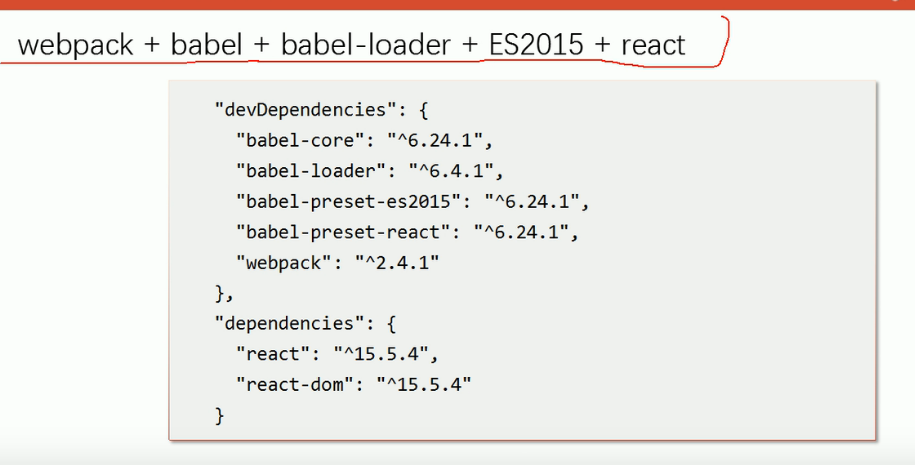
# React

## React环境搭建



### 创建package.json项目身份证：

npm init

### 安装webpack,并且设置为项目依赖:

npm install --save-dev webpack

当然你必须之前已经在-g下安装过了webpack。

### 我们创建一个webpack.config.js文件

这个文件是webpack工作参考 ,我们对着官方文档进行创建

https://webpack.js.org/configuration/

webpack.config.js

|  |
| --- |
| const path = require('path');  module.exports = {  entry: "./app/main.js",    output: {  path: path.resolve(\_\_dirname, "dist"),  filename:"all.js"  }  } |

至此，我们的webpack就已经可以进行标准的CMD模块进行开发了。

### 安装babel-loader

我们参考https://gitthub.com/babel/babel-loader改变web.config.js文件:

|  |
| --- |
| const path = require('path');  module.exports = {  entry: "./app/main.js",  output: {  path: path.resolve(\_\_dirname, "dist"),  filename:"all.js"  },  **module: {**  **rules: [**  **{**  **test: /\.js$/,**  **exclude: /(node\_modules|bower\_components)/,**  **use: {**  **loader: 'babel-loader',**  **options: {**  **presets: ['es2015']**  **}**  **}**  **}**  **]**  **}**  } |

友情提示webpack1和2在配置上差别最大的就是loader的书写方法。module->rules->test/use属性

|  |
| --- |
| npm install --save-dev babel-core  npm install --save-dev babel-loader  npm install --save-dev babel-preset-es2015 |

此时运行webpack命令，将自动的使用CMD构建app文件夹中的main.js文件,并且使用babel翻译

### 安装react

|  |
| --- |
| npm install --save-dev react  npm install --save-dev react-dom |

你可能会问:

①react为什么不是bower来安装,而是npm来安装

②为什么是save-dev而不是save,也就是说为什么是开发依赖,而不是运行依赖

改变webpack.cofig.js

|  |
| --- |
| const path = require('path');  module.exports = {  entry: "./app/main.js",  output: {  path: path.resolve(\_\_dirname, "dist"),  filename:"all.js"  },  module: {  rules: [  {  test: /\.js$/,  exclude: /(node\_modules|bower\_components)/,  use: {  loader: 'babel-loader',  options: {  presets: ['es2015',**'react'**]  }  }  }  ]  }  } |

|  |
| --- |
| npm install --save-dev babel-preset-react |

最终的package.json的代码

|  |
| --- |
| {  "name": "reactjswithleige",  "version": "1.0.0",  "description": "",  "main": "index.js",  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1"  },  "author": "",  "license": "ISC",  "devDependencies": {  "babel-core": "^6.26.0",  "babel-loader": "^7.1.2",  "babel-preset-es2015": "^6.24.1",  "babel-preset-react": "^6.24.1",  "react": "^16.1.1",  "react-dom": "^16.1.1",  "webpack": "^3.8.1"  }  } |

## JSX语法

|  |
| --- |
| **①需要被一个单独的大元素包裹，比如div或者section**  **②标签必须关闭**  **③class要成className, for要写成htmlFor**  **④html注释不能使用,只能使用js注释** |

|  |
| --- |
| JSX中可以使用{}表示临时插入一个js简单表达式,不能使if、for等复杂结构,  可以是&&,||也可以是三元运算符,可以调用函数。{ 3>8 ? 1 :2 } |

|  |
| --- |
| **⑤原生标签比如p、li、div如果要使用自定义属性,必须使用前缀data**  <p data-a=”20”></p>  如果是自定义标签,此时可以随便传属性  <App a=”123456”></App> |

|  |
| --- |
| **⑥JSX表达式用{}单大括号包裹,只能写简单表达式 也可以运行函数**  import React,{Component} from "react";  class App extends Component{  constructor(props){  super(props);  }  **haha(lanmu){**  if(lanmu=="篮球"){  return (  <ul>  <li>科比</li>  <li>姚明</li>  </ul>  )  }else if(lanmu=="足球"){  return (  <ul>  <li>李铁</li>  <li>郝海东</li>  </ul>  )  }  }  render(){  **let xixi= () => {**  return (  <ol>  <li>你好</li>  <li>你好</li>  <li>你好</li>  </ol>  )  }  return (  <div>  <div className="header">header</div>  <div className="body">body</div>  <div className="footer">footer  <h3>JSX语法很好用哦！</h3>  { 3> 8 ? "是":"否" }  <h3>JSX可以调用函数哦</h3>  {this.haha("足球")}  {xixi()}  </div>  </div>  );  }  }  export default App;  一个要用this 一个不需要 体会一下 因为xixi是闭包而haha是类的  函数里面可以写if这些复杂的用法 是不是很好啊 |

|  |
| --- |
| **⑧样式使用双大括号{{}}**  <div style={{"width":"100px","height":"100px","**backgroundColor**":"red"}}></div> |

|  |
| --- |
| **⑨可以使用数组，数组中如果是JSX语法,数组会被自动展开,所以不需要使用ng-repeat指令去展开数组**  **定义一个数组,定义的JSX项目上要求有key属性,只要是重复的数组项目，都要key属性这是要求。**  **let arr=["白板","幺鸡","二条","发财"].map((item,index)=>{**  **return <li key={index} >{item}</li>**  **});**  return (  <div  <ul>  **{arr}**  </ul>  </div>  ); |

参考

|  |
| --- |
| import React,{Component} from "react";  class App extends Component{  constructor(props){  super(props);  }  haha(lanmu){  if(lanmu=="篮球"){  return (  <ul>  <li>科比</li>  <li>姚明</li>  </ul>  )  }else if(lanmu=="足球"){  return (  <ul>  <li>李铁</li>  <li>郝海东</li>  </ul>  )  }  }  render(){  let xixi= () => {  return (  <ol>  <li>你好</li>  <li>你好</li>  <li>你好</li>  </ol>  )  };  let sobj={  "float":"left"  };  let arr=["白板","幺鸡","二条","发财"].map((item,index)=>{  return <li key={index} >{item}</li>  });  return (  <div>  <div className="header">header</div>  <div className="body">body</div>  <div className="footer">footer  <h3>JSX语法很好用哦！</h3>  { 3> 8 ? "是":"否" }  <h3>JSX可以调用函数哦</h3>  {this.haha("足球")}  {xixi()}  <h3>JSX可以设置样式哦</h3>  <div style={{"width":"100px","height":"100px","backgroundColor":"red"}}></div>  <ul>  {arr}  </ul>  </div>  </div>  );  }  }  export default App; |

## React中数据传递

React 中跟数据有关系的就三个东西:state、props、context。数据传递三兄弟。

### ①组建自己身上的属性变化，不会引起视图改变

我们可以在组件的类的定义里面写constructor构造器,里面定义this.a=100,表示给组件的实例绑定一个a属性,值为100。在JSX

|  |
| --- |
| import React,{Component} from "react";  class App extends Component{  //构造函数  constructor(){  **super();//要求调用super**  **this.a=100;**  }  render(){  return (  <div>  <div className="header">header</div>  <div className="body">body</div>  <div className="footer">footer **{this.a}** </div>  </div>  );  }  }  export default App; |

我们增加一个按钮,试着让用户点击按钮,a的值变化

|  |
| --- |
| import React,{Component} from "react";  class App extends Component{  //构造函数  constructor(props){  super(props);  this.a=100;  }  **add(){**  **console.log("我是add");**  **this.a++;**  **console.log(this.a);**  **}**  render(){  return (  <div>  <div className="header">header</div>  <div className="body">body</div>  <div className="footer">footer  <p>{this.a}</p>  <p>  <input type="button" value="按我" **onClick={(this.add).bind(this)}** />  </p>  </div>  </div>  );  }  }  export default App; |

通过案例学习了:

|  |
| --- |
| ①绑定监听使用onClick、onMousedown,onMouseenter,onBlur把on后面的字母大写,React可以自动识辨为React事件。  ②绑定函数的时候,this上下文是有问题的,所以需要使用bind()的方法来设置上下文  ③绑定监听函数的时候,注意用{}而不是双引号””  <input type="button" value="按我" **onClick={(this.add).bind(this)}** /> |



|  |
| --- |
| 所以React中,组建自己的属性的变化不会引发视图的变化！！！ |

### ②闭包中的值，不会引起视图改变

|  |
| --- |
| import React,{Component} from "react";  class App extends Component{  //构造函数  constructor(props){  super(props);  this.a=100;  }  add(){  console.log("我是add");  this.a++;  console.log(this.a);  }  render(){  return (  **let b=99;**  <div>  <div className="footer">footer  <p>{b}</p>  <p><input type="button" value="按我" **onClick={()=>{b++;console.log(b)}}** /></p>    </div>  </div>  );  }  }  export default App; |
| 闭包中的值的变化,不会引起视图的改变！！！ |

### ③三兄弟之一:state

React把组件看成是一个状态机(State Machines)。通过与用户的交互,实现不同状态，然后渲染UI,让用户和数据保存一致。

**React里只需更新组件的state,然后根据新的state重新渲染用户界面（不需要DOM操作）**

|  |
| --- |
| 'use strict'  import React,{Component} from "react";  class App extends Component{  //构造函数  constructor(){  super();  **this.state={**  **a:100,**  **b:200,**  **c:300**  **};**  }  add(){  this.setState({a:this.state.a + 1});  }  render(){  return (  <div>  <div className="header">header</div>  <div className="body">body</div>  <div className="footer">footer  <p>我有状态state</p>  <p>a: {this.state.a}</p>  <p>b: {this.state.b}</p>  <p>c: {this.state.c}</p>  <p><input type="button" value="按我" onClick=**{(this.add).bind(this)}** /></p>  </div>  </div>  );  }  }  export default App; |

|  |
| --- |
| 只有更新三兄弟,才会引发Vitual DOM的改变,从而改变DOM |

使用state

|  |
| --- |
| 定义state,在构造函数中使用this.state属性即可  使用state,在JSX中{this.state.a}  改变state,this.setState({a:this.state.a + 1})//不能写++,因为state属性值只读 |

|  |
| --- |
| state是内部的(所有也叫local state),只有组件自己能改变自己的state,别人想改变自己的state都不可能! |

后面要介绍Redux架构,所有组件自己的state越来越少用了,而变为了Redux中的state，不要混肴。

### ④三兄弟之二：props

|  |
| --- |
| 就是定义在自定义组件标签上面的值,就是props。当props改变的时候,会引发Virtual DOM、的改变,从而引发视图的重绘。 |

|  |
| --- |
| 如果父组件App,想往子组件Header中传值,使用属性  App.jsx：  'use strict'  import React,{Component} from "react";  import Header from "./Header.jsx"  class App extends Component{  //构造函数  constructor(props){  super(props);  }  render(){  return (  <div>  **<Header title="最青春" content="管理平台" version="v2.0" ></Header>**  <div className="body">body</div>  <div className="footer">footer</div>  </div>  );  }  }  export default App;  Header.jsx：**子组件可以无脑使用this.props来枚举传入的属性**  'use strict'  import React,{Component} from 'react';  class Header extends Component{  constructor(){  super();  }  render(){  return (  <h3>**{this.props.title}{this.props.content}{this.props.version}**</h3>  )  }  }  export default Header; |

如果需要在构造函数中使用这个值,此时系统会将props作为构造函数的第一个参数传入：

|  |
| --- |
| 'use strict'  import React,{Component} from 'react';  class Header extends Component{  constructor(**props**,content){  super(props);  this.state = {  version : props.version,  nextPublishVersion:**props.nextPublishVesion**  }  }  render(){  return (  <div>  <h3>{this.props.title}{this.props.content}{this.state.version}</h3>  <h3>we will publish {this.state.nextPublishVersion}  <input type="button" value="查看下一个版本" onClick={()=>{this.setState({nextPublishVersion:this.state.nextPublishVersion+1})}} />  </h3>  </div>  )  }  }  export default Header; |

|  |
| --- |
| 在此组件中,props是只读的,不能修改props的值,如果要修改,用state来接受 |

|  |
| --- |
| props属性可以被验证有效性  安装 **npm install --save-dev prop-types**  'use strict'  import React,{Component} from 'react'  import { PropTypes } from 'prop-types'  class Header extends Component{  constructor(props,content){  super(props);  this.state = {  version : props.version,  nextPublishVersion:props.nextPublishVesion,  abc:props.abc  }  }  render(){  return (  <div>  <h3>{this.props.title}{this.props.content}{this.state.version}</h3>  <h3>we will publish {this.state.nextPublishVersion}  <input type="button" value="查看下一个版本" onClick={()=>{this.setState({nextPublishVersion:this.state.nextPublishVersion+1})}} />  </h3>  <h3>  {this.state.abc}  </h3>  </div>  )  }  }  //定义组件需要传入的参数  //类名.propTypes,值是一个JSON,key就是需要传进来的props的属性名v就是对它的限制  **Header.propTypes={**  **title:PropTypes.string.isRequired, //title属性是一个字符串,必传**  **content:PropTypes.string,**  **nextPublishVesion:PropTypes.number.isRequired**  **}**  export default Header; |

如果非要从下往上传输数据呢？子组件要把数据返回给父组件,此时只能用奇淫巧技,

就是：

父组件传一个函数给子组件,**子组件通过传参数调用函数将数据返回给父组件的函数**,父组件的函数接受实参改变父组件state等值

父组件

|  |
| --- |
| 'use strict'  import React,{Component} from "react";  import Header from "./Header.jsx"  class App extends Component{  //构造函数  constructor(){  super();  this.state={  d : 16  }  }  setD(number){  this.setState({"d":number});  }  render(){  return (  <div>  <Header **d={this.state.d}** setD={**(this.setD).bind(this)**} title="最青春" content="管理平台" version="v2.0" nextPublishVesion={3} ></Header>  <div className="body">body {this.state.d}</div>  <div className="footer">footer</div>  </div>  );  }  }  export default App; |

子组件

|  |
| --- |
| 'use strict'  import React,{Component} from 'react'  import { PropTypes } from 'prop-types'  class Header extends Component{  constructor(props,content){  super(props);  this.state = {  version : props.version,  nextPublishVersion:props.nextPublishVesion,  }  }  render(){  return (  <div>  <h3>{this.props.title}{this.props.content}{this.state.version}</h3>  <h3>we will publish {this.state.nextPublishVersion}  <input type="button" value="查看下一个版本" onClick={()=>{this.setState({nextPublishVersion:this.state.nextPublishVersion+1})}} />  </h3>  <h3>  {this.props.d}  <input type="button" value="改变父组件的d值" onClick={()=>{**this.props.setD(8)**}} />  </h3>  </div>  )  }  }  Header.propTypes={  title:PropTypes.string.isRequired,  content:PropTypes.string.isRequired,  nextPublishVesion:PropTypes.number.isRequired  }  export default Header; |

非常麻烦的,没事儿！明天学习Redux,这个事情将更麻烦

### ⑤三兄弟之：context

上下文的精髓是可以跨级传递数据,爷爷组件可以直接传递数据到孙子组件

语法比较晦涩

**Yeye.jsx**

|  |
| --- |
| "use strict"  import React from "react"  import Baba from "./Baba.jsx"  import { PropTypes } from "prop-types"  class Yeye extends React.Component{  constructor(){  super();  this.state = {  a : 200  };  }  render(){  return (  <div>  <h1>yeye</h1>  <Baba></Baba>  </div>  );  }  **//得到孩子上下文,实际上这里表示一种设置,**  **//返回一个对象,这个对象就是现在这个家族体现共享的上下文**  **//将上下文的a值变为自己状态的a**  **getChildContext(){**  **return {**  **a : this.state.a**  **}**  **}**  }  **//设置child的上下文的类型**  **Yeye.childContextTypes = {**  **a : PropTypes.number.isRequired**  **}**  export default Yeye; |

**Baba.jsx**

|  |
| --- |
| "use strict"  import React from "react"  import Sunzi from "./Sunzi.jsx"  **import { PropTypes } from "prop-types"**  class Baba extends React.Component{  render(){  return (  <div>  <h1>baba</h1>  <Sunzi></Sunzi>  </div>  );  }  }  export default Baba; |

**Sunzi.jsx**

|  |
| --- |
| "use strict"  import React from "react"  **import PropTypes from "prop-types"**  class Sunzi extends React.Component{  constructor(props,**context**){  super();  **console.log(context);//得到上下文**  }  render(){  return (  <div>  <h1>sunzi</h1>  <p>  **{this.context.a}**  </p>  </div>  );  }  }  **//设置上下文的类型**  **Sunzi.contextTypes={**  **a : PropTypes.number**  **}**  export default Sunzi; |

结论

①当祖先元素中更改了上下文的数据,此时所有的子孙元素中的数据都会更改,视图也会更新。

②反之 不成立,可以认为上下文的数据在子孙元素中是只读的。此时又需要使用奇淫技巧,就是在context中共享一个操作祖先元素的的函数,子孙元素通过上下文获取到这个函数,从而操作祖先元素的值。

也就是说,state是自治的不涉及传值得事儿,props是单向的,父亲->儿子;context也是单向的,如果要反向,就要传入一个函数。

context 很少用,传值基本用props。除非特别深的跨级别传值,可以用context。

## 其他API

### ①无状态组件

当我们现在的组件仅仅是为了呈递一些DOM元素,没有state,props等东西,此时可以不用肺经class My extends React.Component{}

而是

|  |
| --- |
| import React from "react"  export default () => {  return (  <div>  <h1>我是My组件</h1>  </div>  )  }; |

### ②默认属性

|  |
| --- |
| constructor(props){  super();  this.state = {  companyName : props.companyName || "杭州市海蒂崽崽网络科技有限公司",  };  } |

### ③组件的声明周期

组件从没有诞生到诞生

到就绪上树

到被移除

挂载: 组件被插到DOM中

更新: 组建被重新渲染,查明DOM是否应该刷新

移除: 组建从DOM中移除

Will 在某个行为之前调用

Did 在某个行文之后调用

|  |
| --- |
| import React,{Component} from 'react';  class Footer extends Component{  constructor(props){  super();  this.state = {  companyName : props.companyName || "杭州市海蒂崽崽网络科技有限公司",  a : 1  };  }  **//在挂载之前立即被调用**  **componentWillMount(){**  console.log("componentWillMount")  }  **//在挂载之后立即被调用。需要DOM节点的初始化操作应该放在这里**  **componentDidMount(){**  console.log("componentDidMount")  }  //在接受一个新的props的时候被调用  //this.props 和 nextProps  //this.setState来改变  componentWillReceiveProps(){  console.log("componentWillReceiveProps")  }  **//当组件做出是否要更新DOM的决定的时候被调用**。门神 有把们都功能  //在改变状态的时候可以选择通过或者不通过  **shouldComponentUpdate(nextProps,nextState){**  // console.log("showComponentUpdate",nextProps,nextState);  if(nextState.a > 0.8){  return true;  }  alert("本次没有随机到0.8的,更新被阻止了");  return false;  }  **//在更新发生之前被调用,没有把们都功能**  **componentWillUpdate(){**  console.log("componentWillUpdate");  }  **//当组件移除和销毁之前被调用**  **componentWillUnmount(){**  console.log("componentWillUnmount");  }  render(){  return (  <div>  <h3>  {this.state.companyName}  {this.state.a}  <input type="button" value="按我" onClick={()=>{this.setState({"a":Math.random()})}} />  </h3>  </div>  );  }  }  export default Footer; |

### ④react实现双向绑定

|  |
| --- |
| import React,{Component} from 'react';  class Footer extends Component{  constructor(props){  super();  this.state = {  companyName : props.companyName || "杭州市海蒂崽崽网络科技有限公司",  txt:""  };  **this.change = (this.change).bind(this);**  }  **change(event){**  **console.log(event.target.value);**  **this.setState({"txt":event.target.value});**  **}**  render(){  return (  <div>  <h3>  {this.state.companyName}  **<input type="text" onInput={this.change}/>**  ***<p>{this.state.txt}</p>***  </h3>  </div>  );  }  }  export default Footer; |

### ⑤ref DOM 钩子

在没有办法实在要操作一个DOM的时候用ref

|  |
| --- |
| import React,{Component} from 'react';  class Footer extends Component{  constructor(props){  super();  this.state = {  companyName : props.companyName || "杭州市海蒂崽崽网络科技有限公司",  };  }  render(){  return (  <div>  <h3>  {this.state.companyName}  <input type="button" value="按我让盒子变色" onClick={  ()=>{**this.refs.box.**style.backgroundColor="orange"}  } />  <div **ref="box"** className="box" style={{  "width":"200px",  "height":"200px",  "backgroundColor":"red"  }}  >  </div>  </h3>  </div>  );  }  }  export default Footer; |

## 小例子

|  |
| --- |
| npm install --save-dev css-loader  npm install --save-dev style-loader  npm install --save-dev less-loader |

|  |
| --- |
| const path = require('path');  module.exports = {  entry: "./app/main.js",  output: {  path: path.resolve(\_\_dirname, "dist"),  filename:"all.js"  },  module: {  rules: [  {  test: /\.js[x]?$/,  exclude: /(node\_modules|bower\_components)/,  use: {  loader: 'babel-loader',  options: {  presets: ['es2015','react']  }  }  },  **{**  **test:/\.css$/,**  **use:[**  **{loader:"style-loader"},**  **{loader:"css-loader"},**  **{loader:"less-loader"}**  **]**  **}**  ]  },  watch:true  } |

# Redux

## 安装 npm install redux --save

## 正确的使用方法:

|  |
| --- |
| 1.首先通过reducer新建store，随时通过store.getState获取状态  2.需要状态变更，store.dispath(action)来修改状态  3.Reducer函数接受state和action,返回新的state，可以用store.subscribe监听每次修改 |

|  |
| --- |
| import { createStore } from 'redux'  //这就是reducer处理函数,参数是状态和新的action  function counter(state=0,action){  switch (action.type){  case '加机关枪':  return state + 1;  case '减机关枪':  return state - 1;  default:  return 10;  }  }  //新建保险箱  const store = createStore(counter)  //查看了开始的状态  const init = store.getState()  console.log(`开始有机器$(init)把`)  //使用一个监听  function listener(){  const current = state.getState()  console.log(`现在有几枪${current}把`)  }  //订阅 ,每次state修改，都会执行listener  store.subscribe(listener);  //提交状态变更的申请  store.dispatch({type:'加机关枪'})  store.dispatch({type:'加机关枪'})  store.dispatch({type:'加机关枪'})  store.dispatch({type:'减机关枪'})  store.dispatch({type:'减机关枪'}) |

# Redux如何和React一起用

## React Redux手动方式链接在一起

|  |
| --- |
| 1. 把store.dispatch方法传递给组件,内部可以调用修改状态 2. subscribe订阅render函数,每次修改都重新渲染 3. Redux 相关内容,移到单独的文件index.redux.js单独管理 |

新建一个index.redux.js

|  |
| --- |
| **const ADD\_GUN = '加机关枪'**  **const REMOVE\_GUN = '减机关枪'**  //reducer  export function **counter**(state=0,action){  switch(action.type){  case ADD\_GUN:  return state+1;  case REMOVE\_GUN:  return state-1;  default:  return 10  }  }  //action creator  export function **addGUN**(){  return {type:ADD\_GUN}  }  //action creator  export function **removeGUN**(){  return {type:REMOVE\_GUN}  } |

新建一个App.js

|  |
| --- |
| import React from 'react'  class App extends React.Component{  render(){  return (  <h1>现在有机枪</h1>  )  }  }  export default App |

新建index.js 创建store 将store以属性传递给App组件

|  |
| --- |
| import React from 'react'  import ReactDOM from 'react-dom'  **import { createStore } from 'redux'**  import App from './App'  **import { counter } from './index.redux'**  **const store = createStore(counter)**  function render(){  ReactDOM.render(  <App **store={store}** />,  document.getElemetById('root')  )  }  render()  **store.subscribe(render)** |

在App.js中来使用store

|  |
| --- |
| import React from 'react'  class App extends React.Component{  render(){  **const store = this.props.store**  **const num = store.getState()**  return (  <h1>现在有机枪**{num}**把</h1>  )  }  }  export default App |

在App.js中需要加一下复杂的功能

|  |
| --- |
| import React from 'react'  **import {addGUN} from './index.redux'**  class App extends React.Component{  render(){  const store = this.props.store  const num = store.getState()  return (  <div>  <h1>现在有机枪{num}把</h1>  <button onClick={(**)=>store.dispatch(addGUN())**}>申请武器</button>  </div>  )  }  }  export default App |

这里可以发现组件里面引用了组件，就有问题进行解耦

在index.js

|  |
| --- |
| import React from 'react'  import ReactDOM from 'react-dom'  import { createStore } from 'redux'  import App from './App'  import { counter , **addGUN** } from './index.redux'  const store = createStore(counter)  function render(){  ReactDOM.render(  <App store={store} **addGUN={addGUN}** />,  document.getElemetById('root')  )  }  render()  store.subscribe(render) |

修改App.js

|  |
| --- |
| import React from 'react'  class App extends React.Component{  render(){  const store = this.props.store  const num = store.getState()  **const adGUN = this.props.addGUN**  return (  <div>  <h1>现在有机枪{num}把</h1>  <button onClick={()=>store.dispatch(addGUN())}>申请武器</button>  </div>  )  }  }  export default App |

以上是同步的那么很多情况是异步的？

## Redux处理异步

|  |
| --- |
| redux处理异步,需要redux-thunk插件  npm install redux-thunk --save  redux默认只处理同步，异步任务需要react-thunk中间件  使用applyMiddleware 开启thunk中间件  开启中间件之后，我们的action（之前是返回对象）可以返回函数,使用dispatch提交action  react-thunk主要是修改了action写的方法 |

在index.js

|  |
| --- |
| import React from 'react'  import ReactDOM from 'react-dom'  import { createStore , **applyMiddleware** } from 'redux'  **import thunk from 'redux-thunk'**  import App from './App'  import { counter , addGUN } from './index.redux'  const store = createStore(counter,applyMiddleware(thunk))  function render(){  ReactDOM.render(  <App store={store} addGUN={addGUN} />,  document.getElemetById('root')  )  }  render()  store.subscribe(render) |

在index.redux.js

|  |
| --- |
| const ADD\_GUN = '加机关枪'  const REMOVE\_GUN = '减机关枪'  //reducer  export function counter(state=0,action){  switch(action.type){  case ADD\_GUN:  return state+1;  case REMOVE\_GUN:  return state-1;  default:  return 10  }  }  //action creator  export function addGUN(){  return {type:ADD\_GUN}  }  //action creator  export function removeGUN(){  return {type:REMOVE\_GUN}  }  **//thunk 主要是修改action creator写的方式**  **export function addGUNAsync(){**  **return dispatch=>{**  **setTimeout(()=>{**  **dispatch(addGUN())**  **},2000);**  **}**  **}** |

在App.js

|  |
| --- |
| class App extends React.Component{  render(){  const store = this.props.store  const num = store.getState()  const addGUN = this.props.addGUN  const removeGUN = this.props.removeGUN  **const addGUNAsync = this.props.addGUNAsync**  return (  <div>  <h1>现在有机枪{num}把</h1>  <button onClick={()=>store.dispatch(addGUN())}>申请武器</button>  <button onClick={()=>store.dispatch(removeGUN())}>申请武器</button>  <button **onClick={()=>store.dispatch(addGUNAsync())}**>托两天提交武器</button>  </div>  )  }  }  export default App |

在index.js

|  |
| --- |
| import React from 'react'  import ReactDOM from 'react-dom'  import { createStore , applyMiddleware } from 'redux'  import thunk from 'redux-thunk'  import App from './App'  import { counter , addGUN, removeGUN, **addGUNAsync** } from './index.redux'  const store = createStore(counter,applyMiddleware(thunk))  function render(){  ReactDOM.render(  <App store={store} addGUN={addGUN} removeGUN={removeGUN}  **addGUNAsync={addGUNAsync}**  />,  document.getElemetById('root')  )  }  render()  store.subscribe(render) |

以上方式是手动连接时很痛苦的过程

## react-redux 自动连接react redux

|  |
| --- |
| npm install react-redux --save  此时忘记subscribe  记住reducer,action和dispatch即可  React-redux提供Provider和connect两个接口链接(替代了subscribe) |

具体使用

|  |
| --- |
| Provider组件在应用最外层,传入store即可,只用一次  Connect负责从外部获取组件需要的参数  Connect可以用装饰器的方式来写 |

在index.js

|  |
| --- |
| import React from 'react'  import ReactDom from 'react-dom'  import { createStore, applyMiddleware, compose } from 'redux'  import thunk from 'redux-thunk'  **import { Provider } from 'react-redux'**  import { counter } from './index.redux'  const store = createStore(counter,compose(  applyMiddleware(thunk),  window.devToolsExtension?window.devToolsExtension(): ()=>{}  ))  ReactDom.render(  (**<Provider store={store}>**  **<App />**  **</Provider>),**  document.getElementById('root')  ) |

在App.js

|  |
| --- |
| import React from 'react'  **import { connect } from 'react-redux'**  **import { addGUN, removeGUN, addGUNAsync } from './index.redux'**  class App extends React.Component{  render(){  return (  <div>  <h1>现在有机枪{this.props.num}吧 </h1>  <button **onClick={this.props.addGUN}**>申请武器</button>  <button **onClick={this.props.removeGUN}**>上交武器</button>  <button **onClick={this.props.addGUNAsync**}>托两天再给</button>  </div>  )  }  }  **const mapStatetoProps = (state)=>{**  **return {num:state}**  **}**  **const actionCreators={ addGUN, removeGUN, addGUNAsync }**  **App = connect(mapStatetoProps,actionCreators)(App)**  export default App |

## connect可以用装饰器来写

|  |
| --- |
| npm run eject 弹出配置  npm install babel-plugin-transform-decorators-legacy --save-dev 安装connect装饰器支持插件  Package.json里babel加上plugins配置 |

|  |
| --- |
|  |

在App.js

|  |
| --- |
| import React from 'react'  import { connect } from 'react-redux'  import { addGUN, removeGUN, addGUNAsync } from './index.redux'  **@connect(**  **//你要state什么属性放到props里**  **state=>({num:state}),**  **//你要什么方法,放到props里,自动dispatch**  **{ addGUN, removeGUN, addGUNAsync }**  **)**  class App extends React.Component{  render(){  return (  <div>  <h1>现在有机枪{this.props.num}吧 </h1>  <button onClick={this.props.addGUN}>申请武器</button>  <button onClick={this.props.removeGUN}>上交武器</button>  <button onClick={this.props.addGUNAsync}>托两天再给</button>  </div>  )  }  }  export default App |

什么数据应该放在React

Redux管理ajax

Redux管理聊天数据

# React-router4路由

|  |
| --- |
| 浏览器和RN均兼容  React开发单页面应用必备,践行路由即组件  动态路由，Route Link Switch |

|  |
| --- |
| npm install react-router-dom --save  router4使用react-router-dom作为浏览器的路由  忘记router2的内容，拥抱router4 |

入门组件

|  |
| --- |
| BrowserRouter,包裹整个应用（当然也有hashRouter）  Route路由对应渲染组件,可嵌套  Link 跳转专用 |

|  |
| --- |
| 其他组件  url参数，Router组件参数可用冒号标识参数  Redirect组件,跳转  Switch只渲染第一次匹配到的Route组件 |

|  |
| --- |
| <Link to=’/’> 跳转对应 route  <Route path=’/’ exact component={App}></Route>  exact完全匹配 这样避免组件重复出现  Switch 这样避免组件重复出现 |

## 和redux配合

|  |
| --- |
| 复杂redux应用,多个reducer，用combineReducers合并  Redirect组件跳转  Switch只渲染一个子Route组件 |