爱创课堂前端培训

# React 16

第3天课堂笔记（本课程共5天）

班级：北京前端训练营16期

讲师：张容铭

日期：2018年11月9日

张容铭老师

微博：@张容铭\_YYQH

QQ : 286031482

E-mail : yuye\_qinghe@qq.com

爱创课堂官网 ：[www.icketang.com](http://www.icketang.com)

目录

[React 16 1](#_Toc18089)

[复习 3](#_Toc23409)

[一、React 4](#_Toc4418)

[1.1组件通信 4](#_Toc29247)

[1.1.1父组件到子组件通信 4](#_Toc27436)

[1.1.2子组件向父组件通信 4](#_Toc811)

[1.1.3兄弟组件间通信 5](#_Toc28551)

[二、项目实战 6](#_Toc5975)

[2.1技术架构 6](#_Toc31597)

[2.2目录部署 6](#_Toc5637)

[2.3效果图 7](#_Toc9569)

[2.4 header组件 9](#_Toc32464)

[2.5混合 9](#_Toc11565)

[二、flux 10](#_Toc16013)

[2.1 flux组成 10](#_Toc4024)

[2.2通信流程 10](#_Toc15429)

[2.3 reflux和redux 11](#_Toc1700)

[三、reflux 11](#_Toc26343)

[3.1组成 11](#_Toc14763)

[3.2通信流程 11](#_Toc4516)

[四、redux 11](#_Toc22551)

[4.1组成部分 11](#_Toc7188)

[4.2通信流程 11](#_Toc17098)

[4.3特点 12](#_Toc3218)

# 复习

异步请求 axios

组件生命周期

创建期：1 defaultProps， 2 constructor， 3 componentWillMount， 4 render， 5 componentDidMount

存在期：componentWillReceiveProps ,shouldComponentUpdate, componentWillUpdate, render, componentDidUpdate

销毁期： componentWillUnmount

子组件

父组件到子组件通信

1 数据：属性，状态，变量等等

2 方法：作为子组件事件回调函数， 在子组件方法中使用

组件约束性

非约束性组件：defaultValue|defaultChecked, 获取 ref， 修改 ref

约束性组件：value|checked onChange， 获取状态，修改状态

下拉框 单选 字符串 多选数组

非react类库

侵入式类库

# 一、React

## 1.1组件通信

### 1.1.1父组件到子组件通信

1 传递数据：状态，属性，变量等

2 传递方法：

子组件事件回调函数

在子组件方法中执行

绑定父组件实例化对象，前者无法访问子组件实例化对象，后者可以访问子组件也可以访问父组件

如果绑定了父组件实例化对象，方法中的this指向父组件实例化对象，方法执行的时候，在方法中修改状态，修改的是谁的呢

### 1.1.2子组件向父组件通信

实现子组件向父组件通信分成三步

第一步 父组件向子组件传递方法，并绑定父组件实例化对象

第二步 子组件执行方法，并传递子组件中的数据

第三步 在父组件方法中，修改父组件状态

这样就实现了子组件到父组件的通信

|  |
| --- |
| 1. // 定义子组件 2. class Child extends Component { 3. // 子组件方法 4. childMethod(e) { 5. // console.log(e.target.value) 6. // 将数据传递给父组件 7. this.props.method(e.target.value) 8. } 9. render() { 10. // 虚拟DOM 11. return ( 12. <div> 13. {/\*作为事件回调函数执行\*/} 14. {/\*<input type="text" onChange={this.props.method}/>\*/} 15. {/\*在子组件方法中执行\*/} 16. {/\*<input type="text" onChange={this.childMethod.bind(this)}/>\*/} 17. {/\*箭头函数简化方法的定义\*/} 18. <input type="text" onChange={e => this.props.method(e.target.value)}/> 19. </div> 20. ) 21. } 22. } 23. // 定义父组件 24. class Parent extends Component { 25. // 构造函数 26. constructor(props) { 27. super(props); 28. // 初始化状态 29. this.state = { 30. msg: '' 31. } 32. } 33. // 传递的方法 34. parentMethod(msg) { 35. // console.log(e.target.value) 36. // 修改状态 37. // this.setState({ 38. // msg: e.target.value 39. // }) 40. // console.log(111, e) 41. this.setState({ msg }) 42. } 43. render() { 44. // 虚拟DOM 45. return ( 46. <div> 47. {/\*1传递方法并绑定父组件\*/} 48. {/\*<Child method={this.parentMethod.bind(this)}></Child>\*/} 49. {/\*简化方法的定义\*/} 50. <Child method={msg => this.setState({ msg })}></Child> 51. <h1>结果：{this.state.msg}</h1> 52. {/\*建议\*/} 53. <h1>{'结果：' + this.state.msg}</h1> 54. </div> 55. ) 56. } 57. } |

### 1.1.3兄弟组件间通信

父组件向子组件通信：为子组件添加属性数据

子组件向父组件通信：父组件传递方法，子组件中执行，并传递数据，父组件方法接收数据，并修改状态

如果这两个组件在同一个父组件中，就构成了兄弟组件的关系，这样就可以实现兄弟组件间的通信

兄弟组件间通信

第一步 父组件向一个子组件传递方法，并绑定父组件实例化对象

第二步 子组件执行方法，并向父组件方法中传递数据

第三步 父组件接收数据，并修改状态，存储在父组件中

第四步 父组件将数据传递给另一个子组件

兄弟组件间通信的必要条件是：具有共同的父组件

就像青年男女找对象，需要一个媒婆介绍

关于不相干的两个组件间的通信，后面我们要学习flux，reflux，redux以及观察者模式来解决。

|  |
| --- |
| 1. // 定义一个子组件 2. class Child extends Component { 3. render() { 4. // 2 子组件执行方法，并传递数据 5. return <input type="text" onChange={e => this.props.methods(e.target.value)} /> 6. } 7. } 8. // 定义另一个子组件 9. class Other extends Component { 10. render() { 11. return <h1>{this.props.msg}</h1> 12. } 13. } 14. // 定义父组件 15. class Parent extends Component { 16. // 定义状态 17. constructor(props) { 18. super(props); 19. // 初始化状态 20. this.state = { 21. msg: '' 22. } 23. } 24. render() { 25. return ( 26. <div> 27. {/\*1 父组件向子组件传递方法\*/} 28. {/\*3 更新状态\*/} 29. <Child methods={ msg => this.setState({ msg }) }></Child> 30. {/\*4 父组件将数据传递给另外一个组件\*/} 31. <Other msg={this.state.msg}></Other> 32. </div> 33. ) 34. } 35. } |

## 二、项目实战

## 2.1技术架构

react, bootstrap, less, es6, ES Module, webpack, 实现pc端单页面应用程序

## 2.2目录部署

data mock数据（模拟数据）

css 样式库

img 所有图片

modules 所有模块

app 应用程序目录

components 所有组件目录

pages 所有页面目录

main.jsx 入口文件

base.less 样式库文件

index.html 项目入口文件

webpack.config.js配置文件

## 2.3效果图



|  |
| --- |
| 1. import React, { Component } from 'react'; 2. // 引入模块 3. // import Header from '../components/header/header' 4. import Header from '../components/header' 5. // 引入页面 6. import Home from '../pages/home/home' 7. // import Start from '../pages/start/start' 8. // import Css from '../pages/css/css' 9. // 抽象页面组件 10. import Page from '../pages/page/page'; 11. // 引入属性约束模块 12. // import PropTypes from 'prop-types'; 13. import './app.less'; 14. // 创建应用程序组件 15. export default class App extends Component { 16. // 构造函数中 17. constructor(props) { 18. super(props); 19. // 将外部的转化成内部 20. this.state = { 21. [props.page]: true 22. } 23. } 24. // 获取选中的页面 25. getChoosePage() { 26. // 遍历state 27. for (let key in this.state) { 28. // 如果是true，就是被选中的页面 29. // key表示页面的名称 30. if (this.state[key]) { 31. return key 32. } 33. } 34. } 35. // 父组件方法，切换页面 36. parentMethod(page) { 37. // console.log(page) 38. // 排他法切换页面 39. let state = this.state; 40. // 清空页面 41. for (let key in state) { 42. // 清空 43. state[key] = false 44. } 45. // console.log(this, state, page) 46. // 设置显示的页面 47. state[page] = true; 48. // 更新状态 49. this.setState(state) 50. } 51. render() { 52. // 解构状态中的数据 53. let { home, start, css, component, js, marker } = this.state; 54. return ( 55. <div> 56. {/\*头部\*/} 57. {/\*父组件向子组件通信，传递选中的页面\*/} 58. {/\*切换页面是一个典型的子组件向父组件通信模型\*/} 59. {/\*传递方法，一定要绑定父组件实例化对象\*/} 60. <Header choosePage={this.getChoosePage()} method={this.parentMethod.bind(this)}></Header> 61. <Home style={home}></Home> 62. {/\*<Start style={start}></Start>\*/} 63. {/\*<Css style={css}></Css>\*/} 64. <Page style={start} title="起步" intro="简要介绍 Bootstrap，以及如何下载、使用，还有基本模版和案例，等等。" url="data/start.json"></Page> 65. <Page style={css} title="全局 CSS 样式" intro="设置全局 CSS 样式；基本的 HTML 元素均可以通过 class 设置样式并得到增强效果；还有先进的栅格系统。" url="data/css.json"></Page> 66. {/\*component页面\*/} 67. <Page style={component} title="组件" intro="无数可复用的组件，包括字体图标、下拉菜单、导航、警告框、弹出框等更多功能。" url="data/component.json"></Page> 68. <Page style={js} title="JavaScript 插件" intro="jQuery 插件为 Bootstrap 的组件赋予了“生命”。可以简单地一次性引入所有插件，或者逐个引入到你的页面中。" url="data/js.json"></Page> 69. {/\*marker页面\*/} 70. <Page style={marker} title="定制并下载 Bootstrap" intro="通过自定义 Bootstrap 组件、Less 变量和 jQuery 插件，定制一份属于你自己的 Bootstrap 版本吧。" url="data/marker.json"></Page> 71. </div> 72. ) 73. } 74. } 75. // 属性约束 76. // App.propTypes = { 77. // // 是字符串，必填 78. // page: PropTypes.string.isRequired 79. // } 80. // 默认属性 81. App.defaultProps = { 82. page: 'home' 83. } |

## 2.4 header组件

不论是在commonjs规范中，还是在ES Module规范中，引入模块，我们要书写目录名称

如果目录中，没有其他组件，我们可以将该文件名定义成index

这样。引入目录名，即可引入这个文件了

|  |
| --- |
| 1. // 引入模块 2. import Header from '../components/header/header' 3. import Header from '../components/header' |

## 2.5混合

react提供了组件，是为了复用虚拟DOM

react提供了混合，是为了复用组件的方法

es5开发中，我们为组件定义mixins属性，是一个数组，即可让组件继承mixins每个成员对象中的属性和方法，类似多继承

es6开发中，已经存在了继承，是extends关键字，因此es6开发中，react就移除了mixins属性

在ES6中使用混合（继承），分成两步

第一步 定义一个混合类，继承Component组件基类

第二步 定义一个组件类，继承混合类

这样就可以让组件类继承混合类以及Component组件基类了

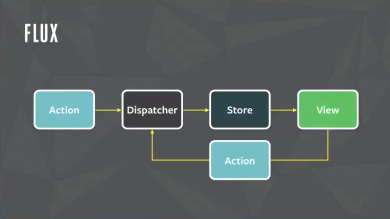
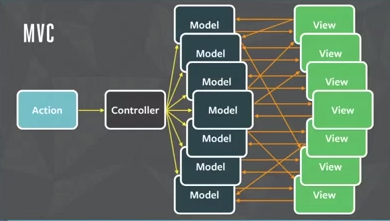
继承后，我们可以重新定义同名属性或者方法，来覆盖继承的

此时组件执行的时候，就会优先使用我们重定义的。

|  |
| --- |
| 1. import React, { Component } from 'react'; 2. import axios from 'axios'; 3. // 引入组件 4. import Banner from '../../components/banner/banner' 5. import Content from '../../components/content/content' 6. // 定义组件 7. export default class Page extends Component { 8. // 构造函数 9. constructor(props) { 10. super(props); 11. // 状态 12. this.state = { 13. data: [] 14. } 15. } 16. // 创建组件，发送请求 17. componentWillMount() { 18. // 请求数据 19. axios.get(this.props.url) 20. // 监听数据返回并存储数据 21. .then(({ data }) => this.setState({ data })) 22. } 23. render() { 24. // 解构属性数据 25. let { title, intro, style } = this.props; 26. return ( 27. <div style={{ display: style ? 'block' : 'none' }}> 28. <Banner title={title} intro={intro}></Banner> 29. <Content data={this.state.data}></Content> 30. </div> 31. ) 32. } 33. } |

# 二、flux

react是一个视图层面的框架，因此对数据的处理很弱，例如，发送请求，我们要使用axios框架，并且实现组件通信的成本也很高。因此react团队就提出了flux思想，是在MVC基础之上衍生



在经典的MVC框架中，M（模型Model）与V（视图View）是可以相互通信的，因此当系统足够复杂的时候，就形成了一个网状的结构，此时我们新增一个模块或者移除一个模块，由于模块之间的耦合关系，成本很高。

所以React团队提出了一个新的思路：flux框架，将各个模块的通信变成单向环状的，因此即使项目中这种环再多，我们也可以抽象成一个。这样就容易维护了

## 2.1 flux组成

由四部分组成

actions：用户或者组件发布的消息

dispatcher：用来捕获消息的模块，并根据消息类型处理数据

store：用来存储数据的，dispatcher将处理结果存储在store中

view：组件视图页面

## 2.2通信流程

一个组件发布一个消息

消息被dispatcher捕获到。并根据消息类型处理数据

将数据处理的结果存储在store中，

store数据更新，提供给其他组件去更新

早期React团队只是提出了这个架构，并开源实现的框架。因此很多工程师根据这一思想实现了很多数据管理的框架，如reflux，redux， vuex等等，他们都是基于观察者模式实现。

## 2.3 reflux和redux

reflux和redux都是实现组件间通信的，在react中应用的时候

reflux是基于组件的状态数据实现组件间通信的。

主要应用在ES5开发中。

redux是基于组件的属性数据实现组件间通信的（工作中用的比较多）。

主要应用在ES6开发中。

不论是哪种框架，在设计之初，都是希望可以在各个框架中使用，因此就抽象出来了，所以在各个框架中使用的时候，还要使用他们相应的插件。早期只能应用react中。

# 三、reflux

reflux是对flux的简化，只有三个模块了

## 3.1组成

actions 组件或者用户发布的消息

store 用来捕获这些消息，并根据消息内容处理数据，并存储

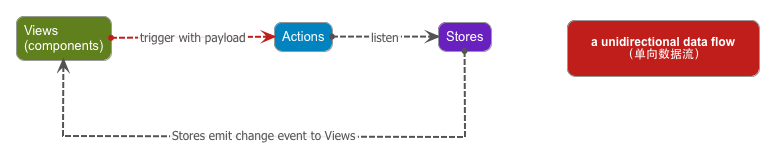
view 组件使用，接收store中的数据，更新。

## 3.2通信流程

一个组件发布一个消息

消息被store捕获，并处理

store将数据处理的结果传递给另一个组件了



# 四、redux

redux也是基于flux思想实现的一个框架，跟flux相似，也是由四部分组成

## 4.1组成部分

actions 组件或者用户发布的消息

reducers 接收actions，并根据action处理数据

store 存储数据的，接收reducer更新的数据

view 视图组件，接收store更新的数据

## 4.2通信流程

一个组件发布一个action

action通过store，被reducer捕获到

reducer处理数据，并将处理的结果传递给store

store数据更新，将数据传递给另一个组件

## 4.3特点

redux是严格按照flux思想实现的框架，因此遵守三条原则

1 单一数据源

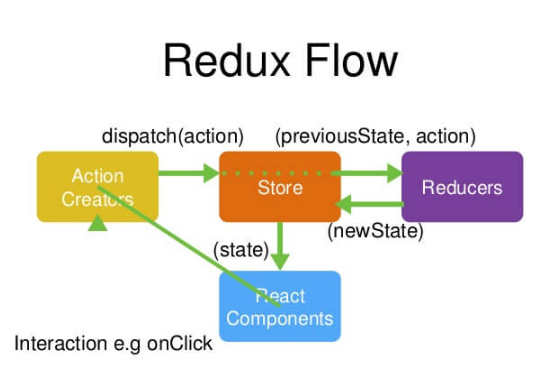
一个应用系统中，只能有且只有一个store对象

2 state是只读的

store中通过state存储数据，state数据是只读的（可以通过getState方法获取），我们不能直接修改，想修改state数据，只能发布actions在reducers中修改

3 函数式编程

每个reducer是用来捕获actions消息，并处理数据的，作者为了简化我们的操作，建议我们函数式编程，也就是收一个reducer就是一个函数，在函数内部实现对数据的修改。



|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |