[ReactDOM.render() 4](#_Toc21245)

[JSX 语法 4](#_Toc11414)

[组件 4](#_Toc22593)

[React.createClass 4](#_Toc22411)

[this.props.children 4](#_Toc4029)

[PropTypes 4](#_Toc20006)

[获取真实的DOM节点 5](#_Toc606)

[state 5](#_Toc27144)

[表单 6](#_Toc28168)

[Ajax 6](#_Toc6465)

[@connect 装饰器 6](#_Toc31063)

[Redux-saga 6](#_Toc23944)

[JSX 简介 6](#_Toc26248)

[在 JSX 中使用表达式 7](#_Toc8700)

[JSX 属性 7](#_Toc18101)

[JSX 代表 Objects 7](#_Toc1022)

[元素渲染 7](#_Toc30649)

[将元素渲染到 DOM 中 8](#_Toc20464)

[更新元素渲染 8](#_Toc19073)

[组件 & Props 8](#_Toc14556)

[函数定义/类定义组件 8](#_Toc23699)

[组件渲染 8](#_Toc19017)

[Props的只读性 8](#_Toc18186)

[State 9](#_Toc19916)

[为一个类添加局部状态 9](#_Toc5736)

[正确地使用状态 9](#_Toc27162)

[生命周期函数 9](#_Toc23010)

[getDefaultProps 9](#_Toc12193)

[getInitialState 9](#_Toc19983)

[componentWillMount 10](#_Toc9972)

[render 10](#_Toc1578)

[componentDidMount 10](#_Toc22413)

[shouldComponentUpdate 10](#_Toc20546)

[componentWillUpdate 10](#_Toc12726)

[componentDidUpdate 10](#_Toc13327)

[componentWillReceiveProps 10](#_Toc23700)

[componentWillUnmount 11](#_Toc10565)

[事件处理 11](#_Toc12710)

[向事件处理程序传递参数 11](#_Toc18837)

[条件渲染 12](#_Toc14656)

[阻止组件渲染 12](#_Toc9160)

[列表 & Keys 12](#_Toc22465)

[Keys 12](#_Toc16532)

[用keys提取组件 12](#_Toc6551)

[表单 13](#_Toc12047)

[受控组件 13](#_Toc15440)

[textarea 标签 13](#_Toc24063)

[select 标签 13](#_Toc17738)

[多个输入的解决方法 13](#_Toc28752)

[状态提升 13](#_Toc15036)

[组合 vs 继承 13](#_Toc28268)

[React理念 14](#_Toc31279)

[定义 UI 状态的最小(但完整)表示 14](#_Toc861)

[深入 JSX 14](#_Toc16291)

[指定 React 元素类型 14](#_Toc11217)

[指定属性值 14](#_Toc19185)

[子代 15](#_Toc11906)

[传递子代 15](#_Toc15659)

[使用 PropTypes 进行类型检查 16](#_Toc12428)

[限制单个子代 16](#_Toc22023)

[属性默认值 16](#_Toc3505)

[静态类型检查 16](#_Toc14743)

[Flow 16](#_Toc7079)

[在项目中添加 Flow 16](#_Toc26850)

[从编译过的代码中剥离 Flow 语法 17](#_Toc30940)

[运行 Flow 17](#_Toc11288)

[添加 Flow 类型注释 17](#_Toc19033)

[TypeScript 17](#_Toc25451)

[在一个项目中添加 TypeScript 18](#_Toc10659)

[配置 TypeScript 编译器 18](#_Toc7141)

[运行 TypeScript 18](#_Toc19152)

[菜鸟教程总结 18](#_Toc28588)

[陕西项目整理 19](#_Toc25647)

[require.ensure() 19](#_Toc13709)

[routes-indexRoute 20](#_Toc27693)

[配置路由 20](#_Toc25037)

[安装 20](#_Toc10078)

[基本用法 20](#_Toc19084)

[嵌套路由 21](#_Toc4049)

[path 属性 21](#_Toc21571)

[通配符 21](#_Toc10272)

[IndexRoute 组件 22](#_Toc28445)

[Redirect 组件 22](#_Toc8151)

[IndexRedirect 组件 22](#_Toc21572)

[Link 23](#_Toc7201)

[IndexLink 23](#_Toc10832)

[histroy 属性 23](#_Toc20193)

[表单处理 23](#_Toc12926)

[路由的钩子 23](#_Toc6056)

# ReactDOM.render()

将模板转为 HTML 语言，并插入指定的 DOM 节点。

ReactDOM.render(

<h1>Hello, world!</h1>,

document.getElementById('example') //插入到id=example中

);

# JSX 语法

允许 HTML 与 JavaScript 的混写。

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。

JSX 允许直接在模板插入 JavaScript 变量。如果这个变量是一个数组，则会展开这个数组的所有成员

# 组件

React 允许将代码封装成组件（component），然后像插入普通 HTML 标签一样，在网页中插入这个组件。

# React.createClass

生成一个组件类。使用标签时生成组件类实例。

所有组件类都必须有render 方法，用于输出组件。

组件类的首字母必须大写，否则会报错。

组件类只能包含一个顶层标签(顶层标签不能有平行标签)，否则会报错。

组件的用法与原生的 HTML 标签完全一致，可以任意加入属性(使用时传值)；组件的属性在组件类的 this.props 对象上获取。

注意：保留字class 属性需要写成 className ，保留字for 属性需要写成 htmlFor 。

# this.props.children

表示组件的所有子节点。

如果当前组件没有子节点，它就是 undefined ;

如果有一个子节点，数据类型是 object ；

如果有多个子节点，数据类型就是 array

React.Children.map(this.props.children, function (child){ } ) 来遍历子节点。

# PropTypes

因组件的属性可以接受任意值；使用PropTypes属性验证组件实例的属性是否符合要求。

var MyTitle = React.createClass({

propTypes: {

title: React.PropTypes.string.isRequired,

},

render: function() {

return <h1> {this.props.title} </h1>;

}

});

getDefaultProps 方法可以用来设置组件属性的默认值。

var MyTitle = React.createClass({

getDefaultProps : function () {

return {

title : 'Hello World'

};

},

render: function() {

return <h1> {this.props.title} </h1>;

}

});

# 获取真实的DOM节点

ref 属性

var MyComponent = React.createClass({

handleClick: function() {

this.refs.myTextInput.focus();

},

render: function() {

return (

<div>

<input type="text" ref="myTextInput" />

<input type="button" value="Focus the text input" onClick={this.handleClick} />

</div>

);

}

});

# state

状态；getInitialState，setState；通过handleClick等调用setState改变state。

# 表单

通过this.state.value获取。通过onChang调用handleChange实现。

# Ajax

组件的数据来源，通常是通过 Ajax 请求从服务器获取，可以使用 componentDidMount 方法设置 Ajax 请求，等到请求成功，再用this.setState 方法重新渲染 UI。

# @connect 装饰器

@connect(({ chart, loading }) => ({

chart,

loading: loading.effects['chart/fetch'],

}))

@connect，dva 所封装的 react-redux 的 @connect 装饰器

list，绑定到此model的redux store。app.state.list。

loading，app.state.loading，来源是 src/index.js 中调用的 dva-loading；app.use(createLoading());。

payload 向 store 进行了一个类型为 chart/fetch 的 dispatch；src/models/chart.js 中就可以找到具体的对应操作。

# Redux-saga

是Redux的一个中间件，主要集中处理react架构中的异步处理工作，被定义为generator(ES6)的形式，采用监听的形式进行工作。

Effect 是一个 javascript 对象，可以通过 yield 传达给 sagaMiddleware 进行执行在， 如果我们应用redux-saga，所有的 Effect 都必须被 yield 才会执行。

put，触发某个action， 作用和dispatch相同：

yield put({ type: 'CLICK' });

call，有阻塞地调用 saga 或者返回 promise 的函数，只在触发某个动作。

状态与属性十分相似，但是状态是私有的，完全受控于当前组件。

使用类就允许我们使用其它特性，例如局部状态、生命周期钩子。

# JSX 简介

JSX 用来声明 React 当中的元素。

const element = <h1>Hello, world!</h1>;

因为 JSX 的特性更接近 JavaScript 而不是 HTML , 所以 React DOM 使用 camelCase 小驼峰命名 来定义属性的名称，而不是使用 HTML 的属性名称。

例如，class 变成了 className，而 tabindex 则对应着 tabIndex。

## 在 JSX 中使用表达式

可以任意地在 JSX 当中使用 JavaScript 表达式，在 JSX 当中的表达式要包含在大括号里。

推荐在 JSX 代码的外面扩上小括号，这样可以防止分号自动插入 的 bug。

## JSX 属性

你可以使用引号来定义以字符串为值的属性：

const element = <div tabIndex="0"></div>;

也可以使用大括号来定义以 JavaScript 表达式为值的属性：

const element = <img src={user.avatarUrl}></img>;

## JSX 代表 Objects

Babel 转译器会把 JSX 转换成一个名为 React.createElement() 的方法调用。

const element = (

<h1 className="greeting">

Hello, world!

</h1>

);

// 注意: 以下示例是简化过的（不代表在 React 源码中是这样）

const element = {

type: 'h1',

props: {

className: 'greeting',

children: 'Hello, world'

}

};

# 元素渲染

元素是构成 React 应用的最小单位。

元素用来描述你在屏幕上看到的内容：

const element = <h1>Hello, world</h1>;

与浏览器的 DOM 元素不同，React 当中的元素事实上是普通的对象，React DOM 可以确保 浏览器 DOM 的数据内容与 React 元素保持一致。

## 将元素渲染到 DOM 中

ReactDOM.render()

## 更新元素渲染

React 只会更新必要的部分

React DOM 首先会比较元素内容先后的不同，而在渲染过程中只会更新改变了的部分。

# 组件 & Props

组件从概念上看就像是函数，它可以接收任意的输入值（称之为“props”）。

## 函数定义/类定义组件

使用JavaScript函数

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

使用 ES6 class

class Welcome extends React.Component {

render() {

return <h1>Hello, {this.props.name}</h1>;

}

}

## 组件渲染

当React遇到的元素是用户自定义的组件，它会将JSX属性作为单个对象传递给该组件，这个对象称之为“props”。

组件的返回值只能有一个根元素。

组件名称必须以大写字母开头。

建议从组件自身的角度来命名props，而不是根据使用组件的上下文命名。

## Props的只读性

无论是使用函数或是类来声明一个组件，它决不能修改它自己的props。

所有的React组件必须像纯函数那样使用它们的props。即不能修改输入值props。

# State

状态与属性十分相似，但是状态是私有的，完全受控于当前组件。

定义为类的组件特性：局部状态，、生命周期钩子。

## 为一个类添加局部状态

添加一个类构造函数来初始化状态 this.state。类组件应始终使用props调用基础构造函数。

在生命周期函数中调用setState()更新this.state，并调度UI更新。

虽然 this.props 由React本身设置以及this.state 具有特殊的含义，但如果需要存储不用于视觉输出的东西，则可以手动向类中添加其他字段。如果你不在 render() 中使用某些东西，它就不应该在状态中。

## 正确地使用状态

不要直接更新状态，构造函数是唯一能够初始化 this.state 的地方。

this.state.comment = 'Hello'; // Wrong，不会重新渲染组件

状态更新可能是异步的

this.props 和 this.state 可能是异步更新的，你不应该依靠它们的值来计算下一个状态。

使用第二种形式的 setState() 来接受一个函数而不是一个对象。 该函数将接收先前的状态作为第一个参数，将此次更新被应用时的props做为第二个参数

this.setState((prevState, props) => ({

counter: prevState.counter + props.increment

}));

状态更新合并

当你调用 setState() 时，React 将你提供的对象合并到当前状态。

# 生命周期函数

<https://www.cnblogs.com/gdsblog/p/7348375.html>

## getDefaultProps

object getDefaultProps()；对象被创建之前执行，只执行一次。执行过一次后，被创建的类会有缓存，映射的值会存在this.props,前提是这个prop不是父组件指定的。

## getInitialState

object getInitialState()；控件加载之前执行，返回值会被用于state的初始化值

## componentWillMount

void componentWillMount()；执行一次，在初始化render之前执行

## render

ReactElement render()

## componentDidMount

void componentDidMount()；在初始化render之后只执行一次，在这个方法内，可以访问任何组件，componentDidMount()方法中的子组件在父组件之前执行。

## shouldComponentUpdate

boolean shouldComponentUpdate(object nextProps, object nextState)；这个方法在初始化render时不会执行，当props或者state发生变化时执行，并且是在render之前，当新的props或者state不需要更新组件时，返回false。通过this.props、this.state获取当前props、state.

## componentWillUpdate

void componentWillUpdate(object nextProps, object nextState)；当props和state发生变化时执行，并且在render方法之前执行，当然初始化render时不执行该方法，需要特别注意的是，在这个函数里面，你就不能使用this.setState来修改状态。这个函数调用之后，就会把nextProps和nextState分别设置到this.props和this.state中。执行后调用render()更新。

## componentDidUpdate

void componentDidUpdate(object prevProps, object prevState)；组件更新结束之后执行，在初始化render时不执行。类似componentDidMount()。

## componentWillReceiveProps

void componentWillReceiveProps(object nextProps)；当props发生变化时执行，初始化render时不执行，在这个回调函数里面，你可以根据属性的变化，通过调用this.setState()来更新你的组件状态，旧的属性还是可以通过this.props来获取,这里调用更新状态是安全的，并不会触发额外的render调用

## componentWillUnmount

void componentWillUnmount()；当组件要被从界面上移除的时候，就会调用componentWillUnmount(),在这个函数中，可以做一些组件相关的清理工作，例如取消计时器、网络请求等

# 事件处理

React事件绑定属性的命名采用驼峰式写法

如果采用 JSX 的语法你需要传入一个函数作为事件处理函数

<button onClick={activateLasers}>

Activate Lasers

</button>

不能使用返回 false 的方式阻止默认行为；必须明确的使用 preventDefault。

HTML：

<a href="#" onclick="console.log('The link was clicked.'); return false">

Click me

</a>

React：

function ActionLink() {

function handleClick(e) {

e.preventDefault();

console.log('The link was clicked.');

}

return (

<a href="#" onClick={handleClick}>

Click me

</a>

);

}

## 向事件处理程序传递参数

通过 arrow functions为事件处理函数传递参数。

<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>

通过 Function.prototype.bind 为事件处理函数传递参数。

<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>

参数 e 作为 React 事件对象将会被作为第二个参数进行传递。箭头函数方式，事件对象必须显式的进行传递，bind 的方式，事件对象以及更多的参数将会被隐式的进行传递。

通过 bind 方式向监听函数传参，在类组件中定义的监听函数，事件对象 e 要放在最后。

# 条件渲染

使用 JavaScript 操作符 if 或条件运算符来创建表示当前状态的元素，然后让 React 根据它们来更新 UI。

if，&&，三木运算符？。

## 阻止组件渲染

让 render 方法返回 null 而不是它的渲染结果即可实现。

组件的 render 方法返回 null 并不会影响该组件生命周期方法的回调。例如，componentWillUpdate 和 componentDidUpdate 依然可以被调用。

# 列表 & Keys

通过map()方法，将数组、Iterator接口的对象映射为列表。

const numbers = [1, 2, 3, 4, 5];

const doubled = numbers.map((number) => number \* 2);

console.log(doubled);

## Keys

Keys可以在DOM中的某些元素被增加或删除的时候帮助React识别哪些元素发生了变化。

key最好是这个元素在列表中拥有的一个独一无二的字符串。通常使用数据的id作为元素的key。

### 用keys提取组件

元素的key只有在它和它的兄弟节点对比时才有意义。

提取出一个ListItem组件，你应该把key保存在数组中的这个<ListItem />元素上，而不是放在<li>元素上。

在map()方法的内部调用元素时，记得为每一个元素加上一个独一无二的key。

元素的key在他的兄弟元素之间应该唯一，不需要是全局唯一。

key会作为给React的提示，但不会传递给你的组件。如需要使用和key相同的值，将其作为属性传递

const content = posts.map((post) =>

<Post

key={post.id}

id={post.id}

title={post.title} />

);

# 表单

## 受控组件

在React中，可变的状态通常保存在组件的状态属性中，并且只能用 setState() 方法进行更新。

通过绑定状态改变事件函数，使用e.target.value传递值；调用setState()重置state数据，刷新显示。

## textarea 标签

在HTML当中，<textarea> 元素通过子节点来定义它的文本内容。

在React中，<textarea>会用value属性来代替。处理方法同上<input type="text"/>。

## select 标签

HTML在子标签使用selected表示选中。

在React中，并不使用之前的selected属性，而在根select标签上用value属性来表示选中项。

## 多个输入的解决方法

当你有处理多个受控的input元素时，你可以通过给每个元素添加一个name属性，来让处理函数根据 event.target.name的值来选择做什么。

使用ES6当中的计算属性名语法来更新与给定输入名称相对应的状态键：

this.setState({

[name]: value //计算name的值，转换为属性名

});

# 状态提升

在React中，状态分享是通过将state数据提升至离需要这些数据的组件最近的父组件来完成的。这就是所谓的状态提升。

调用父组件的setState()方法更新state，并将更新后的state赋给子组件，刷新显示。

# 组合 vs 继承

React 具有强大的组合模型，我们建议使用组合而不是继承来复用组件之间的代码。

如果要在组件之间复用 UI 无关的功能，我们建议将其提取到单独的 JavaScript 模块中。这样可以在不对组件进行扩展的前提下导入并使用该函数、对象或类。

# React理念

props 是一种从父级向子级传递数据的方法

在创建静态版本的时候不要使用 state。State 只在交互的时候使用，即随时间变化的数据。

确定state：通过props从父级传来？随时间推移(用户点击、输入等操作)不变？能否根据state/props算出？

## 定义 UI 状态的最小(但完整)表示

它是通过 props 从父级传来的吗？如果是，他可能不是 state。

它随着时间推移不变吗？如果是，它可能不是 state。

你能够根据组件中任何其他的 state 或 props 把它计算出来吗？如果是，它不是 state。

# 深入 JSX

JSX为 React.createElement(component, props, ...children)方法提供的语法糖。

## 指定 React 元素类型

JSX 的标签名决定了 React 元素的类型。

大写开头的 JSX 标签表示自定义或导入的 React 组件。这些标签将会被编译为同名变量并被引用，所以如果你使用了 <Foo /> 表达式，则必须在作用域中先声明 Foo 变量。

大写开头的 JSX 标签表示内置组件，如 <div>、<span>。

JSX 编译后会调用 React.createElement 方法，所以JSX 代码中必须首先声明 React 变量。

可使用 JSX 中的点表示法来引用 React 组件，如：MyComponents.DatePicker。

在运行时选择类型，不能使用表达式来作为 React 元素的标签。如果需通过表达式来确定 React 元素的类型，需要先将类型赋值给大写开头的变量，再以大写变量创建标签。

## 指定属性值

#### JavaScript 表达式

使用{} 包裹JavaScript 表达式作为一个属性值；

<MyComponent foo={1 + 2 + 3 + 4} /> //foo=10

#### **字符串常量**

可将字符串常量作为属性值传递。该属性值会被解析为HTML非转义字符串。

<MyComponent message="hello world" />

<MyComponent message={'hello world'} />

#### **默认为 True**

如果你没有给属性传值，它默认为 true。

#### **扩展属性**

可使用 ... 作为扩展操作符，来解构传递整个属性对象

const props = {firstName: 'Ben', lastName: 'Hector'};

<Greeting {...props} />;

## 子代

在包含开始和结束标签的 JSX 表达式中，标记之间的内容作为特殊的参数传递：props.children。

### 传递子代

#### **字符串常量**

在开始和结束标签之间放入一个字符串，则 props.children 就是那个字符串。

JSX 会移除空行和开始与结尾处的空格；字符串常量内部的换行会被压缩成一个空格。

#### **JSX**

可通过子代嵌入更多的 JSX 元素；React 组件可通过数组的形式返回多个元素。

#### **JavaScript** 表达式

可将任何 {} 包裹的 JavaScript 表达式作为子代传递。

#### **函数**

使用自定义组件，调用 props.children 来获得传递的子代。

#### **布尔**值、Null 和 Undefined 被忽略

false、null、undefined 和 true 都是有效的子代，但它们不会直接被渲染。

根据条件来确定是否渲染React元素：

{showHeader && <Header />}

如需显示false、true、null、undefined，必须转换成字符串。

String(myVariable)

# 使用 PropTypes 进行类型检查

注意: React.PropTypes 自 React v15.5 起已弃用。请使用 prop-types 库代替。

propTypes 只在开发模式下进行检查。

Greeting.propTypes = {

name: PropTypes.string //根据不同需求使用不同的验证器

};

## 限制单个子代

Greeting.propTypes = {

children: PropTypes.element.isRequired

};

## 属性默认值

通过配置 defaultProps 为 props定义默认值。

Greeting.defaultProps = {

name: 'Stranger'

};

# 静态类型检查

静态类型检查器可以在运行代码之前识别某些类型的问题。对于更大的代码库我们建议使用Flow 或者TypeScript等静态类型检查器。

## Flow

一个针对 JavaScript 代码的静态类型检查器，通过使用特殊的语法为类型注释扩展了 JavaScript语言。

### 在项目中添加 Flow

Yarn方式：

yarn add --dev flow-bin

yarn run flow init

npm方式：

npm install --save-dev flow-bin //安装最新版的 Flow

npm run flow init //创建Flow 配置文件

将 flow 添加到 package.json中的 "scripts" 部分。

{

// ...

"scripts": {

"flow": "flow",

// ...

},

// ...

}

### 从编译过的代码中剥离 Flow 语法

Create React App，Flow被默认剥离。

#### Babel剥离Flow

安装一个特殊的 preset

yarn add --dev babel-preset-flow

npm install --save-dev babel-preset-flow

.babelrc 文件，将 flow preset 加入Babel 配置。

{

"presets": [

"flow",

"react"

]

}

### 运行 Flow

yarn flow

npm run flow

### 添加 Flow 类型注释

默认情况下， Flow 仅检查包含"// @flow"的文件。

[Flow 文档：类型注释](https://flow.org/en/docs/types/)

[Flow 文档：编辑器](https://flow.org/en/docs/editors/)

[Flow 文档： React](https://flow.org/en/docs/react/)

[Linting in Flow](https://medium.com/flow-type/linting-in-flow-7709d7a7e969)

## TypeScript

### 在一个项目中添加 TypeScript

yarn add --dev typescript

npm install --save-dev typescript

配置tsc到 package.json 中的 “scripts” 部分。

"scripts": {

"build": "tsc",

// ...

}

### 配置 TypeScript 编译器

生成tsconfig.json文件，并配置选项。

tsc --init

文件扩展名

.ts，默认的文件扩展名，

.tsx，为包含 JSX 代码使用的特殊扩展名。

### 运行 TypeScript

yarn build

npm run build

TypeScript 的更多信息：

[TypeScript 文档：基本类型](https://www.typescriptlang.org/docs/handbook/basic-types.html)

[TypeScript 文档：从 Javascript 迁徙](http://www.typescriptlang.org/docs/handbook/migrating-from-javascript.html)

[TypeScript 文档： React 和 Webpack](http://www.typescriptlang.org/docs/handbook/react-&-webpack.html)

# 菜鸟教程总结

1，HTML中直接使用react，导入react.min.js、react-dom.min.js、babel.min.js。如使用JSX，则<script> 标签的 type 属性需要设置为 text/babel。

2，通过create-react-app使用react：

npm install -g create-react-app //安装

create-react-app my-app //创建项目

3，JSX中，标签元素添加自定义属性需要使用 data- 前缀。

<p data-myattribute = "somevalue">HappyEveryday!</p>

4，JSX中，{}中为js代码；标签中为html。JSX不能使用if-else语句，使用三目表达式代替。

5，可使用camelCase语法来设置内联样式. React会自动添加 px 。

var myStyle = {

fontSize: 100,

color: '#FF0000'

};

ReactDOM.render(

<h1 style = {myStyle}>菜鸟教程</h1>,

document.getElementById('example')

);

6，标签内注释格式：{/\*注释...\*/}；

标签外注释格式：/\*注释...\*/；

7，在添加属性时， class 属性需要写成 className ，for 属性需要写成 htmlFor。

8，getInitialState 方法用于定义state初始状态；getDefaultProps();设置默认props。

9，验证Props：React.PropTypes；如 title:React.PropTypes.string.isRequired，属性 title 是必须的且是字符串，非字符串类型会自动转换为字符串。

10，React 组件 API

setState，设置状态

replaceState，替换状态

setProps，设置属性

replaceProps，替换属性

forceUpdate，强制更新，调用render()重新渲染

findDOMNode，获取DOM节点，获取render()输出的DOM

isMounted，判断组件挂载状态

11，React生命周期的方法：

componentWillMount 在渲染前调用,在客户端也在服务端。

componentDidMount : 在第一次渲染后调用，只在客户端。之后组件已经生成了对应的DOM结构，可以通过this.getDOMNode()来进行访问。 如果你想和其他JavaScript框架一起使用，可以在这个方法中调用setTimeout, setInterval或者发送AJAX请求等操作(防止异部操作阻塞UI)。

componentWillReceiveProps 在组件接收到一个新的 prop (更新后)时被调用。这个方法在初始化render时不会被调用。

shouldComponentUpdate 返回一个布尔值。在组件接收到新的props或者state时被调用。在初始化时或者使用forceUpdate时不被调用。可以在你确认不需要更新组件时使用。

componentWillUpdate在组件接收到新的props或者state但还没有render时被调用。在初始化时不会被调用。

componentDidUpdate 在组件完成更新后立即调用。在初始化时不会被调用。

componentWillUnmount在组件从 DOM 中移除的时候立刻被调用。

12，React Ajax？

导入axios；import axios from 'axios';

使用axios完成异步请求；axios.get(url).then(result=>{...});

或者使用callback完成。

1. React与SpringBoot配合？

# 陕西项目整理

## require.ensure()

按需加载组件，分模块打包。在路由中使用。

require.ensure(dependencies: String[], callback: function(require), chunkName: String)

require.ensure([], require => { //无依赖

callback(null, require('./src/home/Home'));//错误时返回null？回调函数。

}, 'home');//模块名

配置模块名生效，需配置module.exports的publicPath、chunkFilename。

module.exports={

entry:'./js/entry.js',

output:{

path:path.resolve(\_\_dirname,"./dist"),

filename:'js/a.bundle.js',

publicPath:"./",

chunkFilename:'js/[name].js'

}

## routes-indexRoute

默认路由，默认导航到Home组件。

const routes = {

childRoutes: [{

path: '/',

component: require('./src/Root'),

indexRoute: {

component: require('./src/home/Home')

}

}]

}

## 配置路由

通过管理 URL，实现组件的切换和状态的变化。

### 安装

npm install -S react-router

### 基本用法

路由器Router是React的一个组件。Router组件本身只是一个容器，真正的路由要通过Route组件定义。

import { Router, Route, hashHistory } from 'react-router';

render((

<Router history={hashHistory}>

<Route path="/" component={App}/>

</Router>

), document.getElementById('app'));

### 嵌套路由

用户访问/repos时，会先加载App组件，然后在它的内部再加载Repos组件。

<Router history={hashHistory}>

<Route path="/" component={App}>

<Route path="/repos" component={Repos}/>

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

</Route>

</Router>

嵌套的父组件需通过this.props.children加载子组件，改写App组件：

export default React.createClass({

render() {

return <div>

{this.props.children}

</div>

}

})

子路由也可以不写在Router组件里面，通过routes属性传入Router组件。

let routes = <Route path="/" component={App}>

<Route path="/repos" component={Repos}/>

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

</Route>;

<Router routes={routes} history={browserHistory}/>

### path 属性

Route组件的path属性指定路由的匹配规则。

属性省略后，不管路径是否匹配，总是会加载指定组件。

### 通配符

path属性可以使用通配符。通配符的规则如下。

（1）:paramName

:paramName匹配URL的一个部分，直到遇到下一个/、?、#为止。

这个路径参数可以通过this.props.params.paramName取出。

（2）()

()表示URL的这个部分是可选的。

（3）\*

\*匹配任意字符，直到模式里面的下一个字符为止。匹配方式是非贪婪模式。

（4） \*\*

\*\* 匹配任意字符，直到下一个/、?、#为止。匹配方式是贪婪模式。

<Route path="/hello/:name"> // 匹配 /hello/michael

<Route path="/hello(/:name)"> // 匹配 /hello/michael

<Route path="/files/\*.\*"> // 匹配 /files/hello.jpg

<Route path="/files/\*"> // 匹配 /files/

<Route path="/\*\*/\*.jpg"> // 匹配 /files/path/to/file.jpg

path属性可以使用相对路径（不以/开头）。

路由匹配规则是从上到下执行，一旦发现匹配，不再其余的规则。

带参数的路径一般要写在路由规则的底部。

URL的查询字符串/foo?bar=baz，可以用this.props.location.query.bar获取。

### IndexRoute 组件

显式指定根路由/的子组件，即指定默认情况下加载的子组件；没有路径参数path。

<Router>

<Route path="/" component={App}>

<IndexRoute component={Home}/>

<Route path="accounts" component={Accounts}/>

<Route path="statements" component={Statements}/>

</Route>

</Router>

### Redirect 组件

用于路由的跳转。

<Route path="inbox" component={Inbox}>

{/\* 从 /inbox/messages/:id 跳转到 /messages/:id \*/}

＜Redirect from="messages/:id" to="/messages/:id" />

</Route>

### IndexRedirect 组件

用于访问根路由的时候，将用户重定向到某个子组件。

<Route path="/" component={App}>

＜IndexRedirect to="/welcome" />

<Route path="welcome" component={Welcome} />

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

</Route>

### Link

用于取代<a>元素，生成一个链接，允许用户点击后跳转到另一个路由。

<Link to="/repos">Repos</Link>

通过activeStyle、activeClassName设置样式。

### IndexLink

链接到根路由/，需使用IndexLink组件；IndexLink组件会使用路径的精确匹配，Link默认使用前缀匹配。

<IndexLink to="/" activeClassName="active">Home</IndexLink>

使用Link组件的onlyActiveOnIndex属性，也能达到同样效果。

<Link to="/" activeClassName="active" onlyActiveOnIndex={true}> Home</Link>

### histroy 属性

Router组件的history属性，用来监听浏览器地址栏的变化，并将URL解析成一个地址对象，供 React Router 匹配。

history属性，一共可以设置三种值。

browserHistory，路由通过URL的hash部分（#）切换，URL形式：example.com/#/some/path。

hashHistory，显示正常的路径example.com/some/path

createMemoryHistory，用于服务器渲染。

### 表单处理

表单、按钮跳转

1，browserHistory.push(path)

2，this.context.router.push(path)

### 路由的钩子

每个路由都有onEnter和onLeave钩子，用户进入或离开该路由时触发。

onEnter父到子；onLeave子到父。

通过OnEnter设置跳转地址。

<Route path="inbox" component={Inbox}>

<Route

path="messages/:id"

onEnter={

({params}, replace) => replace(`/messages/${params.id}`)

}

/>

</Route>

### 按需加载

Route 可以定义 getChildRoutes，getIndexRoute 和 getComponents 这几个函数，他们都是异步执行的，并且只有在需要的时候才会调用。这种方式需要配合webpack中有实现代码拆分功能的工具来用，其实就是把路由拆分成小代码块，然后按需加载。

require.ensure(dependencies: String[], callback: function(require), chunkName: String)

getChildRoutes(location, callback) {

require.ensure([], function (require) {

callback(null, [

require('./routes/Announcements'),

require('./routes/Assignments'),

require('./routes/Grades'),

])

})

},

getIndexRoute(location, callback) {

require.ensure([], function (require) {

callback(null, require('./components/Index'))

})

},

getComponents(location, callback) {

require.ensure([], function (require) {

callback(null, require('./components/Course'))

})

}

### 配置路由的三种方式

#### 标签的方式

React.render((

<Router>

<Route path="/" component={App}>

{/\* 当 url 为/时渲染 Dashboard \*/}

<IndexRoute component={Dashboard} />

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

<Route path="inbox" component={Inbox}>

<Route path="messages/:id" component={Message} />

</Route>

</Route>

</Router>

), document.body)

#### 对象配置方式

const routeConfig = [

{ path: '/',

component: App,

indexRoute: { component: Dashboard },

childRoutes: [

{ path: 'about', component: About },

{ path: 'inbox',

component: Inbox,

childRoutes: [

{ path: '/messages/:id', component: Message },

{ path: 'messages/:id',

onEnter: function (nextState, replaceState) {

//do something

}

}

]

}

]

}

]

React.render(<Router routes={routeConfig} />, document.body)

#### 按需加载的路由配置

const CourseRoute = {

path: 'course/:courseId',

getChildRoutes(location, callback) {

require.ensure([], function (require) {

callback(null, [

require('./routes/Announcements'),

require('./routes/Assignments'),

require('./routes/Grades'),

])

})

},

getIndexRoute(location, callback) {

require.ensure([], function (require) {

callback(null, require('./components/Index'))

})

},

getComponents(location, callback) {

require.ensure([], function (require) {

callback(null, require('./components/Course'))

})

}

}