# React第一天 React的初步

## 介绍

- react是有facebook公司的前端开发团队进行开发和维护的一套js库
- react一切皆组件，all in JS
- react是用来构建web用户交互界面
- react提供一套虚拟DOM(virtual dom),机制
- react中facebook专为其提供一套语法糖---jsx

## 优点:

- 一切皆组件，在react中几乎全是使用组件进行开发
- 速度快，react提供了虚拟dom机制,
- 跨浏览器兼容，甚至在IE8中也可以使用
- 单向数据流，适合开发复杂的项目，出了问题容易溯源

## 缺点

- react本身不是一套完整的框架,react需要结合flux和redux等架构来实现完整的项目
- react最多只能算mvc中的view

## react解决的痛点

- 组件化: 在react中一切组件组件
- 开发效率:react中使用组件化思想,使代码变得易于读懂和开发
- 运行效率:react采用的是虚拟dom机制,相比于原生开发效率速度更高
- 可维护:react中使用组件来构建大型项目,使得一些复杂的项目变得易于维护,.
- 在用户体验方面:react采用SPA(单页面应用),使得用户体验更好.

## 下载安装

- 方式一

```
<script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"
crossorigin></script>
```

- 方式二

```
npm i create-react-app -g 全局安装工具链
create-react-app -V  查看版本
create-react-app 项目名称  创建项目
cd xxx  进入项目目录
npm start  开启项目

// 修改默认端口号
node_modules/react-scripts/scripts/start.js
const DEFAULT_PORT = parseInt(process.env.PORT, 10) || 5000;
```

## 项目目录介绍

```
my-app
    -node_modules    #项目的依赖包
    -public          #公共资源文件
        index.html   #项目唯一的一个页面
    -src             #项目目录
        App.css      #根组件的样式
        App.js       #根组件
        App.test.js  #根组件的测试文件
        index.css    #唯一的入口文件的样式
        index.js     #唯一的入口文件
     .gitignore      #上传到Git上忽略的内容
     package-lock.json #安装node_modules 中的依赖包的锁定版本
     package.json    #安装node_modules中相关的依赖项
     READ.me         #项目解读文件
```

## JSX语法：JSX是对js的扩充，允许js和html混写

1. 遇到 <>,JSX 就解析成 HTML;遇到 {}按 JavaScript 解析。
2. 虚拟 DOM 只允许有一个根节点
3. JSX 标签可以相互嵌套注意换行和缩进,增强代码的可读性
4. 建议使用小括号括将 JSX 代码括起来 ,防止分号自动插入的 bug
5. 通过 JSX 语法将数据绑定到虚拟 DOM 对象(是一个js的object),再通过 render 将数据渲染到真实 DOM

## 修改标签内容

内容可以有简单的运算，类似vue的文本插值

## 修改标签属性

```
  <img src={imgUrl} alt="" />
  <a href={url} >中公教育</a>
  {/* 在jsx的语法中，html的类名class要写成className，js中的class还写成class */}
  <div className={className1}>修改我的class</div>
  {/* 在jsx的语法中，style的类型需要是对象，所以需要两个花括号。多个单词的属性名要写成：
fontSize */}
  <div style={{ color: 'red', fontSize: '40px' }}>修改我的style</div>
```

## 条件渲染

```
  <div>条件渲染：简单的用三元表达式写，复杂的用if...else</div>
  <div>
    {
      (function () {
        if (haswude) {
          return (<p>666</p>)
        } else {
          return (<p>555</p>)
        }
      })()
    }
  </div>
```

## 循环渲染

```
<div>循环渲染，类似vue的v-for。使用map方法实现</div>
<ul>
  {
    list.map(item => {
      return (<li key={item.name}>{item.name}---{item.age}</li>)
    })
  }
</ul>
```

## 组件

函数组件和类组件的区别：1.比较熟悉function，不熟悉class，class功能更强大，function性能更好；2.class可以有this；3.class有生命周期钩子，函数组件没有；4.state的处理。函数组件适用于简单的静态页面展示。

## 事件绑定

- 两种绑定方式

```
<button onClick={(e) => this.showData('ES6', e)}>ES6-箭头函数</button>
{/* 默认this是undefined，需要通过bind(this)绑定 */}
<button onClick={this.showData.bind(this, 'ES5')}>ES5-普通函数</button>
{/* <button onClick={this.showData}>ES5-普通函数</button> */}
```

- 传递参数

```
{/* this指向当前组件,参数在e前面 */}
<button onClick={(e) => this.showData('ES6', e)}>ES6-箭头函数</button>
{/* 默认this是undefined，需要通过bind(this)绑定,参数在this后面 */}
<button onClick={this.showData.bind(this, 'ES5')}>ES5-普通函数</button>
{/* <button onClick={this.showData}>ES5-普通函数</button> */}
```

- 事件对象

> 传递事件对象e之后，可以获取e相关的信息，例如e.target.value等

- 阻止默认事件

> e.preventDefault();(写return false不行)

- 阻止冒泡

> e.stopPropagation()

- 改为捕获的方式

> 加上Capture这个关键字，例如：onClickCapture={(e)=>this.clickHandler1('big',e)}

## state 状态机：管理数据状态，可以初始化数据，类似vue中的data

1. 初始化

```
  this.state = {
    age: 18,
    money: 1000,
    msg: ''
  }
```

2. 获取状态:this.state.xxx
3. 修改状态,这是一个异步的方式，想要获取最新的值，可以在回调函数中获取

```
  this.setState({
    money: this.state.money + 100
  }, ()=> {
    console.log(this.state.money);
  })
```

4. react版的双向绑定

```
    <input value={this.state.msg} onChange={(e)=>this.twoWayBind(e)} />

    twoWayBind(e) {
      this.setState({
        msg: e.target.value
      })
    }
```

# React第二天 React组件、生命周期和性能优化

## props的使用

在组件中通过this.props.xx获取，如果想在构造函数中使用props，需要传入参数

```
constructor(props){
  // 调用super之后，才可以使用this
  super(props);
  // state：状态机。管理数据状态，可以初始化数据，类似vue中的data
  this.state = {
    age: 18,
    val: this.props.val // 使用props中的值进行初始化
  }
}
render() {
  console.log(this.props);
  // 在其他函数中可以通过解构赋值获取
  const {age,name} = this.props
  return (
      <div className="alert alert-info" >
          <h1>用户信息</h1>
          <div>name的值为:{name}</div>
          <div>age的值为:{age}</div>
      </div>
  )
}
```

通过props实现组件传值：

1. 父传子：通过组件的属性传递

```
// 在父组件中
<ComC
  info={this.state.msg}
  info2="2"
></ComC>
// 在子组件中
render(props) {
  return (
    <div>
      <mark>{this.state.msg}</mark>
      {/* 子组件通过this.props接受父组件的数据 */}
      <h1>{this.props.info}</h1>
      <h1>{JSON.stringify(this.props)}</h1>
      {/* 子组件触发父组件的函数，并在参数的位置传入数据，传给父组件 */}
    </div>
  )
}
```

子组件不能修改父组件传过来的值，read only property，这是只读属性

2. 子传父

```
// 父组件传给子组件的函数，函数中包含了获取数据之后的操作
fn(data) {
  this.setState({
    childMsg: data
  })
}
// 在父组件中，先向子组件传递一个函数
<ComC
  fn={this.fn.bind(this)}
></ComC>
// 在子组件中调用传过来的函数，函数的参数就是子传父的数据
<button onClick={(e)=>this.props.fn(this.state.msg,e)}>子传父</button>
```

state和props的区别：

1. state: 是组件内部的状态数据,state的数据不能直接修改,如果需要修改需要通过特殊方法setState()进行修改.
2. props:一般用于组件之间通信,并且通常是父组件向子组件传递数据,通过this.props来接收.this.props的数据不能直接修改,如果需要修改,则需要通过父组件中的setState来进行修改.

## 生命周期

生命周期:指对象从创建,到销毁的过程,就是一个生命周期
钩子函数:在某时某刻自动被调用的函数就是钩子函数

- 挂载期

```
//构造函数. 作用: 在初始化时做state 数据的赋值工作
constructor(props){
    super(props)
    console.log('==constructor==');
    // console.log(this.props);
```

```
        this.state = {
            name:'贾玲'
        }
    }
    //从props中获取stats数据
    static getDerivedStateFromProps(props,state){
        console.log('==getDerivedStateFromProps==');
        // 必须return 一个对象或者null
        console.log(props);
        console.log(state);
        return state
    }
    //作用:做jsx模板的解析,
    render() {
        console.log('==render==');
        //报错
        // console.log(document.querySelector('h1').innerHTML);

        const {age} = this.props
        return (
            <div className="alert alert-info">
                <h1>生命周期</h1>
                <div>age:{age}</div>
            </div>
        )
    }
     //作用: 定时器 延时器 轮播图 等等各种请求
    componentDidMount(){
        console.log('==componentDidMount==');
        console.log(document.querySelector('h1').innerHTML);

    }
```

+ 更新期:当props或着state的数据发生改变的时候,处于更新期

```
  // getDerivedStateFromProps
  //应该更新组件吗?
  /**
   * 必须return一个结果 如果是true,表示更新数据  如果是false表示不更数据
   * 应用:在实际开发中,有一些特殊的业务逻辑在更新完数据之后不希望用户看到,这是该钩子函数就可以实现
  */
  shouldComponentUpdate(){
      console.log('==shouldComponentUpdate==');
      return true
  }
  // render...
  // 组件更新完成
  componentDidUpdate(prevProps,prevState){
      console.log('==componentDidUpdate==');
      //获取的是更新之前的props
      console.log(prevProps);
      // 获取更新之前的state
      console.log(prevState);
  }
```

- 销毁期

```
//销毁期
componentWillUnmount(){
    console.log('==componentWillUnmount==');
}
```

## 表单元素处理

在react中没有提供表单的数据双向绑定,在html中input textarea select等等,自己本身有各自的状态,可进行随时修改.
受控组件: 状态受react组件的控制
非受控组件:状态不收react组件的控制

```
<!-- 输入框 -->
<input type="text" value={this.state.name} onChange=
{(e)=>this.setState({name:e.target.value})}  />
<!-- 文本域 -->
<textarea value={this.state.description} onChange=
{(e)=>this.setState({description:e.target.value})}  cols="30" rows="10">
</textarea>
<!-- 单选 -->
<input type="radio"  value="1" checked={this.state.sex==='1'} onChange=
{(e)=>this.setState({sex:e.target.value})} />男
<input type="radio"  value="2" checked={this.state.sex==='2'} onChange=
{(e)=>this.setState({sex:e.target.value})} />女
<!-- 下拉框 -->
<select value={this.state.val4} onChange={this.changeVal4}>
  <option value="school">学校</option>
  <option value="home">家</option>
  <option value="company">公司</option>
</select>
```

多选框:

```
<label>杭州</label>
<input type="checkbox" name="city" value="hangzhou" checked=
{this.state.val3.includes("hangzhou")} onChange={this.changeVal3} />
<label>苏州</label>
<input type="checkbox" name="city" value="suzhou" checked=
{this.state.val3.includes("suzhou")} onChange={this.changeVal3} />
<label>泉州</label>
<input type="checkbox" name="city" value="quanzhou" checked=
{this.state.val3.includes("quanzhou")} onChange={this.changeVal3} />
<!-- js代码如下:  -->
changeVal3(e) {
  if (this.state.val3.includes(e.target.value)) {
    let index = this.state.val3.indexOf(e.target.value)
    // 数组中包含此项，又选择了这一项，就表示要删除
    this.state.val3.splice(index, 1)
  } else {
    this.state.val3.push(e.target.value)
  }
  let arr = this.state.val3;
  this.setState({
    val3: arr
  })
}
```

非受控组件

```
render() {
  return (
    <div className="alert alert-info">
      <h1>非受控组件</h1>
      <div>
          <input type="text" ref={this.input} />
      </div>
      <div>
          <button onClick={()=>this.submit()}>提交</button>
      </div>
    </div>
  )
}
submit(){
    console.log(this.input.value);
}
```

## ref属性：查找节点，DOM元素或者组件,类似vue的ref属性，用来直接操作dom，一般不用

```
constructor(){
    super()
    this.input = React.createRef()
}
render() {
    return (
        <div className="alert alert-info">
            <h1>ref属性</h1>
            <Child ref={this.input} />
        </div>
    )
}
componentDidMount(){
    // console.log(this.input.current);
    this.input.current.innerHTML='XXX'
}
```

## React的性能优化

1.减少操作dom的次数 2.减少涉及到dom的数量

- React.Fragment

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

export default class Box1 extends Component {
    render() {
        return (
            // 系统组件
            //无需向组件添加额外的DOM节点
            // <React.Fragment>
            //     <li>html</li>
            //     <li>css</li>
```

```
            //     <li>js</li>
            // </React.Fragment>

            // 简写形式
            <>
              <li>html</li>
              <li>css</li>
              <li>js</li>
            </>
        )
    }
}
```

- React.memo(无用的渲染)

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

// 没有React.memo
//  const Index = (props)=>{
//      console.log('数据更新了');
//         return (
//             <div>
//                 <h2>{props.msg}</h2>
//             </div>
//         )
// }

// 有React.memo
//作用:方式无用的渲染
const Index = React.memo((props)=>{
    console.log('数据更新了');
       return (
           <div>
               <h2>{props.msg}</h2>
           </div>
       )
})

export default class Box2 extends Component {
    constructor(){
        super()
        this.state = {
            msg:'哈哈哈'
        }
    }
    render() {
        const {msg} = this.state
        return (
            <div>
                <h1>React.memo</h1>
            </div>
        )
    }
}
```

- 纯组件

```
  export default class Child2Child extends PureComponent {
    code ...
    // 使用PureComponent之后，只有父组件传进来的值变化时，才会更新
  }
```

- 错误边界,避免某个组件报错后，整个项目不能运行

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

export default class ErrorBoundary extends Component {
  constructor() {
    super();
    this.state = {
      // 初始化没有错误
      hasError: false,
      error: null
    }
  }
  // 当错误发生而且被捕获时执行
  componentDidCatch(error) {
    this.setState({
      hasError: true,
      error: error
    })
  }
  render() {
    const {hasError, error} = this.state;
    return (
      <>
          {/* this.props.children是这个组件的子组件 */}
        {
          hasError?<mark>代码报错了...</mark>:this.props.children
        }
      </>
    )
  }
}

// 用这个组件将可能会报错的组件包裹起来
<ErrorBoundary>
  <Error></Error>
</ErrorBoundary>
```

## HOC高阶组件:HOC高级组件不是组件，是函数，这个函数的参数和返回值都是组件

```
  const Fn = (C) => {
  // 返回值也是一个组件
  return class extends Component {
    render() {
      return (
        <>
          {/* p是高阶组件加入的功能 */}
          <p>我是一个高级组件...</p>
```

```
          {/* C是组件原有的功能 */}
          <C {...this.props}></C>
        </>
      )
    }
  }
}

class Com1 extends Component {
  render() {
    return (
      <div>我是Com1</div>
    )
  }
}
class Com2 extends Component {
  render() {
    return (
      <div>我是Com2</div>
    )
  }
}

// 通过高阶组件强化已有的组件.Com1本身是组件，通过FN强化后，会返回一个新的组件，就是FnCom1
const FnCom1 = Fn(Com1);
const FnCom2 = Fn(Com2);

export default class Base extends Component {
  render() {
    return (
      <div>
        <p>以下是高阶组件强化后的新组件：</p>
        <FnCom1></FnCom1>
        <FnCom2></FnCom2>
      </div>
    )
  }
}
```

# 第十八天 过渡动画、路由和UI

## 过渡动画

1. 下载react-transition-group

```
cnpm i react-transition-group --save
```

2. CSSTransition实现单元素过渡动画
   (1) unmountOnExit:退出时实现卸载该组件，该属性必须添加
   (2) in:控制元素显示状态的 state 状态数据
   (3) timeout:过渡动画效果持续时间，单位为毫秒
   (4) classNames:指定过渡动画类名前缀(需要自定义两组四个过渡动画需要的 css 类选择器)
   (5) onEnter:元素进入前的回调函数，按需添加
   (6) onEntering:元素进入中的回调函数，按需添加
   (7) onEntered:元素进入后的回调函数，按需添加
   (8) onExit:元素离开前的回调函数，按需添加

(9) onExiting:元素离开中的回调函数，按需添加
(10) onExited:元素离开后的回调函数，按需添加

```
import React, { Component } from 'react'
import './box.css'
// 1.导出CSSTransition组件
import {CSSTransition} from 'react-transition-group'
export default class Box1 extends Component {
    constructor(){
        super()
        this.state = {
            isShow:true,//控制元素的显示和隐藏
            btnName:'隐藏'
        }
    }
    render() {
        return (
            <div>
                <CSSTransition
                    unmountOnExit
                    in={this.state.isShow}
                    timeout="500"
                    classNames="fade"
                    onEntered={()=>this.setState({btnName:'隐藏'})}
                    onExited={()=>this.setState({btnName:'显示'})}
                >
                    <h1>单元素动画效果</h1>
                </CSSTransition>


                <button onClick=
{()=>this.setState({isShow:!this.state.isShow})}>{this.state.btnName}</button>
            </div>
        )
    }
}
```

3. TransitionGroup实现列表元素动画效果

```
import React, { Component } from 'react'
import './box.css'
import {CSSTransition,TransitionGroup} from 'react-transition-group'
export default class Box2 extends Component {
    constructor(){
        super()
        this.state = {
            list:[
                {id:1,name:'vue'},
                {id:2,name:'react'},
                {id:3,name:'angular'},
            ],
            isShow:true,

        }
    }
    render() {
```

```
        const {list} = this.state
        return (
            <div>
                {/*
                TransitionGroup:需要将遍历的列表进行包裹
                CSSTransition:需要将遍历的每一项进行包裹
                 */}
                 <TransitionGroup>
                     {list.map(item=>(
                         <CSSTransition
                            key={item.id}
                            unmountOnExit
                            in={this.state.isShow}
                            timeout={500}
                            classNames="fade"
                         >
                            <div style={{display:'flex',justifyContent:'space-
between',borderBottom:'1px solid #999'}}>
                                <h3>{item.name}</h3>
                                <button onClick={()=>this.del(item.id)}>移除
</button>
                            </div>
                         </CSSTransition>
                     ))}
                 </TransitionGroup>
            </div>
        )
    }

    del(id){
       let list =  this.state.list.filter(item=>item.id!==id)
       this.setState({list})
    }
}
```

4. SwitchTransition实现切换效果

> 如果要实现让一个元素以过渡动画方式离开之后，再让另外一个元素以过渡动画方式进 入;或者新元素以过渡动画方式进入之后，再以过渡动画方式移除旧元素的切换动画效果， 就可以使用 SwitchTransition 组件来实现。

```
import React, { Component } from 'react'
import './box.css'
import {SwitchTransition,CSSTransition} from 'react-transition-group'
export default class Box3 extends Component {
    constructor(){
        super()
        this.state = {
            status:true,//状态
        }
    }
    render() {
        return (
            <div>
                <SwitchTransition>
                <CSSTransition
```

```
                    key={this.state.status ? "Goodbye, world!" : "Hello,
world!"}
                    addEndListener={(node, done) =>
node.addEventListener("transitionend", done, false)}
                    classNames='slide'>
                        <button onClick=
{()=>this.setState({status:!this.state.status})}>{this.state.status ? "Goodbye,
world!" : "Hello, world!"}</button>
                </CSSTransition>
                </SwitchTransition>
            </div>
        )
    }
}
```

## react-router-dom路由模块

> react-router-dom 是 react 官方提供的一个基于 react 的路由模块，但是 react-router-dom 是一个独立的模块，需要单独下载安装才能够使用。

1. 下载

```
cnpm i react-router-dom --save
```

2. 路由实现流程

- Switch

```
  {/* 路由出口 */}
  <Switch>

  </Switch>
```

- HashRouter：带有#
- BrowserRouter ： 常用的方式，不带#

3. 路由规则

- Route

  path:浏览器中访问的地址
  component:访问的组件
  exact:精准匹配，不能用在父路由上
  strict:严格模式 需要结合exact属性一起使用才能有效。检查地址最后有没有斜线

4. 404路由

> 当用户访问一个不存在的路由时,就会显示一个默认的页面,404页面.

```
{/* 404页面 */}
<Route path="/notfound" component={NotFound}></Route>
{/* 404页面的处理，当用户输入了错误地址时，前面的路由都没有匹配成功，就会重定向到一个固定的页面
*/}
<Redirect to="/notfound"></Redirect>
```

```
  <Route path="*" component={NotFound}></Route>
```

5. 路由导航

- Link

```
{/* 导航链接 */}
<Link to="/">首页</Link> |
<Link to="/about">关于我们</Link>
```

- NavLink

```
 <NavLink exact to="/" activeClassName="select">首页</NavLink> |
 <NavLink exact to="/about" activeClassName="select">关于我们</NavLink>
```

- NavLink
  - activeClassName:选中的样式
  - activeStyle:选中的样式
  - exact:精准匹配
  - to: 必填项 to:要跳转的路由
- 路由组件和非路由组件
  包裹在Route组件里的是路由组件，可以使用路由相关的熟悉和方法。没有包裹在Route组件里的是非路由组件，比如App这个组件。需要通过withRouter这个高阶组件，转换为路由组件使用。

6. 编程式导航

```
  // this.props.history.push(url); // push是在历史记录新增一条记录
  // this.props.history.replace(url); // replace是用新的记录替换老的记录
  // this.props.history.go(1); // go里面的参数是非零的整数，大于零表示前进，小于零表示后退
  // this.props.history.goBack(); // 后退一页
```

7. search传参

- URLSearchParams

```
// URLSearchParams传参，不用安装其他内容
let search = new URLSearchParams(this.props.location.search)
console.log(search.get('id'));
console.log(search.get('title'));
```

- queryString.parse

```
 // 方式二,需要安装querystringify
 let params = querystringify.parse(this.props.location.search.slice(1))
```

8. 动态路由

```
修改路由规则
 <Route exact strict path="/goodsDetail/:id" component={NewsDetail}></Route>

 获取参数
 console.log(this.props.match.params);
```

9. 路由懒加载

步骤:

- React.lazy()

```
// 懒加载模式
const Home = React.lazy(()=>import('./pages/Route/Home'))
```

- React.Supense(fallback)

```
index.js
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
  <React.Suspense fallback={<h1>loading...</h1>}>
       <App />
  </React.Suspense>


,
  document.getElementById('root')
);
```

10. 路由嵌套

```
render() {
  return (
    <div>
      <p>我是一个大的页面，包含了一些会切换的子页面</p>
      <NavLink to="/layout/small1">small1</NavLink>
      <NavLink to="/layout/small2">small2</NavLink>
      <NavLink to="/layout/small3">small3</NavLink>
      <br />
      <Button type="primary">Primary Button</Button>
      <Button>Default Button</Button>
      <Switch>
        {/* 路由组件中要写完整的路径，不能只写子路由 */}
        {/* 不要在父路由中写exact，要放在子路由中 */}
        <Route exact path="/layout/small1" component={Small1}></Route>
        <Route exact path="/layout/small2" component={Small2}></Route>
        <Route exact path="/layout/small3" component={Small3}></Route>
      </Switch>
    </div>
  )
```

## antd

1. 下载

```
cnpm i antd --save
```

2. 使用

```
入口文件index.css
import 'antd/dist/antd.css';
```

```
import React, { Component } from 'react'
import { Button } from 'antd';
export default class Index extends Component {
  render() {
    return (
        <div>
            <Button type="primary">Primary</Button>
            <Button>Default</Button>
        </div>
    )
  }
}
```

# react第四天 fetch和redux

## axios数据交互

1. 安装axios

```
cnpm i axios --save
```

2.发起http请求

```
  this.axios.get('/api/menulist').then(res=>{
      console.log(res);
  })

  this.axios.post('/api/menuedit', {pid: '345sdfj'}).then(res=>{
      console.log(res);
  })
```

3. 配置代理之手动配置

- 安装http-proxy-middleware

```
cnpm i http-proxy-middleware --save
```

- 创建setupProxy.js文件

在src下创建文件【setupProxy.js】js

```
const { createProxyMiddleware }=require("http-proxy-middleware");
module.exports=(app)=>{
    app.use(createProxyMiddleware('/api', {
        target: 'http://localhost:3000',/*这里写自己的代理地址*/
        changeOrigin: true,//允许跨域
    }));
}
npm start  #重启项目
```

## fetch

> fetch号称是AJAX的替代品，是在ES6出现的，使用了ES6中的promise对象。Fetch是基于promise设计的。Fetch的代码结构比起ajax简单多了，参数有点 像jQuery ajax。但是，一定记住fetch不是ajax的进一步封装，而是原生js，没有使 用XMLHttpRequest对象。fetch有很多优势:
>
> 1语法简单，更加语义化。
>
> 2基于 标准Promise实现，支持async/await。
>
> 3提供了丰富的API
>
> 4脱离了XHR。fetch是 一种HTTP数据请求的方式，在react中，数据交互仍然使用axios

## 2-1.fetch之get请求

```
fetch('/api/menulist',{
  headers:{
      'Content-Type':'application/x-www-form-urlencoded'
  }
}).then(res=>{
    return res.json();
}).then(res=>{
    console.log(res);
})
```

## 2-2.fetch之post请求

```
fetch('/api/menuedit',{
    method:'POST',//post请求
    headers:{ //请求头,不需要是可以不写使用默认值
        'Content-Type':'application/x-www-form-urlencoded'
    },
    //提交的数据
    body:{
      xx:xx,
      yy:yy
    }
}).then(res=>res.json())
.then(res=>{
    console.log(res);
})
```

## 2-3.fetch之封装

```
// 封装fetch  get post
// 1.引入qs模块，也可以是JSON.stringify方法
import qs from 'qs'

// 封装get
/**
 * 参数一:url  请求的地址
*/
export function myGet(url){
    return fetch(url).then(res=>res.json())
```

```
}

//封装post
/**
 * 参数一:url:  请求的地址
 * 参数二:data:  提交的数据
*/
export function myPost(url,data){
    return fetch(url,{
        method:'post',
        body:qs.stringify(data)
    }).then(res=>res.json())
}
```

例子:

get请求

```
import {myGet, myPost} from '../myFetch'
    myGet('/api/menulist').then(res=>{
        console.log(res);
    })

    myPost('/api/roleadd',{xx:xx}).then(res=>{
        console.log(res);
    })
```

## redux

### 3-1.什么是redux

```
1.Redux 是 JavaScript 状态容器，提供可预测化的状态管理。
2.redux构建一致化的应用，运行于不同的环境（客户端、服务器、原生应用），并且易于测试。
```

### 3-2.为什么使用redux

```
1.React 只是 DOM 的一个抽象层，并不是 Web 应用的完整解决方案
2."只有遇到 React 实在解决不了的问题，你才需要 Redux 。
3.redux使用于多交互,多数据源
```

### 3-3.redux的三大原则

- 单一数据源

```
整个应用的 state 被储存在一棵 object tree 中，并且这个 object tree 只存在于唯一一个 store 中。
```

- state是只读的

```
唯一改变 state 的方法就是触发 action，action 是一个用于描述已发生事件的普通对象。
eg:
action = {type:'',data:''}
```

- 使用纯函数进行修改
  纯函数的要求：函数的返回结果只依赖于它的参数、函数执行过程里面没有副作用

为了描述 action 如何改变 state tree ，你需要编写 reducers。

## 4.使用redux

### 4-1.安装

```
cnpm i redux --save
```

### 4-2.创建仓库store

- store

Store 就是保存数据的地方，你可以把它看成一个容器。整个应用只能有一个 Store。

```
// 1.引入redux
import {createStore} from 'redux'

// 创建store
/**
 * 参数:reducer  纯函数
*/
const store = createStore(reducer)


export default store
```

### 4-3.state

- state

Store对象包含所有数据。如果想得到某个时点的数据，就要对 Store 生成快照。这种时点的数据集合，就叫做 State。

```
// 2.创建state
const initialState = {
    age: 18
    num: 0
}
```

### 4-4.action

- 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 View 发出的通知，表示 State 应该要发生变化了。

Action 是一个对象。其中的type属性是必须的，表示 Action 的名称。

```
 const action = {type:'addNum'}
 store.dispatch(action)
```

#### 4-5.reducer

- reducer

```
Store 收到 Action 以后，必须给出一个新的 State，这样 View 才会发生变化。这种 State 的计算过程就叫做 Reducer。

Reducer 是一个函数，它接受 Action 和当前 State 作为参数，返回一个新的 State。
```

```
// 声明一个reducer函数
/**
 * 参数一:state 只上一次更新的最新数据
 * 参数二:action: 是一个json对象, 接受dispatch中的action
 *       action = {type:'',[data:'']}
*/
const reducer = (state=initialState,action)=>{
    switch (action.type) {
        case 'changeName':
            return {
               ...state,
               name:action.name
            }
        default:
            return state
    }
}
```

## 5.store常用的方法

#### 5-1.getState

- 获取store中状态数据的唯一方法

```
 const {name,age} = store.getState()
```

#### 5-2dispatch

- 将action作为参数触发纯函数的唯一方法

```
const action = {type:'addNum'}
store.dispatch(action)
```

#### 5-3.subscribe

- 订阅store中的状态数据

```
componentDidMount(){
        store.subscribe(()=>{
            this.setState(store.getState())
        })
    }
```

#### 5-4.取消监听

```
// 组件挂载完成,监听订阅
    componentDidMount(){
        this.un = store.subscribe(()=>{
            this.setState(store.getState())
        })
    }
    componentWillUnmount(){
        // 取消订阅
        this.un()
    }
```

# react第五天

## Action Creator：action创建函数是将action作为一个函数进行封装,为了在store和view中进行方便调用

```
  actionCreator = {
    // changeName是一个函数，是actionCreator的一个方法.
    // 调用这个方法，就会返回一个对象，这个对象就是action
    changeName: (name) => ({ type: CHANGENAME, name: name }),
    changeAge: (age) => ({ type: CHANGEAGE, age: age }),
  }
```

## ActionTypes:用来管理整个项目所有的action的type

```
  export const CHANGEAGE = 'changeAge';
  export const CHANGENAME = 'changeName';
```

```
  import {CHANGEAGE, CHANGENAME} from './type'
  // 修改数据的纯函数：reducer
  const reducer = (state=initState, action)=>{
  // action就是一个js的obj，一定会有一个属性叫type：{type: xxx, name: 'Jack'}
  switch (action.type) {
    case CHANGENAME:
      // 返回一个对象，作为新的state
      return {
        ...state,
        name: action.name
      }
    case CHANGEAGE:
      return {
        ...state,
        age: action.age
      }

    default:
```

```
      return state
  }
}
```

```
import {CHANGEAGE, CHANGENAME} from '../store/type'
actionCreator = {
  // changeName是一个函数，是actionCreator的一个方法.
  // 调用这个方法，就会返回一个对象，这个对象就是action
  changeName: (name) => ({ type: CHANGENAME, name: name }),
  changeAge: (age) => ({ type: CHANGEAGE, age: age }),
}
```

将type.js中的变量导入到各个组件以及store中，保证整个项目的每个action的type属性都使用这里的变量，提高项目的可维护性。

## reducer的拆分与合并

> 通过modules拆分不同模块的代码，再通过combineReducers方法将他们联合，最终创建store

```
// 导入不同模块中的内容
import orderReducer from './modules/order'
import userReducer from './modules/user'

// 通过combineReducers方法，合并两个模块中的reducer
let rootReducer = combineReducers({
  order: orderReducer,
  user: userReducer,
})
const store = createStore(rootReducer);

export default store
```

其中，info模块代码如下，user模块类似，详见代码部分。

```
// 初始化数据
const initState = {
  // 订单列表
  info: {}
}

// 统一处理action的type
const type = {
  CHANGEINFO: 'changeInfo'
}

// 统一处理action，这里会生成所有的action
const actions = {
  changeList: (info) => ({type: type.CHANGEINFO, info})
}

const reducer = (state = initState, action)=> {
  switch (action.type) {
    case type.CHANGEINFO:
      return {
        ...state,
        info: action.info
```

```
        }
      default:
        return state
    }
  }

  export default reducer;
```

## redux devTools调试工具：类似vue-devtools，可以在浏览器中看到当前数据状态等信息

1. 打开chrome->扩展程序->选择下载好的devTool工具(地址：https://github.com/zalmoxisus/redux-devtools-extension/releases，选择firefox即可)
2. 安装依赖包
   cnpm i redux-devtools-extension --save-dev
3. 在代码中使用

```
import {composeWithDevTools} from 'redux-devtools-extension'
const store = createStore(rootReducer,composeWithDevTools(applyMiddleware()))
export default store
```

## react-redux

> react-redux是redux的官方react绑定库,即为react准备的比较特殊的redux版本。它能够使你的React组件从redux store中读取数据，并且向store分发actions以更新数据 作用:为了方便组件关联状态

1. 安装
   cnpm i react-redux --save
2. 在src/index.js中使用

```
// 全局引入store，方便全局使用
import store from './store'
// 引入Provider组件，将store放在全局使用
import { Provider } from 'react-redux'
ReactDOM.render(
  // 使用Provider组件，包着其他内容，store就可以全局使用了.
  // 通过父传子的形式，将store的数据传给App
  <Provider store={store}>
    {/* 使用BrowserRouter组件包裹着App组件 */}
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </Provider>,
  document.getElementById('root')
);
```

3. 在组件中使用
   react-redux 提供connect方法，用于从 UI 组件生成容器组件。react-redux将组件分成两类：容器型组件（有数据和逻辑）、UI组件（没有数据和逻辑，只是呈现内容）

```
  let ReactRedux = connect(mapStateToProps, mapDispatchToProps)(ReactReduxUI)
  // ReactReduxUI是ui组件，通过connect()使他变成了容器型组件（ReactRedux），暴露容器性组件出去
  export default ReactRedux
```

4. 容器型组件VS展示型组件

| | 容器型组件 | 展示型组件 |
|---|---|---|
| 关注点 | 逻辑[取数据,更新数据] | UI的展现 |
| 对redux是否感知 | 是 | 否 |
| 读数据 | 从redux的store 中获取 | 从props中获取 |
| 写数据 | 发送redux actions | 调用props的回调 |
| 如何创建 | 通过react-redux connect创建 | 手写 |

5. mapStateToProps类似vuex中的mapState和mapGetters，将数据映射到组件中，用来获取数据
6. mapDispatchToProps类似vuex中的mapAction和mapMutation，将修改数据的方法映射到组件中，用来触发数据的修改

## redux高阶

1. bindActionCreators：将单个或多个ActionCreator转化为dispatch(action)的函数集合形式，统一返回

```
// 导入bindActionCreators，将各个action一起导入
import {bindActionCreators} from 'redux'
const mapDispatchToProps = (dispatch) => ({
  changeInfo: (info) => dispatch(userAction.changeInfo(info)),
  // changeList: (list) => dispatch(orderAction.changeList(list)),
  // addList: (list) => dispatch(orderAction.addList(list)),
  // delList: (list) => dispatch(orderAction.delList(list)),
  // 将修改数据的方法一次性全部导入(订单模块的方法全部导入)
  orderAction: bindActionCreators(orderAction, dispatch)
})
```

## Redux中间件

> Redux middleware 提供了一个分类处理 action 的机会。在 middleware 中，我们可以检阅每一个流过的 action,并挑选出特定类型的 action 进行相应操作，以此来改变 action。
>
> 1. 自己封装中间件
>
> ```
> // store/index.js
> // 1.封装中间件
> const logger = store=>next=>action=>{
>   console.log('调用dispatch的action',action);
>   let result = next(action)
>   console.log('调用dispatch后的state为',store.getState());
>   return result
> }
> const store =
> createStore(rootReducer,composeWithDevTools(applyMiddleware(logger)))
> ```

> 2. redux-logger中间件,使用action的中间件,一般放在参数的最后一位
>
> - 下载
>   cnpm i redux-logger --save
> - 引入
>   // 引入redux-logger
>   import {logger} from 'redux-logger'
> - 使用

```
store/index.js
const store =
createStore(rootReducer,composeWithDevTools(applyMiddleware(logger)))
export default store
```

## 项目目录设计

1. 按照类型划分

```
actions       #action creator
  action1.js
    action2.js
reducers  #reducer函数
  reducer1.js
    reducer2.js
containers    #容器型组件
  container1.jsx
    container2.jsx
components    #展示型组件
  components1.js
    components2.js
```

总结:如果需要修改一个功能,则需要修改好几个文件.

2. 按照功能设计

```
feature1        #功能1
    components  #展示型组件
        compoents1.jsx
    index.j     #容器型组件
    reducer.js #reducer函数
    actrions.js #action creator
feature2        #功能2
    components  #展示型组件
        compoents2.jsx
    index.j     #容器型组件
    reducer.js #reducer函数
   actrions.js #action creator
```

总结:如果按照功能设计来来分.数据会存在重复使用,会造成代码冗余的效果

3. ducks设计模式
   ducks模式将reducer,action Type和actions绑定到同一个目录中.导致减少样板.

```
src
    -components      #公共组件
    pages
```

```
        Order
            index.js    #容器型组件
            components
                Order.jsx    #展示型组件
        User
            index.js    #容器型组件
            components
                User.jsx    #展示型组件
    store
        index.js    #store状态对象
        modules
            Order.js    #子模块:state, action Types action Creator, reducer,
action selector
            User.js     #子模块:state, action Types action Creator, reducer,
action selector
```

# react第六天

## hooks

1. hooks介绍

- 什么是hooks

```
不编写类的情况下使用 state(状态) 和其他 React 功能,hooks是在react16.8新增功能
```

- 主要解决的问题

```
1.用于在函数组件中引入状态管理和生命周期方法.

2.取代高阶组件和render props来实现抽象和可重用性.

3.完全脱离'类',便可写出一个全功能的组件.
```

- 优点

```
1.避免在被广泛使用的函数组件在后期迭代过程,需要承担一些副作用,而必须重构成类组件.

2.帮助函数组件引入状态管理和生命周期方法;

3.hooks出现之后,我们将复用逻辑取到组件顶层,而不是强行提升到父组件中.这样就能够避免HOC和render
props带来的嵌套地域.
```

- react中钩子的作用及常用的钩子

```
从外部引入对象的钩子函数,来做响应的功能.
```

常用的钩子;

```
useState():为函数组件引入状态
useEffect():副作用钩子
useReducer():action钩子
useContext():共享状态钩子
```

2. useState

本身是一个函数,来自react包,参数和返回值,返回一个数组.
作用:状态钩子,为函数组件内提供状态

```
pages/useState/Index.js
import React,{useState} from 'react'

export default ()=>{
    /**
     * 参数:接受一个初始值
     * 返回值:返回一个数组,[val,setval]
    */
    // console.log(useState(0));
    const [count,setCount] = useState(0)
    const [food,setFood] = useState('')
    return (
        <div>
            <h1>useState</h1>
            <div>你点击屏幕了:{count}次</div>
            <button onClick={()=>setCount(count+1)}>add</button>

        </div>
    )
}
```

3. useEffect

作用:useEffect用于处理组件中的effect，通常用于请求数据，事件处理，订阅等 相关操作
语法:他接收两个参数。参数一:进行的异步操作 参数二:是数组，用来给出Effect的 依赖项

类使用组件实现count的更新

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

export default class Effect extends Component {
    state = {
        count:0
    }
    componentDidMount(){
        document.title = `你点击屏幕了${this.state.count}次`
    }
    componentDidUpdate(){
        document.title = `你点击屏幕了${this.state.count}次`
    }

    render() {
        return (
            <div>
                <div>你点击屏幕了:{this.state.count}次</div>
                <button onClick={()=>this.add()}>add</button>
            </div>
        )
    }
```

```
    add(){
        this.setState({count:this.state.count+1})
    }
}
```

useEffect Hook 看做 componentDidMount，componentDidUpdate 和 componentWillUnmount 这三个函数的组合。
1).只有一个回调函数作为参数，相当于 componentDidMount componentDidUpdate
2).第二个参数是[],相当于componentDidMount
3).第二个参数是[count],相当于componentDidMount 和count变化，才执行
4).return:相当于componentWillUnmount

(1).只有一个回调函数作为参数，相当于 componentDidMount + componentDidUpdate

```
import React,{useEffect,useState} from 'react'


export default ()=>{

    const [count,setCount] = useState(0)
    // 参数一: 使用回调函数作为参数,相当于componentDidMount和componentDidUpdate
    useEffect(()=>{
        document.title = `你点击屏幕了${count}次`
    })
    return (
        <div>
            <h1>useEffect</h1>
            <div>你点击屏幕了:{count}次</div>
            <button onClick={()=>setCount(count+1)}>add</button>
        </div>
    )
}
```

(2)第二个参数是[],相当于componentDidMount

```
import React,{useEffect,useState} from 'react'

export default ()=>{
    const [count, setCount] = useState(0)
    /**
     * 1.只有一个参数时,useEffect表示组件挂载完成和自减更新完成的钩子
     * 2.第二个参数[],useEffect表示组件挂载完成componentDidMount
     *
    */
    useEffect(()=>{
        document.title = `你点击屏幕了${count}次`
    },[])
    return (
        <div>
            <div>你点击屏幕了:{count}次</div>
            <button onClick={()=>setCount(count+1)}>add</button>
        </div>
    )
```

```
}
```

(3)第二个参数是[count],相当于componentDidMount 和count变化，才执行

```
import React,{useEffect,useState} from 'react'

export default ()=>{
    const [count, setCount] = useState(0)
    const [name,setName] = useState('于增超')
    /**
     * 1.只有一个参数时,useEffect表示组件挂载完成和自减更新完成的钩子
     * 2.第二个参数[],useEffect表示组件挂载完成componentDidMount
     * 3.第二个参数[count],useEffect表示组件挂载完成componentDidMount和count发生改变时,
才执行
    */
    useEffect(()=>{
        console.log(111);
        document.title = `你点击屏幕了${count}次`
    },[count])
    return (
        <div>
            <div>你点击屏幕了:{count}次</div>
            <button onClick={()=>setCount(count+1)}>add</button>
            <hr />
            <div>{name}</div>
            <button onClick={()=>setName('Tom')}>李玉斌</button>
        </div>
    )
}
```

(4)return:相当于componentWillUnmount

```
import React,{useEffect,useState} from 'react'


export default ()=>{
    const [date,setDate] = useState(new Date())

    useEffect(()=>{
        let timer = setInterval(() => {
            setDate(new Date())
        }, 1000);
        // return表示的是componentWillUnmount
        return ()=>{
                console.log('clear');
                clearInterval(timer)
        }
    },[])

    return (
        <div>
            <div>当前时间为:{date.toLocaleTimeString()}</div>
        </div>
    )
}
```

4. useReducer

```
作用:
(1)useReducer() 提供了状态管理
(2)基本原理是通过用户在页面中发起 action, 从而通过reducer方法来改变state, 从而实现页面和状态的通信
```

语法

```
const [state, dispatch] = useReducer(reducer, initState);
```

```
import React,{useReducer} from 'react'

// 初始值
const initialState =  {
    count:0,
}

// 创建reducer
const reducer = (state,action)=>{
    switch (action.type) {
        case 'add':
            return {
                ...state,
                count:action.count+1
            }
        case 'sub':
            return {
                ...state,
                count:action.count-1
            }

        default:
            throw new Error('没有action的type',action.type)
    }
}

export default ()=>{
    const [state,dispatch] = useReducer(reducer,initialState)
    // console.log(useReducer(reducer,initialState));
    return (
        <div>
            <div>{state.count}</div>
            <button onClick=
{()=>dispatch({type:'add',count:state.count})}>add</button>
            <button onClick=
{()=>dispatch({type:'sub',count:state.count})}>sub</button>
        </div>
    )
}
```

5. useContext

作用:
1.在组件之间共享状态,
2.可以解决react逐层通过props传递
3.她接受React.createContext()的返回结果作为参数

场景:在Hooks中，如果父组件传递数据给子组件，子组件没有使用，向下传递，子 组件的子组件进行使用，如果通过Props传递，可以使用，但是如果层级比较多，就会不方 便，那么可以子组件的子组件可以通过useContext进行接收数据。

Context.jsx

```
import react,{createContext,useState} from 'react'
import Child from './Child'
// 创建myContext
export const myContext = createContext()
console.log(myContext);
export default ()=>{
    const [name,setName] = useState('于增超')
    return (
        <div className="alert alert-info">
            <h1>useContext</h1>
            <myContext.Provider value={name}>
                <Child></Child>
            </myContext.Provider>
        </div>
    )
}
```

Child.jsx

```
import React from 'react'
import Child1 from './Child1'
export default ()=>{
    return (
        <div className="well">
            <h1>子组件</h1>
            <Child1></Child1>
        </div>
    )
}
```

Child1.jsx

```
import React,{useContext} from 'react'
import {myContext} from './Context'
export default ()=>{
    return (
        <div className="alert alert-info">
            <h1>子组件的子组件</h1>
            {useContext(myContext)}
        </div>
    )
}
```

4. 自定义hook

> 构建自己的 Hooks 可以将组件逻辑提取到可重用的函数中。
> 当我们想要在两个 JavaScript 函数之间共享逻辑时，我们会将共享逻辑提取到第三个函数。 组件和 Hook 都是函数，所以这种办法也适用于它们!
> 自定义 Hook 是一个 JavaScript 函数，其名称以 ”use” 开头，可以调用其他 Hook。

Login.jsx

```
import React,{useState} from 'react'

export default ()=>{
    const [name,setName] = useState('')
    const [pass,setPass] = useState('')
    const login = ()=>{
        console.log(name);
        console.log(pass);
    }
    return (
        <div>
            <h1>登录页面</h1>
            <div>用户名
                <input type="text" value={name} onChange=
{(e)=>setName(e.target.value)} />
            </div>
            <div>密码
                <input type="password" value={pass} onChange=
{(e)=>setPass(e.target.value)}  />
            </div>
            <div>
                <button onClick={()=>login()} >登录</button>
            </div>
        </div>
    )
}
```

自定义的hook

```
import React,{useState} from 'react'
// 自定义hook
const useInputVal = ()=>{
    const [val,setVal] = useState()
    return {
        val: val,
        onChange:(e)=>setVal(e.target.value)
    }
}


export default ()=>{
    // console.log(useInputVal());/
    const {reset:name1,...setName} = useInputVal()
    const {reset:pass1,...setPass} = useInputVal()
    const login = ()=>{
        console.log(name1);
        console.log(pass1);
```

```
    }

    return (
        <div>
            <h1>登录页面</h1>
            <div>用户名
                <input type="text" {...setPass} />
            </div>
            <div>密码
                <input type="password" {...setPass}  />
            </div>
            <div>
                <button onClick={()=>login()} >登录</button>
            </div>
        </div>
    )
}
```

5. hooks规则

1.只在顶层使用hook

2.只在react函数中调用hook,不要在普通javascript函数中调用hook