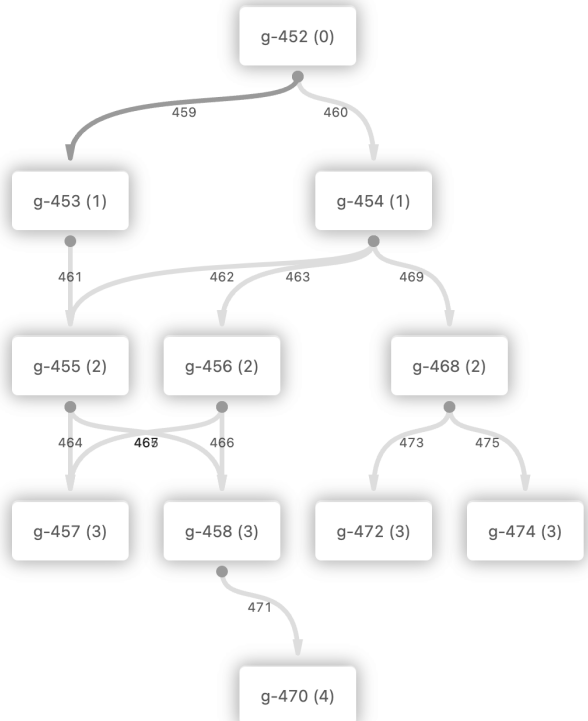
图编辑器的一个简单实现

bifnudozhao@tencent.com

概要

- 需求
- 目标
- 功能集合
- 技术实现
  - 节点渲染
  - 布局算法
  - 边渲染
- 渲染问题
- 节点与边的选择
- 展示

需求



节点

节点作为图的基本元素之一，代表“单独的实体”，可以独立存在。节点并不是指“一个点”，它可以承载任何具备上述描述的东西，比如信息，比如函数，比如功能等。

边

边也是图的基本元素，它与节点的最大区别在于，它不能单独存在，它只能作为两点之间的连接，图没有“射线”的概念，所以边必须有起点以及终点。边并不是指“一条线”，而是一种抽象关系，两者之间的关系。另外边可以有向，可以无向。在有向的概念中，边可以表达一种偏序关系，在无向的概念中，边可以表达一种连接关系。

目标

1. 提供底层的图算法以及渲染能力，暴露节点以及边的定义接口。不考虑过多业务层面的限制。
2. 提供一个基础的图渲染组件。

操作功能

图编辑器只需要负责图相关的操作，图作为一个基本数据结构概念来说非常简单，只有两个元素：节点和边。由此定义其功能如下：

- 创建节点：往图中加入一个孤立的节点
- 选择节点：选择图中的某一个节点
- 删除节点：删除图中的一个给定节点，并且递归删除其相关的无用边以及节点
- 创建边：创建一条连接给定两个节点的边
- 选择边：选择途中的某一条边
- 删除边：删除一条边，并且递归删除其相关的无用边以及节点

展示功能

目前画图相关的库也好，工具也好，实在太多。可以划分为一下类别。下面也顺带给出了各种类型的一些优缺点。

是否可以拖动

- **可拖动**：网格基础，自动对齐，随意拖动
  1. **优点**：自由，随心所欲
  2. **缺点**：对齐麻烦，逼死强迫症
- **不可拖动**：固定位置，自动排版
  1. **优点**：方便，操作快捷，完全没必要考虑排版
  2. **缺点**：遇到极端情况，难以通过自行排版的方式解决

节点展示形式

- **不可配置**：只能从预设类型中选择节点
  1. **优点**：不需要动脑，给什么拖什么
  2. **缺点**：需要自定义节点类型的时候，没有办法自己新增了
- **可配置**：可以自定义节点类型（也就是可以自定义节点所承载的信息形式和类型）
  1. **优点**：非常自由
  2. **缺点**：暂时没想到缺点

世事两难全，结合难度以及使用场景，目前的基础版本实现上述的部分方面

1. **不可拖动**：加强使用体验，自动排版，让用户操作更为顺畅，也方便结合预订规则进行排版
2. **可配置**：流程的每个节点需要进行各种配置

技术实现 - 整体考虑

目前各种库涉及到各种技术实现方式，但比较强大的库基本使用 canvas 实现，canvas 的好处在于灵活自由，但是对实体操作方便比较麻烦，即使加上 sprite 的概念，要在 canvas 里实现还是非常没有必要的。

结合上述的功能定义，采取下述的技术方式

1. 纯 js 图算法的实现
2. 视图层用 React/Vue 组件包裹
3. 使用 DOM（节点） 以及 SVG（线） 实现渲染，从而可以方便各种自定义的复杂节点渲染

技术实现 - 渲染算法

渲染图需要渲染节点以及边，节点在二维平面上，通过一个坐标就能确定位置，但是边则需要两个坐标。从这个角度来看，节点和边肯定是分开两个步骤进行渲染的。

图渲染的算法主要难点在于布局，从布局方式进行考虑的话，会直接影响算法实现的不同。如果采用绝对定位，需要设计出能计算出所有节点以及边的位置的算法，而位置本身也会受到节点和边的尺寸影响。但转念一想，如果采用 flex（相对） 布局，或许能让浏览器帮忙做一些麻烦的事情。

技术细节 - 节点渲染

遍历图可以通过递归和非递归的方式，从 React 组件的角度来考虑，递归算法可以让算法更为优雅。通过递归描述图算法如下。

```
const renderNodes = (node) => {
    renderNode(node)
    node.children.forEach(renderNode)
}
renderNodes(root)
```

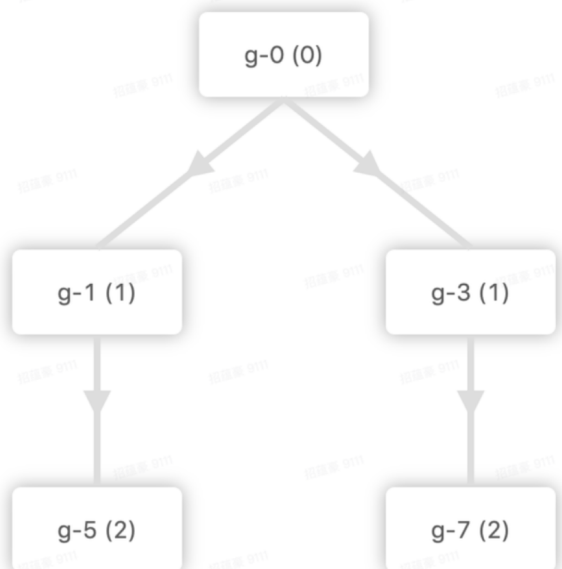
考虑到节点可能有多个父节点，所以上述递归算法会重复渲染同样的节点，这个只需要加一个渲染记录来避免重复渲染即可。

```
const renderedNodes = {}
const renderNodes = (node) => {
    if (renderedNodes[node.id]) return
    renderedNodes[ndoe.id] = true
    renderNode(node)
    node.children.forEach(renderNode)
}
renderNodes(root)
```

技术细节 - 布局算法

看上去很简单，但是如何布局才能让节点正常分布呢?

先从一个正常的图来观察特点。



从上图可以总结出两个特点:

1. 子节点要渲染在父节点下面
2. 子节点是水平排布的

所以上述递归算法里面的两个步骤，可以将结果渲染在两个正常顺序排布的 div 里

```
const renderNode = (node: Node) => {
  return (
    // 渲染在一个节点 div 里，居中放节点
    <div class="flex-col">
      <div>
        {render(node)}
      </div>
      <div class="flex-row">
        {node.children.forEach(child => {
          // 渲染在一个子节点 div 里，每个子节点水平排布
          return <div>{renderNode(child)}</div>
        })}
      </div>
    </div>
  )
}
```

技术细节 - 边渲染

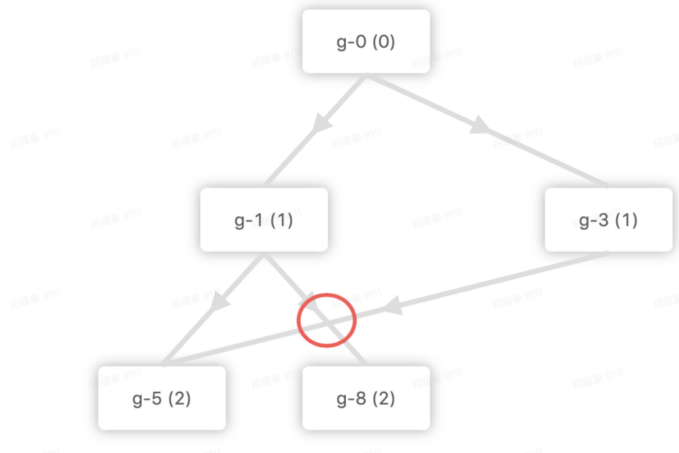
边渲染比节点渲染简单很多，因为节点已经渲染出来了，每个节点的位置都可以确切知道，直接获取节点的坐标位置，然后结合图的边信息即可完成渲染。伪代码如下。

```
const renderEdges = (edges) => {
    const positions = getNodesPositions()
    edges.forEach(edge => {
        const startPos = positions[edge.start]
        const endPos = positions[edge.end]
        renderEdge(startPos, endPos)
    })
}
renderEdges(graph.edges)
```

技术细节 - 渲染问题

如果单纯采用上述算法，会引发一些问题。

没必要的线交叉问题



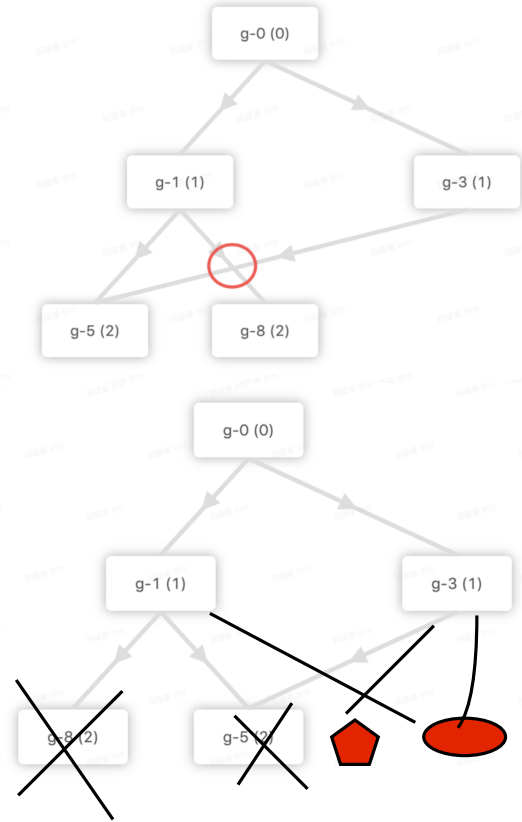
上图重点`g-8`实际上是最后添加的一个节点，节点由于是一个数组，后添加的节点实际上会排在后面，而渲染从左往右渲染，所以就引发了上面的交叉问题了。

技术细节 - 渲染问题

深入地想一层，交叉的本质在于，`g-8`的兄弟`g-5`被更多的父节点依赖了，而一个图，直观上来看，要是入度大的节点在中间，那么它们将“更容易”被父节点用更短的距离引用到，基于这个考虑，做一个简单的优化就能解决上面的问题。就是在渲染子节点数组之前，先对其进行入度的排序。

```
1   node.children = _.sortBy(node.children, 'inDegree')
```

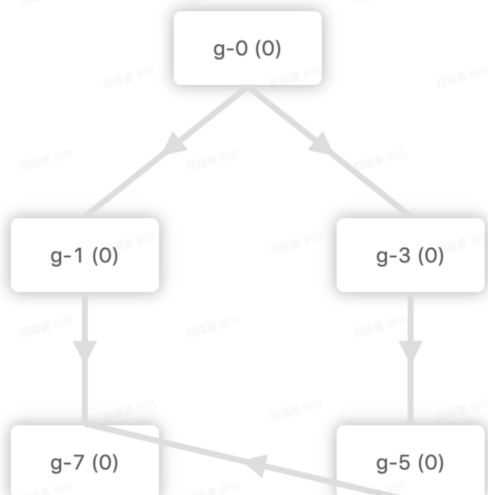
由于只会渲染一次，那么这些节点会被他们的父节点在第一次遇到的时候渲染，所以，依赖越大的就会排到“右边”，而他们的另外的父节点不会对他们的位置进行影响。结果如下



技术细节 - 渲染问题

节点所在层级的影响

由于节点有多个父节点，并且由第一个父节点渲染，这就导致下面的效果。

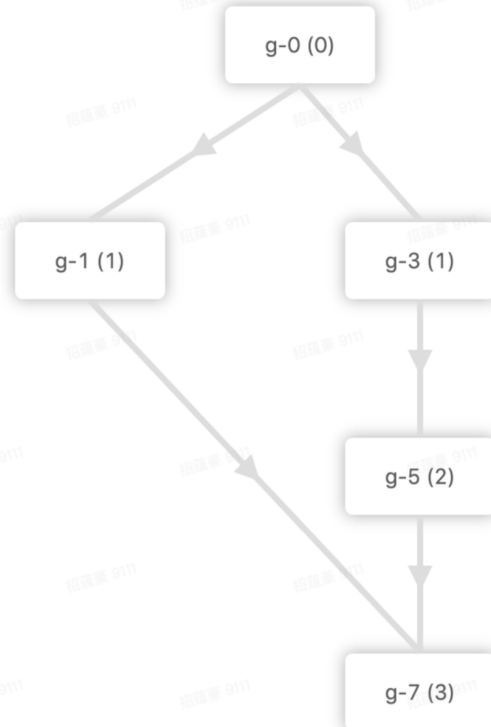


可以看到奇怪的地方在于`g-5`和`g-7`的连线，因为`g-7`在`g-5`下面，但是它被父节点`g-1`先渲染了，导致渲染到`g-5`的时候只能连线过去。

技术细节 - 渲染问题

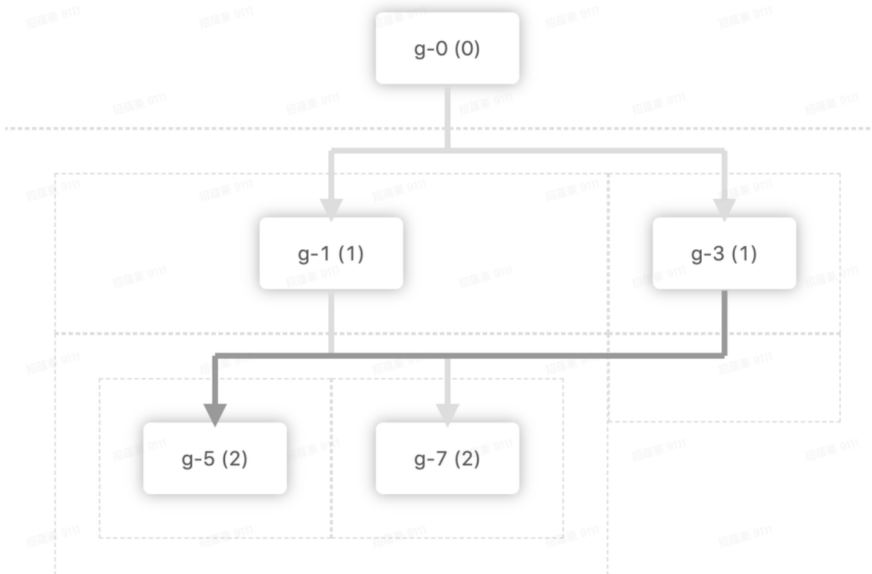
根本原因在于节点的高度问题，只要保证节点渲染在“最大路径”的层级，就不会有上述问题，添加一个方法在图每次变动的时候更新每个节点所在的最大层级 `maxDepth`，并在渲染的时候判断，当前的节点是否在最大层级上，如果是就渲染，不是则不渲染。对算法修改如下。

```
const renderedNodes = {}
const renderNodes = (node, level) => {
    if (renderedNodes[node.id]) return
    if (level < node.maxDepth) return
    renderedNodes[ndoe.id] = true
    renderNode(node)
    node.children.forEach(renderNode, level + 1)
}
renderNodes(root, 0)
```



节点 / 边 的选择

节点的选择比较简单，因为节点都是普通 DOM，并且相互之间都在各自的 div 里，不会有任何交叉，所以直接 hover 就可以选择所有节点。



边的选择不容易，因为边渲染是 svg 节点里的 line 节点，从“常规角度来想”，如果 svg 有长宽，那么一条斜线包裹在一个矩形里，会导致很多空白区域被 svg 所覆盖。

节点 / 边 的选择

使用一个微小的 svg 承载整条边

所以这里需要做一个比较取巧的样式，让外层的 svg 尺寸为 1 x 1（如果是 0 或者不显示的话，整个 svg 节点都不会在渲染树里的）。然后 overflow 为 hidden，这样就可以让 svg 作为一个辅助定位的元素，里面的线，才是这正的可选择内容。

另外还有一个需要解决的地方，那就是边是会交叉的，如果采用正交方式来渲染边，那需要对当前选中的边进行强调（可以看上面的区别）。所以在 hover 的时候（注意是对实际线条的 hover，而不是定位作用的 svg），需要增加其 z-index。具体代码看下图。

```scss
svg.line {
    position: absolute;
    overflow: hidden;
    width: 1px;
    height: 1px;
    z-index: 0;
    & line {
        position: absolute;
        top: 100px; // 线相对于 svg 节点的位移
        left: 100px; // 线相对于 svg 节点的位移
        &:hover {
            z-index: 1;
        }
    }
}
```

图相关算法接口

图相关算法在这里不再详述，毕竟很多地方都有。算法与渲染的分离方式如下。

```
interface Graph {
    nodes: Node[];
    edges: Edge[];

    constructGraph(g: any); // 用户自定义图结构对齐
    addNode(node: Node);
    removeNode(node: Node);
    addEdge(edge: Edge);
    removeEdge(edge: Edge);
}
```

图编辑器接口

算法主要实现上述基础功能，与上面的需求定义对齐。

```
interface GraphEditorProps {
    // 基础操作方法
    onAddNode(node: Node): Node;
    onRemoveNode(node: Node): Node;
    onAddEdge(edge: Edge): Edge;
    onRemoveEdge(edge: Edge): Edge;

    // 操作相关的各种钩子
    beforeAddNode(graph: Graph, node: Node);
    afterAddNode(graphBefore: Graph, graphAfter: Graph, node: Node);
    // ...
}
```

组件暴露基础算法的功能，以及各种操作相关的钩子。

DEMO

Q & A