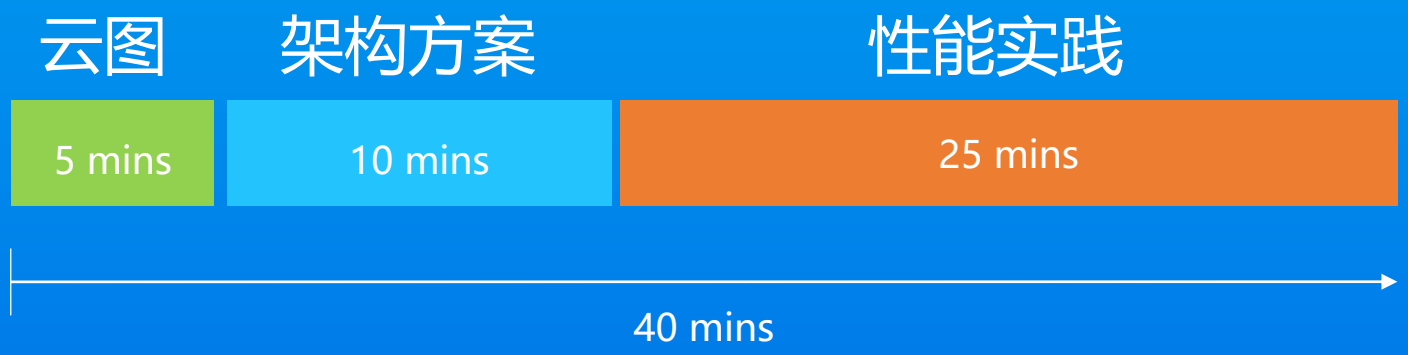
腾讯云

React 数据可视化应用实践

陈津@Tencent



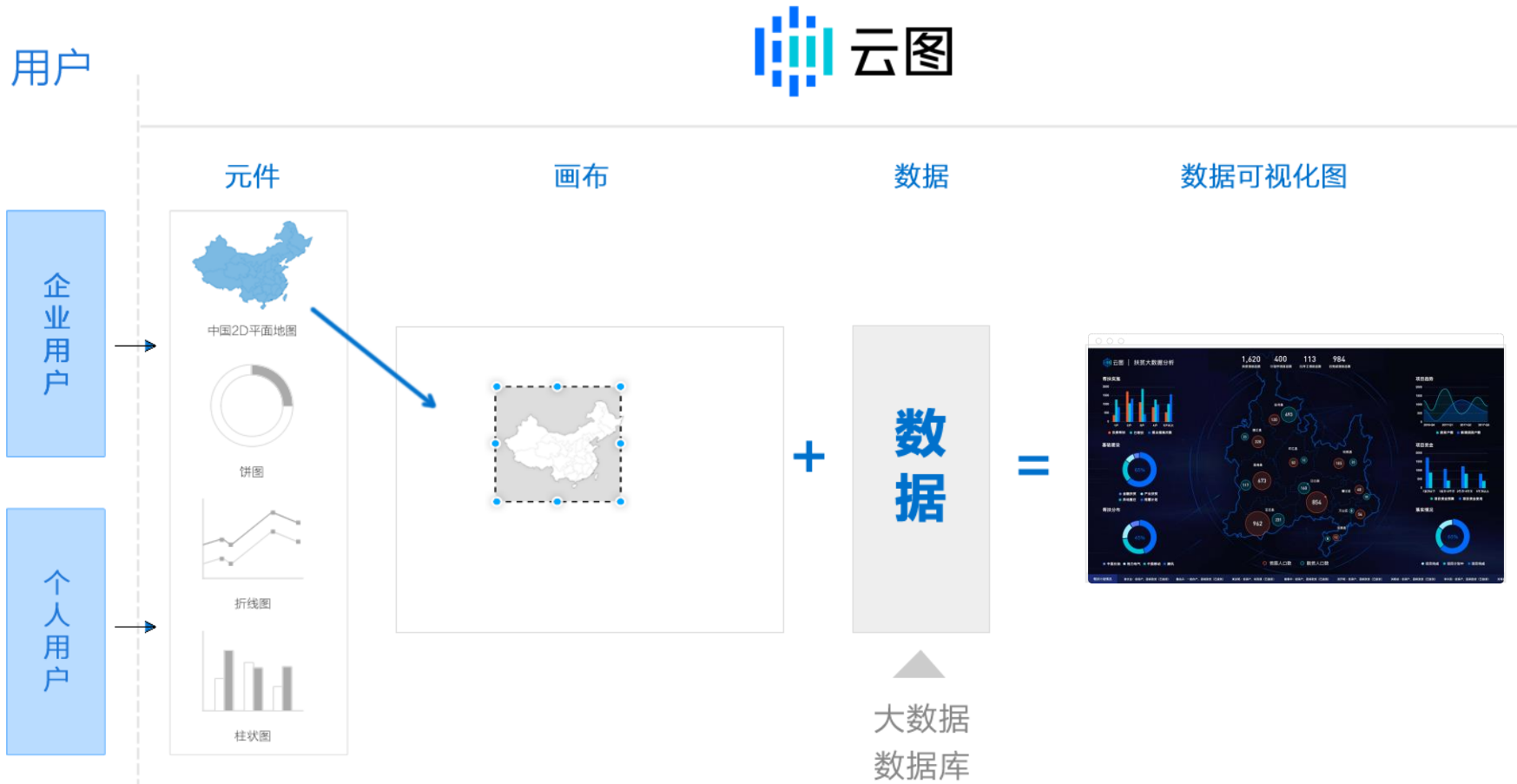
1 云图

遇见可视化

腾讯云

云图产品解构

无缝对接腾讯云大数据处理系统和数据库系统，完整大数据库业务线。



属性

背景

颜色 #0C1830

图片

PNG文件
文件大小：444.38 KB
重新上传 删除

请上传10MB以内JPG、PNG、GIF文件

位置 居中

拉伸 不拉伸

平铺 不平铺

位置大小

大小 1920 宽度 1080 高度

栅格

栅格大小 1

2 架构方案

我们如何做

腾讯云

前端架构

图表组件

- 基础组件
- 图表组件
- 属性定义
- 异步加载

画布

- 组件移动/布局
- 变焦缩放
- 层级控制
- 菜单

属性管理

- 基础组件
- 复合组件
- 数据映射配置

数据管理

- 静态数据
- csv文件
- API
- 腾讯云/公网数据库
- 腾讯云监控

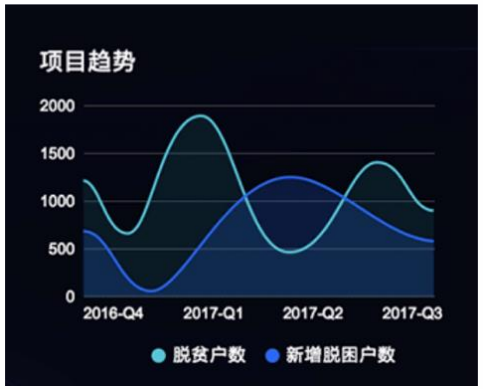
相关技术

- React/Redux/React-Redux/React-Router/Redux-Form
- Bere/TCFF
- D3/Threejs/ECharts/WebGL/Canvas/SVG
- Webpack/Babel
- ES 6/7/8
- TypeScript

图表组件

痛点 图表种类多，配置复杂，如何简化

解决 基础图表子组件化，使得基础图表表现一致，属性配置可复用，并可以进行多选配置



基础组件

图表组件

- 标题
- 坐标轴
- 图例

- 标题
- 坐标轴
- 图例

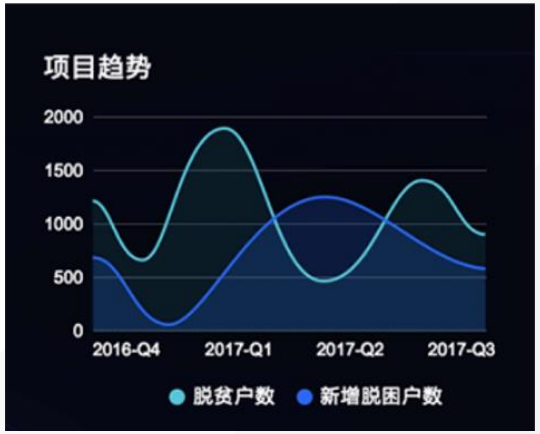
- 线

- 柱

属性管理

实现满足各种场景基础组件，方便动态组合复合组件

标题属性映射



3 性能优化

增强用户体验

腾讯云

性能优化

2 React-Redux

1 React

性能优化 - React

痛点 大屏展示图表多、数据多如何解决可视化编辑以及展示过程中的稳定性、卡顿问题

优化方案

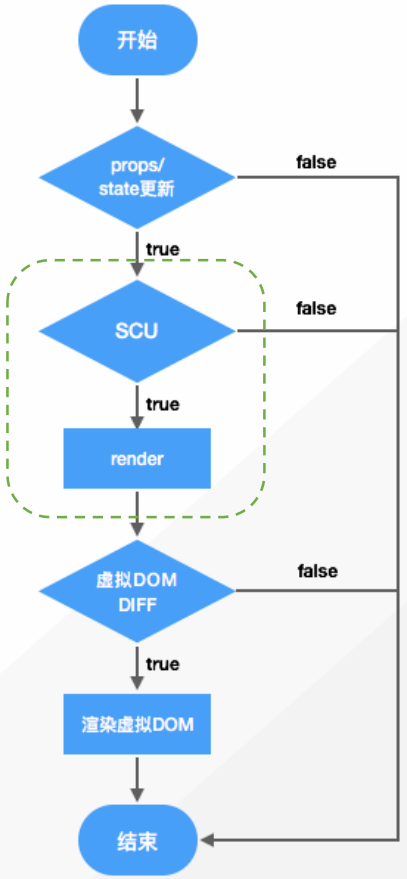
A. 使用PureComponent

B. 组件扁平化属性

C. 布局组件拆分

D. Render减负

React 渲染流程



性能优化 - React - PureComponent

解决 使用浅比较解决大多数情况下重复生成虚拟DOM问题

SCU shallowEqual策略

- 比较props/state 对象引用
- 比较props/state第一层属性

```
const hasOwn = Object.prototype.hasOwnProperty

function is(x, y) {
  if (x === y) {
    return x !== 0 || y !== 0 || 1 / x === 1 / y
  } else {
    return x !== x && y !== y
  }
}

export default function shallowEqual(objA, objB) {
  if (is(objA, objB)) return true

  if (typeof objA !== 'object' || objA === null ||
      typeof objB !== 'object' || objB === null) {
    return false
  }

  const keysA = Object.keys(objA)
  const keysB = Object.keys(objB)

  if (keysA.length !== keysB.length) return false

  for (let i = 0; i < keysA.length; i++) {
    if (!hasOwn.call(objB, keysA[i]) ||
        !is(objA[keysA[i]], objB[keysA[i]])) {
      return false
    }
  }

  return true
}
```

性能优化 - React - PureComponent

特别注意 避免产生对组件进行动态属性赋值的操作

陷阱1

```
// 每次会生成新的 Array 对象
<MyComponent values={this.props.values || []}/>
```



```
// 使用自己预先定义的默认数组，避免每次创建的情况
const defaultArray = [];
<MyComponent values={this.props.values || defaultArray}/>
```

陷阱2

```
// 1、内联函数
<MyComponent onChange={e => this.props.update(e.target.value)}/>

// 2、bind
update (e) {
    this.props.update(e.target.value)
}
<MyComponent onChange={this.update.bind(this)}/>
```



```
constructor(props) {
  super(props);
  // 绑定this
  this.update = this.update.bind(this)
}

update (e) {
    this.props.update(e.target.value)
}

<MyComponent onChange={this.update} />
```

性能优化 - React - 组件扁平化属性设计

解决 浅比较只比较第一层，充分利用浅比较优势

属性发生改变需要生成新的对象

```
{
    inner: {
        width,
        height
    },
    chartRect: {
        left,
        top,
        width,
        height
    },
    chartPadding: {
        left,
        top,
        right,
        bottom
    }
    xAxisSize,
    yAxisSize,
    zAxisSize,
    titleSize,
    legendSize
}
```

属性发生改变需要生成新的对象

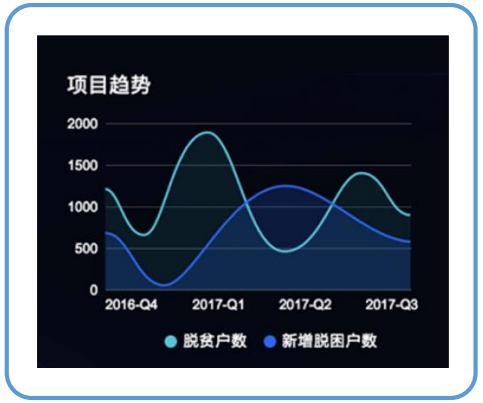
扁平化

```
{
    innerWidth,
    innerHeight,
    chartRectLeft,
    chartRectTop,
    chartRectWidth,
    chartRectHeight,
    chartPaddingLeft,
    chartPaddingTop,
    chartPaddingRight,
    chartPaddingBottom,
    xAxisSize,
    yAxisSize,
    zAxisSize,
    titleSize,
    legendSize
}
```

性能优化 - React - 布局组件拆分

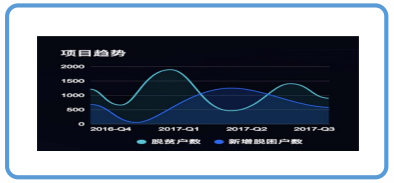
解决 用户在画布中拖动缩放图表导致图表重绘，用户操作感知卡顿

容器组件



鼠标拖动
位置/尺寸持续变化

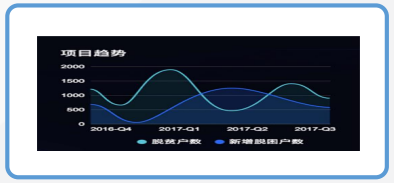
容器组件



容器组件位置/尺寸属性变化只导致容器组件重绘，由容器传递给图表组件的属性未变化，因此图表不会重绘。

鼠标释放
位置/尺寸停止变化

容器组件



用户鼠标松开操作完成，改变图表内部尺寸属性进行重绘。

性能优化 - React-Redux

痛点 大屏展示图表多、数据多如何解决可视化编辑以及展示过程中的稳定性、卡顿问题

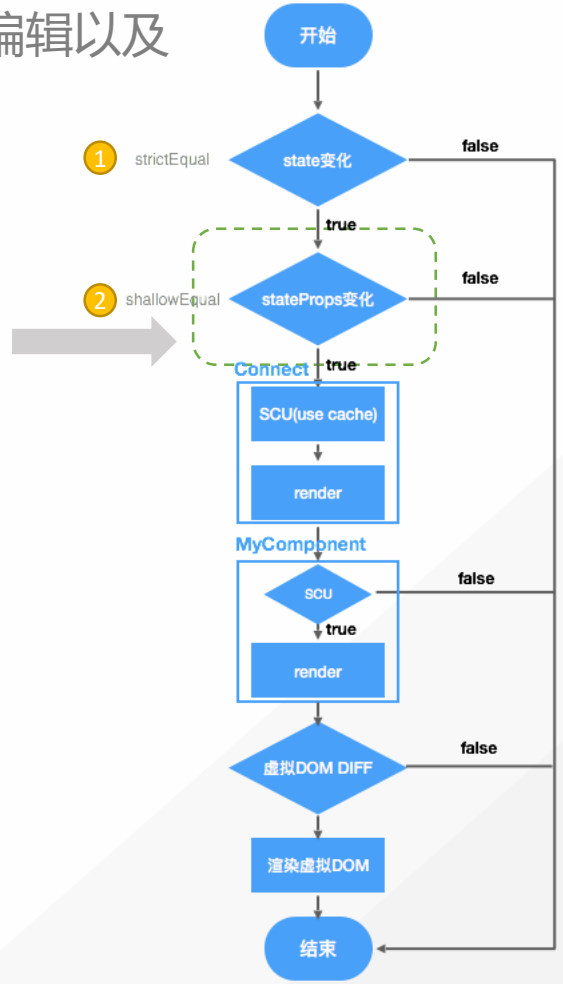
优化方案

A. 合理的状态树设计

B. 合理的connect

C. 尝试不可变数据结构

React-Redux 渲染流程




```
JS selectorFactory.js ×

51
52  function handleNewProps() {
53    if (mapStateToProps.dependsOnOwnProps)
54      stateProps = mapStateToProps(state, ownProps)
55
56    if (mapDispatchToProps.dependsOnOwnProps)
57      dispatchProps = mapDispatchToProps(dispatch, ownProps)
58
59    mergedProps = mergeProps(stateProps, dispatchProps, ownProps)
60    return mergedProps
61  }
62
63  function handleNewState() {
64    const nextStateProps = mapStateToProps(state, ownProps)
65    const stateChanged = !areStatePropsEqual(nextStateProps, stateProps) 2
66    stateProps = nextStateProps
67
68    if (stateChanged)
69      mergedProps = mergeProps(stateProps, dispatchProps, ownProps)
70
71    return mergedProps
72  }
73
74  function handleSubsequentCalls(nextState, nextOwnProps) {
75    const propsChanged = !areOwnPropsEqual(nextOwnProps, ownProps)
76    const stateChanged = !areStatesEqual(nextState, state) 1
77    state = nextState
78    ownProps = nextOwnProps
79
80    if (propsChanged && stateChanged) return handleNewPropsAndNewState()
81    if (propsChanged) return handleNewProps()
82    if (stateChanged) return handleNewState()
83    return mergedProps
84  }
```

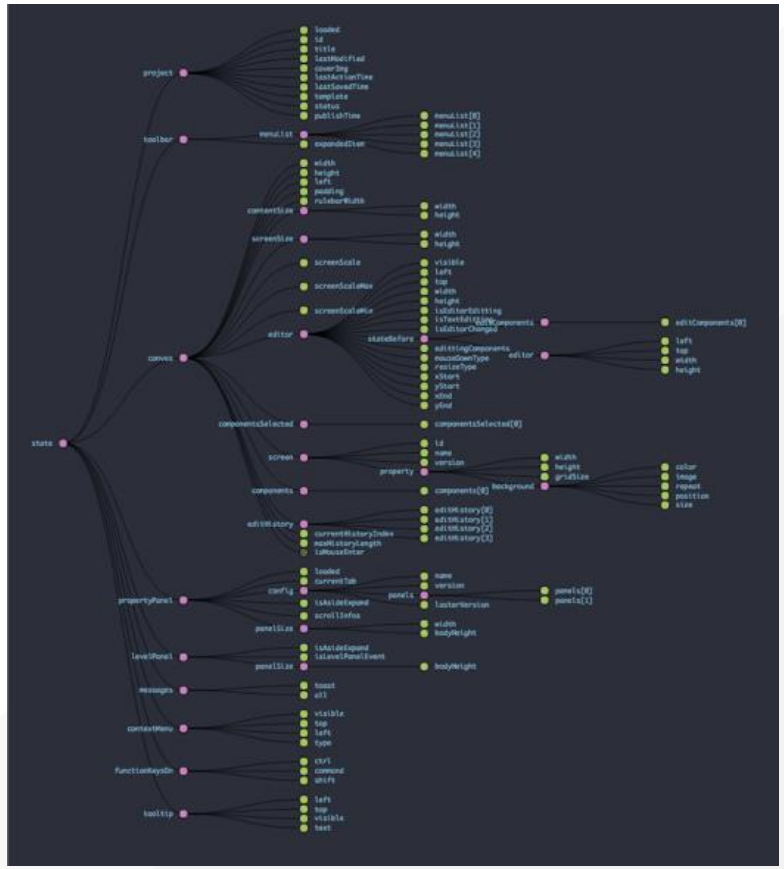
性能优化 - React-Redux - 合理的状态树设计

解决 复杂的状态树下分支状态变更导致交叉渲染

状态树设计思路

A. 扁平化

B. 数据解耦



性能优化 - React-Redux - 合理的connect

解决 复杂的状态树下分支状态变更导致交叉渲染

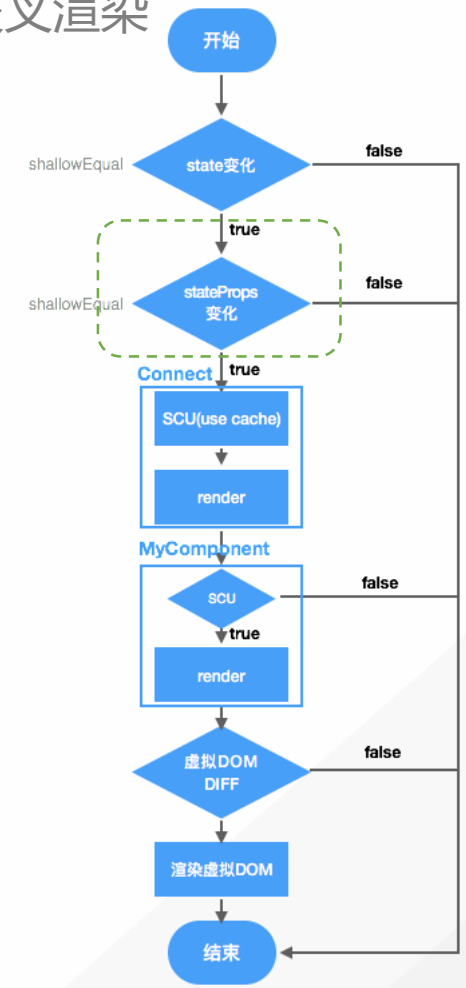
connect思路

A. 扁平化

B. 只取所需

C. 不宜过多

D. Reselect




```
connect(
    mapStateToProps,
    mapDispatchToProps,
    null,
    {
      areStatesEqual = strictEqual,
      areOwnPropsEqual = shallowEqual,
      // 改成你的判定方法
      areStatePropsEqual = shallowEqual,
      areMergedPropsEqual = shallowEqual,
    }
  )(MyComponent)
```

如果需要使用重写connect组件的SCU，建议从react-redux级别重写，避免react-redux冗余的判定。

性能优化 - React-Redux - 尝试不可变数据结构

解决 浅比较带来的内存消耗与复杂控制并使深比较变得简单

不可变数据结构相关库

A. Immutable 侵入性

B. seamless-immutable IE9+