ECharts 3 教程

书栈(BookStack.CN)

目　录

致谢

当前文档 《ECharts 3 教程》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-03-16。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

内容来源：ECharts 官网 https://echarts.baidu.com/index.html

文档地址：http://www.bookstack.cn/books/ECharts3Tutorial

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

5 分钟上手 ECharts

获取 ECharts

你可以通过以下几种方式获取 ECharts。

- 从官网下载界面选择你需要的版本下载，根据开发者功能和体积上的需求，我们提供了不同打包的下载，如果你在体积上没有要求，可以直接下载完整版本。开发环境建议下载源代码版本，包含了常见的错误提示和警告。
- 在 ECharts 的 GitHub 上下载最新的 `release` 版本，解压出来的文件夹里的 `dist` 目录里可以找到最新版本的 echarts 库。
- 通过 npm 获取 echarts， `npm install echarts –save` ，详见“在 webpack 中使用 echarts”
- cdn 引入，你可以在 cdnjs，@latest/dist/">npmcdn 或者国内的 bootcdn 上找到 ECharts 的最新版本。

引入 ECharts

ECharts 3 开始不再强制使用 AMD 的方式按需引入，代码里也不再内置 AMD 加载器。因此引入方式简单了很多，只需要像普通的 JavaScript 库一样用 script 标签引入。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <!-- 引入 ECharts 文件 -->
    <script src="echarts.min.js"></script>
</head>
</html>
```

绘制一个简单的图表

在绘图前我们需要为 ECharts 准备一个具备高宽的 DOM 容器。

```
<body>
    <!-- 为 ECharts 准备一个具备大小（宽高）的 DOM -->
    <div id="main" style="width: 600px;height:400px;"></div>
</body>
```

然后就可以通过 `echarts.init` 方法初始化一个 echarts 实例并通过 `setOption` 方法生成一个简单的柱状图，下面是完整代码。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>ECharts</title>
    <!-- 引入 echarts.js -->
    <script src="echarts.min.js"></script>
</head>
<body>
    <!-- 为ECharts准备一个具备大小（宽高）的Dom -->
    <div id="main" style="width: 600px;height:400px;"></div>
    <script type="text/javascript">
        // 基于准备好的dom，初始化echarts实例
        var myChart = echarts.init(document.getElementById('main'));

        // 指定图表的配置项和数据
        var option = {
            title: {
                text: 'ECharts 入门示例'
            },
            tooltip: {},
            legend: {
                data:['销量']
            },
            xAxis: {
                data: ["衬衫","羊毛衫","雪纺衫","裤子","高跟鞋","袜子"]
            },
            yAxis: {},
            series: [{
                name: '销量',
                type: 'bar',
                data: [5, 20, 36, 10, 10, 20]
            }]
        };

        // 使用刚指定的配置项和数据显示图表。
        myChart.setOption(option);
    </script>
```

```
39.  </body>
40.  </html>
```

这样你的第一个图表就诞生了！ https://echarts.baidu.com/gallery/view.html?c=doc-example/getting-started&reset=1&edit=1 你也可以直接进入 ECharts Gallery 中查看编辑示例

ECharts 中的事件和行为

在 ECharts 的图表中用户的操作将会触发相应的事件。开发者可以监听这些事件，然后通过回调函数做相应的处理，比如跳转到一个地址，或者弹出对话框，或者做数据下钻等等。

在 ECharts 3 中绑定事件跟 2 一样都是通过 on 方法，但是事件名称比 2 更加简单了。ECharts 3 中，事件名称对应 DOM 事件名称，均为小写的字符串，如下是一个绑定点击操作的示例。

```
myChart.on('click', function (params) {
    // 控制台打印数据的名称
    console.log(params.name);
});
```

在 ECharts 中事件分为两种类型，一种是用户鼠标操作点击，或者 hover 图表的图形时触发的事件，还有一种是用户在使用可以交互的组件后触发的行为事件，例如在切换图例开关时触发的 'legendselectchanged' 事件（这里需要注意切换图例开关是不会触发 'legendselected' 事件的），数据区域缩放时触发的 'datazoom' 事件等等。

鼠标事件的处理

ECharts 支持常规的鼠标事件类型，包括 'click' 、 'dblclick' 、 'mousedown' 、 'mousemove' 、 'mouseup' 、 'mouseover' 、 'mouseout' 、 'globalout' 、 'contextmenu' 事件。下面先来看一个简单的点击柱状图后打开相应的百度搜索页面的示例。

```
// 基于准备好的dom，初始化ECharts实例
var myChart = echarts.init(document.getElementById('main'));

// 指定图表的配置项和数据
var option = {
    xAxis: {
        data: ["衬衫","羊毛衫","雪纺衫","裤子","高跟鞋","袜子"]
    },
    yAxis: {},
    series: [{
        name: '销量',
        type: 'bar',
        data: [5, 20, 36, 10, 10, 20]
    }]
};
```

```
// 使用刚指定的配置项和数据显示图表。
myChart.setOption(option);
// 处理点击事件并且跳转到相应的百度搜索页面
myChart.on('click', function (params) {
    window.open('https://www.baidu.com/s?wd=' +
encodeURIComponent(params.name));
});
```

所有的鼠标事件包含参数 `params`，这是一个包含点击图形的数据信息的对象，如下格式：

```
{
    // 当前点击的图形元素所属的组件名称,
    // 其值如 'series'、'markLine'、'markPoint'、'timeLine' 等。
    componentType: string,
    // 系列类型。值可能为：'line'、'bar'、'pie' 等。当 componentType 为 'series' 时有意义。
    seriesType: string,
    // 系列在传入的 option.series 中的 index。当 componentType 为 'series' 时有意义。
    seriesIndex: number,
    // 系列名称。当 componentType 为 'series' 时有意义。
    seriesName: string,
    // 数据名，类目名
    name: string,
    // 数据在传入的 data 数组中的 index
    dataIndex: number,
    // 传入的原始数据项
    data: Object,
    // sankey、graph 等图表同时含有 nodeData 和 edgeData 两种 data,
    // dataType 的值会是 'node' 或者 'edge'，表示当前点击在 node 还是 edge 上。
    // 其他大部分图表中只有一种 data，dataType 无意义。
    dataType: string,
    // 传入的数据值
    value: number|Array
    // 数据图形的颜色。当 componentType 为 'series' 时有意义。
    color: string
}
```

如何区分鼠标点击到了哪里：

```
myChart.on('click', function (params) {
    if (params.componentType === 'markPoint') {
```

```
        // 点击到了 markPoint 上
        if (params.seriesIndex === 5) {
            // 点击到了 index 为 5 的 series 的 markPoint 上。
        }
    }
    else if (params.componentType === 'series') {
        if (params.seriesType === 'graph') {
            if (params.dataType === 'edge') {
                // 点击到了 graph 的 edge（边）上。
            }
            else {
                // 点击到了 graph 的 node（节点）上。
            }
        }
    }
});
```

使用 `query` 只对指定的组件的图形元素的触发回调：

```
chart.on(eventName, query, handler);
```

`query` 可为 `string` 或者 `Object` 。

如果为 `string` 表示组件类型。格式可以是 'mainType' 或者 'mainType.subType'。例如：

```
chart.on('click', 'series', function () {...});
chart.on('click', 'series.line', function () {...});
chart.on('click', 'dataZoom', function () {...});
chart.on('click', 'xAxis.category', function () {...});
```

如果为 `Object` ，可以包含以下一个或多个属性，每个属性都是可选的：

```
{
    <mainType>Index: number // 组件 index
    <mainType>Name: string // 组件 name
    <mainType>Id: string // 组件 id
    dataIndex: number // 数据项 index
    name: string // 数据项 name
    dataType: string // 数据项 type，如关系图中的 'node', 'edge'
    element: string // 自定义系列中的 el 的 name
}
```

例如：

```
chart.setOption({
    // ...
    series: [{
        name: 'uuu'
        // ...
    }]
});
chart.on('mouseover', {seriesName: 'uuu'}, function () {
    // series name 为 'uuu' 的系列中的图形元素被 'mouseover' 时，此方法被回调。
});
```

例如：

```
chart.setOption({
    // ...
    series: [{
        // ...
    }, {
        // ...
        data: [
            {name: 'xx', value: 121},
            {name: 'yy', value: 33}
        ]
    }]
});
chart.on('mouseover', {seriesIndex: 1, name: 'xx'}, function () {
    // series index 1 的系列中的 name 为 'xx' 的元素被 'mouseover' 时，此方法被回调。
});
```

例如：

```
chart.setOption({
    // ...
    series: [{
        type: 'graph',
        nodes: [{name: 'a', value: 10}, {name: 'b', value: 20}],
        edges: [{source: 0, target: 1}]
    }]
});
```

```
chart.on('click', {dataType: 'node'}, function () {
    // 关系图的节点被点击时此方法被回调。
});
chart.on('click', {dataType: 'edge'}, function () {
    // 关系图的边被点击时此方法被回调。
});
```

例如：

```
chart.setOption({
    // ...
    series: {
        // ...
        type: 'custom',
        renderItem: function (params, api) {
            return {
                type: 'group',
                children: [{
                    type: 'circle',
                    name: 'my_el',
                    // ...
                }, {
                    // ...
                }]
            }
        },
        data: [[12, 33]]
    }
})
chart.on('mouseup', {element: 'my_el'}, function () {
    // name 为 'my_el' 的元素被 'mouseup' 时，此方法被回调。
});
```

你可以在回调函数中获得这个对象中的数据名、系列名称后在自己的数据仓库中索引得到其它的信息候更新图表，显示浮层等等，如下示例代码：

```
myChart.on('click', function (parmas) {
    $.get('detail?q=' + params.name, function (detail) {
        myChart.setOption({
            series: [{
                name: 'pie',
                // 通过饼图表现单个柱子中的数据分布
```

```
                data: [detail.data]
            }]
        });
    });
});
```

组件交互的行为事件

在 ECharts 中基本上所有的组件交互行为都会触发相应的事件，常用的事件和事件对应参数在 events 文档中有列出。

下面是监听一个图例开关的示例：

```
// 图例开关的行为只会触发 legendselectchanged 事件
myChart.on('legendselectchanged', function (params) {
    // 获取点击图例的选中状态
    var isSelected = params.selected[params.name];
    // 在控制台中打印
    console.log((isSelected ? '选中了' : '取消选中了') + '图例' + params.name);
    // 打印所有图例的状态
    console.log(params.selected);
});
```

代码触发 ECharts 中组件的行为

上面提到诸如 `'legendselectchanged'` 事件会由组件交互的行为触发，那除了用户的交互操作，有时候也会有需要在程序里调用方法触发图表的行为，诸如显示 tooltip，选中图例。

在 ECharts 2.x 是通过 `myChart.component.tooltip.showTip` 这种形式调用相应的接口触发图表行为，入口很深，而且涉及到内部组件的组织。相对地，在 ECharts 3 里改为通过调用 `myChart.dispatchAction({ type: '' })` 触发图表行为，统一管理了所有动作，也可以方便地根据需要去记录用户的行为路径。

常用的动作和动作对应参数在 action 文档中有列出。

下面示例演示了如何通过 `dispatchAction` 去轮流高亮饼图的每个扇形。
https://echarts.baidu.com/gallery/view.html?c=doc-example/pie-highlight&edit=1&reset=1

ECharts 中的样式简介

本文主要是大略概述，用哪些方法，可以设置设置样式，改变图形元素或者文字的颜色、明暗、大小等。

> 之所以用“样式”这种可能不很符合数据可视化思维的词，是因为，比较通俗易懂。

本文介绍这几种方式，他们的功能范畴可能会有交叉（即同一种细节的效果可能可以用不同的方式实现），但是他们各有各的场景偏好。

- 颜色主题（Theme）
- 调色盘
- 直接样式设置（itemStyle、lineStyle、areaStyle、label、…）
- 视觉映射（visualMap） 其他关于样式的文章，参见：个性化图表的样式，数据的视觉映射。

颜色主题（Theme）

最简单的更改全局样式的方式，是直接采用颜色主题（theme）。例如，在 示例集合 中，可以选择“Theme”，直接看到采用主题的效果。

ECharts4 开始，除了一贯的默认主题外，新内置了两套主题，分别为 `'light'` 和 `'dark'` 。可以这么来使用它们：

```
1.  var chart = echarts.init(dom, 'light');
```

或者

```
1.  var chart = echarts.init(dom, 'dark');
```

其他的主题，没有内置在 ECharts 中，需要自己加载。这些主题可以在 主题编辑器 里访问到。也可以使用这个主题编辑器，自己编辑主题。下载下来的主题可以这样使用：

如果主题保存为 JSON 文件，那么可以自行加载和注册，例如：

```
1.  // 假设主题名称是 "vintage"
2.  $.getJSON('xxx/xxx/vintage.json', function (themeJSON) {
3.      echarts.registerTheme('vintage', JSON.parse(themeJSON))
4.      var chart = echarts.init(dom, 'vintage');
5.  });
```

如果保存为 UMD 格式的 JS 文件，那么支持了自注册，直接引入 JS 文件即可：

```
// HTML 引入 vintage.js 文件后（假设主题名称是 "vintage"）
var chart = echarts.init(dom, 'vintage');
// ...
```

调色盘

调色盘，可以在 option 中设置。它给定了一组颜色，图形、系列会自动从其中选择颜色。可以设置全局的调色盘，也可以设置系列自己专属的调色盘。

```
option = {
    // 全局调色盘。
    color: ['#c23531','#2f4554', '#61a0a8', '#d48265', '#91c7ae','#749f83',
'#ca8622', '#bda29a','#6e7074', '#546570', '#c4ccd3'],

    series: [{
        type: 'bar',
        // 此系列自己的调色盘。
        color:
['#dd6b66','#759aa0','#e69d87','#8dc1a9','#ea7e53','#eedd78','#73a373','#73b9bc',
 '#91ca8c','#f49f42'],
        ...
    }, {
        type: 'pie',
        // 此系列自己的调色盘。
        color: ['#37A2DA', '#32C5E9', '#67E0E3', '#9FE6B8',
'#FFDB5C','#ff9f7f', '#fb7293', '#E062AE', '#E690D1', '#e7bcf3', '#9d96f5',
'#8378EA', '#96BFFF'],
        ...
    }]
}
```

直接的样式设置 itemStyle, lineStyle, areaStyle, label, …

直接的样式设置是比较常用设置方式。纵观 ECharts 的 option 中，很多地方可以设置 itemStyle、lineStyle、areaStyle、label 等等。这些的地方可以直接设置图形元素的颜色、线宽、点的大小、标签的文字、标签的样式等等。

一般来说，ECharts 的各个系列和组件，都遵从这些命名习惯，虽然不同图表和组件

中，`itemStyle`、`label` 等可能出现在不同的地方。

直接样式设置的另一篇介绍，参见 个性化图表的样式。

高亮的样式：emphasis

在鼠标悬浮到图形元素上时，一般会出现高亮的样式。默认情况下，高亮的样式是根据普通样式自动生成的。但是高亮的样式也可以自己定义，主要是通过 `emphasis` 属性来定制。`emphsis` 中的结构，和普通样式的结构相同，例如：

```
option = {
    series: {
        type: 'scatter',

        // 普通样式。
        itemStyle: {
            // 点的颜色。
            color: 'red'
        },
        label: {
            show: true,
            // 标签的文字。
            formatter: 'This is a normal label.'
        },

        // 高亮样式。
        emphasis: {
            itemStyle: {
                // 高亮时点的颜色。
                color: 'blue'
            },
            label: {
                show: true,
                // 高亮时标签的文字。
                formatter: 'This is a emphasis label.'
            }
        }
    }
}
```

注意：在 ECharts4 以前，高亮和普通样式的写法，是这样的：

```
option = {
    series: {
        type: 'scatter',

        itemStyle: {
            // 普通样式。
            normal: {
                // 点的颜色。
                color: 'red'
            },
            // 高亮样式。
            emphasis: {
                // 高亮时点的颜色。
                color: 'blue'
            }
        },

        label: {
            // 普通样式。
            normal: {
                show: true,
                // 标签的文字。
                formatter: 'This is a normal label.'
            },
            // 高亮样式。
            emphasis: {
                show: true,
                // 高亮时标签的文字。
                formatter: 'This is a emphasis label.'
            }
        }
    }
}
```

这种写法 仍然被兼容，但是，不再推荐。事实上，多数情况下，使用者只会配置普通状态下的样式，而使用默认的高亮样式。所以在 ECharts4 中，支持不写 `normal` 的配置方法（即本文开头的那种写法），使得配置项更扁平简单。

通过 visualMap 组件设定样式

visualMap 组件 能指定数据到颜色、图形尺寸的映射规则，详见 数据的视觉映射。

个性化图表的样式

ECharts 提供了丰富的自定义配置选项，并且能够从全局、系列、数据三个层级去设置数据图形的样式。下面我们来看如何使用 ECharts 实现下面这个南丁格尔图：

- https://echarts.baidu.com/gallery/view.html?c=doc-example/tutorial-styling-step5&edit=1&reset=1
- https://echarts.baidu.com/gallery/view.html?c=doc-example/tutorial-styling-step5&edit=1&reset=1

绘制南丁格尔图

5分钟上手ECharts 中讲了如何绘制一个简单的柱状图，这次要画的是饼图，饼图主要是通过扇形的弧度表现不同类目的数据在总和中的占比，它的数据格式比柱状图更简单，只有一维的数值，不需要给类目。因为不在直角坐标系上，所以也不需要 `xAxis` ， `yAxis` 。

```
myChart.setOption({
    series : [
        {
            name: '访问来源',
            type: 'pie',
            radius: '55%',
            data:[
                {value:235, name:'视频广告'},
                {value:274, name:'联盟广告'},
                {value:310, name:'邮件营销'},
                {value:335, name:'直接访问'},
                {value:400, name:'搜索引擎'}
            ]
        }
    ]
})
```

上面代码就能画出一个简单的饼图： https://echarts.baidu.com/gallery/view.html?c=doc-example/tutorial-styling-step0&edit=1&reset=1

这里 `data` 属性值不像入门教程里那样每一项都是单个数值，而是一个包含 `name` 和 `value` 属性的对象，ECharts 中的数据项都是既可以只设成数值，也可以设成一个包含有名称、该数据图形的样式配置、标签配置的对象，具体见 `data` 文档。

ECharts 中的饼图也支持通过设置 `roseType` 显示成南丁格尔图。

```
roseType: 'angle'
```

南丁格尔图会通过半径表示数据的大小。

https://echarts.baidu.com/gallery/view.html?c=doc-example/tutorial-styling-step1&edit=1&reset=1

阴影的配置

ECharts 中有一些通用的样式，诸如阴影、透明度、颜色、边框颜色、边框宽度等，这些样式一般都会在系列的 `itemStyle` 里设置。例如阴影的样式可以通过下面几个配置项设置：

```
itemStyle: {
    // 阴影的大小
    shadowBlur: 200,
    // 阴影水平方向上的偏移
    shadowOffsetX: 0,
    // 阴影垂直方向上的偏移
    shadowOffsetY: 0,
    // 阴影颜色
    shadowColor: 'rgba(0, 0, 0, 0.5)'
}
```

加上阴影后的效果：

https://echarts.baidu.com/gallery/view.html?c=doc-example/tutorial-styling-step2&edit=1&reset=1

`itemStyle` 的 `emphasis` 是鼠标 hover 时候的高亮样式。这个示例里是正常的样式下加阴影，但是可能更多的时候是 hover 的时候通过阴影突出。

```
itemStyle: {
    emphasis: {
        shadowBlur: 200,
        shadowColor: 'rgba(0, 0, 0, 0.5)'
    }
}
```

深色背景和浅色标签

现在我们需要把整个主题改成开始的示例中那样的深色主题，这就需要改背景色和文本颜色。

背景色是全局的，所以直接在 option 下设置 backgroundColor

```
setOption({
    backgroundColor: '#2c343c'
})
```

文本的样式可以设置全局的 textStyle。

```
setOption({
    textStyle: {
        color: 'rgba(255, 255, 255, 0.3)'
    }
})
```

也可以每个系列分别设置，每个系列的文本设置在 label.textStyle。

```
label: {
    textStyle: {
        color: 'rgba(255, 255, 255, 0.3)'
    }
}
```

饼图的话还要将标签的视觉引导线的颜色设为浅色。

```
labelLine: {
    lineStyle: {
        color: 'rgba(255, 255, 255, 0.3)'
    }
}
```

如下：

https://echarts.baidu.com/gallery/view.html?c=doc-example/tutorial-styling-step3&edit=1&reset=1

跟 itemStyle 一样， label 和 labelLine 的样式也有 emphasis 状态。

设置扇形的颜色

扇形的颜色也是在 itemStyle 中设置：

```
itemStyle: {
    // 设置扇形的颜色
    color: '#c23531',
    shadowBlur: 200,
    shadowColor: 'rgba(0, 0, 0, 0.5)'
}
```

https://echarts.baidu.com/gallery/view.html?c=doc-example/tutorial-styling-step4&edit=1&reset=1

跟我们要实现的效果已经挺像了，除了每个扇形的颜色，效果中阴影下面的扇形颜色更深，有种光线被遮住的感觉，从而会出现层次感和空间感。

ECharts 中每个扇形颜色的可以通过分别设置 data 下的数据项实现。

```
data: [{
    value:400,
    name:'搜索引擎',
    itemStyle: {
        color: '#c23531'
    }
}, ...]
```

但是这次因为只有明暗度的变化，所以有一种更快捷的方式是通过 visualMap 组件将数值的大小映射到明暗度。

```
visualMap: {
    // 不显示 visualMap 组件，只用于明暗度的映射
    show: false,
    // 映射的最小值为 80
    min: 80,
    // 映射的最大值为 600
    max: 600,
    inRange: {
        // 明暗度的范围是 0 到 1
        colorLightness: [0, 1]
    }
}
```

最终效果： https://echarts.baidu.com/gallery/view.html?c=doc-example/tutorial-styling-step5&edit=1&reset=1

使用 **Canvas** 或者 **SVG** 渲染

浏览器端图表库大多会选择 SVG 或者 Canvas 进行渲染。对于绘制图表来说，这两种技术往往是可替换的，效果相近。但是在一些场景中，他们的表现和能力又有一定差异。于是，对它们的选择取舍，就成为了一个一直存在的不易有标准答案的话题。

ECharts 从初始一直使用 Canvas 绘制图表（除了对 IE8- 使用 VML）。而 ECharts v4.0 发布了 SVG 渲染器，从而提供了一种新的选择。只须在初始化一个图表实例时，设置 renderer 参数为 `'canvas'` 或 `'svg'` 即可指定渲染器，比较方便。

> SVG 和 Canvas 这两种使用方式差异很大的技术，能够做到同时被透明支持，主要归功于 ECharts 底层库 ZRender 的抽象和实现，形成可互换的 SVG 渲染器和 Canvas 渲染器。

选择哪种渲染器

一般来说，Canvas 更适合绘制图形元素数量非常大（这一般是由数据量大导致）的图表（如热力图、地理坐标系或平行坐标系上的大规模线图或散点图等），也利于实现某些视觉 特效。但是，在不少场景中，SVG 具有重要的优势：它的内存占用更低（这对移动端尤其重要）、渲染性能略高、并且用户使用浏览器内置的缩放功能时不会模糊。例如，我们在一些硬件环境中分别使用 Canvas 渲染器和 SVG 渲染器绘制中等数据量的折、柱、饼，统计初始动画阶段的帧率，得到了一个性能对比图：https://echarts.baidu.com/gallery/view.html?c=doc-example/canvas-vs-svg&reset=1“ 上图显示出，在这些场景中，SVG 渲染器相比 Canvas 渲染器在移动端的总体表现更好。当然，这个实验并非是全面的评测，在另一些数据量较大或者有图表交互动画的场景中，目前的 SVG 渲染器的性能还比不过 Canvas 渲染器。但是同时有这两个选项，为开发者们根据自己的情况优化性能提供了更广阔的空间。

选择哪种渲染器，我们可以根据软硬件环境、数据量、功能需求综合考虑。

- 在软硬件环境较好，数据量不大的场景下（例如 PC 端做商务报表），两种渲染器都可以适用，并不需要太多纠结。
- 在环境较差，出现性能问题需要优化的场景下，可以通过试验来确定使用哪种渲染器。比如有这些经验：
    - 在须要创建很多 ECharts 实例且浏览器易崩溃的情况下（可能是因为 Canvas 数量多导致内存占用超出手机承受能力），可以使用 SVG 渲染器来进行改善。大略得说，如果图表运行在低端安卓机，或者我们在使用一些特定图表如 水球图 等，SVG 渲染器可能效果更好。
    - 数据量很大、较多交互时，可以选用 Canvas 渲染器。 我们强烈欢迎开发者们 反馈 给我们使用的体验和场景，帮助我们更好的做优化。

注：除了某些特殊的渲染可能依赖 Canvas：如炫光尾迹特效、带有混合效果的热力图等，绝大部分功能 SVG 都是支持的。此外，目前的 SVG 版中，富文本、材质功能尚未实现。

如何使用渲染器

ECharts 默认使用 Canvas 渲染。如果想使用 SVG 渲染，ECharts 代码中须包括有 SVG 渲染器模块。

- ECharts 的 预构建文件 中，常用版 和 完整版 已经包含了 SVG 渲染器，可直接使用。而精简版 没有包括。
- 如果 在线自定义构建 ECharts，则需要勾上页面下方的 “SVG 渲染”。
- 如果 线下自定义构建 ECharts，则须引入 SVG 渲染器模块，即：

```
1. import 'zrender/lib/svg/svg';
```

然后，我们就可以在代码中，初始化图表实例时，传入参数 选择渲染器类型：

```
1. // 使用 Canvas 渲染器（默认）
2. var chart = echarts.init(containerDom, null, {renderer: 'canvas'});
3. // 等价于：
4. var chart = echarts.init(containerDom);
5.
6. // 使用 SVG 渲染器
7. var chart = echarts.init(containerDom, null, {renderer: 'svg'});
```

使用 **ECharts GL** 实现基础的三维可视化

ECharts GL （后面统一简称 GL）为 ECharts 补充了丰富的三维可视化组件，这篇文章我们会简单介绍如何基于 GL 实现一些常见的三维可视化作品。实际上如果你对 ECharts 有一定了解的话，也可以很快的上手 GL，GL 的配置项完全是按照 ECharts 的标准和上手难度来设计的。

在看完文章之后，你可以前往 官方示例 和 Gallery 去了解更多使用 GL 制作的示例，对于文章中我们没法解释到的代码，也可以前往 GL 配置项手册 查看具体的配置项使用方法。

如何下载和引入 ECharts GL

为了不再增加已经很大了的 ECharts 完整版的体积，我们将 GL 作为扩展包的形式提供，和诸如水球图这样的扩展类似，如果要使用 GL 里的各种组件，只需要在引入 `echarts.min.js` 的基础上再引入一个 `echarts-gl.min.js` 。你可以从 官网 下载最新版的 GL，然后在页面中通过标签引入：

```
1. <script src="lib/echarts.min.js"></script>
2. <script src="lib/echarts-gl.min.js"></script>
```

如果你的项目使用 webpack 或者 rollup 来打包代码的话，也可以通过 npm 安装后引入

```
1. npm install echarts
2. npm install echarts-gl
```

```
1. // 通过 ES6 的 import 语法引入 ECharts 和 ECharts GL
2. import echarts from 'echarts';
3. import 'echarts-gl';
```

声明一个基础的三维笛卡尔坐标系

引入 ECharts 和 ECharts GL 后，我们先来声明一个基础的三维笛卡尔坐标系用于绘制三维的散点图，柱状图，曲面图等常见的统计图。

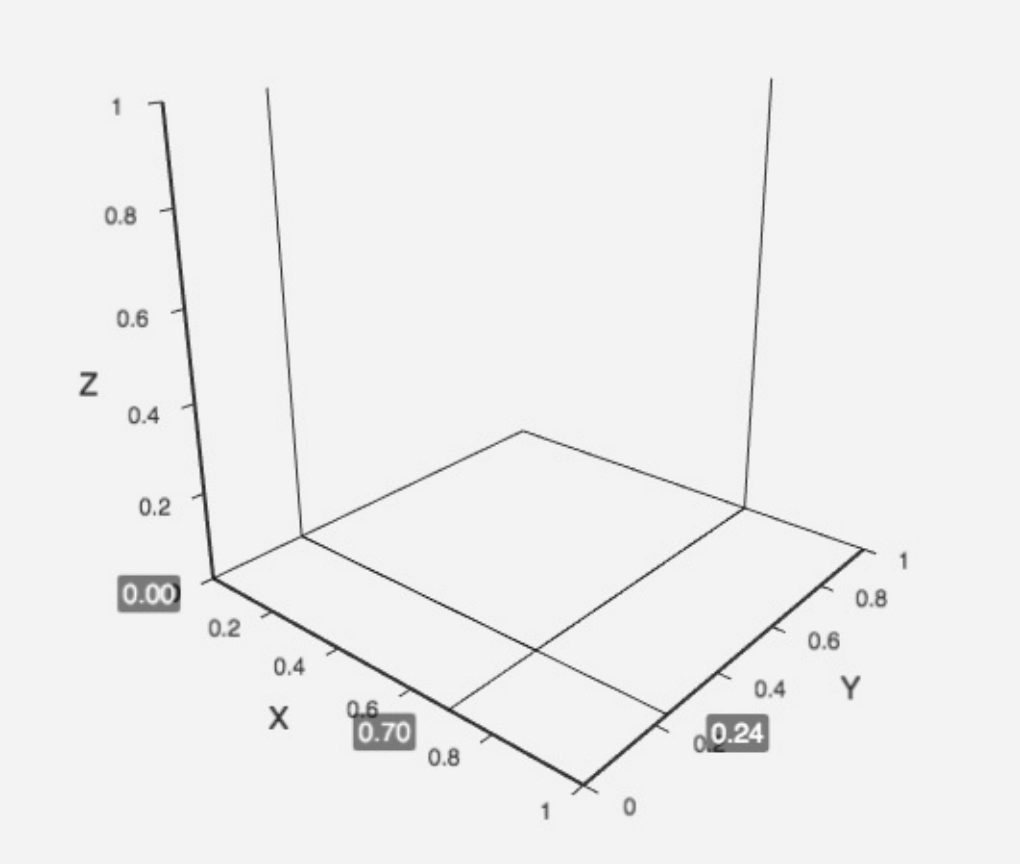
在 ECharts 中我们有 grid 组件用于提供一个矩形的区域放置一个二维的笛卡尔坐标系，以及笛卡尔坐标系上上的 x 轴（xAxis）和 y 轴（yAxis）。对于三维的笛卡尔坐标系，我们在 GL 中提供了 grid3D 组件用于划分一块三维的笛卡尔空间，以及放置在这个 grid3D 上的 xAxis3D, yAxis3D, zAxis3D。

> 小提示：在 GL 中我们对除了 globe 之外所有的三维组件和系列都加了 3D 的后缀用以区分，例如三维的散点图就是 scatter3D，三维的地图就是 map3D 等等。

下面这段代码就声明了一个最简单的三维笛卡尔坐标系

```
var option = {
    // 需要注意的是我们不能跟 grid 一样省略 grid3D
    grid3D: {},
    // 默认情况下, x, y, z 分别是从 0 到 1 的数值轴
    xAxis3D: {},
    yAxis3D: {},
    zAxis3D: {}
}
```

效果如下：



跟二维的笛卡尔坐标系一样，每个轴都会有多种类型，默认是数值轴，如果需要是类目轴的话，简单的设置为 `type: 'category'` 就行了。

绘制三维的散点图

声明好笛卡尔坐标系后，我们先试试用一份程序生成的正态分布数据在这个三维的笛卡尔坐标系中画散点图。

下面这段是生成正态分布数据的代码，你可以先不用关心这段代码是怎么工作的，只需要知道它生成了一份三维的正态分布数据放在 `data` 数组中。

```
function makeGaussian(amplitude, x0, y0, sigmaX, sigmaY) {
    return function (amplitude, x0, y0, sigmaX, sigmaY, x, y) {
        var exponent = -(
                ( Math.pow(x - x0, 2) / (2 * Math.pow(sigmaX, 2)))
                + ( Math.pow(y - y0, 2) / (2 * Math.pow(sigmaY, 2)))
            );
        return amplitude * Math.pow(Math.E, exponent);
    }.bind(null, amplitude, x0, y0, sigmaX, sigmaY);
}
// 创建一个高斯分布函数
var gaussian = makeGaussian(50, 0, 0, 20, 20);

var data = [];
for (var i = 0; i < 1000; i++) {
    // x, y 随机分布
    var x = Math.random() * 100 - 50;
    var y = Math.random() * 100 - 50;
    var z = gaussian(x, y);
    data.push([x, y, z]);
}
```

生成的正态分布的数据大概长这样：

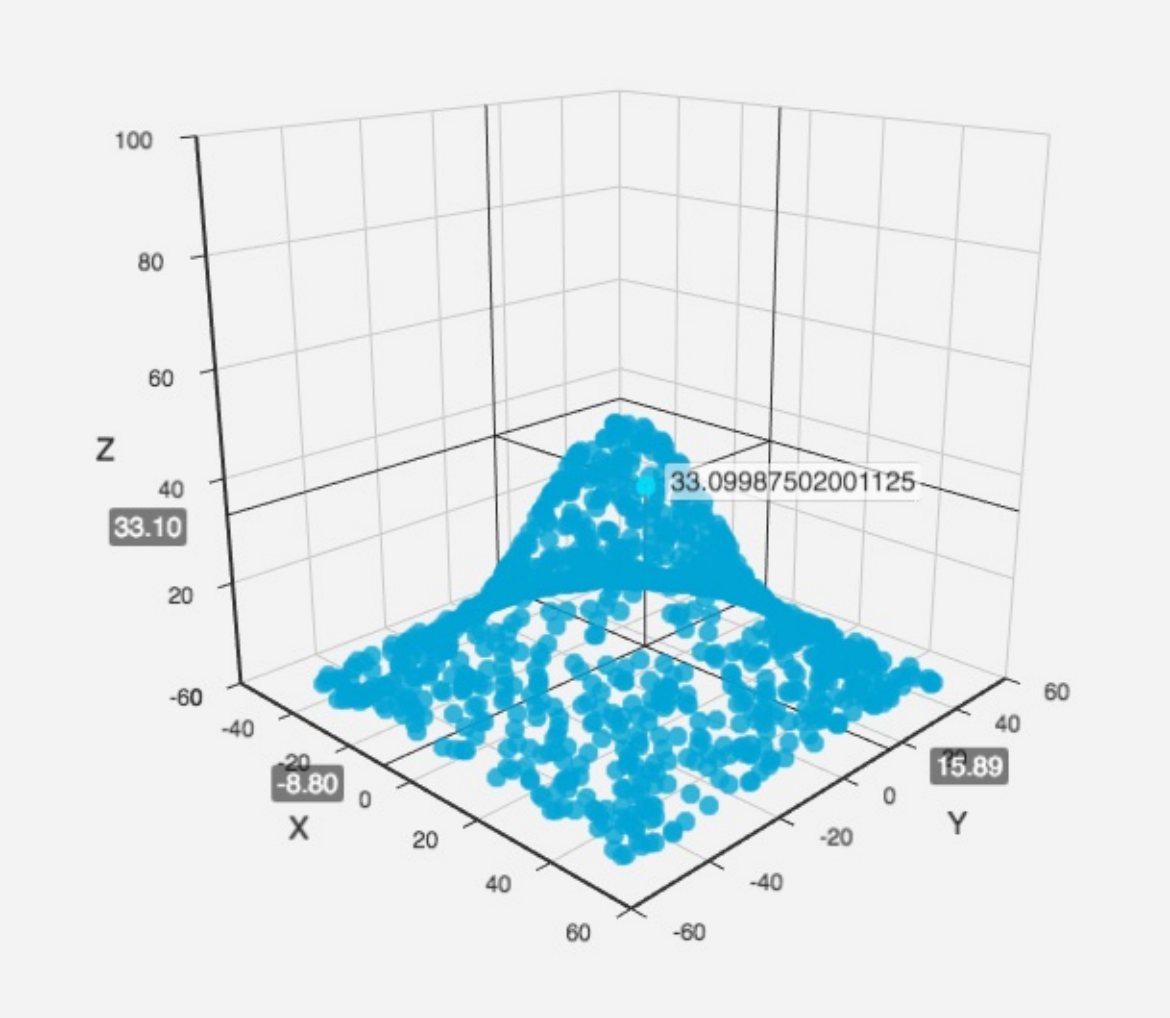
```
[
  [46.74395071259907, -33.88391024738553, 0.7754030099768191],
  [-18.45302873809771, 16.88114775416834, 22.87772504105404],
  [2.9908128281121336, -0.027699444453467947, 49.44400635911886],
  ...
]
```

每一项都包含了 `x` ， `y` ， `z` 三个值，这三个值会分别被映射到笛卡尔坐标系的 x 轴，y 轴和 z 轴上。

然后我们可以使用 GL 提供的 `scatter3D` 系列类型把这些数据画成三维空间中正态分布的点。

```
option = {
    grid3D: {},
    xAxis3D: {},
    yAxis3D: {},
    zAxis3D: { max: 100 },
    series: [{
        type: 'scatter3D',
```

```
        data: data
    }]
}
```



使用真实数据的三维散点图

接下来我们来看一个使用真实多维数据的三维散点图例子。

可以先从 `http://www.echartsjs.com/examples/data/asset/data/life-expectancy-table.json` 获取这份数据。

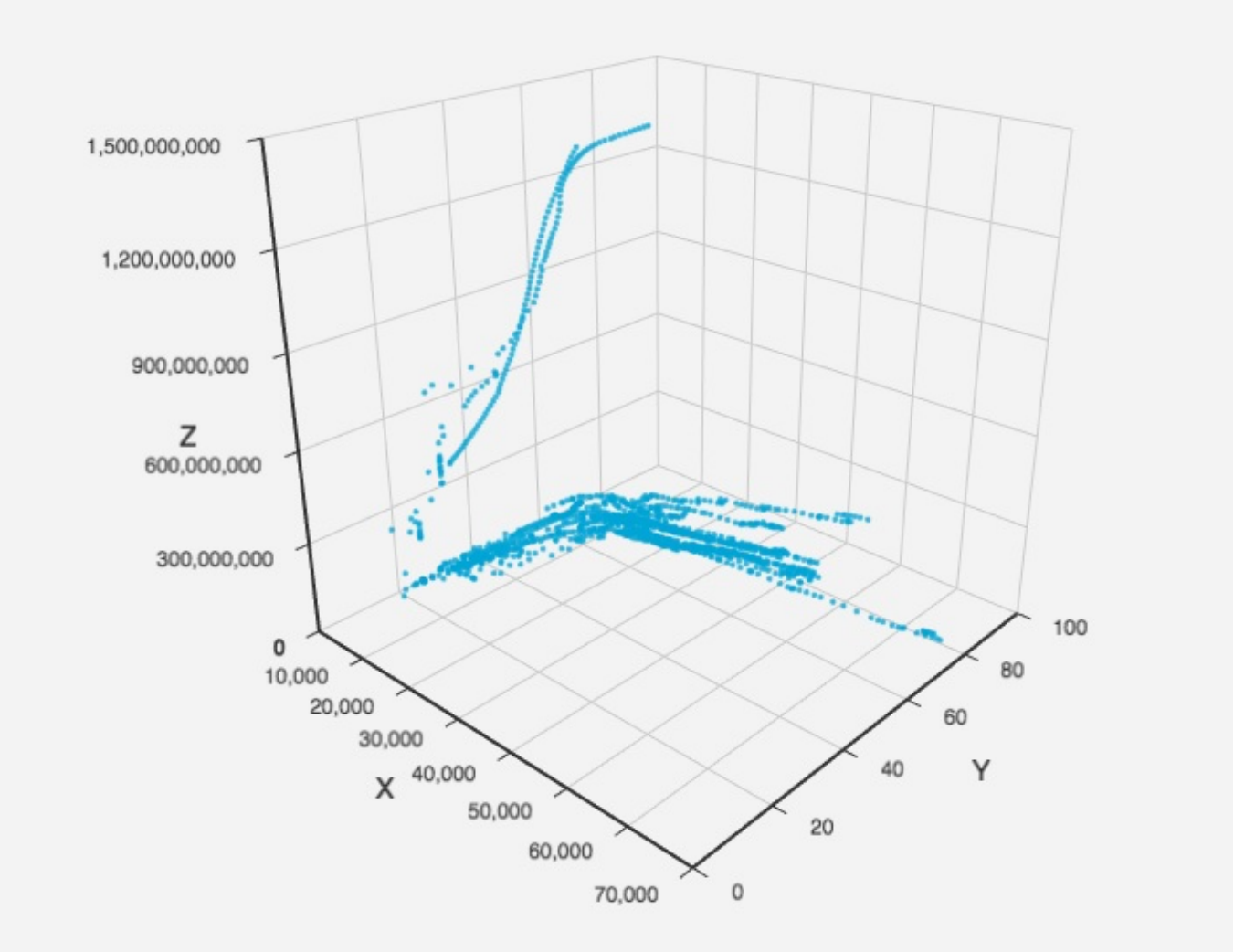
格式化一下可以看到这份数据是很传统转成 JSON 后的表格格式。第一行是每一列数据的属性名，可以从这个属性名看出来每一列数据的含义，分别是人均收入，人均寿命，人口数量，国家和年份。

```
[
["Income", "Life Expectancy", "Population", "Country", "Year"],
[815, 34.05, 351014, "Australia", 1800],
[1314, 39, 645526, "Canada", 1800],
[985, 32, 321675013, "China", 1800],
```

```
[864, 32.2, 345043, "Cuba", 1800],
[1244, 36.5731262, 977662, "Finland", 1800],
...
]
```

在 ECharts 4 中我们可以使用 dataset 组件非常方便地引入这份数据。如果对 dataset 还不熟悉的话可以看dataset使用教程

```
$.get('data/asset/data/life-expectancy-table.json', function (data) {
    myChart.setOption({
        grid3D: {},
        xAxis3D: {},
        yAxis3D: {},
        zAxis3D: {},
        dataset: {
            source: data
        },
        series: [
            {
                type: 'scatter3D',
                symbolSize: 2.5
            }
        ]
    })
});
```



默认会把前三列，也就是收入（Income），人均寿命（Life Expectancy），人口（Population）分别放到 x、 y、 z 轴上。

使用 encode 属性我们还可以将指定列的数据映射到指定的坐标轴上，从而省去很多繁琐的数据转换代码。例如我们将 x 轴换成是国家（Country），y 轴换成年份（Year），z 轴换成收入（Income），可以看到不同国家不同年份的人均收入分布。

```
myChart.setOption({
    grid3D: {},
    xAxis3D: {
        // 因为 x 轴和 y 轴都是类目数据，所以需要设置 type: 'category' 保证正确显示数据。
        type: 'category'
    },
    yAxis3D: {
        type: 'category'
    },
    zAxis3D: {},
    dataset: {
```

```
        source: data
    },
    series: [
        {
            type: 'scatter3D',
            symbolSize: 2.5,
            encode: {
                // 维度的名字默认就是表头的属性名
                x: 'Country',
                y: 'Year',
                z: 'Income',
                tooltip: [0, 1, 2, 3, 4]
            }
        }
    ]
});
```

利用 visualMap 组件对三维散点图进行视觉编码

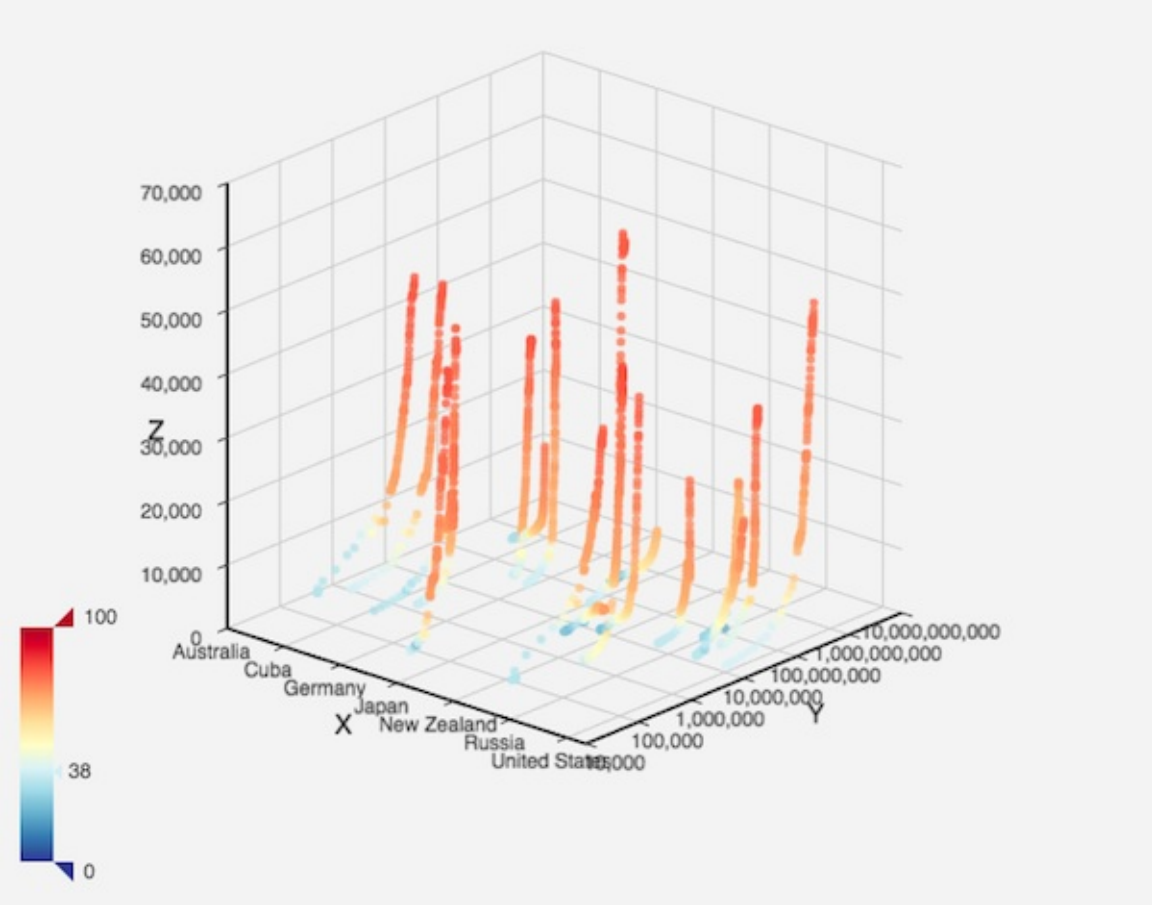
刚才多维数据的例子中，我们还有几个维度（列）没能表达出来，利用 ECharts 内置的 visualMap 组件我们可以继续将第四个维度编码成颜色。

```
myChart.setOption({
    grid3D: {
        viewControl: {
            // 使用正交投影。
            projection: 'orthographic'
        }
    },
    xAxis3D: {
        // 因为 x 轴和 y 轴都是类目数据，所以需要设置 type: 'category' 保证正确显示数据。
        type: 'category'
    },
    yAxis3D: {
        type: 'log'
    },
    zAxis3D: {},
    visualMap: {
        calculable: true,
        max: 100,
```

```
        // 维度的名字默认就是表头的属性名
        dimension: 'Life Expectancy',
        inRange: {
            color: ['#313695', '#4575b4', '#74add1', '#abd9e9', '#e0f3f8', '#ffffbf', '#fee090', '#fdae61', '#f46d43', '#d73027', '#a50026']
        }
    },
    dataset: {
        source: data
    },
    series: [
        {
            type: 'scatter3D',
            symbolSize: 5,
            encode: {
                // 维度的名字默认就是表头的属性名
                x: 'Country',
                y: 'Population',
                z: 'Income',
                tooltip: [0, 1, 2, 3, 4]
            }
        }
    ]
})
```

这段代码中我们又在刚才的例子基础上加入了 `visualMap` 组件，将 `Life Expectancy` 这一列数据映射到了不同的颜色。

除此之外我们还把原来默认的透视投影改成了正交投影。正交投影在某些场景中可以避免因为近大远小所造成的表达错误。

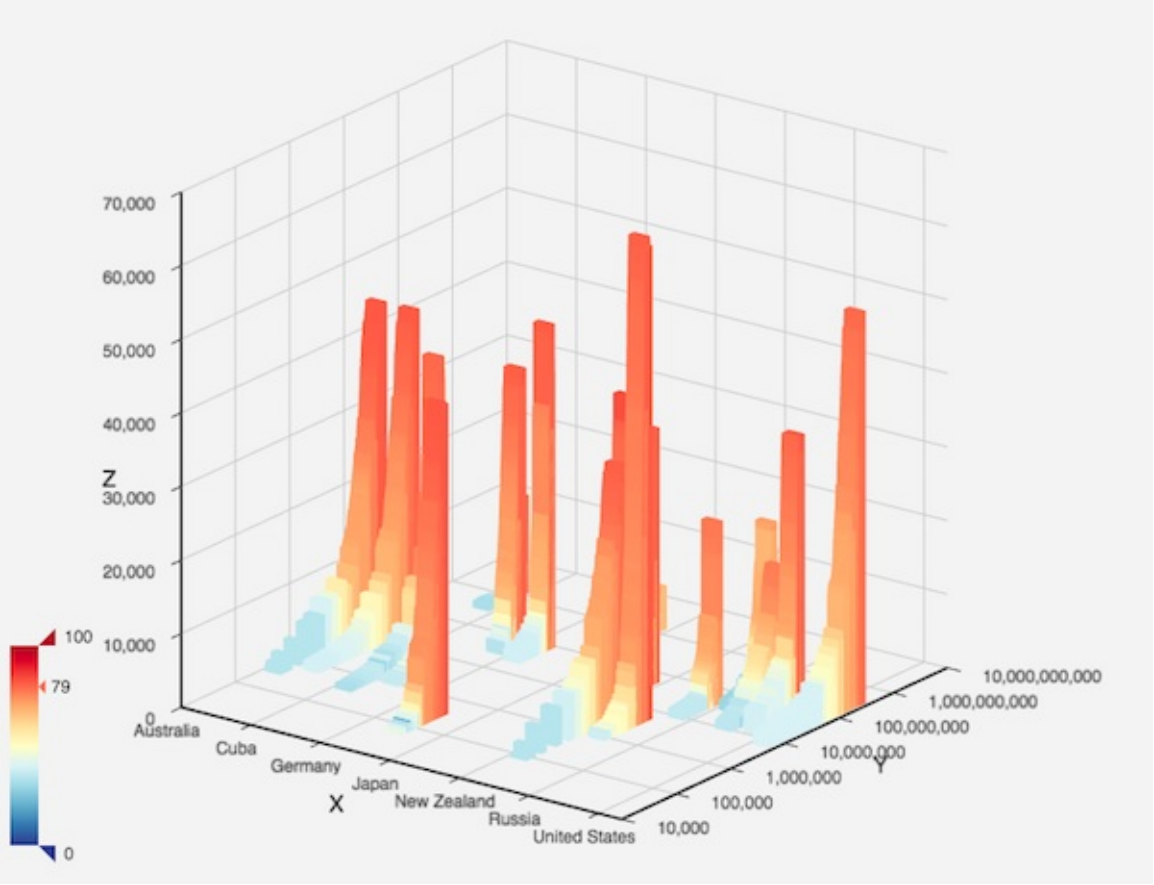


当然，除了 `visualMap` 组件，还可以利用其它的 ECharts 内置组件并且充分利用这些组件的交互效果，比如 `legend`。也可以像 三维散点图和散点矩阵结合使用 这个例子一样实现二维和三维的系列混搭。

在实现 GL 的时候我们尽可能地把 WebGL 和 Canvas 之间的差异屏蔽了到最小，从而让 GL 的使用可以更加方便自然。

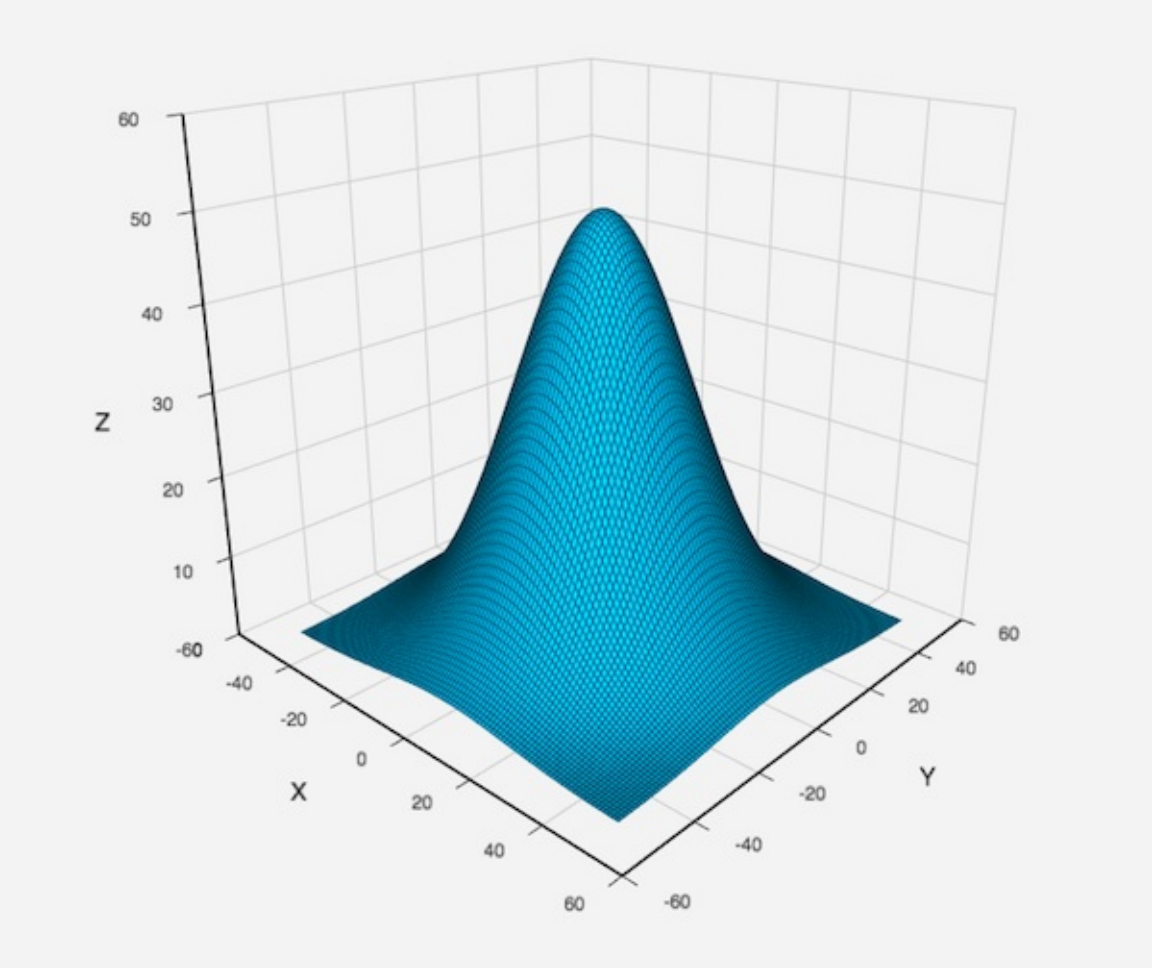
在笛卡尔坐标系上显示其它类型的三维图表

除了散点图，我们也可以通过 GL 在三维的笛卡尔坐标系上绘制其它类型的三维图表。比如刚才例子中将 `scatter3D` 类型改成 `bar3D` 就可以变成一个三维的柱状图。



还有机器学习中会用到的三维曲面图 `surface`，三维曲面图常用来表达平面上的数据走势，刚才的正态分布数据我们也可以像下面这样画成曲面图。

```
var data = [];
// 曲面图要求给入的数据是网格形式按顺序分布。
for (var y = -50; y <= 50; y++) {
    for (var x = -50; x <= 50; x++) {
        var z = gaussian(x, y);
        data.push([x, y, z]);
    }
}
option = {
    grid3D: {},
    xAxis3D: {},
    yAxis3D: {},
    zAxis3D: { max: 60 },
    series: [{
        type: 'surface',
        data: data
    }]
}
```



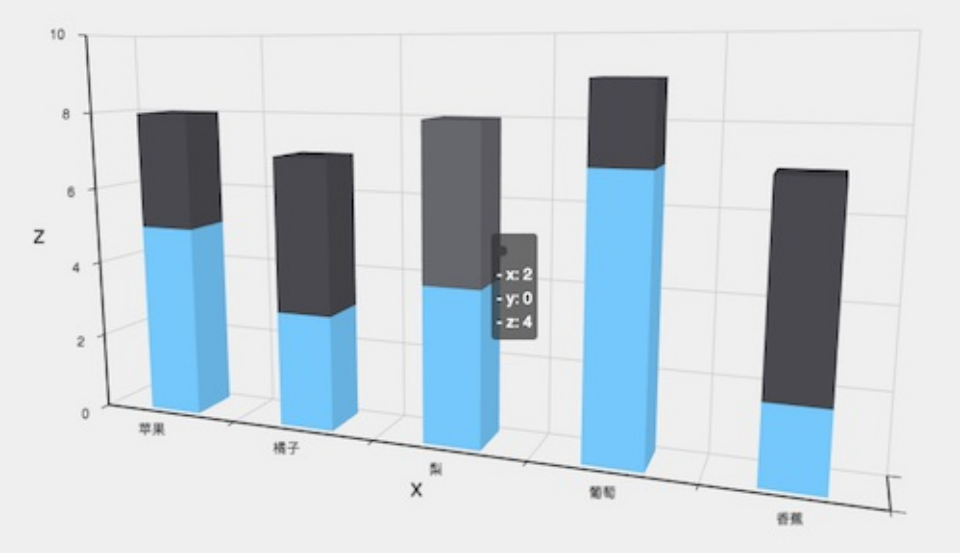
老板想要立体的柱状图效果

最后，我们经常会被问到如何用 ECharts 画只有二维数据的立体柱状图效果。一般来说我们是不推荐这么做的，因为这种不必要的立体柱状图很容易造成错误的表达，具体可以见我们 柱状图使用指南 中的解释。

但是如果有一些其他因素导致必须得画成立体的柱状图的话，用 GL 也可以实现。 ヽ灬豆奶 和 阿洛儿啊 在 Gallery 已经写了类似的例子，大家可以参考。

3D堆积柱状图

3D柱状图


10
8
6
4
2
0
Z
苹果
橘子
梨
X
葡萄
香蕉
- x: 2
- y: 0
- z: 4

使用 dataset 管理数据

ECharts 4 开始支持了 `dataset` 组件用于单独的数据集声明，从而数据可以单独管理，被多个组件复用，并且可以基于数据指定数据到视觉的映射。这在不少场景下能带来使用上的方便。

ECharts 4 以前，数据只能声明在各个“系列（series）”中，例如：

```
option: {
    xAxis: {
        type: 'category',
        data: ['Matcha Latte', 'Milk Tea', 'Cheese Cocoa', 'Walnut Brownie']
    },
    yAxis: {}
    series: [
        {
            type: 'bar',
            name: '2015',
            data: [89.3, 92.1, 94.4, 85.4]
        },
        {
            type: 'bar',
            name: '2016',
            data: [95.8, 89.4, 91.2, 76.9]
        },
        {
            type: 'bar',
            name: '2017',
            data: [97.7, 83.1, 92.5, 78.1]
        }
    ]
}
```

这种方式的优点是，直观易理解，以及适于对一些特殊图表类型进行一定的数据类型定制。但是缺点是，为匹配这种数据输入形式，常需要有数据处理的过程，把数据分割设置到各个系列（和类目轴）中。此外，不利于多个系列共享一份数据，也不利于基于原始数据进行图表类型、系列的映射安排。

于是，ECharts 4 提供了 `数据集` （ `dataset` ）组件来单独声明数据，它带来了这些效果：

- 能够贴近这样的数据可视化常见思维方式：基于数据（ `dataset` 组件来提供数据），指定数据到视觉的映射（由 `encode` 属性来指定映射），形成图表。
- 数据和其他配置可以被分离开来，使用者相对便于进行单独管理，也省去了一些数据处理的步骤。
- 数据可以被多个系列或者组件复用，对于大数据，不必为每个系列创建一份。

- 支持更多的数据的常用格式，例如二维数组、对象数组等，一定程度上避免使用者为了数据格式而进行转换。

入门例子

下面是一个最简单的 `dataset` 的例子：

```
option = {
    legend: {},
    tooltip: {},
    dataset: {
        // 提供一份数据。
        source: [
            ['product', '2015', '2016', '2017'],
            ['Matcha Latte', 43.3, 85.8, 93.7],
            ['Milk Tea', 83.1, 73.4, 55.1],
            ['Cheese Cocoa', 86.4, 65.2, 82.5],
            ['Walnut Brownie', 72.4, 53.9, 39.1]
        ]
    },
    // 声明一个 X 轴，类目轴（category）。默认情况下，类目轴对应到 dataset 第一列。
    xAxis: {type: 'category'},
    // 声明一个 Y 轴，数值轴。
    yAxis: {},
    // 声明多个 bar 系列，默认情况下，每个系列会自动对应到 dataset 的每一列。
    series: [
        {type: 'bar'},
        {type: 'bar'},
        {type: 'bar'}
    ]
}
```

效果如下： https://echarts.baidu.com/gallery/view.html?c=dataset-simple0&edit=1&reset=1“

或者也可以使用常见的对象数组的格式：

```
option = {
    legend: {},
    tooltip: {},
    dataset: {
        // 这里指定了维度名的顺序，从而可以利用默认的维度到坐标轴的映射。
```

```
        // 如果不指定 dimensions，也可以通过指定 series.encode 完成映射，参见后文。
        dimensions: ['product', '2015', '2016', '2017'],
        source: [
            {product: 'Matcha Latte', '2015': 43.3, '2016': 85.8, '2017': 93.7},
            {product: 'Milk Tea', '2015': 83.1, '2016': 73.4, '2017': 55.1},
            {product: 'Cheese Cocoa', '2015': 86.4, '2016': 65.2, '2017': 82.5},
            {product: 'Walnut Brownie', '2015': 72.4, '2016': 53.9, '2017': 39.1}
        ]
    },
    xAxis: {type: 'category'},
    yAxis: {},
    series: [
        {type: 'bar'},
        {type: 'bar'},
        {type: 'bar'}
    ]
};
```

数据到图形的映射

本篇里，我们制作数据可视化图表的逻辑是这样的：基于数据，在配置项中指定如何映射到图形。

概略而言，可以进行这些映射：

- 指定 dataset 的列（column）还是行（row）映射为图形系列（series）。这件事可以使用 series.seriesLayoutBy 属性来配置。默认是按照列（column）来映射。
- 指定维度映射的规则：如何从 dataset 的维度（一个“维度”的意思是一行/列）映射到坐标轴（如 X、Y 轴）、提示框（tooltip）、标签（label）、图形元素大小颜色等（visualMap）。这件事可以使用 series.encode 属性，以及 visualMap 组件（如果有需要映射颜色大小等视觉维度的话）来配置。上面的例子中，没有给出这种映射配置，那么 ECharts 就按最常见的理解进行默认映射：X 坐标轴声明为类目轴，默认情况下会自动对应到 dataset.source 中的第一列；三个柱图系列，一一对应到 dataset.source 中后面每一列。 下面详细解释。

按行还是按列做映射

有了数据表之后，使用者可以灵活得配置：数据如何对应到轴和图形系列。

用户可以使用 `seriesLayoutBy` 配置项，改变图表对于行列的理解。 `seriesLayoutBy` 可取

值：

- 'column'：默认值。系列被安放到 `dataset` 的列上面。
- 'row'：系列被安放到 `dataset` 的行上面。 看这个例子：

```
option = {
    legend: {},
    tooltip: {},
    dataset: {
        source: [
            ['product', '2012', '2013', '2014', '2015'],
            ['Matcha Latte', 41.1, 30.4, 65.1, 53.3],
            ['Milk Tea', 86.5, 92.1, 85.7, 83.1],
            ['Cheese Cocoa', 24.1, 67.2, 79.5, 86.4]
        ]
    },
    xAxis: [
        {type: 'category', gridIndex: 0},
        {type: 'category', gridIndex: 1}
    ],
    yAxis: [
        {gridIndex: 0},
        {gridIndex: 1}
    ],
    grid: [
        {bottom: '55%'},
        {top: '55%'}
    ],
    series: [
        // 这几个系列会在第一个直角坐标系中，每个系列对应到 dataset 的每一行。
        {type: 'bar', seriesLayoutBy: 'row'},
        {type: 'bar', seriesLayoutBy: 'row'},
        {type: 'bar', seriesLayoutBy: 'row'},
        // 这几个系列会在第二个直角坐标系中，每个系列对应到 dataset 的每一列。
        {type: 'bar', xAxisIndex: 1, yAxisIndex: 1},
        {type: 'bar', xAxisIndex: 1, yAxisIndex: 1},
        {type: 'bar', xAxisIndex: 1, yAxisIndex: 1},
        {type: 'bar', xAxisIndex: 1, yAxisIndex: 1}
    ]
}
```

效果如下： https://echarts.baidu.com/gallery/view.html?c=dataset-series-

layout-by&edit=1&reset=1

维度（**dimension**）

介绍 `encode` 之前，首先要介绍“维度（dimension）”的概念。

常用图表所描述的数据大部分是“二维表”结构，上述的例子中，我们都使用二维数组来容纳二维表。现在，当我们把系列（series）对应到“列”的时候，那么每一列就称为一个“维度（dimension）”，而每一行称为数据项（item）。反之，如果我们把系列（series）对应到表行，那么每一行就是“维度（dimension）”，每一列就是数据项（item）。

维度可以有单独的名字，便于在图表中显示。维度名（dimension name）可以在定义在 dataset 的第一行（或者第一列）。例如上面的例子中，`'score'` 、 `'amount'` 、 `'product'` 就是维度名。从第二行开始，才是正式的数据。 `dataset.source` 中第一行（列）到底包含不包含维度名，ECharts 默认会自动探测。当然也可以设置 `dataset.sourceHeader: true` 显示声明第一行（列）就是维度，或者 `dataset.sourceHeader: false` 表明第一行（列）开始就直接是数据。

维度的定义，也可以使用单独的 `dataset.dimensions` 或者 `series.dimensions` 来定义，这样可以同时指定维度名，和维度的类型（dimension type）：

```
var option1 = {
    dataset: {
        dimensions: [
            {name: 'score'},
            // 可以简写为 string，表示维度名。
            'amount',
            // 可以在 type 中指定维度类型。
            {name: 'product', type: 'ordinal'}
        ],
        source: [...]
    },
    ...
};

var option2 = {
    dataset: {
        source: [...]
    },
    series: {
        type: 'line',
        // 在系列中设置的 dimensions 会更优先采纳。
        dimensions: [
```

```
            null, // 可以设置为 null 表示不想设置维度名
            'amount',
            {name: 'product', type: 'ordinal'}
        ]
    },
    ...
};
```

大多数情况下，我们并不需要去设置维度类型，因为会自动判断。但是如果因为数据为空之类原因导致判断不足够准确时，可以手动设置维度类型。

维度类型（dimension type）可以取这些值：

- `'number'`：默认，表示普通数据。
- `'ordinal'`：对于类目、文本这些 string 类型的数据，如果需要能在数轴上使用，须是 `'ordinal'` 类型。ECharts 默认会自动判断这个类型。但是自动判断也是不可能很完备的，所以使用者也可以手动强制指定。
- `'time'`：表示时间数据。设置成 `'time'` 则能支持自动解析数据成时间戳（timestamp），比如该维度的数据是 `'2017-05-10'`，会自动被解析。如果这个维度被用在时间数轴（`axis.type` 为 `'time'`）上，那么会被自动设置为 `'time'` 类型。时间类型的支持参见 `data`。
- `'float'`：如果设置成 `'float'`，在存储时候会使用 `TypedArray`，对性能优化有好处。
- `'int'`：如果设置成 `'int'`，在存储时候会使用 `TypedArray`，对性能优化有好处。

数据到图形的映射（**encode**）

了解了维度的概念后，我们就可以使用 `encode` 来做映射。总体是这样的感觉：

```
var option = {
    dataset: {
        source: [
            ['score', 'amount', 'product'],
            [89.3, 58212, 'Matcha Latte'],
            [57.1, 78254, 'Milk Tea'],
            [74.4, 41032, 'Cheese Cocoa'],
            [50.1, 12755, 'Cheese Brownie'],
            [89.7, 20145, 'Matcha Cocoa'],
            [68.1, 79146, 'Tea'],
            [19.6, 91852, 'Orange Juice'],
            [10.6, 101852, 'Lemon Juice'],
            [32.7, 20112, 'Walnut Brownie']
        ]
```

```
    },
    xAxis: {},
    yAxis: {type: 'category'},
    series: [
        {
            type: 'bar',
            encode: {
                // 将 "amount" 列映射到 X 轴。
                x: 'amount',
                // 将 "product" 列映射到 Y 轴。
                y: 'product'
            }
        }
    ]
};
```

效果如下： https://echarts.baidu.com/gallery/view.html?c=doc-example/dataset-encode-simple0&edit=1&reset=1

`encode` 声明的基本结构如下，其中冒号左边是坐标系、标签等特定名称，如 `'x'` ， `'y'` ， `'tooltip'` 等，冒号右边是数据中的维度名（string 格式）或者维度的序号（number 格式，从 0 开始计数），可以指定一个或多个维度（使用数组）。通常情况下，下面各种信息不需要所有的都写，按需写即可。

下面是 encode 支持的属性：

```
// 在任何坐标系和系列中，都支持：
encode: {
    // 使用 “名为 product 的维度” 和 “名为 score 的维度” 的值在 tooltip 中显示
    tooltip: ['product', 'score']
    // 使用 “维度 1” 和 “维度 3” 的维度名连起来作为系列名。（有时候名字比较长，这可以避免在 series.name 重复输入这些名字）
    seriesName: [1, 3],
    // 表示使用 “维度2” 中的值作为 id。这在使用 setOption 动态更新数据时有用处，可以使新老数据用 id 对应起来，从而能够产生合适的数据更新动画。
    itemId: 2,
    // 指定数据项的名称使用 “维度3” 在饼图等图表中有用，可以使这个名字显示在图例（legend）中。
    itemName: 3
}

// 直角坐标系（grid/cartesian）特有的属性：
```

```
encode: {
    // 把 “维度1”、“维度5”、“名为 score 的维度” 映射到 X 轴：
    x: [1, 5, 'score'],
    // 把“维度0”映射到 Y 轴。
    y: 0
}

// 极坐标系（polar）特有的属性：
encode: {
    radius: 3,
    angle: 2
}

// 地理坐标系（geo）特有的属性：
encode: {
    lng: 3,
    lat: 2
}

// 对于一些没有坐标系的图表，例如饼图、漏斗图等，可以是：
encode: {
    value: 3
}
```

下面给出个更丰富的 `encode` 的示例：
https://echarts.baidu.com/gallery/view.html?c=dataset-encode1&edit=1&reset=1

视觉通道（颜色、尺寸等）的映射

我们可以使用 visualMap 组件进行视觉通道的映射。详见 `visualMap` 文档的介绍。这是一个示例： https://echarts.baidu.com/gallery/view.html?c=dataset-encode0&edit=1&reset=1

默认的映射

指的一提的是，ECharts 针对最常见直角坐标系中的图表（折线图、柱状图、散点图、K线图等）、饼图、漏斗图，给出了简单的默认的映射，从而不需要配置 `encode` 也可以出现图表（一旦给出了 `encode`，那么就不会采用默认映射）。默认的映射规则不易做得复杂，基本规则大体是：

- 在坐标系中（如直角坐标系、极坐标系等）
    - 如果有类目轴（axis.type 为 'category'），则将第一列（行）映射到这个轴上，后

续每一列（行）对应一个系列。
    - 如果没有类目轴，假如坐标系有两个轴（例如直角坐标系的 X Y 轴），则每两列对应一个系列，这两列分别映射到这两个轴上。
- 如果没有坐标系（如饼图）
    - 取第一列（行）为名字，第二列（行）为数值（如果只有一列，则取第一列为数值）。 默认的规则不能满足要求时，就可以自己来配置 `encode`，也并不复杂。 https://echarts.baidu.com/gallery/view.html?c=dataset-default&edit=1&reset=1

几个常见的映射设置方式

问：如何把第三列设置为 X 轴，第五列设置为 Y 轴？

答：

```
series: {
    // 注意维度序号（dimensionIndex）从 0 开始计数，第三列是 dimensions[2]。
    encode: {x: 2, y: 4},
    ...
}
```

问：如何把第三行设置为 X 轴，第五行设置为 Y 轴？

答：

```
series: {
    encode: {x: 2, y: 4},
    seriesLayoutBy: 'row',
    ...
}
```

问：如何把第二列设置为标签？

答：关于标签的显示 label.formatter，现在支持引用特定维度的值，例如：

```
series: {
    label: {
        // `'{@score}'` 表示 “名为 score” 的维度里的值。
        // `'{@[4]}'` 表示引用序号为 4 的维度里的值。
        formatter: 'aaa{@product}bbb{@score}ccc{@[4]}ddd'
    }
}
```

问：如何让第 2 列和第 3 列显示在提示框（tooltip）中？

答：

```
series: {
    encode: {
        tooltip: [1, 2]
        ...
    },
    ...
}
```

问：数据里没有维度名，那么怎么给出维度名？

答：

```
dataset: {
    dimensions: ['score', 'amount'],
    source: [
        [89.3, 3371],
        [92.1, 8123],
        [94.4, 1954],
        [85.4, 829]
    ]
}
```

问：如何把第四列映射为气泡图的点的大小？

答：

```
var option = {
    dataset: {
        source: [
            [12, 323, 11.2],
            [23, 167, 8.3],
            [81, 284, 12],
            [91, 413, 4.1],
            [13, 287, 13.5]
        ]
    },
    visualMap: {
        show: false,
```

```
            dimension: 2, // 指向第三列（列序号从 0 开始记，所以设置为 2）。
            min: 2, // 需要给出数值范围，最小数值。
            max: 15, // 需要给出数值范围，最大数值。
            inRange: {
                // 气泡尺寸：5 像素到 60 像素。
                symbolSize: [5, 60]
            }
        },
        xAxis: {},
        yAxis: {},
        series: {
            type: 'scatter'
        }
    };
```

问：encode 里指定了映射，但是不管用？

答：可以查查有没有拼错，比如，维度名是：`'Life Expectancy'`，encode 中拼成了 `'Life Expecency'`。

数据的各种格式

多数常见图表中，数据适于用二维表的形式描述。广为使用的数据表格软件（如 MS Excel、Numbers）或者关系数据数据库都是二维表。他们的数据可以导出成 JSON 格式，输入到 `dataset.source` 中，在不少情况下可以免去一些数据处理的步骤。

> 假如数据导出成 csv 文件，那么可以使用一些 csv 工具如 dsv 或者 PapaParse 将 csv 转成 JSON。

在 JavaScript 常用的数据传输格式中，二维数组可以比较直观的存储二维表。前面的示例都是使用二维数组表示。

除了二维数组以外，dataset 也支持例如下面 key-value 方式的数据格式，这类格式也非常常见。但是这类格式中，目前并不支持 seriesLayoutBy 参数。

```
dataset: [{
    // 按行的 key-value 形式（对象数组），这是个比较常见的格式。
    source: [
        {product: 'Matcha Latte', count: 823, score: 95.8},
        {product: 'Milk Tea', count: 235, score: 81.4},
        {product: 'Cheese Cocoa', count: 1042, score: 91.2},
        {product: 'Walnut Brownie', count: 988, score: 76.9}
    ]
}, {
```

```
    // 按列的 key-value 形式。
    source: {
        'product': ['Matcha Latte', 'Milk Tea', 'Cheese Cocoa', 'Walnut Brownie'],
        'count': [823, 235, 1042, 988],
        'score': [95.8, 81.4, 91.2, 76.9]
    }
}]
```

多个 dataset 和他们的引用

可以同时定义多个 dataset。系列可以通过 series.datasetIndex 来指定引用哪个 dataset。例如：

```
var option = {
    dataset: [{
        // 序号为 0 的 dataset。
        source: [...],
    }, {
        // 序号为 1 的 dataset。
        source: [...]
    }, {
        // 序号为 2 的 dataset。
        source: [...]
    }],
    series: [{
        // 使用序号为 2 的 dataset。
        datasetIndex: 2
    }, {
        // 使用序号为 1 的 dataset。
        datasetIndex: 1
    }]
}
```

ECharts 3 的数据设置方式（series.data）仍正常使用

ECharts 4 之前一直以来的数据声明方式仍然被正常支持，如果系列已经声明了 series.data，那么就会使用 series.data 而非 `dataset` 。

```
{
    xAxis: {
```

```
        type: 'category'
        data: ['Matcha Latte', 'Milk Tea', 'Cheese Cocoa', 'Walnut Brownie']
    },
    yAxis: {},
    series: [{
        type: 'bar',
        name: '2015',
        data: [89.3, 92.1, 94.4, 85.4]
    }, {
        type: 'bar',
        name: '2016',
        data: [95.8, 89.4, 91.2, 76.9]
    }, {
        type: 'bar',
        name: '2017',
        data: [97.7, 83.1, 92.5, 78.1]
    }]
}
```

其实，series.data 也是种会一直存在的重要设置方式。一些特殊的非 table 格式的图表，如 treemap、graph、lines 等，现在仍不支持在 dataset 中设置，仍然需要使用 series.data。另外，对于巨大数据量的渲染（如百万以上的数据量），需要使用 appendData 进行增量加载，这种情况不支持使用 `dataset` 。

其他

目前并非所有图表都支持 dataset。支持 dataset 的图表有：`line` 、 `bar` 、 `pie` 、 `scatter` 、 `effectScatter` 、 `parallel` 、 `candlestick` 、 `map` 、 `funnel` 、 `custom` 。后续会有更多的图表进行支持。

最后，给出一个示例，多个图表共享一个 `dataset` ，并带有联动交互：
https://echarts.baidu.com/gallery/view.html?c=dataset-link&edit=1&reset=1

在 **webpack** 中使用 **ECharts**

Webpack 是目前比较流行的模块打包工具，你可以在使用 webpack 的项目中轻松的引入和打包 ECharts，这里假设你已经对 webpack 具有一定的了解并且在自己的项目中使用。

npm 安装 ECharts

在 `3.1.1` 版本之前 ECharts 在 npm 上的 package 是非官方维护的，从 `3.1.1` 开始由官方 EFE 维护 npm 上 ECharts 和 zrender 的 package。

你可以使用如下命令通过 npm 安装 ECharts

```
npm install echarts --save
```

引入 ECharts

通过 npm 上安装的 ECharts 和 zrender 会放在 `node_modules` 目录下。可以直接在项目代码中 `require('echarts')` 得到 ECharts。

```
var echarts = require('echarts');

// 基于准备好的dom，初始化echarts实例
var myChart = echarts.init(document.getElementById('main'));
// 绘制图表
myChart.setOption({
    title: {
        text: 'ECharts 入门示例'
    },
    tooltip: {},
    xAxis: {
        data: ['衬衫', '羊毛衫', '雪纺衫', '裤子', '高跟鞋', '袜子']
    },
    yAxis: {},
    series: [{
        name: '销量',
        type: 'bar',
        data: [5, 20, 36, 10, 10, 20]
    }]
});
```

按需引入 ECharts 图表和组件

默认使用 `require('echarts')` 得到的是已经加载了所有图表和组件的 ECharts 包，因此体积会比较大，如果在项目中对体积要求比较苛刻，也可以只按需引入需要的模块。

例如上面示例代码中只用到了柱状图，提示框和标题组件，因此在引入的时候也只需要引入这些模块，可以有效的将打包后的体积从 400 多 KB 减小到 170 多 KB。

```
// 引入 ECharts 主模块
var echarts = require('echarts/lib/echarts');
// 引入柱状图
require('echarts/lib/chart/bar');
// 引入提示框和标题组件
require('echarts/lib/component/tooltip');
require('echarts/lib/component/title');

// 基于准备好的dom，初始化echarts实例
var myChart = echarts.init(document.getElementById('main'));
// 绘制图表
myChart.setOption({
    title: {
        text: 'ECharts 入门示例'
    },
    tooltip: {},
    xAxis: {
        data: ['衬衫', '羊毛衫', '雪纺衫', '裤子', '高跟鞋', '袜子']
    },
    yAxis: {},
    series: [{
        name: '销量',
        type: 'bar',
        data: [5, 20, 36, 10, 10, 20]
    }]
});
```

可以按需引入的模块列表见 https://github.com/ecomfe/echarts/blob/master/index.js

对于流行的模块打包工具 browserify 也是同样的用法，这里就不赘述了。而对于使用 rollup 的自定义构建，参见 自定义构建 ECharts。

在图表中加入交互组件

除了图表外 ECharts 中，提供了很多交互组件。例如：

图例组件 legend、 标题组件 title、 视觉映射组件 visualMap、 数据区域缩放组件 dataZoom、 时间线组件 timeline

下面以 数据区域缩放组件 dataZoom 为例，介绍如何加入这种组件。

数据区域缩放组件（dataZoom）介绍

『概览数据整体，按需关注数据细节』是数据可视化的基本交互需求。 dataZoom 组件能够在直角坐标系（grid）、极坐标系（polar）中实现这一功能。

如下例子： https://echarts.baidu.com/gallery/view.html?c=doc-example/scatter-dataZoom-all&edit=1&reset=1“

- dataZoom 组件是对 数轴（axis） 进行『数据窗口缩放』『数据窗口平移』操作。

> 可以通过 dataZoom.xAxisIndex 或 dataZoom.yAxisIndex 来指定 dataZoom 控制哪个或哪些数轴。

- dataZoom 组件可同时存在多个，起到共同控制的作用。控制同一个数轴的组件，会自动联动。下面例子中会详细说明。
- dataZoom 的运行原理是通过『数据过滤』来达到『数据窗口缩放』的效果。

  数据过滤模式的设置不同，效果也不同，参见：dataZoom.filterMode。
- dataZoom 的数据窗口范围的设置，目前支持两种形式：
  - 百分比形式：参见 dataZoom.start 和 dataZoom.end。
  - 绝对数值形式：参见 dataZoom.startValue 和 dataZoom.endValue。

dataZoom 组件现在支持几种子组件：

- 内置型数据区域缩放组件（dataZoomInside）：内置于坐标系中。
- 滑动条型数据区域缩放组件（dataZoomSlider）：有单独的滑动条操作。
- 框选型数据区域缩放组件（dataZoomSelect）：全屏的选框进行数据区域缩放。入口和配置项均在 toolbox 中。

在代码加入 dataZoom 组件

先只在对单独一个横轴，加上 dataZoom 组件，代码示例如下：

```
option = {
    xAxis: {
        type: 'value'
    },
    yAxis: {
        type: 'value'
    },
    dataZoom: [
        {   // 这个dataZoom组件，默认控制x轴。
            type: 'slider', // 这个 dataZoom 组件是 slider 型 dataZoom 组件
            start: 10,      // 左边在 10% 的位置。
            end: 60         // 右边在 60% 的位置。
        }
    ],
    series: [
        {
            type: 'scatter', // 这是个『散点图』
            itemStyle: {
                opacity: 0.8
            },
            symbolSize: function (val) {
                return val[2] * 40;
            },
            data: [["14.616","7.241","0.896"],["3.958","5.701","0.955"],
["2.768","8.971","0.669"],["9.051","9.710","0.171"],["14.046","4.182","0.536"],
["12.295","1.429","0.962"],["4.417","8.167","0.113"],["0.492","4.771","0.785"],
["7.632","2.605","0.645"],["14.242","5.042","0.368"]]
        }
    ]
}
```

可以看到如下结果： https://echarts.baidu.com/gallery/view.html?c=doc-example/scatter-tutorial-dataZoom-1&edit=1&reset=1

上面的图只能拖动 dataZoom 组件导致窗口变化。如果想在坐标系内进行拖动，以及用滚轮（或移动触屏上的两指滑动）进行缩放，那么要再加上一个 inside 型的 dataZoom组件。直接在上面的 `option.dataZoom` 中增加即可：

```
option = {
```

```
    ...,
    dataZoom: [
        {   // 这个dataZoom组件，默认控制x轴。
            type: 'slider', // 这个 dataZoom 组件是 slider 型 dataZoom 组件
            start: 10,      // 左边在 10% 的位置。
            end: 60         // 右边在 60% 的位置。
        },
        {   // 这个dataZoom组件，也控制x轴。
            type: 'inside', // 这个 dataZoom 组件是 inside 型 dataZoom 组件
            start: 10,      // 左边在 10% 的位置。
            end: 60         // 右边在 60% 的位置。
        }
    ],
    ...
}
```

可以看到如下结果（能在坐标系中进行滑动，以及使用滚轮缩放了）：
https://echarts.baidu.com/gallery/view.html?c=doc-example/scatter-tutorial-dataZoom-2&edit=1&reset=1

如果想 y 轴也能够缩放，那么在 y 轴上也加上 dataZoom 组件：

```
option = {
    ...,
    dataZoom: [
        {
            type: 'slider',
            xAxisIndex: 0,
            start: 10,
            end: 60
        },
        {
            type: 'inside',
            xAxisIndex: 0,
            start: 10,
            end: 60
        },
        {
            type: 'slider',
            yAxisIndex: 0,
            start: 30,
            end: 80
```

```
        },
        {
            type: 'inside',
            yAxisIndex: 0,
            start: 30,
            end: 80
        }
    ],
    ...
}
```

可以看到如下结果： https://echarts.baidu.com/gallery/view.html?c=doc-example/scatter-tutorial-dataZoom-3&edit=1&reset=1

在图表中支持无障碍访问

W3C 制定了无障碍富互联网应用规范集（WAI-ARIA，the Accessible Rich Internet Applications Suite），致力于使得网页内容和网页应用能够被更多残障人士访问。ECharts 4.0 遵从这一规范，支持自动根据图表配置项智能生成描述，使得盲人可以在朗读设备的帮助下了解图表内容，让图表可以被更多人群访问。

默认关闭，需要通过将 `aria.show` 设置为 `true` 开启。开启后，会根据图表、数据、标题等情况，自动智能生成关于图表的描述，用户也可以通过配置项修改描述。

对于配置项：

```
option = {
    aria: {
        show: true
    },
    title: {
        text: '某站点用户访问来源',
        x: 'center'
    },
    series: [
        {
            name: '访问来源',
            type: 'pie',
            data: [
                { value: 335, name: '直接访问' },
                { value: 310, name: '邮件营销' },
                { value: 234, name: '联盟广告' },
                { value: 135, name: '视频广告' },
                { value: 1548, name: '搜索引擎' }
            ]
        }
    ]
};
```

https://echarts.baidu.com/gallery/view.html?c=doc-example/aria-pie&edit=1&reset=1

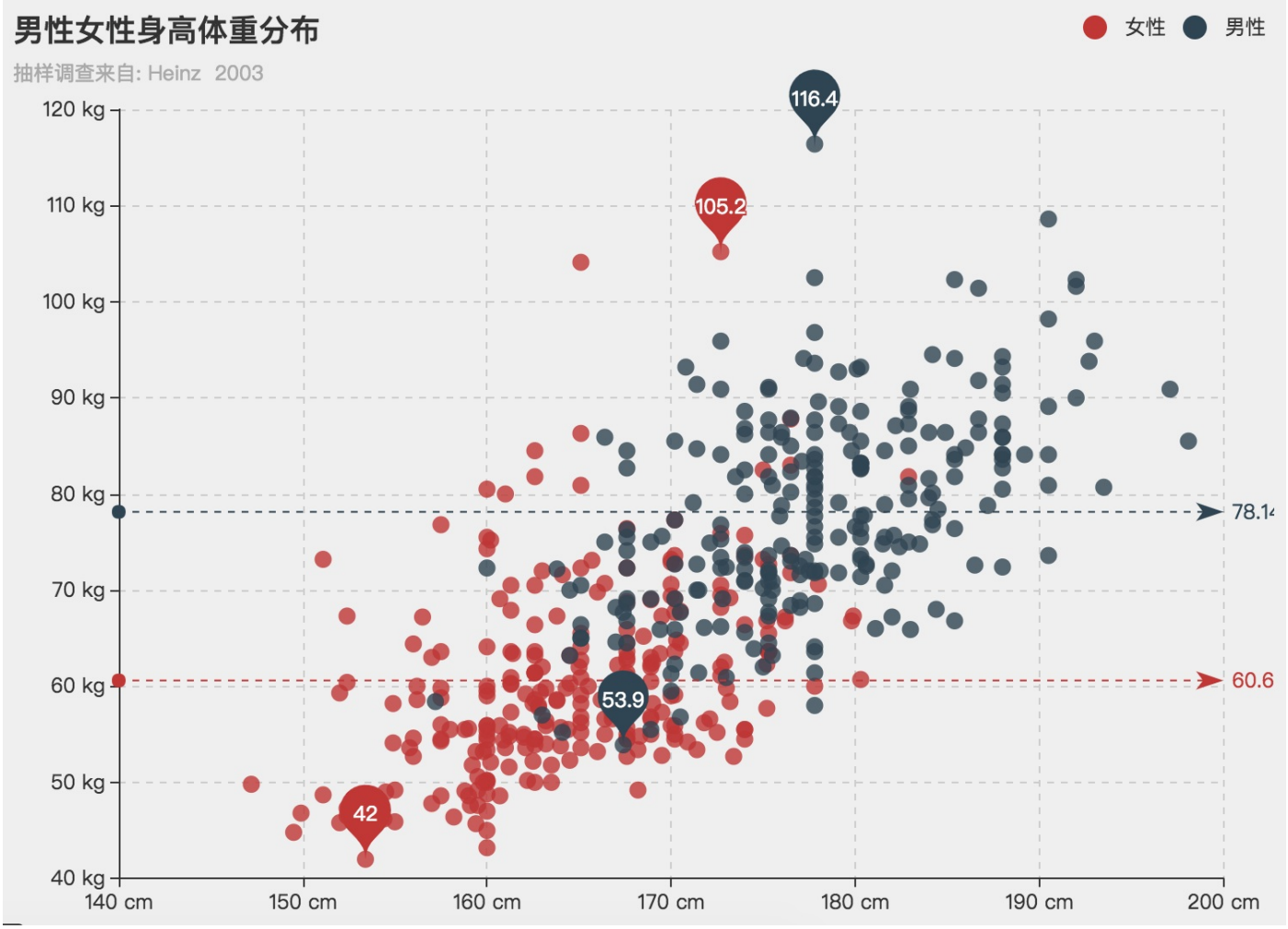
生成的图表 DOM 上，会有一个 `aria-label` 属性，在朗读设备的帮助下，盲人能够了解图表的内容。其值为：

```
这是一个关于“某站点用户访问来源”的图表。图表类型是饼图，表示访问来源。其数据是—直接访问的数据是335，邮件营销的数据是310，联盟广告的数据是234，视频广告的数据是135，搜索引擎的数据
```

```
        是1548。
```

整体修改描述

对于有些图表，默认生成的数据点的描述并不足以表现整体的信息。比如下图的散点图，默认生成的描述可以包含数据点的坐标值，但是知道几百几千个点的坐标并不能帮助我们有效地理解图表表达的信息。



这时候，用户可以通过 `aria.description` 配置项指定图表的整体描述。

定制模板描述

除了整体性修改描述之外，我们还提供了生成描述的模板，可以方便地进行细粒度的修改。

生成描述的基本流程为，如果 `aria.show` 设置为 `true` ，则生成无障碍访问描述，否则不生成。如果定义了 `aria.description`，则将其作为图表的完整描述，否则根据模板拼接生成描述。我们提供了默认的生成描述的算法，仅当生成的描述不太合适时，才需要修改这些模板，甚至使用 `aria.description` 完全覆盖。

使用模板拼接时，先根据是否存在标题 `title.text` 决定使用 `aria.general.withTitle` 还是 `aria.general.withoutTitle` 作为整体性描述。其中，`aria.general.withTitle` 配置项包括模板变量 `'{title}'`，将会被替换成图表标题。也就是说，如果 `aria.general.withTitle` 被设置为 `'图表的标题是：{title}。'`，则如果包含标题 `'价格分布图'`，这部分的描述为 `'图表的标题是：价格分布图。'`。

拼接完标题之后，会依次拼接系列的描述（`aria.series`），和每个系列的数据的描述（`aria.data`）。同样，每个模板都有可能包括模板变量，用以替换实际的值。

完整的描述生成流程请参见 ARIA 文档。

在微信小程序中使用 ECharts

我们接到了很多微信小程序开发者的反馈，表示他们强烈需要像 ECharts 这样的可视化工具。但是微信小程序是不支持 DOM 操作的，Canvas 接口也和浏览器不尽相同。

因此，我们和微信小程序官方团队合作，提供了 ECharts 的微信小程序版本。开发者可以通过熟悉的 ECharts 配置方式，快速开发图表，满足各种可视化需求。

体验示例小程序

在微信中扫描下面的二维码即可体验 ECharts Demo:

下载

为了兼容小程序 Canvas，我们提供了一个小程序的组件，用这种方式可以方便地使用 ECharts。

首先，下载 GitHub 上的 ecomfe/echarts-for-weixin 项目。

其中，`ec-canvas` 是我们提供的组件，其他文件是如何使用该组件的示例。

`ec-canvas` 目录下有一个 `echarts.js`，默认我们会在每次 `echarts-for-weixin` 项目发版的时候替换成最新版的 ECharts。如有必要，可以自行从 ECharts 项目中下载最新发布版，或者从官网自定义构建以减小文件大小。

引入组件

微信小程序的项目创建可以参见微信公众平台官方文档。

在创建项目之后，可以将下载的 ecomfe/echarts-for-weixin 项目完全替换新建的项目，然后将

修改代码；或者仅拷贝 `ec-canvas` 目录到新建的项目下，然后做相应的调整。

如果采用完全替换的方式，需要将 `project.config.json` 中的 `appid` 替换成在公众平台申请的项目 id。 `pages` 目录下的每个文件夹是一个页面，可以根据情况删除不需要的页面，并且在 `app.json` 中删除对应页面。

如果仅拷贝 `ec-canvas` 目录，则可以参考 `pages/bar` 目录下的几个文件的写法。下面，我们具体地说明。

创建图表

首先，在 `pages/bar` 目录下新建以下几个文件： `index.js` 、 `index.json` 、 `index.wxml` 、 `index.wxss` 。并且在 `app.json` 的 `pages` 中增加 `'pages/bar/index'` 。

`index.json` 配置如下：

```
{
  "usingComponents": {
    "ec-canvas": "../../ec-canvas/ec-canvas"
  }
}
```

这一配置的作用是，允许我们在 `pages/bar/index.wxml` 中使用 `<ec-canvas>` 组件。注意路径的相对位置要写对，如果目录结构和本例相同，就应该像上面这样配置。

`index.wxml` 中，我们创建了一个 `<ec-canvas>` 组件，内容如下：

```
<view class="container">
  <ec-canvas id="mychart-dom-bar" canvas-id="mychart-bar" ec="{{ ec }}"></ec-canvas>
</view>
```

其中 `ec` 是一个我们在 `index.js` 中定义的对象，它使得图表能够在页面加载后被初始化并设置。 `index.js` 的结构如下：

```
function initChart(canvas, width, height) {
  const chart = echarts.init(canvas, null, {
    width: width,
    height: height
  });
  canvas.setChart(chart);
```

```
7.
8.    var option = {
9.      ...
10.   };
11.   chart.setOption(option);
12.   return chart;
13. }
14.
15. Page({
16.   data: {
17.     ec: {
18.       onInit: initChart
19.     }
20.   }
21. });
```

这对于所有 ECharts 图表都是通用的，用户只需要修改上面 `option` 的内容，即可改变图表。 `option` 的使用方法参见 ECharts 配置项文档。对于不熟悉 ECharts 的用户，可以参见 5 分钟上手 ECharts 教程。

完整的例子请参见 ecomfe/echarts-for-weixin 项目。

暂不支持的功能

ECharts 中的绝大部分功能都支持小程序版本，因此这里仅说明不支持的功能，以及存在的问题。

以下功能尚不支持，如果有相关需求请在 issue 中向我们反馈，对于反馈人数多的需求将优先支持：

- Tooltip
- 图片
- 多个 zlevel 分层

此外，目前还有一些 bug 尚未修复，部分需要小程序团队配合上线支持，但不影响基本的使用。已知的 bug 包括：

- 安卓平台：transform 的问题（会影响关系图边两端的标记位置、旭日图文字位置等）
- iOS 平台：半透明略有变深的问题
- iOS 平台：渐变色出现在定义区域之外的地方

如有其它问题，也欢迎在 issue 中向我们反馈，谢谢！

富文本标签

在许多地方（如图、轴的标签等）都可以使用富文本标签。例如：

- https://echarts.baidu.com/gallery/view.html?c=pie-rich-text&edit=1&reset=1
- https://echarts.baidu.com/gallery/view.html?c=treemap-obama&edit=1&reset=1

其他一些例子：Map Labels,Pie Labels,Gauge.

原先 echarts 中的文本标签，只能对整块统一进行样式设置，并且仅仅支持颜色和字体的设置，从而导致不易于制作表达能力更强的文字描述信息。

echarts v3.7 以后，支持了富文本标签，能够：

- 定制文本块整体的样式（如背景、边框、阴影等）、位置、旋转等。
- 对文本块中个别片段定义样式（如颜色、字体、高宽、背景、阴影等）、对齐方式等。
- 在文本中使用图片做小图标或者背景。
- 特定组合以上的规则，可以做出简单表格、分割线等效果。 开始下面的介绍之前，先说明一下下面会使用的两个名词的含义：
- 文本块（Text Block）：文本标签块整体。
- 文本片段（Text Fregment）：文本标签块中的部分文本。 如下图示例：https://echarts.baidu.com/gallery/view.html?c=doc-example/text-block-fregment&edit=1&reset=1

文本样式相关的配置项

echarts 提供了丰富的文本标签配置项，包括：

- 字体基本样式设置：`fontStyle`、`fontWeight`、`fontSize`、`fontFamily`。
- 文字颜色：`color`。
- 文字描边：`textBorderColor`、`textBorderWidth`。
- 文字阴影：`textShadowColor`、`textShadowBlur`、`textShadowOffsetX`、`textShadowOffsetY`。
- 文本块或文本片段大小：`lineHeight`、`width`、`height`、`padding`。
- 文本块或文本片段的对齐：`align`、`verticalAlign`。
- 文本块或文本片段的边框、背景（颜色或图片）：`backgroundColor`、`borderColor`、`borderWidth`、`borderRadius`。

- 文本块或文本片段的阴影：`shadowColor`、`shadowBlur`、`shadowOffsetX`、`shadowOffsetY`。
- 文本块的位置和旋转：`position`、`distance`、`rotate`。 可以在各处的 `rich` 属性中定义文本片段样式。例如 `series-bar.label.rich`

例如：

```
label: {
    // 在文本中，可以对部分文本采用 rich 中定义样式。
    // 这里需要在文本中使用标记符号：
    // `{styleName|text content text content}` 标记样式名。
    // 注意，换行仍是使用 '\n'。
    formatter: [
        '{a|这段文本采用样式a}',
        '{b|这段文本采用样式b}这段用默认样式{x|这段用样式x}'
    ].join('\n'),

    // 这里是文本块的样式设置：
    color: '#333',
    fontSize: 5,
    fontFamily: 'Arial',
    borderWidth: 3,
    backgroundColor: '#984455',
    padding: [3, 10, 10, 5],
    lineHeight: 20,

    // rich 里是文本片段的样式设置：
    rich: {
        a: {
            color: 'red',
            lineHeight: 10
        },
        b: {
            backgroundColor: {
                image: 'xxx/xxx.jpg'
            },
            height: 40
        },
        x: {
            fontSize: 18,
            fontFamily: 'Microsoft YaHei',
            borderColor: '#449933',
```

```
            borderRadius: 4
        },
        ...
    }
}
```

> 注意：如果不定义 `rich`，不能指定文字块的 `width` 和 `height`。

文本、文本框、文本片段的基本样式和装饰

每个文本可以设置基本的字体样式：`fontStyle`、`fontWeight`、`fontSize`、`fontFamily`。

可以设置文字的颜色 `color` 和边框的颜色 `textBorderColor`、`textBorderWidth`。

文本框可以设置边框和背景的样式：`borderColor`、`borderWidth`、`backgroundColor`、`padding`。

文本片段也可以设置边框和背景的样式：`borderColor`、`borderWidth`、`backgroundColor`、`padding`。

例如：https://echarts.baidu.com/gallery/view.html?c=doc-example/text-options&edit=1&reset=1“ width=”700” height=”300”>

标签的位置

对于折线图、柱状图、散点图等，均可以使用 `label` 来设置标签。标签的相对于图形元素的位置，一般使用 label.position、label.distance 来配置。

例如：https://echarts.baidu.com/gallery/view.html?c=doc-example/label-position&edit=1&reset=1

> 注意：`position` 在不同的图中可取值有所不同。`distance` 并不是在每个图中都支持。详情请参见 option 文档。

标签的旋转

某些图中，为了能有足够长的空间来显示标签，需要对标签进行旋转。例如：https://echarts.baidu.com/gallery/view.html?c=bar-label-rotation&edit=1&reset=1

这种场景下，可以结合 align 和 verticalAlign 来调整标签位置。

注意，逻辑是，先使用 `align` 和 `verticalAlign` 定位，再旋转。

文本片段的排版和对齐

关于排版方式，每个文本片段，可以想象成 CSS 中的 `inline-block`，在文档流中按行放置。

每个文本片段的内容盒尺寸（`content box size`），默认是根据文字大小决定的。但是，也可以设置 `width`、`height` 来强制指定，虽然一般不会这么做（参见下文）。文本片段的边框盒尺寸（`border box size`），由上述本身尺寸，加上文本片段的 `padding` 来得到。

只有 `'\n'` 是换行符，能导致换行。

一行内，会有多个文本片段。每行的实际高度，由 `lineHeight` 最大的文本片段决定。文本片段的 `lineHeight` 可直接在 `rich` 中指定，也可以在 `rich` 的父层级中统一指定而采用到 `rich` 的所有项中，如果都不指定，则取文本片段的边框盒尺寸（`border box size`）。

在一行的 `lineHeight` 被决定后，一行内，文本片段的竖直位置，由文本片段的 `verticalAlign` 来指定（这里和 CSS 中的规则稍有不同）：

- `'bottom'`：文本片段的盒的底边贴住行底。
- `'top'`：文本片段的盒的顶边贴住行顶。
- `'middle'`：居行中。 文本块的宽度，可以直接由文本块的 `width` 指定，否则，由最长的行决定。宽度决定后，在一行中进行文本片段的放置。文本片段的 `align` 决定了文本片段在行中的水平位置：
- 首先，从左向右连续紧靠放置 `align` 为 `'left'` 的文本片段盒。
- 然后，从右向左连续紧靠放置 `align` 为 `'right'` 的文本片段盒。
- 最后，剩余的没处理的文本片段盒，紧贴着，在中间剩余的区域中居中放置。 关于文字在文本片段盒中的位置：
- 如果 `align` 为 `'center'`，则文字在文本片段盒中是居中的。
- 如果 `align` 为 `'left'`，则文字在文本片段盒中是居左的。
- 如果 `align` 为 `'right'`，则文字在文本片段盒中是居右的。 例如：https://echarts.baidu.com/gallery/view.html?c=doc-example/text-fregment-align&edit=1&reset=1

特殊效果：图标、分割线、标题块、简单表格

看下面的例子： https://echarts.baidu.com/gallery/view.html?c=doc-example/title-block&edit=1&reset=1“ 文本片段的 `backgroundColor` 可以指定为图片

后，就可以在文本中使用图标了：

```
rich: {
    Sunny: {
        // 这样设定 backgroundColor 就可以是图片了。
        backgroundColor: {
            image: './data/asset/img/weather/sunny_128.png'
        },
        // 可以只指定图片的高度，从而图片的宽度根据图片的长宽比自动得到。
        height: 30
    }
}
```

分割线实际是用 border 实现的：

```
rich: {
    hr: {
        borderColor: '#777',
        // 这里把 width 设置为 '100%'，表示分割线的长度充满文本块。
        // 注意，这里是文本块内容盒（content box）的 100%，而不包含 padding。
        // 虽然这和 CSS 相关的定义有所不同，但是在这类场景中更加方便。
        width: '100%',
        borderWidth: 0.5,
        height: 0
    }
}
```

标题块是使用 `backgroundColor` 实现的：

```
// 标题文字居左
formatter: '{titleBg|Left Title}',
rich: {
    titleBg: {
        backgroundColor: '#000',
        height: 30,
        borderRadius: [5, 5, 0, 0],
        padding: [0, 10, 0, 10],
        width: '100%',
        color: '#eee'
    }
}

```

```
// 标题文字居中。
// 这个实现有些 tricky，但是，能够不引入更复杂的排版规则而实现这个效果。
formatter: '{tc|Center Title}{titleBg|}',
rich: {
    titleBg: {
        align: 'right',
        backgroundColor: '#000',
        height: 30,
        borderRadius: [5, 5, 0, 0],
        padding: [0, 10, 0, 10],
        width: '100%',
        color: '#eee'
    }
}
```

简单表格的设定，其实就是给不同行上纵向对应的文本片段设定同样的宽度就可以了。参见本文最开始的 例子。

小例子：实现日历图

在ECharts中，我们新增了日历坐标系，如何快速写出一个日历图呢？
https://echarts.baidu.com/gallery/view.html?c=calendar-simple&edit=1&reset=1

通过以下三个步骤即可实现上述效果：

第一步：引入js文件

下载的最新完整版本echarts.min.js即可，无需再单独引入其他文件哦

```
<script src="echarts.min.js"></script>
<script>
    // ...
</script>
```

第二步：指定DOM元素作为图表容器

和ECharts中的其他图表一样，创建一个DOM来作为绘制图表的容器

```
<div id="main" style="width=100%; height = 400px"></div>
```

使用ECharts进行初始化

```
var myChart = echarts.init(document.getElementById('main'));
```

第三步：配置参数

以常见的日历图为例：calendar坐标 + heatmap图

```
var option = {
    visualMap: {
        show: false
        min: 0,
        max: 1000
    },
    calendar: {
        range: '2017'
    },
```

```
    series: {
        type: 'heatmap',
        coordinateSystem: 'calendar',
        data: [['2017-01-02', 900], ['2017-01-02', 877], ['2017-01-02', 699], ...]
    }
}
myChart.setOption(option);
```

在heatmap图的基础上，加上 `coordinateSystem: 'calendar',` 和 `calendar: { range: '2017' }` heatmap图就秒变为日历图了。

> 若发现图表没有正确显示，你可以检查以下几种可能：- JS文件是否正确加载；- `echarts` 变量是否存在；- 控制台是否报错；- DOM 元素在 `echarts.init` 的时候是否有高度和宽度。- 若为 `type: heatmap` ，检查是否配置了 `visualMap` 。

附完整示例代码

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>ECharts</title>
    <script src="echarts.min.js"></script>
</head>
<body>
    <div id="main" style="width:100%;height:400px;"></div>
    <script type="text/javascript">
        var myChart = echarts.init(document.getElementById('main'));

        // 模拟数据
        function getVirtulData(year) {
            year = year || '2017';
            var date = +echarts.number.parseDate(year + '-01-01');
            var end = +echarts.number.parseDate(year + '-12-31');
            var dayTime = 3600 * 24 * 1000;
            var data = [];
            for (var time = date; time <= end; time += dayTime) {
                data.push([
                    echarts.format.formatTime('yyyy-MM-dd', time),
                    Math.floor(Math.random() * 10000)
                ]);
            }
```

```
            return data;
        }
        var option = {
            visualMap: {
                show: false,
                min: 0,
                max: 10000
            },
            calendar: {
                range: '2017'
            },
            series: {
                type: 'heatmap',
                coordinateSystem: 'calendar',
                data: getVirtulData(2017)
            }
        };
        myChart.setOption(option);
    </script>
</body>
</html>
```

以上就是绘制最简日历图的步骤了，如若还想进一步私人定制，还可以通过自定义配置参数来实现

自定义配置参数

使用日历坐标绘制日历图时，支持自定义各项属性：

- range: 设置时间的范围，可支持某年、某月、某天，还可支持跨年
- cellSize: 设置日历格的大小，可支持设置不同高宽，还可支持自适应auto
- width、height: 也可以直接设置改日历图的整体高宽，让其基于已有的高宽全部自适应
- orient: 设置坐标的方向，既可以横着也可以竖着
- splitLine: 设置分隔线样式，也可以直接不显示
- itemStyle: 设置日历格的样式，背景色、方框线颜色大小类型、透明度均可自定义，甚至还能加阴影
- dayLabel: 设置坐标中 星期样式，可以设置星期从第几天开始，快捷设置中英文、甚至是自定义中英文混搭、或局部不显示、通过formatter 可以想怎么显示就怎么显示；
- monthLabel: 设置坐标中 月样式，和星期一样，可快捷设置中英文和自定义混搭

- yearLabel: 设置坐标中 年样式，默认显示一年，通过formatter 文字可以想显示什么就能通过string function任性自定义，上下左右方位随便选；

完整的配置项参数参见：calendar API

日历坐标系的其他形式

日历坐标系是一种新的 ECharts 坐标系，提供了在日历上绘制图表的能力；所以除了制作常用的日历图外，我们可以在热力图、散点图、关系图中使用日历坐标系。

在日历坐标系中使用热力图： https://echarts.baidu.com/gallery/view.html?c=calendar-heatmap&edit=1&reset=1

在日历坐标系中使用散点图： https://echarts.baidu.com/gallery/view.html?c=calendar-effectscatter&edit=1&reset=1

还可以混合放置不同的图表，例如下例子，同时放置了热力图和关系图：
https://echarts.baidu.com/gallery/view.html?c=calendar-graph&edit=1&reset=1

其他更丰富的效果

灵活利用 ECharts 图表和坐标系的组合，以及 API，还可以实现更丰富的效果。

例如，制作农历： https://echarts.baidu.com/gallery/view.html?c=calendar-lunar&edit=1&reset=1

例如，使用 chart.convertToPixel 接口，在日历坐标系绘制饼图。
https://echarts.baidu.com/gallery/view.html?c=calendar-pie&edit=1&reset=1

小例子：自己实现拖拽

介绍一个实现拖拽的小例子。这个例子是在原生 echarts 基础上做了些小小扩展，带有一定的交互性。通过这个例子，我们可以了解到，如何使用 echarts 提供的 API 实现定制化的富有交互的功能。 https://echarts.baidu.com/gallery/view.html?c=line-draggable&edit=1&reset=1

这个例子主要做到了这样一件事，用鼠标可以拖拽曲线的点，从而改变曲线的形状。例子很简单，但是有了这个基础我们还可以做更多的事情，比如在图中可视化得编辑。所以我们从这个简单的例子开始。

echarts 本身没有提供封装好的『拖拽改变图表』功能，因为现在认为这个功能并不足够有通用性。那么这个功能就留给开发者用 API 实现，这也有助于开发者按自己的需要个性定制。

（一）实现基本的拖拽功能

在这个例子中，基础的图表是一个 折线图 (series-line)。参见如下配置：

```
var symbolSize = 20;

// 这个 data 变量在这里单独声明，在后面也会用到。
var data = [[15, 0], [-50, 10], [-56.5, 20], [-46.5, 30], [-22.1, 40]];

myChart.setOption({
    xAxis: {
        min: -100,
        max: 80,
        type: 'value',
        axisLine: {onZero: false}
    },
    yAxis: {
        min: -30,
        max: 60,
        type: 'value',
        axisLine: {onZero: false}
    },
    series: [
        {
            id: 'a',
            type: 'line',
            smooth: true,
            symbolSize: symbolSize, // 为了方便拖拽，把 symbolSize 尺寸设大了。
            data: data
```

```
        }
    ]
});
```

既然折线中原生的点没有拖拽功能，我们就为它加上拖拽功能：用 `graphic` 组件，在每个点上面，覆盖一个隐藏的可拖拽的圆点。

```
myChart.setOption({
    // 声明一个 graphic component，里面有若干个 type 为 'circle' 的 graphic elements。
    // 这里使用了 echarts.util.map 这个帮助方法，其行为和 Array.prototype.map 一样，但是兼容 es5 以下的环境。
    // 用 map 方法遍历 data 的每项，为每项生成一个圆点。
    graphic: echarts.util.map(data, function (dataItem, dataIndex) {
        return {
            // 'circle' 表示这个 graphic element 的类型是圆点。
            type: 'circle',

            shape: {
                // 圆点的半径。
                r: symbolSize / 2
            },
            // 用 transform 的方式对圆点进行定位。position: [x, y] 表示将圆点平移到 [x, y] 位置。
            // 这里使用了 convertToPixel 这个 API 来得到每个圆点的位置，下面介绍。
            position: myChart.convertToPixel('grid', dataItem),

            // 这个属性让圆点不可见（但是不影响他响应鼠标事件）。
            invisible: true,
            // 这个属性让圆点可以被拖拽。
            draggable: true,
            // 把 z 值设得比较大，表示这个圆点在最上方，能覆盖住已有的折线图的圆点。
            z: 100,
            // 此圆点的拖拽的响应事件，在拖拽过程中会不断被触发。下面介绍详情。
            // 这里使用了 echarts.util.curry 这个帮助方法，意思是生成一个与 onPointDragging
            // 功能一样的新的函数，只不过第一个参数永远为此时传入的 dataIndex 的值。
            ondrag: echarts.util.curry(onPointDragging, dataIndex)
        };
    })
});
```

上面的代码中，使用 convertToPixel 这个 API，进行了从 data 到『像素坐标』的转换，从而得到了每个圆点应该在的位置，从而能绘制这些圆点。 `myChart.convertToPixel('grid', dataItem)` 这句话中，第一个参数 `'grid'` 表示 `dataItem` 在 grid 这个组件中（即直角坐标系）中进行转换。所谓『像素坐标』，就是以 echarts 容器 dom element 的左上角为零点的以像素为单位的坐标系中的坐标。

注意这件事需要在第一次 setOption 后再进行，也就是说，须在坐标系（grid）初始化后才能调用 `myChart.convertToPixel('grid', dataItem)` 。

有了这段代码后，就有了诸个能拖拽的点。接下来要为每个点，加上拖拽响应的事件：

```
// 拖拽某个圆点的过程中会不断调用此函数。
// 此函数中会根据拖拽后的新位置，改变 data 中的值，并用新的 data 值，重绘折线图，从而使折线图同步于被拖拽的隐藏圆点。
function onPointDragging(dataIndex) {
    // 这里的 data 就是本文最初的代码块中声明的 data，在这里会被更新。
    // 这里的 this 就是被拖拽的圆点。this.position 就是圆点当前的位置。
    data[dataIndex] = myChart.convertFromPixel('grid', this.position);
    // 用更新后的 data，重绘折线图。
    myChart.setOption({
        series: [{
            id: 'a',
            data: data
        }]
    });
}
```

上面的代码中，使用了 convertFromPixel 这个 API。它是 convertToPixel 的逆向过程。 `myChart.convertFromPixel('grid', this.position)` 表示把当前像素坐标转换成 grid 组件中直角坐标系的 dataItem 值。

最后，为了使 dom 尺寸改变时，图中的元素能自适应得变化，加上这些代码：

```
window.addEventListener('resize', function () {
    // 对每个拖拽圆点重新计算位置，并用 setOption 更新。
    myChart.setOption({
        graphic: echarts.util.map(data, function (item, dataIndex) {
            return {
                position: myChart.convertToPixel('grid', item)
            };
        })
    });
```

```
});
```

（二）添加 tooltip 组件

到此，拖拽的基本功能就完成了。但是想要更进一步得实时看到拖拽过程中，被拖拽的点的 data 值的变化状况，我们可以使用 tooltip 组件来实时显示这个值。但是，tooltip 有其默认的『显示』『隐藏』触发规则，在我们拖拽的场景中并不适用，所以我们还要手动定制 tooltip 的『显示』『隐藏』行为。

在上述代码中分别添加如下定义：

```
myChart.setOption({
    ...,
    tooltip: {
        // 表示不使用默认的『显示』『隐藏』触发规则。
        triggerOn: 'none',
        formatter: function (params) {
            return 'X: ' + params.data[0].toFixed(2) + '<br>Y: ' +
params.data[1].toFixed(2);
        }
    }
});
```

```
myChart.setOption({
    graphic: echarts.util.map(data, function (item, dataIndex) {
        return {
            type: 'circle',
            ...,
            // 在 mouseover 的时候显示，在 mouseout 的时候隐藏。
            onmousemove: echarts.util.curry(showTooltip, dataIndex),
            onmouseout: echarts.util.curry(hideTooltip, dataIndex),
        };
    })
});

function showTooltip(dataIndex) {
    myChart.dispatchAction({
        type: 'showTip',
        seriesIndex: 0,
        dataIndex: dataIndex
    });
```

```
}

function hideTooltip(dataIndex) {
    myChart.dispatchAction({
        type: 'hideTip'
    });
}
```

这里使用了 dispatchAction 来显示隐藏 tooltip。用到了 showTip、hideTip。

（三）全部代码

总结一下，全部的代码如下：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <script src="http://echarts.baidu.com/dist/echarts.min.js"></script>
</head>
<body>
    <div id="main" style="width: 600px;height:400px;"></div>
    <script type="text/javascript">

    var symbolSize = 20;
    var data = [[15, 0], [-50, 10], [-56.5, 20], [-46.5, 30], [-22.1, 40]];

    var myChart = echarts.init(document.getElementById('main'));

    myChart.setOption({
        tooltip: {
            triggerOn: 'none',
            formatter: function (params) {
                return 'X: ' + params.data[0].toFixed(2) + '<br>Y: ' +
params.data[1].toFixed(2);
            }
        },
        xAxis: {
            min: -100,
            max: 80,
            type: 'value',
            axisLine: {onZero: false}
```

```
        },
        yAxis: {
            min: -30,
            max: 60,
            type: 'value',
            axisLine: {onZero: false}
        },
        series: [
            {
                id: 'a',
                type: 'line',
                smooth: true,
                symbolSize: symbolSize,
                data: data
            }
        ],
    });

    myChart.setOption({
        graphic: echarts.util.map(data, function (item, dataIndex) {
            return {
                type: 'circle',
                position: myChart.convertToPixel('grid', item),
                shape: {
                    r: symbolSize / 2
                },
                invisible: true,
                draggable: true,
                ondrag: echarts.util.curry(onPointDragging, dataIndex),
                onmousemove: echarts.util.curry(showTooltip, dataIndex),
                onmouseout: echarts.util.curry(hideTooltip, dataIndex),
                z: 100
            };
        })
    });

    window.addEventListener('resize', function () {
        myChart.setOption({
            graphic: echarts.util.map(data, function (item, dataIndex) {
                return {
                    position: myChart.convertToPixel('grid', item)
                };
```

```
            })
        });
    });

    function showTooltip(dataIndex) {
        myChart.dispatchAction({
            type: 'showTip',
            seriesIndex: 0,
            dataIndex: dataIndex
        });
    }

    function hideTooltip(dataIndex) {
        myChart.dispatchAction({
            type: 'hideTip'
        });
    }

    function onPointDragging(dataIndex, dx, dy) {
        data[dataIndex] = myChart.convertFromPixel('grid', this.position);
        myChart.setOption({
            series: [{
                id: 'a',
                data: data
            }]
        });
    }

</script>
</body>
</html>
```

有了这些基础，就可以定制更多的功能了。可以加 `dataZoom` 组件，可以制作一个直角坐标系上的绘图板等等。可以发挥想象力。

异步数据加载和更新

异步加载

入门示例中的数据是在初始化后 `setOption` 中直接填入的，但是很多时候可能数据需要异步加载后再填入。 `ECharts` 中实现异步数据的更新非常简单，在图表初始化后不管任何时候只要通过 jQuery 等工具异步获取数据后通过 `setOption` 填入数据和配置项就行。

```
var myChart = echarts.init(document.getElementById('main'));

$.get('data.json').done(function (data) {
    myChart.setOption({
        title: {
            text: '异步数据加载示例'
        },
        tooltip: {},
        legend: {
            data:['销量']
        },
        xAxis: {
            data: ["衬衫","羊毛衫","雪纺衫","裤子","高跟鞋","袜子"]
        },
        yAxis: {},
        series: [{
            name: '销量',
            type: 'bar',
            data: [5, 20, 36, 10, 10, 20]
        }]
    });
});
```

或者先设置完其它的样式，显示一个空的直角坐标轴，然后获取数据后填入数据。

```
var myChart = echarts.init(document.getElementById('main'));
// 显示标题，图例和空的坐标轴
myChart.setOption({
    title: {
        text: '异步数据加载示例'
    },
    tooltip: {},
    legend: {
```

```
        data:['销量']
    },
    xAxis: {
        data: []
    },
    yAxis: {},
    series: [{
        name: '销量',
        type: 'bar',
        data: []
    }]
});

// 异步加载数据
$.get('data.json').done(function (data) {
    // 填入数据
    myChart.setOption({
        xAxis: {
            data: data.categories
        },
        series: [{
            // 根据名字对应到相应的系列
            name: '销量',
            data: data.data
        }]
    });
});
```

如下： https://echarts.baidu.com/gallery/view.html?c=doc-example/tutorial-async&edit=1&reset=1

ECharts 中在更新数据的时候需要通过 `name` 属性对应到相应的系列，上面示例中如果 `name` 不存在也可以根据系列的顺序正常更新，但是更多时候推荐更新数据的时候加上系列的 `name` 数据。

loading 动画

如果数据加载时间较长，一个空的坐标轴放在画布上也会让用户觉得是不是产生 bug 了，因此需要一个 loading 的动画来提示用户数据正在加载。

ECharts 默认有提供了一个简单的加载动画。只需要调用 showLoading 方法显示。数据加载完成后再调用 hideLoading 方法隐藏加载动画。

```
myChart.showLoading();
$.get('data.json').done(function (data) {
    myChart.hideLoading();
    myChart.setOption(...);
});
```

效果如下： https://echarts.baidu.com/gallery/view.html?c=doc-example/tutorial-loading&edit=1&reset=1

数据的动态更新

ECharts 由数据驱动，数据的改变驱动图表展现的改变，因此动态数据的实现也变得异常简单。

所有数据的更新都通过 setOption实现，你只需要定时获取数据，setOption 填入数据，而不用考虑数据到底产生了那些变化，ECharts 会找到两组数据之间的差异然后通过合适的动画去表现数据的变化。

> ECharts 3 中移除了 ECharts 2 中的 addData 方法。如果只需要加入单个数据，可以先 data.push(value) 后 setOption

具体可以看下面示例： https://echarts.baidu.com/gallery/view.html?c=doc-example/tutorial-dynamic-data&edit=1&reset=1

数据的视觉映射

数据可视化是 数据 到 视觉元素 的映射过程（这个过程也可称为视觉编码，视觉元素也可称为视觉通道）。

ECharts 的每种图表本身就内置了这种映射过程，比如折线图把数据映射到『线』，柱状图把数据映射到『长度』。一些更复杂的图表，如 `graph` 、 `事件河流图` 、 `treemap` 也都会做出他们内置的映射。

此外，ECharts 还提供了 visualMap 组件 来提供通用的视觉映射。 `visualMap` 组件中可以使用的视觉元素有： `图形类别（symbol）` 、 `图形大小（symbolSize）` `颜色（color）` 、 `透明度（opacity）` 、 `颜色透明度（colorAlpha）` 、 `颜色明暗度（colorLightness）` 、 `颜色饱和度（colorSaturation）` 、 `色调（colorHue）`

下面对 `visualMap` 组件的使用方式进行简要的介绍。

数据和维度

ECharts中的数据，一般存放于 series.data 中。根据图表类型不同，数据的具体形式也可能有些许差异。比如可能是『线性表』、『树』、『图』等。但他们都有个共性：都是『数据项（dataItem）』的集合。每个数据项含有『数据值（value）』和其他信息（如果需要的话）。每个数据值，可以是单一的数值（一维）或者一个数组（多维）。

例如，series.data 最常见的形式，是『线性表』，即一个普通数组：

```
series: {
    data: [
        {       // 这里每一个项就是数据项（dataItem）
            value: 2323, // 这是数据项的数据值（value）
            itemStyle: {...}
        },
        1212,   // 也可以直接是 dataItem 的 value，这更常见。
        2323,   // 每个 value 都是『一维』的。
        4343,
        3434
    ]
}
```

```
series: {
    data: [
        {                       // 这里每一个项就是数据项（dataItem）
            value: [3434, 129,  '圣马力诺'], // 这是数据项的数据值（value）
```

```
            itemStyle: {...}
        },
        [1212, 5454, '梵蒂冈'],    // 也可以直接是 dataItem 的 value，这更常见。
        [2323, 3223, '瑙鲁'],      // 每个 value 都是『三维』的，每列是一个维度。
        [4343, 23,   '图瓦卢']     // 假如是『气泡图』，常见第一维度映射到x轴，
                                   // 第二维度映射到y轴，
                                   // 第三维度映射到气泡半径（symbolSize）
    ]
}
```

在图表中，往往默认把 value 的前一两个维度进行映射，比如取第一个维度映射到x轴，取第二个维度映射到y轴。如果想要把更多的维度展现出来，可以借助 `visualMap` 。最常见的情况，气泡图（scatter） 使用半径展现了第三个维度。

visualMap 组件

visualMap 组件定义了把数据的『哪个维度』映射到『什么视觉元素上』。

现在提供如下两种类型的visualMap组件，通过 visualMap.type 来区分。

其定义结构例如：

```
option = {
    visualMap: [ // 可以同时定义多个 visualMap 组件。
        { // 第一个 visualMap 组件
            type: 'continuous', // 定义为连续型 viusalMap
            ...
        },
        { // 第二个 visualMap 组件
            type: 'piecewise', // 定义为分段型 visualMap
            ...
        }
    ],
    ...
};
```

连续型（visualMapContinuous）： https://echarts.baidu.com/gallery/view.html?c=doc-example/map-visualMap-continuous&edit=1&reset=1“ 分段型（visualMapPiecewise）： https://echarts.baidu.com/gallery/view.html?c=doc-example/scatter-visualMap-piecewise&edit=1&reset=1“ 分段型视觉映射组件（visualMapPiecewise），有三种模式：

- 连续型数据平均分段： 依据 visualMap-piecewise.splitNumber 来自动平均分割成若干块。
- 连续型数据自定义分段： 依据 visualMap-piecewise.pieces 来定义每块范围。
- 离散数据（类别性数据）： 类别定义在 visualMap-piecewise.categories 中。 视觉映射方式的配置

既然是『数据』到『视觉元素』的映射， `visualMap` 中可以指定数据的『哪个维度』（参见 visualMap.dimension）映射到哪些『视觉元素』（参见 visualMap.inRange 和 visualMap.outOfRange）中。

例一：

```
option = {
    visualMap: [
        {
            type: 'piecewise'
            min: 0,
            max: 5000,
            dimension: 3,       // series.data 的第四个维度（即 value[3]）被映射
            seriesIndex: 4,     // 对第四个系列进行映射。
            inRange: {          // 选中范围中的视觉配置
                color: ['blue', '#121122', 'red'], // 定义了图形颜色映射的颜色列表，
                                                   // 数据最小值映射到'blue'上，
                                                   // 最大值映射到'red'上，
                                                   // 其余自动线性计算。
                symbolSize: [30, 100]              // 定义了图形尺寸的映射范围，
                                                   // 数据最小值映射到30上，
                                                   // 最大值映射到100上，
                                                   // 其余自动线性计算。
            },
            outOfRange: {       // 选中范围外的视觉配置
                symbolSize: [30, 100]
            }
        },
        ...
    ]
};
```

例二：

```
option = {
    visualMap: [
```

```
        {
            ...,
            inRange: {          // 选中范围中的视觉配置
                colorLightness: [0.2, 1], // 映射到明暗度上。也就是对本来的颜色进行明暗度处理。
                                          // 本来的颜色可能是从全局色板中选取的颜色，visualMap组件并不关心。
                symbolSize: [30, 100]
            },
            ...
        },
        ...
    ]
};
```

更多详情，参见 visualMap.inRange 和 visualMap.outOfRange。

旭日图

旭日图（Sunburst）由多层的环形图组成，在数据结构上，内圈是外圈的父节点。因此，它既能像饼图一样表现局部和整体的占比，又能像矩形树图一样表现层级关系。
https://echarts.baidu.com/gallery/view.html?c=sunburst-monochrome&edit=1&reset=1

引入相关文件

旭日图是 ECharts 4.0 新增的图表类型，在官网下载页面下载“完整版” ECharts，并将下载的 JavaScript 文件引入即可创建旭日图。

最简单的旭日图

创建旭日图需要在 `series` 配置项中声明类型为 `'sunburst'` 的系列，并且以树形结构声明其 `data`：

```
var option = {
    series: {
        type: 'sunburst',
        data: [{
            name: 'A',
            value: 10,
            children: [{
                value: 3,
                name: 'Aa'
            }, {
                value: 5,
                name: 'Ab'
            }]
        }, {
            name: 'B',
            children: [{
                name: 'Ba',
                value: 4
            }, {
                name: 'Bb',
                value: 2
            }]
        }, {
            name: 'C',
```

```
            value: 3
        }]
    }
};
```

得到以下结果： https://echarts.baidu.com/gallery/view.html?c=doc-example/sunburst-simple&edit=1&reset=1

颜色等样式调整

默认情况下会使用全局调色盘 color 分配最内层的颜色，其余层则与其父元素同色。在旭日图中，扇形块的颜色有以下三种设置方式：

- 在 series.data.itemStyle 中设置每个扇形块的样式；
- 在 series.levels.itemStyle 中设置每一层的样式；
- 在 series.itemStyle 中设置整个旭日图的样式。 上述三者的优先级是从高到低的，也就是说，配置了 `series.data.itemStyle` 的扇形块将会覆盖 `series.levels.itemStyle` 和 `series.itemStyle` 的设置。

下面，我们将整体的颜色设为灰色 `'#aaa'` ，将最内层的颜色设为蓝色 `'blue'` ，将 `Aa` 、 `B` 这两块设为红色 `'red'` 。

```
var option = {
    series: {
        type: 'sunburst',
        data: [{
            name: 'A',
            value: 10,
            children: [{
                value: 3,
                name: 'Aa',
                itemStyle: {
                    color: 'red'
                }
            }, {
                value: 5,
                name: 'Ab'
            }]
        }, {
            name: 'B',
            children: [{
                name: 'Ba',
```

```
                value: 4
            }, {
                name: 'Bb',
                value: 2
            }],
            itemStyle: {
                color: 'red'
            }
        }, {
            name: 'C',
            value: 3
        }],
        itemStyle: {
            color: '#aaa'
        },
        levels: [{
            // 留给数据下钻的节点属性
        }, {
            itemStyle: {
                color: 'blue'
            }
        }]
    }
};
```

效果为： https://echarts.baidu.com/gallery/view.html?c=doc-example/sunburst-color&edit=1&reset=1

按层配置样式

旭日图是一种有层次的结构，为了方便同一层样式的配置，我们提供了 `levels` 配置项。它是一个数组，其中的第 `0` 项表示数据下钻后返回上级的图形，其后的每一项分别表示从圆心向外层的层级。

例如，假设我们没有数据下钻功能，并且希望将最内层的扇形块的颜色设为红色，文字设为蓝色，可以这样设置：

```
series: {
    // ...
    levels: [
        {
            // 留给数据下钻点的空白配置
        },
```

```
        {
            // 最靠内测的第一层
            itemStyle: {
                color: 'red'
            },
            label: {
                color: 'blue'
            }
        },
        {
            // 第二层 ...
        }
    ]
}
```

在实际使用的过程中，你会发现按层配置样式是一个很常用的功能，能够很大程度上提高配置的效率。

数据下钻

旭日图默认支持数据下钻，也就是说，当点击了扇形块之后，将以该扇形块的数据作为根节点，便于进一步了解该数据的细节。 https://echarts.baidu.com/gallery/view.html?c=sunburst-simple&edit=1&reset=1

当数据下钻后，中间会出现一个用于返回上一层的图形，该图形的样式可以通过 `levels[0]` 配置。

如果不需要数据下钻功能，可以通过将 `nodeClick` 设置为 `false` 关闭；或者将其设为 `'link'` ，并将 `data.link` 设为点击扇形块对应打开的链接。

高亮相关扇形块

旭日图支持鼠标移动到某扇形块时，高亮相关数据块的操作，可以通过设置 `highlightPolicy`，包括以下几种高亮方式：

- `'descendant'` （默认值）：高亮鼠标移动所在扇形块与其后代元素；
- `'ancestor'` ：高亮鼠标所在扇形块与其祖先元素；
- `'self'` ：仅高亮鼠标所在扇形块；
- `'none'` ：不会淡化（downplay）其他元素。 上面提到的“高亮”，对于鼠标所在的扇形块，会使用 `emphasis` 样式；对于其他相关扇形块，则会使用 `highlight` 样式。通过这种方式，可以很方便地实现突出显示相关数据的需求。

具体来说，对于配置项：

```
itemStyle: {
    color: 'yellow',
    borderWidth: 2,
    emphasis: {
        color: 'red'
    },
    highlight: {
        color: 'orange'
    },
    downplay: {
        color: '#ccc'
    }
}
```

`highlightPolicy` 为 `'descendant'` 或 `'ancestor'` 的效果分别为：
https://echarts.baidu.com/gallery/view.html?c=doc-example/sunburst-highlight-descendant&edit=1&reset=1

总结

上面的教程主要讲述的是如何入门使用旭日图，感兴趣的用户可以在 配置项手册 查看更完整的文档。在灵活应用这些配置项之后，就能做出丰富多彩的旭日图了！
https://echarts.baidu.com/gallery/view.html?c=sunburst-book&edit=1&reset=1

服务端渲染

ECharts 可以在服务端进行渲染。例如 官方示例 里的一个个小截图，就是在服务端预生成出来的。

服务端渲染可以使用流行的 headless 环境，例如 puppeteer、headless chrome、node-canvas、jsdom、PhantomJS 等。

这是一些社区贡献的 echarts 服务端渲染方案：

- https://github.com/hellosean1025/node-echarts
- https://github.com/chfw/echarts-scrappeteer
- https://github.com/chfw/pyecharts-snapshot/blob/master/pyecharts_snapshot/phantomjs/snapshot.js

移动端自适应

ECharts 工作在用户指定高宽的 DOM 节点（容器）中。ECharts 的『组件』和『系列』都在这个 DOM 节点中，每个节点都可以由用户指定位置。图表库内部并不适宜实现 DOM 文档流布局，因此采用类似绝对布局的简单容易理解的布局方式。但是有时候容器尺寸极端时，这种方式并不能自动避免组件重叠的情况，尤其在移动端小屏的情况下。

另外，有时会出现一个图表需要同时在PC、移动端上展现的场景。这需要 ECharts 内部组件随着容器尺寸变化而变化的能力。

为了解决这个问题，ECharts 完善了组件的定位设置，并且实现了类似 CSS Media Query 的自适应能力。

ECharts组件的定位和布局

大部分『组件』和『系列』会遵循两种定位方式：

left/right/top/bottom/width/height 定位方式：

这六个量中，每个量都可以是『绝对值』或者『百分比』或者『位置描述』。

- 绝对值

  单位是浏览器像素（px），用 `number` 形式书写（不写单位）。例如 `{left: 23, height: 400}` 。

- 百分比

  表示占 DOM 容器高宽的百分之多少，用 `string` 形式书写。例如 `{right: '30%', bottom: '40%'}` 。

- 位置描述

  - 可以设置 `left: 'center'` ，表示水平居中。
  - 可以设置 `top: 'middle'` ，表示垂直居中。 这六个量的概念，和 CSS 中六个量的概念类似：

- left: 距离 DOM 容器左边界的距离。
- right: 距离 DOM 容器右边界的距离。
- top: 距离 DOM 容器上边界的距离。
- bottom: 距离 DOM 容器下边界的距离。
- width: 宽度。
- height: 高度。 在横向， `left` 、 `right` 、 `width` 三个量中，只需两个量有值即可，因为

任两个量可以决定组件的位置和大小，例如 `left` 和 `right` 或者 `right` 和 `width` 都可以决定组件的位置和大小。纵向，`top`、`bottom`、`height` 三个量，和横向类同不赘述。

`center` / `radius` 定位方式：

- `center`

  是一个数组，表示 `[x, y]`，其中，`x`、`y` 可以是『绝对值』或者『百分比』，含义和前述相同。

- `radius`

  是一个数组，表示 `[内半径, 外半径]`，其中，内外半径可以是『绝对值』或者『百分比』，含义和前述相同。

  在自适应容器大小时，百分比设置是很有用的。

横向（`horizontal`）和纵向（`vertical`）

ECharts的『外观狭长』型的组件（如 `legend`、`visualMap`、`dataZoom`、`timeline` 等），大多提供了『横向布局』『纵向布局』的选择。例如，在细长的移动端屏幕上，可能适合使用『纵向布局』；在PC宽屏上，可能适合使用『横向布局』。

横纵向布局的设置，一般在『组件』或者『系列』的 `orient` 或者 `layout` 配置项上，设置为 `'horizontal'` 或者 `'vertical'`。

于 `ECharts2` 的兼容：

ECharts2 中的 `x/x2/y/y2` 的命名方式仍被兼容，对应于 `left/right/top/bottom`。但是建议写 `left/right/top/bottom`。

位置描述中，为兼容 ECharts2，可以支持一些看起来略奇怪的设置：`left: 'right'`、`left: 'left'`、`top: 'bottom'`、`top: 'top'`。这些语句分别等效于：`right: 0`、`left: 0`、`bottom: 0`、`top: 0`，写成后者就不奇怪了。

Media Query

`Media Query` 提供了『随着容器尺寸改变而改变』的能力。

如下例子，可尝试拖动右下角的圆点，随着尺寸变化，legend 和 系列会自动改变布局位置和方式。
https://echarts.baidu.com/gallery/view.html?c=doc-example/pie-media&edit=1&reset=1

要在 option 中设置 Media Query 须遵循如下格式：

```
option = {
    baseOption: { // 这里是基本的『原子option』。
        title: {...},
        legend: {...},
        series: [{...}, {...}, ...],
        ...
    },
    media: [ // 这里定义了 media query 的逐条规则。
        {
            query: {...},   // 这里写规则。
            option: {       // 这里写此规则满足下的option。
                legend: {...},
                ...
            }
        },
        {
            query: {...},   // 第二个规则。
            option: {       // 第二个规则对应的option。
                legend: {...},
                ...
            }
        },
        {                   // 这条里没有写规则，表示『默认』，
            option: {       // 即所有规则都不满足时，采纳这个option。
                legend: {...},
                ...
            }
        }
    ]
};
```

上面的例子中，`baseOption` 、以及 `media` 每个 option 都是『原子 option』，即普通的含有各组件、系列定义的 option。而由『原子option』组合成的整个 option，我们称为『复合option』。`baseOption` 是必然被使用的，此外，满足了某个 `query` 条件时，对应的 option 会被使用 `chart.mergeOption()` 来 merge 进去。

query:

每个 `query` 类似于这样:

```
{
    minWidth: 200,
    maxHeight: 300,
    minAspectRatio: 1.3
}
```

现在支持三个属性： `width` 、 `height` 、 `aspectRatio` （长宽比）。每个属性都可以加上 `min` 或 `max` 前缀。比如， `minWidth: 200` 表示『大于等于200px宽度』。两个属性一起写表示『并且』，比如： `{minWidth: 200, maxHeight: 300}` 表示『大于等于200px宽度，并且小于等于300px高度』。

option:

`media` 中的 option 既然是『原子 option』，理论上可以写任何 option 的配置项。但是一般我们只写跟布局定位相关的，例如截取上面例子中的一部分 query option:

```
media: [
    ...,
    {
        query: {
            maxAspectRatio: 1           // 当长宽比小于1时。
        },
        option: {
            legend: {                   // legend 放在底部中间。
                right: 'center',
                bottom: 0,
                orient: 'horizontal'    // legend 横向布局。
            },
            series: [                   // 两个饼图左右布局。
                {
                    radius: [20, '50%'],
                    center: ['50%', '30%']
                },
                {
                    radius: [30, '50%'],
                    center: ['50%', '70%']
                }
            ]
        }
    },
    {
        query: {
```

```
            maxWidth: 500                // 当容器宽度小于 500 时。
        },
        option: {
            legend: {
                right: 10,               // legend 放置在右侧中间。
                top: '15%',
                orient: 'vertical'       // 纵向布局。
            },
            series: [                    // 两个饼图上下布局。
                {
                    radius: [20, '50%'],
                    center: ['50%', '30%']
                },
                {
                    radius: [30, '50%'],
                    center: ['50%', '75%']
                }
            ]
        }
    },
    ...
]
```

多个 **query** 被满足时的优先级：

注意，可以有多个 `query` 同时被满足，会都被 `mergeOption`，定义在后的后被 merge（即优先级更高）。

默认 **query**：

如果 `media` 中有某项不写 `query`，则表示『默认值』，即所有规则都不满足时，采纳这个 option。

容器大小实时变化时的注意事项：

在不少情况下，并不需要容器DOM节点任意随着拖拽变化大小，而是只是根据不同终端设置几个典型尺寸。

但是如果容器DOM节点需要能任意随着拖拽变化大小，那么目前使用时需要注意这件事：某个配置项，如果在某一个 `query option` 中出现，那么在其他 `query option` 中也必须出现，否则不能够回归到原来的状态。（ `left/right/top/bottom/width/height` 不受这个限制。）

『复合 **option**』 中的 `media` 不支持 **merge**

也就是说，当第二（或三、四、五 …）次 `chart.setOption(rawOption)` 时，如果 `rawOption` 是 `复合option`（即包含 `media` 列表），那么新的 `rawOption.media` 列表不会和老的 `media` 列表进行 merge，而是简单替代。当然， `rawOption.baseOption` 仍然会正常和老的 option 进行merge。

其实，很少有场景需要使用『复合 option』来多次 `setOption`，而我们推荐的做法是，使用 mediaQuery 时，第一次setOption使用『复合 option』，后面 `setOption` 时仅使用 『原子 option』，也就是仅仅用 setOption 来改变 `baseOption`。

最后看一个和时间轴结合的例子： https://echarts.baidu.com/gallery/view.html?c=doc-example/bar-media-timeline&edit=1&reset=1

自定义构建 **ECharts**

一般来说，可以直接从 echarts 下载页 中获取构建好的 echarts，也可以从 GitHub 中的 `echarts/dist` 文件夹中获取构建好的 echarts，这都可以直接在浏览器端项目中使用。这些构建好的 echarts 提供了下面这几种定制：

- 完全版：`echarts/dist/echarts.js`，体积最大，包含所有的图表和组件，所包含内容参见：`echarts/echarts.all.js`。
- 常用版：`echarts/dist/echarts.common.js`，体积适中，包含常见的图表和组件，所包含内容参见：`echarts/echarts.common.js`。
- 精简版：`echarts/dist/echarts.simple.js`，体积较小，仅包含最常用的图表和组件，所包含内容参见：`echarts/echarts.simple.js`。
  如果对文件体积有更严苛的要求，可以自己构建 echarts，能够仅仅包括自己所需要的图表和组件。自定义构建有几种方式：
- 在线自定义构建：比较方便。
- 使用 `echarts/build/build.js` 脚本自定义构建：比在线构建更灵活一点，并且支持多语言。
- 直接使用构建工具（如 rollup、webpack、browserify）自己构建：也是一种选择。
  下面我们举些小例子，介绍后两种方式。

准备工作：创建自己的工程和安装 echarts

使用命令行，创建自己的工程：

```
1. mkdir myProject
2. cd myProject
```

在 `myProject` 目录下使用命令行，初始化工程的 npm 环境并安装 echarts（这里前提是您已经安装了 npm）：

```
1. npm init
2. npm install echarts --save
```

通过 npm 安装的 echarts 会出现在 `myProject/node_modules` 目录下，从而可以直接在项目代码中得到 echarts，例如：

使用 ES Module：

```
1. import * as echarts from 'echarts';
```

使用 CommonJS：

```
var echarts = require('echarts')
```

下面仅以使用 ES Module 的方式来举例。

使用 echarts 提供的构建脚本自定义构建

在这个例子中，我们要创建一个饼图，并且想自定义构建一个只含有饼图的 echarts 文件，从而能使 echarts 文件的大小比较小一些。

echarts 已经提供了构建脚本 `echarts/build/build.js`，基于 Node.js 运行。我们可以在 `myProject` 目录下使用命令行，看到它的使用方式：

```
node node_modules/echarts/build/build.js --help
```

其中我们在这个例子里会用到的参数有：

- `-i`：代码入口文件，可以是绝对路径或者基于当前命令行路径的相对路径。
- `-o`：生成的 bundle 文件，可以是绝对路径或者基于当前命令行路径的相对路径。
- `–min`：是否压缩文件（默认不压缩），并且去多余的打印错误信息的代码，形成生产环境可用的文件。
- `–lang <language shortcut or file path>`：是否使用其他语言版本，默认是中文。例如：`–lang en` 表示使用英文，`–lang my/langXX.js` 表示构建时使用 `<cwd>/my/langXX.js` 替代 `echarts/lib/lang.js` 文件。
- `–sourcemap`：是否输出 source map，以便于调试。
- `–format`：输出的格式，可选 `'umb'`（默认）、`'amd'`、`'iife'`、`'cjs'`、`'es'`。既然我们想自定义构建一个只含有饼图的 echarts 文件，我们需要创建一个入口文件，可以命名为 `myProject/echarts.custom.js`，文件里会引用所需要的 echarts 模块：

```
// 引入 echarts 主模块。
export * from 'echarts/src/echarts';
// 引入饼图。
import 'echarts/src/chart/pie';
// 在这个场景下，可以引用 `echarts/src` 或者 `echarts/lib` 下的文件（但是不可混用），
// 参见下方的解释：“引用 `echarts/lib/**` 还是 `echarts/src/**`”。
```

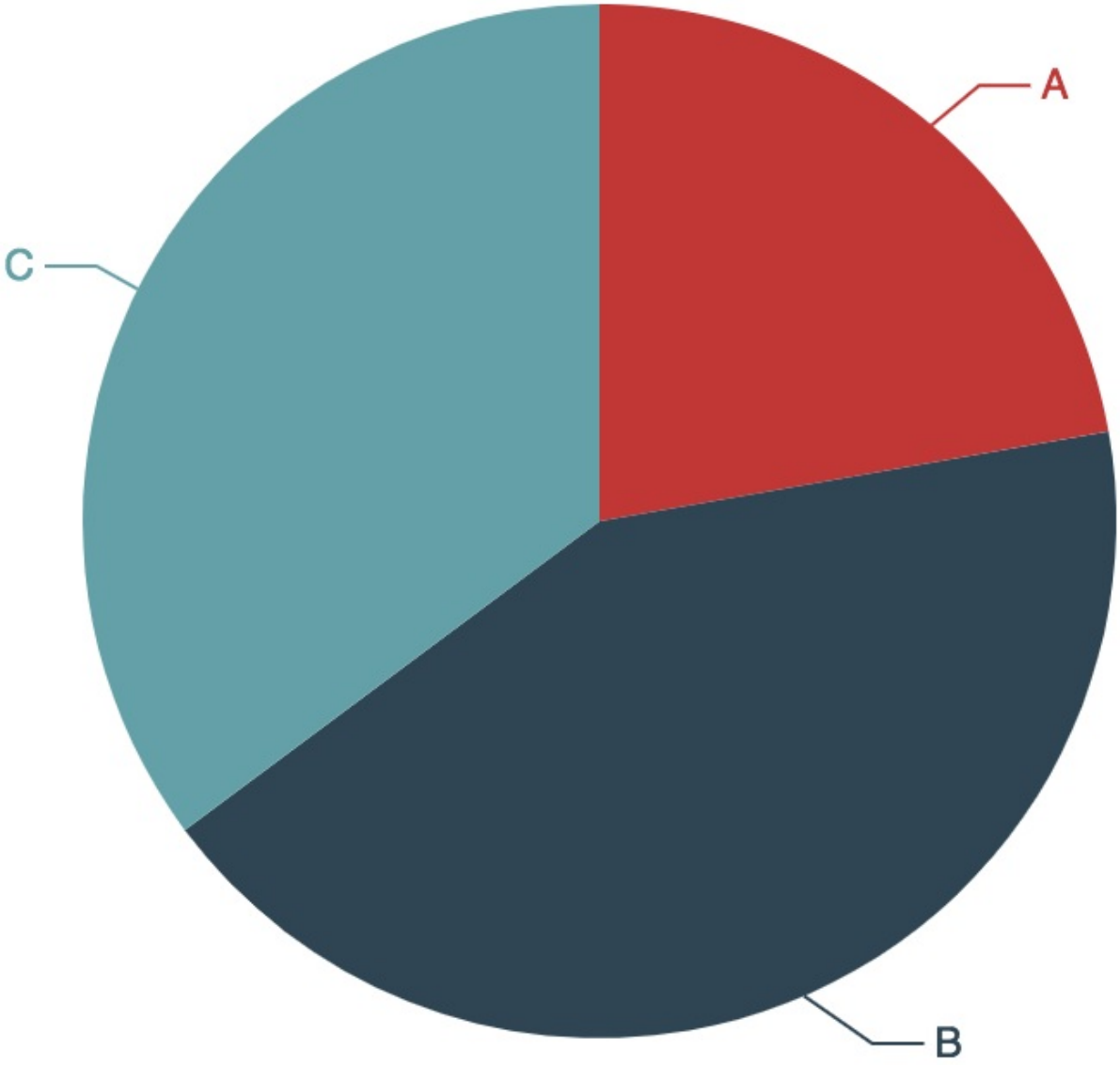
然后我们可以在 `myProject` 目录下使用命令行，这样开始构建：

```
node node_modules/echarts/build/build.js --min -i echarts.custom.js -o
lib/echarts.custom.min.js
```

这样，`myProject/lib/echarts.custom.min.js` 就生成了。我们可以创建 `myProject/pie.html` 来使用它：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>myProject</title>
    <!-- 引入 lib/echarts.custom.min.js -->
    <script src="lib/echarts.custom.min.js"></script>
</head>
<body>
    <div id="main" style="width: 600px;height:400px;"></div>
    <script>
        // 绘制图表。
        echarts.init(document.getElementById('main')).setOption({
            series: {
                type: 'pie',
                data: [
                    {name: 'A', value: 1212},
                    {name: 'B', value: 2323},
                    {name: 'C', value: 1919}
                ]
            }
        });
    </script>
</body>
</html>
```

然后在浏览器里打开 `myProject/pie.html`，就可以看到一个饼图：



允许被引用的模块

在自定义构建中，允许被引用的模块，全声明在 `myProject/node_module/echarts/echarts.all.js` 和 `myProject/node_module/echarts/src/export.js` 中。echarts 和 zrender 源代码中的其他模块，都是 echarts 的内部模块，不应该去引用。因为在后续 echarts 版本升级中，内部模块的接口和功能可能变化，甚至模块本身也可能被移除。

引用 echarts/lib/ 还是 **echarts/src/**

- 项目中如果直接引用 echarts 里的一些模块并自行构建，应该使用 `echarts/lib/` 路径，而不可使用 `echarts/src/` 。
- 当使用构建脚本 `echarts/build/build.js` 打包 bundle，那么两者可以选其一使用（不可混用），使用 `echarts/src/**` 可以获得稍微小一些的文件体积。

> 原因是：目前，`echarts/src/` 中是采用 **ES Module** 的源代码，**`echarts/lib/`** 中是 `echarts/src/` 编译成为 **CommonJS** 后的产物（编译成 **CommonJS** 是为了向后兼容一些不支持 **ES Module** 的老版本 **NodeJS** 和 **webpack**）。因为历史上，各个 **echarts** 扩展、各个用户项目，一直是使用的包路径是 **`echarts/lib/`** ，所以这个路径不应该改变，否则，可能导致混合使用 `echarts/src/` 和 **`echarts/lib/`** 得到两个不同的 echarts 名空间，造成问题。而使用 `echarts/build/build.js` 打包 bundle 时没有涉及这个问题，`echarts/src/**` 中的 ES Module 便于静态分析从而得到稍微小些的文件体积。

直接使用 rollup 自定义构建

上文中介绍了如何使用 echarts 提供的脚本 `echarts/build/build.js` 自定义构建。与此并列的另一种选择是，我们直接使用构建工具（如 rollup、webpack、browserify）自定义构建，并且把 echarts 代码和项目代码在构建成一体。在一些项目中可能需要这么做。下面我们仅仅介绍如何使用 rollup 来构建。webpack 和 browserify 与此类同，不赘述。

首先我们在 `myProject` 目录下使用 npm 安装 rollup:

```
npm install rollup --save-dev
npm install rollup-plugin-node-resolve --save-dev
npm install rollup-plugin-uglify --save-dev
```

接下来创建项目 JS 文件 `myProject/line.js` 来绘制图表，内容为:

```
// 引入 echarts 主模块。
import * as echarts from 'echarts/lib/echarts';
// 引入折线图。
import 'echarts/lib/chart/line';
// 引入提示框组件、标题组件、工具箱组件。
import 'echarts/lib/component/tooltip';
import 'echarts/lib/component/title';
import 'echarts/lib/component/toolbox';

// 基于准备好的dom，初始化 echarts 实例并绘制图表。
echarts.init(document.getElementById('main')).setOption({
    title: {text: 'Line Chart'},
    tooltip: {},
```

```
    toolbox: {
        feature: {
            dataView: {},
            saveAsImage: {
                pixelRatio: 2
            },
            restore: {}
        }
    },
    xAxis: {},
    yAxis: {},
    series: [{
        type: 'line',
        smooth: true,
        data: [[12, 5], [24, 20], [36, 36], [48, 10], [60, 10], [72, 20]]
    }]
});
```

对于不支持 ES Module 的浏览器而言，刚才创建的 `myProject/line.js` 还不能直接被网页引用并在浏览器中运行，需要进行构建。使用 rollup 构建前，需要创建它的配置文件 `myProject/rollup.config.js`，内容如下：

```
// 这个插件用于在 `node_module` 文件夹（即 npm 用于管理模块的文件夹）中寻找模块。比如，代码中有
// `import 'echarts/lib/chart/line';` 时，这个插件能够寻找到
// `node_module/echarts/lib/chart/line.js` 这个模块文件。
import nodeResolve from 'rollup-plugin-node-resolve';
// 用于压缩构建出的代码。
import uglify from 'rollup-plugin-uglify';
// 用于多语言支持（如果不需要可忽略此 plugin）。
// import ecLangPlugin from 'echarts/build/rollup-plugin-ec-lang';

export default {
    name: 'myProject',
    // 入口代码文件，就是刚才所创建的文件。
    input: './line.js',
    plugins: [
        nodeResolve(),
        // ecLangPlugin({lang: 'en'}),
        // 消除代码中的 __DEV__ 代码段，从而不在控制台打印错误提示信息。
        uglify()
    ],
```

```
    output: {
        // 以 UMD 格式输出，从而能在各种浏览器中加载使用。
        format: 'umd',
        // 输出 source map 便于调试。
        sourcemap: true,
        // 输出文件的路径。
        file: 'lib/line.min.js'
    }
};
```

然后在 `myProject` 目录下使用命令行，构建工程代码 `myProject/line.js`：

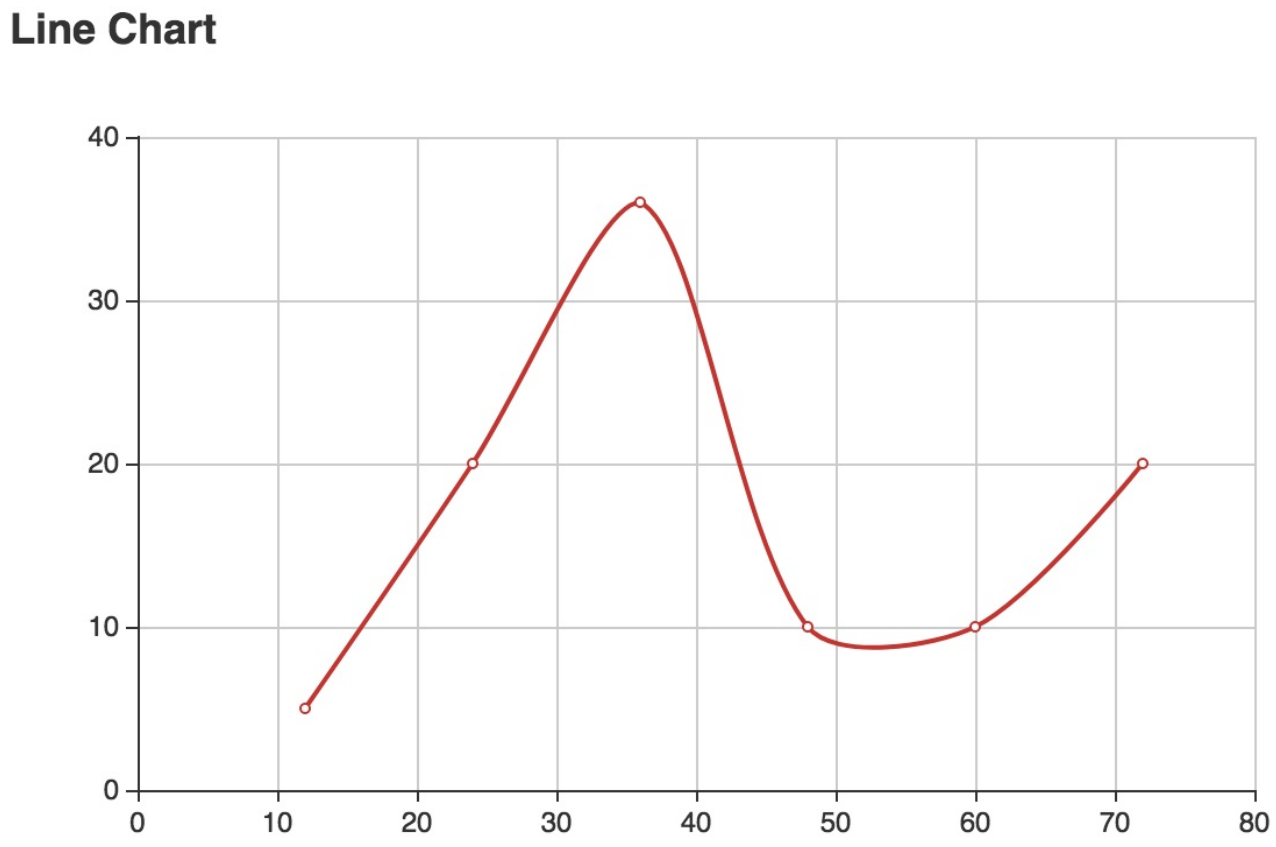
```
./node_modules/.bin/rollup -c
```

> 其中 `-c` 表示让 `rollup` 使用我们刚才创建的 `myProject/rollup.config.js` 文件作为配置文件。

构建生成的 `myProject/lib/line.min.js` 文件包括了工程代码和 echarts 代码，并且仅仅包括我们所需要的图和组件，并且可以在浏览器中使用。我们可以用一个示例页面来测试一下，创建文件 `myProject/line.html`，内容如下：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>myProject</title>
</head>
<body>
    <!-- 为 echarts 准备一个具备大小（宽高）的Dom。 -->
    <div id="main" style="width: 600px;height:400px;"></div>
    <!-- 引入刚才构建好的文件。 -->
    <script src="lib/line.min.js"></script>
</body>
</html>
```

在浏览器里打开 `myProject/line.html` 则会得到如下效果：



多语言支持

上面的例子中能看到，工具箱组件（toolbox）的提示文字是中文。本质上，echarts 图表显示出来的文字，都可以通过 `option` 来定制，改成任意语言。但是如果想“默认就是某种语言”，则需要通过构建来实现。

在上面的例子中，可以在 `echarts/build/build.js` 的参数中指定语言：

```
1. node node_modules/echarts/build/build.js --min -i echarts.custom.js -o
   lib/echarts.custom.min.js --lang en
```

表示使用内置的英文。此外还可以是 `–lang fi` 。

```
1. node node_modules/echarts/build/build.js --min -i echarts.custom.js -o
   lib/echarts.custom.min.js --lang my/langXX.js
```

表示在构建时使用 `myProject/my/langXX.js` 文件来替换 `myProject/node_modules/echarts/lib/lang.js` 文件。这样可以在 `myProject/my/langXX.js` 文件中自定义语言。注意这种方式中，必须指定 `-o` 或者 `–output` 。

另外，上面的 rollup 插件 `echarts/build/rollup-plugin-ec-lang` 也可以传入同样的参数，实现同样的功能。

自定义系列

自定义系列（custom series），是一种系列的类型。它把绘制图形元素这一步留给开发者去做，从而开发者能在坐标系中自由绘制出自己需要的图表。

echarts 为什么会要支持 自定义系列 呢？echarts 内置支持的图表类型是最常见的图表类型，但是图表类型是难于穷举的，有很多小众的需求 echarts 并不能内置的支持。那么就需要提供一种方式来让开发者自己扩展。另一方面，所提供的扩展方式要尽可能得简单，例如图形元素创建和释放、过渡动画、tooltip、数据区域缩放（dataZoom）、视觉映射（visualMap）等功能，尽量在 echarts 中内置得处理，使开发者不必纠结于这些细节。综上考虑形成了 自定义系列（custom series）。

例如，下面的例子使用 **custom series** 扩展出了 **x-range** 图：

更多的例子参见：**custom examples**

下面来介绍开发者怎么使用 自定义系列（custom series）。

（一）renderItem 方法

开发者自定义的图形元素渲染逻辑，是通过书写 renderItem 函数实现的，例如：

```
var option = {
    ...,
    series: [{
        type: 'custom',
        renderItem: function (params, api) {
            // ...
        },
        data: data
    }]
}
```

在渲染阶段，对于 series.data 中的每个数据项（为方便描述，这里称为 dataItem），会调用此 renderItem 函数。这个 renderItem 函数的职责，就是返回一个（或者一组）图形元素定义，图形元素定义 中包括图形元素的类型、位置、尺寸、样式等。echarts 会根据这些 图形元素定义 来渲染出图形元素。如下的示意：

```
var option = {
    ...,
```

```
    series: [{
        type: 'custom',
        renderItem: function (params, api) {
            // 对于 data 中的每个 dataItem，都会调用这个 renderItem 函数。
            // （但是注意，并不一定是按照 data 的顺序调用）

            // 这里进行一些处理，例如，坐标转换。
            // 这里使用 api.value(0) 取出当前 dataItem 中第一个维度的数值。
            var categoryIndex = api.value(0);
            // 这里使用 api.coord(...) 将数值在当前坐标系中转换成为屏幕上的点的像素值。
            var startPoint = api.coord([api.value(1), categoryIndex]);
            var endPoint = api.coord([api.value(2), categoryIndex]);
            // 这里使用 api.size(...) 获得 Y 轴上数值范围为 1 的一段所对应的像素长度。
            var height = api.size([0, 1])[1] * 0.6;

            // shape 属性描述了这个矩形的像素位置和大小。
            // 其中特殊得用到了 echarts.graphic.clipRectByRect，意思是，
            // 如果矩形超出了当前坐标系的包围盒，则剪裁这个矩形。
            // 如果矩形完全被剪掉，会返回 undefined.
            var rectShape = echarts.graphic.clipRectByRect({
                // 矩形的位置和大小。
                x: startPoint[0],
                y: startPoint[1] - height / 2,
                width: endPoint[0] - startPoint[0],
                height: height
            }, {
                // 当前坐标系的包围盒。
                x: params.coordSys.x,
                y: params.coordSys.y,
                width: params.coordSys.width,
                height: params.coordSys.height
            });

            // 这里返回为这个 dataItem 构建的图形元素定义。
            return rectShape && {
                // 表示这个图形元素是矩形。还可以是 'circle', 'sector', 'polygon' 等
等。
                type: 'rect',
                shape: rectShape,
                // 用 api.style(...) 得到默认的样式设置。这个样式设置包含了
                // option 中 itemStyle 的配置和视觉映射得到的颜色。
                style: api.style()
```

```
            };
        },
        data: [
            [12, 44, 55, 60], // 这是第一个 dataItem
            [53, 31, 21, 56], // 这是第二个 dataItem
            [71, 33, 10, 20], // 这是第三个 dataItem
            ...
        ]
    }]
}
```

renderItem 函数提供了两个参数：

- params：包含了当前数据信息（如 seriesIndex 、dataIndex 等等）和坐标系的信息（如坐标系包围盒的位置和尺寸）。
- api：是一些开发者可调用的方法集合（如 api.value() 、api.coord() ）。

renderItem 函数须返回根据此 dataItem 绘制出的图形元素的定义信息，参见 renderItem.return。

一般来说，renderItem 函数的主要逻辑，是将 dataItem 里的值映射到坐标系上的图形元素。这一般需要用到 renderItem.arguments.api 中的两个函数：

- api.value(…)，意思是取出 dataItem 中的数值。例如 api.value(0) 表示取出当前 dataItem 中第一个维度的数值。
- api.coord(…)，意思是进行坐标转换计算。例如 var point = api.coord([api.value(0), api.value(1)]) 表示 dataItem 中的数值转换成坐标系上的点。

有时候还需要用到 api.size(…) 函数，表示得到坐标系上一段数值范围对应的长度。

返回值中样式的设置可以使用 api.style(…) 函数，他能得到 series.itemStyle 中定义的样式信息，以及视觉映射的样式信息。也可以用这种方式覆盖这些样式信息：api.style({fill: 'green', stroke: 'yellow'}) 。

书写完 renderItem 方法后，自定义系列的 90% 工作就做完了。剩下的是一些精化工作。

（二）使坐标轴的范围自适应数据范围

在 直角坐标系（grid）、极坐标系（polar） 中都有坐标轴。坐标轴的刻度范围需要自适应当前显示出的数据的范围，否则绘制出的图形会超出去。所以，例如，在 直角坐标系（grid） 中，使用自定义系列的开发者，需要设定， data 中的哪些维度会对应到 x 轴上，哪些维度会对应到 y 轴上。这件事通过 encode 来设定。例如：

```
option = {
    series: [{
        type: 'custom',
        renderItem: function () {
            ...
        },
        encode: {
            // data 中『维度1』和『维度2』对应到 X 轴
            x: [1, 2],
            // data 中『维度0』对应到 Y 轴
            y: 0
        },
        data: [
            // 维度0  维度1  维度2  维度3
            [   12,   44,   55,   60   ], // 这是第一个 dataItem
            [   53,   31,   21,   56   ], // 这是第二个 dataItem
            [   71,   33,   10,   20   ], // 这是第三个 dataItem
            ...
        ]
    }]
};
```

（三）设定 tooltip

当然，使用 tooltip.formatter 可以任意定制 tooltip 中的内容。但是还有更简单的方法，通过encode 和 dimensions 来设定：

```
option = {
    series: [{
        type: 'custom',
        renderItem: function () {
            ...
        },
        encode: {
            x: [1, 2],
            y: 0,
            // 表示『维度2』和『维度3』要显示到 tooltip 中。
            tooltip: [2, 3]
        },
        // 表示给『维度2』和『维度3』分别取名为『年龄』和『满意度』，显示到 tooltip 中。
        dimensions: [null, null, '年龄', '满意度'],
```

```
        data: [
            // 维度0  维度1  维度2  维度3
            [   12,   44,   55,   60   ], // 这是第一个 dataItem
            [   53,   31,   21,   56   ], // 这是第二个 dataItem
            [   71,   33,   10,   20   ], // 这是第三个 dataItem
            ...
        ]
    }]
};
```

上面，一个简单的 custome series 例子完成了。

下面介绍几个其他细节要点。

（四）超出坐标系范围的截取

与 dataZoom 结合使用的时候，常常使用会设置 dataZoom.filterMode 为 ‘weakFilter’。这个设置的意思是：当 `dataItem` 部分超出坐标系边界的时候， `dataItem` 不会整体被过滤掉。例如：

```
option = {
    dataZoom: {
        xAxisIndex: 0,
        filterMode: 'weakFilter'
    },
    series: [{
        type: 'custom',
        renderItem: function () {
            ...
        },
        encode: {
            // data 中『维度1』和『维度2』对应到 X 轴
            x: [1, 2],
            y: 0
        },
        data: [
            // 维度0  维度1  维度2  维度3
            [   12,   44,   55,   60   ], // 这是第一个 dataItem
            [   53,   31,   21,   56   ], // 这是第二个 dataItem
            [   71,   33,   10,   20   ], // 这是第三个 dataItem
            ...
```

```
        ]
    }]
};
```

在这个例子中，『维度1』和『维度2』对应到 X 轴，`dataZoom` 组件控制 X 轴的缩放。假如在缩放的过程中，某个 dataItem 的『维度1』超出了 X 轴的范围，『维度2』还在 X 轴的范围中，那么只要设置 `dataZoom.filterMode = 'weakFilter'`，这个 dataItem 就不会被过滤掉，从而还能够使用 `renderItem` 绘制图形（可以使用上面提到过的 `echarts.graphic.clipRectByRect` 把图形绘制成被坐标系剪裁过的样子）。参见上面提到过的例子：Profile

（五）关于 dataIndex

开发者如果使用到的话应注意，renderItem.arguments.params 中的 `dataIndex` 和 `dataIndexInside` 是有区别的：

- `dataIndex` 指的 `dataItem` 在原始数据中的 index。
- `dataIndexInside` 指的是 `dataItem` 在当前数据窗口（参见 dataZoom）中的 index。

renderItem.arguments.api 中使用的参数都是 `dataIndexInside` 而非 `dataIndex`，因为从 `dataIndex` 转换成 `dataIndexInside` 需要时间开销。

（六）事件监听

```
chart.setOption({
    // ...
    series: {
        type: 'custom',
        renderItem: function () {
            // ...
            return {
                type: 'group',
                children: [{
                    type: 'circle'
                    // ...
                }, {
                    type: 'circle',
                    name: 'aaa',
                    // 用户指定的信息，可以在 event handler 访问到。
                    info: 12345,
                    // ...
                }]
```

```
            };
        }
    }
});
chart.on('click', {element: 'aaa'}, function (params) {
    // 当 name 为 'aaa' 的图形元素被点击时，此回调被触发。
    console.log(params.info);
});
```

（七）自定义矢量图形

自定义系列能支持使用 SVG PathData 定义矢量路径。从而可以使用矢量图工具中做出的图形。参见：path，以及例子：icons 和 shapes。

更多的自定义系列的例子参见：**custom examples**