## =============================================

更更多干货技术访问小**D**课堂官网 **[https://xdclass.net](https://xdclass.net/)**

小**D**课堂**QQ**客服**Vicky**：**561509994** 公众号搜索：小**D**课堂

## =============================================

### 愿景：**"**让编程不不在难学，让技术与生活更更加有趣**"**

第一章 欢迎来到**React**世界

第一节 详细了了解我们的**React**

### 用于构建用户界面的 JavaScript 库

学习React需要比较牢固的JS基础和熟悉ES6语法

React没有太多的api，可以说用react编程都是在写js

小D课堂---Tim

第二节 精讲安装**node.js**环境

安装**node.js**是傻瓜式安装的，成功安装是会自动配置环境变量的

**npm** 是**node.js**自带的包管理理工具

安装完毕后去命令行执行n**ode -v** 出现版本号即为安装成功执行**npm -v** 出现版本号即为安装成功

## npm install webpack -g 全局安装webpack

如果用不不习惯也可以下载**yarn**，使用**npm install yarn -g**全局下载

**yarn -v**查看版本号

也可以去安装国内的淘宝镜像 **cnpm**

[参考](https://blog.csdn.net/qq_26562641/article/details/72235585)**[node.js](https://blog.csdn.net/qq_26562641/article/details/72235585)**[安装博客地址](https://blog.csdn.net/qq_26562641/article/details/72235585)

**[node.js](http://nodejs.cn/download/)**[官网下载地址](http://nodejs.cn/download/)

第三节 开始安装官方**create-react-app**脚手架并搭建项目

**[react](https://reactjs.org/)**[官网](https://reactjs.org/)

**[create-react-app](https://github.com/search?q=create-react-app)**[脚手架](https://github.com/search?q=create-react-app)全局安装命令

npm isntall -g create-react-app

或者

yarn add create-react-app -g

安装后查看版本 **create-react-app --version**

开始创建项目 **create-react-app xiaodi**

创建项目的同时会帮你自动下载依赖，所以不用自己**npm install**安装依赖了，创建完成后直接 **cd xiaodi**去到当前项目执行**npm start**或**yarn start**

# 第四节 分析项目目录架构并重写一遍搭建的项目

### 项目目录架构

xiaodi

|  |  |  |
| --- | --- | --- |
| !"" | README.md | 项目说明文档 |
| !"" | node\_modules | 依赖文件夹 |
| !"" | package.json | npm依赖 |
| !"" | .gitignore |  |
| !"" | public | 静态资源 |
| # | !"" favicon.ico |  |
| # | !"" index.html |  |
| # | $"" manifest.json |  |
| $"" | src | 源码 |
|  | !"" App.css |  |
|  | !"" App.js |  |
|  | !"" App.test.js | 测试 |
|  | !"" index.css |  |
|  | !"" index.js | 入口js |
|  | !"" logo.svg |  |
|  | $"" serviceWorker.js | 对PWA的支持 |

 愿景：**"**让编程不在难学，让技术与生活更更加有趣**"**

第二章 **React**基础知识讲解

第一节 深入理理解**react**和**react-dom**两个库

**React**的**api**是很少的，基本学习使用一次，以后就再也不用看文档了了，学习**react**，核心就是看**js**的功力

**React**设计之初就是使用**JSX**来描述**UI**，所以解耦了和**DOM**的操作**react**只做逻辑层

**react-dom**做渲染层，去渲染实际的**DOM**

### 备注：换到移动端，可以用别的渲染库

备注：这一节只讲概念没源码

第二节 深度剖析**JSX**的实质

**JSX**语法即**JS**和**html**的混合体，实际的核心逻辑就是用**js**去实现的常见的代码如下

ReactDOM.render(<App/>,document.getElementById('r oot'))

**JSX**的实质就是**React.createElement**的调用

[可以去官网体验](https://zh-hans.reactjs.org/)**[JSX](https://zh-hans.reactjs.org/)**[的写法](https://zh-hans.reactjs.org/)我们**react**里的写法

class HelloMessage extends React.Component { render() {

return (

<div>

Hello {this.props.name}

</div>

);

}

}

ReactDOM.render(

<HelloMessage name="Taylor" />, document.getElementById('hello-example')

);

**JSX**实际的写法

class HelloMessage extends React.Component {

render() {

return React.createElement( "div",

null, "Hello ",

this.props.name

);

}

}

ReactDOM.render(React.createElement(HelloMessage, {

name: "Taylor" }), document.getElementById('hello- example'));

### 备注：这一节只讲概念没源码

第三节 详细讲解**state**变量渲染和**setState**修改数据

在组件里里面我们通过**{}**在**JSX**里里面渲染变量量

如果数据需要修改，并且需要页面同时响应改变，那就需要把变量放在**state**里里面，同时使用**setState**修改

初始化状态**state**

// 初始化状态

this.state = { count: 0

};

更新状态使用**setState,**不能直接**this.state.count=xxx**

// 更更新状态

this.setState({

count: this.state.count + 1

});

### 注意事项

**setState**是异步的，底层设计同一个生命周期会批量操作更新**state**

**setState**第二个参数是一个可选参数，传入一个回调函数可以 获取到最新的**state**

当修改的**state**依赖上一次修改的**state**的值时，可使用以下这种方法修改

this.setState((prevState, prevProps)=>({

//prevState：上一次修改的状态state

//prevProps：上一次修改的属性props count: prevState.count + 1

}), () => {

//这里里可以获取到最新的state console.log(this.state.count);

});

第四节 精讲**props**属性传递

### 父组件向子组件传递属性利用**props**接收使用例子如下：

//父组件传入

<PropsDemo title="Tim老师教react"></PropsDemo>

//子组件使用

//class组件使用

<h1>{this.props.title}</h1>

//函数型组件使用function xxx(props){

return <h1>{props.title}</h1>

}

//解构赋值写法

function xxx({title}){ return <h1>{title}</h1>

}

第五节 实战必备之条件渲染与数据循环

条件渲染写法，一般使用三目表达式

//三目表达式写法

{this.state.showTitle?<h1>{this.props.title}

</h1>:null}

//优化上面三目表达式写法，先在render函数里里定义一个变量量装载 结果

let result=this.state.showTitle?<h1>

{this.props.title}</h1>:null

{result}

//直接使用if else写let result

if(this.state.showTitle){ result=(

<h1>this.props.title</h1>

)

}else{

result=null

}

{result}

数据循环渲染写法

class App extends React.Component{ constructor(props){

super(props) this.state = {

goods: [

{ title: 'html+css基础入门', price: 19.8},

{ title: 'js零基础阶级', price: 29.8},

{ title: 'vue基础入门', price: 19.8},

{ title: 'vue电商单页面项目实战', price:

39.8},

{ title: 'react零基础进阶单页面项目实战', price: 59.8},

]

}

}

render(){ return <div>

<ul>

{this.state.goods.map(good=>{ return <li key={good.title}>

<span>{good.title} </span>

<span>{good.price}元 </span>

</li>

})}

</ul>

</div>

}

}

第六节 详细讲解事件监听的实现

以点击事件为例例子，使用方法如下：

//小驼峰写法，事件名用{}包裹

<button onClick={}></button>

由于**react**的**this**指向问题，所以在事件绑定时要特别注意，否则会 出现**bug**

### 一般使用的事件绑定写法有三种

第一种利利用**bind**绑定，写法如下，这种比较少用

//在constructor里面利用bind绑定继承this，解决方法的

this指向问题constructor(props) {

super(props); this.showTitleFun =

this.showTitleFun.bind(this);

}

showTitleFun(){

//执行行某些操作this.setState({})

}

//在DOM元素上直接使用

<button onClick={this.showTitleFun}>不不显示

title</button>

第二种箭头函数写法

showTitleFun = () => {

//执行行某些操作this.setState({});

};

//在DOM元素上直接使用

<button onClick={this.showTitleFun}>不不显示title</button>

第三种直接使用箭头函数返回一个函数

showTitle(){

//执行行某些操作this.setState({});

};

<button onClick={() => this.showTitleFun()}>不显示

title</button>

第七节 **React**之样式的编写讲解

### 行行内样式写法

<img style={{ width: "100px",height:"100px" }} />

添加类名

<img className="img" />

添加属性

<img src={logo} />

第八节 深入剖析**React**实现双向数据绑定

### React实现input的双向数据绑定要点动态绑定value属性

//在state里里面定义一个变量绑定input的value属性this.state={

inputval:'我是input的初始值'

}

//然后在input里里动态绑定上面定义的变量量

<input

type="text" value={this.state.inputval}

/>

监听input的onChange事件

//定义onChange绑定的事件inputvalChange = e => {

this.setState({

text: e.target.value

});

}

//在input上绑定绑定inputvalChange到onChange上

<input

type="text" value={this.state.inputval} onChange={e =>

this.inputvalChange(e)}

/>

第九节 精讲**React**组件生命周期

**componentWillMount** 组件将要挂载

**componentDidMount** 组件已经挂载**componentWillReceiveProps** 父组件传递的属性有变化，做相应响应

**shouldComponentUpdate** 组件是否需要更更新，返回布尔值，优化点

**componentWillUpdate** 组件将要更更新**componentDidUpdate** 组件已经更更新**componentWillUnmount** 组 件 已 经 销 毁

### 愿景：**"**让编程不不在难学，让技术与生活更更加有趣**"**

第三章 **React**组件的写法

# 第一节 傻瓜组件和聪明组件的区别

### 傻瓜组件也叫展示组件

负责根据**props**显示页面信息聪明组件也叫容器器组件

负责数据的获取、处理理

分清楚展示组件和容器器组件的优势

分离工作组件和展示组件提高组件的重用性

提高组件可用性和代码阅读便便于测试于后续的维护

第二节 深入理解函数式组件

函数式组件是一种无状态组件，是为了创建纯展示组件，这种组件只负责根据传入的**props**来展示，不涉及到要**state**状态的操作

组件不会被实例化，整体渲染性能得到提升组件不能访问**this**对象

组件无法访问生命周期的方法

无状态组件只能访问输入的**props**，同样的**props**会得到同样的渲 染结果，不会有副作用

### 官方文档说：在大部分**React**代码中，大多数组件被写成无状态的组件，通过简单组合可以构建成其他的组件等，这种通过多个简单然后合并成一个大应用的设计模式被提倡。

具体写法如下

//这种写法是新建一个组件页面把组件暴露出去的写法

import React from 'react'

export default function xxx() {

return (

<div>

我是函数式组件

</div>

)

}

//这种写法是在页面内部创建组件不用给外部使用，只供页面内部使用function xxx() {

return (

<div>

我是函数式组件

</div>

)

}

第三节 详细讲解**class**组件的写法

**React.createClass**是**react**刚开始推荐的创建组件的方式，现在基本不会见到了了

**React.Component**是以**ES6**的形式来创建**react**的组件的，是**React**目前极为推荐的创建有状态组件的方式

### 在里面我们可以写入我们的状态、生命周期、构造函数等

具体使用如下所示

import React, { Component } from 'react'

export default class ConditionLoop extends

Component {

render() {

return (

<div>

</div>

)

}

}

 愿景：**"**让编程不在难学，让技术与生活更加有趣**"**

第四章 **React**组件化和**ui**库引入使用

第一节 引入使用**ant-design**组件库

[ant-design官网地址](https://ant.design/docs/react/use-with-create-react-app-cn)

安装**ant-design**：**npm install antd --save**

接下来试用一下**button**组件

import React, { Component } from 'react' import Button from 'antd/lib/button' import "antd/dist/antd.css"

export default class App1 extends Component { render() {

return (

<div>

<Button type="primary">我是antd的按钮</Button>

</div>

)

}

}

第二节 详细讲解配置**ant-design**按需加载

上一节课的使用**antd**例例子实际上加载了了全部的 **antd** 组件的样式

### （对前端性能是个隐患）

我们需要对**antd**进行行配置按需加载，需要对**create-react-app** 的默认配置进行自定义

### 需要更改我们的启动插件

引入 **react-app-rewired** 并修改 **package.json** 里的启动配置。由于新的 **[react-app-rewired@2.x](https://github.com/timarney/react-app-rewired#alternatives)** 版本的关系，你还需要安装 **[customize -cra](https://github.com/arackaf/customize-cra)**。

安 装 命 令 **yarn add react-app-rewired customize -cra**（如果还没安装**yarn**的可以先执行行**npm install yarn -g** 进行安装）

更更改**package.json**文件

/\* package.json -代表没改前的代码，+代表已经更更改的代

码\*/

"scripts": {

-

+

}

"test": "react-scripts test",

"test": "react-app-rewired test",

|  |  |  |
| --- | --- | --- |
| - | "start": | "react-scripts start", |
| + | "start": | "react-app-rewired start", |
| - | "build": | "react-scripts build", |
| + | "build": | "react-app-rewired build", |

### 然后在项目根目录创建一个默认配置，先不用写内容

**config-overrides.js**

用于修改

执行安装 **babel-plugin-import**插件**(**安装命令：**yarn add babel-plugin-import)**

修改**conﬁg-overrides.js**文件内容如下：

const { override, fixBabelImports } = require('customize-cra');

module.exports = override( fixBabelImports('import', {

libraryName: 'antd', libraryDirectory: 'es', style: 'css',

}),

);

### 到这里就成功了（修改配置文件记得重启项目才生效），可以 移除上一节课全量引入**antd.css**的代码了，然后更改引入组件方式

import { Button } from 'antd';

第三节 性能优化之**PureComponent**讲解

**PureComponent**是内部定制了了**shouldComponentUpdate**生命 周期的**Component**

它重写了了一个方法来替换**shouldComponentUpdate**生命周期方法

平常开发过程中设计组件能使用**PureComponent**的地方都尽量量使用

想要使用**PureComponent**特性要记住两个小原则： 确保数据类型是值类型

### 如果是引用类型，确保地址不变，同时不应当有深层次数据变

化

使用**PureComponent**可以省去**shouldComponentUpdate**生命 周期的代码，代码会简单很多

|  |
| --- |
| import React, { Component,PureComponent } from 'react'   class Count extends Component {        render() {          console.log("我是条数")          return (              <div>                  条数：{this.props.count}              </div>          )      }  }   // 解决方案2.使用PureComponent去代替生命周期  class Title extends PureComponent {  //  class Title extends Component {       /\*       // 解决方案1.使用shouldComponentUpdate        shouldComponentUpdate(PreProps){          console.log('标题是否更新')          // 返回ture就是更新          return this.props.title !== PreProps.title      }       \*/        render() {          //当只是count变化时，count组件渲染，title也会重新渲染，影响性能          // 解决方案1.使用shouldComponentUpdate          // 解决方案2.使用PureComponent          console.log("标题")          return (              <div>                  标题：{this.props.title}                </div>          )      }  }  export default class Purememo extends Component {      constructor(){          super()          this.state={              title: "易烊千玺",              count:1          }          // 每2秒count +1            setInterval(()=>{              this.setState({                  count:this.state.count+1              })          },2000)      }      render() {          return (              <div>                  <h2>PureComponent的演示</h2>                  <Title title={this.state.title}></Title>                  <Count count={this.state.count}></Count>              </div>          )      }  } |

第四节 性能优化之**React.memo**讲解

**React.memo**是一个高阶组件的写法

**React.memo**让函数组件也拥有了了**PureComponent**的功能 使用例例子如下：

const MemoComponent = React.memo((props) => { return (

<div>

<p>{props.xxx}</p>

<p>{props.xxx}</p>

</div>

);

});

第五节 **React**高级使用之组件复合写法

### **React**官方说任何一个能用组件继承实现的用组件复合都可以实现，所以可以放心的去使用

组件复合类似于我们在**vue**框架里里面用的组件插槽具体使用方式如下：

//XdDialog

function XdDialog(props) { return (

<div style={{ border: `4px solid ${props.color

|| "blue"}` }}>

{/\* 等同于vue中匿匿名插槽 \*/}

{props.children}

{/\* 等同于vue中具名插槽 \*/}

<div className="abc">{props.footer}</div>

</div>

);

}

function WelcomeXdDialog() {

const confirmBtn = (

<button onClick={() => alert("React真好玩")}>确定

</button>

);

return (

<XdDialog color="green" footer={confirmBtn}>

<h1>欢迎来到小D课堂</h1>

<p>欢迎您来小D课堂学习react！！！</p>

</XdDialog>

);

}

愿景：**"**让编程不不在难学，让技术与生活更更加有趣**"**

第五章 高阶组件**(HOC)**

# 第一节 高阶组件初体验

## 高阶组件—HOC(Higher-Order Components)

### 高阶组件是为了了提高组件的复用率而出现的，抽离出具有相同逻辑或相同展示的组件

高阶组件其实是一个函数，接收一个组件，然后返回一个新的组 件，返回的这个新的组件可以对属性进行行包装，也可以重写部分生命周期

使用例例子如下：

//创建withLearnReact高阶组件，传递一个组件进去，返回一个新的组件NewComponent

const withLearnReact= (Component) => { const NewComponent= (props) =>{

return <Component {...props} name="欢迎大家来小D课

堂学习React"/>

}

return NewComponent

}

第二节 讲解高阶组件的链式调用

使用情况如下：

编写一个高阶组件进行行属性的添加编写一个高阶组件编写生命周期

然后将以上两个高阶组件进行行链式调用

使用例例子如下

import React, { Component } from 'react'

//编写第一个高阶组件。传递一个组件进去，返回一个新的组件 （返回的是函数组件）

const withLearnReact=(Comp)=>{ const NewComponent=(props)=>{

return <Comp {...props} name="欢迎大家来到小D课

堂学习React"></Comp>

}

return NewComponent

}

//编写第二个高阶组件，重写生命周期，注意，重写生命周期需要class组件 （返回的是class组件）

const withLifeCycle= Comp => {

class NewComponent extends Component{

//重写组件的生命周期componentDidMount(){

console.log('重写componentDidMount生命周

期')

}

render(){

return <Comp {...this.props}></Comp>

}

}

return NewComponent

}

class HOC extends Component { render() {

return (

<div>

<h1>欢迎大家体验高阶组件的写法</h1>

{this.props.title}

<p>姓名：{this.props.name}</p>

</div>

)

}

}

export default withLifeCycle(withLearnReact(HOC))

# 第三节 实现高阶组件装饰器写法

### 由于高阶组件链式调用的写法看起来比较的麻烦也不不好理理解。逻辑会看的比较绕

**ES7**中就出现了了装饰器器的语法，专门拿来处理理这种问题的安装支持装饰器器语法的**babel**编译插件



**npm install --save-dev @babel/plugin-proposal-**

**decorators**

更改配置文件代码

const {

override,

fixBabelImports, // 按需加载配置函数

addBabelPlugins // babel插件配置函数

} = require('customize-cra')

module.exports = override(

addBabelPlugins( // 支持装饰器器

[

'@babel/plugin-proposal- decorators',

{

legacy: true

}

]

),

fixBabelImports('import', { // antd 按需加载

libraryName: 'antd', libraryDirectory: 'es', style: 'css'

})

)

使用方式**@**高阶组件名称

高阶组件的声明要放在使用前面

第四节 详细讲解组件通信之上下文**(context)**

上下文**context**有两个角色

**Provider** 数据提供

**Consumer** 数据读取

使用**context**可以避免通过中间元素传递**props**，**context**的设计目 的是为了了共享对于一个组件树而言是**“**全局**”**的数据

不不使用**context**的情况下的代码：

import React, { Component } from "react";

//创建一个传递的数据源let store={

name:"我是Tim",

from:"我来自小D课堂"

}

class Info extends Component{ render(){

return (

<div>

<p>姓名：{this.props.name}</p>

<p>出处：{this.props.from}</p>

</div>

)

}

}

function ToolBar(props){ return (

<div>

<Info {...props}></Info>

</div>

)

}

export default class Context1 extends Component{ render(){

return (

<ToolBar name={store.name} from=

{store.from}></ToolBar>

)

}

}

使用**context**，避免了了中间**props**元素的传递的写法

import React, { Component } from "react";

// 1.创建上下文

const XdContext=React.createContext() const {Provider,Consumer}=XdContext

//创建一个传递的数据源let store={

name:" 我 是 Tim1", from:"我来自小D课堂1"

}

class Info extends Component{ render(){

return (

<Consumer>

{

</p>

</p>

store => {

return (

<div>

<p>姓名：{store.name}

<p>出处：{store.from}

</div>

)

}

}

</Consumer>

)

}

}

function ToolBar(props){ return (

<div>

<Info></Info>

</div>

)

}

export default class Context1 extends Component{ render(){

return (

<Provider value={store}>

<ToolBar></ToolBar>

</Provider>

)

}

}

### 愿景：**"**让编程不不在难学，让技术与生活更更加有趣**"**

第六章 **React**函数式编程之**Hook**

第一节 隆隆重介绍**React Hooks**

***Hook*** 是 **React 16.8** 的新增特性。它可以让你在不不编写 **class** 的情况下使用 **state** 以及其他的 **React** 特性。

在我们继续之前，请记住 **Hook** 是：

### 完全可选的。 你无需重写任何已有代码就可以在一些组件中尝试 **Hook**。但是如果你不不想，你不不必现在就去学习或使用**Hook**。

**100%** 向后兼容的。 **Hook** 不不包含任何破坏性改动。现在可用。 **Hook** 已发布于 **v16.8.0**。

没有计划从 **React** 中移除 **class**。

**Hook** 不不会影响你对 **React** 概念的理理解。 恰恰相反，**Hook** 为已知的 **React** 概念提供了了更更直接的 **API**：**props**， **state**，**context**， **refs** 以及生命周期。接下来的学习我们会发现，**Hook** 还提供了了一种更更强大的方式来组合他们。

**React Hooks**解决了了什什么问题？

### 函数组件不不能使用**state**，一般只用于一些简单无交互的组件， 用作信息展示，即我们上面说的傻瓜组件使用，如果需要交互 更更改状态等复杂逻辑时就需要使用**class**组件了了

**React Hooks**让我们更更好的拥抱函数式编程，让函数式组件也能使用**state**功能，因为函数式组件比**class**组件更更简洁好用， 因为**React Hooks**的出现，相信未来我们会更更多的使用函数式组件

### 副作用问题

我们一般称数据获取、订阅、定时执行行任务、手动修改

**ReactDOM**这些行行为都可以称为副作用

由于**React Hooks**的出现，我们可以使用**useEﬀect**来处理理组件副作用问题，所以我们的函数式组件也能进行行副作用逻 辑的处理理了了

### 有状态的逻辑重用组件

1. 复杂的状态管理理
   1. 之前我们使用**redux**、**dva**、**mobx**第三方状态管理理器器来进行行复杂的状态管理理
   2. 现在我们可以使用**useReducer**、**useContext**配合使用实 现复杂状态管理理，不不用再依赖第三方状态管理理器器

### 开发效率和质量量问题

* 1. 函数式组件比**class**组件简洁，开发的体验更更好，效率更更高 同时应用的性能也更更好

[封装好的React Hooks，可以来这里里学习封装自定义hooks](https://www.hooks.guide/)

第二节 详细介绍新特性**useState**

**useState---** 组 件 状 态 管 理 理 钩 子 **useState**能使函数组件能够使用**state**

### 基本使用如下所示

const [state,setState]=useState(initState)

**state**是要设置的状态

**setState**是更更新**state**的方法，只是一个方法名，可以随意更更改

**initState**是初始的**state**，可以是随意的数据类型，也可以是回调函数，但是函数必须是有返回值

### 完整使用例例子如下所示

import React,{useState} from 'react'

const App=() => {

const [count,setState]=useState(0) return (

<div>

<div>你点击了了{count}次</div>

<button onClick={()=>setState(count+1)}>点击</button>

</div>

)

}

第三节 详细介绍新特性**useEﬀect**

**useEﬀect---**副作用处理理钩子

数据获取、订阅、定时执行行任务、手动修改**ReactDOM**这些行为都可以称为副作用。而**useEﬀect**就是为了处理这些副作用而生的

**useEﬀect** 也 是 **componentDidMount** 、 **componentDidUpdate**和**componentWillUnmount**这几个 生命周期方法的统一

**useEﬀect**的基本使用如下所示

useEffect(callback,array)

**callback:** 回调函数，作用是处理理副作用逻辑

**callback**可以返回一个函数，用作清理理

useEffect(() =>{

//副作用逻辑xxxxxx

return ()=>{

//清理理副作用需要清理理的内容

//类似于componentWillUnmount，组件渲染和组件卸载前执行行的代码

}

},[])

**array(**可选参数**)**：数组，用于控制**useEﬀect**的执行行

### 分三种情况

空数组(与数组中传入常量一致)，则只会执行行一次（即初次渲染**render**）**,**相当于

**componentDidMount**

非空数组，**useEﬀect**会在数组发生改变后执行行 不不填**array**这个数组，**useEﬀect**每次渲染都会执行行

### 完整使用例例子如下所示

import {useState,useEffect} from 'react'

const App=() => {

const [count,setState]=useState(0)

useEffect(() =>{

//更更新页面标题

document.title=`您点击了了${count}次了了哦`

},[count]) return (

<div>

<div>你点击了了{count}次</div>

<button onClick={()=>setState(count+1)}>点击</button>

</div>

)

}

第四节 详细介绍新特性**useContext**

### **context**就是用来更更方便便的实现全局数据共享的，但是由于他并不不 是那么好用，所以我们一般会使用第三方状态管理理器器来实现全局数据共享

**redux dva mobx**

**useContext(context)**是针对**context**上下文提出的一个**Hooks**提 出的一个**API**，它接受**React.createContext()**的返回值作为参 数，即**context**对象，并返回最近的**context**

使用**useContext**是不不需要再使用**Provider**和**Consumer**的

当最近的**context**更更新时，那么使用该**context**的**hook**将会重新渲 染

### 基本使用如下：

const

Context=React.createContext({age:'18',name:'jerry'})

//组件1

const AgeComp= () =>{

//使用useContext

const ctx =useContext(Context) return(

<div>年年龄：{ctx.age}岁</div

)

}

第五节 详细介绍新特性**useReducer**

**useReducer**是**useState**的一个增强体，可以用于处理理复杂的状态管理理

**useReducer**可以完全替代**useState**，只是我们简单的状态管理理用**useState**比较易易用，**useReducer**的设计灵感源自于**redux**的**reducer**

对比一下**useState**和**useReducer**的使用：

//useState的使用方法

const [state,setState]=useState(initState)

//useReducer的使用方法

//const

[state,dispatch]=useReducer(reducer,initState,initAc tion)

**useReducer**的参数介绍

**reducer**是一个函数，根据**action**状态处理理并更更新**state initState**是初始化的**state**

**initAction**是**useReducer**初次执行行时被处理理的**action**

返回值**state**，**dispath**介绍

**state**状态值

**dispatch**是更更新**state**的方法，他接受**action**作为参数**useReducer**只需要调用**dispatch(action)**方法传入**action**即可更更新**state**，使用如下

//dispatch是用来更更新state的，当dispatch被调用的时候， reducer方法也会被调用，同时根据action的传入内容去更更新state action 是 传 入 的 一 个 描 述 操 作 的 对 象 dispatch({type:'add'})

**reducer**是**redux**的产物，他是一个函数，主要用于处理理**action**， 然后返回最新的**state**，可以把**reducer**理理解成是**action**和**state**的转换器器，他会根据**action**的描述去更更新**state**，使用例例子：

(state,action) => Newstate

### 具体使用例例子

import React,{useReducer} from 'react'

const initState={count:0} //state的初始值

const reducer = (state,action) => {

//根据action的描述去更更新state switch(action.type){

//当type是reset时，重置state的值回到初始化时候的值

case 'reset':

return initState;

//当type的值是add时，让count+1 case 'add':

return {count:state.count+1};

//当type的值是reduce时，让count-1 case 'reduce':

return {count:state.count-1};

//当type不不属于上面的任意一个值，state不不做更更改，直接放回当前state

default: return state

}

}

export default function UseReducerComp(){ const

[state,dispatch]=useReducer(reducer,initState) return (

<div>

<p>当前数量量是：{state.count}</p>

<div><button onClick=

{()=>dispatch({type:'reset'})}>重置</button></div>

<div><button onClick=

{()=>dispatch({type:'add'})}>加一</button></div>

<div><button onClick=

{()=>dispatch({type:'reduce'})}>减一</button></div>

</div>

)

}

彩蛋：结合**useContext**会有无限种可能

**useContext**可以解决组件间的数据共享问题，而**useReducer** 可以解决复杂状态管理理的问题，如果把他们结合起来使用，就 可以实现**redux**的功能了了，意味着未来我们可以不不再依赖第三方状态管理理器器了了

第六节 详细讲解官网介绍额外的**Hooks**

**useMemo** 用于性能优化，通过记忆值来避免在每个渲染上执行行高开销的计算

### 适用于复杂的计算场景，例例如复杂的列列表渲染，对象深拷贝等 场景

使用方法如下

const memoValue =useMemo(callback,array)

**callback**是一个函数用于处理理逻辑

**array** 控制**useMemo**重新执行行的数组，**array**改变时才会重新执行行**useMemo**

**useMemo**的返回值是一个记忆值，是**callback**的返回值使用方法如下

const obj1={taller:'180',name:'tom',age:'15'}

const obj2=

{taller:'170',name:'jerry',age:'18',sex:'女女'}

const

memoValue=useMemo(()=>Object.assign(obj1,obj2), [obj1,obj2])

//使用方式

<div>姓名：{memoValue.name}---年年龄：{memoValue.age}

</div>

### 不不能在**useMemo**里里面写副作用逻辑处理理，副作用的逻辑处理理放在

**useEﬀect**内进行行处理理

**UseCallback**和**useMemo**一样，也是用于性能优化的

### 基本使用方法

const memoCallback= useCallback(callback,array)

**callback**是一个函数用于处理理逻辑

**array** 控制**useCallback**重新执行行的数组，**array**改变时才会重新执行行**useCallback**

跟**useMemo**不不一样的是返回值是**callback**本身，而

**useMemo**返回的是**callback**函数的返回值使用方法如下

const obj1={taller:'180',name:'tom',age:'15'}

const obj2=

{taller:'170',name:'jerry',age:'18',sex:'女女'}

const

memoCallback=useCallback(()=>Object.assign(obj1,o bj2),[obj1,obj2])

//使用方式

<div>姓名：{memoCallback().name}---年年龄：

{memoCallback().age}</div>

**useRef** 方便便我们访问操作**dom**

### 使用方法如下

const UseRefComp=()=>{

//创建ref

const inputRef=useRef() const getValue= () => {

//访问ref

console.log(inputRef.current.value)

}

//挂载return (

<div>

<input ref={inputRef} type="text">

<button onClick={getValue}>获取input的

值</button>

</div>

)

}

第七节 自己动手封装一个自定义**Hooks**

### **Hooks**其实说到底就是一个封装好的钩子函数供我们调用

只是我们自己封装的时候要特别注重性能，重复渲染这些问题，官方封装的就比较完美

简单封装一个改变页面标题的 自定义**Hooks**

import React，{useEffect} from 'react'

//封装的Hooks用use开头

const useChangeTitle= (title) =>{ useEffect(()=>{

document.title=title

},[title])

}

exprot default (props)=>{ useChangeTitle("自定义修改标题Hooks") return (

<div>

测试自定义Hooks

</div>

)

}

第八节 深入讲解**React Hooks**的使用规则

只在顶层调用**Hooks**

**Hooks**的调用尽量量只在顶层作用域进行行调用

### 不不要在循环，条件或者是嵌套函数中调用**Hook**，否则可能会无 法确保每次组件渲染时都以相同的顺序调用**Hook**

只在函数组件调用**Hooks**

**React Hooks**目前只支持函数组件，所以大家别在**class**组件或者普通的函数里里面调用**Hook**钩子函数

**React Hooks**的应用场景如下

### 函数组件

自定义**hooks**

在未来的版本**React Hooks**会扩展到**class**组件，但是现阶段不不能再**class**里里使用

### 愿景：**"**让编程不不在难学，让技术与生活更更加有趣**"**

第七章 强大的状态管理理器器**Redux**

第一节 深度学习**Redux**成员及其数据流

## actions

**actions**其实是描述操作的对象，我们调用**dispatch**时需要传入此对象

## store

### **store**是整个应用的数据存储仓库，把我们全局管理理的状态数据 存储起来

它就是我们的后勤勤保障，是我们打仗的政委，专门管理理后勤勤数据

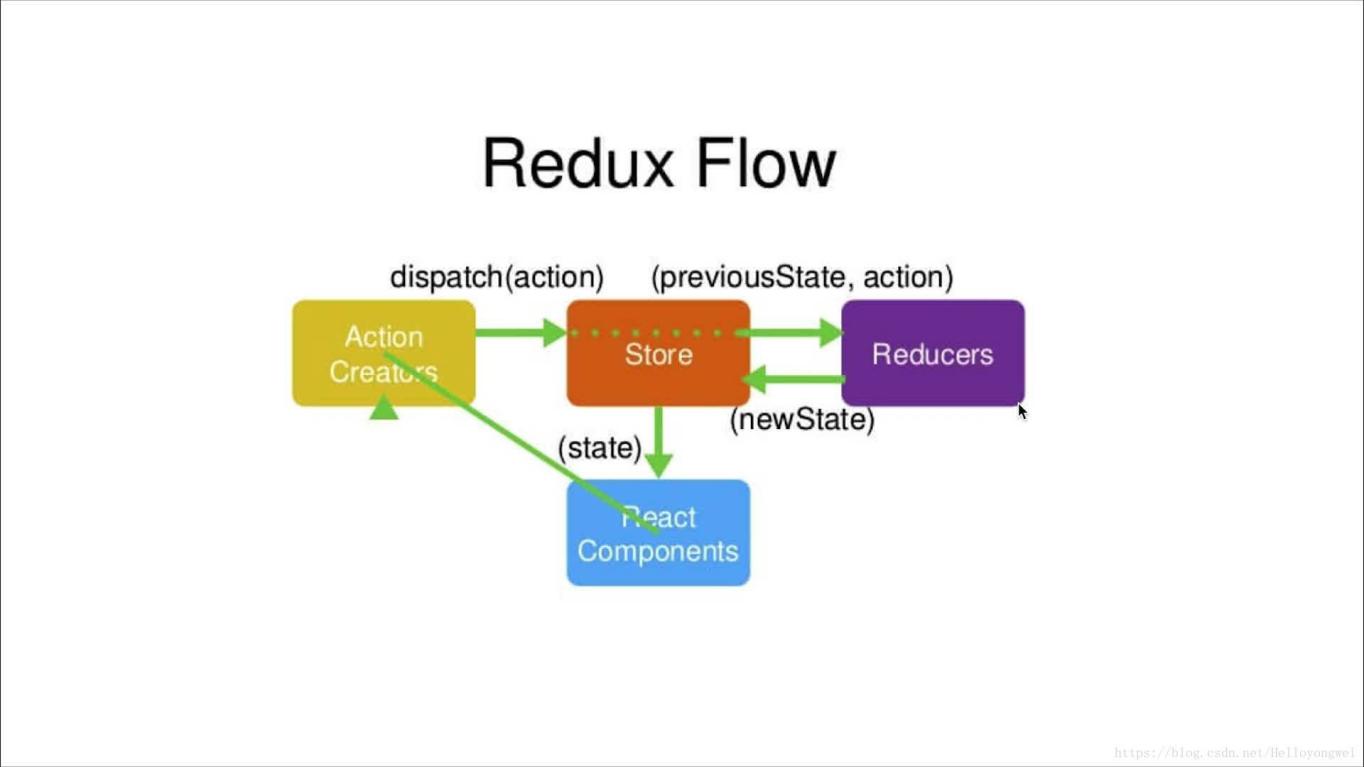
**reducers**

**reducers**接收**action**并更更新**store**

注意：**redux**是一个单独的数据流框架，跟**react**并没有直接的联 系，我们也可以在**JQ**在其他的复杂项目里里使用**redux**进行行数据管 理理，当我们不不知道是否应该是用**redux**的时候，我们都是不不需要 的，因为只有我们很肯定**redux**能帮助我们管理理好复杂项目数据流 的时候他才能发挥它的威力力，简单的项目我们只需要**state+props+context**就够了了

接下来看一下**Redux**数据流的走向





第二节 学习**redux**编写一个累加器器程序

## 安装redux npm install redux --save

编写使用**redux**的步骤**(**过程有点繁琐，别遗漏漏了了任何的一步**)**

1. 从 **redux** 引 入 **createStore** 用 来 创 建 数 据 仓 库 **store createStore**是一个函数，需要传入**reducer**作为参数，返

回值是我们需要的**store**

1. 在使用页面引入数据仓库**store**，

通过**getState()**方法可以获取到数据仓库里里的状态数据

## state

通过**dispatch(action)**可以触发更更改**reducer**函数

每次触发**dispatch**都会触发**store.subscribe()**方法，用来从新触发页面渲染

### 代码展示，对应以上步骤检验

**store.js**

import { createStore } from 'redux'

const fitstReducer = (state=0,action ) => { switch(action.type){

//当传入action的type为add的时候给state+1

case 'add' :

return state+1;

//当传入action的type为reduce的时候给state-1 case 'reduce' :

return state-1;

//当传入的都不不是以上的类型是返回旧的state default:

return state;

}

}

//创建数据仓库，把我们编写的reducer作为参数传入createStore const store=createStore(fitstReducer)

export default store

### FirstRedux.js

import React, { Component } from 'react' import store from './store'

export default class FirstRedux extends Component { render() {

return (

<div>

<h1>尝试使用redux编写一个累加器器</h1>

{/\* 通过getState方法获取数据仓库里里面的状

态数据state \*/}

{store.getState()}

<div>

<button onClick=

{()=>store.dispatch({type:'add'})}>加一</button>

<button onClick=

{()=>store.dispatch({type:'reduce'})}>减一</button>

</div>

</div>

)

}

}

### Index.js

import React from 'react'

import ReactDOM from 'react-dom'

import FirstRedux from './TryRedux/FirstRedux' import store from './TryRedux/store'

const render=()=>{ ReactDOM.render(<FirstRedux >

</FirstRedux>,document.getElementById('root'))

}

render()

store.subscribe(render)

第三节 深入学习**React**封装的**react-redux**进行行改造累加器器

由于**redux**的写法太繁琐，还每次都需要重新调用**render**，不不太符合我们了了解**react**编程

## react-redux横空出世，安装react-redux： npm install react- redux --save

**React-redux**提供了了两个**api**供我们使用**Provider** 顶级组件，功能为给我们提供数据

**connect** 高阶组件，功能为提供数据和方法

以下为使用**react-redux**改造累加器器的代码 只需留留意**index.js**和

**FirstRedux.js**，**store.js**暂时不不用作改变

## FirstRedux.js

import React, { Component } from 'react' import { connect } from 'react-redux'

//写一个返回数据的方法，供connect使用，connect会帮我们把数据转换成props

const mapStateToProps =(state) => { return {

count:state

}

}

//写一个返回dispatch方法的方法供connect使用，connect帮我们把dispatch转换成props

const mapDispatchToProps = dispatch =>{ return {

add:()=>dispatch({type:'add'}), reduce:()=>dispatch({type:'reduce'})

}

}

class FirstRedux extends Component { render() {

return (

<div>

<h1>尝试使用redux编写一个累加器器</h1>

{this.props.count}

<div>

<button onClick=

{()=>this.props.add()}>加一</button>

<button onClick=

{()=>this.props.reduce()}>减一</button>

</div>

</div>

)

}

}

export default connect(mapStateToProps,mapDispatchToProps) (FirstRedux)

### index.js

import

import import import import

React from 'react'

ReactDOM from 'react-dom'

FirstRedux

store from

{ Provider

from './TryRedux/FirstRedux'

'./TryRedux/store'

} from 'react-redux'

ReactDOM.render(

<Provider store={store}>

<FirstRedux ></FirstRedux>

</Provider> , document.getElementById('root'))

第四节 必备技能之高阶组件装饰器器模式进行行简化封装代码

**connect**高阶组件用装饰器器会使我们的代码看起来更更简洁易易懂 使用装饰器器进行行我们的代码优化

**FirstRedux.js**

import React, { Component } from 'react'

import { connect } from 'react-redux' @connect(

state=>({count:state}), dispatch=>({

add:()=>dispatch({type:'add'}), reduce:()=>dispatch({type:'reduce'})

})

)

class FirstRedux extends Component { render() {

return (

<div>

<h1>尝试使用redux编写一个累加器器</h1>

{this.props.count}

<div>

<button onClick=

{()=>this.props.add()}>加一</button>

<button onClick=

{()=>this.props.reduce()}>减一</button>

</div>

</div>

)

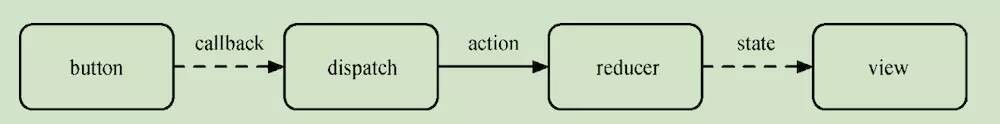
}

}

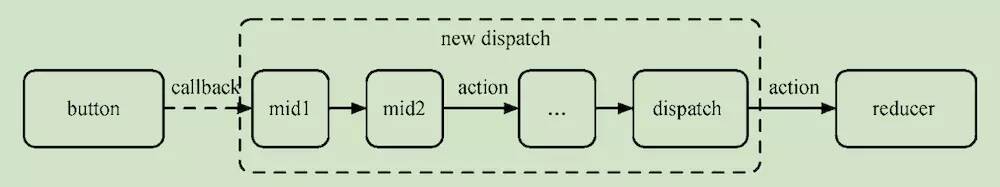
export default FirstRedux

第五节 深度剖析**redux**中间件给我们带来的帮助

由于**redux reducer**默认只支持同步，实现异步任务或者延时任务时，我们就要借助中间件的支持了了

没使用中间件时的**redux**数据流

使用了了中间件**middleware**之后的**redux**数据流

### 这节课我们就学习两个中间件

**redux-thunk** 支持我们**reducer**在异步操作结束后自动执行行安 装 **redux-thunk npm install redux-thunk --save**

**redux-logger** 打印日志记录协助本地调试

## 安装redux-logger npm install redux-logger --save

### 使用中间件例例子

import { createStore,applyMiddleware } from 'redux' import logger from 'redux-logger'

import thunk from 'redux-thunk'

const fitstReducer = (state=0,action ) => { console.log(action) switch(action.type){

//当传入action的type为add的时候给state+1

case 'add' :

return state+1;

//当传入action的type为reduce的时候给state-1 case 'reduce' :

return state-1;

//当传入的都不不是以上的类型是返回旧的state default:

return state;

}

}

//创建数据仓库，把我们编写的reducer作为参数传入createStore

//有一个注意点就是logger最好放在最后，日志最后输出才不会出bug，因为中间件时按顺序执行行

const store=createStore(fitstReducer,applyMiddleware(thunk

,logger))

export default store

第六节 详细讲解抽离**reducer**和**action**进行行统一管理理

第一步新建一个**count.redux.js**存放我们的**reducer**和**action count.redux.js**

//把reducer和action抽离出来再同一个文件下进行行维护

const fitstReducer = (state=0,action ) => { console.log(action) switch(action.type){

//当传入action的type为add的时候给state+1

case 'add' :

return state+1;

//当传入action的type为reduce的时候给state-1 case 'reduce' :

return state-1;

//当传入的都不不是以上的类型是返回旧的state default:

return state;

}

}

const add=()=>({type:'add'})

const reduce=()=>({type:'reduce'}) const asyncAdd=()=> dispatch =>{

setTimeout(()=>{ dispatch({type:'add'})

},1000)

}

export {fitstReducer, add,reduce,asyncAdd}

## index.js

import React from 'react'

import ReactDOM from 'react-dom'

import FirstRedux from './TryRedux/FirstRedux' import { Provider } from 'react-redux'

import { createStore ,applyMiddleware} from 'redux' import thunk from 'redux-thunk'

import logger from 'redux-logger'

import {fitstReducer} from './TryRedux/count.redux'

const store=createStore(fitstReducer,applyMiddleware(thunk

,logger)) ReactDOM.render(

<Provider store={store}>

<FirstRedux ></FirstRedux>

</Provider> , document.getElementById('root'))

## FirstRedux.js

import React, { Component } from 'react' import { connect } from 'react-redux'

import {add,reduce,asyncAdd} from './count.redux' @connect(

state=>({count:state}),

{add,reduce,asyncAdd}

)

class FirstRedux extends Component { render() {

return (

<div>

<h1>尝试使用redux编写一个累加器器</h1>

{this.props.count}

<div>

<button onClick=

{()=>this.props.add()}>加一</button>

<button onClick=

{()=>this.props.reduce()}>减一</button>

<button onClick=

{()=>this.props.asyncAdd()}>延时加1</button>

</div>

</div>

)

}

}

export default FirstRedux

到这一步我们就已经重构完了了，抽离了了**reducer**和**action**。之前的

**store.js**已经可以删除了了

### 愿景：**"**让编程不不在难学，让技术与生活更更加有趣**"**

第八章 页面连接器器之路路由**react- router(**版本**4.x)**

第一节 介绍及安装使用**react-router**

安装**react-router** 我们学习的时候安装的路路由是针对浏览器器的

## npm install react-router-dom --save

### [学习网站](https://reacttraining.com/react-router/)特点：

秉承**react**一切皆组件，路路由也是组件 分布式的配置，分布在你页面的每个角落包含式配置，可匹配多个路路由

第二节 体验**react-router**的写法

使用**react-router**步骤

### 引入顶层路路由组件包裹根组件

import React from 'react'

import { BrowserRouter} from 'react-router-dom' export default function RouterSample() {

return (

<div>

<h1>学习react路路由</h1>

<BrowserRouter>

<App></App>

</BrowserRouter>

</div>

)

}

引入**Link**组件编写路路由导航

**<Link to="/">**首页**</Link>**

**to** 指定跳转去的路路径

引入**Route**组件编写导航配置

**<Route exact path="/" component={Home}></Route>**

**path** 配置路路径

**component** 配置路路径所对应的组件

**exact** 完全匹配，只有路路径完全一致时才匹配编写导航配置对应的**component**组件

第三节 学习**react-router**的路路由传参取参

### 用得比较多的两种传参取参方式

1. 声明式导航路路由配置时配置路路由参数

配置

**<Route path="/detail/:course" component=**

**{Detail}></Route>**

传递

**<Link to="/detail/react">react</Link>**

获取

解构路路由器器对象里里的**match**出来**(macth**获取参数信息**)**

**{match.params.course}**

### 编程式导航式传递参数与获取

解构路路由器器对象获取到导航对象 **history(**用作命令式导航**)**

通过事件执行行**history.push({ pathname: "/", state:**

**{ foo: "bar" } })** 传递的参数装载在**state**里里面从路路由信息解构出**location(**当前的**url**信息**)**对象**location.state** 进行行获取

# 第四节 深入学习**react**嵌套路路由及路路由重定向

### 嵌套路路由写法如下

一级组件

function App(){ return (

<div>

{/\* 路路由导航 \*/}

<ul>

<li>

<Link to="/mine">我的</Link>

</li>

</Route>

)

}

</ul>

{/\* 路路由配置 \*/}

<Route path="/mine" component={Mine}>

</div>

### 二级组件嵌套在一级组件里里进行行显示

function Mine(){ return (

<div>

<h2>个人中心</h2>

<ul>

<li>

<Link to="/mine/userinfo">个人

信息</Link>

</li>

<li>

<Link to="/mine/order">客户订单

</Link>

</li>

</ul>

<Route path="/mine/userinfo" component={()=>(<div>个人信息</div>)}></Route>

<Route path="/mine/order" component=

{()=>(<div>客户订单</div>)}></Route>

<Redirect to="/mine/userinfo">

</Redirect>

</div>

)

}

注意点：嵌套的子路路由需要跟随父路路由且不不设置确切匹配

例例子

父路路由 **/mine**

子路路由 **/mine/xxx**

# 第五节 深入剖析路路由守卫的实现与使用

### 路路由守卫其实就是我们的路路由拦截，当我们有一些页面需要登录之后才有权限去访问这个时候我们的路路由守卫就可以派上用场了了**React**里里的路路由守卫其实也是一个组件，最后返回的还是**Route**组件

 愿景：**"**让编程不不在难学，让技术与生活更更加有趣**"**

第九章 **redux-saga**与**Generator**生成器器

第一节 介绍**redux-saga**和**redux-thunk**的不不同

**redux-saga**是一个用于管理理**redux**应用异步操作的中间件， **redux-saga**通过创建**sagas**将所有异步操作逻辑收集在一个地方集 中处理理，可以用来代替**redux-thunk**中间件

### 工作成员分工：

**reducer**负责处理理**action**的**stage**更更新**sagas**负责协调那些复杂或者异步的操作

**redux-saga**可以处理理各种复杂的异步操作，**redux-thunk**适用于 简单的异步操作场景

**redux-saga**使用**generator**解决异步问题，使用同步方式编写异步 代码，解决回调地域问题

**redux-thunk**可以接受**function**类型的**action**，而**redux-saga**则是纯对象**action**的解决方案

安装**redux-saga**： **npm install redux-saga -S**

第二节 应用**redux-saga**改造路路由守卫登录认证

首先我们编写一个管理理登录的**reducer**和**action**的文件

## user.redux.js

**user.redux.js(**安装了了之前的插件之后进入文件按**rxr**然后回车可以快速创建模板**)**

//初始化state

const initialState = { isLogin:false

}

//reducer

export default (state = initialState, action) => { switch (action.type) {

case 'login':

return { isLogin:true };

default:

return state;

}

}

//redux-saga使用生成action函数const login=function(){

return {type:'login\_request'}

}

export {login}

编写**sagas.js**供我们捕获**action**创建函数返回的**action**

import { call, put, takeEvery } from "redux- saga/effects"

//模拟登陆接口const api = {

login() {

//返回一个promise对象

return new Promise((resolve, reject) => {

//模拟异步登陆setTimeout(() => {

//通过随机数模拟会造成登陆成功和登陆失败概

率各占一半

误 " })

if (Math.random() > 0.5) {

//登陆成功，返回用户id和名字resolve({ id: 1, name: 'tim' })

} else {

//登陸失敗

reject({ message: "用户名或密码错

}

}, 1000)

})

}

}

//编写真正工作的saga ES6的星星函数function\* login(action) {

try {

//call调用异步函数

Generator生成器器

const result = yield call(api.login)

//put派发action，触发reducer

yield put({ type: 'login', result })

} catch (error) {

yield put({ type: 'login\_failed', message: error.message })

}

}

function\* mySaga() {

//事件监听，监听到action来了了就触发上面真正工作的saga yield takeEvery('login\_request', login)

}

export default mySaga

### 创建及应用中间件，把数据源暴暴露露出去

import { createStore, applyMiddleware, combineReducers

} from 'redux'

import logger from 'redux-logger'

import { firstReducer as count } from "./count.redux" import user from "./user.redux"

import createSagaMiddleware from 'redux-saga' import saga from './sagas'

//第一步、创建我们的中间件

const mid = createSagaMiddleware()

//第二部，应用中间件

const store = createStore(

//当我们reducer多的时候我们要做reducer模块化combineReducers({ count, user }), applyMiddleware(mid, logger)

)

//第三部，执行行saga，把监听事件跑起来mid.run(saga)

export default store

最后回到路路由组件**RouterSample.js**里里面进行行修改**(**这里里就不不贴代 码了了**)**

1. 引入**react-redux**的**connect**和**Provider**和其他需要用到的数 据及包
2. 把原来的组件内虚拟的**state**和**auth**状态及登录方法替换成

**redux**内的

|  |
| --- |
| import React,{ Component } from 'react'  import { BrowserRouter as Router, Link, Switch ,Route, Redirect } from 'react-router-dom'  import { connect, Provider } from 'react-redux'  import store from './store'  import {login} from './user.redux'  function App() {      return(       <>          <ul>              <li>                  <Link to='/'>首页</Link>              </li>              <li>                  <Link to='/classes'>课程</Link>              </li>              <li>                  <Link to='/me'>我的</Link>              </li>          </ul>          <Switch>              <Route exact path='/' component={Home}></Route>              <Route path='/classes' component={Classes}></Route>              {/\* <Route path='/me' component={Me}></Route> \*/}              <Route path='/login' component={Login}></Route>              <RouterGuard path='/me' component={Me}></RouterGuard>              <Route path='/detail/:course' component={Detail}></Route>              <Route component={NotFount}></Route>          </Switch>         </>      )  }  // 路由守卫  @connect( state=>({isLogin:state.isLogin}))  class RouterGuard extends Component{        render(){          const { component:Component, ...otherProps } = this.props          return(              <Route {...otherProps} render={(prop)=>(this.props.isLogin? <Component {...prop}></Component> : <Redirect to={{pathname:'/login',from:prop.location.pathname}} ></Redirect>)} ></Route>          )      }  }  @connect(      state=>({isLogin:state.isLogin}),      {login}  )  class Login extends Component{      render(){          // console.log(this.props.location.from)          const from = this.props.location.from ? this.props.location.from : '/'          if(this.props.isLogin){              return <Redirect to={from}></Redirect>          }          return(              <>               <div> 请登录</div>               <button onClick={this.props.login}>登录</button>              </>          )      }  }    export default function RouterEx() {      return (          <Router>              <Provider store={store}>                  <App>                    </App>              </Provider>            </Router>        )  } |

第三节 详解**generator**函数的原理理和使用**(**一**)**

function\* g() { yield "a";

yield "b";

yield "c"; return "ending";

}

console.log(g()); //返回迭代器器Iterator

const gen = g();

// console.log(gen.next()) // 返回结果对象

// // { value: 'a', done: false }

// // value是yield后面表达式的结果

// console.log(gen.next())

// console.log(gen.next())

// console.log(gen.next())

// 使用递归函数执行行生成器器里里面所有步骤

function next(){

let { value, done } = gen.next() // 启 动console.log(value) // 依次打印输出 a b c end if(!done) next() // 直到迭代完成

}

next()

第四节 详解**generator**函数的原理理和使用**(**二**)**

### 例例子一

function\* say() { let a = yield '1' console.log(a) let b = yield '2' console.log(b)

}

let it = say() // 返回迭代器器

console.log(it.next())

// 输 出 { value: '1', done: false }

// a的值并非该返回值，而是下次next参数

console.log(it.next('我是被传进来的1'))

// 输出'我是被传进来的1'

// 输出{ value: '2', done: false }

console.log(it.next('我是被传进来的2'))

// 输出'我是被传进来的2'

// 输出{ value: undefined, done: true }

例例子二 结合**promise**

// 使用Generator顺序执行行两次异步操作

function\* r(num) {

const r1 = yield compute(num); yield compute(r1);

}

// compute为异步操作，结合Promise使用可以轻松实现异步操作队列列

function compute(num) {

return new Promise(resolve => { setTimeout(() => {

const ret = num \* num; console.log(ret); // 输出处理理结果resolve(ret); // 操作成功

}, 1000);

});

}

// 不不使用递归函数调用

let it = r(2);

// {value:Promise,done:false}

// it.next().value.then(num => it.next(num));

// 修改为可处理理Promise的next function next(data) {

let { value, done } = it.next(data); // 启 动

if (!done) { value.then(num => {

next(num);

});

}

}

next();

### 愿景：**"**让编程不不在难学，让技术与生活更更加有趣**"**

第十章 开始使用**umi**开发框架

第一节 介绍企业级 **react** 应用框架**umi**

[umi学习](https://umijs.org/zh/guide/#%E7%89%B9%E6%80%A7)

**umi**特性 @3.2.3版本

### 开箱即用，内置 react、react-router 等

 类 **next.js** 且功能完备的路路由约定，同时支持配置的路路由方

### 式

 完善的插件体系，覆盖从源码到构建产物的每个生命周期高性能，通过插件支持 PWA、以路路由为单元的 code

splitting 等

 支持静态页面导出，适配各种环境，比如中台业务、无线业务、[egg](https://github.com/eggjs/egg)、支付宝钱包、云凤蝶等

开发启动快，支持一键开启 [dll](https://umijs.org/zh/plugin/umi-plugin-react.html#dll) 等

 一键兼容到 **IE9**，基于 [umi-plugin-polyﬁlls](https://umijs.org/zh/plugin/umi-plugin-react.html#polyfills)

 完善的 **TypeScript** 支持，包括 d.ts 定义和 umi test

### 与 **dva** 数据流的深入融合，支持 duck directory、model

的自动加载、code splitting 等等

第二节 详细了了解**dva**是什什么以及它与**umi**的约定

**dva**是一个**React** 应用框架，将**Redux**、**Redux-saga**、**React- router**三个 **React** 工具库包装在一起

路路由： **[React-Router](https://github.com/ReactTraining/react-router/tree/v2.8.1)**

架构： **[Redux](https://github.com/reactjs/redux)**

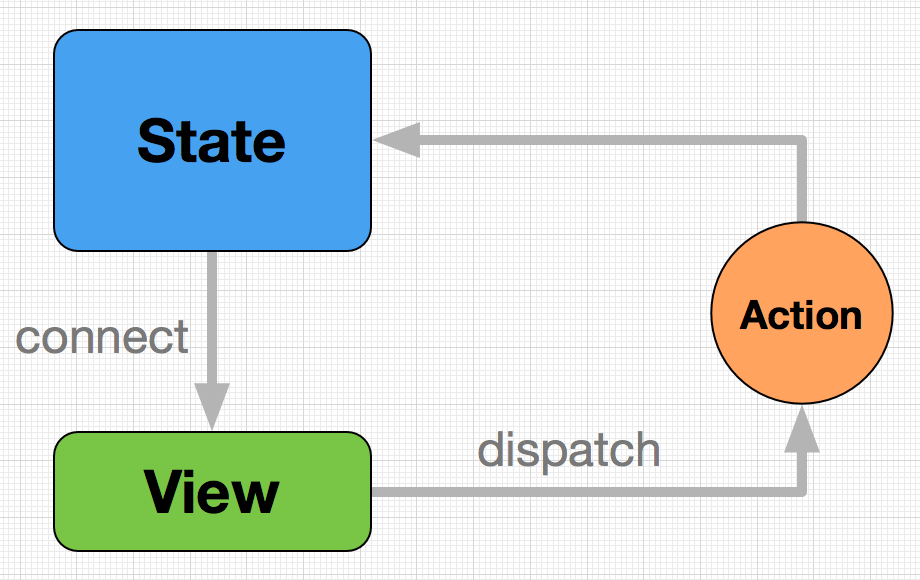
### 异步操作： [Redux-saga](https://github.com/yelouafi/redux-saga)

**dva**是目前**react**最流行行的数据流解决方案（**dva = React-Router**

## + Redux + Redux-saga）

### 数据流向图解





**State**：一个对象，保存整个应用状态

**View**：**React** 组件构成的视图层

**Action**：一个对象，描述事件

**connect** 方法：一个函数，绑定 **State** 到 **View dispatch** 方法：一个函数，发送 **Action** 到 **State dva**与**umi**的约定

**src** 源码

**pages** 页面**components** 组件**layout** 布 局**model** 数据模型

**conﬁg** 配置

**mock** 数据模拟

**test** 测试

全局安装**umi**

## npm install umi -g或yarn global add umi

第三节 使用**umi**开发项目并快速新建页面

### 先新建一个文件夹**(**不不要使用中文**)**

执行行**npm init**生成**package.json**文件配置项**scripts**更更改为如下

"scripts": {

"start": "umi dev", "build": "umi build"

},

按照规范新建一个**src**文件夹

进入到**src**目录下然后执行行**umi g page index**

再创建一个**about**页面**umi g page about**

### 约定式路路由初体验

访问**index**页面**: http://localhost:8000/**

访问**about**页面**: http://localhost:8000/about**

第四节 详细讲解**umi**里里面的嵌套路路由与动态路路由

如果出现**\_layout.js**页面默认为父组件页面

通过属性的**{props.children}**显示子组件内容 动态路路由文件命名为[ ]开头，例例子如下

**users**文件夹下的[**name].js** 即路路由配置为**/users/:name**

# 第五节 创建配置文件编写配置式路路由

配置式路路 由一旦创建，约定式路路 由将会失效创建**conﬁg**文件夹，创建**conﬁg.js**文件

import{

import { defineConfig } from 'umi'

export default defineConfig({

//路由配置  
  routes: [{ path: '/login',component: './login'},

        {path: '/',component: './index'}, //路径是相对于pages

        {

            path: '/about',

            component: './about',

            wrappers: ['./wrappers/auth'] //路由级别的权限校验

        },

        {

            path: '/users',

            component: './users/\_layout',

            routes: [{

                    path: '/users',

                    component: './users/index'

                },

                {

                    path: '/users/:name',

                    component: './users/[name]'

                }, //动态路由的写法

            ]

        },

        {

            component: './notfound'

        }

    ],

antd: {}  //配置ant-design

})

# 第六节 讲解配置式路路 由如何加入路路 由守卫

加入配置项**Routes(**注意点如下**)**

### 加入的路路 由守卫组件路路 径相对于根目录后缀名不能省略

创建路由守卫组件 wrappers

|  |
| --- |
| config.js |

第七节 讲解在**umi**里里如何引入**antd**并使用

安装**antd**： **npm install antd -S**

安装@umijs/preset-react：

## **npm install** @umijs/preset-react **-D**

安装好之后配置**conﬁg**文件夹下的**conﬁg.js**文件增加以下配置

|  |
| --- |
|  |

修改package.json （圈出来的@umijs/preset-react都去掉）

|  |
| --- |
|  |

第八节 在**umi**开发框架里里面引入**dva**进行行开发

**dva**主要是软件分层的概念

### **Page**负责与用户直接打交道：渲染页面、接受用户的操作输入，侧重于展示型和交互逻辑

**Model**负责处理理业务逻辑，为**Page**做数据、状态的读写等操作**Service**主要负责跟**HTTP**做接口对接，跟后端做数据交互，进 行行纯粹的数据读写

使用**dva**，安装@umijs/plugin-dva

npm install @umijs/plugin-dva -S

### **model**可以看成是一个维护页面数据状态的对象，大体结构如下

export default {

namespace: "goods", // model的命名空间，区分多个model state: [], // 初始状态

effects: {// 异步操作

},

reducers: {}

};

第九节 使用**dva**开发模式开发一个商品页面

请求接口需要安装**axios** ： **npm install axios -S**

**mock**数据

//因为本地mock没数据库，所以我们在这里里模拟数据let data=[

{title:'vue单页面电商项目'},

{title:'react单页面后台管理理项目'}

]

export default {

//method url 跟我们node编写的express是一样的

"get /api/goods" :function(req,res){ setTimeout(()=>{

res.json({result:data})

},1000)

}

}

编写**models**代码

import axios from 'axios'

//调接口，本应该写在servicer，但是由于这里里演示我们就先放在

model

function getGoods(){

return axios.get('/api/goods')

}

export default

namespace: model

state: [],

{

"goods", // model的命名空间，区分多个

// 初始状态

effects: {// 异步操作

\*getList(action,{call,put}){ const res=yield call(getGoods) yield

put({type:'initGoods',payload:res.data.result})

}

},

reducers: {

initGoods(state,action){ return action.payload

},

//添加商品addGood(state,action){

return [...state,

{title:action.payload.title}]

}

}

};

### 页面代码省略略，具体看项目源码

|  |
| --- |
| import styles from './goods.css';  import React, { Component } from 'react'  import { connect } from 'umi'  import { Card, Spin, Button } from 'antd'  @connect(    state=>({      goodsList:state.goods,  //获取指定的命名空间里面的state      loading:state.loading  //通过loading命名空间拿到我们的加载状态    }),    {      getList:()=>(        {type:'goods/getList'}  //action的type是需要命名空间为前缀+我们的reducer      ),      addGoods:(title)=>(        {type:'goods/addGoods',payload:{title}}      )    }  )  export default class extends Component {    componentDidMount(){      this.props.getList()    }    render() {      console.log(this.props.loading)      return (        <div>          <h1 className={styles.title}>Page goods</h1>          {/\* 商品列表 \*/}          <div>            {               this.props.loading.models.goods ?              <Spin className={styles.loadstatus} size='large' spinning={true}></Spin>              :              <div>                {                 this.props.goodsList.map((good,index)=>{                return <Card title={`课程${index+1}`} key={index}>                    <div>                      {good.title}                    </div>                </Card>              })              }              <div>                <Button type='primary' onClick={()=>this.props.addGoods("商品3"+new Date().getTime())}>添加商品                </Button>              </div>              </div>              }          </div>        </div>      );    }  } |

 愿景：**"**让编程不不在难学，让技术与生活更更加有趣**"**

第十一章 快速搭建一个后台管理页面框架

第一节 详细了了解如何从**antd**框架里面拿到自己想要的代码

[学习antd框架官网](https://ant.design/index-cn)

如果你想用框架快速的开发，一定要熟悉这些框架的api

由于我们已经安装了了antd并配置好了了，那么我们接下来使用的步骤在组件模块找到自己想用的组件

点击进入查看多种展示的组件形态，找到自己想要的查看自己想使用的组件样式和类型

展开查看代码使用和赋值代码使用图解如下



第二节 开始使用**antd**布局组件进行项目大框架的布局

如果使用约定式路路由，在**src**目录下新建一个**layout**文件夹，里里面的**index.js**将成为项目的布局页面

由于我们现在集中配置路路由了了，所以我们要去路路由配置配置一个嵌套路路由，使**layout**里里的**index.js**成为整个项目的最外层页面

从**antd**里里面复制选中的布局代码 放进**layout/index.js**里里

import React, { Component } from 'react' import { Layout, Menu, Breadcrumb, Icon } from 'antd';

import logo from "../../public/logo.png"

const { SubMenu } = Menu;

const { Header, Content, Sider } = Layout; import styles from './index.css'

export default class index extends Component { render() {

return (

<div>

<Layout>

<Header className="header">

<img src={logo} className=

{styles.logo} />

{['2']} '64px' }}

<Menu

theme="dark" mode="horizontal" defaultSelectedKeys=

style={{ lineHeight:

1</Menu.Item> 2</Menu.Item> 3</Menu.Item>

>

<Menu.Item key="1">nav

<Menu.Item key="2">nav

<Menu.Item key="3">nav

{styles.content}>

</Menu>

</Header>

<Layout className=

<Sider width={200} style={{

background: '#fff' }}>

<Menu

mode="inline"

{['1']}

{['sub1']}

'100%', borderRight: 0 }}

type="user" />

key="1">option1</Menu.Item> key="2">option2</Menu.Item> key="3">option3</Menu.Item> key="4">option4</Menu.Item>

defaultSelectedKeys= defaultOpenKeys= style={{ height:

>

<SubMenu

key="sub1" title={

<span>

<Icon

subnav 1

</span>

}

>

<Menu.Item

<Menu.Item

<Menu.Item

<Menu.Item

type="laptop" />

</SubMenu>

<SubMenu

key="sub2" title={

<span>

<Icon

subnav 2

</span>

}

key="5">option5</Menu.Item> key="6">option6</Menu.Item> key="7">option7</Menu.Item> key="8">option8</Menu.Item>

>

<Menu.Item

<Menu.Item

<Menu.Item

<Menu.Item

type="notification" />

</SubMenu>

<SubMenu

key="sub3" title={

<span>

<Icon

subnav 3

</span>

>

key="9">option9</Menu.Item> key="10">option10</Menu.Item> key="11">option11</Menu.Item> key="12">option12</Menu.Item>

}

<Menu.Item

<Menu.Item

<Menu.Item

<Menu.Item

paddingLeft: '24px' }}>

</SubMenu>

</Menu>

</Sider>

<Layout style={{

<Content

style={{

background:

'#fff',

padding: 24,

}}

>

Content

</Content>

</Layout>

</Layout>

</Layout>

</div>

)

}

}

### 样式代码：

.logo {

float: left; width: 200px; height: 64px;

}

.content {

height: calc(100vh - 64px);

}

更更改路路由配置，让**layout/index.js**成为最外层页面，嵌套路路由写法

//路路由配置

routes: [

{ path: "/login", component: "./login" },

{

path: "/",

component: "../layout",

routes: [

{ path: "/", component: "./index" },

//路路径是相对于pages

{ path: "/goods", component:

"./goods" },

{

path: "/about", component: "./about", Routes:

["./routes/PrivateRoute.js"] //路路由守卫配置编写 路路径相

对于根目录， 后缀名不不能省略略

},

{

path: "/users",

component: "./users/\_layout", routes: [

{ path: "/users/", component: "./users/index" },

{ path: "/users/:name", component: "./users/$name" }

]

},

{ component: "./notfound" }

]

},

]

# 第三节 改变顶部一级导航进行行跳转并改变视图内容

从**umi**引入路路由跳转组件： **import { Link } from 'umi'**

//动态获取当前路路由 解决默认选中样式引起的小bug

const selectedKeys = [this.props.location.pathname];

<Menu theme="dark" mode="horizontal"

selectedKeys={selectedKeys}//改变选中项的选中样式

style={{ lineHeight: '64px' }}

>

<Menu.Item key="/">

<Link to="/">首页</Link>

</Menu.Item>

<Menu.Item key="/about">

<Link to="/about">关于</Link>

</Menu.Item>

<Menu.Item key="/goods">

<Link to="/goods">商品</Link>

</Menu.Item>

</Menu>

# 第四节 贴近实战点击一级菜单切换二级菜单数据

### 创建一个**mock**文件模拟返回二级菜单数据

|  |
| --- |
| // antd版本4.+需要单独下载  import { UserOutlined, LaptopOutlined, NotificationOutlined } from '@ant-design/icons'  //左侧菜单栏数据，根据点击头部一级菜单动态改变  const menuList = [          //第一个一级菜单的二级菜单数据          [  {                  title: '统计报表',                  keyValue: 'sub1',                  iconType: <UserOutlined />,                  children: [{                          title: '浏览页面人次报表',                          keyValue: '1',                          routeurl: '/home/pageview'                      },                      {                          title: '浏览用户人次报表',                          keyValue: '2',                          routeurl: '/home/userview'                      },                  ]              },              {                  title: ' 设 置 ',                  keyValue: 'sub2',                  iconType: <LaptopOutlined />,                  children: [{                          title: '页面设置',                          keyValue: '3',                          routeurl: '/home/setpage'                      },                      {                          title: '语言设置',                          keyValue: '4',                          routeurl: '/home/setlanguage'                      },                  ]              }          ],          //第二个一级菜单的二级菜单数据          [           {              title: ' 关 于 在 线 教 育 ',              keyValue: 'sub1',              iconType: <NotificationOutlined />,              children: [{                      title: '在线教育类别',                      keyValue: '1',                      routeurl: '/about/educationtype'                  },                  {                      title: ' 在 线 教 育 如 何 选 择 ',                      keyValue: '2',                      routeurl: '/about/seleducation'                  },              ]           },          {              title: '关于小D课堂',              keyValue: 'sub2',              iconType: <UserOutlined />,              children: [{                      title: '前端讲师',                      keyValue: '3',                      routeurl: '/about/frontend'                  },                  {                      title: '后端讲师',                      keyValue: '4',                      routeurl: '/about/backend'                  },              ]          }      ],      //第三个一级菜单的二级菜单数据      [{              title: '直播课程',              keyValue: 'sub1',              iconType: <LaptopOutlined />,              children: [{                      title: 'java 零 基 础 进 阶 ',                      keyValue: '1',                      routeurl: '/goods/livejava'                  },                  {                      title: '冲啊！架构师',                      keyValue: '2',                      routeurl: '/goods/livearchitect'                  },              ]          },          {              title: ' 录 播 课 程 ',              keyValue: 'sub2',              iconType: <NotificationOutlined />,              children: [{                      title: '前端课程',                      keyValue: '3',                      routeurl: '/goods/frontcourse'                  },                  {                      title: '后端课程',                      keyValue: '4',                      routeurl: '/goods/backendcourse'                  },              ]          }      ],  ]  export default menuList |

# 第五节 进行页面归类优化项目可读性（一）

### 需要根据每个目录进行页面归类

第七节 引入**ant-design-pro**库并使用其现成的**404**页面

安装**ant-design-pro**： **npm install ant-design-pro -S**

[学习网站](https://v2-pro.ant.design/components/avatar-list-cn)

**404**页面代码

import {Exception} from 'ant-design-pro';

import styles from './notfound.css';

export default function() {

return (

<div>

<Exception type="404" backText="返回首页" redirect="/"/>

</div>

);

}

**type**设置错误类型

**backText**设置按钮的文案，默认不设置为**back to home**

可以在**redirect**设置点击按钮的跳转地址

# 第八节 详细讲解二级菜单栏伸缩状态利利用

**antd**如何实现

### 通过**state**里里面的一个变量量控制伸展或缩进状态

在菜单组件**Sider**加上这三个属性**collapsible collapsed=**

**{this.state.collapsed} onCollapse={this.onCollapse}**

### 然后添加触发的事件和控制状态

//添加控制伸缩的状态state = {

collapsed: false,

};

//添加伸缩事件

onCollapse = collapsed => { console.log(collapsed); this.setState({ collapsed });

};

 愿景：**"**让编程不不在难学，让技术与生活更更加有趣**"**

第十二章 全局管理用户状态及信息

第一节 从**ant-design-pro**的登录页里里面抽取出想要的代码

[查阅ant-design-pro现成的登录页](https://v2-pro.ant.design/components/login-cn)

提取需要的登录代码

import React, { Component } from "react"; import styles from "./login.css";

import { Login } from "ant-design-pro";

import logo from "../../public/logo.png" const { UserName, Password, Submit } = Login;

export default class extends Component {

onSubmit = (err, values) => { console.log("用户输入：", values,err);

};

render() { return (

<div className={styles.loginForm}>

{/\* logo \*/}

<img

className={styles.logo} src={logo}

/>

{/\* 登录表单 \*/}

<Login onSubmit={this.onSubmit}>

<UserName name="username" placeholder="xiaod"

rules={[{ required: true, message: "请输

入用户名" }]}

入密码" }]}

/>

<Password name="password" placeholder="123456"

rules={[{ required: true, message: "请输

/>

<Submit>登录</Submit>

</Login>

</div>

);

}

}

第二节 详细讲解编写登录**mock**接口

新建一个**login.js**的**mock**文件

export default {

"post /api/login"(req, res, next) {

const { username, password } = req.body; console.log(username, password);

if (username == "xiaod" && password == "123456") {

return res.json({ code: 0,

data: {

token: "xdclass",

role: "admin", username: "xiaod",

userimg:'<http://www.gx8899.com/uploads/allimg/2018042010/enei40gszt1.jpg>'

}

});

}

if (username == "tim" && password == "123456")

{

return res.json({ code: 0,

data: {

token: "xdclass", role: "user", username: "tim",

userimg:'<http://www.gx8899.com/uploads/allimg/2018031808/0ubakfmkpgy.jpg>'

}

});

}

return res.status(401).json({ code: -1,

msg: "账号或密码错误"

});

}

};

# 第三节 详细讲解**dva**编写登录功能

新建一个**model**文件 user**.js**

|  |
| --- |
| import axios from 'axios'  // umi3 history 代替  umi/router  import { history } from 'umi'  import { notification } from 'antd'  // 定义state的初始状态  // getItem检测本地存储是否有信息  const inituserInfo = JSON.parse( localStorage.getItem("userinfo")) || {      token:"",      role:"",      username:"",      userimg:""  }  // api请求（登录的接口）  接口调用  function login(data){      // 请求接口并将data(username,password)数据带过去      return axios.post('/api/login',data)  }  export default {      namespace: "user", //如不写namespace，命名空间会自动以此文件的文件名命名      state: inituserInfo,//初始状态      effects: {//异步操作          // 星星函数          \*login(action,{ put, call }){              try {                  const res = yield call(login,action.payload)                  if(res.data.code == 0){                      // 登录成功要把用户信息做缓存                      localStorage.setItem("userinfo", JSON.stringify(res.data.data) )                      yield put({ type:"init", payload: res.data.data })                      history.push('/') //登录成功跳转到首页                  }              } catch (error) {                  // alert('登录失败，账号或密码错误')                  notification.error({                      message:"登录失败",                      description:"登录账号或密码错误"                  })              }            }      },      reducers: {          init(state,action){              return action.payload          }      }  } |

### 如果大家对于**dva**模式的编程还有问题可以直接看登录功能的编写和商品页的编写

第四节 使用**redux**管理的用户信息改造路由守卫组件

|  |
| --- |
| // 此文件的作用 配置路由守卫----权限设置（没有登录去login，已登录去about）  // 页面跳转的两种方式  // 1.命令式 history.push  通常在事件处理中被调用  // 2. 声明式 通常作为 React 组件使用   <Link to="/list">Go to list page</Link>  // import { Link } from 'umi';  import { history, connect } from 'umi'  import React,{ Component } from 'react'  @connect(      state =>({token:state.user.token})  )  export default class extends Component{      render(){          console.log(this.props.token)         if(this.props.token){             return(              <div>                  {this.props.children}              </div>             )         }         history.push('/login')         return null      }  } |

第五节 使用下拉菜单并编写退出登录功能

引入头像组件和下拉菜单组件

引入**connect**

import { Icon, Dropdown ,Avatar} from 'antd';

//注入用户状态及退出登录方法@connect(

state => ({

userInfo: state.user

}),

{

logout:()=>(

{type:'user/logout'} // action的type需要以命名空间为前缀+reducer名称

)

}

)

//定义下拉菜单

const menu = (

<Menu>

<Menu.Item>

个人设置

</Menu.Item>

<Menu.Item onClick=

{()=>this.props.logout()}>

退出登录

</Menu.Item>

</Menu>

);

//使用下拉菜单

<div className={styles.user}>

<Dropdown overlay={menu}>

<div className="ant-

dropdown-link" href="#">

<Avatar size="large" src={this.props.userInfo.userimg} alt=

{this.props.userInfo.username}>

</Avatar>

<span style={{

fontSize: 20, marginLeft: 10 }}>

{this.props.userInfo.username}</span>

<Icon type="down" />

</div>

</Dropdown>

</div>

### 愿景：**"**让编程不不在难学，让技术与生活更更加有趣**"**

第十三章 回顾项目，展望未来

第一节 回顾整个项目及后续发展