web 前端作业——基于 React 组件化思路开发的山寨版创新港图书馆大数据可视化平台

梁子，软件学院，S0083，3120305385

日期：Sun Jul 11 17:08:49 2021

目录

1 场景及功能介绍

本作业拟实现的功能是：一个被目前广泛使用的可视化界面。观察到创新港的图书馆中便有屏幕投放这种前端界面，因此打算仿照之实现一个山寨版本。整体结果图示如下图所示：

从图中可以看出，该页面所实现的功能主要包括以下几个方面：

1. 图书馆进出情况分析；
2. 图书借阅情况分析；
3. 各类基础信息统计；
4. 图书借阅类别分布的统计；
5. 借阅者年级分布的统计；
6. 还书机使用频率曲线；
7. 实时时间显示
8. 没有太大含义的中国地图；

关于实现过程请见下一章。

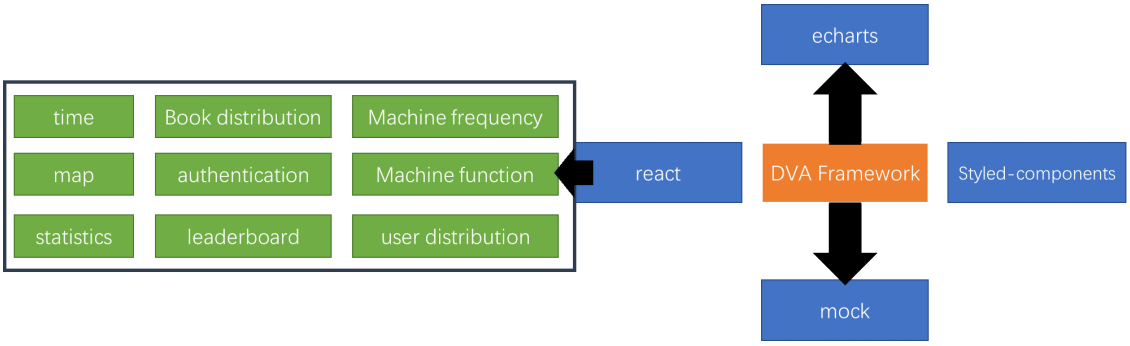
2 界面功能实现

2.1 总体架构

本作业所描述平台整体上包含了两部分：前端实现和后端实现。当然，由于后端实现较为简单（即基于简单数据库的对数据的整合和处理），因此本章重点介绍前端部分。整体上而言，本文使用阿里巴巴的前端框架 dva 进行套装，整体上可分为三大模块：

1. 界面。基于 react 编写相关组件，用以展示每一个模块；
2. 后端数据。使用 json 对象模拟后端数据，将所有后端数据存放在固定的文件夹中；
3. 操作逻辑和前端数据。基于 dva 框架进行存储；
4. 风格样式。使用 styled-components 进行组件化的 css 样式使用；
5. 可视化，基于百度的 echarts 进行可视化。

下图展示了整体上的模型结构实现。下面介绍各个组块的实现原则。



2.2 各组块实现

各个组块的实现均基于 react 进行，主要包括以下 8 个原子级别的组块：

1. 时间组块；
2. 标题组块；
3. 地图组块；
4. 图书馆重要统计值数值展示组块；
5. 图书馆进出人数展示组块；
6. 图书借阅榜组块；
7. 图书借阅类别分析组块；
8. 借阅者年级分布展示组块；
9. 还书机使用功能分布组块；
10. 还书机使用频率组块；

基于这八个原子组块，主要将页面划分为三个部分：

1. 顶部，包含标题和当前时间；
2. 左部，包含图书馆进出人数展示、图书借阅榜两大部分；
3. 中部，包含地图和重要统计数值展示；
4. 右部，包含图书馆的一些部分分析结果，及还书机的使用情况；

之后，这三个部分将组成整体的页面。

前面已提及，为了更好地展示解耦前端的各个部分，上述各个组块的风格样式以及其交互逻辑均是分离开的，而后使用 dva 中的 connect 方法将模型和页面进行拼接。下面以左侧组件试图为例进行介绍。如前面所述，左侧主要包括图书馆进出人数展示的折线图以及滚动的图书借阅榜，基于此二者的 react 组件类中的 render 函数被定义为：

```
render() {
   const { userSitua, trafficSitua, accessFrequency, peakFlow } = this.props;
   return (
     // 风格
     <LeftPage>
 {/* 顶部图表 */}
 <LeftTopBox>
   <BorderBox12 className='left-top-borderBox12'>
     <div className='left-top'>
       <ModuleTitle>
   <i className='iconfont'>&#xe78f;</i>
   <span>今日图书馆进出情况</span>
       </ModuleTitle>

       <div className='title-dis'>
   <span>
     当前参观人数(小时):
     <span className='title-dis-keyword'>{accessFrequency}人</span>
   </span>

   <span>
     今日总人数:
     <span className='title-dis-keyword'>{peakFlow}人</span>
   </span>
       </div>
       {/* 图表 */}
       <TrafficSituation trafficSitua={trafficSitua}></TrafficSituation>
     </div>
   </BorderBox12>
 </LeftTopBox>

 {/* 底部图表 */}
 <LeftBottomBox>
   <BorderBox13 className='left-bottom-borderBox13'>
     <div className='left-bottom'>
       <ModuleTitle>
   <i className='iconfont'>&#xe88e;</i>
   <span>本周图书借阅榜</span>
       </ModuleTitle>
       {/* 图表 */}
       <UserSituation userSitua={userSitua}></UserSituation>
     </div>
   </BorderBox13>
 </LeftBottomBox>
     </LeftPage>
```

```
    );
  }
}
```

从中可以看出，所有的组件均被包括在 <LeftPage> 之内，在之中，划分得到了底部顶部两个 box，而核心的图标区域是两个 react 组件，TrafficSituation 和 UserSituation。下面先对这几个组件进行简单介绍。

首先，LeftPage 以及两个 box 均是针对于 css 样式风格而撰写的组件，借用了 styled-components 的写法，如 LeftTopBox 就包含了如下的样式设定：

```
export const LeftTopBox = styled.div‘
  position: relative;
  height: 4.375rem;
  width: 100%;
  .left-top-borderBox12 {
    width: inherit;
    height: inherit;
    padding: 0.1875rem;
    .left-top {
      width: 100%;
      height: 100%;
      border-radius: 10px;
      background-color: rgba(19, 25, 47, 0.6);
      .title-dis {
  margin-top: 0.1875rem;
  display: flex;
  justify-content: space-around;
  align-items: center;
  font-size: 0.2rem;
  color: #c0c9d2;
  &-keyword {
    padding-left: 0.125rem;
    color: #47dae8;
  }
      }
    }
  }
‘;
```

而两个 Situation，均是 react 化的 echarts 组件，该组件一方面需要满足可视化库 echarts 的一些设定，另一方面又需要满足 react 的封装风格，以出口人流量为例，可以撰写如下:

```
class TrafficSituation extends PureComponent {
  constructor(props) {
    super(props);
    this.state = {
      renderer: 'canvas',
    };
  }

  render() {
    const { renderer } = this.state;
```

```
    const { trafficSitua } = this.props;
    return (
      <div
  style={{
    width: '5.375rem',
    height: '3.125rem',
  }}>
  {trafficSitua ? (
    <Chart renderer={renderer} option={trafficOptions(trafficSitua)} />
  ) : (
    ''
  )}
      </div>
    );
  } //endrender
}

export default TrafficSituation;
```

其中，相关数据信息是从上游，也就是 leftpage 组件流到该组件之内的。关于该数据如何流入到 LeftPage，后续在介绍数据逻辑时进行介绍。可以发现，该代码的核心步骤在于使用 Chart 组件进行数据和选项的配置，关于基础的 Chart 组件如何撰写，echarts 提供了示例代码，如下：

```
export default class Chart extends PureComponent {
  constructor(props) {
    super(props);
    this.state = {
      width: '100%',
      height: '100%',
    };
    this.chart = null;
  }
  // 异步函数
  async componentDidMount() {
    // 初始化图表
    await this.initChart(this.el);
    // 将传入的配置(包含数据)注入
    this.setOption(this.props.option);
    // 监听屏幕缩放，重新绘制 echart 图表
    window.addEventListener('resize', debounce(this.resize, 100));
  }

  componentDidUpdate() {
    // 每次更新组件都重置
    this.setOption(this.props.option);
  }

  componentWillUnmount() {
    // 组件卸载前卸载图表
    this.dispose();
  }
```

```
render() {
  const { width, height } = this.state;

  return (
    <div
className='default-chart'
ref={el => (this.el = el)}
style={{ width, height }}
    />
  );
}


initChart = el => {
  // renderer 用于配置渲染方式 可以是 svg 或者 canvas
  const renderer = this.props.renderer || 'canvas';

  return new Promise(resolve => {
    setTimeout(() => {
this.chart = echarts.init(el, null, {
  renderer,
  width: 'auto',
  height: 'auto',
});
resolve();
    }, 0);
  });
};

setOption = option => {
  if (!this.chart) {
    return;
  }

  const notMerge = this.props.notMerge;
  const lazyUpdate = this.props.lazyUpdate;

  this.chart.setOption(option, notMerge, lazyUpdate);
};
dispose = () => {
  if (!this.chart) {
    return;
  }

  this.chart.dispose();
  this.chart = null;
};
resize = () => {
  this.chart && this.chart.resize();
```

```
  };
  getInstance = () => {
    return this.chart;
  };
}
```

其中，purecomponent 是一种较为特殊的 component，该类仅仅当上游 props 发生改变或自身的 state 发生改变时才对自身进行重新渲染。

当对出口流量的组件形式了解后，另一个问题就是如何配置 echarts 对象，此处直接针对 echarts 官方提供的 API 参数进行修改即可。具体为：

```
export const trafficOptions = (params) => ({
  title: {
    show: false,
  },
  legend: {
    show: true,
    top: '5%',
    textStyle: {
      color: '#c0c9d2',
    },
  },
  tooltip: {
    trigger: 'axis',
    axisPointer: {
      lineStyle: {
  color: {
    type: 'linear',
    x: 0,
    y: 0,
    x2: 0,
    y2: 1,
    colorStops: [
      {
        offset: 0,
        color: 'rgba(0, 255, 233,0)',
      },
      {
        offset: 0.5,
        color: 'rgba(255, 255, 255,1)',
      },
      {
        offset: 1,
        color: 'rgba(0, 255, 233,0)',
      },
    ],
    global: false,
  },
      },
    },
```

```
},
grid: {
  top: '15%',
  left: '10%',
  right: '5%',
  bottom: '10%',
},
xAxis: {
  type: 'category',
  axisLine: {
    show: true,
  },
  splitArea: {
    color: '#f00',
    lineStyle: {
color: '#f00',
    },
  },
  axisLabel: {
    color: '#BCDCF0',
  },
  splitLine: {
    show: false,
  },
  boundaryGap: false,
  data: params.timeList,
},

yAxis: {
  type: 'value',
  min: 0,
  splitLine: {
    show: true,
    lineStyle: {
color: 'rgba(255,255,255,0.1)',
    },
  },
  axisLine: {
    show: true,
  },
  axisLabel: {
    show: true,
    margin: 10,
    textStyle: {
color: '#d1e6eb',
    },
  },
  axisTick: {
    show: false,
  },
```

```
},
series: [
  {
    name: '进入人数',
    type: 'line',
    smooth: true, //是否平滑
    lineStyle: {
normal: {
  color: '#00b3f4',
  shadowColor: 'rgba(0, 0, 0, .3)',
  shadowBlur: 0,
  shadowOffsetY: 5,
  shadowOffsetX: 5,
},
    },
    label: {
show: false,
position: 'top',
textStyle: {
  color: '#00b3f4',
},
    },
    // 去除点标记
    symbolSize: 0,
    // 鼠标放上去还是要有颜色的
    itemStyle: {
color: '#00b3f4',
    },
    // 设置渐变色
    areaStyle: {
normal: {
  color: new echarts.graphic.LinearGradient(
    0,
    0,
    0,
    1,
    [
      {
  offset: 0,
  color: 'rgba(0,179,244,0.3)',
      },
      {
  offset: 1,
  color: 'rgba(0,179,244,0)',
      },
    ],
    false
  ),
  shadowColor: 'rgba(0,179,244, 0.9)',
  shadowBlur: 20,
```

```
},
    },
    data: params.outData,
  },
  {
    name: '走出人数',
    type: 'line',
    smooth: true, //是否平滑
    // 阴影
    lineStyle: {
normal: {
  color: '#00ca95',
  shadowColor: 'rgba(0, 0, 0, .3)',
  shadowBlur: 0,
  shadowOffsetY: 5,
  shadowOffsetX: 5,
},
    },
    label: {
show: false,
position: 'top',
textStyle: {
  color: '#00ca95',
},
    },
    // 去除点标记
    symbolSize: 0,
    itemStyle: {
color: '#00ca95',
    },
    // 设置渐变色
    areaStyle: {
normal: {
  color: new echarts.graphic.LinearGradient(
    0,
    0,
    0,
    1,
    [
      {
  offset: 0,
  color: 'rgba(0,202,149,0.3)',
      },
      {
  offset: 1,
  color: 'rgba(0,202,149,0)',
      },
    ],
    false
  ),
```

```
    shadowColor: 'rgba(0,202,149, 0.9)',
    shadowBlur: 20,
  },
      },
      data: params.inData,
    },
  ],
});
```

通过这种方式，即可完成对一个 echarts 折线图的全部处理了。

另外的问题是，数据是如何从后端流入到当前组件的。依照 react 的思路，一般而言，数据的变动包含两个原则：

1. 对于一个组件节点，如果该组件节点触发了状态变动，则依照变动情况进行处理：若只影响以当前节点为根节点的子树，则在当前节点改变，否则需要找到最小公共子树，在其之上改变状态，或在根节点改变状态。老师上课所写的 todo APP 即是此种逻辑；
2. 对于一个组件节点，其所需要的数据，只能是自身状态提供的，或是父节点传递过来的；

此处自然也是基于这两个原则进行的，不过稍有不同。这里的不同主要体现在数据传送的方法上，同样以 leftpage 为例，数据从 leftpage 传送到各个子组件是遵循上述原则的。不过，由于直接同外界交互，leftpage 获取数据是依照 dva 框架的形式进行的，该交互主要包括三个部分：数据来源，数据处理、数据导出展示。

比如，对于 leftpage，需要获得两个对象，分别是人流量信息和图书排行榜信息，这些信息从后端获取，因此第一个操作是一个 get 操作。如果获取失败，则进入异常处理环节，否则便需要将获取得到的数据传入到 leftpage 组件之后，令之使用 pros 获得。对于上述整个流程，在从后端获取数据的环节，主要代码是：

```
export default function request(url, options) {
  return fetch(url, options)
    .then(checkStatus)
    .then(parseJSON)
    .then(data => ({ data }))
    .catch(err => ({ err }));
}

export const getLeftPageData = async () => {
  return request('/api/leftPageData').then(response => {
    return response.data;
  });
};
```

对于所获得的数据，dva 会设置如下的 model：

```
export default {
  // 命名空间 (必填)
  namespace: 'leftPage',

  // 数据
  state: {},

  // 路由监听
  subscriptions: {
```

```
    setup({ dispatch, history }) {
      return history.listen((location, action) => {
  // 参数可以直接简写成{pathname}
  if (location.pathname === '/') { //当进入当前页面就执行获取数据这一action
    dispatch({ type: 'getLeftPageData' });
  }
      });
    },
  },

  // 异步请求    action处理器，用以对异步动作进行处理
  effects: {
    *getLeftPageData({ payload }, { call, put }) {
      const data = yield call(getLeftPageData);
      if (data) {
  yield put({
    type: 'setData',
    payload: data,
  });
      } else {
  console.log('获取左侧数据数据失败');
      }
    },
  },

  // 同步操作
  reducers: {
    setData(state, action) {
      return { ...state, ...action.payload };
    },
  },
};
```

可以看出，该过程主要包含了如下几个部分：

1. 路由监听，主要是设置何时进行 dispatch，也就是什么时候进行数据获取；
2. effects，负责对异步动作进行处理，此处即是对获取左侧数据进行这一动作进行执行。同时，该过程使用了标准的 yield-put 结构，该结构会在需要时尝试运行一个动作，如果成功，则派发后续动作（put），以调用同步操作 setData；
3. reducers，等价于原始 react 中的 setState。

基于以上的介绍，整体的前端运行方式就较为清晰了。后续是如何对之进行安装和使用。

3 安装与运行

3.1 安装

基于 npm 进行依赖安装

```
npm install
```

主要包括以下依赖：

```
"dependencies": {
   "@jiaminghi/data-view-react": "^1.2.4",
   "dva": "^2.4.1",
   "echarts": "^4.9.0",
   "react": "^16.2.0",
   "react-dom": "^16.2.0",
   "sass-loader": "8.0.2",
   "styled-components": "^5.2.0"
 },
```

其中，sass 的安装可能会出现报错，可以先进行额外的处理。

3.2 运行

运行下列命令，即可在浏览器打开相关页面，F11 进入全屏中即可使用。

```
npm start
```