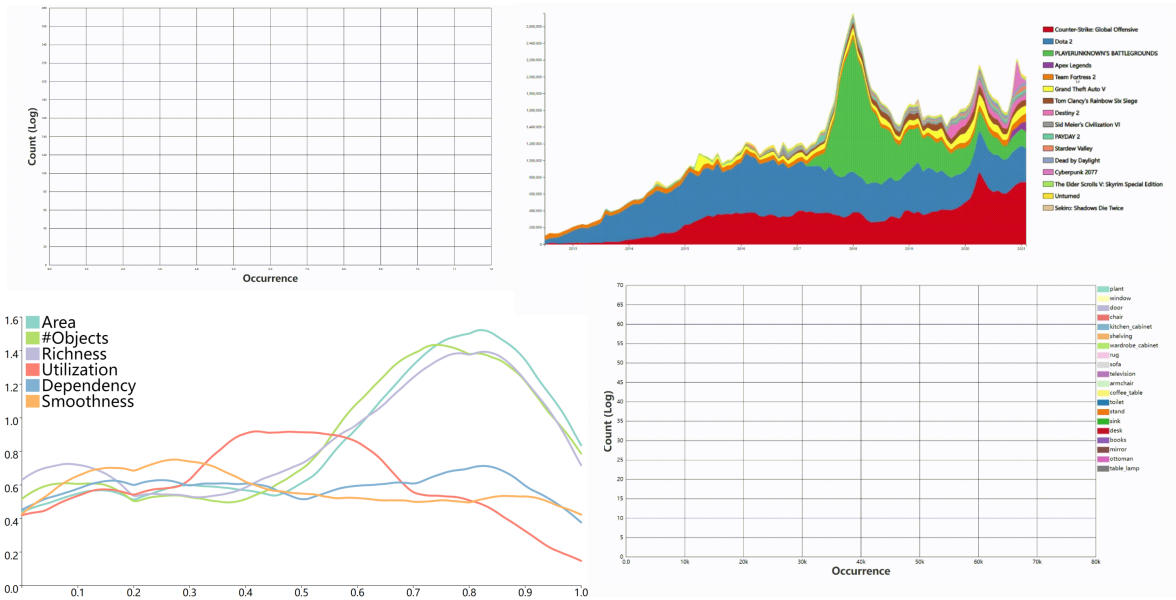
D3.JS – 堆叠与分布

张松海 张少魁
清华大学计算机系
2022

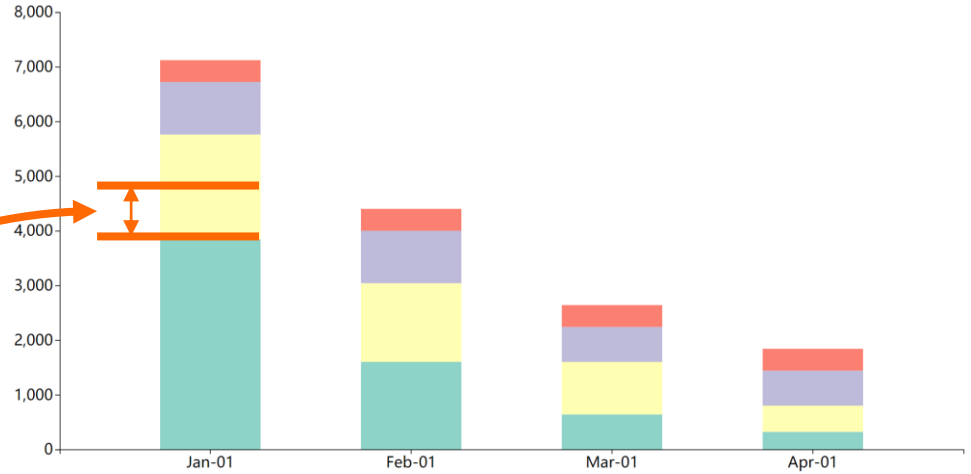
概览

- 使用D3.js的‘stack’接口将数据堆叠：
  - 数据堆叠后的格式。
  - 堆叠的取值与顺序。
- 基于‘stack’实现堆叠柱状图：
  - 嵌套数组的最大值计算。
  - Data-Join嵌套、继承与传递。
- 基于‘stack’实现主题河流：
  - 区域的‘d’属性接口：d3.area()。
- 使用D3.js的‘bin’接口将数据拆分：
  - 数据分布后的格式。
  - 分布的取值与阈值。
- 基于‘bin’实现分布直方图。



使用D3.JS的‘STACK’接口将数据堆叠

- let stack = d3.stack()：
  - 定义一个堆叠**函数**，用于将输入(CSV)数据堆叠。
  - e.g., let stackedData = stack(data)
  - 返回结果以**属性**为单位，将数据分堆。
  - 如下方数据将包含四堆：apples、bananas、cherries、dates。
  - 每堆包含若干对应属性**堆叠后**的位置，即**起点**和**终点**。




```
const naiveData = [
    { month: new Date(2015, 0, 1), apples: 3840, bananas: 1920, cherries: 960, dates: 400 },
    { month: new Date(2015, 1, 1), apples: 1600, bananas: 1440, cherries: 960, dates: 400 },
    { month: new Date(2015, 2, 1), apples: 640, bananas: 960, cherries: 640, dates: 400 },
    { month: new Date(2015, 3, 1), apples: 320, bananas: 480, cherries: 640, dates: 400 }
];
```

数据堆叠后的格式

- 数据原本的形式：四条数据、每条有四个要堆叠的属性。

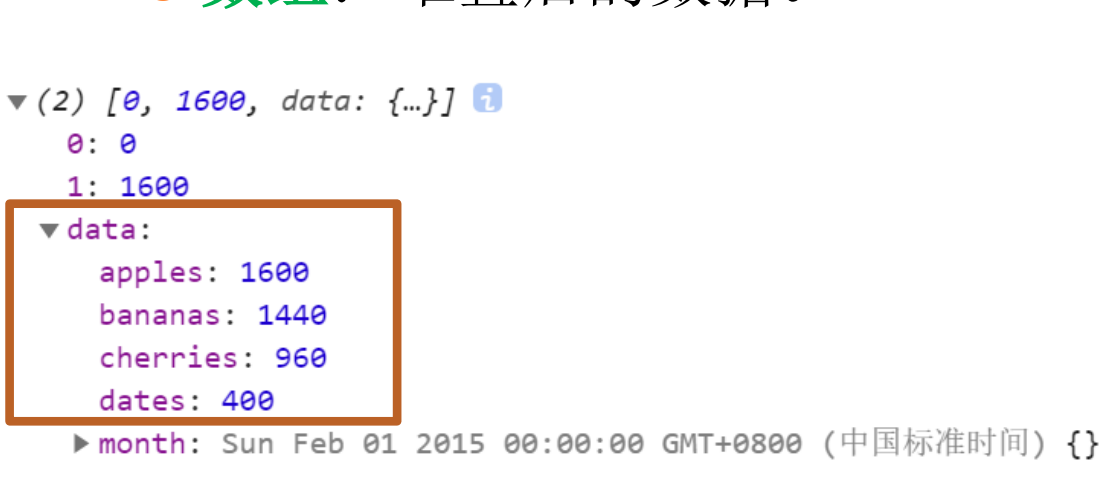
```
const naiveData = [
    { month: new Date(2015, 0, 1), apples: 3840, bananas: 1920, cherries: 960, dates: 400 },
    { month: new Date(2015, 1, 1), apples: 1600, bananas: 1440, cherries: 960, dates: 400 },
    { month: new Date(2015, 2, 1), apples: 640, bananas: 960, cherries: 640, dates: 400 },
    { month: new Date(2015, 3, 1), apples: 320, bananas: 480, cherries: 640, dates: 400 }
];
```

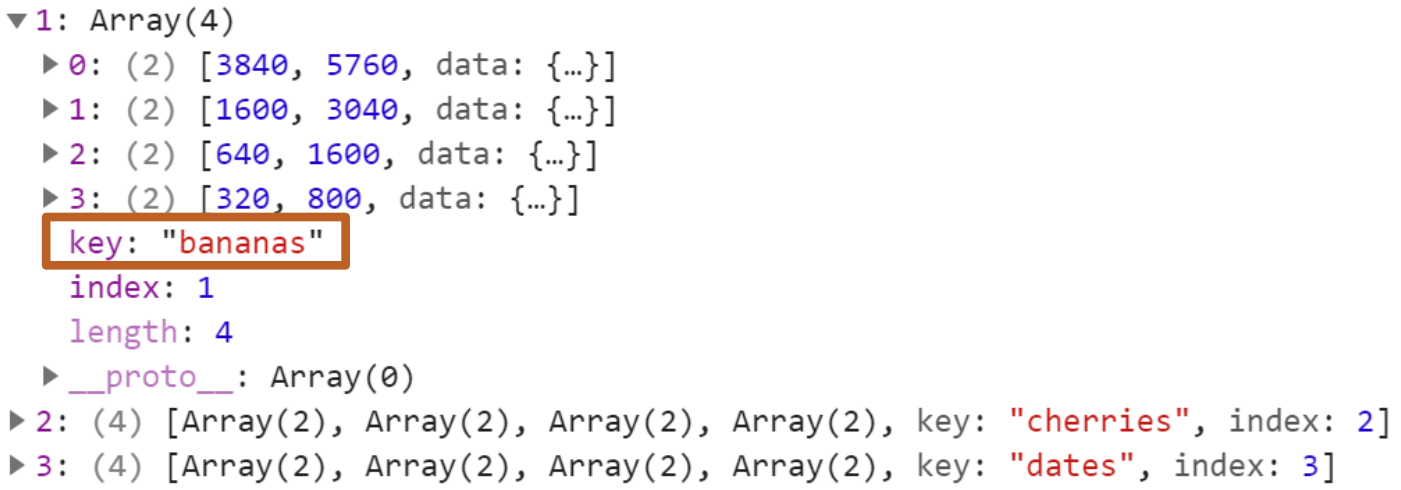
- 返回的形式：

```
[
    [[   0, 3840], [   0, 1600], [   0,  640], [   0,  320]], // apples
    [[3840, 5760], [1600, 3040], [ 640, 1600], [ 320,  800]], // bananas
    [[5760, 6720], [3040, 4000], [1600, 2240], [ 800, 1440]], // cherries
    [[6720, 7120], [4000, 4400], [2240, 2640], [1440, 1840]], // dates
]
```

数据堆叠后的格式

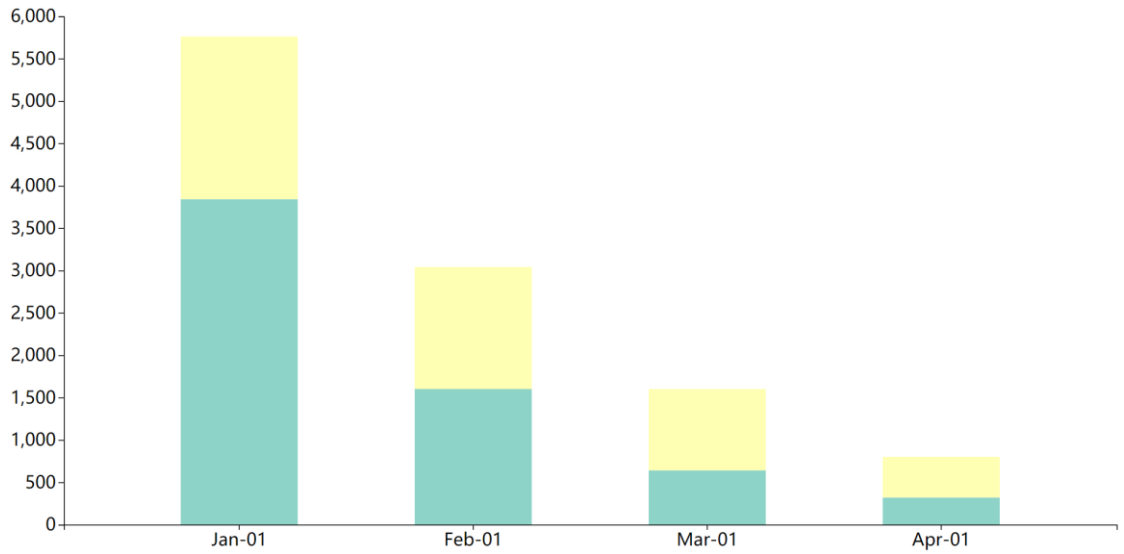
- 堆叠后的数据为 数组的数组的数组 ：
- 数组：长度固定为二，包含堆叠后一特定部分的**起点与终点，包含对原本数据的映射**。
- 数组：长度为原本CSV数据的条目数，包含‘**key**’即数组属于哪个属性。
- 数组：堆叠后的数据。

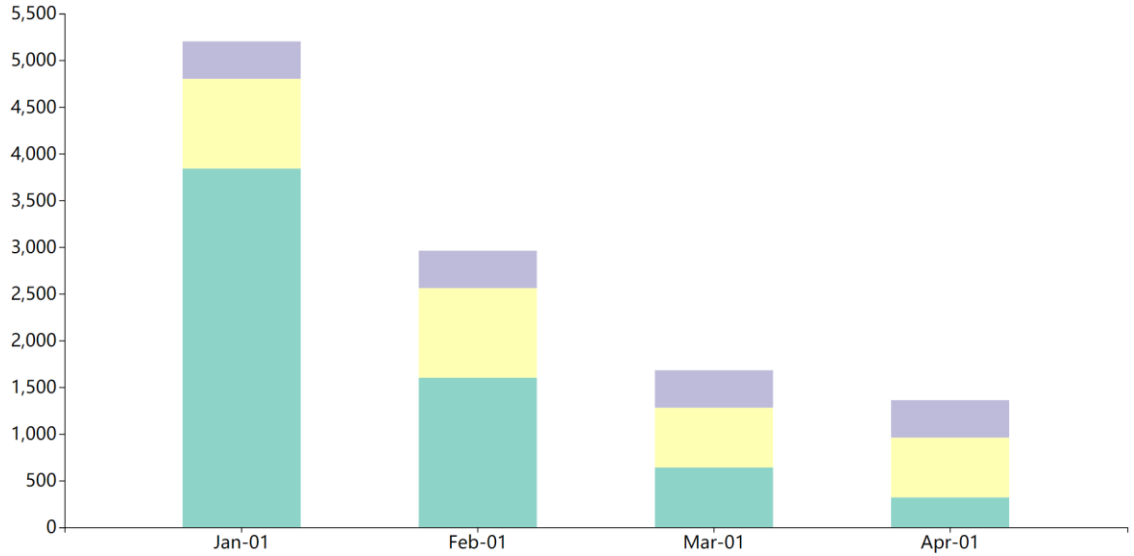




堆叠的取值

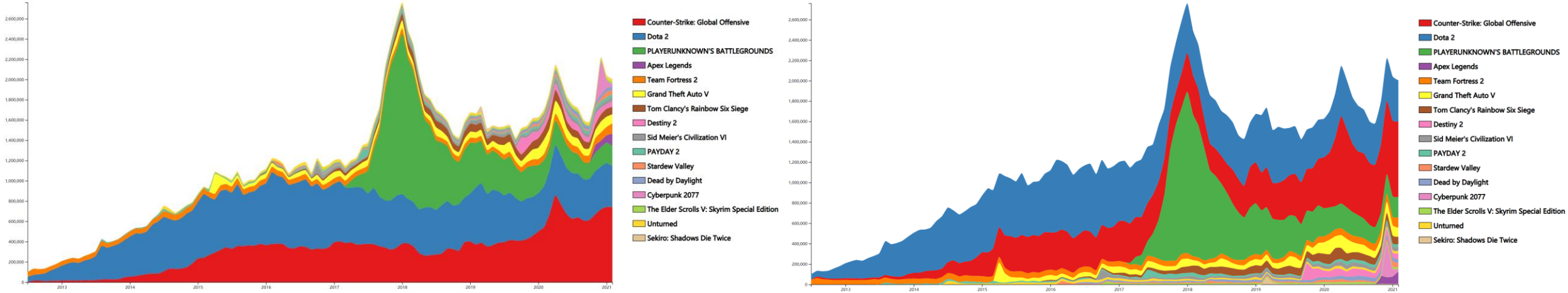
- stack.keys(array):
  - 设置堆叠参考的值（**Key**）有哪些。
  - e.g., stack.keys([“apples”, “bananas”]); // 只堆叠苹果和香蕉。
  - e.g., stack.keys([“apples”, “cherries”, “dates”]); // 堆叠三者。





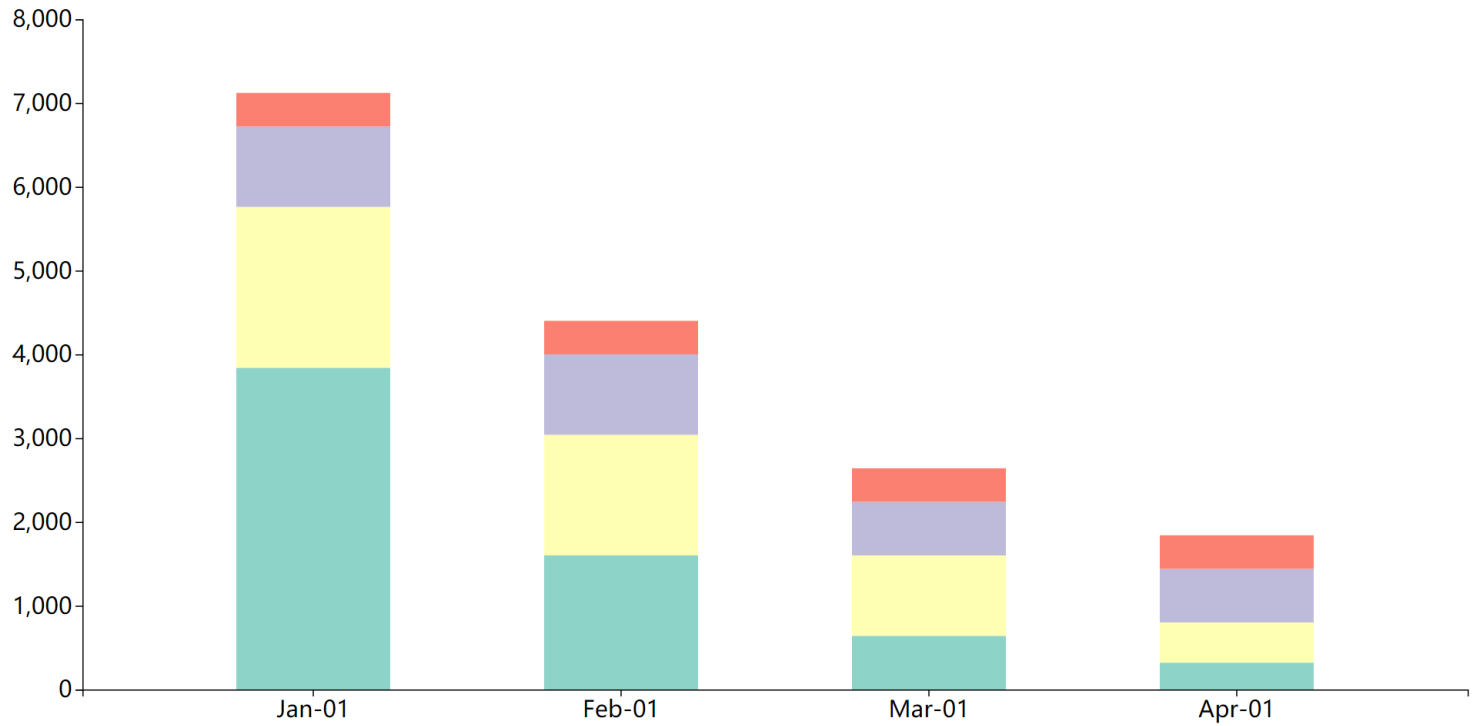
堆叠的顺序

- stack.orders(D3.js的顺序函数)：
  - 设置的每个‘堆’属性排布的顺序。
  - e.g., stack.orders(d3.stackOrderNone); // 随**Key**的顺序。
  - e.g., stack.orders(d3.stackOrderAscending); // 按照堆叠值升序。
  - 根据需求查阅文档即可：https://github.com/d3/d3-shape/blob/v2.1.0/README.md#stack-orders



基于‘STACK’实现堆叠柱状图

- 数据来源：
  - https://github.com/d3/d3-shape/blob/v2.1.0/README.md#stack



取嵌套数组的最大值

- 因需计算比例尺，故需对堆叠后的数据计算最大值与最小值：
  - 而堆叠后的数据又是数组的数组的数组。
  - 需要取嵌套数组的最大、最小值。
- 取嵌套数组的最大值：
  - d3.max([**[1,2,3]**, **[666,233,999]**, **[110,120,119]**], **a** => d3.max(**a**));
- 编程实例：

```
const yScale = d3.scaleLinear()
.domain([0, d3.max(naiveStack, d => d3.max(d, subd => subd[1]))])
.range([innerHeight, 0]).nice();
```

Data-Join嵌套、传递与继承

- 如果 selection.data(data)：
  - data数组会根据图元被分给各个绑定的图元
- 如果 **selection.data(data).attr(...).selectAll(...).data(d => d).join(...)**：
  - 把父节点的数据进一步按数组拆分，分给各个后续的子节点。
- 图元绑定的数据会被‘.append(...)’添加的子节点继承：
  - let gs = svg.selectAll(‘g’).data(data).join(‘g’); // 基于Data-Join添加若干<g>。
  - gs.append(‘rect’); // 为每个<g>添加矩形，每个矩形绑定的数据随<g>。
- 编程实例：

```
g.selectAll('.datagroup').data(naiveStack).join('g')
.attr('class', 'datagroup')
.attr('fill', d => color(d.key))
.selectAll('.datarect').data(d => d).join('rect')
.attr('class', 'datarect')
```

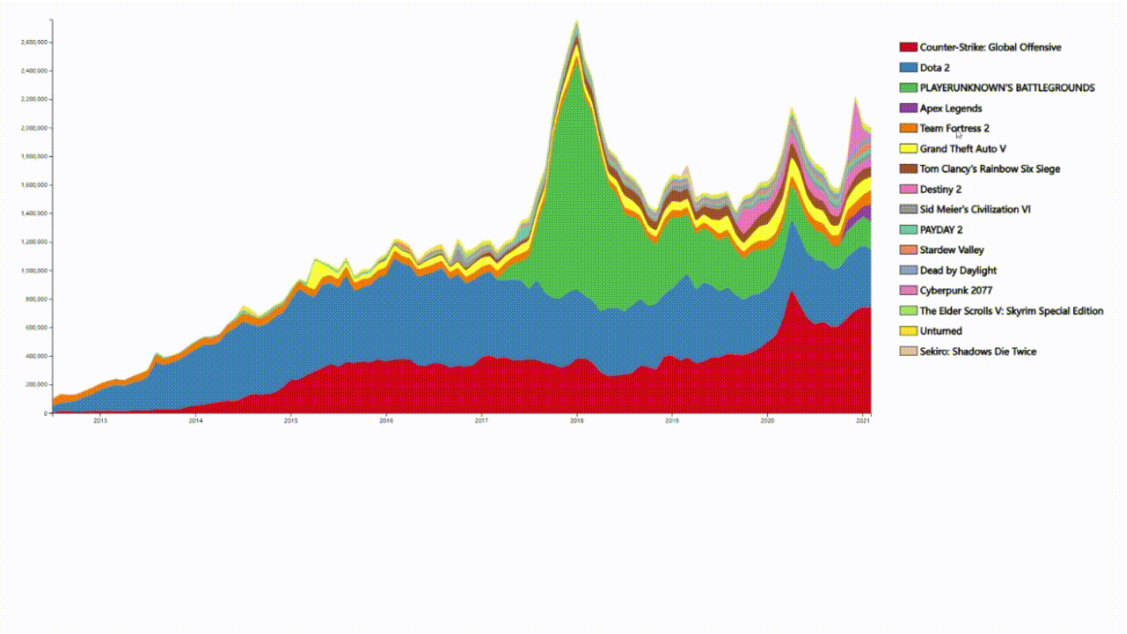
基于‘STACK’实现堆叠柱状图

- 编程实例：

```
// start to do data-join;
g.selectAll('.datagroup').data(naiveStack).join('g')
.attr('class', 'datagroup')
.attr('fill', d => color(d.key))
.selectAll('.datarect').data(d => d).join('rect')
.attr('class', 'datarect')
.attr('y', d => yScale(d[1]))
.attr('x', d => xScale(xValue(d.data)))
.attr('height', d => yScale(d[0]) - yScale(d[1]))
.attr('width', xScale.bandwidth());
```

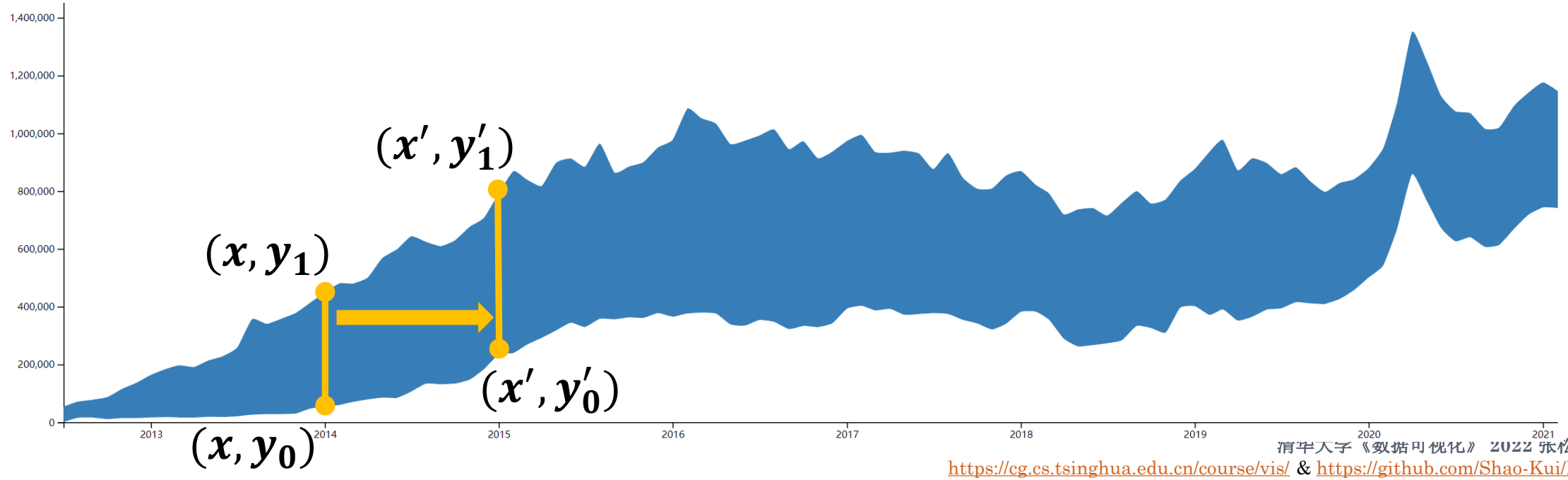
基于‘STACK’实现主题河流

- 数据来源：
  - https://www.kaggle.com/michau96/popularity-of-games-on-steam
- Steam平台各主流游戏热度趋势变化可视化。
- 数据与堆叠的数据：
  - 横轴：时间（年、月）。
  - 纵轴：热度（在线人数）即需要堆叠的内容。



基于‘STACK’实现主题河流

- 使用D3.js实现主题河流，每条‘河流’都是一个图元，如图中每个颜色都只对应一个<path>。
- 主题河流在D3.js中的本质为，随x的变化，在上y_0、下y_1端点分别插值。
  - 故，x、y_0、y_1 需要编程者以数组的形式给出。



D3.AREA(...)

- let path = d3.area()
  - 定义一个‘area’生成函数，输入为数组，输出为<path>的‘d’属性值。
  - path.**x**(d => xScale(d.data.ym)) // 设置x轴的取值随绑定数据的.data.ym属性。
  - path.**y1**(d => yScale(d[1])) // 设置上界y轴的取值随绑定数据的[1]。
  - path.**y0**(d => yScale(d[0])) //设置上界y轴的取值随绑定数据的[1]。
- path.curve(...):
  - 设置‘area’上边界与下边界的拟合方式，同d3.line().curve()
  - e.g., path.curve(d3.curveCardinal.tension(0.5));

基于‘STACK’实现主题河流

- 编程实例：
  - 仍需要对堆叠后的数据取‘嵌套’的最大值与最小值。

```
// define an area generator;
let area = d3.area()
.x(d => xScale(d.data.ym))
.y1(d => yScale(d[1])).y0(d => yScale(d[0]))
.curve(d3.curveCardinal.tension(0.5));

// data-join;
g.selectAll('.riverPath')
.data(stackedData, d => d.key).join('path')
.attr('class', 'riverPath').attr('d', area).attr('fill', d => color(d.key));
```

使用D3.JS的‘BIN’接口将数据拆分

- let histogram = d3.bin():
  - 定义一个函数，用于将输入的数据分区间存放。
  - 结果通常将对象数组转换成对象数组的数组：

```
▶ 0: {modelId: '100', semantic: 'hanger', occur: 2613}
▶ 1: {modelId: '101', semantic: 'kitchen_cabinet', occur: 8499}
▶ 2: {modelId: '102', semantic: 'refrigerator', occur: 18502}
▶ 3: {modelId: '104', semantic: 'desk', occur: 7701}
▶ 4: {modelId: '105', semantic: 'desk', occur: 5716}
▶ 5: {modelId: '106', semantic: 'office_chair', occur: 13731}
▶ 6: {modelId: '107', semantic: 'desk', occur: 4034}
▶ 7: {modelId: '108', semantic: 'desk', occur: 5118}
▶ 8: {modelId: '109', semantic: 'coffee_table', occur: 5395}
▶ 9: {modelId: '110', semantic: 'television', occur: 4051}
▶ 10: {modelId: '111', semantic: 'television', occur: 12345}
▶ 11: {modelId: '112', semantic: 'television', occur: 8150}
▶ 12: {modelId: '113', semantic: 'television', occur: 6605}
▶ 13: {modelId: '114', semantic: 'television', occur: 17944}
▶ 14: {modelId: '115', semantic: 'tv_stand', occur: 4916}
▶ 15: {modelId: '116', semantic: 'tv_stand', occur: 5682}
▶ 16: {modelId: '117', semantic: 'tv_stand', occur: 1907}
▶ 17: {modelId: '118', semantic: 'hanger', occur: 3801}
▶ 18: {modelId: '119', semantic: 'coffee_table', occur: 2592}
▶ 19: {modelId: '120', semantic: 'laptop', occur: 27233}
▶ 20: {modelId: '121', semantic: 'wardrobe_cabinet', occur: 15738}
```

对象的数组，通常对应d3.csv读取后的数据

```
▼ 5: Array(12)
  ▶ 0: {modelId: '120', semantic: 'laptop', occur: 27233}
  ▶ 1: {modelId: '133', semantic: 'door', occur: 27964}
  ▶ 2: {modelId: '149', semantic: 'cup', occur: 26891}
  ▶ 3: {modelId: '225', semantic: 'air_conditioner', occur: 28613}
  ▶ 4: {modelId: '244', semantic: 'glass', occur: 28476}
  ▶ 5: {modelId: '258', semantic: 'vase', occur: 26051}
  ▶ 6: {modelId: '280', semantic: 'switch', occur: 29792}
  ▶ 7: {modelId: '364', semantic: 'car', occur: 28141}
  ▶ 8: {modelId: '627', semantic: 'chandelier', occur: 25056}
  ▶ 9: {modelId: '753', semantic: 'window', occur: 26857}
  ▶ 10: {modelId: '754', semantic: 'window', occur: 29496}
  ▶ 11: {modelId: 's__487', semantic: 'chandelier', occur: 28385}
    x0: 25000
    x1: 30000
    length: 12
  ▶ [[Prototype]]: Array(0)
▼ 6: Array(7)
  ▶ 0: {modelId: '210', semantic: 'window', occur: 34371}
  ▶ 1: {modelId: '211', semantic: 'window', occur: 34776}
  ▶ 2: {modelId: '212', semantic: 'window', occur: 32388}
  ▶ 3: {modelId: '214', semantic: 'door', occur: 32663}
  ▶ 4: {modelId: '377', semantic: 'outdoor_lamp', occur: 31615}
  ▶ 5: {modelId: '623', semantic: 'plant', occur: 32693}
  ▶ 6: {modelId: '726', semantic: 'plant', occur: 30469}
    x0: 30000
    x1: 35000
    length: 7
  ▶ [[Prototype]]: Array(0)
```

值在25000-30000的对象被分到该数组内部

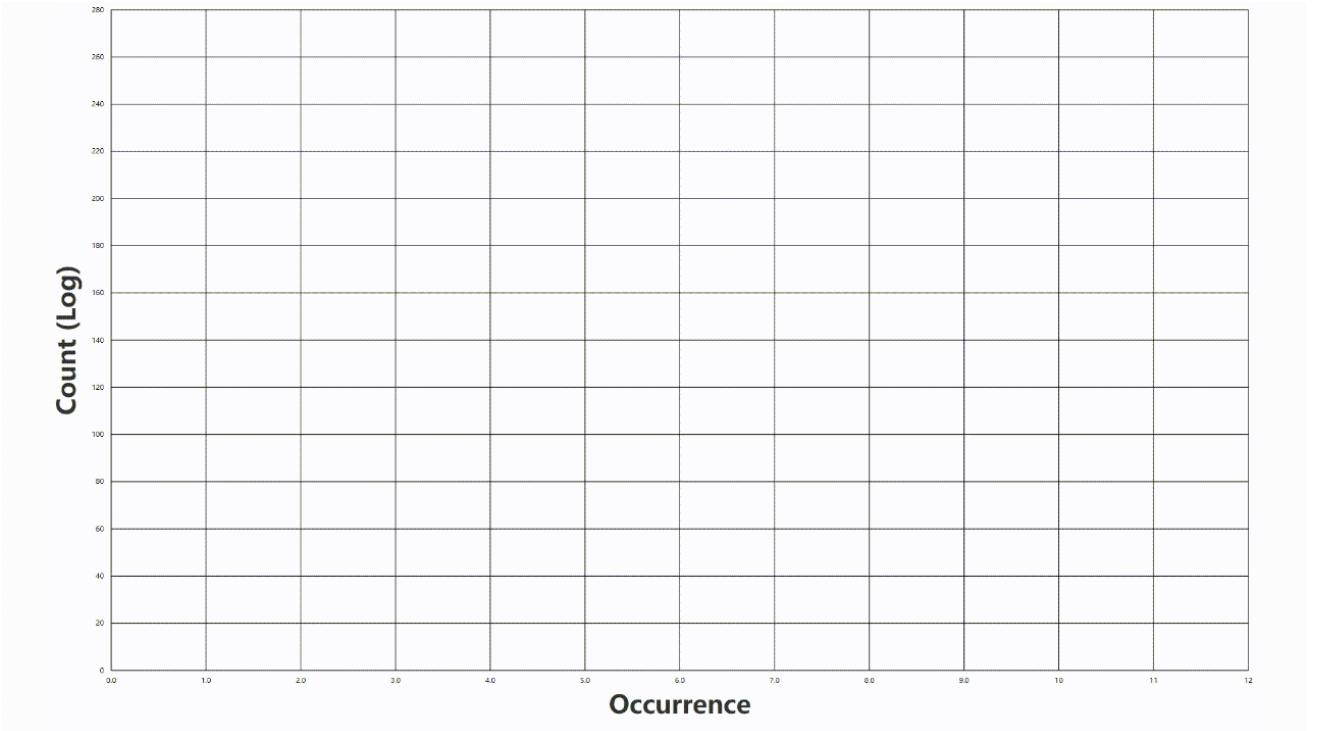
使用D3.JS的‘STACK’接口将数据拆分

- 对象通常包含多个值，仍需基于函数的形式告诉D3.js，如何取对象的值：
- d3.bin().value(d => d.occur);
- 即：对于每个对象，参考其‘occur’属性值来决定分到哪一堆。

- 有多少堆？每个堆的范围如何选取？
- d3.bin().value(…).thresholds([x1, x2, x3, x4, …, xn]);
- 回忆d3.coutour…我们输入数组作为阈值。
- 通常使用scale.ticks(n)接口来直接导出阈值。
- 如：histogram.thresholds(xScale.ticks(20));

基于‘BIN’实现分布直方图

- 数据来源：
  - Song S, Yu F, Zeng A, et al. Semantic scene completion from a single depth image[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2017: 1746-1754.

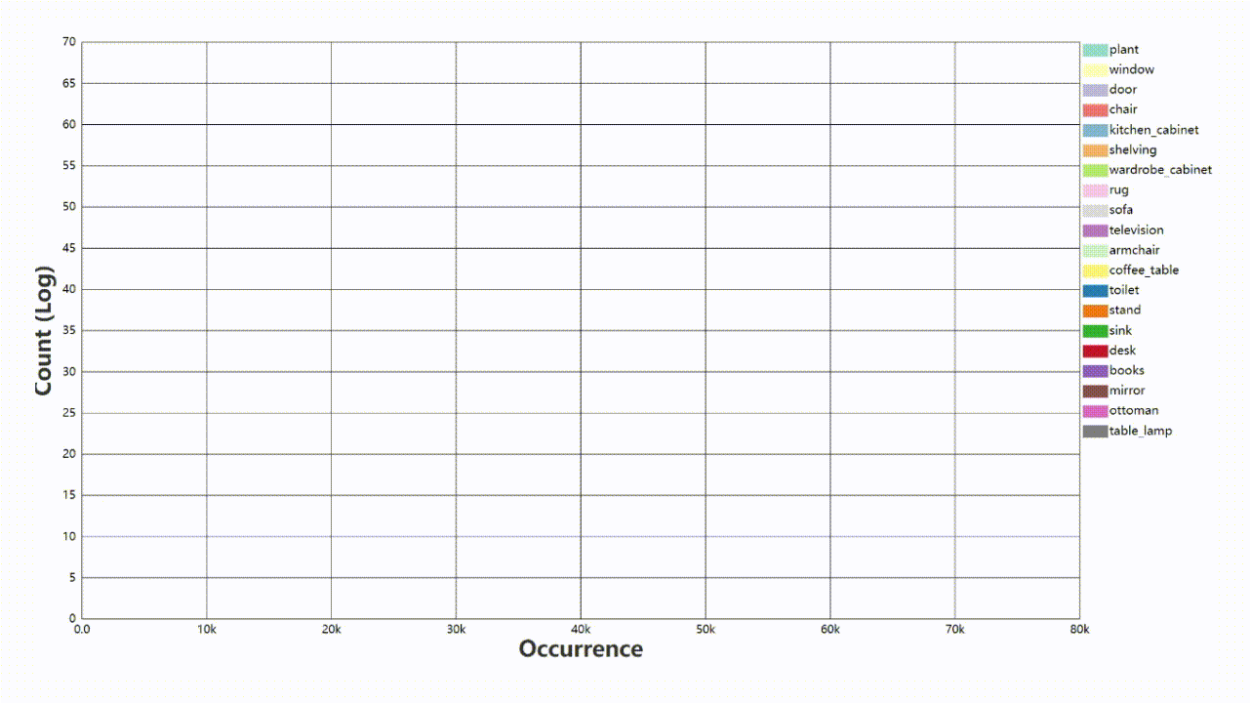


TIP：SCALEORDINAL的特殊调用形式?

- **color** = d3.scaleOrdinal()
- .range(d3.schemePaired.concat(d3.schemeCategory10));
- … .attr('fill', (d, i) => **color**(i))
- 注意到，基于‘bin’实现分布直方图时，我们并没有设置scaleOrdinal比例尺的定义域（domain）…
- 为什么结果仍然是可以跑通的？
- scaleOrdinal本质上在维护一个映射，如果没有显示输入定义域，那么值域会自动基于接受的数据**动态分配**。

Tip：D3.js的数据处理方法

- D3.js也含有其他数据预处理的方法
- d3.layout.cloud(): 用于将文本绘制到画布的各个位置
  - 常用于绘制**词云**。
- 甚至我们可以将数据处理的接口相结合，如实现堆叠分布直方图：



TIP：比例尺不满足‘结合律’

- 计算矩形高度‘height’时，需基于堆叠数据做减法。
- 比例尺不满足‘结合律’：
  - **let linearScale = d3.scaleLinear().domain([1,10]).range([800, 1000])**
  - linearScale(8-6) // 822.22
  - linearScale(8) - linearScale(6) // 44.44
- 故，尽可能先用比例尺映射数据，再做运算！（绿色框）

```
g.selectAll('.datagroup').data(naiveStack).join('g')
.attr('class', 'datagroup')
.attr('fill', d => color(d.key))
.selectAll('.datarect').data(d => d).join('rect')
.attr('class', 'datarect')
.attr('y', d => yScale(d[1]))
.attr('x', d => xScale(xValue(d.data)))
.attr('height', d => yScale(d[0]) - yScale(d[1]))
.attr('width', xScale.bandwidth());
```

Tip：分布密度图（Density Map）？

- 绘制密度图（Density Map）并不需要d3.bin()接口的使用。
- 取而代之，需要自行做‘核密度估计’。
- 核密度估计得到的核函数的累加会被用于插值...
- 插值？d3.line().x(...).y(...).curve(d3.curveBasis);