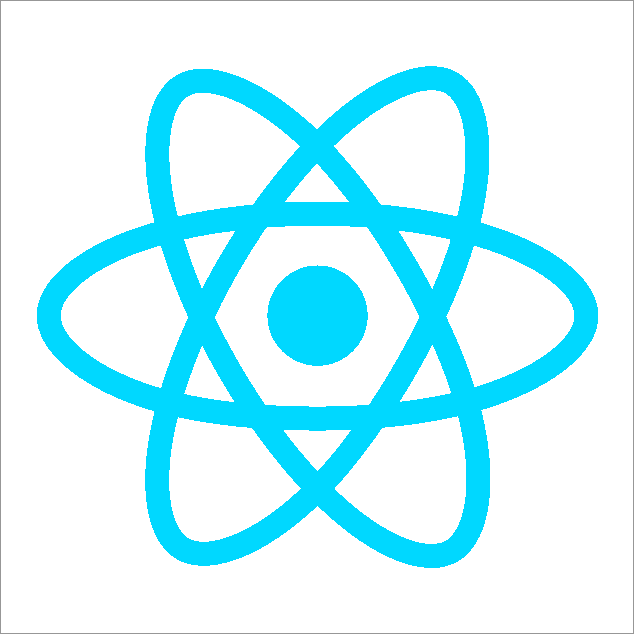
# React框架

1. 认识React





### React简介

React 起源于 Facebook 的内部项目，因为该公司对市场上所有 JavaScript MVC 框架，都不满意，就决定自己写一套，用来架设 Instagram（照片交友） 的网站。做出来以后，发现这套东西很好用，**就在2013年5月开源了**。

Angular1 2009 年 谷歌 MVC 不支持 组件化开发

由于 React 的**设计思想极其独特**，属于革命性创新，性能出众，代码逻辑却非常简单。所以，越来越多的人开始关注和使用，认为它可能是将来 Web 开发的主流工具。

清楚两个概念：

library（库）：小而巧的库，只提供了特定的API；优点就是 船小好掉头，可以很方便的从一个库切换到另外的库；但是代码几乎不会改变；

Framework（框架）：大而全的是框架；框架提供了一整套的解决方案；所以，如果在项目中间，想切换到另外的框架，是比较困难的；

### React与vue的对比

React和Vue的区别可以从组件化 , 开发团队 , 社区 , 移动App开发几方面来看.

#### 组件化方面

**Vue是如何实现组件化的：** 通过 .vue 文件，来创建对应的组件；

template 结构

script 行为

style 样式

**React如何实现组件化**： 大家注意，React中有组件化的概念，但是，并没有像vue这样的组件模板文件；React中，一切都是以JS来表现的；因此要学习React，JS要合格；

#### 开发团队方面

React是由FaceBook前端官方团队进行维护和更新的；因此，React的维护开发团队，技术实力比较雄厚；

Vue：第一版，主要是有作者 尤雨溪 专门进行维护的，当 Vue更新到 2.x 版本后，也有了一个以 尤雨溪 为主导的开源小团队，进行相关的开发和维护；

#### 社区方面

在社区方面，React由于诞生的较早，所以社区比较强大，一些常见的问题、坑、最优解决方案，文档、博客在社区中都是可以很方便就能找到的；

Vue是近两年才火起来的，所以，它的社区相对于React来说，要小一些，可能有的一些坑，没人踩过；

#### 移动APP开发体验方面

Vue，结合 Weex/element-ui/mint-ui ，提供了 迁移到 移动端App开发的体验

React，结合 ReactNative，也提供了无缝迁移到 移动App的开发体验（RN用的最多，也是最火最流行的）；

### React中几个核心概念

React中最核心的概念就是 虚拟DOM 和 Diff算法.

#### 虚拟DOM（Virtual Document Object Model）

**DOM的本质是什么**：浏览器中的概念，用JS对象来表示 页面上的元素，并提供了操作 DOM 对象的API；

比如, var div = document.getElementById(‘app’); 这里的 div 就是dom( 它是个js对象 )

**什么是React中的虚拟DOM**：是框架中的概念，是程序员 用JS对象来模拟 页面上的 DOM 和 DOM嵌套；

**为什么要实现虚拟DOM（虚拟DOM的目的）：**为了实现页面中， DOM 元素的高效更新;

**DOM和虚拟DOM的区别**：

**DOM：**浏览器中，提供的概念；用JS对象，表示页面上的元素，并提供了操作元素的API；

**虚拟DOM：**是框架中的概念；而是开发框架的程序员，手动用JS对象来模拟DOM元素和嵌套关系；

本质： 用JS对象，来模拟DOM元素和嵌套关系；

目的：就是为了实现页面元素的高效更新；

#### Diff算法

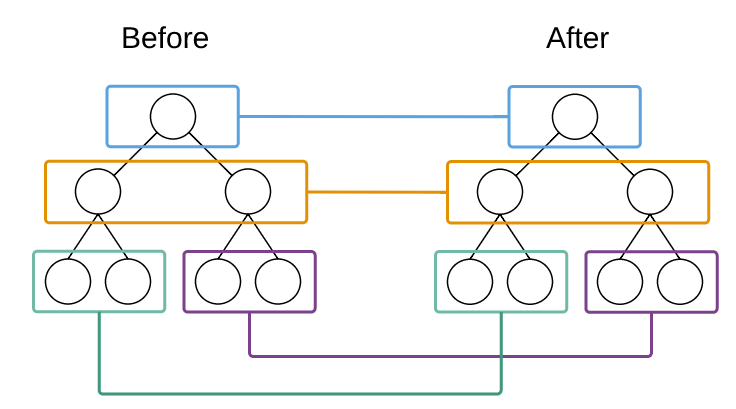
**tree diff:** 新旧两棵DOM树，逐层对比的过程，就是 Tree Diff； 当整颗DOM逐层对比完毕，则所有需要被按需更新的元素，必然能够找到；

**component diff：**在进行Tree Diff的时候，每一层中，组件级别的对比，叫做 Component Diff；

如果对比前后，组件的类型相同，则**暂时**认为此组件不需要被更新；

如果对比前后，组件类型不同，则需要移除旧组件，创建新组件，并追加到页面上；

**element diff:** 在进行组件对比的时候，如果两个组件类型相同，则需要进行 元素级别的对比，这叫做 Element Diff；



### React相关资源

React 中文教程 <https://react.docschina.org/docs/getting-started.html>

React 入门实例 <http://www.ruanyifeng.com/blog/2015/03/react.html>

React 菜鸟教程 <https://www.runoob.com/react/react-tutorial.html>

1. 准备react环境

### 在线编译环境

在线编译环境,也就是非模块化环境.

用script标签引入react , react-dom包.

<script crossorigin src="../react..js"></script>  
<script crossorigin src="../react-dom.js"></script>

react.js 是 React 的核心库;

react-dom.js 是提供与 DOM 相关的功能 , 渲染dom到页面;

const *ele* = React.createElement( 'div',null,'hello,world!' );  
ReactDOM.render(*ele* , document.getElementById( 'app' ));

这样,, 就可以了将一个名为div的 dom元素渲染到app容器中显示.

### 预编译环境

预编译 即 在代码在浏览器运行前先编译; 预编译环境的好处是可以提升用户的网页浏览体验;

babel以模块化的方式和webpack等工具配合实现预编译,, 最终在浏览器端运行的是编译后的代码;

#### 创建基本的webpack 4.x 环境项目

1. 运行npm init -y 快速初始化项目
2. 在项目根目录创建src源代码目录和dist产品目录

在 src 目录下创建 index.html

1. 使用 cnpm 安装 webpack ，运行cnpm i webpack webpack-cli -D

注意：webpack 4.x 提供了 约定大于配置的概念；目的是为了尽量减少 配置文件的体积；

默认约定了：

打包的入口是src -> index.js

打包的输出文件是dist -> main.js

4.x 中 新增了 mode 选项(为必选项)，可选的值为：development 和 production;

#### 配置,保存立即打包

安装 webpack-dev-server

在 package.json 中添加脚本 “dev” : ”webpack-dev-server –open –port 3000 –host 127.0.0.1 --hot”

这样, 执行 npm run dev 即可;

--open 自动打开浏览器

--port 指定端口号

--host 指定主机

### 脚手架环境

搭建脚手架环境步骤:

1. 全局安装create-react-app
   1. 安装命令: npm install create-react-app –g
   2. 安装成功后create-react-app和npx命令就都可以使用了;
2. 使用create-react-app或npx命令创建项目
   1. create-react-app 项目名

或者

* 1. npx create-react-app 项目名

1. 进入项目目录
   1. cd 项目名
2. 启动项目
   1. npm start

注意: 如果执行create-react-app过程中报错,可以尝试以下解决方案:

npm config set registry https:*//registry.npm.taobao.org*

*//配置后通过以下方法验证是否成功*

npm config get registry

**cdn链接**

react <https://unpkg.com/react@16/umd/react.development.js>

react-dom <https://unpkg.com/react-dom@16/umd/react-dom.development.js>

react-router <https://unpkg.com/react-router/umd/react-router.min.js>

react-router-dom <https://unpkg.com/react-router/umd/react-router-dom.min.js>

babel <https://unpkg.com/babel-standalone@6/babel.min.js>

### React Developer Tools

**react-devtools**是一款由 **facebook**开发的有用的 **[Chrome](http://www.cnplugins.com/)**[浏览器扩展](http://www.cnplugins.com/)。通过它我们可以查看应用程序的 **React**组件分层结构,查看组件的props,state等数据，而不是更加神秘的浏览器 **DOM**表示;

安装方式有两种:

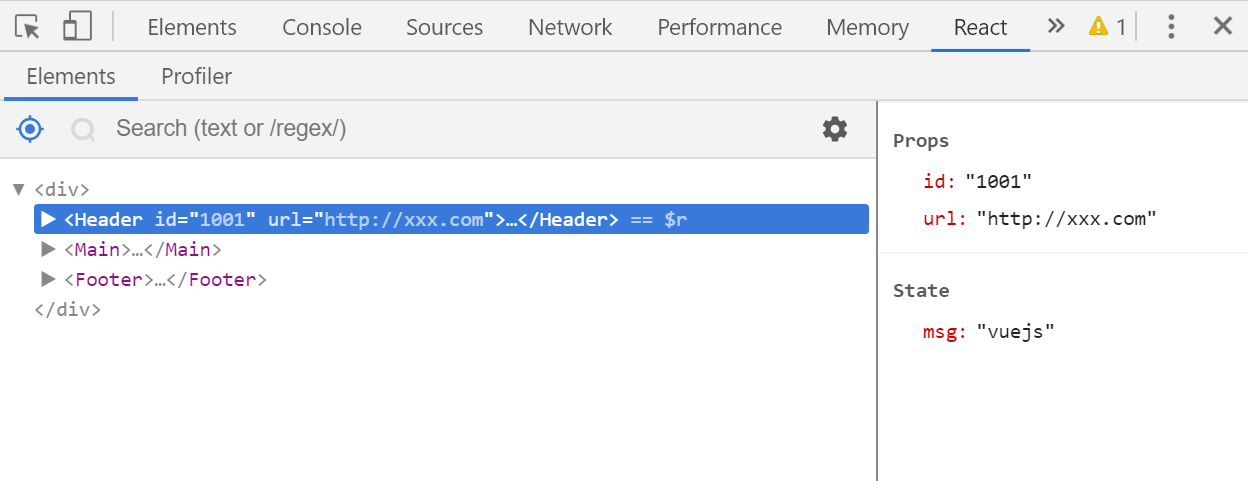
方式一: **chrome商店上直接下载**

**（前提是该电脑已经翻墙）**

方式二: **官网下载源代码，在chrome中加载已解压的扩展程序**

我们介绍这种通用的安装方式.

操作步骤:

1. 在github上下载react-devtools的压缩包;
2. 解压后，进入react-devtools-master工程 执行cnpm install, 下载依赖;
3. 执行npm run build:extension:chrome，编译源程序;
4. 编译后,会在你的项目目录中生成一个新的文件夹，react-devtools -> shells -> chrome -> build -> unpacked文件夹;
5. 打开chrome浏览器的扩展程序标签页 <chrome://extensions/>
6. 打开开发者模式 并且点击加载已解压的扩展程序 并且选择刚才的文件夹(react-devtools -> shells -> chrome -> build -> unpacked)
7. 然后打开一个react页面,F12打 开调试工具,就可以看到在最后多了一个React,随便选中一个组件,右侧可以看到该组件的props,state等数据;

源码地址: <https://github.com/facebook/react-devtools>

1. 使用jsx语法

jsx是一种预编译 JavaScript 语言，允许让你的 JavaScript 和 HTML 混着写;

JSX 全称 JavaScript XML ，是一种扩展的 JavaScript 语言，它允许 HTML 语言直接写在 JavaScript 语言中，不加任何引号，这就是 JSX 语法。

#### 基本语法规则

1. 组件必须有且只能有一个根节点
2. 多个并列标签用一个小括号包裹，防止 JavaScript 自动分号不往后执行的问题。
3. 遇到 HTML 标签 （以 < 开头） 就用 HTML 规则解析
4. 遇到代码块（以 { 开头），就用 JavaScript 规则解析;
5. 单标签不能省略结束标签 , 必须结束 /> ;
6. JSX 允许直接在模板中插入 JavaScript 变量 {变量} ;
7. 如果这个变量是一个数组，则会展开这个数组的所有成员添加到模板中;

语法示例:

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

#### 识别jsx语法

默认 , 浏览器无法识别jsx语法,让浏览器能识别jsx语法必须借助转换器babel;

<script crossorigin src="../react..js"></script>  
<script crossorigin src="../react-dom.js"></script>  
<script src="../babel.min.js"></script>

react.js 是 React 的核心库;

react-dom.js 是提供与 DOM 相关的功能 , 渲染dom到页面;

babel.min.js的作用是将ES2015的代码编译成ES5代码，也可以将JSX编译成javascript函数;

**在webstorm中,设置让webstorm识别jsx语法:**

找到 File -> Settings -> Languages&Frameworks -> JavaScript

将JavaScript language version 设置为 JSX Harmony

#### jsx原理

**const *ele*** = <**div**>hello,world!</**div**>**;**

类似这样的jsx代码,,最终会被babel编译成js函数;

如:

const *ele* = React.createElement('div',null,'hello,world!');

所以,, 用jsx语法,,无非是让程序员可以以dom模板的方式编写UI,,更直观;

#### jsx中的注释

**写法一**（不推荐）：

{  
  // 注释  
  // ...  
}

**写法二**（推荐，把多行写到单行中）：

{ /\* 单行注释 \*/ }

**写法三**（多行）：

{  
  /\*  
  \* 多行注释  
  \*/  
}

注释在babel转换过程中会被忽略掉,, 不会出现在最终的js代码中.

#### jsx中的节点属性

jsx也是JavaScript, 为了不和js中的关键字冲突,在jsx中书写节点属性需注意:

**属性名称:**1. class 需要写成className  
2. for 需要写成htmlFor  
3. tabindex 需要写成 tabIndex ( 保持驼峰命名法 )  
4. value 需要写成 defaultValue

**5.** checked 需要写成 defaultChecked

<div title="hello world" className="hello">hello,world!  
 <div> *<*labelhtmlFor="user" tabIndex="1">user</label>  
 <input type="text" id="user" defaultValue={*user*.name}/>  
 </div>  
</div>

最终编译结果还是 class , for , tabindex , value , checked

**属性值:**  
如果属性值需要绑定动态值,则用{}实现 ( 无需添加引号"" )

<input type="text" id="user" defaultValue={*user*.name}/>

style的值需要这样: {{width:200,fontSize:20}}

<label style={{width:20,fontSize:20}} >user</label>

动态设置style值, 多个style间用,分割

<div style={{display: this.state.isShow ? "block" : "none", color:"pink"}}>

demo

</div>

动态设置className

单个className

<div className={this.state.isActive?"active":"default"}>demo</div>

多个className

//es6模板字符串

className={`title ${index === **this**.state.active ? 'active' : ''}`}

或者

//数组join

className={["title", index === **this**.state.active?"active":null].join(' ')}

#### jsx列表渲染

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。

数组成员可以是简单数据 number, string , 也可以是 dom 元素

const *list* = ['sunyang','lining','chenhe'];

const *list* = [  
 <li key="1">sunyang</li>,  
 <li key="2">lining</li>,  
 <li key="3">chenhe</li>  
];

当数组成员是复合数据时,,需要经过forEach 或 map方法处理后再渲染输出;

const *list* = [  
 { name:'sunyang',age:10 },  
 { name:'lining',age:20 },  
 { name:'chenhe',age:30 }  
];

<ul>  
 {  
 *list*.map(function (item) {  
 return (<li key={item.age}>{item.name}</li>)  
 })  
 }  
</ul>

注意: 关于key,最好不要用map循环的index作为key值;

#### jsx事件处理

**示例:**

<div>  
 { /\* <button onClick={ handleClick }>clickme</button> \*/}  
 <button onClick={ ()=> { handleClick() } }>clickme</button>  
</div>

**需要注意:**

1.事件名必须符合驼峰命名法;  
2.事件值必须绑定一个函数 即 {handleClick}, 不能是字符串形式 “handleClick”  
3.可以是匿名函数;

1. 组件

React支持函数式组件 , class组件;

### 函数式组件

示例:

**定义组件**

function **Header**(){  
 return (  
 <div className="header">  
 <h1>header</h1>  
 </div>  
 )  
}

**使用组件**

const *ele* = (<div>  
 <Header></Header>  
 </div>);  
  
ReactDOM.render(*ele*,document.getElementById('app'));

**函数式组件是“无状态组件”**

注意: 函数式组件的组件名称首字母必须大写;无法实例化,没有state,没有生命周期方法;

### class组件

示例：

class Header extends React.Component{  
 constructor(){  
 super();

}

render(){  
 return (  
 <div className="header">  
 <h1>header</h1>  
 </div>  
 )  
 }  
}

组件中如果定义了constructor构造方法，必须手动用super（）调用父类构造方法；

**注意：**

组件名称首字母必须大写；  
 组件必须继承 React.Component 组件类；  
 组件中 用render方法返回 组件模板；

组件的所有子节点,在组件内通过 this.props.children 获取展示;

### 组件实例属性

组件最重要的三个属性: props , state , refs

props 保存组件外部传入的属性属性,是只读的;

state 保存组件的私有数据,可以用setState修改;

refs 保存组件内节点的引用;

### class组件状态

组件状态数据需要在组件构造方法中定义;

示例:

class Header extends React.Component{  
 constructor(){  
 super();  
 this.state = {

a:100,  
 msg : 'hello class'  
 }  
 }  
 handleClick(){  
 //如果要修改state中的数据,并期望视图更新,则需要用 this.setState方法  
 this.setState({  
 msg:'data changed'  
 });

}  
 render(){  
 return (  
 <div className="header">  
 <h1>header,{this.state.msg}</h1>  
 <button onClick={(e)=>this.handleClick(e)} >change</button>  
 </div>  
 )  
 }  
}

组件构造方法中定义组件状态;

组件模板中可以用 this.state.xxx 访问状态;

修改组件状态,必须用 this.setState({ xxx:yyy })

**注意**: this.setState() 是异步操作,可以在它的回调函数中获取更新后的值;

this.setState( { msg:'data changed'} , ()=>{ console.log( this.state.msg ) } );

### class组件中this绑定问题

这里的this绑定问题主要指 事件处理方法中this的指向问题。

事件处理方法中this 默认指向 undefined ，无法调用setState修改state

#### 方式一 ( 提前bind )：

在构造方法中,提前bind(this)

constructor(props){

super(props);

this.handleClick = this.handleClick.bind(this);

}

然后

<button onClick={ this.handleClick }>点击改变 message</button>

默认接收一个参数 event 事件源对象

不支持额外的参数传递

#### 方式二（bind）：

<button onClick={this.handleClick.bind(this)}>change</button>

this 指向组件实例

默认接收一个参数 event

手动传递的参数会放到函数最前面，event 会作为函数的最后一个参数；

#### 方式三（arrow function）：

自动 bind this

手动传递参数

参数顺序自己指定，event 也需要自己手动传递

<button onClick={ (e)=>this.handleClick(e) }>change</button>

### 组件传值

组件传值包括父组件向子组件传值 , 子组件向父组件传值;

#### 父组件向子组件传值

实现方式: *父组件在子组件身上通过组件属性传递值，，子组件内部用 props 接收值;*

**父组件传递值:**

<Header id="1001" url={[100]}></Header>

**子组件接收值( 函数式组件 ):**

函数式组件用参数 props 接收值

function *Header* ( props ){  
 return (  
 <div className="header">  
 {/\* 组件模板中使用 组件的属性 \*/}  
 <h1>{props.id},{props.url}</h1>  
 </div>  
 )  
}

**接收值( class组件 ):**

**class**组件内部用 this.props 接收值

class Header extends React.Component{  
 render(){  
 return (  
 <div className="header">  
 {/\* 组件模板中使用 组件的属性 \*/}  
 <h1>{this.props.id},{this.props.url}</h1>

</div>  
 )  
 }  
}

this.props 对象的属性与组件的属性一一对应,但是有一个例外，就是 this.props.children 属性。它表示组件的所有子节点。  
  
this.props.children 的值有三种可能：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` 。所以，处理 `this.props.children` 的时候要小心。  
  
React 提供一个工具方法 React.Children 来处理 this.props.children 。我们可以用 React.Children.map 来遍历子节点，而不用担心 this.props.children 的数据类型是 `undefined` 还是 `object`;

#### 子组件向父组件传值

实现方式: 父组件在子组件身上通过组件属性传递值(函数)，，子组件内部用 props 接收值(函数),在特定时机触发函数调用,同时将数据以参数的形式传给父组件;

父组件指定组件属性(函数)

class Header extends React.Component{  
 constructor(){  
 super();  
 this.state = {  
 msg : '点击导航项传递数据到父组件'  
 }  
 }  
 handleClick(item){  
 this.setState({  
 msg:item  
 });  
 }  
 render(){  
 return (  
 <div className="header">  
 <div >点中的导航项为: {this.state.msg}</div>  
 <Nav sendData={ this.handleClick.bind( this ) }></Nav>  
 </div>  
 )  
 }  
}

子组件触发函数调用(传递值)

class Nav extends React.Component{  
 constructor(){  
 super();  
 this.state = {  
 list:["html",'css','javascript',"jquery","vuejs"]  
 }  
 }  
 handleClick(item){  
 //调用传递给子组件的click属性中的方法,同时将参数传递到父组件  
 this.props.sendData( item );  
 }  
 render(){  
 return (  
 <div className="nav">  
 {this.state.list.map(item=>{return <span key={item} onClick={this.handleClick.bind(this,item)}>{item}</span>})}  
 </div>  
 )  
 }  
}

**注意**: 在React组件化开发过程中,有个原则: state写在哪,操作数据的方法就定义在哪;

### Refs & dom

Refs 提供了一种方式，用于访问在 render 方法中创建的 DOM 节点或 React 元素。

不过, 如果可以通过声明式实现，则尽量避免使用 refs;

**创建Refs**

使用 React.createRef() 创建 refs，通过 ref 属性来获得 React 元素。当构造组件时，refs 通常被赋值给实例的一个属性，这样你可以在组件中任意一处使用它们;

class MyComponent extends React.Component {

constructor(props) {

super(props);

this.myRef = React.createRef();

}

render() {

return <div ref={ this.myRef } />;

}

}

**使用Refs**

当一个 ref 属性被传递给一个 render 函数中的元素时，可以使用 ref 中的 current 属性对节点的引用进行访问;

**this**.**input**.current

ref的值取决于节点的类型:

当 ref 属性被用于一个普通的 HTML 元素时，React.createRef() 将接收底层 DOM 元素作为它的 current 属性以创建 ref 。

当 ref 属性被用于一个自定义类组件时，ref 对象将接收该组件已挂载的实例作为它的 current。

**你不能在函数式组件上使用 ref 属性**，因为它们没有实例.

### 组件默认属性

**组件外指定组件默认属性**

组件外指定组件默认属性 com.defaultProps = {}

Header.*defaultProps* = {  
 id: **'1000'**,  
 url: **'file://'**}**;**

**组件内指定组件默认属性**

组件内指定组件默认属性,需要添加 static 修饰符

class Header extends React.Component{  
 //在组件内指定 默认属性,需要添加 static 修饰符  
 static *defaultProps* = {  
 id:'1000',  
 url:'file://'  
 };  
 render(){  
 return (  
 <div className="header">  
 {/\* 组件模板中使用 组件的属性 \*/}  
 <h1>{this.props.id},{this.props.url}</h1>  
 </div>  
 )  
 }  
  
}

### 组件属性类型验证

所谓组件属性类型验证,即可以规定组件属性值的类型.

React.PropTypes 自 React v15.5 起已弃用。使用 [prop-types](https://www.npmjs.com/package/prop-types)库代替;

import PropTypes from 'prop-types';

class Greeting extends React.Component {

render() {

return (

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

);

}

}

Greeting.propTypes = {

name: PropTypes.string

};

具体属性类型验证如下:

MyComponent.propTypes = {

// 你可以将属性声明为以下 JS 原生类型

optionalArray: PropTypes.array,

optionalBool: PropTypes.bool,

optionalFunc: PropTypes.func,

optionalNumber: PropTypes.number,

optionalObject: PropTypes.object,

optionalString: PropTypes.string,

optionalSymbol: PropTypes.symbol,

// 任何可被渲染的元素（包括数字、字符串、子元素或数组）。

optionalNode: PropTypes.node,

// 只有一个 React 元素

optionalElement: PropTypes.element,

// 你也可以声明属性为某个类的实例，这里使用 JS 的

// instanceof 操作符实现。

optionalMessage: PropTypes.instanceOf(Message),

// 你也可以限制你的属性值是某个特定值之一

optionalEnum: PropTypes.oneOf(['News', 'Photos']),

// 限制它为列举类型之一的对象

optionalUnion: PropTypes.oneOfType([

PropTypes.string,

PropTypes.number,

PropTypes.instanceOf(Message)

]),

// 一个指定元素类型的数组

arr: PropTypes.arrayOf(PropTypes.number),

// 一个指定类型的对象

optionalObjectOf: PropTypes.objectOf(PropTypes.number),

// 一个指定属性及其类型的对象

optionalObjectWithShape: PropTypes.shape({

color: PropTypes.string,

fontSize: PropTypes.number

}),

// 你也可以在任何 PropTypes 属性后面加上 `isRequired`

// 后缀，这样如果这个属性父组件没有提供时，会打印警告信息

requiredFunc: PropTypes.func.isRequired,

// 任意类型的数据

requiredAny: PropTypes.any.isRequired,

};

### 组件生命周期

**生命周期**：组件的实例，从 创建、到运行、直到销毁的整个过程，在这个过程中，在特定时刻会触发一些事件，这些事件就叫做组件的生命周期函数；

**组件的生命周期可分成三个状态(阶段)：**

Mounting：已插入真实 DOM( 创建/挂载阶段 )

Updating：正在被重新渲染( 运行/更新阶段 )

Unmounting：已移出真实 DOM( 销毁/卸载阶段 )

**挂载阶段:**

componentWillMount:

render：

componentDidMount:

**更新阶段**: 按需，根据 props 属性 或 state 状态的改变，有选择性的 执行 0 到多次

componentWillReceiveProps:

shouldComponentUpdate:

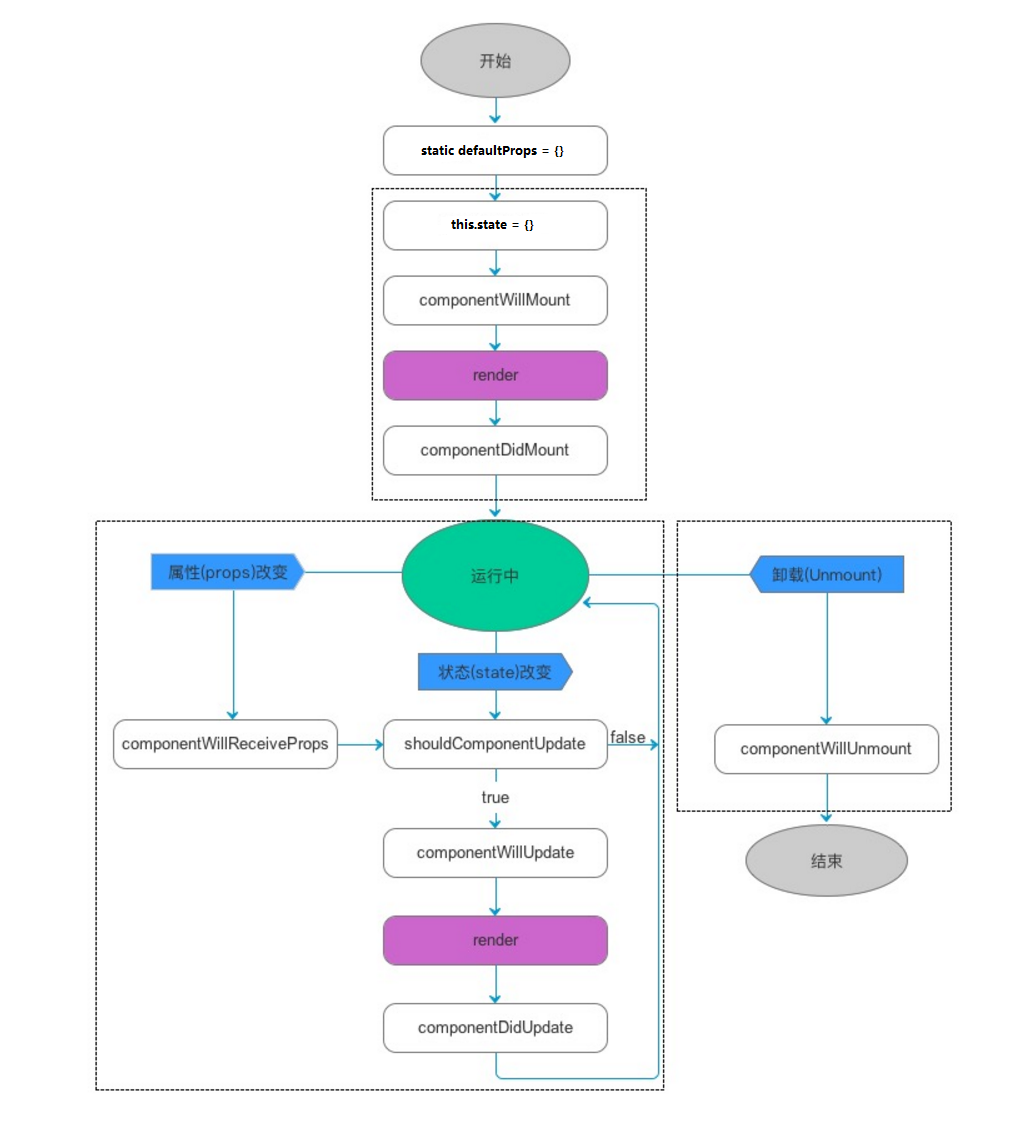
componentWillUpdate:

render:

componentDidUpdate:

**卸载阶段**:

componentWillUnmount:



**组件的生命周期方法有：**

**componentWillMount** : 在渲染前调用;

**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(防止异步操作阻塞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 中移除之前立刻被调用。

**手动卸载组件:**

ReactDOM.unmountComponentAtNode( container );

这个函数会触发componentWillUnmount的调用**;**

**实际应用**:

我们可以在 componentDIdMount 中创建定时器,在 componentWillUnmount 中销毁定时器, 保证定时器资源的及时释放;

可以在componentDIdMount 中发起ajax请求,在 componentWillUnmount 中终止请求, 保证程序运行效率;

1. 路由

在React中想要随着url地址的改变现实不同的内容,路由是最好的选择.

在React中，常用的有两个包可以实现这个需求，那就是react-router和react-router-dom;

本文档介绍的是react-router 4.

**react-router**: 实现了路由的核心功能  
**react-router-dom**: 基于react-router，加入了在浏览器运行环境下的一些功能，例如：Link组件，会渲染一个a标签， BrowserRouter和HashRouter组件，前者使用pushState和popState事件构建路由，后者使用window.location.hash和hashchange事件构建路由。

实际应用中用react-router-dom即可,下面看具体使用:

### 安装库

在线安装

<script crossorigin src="../react-router-dom.5.0.1.min.js"></script>

模块安装

cnpm i react-router-dom --save

react-router-dom依赖react-router，所以我们使用npm安装依赖的时候，只需要安装相应环境下的库即可，不用再显式安装react-router;

### 导入组件

非模块化导入:

var { HashRouter,Route,Link,Switch,Redirect,withRouter } = ReactRouterDOM;

模块化导入:

**import** { HashRouter,Route,Link,Switch } from **'react-router-dom';**

### 路由相关组件

路由相关组件有: BrowserRouter , HashRouter, Switch, Route, Redirect, Link等.

#### 路由器组件

react-router-dom包含多种路由器: BrowserRouter , HashRouter , MemoryRouter , NativeRouter , StaticRouter ;

通常，应用程序将使用其中一个路由器.

这里,重点介绍BrowserRouter , HashRouter:

**BrowserRouter** 使用HTML5历史记录API（pushState，replaceState和popstate事件）的<Router>来保持您的UI与URL同步。  
 basename : string 所有locations的基本URL  
  
**HashRouter**  使用URL的哈希部分（即window.location.hash）的<Router>来保持您的UI与URL同步  
 basename : string 所有locations的基本URL

**注意**: 路由器组件内只能有一个一级子节点.

导入时可为HashRouter 取别名

import {HashRouter as Router,Route,LinK} from 'react-router-dom'

#### 路由组件

**Switch** <Switch>的所有子项应为<Route>或<Redirect>元素 , 渲染与位置匹配的第一个子元素<Route> 或 <Redirect> , <Switch>是独特的，因为它仅仅渲染一个路由。相反，与位置匹配的每个<Route>都会包含  
  
**Route** 元素尝试将其路径与当前历史记录位置（通常是当前浏览器URL）进行匹配,渲染与位置匹配的组件; 如果路由没有路径，因此始终匹配;  
 path 路由路径

**注意**: 没有定义 path 的 <Route> 总是会被匹配;

exact 完全匹配

有子路由的Route不能使用 exact={true} 精确匹配，不然子路由无法匹配

component 需要渲染的组件

component 的值必须是func/class类型的组件,或者内建组件;

render: func 提供一个内联函数，路径匹配时调用,决定渲染哪个组件.

<Route path="/home" render={(props) => <div>Home</div>} />

内联函数默认接收props作为参数.可以拿到和当前路由相关的信息.

children: func

**Redirect** 渲染<Redirect>将导航到新位置。新位置将覆盖历史堆栈中的当前位置;  
 from:string 要重定向的路径名.  
 to: string 要重定向到的网址。  
 to: object 要重定向到的位置。

to: object 示例:

<Redirect to={{

pathname: '/login',

search: '?utm=your+face',

state: {

referrer: currentLocation

}

}} />

使用 <Route> 渲染一些内容有以下三种方式：

<Route component>

<Route render>

<Route children>

在不同的情况下使用不同的方式。在指定的 <Route> 中，你应该只使用其中的一种 , 大多数情况下你会使用 component

**Link**

link组件用来实现路由跳转链接 ( 必须写在Router内 )

**Prompt**

组件作用: 在用户准备离开该页面(组件)时, 弹出提示, 返回true或者false, 如果为true, 则离开页面, 如果为false, 则停留在该页面;

when属性, 就是渲染该组件的条件, 应该传入一个布尔值,值为true时,则渲染该组件( 这个属性不是必须的 );

message:string属性,该属性就是在用户离开页面时, 所提示的文字内容;

message:function属性,返回值为true/false, 返回false就继续停留在当前页面，返回true则跳转到新页面。该属性就是在用户离开页面时, 所提示的文字内容;

### 三个路由属性

三个重要的路由属性:match , location , history;

在组件内获取方式为:

this.props.match

this.props.location

this.props.history

#### match

匹配对象包含以下属性：  
  
 params - （object）从对应于路径的动态段的URL解析的键/值对  
 isExact - （boolean）true如果整个URL匹配（没有尾随字符)  
 path - （string）用于匹配的路径模式。作用于构建嵌套的<Route>  
 url - （string）URL的匹配部分。作用于构建嵌套的<Link>  
  
 class组件内一般用 this.props.match 获取

#### location

代表了应用程序现在的位置,例如:  
 {  
 pathname: '/somewhere',  
 search: '?some=search-string',  
 hash: '#howdy',  
 state: {[userDefined]: true }  
 }  
 class组件内一般用 this.props.location 获取. 你可能发现了history.location，但你不应该使用它，因为它是可变的.  
 可以将location传给Route,Switch,Link组件;

#### history

history对象通常具有以下属性和方法：  
  
 length : （number）历史堆栈中的条目数  
 action : （string）当前动作（PUSH，REPLACE或POP）  
 location : (object) 当前位置。具有以下属性：  
 pathname: URL的路径  
 search: URL查询字符串  
 hash: URL哈希片段  
 state: 位置特定的状态被提供给例如当这个位置被推到堆栈上时，push（路径，状态）。仅在浏览器和内存历史记录中可用。  
 push : (path, [state]) - (function) 将新条目推入历史堆栈  
 replace : (path, [state]) - (function) 替换历史堆栈上的当前条目  
 go(n) : (function) 将历史堆栈中的指针移动n个条目  
 goBack() : (function) 相当于 go(-1)  
 goForward() : (function) 相当于 go(1)  
 block : (function) 防止导航  
 listen: (function) 监听导航  
  
 history 对象是可变的。因此，建议从<Route>的渲染道具访问位置

**注意**: 默认情况下必须是经过路由匹配渲染的组件才存在this.props，才拥有路由参数，才能使用编程式导航的写法，执行this.props.history.push('/detail')跳转到对应路由的页面.

非路由匹配渲染的组件中this.props为{},自然无法使用编程式导航写法;

为了解决在非路由匹配渲染的组件中this.props为{}的情况，需使用高阶组件withRouter;

#### withRouter

withRouter是个高阶组件(高阶组件是参数为组件，返回值为新组件的函数)  
**作用**：对于不是通过路由切换过来的组件，将react-router 的 history、location、match 三个对象传入props对象上;

默认情况下必须是经过路由匹配渲染的组件才存在this.props，才拥有路由参数，才能使用编程式导航的写法，执行this.props.history.push('/detail')跳转到对应路由的页面;  
然而不是所有组件都直接与路由相连（通过路由跳转到此组件）的，当这些组件需要路由参数时，使用withRouter就可以给此组件传入路由参数，此时就可以使用this.props

### 路由跳转传参

方式一: 动态路由参数

定义路由 <Route path="/user/:id(\d+)" component={xx}></Route>  
 用Link跳转传参 <Link to="/user/id" >用户页</Link>

class组件内获取参数 this.props.match.params.id

方式二: 路由对象

用Link跳转传参

<Link to={{pathname: "/user/id" , state: { id:3 } }} >用户页</Link>

class组件内获取 this.props.history.location.state.id

方式三: push函数,隐式传参  
 this.props.history.push({  
 pathname: '/detail',  
 state: {  
 id: 3  
 }  
 })

class组件内获取 this.props.history.location.state.id

1. Redux状态管理

### 安装

npm install --save redux

redux默认提供了createStore,combineReducers 两个重要方法;

导入redux的方法

import { combineReducers, createStore } from 'redux'

### 要点

应用中所有的 state 都以一个对象树的形式储存在一个单一的 store 中。 惟一改变 state 的办法是触发 action，一个描述发生什么的对象。 为了描述 action 如何改变 state 树，你需要编写 reducers。

state

state 的形式取决于你，可以是基本类型、数组、对象、

当 state 变化时需要返回全新的对象，而不是修改传入的参数

action

改变内部 state 惟一方法是 dispatch 一个 action。

action 可以被序列化，用日记记录和储存下来，后期还可以以回放的方式执行

reducers

一个 reducer，是形式为 (state, action) => state 的纯函数。

描述了 action 如何把 state 转变成下一个 state。

store用来存放应用状态,相关操作api有:

{ subscribe, dispatch, getState }

### Redux三大原则

单一数据源 , state是只读的 , 使用纯函数来执行修改;

1. **单一数据源**

**整个应用的 [state](https://www.redux.org.cn/docs/Glossary.html" \l "state) 被储存在一棵 object tree 中，并且这个 object tree 只存在于唯一一个 [store](https://www.redux.org.cn/docs/Glossary.html" \l "store) 中.**

console.log(store.getState())

1. **state是只读的**

**唯一改变 state 的方法就是触发 [action](https://www.redux.org.cn/docs/Glossary.html" \l "action)，action 是一个用于描述已发生事件的普通对象。**

store.dispatch({

type: 'COMPLETE\_TODO',

index: 1

})

1. **使用纯函数来执行修改**

**为了描述 action 如何改变 state tree ，你需要编写 [reducers](https://www.redux.org.cn/docs/Glossary.html" \l "reducer)。**

随着应用变大，你可以把它拆成多个小的 reducers，分别独立地操作 state tree 的不同部分;

function visibilityFilter(state = 'SHOW\_ALL', action) {}

function todos(state = [], action) {}

import { combineReducers, createStore } from 'redux'

let reducer = combineReducers({ visibilityFilter, todos })

let store = createStore(reducer)

### Action

Action 本质上是 JavaScript 普通对象。我们约定，action 内必须使用一个字符串类型的 type 字段来表示将要执行的动作。多数情况下，type 会被定义成字符串常量。当应用规模越来越大时，建议使用单独的模块或文件来存放 action。

import { ADD\_TODO, REMOVE\_TODO } from '../actionTypes'

除了 type 字段外，action 对象的结构完全由你自己决定;

**我们应该尽量减少在 action 中传递的数据**。比如下面的例子，传递 index 就比把整个任务对象传过去要好。

{

type: TOGGLE\_TODO,

index: 5

}

**Action创建函数**

**Action 创建函数** 就是生成 action 的方法;

在 Redux 中的 action 创建函数只是简单的返回一个 action:

function addTodo(text) {

return {

type: ADD\_TODO,

text

}

}

### Reducer

Reducers 指定了应用状态的变化如何响应 [actions](https://www.redux.org.cn/docs/basics/Actions.html) 并发送到 store 的，记住 actions 只是描述了有事情发生了这一事实，并没有描述应用如何更新 state。

**设计State结构**

**原则**: 把所有数据放到一个对象里，每个数据以 ID 为主键，不同实体或列表间通过 ID 相互引用数据。把应用的 state 想像成**数据库**。例如，实际开发中，在 state 里同时存放 todosById: { id -> todo } 和 todos: array<id> 是比较好的方式。

**Action处理**

reducer 就是一个纯函数，接收旧的 state 和 action，返回新的 state。

(previousState, action) => newState

**保持 reducer 纯净非常重要。永远不要在 reducer 里做这些操作**：

1. 修改传入参数；
2. 执行有副作用的操作，如 API 请求和路由跳转；
3. 调用非纯函数，如 Date.now() 或 Math.random()。

function todoApp(state = initialState, action) {

switch (action.type) {

case SET\_VISIBILITY\_FILTER:

return Object.assign({}, state, {

visibilityFilter: action.filter

});

case ADD\_TODO:

return Object.assign({}, state, {

todos: [

...state.todos,

{

text: action.text,

completed: false

}

]

})

case TOGGLE\_TODO:

return Object.assign({}, state, {

todos: state.todos.map((todo, index) => {

if (index === action.index) {

return Object.assign({}, todo, {

completed: !todo.completed

})

}

return todo

})

})

default: return state;

}

}

注意:

1. **不要修改 state**。 使用 [Object.assign()](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign" \t "_blank) 新建了一个副本。不能这样使用 Object.assign(state, { visibilityFilter: action.filter })，因为它会改变第一个参数的值。你必须把第一个参数设置为空对象。你也可以使用 { ...state, ...newState } 达到相同的目的。
2. **在 default 情况下返回旧的 state。遇到未知的 action 时，一定要返回旧的 state**。

**reducer 合成**

抽出一个 reducer 来专门管理 visibilityFilter;

function visibilityFilter(state = SHOW\_ALL, action) {

switch (action.type) {

case SET\_VISIBILITY\_FILTER:

return action.filter

default:

return state

}

}

抽出一个 reducer 来专门管理 todos ;

function todos(state = [], action) {

switch (action.type) {

case ADD\_TODO:

return [

...state,

{

text: action.text,

completed: false

}

]

case TOGGLE\_TODO:

return state.map((todo, index) => {

if (index === action.index) {

return Object.assign({}, todo, {

completed: !todo.completed

})

}

return todo

})

default:

return state

}

}

现在我们可以开发一个函数来做为主 reducer，它调用多个子 reducer 分别处理 state 中的一部分数据，然后再把这些数据合成一个大的单一对象。

function todoApp(state = {}, action) {

return {

visibilityFilter: visibilityFilter(state.visibilityFilter, action),

todos: todos(state.todos, action)

}

}

Redux 提供了 [combineReducers()](https://www.redux.org.cn/docs/api/combineReducers.html) 工具类来做上面 todoApp 做的事情;

import { combineReducers } from 'redux'

const todoApp = combineReducers({

visibilityFilter,

todos

})

这个写法和todoApp写法完全等价;

[combineReducers()](https://www.redux.org.cn/docs/api/combineReducers.html) 所做的只是生成一个函数，这个函数来调用你的一系列 reducer，每个 reducer **根据它们的 key 来筛选出 state 中的一部分数据并处理**，然后这个生成的函数再将所有 reducer 的结果合并成一个大的对象。

combineReducers 接收一个拆分后的多个 reducer 函数组成的对象，返回一个新的 Reducer 函数.

combineReducers 接收一个对象，可以把所有顶级的 reducer 放到一个独立的文件中，通过 export暴露出每个 reducer 函数，然后使用 import \* as reducers 得到一个以它们名字作为 key 的 object：

import { combineReducers } from 'redux'

import \* as reducers from './reducers'

const todoApp = combineReducers(reducers)

### Store

我们知道 [action](https://www.redux.org.cn/docs/basics/Actions.html) 用来描述“发生了什么”， [reducers](https://www.redux.org.cn/docs/basics/Reducers.html) 用来根据 action 更新 state;

**Store** 就是把它们联系到一起的对象。Store 有以下职责：

维持应用的 state；

1. 提供 [getState()](https://www.redux.org.cn/docs/api/Store.html" \l "getState) 方法获取 state；
2. 提供 [dispatch(action)](https://www.redux.org.cn/docs/api/Store.html" \l "dispatch) 方法更新 state；
3. 通过 [subscribe(listener)](https://www.redux.org.cn/docs/api/Store.html" \l "subscribe) 注册监听器;
4. 通过 [subscribe(listener)](https://www.redux.org.cn/docs/api/Store.html" \l "subscribe) 返回的函数注销监听器。

根据已有的 reducer 来创建 store 是非常容易的。我们将最终的reducer导入，并传入 [createStore()](https://www.redux.org.cn/docs/api/createStore.html)。

import { createStore } from 'redux'

import todoApp from './reducers'

let store = createStore(todoApp)

发起Actions

import {

addTodo,

toggleTodo,

setVisibilityFilter,

VisibilityFilters

} from './actions'

// 打印初始状态

console.log(store.getState())

// 每次 state 更新时，打印日志

// 注意 subscribe() 返回一个函数用来注销监听器

const unsubscribe = store.subscribe(() =>

console.log(store.getState())

)

// 发起一系列 action

store.dispatch(addTodo('Learn about actions'))

store.dispatch(addTodo('Learn about reducers'))

store.dispatch(addTodo('Learn about store'))

store.dispatch(toggleTodo(0))

store.dispatch(toggleTodo(1))

store.dispatch(setVisibilityFilter(VisibilityFilters.SHOW\_COMPLETED))

// 停止监听 state 更新

unsubscribe();

### 数据流

**严格的单向数据流**是 Redux 架构的设计核心。

Redux 应用中数据的生命周期遵循下面 4 个步骤：

1. **store调用** [store.dispatch(action)](https://www.redux.org.cn/docs/api/Store.html" \l "dispatch)。

你可以在任何地方调用 [store.dispatch(action)](https://www.redux.org.cn/docs/api/Store.html" \l "dispatch)，包括组件中、XHR 回调中、甚至定时器中;

1. **store 调用传入的 reducer 函数。**

[Store](https://www.redux.org.cn/docs/basics/Store.html) 会把两个参数传入 [reducer](https://www.redux.org.cn/docs/basics/Reducers.html)： 当前的 state 树和 action。

1. **根 reducer 把多个子 reducer 输出合并成一个单一的 state 树.**

Redux 原生提供[combineReducers()](https://www.redux.org.cn/docs/api/combineReducers.html)辅助函数，来把根 reducer 拆分成多个函数，用于分别处理 state 树的一个分支。

1. **store 保存了根 reducer 返回的完整 state 树。**

这个新的树就是应用的下一个 state！所有订阅 [store.subscribe(listener)](https://www.redux.org.cn/docs/api/Store.html" \l "subscribe) 的监听器都将被调用；监听器里可以调用 [store.getState()](https://www.redux.org.cn/docs/api/Store.html" \l "getState) 获得当前 state。

现在，可以应用新的 state 来更新 UI。如果你使用了 [React Redux](https://github.com/gaearon/react-redux" \t "_blank) 这类的绑定库，这时就应该调用 component.setState(newState) 来更新。

### 搭配React使用

需要声明一下, Redux 和 React 之间没有关系。Redux 支持 React、Angular、Ember、jQuery 甚至纯 JavaScript。

#### 安装 React Redux

Redux 默认并不包含 [React 绑定库](https://github.com/reactjs/react-redux" \t "_blank)，需要单独安装。

npm install --save react-redux

react-redux默认提供了connect,Provider

导入相关方法,组件

import { connect,Provider } from 'react-redux'

#### connect 注入数据

react-redux提供connect()方法，用于从UI组件生成容器组件,并且将redux中的某个state数据挂载到UI组件的props上;

connect()有四个参数：mapStateToProps、mapDispatchToProps、mergeProps和options，我们一般常用的是mapStateToProps、mapDispatchToProps这两个参数

**mapStateToProps**负责输入数据，即将state映射到 UI 组件的参数（props）**mapDispatchToProps**负责输出交互逻辑，即将用户对 UI 组件的操作映射成 Action。这样就创建了一个容器组件;

**mapStateToProps**

使用示例:

const mapStateToProps = (state)=> {

console.log('state',state);

return {

isRight:state.login.judge,

}

};

**mapStateToProps**是一个函数，它接收**state**作为参数，返回一个对象。这个对象有一个isRight属性，代表UI组件的同名参数。

**mapStateToProps**会监听 **Store**，每当**state**更新的时候，就会自动执行，重新计算 UI 组件的参数，从而触发 UI 组件的重新渲染。

**mapStateToProps**的第一个参数总是state对象，还可以使用第二个参数，代表容器组件的props对象。

使用**ownProps**作为参数后，如果容器组件的参数发生变化，也会引发 UI 组件重新渲染。

**mapDispatchToProps**

使用示例:

const mapDispatchToProps = (dispatch) => {

return {

onJudge:(data)=>{

dispatch({type:"LOGIN",data});

}

}

};

**mapDispatchToProps**的功能是，用来建立 UI 组件的参数到**store.dispatch**方法的映射。也就是说，它定义了哪些用户的操作应该当作 Action，传给 Store。

如果**mapDispatchToProps**是一个对象，它的每个键名也是对应 UI 组件的同名参数，键值应该是一个函数，会被当作 **Action creator** ，返回的 **Action** 会由 Redux发出

#### Provider传入store

使用指定的 React Redux 组件 [<Provider>](https://github.com/reactjs/react-redux/blob/master/docs/api.md" \l "provider-store" \t "_blank) 来 [魔法般的](https://facebook.github.io/react/docs/context.html" \t "_blank) 让所有容器组件都可以访问 store，而不必显示地传递它。只需要在渲染根组件时使用即可

import React from 'react'

import { Provider } from 'react-redux'

let store = createStore(todoApp)

render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

)

1. 组件通信(附加知识)

在实际开发中,页面经过组件封装之后,组件直接的嵌套层级往往会达到3-5层; 这样会导致最外层组件向最内层组件传值, 最内层组件向最外层组件传值变得比较麻烦.

我们可以借助一个库来实现: pubsub-js

pubsub-js基于典型的消息发布-订阅模式实现数据传递，当嵌套层数比较多,有多个订阅方时，可以用此工具库;

相关方法:

//订阅

//参数一: 订阅id(频道) ; 参数二:附带参数(数据)

//返回值: 订阅引用

pubsub = Pubsub.subscribe(pubsubID, (msg, data) => {

console.log(msg) // 这里将会输出对应设置的 pubsubID

console.log(data) // 这里将会输出对应设置的参数

})

//取消指定的订阅

Pubsub.unsubscribe(pubsub)

取消全部订阅

PubSub.clearAllSubscriptions()

//发布

Pubsub.publish(StationStatisticsID, data)

// 或

PubSub.publishSync(StationStatisticsID, data)

**使用方式:**

安装

npm install pubsub-js --save

使用

// 导入 import PubSub from "pubsub-js"

// 在主动发送数据的地方进行发布

class Data extends React.Component{

pubmsg = ()=>{ PubSub.publish("频道","频道发布的消息") }

render() {

return( <button onClick={this.pubmsg}>Data组件,发布消息</button> )

}

}

// 订阅

class App extends Component {

// 接收数据方订阅,组件将要被渲染的时候进行订阅

componentWillMount() {

PubSub.subscribe("频道", (msg,data)=> { console.log(msg,data) })

}

render() {

return ( <div className="App"> <Data /> </div> );

}

}