**React全家桶(技术栈)**

第1章：React入门

## React简介

官网

1. 英文官网:<https://reactjs.org/>
2. 中文官网: https://react.docschina.org/

介绍描述

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

React的特点

1. 声明式编码
2. 组件化编码
3. React Native 编写原生应用
4. 高效（优秀的Diffing算法）

React高效的原因

1. 使用虚拟(virtual)DOM, 不总是直接操作页面真实DOM。
2. DOM Diffing算法, 最小化页面重绘。

## React的基本使用

### 相关js库

1. react.js：React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JS代码的库。

### 创建虚拟DOM的两种方式

以下两种方式等效

1. 纯JS方式(一般不用)

*const* VDOM **=** React.createElement('h1',{id:'title'},React.createElement('span',{},'Hello,React'))

1. JSX方式

*const* VDOM **=** (  /\* 此处一定不要写引号，因为不是字符串 \*/

      <**h1** id**=**"title">

        <**span**>Hello,React</**span**>

      </**h1**>

    )

### 虚拟DOM与真实DOM

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

## React JSX

### JSX

1. 全称: JavaScript XML
2. react定义的一种类似于XML的JS扩展语法: JS + XML本质是**React.createElement(component, props, ...children)**方法的语法糖
3. 作用: 用来简化创建虚拟DOM
   1. 写法：**var ele = <h1>Hello JSX!</h1>**
   2. 注意1：它不是字符串, 也不是HTML/XML标签
   3. 注意2：它最终产生的就是一个JS对象
4. 标签名任意: HTML标签或其它标签
5. 标签属性任意: HTML标签属性或其它
6. 基本语法规则
   1. 遇到 <开头的代码, 以标签的语法解析: html同名标签转换为html同名元素, 其它标签需要特别解析
   2. 遇到以 { 开头的代码，以JS语法解析: 标签中的js表达式必须用{ }包含

        jsx语法规则：

            1.定义虚拟DOM时，不要写引号。

            2.标签中混入**JS表达式**时要用{}。

            3.样式的类名指定不要用class，要用className。

            4.内联样式，要用style={{key:value}}的形式去写。

            5.只有一个根标签

            6.标签必须闭合

            7.标签首字母

                (1).若小写字母开头，则将该标签转为html中同名元素，若html中无该标签对应的同名元素，则报错。

                (2).若大写字母开头，react就去渲染对应的组件，若组件没有定义，则报错。

js语句（代码）与js表达式的差别

怎么判断是不是表达式，用一个let a = 在左边接，能接到值就是表达式

          1.表达式：一个表达式会产生一个值，可以放在任何一个需要值的地方

                下面这些都是表达式：

                    (1). a

                    (2). a+b

                    (3). demo(1)

                    (4). arr.map()

                    (5). function test () {}

          2.语句(代码)：

                下面这些都是语句(代码)：

                    (1).if(){}

                    (2).for(){}

                    (3).switch(){case:xxxx}

1. babel.js的作用
   1. 浏览器不能直接解析JSX代码, 需要babel转译为纯JS的代码才能运行
   2. 只要用了JSX，都要加上type="text/babel", 声明需要babel来处理

### 渲染虚拟DOM(元素)

        关于虚拟DOM：

          1.本质是Object类型的对象（一般对象）

          2.虚拟DOM比较“轻”，真实DOM比较“重”，因为虚拟DOM是React内部在用，无需真实DOM上那么多的属性。

          3.虚拟DOM最终会被React转化为真实DOM，呈现在页面上。

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

## 模块与组件、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. 为什么要拆成模块：随着业务逻辑增加，代码越来越多且复杂。
3. 作用：复用js, 简化js的编写, 提高js运行效率

### 组件

1. 理解：用来实现局部功能效果的代码和资源的集合(html/css/js/image等等)
2. 为什么要用组件： 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. 使用React开发者工具调试

### 2.1.2. 效果

函数式组件：

    //1.创建函数式组件

*function* MyComponent(){

**return** <**h2**>我是用函数定义的组件(适用于【简单组件】的定义)</**h2**>

    }

    //2.渲染组件到页面

    ReactDOM.render(<*MyComponent*/>,document.getElementById('test'))

    /\*

      执行了ReactDOM.render(<MyComponent/>.......之后，发生了什么？

          1.React解析组件标签，找到了MyComponent组件。

          2.发现组件是使用函数定义的，随后调用该函数，将返回的虚拟DOM转为真实DOM，随后呈现在页面中。

    \*/

类式组件：

    //1.创建类式组件

*class* MyComponent **extends** React.*Component* {

      render(){

        //render是放在哪里的？—— MyComponent的原型对象上，供实例使用。

        //render中的this是谁？—— MyComponent的实例对象 <=> MyComponent组件实例对象。

        console.log('render中的this:',this);

**return** <**h2**>我是用类定义的组件(适用于【复杂组件】的定义)</**h2**>

      }

    }

    //2.渲染组件到页面

    ReactDOM.render(<*MyComponent*/>,document.getElementById('test'))

    /\*

      执行了ReactDOM.render(<MyComponent/>.......之后，发生了什么？

          1.React解析组件标签，找到了MyComponent组件。

          2.发现组件是使用类定义的，随后new出来该类的实例，并通过该实例调用到原型上的render方法。

          3.将render返回的虚拟DOM转为真实DOM，随后呈现在页面中。

    \*/

### 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. 渲染类组件标签的基本流程

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

## 2.2. 组件实例三大核心属性1: state

### 2.2.1. 练习

*class* Weather **extends** React.*Component*{

*constructor*(*props*){

        console.log('constructor');

        super(props)

        this.state **=** {isHot:false,wind:'微风'}

        //解决changeWeather中this指向问题

        this.changeWeather **=** this.changeWeather.bind(this)

      }

      //render调用几次？ ———— 1+n次 1是初始化的那次 n是状态更新的次数

      render(){

*const* {isHot,wind} **=** this.state

**return** <**h1** onClick**=**{this.changeWeather}>今天天气很{isHot **?** '炎热' **:** '凉爽'}，{wind}</**h1**>

      }

      changeWeather(){

        //changeWeather放在哪里？ ———— Weather的原型对象上，供实例使用

        //由于changeWeather是作为onClick的回调，所以不是通过实例调用的，是直接调用

        //类中的方法默认开启了局部的严格模式，所以changeWeather中的this为undefined

*const* isHot **=** this.state.isHot

        //严重注意：状态必须通过setState进行更新,且更新是一种合并，不是替换。

        this.setState({isHot:**!**isHot})

        //this.state.isHot = !isHot //这是错误的写法，状态(state)不可直接更改

      }

    }

    ReactDOM.render(<*Weather*/>,document.getElementById('test'))

简写方式

    //1.创建组件

*class* Weather **extends** React.*Component*{

      //初始化状态

      state **=** {isHot:false,wind:'微风'}

      render(){

*const* {isHot,wind} **=** this.state

**return** <**h1** onClick**=**{this.changeWeather}>今天天气很{isHot **?** '炎热' **:** '凉爽'}，{wind}</**h1**>

      }

      //自定义方法————要用赋值语句的形式+箭头函数

      changeWeather **=** ()*=>*{

*const* isHot **=** this.state.isHot

        this.setState({isHot:**!**isHot})

      }

    }

    //2.渲染组件到页面

    ReactDOM.render(<*Weather*/>,document.getElementById('test'))

### 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key-value的组合)
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. 强烈注意

1. 组件中render方法中的this为组件实例对象
2. 组件自定义的方法中this为undefined，如何解决？
   1. 强制绑定this: 通过函数对象的bind()
   2. 箭头函数
3. 状态数据，不能直接修改或更新

## 2.3. 组件三大核心属性2: props

### 2.3.1. 理解

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

### 2.3.2. 作用

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

### 2.3.3. 编码操作

1. 内部读取某个属性值

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

1. 对props中的属性值进行类型限制和必要性限制

第一种方式（React v15.5 开始已弃用）：

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

第二种方式（新）：使用prop-types库进限制（需要引入prop-types库）

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

1. 扩展属性: 将对象的所有属性通过props传递

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

1. 默认属性值：

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

1. 组件类的构造函数

*constructor*(*props*){

        //构造器是否接收props，是否传递给super，取决于：是否希望在构造器中通过this访问props

        // console.log(props);

        super(props)

        console.log('constructor',this.props);

      }

****官方文档内容：****

****如果不初始化 state 或不进行方法绑定，则不需要为 React 组件实现构造函数。****

在 React 组件挂载之前，会调用它的构造函数。在为 React.Component 子类实现构造函数时，应在其他语句之前前调用 super(props)。否则，this.props 在构造函数中可能会出现未定义的 bug。

通常，在 React 中，构造函数仅用于以下两种情况：

* 通过给 this.state 赋值对象来初始化[内部 state](https://zh-hans.reactjs.org/docs/state-and-lifecycle.html)。
* 为[事件处理函数](https://zh-hans.reactjs.org/docs/handling-events.html)绑定实例

### 2.3.4. 类组件案例

*class* Person **extends** React.*Component*{

      render(){

*const* {name,age,sex,children} **=** this.props

        //props是只读的

        //this.props.name = 'jack' //此行代码会报错，因为props是只读的

**return** (

          <**ul**>

            <**li**>姓名：{name}</**li**>

            <**li**>性别：{sex}</**li**>

            <**li**>年龄：{age**+**1}</**li**>

            <**li**>子集:{children **&&** children.join(',')}</**li**>

          </**ul**>

        )

      }

    }

    //对标签属性进行类型、必要性的限制

    Person.propTypes **=** {

      name:PropTypes.string.isRequired, //限制name必传，且为字符串

      sex:PropTypes.string,//限制sex为字符串

      age:PropTypes.number,//限制age为数值

      speak:PropTypes.func,//限制speak为函数

      children:PropTypes.array

    }

    // //指定默认标签属性值

    Person.defaultProps **=** {

      sex:'男',//sex默认值为男

      age:18 //age默认值为18

    }

    ReactDOM.render(<*Person* name**=**{100} speak**=**{speak} children**=**{[2,345,6]}/>,document.getElementById('app'))

    ReactDOM.render(<*Person* name**=**"tom" age**=**{18} sex**=**"女"/>,document.getElementById('app2'))

*const* p **=** {name:'老刘',age:18,sex:'女'}

    ReactDOM.render(<*Person* {**...**p}/>,document.getElementById('app3'))

*function* speak(){

      console.log('我说话了');

    }

### 2.3.5. props简写方式

将对props的配置放在类中，不要写在外面。但是写在类中如果直接赋值，那边这个变量将变成类的实例成员，如何挂载到类本身上去呢？使用static申明，表示类的静态成员。

**static** propTypes **=** {

        name:PropTypes.string.isRequired, //限制name必传，且为字符串

        sex:PropTypes.string,//限制sex为字符串

        age:PropTypes.number,//限制age为数值

      }

### 2.3.6. 函数组件案例

*function* Person (*props*){

*const* {name,age,sex} **=** props

**return** (

          <**ul**>

            <**li**>姓名：{name}</**li**>

            <**li**>性别：{sex}</**li**>

            <**li**>年龄：{age}</**li**>

          </**ul**>

        )

    }

    Person.propTypes **=** {

      name:PropTypes.string.isRequired, //限制name必传，且为字符串

      sex:PropTypes.string,//限制sex为字符串

      age:PropTypes.number,//限制age为数值

    }

    //渲染组件到页面

    ReactDOM.render(<*Person* name**=**"jerry"/>,document.getElementById('app'))

## 2.4. 组件三大核心属性3: refs与事件处理

### 2.4.2. 理解

组件内的标签可以定义ref属性来标识自己

### 2.4.3. 编码

1. 字符串形式的ref

**<input ref="input1"/>**

*class* Mycom **extends** React.*Component*{

      render(){

**return**(

          <**div**>

            <**input** type**=**"text" ref**=**"input1"/>

            <**button** onClick**=**{this.showDate}>btn</**button**>

            </**div**>

        )

      }

      showDate **=** ()*=>*{

*const* {input1} **=** this.refs

        alert(input1.value)

      }

    }

    ReactDOM.render(<*Mycom* />,document.getElementById('app'))

1. 回调形式的ref

**<input ref={(c)=>{this.input1 = c}}/>**

****官方文档内容：****

关于回调 refs 的说明

如果 ref 回调函数是以内联函数的方式定义的，在更新过程中它会被执行两次，第一次传入参数 null，然后第二次会传入参数 DOM 元素。这是因为在每次渲染时会创建一个新的函数实例，所以 React 清空旧的 ref 并且设置新的。通过将 ref 的回调函数定义成 class 的绑定函数的方式可以避免上述问题，但是大多数情况下它是无关紧要的

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

1. createRef创建ref容器·

**myRef = React.createRef()**

**<input ref={this.myRef}/>**

        React.createRef调用后可以返回一个容器，该容器可以存储被ref所标识的节点,该容器是“专人专用”的

      myRef **=** React.createRef()

      render() {

**return** (

          <**div**>

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

            <**button** onClick**=**{this.showInfo}>show</**button**>

          </**div**>

        )

      }

      showInfo **=** () *=>* {

        console.log(this.myRef.current.value);

        // alert(this.myRef)

      }

### 2.4.4. 事件处理

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

## 2.5. 收集表单数据

### 2.5.1. 效果

包含表单的组件分类

* 1. 受控组件

*class* Login **extends** React.*Component*{

      handleSubmit **=** (*event*)*=>*{

        event.preventDefault() //阻止表单提交

*const* {username,password} **=** this

        alert(`你输入的用户名是：${username.value},你输入的密码是：${password.value}`)

      }

      render(){

**return**(

          <**form** onSubmit**=**{this.handleSubmit}>

            用户名：<**input** ref**=**{*c* *=>* this.username **=** c} type**=**"text" name**=**"username"/>

            密码：<**input** ref**=**{*c* *=>* this.password **=** c} type**=**"password" name**=**"password"/>

            <**button**>登录</**button**>

          </**form**>

        )

      }

    }

* 1. 非受控组件

*class* Login **extends** React.*Component*{

      //初始化状态

      state **=** {

        username:'', //用户名

        password:'' //密码

      }

      saveUsername **=** (*event*)*=>*{

        this.setState({username:event.target.value})

      }

      savePassword **=** (*event*)*=>*{

        this.setState({password:event.target.value})

      }

      //表单提交的回调

      handleSubmit **=** (*event*)*=>*{

        event.preventDefault() //阻止表单提交

*const* {username,password} **=** this.state

        alert(`你输入的用户名是：${username},你输入的密码是：${password}`)

      }

      render(){

**return**(

          <**form** onSubmit**=**{this.handleSubmit}>

            用户名：<**input** onChange**=**{this.saveUsername} type**=**"text" name**=**"username"/>

            密码：<**input** onChange**=**{this.savePassword} type**=**"password" name**=**"password"/>

            <**button**>登录</**button**>

          </**form**>

        )

      }

    }

### 2.5.2. 高阶函数和柯里化

当一个表单中需要填写的数据多的时候，就不可能像上面一样，一个元素写一个事件，为了效率必须复用函数。

*class* Mycom **extends** React.*Component* {

      state **=** {

        username: '', //用户名

        password: '' //密码

      }

      saveForm **=** (*type*) *=>* {

**return** (*e*)*=>*{

          this.setState({[type]:e.target.value})

        }

      }

      saveForm2 **=** (*type*,*e*) *=>* {

        this.setState({[type]:e.target.value})

      }

      //表单提交的回调

      handleSubmit **=** (*event*) *=>* {

        event.preventDefault()

*const* { username, password } **=** this.state

        alert(`你输入的用户名是：${username},你输入的密码是：${password}`)

      }

      render() {

**return** (

          <**form** onSubmit**=**{this.handleSubmit}>

            用户名：<**input** onChange**=**{this.saveForm('username')} type**=**"text" name**=**"username" />

            密码：<**input** onChange**=**{(*e*)*=>*this.saveForm2('password',e)} type**=**"password" name**=**"password" />

            <**button**>登录</**button**>

          </**form**>

        )

      }

    }

## 2.6. 组件的生命周期

### 2.6.3. 生命周期流程图(旧)



生命周期的三个阶段（旧）

**1. 初始化阶段:** 由ReactDOM.render()触发---初次渲染

* + - 1. constructor()
      2. componentWillMount()
      3. render()
      4. componentDidMount()

**2. 更新阶段:** 由组件内部this.setSate()或父组件重新render触发

1. shouldComponentUpdate()
2. componentWillUpdate()
3. render()
4. componentDidUpdate()

**3. 卸载组件:** 由ReactDOM.unmountComponentAtNode()触发

1. componentWillUnmount()

### 2.6.4. 生命周期流程图(新)



生命周期的三个阶段（新）

**1. 初始化阶段:** 由ReactDOM.render()触发---初次渲染

1. constructor()
2. **getDerivedStateFromProps**
3. render()
4. componentDidMount()

**static** getDerivedStateFromProps(*props*,*state*){

        console.log('getDerivedStateFromProps',props,state);

**return** null

      }

**2. 更新阶段:** 由组件内部this.setSate()或父组件重新render触发

1. **getDerivedStateFromProps**
2. shouldComponentUpdate()
3. render()
4. **getSnapshotBeforeUpdate**
5. componentDidUpdate()

**3. 卸载组件:** 由ReactDOM.unmountComponentAtNode()触发

1. componentWillUnmount()

### 2.6.6. 即将废弃的勾子

1. componentWillMount
2. componentWillReceiveProps
3. componentWillUpdate

现在使用会出现警告，下一个大版本需要加上UNSAFE\_前缀才能使用，以后可能会被彻底废弃，不建议使用。

### 2.6.7. 快照案例

*class* NewsList **extends** React.*Component* {

      state **=** {

        arr:[]

      }

      render() {

**return** (

            <**ul** ref**=**"ul">

            {

              this.state.arr.map((*item*,*index*)*=>*{

**return** <**li** key**=**{index}>{item}</**li**>

              })

            }

          </**ul**>

        )

      }

      getSnapshotBeforeUpdate(){

**return** this.refs.ul.scrollHeight

      }

      componentDidUpdate(*preProps*,*preState*,*snapValue*){

        this.refs.ul.scrollTop **+=** this.refs.ul.scrollHeight**-**snapValue

      }

      componentDidMount(){

*let* timer **=** setInterval(() *=>* {

*const* {arr} **=**this.state

*const* news **=** '新闻' **+**(arr.length**+**1)

          this.setState({arr:[news,**...**arr]})

        }, 1000);

      }

    }

    ReactDOM.render(<*NewsList* />, document.getElementById('app'))

## 2.7. 虚拟DOM与DOM Diffing算法

### 2.7.1. 效果

### 2.7.2. 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

### 3.1.2. 创建项目并启动

**第一步**，全局安装：npm i -g create-react-app

**第二步**，切换到想创项目的目录，使用命令：create-react-app hello-react

**第三步**，进入项目文件夹：cd hello-react

**第四步**，启动项目：npm start

### 3.1.3. react脚手架项目结构

public ---- 静态资源文件夹

favicon.icon ------ 网站页签图标

**index.html -------- 主页面**

logo192.png ------- logo图

logo512.png ------- logo图

manifest.json ----- 应用加壳的配置文件

robots.txt -------- 爬虫协议文件

src ---- 源码文件夹

App.css -------- App组件的样式

**App.js --------- App组件**

App.test.js ---- 用于给App做测试

index.css ------ 样式

**index.js ------- 入口文件**

logo.svg ------- logo图

reportWebVitals.js

--- 页面性能分析文件(需要web-vitals库的支持)

setupTests.js

---- 组件单元测试的文件(需要jest-dom库的支持)

 <!-- %PUBLIC\_URL%代表public文件夹的路径 -->

    <**link** rel="icon" href="%PUBLIC\_URL%/favicon.ico" />

    <!-- 开启理想视口，用于做移动端网页的适配 -->

    <**meta** name="viewport" content="width=device-width, initial-scale=1" />

    <!-- 用于配置浏览器页签+地址栏的颜色(仅支持安卓手机浏览器) -->

    <**meta** name="theme-color" content="red" />

    <**meta**

      name="description"

      content="Web site created using create-react-app"

    />

    <!-- 用于指定网页添加到手机主屏幕后的图标 -->

    <**link** rel="apple-touch-icon" href="%PUBLIC\_URL%/logo192.png" />

    <!-- 应用加壳时的配置文件 -->

    <**link** rel="manifest" href="%PUBLIC\_URL%/manifest.json" />

ReactDOM.render(

  <*React.StrictMode*>

    <*App* />

  </*React.StrictMode*>,

  document.getElementById('root')

);

### 3.1.4. react中的模块化

一 、组件模块化

1 有些js文件里面是js代码，有些js文件其实写的是一个组件。为了区分，方法一：组件js第一个首字母大写。方法二：js组件写jsx

2 一个组件一般都在一个文件夹里面，包含css文件。引入的时候写全部路径太麻烦，可以将文件夹中组件名字改为index.js ，引入时只写到文件夹名就可以了，默认会找下面的index文件作为目标文件

1. 样式模块化

1 使用css预处理，最外层标签名为组件名，避免与其他样式冲突

2 把css文件改为index.module.css 引入的时候。（webpack中有配置设置的）

**import** hello **from** './index.module.css'

### 3.1.4. 功能界面的组件化编码流程（通用）

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

3.1 动态显示初始化数据

3.1.1 数据类型

3.1.2 数据名称

3.1.2 保存在哪个组件?

3.2 交互(从绑定事件监听开始)

## 3.2. 组件的组合使用-TodoList

## 一、todoList案例相关知识点

    1.拆分组件、实现静态组件，注意：className、style的写法

    2.动态初始化列表，如何确定将数据放在哪个组件的state中？

          ——某个组件使用：放在其自身的state中

          ——某些组件使用：放在他们共同的父组件state中（官方称此操作为：状态提升）

    3.关于父子之间通信：

        1.【父组件】给【子组件】传递数据：通过props传递

        2.【子组件】给【父组件】传递数据：通过props传递，要求父提前给子传递一个函数

    4.注意defaultChecked 和 checked的区别，类似的还有：defaultValue 和 value

    5.状态在哪里，操作状态的方法就在哪里

第4章：React ajax

## 4.2. axios

### 4.2.1. 文档

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

### 4.2.2. 相关API

1. GET请求

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

1. POST请求

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

## 4.4. 消息订阅-发布机制

1. 工具库: PubSubJS
2. 下载: npm install pubsub-js --save
3. 使用:
   * + 1. import PubSub from 'pubsub-js' //引入
       2. PubSub.subscribe('delete', function(data){ }); //订阅
       3. PubSub.publish('delete', data) //发布消息

### 4.4.1. PubSubjs使用案例

a组件 （a b 为兄弟组件）

**import** React, { Component } **from** 'react'

**import** PubSub **from** 'pubsub-js'

**export** **default** *class* pubsub **extends** *Component* {

  publish **=** ()*=>*{

    PubSub.publish('MY TOPIC', 'hello world!');

  }

  render() {

**return** (

      <**div**>

        <**h1**>hello 我是一个A组件 我是发布者</**h1**>

        <**button** onClick**=**{this.publish}>点击发布</**button**>

      </**div**>

    )

  }

}

b组件

**import** React, { Component } **from** 'react'

**import** PubSub **from** 'pubsub-js'

**export** **default** *class* pubsub **extends** *Component* {

  state **=** { msg: 'null' }

  mySubscriber **=** (*\_*, *data*) *=>* {

    console.log(\_);

    console.log(data);

    this.setState({ msg: data })

  };

  componentDidMount () {

    this.token **=** PubSub.subscribe('MY TOPIC', this.mySubscriber);

  }

  render () {

*const* { msg } **=** this.state

**return** (

      <**div**>

        <**h1**>hello 我是一个B组件 我是接受者</**h1**>

        <**p**>接受显示 :{msg}</**p**>

      </**div**>

    )

  }

}

## 4.5. 扩展：Fetch

### 4.5.1. 文档

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

### 4.5.2. 特点

1. fetch: 原生函数，不再使用XmlHttpRequest对象提交ajax请求
2. 老版本浏览器可能不支持

### 4.5.3. 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)    }) |

第5章：React路由

## 5.1. 相关理解

### 5.1.1. 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，只会做页面的**局部更新。**
4. 数据都需要通过ajax请求获取, 并在前端异步展现。

### 5.1.2. 路由的理解

1. **什么是路由?**
   1. 一个路由就是一个映射关系(key:value)
   2. key为路径, value可能是function或component
2. **路由分类**
   1. 后端路由：
      1. 理解： value是function, 用来处理客户端提交的请求。
      2. 注册路由： router.get(path, function(req, res))
      3. 工作过程：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
   2. 前端路由：
      1. 浏览器端路由，value是component，用于展示页面内容。
      2. 注册路由: <Route path="/test" component={Test}>
      3. 工作过程：当浏览器的path变为/test时, 当前路由组件就会变为Test组件

### 5.1.3. react-router-dom的理解

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

下载react-router-dom: npm install --save react-router-dom

## 5.2. react-router-dom相关API

### 5.2.1. 内置组件

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

      1.导航区的a标签改为Link标签

            <Link to="/xxxxx">Demo</Link>

      2.展示区写Route标签进行路径的匹配

            <Route path='/xxxx' component={Demo}/>

      3.<App>的最外侧包裹了一个<BrowserRouter>或<HashRouter>

路由组件和一般组件

## 四、路由组件与一般组件

      1.写法不同：

            一般组件：<Demo/>

            路由组件：<Route path="/demo" component={Demo}/>

      2.存放位置不同：

            一般组件：components

            路由组件：pages

      3.接收到的props不同：

            一般组件：写组件标签时传递了什么，就能收到什么

            路由组件：接收到三个固定的属性

                      history:

                            go: ƒ go(n)

                            goBack: ƒ goBack()

                            goForward: ƒ goForward()

                            push: ƒ push(path, state)

                            replace: ƒ replace(path, state)

                      location:

                            pathname: "/about"

                            search: ""

                            state: undefined

                      match:

                            params: {}

                            path: "/about"

                            url: "/about"

## 六、Switch的使用

        1.通常情况下，path和component是一一对应的关系。

        2.Switch可以提高路由匹配效率(单一匹配)。

                <*Switch*>

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

                  <*Route* path**=**"/home" component**=**{Home}/>

                  <*Route* path**=**"/home" component**=**{Test}/>

                </*Switch*>

## 七、解决多级路径刷新页面样式丢失的问题

        1.public/index.html 中 引入样式时不写 ./ 写 / （常用）

        2.public/index.html 中 引入样式时不写 ./ 写 %PUBLIC\_URL% （常用）

        3.使用HashRouter

## 八、路由的严格匹配与模糊匹配

        1.默认使用的是模糊匹配（简单记：【输入的路径】必须包含要【匹配的路径】，且顺序要一致）

        2.开启严格匹配：<Route exact={true} path="/about" component={About}/>

        3.严格匹配不要随便开启，需要再开，有些时候开启会导致无法继续匹配二级路由

## 九、Redirect的使用

        1.一般写在所有路由注册的最下方，当所有路由都无法匹配时，跳转到Redirect指定的路由

        2.具体编码：

            <Switch>

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

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

              <Redirect to="/about"/>

            </Switch>

### 5.2.2. 其它

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

## 5.3. 基本路由使用

### 5.3.1. 自定义封装navlink组件

              <*MyNavLink* to**=**"/about">About</*MyNavLink*>

              <*MyNavLink* to**=**"/about" children**=**"About"></*MyNavLink*>

**export** **default** *class* MyNavLink **extends** *Component* {

  render() {

    // console.log(this.props);

**return** (

      <*NavLink* activeClassName**=**"atguigu" className**=**"list-group-item" {**...**this.props}/>

      <*NavLink* activeClassName**=**"atguigu" className**=**"list-group-item" {**...**this.props}>{this.props.children}</*NavLink*>

    )

  }

}

## 5.4. 嵌套路由使用

## 十、嵌套路由

        1.注册子路由时要写上父路由的path值

        2.路由的匹配是按照注册路由的顺序进行的

父级

              <*MyNavLink* to**=**"/about">About</*MyNavLink*>

              <*MyNavLink* to**=**"/home">Home</*MyNavLink*>

                {/\* 注册路由 \*/}

                <*Switch*>

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

                  <*Route* path**=**"/home" component**=**{Home}/>

                  <*Redirect* to**=**"/about"/>

                </*Switch*>

子级

                <*MyNavLink* to**=**"/home/news">News</*MyNavLink*>

                <*MyNavLink* to**=**"/home/message">Message</*MyNavLink*>

            {/\* 注册路由 \*/}

            <*Switch*>

              <*Route* path**=**"/home/news" component**=**{News}/>

              <*Route* path**=**"/home/message" component**=**{Message}/>

              <*Redirect* to**=**"/home/news"/>

            </*Switch*>

## 5.5. 向路由组件传递参数数据

## 十一、向路由组件传递参数

        1.params参数

              路由链接(携带参数)：<Link to='/demo/test/tom/18'}>详情</Link>

              注册路由(声明接收)：<Route path="/demo/test/:name/:age" component={Test}/>

              接收参数：this.props.match.params

        2.search参数

              路由链接(携带参数)：<Link to='/demo/test?name=tom&age=18'}>详情</Link>

              注册路由(无需声明，正常注册即可)：<Route path="/demo/test" component={Test}/>

              接收参数：this.props.location.search

              备注：获取到的search是urlencoded编码字符串，需要借助querystring解析

        3.state参数

              路由链接(携带参数)：<Link to={{pathname:'/demo/test',state:{name:'tom',age:18}}}>详情</Link>

              注册路由(无需声明，正常注册即可)：<Route path="/demo/test" component={Test}/>

              接收参数：this.props.location.state

  备注：三种方式刷新都可以保留住参数

**import** qs **from** 'querystring'

*let* obj **=** qs.parse(this.props.location.search.slice(1))

## 5.6. 多种路由跳转方式

路由跳转操作分为push和replace，默认采用push。

 <*Link* replace to**=**{{pathname:'/home/message/detail'}>{msgObj.title}</*Link*>

## 十二、编程式路由导航

          借助this.prosp.history对象上的API对操作路由跳转、前进、后退

              -this.prosp.history.push()

    //push跳转+携带params参数

     this.props.history.push(`/home/message/detail/${id}/${title}`)

    //push跳转+携带search参数

     this.props.history.push(`/home/message/detail?id=${id}&title=${title}`)

    //push跳转+携带state参数

    this.props.history.push(`/home/message/detail`,{id,title})

              -this.prosp.history.replace()

              this.props.history.goBack()

              this.props.history.goForward()

              this.props.history.go(**-**2)

## 十三、BrowserRouter与HashRouter的区别

      1.底层原理不一样：

            BrowserRouter使用的是H5的history API，不兼容IE9及以下版本。

            HashRouter使用的是URL的哈希值。

      2.path表现形式不一样

            BrowserRouter的路径中没有#,例如：localhost:3000/demo/test

            HashRouter的路径包含#,例如：localhost:3000/#/demo/test

      3.刷新后对路由state参数的影响

            (1).BrowserRouter没有任何影响，因为state保存在history对象中。

            (2).HashRouter刷新后会导致路由state参数的丢失！！！

      4.备注：HashRouter可以用于解决一些路径错误相关的问题。

## 5.7. withRouter

上面所说这些路由跳转api，只有在路由组件中才有效果，在一般组件中是没有这些api的。

**import** {withRouter} **from** 'react-router-dom'

**export** **default** withRouter(Header)

//withRouter可以加工一般组件，让一般组件具备路由组件所特有的API

//withRouter的返回值是一个新组件

## 5.8. BrowserRouter和Hashrouter的区别

## 十三、BrowserRouter与HashRouter的区别

      1.底层原理不一样：

            BrowserRouter使用的是H5的history API，不兼容IE9及以下版本。

            HashRouter使用的是URL的哈希值。

      2.path表现形式不一样

            BrowserRouter的路径中没有#,例如：localhost:3000/demo/test

            HashRouter的路径包含#,例如：localhost:3000/#/demo/test

      3.刷新后对路由state参数的影响

            (1).BrowserRouter没有任何影响，因为state保存在history对象中。

            (2).HashRouter刷新后会导致路由state参数的丢失！！！

      4.备注：HashRouter可以用于解决一些路径错误相关的问题。

第6章：React UI组件库

## 6.1.流行的开源React UI组件库

### 6.1.1. 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>

### 6.1.2. ant-design(国内蚂蚁金服)

1. 官网: <https://ant.design/index-cn>
2. Github: <https://github.com/ant-design/ant-design/>

## 十四、antd的按需引入+自定主题

https://ant.design/docs/react/use-with-create-react-app-cn

第7章：redux

## 7.1. redux理解

### 7.1.1. 学习文档

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

### 7.1.2. redux是什么

1. redux是一个专门用于做**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，组件通讯。

### 7.1.4. redux工作流程



## 7.2. redux的三个核心概念

### 7.2.1. action

1. 动作的对象
2. 包含2个属性
   * type：标识属性, 值为字符串, 唯一, 必要属性
   * data：数据属性, 值类型任意, 可选属性
3. 例子：{ type: 'ADD\_STUDENT',data:{name: 'tom',age:18} }

### 7.2.2. reducer

1. 作用：初始化状态、加工状态。
2. 加工时，根据旧的state和action， 产生新的state的**纯函数。**

### 7.2.3. store

1. 将state、action、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

Redux只负责管理状态，至于状态的改变驱动着页面的展示，要靠我们自己写。

解决办法：在index.js中监测store中状态的改变，一旦发生改变重新渲染<App/>z

ReactDOM.render(<*App*/>,document.getElementById('root'))

store.subscribe(()*=>*{

  ReactDOM.render(<*App*/>,document.getElementById('root'))

})

还有一种写法

  componentDidMount(){

    store.subscribe(()*=>*{

      this.setState({})

    })

  }

## 7.3. redux的核心API

### 7.3.1. createstore()

作用：创建包含指定reducer的store对象

### 7.3.2. store对象

1. 作用: redux库最核心的管理对象
2. 它内部维护着:
   * + - 1. state
         2. reducer
3. 核心方法:
   * + - 1. getState()
         2. dispatch(action)
         3. subscribe(listener)
4. 具体编码:
   * + - 1. store.getState()
         2. store.dispatch({type:'INCREMENT', number})
         3. store.subscribe(render)

### 7.3.3. applyMiddleware()

作用：应用上基于redux的中间件(插件库)

### 7.3.4. combineReducers()

作用：合并多个reducer函数

## 7.4. 使用redux编写应用

案例



store.js

**import** { createStore } **from** "redux";

//引入为Count组件服务的reducer

**import** countReducer **from** './count\_reducer'

// 一个reducer对应一个组件

**export** **default** createStore(countReducer)

constant.js

/\*

  该模块是用于定义action对象中type类型的常量值，目的只有一个：便于管理的同时防止程序员单词写错

\*/

**export** *const* INCREMENT **=** 'increment'

**export** *const* DECREMENT **=** 'decrement'

count\_action.js

/\*

  该文件专门为Count组件生成action对象

\*/

**import** {INCREMENT,DECREMENT} **from** './constant'

**export** *const* createIncrementAction **=** *data* *=>* ({type:INCREMENT,data})

**export** *const* createDecrementAction **=** *data* *=>* ({type:DECREMENT,data})

count\_reducer.js

**import** {INCREMENT,DECREMENT} **from** './constant'

*const* initState **=** 0

**export** **default** *function* countReducer (*preState* **=** initState, *action*) {

*const* { type, data } **=** action

**switch** (type) {

**case** INCREMENT:

**return** preState **+** data

**case** DECREMENT:

**return** preState **-** data

**default**:

**return** preState

  }

}

Jsx

  increment **=** ()*=>*{

*const* {value} **=** this.selectNumber

    store.dispatch(createIncrementAction(value**-**0))

  }

        <**h1**>当前求和为：{store.getState()}</**h1**>

## 7.5. redux异步编程

### 7.5.1理解：

1. redux默认是不能进行异步处理的,
2. 某些时候应用中需要在**redux中执行异步任务**(ajax, 定时器)

### 7.5.2. 使用异步中间件

npm install --save redux-thunk

## 3.求和案例\_redux异步action版

     (1).明确：延迟的动作不想交给组件自身，想交给action

     (2).何时需要异步action：想要对状态进行操作，但是具体的数据靠异步任务返回。

     (3).具体编码：

          1).yarn add redux-thunk，并配置在store中

**import** {createStore,applyMiddleware} **from** 'redux'

**import** countReducer **from** './count\_reducer'

**import** thunk **from** 'redux-thunk'

**export** **default** createStore(countReducer,applyMiddleware(thunk))

          2).创建action的函数不再返回一般对象，而是一个函数，该函数中写异步任务。

//异步action，就是指action的值为函数,异步action中一般都会调用同步action。

**export** *const* createIncrementAsyncAction **=** (*data*,*time*) *=>* {

**return** (*dispatch*)*=>*{

    setTimeout(()*=>*{

      dispatch(createIncrementAction(data))

    },time)

  }

}

          3).异步任务有结果后，分发一个同步的action去真正操作数据。

     (4).备注：异步action不是必须要写的，完全可以自己等待异步任务的结果了再去分发同步action。

Jsp中调用：

  //异步加

  incrementAsync **=** ()*=>*{

*const* {value} **=** this.selectNumber

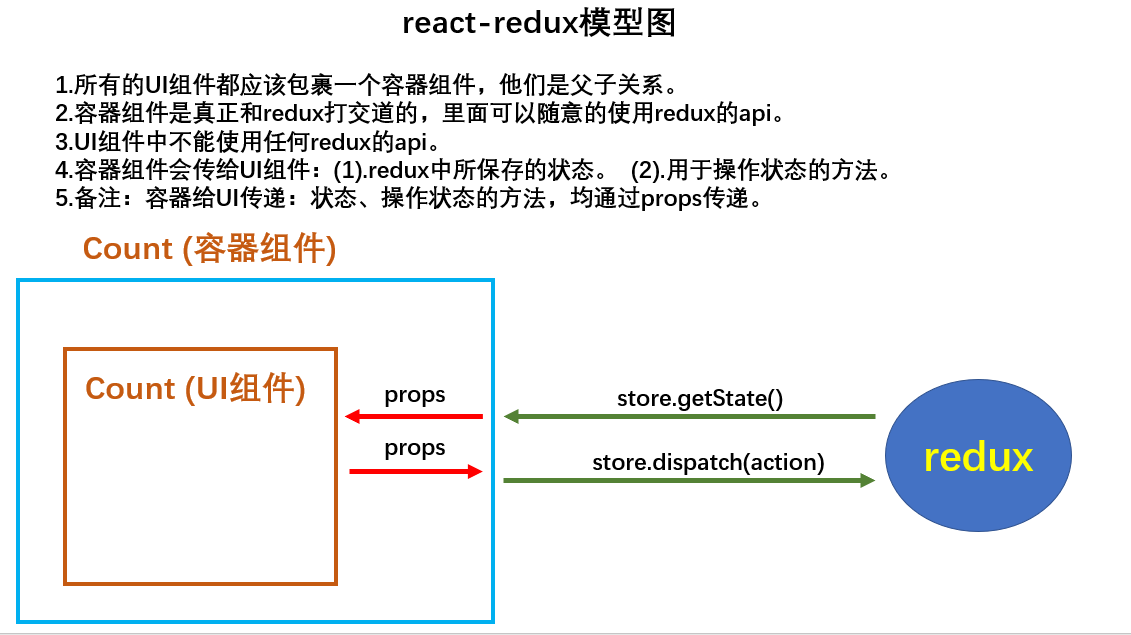
      store.dispatch(createIncrementAsyncAction(value**\***1,500))

  }

## 7.6. react-redux

### 7.6.1. 理解

一个react插件库，专门用来简化react应用中使用redux



使用react-redux，状态改变会自动触发render渲染，因为在connect的过程中帮我们写入了触发渲染的这一步操作。

一个组件要和redux“打交道”要经过哪几步？

              (1).定义好UI组件---不暴露

              (2).引入connect生成一个容器组件，并暴露，写法如下：

                  connect(

                    state => ({key:value}), //映射状态

                    {key:xxxxxAction} //映射操作状态的方法

                  )(UI组件)

              (4).在UI组件中通过this.props.xxxxxxx读取和操作状态

### 7.6.2. react-Redux将所有组件分成两大类

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

**export** **default** *class* Count **extends** *Component* {

  increment **=** ()*=>*{

*const* {value} **=** this.selectNumber

    this.props.jia(value**-**0)

  }

  render() {

    console.log('UI组件接收到的props是',this.props);

**return** (

      <**div**>

        <**h1**>当前求和为：{this.props.count}</**h1**>

    )

  }

}

1. 容器组件
   * 1. 负责管理数据和业务逻辑，不负责UI的呈现
     2. 使用 Redux 的 API
     3. 一般保存在containers文件夹下

//引入Count的UI组件

**import** CountUI **from** '../../components/count2'

//引入action

**import** {createIncrementAction,createIncrementAsyncAction} **from** '../../redux/count\_action'

**import** {connect} **from** 'react-redux'

//mapStateToProps用于传递状态

*function* mapStateToProps(*state*){

  console.log(arguments);

**return** {count:state}

}

//mapDispatchToProps用于传递操作状态的方法

*function* mapDispatchToProps(*dispatch*){

  console.log(arguments);

**return** {

    jia:*number* *=>* dispatch(createIncrementAction(number)),

    jiaAsync:(*number*,*time*) *=>* dispatch(createIncrementAsyncAction(number,time))

  }

}

**export** **default** connect(mapStateToProps,mapDispatchToProps)(CountUI)

备注1：容器组件中的store是靠props传进去的，而不是在容器组件中直接引入

备注2：mapDispatchToProps，也可以是一个对象

  //mapDispatchToProps的一般写法

  /\* dispatch => ({

    jia:number => dispatch(createIncrementAction(number)),

    jiaAsync:(number,time) => dispatch(createIncrementAsyncAction(number,time)),

  }) \*/

  //mapDispatchToProps的简写

  {

    jia:createIncrementAction,

    jiaAsync:createIncrementAsyncAction,

  }

### 7.6.3. 相关API

1. Provider：让所有组件都可以得到state数据

**import** {Provider} **from** 'react-redux'

**<Provider store={store}>**

**<App />**

**</Provider>**

--------------------------------------------------

**import** store **from** './redux/store'

 <*Count2* store**=**{store}></*Count2*>

1. connect：用于包装 UI 组件生成容器组件

**import { connect } from 'react-redux'**

**connect(**

**mapStateToprops,**

**mapDispatchToProps**

**)(Counter)**

1. mapStateToprops：将外部的数据（即state对象）转换为UI组件的标签属性

**const mapStateToprops = function (state) {**

**return {**

**value: state**

**}**

**}**

1. mapDispatchToProps：将分发action的函数转换为UI组件的标签属性

### 7.6.4. react-redux优化

上面的方法创建了2个文件，一个containers下的容器组件，一个components下的ui组件，就存在一个问题，每使用一个这样的功能就需要创建2个文件，其实这个2个组件的内容完全可以写在一起（放containers下 其实放哪都可以 规范性考虑）。一个文件中创建2个组件，ui组件直接调用，容器组件暴露出去。

### 7.6.5. react-redux数据共享

## 6.求和案例\_react-redux数据共享版

      (1).定义一个Pserson组件，和Count组件通过redux共享数据。

      (2).为Person组件编写：reducer、action，配置constant常量。

      (3).重点：Person的reducer和Count的Reducer要使用combineReducers进行合并， 合并后的总状态是一个对象！！！

**import** { createStore ,applyMiddleware,combineReducers} **from** "redux";

//汇总所有的reducer变为一个总的reducer

*const* allReducer **=** combineReducers({

  sum:countReducer,

  people:personReducer

})

**export** **default** createStore(allReducer,applyMiddleware(thunk))

      (4).交给store的是总reducer，最后注意在组件中取出状态的时候，记得“取到位”。

**export** **default** connect(

  (*state*)*=>*({personInfo:state.people,numberSum:state.sum}),

  {addOne:createAddPersonAction}

)(Person)

## 7.7. 使用上redux调试工具

### 7.7.1. 安装chrome浏览器插件



### 7.7.2. 下载工具依赖包

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

#react-redux开发者工具的使用

      (1).yarn add redux-devtools-extension

      (2).store中进行配置

          import {composeWithDevTools} from 'redux-devtools-extension'

          const store = createStore(allReducer,composeWithDevTools(applyMiddleware(thunk)))

## 8.react-redux最终版

      (1).所有变量名字要规范，尽量触发对象的简写形式。

      (2).reducers文件夹中，编写index.js专门用于汇总并暴露所有的reducer

/\* 该文件用于汇总所有的reducer为一个总的reducer\*/

//引入combineReducers，用于汇总多个reducer

**import** {combineReducers} **from** 'redux'

//引入为Count组件服务的reducer

**import** count **from** './count'

//引入为Person组件服务的reducer

**import** persons **from** './person'

//汇总所有的reducer变为一个总的reducer

**export** **default**  combineReducers({

  count,

  persons

})

## 7.8. 纯函数和高阶函数

### 7.8.1. 纯函数

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

### 7.8.2. 高阶函数

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

第8章：React扩展

## **1. setState**

### **setState更新状态的2种写法**

(1). setState(stateChange, [callback])------对象式的setState

1.stateChange为状态改变对象(该对象可以体现出状态的更改)

2.callback是可选的回调函数, 它在状态更新完毕、界面也更新后(render调用后)才被调用

(2). setState(updater, [callback])------函数式的setState

1.updater为返回stateChange对象的函数。

2.updater可以接收到state和props。

4.callback是可选的回调函数, 它在状态更新、界面也更新后(render调用后)才被调用。

总结:

1.对象式的setState是函数式的setState的简写方式(语法糖)

2.使用原则：

(1).如果新状态不依赖于原状态 ===> 使用对象方式

(2).如果新状态依赖于原状态 ===> 使用函数方式

(3).如果需要在setState()执行后获取最新的状态数据, 要在第二个callback函数中读取

## **2. lazyLoad**

### **路由组件的lazyLoad**

//1.通过React的lazy函数配合import()函数动态加载路由组件 ===> 路由组件代码会被分开打包

const Login = lazy(()=>import('@/pages/Login'))

//2.通过<Suspense>指定在加载得到路由打包文件前显示一个自定义loading界面

<Suspense fallback={<h1>loading.....</h1>}>

<Switch>

<Route path="/xxx" component={Xxxx}/>

<Redirect to="/login"/>

</Switch>

</Suspense>

## **3. Hooks**

#### **1. React Hook/Hooks是什么?**

(1). Hook是React 16.8.0版本增加的新特性/新语法

(2). 可以让你在函数组件中使用 state 以及其他的 React 特性

#### **3. State Hook**

(1). State Hook让函数组件也可以有state状态, 并进行状态数据的读写操作

(2). 语法: const [xxx, setXxx] = React.useState(initValue)

(3). useState()说明:

参数: 第一次初始化指定的值在内部作缓存

返回值: 包含2个元素的数组, 第1个为内部当前状态值, 第2个为更新状态值的函数

(4). setXxx()2种写法:

setXxx(newValue): 参数为非函数值, 直接指定新的状态值, 内部用其覆盖原来的状态值

setXxx(value => newValue): 参数为函数, 接收原本的状态值, 返回新的状态值, 内部用其覆盖原来的状态值

**export** **default** *function* HookExample() {

*const* [state,setNmuber] **=** React.useState(0)

*const* addHandle **=** ()*=>*{

    setNmuber((*state*)*=>*(state**+**1))

  }

**return** (

    <**div**>

      <**h1**>this is {state}</**h1**>

      <**button** onClick**=**{addHandle}>++</**button**>

    </**div**>

  )

}

#### **4. Effect Hook**

(1). Effect Hook 可以让你在函数组件中执行副作用操作(用于模拟类组件中的生命周期钩子)

(2). React中的副作用操作:

发ajax请求数据获取

设置订阅 / 启动定时器

手动更改真实DOM

(3). 语法和说明:

useEffect(() => {

// 在此可以执行任何带副作用操作

return () => { // 在组件卸载前执行

// 在此做一些收尾工作, 比如清除定时器/取消订阅等

}

}, [stateValue]) // 如果指定的是[], 回调函数只会在第一次render()后执行

(4). 可以把 useEffect Hook 看做如下三个函数的组合

componentDidMount()

componentDidUpdate()

componentWillUnmount()

*const* [count,setCount] **=** React.useState(0)

React.useEffect(()*=>*{

*let* timer **=** setInterval(()*=>*{

      setCount(*count* *=>* count**+**1 )

    },1000)

**return** ()*=>*{

      clearInterval(timer)

    }

  },[count])

#### **5. Ref Hook**

(1). Ref Hook可以在函数组件中存储/查找组件内的标签或任意其它数据

(2). 语法: const refContainer = useRef()

(3). 作用:保存标签对象,功能与React.createRef()一样

*const* myRef **=** React.useRef()

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

*function* show(){

    alert(myRef.current.value)

  }

## **4. Fragment**

使用

<Fragment><Fragment>

<></>

作用

可以不用必须有一个真实的DOM根标签了

## **5. Context**

理解

一种组件间通信方式, 常用于【祖组件】与【后代组件】间通信

使用

1) 创建Context容器对象：

const XxxContext = React.createContext()

2) 渲染子组时，外面包裹xxxContext.Provider, 通过value属性给后代组件传递数据：

<xxxContext.Provider value={数据}>

子组件

</xxxContext.Provider>

3) 后代组件读取数据：

//第一种方式:仅适用于类组件

static contextType = xxxContext // 声明接收context

this.context // 读取context中的value数据

//第二种方式: 函数组件与类组件都可以

<xxxContext.Consumer>

{

value => ( // value就是context中的value数据

要显示的内容

)

}

</xxxContext.Consumer>

注意

在应用开发中一般不用context, 一般都用它的封装react插件

## **6. 组件优化**

### **Component的2个问题**

1.只要执行setState(),即使不改变状态数据, 组件也会重新render() ==> 效率低

2.只当前组件重新render(), 就会自动重新render子组件，纵使子组件没有用到父组件的任何数据 ==> 效率低

效率高的做法

只有当组件的state或props数据发生改变时才重新render()

原因

Component中的shouldComponentUpdate()总是返回true

解决

办法1:

自己手动重写shouldComponentUpdate()方法

比较新旧state或props数据, 如果有变化才返回true, 如果没有返回false

办法2:

使用PureComponent

PureComponent重写了shouldComponentUpdate(), 只有state或props数据有变化才返回true

**import** React, { PureComponent } **from** 'react'

**export** **default** *class* Parent **extends** *PureComponent* {

注意:

只是进行state和props数据的浅比较, 如果只是数据对象内部数据变了, 返回false ，不要直接修改state数据, 而是要产生新数据

项目中一般使用PureComponent来优化。

## **7. render props**

如何向组件内部动态传入带内容的结构(标签)?（插槽）

Vue中:

使用slot技术, 也就是通过组件标签体传入结构 <A><B/></A>

React中:

使用children props: 通过组件标签体传入结构

使用render props: 通过组件标签属性传入结构,而且可以携带数据，一般用render函数属性

### **children props**

<A>

<B>xxxx</B>

</A>

{this.props.children}

问题: 如果B组件需要A组件内的数据, ==> 做不到

### **render props**

<A render={(data) => <C data={data}></C>}></A>

A组件: {this.props.render(内部state数据)}

C组件: 读取A组件传入的数据显示 {this.props.data}

## **8. 错误边界**

理解：

错误边界(Error boundary)：用来捕获后代组件错误，渲染出备用页面

特点：

只能捕获后代组件生命周期产生的错误，不能捕获自己组件产生的错误和其他组件在合成事件、定时器中产生的错误

使用方式：

getDerivedStateFromError配合componentDidCatch

// 生命周期函数，一旦后台组件报错，就会触发

static getDerivedStateFromError(error) {

console.log(error);

// 在render之前触发

// 返回新的state

return {

hasError: true,

};

}

componentDidCatch(error, info) {

// 可以统计页面的错误。发送请求发送到后台去记录

console.log(error, info);

}

**export** **default** *class* Parent **extends** *Component* {

  state **=** {

    hasError:'' //用于标识子组件是否产生错误

  }

  //当Parent的子组件出现报错时候，会触发getDerivedStateFromError调用，并携带错误信息

**static** getDerivedStateFromError(*error*){

    console.log('@@@',error);

**return** {hasError:error}

  }

  componentDidCatch(){

    console.log('此处统计错误，反馈给服务器，用于通知编码人员进行bug的解决');

  }

  render() {

**return** (

      <**div**>

        <**h2**>我是Parent组件</**h2**>

        {this.state.hasError **?** <**h2**>当前网络不稳定，稍后再试</**h2**> **:** <*Child*/>}

      </**div**>

    )

  }

}

## **9. 组件通信方式总结**

组件间的关系：

父子组件、兄弟组件（非嵌套组件）、祖孙组件（跨级组件）

几种通信方式：

1.props：

(1).children props

(2).render props

2.消息订阅-发布：

pubs-sub、event等等

3.集中式管理：

redux、dva等等

4.conText:

生产者-消费者模式

比较好的搭配方式：

父子组件：props

兄弟组件：消息订阅-发布、集中式管理

祖孙组件(跨级组件)：消息订阅-发布、集中式管理、conText(开发用的少，封装插件用的多)