分析工具-自定义分析树-设计实践

做一个有趣的树?

- 关键三要素
  - 根指标
  - 子节点
  - 下钻条件

最初的探索

其实市面上流程图的图表库还是很多的，**最初我们想法也是基于图表库进行分析树的设计**，而且指南针之前的一些项目有基于图表库做过一些流程图，树图的经验的，因此我们调研了实现分析树的几种形式...

这是我们开发之前的UI稿~：

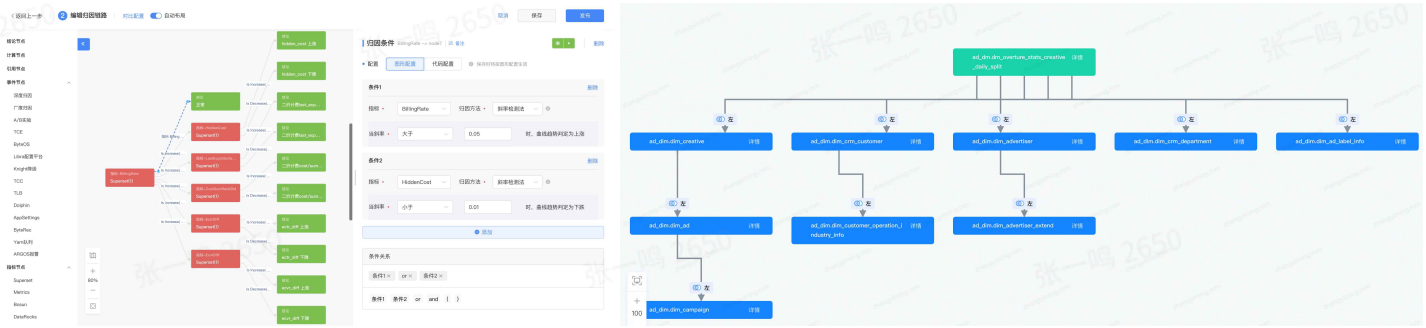
UI的需求实际上可以分为几个部分：

1. 每个节点其实都是自适应高度的，当修改节点的自适应高度后实际上整个树图的布局都要再次自适应布局
2. 节点强烈的和UI组件交互的需求，每个节点的click，hover都是有交互事件
3. 类似泳道概念：每一排节点上实际都是有一个类似泳道的交互
4. 框选：来自同一个父节点的子节点需要具备框选的能力

方案	优势	劣势
mxgraph	1. 老牌流程图组件库； 2. 很方便地实现流程图编辑等操作；	1. 使用起来非常复杂，可扩展性很差； 2. 性能一般，遇到较多的节点会有卡顿的情况； 3. 自动布局算法较差，无法实现设计稿上的自动布局； 4. 很难实现泳道和已选区域的绘制；
g6	1. 蚂蚁金服出品前端组件库； 2. 基于Canvas绘制，性能较好；	1. 使用成本高，可扩展性较差； 2. 还原设计稿需要较大时间和精力；
Canvas + ZRender	1. 通过Canvas绘制，理论上能绘制任何图形；	1. 实现起来较为复杂； 2. 依赖自动布局算法； 3. 无法使用复杂的BUI组件；
SVG + Table	1. 实现方式较为简单，性能较好；	1. 并不采用通用流程图组件思想实现，针对图的可扩展性较差(比如任意拖拽节点，自

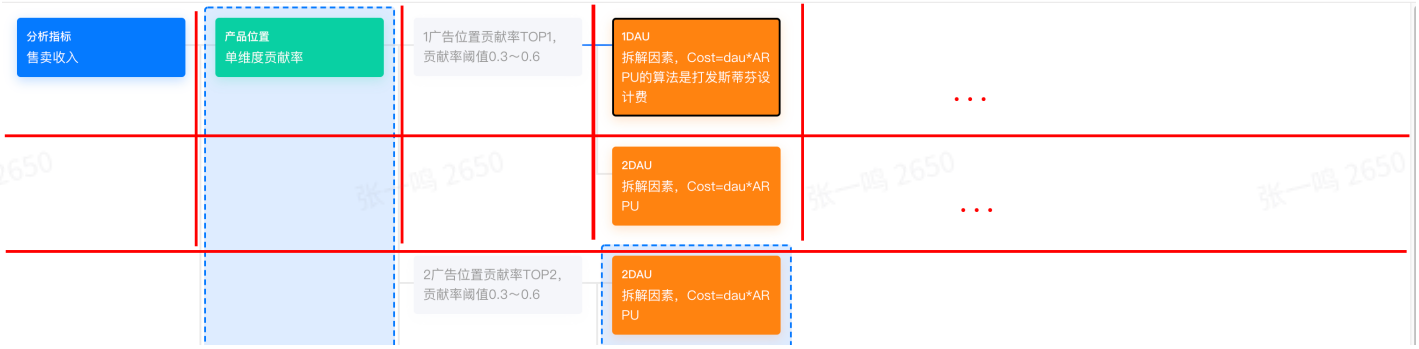
	2. 能够使用table做高度自适应的自动布局； 3. 方便绘制泳道和已选区域 4. 能够使用复杂的BUI组件 5. 响应式	动定位等)

Mxgraph&G6在面对这个UI稿的几个难点就明显有点水土不服了...



那没有轮子怎么办呢，bingo，我们只能自己造轮子了

- Canvas + ZRender
- SVG + Table
  - 把节点放到表格中，这并不是一个普通流程图的思想，但是这恰巧满足了我们的UI需求，因为我们都知道**直接绘制DOM是easy的！直接使用组件是清晰的！**



架构设计

5. 数据结构设计

如何设计数据结构?

A 图数据结构：

JavaScript

```
const data = {
  // 点集
  nodes: [
    {
      id: 'node1',
      x: 100,
      y: 200,
    },
    {
      id: 'node2',
      x: 300,
      y: 200,
    },
  ],
  // 边集
  edges: [
    // 表示一条从 node1 节点连接到 node2 节点的边
    {
      source: 'node1',
      target: 'node2',
    },
  ],
};
```

B 树数据结构：

JavaScript

```
const data = {
  id: 'node1',
  x: 100,
  y: 200,
  children: [{
      id: 'node2',
      x: 300,
      y: 200,
      parent: 'node1',
  }]
};
```

数据结构类型	A	B
优点	1. 可以表示有环图 2. 边上可以定义数据	1. 获取children和parent比较简单 2. 不用手动维护节点和边的关系
缺点	1. 获取children和parent比较复杂 2. 要维护节点和边的关系	1. 不能表示有环图 2. 边上无法定义数据

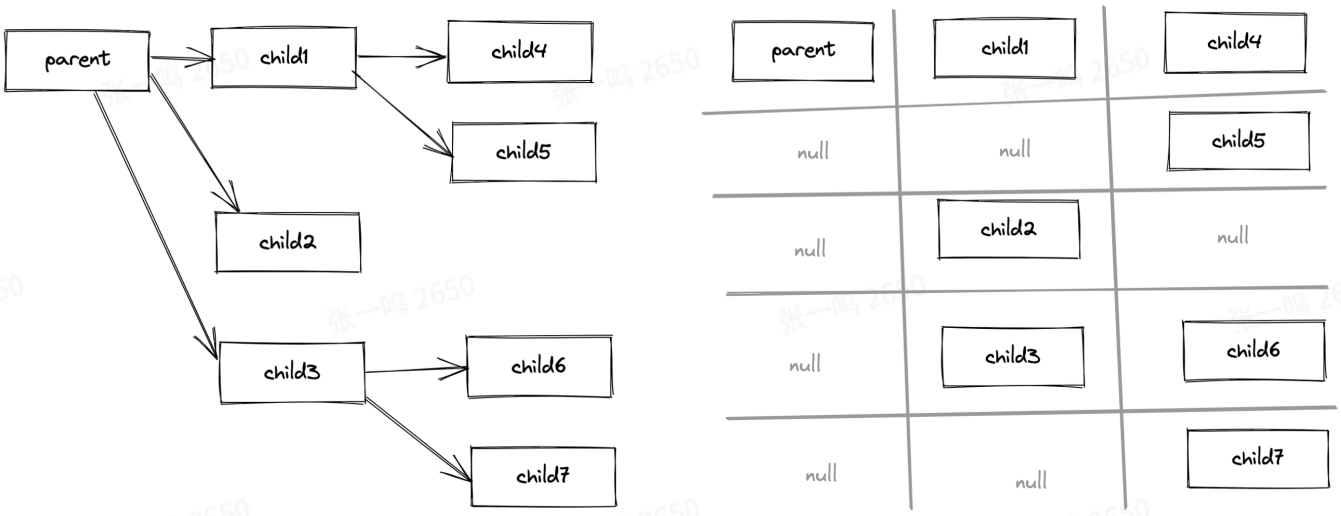
对于我们分析树所需要的场景，B显然是一种更好的方案

树的绘制

6. 自动布局

a. 使用递归函数，计算每个节点在表格中的位置(x, y)，然后再把这个树打平，构建一个treeMap。再根据map构建一个tableArray数组，填入表格中，就实现了分析树的自动布局；

b. 监听treeData等数据的变化，获取每个节点的offsetLeft和offsetTop等参数，使用svg绘制边和平行维度等；



https://excalidraw.com/

JavaScript

```
getTreeMap() {
  const edge = [];
  const treeMap = {};
  let deepY = 0;
```

```
  function travel(node, x, y) {
    node.x = x;
    node.y = y;
    deepY = max([y, deepY]);
    treeMap[node.nodeId] = node;
    if (!node.ref && node.children) {
      node.children.forEach((item, index) => {
        let nextY = y + index;
        if (deepY > y) {
          nextY = deepY + 1;
        }
        edge.push({
          source: node,
          target: item,
        })
        travel(item, x + 1, nextY)
      })
    }
  }
  travel(this.treeData, 0, 1);
  this.edge = edge;
  return treeMap;
},
getTableArray() {
  const tableArray = Object.values(this.treeMap);
  let maxX = 0;
  let maxY = 0;

  tableArray.forEach((item) => {
    if (item.x > maxX) {
      maxX = item.x;
    }
    if (item.y > maxY) {
      maxY = item.y;
    }
  });
  const arrayTable = new Array(maxY + 1).fill(null).map(() => new Array(maxX + 1).fill(null));
  tableArray.forEach((item) => {
    arrayTable[item.y][item.x] = item;
  });
  return arrayTable;
},
```

HTML

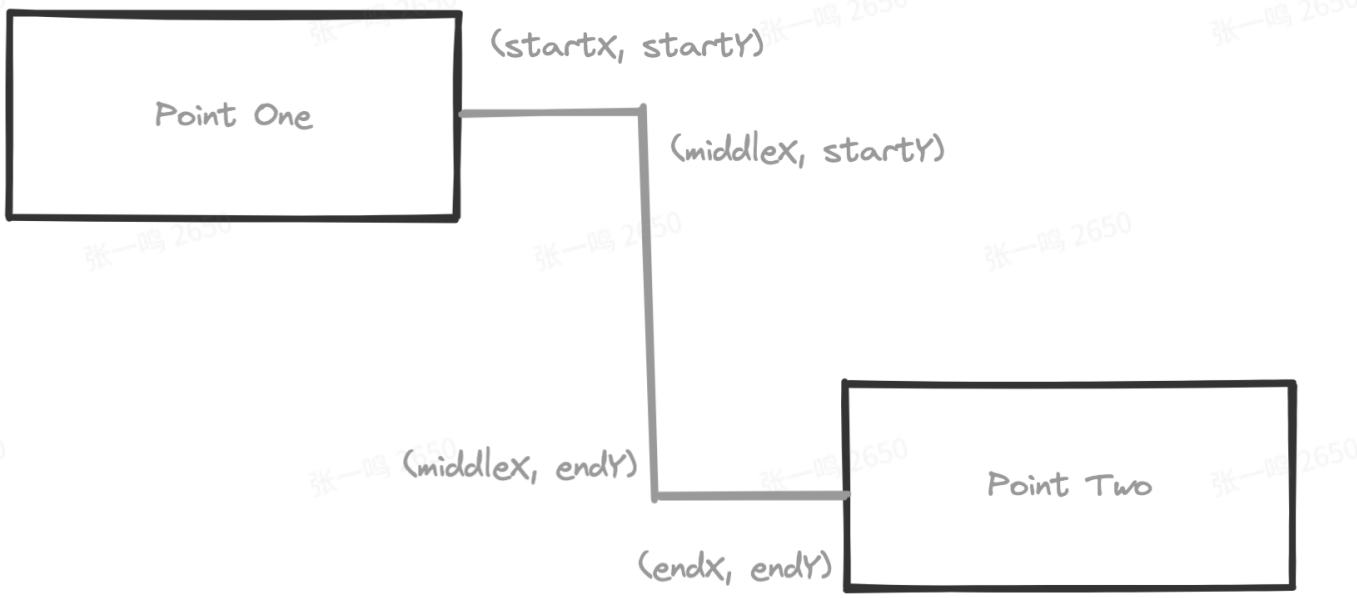
```
<template>
  <div class="tree-graph-wrapper">
    <div class="tree-graph">
      <div :class="graphClass" :style="graphStyle">
        <svg ref="svg" class="svg-layer"></svg>
        <div class="bulk-layer">
          <graph-bulk
            v-for="(node, key) in bulkData"
            :id="node.nodeId"
            :isBulk="isBulk"
            :key="generateBulkKey(bulkGeometrys[node.nodeId], key)"
            :node="node"
            :geometry="bulkGeometrys[node.nodeId]"
            :bulkData="bulkData"
            :treeMap="treeMap"
            :isRead="isRead"
            :ref="node.nodeId"
            @select="handleCardClick"
          />
        </div>
        <bulk-choose-tips :selectNode="selectNode" :isBulk="isBulk"/>
        <tr v-for="(x, xindex) in tableArray" :key="generateTrKey(x)">
          <td v-for="(node, yindex) in x" :key="yindex"
:style="selectBulkColumnStyle(yindex)">
            <div v-if="xindex === 0" :style="mockCardStyle"/>
            <graph-card v-else-if="node"
              :id="node.nodeId"
              :node="node"
              :selectNode.sync="_selectNode"
              :bulkData="bulkData"
              :treeMap="treeMap"
              :ref="node.nodeId"
              :isBulk="isBulk"
              :isRead="isRead"
              @click.native="handleCardClick(node)"
            />
          </td>
        </tr>
      </div>
    </div>
    <graph-tool v-if="!isNavigation" v-model="scale" v-bind="$props"/>
  </div>
</template>
```

```
42  </template>
```

7. 线的绘制

节点的步骤完成后，线的绘制其实也很明朗：

- 遍历节点之间的关系（source&target）
- 然后需要计算坐标，然后再用svg画出来就可以了~



Kotlin

```
drawLines() {
  if (this.isDrawLine) {
    return;
  }
  this.isDrawLine = true;
  this.$nextTick(() => {
    this.draw.clear();
    this.polylines.clear();
    this.edge.forEach((edge) => {
      const {source, target} = edge;
      const sourceDom = this.$refs?.[source.nodeId]?.[0].$el;
      const targetDom = this.$refs?.[target.nodeId]?.[0].$el;
      if (!sourceDom || !targetDom) {
        // eslint-disable-next-line no-console
        console.warn('不存在dom节点');
        return;
      }
      const {offsetLeft: x1, offsetTop: y1, offsetWidth: width1} = sourceDom;
      const {offsetLeft: x2, offsetTop: y2} = targetDom;
      const startx = x1 + width1;
      let starty = y1 + defaultHeight / 2;
      let endx = x2;
      let endy = y2 + defaultHeight / 2;
      const middlex = (startx + endx) / 2;
      const points = [startx, starty, middlex, starty, middlex, endy, endx,
endy];
      const polyline = this.draw.polyline(points);

      this.polylines.set(edge, polyline);
    });
    this.highlightLine();
    this.isDrawLine = false;
  })
},
```

8. 平行维度节点（框选节点）

a. 计算平行维度节点包含节点的边界，再绘制出矩形框；

b. 平行维度选择时泳道绘制：使用css td:nth-child(even)


彭晨引流电商分析树-1t1321231
编辑
平行维度挑选
预览
保存
拆解因素占比 ≥ 0.2
归因
引流电商-是否为新客户(ID: 17)
单维度贡献率
单维度值贡献率 ≥ 0.384，top 3
归因
客户类型(ID: 19)
单维度贡献率
计划标的二级分类(ID: 20)
单维度贡献率
引流电商-客户引流消耗区间(ID: 18)
单维度贡献率
平行维度挑选(ID：21)
编辑
已选节点
客户类型(ID：19)、计划标的二级分类(ID：20)、引流电商-客户引流消耗区间(ID：18)
挑选类型
贡献率
其他属性
贡献率TOP1，阈值 ≥ 0.4
拆解因素占比 ≥ 0
本地广告消耗(ID: 27)
指标拆解
镜像节点
引流电商客户ARPU(ID: 6)
指标拆解


彭晨引流电商分析树-1t1321231
编辑
平行维度挑选
预览
保存
拆解因素占比 ≥ 0.2
归因
引流电商-是否为新客户(ID: 17)
单维度贡献率
单维度值贡献率 ≥ 0.384，top 3
归因
客户类型(ID: 19)
单维度贡献率
计划标的二级分类(ID: 20)
单维度贡献率
引流电商-客户引流消耗区间(ID: 18)
单维度贡献率
平行维度挑选(ID：21)
已选节点
客户类型(ID：19)、计划标的二级分类(ID：20)、引流电商-客户引流消耗区间(ID：18)
挑选类型
贡献率
其他属性
贡献率TOP1，阈值 ≥ 0.4
拆解因素占比 ≥ 0
本地广告消耗(ID: 27)
指标拆解
镜像节点
引流电商客户ARPU(ID: 6)
指标拆解

JavaScript

```
calcBulkGeometrys() {
  const bulkGeometrys = this.bulkData.reduce((obj, bulk) => {
    const box = bulk.nodeIds.reduce((obj, item) => {
      if (this.$refs?.[item]?.[0]) {
        const dom = this.$refs[item][0].$el;
        const {offsetLeft: x, offsetTop: y, offsetWidth: width, offsetHeight: height} = dom;
        return {
          x: [...obj.x, x, x + width],
          y: [...obj.y, y, y + height],
        };
      } else {
        return obj;
      }
    }, {x: [], y: []});
    let maxX = max(box.x) + bulkPadding;
    let minX = min(box.x) - bulkPadding;
    let maxY = max(box.y) + bulkPadding;
    let minY = min(box.y) - bulkPadding;
    return {
      ...obj,
      [bulk.nodeId]: {minX, minY, maxX, maxY},
    };
  }, {});
  this.bulkGeometrys = bulkGeometrys;
},
```

SCSS

```
  td:nth-child(1), td:nth-child(even) {
    background:#F4F4F4;
    background-clip: content-box;
  }
```

9. 其他交互细节

图的缩放

a. 使用 transform: scale css属性来实现图的缩放；

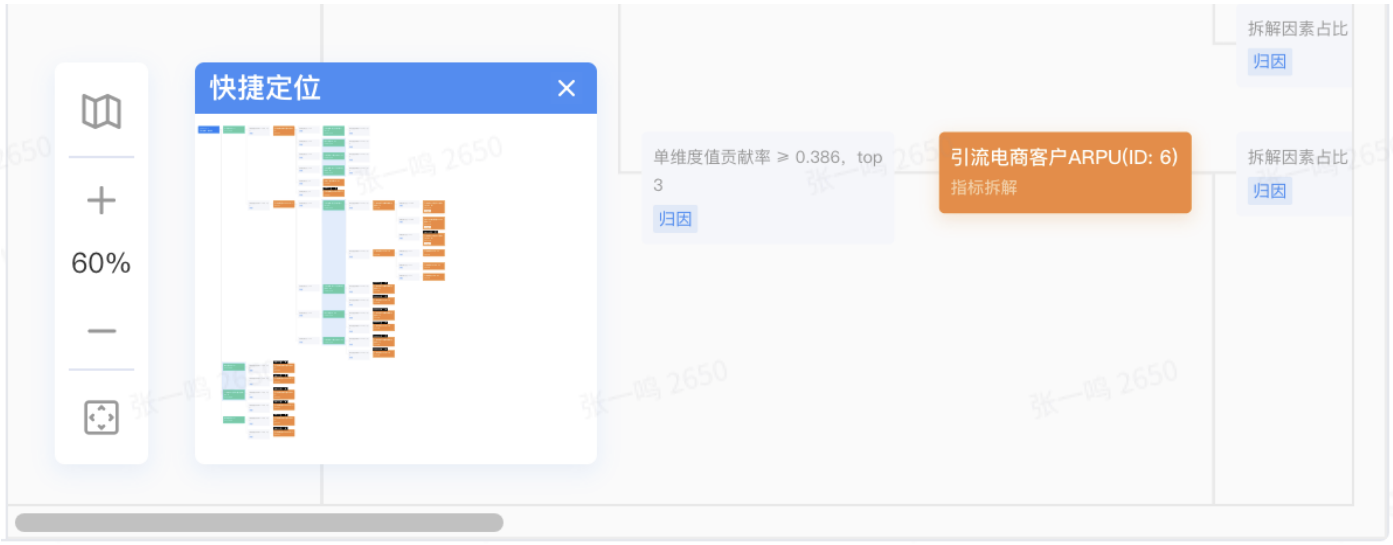
b. 触摸板双指操作 == 按住ctrl键滚动

c. 优点：实现起来简单，性能较高；配合dom+svg的方案缩放起来不会失真。

JavaScript

```
handleTwoFingerZoom: throttle(function(e) {
  if (e.ctrlKey) {
    e.preventDefault();
    e.stopImmediatePropagation();
    const s = Math.exp(-e.deltaY / 100);
    const scale = this.limitScale(this.value * s);
    this.$emit('input', scale);
  }
}),
```

缩略图



d. 使用组件循环引用，将tree-graph组件作为子组件，$props透传；

e. 使用使用 webpack 的异步 `import` 解决vue组件循环引用问题；

f. 缺点：在渲染较大分析树时（同时拥有近200个节点），性能较差

HTML

```
<template>
  <div class="graph-navigation" :style="navigationStyle">
    <div class="navigation-title">
      <div class="navigation-title-text">{{I18n.t('快捷定位')}}</div>
      <byted-icon class="navigation-title-close" name="close" color="#fff" @click="handleClose"></byted-icon>
    </div>
    <tree-graph :isNavigation="true" v-bind="$attrs" :graphScale="graphScale"/>
  </div>
</template>
```

JavaScript

```
  components: {
    // 使用 webpack 的异步 import，解决vue组件循环引用问题
    TreeGraph: () => import('../index.vue'),
  },
```

滚动到指定节点

使用 scrollIntoView({behavior: "smooth", block: "center", inline: "center"})

JavaScript

```
scrollNodeIntoView(nodeId) {
  this.$refs?.[nodeId]?.[0]?.$el?.scrollIntoView({behavior: "smooth", block: "center", inline: "center"});
},
```

数据结构

10. 镜像节点 - Proxy&Reflect

镜像节点是一种特殊的节点-改变源节点或者改变镜像节点，另外的节点都会被更新，但只是部分属性更新，比如镜像节点的parent属性和源节点的parent属性就是不同的。

- 最初的想法是，使用一个特殊的function去处理：特殊逻辑特殊处理，这没什么问题...

- 但实际上如果我们想使用这些function，我们就可能在watch中来回穿梭，去寻找，定位这些复杂的逻辑，这样的情况显示是不利于维护的，这个方案很快就被我们否定了...

- 镜像节点的这些特性，很容易让我们想到Proxy这种数据结构：我们试图让两个节点共享同一个数据资源，但是又想要保证节点的一些独特属性（比如parent）
  - Proxy可以通过handler中的get和set对想要的属性进行数据劫持，达到同步更新的效果，这里每一次新建一个镜像节点就是创建了一个Proxy，同时镜像节点有一个特殊的属性res，通过Reflect来获取源节点的属性，这个相当于对源节点的引用

JavaScript

```javascript
export function createMirrorNodeProxy(node, outerOptions = {}) {
  const options = {...outerOptions};
  const mirrorHandlers = {
    get(target, key) {
      if (options[key]) {
        return options[key];
      }
      let res = Reflect.get(...arguments);
      return res;
    },
    set(targt, key, value) {
      if (!isNil(options[key])) {
        options[key] = value;
        return true;
      }
      let res = Reflect.set(...arguments);
      return res;
    },
  };
  return new Proxy(node, mirrorHandlers);
}
```

这里相当于只是使用了Proxy的基本特性，值得注意的是，在Vue框架中需要把对Proxy的转换放到响应式之后去处理，在回填信息时这里是需要注意的。

Kotlin

```kotlin
this.treeData = treeData;
// 兼容proxy在Vue中响应式问题
this.treeData = parseProxyNode(this.treeData);
```

> Proxy当然还可以做很多更有意思的事情，利用handler，是可以操作对属性进行各种操作，比如值修正和计算属性等等

11. 指标&维度全局处理

指标和维度的基本信息在编辑整个分析树的过程中都是很重要的，因此整个处理流程我们迭代了两个版本，也做了一定的优化...

最初版本的处理

指标和维度的基本信息在编辑整个分析树的过程中都是很重要的，因此我们选择在编辑页开始加载时就拿到全量的指标和维度信息，并将该信息放到vuex中，并对这些数据使用了Object.freeze

JavaScript

```
SET_METRICS_DATA: (state, metricsData) => {
  state.metricsData = Object.freeze(metricsData);
},
```

- indexdDB

之前有提到会在编辑页加载前全量拉取指标&维度信息，这个接口还是很慢的，延迟渲染会影响首屏的显示效果，因此使用了indexdDB缓存指标&维度信息。

JavaScript

```
GetMetrics({commit, state}) {
  if (state.metricsData) {
    return Promise.resolve(state.metricsData);
  } else {
    // 先取indexeddb中缓存
    idbDataManage.get('metrics').then((data) => {
      if (!isEmpty(data)) {
        commit('SET_METRICS_DATA', data);
      }
    })
    return getMetrics().then((data) => {
      if (!isEmpty(data?.data)) {
        const metricsData = {};
        data.data.forEach((item) => {
          metricsData[item.id] = item;
        });
        commit('SET_METRICS_DATA', metricsData);
        // 强制更新缓存中数据
        idbDataManage.set('metrics', metricsData);
      } else {
        throw new Error('get metrics error');
      }
    });
  }
},
```

第二个版本

考虑到指标个数逐渐增长的趋势，一次性获取全量信息的行为就显得十分的不合理，因此我们决定对获取指标信息再次改造，从全量改造为按需获取

但由于之前是信息是放到Vuex中的，如果直接改为按需获取实际上会侵入业务代码的，最好的改造方案是只改动Vuex中的代码，那怎么改造呢?

无法复制加载中的内容

JavaScript

```
const cacheMetrics = {};

const metricProxyHandlers = {
  get(target, key) {
    let res = Reflect.get(...arguments);
    if (isString(key) && !isNaN(parseInt(key)) && key > 0) {
      const metricInfo = cacheMetrics[key];
      if (!metricInfo) {
        batchGetMetricsV2(key).then((data) => {
          if (data) {
            Object.entries(data).forEach(([metricId, metricInfo]) => {
              cacheMetrics[metricId] = metricInfo;
            });
            store.commit('SET_METRICS_DATA', {});
            store.commit('SET_IS_METRIC_DATA_INIT', true);
          }
        });
      } else {
        res = metricInfo;
      }
    }
    return res;
  },
  set(target, key, value) {
    cacheMetrics[key] = value;
    return true;
  }
}
```

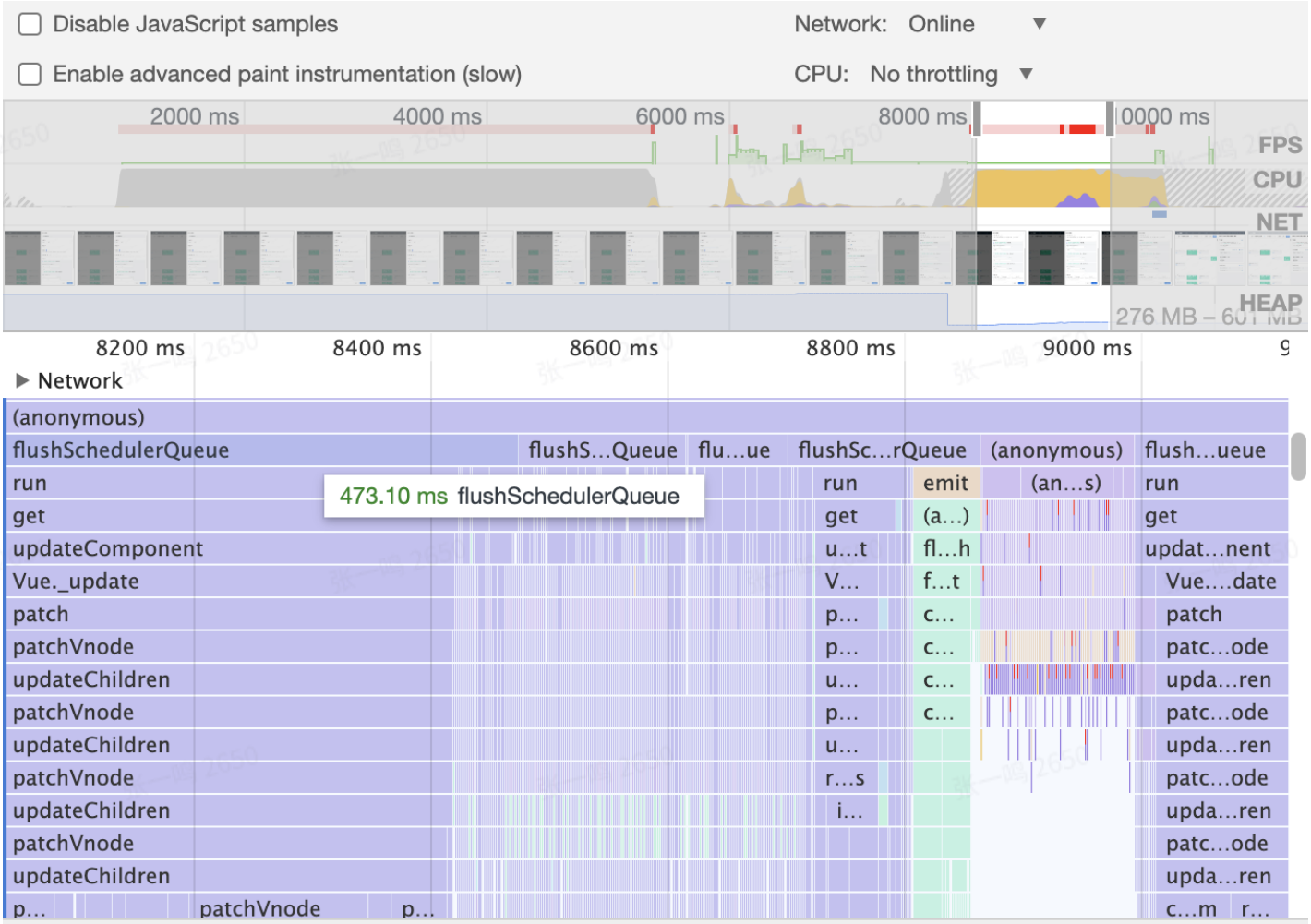
但是这个版本实际上在用户的体验可能会稍微差些，因为这个版本的缓存方案我们还没有发布，这也是我们在性能上做的一些牺牲

其他

12. 超大分析树节点刷新性能问题

遇到的问题：

自定义分析树上线后，很快就遇到了超大分析树的性能问题，主要集中在切换，删除，新增的节点会出现性能卡顿。chrome的火焰图可以看出，在进行这些操作时vue会进行大量的patch操作。



解决办法：实际上我犯了一个很愚蠢的问题，使用index做key...

设计原则

13. 组件是一种编程抽象，目的是复用。

a. DRY原则：Don't repeat yourself，不要开发重复的功能；

b. 三次原则：当某个功能第三次出现时，才进行"抽象化"；

> 软件的首要技术使命：管理复杂度

14. computed 优先于 watch

a. 滥用watch会导致数据流向不清晰（熵增）；
b. 计算属性是基于它们的响应式依赖进行缓存的；

15. 最好不要写出超大组件：

a. 当组件超过500行，就要准备拆分；
b. 当组件超过700行，就要开始重构；
c. 当组件超过1000行，就很难维护了；
d. 合适的组件行数一般在30~400行；

16. 不要滥用mixin和provide

a. mixin很容易发生冲突，并且可重用性是有限的；

> · 在 Vue 2 中，mixin 是将部分组件逻辑抽象成可重用块的主要工具。但是，他们有几个问题：
> · mixin 很容易发生冲突：因为每个特性的属性都被合并到同一个组件中，所以为了避免 property 名冲突和调试，你仍然需要了解其他每个特性。
> · 可重用性是有限的：我们不能向 mixin 传递任何参数来改变它的逻辑，这降低了它们在抽象逻辑方面的灵活性

b. provide使用较多会使重构变得困难，并且它提供的props是非响应式的；

整个treeConfig组件中，作为父级组件，为子组件提供了两个比较重要的依赖注入，分别是getNextNodeId和calcTreeData，用于获得递增节点id和生成和后端交互的treeNode。

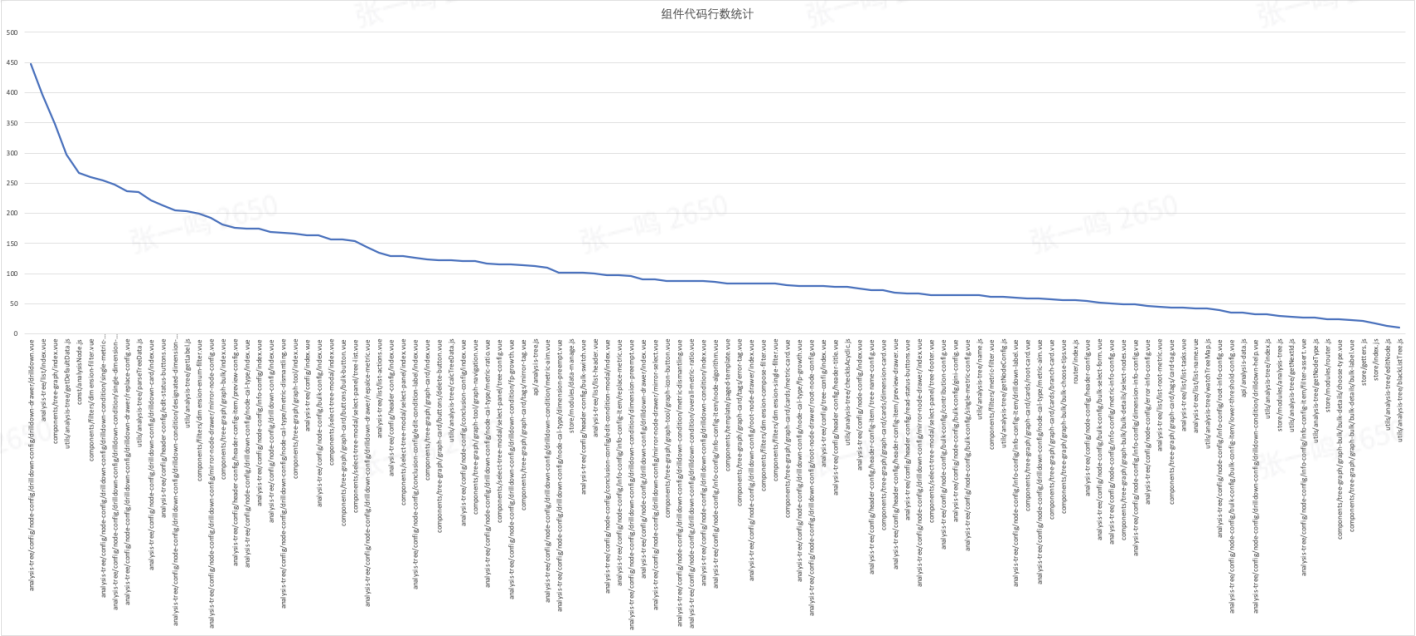
> 然而，依赖注入还是有负面影响的。它将你应用程序中的组件与它们当前的组织方式耦合起来，使重构变得更加困难。同时所提供的 property 是非响应式的。

JavaScript

```
provide() {
 return {
  getNextNodeId: this.getNextNodeId,
  calcTreeData: this.calcTreeData,
 };
},
```

开发成果

17. 组件的代码行数在可控范围内



新增115个vue/js文件，大多数代码行数在30 ~ 300之间，平均每个组件代码行数在110行左右。

以下是代码行数前十的组件，前三个组件需要在接下来开发注意拆分，不让代码行数过于膨胀。