# React

## React入门

### React基础

路由库

React-Router

消息管理库

PubSub

集中式管理的库

Redux

精美的UI组件库

Ant-Design

用于构建用户界面的JavaScript库

1. 发送请求获取数据
2. 处理数据(过滤、整理格式等)
3. 操作DOM呈现页面

React是一个将数据渲染为HTML视图的开源JavaScript库

FaceBook开发，且开源

1. 原生JavaScript操作DOM繁琐、效率低(DOM-API操作UI)
2. 使用JavaScript直接操作DOM，浏览器会进行大量的重绘重排
3. 原生JavaScript没有组件化编码方案，代码复用率低

模块化拆JS

组件化拆html\css\js及构成这个局部功能的图片字体音视频

React的特点

1. 采用组件化模式、声明式编码(对比命令式编码)，提高开发效率及组件复用率
2. 在React Native中可以使用React语法进行移动端开发
3. 使用虚拟DOM+优秀的Diffing算法，尽量减少与真实DOM交互

React高效的原因

使用虚拟(virtual)DOM, 不总是直接操作页面真实DOM。

DOM Diffing算法, 最小化页面重绘。

babel.min.js作用：ES6转ES5 jsx转js

react.development.js核心库

react-dom.development.js扩展库

### hello\_react案例

<!-- 准备好一个“容器” -->

<!-- 引入react核心库 -->

<!-- 引入react-dom，用于支持react操作DOM -->

<!-- 引入babel，用于将jsx转为js -->

  <script *type*="text/babel" > /\* 此处一定要写babel \*/

    //1.创建虚拟DOM

    const VDOM = <h1>Hello,React</h1> /\* 此处一定不要写引号，因为不是字符串 \*/

//2.渲染虚拟DOM到页面 虚拟DOM，容器

//把虚拟DOM渲染到容器里面

    ReactDOM.render(VDOM,document.getElementById('test'))

  </script>

同一个容器不能被渲染两次

### 虚拟DOM的两种创建方式

1. 纯JS方式(一般不用)
2. JSX方式

<script *type*="text/babel" > /\* 此处一定要写babel \*/

//1.创建虚拟DOM

    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>

    )

不用jsx，不用导入babal

语法：React.createElement(标签名,标签属性，标签内容)

<script *type*="text/javascript" >

//1.创建虚拟DOM

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

    //2.渲染虚拟DOM到页面

### 虚拟DOM与真实DOM

 /\*

        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。

     \*/



### jsx语法规则

全称: JavaScript XML

react定义的一种类似于XML的JS扩展语法: JS + XML

XML早期用于存储和传输数据，后来没有了，用JSON

XML:<student><name>Tom</name><age>19</age></student>

JSON:”{“name”:”Tom”,”age”:19}”

JSON:parse(将JSON字符串快速转换成JS对象)

JSON:stringfy(将JS对象快速转换成JSON字符串)

本质是React.createElement(component, props, ...children)方法的语法糖

基本语法规则

/\*

        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。

     \*/

  <div>

        <h2 *className*="title" *id*={myId.toLowerCase()}>

          <span *style*={{color:'white',fontSize:'29px'}}>{myData.toLowerCase()}</span>

        </h2>

        <h2 *className*="title" *id*={myId.toUpperCase()}>

          <span *style*={{color:'white',fontSize:'29px'}}>{myData.toLowerCase()}</span>

        </h2>

        <input *type*="text"/>

      </div>

### jsx练习

//模拟一些数据

    const data = ['Angular','React','Vue']

    //1.创建虚拟DOM

    const VDOM = (

      <div>

        <h1>前端js框架列表</h1>

        <ul>

          {

            data.map((*item*,*index*)=>{

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

            })//key是虚拟DOM的唯一标识，一般使用id

          }

        </ul>

      </div>

    )

花括号里面是变量

 /\*

      一定注意区分：【js语句(代码)】与【js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，可以被赋值

                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}

   \*/

### 组件与模块

模块

理解：向外提供特定功能的js程序, 一般就是一个js文件

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。

作用：复用js, 简化js的编写, 提高js运行效率

组件

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)

为什么要用组件： 一个界面的功能更复杂

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

## React面向组件编程

### 使用React开发工具调式

在chrome中扩展程序中



加载已经解压的扩展程序

导入D:\vs code html\尚硅谷2021版React技术全家桶全套完整版（零基础入门到精通 男神天禹老师亲授）\react全家桶资料\06\_其他\react\_dev\_tools



点击针管图标加载到浏览器菜单栏上

打开jsx或美团官网开发工具会变颜色

### 函数式组件

函数式组件：用函数定义的组件(适用于【简单组件】的定义)

 //1.创建函数式组件

    function MyComponent(){

      console.log(*this*); //此处的this是undefined，因为babel编译后开启了严格模式

      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'))

    /\*

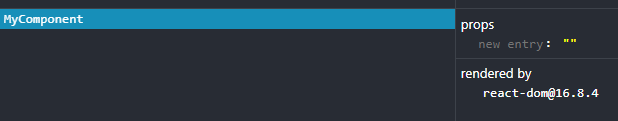
      执行了ReactDOM.render(<MyComponent/>.......之后，发生了什么？

          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。

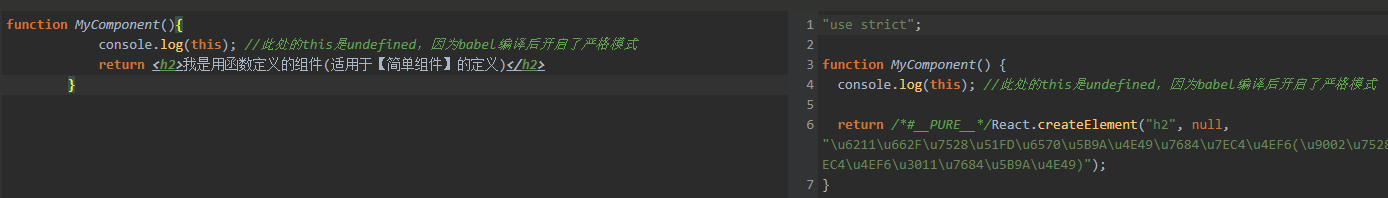
    \*/

Shift+点击刷新，强制刷新



严格模式：禁止自定义函数里面的this指向window

babel官网https://www.babeljs.cn/点击试一试



### 类的概念

<script *type*="text/javascript" >

    /\*

      总结：

        1.类中的构造器不是必须要写的，要对实例进行一些初始化的操作，如添加指定属性时才写。

        2.如果A类继承了B类，且A类中写了构造器，那么A类构造器中的super是必须要调用的。

        3.类中所定义的方法，都放在了类的原型对象上，供实例去使用。

    \*/ //创建一个Person类

    class Person {

      //构造器方法

      constructor(*name*,*age*){

        //构造器中的this是谁？—— 类的实例对象p1,p2

*this*.name = *name*

*this*.age = *age*

      }

      //一般方法

      speak(){

        //speak方法放在了哪里？——类的原型对象上，供实例使用

        //通过Person实例调用speak时，speak中的this就是Person实例

        console.log(`我叫${*this*.name}，我年龄是${*this*.age}`);

      }

}

}

//创建一个Person的实例对象

const p1=new Person(“tom”,18)

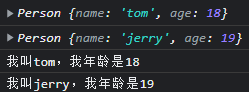
const p2=new Person(“jerry”,19)

console.log(p1)

console.log(p2)

p1.speak()

p2.speak()



 //创建一个Student类，继承于Person类

      class Student extends Person {

        constructor(*name*, *age*, *grade*) {

*super*(*name*, *age*);

*this*.grade = *grade*;

*this*.school = "尚硅谷";

        }

        //重写从父类继承过来的方法

        speak() {

          console.log(

            `我叫${*this*.name}，我年龄是${*this*.age},我读的是${*this*.grade}年级`

          );

*this*.study();

        }

        study() {

          //study方法放在了哪里？——类的原型对象上，供实例使用

          //通过Student实例调用study时，study中的this就是Student实例

          console.log("我很努力的学习");

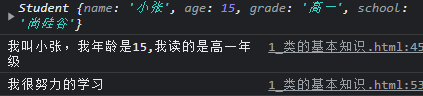
        }

      }

const s1 = **new** Student("小张", 15, "高一");

      console.log(s1);

      s1.speak();



class Car {

        constructor(*name*, *price*) {

*this*.name = *name*;

*this*.price = *price*;

          // this.wheel = 4

        }

        //类中可以直接写赋值语句,如下代码的含义是：给Car的实例对象添加一个属性，名为a，值为1

        a = 1;

        wheel = 4;

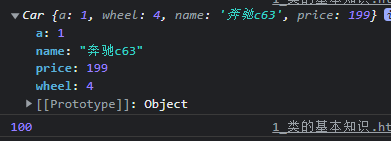
        static demo = 100;

      }

      const c1 = **new** Car("奔驰c63", 199);

      console.log(c1);

      console.log(Car.demo);



### 类式组件

类式组件：用类定义的组件(适用于【复杂组件】的定义

 //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'))

    /\*

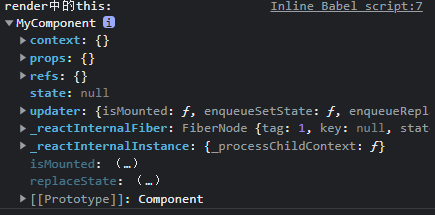
      执行了ReactDOM.render(<MyComponent/>.......之后，发生了什么？

          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。

    \*/



注意

组件名必须首字母大写

虚拟DOM元素只能有一个根元素

虚拟DOM元素必须有结束标签

渲染类组件标签的基本流程

React内部会创建组件实例对象

调用render()得到虚拟DOM, 并解析为真实DOM

插入到指定的页面元素内部

## State的基本使用

人 状态 影响 行为

组件 状态 驱动 页面

state:null null是一个空的对象

  //构造器调用几次？ ———— 1次

      constructor(*props*){

        console.log('constructor');

*super*(*props*)

        //初始化状态

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

        //解决changeWeather中this指向问题

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

      }

### 事件绑定

<button *id*="btn1">按钮1</button>

    <button *id*="btn2">按钮2</button>

    <button *onclick*="demo()">按钮3</button>

    <script *type*="text/javascript" >

      const btn1 = document.getElementById('btn1')

      btn1.addEventListener('click',()=>{

        alert('按钮1被点击了')

      })

      const btn2 = document.getElementById('btn2')

      btn2.onclick = ()=>{

        alert('按钮2被点击了')

      }

      function demo(){

        alert('按钮3被点击了')

      }

一、call,apply,bind的相同点：

都是改变this指向的；

第一个参数都是this要指向的对象；

都可以利用后续参数传参；

二、call,apply,bind的区别：

call和bind的参数是依次传参，一一对应的；

但apply只有两个参数，第二个参数为数组；

call和apply都是对函数进行直接调用，而bind方法返回的仍是一个函数；

### 类中方法中的this

非严格模式下this指向window，严格模式下this指向undefined

  Function changeWeather(){

Console.log(this) //window

}

  Function changeWeather(){

“use strict”

Console.log(this) //undefined

}

class Person {

        constructor(*name*,*age*){

*this*.name = *name*

*this*.age = *age*

        }

        study(){

          //study方法放在了哪里？——类的原型对象上，供实例使用

          //通过Person实例调用study时，study中的this就是Person实例

          console.log(*this*);//undefined

        }

      }

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

      const p1 = **new** Person('tom',18)

      p1.study() //通过实例调用study方法

      const x = p1.study

      x()

### state

  //解决changeWeather中this指向问题

        this.dom= this.changeWeather.bind(this)

将changeWeather中的this绑定到Weather中的实例对象上赋值给dom属性

bind:生成新的函数、改变函数的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(){

        console.log('render');

        //读取状态

        const {isHot,wind} = *this*.state

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

      }

//changeWeather调用几次？ ———— 点几次调几次

      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

//通过Weather实例调用changeWeather时，changeWeather中的this就是Weather实例

       //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

        console.log('changeWeather');

        //获取原来的isHot值

        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，要借助一个内置API去更改。

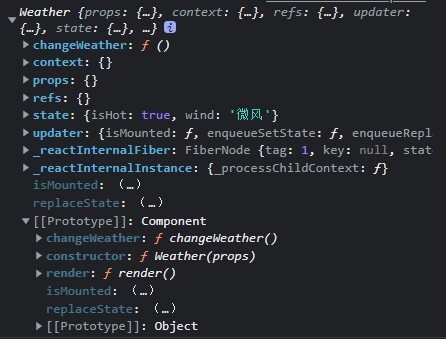
*this*.setState({isHot:!isHot})

        console.log(*this*);

        //严重注意：状态(state)不可直接更改，下面这行就是直接更改！！！

        //this.state.isHot = !isHot //这是错误的写法

      }



原型和构造函数自身都有changeWeather方法，优先使用构造函数自身的方法，通过原型上的方法生成一个新的方法挂载到实例，实例再进行调用

构造器里面：

继承，初始化状态，改变this指向问题

render：读取状态，渲染页面

方法：获取原来的值，更新状态

### state简写方式

   class Car {

        constructor(*name*, *price*) {

*this*.name = *name*;

*this*.price = *price*;

          // this.wheel = 4

        }

        //类中可以直接写赋值语句,如下代码的含义是：给Car的实例对象添加一个属性，名为a，值为1

        a = 1;

        wheel = 4;

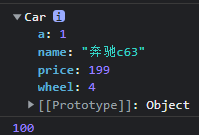
        static demo = 100;

      }

      const c1 = **new** Car("奔驰c63", 199);

      console.log(c1);

      console.log(Car.demo);



可以不写在构造器里面

changeWeather(){}在Weather的原型对象上

  //初始化状态

      state = {isHot:false,wind:'微风'}

//自定义方法————要用赋值语句的形式+箭头函数

//放在了Weather所缔造的实例自身

      changeWeather =funcrion(){

        const isHot = *this*.state.isHot

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

      }

普通函数的话Weather的原型对象里面没有changeWeather，放到了Weather所缔造的实例自身

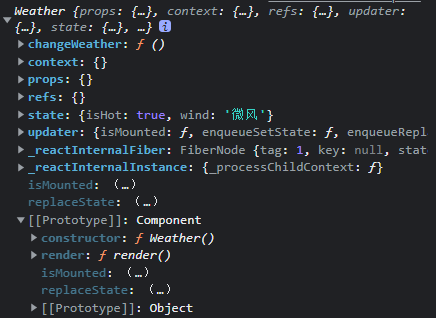
箭头函数没有自己的this，将外层函数的this作为箭头函数的this使用,this是Weather的实例对象

changeWeather = ()=>{

        const isHot = *this*.state.isHot

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

      }



2.2.2. 理解

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)

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

组件中render方法中的this为组件实例对象

组件自定义的方法中this为undefined，如何解决？

a.强制绑定this: 通过函数对象的bind()

b.箭头函数

状态数据，不能直接修改或更新

## Props的基本使用

### Props

   render(){

        // console.log(this);

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

        return (

          <ul>

            <li>姓名：{name}</li>

            <li>性别：{sex}</li>

            <li>年龄：{age+1}</li>

          </ul>

        )

      }

//渲染组件到页面

ReactDOM.render(<*Person* *name*="jerry"

*sex*="男"/>,document.getElementById('test1'))

const p = {name:'老刘',age:18,sex:'女'}

    // console.log('@',...p);

    // ReactDOM.render(<Person name={p.name} age={p.age} sex={p.sex}/>,document.getElementById('test3'))

ReactDOM.render(<*Person* {...p}/>,document.getElementById('test3'))

在react和babel中...p可以展开对象，仅仅使用于标签属性的传递,批量传递标签属性(props)

展开运算符

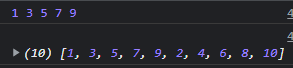
let arr1 = [1, 3, 5, 7, 9];

      let arr2 = [2, 4, 6, 8, 10];

      console.log(...arr1); //展开一个数组

      let arr3 = [...arr1, ...arr2]; //连接数组

      console.log(arr3);



  function sum1(...*numbers*) {

        console.log("@", *numbers*);

      }

      console.log(sum1(1, 2, 3, 4));



 //在函数中使用

      function sum(...*numbers*) {

        return *numbers*.reduce((*preValue*, *currentValue*) => {

          return *preValue* + *currentValue*;

        });

      }

      console.log(sum(1, 2, 3, 4));//10

     //构造字面量对象时使用展开语法

      let person = { name: "tom", age: 18 };

      let person2 = { ...person };//克隆后的对象

      //console.log(...person); //报错，展开运算符不能展开对象

  person.name = "jerry";

      console.log(person2);

      console.log(person);



//合并

      let person3 = { ...person, name: "jack", address: "地球" };

      console.log(person3);



### 对props进行限制

将ReactDOM.render 里面的age=”19”改成age={19}就可以进行

<li>年龄：{age+1}</li>年龄加一

//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

    }

    <!-- 引入prop-types，用于对组件标签属性进行限制 -->

    <script *type*="text/javascript" *src*="../js/prop-types.js"></script>

导入了PropTypes属性

//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

      };

默认情况下ReactDOM.render里面sex和age可以不传

 //渲染组件到页面

      ReactDOM.render(

        <*Person* *name*="jack" *speak*={speak} />,

        document.getElementById("test1")

      );

  function speak() {

        console.log("我说话了");

      }

### props的简写方式

//props是只读的

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

将Person.propTypes和Person.defaultProps放到Person类里面

Person.demo在类Person外面用static demo写到Person里面

在propTypes和defaultProps前面加上static写到Person里面

### 类式组件中的构造器与props

  constructor(*props*){

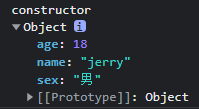
        //构造器是否接收props，是否传递给super，取决于：是否希望在构造器中通过this访问props

        // console.log(props);

*super*(*props*)

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

      }



### 函数式组件使用props

//创建组件

    function Person (*props*){

      const {name,age,sex} = *props*

}

函数没有static，只能把 Person.propTypes、 Person.defaultProps 写到函数外，可以操作props，函数可以传参，通过hook操作state、refs

## Ref的基本使用

### 字符串形式的ref

  //展示左侧输入框的数据

        showData = () => {

          const { input1 } = *this*.refs;

          alert(input1.value);

        };

 <input *ref*="input1" *type*="text" *placeholder*="点击按钮提示数据" />

组件内的标签可以定义ref属性来标识自己

refs拿到的是虚拟DOM转换成真实DOM后的节点，真正的节点，字符串形式的ref，请勿过度用refs

### 回调形式的ref

箭头函数const trim = () => trim.replace

转换成普通函数const trim=function(){return trim.replace}

 <input *ref*={*c* => *this*.input1 = *c* } *type*="text" *placeholder*="点击按钮提示数据"/>

c就是input这个节点

  //展示左侧输入框的数据

      showData = ()=>{

        const {input1} = *this*

        alert(input1.value)

      }

内联函数的形式

 <input

*ref*={(*c*) => {

*this*.input1 = *c*;

                  console.log("@", *c*);

                }}

*type*="text"

              />

函数式的ref第一次渲染时候就会调用函数，就把当前函数传进来



更新的时候，render会执行一次，不确定之前c节点ref函数接到了什么，做了什么动作，就将c节点ref函数清空，传进去null，第二次执行时候才会把之前的c节点ref函数传进去



注释符号

 {/\* \*/}

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

类绑定的回调的形式

saveInput = (*c*) => {

*this*.input1 = *c*;

          console.log("@", *c*);

        };

回调函数已经存放到实例自身了，就不会频繁的调用，更新的时候只会渲染第一次，第一次渲染之后更新就不会执行了

### createRef的使用

  /\*

        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()

 <input *ref*={*this*.myRef} *type*="text" *placeholder*="点击按钮提示数据"/>

就把当前ref所在的节点直接存储到myRef容器里面

//展示左侧输入框的数据

        showData = () => {

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

          alert(*this*.myRef.current.value);

        };

this.myRef



this.myRef.input得到input节点

## 事件处理

### react中的事件处理

通过onXxx属性指定事件处理函数(注意大小写)

React使用的是自定义(合成)事件, 而不是使用的原生DOM事件 ——为了更好的兼容性

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) ——为了高效

通过event.target得到发生事件的DOM元素对象 ——不要过度的使用ref

  //展示右侧输入框的数据

      showData2 = (*event*)=>{

        alert(*event*.target.value);

      }

//event.target获取input节点

<input *onBlur*={*this*.showData2} *type*="text" *placeholder*="失去焦点提示数据"/>

## 组件

### 非受控组件

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

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

   </form>

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}`)

      }

ajax页面无刷新交互

现用现取

### 受控组件

onChange勾选框由未勾选变成勾选，只要发生改变就会调用该函数回调

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

//初始化状态

      state = {

        username:'', //用户名

        password:'' //密码

      }

      //保存用户名到状态中

      saveUsername = (*event*)=>{

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

      }

存进去的是值，取出来的是值，

受控组件：输入的dome维护到状态里面，等需要用的时候，从状态里面取出来

类似于vue里面的双向数据绑定

推荐受控组件，非受控组件有多个ref回调

### 函数柯里化

 //#region

        /\*

          高阶函数：如果一个函数符合下面2个规范中的任何一个，那该函数就是高阶函数。

                  1.若A函数，接收的参数是一个函数，那么A就可以称之为高阶函数。

                  2.若A函数，调用的返回值依然是一个函数，那么A就可以称之为高阶函数。

                  常见的高阶函数有：Promise、setTimeout、arr.map()等等

          函数的柯里化：通过函数调用继续返回函数的方式，实现多次接收参数最后统一处理的函数编码形式。

            function sum(a){

              return(b)=>{

                return (c)=>{

                  return a+b+c

                }

              }

            }

const result = sum(1)(2)(3)

      console.log(result);

          \*/

    //#endregion

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

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

  //保存表单数据到状态中

      saveFormData = (*dataType*)=>{

        return (*event*)=>{

*this*.setState({[*dataType*]:*event*.target.value})

        }

      }

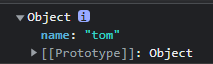
参数变量用[]

  let a = 'name'

      let obj = {} // {name:'tom'}

      obj[a] = 'tom'

      console.log(obj);



### 不用柯里化的写法

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

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

      //保存表单数据到状态中

      saveFormData = (*dataType*,*event*)=>{

*this*.setState({[*dataType*]:*event*.target.value})

      }

## 生命周期(旧)依赖包v16.8.4

### 引出生命周期

  death = ()=>{

        //卸载组件

        ReactDOM.unmountComponentAtNode(document.getElementById('test'))

      }

<button *onClick*={*this*.death}>不活了</button>

  state = {opacity:1}

//组件挂完毕

      componentDidMount(){

        console.log('componentDidMount');

*this*.timer = setInterval(() => {

          //获取原状态

          let {opacity} = *this*.state

          //减小0.1

          opacity -= 0.1

          if(opacity <= 0) opacity = 1

          //设置新的透明度

*this*.setState({opacity})

        }, 200);

      }

和render是同一个级别的，通过类的实例对象调用，不用赋值语句和箭头函数

  //组件将要卸载

      componentWillUnmount(){

        //清除定时器

        clearInterval(*this*.timer)

      }

render初始化渲染、状态更新之后

 //生命周期回调函数 <=> 生命周期钩子函数 <=>

//生命周期函数 <=> 生命周期钩子

### 组件挂载流程

组件从创建到死亡它会经历一些特定的阶段。

React组件中包含一系列勾子函数(生命周期回调函数), 会在特定的时刻调用。

我们在定义组件时，会在特定的生命周期回调函数中，做特定的工作。



挂载时

  //构造器

      constructor(*props*){

        console.log('Count---constructor');

*super*(*props*)

        //初始化状态

*this*.state = {count:0}

      }

  //组件将要挂载的钩子

      componentWillMount(){

        console.log('Count---componentWillMount');

      }

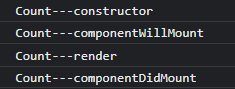
      //组件挂载完毕的钩子

      componentDidMount(){

        console.log('Count---componentDidMount');

      }

渲染中



### setState流程

卸载时

  //卸载组件按钮的回调

      death = ()=>{

        ReactDOM.unmountComponentAtNode(document.getElementById('test'))

      }

  //组件将要卸载的钩子

      componentWillUnmount(){

        console.log('Count---componentWillUnmount');

      }

 <button *onClick*={*this*.death}>卸载组件</button>



setState()

//加1按钮的回调

      add = ()=>{

        //获取原状态

        const {count} = *this*.state

        //更新状态

*this*.setState({count:count+1})

      }

属性是否应该被更新，不写默认为true

   //控制组件更新的“阀门”

      shouldComponentUpdate(){

        console.log('Count---shouldComponentUpdate');

        return true

      }

  //组件将要更新的钩子

      componentWillUpdate(){

        console.log('Count---componentWillUpdate');

      }

渲染中

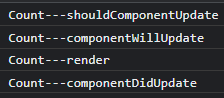
  //组件更新完毕的钩子

      componentDidUpdate(){

        console.log('Count---componentDidUpdate');

      }

 <button *onClick*={*this*.add}>点我+1</button>



### forceUpdate流程

  //控制组件更新的“阀门”

      shouldComponentUpdate(){

        console.log('Count---shouldComponentUpdate');

        return false

      }

阀门关闭也能更新

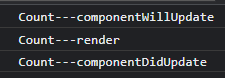
//强制更新按钮的回调

      force = ()=>{

*this*.forceUpdate()

      }

 <button *onClick*={*this*.force}>不更改任何状态中的数据，强制更新一下</button>



### 父组件render流程

A组件里面的

 //初始化状态

        state = { carName: "奔驰" };

changeCar = () => {

*this*.setState({ carName: "奥拓" });

        };

写在实例对象里面不需要加this，也不需要解构对象

写在构造器里面需要加this，引用也需要解构对象

render里面的

<div>

              <div>我是A组件</div>

              <button *onClick*={*this*.changeCar}>换车</button>

              <*B* *carName*={*this*.state.carName} />

            </div>

B组件里面的

 render() {

          console.log("B---render");

          return <div>我是B组件，接收到的车是:{*this*.props.carName}</div>;

        }

 //组件将要接收新的props的钩子

        componentWillReceiveProps(*props*) {

          console.log("B---componentWillReceiveProps", *props*);

        }

第一次传的奔驰不算，第二次更新后调用render的奥拓才算

//控制组件更新的“阀门”

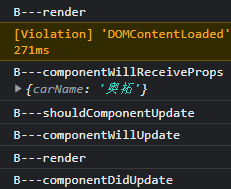
        shouldComponentUpdate()

 //组件将要更新的钩子

        componentWillUpdate()

 //组件更新完毕的钩子

        componentDidUpdate()



### 总结

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() =====> 常用

                          一般在这个钩子中做一些初始化的事，例如：开启定时器、发送网络请求、订阅消息

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

                  1.  shouldComponentUpdate()

                  2.  componentWillUpdate()

                  3.  render() =====> 必须使用的一个

                  4.  componentDidUpdate()

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

                  1.  componentWillUnmount()  =====> 常用

                          一般在这个钩子中做一些收尾的事，例如：关闭定时器、取消订阅消息

    \*/

## 生命周期(新)依赖包v17.0.1

对比新旧生命周期

在官方搜索CDN，在百度打开链接，保存到文件即可

<https://www.bootcdn.cn/>

点击babel-core 复制链接在百度打开，保存到文件即可



componentWillMount和componentWillUpdate和componentWillReceiveProps要加UNSAFE\_前缀

这里的UNSAFE\_前缀不是指安全性，而是表示使用这些生命周期的代码在React的未来版本中更有可能出现bug，尤其是在启用异步渲染之后。

React官网 文档 核心概念 State&生命周期 详细的组件API参考文档

v17.0.1废弃了componentWillMount和componentWillUpdate和componentWillReceiveProps三个钩子，添加了getDerivedStateFromProps

和  getSnapshotBeforeUpdate的钩子。

### getDerivedStateFromProps

Derived导出的，派生的

  constructor(*props*)

  //若state的值在任何时候都取决于props，那么可以使用getDerivedStateFromProps

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

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

        return null

      }

渲染中render

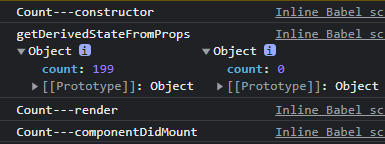
  componentDidMount()

 //渲染组件

    ReactDOM.render(<*Count* *count*={199}/>,document.getElementById('test'))

即state的值在任何时候都取决于props

派生状态会导致代码冗余，并使组件难以维护，使用场景极其罕见



### getSnapshotBeforeUpdate

Snapshot快照

  //在更新之前获取快照

      getSnapshotBeforeUpdate(){

        console.log('getSnapshotBeforeUpdate');

        return 'atguigu'

      }

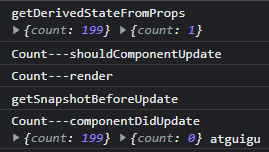
  //组件更新完毕的钩子

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

        console.log('Count---componentDidUpdate',*preProps*,*preState*,*snapshotValue*);

      }

此用法并不常用



### getSnapshotBeforeUpdate案例

 return(

          <div *className*="list" *ref*="list">

            {

*this*.state.newsArr.map((*n*,*index*)=>{

                return <div *key*={*index*} *className*="news">{*n*}</div>

              })

            }

          </div>

        )

  state = {newsArr:[]}

      componentDidMount(){

        setInterval(() => {

          //获取原状态

          const {newsArr} = *this*.state

          //模拟一条新闻

          const news = '新闻'+ (newsArr.length+1)

          //更新状态

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

        }, 1000);

      }

//滚动之前的屏幕高度

  getSnapshotBeforeUpdate(){

        return *this*.refs.list.scrollHeight

      }

//滚动之后顶部向上滚动的高度

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

*this*.refs.list.scrollTop += *this*.refs.list.scrollHeight - *height*

      }

### 总结

  /\*

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

                1.  constructor()

                2.  getDerivedStateFromProps

                3.  render()

                4.  componentDidMount() =====> 常用

                      一般在这个钩子中做一些初始化的事，例如：开启定时器、发送网络请求、订阅消息

        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()  =====> 常用

                      一般在这个钩子中做一些收尾的事，例如：关闭定时器、取消订阅消息

      \*/

重要的勾子

render：初始化渲染或更新渲染调用

componentDidMount：开启监听, 发送ajax请求

componentWillUnmount：做一些收尾工作, 如: 清理定时器

即将废弃的勾子

componentWillMount

componentWillReceiveProps

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。

## DOM的diffing算法

toTimeString() 方法：把 Date 对象的时间部分转换为字符串

逐层对比，最小的力度是标签(节点)

/\*

   经典面试题:

      1). react/vue中的key有什么作用？（key的内部原理是什么？）

      2). 为什么遍历列表时，key最好不要用index?

      1. 虚拟DOM中key的作用：

          1). 简单的说: key是虚拟DOM对象的标识, 在更新显示时key起着极其重要的作用。

          2). 详细的说: 当状态中的数据发生变化时，react会根据【新数据】生成【新的虚拟DOM】,

                        随后React进行【新虚拟DOM】与【旧虚拟DOM】的diff比较，比较规则如下：

                  a. 旧虚拟DOM中找到了与新虚拟DOM相同的key：

                        (1).若虚拟DOM中内容没变, 直接使用之前的真实DOM

                        (2).若虚拟DOM中内容变了, 则生成新的真实DOM，随后替换掉页面中之前的真实DOM

                  b. 旧虚拟DOM中未找到与新虚拟DOM相同的key

                        根据数据创建新的真实DOM，随后渲染到到页面

      2. 用index作为key可能会引发的问题：

                1. 若对数据进行：逆序添加、逆序删除等破坏顺序操作:

                        会产生没有必要的真实DOM更新 ==> 界面效果没问题, 但效率低。

                2. 如果结构中还包含输入类的DOM：

                        会产生错误DOM更新 ==> 界面有问题。

                3. 注意！如果不存在对数据的逆序添加、逆序删除等破坏顺序操作，

                  仅用于渲染列表用于展示，使用index作为key是没有问题的。

      3. 开发中如何选择key?:

                1.最好使用每条数据的唯一标识作为key, 比如id、手机号、身份证号、学号等唯一值。

                2.如果确定只是简单的展示数据，用index也是可以的。

   \*/

    /\*

    慢动作回放----使用index索引值作为key

      初始数据：

          {id:1,name:'小张',age:18},

          {id:2,name:'小李',age:19},

      初始的虚拟DOM：

          <li key=0>小张---18<input type="text"/></li>

          <li key=1>小李---19<input type="text"/></li>

      更新后的数据：

          {id:3,name:'小王',age:20},

          {id:1,name:'小张',age:18},

          {id:2,name:'小李',age:19},

      更新数据后的虚拟DOM：

          <li key=0>小王---20<input type="text"/></li>

          <li key=1>小张---18<input type="text"/></li>

          <li key=2>小李---19<input type="text"/></li>

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

  慢动作回放----使用id唯一标识作为key

      初始数据：

          {id:1,name:'小张',age:18},

          {id:2,name:'小李',age:19},

      初始的虚拟DOM：

          <li key=1>小张---18<input type="text"/></li>

          <li key=2>小李---19<input type="text"/></li>

      更新后的数据：

          {id:3,name:'小王',age:20},

          {id:1,name:'小张',age:18},

          {id:2,name:'小李',age:19},

      更新数据后的虚拟DOM：

          <li key=3>小王---20<input type="text"/></li>

          <li key=1>小张---18<input type="text"/></li>

          <li key=2>小李---19<input type="text"/></li>

   \*/

*this*.state.persons.map((*personObj*,*index*)=>{

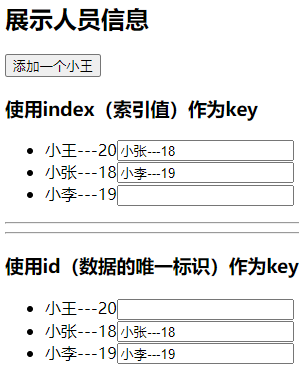
                return <li *key*={*index*}>{*personObj*.name}---{*personObj*.age}<input *type*="text"/></li>

              })

*this*.state.persons.map((*personObj*)=>{

                return <li *key*={*personObj*.id}>{*personObj*.name}---{*personObj*.age}<input *type*="text"/></li>

              })



## 脚手架

### 初始化react脚手架

使用create-react-app创建react应用

react脚手架

xxx脚手架: 用来帮助程序员快速创建一个基于xxx库的模板项目

包含了所有需要的配置（语法检查、jsx编译、devServer…）

下载好了所有相关的依赖

可以直接运行一个简单效果

react提供了一个用于创建react项目的脚手架库: create-react-app

项目的整体技术架构为: react + webpack + es6 + eslint

使用脚手架开发的项目的特点: 模块化, 组件化, 工程化

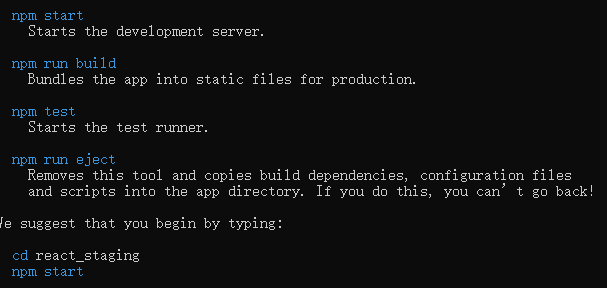
创建项目并启动

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



### yarn

yarn的安装:

下载node.js，使用npm安装

npm install -g yarn

查看版本：

yarn --version

安装node.js,下载yarn的安装程序:

提供一个.msi文件，在运行时将引导您在Windows上安装Yarn

Yarn 淘宝源安装，分别复制粘贴以下代码行到黑窗口运行即可

yarn config set registry https://registry.npm.taobao.org -g

yarn config set sass\_binary\_site http://cdn.npm.taobao.org/dist/node-sass -g

yarn的常用命令：

安装yarn

npm install -g yarn

安装成功后，查看版本号：

yarn --version

创建文件夹 yarn

md yarn

进入yarn文件夹

cd yarn

初始化项目

yarn init // 同npm init，执行输入信息后，会生成package.json文件

yarn的配置项：

yarn config list // 显示所有配置项

yarn config get <key> //显示某配置项

yarn config delete <key> //删除某配置项

yarn config set <key> <value> [-g|--global] //设置配置项

安装包：

yarn install //安装package.json里所有包，并将包及它的所有依赖项保存进yarn.lock

yarn install --flat //安装一个包的单一版本

yarn install --force //强制重新下载所有包

yarn install --production //只安装dependencies里的包

yarn install --no-lockfile //不读取或生成yarn.lock

yarn install --pure-lockfile //不生成yarn.lock

添加包（会更新package.json和yarn.lock）：

yarn add [package] // 在当前的项目中添加一个依赖包，会自动更新到package.json和yarn.lock文件中

yarn add [package]@[version] // 安装指定版本，这里指的是主要版本，如果需要精确到小版本，使用-E参数

yarn add [package]@[tag] // 安装某个tag（比如beta,next或者latest）

//不指定依赖类型默认安装到dependencies里，你也可以指定依赖类型：

yarn add --dev/-D // 加到 devDependencies

yarn add --peer/-P // 加到 peerDependencies

yarn add --optional/-O // 加到 optionalDependencies

//默认安装包的主要版本里的最新版本，下面两个命令可以指定版本：

yarn add --exact/-E // 安装包的精确版本。例如yarn add foo@1.2.3会接受1.9.1版，但是yarn add foo@1.2.3 --exact只会接受1.2.3版

yarn add --tilde/-T // 安装包的次要版本里的最新版。例如yarn add foo@1.2.3 --tilde会接受1.2.9，但不接受1.3.0

发布包

yarn publish

移除一个包

yarn remove <packageName>：移除一个包，会自动更新package.json和yarn.lock

更新一个依赖

yarn upgrade 用于更新包到基于规范范围的最新版本

运行脚本

yarn run 用来执行在 package.json 中 scripts 属性下定义的脚本

显示某个包的信息

yarn info <packageName> 可以用来查看某个模块的最新版本信息

缓存

yarn cache

yarn cache list # 列出已缓存的每个包

yarn cache dir # 返回 全局缓存位置

yarn cache clean # 清除缓存

### 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

%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*="#000000" />

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" />

若浏览器不支持js则展示标签中的内容

 <noscript>You need to enable JavaScript to run this app.</noscript>





### 文件介绍src

index.js入口文件

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

  <*React.StrictMode*>

    <*App* />

  </*React.StrictMode*>

);

reportWebVitals();

reportWebVitals.js用于记录页面中的性能，渲染出来的性能

index.js到index.html到App.js

### 一个简单的Hello文件

index.js文件

引入react核心库

引入ReactDOM

引入App组件

渲染App到页面

App.js文件

创建外壳组件App

创建并暴露App组件

实现页面局部功能代码和资源的集合叫组件

样式的模块化

将index.css改成index.module.css，就可以使用from接住

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

 return <h2 *className*={hello.title}>Hello,React!</h2>

### react插件的安装

安装插件ES7+ React/Redux/React-Native

rcc react\class\component react中的类式组件

import React, { Component } from 'react'

export default class index extends Component {

  render() {

    return (

      <div>index</div>

    )

  }

}

rfc react\function\component react中的函式组件

import React from 'react'

export default function index() {

  return (

    <div>index</div>

  )

}

imp引入

## TodoList案例

### 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

动态显示初始化数据

1 .数据类型

2 .数据名称

3.保存在哪个组件?

交互(从绑定事件监听开始)

3.2. 组件的组合使用-TodoList

功能: 组件化实现此功能

1. 显示所有todo列表

2. 输入文本, 点击按钮显示到列表的首位, 并清除输入的文本

分为App、Header、List、Item、footer五个组件

找到静态页面

D:\vs code html\尚硅谷2021版React技术全家桶全套完整版（零基础入门到精通 男神天禹老师亲授）\react全家桶资料\04\_静态页面\todos\_page

### 动态初始化列表

在App.jsx中

//状态在哪里，操作状态的方法就在哪里

  //初始化状态

  state = {todos:[

{id:'001',name:'吃饭',done:true},

]}

const {todos} = *this*.state

  <*List* *todos*={todos} *updateTodo*={*this*.updateTodo} *deleteTodo*={*this*.deleteTodo}/>

在List文件index.jsx

const {todos,updateTodo,deleteTodo} = *this*.props

 {

          todos.map( *todo* =>{

            return <*Item* *key*={*todo*.id} {...todo} *updateTodo*={updateTodo} *deleteTodo*={deleteTodo}/>

          })

        }

{...todo}可以批量传id、name、done

  <input *type*="checkbox" *checked*={done} *onChange*={*this*.handleCheck(id)}/>

checked默认打勾不可以取消(如果取消要写onchange)

defaultChecked打勾可以取消

### 添加todo

Header文件index.jsx

 <input *onKeyUp*={*this*.handleKeyUp} *type*="text" *placeholder*="请输入你的任务名称，按回车键确认"/>

  //键盘事件的回调

  handleKeyUp = (*event*)=>{

//解构赋值获取keyCode,target

//keyCode代表键盘的ASCALL码键盘标识

    const {keyCode,target} = *event*

    //判断是否是回车按键

    if(keyCode !== 13) return

//添加的todo名字不能为空

//trim去空格

    if(target.value.trim() === ''){

      alert('输入不能为空')

      return

    }

    //准备好一个todo对象

    const todoObj = {id:nanoid(),name:target.value,done:false}

    //将todoObj传递给App

*this*.props.addTodo(todoObj)

    //清空输入

    target.value = ''

  }

在App.jsx

  //addTodo用于添加一个todo，接收的参数是todo对象

  addTodo = (*todoObj*)=>{

    //获取原todos

    const {todos} = *this*.state

    //追加一个todo

    const newTodos = [*todoObj*,...todos]

    //更新状态

*this*.setState({todos:newTodos})

  }

  <*Header* *addTodo*={*this*.addTodo}/>

在Header文件index.jsx

生成唯一id

npm i uuid

uuid库太大了

换成yarn add naoid

import {nanoid} from 'nanoid'

### 鼠标移入效果

在Item文件中的index.jsx

const {mouse} = *this*.state

  <li *style*={{backgroundColor:mouse ? '#ddd' : 'white'}} *onMouseEnter*={*this*.handleMouse(true)} *onMouseLeave*={*this*.handleMouse(false)}>

  state = {mouse:false} //标识鼠标移入、移出

  //鼠标移入、移出的回调

  handleMouse = (*flag*)=>{

    return ()=>{

*this*.setState({mouse:*flag*})

    }

  }

### 勾选和取消某一个todo

在Item文件中index.jsx

 const {id,name,done} = *this*.props

  <input *type*="checkbox" *checked*={done} *onChange*={*this*.handleCheck(id)}/>

  //勾选、取消勾选某一个todo的回调

  handleCheck = (*id*)=>{

return (*event*)=>{

//event.target.value得到的值为on或off

//event.target.checked得到的值为true或false，因为type值为checkbox

*this*.props.updateTodo(*id*,*event*.target.checked)

    }

  }

在App.jsx中

//状态在哪里，操作状态的方法就在哪里

  //updateTodo用于更新一个todo对象

  updateTodo = (*id*,*done*)=>{

    //获取状态中的todos

    const {todos} = *this*.state

    //匹配处理数据

    const newTodos = todos.map((*todoObj*)=>{

      if(*todoObj*.id === *id*) return {...*todoObj*,done}

      else return *todoObj*

    })

*this*.setState({todos:newTodos})

  }

  <*List* *todos*={todos} *updateTodo*={*this*.updateTodo} *deleteTodo*={*this*.deleteTodo}/>

在List文件中的index.jsx

 const {todos,updateTodo,deleteTodo} = *this*.props

  todos.map( *todo* =>{

            return <*Item* *key*={*todo*.id} {...todo} *updateTodo*={updateTodo} *deleteTodo*={deleteTodo}/>

          })

### 对props进行限制

在Header文件中的index.jsx

import PropTypes from 'prop-types'

在终端中安装这个库

yarn add prop-types

  //对接收的props进行：类型、必要性的限制

  static propTypes = {

    addTodo:PropTypes.func.isRequired

  }

同样给List文件中的index.jsx

//对接收的props进行：类型、必要性的限制

  static propTypes = {

    todos:PropTypes.array.isRequired,

    updateTodo:PropTypes.func.isRequired,

    deleteTodo:PropTypes.func.isRequired,

  }

### 删除一个todo

在Item文件中index.jsx

  //删除一个todo的回调

  handleDelete = (*id*)=>{

    if(window.confirm('确定删除吗？')){

*this*.props.deleteTodo(*id*)

    }

  }

//函数传参onClick不用高阶柯里化，直接返回一个函数

 <button *onClick*={()=> *this*.handleDelete(id) } *className*="btn btn-danger" *style*={{display:mouse?'block':'none'}}>删除</button>

在App.jsx

  //deleteTodo用于删除一个todo对象

  deleteTodo = (*id*)=>{

    //获取原来的todos

    const {todos} = *this*.state

    //删除指定id的todo对象

    const newTodos = todos.filter((*todoObj*)=>{

      return *todoObj*.id !== *id*

    })

    //更新状态

*this*.setState({todos:newTodos})

  }

### 实现底部功能

在Footer文档中的index.jsx

 //已完成的个数

    const doneCount = todos.reduce((*pre*,*todo*)=> *pre* + (*todo*.done ? 1 : 0),0)

    //总数

    const total = todos.length

箭头函数后面只有一句话可以省略return

  <input *type*="checkbox" *onChange*={*this*.handleCheckAll} *checked*={doneCount === total && total !== 0 ? true : false}/>

defaultChecked只会在第一次渲染时起作用

  //全选checkbox的回调

  handleCheckAll = (*event*)=>{

*this*.props.checkAllTodo(*event*.target.checked)

  }

在App.jsx中

  //checkAllTodo用于全选

  checkAllTodo = (*done*)=>{

    //获取原来的todos

    const {todos} = *this*.state

    //加工数据

    const newTodos = todos.map((*todoObj*)=>{

      return {...*todoObj*,done}

    })

    //更新状态

*this*.setState({todos:newTodos})

  }

在Footer文件的index.jsx

  //清除已完成任务的回调

  handleClearAllDone = ()=>{

*this*.props.clearAllDone()

  }

在App.jsx

  //clearAllDone用于清除所有已完成的

  clearAllDone = ()=>{

    //获取原来的todos

    const {todos} = *this*.state

    //过滤数据

const newTodos = todos.filter((*todoObj*)=>{

//todoObj.done===false 取反就是原来的false返回true，符合条件筛选到新数组中

      return !*todoObj*.done

    })

    //更新状态

*this*.setState({todos:newTodos})

  }

### 总结

**## 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

## Ajax

### 脚手架配置代理方法1

前置说明

React本身只关注于界面, 并不包含发送ajax请求的代码

前端应用需要通过ajax请求与后台进行交互(json数据)

react应用中需要集成第三方ajax库(或自己封装)

常用的ajax请求库

jQuery: 比较重, 如果需要另外引入不建议使用

axios: 轻量级, 建议使用

封装XmlHttpRequest对象的ajax

promise风格

可以用在浏览器端和node服务器端

终端yarn add axios

import axios from 'axios'

D:\vs code html\尚硅谷2021版React技术全家桶全套完整版（零基础入门到精通 男神天禹老师亲授）\react全家桶资料\06\_其他\测试代理服务器

node中的服务器代码

浏览器插件FeHelper(前端助手)

同源策略限制在浏览器里面 在中间代理人(服务器)里面不限制

在项目里面中的package.json

加入”proxy”:”http://localhost:5000”

 axios.get('http://localhost:3000/students').

将5000改成3000

服务器api接口为

app.listen(5000,(*err*)=>{

  if(!*err*) console.log('服务器1启动成功了,请求学生信息地址为：http://localhost:5000/students');

})

优先会请求public下的资源然后请求接口服务器的资源

3000资源没有的才会去请求5000资源的

1. 优点：配置简单，前端请求资源时可以不加任何前缀。

2. 缺点：不能配置多个代理。

3. 工作方式：上述方式配置代理，当请求了3000不存在的资源时，那么该请求会转发给5000 （优先匹配前端资源）

### 脚手架配置代理方法2

如果有多个服务器，端口不一样，那么package.json中的代理proxy配置就不适用了。

解决：创建setupProxy.js文件

不能用ES6语法，要用CommonJs语法

const proxy = require('http-proxy-middleware')

*module*.*exports* = function(*app*){

*app*.use(

    proxy('/api1',{ //遇见/api1前缀的请求，就会触发该代理配置

      target:'http://localhost:5000', //请求转发给谁

      changeOrigin:true,//控制服务器收到的请求头中Host的值

      pathRewrite:{'^/api1':''} //重写请求路径(必须)

    }),

    proxy('/api2',{

      target:'http://localhost:5001',

      changeOrigin:true,

      pathRewrite:{'^/api2':''}

    }),

  )

}

 axios.get('http://localhost:3000/api1/students')

带上api1前缀

在服务器1中

app.use((*request*,*response*,*next*)=>{

  console.log('请求来自于',*request*.get('Host'));

  next()

})

终端打印

请求来自于 localhost:5000

如果没有

changeOrigin:true,

终端打印

请求来自于 localhost:3000

如果没有

pathRewrite:{'^/api1':''}

走了代理，请求了5000资源，请求地址为

/api1/students

加了请求地址为 /students

1. 优点：可以配置多个代理，可以灵活的控制请求是否走代理。

2. 缺点：配置繁琐，前端请求资源时必须加前缀。

## Github搜索案例

### 静态组件

找到静态页面

D:\vs code html\尚硅谷2021版React技术全家桶全套完整版（零基础入门到精通 男神天禹老师亲授）\react全家桶资料\04\_静态页面\users\_page

引入第三方库

最好放到public里面，在public中创建css

在index.html中导入

 <link *rel*="stylesheet" *href*="/css/bootstrap.css">

组件有List组件，Search组件

在List组件中index.jsx

 <a *rel*="noreferrer" *href*={*userObj*.html\_url} *target*="\_blank">

a标签需要加上ref=”noreferrer”

  <img *alt*="head\_portrait" *src*={*userObj*.avatar\_url} *style*={{width:'100px'}}/>

img标签需要加上alt=”head\_portrait”

引入的bootstrap样式会干扰之前项目的样式

### axios发送请求

在Search文档下的index.jsx

import axios from 'axios'

  <input *ref*={*c* => *this*.keyWordElement = *c*}

//获取用户的输入(连续解构赋值+重命名)

    const {keyWordElement:{value:keyWord}} = *this*

请求地址: <https://api.github.com/search/users?q=xxxxxx>

<https://api.github.com/search/users?q=aiguigu>

//发送网络请求

    axios.get(`/api1/search/users?q=${keyWord}`).then

localhost:3000和api.github.com产生了跨域

D:\vs code html\尚硅谷2021版React技术全家桶全套完整版（零基础入门到精通 男神天禹老师亲授）\react全家桶资料\05\_所需服务器\server

就是github所学要用的服务器

cmd npm start



<http://localhost:5000/search/users?q=aiguigu>

先找5000资源，然后再找github

<http://localhost:5000/search/users2?q=aiguigu>

要是github没反应，5000资源也没有反应

这是一个模拟的数据，加载速度快

用3000端口去请求5000端口，形成了跨域，配置代理

//发送网络请求

    axios.get(`http://localhost:3000/api1/search/users?q=${keyWord}`).then

在3000端口请求3000端口资源，可以省略http://localhost:3000

### 展示数据

在App.jsx

  state = { //初始化状态

    users:[], //users初始值为数组

    isFirst:true, //是否为第一次打开页面

    isLoading:false,//标识是否处于加载中

    err:'',//存储请求相关的错误信息

  }

  //更新App的state

  updateAppState = (*stateObj*)=>{

*this*.setState(*stateObj*)

  }

<*Search* *updateAppState*={*this*.updateAppState}/>

在Search文件里index.jsx

//请求成功后通知App更新状态

*this*.props.updateAppState({isLoading:false,users:*response*.data.items})

在App.jsx

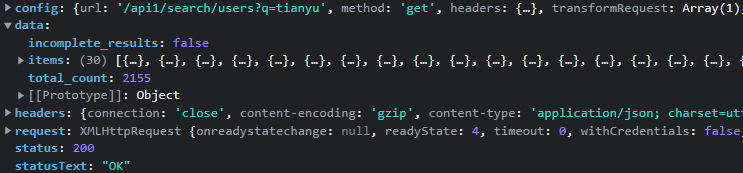
<*List* {...*this*.state}/>

在List的index.jsx

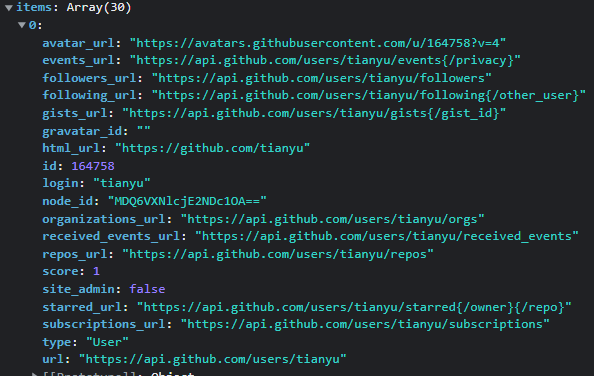
 const {users,isFirst,isLoading,err} = *this*.props

  users.map((*userObj*)=>{

console.log(response)



console.log(response.data.items)





将id，avatar\_url，html\_url，login分别填写

  <div *key*={*userObj*.id} *className*="card">

key写到最外侧

### 完成案例

List包括users、first、loading、err

在App.jsx

  state = { //初始化状态

    users:[], //users初始值为数组

    isFirst:true, //是否为第一次打开页面

    isLoading:false,//标识是否处于加载中

    err:'',//存储请求相关的错误信息

  }

<*List* {...*this*.state}/>

在Search的index.jsx

 //发送请求前通知App更新状态

*this*.props.updateAppState({isFirst:false,isLoading:true})

 //请求成功后通知App更新状态

*this*.props.updateAppState({isLoading:false,users:*response*.data.items})

 //请求失败后通知App更新状态

*this*.props.updateAppState({isLoading:false,err:*error*.message})

在List的index.jsx

   isFirst ? <h2>欢迎使用，输入关键字，随后点击搜索</h2> :

          isLoading ? <h2>Loading......</h2> :

          err ? <h2 *style*={{color:'red'}}>{err}</h2> :

## 消息订阅与发布技(兄弟组件)

### pubsub

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

在终端中安装yarn add pubsub-js

在List的index.jsx

import PubSub from 'pubsub-js'

  state = { //初始化状态

    users:[], //users初始值为数组

    isFirst:true, //是否为第一次打开页面

    isLoading:false,//标识是否处于加载中

    err:'',//存储请求相关的错误信息

  }

const {users,isFirst,isLoading,err} = *this*.state

App.jsx中

<*Search*/>

        <*List*/>

谁接收消息谁就订阅消息

谁发送消息谁就发布消息

在List的index.jsx

  componentDidMount(){

*this*.token = PubSub.subscribe('atguigu',(*\_*,*stateObj*)=>{

*this*.setState(*stateObj*)

    })

  }

\_占个位也就是确定数据来源

在Search的index.jsx

import PubSub from 'pubsub-js'

//发送请求前通知List更新状态

    PubSub.publish('atguigu',{isFirst:false,isLoading:true})

在List的index.jsx

  componentWillUnmount(){

    PubSub.unsubscribe(*this*.token)

  }

### fetch发送请求

 XMLHttpRequest(XHR)由第三方库jQuery，axios封装都需要下载，而fetch是内置的，和xhr是并列的，可以不借助xhr进行网络请求，不需要下载，有浏览器就能使用，也是promise风格，用的少

文档

<https://github.github.io/fetch/>

<https://segmentfault.com/a/1190000003810652>

特点

fetch: 原生函数，不再使用XmlHttpRequest对象提交ajax请求

老版本浏览器可能不支持

//#可以折叠//#

关注分离

    //发送网络请求---使用fetch发送（未优化）

    /\* fetch(`/api1/search/users2?q=${keyWord}`).then(

      response => {

        console.log('联系服务器成功了');

        return response.json()

      },

      error => {

        console.log('联系服务器失败了',error);

        return new Promise(()=>{})

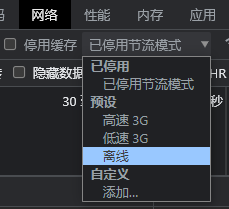
      }

    ).then(

      response => {console.log('获取数据成功了',response);},

      error => {console.log('获取数据失败了',error);}

    ) \*/



设置浏览器离线模式

输出联系服务器失败了

response.json()是原型里面的json方法，不是window里面内置的方法

返回的是一个Promise实例对象里面

返回失败保存在pending里面

运用了.then的链式调用

如果.then所指定的成功的返回值是一个promise实例对象，那么就把该实例对象作为外侧.then方法的返回值

如果.then所指定的失败的返回值是一个pending状态的promise实例对象，那么外侧的.then肯定也是一个pending状态的promise实例

如果.then里面成功后抛出异常，外侧的.then的promise实例返回的状态就为失败的，失败的原因呢就为你所抛出的异常

终端promise链，return new Promise()

//发送网络请求---使用fetch发送（优化）

    try {

      const response= await fetch(`/api1/search/users2?q=${keyWord}`)

      const data = await response.json()

      console.log(data);

      PubSub.publish('atguigu',{isLoading:false,users:data.items})

    } catch (error) {

      console.log('请求出错',error);

      PubSub.publish('atguigu',{isLoading:false,err:error.message})

    }

用catch进行统一错误信息的处理

  search = async()=>{

用两个await等到promise实例对象返回的值

await只能等到成功的结果，用try、catch处理异常的结果

使用率不高，兼容性差

谷歌浏览器

### 总结github搜索案例

**## 二、github搜索案例相关知识点**

    1.设计状态时要考虑全面，例如带有网络请求的组件，要考虑请求失败怎么办。

    2.ES6小知识点：解构赋值+重命名

          let obj = {a:{b:1}}

          const {a} = obj; //传统解构赋值

          const {a:{b}} = obj; //连续解构赋值

          const {a:{b:value}} = obj; //连续解构赋值+重命名

    3.消息订阅与发布机制

          1.先订阅，再发布（理解：有一种隔空对话的感觉）

          2.适用于任意组件间通信

          3.要在组件的componentWillUnmount中取消订阅

    4.fetch发送请求（关注分离的设计思想）

          try {

            const response= await fetch(`/api1/search/users2?q=${keyWord}`)

            const data = await response.json()

            console.log(data);

          } catch (error) {

            console.log('请求出错',error);

          }

## React路由

### 对SPA应用的理解

单页Web应用（single page web application，SPA）。

整个应用只有一个完整的页面。

点击页面中的链接不会刷新页面，只会做页面的局部更新。

数据都需要通过ajax请求获取, 并在前端异步展现。

单页面多组件

### 对路由的理解

什么是路由?

一个路由就是一个映射关系(key:value)，每一个路径(/home)对应一个组件

key为路径, value可能是function或component

路由分类

后端路由：

理解： value是function, 用来处理客户端提交的请求。

注册路由： router.get(path, function(req, res))

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

前端路由：

浏览器端路由，value是component，用于展示页面内容。

注册路由: <Route path="/test" component={Test}>

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

### 前端路由原理

浏览器的历史记录是一个栈的结构，

push是追加历史记录，replace是替换当前记录

 // let history = History.createBrowserHistory() //方法一，直接使用H5推出的history身上的API

    let history = *History*.createHashHistory() //方法二，hash值（锚点）

### 路由的基本使用

react-router-dom的理解

react的一个插件库。

专门用来实现一个SPA应用。

基于react的项目基本都会用到此库。

react-router分别给三种开发人员用web、native(react人员)、any

react-router-dom专门给web开发人员用的

路由Route 路由器Router

yarn add react-router-dom@5

被前端路由器检测到

变化的抽成组件

在App.jsx

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

{/\* 原生html中，靠<a>跳转不同的页面 \*/}

              {/\* <a className="list-group-item" href="./about.html">About</a>

              <a className="list-group-item active" href="./home.html">Home</a> \*/}

              {/\* 在React中靠路由链接实现切换组件--编写路由链接 \*/}

              <*Link* *className*="list-group-item" *to*="/about">About</*Link*>

              <*Link* *className*="list-group-item" *to*="/home">Home</*Link*>

Router分为两种，一种为BrowserRouter，一种为HashRouter

import Home from './components/Home'

import About from './components/About'

{/\* 注册路由 \*/}

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

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

在index.js中

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

  <*BrowserRouter*>

    <*App*/>

  </*BrowserRouter*>,

路由器包裹App

**## 三、路由的基本使用**

      1.明确好界面中的导航区、展示区

      2.导航区的a标签改为Link标签

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

      3.展示区写Route标签进行路径的匹配

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

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

### 路由组件和一般组件

两个link在浏览器中转变成a标签了，to转变成href属性

#后面的资源属于前台资源，不会作为资源发送给服务器

**## 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"

Ctrl+]整体向右移动

Ctrl+[整体向左移动

### Navlink的使用

  <*NavLink* *activeClassName*="atguigu" *className*="list-group-item" *to*="/about">About</*NavLink*>

  <*NavLink* *activeClassName*="atguigu" *className*="list-group-item" *to*="/home">Home</*NavLink*>

Navlink标签高亮显示

activeClassName可以自定义样式

在public文件里面的index.html里面添加

*.atguigu* {

        background-color: rgb(209, 137, 4) !important;

        color: white !important;

      }

### 封装Navlink的使用

在components文件下创建MyNavLink文件的index.jsx

在App.jsx中

import MyNavLink from './components/MyNavLink'

在MyNavLink中的index.jsx

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

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

在App.js中

{/\* 在React中靠路由链接实现切换组件--编写路由链接 \*/}

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

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

标签里面的的About收集到children里面，一样可以通过this.props去收集，标签体内容是一个特殊的标签属性

**## 五、NavLink 与封装 NavLink**

          1.NavLink可以实现路由链接的高亮，通过activeClassName指定样式名

2.标签体内容是一个特殊的标签属性

          3.通过this.props.children可以获取标签体内容

### Switch的使用

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

 {/\* 注册路由 \*/}

                <*Switch*>

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

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

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

                </*Switch*>

**## 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)。

### 解决样式丢失问题

<http://localhost:3000/css/bootstrap.css>

http://localhost:3000就是脚手架内置的服务器，就是通过webpack、devServer开启的服务器，react通过webpack配置的public文件夹是localhost:3000这台内置服务器的根路径，localhost:3000就代表我们的脚手架

如果你请求了一个不存在的资源 例如localhost:3000/css/favicon.ico，那么我就把public里面的index.html给你

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

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

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

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

点击没问题，刷新浏览器样式就丢了

之前请求路径为:http://localhost:3000/css/bootstrap.css

现在请求路径为:http://localhost:3000/atguigu/css/bootstrap.css

刷新导致路径变了

解决一：

  <link *rel*="stylesheet" *href*="/css/bootstrap.css" />

将./css/bootstrap.css中的.去掉

./是在当前目录下去取资源会带上atguigu

去掉.就会取localhost:3000下去取资源

解决二：

将.改成%PUBLIC\_URL%

<link *rel*="stylesheet" *href*="%PUBLIC\_URL%/css/bootstrap.css" />

%PUBLIC\_URL%代表代表的就是public这个东西的绝对路径，绝对路径不会出现问题

解决三：

<link *rel*="stylesheet" *href*="./css/bootstrap.css" />

不用BrowerRouter，用HashRouter

#后面的东西都认为是前端的资源，根本就不带给3000这台服务器

打包之后会显示高亮logo



一般使用解决一的方案

**## 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

### 路由的模糊匹配与严格匹配

模糊匹配：

最左前缀匹配

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

匹配可以对上

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

精准匹配：

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

匹配不可以对上

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

**## 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

### Redirect的使用

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

默认路由

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

**## 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

### 嵌套路由

在Home下创建Message的index.jsx，News的index.jsx

在Home下的index.jsx

  <li>

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

              </li>

              <li>

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

              </li>

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

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

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

import News from './News'

import Message from './Message'

路由的注册是由顺序的，从最先开始注册的路由进行逐层匹配，Home组件下的index.jsx的/news最先会去App.jsx下的/home和/about中匹配，都没匹配就会找重定向的<Redirect to="/about"/>

Message、News都是Home的子组件，所以将/news改成/home/news

Home组件下的index.jsx的/home/news匹配App.jsx下的/home和/about，发现和/home匹配，由于是默认为模糊匹配

**## 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

### 向路由组件传递params参数

创建Message组件下的Detail的index.jsx

在Message组件下的index.jsx

  state = {

    messageArr:[

      {id:'01',title:'消息1'},

      {id:'02',title:'消息2'},

      {id:'03',title:'消息3'},

    ]

  }

模板字符串

messageArr.map((*msgObj*)=>{

              return (

                <li *key*={*msgObj*.id}>

                  {/\* 向路由组件传递params参数 \*/}

                  <*Link* *to*={`/home/message/detail/${*msgObj*.id}/${*msgObj*.title}`}>{*msgObj*.title}</*Link*>

                </li>

              )

            })

 {/\* 声明接收params参数 \*/}

        <*Route* *path*="/home/message/detail/:id/:title" *component*={Detail}/>

在Detail的index.jsx

const DetailData = [

  {id:'01',content:'你好，中国'},

  {id:'02',content:'你好，尚硅谷'},

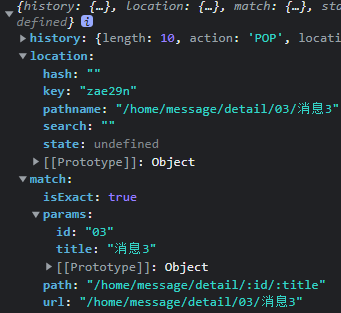
  {id:'03',content:'你好，未来的自己'}

]

ajax中有三种传参方式，query、params、body

body请求体参数还有两种编码形式，一个是urlencode，一个是json

打印this.props



 // 接收params参数

    const {id,title} = *this*.props.match.params

查找对应的id

 const findResult = DetailData.find((*detailObj*)=>{

      return *detailObj*.id === id

    })

### 向路由组件传递search参数

在Message组件下的index.jsx

   {/\* 向路由组件传递params参数 \*/}

                  {/\* <Link to={`/home/message/detail/${msgObj.id}/${msgObj.title}`}>{msgObj.title}</Link> \*/}

                  {/\* 向路由组件传递search参数 \*/}

                  <*Link* *to*={`/home/message/detail/?id=${*msgObj*.id}&title=${*msgObj*.title}`}>{*msgObj*.title}</*Link*>

 {/\* 声明接收params参数 \*/}

        {/\* <Route path="/home/message/detail/:id/:title" component={Detail}/> \*/}

        {/\* search参数无需声明接收，正常注册路由即可 \*/}

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

在Detail组件下的index.jsx

// 接收params参数

    // const {id,title} = this.props.match.params

    // 接收search参数

    const {search} = *this*.props.location

    const {id,title} = *qs*.parse(search.slice(1))

打印this.props

会有location下的search

search:”?id=01&title=消息1”

要将它转换成{id:’01’,title:’xx1’}

import qs from 'querystring'

key=value&key=value这种形式的编码叫做urlencoded编码

qs.stringify(obj)将一个对象转换成urlencoded编码形式

qs.parse(str)将一个字符串转换成一个对象

### 向路由组件传递state参数

在Message下的index.jsx

{/\* 向路由组件传递params参数 \*/}

                  {/\* <Link to={`/home/message/detail/${msgObj.id}/${msgObj.title}`}>{msgObj.title}</Link> \*/}

                  {/\* 向路由组件传递search参数 \*/}

                  {/\* <Link to={`/home/message/detail/?id=${msgObj.id}&title=${msgObj.title}`}>{msgObj.title}</Link> \*/}

                  {/\* 向路由组件传递state参数 \*/}

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

    {/\* 声明接收params参数 \*/}

        {/\* <Route path="/home/message/detail/:id/:title" component={Detail}/> \*/}

        {/\* search参数无需声明接收，正常注册路由即可 \*/}

        {/\* <Route path="/home/message/detail" component={Detail}/> \*/}

        {/\* state参数无需声明接收，正常注册路由即可 \*/}

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

第一个{}表示js表达式后面的{}表示对象

在Detail下的index.jsx

    // 接收params参数

    // const {id,title} = this.props.match.params

    // 接收search参数

    // const {search} = this.props.location

    // const {id,title} = qs.parse(search.slice(1))

    // 接收state参数

    const {id,title} = *this*.props.location.state || {}

 const findResult = DetailData.find((*detailObj*)=>{

      return *detailObj*.id === id

    }) || {}

打印this.props

会有location下的state

state:{id:”01”,title:”消息1”}

清除浏览器缓存

在浏览器里面的设置下，找到隐私设置和安全性，清除浏览器数据

### 总结三个传递参数方式

**## 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：刷新也可以保留住参数

对象需要{}来解析，字符串不需要

### push与replace

  {/\* 向路由组件传递state参数 \*/}

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

默认为push模式，添加replace进行转换就不会留下历史痕迹

### 编程式路由导航

  {/\* 向路由组件传递state参数 \*/}

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

  &nbsp;<button *onClick*={()=> *this*.pushShow(*msgObj*.id,*msgObj*.title)}>push查看</button>

                  &nbsp;<button *onClick*={()=> *this*.replaceShow(*msgObj*.id,*msgObj*.title)}>replace查看</button>

  replaceShow = (*id*,*title*)=>{

    //replace跳转+携带params参数

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

    //replace跳转+携带search参数

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

    //replace跳转+携带state参数

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

  }

  pushShow = (*id*,*title*)=>{

    //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})

  }

Link的默认push和push以及replace区别

Link的默认push回退是直接http://localhost:3000/home/message/detail/03/消息3回退到http://localhost:3000/home/message/detail(老版本和push没有区别)

而push回退是http://localhost:3000/home/message/detail/02/消息2

replace回退是http://localhost:3000/home/news没有缓存记录

 <button *onClick*={*this*.back}>回退</button>&nbsp;

        <button *onClick*={*this*.forward}>前进</button>&nbsp;

        <button *onClick*={*this*.go}>go</button>

  back = ()=>{

*this*.props.history.goBack()

  }

  forward = ()=>{

*this*.props.history.goForward()

  }

  go = ()=>{

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

  }

**## 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()

                -this.prosp.history.replace()

                -this.prosp.history.goBack()

                -this.prosp.history.goForward()

                -this.prosp.history.go()

在News下的index.jsx

  /\* componentDidMount(){

    setTimeout(()=>{

      this.props.history.push('/home/message')

    },2000)

  } \*/

### withRouter的使用

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

驼峰命名的是组件

withRouter是函数

作用：能够接收一般组件，然后把一般组件的身上加上了路由组件所特有的三个东西history、localhost、match

### BrowserRouter与HashRouter

**## 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！！！没有用history对象这个api

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

## antd

### 基本使用

流行的开源React UI组件库

material-ui(国外)

官网: [http://www.material-ui.com/#/](http://www.material-ui.com/" \l "/)

github: <https://github.com/callemall/material-ui>

ant-design(国内蚂蚁金服)

官网: <https://ant.design/index-cn>

Github: <https://github.com/ant-design/ant-design/>

安装yarn add antd

按钮日期选择框

import { Button,DatePicker } from 'antd';

import ‘antd/dist/antd.css’

微信微博搜索图标

import {WechatOutlined,WeiboOutlined,SearchOutlined} from '@ant-design/icons'

<*WechatOutlined* />

        <*WeiboOutlined* />

        <*DatePicker*/>

 <*Button* *type*="primary" *icon*={<*SearchOutlined* />}>

          Search

        </*Button*>

onChange 时间发生变化的回调 function(date: moment, dateString: string)

const { RangePicker } = DatePicker;

选择时间段

<*RangePicker*/>

适合于后台管理系统

### 样式的按需导入

官网点击文档 版本3.x.x

点击在create-react-app中使用

而且上面的例子加载了全部的 antd 组件的样式（gzipped 后一共大约 60kb）。

此时我们需要对 create-react-app 的默认配置进行自定义，这里我们使用 [react-app-rewired](https://github.com/timarney/react-app-rewired) （一个对 create-react-app 进行自定义配置的社区解决方案）。

引入 react-app-rewired 并修改 package.json 里的启动配置。由于新的 [react-app-rewired@2.x](https://github.com/timarney/react-app-rewired" \l "alternatives) 版本的关系，你还需要安装 [customize-cra](https://github.com/arackaf/customize-cra)。

然后在项目根目录创建一个 config-overrides.js 用于修改默认配置。

终端中yarn eject

暴露react的webpack配置

 config-overrides.js 用于看不见js配置还要改的js配置，将需要改的js配置写到里面。

用[customize-cra](https://github.com/arackaf/customize-cra)去执行 config-overrides.js

如果不用react-app-rewired，就只能使用react-scripts start去执行项目，不能用短命令yarn start去执行项目。

第一步：安装yarn add react-app-rewired customize-cra

第二步：

/\* package.json \*/ "scripts": {

- "start": "react-scripts start",

+ "start": "react-app-rewired start",

- "build": "react-scripts build",

+ "build": "react-app-rewired build",

- "test": "react-scripts test",

+ "test": "react-app-rewired test", }

将红色的改成绿色的 react-scripts改成react-app-rewired

eject不用改

"eject": "react-scripts eject"

第三步：创建根目录下的config-overrides.js跟package.json同级

第四步：使用 babel-plugin-import[#](https://3x.ant.design/docs/react/use-with-create-react-app-cn" \l "%E4%BD%BF%E7%94%A8-babel-plugin-import)

注意：antd 默认支持基于 ES module 的 tree shaking，js 代码部分不使用这个插件也会有按需加载的效果。

[babel-plugin-import](https://github.com/ant-design/babel-plugin-import) 是一个用于按需加载组件代码和样式的 babel 插件（[原理](https://3x.ant.design/docs/react/getting-started-cn" \l "%E6%8C%89%E9%9C%80%E5%8A%A0%E8%BD%BD)），现在我们尝试安装它并修改 config-overrides.js 文件。

安装：yarn add babel-plugin-import

Ctrl+Alt+方向键可以多行选择

//配置具体的修改规则

const { override, fixBabelImports,addLessLoader} = require('customize-cra');

*module*.*exports* = override(

  fixBabelImports('import', {

    libraryName: 'antd',

    libraryDirectory: 'es',//es的模块化规范

    style: true,

  }),

  addLessLoader({

    lessOptions:{

      javascriptEnabled: true,

      modifyVars: { '@primary-color': 'green' },

    }

  }),

);

### 自定义主题

antd使用less写的改成了用css写的

改less里面的变量

安装less和less编译less-loader

yarn add less less-loader

在config-overrides.js中配置

添加addLessLoader

const { override, fixBabelImports,addLessLoader} = require('customize-cra');

将style:css，改成style:true

style: true,

antd老版本要包一层lessOptions

  addLessLoader({

    lessOptions:{

      javascriptEnabled: true,//允许用js去修改antd底层的less文件

      modifyVars: { '@primary-color': 'green' },//修改颜色变量

    }

  }),

**## 十四、antd 的按需引入+自定主题**

        1.安装依赖：yarn add react-app-rewired customize-cra babel-plugin-import less less-loader

        2.修改package.json

            ....

              "scripts": {

                "start": "react-app-rewired start",

                "build": "react-app-rewired build",

                "test": "react-app-rewired test",

                "eject": "react-scripts eject"

              },

            ....

        3.根目录下创建config-overrides.js

            //配置具体的修改规则

            const { override, fixBabelImports,addLessLoader} = require('customize-cra');

            module.exports = override(

              fixBabelImports('import', {

                libraryName: 'antd',

                libraryDirectory: 'es',

                style: true,

              }),

              addLessLoader({

                lessOptions:{

                  javascriptEnabled: true,

                  modifyVars: { '@primary-color': 'green' },

                }

              }),

            );

          4.备注：不用在组件里亲自引入样式了，即：import 'antd/dist/antd.css'应该删掉

## redux

### 简介

学习文档

英文文档: <https://redux.js.org/>

中文文档: <http://www.redux.org.cn/>

Github: <https://github.com/reactjs/redux>

redux是什么

redux是一个专门用于做状态管理的JS库(不是react插件库)。

它可以用在react, angular, vue等项目中, 但基本与react配合使用。

作用: 集中式管理react应用中多个组件共享的状态。

什么情况下需要使用redux

某个组件的状态，需要让其他组件可以随时拿到（共享）。

一个组件需要改变另一个组件的状态（通信）。

总体原则：能不用就不用, 如果不用比较吃力才考虑使用。

### 工作流程



action

动作的对象

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

例子：{ type: 'ADD\_STUDENT',data:{name: 'tom',age:18} }

reducer

用于初始化状态、加工状态。

加工时，根据旧的state和action， 产生新的state的纯函数。

store

将state、action、reducer联系在一起的对象

如何得到此对象?

import {createStore} from 'redux'

import reducer from './reducers'

const store = createStore(reducer)

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

### 求和案例\_纯react版

在cmd终端命令中创建项目

npx create-react-app redux\_test

在components下的Count下的index.jsx

  state = {count:0}

  //加法

  increment = ()=>{

    const {value} = *this*.selectNumber

    const {count} = *this*.state

*this*.setState({count:count+value\*1})

  }

 <select *ref*={*c* => *this*.selectNumber = *c*}>

<button *onClick*={*this*.increment}>+</button>&nbsp;

### 精简版

创建redux下的store.js和count\_reducer.js

安装yarn add redux

在store.js中

/\*

  该文件专门用于暴露一个store对象，整个应用只有一个store对象

\*/

//引入createStore，专门用于创建redux中最为核心的store对象

import {createStore} from 'redux'

//引入为Count组件服务的reducer

import countReducer from './count\_reducer'

//暴露store

export default createStore(countReducer)

createStore被启用名字改成了legacy\_createStore

在count\_reducer.js中

/\*

  1.该文件是用于创建一个为Count组件服务的reducer，reducer的本质就是一个函数

  2.reducer函数会接到两个参数，分别为：之前的状态(preState)，动作对象(action)

\*/

const initState = 0 //初始化状态

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

  // console.log(preState);

  //从action对象中获取：type、data

  const {type,data} = *action*

  //根据type决定如何加工数据

  switch (type) {

    case 'increment': //如果是加

      return *preState* + data

    case 'decrement': //若果是减

      return *preState* - data

    default:

      return *preState*

  }

}

在Count下的index.jsx中

//引入store，用于获取redux中保存状态

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

<h1>当前求和为：{store.getState()}</h1>

只能在Count组件里面进行检测

  /\* componentDidMount(){

    //检测redux中状态的变化，只要变化，就调用render

    store.subscribe(()=>{

      this.setState({})

    })

  } \*/

获取store里面的count值也就是总数和

const count = store.getState()

换种检测的方法

在index.js

import store from './redux/store'

针对所有组件进行检测，有diff算法在，不会产生效率低

store.subscribe(()=>{

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

})

**## 1.求和案例\_redux精简版**

    (1).去除Count组件自身的状态

    (2).src下建立:

            -redux

              -store.js

              -count\_reducer.js

    (3).store.js：

          1).引入redux中的createStore函数，创建一个store

          2).createStore调用时要传入一个为其服务的reducer

          3).记得暴露store对象

    (4).count\_reducer.js：

          1).reducer的本质是一个函数，接收：preState,action，返回加工后的状态

          2).reducer有两个作用：初始化状态，加工状态

          3).reducer被第一次调用时，是store自动触发的，

                  传递的preState是undefined,

                  传递的action是:{type:'@@REDUX/INIT\_a.2.b.4}

    (5).在index.js中监测store中状态的改变，一旦发生改变重新渲染<App/>

        备注：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。

### 完整版

在redux中创建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})

在redux中创建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'

在Count下的index.jsx

//引入actionCreator，专门用于创建action对象

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

在rdux中创建count\_reducer.js

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

 case INCREMENT: //如果是加

      return *preState* + data

    case DECREMENT: //若果是减

      return *preState* - data

**## 2.求和案例\_redux完整版**

    新增文件：

      1.count\_action.js 专门用于创建action对象

      2.constant.js 放置容易写错的type值

### 异步action版

action包括Object(同步)和function(异步)

在Count下的index.jsx

//引入actionCreator，专门用于创建action对象

import {

  createIncrementAction,

  createDecrementAction,

  createIncrementAsyncAction

} from '../../redux/count\_action'

在count\_action.js中

//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，异步action不是必须要用的。

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

  return (*dispatch*)=>{

    setTimeout(()=>{

      dispatch(createIncrementAction(*data*))

    },*time*)

  }

}

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

在store下的index.jsx

导入可以处理异步函数的中间件

//引入createStore，专门用于创建redux中最为核心的store对象

import {createStore,applyMiddleware} from 'redux'

//引入redux-thunk，用于支持异步action

import thunk from 'redux-thunk'

//暴露store

export default createStore(countReducer,applyMiddleware(thunk))

收到一个项目时候，npm install安装各种依赖包，项目中会出现node\_modules目录。

终端中yarn add redux-thunk thunk转换

**## 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

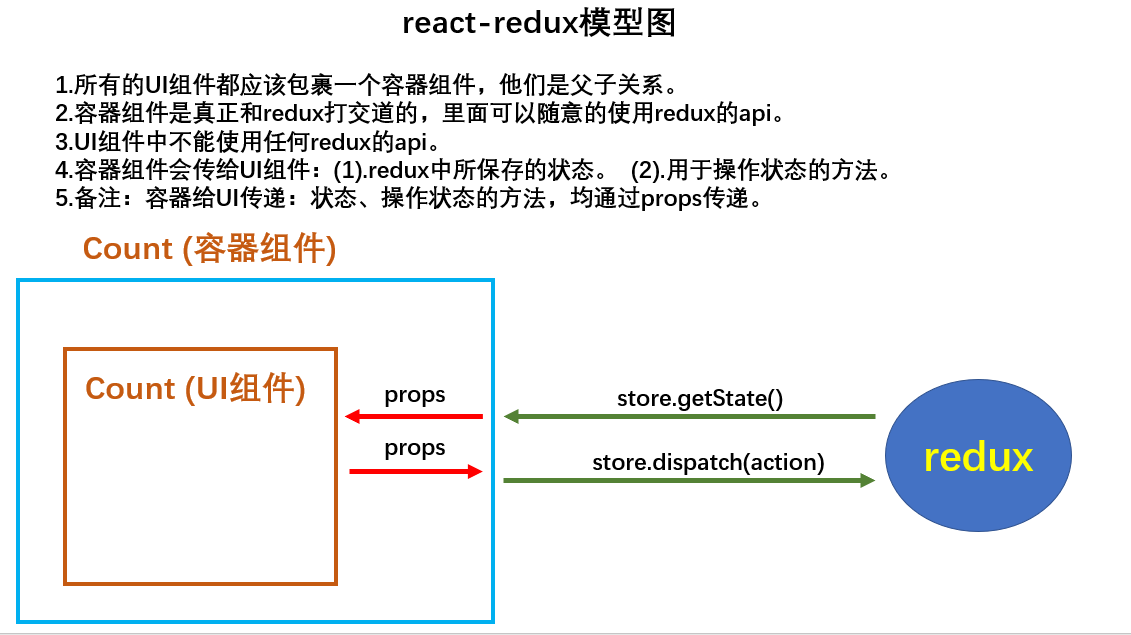
          2).创建action的函数不再返回一般对象，而是一个函数，该函数中写异步任务。

          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。

reducer减速机，还原剂，缩减者

### 对react-redux的理解



createstore()

作用：创建包含指定reducer的store对象

store对象

作用: redux库最核心的管理对象

它内部维护着:

state

reducer

核心方法:

getState()

dispatch(action)

subscribe(listener)

具体编码:

store.getState()

store.dispatch({type:'INCREMENT', number})

store.subscribe(render)

applyMiddleware()

作用：应用上基于redux的中间件(插件库)

combineReducers()

作用：合并多个reducer函数

redux异步编程

理解：

redux默认是不能进行异步处理的,

某些时候应用中需要 在redux中执行异步任务(ajax, 定时器)

使用异步中间件

npm install --save redux-thunk

react-redux

理解

一个react插件库

专门用来简化react应用中使用redux

react-Redux将所有组件分成两大类

UI组件

只负责 UI 的呈现，不带有任何业务逻辑

通过props接收数据(一般数据和函数)

不使用任何 Redux 的 API

一般保存在components文件夹下

容器组件

负责管理数据和业务逻辑，不负责UI的呈现

使用 Redux 的 API

一般保存在containers文件夹下

### 连接容器组件与UI组件

创建contains下的Count的index.jsx

yarn add react-redux

//引入Count的UI组件

import CountUI from '../../components/Count'

//引入connect用于连接UI组件与redux

import {connect} from 'react-redux'

//使用connect()()创建并暴露一个Count的容器组件

export default connect()(CountUI)

在App.jsx中

import React, { Component } from 'react'

import Count from './containers/Count'

import store from './redux/store'

    {/\* 给容器组件传递store \*/}

        <*Count* *store*={store} />

### react-redux的基本使用

在index.js中

//监测redux中状态的改变，如redux的状态发生了改变，那么重新渲染App组件

store.subscribe(()=>{

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

})

在App.jsx

  {/\* 给容器组件传递store \*/}

        <*Count* *store*={store} />

在containers下Count的index.js中

函数中返回的对象中的key就作为传递给UI组件props的key，value就作为传递给UI组件props的value---状态

/\*

  1.mapStateToProps函数返回的是一个对象；

  2.返回的对象中的key就作为传递给UI组件props的key,value就作为传递给UI组件props的value

  3.mapStateToProps用于传递状态

\*/

function mapStateToProps(*state*){

  return {count:*state*}

}

在components下Count的index.jsx

 <h1>当前求和为：{*this*.props.count}</h1>

//引入action

import {

  createIncrementAction,

  createDecrementAction,

  createIncrementAsyncAction

} from '../../redux/count\_action'

函数中返回的对象中的key就作为传递给UI组件props的key，value就作为传递给UI组件props的value---操作状态的方法

dispatch已经传入了store

/\*

  1.mapDispatchToProps函数返回的是一个对象；

  2.返回的对象中的key就作为传递给UI组件props的key,value就作为传递给UI组件props的value

  3.mapDispatchToProps用于传递操作状态的方法

\*/

function mapDispatchToProps(*dispatch*){

  return {

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

    jian:*number* => dispatch(createDecrementAction(*number*)),

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

  }

}

//异步加

  incrementAsync = ()=>{

    const {value} = *this*.selectNumber

*this*.props.jiaAsync(value\*1,500)

  }

**## 4.求和案例\_react-redux基本使用**

      (1).明确两个概念：

            1).UI组件:不能使用任何redux的api，只负责页面的呈现、交互等。

            2).容器组件：负责和redux通信，将结果交给UI组件。

      (2).如何创建一个容器组件————靠react-redux 的 connect函数

              connect(mapStateToProps,mapDispatchToProps)(UI组件)

                -mapStateToProps:映射状态，返回值是一个对象

                -mapDispatchToProps:映射操作状态的方法，返回值是一个对象

      (3).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

      (4).备注2：mapDispatchToProps，也可以是一个对象

### 优化1\_简写mapDispatch

在contains下的Count下index.jsx

//使用connect()()创建并暴露一个Count的容器组件

export default connect(

*state* => ({count:*state*}),

  //mapDispatchToProps的一般写法

  /\* dispatch => ({

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

    jian:number => dispatch(createDecrementAction(number)),

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

  }) \*/

  //mapDispatchToProps的简写

  {

    jia:createIncrementAction,

    jian:createDecrementAction,

    jiaAsync:createIncrementAsyncAction,

  }

)(Count)

### 优化2\_Provider组件的使用

不需要这个，由于使用了connect

//监测redux中状态的改变，如redux的状态发生了改变，那么重新渲染App组件

store.subscribe(()=>{

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

})

在index.js

Provider可以统一给所有组件传递store

import {Provider} from 'react-redux'

ReactDOM.render(

  <*Provider* *store*={store}>

    <*App*/>

  </*Provider*>,

  document.getElementById('root')

)

### 优化3\_整合UI组件与容器组件

//定义UI组件

class Count extends Component {



**## 5.求和案例\_react-redux优化**

      (1).容器组件和UI组件整合一个文件

      (2).无需自己给容器组件传递store，给<App/>包裹一个<Provider store={store}>即可。

      (3).使用了react-redux后也不用再自己检测redux中状态的改变了，容器组件可以自动完成这个工作。

      (4).mapDispatchToProps也可以简单的写成一个对象

      (5).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

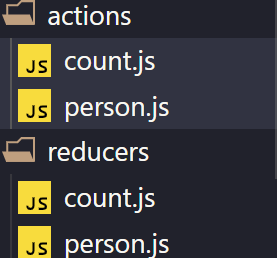
                    {key:xxxxxAction} //映射操作状态的方法

                  )(UI组件)

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

### 数据共享\_编写Person组件

分别在actions和reducers中创建count.js和person.js



### 编写Person组件的reducer

在constant.js中定义常量

在actions下的person.js中

import {ADD\_PERSON} from '../constant'

//创建增加一个人的action动作对象

export const createAddPersonAction = *personObj* => ({type:ADD\_PERSON,data:*personObj*})

在reducers下的person.js中

初始化状态和加工状态

import {ADD\_PERSON} from '../constant'

//初始化人的列表

const initState = [{id:'001',name:'tom',age:18}]

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

  // console.log('personReducer@#@#@#');

  const {type,data} = *action*

  switch (type) {

    case ADD\_PERSON: //若是添加一个人

      return [data,...*preState*]

    default:

      return *preState*

  }

}

创建Person下的index.jsx

### 完成数据共享

在store.js中

//引入createStore，专门用于创建redux中最为核心的store对象

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

//引入为Count组件服务的reducer

import personReducer from './reducers/person'

//汇总所有的reducer变为一个总的reducer

const allReducer = combineReducers({

  he:countReducer,

  rens:personReducer

})

在countainers中的Person的indxe.jsx

import {createAddPersonAction} from '../../redux/actions/person'

    const name = *this*.nameNode.value

    const age = *this*.ageNode.value

    const personObj = {id:nanoid(),name,age}

*this*.props.jiaYiRen(personObj)

export default connect(

*state* => ({yiduiren:*state*.rens,he:*state*.he}),//映射状态

  {jiaYiRen:createAddPersonAction}//映射操作状态的方法

)(Person)

*this*.props.yiduiren.map((*p*)=>{

              return <li *key*={*p*.id}>{*p*.name}--{*p*.age}</li>

            })

在containers中的Count的index.jsx

<h2>我是Count组件,下方组件总人数为:{*this*.props.renshu}</h2>

*state* => ({

    count:*state*.he,

    renshu:*state*.rens.length

  }),

**## 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！！！

      (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”。

### 相关API

Provider：让所有组件都可以得到state数据

**<Provider store={store}>**

**<App />**

**</Provider>**

connect：用于包装 UI 组件生成容器组件

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

**connect(**

**mapStateToprops,**

**mapDispatchToProps**

**)(Counter)**

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**

**}**

**}**

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

### 纯函数

  //preState.unshift(data) //此处不可以这样写，这样会导致preState被改写了，personReducer就不是纯函数了。

      return [data,...*preState*]

界面不刷新，由于preState返回的不是一个新数组

在containers中的Person下index.jsx

 const age = *this*.ageNode.value\*1

纯函数

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)

必须遵守以下一些约束

不得改写参数数据

不会产生任何副作用，例如网络请求，输入和输出设备

不能调用Date.now()或者Math.random()等不纯的方法

redux的reducer函数必须是一个纯函数

高阶函数

理解: 一类特别的函数

情况1: 参数是函数

情况2: 返回是函数

常见的高阶函数:

定时器设置函数

数组的forEach()/map()/filter()/reduce()/find()/bind()

promise

react-redux中的connect函数

作用: 能实现更加动态, 更加可扩展的功能

### redux开发工具

安装在应用商店搜索redux\_dev\_tools安装

访问不到外网，就在扩展程序里面，加载已解压的扩展程序，打开文件夹D:\vs code html\react\react全家桶资料\06\_其他\redux\_dev\_tools

下载工具依赖包

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

在store.js

//引入redux-devtools-extension

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

//暴露store

export default createStore(reducer,composeWithDevTools(applyMiddleware(thunk)))

**## 7.求和案例\_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)))

### 最终版

**## 8.求和案例\_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

在index.js

  /\* 此处需要用Provider包裹App，目的是让App所有的后代容器组件都能接收到store \*/

  <*Provider* *store*={store}>

在App.jsx

import Count from './containers/Count' //引入的Count的容器组件

import Person from './containers/Person' //引入的Person的容器组件

store.js中不引入任何的reducers，创建reducers文件下的index.js

在store.js中

//引入汇总之后的reducer

import reducer from './reducers'

在reducers下的index.js

/\*

  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

})

### 项目打包运行

npm run build

在一个index.html文件中

npm i serve -g

然后输入serve

serve a 可以启动该目录下的a目录的index.html

在react中 serve build启动服务器 从开发环境转换成生产环境

## 扩展

### setState

**## 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 原状态就是state对象里面的状态

        (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

```

//对象式的setState

    /\* //1.获取原来的count值

    const {count} = this.state

    //2.更新状态

    this.setState({count:count+1},()=>{

      console.log(this.state.count);

    })

//console.log('12行的输出',this.state.count); //0 \*/

//setState是同步的方法，由程序员在主线上调用的方法，引起react后续更新状态的变化是异步的

    //函数式的setState

*this*.setState( *state* => ({count:*state*.count+1}))

### lazyLoad

**## 2. lazyLoad**

**### 路由组件的lazyLoad**

```js

  //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*>

安装yarn add react-router-dom

在2\_lazyLoad下的index.jsx

import Loading from './Loading' //本身也懒加载

const Home = lazy(()=> import('./Home') )

const About = lazy(()=> import('./About'))

Suspense悬念

 <*Suspense* *fallback*={<*Loading*/>}>

                  {/\* 注册路由 \*/}

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

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

                </*Suspense*>

### state\_Hook

函数式组件

function Demo(){

  //console.log('Demo');

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

  //加的回调

  function add(){

    //setCount(count+1) //第一种写法

    setCount(*count* => *count*+1 )

  }

  <h2>当前求和为：{count}</h2>

      <button *onClick*={add}>点我+1</button>

**## 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

```

**#### 2. 三个常用的Hook**

```

(1). State Hook: React.useState()

(2). Effect Hook: React.useEffect()

(3). Ref Hook: React.useRef()

```

**#### 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

### EffectHook

**#### 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()

*React*.useEffect(()=>{

    let timer = setInterval(()=>{

      setCount(*count* => *count*+1 )

    },1000)

    return ()=>{

      clearInterval(timer)

    }

  },[])

  //卸载组件的回调

  function unmount(){

    ReactDOM.unmountComponentAtNode(document.getElementById('root'))

  }

<button *onClick*={unmount}>卸载组件</button>

### RefHook

const myRef = *React*.useRef()

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

<button *onClick*={show}>点我提示数据</button>

  //提示输入的回调

  function show(){

    alert(myRef.current.value)

  }

**#### 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

### Fragment

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

<*Fragment* *key*={1}>//只接受key属性

        <input *type*="text"/>

        <input *type*="text"/>

      </*Fragment*>

**## 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*

### Context

//创建Context对象

const MyContext = *React*.createContext()

const {Provider,Consumer} = MyContext

B

<*Provider* *value*={{username,age}}>

          <*B*/>

        </*Provider*>

C

/\* class C extends Component {

  //声明接收context

  static contextType = MyContext

  render() {

    const {username,age} = this.context

    return (

      <div className="grand">

        <h3>我是C组件</h3>

        <h4>我从A组件接收到的用户名:{username},年龄是{age}</h4>

      </div>

    )

  }

C函数式

  <*Consumer*>

        {*value* => `${*value*.username},年龄是${*value*.age}`}

      </*Consumer*>

**## 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**

```js

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, 一般都它的封装react插件

Token:ghp\_URaF50og1dQGkDCN7ty5vj1R6Js1nq2ydjM9

Id:e3d54b612f66bcd9c7fb23469273fc90

### PureComponent

**## 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()*

*>*

*> 2. 只当前组件重新render(), 就会自动重新render子组件 ==> 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:

    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

    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

方法一：

Parent

  /\* shouldComponentUpdate(nextProps,nextState){

    // console.log(this.props,this.state); //目前的props和state

    // console.log(nextProps,nextState); //接下要变化的目标props，目标state

    return !this.state.carName === nextState.carName

  } \*/

Child

  /\* shouldComponentUpdate(nextProps,nextState){

    console.log(this.props,this.state); //目前的props和state

    console.log(nextProps,nextState); //接下要变化的目标props，目标state

    return !this.props.carName === nextProps.carName

  } \*/

方法二：

export default class Parent extends PureComponent

    const {stus} = *this*.state

*this*.setState({stus:['小刘',...stus]})

工程化 移动端 mobx mongdb reacnative uniapp

### renderProps

<h3>我是Parent组件</h3>

        <*A* *render*={(*name*)=><B *name*={*name*}/>}/>

state = {name:'tom'}

  render() {

    console.log(*this*.props);

    const {name} = *this*.state

    return (

      <div *className*="a">

        <h3>我是A组件</h3>

        {*this*.props.render(name)}

      </div>

<div *className*="b">

        <h3>我是B组件,{*this*.props.name}</h3>

      </div>

**## 7. render props**

**### 如何向组件内部动态传入带内容的结构(标签)?**

  Vue中:

    使用slot技术, 也就是通过组件标签体传入结构  <AA><BB/></AA>

  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, 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}

### ErrorBoundary

**## 8. 错误边界**

**#### 理解：**

错误边界：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

```js

// 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) {

    // 统计页面的错误。发送请求发送到后台去

    console.log(error, info);

}

  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>

    )

  }

### 组件间通信方式总结

**## 9. 组件通信方式总结**

**#### 方式：**

    props：

      (1).children props

      (2).render props

    消息订阅-发布：

      pubs-sub、event等等

    集中式管理：

      redux、dva等等

    conText:

      生产者-消费者模式

**#### 组件间的关系**

    父子组件：props

    兄弟组件(非嵌套组件)：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(用的少)