React全家桶

1.React介绍

1.React起源与发展

React 起源于 Facebook 的内部项目，因为该公司对市场上所有 JavaScript MVC 框架，都不满意，就决

定自己写一套，用来架设Instagram 的网站。做出来以后，发现这套东西很好用，就在2013年5月开源

了。

2.React与传统MVC的关系

轻量级的视图层库!

React不是一个完整的MVC框架，最多可以认为是MVC中的V（View），甚至React并不非常认可MVC开

发模式；React 构建页面 UI 的库。可以简单地理解为，React 将界面分成了各个独立的小块，每一个块

就是组件，这些组件之间可以组合、嵌套，就成了我们的页面。

3.React的特性

1.高效

2.性能好

3.单向数据流
4.组件化

2.cra（create-react-app）

近期 create-react-app更新了

他里面对node的版本有要求了 node的版本不能低于14了

注意:win7系统node的版本不能大于12（需要更新系统）

安装

1.全局安装**create-react-app**

```
npm install -g create-react-app
```

查看版本 create-react-app --version

2.cd到指定文件夹下

3.创建项目

```
create-react-app 项目名          your-app 注意命名方式
```

注意：如果不想全局安装可以使用npx来进行项目的安装

```
npx create-react-app myapp 也可以实现相同的效果
```

这需要等待一段时间，这个过程实际上会安装三个东西

react: react的顶级库

react-dom: 因为react有很多的运行环境，比如app端的react-native, 我们要在web上运行就使用

react-dom

react-scripts: 包含运行和打包react应用程序的所有脚本及配置

出现下面的界面，表示创建项目成功

```
Success! Created your-app at /dir/your-app
Inside that directory, you can run several commands:
npm start
Starts the development server.
npm run build
Bundles the app into static files for production.
npm test Starts the test runner. npm run eject Removes this tool and
copies build dependencies, configuration files and scripts into the app
directory. If you do this, you can’t go back! We suggest that you begin
by typing:

cd your-app
npm start

Happy hacking!
```

4.cd到你创建的项目下

5.npm start 启动项目

文件结构

生成项目的目录结构如下:

├── README.md 使用方法的文档

├── node_modules 所有的依赖安装的目录

├── package-lock.json 锁定安装时的包的版本号,保证团队的依赖能保证一致。

├── package.json

├── public 静态公共目录

└── src 开发用的源代码目录

常见问题

npm安装失败

1.切换为npm镜像为淘宝镜像

```
  npm config set registry https://registry.npm.taobao.org
```

2.使用yarn，如果本来使用yarn还要失败，还得把yarn的源切换到国内

yarn就是和npm一样 都是一个包管理工具

yarn是由 facebook推出的一个包管理工具

yarn安装：**npm install -g yarn**

yarn和**npm** 的对照表

功能	**npm**	**yarn**
初始化	npm init	yarn init
安装依赖	npm install	yarn install或者 yarn
新增依赖	npm install --save xxx	yarn add xxx
全局安装	npm install -g xxx	yarn global add xxx
同时下载多个	npm install --save xx1 xx2	yarn add xx1 xx2
删除依赖	npm uninstall --save xxx	yarn remove xxx

如果觉得yarn默认下载很慢 那么我们可以把yarn切换成淘宝镜像地址

```
yarn config set registry https://registry.npm.taobao.org/
```

3.如果还没有办法解决，请删除node_modules及package-lock.json然后重新执行 npm install命令

4.再不能解决就删除node_modules及package-lock.json的同时清除npm缓存 npm cache clean --force 之后再执行 npm install 命令

3.编写第一个react应用程序

react开发需要引入多个依赖文件：react.js、react-dom.js，分别又有开发版本和生产版本，create

react-app里已经帮我们把这些东西都安装好了。把通过CRA创建的工程目录下的src目录清空，然后在

里面重新创建一个index.js. 写入以下代码:

```
// 从 react 的包当中引入了 React。只要你要写 React.js 组件就必须引入React，因为react里有 一种语法叫JSX，稍后会讲到JSX，要写JSX，就必须引入React
import React from 'react'
// ReactDOM 可以帮助我们把 React 组件渲染到页面上去，没有其它的作用了。它是从 react-dom 中 引入的，而不是从 react 引入。
import ReactDOM from 'react-dom'
// ReactDOM里有一个render方法，功能就是把组件渲染并且构造 DOM 树，然后插入到页面上某个特定的 元素上
ReactDOM.render(
// 这里就比较奇怪了，它并不是一个字符串，看起来像是纯 HTML 代码写在 JavaScript 代码里面。语 法错误吗？这并不是合法的 JavaScript 代码，“在 JavaScript 写的标签的”语法叫 JSX- JavaScript XML。
    <h1>欢迎进入React的世界</h1>,
// 渲染到哪里
    document.getElementById('root')
)
```

4.JSX与组件

jsx== javascript and xml 他是一个新的语法扩展 在react中使用jsx的语法来进行页面内容的描述

在实际开发中，JSX 在产品打包阶段都已经编译成纯 JavaScript，不会带来任何副作用，反而会让代码

更加直观并易于维护。 编译过程由Babel 的 JSX 编译器实现。

jsx 当遇见**<** 当**html**解析遇见**{}**当**js**解析

原理是什么呢?

要明白JSX的原理，需要先明白如何用 JavaScript 对象来表现一个 DOM 元素的结构?

看下面的DOM结构

```
    <div class='app' id='appRoot'>
        <h1 class='title'>欢迎进入React的世界</h1>
        <p>React.js 是一个帮助你构建页面 UI 的库 xixi大帅哥 </p>
    </div>
```

上面这个 HTML 所有的信息我们都可以用 JavaScript 对象来表示:

```
{
```

```
        tag: 'div',
        attrs: {//标签的属性表示
            className: 'app',
            id: 'appRoot'
        },
        children: [//子dom
            {
                tag: 'h1',//节点
                attrs:{ className: 'title' },//属性
                children:
                    ['欢迎进入React的世界']//文本节点
            },
            {
                tag:'p',
                attrs: null,
                children:
                    ['React.js 是一个构建页面 UI 的库']
            }
        ]
    }
```

但是用 JavaScript 写起来太长了，结构看起来又不清晰，用 HTML 的方式写起来就方便很多了。

于是 **React.js** 就把 **JavaScript** 的语法扩展了一下，让 **JavaScript** 语言能够支持这种直接在 **JavaScript** 代

码里面编写类似 **HTML** 标签结构的语法，这样写起来就方便很多了。编译的过程会把类似 **HTML** 的 **JSX**

结构转换成 **JavaScript** 的对象结构。

下面就是jsx的语法：

```
import React from 'react'
import ReactDOM from 'react-dom'

ReactDOM.render(
    <div className='app' id='appRoot'>
        <h1 className='title'>欢迎进入React的世界</h1>
        <p>React.js 是一个构建页面 UI 的库 </p>
    </div>,
    document.getElementById('root')
)
```

编译之后将得到这样的代码：

```
import React from 'react'
```

```
import ReactDOM from 'react-dom'

ReactDOM.render(
        React.createElement("div", {
            className: 'app', id: 'appRoot'
        },
        React.createElement( "h1",
        { className: 'title' }, "欢迎进入React的世界"
        ),
        React.createElement( "p",
        null, "React.js 是一个构建页面 UI 的库"
        ),
    document.getElementById('root')
)
```

React.createElement 会构建一个 JavaScript 对象来描述你 HTML 结构的信息，包括标签名、属性、

还有子元素等, 语法为

```
React.createElement( type, [props], [...children] )
```

组件

扩展

模块与模块化

模块：用来封装可以重复使用的js代码块

模块化：整个项目都是使用模块的方式来完成的

组件与组件化

组件： 用来封装重复使用的ui代码块

组件化：整个项目都是使用组件的方式来完成的

组件的概念

组件就是把ui部分拆分成一个个独立的并且可以重复使用的部件 在吧这些部件拼装在一起 形成一个页面

组件的分类

函数组件/无状态组件/工厂组件

1.函数组件的首字母必须大写首字母必须大写首字母必须大写首字母必须大写首字母必须大写首字母必须大写首字母必须大写

2.函数中必须有一个return　return 一段jsx

```
// 我是函数组件的例子
import React from 'react'
import ReactDOM from 'react-dom'


// 函数组件
let Fun=()=>{
    return (
        <div>
            <h1>我是一个函数组件</h1>
        </div>
    )
}

ReactDOM.render(
    <Fun></Fun>,
    document.getElementById('root')
)
```

类组件

ES6的加入让JavaScript直接支持使用class来定义一个类，react创建组件的方式就是使用的类的继承，

ES6 class 是目前官方推荐的使用方式，它使用了ES6标准语法来构建

1.在类中必须必须必须有一个render的方法　其中必须要有一个return 一段jsx

2.这个类要继承React.Component

使用class定义一个类 并且继承于react的组件 在其中必须有一个render的方法 在return一段jsx

```
import React from 'react'
import ReactDOM from 'react-dom'


// 类组件
class Demob extends React.Component {
    render() {// 就是渲染的意思　执行这个render他就会执行jsx
```

```
        return (
            <div>
                我是类组件
            </div>
        )
    }
}

ReactDOM.render(
    <Demob></Demob>,
    document.getElementById('root')
)
```

jsx使用数据

jsx 当遇见**<** 当**html**解析遇见**{}**当**js**解析

组件中怎么使用变量呢?

函数组件

使用变量

```
import React from 'react'

export default function Fcbdemo() {

    let text="你好我是函数组件的变量"
  return (
    <div>Fcbdemo------{text}</div>
  )
}
```

{}是当js解析

```
import React from 'react'

export default function Fcbdemo() {
    let text="你好我是函数组件的变量"
    let i=9;
    let k=5;
    let bool=false;
    function fun(){
        return "我是函数"
    }
  return (
    <div>
        Fcbdemo------{text}---{i+k}--{bool?"你好":"你坏"}--{fun()}
    </div>
```

```
  )
}
```

类组件

```
import React, { Component } from 'react'



export default class classdemo extends Component {
  render() {
      let text="我是类组件的变量"
    return (
      <div>classdemo--{text}</div>
    )
  }
}
```

属性插变量

使用单大括号来进行属性查变量的设置

类组件函数组件语法相同

```
// 创建函数组件
let Fcdemo = () => {
    let text="点我去百度";
    let ahref="http://www.baidu.com"
    return (
        <div>
            <a href={ahref}>{text}</a>
        </div>

    )
}

export default Fcdemo
```

组件的样式

行内样式

行内样式需要写入一个样式对象

注意语法 第一个**{}**是**jsx**的语法 第二个*{}*是对象的语法

函数组件

```
// 我是函数组件的例子
import React from 'react'
import ReactDOM from 'react-dom'

let text = "你好"
// 函数组件
let Fun = () => {
    return (
        <div>
            {/* 如果样式是多个单词 那么必须使用小驼峰命名把  把-去掉 后面的单词大写
*/}
            <h1 style={{ color: 'red', backgroundColor: 'yellow' }}>设置
行内样式</h1>
        </div>
    )
}

ReactDOM.render(
    <Fun></Fun>,
    document.getElementById('root')
)
```

类组件

```
import React from 'react'
import ReactDOM from 'react-dom'

let text = "你好"
// 类组件
class Demob extends React.Component {
    render() {// 就是渲染的意思  执行这个render他就会执行jsx
        return (
            <div>
                {/* 如果样式是多个单词 那么必须使用小驼峰命名把  把-去掉 后面的单
词大写 */}
                <h1 style={{ color: 'red', backgroundColor: 'yellow' }}>
设置行内样式--{text}</h1>
            </div>
        )
    }
}
```

```
ReactDOM.render(
    <Demob></Demob>,
    document.getElementById('root')
)
```

类样式--className

React推荐我们使用行内样式，因为React觉得每一个组件都是一个独立的整体

其实我们大多数情况下还是大量的在为元素添加类名，但是需要注意的是， class 需要写成 **className** （因为毕竟是在写类js代码，而 class 是js关键字）

```
<p className="hello">Hello world</p>
```

css需要单独的新建一个文件 写入你的css内容

必须要把你写的css和你要使用样式的组件关联起来

使用 import “css的路径”

```
import React, { Component } from 'react'
// 引用样式
import "./classdemo.css"

export default class classdemo extends Component {
  render() {
      let text="我是类组件的变量"
    return (
      <div>
          classdemo--{text}

          <h2 className='xiaoming'>我是类样式</h2>
     </div>
    )
  }
}
```

扩展使用scss

下载 npm install --save sass-loader@10 node-sass@6

直接编写scss文件 然后在需要使用的组件内 使用 import “你的样式文件路径即可”

多行html 空标签

在react的组件中多行html必须有一个父容器包裹 所以通常我们使用div来进行包裹 但是有的时候这些div是多余的 会在页面生成很多无用的代码

```
import React from 'react'
import ReactDOM from 'react-dom'

class Demob extends React.Component {
    render() {
        return (
            // 多行标签必须有一个父容器包裹
            <div>
                <h1>你好我是一个标签</h1>
                <h1>你好我是一个标签</h1>
                <h1>你好我是一个标签</h1>
                <h1>你好我是一个标签</h1>
                <h1>你好我是一个标签</h1>
            </div>

        )
    }
}

ReactDOM.render(
    <Demob></Demob>,
    document.getElementById('root')
)
```

空标签

空标签 在页面是不进行展示的 它的作用仅仅就是用来描述多行标签的一个包裹作用

写法1:

<></>

```
import React from 'react'
import ReactDOM from 'react-dom'

class Demob extends React.Component {
    render() {
        return (
            // 空标签
            <>
                <h1>你好我是一个标签</h1>
                <h1>你好我是一个标签</h1>
                <h1>你好我是一个标签</h1>
```

```
                <h1>你好我是一个标签</h1>
                <h1>你好我是一个标签</h1>
            </>

        )
    }
}

ReactDOM.render(
    <Demob></Demob>,
    document.getElementById('root')
)
```

写法2:

Fragment空标签

```
import React from 'react'
import ReactDOM from 'react-dom'

class Demob extends React.Component {
    render() {
        return (
            // 空标签
            <React.Fragment>
                <h1>你好我是一个标签</h1>
                <h1>你好我是一个标签</h1>
                <h1>你好我是一个标签</h1>
                <h1>你好我是一个标签</h1>
                <h1>你好我是一个标签</h1>
            </React.Fragment>

        )
    }
}

ReactDOM.render(
    <Demob></Demob>,
    document.getElementById('root')
)
```

动态属性

属性插入变量使用：属性={属性值}

```
import React from 'react'
import ReactDOM from 'react-dom'
```

```
class Demob extends React.Component {
    render() {
        let text="点我去百度"
        let ahref="http://www.baidu.com"
        return (
            <>
                <h1>属性插变量</h1>
                {/* react中属性插变量    属性={你要插的值} */}
                <a href={ahref}>{text}</a>
            </>
        )
    }
}

ReactDOM.render(
    <Demob></Demob>,
    document.getElementById('root')
)
```

事件处理

在react中事件的绑定 使用小驼峰命名法

例：onclick 在react中 onClick onchange 在react中 onChange

绑定完事件之后在调用函数的时候不加()不加() 因为加了函数就自动调用了

绑定完事件之后在调用函数的时候不加()不加() 因为加了函数就自动调用了

绑定完事件之后在调用函数的时候不加()不加() 因为加了函数就自动调用了

绑定完事件之后在调用函数的时候不加()不加() 因为加了函数就自动调用了

基本事件操纵

事件绑定 使用小驼峰命名法 鼠标左键点击事件 onclick------» onClick onchange------» onChange

类组件

```
import React from 'react'
import ReactDOM from 'react-dom'

class Demob extends React.Component {
    fun=()=>{
        console.log("你好")
    }
    render() {
        return (
```

```
            <>
                {/* 函数注意 */}
                <button onClick={this.fun}>点我触发函数</button>
            </>
        )
    }
}

ReactDOM.render(
    <Demob></Demob>,
    document.getElementById('root')
)
```

函数组件

```
// 我是函数组件的例子
import React from 'react'
import ReactDOM from 'react-dom'
let demo=()=>{
    console.log("事件")
}

let Fun=()=>{

    return (
        <div>
            <h1>我是一个函数组件</h1>
            <button onClick={demo}>点我调用事件</button>
        </div>
    )
}

ReactDOM.render(
    <Fun></Fun>,
    document.getElementById('root')
)
```

函数参数传递

因为在react中函数调用的时候不加() 那我我们如果要传递函数的实参怎么传递?

1.使用bind方式进行传递

函数组件

```
<button onClick={fun.bind(this,"你好我是实参")}>点我</button>
```

类组件

```
 <button onClick={this.fun.bind(this,"我是实参1","我是实参2")}>点我传递函数实参</button>
```

2.使用箭头函数调用函数进行传递

函数组件

```
<button onClick={()=>{fun("我是参数")}}>点我</button>
```

类组件

```
<button onClick={()=>{this.funb(1111,2222)}}>点我传递实参2</button>
```

阻止事件传播与默认行为

同原生js

事件对象

使用event

```
import React, { Component,Fragment } from 'react'

export default class Com extends Component {
    // 直接传值event即可得到事件对象
    fun=(event)=>{
        console.log("您在输入框中是",event.target.value)
    }
  render() {
    return (
      <Fragment>

          <h1>事件对象</h1>
          <input type="text" onInput={this.fun}/>

      </Fragment>
    )
  }
}
```

遍历展示

react推荐使用map方法来进行数据的便利

```
import React, { Component } from 'react'
let arr = [111, 222, 333, 444, 555, 666, 777];
export default class democ extends Component {
    render() {
        return (
            <React.Fragment>
                <h1>便利数据</h1>
                <ul>

                    {
                        arr.map((v,i)=>{
                            return (
                                <li key={i}>{v}</li>
                            )
                        })
                    }

                </ul>


            </React.Fragment>
        )
    }
}
```

便利复杂数据

```
import React, { Component } from 'react'
let arr = [111, 222, 333, 444, 555, 666, 777];
let obj=[
    {name:"xix1i",age:181},
    {name:"xix2i",age:182},
    {name:"xix3i",age:183},
    {name:"xix4i",age:184},
    {name:"xix5i",age:185},
    {name:"xix6i",age:186}
]
export default class democ extends Component {
    render() {
        return (
            <React.Fragment>
                <h1>便利数据</h1>
                <ul>
```

```
                    {
                        arr.map((v,i)=>{
                            return (
                                <li key={i}>{v}</li>
                            )
                        })
                    }

                </ul>

                <hr />

                <table border="1">
                    <tbody>
                    {
                        obj.map((v,i)=>{
                            return (
                                <tr key={i}>
                                    <td>{v.name}</td>
                                    <td>{v.age}</td>
                                </tr>
                            )
                        })
                    }
                    </tbody>
                </table>



            </React.Fragment>
        )
    }
}
```

ref

函数组件 无状态组件 不能使用**ref**

ref 标识组件内部的元素 就给组件的dom元素起个名字

函数组件是不能直接使用ref的

ref写法的分类

1.字符串(官方已经不推荐使用了)

2.回调函数（官方推荐方式）

回调函数的方式 就是在dom节点上挂载一个函数 函数的入参 就是dom节点

```
import React from 'react'
import ReactDOM from 'react-dom'

class Demob extends React.Component {
    fun=()=>{
        console.log(this.demoinput.value)
    }
    render() {
        return (
            <div>
                <h1>ref的例子</h1>
                {/* 回调函数的方式创建ref */}
                {/* <input type="text" ref={(text这个变量代表的就是当前这个
input标签)=>{this.demoinput(随便创建一个变量)=text}}/> */}
                <input type="text" ref={(text)=>{this.demoinput=text}}/>
                <button onClick={this.fun}>点我得到输入框的值</button>
            </div>
        )
    }
}

ReactDOM.render(
    <Demob></Demob>,
    document.getElementById('root')
)
```

3.React.createRef()（16.8版本新增的 官方推荐）

react 16.8新增的一种方式 我们通过初始化createRef从而得到ref对象 插入到页面中

```
import React from 'react'
import ReactDOM from 'react-dom'

class Demob extends React.Component {
    // 1.创建
    inputref=React.createRef()
    fun=()=>{
        // 3.使用
        console.log(this.inputref.current.value)
    }
    render() {
        return (
```

```
            <div>
                <h1>createRef</h1>
                {/* 2.绑定 */}
                <input type="text" ref={this.inputref}/>
                <button onClick={this.fun}>点我得到输入框的值</button>
            </div>
        )
    }
}

ReactDOM.render(
    <Demob></Demob>,
    document.getElementById('root')
)
```

扩展--插入字符串标签

在vue中插入字符串标签使用v-html这个指令即可完成

在今后实际工作的时候 会出现后台给你返回的就是一大段生成好的html 你需要把他展示在页面上

```
import React, { Component } from 'react'

export default class demod extends Component {
    state={
        newhtml:"<h1>我是h1我是一段字符串标签</h1>"
    }
  render() {
    return (
      <div>
          demod
          {/* 插入字符串标签 */}
         <div dangerouslySetInnerHTML={{__html:this.state.newhtml}}>

         </div>
        </div>
    )
  }
}
```

条件渲染

在开发中 创建不同的组件来封装我们需要完成的各个内容 但是我们需要根据程序的状态变化来渲染其中一部分内容

if语句来看进行条件渲染

在react中if条件渲染是最简单的 但是但是但是但是但是 注意 在jsx不允许出现if

```
import React, { Component } from 'react'

export default class demo extends Component {
    render() {
        let newhtml=""
        let num=1

        if(num==1){
            newhtml= <h1>吃饭</h1>
        }else if(num==2){
            newhtml= <h1>睡觉</h1>
        }else{
            newhtml=<h1>上厕所</h1>
        }

        return (
            <div>
                    {newhtml}

            </div>
        )
    }
}
```

三元运算符

使用图片

1.把图片放到public文件夹中 直接使用图片名

2.不在public下 我们可以使用require()来进行引用

```
<img src={require("../assets/2.webp")} />
```

3.不在public下 我们也可以使用导入式

```
import React, { Component } from 'react'
```

```
// 1.引入图片
import imgA from "../assets/2.webp"
export default class demob extends Component {
    render() {
        return (
            <div>
                {/* <img src="img/1.webp" /> */}



                {/* <img src={require("../assets/2.webp")} /> */}

                {/* 2.使用 */}
                <img src={imgA} />
            </div>
        )
    }
}
```

5 组件的数据挂载方式

状态state

状态（变量/数据）机制 在**react**中我们只需要把数据定义在**state**中 那么数据改变了页面也会发生改变

通过状态就可以让页面内容在不操作dom的情况下 进行相应式的改变

函数组件也叫无状态组件 所以函数组件中不能使用状态 （react16.8新增了一个叫HOOK的技术可以实现现在听听就好）

函数组件也叫无状态组件 所以函数组件中不能使用状态

函数组件也叫无状态组件 所以函数组件中不能使用状态

函数组件也叫无状态组件 所以函数组件中不能使用状态

函数组件也叫无状态组件 所以函数组件中不能使用状态

使用

1.定义状态数据方式1

```
import React, { Component } from 'react'

export default class demob extends Component {
    // 初始化state状态数据需要放到constructor中进行初始化
    // es6中继承的规则中得知 子类是可以不写constructor 他会在实例化的时候
    // 自动补充一个

    // 但是如果你写了 那么必须在其中写super()  因为super就是调用父类的构造方法
    // 此时子类才有this
    constructor() {
        super()
        // 初始化state
        this.state={
            text:"我是字符串",
            num:888,
            bool:true,
            arr:[1111,2222,333],
            obj:{
                name:"xixi"
            }
        }
    }
    render() {
        return (
            <>

                <h1>我是测试state使用的例子</h1>


            </>
        )
    }
}
```

定义状态数据方式2

这种方式也是没有问题的 而且简单 但是 你去公司了 你看看你同事这样子写没有 如果没有你就不要用了

```
import React, { Component } from 'react'

export default class demob extends Component {

     state={
```

```
            text:"我是字符串",
            num:888,
            bool:true,
            arr:[1111,2222,333],
            obj:{
                name:"xixi"
            }
        }

    render() {
        return (
            <>

                <h1>我是测试state使用的例子</h1>


            </>
        )
    }
}
```

2.使用state数据

this.state 是纯js对象,在vue中，data属性是利用 Object.defineProperty 处理过的，更改data的数据的时候会触发数据的 getter 和 setter ，但是React中没有做这样的处理，如果直接更改的话，react是无法得知的，所以，需要使用特殊的更改状态的方法 **setState** 。

```
import React, { Component } from 'react'

export default class demob extends Component {
    // 初始化state状态数据需要放到constructor中进行初始化
    // es6中继承的规则中得知 子类是可以不写constructor 他会在实例化的时候
    // 自动补充一个

    // 但是如果你写了 那么必须在其中写super()  因为super就是调用父类的构造方法
    // 此时子类才有this
    constructor() {
        super()
        // 初始化state
        this.state={
            text:"我是字符串",
            num:888,
            bool:true,
            arr:[1111,2222,333],
            obj:{
                name:"xixi"
            }
```

```
        }
    }
    render() {
        return (
            <>
                {/* 2.使用state数据 */}
                <h1>我是测试state使用的例子----{this.state.num}</h1>


            </>
        )
    }
}
```

3.修改state的数据

this.setState({你修改谁:修改成什么})

修改**state**的数据必须使用**setState**修改之后页面才会发生改变

```
import React, { Component } from 'react'

export default class demob extends Component {
    // 初始化state状态数据需要放到constructor中进行初始化
    // es6中继承的规则中得知 子类是可以不写constructor 他会在实例化的时候
    // 自动补充一个

    // 但是如果你写了 那么必须在其中写super()  因为super就是调用父类的构造方法
    // 此时子类才有this
    constructor() {
        super()
        // 初始化state
        this.state={
            text:"我是字符串",
            num:888,
            bool:true,
            arr:[1111,2222,333],
            obj:{
                name:"xixi"
            }
        }
    }

    fun=()=>{
        // 修改state的数据
        this.setState({
            num:123,
            text:"xixi"
        })
```

```
    }


    render() {
        return (
            <>
                {/* 2.使用state数据 */}
                <h1>我是测试state使用的例子----{this.state.num}---
{this.state.text}</h1>
                <button onClick={this.fun}>点我修改</button>

            </>
        )
    }
}
```

调用了setState之后发生了什么?

setState是异步的

（如果有大量数据修改的话不会因为修改数据而造成程序的卡顿）

```
import React, { Component } from 'react'

export default class demob extends Component {
    // 初始化state状态数据需要放到constructor中进行初始化
    // es6中继承的规则中得知 子类是可以不写constructor 他会在实例化的时候
    // 自动补充一个

    // 但是如果你写了 那么必须在其中写super()  因为super就是调用父类的构造方法
    // 此时子类才有this
    constructor() {
        super()
        // 初始化state
        this.state={
            text:"我是字符串",
            num:888,
            bool:true,
            arr:[1111,2222,333],
            obj:{
                name:"xixi"
            }
        }
    }

    fun=()=>{
        // 修改state的数据
        // this.setState({
```

```
        //      num:123,
        //      text:"xixi"
        // })
        // 下面的console.log打印的结果是修改之后的  还是修改之前的?
        // 是修改之前的结果  所以从而证明了setState是一个异步任务
        // console.log(this.state.num)

        // 但是我就是想setState修改完数据之后  打印新的结果怎么办?
        // 因为setState是异步   异步都会有回调函数
        this.setState({
            num:123,
            text:"xixi"
        },()=>{
            // setState第二个参数是一个回调函数  当数据修改完他才会调用
            console.log(this.state.num)

        })

    }


    render() {
        return (
            <>
                {/* 2.使用state数据 */}
                <h1>我是测试state使用的例子----{this.state.num}---
{this.state.text}</h1>
                <button onClick={this.fun}>点我修改</button>

            </>
        )
    }
}
```

调用了setState之后会自动触发render渲染

render就是渲染方法 只有render方法执行了 那么页面才会根据数据的改变而随之发生改变

render就是渲染方法 只有render方法执行了 那么页面才会根据数据的改变而随之发生改变

render就是渲染方法 只有render方法执行了 那么页面才会根据数据的改变而随之发生改变

render就是渲染方法 只有render方法执行了 那么页面才会根据数据的改变而随之发生改变

扩展-----强制刷新

强制触发render渲染

```
import React, { Component } from 'react'

export default class demob extends Component {
    constructor(){
        super()

      this.xiaoming="你好"
    }
    fun=()=>{
      this.xiaoming="你坏"
      console.log(this.xiaoming)
      // 我们可以强制触发render重新渲染页面
      this.forceUpdate()
    }
  render() {
    return (
      <>
          <h1>强制刷新</h1>
         <h1>{this.xiaoming}</h1>
         <button onClick={this.fun}>点我修改</button>




      </>
    )
  }
}
```

生命周期

1.挂载阶段

```
import React, { Component } from 'react'

export default class index extends Component {
    constructor(){
        super()
        // 1.constructor
        console.log("constructor")
```

```
        this.state={
            xxxxx:'数据'
        }
    }
    // 2componentWillMount
    componentWillMount = () => {
        console.log("创建之前")
    }
    // 4.componentDidMount
    componentDidMount = () => {
      console.log("创建之后")
    }

  render() {
    // 3.render
    console.log("render")
    return (
      <div>
        1111
      </div>
    )
  }
}
```

2.修改阶段

3.销毁阶段

6.组件传值

npm config set registry https://registry.npm.taobao.org

组件默认是一个完整独立的个体。组件与组件之间的数据默认是不能相互使用的

正向传值--props

函数组件

子组件

```
// 1.把props当成函数的形参传入
let Zi=(props)=>{
    return (
        <div>
            {/* 2.使用props来进行数据的展示 */}
            zizizizizizi--{props.title}
        </div>
    )
}

export default Zi
```

父组件进行传递

```
import Zi from "./zi.jsx"

let Fu=()=>{
    return (
        <div>
            FFUFUFUFUFUFUF
            {/* 父组件传递 */}
            <Zi title="我是父组件的书"></Zi>
        </div>
    )
}

export default Fu
```

公司的写法

父组件

```
import Zi from "./zi.jsx"

let Fu=()=>{
    // 定义数据
    let obj={
        title:"你好么么哒！！！！！"
    }
    return (
        <div>
            FFUFUFUFUFUFUF
            {/* 父组件使用扩展运符传递 */}
            <Zi {...obj}></Zi>
        </div>
    )
}

export default Fu
```

子组件

```
import React from 'react'

// 1.形参传入props
export default function Zi(props) {

    let {title}=props

  return (
    <div>
        Zi
        <span>
            {title}
        </span>

    </div>
  )
}
```

类组件

1.子组件 this.props.xxx

```
import React, { Component } from 'react'

export default class zi extends Component {
  render() {
    return (
      <div>
          zi
          {/* 1.子组件定义props */}
          <h1>父组件的数据式-----{this.props.title}</h1>
      </div>
    )
  }
}
```

2.父组件传递

```
import React, { Component } from 'react'
import Zi from "./zi.jsx"
export default class fu extends Component {
  render() {
    return (
```

```
        <div>
            fu
            {/* 父组件给子组件传递数据 */}
          <Zi title="我式父组件的数据"></Zi>
        </div>
      )
    }
}
```

公司写法

子组件中使用结构来优化代码

```
import React, { Component } from 'react'

export default class zi extends Component {
  render() {
    //   使用解构的方式见到了this。props的重复出现率
    let {title,age,name,sex,love}=this.props
    return (
      <div>
          zi
          {/* 1.子组件定义props */}
          <h1>父组件的数据式-----{title}</h1>
          <h1>父组件的数据式-----{age}</h1>
          <h1>父组件的数据式-----{name}</h1>
          <h1>父组件的数据式-----{sex}</h1>
          <h1>父组件的数据式-----{love}</h1>

      </div>
    )
  }
}
```

父组件使用扩展运算符传递数据

```
import React, { Component } from 'react'
import Zi from "./zi.jsx"
export default class fu extends Component {
  render() {
      let obj={
          title:"我是title",
          name:"我是name",
          age:18,
          sex:"男",
          love:"女"
      }
```

```
    return (
      <div>
          fu
          {/* 扩展运算符快速传递数据 */}
        <Zi {...obj}></Zi>
      </div>
    )
  }
}
```

this.props.children

思考

在react组件调用中我们的开标前和关标签中能否插入内容？

不能 因为组件是一个完整的独立的个体 默认不能插入

this.props.children 他表示所有组件的子节点（默认写上没有任何作用 在组件被调用的时候如果我们在他的开关标签中插入dom元素 那么this.props.chilren 就会接收并且显示）

逆向传值--使用props接收一个函数

子组件把数据给父组件

子组件

```
import React, { Component } from 'react'

export default class zi extends Component {
  render() {
    return (
      <div>
          zizizizzizi
          {/* 1.逆向传值必须通过事件来触发 一个父组件传递过来的函数*/}
          <button onClick={this.props.demofun}>点我进行逆向传值</button>
      </div>
    )
  }
}
```

父组件

```
import React, { Component } from 'react'
```

```
import Zi from "./zi.jsx"
export default class fu extends Component {
    fufun=()=>{

    }
  render() {
    return (
      <div>
          fuffufufufuf
          {/* 2.父组件给子组件传递一个函数 */}
          <Zi demofun={this.fufun}></Zi>

      </div>
    )
  }
}
```

子组件

```
import React, { Component } from 'react'

export default class zi extends Component {
  render() {
    return (
      <div>
          zizizizzizi
          {/* 1.逆向传值必须通过事件来触发 一个父组件传递过来的函数*/}
          {/* 3.给函数进行实参的传递 */}
          <button onClick={this.props.demofun.bind(this,"我是子组件的数
据")}>点我进行逆向传值</button>
      </div>
    )
  }
}
```

父组件

```
import React, { Component } from 'react'
import Zi from "./zi.jsx"
export default class fu extends Component {
    // 4.父组件设置形参接收子组件绑定的实参
    fufun=(text)=>{
            console.log("我是父组件的函数",text)
    }
  render() {
    return (
      <div>
```

```
            fuffufufufuf
            {/* 2.父组件给子组件传递一个函数 */}
            <Zi demofun={this.fufun}></Zi>

        </div>
      )
    }
}
```

同胞传值

Pubsub-js

react中默认是不能进行同胞传值的 如果我们要进行 那么必须依赖 **pubsub-js**（是**-js** 千万不要记错了）库来实现

1.npm install --save pubsub-js

2.抛出 在需要传递的组件中使用 Pubsub.publish(“自定义事件名”,"数据") publish创建自定义事件

```
import React, { Component } from 'react'
// 1.引用pubsub-js
import Pubsub from "pubsub-js"
export default class zia extends Component {
    fun=()=>{
        // 2.publish抛出自定义事件
        Pubsub.publish("zia","我是zia的数据么么哒！！！！！")
    }
  render() {
    return (
      <div>zia
          <button onClick={this.fun}>点我把数据传递到zib</button>

      </div>
    )
  }
}
```

3.接收 在需要接收数据的组件中使用Pubsub.subscribe("你监听的事件",()=>{})subscribe监听自定义事件

```
import React, { Component } from 'react'
// 3.引用pubsub
```

```
import Pubsub from "pubsub-js"
export default class zib extends Component {
    constructor(){
        super()
        console.log("1.react初始化数据")
    }
    componentWillMount(){
        console.log("2.在渲染之前调用")
    }
    componentDidMount() {
    //     接收    监听同胞传值的自定义事件
        // 回调函数的第一个形参是你监听的自定义事件的名字
        // 回调函数的第二个形参就是自定义事件上绑定的数据
    Pubsub.subscribe("zia",(a,b)=>{
        console.log(a)
        console.log(b)
    })

     }
  render() {
      console.log("3开始渲染")
    return (
      <div>zib</div>
    )
  }
}
```

状态提升--中间人模式

React中的状态提升概括来说,就是将多个组件需要共享的状态提升到它们最近的父组件

上.在父组件上改变这个状态然后通过props分发给子组件.

跨组件传值

context对象--上下文对象

react 组件间传递数据是通过 props 向下，是单向传递的，从父级一层一层地通过 props 地向下传递到子子孙孙，有的时候我们组件一层一层的嵌套多层，这样这种方式一层一层传递麻烦，如果想跃层传递，这就会用到 context

context：上下文对象

context很好的解决了跨组件传值的复杂度。可以快速的进行跨组件数据的传递。

想要使用context进行跨组件传值那么就要使用createContext()方法同时方法中给我们提供了两个对象：

Provider对象　生产者---->用来生产数据
Consumer对象　消费者---->用来使用数据

```
import React, { Component } from 'react'
// 1.创建context对象
const GlobalContext = React.createContext()

class Zia extends Component {
    render() {
        return (
            <div>
                我是第一个子组件
            </div>
        )
    }
}
class Zib extends Component {
    render() {
        return (

            <div>
                我是第2个子组件
                {/* 3.任意组件引入GlobalContext并调用context，使用
GlobalContext.Consumer（消费者） */}

                <GlobalContext.Consumer>
                    {/* 4.在回调函数中直接使用生产者的数据 */}
                    {

                        (value) => {
                            return <h1>{value.name}</h1>
                        }
                    }
                </GlobalContext.Consumer>
            </div>

        )
    }
}
export default class fu extends Component {
    render() {
        return (
            // 2.在根组件件引入GlobalContext，并使用GlobalContext.Provider生
产者
            // 并且使用value属性传入数据
            <GlobalContext.Provider value={{ name: "xixi", age: 18 }}>
```

```
                <div>
                    我是根组件
                    <Zia />
                    <Zib />
                </div>
            </GlobalContext.Provider>
        )
    }
}
```

redux

redux是什么?

redux就是一个javascript的状态管理工具 可以集中的管理react中多个组件的状态 让我们组件之间数据传递变得非常的简单

redux是一个第三方的 也就是说他可以在**react**中用 也可以在**vue**中进行使用

如果组件需要进行跨层级传值 传统的方式 就是组件一层层的进行逆向传值传递到他们最近的一个父组件身上 然后再一层层的进行正向传值

redux的三大原则

1.单一数据源 ： 整个项目的数据都被存储在一个**store**对象中

2.state是只读的：如果我们想改变**state**的数据 那么必须触发**action**里面的修改动作来执行修改

3.使用纯函数来进行修改：**reducer**就是一个函数 我们通过**reducer**中的**state**和**action**动作来进行数据的修改

redux 中常用属性与方法

createStore: 创建redux对象

reducer：就是一个函数 这个函数中保存的就是我们使用的状态数据 与 修改这个状态数据的动作

getState():读取redux中的数据

subscribe()： 在redux中监听store的数据改变 当store的数据改变了 这个subscribe就会触发

combineReducers（） 就是把多个reducer合并成一个

redux使用

1.下载redux　npm install --save redux

2.在项目的文件夹中创建一个store文件夹（容纳redux的代码）

3.在新建一个文件容纳基本代码

```
// 1.创建redux对象（先引用redux创建方法createStore）
import {createStore} from "redux"
// 6.创建创建state数据
let data={
    name:"xixi",
    age:18,
    sex:"男"
}
// 5.创建reducer 是一个方法  其中保存的就是redux中存储的变量和修改变量的方法
// state就是数据状态
// action 修改上面数据状态的一些动作
// 7.把上面的数据传递给state
let reducer=(state=data,action)=>{
    // 8把state return
    return state
}
// 2.开始创建redux对象
// 4.把状态和修改状态的方法传递到初始化redux中
let store=createStore(reducer)
// 3.暴露redux对象
export default store
```

读取redux中的数据

store.getState().你要读取的数据

```
import React, { Component } from 'react'
// 1.引用redux
import store from "../store/index.js"
export default class demoa extends Component {
    // 2.把数据复制给组件的state中进行保存
    state={
        name:store.getState().name
    }

  render() {
    return (
      <div>
          <h1>redux的基本使用</h1>
          {/* 3.读取 */}
          <h1>使用数据---{this.state.name}</h1>
      </div>
```

```
    )
  }
}
```

基本数据修改

我们需要通过dispatch（）来调用写在action中的修改动作

千万不要和**vuex**搞混了 因为在**vuex**中 **dispatch**触发**action**是进行异步操纵的触发器

但是但是 在**redux**中 **dispatch**触发 的这个**action**里面存储的是修改状态的动作

```
  fun=()=>{
        // 通过dispatch来修改数据
        // store.dispach({type:"你要触发的修改动作"})
        store.dispach({type:"USER_UP_NAME"})
    }
```

编写修改数据的动作

```
import {createStore} from "redux"

let data={
    name:"xixi",
    age:18,
    sex:"男"
}

let reducer=(state=data,action)=>{
    // 创建修改动作
    switch (action.type) {
        case "USER_UP_NAME":
            console.log({...state,name:"我变了"})
            return {...state,name:"我变了"}
            break;

        default:
            return state
            break;
    }
}
let store=createStore(reducer)

export default store
```

但是大家运行后发现 数据修改了 但是页面并没有发生改变

原因很简单 因为 虽然你数据变了 但是组件中的**render**并没有重新执行 那么页面当然不会修改了

subscribe() 监听redux state的状态 改变就会触发

```
import React, { Component } from 'react'
// 1.引用redux
import store from "../store/index.js"
export default class demoa extends Component {
    // 2.把数据复制给组件的state中进行保存
    state={
        name:store.getState().name
    }

    // 我们可以监控这redux中的state数据  如果redux中的数据改变了
    // 我重启读取下并且在触发组件中的render那么 页面的内容就会改变
    componentDidMount() {

        store.subscribe(()=>{
            // 当redux中的state数据改变了 那么subscribe就会触发
            this.setState({
                name:store.getState().name
            })
        })

     }

    fun=()=>{
        // 通过dispatch来修改数据
        // store.dispach({type:"你要触发的修改动作"})
        store.dispatch({type:"USER_UP_NAME"})
    }

  render() {
    return (
      <div>
          <h1>redux的基本使用</h1>
          {/* 3.读取 */}
          <h1>使用数据---{this.state.name}</h1>
          <button onClick={this.fun}>点我修改上面的数据</button>
      </div>
    )
  }
}
```

合并reducer（把redux拆分成一个个的模块）

随着项目的体积越来越大 项目的state和修改的动作也会越来越多

1.新建一个reducer.js（就是一个合并工厂 把今后拆分的一个个的小模块合并起来）

2.新建一个文件夹modules 里面放置我们拆分的一个个的小模块

3.开始拆分 把原来写在一起的state和原来写在一起的动作拆分出来

```
// 里面存放的就是demoa的state数据和demoa的修改动作
let data={
    name:"xixi",

}

let demoam=(state=data,action)=>{
    // 创建修改动作
    switch (action.type) {
        case "USER_UP_NAME":
            console.log({...state,name:"我变了"})
            return {...state,name:"我变了"}
            break;

        default:
            return state
            break;
    }
}

export default demoam
```

4.开始合并reducers.js中进行

```
// reducer合并工厂中吧modules文件夹中多个小模块进行合并
// 1.把你要合并的所有模块引用进来
import demoam from "./modules/demoam.js"
import demobm from "./modules/demobm.js"
// 2.引用合并模块的方法
import {combineReducers} from "redux"
// 3.开始合并
let reducer=combineReducers({
    demoam,
    demobm
})
// 4.暴露
export default reducer
```

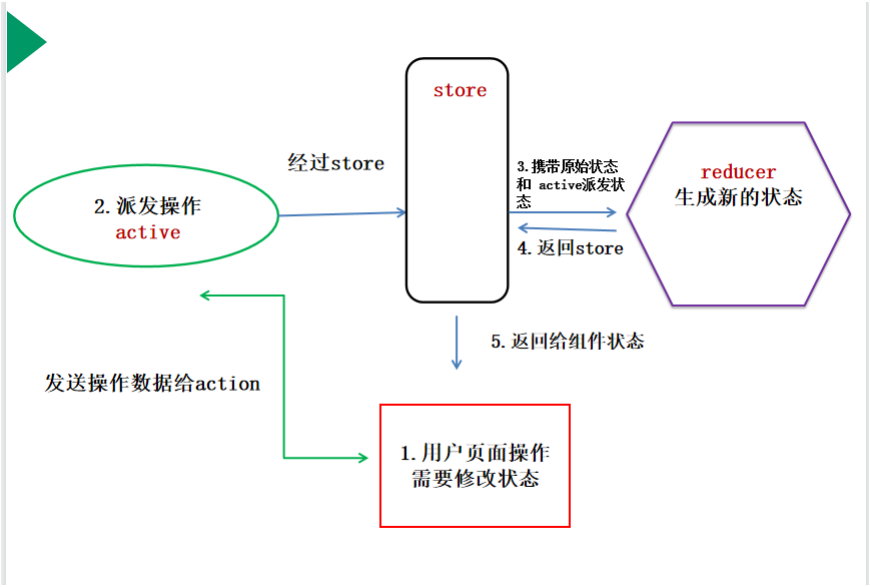
5.把合并好的模块 注入到redux实例中

```
import {createStore} from "redux"

// 引用合并好的reducer
import reducer from "./reducers.js"

let store=createStore(reducer)

export default store
```

大家会发现 我们合并好模块之后 在页面不显示数据了 因为我们把内容都合并成了模块所以要使用的时候

store.getState().模块名.xxxx

redux的数据执行流程



react-redux

react-redux 是一个专门为react开发的状态管理工具 而redux是第三方的

之前redux的写法 和react的耦合度太高 （在react中吧第三方的redux集成在项目里面 会造成代码的可读性太低）

react-redux 就可以简化我们在**react**中使用**redux**的复杂度

使用

1.下载 npm install --save react-redux

2.我们需要在项目的全局组件之上 设置Provider 发布者

```
import React from 'react';
import ReactDOM from 'react-dom';
```

```
import App from "./components/demob.jsx"

// 1.引用provider
import { Provider } from "react-redux"
import store from "./store/index.js"
ReactDOM.render(
  // 2.使用provider把redux对象传递到所有的子组件身上
  // 3.传入store对象
  <Provider store={store}>
    <App />
  </Provider>,

  document.getElementById('root')
);
```

2.设置组件与redux的连接

```
import React, { Component } from 'react'
// 1.引用react-redux给我们提供的连接方法
// connect是一个函数  当这个函数被调用的时候就是一个高阶组件
import {connect} from "react-redux"
class demob extends Component {
    add=()=>{

    }
    del=()=>{

    }
  render() {
    return (
      <div>
          demob
          <h1>读取redux存储的age</h1>
          <button onClick={this.add}>点我+1</button>
          <button onClick={this.del}>点我-1</button>
    </div>
    )
  }
}
// connect是一个函数  当这个函数被调用的时候就是一个高阶组件
// 第一个() 就是调用函数的语法
// 第二个() 就是高阶组件的语法
export default connect()(demob)
```

3.得到redux中的数据

```
import React, { Component } from 'react'
// 1.引用react-redux给我们提供的连接方法
// connect是一个函数  当这个函数被调用的时候就是一个高阶组件
import {connect} from "react-redux"
class demob extends Component {
    add=()=>{

    }
    del=()=>{

    }
  render() {
    return (
      <div>
          demob
          {/* 使用react-redux读取数据 this.props.state.模块名.xxx */}
          <h1>读取redux存储的age--{this.props.state.demobm.age}</h1>
          <button onClick={this.add}>点我+1</button>
          <button onClick={this.del}>点我-1</button>
    </div>
    )
  }
}
// connect是一个函数  当这个函数被调用的时候就是一个高阶组件
// 第一个() 就是调用函数的语法
// 第二个() 就是高阶组件的语法
// 形参的state今后就是redux中的数据
export default connect(state=>({state}))(demob)
```

修改

在组件中调用dispatch就可以直接完成修改操作

```
    add=()=>{
        // react-redux修改数据
        // 还是使用dispatch触发action的动作
        this.props.dispatch({type:"AGE_NUMBER_ADD_ONE",num:3})
    }
```

7 表单中的受控组件与非受控组件

非受控组件

受控组件

HOC高阶组件

在React组件的构建过程中，常常有这样的场景，有一类功能需要被不同的组件公用。（vue中使用mixins）

HOC不仅仅是一个方法，确切说应该是一个组件工厂，获取低阶组件（功能较少），生成高阶组件（添加多个组件复用的功能）

HOC--参数是组件同时返回值也是组件

高阶组件（HOC）是 React 中用于重用组件逻辑的高级技术。 HOC 本身不是 React API 的一部分。 它们是从 React 构思本质中浮现出来的一种模式。

例子：

比如在多个函数中我们都想使用一个相同的方法。这个方法就是自动发出一段异步请求 并且把请求成功的数据赋值给一个state在页面展示。

那么传统方式我们就需要把这个请求在每个组件中都是用一次 组件少了可以。但是组件变多就很麻烦 那么我们就需要想办法解决这个重复的代码 所以我们可以使用HOC来进行组件中公共内容的复用

大家发现下面的例子 这个请求可能会在多个组件都要使用 那么我们可以把他设置成一个HOC

```
import axios from 'axios'
import React, { Component } from 'react'

export default class two extends Component {
    state={
        text:""
    }
//    发送请求
   async componentDidMount (){
      let {data}= await axios({url:"/api/data/cityinfo/101320101.html"})
    //   把数据赋值给state 并且在下方展示
      this.setState({
          text:data.weatherinfo.city
      })
    }
```

```
  render() {
    return (
      <div>
        {this.state.text}
      </div>
    )
  }
}
```

HOC创建

1.新建withxxx文件 用来容纳HOC（高阶组件明明规则 是withxxx 比如withRouter）

```
// HOC本质就是一个函数
export default ()=>{

}
```

2.使用HOC 在需要服用的组件中把刚才封装的HOC引用使用

```
import axios from 'axios'
import React, { Component } from 'react'
// 1.引用高阶组件
import withLink from "./withLink.js"
// 2.修改暴漏在最下面
class two extends Component {
    state={
        text:""
    }

   async componentDidMount (){
      let {data}= await axios({url:"/api/data/cityinfo/101320101.html"})

      this.setState({
          text:data.weatherinfo.city
      })
    }

  render() {
    return (
      <div>
        {this.state.text}
      </div>
    )
  }
}
```

```
// 3.把当前组件当成函数的实参传入进去（因为高阶组件参数是一个组件）
export default withLink(two)
```

3.在高阶组件中设置行参接受传递进来的组件并且返回一个组件

```
// HOC本质就是一个函数

import React from "react"

// 设置行参接受（首字母要大写）
export default (Component)=>{
    // 并且返回值函数一个组件(组件可以是函数组件也可以是类组件)
    return class withLinkComponent extends React.Component{

        render(){
            return (
                <>

                </>
            )
        }
    }
}
```

4.移植逻辑

```
// HOC本质就是一个函数

import React from "react"
import axios from "axios"

// 设置行参接受（首字母要大写）
export default (Component) => {
    // 并且返回值函数一个组件(组件可以是函数组件也可以是类组件)
    return class withLinkComponent extends React.Component {

        // 把之前的逻辑放置进来
        state = {
            text: ""
        }

        async componentDidMount() {
            let { data } = await axios({ url:
"/api/data/cityinfo/101320101.html" })

            this.setState({
                text: data.weatherinfo.city
            })
```

```
        }
        // 把之前的逻辑放置进来

        render() {
            return (
                <>
                    {/* 把数据传递出去 */}
                    <Component {...this.state}></Component>
                </>
            )
        }
    }
}
```

组件中使用

```
import axios from 'axios'
import React, { Component } from 'react'
// 1.引用高阶组件
import withLink from "./withLink.js"
// 2.修改暴漏在最下面
class two extends Component {


  render() {
    return (
      <div>
          {/* 组件内使用高阶组件的数据 */}
        {this.props.text}
      </div>
    )
  }
}
// 3.把当前组件当成函数的实参传入进去（因为高阶组件参数是一个组件）
export default withLink(two)
```

高阶组件---反向继承

HOC的反向继承 说的简单点就是渲染劫持 在使用HOC的时候可以进行一个高阶组件中的条件渲染

8.路由

根据url的不同来切换对应的组件页面

路由可以实现spa单页面应用　一个项目只有一个完整的页面 我们通过切换页面的显示内容已达到不刷新页面进行切换的效果

路由分类

react-router库

仅仅只包含了路由最基本的功能没有一些辅助的api 但是他轻量级

react-router-dom库

除了基本的路由功能以外 还有很多便捷性的api方便我们开发者实现路由功能

```
npm install --save react-router-dom@5
```

路由模式

HashRouter

带#　不会丢失

BrowerRouter

不带#　上线刷新404

实现

1.下载　npm install --save react-router-dom@5

2.设置路由模式 index.js中设置

```
import React from 'react';
import ReactDOM from 'react-dom';

// 1.引用路由模式
import {HashRouter} from "react-router-dom"

ReactDOM.render(
  // 2.设置路由模式  包裹根组件
  <HashRouter>
    <xxxx></xxxx>
  </HashRouter>,

document.getElementById("root"))
```

3.开始设置路由页面 写在views或者pages

4.设置路由规则与出口 新建router文件夹在新建index.js

```
import React, { Component } from 'react'
// 1.把你要用的路由页面进行引用
import Home from "../views/home.jsx"
import Phone from "../views/phone.jsx"
import User from "../views/user.jsx"
import Shop from "../views/shop.jsx"
// 2-1 引用Route
import {Route} from "react-router-dom"

export default class index extends Component {
  render() {
    return (
      <>
        {/* 路由规则与路由出口 */}
        {/* <Route path="/你的路径" component={你要引用的组件}/> */}
        {/* 2-2.配置出口与规则 */}
        <Route path="/home" component={Home}/>
        <Route path="/shop" component={Shop}/>
        <Route path="/user" component={User}/>
        <Route path="/phone" component={Phone}/>


      </>
    )
  }
}
```

5 设置路由配置组件为根组件index.js中

```
import React from 'react';
```

```
import ReactDOM from 'react-dom';
// 引用路由配置组件
import Index from "./router/index.js"

// 1.引用路由模式
import {HashRouter} from "react-router-dom"

ReactDOM.render(
  // 2.设置路由模式  包裹根组件
  <HashRouter>
    {/* 注入路由组件 */}
    <Index></Index>
  </HashRouter>,

document.getElementById("root"))
```

路由导航

声明式

<关键字 to="去哪里"></关键字>

Link 就是一个组基本的路由导航

NavLink 处理基本路由跳转的功能以外 还添加了自动选中类名的设置 active类名

但是 如果这个active的类名 已经存在了怎么办?

修改navlink的选中类名 activeClassName="你要修改的类名"

注意

有的同学**navlink**可能在电脑上不加类名 如果出现这种问题 那么就不要用**vscode**内置的**cmd**打开项目 而是用外部的**cmd**启动项目即可解决

编程试

push方法在路由页面中跳转 this.props.history.push("/xxxx")

常见问题

如果编程式导航中跳转的话 那么会出现 **push of undefined**的错误

原因:

是因为编程式导航只能在被路由所管理的页面中进行使用（被管理的页面是指 在路由规则中配置过的页面）因为不是被路由所管理的页面就不具备路由所提供的 三个属性(location match history)

所以就不能使用this.props.history 所以就会报错

解决方式：

使用withRouter 高阶组件（HOC）来解决 因为通过withRouter 可以让不是理由所跳转的页面也具有路由的三个属性

使用：

1.引用withRouter

```
import {withRouter} from "react-router-dom"
```

2.修改组件的export default 到最下面

3.使用withRouter来设置当前组件

```
export default withRouter(当前组件)
```

更多跳转方式

replace() 替换当前路径

goBack()后退

goForward()前进

二级多级路由

1.编写二级路由页面

2.配置规则与出口 只需要在对应的一级路由页面中进行route的配置

```
import React, { Component } from 'react'

import {Route,NavLink} from "react-router-dom"

import Era from "./er/era.jsx"
import Erc from "./er/erc.jsx"
export default class phone extends Component {
  render() {
    return (
      <div>
          phone
          <NavLink to="/phone/era">era</NavLink>
          <NavLink to="/phone/erc">erc</NavLink>

          {/* 配置二级路由规则与出口 */}
          <Route path="/phone/era" component={Era}/>
          <Route path="/phone/erc" component={Erc}/>
        </div>
    )
```

```
    }
}
```

404页面

1.创建页面

2.配置404页面规则

```
<>
      {/* 路由导航的组件 */}
       <Bb></Bb>
       {/* 路由规则与路由出口 */}
       {/* <Route path="/你的路径" component={你要引用的组件}/> */}
       {/* 2-2.配置出口与规则 */}
       <Route path="/home" component={Home}/>
       <Route path="/shop" component={Shop}/>
       <Route path="/user" component={User}/>
       <Route path="/phone" component={Phone}/>


       {/* 404页面必须放在最下面 */}
       <Route component={No}/>

    </>
```

大家会发现当我们加入了404页面之后 每个页面都会把404显示出来 因为reacrt的路由在匹配到之后 还会一直向下进行 直到最后

switch 唯一渲染

用switch包裹的内容会渲染上之后不继续向下进行 只会匹配到第一个匹配的内容

```
{/* 唯一渲染 */}
       <Switch>
           {/* 2-2.配置出口与规则 */}
           <Route path="/home" component={Home}/>
           <Route path="/shop" component={Shop}/>
           <Route path="/user" component={User}/>
           <Route path="/phone" component={Phone}/>


           {/* 404页面必须放在最下面 */}
           <Route component={No}/>
       </Switch>
```

重定向与精准匹配

重新定位方向

```
{/* 唯一渲染 */}
        <Switch>
            {/* 2-2.配置出口与规则 */}
            <Route path="/home" component={Home}/>
            <Route path="/shop" component={Shop}/>
            <Route path="/user" component={User}/>
            <Route path="/phone" component={Phone}/>

            {/* 重定向 */}
            {/* 精准匹配   exact */}
            <Redirect from="/" to="/home" exact/>
            {/* 404页面必须放在最下面 */}
            <Route component={No}/>
        </Switch>
```

路由传参

params方式

1.配置路径

```
  <Route path="/home/:xxx" component={Home}>
```

2.发送

声明式

```
  <NavLink to="/phone/数据"></NavLink>
```

编程式

```
  this.props.history,push("/phone/数据")
```

3.接收

```
this.props.match.params.xxx
```

优势： 刷新地址栏，参数依然存在

缺点： 只能传字符串，并且，如果传的值太多的话，url会变得长而丑陋。

query方式

1.发送

```
<Link to={{ pathname : '/d' , query : { name : 'sunny' }}}>点我去d</Link>
```

2.接受

```
this.props.location.query.xxx
```

优势：传参优雅，传递参数可传对象；

缺点：刷新地址栏，参数丢失

state方式

1.发送

```
<Link to={{ pathname : '/d' , state: { name : 'sunny' }}}>点我去d</Link>
```

2.接受

```
this.props.location.state.name
```

优势：传参优雅地址栏不显示传递的数据，传递参数可传对象；

缺点：刷新地址栏，参数丢失

路由拦截

render调用一个函数那么我们就可以决定什么时候渲染他 同时传入props那么就可以在路由组件中使用history: {...}, location: {...}, match: {...}这几个对象

```
    <Route path="/home" render={(props)=>{return <Home {...props}/>}}>
```

9.数据请求

扩展--数据请求在那个地方自动发送

在**react16x**之后**componentWillMount**会有一个问题 就是可能会被执行多次 所以最好不要在当前钩子中进行数据请求的发送

最好在**componentDidMount**中进行数据请求的发送

axios

同vue

原生jquery等等方式

1.下载jquery npm install --save jquery

```
import React, { Component } from 'react'
// 1.引用
import $ from "jquery"
export default class democ extends Component {
    componentDidMount(){
        $.ajax({
            url:"http://localhost:8888/one",
            type:"GET",
            success(ok){
                console.log(ok)
            }
        })
    }
  render() {
    return (
      <div>democ</div>
    )
  }
}
```

fetch Es提供的最新方式

fetch 最新的前后台异步数据交互的方式　传统的方式 axios或者是原生 jqueryajax 都是基于XMLHttpRequest方法来进行实现的　但是fetch不是因为Es最新规范中给我们提供了fetchAPI通过这个fetch

Api来进行前后台的数据交互

```
import React, { Component } from 'react'

export default class demob extends Component {
    componentWillMount(){
        console.log("will")
    }
    componentDidMount() {
        console.log("did")
        // fetch使用
        fetch("http://localhost:8888/one")
```

```
        // 把数据转换成json
        .then(ok=>ok.json())
        .then((ok)=>{
            console.log(ok)
        })

    }
  render() {
    return (
      <div>demob</div>
    )
  }
}
```

其他的方式

```
import React, { Component } from 'react'

export default class demob extends Component {
    componentWillMount(){
        console.log("will")
    }
    componentDidMount() {
        console.log("did")
        // fetch使用
        fetch("http://localhost:8888/one")
        // 把数据转换成json
        .then(ok=>ok.json())
        .then((ok)=>{
            console.log(ok)
        })

        // 发送数据get
        fetch("url/?key=val&key=val",{method:"GET"})
        // 把数据转换成json
        .then(ok=>ok.json())
        .then((ok)=>{
            console.log(ok)
        })

        // 发送post
        fetch(
            "url",{
                method:"POST",
                body:"key=val&key=val"
            }
        )
        // 把数据转换成json
```

```
        .then(ok=>ok.json())
        .then((ok)=>{
            console.log(ok)
        })

    }
  render() {
    return (
      <div>demob</div>
    )
  }
}
```

fetch VS axios VS ajax区别

1.传统的ajax 就是值使用XMLHttpRequest方法实现的数据请求 他隶属于原生的js 核心就是XMLHttpRequest对象 如果多个请求有先后顺序的话 那么容易造成回调地狱问题

jqueryajax 就是对原生XMLHttpRequest封装

2.axios 是基于promise封装的 本质上还是XMLHttpRequest的封装 只不过他是基于最新的语法进行封装的

3.fetch 就是原生js最新标准 和XMLHttpRequest没有半点关系

跨域

方式1

同vue一样也是对devServer进行代理跨域 但是唯一的不同的写跨域的文件不一样

需要到node_modules/react-scripts/config/webpackDevServer.config.js中进行配置

找到proxy 进行设置

```
proxy:{
        "/api（可以随便写）":{
            target:"请求地址",
            changeOrigin:true,
            "pathRewrite":{
              "^/api（和上面一样）":"/"
            }
        }
    },
```

但是不要忘了修改请求的地址为/api　与重启项目

方式2:http-proxy-middleware

1.下载：npm install http-proxy-middleware --save

2.在项目的**src**路径下创建setupProxy.js

3.写入如下内容

```
const { createProxyMiddleware } = require('http-proxy-middleware');

module.exports = function(app) {
  app.use(
    '/api',
    createProxyMiddleware({
      target: 'http://www.weather.com.cn/',
      changeOrigin: true,
      pathRewrite:{
        "^/api":"/"
      }
    })
  );
};
```

json-server模拟数据

在项目中用来进行模拟数据的

1.下载　npm install -g json-server

2.查看版本 json-server --version

3.创建模拟数据文件夹mock 在其中写入数据的json文件

```
{
    "one":[
        {"name":"xixi","age":18}
    ]
}
```

4.启动

（4-1）cd到mock文件夹下

（4-2）json-server --watch 你的json文件名字 --port 你的端口

疑问 上面这个名字太难记了 我怎么修改他?

1.cd到mock文件夹下 npm init -y 初始化一个package.json 文件

2。到package.json文件中 在scripts节点中设置你的启动命令

```
{
  "name": "mock",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
      我是配置启动命令的
    "xiaoming":"json-server --watch data.json --port 8877",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

3. 启动 npm run 你的配置名字即可

扩展---弹射eject

大家发现刚才跨域的方式路径非常深 麻烦 我们有没有办法把这些配置文件放到合适好找的地方

就可以使用弹射来完成（把层级很深的配置文件 弹射到根路径下）

注意：去公司没事别弹 弹之前 问清楚经理 可以你在弹 因为弹射是不可逆的 弹出来就回不去了

注意：去公司没事别弹 弹之前 问清楚经理 可以你在弹 因为弹射是不可逆的 弹出来就回不去了

注意：去公司没事别弹 弹之前 问清楚经理 可以你在弹 因为弹射是不可逆的 弹出来就回不去了

注意：去公司没事别弹 弹之前 问清楚经理 可以你在弹 因为弹射是不可逆的 弹出来就回不去了

注意：去公司没事别弹 弹之前 问清楚经理 可以你在弹 因为弹射是不可逆的 弹出来就回不去了

注意：去公司没事别弹 弹之前 问清楚经理 可以你在弹 因为弹射是不可逆的 弹出来就回不去了

第一次弹射可能会出错 所以你在git add 在commit提交一次

使用npm run eject 进行弹射

就会发现项目的根路径下 有了那么配置文件

10 HOOK

react HOOK 是react16.8新增的一个特性 主要作用 就是让无状态组件/函数组件 可以使用状态 **ref**等一些特性

HOOK 不能在 class组件中使用

类组件与函数组件的区别

无状态组件 的创建比类组件的可读性更好 就是大大减少了组件编写代码量 在组件内容只有一个jsx编写效率更高

函数组件中不用this 因为函数组件没有实例化的过程

useState

useState 是reactHOOK给我们提供的 最基本最常用的一个HOOK 主要作用就是用来管理当前本地的状态

useState() 返回值是一个数组（长度为2）数组的第一项标识的是当前的值 数组的第二项标识的时候修改这个值的函数

let [xiaoming , setXiaoming]=useState(初始值)

创建与读取

```
import {useState} from "react"
let Funcom=()=>{
    // 使用useState()c创建函数组件的状态
    let [xiaoming,setxiaoming]=useState("你好么么哒!!!")
    return (
        <div>
            {/* 读取 */}
            <h1>我是一个函数组件--{xiaoming}</h1>
        </div>
    )
}
export default Funcom
```

修改

```
import {useState} from "react"
let Funcom=()=>{
    // 使用useState()创建函数组件的状态
    let [xiaoming,setxiaoming]=useState("你好么么哒!!!")
    return (
        <div>
            {/* 读取 */}
            <h1>我是一个函数组件--{xiaoming}</h1>
            {/* 修改数据 */}
            <button onClick={()=>{setxiaoming(xiaoming="你好呵呵哒")}}>点我修改</button>
        </div>
    )
}
export default Funcom
```

创建多个状态呢

1.你写多个**useState**

```
import {useState} from "react"
let Funcom=()=>{
    // 1.你写多个useState了解
    let [xiaoming,setxiaoming]=useState("1")
    let [xiaohong,setxiaohong]=useState("2")

    return (
        <div>
          {xiaoming}---{xiaohong}
        </div>
    )
}
export default Funcom
```

2.一次行创建多个值

```
import {useState} from "react"
let Funcom=()=>{
    // 1.你写多个useState了解
    // let [xiaoming,setxiaoming]=useState("1")
    // let [xiaohong,setxiaohong]=useState("2")

    // 2.一次性创建多个值
        let [xiaoming,setxiaoming]=useState({
            dataa:"第一个值1",
            datab:"第一个值2",
            datac:"第一个值3",
            datad:"第一个值4",
            datae:"第一个值5",
            dataf:"第一个值6"
        })
    return (
        <div>
          {/* {xiaoming}---{xiaohong} */}

          {xiaoming.dataa}----{xiaoming.datad}
        </div>
    )
}
export default Funcom
```

一次性创建多个值怎么修改

1.多个useState 要修改的话就依次调用其中的修改方法

2.一次行创建多个值的方式如何修改呢

```
import {useState} from "react"
let Funcom=()=>{
    // 1.你写多个useState了解
    // let [xiaoming,setxiaoming]=useState("1")
    // let [xiaohong,setxiaohong]=useState("2")

    // 2.一次性创建多个值
        let [xiaoming,setxiaoming]=useState({
            dataa:"第一个值1",
            datab:"第一个值2",
            datac:"第一个值3",
            datad:"第一个值4",
            datae:"第一个值5",
            dataf:"第一个值6"
```

```
        })

        let fun=()=>{
        //     一次性创建多个的修改操作     不要忘了保留原始数据
        setxiaoming({...xiaoming,datad:"我被改了"})
        }

    return (
        <div>
          {/* {xiaoming}---{xiaohong} */}

          {xiaoming.dataa}----{xiaoming.datad}
          <button onClick={fun}>点我修改</button>
        </div>
    )
}
export default Funcom
```

useRef

就是可以让函数组件使用ref的一个技术

```
import {useRef} from "react"
let Funcom=()=>{
//    1.创建出useRef
    let xiaoming=useRef(null)

    let fun=()=>{
        // 3.获取
        console.log(xiaoming.current.value);
    }
    return (
        <div>
            {/* 2.绑定使用 */}
          <input type="text" ref={xiaoming}/>
          <button onClick={fun}>点我得到输入框的值</button>
        </div>
    )
}
export default Funcom
```

useEffect

Function Component 不存在生命周期，所以不要把 *Class Component* 的生命周期概念搬过来试图对号入座。

useEffect就可以让函数组件使用生命周期

语法：

```
useEffect(第一个参数是一个函数,第二个参数是一个数组)
```

就是如下三个钩子函数的综合体

```
// 渲染完毕

componentDidMount() {



}

// 修改完毕

componentDidUpdate() {



}

// 准备销毁

componentWillUnmount() {



}
```

模拟componentDidMount

useEffect(()=>{console.log('第一次渲染时调用')},[])

第二个参数为一个空数组，可以模拟**componentDidMount**

因为函数组件每次更新的时候就是组件会被全部调用一次 传递了空数组就相当于欺骗了**react** 我要监听某个内容 所以只是第一次调用

```
import {useEffect} from "react"
let Funcom=()=>{
    useEffect(()=>{
        document.title="么么哒"
        console.log("我自动执行了");
    },[])
    return (
        <div>
```

```
            <h1>函数组件可以使用生命周期</h1>
        </div>
    )
}
export default Funcom
```

模拟componentDidUpdate

没有第二个参数代表监听所有的属性更新但是注意首次也会触发

因为函数组件每次更新的时候就是组件会被全部调用一次 什么都不传那么每次修改就会触发

```
useEffect(()=>{console.log('任意属性该改变')})
```

```
import React, { useState, useEffect } from 'react'

export default function Demo() {
    let [xiaoming, setXiaoming] = useState("666")
    useEffect(() => {
        console.log("谁修改我都会触发")
    })
    let fun = () => {
        setXiaoming("我变了")
    }
    return (
        <>
            <div>demo--{xiaoming}</div>
            <button onClick={fun}>点我修改</button>
        </>
    )
}
```

监听多个属性的变化需要将属性作为数组传入第二个参数。

因为函数组件每次更新的时候就是组件会被全部调用一次 传递了值就会告诉**react**我要监听某个内容 所以只有这些值改变的时候就会被调用

```
useEffect(()=>{console.log('n变了')},[n,m])
```

```
import React, { useState, useEffect } from 'react'

export default function Demo() {
    let [xiaoming, setXiaoming] = useState("666")
    let [xiaohong, setXiaohong] = useState("777")
    // 因为监听了xiaohong 所以修改小明的时候不会触发
```

```
    useEffect(() => {
        console.log("谁修改我都会触发")
    },[xiaohong])
    let fun = () => {
        setXiaoming("我变了")
    }
    return (
        <>
            <div>demo--{xiaoming}--{xiaohong}</div>
            <button onClick={fun}>点我修改</button>
        </>
    )
}
```

模拟componentWillUnmount

通常，组件卸载时需要清除 effect 创建的诸如订阅或计时器 ID 等资源

useEffect函数返回的函数可以表示组件死亡

```
import React,{useState,useEffect} from 'react'

function Zi() {
    useEffect(()=>{
        // 调用子组件的时候执行一个定时函数
       let time= setInterval(()=>{
            console.log("你好")
        },1000)
        // 在effect中return 一个函数就是在销毁的时候执行
        return ()=>{
            console.log("我被销毁了")
            clearInterval(time)
        }
    })
    return (
      <div>我是zi组件</div>
    )
  }

export default function Demo() {
    let [bool,setbool]=useState(true)
  return (
    <div>
        我是父组件
        <button onClick={()=>{setbool(!bool)}}>点我显示和隐藏子组件</button>
        {bool&& <Zi></Zi>}
```

```
    </div>
  )
}
```

useContext

接收一个 context 对象（React.createContext 的返回值）并返回该 context 的当前值。

可以直接获取到**context**对象的**Consumer**消费者中的数据

在没有使用的时候跨组件传值

```
import React, { createContext } from 'react'
let context = createContext()
function Sun() {
    return (
        <div>
            孙
            <context.Consumer>
                {
                    (value)=>{
                        return (
                            <h1>{value.name}</h1>
                        )
                    }
                }
            </context.Consumer>
        </div>
    )
}

function Zi() {
    return (
        <div>
            子
            <Sun></Sun>
        </div>
    )
}

export default function Fu() {
    return (
        <context.Provider value={{name:"xixi",age:18}}>
            <div>
                父
                <Zi></Zi>
            </div>
```

```
            </context.Provider>
        )
    }
```

使用useContext

```
import React, { createContext,useContext } from 'react'
let context = createContext()
function Sun() {
    // 使用useContext可以直接得到Consumer中的数据
    let value=useContext(context)
    return (
        <div>
            孙
            <h1>{value.name}</h1>
        </div>
    )
}

function Zi() {
    return (
        <div>
            子
            <Sun></Sun>
        </div>
    )
}

export default function Fu() {
    return (
        <context.Provider value={{name:"xixi",age:18}}>
            <div>
                父
                <Zi></Zi>
            </div>
        </context.Provider>
    )
}
```

useCallback

思考下面的代码

我们直到当修改数据的时候这个函数是会被重新调用的 那么为什么每次++的时候会从上一次的结果加1

而不是从原始数据1开始计算呢?

所以可以理解为**useState**是一个记忆函数可以记住上次的状态

```
import { useState } from "react"
let Funcom = () => {
    let [xiaoming, setxiaoming] = useState(1)
    return (
        <div>
            <h1>{xiaoming}</h1>
            <button onClick={() => { setxiaoming(++xiaoming) }}>点我修改
</button>
        </div>
    )
}
export default Funcom
```

useReducer

就是在react中 函数组件如果对一个state有多次修改 那么可以使用useReducer来对其内容进行简化

语法:

let [保存这个值得变量，是对这个变量修改的一个方法]=useReducer（修改数据的reducer方法,"初始化值"）

```
import {useReducer} from "react"
let Funcom=()=>{
    // 因为useReducer是对一个数据需要有多次修改的时候使用的
    // state就是当前的这个状态
    // dispat触发修改操作

    let [state,dispatch]=useReducer(reducer,18)
    function reducer(state,action){
        switch (action.type) {
            case "ADD":
                return state+1
                break;
            case "DEL":
```

```
                return state-1
                break;

            default:
                return state
                break;
        }
    }
    var add=()=>{
        dispatch({type:"ADD"})
    }
    var del=()=>{
        dispatch({type:"DEL"})
    }

    return (
        <div>

            <h1>useReducer--{state}</h1>
            <button onClick={add}>+1</button>
            <button onClick={del}>-1</button>
        </div>
    )
}

export default Funcom
```

或者

```
import {useReducer} from "react"
let Funcom=()=>{
    // 因为useReducer是对一个数据需要有多次修改的时候使用的
    // state就是当前的这个状态
    // dispat触发修改操作
    var reducer=(state,action)=>{
        switch (action.type) {
            case "ADD":
                return state+1
                break;
            case "DEL":
                return state-1
                break;

            default:
                return state
                break;
        }
    }
    let [state,dispatch]=useReducer(reducer,18)
```

```
    var add=()=>{
        dispatch({type:"ADD"})
    }
    var del=()=>{
        dispatch({type:"DEL"})
    }

    return (
        <div>

            <h1>useReducer--{state}</h1>
            <button onClick={add}>+1</button>
            <button onClick={del}>-1</button>
        </div>
    )
}

export default Funcom
```

#

全球新闻发布系统

1.项目前期准备

项目初始化

1 create-react-app 项目名 创建项目

2.在项目中删除没有的初始化文件 并且新建index.js（全局配置文件）与app.js

index.js 全局配置文件

```
import React from 'react'
import ReactDOM from 'react-dom'

import App from "./app.js"
ReactDOM.render(
    <App></App>,
    document.getElementById('root')
)
```

app.js 根组件

```
let App=()=>{
    return (
        <>
            我是根组件
        </>
    )
}
export default App
```

css模块化

react中使用普通的css样式表会造成作用域的冲突，css定义的样式的作用域是全局，在Vue中我们还可以使用scope来定义作用域，但是在react中并没有指令一说，所以只能另辟蹊径了。 所以我们可以使用css模块化来解决这个问题

在传统代码中如果有两个组件 并且有相同的类名 在渲染组件的时候样式就会被污染

css模块化

1.创建xxx.module.css文件用来编写css （不要用标签选择器 尽量用class选择器）写入你的样式

```
.demoh{
    color:yellow;
}
```

2.在组件中引用css 并且使用

```
import React, { Component } from 'react'
// 1.引用
import style from "./demo.module.css"
export default class demo extends Component {
  render() {
    return (
      <>
      {/* 2.使用 */}
        <h2 className={style.demoh}>我是第一个组件</h2>
      </>
    )
  }
}
```

项目中使用scss与反向代理

scss

如果在项目上面直接使用scss（尝试修改css文件）

项目会报 如下错误

```
Compiled with problems:X

ERROR in ./src/demo.module.scss (./node_modules/_css-loader@6.7.1@css-
loader/dist/cjs.js??
ruleSet[1].rules[1].oneOf[8].use[1]!./node_modules/_postcss-
loader@6.2.1@postcss-loader/dist/cjs.js??
ruleSet[1].rules[1].oneOf[8].use[2]!./node_modules/_resolve-url-
loader@4.0.0@resolve-url-loader/index.js??
ruleSet[1].rules[1].oneOf[8].use[3]!./node_modules/_sass-
loader@12.6.0@sass-loader/dist/cjs.js??
ruleSet[1].rules[1].oneOf[8].use[4]!./src/demo.module.scss)

Module Error (from ./node_modules/_sass-loader@12.6.0@sass-
loader/dist/cjs.js):
Cannot find module 'sass'
Require stack:
- C:\Users\54407\Desktop\react全家桶\myapp\node_modules\_sass-
loader@12.6.0@sass-loader\dist\utils.js
- C:\Users\54407\Desktop\react全家桶\myapp\node_modules\_sass-
loader@12.6.0@sass-loader\dist\index.js
- C:\Users\54407\Desktop\react全家桶\myapp\node_modules\_sass-
loader@12.6.0@sass-loader\dist\cjs.js
- C:\Users\54407\Desktop\react全家桶\myapp\node_modules\_loader-
runner@4.2.0@loader-
```

所以我们下载scss模块

cnpm install --save sass

即可在项目中安装sass

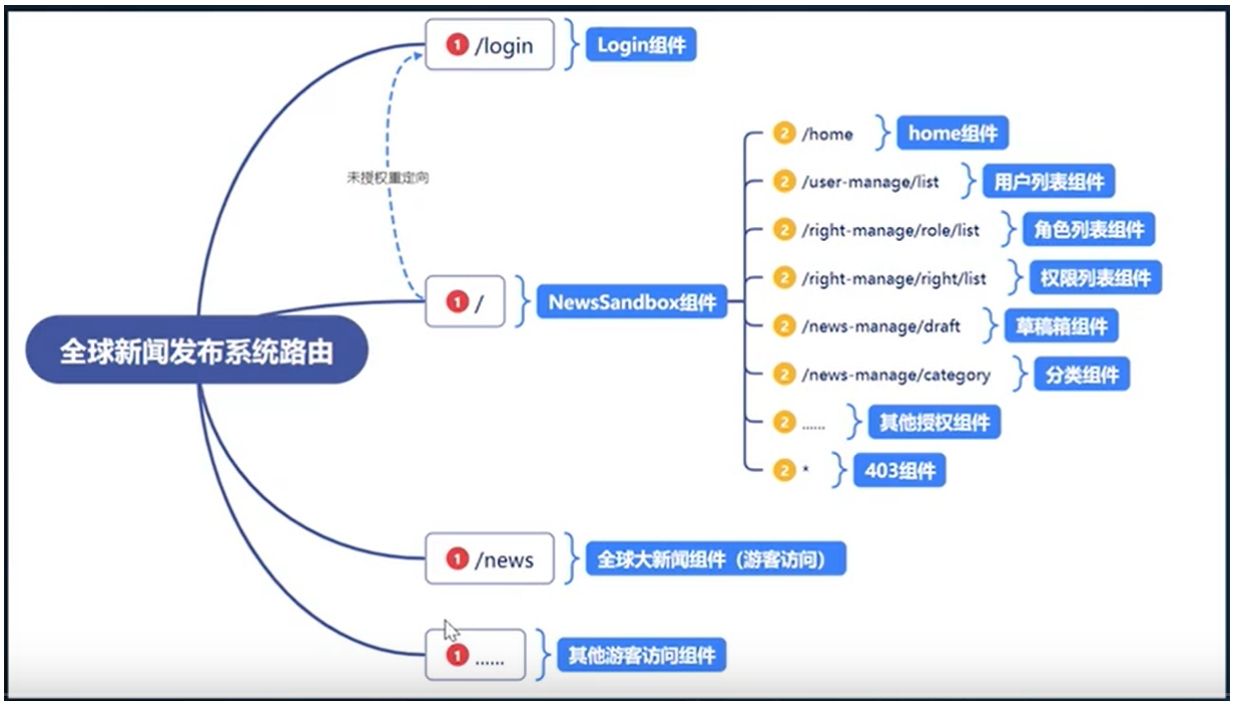
反向代理

详见之前内容

2.路由

路由架构

在没有授权登录的时候不会进入到NewSandBox相关内容 会被重定向到/login



路由创建

1.下载路由 npm install --save react-router-dom@5

2.创建路由文件夹与容纳路由的文件(router文件夹与index.js路由文件)

3.设置路由 切记路由模式设置成Hash模式 Browser模式的话打包上线会有404(也可以打包项目时候修改 不形象项目开发)

4.设置路由模式

```
import React, { Component } from 'react'
// 引用路由模式
import {HashRouter} from "react-router-dom"
export default class index extends Component {
  render() {
    return (
        // 设置路由模式
       <HashRouter>


       </HashRouter>

    )
  }
}
```

5.创建components 与 views 文件夹 并且创建登录页面 与 首页NewSandBox 路由页面与路由配置

6.设置基础页面权限

```
import React, { Component } from 'react'
// 引用路由模式
import { HashRouter, Route, Redirect, Switch } from "react-router-dom"
import Login from "../views/login.jsx"
import NewSandBox from "../views/NewSandBox.jsx"
export default class index extends Component {
  render() {
    return (
      // 设置路由模式
      <HashRouter>
        <Switch>
          <Route path="/login" component={Login} />
          {/* 判断用户是否登陆过如果有那么就进入首页 否则 重定向到登录页面 */}
          {/* <Route path="/" render={() =>{return 判断用户是否登陆过 ?
<NewSandBox /> : <Redirect to="/login" />*/}
          <Route path="/" render={() =>{return
localStorage.getItem("token") ? <NewSandBox /> : <Redirect to="/login"
/>
          } } />
        </Switch>

      </HashRouter>

    )
  }
}
```

3首页基本设置

下图大家会发现在首页中 左侧 与 右侧顶部 是不会变化的 右侧下面区域才是我们需要每次点击导航需要改变的内容所以我们可以使用二级路由来完成

1.创建左侧和右侧顶部的组件 在components下创建leftList.jsx与rightTop.jsx用来容纳

2.在首页NewSandBox.jsx中引用两个组件

```
// 引用左侧与右侧顶部的组件
import LeftList from "../components/leftList.jsx"
import RightTop from "../components/rightTop.jsx"
let NewSandBox=()=>{
  return (
      <div>
          <div>
              {/* 使用 */}
              <LeftList />
              <RightTop />
           </div>
      </div>
  )
}
export default NewSandBox
```

3.设置右侧下半部分的二级路由

1.创建二级路由页面 分别为在views文件夹下

创建home文件夹 home.jsx(首页欢迎页面)

创建user-manage文件夹并在其中创建list.jsx用户列表页面

创建right-manage文件夹中新建 list.jsx 角色列表组件 roleList.jsx权限列表组件

创建no文件夹 no.jsx 404错误页面

2.配置路由规则 在NewSandBox.jsx中配置

```
// 引用左侧与右侧顶部的组件
import LeftList from "../components/leftList.jsx"
import RightTop from "../components/rightTop.jsx"
```

```
// 引用二级路由页面
import Home from "./home/home.jsx"
import Rlist from "../views/right-manage/list.jsx"
import RoleList from "../views/right-manage/roleList.jsx"
import Ulist from "../views/user-manage/list.jsx"
import No from "../views/no/no.jsx"

import { Route, Switch, Redirect } from "react-router-dom"
let NewSandBox = () => {
  return (
    <div>
      <div>
        {/* 使用 */}
        <LeftList />
        <RightTop />

        {/* 配置二级路由规则 */}
        <Switch>
          <Route path="/home" component={Home} />
          <Route path="/user-manage/list" component={Ulist} />
          <Route path="/right-manage/list" component={Rlist} />
          <Route path="/right-manage/roleList" component={RoleList} />
          {/* 设置重定向保证第一次进入就显示欢迎页面home */}
          <Redirect from="/" to="/home" exact />
          {/* 在设置404错误页面 */}
          <Route component={No} />
        </Switch>
      </div>
    </div>
  )
}
export default NewSandBox
```

4.Antd使用

Antd 式蚂蚁集团 开发的一款企业级的后台pc端 React UI类库。

官网：https://ant-design.gitee.io/index-cn

大家一定要记住 文档在手 天下我有

1.基本使用

1.下载： yarn add antd 或 npm install --save antd

2.根据文档提示 我们可以使用它的button组件

```
import React, { Component } from 'react'
// 引用指定内容
import { Button } from 'antd';
export default class Home extends Component {
  render() {
    return (
      <div>
          首页欢迎
          {/* 使用 */}
          <Button type="primary">Button</Button>
      </div>
    )
  }
}
```

3.设置样式antd样式

官方式这样说的：

修改 `src/App.css`，在文件顶部引入 `antd/dist/antd.css`

但是我们可以：

在views文件夹设置一个css 在其中引用antd的样式

```
@import '~antd/dist/antd.css';
```

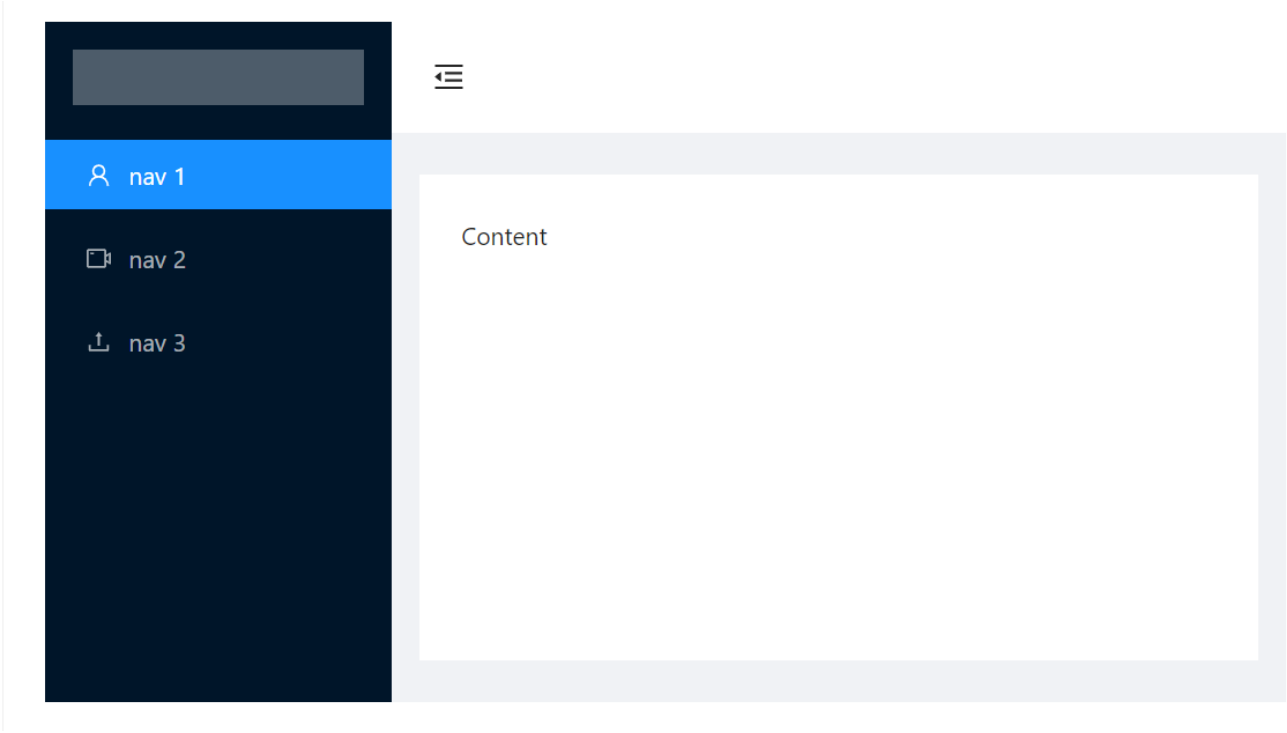
并且把刚设置的css 在views下的index.js中引用

```
import "./index.css";
```

到此antd就已经成功安装到我们的项目中了

2.layout布局

在官网中发现下面的布局很符合我们的要求



分析代码我们发现需要改造之前的内容

1.修改整体页面布局NewSandBox.jsx中 把最外层的div 修改成layout 标签 但是不要忘了引用

```
xxxxxx
xxxxx
xxxx
// 引用
import { Layout } from 'antd';

let NewSandBox = () => {
  return (
    // 修改成Layout
    <Layout>
    xxxxxx
    </Layout>
  )
}
export default NewSandBox
```

2.设置左边导航修改Leftlist.jsx中设置

```
// 不要忘了引用
import { Layout, Menu } from 'antd';

import {
    UserOutlined,
```

```
    VideoCameraOutlined,
    UploadOutlined,
  } from '@ant-design/icons';

const { Sider } = Layout;


let LeftList = () => {
    return (
        <Sider trigger={null} collapsible >
         <div className="logo" ></div>
            <Menu theme="dark" mode="inline" defaultSelectedKeys=
{['1']}>
                <Menu.Item key="1" icon={<UserOutlined />}>
                    nav 1
                </Menu.Item>
                <Menu.Item key="2" icon={<VideoCameraOutlined />}>
                    nav 2
                </Menu.Item>
                <Menu.Item key="3" icon={<UploadOutlined />}>
                    nav 3
                </Menu.Item>
            </Menu>
        </Sider>
    )
}
export default LeftList
```

3.设置有测的布局容器 在NewSandBox.jsx设置右侧内容的容器为layout

```
import LeftList from "../components/leftList.jsx"
import RightTop from "../components/rightTop.jsx"
// 引用二级路由页面
import Home from "./home/home.jsx"
import Rlist from "../views/right-manage/list.jsx"
import RoleList from "../views/right-manage/roleList.jsx"
import Ulist from "../views/user-manage/list.jsx"
import No from "../views/no/no.jsx"
// 引用左侧与右侧顶部的组件
import { Route, Switch, Redirect } from "react-router-dom"
// 引用
import "./NewSandBox.css"
import { Layout } from 'antd';
const {Content } = Layout;
let NewSandBox = () => {
  return (
    // 修改成Layout
    <Layout className="site-layout">
```

```
          {/* 使用 */}
          <LeftList />
          设置有测容器为Layout
          <Layout>
              xxxxxxxx
          </Layout>

      </Layout>
    )
}
export default NewSandBox
```

4.设置右侧顶部在 rightTop.jsx中

```
import React, { Component,useState } from 'react'
import { Layout } from 'antd';
import {
  MenuUnfoldOutlined,
    MenuFoldOutlined
  } from '@ant-design/icons';

// 但是Header还需要引用
const { Header } = Layout;
let RightTop=()=>{
    创建变量
  const [collapsed]=useState(false)
  return (
    <Header className="site-layout-background" style={{ padding: "0
16px" }}>

    {/* {React.createElement(this.state.collapsed ? MenuUnfoldOutlined :
MenuFoldOutlined, {
      className: 'trigger',
      onClick: this.toggle,
    })} */}
    {
      collapsed?<MenuUnfoldOutlined/>:<MenuFoldOutlined/>
    }
  </Header>
  )
}
export default RightTop
```

5.使用content包裹路由设置内容 在NewSandBox中设置

```
import LeftList from "../components/leftList.jsx"
import RightTop from "../components/rightTop.jsx"

// 引用二级路由页面
import Home from "./home/home.jsx"
import Rlist from "../views/right-manage/list.jsx"
import RoleList from "../views/right-manage/roleList.jsx"
import Ulist from "../views/user-manage/list.jsx"
import No from "../views/no/no.jsx"

// 引用左侧与右侧顶部的组件
import { Route, Switch, Redirect } from "react-router-dom"
let NewSandBox = () => {
  return (
    // 修改成Layout
    <Layout>
      <div>
        {/* 使用 */}
        <LeftList />

        <Layout className="site-layout">
          <RightTop />
          {/* 配置二级路由规则 */}
      {/* 使用content组件包裹路由  不要忘了上面引用*/}
          <Content
            className="site-layout-background"
            style={{
              margin: '24px 16px',
              padding: 24,
              minHeight: 280,
            }}
          >
           xxxxxxx

          </Content>
        </Layout>
      </div>
    </Layout>
  )
}
export default NewSandBox
```

6.引用样式在views下新建NewSandBox.css 并且在NewSandBox。.js中引用

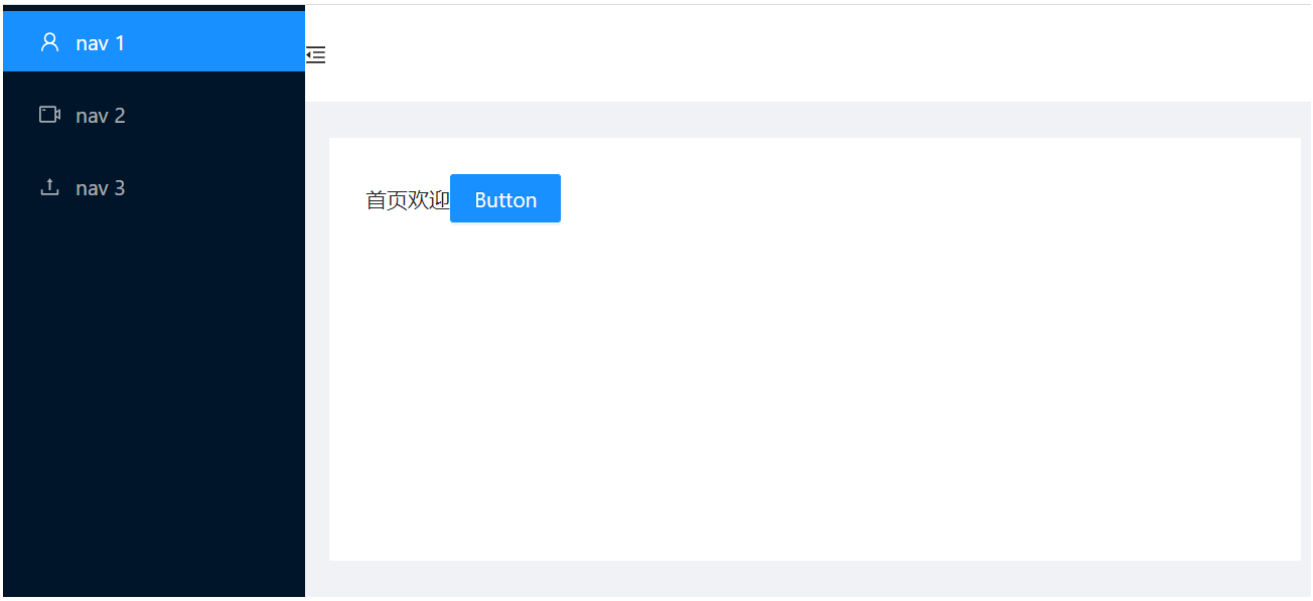
```
#components-layout-demo-custom-trigger .trigger {
    padding: 0 24px;
    font-size: 18px;
    line-height: 64px;
```

```
    cursor: pointer;
    transition: color 0.3s;
  }

  #components-layout-demo-custom-trigger .trigger:hover {
    color: #1890ff;
  }

  #components-layout-demo-custom-trigger .logo {
    height: 32px;
    margin: 16px;
    background: rgba(255, 255, 255, 0.3);
  }

  .site-layout .site-layout-background {
    background: #fff;
  }
```

NewSandBox中引用

```
import "./NewSandBox.css"
```

完成如下效果



5.样式优化

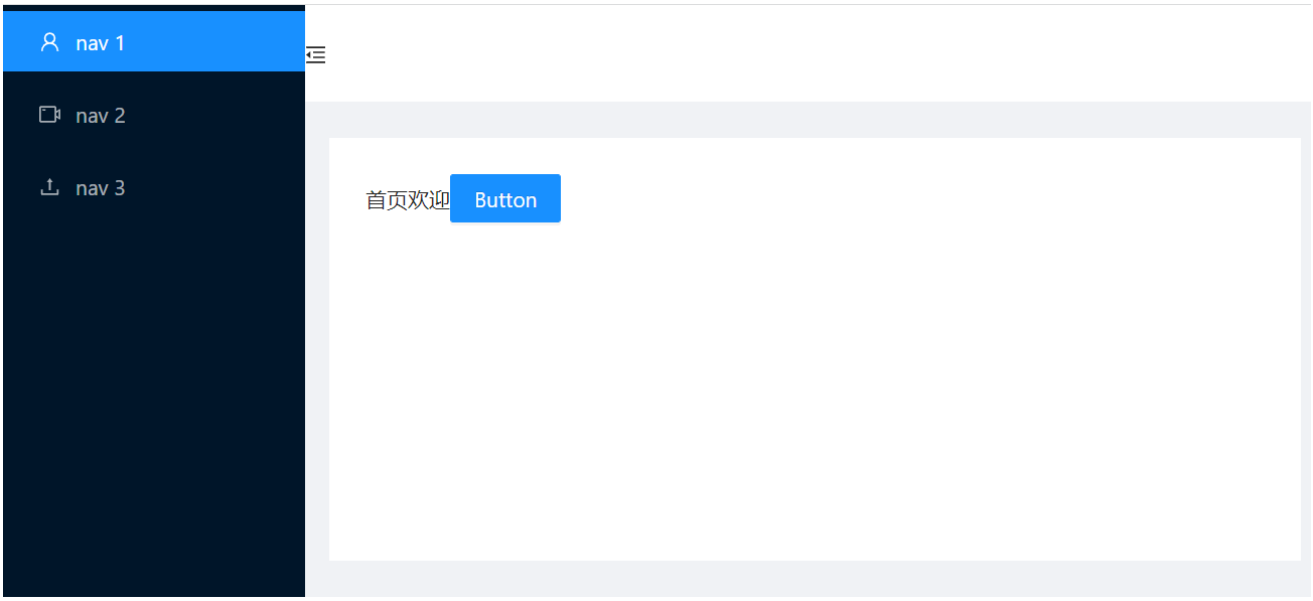
1.设置页面高度100%；抓取后发现我们只需要把容器的height设置成100%即可

```
<!DOCTYPE html>
<html lang="en">
▶<head>…</head>
▼<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  ▼<div id="root">
    ▶<section class="ant-layout ant-layout-has-sider">…</section> flex
     == $0
  </div>
  <!--
        This HTML file is a template.
        If you open it directly in the browser, you will see an empty page.

        You can add webfonts, meta tags, or analytics to this file.
```

在NewSandBox.css中设置

```
#root,.ant-layout{
     height: 100%;
 }
```

2.设置右边header部分的图标部分rightTop中设置



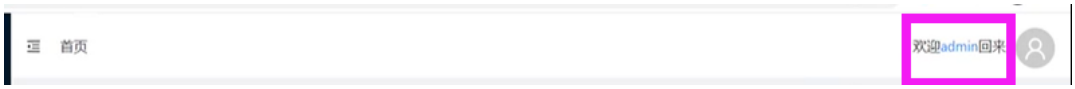

```
  设置下padding的位置
  <Header className ="site-layout-background" style={{ padding: "0 16px"
}}>
```

2.设置右边header部分的图标点击之后切换显示

```
import React, { useState } from 'react'
import { Layout } from 'antd';
import {
  MenuUnfoldOutlined,
    MenuFoldOutlined
  } from '@ant-design/icons';
```

```
// 但是Header还需要引用
const { Header } = Layout;
let RightTop=()=>{
  const [collapsed,upcollapsed]=useState(false)
  // 修改下控制icon的变量取反
  let iconup=()=>{
    upcollapsed(!collapsed)
  }
  return (
    <Header className="site-layout-background" style={{ padding: "0
16px" }}>

    {/* {React.createElement(this.state.collapsed ? MenuUnfoldOutlined :
MenuFoldOutlined, {
      className: 'trigger',
      onClick: this.toggle,
    })} */}
    {
      collapsed?<MenuUnfoldOutlined onClick={iconup}/>:<MenuFoldOutlined
onClick={iconup}/>
    }
  </Header>
  )
}
export default RightTop
```

3.完成文字内容



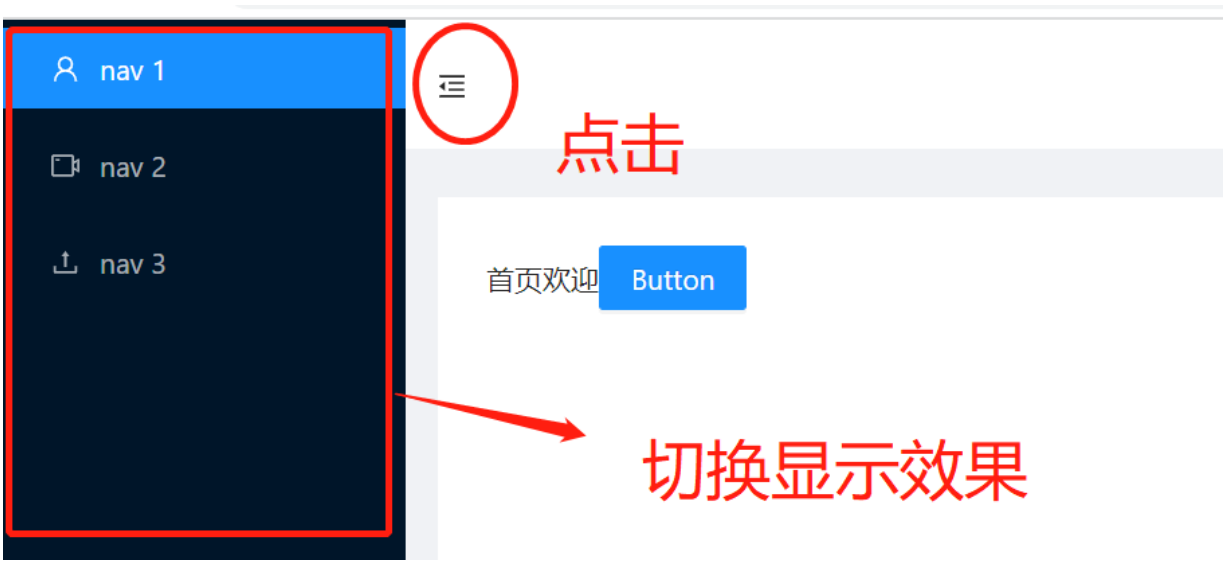

```
<span style={{float:"right"}}>欢迎xx</span>
```

4.设置鼠标移入下拉框


```
import React, { useState } from 'react'
import { Layout, Menu, Dropdown, Avatar } from 'antd';
import { DownOutlined } from '@ant-design/icons';

import {
  MenuUnfoldOutlined,
```

```
  MenuFoldOutlined,
  UserOutlined
} from '@ant-design/icons';

// 但是Header还需要引用
const { Header } = Layout;
let RightTop = () => {
 xxxxxxxx

  let menu = (
    <Menu>
      <Menu.Item>
        11
      </Menu.Item>
      <Menu.Item>
        22
      </Menu.Item>
      <Menu.Item >
        33
      </Menu.Item>
      <Menu.Item danger>退出登录</Menu.Item>
    </Menu>
  );
  return (
    <Header className="site-layout-background" style={{ padding: "0
16px" }}>

    xxxxxx

      <div style={{ float: "right" }}>
        <span >欢迎xx</span>

        {/* 设置下拉框 不要忘了引用*/}
        <Dropdown overlay={menu}>
          <Avatar size="large" icon={<UserOutlined />} />
        </Dropdown>
      </div>

    </Header>
  )
}
export default RightTop
```

6设置左侧导航



去文档找到siderhttps://ant-design.gitee.io/components/layout-cn/#Layout.Sider 会发现可以设置

collapsed 当前收起状态

```
// 不要忘了引用
import { Layout, Menu } from 'antd';

import {
    UserOutlined,
    VideoCameraOutlined,
    UploadOutlined,
  } from '@ant-design/icons';

const { Sider } = Layout;


let LeftList = () => {
    return (
        // 设置之后会发现左侧菜单样式
        <Sider trigger={null} collapsible collapsed={true}>
            <div className="logo" ></div>
            <Menu theme="dark" mode="inline" defaultSelectedKeys=
{['1']}>
                <Menu.Item key="1" icon={<UserOutlined />}>
                    nav 1
                </Menu.Item>
                <Menu.Item key="2" icon={<VideoCameraOutlined />}>
                    nav 2
                </Menu.Item>
```

```
                <Menu.Item key="3" icon={<UploadOutlined />}>
                    nav 3
                </Menu.Item>
            </Menu>
        </Sider>
    )
}
export default LeftList
```

设置左侧导航标题文字

```
// 不要忘了引用
import { Layout, Menu } from 'antd';

import {
    UserOutlined,
    VideoCameraOutlined,
    UploadOutlined,
  } from '@ant-design/icons';

const { Sider } = Layout;


let LeftList = () => {
    return (
        // 设置之后会发现左侧菜单样式
        <Sider trigger={null} collapsible collapsed={false}>
            {/* 设置左侧导航标题文字 */}
            <div className="logo" >全球新闻发布系统</div>
          xxxxxxx
          xxxxxx
          xxxxxx
        </Sider>
    )
}
export default LeftList
```

设置文字样式并且不要忘了在组件中引用

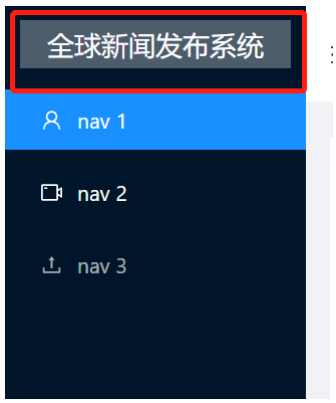
在components中新建index.css

```
.logo{
    line-height: 32px;
    color: white;
    background-color: rgba(255,255,255,.3);
    font-size: 18px;
    margin: 10px;
    text-align: center;

}
```

在组件中引用

```
import { Layout, Menu } from 'antd';
// 引用样式
import "./index.css"
import {
    UserOutlined,
    VideoCameraOutlined,
    UploadOutlined,
  } from '@ant-design/icons';

const { Sider } = Layout;
```

完成如下效果



导航设置伸缩效果 并且引用所需模块

```
// 不要忘了引用
import { Layout, Menu } from 'antd';
// 引用样式
import "./index.css"
import {
    UserOutlined,
    VideoCameraOutlined,
    UploadOutlined,
    MailOutlined
} from '@ant-design/icons';
```

```
const { Sider } = Layout;
const { SubMenu } = Menu;



let LeftList = () => {
    return (
        // 设置之后会发现左侧菜单样式
        <Sider trigger={null} collapsible collapsed={false}>
            {/* 设置左侧导航标题文字 */}
            <div className="logo" >全球新闻发布系统</div>
            <Menu theme="dark" mode="inline" defaultSelectedKeys=
{['1']}>
                  xxxxxx
                  xxxx
                <SubMenu key="sub1" icon={<MailOutlined />}
title="Navigation One">
                    <Menu.Item key="5">Option 5</Menu.Item>
                    <Menu.Item key="6">Option 6</Menu.Item>
                    <Menu.Item key="7">Option 7</Menu.Item>
                    <Menu.Item key="8">Option 8</Menu.Item>
                </SubMenu>
            </Menu>
        </Sider>
    )
}
export default LeftList
```

但是上面的左侧导航是我们写死的

在工作中侧边导航拦都是后台给我们的数据自动生成的 所以我们改造下侧边导航兰 让她变成动态的

所以我们分析导航发现 这个数据最少要有三个数据 分别是 icon key是点击的路由路径 和文字

```
let listdata=[
    {
        key:??
        icon:??
        title:??
    }
]
```

但是我们今后导航还有二级的嵌套

```
let listdata=[
    {
        key:??,
        icon:??,
        title:??,
        children:[
            设置嵌套的内容

        ]
    }
]
```

所以根据效果我们完成动态导航数据


```
// 不要忘了引用
import { Layout, Menu } from 'antd';
// 引用样式
import "./index.css"
import {
    UserOutlined,
    VideoCameraOutlined,
    UploadOutlined,
    MailOutlined
} from '@ant-design/icons';

const { Sider } = Layout;
const { SubMenu } = Menu;
```

```
// 设置导航数据
let listdata=[
    {
        icon:<UserOutlined />,
        title:"首页",
        key:"/home"
    },
    {
        icon:<VideoCameraOutlined />,
        title:"用户管理",
        key:"/user-manage",
        children:[
            {
                icon:<UserOutlined />,
                title:"用户列表",
                key:"/user-manage/list"
            },
        ]
    },
    {
        icon:<VideoCameraOutlined />,
        title:"权限管理",
        key:"/right-manage",
        children:[
            {
                icon:<UserOutlined />,
                title:"角色列表",
                key:"/right-manage/list"
            },
            {
                icon:<UserOutlined />,
                title:"权限列表",
                key:"/right-manage/roleList"
            },
        ]
    },
]

let LeftList = () => {
    return (
      xxxxxxx
    )
}
export default LeftList
```

开始动态渲染

```
// 不要忘了引用
import { Layout, Menu } from 'antd';
```

```
// 引用样式
import "./index.css"
import {
    UserOutlined,
    VideoCameraOutlined,
    UploadOutlined,
    MailOutlined
} from '@ant-design/icons';

const { Sider } = Layout;
const { SubMenu } = Menu;

// 设置导航数据
let listdata=[
    {
        icon:<UserOutlined />,
        title:"首页",
        key:"/home"
    },
    {
        icon:<VideoCameraOutlined />,
        title:"用户管理",
        key:"/user-manage",
        children:[
            {
                icon:<UserOutlined />,
                title:"用户列表",
                key:"/user-manage/list"
            },
        ]
    },
    {
        icon:<VideoCameraOutlined />,
        title:"权限管理",
        key:"/right-manage",
        children:[
            {
                icon:<UserOutlined />,
                title:"角色列表",
                key:"/right-manage/list"
            },
            {
                icon:<UserOutlined />,
                title:"权限列表",
                key:"/right-manage/roleList"
            },
        ]
    },
]
```

```
let LeftList = () => {
    // 创建便利函数
    let showList=(data)=>{
      return  data.map(v=>{
            // 因为数据可能有嵌套的所以我们判断是否有children
            if(v.children){
                return (
                    <SubMenu key="sub1" icon={<MailOutlined />}
title="Navigation One">

                    </SubMenu>
                )
            }else{
              return (
                <Menu.Item key="1" icon={<UserOutlined />}>

                </Menu.Item>
                  )
            }
        })
    }
    return (
        // 设置之后会发现左侧菜单样式
        <Sider trigger={null} collapsible collapsed={false}>
            {/* 设置左侧导航标题文字 */}
            <div className="logo" >全球新闻发布系统</div>
            <Menu theme="dark" mode="inline" defaultSelectedKeys=
{['1']}>
                {/* 调用便利函数把要便利的数据传入 */}
                {showList(listdata)}


            </Menu>
        </Sider>
    )
}
export default LeftList
```

设置数据 并且使用递归便利二层级内容

```
// 不要忘了引用
import { Layout, Menu } from 'antd';
// 引用样式
import "./index.css"
import {
    UserOutlined,
    VideoCameraOutlined,
    UploadOutlined,
```

```
    MailOutlined
} from '@ant-design/icons';

const { Sider } = Layout;
const { SubMenu } = Menu;

// 设置导航数据
let listdata = [
    {
        icon: <UserOutlined />,
        title: "首页",
        key: "/home"
    },
    {
        icon: <VideoCameraOutlined />,
        title: "用户管理",
        key: "/user-manage",
        children: [
            {
                icon: <UserOutlined />,
                title: "用户列表",
                key: "/user-manage/list"
            },
        ]
    },
    {
        icon: <VideoCameraOutlined />,
        title: "权限管理",
        key: "/right-manage",
        children: [
            {
                icon: <UserOutlined />,
                title: "角色列表",
                key: "/right-manage/list"
            },
            {
                icon: <UserOutlined />,
                title: "权限列表",
                key: "/right-manage/roleList"
            },
        ]
    },
]

let LeftList = () => {
    // 创建便利函数
    let showList = (data) => {
       return data.map(v => {
            // 因为数据可能有嵌套的所以我们判断是否有children
```

```
            if (v.children) {
                return (
                    <SubMenu key={v.key} icon={v.icon} title={v.title}>
                        {/* 使用递归方式在次调用便利二层级内容 */}
                        {showList(v.children)}
                    </SubMenu>
                )
            } else {
                return (
                    <Menu.Item key={v.key} icon={v.icon}>
                        {v.title}
                    </Menu.Item>
                )

            }
        })
    }
    return (
        // 设置之后会发现左侧菜单样式
        <Sider trigger={null} collapsible collapsed={false}>
            {/* 设置左侧导航标题文字 */}
            <div className="logo" >全球新闻发布系统</div>
            <Menu theme="dark" mode="inline" defaultSelectedKeys=
{['1']}>
                {/* 调用便利函数把要便利的数据传入 */}
                {showList(listdata)}


            </Menu>
        </Sider>
    )
}
export default LeftList
```

设置路由导航点击跳转

```
xxxx
xxxx
// 函数组件设置props
let LeftList = (props) => {
    // 创建便利函数
    let showList = (data) => {
       return data.map(v => {
            // 因为数据可能有嵌套的所以我们判断是否有children
            if (v.children) {
                return (
                    <SubMenu key={v.key} icon={v.icon} title={v.title}>

```

```
                        {/* 使用递归方式在次调用便利二层级内容 */}
                        {showList(v.children)}
                    </SubMenu>
                )
            } else {
                return (
                    // 添加点击事件调用编程试导航
                    <Menu.Item key={v.key} icon={v.icon} onClick={
                        ()=>{
                            props.history.push(v.key)
                        }
                    }>
                        {v.title}
                    </Menu.Item>
                )

            }
        })
    }
xxxxx
xxxxx
xxxx
}
export default LeftList
```

但是报错了 所以我们使用withRouter高阶组件完成

```
import {withRouter} from "react-router-dom"
    xxxxxxx
    xxxxxx
export default withRouter(LeftList)
```

7.权限列表

1.使用json-server模拟后台对应接口 启动mock文件夹下提供的json数据

但是大家发现我们两个导航是不同的接口 那么怎么把它们合并起来呢?

所以我们可以使用json-server的**_embed**联查方式得到就是用来获取包含下级资源的数据

在组件中发送请求尝试获取

```
import {withRouter} from "react-router-dom"
import {useEffect,useState} from "react"
import $http from "axios"
// 不要忘了引用
import { Layout, Menu } from 'antd';
// 引用样式
import "./index.css"
import {
    UserOutlined,
    VideoCameraOutlined,
    UploadOutlined,
    MailOutlined
} from '@ant-design/icons';

const { Sider } = Layout;
const { SubMenu } = Menu;

// 设置导航数据
let listdata = [
    {
        icon: <UserOutlined />,
        title: "首页",
        key: "/home"
    },
```

```
    {
        icon: <VideoCameraOutlined />,
        title: "用户管理",
        key: "/user-manage",
        children: [
            {
                icon: <UserOutlined />,
                title: "用户列表",
                key: "/user-manage/list"
            },
        ]
    },
    {
        icon: <VideoCameraOutlined />,
        title: "权限管理",
        key: "/right-manage",
        children: [
            {
                icon: <UserOutlined />,
                title: "角色列表",
                key: "/right-manage/list"
            },
            {
                icon: <UserOutlined />,
                title: "权限列表",
                key: "/right-manage/roleList"
            },
        ]
    },
]
let LeftList = (props) => {

    // 发送请求
  useEffect(()=>{
    $http("http://localhost:8888/rights?_embed=children").then((ok)=>{
        console.log(ok.data)
    })

    },[])

    let showList = (data) => {
       return data.map(v => {
            if (v.children) {
                return (
                    <SubMenu key={v.key} icon={v.icon} title={v.title}>
                        {showList(v.children)}
                    </SubMenu>
                )
            } else {
```

```
                return (
                    <Menu.Item key={v.key} icon={v.icon} onClick={
                        ()=>{
                            props.history.push(v.key)
                        }
                    }>
                        {v.title}
                    </Menu.Item>
                )

            }
        })
    }
    return (
        // 设置之后会发现左侧菜单样式
        <Sider trigger={null} collapsible collapsed={false}>
            {/* 设置左侧导航标题文字 */}
            <div className="logo" >全球新闻发布系统</div>
            <Menu theme="dark" mode="inline" defaultSelectedKeys=
{['1']}>
                {/* 调用便利函数把要便利的数据传入 */}
                {showList(listdata)}


            </Menu>
        </Sider>
    )
}
export default withRouter(LeftList)
```

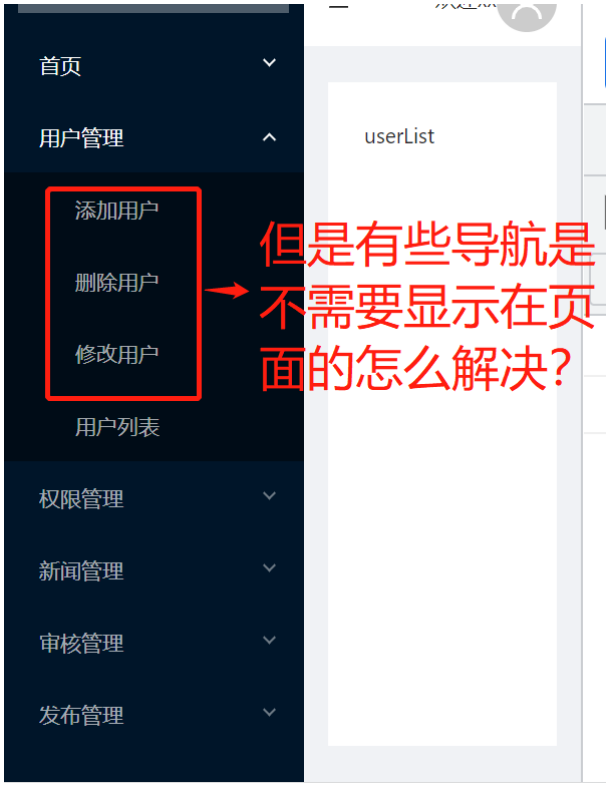
把请求来得数据动态生成 把请求得数据交给useState 并且传入到便利函数中

```
import {withRouter} from "react-router-dom"
import {useEffect,useState} from "react"
import $http from "axios"
// 不要忘了引用
import { Layout, Menu } from 'antd';
// 引用样式
import "./index.css"
import {
    UserOutlined,
    VideoCameraOutlined,
    UploadOutlined,
    MailOutlined
} from '@ant-design/icons';

const { Sider } = Layout;
const { SubMenu } = Menu;
```

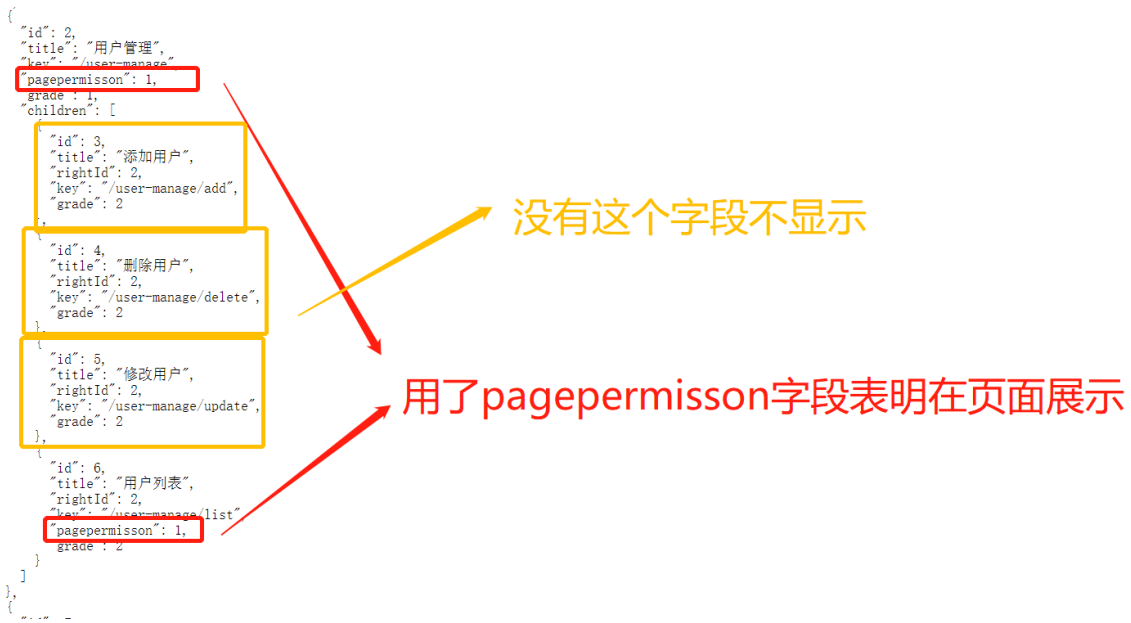
```
let LeftList = (props) => {
    // 创建接收变量
    let [listdata,setlistdata]=useState([])

    // 发送请求
  useEffect(()=>{
    $http("http://localhost:8888/rights?_embed=children").then((ok)=>{
        console.log(ok.data)
        // 修改
        setlistdata(ok.data)
    })

    },[])

    let showList = (data) => {
       return data.map(v => {
            if (v.children) {
                return (
                    <SubMenu key={v.key} icon={v.icon} title={v.title}>
                        {showList(v.children)}
                    </SubMenu>
                )
            } else {
                return (
                    <Menu.Item key={v.key} icon={v.icon} onClick={
                        ()=>{
                            props.history.push(v.key)
                        }
                    }>
                        {v.title}
                    </Menu.Item>
                )

            }
        })
    }
    return (
        // 设置之后会发现左侧菜单样式
        <Sider trigger={null} collapsible collapsed={false}>
            {/* 设置左侧导航标题文字 */}
            <div className="logo" >全球新闻发布系统</div>
            <Menu theme="dark" mode="inline" defaultSelectedKeys=
{['1']}>
                {/* 调用便利函数把要便利的数据传入 */}
                {showList(listdata)}


            </Menu>
```

```
            </Sider>
        )
    }
    export default withRouter(LeftList)
```

设置指定内容显示



在工作中有的时候后台给我们的数据有的时候有些内容是不需要显示的（我们可以在项目开发前期和后台对接好 那么后台就会把这写不需要在页面显示的内容 在数据上进行标识 比我们们现在的数据）



所以我们在渲染的时候可以判断当前这个字段是否为1 如果为1 就渲染当前内容 否则不渲染

```
import {withRouter} from "react-router-dom"
import {useEffect,useState} from "react"
import $http from "axios"
// 不要忘了引用
import { Layout, Menu } from 'antd';
// 引用样式
import "./index.css"
import {
    UserOutlined,
    VideoCameraOutlined,
    UploadOutlined,
    MailOutlined
} from '@ant-design/icons';

const { Sider } = Layout;
const { SubMenu } = Menu;


let LeftList = (props) => {
    // 创建接收变量
    let [listdata,setlistdata]=useState([])

    // 发送请求
  useEffect(()=>{
    $http("http://localhost:8888/rights?_embed=children").then((ok)=>{
        console.log(ok.data)
        // 修改
        setlistdata(ok.data)
    })

    },[])

    let showList = (data) => {
       return data.map(v => {
        //    设置判断
            if (v.children&&v.pagepermisson==1) {
                return (
                    <SubMenu key={v.key} icon={v.icon} title={v.title}>
                        {showList(v.children)}
                    </SubMenu>
                )
            } else {
                return (
                    v.pagepermisson==1&&<Menu.Item key={v.key} icon=
{v.icon} onClick={
                        ()=>{
                            props.history.push(v.key)
                        }
                    }>
```

```
                        {v.title}
                    </Menu.Item>
                )

            }
        })
    }
    return (
        // 设置之后会发现左侧菜单样式
        <Sider trigger={null} collapsible collapsed={false}>
            {/* 设置左侧导航标题文字 */}
            <div className="logo" >全球新闻发布系统</div>
            <Menu theme="dark" mode="inline" defaultSelectedKeys=
{['1']}>
                {/* 调用便利函数把要便利的数据传入 */}
                {showList(listdata)}


            </Menu>
        </Sider>
    )
}
export default withRouter(LeftList)
```

设置图标

因为我们不能给后台约束我们具体要什么图标 所以我们可以自定添加显示逻辑

```
import {withRouter} from "react-router-dom"
import {useEffect,useState} from "react"
import $http from "axios"
// 不要忘了引用
import { Layout, Menu } from 'antd';
// 引用样式
import "./index.css"
import {
    UserOutlined,
    VideoCameraOutlined,
    UploadOutlined,
    MailOutlined
} from '@ant-design/icons';

const { Sider } = Layout;
const { SubMenu } = Menu;

// 设置导航数据
let listdata = [
    {
        icon: <UserOutlined />,
```

```
        title: "首页",
        key: "/home"
    },
    {
        icon: <VideoCameraOutlined />,
        title: "用户管理",
        key: "/user-manage",
        children: [
            {
                icon: <UserOutlined />,
                title: "用户列表",
                key: "/user-manage/list"
            },
        ]
    },
    {
        icon: <VideoCameraOutlined />,
        title: "权限管理",
        key: "/right-manage",
        children: [
            {
                icon: <UserOutlined />,
                title: "角色列表",
                key: "/right-manage/list"
            },
            {
                icon: <UserOutlined />,
                title: "权限列表",
                key: "/right-manage/roleList"
            },
        ]
    },
]



// 创建图标数组
let dataicon={
    "/home":<UserOutlined />,
    "/user-manage":<VideoCameraOutlined />,
    "/user-manage/list":<UserOutlined />
    // ....

}
let LeftList = (props) => {
    // 创建接收变量
    let [listdata,setlistdata]=useState([])

    // 发送请求
```

```
  useEffect(()=>{
    $http("http://localhost:8888/rights?_embed=children").then((ok)=>{
        console.log(ok.data)
        // 修改
        setlistdata(ok.data)
    })

    },[])

    let showList = (data) => {
       return data.map(v => {
        //    设置判断
            if (v.children&&v.pagepermisson==1) {
                return (
                    // 插入图标
                    <SubMenu key={v.key} icon={dataicon[v.key]} title=
{v.title}>
                        {showList(v.children)}
                    </SubMenu>
                )
            } else {
                return (
                      // 插入图标
                    v.pagepermisson==1&&<Menu.Item key={v.key} icon=
{dataicon[v.key]} onClick={
                        ()=>{
                            props.history.push(v.key)
                        }
                    }>
                        {v.title}
                    </Menu.Item>
                )

            }
        })
    }
    return (
        // 设置之后会发现左侧菜单样式
        <Sider trigger={null} collapsible collapsed={false}>
            {/* 设置左侧导航标题文字 */}
            <div className="logo" >全球新闻发布系统</div>
            <Menu theme="dark" mode="inline" defaultSelectedKeys=
{['1']}>
                {/* 调用便利函数把要便利的数据传入 */}
                {showList(listdata)}


            </Menu>
        </Sider>
```

```
    )
}
export default withRouter(LeftList)
```

但是现在大家发现首页本身是没有二级的但是现在有二级箭头我们可以取消他

只需要判断长度即可

```
 //     设置判断长度是否大于0
            if (v.children.length>0&&v.pagepermisson==1) {
                return (
                    // 插入图标
                    <SubMenu key={v.key} icon={dataicon[v.key]} title=
{v.title}>
                        {showList(v.children)}
                    </SubMenu>
                )
            } else {
                return (
                      // 插入图标
                    v.pagepermisson==1&&<Menu.Item key={v.key} icon=
{dataicon[v.key]} onClick={
                        ()=>{
                            props.history.push(v.key)
                        }
                    }>
                        {v.title}
                    </Menu.Item>
                )

            }
        })
```

但是报错了

leftList.jsx:83 Uncaught TypeError: Cannot read properties of undefined (reading 'length')

是因为有的里面没有children所以有个判断的小技巧 使用三元运算符 如果?前面是undefined那么就不执行.length

```
v.children?.length>0
```

```
 if (v.children?.length>0&&v.pagepermisson==1) {
                return (
```

```
                    // 插入图标
                    <SubMenu key={v.key} icon={dataicon[v.key]} title=
{v.title}>
                        {showList(v.children)}
                    </SubMenu>
                )
            } else {
                return (
                      // 插入图标
                    v.pagepermisson==1&&<Menu.Item key={v.key} icon=
{dataicon[v.key]} onClick={
                        ()=>{
                            props.history.push(v.key)
                        }
                    }>
                        {v.title}
                    </Menu.Item>
                )

            }
```

但是现在每次刷新都只显示第一项高亮 我们想选中谁谁高亮 哪怕是刷新之后 我们就要设置Menu的defaultSelectedKeys属性初始选中的菜单项 key 数组

我们可以通过props.location得到当前页面的路由路径 在传入defaultSelectedKeys即可

```
xxxxx
xxxxx

    console.log(props.location.pathname)//得到信息
    let selectKeys=props.location.pathname
    return (
        <Sider trigger={null} collapsible collapsed={false}>
            <div className="logo" >全球新闻发布系统</div>
            {/* 传入到初始选中的菜单项 key */}
            <Menu theme="dark" mode="inline" defaultSelectedKeys=
{selectKeys}>
                {showList(listdata)}


            </Menu>
        </Sider>
    )
}
export default withRouter(LeftList)
```

但是一刷新二级菜单是折叠的 我们让选中之后打开这个二级菜单

所以我们需要把一级路由路径截取出来 传入到defaultOpenKeys这个属性中初始展开的 SubMenu 菜单项 key 数组

```
    console.log(props.location.pathname)//得到信息
    let selectKeys=props.location.pathname
      // 截取到一级路由路径 注意这是一个数组
    let openKeys=    let openKeys=[
"/"+props.location.pathname.split("/")[1] ]
    return (
        <Sider trigger={null} collapsible collapsed={false}>
            <div className="logo" >全球新闻发布系统</div>
            {/* 传入到初始选中的菜单项 key */}
            <Menu theme="dark" mode="inline" defaultSelectedKeys=
{selectKeys}
            // 在设置一级导航展开
                defaultOpenKeys={openKeys}
            >
                {showList(listdata)}


            </Menu>
        </Sider>
    )
}
export default withRouter(LeftList)
```

8.权限列表

我们可以通过权限列表页面的操纵 控制当前页面列表权限显示和隐藏

 11111

大家看上面发现权限列表最基本得到就是一个表格

所以我们就是antd先完成页面表格创建 roleList.jsx

表格创建

根据antd官网会发现表格创建极其简单

```
// 不要忘了引用
import {Table} from "antd"
const dataSource = [
  {
    key: '1',
    name: '胡彦斌',
    age: 32,
```

```
    address: '西湖区湖底公园1号',
  },
  {
    key: '2',
    name: '胡彦祖',
    age: 42,
    address: '西湖区湖底公园1号',
  },
];

const columns = [
  {
    title: '姓名',
    dataIndex: 'name',
    key: 'name',
  },
  {
    title: '年龄',
    dataIndex: 'age',
    key: 'age',
  },
  {
    title: '住址',
    dataIndex: 'address',
    key: 'address',
  },
];

let LoleList=()=>{
  return (
      <div>
          <Table dataSource={dataSource} columns={columns} />
      </div>
  )
}
export default LoleList
```

但是页面展示内容需要从后台读取所以我们修改展示数据 使用useState创建成状态

```
import {useState} from "react"
// 不要忘了引用
import {Table} from "antd"
const columns = [
  {
    title: '姓名',
    dataIndex: 'name',
    key: 'name',
  },
  {
```

```
    title: '年龄',
    dataIndex: 'age',
    key: 'age',
  },
  {
    title: '住址',
    dataIndex: 'address',
    key: 'address',
  },
];

let LoleList=()=>{
  // 设置状态
  let [dataSource,setdataSource]=useState([
    {
      key: '1',
      name: '胡彦斌',
      age: 32,
      address: '西湖区湖底公园1号',
    },
    {
      key: '2',
      name: '胡彦祖',
      age: 42,
      address: '西湖区湖底公园1号',
    },
  ])
  return (
      <div>
          <Table dataSource={dataSource} columns={columns} />
      </div>
  )
}
export default LoleList
```

设置请求把展示数据动态获取过来

```
import {useState,useEffect} from "react"
// 不要忘了引用
import {Table} from "antd"
import $http from "axios"
const columns = [
  {
    title: '姓名',
    dataIndex: 'name',
    key: 'name',
  },
  {
    title: '年龄',
```

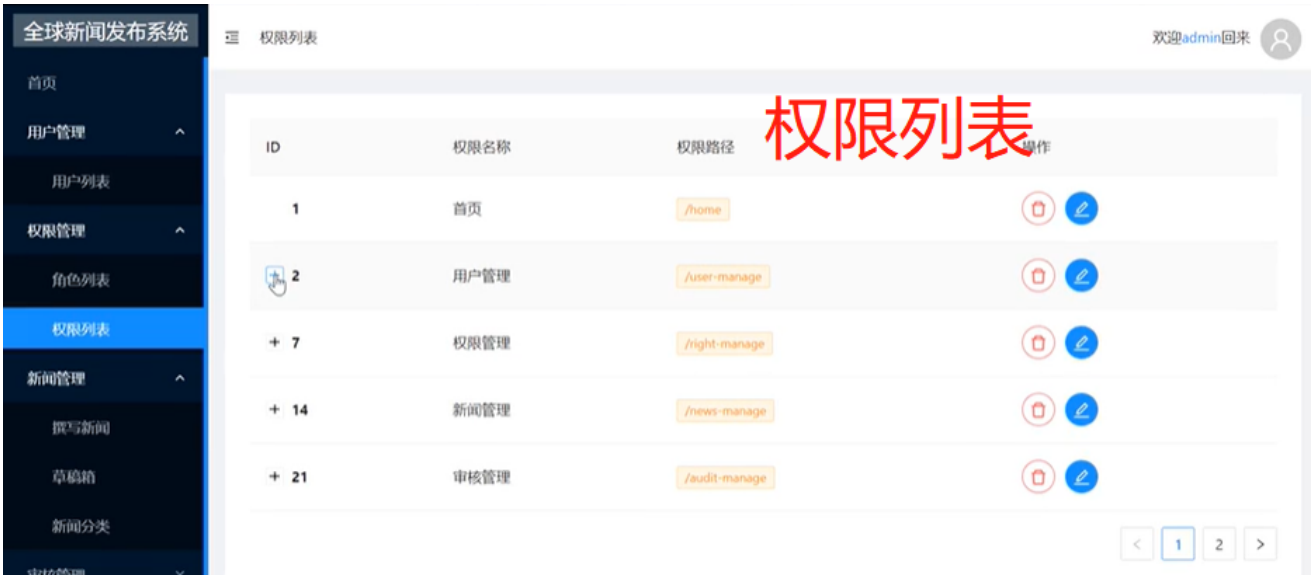
```
    dataIndex: 'age',
    key: 'age',
  },
  {
    title: '住址',
    dataIndex: 'address',
    key: 'address',
  },
];

let LoleList=()=>{
  // 设置状态
  let [dataSource,setdataSource]=useState([])

  // 设置请求
  useEffect(()=>{
    $http("http://localhost:8888/rights").then((ok)=>{
      console.log(ok.data)
      // 把请求来得数据赋值给页面展示变量
      setdataSource(ok.data)
    })
  },[])
  return (
      <div>
          <Table dataSource={dataSource} columns={columns} />
      </div>
  )
}
export default LoleList
```

会发现页面没有展示内容 所以我们根据上面的图片修改 表格的表头

```
import {useState,useEffect} from "react"
// 不要忘了引用
import {Table} from "antd"
import $http from "axios"
// 修改表头
const columns = [
  {
    title: 'ID',
    dataIndex: 'id',

  },
  {
    title: '权限名称',
    dataIndex: 'title'
  },
  {
    title: '权限路径',
```

```
    dataIndex: 'key'
  },
  {
    title: '操作'
  },
];

let LoleList=()=>{
  // 设置状态
  let [dataSource,setdataSource]=useState([])

  // 设置请求
  useEffect(()=>{
    $http("http://localhost:8888/rights").then((ok)=>{
      console.log(ok.data)
      // 把请求来得数据赋值给页面展示变量
      setdataSource(ok.data)
    })
  },[])
  return (
      <div>
          <Table dataSource={dataSource} columns={columns} />
      </div>
  )
}
export default LoleList
```

修改表格样式 ID的粗体　权限路径的标签效果　和操作上的按钮




```
import { useState, useEffect } from "react"
// 不要忘了引用
import { Table, Tag,Button } from "antd"
import $http from "axios"
```

```
import {
  DeleteOutlined,
  EditOutlined,
} from '@ant-design/icons';
// 修改表头
const columns = [
  {
    title: 'ID',
    dataIndex: 'id',
    render: (id) => {//render返回一个jsx就会覆盖原有的展示内容
      return (
        <b>{id}</b>
      )
    }

  },
  {
    title: '权限名称',
    dataIndex: 'title'
  },
  {
    title: '权限路径',
    dataIndex: 'key',
    render: (key) => {//render返回一个jsx就会覆盖原有的展示内容
      return (
        <Tag color="orange">{key}</Tag>
      )
    }

  },
  {
    title: '操作',
    render: (key) => {//render返回一个jsx就会覆盖原有的展示内容
      return (l
        <>
          <Button type="primary" danger  shape="circle">
          <DeleteOutlined />
          </Button>
          <Button type="primary" shape="circle">
          <EditOutlined />
          </Button>
        </>
      )
    }
  },
];

let LoleList = () => {
```

```
  // 设置状态
  let [dataSource, setdataSource] = useState([])

  // 设置请求
  useEffect(() => {
    $http("http://localhost:8888/rights").then((ok) => {
      console.log(ok.data)
      // 把请求来得数据赋值给页面展示变量
      setdataSource(ok.data)
    })
  }, [])
  return (
    <div>
      <Table dataSource={dataSource} columns={columns} />
    </div>
  )
}
export default LoleList
```

但是今后数据太多页面会很难看 所以我们可以进行分页对数据进行处理 可以使用pagination这个属性进行分页的设置

```
<div>
      {/* 设置分页并且设置每页显示条数 */}
      <Table dataSource={dataSource} columns={columns} pagination=
{{pageSize:5}}/>
    </div>
```

设置表格内容展开

大家会发现我们表格中可能有树状图效果 所以我们可以使用antd中table的树形数据展示 来解决

设置请求为带有子项的数据

```
  useEffect(() => {
    $http("http://localhost:8888/rights?_embed=children").then((ok) => {
      console.log(ok.data)
      // 把请求来得数据赋值给页面展示变量
      setdataSource(ok.data)
    })
  }, [])
```

在表格上设置dataSource={数据}

```
  <div>
       {/* 设置分页并且设置每页显示条数 */}
       <Table dataSource={dataSource} columns={columns} pagination=
{{pageSize:5}}/>
    </div>
```

但是第一项没有子内容 还还是有加号提示 所以我们看请求来的数据发现有children 但是其中没有数据 但是渲染的时候他也会渲染上

那么我们移除children长度为0的数据内容

```
  useEffect(() => {
    $http("http://localhost:8888/rights?_embed=children").then((ok) => {
      console.log(ok.data)
      // 根据数据中是否存在children
     let newdata= ok.data.map((v)=>{
      //  判断如果没有children
        if(v.children.length==0){
            // 删除对象中的children
            delete v.children
        }
        return v
      })
      // 把处理好得数据赋值给页面展示变量
      setdataSource(newdata)
    })
  }, [])
```

设置表格的删除点击弹框

```
let LoleList = () => {
  // 修改表头
```

```
const columns = [
  {
    title: 'ID',
    dataIndex: 'id',
    render: (id) => {//render返回一个jsx就会覆盖原有的展示内容
      return (
        <b>{id}</b>
      )
    }

  },
  {
    title: '权限名称',
    dataIndex: 'title'
  },
  {
    title: '权限路径',
    dataIndex: 'key',
    render: (key) => {//render返回一个jsx就会覆盖原有的展示内容
      return (
        <Tag color="orange">{key}</Tag>
      )
    }

  },
  {
    title: '操作',
    // 设置形参当前形参就是你点击的那条内容
    render: () => {
      return (
        <>
        {/* 绑定事件调用函数    设置函数实参传递*/}
          <Button type="primary" danger  shape="circle" onClick={()=>
{del()}}>
          <DeleteOutlined />
          </Button>
          <Button type="primary" shape="circle">
          <EditOutlined />
          </Button>
        </>
      )
    }
  },
];

// 创建删除函数     接收数据
let del=(item)=>{
  confirm({
    title: '是否删除?',
```

```
      onOk() {
        console.log();
      },
      onCancel() {
        console.log('Cancel');
      },
    });
  }

    // 设置状态
    let [dataSource, setdataSource] = useState([])


  }
  export default LoleList
```

得到点击具体是哪个数据的删除

```
let LoleList = () => {
    // 修改表头
  const columns = [
    {
      title: 'ID',
      dataIndex: 'id',
      render: (id) => {//render返回一个jsx就会覆盖原有的展示内容
        return (
          <b>{id}</b>
        )
      }

    },
    {
      title: '权限名称',
      dataIndex: 'title'
    },
    {
      title: '权限路径',
      dataIndex: 'key',
      render: (key) => {//render返回一个jsx就会覆盖原有的展示内容
        return (
          <Tag color="orange">{key}</Tag>
        )
      }

    },
    {
      title: '操作',
```

```
    // 设置形参当前形参就是你点击的那条内容
    render: (item) => {
      return (
        <>
        {/* 绑定事件调用函数    设置函数实参传递*/}
          <Button type="primary" danger  shape="circle" onClick={()=>
{del(item)}}>
          <DeleteOutlined />
          </Button>
          <Button type="primary" shape="circle">
          <EditOutlined />
          </Button>
        </>
      )
    }
  },
];

// 创建删除函数     接收数据
let del=(item)=>{
  confirm({
    title: '是否删除?',
    onOk() {
      console.log(dataSource);
    },
    onCancel() {
      console.log('Cancel');
    },
  });
}

  // 设置状态
  let [dataSource, setdataSource] = useState([])


}
export default LoleList
```

开始删除

```
// 创建删除函数     接收数据
let del=(item)=>{
  confirm({
    title: '是否删除?',
    onOk() {
      console.log(item);//里面有个id就是我们要删除哪一项
      // 开始过滤   我们只要把要删除的内容从展示数据中移除   即可
      setdataSource(dataSource.filter(v=>v.id!=item.id))
```

```
      },
      onCancel() {
        console.log('Cancel');
      },
    });
  }
```

删除数据

```
// 创建删除函数     接收数据
let del=(item)=>{
  confirm({
    title: '是否删除?',
    onOk() {
      console.log(item);//里面有个id就是我们要删除哪一项
      // 开始过滤   我们只要把要删除的内容从展示数据中移除   即可
      setdataSource(dataSource.filter(v=>v.id!=item.id))

      // 执行json-server的函数
      $http.delete(`http://localhost:8888/rights/${item.id}`)

    },
    onCancel() {
      console.log('Cancel');
    },
  });
}
```

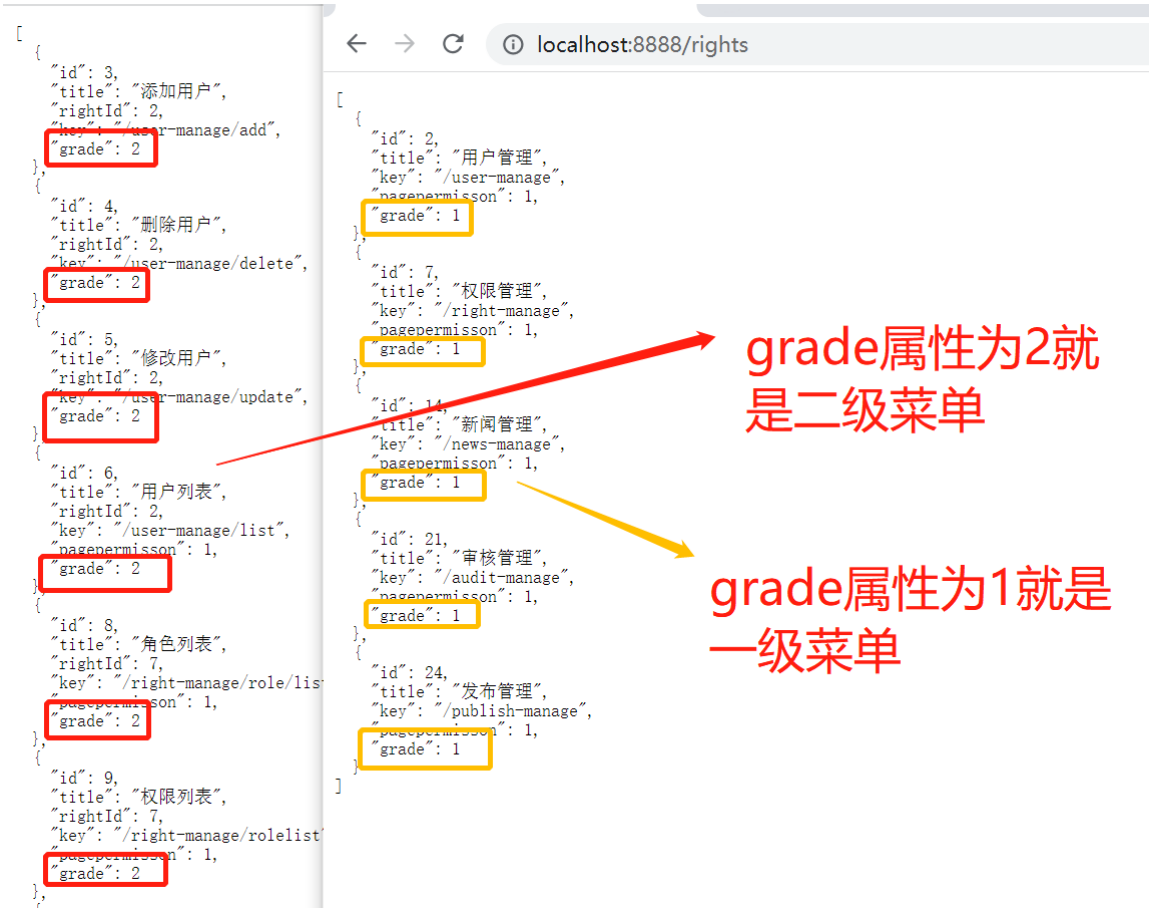
二级删除

但是我们现在在二级中进行删除的话 就会发现删除不了

刚才我们完成的删除只能删除一级的 那么想删除2级的怎么办呢?

所以我们需要先判断 当前用户点击的是一级删除还是二级删除

观察数据 发现




```
    onOk() {
        console.log(item);//里面有个id就是我们要删除哪一项
        if (item.grade == 1) {
          // 执行一级删除
          // 开始过滤   我们只要把要删除的内容从展示数据中移除   即可
          setdataSource(dataSource.filter(v => v.id != item.id))

          // 执行json-server的函数
          $http.delete(`http://localhost:8888/rights/${item.id}`)
        } else {
          console.log("执行二级删除")
        }

      },
```

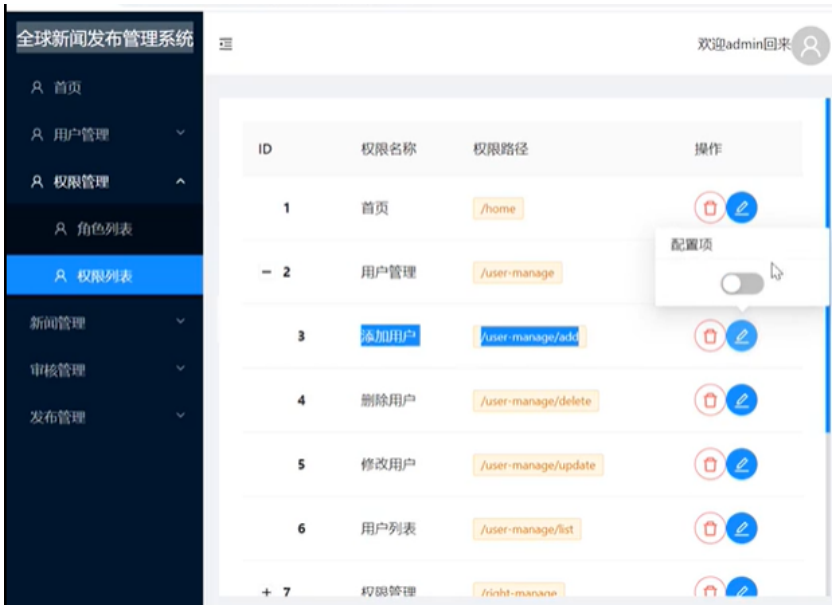
编写二级删除的逻辑 比较麻烦要注意

```
    onOk() {
        console.log(item);//里面有个id就是我们要删除哪一项
        if (item.grade == 1) {
          // 执行一级删除
          // 开始过滤   我们只要把要删除的内容从展示数据中移除   即可
          setdataSource(dataSource.filter(v => v.id != item.id))

          // 执行json-server的函数
```

```
            $http.delete(`http://localhost:8888/rights/${item.id}`)
          } else {
            console.log("执行二级删除")
            // 找到你点击的是那个一级数据
            // v.id是你便利数据的时候id
            // item.rightId是 一级数据的标识项
           let list= dataSource.filter(v => v.id == item.rightId)
            console.log(list)
            // 在对应的数据中便利 把不是点击的数据替换原始数据
            list[0].children= list[0].children.filter(v => v.id !=
item.id)
            console.log(dataSource)

            // 修改数据   但是注意要使用扩展运算符
            setdataSource([...dataSource])

                // 执行json-server的函数
            $http.delete(`http://localhost:8888/children/${item.id}`)
          }

        },
```

设置切换显示隐藏当前导航

因为之前咱们做的都是删除   我也可以设置一个修改来控制是否显示导航



完成页面 可以使用气泡卡片和开关组件完成

```
 <Popover content={
              ( <div><Switch defaultChecked /></div>)}
              title="请选择" trigger="hover">
            {/* 设置是否能使用 判断如果没有就不让使用*/}
              <Button type="primary" shape="circle">
                <EditOutlined />
              </Button>
            </Popover>
```

同时我们让没有在页面展示的配置项不能选择

因为我们数据中有一个"pagepermisson": 1数据是设置当前配置项是否显示的 所以 我们就根据这个属性来进行设置

```
    {/* 给按钮添加气泡弹框 */}
            {/* 就算下面按钮禁用了鼠标移入也有弹出框 所以禁止 */}
            <Popover content={
              ( <div><Switch defaultChecked onClick={switchFun}/>
</div>)}
              title="请选择" trigger=
{item.pagepermisson==undefined?'':"hover"}>
            {/* 设置是否能使用 判断如果没有就不让使用*/}
              <Button type="primary" shape="circle" disabled=
{item.pagepermisson==undefined}>
                <EditOutlined />
              </Button>
            </Popover>
```

设置勾选事件调用函数 完成修改

```
      {/* 就算下面按钮禁用了鼠标移入也有弹出框 所以禁止 */}
            <Popover content={
              ( <div><Switch defaultChecked onClick={()=>
{switchFun(item)}}/></div>)}
              title="请选择" trigger=
{item.pagepermisson==undefined?'':"hover"}>
            {/* 设置是否能使用 判断如果没有就不让使用*/}
              <Button type="primary" shape="circle" disabled=
{item.pagepermisson==undefined}>
                <EditOutlined />
              </Button>
            </Popover>
```

开始执行修改函数

```
```

```
let switchFun=(item)=>{
    console.log(item)//打印你勾选的哪一项
    // 修改这一项的pagepermisson数据 1变0  0变1  就能进行显示和隐藏了
    item.pagepermisson=item.pagepermisson==1?0:1

    // 修改数据
    setdataSource([...dataSource])
    console.log(item.pagepermisson)

    // 修改数据
    if(item.pagepermisson==1){
      // 修改1级别
      $http.patch(`http://localhost:8888/rights/${item.id}`,
{pagepermisson:item.pagepermisson})
    }else{
       // 修改2级别
      $http.patch(`http://localhost:8888/children/${item.id}`,
{pagepermisson:item.pagepermisson})
    }

  }
```