

D3.js

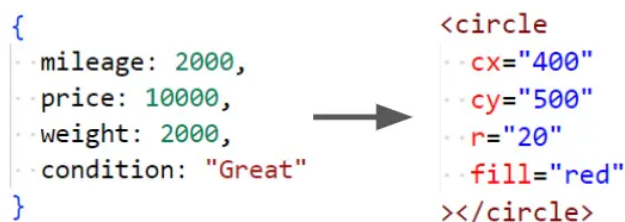
D3.js简介

Data-Driven Documents

官网: <https://d3js.org/>

中文文档: https://github.com/xswei/d3js_doc

构造了从数据到HTML元素的框架



The diagram illustrates the mapping from a data object to an SVG element. On the left, a JavaScript object is shown with properties: `mileage: 2000`, `price: 10000`, `weight: 2000`, and `condition: "Great"`. An arrow points to the right, where the corresponding SVG `<circle>` element is shown. The attributes of the circle are: `cx="400"` (derived from `mileage * 0.2`), `cy="500"` (derived from `price * 0.05`), `r="20"` (derived from `weight * 0.01`), and `fill="red"` (derived from the `condition` value).

D3.js和原生JavaScript的比较

原生JavaScript: 数据和元素之间不耦合

需要遍历数据设置相应元素的属性, 遇到数据修改时, 需要重新找到数据对应的元素。

```
let container = document.getElementById('container')
let svgNS="http://www.w3.org/2000/svg" // svg元素的命名空间
for(let d of data){
  let circle = document.createElementNS(svgNS, 'circle')
  circle.setAttribute('cx', d.mileage * 0.2)
  circle.setAttribute('cy', d.price * 0.05)
  circle.setAttribute('r', d.weight * 0.01)
  circle.setAttribute('fill', dist[d.condition])
  container.appendChild(circle)
}
```

D3.js: 元素和数据绑定

属性值可以通过函数形式表达, 修改起来更加容易定位

```
d3.select('#container')
  .selectAll('circle')
  .data(data)
  .enter()
  .append('circle')
  .attr('cx', d=>d.mileage * 0.2)
  .attr('cy', d=>d.price * 0.05)
  .attr('r', d=>d.weight * 0.01)
  .attr('fill', d=>dist[d.condition])
```

D3.js封装了一些常见的可视化工具和方法

举例:

算法

d3.force 力导向算法

d3.pack 圆打包算法

数据结构

d3.hierarchy 层次结构

- d3.histogram 直方图
- 工具
 - d3.axis 坐标轴
 - d3.path 路径
- 交互
 - d3.zoom 缩放
 - d3.drag 拖拽

D3.js基础

总览

D3.js基本操作

- 数据读取和预处理（数据读取以d3.csv为例、预处理可以提一下d3.extent）
- 元素选择（d3.selectAll和d3.select）
- 数据和元素的耦合（selection.data、d3.join）
- 元素属性修改（selection.attr、selection.style、selection.property）

D3.js可视化示例：

- 散点图（涉及d3.scale、d3.axis）
- 堆叠条形图（每个datum对应多个元素）

D3.js导入

```
1 <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
```

D3.js基本操作

数据读取和预处理

以d3.csv为例：

由于文件读取可能有延迟，所以需要以回调函数的形式写处理数据以及后续可视化的方法，不能直接在d3.csv语句后面直接写数据的处理（异步问题）

```
d3.csv('./data.csv').then(data=>{  
  console.log(data)  
  // do something with data  
})
```

id	sepal_length	sepal_width	petal_length	petal_width	species
0	6.4	2.8	5.6	2.2	virginica
1	5	2.3	3.3	1	versicolor
2	4.9	2.5	4.5	1.7	virginica
3	4.9	3.1	1.5	0.1	setosa
4	5.7	3.8	1.7	0.3	setosa

```
[
  {
    "id": "0",
    "sepal_length": "6.4",
    "sepal_width": "2.8",
    "petal_length": "5.6",
    "petal_width": "2.2",
    "species": "virginica"
  },
  {
    "id": "1",
    "sepal_length": "5",
    "sepal_width": "2.3",
    "petal_length": "3.3",
    "petal_width": "1",
    "species": "versicolor"
  }
]
```

其他数据：d3.json、d3.text、d3.image等

数据预处理：

会涉及查找最大最小值、计算平均值等操作。可以使用JavaScript原生方法实现，也可以使用d3提供的一些封装（如d3.extent、d3.mean等）

元素选择

- (1) 基本遵循css对标签的定位写法（标签名、id、class）
- (2) 可以使用链式写法逐级查找（当然也可以不写成链式的）

d3.select('#container') // 选中id="container"的标签A

.select('svg') // 选中标签A内部的<svg>标签

.selectAll('.element') // 选中<svg>标签内部所有的class="element"的标签

非链式的写法：

```
let container = d3.select('#container')
```

```
let svg = container.select('svg')
```

```
svg.selectAll('.element')
```

元素属性修改

通用方法：

attr(attrName, attrValue) // 设置attribute

style(styleName, styleValue) // 设置style

classed(className, isInClass) // class属性的特殊写法

property(propertyName, propertyValue) // 设置property，不常用

html(html) // 设置元素内部html

特殊方法：

text(text) // 设置元素内部text（针对<text>等文本标签）

示例：

```
svg.selectAll('.element')
```

```
.data(data, d=>d.id)
```

```
.update()
```

```
.attr('transform', d=>`translate(${d.x},${d.y})`)
```

```
.style('opacity', 0.7)
```

```
<g
```

```
  transform="translate(100,100)"
```

```
  style="opacity: 0.7"
```

```
></g>
```

D3.js可视化示例

散点图

零、文件结构

我们的这张图里面有什么东西是用 HTML 决定的
然后这些东西在哪个位置长什么样？是用是chart.js决定的
然后这些东西是什么颜色是style.css 决定的

一、准备工作

(1) f12可以看到，d3是很规整的，需要层层嵌套的，首先给一个div，在div中画一个叫box的svg，这个svg上有content这个group（创造content组是为了方便一起操作，而且看着比较干净）（默认在左上角所以需要transform到正确的位置），content里面又有坐标轴和散点图的dot-area的group。

class/id都讲过，这里可以再提一下

```
...
  <div id="box-div">
    <svg id="box" width="600" height="600">
      <g id="content" style="transform: translate(50px, 50px);">
        <g id="dot-area">...</g>
        <g class="axis" fill="none" font-size="10" font-family="sans-serif" text-anchor="middle" style="transform: translateY(500px);">...</g>
        <g class="axis" fill="none" font-size="10" font-family="sans-serif" text-anchor="end">...</g>
      </g>
    </svg>
  </div>
...
```

(2) 以下代码用来定义这个区域中的各个图元，先画一个正方形，给他一些边距，然后content的长宽就是这个正方形长宽-边距

.attr是规定这个图元中的属性（英文是attribute）

.append是链式语法（前面应该讲过），不断地加东西而不会影响前面的

translate的特殊写法 `translate(\${px},\${px})` 是因为此处需要的是一个字符串，但这样框住之后，dms.margin.top 就变成一个单纯的字符串了，它就不是值了，所以外面要加一个美元符号，加一个大括号，把外面的单引号变成点号

```

const dms = {
  width: 600,
  height: 600,
  margin: {
    top: 50,
    right: 50,
    bottom: 50,
    left: 50
  }
}

dms.contentWidth = dms.width - dms.margin.left - dms.margin.right
dms.contentHeight = dms.height - dms.margin.top - dms.margin.bottom

const box = d3.select('#box-div').append('svg')
  .attr('id', 'box')
  .attr('width', dms.width)
  .attr('height', dms.height)

const content = box.append('g')
  .attr('id', 'content')
  .style('transform', `translate(${dms.margin.left}px, ${dms.margin.top}px)`)

const dotArea = content.append('g')
  .attr('id', 'dot-area')

const widAxis = content.append('g')
  .attr('class', 'axis')
  .style('transform', `translateY(${dms.contentHeight}px)`)

const lenAxis = content.append('g')
  .attr('class', 'axis')

```

(3) 读取csv, 前面应该会讲

```

d3.csv('../data/data.csv')
  .then(drawDotChart)

function drawDotChart(dataset) {
  const widGet = d => d.wid
  const lenGet = d => d.len

```

function偷懒写法 把函数装进变量里, 传进来一个d, 返回d.wid

(4) 因为这里没有时间数据, 所以可以都用线性比例尺

domain是定义域

range是值域

.nice是让它变好看 (更规整)

```

const widScale = d3.scaleLinear()
  .domain(d3.extent(dataset, widGet))
  .range([0, dms.contentWidth])
  .nice()

const lenScale = d3.scaleLinear()
  .domain(d3.extent(dataset, lenGet))
  .range([dms.contentHeight, 0])
  .nice()

```

(5) 开始画小圆点，定义数据源、坐标和半径

```
dotArea.selectAll('circle')
  .data(dataset)
  .join('circle')
  .attr('cx', d => widScale(widGet(d)))
  .attr('cy', d => lenScale(lenGet(d)))
  .attr('r', 3)
```

(6) 最后画坐标轴，ticks是刻度，text是写在旁边的图例文字

```
const widAxisGen = d3.axisBottom()
  .scale(widScale)
  .ticks(8)

widAxis.call(widAxisGen) //sepal_length,sepal_width

widAxis.append('text')
  .attr('class', 'title')
  .attr('x', dms.contentWidth-10)
  .attr('y', -20)
  .text('sepal_width')

const lenAxisGen = d3.axisLeft()
  .scale(lenScale)
  .ticks(8)

lenAxis.call(lenAxisGen)

lenAxis.append('text')
  .attr('class', 'title')
  .attr('x', 20)
  .attr('y', -20)
  .text('sepal_length')
```

直方图

零、代码结构

写过上一个散点图后，我们不难发现，代码每部分是有各自的功能的
包括尺寸、组件、数据、坐标区间、画图、标注，
接下来我们会讲柱状图的各部分

(1) 尺寸，包括宽度、高度、边距，以及内容宽度、内容高度

```

1  const dms = {
2    width: 800,
3    height: 500,
4    margin: {
5      top: 50,
6      right: 50,
7      bottom: 100,
8      left: 50
9    }
10 }
11
12 dms.contentWidth = dms.width - dms.margin.left - dms.margin.right
13 dms.contentHeight = dms.height - dms.margin.top - dms.margin.bottom
--

```

(2) 组件，我们要用到的是box、content，用来画bar的区域，还有x轴

```

const box = d3.select('#box-div').append('svg')
  .attr('id', 'box')
  .attr('width', dms.width)
  .attr('height', dms.height)

const content = box.append('g')
  .attr('id', 'content')
  .style('transform', `translate(${dms.margin.left}px, ${dms.margin.top}px)`)

const barArea = content.append('g')
  .attr('id', 'bar-area')

const widAxis = content.append('g')
  .attr('id', 'wid-axis')
  .style('transform', `translateY(${dms.contentHeight+10}px)`)

```

(3) 接着读入数据，还是从我们这个鸢尾花数据中读，横坐标是sepal_width的区间，纵坐标是有多少朵花的sepal_width落在这个区间，所以叫count，这个countget的值就是这个数据的数量，注意这里的d.length不是我们前面说的d.len（数据len这一列）而是指数据的频次。

```

35 d3.csv('../data/data.csv')
36   .then(drawBarChart)
37
38 function drawBarChart(dataset) {
39   const widGet = d => d.wid
40   const countGet = d => d.length

```

(4) 接着是坐标变换 wideScale是横坐标，界定定义域、值域就可以了。

我们这里写的是histogram，普通的柱状图bar chart每一个柱子就代表数据，数据多大它多长，但是histogram帮助我们划分区间，看落在每个区间有多少朵花。

引入d3.histogram，它可以帮我们判断个数

它的值就是前面获得的sepal_width，定义域就是wideScale（横坐标多定义域），因为横坐标经过了.nice，histGen要和它保持一致。

最后传入dataset，得到hist，hist就是这个分组，每组有多少个。

可以打印一下，得到x0,x1，是代表它柱的左边和右边，length 是它的个数，它的个数的后来就会变成柱的高度。

接着就需要用Scale把它的length转化为高度，countScale就是起这个作用，定义域是0到最大值，值域是内容高度到0。

colorScale是为了做出渐变的效果，这里先不讲。

bar画出来都是挤在一起的，很不好看，所以给一个边距，边距为4

```
const widScale = d3.scaleLinear()  
  .domain(d3.extent(dataset, widGet))  
  .range([0, dms.contentWidth])  
  .nice()  
  
const histGen = d3.histogram()  
  .value(widGet)  
  .domain(widScale.domain())  
  .thresholds(10)  
  
const hist = histGen(dataset)  
  
const countScale = d3.scaleLinear()  
  .domain([0, d3.max(hist, countGet)])  
  .range([dms.contentHeight, 0])  
  .nice()  
  
const colorScale = d3.scaleLinear()  
  .domain(d3.extent(hist, countGet))  
  .range(['lightskyblue', 'cornflowerblue'])  
  .nice()  
  
const barPadding = 4
```

(5) 接着开始画图，画柱子的部分以及画上x、y轴，加上title

D3.js的讲解形式：

以接口为核心，接口≈D3js提供的函数调用等。

对于每个独立接口进行讨论：输入、输出、用法等。

通过编程实例(code)来讨论每个接口的应用。

交互

总览

d3.js交互方法基础

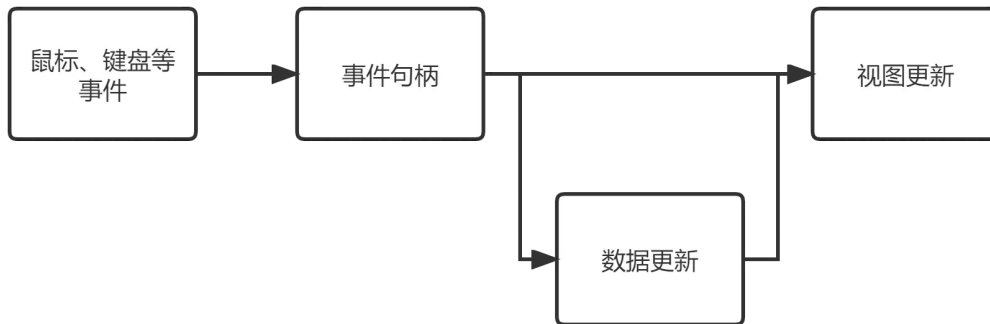
- 事件绑定

- 数据选中与更新

- 过渡动画

可视化示例

- 树图的例子

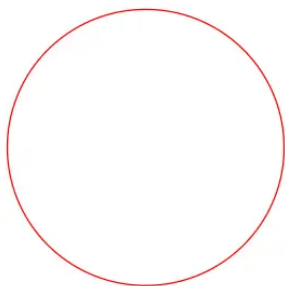


事件绑定

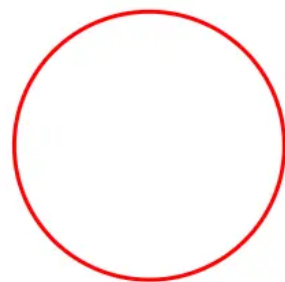
on方法设置监听事件（主要涉及鼠标事件）

```
d3.select('circle')
  .on('mouseover', function(){
    d3.select(this)
      .attr('stroke-width', 3)
  })
  .on('mouseout', function(){
    d3.select(this)
      .attr('stroke-width', 1)
  })
```

初始时：



鼠标hover上时



取消绑定事件

```
d3.select('circle')
  .on('click', null)
```

其他事件：mousemove、mouseenter、mouseleave、dblclick等

d3.js封装好的一些事件：brush、drag、zoom



```
let brush = d3.brush()  
...extent([[0, 0], [300, 300]]) // 可以刷选的范围  
...on('start', function(e){  
...})  
...on('brush', function(e){  
...// e.selection是刷选范围的坐标[[x0, y0], [x1, y1]]  
...})  
...on('end', function(e){  
...})  
svg.call(brush)
```

```
let drag = d3.drag()  
...on('start', function(e){  
...d3.select(this).attr('stroke', 'black')  
...})  
...on('drag', function(e){  
...let circle = d3.select(this)  
...circle  
...    .attr('cx', Number(circle.attr('cx')) + e.dx)  
...    .attr('cy', Number(circle.attr('cy')) + e.dy)  
...})  
...on('end', function(e){  
...d3.select(this).attr('stroke', 'red')  
...})  
circle.call(drag)
```



```
let zoom = d3.zoom()  
...extent([[0, 0], [300, 300]]) // 窗口的范围  
...scaleExtent([0.1, Infinity]) // 缩放的比例范围  
...translateExtent([[0, 0], [400, 400]]) // 世界的范围  
...on('zoom', function(e){  
...    circle.attr('cx', e.transform.applyX(x0))  
...    circle.attr('cy', e.transform.applyY(y0))  
...    circle.attr('r', e.transform.k * r0)  
...})  
svg.call(zoom)
```

数据选中与更新

在刷选和缩放交互中，通常会涉及到数据的更新，进而引发渲染元素的更新

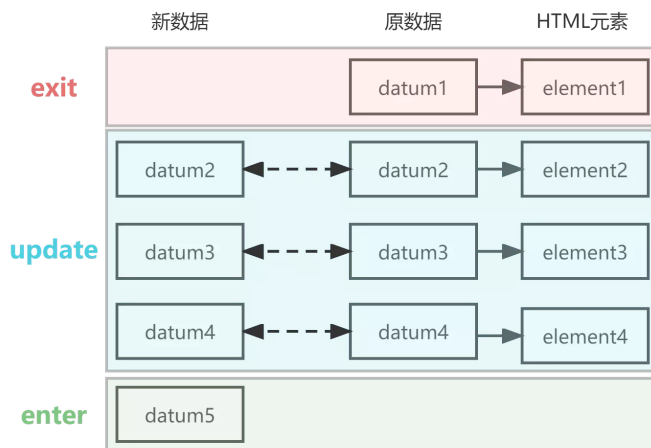
(1) 数据更新时，可以使用join函数进行处理

数据绑定

```
svg.selectAll('.element')  
  .data(data, d=>d.id)
```

d=>d.id用来确定数据前后是否发生变化

数据和元素关系的三种情况：



```
svg.selectAll('.element')
  .data(data, d=>d.id)
  .enter() // 选取enter部分
  .append('g') // 增加新的元素<g>
```



```
svg.selectAll('.element')
  .data(data, d=>d.id)
  .exit() // 选取exit部分
  .remove() // 移除exit部分对应的元素
```



更简洁的写法: join

```
svg.selectAll('.element')
  .data(data, d=>d.id)
  .join(
    .enter=>{}, // enter部分的处理函数
    .update=>{}, // update部分的处理函数
    .exit=>{}, // exit部分的处理函数
  )
```

(2) 当选中某一部分数据时, 可以通过`selection.filter`将数据对应的元素筛选出来进行相应的操作

```
d3.selectAll('circle')
  .filter(d=>d.radius > 10)
```

(3) `selection.merge`: 合并两个选择集

```

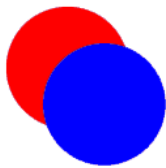
circle = svg.selectAll("circle") // update部分
  .data(data)
  .style("fill", "blue")

circle.exit().remove() // exit部分

circle = circle.enter()
  .append("circle") // enter部分
  .style("fill", "green")
  .merge(circle) // enter部分和update部分合并
  .style("stroke", "black")

```

(4) selection.raise和order: 将置于下方的内容提到上面



```

let circles = svg
  .selectAll('circle')
  .data(data, d=>d.id)
  .enter()
  .append('circle')
  .attr('cx', d=>d.cx)
  .attr('cy', d=>d.cy)
  .attr('r', d=>d.r)
  .attr('fill', d=>d.fill)
  .on('mouseover', function(){
    d3.select(this).raise() // 将选中节点提升到最上面
  })
  .on('mouseout', function(){
    circles.order() // 恢复数据对应的顺序
  })

```

过渡动画

触发鼠标事件改变元素时，如何实现一个平滑的过渡？

- 使用selection.transition函数

```

let circle = svg
  ..append('circle')
  ..attr('cx', 100)
  ..attr('cy', 100)
  ..attr('r', 50)
  ..attr('fill', 'red')

circle
  ..transition()
  ..delay(1000) // 延迟1000ms执行过渡
  ..duration(2000) // 过渡时间为2000ms
  ..attr('cx', 150)
  ..attr('fill', 'blue')

```



-
- transition的监听函数

可以在过渡的各个阶段执行各种操作

```

circle
  ..transition()
  ..delay(1000)
  ..duration(2000)
  ..attr('cx', 150)
  ..attr('fill', 'blue')
  ..on('start', function(){
    console.log('过渡开始')
  })
  ..on('end', function(){
    console.log('过渡完成')
  })
  ..on('interrupt', function(){
    console.log('过渡被中止')
  })

```

连续多个transition



```

circle
  .transition()
  .delay(1000)
  .duration(2000)
  .attr('cx', 150)
  .transition() // 上面的过渡完成之后，紧接着执行下面的过渡
  .delay(500)
  .duration(1000)
  .attr('fill', 'blue')

```

Q：哪些属性可以过渡？

A：

大多数属性可以通过内置的默认方法实现过渡

常见的不能过渡的：这些在过渡一开始就会立即变成最终的值，不会有过渡的效果

(1) .text() // 设置标签内的文字

变通的做法：设置不透明度

```

.text("文本内容")
.style('opacity', 0) // 设置不透明度为0
.transition()
.duration(500)
.style('opacity', 1) // 逐渐过渡到不透明度为1

```

(2) .style('display', 'none') // 是否显示

变通的做法：将opacity过渡到0，必要时可以在'end'监听函数里再设置display为none

```

.transition()
.duration(500)
.style('opacity', 0)

```

(3) path.attr('d') // path的路径

方法：使用attrTween方法自定义插值

(4) .property('scrollTop/scrollLeft', number)

方法：使用tween方法自定义插值（后面会讲）

Q：过渡的中间帧是怎么生成的？（两个问题：怎么样的速度，每一帧怎么画）

A：

一、过渡速度的变化：transition.ease

d3.easeCubic（默认值）



d3.easeLinear



二、过渡中间帧的计算

对于大多数属性，都有内置的默认的过渡方法（插值函数）。

自定义使用的场景：

- (1) 默认方法不能满足需求的（例如，从红到蓝的过渡，不希望中间有一段白色）
- (2) 不存在默认过渡方法的变量

方法：

```
.attrTween(name, factory)
.styleTween(name, factory)
.tween(name, value)
```

示例：

scrollTop

```
let scrollTop = document.body.getBoundingClientRect().height - window.innerHeight
d3.transition()
  .delay(1000)
  .duration(2000)
  .tween("scroll", scrollTween(scrollTop))
function scrollTween(offset) {
  return function() {
    let i = d3.interpolateNumber(
      window.pageYOffset || document.documentElement.scrollTop, offset
    )
    return function(t) {
      d3.select(document.documentElement).property('scrollTop', i(t))
    }
  }
}
```



多视图联动

可视化的一致性

- 当在一个视图中选中部分数据时，其他使用相同数据的视图中也应该有相应的选中效果
- 这个如何在编程中实现？

可视化状态

一些全局的变量，标记当前可视化处于什么状态。例如：

- 某一个数据点被鼠标选中=>相应的数据点需要高亮显示
- 部分数据被刷选出来=>除了一个用于刷新的缩略图，其他只显示刷选框内的数据

当状态发生改变时，需要更新与这个状态相关的视图

例子：国际组织成员国可视化

Step1：需要哪些视图？

需要哪些视图应该根据自己想要探索的目标来决定，而且不同视图之间的功能应该相互补充，而不是为了多视图而多视图。

- 标题和统计信息。给出数据的概览性叙述，让我们对数据有一个大概的认识；
- 图例。解释数据是如何被编码的；
- 集合视图。展示国际组织之间的关系；

- 国家列表。展示每个国家的信息。

Step2: 定义需要哪些状态变量?

比如这里我希望更清晰看到每一个国家和每一个组织，所以需要在鼠标选中这个元素时高亮相关内容，这样可以看得更清楚。所以我希望以下状态变量：

当前选中了哪个国家。进一步可以分成鼠标悬停在哪个国家上countryHovered、鼠标点击了哪个国家countryClicked。

- 鼠标交互的逻辑：
 - 悬停时给出选中状态的预览，取消悬停时就会退回原来的状态；
 - 当鼠标点击之后，会覆盖原来的状态，即使鼠标移开也会继续保持新的状态。
- 如何用代码表示：

```
1 const countrySelected = countryHovered || countryClicked;
```

当前选中了哪个组织。当然也包括organizationHovered和organizationSelected

Step3: 这些状态变量如何影响视图效果?

	选中一个组织	选中一个国家	同时选中一个国家和一个组织
标题和统计信息	显示选中组织的统计信息	显示选中国家的统计信息	显示选中国家的统计信息
集合视图	高亮选中的组织，同时未选中的组织中的国家不能被选中	高亮选中的国家，同时不包含该国家的组织不能再被选中	高亮选中的国家，次高亮选中的组织
图例	高亮选中的组织	高亮选中国家的相关组织	高亮选中的组织
国家列表	高亮选中组织的相关国家并将其排列在前面	高亮选中的国家并将其排列在前面	高亮选中的组织，并将选中的国家进一步加粗

到这里就可以看到每个视图对于各个状态的依赖情况。

Step4: 绘制各个视图

每个视图都有一些固定的函数：

- render。给定这个视图所需要的数据，以及初始状态，绘制整个视图。可以绑定一些鼠标或键盘事件来触发相应的状态改变；
- update。当状态发生改变时，可以调用这个函数更新当前视图。

如果对编程比较熟悉，甚至可以将这两个内容合并成一个函数。无论是初始绘制还是更新状态都可以调用它

Step5: 设置状态与视图的联动

state.js

```
1 import { update as updateTitle } from './title.js';
2 import { update as updateSet } from './set.js';
```

```

3 import { update as updateLegend } from "./legend.js";
4 import { update as updateList } from "./list.js";
5
6 let countryHovered = null;
7 let countryClicked = null;
8
9 function updateCountryHovered(newCountryHovered) {
10   countryHovered = newCountryHovered;
11   updateTitle();
12   updateSet();
13   updateLegend();
14   updateList();
15 }
16 function updateCountryClicked(newCountryClicked) {
17   countryClicked = newCountryClicked;
18   updateTitle();
19   updateSet();
20   updateLegend();
21   updateList();
22 }
23
24 export {
25   countryHovered,
26   updateCountryHovered,
27   countryClicked,
28   updateCountryClicked,
29 }

```

list.js

```

1 import { countryHovered, countryClicked } from "./state.js";
2
3
4 function render(container) {
5
6 }
7
8 function update() {
9   const country = countryHovered || countryClicked;
10   d3.selectAll("g")
11     .data(data)
12     .join("g")
13     .attr(...) // 其他的全部恢复默认
14     ...
15     .filter(d => d.id === country)
16     .attr(...) // 选中高亮
17 }
18
19 export {

```

```
20   render,  
21   update,  
22 }
```

main.js

```
1  import { render as renderTitle } from "./title.js";  
2  import { render as renderSet } from "./set.js";  
3  import { render as renderLegend } from "./legend.js";  
4  import { render as renderList } from "./list.js";  
5  
6  const container = d3.select("#container"); // 整个画布  
7  
8  const titleContainer = container.select("#title");  
9  renderTitle(titleContainer);  
10  
11 const setContainer = container.select("#set");  
12 renderSet(setContainer);  
13  
14 const legendContainer = container.select("#legend");  
15 renderLegend(legendContainer);  
16  
17 const listContainer = container.select("#list");  
18 renderList(listContainer);
```