**React全家桶**

# 注意点：

1. 在jsx中的class必须写成className
2. 在jsx中使用变量需要用{}包裹
3. 只能有一个根节点
4. 注释要写在{}中，{/\* \*/}

# 第1章：React入门

## React的基本认识

### 官网

1. 英文官网:<https://reactjs.org/>
2. 中文官网: <https://doc.react-china.org/>

### 介绍描述

1. 用于动态构建用户界面的 JavaScript 库(只关注于View)
2. 由Facebook开源

### React的特点

1. Declarative(声明式编码)
2. Component-Based(组件化编码)
3. Learn Once, Write Anywhere(React Native/支持客户端与服务器渲染)
4. 高效
5. 单向数据流

### React高效的原因

1. 虚拟(virtual)DOM, 不总是直接操作DOM (文档/元素/属性/文本对象)
2. DOM Diff算法, 最小化页面重绘

## React的基本使用

注意: 此时只是测试语法使用, 并不是真实项目开发使用

### 效果



### 相关js库

1. react.js: React的核心库
2. react-dom.js: 提供操作DOM的react扩展库
3. babel.min.js: 解析JSX语法代码转为纯JS语法代码的库

### 在页面中导入js

|  |
| --- |
| *// react核心库*  <**script type="text/javascript" src="../js/react.development.js"**></**script**>  *// 用于操作DOM* <**script type="text/javascript" src="../js/react-dom.development.js"**></**script**>  *// 用于解析jsx为js* <**script type="text/javascript" src="../js/babel.min.js"**></**script**> |

### 编码

|  |
| --- |
| <**script type="text/babel"**> *//必须声明babel，因为我们写的是jsx，需要借助babel翻译  // 创建虚拟DOM元素，注：下面的<h1>不是html中的<h1>，但是它与html中的<h1>一一对应* **const *vDom*** = <**h1**>Hello React</**h1**> *// 千万不要加引号*  *// 借助react把虚拟DOM渲染到页面，变为真实DOM* **ReactDOM**.render(***vDom***, **document**.getElementById(**'test'**)) </**script**> |

### Html使用react的代码

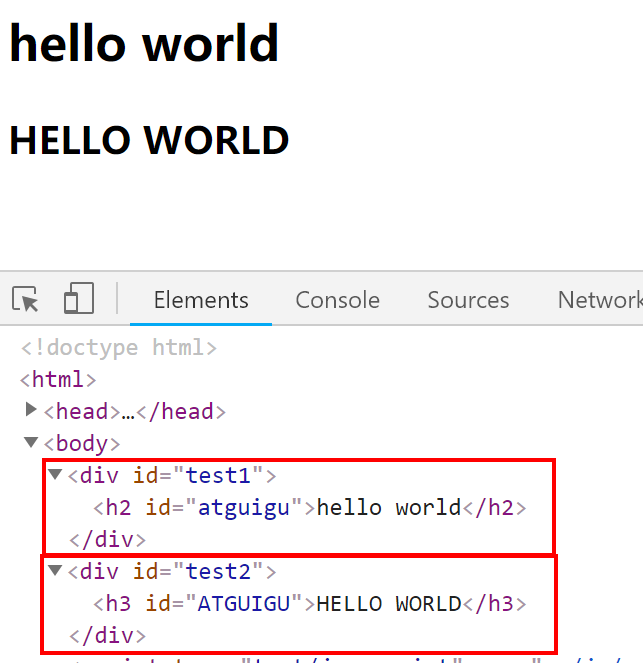
|  |
| --- |
| *<div id=***"test"***></div>*  *// react核心库*  <**script type="text/javascript" src="../js/react.development.js"**></**script**>  *// 用于操作DOM* <**script type="text/javascript" src="../js/react-dom.development.js"**></**script**>  *// 用于解析jsx为js* <**script type="text/javascript" src="../js/babel.min.js"**></**script**>  <**script type="text/babel"**> *//必须声明babel，因为我们写的是jsx，需要借助babel翻译  // 创建虚拟DOM元素，注：下面的<h1>不是html中的<h1>，但是它与html中的<h1>一一对应* **const *vDom*** = <**h1**>Hello React</**h1**> *// 千万不要加引号*  *// 借助react把虚拟DOM渲染到页面，变为真实DOM* **ReactDOM**.render(***vDom***, **document**.getElementById(**'test'**)) </**script**> |

### 使用React开发者工具调试



## React JSX

### 效果



### 虚拟DOM

1. React提供了一些API来创建一种 `特别` 的一般js对象
   1. var element = React.createElement('h1', {id:'myTitle'},'hello')
   2. 上面创建的就是一个简单的虚拟DOM对象
2. 虚拟DOM对象最终都会被React转换为真实的DOM
3. 我们编码时基本只需要操作react的虚拟DOM相关数据, react会转换为真实DOM变化而更新界面

### JSX

1. 全称: JavaScript XML
2. react定义的一种类似于XML的JS扩展语法: XML+JS
3. 本质是React.createElement(component, props, ...children) 方法的语法糖
4. 作用: 用来创建react虚拟DOM(元素)对象
   1. var ele = <h1>Hello JSX!</h1>
   2. 注意1: 它不是字符串, 也不是HTML/XML标签
   3. 注意2: 它最终产生的就是一个JS对象
5. 标签名任意: HTML标签或其它标签
6. 标签属性任意: HTML标签属性或其它
7. **基本语法规则**
   1. 遇到 <开头的代码, 以标签的语法解析: html同名标签转换为html同名元素, 其它标签需要特别解析
   2. 遇到以 { 开头的代码，以JS语法解析: **标签中的js代码必须用{ }包含**
8. babel.js的作用
   1. 浏览器不能直接解析JSX代码, 需要babel转译为纯JS的代码才能运行
   2. 只要用了JSX，都要加上type="text/babel", 声明需要babel来处理

### 渲染虚拟DOM(元素)

1. 语法: ReactDOM.render(virtualDOM, containerDOM)
2. 作用: 将虚拟DOM元素渲染到页面中的真实容器DOM中显示
3. 参数说明
   1. 参数一: 纯js或jsx创建的虚拟dom对象
   2. 参数二: 用来包含虚拟DOM元素的真实dom元素对象(一般是一个div)

### 建虚拟DOM的2种方式

1. 纯JS(一般不用)【不利于嵌套的html元素】

<script type=”text/javascript”>

*<div id=***"test"***></div>*

**const myh1 = React.createElement('h1', {title: 'hello'}, 'Hello')**

**ReactDOM.render(myh1,document.getElementById('test'))**

</script>

1. JSX:【直接书写内容,注：在jsx中不能使用class添加类，需要只用className添加】

*<script type=”text/babel”>*

*<div id=***"test"***></div>*

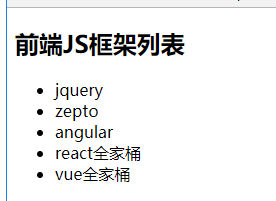
**const myh1 =***<h1 id={myTitle} className=”red”>{title}</h1>*

**ReactDOM.render(myh1,document.getElementById('test'))**

*</script>*

### JSX练习

需求: 动态展示列表数据【数据数组展示为标签数组】



**import React from 'react'**

**import ReactDOM from 'react-dom'**

**const arr1 = ['2','4','6','8','10']**

**ReactDOM.render(<div>**

**{*/\* 将普通字符串数组，转为jsx数组并渲染到页面上 \*/*}**

**{*/\* {arr1.map(item => <h2>{item}</h2>)} \*/*}**

**{arr1.map(function(item,i){**

**return <h2 *key*={i}>{item}</h2>**

**})}**

**</div>,document.getElementById('app'))**

## 模块与组件和模块化与组件化的理解

### 模块

1. 理解: 向外提供特定功能的js程序, 一般就是一个js文件
2. 为什么: js代码更多更复杂
3. 作用: 复用js, 简化js的编写, 提高js运行效率

### 组件

1. 理解: 用来实现特定(局部)界面功能效果的代码集合(html/css/js/image)
2. 为什么: 一个界面的功能更复杂
3. 作用: 复用编码, 简化项目编码, 提高运行效率

### 模块化

当应用的js都以模块来编写的, 这个应用就是一个模块化的应用

### 组件化

当应用是以多组件的方式实现, 这个应用就是一个组件化的应用



# 第2章：React面向组件编程

## 2.1. 基本理解和使用

### 2.1.1. 效果



### 2.1.2. 自定义组件(Component) :

1. 定义组件(2种方式)

|  |
| --- |
| **/\*方式1: 工厂函数组件(无状态/简单组件)\*/**  **function** *MyComponent* () {  **return** <**h2**>工厂函数组件(简单组件)</**h2**> }  **/\*方式2: ES6类组件(有状态/复杂组件)\*/**  **class** MyComponent2 **extends React**.*Component* {  render () { // 重写父类方法  **console**.log(**this**) *// MyComponent2的实例对象* **return** <**h2**>ES6类组件(复杂组件)</**h2**>  } } |

1. 渲染组件标签【如果渲染标签的时候，首字母小写，那么会尝试找html标签；首字母大写，那么尝试会寻找组件标签】

|  |
| --- |
| **ReactDOM**.render(<**MyComponent** />, **document**.getElementById(**'example1'**)) |

### 2.1.3. 注意

1. **组件名必须首字母大写**
2. 虚拟DOM元素只能有一个根元素
3. 虚拟DOM元素必须有结束标签

### 2.1.4. render()渲染类组件标签的基本流程

1. React内部会创建组件实例对象
2. 调用render()得到虚拟DOM, 并解析为真实DOM
3. 插入到指定的页面元素内部

## 2.2. 组件三大属性1: state

### 2.2.1. 效果



### 2.2.2. 理解

1. state是组件对象最重要的属性, 值是对象(可以包含多个数据)
2. 组件被称为"状态机", 通过更新组件的state来更新对应的页面显示(重新渲染组件)

### 2.2.3. 编码操作

1. 初始化状态:

*constructor (props) {*

*super(props)*

*this.state = {*

*stateProp1 : value1,*

*stateProp2 : value2*

*}*

*}*

1. 读取某个状态值

*this.state.statePropertyName*

1. 更新状态---->组件界面更新

*this.setState({*

*stateProp1 : value1,*

*stateProp2 : value2*

*})*

### 2.2.4. 强烈注意

1. 组件内置的方法中的this为组件对象
2. 在组件类中自定义的方法中this为undefined
   1. 强制绑定this: 通过函数对象的bind()
   2. 箭头函数(ES6模块化编码时才能使用)
3. 状态数据，不能直接修改或更新

## 2.3. 组件三大属性2: props

### 2.3.1. 效果

*需求: 自定义用来显示一个人员信息的组件  
 1). 姓名必须指定  
 2). 如果性别没有指定, 默认为男*

*3). 如果年龄没有指定, 默认为18*



### 2.3.2. 理解

1. 每个组件对象都会有props(properties的简写)属性
2. 组件标签的所有属性都保存在props中

### 2.3.3. 作用

1. 通过标签属性从组件外向组件内传递变化的数据
2. 注意: 组件内部不要修改props数据

### 2.3.4. 编码操作

1. 内部读取某个属性值

**this**.**props**.**name**

1. 对props中的属性值进行类型限制和必要性限制

**第一种方式（React v15.5 开始已弃用）：**

*Person*.**propTypes** = {  
 **name**: **React**.**PropTypes**.**string**.isRequired,  
 **age**: **React**.**PropTypes**.**number**.isRequired  
}

**第二种方式（新）：**

使用prop-types库进限制（需要引入prop-types库）

*Person*.**propTypes** = {  
 **name**: **PropTypes**.**string**.isRequired,  
 **age**: **PropTypes**.**number**.isRequired  
}

1. 扩展属性: 将对象的所有属性通过props传递

<**Person** {...***person***}/>

1. 默认属性值

Person.**defaultProps** = {  
 **age**: 18,  
 **sex**:**'男'**}

1. 组件类的构造函数

**constructor**(props){  
 **super**(props)  
 **console**.log(props)*//打印所有属性*}

### 2.3.5. 面试题

问题: 请区别一下组件的props和state属性

1. state: 组件自身内部可变化的数据
2. props: 从组件外部向组件内部传递数据, 组件内部只读不修改

## 2.4. 组件三大属性3: refs与事件处理

### 2.4.1. 效果

*需求: 自定义组件, 功能说明如下:  
 2. 点击按钮, 提示第一个输入框中的值  
 3. 当第2个输入框失去焦点时, 提示这个输入框中的值*



### 2.4.2. 组件的3大属性之二: refs属性

1. 老版本语法

组件内的标签都可以定义ref属性来标识自己

<**input type="text" ref=**{input => **this**.**userInput** = input}/>

在组件中可以通过this.refs.content来得到对应的真实DOM元素

1. 新版本语法

使用 React.createRef() 创建用于保存dom元素的ref对象

constructor(props) {

super(props)

this.myRef = React.createRef()

}

通过 ref 标签属性将DOM元素对象保存到ref对象的current属性上

<input type="text" ref={this.myRef}/>

获取指定了ref的dom元素对象

const input = this.myRef.current

### 2.4.3. 事件处理

1. 通过onXxx属性指定组件的事件处理函数(注意大小写)
   1. React使用的是自定义(合成)事件, 而不是使用的原生DOM事件
   2. React中的事件是通过事件委托方式处理的(委托给组件最外层的元素)
2. 通过event.target得到发生事件的DOM元素对象

*<input onFocus={this.handleClick}/>*

*handleFocus(event) {*

*event.target //返回input对象*

*}*

## 2.5. 组件的组合

### 2.5.1. 效果

功能: 组件化实现此功能

1. 显示所有todo列表  
 2. 输入文本, 点击按钮显示到列表的首位, 并清除输入的文本



### 2.5.2. 功能界面的组件化编码流程(通用/无比重要)

1. 拆分组件: 拆分界面,抽取组件
2. 实现静态组件: 使用组件实现静态页面效果
3. 实现动态组件
   1. 动态显示初始化数据
      1. 数据类型
      2. 数据名称
      3. 保存在哪个组件?
   2. 交互(从绑定事件监听开始)

## 2.6. 收集表单数据

### 2.6.1. 效果

需求: 自定义包含表单的组件  
 1. 输入用户名密码后, 点击登陆提示输入信息  
 3. 不提交表单



### 2.6.2. 理解

1. 问题: 在react应用中, 如何收集表单输入数据
2. 包含表单的组件分类
   1. 受控组件
   2. 非受控组件

## 2.7. 组件生命周期

### 2.7.1. 效果

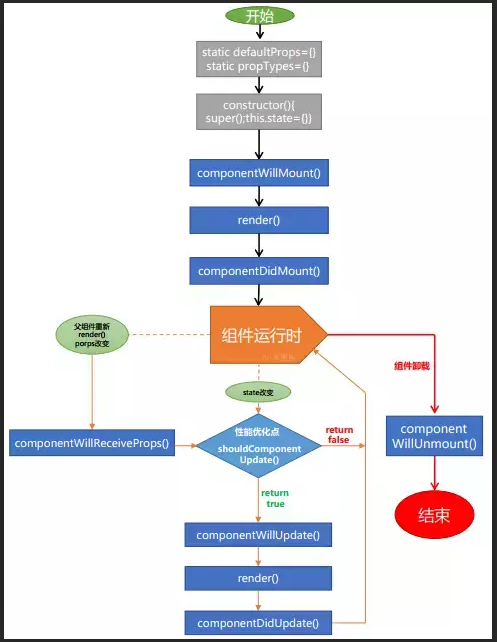
需求: 自定义组件  
 1. 让指定的文本做显示/隐藏的渐变动画  
 2. 切换持续时间为2S  
 3. 点击按钮从界面中移除组件界面



### 2.7.2. 理解

1. 组件对象从创建到死亡它会经历特定的生命周期阶段
2. React组件对象包含一系列的勾子函数(生命周期回调函数), 在生命周期特定时刻回调
3. 我们在定义组件时, 可以重写特定的生命周期回调函数, 做特定的工作

### 2.7.3. 生命周期流程图



### 2.7.4. 生命周期详述

1. 组件的三个生命周期状态:

\* Mount：插入真实 DOM

\* Update：被重新渲染

\* Unmount：被移出真实 DOM

1. React 为每个状态都提供了勾子(hook)函数

\* componentWillMount()

\* componentDidMount()

\* componentWillUpdate()

\* componentDidUpdate()

\* componentWillUnmount()

1. 生命周期流程:
   1. 第一次初始化渲染显示: ReactDOM.render()

\* constructor(): 创建对象初始化state

\* componentWillMount() : 将要插入回调

\* render() : 用于插入虚拟DOM回调

\* componentDidMount() : 已经插入回调

* 1. 每次更新state: this.setState()

\* componentWillUpdate() : 将要更新回调

\* render() : 更新(重新渲染)

\* componentDidUpdate() : 已经更新回调

* 1. 移除组件: ReactDOM.unmountComponentAtNode(containerDom)

\* componentWillUnmount() : 组件将要被移除回调

### 2.7.5. 重要的勾子

1. render(): 初始化渲染或更新渲染调用
2. componentDidMount(): 开启监听, 发送ajax请求
3. componentWillUnmount(): 做一些收尾工作, 如: 清理定时器
4. componentWillReceiveProps(): 后面需要时讲

## 2.8. 虚拟DOM与DOM Diff算法

### 2.8.1. 效果



|  |
| --- |
| **class** HelloWorld **extends React**.*Component* {  **constructor**(props) {  **super**(props)  **this**.**state** = {  **date**: **new** Date()  }  }   componentDidMount () {  setInterval(() => {  **this**.setState({  **date**: **new** Date()  })  }, 1000)  }   render () {  **console**.log(**'render()'**)  **return** (  <**p**>  Hello, <**input type="text" placeholder="Your name here"**/>!**&nbsp;** It is {**this**.**state**.**date**.toTimeString()}  </**p**>  )  } } **ReactDOM**.render(  <**HelloWorld**/>,  **document**.getElementById(**'example'**) ) |

### 2.8.2. 基本原理图



# 第3章：react应用(基于react脚手架)

## 3.1. 使用create-react-app创建react应用

### 3.1.1. react脚手架

1. xxx脚手架: 用来帮助程序员快速创建一个基于xxx库的模板项目
   1. 包含了所有需要的配置
   2. 指定好了所有的依赖
   3. 可以直接安装/编译/运行一个简单效果
2. react提供了一个用于创建react项目的脚手架库: create-react-app
3. 项目的整体技术架构为: react + webpack + es6 + eslint
4. 使用脚手架开发的项目的特点: 模块化, 组件化, 工程化

### 3.1.2. 创建项目并启动

npm install -g create-react-app

create-react-app hello-react

cd hello-react

npm start

### 3.1.3. react脚手架项目结构

ReactNews

|--node\_modules---第三方依赖模块文件夹

|--public

*|--* *index.html-----------------主页面*

|--src------------源码文件夹

*|--components-----------------react组件*

*|--index.js-------------------应用入口js*

|--.gitignore------git版本管制忽略的配置

|--package.json----应用包配置文件

|--README.md-------应用描述说明的readme文件

## 3.2. demo: 评论管理

### 3.2.1. 效果



### 3.2.2. 拆分组件

应用组件: App

\* state: comments/array

添加评论组件: CommentAdd

\* state: username/string, content/string

\* props: add/func

评论列表组件: CommentList

\* props: comment/object, delete/func, index/number

评论项组件: CommentItem

\* props: comments/array, delete/func

### 3.2.3. 实现静态组件

### 3.2.4. 实现动态组件

动态展示初始化数据

\* 初始化状态数据

\* 传递属性数据

响应用户操作, 更新组件界面

\* 绑定事件监听, 并处理

\* 更新state

### 3.2.4. 面试题: React组件化编码的3步与2个重要问题

|  |
| --- |
| *编码的3步  1). 拆分组件  2). 实现静态组件  3). 实现动态组件  a. 动态显示初始化数据  1. 类型  2. 名称  3. 保存在哪个组件  b. 交互 2个重要问题  1). 状态数据保存在哪个组件? 看是某个组件需要, 还是某些组件需要(给父组件)  2). 更新状态数据的行为在哪个组件? 状态在哪个组件, 行为就定义在哪个组件* |

# 第4章：react ajax

## 4.1. 理解

### 4.1.1. 前置说明

1. React本身只关注于界面, 并不包含发送ajax请求的代码
2. 前端应用需要通过ajax请求与后台进行交互(json数据)
3. react应用中需要集成第三方ajax库(或自己封装)

### 4.1.2. 常用的ajax请求库

1. jQuery: 比较重, 如果需要另外引入不建议使用
2. axios: 轻量级, 建议使用
   1. 封装XmlHttpRequest对象的ajax
   2. promise风格
   3. 可以用在浏览器端和node服务器端
3. fetch: 原生函数, 但老版本浏览器不支持
   1. 不再使用XmlHttpRequest对象提交ajax请求
   2. 为了兼容低版本的浏览器, 可以引入兼容库fetch.js

### 4.1.3. 效果

需求:  
 1. 界面效果如下  
 2. 根据指定的关键字在github上搜索匹配的最受关注的库  
 3. 显示库名, 点击链接查看库

4. 测试接口: https://api.github.com/search/repositories?q=r&sort=stars

**

## 4.2. axios

### 4.2.1. 文档

<https://github.com/axios/axios>

### 4.2.2. 相关API

1. GET请求

|  |
| --- |
| axios.get('/user?ID=12345')  .then(function (response) {  console.log(response.data);  })  .catch(function (error) {  console.log(error);  });  axios.get('/user', {  params: {  ID: 12345  }  })  .then(function (response) {  console.log(response);  })  .catch(function (error) {  console.log(error);  }); |

1. POST请求

|  |
| --- |
| axios.post('/user', {  firstName: 'Fred',  lastName: 'Flintstone'  })  .then(function (response) {  console.log(response);  })  .catch(function (error) {  console.log(error);  }); |

## 4.3. Fetch

### 4.3.1. 文档

1. <https://github.github.io/fetch/>
2. <https://segmentfault.com/a/1190000003810652>

### 4.3.2. 相关API

1. GET请求

|  |
| --- |
| fetch(url).then(function(response) {  return response.json()  }).then(function(data) {  console.log(data)  }).catch(function(e) {  console.log(e)  }); |

1. POST请求

|  |
| --- |
| fetch(url, {  method: "POST",  body: JSON.stringify(data),  }).then(function(data) {  console.log(data)  }).catch(function(e) {  console.log(e)  }) |

## 4.4. 测试: 获取github上最受欢迎的库



## 4.5. demo: github用户搜索

### 4.5.1. 效果



### 4.5.2. 拆分组件

App

\* state: searchName/string

Search

\* props: setSearchName/func

List

\* props: searchName/string

\* state: firstView/bool, loading/bool, users/array, errMsg/string

### 4.5.3. 编写静态组件

### 4.5.4. 编写动态组件

componentWillReceiveProps(nextProps): 监视接收到新的props, 发送ajax

使用axios库发送ajax请求

# 第5章：几个重要技术总结

## 5.1. 组件间通信

### 5.1.1. 方式一: 通过props传递

1. 共同的数据放在父组件上, 特有的数据放在自己组件内部(state)
2. 通过props可以传递一般属性和函数属性, 只能一层一层传递
3. 一般属性-->父组件传递数据给子组件-->子组件读取数据
4. 函数属性-->子组件传递数据给父组件-->子组件调用函数

### 5.1.2. 方式二: 使用消息订阅(subscribe)-发布(publish)机制

1. 工具库: PubSubJS
2. 下载: npm install pubsub-js --save
3. 使用:

import PubSub from 'pubsub-js' //引入

PubSub.subscribe('delete', function(data){ }); //订阅

PubSub.publish('delete', data) //发布消息

### 5.1.3. 方式三: redux

后面专门讲解

### 5.1.4. 面试题: 比较react中组件间3种通信方式

|  |
| --- |
| *1). 方式一: 通过props传递  通过props可以传递一般数据和函数数据,   一般数据-->父组件向子组件  函数数据-->子组件向父组件通信  缺点: 只能一层一层传递/兄弟组件必须借助父组件 2). 方式二: 使用消息订阅(subscribe)-发布(publish)机制  实现库: pubsub-js  组件A: 发布消息(相当于触发事件)  组件B: 订阅消息, 接收消息, 处理消息(相当于绑定事件监听)  优点: 对组件关系没有限制 3). 方式三: redux  通过redux可以实现任意组件间的通信  集中式管理多个组件共享的状态, 而pubsub-js并不是集中式的管理* |

## 5.2. 事件监听理解

### 5.2.1. 原生DOM事件

1. 绑定事件监听
   1. 目标元素
   2. 事件名(类型): 只有有限的几个, 不能随便写
   3. 回调函数: 接收数据并处理
2. 触发事件
   1. 用户操作界面
   2. 事件名(类型)
   3. 数据(event)

### 5.2.2. 自定义事件(消息机制)

1. 绑定事件监听(订阅消息)
   1. 目标元素
   2. 事件名(类型): 任意
   3. 回调函数: 通过形参接收数据, 在函数体处理事件
2. 触发事件(编码, 发布消息)
   1. (事件名类型): 与绑定的事件监听的事件名一致
   2. 数据: 会自动传递给回调函数

## 5.3. ES6常用新语法

1. 定义常量/变量: const/let
2. 解构赋值: let {a, b} = this.props import {aa} from 'xxx' function fn({name}) {} fn({name: ‘tom’})
3. 对象的简洁表达: {a, b}
4. 箭头函数:
   1. 常用场景

\* 组件的自定义方法: xxx = () => {}

\* 参数匿名函数

* 1. 优点:

\* 简洁

\* 没有自己的this,使用引用this查找的是外部this

1. 扩展(三点)运算符: 拆解对象(const MyProps = {}, <Xxx {...MyProps}>)
2. 类: class/extends/constructor/super
3. ES6模块化: export default | import
4. promise
5. async/await

# 第6章：react-router4

## 6.1. 相关理解

### 6.1.1. react-router-dom的理解

1. react的一个插件库
2. 专门用来实现一个SPA应用
3. 基于react的项目基本都会用到此库

### 6.1.2. SPA的理解

1. 单页Web应用（single page web application，SPA）
2. 整个应用只有一个完整的页面
3. 点击页面中的链接不会刷新页面, 本身也不会向服务器发请求
4. 当点击路由链接时, 只会做页面的局部更新
5. 数据都需要通过ajax请求获取, 并在前端异步展现

### 6.1.3. 路由的理解

1. 什么是路由?
   1. 一个路由就是一个映射关系(key:value)
   2. key为路由路径, value可能是function/component
2. 路由分类
   1. 后台路由: node服务器端路由, value是function, 用来处理客户端提交的请求并返回一个响应数据
   2. 前台路由: 浏览器端路由, value是component, 当请求的是路由path时, 浏览器端前没有发送http请求, 但界面会更新显示对应的组件
3. 后台路由
   1. 注册路由: router.get(path, function(req, res))
   2. 当node接收到一个请求时, 根据请求路径找到匹配的路由, 调用路由中的函数来处理请求, 返回响应数据
4. 前端路由
   1. 注册路由: <Route path="/about" component={About}>
   2. 当浏览器的hash变为/about时, 当前路由组件就会变为About组件

### 6.1.4. 前端路由的实现

1. history库
   1. 网址: <https://github.com/ReactTraining/history>
   2. 管理浏览器会话历史(history)的工具库
   3. 包装的是原生BOM中window.history和window.location.hash
2. history API
   1. History.createBrowserHistory(): 得到封装window.history的管理对象
   2. History.createHashHistory(): 得到封装window.location.hash的管理对象
   3. history.push(): 添加一个新的历史记录
   4. history.replace(): 用一个新的历史记录替换当前的记录
   5. history.goBack(): 回退到上一个历史记录
   6. history.goForword(): 前进到下一个历史记录
   7. history.listen(function(location){}): 监视历史记录的变化
3. 测试

|  |
| --- |
| <!DOCTYPE **html**> <**html lang="en"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>history test</**title**> </**head**> <**body**>  <**p**><**input type="text"**></**p**>  <**a href="/test1" onclick="return** *push*(**'/test1'**)**"**>test1</**a**><**br**><**br**>  <**button onClick="***push*(**'/test2'**)**"**>push test2</**button**><**br**><**br**>  <**button onClick="***back*()**"**>回退</**button**><**br**><**br**>  <**button onClick="***forword*()**"**>前进</**button**><**br**><**br**>  <**button onClick="***replace*(**'/test3'**)**"**>replace test3</**button**><**br**><**br**>   <**script type="text/javascript" src="https://cdn.bootcss.com/history/4.7.2/history.js"**></**script**>  <**script type="text/javascript"**>  **let *history*** = **History**.createBrowserHistory() *// 方式一  // history = History.createHashHistory() // 方式二  // console.log(history)* **function** *push* (to) {  ***history***.push(to)  **return false** }   **function** *back*() {  ***history***.goBack()  }   **function** *forword*() {  ***history***.goForward()  }   **function** *replace* (to) {  ***history***.replace(to)  }   ***history***.listen((location) => {  **console**.log(**'请求路由路径变化了'**, location)  })  </**script**> </**body**> </**html**> |

## 6.2. react-router相关API

### 6.2.1. 组件

1. <BrowserRouter>
2. <HashRouter>
3. <Route>
4. <Redirect>
5. <Link>
6. <NavLink>
7. <Switch>

### 6.2.2. 其它

1. history对象
2. match对象
3. withRouter函数

## 6.3. 基本路由使用

### 6.3.1. 效果



### 6.3.2. 准备

1. 下载react-router-dom: npm install --save react-router-dom@4
2. 引入bootstrap.css: <link rel="stylesheet" href="/css/bootstrap.css">

### 6.3.3. 路由组件: views/about.jsx

|  |
| --- |
| **import** React **from 'react' export default function** About() {  **return** <**div**>About组件内容</**div**> } |

### 6.3.4. 路由组件: views/home.jsx

|  |
| --- |
| **import** React **from 'react' export default function Home**() {  **return** <**div**>Home组件内容</**div**> } |

### 6.3.5. 包装NavLink组件: components/my-nav-link.jsx

|  |
| --- |
| **import** React **from 'react' import** {**NavLink**} **from 'react-router-dom'  export default function** MyNavLink(props) {  **return** <**NavLink** {...props} **activeClassName='activeClass'**/> } |

### 6.3.6. 应用组件: components/app.jsx

|  |
| --- |
| **import** React **from 'react' import** {**Route**, **Switch**, **Redirect**} **from 'react-router-dom' import** *MyNavLink* **from './components/my-nav-link' import** *About* **from './views/about' import** *Home* **from './views/home'  export default class** App **extends** React.Component {   render () {  **return** (  <**div**>   <**div className="row"**>  <**div className="col-xs-offset-2 col-xs-8"**>  <**div className="page-header"**>  <**h2**>React Router Demo</**h2**>  </**div**>  </**div**>  </**div**>   <**div className="row"**>  <**div className="col-xs-2 col-xs-offset-2"**>  <**div className="list-group"**>  {*/\*导航路由链接\*/*}  <**MyNavLink className="list-group-item" to='/about'** >About</**MyNavLink**>  <**MyNavLink className="list-group-item" to='/home'**>Home</**MyNavLink**>  </**div**>  </**div**>  <**div className="col-xs-6"**>  <**div className="panel"**>  <**div className="panel-body"**>  {*/\*可切换的路由组件\*/*}  <**Switch**>  <**Route path='/about' component=**{*About*} />  <**Route path='/home' component=**{*Home*} />  <**Redirect to='/about'** />  </**Switch**>  </**div**>  </**div**>  </**div**>  </**div**>  </**div**>  )  } } |

### 6.3.7. 自定义样式: index.css

|  |
| --- |
| .**activeClass** {  **color**: **red !important**; } |

### 6.3.8. 入口JS: index.js

|  |
| --- |
| **import** React **from 'react' import** ReactDOM **from 'react-dom' import** {**BrowserRouter**, **HashRouter**} **from 'react-router-dom' import** App **from './components/app'  import './index.css'** ReactDOM.render(  (  <**BrowserRouter**>  <**App** />  </**BrowserRouter**>  */\*<HashRouter>  <App />  </HashRouter>\*/* ),   **document**.getElementById(**'root'**) ) |

## 6.4. 嵌套路由使用

### 6.4.1. 效果



### 6.4.2. 二级路由组件: views/news.jsx

|  |
| --- |
| **import** React **from 'react' export default class** News **extends** React.Component {  **state** = {  **newsArr**: [**'news001'**, **'news002'**, **'news003'**]  }   render () {  **return** (  <**div**>  <**ul**>  {  **this**.**state**.**newsArr**.map((news, index) => <**li key=**{index}>{news}</**li**>)  }  </**ul**>  </**div**>  )  } } |

### 6.4.3. 二级路由组件: views/message.jsx

|  |
| --- |
| **import** React **from 'react' import** {**Link**, **Route**} **from 'react-router-dom'  export default class** Message **extends** React.Component {  **state** = {  **messages**: []  }   componentDidMount () {  *// 模拟发送ajax请求* setTimeout(() => {  **const** data = [  {**id**: 1, **title**: **'Message001'**},  {**id**: 3, **title**: **'Message003'**},  {**id**: 6, **title**: **'Message006'**},  ]  **this**.setState({  **messages**: data  })  }, 1000)  }    render () {  **const** path = **this**.**props**.**match**.**path   return** (  <**div**>  <**ul**>  {  **this**.**state**.**messages**.map((m, index) => {  **return** (  <**li key=**{index}>  <**Link to='???'**>{m.**title**}</**Link**>  </**li**>  )  })  }  </**ul**>  </**div**>  )  } } |

### 6.4.4. 一级路由组件: views/home.jsx

|  |
| --- |
| **import** React **from 'react' import** {**Switch**, **Route**, **Redirect**} **from 'react-router-dom' import** *MyNavLink* **from './components/my-nav-link' import** News **from './views/news' import** Message **from './views/message'  export default function** Home() {  **return** (  <**div**>  <**h2**>Home组件内容</**h2**>  <**div**>  <**ul className="nav nav-tabs"**>  <**li**>  <**MyNavLink to='/home/news'**>News</**MyNavLink**>  </**li**>  <**li**>  <**MyNavLink to="/home/message"**>Message</**MyNavLink**>  </**li**>  </**ul**>  <**Switch**>  <**Route path='/home/news' component=**{News} />  <**Route path='/home/message' component=**{Message} />  <**Redirect to='/home/news'**/>  </**Switch**>  </**div**>  </**div**>  ) } |

## 6.5. 向路由组件传递参数数据

### 6.5.1. 效果



### 6.5.2. 三级路由组件: views/message-detail.jsx

|  |
| --- |
| **import** React **from 'react' const** messageDetails = [  {**id**: 1, **title**: **'Message001'**, **content**: **'我爱你, 中国'**},  {**id**: 3, **title**: **'Message003'**, **content**: **'我爱你, 老婆'**},  {**id**: 6, **title**: **'Message006'**, **content**: **'我爱你, 孩子'**}, ] **export default function** MessageDetail(props) {   **const** id = props.**match**.params.**id  const** md = messageDetails.find(md => md.**id**===id\*1)   **return** (  <**ul**>  <**li**>ID: {md.**id**}</**li**>  <**li**>TITLE: {md.**title**}</**li**>  <**li**>CONTENT: {md.**content**}</**li**>  </**ul**>  ) } |

### 6.5.3. 二级路由组件: views/message.jsx

|  |
| --- |
| **import** React **from 'react' import** {**Link**, **Route**} **from 'react-router-dom' import** *MessageDetail* **from "./views/message-detail"  export default class** Message **extends** React.Component {  **state** = {  **messages**: []  }   componentDidMount () {  *// 模拟发送ajax请求* setTimeout(() => {  **const** data = [  {**id**: 1, **title**: **'Message001'**},  {**id**: 3, **title**: **'Message003'**},  {**id**: 6, **title**: **'Message006'**},  ]  **this**.setState({  **messages**: data  })  }, 1000)  }   render () {  **const** path = **this**.**props**.**match**.**path   return** (  <**div**>  <**ul**>  {  **this**.**state**.**messages**.map((m, index) => {  **return** (  <**li key=**{index}>  <**Link to=**{**`**${path}**/**${m.**id**}**`**}>{m.**title**}</**Link**>  </**li**>  )  })  }  </**ul**>  <**hr**/>  <**Route path=**{**`**${path}**/:id`**} **component=**{*MessageDetail*}></**Route**>  </**div**>  )  } } |

## 6.6. 多种路由跳转方式

### 6.6.1. 效果



### 6.6.2. 二级路由: views/message.jsx

|  |
| --- |
| **import** React **from 'react' import** {**Link**, **Route**} **from 'react-router-dom' import** *MessageDetail* **from "./views/message-detail"  export default class** Message **extends** React.Component {  **state** = {  **messages**: []  }   componentDidMount () {  *// 模拟发送ajax请求* setTimeout(() => {  **const** data = [  {**id**: 1, **title**: **'Message001'**},  {**id**: 3, **title**: **'Message003'**},  {**id**: 6, **title**: **'Message006'**},  ]  **this**.setState({  **messages**: data  })  }, 1000)  }   *ShowDetail* = (id) => {  **this**.**props**.**history**.**push**(**`/home/message/**${id}**`**)  }   *ShowDetail2* = (id) => {  **this**.**props**.**history**.**replace**(**`/home/message/**${id}**`**)  }   *back* = () => {  **this**.**props**.**history**.**goBack**()  }   *forward* = () => {  **this**.**props**.**history**.**goForward**()  }   render () {  **const** path = **this**.**props**.**match**.**path   return** (  <**div**>  <**ul**>  {  **this**.**state**.**messages**.map((m, index) => {  **return** (  <**li key=**{index}>  <**Link to=**{**`**${path}**/**${m.**id**}**`**}>{m.**title**}</**Link**>  **&nbsp;** <**button onClick=**{() => **this**.*ShowDetail*(m.**id**)}>查看详情(push)</**button**>**&nbsp;** <**button onClick=**{() => **this**.*ShowDetail2*(m.**id**)}>查看详情(replace)</**button**>  </**li**>  )  })  }  </**ul**>  <**p**>  <**button onClick=**{**this**.*back*}>返回</**button**>**&nbsp;** <**button onClick=**{**this**.*forward*}>前进</**button**>**&nbsp;** </**p**>  <**hr**/>  <**Route path=**{**`**${path}**/:id`**} **component=**{*MessageDetail*}></**Route**>  </**div**>  )  } } |

## 6.7. 面试题: 说说对react-router的理解

|  |
| --- |
| *1). 下载: npm install --save react-router-dom 2). 是什么: 用来实现SPA的react插件 3). 相关API:  a. 组件  <HashRouter> / <BrowserRouter>  <Route>  <Redirect>  <NavLink> / <Link>  <Switch>  b. 对象或函数  props.history对象  props.match对象  props.location对象  withRouter函数* |

# 第7章：react UI组件库

## 7.1. 最流行的开源React UI组件库

### 7.1.1. material-ui(国外)

1. 官网: <http://www.material-ui.com/#/>
2. github: <https://github.com/callemall/material-ui>

### 7.1.2. ant-design(国内蚂蚁金服)

1. PC官网: <https://ant.design/index-cn>
2. 移动官网: <https://mobile.ant.design/index-cn>
3. Github: <https://github.com/ant-design/ant-design/>
4. Github: <https://github.com/ant-design/ant-design-mobile/>

## 7.2. ant-design使用入门

### 7.2.1. 下载组件库包

|  |
| --- |
| **yarn add antd** |

### 7.2.2. 基本使用

1. App.jsx

|  |
| --- |
| **import** React, {Component} **from 'react'** **import** {Button, message} **from 'antd'  class** App **extends** Component {   *handleClick* = () => {  message.info(**'点击了'**)  }    render() {  **return** (  <**div className="App"**>  <**Button type="primary" onClick=**{**this**.*handleClick*}>Button</**Button**>  </**div**>  );  } }  **export default** App |

1. index.js

|  |
| --- |
| *// 引入antd整个css* **import 'antd/dist/antd.min.css'** |

1. 问题

需要手动引入样式

会打包antd所有组件的js/css

### 7.2.3. 实现组件的按需打包

1. 下载依赖模块

|  |
| --- |
| **yarn add react-app-rewired@1.6.2 babel-plugin-import -D** |

1. 定义加载配置的js模块: config-overrides.js

|  |
| --- |
| **const** {injectBabelPlugin} = require(**'react-app-rewired'**)  module.exports = **function** override(config, env) {  config = injectBabelPlugin(  [**'import'**, {**libraryName**: **'antd'**, **libraryDirectory**: **'es'**, **style**: **'css'**}],  config,  )  **return** config } |

1. 修改配置: package.json

|  |
| --- |
| **"scripts"**: {  **"start"**: **"react-app-rewired start"**,  **"build"**: **"react-app-rewired build"**,  **"test"**: **"react-app-rewired test"**,  **"eject"**: **"react-scripts eject"** } |

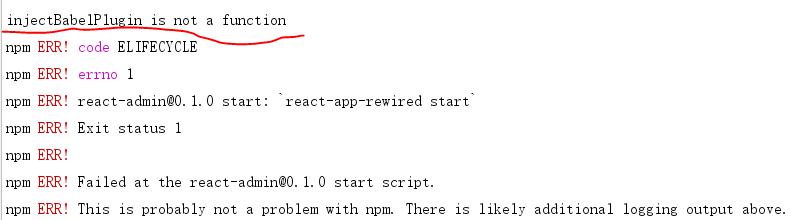
1. 去除样式的引入: index.js

|  |
| --- |
| *// import 'antd/dist/antd.min.css'* |

### 7.2.4. 解决打包异常(工具包版本的问题)

1. 问题1:

**错误:**



**原因:**

react-app-rewired的版本高(2.0.1)了, 不兼容

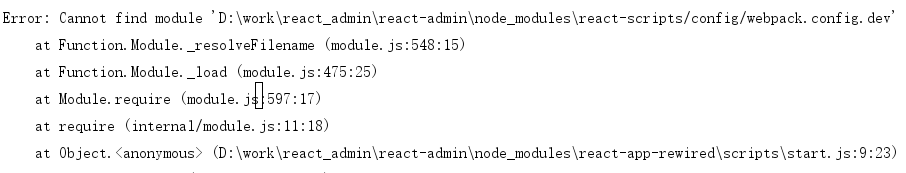
**解决:**

卸载后下载一个低点的版本(1.6.2)

|  |
| --- |
| **yarn remove react-app-rewired**  **yarn add react-app-rewired@1.6.2 -D** |

1. 问题2:

**错误:**



**原因:**

react-scripts的版本高(2.1.3)了, 不兼容

**解决:**

卸载后下载一个低点的版本(2.1.1)

|  |
| --- |
| **yarn remove react-scripts**  **yarn add** [**react-scripts@2.1.1**](mailto:react-scripts@2.1.1) **-D** |

### 7.2.5. 自定义antd主题

需求:

使antd的默认基本颜色从Blue变为Green

实现:

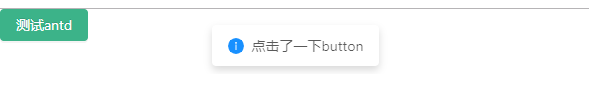
下载工具包:

|  |
| --- |
| **yarn add react-app-rewire-less -D** |

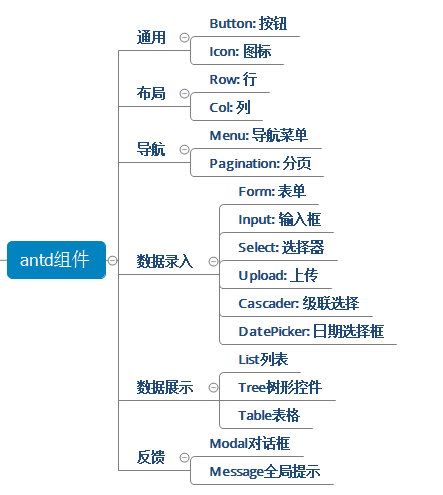
修改config-overrides.js

|  |
| --- |
| **const** rewireLess = require(**'react-app-rewire-less'**) **const** {*injectBabelPlugin*} = require(**'react-app-rewired'**);  **module**.exports = **function** override(config, env) {  config = *injectBabelPlugin*(  [**'import'**, {**libraryName**: **'antd'**, **libraryDirectory**: **'es'**, **style**: **true**}],  config,  );   config = rewireLess.withLoaderOptions({  **modifyVars**: {**"@primary-color"**: **"#1DA57A"**},  **javascriptEnabled**: **true**,  })(config, env);   **return** config; }; |

### 7.2.6. 应用中使用的组件



### 7.2.7. 项目中用到的antd组件



# 第8章：redux

## 8.1. redux理解

### 8.1.1. 学习文档

1. 英文文档: <https://redux.js.org/>
2. 中文文档: <http://www.redux.org.cn/>
3. Github: <https://github.com/reactjs/redux>

### 8.1.2. redux是什么?

1. redux是一个独立专门用于做状态管理的JS库(不是react插件库)
2. 它可以用在react, angular, vue等项目中, 但基本与react配合使用
3. 作用: 集中式管理react应用中多个组件共享的状态

### 8.1.3. redux工作流程



### 8.1.4. 什么情况下需要使用redux

1. 总体原则: 能不用就不用, 如果不用比较吃力才考虑使用
2. 某个组件的状态，需要共享
3. 某个状态需要在任何地方都可以拿到
4. 一个组件需要改变全局状态
5. 一个组件需要改变另一个组件的状态

## 8.2. redux的核心API

### 8.2.1. createStore()

1. 作用:

创建包含指定reducer的store对象

1. 编码:

import {createStore} from 'redux'

import counter from './reducers/counter'

const store = createStore(counter)

### 8.2.2. store对象

1. 作用:

redux库最核心的管理对象

1. 它内部维护着:

state

reducer

1. 核心方法:

getState()

dispatch(action)

subscribe(listener)

1. 编码:

store.getState()

store.dispatch({type:'INCREMENT', number})

store.subscribe(render)

### 8.2.3. applyMiddleware()

1. 作用:

应用上基于redux的中间件(插件库)

1. 编码:

import {createStore, applyMiddleware} from 'redux'

import thunk from 'redux-thunk' // redux异步中间件

const store = createStore(

counter,

applyMiddleware(thunk) // 应用上异步中间件

)

### 8.2.4. combineReducers()

1. 作用:

合并多个reducer函数

1. 编码:

export default combineReducers({

user,

chatUser,

chat

})

## 8.3. redux的三个核心概念

### 8.3.1. action

1. 标识要执行行为的对象
2. 包含2个方面的属性
   1. type: 标识属性, 值为字符串, 唯一, 必要属性
   2. xxx: 数据属性, 值类型任意, 可选属性
3. 例子:

const action = {

type: 'INCREMENT',

data: 2

}

1. Action Creator(创建Action的工厂函数)

const increment = (number) => ({type: 'INCREMENT', data: number})

### 8.3.2. reducer

1. 根据老的state和action, 产生新的state的纯函数
2. 样例

export default function counter(state = 0, action) {

switch (action.type) {

case 'INCREMENT':

return state + action.data

case 'DECREMENT':

return state - action.data

default:

return state

}

}

1. 注意
   1. 返回一个新的状态
   2. 不要修改原来的状态

### 8.3.3. store

1. 将state,action与reducer联系在一起的对象
2. 如何得到此对象?

import {createStore} from 'redux'

import reducer from './reducers'

const store = createStore(reducer)

1. 此对象的功能?

getState(): 得到state

dispatch(action): 分发action, 触发reducer调用, 产生新的state

subscribe(listener): 注册监听, 当产生了新的state时, 自动调用

## 8.4. 使用redux编写应用

### 8.4.1. 效果



### 8.4.2. 下载依赖包

npm install --save redux

### 8.4.3. redux/action-types.js

|  |
| --- |
| */\* action对象的type常量名称模块  \*/* **export const** INCREMENT = **'increment' export const** DECREMENT = **'decrement'** |

### 8.4.4. redux/actions.js

|  |
| --- |
| */\* action creator模块  \*/* **import** {INCREMENT, DECREMENT} **from './action-types'  export const** *increment* = number => ({**type**: INCREMENT, number}) **export const** *decrement* = number => ({**type**: DECREMENT, number}) |

### 8.4.5. redux/reducers.js

|  |
| --- |
| */\* 根据老的state和指定action, 处理返回一个新的state  \*/* **import** {INCREMENT, DECREMENT} **from '../constants/ActionTypes'  import** {INCREMENT, DECREMENT} **from './action-types'  export function** *counter*(state = 0, action) {  **console**.log(**'counter'**, state, action)  **switch** (action.**type**) {  **case** INCREMENT:  **return** state + action.**number  case** DECREMENT:  **return** state - action.**number  default**:  **return** state  } } |

### 8.4.6. components/app.jsx

|  |
| --- |
| */\* 应用组件  \*/* **import** React, {Component} **from 'react' import** PropTypes **from 'prop-types' import** \* **as** actions **from '../redux/actions'  export default class** App **extends** Component {   **static** *propTypes* = {  **store**: PropTypes.*object*.isRequired,  }   *increment* = () => {  **const** number = **this**.**refs**.numSelect.**value** \* 1  **this**.**props**.**store**.dispatch(actions.*increment*(number))  }   *decrement* = () => {  **const** number = **this**.**refs**.numSelect.**value** \* 1  **this**.**props**.**store**.dispatch(actions.*decrement*(number))  }   *incrementIfOdd* = () => {  **const** number = **this**.**refs**.numSelect.**value** \* 1   **let** count = **this**.**props**.**store**.getState()  **if** (count % 2 === 1) {  **this**.**props**.**store**.dispatch(actions.*increment*(number))  }  }   *incrementAsync* = () => {  **const** number = **this**.**refs**.numSelect.**value** \* 1  setTimeout(() => {  **this**.**props**.**store**.dispatch(actions.*increment*(number))  }, 1000)  }   render() {  **return** (  <**div**>  <**p**>  click {**this**.**props**.**store**.getState()} times {**' '**}  </**p**>  <**select ref="numSelect"**>  <**option value="1"**>1</**option**>  <**option value="2"**>2</**option**>  <**option value="3"**>3</**option**>  </**select**>{**' '**}  <**button onClick=**{**this**.*increment*}>+</**button**>  {**' '**}  <**button onClick=**{**this**.*decrement*}>-</**button**>  {**' '**}  <**button onClick=**{**this**.*incrementIfOdd*}>increment if odd</**button**>  {**' '**}  <**button onClick=**{**this**.*incrementAsync*}>increment async</**button**>  </**div**>  )  } } |

### 8.4.7. index.js

|  |
| --- |
| **import** React **from 'react' import** ReactDOM **from 'react-dom' import** {createStore} **from 'redux'  import** App **from './components/app' import** {*counter*} **from './redux/reducers'** *// 根据counter函数创建store对象* **const** store = createStore(*counter*)  *// 定义渲染根组件标签的函数* **const** *render* = () => {  ReactDOM.render(  <**App store=**{store}/>,  **document**.getElementById(**'root'**)  ) } *// 初始化渲染 render*()  *// 注册(订阅)监听, 一旦状态发生改变, 自动重新渲染* store.subscribe(*render*) |

### 8.4.8. 问题

1. redux与react组件的代码耦合度太高
2. 编码不够简洁

## 8.5. react-redux

### 8.5.1. 理解

1. 一个react插件库
2. 专门用来简化react应用中使用redux

### 8.5.2. React-Redux将所有组件分成两大类

1. UI组件
   1. 只负责 UI 的呈现，不带有任何业务逻辑
   2. 通过props接收数据(一般数据和函数)
   3. 不使用任何 Redux 的 API
   4. 一般保存在components文件夹下
2. 容器组件
   1. 负责管理数据和业务逻辑，不负责UI的呈现
   2. 使用 Redux 的 API
   3. 一般保存在containers文件夹下

### 8.5.3. 相关API

1. Provider

让所有组件都可以得到state数据

<Provider store={store}>  
    <App />  
  </Provider>

1. connect()

用于包装 UI 组件生成容器组件

import { connect } from 'react-redux'  
  connect(  
    mapStateToprops,  
    mapDispatchToProps  
  )(Counter)

1. mapStateToprops()

将外部的数据（即state对象）转换为UI组件的标签属性  
  const mapStateToprops = function (state) {  
   return {  
     value: state  
   }  
  }

1. mapDispatchToProps()

将分发action的函数转换为UI组件的标签属性

简洁语法可以直接指定为actions对象或包含多个action方法的对象

### 8.5.4. 使用react-redux

1. 下载依赖包

npm install --save react-redux

1. redux/action-types.js

不变

1. redux/actions.js

不变

1. redux/reducers.js

不变

1. components/counter.jsx

|  |
| --- |
| */\* UI组件: 不包含任何redux API  \*/* **import** React **from 'react' import** PropTypes **from 'prop-types'  export default class** Counter **extends** React.Component {   **static** *propTypes* = {  **count**: PropTypes.*number*.isRequired,  **increment**: PropTypes.*func*.isRequired,  **decrement**: PropTypes.*func*.isRequired  }   *increment* = () => {  **const** number = **this**.**refs**.numSelect.**value** \* 1  **this**.**props**.**increment**(number)  }   *decrement* = () => {  **const** number = **this**.**refs**.numSelect.**value** \* 1  **this**.**props**.**decrement**(number)  }   *incrementIfOdd* = () => {  **const** number = **this**.**refs**.numSelect.**value** \* 1  **let** count = **this**.**props**.**count  if** (count % 2 === 1) {  **this**.**props**.**increment**(number)  }  }   *incrementAsync* = () => {  **const** number = **this**.**refs**.numSelect.**value** \* 1  setTimeout(() => {  **this**.**props**.**increment**(number)  }, 1000)  }   render() {  **return** (  <**div**>  <**p**>  click {**this**.**props**.**count**} times {**' '**}  </**p**>  <**select ref="numSelect"**>  <**option value="1"**>1</**option**>  <**option value="2"**>2</**option**>  <**option value="3"**>3</**option**>  </**select**>{**' '**}  <**button onClick=**{**this**.*increment*}>+</**button**>  {**' '**}  <**button onClick=**{**this**.*decrement*}>-</**button**>  {**' '**}  <**button onClick=**{**this**.*incrementIfOdd*}>increment if odd</**button**>  {**' '**}  <**button onClick=**{**this**.*incrementAsync*}>increment async</**button**>  </**div**>  )  } } |

1. containters/app.jsx

|  |
| --- |
| */\* 包含Counter组件的容器组件  \*/* **import** React **from 'react'** *// 引入连接函数* **import** {connect} **from 'react-redux'** *// 引入action函数* **import** {*increment*, *decrement*} **from '../redux/actions'  import** Counter **from '../components/counter'** *// 向外暴露连接App组件的包装组件* **export default** connect(  state => ({**count**: state}),  {*increment*, *decrement*} )(Counter) |

1. index.js

|  |
| --- |
| **import** React **from 'react' import** ReactDOM **from 'react-dom' import** {createStore} **from 'redux' import** {Provider} **from 'react-redux'  import** App **from './containers/app' import** {*counter*} **from './redux/reducers'** *// 根据counter函数创建store对象* **const** store = createStore(*counter*)  *// 定义渲染根组件标签的函数* ReactDOM.render(  (  <**Provider store=**{store}>  <**App** />  </**Provider**>  ),  **document**.getElementById(**'root'**) |

### 8.5.5. 问题

1. redux默认是不能进行异步处理的,
2. 应用中又需要在redux中执行异步任务(ajax, 定时器)

## 8.6. redux异步编程

### 8.6.1. 下载redux插件(异步中间件)

npm install --save redux-thunk

### 8.6.2. index.js

|  |
| --- |
| **import** {createStore, *applyMiddleware*} **from 'redux' import** thunk **from 'redux-thunk'** *// 根据counter函数创建store对象* **const** store = createStore(  *counter*,  *applyMiddleware(thunk)* *// 应用上异步中间件* ) |

### 8.6.3. redux/actions.js

|  |
| --- |
| *// 异步action creator(返回一个函数)* **export const** *incrementAsync* = number => {  **return** dispatch => {  setTimeout(() => {  dispatch(*increment*(number))  }, 1000)  } } |

### 8.6.4. components/counter.jsx

|  |
| --- |
| *incrementAsync* = () => {  **const** number = **this**.**refs**.numSelect.**value**\*1  **this**.**props**.**incrementAsync**(number) } |

### 8.6.5. containers/app.jsx

|  |
| --- |
| **import** {*increment*, *decrement*, *incrementAsync*} **from '../redux/actions'**  *// 向外暴露连接App组件的包装组件* **export default** connect(  state => ({**count**: state}),  {*increment*, *decrement*, *incrementAsync*} )(Counter) |

## 8.7. 使用上redux调试工具

### 8.7.1. 安装chrome浏览器插件



### 8.7.2. 下载工具依赖包

npm install --save-dev redux-devtools-extension

### 8.7.3. 编码

|  |
| --- |
| **import** { **composeWithDevTools** } **from 'redux-devtools-extension'**  **const** store = createStore(  *counter*,  **composeWithDevTools**(*applyMiddleware*(thunk)) ) |

## 8.8. 相关重要知识: 纯函数和高阶函数

### 8.8.1. 纯函数

1. 一类特别的函数: 只要是同样的输入(实参)，必定得到同样的输出(返回)
2. 必须遵守以下一些约束
   1. 不得改写参数数据
   2. 不会产生任何副作用，例如网络请求，输入和输出设备
   3. 不能调用Date.now()或者Math.random()等不纯的方法
3. redux的reducer函数必须是一个纯函数

### 8.8.2. 高阶函数

1. 理解: 一类特别的函数
   1. 情况1: 参数是函数
   2. 情况2: 返回是函数
2. 常见的高阶函数:
   1. 定时器设置函数
   2. 数组的forEach()/map()/filter()/reduce()/find()/bind()
   3. promise
   4. react-redux中的connect函数
3. 作用:
   1. 能实现更加动态, 更加可扩展的功能