表，不同于使用 Qcharts，我们需要创建 SpriteJS 的容器。通过前面的课程我们已经知道，SpriteJS 创建场景（Scene）对象作为其他元素的根元素容器。接下来，我们一起看下具体的操作过程。

 复制代码

```
1 const container = document.getElementById('stage');
2 const scene = new Scene({
3   container,
4   width: 1200,
5   height: 1200,
6 });
7
```

如上面代码所示，我们先创建一个 Scene 对象，与 QCharts 的 Chart 对象一样，它需要一个 HTML 容器，这里我们使用页面上一个 id 为 stage 的元素。我们设置了参数 width 和 height 为 1200，也就是把 Canvas 对象的画布宽高设为 1200 \* 1200。

接着，我们准备数据。与使用 QCharts 必须要按照格式给出 JSON 数据不同，使用 D3.js 的时候数据格式比较自由。这里，我们直接用了一个数组：

```
1 const dataset = [125, 121, 127, 193, 309];
2
```

[复制代码](#)

然后，我们使用 D3.js 的方法对数据进行映射：

```
1 const scale = d3.scaleLinear()
2   .domain([100, d3.max(dataset)])
3   .range([0, 500]);
4
```

[复制代码](#)

D3.js 在设计上采用了一些函数式编程思想，这里的 scaleLinear、.domain 和 .range 都是高阶函数，它们返回一个 scale 函数，这个函数把一组数值线性映射到某个范围，这里，我们就是将数值映射到 500 像素区间，数值是从 100 到 309。

那么这个 scale 函数要怎么使用呢？别着急，我们先往下看。

有了数据 dataset 和处理数据的 scale 方法之后，我们使用 d3-selection（这是 d3 中的一个子模块，我们是通过 CDN 来加载 d3 的，所以已经默认包含了 d3-selection）来创建并选择 layer 对象。

在 SpriteJS 中，场景 Scene 可以由多个 Layer 构成，针对每个 Layer 对象，SpriteJS 都会创建一个实际的 Canvas 画布。

```
1 const fglayer = scene.layer('fglayer');
2 const s = d3.select(fglayer);
```

[复制代码](#)

如上面的代码所示，我们先创建了一个 fglayer，它对应一个 Canvas 画布，然后通过 d3.select(fglayer)，将对应的 fglayer 元素经过 d3 包装后返回。

接着，我们在 fglayer 元素上进行迭代操作。你先认真看完代码，我再来解释。

[复制代码](#)

```
1  const colors = ['#fe645b', '#feb050', '#c2af87', '#81b848', '#55abf8'];
2  const chart = s.selectAll('sprite')
3    .data(dataset)
4    .enter()
5    .append('sprite')
6    .attr('x', 450)
7    .attr('y', (d, i) => {
8      return 200 + i * 95;
9    })
10   .attr('width', scale)
11   .attr('height', 80)
12   .attr('bgcolor', (d, i) => {
13     return colors[i];
14   });
```

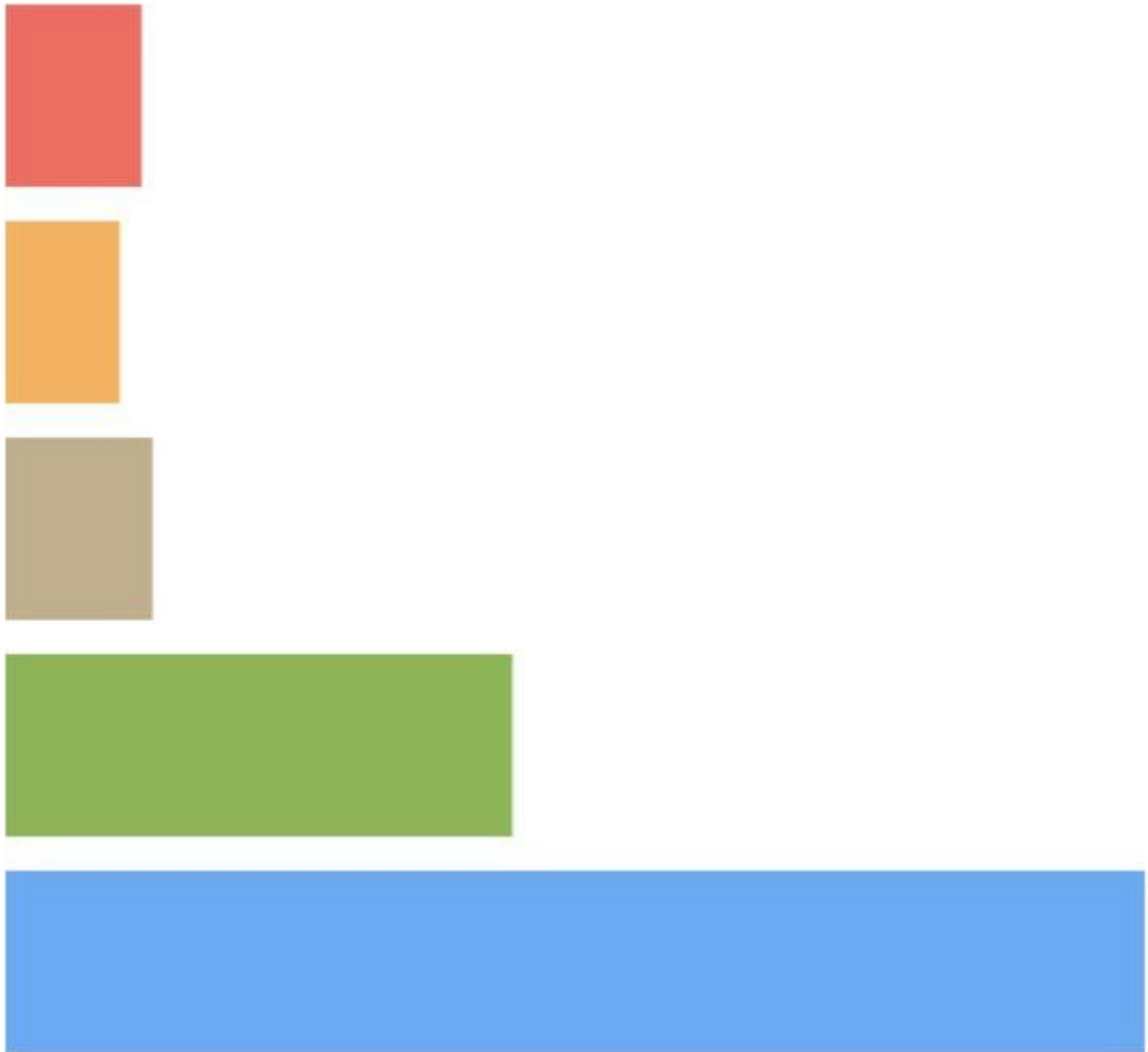
我们从第 2 行代码开始看，其中，selectAll 用来返回 fglayer 下的 sprite 子元素，对于 SpriteJS 来说，sprite 元素是基本元素，用来表示一个图形。不过，现在 fglayer 下还没有任何子元素，所以 selectAll( 'sprite' ) 本应该返回空的元素，但是，d3 通过 data 方法迭代数据集，也就是之前有 5 个元素的数组，然后通过执行 enter() 和 append( 'sprite' )，这样就在 fglayer 下添加了 5 个 sprite 子元素。enter() 方法是告诉 d3-selection，当数据集的数量大于 selectAll 选中的元素数量时，通过 append 添加元素补齐数量。

从第 6 行代码开始，我们给每个 sprite 元素迭代设置属性。注意，append 之后的 attr 是默认迭代设置每个 sprite 元素的属性，如果是常量就直接设置，如果是不同的值，就通过迭代算子来设置。迭代算子有两个参数，第一个是 dataset 中对应的数据，第二个是迭代次数，从 0 开始，因为有五项数据，所以会迭代 5 次。如果你对 jQuery 比较熟悉，你应该能比较容易理解上面这种批量迭代操作的形式。

最后，我们根据数据集的每个数据依次设置一个 sprite 元素，将 x 坐标值设置为 450，y 坐标值设置为从 200 开始，每个元素占据 95 个像素值，然后将 width 设置为用 scale 计

算后的数据项的值，这里我们就用到前面 `linearScale` 高阶函数生成的 `scale` 函数，直接将它作为算子。我们将 `height` 值设为固定的 80，表示元素的高度。这样一来，元素之间就会有  $95 - 80$ ，即 15 像素的空隙。最后我们给元素设置一组不同的颜色值。

我们最终显示出来的效果如下图：



这里我们在画布上显示了五个不同颜色的矩形条，它们对应数组的 125、121、127、193、309。但它还不是一个完整的图表，我们需要给它增加辅助信息，比如坐标轴。添加坐标轴的代码如下所示。

```
1 const axis = d3.axisBottom(scale).tickValues([100, 200, 300]);
2 const axisNode = new SpriteSvg({
3   x: 420,
```

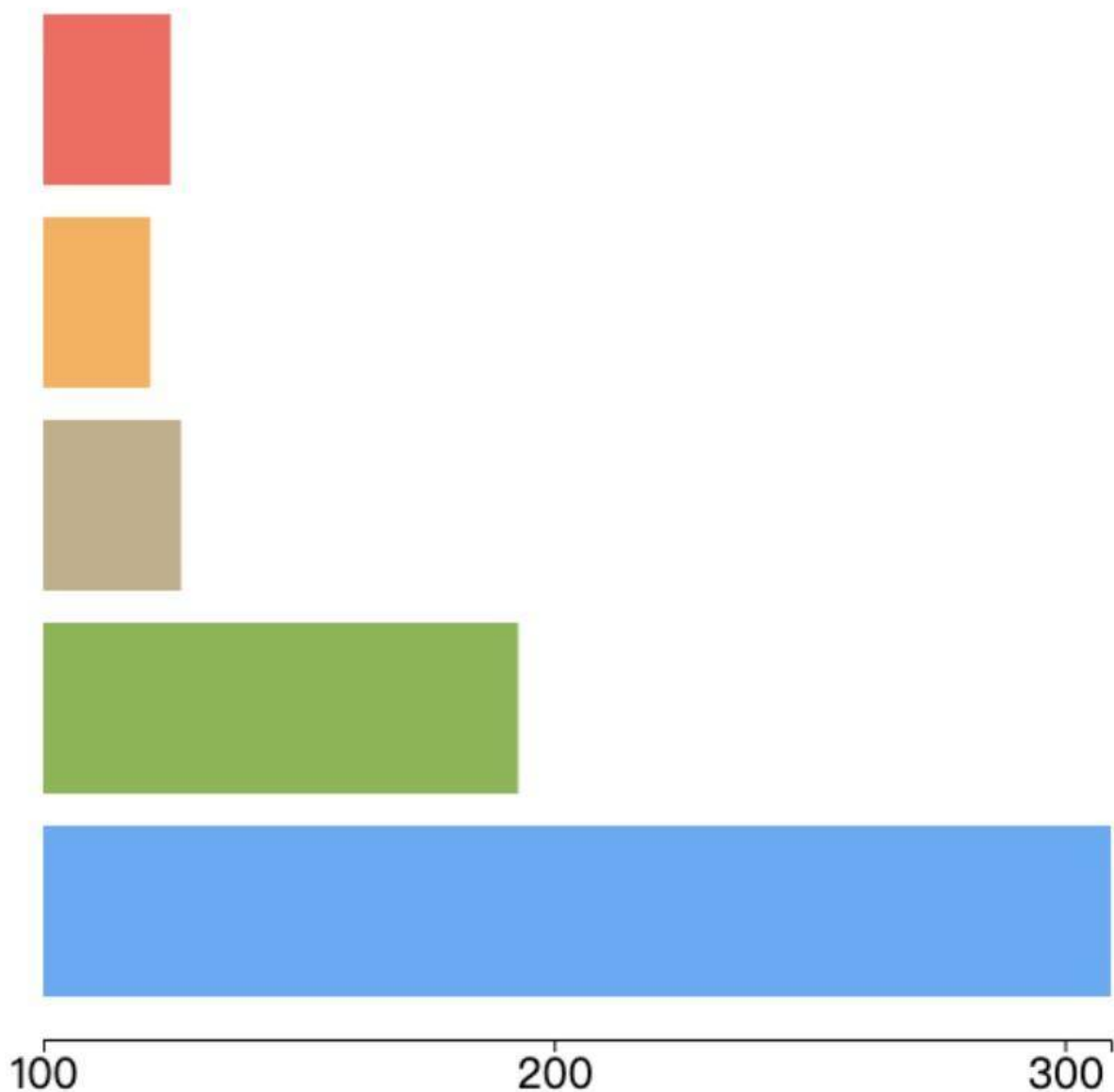
[复制代码](#)

```
4    y: 680,  
5  });  
6  d3.select(axisNode.svg)  
7    .attr('width', 600)  
8    .attr('height', 60)  
9    .append('g')  
10   .attr('transform', 'translate(30, 0)')  
11   .call(axis);  
12  
13  
14  axisNode.svg.children[0].setAttribute('font-size', 20);  
15  fglayer.append(axisNode);
```

如上面代码所示，我们通过 `d3.axisBottom` 创建一个底部的坐标。我们可以通过 `tickValues` 给坐标轴传要显示的刻度值，这里我们显示 100、200、300 三个刻度。同样我们可以用 `scale` 函数将这些数值线性映射到 500 像素区间，值从 100 到 309。

`axisBottom` 本身是一个高阶函数，它返回 `axis` 函数用来绘制坐标轴，不过这个函数是使用 `svg` 来绘制坐标轴的。好在 `SpriteJS` 支持 `SpriteSvg` 对象，它可以绘制一个 SVG 图形，然后将这个图形以 `WebGL` 或者 `Canvas2D` 的方式绘制到画布上。

我们先创建 `SpriteSvg` 类的对象 `axisNode`，然后通过 `d3.select` 选中对象的 `svg` 属性，进行正常的 `svg` 属性设置和创建 `svg` 元素操作，最终将 `axisNode` 添加到 `fglayer` 上，这样就能将坐标轴显示出来了。



这样，我们就实现了一个简陋的条形图。简陋是因为和 QCharts 的柱状图相比，它现在只有图形主体部分和一个简单的 x 坐标轴，缺少 y 坐标轴、图例、提示信息、辅助网格等信息，不过这些用 D3.js 也都能创建，我觉得这部分内容，你可以自己试着实现，我就不多说了，如果遇到问题记得在留言区提问。

总的来说，在创建简单的图表的时候，使用 D3.js 比直接使用图表库还是要复杂很多的。但比较好的一点是，D3.js 对数据格式没有太多硬性要求，我们可以直接使用一个简单的数组，然后在后面绘图的时候再进行迭代。那麻烦一点的是，因为没有现成的图表对象，所以我们要自己处理数据、显示属性的映射，好在 D3.js 提供了 `linearScale` 这样的工具函数来创建数据映射。



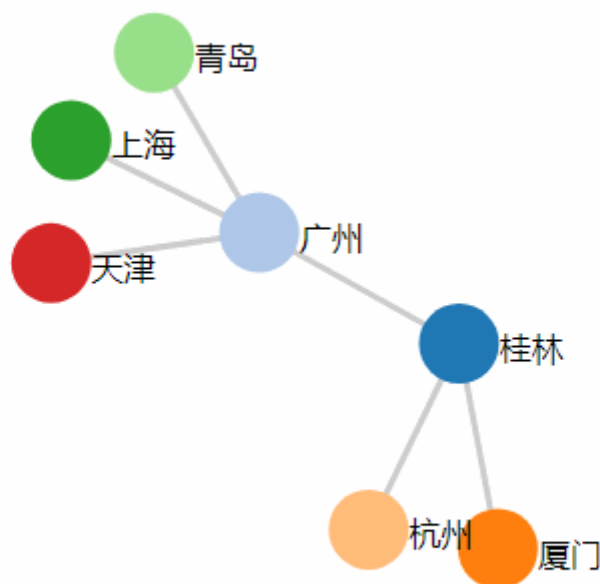
处理好数据映射之后，我们需要自己通过 d3-selection 来遍历元素，完成属性的设置，从而把图形渲染出来。而且，对于坐标轴等其他附属信息，d3 也没有现成的对象，我们也需要通过遍历元素进行绘制。

这里顺便提一下，虽然我们使用 SpriteJS 作为图形库来讲解，但 d3 并没有强限制定图形库，所以我们无论是采用 SVG、原生 Canvas2D 还是 WebGL，又或者是采用 ThreeJS 等其他图形库，都可以进行渲染。只不过，d3-selection 依赖于 DOM 操作，所以 SVG 和 SpriteJS 这种与 DOM API 保持一致的图形系统，使用起来会更加方便一些。

## 使用 D3.js 绘制力导向图

讲完了用 D3.js 绘制简单条形图的方法，接下来，我们看看怎么用 D3.js 绘制更加复杂的图形，比如力导向图。

力导向图也是一种比较常见的可视化图表，它非常适合用来描述关系型信息。比如下图就是一个经典的力导向图应用。



我们看到，力导向图不仅能够描绘节点和关系链，而且在移动一个节点的时候，图表各个节点的位置会跟随移动，避免节点相互重叠。

那么究竟如何用 D3.js 实现一个简单的力导向图呢？我们来看一个例子。

力导向图，顾名思义，我们通过模拟节点之间的斥力，来保证节点不会相互重叠。在 D3.js 中提供了模拟斥力的方法。

[复制代码](#)

```
1 const simulation = d3.forceSimulation()  
2   .force('link', d3.forceLink().id(d => d.id)) //节点连线  
3   .force('charge', d3.forceManyBody()) // 多实体作用  
4   .force('center', d3.forceCenter(400, 300)); // 力中心
```

如上面代码所示，我们创建一个 d3 的力模型对象 simulation，通过它来模拟示例，然后我们设置节点连接、多实体相互作用、力中心点。

接着，我们读取数据。这里我准备了一份 [JSON 数据](#)。我们可以用 d3.json 来读取数据，它返回一个 Promise 对象。

[复制代码](#)

```
1 d3.json('https://s0.ssl.qhres.com/static/f74a79ccf53d8147.json').then(graph =>  
2   ...  
3 });
```

我们先用力模型来处理数据：

[复制代码](#)

```
1 simulation  
2   .nodes(graph.nodes)  
3   .on('tick', ticked);  
4  
5  
6 simulation.force('link')  
7   .links(graph.links);  
8
```

接着，我们再绘制节点：

```
1 d3.select(layer).selectAll('sprite')
2   .data(graph.nodes)
3   .enter()
4   .append('sprite')
5   .attr('pos', (d) => {
6     return [d.x, d.y];
7   })
8   .attr('size', [10, 10])
9   .attr('border', [1, 'white'])
10  .attr('borderRadius', 5)
11  .attr('anchor', 0.5);
12
```

[复制代码](#)

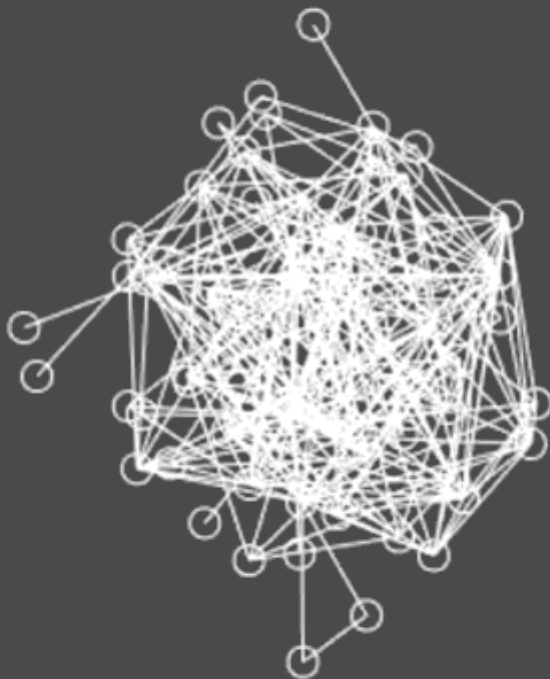
然后，我们再绘制连线：

```
1 d3.select(layer).selectAll('path')
2   .data(graph.links)
3   .enter()
4   .append('path')
5   .attr('d', (d) => {
6     const [sx, sy] = [d.source.x, d.source.y];
7     const [tx, ty] = [d.target.x, d.target.y];
8     return `M${sx} ${sy} L ${tx} ${ty}`;
9   })
10  .attr('name', (d, index) => {
11    return `path${index}`;
12  })
13  .attr('strokeColor', 'white');
14
```

[复制代码](#)

这里我们依然是用 d3-selection 的迭代，给 SpriteJS 的 sprite 和 path 元素设置了一些属性，这些属性有的与我们的数据建立关联，有的是单纯的样式。这里面没有特别难的地方，我就不一一解释了，最好的理解方法是实践，所以我建议你亲自研究一下示例代码，修改一些属性，看看结果有什么变化，这样能够加深理解。

将节点和连线绘制完成之后，力导向图的初步结果就呈现出来了。



因为力向导图有一个特点就是，在我们移动一个节点的时候，其他节点也会跟着移动。所以，我们还要实现拖动节点的功能。D3.js 支持处理拖拽事件，所以我们只要分别实现一下对应的事件回调函数，完成时间注册就可以了。首先是三个事件回调函数。

[复制代码](#)

```
1 function dragstarted(event) {  
2   if(!event.active) simulation.alphaTarget(0.3).restart();  
3  
4  
5   const [x, y] = [event.subject.x, event.subject.y];  
6   event.subject.fx0 = x;  
7   event.subject.fy0 = y;  
8   event.subject.fx = x;  
9   event.subject.fy = y;
```

```
10
11   const [x0, y0] = layer.toLocalPos(event.x, event.y);
12   event.subject.x0 = x0;
13   event.subject.y0 = y0;
14 }
15
16
17 function dragged(event) {
18   const [x, y] = layer.toLocalPos(event.x, event.y),
19     {x0, y0, fx0, fy0} = event.subject;
20   const [dx, dy] = [x - x0, y - y0];
21
22
23   event.subject.fx = fx0 + dx;
24   event.subject.fy = fy0 + dy;
25 }
26
27
28 function dragended(event) {
29   if(!event.active) simulation.alphaTarget(0);
30   event.subject.fx = null;
31   event.subject.fy = nul
32 }
```

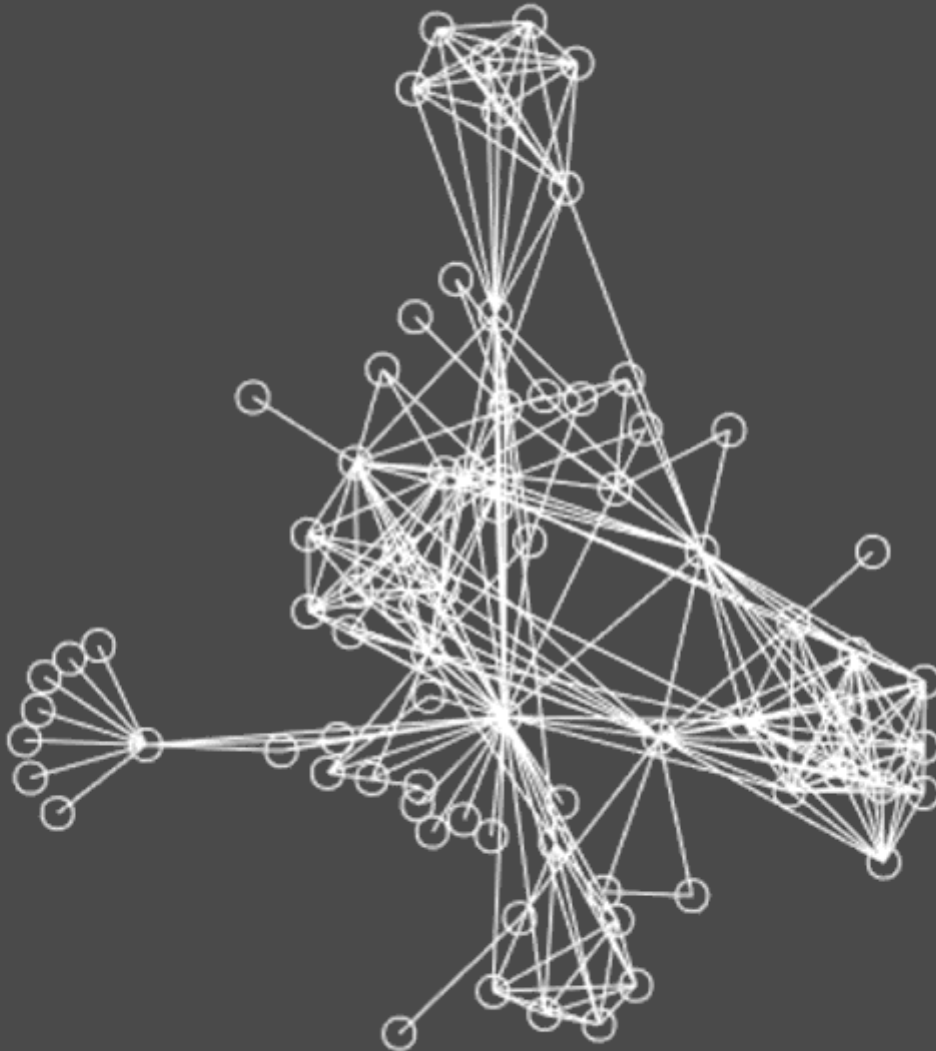
其中 `dragstarted` 处理开始拖拽的事件，这个时候，我们通过前面创建的 `simulation` 对象启动力模拟，记录一下当前各个节点的 `x`、`y` 坐标。因为默认的坐标是 DOM 事件坐标，我们通过 `layer.toLocalPos` 方法将它转换成相对于 `layer` 的坐标。接着 `dragged` 处理拖拽中的事件，同样也是转换 `x`、`y` 坐标，计算出坐标的差值，然后更新 `fx`、`fy`，也就是事件主体的当前坐标。最后，我们用 `dragended` 处理拖拽结束事件，清空 `fx` 和 `fy`。

接着，我们将三个事件处理函数注册到 `layer` 的 `canvas` 上：

```
1   d3.select(layer.canvas)
2     .call(d3.drag()
3       .container(layer.canvas)
4       .subject(dragsubject)
5       .on('start', dragstarted)
6       .on('drag', dragged)
7       .on('end', dragended));
8
```

[复制代码](#)

这样就实现了力导向图拖拽节点的交互，d3 会自动根据新的节点位置计算布局，避免节点的重叠。



## 要点总结

这节课，我们主要学习了使用数据驱动框架来绘制图表。

与直接使用图表库不同，使用数据驱动框架不要求固定格式的数据格式，而是通过对原始数据的处理和对容器迭代、创建新的子元素，并且根据数据设置属性，来完成从数据到元素结构和属性的映射，然后再用渲染引擎将它最终渲染出来。

那你可能有疑问了，我们应该在什么时候选择图表库，什么时候选择数据驱动框架呢？通常情况下，当需求比较明确可以用图表库，并且样式通过图表库 API 设置可以实现的时候，我们倾向于使用图表库，但是当需求比较复杂，或者样式要求灵活多变的时候，我们可以考虑使用数据驱动框架。

数据驱动框架可以灵活实现各种复杂的图表效果，我们前面举的两个图表例子虽然只是个例，但也会在实战项目中经常用到。除此之外，使用 D3.js 和 SpriteJS 还可以实现其他复杂的图表，比如说，地图或者一些 3D 图表，以及我们在前面的课程中实现的 3Dgithub 代码贡献图，就是使用 D3.js 和 SpriteJS 来实现的。

D3.js 和 SpriteJS 的使用都比较复杂，你是不可能用一节课系统掌握的，我们只有继续深入学习，并动手实践、积累经验，才能在可视化项目中得心应手地使用它们，来实现各种各样的可视化需求。

## 小试牛刀

最后，我给你出了两个实践题。希望你能结合 D3.js 和 SpriteJS 的官方文档，花点时间仔细阅读和学习，再通过动手实践和反复练习，最终掌握它们。

1. 请你完善我们课程中讲到的条形图，给它实现 y 轴、图例和提示信息。
2. 你可以将上一节课用 QCharts 图表库实现的图表改用 D3.js 实现吗？动手试一试，体会一下它们使用方式和思路上的不同。

关于可视化图表的实战课程就讲到这里了，如果你对于图表绘制，还有什么疑问和困惑，欢迎你在留言区告诉我。我们下节课再见！

---

## 源码

课程中完整示例代码详见 [🔗 GitHub 仓库](#)

## 推荐阅读

[🔗 D3.js 的官方文档](#)



[提建议](#)

## 更多课程推荐

# 数据结构与算法之美

为工程师量身打造的数据结构与算法私教课

王争

前 Google 工程师



立省 ¥40



破 90000 订阅特惠，到手价 ¥89

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 37 | 实战（一）：如何使用图表库绘制常用数据图表？

下一篇 39 | 实战（三）：如何实现地理信息的可视化？

## 精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。



