**React推荐使用jsx或者js文件来表示组件，react支持class组件和function组件2种形式，react使用{}包裹变量，这点需要注意。**

React 一直都提倡使用函数组件，但是有时候需要使用 state 或者其他一些功能时，只能使用类组件，因为函数组件没有实例，没有生命周期函数，只有类组件才有,所以才有 Hooks ,让你在不编写 class 的情况下使用 state 以及其他的 React 特性

**（1）class组件**

import React from 'react';

export default class NewComponent extends React.Component {

constructor(props) {

super(props);

this.state = {

name: 'xx'

};

this.handleClick = this.handleClick.bind(this);

}

handleClick() {

this.setState({

num: this.state.num + 1,

})

}

render() {

rerurn(<div onclick=this.handleClick>{name}</div>);

}

}

// export default NewComponent;

**类组件的不足:**

①状态逻辑难复用： 在组件之间复用状态逻辑很难，可能要用到 render props

（渲染属性）或者 HOC（高阶组件），但无论是渲染属性，还是高阶组件，都会在原先的组件外包裹一层父容器（一般都是 div 元素），导致层级冗余

②趋向复杂难以维护：

1).在生命周期函数中混杂不相干的逻辑（如：在 componentDidMount 中注册事件以及其他的逻辑，在 componentWillUnmount 中卸载事件，这样分散不集中的写法，很容易写出 bug ）

2).类组件中到处都是对状态的访问和处理，导致组件难以拆分成更小的组件

③this 指向问题：

父组件给子组件传递函数时，必须绑定 this

**（2）function组件**

hooks的出现赋予了function组件管理state的能力。

import React, { useState } from 'react';

export default function NewComponent() {

const [name, setName] = useState('');

return (<div>{name}</div>);

}

//export default NewComponent;

**React props**（属性）

react中的props也与vue一样可以传递静态或动态props，静态props一般传递字符串。

state 更新和修改数据（可以根据与用户交互来改变可以根据与用户交互来改变）

props 传递数据

**defaultProps** 组件名.defaultProps = {XX: 'xx'};

static defaultProps ={XX: 'xx'};

**propTypes** 组件名.propTypes= {XX:PropTypes.string.isRequired }

static propTypes = {XX: PropTypes.string.isRequired }

1. 函数组件获取props(使用props参数获取父组件传下来的props)：

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

静态props：

const element = <Welcome name="Sara" />;

1. class组件使用this.props获取组件props：

class Welcome extends React.Component {

constructor(props) {

super(props);

}

render() {

const { name } = this.props;

return <div>{name}</div>;

}

}

动态props：

<Welcome name={name} />

**React State**(状态)

class组件使用state来管理组件内的数据

this.state={ X:x }

this.setState({X:x})

hooks的出现使得函数组件也具备管理state的能力

useState

1. class组件state

class组件在构造函数（constructor）中定义组件内数据（state），修改数据必须通过setState修改，不能直接修改state，这点非常重要。

class Welcome extends React.Component {

constructor(props) {

super(props);

this.state = {

name: 'xx'

};

this.changeName = this.changeName.bind(this);

}

changeName() {

this.setState({

name: 'new name'

});

}

render() {

const { name } = this.state;

return <div onClick={this.changeName}>{name}</div>;

}

}

关于class组建的setState有以下两点说明：

1.setState更新是异步的，但是在setTimeout和原生事件中是同步的。

2.setState更新的是组件的部分数据，react会自动将数据合并。

当需要使用上一个state值时，可以让 setState() 接收一个函数而不是一个对象。这个函数用上一个 state 作为第一个参数，将此次更新被应用时的 props 做为第二个参数：

this.setState((state, props) => ({

counter: state.counter + props.increment

}));

1. function组件useState

react 16.0之前函数组件只是纯的渲染组件，hooks的出现赋予了函数组件管理state的能力。

useState返回一个state，以及更新state的函数。如果新的 state 需要通过使用先前的 state 计算得出，那么可以将函数传递给 setState。该函数将接收先前的 state，并返回一个更新后的值

import React, { useState } from 'react';

function Counter({initialCount}) {

const [count, setCount] = useState(initialCount);

return (

<>

Count: {count}

<button onClick={() => setCount(initialCount)}>Reset</button>

<button onClick={() => setCount(prevCount => prevCount - 1)}>-</button>

<button onClick={() => setCount(prevCount => prevCount + 1)}>+</button>

</>

);

}

关于setState有以下三点说明：

1.与 class 组件中的 setState 方法不同，useState 不会自动合并更新对象。

2.只能在函数最外层调用 Hook。不要在循环、条件判断或者子函数中调用。

3.只能在 React 的函数组件或自定义hook中调用 Hook。不要在其他 JavaScript 函数中调用。

**每次渲染都是独立的闭包**

闭包作用：

读取函数内部的变量

使这些变量常驻内存（延长外部函数局部变量的生命周期）

缺点：

常驻内存，增加内存使用量。

使用不当会很容易造成内存泄露。

- 每一次渲染都有它自己的 Props 和 State

- 每一次渲染都有它自己的事件处理函数

- 当点击更新状态的时候，函数组件都会重新被调用，那么每次渲染都是独立的，取到的值不会受后面操作的影响

function Counter2(){

let [number,setNumber] = useState(0);

function alertNumber(){

setTimeout(()=>{

// alert 只能获取到点击按钮时的那个状态

alert(number);

},3000);

}

return (

<>

<p>{number}</p>

<button onClick={()=>setNumber(number+1)}>+</button>

<button onClick={alertNumber}>alertNumber</button>

</>

)

}

**惰性初始化 state**

- initialState 参数只会在组件的初始化渲染中起作用，后续渲染时会被忽略

- 如果初始 state 需要通过复杂计算获得，则可以传入一个函数，在函数中计算并返回初始的 state，此函数只在初始渲染时被调用

function Counter5(props){

console.log('Counter5 render');

// 这个函数只在初始渲染时执行一次，后续更新状态重新渲染组件时，该函数就不会再被调用

function getInitState(){

return {number:props.number};

}

let [counter,setCounter] = useState(getInitState);

return (

<>

<p>{counter.number}</p>

<button onClick={()=>setCounter({number:counter.number+1})}>+</button>

<button onClick={()=>setCounter(counter)}>setCounter</button>

</>

)

}

**React Hooks**

React Hooks 的意思是，组件尽量写成纯函数，如果需要外部功能和副作用，就用钩子把外部代码"钩"进来。

React 默认提供的四个最常用的钩子。

useState()

useReducer()

userContext()

useEffect() (componentDidMount )

**useState()** 状态钩子(useState返回一个数组：一个 state，一个更新 state 的函数)

[xx,setXx]=useState('')

setXx('')

**useReducer()** action 钩子

const [state, dispatch] = useReducer(reducer, {xx:''});

// Reducer函数和状态的初始值作为参数

const reducer=(state, action) => newState // Reducer函数 （翻译：减速器）

调度函数: dispatch(action)

import React, { useReducer } from "react";

import ReactDOM from "react-dom";

import "./styles.css";

const myReducer = (state, action) => {

switch(action.type) {

case('countUp'):

return {

...state,

count: state.count + 1

}

default:

return state

}

}

function App() {

const [state, dispatch] = useReducer(myReducer, { count: 0 })

return (

<div className="App">

<button onClick={() => dispatch({ type: 'countUp' })}>

+1

</button>

<p>Count: {state.count}</p>

</div>

);

}

const rootElement = document.getElementById("root");

ReactDOM.render(<App />, rootElement);

**useContext()** 共享状态钩子 context （上下文）

import React, { useContext } from "react";

import ReactDOM from "react-dom";

import "./styles.css";

//第一步： 组件外部建立一个 Context

const Context = React.createContext({});

const A = () => {

//第三步：useContext()钩子函数用来引入 Context 对象，从中获取username属性

const { username } = useContext(Context)

return (

<div >

<p>AAAA</p>

<p>{username}</p>

</div>

)

}

const B = () => {

//第三步：useContext()钩子函数用来引入 Context 对象，从中获取username属性

const { username } = useContext(Context)

return (

<div >

<h1>BBBB</h1>

<p>{username}</p>

<p >冲冲!</p>

</div>

)

}

function App() {

return (

//第二步：XX.Provider提供了一个 Context 对象，这个对象可以被子组件共享

<Context.Provider value={{ username: 'CHJ' }}>

<div >

<A /> //子组件A

<B /> //子组件B

</div>

</Context.Provider>

);

}

const rootElement = document.getElementById("root");

ReactDOM.render(<App />, rootElement);

**useEffect()** 副作用钩子(最常见的就是向服务器请求数据)

useEffect(() => {// Async Action}, [dependencies])

第一个参数是一个函数，异步操作的代码放在里面。

第二个参数是一个数组，用于给出 Effect 的依赖项，只要这个数组发生变化，useEffect()就会执行。

渲染元素：

**ReactDOM.render(template,targetDOM)**

**react组件通信方式汇总**

### 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可以传递一般数据和函数数据, 只能一层一层传递
3. 一般数据-->父组件传递数据给子组件-->子组件读取数据
4. 函数数据-->子组件传递数据给父组件-->子组件调用函数

### 方式二:redux

**父组件向子组件通信:**

父组件通过props向子组件传递需要的信息

<Child msg={msg}/>

在子组件里直接通过props获取父组件传递过来的参数，如下：

const msg=this.props.msg

**子组件向父组件通信：**

**解决：父组件定义函数，传递给子组件，子组件调用**

①子组件更新组件状态，通过回调函数的方式传递给父组件。  
②子组件调用父组件通过props传给它的函数更新父组件state，进而完成子组件向父组件的通讯。

父组件：

import React, { Component } from 'react';

import logo from './logo.svg';

import './App.css';

//导入子组件

import Child from './child.js';

class App extends Component {

constructor(props){

super(props);

this.state = {

msg: '父组件初始msg'

}

}

//父组件回调函数，更新state，进而更新父组件。

callback=(msg)=>{

// setState方法,修改msg参数,值是由子组件传过来。

this.setState({msg});

}

render() {

return (

<div className="App">

<p>子组件传值实验: {this.state.msg}</p>

<Child callback={this.callback} ></Child>

</div>

);

}

}

export default App;

子组件：

import React from "react";

class Child extends React.Component{

constructor(props){

super(props);

this.state={

msg: '子组件msg传值'

}

}

//通过props调用回调函数传值

trans=()=>{

this.props.callback(this.state.msg);

}

render(){

return(

<div>

<button onClick={this.trans}>激发trans事件，传值给父组件</button>

</div>

)

}

}

export default Child;

**React**

# 第1章：React

## 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(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
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

|  |
| --- |
| <**script type="text/javascript" src="../js/react.development.js"**></**script**> <**script type="text/javascript" src="../js/react-dom.development.js"**></**script**> <**script type="text/javascript" src="../js/babel.min.js"**></**script**> |

### 编码

|  |
| --- |
| <**script type="text/babel"**> //必须声明babel  // 创建虚拟DOM元素  **const vDom** = <**h1**>Hello React</**h1**> // 千万不要加引号  // 渲染虚拟DOM到页面真实DOM容器中  **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
2. var element = React.createElement('h1', {id:'myTitle'},'hello')
3. 上面创建的就是一个简单的虚拟DOM对象
4. 虚拟DOM对象最终都会被React转换为真实的DOM
5. 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. 作用: 用来创建react虚拟DOM(元素)对象
4. var ele = <h1>Hello JSX!</h1>
5. 注意1: 它不是字符串, 也不是HTML/XML标签
6. 注意2: 它最终产生的就是一个JS对象
7. 标签名任意: HTML标签或其它标签
8. 标签属性任意: HTML标签属性或其它
9. 基本语法规则
10. 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

1. 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
2. babel.js的作用
3. 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
4. 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
4. 参数一: 纯js或jsx创建的虚拟dom对象
5. 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)

React.createElement('h1', {id:'myTitle'}, title)

1. JSX:

<h1 id='myTitle'>{title}</h1>

### JSX练习

需求: 动态展示列表数据



## 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)功能效果的代码集合(html/css/js)
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(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(复杂组件)\*/  class MyComponent2 extends React.*Component* {  render () {  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

|  |
| --- |
| 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()渲染组件标签的基本流程

1. React内部会创建组件实例对象
2. 得到包含的虚拟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) //将props交给父对象

//初始化状态

this.state = {

stateProp1 : value1,

stateProp2 : value2

}

}

1. 读取某个状态值

this.state.statePropertyName

const {statePropertyName} =this.state

1. 更新状态---->组件界面更新

this.setState({

stateProp1 : value1,

stateProp2 : value2

})

注意

1. render()里用不到的state，不应该声明在state里

2）不能直接使用this.state.xxx = xxx的方式来改变一个state的值，应该使用this.setState()

3）状态更新可能是异步的 React可以将多个setState()调用合并成一个调用来提升性能。且由于this.props和this.state可能是异步更新的，所以不应该依靠它们的值来计算下一个状态。这种情况下，可以给setState传入一个函数，如：

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

counter: prevState.counter + props.increment

}));

## 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 美[ˈprɑːpərti]

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

（通过这种参数机制，React 组件可以接受外部消息。）

1. 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.propertyName

1. 对props中的属性值进行类型限制和必要性限制

Person.propTypes = {

name: React.PropTypes.string.isRequired,

age: React.PropTypes.number.isRequired

}

1. 扩展属性: 将对象的所有属性通过props传递

const person={X:x}

<Person {...person}/>

1. 默认属性值

Person.defaultProps = {

name: 'Mary'

}

1. 组件类的构造函数

constructor (props) {

super(props)

console.log(props) // 查看所有属性

}

**props.children** 表示当前组件“包裹”的所有内容。

### 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大属性之三: refs属性

1. 组件内的标签都可以定义ref属性来标识自己
2. <input type="text" **ref={input => this.msgInput = input}**/>
3. 回调函数在组件初始化渲染完或卸载时自动调用
4. 在组件中可以通过this.msgInput来得到对应的真实DOM元素
5. 作用: 通过ref获取组件内容特定标签对象, 进行读取其相关数据

### 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)
2. React使用的是自定义(合成)事件, 而不是使用的原生DOM事件
3. 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)
4. 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

}

### 2.4.4. 强烈注意

1. 函数组件内置的方法中的this为组件对象
2. 类组件自定义的方法中this为null
3. 强制绑定this: 通过函数对象的**bind()**
4. 箭头函数(ES6模块化编码时才能使用) xx=()=>{}

事件处理程序传递参数 我们可以为事件处理程序传递额外的参数，方式有以下两种：

<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>

<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>

需要注意的是，使用箭头函数的情况下，参数e要显式传递，而使用bind的情况下，则无需显式传递（参数e会作为最后一个参数传递给事件处理程序）

## 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(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
4. 动态显示初始化数据
5. 交互功能(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
3. 受控组件: 表单项输入数据能自动收集成状态
4. 非受控组件: 需要时才手动读取表单输入框中的数据（不能直接控制状态的组件）

**受控组件：**

1. 设置value值，value由state控制

2. value值一般在onChange事件中通过setState进行修改

<input

type="text"

value={this.state.value}

onChange={(e) => {

this.setState({

value: e.target.value.toUpperCase(),

});

}}

/>

react中数据是单向流动的，表单的数据来源于组件的state,并通过props传入,这也称为单向数据绑定.然后,我们又通过onChange事件处理器将新的表单数据写回到state,完成了双向数据绑定.

特点:双向绑定

**非受控组件：**

1. 不设置value值

2. 通过ref获取dom节点然后再取value值

## <input type="text" **ref={i => this.name = i}** defaultValue="BeiJing" />

取值方法：this.name.value

特点：任何时候都不需要改变组件的value值

## 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. 生命周期流程:

①**Mount** 挂载， 第一次初始化渲染显示: ReactDOM.render()

constructor(): 创建对象初始化state

加载的时候调用一次，始化state,接受两个参数：props和context

~~componentWillMount()~~ (V16.4移除) UNSAFE\_componentWillMount():

将要插入回调

render() :

用于插入虚拟DOM回调 挂载渲染组件

componentDidMount() :

已经插入回调，组件第一次渲染已完成，此时Dom节点已经生成，可在这调用ajax请求，返回数据setState后组件会重新渲染

②**Update** 更新，每次更新state: this.setSate()

getDerivedStateFromProps() （v16.4新增）

根据当前的props来更新组件的state,每个render都会调用此方法 (触发一些回调，如动画或页面跳转等)

shouldComponentUpdate()

组件接收新的props或state时调用，return true 就会更新dom， return false就能阻止更新，

主要用于性能优化(部分更新)

~~componentWillUpdate()~~ (V16.4移除) UNSAFE\_componentWillUpdate()

getSnapshotBeforeUpdate() （v16.4新增）

在元素被渲染并写入DOM之前调用，使得组件能在发生更改之前从 DOM 中捕获一些信息，此生命周期的任何返回值将作为参数传递给 componentDidUpdate() (想获得聊天窗口中的滚动位置，可以通过这个方法获取信息)

render()

重新挂载(渲染)组件 (创建虚拟dom,进行diff算法，更新dom树都在此进行)

componentDidUpdate()

会在更新(dom已经更新)后会被立即调用，首次渲染不会执行此方法(可以在这里获取dom)

③**Unmount** 卸载，移除组件: ReactDOM.unmountComponentAtNode(containerDom)

componentWillUnmount()

组件将要被移除回调 (取消计时器、网络请求等)

### 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():~~ 已加载组件收到新的参数时调用(V16.4移除)

UNSAFE\_componentWillReceiveProps()

## 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
2. 包含了所有需要的配置
3. 指定好了所有的依赖
4. 可以直接安装/编译/运行一个简单效果
5. react提供了一个用于创建react项目的脚手架库: create-react-app
6. 项目的整体技术架构为: react + webpack + es6 + eslint
7. 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

|--scripts

|-- build.js-------------------build打包引用配置

|-- start.js-------------------start运行引用配置

|--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

# 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
3. 封装XmlHttpRequest对象的ajax
4. promise风格
5. 可以用在浏览器端和node服务器端
6. fetch: 原生函数, 但老版本浏览器不支持
7. 不再使用XmlHttpRequest对象提交ajax请求
8. 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);  })  .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. demo: github users

### 4.4.1. 效果



### 4.4.2. 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

### 4.4.3. 编写静态组件

### 4.4.4. 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可以传递一般数据和函数数据, 只能一层一层传递
3. 一般数据-->父组件传递数据给子组件-->子组件读取数据
4. 函数数据-->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

## 5.2. 事件监听理解

### 5.2.1. 原生DOM事件

1. 绑定事件监听
2. 事件名(类型): 只有有限的几个, 不能随便写
3. 回调函数
4. 触发事件
5. 用户操作界面
6. 事件名(类型)
7. 数据()

### 5.2.2. 自定义事件(消息机制)

1. 绑定事件监听
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
4. 触发事件(编码)
5. 事件名(类型): 与绑定的事件监听的事件名一致
6. 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'
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

# 第6章：react-router4

## 6.1. 相关理解

### 6.1.1. react-router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?
2. 一个路由就是一个映射关系(key:value)
3. key为路由路径, value可能是function/component
4. 路由分类
5. 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

1. 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

1. 后台路由
2. 注册路由: router.get(path, function(req, res))
3. 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

1. 前端路由
2. 注册路由: <Route path="/about" component={About}>
3. 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
2. 网址: <https://github.com/ReactTraining/history>
3. 管理浏览器会话历史(history)的工具库
4. 包装的是原生BOM中window.history和window.location.hash
5. history API
6. History.createBrowserHistory(): 得到封装window.history的管理对象
7. History.createHashHistory():得到封装window.location.hash的管理对象
8. history.push(): 添加一个新的历史记录
9. history.replace(): 用一个新的历史记录替换当前的记录
10. history.goBack(): 回退到上一个历史记录
11. history.goForword(): 前进到下一个历史记录
12. history.listen(function(location){}): 监视历史记录的变化
13. 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**> |

**React-Router库**：

· react-router 核心组件

· react-router-dom 应用于浏览器端的路由库（单独使用包含了react-router的核心部分）

· react-router-native 应用于native端的路由

React Router版本中已不需要路由配置，现在一切皆组件。

ReactRouter中提供了以下三大组件：

## **<Router>**是所有路由组件共用的底层接口组件，它是路由规则制定的**最外层的容器**。

## **<Route>**路由规则匹配，并**显示**当前的规则对应的组件。

## **<Link>**路由**跳转**的组件

三大组件使用的关系：



**Demo**

|  |
| --- |
| import React, { Component } from 'react';  import { HashRouter as Router, Link, Route } from 'react-router-dom';  import './App.css';  const Home = () => (  <div>  <h2>Home</h2>  </div>  )  const About = () => (  <div>  <h2>About</h2>  </div>  )  const Product = () => (  <div>  <h2>Product</h2>  </div>  )  class App extends Component {  render() {  return (  <Router>  <div className="App">  <Link to="/">Home</Link>  <Link to="/About">About</Link>  <Link to="/Product">Product</Link>  <hr/>  <Route path="/" exact component={Home}></Route>  <Route path="/about" component={About}></Route>  <Route path="/product" component={Product}></Route>  </div>  </Router>  );  }  }  export default App; |

## **<[Router>组件](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=router%e7%bb%84%e4%bb%b6)**

Router组件的**history**属性，用来监听浏览器地址栏的变化，并将URL解析成一个地址对象：

**history**的三个属性：①browserHistory ②**has**hHistory ③createMemoryHistory

<BrowserRouter>  浏览器的路由组件

<HashRouter>  URL格式为Hash路由组件

<MemoryRouter>  内存路由组件

<NativeRouter>  Native的路由组件

<StaticRouter> 地址不改变的静态路由组件

**[<BrowserRouter>组件](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=browserrouter%e7%bb%84%e4%bb%b6)**

<BrowserRouter>主要使用在浏览器中，也就是WEB应用中。它利用HTML5 的history API来同步URL和UI的变化。当我们点击了程序中的一个链接之后,<BrowserRouter>就会找出与这个URL匹配的Route，并将他们对应的组件渲染出来。 <BrowserRouter>是用来管理我们的组件的，那么它当然要被放在最顶级的位置，而我们的应用程序的组件就作为它的一个子组件而存在。

|  |
| --- |
| import \* as React from 'react';  import \* as ReactDOM from 'react-dom';  import { BrowserRouter } from 'react-router-dom';  ReactDOM.render(  <BrowserRouter>  <App/>  </BrowserRouter>,  document.body); |

BrowserRouter组件提供了四个属性。

basename: 字符串类型，路由器的默认根路径

forceRefresh: 布尔类型，在导航的过程中整个页面是否刷新

getUserConfirmation: 函数类型，当导航需要确认时执行的函数。默认是：window.confirm

keyLength: 数字类型location.key 的长度。默认是 6

**[basename 属性](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=basename-%e5%b1%9e%e6%80%a7)**

当前位置的基准 URL。如果你的页面部署在服务器的二级（子）目录，你需要将 basename 设置到此子目录。正确的 URL 格式是前面有一个前导斜杠，但不能有尾部斜杠

|  |
| --- |
| <BrowserRouter basename="/admin"/>  ...  <Link to="/home"/> // 被渲染为 <a href="/admin/home">  ...  </BrowserRouter> |

**[getUserConfirmation: func](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=getuserconfirmation-func)**

当导航需要确认时执行的函数。默认使用 [window.confirm](https://developer.mozilla.org/en-US/docs/Web/API/Window/confirm" \t "https://malun666.github.io/aicoder_vip_doc/" \l "/pages/_blank)。

confirm()方法用于显示一个带有指定消息和确认及取消按钮的对话框。

如果访问者点击"确定"，此方法返回true，否则返回false。

// 使用默认的确认函数

const getConfirmation = (message, callback) => {

const allowTransition = window.confirm(message)

callback(allowTransition)

}

<BrowserRouter getUserConfirmation={getConfirmation}/>

**[forceRefresh: bool](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=forcerefresh-bool)**

当设置为 true 时，在导航的过程中整个页面将会刷新。 只有当浏览器不支持 [HTML5 的 history API](http://caniuse.com/" \l "feat=history" \t "https://malun666.github.io/aicoder_vip_doc/" \l "/pages/_blank) 时，才设置为 true。

// **in** 判断下表是否在数组中/判断成员是否在对象中

const supportsHistory = 'pushState' **in** window.history

<BrowserRouter forceRefresh={!supportsHistory}/>

**[children: node](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=children-node)**

## 渲染[单一子组件（元素）](https://facebook.github.io/react/docs/react-api.html" \l "react.children.only" \t "https://malun666.github.io/aicoder_vip_doc/" \l "/pages/_blank)。

**<HashRouter> 组件**

HashRouter使用URL的hash(例如：window.location.hash) 来保持UI和URL的同步。

跟BrowserRouter类似，它也有：basename、getUserConfirmation、children属性，而且是一样的。

**[hashType: string](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=hashtype-string)**

window.location.hash 使用的 hash 类型。有如下几种：

**"slash"**- 后面跟一个斜杠，例如 #/ 和 #/sunshine/lollipops

"noslash" - 后面没有斜杠，例如 # 和 #sunshine/lollipops

"hashbang" - Google风格的["ajax crawlable"](https://developers.google.com/webmasters/ajax-crawling/docs/learn-more" \t "https://malun666.github.io/aicoder_vip_doc/" \l "/pages/_blank)，例如 #!/ 和 #!/sunshine/lollipops

默认为 "slash"。

## [MemoryRouter](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=memoryrouter)组件

## 主要用在ReactNative这种非浏览器的环境中，因此直接将URL的history保存在了内存中。 StaticRouter 主要用于服务端渲染。

## **<[Link>组件](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=link%e7%bb%84%e4%bb%b6)**

## Link就像是一个个的路牌，为我们指明组件的位置。Link使用声明式的方式为应用程序提供导航功能，定义的Link最终会被渲染成一个a标签。Link使用to这个属性来指明目标组件的路径，可以直接使用一个字符串，也可以传入一个对象。

import { Link } from 'react-router-dom'

// 字符串参数

<Link to="/query">查询</Link>

// 对象参数

<Link to={{

pathname: '/query',

search: '?key=name',

hash: '#hash',

state: { fromDashboard: true }

}}>查询</Link>

**[属性： to](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=%e5%b1%9e%e6%80%a7%ef%bc%9a-to)**

需要跳转到的路径(pathname)或地址（location）。

**[属性：replace: bool](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=%e5%b1%9e%e6%80%a7%ef%bc%9areplace-bool)**

当设置为true 时，点击链接后将使用新地址替换掉访问历史记录里面的原地址。

当设置为false 时，点击链接后将在原有访问历史记录的基础上添加一个新的纪录。(默认)

<Link to="/courses" replace />

**[NavLink组件](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=navlink%e7%bb%84%e4%bb%b6)**

NavLink是一个特殊版本的Link，可以使用activeClassName来设置Link被选中时被附加的class，使用activeStyle来配置被选中时应用的样式。此外，还有一个**exact**属性,此属性要求location完全匹配才会附加class和style。这里说的匹配是指地址栏中的URl和这个Link的to指定的location相匹配。

// 选中后被添加class selected

<NavLink to={'/'} exact activeClassName='selected'>Home</NavLink>

// 选中后被附加样式 color:red

<NavLink to={'/gallery'} activeStyle={{color:red}}>Gallery</NavLink>

activeClassName默认值为 active

**[属性](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=%e5%b1%9e%e6%80%a7)**

to 可以是字符串或者对象，同Link组件

exact 布尔类型，完全匹配时才会被附件class和style

activeStyle Object类型

activeClassName 字符串类型

strict: bool类型，当值为 true 时，在确定位置是否与当前 URL 匹配时，将考虑位置 pathname 后的斜线。

**<[Route>组件](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=route%e7%bb%84%e4%bb%b6)**

<Route>应该是react-route中最重要的组件了，它的作用是当location与Route的path匹配时渲染Route中的Component。如果有多个Route匹配，那么这些Route的Component都会被渲染。(显示)

## **exact属性**

## <Route>与<Link>类似，<Route>也有一个**exact**属性，作用也是要求location与Route的path绝对匹配。

// 当location形如 http://location/时，Home就会被渲染。

// 因为 "/" 会匹配所有的URL，所以这里设置一个exact来强制绝对匹配。

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

<Route path="/about" component={About}/>

**[<Route>的三种渲染方式](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=route%e7%9a%84%e4%b8%89%e7%a7%8d%e6%b8%b2%e6%9f%93%e6%96%b9%e5%bc%8f)**

①**component**: 这是最常用也最容易理解的方式，给什么就渲染什么。

②**render**: render的类型是function，Route会渲染这个function的返回值。

因此它的作用就是附加一些额外的逻辑。

③**children**: 这是最特殊的渲染方式。

1. 它同render类似,是一个function。不同的地方在于它会被传入一个**match参数**来告诉你这个Route的path和location匹配上没有。
2. 即使path没有匹配上，我们也可以将它渲染出来。秘诀就在于前面一点提到的match参数。我们可以根据这个参数来决定在匹配的时候渲染什么，不匹配的时候又渲染什么。

// 在匹配时，容器的class是light，<Home />会被渲染

// 在不匹配时，容器的class是dark，<About />会被渲染

<Route path='/home' **children**={({ match }) => (

<div className={match ? 'light' : 'dark'}>

{match ? <Home/>:<About>}

</div>

)}/>

所有路由中指定的组件将被传入以下三个 props :

match

location

history

**match.params.**透过这个属性，我们可以拿到从location中解析出来的参数。当然，如果想要接收参数，我们的Route的path也要使用特殊的写法。

<Link to='/p/1' />

<Link to='/p/2' />

<Link to='/p/3' />

......

<Route path='/p/:id'

render={(match)=<h3>当前文章ID:{match.params.id}</h3>)} />

三个Link是一个文章列表中三个链接，分别指向三篇id不同的文章。而Route用于渲染文章详情页。注意path='/p/:id' ，location中的对应的段会被解析为id=1 这样的键值。最终这个键值会作为param的键值存在。Route中的组件可以使用**this.props.match.params.**id来获取，示例中使用了结构赋值

**[location](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=location)**

location 是指你当前的位置，下一步打算去的位置，或是你之前所在的位置，形式大概就像这样：

{

key: 'ac3df4', // 在使用 hashHistory 时，没有 key

pathname: '/somewhere'

search: '?some=search-string',

hash: '#howdy',

state: {

[userDefined]: true

}

}

你使用以下几种方式来获取 location 对象：

在 [Route component](https://malun666.github.io/aicoder_vip_doc/" \l "/./Route?id=component) 中，以**this.props.location** 的方式获取，

在 [Route render](https://malun666.github.io/aicoder_vip_doc/" \l "/./Route?id=render-func) 中，以 **({ location }) => ()** 的方式获取，

在 [Route children](https://malun666.github.io/aicoder_vip_doc/" \l "/./Route?id=children-func) 中，以 **({ location }) => ()**的方式获取，

在 [withRouter](https://malun666.github.io/aicoder_vip_doc/" \l "/./withRouter) 中，以 **this.props.location** 的方式获取。

你也可以在 **history.location** 中获取 location 对象，但是别那么写，因为 history 是可变的。更多信息请参见 [history 文档](https://malun666.github.io/aicoder_vip_doc/" \l "/./history)。

location 对象不会发生改变，因此你可以在生命周期的钩子函数中使用 location 对象来查看当前页面的位置是否发生改变，这种技巧在获取远程数据以及使用动画时非常有用。

componentWillReceiveProps(nextProps) {

if (nextProps.location !== this.props.location) {

// 已经跳转了！

}

}

通常情况下，你只需要给一个字符串当做 location ，但是，当你需要添加一些 location 的状态时，你可以对象的形式使用 location 。并且当你需要多个 UI ，而这些 UI 取决于历史时，例如弹出框（modal），使用location 对象会有很大帮助。

// 通常你只需要这样使用 location

<Link to="/somewhere"/>

// 但是你同样可以这么用

const location = {

pathname: '/somewhere'

state: { fromDashboard: true }

}

<Link to={location}/>

<Redirect to={location}/>

history.push(location)

history.replace(location)

最后，你可以把 location 传入以下组件：

[Route]

[Switch]

这样做可以让组件不使用路由状态（router state）中的真实 location，因为我们有时候需要组件去渲染一个其他的 location 而不是本身所处的真实 location，比如使用动画或是等待跳转时。

**[history](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=history)**

本文档中的「history」以及「history对象」请参照 [history 包](https://github.com/ReactTraining/history" \t "https://malun666.github.io/aicoder_vip_doc/" \l "/pages/_blank)中的内容。 History 是 React Router 的两大重要依赖之一（除去 React 本身），在不同的 Javascript 环境中，history 以多种形式实现了对于 session 历史的管理。

我们会经常使用以下术语：

「browser history」 - history 在 DOM 上的实现，经常使用于支持 HTML5 history API 的浏览器端。

「hash history」 - history 在 DOM 上的实现，经常使用于旧版本浏览器端。

「memory history」 - 一种存储于内存的 history 实现，经常用于测试或是非 DOM 环境（例如 React Native）。

history 对象通常会具有以下属性和方法：

history.length -（number类型）指的是 history 堆栈的数量。

history.action -（string类型）指的是当前的动作（action），例如 PUSH，REPLACE 以及 POP 。

history.location -(object类型)是指当前的位置（location），location 会具有如下属性：

pathname -(string类型)URL路径。

search -(string类型)URL中的查询字符串（query string）。

hash -(string类型)URL的 hash 分段。

state -(string类型)是指 location 中的状态，

例如在 push(path, state) 时，state会描述什么时候 location 被放置到堆栈中等信息。这个 state 只会出现在 browser history 和 memory history 的环境里。

history.push(path, [state]) -(function 类型)在 hisotry 堆栈顶加入一个新的条目。

history.replace(path, [state]) -(function 类型)替换在history堆栈中的当前条目。

history.go(n)  -(function 类型)将 history 堆栈中的指针向前移动 n 。

history.goBack()  -(function 类型)等同于 go(-1) 。

history.goForward()  -(function 类型)等同于 go(1) 。

history.block(prompt) -(function 类型)阻止跳转，（请参照 [history 文档](https://github.com/ReactTraining/history" \l "blocking-transitions" \t "https://malun666.github.io/aicoder_vip_doc/" \l "/pages/_blank)）

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

**[match](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=match)**

match 对象包含了 <Route path> 如何与URL匹配的信息。match 对象包含以下属性：

**params** -（ object 类型）即路径参数，通过解析URL中动态的部分获得的键值对。

isExact - 当为 true 时，整个URL都需要匹配。

path  -（ string 类型）用来做匹配的路径格式。在需要嵌套 <Route> 的时候用到。

url  -（ string 类型）URL匹配的部分，在需要嵌套 <Link> 的时候会用到。

你可以在以下地方获取 match 对象：

在 Route component 中，以 this.props.match 方式。

在 Route render中，以 ({ match }) => () 方式。

在 Route children中，以 ({ match }) => () 方式

**[Redirect组件](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=redirect%e7%bb%84%e4%bb%b6)**

当这个组件被渲染是，location会被重写为Redirect的to指定的新location。

作用:登录重定向，比如在用户点了登录并验证通过之后，将页面跳转到个人主页。

<Redirect to="/new"/>

**[Switch组件](https://malun666.github.io/aicoder_vip_doc/" \l "/pages/Vip3_react_router?id=switch%e7%bb%84%e4%bb%b6)**

渲染匹配地址(location)的第一个并且作为子元素的 <Route>或者<Redirect>

这与只使用一堆<Route>有什么不同？

<Switch>的独特之处是独它仅仅渲染一个路由。

相反地(不使用<Switch>包裹的情况下)，每一个包含匹配地址(location)的<Route>都会被渲染。

思考下面的代码：

<Route path="/about" component={About}/>

<Route path="/:user" component={User}/>

<Route component={NoMatch}/>

如果现在的URL是 /about ，那么 <About>, <User>, 还有 <NoMatch> 都会被渲染，因为它们都与路径(path)匹配。

import { Switch, Route } from 'react-router'

<Switch>

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

<Route path="/about" component={About}/>

<Route path="/:user" component={User}/>

<Route component={NoMatch}/>

</Switch>

如果我们处于 /about, <Switch> 将开始寻找匹配的 <Route>。

<Route path="/about"/> 将被匹配， <Switch> 将停止寻找匹配并渲染<About>

## 6.2. react-router相关API

### 6.2.1. 组件

1. <BrowserRouter> 浏览器的路由组件 h5后
2. <HashRouter>   URL格式为Hash路由组件
3. <Route>
4. <Redirect>
5. <Link> 路由连接
6. <NavLink> 导航路由连接
7. <Switch> 渲染匹配地址(location)的第一个 <Route>或者<Redirect>

### 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: npm install --save react-router@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** About() {  **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**>  )  } } |

# 第7章：react-ui

## 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/#/](http://www.material-ui.com/" \l "/)
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-mobile使用入门

### 7.2.1. 效果



### 7.2.2. 使用create-react-app创建react应用

npm install create-react-app -g

create-react-app antm-demo

cd antm-demo

npm start

### 7.2.3. 搭建antd-mobile的基本开发环境

1. 下载

npm install antd-mobile --save

1. src/App.jsx

|  |
| --- |
| **import** React, {Component} **from 'react'** *// 分别引入需要使用的组件* **import** Button **from 'antd-mobile/lib/button' import** Toast **from 'antd-mobile/lib/toast'  export default class** App **extends** Component {  *handleClick* = () => {  Toast.*info*(**'提交成功'**, 2)  }   render() {  **return** (  <**div**>  <**Button type="primary" onClick=**{**this**.*handleClick*}>提交</**Button**>  </**div**>  )  } } |

1. src/index.js

|  |
| --- |
| **import** React **from 'react'**; **import** ReactDOM **from 'react-dom' import** App **from "./App"** *// 引入整体css* **import 'antd-mobile/dist/antd-mobile.css'** ReactDOM.render(<**App** />, **document**.getElementById(**'root'**)) |

1. index.html

|  |
| --- |
| <**meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, minimum-scale=1, user-scalable=no"** />  <**script src="https://as.alipayobjects.com/g/component/fastclick/1.0.6/fastclick.js"**></**script**> <**script**>  **if** (**'addEventListener'** in **document**) {  **document**.addEventListener(**'DOMContentLoaded'**, **function**() {  FastClick.attach(**document**.**body**);  }, **false**);  }  **if**(!**window**.**Promise**) {  **document**.writeln(**'<script src="https://as.alipayobjects.com/g/component/es6-promise/3.2.2/es6-promise.min.js"'**+**'>'**+**'<'**+**'/'**+**'script>'**);  } </**script**> |

### 7.2.4. 实现按需打包(组件js/css)

1. 下载依赖包

yarn add react-app-rewired --dev

yarn add babel-plugin-import --dev

1. 修改默认配置:

* package.json

|  |
| --- |
| **"scripts"**: {  **"start"**: **"react-app-rewired start"**,  **"build"**: **"react-app-rewired build"**,  **"test"**: **"react-app-rewired test --env=jsdom"** } |

* config-overrides.js

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

1. 编码

|  |
| --- |
| *// import 'antd-mobile/dist/antd-mobile.css'*  *// import Button from 'antd-mobile/lib/button' // import Toast from 'antd-mobile/lib/toast'* **import** {Button, Toast} **from 'antd-mobile'** |

# 第8章：redux

## 8.1. redux理解

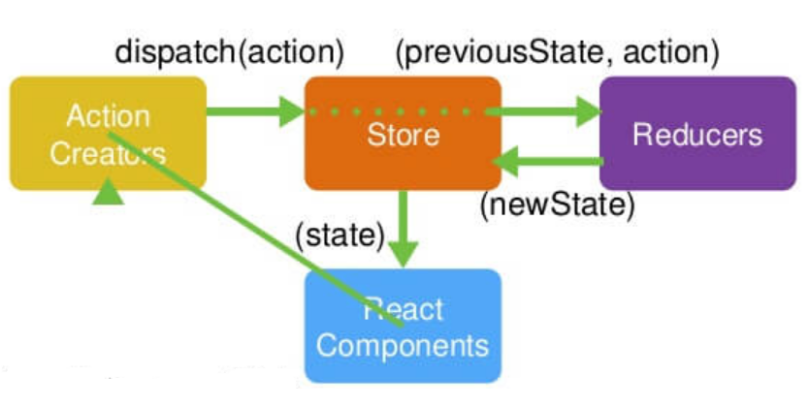
### 8.1.1. 学习文档

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) 分发事件

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)
2. 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
2. 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
3. **type**: 标识属性, 值为字符串, 唯一, 必要属性
4. xxx: 数据属性, 值类型任意, 可选属性
5. 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
2. 返回一个新的状态
3. 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
2. 只负责 UI 的呈现，不带有任何业务逻辑
3. 通过props接收数据(一般数据和函数)
4. 不使用任何 Redux 的 API
5. 一般保存在components文件夹下
6. 容器组件
7. 负责管理数据和业务逻辑，不负责UI的呈现
8. 使用 Redux 的 API
9. 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() connect(state=>state,actions)()

用于包装 UI 组件生成容器组件

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

1. mapStateToprops() state=>state

将 store 中的数据作为 props 绑定到组件上。

  const mapStateToprops = function (state) {   
   return {   
     value: state   
   }   
  }

1. mapDispatchToProps() actions

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，必定得到同样的输出
2. 必须遵守以下一些约束
3. 不得改写参数
4. 不能调用系统 I/O 的API
5. 能调用Date.now()或者Math.random()等不纯的方法
6. 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:
3. 定时器设置函数
4. 数组的map()/filter()/reduce()/find()/bind()
5. react-redux中的connect函数
6. 作用:

能实现更加动态, 更加可扩展的功能

**组件的概念**

**jsx语法**

必须闭合

顶层只能有一个标签

HTML 原生标签使用小写，自定义的组件标签首字母大写

允许js与jsx混写，{}进入js上下文

**组件定义**

继承React.Component基类，重写render方法

标签属性传值：this.props

标签内容传值：this.props.children

组件内部状态：this.state

**组件生命周期**

componentDidMount：组件挂载后自动调用

componentWillUnmount：组件卸载前自动调用

componentDidUpdate：UI 每次更新后调用

“受控”与“非受控”两个概念，区别在于这个组件的状态是否可以被外部修改。一个设计得当的组件应该同时支持“受控”与“非受控”两种形式，即当开发者不控制组件属性时，组件自己管理状态，而当开发者控制组件属性时，组件该由属性控制。

**定义全局 CSS**

CSS Modules 默认是局部作用域的，想要声明一个全局规则，可用 :global 语法。

比如：

.title {

color: red;

}

:global(.title) {

color: green;

}

然后在引用的时候：

<App className={styles.title} /> // red

<App className="title" /> // green

**classnames Package**

在一些复杂的场景中，一个元素可能对应多个 className，而每个 className 又基于一些条件来决定是否出现。这时，[classnames](https://github.com/JedWatson/classnames" \t "https://dvajs.com/knowledgemap/_blank) 这个库就非常有用。

import classnames from 'classnames';const App = (props) => {

const cls = classnames({

btn: true,

btnLarge: props.type === 'submit',

btnSmall: props.type === 'edit',

});

return <div className={ cls } />;

}

这样，传入不同的 type 给 App 组件，就会返回不同的 className 组合：

<App type="submit" /> // btn btnLarge<App type="edit" /> // btn btnSmall

子组件的 state 可以上提 (state hoisting)，由父组件来管理：

子组件间接回调到父组件的 setState 的方法来改变父组件的 state；(子=>父)

新的 state 通过 props 的形式把再次被子组件获悉。(父=>子)

react、redux、react-redux之间的关系

**react**就是一个状态机，用于构建用户界面的。react本身是有状态管理机制的。定义初始状态，根据状态渲染界面，修改状态是通过setState来实现的。也就是单独使用react可以完成一个不大的项目。

**redux**是状态管理器，只是用来管理状态的，它是一个独立的状态管理工具，可以在任何地方使用 ，要完成大的项目就必须要使用redux了。

问题：

创建一个window.store = createStore(reducer)，然后在需要的地方通过store.getState()去获取数据，通过store.dispatch去更新数据，通过store.subscribe去订阅数据变化然后进行setState...如果很多地方都这样做一遍，实在是不堪其重，而且，还是没有避免掉全局变量的不优雅。

**React-Redux**

由于全局变量有诸多的缺点，那就换个思路，把store直接集成到React应用的顶层props里面，只要各个子组件能访问到顶层props就行了

componentDidMount() {

this.props.dispatch({

type: 'cards/queryList',

});

}

**redux简介**

·redux是react全家桶的一员，它试图为 React 应用提供「可预测化的状态管理」机制。

·Redux是将整个应用状态存储到到一个地方，称为store

·里面保存一棵状态树(state tree)

·组件可以派发(dispatch)行为(action)给store,而不是直接通知其它组件

·其它组件可以通过订阅store中的状态(state)来刷新自己的视图

1、安装 ：

npm install --save redux

2、redux核心

① State

state是数据集合

② action

State的变化，会导致View的变化。但是，用户接触不到 State，只能接触到View 所以，State的变化必须是 View导致的。

action就是改变state的指令，有多少操作state的动作就会有多少action。

③reducer 加工函数

action发出命令后将state放入reucer加工函数中，返回新的state。

④store

let store = createStore(reducers);

Store 就是把它们联系到一起的对象。Store 有以下职责：

维持应用的 state；

提供 getState() 方法获取 state；

提供 dispatch(action) 方法更新 state；

通过 subscribe(listener) 注册监听器;

通过 subscribe(listener) 返回的函数注销监听器。

通过store.getState()来了解工厂中商品的状态， 使用store.dispatch发送action指令。

**redux的经典案例**

·定义reducer函数根据action的类型改变state

·actions 定义指令

·通过createStore创建store

·调用store.dispatch()发出修改state的命令

import { createStore } from 'redux'

const reducer = (state = {count: 0}, action) => {

switch (action.type){

case 'INCREASE': return {count: state.count + 1};

case 'DECREASE': return {count: state.count - 1};

default: return state;

}

}

const actions = {

increase: () => ({type: 'INCREASE'}),

decrease: () => ({type: 'DECREASE'})

}

const store = createStore(reducer);

store.subscribe(() =>

console.log(store.getState())

);

store.dispatch(actions.increase()) // {count: 1}

store.dispatch(actions.increase()) // {count: 2}

store.dispatch(actions.increase()) // {count: 3}

可以直接在react component上使用store.dispatch，但是这样不太方便，这个时候我们需要react-redux

class Todos extends Component {

render(){

return(

<div onCLick={()=>store.dispatch(actions.delTodo()) }>test</div>

)

}

}

react-redux

Redux 官方提供的 React 绑定库。 具有高效且灵活的特性。

安装：

npm install --save react-redux

核心

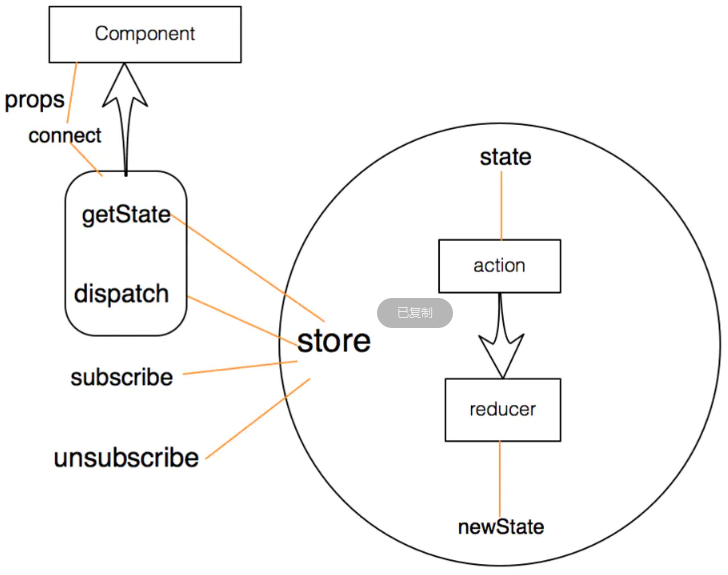
< Provider store>

connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])

Provider 内的任何一个组件（比如这里的 Comp），如果需要使用 state 中的数据，就必须是「被 connect 过的」组件——使用 connect 方法对「你编写的组件（MyComp）」进行包装后的产物。

这个函数允许我们将 store 中的数据作为 props 绑定到组件上。

简单的流程如下图所示：

react-redux中的connect方法将store上的getState 和 dispatch 包装成组件的props。

将之前直接在组件上dispatch的代码修改为如下：

import React, { Component } from 'react';

import {connect} from 'react-redux';

import store from '../store';

import actions from '../store/actions/list';

class Todos extends Component {

render(){

return(

<div onCLick={()=>this.props.del\_todo() }>test</div>

)

}

}

export default connect(

state=>state,

actions

)(Todos);

Provider 能拿到关键的store并传递给每个子组件

**connect如何工作的？**

connect() 接收四个参数，它们分别是 mapStateToProps ， mapDispatchToProps， mergeProps 和 options 。

mapStateToProps这个函数允许我们将 store 中的数据作为 props 绑定到组件上。

**reducer.js**

export default function (

state = { lists: [{text:'移动端计划'}],newType:'all'}, action) {

switch (action.type) {

case types.ADD\_TODO:

return {...state,lists:[...state.lists,{text:action.text}]}

case types.TOGGLE\_TODO:

return {...state,lists:state.lists.map((item,index)=>{

if(index == action.index){

item.completed = !item.completed

}

return item

})}

case types.DEL\_TODO:

return {...state,lists:[...state.lists.slice(0,action.index),...state.lists.slice(action.index+1)]}

case types.SWITCH\_TYPE:

console.log({...state,newType:action.newType})

return {...state,newType:action.newType}

default:

return state;

}

}

在reducer.js中，定义了初始化的state，通过connect方法，我们就能使用this.props.lists拿到初始化的state

import React, { Component } from 'react';

import store from '../store';

import actions from '../store/actions/list';

import {connect} from 'react-redux';

class Todos extends Component {

render(){

return(

{

+ <ul>

+ this.props.state.lists.map(list =>(

+ <li>{list.text}</li>

+ ))

+ </ul>

}

<div onCLick={()=>this.props.del\_todo() }>test</div>

)

}

}

export default connect(

state=>state,

actions

)(Todos);

当 state 变化，或者 ownProps 变化的时候，mapStateToProps 都会被调用，计算出一个新的 stateProps，（在与 ownProps merge 后）更新给 MyComp。

mapDispatchToProps(dispatch, ownProps): dispatchProps connect 的第二个参数是 mapDispatchToProps，它的功能是，将 action 作为 props 绑定到 MyComp 上。

action.js

import \* as types from "../action-types";

export default{

add\_todo(text){

return { type: types.ADD\_TODO, text: text}

},

del\_todo(idx){

return {type:types.DEL\_TODO, index: idx}

},

toggle\_todo(index){

return {type:types.TOGGLE\_TODO, index}

},

del\_todo(index){

return {type:types.DEL\_TODO, index}

},

switch\_type(newType){

return {type:types.SWITCH\_TYPE, newType}

}

}

我在action.js中定义的修改状态的命令，会通过connect 的 mapDispatchToProps方法变为props绑定在react组件上。

我们可以方便得使用去调用

<div onCLick={()=>this.props.del\_todo() }>test</div>

了解到这里，我们会发现并没有使用store.dispatch方法去发出命令，但是state已经修改，view也变化了，那么到底发生了什么？

store.dispatch(actions.increase())

关键的是connect()

connect原理简化版

import React,{Component} from 'react';

import {bindActionCreators} from 'redux';

import propTypes from 'prop-types';

export default function(mapStateToProps,mapDispatchToProps){

return function(WrapedComponent){

class ProxyComponent extends Component{

static contextTypes = {

store:propTypes.object

}

constructor(props,context){

super(props,context);

this.store = context.store;

this.state = mapStateToProps(this.store.getState());

}

componentWillMount(){

this.unsubscribe = this.store.subscribe(()=>{

this.setState(mapStateToProps(this.store.getState()));

});

}

componentWillUnmount(){

this.unsubscribe();

}

render(){

let actions= {};

if(typeof mapDispatchToProps == 'function'){

actions = mapDispatchToProps(this.store.disaptch);

}else if(typeof mapDispatchToProps == 'object'){

console.log('object', mapDispatchToProps)

actions = bindActionCreators(mapDispatchToProps,this.store.dispatch);

}

return <WrapedComponent {...this.state} {...actions}/>

}

}

return ProxyComponent;

}

}

1. state的返回 connect中对于Provided父组件上传来的store,通过将状态返回

mapStateToProps(this.store.getState());

通过 Redux 的辅助函数 bindActionCreators()，用dispatch监听每一个action。

bindActionCreators(mapDispatchToProps,this.store.dispatch);

所以调用props上的方法时，会自动发起store.dispach（XXX）事件，发出命令