**React面试题**

1. **React基础知识（常用概念）**
2. **谈谈你对react.js的理解**

两句话描述：

ReactJs把视图更新简化为一个render函数

render函数接收两个参数，分别是配置项和状态

React并不是一个框架，React提供了一些新颖的概念、库 和编程原则让你能够同时在服务端和客户端编写快速、紧凑、漂亮的代码来构建 你的web应用。

细致描述：

ReactJs是一个专注于View的Web前端应用。Web前端的View就是浏览器中的Dom元素，改变View的唯一途径就是修改浏览器中的Dom元素，因此ReactJs的核心任务就是如何修改Dom元素，作为一个成功的框架，ReactJs使修改Dom元素变得高效而又简单。

ReactJs把修改Dom的操作简化成一个函数renderInto(parentDom, props, states)=>htmlString，这个函数的意图就是根据props，states计算出视图对应的html字符串并添加为parentDom的子节点。props和states就是普通的javascript对象，这个函数的核心逻辑就是计算html元素的机构及元素属性然后拼接成字符串返回。作为框架，ReactJs用JSX形式的DSL解决了拼接html的任务并接管了更新到parentDom的职责。

1. **reactjs的有什么特点？**

React的有两个特点：组件化和高效的虚拟DOM

对React应用而言，你需要分割你的页面，使其成为一个个的组件。也就是说，你的应用是由这些组件组合而成的。你可以通过分割组件的方式去开发复杂的页面或某个功能区块，并且组件是可以被复用的。这个过程大概类似于用乐高积木去瓶装不同的物体。我们称这种编程方式称为 ****组件驱动开发****。

React的一大特点是其所拥有的虚拟DOM，它让页面渲染变得非常的高效，并且比直接操纵DOM变得更为可控。这两大特点的组合使得React具有强大的自上而下的页面渲染能力。

**3.使用react会涉及哪些常用的概念和技术？**

ES6 React

虚拟DOM（virtual DOM）

组件驱动开发（component-driven development）

不变性（immutability）

自上而下的渲染（top-down rendering）

渲染路径和优化

打包工具, ES6, 构建请求, debugging, 路由等

同构React（isomorphic React）

### **4.Virtual DOM**

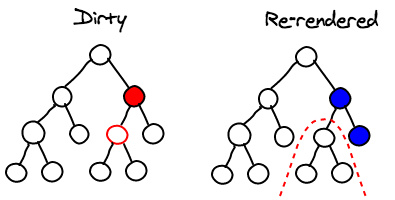
为了跟踪模型层的变化，并且将其应用到DOM中（也就是渲染），我们需要注意两个 重要的事情：

1. 数据是什么时候改变的
2. 哪一个（些）DOM元素需要被更新

对于（1）而言，React提供了一个观察者模型用于替代传统的脏检查（dirty checking）， 也就是持续的检查模型的变化。这也就是解释了为什么React不需要计算哪些发生 了改变的原因，因为它会立即知道。这个过程减少了计算量，并它应用程序变得 更平滑。但这里真正有趣的是，****React是如何管理DOM操纵的****。

对于DOM改变（2）而言，React在内存中构建了DOM的树形表示，并且计算出哪个 DOM元素应该被改变。对浏览器而言，DOM操纵是比较耗费性能的，因此我们更倾向于 让其变得最小化。幸运的是，React视图尽可能少的触及到DOM元素。给予对象表示而言， 更少的DOM操纵意味着计算会更快，因此DOM改变也被尽可能的减少。

React在底层实现了一个diffing算法，该算法使用DOM的树形表示法，当某个 节点发生变化（标记为dirty）时它会重新计算整个子树，你会注意到你的模型发生 了改变，因为整个子树在之后会被重新渲染。



### **5.如何在服务端渲染**

因为React在DOM表示时使用了一个虚拟（假的）DOM，因此借助于这种方式使得在服务端 渲染输出HTML称为可能（不借助于JSDom, PhantomJS等）。React还能智能的识别出 服务端渲染出来的页面标记，并在客户端只为这些标记添加事件处理器，这对构建 同构web app非常有用。

有意思的是，React渲染出来的HTML标记都包含了data-reactid属性，这有助于 React中追踪DOM节点。

### 6.**如何使用React的方式来思考组件开发**

对于构建一个可过滤的产品列表而言，通常其包括如下的组件结构：

<https://facebook.github.io/react/docs/thinking-in-react.html>

## 7.**组件生命周期**

每个React组件在加载时都有特定的生命周期，在此期间不同的方法会被执行。 下面简单介绍React组件的生命周期：

### **componentWillMount**

该方法会在组件render之前执行，并且永远只执行一次。

### **componentDidMount**

该方法会在组件加载完毕之后立即执行。此时，组件已经完成了DOM结构的渲染， 并可以通过 **this**.getDOMNode() 方法来访问。

### **componentWillReceiveProps**

组件接收到一个新的prop时会被执行，且该方法在初始render时不会被调用。

### **shouldComponentUpdate**

在组件接收到新的props或state时被执行。

### **componentWillUpdate**

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

### **componentDidUpdate**

在组件完成更新后立即执行。在初始化时不会被执行。 一般会在组件完成更新后被使用。

### **componentWillUnMount**

在组件从DOM中unmount后立即执行。该方法主要用来执行一些必要的清理任务。

## 8.**在打包时使用Webpack和Babel**

我们会经常用到一些工具，首先一个是node.js的模块系统和它的包管理工具npm。 我们会编写node风格的代码来require我们需要的东西。并且react本身也是一个独立的 npm包。

通常你有两种选择，commonJS或者ES6：

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **var** React = require('react/addon');    **var** MyComponent = React.createClass({  *// do something*  });    module.exports = MyComponent; |

或者

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5 | import React from 'react/addons';  **class** MyComponent **extends** React.Component {  *// do something use es6*  }  export **default** MyComponent; |

例如，我们会使用[debug](https://www.npmjs.com/package/debug)模块来调试， 使用[superagent](https://www.npmjs.com/package/superagent)模块来编写请求。

现在，我们有了Node的依赖管理系统，并且使用npm来提供模块。下面我们需要做的 事：选择一个合适的库来打包我们的代码，并且能够让其运行在浏览器上。

因此我们需要一个打包器。目前最流行的解决方案包括两个，分别是[Browserify](http://browserify.org/)和 [Webpack](http://webpack.github.io/)。我们选择使用Webpack，因为Webpack 更适合于React社区。

1. **Webpack是如何让工作的？**

Webpack用于打包我们的代码，并且包含进我们需要的包，然后输出为浏览器可运行的 文件。因为我们使用JSX和ES6，因此我们需要相应的工具来将其转换为ES5。事实上, Babel能够同时做这两件事。使用Webpack能够很轻松的完成这些任务，因为Webpack 是面向配置的。

使用如下命令开始：

Shell

|  |  |
| --- | --- |
| 1  2  3  4 | npm init  npm install webpack --save-dev  npm install babel --save-dev  npm install babel-loader --save-dev |

然后创建 webpack.config.js 文件，我们需要使用ES5来编写该文件，因为它是 webpack的配置文件。一个典型的配置方式如下：

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | **var** path = require('path');    module.exports = {    entry: path.resolve(\_\_dirname, '../src/client/scripts/client.js'),    output: {      path: path.resolve(\_\_dirname, '../dist'),      filename: 'bundle.js'    },      module: {      loaders: [        {          test: /src\/.+.js$/,          exclude: /node\_modules/,          loader: 'babel'        }      ]    }  }; |

运行 webpack 命令你可以执行打包流程。这之后你可以只在页面中包含 bundle.js 即可。 如下：

|  |  |
| --- | --- |
| 1 | <script src='bundle.js'></script> |

(提示：你可以使用 node-static 来存放你的静态资源文件，使用 npm install -g node-static 来安装，并使用 static .来启动)。

### **10.项目结构**

一个典型的项目结构你可以参考[这个仓库](https://github.com/RisingStack/react-way-getting-started)。

config/

    app.js

    webpack.js (js config over json -> flexible)

src/

  app/ (the React app: runs on server **and** client too)

    components/

      \_\_tests\_\_ (Jest test folder)

      AppRoot.jsx

      Cart.jsx

      Item.jsx

    index.js (just **to** export app)

    app.js

  client/  (only browser: attach app **to** DOM)

    styles/

    scripts/

      client.js

    index.html

  server/

    index.js

    server.js

.gitignore

.jshintrc

**package**.json

README.md

**11.如何测试react组件？**

对于React组件的测试，这里推荐使用[Jest](https://facebook.github.io/jest/), Jest也是由Facebook提供的测试框架，并且有很多强大的特性，但这里并会详细的 介绍它们。

关于Jest，我推荐你阅读和尝试来自Facebook的[https://facebook.github.io/jest/docs/tutorial-react.html#content](https://facebook.github.io/jest/docs/tutorial-react.html" \l "content)。

对于ES6代码的测试，你可以参考<https://github.com/facebook/jest/tree/master/examples/react-es6>。

1. **React路由**
2. **谈谈你对react路由的理解**

[React Router](https://github.com/rackt/react-router) 是专为 React 设计的路由解决方案，在使用 React 来开发 SPA （单页应用）项目时，都会需要路由功能，而 React Router 应该是目前使用率最高的。

React Router 并不是 Facebook 的 React 官方团队开发的，但是据说有官方人员参与开发。React Router 的设计思想来源于 Ember 的路由，如果原来有用过 Ember 的路由，那么应该对 React Router 不会陌生。

### **什么是路由？**

对于没有开发过后端，也没有开发过 SPA 的前端来说，「路由」这个名词可能会让人比较困惑，这里的路由并不是指「硬件路由」，也不是网络七层协议中的「网络层路由」，但是其思想原理是一样的。我尽量简单通俗的介绍一下。

假如我们有一台提供 Web 服务的服务器的网络地址是：10.0.0.1，而该 Web 服务又提供了三个可供用户访问的页面，其页面 URI 分别是：

|  |  |
| --- | --- |
| 1  2  3 | http:*//10.0.0.1/*  http:*//10.0.0.1/about*  http:*//10.0.0.1/concat* |

那么其路径就分别是 /，/about，/concat。

当用户使用 http://10.0.0.1/about 来访问该页面时，Web 服务会接收到这个请求，然后会解析 URI 中的路径 /about，在 Web 服务的程序中，该路径对应着相应的处理逻辑，程序会把请求交给路径所对应的处理逻辑，这样就完成了一次「路由分发」，这个分发就是通过「路由」来完成的。

### **前端路由**

前端的路由和后端的路由在实现技术上不一样，但是原理都是一样的。在 HTML5 的 history API 出现之前，前端的路由都是通过 hash 来实现的，hash 能兼容低版本的浏览器。如果我们把上面例子中提到的 3 个页面用 hash 来实现的话，它的 URI 规则中需要带上 #。

|  |  |
| --- | --- |
| 1  2  3 | http:*//10.0.0.1/*  http:*//10.0.0.1/#/about*  http:*//10.0.0.1/#/concat* |

Web 服务并不会解析 hash，也就是说 # 后的内容 Web 服务都会自动忽略，但是 JavaScript 是可以通过 window.location.hash 读取到的，读取到路径加以解析之后就可以响应不同路径的逻辑处理。

[history](https://html.spec.whatwg.org/multipage/browsers.html" \l "the-history-interface) 是 HTML5 才有的新 API，可以用来操作浏览器的 session history (会话历史)。基于 history 来实现的路由可以和最初的例子中提到的路径规则一样。

|  |  |
| --- | --- |
| 1  2  3 | http:*//10.0.0.1/*  http:*//10.0.0.1/about*  http:*//10.0.0.1/concat* |

用户可能都察觉不到该访问地址是 Web 服务实现的路由还是前端实现的路由。

从性能和用户体验的层面来比较的话，后端路由每次访问一个新页面的时候都要向服务器发送请求，然后服务器再响应请求，这个过程肯定会有延迟。而前端路由在访问一个新页面的时候仅仅是变换了一下路径而已，没有了网络延迟，对于用户体验来说会有相当大的提升。

说了这么多的「路由基础」，该回头来说说 React Router 了。

### **配置路由**

使用 React Router 来配置上面例子中的三个页面，每个页面分别对应着一个 React Component。

/about 页面的入口文件 [about.js](https://github.com/chenmnkken/react-demo/blob/master/react-router-demo/src/js/component/about.js)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import React from 'react';    **class** About **extends** React.Component {      render () {  **return** (  **This** **is** About page.          );      }  };    export **default** About; |

/concat 页面的入口文件 [concat.js](https://github.com/chenmnkken/react-demo/blob/master/react-router-demo/src/js/component/concat.js)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | import React from 'react';    const Concat = () => {  **return** (  **This** **is** Concat page.      );  };    export **default** Concat; |

/ 首页对应的是 [app.js](https://github.com/chenmnkken/react-demo/blob/master/react-router-demo/src/js/component/app.js)，它也是整个 React Component 的入口文件。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import React from 'react';  import {Link} from 'react-router';    **class** App **extends** React.Component {      render () {  **return** (                    React Router Demo                        by stylechen.com                    Home                      About                      Concat                    {**this**.props.children}            )      }  };    export **default** App; |

app.js 中的 Link 组件是 React Router 提供的组件，用于链接到相应页面。如果直接使用 a 标签的话相当于页面跳转了，而使用 Link 只是应用内的路由跳转，页面跳转意味着先重新加载整个页面，然后才是应用内部的路由跳转。

[index.js](https://github.com/chenmnkken/react-demo/blob/master/react-router-demo/src/js/index.js) 中包含了 route 的配置，同时在该文件中对 React Component 进行 render。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import React from 'react';  import ReactDOM from 'react-dom';  import createBrowserHistory from 'history/lib/createBrowserHistory';    import App from './component/app';  import About from './component/about';  import Concat from './component/concat';    import {Router, Route} from 'react-router';    const history = createBrowserHistory();    const router = (    );    ReactDOM.render(      router,      document.getElementById('root')  ); |

Route 组件就是用于配置路由，path 属性用于配置路径，component 就是对应的 React Component。

Route 组件支持嵌套，嵌套时子组件的路径可以继承父组件的路径，上面的 about 嵌套后就成了 /about，当然也可以直接以根路径为开头。

此时再看看 app.js 中的 this.props.children，About 和 Concat 两个页面组件其实就是以这种形式插入到父组件中，只是插入的时候用 React Router 提供的组件再包装了一下使它们可以支持路由。

Router组件还有一个重要的属性，那就是history，这可以配置使用history来实现路由，如果没有配置这个属性则默认使用hash。

### **路由参数**

假如我们有很多 list 页面，这些页面除了动态内容不同，其他的页面部分都相同，这个时候需要怎么配置路由和组件呢？

这种场景就需要用到路由的参数功能，增加一条包含参数的路由配置。

|  |  |
| --- | --- |
| 1 | import List from './component/list'; |

注意 path 属性中的 :id 就是该路由的参数（param）。再来看看 List 页面的组件。

/list 对应了 [list.js](https://github.com/chenmnkken/react-demo/blob/master/react-router-demo/src/js/component/list.js)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import React from 'react';    **class** List **extends** React.Component {      render () {  **return** (    **This** **is** List page.                  The list page id **is**                      {**this**.props.params.id}            );      }  };    export **default** List; |

在 List 组件中，可以直接通过 this.props.params.id 来访问实际的参数值（这里的id key 就和定义路径的 :id 相对应），React Router 将路由的数据都通过 props 传递给了页面组件，这样就可以非常方便的访问路由相关的数据了。

### **组件的生命周期**

每个 React Component 都有生命周期，按照常规的策略，当调用父组件的 render 的时候，会将所有的页面子组件也进行 render，这种逻辑显然不合理了。那么当一个 React 的应用有了路由功能后，它的生命周期会如何处理呢？

当切换路由的路径的时候才去 render 对应的页面组件，给 About 绑定两个组件装载和卸载的事件，就可以测试出来了。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import React from 'react';    **class** About **extends** React.Component {      componentDidMount () {          console.log('mount');      }        componentWillUnmount () {          console.log('un mount');      }        render () {  **return** (  **This** **is** About page.          );      }  };    export **default** About; |

在切换到 /about 页面的时候，会打印出 mount，而当离开页面的时候会打印出 un mount。路由的这种进入时装载组件离开时卸载组件的策略就可以做到合理利用「资源」，不会一下把所有的组件都装载进来使内存占用飙升，也不会离开时没有卸载而时内存泄漏。

1. **其他**

# **1.**调用 setState 之后发生了什么？

在代码中调用setState函数之后，React 会将传入的参数对象与组件当前的状态合并，然后触发所谓的调和过程（Reconciliation）。经过调和过程，React 会以相对高效的方式根据新的状态构建 React 元素树并且着手重新渲染整个UI界面。在 React 得到元素树之后，React 会自动计算出新的树与老树的节点差异，然后根据差异对界面进行最小化重渲染。在差异计算算法中，React 能够相对精确地知道哪些位置发生了改变以及应该如何改变，这就保证了按需更新，而不是全部重新渲染。

# 2.React 中 Element 与 Component 的区别是？

React Element 是描述屏幕上所见内容的数据结构，是对于 UI 的对象表述。典型的 React Element 就是利用 JSX 构建的声明式代码片然后被转化为createElement的调用组合。而 React Component 则是可以接收参数输入并且返回某个 React Element 的函数或者类。

# 3.在什么情况下你会优先选择使用 Class Component 而不是 Functional Component？

在组件需要包含内部状态或者使用到生命周期函数的时候使用 Class Component ，否则使用函数式组件。

# 4.React 中 refs 的作用是什么？

Refs 是 React 提供给我们的安全访问 DOM 元素或者某个组件实例的句柄。我们可以为元素添加ref属性然后在回调函数中接受该元素在 DOM 树中的句柄，该值会作为回调函数的第一个参数返回：

class CustomForm extends Component {

handleSubmit = () => {

console.log("Input Value: ", this.input.value)

}

render () {

return (

<form onSubmit={this.handleSubmit}>

<input

type='text'

ref={(input) => this.input = input} />

<button type='submit'>Submit</button>

</form>

)

}

}

上述代码中的input域包含了一个ref属性，该属性声明的回调函数会接收input对应的 DOM 元素，我们将其绑定到this指针以便在其他的类函数中使用。另外值得一提的是，refs 并不是类组件的专属，函数式组件同样能够利用闭包暂存其值：

function CustomForm ({handleSubmit}) {

let inputElement

return (

<form onSubmit={() => handleSubmit(inputElement.value)}>

<input

type='text'

ref={(input) => inputElement = input} />

<button type='submit'>Submit</button>

</form>

)

}

# 5.React 中 keys 的作用是什么？

Keys 是 React 用于追踪哪些列表中元素被修改、被添加或者被移除的辅助标识。

render () {

return (

<ul>

{this.state.todoItems.map(({task, uid}) => {

return <li key={uid}>{task}</li>

})}

</ul>

)

}

在开发过程中，我们需要保证某个元素的 key 在其同级元素中具有唯一性。在 React Diff 算法中 React 会借助元素的 Key 值来判断该元素是新近创建的还是被移动而来的元素，从而减少不必要的元素重渲染。此外，React 还需要借助 Key 值来判断元素与本地状态的关联关系，因此我们绝不可忽视转换函数中 Key 的重要性。

# 6.如果你创建了类似于下面的Twitter元素，那么它相关的类定义是啥样子的？

<Twitter username='tylermcginnis33'>

{(user) => user === null

? <Loading />

: <Badge info={user} />}

</Twitter>

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

import fetchUser from 'twitter'

// fetchUser take in a username returns a promise

// which will resolve with that username's data.

class Twitter extends Component {

// finish this

}

如果你还不熟悉回调渲染模式（Render Callback Pattern），这个代码可能看起来有点怪。这种模式中，组件会接收某个函数作为其子组件，然后在渲染函数中以props.children进行调用：

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

import fetchUser from 'twitter'

class Twitter extends Component {

state = {

user: null,

}

static propTypes = {

username: PropTypes.string.isRequired,

}

componentDidMount () {

fetchUser(this.props.username)

.then((user) => this.setState({user}))

}

render () {

return this.props.children(this.state.user)

}

}

这种模式的优势在于将父组件与子组件解耦和，父组件可以直接访问子组件的内部状态而不需要再通过Props传递，这样父组件能够更为方便地控制子组件展示的UI界面。譬如产品经理让我们将原本展示的Badge替换为Profile，我们可以轻易地修改下回调函数即可：

<Twitter username='tylermcginnis33'>

{(user) => user === null

? <Loading />

: <Profile info={user} />}

</Twitter>

# 7.Controlled Component 与 Uncontrolled Component 之间的区别是什么？

React 的核心组成之一就是能够维持内部状态的自治组件，不过当我们引入原生的HTML表单元素时（input,select,textarea 等），我们是否应该将所有的数据托管到 React 组件中还是将其仍然保留在 DOM 元素中呢？这个问题的答案就是受控组件与非受控组件的定义分割。受控组件（Controlled Component）代指那些交由 React 控制并且所有的表单数据统一存放的组件。譬如下面这段代码中username变量值并没有存放到DOM元素中，而是存放在组件状态数据中。任何时候我们需要改变username变量值时，我们应当调用setState函数进行修改。

class ControlledForm extends Component {

state = {

username: ''

}

updateUsername = (e) => {

this.setState({

username: e.target.value,

})

}

handleSubmit = () => {}

render () {

return (

<form onSubmit={this.handleSubmit}>

<input

type='text'

value={this.state.username}

onChange={this.updateUsername} />

<button type='submit'>Submit</button>

</form>

)

}

}

而非受控组件（Uncontrolled Component）则是由DOM存放表单数据，并非存放在 React 组件中。我们可以使用 refs 来操控DOM元素：

class UnControlledForm extends Component {

handleSubmit = () => {

console.log("Input Value: ", this.input.value)

}

render () {

return (

<form onSubmit={this.handleSubmit}>

<input

type='text'

ref={(input) => this.input = input} />

<button type='submit'>Submit</button>

</form>

)

}

}

竟然非受控组件看上去更好实现，我们可以直接从 DOM 中抓取数据，而不需要添加额外的代码。不过实际开发中我们并不提倡使用非受控组件，因为实际情况下我们需要更多的考虑表单验证、选择性的开启或者关闭按钮点击、强制输入格式等功能支持，而此时我们将数据托管到 React 中有助于我们更好地以声明式的方式完成这些功能。引入 React 或者其他 MVVM 框架最初的原因就是为了将我们从繁重的直接操作 DOM 中解放出来。

# 8.在生命周期中的哪一步你应该发起 AJAX 请求？

我们应当将AJAX 请求放到 componentDidMount 函数中执行，主要原因有下：

React 下一代调和算法 Fiber 会通过开始或停止渲染的方式优化应用性能，其会影响到 componentWillMount 的触发次数。对于 componentWillMount 这个生命周期函数的调用次数会变得不确定，React 可能会多次频繁调用 componentWillMount。如果我们将 AJAX 请求放到 componentWillMount 函数中，那么显而易见其会被触发多次，自然也就不是好的选择。

如果我们将 AJAX 请求放置在生命周期的其他函数中，我们并不能保证请求仅在组件挂载完毕后才会要求响应。如果我们的数据请求在组件挂载之前就完成，并且调用了setState函数将数据添加到组件状态中，对于未挂载的组件则会报错。而在 componentDidMount 函数中进行 AJAX 请求则能有效避免这个问题。

# 9.shouldComponentUpdate 的作用是啥以及为何它这么重要？

shouldComponentUpdate 允许我们手动地判断是否要进行组件更新，根据组件的应用场景设置函数的合理返回值能够帮我们避免不必要的更新。

# 10.如何告诉 React 它应该编译生产环境版本？

通常情况下我们会使用 Webpack 的 DefinePlugin 方法来将 NODE\_ENV 变量值设置为 production。编译版本中 React 会忽略 propType 验证以及其他的告警信息，同时还会降低代码库的大小，React 使用了 Uglify 插件来移除生产环境下不必要的注释等信息。

# 11.为什么我们需要使用 React 提供的 Children API 而不是 JavaScript 的 map？

React.Children.map(props.children, () => ) instead of props.children.map(() => )

props.children并不一定是数组类型，譬如下面这个元素：

<Parent>

<h1>Welcome.</h1>

</Parent>

如果我们使用props.children.map函数来遍历时会受到异常提示，因为在这种情况下props.children是对象（object）而不是数组（array）。React 当且仅当超过一个子元素的情况下会将props.children设置为数组，就像下面这个代码片：

<Parent>

<h1>Welcome.</h1>

<h2>props.children will now be an array</h2>

</Parent>

这也就是我们优先选择使用React.Children.map函数的原因，其已经将props.children不同类型的情况考虑在内了。

# 12.概述下 React 中的事件处理逻辑

为了解决跨浏览器兼容性问题，React 会将浏览器原生事件（Browser Native Event）封装为合成事件（SyntheticEvent）传入设置的事件处理器中。这里的合成事件提供了与原生事件相同的接口，不过它们屏蔽了底层浏览器的细节差异，保证了行为的一致性。另外有意思的是，React 并没有直接将事件附着到子元素上，而是以单一事件监听器的方式将所有的事件发送到顶层进行处理。这样 React 在更新 DOM 的时候就不需要考虑如何去处理附着在 DOM 上的事件监听器，最终达到优化性能的目的。

# 13.createElement 与 cloneElement 的区别是什么？

createElement 函数是 JSX 编译之后使用的创建 React Element 的函数，而 cloneElement 则是用于复制某个元素并传入新的 Props。

# 14.传入 setState 函数的第二个参数的作用是什么？

该函数会在setState函数调用完成并且组件开始重渲染的时候被调用，我们可以用该函数来监听渲染是否完成：

this.setState(

{ username: 'tylermcginnis33' },

() => console.log('setState has finished and the component has re-rendered.')

)

# **15.下述代码有错吗？**

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

return {

streak: prevState.streak + props.count

}

})

这段代码没啥问题，不过只是不太常用罢了。

1. **React组件之间的交流方式**

### **（1）**【父组件】向【子组件】传值

这个是相当容易的，在使用 React 开发的过程中经常会使用到，主要是利用 props 来进行交流。例子如下：

**// 父组件var MyContainer = React.createClass({**

**getInitialState: function () {**

**return {**

**checked: true**

**};**

**},**

**render: function() {**

**return (**

**<ToggleButton text="Toggle me" checked={this.state.checked} />**

**);**

**}**

**});**

**// 子组件var ToggleButton = React.createClass({**

**render: function () {**

**// 从【父组件】获取的值**

**var checked = this.props.checked,**

**text = this.props.text;**

**return (**

**<label>{text}: <input type="checkbox" checked={checked} /></label>**

**);**

**}**

**});**

进一步讨论

如果组件嵌套层次太深，那么从外到内组件的交流成本就变得很高，通过 props 传递值的优势就不那么明显了。（PS：所以我建议尽可能的减少组件的层次，就像写 HTML 一样，简单清晰的结构更惹人爱）

**// 父组件var MyContainer = React.createClass({**

**render: function() {**

**return (**

**<Intermediate text="where is my son?" />**

**);**

**}**

**});**

**// 子组件1：中间嵌套的组件var Intermediate = React.createClass({**

**render: function () {**

**return (**

**<Child text={this.props.text} />**

**);**

**}**

**});**

**// 子组件2：子组件1的子组件var Child = React.createClass({**

**render: function () {**

**return (**

**<span>{this.props.text}</span>**

**);**

**}**

**});**

### **（2）**【子组件】向【父组件】传值

接下来，我们介绍【子组件】控制自己的 state 然后告诉【父组件】的点击状态，然后在【父组件】中展示出来。因此，我们添加一个 change 事件来做交互。

**// 父组件var MyContainer = React.createClass({**

**getInitialState: function () {**

**return {**

**checked: false**

**};**

**},**

**onChildChanged: function (newState) {**

**this.setState({**

**checked: newState**

**});**

**},**

**render: function() {**

**var isChecked = this.state.checked ? 'yes' : 'no';**

**return (**

**<div>**

**<div>Are you checked: {isChecked}</div>**

**<ToggleButton text="Toggle me"**

**initialChecked={this.state.checked}**

**callbackParent={this.onChildChanged}**

**/>**

**</div>**

**);**

**}**

**});**

**// 子组件var ToggleButton = React.createClass({**

**getInitialState: function () {**

**return {**

**checked: this.props.initialChecked**

**};**

**},**

**onTextChange: function () {**

**var newState = !this.state.checked;**

**this.setState({**

**checked: newState**

**});**

**// 这里要注意：setState 是一个异步方法，所以需要操作缓存的当前值**

**this.props.callbackParent(newState);**

**},**

**render: function () {**

**// 从【父组件】获取的值**

**var text = this.props.text;**

**// 组件自身的状态数据**

**var checked = this.state.checked;**

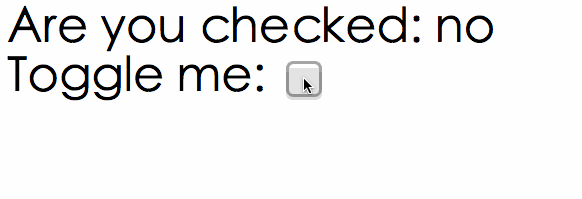
**return (**

**<label>{text}: <input type="checkbox" checked={checked} onChange={this.onTextChange} /></label>**

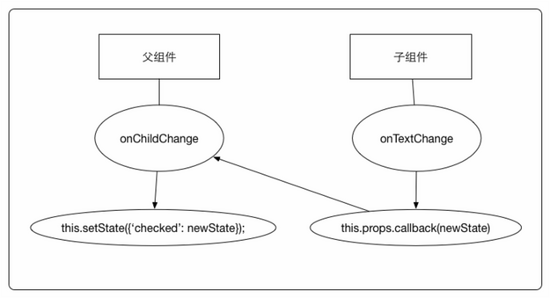
**);**

**}**

**});**



我觉得原文作者用代码不是很直观，接下来我话一个流程走向简图来直观描述一下这个过程：



这样做其实是依赖 props 来传递事件的引用，并通过回调的方式来实现的，这样实现不是特别好，但是在没有任何工具的情况下也是一种简单的实现方式

这里会出现一个我们在之前讨论的问题，就是组件有多层嵌套的情况下，你必须要一次传入回调函数给 props 来实现子组件向父组件传值或者操作。

### Tiny-Tip: React Event System

在 onChange 事件或者其他 React 事件中，你能够获取以下东西：

* 【this】：指向你的组件
* 【一个参数】：这个参数是一个 ***[React 合成事件](http://wiki.jikexueyuan.com/project/react/event-system.html" \t "http://blog.csdn.net/sinat_17775997/article/details/_blank)*** ，SyntheticEvent。

React 对所有事件的管理都是自己实现的，与我们之前使用的 onclick、onchange 事件不一样。从根本上来说，他们都是绑定到 body 上。

**document.on('change', 'input[data-reactid=".0.2"]', function () {...});**

上面这份代码不是来自于 React，只是打一个比方而已。

如果我没有猜错的话，React 真正处理一个事件的代码如下：

**var listenTo = ReactBrowserEventEmitter.listenTo;**

**...function putListener(id, registrationName, listener, transaction) {**

**...**

**var container = ReactMount.findReactContainerForID(id);**

**if (container) {**

**var doc = container.nodeType === ELEMENT\_NODE\_TYPE ? container.ownerDocument : container;**

**listenTo(registrationName, doc);**

**}**

**...**

**}// 在监听事件的内部，我们能发现如下：**

**target.addEventListener(eventType, callback, false);**

这里有所有 React 支持的事件： ***[中文文档-事件系统](http://reactjs.cn/react/docs/events.html" \t "http://blog.csdn.net/sinat_17775997/article/details/_blank)***

多个子组件使用同一个回调的情况

**// 父组件var MyContainer = React.createClass({**

**getInitialState: function () {**

**return {**

**totalChecked: 0**

**};**

**},**

**onChildChanged: function (newState) {**

**var newToral = this.state.totalChecked**

**+ (newState ? 1 : -1);**

**this.setState({**

**totalChecked: newToral**

**});**

**},**

**render: function() {**

**var totalChecked = this.state.totalChecked;**

**return (**

**<div>**

**<div>How many are checked: {totalChecked}</div>**

**<ToggleButton text="Toggle me"**

**initialChecked={this.state.checked}**

**callbackParent={this.onChildChanged}**

**/>**

**<ToggleButton text="Toggle me too"**

**initialChecked={this.state.checked}**

**callbackParent={this.onChildChanged}**

**/>**

**<ToggleButton text="And me"**

**initialChecked={this.state.checked}**

**callbackParent={this.onChildChanged}**

**/>**

**</div>**

**);**

**}**

**});**

**// 子组件var ToggleButton = React.createClass({**

**getInitialState: function () {**

**return {**

**checked: this.props.initialChecked**

**};**

**},**

**onTextChange: function () {**

**var newState = !this.state.checked;**

**this.setState({**

**checked: newState**

**});**

**// 这里要注意：setState 是一个异步方法，所以需要操作缓存的当前值**

**this.props.callbackParent(newState);**

**},**

**render: function () {**

**// 从【父组件】获取的值**

**var text = this.props.text;**

**// 组件自身的状态数据**

**var checked = this.state.checked;**

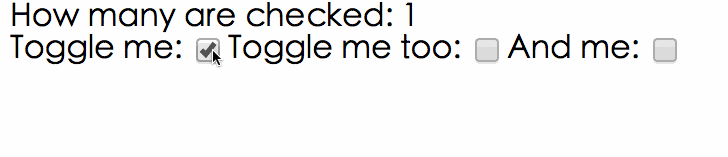
**return (**

**<label>{text}: <input type="checkbox" checked={checked} onChange={this.onTextChange} /></label>**

**);**

**}**

**});**



这是非常容易理解的，在父组件中我们增加了一个【totalChecked】来替代之前例子中的【checked】，当子组件改变的时候，使用同一个子组件的回调函数给父组件返回值。

### **（3）**没有任何嵌套关系的组件之间传值

如果组件之间没有任何关系，组件嵌套层次比较深（个人认为 2 层以上已经算深了），或者你为了一些组件能够订阅、写入一些信号，不想让组件之间插入一个组件，让两个组件处于独立的关系。对于事件系统，这里有 2 个基本操作步骤：订阅（subscribe）/监听（listen）一个事件通知，并发送（send）/触发（trigger）/发布（publish）/发送（dispatch）一个事件通知那些想要的组件。

下面讲介绍 3 种模式来处理事件，你能 ***[点击这里](https://github.com/millermedeiros/js-signals/wiki/Comparison-between-different-Observer-Pattern-implementations" \t "http://blog.csdn.net/sinat_17775997/article/details/_blank)*** 来比较一下它们。

简单总结一下：

(1) Event Emitter/Target/Dispatcher

特点：需要一个指定的订阅源

**// to subscribe**

**otherObject.addEventListener(‘click’, function() { alert(‘click!’); });// to dispatchthis.dispatchEvent(‘click’);**

(2) Publish / Subscribe

特点：触发事件的时候，你不需要指定一个特定的源，因为它是使用一个全局对象来处理事件（其实就是一个全局广播的方式来处理事件）

**// to subscribe**

**globalBroadcaster.subscribe(‘click’, function() { alert(‘click!’); });// to dispatch**

**globalBroadcaster.publish(‘click’);**

(3) Signals

特点：与Event Emitter/Target/Dispatcher相似，但是你不要使用随机的字符串作为事件触发的引用。触发事件的每一个对象都需要一个确切的名字（就是类似硬编码类的去写事件名字），并且在触发的时候，也必须要指定确切的事件。（看例子吧，很好理解）

**// to subscribe**

**otherObject.clicked.add(function() { alert(‘click’); });// to dispatchthis.clicked.dispatch();**

如果你只想简单的使用一下，并不需要其他操作，可以用简单的方式来实现：

**// 简单实现了一下 subscribe 和 dispatchvar EventEmitter = {**

**\_events: {},**

**dispatch: function (event, data) {**

**if (!this.\_events[event]) { // 没有监听事件**

**return;**

**}**

**for (var i = 0; i < this.\_events[event].length; i++) {**

**this.\_events[event][i](data);**

**}**

**},**

**subscribe: function (event, callback) {**

**// 创建一个新事件数组**

**if (!this.\_events[event]) {**

**this.\_events[event] = [];**

**}**

**this.\_events[event].push(callback);**

**}**

**};**

**otherObject.subscribe('namechanged', function(data) { alert(data.name); });this.dispatch('namechanged', { name: 'John' });**

如果你想使用 Publish/Subscribe 模型，可以使用： ***[PubSubJS](https://github.com/mroderick/PubSubJS" \t "http://blog.csdn.net/sinat_17775997/article/details/_blank)***

React 团队使用的是： ***[js-signals](http://millermedeiros.github.io/js-signals/" \t "http://blog.csdn.net/sinat_17775997/article/details/_blank)*** 它基于 Signals 模式，用起来相当不错。

### Events in React

使用 React 事件的时候，必须关注下面两个方法：

**componentDidMount**

**componentWillUnmount**

在处理事件的时候，需要注意：

在 componentDidMount 事件中，如果组件挂载（mounted）完成，再订阅事件；当组件卸载（unmounted）的时候，在 componentWillUnmount 事件中取消事件的订阅。

（如果不是很清楚可以查阅 React 对生命周期介绍的文档，里面也有描述。原文中介绍的是 componentWillMount 个人认为应该是挂载完成后订阅事件，比如Animation这个就必须挂载，并且不能动态的添加，谨慎点更好）

因为组件的渲染和销毁是由 React 来控制的，我们不知道怎么引用他们，所以EventEmitter 模式在处理组件的时候用处不大。

pub/sub 模式可以使用，你不需要知道引用。

下面来一个例子：实现有多个 product 组件，点击他们的时候，展示 product 的名字。

(我在例子中引入了之前推荐的 PubSubJS 库，如果你觉得引入代价太大，也可以手写一个简版，还是比较容易的，很好用哈，大家也可以体验，但是我还是不推荐全局广播的方式)

**// 定义一个容器var ProductList = React.createClass({**

**render: function () {**

**return (**

**<div>**

**<ProductSelection />**

**<Product name="product 1" />**

**<Product name="product 2" />**

**<Product name="product 3" />**

**</div>**

**);**

**}**

**});// 用于展示点击的产品信息容器var ProductSelection = React.createClass({**

**getInitialState: function() {**

**return {**

**selection: 'none'**

**};**

**},**

**componentDidMount: function () {**

**this.pubsub\_token = PubSub.subscribe('products', function (topic, product) {**

**this.setState({**

**selection: product**

**});**

**}.bind(this));**

**},**

**componentWillUnmount: function () {**

**PubSub.unsubscribe(this.pubsub\_token);**

**},**

**render: function () {**

**return (**

**<p>You have selected the product : {this.state.selection}</p>**

**);**

**}**

**});**

**var Product = React.createClass({**

**onclick: function () {**

**PubSub.publish('products', this.props.name);**

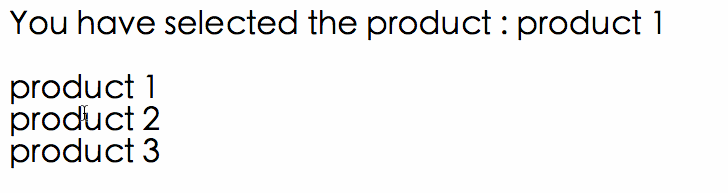
**},**

**render: function() {**

**return <div onClick={this.onclick}>{this.props.name}</div>;**

**}**

**});**



### ES6: yield and js-csp

ES6 中有一种传递信息的方式，使用生成函数(generators)和 yield 关键字。可以看一下 ***[https://github.com/ubolonton/js-csp.](https://github.com/ubolonton/js-csp." \t "http://blog.csdn.net/sinat_17775997/article/details/_blank)***

（这里我写一个简单的 DEMO 介绍一下这种新的传递方式，其实大同小异）

**function\* list() {**

**for(var i = 0; i < arguments.length; i++) {**

**yield arguments[i];**

**}**

**return "done.";**

**}var o = list(1, 2, 3);var cur = o.next;while(!cur.done) {**

**cur = o.next();**

**console.log(cur);**

**}**

通常来说，你有一个队列，对象在里面都能找到一个引用，在定义的时候锁住，当发生的时候，立即打开锁执行。**[js](http://lib.csdn.net/base/javascript" \o "JavaScript知识库" \t "http://blog.csdn.net/sinat_17775997/article/details/_blank)**-csp 是一种解决办法，也许以后还会有其他解决办法。

1. **react的Diff算法**

[react](http://lib.csdn.net/base/react" \o "React知识库" \t "http://blog.csdn.net/future_challenger/article/details/_blank)的组件在渲染之后组成了一个树形结构。在React绘制的时候，会在内存里对应每一个组件建立一个节点，并最终形成一个和组件树结构一样的树。我们就叫这个树叫影子树（这个叫法不是出自官方）。我们可以理解为这个**影子树包含了React App组建的结构和一些属性值**。

在组件发生变化的时候（一般是调用了setState），React会形成一个影子树二号。然后对比影子树1号和影子树2号的不同。

我们知道对比两个树的最小不同的时间复杂度是O(n3   
)，n是树里的节点数。这个复杂度下，稍有量级的应用都会遇到一个问题：无法忽略的慢。于是，FB的同学们使用了更加高效的启发式算法，把复杂度降低到了O(n)。

但是，不管是什么算法最后都需要对比两个节点的不同。有三种情况需要考虑：

### （1）节点之间的比较

节点，英语里的Node，包括两种类型:一个是React组件，一个是HTML的DOM。下文也是同样的含义。

#### 节点类型不同

如果是HTML DOM不同的话，直接使用新的替换旧的。

如果是组件类型不同的话也直接使用新的替换旧的。

#### HTML DOM类型相同

在React里样式并不是一个纯粹的字符串，而是一个对象，这样的话在样式发生改变的时候只需要改变替换变化以后的样式。修改完当前节点之后，递归处理该节点的子节点。

#### 组件类型相同

组件类型相同的，使用React机制处理。一般是使用新的props替换掉旧的props，并在之后调用组件的componentWill/DidReceiveProps方法，之前的组件的render方法会被调用。节点的比较机制开始递归作用于这个它的子节点上。

### （2）两个列表之间的比较

一列节点中的一个发生了改变，React并没有什么好方法来处理这个问题。循环新旧两个列表，并找出不同是React唯一的处理方法。

但是，有一个可以把这个算法的复杂度降低的办法。那就是我们在生成一列节点的时候给每一个节点上添加一个key。**这个key只需要在这一列节点中唯一，不需要全局唯一**。

### （3）取舍

需要注意的是，上面的启发式算法是基于两点假设:   
\*类型想听的节点总是生成同样的树，而类型不同的节点也总是生成不同的树。   
\*可以为多次render都表现稳定的节点设置key。

上面的节点之间的比较算法基本就是基于这两个假设而实现的。也就是**要提高React应用的效率，需要我们按照这两点假设来开发**。

否则，React会重获整个APP。那就是噩梦一样的情况了。

1. **React相较于其他框架的优点**
2. **react和angular各有什么优缺点？又分别适用什么样的场景？**

facebook喜欢独立，小巧，快速，创新。  
抛开flux,react route不谈，reactjs作为view本身非常独立，不仅可以在浏览器上用，还可以在server端结合nodejs做模板。  
除了独立，reactjs还很灵活小巧，用起来像拼lego的积木，一个component接着另一个component。  
在性能方面，由于运用了virtual dom技术，reactjs只在调用setstate的时候会更新dom，而且还是先更新virtual dom，然后和实际dom比较，最后再更新实际dom。这个过程比起angularjs, knockoutjs的bind方式来说，一是更新dom的次数少，二是更新dom的内容少，速度肯定是快了的。另外reactjs用了jsx，这个相当于半个新语言了。

google发展全面，组织结构清晰，分工明确，业务扩展虽缓慢，但是发展良好。  
angularjs是一个m-v-whateever framework。framework的一个特点就是很全面，除了m-v-whatever面面俱到，它还自带了很多$开头的service，$http, $route, $q(defer),$cookie等等，基本上只要你在做web开发用过的东西，它都有一个。  
angularjs的架构清晰，分工明确，model，view，controller谁在什么时候做什么事情说的很清楚，整个框架充满了DI的思路，耦合性非常低，对象都是被inject的，也就是说每个对象都可以轻易被替换而不影响其他对象。  
此外，angularjs的扩展性不错，但是略微有点笨拙，你可以自定义directive，当然如果嫌麻烦，只封装到controller+html很多时候也够了。

至于应用场景根据上述总结，我个人觉得企业项目，或者说业务比较正规的用angularjs比较好。对于那种倾向于比较好玩的，有趣的，有创意的项目用reactjs比较好。

**2.使用react之后有哪些经验与教训**

## **（1） 多个简单组件比一个高度定制化组件要好**

改变 React 组件行为依赖于你传递的 props，这是个很有用的功能。在项目初期就养成一个好的编程习惯对未来很有好处：创建一些通用组件，使其在其他地方也可以使用。

当你开始了项目以后，对于一个组件你可能会不断地扩展这个组件的使用外延。一段时间以后你会发现，你会疲于修复 bug，在 A 场景下修复好以后，又发现在场景 B 和场景 C 下又发现了 bug。

我的建议：如果一个组合组件导致了 bug，那么把它分解成若干个简单组件，即便代码重复也值得。

## **（2）如果你发现了库中有 bug，尽量提 Issue 和 Pull Request**

只要你使用 React，就一定会用到开源软件，不论是 React 核还是 1000 多个可用的 NPM 模块。如果你发现了库中有 bug，那就提 Issue 上去。更好的情况是，如果你 fix 了一个 bug，一定要提 Pull Request 把修复的代码整合进去，因为使用这个库并且遇到 bug 的并不只是你一个人，这样做会使整个生态变得更好。

我的建议：回馈社区，即便你只是修正了文档中的拼写错误也好。

## **（3） 首先完成一次 build 过程，然后再写 React 代码**

我知道这并不是一个常见的场景，但是我就遇到过，当我开始一个外包项目并且开始 build 的时候，发现有一些依赖编译的包不存在。进而才发现实际上这个 React 是用于一个 Backbone 项目中的。在 Backbone 中实现 React 组件其实非常简单，使用 Redux 可以在这两个之间进行通信。它们之间的通信必须要通过 Browserify 或者 Webpack 编译到一起才可以。

我的建议：如果在一个现有的项目中应用 React，首先用 Browserify 或者 Webpack 走一遍 build 过程比较好。

## **（4） 对于简单的数据可视化，原生 SVG >= D3**

D3 在数据可视化方面做的非常棒，但是如果你只是要做一些简单的数据可视化，那么渲染原生 SVG 就可以满足你的工作需求了。

我的建议：学习一些 SVG 基础，在你需要更复杂功能之前这就够了。Youtube 的 [前端中心频道](https://www.youtube.com/channel/UCbxzdZTDn4YB4Z-ukch2ivw" \t "http://web.jobbole.com/91075/_blank) 前几天刚好播放了 [关于 SVG 的节目](https://www.youtube.com/watch?v=af4ZQJ14yu8" \t "http://web.jobbole.com/91075/_blank) ，值得一看。

## **（5） 如果你只有两周的项目期限，保持功能精简**

如果你要做的工作只是渲染，那么 React 和 React-DOM 就足够了。

Redux 的处理很耗费时间和精力，它对于处理大项目中的多层 UI 很有用。但是如果你的项目很简单的话，那么通过传递 props 和 callback 就好了，不必要使用 Redux。

我的建议：模板项目是非常有用的，但是如果你想保持项目精简的话，从 React 和 React-DOM 开始是一个很好的选择。

## **（6） 单独依赖于 CSS 动画来移动元素会很慢**

我没有办法精确地告诉你什么时候会看到帧速率显著下降，也许是 30 个元素的时候，也许是 300 个元素的时候。但是可以告诉以一些对你有帮助的经验。充分利用 React 的虚拟 DOM，并且保证只在需要的时候进行渲染和重渲染。

就是说只渲染那些在视图窗口中可见的组件。

我的建议：在低配机器上进行测试，同时还要测试边界数据来看一下你的程序在受限的情况下表现如何。

## **（7） 研究模板是一种很好的开始方法，但是…**

如果你正在学习新库或者新框架，从模板项目开始是比较好的方法。如果你们公司有自己的模板就更好了。

但是不要认为模板项目可以解决所有问题。实际工作中你会发现，它所实现的根本就不是你想要的那样。如果你没有自己从头开始构建一个项目，那后面可能会出现很多问题。另外，如果你对一个模板项目的各种特征不是很了解的话，你很可能会重构一个它已经存在的功能。

我的建议：把模板项目用在它们最擅长的地方——快速启动项目。不要害怕重构它们，你要让它们按照你的意志去运作，另外，最好提前了解你所使用的模板项目的特性。

## **（8） 保持组件、connected 组件和容器的可控**

用 Redux 来 build 的时候，最好要限制组件的个数，同时要尽量保证 actions/reducers 的可复用性。

组件应该只依赖于自己的 props。

Connected 组件应该通过 Redux 来访问应用数据，并且应该只渲染自己的 DOM 和子组件。

容器充当一个演奏师的角色，取数据并且渲染组件。

我的建议：注意保持命名和功能的一致性。

## **（9） 严格的代码检查是一把双刃剑**

我开发过各种各样的项目，经历过各种形式的代码检查。从一点代码检查都没有的项目，到甚至连一个悬空逗号都会拒绝提交的项目。

我想我们大多数人都会同意代码质量不仅仅是你用对了空格符和制表符。当打开一个文件时候，代码给你那种赏心悦目的感觉会让你写代码而舒服，而且你的注意力可以专注于你的代码逻辑。

代码检查也可以帮助你发现一些错误，比如常量分配和书写错误，甚至经典的“Undefined is not a function”错误。

我的建议：拥抱你的团队所要求的代码审查规则吧。我在 JS 中使用 ESLint，在 Sass 项目 Atom 中使用 scss-lint。你也可以设置规则来方便转换，如果你要求很严格，不想让不好的代码提交上去的话， [pre-push](https://www.npmjs.com/package/pre-push" \t "http://web.jobbole.com/91075/_blank) 会执行一个 npm 脚本来做 push 前的检查。

## **（10） 在 Express 项目中进行 React 渲染是可行的**

有很多博文介绍如何安装普通应用，但是大多数都假设你是想要一个单页面应用，很少文章介绍如何在一个多页面 Express 应用中渲染单 React 组件。

我的建议：这种情况下，ReactDOMServer 会成为你的朋友。你可以把组件都当成是页面片段，把它们传递给一个模板来渲染。

## **（11） Saga 使我的大脑一团浆糊**

Saga 是 Redux 中间件，基于 ES6 的生成器新特性。“生成器定义了可以保持自己 state 的迭代算法”。在实践中它和正常的 JavaScript 工作流是不一样的，因为在另一个基于 Promise 代码执行的时候，这个函数可以在执行过程中暂停。

我应该不是第一个，也不是最后一个说这话的人，Saga 让我的大脑一团浆糊。刚学习 Saga 的时候一团乱麻，不过最终我还是征服了它。不过回头想想，如果我一开始就理解了生成器的话，可能事情发展会变的好一些。

我的建议：如果你是第一次使用 Saga，并且团队中没人有相关的经验的话，你最好还是先理解 Promise 和 Generator，会对你很有帮助。

1. **react与vue的区别**

## **React 和 Vue 之间有多大的区别？**

React 和 Vue 的相似性多于差异性：

* 都是用于创建 UI 的 JavaScript 库
* 都是快速和轻量级的
* 都有基于组件的架构
* 都使用虚拟 DOM
* 都可以放在单独的 HTML 文件中，或者在更复杂的 Webpack 设置中的一个模块
* 都有独立但常用的路由器和状态管理库

它们最大的区别在于 Vue 通常使用 HTML 模板文件，而 React 是完全使用 JavaScript。Vue 还有具有可变状态和称为 “reactivity” 的重新渲染的自动系统。

我们将在下面一一道来。

## **Components**

使用 Vue.js，组件将使用 API 方法 .component 进行声明，该方法接收 id 和定义对象的参数。你可能会注意到 Vue 组件中熟悉的方面，以及不太熟悉的方面：

JavaScript



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | Vue.component('my-component', {    *// Props*    props: [ 'myprop' ],    *// Local state*    data() {  **return** {        firstName: 'John',        lastName: 'Smith'      }    },    *// Computed property*    computed: {      fullName() {  **return** **this**.firstName + ' ' + **this**.lastName;      }    },    *// Template*    template:      <div>        <p>Vue components typically have **string** templates.</p>        <p>Here's some local state: {{ firstName }}</p>        <p>Here's a computed value: {{ fullName }}</p>        <p>Here's a prop passed down from the parent: {{ myprop }}</p>      </div>    ,      // Lifecycle hook    created() {      setTimeout(() => {        this.message = 'Goodbye World'      }, 2000);    }  }); |

## **Template**

你注意到组件有一个 template 属性，它是一个 HTML 标记的字符串。Vue 库包括一个编译器，它将在运行时将模板字符串转换为 render 函数。这些渲染由虚拟 DOM 实现。

你也可以选择不使用模板，而用自定义 render 函数。你甚至可以使用 JSX。但是切换到 Vue 只是为了做这一点好像有点“作”。

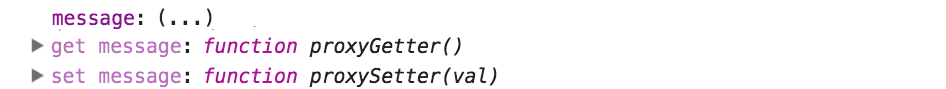
## **生命周期**

Vue 中的组件具有和 React 类似的生命周期方法。例如，当组件状态准备就绪时，但在组件已经挂载(mounted)到页面中之前，将会触发 created。

一个很大的区别：没有 shouldComponentUpdate。因为 Vue 的响应式系统（reactivity system）不需要它。

## **Re-rendering**

Vue 初始化步骤之一是遍历所有数据属性并将其转换为 getter 和 setter。如果你看下面的内容，你可以看到 message 数据属性如何添加一个 get 和 set 函数：



Vue 在访问或修改属性时添加了这些 getter 和 setter 来启用依赖关系跟踪和更改通知。

## **状态管理**

在 Vue 中改变一个组件的状态，你不需要 setState 方法，只需要变异（mutate）。

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5 | *// React*  **this**.setState({ message: 'Hello World' });    *// Vue*  **this**.message = 'Hello World'; |

当 message 的值被变异(mutation)改变时，它将触发 setter。set 方法将设置一个新值，但也将执行一个辅助任务，通知 Vue 值已更改，依赖该页面的任何部分可能需要重新渲染。

如果 message 作为一个 prop 传递给任何组件，Vue 知道它依赖于这个将自动重新渲染。这就是为什么在 Vue 中不需要 shouldComponentUpdate 的原因。

## **主模板**

关于主模板文件，Vue 更像 Angular。与 React 一样，Vue 需要挂载在页面的某个位置。

|  |  |
| --- | --- |
| 1  2  3 | <body>    <div id="root"></div>  </body> |

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | *// React*  ReactDOM.render('...', document.getElementById('root'));    *// Vue*  **new** Vue({    el: '#root'  }); |

不同于 React 的是，你可以继续添加这个主 index.html ，因为它是根组件的模板。

XHTML

|  |  |
| --- | --- |
| 1  2  3  4 | **<div** id="root"**>**  **<div>**You can add more markup to index.html**</div>**  **<my-component** v-bind:myprop="myval"**></my-component>**  **</div>** |

还可以通过使用 x-template 或 inline-template 等 HTML 功能来定义 index.html 中的子组件模板。尽管它将模板与组件定义的其余部分分开，但这不被认为是最佳实践。

## **指令(directives)**

Vue 允许你通过指令逻辑来增强你的模板，再次同 Angular 一样。这些特殊的 HTML 属性拥有 v- 前缀，例如，v-if 用于条件渲染，v-bind 用于将表达式绑定到常规 HTML 属性。

JavaScript



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **new** Vue({    el: '#app',    data: {      mybool: **true**,      myval: 'Hello World'    }  }); |

XHTML

|  |  |
| --- | --- |
| 1  2  3  4 | **<div** id="app"**>**  **<div** v-if="mybool"**>**This renders if mybool is truthy.**</div>**  **<my-component** v-bind:myprop="myval"**></my-component>**  **</div>** |

分配给一个指令的值是一个 JavaScript 表达式，所以你可以参考数据属性，包括三元运算符等。

## **工作流**

尽管社区已经建立了[create-vue-app](https://www.npmjs.com/package/create-vue-app)，但是 Vue 官方还没有一个与 create-react-app 的等效物。

官方建议使用 vue-cli 初始化项目。它可以从一个简单的项目生成一个 HTML 文件，一个完整的 Webpack + 服务端渲染项目：

JavaScript

|  |  |
| --- | --- |
| 1 | $ vue init template-**name** project-**name** |

## **单 HTML 文件项目**

Vue 作者尤雨溪将他的项目称为“渐进式框架”，因为它可以扩展到复杂的应用程序，或缩小到简单的应用程序。

当然了，React 也可以做到这一点。不同的是，Vue 项目通常使用较少的 ES6 功能，很少使用 JSX，所以通常不需要添加 Babel。此外，Vue 库全部都在一个文件中，没有相应的 ReactDOM 的单独文件。

以下是将 Vue 添加到单个 HTML 文件项目的方法：

XHTML

|  |  |
| --- | --- |
| 1 | <script src="https://unpkg.com/vue/dist/vue.js"></script> |

注意：如果你不打算使用模板字符串，因此不需要模板编译器，则会有一个较小的 Vue 构建，省略了这个称为 vue.runtime.js 的文件。大于 20KB。

## **单文件组件**

如果你乐意使用 Webpack 工具作为项目添加构建步骤，则可以使用 Vue 的单文件组件（SFC）。这些文件具有 .vue 扩展名，并将组件模板，JavaScript 配置和样式全部封装在一个文件中：

XHTML

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **<template>**  **<div** class="my-class"**>**{{ message }}**</div>**  **</template>**  <script>    export **default** {      data() {        message: 'Hello World'      }    }  </script>  <style>    .my-class { font-weight: bold; }  </style> |

这些毫无疑问是 Vue 最酷的功能之一。因为你使用 HTML 标记获得“正确”的模板，但 JavaScript 正好在那里，因此模板和逻辑之前没有尴尬的分离。

有一个名为 vue-loader 的 Webpack 加载器负责处理 SFC。在构建过程中， 模板被转换为一个渲染函数，因此这是浏览器中精简版 vue.runtime.js 的完美用例。

## **Redux and more**

Vue 还有一个名为 Vuex 的基于 Flux 的状态管理库。再次，它类似于 Redux，但有一些差异。