🡪

前端的三大框架angular （Google Supported）， react （Facebook Supported）， vue。

React的优势：

使用虚拟DOM （操作React虚拟DOM, 比操作实际DOM花费少）

性能性很高。

解决各终端的问题包括PC，移动端问题

React的缺点：

需要插件的支持。

学习曲线高

P.s 网页的发展从以前的单网页开始，当数据进行变化或者用户点击进行交互以后，以前的网页需要整体的整份的网页进行更新。 后来为了简化，我们想要网页进行部分更新， 例如Jquery的插件的引入。 再后来随着交互的程度越来越高，我们想当数据变化的时候以最少代价自动更新部门网页，此时React技术出现。我们通过将网页进行控件化，并且和数据进行绑定，并且使用React的虚拟DOM技术，可以最快,花费最少的改变网页。

React Native用于开发移动端手机应用。

🡪

JavaScript库与框架的区别。库是封装了原生的Javascript，还是使用原js的思想，例如JQuery, Zepto等。而框架则是从本质上修改了思想，以不同的方式去操作。

🡪

React有自己自定义的语法JSX，JSX语言可以允许将HTML，XML等一些其他语言混合Javascript一起写。遇到 HTML 标签（以 < 开头），就用 HTML 规则解析；遇到代码块（以 {开头），就用 JavaScript 规则解析。并且JSX语言不能直接被识别，需要最终转化为Javascript语言才可以。（并且我们在JSX语法中写入的形似html标签，在经过编译以后，将变为javascript语法给浏览器，而不会因为像html语言而不变）

这个转化的过程，如果我们只是随手写一个小型的React,我们可以通过外置引入额外的JS文件来帮助我们完成这个过程，例如babel/browser.js.

如果是比较有规模的react app, 或者是标准做法应该是通过特定的构建工具，例如React推荐的webpack来有结构的进行打包转化处理。(其中Webpack配置中，我们便会配置jsx通过babel来进行转化为Javascript)

🡪

一些React的基本规则：

定义html标签的时候不需要引号， 即 let a = <div>hello react!</div> 便可以。

React要求每一个组件必须返回**一**个Top-Level的html元素。在书写多便签的时候必须包含根元素，并且允许自由缩进，即是对缩进没有要求。

例如：

let a = <div>AAA</div><span>BBB</span> 🡪不合法，必须有根元素

let a = <div>

<div>AAA</div>

<span>BBB</span>

</div> 🡪 合法并且缩进被允许， 并且可以在根元素的前后加括号表示整体

P.s React从16版本开始为我们提供了<React.Fragment>。<React.Fragment>的引入主要可以解决2个问题，第一个是使得语义更加清晰，不需要引入无实际意义的Top-level包裹元素。第二个原因是有些情况下无法添加一个包裹元素。例如一个组件的返回是多个<td>用于插入父组件中。我们如果像这个组件加入一个包裹元素，则不符合html表格的定义。

P.s 另外<React.Fragment></React.Fragment>也可以使用<></>代替

例如

render () {

return （

<React.Fragment> // 不需要引入无实际意义的Top-level包裹元素

<div>1</div>

<div>2</div>

<div>3</div>

</React.Fragment>

）

}

例如：

export function Summary(props){

//无法提供一个额外的Top-Level包裹元素而不违反html表格的定义，使用<React.Fragment>才可以解决

return <React.Fragment>

<td>A</td>

<td>B</td>

</React.Fragment>

}

书写**单标签**的时候，必须在结尾加上/

<img>不合法 一定要<img/>

<input>不合法 一定要<input/>

<br>不合法 一定要<br/>

指定class类型的时候， 要使用 className关键字， <div class=’aaa’></div>不合法，<div className=’aaa’></div>才合法.

->React的JSX使用className来指定HTML的元素的根本原因是，JSX最终会转为JS语言。JS语言会使用DOM API去进行创建元素，配置元素的class。而DOM API配置元素的class便是使用element.className. 所以JSX也使用className来进行配置html元素的class.

***在jsx中使用单花括号{}，将会认为花括号中的语法为javascript语法。JSX会对其进行渲染，运行求值。***

***例如 let b= <div>{a}</div> 🡪 a为js变量. 并且这个b所代表的不是普通的html标签，是React所构建的虚拟DOM.我们通常只在html标签中加入{}***

🡪

React开发模式

第一种是直接引入React.js启动React。并且引入react-dom.js启用虚拟DOM， 引入babel使得可以在<script>中写入jsx语言。在Html中引入的顺序必须为react.js, react-dom.js, babel里面的browser.js

另一种一种是脚手架模式基于webpack.

🡪

一个重要的react渲染方法ReactDOM.render(插件，目标)

例如：

<script type=”text/babel”> 🡪不使用text/javascript, 如果申明javascript那么浏览器将认为是javascript语法，将会报错。使用babel表示这不是普通的js语法

ReactDOM.render( 🡪 React渲染方法

<h1>hello world! </h1> 🡪 插入的内容

documents.getElementById(‘xx’)) 🡪插入的目标

🡪

Jsx的style里面使用组件样式对象， 所以指定时候需要两层花括号。第一层告诉JSX使用javascript解析，第二层表示这是个对象

例如 {{‘background’:’red’}}， 第一层的花括号是告诉jsx这将会是一个js代码，第二层的花括号则是js中的对象定义中的花括号。

🡪

Jsx中对控件事件的指定使用的是驼峰命名法（也就是第一个单词小写，以后的每个单词的首字母大写）

例如 onClick, onMouseOver

🡪

JS中的两种面向对象

第一种是ES6规则之前的：不是严格的OOP语言，定义类的时候，通过Function关键字来定义,通过原型链的方法来实现继承

第二种是ES6规则，也就是可以使用class来定义类，然后在class中定义有初始化方法constructor，以及定义各种方法。使用extends关键字来继承等（并且在子类的constructor函数中必须调用super（）父类方法先）。使用new来新建对象。需要注意的是ES6虽然采用了类似于JAVA, C++这些class的关键字，使得更加容易被接受，但是这些语法等同于语法糖，真正Javascript底层上的实现还是通过原型链来实现的。

例子：

class Person {

constructor(name, age){ 🡪定义初始变量

this.name = name;

this.age = age;

}

say(){ 🡪 申明方法的时候不需要function关键字。定义为实例方法，需要实例化以后使用

console.log ('my name is ' + this.name + ', ' + this.age + ' years old');

}

static print(){ 🡪定义静态方法，不需要实例化类便可以使用

return 'hello world';

}

}

🡪ES6中，没有静态属性，只有静态方法

🡪类中的方法互相调用，需要使用this.方法名()

🡪extends可以继承方法和私有属性

🡪React.js推荐使用ES6规则编写组件

🡪

基本的例子使用React的组件

<div id=’app’></div> 🡪定义一个HTML用于渲染的元素

<script type=”text/babel”>

Class MyComponents extends React.Component{ 🡪 定义一个组件，并且**组件名的首字母一定要大写**， 并且最主要是继承React中的组件.组件中定义render方法，在render方法中return需要显示的内容。

render(){ 🡪 render方法用于组件渲染

return (<div>Hellp React</div>) 🡪用于返回此组件渲染的内容

}

}

ReactDOM.render(<MyComponents/>, app) 🡪使用此组件对id为app的html元素进行渲染，并且单标签闭合

</script>

🡪

Javascript中this的多变性

--普通函数中的this表示调用此函数时的对象，即是指向调用者。

--this指向的可变性。

--如果javascript内嵌在html中，例如onClick=”…”那么此时的调用者是Window和在外部写是一样的。

当在全局作用域时，this指向全局；

当在某个对象中使用this时，this指向该对象；

当把某个对象的方法赋值给另外一个对象时，this会指向后一个对象。

例如:

var a = {

user:"管理员",

fn:function(){

console.log(this.user);

}

}

var b = a.fn;

b (); //undefined

当b=a.fn的时候，我们把a对象的方法fn赋值给方法对象b, 在执行的时候，由于是window对象调用的，所以this将向全局，也就是window对象，而window对象中没有user, 所以为undefined.

例子2:

var a =5;

var b= {

a: 12,

show () {

alert(this.a);

}

};

b.show(); 🡪为12因为是b调用，那this就是b,所以为12

c = b.show;

c(); 🡪 现在是window调用，所以为5

c = b.show.bind(b);

c(); 🡪现在为window调用，但是改变了this为b,所以为12

--this的使用场合有：在全局环境中使用；在构造函数中使用，在对象的方法中使用。

--this的使用注意点，最重要的一点就是要避免多层嵌套使用this对象。

为了固定this的指定，我们可以使用call, apply 以及bind方法来规定this的指向。

Call🡪**第一个参数为this所应该指向的对象，**剩下的参数为函数所需要的参数，以形参的方式“x，x, x..”指定，并且马上执行。

Apply🡪**第一个参数为this所应该指向的对象**，剩下的参数为函数所需要的参数，以数组[]方式指定，并且马上执行。

Bind🡪**第一个参数为this所应该指向的对象**，剩下的参数才为函数所需要的参数，以形参的方式“x，x, x..”指定，并且不会马上调用函数，而是返回的是指定过this作用域的函数。

🡪

ReactDOM.render(<MyComponents a=’12’/>, app)

这样a=12将会传入组件当中，React将其看作是key=a, value=12的参数。存在参数中的props的字典当中。使用this.props.a获取。

🡪

在react.js中自定义组件的时候，我们会在render写入代码，在组件中定义其他方法。需要注意的是在定义的方法中，和render当中，this默认的指向不同，有时候需要在调用其他定义方法的时候使用call,apply,bind方法改变指向。

🡪

JavscriptScirpt的两大开发IDE, WebStrom由Jetbrains开发，付费软件以及Atom由Github上的开源软件。

在本地的IDE调试JS代码的时候，需要node.js的支持。

🡪

React组件中props和state的区别：

--props是一个从外部传进组件的参数，主要作用就是从父组件向子组件传递数据，它具有可读性和不可变性，只能通过外部组件主动传入新的props来重新渲染子组件，否则子组件的props以及展现形式不会改变。

--props主要有两种类型。一种是Data Props，父组件用于传递某些变量给子组件。第二种是Function Props, 父组件定义的回调函数用于传递给子组件调用。

--

state的主要作用是用于组件保存、控制以及修改自己的状态，它只能在constructor中初始化，它算是组件的私有属性，不可通过外部访问和修改，只能通过组件内部的this.setState({})来修改，修改state属性会导致组件的重新渲染。或者this.setState = {},或者通过this.state.field = xxx 修改state中属性，但是这样不会重新渲染组件,并且React不推荐使用this.state.field去改变state.

state不同于props的一点是，state是可以被改变的。

🡪

Javascript中的箭头函数主要有两个作用：

第一是箭头函数可以相当于申明一个**匿名函数**，使其更加简洁的语法。

***第二是由箭头函数所申明的函数，不会改变this本来的绑定。***

🡪

React获取元素的其中三种方法：

--获取事件源元素，使用事件对象e.target属性

--获取非事件源元素，通过React提供的ref机制。

例如，<input ref=”input\_ref” />, 然后通过this.refs返回的字典形式中，以this.refs.input\_ref获取

--使用原生的Javascript获取函数获取

例如函数，getElementbyId, getElementByTagName,等等

🡪

事件冒泡：底层元素的事件冒泡将事件传到父类组件

首先在react中的事件对象e,是经过React封装过的，不是原生的事件对象。而e.nativeEvent才是原生的事件对象。所以阻止事件冒泡时，通过e.nativeEvent.stopImmediatePropagation()

🡪

我们在JSX语法范围中创建的HTML元素，是经过React封装过的，不是最原生HTML DOM元素，是React的虚拟DOM.

例如：

在JSX范围中 <li>ABC</li> 在浏览器中可以看到这个元素是React元素(Symbol(react.element))，而不是原始的HTMLDOM.

🡪

react组件中的render方法将会在几种情况调用：

1. 首次加载

2. setState({})改变组件内部state。

注意： 此处是说通过setState({})方法改变。通过this.setState = {}, 或者this.state.field都不会调用redner方法

3. 接受到新的props, 例如从父组件更新的子组件的props值

4. 调用this.forceUpdate

🡪

当我们在componentWillMount以及componentDidUpdated中使用改变状态的方法时候，注意是否会引入死循环。

在这两个方法中使用this.state.field是可以的，因为不会调用render函数，使用this.setState = {}也是可以的因为也不会调用render方法。但是使用this.setState({})是不行的，因为产生死循环。

死循环为：componentWillMount或者compoenentDidUpdated -> this.setState({})->render() -> componentWIllMount/compomentDidUpdated…..

🡪

判断是否可以做为一个state的条件:

  1、变量如果是通过props从父组件中获取，就不是一个状态，它会是一个props

  2、如果这个变量可以通过其他的状态state或者属性props 通过数据处理得到，就不是一个状态

　　3、如果变量在render中没有使用到，那就不是一个state

  4、变量在整个生命周期中都保持不变时，也不是一个状态

其实使用的时候最多的使用到的就是state和props，他们两个是有很大的区别的，最主要的区别就是：

　　State是可变的，是组件内部维护的一组用于反映组件UI变化的状态集合；

　　而Props对于使用它的组件来说，是只读的，要想修改Props，只能通过该组件的父组件修改。在组件状态上移的场景中，父组件正是通过子组件的Props,传递给子组件其所需要的状态。

　　在使用state的时候， 如果我们企图直接修改state中的某一个值之后立即直接打印（使用）他，就会发现，他其实并没有改变。

　　就像下面的例子，企图通过点击事件之后就使用修改之后的state的值，但是会发state中的并没有被立即修改，还是原先的值，那是因为setState就相当于是一个异步操作，不会立即被修改



🡪

React在html元素中加入**自定义的属性**的时候，遵循html5的方法，需要全部小写，并且采用前缀为data-自定义属性名。

例如：

非法：<input myindex=”AAA” type=”button” value=”Click me”/>

合法：<input data-myindex=”AAA” type=”button” value=”Click me”/>

🡪

**函数多次bind以后，this的指向只遵从第一次bind**.例如：

function say () {

alert(this.x);

};

var a = say.bind({x: 1});

var b = a.bind({x: 2});

这里的输出将会是1，不是2. 在bind了第一次以后，第二次的bind中的this不会被设置。

🡪

ReactJS中如果我们需要循环生成html元素，例如<a>, <li>等元素，我们需要为其制定key属性，这样在React才不会有警告each child in an array or iterator should have a unique "key" prop. 对于非循环的html元素，我们不需要指定其key值也可以。

🡪

如果我们在html中，想要渲染Javascript内容，我们需要花括号{变量名}, 如果没有加入花括号则会报错SyntaxError: Unterminated JSX contents

🡪

注意React的this.state={}和this.setState()的不同.this.state通常用于设置控件的初始化状态所以是将一个字典赋值给this.state={“A”:”B”}属性，而this.setState是更新状态，是调用setState方法，所以是将一个代表状态更新的字典赋值给setState方法, this.setState({“A”:”B”})。

🡪

React的事件指派原理，当我们向React中添加事件指派的时候，React会将事件处理器安放在document的根节点当中，然后当事件被触发的时候，React将事件给对应的控件并且当控件unmount的时候，React也会自动的把事件触发器移除。

React的事件指派采用的是驼峰命名发例如onClick而不是html中的onclick, onMouseMove而不是html中的onmousemove等。

常用的方法是我们在html标签中制定onClick = {this.xxx.bind(this)}，然后在控件中定义xxx事件处理方法，而不是直接在onClick中写事件处理函数逻辑。

🡪

JSX语言中需要注意的几点：

**--Tag Attributes are camel cased所有的标签属性都必须遵循驼峰命名法**

例如 HTML中<input type=”text” maxlength=”30” />

JSX中是<input type=”text” maxLength=”30” />

**--All elements must be balanced（所有的标签都必须闭合）**

例如 HTML中可以<br>. <img>

JSX中必须<br/>, <img/>

**--Atrribute Names are based on the DOM API(标签的属性名将根据JSX 虚拟DOM更改)**

例如 HTML中<div id=’box” class=”some-class”></div>

JSX中<div id=’box’ className=”some-class”></div>

**--JSX中返回中必须有一个根节点**

例如：render方法中return (<h1>AAA</h1><h2>BBB</h2>)是非法的

Render方法中return (<div>(<h1>AAA</h1><h2>BBB</h2>)</div>)才是合法的，通过div包裹成只有一个根节点

**--不允许if关键字出现在jsx语言的html标签当中**

例如 <div className={if (condition) {“salutation”}}>Hello</div>

解决办法有两种：

--使用三元操作符号，如果逻辑并不复杂，那么我们可以使用三元操作符号，从而不出现if关键字

<div className= {condition? “salutation”:””}>Hello</div>

--将判断逻辑移出标签中判断。用于比较复杂的逻辑判断。

render () {

let className=””;

If(condition) { # 如果condition为false时候，React也不会产生空的className

className=” salutation”;

}

return (

<div className={className}>Hello</div>

)

}

**--使用javascript的字典来定义样式，同样的样式中的属性采用驼峰命名法则**

例如 <div style={{‘width’:100, ‘backgroundColor’:’#ee9900’}} >Hello World</div>

也可以

let styleDict = {

‘width’:100,

‘backgroundColor’: #ee9900

}

<div style={styleDict}>Hello World</div>

🡪

React的JSX语法的注释格式为 {/\* 注释内容 \*/}

32.

React中使用Markdown语言范例：

首先什么是Markdown语言：

Markdown是一种「标记语言」，通过在纯文本中加入特殊的标记，便可以让文字渲染出特定的格式。并且在跨平台上的支持度很高，可以快速的打包成HTML文档等格式。

例如规则是：

|  |  |
| --- | --- |
| #，##，###，####，#####，###### | 一级标题，二级标题，三级标题，四级标题，五级标题，六级标题 |
| ##  -  -  -  … | 无序列表  项1  项2  项3 |
| ##  1.。。。  2.。。。  3.。。。 | 有序列表  项1  项2  项3 |
| [显示文本](链接地址) | 插入超链接 |
| ！[](链接地址) | 插入超链接所指向的图片 |
| >引用 | 插入引用的内容 |
| \*文字 | 该文字显示斜体效果 |
| \*\*文字\*\* | 该文字为粗体效果 |
| | **Tables** | **Are** | **Cool** | | --- | --- | --- | | col 3 is | right-aligned | $1600 | | col 2 is | centered | $12 | | zebra stripes | are neat | $1 |   表格：  | Tables | Are | Cool |  | ------------- |:-------------:| -----:|  | col 3 is | right-aligned | $1600 | 成为  | col 2 is | centered | $12 |  | zebra stripes | are neat | $1 | | |

并且还可以进行流程图等高级图标的绘制。

在React中我们使用markdown语言时， 首先需要安装marked包，我们可以在package.json中安装marked包，然后在JSX中引入。最后需要注意的是，由于React处于安全性的考虑，默认是不允许例如markdown这样动态生成内容的语句，因为这样有可能遭到插入式语句的攻击。我们需要使用dangerouslySetInnerHtml={{\_\_html: markdown的内容}}来让React特殊允许的生成。

🡪

**React的Key和Ref**

**Key属性:** React在更新状态的时候，有React自己的算法让自己的虚拟DOM最快更新。但是为了达到最好的效果，我们需要在进行循环生成节点的时候，为**每一个最外层的节点**提供key属性来帮助React识别(如果不提供，React也会给出警告) 。

需要Key的原因时，以后更新组件的时候。如果key相同则React可以认为是更新。如果是Key新增则是增加新的元素，如果Key不存在了则删除该元素。

例如：

render () {

let tasks = this.props.tasks.map(task=> (

<li key={task.id} ….> //为每一个最外层的节点提供key属性

<input type=” checkbox” defaultChecked=…. /> //非最外层可以不需要指定key属性

</li>

));

}

**Ref属性：**一般来说，我们都会使用React的虚拟DOM来操作DOM, 在极少情况下我们会去操作原生的DOM. 如果特殊情况下我们想获取原生的DOM, React使用this.refs.ref值来让我们获取原生的DOM对象。

例如：

<input ref=” myInput”>

我们可以使用this.ref.myInput来获取到input的原生DOM对象。

🡪

组件的属性申明验证(Props Validation)

由于我们的组件是本着重用的原则，所以当被别人复用或者再次检测组件的时候，我们希望通过某些方法规定组件需要哪些属性以及类型，或者为组件某些属性在非指定的情况下提供默认值。

(在React15.5以后propsTypes提供在props-types库中)

**propTypes（申明组件属性的类型以及要求）**

例如：

class Greeter extends Component {

render () {

return(<h1>{this.props.salutation}</h1>)

}

}

Greeter（组件类名）.propsTypes= {

salutation: PropTypes.string.isRequried

//表示这个组件需要salutation这个属性，类型为字符串并且必须指定，如果创建组件的时候没有指定将会有警告

salutation2: PropsTypes.string

//表示salutation2作为组件属性，类型为字符串但是非必要

}

一些常用的propType规定

|  |  |
| --- | --- |
| PropTypes.array | 属性必须是数组 |
| PropTypes.bool | 属性必须是布尔值 |
| PropTypes.func | 属性必须是方法 |
| PropTypes.number | 属性必须是数字 |
| PropTypes.object | 属性必须是对象 |
| PropTypes.string | 属性必须是字符串 |
| PropTypes.oneofType | 属性必须是所列举的类型范围之一 |
| PropTypes.arrayOf | 属性必须是数组并且数组中的元素类型为指定的类型 |
| PropTypes.objectof | 属性必须是对象且其所含的属性必须是指定类型 |
| PropTypes.shape | 属性必须是对象，且必须包含所列举出的属性 |
| PropTypes.element | 属性必须是React元素 |
| PropTyoes.instanceof | 属性必须是所指定类的实例对象 |

除此之外，我们还可以**自定义propTypes**的检查,规则为将提供三个参数. props为所有属性的字典， propName验证的属性名， componentName组件名。当符合规则的时候propTypes不需要返回，当不符合规则的时候，应该返回Error对象。

例如:

let titlePropType = (props, propName, componentName) => {

if (props[propName] {

let value = props[propName]

if (typeof value! == ‘string’ || value.length> 80){ //条件为字符串并且长度小于80

return new Error(`${proName} in ${componentName} is longer than 80`) // ES6的模板字符串写法

}

}

}

**defaultProps（申明组件属性的默认值，如果父组件不提供值）**

Greeter.defaultProps = {

Salutation3: “Hello World” //如果父组件不提供Greeter组件的Salutation3属性，那么默认值为Hello World

}

🡪

React设计中其中很重要的一个方面是，我们将为控件设置多少个state, 以及将这些state放在哪个空间当中。

**识别state:**

State将设置在引起变化的起点，它的变化将引起其他变化。例如一个变量变化，将引起显示的文字不同， 搜索出来的结果不同。那么该变量作为一个state.

**放置state:**

State通常放置在公共组件当中，放置在层次化的上层。例如state变量，将引起组件1，组件2的变化，那么该state放置在包裹组件1，组件2的父组件当中，通过props传给子组件1，子组件2.同时子组件1，子组件2也通过父组件提供的回调函数在自身组件中调用，反向改变父组件中的这个state.

🡪

**React中的一个重要原则，state只应该被拥有它的组件更改**，如果是作为props传到其他组件进行修改的时候，也应该在子组件中使用拥有该state的父组件所提供的回调函数进行修改。

🡪

React中的一个设计原则，当我们的组件需要从远端进行获取数据的时候，例如restful等方法，我们一般会设计一个包裹控件 (container component)。这个控件包裹其他控件，并且不含有任何的业务逻辑。它的逻辑只是调用restful来获取数据，作为自己的state然后传递给其他业务逻辑控件作为他们的props.

🡪

Babel在React中承担了两个作用. Babel会将JSX语法变为浏览器所识别的Javascript语法， 另一方面Babel会将ES6的语法变为ES5语法，从而使对ES6支持度不好的浏览器也可以执行。需要注意的是Babel转换的是Es6的语法，但是并不转换ES6的API.需要转换ES6的API时，需要使用Babel-pollyfill.

ES6的语法和API区别：

(x => x \* 2) (1); //这是ES6的语法，新添的箭头写法

var b = Array.of(1,2,4); //这是ES6 Array新添的API

所以我们使用React开发前端，我们可以在开发的时候使用ES6的语法和API, 使用React的JSX语言， 当Babel转移以后给浏览器的是ES5规格的纯Javascript语法。 所以大部分甚至只支持ES5的浏览器都可以运行。

🡪

在React中，我们不应该直接使用this.state.field = xx来直接改变state的值。但是有些时候，我们会不小心使用其他方法，其效果也等同于直接使用了this.state.field的方式来直接改变了控件的state.

例如:

Constructor () {

this.state = {

info = {

“name”: Hui

}

}

}

let newInfo = this.state.info

newInfo.name = “NewName” // 这步实际上等于直接更改， 相当于this.state.info.name = “NewName”

this.setState({info:info});

由于在Javascript中对象，数组，字典都是引用类型。所以我们新的变量名都是指向同一个对象， 我们使用新的变量名进行改变的时候，实际上便等于直接对控件的状态进行直接的更改。

解决方法：

对于这些引用类型的state进行更改的时候， 我们需要使用ES6提供的non-destrutive, immuntable方法（例如Object.assign, filter, concat等）， 这些方法的本质是，不改变原对象并且该需要的更改应该在原对象一模一样的内容上并且返回。

例如上述例子：

//我们使用assgin 这个non-destructive, immuntable方法来更改

let newInfo = Object.assign({}, this.state.info, {“name”: “NewName”})

// assgin方法将所有对象逐一放入到目标对象当中， 并且当后面有相同的域指定时， 覆盖前面的值。

this.setState({“info”: newInfo}

这样我们通过获得全新的newInfo变量指向不同的对象， 便不会影响原控件中的状态并且通过this.setState方法来改变控件的状态。

**更新单层嵌套式状态**

如果状态是nested的结构，并且嵌套层数只有一层，则可以使用…或者Object.Assign进行更新。需要注意的是…或者Object.Assign都是浅复制，所以只能处理嵌套层数只有一层的情况。

例如；

this.state = {

someProperty: {

name: “AAA”

flag:true

}

}

当我们要更新someProperty中的flag为false的时候，

this.setState({ someProperty.flag: false }); // 非法，无法识别someProperty.flag

this.setState({someProperty: {flag:false}};//合法，但是name将会丢失

方式1:采用… ES6的扩展符号

this.setState({

someProperty: {

...this.state.someProperty,

flag: false,

},

});

方式2:采用Object.assign方式

this.setState({

someProperty: Object.assign({}, this.state.someProperty, {

flag: false

}),

});

**修改状态中的多层内嵌引用型变量(Nested Object)**

当我们修改state中的简单类型（数字， 字母等）， 或者非内嵌引用型变量的时候，我们可以通过直接修改或者non-destructive方法来避免调用this.state.field来修改。但是如果是内嵌引用类型变量，例如字典中的域对应另一个引用类型，我们要修改其中的值时，我们默认使用assign或者其他non-destructive方法时候，由于默认使用的都是浅层拷贝，所以还是会直接使用了this.state.field来进行修改。

我们可以使用深拷贝然后进行修改，虽然可行但是比较复杂操作。React为我们提供了add-ons package（update 包）来帮助我们去修改这样的内嵌引用型变量。

安装update包： npm install react-addons-update;

引用update包：import update from ‘react-addons-update‘;

Update包引用的update方法，需要两个参数，第一个参数是哪一个对象需要改变， 第二个参数是是一个对象，该对象说明了，哪里需要改变（描述数据结构）以及需要进行的操作关键字，以及改变以后的值。

例如：

Let newStudent = update (student, {grades: {$push: [‘A’]}}

// 从而对student对象中的第一层grades所对应的array的域加入新的元素” A”, 不影响原对象的情况下， 返回新的对象newStudent

Let newStudent = update (student, {grades: {$set: [‘A’,’A’,’B’]}})

//对student对象中的第一层grades所对应的array的域，设置值其值为” A”,”A”, “B”

Let newTicket = update (originalTicket, {

arrival: {

airport: {$set: “MCO”}})

//对originalTicket对象中的第一层arrival所对象的对象中的airport所对的值设置为” MCO”, 不影响原对象的情况下， 返回新的对象newTicket

Update方法的操作关键字：

|  |  |
| --- | --- |
| $push | 将元素放入数组尾部当中  Let initialArray = [1,2,3]  Let newArray = update (initialArray, {$push: [4]});  // newArray为[1,2,3,4] |
| $unshift | 将元素放入数组头部当中  Let initialArray = [1,2,3];  Let newArray = update (initialArray, {$unshift:[0]}}  // newArray = [0,1,2,3] |
| $splice | 讲元素切片或者从切片出插入元素代替  Let initialArray = [1,2,’a’];  Let newArray = update(initialArray, {$splice: [[2,1,3,4]]});// 从2开始切，切1个，加入元素3,4  //newArray为[1,2,3,4] |
| $set | 改变数值 |
| $merge | 将对象放入对象当中 |
| $apply | 将原对象的元素放入指定的方法当中,设置为方法的返回值  Let obj = {a:5, b:3}  Let newObj = update (obj, {b: {$apply:(value) =>value\*2}});  //newObj为{a:5, b:6} |

🡪

onChange={this.toggleTodo} 与onChange={this.toggleTodo()}的异同

因为onChange需要指定一个函数，所以this.toggleTodo是可以的传入一个function引用。 但是this.toggleTodo()是不行的，因为这个意思是调用函数。

另外onChange={this.toggleTodo}也适用于toggleTodo需要参数的时候，加入toggleTodo需要一个参数，那么我们通过这样将toggleTodo引用传递时，在调用的时候toggleTodo(参数1)传入参数也可以顺利调用。

🡪

一个组件的状态(states)和属性(props)：

原则是一个组件的状态只可以自己修改，并且通过自身的this.setstate({})来修改。一个组件的属性，不能自己直接修改。子组件的属性通过父组件传递的方法来修改。

🡪

关于React什么时候需要的是方法的定义，什么时候需要调用方法

<input …. onChange={(e)=> this.props.callback(e.target.checked) } …/>

# 表示onChange的时候，通过箭头函数整体**定义了一个方法, 并没有马上调用**。 这个方法由于是(e),所以传入参数e到方法体中，这个方法体只有一句话，便是调用this.props.callback方法，使用e.target.checked作为参数。

<input .. onChange={()=>this.props.callback(this.props.item)}/>

# 表示onChange的时候，通过箭头函数整体**定义了一个方法, 并没有马上调用**。 这个方法由于是(),所以没有传入参数到方法体中，这个方法体只有一句话，便是调用this.props.calback方法，使用this.props.item作为参数。

render = () => {

return (

Number of things {this.isEven()} #马上调用。并且使用返回值插入HTML当中。没有问题。

Number of things {() = >this.isEven()} #定了一个不改变this的方法， 这个方法的方法体只有一句话，调用isEven方法。但是这是定义方法不是马上调用，所以出错，我们需要返回值。只定义方法不调用是非法的在这里。

Number of things {this.isEven} # 定义方法，并没有马上调用。非法原因和上句相同。

)

}

🡪

this.setState({  
 count: this.state.count ++ # 应该改为count: this.state.count + 1只是读取this.state.count  
})

这样会得到非法的警告。Do not direct mutate the state. 因为我们使用了this.state.count++,这句话相当于this.state.count = this.state.count + 1.

🡪

React的虚拟DOM设计：

所有的 DOM 变动，都先在虚拟 DOM 上发生，然后再将实际发生变动的部分，反映在真实 DOM上，这种算法叫做 [DOM diff](http://calendar.perfplanet.com/2013/diff/) ，它可以极大提高网页的性能表现。

🡪

React使用循环生成元素，实现将元素放到**数组**当中。然后在render方法return的时候，使用{layout}, 也就是数组，每个元素都是虚拟DOM, 来集体生成。

另外在JSX中，<div>AAA</div>是React的虚拟DOM, 而”<div>AAA</div>”是字符串不是React 虚拟DOM不会生成HTML元素。

**for**(**let** i =0; i< **this**.props.squares.length; i = i+3){  
 layout.push(<div className="board-row" key={i}>  
 {**this**.renderSquare(i)}  
 {**this**.renderSquare(i+1)}  
 {**this**.renderSquare(i+2)}  
 </div>  
 );  
}

🡪

status = <div>;  
status += <p style={{"display":"inline"}}> {"Next Play: " + (**this**.state.xIsNext? "X": "O")}</p>;  
status += <button style={{"display":"inline"}}>Reverse Order</button>;  
status += </div>;

这样React无法正常工作。因为第一步status= <div>; 此时status的类型是React**虚拟DOM对象，不是字符串**。所以后续 status += 的拼接操作是无法成立的，字符串才可以这样。

正确写法是：

status = <div>  
 <p style={{"display":"inline"}}> {"Next Play: " + (**this**.state.xIsNext? "X": "O")}</p>  
 <button style={{"display":"inline"}}>Reverse Order</button>  
 </div>

🡪

ReactJS根据条件指定样式：

<button  
 className="square"  
 style = {**this**.props.winningcell? {"backgroundColor": "yellow"}:{}}  
 onClick={()=>**this**.props.onClick()}>  
 {**this**.props.value}  
</button>

正确的写法是style=。。。条件在后面，无论如何都会有style属性给ReactDOM.

而不是在条件中再去定是否有style属性，错误写法如下：

<button  
 className="square"  
 {**this**.props.winningcell? style = {"backgroundColor": "yellow"}:{}}  
 onClick={()=>**this**.props.onClick()}>  
 {**this**.props.value}  
</button>

🡪

Style属性对HTML元素的ReactDOM有效，而对于自定义的组件无效。

例如：

<button  
 className="square"  
 style = {**this**.props.winningcell? {"backgroundColor": "yellow"}: {}}  
 onClick= {() =>**this**.props.onClick()}>  
 {**this**.props.value}  
</button>

是有效的，对应HTML元素button.

**return** <Square  
 style = {**this**.props.winningcell? {"backgroundColor": "yellow"}: {}}  
/>;

而对于自己定义的React组件Sqaure, style效果只是props而已。

🡪

npx create-react-app [app名称] ， 创建一个React App里面并且预先安装一定JS包用于快速开发， 例如， react本身打包工具webpack, 转译器babel等。如果还需要其他包我们可以自行安装，例如bootstrap, redux, react-redux, react-router等。

🡪

React组件无状态函数式组件申明：

React组件的申明可以采用 class 组件名 extends React.Component的类的申明写法，除此以外我们还可以使用无状态函数式组件方式。它本质上就是一个常规的函数，接收一个 props 参数并返回一个元素。返回元素的方式便是JS函数的return方式。

# 例如：

function Item(props) {//接收props参数

return (

<div className='item'> //返回一个组件

{props.title}

<span

className='deleteItem'

onClick={props.remove(props.id)}

> x </span>

</div>

)

}

同时使用ES6 的箭头函数和解构，我们也可以这样编写：

const Item = ({id, title, remove}) =>

<div className='item'>

{title}

<span

className='deleteItem'

onClick={remove(id)}

> x </span>

</div>

函数式组件，有时也被称为无状态组件，没有任何生命周期方法，意味着每次上层组件状态发生变更时它们都会重新渲染，这就是因为缺少 shouldComponentUpdate 方法导致的。这也同样意味着您不能定义某些基于组件挂载和卸载的行为。

在函数式组件中既不能使用 this 关键字，也没有this.props或this.states或访问到 ref。

无状态函数式组件主要的设计概念是使得容器型组件 (container component) 和 展示型组件 (presentational component) 的概念就会变得非常清晰。通过将逻辑和数据处理与 UI 展示剥离开来，我们就可以避免在展示型组件中处理任何的状态。无状态函数式组件强制实施了这样的风格，因为您无论如何都没有办法处理本地状态了。它强制您将任何的状态处理移交至上层的组件树，而让下层的组件只做它们所做的——关注 UI 的展示。

P.s 无状态组件(stateless component)由于无自身状态，所以在给定props的前提下，生成的html元素都肯定是一样的。而相比之下，有状态组件(stateful component)在给定自身的props前提下，可以使用自己独立的state生成不同的html元素，所以生成的html元素可以不一样。

P.s函数式组件和有状态组件语法上的区别

函数式组件，使用return返回要生成的内容。有状态组件则是以render()方法中的return方式。

函数式组件由于没有this,使用props.xxx来使用props。有状态组件则是以this.props.xxx来使用props.

🡪

我们可以使用create-react-app来快速开发react项目。使用create-react-app会默认为我们创建node\_modules文件夹，public文件夹，src文件夹以及在此文件夹下的App.css, App.js, index.css, index.js以及项目文件夹下的package.json

node\_moduels为create-react-app默认安装的包，其中必须JS模块包括react.js, react-dom.js包，Babel, webpac, react-development web server等。

Public文件夹包括了我们稍后为项目添加的静态文件。

src文件夹下：

App.css 为此项目由开发者开发的最顶层组件的CSS文件。

App.js 为此项目由开发者开发最顶层组件代码

index.css 为默认样式。

index.js 为默认的js文件，同时也是webpack打包的入口。指定默认行为渲染APP组件到index.html, id为root的元素中。

项目文件夹下：

.gitignore为我们向Git保存代码默认忽略文件清单

package.json 配置本地npm配置。

P. s一般结构为一个页面就是一个整体的App组件.在App.js渲染出其他所有的子组件。在index.js中规定在某个地方渲染出一个App整体组件。

🡪

React项目使用静态文件的方式

如果不放在public文件夹下：我们可以在JS文件中进行引入。

例如

import “./App.css” # 从App.css引入，并且不需要在JS文件中引用css的内容，引入的时候无须命名

import reactLogo from “./logo.svg” #从logo.svg引入一张图片，由于需要在JS文件中的HTML标签中引用所以需要加入命名reactLogo

🡪引入的文件大小不超过10Kb,则该文件会被Webpack打包在bundle.js中。

🡪引入的文件超过10Kb,则该文件会在用户访问页面时，单独请求，并且文件名将自动加入checksum

🡪index.css会自动加入

如果放在public文件夹下则在JS文件中使用process.env.PUBLIC\_URL + 文件名获取

🡪

状态型组件应该有其构造方法constructor,并且构造方法接受一个参数props.并且在构造方法第一句中调用super(props),如果需要定义初始状态，也在constructor当中。

🡪

对于有状态组件我们使用React的this.setState({}) 来更新它的状态。需要注意的是React在我们调用this.setState({})的时候，并不总是马上更新。React根据自身情况，会将其异步自动分组的的更新。这有时会产生和我们预想结果不一样的情况。

例如：

情景1：依赖一个state的值，设置另一个state的值

handleClick = () => {

this.setState({counter: this.state.counter +1});

this.setState({hasButtonClicked: this.state.counter > 0});

}

由于React并不总是进行同步一次次的更新，所以执行handleClick时，设置hasButtonClicked的时候，设置counter可能还没有生效，导致hasButtonClicked并不会设为true.

解决办法：使用setState的回调函数设置

我们可以在setState方法中，传入回调函数作为其第二个参数。这个回调函数会在React设置完state以后才调用。

handleClick = () => {

this.setState ({counter: this.state.counter +1 },

() => this.setState({hasButtonClicked: this.state.counter > 0})); //采用回调函数

}

情景2: 多次的对同一个state进行更新

handleClick = () => {

for (let i=0; i< 5; i++) {

this.setState({counter: this.state.counter + 1});

}

}

由于React并不总是进行同步一次次的更新，所以产生的效果只会让this.state.counter增加1.

解决办法：

我们可以在调用setState方法时，传入回调函数[第一个参数为现在的state, 第二个参数为props,要求返回一个对象字典代表改变后的状态]作为其第一个参数。这样React便可以多次重复的对同一个state进行修改。

handleClick = () => {

for (let i=0; i<5; i++) {

this.setState((state, props) => {return (counter: state.counter + 1)});

}

}

🡪

<React.StrictMode>为React的严格模式，与 Fragment 一样，StrictMode 不会渲染任何可见的 UI。它为其后代元素触发额外的检查和警告。严格模式检查仅在开发模式下运行；它们不会影响生产构建。

需要注意的是开启严格模式会有可能引起某些副作用，例如生命周期会被调用两次，多次调用setState方法等。

🡪

React中为html元素指定事件监听器，当事件触发的时候，React将默认传入React类型event作为参数。而在普通父子组件传递方法作为props的时候，子组件调用父组件传递而来的属性方法时，不存在该事件对象。

🡪