==========

更多干货技术访问小D课堂官网 https://xdclass.net

小D课堂QQ客服Vicky：561509994

公众号搜索：小D课堂

==========

愿景："让编程不在难学，让技术与生活更加有趣"

第一章 欢迎来到React世界

第一节 详细了解我们的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自带的包管理工具**
  - **安装完毕后去命令行执行node -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node.js安装博客地址
- node.js官网下载地址

第三节 开始安装官方create-react-app脚手架并搭建项目

- react官网
- create-react-app脚手架
    - 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，让技术与生活更加有趣"

第二章 React基础知识讲解

第一节 深入理解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，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('root'))
```

- **JSX的实质就是React.createElement的调用**
- **可以去官网体验JSX的写法**
- **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

- **在组件里面我们通过{}在JSX里面渲染变量**

- **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函数里定义一个变量装载结果
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**
  - **第一种利用bing绑定，写法如下，这种比较少用**

```
//在constructor里面利用bing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(){
  //执行某些操作
  this.setState({})
}
//在DOM元素上直接使用
<button onClick={this.showTitleFun}>不显示
title</button>
```

  - **第二种箭头函数写法**　最常用的方法

```
showTitleFun = () => {
    //执行某些操作
    this.setState({});
  };
  //在DOM元素上直接使用
  <button onClick={this.showTitleFun}>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里动态绑定上面定义的变量
        <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 组件是否需要更新，返回布尔值，优化点
- componentWillUpdate 组件将要更新
- componentDidUpdate 组件已经更新
- componentWillUnmount 组件已经销毁 比如在用定时器的时候，组件销毁之后取消定时器

愿景："让编程不在难学，让技术与生活更加有趣"

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
- 聪明组件也叫容器组件
  - 负责数据的获取、处理
- 分清楚展示组件和容器组件的优势
  - 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**
- **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>
        )
    }
}
```

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

蚂蚁金服开源的组件库

- ant-design官网地址
- **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>
    )
  }
}
```

需要import整个antd的css，需要优化，后面会讲到按需加载

第二节 详细讲解配置ant-design按需加载

- 上一节课的使用antd例子实际上加载了全部的 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 版本的关系，你还需要安装 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npm install yarn -g 进行安装）
        - 更改package.json文件 eject不需要改动

```
/* package.json -代表没改前的代码，+代表已经更改的代码*/
"scripts": {
-   "start": "react-scripts start", 旧
+   "start": "react-app-rewired start", 新
-   "build": "react-scripts build",旧
+   "build": "react-app-rewired build", 新
-   "test": "react-scripts test", 旧
+   "test": "react-app-rewired test", 新
}
```

    - 然后在项目根目录创建一个 `config-overrides.js` 用于修改默认配置，先不用写内容
    - 执行安装 babel-plugin-import插件(安装命令：yarn add babel-plugin-import) 这是一个按需加载的插件
    - 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shouldComponentUpdate生命周期的Component 判断组件是否应该更新：判断传入的state是否更新，如果更新了的话，就更新组件
  - 它重写了一个方法来替换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

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PureComponent的功能
- 使用例子如下：

```
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>
  );
}
```

这个符号不是单引号，是键盘上数字1左边的符号

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包装，也可以重写部分生命周期
- 使用例子如下：

高阶组件命名都是用with开头

```
//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

链式调用的写法比较繁琐，下一节组件装饰器的写法可以解决这个问题

- 使用情况如下：
  - 编写一个高阶组件进行属性的添加
  - 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链式调用

- **使用例子如下**

```
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好理解。逻辑会看的比较绕
- ES7中就出现了装饰器的语法，专门拿来处理这种问题的
- 安装支持装饰器语法的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
        [
```

```
            '@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的设计目的是为了共享对于一个组件树而言是“全局”的数据
- 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
```

更简洁的方法：不写这一行，下面改成XdContext.Consumer和XdContext.Provider

```
//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> 这里写XdContext.Consumer
                {
                    store => {
                        return (
                            <div>
                                <p>姓名：{store.name}
</p>
                                <p>出处：{store.from}
</p>
                            </div>
                        )
                    }
                }
            </Consumer> 这里写XdContext.Consumer
        )
    }
}
function ToolBar(props){
    return (
        <div>
            <Info></Info>
        </div>
    )
}
export default class Context1 extends Component{
    render(){
        return (
```

```
                <Provider value={store}>   这里写XdContext.Provider
                    <ToolBar></ToolBar>
                </Provider>   这里写XdContext.Provider
            )
        }
    }
```

愿景："让编程不在难学，让技术与生活更加有趣"

第六章 React函数式编程之Hook

第一节 隆重介绍React Hooks

- ***Hook*** 是 React 16.8 的新增特性。它可以让你在不编写 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。但是如果你不想，你不必现在就去学习或使用 Hook。
  - 100% 向后兼容的。 Hook 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 不会影响你对 React 概念的理解。 恰恰相反，Hook 为已知的 React 概念提供了更直接的 API：props， state，context，refs 以及生命周期。接下来的学习我们会发现，Hook 还提供了一种更强大的方式来组合他们。
- React Hooks解决了什么问题?

1. **函数组件不能使用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，如果需要交互更改状态等复杂逻辑时就需要使用class组件了**

   React Hooks让我们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组件更简洁好用，因为React Hooks的出现，相信未来我们会更多的使用函数式组件

2. **副作用问题**
   - **我们一般称数据获取、订阅、定时执行任务、手动修改ReactDOM这些行为都可以称为副作用**
   - **由于React Hooks的出现，我们可以使用useEffect来处理组件副作用问题，所以我们的函数式组件也能进行副作用逻辑的处理了**
3. **有状态的逻辑重用组件**
4. **复杂的状态管理**
   1. **之前我们使用redux、dva、mobx第三方状态管理器来进行复杂的状态管理**
   2. **现在我们可以使用useReducer、useContext配合使用实现复杂状态管理，不用再依赖第三方状态管理器**
5. **开发效率和质量问题**
   1. **函数式组件比class组件简洁，开发的体验更好，效率更高同时应用的性能也更好**

- 封装好的React Hooks，可以来这里学习封装自定义hooks

第二节 详细介绍新特性useState

- **useState---组件状态管理钩子**
  - **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是更新state的方法，只是一个方法名，可以随意更改**
  - **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**
- **完整使用例子如下所示**

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

const App=() => {
    const [count,setState]=useState(0)
    return (
      <div>
          <div>你点击了{count}次</div>
          <button onClick={()=>setState(count+1)}>点
击</button>
      </div>
    )
}
```

第三节 详细介绍新特性useEffect

- **useEffect---副作用处理钩子**
  - **数据获取、订阅、定时执行任务、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)   array可写可不写
```

  - **callback: 回调函数，作用是处理副作用逻辑**
    - **callback可以返回一个函数，用作清理**

```
useEffect(() =>{
          //副作用逻辑
          xxxxxx
        return ()=>{
        //清理副作用需要清理的内容
        //类似于componentWillUnmount，组件渲染和组件卸载前执行的代码
        }
     },[])
```

  - **array(可选参数)：数组，用于控制useEffect的执行**
    - **分三种情况**
    - **空数组，则只会执行一次（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**
      - **不填array这个数组，useEffect每次渲染都会执行**

- 完整使用例子如下所示

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

const App=() => {
    const [count,setState]=useState(0)

    useEffect(() =>{
        //更新页面标题
        document.title=`您点击了${count}次了哦`
    },[count])
    return (
      <div>
          <div>你点击了{count}次</div>
          <button onClick={()=>setState(count+1)}>点
击</button>
      </div>
    )
}
```

第四节 详细介绍新特性useContext

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

  最好把所有数据来源放到同一个folder里面，这样比较好维护

  - 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是不需要再使用Provider和Consumer的**
- **当最近的context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，可以用于处理复杂的状态管理**
- **useReducer可以完全替代useState，只是我们简单的状态管理用useState比较易用，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状态处理并更新state**
  - **initState是初始化的state**
  - **initAction是useReducer初次执行时被处理的action**
- **返回值state，dispath介绍**
  - **state状态值**
  - **dispatch是更新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即可更新state，使用如下**

```
//dispatch是用来更新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action，然后返回最新的state，可以把reducer理解成是action和state的转换器，他会根据action的描述去更新state，使用例子：**

```
(state,action) => Newstate
```

- **具体使用例子**

```
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不属于上面的任意一个值，state不做更改，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的问题，如果把他们结合起来使用，就可以实现redux的功能了，意味着未来我们可以不再依赖第三方状态管理器了

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高开销的计算
  - 适用于复杂的计算场景，例如复杂的列表渲染，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逻辑
    - array 控制useMemo重新执行的数组，array改变时才会重新执行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'}

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

//使用方式
<div>姓名：{memoValue.name}---年龄：{memoValue.age}
</div>
```

- **不能在useMemo里面写副作用逻辑处理，副作用的逻辑处理放在useEffect内进行处理**

- **UseCallback和useMemo一样，也是用于性能优化的**
  - **基本使用方法** 唯一的不同是返回的值不同，useCallback返回的是callback整个函数

    ```
    const memoCallback= useCallback(callback,array)
    ```

    - **callback是一个函数用于处理逻辑**
    - **array 控制useCallback重新执行的数组，array改变时才会重新执行useCallback**
    - **跟useMemo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'}

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

//使用方式
<div>姓名：{memoCallback().name}---年龄：
{memoCallback().age}</div>
```

- **useRef 方便我们访问操作dom**
  - **使用方法如下**

用来方便我们获取表单里面的元素，可以是value也可以是function

```
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** 自己封装的Hooks命名必须用use开头
- **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的调用尽量只在顶层作用域进行调用**

    - 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，但是现阶段不能再class里使用

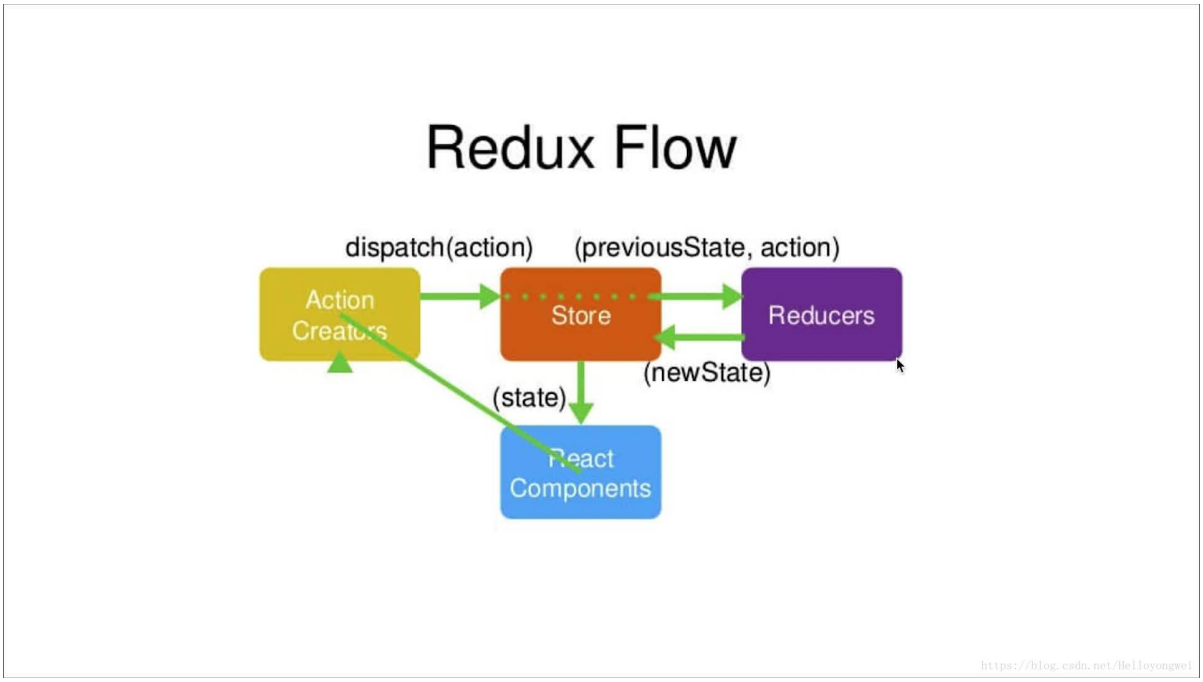
愿景："让编程不在难学，让技术与生活更加有趣"

第七章 强大的状态管理器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　　一般是用type
- store
    - store是整个应用的数据存储仓库，把我们全局管理的状态数据存储起来
    - 它就是我们的后勤保障，是我们打仗的政委，专门管理后勤数据
- reducers　真正干活的
    - reducers接收action并更新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使用redux进行数据管理，当我们不知道是否应该是用redux的时候，我们都是不需要的，因为只有我们很肯定redux能帮助我们管理好复杂项目数据流的时候他才能发挥它的威力，简单的项目我们只需要state+props+context就够了
- 接下来看一下Redux数据流的走向
-



第二节 学习redux编写一个累加器程序

- 安装redux npm install redux --save redux只支持同步的写法
- 编写使用redux的步骤(过程有点繁琐，别遗漏了任何的一步)
  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

2. **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的状态数据state**
    - **通过dispatch(action)可以触发更改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</h1>
                {/* 通过getState方法获取数据仓库里面的状
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进行改造累加器

- 由于redux的写法太繁琐，还每次都需要重新调用render，不太符合我们了解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改造累加器的代码 只需留意index.js和FirstRedux.js，store.js暂时不用作改变
- 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</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 React from 'react'
import ReactDOM from 'react-dom'
import FirstRedux from './TryRedux/FirstRedux'
import store from './TryRedux/store'
import { Provider } from 'react-redux'
ReactDOM.render(
    <Provider store={store}>
        <FirstRedux ></FirstRedux>
    </Provider>  ,
document.getElementById('root'))
```

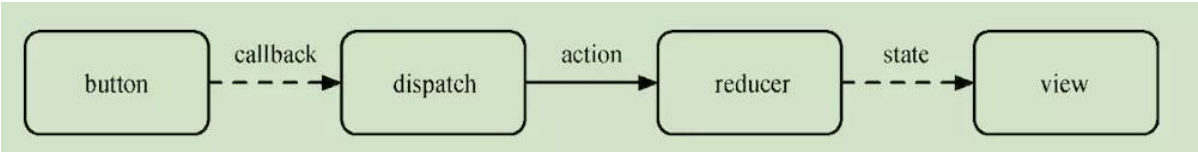
第四节 必备技能之高阶组件装饰器模式进行简化封装代码

- **connect高阶组件用装饰器会使我们的代码看起来更简洁易懂**
- **使用装饰器进行我们的代码优化**
- **FirstRedux.js**

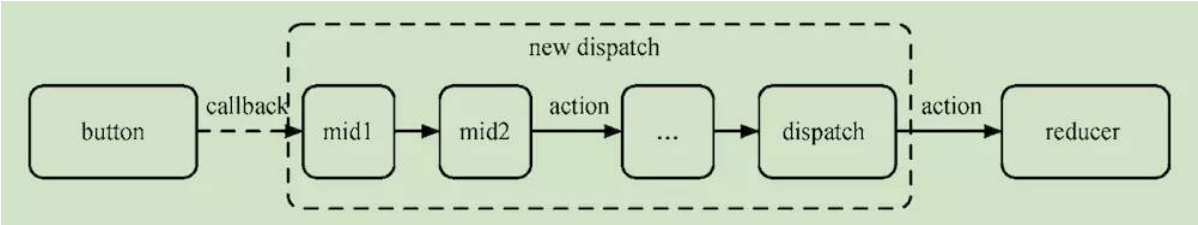
```
import React, { Component } from 'react'
import { connect } from 'react-redux'
@connect(
    state=>({count:state}),
    dispatch=>({
        add:()=>dispatch({type:'add'}),
        reduce:()=>dispatch({type:'reduce'})
    })

)
```

```
class FirstRedux extends Component {
    render() {
        return (
            <div>
                <h1>尝试使用redux编写一个累加器</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
- 没使用中间件时的redux数据流
- 

- 使用了中间件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
    - 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
```

```
const
store=createStore(fitstReducer,applyMiddleware(thunk
,logger))
export default store
```

第六节 详细讲解抽离reducer和action进行统一管理

- **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</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，抽离了reducer和action。之前的store.js已经可以删除了**

第八章 页面连接器之路由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由是针对浏览器的
  - npm install react-router-dom --save
  - 学习网站
- 特点：
  - 秉承react一切皆组件，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，可匹配多个路由

第二节 体验react-router的写法

- 使用react-router步骤 每一个使用到的组件都要引入
  - 引入顶层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路由</h1>
            <BrowserRouter>
                <App></App>
            </BrowserRouter>
        </div>
    )
}
```

- 引入Link组件编写路由导航
  - `<Link to="/">首页</Link>`
  - to 指定跳转去的路径
- 引入Route组件编写导航配置　component可以变成render，如果同时出现，component的优先级较高
  - `<Route exact path="/" component={Home}></Route>`
  - path 配置路径
  - component 配置路径所对应的组件
  - exact 完全匹配，只有路径完全一致时才匹配　首页一般需要用exact
- 编写导航配置对应的component组件

第三节 学习react-router的路由传参取参

- 用得比较多的两种传参取参方式

1. 声明式导航路由配置时配置路由参数

- 配置
  - 占位符，用来后面传递参数
  - `<Route path="/detail/:course" component={Detail}></Route>`
- 传递
  - `<Link to="/detail/react">react</Link>`
- 获取
  - 解构路由器对象里的match出来(macth获取参数信息)
  - `{match.params.course}`

2. 编程式导航式传递参数与获取
   - 解构路由器对象获取到导航对象 history(用作命令式导航)
   - 通过事件执行`history.push({ pathname: "/", state: { foo: "bar" } })`传递的参数装载在state里面
   - 从路由信息解构出location(当前的url信息)对象`location.state`进行获取

第四节 深入学习react嵌套路由及路由重定向

- 嵌套路由写法如下
  - 一级组件

```
function App(){
    return (
        <div>
            {/* 路由导航 */}
            <ul>
                <li>
                    <Link to="/mine">我的</Link>
                </li>
```

```
            </ul>
            {/* 路由配置 */}
           <Route path="/mine" component={Mine}>
</Route>
        </div>
    )
}
```

- **二级组件嵌套在一级组件里进行显示**

```
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：嵌套的子路由需要跟随父路由且不设置确切匹配
  - 例子
    - 父路由 /mine
    - 子路由 /mine/xxx

第五节 深入剖析路由守卫的实现与使用

- 路由守卫其实就是我们的路由拦截，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由守卫就可以派上用场了
- React里的路由守卫其实也是一个组件，最后返回的还是Route组件

愿景："让编程不在难学，让技术与生活更加有趣"

第九章 redux-saga与Generator生成器

第一节 介绍redux-saga和redux-thunk的不同

- redux-saga是一个用于管理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，可以用来代替redux-thunk中间件

- 工作成员分工：
  - reducer负责处理action的stage更新
  - sagas负责协调那些复杂或者异步的操作
- redux-saga可以处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 易读性强
- 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改造路由守卫登录认证

- 首先我们编写一个管理登录的reducer和action的文件 user.redux.js
- user.redux.js(安装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
                if (Math.random() > 0.5) {
                    //登陆成功，返回用户id和名字
                    resolve({ id: 1, name: 'tim' })
                } else {
                    //登陆失败
                    reject({ message: "用户名或密码错
误" })
                }
```

```
            }, 1000)
        })
    }
}
//编写真正工作的saga ES6的星星函数   Generator生成器
function* login(action) {
    try {
        //call调用异步函数
        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saga，把监听事件跑起来
mid.run(saga)

export default store
```

- **最后回到路由组件RouterSample.js里面进行修改(这里就不贴代码了)**
    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**

第三节 详解generator函数的原理和使用(一)

```
function* g() {
  yield "a";
  yield "b";
  yield "c";
  return "ending";
}
console.log(g()); //返回迭代器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())

// 使用递归函数执行生成器里面所有步骤
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函数的原理和使用(二)

- **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
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 }
```

- **例子二 结合promise**

```
// 使用Generator顺序执行两次异步操作
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
  function compute(num) {
    return new Promise(resolve => {
      setTimeout(() => {
        const ret = num * num;
        console.log(ret); // 输出处理结果
        resolve(ret); // 操作成功
      }, 1000);
    });
  }

//   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："让编程不在难学，让技术与生活更加有趣"

第十章 开始使用umi开发框架

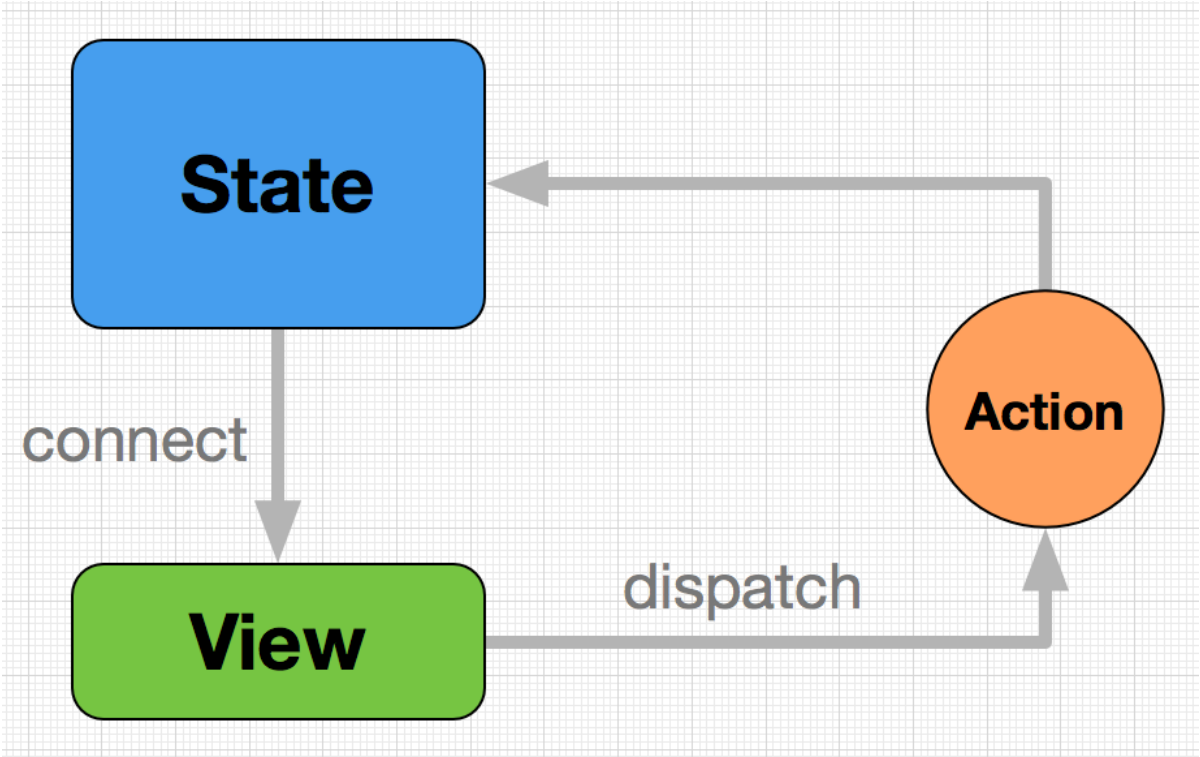
第一节 介绍企业级 react 应用框架umi

- umi学习　　蚂蚁金服的开源框架
- **umi特性**
  - 📦 **开箱即用**，内置 react、react-router 等
  - 🏈 **类 next.js 且功能完备的路由约定**，同时支持配置的路由方式
  - 🎉 **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、支付宝钱包、云凤蝶等
  - 🏎 **开发启动快**，支持一键开启 dll 等
  - 🐠 **一键兼容到 IE9**，基于 umi-plugin-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 详细了解dva是什么以及它与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**
  - **路由：** **React-Router**
  - **架构：** **Redux**
  - **异步操作：** Redux-saga
- **dva是目前react最流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(不要使用中文)
- 执行npm init生成package.json文件
  - 配置项scripts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umi g page index
- 再创建一个about页面umi g page about
- 约定式路由初体验
  - 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里面的嵌套路由与动态路由

- 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
- 动态路由文件命名为$开头，例子如下
  - users文件夹下的$name.js 即路由配置为/users/:name

第五节 创建配置文件编写配置式路由

- 配置式路由一旦创建，约定式路由将会失效
- 创建config文件夹，创建config.js文件

```
export default {
    routes: [
      { path: "/", component: "./index" }, // 路径相对于
pages
      {
        path: "/about",
        component: "./about",
      },
      {
        path: "/users",
        component: "./users/_layout",
        routes: [
          { path: "/users/", component: "./users/index"
},
          { path: "/users/:name", component:
"./users/$name" }
```

```
        ]
      },
    ]
  };
```

第六节 讲解配置式路由如何加入路由守卫

- 加入配置项Routes(注意点如下)
  - 加入的路由守卫组件路径相对于根目录
  - 后缀名不能省略
- 创建路由守卫组件

```
import Redirect from "umi/redirect";

export default props => {
  // 50%概率需要去登录页面
  if (Math.random()>0.5) {
    return <Redirect to="/login" />;
  }
  return (
    <div>
      {props.children}
    </div>
  );
};
```

第七节 讲解在umi里如何引入antd并使用

-S：运行时的依赖，上线时要打包一起上线

- 安装antd： npm install antd -S

-D：开发依赖，只需在本地使用，上线时不需打包

- 安装umi-plugin-react：npm install umi-plugin-react -D
- 安装好之后配置config文件夹下的config.js文件增加以下配置跟routes同级

```
plugins: [
        [
          "umi-plugin-react",
          {
            antd: true,
          }
        ]
      ],
```

第八节 在umi开发框架里面引入dva进行开发

- 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，进行纯粹的数据读写
- 使用dva，在配置文件config.js里面把开关打开即可

```
plugins: [
      [
        "umi-plugin-react",
        {
          antd: true,
          dva:true
        }
      ]
    ]
```

- **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
  },                          除了state必须有，其他的都可以没有
  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，所以我们在这里模拟数据
let data=[
    {title:'vue单页面电商项目'},
    {title:'react单页面后台管理项目'}
]
```

请求的路径

请求的方法

```
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，但是由于这里演示我们就先放在model
function getGoods(){
    return axios.get('/api/goods')
}

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
        *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，具体看项目源码

愿景："让编程不在难学，让技术与生活更加有趣"

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
- 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antd并配置好了，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

- **如果使用约定式路由，在src目录下新建一个layout文件夹，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由了，所以我们要去路由配置配置一个嵌套路由，使layout里的index.js成为整个项目的最外层页面**
- **从antd里面复制选中的布局代码 放进layout/index.js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} />
                        <Menu
                            theme="dark"
                            mode="horizontal"
                            defaultSelectedKeys=
{['2']}
                            style={{ lineHeight:
'64px' }}
                        >
                            <Menu.Item key="1">nav
1</Menu.Item>
                            <Menu.Item key="2">nav
2</Menu.Item>
                            <Menu.Item key="3">nav
3</Menu.Item>
                        </Menu>
                    </Header>
                    <Layout className=
{styles.content}>
                        <Sider width={200} style={{
background: '#fff' }}>
                            <Menu
                                mode="inline"
```

```
                                defaultSelectedKeys=
{['1']}
                                defaultOpenKeys=
{['sub1']}
                                style={{ height:
'100%', borderRight: 0 }}
                            >
                                <SubMenu
                                    key="sub1"
                                    title={
                                        <span>
                                            <Icon
type="user" />
                                            subnav 1
                                        </span>
                                    }
                                >
                                    <Menu.Item
key="1">option1</Menu.Item>
                                    <Menu.Item
key="2">option2</Menu.Item>
                                    <Menu.Item
key="3">option3</Menu.Item>
                                    <Menu.Item
key="4">option4</Menu.Item>
                                </SubMenu>
                                <SubMenu
                                    key="sub2"
                                    title={
                                        <span>
                                            <Icon
type="laptop" />
                                            subnav 2
                                        </span>
                                    }
```

```
                                    >
                                        <Menu.Item
key="5">option5</Menu.Item>
                                        <Menu.Item
key="6">option6</Menu.Item>
                                        <Menu.Item
key="7">option7</Menu.Item>
                                        <Menu.Item
key="8">option8</Menu.Item>
                                    </SubMenu>
                                    <SubMenu
                                        key="sub3"
                                        title={
                                            <span>
                                                <Icon
type="notification" />
                                                subnav 3
                                            </span>
                                        }
                                    >
                                        <Menu.Item
key="9">option9</Menu.Item>
                                        <Menu.Item
key="10">option10</Menu.Item>
                                        <Menu.Item
key="11">option11</Menu.Item>
                                        <Menu.Item
key="12">option12</Menu.Item>
                                    </SubMenu>
                                </Menu>
                            </Sider>
                            <Layout style={{
paddingLeft: '24px' }}>
                                <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);
  }
```

更改路由配置，让layout/index.js成为最外层页面，嵌套路由写法

```
//路由配置
    routes: [
        { path: "/login", component: "./login" },
        {
            path: "/",
            component: "../layout",
```

```
            routes: [
                { path: "/", component: "./index" },
//路径是相对于pages
                { path: "/goods", component:
"./goods" },
                {
                    path: "/about",
                    component: "./about",
                    Routes:
["./routes/PrivateRoute.js"]  //路由守卫配置编写 路径相
对于根目录，  后缀名不能省略
                },
                {
                    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跳转并改变视图内容

- 从umi引入路由跳转组件： import Link from 'umi/link'

```
//动态获取当前路由 解决默认选中选中样式引起的小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

```
//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: 'user',
            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: 'laptop',
            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: 'notification',
            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: 'user',
            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: 'laptop',
            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: 'notification',
            children: [
                {
                    title: '前端课程',
                    keyValue: '3',
                    routeurl:'/goods/frontcourse'
                },
                {
                    title: '后端课程',
                    keyValue: '4',
                    routeurl:'/goods/backendcourse'
                },
```

```
            ]
        }
    ],

]
export default menuList
```

- 页面初始化时在componentDidMount生命周期给二级菜单一个初始化数据
- 点击一级菜单时进行二级菜单的数据切换

第五节 进行页面归类优化项目可读性(一)

- 如果全部页面都直接放在pages根目录下，那当项目迭代起来后会变得很乱，导致后期维护成本很高
- 我们需要根据每个目录进行页面归类

第六节 进行页面归类优化项目可读性(二)

- 如果全部页面都直接放在pages根目录下，那当项目迭代起来后会变得很乱，导致后期维护成本很高
- 我们需要根据每个目录进行页面归类
- 修复一个小bug，点击一级菜单下的二级菜单时，一级菜单不再选

中

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
- 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用antd如何实现

- **通过state里面的一个变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："让编程不在难学，让技术与生活更加有趣"

第十二章 全局管理用户状态及信息

第一节 从ant-design-pro的登录页里面抽取出想要的代码

- 查阅ant-design-pro现成的登录页
- **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" }]}
          />
          <Password
            name="password"
            placeholder="123456"
            rules={[{ required: true, message: "请输
入密码" }]}
          />
          <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:'https://ss0.bdstatic.com/94oJfD_bAAcT8t7mm9
GUKT-xh_/timg?
image&quality=100&size=b4000_4000&sec=1566400577&di=
52614f11afb03df632301e7d5f6486f6&src=http://m.360buy
img.com/pop/jfs/t25291/51/240832757/36633/7109614a/5
b696d12N1fa998f1.jpg'
          }
        });
      }
      if (username == "tim" && password == "123456")
{
        return res.json({
          code: 0,
          data: {
            token: "xdclass",
            role: "user",
            username: "tim",
            userimg:'https://timgsa.baidu.com/timg?
image&quality=80&size=b9999_10000&sec=1566410664355&
di=2472237155ce7797ffb74d79a41d17ff&imgtype=0&src=ht
tp%3A%2F%2Fi2.hexun.com%2F2018-11-
06%2F195121843.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 login.js
- 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

```
import Redirect from "umi/redirect";
import { connect } from "dva";
import React, { Component } from "react";
@connect(
  state=>({token:state.user.token})
)
export default class extends Component {
  render() {
    if (!this.props.token) {
      return <Redirect to="/login" />;
    }
    return (
      <div>
        {this.props.children}
      </div>
    );
  }
```

```
};
```

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："让编程不在难学，让技术与生活更加有趣"

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