D3 学习笔记

2017年7月18日　　11:01

- ❖ **本笔记约定格式 & 样式：**
  - 普通标题或正文，最大字号为12号，每一层级字号缩减0.5（微软雅黑）
  - 高亮词（微软雅黑）
  - 链接（微软雅黑）
  - 举例、补充、资料等信息（字体不做限制）
  - `code （Consolas）`
- ❖ **在线 API**
- ❖ **基于《D3.js数据可视化实战手册》**

---

1. 基础

1) 选择器，语法：selector，同 css3 的选择器语法

- ◇ d3.select 方法接受一个 css3 选择器字符串或待操作对象的引用。如果选集中有多个元素，则返回第一个。选择多个元素请用 selectAll 方法。
- ◇ 子选择器， 语法：selector selector

2) 修饰函数：部分 D3 修饰函数会返回一个新的选集，例如 select、append、insert 函数。

(1) selection.classed 用来添加、删除选定元素上的 css class。

检测 p 元素是否有名为 goo 的 class

```
d3.select('p').classed('goo')
```

为 p 元素添加 goo class

```
d3.select('p').classed('goo', true)
```

移除 p 元素上的 goo class。 classed 方法也接收函数作为参数传入，从而可以动态地添加或移除 css class

```
d3.select('p').classed('goo', function() {
    return false
})
```

(2) selection.style 用来给选定元素添加指定样式

获取 p 元素的 font-size

```
d3.select('p').style('font-size')
```

将 font-size 的值设为 10px

```
d3.select('p').style('font-size', '10px')
```

将 font-size 的值设为某个函数的运算结果。 style 方法也接受函数作为参数传入，从而可以动态地改变样式的值

```
d3.select('p').style('font-size', function() {
    return normalFontSize + 10
```

```
})
```

- ◇ 另外还有 text()、html()、attr()，用法类似以上两种，但 attr() 的参数不接受传入 function(){}

3) 函数的级联

(1) 同 javascript 的级联调用

(2) D3 级联处理可以生成任意复杂的结构。事实上，典型的基于 D3 的数据可视化结构正是这样创建的。许多可视化项目都只简单包含一个 HTML 骨架，然后用 D3 来创建剩余部分。

(3) 注意，级联的修饰函数的参数如果是一个匿名函数，这个匿名函数里面的参数 d（绑定到图形的数据）会被其他修饰函数或子元素所继承。

4) 遍历：selection.each

接受迭代函数作为参数传入，迭代函数有两个参数 d（绑定的数据）和 i（索引），以及一个隐含参数 this 指向当前 DOM 元素的引用。

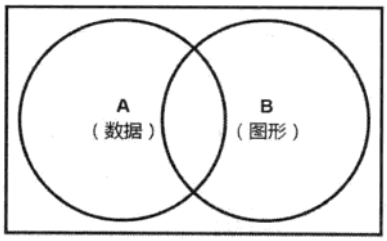
2. 数据与图形

> 数据和信息的差别：数据是纯粹的事实。“纯粹”意味着这种事实没有经过任何处理，其意义也没有得到揭示。而信息是数据处理的结果，它揭示了数据的意义。（Rob P.、 S.Morris 与 Coronel C. 2009）

1) D3 可视化的基石：进入-更新-退出（enter-update-exit）模式

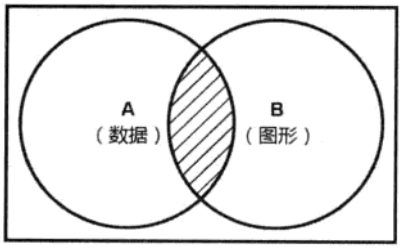
d3.select(selector).data(data)

示例：用两个非空集合分别表示数据集（A）和图形集（B）：



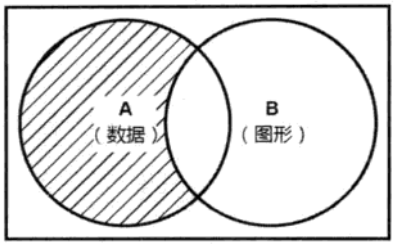
(1) 更新状态

- ◇ 使用 selection.data(data) 函数选择交集（A∩B），建立数据集和图形集的联系。
- ◇ 其中， d3.select(selector) 选择的 D3 对象集合组成了 B
- ◇ 而 data 函数中的参数 data 组成了集合 A
- ◇ 针对 A∩B 得到的新集合调用相关函数，对其中的元素进行更新。这个新集合所在的状态称为更新状态（Update Mode）



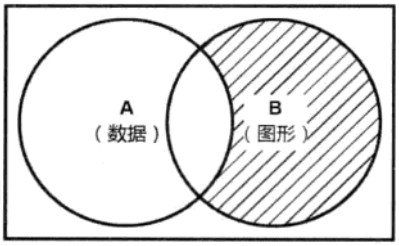
(2) 进入状态

- ◇ 使用 selection.data(data).enter() 函数选择差集（A\B），建立还未被可视化的数据与图形的关联。这个集合的状态称为进入状态（Enter Mode）
- ◇ 个人理解：通常首次绑定数据时不会出现这种状态，因为 D3 根据数据生成图形，首次加载的数据理应全部绑定了图形，至于进入状态集合里面的数据是怎么来的，是在页面上的图形被手动删除或获取到新数据后产生的。



(3) 退出状态

使用 selection.data(data).exit() 函数选择差集（B\A），即在一开始为 A 集合中的每条数据分别与 B 集合里面的每个图形建立了联系后，再从 A 集合里面删除某些数据，而 B 集合与之关联的图形就不再具有数据绑定，形成了一个新的没有数据绑定的图形集合，这个集合的状态成为退出状态（Exit Mode）



2) 数据绑定：

> D3在相应的DOM元素中添加了一个名为 __data__ 的属性，并通过这个属性将数据和图形联系起来。这样，当选定元素的数据集合更新的时候，D3可以正确地计算出图形集合和数据集合的交集与差。我们可以直接用程序或者通过调试器得到附加在DOM元素上的 __data__ 属性的值。

(1) 处理数组

①. d3.min：返回最小元素
②. d3.max：返回最大元素
③. d3.extent：返回最小和最大的元素组成的数组
④. d3.sum：返回所有元素的和
⑤. d3.medium：返回中间值

⑥. d3.mean：返回数组的平均值

⑦. d3.ascending/d3.descending：排序

```
d3.ascending = function (a, b) {
    return a < b ? -1 : a > b ? 1 : 0
}

d3.descending = function (a, b) {
    return b < a ? -1 : b > a ? 1 : 0
}
```

◇ 更多函数请参考 D3在线API#数组

(2) 数据过滤：selection.filter()

◇ 参数为一个匿名函数，这个匿名函数同样有两个参数 d 和 i 以及一个隐含的引用this，代表当前图形的 DOM 元素。

◇ 选集中的每一个元素作为参数依次调用这个匿名函数

◇ 匿名函数返回一个布尔值： true 以及其他可以转化为 true 的 javascript 真值（即与假值 false、null、undefined、0 和 NaN 相对应的值）：相应的图形元素会被包含在 filter 函数新生成的选集中

(3) 数据排序：selection.sort()

①. 参数为一个比较函数

②. 这个函数接受两个参数（a， b），分别为要比较的对象或值

③. 返回一个正数、负数或0

④. 如果返回负数，则 a 将排在 b 之前；正数相反

⑤. 如果为 0，顺序随机决定

⑥. sort 函数会返回一个新的选集

(4) 加载数据：

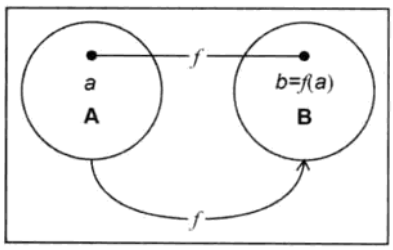
```
d3.json('data.json', function (error, json) {
    //callback
})
```

◇ 可使用 d3.xhr 自定义请求数据类型及头部信息

◇ D3 还支持加载 CSV、TSV、TXT、HTML 以及 XML 格式数据的函数

3. 尺度（比例尺）与插值

对于函数f，集合A是它的定义域，集合B是它的值域。若A代表数据域，B代表图形域，那么函数f就是d3中将集合A映射到集合B的一个尺度。



函数 f 为 A 到 B 的映射

1) 基本操作

(1) 创建一个尺度

```
var scale = d3.scale.linear()
```

domain：绑定数据

```
    .domain([1, 10])
```

range：设置图形

```
    .range([1, 10])
```

注意：range() 和 rangeRound() 基本一样，但是后者会将输出的数字进行舍入取整。

调用尺度

```
selection.text(function (d) {
    return d3.round(scale(d), 2)
})
```

(2) 创建一个时间尺度

```
var start = new Date(2013, 0, 1),
    end = new Date(2013, 11, 31),
    range = [0, 1200],
    time = d3.time.scale()
        .domain([start, end])
        .rangeRound(range)
```

调用时间尺度

```
d3.html(function (d) {
    var format = d3.time.format('%x')
    return format(d) + '-' + scale(d) + 'px'
})
```

◇ 补充： d3.time.format 为 d3 内置的时间格式化库，用来操作 javascript的 Date 对象非常有用。详细内容见 d3在线api#time

2) 尺度分类：

(1) 数值尺度

- 线性尺度：d3.scale.linear()
- 幂级尺度：d3.scale.pow().exponent()

  exponent 函数用来指定幂级

  如：f(n) = n² 等价于 d3.scale.pow().exponent(2)
- 对数尺度：d3.scale.log()

(2) 时间尺度：d3.time.scale()

(3) 有序尺度：d3.scale.ordinal()

3) 插值器

> 给定函数f(x)在x0, x1, …, xn处的值。现有x'，其值在上述取值点之间。那么，求f(x')值的过程叫做插值。 (Kreyszig E & Kreyszig H & Norminton E.J. (2010))

(1) 数值插值器

◇ 使用 d3.interpolateNumber 函数创建了一个值域为 [0， 100] 的 interpolate 函数，并使用这个函数对指定的数字进行插值

```
var interpolate = d3.interpolateNumber(0, 100)
interpolate(0.1);                          //=>10
interpolate(0.99);                         //=>99
```

◇ 以上代码与如下代码是等价的。

```
function interpolate(t) {
    return a * (1 - t) + b * t
}
```

◇ 在这个函数中，a 代表了值域的开始，b 代表了值域的终止。t 为取值范围从 0 到 1 的浮点数。

(2) 字符串插值

为字符串中内嵌的数字进行插值，例如 css 字体样式。

(3) 颜色插值

不是所有的浏览器否支持 HSL 或者 L*a*b 的颜色空间，因此 D3 的颜色插值函数，无论原颜色空间是何种类型，总是返回 RGB 空间的颜色值。

(4) 复合对象插值

◇ 如果需要插值的数据不是一个简单的值，而是包含多个不同值的对象，例如一个由宽度、高度和颜色属性构成的矩形对象，也被支持。如：

```
var compoundScale = d3.scale.pow()
    .exponent(2)
    .domain([0, 10])
    .range([
        {color: '#add8e6', height: '15px'},
        {color: '#4169e1', height: '150px'}
    ])
```

◇ 从以上数据结构可以看出该尺度是由两个不同类型的值组成的，一个是 RGB 颜色值，而另一个是 CSS 高度样式。使用这种复合尺度插值时，D3 会遍历对象中的所有成员，并对其每一个组成部分应用相应的简单插值规则。因此，换句话说，在本例中，D3 会使用从 #add8e6 到 #4169e1 的尺度对 color 变量进行插值，使用从 15px 到 150px 的尺度对 height 变量进行插值。

◇ 这种算法的递归特性使 D3 甚至可以对嵌套对象进行插值。如：

```
{
    color: '#add8e6',
    size: {
        height: '15px',
        width: '25px'
    }
}
```

◇ 当值域的起始对象和结束对象的属性不一致时，D3 并不会报错，而会将这些不一致的属性看作常量。例如，下面这段代码的尺度函数会把所有 div 元素的高度渲染为 15px。

```
var compoundScale = d3.scale.pow()
```

```
    .exponent(2)
    .domain([0, 10])
    .range([
        {color: '#add8e6', height: '15px'},
        {color: '#4169e1'}
    ])
```

(5) 自定义插值器

◇ 当 D3 内置的插值器不能满足需求时，可以考虑实现一个自定义插值器。

```
d3.interpolators.push(function(a, b){
    var re = /^\$([0-9,.]+)$/,
    var ma, mb, f = d3.format(",.02f")
    if (
        (ma = re.exec(a))
        && (mb = re.exec(b))
    ){
        a = parseFloat(ma[1])
        b = parseFloat(mb[1]) - a
        return function(t) {
            return "$" + f(a + b * t)
        }
    }
})
```

◇ 向全局变量 d3.interpolators 数组中添加一个插值器函数，即可通过这种方式使其全局可见。插值器函数是一个工厂函数，将值域的开始（a）和结束（b）作为输入，并返回一个插值函数的实现。和 D3 内置的插值器调用方法一致。

◇ d3.interpolators 在默认情况下包含以下插值器：

- 数字插值器
- 字符插值器
- 颜色插值器
- 对象插值器
- 数组插值器

◇ 插值器的原理：

- 事实上，将 d3.interpolators 看作一个数组并不恰当。它更像一个栈，虽然并没有严格实现为栈结构，新的插值器总是被放在栈顶。因此，在这种情况下，新添加的插值器反而会被优先选取。
- 根据用户传入的字符串，使用正则表达式去匹配参数 a 和 b，如果两个参数都符合匹配规则，即创建并返回插值函数。否则 D3 会继续遍历 d3.interpolators 数组，直到找到一个合适的插值器。

4. 坐标轴

尽量使用 SVG 元素代替 HTML 元素，因为 D3 中 Axis 组件仅支持 SVG！

1）坐标轴基础

◇ 声明变量

```
var height = 500,
    width = 500,
```

```
    margin = 25,
    offset = 50,
    axisWidth = width - 2 * margin
```

(1) 创建svg画布：

```
var svg = d3.select("body").append("svg")
```

以及一些属性设置：

```
.attr("class", "axis")
.attr("width", width)
.attr("height", height)
```

(2) 创建一个Axis组件

```
var axis = d3.svg.axis()
    .scale(scale)
    .orient(orient)
    .ticks(5)
```

①. axis 用来生成一个包含 D3 中数值尺度、时间尺度和有序尺度的容器。以上代码中 scale 方法为 Axis 提供了尺度。

②. orient 方法用于设置 axis 的朝向

- top：水平坐标轴，标题位于坐标轴之上
- bottom：水平坐标轴，标题位于坐标轴之下
- left：竖直坐标轴，标题位于坐标轴左面
- right：竖直坐标轴，标题位于坐标轴右面

③. ticks 方法设置坐标轴的刻度，但 D3 可能会根据可用空间和它自己的计算多画几个或者少画几个

(3) 创建 svg:g，渲染坐标轴所需要的全部 SVG 结构都会放在它里面。

```
svg.append("g")
    .attr("transform", function(){
        if (["top", "bottom"].indexOf(orient) >= 0)
            return "translate(" + margin + ", " + offset + ")"
        else
            return "translate(" + offset + ", " + margin + ")"
    })
    .call(axis)
```

①. 使用 transform 属性来计算在 svg 画布上绘制坐标轴的位置

②. 使用 translate 来变换坐标轴的偏移量，通过这个方法我们可以用 x、y 坐标来移动元素。

③. 将 axis 作为参数传入 d3.selection.call 函数。d3.selection.call 会在当前选集上调用该参数（axis）代表的方法，即 d3.selection.call 函数的参数应该满足下面的格式：

```
function foo (selection [, other]) {
    // code
}
```

◇ D3 的 Axis 组件被调用后，会自动创建所有需要的 SVG 元素。

2) 自定义刻度线

创建坐标轴并设置尺度

```
var axis = d3.svg.axis()
    .scale(scale)
```

ticks：控制坐标轴上的刻度个数

```
    .ticks(5)
```

ticksSubdivide：设置每个刻度之间的小刻度数

```
    .tickSubdivide(5)
```

tickPadding：设置标签数字跟坐标轴的距离（px）

```
    .tickPadding(10)
```

tickFormat：给每一个值后面加上百分号

```
    .tickFormat(function (v) {
        return v + "%"
    })
```

3) 绘制表格线

以下以 x 轴为例，y轴同理：

(1) 当在坐标轴上使用了 translate 时，所有的子元素的参考坐标系都发生了改变。因为所有的 svg:g 元素中的所有子元素，都自动地进行了基础坐标系转换，故而这种变换也同样应用到了这些新的坐标值上。

在 svg:g 上执行以下代码后，

```
.attr("transform", function () {
    return "translate(" + margin + ", " + (height - margin) + ")"
})
```

在 svg:g 下创建一个坐标为 (0, 0) 的点，在把这个点画到 SVG 画布上之后，发现它的实际坐标为 (margin, height - margin) 。

(2) 绘制坐标轴以后，坐标轴上所有的刻度都是由 svg:g 元素封装起来的，可以通过选择 g.tick 来获取选集。

获取 g.x-axisg.tick 集合

```
d3.selectAll("g.x-axisg.tick")
```

给 svg:g 元素附加一个 line 元素，并设置样式

```
  .append("line")
    .classed("grid-line", true)
```

line 元素是由 SVG 提供的最简单的图形，是一条由点 (x1, y1) 到点 (x2, y2) 的直线

```
    .attr("x1",0)
    .attr("y1",0)
    .attr("x2",0)
    .attr("y2", -(height - 2 * margin))
```

补充：

① 在 SVG 坐标体系中，(0, 0) 是 SVG 画布的左上角。

② 在以上代码中，将 x1、 y1 和 x2 都设置为 0，因为所有的 g.tick 元素都已经通过 translate 转换成它在坐标轴的位置了，所以要画竖直的网格线，只需要改变一下 y2 属性的值。

③ y2 值为 -(height - 2 * margin) ：为负数是因为整个 g.x-axis 已经转变为 (height - margin) 了。所以在最终的绝对坐标体系里，

```
y2 = (height - margin) - (height - 2 * margin) = margin
```

这个就是 x 轴上绘制的那条竖直网格线的顶端。

④ svg:line 元素就是 svg:g 容器 (之前选中的 g.tick 集合) 里面的网格线。

⑤ y 轴的网格线方法相同，唯一的区别是对于 x 轴的网格线设置的是 y2 的值，对于y轴，要设置 x2 的值，因为 y 轴的网格线是水平的。

4) 动态调节坐标轴尺度

◇ 通过改变坐标轴的 domain 来调节调节尺度，从而动态调节坐标轴尺度。

先用 remove 函数删除所有网格线

```
var lines = d3.selectAll("g.x-axisg.tick")
    .select("line.grid-line")
    .remove()
```

再依据坐标轴上新的刻度来重画所有的网格线

```
lines = d3.selectAll("g.x-axisg.tick")
    .append("line")
    .classed("grid-line", true)

lines.attr("x1", 0)
    .attr("y1", 0)
    .attr("x2", 0)
    .attr("y2", -yAxisLength)
```

◇ 用这个方法，在调节坐标轴的尺度这一过程中，所有的网格线也和刻度保持了一致。

5. 过渡与动画

动画是由一系列静止图像的快速演变形成的。人类的眼部和脑部借助正后象、飞现象（是一种错视现象，描述一连串静态图片却会造成移动的错觉）和 beta 运动，产生了连续影像的错觉。

D3 过渡使我们可以在网页上用 HTML 和 SVG 创造计算机动画。D3 过渡实现了一种基于插值的动画。受计算机本身特性影响，大多数计算机动画都是基于插值的。顾名思义，这里的基础是插值。

1) 过渡

◇ 过渡基于插值和尺度。随着时间不断变换值，形成了动画。每个过渡都有起始和结束值（在动画中也称为关键帧），然后使用不同的算法和插值器，在帧与帧之间插入中间值（in-betweening ，简称 tweening）

◇ 使用 d3.selection.transition 函数来定义一个过渡。

◇ transition 函数：在获取到的原集合的基础上添加了一些额外的函数，返回一个具备过渡能力的新选集。

◇ duration 函数：设置过渡效果的持续时间，以毫秒为单位

◇ 基于插值的动画在使用插值器设置中间值时，通常需要指定起始值和结束值。D3 过渡将 transition 函数调用点之前的所有值作为起始值，将调用完毕后设置的值作为结束值。如果缺失起始值，它将试图使用计算出的样式，如果缺失结束值，则将当前值作为常量。

2) 单元素动画

```
d3.select("body").append("div")
    .classed("box", true)
    .style("background-color", "#e9967a")
```

获取一个具备过渡能力的新选集

```
    .transition()
```

设置过渡效果的持续时间

```
    .duration(5000)
```
过渡属性

起始值为 #e9967a 结束值为 add8e6
```
    .style("background-color", "#add8e6")
```
起始值为既定样式 且大于结束值 600px
```
    .style("margin-left", "600px")
```
起始值为既定样式 且大于结束值 100px
```
    .style("width", "100px")
```
起始值为既定样式 且大于结束值 100px
```
    .style("height", "100px")
```

3) 多元素动画

实例：使用数据驱动多元素过渡来生成一个变化的柱状图（完整过一遍“进入-更新-退出”模式）

(1) data 函数的第二个参数——对象标识函数。

> 对象标识函数：确保返回对象的一致性，使数据与图形之间的绑定更稳定。每一条数据都应该具备一个唯一标识，以便 D3 确认图形集里面的元素是否绑定了唯一的数据。这样，即便绑定的数据值发生了变化，相同 id 的数据仍然对应同一个图形元素。

在本例中，对象一致性非常关键，否则滑动效果是不可能完成的。

```
var selection = d3.select("body")
    .selectAll("div.v-bar")
    .data(
        data,
```
第二个参数为对象标识函数
```
        function (d) {
            return d.id
        }
    )
```

(2) 使用 enter 函数创建条形图，并基于索引值计算每个条形图的 left 位置属性。

```
selection.enter()
    .append("div")
        .attr("class", "v-bar")
        .style("position", "fixed")
        .style("top", "100px")
        .style("left", function (d, i) {
```
计算 left 值并返回，这里是实现条形图从右至左滑动的关键
```
            return i * (30 + 2) + "px"
        })
```
将条形图的 height 设为 0，实现从无增长到指定高度的动画效果
```
        .style("height",  "0px")
    .append("span")
```

“进入”模式并未调用过渡，所以这里指定的值都将作为过渡的起始值。

(3) update

```
selection
```

为所有更新都引入过渡，在样式变化之前调用transition，以创建具备过渡能力的选集

```
    .transition().duration(5000)
```

然后创建过渡样式

```
    .style("top", function (d) {
        return 100 - d.value + "px"
    })
    .style("left", function (d, i) {
        return i * (30 + 2) + "px"
    })
    .style("height", function (d) {
        return d.value + "px"
    })
    .select("span")
        .text(function (d) {
            return d.value
        })
```

(4) exit

```
selection.exit()
```

为了不影响整体的左移动画效果，获取到新的图形选集后，并不立即执行 remove 函数，而是在这个集合上继续添加过渡效果

```
    .transition().duration(5000)
    .style("left", function (d, i) {
        return -1 * (30 + 2) + "px"
    })
```

使其平滑地移出可视范围之后再执行 remove 函数

```
    .remove()
```

(5) 剩下的事情就是使用 setIntervel 函数，每秒重新绘制图表就完成了。

4) 缓动函数 ease

(1) D3内置的缓动模式

```
var data = [
    "linear", "cubic", "cubic-in-out", "sin", "sin-out", "exp",
    "circle", "back", "bounce",
    function (t) {
        return t * t
    }
]
```

(2) 绑定数据

“更新”模式—— div 元素已经绑定了以上的数据（data）

```
d3.selectAll("div").each(function (d) {
    d3.select(this)
```

调用过渡函数之后，调用缓动，参数 d 为 data 数组的数据

```
        .transition().ease(d)
        .duration(1500)
        .style("left", "10px")
})
```

(3) 补充

- 当一个过渡生成均匀的值变化时，称为线性缓动。
- 所有的内嵌缓动函数都用其缓动效果命名
- (1) 中 data 数组中最后一个函数为自定义的缓动函数（这里为二次曲面缓动）

◇ d3.ease 函数的参数不能像其他属性那样使用一个匿名函数来定义不同的缓动效果

无效写法

```
d3.transition().ease(function (d) {
    return d
})
```

ease 函数不支持这种写法，但可以使用第 ①、② 步的替代方案。另一种替代方案是使用中间帧（见下文）。在实际项目中，很少会对每个元素都定义缓动行为

◇ 无法对每个元素/属性都自定义缓动函数

5) 中间帧（本例用到的量化尺度：d3.scale.quantize）

(1) 未使用中间帧，这一过渡使用 D3 字符串插值器，实现简单连续性插值

```
body.append("div").append("input")
    .attr("type", "button")
    .attr("class", "countdown")
    .attr("value", "0")
    .style("width", "150px")
    .transition().duration(duration).ease("linear")
        .attr("value", "9")
        .style("width", "400px")
```

(2) 使用中间帧：通过工厂函数 tween 实现

```
body.append("div").append("input")
    .attr("type", "button")
    .attr("class", "countdown")
    .attr("value", "0")
    .transition().duration(duration).ease("linear")
```

传入匿名函数（中间帧计算函数），attrTween

```
        .attrTween("value", function () {
```

定义量化尺度，将值域 [0, 1] 映射到 [1, 9]

```
            var interpolate = d3.scale.quantize()
                .domain([0, 1])
                .range([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

定义实际中间帧计算函数

```
            return function (t) {
                return interpolate(t)
            }
        })
```

以下同理 styleTween

```
        .styleTween("width", function () {
            var interpolate = d3.scale.quantize()
                .domain([0, 1])
                .range([150, 200, 250, 350, 400])

            return function (t) {
                return interpolate(t) + "px"
            }
        })
```

补充：

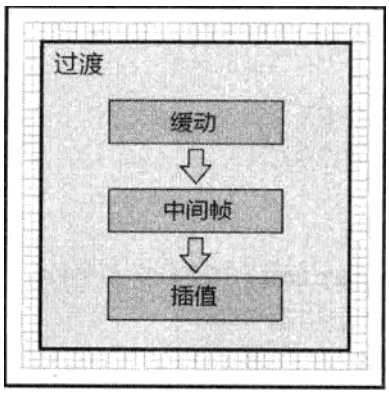
① D3 中，tween 函数是一个工厂函数，用来构造执行中间帧计算的最终函数。

② 实际中间帧计算函数：通过量化尺度对传入的时间参数插值，最终生成了跳跃的整数效果。

③ 本例的中间帧计算使用了线性缓动，但实际上，D3 对缓动中间帧计算有强大的支持，我们可以在自定义的中间帧计算中，结合前面提到的任意缓动函数，来

实现更加复杂的过渡效果。

④. 过度关系图



过渡关系图

- ◇ D3 在这三个级别都支持自定义，这为我们提供了极大的灵活性。
- ◇ 尽管自定义中间帧计算通常用插值实现，但并没有严格限定 tween 函数中应该使用什么。所以完全抛开 D3 插值器，自行实现中间帧计算函数是可行的。

6) 级联过渡

有些时候，做再多的缓动和中间帧，对于单个过渡来说是不够的，例如，想要模拟div元素的远距传动，即将div元素先压缩为一束光线，然后传送到页面的另一个位置，最后把div还原为原始尺寸。

当级联过渡效果时，每一个过渡都将会在前一过渡达到完成状态之后才起作用。

创建一个 D3 选集并设置过渡起始值

```
d3.select("body")
    .append("div")
    .style("position", "fixed")
    .style("background-color", "steelblue")
    .style("left", "10px")
    .style("width", "80px")
    .style("height", "80px")
```

使用 call 函数对选集执行过渡，匿名函数的参数 div 即是该选集

```
    .call(function (div) {
```

定义和初始化了第一个过渡：压缩

```
        div.transition().duration(300)
            .style("width", "200px")
            .style("height", "1px")
```

定义第二个过渡：传送

```
        .transition().duration(100)
            .style("left", "600px")
```

定义第三个过渡：还原

```
        .transition().duration(300)
            .style("left", "800px")
            .style("height", "80px")
            .style("width", "80px")
    })
```

补充：还可把 call 函数的参数（匿名函数）单独提取出来，以便对复杂过渡的重用。

7) 选择性过渡

数据

```
var data = ["Cat", "Dog", "Cat", "Dog", "Cat", "Dog", "Cat", "Dog"],
    duration = 1500
```

生成选集并设置过渡的起始值

```
d3.select("body").selectAll("div")
    .data(data)
.enter()
.append("div")
    .attr("class", "fixed-cell")
    .style("top", function (d, i) {
        return i * 40 + "px"
    })
    .style("background-color", "steelblue")
    .style("color", "white")
    .style("left", "500px")
    .text(function (d) {
        return d
    })
```

生成具有过渡能力的选集

```
    .transition().duration(duration)
        .style("left", "10px")
```

使用 filter 函数生成仅包含“Cat”的子选集

```
    .filter(function (d) {return d == "Cat"})
```

使这个子选集具备过渡能力

```
        .transition()
        .duration(duration)
```

设置过渡的结束值

```
            .style("left", "500px")
```

补充：

- ◇ 将以上给子选集添加过渡能力的过渡，实质上是创建级联过渡，因为只有在上一个过渡完成后才会被触发。
- ◇ 将级联过渡和选择性过渡结合起来，就能生成一些非常有趣的数据驱动动画，这才是可视化非常实用的工具。

8) 监听过渡事件 transition.each

```
transition.each([type, ]listener)
```

- ◇ 如果不传 type，效果等同于 selection.each
- ◇ type事件： ‘start’ ， ‘end’ ， ‘interrupt’
- ◇ listener：事件监听器

9) 自定义插值的过渡

一旦注册了自定义插值器，剩下的过渡基本不再需要任何自定义设置了，因为基于待插值和过渡的数据，D3 能够自动选择强档的插值器

10) 定时器（d3.timer）：D3 过渡生成动画帧的底层实现

```
d3.timer(function[, delay[, time]])
```

- ◇ 启动一个自定义动画计时器，重复地调用（上一次调用执行完后立即执行下一次调用）指定的函数（function），直到它返回 true 。
- ◇ 计时器启动后没有办法把它取消，所以一定要确保完成时，计时器函数返回 true 。

- delay 用于指定计时器给定函数延迟执行的毫秒数，默认为 Date.now
- 使用延迟（delay）和 事件（time）指定 function 应该开始被调用的相对和绝对时刻：
  十月二十九日午夜前四个小时
  ```
  d3.timer(notify, -4 * 1000 * 60 * 60, +new Date(2012, 09, 29))
  ```
- 定时器不依赖于 D3 过渡或插值，就可以创建一个自定义动画。在 D3 过渡的内部实现中，使用了相同的计时器来生成动画。二者的不同之处在于浏览器的支持情况，即 d3.timer 函数实质上使用了动画帧，否则将转而调用 setTimeout 函数。

6. SVG（w3C）或（D3 gitHub）

1) SVG元素
   - line：直线
   - circle：圆
   - rect：矩形
   - polygon：多边形
   - ellipse：椭圆
   - ployline：折线

   这里着重记录 D3 的 svg 图形生成功能

2) D3 SVG 图形生成器
   svg:path（路径）的 d 属性：大写为绝对坐标，小写为相对坐标
   - `M/m`：将操作位置移动到点 (x y)+
   - `Z/z`：将当前路径闭合
   - `L/l`：将从当前操作点连线到 (x y)+
   - `H/h`：将水平连线到 x+
   - `V/v`：将垂直连线到 y+
   - 三次贝塞尔曲线
     - C/c：使用当前的操作点与终结点 (x y) 绘制曲线，其中起始点的控制点为 (x1, y1)，终结点的控制点为 (x2 y2)
       参数 (x1 y1 x2 y2 x y)+
     - S/s：将从当前操作点绘制曲线至终结点 (x y)， (x2 y2) 为终结点的控制点，起始点的控制点将根据上一个终结点的控制点与当前操作点决定。
       参数 (x2 y2 x y)+
   - 二次贝塞尔曲线
     - Q/q：使用当前的操作点，与终结点 (x y) 绘制贝塞尔曲线，其控制点为 (x1, y1)
       参数 (x1 y1 x y)+
     - T/t：使用当前操作点与终结点 (x y)+ 绘制绘制曲线，其控制点将根据上一个命令的控制点与当前操作点来决定。
       参数 (x y)+

◇ A/a：椭圆曲线

参数 (rx ry x-axis-rotation large-arc-flag sweep-flag x y)+

3) 线条生成器 d3.svg.line

> 虽然称之为线条生成器，但是它和 svg 的 line 元素没有什么关系，是由 svg:path 实现的

```
var data = [
    [
        {x:0, y:5}, {x:1, y:9}, {x:2, y:7},
        {x:3, y:5}, {x:4, y:3}, {x:6, y:4},
        {x:7, y:2}, {x:8, y:3}, {x:9, y:2}
    ],
```

d3.range([*start*,]*stop*[, *step*]) 函数详情

```
    d3.range(10)
```

d3.map([object][, key]) 函数详情

```
        .map(function (i) {
            return {x:i, y:Math.sin(i) + 5}
        })
]
```

生成线条

```
var line = d3.svg.line()
    .x(function (d) {
        return x(d.x)
    })
    .y(function (d) {
        return y(d.y)
    })
```

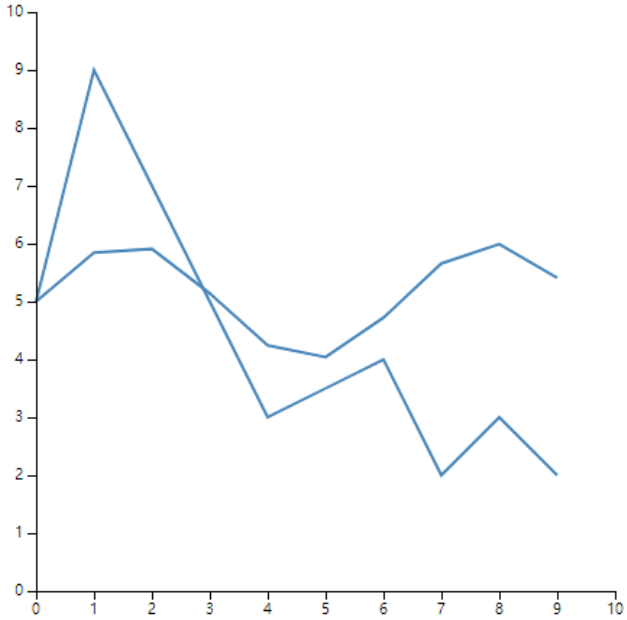
选取集合

```
d3.select("body").append("svg")
    .selectAll("path")
        .data(data)
    .enter()
        .append("path")
        .attr("class", "line")
```

给 path 的 d 属性赋值

```
        .attr("d", function (d) {
            return line(d)
        })
```

假如坐标轴等相关代码已创建完毕，以上代码将生成如下坐标轴中的两条折线条


10
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5
6
7
8
9
10