React

**简介**：

当2014年Facebook推出React时，给整个业界带来全新的看待网页应用开发的方式，和React一同问世的Flux，也克服传统MVC框架的很多弊病。技术在不断发展，在2015年，Flux的一个变体Redux出现，进一步优化了Flux的功能。

**优秀之处：**

1. **专注视图层：（**不处理路由，数据，以及逻辑的处理**）**
2. **VirtualDom：**只对虚拟dom更新，不对真实dom更新；只对dom进行必要的更新，实现的重绘的最小化；
3. **jsx语法：**
4. 高内聚，低耦合
5. onClick 不是原生dom中的onclick事件，这里是onClict是通过react的事件委托方式，触发挂在顶层dom节点上的一个事件处理函数（提高了性能）；
6. 可以将样式一起写入组件文件中；
7. 关于jsx语法的说法：

* 在jsx语法中可以直接写入组件，注意组件为了和普通标签区别，必须第一个字母大写；
* jsx语法是历史的进步还是倒退；（样式，结构，逻辑于一体）

1. **活跃的生态圈：**（开源项目）
2. **组件式开发**
3. **手动搭建项目：**
4. npm init
5. 安装项目依赖

cnpm install --save-dev babel-core babel-loader babel-preset-stage-2 babel-plugin-transform-react-jsx babel-plugin-transform-runtime babel-preset-es2015 babel-preset-react babel-preset-stage-0 css-loader html-loader html-webpack-plugin react react-dom style-loader url-loader webpack webpack-dev-server

1. 配置babelrc文件：

{

"presets": [

"es2015",

"stage-0",

"react"

],

"plugins": ["transform-runtime", "transform-react-jsx"]

}

1. webpack.config.js的配置文件：

除了如下配置有更改，其他可以完全遵照vue的配置

{

test: /\.js|.jsx$/,

use: [

"babel-loader",

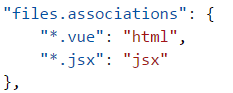
],

exclude: /node\_modules/

},

1. 在vsCode中开发react的配置：

安装jsx插件配置如下：



安装React/Redux/react-router Snippets插件，可以帮助我们快速生成一定的语句；

所有的react组件使用jsx后缀来标识；

1. **自动化构建项目：**
2. 初始化：

全局安装：

npm install –-global create- react- app

初始化项目：

create- react- app myReactApp

启动服务：

cd myReactApp

npm start

1. 组件写法：

import React,{Component} from "react"

import ClickCount from "./ClickCount.jsx"

class Main extends Component {

constructor(props){

super(props)

}

render(){

return (

<ul>

<li><ClickCount name="first" num={1}></ClickCount></li>

<li><ClickCount name="second" num={10}></ClickCount></li>

<li><ClickCount name="third" num={7}></ClickCount></li>

</ul>

)

}

}

export default Main

注：对于jsx语法的解析必须引入React，虽然这里不用，但是如果不引入会因无法解析jsx而报错；组件的定义必须遵循首字母大写

1. jsx语法使用：

属性：

1、属性值直接为字符串

2、动态解析属性值

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

react的jsx：

const element = (

<h1 className="greeting">

Hello, world!

</h1>

);

用React.createElement()同样可以达到同样的效果；

const element = React.createElement(

'h1',

{className: 'greeting'},

'Hello, world!'

);

1. 项目的分解：

npm run eject 进行项目配置文件的弹射；

1. react的工作方式：
2. 从繁琐的dom处理中解脱出来，响应式，函数式编程思维；输入相同的数据，将产出相同的效果；数据更新，dom同时做出更新；
3. Virtual Dom

Dom：结构化文本的抽象表达式；

虚拟dom：对dom树的抽象，不会触及浏览器，只存在于js内存的空间树形结构，每次的数据刷新，render函数的执行都会触发一个新旧虚拟dom 的对比，如果无改变，则不更新，有改变，只更新相应dom结构；

1. **元素的渲染**
2. **jsx语法将元素渲染到dom中**
3. **更新元素渲染：**

React 元素都是immutable 不可变的。当元素被创建之后，你是无法改变其内容或属性的；更新界面的唯一方法就是重新挂载节点

function tick() {

const element = (

<h2>{new Date().toLocaleTimeString()}.</h2>

);

ReactDOM.render(

element,

document.getElementById('root')

);

}

setInterval(tick, 1000);

1. **只会更新必要的部分(可以在谷歌开发者工具中进行dom更新的查看)**
2. **在react的jsx语法中，如果标签没有子元素，可以直接使用闭合标签；**
3. **在react的组件必须首字母进行大写，否则React.createEelement则无法查找到组件，进行编译；**
4. **jsx支持标签名为表达式；，如果使用动态标签名，则需先将表达式解析赋值给一个变量；**
5. **React组件以及基本语法：**
6. 组件的设计要素：

高内聚：把逻辑紧密相关的内容放在一个组件中；

低耦合：不同组件之间的依赖关系要尽量弱化；尽量保持每个组件的独立性；

1. 组件的生命周期：
2. 装载过程：
   1. Constructor:

初始化state；

绑定成员函数的this环境（bind、：： ）防止以解构的方式进行全局作用域进行调用；

例：this.fn=this.fn.bind(this)或this.fn=::this.fn

* 1. GetInitialState：初始化state值,必须结合React.createClass()使用，使用es6的方式，此函数无效果；

const Test=React.createClass({

getInitialState:function(){

return {

name:"zhangsan"

}

},

getDefaultProps:function(){

return {

name:"zhangsan" //这里的zhangsan相当于默认值

}

}

})

* 1. GetDefaultProps：初始化props值,必须结合React.createClass()使用，使用es6的方式，此函数无效果，用属性defaultProps代替；

class Test extends Component {

constructor(props){

super(props)

this.state={data:new Date()}

}

render(){

return (

<ul>

<li><ClickCount name="third" num={7}></ClickCount></li>

</ul>

)

}

}

Test.defaultProps={

name:"moren"

}

* 1. ComponentWillMount：在render函数调用之前进行调用；
  2. Render：并不做实际的渲染动作，只返回一个jsx描述结构，最后渲染与否由react来决定；必须是个纯函数，不能设计到数据的更变（this.setState）；

render(){

return (

<ul>

<li><ClickCount name="third" num={7}></ClickCount></li>

</ul>

)

}

* 1. ComponentDidMount ：在render函数调用之后进行调用；但不是在render函数调用之后立即调用，而是当所有的dom树挂载并渲染完成后才会调用，这是因为render函数并不进行渲染，而只是返回一个jsx对象，渲染的工作由react库进行，只有当所有的组件的jsx对象组合完毕之后，react通过对比后才会渲染，所有此钩子函数是发生的所有组件的render函数都执行后才会执行；（只能在浏览器端触发）

注：与angular中的link或者post函数有点相似，但是这里不仅指子组件，而是当前组件中的所有组件，包括兄弟组件；提供了dom操作的接口

1. 更新过程：
   1. ComponentWillReceiveProps: （参数只有props）传入的props的改变或者组件进行刷新（forceUpdate函数触发组件的刷新）都会触发此函数，但是通过this.setState改变的数据则不会触发此函数，
   2. ShouldComponentUpdate：react组件的钩子函数两个需要有返回值的钩子函数之一，另一个为render，此钩子函数的返回值为一个bolen值，如果为true时，则prop的改变以及state的改变都会引起组件的刷新，如果为false时，则不再进行渲染；此钩子函数接受两个参数，一个是nextProps，一个是nextState，可以将将要更新的值和此时的做对比，然后返回true和false来进行性能的校优；
   3. ComponentWillUpdate：跟componentWillMount相似
   4. Render
   5. ComponentDidUpdate：跟componentDidMount相似
2. 卸载过程：

ComponentWillUnmount:此钩子函数可以在组件卸载前执行，可以进行手动添加dom元素的删除，以及计时器和事件监听的移除；

1. **React组件的数据**

**组件内部数据类型：**

两种数据结构，prop和state，这两种数据的改变都会引起组件的重新渲染；

* 1. Prop：组件的外部接口，接受外部数据；跟html的属性的书写类似，但是除了接受字符串之外，还可以接受js对象，数字等的；如果组件进行数据反馈给外界，可以通过prop传入组件一个函数；
* Prop的读取：

由class类的constructor函数接受，以及super（）方法调用时的传入；

最后的props为一个对象，键值分别为传入时的属性名；这里比较方便的是使用es6的解构赋值；

* PropTypes的检查：

可以通过增加类的propTypes属性来定义prop规格，在constructor函数中进行如下定义：

Index.propTypes={ //组件全局进行配置

caption:PropTypes.string.isRequired,

initValue:PropTypes.number

}

注意：此属性不会影响组件的渲染，只是做到在开发过程中辅助开发；

* 1. State：由于组件不能改变出入的prop，所以当组件要进行自己的状态纪录时就需要用到state；
* 初始化：state的初始化可以在constructor中，通过this.state进行设置，值必须为一个js对象的格式，通常将prop传入的外部值赋给state，以便后续操作；设置默认值时，可以用 | | 操作符进行；
* 读取和更新state

更新使用this.setState({count:this.state.count++})方法；

读取为：this.state.count这种方式

* 1. Prop和state的对比：

Prop：定义外部接口；赋值在外部环境使用组件时；组件内部不能更改

State:纪录内部状态；赋值在组件内部；在组件内部可以进行更改；

1. **react中css的样式书写**
2. **全局引入css文件**

此种方式会将css文件在全局引入，也就是说当前样式文件中书写的所有样式，在任何一个组件都可以引用，此种引用方式有可能会造成全局的污染，变量名冲突，又或者说不符合组件式开发的思想；此种方式一般是引入全局的css文件；

import “app.css”，或者html中的link

1. **局部作用域（css modules的方式）**

此种方式采用的是css modules的做法，通过此做法，最后会将css文件中类名和dom中类名形成唯一的hash值，这样就不会造成类名的冲突了，使用场景，一般为组件内部的样式防止全局书写造成的类名污染；

注意webpack.config.js文件更改：use: ["style-loader","css-loader?module"],

比如如下写法：

app.css:

.title {

color: red;

}

jsx:

import style from './App.css';

export default () => {

return (

<h1 className={style.title}>

Hello World

</h1>

);

};

最后会编译成如下代码：

<h1 class="\_3zyde4l1yATCOkgn-DBWEL">

Hello World

</h1>

.\_3zyde4l1yATCOkgn-DBWEL {

color: red;

}

**注：在css Modules中还提供了一种css全局作用域的书写方式，也就是说如果通过此种方式注册的css类名不会被替换为hash值，书写方式如下：**

**css：**

**:global(.title){**

**color:red;**

**}**

1. **行内样式：**

**在jsx的dom上直接使用style属性进行样式的设置，样式的值为一个对象，这种方式的书写，可以支持css样式以js对象的方式进行书写，如果不想将组件的内部样式进行作用域隔离使用css modules的方式进行书写，则可以此种方式进行书写，可以直接在render函数中定义样式对象，也可以通过专门外部样式js文件配置样式对象，然后引入到dom进行使用，书写方式如下：**

**var style={**

**color:red;**

**}**

**<div style={style}>天气真好</div>**

**注：在书写css属性时，一定要采用驼峰式写法；**

1. **事件处理**
2. **react中的事件绑定属性采用的是驼峰式写法：**

<button onClick={this.clickfn}>点击</button>

1. **处理react使用class类定义方法时，无法自动绑定this的弊端：**

* **在constructor中使用bind为事件函数绑定this**

this.clickfn=this.clickfn.bind(this)

* **在dom中调用函数时，先使用箭头函数绑定this，箭头函数的this是在定义时形成的**

<button onClick={()=>this.clickfn()}>点击</button>

1. **jsx的onClick和html的onclick不同之处**

* **html的onclick弊端：**

注册的事件处理函数都是全局环境，污染了全局环境；

使用onclick的dom元素，如果在dom树中删除时，必须手动的注销事件处理器，否则会造成内容泄露；

* **jsx的onClick:**

挂载的每一个函数都是在组件内部，而不是全局；

无论多少个onClick都是采用的事件委托的方式，在dom树顶层添加一个事件监听函数，此函数会根据具体组件分配具体的函数**；**

1. **条件判断：**

**在react的jsx语法中，如果遇到js语句使用{}进行包裹；**

1. **if语句的使用：**

render() {

var element=null

if(this.state.stateT){

element= <div > stateT为true时 < /div>

}else{

element= <div > stateT为false时 < /div>

}

return (

<div>

<button onClick={this.changeFn}>切换</button>

{element}

</div>

)

}

1. **jsx与运算符&&的结合：**

render(){

return (

<div>

<button onClick={this.changeFn}>切换</button>

{this.state.stateT&& <div > stateT为false时 < /div>}

</div>

)

}

1. **三目运算符**

render(){

return (

<div>

<button onClick={this.changeFn}>切换</button>

{this.state.stateT?<div>stateT为true时</div>:<div> stateT为false时</div>}

</div>

)

}

1. **在jsx语法中，render可以返回null，表示不进行任何dom的渲染，但是钩子函数会执行；**
2. **列表和keys**
3. **基础用法：（key一般采用数据id，或者数据在数组中的索引）**

render() {

var element=this.state.list.map((i,index)=><li key={index}>{i}</li>)

return (

<div>

<button onClick={this.changeFn}>切换</button>

{element}

</div>

)

}

注：如果后续进行数据的重新排序，则最好不要使用索引当作key值，否则，由于key值和dom的内容同时改变，则会引起全部dom节点的重绘；如果使用的是id，则只会调整dom节点的顺序，不会引起重绘dom树；（可以用上面例子演示，推荐使用id作为标识，而不是index）

1. **key值只需要在兄弟元素之间唯一就可以，不需要是在全局唯一的**

render() {

var element=this.state.list.map((i,index)=><li key={i}>{i}</li>)

var element2=this.state.list.map((i,index)=><li key={i}>{i+1}</li>)

return (

<div>

<button onClick={this.changeFn}>切换</button>

<ul>

{element}

</ul>

<ul>

{ element2}

</ul>

</div>

)

}

1. **在jsx中使用map进行遍历**

render() {

return (

<ul>

<button onClick={this.changeFn}>切换</button>

{this.state.list.map((i)=><li key={i}>{i}</li>)}

</ul>

)

}

1. **表单**

**在vue中，我们知道由于vue的mvvm架构，v与m之间是响应式的，可以实时同步，但是在react中不存在数据的双向数据绑定，只是负责view层的渲染，所以我们在react中取表单中的值时需要一定的方式**

1. **受控组件（采用react的单向数据流方式实现vue一样的双向数据绑定）**

changeFn(event){

this.setState({

name:event.target.value

})

}

render() {

return (

<div>

<p>表单</p>

<label htmlFor="name">name:</label>

<input type="text" id="name" value={this.state.name} onChange={this.changeFn}/>

<p>{this.state.name}</p>

</div>

)

}

**如果想对从input中获取到的值进行格式化时，则如下操作就可以实现：**

this.setState({

name:event.target.value.toUpperCase()

})

1. **textarea标签在react中设置和读取值都采用value的方式**
2. **在react中的使用select，并实时获取其选择的值：**

render() {

return (

<div>

<p>表单</p>

<label htmlFor="name">name:</label>

<select value={this.state.name} onChange={this.changeFn}>

<option value="one">one</option>

<option value="two">two</option>

<option value="three">three</option>

</select>

<p>{this.state.name}</p>

</div>

)

}

1. **在react中实现多表单元素值得获取**

changeFnAll(event){

let target=event.target

let name=target.name

let value=target.type==="checkbox"?target.checked:target.value

this.setState({

[name]:value

})

}

// 实现多表单元素值的获取

render() {

return (

<div>

<p>表单</p>

<input type="checkbox" value={this.state.checkbox} onChange={this.changeFnAll} name="checkbox"/>

<input type="text" value={this.state.text} onChange={this.changeFnAll} name="text"/>

<p>{this.state.checkbox.toString()}</p>

<p>{this.state.text}</p>

</div>

)

}

1. **组件间通讯**
2. **父组件向子组件传递数据：**
3. **字符串常量的传递：**

<MyComponent message="hello world" />

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

1. **没有给属性传值，它默认为 true**

<MyTextBox autocomplete />

**不建议如下写法，容易与es6的对象简洁表示法冲突**

<MyTextBox autocomplete={true} />

1. **使用扩展运算符进行属性的设置**

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

<Greeting {...props} />;

1. **子组件向父组件传递数据**

只能通过将父组件的一个方法用prop的方式传入子组件，在子组件触发更变时，调用这个传入的方法，然后把子组件更变后的值以参数的形式传入父组件；从而达到父组件和子组件的通信的效果；

1. **兄弟组件间的通讯**
2. **状态提升**

对于没有直接关联关系的两个节点，就如 Child\_1 与 Child\_2 之间的关系，他们唯一的关联点，就是拥有相同的父组件。参考之前介绍的两种关系的通讯方式，如果我们向由 Child\_1 向 Child\_2 进行通讯，我们可以先通过 Child\_1 向 Parent 组件进行通讯，再由 Parent 向 Child\_2 组件进行通讯

1. **观察者模式**

也就是像angular和vue的事件派发机制，在这里react没有内置的事件派发模式，所以需要手动封装一个事件派发机制；

**如下代码：**

var watchObj = {

arrFn: {},

on(name, fn) {

if (this.arrFn[name] === undefined) {

this.arrFn[name] = []

}

this.arrFn[name].push(fn)

},

emit() {

let name, args;

if (arguments.length == 0) {

return false;

}

name = arguments[0]

args = [].concat(Array.prototype.slice.call(arguments, 1));

if (this.arrFn[name] !== undefined && this.arrFn[name].length > 0) {

this.arrFn[name].forEach(function(i) {

i(...args)

}, this);

}

}

}

export default watchObj

1. **组合与继承**

**在vue中的有一种组件特性叫内容分发，可以在父组件中自定义子组件的dom结构，在react中也提供了这么一种机制**

1. **包含关系：(在子组件通过this.props.children获取父组件中写在子组件开闭标签间的dom节点)**

**父组件：**

render() {

return (

<div>

<Children>

<p>相当于vue的内容分发</p>

</Children>

</div>

)

}

**子组件：**

render() {

return (

<div>

<p>这里是子组件</p>

{this.props.children}

</div>

)

}

1. **子组件中动态渲染其他组件：(在父组件中将其他组件动态绑定到prop属性上，在子组件中通过相应的属性来获取)**

**父组件：**

render() {

return (

var one= <span>这里为true</span>

var two=<span>这里为false</span>

return (

<div>

<Slot one={one} two={two}></Slot>

</div>

)

)

}

**子组件：**

render() {

return <div>

<p> 这里是子组件</p>

{this.state.state?this.props.one:this.props.two}

</div>

}

1. **高阶组件：**

**高阶组件（HOC）是react中对组件逻辑进行重用的高级技术。但高阶组件本身并不是React API。它只是一种模式，这种模式是由react自身的组合性质必然产生的**

高阶组件就是一个函数，且该函数接受一个组件作为参数，并返回一个新的组件

**注：**

1. **在使用高阶组件时，一定不能修改原始组件的任何地方（指样式或结构）；**
2. **高阶组件和**容器组件**的相似之处。容器组件是专注于在高层次和低层次关注点之间进行责任划分的策略的一部分。容器组件会处理诸如数据订阅和状态管理等事情，并传递props属性给展示组件。而展示组件则负责处理渲染UI等事情。高阶组件使用容器组件作为实现的一部分。你也可以认为高阶组件就是参数化的容器组件定义。**
3. **对于**

**需求：在两个projectList组件和userList组件中需要对传入的数据做排序和添加时间属性；**

**Hoc高阶组件：**

import React, { Component } from 'react'

// 给一个数组中对象添加时间属性和排序的功能

const hocComponentFn = (Wcomponent) => {

return class extends Component {

constructor(props) {

super(props)

this.state ={...this.props}

}

// 产生并格式化日期

dateFormat() {

let date = new Date()

let fmt="yyyy-MM-dd hh:mm:ss"

var o = {

"M+": date.getMonth() + 1, //月份

"d+": date.getDate(), //日

"h+": date.getHours(), //小时

"m+": date.getMinutes(), //分

"s+": date.getSeconds(), //秒

"q+": Math.floor((date.getMonth() + 3) / 3), //季度

"S": date.getMilliseconds() //毫秒

};

if (/(y+)/.test(fmt))

fmt = fmt.replace(RegExp.$1, (date.getFullYear() + "").substr(4 - RegExp.$1.length));

for (var k in o)

if (new RegExp("(" + k + ")").test(fmt))

fmt = fmt.replace(RegExp.$1, (RegExp.$1.length == 1) ? (o[k]) : (("00" + o[k]).substr(("" + o[k]).length)));

return fmt;

}

// 排序以及添加时间

handle() {

let data = [...this.state.data]

data.sort((a, b) => {

return a.id > b.id ? -1 : 1

})

data = data.map((i) => {

i.time=this.dateFormat()

return i

})

console.log(data)

this.setState({

data:data

})

}

componentWillMount(){

this.handle()

}

// 将原组件使用render进行抛出

render() {

return (

<Wcomponent {...this.state}/>

)

}

}

}

export default hocComponentFn

**ProjectList渲染组件：**

import React, { Component } from 'react'

class ProjectList extends Component {

constructor(props) {

super(props)

}

render() {

const $el=this.props.data.map((i,index)=>{

return <p key={index}>{i.id}----{i.title}----{i.time}</p>

})

return (

<div>

{$el}

</div>

)

}

}

export default ProjectList

**UserList渲染组件：**

import React, { Component } from 'react'

class UserList extends Component {

constructor(props) {

super(props)

}

render() {

const $el=this.props.data.map((i,index)=>{

return <p key={index}>{i.id}----{i.name}----{i.time}</p>

})

return (

<div>

{$el}

</div>

)

}

}

export default UserList

**数据传入，渲染projectList和userList的父组件：**

import React, { Component } from 'react'

import sortHoc from "./Hoc"

import ProjectList from "./projectList"

import UserList from "./userList"

const WithProjectList=sortHoc(ProjectList)

const WithUserList=sortHoc(UserList)

class Index extends Component {

constructor(props) {

super(props)

this.state = {

userList: [

{id:3, name: "zhangsan" },

{id:1, name: "lisi" },

{id:2, name: "wangwu" }

],

projectList: [

{ id:4,title: "电脑" },

{ id:2, title: "手机" },

{ id:7, title: "笔记本" }

]

}

}

render() {

return (

<div>

<h1>projectList组件</h1>

<WithProjectList data={this.state.projectList}/>

<hr/>

<h1>userList组件</h1>

<WithUserList data={this.state.userList}/>

</div>)

}

}

export default Index

1. **PropTypes类型检查**

**在react的15.5版本后，React.PropTypes已经废弃了，需要单独引入prop-types库进行验证**

import PropTypes from 'prop-types'

class Children extends Component {

constructor(props) {

super(props)

}

render() {

return (

<div>

<p>这里是子组件</p>

{this.props.name}

</div>

)

}

}

Children.propTypes={

name:PropTypes.string

}

**prop-types库可以校验的类型如下：**

import PropTypes from 'prop-types';

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,

// 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

optionalArrayOf: 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,

// 你也可以指定一个自定义验证器。它应该在验证失败时返回

// 一个 Error 对象而不是 `console.warn` 或抛出异常。

// 不过在 `oneOfType` 中它不起作用。

customProp: function(props, propName, componentName) {

if (!/matchme/.test(props[propName])) {

return new Error(

'Invalid prop `' + propName + '` supplied to' +

' `' + componentName + '`. Validation failed.'

);

}

},

// 不过你可以提供一个自定义的 `arrayOf` 或 `objectOf`

// 验证器，它应该在验证失败时返回一个 Error 对象。 它被用

// 于验证数组或对象的每个值。验证器前两个参数的第一个是数组

// 或对象本身，第二个是它们对应的键。

customArrayProp: PropTypes.arrayOf(function(propValue, key, componentName, location, propFullName) {

if (!/matchme/.test(propValue[key])) {

return new Error(

'Invalid prop `' + propFullName + '` supplied to' +

' `' + componentName + '`. Validation failed.'

);

}

})

};